Mercurial による分散構成管理

Bryan O’Sullivan

本書はリビジョン a24b370a16ee の成果物を元に翻訳したものです。

目 次

図目次

Preface

分散構成管理は、比較的新しい領域であり、未開の地を切り開こうとする人々の意欲によって、発展著しいものがあります。

私が分散構成管理に関して筆を執っているのは、この分野が手引き書を書く価値のある重要なテーマであるという確信からです。執筆の題材として Mercurial を選択したのは、分散構成管理の概要を学習するのに適した容易さと、他の多くの構成管理ツールでは適用の難しい実践の場からの要望への適用性の、２つを併せ持っているためです。

0.1 This book is a work in progress

本書は、読者の役に立つことを願って、執筆途中から公開しています。その一方で、読者が本書を利用することが、一種の査読として機能することも期待しています。

0.2 About the examples in this book

本書では、コードのサンプルに関して、通例とは異なる手法を採用しています。全てのサンプルは “生きた”— シェルスクリプトにより実際に Mercurial コマンドを実行した結果を使用した—サンプルです。本書は常にソースファイルから「ビルド」され、全てのサンプルスクリプトの自動実行と、その結果と期待する結果との比較が行われます。

この手法の利点は、本書が冒頭で言及している Mercurial の版における振る舞いを**厳密に**記述していることになるため、サンプルが常に正確である点にあります。執筆対象となる Mercurial の版を変更し、その結果コマンドの出力が変化した場合、本書のビルドは失敗します。

この手法のわずかな欠点は、サンプルにおいて目にする日時情報が、同じコマンドを人手で入力した際とは異なる方法で、“押し潰され” がちな点です。複数のコマンドを毎秒入力し続けるのは人手では無理ですが、例示されている実行結果の日時情報によれば、本書のビルドに使用される自動化スクリプトは、１秒間に実に多くのコマンドを実行しています。

このため、本書のサンプルにおける連続した複数回のコミットは、まるで同一時刻に起きたことのように見えます。この現象は 9.5 節における `bisect` の例に見ることができます。

以上のことから、本書のサンプルを見る際には、コマンドの出力における日時情報に、必要以上の注意を払わないようにしてください。その代わり、サンプルにおいて目にする挙動や、その再現性に関しては、**確信**を持っていただいて構いません。

0.3 Colophon—this book is Free

本書は Open Publication License 下における利用を許可し、もっぱら Free Software ツールを使用して生成されます。組版には LaTeX、図版には Inkscape を使用しています。

本書の全ソースコードは、`http://hg.serpentine.com/mercurial/book` にある Mercurial リポジトリで公開されています。

第1章　Introduction

1.1　About revision control

構成管理とは、複数の版を持つ情報群を管理する手順のことです。最も単純な手法では、多くの人々がこれを手動で行います。ファイル更新時には、直前の版に利用した値よりも大きな値を割り当ててから、その値を含めた新しい名前でファイルを保存する、といった具合です。

しかしながら、たった１つのファイルであっても、複数の版を手動で管理する作業は間違いがちですので、この手順を自動化するソフトウェアツールには長い歴史があります。初期の構成管理を自動化するツールは、単一ユーザによる単一ファイルの版管理の補助を意図していました。ここ数十年の間に、構成管理ツールの適用範囲は大変拡大されてきました。現在では、複数のファイルに対する複数のユーザの共同作業を管理するまでになっています。今時の最善の構成管理ツールは、共同作業する数千人のユーザによって、数十万のフィルからなるプロジェクトのデータが複製されても、びくともしません。

1.1.1　Why use revision control?

プロジェクトにおいて、読者であるあなたや、あなたのチームが自動化された構成管理ツールを使用したくなるのは、以下のような理由があるからではないでしょうか。

- プロジェクトの歴史と発展を記録してくれるので、自分でそれを記録する必要が無いため。構成管理ツールを使用することで、変更毎に、**何時**、**誰が**、**何故**、**何を**変更したかの記録を見ることができます。
- 他のメンバーとの共同作業が容易になるため。例えば、潜在的に両立しない変更がほぼ同時に行われた際に、構成管理ツールはそのことを検出した上で、このような衝突の解消を手助けしてくれます。
- 間違いからの復旧を手助けしてくれるため。変更実施した後で間違いに気付いた場合、複数のファイルに渡る間違いであっても、以前の状態に復旧することができます。実のところ、**本当に**良い構成管理ツールであれば、問題が混入した時点の厳密な割り出しを効果的に探し出すことができるでしょう（詳細は、9.5 節を参照してください）[1]。
- プロジェクトの複数の版の間での同時作業や、版の間での行き来を補助してくれるため。

これらの理由の殆どが— 少なくとも理屈の上では—一人きりのプロジェクトでも、百人と共同作業するプロジェクトでも有効です。

これら２つの規模の異なるケース（“lone hacker” と “huge team”）のそれぞれにおいて、構成管理ツールの実用性に関する重要な問題は、ツールから得られる**利益**とその**コスト**をどのように比較するか、という点にあります。理解や使用が難しい構成管理ツールは、コストが高く付くでしょう。

構成管理のツールとプロセス抜きでは、500 人からなるプロジェクトはおそらく自分自身の重みで、すぐにでも崩れてしまうでしょう。この場合、構成管理ツール**抜き**には失敗が保証されたようなものですから、それを思えば、構成管理ツールを利用するコストについては考えるまでも無いでしょう。

一方で、一人での “quick hack” の場合、構成管理ツールを使うコストはプロジェクト全体のコストと同一の筈ですから、構成管理を使う余地は殆ど無いように見えるかもしれません。しかし、それは本当でしょか？

Mercurial はこれら**両方**の規模の開発を上手にサポートします。わずか数分で基本を習得でき、その低オーバヘッドのお陰で最も小さなプロジェクトにも簡単に構成管理を適用できます。

[1]訳注: つまり、それができる Mercurial は**本当に**良い構成管理ツールだ、ということですね（笑）

構成管理ツールの単純さは、難解な概念や、**本当に**やろうとしていることと心理的に競合するコマンド列といったものを、大量に身に付ける必要が無いことを意味します。同時に、Mercurial の高性能さと P2P 的特性は、大きなプロジェクトへの利用へと苦も無く拡大できます。

運営の下手なプロジェクトを救える構成管理ツールはありませんが、良いツールを選択することで、プロジェクトでの作業における滑らかさが全く違ってきます。

1.1.2 The many names of revision control

構成管理は多様な領域なので、実際には統一された名前や頭字語語がありません。
よく目にする一般的な名称および略称を以下に列挙します。

- Revision control (RCS)
- Software configuration management (SCM), or configuration management
- Source code management
- Source code control, or source control
- Version control (VCS)

これらの用語は実際にはそれぞれ異なる意味を持っている、と主張する人もいますが、実際にはお互いに非常に重複した意味を持っているので、これらに対して個別にあれこれ言うことには賛同もできませんし、有用性もありません[2]。

1.2 A short history of revision control

最も有名な昔の構成管理ツールは、Bell Labs の Marc Rochkind が 1970 年代初頭に実装した SCCS （Source Code Control System）です。SCCS は個別のファイルに対して機能し、プロジェクトに従事する全ての作業者は、単一システム上の共有作業領域へのアクセス権が必要でした。ある時点でのあるファイルの変更は、ただ一人の作業者のみが可能で、ファイルのアクセスはロックにより調停されていました。ファイルをロックしたまま開放し忘れてしまい、管理者の補助無しには他の人がファイルを変更できなくしてしまうことは、良くあることでした。

SCCS のフリーな代替ツールとして 1980 年代初頭に Walter Tichy が RCS （Revison Control System）と呼ぶプログラムを開発しました。SCCS と同様、RCS の利用には、単一の共有作業領域での作業と、複数の作業者が同時に改変するのを防ぐためのロックが必要でした。

1980 年代後期、Dick Grune は RCS を用いて、当初 cmt と呼ばれるシェルスクリプト群を実装し、後にこれらは CVS （Concurrent Versions System）と改名されました。CVS における大きな変革は、各開発者ごとの作業領域において、開発者が平行且つ幾分独立した作業ができるようになったことです。SCCS や RCS では良くあった、いつでも他人の足を踏んでしまう状況が、開発者ごとの作業領域の導入によって防がれるようになりました。各開発者は、プロジェクトに関する全てのファイルの複製を持ち、各自の複製を独立して変更することができました。中央のリポジトリへの変更のコミットに先立って、変更内容のマージをする必要がありました。

Brian Berliner は Grune のオリジナルスクリプトを元に C で書き直し、以来現代版の CVS へと発展するコードを 1989 にリリースしました。CVS はその後、「クライアント・サーバ」アーキテクチャの導入により、ネットワーク接続越しの操作を可能とする機能を獲得しました。CVS のアーキテクチャは中央集約的なもので、サーバのみがプロジェクトの履歴のこぴーを持っています。クライアント側の作業領域は、プロジェクトファイルの最新版を複製したものと、サーバの場所等を知るためのわずかなメタデータを持っているだけです。CVS は非常に成功していて、おそらく世界で最も広く使用されている構成管理システムでしょう。

Sun Microsystems は 1990 年代初頭に、TeamWare と呼ばれる分散構成管理システムのはしりとなるものを開発しました。TeamWare における（個人の）作業領域は、プロジェクトの完全な複製を格納しています。TeamWare には「中

[2]訳注：昨今のソフトウェア開発における用法を鑑みて、原文で “revision control” となっている箇所は、意図的に “構成管理”(configuration management) と訳しています。

央リポジトリ」という概念がありません（CVS は履歴格納を RCS に依存していましたが、TeamWare は SCCS を利用していました）。

1990 年代が進むにつれて、問題意識から CVS に関する問題が多く顕在化してきました。例えば CVS は、複数のファイルに対する同時更新を、論理的に不可分な単一の作用としてまとめる替わりに、ファイルごとに個別に記録しています。また、ファイル階層を上手く管理できないため、ファイルやディレクトリを改名することで、容易にリポジトリを混乱させることができます。なお悪いことに、CVS 自身のソースコードは読むにも保守するにも難解なため、アーキテクチャ上の問題点を修正する “苦痛度” は法外なものでした。

CVS の開発を行っていた Jim Blandy および Karl Fogel の二人は、より良いアーキテクチャを持ち、尚且つコードが綺麗なツールで CVS を置き換えるプロジェクトを、2001 年に始めました。結果として生み出された Subversion は、CVS の中央集約型クライアント／サーバモデルからは離れなかったものの、複数ファイルの不可分コミットや、より良い名前空間の管理、および CVS よりも概ね良好なツールと言うに足るその他の多くの機能を持っています。初回のリリース以来、その人気は速やかに上昇しています。

それと概ね同時期に、Graydon Hoare は Monotone と呼ばれる野心的な分散構成管理システムに取り掛かり始めました。Monotone は、CVS 設計上の多くの問題に取り組み、P2P アーキテクチャを持つ一方で、多くの革新的な点において初期の（そしてその後の）構成管理ツールから飛び抜けています。Monotone は、暗号で用いられるハッシュ値を識別子として使用しており、異なる由来のコードにとって不可欠な “信頼” の概念を持っています。

Mercurial は 2005 年に誕生しました。設計上の幾つかの見地において Monotone から影響を受ける一方で、Mercurial は利用の簡便性、性能の高さ、および大規模プロジェクトへの適用性に主眼を置いています。

1.3 Trends in revision control

過去４０年に渡る構成管理ツールの開発と利用における紛れも無い傾向として、構成管理ツールの利用者は、利用しているツールの機能に精通すると共に、ツールの制約によって抑制されるようです。XXXXXX There has been an unmistakable trend in the development and use of revision control tools over the past four decades, as people have become familiar with the capabilities of their tools and constrained by their limitations.

最初の世代は、単一ファイルを各自のコンピュータで管理することから始まりました。この世代のツールは、手動による場当たりな構成管理に比べれば大きな前進ではありましたが、排他による操作モデルと、単一コンピュータ上での利用を前提とした設計のため、小さく緊密なチームでの利用に限定されていました。

第二世代は、ネットワーク主体のアーキテクチャへの移行と、プロジェクト全体の一括管理によって、これらの制約を緩和しました。しかし、プロジェクト規模が大きくなればなるほど、新たな問題が発生しました。クライアントはサーバと頻繁に連携する必要があるため、サーバは大規模プロジェクトへの適用が問題になりました。信頼性の低いネットワーク接続では、遠隔ユーザがサーバと全く連携ができないこともありました。オープンソースプロジェクトが匿名の読み込み専用アクセスを開放するにつれ、リポジトリへのコミット権限を持たない人々は、構成管理ツールの通常の方法では自分たちの変更が記録できず、それ故にプロジェクトに対して働きかけることができないことに気付き始めました。

現世代の構成管理ツールは、事実上 P2P です。これらは、単一の中央サーバに対する依存を持たず、そのため構成管理データを必要な場所に分散することが可能です。インターネットを介した連携における課題は、技術的な制約に関するものから、選択 (of what ?) と合意 (of what) 形成の問題へと移行しつつあります XXXX。Collaboration over the Internet has moved from constrained by technology to a matter of choice and consensus. 最新のツールは、オフライン状況でも無制限に独立して操作でき、ネットワーク接続は他のリポジトリとの同期にのみ必要とされます。

1.4 A few of the advantages of distributed revision control

前世代への対抗馬として、ここ数年の間に分散構成管理ツールが堅牢且つ便利になってきてはいるものの、古いツールを利用している人々は、必ずしも分散構成管理ツールの長所に気付いているわけではありません。中央集約的型（ツール）と比較して、分散型（ツール）の優れている点が幾つかあります。

開発者個人にとっては、中央集約型と比較した場合、概ねいつでも分散型の方が高速です。これは、中央集約型では殆どのメタデータが中央サーバ上にしか存在しないため、多くの定型処理の度にネットワーク越しにサーバとの通信が必要、という単純な理由のためです。分散型の場合は、全てのメタデータを手元に格納しています。他の全てが

同じだとしても、ネットワーク越しの通信は中央集約型にとってのオーバヘッドとなります。構成管理ツールとの対話に多くの時間を費やそうと言うのですから、テキパキと動く応答性の良いツールの価値を軽視してはいけません。

繰り返しになりますが、分散型はメタデータを何箇所にも複製できるので、サーバ環境の気まぐれ[3]は気になりません。中央集約型でサーバが火を噴いた場合には、バックアップメディアの信頼性と、最後のバックアップが最近のものであることを祈るに違いありません。分散型の場合、各開発者のコンピュータ上に無数のバックアップが存在することになります。

分散型は中央集約型の場合よりも、ネットワークの信頼性による影響を受けません。それどころか、非常に限定的な幾つかのコマンドを除けば、中央集約型ではネットワーク接続抜きには何もできません。分散型の場合、作業中にネットワーク接続が切れても、その事に気付かないかもしれません。他のコンピュータ上のリポジトリとの連携だけはできなくなりますが、手元のリポジトリとの連携と比べれば、そのような連携が必要な事態はわずかなものです。分散しているな共同作業チームの場合には、これは重要です。

1.4.1 Advantages for open source projects

ソースをハッキングしてみようと思ったオープンソースのプロジェクトが、分散構成管理ツールを使用していた場合、自身をプロジェクトの “中核” とみなす人達と直ちに対等になれます。彼らがリポジトリを公開していれば、内部の人達と同じツール・同じ手順で、プロジェクトの履歴のコピーや、変更の実施、作業の記録といったことを、すぐにでも行うことができます。中央集約型の場合はそれとは対照的に、中央のサーバに対する変更コミットの権限を与えられない限り、“読み込み専用” モードでしか使うことができません。コミット権限が付与されるまでは変更の記録はできず、中央のリポジトリとの同期の際には常に手元での変更が破損する危険を抱えています。

The forking non-problem

分散構成管理ツールは、プロジェクトを “分裂” させ易くしてしまうため、オープンソースプロジェクトにとってある種の危険要因となる、と言われてきました。分裂は、これ以上一緒に開発を継続できないと結論付ける原因となるような、開発グループ間での意見や特性の相違のがある場合に発生します。両陣営は、プロジェクトのソースコードの概ね完全なコピーを持って、お互いの方向へと分かれてゆきます。

時には、分裂した各陣営が、お互いの相違に折り合いを付ける決定をすることがあります。中央集約型の構成管理システムでは、折り合いを付けるための**技術的な**処理が苦しく、大部分は手動で実施しなければなりません。誰の変更履歴が “生き残る” のかを決定した上で、何とかして他のチームの変更をソースツリーに移植しなければなりません。この作業は通常、他方の履歴情報の一部ないし全部を失うことになります。

分散型にとっては、分裂こそがプロジェクトを発展させる**唯一の**方法なのです。個々の変更は、全て潜在的な分裂点なのです。分裂は常に発生している全く基本的な事象なので、分散構成管理は実際に分裂を上手く**マージ**できなければならない、という点にこの考え方の強みがあります。

全ての人の全ての作業が、常に分裂とマージの観点から組み立てられた場合、オープンソース世界が “分裂” として言及するものは、**純粋に**社会的な問題となるでしょう。どちらかといえば、分散型は分裂の可能性を**低下**させています。

- 中央集約型が招いてしまう “内部”（コミット権限を持つ人々）と “外部”（持たざる人々）といった社会的区分を無くします。
- 構成管理ソフトウェアの視点では、単なるマージに過ぎませんので、社会的分裂の後の和解を容易にします。

プロジェクト全般への緊密な統治の維持が中央集約型ツールによって得られる、と信じているために、分散型に抵抗する人もいます。しかし、そういった期待の元で CVS ないし Subversion によるリポジトリを公開しても、無数に存在するツールによって、プロジェクト全体の履歴を（例え遅いとは言え）取り出し、あなたの制御の及ばない場所で再構築することができてしまいます。“プロジェクト全般への緊密な統治の維持” が錯覚である一方、So while your control in this case is illusory, you are foregoing the ability to fluidly collaborate with whatever people feel compelled to mirror and fork your history. XXXXXX

[3]訳注: 特定のサーバの動作不良等

1.4.2 Advantages for commercial projects

多くの商業プロジェクトは、世界中に散らばったチームが請け負っています。中央のサーバから遠く離れたメンバーは、コマンド実行の遅さや、おそらく殆ど信頼性の無いサーバとの接続を目にすることでしょう。商業的な構成管理システムは、遠隔サイト複製[4]の追加機能によるこれらの問題を解決しようとしていますが、通常、こういった機能は高価で保守が大変です。分散型の場合は、そもそもこういった問題で悩む必要がありません。更に、例えばサイトごとに一台ずつという塩梅で、信頼できるサーバを複数立ち上げることも簡単ですので、高価で距離のあるネットワーク経路越しのリポジトリ間で、余計な通信をする必要はありません。

中央集約型の構成管理システムは、相対的にスケーラビリティが低い傾向にあります。高価な中央集約システムだからといって、平行利用する数ダースのユーザの負荷によってダウンしてしまうことは、有り得ないことではありません。繰り返しになりますが、高負荷におけるダウンに対する典型的な対応は、高価で古臭い複製機能の利用です。分散型ツールを使用する場合、中央サーバ– 仮に持っているとしても一台だけでしょうが–における負荷は非常に低いので、もっと大人数のチームの要求を単一の安価なサーバで捌くことができますし、負荷分散は単にスクリプト作成の問題となります。

顧客の元に出て問題対応するメンバーがいる場合、分散構成管理は有益です。他のビルドからは隔離された状態で特別なビルドのために複数の修正を試したり、障害や退行の要因をソースの修正履歴から効果的に検索したりといったことを、客先環境で自社のネットワークに接続すること無しに行うことができます。

1.5 Why choose Mercurial?

Mercurial は、とりわけ構成管理システムとして良い選択をしたと言える、類を見ない特徴を持っています。

- 習得・利用が容易
- 軽量
- 規模拡大に耐え得る
- 改造が容易

構成管理システムに慣れ親しんでいるのであれば、Mercurial を使えるようになるのに５分も掛からない筈です。そうでない場合でも、更に数分以上は掛からないでしょう。Mercurial のコマンドや機能群は、全体的に統一性と一貫性が保たれていますので、沢山の例外事項ではなく、少数の一般的な方法だけを覚えておけば良いのです。

小さなプロジェクトの場合、すぐにでも Mercurial を使い始めることができるでしょう。新たな変更やブランチを生成し、変更を（同一ホストないしネットワーク越しで）持ち歩いたり、履歴参照や状態確認といった全ての操作が高速です。元来非常に高速な操作に加えて、目に見えるオーバーヘッドが少ないために、Mercurial は俊敏さを保ち、利用者の作業を妨げることを避けることができます。

Mercurial の有用性は小さなプロジェクトに限定されません。数百から数千のメンバを持ち、ソースコードが数万ファイル・数百メガバイトに及ぶプロジェクトでも採用されています。

Mercurial の基本機能に満足できない場合でも、容易に拡張することができます。Mercurial は処理のスクリプト化に適しており、Python を使って綺麗に実装されていることが、「イクステンション」という形式での機能追加を容易にしています。「障害特定の補助」から「性能向上」といった広い範囲で、評判の良い有用な多くのイクステンションが既に提供されています。

1.6 Mercurial compared with other tools

この先を読む前に、著者自身の経験／関心／（あえて言いますが）偏見といったものが、本節に反映せざるを得ない点をご理解ください。著者は、以下にあげる構成管理ツールのそれぞれを、最長で数年程度使用した経験があります。

[4]訳注：“保守が大変” と言っていることから、この場合の複製は “サーバの複製” を指しているのかな？

1.6.1 Subversion

Subversion は CVS の置き換えを目指して開発された、評判のよい構成管理ツールです。Subversion は中央集約型の「クライアント／サーバ」アーキテクチャを持っています。

Subversion と Mercurial は、同じ作用を持つ似たような名前のコマンドを持っているので、一方に馴染みのあるユーザは他方の用法を容易に習得できます。これらは両方とも全ての著名な OS 上で利用可能です。

Subversion は履歴を意識したマージ機能を持っていないので、どのリビジョンのブランチ間でマージすべきかを、ユーザ自身が厳密に指定することを強制します。この指定ができなかったり間違えたりした場合、マージにおける不必要な衝突を手動で解決する羽目になります。

著者がベンチマーク計測した限りでは、Subversion の全ての構成管理操作において、Mercurial は性能の面で相当に優位にいます。筆者の比較によると、Subversion の 1.4.3 版における *ra_local* ファイル格納（利用可能な最速のアクセス機能）と比較した場合、２倍から６倍程度の優位性がありました。ネットワーク越しのリポジトリを必要とする、より現実的な配置の場合、Subversion は相当に不利な状況になるでしょう。多くの Subversion コマンドはサーバとの連携が必要な上に、Subversion は有用な複製機能を持っていないため、少々大きめのプロジェクトの場合、サーバの性能がボトルネックとなるでしょう。

それに加えて、ファイルの更新の検索（`status`）や現行版との差分表示（`diff`）といった、幾つかの共通操作におけるネットワーク処理を回避するために、Subversion は相当な格納オーバヘッドを抱え込んでいます。Mercurial のリポジトリがプロジェクトの完全な履歴を保持しているにも関わらず、Subversion が抱え込む作業コピーは、Mercurial リポジトリと作業領域ディレクトリのサイズと、結果としておおよそ同サイズか、あるいはそれ以上になることが多いです。

構成管理関連のサードパーティツールに関しては、その差は徐々に埋まってはいるもの、Mercurial と比較して、現時点では Subversion の方がより多くのサポートを受けることができます。また、Mercurial と同様に Subversion は素晴らしいユーザマニュアルがあります。

Subversion リポジトリから Mercurial リポジトリへの、正確で完全な変更履歴の取り込みを行うツールが幾つもありますので、古いツールからの移行は比較的容易です。

1.6.2 Git

git は、Linux カーネルソースツリーを管理するために開発された分散構成管理ツールです。Mercurial と同様に、その初期の設計は Monotone から影響を受けています。

git は圧倒的なまでのコマンド群を持っており、1.5.0 版においては 139 個の独立したコマンドがあります。これらは習得が難しいとの評判です。ユーザマニュアルが存在せず、個別のコマンドに関する文書があるのみです。

性能の面では git は非常に高速です。少なくとも Linux においては、Mercurial よりも git の方が早いケースが幾つかあります。しかしながら本書の執筆時点では、Windows 環境における性能（および一般的なサポート）に関しては Mercurial に及びません。

Mercurial のリポジトリは保守の必要がありませんが、git リポジトリは手動によるメタデータの “詰め直し” を頻繁に行う必要があります。この詰め直しをしない場合、利用領域が速やかに増加する一方で、性能が低下してしまいます。厳格且つ頻繁に詰め直しをしない git リポジトリを沢山抱えるサーバは、バックアップの間、非常に disk-bound になりますし、結果として、日時バックアップ処理に２４時間以上を要するようになってしまった例が、いくつもあります。詰め替えによって鮮度が保たれている git リポジトリは、Mercurial のリポジトリよりもわずかに小さいですが、詰め替えされていない場合はかなりの大きさです。

git の基本部分は C で実装されています。多くの git コマンドはシェルないし Perl のスクリプトにより実装されていますが、その品質は非常に幅が広いです。致命的とみなすべきエラーが発生している中で闇雲に処理を続けるスクリプトを、何度か見かけたことがあります。

1.6.3 CVS

CVS はおそらく世界中で最も広く使用されている構成管理ツールです。その歴史の長さと、内部的なまとまりの無さから、長い間、本質的には保守されてきませんでした。

CVS は中央集約型の「クライアント／サーバ」アーキテクチャを持っています。CVS は関連するファイルの変更を不可分コミットへとグループ化しないため、例えば、「ある利用者による成果のコミットが、マージの必要性から

部分的にしか成功しなかった場合、他の利用者からは彼の意図した変更の一部しか見ることができない」といった、“ビルドを乱す” 行為が容易に行えてしまいます。これは、プロジェクト履歴に対する作業の進め方にも影響します。とあるタスクの一部として、あるメンバが行った変更を全て表示しようとした場合、関連する各ファイル（どのファイルがそうであるかを知っていれば、の話ですが）に対して行われた変更の、個々のコミットログと日付を手動で確認する必要があります。

CVS のタグやブランチの考え方は混乱しているため、それについて説明する気にもなれません。ファイルやディレクトリの改名がサポートされていないため、リポジトリが簡単に雑然としてしまいます。内部的な整合性をチェックする機能も持たないため、リポジトリが破損しているのか否かを判定したり、どのように破損しているのかをしることは、一般には不可能です。現存・新規のいずれのプロジェクトに対しても、CVS はお薦めできません。

Mercurial は CVS のリポジトリを取り込むことができます。しかし、いくつかの注意が必要で、これは CVS のリポジトリを取り込むことのできる、他の構成管理ツールに対しても同様です。CVS は不可分コミットを持っておらず、ファイルシステム階層の履歴管理も行っていないため、CVS から履歴を正確且つ厳密に再構築することは不可能です。幾分かの推測が必要であり、改名は通常検知できません。高度な CVS 管理の多くが手動で行われ、それ故に間違いやすいことから、CVS からの取り込みを行うツールにとって、破損したリポジトリからの取り込みは複数の問題に行き当たるのが常です（筆者の個人的経験から思い出せる、面白くも無い問題の例としては、完全に偽物のタイムスタンプや、１０年以上ロックされたままのファイルなどがあります）。

1.6.4 Commercial tools

Perforce は中央集約型の「クライアント／サーバ」アーキテクチャを持っていますが、クライアント側では全くキャッシュを行っていません。近年の構成管理ツールと異なり、編集対象となる全てのファイルに関して、Perforce はコマンド実行によるサーバへの通知をユーザに対して要求します。

Perforce の性能は小規模なチームでは非常に良好ですが、ユーザ数が数ダースを超える頃から急速に低下します。少々大規模な開発向けの Perforce インストールは、ユーザアクセスによる負荷を上手く処理するために、「プロキシ」の配置が要求されます。

第2章　A tour of Mercurial: the basics

2.1　Installing Mercurial on your system

一般的な全ての OS 向けに、ビルド済みの Mercurial バイナリ版が提供されています。バイナリ版を使用することで、簡単に Mercurial をセットアップすることができます。

2.1.1　Linux

Linux ディストリビューションは、それぞれ固有のパッケージ管理ツール、パッケージ作成方針、ならびに開発ペースを持っていますので、全てのバイナリ版 Mercurial のインストール手順に関する包括的な説明を行うのは困難です。また、バイナリ版のインストールによって利用可能な Mercurial のバージョンは、当該ディストリビューションのパッケージ保守担当者が、どの程度活発であるかによって異なります。

簡便化のため、著名な Linux ディストリビューションにおける、コマンドラインを用いた Mercurial のインストールに限定して説明します。殆どのディストリビューションでは、`mercurial` という名前のパッケージを探したならば、クリックひとつで Mercurial がインストールできるような、グラフィカルなパッケージ管理ツールが提供されています。

Debian [1]
```
apt-get install mercurial
```

Fedora Core [1]
```
yum install mercurial
```

Gentoo [1]
```
emerge mercurial
```

OpenSUSE [1]
```
yum install mercurial
```

Ubuntu Ubuntu の Mercurial パッケージは非常に古いので、使用すべきではありません。できれば、Debian パッケージをリビルドしてインストールしてください。おそらく Mercurial をソースからビルドする方が簡単でしょう。その場合の詳細は、A.1 節を参照してください。

2.1.2　Mac OS X

Mac OS X 向けの Mercurial インストーラは、Lee Cantey によって `http://mercurial.berkwood.com` で公開されています。このパッケージは、Intel および Power の両 Mac で動作します。このインストーラを使用する前に、Universal MacPython [BI] と互換性のある Python をインストールする必要があります。Lee 氏のサイトにある手順を踏めば、簡単にインストールできます。

2.1.3　Solaris

未校。XXX

2.1.4　Windows

Windows 向けの Mercurial インストーラは、Lee Cantey によって `http://mercurial.berkwood.com` で公開されています。このパッケージは他のパッケージへの依存性がありませんので、単独で利用できます。

備考: 基底状態の Windows 版 Mercurial は、Windows と Unix の改行形式の自動変換は行いません。Unix 利用者と変更成果を共有したい場合は、少々追加設定を行う必要があります。詳細未校 XXX。

2.2 Getting started

Mercurial を使い始めるにあたり、実際に利用可能な Mercurial コマンドのバージョンを確認するため、“hg version” コマンドを使ってみましょう。実際のバージョン情報にはそれほど重要性はありませんが、何も表示されない場合は対処が必要です。

```
$ hg version
Mercurial Distributed SCM (version 1.3)

Copyright (C) 2005-2009 Matt Mackall <mpm@selenic.com> and others
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

2.2.1 Built-in help

Mercurial は組み込みヘルプ機能を持っています。この機能は、コマンドの実行方法を思い出せない場合に有用です。何をして良いのか完全にわからなくなってしまった場合は、単に “hg help” を実行することで、それぞれがどのような機能を持っているかの説明が付いた、簡単なコマンド一覧が表示されます。以下に示すような形式で、特定のコマンドについて “hg help” を実行した場合、そのコマンドに関する詳細な情報が表示されます。

```
$ hg help init
hg init [-e CMD] [--remotecmd CMD] [DEST]

create a new repository in the given directory

    Initialize a new repository in the given directory. If the given
    directory does not exist, it will be created.

    If no directory is given, the current directory is used.

    It is possible to specify an ssh:// URL as the destination.
    See 'hg help urls' for more information.

options:

 -e --ssh        specify ssh command to use
    --remotecmd  specify hg command to run on the remote side

use "hg -v help init" to show global options
```

更に多くの詳細な（通常は必要としない）情報を表示するには、“hg help -v” を実行します。-v オプションは --verbose の省略形で、通常よりも多くの情報を Mercurial に表示させます。

2.3 Working with a repository

Mercurial では、全てが**リポジトリ**に閉じています。例えば、あるプロジェクトのために作成したリポジトリには、プロジェクトに “属する” 全てのファイルだけでなく、ファイルに関する履歴情報も格納されています。

リポジトリはファイルシステム上にある只のディレクトリツリーですので、Mercurial が特別扱いするということ以外には、通常のディレクトリやファイルと比較して特に変わっている点はありません。コマンド行やファイルブラウザを利用して、任意の時点で改名や削除することができます。

2.3.1 Making a local copy of a repository

リポジトリの**複製**は、少々特別です。通常のディレクトリ複製のコマンドでもリポジトリを複製できますが、Mercurial 組み込みの複製コマンドを使用した方が良いでしょ。このコマンドは、既存のリポジトリと同一の複製を生成するため、“`hg clone`” と呼ばれています。

```
$ hg clone http://hg.serpentine.com/tutorial/hello
destination directory: hello
requesting all changes
adding changesets
adding manifests
adding file changes
added 5 changesets with 5 changes to 2 files
updating working directory
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

チュートリアル用のリポジトリからの複製に成功したなら、ローカルファイルシステム上に `hello` という名前のディレクトリがある筈です。このディレクトリにはファイルが幾つか格納されていることでしょう。This directory will contain some files.

```
$ ls -l
total 4
drwxr-xr-x 3 vmuser vmuser 4096 Jul 20 21:59 hello
$ ls hello
Makefile  hello.c
```

これらのファイルは、複製元になったリポジトリにおけるファイルと、全く同じ内容と履歴情報を持っています。

全ての Mercurial リポジトリは、機能提供に必要なものを全て格納しているため、それ自体で完結している、独立した存在です。リポジトリには、プロジェクトに属するファイルの私的な複製と履歴情報が格納されます。複製されたリポジトリは、複製元となったリポジトリの位置を記憶していますが、特に明示的な指示をしない限り、複製元リポジトリとの連携（および、それ以外のリポジトリとの連携も）は行われません。

それぞれのリポジトリは、他のリポジトリに影響を及ぼすことの無い、私的な “箱庭” と言えますから、自身のリポジトリで自由に実験ができるわけです。

2.3.2 What’s in a repository?

リポジトリ内部を仔細に見てみると、`.hg` という名前のディレクトリがあることに気が付くことでしょう。このディレクトリは、Mercurial がリポジトリのメタデータを格納しているディレクトリです。

```
$ cd hello
$ ls -a
.  ..  .hg  Makefile  hello.c
```

`.hg` およびその配下のディレクトリの内容は、Mercurial が私的に使用するものです。リポジトリにおけるそれ以外のディレクトリ・ファイルは、自由に利用して構いません。

用語の定義をするにあたり、`.hg` ディレクトリを “本当の” リポジトリとするなら、それと共存する他のファイル・ディレクトリは**作業領域ディレクトリ**にあるもの、と呼ばれます。両者の区分を簡単に言うなら、**リポジトリ**がプロジェクトの**履歴**を保持する一方で、**作業領域ディレクトリ**は、履歴上のとある時点におけるプロジェクトの**スナップショット**を保持する、と言えます。

2.4 A tour through history

馴染みの無い新しいリポジトリに対しては、まずはその履歴を参照してみようと思うことでしょう。“`hg log`” コマンドは、履歴情報を出力します。

```
$ hg log
changeset:   4:2278160e78d4
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:16:53 2008 +0200
summary:     Trim comments.

changeset:   3:0272e0d5a517
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:08:02 2008 +0200
summary:     Get make to generate the final binary from a .o file.

changeset:   2:fef857204a0c
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:05:04 2008 +0200
summary:     Introduce a typo into hello.c.

changeset:   1:82e55d328c8c
user:        mpm@selenic.com
date:        Fri Aug 26 01:21:28 2005 -0700
summary:     Create a makefile

changeset:   0:0a04b987be5a
user:        mpm@selenic.com
date:        Fri Aug 26 01:20:50 2005 -0700
summary:     Create a standard "hello, world" program

```

このコマンドの基底動作では、プロジェクトに加えられた個々の変更の記録に対して簡単な出力を行います。Mercurial の用語では、複数のファイルに対する変更を保持し得ることから、記録されたこれらの出来事を**チェンジセット**と呼称します。

“`hg log`” の出力形式における各欄は、以下のようになっています。

`changeset` この欄は、10 進数、コロン（colon: :）および 16 進数の連続形式となっています。２つの数値は共にチェンジセットの**識別子**です。16 進数のものよりも、10 進数の方が短く、入力が容易であることから、２つの識別氏が存在します。

`user` チェンジセットの作成者に関する識別情報です。この欄は自由形式ですが、殆どの場合、人名と電子メールアドレスが格納されます。

`date` チェンジセットが作成された日時と、そのタイムゾーンです（日時は当該タイムゾーンにおける値ですので、チェンジセットの作成者にとっての日時を表します）。

`summary` チェンジセット作成者が、作成の際にチェンジセットの説明として入力したメッセージの最初の行です。

基底動作における “hg log” の出力は、単純な要約ですので、多くの詳細データが欠けています。
図 2.4 は、履歴の “動向” を把握し易くするために、hello リポジトリにおける履歴を図示したものです。本章および以降の章において、何度かこの図に立ち返ることになることでしょう。

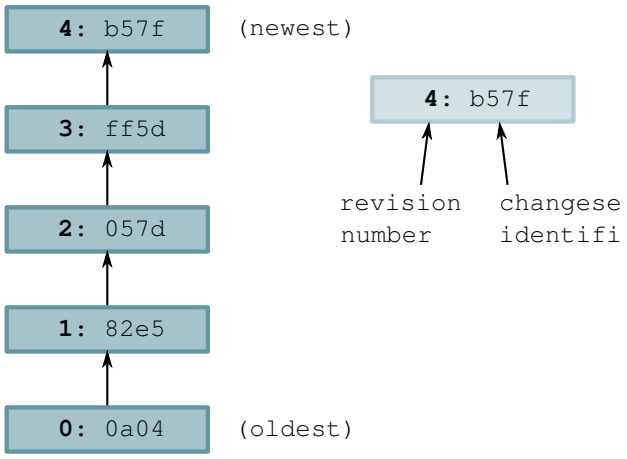

図 2.1: Graphical history of the hello repository

2.4.1 Changesets, revisions, and talking to other people

英語が不正確さで悪名高い言語であり、計算機科学では用語の混乱はいつものことですので、構成管理の分野では、同じことを表す複数の用語や言い回しが存在します。Mercurial での履歴管理について話をする場合、“チェンジセット”（changeset）という用語が時には “チェンジ”（change）や（文書の場合は）“cset” などと省略されていたり、チェンジセットという言い回しが、“リビジョン”（revision）ないし “rev” を表すものとして使用されたりするのを目にするかもしれません。
“チェンジセット” の概念をどのような**用語**で表そうが問題ではありませんが、“**特定の**チェンジセット” を指すための**識別子**は非常に重要です。“hg log” の出力における changeset 欄が、10 進数と 16 進数の両方の識別子を使ってチェンジセットを識別している、ということを思い出してください。

- 10 進数の識別子（= リビジョン番号）が、**当該リポジトリでのみ有効な値**である一方で、
- 16 進数の識別子は、**全ての**複製リポジトリに渡って、厳密にチェンジセットを識別可能な**恒久普遍の識別子**です。

この区別は重要です。電子メールで他の人と “リビジョン 33” の話をした場合、相手のリビジョン 33 は、自分の意図するそれとは高い確率で**異なります**。これは、リビジョン番号の割り付けが、当該チェンジセットがリポジトリに認識された順序に依存しており、チェンジセットの認識順序が同一であることを、異なるリポジトリの間では保障できないためです。３つのチェンジセット a,b,c が、とあるリポジトリでは 0,1,2 の順序で認識される一方で、別なリポジトリでは 1,0,2 の順序で認識される、といったことは容易に起こり得ます。
Mercurial がリビジョン番号を使用しているのは、純粋に記述簡略化の利便性のためです。他の人とチェンジセットに関して話をする場合や、何らかの理由（例えば、障害報告における記録）によってチェンジセットに関する記録を残す場合は、16 進数の識別子を使いましょう。

2.4.2 Viewing specific revisions

“hg log” の出力を単一のリビジョンのものに限定する場合、-r（ないし --rev）オプションを使用します。10 進数のリビジョン番号と、16 進数のチェンジセット識別子のどちらも使用できますし、必要に応じて複数のリビジョンを指定することもできます。

```
$ hg log -r 3
changeset:   3:0272e0d5a517
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:08:02 2008 +0200
summary:     Get make to generate the final binary from a .o file.

$ hg log -r ff5d7b70a2a9
abort: unknown revision 'ff5d7b70a2a9'!
$ hg log -r 1 -r 4
changeset:   1:82e55d328c8c
user:        mpm@selenic.com
date:        Fri Aug 26 01:21:28 2005 -0700
summary:     Create a makefile

changeset:   4:2278160e78d4
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:16:53 2008 +0200
summary:     Trim comments.

```

個別に列挙すること無しに複数のリビジョンの履歴を参照したい場合は、**範囲記法**を使用します。この記法は、“*a* から *b* の間の全てのリビジョン” という意図を表現します。

```
$ hg log -r 2:4
changeset:   2:fef857204a0c
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:05:04 2008 +0200
summary:     Introduce a typo into hello.c.

changeset:   3:0272e0d5a517
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:08:02 2008 +0200
summary:     Get make to generate the final binary from a .o file.

changeset:   4:2278160e78d4
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:16:53 2008 +0200
summary:     Trim comments.

```

Mercurial はりビジョンの記述順序に忠実に振舞いますので、“`hg log -r 2:4`” というコマンド起動が 2,3,4 の順序で表示する一方、“`hg log -r 4:2`” というコマンド起動は 4,3,2 の順序で表示します。

2.4.3 More detailed information

目当てのチェンジセットが既に判明している場合は “`hg log`” が出力する概要情報は有用ですが、あるチェンジセットが目当てのものか否かを判定しようとする場合には、変更についての完全な説明文や、変更されたファイルの一覧が必要になることでしょう。“`hg log`” コマンドの`-v`（ないし `--verbose`）オプションは、これら追加の詳細情報を表示します。

```
$ hg log -v -r 3
changeset:   3:0272e0d5a517
user:        Bryan O'Sullivan <bos@serpentine.com>
```

```
date:        Sat Aug 16 22:08:02 2008 +0200
files:       Makefile
description:
Get make to generate the final binary from a .o file.


```

説明文と変更内容の両方を見たい場合は、`-p`（ないし `--patch`）オプションを付加してください。このオプションにより、変更内容が *unified diff* 形式（これまでに unified diff 形式を見たことが無いのでしたら、12.4 節に概要の説明があります）で出力されます。

```
$ hg log -v -p -r 2
changeset:   2:fef857204a0c
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:05:04 2008 +0200
files:       hello.c
description:
Introduce a typo into hello.c.


diff -r 82e55d328c8c -r fef857204a0c hello.c
--- a/hello.c        Fri Aug 26 01:21:28 2005 -0700
+++ b/hello.c        Sat Aug 16 22:05:04 2008 +0200
@@ -11,6 +11,6 @@

 int main(int argc, char **argv)
 {
-        printf("hello, world!\n");
+        printf("hello, world!\");
         return 0;
 }

```

2.5 All about command options

Mercurial のコマンド探検をここで少々中断して、Mercurial コマンドの動作パターンについて説明しましょう。本章におけるツアーを続けるにつれて、このことを覚えておいて良かったと思うことでしょう。

Mercurial は、コマンドに対して指定可能なオプションの取り扱いに関して、近年の Linux および Unix システムに共通のオプション記述慣習を踏襲した、一貫した素直な扱い方を採用しています。

- 全てのオプションはロングネーム（long name）を持っています。例えば、既に見てきたように、“`hg log`” コマンドは `--rev` オプションを受け付けます。
- 殆どのオプションがショートネーム（short name）も持っています。`--rev` オプションの代わりに`-r` を使用できます（ショートネームを持たないオプションがあるのは、それらのオプションが滅多に利用されないためです[1]）。
- ロングネームオプションは２つのマイナス記号[2]で始まります（例: `--rev`）がショートネームオプションは１つのマイナス記号で始まります（例: `-r`）。

[1]訳注: 訳者のコマンド開発経験では、ショートネームの候補となるアルファベットが複数のオプションの間で重なる場合、あえてショートネームを設定しない、という場合もあります。

[2]訳注: 原文では “dash(es)” ですが、「ダッシュ（ダーシ）」や「ハイフン」よりも、PC における入力では直接的な、「マイナス記号」を訳語に当てました。

- オプションの命名と用法は、コマンド間で一貫性が取られています[3]。例えば、チェンジセット識別子やリビジョン番号を指定可能なコマンドは、全て `-r` および `--rev` オプションを受理します。

本書の実行例では、ロングネームオプションの代わりにショートネームオプションを使用します。これは単に筆者の好みというだけのことですので、特に気にする必要はありません。

何らかの表示を行うコマンドの多くは、`-v`（ないし `--verbose`）オプションを付与することでより多くの情報の表示を、`-q`（ないし `--quiet`）オプションを付与することで表示を抑止することができます。

2.6 Making and reviewing changes

この時点で、Mercurial における履歴を把握できていますので、変更の実施や、その検証を行ってみましょう。

まず始めにすべきことは、独自の実験を元々のリポジトリから隔離することです。リポジトリの複製に、先程は “`hg clone`” を使用しましたが、この時点での遠隔リポジトリからの複製は必要ありません。既に手元にある複製リポジトリから複製すれば良いのです。ローカルリポジトリの複製は、ネットワーク越しの複製よりも非常に高速ですし、多くの場合においてディスク領域消費も少なくて済みます[4]。

```
$ cd ..
$ hg clone hello my-hello
updating working directory
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd my-hello
```

話は逸れますが、作業に着手しようとした際に、作業用サンドボックスとしての一時的な複製を何時でも作成できますので、遠隔リポジトリの複製を “まっさらな” 状態で保つように心掛けるのが良いでしょ。こうすることで、複数の作業を平行に行うことができますし、作業完了後にそれらを統合するまでは、互いの作業を隔離された状態にすることができます。ローカルリポジトリの複製は低コストですから、リポジトリの複製および破棄にはオーバヘッドが殆どありません。

`my-hello` リポジトリには、典型的な “hello, world” プログラムが格納された `hello.c` ファイルがあります。ではここで、いにしえの `sed` コマンドを使用して、２行目を出力するように変更してみましょう。（変更のために `sed` を使用するのは、単にスクリプトによる自動化が簡単であるからです。自動化の必要が無ければ、おそらく `sed` を使用する必要は無いでしょう。好みのエディタで編集をしてください。）。

```
$ sed -i '/printf/a\\tprintf("hello again!\\n");' hello.c
```

“`hg status`” コマンドにより、リポジトリ配下のファイルの状況に関する Mercurial の認識が表示されます。

```
$ ls
Makefile  hello.c
$ hg status
M hello.c
```

幾つかのファイルに対しては、“`hg status`” コマンドは特に何も表示しませんが、`hello.c` に対しては “`M`” で始まる行を表示します。明示的に指定しない限り、変更されていないファイルに対して “`hg status`” は何も表示しません。

“`M`” 表示は、Mercurial が `hello.c` ファイルの変更を検知していることを表します。ファイルの変更に先立って（あるいは変更の後に）、Mercurial に対して**通知**する必要はありません。Mercurial 自身で変更の実施を検知することができます。

[3] 訳注： 訳者が以前、オプションを追加するパッチを提案した際には、パッチの機能的な話とは別に、「○○のコマンドでは××というう命名になっているから、それに倣ってね」と指摘されたことがあり、「一貫性がとられている」との主張は伊達ではありません。

[4] 訳注: 詳細は “Avoiding seeks” にありますが、Mercurial はローカルリポジトリの複製の際に、ディスクヘッドのシーク回避のために、ファイルの複製ではなく所謂 “ハードリンク” を実施します。

“hg status” の表示は、hello.c を変更したことを知るのに役立ちますが、**どのような**変更を行ったのかを厳密に知りたい場合も有るでしょう。変更内容を知るためには、“hg diff” コマンドを使用します。

```
$ hg diff
diff -r 2278160e78d4 hello.c
--- a/hello.c        Sat Aug 16 22:16:53 2008 +0200
+++ b/hello.c        Mon Jul 20 21:59:02 2009 +0000
@@ -8,5 +8,6 @@
 int main(int argc, char **argv)
 {
         printf("hello, world!\");
+        printf("hello again!\n");
         return 0;
 }
```

2.7 Recording changes in a new changeset

変更内容に満足して、新規チェンジセットに変更内容を記録するに足る状況に到達するまでは、ファイルの内容を変更し、ビルドと変更内容に対する試験を行い、“hg status” および “hg diff” による変更内容を確認する、という作業を繰り返します。

“hg commit” コマンドを用いることで、チェンジセットを新たに作成することができます。通常これを “コミットの実施”（“making a commit”）ないし “コミットする”（“committing”）と言います。

2.7.1 Setting up a username

最初に “hg commit” 実行を行う際には、必ずしも実行が成功[5]するとは限りません。チェンジセットのコミットの際に Mercurial は、コミットしたユーザの名前と電子メールアドレスを、チェンジセット毎に記録しますので、誰もが後からチェンジセット作成者を知ることができます。Mercurial は以下の手順で、変更内容と共に記録する妥当なユーザ名を自動的に検出しようとします。

1. “hg commit” コマンド起動の際に-u オプションによってユーザ名を指定した場合、常にその値が優先的に使用されます。
2. 次に HGUSER 環境変数設定の有無が確認されます。
3. ホームディレクトリ直下に、username 要素を持つ.hgrc[6]がある場合、その値が使用されます。このファイルに書くべき内容に関しては、2.7.1 節を参照してください。
4. EMAIL 環境変数が設定されている場合は、その値が使用されます。
5. それ以外の場合、Mercurial は稼動しているシステムにユーザとホストの名前を問い合わせた上で、電子メールアドレス形式のユーザ名を生成し、これを使用します。この方法で生成されたユーザ名は往々にして役に立たないため、Mercurial は警告を表示します。

上記の方法が全て失敗した場合、Mercurial によるコミットは失敗し、エラーメッセージを表示します。そのような場合では、明示的にユーザ名を指定しない限り、コミットは成功しないでしょう。

HGUSER 環境変数と “hg commit” コマンドへの-u オプション指定は、Mercurial 設定ファイル中の username 設定を**無効にする**点に注意してください。通常の使用において、自身のユーザ名を簡単且つ確実に指定するには、.hgrc ファイルで指定するのが良いでしょ。記述方法に関する詳細は後述します。

[5]訳注: ここで言う “成功” とは、コマンド実行そのものの成功というよりは、“思った通りのチェンジセットを生成” することに対する成功に近いニュアンスです。

[6]訳注: Windows 向けバイナリ版の場合、HOME 環境変数が指すディレクトリ、ないし C:
Documents and Settings
USERNAME 配下の Mercurial.ini が用いられます。

Creating a Mercurial configuration file

ユーザ名を設定するには、まずは好みのエディタを使って、ホームディレクトリ直下に.hgrc ファイルを作成します。Mercurial はこのファイルから利用者の個人設定を参照します。.hgrc の内容は、まずは以下のようになるでしょう。

```
# This is a Mercurial configuration file.
[ui]
username = Firstname Lastname <email.address@domain.net>
```

“[ui]” 行は、設定ファイルの**セクション**開始を意味し、“username = ...” という記述行は “ui セクションにおける username 項目への値の設定” とみなされます。一度セクションが開始されたなら、新たなセクションが開始されるか、ファイルの末尾に達するまで当該セクションが続きます。空の行と、“#” の次の文字から行末までは、Mercurial によってコメントとみなされ無視されます。

Choosing a user name

username 設定項目は、Mercurial に与える値ではありますが、リポジトリを参照する他の利用者のための情報ですので、任意の文字を使用可能です。殆どの利用者は、名前と電子メールアドレスを用いた前述のような形式を用いています。

> **備考:** Mercurial の組み込みウェブサーバ機能では、スパムメールの送付者が利用する電子メールアドレス自動収集ツールに対して、電子メールアドレスを難読化することが可能です。この機能を用いることで、Mercurial リポジトリをウェブ上に公開した際に、益体も無いメール受信の増加を抑止することができます。

2.7.2 Writing a commit message

当該チェンジセットでの変更内容を説明するメッセージを入力するために、Mercurial はコミットの際にエディタを起動します。このメッセージを**コミットメッセージ**と呼び、読み手に変更の内容と理由を伝えるために記録されるもので、コミット後の “hg log” コマンドにより表示されます。

```
$ hg commit
```

“hg commit” コマンドが起動するエディタは、“HG:” で始まる数行が後に続く空行を表示していることでしょう。

```
空行
HG: changed hello.c
```

Mercurial は “HG:” で始まる行を無視します。これらの行は、チェンジセットへの変更記録対象となるファイルの一覧を、コミットしようとしているユーザに知らせるためだけのものです。そのため、これらの行の変更や削除は何も意味を持ちません。

2.7.3 Writing a good commit message

“hg log” はコミットメッセージの最初の 1 行しか表示しませんので、最初の 1 行だけで意味の通じる内容にするのが良いでしょう。この方針から**外れている**ために、読み難いコミットメッセージの実例を以下に示します。

```
changeset:   73:584af0e231be
user:        Censored Person <censored.person@example.org>
date:        Tue Sep 26 21:37:07 2006 -0700
summary:     include buildmeister/commondefs.   Add an exports and install
```

コミットメッセージの２行目以降に関しては、特に厳密なルールは存在しません。コミットメッセージに対して、プロジェクト運用上の方針として何らかの形式を要求するかもしれませんが、Mercurial 自身が解釈や忖度をすることはありません。

筆者の個人的な好みは、“hg log --patch” を一瞥しただけでは判断できない事柄について、簡潔でありながら有益な情報をもたらすようなコミットメッセージです。

2.7.4 Aborting a commit

コミットメッセージの記述中にコミットを取りやめを決意した場合には、編集中のファイルを保存せずにエディタを終了すれば良いのです。この場合、リポジトリと作業領域ディレクトリのいずれに対しても、何ら操作は加えられません。

引数無しで “hg commit” コマンドを実行した場合、“hg status” および “hg diff” によって報告された全ての変更内容が記録されます。

2.7.5 Admiring our new handiwork

コミットが完了したなら、今しがた新規作成したチェンジセットを “hg tip” コマンドで表示することができます。このコマンドは “hg log” と同一の出力を行いますが、表示されるのはリポジトリにおける最新のリビジョンだけです。

```
$ hg tip -vp
changeset:   5:30682a1ee732
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:02 2009 +0000
files:       hello.c
description:
Added an extra line of output


diff -r 2278160e78d4 -r 30682a1ee732 hello.c
--- a/hello.c        Sat Aug 16 22:16:53 2008 +0200
+++ b/hello.c        Mon Jul 20 21:59:02 2009 +0000
@@ -8,5 +8,6 @@
 int main(int argc, char **argv)
 {
         printf("hello, world!\");
+        printf("hello again!\n");
         return 0;
 }

```

リポジトリにおける最新のリビジョンを tip リビジョン、あるいは単に tip と呼びます。

2.8 Sharing changes

先の記述で、Mercurial におけるリポジトリは、それ自身で完結している旨述べました。これは即ち、たった今新規に作成したチェンジセットは、手元の my-hello リポジトリにしか存在しないことを意味します。この変更内容を他のリポジトリへと伝播する方法を、順に見てゆきましょう。

2.8.1 Pulling changes from another repository

まず始めに、元々の hello リポジトリを複製して、たった今新規に作成した変更のコミットされていないリポジトリを作成しましょう。この複製したリポジトリを、hello-pull と呼びます。

```
$ cd ..
$ hg clone hello hello-pull
updating working directory
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

“hg pull” コマンドにより、my-hello から hello-pull へと変更を取り込みます。しかしながら、未知の変更を闇雲にリポジトリに取り込むのは、あまりぞっとしません。Mercurial が提供する “hg incoming” コマンドは、実際に変更を取り込む事無く、“hg pull” によってリポジトリに取り込まれる**予定**のチェンジセットを表示します。

```
$ cd hello-pull
$ hg incoming ../my-hello
comparing with ../my-hello
searching for changes
changeset:   5:30682a1ee732
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:02 2009 +0000
summary:     Added an extra line of output

```

（勿論、“hg incoming” コマンドを実行したリポジトリに対して、“hg pull” による変更取り込みの機会よりも前に、より多くの変更を追加することは可能ですので、実際の変更取り込みは予定とは異なる可能性が有ります。）

リポジトリへの変更の取り込みは、どのリポジトリから取り込むかを指示しつつ、“hg pull” コマンドを実行するという簡単なものです。

```
$ hg tip
changeset:   4:2278160e78d4
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:16:53 2008 +0200
summary:     Trim comments.

$ hg pull ../my-hello
pulling from ../my-hello
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
(run 'hg update' to get a working copy)
$ hg tip
changeset:   5:30682a1ee732
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:02 2009 +0000
summary:     Added an extra line of output

```

実施前後の “hg tip” 出力から見て取れるように、手元のリポジトリへの変更内容の反映が成功しています。取り込んだ変更内容を作業領域ディレクトリにおいて参照するためには、もうひと手順必要です。

2.8.2 Updating the working directory

リポジトリと作業領域ディレクトリの関係について、これまでは大雑把にしか説明してきませんでした。2.8.1 節で実行した “hg pull” コマンドは、リポジトリへの変更の取り込みを行いますが、確認してみればわかるように、作業

領域には何ら影響を及ぼしません。これは、“hg pull” の（基底の）挙動が、作業領域に影響を及ぼさないものであるためです。作業領域の更新には、“hg pull” ではなく “hg update” を用います。

```
$ grep printf hello.c
        printf("hello, world!\");
$ hg update tip
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ grep printf hello.c
        printf("hello, world!\");
        printf("hello again!\n");
```

“hg pull” 実行時に作業領域を自動的に更新しないことは、一見奇異に見えるかもしれませんが、実はそれには理由が有ります。“hg update” を用いることで、リポジトリに記録された**任意の版**の状態へと、作業領域ディレクトリの内容を更新することができます。作業領域ディレクトリを—例えば、バグの原因調査などのために— 古い版にして作業していた場合などは、“hg pull” 実行が作業領域ディレクトリを最新の版に自動的に更新してしまうのは、あまりよろしくないでしょう。

しかし、“hg pull” 〜 “hg update” という流れは非常に頻繁な作業ですから、“hg pull” に -u オプションを指定することで、Mercurial はこれら 2 つを組み合わせた機能を提供します。

```
hg pull -u
```

2.8.1 節での-u オプションを指定しない “hg pull” 実行の出力には、作業領域ディレクトリの更新に明示的な手順が必要であることを示す、注意喚起のメッセージが表示されているのが見て取れます。

```
(run 'hg update' to get a working copy)
```

作業領域ディレクトリがどの版の内容に基づいているかを見るには、“hg parents” コマンドを使用します。

```
$ hg parents
changeset:   5:30682a1ee732
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:02 2009 +0000
summary:     Added an extra line of output

```

図 2.4 では、個々のチェンジセットを繋ぐ矢印が描かれています。矢印の**根元**にあたるチェンジセットが親を、そして矢印の**先**にあたるチェンジセットが子を表しています。同じように、作業領域ディレクトリも親を持っており、現時点で保持している作業領域ディレクトリの内容は、そのチェンジセットに基づいたものです。

作業領域ディレクトリの内容を特定の版のものにする場合、“hg update” コマンドにリビジョン番号ないしチェンジセット ID を指定します。

```
$ hg update 2
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg parents
changeset:   2:fef857204a0c
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Sat Aug 16 22:05:04 2008 +0200
summary:     Introduce a typo into hello.c.

