Mercurialによる分散リビジョン管理

Bryan O'Sullivan

この書籍は Mercurial rev unknown によって管理される rev 8a3041e6f3cb, dated 2009-07-11 19:25 +0900, から製版された.

Contents

List of Figures

はじめに

分散リビジョンコントロールは比較的新しい領域であり，これまでの間違ったやりかたから抜け出そうという人々の思いで急速に発展している．

筆者が分散リビジョンコントロールの本を書いた理由は，これがフィールドガイドを書くのに相応しい重要なテーマだと考えたからである．対象として Mercurial を選んだ理由は，全体の学習が最も容易なツールであり，他のリビジョンコントロールツールでは得がたいスケーラビリティを有しているからだ．

0.1 この本は現在執筆中である

この本はまだ執筆中であるが，これが読者にとって有用であると信じて公開することにした．また，読者からも彼らの求めに合わせて貢献があることを期待している．

0.2 この本の例について

この本はコードサンプルに普通と違うアプローチを取っている．全ての例は“生きている”—全ての例は Mercurial コマンドを起動するシェルスクリプトの結果である．この本では，画像がソースから生成される度に，スクリプトが自動的に起動する．現在の結果は期待される結果と比較される．

このアプローチの利点は，例が常に正確であることで，記述は本の表紙で言及している Mercurial のバージョンでの挙動と**厳密に**一致する．筆者が Mercurial のバージョンを更新し，コマンドの出力が変わるとビルドは失敗する．

このアプローチの小さな欠点は，例の中に現れる日付と時刻が，人間のタイプでは有り得ないほど“固まって”しまっていることだ．人間であれば，1 つのコマンドを実行するのに数秒はかかり，タイムスタンプは広い範囲に拡がるはずのところが，この本で例を作成するスクリプトでは，1 秒の間に多くのコマンドを実行してしまう．

このため，例の中に現れる連続したコミットは，同じ秒に起きたようになっている．例えば， 9.5の節の `bisect` の例でこれが見られる．

従って，例を読む時には，日付と時刻にあまり拘らないようにしていただきたい．しかし，そこで見られるふるまいは正確であり，再現可能であることは間違いのないものである．

0.3 背表紙—この本は無料である

この本は Open Publication License の下でライセンスされ，全体に渡ってフリーソフトウェアツールを使って作られている．組版は LaTeX を使って行われ，図版は Inkscape を使って行われている．

この本の完全なソースは，`http://hg.serpentine.com/mercurial/book` にて Mercurial リポジトリとして公開されている．

Chapter 1

導入

1.1 リビジョンコントロール

リビジョンコントロールとは，複数のバージョンの情報を管理するプロセスである．最も単純な方法は，ファイルを変更したら，それまでのバージョンよりも大きな数字を含む新たな名前でセーブを行うなどの方法で全て手で行うことである．

たった一つのファイルに対しても，複数のバージョンを手で管理することは間違いを起こし易い作業で，このプロセスを自動化するソフトウェアツールがかなりの昔から提供されてきた．最初の自動化されたリビジョンコントロールツールは，一人のユーザを対象として，1 つのファイルのリビジョンを管理するために作られた．数十年が経ち，リビジョンコントロールの取り扱うスコープは大いに拡大した．今では複数人による複数ファイルの編集をも管理することができる．現代の最高のリビジョン管理ツールは数千人による，数十万個のファイルを擁するプロジェクトにも対応する．

1.1.1 なぜリビジョンコントロールを使うのか?

プロジェクトのために自動化されたリビジョンコントロールツールを使おうと考える理由は数多くある．

- リビジョン管理ツールは，プロジェクトの履歴と進化を記録するため，自分自身で記録する必要がない．全ての変更に対して**誰が何のためにいつ何を**変更したのかが記録される．
- リビジョンコントロールソフトウェアは他の人との共同作業を助ける．例えば，複数のユーザが同時に互換性のない変更を行った場合，ソフトウェアの支援でコンフリクトを特定し解決することができる．
- リビジョン管理ツールは犯したミスからの回復を助ける．加えた変更が後で間違いであったと分かった時，1 つまたは複数のファイルへの変更を破棄することができる．実際のところ，**真に**優れたリビジョンコントロールツールは紛れ込んだ問題を特定するのを支援する機能を持つ．（詳細については 9.5節を参照.）
- リビジョン管理ツールは，プロジェクトの複数のバージョンでの同時作業や，リビジョン間の移行を支援する．

これらの理由の多くは自分自身のプロジェクトで作業していても，100 人の共同作業者と作業していても少なくとも理論的には等しく有意である．

リビジョンコントロールが実用的であるかどうか判断する鍵は，さまざまな開発スケール（“1 人のハッカー” レベルから “大規模チーム” レベルまで）において，支払うコストに対してどれだけ効能が得られるかということである．理解や使用が困難なリビジョンコントロールツールは高いコストを課すことになる．

500 人からなるプロジェクトでは，リビジョンコントロールツールなしでは殆んど立ち行かない．この場合，リビジョンコントロールなしでは明らかに失敗するため，リビジョンコントロールを行うコストは特に問題とはならない．

一方，1 人の開発者による “クイックハック” はリビジョンコントロールツールを使うにはあまりふさわしくない．なぜなら，ツールを使うコストがほぼプロジェクトのコストそのものであるからだ．これは正しいだろうか？

Mercurial はこれらの開発スケールの両方をサポートしている．基礎的な使用法は数分で学ぶことができ，リビジョンコントロールを最小規模のプロジェクトに簡単に取り入れることができる．単純であるため，難解なコンセプトやコマンドシーケンスに意識の多くを占められ，本当にやりたいことが疎かになることもない．また同時に Mercurial の性能の高さや，ピアツーピアの性質のために，大規模プロジェクトにも苦労する事なくスケールすることができる．

お粗末なプロジェクトを救済するようなリビジョンコントロールは存在しないが，良いツールの選択は，作業するプロジェクトの堅実さに大きな差をもたらす．

1.1.2 様々なリビジョンコントロール

リビジョンコントロールは大きな幅をもつ分野であり，そのため多くの呼び名とその短縮形が知られている：

- リビジョンコントロール (Revision control (RCS))
- ソフトウェア設定マネジメントまたは設定マネジメント (Software configuration management (SCM), or configuration management)
- ソースコードマネジメント (Source code management)
- ソースコードコントロールまたはソースコントロール (Source code control, or source control)
- バージョンコントロール (Version control (VCS))

これらの用語は各々違う意味を持つのだと主張する人々もいる．しかし実質的にはこれらは互いに大きく重なっており，わざわざ区別することは一般的でなく，また有用でもない．

1.2 リビジョンコントロールの歴史

最も知られている古いリビジョンコントロールツールは，Marc Rochkind が 1970 年代初頭に Bell 研究所で書いた SCCS (Source Code Control System) である． SCCS は個々のファイルに対して動作し，プロジェクトの共同作業者には同一マシン上の共有ワークスペースへのアクセスが必要であった．ファイルへのアクセスの調停はロックによって行われ，あるファイルを変更できるのは常に一人のユーザだけであった．ファイルをロックした後，ロックの解除を忘れることは日常的にあり，こうなると管理者の助けなしに他の開発者がファイルを変更することはできなかった．