$ hg update
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

明示的な版指定をしなかった場合、上記の例における 2 つ目の “hg update” 実行で見て取れるように、“hg update” は tip が指定されたものとして振舞います。

2.8.3 Pushing changes to another repository

Mercurial では、現在作業を行っているリポジトリから他のリポジトリへの、変更内容の反映が可能です。先に示した “hg pull” の例と同様に、まずは変更反映先とするための一時的なリポジトリを作成します。

```
$ cd ..
$ hg clone hello hello-push
updating working directory
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

“hg outgoing” コマンドは、他のリポジトリへの反映対象となるチェンジセットを一覧表示します。

```
$ cd my-hello
$ hg outgoing ../hello-push
comparing with ../hello-push
searching for changes
changeset:   5:30682a1ee732
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:02 2009 +0000
summary:     Added an extra line of output

```

そして “hg push” コマンドが実際の反映作業を行います。

```
$ hg push ../hello-push
pushing to ../hello-push
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

“hg pull” と同様に、“hg push” コマンドは変更反映先のリポジトリ側において、作業領域ディレクトリの更新は行いません（“hg pull” と違い、“hg push” は変更反映先のリポジトリ側での作業領域ディレクトリを更新する-u オプションを持ちません）。

当該リポジトリが既に相当するチェンジセットを持っている場合、変更の取り込みあるいは反映を行うとどうなるのでしょう？驚くようなことは何も起こりません。

```
$ hg push ../hello-push
pushing to ../hello-push
searching for changes
no changes found
```

2.8.4 Sharing changes over a network

先の幾つかの節で触れたコマンドの利用は、手元にあるリポジトリにのみ限定されているわけではありません。全く同様の形式で、ネットワーク接続経由でも機能します。ローカルファイルシステムのパスの代わりに、URL を指定すれば良いのです。

```
$ hg outgoing http://hg.serpentine.com/tutorial/hello
comparing with http://hg.serpentine.com/tutorial/hello
searching for changes
changeset:   5:30682a1ee732
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:02 2009 +0000
summary:     Added an extra line of output

```

この例では、遠隔リポジトリに対して反映可能な変更の一覧を見ることができますが、このリポジトリは匿名での変更反映を許すようには当然ですが設定されていません。

```
$ hg push http://hg.serpentine.com/tutorial/hello
pushing to http://hg.serpentine.com/tutorial/hello
searching for changes
ssl required
```

第3章　A tour of Mercurial: merging work

前章においては、リポジトリの複製、リポジトリでのチェンジセットの生成、ならびに “hg push” および “hg pull” によるリポジトリ間でのチェンジセットの授受を見てきました。次の段階として、別々のリポジトリにおける変更の**マージ**（merge）について見てみましょう。

3.1　Merging streams of work

分散構成管理ツールにおいて、マージは作業の基本です。

- Alice と Bob が、共同作業しているプロジェクトのリポジトリから複製した、個人的なリポジトリを持っているものとします。Alice は自分のリポジトリにおいてバグを修正しました。Bob は自分のリポジトリにおいて機能を追加しました。二人は、バグフィックスと新機能の両方を含むリポジトリを共有したいと思うでしょう。
- 筆者は、個別のリポジトリによって、お互いが安全に隔離された複数の異なる作業を、同一プロジェクトにおいて同時に実施することが頻繁にあります。この形式での作業では、あるリポジトリにおける成果を、他のリポジトリに対して頻繁にマージする必要があります。

マージは必要に応じて実施するありふれた作業ですので、Mercurial では簡単に行えるようになっています。それでは、マージ手順を見て行きましょう。もう一度リポジトリの複製を行い（もう何度も複製しましたよね？）、そのリポジトリにおいて変更を行います。

```
$ cd ..
$ hg clone hello my-new-hello
updating working directory
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd my-new-hello
$ sed -i '/printf/i\\tprintf("once more, hello.\\n");' hello.c
$ hg commit -m 'A new hello for a new day.'
```

この時点で、内容の異なる２つの hello.c のコピーが存在するはずです。２つのリポジトリの履歴は、図 3.1 に示すように、枝分かれしています。

```
$ cat hello.c
/*
 * Placed in the public domain by Bryan O'Sullivan.  This program is
 * not covered by patents in the United States or other countries.
 */

#include <stdio.h>

int main(int argc, char **argv)
{
        printf("once more, hello.\n");
        printf("hello, world!\");
        return 0;
}
```

```
$ cat ../my-hello/hello.c
/*
 * Placed in the public domain by Bryan O'Sullivan.  This program is
 * not covered by patents in the United States or other countries.
 */

#include <stdio.h>

int main(int argc, char **argv)
{
        printf("hello, world!\");
        printf("hello again!\n");
        return 0;
}
```

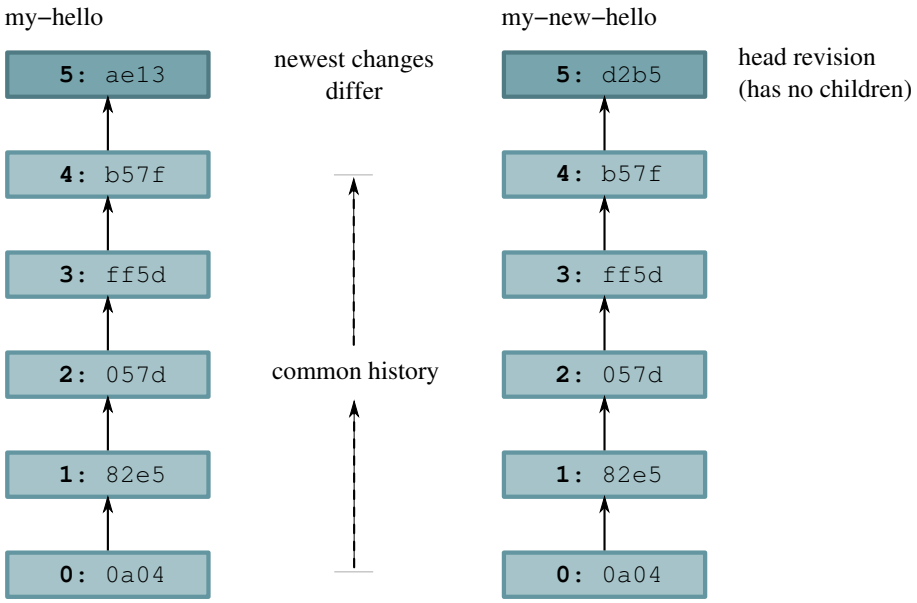

図 3.1: Divergent recent histories of the `my-hello` and `my-new-hello` repositories

“`hg pull`” を行っても、作業領域ディレクトリには影響を及ぼさないことは既に説明したとおりですので、`my-hello` から “`hg pull`” してみましょう。

```
$ hg pull ../my-hello
pulling from ../my-hello
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

作業領域ディレクトリには影響を及ぼしていませんが、“`hg pull`” コマンドは “heads” について何か警告しています。

3.1.1 Head changesets

“head” とは、リポジトリ中において、子孫（ないし子供）となるチェンジセットが存在しないチェンジセットのことです。リポジトリにおける最も最新のリビジョンは、一切の子チェンジセットを持ちませんから、従って tip リビジョンは head となりますが、１つのリポジトリには複数の head が存在しえます。

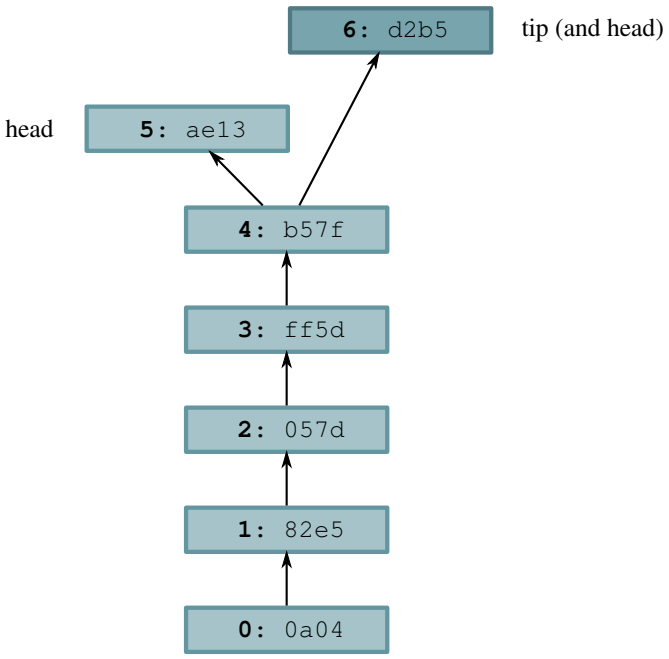

図 3.2: Repository contents after pulling from `my-hello` into `my-new-hello`

`my-hello` から `my-new-hello` への “`hg pull`” による影響を、図 3.2 で見ることができます。既に `my-new-hello` に存在していた履歴には手が付けられていませんが、新しいリビジョンが追加されています。図 3.2 からは、新しいリポジトリ（`my-new-hello`）において、**チェンジセット識別子**は同じままでも、**リビジョン番号**が異なる様が読み取れます（そして、図らずも、チェンジセットについて話をする際に、リビジョン番号を使用するのが良くない、という好例になっています）。“`hg heads`” コマンドにより、リポジトリの head を見ることができます。

```
$ hg heads
changeset:   6:30682a1ee732
tag:         tip
parent:      4:2278160e78d4
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:02 2009 +0000
summary:     Added an extra line of output

changeset:   5:88c00c2c8751
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:07 2009 +0000
summary:     A new hello for a new day.

```

3.1.2 Performing the merge

作業領域ディレクトリを、（`my-hello` から取り込んだ）新たな tip リビジョンに更新するために、いつものように “`hg update`” コマンドを実行すると、どうなるでしょう？

```
$ hg update
abort: crosses branches (use 'hg merge' or 'hg update -C')
```

Mercurial から、“`hg update`” コマンドではマージが行われない旨が通達されます。マージの実施が必要と思われる場合、強制的な実行をしない限りは “`hg update`” コマンドによる作業領域ディレクトリの更新は行われません。“`hg update`” コマンドの代わりに、“`hg merge`” コマンドを用いて 2 つの head をマージします。

```
$ hg merge
merging hello.c
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

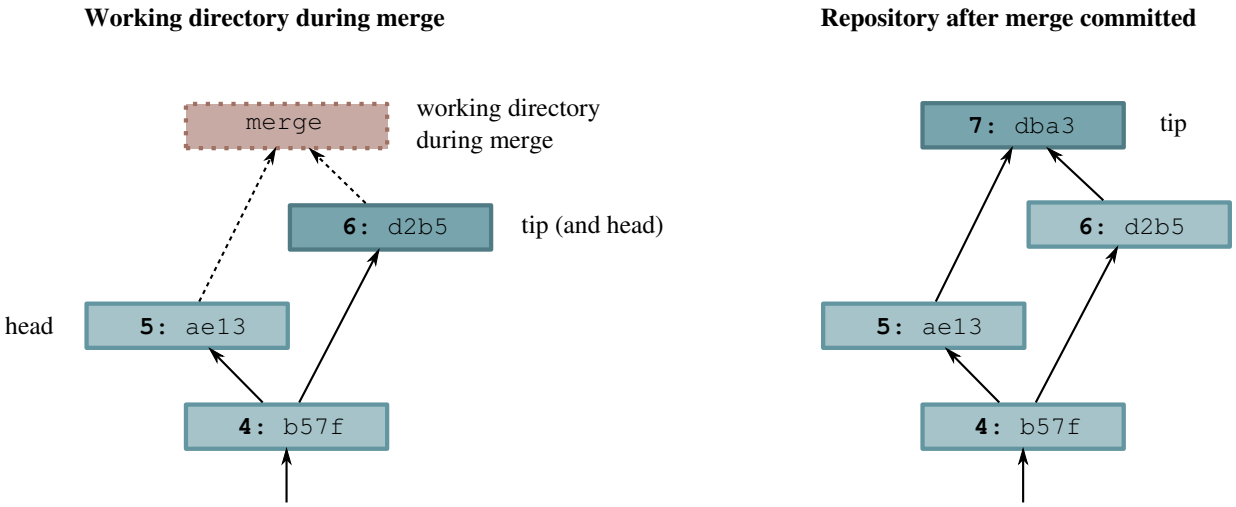

図 3.3: Working directory and repository during merge, and following commit

“hg merge” コマンドによって、“hg parents” コマンドの出力、および hello.c の内容の変更という形で、**両方**の head の変更内容が作業領域ディレクトリに反映されます。

```
$ hg parents
changeset:   5:88c00c2c8751
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:07 2009 +0000
summary:     A new hello for a new day.

changeset:   6:30682a1ee732
tag:         tip
parent:      4:2278160e78d4
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:02 2009 +0000
summary:     Added an extra line of output

$ cat hello.c
/*
 * Placed in the public domain by Bryan O'Sullivan.  This program is
 * not covered by patents in the United States or other countries.
 */

#include <stdio.h>

int main(int argc, char **argv)
{
        printf("once more, hello.\n");
        printf("hello, world!\");
        printf("hello again!\n");
        return 0;
}
```

3.1.3 Committing the results of the merge

結果を “hg commit” するまでは、“hg parents” はマージの際には常に２つの親（チェンジセット）を表示します。

```
$ hg commit -m 'Merged changes'
```

これで、新しい tip リビジョンが作成されました。先述した２つの head の**両方**を親に持つ点に注意してください。これらは、先に “hg parents” で表示したリビジョンと一致します。

```
$ hg tip
changeset:   7:e947038a4c90
tag:         tip
parent:      5:88c00c2c8751
parent:      6:30682a1ee732
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:08 2009 +0000
summary:     Merged changes

```

作業領域ディレクトリがマージの際にどのようになっているのか、そしてコミットによってどのようにリポジトリに作用するのかを、図 3.3 から読み取ることができます。マージの際に作業領域ディレクトリの親であった２つのチェンジセットは、コミットの際には新たなチェンジセットにとっての親チェンジセットとなります。

3.2 Merging conflicting changes

殆どのマージ作業は簡単に済みますが、時にはマージ対象のチェンジセット同士が、同じファイルの同じ部位を変更している場合があります。両者の変更内容が同一で無ければ、マージは**衝突**（conflict）を生じるため、両者の異なる変更内容を両立させて何らかの一貫性の取れた状態にするための決断が必要です。

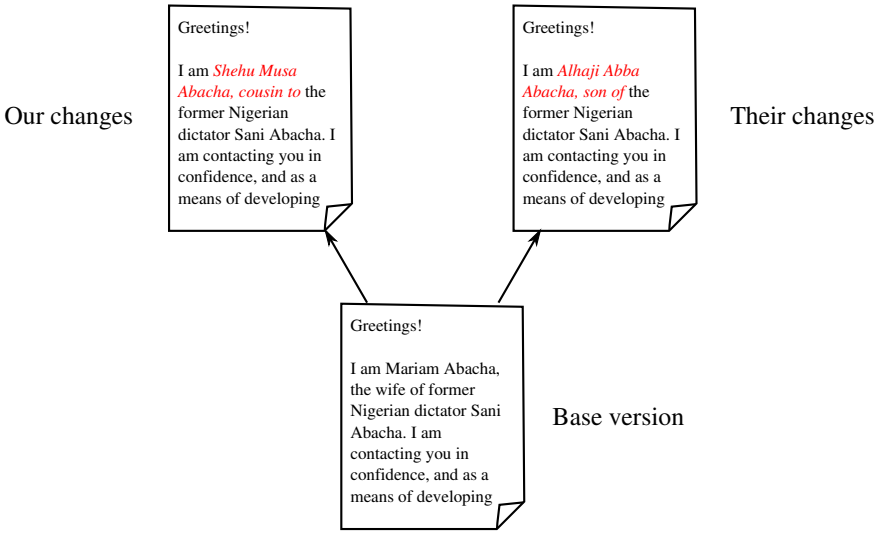

図 3.4: Conflicting changes to a document

文書に対する２つの変更の衝突の例を、図 3.4 が図示しています。両者はファイルの同じ版を元にしていますが、一方が変更を行う傍ら、他方が同じ段落に対して異なる変更をしてしまいます。変更の衝突を解消する作業とは、そのファイルがどのようになっているべきかを決定することに他なりません。

Mercurial には衝突を扱う機能が組み込まれていません。その代わりに、hgmerge と呼ばれる外部プログラムを実行します。このプログラムは、Mercurial に添付されるシェルスクリプト[1]ですが、別なプログラムを起動させることもできます。hgmerge の基底動作では、幾つかの著名なマージツールのうち、稼働環境においてインストールされていると思われるものを探します。まず始めに、非対話的マージツール[2]を実行してみますが、（人手によって解決する必要性があるために）それが失敗した場合や、そもそもそれらのツールが提供されていない場合、他のグラフィカルなマージツールの起動を試みます[3]。

HGMERGE 環境変数に起動対象プログラムないしスクリプト名を設定することで、Mercurial に hgmerge 以外を起動させる事もできます

3.2.1 Using a graphical merge tool

著者のお薦めのグラフィカルなマージツールは kdiff3 なので、グラフィカルなファイルマージツールに求められる機能について、これを題材に説明しようと思います。作業中の画面イメージが図 3.2.1 にあります。着目している１つのファイルに対して、３つの異なるリビジョンが存在することから、マージ方法は**３方向マージ**（three-way merge）と呼ばれています。それゆえ、マージツールはウィンドウ上部を３つの区画に分割しています。

- 左端に表示されているのは、ファイルの**元**（base）の版、つまりマージ対象としている２つの版にとって、最も新しい分岐元となっている版です。
- 中央に表示されているのは、マージ “先” の版[4]ですので、作業領域ディレクトリにおける変更内容が表示されます。
- 右端に表示されているのは、マージ “元”[5]ですので、マージしようとしているチェンジセットに由来する内容が表示されます。

これらの区画の下方に表示されているのは、現時点でのマージ**結果**です。マージにおける作業とは、画面上に赤字で表示された[6]、慎重なファイルのマージが必要とされる未解決の衝突を、妥当な内容で置き換えることです。

これら４つの区画は**互いに固定**されているので、いずれかの区画をスクロールさせた場合には、他の区画も相応の場所を表示するように更新されます。

ファイル中の個々の衝突箇所において、衝突を解消するために、元版／マージ先版／マージ元版のテキストを（それらの組み合わせも含めて）任意に選択することができます。また、更なる変更を行うために、マージ結果を直接手で入力することもできます。

ここで紹介し切れないほど**多くの**ファイルマージツールが存在します。これらはそれぞれ、稼動可能プラットホームや、特徴的な得手不得手などの点で異なります。殆どのツールはテキストファイルのマージに特化していますが、中には特定のファイルフォーマット（一般には XML）に特化したものもあります。

3.2.2 A worked example

本節での例では、前述の図 3.4 におけるファイル更新の履歴を再現します。元となる版のファイルを格納したリポジトリを作成することから始めましょう[7]。

```
$ cat > letter.txt <<EOF
> Greetings!
> I am Mariam Abacha, the wife of former
> Nigerian dictator Sani Abacha.
> EOF
```

[1]訳注: /bin/sh 向けだから、ということなのでしょうが、Windows のバイナリ版には添付されていません。
[2]訳注: diff3 や merge など
[3]訳注: 例えば、diff3 によるマージを行い、衝突が検出された場合はそのファイルごとに、EDITOR 環境変数で定義されるエディタ（ないし vi）を起動して、それぞれのチェンジセットに由来する変更の間での調停を要求してきます。
[4]訳注: 原文では「“our” version」
[5]訳注: 原文では「“their” version」
[6]訳注: diff3 が行単位での衝突表示であるのに比べて、GUI である利点が生きています。
[7]訳注: 実行例では、新規のリポジトリである scam の “hg init” が抜けています。

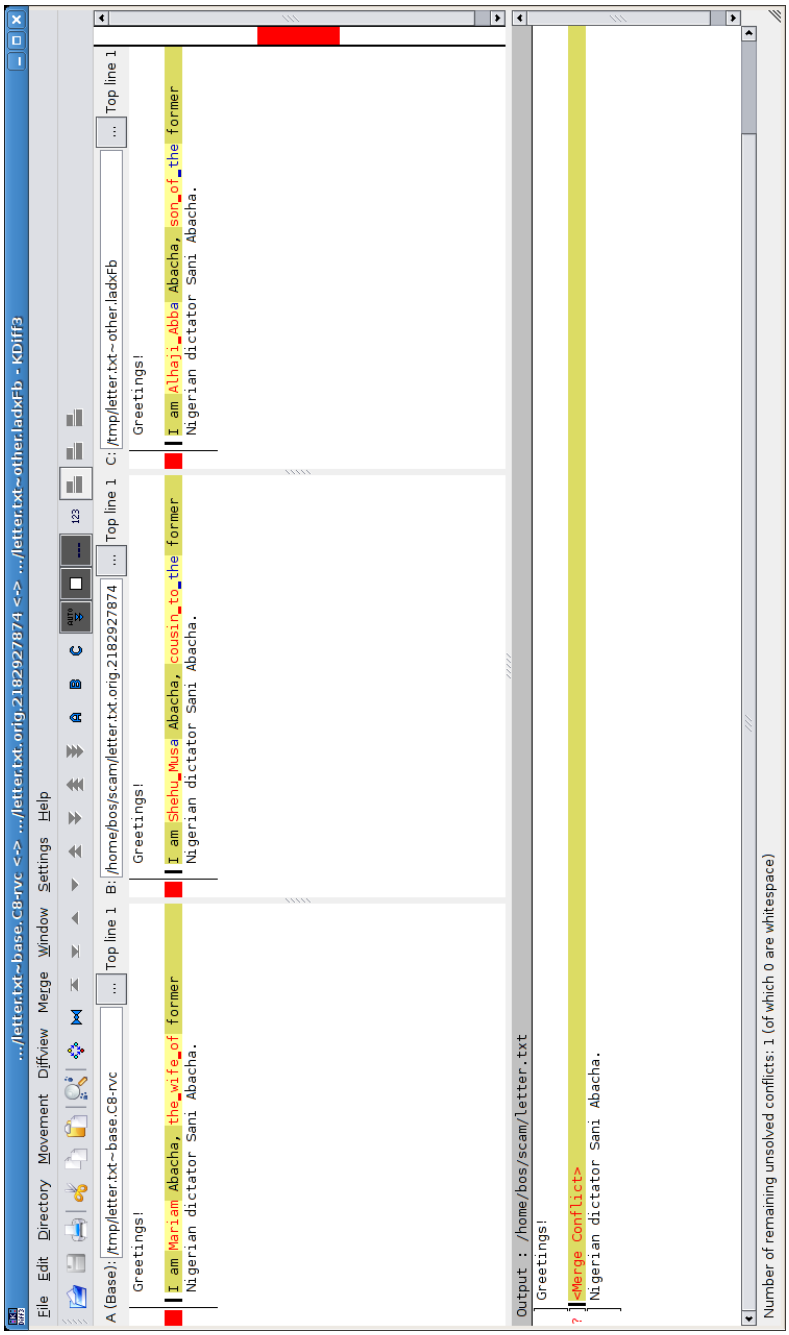

図 3.5: Using `kdiff3` to merge versions of a file

```
$ hg add letter.txt
$ hg commit -m '419 scam, first draft'
```

次に、リポジトリを複製し、ファイルを変更します。

```
$ cd ..
$ hg clone scam scam-cousin
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd scam-cousin
$ cat > letter.txt <<EOF
> Greetings!
> I am Shehu Musa Abacha, cousin to the former
> Nigerian dictator Sani Abacha.
> EOF
```

```
$ hg commit -m '419 scam, with cousin'
```

もう一つリポジトリを複製し、他の利用者によるファイルへの変更を模擬的に再現します（この模擬的な実行は、タスクごとに隔離したリポジトリの間でのマージどころか、それらのマージの際の衝突を解消することですら、決して珍しいことではない、ということを暗示しています）。

```
$ cd ..
$ hg clone scam scam-son
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd scam-son
$ cat > letter.txt <<EOF
> Greetings!
> I am Alhaji Abba Abacha, son of the former
> Nigerian dictator Sani Abacha.
> EOF
$ hg commit -m '419 scam, with son'
```

同一ファイルに２つの異なる版ができたので、マージ実施の環境が整いました。

```
$ cd ..
$ hg clone scam-cousin scam-merge
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd scam-merge
$ hg pull -u ../scam-son
pulling from ../scam-son
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
not updating, since new heads added
(run 'hg heads' to see heads, 'hg merge' to merge)
```

マージにおける対話的な処理の部分が、本書における実行例の自動実行機構 refsec:automated-example-running を損ねるため、この例では Mercurial の `hgmerge` を使用しません。その代わりに、`HGMERGE` を設定することで、Mercurial に非対話的な `merge` コマンドを実行させます。このコマンドは多くの Unix 的なシステムに同梱されています。以下の例を実際に試す際には、`HGMERGE` をわざわざ設定する必要はありません。

```
$ export HGMERGE=merge
$ hg merge
merging letter.txt
merge: warning: conflicts during merge
merging letter.txt failed!
0 files updated, 0 files merged, 0 files removed, 1 files unresolved
use 'hg resolve' to retry unresolved file merges or 'hg up --clean' to abandon
$ cat letter.txt
Greetings!
<<<<<<< /tmp/tour-merge-conflictXc7qDZ/scam-merge/letter.txt
I am Shehu Musa Abacha, cousin to the former
=======
I am Alhaji Abba Abacha, son of the former
>>>>>>> /tmp/letter.txt~other.Fi_RTZ
Nigerian dictator Sani Abacha.
```

merge コマンドは衝突を解消せずに、どの行における変更が衝突していて、その変更がどのチェンジセットに由来するのかを示す**マージマーク**を、衝突が検出されたファイルに書き込みます。

Mercurial は、merge の終了コードがマージ処理[8]失敗を示す場合、マージ処理を再実行する手順を表示します。ここで提示される手順は、マージ作業の途中で混乱してしまったり、間違ってしまったことに気付いて、グラフィカルなマージツールを中途終了させた場合などに役立ちます。

自動ないし手動のマージが失敗した場合であっても、関連の有るファイルを直接 “修正” した上で、マージ結果をコミットすることも可能です。

```
$ cat > letter.txt <<EOF
> Greetings!
> I am Bryan O'Sullivan, no relation of the former
> Nigerian dictator Sani Abacha.
> EOF
$ hg commit -m 'Send me your money'
abort: unresolved merge conflicts (see hg resolve)
$ hg tip
changeset:   2:ad1a49b7184f
tag:         tip
parent:      0:ac90448432c1
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:59:09 2009 +0000
summary:     419 scam, with son

```

3.3 Simplifying the pull-merge-commit sequence

ここまでに述べてきた変更マージの手順は単純なものですが、３つのコマンドを順に実行する必要があります。

```
hg pull
hg merge
hg commit -m 'Merged remote changes'
```

最後のコミットの際には、通常は面白くも無い “決まりきった” 内容にならざるを得ませんが、コミットメッセージを入力する必要があります。

可能であれば、必要とされる手順を低減させたいものです。実際に Mercurial は、これを可能とする fetch と呼ばれるイクステンションが同梱されています。

Mercurial は、取り扱いの利便性上から中核機能を小さく簡潔に保つ一方で、機能追加を可能にするための柔軟な拡張（イクステンション）機構を提供しています。コマンドラインから利用できる Mercurial コマンドを追加するイクステンションもあれば、例えばサーバ機能を拡張するような、“舞台裏” で機能するイクステンションもあります。

fetch イクステンションは、予想したこととは思いますが、“hg fetch” と呼ばれる新しいコマンドを追加します。“hg fetch” コマンドは、“hg pull” ／ “hg update” ／ “hg merge” ／ “hg commit” の組み合わせのように振舞います。まずは他のリポジトリから作業中のリポジトリへ変更を取り込みます。取り込んだチェンジセットによる新たな head の追加が検知[9]された場合、マージを開始し、自動的に生成されたコミットメッセージを使ってコミットを行います。新たな head の追加が無かった場合、“hg fetch” コマンドは作業領域ディレクトリを tip リビジョンで更新します。

fetch イクステンションは簡単に有効化できます。.hgrc ファイルを編集し、[extensions] セクション（無い場合は作成してください）に移動し、“fetch ” で始まる行を追加します。

```
[extensions]
fetch =
```

[8]訳注: より正確には「マージにおける衝突の解消」

[9]訳注: 他のリポジトリからの取り込みにより、３つ以上の head がリポジトリに存在するようになった場合は、マージ対象の特定ができないため、取り込みのみで処理を中断します。

（通常は、“=” の右辺にイクステンションの位置を指定しますが、`fetch` イクステンションは標準の配布物に同梱されているので、Mercurial は `fetch` を探し出すことができます）

第4章　Behind the scenes

多くの構成管理システムと異なり、Mercurial が基にしている概念は非常に単純なので、Mercurial のプログラムが実際にどのように動作するのかを理解するのは簡単です。そのような知識は必要無いかもしれませんが、筆者は内情に関する “概念理解” が有用であると考えています。

筆者自身は、内情を理解することで、Mercurial が**安全性**と**効率**に留意して設計されている、という確信を得ることができました。また、構成管理操作を行った際にソフトウェアがどのように機能するのかを、容易に覚えておけるのであれば、構成管理ツールの振る舞いに驚かされる機会が減る、という点も非常に重要です。

この章では、最初に Mercurial の設計における中核的な概念について説明した上で、実装における興味深い点に関する詳細を幾つか取り上げようと思います。

4.1　Mercurial’s historical record

4.1.1　Tracking the history of a single file

ファイルの変更を追跡する場合、Mercurial はファイルの履歴を *filelog* と呼ばれるメタデータオブジェクト形式で保存します。filelog に記録される個々の要素は、追跡対象ファイルの、とあるリビジョンを再現するのに十分な情報を保持しています。filelog は `.hg/store/data` ディレクトリ配下にファイルとして保存されており、履歴情報と、Mercurial のリビジョン検索を補助するインデックスの、２種類の情報を保持しています。

サイズが大きかったり変更履歴の多いファイルの場合、filelog を履歴情報（拡張子 “`.d`”）とインデックス（拡張子 “`.i`”）の２つに分離して保存されます。変更履歴がそれほど無い小さなファイルの場合、履歴情報とインデックスは “`.i`” 拡張子を持つ単一のファイルに保存されます。作業領域ディレクトリ中のファイルと、その変更履歴を追跡するためのリポジトリ中の filelog ファイルの対応を、図 4.1 に示します。

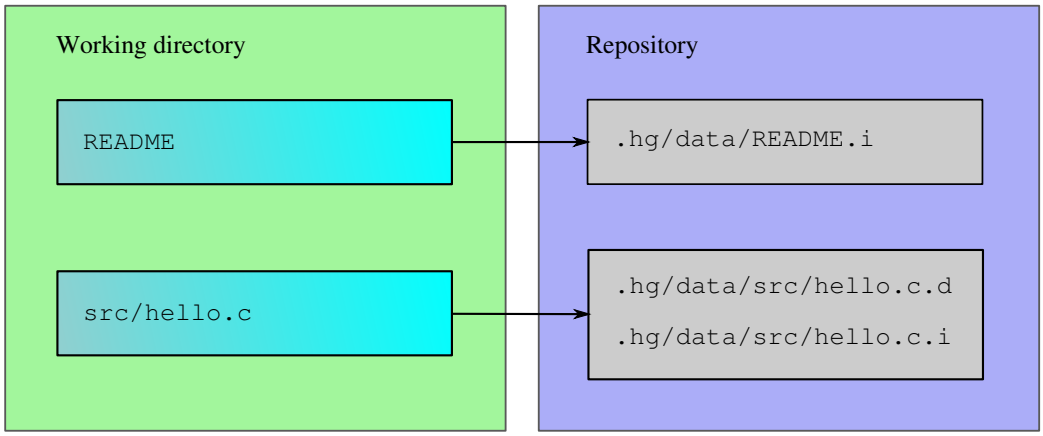

図 4.1: Relationships between files in working directory and filelogs in repository

4.1.2　Managing tracked files

追跡対象ファイルの情報をまとめるために、Mercurial は *manifest* と呼ばれる構造を使用しています。manifest に記録される個々の要素は、当該チェンジセットにおけるファイルの一覧や、各ファイルのリビジョン、幾つかのファイルのメタデータといった、個々のチェンジセットごとのファイルに関する情報を保持しています。

4.1.3 Recording changeset information

changelog は、チェンジセットのコミット主や、コミット時のログメッセージ、その他チェンジセットに関する幾つかの情報や、manifest のリビジョンといった、個々のチェンジセットに関する情報を保持しています。

4.1.4 Relationships between revisions

changelog、manifest ないし filelog における個々のリビジョンは、直接の親リビジョン（マージを行ったリビジョンの場合は、マージ対象となった２つの親リビジョン）への参照を保持しています。今述べたように、各構造に**またがった**関連性をもち、それらは必然的に階層構造を持っています。

リポジトリ中の全てのチェンジセットに関して、changelog には厳密に１つのリビジョンが保存されます。changelog における各リビジョンは、manifest 中のリビジョンへの参照を保持しています。manifest 中の各リビジョンは、チェンジセットが生成された際の各ファイルのリビジョンに対応する filelog 中のリビジョンへの参照を保持しています。この関連性を図 4.2 に示します。

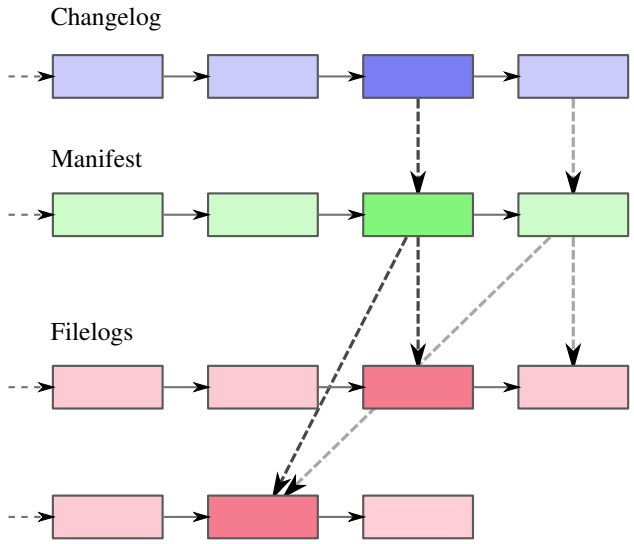

図 4.2: Metadata relationships

図からもわかるように、changelog、manifest および filelog が保持するリビジョン情報間の関係は、必ずしも “１対１” というわけではありません。２つのチェンジセットの間で manifest が変更されていない場合、それらのチェンジセットに対応する changelog 要素は、manifest 中の同じリビジョンを参照します。２つのチェンジセットの間で Mercurial が追跡するファイルが変更されていない場合、それらのチェンジセットに対応する manifest 要素は、filelog 中の同じリビジョンを参照します。

4.2 Safe, efficient storage

changelog、manifest および filelog は、*revlog* と呼ばれる同じ構造により構成されています。

4.2.1 Efficient storage

revlog は **差分**手法という仕組みを使用して、リビジョン情報を効率的に格納しています。差分手法では、ファイルの各リビジョンごとに完全な複製を保持する代わりに、旧リビジョンから新リビジョンへの変形に必要な情報を保持します。多くのファイルでのデータ格納において、差分手法は一般的に完全な複製の場合の数パーセント程度のサイズになります。

旧式の構成管理システムでは、テキスト形式のファイルでしか差分手法が適用できないものもあります。それらのシステムにおけるバイナリファイルの格納は、完全なスナップショットか、テキスト表現形式への変換によって行わ

れますが、これらは共に不経済な手法です。任意のバイナリデータを含むファイルであっても、Mercurial は差分を効率的に扱うことができますので、テキストを特別扱いする必要はありません[1]。

4.2.2 Safe operation

Mercurial は revlog の末尾にデータを**追加**するだけで、書き込まれた後からファイルの一部を改変するようなことは行いません。既存データの改変を必要とする仕組みと比較した場合、この手法は堅牢且つ効率的です。

それに加えて、Mercurial は複数のファイルにまたがった全ての書き込みを、単一の**トランザクション**の一部として扱います。トランザクションは**不可分**なものとして扱われますので、トランザクション全体が成功すれば結果の全てが利用者に見えるようになりますが、トランザクションの一部でも失敗した場合には、全ての書き込み操作は取り消されます。一方はデータの読み込みを行い他方はデータの書き出しを行うような、２つの Mercurial プロセスを同時に実行した場合でも、この不可分保証により、読み込みを混乱させるような部分的な書き込みデータを、データ読み込み側のプロセスが読み込むことはありません[2]。

Mercurial がファイルへの追加しか行わないことが、トランザクションの不可分性保証の提供を容易にしています。トランザクション保証が容易である程、それが正しく機能していることを確信できる筈です。

4.2.3 Fast retrieval

初期の構成管理システムが共に陥っていた**非効率な復旧**問題の落とし穴を、Mercurial は上手に回避しています。殆どの構成管理システムは、“スナップショット” に対する変更の追加的な連続として、リビジョンの内容を保持していました。この手法の場合、特定のリビジョンを再構築するには、最初にスナップショットを読み込み、続いて対象リビジョンとの間の全ての差分データを読み込む必要があります。ファイルの履歴が積み重なるほど、差分データを読み込まなければ成らないリビジョンが増加し、特定のリビジョンの再構築に時間が必要となります。

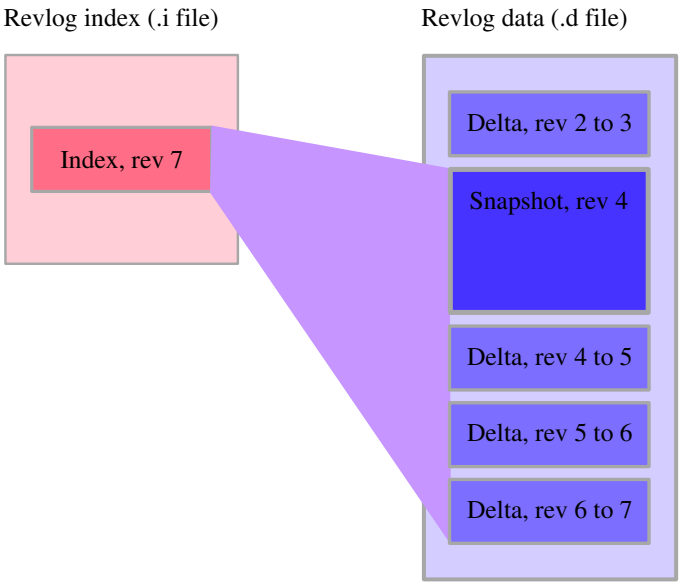

図 4.3: Snapshot of a revlog, with incremental deltas

Mercurial がこの問題の解決に使用している手法は、簡単なものですが効果的です。前回のスナップショット作成時点から、固定された閾値を超えて差分情報が蓄積された際には、差分情報の蓄積ではなく、新たなスナップショット（勿論圧縮は行います）を保存する、というものです。この手法は、**任意の**リビジョンにおけるファイルを素早く再構築できます。この手法は非常に有効であるため、他の幾つかの構成管理システムにも取り込まれています。

図 4.3 の概要が示すように、Mercurial は、revlog のインデックスファイルにおける各要素に、特定のリビジョンの再構築の際に読み込みが必要とされる、データファイル中の要素の範囲を格納します。

[1]訳注: ¡code¿cvs add¡/code¿ における¡code¿-kb¡/code¿ 指定の欠落によるファイル内容の破損、といった心配はありません。尤も、Mercurial の基底動作では、キーワードの置換等を行いませんので、そもそも心配する必要が無いのですが…。

[2]訳注: 厳密にはこの記述は正しくありません。詳細は 10.3 節を参照してください。

Aside: the influence of video compression

動画圧縮を熟知しているか、ケーブルないし衛星によるデジタルテレビ配信を視聴したことがあるならば、たいていの動画圧縮形式において各動画フレームが、先行するフレームとの差分で保持されていることをご存知かもしれません。加えてそれらの形式では、圧縮率を向上させるために “非可逆” 圧縮手法を用いていますので、フレーム間差分の数に応じて視覚的エラーが蓄積されます。

動画配信の場合、時折の信号異常による “欠落” が有り得ますし、可逆圧縮過程により生じる誤差の蓄積を制限する必要もあるため、動画圧縮側では定期的に完全なフレーム（“キーフレーム” と呼ばれます）を圧縮形式の中に挿入します。これは動画信号が中断されても、次のキーフレームの到着時点からの再開が可能であることを意味します。符号化エラーの蓄積も、個々のキーフレームでクリアされます。

4.2.4 Identification and strong integrity

差分ないしスナップショット情報のデータに対して、revlog 要素は暗号化に用いられるハッシュ値を計算して保持しています。これにより、リビジョンに関する情報の偽造を困難にすると同時に、不慮の破損の検出が容易になります。

ハッシュ値の算出は、単なる破損の検出以上のものをもたらします。ハッシュ値は各リビジョンの識別子として使用されます。Mercurial のエンドユーザとして目にするチェンジセット識別子のハッシュ値は、changelog のリビジョンに由来する値です。filelog や manifest でもハッシュ値を使用していますが、Mercurial ではこれらは舞台裏のみで使用されています。

特定リビジョンのファイルを再構築する場合や、他のリポジトリからチェンジセットを取り込んだ場合、Mercurial はハッシュ値が正しいことを確認します。一貫性に問題があることが検出された場合、警告を発した上で、進行中の全ての処理を停止します。

Mercurial が定期的に差し込んでいるスナップショットは、特定リビジョンの再構築の際の効率に加えて、部分的なデータの破損に対する堅牢性をもたらしてます。ハードウェアエラーやシステムのバグによって、revlog が部分的に破損した場合、破損を免れた revlog のデータから、破損した部位の前後共に、一部（あるいは殆どの）リビジョンを復旧することが可能です。差分のみを保持するモデルを採用する構成管理システムでは、このようなことはできません。

4.3 Revision history, branching, and merging

全ての Mercurial の revlog 要素は、通常は**親**と言われる直前のリビジョンの識別子を保持しています。実際には、各 revlog 要素は 1 つではなく 2 つの親の情報を保持できます。Mercurial は “空識別子”（null ID）と呼ばれる特別なハッシュ値を使って、“親不在” を表現します[3]。このハッシュ値は単純に 0 が連続した文字列です。

revlog の概念図を図 4.4 に見ることができます。filelog や manifest、changelog の全てが同じ構造を持っており、個々の要素が保持している、差分やスナップショットといったデータの種別が異なるだけです。

revlog における最初のリビジョン（図における底位置のリビジョン）は、2 つの親リビジョン格納領域の両方に空識別子を保持しています。“通常の” リビジョンでは、第 1 親の格納領域には親リビジョンの識別子が、第 2 親の格納領域には空識別子が格納され、親リビジョンが 1 つしかないことを表します。親リビジョンの識別子として同じ識別子を格納するリビジョン同士は、互いにブランチとなります。ブランチをマージしたリビジョンは、統合された両方のリビジョンの識別子を親リビジョンの識別子として格納します。

4.4 The working directory

Mercurial は、リポジトリで構成管理されているファイルの、特定のリビジョンにおけるスナップショットを作業領域ディレクトリに保持します。

作業領域ディレクトリは、どのリビジョンのスナップショットを保持しているのかを “知っています”。作業領域ディレクトリを特定のリビジョンで更新しようとした場合、Mercurial は (1) 相応しいリビジョンの manifest を参照し、(2) 当該リビジョンのコミット時点での管理対象ファイルを特定し、(3) 作業領域ディレクトリ中のファイルが保持すべき内容を決定します。その上で、当該チェンジセットのコミット時点と同じ内容を持つように、作業領域ディレクトリ中に各ファイルの複製を再生成します。

[3]訳注: つまり、多くの revlog 要素は、一方の親リビジョンとして空 ID を保持しています。

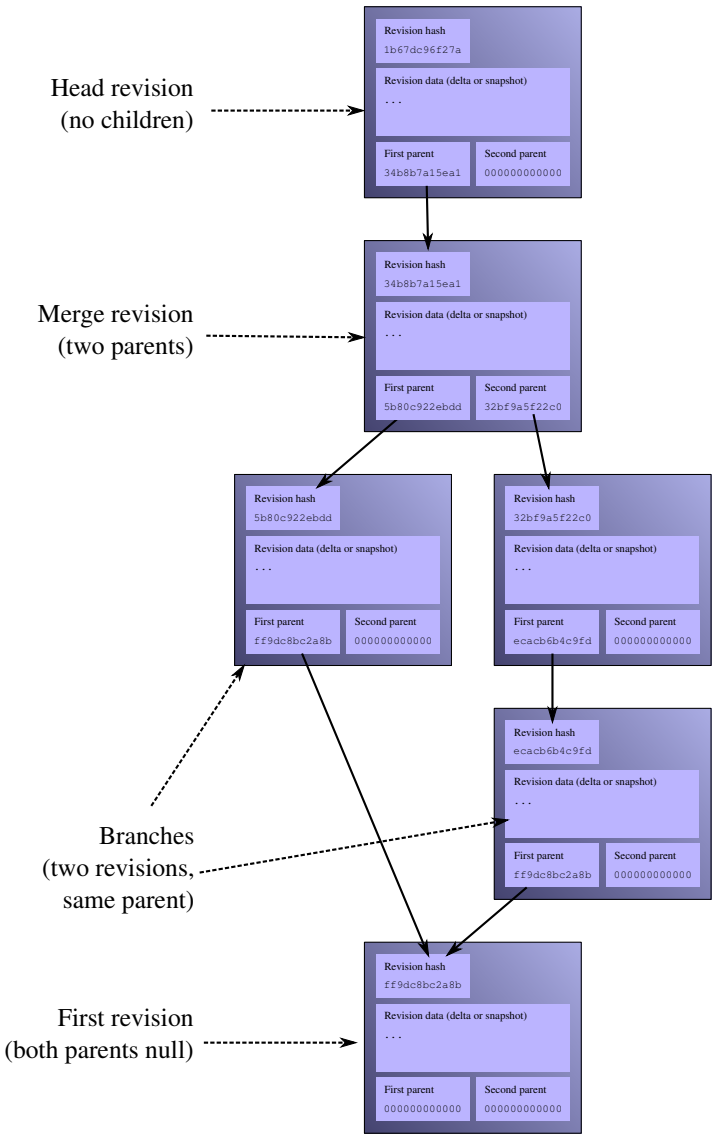

図 4.4:

dirstate 形式には、作業領域ディレクトリがどのチェンジセットで更新されているかとか、作業領域で Mercurial により構成管理されているファイルの一覧など、作業領域ディレクトリに関する Mercurial の管理情報が格納されています。

個々のリビジョンに関する revlog 要素は、２つの親リビジョン識別子を格納する領域を持っていますので、通常のリビジョン（１つの親リビジョンだけを参照）も、２つのリビジョンをマージするリビジョンも表現可能ですが、dirstate 形式も２つの親リビジョン識別子を格納する領域を持っています。“`hg update`” コマンドを実行した際には、指定したチェンジセットは “第１親”（first parent）として保持され、第２親は空識別子を保持します。チェンジセットとの “`hg merge`” を行った際には、dirstate 形式が保持する第１親は変化しませんが、第２親は “`hg merge`” コマンドに指定されたチェンジセットに設定されます。“`hg parents`” コマンドにより、dirstate 形式が保持する親リビジョンの識別子を表示できます。

4.4.1 What happens when you commit

dirstate 形式が親リビジョン情報を保持するのは、何も覚え書きのためだけではありません。Mercurial は dirstate 形式の持つ親リビジョン情報を、コミットの際の**新規チェンジセットの親チェンジセット**として使用します。

図 4.5 は、１つの親チェンジセットのみを持つ、通常の作業領域ディレクトリを表しています。図における作業領域ディレクトリの親チェンジセットは、リポジトリにおける最新で且つ子を持たないチェンジセットですので、*tip* と呼ばれます。

作業領域ディレクトリそのものを、“コミットしようとしているチェンジセット” と捉えるとわかりやすいでしょう。

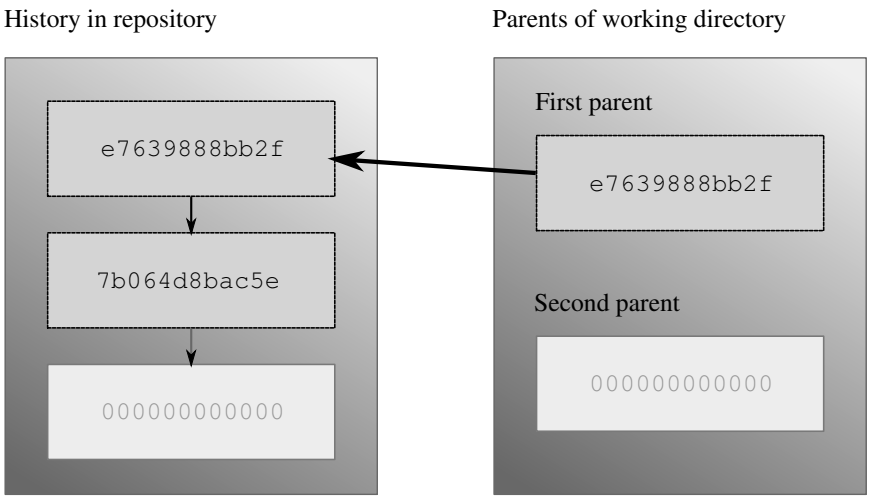

図 4.5: The working directory can have two parents

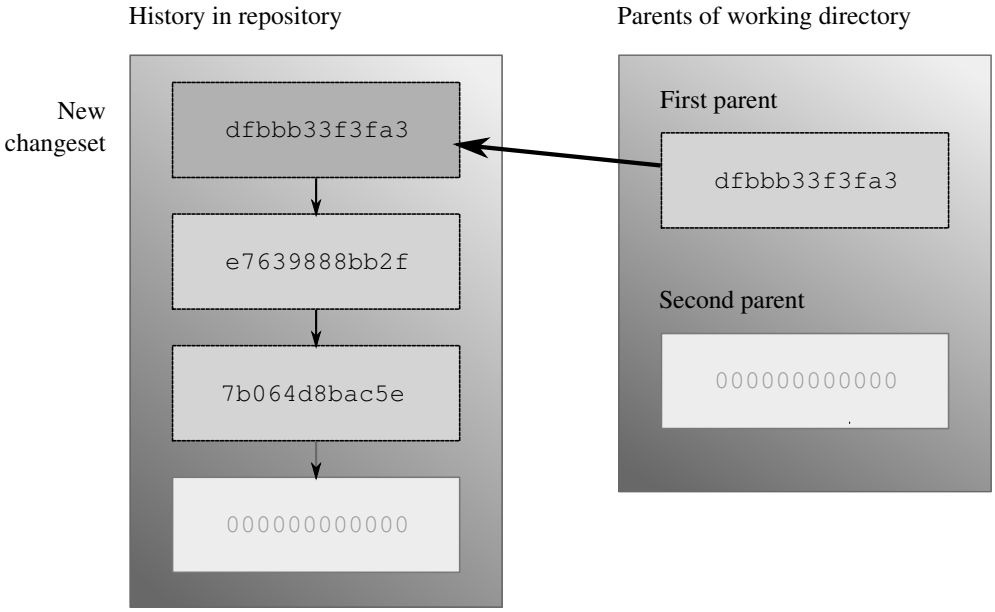

図 4.6: The working directory gains new parents after a commit

Mercurial に対して追加／削除／改名ないし複製を指示したファイルは、既に Mercurial により構成管理されているファイルへの変更と同様に、そのチェンジセットに反映されます。その新たなチェンジセットには、作業領域ディレクトリと同じ親チェンジセットが設定されます。

コミットが完了したなら、Mercurial や作業領域ディレクトリの親チェンジセットの情報を更新します。第 1 親にはコミットにより新たに生成されたチェンジセットの識別子が設定され、第 2 親には空識別子が設定されます。コミット後の模式図を、図 4.6 に示します。Mercurial はコミットの際に、作業領域ディレクトリ中のファイルには一切触れず、単に dirstate の親チェンジセット情報を書き換えるだけです。

4.4.2 Creating a new head

現時点での tip 以外のチェンジセットでの作業領域ディレクトリの更新は、良くあることです。例えば、先週火曜日時点でのプロジェクトの状態を調べたり、どのチェンジセットがバグを持ち込んだのかを調べる、といった状況です。このような状況での自然な行為は、作業領域ディレクトリを希望のチェンジセットで更新し、当該チェンジセットをコミットした時点でのファイルの内容を、作業領域ディレクトリ中のファイルを参照して確認する、というものでしょう。この行為による影響を、図 4.7 に示します。

作業領域ディレクトリを以前のチェンジセットで更新した場合、何らかの変更を行ってコミットしたなら、Mercurial はどのように振舞うのでしょうか？Mercurial はこれまでに説明してきた場合と同じように振舞います。作業領域ディ

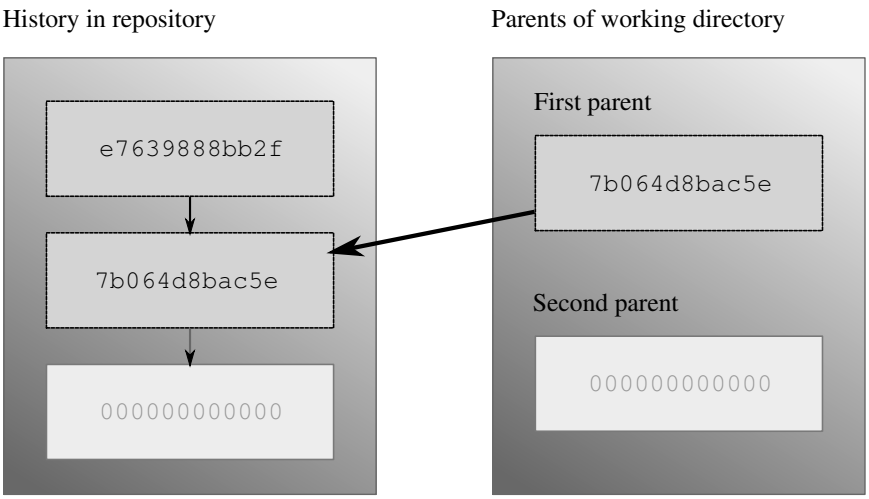

図 4.7: The working directory, updated to an older changeset

レクトリの親チェンジセットが、新規に作成されるチェンジセットの親になります。新規作成されるチェンジセットは子を持たず、よって新たな tip チェンジセットとなります。コミットの結果、リポジトリには子を持たないチェンジセットが 2 つ存在し、これらは *head* と呼ばれます。この状況を図 4.8 に示します。

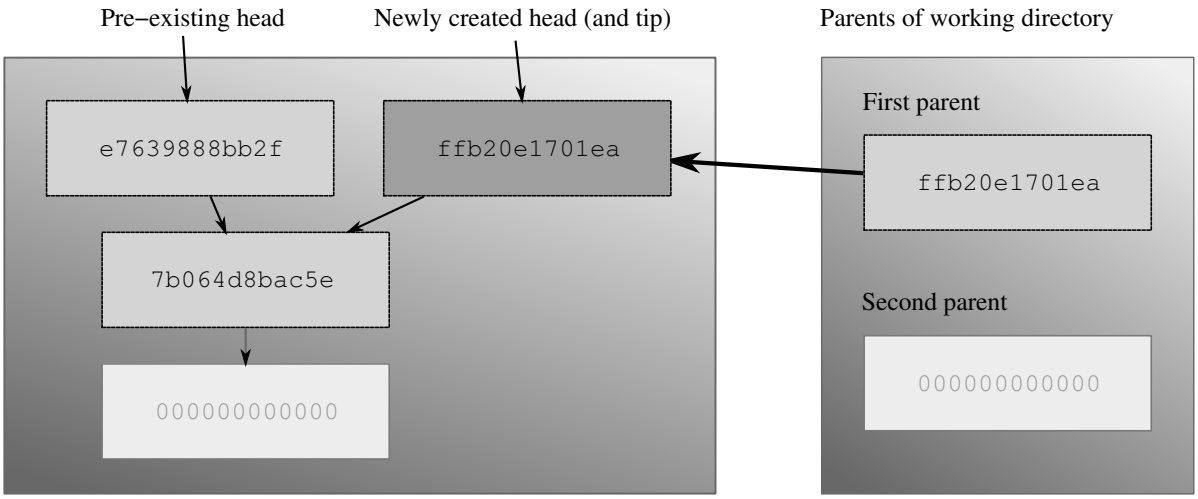

図 4.8: After a commit made while synced to an older changeset

備考: Mercurial に馴染みの無い方は、引数無しで “`hg pull`” コマンドを実行した場合の、良くある「間違い」を気に留めて置いてください。“`hg pull`” コマンドの基底動作は、作業領域ディレクトリの更新を**行いません**ので、リポジトリへの新規チェンジセットの取り込みは行われても、作業領域ディレクトリは “`hg pull`” コマンド実行前のままです。作業領域ディレクトリは当該時点での tip と同期していないため、“`hg pull`” の実行後に何らかの変更を行いコミットした場合、結果として新たな head を生成することになります。
括弧付きで「間違い」と述べたのは、この状況を修復するのに必要なことが、“`hg merge`” してから “`hg commit`” すれば良いだけだからです。言い換えるなら、このようなケースは全然深刻な状況ではない、ということです。Mercurial に慣れていない人はビックリするかもしれませんが…。このような事態を回避する別の方法や、初心者にとって意外に感じるこのような振る舞いを Mercurial がとる理由について、後ほど説明したいと思います。

4.4.3 Merging heads

“hg merge” コマンド実行の際に、Mercurial は作業領域ディレクトリの第１親は変更せずに、第２親をマージ対象として指定したチェンジセットに変更します。この様子を図 4.9 に示します。

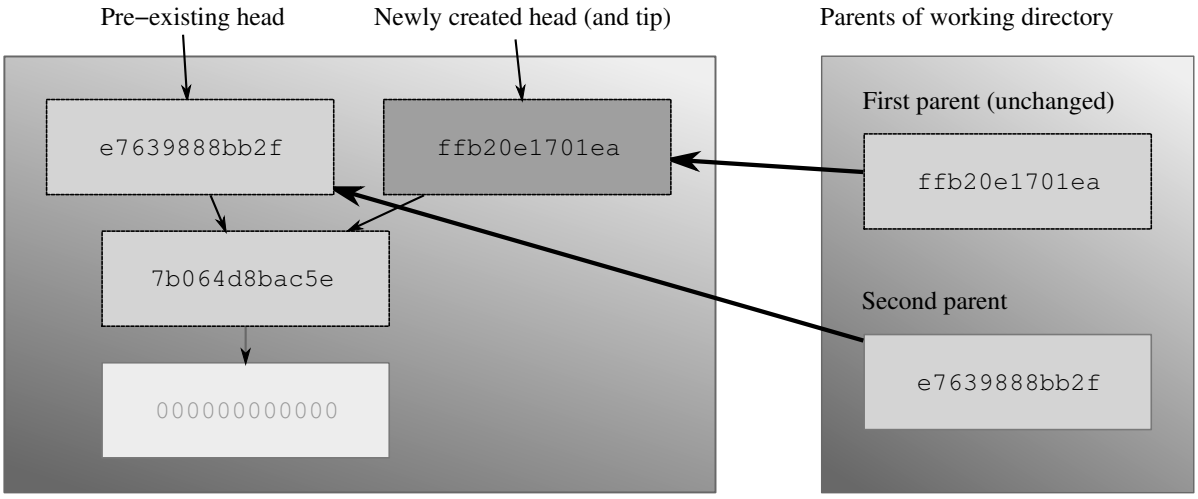

図 4.9: Merging two heads

２つのチェンジセットにおいて管理されるファイルをマージするため、Mercurial は作業領域ディレクトリを変更します。多少簡便化して説明すると、両方のチェンジセットの manifest に含まれる全てのファイルに対して、概ね以下のようにマージ処理が実施されます。

- どちらのチェンジセットでもファイルを変更していない場合、そのファイルに対しては何も行われません。
- 一方のチェンジセットが変更しているファイルを、他方が変更していない場合、変更内容を反映したファイルを作業領域ディレクトリに複製します。
- 一方のチェンジセットが削除したファイルは、他方の削除に関わらず、作業領域ディレクトリから削除されます。
- 一方のチェンジセットが削除したファイルを、他方が変更していた場合、ファイルの変更と削除のどちらを採用するのか、ユーザに対して問い合わせます。
- 両方のチェンジセットがファイルを変更している場合、内容のマージ結果をファイルに保存するために、外部マージプログラムが起動されます。この場合、ユーザによる対話的操作が必要になるかもしれません。
- 一方のチェンジセットが変更しているファイルを、他方が改名したり複製したりしている場合、変更内容が新しいファイルにも伝播するようにします。

他にも細かい話—特にマージに関しては細かい話が沢山あります—がありますが、マージに関連する一般的な振る舞いの種類はこの程度です。ご覧の様に、殆どの状況が全く自動的に処理されますし、実際のマージでも殆どの場合、衝突解消のための対話的な入力無しに自動的に完了します。

マージ後のコミットの際に処理される事柄を考える場合は、先にも述べましたが、作業領域ディレクトリを “コミットしようとしているチェンジセット” と捉えるとわかりやすいでしょう。“hg merge” コマンドが完了した後の作業領域ディレクトリは、親チェンジセットを２つ持ち、コミットによって生成される新たなチェンジセットは、これらを親チェンジセットとします。

Mercurial では繰り返しマージすることが可能ですが、Mercurial はリビジョンおよび作業領域ディレクトリの両方に対して、一度に２つの親リビジョンしか追跡できないため、個々のマージの都度コミットする必要があります。複数のチェンジセットの一括マージは技術的には可能でしょうが、ユーザが混乱したり、ひどく乱雑なマージが行われるであろうことは目に見えています。