Walter Tichy は，1980 年代初頭に SCCS の代替となるフリーのバージョンコントロールツールを開発した．彼は自らのシステムを RCS (Revison Control System) と呼んだ．SCCS 同様，RCS は開発者たちに単一の共有ワークスペースと，ファイルを同時に複数人が変更することのないようにロックを要求した．

1980 年代後半に Dick Grune は RCS を呼び出すシェルスクリプトによるバージョン管理システムを作った．初期に cmt と呼ばれたこのシステムは，後に CVS (Concurrent Versions System) と改名された． CVS の大きな革新は，開発者達に同時に個別のワークスペースで作業することを許したことである．ワークスペースを個別にしたことで，開発者は SCCS や RCS で良くあったように，他の開発者の作業を妨げることがなくなった．このモデルでは，中央のリポジトリに変更をコミットする前に，変更結果をマージする必要があった．

Brian Berliner は Grune のオリジナルスクリプトを受け継いで，それを C で書き直し，現在の CVS へと繋がるコードを 1989 年にリリースした．その後，CVS はネットワークを経由した動作や，クライアントサーバアーキテクチャを備えていった．CVS のアーキテクチャは中央集中型で，サーバのみがプロジェクトの履歴を保存する．クライアントのワークスペースはプロジェクトの最新バージョンのファイルのコピーであり，サーバの所在を示す僅かなメタデータが付加されていた．CVS は大成功を収め，おそらく世界で最も広く用いられたリビジョンコントロールシステムとなった．

1990 年代初頭，Sun Microsystems は TeamWare と呼ばれる初期の分散リビジョンコントロールシステムを開発した．TeamWare ワークスペースはプロジェクトの履歴の完全なコピーを持っていた．TeamWare には中央

リポジトリという概念はなかった.（CVS が履歴の記録に RCS を使っていたように，TeamWare は SCCS を用いていた.）

1990 年代中頃になると，CVS の問題が広く知られるようになってきた． CVS は一度に複数のファイルに対して行われる変更を論理的にアトミックな操作[1]としてグループ化するのではなく，ファイル毎に個別に記録していた．CVS のファイルヒエラルキーの管理は不十分で，ファイルやディレクトリをリネームすると簡単にリポジトリが混乱した．さらに悪いことに，CVS のソースコードは読みにくく，メンテナンスも難しかったため，アーキテクチャの問題を解決するのは不可能なレベルと言えた.

2001 年，CVS を開発していた Jim Blandy と Karl Fogel の 2 人の開発者が CVS を置き換える，より優れたアーキテクチャと奇麗なコードを持つツールのプロジェクトを始めた．その成果物である Subversion は CVS の集中型クライアントサーバモデルを改めることはしなかったが，複数ファイルのアトミックなコミットを追加し，名前空間の管理も改良していた．また CVS よりも優れた数多くの機能も追加された．Subversion は最初のリリースから急速に人気を獲得していった.

ほぼ時を同じくして，Graydon Hoare は Monotone と呼ばれる野心的な分散リビジョンコントロールシステムの開発を始めた．Monotone は CVS の数多くの設計上の瑕疵を修正し，ピアツーピアアーキテクチャを持っている． Monotone は初期の（あるいは後続の）リビジョンコントロールツールよりも先進的な機能を持っている．Monotone は暗号化されたハッシュを識別子として使用し，様々な出処のコードに対して “信頼性” の概念を持っていた.

Mercurial は 2005 年に誕生した．デザインのいくつかの面は Monotone に影響を受けているが，Mercurial は使いやすさ，高性能，大規模プロジェクトへのスケーラビリティにフォーカスしている.

1.3 リビジョンコントロールのトレンド

リビジョンコントロールツールの開発と使用において，過去 20 年間ツールに親しみ，ツールの制限を知るに従って，紛れもないトレンドが存在している.

第一世代のツールは単一のファイルを個別のコンピュータの上で管理した．アドホックな手動によるリビジョンコントロールと比べて大幅な進歩があったが，ロックモデルと単一コンピュータへの依存のため，利用は小規模で緊密なチームに限られた.

第二世代のツールは，ネットワーク中心のアーキテクチャに移行することで，それまでの制限を緩和し，プロジェクト全体を同時に管理した．プロジェクトが大きく成長すると，新たな問題に直面した．クライアントとサーバが頻繁に通信するため，大規模プロジェクトではサーバの規模が問題になった．信頼性のないネットワーク接続はリモートユーザがサーバと通信するのを妨げた．オープンソースプロジェクトがコミット権のないユーザにも匿名の読み出し専用アクセスを提供するようになると，ユーザらはこれらのツールが行った変更を記録できず，プロジェクトとのやりとりが不便であることに不満を募らせていった.

現行世代のリビジョンコントロールツールは，ピアツーピアである．これらのシステムの全てが単一の中央サーバへ依存しなくなっており，リビジョンコントロールデータを必要なところへ分散させることができるようになっている．インターネットを通じた共同作業は技術的制約から離れて，選択と合意によって行われるようになった．現代のツールはオフラインのままでも，自律的にも動作するようになっている．ネットワーク接続は変更を別のリポジトリと同期させる時にのみ必要である.

1.4 分散リビジョンコントロールの利点

分散リビジョンコントロールツールはもう数年も前から前世代のツールと同様に堅牢かつ有用なものと認められているにもかかわらず，古いツールのユーザたちはその利点を知らない．分散ツールが中央集中ツールよりも優れいている点は多々ある.

個人の開発者にとっては，分散ツールはほとんどの場合，中央集中ツールよりも高速である．その理由は，中央集中ツールではほとんどのメタデータは中央サーバで単一コピーとして保管されており，通常のオペレーションの多くをネットワークを経由して行う必要があるからだ．分散ツールは全てのメタデータをローカルに保存する．その他も同様で，中央集中ツールはネットワーク越しの通信によってオーバヘッドを生じる．開発

[1] 訳注：あたかも原子のように，複数の要素に分解できない操作を言う

中，リビジョンコントロールソフトウェアの操作を幾度となく行うことを考えれば，ツールの僅かなオーバヘッドでも過小評価すべきではない．

分散ツールはメタデータを様々な場所に複製するため，サーバインフラストラクチャの障害に関わりなく動作する．もし中央集中システムを使っていて，サーバが火災にあったとしたら，信頼できるバックアップメディアに最近作成したバックアップコピーが残っており，それがまともに機能することを祈ることになる．分散ツールでは，協力者のコンピュータの中に数多くのバックアップが残されている．

分散ツールでは，ネットワークの信頼性の与える影響は集中ツールに比べて遥かに小さい．集中ツールは，ネットワークに接続しなければ，大きな制限のあるいくつかのコマンド以外は使用できない．分散ツールでは作業中にネットワーク接続が断たれたとしてもそれに気づくことすらないだろう．他のコンピュータのリポジトリとの通信を行う動作のみが影響を受ける．このような動作はローカルでの動作より相対的に少ないはずだ．これが重大な問題となるのは，広範囲にわたるチームで作業をしている場合であろう．

1.4.1 オープンソースプロジェクトでの利点

オープンソースプロジェクトが好きになり，作業を始めようとするとき，プロジェクトが分散リビジョンコントロールツールを使っていれば，ただちにプロジェクトのコアメンバーの仲間となる．彼らがリポジトリを公開していれば，直ちにプロジェクト履歴をコピーし，変更を行い，内部のメンバーが使っているのと全く同じツールを用いて作業結果を記録することができる．対称的にメンバーが中央集中型のツールを使っている場合，誰かがあなたに変更を中央のサーバにコミットする許可を与えない限り，ツールをリードオンリーモードで使用することになる．それまでは変更を記録することはできず，あなたのローカルな変更はリポジトリのクライアントコピーを更新するたびに破壊されるリスクを伴う．