4.5 Other interesting design features

これまでの節で、Mercurial が信頼性と性能へ注意深く配慮を払っていることを説明するために、設計における最も重要な側面の幾つかに焦点を当ててきました。しかし、詳細事項への配慮は、これだけに留まりません。Mercurial の構成において筆者の個人的な興味をそそる側面が多数あります。これまでの “big ticket” な側面とは別に、いくつかを選んで詳細を説明しようと思いますので、これらに興味があれば、良い設計のシステムの考案の際に有用な、より良い発想を得ることができるでしょう。

4.5.1 Clever compression

Mercurial はスナップショットと差分のそれぞれに対して、圧縮が有効である場合には圧縮形式で保存します。Mercurial は常にスナップショットないし差分の圧縮を**試行**しますが、非圧縮な状態よりもサイズが小さい場合に限り、圧縮形式での保存を行います。

このことは、例えば `zip` アーカイブや JPEG 画像のように、元々圧縮形式の内容を持つファイルの格納の際に、Mercurial が “適切な処置” を行うこと意味します。これらのファイルは Mercurial による 2 度目の圧縮の際には、最初のサイズよりも大きくなるのが一般的ですので、Mercurial は `zip` や JPEG ファイルをそのまま保存します。

圧縮形式のファイルのリビジョン間の差分は、一般的にはスナップショットよりも大きくなりますので、この場合でも Mercurial は “適切な処置” を行います。ファイルのスナップショットそのものを保存する場合の許容範囲を、差分情報のサイズが超えることが判明した場合、Mercurial はスナップショットを保存しますので、繰り返しになりますが、差分のみを保持するモデルよりもディスク容量が節約できます。

Network recompression

Mercurial はディスクへの履歴保存の際に、性能に対する圧縮率がそこそこ良好でバランスの取れている “収縮”（deflate）圧縮アルゴリズム（著名な `zip` アーカイブ形式が同等のものを使用しています）を使用しています。しかし、ネットワーク越しのデータ転送の際には、Mercurial は履歴データを圧縮しません。

ネットワーク接続が HTTP 経由の場合、Mercurial はデータ通信の経路全体を、より良い圧縮率を得られる圧縮アルゴリズム（`bzip2` 圧縮として広く使用されている Burrows-Wheeler アルゴリズム）で再圧縮します。リビジョン情報個別の圧縮ではなく、`bzip2` アルゴリズムと通信経路全体の圧縮という組み合わせにすることで、転送データ量を大幅に低減することができますので、殆ど全てのネットワーク形態において良好な性能を発揮できます。

（`ssh` での接続の場合、`ssh` 自身が圧縮を行うことができるので、Mercurial は接続経路の再圧縮を**行いません**[4]）

4.5.2 Read/write ordering and atomicity

不完全な書き込み内容が利用されることのないように保証する上では、ファイルへの追加書き込みだけが全てではありません。もう一度、図 4.2 を見ていただければわかるように、changelog 中のリビジョン要素は manifest 中のリビジョン要素を、manifest 中のリビジョン要素は filelog 中のリビジョン要素を指しています。この階層構造は意図的なものなのです。

データ書き込みの際には、filelog および manifest への書き込みでトランザクションが開始され、これらへの書き込みが完了するまでは changelog への書き込みは行われません。読み込みの際には、changelog を起点として manifest、filelog の順序で読み込みを行います。

changelog への書き込みに先立って、常に filelog および manifest への書き込みが完了しているので、changelog からの不完全な manifest への参照を読み込むことも、manifest からの不完全な filelog への参照を読み込むこともありません。

[4] 訳注: 訳者の経験では、サーバ側の Python が zlib を使用できない場合、ssh での push/pull が機能しなかったので、Mercurial のサイトにも同様の記述がありますが、この記述は少々辻褄が合わない気がします。

4.5.3 Concurrent access

読み書き手順と不可分性保証により、例え読み込みの最中に書き込みが行われるとしても、Mercurial は読み込みにおけるリポジトリの**排他**を必要としません。この特性は大規模化の際に非常に影響があります。任意の数の Mercurial プロセスが、書き出しプロセスの有無に関わらず、リポジトリに対して同時読み出しを安全に行うことができます。

読み出しにおける排他不要の特性は、多ユーザシステム上でリポジトリを公開している際に、複製（“`hg clone`”）や変更の取り込み（“`hg pull`”）のために、他のユーザに（あなたの）リポジトリへの**書き込み**を許可する必要[5]が無いことを意味します。読み出しを行う他のユーザには、**読み出し**権限のみの公開で済みます（この性質は構成管理システムに共通の特性では**ありません**ので、一般的なものだとは思わないでください。多くの構成管理システムでは、読み出しユーザであっても、安全な読み出しのためにはリポジトリを排他する権限が必要であり、そのためには最低でも１つのディレクトリに対する書き込み権限が必要なため、安全性と管理上で面倒な問題の原因となり得ます。）。

Mercurial が排他を行うのは、一度に１つのプロセスのみがリポジトリに書き込むのを保証場合だけです（排他に適さないと言われる NFS のようなファイルシステム[6]であっても、安全に排他できる仕組みを用いています）。リポジトリが他のプロセスにより排他されている場合、書き込みを行うプロセスは、リポジトリの排他が解除されるまで暫く待って再度排他を試行しますが、長時間に渡って排他されたままの場合は、時間切れとみなされます。そのため、例えば人知れずシステムが停止したとしても、自動化された日次処理が停止したままになったり、停止しない処理が次々と積み上がったりすることはありません。

Safe dirstate access

dirstate 形式ファイルからのリビジョン情報の読み出しに際して、Mercurial はファイルに対する排他を行ったりはせず、書き込みの際にのみ排他を行います。不完全な書き込みを dirstate 形式ファイルから読み出してしまうことを回避するため、Mercurial は対象 dirstate 形式ファイルと同じディレクトリに特有の名前でファイルを書き出し、この一時ファイルを `dirstate` ファイルへと不可分な操作で改名します。そのため、`dirstate` という「名前の」ファイルは、不完全な書き込みを持たない完全な内容であることが保証されます。

4.5.4 Avoiding seeks

比較的大量のデータ読み込み処理に対してすら、ディスクヘッドのシークは非常にコストが高くつくため、Mercurial の性能確保の重要な点は、ディスクヘッドのシークを極力回避することにあります。

例えば dirstate 形式のようなデータが、単一のファイルに保存される理由がここにあります。Mercurial により構成管理されるディレクトリごとに `dirstate` ファイルが存在する場合は、ディレクトリごとにディスクヘッドのシークが発生し得ます。そのようなディスクヘッドのシークを回避するために、Mercurial は一度に単一の `dirstate` ファイル全体を読み込みます[7]。

ローカルストレージにおけるリポジトリの複製の際には、Mercurial は “書き出し時複製” の仕組みも使用します。複製元リポジトリから複製先に個々の revlog ファイルを複製する代わりに、“ハードリンク” を使用することで、“２つのファイル名が同一内容のファイルを参照” することを手早く表明します。一方の revlog ファイルに書き込みを行う際には、Mercurial は当該ファイルのハードリンクを確認します。当該ファイルが複数のリポジトリから参照されいている場合、Mercurial は当該リポジトリ用に revlog の新たな複製を作成します。

何人かの構成管理ツールの開発者により、この方法— 完全にリポジトリ固有のものとしてファイルを複製する— がディスク使用量削減にそれほど効果的でないとの指摘を受けています。それは事実ではありますが、ディスク容量の確保は安価であり、OS への複製要求を遅延することにより高い性能を得ることができます。別な仕組みを用いる場合、性能が低下しソフトウェアの複雑さが増しますので、日々の利用における “体感” に非常に影響を及ぼします[8]。

---

[5]訳注: プロセス間で排他を行う場合、排他用のファイルを用いるか、ディレクトリそのものに排他設定を行うのが一般的ですが、そのためには書き込み権限が必要です。

[6]訳注: 構成管理システムに限らず、排他の実現に ¡code¿creat(EXCL)¡/code¿ で生成されるファイルを使用しているために、NFS では適切に排他できないプログラムが多数存在します。

[7]訳注: ディスクの利用が進んで空きブロックが断片化された場合、不連続なブロックが割り当てられますから、必ずしも「単一ファイル」＝「ヘッドのシークが回避可能」ではありませんが、少なくとも「ヘッドのシークを低減」することは可能です。

[8]訳注: つまり、Mercurial でのハードリンクの使用は、複製を行うことによるディスクヘッドのシークを低減するのが主眼で、ディスク使用量の低減が主眼ではない、ということです。

4.5.5 Other contents of the dirstate

ファイルの変更の際の Mercurial への通知が必要ないことから、ファイル変更の有無を効率的に判定するために、特別な情報を格納した dirstate 形式ファイルを使用します。作業領域ディレクトリ中の全てのファイルに対して、Mercurial はファイルの最終変更日時とその時点でのサイズを dirstate 形式ファイルに格納しています。

“`hg add`”、“`hg remove`”、“`hg rename`” ないし “`hg copy`” を明示的に使用した場合、Mercurial はこの情報を更新しますので、コミット時の振る舞いを特定できます。

Mercurial が作業領域ディレクトリ中のファイルを確認する場合、最初にファイルの変更日時を確認します。変更日時が同一ならば、ファイルは変更されていない筈です。ファイルサイズが異なっているならば、ファイルは変更されている筈です。変更日時が異なっているのにファイルサイズが同一の場合にのみ、ファイルの内容が異なっているか否かを判定するために Mercurial は実際にファイルの内容を読み込みます[9]。このように僅かな追加情報を格納することで、Mercurial が必要とする読み込みデータ量を劇的に減らすことができ、他の構成管理システムと比較して大幅に性能が改善されています。

[9]訳注: Windows 環境での改行変換を行っているような場合、バイナリ版とソース版でファイルサイズの算出手順に違いがあるらしく、“`hg diff`” が何も出力しないのに、“`hg state`” では「変更」扱いされることが稀にあります。

第5章　Mercurial in daily use

5.1　Telling Mercurial which files to track

ファイルの管理を指示しない限り、リポジトリ中のファイルに対して Mercurial は何も行いません。“hg status” コマンドは、Mercurial の管理下に無いファイルを “?” を表示することで知らせてくれます

Mercurial による構成管理を指示するには、“hg add” コマンドを使用します。ファイルの構成管理を指示したファイルの “hg status” による表示は、“?” から “A” へと変化します。

```
$ hg init add-example
$ cd add-example
$ echo a > a
$ hg status
? a
$ hg add a
$ hg status
A a
$ hg commit -m 'Added one file'
$ hg status
```

“hg commit” を実行した直後は、コミット前に追加したファイルが “hg status” により表示されることはありません。これは、“興味深い” ファイル— 変更したり、Mercurial に何らかの操作を要求したファイル—について表示するのが “hg status” の役割だからです。数千のファイルから成るリポジトリがある場合、構成管理されてはいても特に変更されていないファイルの一覧（後述するように、そのようなファイル一覧の情報を得ることもできます）を欲しいと思うことは稀です。

一旦ファイルを追加したとしても、そのファイルに対して Mercurial はすぐには何も行いません。その代わり、次にコミットを行った際にファイル状態のスナップショットを作成します。Mercurial はそれ以降、構成管理下から除外するまで、コミットの際には常に当該ファイルの変更状況を確認します。

5.1.1　Explicit versus implicit file naming

Mercurial の有用な振る舞いとして、Mercurial のコマンドにディレクトリ名を指定した場合、その指定を “当該ディレクトリ配下の全てのファイル[1]に対する操作の実施” が要求されたものとみなします。

```
$ mkdir b
$ echo b > b/b
$ echo c > b/c
$ mkdir b/d
$ echo d > b/d/d
$ hg add b
adding b/b
adding b/c
adding b/d/d
$ hg commit -m 'Added all files in subdirectory'
```

[1]訳注: 当該ディレクトリ直下のファイルならびに、サブディレクトリ以下のファイル全て

先の例で a ファイルを構成管理対象に追加した際には、Mercurial は追加されたファイルのファイル名を表示していませんが、この例では構成管理対象に追加されたファイルを表示している点に注意してください。

先の例では、追加するファイル名をコマンドラインで明示的に指定しましたので、そのような場合は利用者自身が自分の振る舞いを理解しているものとみなし、Mercurial は何も表示しません。

しかし、ディレクトリ名を指定することでファイル名を**暗示**した場合、Mercurial は特別に操作対象となった個々のファイル名を表示します。こうすることで何が実施されたのかが明確になるため、ひっそりとやっかいな問題が発生する可能性を低減します。この振る舞いは殆どの Mercurial コマンドに共通しています。

5.1.2 Aside: Mercurial tracks files, not directories

ディレクトリは Mercurial による構成管理の対象にはなりません。その代わり、Mercurial はファイルのパスを構成管理します。ファイルの生成の際には、それに先立ってパスに含まれる存在しないディレクトリを全て作成します。ファイルの削除の際には、削除されたファイルへのパスに含まれる空ディレクトリを全て削除します。たわいも無いことに聞こえるかもしれませんが、Mercurial が完全に空っぽのディレクトリを取り扱えない、という小さいながらも実用上重大な性質を示しています。

空のディレクトリが有用なことは滅多に無いですし、妥当な効果を得るための控えめな回避方法があります。Empty directories are rarely useful, and there are unintrusive workarounds that you can use to achieve an appropriate effect. それ故に、空のディレクトリを扱うことによる限定的な有益性が、それに必要とされる複雑さに見合うものではない、と Mercurial の開発陣は判断しました。

空のディレクトリをリポジトリで管理したい場合、複数の実現方法があります。１つは当該ディレクトリ直下の “隠し” ファイルを “`hg add`” することです。UNIX ライクなシステムでは、ピリオド（“.”）で始まる名前のファイルは、殆どのコマンドや GUI ツールから隠しファイルとして扱われます。この手法を図 5.1 に示します。

```
$ hg init hidden-example
$ cd hidden-example
$ mkdir empty
$ touch empty/.hidden
$ hg add empty/.hidden
$ hg commit -m 'Manage an empty-looking directory'
$ ls empty
$ cd ..
$ hg clone hidden-example tmp
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ ls tmp
empty
$ ls tmp/empty
```

図 5.1: Simulating an empty directory using a hidden file

空ディレクトリを必要とする場合のもう一つの解決方法は、自動化されたビルドスクリプトで必要になる都度作成する、というものです。

5.2 How to stop tracking a file

リポジトリにとって不要になった[2]ファイルがある場合は、“`hg remove`” コマンドを使用ます。このコマンドはファイルを削除しつつ、Mercurial に構成管理対象からファイルを除外する旨を通知します。削除されたファイルは、“`hg status`” の出力では “`R`” 付きで表示されます。

[2]訳注: 構成管理の必要性がなくなった

```
$ hg init remove-example
$ cd remove-example
$ echo a > a
$ mkdir b
$ echo b > b/b
$ hg add a b
adding b/b
$ hg commit -m 'Small example for file removal'
$ hg remove a
$ hg status
R a
$ hg remove b
removing b/b
```

“hg remove” によるファイルの削除を行うと、作業領域ディレクトリに同名のファイルを再度作成したとしても、Mercurial はそのファイルを構成管理対象から除外します。同名ファイルを再生成し Mercurial による構成管理を行う場合には、単純にそのファイルを “hg add” してください。Mercurial は新規に管理対象に加えられたファイルが、以前管理していた同名のファイルとは無関係であるとみなします。

5.2.1 Removing a file does not affect its history

重要な事ですので、“hg remove” コマンドによる操作が持つ影響は 2 つだけである、と理解してください。

- 作業領域ディレクトリから、現時点のファイルを削除します
- Mercurial に対して、次回のコミット以降、当該ファイルを構成管理対象から除外するように通知します

“hg remove” コマンドによる操作は、ファイルの**変更履歴**には一切変更を加え**ません**。
作業領域ディレクトリを “hg remove” で削除したファイルがまだ構成管理されていた時点のチェンジセットで更新した場合、そのチェンジセットがコミットされた時点の内容で、作業領域ディレクトリに当該ファイルが再生成されます。その後で、当該ファイルが “hg remove” で削除された時点のチェンジセットで更新すると、Mercurial は再び当該ファイルを作業領域から削除します。

5.2.2 Missing files

“hg remove” コマンドを使用せずに作業領域ディレクトリから削除したファイルを、Mercurial は**行方不明**とみなします。行方不明のファイルは、“hg status” の出力では “!” 付きで表示されます。Mercurial のコマンド群全般は、行方不明のファイルに関しては何も行いません。

```
$ hg init missing-example
$ cd missing-example
$ echo a > a
$ hg add a
$ hg commit -m 'File about to be missing'
$ rm a
$ hg status
! a
```

“hg status” が行方不明として表示するファイルがリポジトリ中にある場合[3]、ファイル削除後の任意の時点で “hg remove --after” を実行することで当該ファイルを構成管理対象から除外する意思があることを Mercurial に通知することができます。

[3]訳注: つまり手動でファイルを削除した場合

```
$ hg remove --after a
$ hg status
R a
```

その一方で、行方不明とされているファイルが意図せずに削除してしまったものなら、“hg revert” に当該ファイル名を指定することで、変更されていない状態にファイルを復旧することができます。

```
$ hg revert a
$ cat a
a
$ hg status
```

5.2.3 Aside: why tell Mercurial explicitly to remove a file?

ファイル削除の意思表示を一々 Mercurial に示す必要性について、疑問に思われるかもしれません。Mercurial の開発初期における削除方法は、そのように思う人にとっては望ましいものかもしれません。Mercurial は “hg commit” コマンド実行時にファイルの不在を自動的に検知し、当該ファイルを構成管理対象から除外していたのです。実際問題、この削除方法では、不慮の事態で通知も無くファイルが削除される事態が容易に起こり得ます。

5.2.4 Useful shorthand—adding and removing files in one step

Mercurial は、構成管理対象へのファイルの追加と除外を行う、組み合わせコマンドである “hg addremove” を提供しています。

```
$ hg init addremove-example
$ cd addremove-example
$ echo a > a
$ echo b > b
$ hg addremove
adding a
adding b
```

“hg commit” コマンドも、コミット実施の直前に “hg addremove” と同じ方針で構成管理対象への追加／除外を行う -A オプションを提供しています。

```
$ echo c > c
$ hg commit -A -m 'Commit with addremove'
adding c
```

5.3 Copying files

Mercurial はファイルの複製を行う “hg copy” コマンドを提供しています。このコマンドでファイルを複製した場合、Mercurial はそのファイルが元ファイルの複製であることを記録します。チェンジセットのマージの際には、Mercurial はこの複製ファイルを特別扱いします。

5.3.1 The results of copying during a merge

複製ファイルのマージの際には、変更内容が複製ファイルまで “追従” してきます。このことが持つ意味を上手く説明するために、簡単な例を作成しましょう。これまでの例と同様に、１つだけファイルを持つ簡易的なリポジトリを作成します。

```
$ hg init my-copy
$ cd my-copy
$ echo line > file
$ hg add file
$ hg commit -m 'Added a file'
```

マージを行うためには、別々の作業を平行して行う必要がありますので、リポジトリを複製しましょう。

```
$ cd ..
$ hg clone my-copy your-copy
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

最初のリポジトリに戻り、“hg copy” コマンドで最初に作成したファイルを複製します。

```
$ cd my-copy
$ hg copy file new-file
```

複製後の “hg status” コマンドの出力では、複製されたファイルは単に追加された普通のファイルと同じように見えます。

```
$ hg status
A new-file
```

しかし-C オプション付きで “hg status” を実行することで、別な行が表示されます。この行は、新たに追加されたファイルの複製**元**であることを意味します。

```
$ hg status -C
A new-file
  file
$ hg commit -m 'Copied file'
```

複製したリポジトリに戻り、平行して変更作業を行います。複製元になったファイルに対して行を追加します。

```
$ cd ../your-copy
$ echo 'new contents' >> file
$ hg commit -m 'Changed file'
```

このリポジトリでは複製元の file が変更されました。最初のリポジトリから変更内容を “hg pull” して２つの head をマージする際に Mercurial は、file に対してだけ行った変更内容を、その複製である new-file にまで伝播させます。

```
$ hg pull ../my-copy
pulling from ../my-copy
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg merge
merging file and new-file to new-file
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ cat new-file
line
new contents
```

5.3.2 Why should changes follow copies?

ファイルの複製に対してる変更が伝播される挙動は、難解に思えるかもしれませんが、多くの場合は非常に好ましい振る舞いとなります。

まずは、この伝播がマージの時**だけ**に行われる、ということに注意してください。ファイルを “hg copy” で複製し、それに引き続き複製元ファイルを変更する、という通所の作業においては何も特別なことは行われません。

もう一点、変更を取り込んだリポジトリが、ファイルを複製したことを**知らなかった**場合に限り、変更内容が複製先ファイルに伝播する、ということにも注意してください。

Mercurial がこのように振舞うのは以下のような理由のためです。例えば筆者が、ソースファイルに対して重要なバグ修正を行い、変更内容をコミットしたとします。その変更作業が行われている間に、バグの顕在化やその修正を待つ事無く、当該ファイルを “hg copy” で複製し、その複製先ファイルの変更を読者が始めてしまうかもしれません。

読者が筆者の変更を取り込んでマージした際に、Mercurial が複製への変更の反映を**行わない**場合、読者の複製先ファイルはバグを含んでいるため、手動でバグ修正を反映させる必要性を思い出さない限り、バグは複製先ファイルに**残り続ける**でしょう。

バグ修正に関する変更内容の、複製元から複製先への自動反映により、Mercurial はこの手の問題を回避しています。筆者の知る限り Mercurial は、複製ファイルに対するこのような変更伝播を行う**唯一の**構成管理システムです。

ファイルの複製とそれに続くマージの実施が一旦変更履歴に記録されたなら、複製元ファイルから複製先ファイルへのそれ以上の変更反映は通常は不要なので、Mercurial はマージ時点までは複製へ変更を伝播させますが、それ以上は行いません。

5.3.3 How to make changes *not* follow a copy

仮に、何らかの理由により、複製ファイルへの自動的な変更反映が必要ないと判断したなら、システムの通常の方法（Unix 的なシステムの場合なら cp）でファイルを複製し、“hg add” により手動で複製ファイルを構成管理対象に追加してください。ですが、その前に 5.3.2 節を読み直して、Mercurial による自動変更反映の適切性を十分に検討してください。

5.3.4 Behaviour of the “hg copy” command

“hg copy” コマンドを使用した場合、Mercurial は即座に作業領域ディレクトリに個々のファイルの複製を作成します。そのため、ファイルに修正を加えた後で、その変更をチェンジセットとしてコミットすることなく “hg copy” を行った場合、複製先ファイルはその時点までの変更内容も含んでいることになります（この振る舞いについてここで述べたのは、少々直感に反するように感じられたからです）。

“hg copy” は Unix の cp コマンドと同様に振舞います（“hg cp” という別名方が好みであれば、こちらも使用できます）。末尾の引数は**複製先**を、それ以外の先行する引数は**複製元**を意味します。複製元に単一のファイルを、複製先に存在しないパスを指定した場合、Mercurial は複製先に指定した名前で新たなファイルを生成します。

```
$ mkdir k
$ hg copy a k
$ ls k
a
```

複製先がディレクトリの場合、Mercurial は複製元ファイルを当該ディレクトリに複製します。

```
$ mkdir d
$ hg copy a b d
$ ls d
a  b
```

ディレクトリの複製の場合は、再帰的且つディレクトリ構成を保持しつつ複製されます。

```
$ hg copy c e
copying c/a/c to e/a/c
```

複製元と複製先の両方がディレクトリの場合[4]、複製元のディレクトリ構造は、複製先ディレクトリ配下で再構築されます。

```
$ hg copy c d
copying c/a/c to d/c/a/c
```

手動でファイルを複製した後で、当該ファイルが複製であることを Mercurial に通知するには、“hg remove” の場合と同様に、--after 付きで “hg copy” コマンドを使用します。

```
$ cp a z
$ hg copy --after a z
```

5.4 Renaming files

ファイルを複製するよりも、むしろ改名の方が必要とされるのではないでしょうか。ファイルの改名よりも “hg copy” コマンドの方を先に説明したのは、Mercurial が複製と改名を本質的には同等に扱っているためです。そのため、ファイルの複製における Mercurial の挙動を知ることで、ファイルの改名で期待される振る舞いを知ることができます。

“hg rename” コマンドを使用した場合、Mercurial は個々の改名元ファイルの複製を作成し、その上で改名元ファイルを削除し、それらを構成管理対象から除外します。

```
$ hg rename a b
```

[4]訳注: 先の「ディレクトリの複製の場合」は、「複製先ディレクトリが存在しない場合」を指します。

“hg status” コマンドの出力から、新たに複製されたファイルが構成管理対象に追加され、改名元ファイルが除外されていることが読み取れます。

```
$ hg status
A b
R a
```

“hg copy” 実行の場合と同様に、-C オプション付きで “hg status” コマンドを実行することで、構成管理対象に追加されたファイルが実際には、今は削除されてしまったファイルの複製ファイル、と Mercurial にみなされていることがわかります。

```
$ hg status -C
A b
  a
R a
```

“hg remove” および “hg copy” と同様に、--after オプションを指定することで、実際に改名した後で Mercurial にその旨を通知することができます。それ以外の殆どの点で、“hg rename” コマンドの振る舞い並びに指定可能なオプションは、“hg copy” コマンドと同じです。

5.4.1 Renaming files and merging changes

Mercurial の改名が「複製と削除」として実装されているため、複製の後でのマージの場合と同様に、改名の後でマージをした場合には変更が伝播されます。

あるユーザがファイルを修正し、別のユーザがそのファイルを別なファイルに改名した場合、両者がお互いの変更をマージすると、一方が行った改名元ファイルへの修正は改名先ファイルへと伝播します（この振る舞いは “普通の作業” で期待するであろう類のものですが、全ての構成管理システムがこのように振舞うわけではありません）。

複製先に対する変更の伝播が、利用者にとっておそらく有用と思われる機能ですから、ファイルの改名においても変更の伝播が重要であろうことは、明らかといえるでしょう。変更伝播機能が無い場合、ファイルの改名によって変更は簡単に行く先を失ってしまうことでしょう。

5.4.2 Divergent renames and merging

名前の広がり（diverging names）は、二人の開発者がとあるファイル— これを foo と呼びます— を各自のリポジトリで扱うことで発生します。

```
$ hg clone orig anne
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg clone orig bob
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Anne がファイルを bar に改名します。

```
$ cd anne
$ hg mv foo bar
$ hg ci -m 'Rename foo to bar'
```

その一方で、Bob がファイルを quux に改名します。

```
$ cd ../bob
$ hg mv foo quux
$ hg ci -m 'Rename foo to quux'
```

個々の開発者がファイルの命名に関する異なる意向を表明したわけですから、筆者はこの事態を衝突と捉えるのが良いと思います。

この場合のマージはどのように振舞うべきだと思いますか？改名による枝分かれが生じるチェンジセットのマージの場合、Merging は常に**両方**の改名先ファイルを維持します。

```
# See http://www.selenic.com/mercurial/bts/issue455
$ cd ../orig
$ hg pull -u ../anne
pulling from ../anne
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
1 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg pull ../bob
pulling from ../bob
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg merge
warning: detected divergent renames of foo to:
 bar
 quux
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ ls
bar  quux
```

筆者個人にとってこの振る舞いは大変意外であり、それがここでこの振る舞いを説明している理由でもあります。筆者は Mercurial に、bar を残すか、quux を残すか、あるいは両方を残すか、という選択肢による確認を行うことを期待していたのです。

実際には、ファイルの改名を行った場合、改名元ファイルを使用したビルドを行う他のファイル（例えば makefile）の修正が行われるであろうことを意味します。そのため、Anne がファイルを改名し、改名後のファイルでビルドが実施されるように Makefile を修正した場合、一方で Bob が同様の修正を別な名前で行っていますから、マージの際には作業領域ディレクトリに異なる名前のファイルのコピーが存在し、**且つ** Anne と Bob の Makefile への修正箇所が衝突している筈です。

他の利用者もこの振る舞いに意外性を感じているようです。詳細は Mercurial バグ番号 455 を参照してください。

5.4.3 Convergent renames and merging

異なる**複製元**ファイルが同じファイルを**複製先**とした際に、改名による別な種類の衝突が発生します。この場合、Mercurial は通常のマージ機構を使用し、適切な解決への誘導を要求してきます。

5.4.4 Other name-related corner cases

Mercurial は、一方がファイルに使用した名前を他方がディレクトリに使用した場合に、マージが失敗するバグが長い間残っています。この問題は Mercurial バグ番号 29 に詳細があります。

```
$ hg init issue29
$ cd issue29
$ echo a > a
$ hg ci -Ama
adding a
$ echo b > b
$ hg ci -Amb
adding b
$ hg up 0
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ mkdir b
$ echo b > b/b
$ hg ci -Amc
adding b/b
created new head
$ hg merge
abort: Is a directory: /tmp/issue29rmVkvG/issue29/b
```

5.5 Recovering from mistakes

幾つかのありがちな間違いから復旧するために、Mercurial は有用なコマンドを幾つか提供しています。

“hg revert” コマンドは、作業領域ディレクトリに対する変更を取り消します。例えば、うっかりファイルを “hg add” してしまった場合に、追加してしまったファイル名を指定して “hg revert” を実行することで、ファイルには一切変更を加える事無く Mercurial による構成管理対象から除外することができます。ファイルへの間違った変更を取り消すのにも “hg revert” が利用できます。

“hg revert” コマンドは未コミットな変更に対して有効である、ということは憶えておきましょう。但し、一旦変更をコミットした後で変更内容が間違いであることに気が付いた場合でも、選択肢は限られてはいますが対処することはできます。

“hg revert” コマンドに関する詳細と、コミット済みの変更に関する対処の詳細に関しては、9 章を参照してください。

第6章　Collaborating with other people

Mercurial は完全に非中央集約的なツールであるため、利用者相互の連携に関しては何ら制約を課すことをしません。ですが、分散構成管理に馴染みが無いのであれば、いくつかのツールや使用例を知っておくことは、妥当な作業手順のモデルを考える際に役に立ちます。

6.1　Mercurial's web interface

Mercurial は、いくつかの有用な機能を提供する、強力なウェブインタフェースを持っています。

対話的な利用の場合、ウェブインタフェース経由で１つないし複数のリポジトリの閲覧ができます。リポジトリ履歴の参照や、個々の変更（コミットメッセージや差分）の検証、および各ディレクトリやファイルの内容の参照、といったことができます。

通知に関しても、ウェブインタフェースは、リポジトリにおける変更に関する RSS 配信機能を提供します。お気に入りのツールを使ってリポジトリを "購読" することもできますし、リポジトリにおける活動状況の自動通知を即座に行うこともできます。リポジトリ提供者側における追加設定が不要であることから、筆者自身は、変更通知のメーリングリストよりも、「RSS 配信を購読」するモデルの方が非常に便利だと思います。

ウェブインタフェースにより、遠隔ユーザによるリポジトリの複製や変更の取り込み、および（サーバ側でそれを許可しているならば）変更の受理が可能になります。Mercurial の HTTP トンネリングプロトコルでは、積極的にデータの圧縮を行いますので、狭い帯域のネットワーク接続経由でも効率よく機能します。

ウェブインタフェースを触ってみる最も簡単な方法は、Mercurial のマスタリポジトリである `http://www.selenic.com/repo/hg?style=gitweb` のような、既存のリポジトリにウェブブラウザで接続してみることです。

自身でリポジトリのウェブインタフェースを提供することに興味がある場合、Mercurial には２つの選択肢があります。１つは "`hg serve`" コマンドを使用するもので、短期間の "軽量な" 稼動の場合に最適です。このコマンドの利用に関する詳細は、6.4 節を参照してください。長期的且つ常時利用可能な稼動を望む場合は、Mercurial に組み込まれている CGI （Common Gateway Interface）機能が、一般的な全てのウェブサーバで利用可能です。CGI 設定の詳細は、6.6 節を参照してください。

6.2　Collaboration models

適切な柔軟性を持つツールを使うことで、作業手順の決定は、技術的な問題から組織工学的（social engineering）な問題へと変わります。Mercurial は、プロジェクトにおける作業手順の構成に関して殆ど制限を課さないため、個別の要望に沿ったモデルの設定と運用は利用者次第となります。

6.2.1　Factors to keep in mind

いずれのモデルにおいても心得ておくべき最も重要な点は、それを利用する人々の要望と能力にどれだけ適合するか、ということです。これは自明に見えるかもしれませんが、ほんの少しの間でもこのことを忘れてはいけません。

筆者は以前、完璧と思える作業手順モデルを構築したのですが、開発チームに少なからぬ量の驚きと不和をもたらしました。複雑なブランチ群が必要な理由と、それらの間における変更の取り扱いについて説明しようと試みましたが、チームのメンバーの何人かが異を唱えてきたのです。彼らは聡明な人達でしたが、作業における制約に注意を払う[1]ことも、筆者が唱えるモデルの細部における制約の重要性に向き合おうともしませんでした。

近い将来の社会的・技術的な問題から目を背けないでください。どんな計画を実施しようとも、間違いや問題が発生した場合に備えるべきです。予想可能な問題に対して、自動的な防御や即時復旧のための仕組みの追加を考慮しましょう。例えば、リリース向けではない変更のためのブランチを作成しようとした場合、他の作業者がリリース用ブ

[1]訳注:「ルールを守る」の意か？

ランチにうっかりマージしてしまう可能性について、早い時点で考慮したほうが良いでしょう。不適切なブランチからチェンジセットをマージさせないフックを記述することで、この問題に関しては回避可能です。

6.2.2 Informal anarchy

持続可能性の点から “何でもアリ” なやり方はお薦めしませんが、簡単に把握することができるモデルであり、いくつかの特異な状況では非常に良く機能します。

一つの例として、多くのプロジェクトが、直接会うことの稀な弱くまとまった協力者グループを持っている As one example, many projects have a loose-knit group of collaborators who rarely physically meet each other. 時折の “全力疾走”（sprints）[2]を設けることで、距離によって隔てられた作業に打ち勝つグループもあります。全力疾走の機会では、多くの人が共に同じ場所（会社の会議室やホテルの会議室の類）に集まり、数日程度を閉じこもって過ごし、少量のプロジェクトに集中してハッキングを行います。

全力疾走は、大掛かりなサーバインフラを必要としない “`hg serve`” コマンドを利用するのにちょうど良い機会です。以下の 6.4 節を読むことで、すぐにでも “`hg serve`” を使い始めることができます。そうしたなら、周囲の人達にサーバを実行中であることを伝え、インスタントメッセンジャー等を使用して URL を送れば、共同作業する上での折り返し地点まで辿り着きました。ブラウザに教えられた URL を入力すれば、彼らはすぐにでもあなたの変更をレビューすることができますし、あなたからバグフィックスを入手してそれを検証したり、新機能が含まれるブランチを複製してそれを試してみたりすることができます。

その場限りのこのような形式で事を進めることの利点と欠点は、あなたによる変更の存在と、どこでアクセス可能かを知る人だけが、それを参照することができる、という点にあります。このような非公式な手法は、複数の異なるリポジトリからの取り込みが各自に要求されるため、数人以上に対しては単純に規模の拡大ができません。

6.2.3 A single central repository

小規模なプロジェクトにおいて、中央集約的な構成管理ツールからの移行する最も簡単な方法は、単一の共有リポジトリを経由して変更のやり取りをする、というものです。この体制は、より野心的な作業手順体系のための最も基本的な “構成要素” でもあります。

開発者（contributor）は、共有リポジトリの複製を行うことで作業を開始します。必要な時にいつでも変更の取り込みを行えますし、開発者の何人かは（全員でも可）、外部に公開可能になった際に変更を共有リポジトリに反映させる権限を持ちます。

このモデルであっても、共有リポジトリを経由せずにお互いの変更を直接 “`hg pull`” することは、開発者にとっては意義のあることです。例えば、暫定的なバグ修正を行ったものの、共有リポジトリにその修正を公開した場合に、その修正を取り込んだ他の開発者の作業に支障をきたす恐れがある、という場合を考えてみましょう。バグ修正を含む自分のリポジトリから一時的なリポジトリを複製し、複製先で修正内容を検証してもらえるように他の開発者にお願いすることで、潜在的な損害を低減することができます。このようにすることで、潜在的な危険性を持つ変更であっても、簡単な検証が済むまでは公開されないようにすることができます。

この種のやり取りの場合は、共有リポジトリへの安全な変更反映のために `ssh` プロトコルを使用するのが一般的です（6.5 節参照）。読み出し専用リポジトリを、CGI を使用して HTTP 経由で公開することも可能です（6.6 節参照）。リポジトリへの変更反映が必要ない場合や、リポジトリの履歴をウェブブラウザ経由で参照したい場合には、HTTP 経由での公開で十分ニーズが満たされます。

6.2.4 Working with multiple branches

一定以上の規模を持つプロジェクトにおいては、作業の進展が同時に複数の「前線」で行われることは自然な成り行きです。ソフトウェア開発の場合、どのプロジェクトでも、一定期間ごとに公式リリースを行うのが一般的です。各リリースは最初の公開の後に、一定期間の “保守状態”（maintenance mode）となることがあります。保守リリースではバグ修正のみを扱い、新規機能については取り扱わないのが通例です。これら保守リリースと平行して、（場合によっては複数の）将来のリリースに向けた開発が進行します。方向性の少し異なる、これら進行中の個々の開発を指すのに、一般的に “ブランチ” という表現を使います。

[2]訳注: オフ会とかですね。

Mercurial は特に、複数の異なるブランチを同時に管理することに適しています。それぞれの “開発指向” ごとに、別々の共有用リポジトリを用意することで、必要になる都度、あるリポジトリから別のリポジトリへのマージを行えば良いのです。各リポジトリは互いに独立していますから、誰かが明示的にマージしない限りは開発ブランチにおける不安定な変更が、安定版のためのブランチに影響を与えることはありません。

ブランチごとにリポジトリを用意する遣り方の実際の例を以下に示します。中央のサーバに “メインブランチ” があるものとします。

```
$ hg init main
$ cd main
$ echo 'This is a boring feature.' > myfile
$ hg commit -A -m 'We have reached an important milestone!'
adding myfile
```

開発者はメインブランチから複製し、変更、変更のテスト、コミットの後に、変更をメインブランチのリポジトリに反映します。

メインブランチがリリースのマイルストーンに達したならば、マイルストーンとなるリビジョンに “hg tag” コマンドで永続的な名前を付与します。

```
$ hg tag v1.0
$ hg tip
changeset:   1:be9e2ee397cd
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:28 2009 +0000
summary:     Added tag v1.0 for changeset 28be537c0906

$ hg tags
tip                                1:be9e2ee397cd
v1.0                               0:28be537c0906
```

メインブランチでは開発が継続しているとします。

```
$ cd ../main
$ echo 'This is exciting and new!' >> myfile
$ hg commit -m 'Add a new feature'
$ cat myfile
This is a boring feature.
This is exciting and new!
```

リリースマイルストーン後の任意の時点でリポジトリを複製した開発者は、リリースマイルストーンで記録されたタグを使うことで、タグが付与されたリビジョンがコミットされた時点と厳密に一致する作業領域ディレクトリを “hg update” コマンドにより複製することができます。

```
$ cd ..
$ hg clone -U main main-old
$ cd main-old
$ hg update v1.0
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cat myfile
This is a boring feature.
```

それに加えて、メインブランチでのタグ付けの後で、サーバ上のメインブランチを、新たな “安定版” ブランチ（のリポジトリ）へと複製することもできます[3]。

```
$ cd ..
$ hg clone -rv1.0 main stable
requesting all changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

安定版ブランチに対して変更する必要がある場合、開発者は**安定版ブランチ**のリポジトリから複製し、変更、変更のテスト、コミットの後に、変更を安定版ブランチのリポジトリに反映します。

```
$ hg clone stable stable-fix
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd stable-fix
$ echo 'This is a fix to a boring feature.' > myfile
$ hg commit -m 'Fix a bug'
$ hg push
pushing to /tmp/branchingFZhfNl/stable
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

Mercurial のリポジトリはお互いに（物理的に）独立しており、リポジトリ間での変更の自動的なやり取りは行われないため、安定版ブランチととメインブランはお互いに**隔離**されています。メインブランチに加えた変更が安定版ブランチに “漏れ出す” ことはありませんし、その逆に関しても同様です。

安定版ブランチにおける全てのバグ修正を、メインブランチに反映したい場合もあるでしょう。メインブランチでバグ修正を再度（手動で）行う代わりに、安定版ブランチから取り込んだ変更をメインブランに対してマージすることで、安定版ブランチにおける変更をメインブランチに持ち込むことができます。

```
$ cd ../main
$ hg pull ../stable
pulling from ../stable
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg merge
merging myfile
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg commit -m 'Bring in bugfix from stable branch'
$ cat myfile
This is a fix to a boring feature.
This is exciting and new!
```

[3] 訳注: メインブランチと安定版ブランチの各リポジトリは、必ずしも同一サーバで運用される必要はありません。

この時点でのメインブランチは、安定版ブランチには無い変更を保持していますが、安定版ブランチにおける全てのバグ修正を保持しています。安定版ブランチは、メインブランチにのみ含まれる変更には影響を受けないままです。

6.2.5 Feature branches

大規模プロジェクトで有効な変更管理方法は、開発チームを小さなグループに分割することです。プロジェクト全体が参照する単一の “マスター” ブランチから複製した共有ブランチ（= リポジトリ）を、各グループごとにそれぞれ持ちます。個々のブランチ上で作業する開発メンバーは、他のブランチにおける開発作業とは隔離されています。

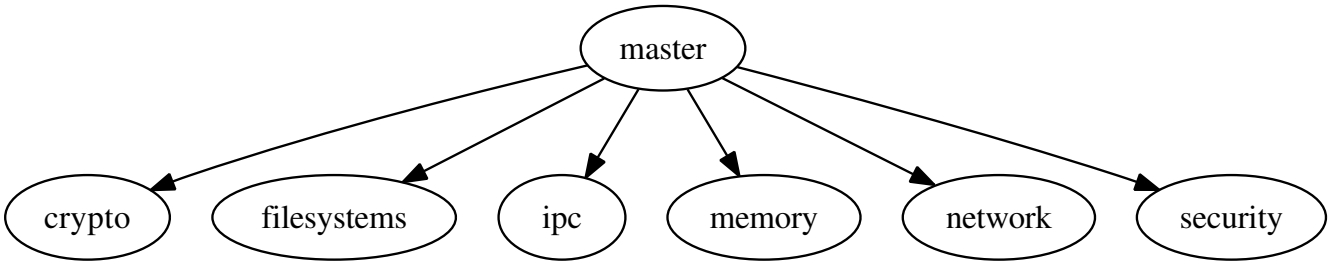

図 6.1: Feature branches

とある機能が適切な状況[4]に到達したと判断されたなら、当該機能の担当チームでは、マスターブランチ（のリポジトリ）から反映した変更を自チームのブランチ（のリポジトリ）へとマージしてから、マスターブランチへとマージ結果を反映すれば良いのです。

6.2.6 The release train

プロジェクトによっては、“train” モデルで運用されている場合もあります。“train” モデルで運用されているプロジェクトでは、リリースは数ヶ月ごとに設定されており、“train” が出発準備完了した段階[5]で提供可能な機能だけがリリースに含まれます。

このモデルは、先に説明した機能別ブランチによる作業と似ています。“train” モデルの場合、機能別ブランチが列車に乗りそこなった場合、当該機能の担当チームでは、自チームの機能ブランチ（リポジトリ）に対して、リリース列車に含まれる変更の取り込みおよびマージを行い、マージ結果に対して作業を継続する必要がある点が異なります。このマージ作業を行うことで、次回リリースの際に、当該機能が整合性を保つことができます。

6.2.7 The Linux kernel model

Linux カーネルの開発体制は、浅い階層構造と、それを取り囲む一見混沌とした群集から成り立っています。殆どの Linux 開発者が、Mercurial と同等の機能を持った分散構成管理ツールである `git` コマンドを使用しているので、Linux カーネル開発における作業手順の説明は Mercurial 利用にとっても有用性を持っています。気に入ったアイデアがあれば、ツールを超えて手法を利用することは可能なのですから。

コミュニティの中心には、Linux を創り出した Linus Torvalds 氏がいます。彼は単一のソースリポジトリを公開しており、開発コミュニティ全体にとっては、このリポジトリが “権威ある” 現行ソースツリーとみなされます。誰もが Torvalds 氏のソースツリーを複製できますが、誰のツリーから変更を取り込むかという点に関して、Torvalds 氏は非常に慎重な選択をしています。

Torvalds 氏には “信頼できる補佐役” が何人かいます。彼ら補佐役が公開している変更は、レビューが行われていなくても、大概は Torvalds 氏により取り込まれます。補佐役のうちの何人かは、“保守担当者” として承認されており、カーネルの特定のサブシステムに関する責任を負っていますとあるカーネルハッカーがサブシステムへの変更を行い、その変更を最終的に Torvalds 氏のツリーに取り込んで欲しいと考えた場合、当該サブシステムの保守担当者が誰であ

[4]訳注: “コンパイルエラーが無くなった” 状況なのか、“単体テストが完了した” 状況なのかは、各プロジェクトの構成管理方針次第となります。
[5]訳注: 「数ヶ月ごとに設定されたリリーススケジュール」を「時刻表通りの発車時刻」に例えている模様。

るかを調べて、その担当者に変更の採用をお願いする必要があります。保守担当者が変更のレビューの後に採用に同意した場合、その変更は手順に従い Torvalds 氏へと渡されます。

個々の補佐役は、変更のレビュー・承認および公開に関するそれぞれの手法を持っており、Torvalds 氏への変更送付時期の判断に関しても、それは当てはまります。それに加えて、異なる目的向けの良く知られたブランチがいくつか存在します。例えば、古い版のカーネルの “安定版” リポジトリが、必要に応じて深刻な障害の修正を適用するために、少数の人々により保守されています。何人かの保守担当者は、複数のソースツリーを公開しています。１つは実験的な変更のためのもの、１つは上流リポジトリから配布しようとしている変更のためのもの、といった按配です。他の保守担当者は、ソースツリーを１つだけ公開しています。

Linux におけるこのモデルは、２つの注目に値する特徴を持っています。１つ目は “取り込み限定”（pull only）である点です。保守担当者以外が変更を反映できるソースツリーが殆ど無く、他の人が管理しているソースツリーに変更を反映する方法が無いことから、変更を反映させたいソースツリーの保守担当者に対して、変更の採用を**お願い**する必要があります。

２つ目の特徴は、知名度と評判に基づいている点です。名の知られていない開発者からの変更依頼の場合、Torvalds 氏が依頼メールを受け取ったのなら、返信もせずに無視してしまうでしょう。しかし、サブシステムの保守担当者が依頼メールを受け取った場合、内容がレビューされた上で、それが保守担当者の基準を満たしていれば、おそらくその変更は採用されるでしょう。より “良い” 変更で貢献する程、保守担当者はあなたの判断を信頼するでしょうし、あなたの変更依頼が受理度される度合いも増すでしょう。あなたが有名になり、Torvalds 氏がまだ受理していない息の長いブランチの保守を行うようになれば、あなたの作業内容に追従するために、似たような興味を持つ人々があなたの変更を定期的に取り込むようになるでしょう。

知名度や評判は、必ずしもサブシステムや “人的” 境界を越えるわけではありません。専らストレージ系で著名なハッカーが、ネットワークのバグ修正を試みた場合、ネットワークサブシステムの保守担当者による監査は、全くの部外者による変更と同程度となるでしょう。

より整然としたプロジェクト従事の経験を持つ人にとって、相当に無秩序な Linux カーネルの開発手順は、全く非常識なものに見えることでしょう。この開発形態は、個人の気まぐれの影響を受けやすいのです。作業は各自の都合の良い時に、驚くべきペース行われます。それでもなお Linux は、成功を収めた重要なソフトウェアの１つとなっています。

6.2.8 Pull-only versus shared-push collaboration

他の人のリポジトリからは変更の反映のみしかしないモデルと、複数の人々が共有リポジトリへの変更反映を行うことができる開発モデルの、どちらが “より良い” モデルであるかは、オープンソースコミュニティにおいて継続的な議論の的になっています。です。

共有リポジトリ＋反映モデルの支持者は、その手法を積極的に使用するツールを使用する傾向にあります。Subversion のような中央集約的な構成管理ツールを使用している場合、採用するモデルの選択肢はありません。共有リポジトリ＋反映モデルがツールによって強制されるため、他のモデルを使用するには、そのツール上で独自の手法（例えば、手動で `patch` を宛てる、など）を駆使する必要があります。

Mercurial のような適切な分散構成管理ツールであれば、両方のモデルを選択可能です。利用者間の連携形態は、ツールにより強制される歪んだものではなく、固有の要望や好みに基づいて構築することができます。

6.2.9 Where collaboration meets branch management

共有リポジトリを構築し、各作業者が手元のリポジトリと共有リポジトリとの間で、変更の伝播を開始し始めたなら、チーム内の開発の方向性を同時に複数管理するという、連携に関することではありつつも、微妙に異なる難問に直面することでしょう。この問題は開発チームの連携方式と密接に関連してはいるものの、改めて取り上げる価値があるほど非常に込み入った話であることから、8 章で改めて説明します。

6.3 The technical side of sharing

本章の残りは、共同作業者に対してデータの提供を行う上での問題点に割きたいと思います。

6.4 Informal sharing with “hg serve”

Mercurial の “hg serve” コマンドは、小さく緊密で足並みの早い集団での利用に大変適しています。“hg serve” コマンドはまた、ネットワーク越しでの Mercurial コマンドの利用感を掴むための、素晴らしい手段を提供しています。

リポジトリ配下において “hg serve” を実行することで、１秒も経たずに特製の HTTP サーバが起動します。実行が停止されるまでの間にこの HTTP サーバは、任意のクライアントからの接続を受理し、当該リポジトリ中のデータの提供を行います。たった今起動したばかりのサーバの URL を知っていて、ネットワーク越しにサーバが稼動しているコンピュータと通信できるなら、ウェブブラウザや Mercurial を利用して、誰もがリポジトリからデータを読み出すことができます。ノート PC 上で稼動する “hg serve” プロセスの URL は、http://my-notepc.local:8000/ のような形式になります。

“hg serve” コマンドは汎用ウェブサーバでは**ありません**。このコマンドを使用することで２つの事が可能になります。

- 一般的なウェブブラウザ経由でのサービス対象リポジトリの履歴の閲覧
- Mercurial プロトコルによる通信を行うことで、リポジトリ内チェンジセットの “hg clone” ないし “hg pull”

とりわけ遠隔ユーザによる対象リポジトリの**変更**を許可しないことから、“hg serve” は読み出し専用としての利用が想定されています。

Mercurial を既に利用し始めているのであれば、自身のコンピュータ上のリポジトリを対象として “hg serve” を利用することができますから、ネットワーク越しに公開されているリポジトリの場合と同様に、“hg clone” や “hg incoming” のようなコマンドを使用して、“hg serve” によって起動されたサーバと通信してみましょう。ネットワーク経由で公開されているリポジトリに対するコマンドの使用方法を、手早く習得する一助に “hg serve” を使用するのも良いでしょう。

6.4.1 A few things to keep in mind

“hg serve” は、ネットワーク越しの読み出し操作を認証無しで全て許可しているため、対象リポジトリからデータを読み出すために誰が接続して来るのかを、気にしなくて良い（あるいは完全に制御できる）環境でのみ “hg serve” を使うようにすべきです。

コンピュータやネットワークへのファイヤウォールの導入状況について、“hg serve” コマンドは一切関知しません。ファイヤウォールの検出も制御もできません。実行中の “hg serve” プロセスとの通信ができない場合は、（理商社が正しい URL を使用していることを確認した**後**で）ファイアウォールの設定を確認すべきです。

“hg serve” によるネットワーク接続の受け付けは、通常は 8000 番ポートで行われます。当該ポートが既に他のプロセスにより使用されていた場合は、-p オプションを使用することで、接続受け付けポート番号を指定することができます。

“hg serve” 起動の際には通常何も出力されませんので、少々不安になるかもしれません。“hg serve” が適切に稼動していることを確認したり、共同作業者に送付する URL を知りたいのであれば、-v オプション付きで “hg serve” を起動してください。

6.5 Using the Secure Shell (ssh) protocol

Secure Shell （ssh）プロトコルを使用することで、ネットワーク接続越しに安全に変更内容の取り込み・反映を行うことができます。この接続方法を正しく機能させるには、クライアントあるいはサーバ側で少々設定が必要かもしれません。

ssh に馴染みがないのであれば、他のコンピュータと安全に通信するためのネットワークプロトコルである、と理解しておいてください。Mercurial で ssh を利用するには、サーバへのログインおよびコマンド実行ができるように、サーバ側にユーザアカウントを（必要であれば複数）用意する必要があります。

（ssh について詳しい場合、以降の説明はおそらく非常に初歩的に感じるでしょう）

6.5.1 How to read and write ssh URLs

ssh プロトコルを利用する場合の URL は、概ね以下のような形式を持ちます。

```
ssh://bos@hg.serpentine.com:22/hg/hgbook
```

1. “ssh://’ 部分が Mercurial に ssh プロトコルの利用を指示します
2. “bos@” 部分がサーバへのログインにおけるユーザ名を表します。サーバでのユーザ名がローカルマシン上のユーザ名と一致する場合は、この部分を省略できます。
3. “hg.serpentine.com” 部分はログイン先サーバのホスト名を表します。
4. “:22” 部分はサーバに接続する際のポート番号を表します。ssh 接続における既定ポート番号は 22 番ですので、22 番**以外**のポートを使用する場合のみ指定が必要です。
5. URL の残りの部分はサーバ上におけるリポジトリのパスを表します。

ssh プロトコルにおける URL 表記のパス要素部分には、値の解釈に関する標準的な手法がないために、混乱の余地が多々あります。一群のプログラムは、パス要素部分に関して他のプログラムと異なる振る舞いをします。このような状況は理想的ではありませんが、状況が変わりそうにはありません。ですから以降の説明は注意深く読んでください。

Mercurial はパス部分を、サーバにログインするユーザの、ホームディレクトリに対する相対パスとみなします。例えば、サーバにおける foo ユーザのホームディレクトリが/home/foo である場合、ssh プロトコルにおける URL のパス要素が bar であれば、その URL により**実際に**参照されるのは/home/foo/bar ディレクトリです。

他のユーザのホームディレクトリに対する相対パスを指定する場合は、チルダ文字（ ）にユーザ名（ここでは otheruser とします）を続けたパスで始まる、以下のような表記になります。

```
ssh://server/~otheruser/hg/repo
```

絶対パスによる指定を行う場合は、以下のようにパス要素をダブルスラッシュで始めます。

```
ssh://server//absolute/path
```

6.5.2 Finding an ssh client for your system

殆ど全ての Unix ライクなシステムには OpenSSH が事前導入されています。Unix ライクなシステムを使用している場合、which ssh と入力することで ssh コマンド（通常は /usr/bin にインストールされています）のインストールの有無を確認することができます。予想に反してインストールされていなかった場合には、システム添付のドキュメントを参照してインストール方法を調べてください。

Windows の場合、妥当な ssh クライアントを選択してダウンロードする必要があります。主な選択肢は 2 つあります。

- Simon Tatham 氏による PuTTY [Tat] は、ssh クライアントコマンド一式を提供しています。
- 面倒な事への耐性が高い方なら、Cygwin 上の OpenSSH を使うのも良いでしょう。

どちらの場合でも、Mercurial が ssh クライアントコマンドを探し出せるように Mercurial.ini ファイルを編集する必要があるでしょう。例えば PuTTY を使用するなら、コマンド行で実行する ssh クライアントとして plink を実行することになります。

```
[ui]
ssh = C:/path/to/plink.exe -ssh -i "C:/path/to/my/private/key"
```

備考: plink へのパスが空白文字を含む場合、Mercurial は plink コマンドを正しく起動できません（ですので C:
Program Files にインストールするのは、よくありません）。

6.5.3 Generating a key pair

ssh クライアントを使用する度に、毎回パスワード入力を繰り返さなくても良い様に、鍵対（key pair）[6]を生成することをおすすめします。Unix ライクなシステム[7]では、ssh-keygen コマンドで鍵対を生成します。Windows 上で PuTTY を使用している場合は、puttygen コマンドで鍵対を生成します。

鍵対を生成する場合、パスフレーズで鍵を守るようにするのが、一般には**非常に**賢明とされています（ssh プロトコルによる安全なネットワークを、自動化された処理において使用する場合を除く）。

しかし、単に鍵対を生成しただけでは不十分です。ネットワーク経由でログインするサーバ側アカウントにおいて、承認鍵一覧に公開鍵を追加登録する必要があります。OpenSSH が導入されているサーバでの公開鍵の追加は、当該アカウントの .ssh ディレクトリ配下の authorized_keys ファイルに公開鍵の内容を追加することで行われます。

Unix ライク名システムでは、公開鍵は通常 .pub 拡張子を持っています。Windows 上で puttygen を使用する場合は、任意のファイル名で保存可能ですし、公開鍵の内容が表示されているウィンドウから authorized_keys へ直接貼り付け（paste）ることも可能です

6.5.4 Using an authentication agent

認証エージェントは、パスフレーズをメモリ上に格納するデーモンプロセスです（そのため、ログアウト後に再度ログインした場合、パスフレーズは失われます）。認証エージェントの稼動を検知すると、ssh クライアントは認証エージェントにパスフレーズの問い合わせを行います。認証エージェントが稼動していないか、あるいは必要なパスフレーズを記憶していない場合は、Mercurial によるサーバ連携（例: “hg push” や “hg pull”）の都度、パスフレーズの入力が必要です。

認証エージェントによるパスフレーズ保存の欠点は、入念に準備した攻撃者にとっては、たとえ定期的に再起動しているシステムであっても XXXXXX power-cycled XXXX パスフレーズの平分を復元可能である点です。この問題が許容可能なものか否かは、各自で判断する必要があります。認証エージェントを使用することで、繰り返しパスフレーズを入力する手間を大幅に低減することができます。

Unix ライク名システムでは、認証エージェントは ssh-agent という名前で、ssh-add コマンドを使ってエージェントの記憶領域にパスフレーズを保存します。Windows 上で PuTTY を使用する場合は、pageant コマンドが認証エージェントして振舞います。システムトレイに追加されたアイコンをクリックすることで、格納されたパスフレーズの管理を行うことができます。

6.5.5 Configuring the server side properly

初心者にとって ssh の設定は面倒なので、問題が発生する状況も多岐に渡ります。Add Mercurial on top, and there’s plenty more scope for head-scratching. XXXXX 問題発生の可能性は、クライアント側ではなくサーバ側の方が高いです。ありがたいことに、一旦正しく動作する設定ができてしまえば、通常は無期限に正しく動作し続けます。

Mercurial で ssh サーバと通信をしてみる前に、通常の ssh ないし putty コマンドによるサーバとの通信を確認するのが無難です。直接コマンドを使用した際に問題が発生したならば、Mercurial が機能しないことは確実です。更に悪いことに、Mercurial を介しての ssh サーバとの連携は、根本的な原因が隠れてしまいます。ssh に関連する Mercurial の問題を解決する場合は、Mercurial の不具合を疑う**前に**、ssh クライアントコマンドの直接実行が機能することを確認してください。

サーバ側で最初に確認すべき事は、あるマシンからサーバマシンへの実際のログインの可否です。ssh ないし putty でログインできない場合、表示されるエラーメッセージから問題特定のヒントが得られるかもしれません。よくある問題には以下のようなものがあります。

[6]訳注: 「公開鍵」（public key）と「秘密鍵」（private key）の対が生成されます。
[7]訳注: Windows の Cygwin 環境含む

- “connection refused” が表示される場合は、ssh サーバプロセスが起動されていないか、ファイヤーウォール設定によりネットワーク接続できないことが原因です。
- “no route to host” が表示される場合は、接続先のサーバアドレスが間違っているか、ファイヤーウォールによって接続が厳重に禁止されていることが原因です。
- “permission denied” が表示される場合は、サーバ接続の際のユーザ名、パスフレーズ、ないしサーバ側ユーザのパスワードの入力を間違えていることが原因です。