フォークしても問題なし

オープンソースプロジェクトで分散リビジョンコントロールツールを用いることは，プロジェクトの開発をフォークさせるリスクがあると言われている．意見の相違や，開発者のグループ間での態度の違いから，彼らがそれ以上共に作業を続けることができないと決断することでフォークは起こる．双方の陣営はプロジェクトのソースコードのほぼ完全なコピーからそれぞれの方向に別れていく．

時にはフォークした陣営が，互いのコードの差異を解消することもある．中央集中リビジョンコントロールシステムでは，差異を**技術的に**解決する過程に困難を伴い，多くの場合，手動で解消する必要がある．どのリビジョン履歴を残すのか決め，ほかのチームによる変更をツリーへなんらかの方法で継ぎ木する必要がある．この操作では，通常，一方のリビジョン履歴の一部または全てを失うことになる．

分散ツールは，フォークをプロジェクトを進めるための一つの方法として扱うにすぎない．行った変更すべては潜在的にフォークポイントになりうる．分散リビジョンコントロールツールでは，フォークは日常的に起き，これを取り扱うことは動作の根本である．そのためフォーク間の**マージ**には極めて優れており，これが分散ツールによるアプローチの大きな利点となっている．

各人が行う全ての部分作業がフォークとマージに位置付けられるならば，オープンソース界は“フォーク”を**純粋に**社会的な事象として扱うだろう．いずれにせよ分散ツールはフォークの蓋然性を**下げる**：

- 中央集中的なツールが課す，コミット権を持った内部の人間と持たない外部の人間の社会的な区別を取り去る
- リビジョンコントロールソフトウェアの観点からは同じマージであるため，開発コミュニティのフォーク後に生じた差異を解消するのが容易になる．

プロジェクトを厳格にコントロールしたいために分散ツールに抗う人々もいる．彼らは中央集中ツールがこのようなコントロールを与えると考えている．しかしそう思っていても，CVS や Subversion リポジトリを公開すれば，（時間はかかっても）プロジェクトの履歴全体を取得して，コントロールの手の及ばないどこかでそれを再現する方法はいくらでもある．結局，履歴をミラーし，フォークするような流動的な協力を排除するようなコントロールは非現実的である．

1.4.2 商用プロジェクトでの利点

商用プロジェクトの多くは地理的に広がったチームによって開発されている．中央サーバは，遠く離れた協力者からはコマンド実行が遅かったり，信頼性が低かったりするように見える．商用リビジョンコントロールシステムはこの問題の解決にリモートサイトの複製を作成するアドオンを提供している．これらの多くは高価だったり，管理が複雑だという問題を持っている．分散システムにはそもそもこれらの問題がない．さらに好ましいことに，複数のサイト毎に正式なサーバを簡単に設定することができ，高価な長距離のネットワークリンク上に冗長な通信を行うことがない．

中央集中型のリビジョンコントロールシステムのスケーラビリティは相対的に小さく，数十人のユーザの同時アクセスによる負荷で中央の高価なシステムが停止することも珍しくない．しかし，これに高価で複雑な複製機能を追加することがよく行わてしまう．ただ一つの中央サーバのみを持つ場合でも，分散ツールを用いることで（データはすべてあらゆるところに複製されるため）中央サーバの負荷は数分の一に抑えられる．このため単一の安価なサーバで大きなチームの需要を満たすことができ，負荷分散のためのデータの複製もスクリプトだけで実現できる．

顧客の側でこの領域の問題解決を行う従業員がいれば，分散リビジョンコントロールの利益を得ることができる．ツールを使うことで，カスタムビルド，互いに独立した修正のテスト，プロジェクトの履歴からバグやリグレッションの原因の探索などを顧客の環境でネットワークに接続する必要なく実現できる．

1.5 Mercurial を選ぶ理由

Mercurial は，リビジョンコントロールシステムとして選択するのにふさわしいユニークな性質を持っている．

- 学習と利用が簡単
- 軽量である
- 極めて良好にスケールする
- カスタマイズが容易である

読者がリビジョンコントロールに慣れているのであれば， Mercurial を 5 分以内に使い始めることができるだろう． 仮に 5 分が無理でも，あと数分もあれば十分に違いない． Mercurial のコマンドと機能セットは全体に均一で一貫性があるので，多数の例外を覚えるのではなく，共通するルールさえ覚えておけばよい．

小さなプロジェクトでは，すぐに Mercurial を使い始めることができる．新しい変更とブランチを作り,（ローカルやネットワーク越しに）変更を転送し，履歴と状態に関する動作はすべて高速である． Mercurial の個々の動作は極めて高速であり，全体としても素早く，オーバヘッドがほとんど知覚できないような動作をするようになっている．

Mercurial の有用性は，小さなプロジェクトに限らない．Mercurial は数百人から数千人の貢献者を擁し，数万のファイルからなる数百メガバイトにも及ぶソースコードからなるプロジェクトでも有効である．

Mercurial の核になる機能が十分でなかった場合，拡張することはたやすい．Mercurial はスクリプトとよく馴染む．また整った内部構造と Python による実装のため，拡張の形で新しい機能を追加するのが容易である．バグの識別を助けるものから，性能を改善するものまで，人気の高い有用な拡張がすでに数多く存在している．

1.6 Mercurial と他のツールの比較

この節を読む前に，これは筆者自身の体験と興味，そして（敢えて言えば）バイアスがかかっていることを理解して欲しい．筆者は下記のリビジョンコントロールツールのすべてを使用したことがあり，その使用期間は多くの場合数年に及ぶ．

1.6.1 Subversion

Subversion は，CVS を置き換えるべく開発された，人気の高いリビジョンコントロールツールである．これも中央集中型のクライアントサーバアーキテクチャを持つ．

Subversion と Mercurial では，同じ動作のためのコマンドに同様の名前が付けられているため，どちらかに慣れていればもう一方を学ぶのはたやすい．どちらのツールもすべての人気の高いオペレーティングシステムに移植されている．

Mercurial は筆者がベンチマークを行った全てのリビジョンコントロール操作において Subversion よりも明確な性能上の優位性を持っている．筆者は 2 から 6 までの範囲で優位性を Subversion 1.4.3 で利用可能な中で最も高速な *ra_***ローカル**ファイル保存と比較した．より現実的な利用ではネットワークを利用したファイルサービスを用いることになり，Subversion では明確に大きな不利がある．多くの Subversion コマンドはサーバと通信する必要があり，Subversion は実用的な複製機構を持たないため，一定以上の規模のプロジェクトでは，サーバキャパシティとネットワークのバンド幅がボトルネックとなるからである．

さらに，Subversion は更新されたファイルの探索 (`status`) と，現在のリビジョンに対する差分のを表示 (`diff`) などのいくつかの操作でネットワークトランザクションを避けるために明確なストレージオーバヘッドをもたらす．結果として，Subversion のワーキングコピーは，プロジェクトの完全な履歴を含む Mercurial のリポジトリ及びワーキングディレクトリと同じかより大きなサイズとなってしまう．

Subversion には多くのサードパーティツールがある．Mercurial はこの点でかなり遅れている．ギャップは縮まりつつあるが，Mercurial の GUI ツールのいくつかは，対応する Subversion 用のものよりも秀でている．Subversion には Mercurial と同様に優れたユーザマニュアルがある．

Subversion はリビジョンの履歴をクライアント内に持たないため，サイズが大きく内容の明らかでない多数のバイナリファイルを取り扱うのには向いている．圧縮の効かない 10MB のファイルに対して 50 リビジョンをチェックインする場合，Subversion のクライアント側のストレージの使用量は一定である．分散 SCM では，ストレージ使用量は各々のリビジョン間での差分が大きいため，リビジョン数に比例してすぐに増えてしまう．

加えて，異なるバージョンのバイナリファイルをマージするのは不可能であったり，困難であったりする．Subversion ではファイルロック機能によってユーザが一時的にファイルへの変更をコミットする排他的な権利を得ることができる．これはバイナリファイルを広汎に使うプロジェクトでは大きな利点になり得る．

Mercurial は Subversion リポジトリからリビジョン履歴をインポートすることができる．また，リビジョン履歴を Subversion リポジトリにエクスポートすることも可能だ．このため，移行を決める前に Mercurial を試したり，Mercurial を Subversion と並行して利用することが容易である．履歴の変換は漸進的に行えるため，最初に変換を行った後，新たな変更を取り込むために追加の変換を行うことができる．

1.6.2 Git

Git は Linux カーネルのソースツリーを扱うために開発された分散リビジョンコントロールツールである．Mercurial 同様，その初期のデザインは多少 Monotone に影響を受けている．

Git は非常に大規模なコマンドセットを持っている．version 1.5.0 では 139 にも及ぶコマンドが用意されている．このため，Git はしばしば習得しづらいという評判されている．Mercurial は Git と比べると簡潔さに強くフォーカスしている．

性能に関しては，Git は極めて高速である．Linux 環境でのテストでは，いくつかの操作は Git は Mercurial よりも高速であった．しかし Windows 環境では，この本の執筆時点では，性能や Git の提供する機能は Mercurial に大きく劣った．

Mercurial リポジトリはメンテナンスを必要としないが，Git リポジトリはしばしば手動でメタデータの “repacks” が必要となる．これを行わない場合，性能は劣化し，必要な記憶領域も急速に拡大する．Git リポジトリを規則正しく頻繁に repack しないサーバではバックアップの際にディスク使用が極めて多くなり，結果として毎日のバックアップに 24 時間以上を費やすことになる．新規に pack された Git リポジトリは Mercurial リポジトリよりやや小さいが，unpack 状態の Git リポジトリは数段大きくなる．

Git のコアは C で書かれている．Git コマンドの多くはシェルや Perl のスクリプトとして実装されており，それらのスクリプトの品質は様々である．数度にわたって致命的なエラーがあるにもかかわらず，スクリプトが盲滅法に動作することを体験した．

Mercurial は Git リポジトリからリビジョン履歴をインポートすることができる．

1.6.3 CVS

CVS はおそらく世界中で最も広範に使用されているリビジョンコントロールツールであろう．その古さと内部の乱雑さのため，長らく軽いメンテナンスのみが行われてきた．

CVS は集中型のクライアントサーバアーキテクチャを持つ．CVS は，関連したファイル変更をグループ化し，アトミックにコミットすることができない．このためユーザがコミットによってビルドを破壊することが起こる．誰かが変更の一部を成功裏にコミットしたものの，残りの部分はマージが必要なためブロックされている場合，他のユーザは必要な変更の一部しか見ることができない．同じことはプロジェクトの履歴に対しても起こる．誰かの作業に関係する変更すべてを参照したい場合,（どのファイルが何なのか知っていれば）関連する各ファイルについて，変更の説明とタイムスタンプを自力で調べて変更を特定する必要がある．

CVS のタグとブランチは語る気さえ起きないような混乱した形式を持っている．きちんとしたファイルやディレクトリのリネームをサポートしないため，リポジトリが簡単に壊れてしまう． cvs は内部で一貫性をチェックする機能がなく，リポジトリの破損を知らせることもできない．筆者は既存または新規を問わず，どのようなプロジェクトであっても CVS の使用は薦めない．

Mercurial は CVS のリビジョン履歴をインポートすることができる．しかし他のリビジョンコントロールツールの CVS インポーター同様いくつかの制限もある．CVS にはアトミックチェンジがなく，ファイルシステム階層をバージョン管理する能力もないため，CVS の履歴を完全かつ正確に再現することはできない．再現にはいくつかの推測が入り，リネームは通常再現されない．CVS では高度な管理の多くが手動で行われるため，間違いが起こりやすく，破損したリポジトリで CVS インポーターが同時に複数の問題に見舞われることがよくある.（筆者の不愉快な体験では，10 年以上にわたってロックされたままの完全に無意味なリビジョンタイムスタンプやファイルを見たことがある.）

Mercurial は CVS リポジトリからリビジョン履歴をインポートすることができる．

1.6.4 商用ツール

Perforce は集中型のクライアント・サーバアーキテクチャを持ち，いかなるデータもクライアント側でキャッシュしない．現在のリビジョンコントロールツールとことなり，Perforce ではユーザがどのファイルを編集するのかサーバに知らせるコマンドを実行する必要がある．

Perforce の性能は，小規模なチームでの作業においてはかなり良い．しかしユーザ数が数ダース以上に増加するに従って，急激に悪くなっていく．かなり大規模な Perforce 環境では，ユーザの操作による負荷を軽減するためのプロキシが必要になる．

1.6.5 リビジョンコントロールツールを選ぶ

CVS を除いて，上記のツールは特定の作業スタイルに合った固有の強みを持っている．すべての状況でベストであるツールというものは存在しない．

例えば，頻繁に変更されるバイナリファイルを扱う場合は Subverrsion を選ぶのが良い．Subversion の集中型の性質とファイルロックのサポートはバイナリファイルの取り扱いに適している．

個人的には，数年にわたって，Mercurial の簡潔さ，性能，優れたマージサポートは，乗り換えるに値する組合せだと感じている．

1.7 他のツールから Mercurial への移行

Mercurial には `convert` という拡張が同梱されている．この拡張はリビジョン履歴を他のリビジョンコントロールツールからインクリメンタルにインポートする.「インクリメンタル」というのは，まずプロジェクトの履歴すべてを変換した後で，元のリポジトリに起きた変更をさらに変換して取り込めるという意味である．

`convert` は次のリビジョンコントロールツールをサポートする:

- Subversion
- CVS
- Git
- Darcs

加えて，convert は Mercurial から Subversion に変更をエクスポートすることができる．これにより，Subversion から Mercurial に切替える前に，行った作業の結果を失うことなく，両者を並行して試すことができる．

“hg convert” コマンドを使うのはたやすい．単にソースリポジトリのパスや URL を渡すだけで，使用可能なリポジトリを作成する．また，オプションで変換後のリポジトリの名前を渡すこともできる．最初の変換後，同じコマンドを再び実行すると新しい変更を取り込むことができる．

Chapter 2

Mercurialツアー: 基本

2.1 システムへのMercurialのインストール

人気の高いオペレーティングシステムではMercurialのビルド済みバイナリパッケージが提供されている．これを用いれば読者の計算機の上で直ちにMercurialを使うことができる．