これまでの話をまとめると、サーバマシン上の ssh サーバプロセスとの通信に問題がある場合、まずはサーバプロセスの稼動状況を確認してください。多くのシステムでは、ssh 自体はインストールされていますが、初期状態では無効化されている場合があります。この確認が済んだなら、次に確認するのは、ssh サーバプロセスが外部からの接続を受け付けるポート（通常は 22 番）に対する外部からの接続を、サーバのファイヤーウォール設定が許可しているか否かです。これら 2 つの確認を済ませるまでは、突拍子もない設定ミスの可能性に関して心配する必要はありません。

秘密鍵用パスフレーズの保持のために、クライアント側で認証エージェントを使用している場合は、パスフレーズやパスワードの問い合わせを受ける事無く、サーバにログインできていなければなりません。パスフレーズを問い合わせるプロンプトが表示される場合、問題の可能性のあるものが幾つかあります。

- `ssh-add` ないし `pageant` によるパスフレーズの格納を忘れているのかもしれません。
- 想定しているものとは別な鍵のパスフレーズを格納しているのかもしれません。

サーバ側ユーザのパスワードの問い合わせがあった場合、別な問題の可能性を検討する必要があります。

- サーバ側ユーザの、ホームディレクトリないし `.ssh` ディレクトリの権限設定が、過度に緩く設定されているのかもしれません。ssh サーバプロセスはその場合、`authorized_keys` ファイルの信頼性が低いものとして読み込みを行いません。例えば、ホームディレクトリないし `.ssh` ディレクトリが、グループに対する書き込み権限を設定されている場合、パスワード問い合わせが行われる、といった症状が見られます。
- `authorized_keys` ファイルそのものに問題がある可能性もあります。このファイルへの書き込み権限が所有者以外にも設定されている場合、ssh サーバプロセスはファイルの信頼性が低いものとして読み込みを行いません。

以下のコマンド実行に対して、（サーバ側の）現在時刻を表示する 1 行だけが出力される、という状態が理想的です。

```
1 ssh myserver date
```

上記のような非対話的なコマンド実行の場合にも、バナー表示やそれに類する表示が行われるような設定が、連携先サーバ側で行われている場合には、この先の手順に進む前に、対話的な実行[8]の時にのみ、これらが表示されるように設定変更してください。これを怠ると、バナー等の表示が Mercurial の出力を混乱させてしまいます。更に問題なことに、バナー等の表示は Mercurial コマンドの遠隔実行における潜在的な問題と成り得ます。非対話的な ssh 連携において、Mercurial は極力バナー等の表示の検知ならびに無視に努めますが、必ずしも全てが無視できるわけではありません（サーバ側でログイン時実行スクリプトをカスタマイズする場合、`tty -s` コマンドの戻り値を判定することで、当該スクリプトが現在対話シェルで実行されているか否かを判定することができます）[9]。

素の ssh によるサーバ連携が機能することを確認したならば、次に確認するのは、サーバ側での Mercurial 実行の可否です。以下のコマンド実行が正しく機能することを確認してください。

```
1 ssh myserver hg version
```

[8]訳注: 「ssh によるログイン時」の意

[9]訳注: ログインスクリプトでの出力以外でも、フック実行時に標準出力に対して何らかの表示があった場合、Mercurial は「連携における想定外のデータ授受」とみなすため、注意が必要です。

通常の “hg version” 出力ではなくエラーメッセージが表示される場合、大概は /usr/bin に Mercurial がインストールされていないことが原因です。その場合でも、必ずしも /usr/bin にインストールする必要はありません。しかし、考え得る以下の幾つかの原因に関して確認が必要です。

- Mercurial は本当にサーバにインストールされていますか？変な質問と思われるかもしれませんが、これは非常に重要な確認事項です。
- シェルのコマンドサーチパス（通常は PATH 環境変数で設定）の設定が単に不適切なのかもしれません。
- ひょっとしたら、PATH 環境変数が hg コマンドの格納場所を指すように設定されるのは対話的なログイン時にのみ、という可能性もあります。PATH 環境変数の設定を不適当な起動スクリプトで行っている場合に、このような現象が発生します。各自の使用しているシェルのドキュメントを確認してみましょう[10]。
- PYTHONPATH 環境変数による Mercurial の Python モジュール格納ディレクトリの参照が必要であるケースもあります。不適切な設定だったり、対話的ログイン時にのみ設定されている可能性があります。

ssh 経由での “hg version” コマンド実行が成功したなら準備は完了です。サーバ・クライアントは共に問題解決済みとなりました。サーバ上で公開されている リポジトリに、当該ユーザ名による Mercurial でのアクセスが可能になっている筈です。ここまでの確認をクリアした上で、Mercurial と ssh の連携において問題が発生した場合、問題発生の状況をより明確にするために、--debug オプションを付けての実行を試してみてください。

6.5.6 Using compression with ssh

ssh プロトコルを使用する場合、ssh プロトコル自身が通信時にデータ圧縮を行うため、Mercurial は圧縮を行いません。しかし、ssh クライアントの（通常の）基底動作では、圧縮を**行いません**。

高速な LAN の場合を除けば（無線ネットワークであっても）、通信時の圧縮は Mercurial のネットワーク経由の処理を顕著に高速化します。例えば WAN 経由での連携の場合、かなり大きなリポジトリの複製に要する時間が 51 分から 17 分に低減した、との性能計測報告もあります。

ssh と plink の両方とも、通信時圧縮を有効化する-C オプションを受け付けます。hgrc ファイルを以下のように編集することで、ssh プロトコル利用の際に常に圧縮を行うように Mercurial に対して指定できます。

```
[ui]
ssh = ssh -C
```

ssh を使用している場合は、連携先サーバとの通信の際には常に圧縮を行うように設定することもできます。この設定を行うには、ホームディレクトリ配下の.ssh/config ファイル（無い場合は新規に作成します）に以下のように記述します。

```
Host hg
  Compression yes
  HostName hg.example.com
```

上記の記述は、hg という別名（alias）を作成します。ssh 実行の際のコマンド行記述や、Mercurial の ssh プロトコルにおける URL として、hg を（ホスト名として）使用した場合、ssh は通信時圧縮を行いつつ hg.example.com に接続します。この設定により、入力の便利な省略名と、圧縮指定の両方を手にすることができます。

6.6 Serving over HTTP using CGI

意気込み次第では、Mercurial の CGI インタフェースの設定は、数分のものを数時間にしてしまう可能性があります。

[10]訳注: 例えば bash の場合、対話的ログインか否かで.bashrc、.bash_profile、.profile および.login といった各ファイルの読み込みの有無が変化します。また、ディストリビューションによっては、非対話的な実行の際には、/etc/bashrc による/etc/profile.d 配下の設定ファイル読み込みが行われない場合があります（2.6.x 系カーネルベースのものは読み込まない方針の模様）ので、PYTHONPATH の件も含めて、システムワイドな設定を行う方は注意が必要です。ssh myserver env 実行で出力される環境変数一覧を確認してみましょう。

最も単純な例から初めて、より複雑な設定へと向けて進めてゆきましょう。最も基本的なケースですら、ウェブサーバの設定ファウルの読み書きを行う必要が出てくることでしょう。

> **備考:** ウェブサーバの設定は複雑で、扱いにくく、且つシステム依存性の高い作業です。そのため本書では、発生するであろう問題のケースを全て網羅するような手順を示すことができません。以降の記述は、慎重さと各自の判断をもって読み進めるようにしてください。沢山間違えたり、サーバのエラーログ解析に時間を費やす覚悟が必要でしょう。

6.6.1 Web server configuration checklist

読み進める前に、システムの設定状況に関する幾つかの確認を行いましょう。

1. ウェブサーバはインストールされていますか？Mac OS X は Apache がインストールされた状態で出荷されますが、多くのシステムではウェブサーバはインストールされていません。
2. ウェブサーバがインストールされている場合、それは実際に稼動していますか？ウェブサーバがインストールされていた場合でも、多くのシステムの基底状態は、ウェブサーバが無効化されています。
3. CGI を稼動させようとしているディレクトリは、ウェブサーバの設定で CGI の実行が許可されていますか？多くのウェブサーバの基底状態は、CGI プログラムの実行機能が明示的に無効化されています。

ウェブサーバがインストールされていない場合や、Apache ウェブサーバの設定経験があまり無い場合には、Apache ウェブサーバの代わりに `lighttpd` ウェブサーバの利用をお薦めします。Apache ウェブサーバの設定は、凝っていて且つわかりにくいという評判に見合うものがあります。`lighttpd` は Apache ウェブサーバ程の機能は無いものの、足りない機能の殆どが Mercurial リポジトリの運用には関係ないものです。それに加えて、明らかに `lighttpd` は Apache ウェブサーバよりも簡単に利用が開始できます。

6.6.2 Basic CGI configuration

Unix 的なシステムを利用している場合、ウェブページとして公開するための `public_html` のようなディレクトリを、ホームディレクトリ配下に持つのが共通認識となっています。このディレクトリ直下に置いた `foo` という名前のファイルは、`http://www.example.com/~username/foo` という URL で参照可能になります。

設定を始めるに当たって、Mercurial のインストール先に格納されている `hgweb.cgi` スクリプトの所在を確認してください。システム上の所在がすぐにはわからなかった場合は、Mercurial のマスターリポジトリから `http://www.selenic.com/repo/hg/raw-file/tip/hgweb.cgi` を直接ダウンロードしてください。

上記スクリプトを `public_html` 配下に配置し、実行可能となるように権限設定を行います。

```
cp .../hgweb.cgi ~/public_html
chmod 755 ~/public_html/hgweb.cgi
```

`chmod` コマンドへの `755` 引数指定は、スクリプトに実行可能権限を付与する以上の付加的な指定を意味します。この設定により、スクリプトが誰からも実行可能になると同時に、“group” および “other” による書き込み権限が**剥奪**されます。これらの書き込み権限を有効なままにした場合、Apache の `suexec` サブシステムは、おそらくスクリプトの実行を拒否するでしょう。実のところ `suexec` は、スクリプトが配置されている**ディレクトリ**に対する “group” および “other” による書き込み権限が剥奪されていることも要求します。

```
chmod 755 ~/public_html
```

What could *possibly* go wrong?

CGI を配置したならば、ウェブブラウザを起動して `http://myhostname/~myuser/hgweb.cgi` に相当する URL にアクセスしてみましょう。但し、ちょっとした失敗には**身構えておいてください**。所望の URL へのアクセスが失敗する公算は非常に高く、その理由は多岐に渡ります。実際のところ、以下の起こり得るエラー要因の全てで躓く可能性がありますから、この先は注意深く読み進めてください。以下で述べる問題は、まっさらな状態からインストールした Apache を使い、この実例を行うために新たに生成したユーザアカウントで、Fedora 7 上で作業を実施した際に、筆者が実際に直面した全ての問題です。

使用しているウェブサーバは、ユーザ毎のディレクトリを無効化しているかもしれません。Apache を使用している場合は、設定ファイル中に `UserDir` 指定の有無を確認してください。この指定が無い場合、ユーザ毎ディレクトリは無効になります。指定が有っても無効化されている場合も、ユーザ毎ディレクトリは無効になります。有効な `UserDir` 指定がある場合、`UserDir` 指定で記述されている文字列（例えば `public_html`）が、ホームディレクトリ直下で Apache が参照するサブディレクトリ名になります。

ファイルのアクセス権限が厳しすぎる可能性もあります。ウェブサーバは、対象となるユーザのホームディレクトリ、および `public_html` 配下のファイル・ディレクトリの読み込みができなければなりません。適切な権限設定を行うための簡単な手順を以下に示します。

```
chmod 755 ~
find ~/public_html -type d -print0 | xargs -0r chmod 755
find ~/public_html -type f -print0 | xargs -0r chmod 644
```

権限設定に関する他の要因の可能性がある場合は、ブラウザでの所望の URL アクセス時に、完全に空の画面が表示されることでしょう。この場合は、おそらくアクセス権限が**緩すぎる**のでしょう。例えば Apache の `suexec` サブシステムは、group ないし other に書き込み権限が付与されたスクリプトは実行しません。

使用しているウェブサーバが、ユーザ毎ディレクトリ配下の CGI プログラムの実行を、禁止するように設定されている可能性も有ります。筆者の Fedora 7 システムにおける Apache の、初期状態のユーザ毎設定を以下に示します。

```
<Directory /home/*/public_html>
    AllowOverride FileInfo AuthConfig Limit
    Options MultiViews Indexes SymLinksIfOwnerMatch IncludesNoExec
    <Limit GET POST OPTIONS>
        Order allow,deny
        Allow from all
    </Limit>
    <LimitExcept GET POST OPTIONS>
        Order deny,allow
        Deny from all
    </LimitExcept>
</Directory>
```

対象となる Apache 設定ファイル中に似たような `Directory` 設定がある場合、`Options` 指定に注目してください。`ExecCGI` が指定されていない場合は一覧末尾にこれを追加し、ウェブサーバを再起動してください。

Apache が CGI を実行するのではなく、CGI スクリプトの内容そのものを返却してきた場合は、以下の記述を（既に記述があるならば）有効化するなり追加するなりしてください。

```
AddHandler cgi-script .cgi
```

次に問題の発生し得るケースでは、Python のバックトレースが表示され、`mercurial` 関連モジュールがインポート（import）できない旨を伝えていることでしょう。所望の結果は得られていませんが、ウェブサーバは CGI スクリプトの実行を行うようになったので、先程の状態からは前進しています！インポートができない旨のエラーは、システムワイドで利用可能な Mercurial ではなく、おそらく個人的にインストールした Mercurial を実行している場合にのみ発生します。ウェブサーバが CGI プログラムを実行する場合、各個人の対話的ログインセッションで実施されている環境変数指定が無い、ということを忘れないでください。このエラーが発生した場合は、`PYTHONPATH` 環境変数設定が適切になるように `hgweb.cgi` の記述を編集してください。

最終的に、/path/to/repository が見つからない旨を伝える Python のバックトレースが**確実に**表示されることでしょう。hgweb.cgi スクリプトを編集して、文字列 /path/to/repository を実際に公開したいリポジトリへの絶対パスで置き換えてください。

ここまで来れば、ウェブブラウザでページをリロードした際に、綺麗に HTML で整形されたリポジトリ履歴の表示を見ることができる筈です。お疲れ様です。

Configuring lighttpd

徹底的に実験するために、これまで Apache に関して説明したのと同様に、近年人気が高まっている lighttpd ウェブサーバで、同じリポジトリを公開するための設定記述に挑戦してみました。Apache についてこれまで概説してきた全ての問題は既に克服済みですし、その殆どはウェブサーバ実装に依存しません。結果として、ファイル・ディレクトリの権限設定が妥当であることと、hgweb.cgi スクリプトが適切に改変済みであることは、ある程度確信できます。

一旦 Apache での公開に成功していれば、lighttpd でのリポジトリ公開は簡単（言い換えるなら、lighttpd を使用する場合でも、前述の Apache に関する説明を読むべきと言えます）です。初期状態で mod_cgi および mod_userdir が無効化されていた場合、これらを有効化するために、まずは、設定ファイルの mod_access セクションを編集する必要があります。その後、これらのモジュールを設定するために、設定ファイル末尾に数行ほど追加します。

```
1 userdir.path = "public_html"
2 cgi.assign = ( ".cgi" => "" )
```

この記述により、lighttpd はユーザ毎のディレクトリおよび CGI を認識します。Apache よりも前に lighttpd の設定をしたとしたら、殆ど間違いなく、Apache の設定の際に経験したのと同じシステムレベルの設定ミスを犯したことでしょう。しかし Apache の使用経験が１０年以上あり、且つ初めての lighttpd 使用ではあるものの、Apache の設定よりも lighttpd のそれは著しく容易であると思われます。

6.6.3 Sharing multiple repositories with one CGI script

単一のリポジトリのみしか公開できないというのは、hgweb.cgi スクリプトの悩ましい制約です。同じスクリプト[11]を異なる名前で複製する、という面倒な方法よりは、hgwebdir.cgi スクリプトの使用がお薦めです。

hgwebdir.cgi の設定手順は、hgweb.cgi よりも多少込み入っています。まず始めにスクリプトのコピーを入手します。手近に無い場合は Mercurial のマスターリポジトリから http://www.selenic.com/repo/hg/raw-file/tip/hgwebdir.cgi を直接ダウンロードしてください。

public_html 配下に上記スクリプトを配置し、実行可能となるように権限設定を行います。

```
1 cp .../hgwebdir.cgi ~/public_html
2 chmod 755 ~/public_html ~/public_html/hgwebdir.cgi
```

基本的な設定が済んだなら、ブラウザで http://myhostname/~myuser/hgwebdir.cgi にアクセスしてみましょう。空のリポジトリリストが表示される筈です。何も表示されないか、エラーメッセージが表示される場合は、6.6.2 節で説明した潜在的問題一覧を一通り確認してください。

hgwebdir.cgi スクリプトは外部設定ファイルを必要とします。基底状態の hgwebdir.cgi スクリプトは、自身と同じディレクトリに格納された hgweb.config ファイルを読み込もうとします。このファイルを生成し、誰に対しても読み出し権限を付与しなければなりません。このファイルの記述形式は、Windows における “ini” ファイルのそれと同じで、Python の ConfigParser [Pyt] により解析可能な形式です。

最も簡単に hgwebdir.cgi を設定するには、collections という名前のセクションを設定してください。このセクションを記述することで、名付けたディレクトリ配下の**全ての**リポジトリを自動的に公開します。このセクションの記述は以下のようになります。

```
1 [collections]
2 /my/root = /my/root
```

[11]訳注: 厳密には、公開対象リポジトリのパスが異なるのですが、概ね「同じ」と言って良いでしょう。

Mercurial はこの記述を解釈するに当たり、“=” 記号の**右辺**に記述されたディレクトリ階層下でリポジトリを探し、“=” 記号の**左辺**のテキストに合致する部分を、ウェブインタフェースでの一覧表示で実際に公開される名前から除外します。除外処理の後に残ったパス要素は、“仮想パス” と呼ばれます。

例として/my/root/this/repo にリポジトリがあるとした場合、CGI スクリプトは冒頭の/my/root 部分を名前から除外し、仮想パスとして this/repo を持つリポジトリとして公開します。CGI スクリプトの基底 URL を http://myhostname/~myuser/hgwebdir.cgi とすると、このリポジトリの完全な URL は、http://myhostname/~myuser/hgwebdir.cgi/this/repo となります。

この設定記述例での左辺を /my/root から/my に変更した場合、hgwebdir.cgi はリポジトリ名から/my のみをｚｙ歩外するので、仮想パスは this/repo ではなく root/this/repo となります。

hgwebdir.cgi は、設定ファイル中の collections セクションで列挙された個々のディレクトリに対して、再帰的にリポジトリを探しますが、見つかったリポジトリから更に下への再帰的探索は行いません。

collections の機構は、多くのリポジトリを “fire and forget” 作法で公開するのに適しています。CGI や設定ファイルの記述は一度で事足ります。設定が済んだなら、hgwebdir.cgi に探索を指示したディレクトリ階層配下との間でリポジトリの移動を行うだけで、リポジトリの公開・非公開を任意の時点で行うことができます。

Explicitly specifying which repositories to publish

hgwebdir.cgi スクリプトは collections による公開の仕組みに加えて、特定の一覧指定によるリポジトリ公開をすることもできます。この方法での公開をするには、以下のような形式の内容を持つ paths セクションを記述する必要があります。

```
1 [paths]
2 repo1 = /my/path/to/some/repo
3 repo2 = /some/path/to/another
```

上記の例では、個々の定義の左辺が仮想パス（URL 中に現れるパス要素）、右辺がリポジトリへのパスとなります。仮想パスの指定と、ファイルシステム上のリポジトリ位置には、何の関連性も無い点に注意してください。

単一の設定ファイル中で collections と paths の両方を同時に使用することも可能です。

> **備考:** 同一の仮想パスに複数のリポジトリが関連付けられている場合、hgwebdir.cgi はエラーを通知しません。その代わりに、hgwebdir.cgi の振る舞いは予想できないものとなります。

6.6.4 Downloading source archives

Mercurial のウェブインタフェース経由で、任意のリビジョンのアーカイブをダウンロードすることが可能です。このアーカイブには、当該リビジョンにおける作業領域ディレクトリのスナップショットが格納されますが、リポジトリデータ部分は含まれません。

この機能は既定状態では無効化されています。この機能を有効化するには、allow_archive 項目を hgrc ファイルの [web] セクションに追加してください[12]。

6.6.5 Web configuration options

Mercurial のウェブインタフェース（“hg serve” コマンドおよび hgweb.cgi ないし hgwebdir.cgi スクリプト）には変更可能な設定項目が多数あります。これらの設定項目は [web] セクションに属しています。

allow_archive Mercurial のアーカイブダウンロード機能を有効化するか否かを指定。この機能を有効化した場合ウェブインタフェースの利用者は、リポジトリ中の参照可能な任意のリビジョンのアーカイブをダウンロードできます。この機能を有効化するには、以下に列挙されるキーワードの並びを allow_archive 項目に指定する必要があります。

[12]訳注: このことから、アーカイブダウンロードの有効化・無効化設定が、hgwebdir.cgi 単位ではなく、リポジトリ単位での設定であることがわかります。

bz2 bzip2 圧縮された tar アーカイブ形式。この形式は最も高い圧縮率を得られますが、サーバ側の CPU を最も酷使します。

gz gzip 圧縮された tar アーカイブ形式。

zip LZW 圧縮された zip アーカイブ形式。この形式は圧縮率が最も劣りますが、Windows 環境では広く使用されています。

値を指定しなかったり、allow_archive 項目そのものを指定しなかった場合、アーカイブダウンロード機能は無効化されます。利用可能な全てのアーカイブ形式を有効化する記述例を以下に示します。

```
[web]
allow_archive = bz2 gz zip
```

allowpull ウェブインタフェース経由での HTTP 越しの “hg pull” および “hg clone” を許可するか否かを指定する真偽値。no ないし false が指定された場合、ウェブインタフェースの “人間向け” 部分のみが有効化されます。

contact リポジトリの管理を行う人物・組織を特定するための任意の（但し極力簡潔な）文字列。通常この値は、管理者ないしメーリングリストの名前と電子メールアドレスです。多くの場合、この情報はリポジトリ毎の .hg/hgrc ファイルに記述しますが、全てのリポジトリが同一の保守担当により保守されている場合、大域的な hgrc ファイルに記述するのも良いでしょう。

maxchanges ページ毎に表示されるチェンジセットの最大数（既定値）を表す数値。

maxfiles ページ毎に表示される変更ファイルの最大数（既定値）を表す数値。

stripes テーブル表示における可読性向上のために、各行の色を互い違いに “縞模様” とする際に、何行毎に色を変更するかの数値。

style Mercurial がウェブインタフェースを表示する際に使用するテンプレート。Mercurial は default および gitweb の 2 つのウェブインタフェース用テンプレートを同梱しています（後者の方が見栄えが良いです）。自前でカスタマイズしたテンプレートを指定することもできます。詳細は 11 節を参照してください。gitweb スタイルの利用方法を以下に示します。

```
[web]
style = gitweb
```

templates テンプレートファイルの参照先ディレクトリを示すパス。Mercurial の既定値では、インストール先ディレクトリを参照します。

hgwebdir.cgi を使用する場合、幾つかの設定項目に関しては利便性上、hgrc ファイルに記述する代わりに、hgweb.config ファイルの [web] セクションに記述することができます。記述可能な設定項目は、motd および style です。

Options specific to an individual repository

ユーザ毎ないし大域的な hgrc ファイルではなく、リポジトリ毎の .hg/hgrc で記述すべき [web] セクションの設定項目が幾つかあります。

description リポジトリの内容ないし目的を記述した任意の（但し極力簡潔な）文字列。

name ウェブインタフェースにおけるリポジトリ参照名を示す文字列。この値は、リポジトリのパス[13]の末尾要素を用いた既定名を上書きします。

[13]訳注: 仮想パス？

Options specific to the “hg serve” command

hgrc ファイルの [web] セクションにおける設定項目の幾つかは、“hg serve” コマンド専用の項目です。

accesslog アクセスログを書き出すファイルのパス。“hg serve” コマンドの基底動作でのアクセスログ出力先は、ファイルではなく標準出力です。ログ要素は、多くのウェブサーバにおいて利用される標準的な“複合”（combined）ファイル形式で出力されます。

address 外部からの接続を受け付けるアドレスを指定する文字列。基底動作では、“hg serve” コマンドは全てのアドレスで接続を受け付けます。

errorlog エラーログを書き出すファイルのパス。“hg serve” コマンドの基底動作でのエラーログ出力先は、ファイルではなく標準エラー出力です。

ipv6 IPv6 プロトコル利用の有無を指定する真偽値。基底動作では IPv6 はサポートされません。

port “hg serve” コマンドが接続を受け付ける TCP ポートの番号を指定する数値。基底動作では、8000 番ポートが使用されます。

Choosing the right hgrc file to add [web] items to

Apache や lighttpd のようなウェブサーバは、リポジトリ所有者とは異なるユーザ権限で稼動する可能性がある、という点は重要ですので忘れないようにしてください。ウェブサーバによって起動される hgweb.cgi のような CGI スクリプトは通常、ウェブサーバと同一のユーザ権限で稼動します。

個人の hgrc ファイルに [web] セクションを記述しても、CGI スクリプトはその設定を読み込みません。個人の hgrc ファイルに記述した設定は、当該ユーザ自身で “hg serve” コマンドを実行した場合にのみ効力を発揮します。CGI スクリプトの挙動に所望の設定を反映するには、ウェブサーバが稼動される際のユーザのホームディレクトリに hgrc ファイルを作成して所望の設定を記述するか、あるいはシステムワイドな hgrc ファイルに所望の設定を追加してください。

第7章　File names and pattern matching

Mercurial は、一貫性と表現力を兼ね備えた方法でファイル名を扱う仕組みを提供しています。

7.1　Simple file naming

Mercurial は “under the hood” において、ファイル名を取り扱う統一された仕組みを用いています。ファイル名に関する全てのコマンドの挙動は統一されています。ファイル名に対するコマンドの挙動は、以下のようになっています。

コマンド行で実ファイル名を明示的に指定した場合、Mercurial は指定されたファイル名に厳密に作用します。

```
$ hg add COPYING README examples/simple.py
```

ディレクトリ名を指定した場合、Mercurial はその指定を、“当該ディレクトリならびにサブディレクトリ中の全てのファイル” とみなします。Mercurial は当該ディレクトリ配下のファイル・サブディレクトリを、アルファベット順に走査します。あるディレクトリの走査中にサブディレクトリに遭遇した場合、当該ディレクトリの走査よりも先に、サブディレクトリの走査を実施します[1]。

```
$ hg status src
? src/main.py
? src/watcher/_watcher.c
? src/watcher/watcher.py
? src/xyzzy.txt
```

7.2　Running commands without any file names

ファイル名を引数に取る Mercurial コマンドは、引数ないしパターン指定無しで起動された場合も、有用な基底時動作が定められています。コマンドに期待される振る舞いは、コマンドの用途に依存します。ファイル名指定無しの起動において、コマンドがどのように振舞うのかを推測するための、一般的な目安となる幾つかのルールを以下に示します。

- 殆どのコマンドは作業領域ディレクトリ全体に作用します。例えば、“`hg add`” コマンドなどがそうです。
- 復旧が困難あるいは不可能な作用を及ぼすコマンドの場合、少なくとも 1 つ以上の名前ないしパターン（後述します）の明示的な指定を求める筈です。この挙動により、例えば引数無しの “`hg remove`” 起動のような、不慮の事態によるファイルの削除等を防ぐことができます。

この振る舞いがそぐわない状況であれば、簡単に振る舞いを変えることができます。作業領域ディレクトリ全体に作用するコマンドであれば、“`.`” を指定することで、コマンドの作用を現在のディレクトリおよびその配下に限定することができます。

[1] 訳注: 深さ優先（depth first）

```
$ cd src
$ hg add -n
adding ../MANIFEST.in
adding ../examples/performant.py
adding ../setup.py
adding main.py
adding watcher/_watcher.c
adding watcher/watcher.py
adding xyzzy.txt
$ hg add -n .
adding main.py
adding watcher/_watcher.c
adding watcher/watcher.py
adding xyzzy.txt
```

ルート以外のディレクトリでコマンドを実行した場合でも、リポジトリのルートに対する相対的なファイル名を表示するコマンドもあります。このようなコマンドは、明示的な名前を指定することで、現在のディレクトリ位置に対する相対的なファイル名を表示するようになります。非ルートディレクトリでの “hg status” 起動の際に “hg root” コマンドの出力を指定することで、対象を作業領域ディレクトリ全体に維持したまま、現在のディレクトリ位置に対する相対的なファイル名を表示させることができます。

```
$ hg status
A COPYING
A README
A examples/simple.py
? MANIFEST.in
? examples/performant.py
? setup.py
? src/main.py
? src/watcher/_watcher.c
? src/watcher/watcher.py
? src/xyzzy.txt
$ hg status `hg root`
A ../COPYING
A ../README
A ../examples/simple.py
? ../MANIFEST.in
? ../examples/performant.py
? ../setup.py
? main.py
? watcher/_watcher.c
? watcher/watcher.py
? xyzzy.txt
```

7.3 Telling you what’s going on

先の節における “hg add” コマンド実行例は、Mercurial コマンドに関するもう一つの有益な事柄を示しています。コマンド行で明示的な指定をしていないファイルに対してコマンドが作用する場合、通常は対象ファイル名を表示しますので、思わぬコマンドの実行結果に後から驚かされることはありません。

これは**驚きを最小**にする原則に則ったものです。コマンド行で厳密なファイル名を指定した場合には、それを復唱する必要は無いでしょう。ファイル名・ディレクトリ名ないしパターン（後述します）を指定しないことで**暗に指定された**対象ファイルに Mercurial が作用する場合、どのファイルを対象とするのかを通知するのは安全性の上で有用です。

上記方針に沿って振舞うコマンド群は、`-q` オプションを指定することで、その出力を抑止することができます。明示的にファイル名等を指定した場合でも、`-v` オプションを指定することで、全ての対象ファイル名を表示させることができます。

7.4 Using patterns to identify files

ファイル名・ディレクトリ名による指定に加えて、Mercurial では**パターン**によるファイル指定機能が使用できます。Mercurial のパターン操作は表現力に富んだものです。

Linux や MacOS のような Unix 的システムでは、ファイル名とパターンとの間の突合せは通常シェルがその役目を負います。これらのシステムでは、パターンを指定している旨を Mercurial に対して明示的に指示する必要があります[2]。Windows においては、シェルによるパターンの展開が行われませんので、Mercurial は自動的に指定されたものがパターンであると認識し、ファイル名へと展開します。

コマンド行において、ファイル名を指定する場所でパターンを使用するには、以下のように記述します。

```
syntax:patternbody
```

パターンの記述は、パターンの種類を識別するための短い文字列、コロン、そして実際のパターンを連結したものです。

Mercurial は２種類のパターン形式に対応しています。最も利用頻度が高いものは `glob` と呼ばれ、Unix のシェルによるパターンマッチングと同様の機能を持つもので、その振る舞いは Windows のコマンドプロンプトユーザにも馴染みがあることでしょう。

Windows において Mercurial が自動的にパターンマッチングを行う場合、`glob` 形式とみなされます。そのため、Windows においては “`glob:`” 接頭辞を省略可能ですが、明示的に指定することも可能です。

`re` 形式は、`glob` 形式よりも強力で、regexps としても知られる正規表現を使用したパターンの記述が可能です。

ちなみに、以降の例では、全てのパターン指定を注意深く引用符で囲むことで、Mercurial の処理の前にシェルによって展開されてしまうことを防いでいる、という点に注意してください。

7.4.1 Shell-style `glob` patterns

`glob` 形式によるマッチングの際に、使用可能なパターンについての概要を以下に示します。

パターン “`*`” は、同一ディレクトリ内で任意の文字列に合致します。

```
$ hg add 'glob:*.py'
adding main.py
```

パターン “`**`” は、ディレクトリ境界を超えて任意の文字列に合致します。このパターンは Unix における標準的なものではありませんが、幾つかの著名なシェル実装で採用されており、非常に便利です。

```
$ cd ..
$ hg status 'glob:**.py'
A examples/simple.py
A src/main.py
? examples/performant.py
? setup.py
? src/watcher/watcher.py
```

パターン “`?`” は、単一の文字に合致します。

[2]訳注: シェルによる特殊文字展開の抑止の話であれば、“Mercurial に対して” ではなく、“シェルに対して” なのでは？それとも Windows バイナリ版では振る舞いが異なる？

```
$ hg status 'glob:**.?'
? src/watcher/_watcher.c
```

パターン "[" は、**文字集合**（character class）の開始を意味します。このパターンは当該集合に属する任意の一文字に合致します。集合指定は "]" によって終了します。集合指定には、"abcdef" の省略指定である "a-f" 形式の**範囲**指定を、複数含めることが可能です。

```
$ hg status 'glob:**[nr-t]'
? MANIFEST.in
? src/xyzzy.txt
```

文字集合指定において "[" の直後の文字が "!" [3]の場合、集合指定は**反転**され、集合に属さない任意の一文字に合致します。

パターン "{" はサブパターンのグループ化の開始を意味し、グループ中の何れかのサブパターンが合致した場合は、グループ全体が合致したものとみなされます。グループ指定におけるサブパターンの区切りには "," が使用され、"}" がグループの終了を意味します。

```
$ hg status 'glob:*.{in,py}'
? MANIFEST.in
? setup.py
```

Watch out!

任意のディレクトリにおけるパターン合致が必要な場合は、単一ディレクトリ内でのマッチングしか行わない "*" を使用すべきでは無い、という点は忘れないようにしてください。"*" の代わりに "**" を使用しましょう。両者の違いを以下で説明します。

```
$ hg status 'glob:*.py'
? setup.py
$ hg status 'glob:**.py'
A examples/simple.py
A src/main.py
? examples/performant.py
? setup.py
? src/watcher/watcher.py
```

7.4.2 Regular expression matching with `re` patterns

Mercurial は（Python の内部的な正規表現エンジンを利用しているので）Python が受け付けるのと同じ正規表現を受け付けます。この正規表現は Perl の正規表現文法を基にしており、最も多用されている（例えば Java でも使用されています）方言です。

正規表現パターンはそれほど多用されるものではないので、Mercurial の正規表現の詳細に関してここでは説明しません。Perl 形式の正規表現は様々な形式で、多くのウェブサイトや出版物において余す所無く説明されています。その代わりここでは、Mercurial で正規表現を使用する必要に迫られた際に、知っておくべき幾つかの事柄について説明しようとおもいます。

[3]訳注: 正規表現における "^" による反転と異なる点に注意

正規表現は、リポジトリルートからの相対的なファイル名全体に対して適用されます。言い換えるなら、`foo` サブディレクトリで作業している場合でも、このディレクトリ配下のファイルに対してマッチングを行うなら、指定するパターンは “`foo/`” で始まっていなければなりません。

Perl 形式の正規表現に馴染んでいる場合、Mercurial の正規表現は *rooted* である点に注意してください[4]。正規表現は文字列先頭からマッチングを実施しますので、文字列途中に対するマッチングは行われません。任意の位置に対してマッチングを実施させたい場合、パターンの記述を “`.*`” で始める必要があります。

7.5 Filtering files

Mercurial が多様な方法を提供しているものは、ファイルの指定方法だけではありません。Mercurial は**フィルタ**によるファイル選別の機能も提供しています。ファイル名指定を受け付けるコマンドは、以下の２つのフィルタリングオプションも受け付けます。

- `-I` ないし `--include` により、合致したファイルのみを処理対象とみなすパターンを指定できます。
- `-X` ないし `--exclude` により、合致したファイルを処理対象から**除外**するパターンを指定できます。

複数の `-I` および `-X` オプションを、コマンド行で好きなように混在させることができます。Mercurial の基底動は、指定されたパターンを “`glob`” 形式とみなして解釈します（必要であれば明示的に “`glob`” を指定することも可能です）。

`-I` フィルタは、“合致したファイルのみを処理対象とする” ものと解釈すれば良いでしょう。

```
$ hg status -I '*.in'
? MANIFEST.in
```

`-X` フィルタは、“合致しないものを処理対象とする” ものと解釈することができます。

```
$ hg status -X '**.py' src
? src/watcher/_watcher.c
? src/xyzzy.txt
```

7.6 Ignoring unwanted files and directories

※ 原文未稿

7.7 Case sensitivity

Linux（ないし他の Unix 系 OS）と、MacOS ないし Windows が混在する開発環境で作業する場合、ファイル名における文字の大小（“N” と “n”）の扱い方針が全く異なる、という知識を心に留めておく必要があります。良くある事では無いかもしれませんし、容易に解決できる可能性もありますが、知らない状況で遭遇した場合、非常に驚かされる問題でもあります。

OS およびファイルシステムに応じて、ファイルおよびディレクトリ名の**文字の大小**の扱いは異なります。名前における文字の大小の一般的な扱い方を、以下に３つ示します。

- 完全に文字の大小を無視: ファイルの生成およびその後の扱いにおいて、文字の大文字・小文字は同じものとして扱われます。古い DOS 風のシステムで一般的な扱い方です。

[4] 訳注: 暗黙のうちに “`^`” が付与される、と理解すれば良いでしょう。

- 文字の大小は保持されるが無視: ファイルないしディレクトリ生成の際には、名前における文字の大小は保存され、OS による検索や表示が可能です。存在するファイルが検索される場合、文字の大小は無視されます。Windows や MacOS では標準的な仕様です。`foo` と `FoO` は同じファイルとみなされます。大文字と小文字の互換性ある扱いは、**ケースフォールディング**（case folding）とも呼ばれます。
- 文字の大小を区別: 名前における文字の大小は常に意味を持ちます。`foo` と `FoO` は異なるファイルとして区別されます。これは Linux や Unix における通常の振る舞いです。

Unix 的なシステムの上では、上記の大文字・小文字の取り扱い形式のうちの “任意” のものが（あるいは全てが同時に）要求される可能性があります。例えば、FAT32 ファイルシステムでフォーマットされた USB 小型メモリモジュールを Linux で使用する場合、そのファイルシステム上での Linux の振る舞いは、文字の大小は保持しつつ無視するものとなります。

7.7.1 Safe, portable repository storage

Mercurial のリポジトリ格納機能は、文字大小の区別の可否に**影響を受けません**。リポジトリの保存先ファイル名は元ファイル名を変換したものなので、ファイルシステムにおける大文字小文字の区別の可否に関わり無く、構成管理情報を格納できます。つまり、OS の標準的な複製ツールを使用して、Mercurial のリポジトリを例えば USB 小型メモリモジュールに複製し、Mac、Windows PC および Linux の間で持ち運ぶことができます。

7.7.2 Detecting case conflicts

作業領域ディレクトリにおける操作の際には、Mercurial は作業領域を載せているファイルシステムの命名方針に従います。ファイルシステムが文字の大小は保持しつつ無視するものであった場合、文字の大小のみが異なる名前を Mercurial は同じものとみなします。

この方針の重要な点は、文字大小を区別する（一般的な Linux や Unix における）ファイルシステムにおいて、文字大小を区別できない（Windows や MacOS の）ユーザが取り扱えないようなチェンジセットをコミットすることが可能である点です。Linux の利用者が `myfile.c` と `MyFile.C` という名前の２つのファイルに対する変更をコミットした場合、変更内容はリポジトリに正しく保存されます。他の Linux 利用者の作業領域ディレクトリにおいても、これらのファイルは異なるファイルとして正しく存在します。

Mercurial のリポジトリ格納機構が文字大小の扱いの可否に影響を受けないため、Windows ないし MacOS 利用者がこの変更を取り込んでも、最初は問題が発生しません。しかし、作業領域ディレクトリを当該チェンジセットで “`hg update`” しようとした場合、あるいは当該チェンジセットと “`hg merge`” しようとした場合、ファイルシステムが同じファイルとして扱う２つのファイルの衝突を見つけた Mercurial によって、“`hg update`” ないし “`hg merge`” は禁止されます。

7.7.3 Fixing a case conflict

他のメンバーが Linux や Unix を使用している混在環境で Windows ないし MacOS を使用していて、“`hg update`” あるいは “`hg merge`” の際に Mercurial が文字大小の衝突を報告する場合、問題の解決手順は簡単です。

手近な Linux ないし Unix 利用者を探し、問題のリポジトリを “`hg clone`” してから、問題のファイルないしディレクトリを大文字小文字の衝突が発生しないように、Mercurial の “`hg rename`” コマンドで改名をすれば良いのです。その後、変更をコミットし、“`hg pull`” ないし “`hg push`” で Windows や MacOS に変更を取り込み、“`hg update`” によって衝突しない名前で変更内容を取り出します。

大文字小文字の衝突を生じさせるチェンジセットそのものは、プロジェクトの履歴に残っており、当該チェンジセットを Windows や MacOS 上で作業領域ディレクトリに取り出すことはできませんが、開発を継続することは可能です[5]。

[5]訳注: 文字の大小とは関係ありませんが、Windows は “`con`” や “`aux`” が特別扱いされるため、例えばこれらの名前を利用したディレクトリがある場合などは、リポジトリの “`hg pull`” そのものができません。

備考: 0.9.3 版以前の Mercurial は、大文字小文字に影響を受けないリポジトリ格納機構も、大文字小文字の名前衝突検知機能もありませんでした。Mercurial の旧版を Windows や MacOS で使用している場合、Mercurial の更新をお薦めします。

第8章　Managing releases and branchy development

Mercurial は、同時並行的に開発を進めるようなプロジェクトを管理できる仕組みを持っています。これらの仕組みを理解するために、まずは一般的なソフトウェア開発の仕組みを眺めてみましょう。

多くのソフトウェアプロジェクトでは、重要な新規機能を含む “メジャー” リリースを間欠的に発行します。それと平行して “マイナー” リリースも発行することがあります。多くの場合、マイナーリリースは元にしたメジャーリリースと同一ですが、バグの修正がなされています。

この章では、「リリース」のようなプロジェクトのマイルストーンの、記録を保持する方法から説明を始めたいと思います。その後で、プロジェクトにおけるフェーズ移行での作業の流れや、その際の作業や成果物を Mercurial によって分離／管理する方法を説明します。

8.1　Giving a persistent name to a revision

特定のリビジョンを “リリース” と呼ぶことに決定したなら、そのリビジョンの ID を記録するべきです。リビジョンの ID を記録することで、後日何らかの理由（例えばバグの再現や、新規プラットフォームへの移植等）で必要になった際にリリースを再現することができます。

```
$ hg init mytag
$ cd mytag
$ echo hello > myfile
$ hg commit -A -m 'Initial commit'
adding myfile
```

“hg tag” コマンドを利用することで、Mercurial は任意のリビジョンに永続的な名前を付与します。読者の予想通り、この名前のことを “タグ” と呼びます。

```
$ hg tag v1.0
```

リビジョンにとって、タグは “象徴的な名前”(symbolic name) 以外の何者でもありません。タグは純粋に利便性のために存在するもので、リビジョンを参照する際の手軽で永続的な手段となります。Mercurial は、利用者の用いるタグ名の意味を解釈したりしません。曖昧さが無く解析できることを保証するために必要な少々の制約を除いては、タグ名に何らかの制約をつけたりすることもありません。以下のいずれの文字もタグ名には使用できません。

- コロン (ASCII 58, “:”)
- 行頭移動[1] (ASCII 13, “\r”)
- 改行 (ASCII 10, “\n”)

“hg tags” コマンドを使用することで、リポジトリが保持しているタグを表示させることができます。“hg tags” コマンドの出力において、個々のタグ付けされたリビジョンは、始めにタグ名で、次にリビジョン番号で、最後に一意のリビジョンハッシュ値で識別されます。

[1] carriage return

```
$ hg tags
tip                                1:8997f6fa7740
v1.0                               0:d9f38a9cee5b
```

tip タグが “hg tags” コマンドの出力に列挙されていることに注意してください。tip は、常にリポジトリ中の最新のリビジョンを指す “流動的な” 特殊タグです。

“hg tags” コマンドの出力では、タグはリビジョン番号の逆順（降順）で列挙されています。これは最新のタグは古いタグよりも先に列挙されることを意味し、それは同時に “hg tags” が出力するタグ一覧の最初に tip が表示されることも意味します。

“hg log” コマンドの実行時に、タグと関連付けられたリビジョンを表示する場合、“hg log” コマンドはタグを表示します。

```
$ hg log
changeset:   1:8997f6fa7740
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:56 2009 +0000
summary:     Added tag v1.0 for changeset d9f38a9cee5b

changeset:   0:d9f38a9cee5b
tag:         v1.0
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:56 2009 +0000
summary:     Initial commit

```

Mercurial コマンドに対してリビジョン識別子を指定する必要がある場合、リビジョン識別子を指定する位置では、常にタグ名を使用することができます。Mercurial の内部では、タグ名を対応するリビジョン識別子に変換してから使用しています。

```
$ echo goodbye > myfile2
$ hg commit -A -m 'Second commit'
adding myfile2
$ hg log -r v1.0
changeset:   0:d9f38a9cee5b
tag:         v1.0
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:56 2009 +0000
summary:     Initial commit

```

単一のリポジトリが保持できるタグの数にも、単一のリビジョンに付与できるタグの数にも制限はありません。現実的な問題として、タグは単にリビジョンの特定を補助するものですから、“過剰に”（具体的な数はプロジェクトに応じて異なりますが）タグを付与するのはよろしくありません。多くのタグがあると、リビジョンを特定する利便性が早々に減少してしまいます。

例えば、あるプロジェクトでは数日毎の頻度でマイルストーンを設定しているとすると、それぞれのマイルストーンにタグを付与するのは極めて合理的です。しかし、全てのリビジョンで確実に綺麗なビルドができる継続的（continuous）なビルドシステムがある場合は、綺麗なビルド毎にタグを付与すると、大量のノイズを持ち込むことになります。その代わりに、ビルドが失敗するリビジョン（この事態が稀だと仮定しています！）にタグを付与するか、ビルドの可否を追跡するタグの使用を止めるのが良いでしょう。

必要の無くなったタグを削除したい場合は “hg tag --remove” コマンドを使用します。

```
$ hg tag --remove v1.0
$ hg tags
tip                                3:b353803674a7
```

任意の時点でタグの関連付けを変更することもできますので、新規の “hg tag” コマンド実行により、同一のタグが異なるリビジョンを識別するようになります。**本当に**タグを更新したいことを Mercurial に伝えるために、-f オプションを使用しなければなりません。

```
$ hg tag -r 1 v1.1
$ hg tags
tip                                4:8c84a68a62fa
v1.1                               1:8997f6fa7740
$ hg tag -r 2 v1.1
abort: tag 'v1.1' already exists (use -f to force)
$ hg tag -f -r 2 v1.1
$ hg tags
tip                                5:d242de1da2c9
v1.1                               2:6acb1fe9afed
```

タグの更新後も、タグが以前に識別していたリビジョンに関する永続的な記録が残りますが、Mercurial がそれを使用することはありません。このように、間違ったリビジョンへのタグの付与には何の不利益もありませんので、タグ付けを間違ったなら、正しいリビジョンにタグを付与し直せばよいのです。

Mercurial は、リポジトリ中のリビジョン管理された通常ファイルにタグの情報を格納しています。何らかのタグを付与すると、.hgtags ファイル中にそのタグを見つけることができるでしょう。“hg tag” コマンドを実行すると、Mercurial はこのファイルを変更し、自動的に変更をコミットします。このことは、“hg tag” コマンドを実行した際には、常に対応するチェンジセットを “hg log” コマンドの出力で見ることができる、ということを意味しています。

```
$ hg tip
changeset:   5:d242de1da2c9
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:57 2009 +0000
summary:     Added tag v1.1 for changeset 6acb1fe9afed

```

8.1.1 Handling tag conflicts during a merge

.hgtags ファイルを気にする必要は殆どありませんが、時にはマージの際にその存在が意識されることがあります。このファイルの形式は単純で、連続した行から構成されています。各行はチェンジセットのハッシュ値で始まり、空白とタグ名が続きます。

マージにおける.hgtags ファイルの衝突を解消する際には、.hgtags ファイル修正にひねりが必要です。リポジトリ中のタグを解析する場合、Mercurial は**決して**.hgtags ファイルのワーキングコピーを参照することはありません。その代わりに、Mercurial は**最も最近コミットされた**ファイルのリビジョンを調べます。

このような設計の残念な結果として、マージした .hgtags ファイルが、その変更をコミットした**後も**正しい状態であることを、実際に検証することができません。マージの際に .hgtags ファイルの衝突を解消する際には、コミット後に “hg tags” コマンドの実行を忘れずに行ってください。.hgtags ファイルに不正があった場合、“hg tags” コマンドは不正の場所を報告しますので、その箇所を修正してコミットすれば良いのです。変更内容の正しさを確認するために、変更の後で、再度 “hg tags” コマンドを実行してください。

8.1.2 Tags and cloning

“hg clone” コマンドが特定のチェンジセットを指定して厳密な複製を作成するための-r オプションを持っていることに気付いているかもしれません。新しい複製は、指定したリビジョンよりも後に生じた履歴情報を一切持っていません。このことがタグと相互作用した場合、、油断していると驚かされる事態になります。

タグの生成が、.hgtags ファイルへの格納の際に、一つのリビジョンとして扱われることを思い出せば、タグが記録されたチェンジセットが、タグの付与対象となる（古い）チェンジセットを参照するのは当然のことです。タグ foo 時点のリポジトリを複製するために “hg clone -r foo” を実行した場合、複製されたリポジトリは、複製する際に使用された**タグの作成に関する履歴を持っていません**。新しいリポジトリには、プロジェクト履歴の完全なサブセットが含まれますが、唯一、指定に用いたタグの情報は**含まれていません**。

8.1.3 When permanent tags are too much

Mercurial のタグは構成管理されており、プロジェクトの履歴と一体化しているため、誰かが作成したタグは、一緒に作業を行っている誰もが見ることができます。しかし、リビジョンに名前を付けることは、リビジョン 4237e45506ee が実は v2.0.2 である、ということを書き留めておく以上の有用性があります。巧妙なバグを追跡する際に、“アンがこのリビジョンで症状を見かけた” といった類の備忘録として、タグを付与したい場合もあるでしょう。

このような場合、**ローカル**なタグが最適です。-l オプション付きで “hg tag” コマンドを起動することで、ローカルタグを作成することができます。このコマンド実行の場合、タグは .hg/localtags ファイルに格納されます.hgtags と異なり.hg/localtags は構成管理されません。-l によって作成したタグは、現在作業をしているリポジトリに留まり続けます[2]。

8.2 The flow of changes—big picture vs. little

ここで、本章の冒頭で述べた概略に戻り、複数の平行した開発が同時に行われているプロジェクトについて考えて見ましょう。

新しい “主” リリースや、最新の主リリースに対する新たなマイナーバグ修正、現在は保守状態にあるような古いリリースに対する予期せぬ “hot fix” のための push があるでしょう。

開発における様々な平行した方向を参照するための一般的な方法は、“ブランチ” と呼ばれるものです。しかし、Mercurial が**全ての履歴**を「ブランチとマージの連続」として扱っていることを、既に何度も見てきました。実際には、表面的には関係しているようで、その実、たまたま同じ名前であるだけの２つの概念を扱っているのです。

- “巨視的な” ブランチは、プロジェクト発展の広がりを表し、名前をつけたり、話題に上ったりします。
- “微視的な” ブランチは、日々の開発活動と、変更マージの成果です。このブランチは、コードがどのように開発されていったのかを物語ります。

8.3 Managing big-picture branches in repositories

Mercurial において “巨視的な” ブランチを隔離する最も簡単な方法は、隔離用のリポジトリを用意することです。例えば、既にある共有リポジトリ—これを myproject と呼称します—が “1.0” というマイルストーンに到達している場合、1.0 リリースのために使用したリビジョンにタグを付与することで、1.0 版に対する来るべき保守リリースの準備を行います。

```
$ cd myproject
$ hg tag v1.0
```

タグ付けした時点と同じ内容の myproject-1.0.1 という名の新しい共有リポジトリを複製します。

[2]訳注: “hg clone”、“hg pull” や “hg push” によって他のリポジトリにコピーされることがありません

```
$ cd ..
$ hg clone myproject myproject-1.0.1
updating working directory
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

その後、来る 1.0.1 マイナーリリースに含めるべきバグ修正の作業が必要になったなら、myproject-1.0.1 リポジトリを複製し変更を行って、その成果を反映します。

```
$ hg clone myproject-1.0.1 my-1.0.1-bugfix
updating working directory
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd my-1.0.1-bugfix
$ echo 'I fixed a bug using only echo!' >> myfile
$ hg commit -m 'Important fix for 1.0.1'
$ hg push
pushing to /tmp/branch-repolpQVYV/myproject-1.0.1
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

その間、次のメジャーリリースへ向けた開発作業は、マイナーリリースに関する作業とは隔離された状態で、myproject リポジトリにおいて活発に続けられます。

```
$ cd ..
$ hg clone myproject my-feature
updating working directory
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd my-feature
$ echo 'This sure is an exciting new feature!' > mynewfile
$ hg commit -A -m 'New feature'
adding mynewfile
$ hg push
pushing to /tmp/branch-repolpQVYV/myproject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

8.4 Don't repeat yourself: merging across branches

保守用ブランチでバグ修正を行ったとすると、多くの場合、プロジェクトのメインブランチに（そしてそれ以外の保守ブランチにおいても）同じバグが存在する可能性があります。同じバグを何度も直したいと思う開発者は稀ですから、同じ作業を繰り返すことなくバグ修正を管理するために Mercurial が提供する幾つかの方法を見てみましょう。

最も単純な方法は、作業対象ブランチから複製したローカルリポジトリへ、保守ブランチから変更を pull することです。

```
$ cd ..
$ hg clone myproject myproject-merge
```

```
updating working directory
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg pull ../myproject-1.0.1
pulling from ../myproject-1.0.1
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

その上で２つのブランチのそれぞれのヘッドをマージし、その成果をメインブランチに反映します。

```
$ hg merge
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg commit -m 'Merge bugfix from 1.0.1 branch'
$ hg push
pushing to /tmp/branch-repolpQVYV/myproject
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 1 changes to 1 files
```

8.5 Naming branches within one repository

多くの場合は、リポジトリの分離によってブランチを分離するのが適切な遣り方です。単純ですから理解も簡単ですし、それ故に間違えることがありません。作業しているブランチと、コンピュータ上の（リポジトリ）ディレクトリの間で、１対１の関係ができていますので、ブランチ／リポジトリ中のファイルに対して、（Mercurial を意識しない）通常のツールを使用することもできます。

あなたが（そして共同作業者も）“パワーユーザー” よりも高いレベルにあるのであれば、ブランチ (that you can consider XXXX) を扱う別な方法があります。前の節では、“微視的” ブランチと “巨視的” ブランチの、利用者レベルでの区別について言及しました。単一のリポジトリ中で、常に複数の “微視的な” ブランチ（例えば、変更の pull 後にマージしていない状態）を扱っている一方で、Mercurial は複数の “巨視的な” ブランチを扱うこと**も**できます。

Mercurial が “巨視的な” ブランチを扱う際の要点は、ブランチに永続的な**名前**を付けるところにあります。前述のように `default` という名前のブランチが常に存在しますので、ブランチへの命名を行う前であっても、探せば `default` ブランチの跡を見つけることができます。

例えば、“`hg commit`” コマンドを実行すると、エディタが起動されてコミットメッセージを入力できます[3]が、末尾の “`HG: branch default`” を含む行を見てください。これは、`default` という名前のブランチに対してコミットしている、ということを表しています。

ブランチに名前をつけるには、まずは “`hg branches`” を使用します。このコマンドは、リポジトリ中に既に存在する名前付きブランチと、個々のブランチにおける先頭（tip）リビジョンがどれかを列挙します。

```
$ hg tip
changeset:   0:893159f5cd87
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:30 2009 +0000
summary:     Initial commit
```

[3]訳注: Emacs の hg-mode.el を使用している場合は見られません

```
$ hg branches
default                        0:893159f5cd87
```

実行例では、名前付きブランチを生成する前ですから、唯一存在する default だけが表示されます。

どれが “現在の” ブランチかを知るには、引数無しで “hg branch” コマンドを実行します。このコマンドは、現在のチェンジセットの親チェンジセットが、どのブランチ上にあるものかを表示します。

```
$ hg branch
default
```

新しいブランチを作成するには、再度 “hg branch” コマンドを実行しますが、今回は生成するブランチ名を引数として指定します。

```
$ hg branch foo
marked working directory as branch foo
$ hg branch
foo
```

ブランチ生成後、“hg branch” コマンドによりどのような副作用を生じたのか、怪しむかもしれません。“hg status” や “hg tip” の出力はどうなっているでしょうか？

```
$ hg status
$ hg tip
changeset:   0:893159f5cd87
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:30 2009 +0000
summary:     Initial commit

```

作業領域に変更は加えられていませんし、履歴に変化もありません。このことが示唆しているように、“hg branch” コマンドの実行は何ら永続的な効果を持ちません。このコマンドは、**次回の**チェンジセットのコミットの際に、何というブランチ名を使用するかを Mercurial に伝えるだけです。

変更をコミットすると、Mercurial はコミットされたチェンジセットにブランチ名を記録します。一旦 default ブランチから他のブランチに切り替えてコミットしたなら、“hg log”、“hg tip” やそれに類する出力を持つコマンドの出力に、新たなブランチ名が表示されていることでしょう。

```
$ echo 'hello again' >> myfile
$ hg commit -m 'Second commit'
$ hg tip
changeset:   1:9a664847a614
branch:      foo
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:31 2009 +0000
summary:     Second commit

```

“hg log” に類するコマンドは、default ブランチ以外に属する全てのチェンジセットに対して、ブランチ名を表示します。そのため、名前付きブランチを使わない限り、ブランチに関する情報を見ることはありません。

名前付きブランチを作成し、そのブランチ名で変更をコミットしたならば、その変更に連なるその後のコミットは、同じブランチ名を引き継ぎます。“hg branch” コマンドにより、任意の時点でブランチ名を変更することができます。

```
$ hg branch
foo
$ hg branch bar
marked working directory as branch bar
$ echo new file > newfile
$ hg commit -A -m 'Third commit'
adding newfile
$ hg tip
changeset:   2:5417f05f786a
branch:      bar
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:31 2009 +0000
summary:     Third commit

```

ブランチ名はかなり長い寿命を持つため、実際にはこのようなブランチ名の変更はそれほど頻繁に実行することは無いでしょう（このことは規約ではなく、あくまで感想です）。

8.6 Dealing with multiple named branches in a repository

リポジトリに複数の名前付きブランチがある場合、“hg update” や “hg pull -u” といったコマンド実行の際に、Mercurial は作業領域ディレクトリが属するブランチを覚えていて、“リポジトリ全体” の tip リビジョンではなく、そのブランチの tip リビジョンで作業領域ディレクトリを更新します。別な名前付きブランチのリビジョンで更新したい場合は、“hg update” コマンドに-C オプションを指定しなければなりません。

この振る舞いは少々微妙ですから、実例で見てみましょう。始めに、どのブランチ上で作業しているのかと、どんなブランチがリポジトリ中に有るのかを確認します。

```
$ hg parents
changeset:   2:5417f05f786a
branch:      bar
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:31 2009 +0000
summary:     Third commit

$ hg branches
bar                            2:5417f05f786a
foo                            1:9a664847a614 (inactive)
default                        0:893159f5cd87 (inactive)
```

現在 bar ブランチ上にいますが、古い “hg foo” ブランチも存在します。

foo ブランチおよび bar ブランチの tip リビジョンへの移動は、変更履歴上を直線的に前後することしか必要としないため、“hg update” コマンドに-C オプションを指定すること無しに、それぞれの tip リビジョンへの更新を行うことができます。

```
$ hg update foo
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
```

```
$ hg parents
changeset:   1:9a664847a614
branch:      foo
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:31 2009 +0000
summary:     Second commit

$ hg update bar
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg parents
changeset:   2:5417f05f786a
branch:      bar
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:31 2009 +0000
summary:     Third commit

```

foo ブランチに戻るために "hg update" コマンドを実行すると、foo ブランチ上に留まったままで bar ブランチの tip リビジョンには移動しません。

```
$ hg update foo
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg update
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

foo ブランチでの変更のコミットにより、新たなヘッドが生成されます。

```
$ echo something > somefile
$ hg commit -A -m 'New file'
adding somefile
created new head
$ hg heads
changeset:   3:1c0fb9d007f6
branch:      foo
tag:         tip
parent:      1:9a664847a614
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:32 2009 +0000
summary:     New file

changeset:   2:5417f05f786a
branch:      bar
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:31 2009 +0000
summary:     Third commit

```

foo ブランチから bar ブランチへの更新は、履歴を "横っ飛び" しないとできませんから、Mercurial は "hg update" コマンドへの -C オプションの指定を必要とします。

```
$ hg update bar
1 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg update -C bar
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

8.7 Branch names and merging

お気づきの事とは思いますが、Mercurial におけるマージ処理は対称的ではありません。リビジョン番号 17 のものと 23 のもの、２つのヘッドをリポジトリが持っているものとしましょう。リビジョン 17 へと “hg update” してからリビジョン 23 と “hg merge” した場合、Mercurial はリビジョン 17 をマージの第１親、リビジョン 23 を第２親として記録します。一方で、リビジョン 23 へと “hg update” してからリビジョン 17 と “hg merge” した場合、リビジョン 23 がマージの第１親、リビジョン 17 が第２親として記録されます。

この振る舞いが、マージを行った際の Mercurial のブランチ名選択に影響します。マージ後にその結果をコミットすると、Mercurial は第１親のブランチ名を維持しようとします。第１親のブランチ名が foo で、bar ブランチのリビジョンとマージした場合、マージ後のブランチ名は foo のままとなります。

リポジトリ中に同じブランチ名の複数のヘッドが存在することは、それほど珍しいことではありません。例えば、私とあなたが foo ブランチで作業しているとします。二人がそれぞれ異なる変更をコミットし、私があなたの変更を pull しました。この時点で私のリポジトリには、foo ブランチ上に２つのヘッドが存在します。マージの結果、foo ブランチ上の２つのヘッドは期待通り１つになります。

しかし、私が bar ブランチで作業していて、foo ブランチの成果をマージした場合、マージの結果は bar ブランチ上に留まります。

```
$ hg branch
bar
$ hg merge
abort: branch 'bar' has one head - please merge with an explicit rev
$ hg commit -m 'Merge'
nothing changed
$ hg tip
changeset:   3:1c0fb9d007f6
branch:      foo
tag:         tip
parent:      1:9a664847a614
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:32 2009 +0000
summary:     New file

```

より具体的な例として、bleeding-edge ブランチで作業していて、最新の成果を stable ブランチから持ち込みたいと思ったとします。この場合、stable ブランチの成果を pull してマージした段階で、Mercurial は “適切な” ブランチ名 (bleeding-edge) を選択します。

8.8 Branch naming is generally useful

寿命の長い複数のブランチが単一リポジトリで共存している状況だけが、名前付きブランチの利用できる状況だとは考えないでください。リポジトリ１つにブランチ１つの状況であっても、名前付きブランチは有用です。

単純な例としては、ブランチに名前を付与することで、チェンジセットがどのブランチに由来するかの恒久的な記録を得ることができます。この記録は、寿命の長いブランチを持つプロジェクトの履歴を辿る際に、多くの情報をもたらすことでしょう。

リポジトリを共有して作業している場合、pretxnchangegroup フックをそれぞれのリポジトリに対して設定することで、“不正な” ブランチ名を持つ変更が持ち込まれるのを防ぐことができます。この手法は単純ですが、“血の滴る刃” とでも言うべき（不安定な）ブランチの成果を、誤って “安定した” ブランチへと持ち込むことを防ぐには効果的です。このようなフックは、共有リポジトリの hgrc ファイルに以下のように記述します。

```
[hooks]
pretxnchangegroup.branch = hg heads --template 'branches ' | grep mybranch
```

第9章　Finding and fixing your mistakes

人は間違えるものですが、その結果をより上手に扱ってこそ、優れた構成管理システムと言えます。この章では、プロジェクトに忍び込んだ問題を発見した際に、使える手法について説明します。Mercurial は、問題の元を隔離し適切に処理するための優れた機能を持っています。

9.1　Erasing local history

9.1.1　The accidental commit

筆者は、時として考えるよりも先に入力してしまう、という根深い問題を抱えているため、不完全であったり、単純に間違った内容のチェンジセットをコミットしてしまうことがあります。筆者の場合、不完全なチェンジセットをコミットしてしまうのは、新しいソースファイルを作成したのに “`hg add`” の実行を忘れている場合が殆どです。“単純に間違っている” チェンジセットをコミットしてしまうケースには、特に共通点はありませんが、but 非常に迷惑 (no less annoying) XXXXX。

9.1.2　Rolling back a transaction

Mercurial が、リポジトリへの個々の変更を**トランザクション**として扱っていることを 4.2.2 節で述べました。チェンジセットをコミットしたり、他のリポジトリから変更を pull する際に、Mercurial は常に処理したことを記録しています。“`hg rollback`” コマンドを使用することで、きっちり一回分の処理を元に戻す、別な言い方をするなら、**巻き戻す**ことができます（このコマンドを使用する際の重要な注意が述べられていますので、9.1.4 節を参照してください）。

新しくファイルを作成したのに、そのファイルに対して “`hg add`” コマンドを実行するのを忘れてコミットしてしまう、という筆者のよくやる間違いは、以下のようなものです。

```
$ hg status
M a
$ echo b > b
$ hg commit -m 'Add file b'
```

コミット後の “`hg status`” 出力を見れば、すぐさま間違いを確証できます。

```
$ hg status
? b
$ hg tip
changeset:   1:f09f6de362c3
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:48:31 2009 +0000
summary:     Add file b

```

先のコミットは、`a` の変更は捉えていますが、新規のファイル `b` は把握していません。同僚と共有しているリポジトリに、このチェンジセットを反映してしまったら、同僚がこのチェンジセットを取り込んだ際に、`a` 中の何かが、同僚のリポジトリには存在しない `b` を参照してしまいます。そうなれば、私は同僚の憤りの対象になってしまうでしょう。

しかし、幸いなことに、チェンジセットを共有リポジトリへと反映する前に、自分の間違いを見つけています。“hg rollback” コマンドを使うことで、Mercurial は最後のチェンジセットを消してくれます。

```
$ hg rollback
rolling back last transaction
$ hg tip
changeset:   0:f58dc9900f04
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:48:31 2009 +0000
summary:     First commit

$ hg status
M a
? b
```

リポジトリの履歴上、最早最前のチェンジセットは存在しませんので、作業領域ディレクトリは、再び a ファイルが変更されている状態だとみなされます。コミット後のロールバックは、作業領域ディレクトリをコミット前の状態そのままに戻し、チェンジセットは完全に消去されます。そうなったなら、安全に b ファイルを “hg add” し、再度コミットすることができます。

```
$ hg add b
$ hg commit -m 'Add file b, this time for real'
```

9.1.3 The erroneous pull

１つの プロジェクトで、別々に開発の進んでいるブランチを Mercurial で保守する場合、それぞれ異なるリポジトリで保守することが一般的な慣習となっています。開発チームは、プロジェクトの “0.9” リリース用に共有リポジトリを持つ一方で、異なる変更履歴を持つ “1.0” リリース用のリポジトリを別途持つかもしれません。

この場合、ローカルな “0.9” リポジトリがあって、そこに偶然 “1.0” 用共有リポジトリの成果を取り込んだ場合、面倒な事態になることが想像できます。最悪の場合、十分な注意を払わないために、“1.0” のリポジトリから取り込んだ変更を “0.9” の共用リポジトリへと反映してしまったチーム全体を混乱させてしまうでしょう（この恐ろしいケースに関しては、後ほど解決方法を示しますので御安心を。）。しかし、Mercurial は成果取り込み先の URL を表示するか、Mercurial が怪しげな大量の変更をリポジトリに取り込んだことが表示されますから、すぐに気付く方があり得ます[1]。

“hg rollback” コマンドは、今まさに取り込んだ全てのチェンジセットを、きちんと綺麗にします。Mercurial は、一回の “hg pull” 起動により取り込まれるチェンジセット全体を、単一のトランザクションに分類するので、一回の “hg rollback” 起動でこの失敗を取り消すことができます。

9.1.4 Rolling back is useless once you've pushed

“hg rollback” は、一旦他のリポジトリに反映した変更でも、（手元のリポジトリにおいては）無かったことにできます。取り消しにより変更は完全に消されますが、それができるのは、“hg rollback” を実施したリポジトリにおける取り消し**のみ**です。取り消しは履歴を削除しますので、変更の取り消しをリポジトリ間で伝播する手段が無いのです。

変更を他のリポジトリ–典型的な例では共有リポジトリ–に反映した場合、本質的には、その変更は “野生に逃げ出し” ており、取り消しとは別な方法で間違いを埋め合わせる必要があります。変更を他のリポジトリに反映し、（手元のリポジトリで）その変更を取り消した後で、変更を反映したリポジトリから変更を取り込んだ時には、取り消した変更が（手元のリポジトリに）再び現れます。

[1]訳注: “display the URL it's pulling from” の関係がよくわからない

（取り消したい変更が、変更を反映したリポジトリにおける最新のもので、**且つ**、誰もそれをそのリポジトリから取り込んでいないことが確実である場合、その変更を取り消すこともできますが、取り消しが機能することには依存しないようにしてください。遅かれ早かれ変更は直接触ることのできない（あるいは存在を忘れていた）リポジトリへと反映され、回りまわって戻ってきた時に噛み付かれてしまいます。）

9.1.5 You can only roll back once

Mercurial は、当該リポジトリにおける最も最新のトランザクションを、１つだけトランザクションログに記録します。そのため、取り消せるトランザクションは１つ分だけです。トランザクションを１つ取り消した後で、その前のトランザクションも取り消せることを期待しても、期待通りの結果は得られません。

```
$ hg rollback
rolling back last transaction
$ hg rollback
no rollback information available
```

あるリポジトリでトランザクションの取り消しを行った場合、別な変更をコミットするなり取り込むなりしない限り、そのリポジトリで取り消しを行うことはできません。

9.2 Reverting the mistaken change

ファイルを変更した後で、ファイルの変更が全く必要ないことに気付いた場合、変更をコミットする前であれば、“hg revert” コマンドが利用できます。このコマンドは、作業領域ディレクトリの親チェンジセットを参照し、ファイルの内容を元の状態に戻します。（説明すると長くなりますが、通常の場合、このコマンドは変更を取り消します。）

“hg revert” コマンドの機能を、ちょっとしたサンプルで説明します。Mercurial により既に構成管理されているファイルを変更します。

```
$ cat file
original content
$ echo unwanted change >> file
$ hg diff file
diff -r d7249497af8c file
--- a/file        Mon Jul 20 21:58:41 2009 +0000
+++ b/file        Mon Jul 20 21:58:42 2009 +0000
@@ -1,1 +1,2 @@
 original content
+unwanted change
```

変更が必要ない場合、単純に “hg revert” コマンドをファイルに適用します。

```
$ hg status
M file
$ hg revert file
$ cat file
original content
```

“hg revert” コマンドは、ある程度の安全性を確保するために、.orig 拡張子付きのファイルに、変更されたファイルの内容を保存します。

```
$ hg status
? file.orig
$ cat file.orig
original content
unwanted change
```

“hg revert” コマンドが扱うことのできる状況を以下にまとめます。個々の状況に関する詳細は、以後の節で説明します。

- ファイルが変更されていたなら、変更前の状態に戻します。
- ファイルが “hg add” されていたなら、ファイルの “追加” されている状態を取り消しますが、ファイルそのものには何も変更を行いません。
- Mercurial への指示無くファイルを削除していたなら、変更前[2]の状態に戻します。
- “hg remove” コマンドでファイルを削除していたなら、ファイルの “削除された” 状態を取り消し、変更前の状態に戻します。

9.2.1 File management errors

“hg revert” は変更されたファイル以外に対しても有用なコマンドです。このコマンドは、Mercurial の全てのファイル管理コマンド—“hg add” や “hg remove” など— の実施を反転させます。

ファイルに対して “hg add” を行った後で、そのファイルを Mercurial で構成管理する必要が無いことに気付いたなら、“hg revert” によりファイルの追加を取り消せます。Mercurial はファイル自体には何も変更を行いませんので安心してください。ファイル追加の取り消しは、ファイルに対して “印を消す” だけです。

```
$ echo oops > oops
$ hg add oops
$ hg status oops
A oops
$ hg revert oops
$ hg status
? oops
```

同様に、ファイルに対して “hg remove” を行った後でも、“hg revert” を使うことで、作業領域ディレクトリの親チェンジセットにおける状態に、ファイルの内容を復旧することができます。

```
$ hg remove file
$ hg status
R file
$ hg revert file
$ hg status
$ ls file
file
```

これは、Mercurial を通さずに手動で削除したファイル（Mercurial の用語ではこの種のファイルが “紛失”(missing) と呼ばれることを思い出してください）であっても機能します。

[2]訳注: “削除前” ではない点に注意

```
$ rm file
$ hg status
! file
$ hg revert file
$ ls file
file
```

“hg copy” されたファイルに取り消しを行った場合、複製先ファイルは作業領域ディレクトリに、構成管理されない状態でそのまま残ります。複製操作は複製元ファイルには何も作用しないので、取り消しの際に Mercurial は複製元ファイルに対して特に何もしません。

```
$ hg copy file new-file
$ hg revert new-file
$ hg status
? new-file
```

A slightly special case: reverting a rename

ファイルに対して “hg rename” を行った場合、覚えていて欲しいことがあります。“hg rename” 実行に対して “hg revert” を行う際には、以下に示すように、変更後のファイル名を指定しただけでは不十分です。

```
$ hg rename file new-file
$ hg revert new-file
$ hg status
? new-file
```

“hg status” コマンドの出力からもわかるように、変名後のファイルは既に未追加状態と認識されていますが、変名**前**のファイルは未だに削除状態と認識されています！これは（少なくとも著者にとっては）直感に反しますが、扱いは簡単です。

```
$ hg revert file
no changes needed to file
$ hg status
? new-file
```

“hg rename” の取り消しを行うには、変名前後のファイル名を**両方**指定することを忘れないでください。
（ちなみに、ファイルの変名後に、変名後のファイルを変更し、それから変名前後のファイル名の両方を指定して取り消しを行った場合、Mercurial は変名の際に削除されたファイル[3]を何も変更されていない状態に戻します。変名後のファイルに対する変更を変名前ファイルに反映したい場合には、変名後ファイルから変名前ファイルへのコピーを忘れないでください。）
変名の取り消しにおけるこれらの厄介な側面は、おそらく Mercurial の小さなバグに由来するものです。

9.3 Dealing with committed changes

ある変更 *a* をコミットし、その上で別の変更 *b* をコミットした後で、変更 *a* が間違っていたことに気付いたとします。Mercurial には、チェンジセットそのものを自動的に “無かったことにする” 機能や、チェンジセットの一部を手動で無効にするための情報を提供する機能があります。

[3]訳注: 変名前のファイル

この節を読む前に、覚えておいて欲しいことが幾つかあります。“hg backout” コマンドによる変更の取り消しは、履歴を**追加**することで行われるものであり、変更そのものを修正したり削除したりするものではありません。そのため、バグの修正をするのには向いていますが、破壊的な結果を伴う取り消しといった用途には向いていません。そのような取り消しに関しては、 9.4 節を参照してください。

9.3.1 Backing out a changeset

“hg backout” コマンドは、自動化された形式でチェンジセットの効果全体を “取り消し” ます。Mercurial の履歴は改変できないので、このコマンドは取り消したいチェンジセットを取り除いたりは**しません**。その代わりにこのコマンドは、取り消したいチェンジセットによる改変内容を**反転**させる、新たなチェンジセットを作成します。

“hg backout” コマンドの操作は少々複雑ですので、例を使って説明します。まずは単純なチェンジセットを幾つか持つリポジトリを作成します。

```
$ hg init myrepo
$ cd myrepo
$ echo first change >> myfile
$ hg add myfile
$ hg commit -m 'first change'
$ echo second change >> myfile
$ hg commit -m 'second change'
```

“hg backout” コマンドは、“bakc out” 対象とする単一のチェンジセット識別子を引数に取ります。通常、“hg backout” はコミットメッセージを書くためにテキストエディタを起動しますので、変更を back out する理由を記録することができます。この例では、-m オプションを用いることで、コマンドラインからコミットメッセージを与えています。

9.3.2 Backing out the tip changeset

以下の例では、最後にコミットしたチェンジセットを back out します。

```
$ hg backout -m 'back out second change' tip
reverting myfile
changeset 2:b21d3aa081f7 backs out changeset 1:667628b6b938
$ cat myfile
first change
```

myfile が既に 2 行目を持たないことがおわかりでしょう。“hg log” 出力を見れば、“hg backout” コマンドが何を行ったかを理解できます。

```
$ hg log --style compact
2[tip]   b21d3aa081f7   2009-07-20 21:58 +0000   bos
  back out second change

1   667628b6b938   2009-07-20 21:58 +0000   bos
  second change

0   4494b9a1bd19   2009-07-20 21:58 +0000   bos
  first change

```

“hg backout” が生成した新しいチェンジセットは、back out したチェンジセットの子チェンジセットとなる点に注意してください。変更履歴を図示した 9.3.2 図を見れば、このことがわかるでしょう。ご覧の通り、履歴は見事に一直線です。

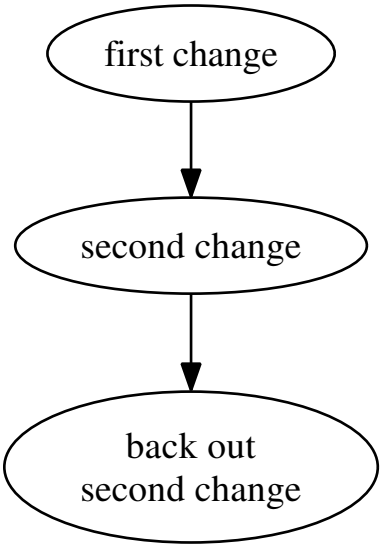

図 9.1: Backing out a change using the “hg backout” command

9.3.3 Backing out a non-tip change

最後にコミットしたチェンジセット以外を back out したい場合、“hg backout” コマンドに--merge オプションを指定してください。

```
$ cd ..
$ hg clone -r1 myrepo non-tip-repo
requesting all changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 1 files
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd non-tip-repo
```

このコマンド実行は、任意のチェンジセットを、簡単で素早い “一回限りの” 操作で back out できます。

```
$ echo third change >> myfile
$ hg commit -m 'third change'
$ hg backout --merge -m 'back out second change' 1
reverting myfile
created new head
changeset 3:b21d3aa081f7 backs out changeset 1:667628b6b938
merging with changeset 3:b21d3aa081f7
merging myfile
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

back out 完了後の myfile の内容には、１回目と３回目の変更に相当する内容は見ることができますが、２回目の変更に相当する内容は見ることができないでしょう。

```
$ cat myfile
first change
third change
```

履歴を図示した 9.3.3 図に見られるように、このような状況の場合、Mercurial は実際には**２つ**のチェンジセットをコミットします（Mercurial が自動的にコミットしたも[4]のは矩形で示してあります）。Mercurial は back out 処理を始める前に、現時点での作業領域ディレクトリにおける親チェンジセットを覚えておきます。その上で対象チェンジセットを back out し、チェンジセットとしてコミットします。最後に、作業領域ディレクトリの親チェンジセットとマージした結果をコミットします footnote 訳注: 前述のように、自動的にはコミットされません。

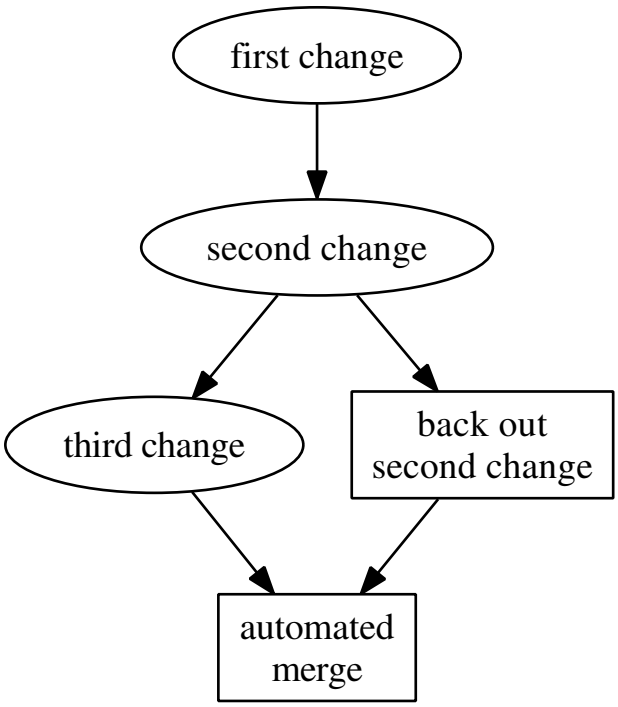

図 9.2: Automated backout of a non-tip change using the “`hg backout`” command

結果として、back out したいチェンジセットによる変更内容を取り消すための、幾つかの余分な履歴のみを伴って、“以前の状態への復旧” が行われます。

Always use the `--merge` option

実のところ、back out 対象のチェンジセットが tip か否かに関わらず、`--merge` オプションは “正しく機能” します（back out 対象が tip の場合は、必要が無いのでマージしようとはしません）ので、“`hg backout`” コマンドを実行する際には**常に**`--merge` オプションを指定するべきでしょう。

9.3.4 Gaining more control of the backout process

先の記述では、変更の back out の際の`--merge` オプションの常用を推奨しましたが、その一方で、back out 対象となるチェンジセットのマージ方法を、“`hg backout`” コマンドの利用者が決定することもできます。back out 処理を手動で制御する必要は滅多にありませんが、手動制御の方法を知ることは、“`hg backout`” が自動的に行っていることの内情を理解する上で有用です。手動制御の説明のために、最初に作成したリポジトリを複製しますが、ここでは back out は行いません。

```
$ cd ..
$ hg clone -r1 myrepo newrepo
requesting all changes
adding changesets
adding manifests
```

[4]訳注: 実行例で Mercurial が出力するメッセージを見ればわかるように、マージされたチェンジセットのコミットは利用者責任となっているため、「自動的にコミット」したものではなく「自動的に生成したもの」が正しい表現です。

```
adding file changes
added 2 changesets with 2 changes to 1 files
updating working directory
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd newrepo
```

先の例と同様に、第３のチェンジセットをコミットし、その上でその親を back out した結果を見てみましょう。

```
$ echo third change >> myfile
$ hg commit -m 'third change'
$ hg backout -m 'back out second change' 1
reverting myfile
created new head
changeset 3:3099a8f9eb11 backs out changeset 1:667628b6b938
the backout changeset is a new head - do not forget to merge
(use "backout --merge" if you want to auto-merge)
```

新たなチェンジセットも第３のチェンジセット同様に、back out 対象のチェンジセットの子になりますので、それまで tip だったチェンジセット[5]の子**ではなく**、新たなヘッドになります。“hg backout” コマンドは、このことを告げる非常にはっきりとしたメッセージを表示しています。

```
$ hg log --style compact
3[tip]:1   3099a8f9eb11   2009-07-20 21:58 +0000   bos
  back out second change

2   c0b937bbb3d7   2009-07-20 21:58 +0000   bos
  third change

1   667628b6b938   2009-07-20 21:58 +0000   bos
  second change

0   4494b9a1bd19   2009-07-20 21:58 +0000   bos
  first change

```

ここでも、履歴を図示した 9.3.4 図を見ることで、どういった状況にあるのかが理解し易いと思います。この図から、“hg backout” コマンドを tip 以外のチェンジセットに適用した際に、Mercurial が新しいヘッドをリポジトリに追加する（Mercurial により追加されたチェンジセットは矩形で表しています）ことがよくわかります。

“hg backout” コマンドの実行が完了すると、作業領域ディレクトリの親チェンジセットが、新しい “backout” チェンジセットになります。

```
$ hg parents
changeset:   2:c0b937bbb3d7
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:21 2009 +0000
summary:     third change

```

この時点で、２つの独立した変更のまとまり[6]が存在します。

[5]訳注: 第３のチェンジセットのこと
[6]訳注: マージが必要な「複数ヘッド状態」のことを指していると思われます

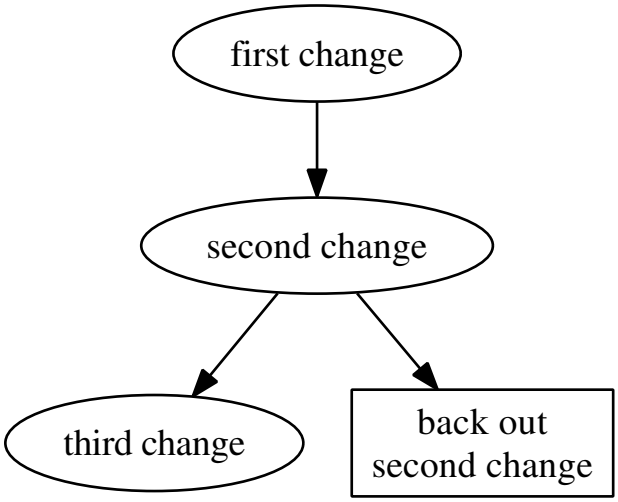

図 9.3: Backing out a change using the “hg backout” command

```
$ hg heads
changeset:   3:3099a8f9eb11
tag:         tip
parent:      1:667628b6b938
user:        Bryan O’Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:21 2009 +0000
summary:     back out second change

changeset:   2:c0b937bbb3d7
user:        Bryan O’Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:21 2009 +0000
summary:     third change

```

この時点で、myfile はどのような内容であることが期待されるかを考えてみましょう。第 1 の変更は back out していませんから、それに関する内容は存在していなければなりません。第 2 の変更は back out しましたので、それに関する内容は消失していなければなりません。履歴図で別個のヘッドとして図示されているように、第 3 の変更に関する内容が myfile に存在しては**なりません**。

```
$ cat myfile
first change
second change
third change
```

第 3 の変更の内容をファイルに取り込むには、2 つのヘッドをいつものようにマージすれば良いのです。

```
$ hg merge
merging myfile
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don’t forget to commit)
$ hg commit -m ’merged backout with previous tip’
$ cat myfile
first change
third change
```

マージすることで、リポジトリ中の履歴は 9.4 図に示すようになります。

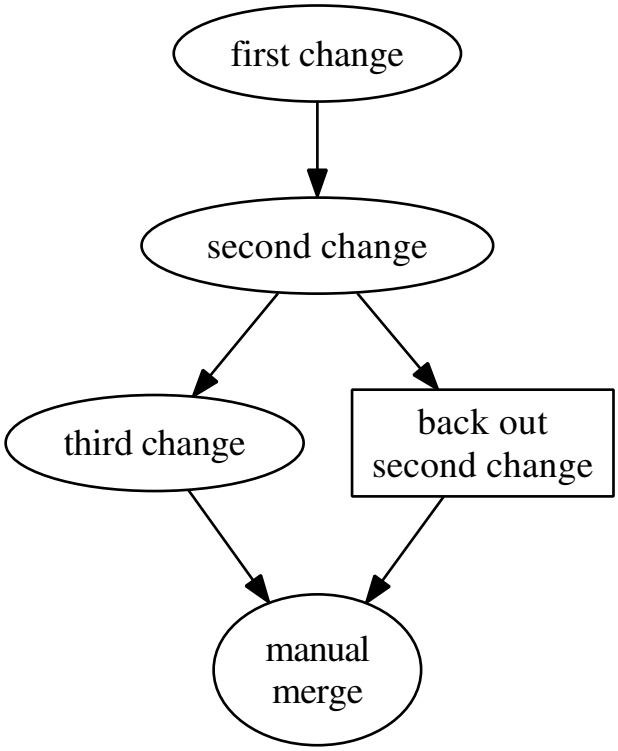

図 9.4: Manually merging a backout change

9.3.5 Why “hg backout” works as it does

“hg backout” コマンドの振る舞いを簡単にまとめると以下のようになります。

1. 作業領域ディレクトリが “クリーン” な状態、即ち “hg status” の出力が空であることを確認します。
2. その時点での作業領域ディレクトリの親チェンジセットを覚えておきます。以下、このチェンジセットを orig と呼称します。
3. 作業領域ディレクトリを back out 対象チェンジセットに同期するために、“hg update” と同等の処理を行います。以下、このチェンジセットを backout と呼称します。
4. backout の親チェンジセットを調べます。以下、この親チェンジセットを parent と呼称します。
5. backout チェンジセットが影響する個々のファイルに対して、“hg revert -r parent” 相当の処理を行い、backout チェンジセットがコミットされる前の内容に復元します。
6. 復元結果を新しいチェンジセットとしてコミットします。このチェンジセットの親は backout です。
7. コマンドラインで --merge が指定されていた場合、新しいチェンジセットと orig をマージし、その結果をコミットします。

作業領域ディレクトリを弄繰り回すことなく “hg backout” コマンド相当の効果を得るもう一つの方法は、back out されるチェンジセットに対して “hg export” することで得た diff ファイルを、作用を反転させる --reverse オプションを指定した patch コマンドに用いることです。この方法は非常に簡単に感じるでしょうが、全く上手く機能しません。

“hg backout” が update、commit、merge および再度の commit を行うのは、back out 対象のチェンジセットと現在の tip の**間**の全てのチェンジセットを扱う際に、良好な結果を得るための最善の機会を Mercurial のマージ機構に与えるためです。

例えば、プロジェクトの履歴から、100 リビジョン分前のチェンジセットを back out しようとした場合、patch がパッチの適用可否を判定するコンテキスト情報を、back out 対象との間にあるチェンジセットが “破壊” してしまうかもしれない（この意味がわからない場合は、12.4 節の patch に関する説明を参照してください）ので、patch コマン

ドが反転 diff を綺麗に適用できることは期待できません。Mercurial のマージ機構は、ファイルやディレクトリの変名、ファイル権限の変更や、バイナリファイルの変更といった `patch` コマンドが扱うことのできないものも扱うことができます。

9.4 Changes that should never have been

変更内容を取り消そうとした場合の殆どは、“`hg backout`” コマンドの利用が妥当です。“`hg backout`” コマンドは、元のチェンジセットのコミットと、後からそれを取り消した際の両方に関して、正確で永続的な記録を残します。

しかし、非常に稀な状況ですが、リポジトリ中に存在して欲しくない変更をコミットしてしまうかもしれません。例えば、ソースファイルと同様にオブジェクトファイルをコミットしてしまうような事態は、滅多に無いので通常は「間違い」とみなされます。オブジェクトファイルには本質的な価値はありませんし、非常に**サイズが大きい**ですから、リポジトサイズや複製／変更取り込みに要する時間が増加してしまいます。

XXXXXXXXXX Before I discuss the options that you have if you commit a “brown paper bag” change (the kind that’s so bad that you want to pull a brown paper bag over your head), let me first discuss some approaches that probably won’t work. XXXXXXXXXX

Mercurial は履歴を「蓄積的なもの」— 全ての変更が先行する変更の上に適用される—として扱いますので、破壊的な影響を持つチェンジセットに対してであっても、それを破棄することは通常はできません。9.1.2 節で詳細を述べますが、例外的に “`hg rollback`” コマンドを安全に使用できるのは、変更をコミットした直後で、別なリポジトリへ “`hg push`” も “`hg pull`” もされていない場合だけです。

不適切なチェンジセットを他のリポジトリへ “`hg push`” してしまった**後でも**、“`hg rollback`” コマンドにより、ローカルなリポジトリでそのチェンジセットを破棄することはできますが、それはおそらく本来やりたかったことでは無い筈です。遠隔リポジトリ中には不適切なチェンジセットが存在し続けますので、次に変更の取り込みを行った際には、その変更が再びローカルリポジトリに現れるかもしれません。

このような状況が発生した場合、どのリポジトリが不適切なチェンジセットを保持しているかを把握しているなら、それら**全ての**リポジトリからの不適切なチェンジセットの除去を、**試みる**ことが可能です。勿論、これは申し分の無い解法ではありません。たった一つでも抹消し損ねたリポジトリがあれば、“野に放たれた” ままのチェンジセットは更に伝播してしまうでしょう。

除去したいチェンジセットの**後に**、幾つかのチェンジセットをコミットしてしまった場合、取り得る選択肢は更に限られてしまいます。Mercurial は、チェンジセットに手をつけないままで、履歴に “穴を開ける” 機能は提供していません。

XXX This needs filling out. `examples` ディレクトリ配下の `hg-replay` スクリプトは機能しますが、チェンジセットのマージを行いません。重大な手抜きです。

9.4.1 Protect yourself from “escaped” changes

ローカルリポジトリにコミットした幾つかのチェンジセットが、“`hg push`” ないし “`hg pull`” 等によってそれらが他のリポジトリへと反映されたからといって、そのこと自体は必ずしも大失敗というわけではありません。ある種の不正なチェンジセットに対して、あらかじめ自己防衛することも可能です。開発チームが変更を中央のリポジトリから “`hg pull`” するような体制の場合、事故防衛は非常に簡単です。

中央のリポジトリの幾つかのフックを、追加されるチェンジセットの検証を行うように設定する（10 章を参照してください）ことで、ある種の不正なチェンジセットが、中央リポジトリに全く反映されないように自動化することができます。設定が適切であれば中央のリポジトリに反映できなくなるため、このようなチェンジセットは自然と “死に絶え” ます。なお良いことに、この手法は明示的な介入を必要としません。

例えば、当該チェンジセットが実際にコンパイル可能かどうかを検証する incoming フックは、うっかり “ビルドできなくしてしまう” ことを防止できます。

9.5 Finding the source of a bug

バグをもたらしたチェンジセットを back out できるのは非常に結構なのですが、どのチェンジセットを back out すべきかを知っている必要があります。Mercurial には、チェンジセット特定の自動化と非常に効率的な実施を補助する、

bisect と呼ばれる重要な拡張があります。

チェンジセットによる変更は振る舞いに変化をもたらすので、その変化を簡単な２値テストによりそれを特定することができる、というのが bisect 拡張の原理です。どのコード片が変化をもたらしているのかはわからなくても、バグの有無を試験する方法はわかるでしょう。bisect 拡張は、バグの原因となったコードをもたらしたチェンジセットを探すのに、あなたのテストプログラムを直接使用します。

bisect 拡張の適用方法を理解しやすいように、幾つかのシナリオを例示します。

- 数週間前には見られなかったバグが、最新の版で発見されましたが、何時それが混入されたのかがわかりません。この場合、binary test でバグの有無を調べることができます[7]。
- 大急ぎでバグを修正し、開発チームのバグデータベースの状態を「クローズ」にできるようになりました。「クローズ」状態にする際に、バグデータベースがチェンジセット ID を求めてきましたが、どのチェンジセットでバグを修正したのか覚えていませんでした。ここで再び binary test でバグの有無を調べることができます。
- ソフトウェアが正しく動作していますが、以前計測した時よりも 15% 遅くなってました。どのチェンジセットが性能低下の要因となっているのかを知りたいです。この場合、binary test はソフトウェアの性能を計測し、“早い” のか “遅い” のかを判定します。
- ここ最近、出荷したプロジェクトの構成要素のサイズが爆発的に大きくなっていて、プロジェクトのビルド手順の何らかが変更されたのではないかと推測しています[8]。

これらの例から、bisect 拡張がバグの元を探すだけのものでないことは明らかでしょう。その特性に関する２値テストを書けるなら、リポジトリにおける（ソースツリー中のファイルに対する単純な文字列検索では探し出せない）任意の “特性の出現” を探し出すことができます。

利用者と Mercurial のそれぞれが、検索処理においてどの部分に責任を負うのかをはっきりとさせるために、ここでもう少し用語の説明をしましょう。**テスト**(test) とは、bisect 拡張がチェンジセットを選択する際に、**利用者**が実行するものです。**調査**(probe) とは、あるリビジョンの良否を判定するために bisect が実行するものです。最後に、“bisect” という言葉を、“bisect 拡張を用いた検索” の代用として、名詞および動詞として使用します。

検索処理を自動化する簡単な方法の一つが、全てのチェンジセットを調査する遣り方です。しかしながら、この遣り方には殆どスケーラビリティがありません。１つのチェンジセットのテストに１０分必要で、リポジトリに１万のチェンジセットがあったとすると、徹底的に調査する遣り方では、バグをもたらしたチェンジセットを見つけるのに、平均で３５ **日**必要です。検索対象を最新の５００チェンジセットに限定できるとしても、バグをもたらしたチェンジセットを見つけるのには、それでもなお４０時間必要です。

bisect 拡張は、確認するチェンジセット数に対して**対数**のオーダーで検索（この種の検索は “二分探索” と呼ばれます）できるように、プロジェクト履歴の “形” に関する情報を利用します。この方法により、仮にテストあたりの所要時間が１０分掛かるとしても、１万チェンジセットに対する検索は２時間以内で終わります。検索対象を最新の５００チェンジセットに限定できるならば、１時間以内に検索できるでしょう。

bisect 拡張は、Mercurial で管理されているプロジェクトの履歴の持つ “枝分かれ” の特質をわかっていますので、リポジトリにおける枝分かれ・マージ・複数ヘッドの扱いも問題ありません。単一の調査で履歴の枝分かれ全体を刈り取る[9]ことができるため、bisect 拡張は効率的に検索することができるのです。

9.5.1 Using the `bisect` extension

ここでは bisect 拡張の実行例を示します。Mercurial 自体の簡便性を維持するために、bisect は拡張機能として提供されます。そのため、明示的に有効にしなければ、その機能は提供されません。bisect 拡張を有効にするには、（存在しない場合には） hgrc に以下のセクションヘッダを追加し：

```
1 [extensions]
```

[7]訳注: 「バグの有無」という２値を判定するテストを用いることで、バグの混入したチェンジセットを探します
[8]訳注: ビルド結果の「構成要素サイズの大小」という２値を判定するテストを用いることで、変更が混入されたチェンジセットを探します
[9]訳注: 「枝分かれ先全体を検索対象から除外する」の意

続いて、bisect 拡張を有効化するための行をこのセクションに追加します[10]。

```
hbisect =
```

> **備考:** bisect 拡張の名前の先頭に “h” が付くのは間違っていません。この文字が付くのは、Mercurial が Python で実装されていて、Python の標準ライブラリの bisect を使用しているためです。誤って “hbisect” から “h” を省略した場合、hgrc ファイルの記述のスペルを修正するまでは、Mercurial は Python 標準の bisect パッケージを見つけ出し、それを Mercurial 拡張として利用しようとしてクラッシュし続けることでしょう。

bisect 拡張を隔離して利用するために、リポジトリを作成しましょう。

```
$ hg init mybug
$ cd mybug
```

ループによって幾つかの些細な変更を行い、その中の特定の変更が “バグ” を持つようにする、という単純な方法で、バグを持ったプロジェクトのシミュレーションを行います。このループは 50 のチェンジセットを生成し、それぞれが 1 つのファイルをリポジトリに追加します。ここでは、ファイルが “i have a gub” というテキストを含んでいることをもって、“バグ” とみなします。

```
$ buggy_change=22
$ for (( i = 0; i < 35; i++ )); do
>   if [[ $i = $buggy_change ]]; then
>     echo 'i have a gub' > myfile$i
>     hg commit -q -A -m 'buggy changeset'
>   else
>     echo 'nothing to see here, move along' > myfile$i
>     hg commit -q -A -m 'normal changeset'
>   fi
> done
```

それでは、bisect 拡張の使用方法を理解しましょう。bisect 拡張に関しても、通常の Mercurial の組み込み help 機能が使用できます。

```
$ hg help bisect
hg bisect [-gbsr] [-c CMD] [REV]

subdivision search of changesets

    This command helps to find changesets which introduce problems. To
    use, mark the earliest changeset you know exhibits the problem as
    bad, then mark the latest changeset which is free from the problem
    as good. Bisect will update your working directory to a revision
    for testing (unless the -U/--noupdate option is specified). Once
    you have performed tests, mark the working directory as good or
    bad, and bisect will either update to another candidate changeset
    or announce that it has found the bad revision.

    As a shortcut, you can also use the revision argument to mark a
    revision as good or bad without checking it out first.

```

[10] 1.0 版以降の Mercurial では、bisect 機能は基本機能に取り込まれていますので、「拡張機能の有効化」は不要です

```
    If you supply a command, it will be used for automatic bisection.
    Its exit status will be used to mark revisions as good or bad:
    status 0 means good, 125 means to skip the revision, 127
    (command not found) will abort the bisection, and any other
    non-zero exit status means the revision is bad.

options:

 -r --reset     reset bisect state
 -g --good      mark changeset good
 -b --bad       mark changeset bad
 -s --skip      skip testing changeset
 -c --command   use command to check changeset state
 -U --noupdate  do not update to target

use "hg -v help bisect" to show global options
$ hg bisect help
abort: unknown revision 'help'!
```

bisect 拡張は段階を踏んで機能します。各段階は以下のように進みます。

1. ２値テストを実行します。
   - テストが成功した場合、“hg bisect good” コマンドにより bisect 拡張にそのことを伝えます。
   - テストが失敗した場合、“hg bisect bad” コマンドにより bisect 拡張にそのことを伝えます。
2. bisect 拡張は伝えられた情報を元に、次にテストすべきチェンジセットを決定します。
3. bisect 拡張は、作業領域ディレクトリをそのチェンジセットで更新しますので、以上の手順を繰り返します。

２値テストの結果が “成功” から “失敗” に変化した点を示す、一意なチェンジセットを bisect 拡張が特定できた時点で、この手順は終了します。
検索の開始に当たっては、“hg bisect init” コマンドの実行が必要です。

```
$ hg bisect init
(use of 'hg bisect <cmd>' is deprecated)
```

今回の実行例で使用する２値テストは簡単なもので、リポジトリ中の何れかのファイルが “i have a gub” 文字列を含んでいるか否かを判定します。含んでいる場合、そのチェンジセットは “バグの要因となる” チェンジセットです。慣習上、検索対象となる特性を持っているチェンジセットを “bad”、持っていないチェンジセットを “good” と呼びます。多くの場合、作業領域ディレクトリが同期しているリビジョン（通常は tip）はバグを持つチェンジセットにより問題を抱えているものですから、これを “bad” とみなします。

```
$ hg bisect bad
(use of 'hg bisect <cmd>' is deprecated)
```

次の作業は、バグが**無い**チェンジセットを指定することです。bisect 拡張は最初の “good” と “bad” のチェンジセット間の検査状況を “括弧” で括って表示するでしょう。今回の事例では、リビジョン 10 にはバグがありません（最初の “good” チェンジセットの選択に関しては、後ほど補足があります）。

```
$ hg bisect good 10
(use of 'hg bisect <cmd>' is deprecated)
Testing changeset 22:2ce2f0f89efb (24 changesets remaining, ~4 tests)
0 files updated, 0 files merged, 12 files removed, 0 files unresolved
```

コマンド出力には以下の意味があります。

- バグをもたらしたチェンジセットの特定までに、どれだけのチェンジセットに対して考慮が必要であるか、また、どれだけのテストを要求するかを表示します。
- bisect 拡張は次にテストすべきチェンジセットへと作業領域ディレクトリを更新し、どのチェンジセットがテスト対象であるのかを表示します。

早速作業領域ディレクトリでテストをしてみましょう。grep を使用して、作業領域ディレクトリの “bad” ファイルの有無を調べ、ファイルが無ければそのリビジョンは “good” です。

```
$ if grep -q 'i have a gub' *
> then
>   result=bad
> else
>   result=good
> fi
$ echo this revision is $result
this revision is bad
$ hg bisect $result
(use of 'hg bisect <cmd>' is deprecated)
Testing changeset 16:bfafabfda31c (12 changesets remaining, ~3 tests)
0 files updated, 0 files merged, 6 files removed, 0 files unresolved
```

このテストは完全に自動化できそうですので、シェル関数にしてしまいましょう。

```
$ mytest() {
>   if grep -q 'i have a gub' *
>   then
>     result=bad
>   else
>     result=good
>   fi
>   echo this revision is $result
>   hg bisect $result
> }
```

これで、テスト手順全体を単一の mytest コマンドで実行できます。

```
$ mytest
this revision is good
(use of 'hg bisect <cmd>' is deprecated)
Testing changeset 19:fcf0466d2d19 (6 changesets remaining, ~2 tests)
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

テスト手順が記録されたコマンドをあと数回起動することで、当初の目的が達成されます。

```
$ mytest
this revision is good
(use of 'hg bisect <cmd>' is deprecated)
Testing changeset 20:53bd36dcbf3f (3 changesets remaining, ~1 tests)
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ mytest
this revision is good
(use of 'hg bisect <cmd>' is deprecated)
Testing changeset 21:a01fa4c0a57b (2 changesets remaining, ~1 tests)
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ mytest
this revision is good
(use of 'hg bisect <cmd>' is deprecated)
The first bad revision is:
changeset:   22:2ce2f0f89efb
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:25 2009 +0000
summary:     buggy changeset

```

40 程のチェンジセット全体の検索にも関わらず、bisect 拡張はわずか５回のテストで “バグ” をもたらしたチェンジセットを特定できました。調査対象チェンジセット数に対して、bisect 拡張は対数のオーダーでテスト対象を選定するので、チェンジセットを追加しただけテスト回数が増加する “力尽く” の手法よりも有利です。

9.5.2 Cleaning up after your search

リポジトリにおける bisect 拡張の使用が終わったなら、検索に使用していた情報を “hg bisect reset” コマンドにより破棄することができます。bisect 拡張はそれほど多くの領域を消費するわけではありませんので、この作業を忘れても問題にはなりません。しかし、“hg bisect reset” を実行するまでは、bisect はそのリポジトリで別の検索を開始させてくれません。

```
$ hg bisect reset
(use of 'hg bisect <cmd>' is deprecated)
```

9.6 Tips for finding bugs effectively

9.6.1 Give consistent input

bisect 拡張には、実施した全てのテストの結果が正しく指定されなければなりません。本当はテストが成功していたにも関わらず、テストの失敗を bisect 拡張に伝えた場合、矛盾した結果を出す**かも**しれません。テスト結果に対して矛盾が検知された場合、bisect は、特定のチェンジセットが “good” でも “bad” でもある、と言ってきます。しかし、この検知は完璧に行われるわけではないので、間違ったチェンジセットをバグの要因として報告するでしょう。

9.6.2 Automate as much as possible

筆者が bisect 拡張を使い始めた頃は、検索のためのテストをコマンドラインで手動で実行していましたが、少なくとも私には、この手法は馴染みません。何度か bisect を使用した後で、最終的に正しい結果を得る前に、いつも手違いのために何度も検索をやり直していることに気付きました。

bisect 拡張を手動で駆動していた際には、小さなリポジトリにおける単純な検索であっても問題が発生していました。テストの内容が複雑であったり、bisect が要求するテスト実行回数が増えれば、それだけテスト実行における操作ミスの可能性は高まります。テストを自動化するようになって以来、非常に良好な結果を得られています。

テスト自動化のための鍵は２つあります。

- 常に同じ「症状」をテストすることと、
- 常に一貫した入力を “hg bisect” コマンドに与えること

前述の実行例では、grep コマンドにより「症状」を調べていて、if ステートメントが「検査」の結果を受けて “hg bisect” コマンドに同じ入力を与えることを保証していました。mytest 関数が、これらを再現しやすい形式に統合したことで、全てのテストが均一で整合性の取れたものになっています。

9.6.3 Check your results

bisect による検索の出力結果は与えた情報程度にしか正しくないので、bisect により “good” と報告されたチェンジセットを、絶対的に正しいものとみなさないでください。報告内容をクロスチェックする簡単な方法は、以下のようなチェンジセットのそれぞれに対して、手動で自身のテストを実行してみることです。

- 最初の “bad” リビジョンであると報告されたもの（以下、「障害チェンジセット」と呼称）。あなたのテストはこれに関して “bad” と報告しなければなりません。
- 上記チェンジセットの親チェンジセット（マージされた場合は両方の親）。あなたのテストはこれ（これら）に関して “good” と報告しなければなりません。
- 障害チェンジセットの子チェンジセット。あなたのテストはこれに関して “bad” と報告しなければなりません。

9.6.4 Beware interference between bugs

あるバグを探す際に、他のバグの存在により混乱させられる可能性もあります。例えば、リビジョン 100 でソフトウェアがクラッシュし、リビジョン 50 では正しく動作していたとします。あなたの知らない間に、ソフトウェアをクラッシュさせる別のバグを、他の人がリビジョン 60 で入れてしまい、それをリビジョン 80 で修正した場合、なんらかの方法で検索結果を混乱させるかもしれません。

他のバグの存在によって、探しているバグが完全に “覆い隠される” かもしれず、探しているバグがその存在を示す機会を得る前に他のバグが発生している、と言えます。他のバグを回避したテストが（例えば、そのバグがプロジェクトのビルドを阻害するなどの理由で）できないために、特定のチェンジセットにおける検索対象のバグの有無を明言できない場合、bisect 拡張の助けを直接受けることはできません。その替わり、他のバグが存在するチェンジセットを手動で取り除くことで、“周辺” での別な検索を行いましょう。

バグの存在に関するテストが十分明確でない場合には、別な問題が発生し得ます。“プログラムのクラッシュ” でバグの有無を確認している場合、ソフトウェアをクラッシュさせる全然関係ないバグにより、検索対象であるバグが覆い隠されてしまい、両方とも同じものとみなされるために、bisect が惑わされてしまいます。

9.6.5 Bracket your search lazily

検索における終端の印となる “good” および “bad” なチェンジセットの最初の選択は、通常は簡単なことですが、そうであっても多少は議論の余地があります。bisect の立場から見た場合、“最新” のチェンジセットは通例では “bad” で、最古のチェンジセットは “good” です。

bisect の使用に当たって “good” にふさわしいチェンジセットがどれかを思い出すのが難しい場合には、でたらめにテストするのも悪くはないでしょう。どうあってもバグの兆候が見出せない（例えば、バグの発生に関連する機能がまだ提供されていない）ものや、他の問題が（前述したように）バグを覆い隠してしまうようなものを、テスト候補のチェンジセットから除外するのを忘れないようにしましょう。

数千のチェンジセット、ないし数ヶ月の履歴の “初期” のものが最終結果だったとしても、対数オーダーの振る舞いのお陰で、bisect が実施しなければならない総回数が数回増えるだけです。

第10章　Handling repository events with hooks

Mercurial は、リポジトリに発生したイベントに応じて、自動的な処理を実行する強力な仕組みを提供しています。幾つかの状況では、イベントに対する Mercurial の応答結果を制御することもできます。

Mercurial が利用するこれらの処理は、**フック**（hook）と呼ばれています。構成管理システムによってはフックを“トリガ”と呼ぶこともありますが、これらは共に同じ考え方を指します。

10.1　An overview of hooks in Mercurial

Mercurial が提供するフックの簡単なリストを示します。これらのフックに関する詳細は 10.8 節で説明します。

`changegroup` 外部リポジトリからチェンジセット群が持ち込まれた後に実行されます。

`commit` ローカルリポジトリにおいて新たなチェンジセットが作成された後に実行されます。

`incoming` 外部リポジトリから持ち込まれた新たなチェンジセット毎に１回づつ実行されます。持ち込まれたチェンジセットの**まとまり**の単位で起動される `changegroup` との違いに注意してください。

`outgoing` 外部リポジトリへチェンジセット群が転送された後に実行されます。

`prechangegroup` 外部リポジトリからチェンジセット群が持ち込まれる前に実行されます。

`precommit` 制御用。ローカルリポジトリへのコミット前に実行されます。

`preoutgoing` 制御用。外部リポジトリへチェンジセット群が転送される前に実行されます。

`pretag` 制御用。タグ生成前に実行されます。

`pretxnchangegroup` 制御用。外部からローカルリポジトリへとチェンジセット群が持ち込まれた際に、変更を恒久的なものにするトランザクションが完了する前に実行されます。

`pretxncommit` 制御用。ローカルリポジトリにおいて新たなチェンジセットが作成された際に、変更を恒久的なものにするトランザクションが完了する前に実行されます。

`preupdate` 制御用。作業領域ディレクトリの更新・マージが実施される前に実行されます。

`tag` タグが生成された後に実行されます。

`update` 作業領域ディレクトリの更新・マージが完了した後に実行されます。

“制御用”と書かれているフックは、処理の継続性の可否を判定する機能を持っています。フックの実行が成功した場合、フックに対応する処理は継続されますが、フックの実行が失敗した場合、対応する処理は許可されないか実行しなかったことになります（どちらになるかはフックに応じて決まります）。

10.2　Hooks and security

10.2.1　Hooks are run with your privileges

リポジトリにおいて Mercurial のコマンドを実行し、そのコマンドがフックを起動することになった場合、**コマンド実行者の**システム上において、**コマンド実行者の**ユーザアカウントにより、**コマンド実行者の**権限レベルで実行されます。フックは任意の実行コードですから、十分な配慮を持って扱う必要があります。誰が作成して何をするフックなのかを熟知している確信無しに、フックをインストールしないでください。

時には、自分でインストールしたのではないフックに晒されるかもしれません。馴染みの無いシステム上で Mercurial を使用する際には、Mercurial がシステム共通の `hgrc` ファイルで定義されたフックを実行するかもしれません。

他のユーザが所有するリポジトリで作業する場合、Mercurial はそのユーザのリポジトリで定義されたフックを実行できますが、それは “あなたの” 権限で実行されます。例えば、あるリポジトリから “`hg pull`” した際に、そのリポジトリの `.hg/hgrc` ファイルが `outgoing` フックを定義していた場合、リモートリポジトリの所有者で無かったとしても、フックはあなたのアカウントで実行されます。

> **備考:** この原則は、ローカルファイルシステムかネットワークファイルシステム上のリポジトリから pull した場合にのみ適用されます。http や ssh 経由で pull した場合、フックが実行される際のアカウントは、サーバ上でサーバプロセスを実行するアカウントです。

リポジトリにおけるフックの定義状況を見るには、“`hg showconfig hooks`” コマンドが利用できます。あるリポジトリで作業中に、自分の所有していない別なリポジトリとの連携（例：“`hg pull`” ないし “`hg incoming`”）が必要になった場合、リポジトリのフック定義状況を確認すべきです[1]。

10.2.2 Hooks do not propagate

Mercurial では、フック設定の構成管理は行われないため、リポジトリの clone ないし pull の際に、フック設定は伝播しません。その理由は簡単で、フックは完全に任意の実行コードだからです。フックは、コマンド実行者のマシン上において、コマンド実行者のユーザアカウントにより、コマンド実行者の権限レベルで実行されます。

フックの構成管理の実装は、構成管理システム利用者のアカウントを弱体化させる上で、容易に悪用可能な方法を提供してしまうため、あらゆる分散構成管理システムにとって極めて無謀と言えます。

Mercurial はフックを伝播しないため、共通のプロジェクトでの他のメンバーとの連携の際には、彼らが自分と同じ Mercurial のフックを利用していることや、彼らがフックを正しく設定していることを仮定してはいけません。彼らにフックの使用を期待するのであれば、それを文書化すべきです。

企業のイントラネットの場合、例えば Mercurial の “標準的な” インストールを NFS 上で行い、組織で共通の `hgrc` ファイルで全てのユーザが使用すべきフックを定義する、といったことが可能であるため、フックの管理は幾分容易になります。しかし、それでも後述するような制限が生じます。

10.2.3 Hooks can be overridden

Mercurial は、再定義によるフックの上書きを許しています。フック指定に空文字列を設定することでフック設定を無効にすることもできますし、希望通りに振る舞いを変えることもできます。

幾つかのフックを定義した、マシンないし組織共通の `hgrc` ファイルを配備したとしても、利用者によるフックの無効化や上書きが行われる可能性があることを、理解しておく必要があります。

10.2.4 Ensuring that critical hooks are run

他のメンバーに実施して欲しくない事柄について纏めた方針を、強制したいことも時にはあるかもしれません。例えば、全てのチェンジセットには必ず厳密なテスト一式に通っていて欲しい、と思うかもしれません。この要望を実現するために、組織共通の `hgrc` ファイルでフックを定義したとしても、モバイル PC からアクセスする遠隔ユーザ等には機能しませんし、勿論ローカルユーザにとってもフックの上書きによって無効化が可能です。

（プロジェクトにおける）Mercurial の利用方針として、メンバーが変更伝播する際には、関門の機能を果たすように適切に設定された周知の “正規” サーバを通す、と策定することで、フックによる利用方針の強制を代替することが可能です。

実現方法の一つとして、ソーシャルエンジニアリングと技術の組み合わせによるものがあります。アクセス制限付きアカウントを用意し、当該アカウントで管理されたリポジトリに、各メンバーはネットワーク経由で変更を push できるようにしますが、そのアカウントでログインしたり、通常のシェルコマンドを実行したりできないようにします。このままでは、メンバーは「ゴミ」を含むようなチェンジセットのコミットも可能です。

[1] 訳注: “XXX” が付与されていることから原文未完？

メンバーが pull するサーバーへと誰かがチェンジセットを push した場合、そのチェンジセットが永続化される前にサーバーはテストを実施[2]し、テスト一式に通らなければそのチェンジセットを拒否します。メンバーがこのフィルタサーバからしかチェンジセットの pull をしないのであれば、メンバーが pull する全てのチェンジセットは、自動的に点検されていることが保証されます。

10.3 Care with `pretxn` hooks in a shared-access repository

多くの人により共有されているリポジトリに対して、フックによる自動実行を設定する場合、実施方式には注意が必要です。

Mercurial がリポジトリにロックを掛けるのは、リポジトリに書き込みを行う時だけであり、且つロックに対して注意を払うのは、Mercurial の書き込み処理の部分的な箇所だけです。書き込みロックは、複数の処理の同時書き込みによるリポジトリ破損を防ぐことで、お互いの書き込み内容を保護します。

Mercurial はデータの読み込み書き出し順序に注意を払っていますから、リポジトリからのデータ読み込みの際にロックは必要ありません。Mercurial がリポジトリからデータを読み込む際には、ロックに対して注意を払いません。ロックを必要としないこの仕組みは、性能と平行性を大きく向上させています。

しかしながら、「ロックされない」ということは、それを知らないと、大きな性能向上と引き換えにトラブル発生の潜在的な危険性を持っています。この危険性について説明するには、リポジトリへのチェンジセットの追加、およびそれらチェンジセットの読み出しを、Mercurial がどういった手順で行うかについて、幾分詳細な知識が必要となります。

Mercurial がメタデータを**書き出す**際には、対象ファイルにメタデータを直接書き出します。最初に filelog にメタデータを書き出し、次に manifest のデータ（これには、filelog に書き出した新しいデータへのポインタが含まれます）、そして changelog のデータ（これには、manifest に書き出した新しいデータへのポインタが含まれます）が書き出されます。個々のファイルへの最初の書き出しの前に、Mercurial は個々のファイルの終端位置情報をトランザクションログに記録します。Mercurial によりトランザクションが巻き戻される際には、トランザクション開始時点のサイズにまで個々のファイルが切り詰められます。

Mercurial がメタデータを**読み込む**際には、changelog を読み込んだ後でその他のファイルの読み込みを行います。データ読み込みの際には、先に読み込んだ changelog から到達可能な manifest や filelog の部分にしかアクセスしないので、不十分な書き出し中のデータを読むことはありません。

幾つかの制御用フックの（`pretxncommit` や `pretxnchangegroup`）は、トランザクションの完了直前に実行されます。この時点で全てのメタデータは書き出し済みですが、Mercurial はトランザクションを巻き戻すことで、新たに書き出されたデータを破棄することができます。

トランザクション完了前のチェンジセットは永続性が確定しておらず、そのため “本当に存在する” とみなすことができないことから、トランザクション完了前に実行される制御用フックが終了までに長時間を要する場合、永続性が確定していないチェンジセットのメタデータが、平行して動作している他の処理により読み出される時間帯が発生します。フックの実行時間が長くなる程、この時間帯が長くなります。

10.3.1 The problem illustrated

原則的に `pretxnchangegroup` フックは、集約用リポジトリでの受け入れ前に、新規チェンジセットのビルドやテストを自動化するのに適しています。この用法は “ビルドを失敗させる” 変更が集約用リポジトリに反映されないことを保証します。しかし、`pretxnchangegroup` フックによるテスト途上の変更を、他の利用者が pull できてしまうようでは、テストの有用性が無くなってしまいます。リポジトリ内容の整合性に疑いを持たない利用者は、ビルドを失敗させる潜在的な可能性を持つテスト未実施の変更を、自身のリポジトリへと反映してしまうからです。

このような難題への最も安全な技術的解法は、“門番” リポジトリの利用を**単方向**に限定してしまうことです。門番リポジトリは、外部からのチェンジセットの push は許しても、pull はできないようにします（`preoutgoing` フックでそのような行為を禁止します）。新しいチェンジセットにおけるビルドないしテストが成功したならば、そのチェンジセットを別なリポジトリへと push するように `changegroup` フックを設定し、利用者はそちらのリポジトリから pull **できる**ようにしましょう。

[2] 訳注: テスト実施はフックで実現されますが、(1) フックの実行はアクセス制限付きアカウントの権限で実行され、(2) リモートからの push の場合はフックの上書きができない、ということから、セキュリティ・フック設定の問題が共に解消されます。

実際問題、このような集約されたボトルネックを設けることは、あまり良いアイディアではなく (XXXX ?)、In practice, putting a centralised bottleneck like this in place is not often a good idea トランザクションの漏洩[3]は問題になりません。チェンジセットを取り扱う時間よりもそれをテストするのに時間を要する状況では、プロジェクトの大きさ—およびビルド・テストに要する時間—が増加するほど、“購入前の試用” 手法により壁の内側に素早く走りこめます。XXXXX ???? As the size of a project—and the time it takes to build and test—grows, you rapidly run into a wall with this “try before you buy” approach, where you have more changesets to test than time in which to deal with them. 避けられない結果は、すべてが巻き込まれた部分におけるフラストレーションです。XXXXXXX ???? The inevitable result is frustration on the part of all involved.