2.1.1 Windows

Windows向けMercurialの最も優れたバージョンであるTortoiseHgは，http://bitbucket.org/tortoisehg/stable/wiki/Homeで入手できる．このパッケージは外部のパッケージへ依存せず，すぐに動作する．コマンドラインインタフェースとグラフィカルユーザインタフェースの両方が利用できる．

2.1.2 Mac OS X

Lee CanteyはMac OS X向けのMercurialインストーラをhttp://mercurial.berkwood.comで配布している．

2.1.3 Linux

Linuxの各ディストリビューションはそれぞれ独自のパッケージツール，ポリシー，開発ペースを持っているため，全てを網羅してMercurialバイナリのインストール手順を述べることは困難である．それぞれのディストリビューションで利用可能なMercurialのバージョンは，パッケージメンテナの活動に依存する．

単純化のために，大多数のLinuxディストリビューションでコマンドラインからMercurialをインストールする方法に限定して説明することにする．これらの大半のディストリビューションでは，Mercurialをクリック1回でインストールできるようなグラフィカルなパッケージマネージャを用意している．

Debian and Ubuntu
```
apt-get install mercurial
```

Fedora and OpenSUSE
```
yum install mercurial
```

Gentoo
```
emerge mercurial
```

Ubuntuの Mercurialパッケージは Debianのものに比べて無視できないほどの遅れがある（これを書いている時点では7カ月）．このため，Ubuntuでは Debianではすでに解決された問題に直面するかもしれない．

2.1.4 Solaris

http://www.sunfreeware.comからビルド済みの Mercurialパッケージが入手できる．

2.2 Mercurialを使う

まず“hg version”コマンドでMercurialが実際に正しくインストールされたかを確認してみよう．表示されるかどうかが問題であって，表示されるバージョン情報は特に重要ではない．

```
$ hg version
Mercurial Distributed SCM (version 1.2.1)

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 組み込みヘルプ

Mercurialは組み込みのヘルプシステムを備えている．これはコマンドの使い方に詰まった時に役に立つ．何を実行したら良いのか全く分からない場合は単に“hg help”を実行する．これはコマンドの一部を説明と共にリスト表示する．特定のコマンドについてヘルプが見たい場合は，下記のようにするとより詳細な情報が得られる．

```
$ 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 is 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
```

通常必要としないような極めて詳しい説明が必要な場合は“hg help -v”を実行する．-vは--verboseオプションの短縮形で，より詳細な情報を表示するオプションである．

2.3 リポジトリを使った作業

Mercurialでは**リポジトリ内**で一切が起きる．プロジェクトのリポジトリは所属する全てのファイルとそれらの履歴情報を持つ．

リポジトリには特に不思議なところはない．リポジトリはMercurialが特別の扱いをするだけのファイルシステム上の単なるディレクトリツリーにすぎない．リポジトリはコマンドラインやファイルブラウザからいつでも名前を変えたり消去することができる．

2.3.1 リポジトリのローカルコピーを作る

リポジトリの**コピー**はやや特殊である．通常のファイルコピーコマンドを使ってリポジトリのコピーを作成することもできるが，Mercurial の組み込みコマンドを使ってコピーするのが一番良い．このコマンドは“hg clone”と呼ばれ，既存のリポジトリの完全なコピーを作成する．

```
$ 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
```

“hg clone”コマンドを使う利点の一つは，上で見たようにリポジトリをネットワーク越しにクローンできる点である．またもう一点，このコマンドはどこからクローンしたのかを記録するため，新たな変更を別のリポジトリから取得しようとする際に便利である．

クローンが成功すると hello というローカルディレクトリができる．このディレクトリにはオリジナルと同一のファイルが含まれる．

```
$ ls -l
total 0
drwxr-xr-x 3 yaz users 120 Jun  9 06:07 hello
$ ls hello
Makefile  hello.c
```

これらのファイルはクローンしたリポジトリ内と全く同じ内容と履歴を持っている．

全ての Mercurial リポジトリは完全かつ自己充足的で独立である．リポジトリはプロジェクトに属すファイルのプライベートコピーと履歴を持つ．今述べたように，クローンされたリポジトリはクローン元のリポジトリの場所を記憶しているが，ユーザが指示しない限り，そのリポジトリや他のリポジトリと通信を行うことはない．

今の段階ではローカルなリポジトリとは外部へ何の影響も及ぼさないプライベートな“サンドボックス”で，この中でどんなことでも試すことができると解釈しておけば十分である．

2.3.2 リポジトリには何が含まれるか？

リポジトリの内部をより詳しく見てみると，.hg というディレクトリがあるのに気づく．Mercurial はここにリポジトリのためのメタデータを保管している．

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

.hg ディレクトリの中身と，このディレクトリのサブディレクトリは Mercurial 専用のものである．リポジトリのそれ以外のファイルとディレクトリはユーザに属す．

ここで少々用語を定義しようと思う．.hg ディレクトリを“リアル”リポジトリ，このディレクトリと一緒に扱われるファイルやディレクトリを**ワーキングディレクトリ**と呼ぶことにする．これらを簡単に区別するために，**リポジトリ**はプロジェクトの**履歴**を保存し，**ワーキングディレクトリ**はプロジェクトの履歴の中のある時点の**スナップショット**を持つと覚えると良い．

2.4 履歴を辿る

未知のリポジトリに対してまずしようと思うことは，そのリポジトリでの変更の履歴を知ることだろう．履歴は“hg log”コマンドで見ることができる．