より大規模化可能な手法は、push 前に各自でビルド・テストを実施してもらい、push の **後**に中央で自動的にビルド・テストを行うことで、全てのチェンジセットが良好であることを確認する、というものです。この手法の利点は、リポジトリにおけるチェンジセットの受理進度に関して、制限が課されることが無い点にあります。

10.4 A short tutorial on using hooks

Mercurial のフックは簡単に書けます。“`hg commit`” が完了した際に実行され、作成したばかりのチェンジセットのハッシュ値を表示するだけの、簡単なフックを書いてみましょう。

```
$ hg init hook-test
$ cd hook-test
$ echo '[hooks]' >> .hg/hgrc
$ echo 'commit = echo committed $HG_NODE' >> .hg/hgrc
$ cat .hg/hgrc
[hooks]
commit = echo committed $HG_NODE
$ echo a > a
$ hg add a
$ hg commit -m 'testing commit hook'
committed aad7abef6e2d7e2c6d86cf554d5391d2d2efe3b2
```

図 10.1: A simple hook that runs when a changeset is committed

全てのフックは、10.1 の例における形式を踏襲します。`hgrc` ファイルの `[hooks]` セクションにエントリを追加します。左辺は実行契機になるイベントの名前で、右辺は実行される処理です。見てわかるように、フックにおいては任意のシェルコマンドを実行できます。環境変数（例における `HG_NODE` を参照してください）を用いて、Mercurial はフックに付加情報を渡します。

10.4.1 Performing multiple actions per event

10.4.1 の例に示すような、特定の種類のイベントに対して 1 つ以上のフックを定義したい状況が、しばしば発生することでしょう。Mercurial では、フック名の末尾に**拡張子**を付与することで、同一イベントへの複数フックの定義が可能になります。拡張子の付与は、フック名に、ピリオド（“.” 文字）と任意に選んだ文字列を続けることで行います。例えば、`commit` が発生した場合、Mercurial は `commit.foo` および `commit.bar` フックを実行します。

あるイベントに複数のフックが定義されている際に、その実行順序を明確に定義するために、Mercurial はフックを拡張子で整列させ、フックコマンドをこの整列された順序で実行します。上記の例では、`commit.foo` の前に `commit.bar` を、これらの前に `commit` を実行します。

新しいフックを定義する際に、何らかの説明的な拡張子を使用するのは良いアイディアです。そうすることで、そのフックが何をするためのものかを思い出しやすくなります。フックの実行が失敗した場合、フック名と拡張子を含

[3]訳注: 永続化未確定のチェンジセットが見えてしまうこと

```
$ echo 'commit.when = echo -n "date of commit: "; date' >> .hg/hgrc
$ echo a >> a
$ hg commit -m 'i have two hooks'
committed 8c33eeeac40ef9219a93dc3ea3da7bd3fa55896f
date of commit: Mon Jul 20 21:48:17 GMT 2009
```

図 10.2: Defining a second `commit` hook

むエラーメッセージが表示されますから、フックが失敗した理由に関して、説明的な拡張子から即製のヒントを得ることができます（例に関しては、10.4.2 節を参照してください）。

10.4.2 Controlling whether an activity can proceed

先の例では、コミット操作が完了した後で実行される `commit` フックを使用しました。このフックは、操作が完了した後で実行される Mercurial のフックの 1 つです。これらのフックは、操作そのものに影響を及ぼすことはありません。

Mercurial では、操作が開始される前や、操作が完了するまでの間に発生するイベントが定義されています。これらのイベントの際に起動されるフックは、操作を継続可能か中断すべきかを判断することができます。

`pretxncommit` フックは、コミット操作が概ね終了した後、コミットが完了する前の段階で起動されます。言い換えるなら、チェンジセットを表すメタデータがディスクに書き込まれてはいるものの、トランザクションが未だ完了していない状況で起動されます。`pretxncommit` フックは、トランザクションを完了させるのか、あるいは巻き戻すべきかを決定することができます。

`pretxncommit` フックが終了状態値として 0 を返却した場合、トランザクションは完了し、コミット操作は終了しますので、`commit` フックが実行されます。`pretxncommit` フックが終了状態として非 0 を返却した場合、トランザクションは巻き戻され、チェンジセットを表すメタデータは削除され、`commit` フックは実行されません。

```
$ cat check_bug_id
#!/bin/sh
# check that a commit comment mentions a numeric bug id
hg log -r $1 --template {desc} | grep -q "\<bug *[0-9]"
$ echo 'pretxncommit.bug_id_required = ./check_bug_id $HG_NODE' >> .hg/hgrc
$ echo a >> a
$ hg commit -m 'i am not mentioning a bug id'
transaction abort!
rollback completed
abort: pretxncommit.bug_id_required hook exited with status 1
$ hg commit -m 'i refer you to bug 666'
committed 940df882cbfe637969eb2c546c0de01f814b8849
date of commit: Mon Jul 20 21:48:18 GMT 2009
```

図 10.3: Using the `pretxncommit` hook to control commits

例 10.4.2 中のフックは、コミット時のコメントがバグ ID を含んでいることを確認しています。コメントがバグ ID を含んでいる場合、コミットは完了します。そうでなければ、コミット操作は巻き戻されます。

10.5 Writing your own hooks

-v オプション付き、あるいは verbose 設定項目を “true” にして Mercurial を実行するのが、フック実装の際には有用であることに気付くかもしれません。このようにして Mercurial を実行することで、それぞれのフックを起動する際に事前にメッセージを表示します。

10.5.1 Choosing how your hook should run

フックを実装する際には、通常のプログラム—典型的にはシェルスクリプト— としても実装できますが、Python 関数としても実装でき、その場合は Mercurial プロセス内で実行されます。

外部プログラムとしてフックを実装する利点は、Mercurial の内部事情に関して知る必要が無い点にあります。付加的な情報の取得のために、通常の Mercurial コマンドを起動することもできます。その利点と引き換えに、外部（プログラムとしての）フックは、プロセス内フックよりも低速[4]です。

Python 関数によるプロセス内フックは、全ての Mercurial API にアクセスでき、他のプロセスを “生成” する必要はありませんので、基本的に外部フックよりも高速です。フックが必要とする多くの情報の入手も、Mercurial コマンドから得るよりも、Mercurial API から得る方が容易です。

Python の利用が苦にならないか、高い実行性能が要求される場合、Python でのフック実装を選択すべきです。しかしながら、簡単なフックで、性能を気にする必要が無い（おそらく多くのフックがそうです）のであれば、シェルスクリプトでの実装で十分です。

10.5.2 Hook parameters

Mercurial がフックを起動する際には、明確に定義されたパラメータがフックに渡されます。Python でのフック実装の場合、パラメータはキーワード引数としてフック関数に渡されます。外部プログラムでのフック実装の場合、パラメータは環境変数として渡されます。

フック実装が Python ・シェルスクリプトのいずれであるかで、フック固有のパラメータ名とその値が決まります[5]真偽値パラメータは、Python フックでは真偽値型として表現されますが、外部フックに対しては “1”（“true” 値として）ないし “0”（“false” 値として）を持つ環境変数で表現されます。フックパラメータが foo という名前である場合、Python フックのキーワード引数の名前も foo ですが、外部フックの環境変数名は HG_FOO となります。

10.5.3 Hook return values and activity control

実行が成功したフックは、外部フックの場合は終了コード 0 で、プロセス内フックの場合は真偽値 “False” で終了しなければなりません[6]。フックの実行失敗は、外部フックの場合は非 0 の終了コードで、プロセス内フックの場合は真偽値 “true” で表されます。プロセス内ふっくが例外を浮揚した場合、フック実行は失敗したと見做されます。

操作の継続性を制御できるフックの場合、0 ／ false は継続の “許可” を、非 0 ／ true ／例外は “拒否” を意味します。

10.5.4 Writing an external hook

hgrc ファイルに外部フックを記述した場合、hgrc ファイルに記述したフックの内容は、シェルプロセスに渡され、そのシェルプロセスによって解釈されます。これは、フック記述の本体に、通常のシェルコマンドラインと同様の構造を用いることができる、ということを意味しています。

実行可能なフックは、常にリポジトリのルートディレクトリ直下で実行されます。

個々のフックパラメータは環境変数経由で渡されますが、環境変数名には、大文字化され、接頭辞として “HG_” が付与された名前が用いられます。

フックパラメータを例外とすれば、Mercurial はフック実行時に環境変数の改変を行いません。それぞれに異なる環境変数設定をしている多くのユーザによって実行される、組織全体で共用されるフックを実装する際には、この知識

[4]訳注: 後述されますが、外部プログラムによるフックが「低速」であるのは、(1) 外部プロセスとしてのフック起動と、(2)Mercurial リポジトリへのアクセスに関する部分で、外部プロセスの実行そのものが低速なわけではありません。

[5]訳注: 原文は「Whether your hook is written in Python or as a shell script, the hook-specific parameter names and values will be the “same”」

[6]訳注:Mercurial の配布物に含まれる hgext 配下のフックは、結構な確率で、False 無しの return や、明示的な return 無しの実装ですが、Python の言語仕様上、これらは False と “ほぼ等価” な None とみなされます。

が役に立つでしょう。複数ユーザにより実行される状況下では、フックの試験環境で設定されていた環境変数が、実行時に設定されていることを期待してはいけません。

10.5.5 Telling Mercurial to use an in-process hook

プロセス内フックを hgrc ファイルで設定する際の文法は、実行可能フック[7]設定の際のそれとは少々異なりますフック設定は、接頭辞 “python:” に続き、フックとして使用する呼び出し可能オブジェクト[8]の完全修飾された名前が記述されていなければなりません。

フック定義が存在するモジュールは、フック実行時に自動的に import されます。モジュール名と PYTHONPATH 設定が正しければ、きっと動作する筈です[9]。

以下に示す hgrc ファイルの引用例は、前述した表記に関する文法と意味を例示しています。

```
[hooks]
commit.example = python:mymodule.submodule.myhook
```

Mercurial が commit.example フックを起動する際には、mymodule.submodule を import し、myhook という名前の呼び出し可能オブジェクトを探し出して起動します。

10.5.6 Writing an in-process hook

以下に示す最も単純なプロセス内フックは、フックとしては何もしませんが、フック API の基本的な概要を例示できます。

```
def myhook(ui, repo, **kwargs):
    pass
```

Python フック[10]の最初の引数は、常に mercurial.ui.ui オブジェクトです。第２引数はリポジトリオブジェクトですが、現在の Mercurial の実装では、そのインスタンスは常に mercurial.localrepo.localrepository です。これらに続くその他の引数はキーワード引数として渡されます。渡される内容は起動されるフック（の種類）に依存しますが、上記例における **kwargs のように、キーワード引数辞書に落とし込む[11]ことで、興味の無い引数を無視することができます。

10.6 Some hook examples

10.6.1 Writing meaningful commit messages

有用なコミットメッセージが非常に短い、という状況は想像し難いものがあります。図 10.4 に示す単純な pretxncommit フックは、１０バイトよりも短いメッセージでのチェンジセットのコミットを妨げます。

10.6.2 Checking for trailing whitespace

コミットに関する興味深いフックの利用は、綺麗なコードでの実装を補助するというものです。簡単な “綺麗なコード” の例としては、変更が追加する新しい行には “末尾空白” が含まれていてはならない、という格言があります。末尾空白とは、空白文字およびタブ（tab）文字の連続が行末にあることを意味します。多くの場合、末尾空白は必要の無い不可視の雑音みたいなものですが、時には問題を含むことから、それらが取り除かれることを望みます。

precommit と pretxncommit のいずれのフックでも、末尾空白問題を通知することが可能です。precommit フックを使用した場合、フックはコミット対象ファイルを知ることができないので、リポジトリ中の変更されたファイル全てに対して末尾空白を確認してしまいます。そうすると、ファイル foo の変更のみをコミットしたい場合でも、bar

[7]訳注: 「外部フック」の意
[8]訳注: callable object
[9]訳注: “just work” のニュアンスは？
[10]訳注：プロセス内フックの意
[11]XXXXX: Python 固有の訳語を確認

```
$ cat .hg/hgrc
[hooks]
pretxncommit.msglen = test ‘hg tip --template {desc} | wc -c‘ -ge 10
$ echo a > a
$ hg add a
$ hg commit -A -m ’too short’
transaction abort!
rollback completed
abort: pretxncommit.msglen hook exited with status 1
$ hg commit -A -m ’long enough’
```

図 10.4: A hook that forbids overly short commit messages

ファイルが末尾空白を含んでいたなら、precommit フックでのチェックは、bar の問題を理由に foo のコミットを妨げてしまいます。これではいけません。

pretxncommit フックで実現する場合、コミットのトランザクションが完了する直前までチェックが行われません。このため、末尾空白問題の確認を、厳密にコミット対象のファイルだけに行うことができます。しかし、コミットメッセージを対話的に入力した後であっても、フックの実行が失敗[12]した場合、トランザクションは巻き戻されてしまいますので、末尾空白を取り除いた後で再び “hg commit” コマンド実行した際には、もう一度コミットメッセージを入力する必要があります。

```
$ cat .hg/hgrc
[hooks]
pretxncommit.whitespace = hg export tip | (! egrep -q ’^\+.*[ \t]$’)
$ echo ’a ’ > a
$ hg commit -A -m ’test with trailing whitespace’
adding a
transaction abort!
rollback completed
abort: pretxncommit.whitespace hook exited with status 1
$ echo ’a’ > a
$ hg commit -A -m ’drop trailing whitespace and try again’
```

図 10.5: A simple hook that checks for trailing whitespace

図 10.5 では、末尾空白をチェックする簡単な pretxncommit フックを紹介しています。このフックは短いですが、非常に有用です。変更により何れかのファイルに対して末尾空白を含む行が追加された場合、このフックはエラーステータスで終了しますが、不愉快なファイルや行の特定を補助する情報を何ら表示しません[13]。このフックは、改変されていない行には注意を払わず、末尾空白問題を持ち込む行にのみ注意を払う、という優れた特質も持っています。

図 10.6 は先の例よりは複雑ですが、より有用なフックの例を示しています[14]。このフックは unified diff 形式を解析して、末尾空白を追加する行の有無を判定し、そのようなファイルの名前と行番号を表示します。それに加えてこのフックは、チェンジセットが末尾空白を追加することを検知した場合、実行を終了して Mercurial にトランザクションの巻き戻しを伝える前に、コミットメッセージを保存してそのファイル名を表示しますので、問題点を修正した後のコミットの際には、“hg commit -l *filename*” を使ってコミットメッセージを再利用することができます。

図 10.6 ファイルから末尾空白を取り除く perl の一行記述の用法を示します。この方法はここに再掲するに足るだ

[12] 訳注: 末尾空白が検出されることでの「失敗」

[13]訳注：フック実行のコマンドラインからわかるように、export 出力（＝ patch 形式）に対して (e)grep を適用していますから、ファイル名や行番号に対しては何ら認識されていません。

[14]訳注: check_whitespace.py の内容が不明。図中でソースを cat すべき XXXX

```
$ cat .hg/hgrc
[hooks]
pretxncommit.whitespace = .hg/check_whitespace.py
$ echo 'a ' >> a
$ hg commit -A -m 'add new line with trailing whitespace'
a, line 2: trailing whitespace added
commit message saved to .hg/commit.save
transaction abort!
rollback completed
abort: pretxncommit.whitespace hook exited with status 1
$ sed -i 's, *$,,' a
$ hg commit -A -m 'trimmed trailing whitespace'
a, line 2: trailing whitespace added
commit message saved to .hg/commit.save
transaction abort!
rollback completed
abort: pretxncommit.whitespace hook exited with status 1
```

図 10.6: A better trailing whitespace hook

けの、簡潔さと有用性を持っています[15]。

```
perl -pi -e 's,                         s+$,,' filename
```

10.7 Bundled hooks

Mercurial の配布版には、幾つかのフックが添付されています。添付フックは Mercurial ソースツリーの `hgext` ディレクトリに格納されています。Mercurial のバイナリ配布版を使用している場合には、パッケージのインストーラーが Mercurial をインストールした位置にある `hgext` ディレクトリに格納されています。

10.7.1 `acl`—access control for parts of a repository

`acl` 拡張により、ネットワーク上のサーバに対してチェンジセットを push 可能な遠隔ユーザを制限することができます。リポジトリの一部（勿論全体も）を保護することができますので、特定のユーザに対しては、保護された部分に影響を及ぼさないチェンジセットのみの push が可能です。

この拡張は push 対象のチェンジセットをコミットしたユーザ**ではなく**、push を実施するユーザの身元情報を元にアクセス制御を行います。遠隔ユーザを認証する監禁（lock-downed）サーバが存在する環境で、特定のユーザだけが監禁サーバへのチェンジセットの push が許されることを確実にしたい場合でなければ、このフックの使用は意味がありません。

Configuring the `acl` hook

持ち込まれるチェンジセットを管理するために、`acl` フックは `pretxnchangegroup` フックとして用います。`pretxnchangegroup` フックとして用いられることで、外来のチェンジセットにより変更されるファイルを知ることができるため、“禁止されている” ファイルへの変更を行うチェンジセット群に対しては、トランザクションの巻き戻しが行われます。

```
[hooks]
pretxnchangegroup.acl = python:hgext.acl.hook
```

[15]訳注: コードの表示が（HTML 形式だと）２行に分割されている XXXX

acl 拡張は３つのセクションで設定されます。

[acl] セクションには、フックが注意を払うべき外来チェンジセットの出所を列挙する sources エントリだけが記述されます。通常はこのセクションを設定する必要はありません。

serve リモートリポジトリからの http ないし ssh 経由のチェンジセットに対して制御を行います。これは sources の既定値で、通常はこの設定項目に対して行う唯一の設定です。

pull ローカルリポジトリからの pull 経由のチェンジセットに対して制御を行います。

push ローカルリポジトリからの push 経由のチェンジセットに対して制御を行います。

bundle 他のリポジトリからの bundle 経由のチェンジセットに対して制御を行います。

[acl.allow] セクションは、リポジトリへのチェンジセット追加を許可されているユーザを決定します。このセクションが存在しない場合、明示的に禁止されていないユーザは、誰でもチェンジセットの追加をできます。このセクションが存在する場合、明示的に許可されていないユーザは、誰もチェンジセットの追加ができません（ですので、このセクションを空にした場合、全てのユーザがチェンジセットの追加を禁止されます）。

[acl.deny] セクションは、リポジトリへのチェンジセット追加を禁止されているユーザを決定します。このセクションが記述されない場合、全てのユーザはチェンジセットの追加を許可されます[16]。

[acl.allow] および [acl.deny] セクションの文法は同一です。各エントリの左辺は、リポジトリルート相対でのファイルないしディレクトリのマッチングパターンで、右辺はユーザ名となっています。

以下の例では、ユーザ docwriter がリポジトリの docs 配下に対する変更の push のみが許可されている一方で、ユーザ intern は source/sensitive 以外の任意のディレクトリ・ファイルに対する変更を push 可能です[17]。

```
[acl.allow]
docs/** = docwriter

[acl.deny]
source/sensitive/** = intern
```

Testing and troubleshooting

acl フックを試してみたい場合、Mercurial のデバッグ出力を有効にして実行しましょう。--debug オプションを指定し難い（あるいは不可能な）サーバ上で実行することもあるでしょうから、サーバ側の hgrc ファイルでデバッグ出力を有効化できることをお忘れなく。

```
[ui]
debug = true
```

これを有効にすることで、当該ユーザによる push を許可・禁止する理由を判断するに足る情報を表示することでしょう。

10.7.2 bugzilla—integration with Bugzilla

bugzilla 拡張は、コミットメッセージにバグ ID を検知した際に Bugzilla バグへのコメント追加を行います。このフックを共有サーバに設定することで、このサーバへのリモートからの変更伝播の際には、常にこのフックが実行されます。

このフックは Bugzilla バグに、以下のようなコメントを追加します（方法は後述しますが、コメント内容は変更できます）。

[16]訳注：原文は「no users are denied」ですが、acl.py の実装上は「禁止しない」と「許可」は等価です。

[17]訳注：設定の判定順序は (1) 禁止 (2) 許可の順序で行われ、(1) 禁止設定があり、当該ユーザのアクセスが明示的に禁止されている場合と、(2) 許可設定があり、当該ユーザのアクセスが明示的に許可されて「いない」場合に、不正アクセスとみなされ、それ以外の場合はアクセスが許可されます。

```
Changeset aad8b264143a, made by Joe User <joe.user@domain.com> in
the frobnitz repository, refers to this bug.

For complete details, see
http://hg.domain.com/frobnitz?cmd=changeset;node=aad8b264143a

Changeset description:
      Fix bug 10483 by guarding against some NULL pointers
```

このフックの価値は、チェンジセット（のコミットメッセージ）がバグを参照している際に、バグ情報を更新する手順を自動化する点にあります。フックの設定を適切に行うことで、Bugzilla バグから参照元チェンジセットへと、一直線に到達することが容易になります。

このフックの実装を足掛りにして、より高度な Bugzilla との統合を図ることも可能です。例えば:

- サーバに push される全てのチェンジセットには、コミットメッセージに適切なバグ ID が含まれていることを要求: この場合、`pretxncommit` フックに当該条件を検証するフックを設定するのが良いでしょう。コミットメッセージがバグ ID を含まないチェンジセットは、フックによって拒否されるようになります。
- 新規のチェンジセットに対して、簡単なコメントの付与と同様に、バグの**状態**の自動的な変更を許可: 例えば、“fixed bug 31337” というコミットメッセージの文字列を、バグ 31337 の状態の “requires testing” への更新、と認識させる、といった拡張も考えられます。

Configuring the `bugzilla` hook

`bugzilla` フックは、サーバ側の `hgrc` 中で `incoming` フックとして設定しなければなりません。

```
[hooks]
incoming.bugzilla = python:hgext.bugzilla.hook
```

機能特化されたフックの性質と、Bugzilla が元々この種の統合を念頭に置いていないことから、このフックの設定は何かと複雑になります。

フックの設定に先立って、フックが実行されるホスト（群）に対して、MySQL の Python バインディングをインストールしてください。対象ホストにおいてバイナリパッケージが見当たらない場合、[Dus] からダウンロードできます。

フックの設定は、`hgrc` ファイルの `[bugzilla]` セクションに記述されます。

`version` サーバにインストールされている Bugzilla のバージョン。Bugzilla のデータベーススキーマは時折変更されますので、どのスキーマが使用されているのかを厳密に知っている必要があります。今のところ、サポート対象は `2.16` のみです。

`host` Bugzilla のデータが格納されている MySQL サーバが稼動しているホスト名。MySQL サーバは、`bugzilla` フックが実行される全てのホストに対して、接続を許可している必要があります。

`user` MySQL サーバへの接続時に使用するユーザ名。MySQL サーバは、`bugzilla` フックが実行される全てのホストに対して、このユーザ名での接続を許可している必要があります。このユーザは、Bugzilla が使用するテーブルに対して読み取り・変更の両方の権限が必要です。この項目の既定値は、MySQL サーバにおける Bugzilla の標準的なユーザ名である `bugs` です。

`password` 上記ユーザの MySQL サーバにおけるパスワード。この値は平文で格納されるため、権限を持たないユーザがこの情報の書かれた `hgrc` ファイルを覗くことが無いようにしなければなりません。

`db` MySQL サーバにおける Bugzilla データベースの名前。この項目の既定値は、MySQL サーバにおける Bugzilla の標準的なデータベース名である `bugs` です。

`notify` フックによるバグへのコメント付与時に、Bugzilla による購読者への電子メール通知を実施したい場合、データベースを更新する毎にコマンドを実行させる必要があります。実行するコマンドは Bugzilla のインストール場所に依存しますが、`/var/www/html/bugzilla` にインストールしたとすると、通常は以下のようになります。

```
cd /var/www/html/bugzilla && ./processmail %s nobody@nowhere.com
```

Bugzilla の processmail プログラムは、バグ ID（フックにより “%s” が バグ ID に置換されます）と、電子メールアドレスを必要とします。このプログラムは、実行時ディレクトリへのファイル書き出しの権限も必要とします。Bugzilla とフックが同じサーバ上にインストールされていない場合、Bugzilla がインストールされているサーバ上で processmail を起動する方法を見つけ出す必要があります。

Mapping committer names to Bugzilla user names

既定状態の bugzilla フックは、チェンジセットをコミットしたユーザの電子メールアドレスを、バグの更新を行う Bugzilla ユーザ名として使用することを試みます。この挙動が状況に即さない場合、[usermap] セクションを使用して、チェンジセットをコミットしたユーザの電子メールアドレスを Bugzilla のユーザ名に変換することができます。[usermap] セクションの個々の要素は、左辺に電子メールアドレス、右辺に Bugzilla ユーザ名を保持します。

```
[usermap]
jane.user@example.com = jane
```

通常の hgrc ファイルに [usermap] データを直接保持することもできますが、bugzilla フックに外部の usermap ファイルから情報を読み込むように指示することもできます。後者の場合、例えば usermap データそのものを、利用者が改変可能なリポジトリに格納することもできます。そうすることで、利用者自身が usermap 中の各自の要素を保守することができます。この場合の hgrc ファイルは以下のように記述されます。

```
# 通常の hgrc ファイルは usermap 外部ファイルを参照
[bugzilla]
usermap = /home/hg/repos/userdata/bugzilla-usermap.conf
```

usermap が参照するファイルの内容は、以下のようになります。

```
# bugzilla-usermap.conf は hg リポジトリ内に配置
[usermap]
stephanie@example.com = steph
```

Configuring the text that gets added to a bug

Mercurial のテンプレート形式で記述することで、bugzilla フックが追加するコメントの内容を設定することが可能です。幾つかの（[bugzilla] セクションにおける） hgrc 要素により、（テンプレートの？）振る舞いを制御することができます。

strip URL における部分パス名（a partial path for a URL）を生成する際に、リポジトリにおけるパス名から取り除くパス要素の数を指定します。例えば、サーバにおけるリポジトリ群が /home/hg/repos 配下にあり、/home/hg/repos/app/tests のリポジトリを対象とする場合、strip を 4 とすることで、app/tests という部分パスを得ることができます。bugzilla フックはこの部分パス名を、テンプレートの適用の際に webroot という名前で利用可能にします。

template 使用するテンプレートテキストを指定します。通常のチェンジセット関連の置換に加えて、このテンプレートでは hgweb（後述例にあるように hgweb 項目で設定します）および webroot（前述のように strip によって生成されるパスです）が使用できます。

これらに加えて、hgrc ファイルの [web] セクションに baseurl 項目を追加することができます。Bugzilla コメントからのチェンジセット参照に使用するリンクの URL を構築する際の基底文字列として bugzilla フックはテンプレート展開の際にこの値を使用します。例えば：

```
[web]
baseurl = http://hg.domain.com/
```

bugzilla フックの設定例を以下に示します[18]。

```
[bugzilla]
host = bugzilla.example.com
password = mypassword
version = 2.16
# サーバ側リポジトリは /home/hg/repos にあるため、
# 冒頭の 4 つのセパレータ[19]を除外
strip = 4
hgweb = http://hg.example.com/
usermap = /home/hg/repos/notify/bugzilla.conf
template = Changeset {node|short}, made by {author} in the {webroot}
  repo, refers to this bug.            nFor complete details, see
  {hgweb}{webroot}?cmd=changeset;node={node|short}        nChangeset
  description:              n              t{desc|tabindent}
```

Testing and troubleshooting

bugzilla フック設定において最も良くある問題は、Bugzilla の processmail スクリプト実行に関するものと、コミットユーザ名から Bugzilla ユーザ名への変換に関するものです。

先の 10.7.2 節からの説明で述べたように、Mercurial プロセスをサーバで実行するユーザが、processmail スクリプトを実行するユーザでもあります。processmail スクリプトは Bugzilla が設定ディレクトリ中のファイルに何らかの情報を書き出す契機となるため、通常 Bugzilla の設定ファイルは Bugzilla が動作するウェブサーバの実行者の権限下にあります。

processmail 実行の際には、sudo コマンドを利用するなどして適切なユーザ権限で実行しましょう。sudoers フィルの設定例を以下に示します。

```
hg_user = (httpd_user) NOPASSWD: /var/www/html/bugzilla/processmail-wrapper %s
```

この例では、hg_user ユーザは、processmail-wrapper プログラムを httpd_user ユーザの権限下で実行することができます。

processmail プログラムは Bugzilla をインストールしたディレクトリ直下での実行が必要ですが、sudoers ファイルにはそのような制約を記述することができないので、このような間接実行のためのラッパースクリプトが必要となります。ラッパースクリプトの内容は以下のように簡単なものです。

```
#!/bin/sh
cd ‘dirname $0‘ && ./processmail "$1" nobody@example.com
```

processmail に指定する電子メールアドレスは、どのようなものでも構いません。

[usermap] が正しく設定されていない場合、チェンジセットをサーバに push した際に bugzilla フックによりエラーメッセージが表示されます。エラーメッセージは以下のようなものです。

```
cannot find bugzilla user id for john.q.public@example.com
```

このメッセージは、コミットしたユーザの電子メールアドレス john.q.public@example.com が有効な Bugzilla ユーザ名ではないか、john.q.public@example.com を有効な Bugzilla ユーザ名に変換するエントリが rcsectionusermap に記述されていないことを意味します。

10.7.3 notify—send email notifications

Mercurial の組み込みウェブサーバにより、全てのリポジトリに対してチェンジセット情報の RSS 配信機能が提供されますが、電子メールによる変更通知が選択される場合が多いです。notify フックは、購読者が興味を持つ新たなチェンジセットごとに、電子メールアドレス（群）に宛てて通知を行います。

[18]訳注: 原文の “
n” が正しく機能していないため、例示のレイアウトが乱れている

notify はテンプレート駆動型のフックですので、bugzilla フックと同様に、送信される通知の内容をカスタマイズすることができます。

既定状態では notify フックはチェンジセットごとの差分情報を取り込みますが、差分情報の量を制限したり、この機能を完全に停止することもできます。購読者による変更の即時レビューを想定する場合、指定された URL をクリックするよりも、差分情報を取り込むほうが有用です。

Configuring the `notify` hook

notify フックは、新たなチェンジセットごとに 1 通の電子メールを送信することもできれば、（単独の “hg pull” ないし “hg push” によりリポジトリに追加される）新たなチェンジセット群ごとに送信することもできます。

```
[hooks]
# チェンジセット群ごとに 1 通のメールを送信
changegroup.notify = python:hgext.notify.hook
# チェンジセットごとに 1 通のメールを送信
incoming.notify = python:hgext.notify.hook
```

このフックの設定情報は、hgrc ファイルの [notify] セクションに記述されます。

test 既定状態では、このフックは全くメールを送信しません。その替わり、送信する**であろう**メッセージを表示します。この項目を false にすることで電子メールが送信されるようになります。基底状態で電子メールの送信が停止されているのは、この拡張（／フック）をきちんと設定するのには幾分かの試行錯誤が必要なので、設定試行中に “壊れた” 通知を購読者に送信してしまうためです。

config 購読情報を保持している設定ファイルへのパス。この情報は hgrc とは分離されているので、このファイルそのものを対象リポジトリで管理することも可能です。こうすることで、対象リポジトリを複製し、購読設定を更新した上で、変更をサーバに “hg push” で戻すことができます。

strip リポジトリに対する購読者の有無を判定する際に、リポジトリのパス冒頭から取り除くパス区切りの数[20]。例えば、サーバ上のリポジトリが /home/hg/repos 配下にあり、notify が /home/hg/repos/shared/test というリポジトリを認識している場合、strip を 4 に設定することで notify による購読者とのパターンマッチングは、パスを shared/test と認識した上で行われます。

template メッセージ送信の際に使用されるテンプレートテキスト。このテンプレートは、メッセージのヘッダとボディの両方の内容を指定します。

maxdiff メッセージ末尾に付与される差分データの最大行数。この行数よりも大きい場合、差分データは切り詰められます。この値の既定値は 300 に設定されています。この値を 0 にした場合、通知の電子メールに差分データは付与されません。

sources 配慮すべきチェンジセットの由来元の一覧。この設定により例えば、遠隔ユーザがサーバを経由して当該リポジトリへ “hg push” したチェンジセットに対してのみ notify が電子メールで通知する、といった設定をすることができます。ここで記述可能な由来元の一覧は、10.8.3 節を参照してください。

[web] セクションで baseurl 項目を設定している場合、テンプレート中で webroot として使用することができます。notify 設定情報の一式を以下に示します。

```
[notify]
# 実際に電子メールを送るか否か
test = false
# 通知を行うリポジトリ自身の中に置かれている購読者情報
config = /home/hg/repos/notify/notify.conf
# リポジトリが /home/hg/repos 配下にあるので "/" 文字を 4 つ除去
strip = 4
template = X-Hg-Repo: {webroot}           n
```

[20] 訳注: ここでは strip 対象を “leading path separator characters” と表現しているが、[bugzilla] の説明では “leading path elements” と表現している。統一的な表現が必要と思われる。

```
  Subject: {webroot}: {desc|firstline|strip}        n
  From: {author}               n
                    n
  changeset {node|short} in {root}          n
  details: {baseurl}{webroot}?cmd=changeset;node={node|short}    n
  description:                n
                           t{desc|tabindent|strip}

[web]
baseurl = http://hg.example.com/
```

この設定により、以下のようなメッセージが生成されます。

```
X-Hg-Repo: tests/slave
Subject: tests/slave: Handle error case when slave has no buffers
Date: Wed,  2 Aug 2006 15:25:46 -0700 (PDT)

changeset 3cba9bfe74b5 in /home/hg/repos/tests/slave
details: http://hg.example.com/tests/slave?cmd=changeset;node=3cba9bfe74b5
description:
        Handle error case when slave has no buffers
diffs (54 lines):

diff -r 9d95df7cf2ad -r 3cba9bfe74b5 include/tests.h
--- a/include/tests.h      Wed Aug 02 15:19:52 2006 -0700
+++ b/include/tests.h      Wed Aug 02 15:25:26 2006 -0700
@@ -212,6 +212,15 @@ static __inline__ void test_headers(void *h)
[...snip...]
```

Testing and troubleshooting

既定値のままでは notify 拡張は**一切のメールを送信しません**ので、test 項目を明示的に false で設定することを忘れないでください。この設定を行うまでは、notify 拡張は送信する**であろう**メッセージを表示します。

10.8 Information for writers of hooks

10.8.1 In-process hook execution

プロセス内フックは、以下の引数形式で起動されます。

```
def myhook(ui, repo, **kwargs):
    pass
```

ui 引数は mercurial.ui.ui オブジェクト、repo 引数は mercurial.localrepo.localrepository オブジェクトです。**kwargs パラメータの持つ名前と値は、起動されるフックの種類に依存し、以下の共通の特徴を持っています。

- node ないし parent*N* という名前の引数は、16 進数のチェンジセット ID を保持しています。空の文字列は、0 続きの文字列の代わりに “null チェンジセット ID” を意味します。
- url という名前の引数は、それが特定可能であれば、遠隔リポジトリの URL を表します。
- 真偽値引数は、Python の bool オブジェクトで表されます。

プロセス内フックは、（外部フックがリポジトリ直下で実行されるのと違い）プロセスの作業ディレクトリを変更せずに起動されます。プロセスの作業ディレクトリを移動させると、Mercurial API の呼び出しが失敗する要因と成りえますので、プロセス内フックは作業ディレクトリを変更してはいけません。

（プロセス内）フックが真偽値 “false” を返却した場合、フック呼び出しは成功したものとみなされます。真偽値 “true” が返却されるか、例外が浮揚された場合、フック呼び出しは失敗したものとみなされます。起動の慣習を理解するには、“失敗したか否かを通知する” と覚えるのが良いでしょう。

チェンジセット ID は、Mercurial API が常用しているバイナリハッシュ形式ではなく、Python フックに 16 進文字列の形式で渡される点に注意してください。16 進ハッシュ値をバイナリハッシュ値形式に変換するには、`mercurial.node.bin` 関数を使用してください。

10.8.2 External hook execution

プロセス外フック（の起動文字列）は、Mercurial を実行しているシェルに渡されます。そのため、変数置換やコマンド出力のリダイレクトといった、シェルの機能が利用可能です。プロセス外フックは、（プロセス内フックが Mercurial が起動されたディレクトリで実行されるのと違い）リポジトリルート直下で実行されます。

フック引数は、環境変数を経由して渡されます。個々の環境変数の名前は、大文字で且つ “HG_” 接頭辞が付与された形式に変換されます。例えば、引数名が “node” の場合、当該引数を表す環境変数の名前は “HG_NODE” となります。

真偽値引数は、“true” が文字列 “1” で、“false” が文字列 “0” で表されます。環境変数 HG_NODE、HG_PARENT1 ないし HG_PARENT2 は、チェンジセット ID を 16 進文字列で保持します。“空のチェンジセット ID” は、“0” の連続ではなく空の文字列として表現されます。環境変数 HG_URL は、それが特定可能な場合に限り、遠隔リポジトリの URL を保持します。

プロセス外フックが終了コード 0 で終了した場合、フックの実行は成功したものとみなされます。終了コードが 0 以外の場合、フックの実行は失敗したものとみなされます。

10.8.3 Finding out where changesets come from

ローカルリポジトリと他のリポジトリの間のチェンジセットの転送に関わるフックは、“向こう側” の情報を知ることができる場合があります。Mercurial は、チェンジセットが**どのようにして**転送されたのかと、多くの場合、**どのリポジトリ**との間でチェンジセットが転送されるのかも知っています。

Sources of changesets

Mercurial はリポジトリ間でチェンジセットを転送する意図を、フックに対して事前（ないし事後に）通知します。この情報は、Python によるプロセス内フックの場合は `source` という名前の引数で、外部フックの場合は HG_SOURCE という名前の環境変数で、Mercurial からフックに渡されます。

`serve` 遠隔リポジトリとの間を、http ないし ssh 経由でチェンジセットが転送されます。

`pull` あるリポジトリから他のリポジトリへ、“`hg pull`” によりチェンジセットが転送されます。

`push` あるリポジトリから他のリポジトリへ、“`hg push`” によりチェンジセットが転送されます。

`bundle` あるリポジトリから他のリポジトリへ、“`hg bundle`” によりチェンジセットが転送されます。

Where changes are going—remote repository URLs

Mercurial は、リポジトリ間でのチェンジセット転送処理における “向こう側” の位置を、可能であればフックに知らせます。この情報は、Python によるプロセス内フックの場合は `url` という名前の引数で、外部フックの場合は HG_URL という名前の環境変数で、Mercurial からフックに渡されます。

この情報は常にわかるというわけではありません。http ないし ssh 経由でサービスを提供しているリポジトリにおいてフックが起動された場合、Mercurial は遠隔リポジトリを特定することはできませんが、クライアントがどのアドレスから接続しているのかは特定することができます。このような場合、URL は以下のいずれかの形式になります。

- `remote:ssh:`*`ip-address`*—与えられた IP アドレスからの ssh 遠隔接続。

- `remote:http:ip-address`—与えられた IP アドレスからの http 遠隔接続。クライアントが SSL を使用した場合、`remote:https:ip-address` 形式になります。
- Empty—遠隔接続に関する情報を取得できなかった場合。

10.9 Hook reference

10.9.1 changegroup—after remote changesets added

このフックは、例えば “hg pull” ないし “hg unbundle” によって、あらかじめ存在しているチェンジセットの一群が、リポジトリに追加された後に実行されます。これらの操作は任意個のチェンジセットを追加できますが、このフックは各操作毎に 1 回づつ実行されます。このことは、チェンジセットがまとまって追加されるか否かに関わらず、incoming フックの実行がチェンジセット毎に実行されるのと対照的です。

追加されたチェンジセットに対する自動化されたビルド・テストの開始契機としたり、バグデータベースの更新、リポジトリが新たなチェンジセットを取り込んだことの購読者への通知、といったものが、このフックに想定される用途の一部です。

このフックに渡されるパラメータは:

node チェンジセット ID。追加される一群の中の最初のチェンジセットの ID。このチェンジセットから tip まで（tip 自身も含む）の全てのチェンジセットが、単独の “hg pull”、“hg push” ないし “hg unbundle” 操作により追加されたことになります。

source 文字列。チェンジセットの由来元を表します。詳細は 10.8.3 節を参照してください。

url URL。特定できる場合に限り、遠隔リポジトリの場所を表します。詳細は 10.8.3 節を参照してください。

要別途参照: incoming （10.9.3 節）、prechangegroup （10.9.5 節）、pretxnchangegroup （10.9.9 節）

10.9.2 commit—after a new changeset is created

このフックは、新しいチェンジセットが作成された後で実行されます。

このフックに渡されるパラメータは:

node チェンジセット ID。新しくコミットされたチェンジセットの ID。

parent1 チェンジセット ID。新しくコミットされたチェンジセットにとって、第 1 親となるチェンジセットの ID。

parent2 チェンジセット ID。新しくコミットされたチェンジセットにとって、第 2 親となるチェンジセットの ID。

要別途参照: precommit （10.9.6 節）、pretxncommit （10.9.10 節）

10.9.3 incoming—after one remote changeset is added

このフックは、例えば “hg push” によって、あらかじめ存在しているチェンジセットが、リポジトリに追加された後に実行されます。複数のチェンジセットが単一の操作で追加された場合でも、このフックは追加された個々のチェンジセット毎に実行されます。

このフックを changegroup フック（10.9.1 節参照）と同様の目的に使用することができます。一群のチェンジセット毎のフック起動の方が便利な場合もありますが、時にはチェンジセットごとのフック起動も便利です。

このフックに渡されるパラメータは:

node チェンジセット ID。新しく追加されたチェンジセットの ID。

source 文字列。チェンジセットの由来元を表します。詳細は 10.8.3 節を参照してください。

url URL。特定できる場合に限り、遠隔リポジトリの場所を表します。詳細は 10.8.3 節を参照してください。

要別途参照: changegroup （10.9.1 節）、prechangegroup （10.9.5 節）、pretxnchangegroup （10.9.9 節）

10.9.4 `outgoing`—after changesets are propagated

このフックは、例えば “`hg push`” ないし “`hg bundle`” によって、他のリポジトリへとチェンジセットの一群が伝播した後に実行されます。

チェンジセットが外部に伝播したことの管理者への通知などは、このフックに想定される用途の 1 つです。

このフックに渡されるパラメータは:

`node` チェンジセット ID。他のリポジトリへと伝播する一群の中の最初のチェンジセットの ID。

`source` 文字列。伝播操作の発行由来を表します（10.8.3 節を参照してください）。遠隔クライアントからの “`hg pull`” 要求の場合、`source` は `serve` となります。チェンジセット群を取得しようとするクライアントがローカルホスト上に居る場合、クライアントの操作種別に応じて、`source` の値は `bundle`、`pull` ないし `push` のいずれかになります。

`url` URL。特定できる場合に限り、遠隔リポジトリの場所を表します。詳細は 10.8.3 節を参照してください。

要別途参照：`preoutgoing` （10.9.7 節）

10.9.5 `prechangegroup`—before starting to add remote changesets

この制御用フックは、他のリポジトリからのチェンジセット群の追加が Mercurial により開始される直前に実行されます。

このフックはチェンジセット群の転送開始が許可される前に実行されるため、フック自体は追加されるチェンジセットに関する情報を得ることができません。このフックの実行が失敗した場合、チェンジセット群は転送されません。

このフックの用途の一つに、リポジトリに対する外部からのチェンジセット追加の禁止があります。例えば、ローカルホスト上の管理者がリポジトリを変更できる一方で、利用者がサーバ経由で変更を “`hg push`” できないように、一時的ないし永久に “凍結” することもできます。

このフックに渡されるパラメータは:

`source` 文字列。チェンジセットの由来元を表します。詳細は 10.8.3 節を参照してください。

`url` URL。特定できる場合に限り、遠隔リポジトリの場所を表します。詳細は 10.8.3 節を参照してください。

要別途参照：`changegroup` （10.9.1 節）、`incoming` （10.9.3 節）、`pretxnchangegroup` （10.9.9 節）

10.9.6 `precommit`—before starting to commit a changeset

このフックは、Mercurial が新たなチェンジセットをコミットする前に実行されます。コミットされるファイル、コミットメッセージないし日付といった、コミットに関するメタデータを Mercurial が揃える前に実行されます。

このフックの用途の一つに、チェンジセットの受け入れを許す一方での、新たなチェンジセットのコミットの禁止があります。他の用途ととしては、ビルドやテストを実施し、それらが成功した場合にのみコミットを許可する、というものもあります。

このフックに渡されるパラメータは:

`parent1` チェンジセット ID。作業領域ディレクトリにとって、第 1 親となるチェンジセットの ID。

`parent2` チェンジセット ID。作業領域ディレクトリにとって、第 2 親となるチェンジセットの ID。

コミットが進行した場合、作業領域ディレクトリの（両）親が、新たなチェンジセットの親となります。

要別途参照：`commit` （10.9.2 節）、`pretxncommit` （10.9.10 節）

10.9.7 `preoutgoing`—before starting to propagate changesets

このフックは、Mercurial が外部に転送されるチェンジセットを特定する直前に実行されます。
チェンジセットが他のリポジトリへ転送されるのを防ぐことは、このフックに想定される用途の 1 うです。
このフックに渡されるパラメータは:

`source` 文字列。当該リポジトリに対するチェンジセットの取得要求の発行由来を表します（10.8.3 節を参照してください）。このパラメータが取り得る値に関しては、`outgoing` の `source` パラメータに関する 10.9.4 節の記述を参照してください。

`url` URL。特定できる場合に限り、遠隔リポジトリの場所を表します。詳細は 10.8.3 節を参照してください。

要別途参照: `outgoing` （10.9.4 節）

10.9.8 `pretag`—before tagging a changeset

この制御フックは、タグが生成される前に実行されます。フックの実行が成功した場合、タグの生成は継続され、フックの実行が失敗した場合、タグは生成されません。
このフックに渡されるパラメータは:

`local` 真偽値。タグがリポジトリに対してローカルなもの（`.hg/localtags` に情報が格納される）なのか、Mercurial に管理されるもの（`.hgtags` に情報が格納）なのかを表します。

`node` チェンジセット ID。タグ付けされるチェンジセットの ID。

`tag` 文字列。作成されるタグの名前。

生成されるタグが構成管理対象となる場合、`precommit` （10.9.2 節）および `pretxncommit` （10.9.10 節）フックも実行されます。
要別途参照：`tag` （10.9.12 節）

10.9.9 `pretxnchangegroup`—before completing addition of remote changesets

この制御フックは、トランザクション— このトランザクションは、他のリポジトリからの一群のチェンジセットの追加を管理します—が完了する前に実行されます。フックの実行が成功した場合、トランザクションは完了し、全てのチェンジセットがリポジトリにおいて永続化されます。フックの実行が失敗した場合、トランザクションは巻き戻され、チェンジセットに関するデータは破棄されます。
このフックは、「ほぼ追加された」チェンジセットに関するメタデータにアクセスできますが、永続化されるような操作[21]をこれらのデータに基づいて行うべきではありません作業ディレクトリも変更すべきではありません。
このフックの実行中に、他の Mercurial プロセスが同じリポジトリにアクセスしてきた場合、このプロセスからは、「ほぼ追加された」チェンジセットが永続化されたもののように見えます。この状況を回避する手順を踏まないと、競合状態になりかねません。
このフックは、チェンジセット群に対する診断に利用可能です。フックの実行が失敗した場合、トランザクションが巻き戻され、全てのチェンジセットが “拒否” されます。
このフックに渡されるパラメータは:

`node` チェンジセット ID。追加される一群の中の最初のチェンジセットの ID。このチェンジセットから `tip` まで（`tip` 自身も含む）の全てのチェンジセットが、単独の “`hg pull`”、“`hg push`” ないし “`hg unbundle`” 操作により追加されたことになります。

`source` 文字列。チェンジセットの由来元を表します。詳細は 10.8.3 節を参照してください。

`url` URL。特定できる場合に限り、遠隔リポジトリの場所を表します。詳細は 10.8.3 節を参照してください。

要別途参照：`changegroup` （10.9.1）、`incoming` （10.9.3）、`prechangegroup` （10.9.5）

[21]訳注: 例えば、外部の DBMS へのデータ格納や、公開用ファイルへの書き出し等。

10.9.10 `pretxncommit`—before completing commit of new changeset

この制御フックは、トランザクション— このトランザクションは、新たなチェンジセットのコミットを管理します—が完了する前に実行されます。フックの実行が成功した場合、トランザクションは完了し、チェンジセットがリポジトリにおいて永続化されます。フックの実行が失敗した場合、トランザクションは巻き戻され、コミットに関するデータは破棄されます。

このフックは、「ほぼ新規作成された」チェンジセットに関するメタデータにアクセスできますが、永続化されるような操作をこれらのデータに基づいて行うべきではありません作業ディレクトリも変更すべきではありません。

このフックの実行中に、他の Mercurial プロセスが同じリポジトリにアクセスしてきた場合、このプロセスからは、「ほぼ新規作成された」チェンジセットが永続化されたもののように見えます。この状況を回避する手順を踏まないと、競合状態になりかねません。

このフックに渡されるパラメータは:

`node` チェンジセット ID。新しくコミットされたチェンジセットの ID。

`parent1` チェンジセット ID。新しくコミットされたチェンジセットにとって、第 1 親となるチェンジセットの ID。

`parent2` チェンジセット ID。新しくコミットされたチェンジセットにとって、第 2 親となるチェンジセットの ID。

要別途参照：`precommit` （10.9.6 節）

10.9.11 `preupdate`—before updating or merging working directory

この制御フックは、作業領域ディレクトリにおける “`hg update`” ないし “`hg merge`” の実施前に実行されます。このフックは、Mercurial の “`hg update`” 実施前確認が “`hg update`” ないし “`hg merge`” を実行可能と判断した場合にしか実行されません。フックの実行が成功した場合、“`hg update`” ないし “`hg merge`” の実行は継続されますが、フックの実行が失敗した場合、“`hg update`” ないし “`hg merge`” は実行されません。

このフックに渡されるパラメータは:

`parent1` チェンジセット ID。作業領域ディレクトリが “`hg update`” される親チェンジセットの ID。作業領域ディレクトリが “`hg merge`” される場合は、現在の親チェンジセットと同じになります。

`parent2` チェンジセット ID。作業領域ディレクトリが “`hg merge`” される場合にのみ設定されます。作業領域ディレクトリの “`hg merge`” 対象となるチェンジセットの ID。

要別途参照：`update` （10.9.13 節）

10.9.12 `tag`—after tagging a changeset

このフックは、タグが生成された後で実行されます。

このフックに渡されるパラメータは:

`local` 真偽値。タグがリポジトリに対してローカルなもの（`.hg/localtags` に情報が格納される）なのか、Mercurial に管理されるもの（`.hgtags` に情報が格納）なのかを表します。

`node` チェンジセット ID。タグ付けされるチェンジセットの ID。

`tag` 文字列。作成されるタグの名前。

生成されるタグが構成管理対象となる場合、このフックの実行に先立って `commit` フック（10.9.2 節）が実行されます。

要別途参照：`pretag` （10.9.8 節）

10.9.13 `update`—after updating or merging working directory

このフックは、作業領域ディレクトリにおける “`hg update`” ないし “`hg merge`” が完了した際に実行されます。“`hg merge`” は失敗し得る（外部コマンドの `hgmerge` が各ファイルにおける衝突の解消に失敗した場合）ので、このフックには “`hg update`” ないし “`hg merge`” の成否が伝えられます。

`error` 真偽値。“`hg update`” ないし “`hg merge`” 実行が成功したか否かを表します。

`parent1` チェンジセット ID。新しくコミットされたチェンジセットにとって、第１親となるチェンジセットの ID。

`parent2` チェンジセット ID。新しくコミットされたチェンジセットにとって、第２親となるチェンジセットの ID。

要別途参照：`preupdate` （10.9.11 節）

第11章　Customising the output of Mercurial

Mercurial は、情報表示の体裁を制御する強力な仕組みを提供しています。この仕組みはテンプレートに基づいており、テンプレートを使用することで、単発のコマンド出力の固有化も、Mercurial 組み込みのウェブインタフェースの見かけ全体のカスタマイズもできます。

11.1　Using precanned output styles

Mercurial には即使用できる出力「様式」の幾つかが同梱されています。「様式」とは、誰かによって書かれて、Mercurial が探し出せる何処かにインストールされた、事前に用意されたテンプレートのことです。

Mercurial に同梱された「様式」を見る前に、Mercurial の標準的な出力を見てみましょう。

```
$ hg log -r1
changeset:   1:1f5a344665d8
tag:         mytag
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:48:34 2009 +0000
summary:     added line to end of <<hello>> file.

```

この出力は有益ではありますが、チェンジセット毎に５行という多くの表示領域が必要です `compact` 様式は、表題等を省くことで、この出力を３行に低減します。

```
$ hg log --style compact
3[tip]   2677318af26d   2009-07-20 21:48 +0000   bos
  Added tag v0.1 for changeset 8899b18d5d72

2[v0.1]   8899b18d5d72   2009-07-20 21:48 +0000   bos
  Added tag mytag for changeset 1f5a344665d8

1[mytag]   1f5a344665d8   2009-07-20 21:48 +0000   bos
  added line to end of <<hello>> file.

0   cd0d035975cb   2009-07-20 21:48 +0000   bos
  added hello

```

`changelog` 様式からは、Mercurial のテンプレートエンジンの持つ表現力を垣間見ることができます。この様式は、GNU プロジェクトの changelog ガイドライン [RS] に沿った出力を行います。

```
$ hg log --style changelog
2009-07-20  Bryan O'Sullivan  <bos@serpentine.com>

        * .hgtags:
        Added tag v0.1 for changeset 8899b18d5d72
        [2677318af26d] [tip]

        * .hgtags:
```

```
        Added tag mytag for changeset 1f5a344665d8
        [8899b18d5d72] [v0.1]

        * goodbye, hello:
        added line to end of <<hello>> file.

        in addition, added a file with the helpful name (at least i hope
        that some might consider it so) of goodbye.
        [1f5a344665d8] [mytag]

        * hello:
        added hello
        [cd0d035975cb]

```

Mercurial の既定出力様式が `default` という名前であることを知っても驚くほどのことは無いでしょう。

11.1.1 Setting a default style

好みの様式の名前を `hgrc` ファイルで指定することで、Mercurial がコマンド実行の際に使用する出力様式を変える事ができます。

```
[ui]
style = compact
```

自分自身で様式を定義した場合、自分の様式ファイルへのパスを指定する方法と、自分の様式ファイルを Mercurial が探し出せる場所へコピーする方法（一般には Mercurial がインストールされたディレクトリ直下の `templates` ディレクトリ）のどちらででも、自分の様式ファイルを使うことができます。

11.2 Commands that support styles and templates

“`log` 的な” 全ての Mercurial コマンドに対して、様式やテンプレートを適用できます。例えば、“`hg incoming`”、“`hg log`”、“`hg outgoing`” および “`hg tip`” がそうです[1]。

筆者がこのマニュアルを執筆している時点では、様式やテンプレートに対応しているコマンドは、それ程多くありません。しかし、対応済みのコマンドは、出力のカスタマイズが必要性が非常に高いコマンド群でしたので、Mercurial ユーザのコミュニティからは、他のコマンドにおける様式やテンプレートへの対応の要望は、今のところあまりありません。

11.3 The basics of templating

Mercurial で言うテンプレートとは、大雑把に言うなら一片のテキストです。決して変更されない部分がある一方で、必要に応じて**展開**や新たなテキストでの置換が実施されます。

詳細を説明する前に、Mercurial の通常出力の簡単な例をもう一度見てみましょう。

```
$ hg log -r1
changeset:   1:1f5a344665d8
tag:         mytag
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:48:34 2009 +0000
summary:     added line to end of <<hello>> file.

```

[1]訳注: Mercurial 0.9.5 版時点では、これ以外に “`hg heads`” および “`hg parents`” がテンプレートをサポートしています。

それでは、出力を変えるためのテンプレートを指定して、同じコマンドを実行してみましょう。

```
$ hg log -r1 --template 'i saw a changeset\n'
i saw a changeset
```

上記の例は、可能な限り最も簡単なテンプレートとして、チェンジセット毎に表示される静的なテキストを指定するだけの例です。"hg log" コマンドに対する--template オプション指定は、チェンジセット毎の表示の際に使用するテンプレートとして、指定されたテキストを使用することを Mercurial に指示します。

上記のテンプレート文字列は、"\n" で終了している点に注意してください。これは**エスケープシーケンス**と呼ばれるもので、個々のテンプレート要素の終端で改行を表示することを Mercurial に指示します。この改行を省略した場合、Mercurial は個々の出力要素を単一行で出力します。エスケープシーケンスに関する詳細は、11.5 節を参照してください。

常に固定された文字列を表示するテンプレートは、あまり有用とは言えませんので、もう少し複雑なものに挑戦してみましょう。

```
$ hg log --template 'i saw a changeset: {desc}\n'
i saw a changeset: Added tag v0.1 for changeset 8899b18d5d72
i saw a changeset: Added tag mytag for changeset 1f5a344665d8
i saw a changeset: added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some might consider it so) of goo
i saw a changeset: added hello
```

ご覧の通り、テンプレート中の "{desc}" 文字列は、チェンジセット毎のログメッセージで置換されて出力されます。波括弧（"{" 及び "}"）で囲まれたテキストが検出された際には、どんなテキストが囲まれていた場合でも常に、括弧およびテキスト部分の展開が Mercurial により試みられます。波括弧そのものを表示したい場合は、11.5 節で述べる方法で、波括弧をエスケープしなければなりません。

11.4 Common template keywords

以下のキーワードを使用することで、すぐにでも簡単なテンプレートを書くことができます。

author 文字列。チェンジセットの作成者。チェンジセット作成後は変更されません。

branches 文字列。チェンジセットがコミットされたブランチの名前。ブランチ名が default の場合は空です。

date 日付情報。チェンジセットがコミットされた日時。この値は可読性が**ありません**ので、適切に文字列化するフィルタに渡す必要があります。フィルタに関する詳細は 11.6 節を参照してください。日時は数値の対として表されます。最初の数値は Unix UTC タイムスタンプ（1970 年 1 月 1 日からの経過秒）で、２つ目の数値はコミットの際の UTC からのタイムゾーンオフセット秒数です。

desc 文字列。チェンジセットのログメッセージ。

files 文字列リスト。当該チェンジセットで変更・追加ないし削除された全てのファイル。

file_adds 文字列リスト。当該チェンジセットで追加されたファイル。

file_dels 文字列リスト。当該チェンジセットで削除されたファイル。

node 文字列。チェンジセット識別用ハッシュ値を４０文字の１６進数文字列化したもの。

parents 文字列リスト。チェンジセットの親。

rev 整数値。リポジトリローカルなチェンジセットのリビジョン番号。

```
$ hg log -r1 --template 'author: {author}\n'
author: Bryan O'Sullivan <bos@serpentine.com>
$ hg log -r1 --template 'desc:\n{desc}\n'
desc:
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some might consider it so) of goo
$ hg log -r1 --template 'files: {files}\n'
files: goodbye hello
$ hg log -r1 --template 'file_adds: {file_adds}\n'
file_adds: goodbye
$ hg log -r1 --template 'file_dels: {file_dels}\n'
file_dels:
$ hg log -r1 --template 'node: {node}\n'
node: 1f5a344665d8e800e1d88631d873fada9816c8f5
$ hg log -r1 --template 'parents: {parents}\n'
parents:
$ hg log -r1 --template 'rev: {rev}\n'
rev: 1
$ hg log -r1 --template 'tags: {tags}\n'
tags: mytag
```

図 11.1: Template keywords in use

tags 文字列リスト。当該チェンジセットに関連付けられたタグ。

幾つか実験してみることで、これらのキーワードを使用した際に期待される動作を見ることができます。図 11.1 を参照してください。

前述したように、date キーワードは可読性のある出力を生成しませんので、特別扱いする必要があります。そのためには *filter* を使う必要がありますが、詳細は 11.6 節を参照してください。

```
$ hg log -r1 --template 'date: {date}\n'
date: 1248126514.00
$ hg log -r1 --template 'date: {date|isodate}\n'
date: 2009-07-20 21:48 +0000
```

11.5 Escape sequences

Mercurial のテンプレートエンジンは、最も広く使われている文字列エスケープシーケンスを認識します。バックスラッシュ（"\"）を検知した際には、それに続く文字を見て、それら 2 つの文字を以下に示すような単独の文字に置換します。

\\ バックスラッシュ（"\"）／ ASCII 134。

n 改行／ ASCII 12.

r 行頭／ ASCII 15.

t タブ／ ASCII 11.

v 垂直タブ／ ASCII 13.

\{ 開き波括弧（"{"）／ ASCII 173.

\} 閉じ波括弧（“}”）／ ASCII 175.

上記のように、“\”、“{” ないし “{” そのものを含むテンプレートを使用したい場合、これらはエスケープされなければなりません。

11.6 Filtering keywords to change their results

テンプレート展開における結果のうちの幾つかは、直ちに使えるほど簡便なものではありません。Mercurial は、キーワードの展開結果を変更するために、任意の**フィルタ**の連鎖を指定することを求めてきます。上記の実行例において既に、一般的なフィルタである `isodate` を、日付を読めるようにするために使用しています。

Mercurial がサポートする最も一般的に使用されるフィルタのリストを、以下に示します。任意のテキストに適用できるフィルタもあれば、特定の状況下でのみ適用可能なものもあります。個々のフィルタの説明は、名前に続いて利用可能な状況を提示し、それに効果の説明が続く形式となっています。

`addbreaks` 任意のテキストに適用可能。XHTML の “<br/>” タグを、最終行を除く各行の末尾に付与します。例えば “foo\nbar” は “foo
\nbar” となります。

`age` `date` キーワードに適用可能。現在時刻に対する日付の年齢を描画します。“10 minutes” のような文字列を生成します。

`basename` 任意のテキストに適用可能ですが、`files` キーワードやその相対値に対して適用するのが最も有用です。テキストをパスとして扱い、そのベースネームを返します。例えば “foo/bar/baz” は “baz” となります。

`date` `date` キーワードに適用可能。Unix の `date` コマンドと同等のフォーマットで日付を描画しますが、タイムゾーンを含みます。“Mon Sep 04 15:13:13 2006 -0700” のような文字列を生成します。

`domain` 任意のテキストに適用可能ですが、`author` キーワードに対して適用するのが最も有用です。電子メールアドレスと思しき最初の文字列を見つけ出し、ドメイン部分のみを取り出します。例えば “Bryan O’Sullivan <bos@serpentine.com>” は “serpentine.com” となります。

`email` 任意のテキストに適用可能ですが、`author` キーワードに対して適用するのが最も有用です。電子メールアドレスと思しき最初の文字列を見つけ出します。例えば “Bryan O’Sullivan <bos@serpentine.com>” は “bos@serpentine.com” となります。

`escape` 任意のテキストに適用可能。XML/XHTML の特殊文字である “&”、“<” および “>” を、XML の実体参照形式で置き換えます。

`fill68` 任意のテキストに適用可能。テキストを 68 桁に収まるように行を折り返します。`tabindent` フィルタ実施後も 80 桁の固定フォント幅の画面に収めたい場合、`tabindent` フィルタに渡す前のテキストに適用するのが良いでしょう。

`fill76` 任意のテキストに適用可能。76 桁に収まるように行を折り返します。

`firstline` 任意のテキストに適用可能。テキストの最初の行を、改行等を含まない形式で取り出します。

`hgdate` `date` キーワードに適用可能。可読性のある数値の組として日付を描画します。“1157407993 25200” のような文字列を生成します。

`isodate` `date` キーワードに適用可能。ISO 8601 形式の文字列として日付を描画します。“2006-09-04 15:13:13 -0700” のような文字列を生成します。

`obfuscate` 任意のテキストに適用可能ですが、`author` キーワードに対して適用するのが最も有用です。入力テキストに対応する XML 実体参照シーケンスを生成します。典型的な電子メールアドレス収集を行うスパムボット（spambot）に対する対抗策の 1 つとして利用可能です。

`person` 任意の文字列に適用可能ですが、`author` キーワードに対して適用するのが最も有用です。電子メールアドレスより前の部分を取り出します。例えば “Bryan O’Sullivan <bos@serpentine.com>” は “Bryan O’Sullivan” となります。

rfc822date date キーワードに適用可能。電子メールヘッダと同じ形式で日付を描画します。“Mon, 04 Sep 2006 15:13:13 -0700” のような文字列を生成します。

short チェンジセットハッシュ値に適用可能です。チェンジセットハッシュの短縮形式、即ち 12 桁の 16 進文字列を生成します。

shortdate date キーワードに適用可能。年月日形式で日付を描画します。“2006-09-04” のような文字列を生成します。

strip 任意のテキストに適用可能。冒頭ならびに末尾の空白文字を全て除外します。

tabindent 任意のテキストに適用可能。最初の行を除く全ての行がタブ文字で始まるようにします。

urlescape 任意のテキストに適用可能。URL 解析の際に “特殊文字” とされる文字をエスケープします。例えば foo bar は foo%20bar になります。

user 任意の文字列に適用可能ですが、author キーワードに対して適用するのが最も有用です。電子メールアドレスから “ユーザ” 部分を取り出します。例えば “Bryan O’Sullivan <bos@serpentine.com>” は “bos” となります。

> **備考:** 適用対象外のデータに対してフィルタの適用を試みた場合、Mercurial は実行に失敗して Python の例外を表示します。例えば、desc キーワードに isodate フィルタを適用するのはよろしくありません。

11.6.1 Combining filters

所定の形式での出力を得るために、簡単にフィルタを組み合わせることができます。以下の例では、ログメッセージの冒頭・末尾の空白を除外し、68 桁に収まるように改行した後で、さらに８文字分（タブ文字が慣習的に８文字として扱われる Unix 的な環境では）の字下げが、フィルタ連鎖により実施されます。

```
$ hg log -r1 --template 'description:\n\t{desc|strip|fill68|tabindent}\n'
description:
        added line to end of <<hello>> file.

        in addition, added a file with the helpful name (at least i hope
        that some might consider it so) of goodbye.
```

テンプレートにおける “\t”（タブ文字）の利用は、最初の行の強制的な字下げを行うためのものであることに注意してください。tabindent が最初の行**以外の**全ての行を字下げするために、このタブ文字が必要です。

連鎖におけるフィルタの順序が重要である点に留意してください。最初のフィルタがキーワードの置換結果に適用され、２つ目のフィルタが最初のフィルタの適用結果に適用される、という具合です。例えば、fill68|tabindent という記述は tabindent|fill68 とは全く違った結果となります。

11.7 From templates to styles

コマンド行でのテンプレート指定は、手早く簡単に出力を整形する手段を提供します。しかし、テンプレートは冗長に成りがちですから、テンプレートに名前付けできれば便利になります。様式（sytle）ファイルは、名前が付けられ、ファイルに保存されたテンプレートのことです。

それ以上に、コマンド行での --template オプション使用では引き出せなかった Mercurial のテンプレートエンジンの能力を、様式ファイルを用いることで引き出すことができます。

```
$ hg log -r1 --template '{author}\n'
Bryan O'Sullivan <bos@serpentine.com>
$ hg log -r1 --template '{author|domain}\n'
serpentine.com
$ hg log -r1 --template '{author|email}\n'
bos@serpentine.com
$ hg log -r1 --template '{author|obfuscate}\n' | cut -c-76
&#66;&#114;&#121;&#97;&#110;&#32;&#79;&#39;&#83;&#117;&#108;&#108;&#105;&#11
$ hg log -r1 --template '{author|person}\n'
Bryan O'Sullivan
$ hg log -r1 --template '{author|user}\n'
bos
$ hg log -r1 --template 'looks almost right, but actually garbage: {date}\n'
looks almost right, but actually garbage: 1248126514.00
$ hg log -r1 --template '{date|age}\n'
3 seconds
$ hg log -r1 --template '{date|date}\n'
Mon Jul 20 21:48:34 2009 +0000
$ hg log -r1 --template '{date|hgdate}\n'
1248126514 0
$ hg log -r1 --template '{date|isodate}\n'
2009-07-20 21:48 +0000
$ hg log -r1 --template '{date|rfc822date}\n'
Mon, 20 Jul 2009 21:48:34 +0000
$ hg log -r1 --template '{date|shortdate}\n'
2009-07-20
$ hg log -r1 --template '{desc}\n' | cut -c-76
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some m
$ hg log -r1 --template '{desc|addbreaks}\n' | cut -c-76
added line to end of <<hello>> file.<br/>
<br/>
in addition, added a file with the helpful name (at least i hope that some m
$ hg log -r1 --template '{desc|escape}\n' | cut -c-76
added line to end of &lt;&lt;hello&gt;&gt; file.

in addition, added a file with the helpful name (at least i hope that some m
$ hg log -r1 --template '{desc|fill68}\n'
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope
that some might consider it so) of goodbye.
$ hg log -r1 --template '{desc|fill76}\n'
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some
might consider it so) of goodbye.
$ hg log -r1 --template '{desc|firstline}\n'
added line to end of <<hello>> file.
$ hg log -r1 --template '{desc|strip}\n' | cut -c-76
added line to end of <<hello>> file.

in addition, added a file with the helpful name (at least i hope that some m
$ hg log -r1 --template '{desc|tabindent}\n' | expand | cut -c-76
added line to end of <<hello>> file.

       in addition, added a file with the helpful name (at least i hope tha
$ hg log -r1 --template '{node}\n'
1f5a344665d8e800e1d88631d873fada9816c8f5
$ hg log -r1 --template '{node|short}\n'
1f5a344665d8
```


図 11.2: Template filters in action

11.7.1 The simplest of style files

以下に示す簡単な様式ファイルは、１行だけのものです。

```
$ echo 'changeset = "rev: {rev}\n"' > rev
$ hg log -l1 --style ./rev
rev: 3
```

この様式記述は、“チェンジセットを表示する際には、右辺のテキストをテンプレートとして使用せよ” と Mercurial に指示します。

11.7.2 Style file syntax

様式ファイルの文法は簡単です。

- ファイルは一行づつ処理されます。
- 行頭および行末の空白は無視されます。
- 空行は読み飛ばされます。
- “#” ないし “;” のいずれかで始まる行は、行全体がコメントとみなされ、空行と同様に読み飛ばされます。
- 行はキーワードで開始されます。キーワードは英字ないし下線（underscore）で開始され、任意個数の英数字ないし下線が続きます（正規表現で書くなら、キーワードは “`[A-Za-z_][A-Za-z0-9_]*`.” に合致しなければなりません）。
- キーワードに続く要素は文字 “=” でなければなりませんが、前後に任意個の空白文字があっても構いません。
- 行の残り部分が引用符（シングルクォートないしダブルクォート）で囲まれている場合、その部分はテンプレートの本体とみなされます。
- 行の乗り部分が引用符で囲まれて**いない**場合、その部分は、テンプレート本体を内容として持つファイルのファイル名とみなされます。

11.8 Style files by example

様式ファイルの記述を説明するために、幾つかの例を示します。様式ファイル一式を通して読むよりも、非所に簡単な例から始めて、幾つかの複雑な例を通し読みすることで、通常の様式ファイル作成手順を示そうと思います。

11.8.1 Identifying mistakes in style files

様式ファイル中に問題があった場合、Mercurial はそっけないエラーメッセージを表示しますが、意味するところがわかってしまえば、そのメッセージは非常に有用です。

```
$ cat broken.style
changeset =
```

`broken.style` は、`changeset` キーワードを定義しようとしているものの、その内容が記述されていない点に注目してください。このような様式ファイルが指定された場合、Mercurial は即座にメッセージを表示します。

```
$ hg log -r1 --style broken.style
** unknown exception encountered, details follow
** report bug details to http://mercurial.selenic.com/bts/
** or mercurial@selenic.com
** Mercurial Distributed SCM (version 1.3)
** Extensions loaded:
Traceback (most recent call last):
  File "/home/vmuser/bin/hg", line 27, in <module>
    mercurial.dispatch.run()
  File "/home/vmuser/lib/python/mercurial/dispatch.py", line 16, in run
    sys.exit(dispatch(sys.argv[1:]))
  File "/home/vmuser/lib/python/mercurial/dispatch.py", line 27, in dispatch
    return _runcatch(u, args)
  File "/home/vmuser/lib/python/mercurial/dispatch.py", line 43, in _runcatch
    return _dispatch(ui, args)
  File "/home/vmuser/lib/python/mercurial/dispatch.py", line 449, in _dispatch
    return runcommand(lui, repo, cmd, fullargs, ui, options, d)
  File "/home/vmuser/lib/python/mercurial/dispatch.py", line 317, in runcommand
    ret = _runcommand(ui, options, cmd, d)
  File "/home/vmuser/lib/python/mercurial/dispatch.py", line 501, in _runcommand
    return checkargs()
  File "/home/vmuser/lib/python/mercurial/dispatch.py", line 454, in checkargs
    return cmdfunc()
  File "/home/vmuser/lib/python/mercurial/dispatch.py", line 448, in <lambda>
    d = lambda: util.checksignature(func)(ui, *args, **cmdoptions)
  File "/home/vmuser/lib/python/mercurial/util.py", line 370, in check
    return func(*args, **kwargs)
  File "/home/vmuser/lib/python/mercurial/commands.py", line 2030, in log
    displayer = cmdutil.show_changeset(ui, repo, opts, True, matchfn)
  File "/home/vmuser/lib/python/mercurial/cmdutil.py", line 978, in show_changeset
    t = changeset_templater(ui, repo, patch, opts, mapfile, buffered)
  File "/home/vmuser/lib/python/mercurial/cmdutil.py", line 742, in __init__
    'filecopy': '{name} ({source})'})
  File "/home/vmuser/lib/python/mercurial/templater.py", line 160, in __init__
    if val[0] in "'\"":
IndexError: string index out of range
```

このメッセージは威圧的に見えますが、読み解くのはそれほど難しくありません。

- 最初の要素は、単に Mercurial が “実行をあきらめました” と通知しています。

```
abort: broken.style:1: parse error
```

- 次の要素は、エラーの要因が格納された様式ファイルの名前です。

```
abort: broken.style:1: parse error
```

- ファイル名の次は、エラーが発生した行番号になります。

```
abort: broken.style:1: parse error
```

- 最後に、問題の説明が記述されます。

```
abort: broken.style:1: parse error
```

問題の説明は（この例のように）常に明確であるとは限りませんが、暗号めいたものであったとしても、様式ファイル中の問題となる行を目視確認して間違いを見つける上では、殆どの場合は取るに足らない説明です。

11.8.2 Uniquely identifying a repository

短い文字列を識別子として Mercurial リポジトリを “概ね一意に” 識別[2]したい場合、リポジトリの最初のリビジョンを使用するのが良いでしょう。

```
$ hg log -r0 --template '{node}'
84c0f05ef0a48b978212fbe41c27597bd50397a0
```

この値は一意であることが保証されていませんが、それでも多くの場合において有用です。

- 完全に空のリポジトリではリビジョン 0 が存在しないため、この方法は機能しません。
- 以前は別々だった複数のリポジトリをマージしたものと、マージ前のリポジトリを併用している場合（このような事態は非常に稀ではありますが）、それらのリポジトリの間では、この方法による識別は機能しません。

リポジトリ識別子の利用例を以下に示します。

- サーバ上のリポジトリを管理しているデータベースでの、テーブルにおけるキーとしての使用
- { **リポジトリ識別子**, **リビジョン識別子**} というタプルの一部としての使用。ビルドや他の自動化された処理を実施する際に、このタプル情報を保存しておくことで、後に処理を “再現” することが可能です。

11.8.3 Mimicking Subversion's output

例えば Subversion のような、他の構成管理ツールのデフォルト出力形式をまねてみましょう。

```
$ svn log -r9653
------------------------------------------------------------------------
r9653 | sean.hefty | 2006-09-27 14:39:55 -0700 (Wed, 27 Sep 2006) | 5 lines

On reporting a route error, also include the status for the error,
rather than indicating a status of 0 when an error has occurred.

Signed-off-by: Sean Hefty <sean.hefty@intel.com>

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

Subversion の出力様式はかなり単純ですので、出力内容をファイルに保存し、出力テキスト中で Subversion により（動的に）生成される部分を、展開されるテンプレート値[3]で置き換えるのは容易でしょう。

```
$ cat svn.template
r{rev} | {author|user} | {date|isodate} ({date|rfc822date})

{desc|strip|fill76}

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

このテンプレートによる出力が、Subversion により生成される出力様式から逸脱する場合[4]が幾つかあります。

[2]訳注: ここで言う「リポジトリの識別」は、むしろ「プロジェクトの識別」に近いニュアンスと思われます。
[3]訳注： キーワードのこと？
[4] 訳注： “a few small ways” よりは “a few small point” で、「逸脱する箇所」の方が良くないか？

- Subversion は、“可読性のある” 日付（上記の出力例における “Wed, 27 Sep 2006”）を丸括弧の中に表示します。Mercurial のテンプレートエンジンは、時刻とタイムゾーンの無いこの形式で日付を表示する手段を提供していません。
- テンプレート末尾に “-” 文字を一杯に使った行の表示を配置することで Subversion の “分離” 線をまねています。Subversion の出力に似せるため、出力の最初の分離線表示には、テンプレートエンジンの header キーワードを使用しています（後述します）。[5]
- Subversion の出力は、ヘッダ部にコミットメッセージの行数が表示されます。Mercurial ではこれに相当する情報を表示することができません。処理対象となるデータの行数を数え上げるフィルタを、テンプレートエンジンが現時点では提供していないためです。

Subversion の出力例を元に、上記テンプレートのようなキーワード・フィルタへの置き換えを行う作業は、せいぜいが１～２分で済む作業です。様式ファイルは、単にこのテンプレートを参照すれば良いのです。

```
$ cat svn.style
header = '------------------------------------------------------------------------\n\n'
changeset = svn.template
```

テンプレートファイルテキストを様式ファイルで直接設定するには、引用符で囲み、改行文字を “n” で置き換えれば良いのですが、様式ファイルを非常に読み難くしてしまいます。テンプレートを様式ファイルに直接記述するか、テンプレートファイルに記述したものを様式ファイルから参照するかを決める際には、可読性を基準とするのが良いでしょう。様式ファイルの大きさや複雑さが高まる場合は、テンプレートテキストを記述するのではなく、外部ファイルに出してしまいましょう。

[5] 訳注：これは deviate な点ではない気が…

第12章　Managing change with Mercurial Queues

12.1　パッチ管理問題

ソフトウェアパッケージをソースからインストールする必要があるのに、パッケージ使用前に修正しておかなければならないバグをソース中に発見してしまう、というような事態はよくあることです。変更の後、暫くパッケージのことを忘れていると、数ヵ月後にパッケージを新しい版で更新する必要が出てきたとします。パッケージの新しい版が未だにバグを残していたなら、古い版のソースツリーから修正内容を抽出して、新しい版に適用しなければなりません。このような作業は退屈で間違いを起こしやすいものです。

これは“パッチ管理”問題の単純なケースです。自分では変更することができない“上流”のソースツリーがあるとします。上流のソースツリーの上でローカルな修正を行う必要があるなら、上流ソースの新しい版に対してローカルな修正を適用できるように、そういった修正を別途管理したいと思うでしょう。

パッチ管理問題はさまざまな状況で発生します。オープンソースソフトウェアプロジェクトのユーザが、プロジェクトのメンテナンス担当へ、バグ修正や新規機能をパッチ形式で送付する状況が、おそらく最もわかりやすい状況でしょう。

オープンソースソフトウェアを含むオペレーティングシステムの配布者は、配布するパッケージに対する変更を頻繁に行うので、自分たちの環境においてビルドを行うのは当然のことです。

整備の上で幾つか変更を行いたい場合、標準的な `diff` および `patch` プログラム（これらのツールに関する議論は12.4節を参照のこと）を使用して、単一のパッチを管理することは簡単です。しかし、一旦変更の数が増え始めると、単一のパッチの管理は関連性の無い“成果の塊”に感じ始めるため、例えば、単一のパッチは単一のバグ修正のみを含む（パッチは複数のファイルを修正するかもしれませんが、“単一の事”しか行わない）ようになるでしょうから、異なるバグやローカルな修正に必要とされるパッチを、いくつも抱えることになるかもしれません。このような状況で、上流のパッケージ保守担当者にバグ修正のパッチを送ったとすると、彼らはその後のリリースにおいてその修正を取り込むでしょうから、新しい版への更新の際には、そのパッチの適用を取りやめることができます。

上流のソースツリーに対して単一のパッチを保守することは、退屈で間違いやすいですが難しくはありません。しかし、保守しなければならないパッチの数が増えるにしたがい、問題の複雑さはすみやかに増加します。すくなからぬパッチを抱え込むことで、適用の有無を把握したり、それらを保守することが、「面倒なこと」から「圧倒されること」へと変化するでしょう。

幸いなことに、Mercurial は Mercurial Queues （あるいは単に“MQ”）と呼ばれる、パッチ管理問題を簡素化する強力な拡張機能を持っています。

12.2　Mercurial Queues 以前

1990 年代後半、何人かの Linux カーネル開発者達は、Linux カーネルの挙動を変える“パッチ系列”の保守を始めていました。幾つかの系列は安定性に、幾つかは網羅性に、その他の系列はより実験的な部分に焦点を当てていました。

これらのパッチのサイズは速やかに巨大化しました。2002 年、Andrew Morton が、自分のパッチキュー管理作業を自動化するのに用いていた、幾つかのシェルスクリプトを発表しました。Andrew は、Linux カーネルソース上での数百（時には数千）のパッチの管理に、これらのスクリプトを上手に利用していました。

12.2.1　A patchwork quilt（訳注：継ぎはぎの上掛け）

2003 年の初頭、Andreas Gruenbacher と Martin Quinson は、Andrew によるスクリプトの手法を取り入れて、“patchwork quilt” [AG] あるいは単に“quilt” (これについて述べた論文は [Gru05] を参照のこと) と呼ばれるツールを発表しました。パッチ管理が大幅に自動化されることから、quilt はオープンソース開発者の間で瞬く間に大きな支持を得ました。

quilt は、最上位のディレクトリにおいて**パッチのスタック**を管理します。管理開始の際には、quilt に対してディレクトリツリーを管理する旨と、どのファイルを管理したいのかを伝えます。quilt はこれらのファイルの名前と内容を別な場所に保存します。バグの修正の際には、新しいパッチを（単一のコマンドを使用して）作成し、修正する必要の有るファイルの編集を行い、パッチを “refresh” します。

refresh の段階で quilt はディレクトリツリーを走査します。quilt は実施された全ての変更でパッチを更新します。最上位のディレクトリにおいて作成した別なパッチを用いることで、“１つのパッチが適用されたツリー” から “２つのパッチが適用されたツリー” へと変化させるために必要な変更を、追跡することができます。

ツリーに対するパッチの適用状況を**変更**することもできます。パッチを “pop” すると、そのパッチによる変更はディレクトリツリーから取り除かれます。しかし、quilt はどのパッチが取り除かれたのかを覚えているので、取り除かれたパッチを再び “push” することができ、ディレクトリツリーには当該パッチによる変更が復元されます。最も重要な点は、“refresh” コマンドの実行と、それによる最上位のパッチの内容更新が任意の時点にできることです。これは、パッチの適用状況と、そのパッチによる変更内容の両方を、任意の時点で変更できることを意味します。

quilt は変更制御ツールを意識しないため、展開された tarball の最上位ディレクトリにおいても、Subversion リポジトリにおいても同等に機能します。

12.2.2　patchwork quilt から Mercurial Queues へ

2005 年中旬、quilt 的な振る舞いを Mercurial に追加するための、Mercurial Queues と呼ばれる拡張機能が、Chris Mason により実装されました。

quilt と MQ の大きな違いは、quilt が変更制御システムを意識しないのに対して、MQ が Mercurial に**統合**されていることです。push される個々のパッチは、Mercurial のチェンジセットとして表現されます。パッチを pop することで、チェンジセットは取り除かれます。

変更制御システムを意識しないことから、Mercurial と MQ を利用できない状況について知る上で、依然として quilt は非常に有用なソフトウェアです。

12.3　MQ の大きな利点

パッチと変更管理の統一を通して MQ が提供するものの価値を、誇張し過ぎることはありません。

フリーソフトウェアおよびオープンソースの世界でパッチが利用され続けるのは、変更管理ツールが年々その機能を向上させているにも関わらず、パッチが**軽快さ**を持っていることが大きな理由の一つです。

伝統的な変更制御ツールは、実施したことに関する全てを、永続的で取り消しの出来ないものとして記録します。この振る舞いに大きな価値がある一方で、幾分堅苦しくもあります。過激な実験を行おうとする場合、自分が行おうとすることに慎重になるか、必要とされない〜なお悪いことには、誤解や不安定の元となる〜失敗と間違いの記録を、永続的な履歴記録中に残す危険を冒す必要があります。

対照的に、MQ における分散履歴管理とパッチの結合により、あなたの作業を容易に隔離することができます。あなたのパッチは通常の変更履歴の上で存続し続け、望む時にそれらの実施／取り消しを行うことが出来ます。そのパッチが気に入らない場合、それを取りやめることができます。そのパッチが完全には望むものでない場合、望む姿に洗練させるまで、必要なだけ何度でも修正することが出来ます。

例えば、パッチと変更管理の統合により、パッチの理解とその効果〜および元になったコードとの連携〜のデバッグが、**非常に**簡単になります。全ての適用済みパッチが関連したチェンジセットを持っているので、どのチェンジセットとパッチがそのファイルに影響を及ぼしているのかを、“`hg log` *`filename`*” によって見ることが出来ます。`bisect` 拡張を用いることで、バグが持ち込まれたり修正された時点を見るために、全てのチェンジセットと適用済みパッチを通しての二分探索を行うことができます。“`hg annotate`” コマンドを用いることで、ソースファイルの特定の行を変更したのが、どのチェンジセットやパッチであるかを見ることが出来ます。

12.4　パッチの理解

MQ は、それがパッチ指向の特性を持つことを表に出しているため、パッチがどういったものであるかや、パッチとともに機能するツールに関することがらを理解する手助けになります。

伝統的な Unix の diff コマンドは、２つのファイルを比較し両者の違いを表示します。patch コマンドは、この違いをファイルに対する**変更**とみなします。これらのコマンドの簡単な動作例として、図 12.4 を見てください。

```
$ echo 'this is my first line' > oldfile
$ echo 'my first line is here' > newfile
$ diff -u oldfile newfile > tiny.patch
$ cat tiny.patch
--- oldfile        2009-07-20 21:58:49.000000000 +0000
+++ newfile        2009-07-20 21:58:49.000000000 +0000
@@ -1 +1 @@
-this is my first line
+my first line is here
$ patch < tiny.patch
patching file oldfile
$ cat oldfile
my first line is here
```

図 12.1: diff および patch コマンドの利用例

diff が生成する（そして、patch が入力する）ファイルの形式は “パッチ（patch）” ないし “差分（diff）” と呼ばれます。パッチと差分の間に違いはありません（以後は、より一般的に使用される “パッチ” という呼称を使用します）。

パッチファイルは、任意のテキストから始めることができます。patch コマンドはこのテキストを無視しますが、MQ はチェンジセットを生成する際のコミットメッセージとみなします。パッチ内容を開始を見つけるために、patch は “diff -” で始まる最初の行を探します。

MQ は *unified* 差分と共に機能します（patch はそれ以外の何種類かの差分形式でも機能しますが、MQ は *unified* 差分でないと機能しません）。unified 差分は２種類のヘッダを持っています。**ファイルヘッダ** *header* には、変更対象となるファイルのファイル名が記述され、patch コマンドが新規のファイルヘッダを見つけた際には、変更を行うために当該する名前のファイルを探します。

ファイルヘッダに続いて、*hunk* 列が記述されます。それぞれの hunk はヘッダで開始され、その hunk により変更される対象の、ファイルにおける行番号の範囲を識別します。ヘッダに続く hunk は、ファイルの改変されない部分からなる数行のテキストが前後に付加されます。これらの改変されない部分のことを、hunk に対する**コンテキスト**と呼びます。後続の hunk との間に少量のコンテキストしかない場合、diff は新たな hunk ヘッダを表示しません。変更内容の間に数行のコンテキスト行を置いて、hunk をそのまま続けます。

コンテキストの個々の行は空白文字で始まります。hunk 内部では、“-” で始まる行は “削除される行” を、“+” で始まる行は “挿入される行” を意味します。例えば、変更される行は、１行の削除と１行の挿入で表現されます。

パッチのより微妙な側面に関しては後ほど（ 12.6 節にて）説明しますが、MQ を利用するに当たってはここまでの知識で十分です。

12.5 Mercurial Queues の利用

MQ は Mercurial の拡張として実装されているので、利用の前に明示的に有効化する必要があります（ダウンロードの必要はありません。MQ は通常の Mercurial の配布物に含まれています）。MQ を有効にするには、~/.hgrc ファイルを編集し、 12.5 に示す行を追加してください。

```
[extensions]
hgext.mq =
```

図 12.2: MQ 拡張有効化のために ~/.hgrc に追加する内容

拡張が有効化されると、いくつかの新しいコマンドが有効化されます。“hg help” を使って “hg qinit” コマンドの利用可否を見ることで、拡張が機能することを確認できます。 12.3 の例を参照してください。

```
$ hg help qinit
hg qinit [-c]

init a new queue repository

    The queue repository is unversioned by default. If
    -c/--create-repo is specified, qinit will create a separate nested
    repository for patches (qinit -c may also be run later to convert
    an unversioned patch repository into a versioned one). You can use
    qcommit to commit changes to this queue repository.

options:

 -c --create-repo  create queue repository

use "hg -v help qinit" to show global options
```

図 12.3: MQ 利用可否の確認

MQ は**全ての** Mercurial リポジトリで利用でき、コマンドはそのリポジトリにしか作用しません。利用開始の際には、“hg qinit” コマンドによりリポジトリの準備を行います（ 12.4 参照）。このコマンドは、.hg/patches と呼ばれる空のディレクトリを作成し、MQ はこのディレクトリにメタデータを格納します。多くの Mercurial コマンドと同様、“hg qinit” コマンドは実行が正常に終了した場合には、特に何も表示しません。

```
$ hg init mq-sandbox
$ cd mq-sandbox
$ echo 'line 1' > file1
$ echo 'another line 1' > file2
$ hg add file1 file2
$ hg commit -m'first change'
$ hg qinit
```

図 12.4: MQ 利用に向けたリポジトリの準備

12.5.1 新しいパッチの作成

新しいパッチで作業を開始するには、“hg qnew” コマンドを使います。このコマンドは作成するパッチの名前を引数に取ります。例 12.5 に示すように、MQ はこれを .hg/patches ディレクトリ中の実ファイルの名前とみなします。

.hg/patches ディレクトリ配下にはそれ以外にも、series と status という２つの新しいファイルが作成されます。series は、そのリポジトリにおいて MQ が管理する全てのパッチの一覧を、１行１パッチで保持しています。status はそのリポジトリにおいて MQ が**適用**した全てのパッチを追跡するための、内部帳簿的な用途に使用されます。

```
$ hg tip
changeset:   0:5ddbf74425a0
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:51 2009 +0000
summary:     first change

$ hg qnew first.patch
$ hg tip
changeset:   1:a3c1798530eb
tag:         qtip
tag:         first.patch
tag:         tip
tag:         qbase
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:51 2009 +0000
summary:     [mq]: first.patch

$ ls .hg/patches
first.patch  series  status
```

図 12.5: 新しいパッチの作成

備考: 例えば、パッチの適用順序を変更したいような場合、series を手動で変更したい場合があるかもしれません。しかし、MQ の認識状況を容易に損なうことから、手動での status 編集は殆ど全ての場合において不適切です。

新しいパッチを作成したならば、普段と同じように作業領域ディレクトリのファイルを編集できます。“hg diff” や “hg annotate” といった、Mercurial の全ての通常コマンドはそれ以前と全く同様に機能します。

12.5.2 パッチの refresh

作業内容を保存する段階になったなら、作業中のパッチを更新するために “hg qrefresh” を使用します（図 12.5 参照）。このコマンドは、作業領域ディレクトリでの変更内容をパッチへと格納し、対応するチェンジセットを、それらの変更内容を保持するように更新します。

“hg qrefresh” コマンドはいつでも何度でも実行できるので、作業の “チェックポイント” として利用するのも良いでしょう。都合の良い時にパッチの refresh を実施することで、実験的な作業を行ってみて、それがうまく機能しない場合には、直近の refresh 時点までの変更を、“hg revert” コマンドにより取り消すことができます。

12.5.3 パッチの積み重ねと追跡

パッチに対する作業を終えるか、他のパッチに対する作業が必要になったなら、再度 “hg qnew” コマンドを実行することで、新しいパッチを作成します。Mercurial は、新規に作成したこのパッチを、既存のパッチの最上位に適用します。図 12.8 を参照してください。先に作業していたパッチに含まれる変更は、この新しいパッチの文脈の一部として含まれます（“hg annotate” 出力を見れば、このことは明らかです）。

これまでは、“hg qnew” と “hg qrefresh” を除いて、Mercurial の通常コマンドのみを使用するように注意してきました。しかし、図 12.5.3 に示すように、パッチに関する作業を行う際により便利な多くのコマンドを、MQ は提供しています。

- “hg qseries” コマンドは MQ が当該リポジトリ中で管理している全てのパッチの一覧を、古いものから新しいもの（最も最近**作成されたもの**）の順序で一覧表示します。

```
$ echo 'line 2' >> file1
$ hg diff
diff -r a3c1798530eb file1
--- a/file1        Mon Jul 20 21:58:51 2009 +0000
+++ b/file1        Mon Jul 20 21:58:52 2009 +0000
@@ -1,1 +1,2 @@
 line 1
+line 2
$ hg qrefresh
$ hg diff
$ hg tip --style=compact --patch
1[qtip,first.patch,tip,qbase]   b353a00003e1   2009-07-20 21:58 +0000   bos
  [mq]: first.patch

diff -r 5ddbf74425a0 -r b353a00003e1 file1
--- a/file1        Mon Jul 20 21:58:51 2009 +0000
+++ b/file1        Mon Jul 20 21:58:52 2009 +0000
@@ -1,1 +1,2 @@
 line 1
+line 2

```

図 12.6: パッチの refresh

- “hg qapplied” コマンドは、MQ が当該リポジトリで**適用した**全てのパッチの一覧を、古いものから新しいもの（最も最近適用されたもの）の順序で一覧表示します。

12.5.4 パッチの積み重ねの操作

“管理されている” パッチと “適用されている” それの間に違いがあることを、先の記述では暗に示していますが、実際に両者の間には違いがあります。MQ は適用すること無しに、パッチをリポジトリ中で管理することができます。

適用されたパッチは、リポジトリ中に対応するチェンジセットを持ち、パッチとチェンジセットの効果は作業領域ディレクトリにおいて見ることができます。“hg qpop” コマンドを使用して、パッチの適用を取り消すこともできます。

MQ は取り除かれたパッチを**管理**し続けますが、そのパッチはもはやリポジトリ中に対応するチェンジセットを持たず、作業領域ディレクトリにはパッチによる変更の痕跡は残されていません。図 12.10 に、適用されたパッチと追跡されているそれの違いを示します。

“hg qpush” コマンドを使用することで、未適用パッチの再適用、ないし取り除きを行うことができます。この操作によりパッチに対応する新しいチェンジセットが作成され、パッチによる変更は再び作業領域ディレクトリに現れます。図 12.11 に、“hg qpop” および “hg qpush” の実施例を示します。図のように 1 つないし 2 つのパッチを一度取り除いても、“hg qseries” の出力は変化しませんが、その一方で “hg qapplied” の出力は変化します。

12.5.5 複数パッチの適用 (push) および取り消し (pop)

“hg qpush” および “hg qpop” のそれぞれが、デフォルトでは一度に一つのパッチに対して処理を行う一方で、一度に複数のパッチの適用や取り消しを行うこともできます。“hg qpush” に -a オプションを指定することにより、全ての未適用パッチの適用が、“hg qpop” に -a オプションを指定することにより、全ての適用済みパッチの取り消しを行うことができます。（それ以外の複数パッチの適用／取り消しの方法に関しては、 12.7 節を参照してください。）

```
$ echo 'line 3' >> file1
$ hg status
M file1
$ hg qrefresh
$ hg tip --style=compact --patch
1[qtip,first.patch,tip,qbase]   9eb545de58b2   2009-07-20 21:58 +0000   bos
  [mq]: first.patch

diff -r 5ddbf74425a0 -r 9eb545de58b2 file1
--- a/file1        Mon Jul 20 21:58:51 2009 +0000
+++ b/file1        Mon Jul 20 21:58:52 2009 +0000
@@ -1,1 +1,3 @@
 line 1
+line 2
+line 3

```

図 12.7: 複数回のパッチ refresh による変更の蓄積

12.5.6 安全確認とその無効化

いくつかの MQ コマンドは、処理の前に作業領域ディレクトリの確認を行い、何らかの改変が検出された場合には処理を中断します。この確認は、パッチに取り込まれていない変更内容を失わないために行われます。図 12.13 に例を示します。“hg qnew” コマンドは未取り込みの変更（このケースでは file3 の “hg add” に起因するもの）がある場合、新しいパッチを生成しません。

作業領域ディレクトリを確認するコマンドは、すべて “了解済み” オプションを取ることができ、そのオプションは常に -f と名づけられています。-f オプションの厳密な意味はコマンドごとに異なります。例えば、“hg qnew -f” は新たに生成されるパッチに未取り込みの変更を全て取り込みますが、“hg qpop -f” は取り消されるパッチが影響を及ぼすファイルに対する変更を元に戻します[1]。利用する前に各コマンドの -f オプションのドキュメントを確認しましょう！

12.5.7 複数パッチの一括処理

“hg qrefresh” コマンドは、常に**最上位の**適用済みパッチを更新します。これは、あるパッチに対する操作を（refresh することで）中断し、取り消し（pop）ないし適用（push）により別のパッチを最上位に持ってくることで、そのパッチに対して作業することができることを意味します。

この機能によって可能になることを例によって示します。２つのパッチによって新しい機能を開発しているものとしましょう。１つ目のパッチはソフトウェアの中核機能の変更を、そして２つ目のパッチは — １つ目のパッチの上で — 中核機能の変更を使用するためのユーザーインタフェース (UI) の変更を行います。UI へのパッチの作業中に、中核機能へのパッチにバグを見つけたとしても、それを修正するのは簡単なことです。UI へのパッチに対する “hg qrefresh” により作業中の変更を保存した後に、“hg qpop” により操作対象パッチを中核機能へのそれに変更します（パッチスタックを下へと移動します）。中核機能へのパッチのバグを修正し、“hg qrefresh” によってパッチへの反映を行った後に、“hg qpush” により操作対象パッチを UI へのパッチに戻すことで、やりかけの作業を継続することができます。

12.6 パッチに関して更に詳しく

MQ はパッチの適用に GNU patch コマンドを使用しますので、patch コマンドの動作とパッチそのものに関して、より詳細な情報を知ることは有用です。

[1]訳注: 「パッチの影響を元に戻す」のではなく、「パッチが影響を及ぼすファイル」を全て元に戻す、の意

```
$ hg qnew second.patch
$ hg log --style=compact --limit=2
2[qtip,second.patch,tip]   f44a4270d8a8   2009-07-20 21:58 +0000   bos
  [mq]: second.patch

1[first.patch,qbase]   9eb545de58b2   2009-07-20 21:58 +0000   bos
  [mq]: first.patch

$ echo 'line 4' >> file1
$ hg qrefresh
$ hg tip --style=compact --patch
2[qtip,second.patch,tip]   90c5baabf0e2   2009-07-20 21:58 +0000   bos
  [mq]: second.patch

diff -r 9eb545de58b2 -r 90c5baabf0e2 file1
--- a/file1        Mon Jul 20 21:58:52 2009 +0000
+++ b/file1        Mon Jul 20 21:58:53 2009 +0000
@@ -1,3 +1,4 @@
 line 1
 line 2
 line 3
+line 4

$ hg annotate file1
0: line 1
1: line 2
1: line 3
2: line 4
```

図 12.8: １つ目の上に積み重ねられる２つ目のパッチ

12.6.1 除去数

パッチのファイルヘッダを見ると、実際のパス名には現れない余分な要素を先頭に持っていることに気が付くでしょう。これは以前にパッチが生成されていた方法の名残です（今でもこの方法を用いていますが、近年の構成管理ツールでは稀です）。

Alice が tarball を展開してファイルを編集した後で、パッチを作成しようと考えたとします。作業領域ディレクトリを改名し、再度 tarball を展開（この展開のために改名することが必要になります）し、`diff` コマンドに`-r` および `-N` オプションを指定することで、改変前のディレクトリと改変後のディレクトリの間で再帰的にパッチを生成します。一方には改変前のディレクトリ名が全てのファイルのパス冒頭に付加され、他方には改変後のディレクトリ名が同様に付加されます。

Alices からパッチを受け取った人物の環境に、改変前と改変後ディレクトリの両方と厳密に一致する名前のディレクトリがある、というのはありそうもない事ですから、`patch` コマンドは、パッチ適用時にパス名要素の何番目までを取り除くかを指す`-p` オプションを持っています。このオプションに指定される数を**除去数**（strip count）と呼びます。

“`-p1`” オプションは、“除去数を１とみなす” ことを意味します。`patch` コマンドが、ファイルヘッダにおいてファイル名 `foo/bar/baz` を検知した場合、`foo` 部分を除去した `bar/baz` というファイルに対してパッチをあてます（厳密なことを言えば、除去数は除去される**パス区切り**（およびそれに付随する要素）の数を指します。除去数１は、`foo/bar` を `bar` にしますが、`/foo/bar`（先頭のスラッシュに注意）は `foo/bar` になります）。

パッチにおける “標準の” 除去数は１ですので、ほとんど全てのパッチは取り除かれる先頭要素を１つ含んでいます。Mercurial の “`hg diff`” コマンドはこの形式でパス名を生成しますので、“`hg import`” コマンドや MQ は除去数１のパッチを期待しています。

除去数が１ではないパッチをパッチキューに追加しようとした場合、現時点で `-p` オプションを持っていない “`hg qimport`” （ Mercurial バグ番号 311 参照のこと）では取り込むことができません。その場合、“`hg qnew`” で新規パッチを MQ 上に作成し、“`patch -pN`” によりパッチを適用、“`hg addremove`” でパッチにより追加／削除されたファイ

```
$ hg qseries
first.patch
second.patch
$ hg qapplied
first.patch
second.patch
```

図 12.9: “hg qseries” および “hg qapplied” によるパッチの積み重ねの習得

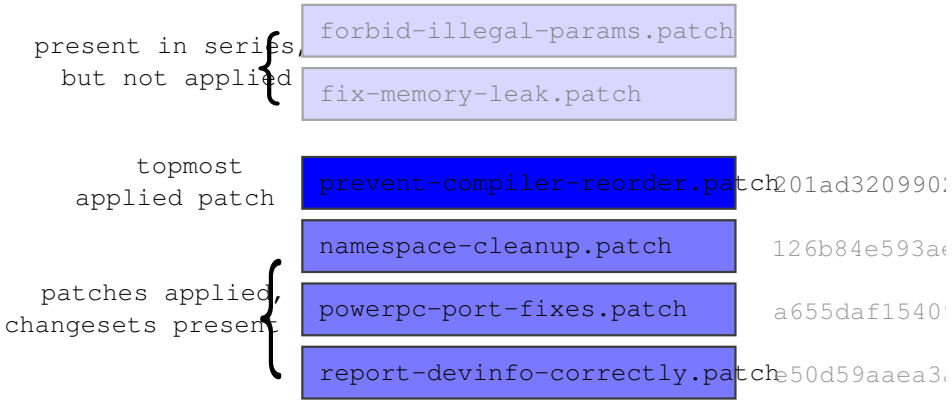

図 12.10: MQ のパッチの積み重ねにおける適用済みパッチと未適用パッチ

ルを特定し、“hg qrefresh” を行うのが最善の方法です。このような面倒な手順はいずれ不要になるかもしれません。詳細は Mercurial バグ番号 311 を参照してください。

12.6.2 パッチ適用手順

patch が hunk を適用する際には、it tries a handful of (successively はどこに掛かる？) successively less accurate strategies to try to make the hunk apply XXXXX 用心深いこの方法により、古い版のファイルで生成されたパッチであっても、新しい版のファイルに適用することが、多くの場合で可能となります。

patch コマンドは、最初は hunk における行番号、コンテキストおよび変更対象テキストの厳密一致を試みます。厳密一致ができない場合、行番号に関する情報を無視し、コンテキストのみの厳密一致を試みます。これが成功した場合、patch コマンドは、hunk が適用されたことと、元の行番号から**オフセット分**ずれていることを表示します。

コンテキストのみによる一致が失敗した場合、patch は冒頭および末尾行を取り除いたコンテキストを用いて、**縮小**コンテキストのみによる一致を試みます。縮小コンテキストによる hunk 適用が成功した場合、**あいまいな要因**を元に hunk が適用されたことを表示します（この時示される数値は、patch コマンドがパッチ適用前にコンテキストから取り除いた行数です）。

これらのどの技法でも適用できない場合、patch コマンドは争点となっている hunk が却下された旨を表示します。patch コマンドは却下された hunk （単に “reject” とも呼ばれます）を同名で .rej 拡張子を持つファイルに保存します。更にその上で、パッチ適用前のファイルのコピーを .orig 拡張子付きで保存します。拡張子無しのファイルは、適切にの適用**された** hunk による変更を含んでいます。ファイル foo を変更する６つの hunk を持つパッチがあり、そのうちの１つが適用できなかった場合、変更前の内容を持つ foo.orig、適用できなかった hunk を１つ持つ foo.rej および適用できた５つの hynk による変更を含む foo の３つのファイルができます。

12.6.3 パッチの実現上の癖

patch コマンドのファイルへの作用を知る上で、有用な事がいくつかあります。

- わかりきった事ですが、patch はバイナリファイルを扱えません。

```
$ hg qapplied
first.patch
second.patch
$ hg qpop
now at: first.patch
$ hg qseries
first.patch
second.patch
$ hg qapplied
first.patch
$ cat file1
line 1
line 2
line 3
```

図 12.11: 適用パッチの積み重ねの変更

```
$ hg qpush -a
applying second.patch
now at: second.patch
$ cat file1
line 1
line 2
line 3
line 4
```

図 12.12: 全ての未適用パッチの適用

- 実行ビットも扱えませんので、新しいファイルを作成する際には、読み取り可能にはしますが、実行可能にはしません。
- patch は、削除対象ファイルと空ファイルの差分をもって、ファイルの削除を表します。そのため、“ファイルを削除する” ことは、パッチにおいては “全ての行が削除される” ように見えます。
- 空のファイルと追加対象ファイルの差分をもって、ファイルの追加を表します。そのため、“ファイルを追加する” ことは、パッチにおいては “全ての行が追加される” ように見えます。
- 古い名前のファイルの削除と新しい名前のファイルの追加をもって、ファイルの改名を表します。これは、ファイルの改名を行うパッチのサイズ（footprint）が大きくなることを意味します（パッチにおけるファイルの改名やコピーを Mercurial が推測することは、現状では行われないことにも留意してください）。
- patch は空のファイルを表現できませんので、“空のファイルをツリーに追加する” ことをパッチで表現することは出来ません。

12.6.4 あいまいさに注意

オフセット付きや、あいまいな要因を元にしている場合であっても、パッチの適用は完全に成功することが多いのですが、一方でこのような厳密性を欠いた適用手法は、おのずとファイルへのパッチ適用が不完全である可能性を残してしまいます。最も典型的な事例は、パッチを２度適用してしまうことや、不適切な位置に適用してしまうことです。patch や “hg qpush” がオフセットやあいまい要因に関して言及した際には、ファイルが適切に変更されていることを後から確認してください。

```
$ echo 'file 3, line 1' >> file3
$ hg qnew add-file3.patch
$ hg qnew -f add-file3.patch
abort: patch "add-file3.patch" already exists
```

図 12.13: 強制的なパッチの生成

オフセット付きや、あいあまいな要因を元に適用されたパッチを refresh するのが、多くの場合においておすすめなのは、パッチの refresh が、パッチを綺麗に適用するための新しいコンテキスト情報を生成するからです。ただし、パッチを refresh することで、元ファイルの異なる版に対してパッチの適用が失敗するようになる場合があるため、“多くの場合” おすすめですが、“常に” ではありません。ソースツリーの複数の版に対して適用可能なパッチを保守するような場合、パッチ適用処理の結果を検証する機会を得ることが出来るので、パッチにあいまい要因を持たせておくのは許容範囲です。

12.6.5 却下された hunk の取り扱い

パッチの適用に失敗すると、“hg qpush” はエラーメッセージを表示して終了します。.rej ファイルが残されている場合、それ以上のパッチを push したり他の作業をする前に、却下された hunk の修正を行うことが一般的には最善です。

パッチの適用対象であるソースの更新により、**それまでは**きちんと適用できていたパッチが適用できなくなった場合の Mercurial Queues の使い方の詳細に関しては、 12.8 節を参照してください。

残念なことに、却下された hunk を扱うための決定的な技法は存在しません。多くの場合、.rej ファイルを参照しながら、対象ファイルを編集し、却下された hunk を手動で適用しなければなりません。

思い切った事も辞さないのであれば、パッチの適用に関しては patch よりも強力な、wiggle [Bro] と呼ばれるツールが、Linux カーネルハッカーの Neil Brown により書かれています。

patch により却下された hunk の適用を自動化するために、簡便な手法を用いる mpatch [Mas] と呼ばれるツールも、別の Linux カーネルハッカーの Chris Mason （Mercurial Queues の作者です）により書かれています。mpatch は、４つのよくある理由で却下された hunk の適用を助けることができます。

- hunk 中程のコンテキストが変更された。
- hunk のコンテキストの、先頭あるいは末尾の一方が見当たらない。
- 大きな hunk よりも—全部なり一部なりが— 小さな hunk に分割された方が適用しやすい。
- 現時点でのファイルとわずかに内容の異なる行を hunk が削除しようとしている。

wiggle ないし mpatch を使用する際には、実施結果に対して二重に注意を払う必要があります。実のところ mpatch は、処理の完了時に自動的にマージプログラムへと誘導することで、ツール出力の二重確認の手法を強要していますので、mpatch の実行結果を確認し、残されたマージ処理を完了させることが出来ます。

12.7 MQ で最高性能を出すために

MQ は大量のパッチの取り扱いを効率よく実施します。2006 EuroPython conference [O’S06] での講演のために、2006 年中旬に性能実験を実施しました。適用パッチとして、1,738 個のパッチを持つ Linux 2.6.17-mm1 パッチ系列を使用しています。Linux 2.6.12-rc2 から Linux 2.6.17 にかけての、27,472 のリビジョン全てを持つ Linux カーネルリポジトリに対して、これらのパッチを適用したのです。

旧式の遅いラップトップ PC 上で、1,738 個のパッチ全てを “hg qpush -a” するのに 3.5 分、それらを “hg qpop -a” するのに 30 秒かかりました（新しいラップトップなら、全てのパッチを push する時間は 2 分まで下がりました）。

最も大きなパッチの１つ（22,779 行の変更を 287 のファイルに対して行います）を 6.6 秒で “hg qrefresh” できています。

MQ が巨大なソースツリーで作業するのに適しているのは明らかですが、最高の性能を出すために知っておいたほうが良い幾つかのコツがあります。

最初のコツは、“一括” 操作を行うことです。“hg qpush” および “hg qpop” の実行の際には、何ら変更がされていないことと、“hg qrefresh” し忘れがないことを確認するために、常に作業領域ディレクトリを走査しています。小さなソースツリーの場合は、この走査に要する時間は気になりません。しかし、中程度（10,000 ファイル程度）のソースツリーでは、１秒からそれ以上の時間が必要です。

“hg qpush” および “hg qpop” コマンドでは、複数パッチを一括して push および pop する際に、作業を切り上げる “到達パッチ” を指定することができます。到達パッチ指定付きで実行することで、“hg qpush” は指定したパッチが適用スタックの最上位になるまでパッチの適用を行います。“hg qpop” の場合は、到達パッチが適用スタックの最上位になるまでパッチの取り消しを行います。

到達パッチの指定には、パッチの名前か数値が使用できます。数値指定の場合、パッチは０から数え始めるため、最初のパッチは０、次のパッチの１となります。

12.8　元ソース変更時のパッチの更新

直接変更することのできないリポジトリに対して、パッチスタックを持つことはよくある事です。第三者のソースに対する変更や、元ソースの更新頻度よりも開発に時間の掛かる機能を実装している場合、元ソースの更新との同期や、適用できなくなったパッチの hunk を修正する必要があります。このような作業は、パッチ系列の**リベース**と呼ばれます。

リベースの一番単純な方法は、パッチに対して “hg qpop　-a” を行い、“hg pull” で元ソースの変更をリポジトリに取り込み、最後に “hg qpush　-a” でパッチを再適用します。MQ によるパッチ適用では、衝突が検出されている間は適用できないパッチの適用を止めることで、衝突の解消とパッチの “hg qrefresh” を行う機会を設けつつ、パッチスタック中の全てのパッチを更新し終わるまでパッチの適用を継続します。

元ソースの変更がパッチの適用具合に悪影響を及ぼす心配が無いのであれば、この手法は手軽で且つ上手く機能するでしょう。しかしながら、元ソースで頻繁に更新される部分に触れるようなパッチスタックの場合、却下された hunk の手動での修正は、すぐにでも面倒な作業と化すでしょう。

リベース処理を部分的に自動化する事は可能です。元ソースの幾つかのリビジョンに対してきちんと適用できるパッチであれば、異なるリビジョンとパッチとの間での衝突に対して、事前の適用情報を用いた解消を MQ により行うことができます。

手順は少々込み入っています。

1. 開始に当たって、パッチがきちんと適用できている最上位リビジョンに対して “hg qpush -a” により全てのパッチを適用します。
2. “hg qsave　-e　-c” を用いてパッチディレクトリのバックアップを保存します。このコマンドの実行の際には、パッチを保存したディレクトリの名前を表示します。*N* を小さい整数とした場合、.hg/patches.*N* という形式の名前のディレクトリにパッチが保存されます。適用されたパッチ以外に、“保存されたチェンジセット” もコミットしますが、これは内部的な情報と、series および status の状態を記録するためです。
3. hgcmdpull により、更新をリポジトリに取り込みます（“hg pull -u” を用いない理由は、以降の記述を参照してください）。
4. “hg update -C” を用いて最新の tip リビジョンに更新することで、適用したパッチを無効にしてください。
5. “hg qpush　-m　-a” を用いて全てのパッチをマージします。“hg qpush” への　-m オプション指定により、パッチ適用に失敗した際に、MQ は 3-way マージを実施します。

“hg qpush　-m” 実施の際には、series ファイルに列挙されたそれぞれのパッチは通常通り適用されます。あいまい要因を元にパッチが適用されたり、パッチの適用が却下された場合、MQ は “hg qsave” により保存されたパッチ

キューを参照し、パッチに対応するチェンジセットを用いた 3-way マージを行います。このマージ処理には Mercurial の通常のマージ機構が利用されますので、衝突の解消の際には GUI マージツールが起動されるかもしれません。

パッチの影響を解消し終えると、マージ結果を元に MQ によるパッチの refresh が行われます。

この手順を終えたリポジトリには、古いパッチキューに相当するチェンジセットを元にした余分な head と、`.hg/patches.N` に保存された古いパッチキューが残ります。余分な head の削除は、“`hg qpop -a -n patches.N`” ないし “`hg strip`” で行うことができます。バックアップとしての必要性がなくなったなら、`.hg/patches.N` も削除してしまって構いません。

12.9 パッチの指定

パッチを操作する MQ コマンドにおけるパッチの指定は、パッチの名前か数値で行います。名前による指定は非常にわかりやすいでしょう。例えば、“`hg qpush`” コマンドへの `foo.patch` の指定により、`foo.patch` が適用されるまでパッチの適用が繰り返されます。

短縮形式として、名前と数値オフセットの両方を指定することもできます。`foo.patch-2` は “`foo.patch` パッチの２つ前” を、`bar.patch+4` は “`bar.patch` パッチの４つ後ろ” を意味します。

数値によるパッチの指定はそれほど難しくありません。“`hg qseries`” により最初に表示されるパッチは０、２番目は１、となっています（そう、０から数え始める仕組みです）。

MQ は、通常の Mercurial コマンドの利用時におけるパッチ操作も簡便にします。チェンジセット識別子を受け付ける全てのコマンドは、適用済みのパッチ名も受け付けます。リポジトリ中に元々あった通常のタグに加えて、パッチ適用の際の起点となるリビジョンにタグ[2]が付与されます。それに加えて、`qbase` および `qtip` タグにより、最下位および最上位の適用ずみパッチをそれぞれ指定できます。

Mercurial の通常タグに対するこれらの拡張は、パッチの取り扱いをより簡便にします。

- 最新の一連の変更を元に、メーリングリストへパッチ爆弾（patchbomb）を投稿したい場合には？

```
1 hg email qbase:qtip
```

（“パッチ爆弾” については 14.4 節を参照してください）

- `foo.patch` 以降のパッチで、特定のディレクトリ配下のファイルに関与しているものを、全て知りたい場合には？

```
1 hg log -r foo.patch:qtip subdir
```

パッチの名前を利用可能にするために、MQ は Mercurial の持つ内部タグ機能を使用しているので、パッチを名前で指定する場合には、その名前を全て入力する必要はありません。

パッチの名前をタグで実現することで、“`hg log`” コマンドの実行時に、その出力の一部としてタグとしてのパッチ名が表示される、という副作用も得られます。このことにより、適用済みのパッチと “通常の” リビジョンを、視覚的に識別することを容易にします。適用済みパッチと連携する Mercurial の通常コマンドの実行例を図 12.14 に示します。

12.10 知っておくと便利な事柄

MQ の利用に関して、独立した節を設ける程ではないものの、知っておいたほうが良い事柄が幾つかあります。ここでは、そういった事柄を集めてみました。

- “`hg qpop`” でパッチを取り消した後に、“`hg qpush`” で再度適用した場合、その時点での適用済みパッチに相当するチェンジセットは、pop/push する前のチェンジセットとは**異なる識別子**を持ちます。識別子が異なる理由は **??** 節を参照してください。

[2] `qparent`

```
$ hg qapplied
first.patch
second.patch
$ hg log -r qbase:qtip
changeset:   1:2c9f9a14ed72
tag:         first.patch
tag:         qbase
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:48:24 2009 +0000
summary:     [mq]: first.patch

changeset:   2:1b8fcc30cf3e
tag:         qtip
tag:         second.patch
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:48:24 2009 +0000
summary:     [mq]: second.patch

$ hg export second.patch
# HG changeset patch
# User Bryan O'Sullivan <bos@serpentine.com>
# Date 1248126504 0
# Node ID 1b8fcc30cf3e3d7640a082057402061e7832301f
# Parent  2c9f9a14ed7225ad61f5178a726f1f72da46528e
[mq]: second.patch

diff -r 2c9f9a14ed72 -r 1b8fcc30cf3e other.c
--- /dev/null        Thu Jan 01 00:00:00 1970 +0000
+++ b/other.c        Mon Jul 20 21:48:24 2009 +0000
@@ -0,0 +1,1 @@
+double u;
```