```
$ 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

```

デフォルトでは，このコマンドはプロジェクトに対して行われた変更の各々について簡潔なパラグラフを表示する． Mercurial の用語では，履歴中の変更のイベントを**チェンジセット**と呼ぶ．その理由は，複数のファイルに対する変更の記録を持ち得るからである．

“hg log”から出力される記録の各フィールドは次のようになっている．

changeset 番号，それに続くコロンおよび 16 進文字列．これはチェンジセットの**識別子**である．16 進文字列は固有の識別子で，同一の識別子は常に同じチェンジセットを指す．番号は短く入力も 16 進文字列より容易であるが，チェンジセットに固有ではない．同じ番号でもリポジトリの別のクローンでは，違うチェンジセットを指す可能性がある．番号は単にローカルな利便のために提供されている．

user チェンジセット作成者．このフィールドの書式は自由だが，ほとんどの場合氏名と email アドレスである．

date チェンジセットが作成された日付と時刻およびタイムゾーン． （日時はチェンジセットの作成者の属すタイムゾーンにローカルである．）

summary テキストメッセージの最初の行はチェンジセットの説明に入力されたチェンジセットの作成者である．

- 上のリストの最初のチェンジセットのように tag フィールドを持つチェンジセットもある．タグはチェンジセットを識別するためのもう一つの方法で，覚えやすい名前を自由に付けることができる．（タグ tip は特別で，常にリポジトリの最新のチェンジセットを指す．）

“hg log” のデフォルト出力は要約にすぎず，多くの詳細情報を欠いている．

図 2.1は hello リポジトリの履歴をグラフ表示したものである．この表示によってどの方向の履歴が順方向なのか理解しやすくなっている．今後，この図を何度か参照する．

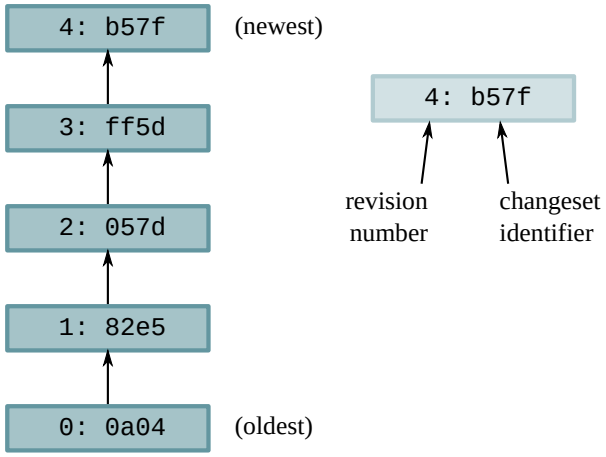

Figure 2.1: hello リポジトリの履歴グラフ

2.4.1 チェンジセット, リビジョン, 他のユーザとのやりとり

英語はいい加減なことで悪名の高い言語であり，コンピュータサイエンスでは専門用語の混乱が甚だしい．(4 人が 1 つの用語を使うことなど有り得ない.）リビジョンコントロールは沢山の同義語を持っている．他の人と Mercurial の履歴について話すとき，“チェンジセット” がしばしば “チェンジ” に略されたり，書き言葉では “cset” などとされたり，時にはチェンジセットが “リビジョン” や “rev” と表されたりすることに気づくだろう．

“a changeset” というチェンジセットの**呼び名**は自由だが，“a *specific* changeset” を参照する**識別子**は非常に重要である．“hg log” コマンドの出力に含まれる changeset フィールドは，あるチェンジセットを番号と 16 進文字列で表していたことを思い出して欲しい．

- リビジョン番号は**そのリポジトリに限って有効**な簡便記法である．
- 16 進文字列は**永続的かつ不変の識別子**で，リポジトリのコピー**全て**で常に特定のチェンジセットを示す．

この区別は重要である．誰かに “revision 33” と言った時，そのリビジョン 33 が自分のリポジトリのリポジトリのものとは**違うものである**可能性は高い．その理由は，リビジョン番号はリポジトリに変更が現れた順序によって決まり，Mercurial では別のリポジトリで同じ変更が同じ順序で起こる保証はないためである． 3 つの変更 a,b,c は，あるリポジトリで $0,1,2$ の順で起こり，別のリポジトリでは $0,2,1$ の順序で起こり得る．

Mercurial はリビジョン番号を単に便利のための略記法として用いる．誰かとチェンジセットについて議論したり,（バグ報告などのために）チェンジセットを記録したい場合は 16 進の識別子を利用すべきである．

2.4.2 特定のリビジョンを見る

“hg log” の出力をある 1 つのリビジョンに制限するためには，-r（または --rev）オプションを用いる．リビジョン番号も 16 進文字列のチェンジセット識別子も利用可能であり，指定できるリビジョンの数に制限はない．

```
$ 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 0272e0d5a517
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 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.

```

いくつかのリビジョンの履歴を，いちいちリビジョンを指定することなく見たい時は，**範囲記法**が使える．これにより“*a* から *b* までに含まれる全てのリビジョン”を指定することができる．

```
$ 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 はリビジョンが指定された順序を考慮する．“`hg log -r 2:4`”は 2,3,4 の順に表示し，“`hg log -r 4:2`”は 4,3,2 の順に表示を行う．

2.4.3 より詳細な情報

探しているものがはっきりしている場合は“`hg log`”コマンドによるサマリー情報は有用だが，どのチェンジセットが見つけようとするチェンジセットなのか決める時には変更の完全な記述や変更されたファイルのリストが必要になるかもしれない．“`hg log`”コマンドの`-v`（または`--verbose`）オプションでより詳細な情報を見ることができる．

```
$ 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.


```

説明と変更の内容の両方を見たい場合は`-p`（または`--patch`）オプションを追加する．このオプションは変更の内容を *unified diff* 形式で表示する．(unified diff 形式を見たことがなければ概略を 12.4節で見ることができる.)

```
$ 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;
 }

```

`-p` オプションは驚異的に便利なので覚えておくと良い.

2.5 コマンドオプションのすべて

Mercurial コマンドを試すのを一休みして，コマンドの動作パターンについて議論してみよう．これはこのツアーを続ける上で役に立つだろう.

Mercurial はコマンドにオプションを渡す際に一貫性のある直接的なアプローチを取っている．これは現代の Linux および Unix に共通のオプションに関する習慣に基づいている.

- 全てのオプションは長形式のオプションを持つ．例えば，すでに見ているように“`hg log`”コマンドは`--rev`オプションを受け付ける.
- ほとんどのオプションは短縮名を持つ．`--rev` オプションの代わりに`-r` が使える．(いくつかのオプションで短縮名が使えない理由は，そのオプションがほとんど使用されないためである.)
- 長形式のオプションは 2 つのダッシュで始まる（例 `--rev`）．一方短形式のオプションは 1 つのダッシュで始まる（例 `-r`）.

- オプションの命名と使用法はコマンド間で一貫している．例を挙げると，チェンジセット ID またはリビジョン番号を指定させるコマンドの全てで`-r`と`--rev`の両方を受け付ける．
- 短形式のオプションを使っている時はあ，複数のオプションを組み合わせて入力の手間を省くことができる．例えば`-v -p -r 2`は`-vpr2`と書くことができる．

この本の例の全てで短形式のオプションを用いる．これは単に筆者の好みのためで，深い意味はない．

表示出力を行うほとんどのコマンドで`-v`（または`--verbose`）オプションを付けるとより詳細な出力を行い，`-q`（または`--quiet`）オプションを付けるとより簡潔な出力となる．

> **Note:** オプション名の一貫性
> Mercurial コマンドでは，同じ対象を扱う際，ほとんど常に同じオプション名を使うようになっている．例を挙げると，チェンジセットを扱うコマンドの場合は常に`--rev`や`-r`でチェンジセットの指定ができる．この一貫性により，組まんどのオプションが覚えやすくなっている．

2.6 変更の仕方，変更のレビュー

Mercurial で履歴を見る方法については理解した．ここでは何か変更を行って，これを見てみよう．

まず最初にこの実験を本来のリポジトリから隔離するために“`hg clone`”を行う．すでにローカルなリポジトリを持っているので，これをクローンするだけでよく，リモートのリポジトリをコピーする必要はない．ローカルなクローンはネットワーク越しに行うクローンよりもずっと速く，多くの場合使用するディスク容量も少ない．[1]一ファイルシステム上でリポジトリのクローンを行う場合， Mercurial はハードリンクを使い，内部メタデータをコピーオンライトで共有し，ディスク使用量の節約を行う．この説明がよくわからなくても心配する必要はない．この動作はすべて透過的かつ自動的に行われるため，気にする必要はない．

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

また，何か作業をしたい時にサンドボックスとするために一時的なクローンを作成し，リモートリポジトリのコピーを“清潔”に保つことはしばしば役に立つ．これにより複数の作業を並行して行うことができ，かつ，作業が完了し，再び統合するまで他の作業から隔離されたままにしておくことができる．リポジトリのローカルクローンはとても手軽で，クローンと廃棄にほとんどオーバヘッドがない．

`my-hello`リポジトリ内にはクラシックな“hello, world”プログラムである`hello.c`がある．

```
$ 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("hello, world!\");
        return 0;
}
```