図 12.14: MQ のタグ機能を使用したパッチの操作

- 少なくとも、パッチスタック上のパッチによるチェンジセット群の “パッチ性” を保ちたいのであれば、他のブランチとそれらを “hg マージ” すべきではありません。“hg マージ” した場合、それ自体は成功するでしょうが、結果として MQ が混乱してしまうでしょう。

12.11 リポジトリにおけるパッチの管理

MQ が利用する .hg/patches ディレクトリが Mercurial の作業領域ディレクトリの外にあるため、MQ の “下にある”Mercurial のリポジトリは、パッチの管理や存在に関して何も認識していません。

このことは、パッチディレクトリの内容をそれ自身の Mercurial リポジトリを用いて管理できる、という興味深い可能性をもたらします。例えば、パッチに関する作業を行い、“hg qrefresh” をした後で、パッチの現状を “hg commit” することで、後からその状態へとパッチを “巻き戻す”（roll back）することができるなど、有用な機能を提供します。

複数のリポジトリの間で、同一パッチスタックの異なる版を共有することも出来ます。筆者は Linux カーネル機能の開発の際にこの手法を使用しています。複数の CPU アーキテクチャごとにそれぞれ真新しいカーネルソースのコピーを用意し、それぞれに作業中のパッチを含むリポジトリを複製します。別なアーキテクチャで変更内容の試験を行う際には、対応するカーネルソースのパッチリポジトリへ現時点のパッチを push し、全てのパッチを最適用（pop 後に push）した後に、そのカーネルのビルドおよび試験を行います。

リポジトリ形式の上でパッチを管理することで、適用対象のソースに対する制御の可否に関わり無く、開発者同士でお互いに衝突すること無しに、同じパッチ系列に対する作業を実施できます

12.11.1　MQ のパッチリポジトリサポート

MQ は .hg/patches ディレクトリを自身のリポジトリとして、パッチ操作を補助しますが、“hg qinit” での初期化の際に -c オプションを指定することで、.hg/patches ディレクトリを Mercurial リポジトリとして作成することが出来ます。

> **備考:**　-c オプションの指定を忘れた場合、任意の時点で .hg/patches ディレクトリで “hg init” を実行してください。status を履歴管理しようと思うことは**本当に**ありませんから、.hgignore ファイルに status を追加するのを忘れないでください（“hg qinit　-c” は、この作業を自動的に行います）。

利便性上、.hg/patches ディレクトリが Mercurial リポジトリである場合、MQ は作成・取り込みを行ったパッチの全てを自動的に “hg add” します。

最後になりますが、MQ は .hg/patches において “hg commit” を実行する短縮コマンド “hg qcommit” を提供していますので、（ディレクトリ移動等の）煩わしいキー入力が省略できます。

12.11.2　幾つかの注意点

MQ によるパッチのリポジトリ管理のサポートは、限定的なものです。

MQ は、パッチディレクトリに対して行われた変更を、自動的に検出することはできません。“hg pull” の実行や、手動での編集、あるいは “hg update” の実行によるパッチや series の変更を行った場合、パッチ適用対象のリポジトリにおいて “hg qpop　-a” の後に “hg qpush　-a” を行って、それらの変更を有効にする必要があります。この作業を忘れた場合、MQ は適用されているパッチがどれなのか混乱してしまうでしょう。

12.12　パッチ操作のためのサードパーティー製ツール

暫くの間、パッチを使った作業をしていると、扱っているパッチの解釈や操作を補助するツールが、欲しくてたまらなくなっているに違いありません。

diffstat コマンド [Dic] は、パッチによって各ファイルがどれだけ変更されるかを表すヒストグラムを生成します。どのファイルが、どの程度の影響を受けるのか、といった全体的な “感覚を掴む” には良い方法です（diffstat の -p オプション利用は勿論良いのですが、ファイル名の前置詞に対して行う -p オプションの巧妙な処理は、少なくとも筆者にとってはわかりにくいです）。

patchutils パッケージ [Wau] は貴重な存在です。このパッケージは、“Unix の理念” に従って、それぞれがパッチに対して単一の処理を行う小さなツールの集まりです。patchutils の中で筆者が最も利用しているのは、パッチファイルから一部を展開する filterdiff です。例えば、あるパッチが数ダースのディレクトリに渡って数百のファイルを変更する場合、filterdiff を起動することで、指定したパターンに名前が合致するファイルにだけ変更を行う、小さなパッチを生成することが出来ます。それ以外の例については、 13.9.2 節を参照してください。

12.13　パッチを扱う良い方法

一連のパッチが、フリーソフトウェアやオープンソースプロジェクトへ送付するものであろうと、あなたの作業における定期的な変更手続きとみなされるものであろうとも、より良く作業するための、簡単に利用できる手法があります。

まずは、パッチに説明的な名前をつけましょう。例えば rework-device-alloc.patch といった名前は、そのパッチが何を行うものかというヒントをすばやく与えてくれるので、良い名前と言えるでしょう。名前は長くても問題にはなりません。名前を入力することはそれほど多くはないでしょうが、“hg qapplied” や “hg qtop” といったコマンドは、何度も何度も実行するものですから。多くのパッチを扱う場合や、多くの異なるタスクに手一杯でパッチに多くの注意を割けないような場合、名前の適切さはとりわけ重要です。

次に、どのパッチに対して作業しているのかに注意しましょう。“hg qtop” コマンドを—例えば、“hg tip -p” を指定しつつ— 使用して頻繁にパッチの名前を見ることで、どんな作業をしているのかを確認しましょう。筆者は作業

```
$ diffstat -p1 remove-redundant-null-checks.patch
 drivers/char/agp/sgi-agp.c        |    5 ++---
 drivers/char/hvcs.c               |   11 +++++++------
 drivers/message/fusion/mptfc.c    |    6 ++-----
 drivers/message/fusion/mptsas.c   |    3 +--
 drivers/net/fs_enet/fs_enet-mii.c |    3 +--
 drivers/net/wireless/ipw2200.c    |   22 ++++++----------------
 drivers/scsi/libata-scsi.c        |    4 +---
 drivers/video/au1100fb.c          |    3 +--
 8 files changed, 19 insertions(+), 38 deletions(-)
$ filterdiff -i '*/video/*' remove-redundant-null-checks.patch
--- a/drivers/video/au1100fb.c~remove-redundant-null-checks-before-free-in-drivers
+++ a/drivers/video/au1100fb.c
@@ -743,8 +743,7 @@ void __exit au1100fb_cleanup(void)
 {
         driver_unregister(&au1100fb_driver);

-        if (drv_info.opt_mode)
-                kfree(drv_info.opt_mode);
+        kfree(drv_info.opt_mode);
 }

 module_init(au1100fb_init);
```

図 12.15: diffstat、filterdiff および lsdiff コマンド

中に何度も意図しないパッチに対して “hg qrefresh” を実行してしまったことがありますが、間違ったパッチに取り込んでしまった変更を正しいパッチに移動させるのは、往々にして手のかかるものです。

上記の理由から、 12.12 節で紹介している diffstat や filterdiff のようなサードパーティー製ツールの学習に、少しでも良いので時間を費やすべきです。前者はパッチの及ぼす変更に関してすばやい見解を得ることが、後者はパッチ中の hunk を選択的に継ぎ合わせて異なるパッチに組み上げることができます。

12.14 MQ クックブック

12.14.1 “些細な” パッチの管理

真新しい Mercurial リポジトリにファイルを投入するのは、非常にオーバーヘッドが低いので、単にダウンロードしたソース tarball に対して変更を加えるのだとしても、MQ によりパッチ管理を行うことは非常に理にかなっています。

まずはソース tarball のダウンロードと展開を行い、Mercurial リポジトリに投入します。

```
$ download netplug-1.2.5.tar.bz2
$ tar jxf netplug-1.2.5.tar.bz2
$ cd netplug-1.2.5
$ hg init
$ hg commit -q --addremove --message netplug-1.2.5
$ cd ..
$ hg clone netplug-1.2.5 netplug
updating working directory
18 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

次にパッチスタックを作成し、変更を行います。

```
$ cd netplug
$ hg qinit
$ hg qnew -m 'fix build problem with gcc 4' build-fix.patch
$ perl -pi -e 's/int addr_len/socklen_t addr_len/' netlink.c
$ hg qrefresh
$ hg tip -p
changeset:   1:98dfa9fc14a2
tag:         qtip
tag:         build-fix.patch
tag:         tip
tag:         qbase
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Mon Jul 20 21:58:50 2009 +0000
summary:     fix build problem with gcc 4

diff -r c57121a805d2 -r 98dfa9fc14a2 netlink.c
--- a/netlink.c        Mon Jul 20 21:58:49 2009 +0000
+++ b/netlink.c        Mon Jul 20 21:58:50 2009 +0000
@@ -275,7 +275,7 @@
          exit(1);
      }

-    int addr_len = sizeof(addr);
+    socklen_t addr_len = sizeof(addr);

      if (getsockname(fd, (struct sockaddr *) &addr, &addr_len) == -1) {
          do_log(LOG_ERR, "Could not get socket details: %m");

```

数週間から数ヵ月経ってから、そのパッケージの著者が新しい版をリリースしたとします。まずはリポジトリに変更を取り込みます。

```
$ hg qpop -a
patch queue now empty
$ cd ..
$ download netplug-1.2.8.tar.bz2
$ hg clone netplug-1.2.5 netplug-1.2.8
updating working directory
18 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd netplug-1.2.8
$ hg locate -0 | xargs -0 rm
$ cd ..
$ tar jxf netplug-1.2.8.tar.bz2
$ cd netplug-1.2.8
$ hg commit --addremove --message netplug-1.2.8
```

上記手順で "hg locate" により始まるパイプラインは、作業領域ディレクトリ中の全てのファイルを削除しますので、"hg commit" の --addremove オプションは、新しい版においてどのファイルが本当に追加／削除されたのかを判定できます。

最後に、新しくなったソースツリーの最上位でパッチを適用します。

```
$ cd ../netplug
$ hg pull ../netplug-1.2.8
pulling from ../netplug-1.2.8
searching for changes
adding changesets
adding manifests
```

```
adding file changes
added 1 changesets with 12 changes to 12 files
(run 'hg update' to get a working copy)
$ hg qpush -a
(working directory not at a head)
applying build-fix.patch
now at: build-fix.patch
```

12.14.2 パッチ全体の結合

MQ はパッチ全体を結合する “hg qfold” コマンドを提供しています。このコマンドは、名前を指定したパッチを指定した順序で、最上位の適用済みパッチへと “結合” し、それらの説明文を最上位パッチの説明文末尾へ追加します。結合対象のパッチは、結合の時点で未適用でなければなりません。

パッチの結合順序は重要です。最上位の適用済みパッチが foo で、そこに “hg qfold” と quux を “hg qfold” する場合、順に foo、bar そして quux と適用するのと同じ効果を持つパッチができあがります。

12.14.3 パッチの一部の他のパッチへの併合

パッチの**一部**を他のパッへ併合するのは、パッチ全体を結合するよりも面倒です。

あるファイル（群）に対する変更全体を移動したい場合、filterdiff の -i および-x オプションを用いることで、パッチから切り出す変更点を選択して、その結果を併合先パッチへと取り込むことでができます。通常は取り込み元となったパッチそのものは変更したくないものです。そこで、MQ は取り込み元パッチを “hg qpush” する際に、取り込まれた分の hunk が拒否されたことが報告されますから、“hg qrefresh” でパッチを更新することで、重複した hunk を取り除くことができます。

１つのファイルに対する複数の hunk を持つパッチの一部だけが欲しい場合、事態はもう少し厄介ですが、それでも部分的に自動化することができます。“lsdiff -nvv” を使うことで、パッチに関するメタデータを表示させます。

```
$ lsdiff -nvv remove-redundant-null-checks.patch
22        File #1          a/drivers/char/agp/sgi-agp.c
        24        Hunk #1        static int __devinit agp_sgi_init(void)
37        File #2          a/drivers/char/hvcs.c
        39        Hunk #1        static struct tty_operations hvcs_ops =
        53        Hunk #2        static int hvcs_alloc_index_list(int n)
69        File #3          a/drivers/message/fusion/mptfc.c
        71        Hunk #1        mptfc_GetFcDevPage0(MPT_ADAPTER *ioc, in
85        File #4          a/drivers/message/fusion/mptsas.c
        87        Hunk #1        mptsas_probe_hba_phys(MPT_ADAPTER *ioc)
98        File #5          a/drivers/net/fs_enet/fs_enet-mii.c
        100        Hunk #1        static struct fs_enet_mii_bus *create_bu
111        File #6          a/drivers/net/wireless/ipw2200.c
        113        Hunk #1        static struct ipw_fw_error *ipw_alloc_er
        126        Hunk #2        static ssize_t clear_error(struct device
        140        Hunk #3        static void ipw_irq_tasklet(struct ipw_p
        150        Hunk #4        static void ipw_pci_remove(struct pci_de
164        File #7          a/drivers/scsi/libata-scsi.c
        166        Hunk #1        int ata_cmd_ioctl(struct scsi_device *sc
178        File #8          a/drivers/video/au1100fb.c
        180        Hunk #1        void __exit au1100fb_cleanup(void)
```

このコマンドは、３つの異なる数値の類を表示します。

- （最初のカラムは）改変対象の個々のファイルをパッチ中で識別するための**ファイル番号**で、

- （字下げされた次の行には）変更されるファイルでの hunk の開始行番号と、
- （同じ行に）hunk を識別するための ***hunk* 番号**

必要なファイル番号や hunk 番号を特定するためには、視覚的な精査やパッチの読解が必要とされますが、それらの数値を `filterdiff` の`--files` や`--hunks` といったオプションに指定することで、ファイルや hunk を正確に選択することができます。

一度 hunk を取り出してしまえば、結合先パッチの末尾に結合して 12.14.2 節の残りの作業を再開することができます。

12.15　quilt と MQ の違い

既に quilt を熟知しているのであれば、MQ は同様のコマンド群を持っていますが、その働きにはいくらかの違いがあります。

殆どの quilt コマンドに対して、“q” で始まる対応する MQ のコマンドがあることに気付くことでしょう。但し、quilt の `add` および `remove` コマンドに対応するのが、Mercurial の通常の “`hg add`” および “`hg remove`” であるのが例外です。また、MQ には quilt の `edit` に対応するコマンドはありません。

第13章　Advanced uses of Mercurial Queues

Mercurial Queues の用法を真っ正直に話題にするのは簡単ですが、少々抑制を効かせて、込み入った開発環境での作業に役立つような、あまり利用されない機能を幾つか説明しようと思います。

この章では、Linux カーネル向けの Infiniband デバイスドライバ開発において、管理に用いていた技法を使用例として取り上げます。このデバイスドライバは（一般のデバイスドライバ程度には）大きく、35 のソースファイルにまたがった 25,000 行からなっており、少数の開発チームにより保守されています。

この章で扱っている対象は Linux に特化したものですが、自身が所有していないコードを元に多くの開発を行う必要がある局面で、同様の方針が適用できるでしょう。

13.1　The problem of many targets

Linux カーネルは頻繁に変更され、内部的には決して安定していません。開発者はリリースの間に度々思い切った変更を行います。このため、Linux カーネルの特定のリリース版で機能するドライバーの版は、概して他の版においては**コンパイル**すら通らない場合があります。

ドライバの保守を行うためには、いくつかの個別の Linux の版を意識する必要があります。

- 第一には、メインの Linux カーネル開発ツリーです。この場合のコードの保守は、カーネルコミュニティの他の開発者と共有され、彼らがカーネルのサブシステムに対して行うのと同程度に、“開発しながらの” 変更が行われます。
- 開発しているドライバを利用することができない古い Linux ディストリビューションを使用している顧客の要望に応えるために、古い Linux カーネルの版に対する幾つかの “バックポート” の保守も必要です。（コードの**バックポート**には、そのコードの開発対象となる版よりも古い版の環境で稼動させるための、コードの改変が必要です）
- 最後になりますが、顧客の利用しているカーネルやディストリビューションの、全体に対する更新を強いることなく新規機能を提供するために、ソフトウェアのリリーススケジュールは、Linux ディストリビューションやカーネル開発者が利用しているカーネルと、必ずしも足並みを揃えるわけではありません。

13.1.1　Tempting approaches that don’t work well

複数の異なる環境を対象としなければならない一連のソフトウェアの保守には、２つの “標準的な” 方法があります。

１つ目の方法は、それぞれが単一の環境を対象とする複数のブランチを管理する方法です。この方法の問題点は、リポジトリ間での変更の往来[1]において、鉄の規律でもって望む必要が有ることです。新しい機能やバグの修正は “真新しい” リポジトリで始めなければならず、その後で全てのバックポート用リポジトリに浸透させます。バックポートでの変更は、その伝播が更にブランチ限定されます。所属外のブランチに適用されるようなバックポート向けの変更は、おそらくドライバのコンパイルを妨げるでしょう。

２つ目の方法は、個々のコード片の有効／無効を、意図する対象に依存して切り替えるための条件文で埋められた、単一のソースツリーを保守する方法です。これらの “ifdef” 記述は、Linux カーネルツリーでは許されていないので、これらを取り除いて綺麗なツリーを生成するための、手動ないし自動の手順が必要です。この流儀で保守されるコードベースは早々に、理解も保守も困難な条件分岐の「鼠の巣」となるでしょう。

これらのいずれの手法も、正当なソースツリーのコピーを “所有” していない状況には適合しません。標準カーネルと共に配布される Linux ドライバの場合、Linus 氏のソースツリーは、世界中が正統とみなすコードのコピーから構

[1]訳注: いわゆる「マージ」のこと

成されます。上流リポジトリにおける “私の” ドライバは、Linus 氏のソースツリー上に改変内容が反映されるまでには、知らないうちに見知らぬ人々によって異なる版に改変されているかもしれません。

これらの手法は、上流リポジトリへのパッチの体裁を整えるのを難しくしてしまう、という欠点も持っています。

Mercurial Queues は、これまで述べてきた状況での開発を管理するための、良い候補と言えます。まさにこのような状況において、MQ は作業を快適にする更に幾つかの付加的機能を持っています。

13.2 ガードによる条件付きパッチ適用

おそらく、多くの対象環境に対する健全性を保守する方法は、所定の状況ごとに適用される特定のパッチを選択できること、と言えるでしょう。MQ は、上記の機能を持つ “ガード”（quilt の `guards` コマンドに由来します）と呼ばれる機能を提供します。まずはじめに、実験のための簡素なリポジトリを作成しましょう。

```
$ hg qinit
$ hg qnew hello.patch
$ echo hello > hello
$ hg add hello
$ hg qrefresh
$ hg qnew goodbye.patch
$ echo goodbye > goodbye
$ hg add goodbye
$ hg qrefresh
```

この手順により、異なるファイルを操作するので互いには依存性の無い２つのパッチを持つ、小さなリポジトリが得られます。

条件付き適用の考え方は、任意の単純な文字列からなる**ガード**された “札” (tag) をパッチに付与しておき、パッチ適用の際に、使用すべき特定のガードを MQ に対して教える、というものです。あらかじめ選択しておいたガードに応じて、MQ はガードされたパッチを適用するか見送るかを決定します。

個々のパッチは任意の数のガードを持つことができ、それぞれのガードは**ポジティブ**（“ガード選択時にパッチを適用する場合”）か**ネガティブ**（“ガード選択時にパッチ適用を見送る”）のどちらかです。ガードを持たないパッチは常に適用されます。

13.3 パッチのガードを制御する

“`hg qguard`” コマンドは、どのガードをパッチに適用するかを決定するか、さもなくば現時点で有効なガードを表示します。引数が無い場合、現在の最上位パッチのガードを表示します。

```
$ hg qguard
goodbye.patch: unguarded
```

パッチにポジティブなガードを設定するには、ガード名の接頭辞として “+” を付与します。

```
$ hg qguard +foo
$ hg qguard
goodbye.patch: +foo
```

パッチにネガティブなガードを設定するには、ガード名の接頭辞として “-” を付与します。

```
$ hg qguard hello.patch -quux
hg qguard: option -u not recognized
hg qguard [-l] [-n] -- [PATCH] [+GUARD]... [-GUARD]...

set or print guards for a patch

    Guards control whether a patch can be pushed. A patch with no
    guards is always pushed. A patch with a positive guard ("+foo") is
    pushed only if the qselect command has activated it. A patch with
    a negative guard ("-foo") is never pushed if the qselect command
    has activated it.

    With no arguments, print the currently active guards.
    With arguments, set guards for the named patch.
    NOTE: Specifying negative guards now requires '--'.

    To set guards on another patch:
      hg qguard -- other.patch +2.6.17 -stable

options:

 -l --list  list all patches and guards
 -n --none  drop all guards

use "hg -v help qguard" to show global options
$ hg qguard hello.patch
hello.patch: unguarded
```

備考: “hg qguard” コマンドは、パッチにガードを設定しますが、パッチのガード設定を**変更**したりはしません。つまり、パッチに “hg qguard +a +b” を適用した後に、同じパッチに “hg qguard +c” を適用した場合、このパッチに設定されているガードは +c **だけ**となります。

Mercurial は、解釈・手動編集が共に容易な形式で、ガード情報を series に格納します（言い換えるなら、“hg qguard” コマンドを利用する必要は無く、series ファイルを直接編集しても構いません）。

```
$ cat .hg/patches/series
hello.patch
goodbye.patch #+foo
```

13.4 使用するガードの選択

“hg qselect” コマンドは、有効にするガードを決定します。ガードが決定することで、次に “hg qpush” を実行した際に MQ が適用するパッチが決定されます。このコマンドはそれ以外の働きをしません。特に、既に適用済みのパッチに対しては、一切何も行いません。

引数が指定されない場合、“hg qselect” コマンドは、現時点で有効になっているガードを１行に１つづつ表示します。個々の引数は、適用されるガードの名前とみなされます。

```
$ hg qpop -a
patch queue now empty
$ hg qselect
no active guards
```

```
$ hg qselect foo
number of unguarded, unapplied patches has changed from 1 to 2
$ hg qselect
foo
```

現在選択されているガードの一覧が guards ファイルに格納されていますので、興味があれば見てみるのも良いでしょう。

```
$ cat .hg/patches/guards
foo
```

“hg qpush” を実行することで、ガード選択の効果を見ることができます。

```
$ hg qpush -a
applying hello.patch
applying goodbye.patch
now at: goodbye.patch
```

“+” ないし “-” で始まる名前はガード名にはできません。空白文字を含むものもガード名にはなれませんが、それいがいの大抵の文字は使用可能です。不正なガード名の使用は、MQ により警告されます。

```
$ hg qselect +foo
abort: guard '+foo' starts with invalid character: '+'
```

ガード選択の変更は、適用されるパッチを切り替えます。

```
$ hg qselect quux
number of guarded, applied patches has changed from 0 to 1
$ hg qpop -a
patch queue now empty
$ hg qpush -a
applying hello.patch
skipping goodbye.patch - guarded by ['+foo']
now at: hello.patch
```

ネガティブなガードがポジティブなガードに優先することを、以下の例で見ることができます。

```
$ hg qselect foo bar
number of unguarded, unapplied patches has changed from 0 to 1
$ hg qpop -a
patch queue now empty
$ hg qpush -a
applying hello.patch
applying goodbye.patch
now at: goodbye.patch
```

13.5 MQ のパッチ適用ルール

パッチ適用の有無を判定する際に、MQ は以下のルールを使用します。

- ガード無しパッチは常に適用されます。
- 現在選択されているガードにマッチするネガティブガードがある場合、パッチは適用されません。
- 現在選択されているガードにマッチするポジティブガードがある場合、パッチは適用されます。
- 現在選択されているガードにマッチするガードが何も無い場合、パッチは適用されません。

13.6 Trimming the work environment

先に述べた Linux カーネル向けの Infiniband デバイスドライバ開発でのパッチ適用では、Linux カーネルの通常のソースツリーは使用しません。その代わり、Infiniband デバイスドライバ開発に関連するソース／ヘッダのみを含むリポジトリを作成し、そこに対してパッチを適用するようにします。このリポジトリのサイズはカーネルリポジトリの 1% に収まるため、作業を行うのも簡単です。

縮小版のリポジトリを作成したならば、パッチの “適用対象” となるバージョンを選択します[2]。XXXXXXXXXXXX This is a snapshot of the Linux kernel tree as of a revision of my choosing. XXXXXXXXXXXX 適用対象を選択する際に筆者は、当該リビジョンのカーネルリポジトリにおけるチェンジセット ID を、コミットメッセージ[3]中に記録しておきます。カーネルツリー中の開発に関連する部位に関して、スナップショットによって “状況” と内容が特定できるため、縮小版リポジトリと通常版のカーネルツリーのいずれに対しても、パッチの適用が可能になります。XXXXXX Since the snapshot preserves the “shape” and content of the relevant parts of the kernel tree, I can apply my patches on top of either my tiny repository or a normal kernel tree.

通常は、パッチの適用対象となるソースツリーのベースには、上流リポジトリの直近のスナップショットを使用すべきです。そうすることで、作成したパッチを上流リポジトリの担当者へ送付する際に、殆ど（あるいは全く）改変の必要が無くなるでしょう。

13.7 Dividing up the `series` file

筆者は、`series` に列挙されるパッチを、幾つかの論理的なまとまりに分類しています。それぞれのパッチ分類は、その後に列挙されるパッチの意図を記述したコメントブロックで開始されます。

筆者の扱っているパッチ分類は、以下のような並びになっています。分類の順序は重要なので、分類を紹介した後で説明します。

- “受理済み (accepted)” 分類: 開発チームが Infiniband サブシステムの保守担当に送付して、既に受理はされているものの、縮小版リポジトリが元にしているスナップショットには、まだ反映されていないパッチの分類です。これらは、上流リポジトリの保守担当のリポジトリと同じ状態を得るために、ソースツリーを変換する “読み出し限定” パッチです。
- “再作業 (rework)” 分類: 上流リポジトリの保守担当に送付したものの、受理に当たって変更を要求されたパッチの分類。
- “保留 (pending)” 分類: 上流リポジトリの保守担当に送付こそしていないものの、既に作業を終えたパッチの分類。しばらくの間は “読み出し限定” として扱われます。上流リポジトリの保守担当により受理されれば、このパッチを “受理済み” 分類の末尾へと移動します。受理に当たって変更が要求された場合、“再作業” 分類の先頭へと移動します。
- “作業中 (in progress)” 分類: 目下のところ活発に作業が行われているパッチの分類。この分類のパッチは、外部に公開すべきではありません。

[2] 訳注: ここで言う「choose」（選択）は、“`hg update`” 実行を指すのではないか？そうであれば、次文が「これは〜スナップショットだ」というのも理解できる。

[3] 訳注: パッチの？ XXXXXX

- “バックポート (backport)” 分類: 古い版のカーネルのソースツリーに適合させるためのパッチの分類。
- “内部用 (do not ship)” 分類: 何らかの理由により、上流リポジトリの保守担当へは送付されないパッチの分類。このようなパッチの例としては、ドライバ識別用の埋め込み文字列の変更を行うことで、ソースツリーのものとは異なるドライバ実装の版[4]と、ディストリビューションベンダによって配布されるドライバ実装の版の間で、動作確認等における区別を容易にするパッチがあります。

ではここで、パッチ分類尾をこの順番にする理由に戻りましょう。コンテキストの変更が発生することで、スタック上方のパッチへの再作業[5]が必要になることが無いように、スタック中で底にあるパッチほど安定していて欲しいものです。変更されにくいパッチ群を `series` ファイルの冒頭に置くことで、この目的を達成することができます。

他のパッチの適用を極力上流リポジトリの状態に近いソースツリーへ行うために、ソースツリーの変換に必要と思われるパッチも重要です。受理済みのパッチも暫くの間保持しているのはそのためです。

“バックポート” および “内部用” パッチは、`series` 末尾近辺を転々とします。バックポートパッチは他の全てのパッチ適用の上で適用されなければなりませんし、その上、“内部用” パッチは不都合が無いように内部に留まり続ける必要があります。

13.8 Maintaining the patch series

筆者の作業の際には、パッチ適用を制御するために複数のガードを使用しています。

- “受理済み” パッチには、`accepted` ガードが付与されます。このガードは殆どの場合に有効とされます。既にパッチが適用されているソースツリーにパッチを適用する際には、パッチを適用させないようにすることが[6]できるので、後続のパッチ群は綺麗に適用されます。
- 作業は “完了” しているものの、上流リポジトリの保守担当に送付されていないパッチ[7]には、何もガードが付与されません。上流リポジトリのコピーに対してパッチスタックを適用する場合、特に何もガードを指定しなくても、適度に安全なソースツリーを得ることができます。
- 上流リポジトリの保守担当への（再）送付に当たって、再作業が必要なパッチには `rework` ガードが付与されます。
- 目下開発作業中にあるパッチ[8]には、`devel` ガードが付与されます。
- バックポートパッチには、適用対象カーネルのバージョンを指定する複数のガードが付与されます。例えば、2.6.9 版へのバックポートを行うパッチには、`2.6.9` ガードが付与されます。

これらのガード分類により、最終的にどのようなソースツリーが得られるかを決定する際に、少なからぬ柔軟性を得ることができます。多くの場合、適切なガードの選択は構築手順の中で自動化されていますが、特別な状況向けにガードの調整を手動で行うことも可能です。

13.8.1 The art of writing backport patches

MQ を使用することで、バックポートパッチの作成は単純な作業となります。旧版のカーネル配下においてもドライバが正常に稼動するように、旧版のカーネルにおいて提供されていない機能を使用するコードの変更が、バックポートパッチのすべきことの全てです。

良いバックポートパッチを書く際のゴールは、対象とする旧版カーネル向けに書いたかのように、あなたのコードを変更するようなパッチにすることです。パッチがでしゃばらない程、理解と保守が容易になります。コード中の大量の `#ifdef`（条件に応じて適用されるコード片）による “鼠の巣” 化を避けるためにバックポートパッチ群を書くの

[4]訳注: 開発中のドライバのこと？
[5]訳注: “`hg qrefresh`” の実行によるパッチの修正のこと
[6]訳注: `accepted` ガード付きパッチを無効にすることで
[7]訳注: 先の分類で言うところの “保留 (pending)”
[8]訳注: 先の分類で言うところの “作業中 (in progress)”

であれば、バージョン依存な #ifdef をパッチに持ち込むべきではありません。バージョン依存な #ifdef を使用する替わりに、個々のパッチはバージョンに依存しない変更を行うようにして、パッチの適用をガードによって制御すべきです。

“通常” のパッチと、その適用結果を更に変更するバックポートパッチとを、別個のグループに分離するのには２つの理由があります。第１の理由は、これらのパッチが混ざり合った場合に、上流リポジトリの保守担当へのパッチ送付の自動化の際に、patchbomb 拡張のようなツールを使うことが難しくなるためです。第２の理由は、後続の通常パッチの適用コンテキスト[9]をバックポートパッチが混乱させてしまい、通常パッチの適用前に適用されたバックポートパッチ**抜き**では、通常パッチを綺麗に適用することができなくなってしまうためです。

13.9 Useful tips for developing with MQ

13.9.1 Organising patches in directories

MQ を利用した実在するプロジェクトで作業をしているのであれば、多くのパッチを蓄積することも難しいことではありません。例えば、筆者は 250 を超えるパッチを抱えたパッチリポジトリを持っています。

パッチを個別の論理的なまとまりに分類できるのであれば、MQ はパッチ名にパス区切りが含まれていても問題ないので、それぞれのパッチを異なるディレクトリに格納することもできます[10]。

13.9.2 Viewing the history of a patch

長期間にわたってパッチの開発を行う場合、12.11 節で述べたように、パッチをリポジトリで管理するのが良いでしょう。その場合は早々に、パッチの変更履歴の参照に “hg diff” が使えないことに気付くことでしょう。これは実際のコードの二次派生物 (差分の差分) を見ていること以外にも、タイムスタンプやパッチ更新時のディレクトリ名等を改変することで MQ が雑音を加えてしまっていることに原因があります。

Mercurial に同梱されている extdiff 拡張を使うことで、２つの版のパッチ差分を幾分読みやすいものにすることができます。この拡張を使うためには、サードパーティーパッケージである patchutils [Wau] が必要です。このパッケージが提供する interdiff というコマンドは、差分間の差分を１つの差分として表示します。同じ差分の２つの版[11]に対してこのコマンドを適用すると、最初の版から次の版へと変更するための差分を生成します。

いつものように、hgrc ファイルの [extensions] セクションに行を追加することで、extdiff 拡張を有効化することができます。

```
[extensions]
extdiff =
```

interdiff コマンドは２つのファイル名の指定が必要ですが、extdiff 拡張は、それぞれ任意の数のファイルを配下に持つ、２つのディレクトリに対して動作するプログラムの指定が必要です。そのため、これら２つのディレクトリ配下の個々のファイル対に対して interdiff を実行する小さなプログラムが必要です。本書のソースコードリポジトリにおける examples ディレクトリ配下に、hg-interdiff として格納されています。

```
#!/usr/bin/env python
#
# Adapter for using interdiff with mercurial's extdiff extension.
#
# Copyright 2006 Bryan O'Sullivan <bos@serpentine.com>
#
# This software may be used and distributed according to the terms of
# the GNU General Public License, incorporated herein by reference.

import os, sys

```

[9]訳注: patch ファイルにおける「コンテキスト」

[10]訳注: MQ はパッチ内容の保存先として、パッチ名と同名のファイルを作成するため、パッチ名にパス区切りが含まれる場合、MQ は自動的にサブディレクトリを作成します

[11]訳注: 「同じパッチの異なる版」の意か？

```
def walk(base):
    # yield all non-directories below the base path.
    for root, dirs, files in os.walk(base):
        for f in files:
            path = os.path.join(root, f)
            yield path[len(base)+1:], path

# create list of unique file names under both directories.
files = dict(walk(sys.argv[1]))
files.update(walk(sys.argv[2]))
files = files.keys()
files.sort()

def name(base, f):
    # interdiff requires two files; use /dev/null if one is missing.
    path = os.path.join(base, f)
    if os.path.exists(path):
        return path
    return '/dev/null'

ret = 0

for f in files:
    if os.system('interdiff "%s" "%s"' % (name(sys.argv[1], f),
                                          name(sys.argv[2], f))):
        ret = 1

sys.exit(ret)
```

hg-interdiff がシェルのコマンド検索パス上に有る場合、MQ のパッチディレクトリから以下のようにして起動することができます。

```
hg extdiff -p hg-interdiff -r A:B my-change.patch
```

おそらくこの長たらしいコマンドを何度も使うことになるでしょうから、再度 hgrc を編集して、hgext を Mercurial の普通のコマンド並に使えるようにしましょう。

```
[extdiff]
cmd.interdiff = hg-interdiff
```

この記述により interdiff が hgext から利用できるようになりますので、先の “hg extdiff” 起動も短くなって幾分使いやすくなるでしょう。

```
hg interdiff -r A:B my-change.patch
```

備考: interdiff コマンドは、場合だけ正しく機能します。The interdiff command works well only if the underlying files against which versions of a patch are generated remain the same. パッチの生成・ファイルの変更およびパッチの更新を行った場合、interdiff は有用な出力を生成しないことがあります。

extdiff 拡張は、MQ パッチの表示機能の向上に留まらない有用なものです。extdiff 拡張に関する詳細は、14.2 節を参照してください。

第14章　Adding functionality with extensions

Mercurial は機能性の見地から見た場合には申し分無い一方で、変り種の機能群は故意に除外されています。簡潔さを保つ遣り方は、保守担当と利用者の両方に対してソフトウェアの扱いやすさを維持します。

しかし Mercurial は、利用者を杓子定規なコマンド群の只中に利用者を閉じ込めるようなことはしません。**イクステンション**（この種のものは**プラグイン**と呼ばれることもあります）として機能を追加することができるのです。幾つかのイクステンションについては、既に前の章で話題にしています。

- 3.3 節では `fetch` イクステンションを取り上げています。このイクステンションは、新たな変更の取得と手元の変更へのマージを、単一のコマンド “`hg fetch`” で実施します。
- `bisect` は、バグの原因となる変更を効率的に検索するイクステンションで、9.5 節で取り上げました。
- 10 章では、フックに関連した有用な機能を持つイクステンションを取り上げました。`acl` はアクセスコントロールリストの機能を、`bugzilla` は Bugzilla バグ追跡システムとの統合を、`notify` は変更追加時における電子メール通知の機能を、Mercurial に追加します。
- Mercurial Queues パッチ管理イクステンションは、２つの章と１つの appendix を丸々費やすに値する価値を持っています。12 章は基本を、13 章はより進んだ話題を、そして appendix **??** は各コマンドの詳細を取り上げています。

本章では、上記以外の Mercurial で利用可能な幾つかのイクステンションについて取り上げ、その上で、自分でイクステンションを実装する際に必要と思われる仕組みについて、簡単に触れようと思います。

- 14.1 節では、`inotify` イクステンションによる**絶大な**性能改善の可能性について取り上げます。

14.1　Improve performance with the `inotify` extension

一般的な Mercurial の操作が 100 倍速くなることに興味があるのでしたら、ぜひこの節を読んでください。

Mercurial は通常の環境であっても高い性能でを発揮します。XXXX 否定の接続の筈 XXXX 例えば “`hg status`” コマンドの実行の際には、ファイルの状態を表示するために、リポジトリ配下の殆ど全てのディレクトリとファイルに対する走査が必要です。他の多くの Mercurial コマンドも、舞台裏では同様の作業を必要としています。例えば “`hg diff`” コマンドは、状態比較機構[1]を用いることで、明らかに変更されていないファイルに対して、実行コストの高い比較処理が実施されることを回避しています。

ファイル状態の取得は性能確保上重要なことなので、Mercurial の開発者達は、ギリギリのところまでこの部分の実装を最適化してきました。しかし “`hg status`” 実行の際には、前回の確認以降の変更の有無を知るために、コストの高いシステムコールを、Mercurial の管理下にあるファイル毎に最低１回発行する必要がある、という事実は回避しようがありません。一定以上の大きさのリポジトリでは、この処理には長い時間がかかります。

影響の大きさを数値化すべく、150,000 のファイルを管理するリポジトリで実験を行った結果、いずれのファイルも変更**されていない**場合であっても、“`hg status`” の実行には１０秒を要します。

多くの近代的 OS は、ファイル更新の通知機構を備えています。適切なサービスにプログラムを登録しておくことで、対象となるファイルに関する生成・変更・削除といったイベントが発生する都度、OS が通知をしてくれます。Linux 環境では、`inotify` と呼ばれるカーネルコンポネントが通知機構を提供します。

Mercurial の `inotify` イクステンションは、カーネルの `inotify` と連携することで、“`hg status`” コマンドを最適化します。`inotify` イクステンションは２つの要素から構成されています。デーモン部分がバックグラウンドで稼動

[1]訳注: 4.5.5 節参照

することで、inotify カーネルコンポネントから通知を受け取ります。デーモン部分は、通常の Mercurial コマンドからの接続要求も受け付けます。inotify イクステンションは、ファイルシステムの走査の代替としてデーモンを必要とするため、Mercurial の挙動そのものを改変します。デーモンはリポジトリ状態に関する完全な情報を保持しているので、リポジトリ配下のディレクトリやファイルを走査すること無しに、即座に応答を返すことができます。

先に述べたとおり、通常の Mercurial では、150,000 のファイルを管理するリポジトリでの “hg status” 実行に１０秒を要しました。inotify イクステンションを有効にすることで、実行に要する時間は *1000* **倍**早い 0.1 秒まで低減できました。

話を先に進める前に、以下の点に注意してください。

- inotify は Linux 環境固有のイクステンションです。Linux の inotify サブシステムと直接連携するため、他の OS 環境下では機能しません。
- 2005 年初旬以後にリリースされた Linux ディストリビューションでの利用をお薦めします。それ以前のディストリビューションは、inotify が組み込まれていないか、必要な API を glibc が提供していないものと思われます[2]。
- 全てのファイルシステムが inotify イクステンションの利用に適しているとは限りません。典型的な例としては、同一のネットワークファイルシステムを、Mercurial を稼動させる複数のシステムでマウントしているような場合です。カーネルの inotify サブシステムは、リモートホストでの変更を知る術を持ちません。殆どのローカルファイルシステム（例えば ext3、XFS や ReiserFS）は、上手く機能する筈です。

inotify イクステンションは、2007 年 5 月の時点では Mercurial に同梱されていません[3]ので、他のイクステンションと比較して多少の準備作業が必要ですが、性能向上にはそれだけの価値があります。

inotify イクステンションは目下、Mercurial ソースコードへのパッチと、inotify サブシステム連携の Python バインディングライブラリの２つの要素から構成されています。

> **備考:** inotify の Python バインディングライブラリには**２種類**あります。１つは pyinotify と呼ばれるもので、幾つかの Linux ディストリビューションには python-inotify という名前で同梱されています。実用に供するには非常にバグが多く効率も悪いので、このライブラリは使うべきでは**ありません**。

事を進めるに当たっては、既に機能しているインストール済み Mercurial を複製するのが良いでしょう。To get going, it’s best to already have a functioning copy of Mercurial installed. XXXXXX

> **備考:** 以下の手順を踏む場合、最も最新の “最先端な”Mercurial 実装で、既にインストール済みの Mercurial を**置き換える**ことになります。これは警告です。

1. inotify の Python バインディングのリポジトリを複製します。ビルドおよびインストールを行ってください。

```
hg clone http://hg.kublai.com/python/inotify
cd inotify
python setup.py build --force
sudo python setup.py install --skip-build
```

2. Mercurial の crew リポジトリを複製します。Mercurial Queues により crew リポジトリのローカルコピー[4]にパッチを当てる為に、inotify パッチのリポジトリも複製してください。

[2] 訳注: man ページによれば、inotify の利用に当たっては、2.6.13 版以後のカーネルと 2.4 版以後の glibc が必要だそうです。
[3] 訳注: 2007 年 10 月の 0.9.5 版段階でも同梱されていません
[4] 訳注：ここでは crew から更に inotify を複製していますが、inotify イクステンション利用のためだけにビルドする場合、直接 crew で作業しても問題無い筈です。

```
hg clone http://hg.intevation.org/mercurial/crew
hg clone crew inotify
hg clone http://hg.kublai.com/mercurial/patches/inotify inotify/.hg/patches
```

3. Mercurial Queues イクステンション（mq）が利用可能であることを確認してください。MQ を利用したことが無い場合、まずは 12.5 節を読んでください。

4. inotify （ローカル）リポジトリに移動して、“hg qpush” コマンドの -a オプションを使用して、全ての inotify パッチを適用してください。

```
cd inotify
hg qpush -a
```

   “hg qpush” がエラーメッセージを表示した場合は、作業を継続せずに開発コミュニティに助けを求めてください。

5. パッチ適用版の Mercurial をビルドおよびインストールします。

```
python setup.py build --force
sudo python setup.py install --skip-build
```

適切にパッチが適用された版の Mercurial が一旦できてしまえば、inotify イクステンションを有効にするために必要なことは、hgrc ファイルに以下の記述を追加することだけです。

```
[extensions]
inotify =
```

inotify イクステンションが有効化されると、リポジトリの状態を必要とするコマンドの初回起動の時点で、Mercurial は自動的且つ透過的に状態管理用デーモンを起動します。状態管理デーモンは、リポジトリごとに起動されます。

状態管理デーモンはひそやかに起動され、バックグラウンドで実行し続けます。inotify イクステンションを有効にした複数のリポジトリで、幾つかのコマンドを実行した後に、実行中のプロセス一覧を見れば、カーネルからの通知と Mercurial からの問い合わせの両方を待っている複数の hg プロセスを見ることができる筈です。

inotify イクステンションを有効にした際でも、リポジトリにおける Mercurial コマンドの初回起動は、通常の Mercurial コマンド実行と同程度の性能で実行されます。これは状態管理デーモンによる通常の状態走査が必要なためで、後にカーネルからの更新通知を受け取る際の基底状態となります。しかし、これ以降の状態確認の必要な**全ての**コマンド実行は、どんなに小さなサイズのリポジトリであっても、目に見えて速くなっている筈です。リポジトリが大きければ大きいほど、目に見えて性能が大きく改善されることでしょう。inotify デーモンは、どんなサイズのリポジトリであっても、状態取得操作を殆ど瞬時に終了させることができます。

“hg inserve” コマンドにより、状態管理デーモンを手動で起動することもできます。手動での起動により、デーモンの実行に関して幾分明瞭な制御を手にすることができます。このコマンドの起動は、当然 inotify イクステンションが有効になっている場合に限り使用可能です。

inotify イクステンションを使用している際には、状態関連コマンドの実行全般がそれ以前と比較して速くなっている点を除けば、Mercurial の挙動は**全く変わらない**筈です。

とりわけ、コマンドの出力は異ならず、同じ結果を返す筈です。inotify イクステンションの有無で異なる結果が変える場合、障害として報告をしてください。

14.2 Flexible diff support with the `extdiff` extension

Mercurial の組み込み “hg diff” コマンドは、unified 差分をそのまま出力します。

```
$ hg diff
diff -r 77cc4812a4a6 myfile
--- a/myfile        Mon Jul 20 21:58:44 2009 +0000
+++ b/myfile        Mon Jul 20 21:58:44 2009 +0000
@@ -1,1 +1,2 @@
 The first line.
+The second line.
```

変更内容の表示に外部ツールを使いたい場合は、extdiff イクステンションが良いでしょう。extdiff イクステンションにより、変更内容表示に例えばグラフィカルな外部差分ツールが利用できるようになります。

extdiff イクステンションは Mercurial に同梱されているので簡単に利用できます。hgrc ファイルの [extensions] セクションに、イクステンションを有効にする記述を 1 行追加するだけで良いのです。

```
[extensions]
extdiff =
```

この設定により、“hg extdiff” コマンドが利用可能になりますが、基底状態ではこのコマンドは、組み込みの “hg diff” コマンドと同じ形式の unified 差分を、システムの diff コマンドにより生成します。

```
$ hg extdiff
--- a.77cc4812a4a6/myfile        2009-07-20 21:58:44.000000000 +0000
+++ /tmp/extdiff3rhJme/a/myfile        2009-07-20 21:58:44.000000000 +0000
@@ -1 +1,2 @@
 The first line.
+The second line.
```

組み込みの “hg diff” コマンドの結果出力と厳密には一致しません[5]が、同じオプションを指定してもシステム[6]ごとに（システムの）diff コマンドの出力が異なるからです。

上記の出力結果に “making snapshot” 行が含まれていることからも察することができますが、“hg extdiff” コマンドはソースツリーに関するスナップショットを 2 つ作成します。1 つ目のスナップショットはソースのリビジョンのもので、2 つ目は作業領域ディレクトリにおける対象リビジョンのものです[7]。“hg extdiff” コマンドはこれらのスナップショットを一時ディレクトリに作成し、これらのディレクトリ名を引数にして外部の差分表示ツールを起動し、その後一時ディレクトリを削除します。実行効率上、2 つのリビジョンの間で差分のあるディレクトリ・ファイルのスナップショットだけが作成されます。

スナップショットディレクトリの名前は、元となるリポジトリのベース名と同じ名前を持ちます。/quux/bar/foo というリポジトリの場合、個々のスナップショットのディレクトリ（ベース）名は foo となります。対応するチェンジセット ID がある場合、スナップショットのディレクトリ名にはチェンジセット ID が付与されます。a631aca1083f 版に対するスナップショットのディレクトリ名は foo.a631aca1083f となります。作業領域ディレクトリの現行状態に対するスナップショットは、チェンジセット ID が付与されませんので、この例では単に foo という名前になります。実際の挙動を見るために、再度前出の “hg extdiff” の実行例を見てみましょう。差分出力のヘッダ部に、スナップショットディレクトリの名前が埋め込まれているのに気付くことでしょう。

“hg extdiff” コマンドには、2 つの重要なオプションがあります。 -p オプションは、システムの diff コマンドの代替として使用される差分表示プログラムを指定します。 -o オプションは、“hg extdiff” が外部の差分表示プログラム起動時に指定するオプション（デフォルトでは “-Npru” が指定され、diff を使用する場合にのみ意味を持ちます）を指定します。それ以外の点では、“hg extdiff” コマンドは組み込みの “hg diff” コマンドと同様に振舞いますので、オプション名やオプション指定の文法、比較対象リビジョンを指定する引数、比較したいファイル名の指定などは、組み込みの “hg diff” と同じように指定できます。

[5] 訳注: どの部分を指して「一致しない」と言っているのか？
[6] 訳注: ここで言う「system」とは？ XXXXX
[7] 訳注: 作業領域ディレクトリの「親リビジョン」と「現行状態」

実行例として、（通常の “hg diff” による）unified 差分の代わりに、システム標準の diff コマンドによる context 差分（-c オプション使用）を、デフォルトの３行ではなく５行の context 行（-C オプションでの 5 指定）で表示する方法を示します。

```
$ hg extdiff -o -NprcC5
*** a.77cc4812a4a6/myfile        Mon Jul 20 21:58:45 2009
--- /tmp/extdiff3rhJme/a/myfile        Mon Jul 20 21:58:44 2009
***************
*** 1 ****
--- 1,2 ----
  The first line.
+ The second line.
```

グラフィカルな差分ツールの起動は非常に簡単です。kdiff3 起動の例を示します。

```
hg extdiff -p kdiff3 -o ''
```

利用する差分表示コマンドがディレクトリ指定を扱えない場合でも、簡単なスクリプトを使うことでその問題を解決できます。そのようなスクリプトによる mq イクステンションと interdiff コマンドの連携例は、13.9.2 節を参照してください。

14.2.1 Defining command aliases

“hg extdiff” コマンドや利用する差分表示コマンドの、両方のオプションを覚えておくのは面倒ですので、extdiff イクステンションは、使用する差分表示コマンドを正しいオプションで起動する**新しい**コマンドを定義できるようになっています。

新しいコマンド定義のために必要なのは、hgrc ファイルを編集し、[extdiff] という名前のセクションを追加するだけです。このセクションでは、複数のコマンドを定義することができます。以下に kdiff3 コマンドを追加する例を示します。一旦定義してしまえば、“hg kdiff3” と入力するだけで extdiff イクステンションが kdiff3 を起動します。

```
[extdiff]
cmd.kdiff3 =
```

定義の右辺を上記例のように空にした場合、extdiff イクステンションは、定義したコマンドの名前を実行すべき外部プログラムの名前と見なします。しかし、これらの名前が一致している必要はありません。以下の例では、kdiff3 を実行するコマンドを “hg wibble” という名前で定義しています。

```
[extdiff]
cmd.wibble = kdiff3
```

差分表示プログラム起動の際のデフォルトオプションも指定することができます。“opts.” 接頭辞に続いて、オプションを適用したいコマンド名を記述してください。以下の例では、vim エディタの DirDiff 拡張を実行する “hg vimdiff” コマンドを定義しています。

```
[extdiff]
cmd.vimdiff = vim
opts.vimdiff = -f '+next' '+execute "DirDiff" argv(0) argv(1)'
```

14.3 Cherrypicking changes with the `transplant` extension

※ Brendan とチャットでの話し合いが必要

14.4 Send changes via email with the `patchbomb` extension

多くのプロジェクトでは、共有リポジトリに最終成果をコミットする前に、変更内容をメーリングリストに投稿して査読や論評を行う“変更レビュー”の文化を持っています。リポジトリへのアクセス権を持たない人々からの変更依頼を適用する、門番の役割を果たす人々がいるプロジェクトもあります。

Mercurial の `patchbomb` イクステンションを利用することで、レビューや提案のための電子メールによる変更送信が容易になります。このイクステンションの名前は、変更がパッチ形式で整形され、１チェンジセット毎に１つの電子メールで送信されることに由来しています。電子メールによる一連の変更の送信が、受信者のメールボックスにとって“爆撃”（bombing）のようであることから、“patchbomb”と呼ばれています。

`patchbomb` イクステンションの基本的な設定記述は、いつものように `hgrc` への１行か２行程度の記述だけです。

```
[extensions]
patchbomb =
```

一旦イクステンションを有効にしたならば、“`hg email`”という新たなコマンドが利用可能になります。

“`hg email`”コマンドの安全且つ最善の実行手順は、**必ず** `-n` オプションを付けて一旦実行してみることです。 `-n` オプション付きの実行は、実際の電子メール送信は行わずに、送信**されるであろう**内容を表示します。変更内容にざっと目を通して、送信内容が適切であることを確認したならば、 `-n` オプション抜きで再度“`hg email`”コマンドを実行してください。

“`hg email`”コマンドは、他の Mercurial コマンドと同様のリビジョン指定が可能です。例えば以下の実行例では、リビジョン７から `tip` までの全てのリビジョン（リビジョン７および `tip` も含みます）が送信されます。

```
hg email -n 7:tip
```

比較対象の**リポジトリ**を指定することもできます。リビジョン指定無しでリポジトリを指定した場合、“`hg email`”コマンドは、遠隔リポジトリに存在しないローカルリポジトリの全てのリビジョンを送信します。リビジョンないし（ `-b` オプションによる）ブランチ名を追加指定することで、送信されるリビジョンを制限することができます。

送信先アドレスを指定しない“`hg email`”実行は完璧に安全で、その場合には“`hg email`”は対話的に入力を求めてきます（Linux や Unix ライクなシステムを利用している場合、これらのヘッダ値入力の際には、`readline` 様式の編集機能が利用可能です）。

単一のリビジョンだけを送信する場合、“`hg email`”コマンドの基底動作では、コミットメッセージの最初の１行を送信する電子メールのサブジェクトに利用します。

複数のリビジョンを送信する場合、“`hg email`”コマンドはチェンジセット毎に電子メールを送信します。この場合、送信しようとする一連の変更の目的を記述した前置きの電子メールを、一連のメール送信の先触れとして送信します。

14.4.1 Changing the behaviour of patchbombs

電子メールによる変更内容送信の形式が、全てのプロジェクトで厳密に同じわけでは無いことから、`patchbomb` イクステンションは、コマンド行でのオプション指定による幾つかの適合処理を実施します。

- コマンド行での `-s` オプションにより、前置きメッセージのサブジェクトを指定できます。このオプションには、サブジェクトとして使用するテキストを指定します。
- `-f` オプションにより、電子メールの送信元アドレスを変更できます。このオプションには、送信元アドレスとして使用する電子メールアドレスを指定します。
- 基底動作では、電子メールごとに unified 差分（形式の詳細に関しては 12.4 節を参照してください）を送信します。 `-b` オプションを指定することで、バイナリバンドル形式での送信を選択できます。
- unified 差分の通常の出力[8]はメタデータヘッダから始まります。 `--plain` オプションを指定することで、これらを省略した簡素な形式の差分を送信することができます。

[8]訳注: 「Mercurial における通常の出力」の意味？それとも「patchbomb における通常の出力」の意味？

- 差分部分は通常、パッチの説明部分と同じ MIME パートに“並べて”送信されます。メールの最初の MIME パートからしか引用できないメールツールもあるため、最も多くの読み手にとって、一番容易に差分を引用して返信できるのがこの形式です。説明部分と差分部分を別々の MIME パートとして送信したい場合は、 `-a` オプションを指定してください。

- `-m` オプションを指定することで、電子メールでの送信の替わりに、`mbox` 形式のメールフォルダへの書き込みを行うことができます。このオプションには、書き込み先ファイル名を指定します。

- 各パッチおよび前置きメッセージに対して、`diffstat` 形式の要約を付与したい場合は、 `-d` オプションを指定してください。`diffstat` コマンドは、パッチ適用先ファイル名と、影響を受ける行数、および各ファイル毎の変更量を表すヒストグラムを一覧表示します。メールの読み手は、この情報からパッチの複雑度に関する質的な一覧性を得ることができます。

付録A　Installing Mercurial from source

A.1　On a Unix-like system

（2.3 ないしそれ以後の）新しい版の Python が利用可能な Unix 的なシステムを利用している場合は、Mercurial をソースファイルからインストールするのは簡単です。

1. 最新版の tar アーカイブ（tarball）を http://www.selenic.com/mercurial/download からダウンロード。
2. tar アーカイブを展開:

```
gzip -dc mercurial-version.tar.gz | tar xf -
```

3. ソースディレクトリに移動して、インストール用スクリプトを実行。以下の手順は、ビルドした Mercurial をホームディレクトリ配下にインストールします。

```
cd mercurial-version
python setup.py install --force --home=$HOME
```

インストールが完了したなら、ホームディレクトリ直下の bin ディレクトリに Mercurial がインストールされます。シェルのコマンド検索パスへの bin ディレクトリの追加を忘れないようにしてください。

Mercurial の実行に必要な Mercurial パッケージを探し出せるように、PYTHONPATH 環境変数の設定も必要となるでしょう。例えば著者のラップトップでは、PYTHONPATH 環境変数に/home/bos/lib/python を設定しています。実際に PYTHONPATH 環境変数に設定する値は、各自の環境で Python がどのように設定されているかに依存しますが、設定すべき値を得るのは簡単です。設定値に確信が持てない場合、上記のインストール用スクリプトの出力を見て、mercurial ディレクトリの内容がインストールされる先を確認してください。

A.2　On Windows

Windows 上で Mercurial をソースからビルドするには、様々なツール、相当な技術的知識に加えて、少なからぬ忍耐が要求されます。“気軽に使ってみたい” 場合には、ソースからのビルドは**全くお薦めできません**。Mercurial そのものをハックするので無い限り、バイナリ版の利用をお薦めします[1]。

Windows 上で Mercurial をソースからビルドする場合、多くの厄介事が起きることを覚悟した上で、Mercurial の Wiki 上にある http://www.selenic.com/mercurial/wiki/index.cgi/WindowsInstall に示されている “苦難の道” を辿ってください。

[1]訳注: どうしても最新の Mercurial ソースを利用したい場合、Windows ネイティブな振る舞いは期待できませんが、Cygwin 上で Mercurial をビルドするという手もあります。

付 録B　Open Publication License

Version 1.0, 8 June 1999

B.1　Requirements on both unmodified and modified versions

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

> Copyright (c) *year* by *author's name or designee*. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v*x.y* or later (the latest version is presently available at `http://www.opencontent.org/openpub/`).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section B.6).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

B.2　Copyright

The copyright to each Open Publication is owned by its author(s) or designee.

B.3　Scope of license

The following license terms apply to all Open Publication works, unless otherwise explicitly stated in the document.

Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

Severability. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

No warranty. Open Publication works are licensed and provided "as is" without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

B.4　Requirements on modified works

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.

2. The person making the modifications must be identified and the modifications dated.
3. Acknowledgement of the original author and publisher if applicable must be retained according to normal academic citation practices.
4. The location of the original unmodified document must be identified.
5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

B.5 Good-practice recommendations

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.
2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.
3. Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

B.6 License options

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

1. To prohibit distribution of substantively modified versions without the explicit permission of the author(s). "Substantive modification" is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

   To accomplish this, add the phrase "Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder." to the license reference or copy.
2. To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

   To accomplish this, add the phrase "Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder." to the license reference or copy.

関連図書

[AG] Jean Delvare Andreas Gruenbacher, Martin Quinson. Patchwork quilt. http://savannah.nongnu.org/projects/quilt.

[BI] Ronald Oussoren Bob Ippolito. Universal macpython. http://bob.pythonmac.org/archives/2006/04/10/python-and-universal-binaries-on-mac-os-x/.

[Bro] Neil Brown. `wiggle`–apply conflicting patches. http://cgi.cse.unsw.edu.au/~neilb/source/wiggle/.

[Dic] Thomas Dickey. `diffstat`–make a histogram of `diff` output. http://dickey.his.com/diffstat/diffstat.html.

[Dus] Andy Dustman. Mysql for python. http://sourceforge.net/projects/mysql-python.

[Gru05] Andreas Gruenbacher. How to survive with many patches (introduction to `quilt`). http://www.suse.de/~agruen/quilt.pdf, June 2005.

[Mas] Chris Mason. `mpatch`–help solve patch rejects. http://oss.oracle.com/~mason/mpatch/.

[O'S06] Bryan O'Sullivan. Achieving high performance in mercurial. In *EuroPython Conference*, July 2006. XXX.

[Pyt] Python.org. `ConfigParser`—configuration file parser. http://docs.python.org/lib/module-ConfigParser.html.

[RS] GNU Project volunteers Richard Stallman. Gnu coding standards—change logs. http://www.gnu.org/prep/standards/html_node/Change-Logs.html.

[Tat] Simon Tatham. Putty—open source ssh client for windows. http://www.chiark.greenend.org.uk/~sgtatham/putty/.

[Wau] Tim Waugh. `patchutils`–programs that operate on patch files. http://cyberelk.net/tim/patchutils/.

索引