[1]同

このファイルを 2 行目が出力されるように編集する．

```
# ... edit edit edit ...
$ 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("hello, world!\");
        printf("hello again!\n");
        return 0;
}
```

リポジトリ内のファイルに対して Mercurial が把握している内容を “hg status” コマンドで見ることができる．

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

“hg status” コマンドはいくつかのファイルに対しては出力を行わないが，hello.c に対して “M” を表示する．特に指示をしない場合，“hg status” コマンドは変更されていないファイルに対しては何も出力しない．

“M” の意味は，hello.c に対して行われた変更を Mercurial が把握したということである．ファイルの変更前に Mercurial にこれから変更するファイルを登録する必要はないし，変更後に行う必要もない．変更されたファイルの発見は自動的に行われる．

hello.c を編集したことが分かるのは多少役に立つが，知りたいのはむしろ**何を**変更したのかである．“hg diff” コマンドを使えばこれを知ることができる．

```
$ hg diff
diff -r 2278160e78d4 hello.c
--- a/hello.c        Sat Aug 16 22:16:53 2008 +0200
+++ b/hello.c        Tue Jun 09 06:07:26 2009 +0000
@@ -8,5 +8,6 @@
 int main(int argc, char **argv)
 {
         printf("hello, world!\");
+        printf("hello again!\n");
         return 0;
 }
```

> **Note:** パッチについて
> 上の出力の読み方が分からない場合は，12.4節を参照されたい．

2.7 新たなチェンジセットへ変更を記録する

ファイルを変更し，ビルドとテストを行い，“hg status” と “hg diff” を使って変更のレビューを行い，気の済むところまでこれを繰返し，仕事の結果を新たなチェンジセットに記録する．

新たなチェンジセットを作成するには“hg commit”コマンドを使う．この操作をよく“コミットする”とか“コミット”と呼ぶ．

2.7.1 ユーザ名を設定する

最初に“hg commit”を実行する場合，実行が成功するかどうかは保証されていない．すべてのコミットでMercurialはユーザの名前とアドレスを記録し，後で誰がその変更を行ったのか分かるようにしている． Mercurialは変更をコミットする際に自動的に妥当なユーザ名を付けようとする．ユーザ名の推測は以下のような順序で行われる：

1. “hg commit”コマンドに-uオプションとユーザ名を付けた場合，これが最も優先される．
2. 環境変数HGUSERを設定している場合は次にこれが参照される．
3. ホームディレクトリに.hgrcファイルを作っている場合，usernameエントリが次に参照される．このファイルの内容は2.7.1節で説明している．
4. 環境変数EMAILを設定しているなら次にこれが参照される．
5. Mercurialはシステムにローカルユーザ名とホスト名を問い合わせ，これらからコミットユーザ名を構成する．このユーザ名は役に立たないことが多いため，この方法を使った場合は警告メッセージが表示される．

これらのメカニズムのすべてが失敗した場合，Mercurialはエラーメッセージを表示して実行を断念する．この場合，ユーザ名を設定するまでコミットすることはできない．

環境変数HGUSERと“hg commit”コマンドの-uオプションは，Mercurialがデフォルトで設定するユーザ名を**オーバライド**する方法だと考えるとよい．通常の利用法でユーザ名を設定する最も簡単で最も頑強な方法は.hgrcファイルを作成することである．このやり方について次に述べる．

Mercurialの設定ファイルを作成する

ユーザ名を設定するには好みのエディタでホームディレクトリに.hgrcというファイルを作成する． Mercurialはこのファイルから個人設定を取得し，使用する．.hgrcファイルの最初の内容は以下のような書式にする．[2]

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

“[ui]”の行で設定ファイルの**セクション**が始まる．“username = ...”の行は“uiセクションのusername項目の値を設定する”と読むことができる．セクションは新たな別のセクションが始まるか，ファイルが終了するまで続く．Mercurialは空行を無視し，“#”から行末までをコメントとして扱う．

ユーザ名を選ぶ

usernameの設定に使用する文字列は，他のユーザが読む情報であるに過ぎず，Mercurialによって処理されるものではないため，どのような文字列でも構わない．多くのユーザが従う習慣は，上の例のように名前とemailアドレスを使うものである．

> **Note:** Mercurialの組み込みウェブサーバはemailアドレスをスパマーが使うemail収拾ツールに拾われ難くするように改変し，Mercurialリポジトリをウェブで公開してもスパムを受け取り難くしている．

[2] Windowsでの適切なディレクトリを示すこと．

2.7.2 コミットメッセージを書く

変更をコミットする時，Mercurial はチェンジセットで行った変更について説明するメッセージを入力させるためにテキストエディタを起動する．これは**コミットメッセージ**と呼ばれる．これは何をなぜ変更したのか読み手に伝えるもので，コミット完了後に “hg log” で表示することができる．

```
$ hg commit
```

“hg commit” コマンドが起動するエディタは，空行と “HG:” で始まる数行をすでに含んでいる．

```
is is where I type my commit comment.

: Enter commit message.  Lines beginning with 'HG:' are removed.
: --
: user: Bryan O'Sullivan <bos@serpentine.com>
: branch 'default'
: changed hello.c</programlisting>
```

Mercurial は “HG:” で始まる行を無視する．これらの行はどのファイルへの変更なのかをユーザに伝えるだけの目的で存在する．これらの行を変更したり消去したりしても何の影響も与えない．

2.7.3 よいコミットメッセージの書き方

デフォルトでは “hg log” はコミットメッセージの最初の行しか表示しないためコミットメッセージの最初の行は 1 行で完結するように書くとよい．ガイドラインに**従わない**ため読めないコミットメッセージの例を示す．

```
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残りの部分には厳格なルールはない．プロジェクトでフォーマットに関して要求するポリシーがあったとしても，Mercurial はコミットメッセージを中断したり，特別の配慮をすることはない．

個人的には，簡潔かつ情報があり，“hg log --patch” を一見しただけではわからない部分について説明したコミットメッセージが好みである．

2.7.4 コミットを中止する

コミットメッセージを編集中にコミットを取り止めたくなった時は，編集中のファイルをセーブせずにエディタを終了すればよい．こうするとリポジトリにもワーキングディレクトリにも何も起きない．

“hg commit” コマンドを引数なしで実行している場合，“hg status” コマンドや “hg diff” コマンドに現れるそれまでに行った変更は保存される．

2.7.5 新たな作業を称賛する

コミットの完了後，“hg tip” コマンドで今作成したチェンジセットを知ることができる．このコマンドの出力はリポジトリの最新リビジョンのみを表示するという点を除けば “hg log” と全く同じである．

```
$ hg tip -vp
changeset:   5:090dbfe691cf
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:26 2009 +0000
```

```
files:       hello.c
description:
Added an extra line of output


diff -r 2278160e78d4 -r 090dbfe691cf hello.c
--- a/hello.c        Sat Aug 16 22:16:53 2008 +0200
+++ b/hello.c        Tue Jun 09 06:07:26 2009 +0000
@@ -8,5 +8,6 @@
 int main(int argc, char **argv)
 {
         printf("hello, world!\");
+        printf("hello again!\n");
         return 0;
 }

```

リポジトリの最新リビジョンは *tip* **リビジョン**または単に *tip* と呼ばれる．

“hg tip” コマンドでは “hg log” コマンドのオプションの多くが使える．上記の-v は “冗長” な表示を行うし，-p は “パッチの表示” を行う．-p でパッチが表示されるのは，前述のオプション名の一貫性を示すもう一つの例である．

2.8 変更を共有する

Mercurial のリポジトリは自己充足的であるとすでに述べた．つまり，今作成したチェンジセットは my-hello リポジトリにのみ存在している．この変更を他のリポジトリに波及させるいくつかの方法を見てみよう．

2.8.1 他のリポジトリから変更を pull する

始めるに当たって，今コミットした変更を含まないオリジナルの hello リポジトリをクローンする．この一時的なリポジトリを hello-pull と呼ぶことにする．

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

“hg pull” コマンドで変更を my-hello から hello-pull に取り込む．未知の変更を盲目的にリポジトリに pull することは少々恐ろしい．Mercurial には “hg pull” コマンドでどのような変更がリポジトリに取り込まれるのかを実際には pull することなく表示する “hg incoming” コマンドがある．

```
$ cd hello-pull
$ hg incoming ../my-hello
comparing with ../my-hello
searching for changes
changeset:   5:090dbfe691cf
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:26 2009 +0000
summary:     Added an extra line of output

```

おそらく読者はネットワーク上のどこかからリポジトリを pull するに違いない．変更を pull する前，“hg incoming” の出力を見ている間に誰かがリモートリポジトリに何かコミットすることも有り得る．この場合，“hg incoming” で見たものよりも多くの変更が pull されることになる．

変更をリポジトリに取り込むのは，取り込み元のリポジトリを指定して “hg pull” コマンドを実行するだけのシンプルな操作である．

```
$ 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:   5:090dbfe691cf
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:26 2009 +0000
summary:     Added an extra line of output

```

pull の前後の “hg tip” 出力でリポジトリへの変更の取り込みに成功したかどうかがわかる．ワーキングディレクトリに変更を反映させるにはもう 1 ステップが必要である．

2.8.2 ワーキングディレクトリを更新する

ここまではリポジトリとワーキングディレクトリの関係をあまり明確にせずに来た．2.8.1節で “hg pull” コマンドを使って変更をリポジトリに取り込んだが，注意深く見ると，それらの変更はワーキングディレクトリに反映されていなかった．これは，“hg pull” コマンドは（デフォルトでは）ワーキングディレクトリに手を付けないためである．ワーキングディレクトリを操作するためには “hg update” コマンドを使う．

```
$ 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” がワーキングディレクトリを自動的に更新しないのはいささか奇妙に見えるかもしれない．しかしこれにはちゃんとした理由がある．ワーキングディレクトリは “hg update” でリポジトリのいかなるバージョンにも更新できる．バグの原因を探るためにワーキングディレクトリを古いリビジョンにしている時，“hg pull” を実行したためにワーキングディレクトリが最新のリビジョンになったとしたら，受ける被害は少なくない．

しかしながら,「pull 後に更新」という操作はとてもよく行われるため，“hg pull” コマンドにはこれら 2 つの操作を一度に行うオプション，-u がある．

```
hg pull -u
```

2.8.1節で-u オプションなしで実行されている “hg pull” コマンドの出力を改めて見てみると，ワーキングディレクトリを明示的に更新しなければならないことを示す

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

という注意書きがあるのに気づく．

ワーキングディレクトリの現在のリビジョンを調べるには “hg parents” コマンドを使う．

```
$ hg parents
changeset:   5:090dbfe691cf
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:26 2009 +0000
summary:     Added an extra line of output

```

図 2.1を見直すと，各々のチェンジセットを結ぶ矢印がある．矢印の出ているところが親で，矢印が示しているところが子である．ワーキングディレクトリは同様に 1 つの親を持ち，これがワーキングディレクトリが現在更新されているチェンジセットである．

特定のリビジョンにワーキングディレクトリを更新するためには，“hg update” コマンドにリビジョン番号またはチェンジセット ID を渡す．

```
$ 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
```

明示的にリビジョンを渡さなかった場合，上の例で 2 回目の “hg update” の実行のように “hg update” コマンドはワーキングディレクトリを tip リビジョンへ更新する．

2.8.3 他のリポジトリに変更を push する

Mercurial では現在使用しているリポジトリから他のリポジトリに変更を push することができる．前述の “hg pull” コマンドのように，一時的なリポジトリを作ってそこに変更を push してみよう．

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

他のリポジトリに push される変更は “hg outgoing” コマンドで知ることができる．

```
$ cd my-hello
$ hg outgoing ../hello-push
comparing with ../hello-push
searching for changes
changeset:   5:090dbfe691cf
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:26 2009 +0000
summary:     Added an extra line of output

```

"hg push" コマンドで実際に push を行う.

```
$ 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" コマンドと同様に，"hg push" コマンドは push 先のリポジトリのワーキングディレクトリの更新を行わない．"hg pull" コマンドと違って "hg push" コマンドは，ワーキングディレクトリの更新を行う-u オプションを持たない．この非対称性は意図的なもので，push 先のリポジトリはリモートサーバ上にあるかもしれず，複数のユーザから共有されている可能性がある．もし誰かが作業中のワーキングディレクトリを更新したら，その作業内容は台無しになってしまう．

リポジトリがある変更をすでに持っているとき，同じ変更を pull または push しようとすると何が起きるだろうか？特に驚くようなことは起きない.

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

2.8.4 変更をネットワークを通じて共有する

これより前のいくつかの節で扱ったコマンドは，ローカルリポジトリだけに限定されない．どのコマンドもネットワーク経由でも全く同じに動作する．違いはローカルパスの代わりに URL を渡すだけである.

```
$ hg outgoing http://hg.serpentine.com/tutorial/hello
comparing with http://hg.serpentine.com/tutorial/hello
searching for changes
changeset:   5:090dbfe691cf
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:26 2009 +0000
summary:     Added an extra line of output

```

この例では，リポートリポジトリに push しようとするが，当然のことながら匿名のユーザに push を許可しないように設定されているため，push できない.

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

Chapter 3

Mercurialツアー: マージ

我々は既にリポジトリのクローン，リポジトリに対する変更，1 つのリポジトリからの pull と別の 1 つのリポジトリに対する push を行った．次のステップは別のリポジトリから変更を**マージ**することである．

3.1 複数の作業結果をマージする

マージは分散リビジョンコントロールツールでの作業において不可欠の部分である．

- アリスとボブは共同作業しているプロジェクトのリポジトリコピーを各々持っている．アリスは自分のリポジトリでバグを修正し，ボブは彼のリポジトリで新機能を追加した．彼らはバグ修正と新機能の両方を持つリポジトリを共有したいと考えている．
- 私は一つのプロジェクトの別々のタスクに対して同時に作業を行うことをよく行っている．このような作業スタイルでは，自分の作業の一つから別の作業の一つへ結果をマージしたいと思うことがしばしばある．

上で述べたようにマージを行いたい状況はとても多いため， Mercurial ではマージを簡単に行えるようになっている．まず別のリポジトリをクローンし，変更を加えるところから始めよう．

```
$ 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
# The file new-hello.c is lightly edited.
$ cp ../new-hello.c hello.c
$ hg commit -m 'A new hello for a new day.'
```

今，内容の異なった 2 つの `hello.c` がある． 2 つのリポジトリの履歴は図 3.1に示すように分かれている．

```
$ 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("once more, hello.\n");
        printf("hello again!\n");
        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;
}
```

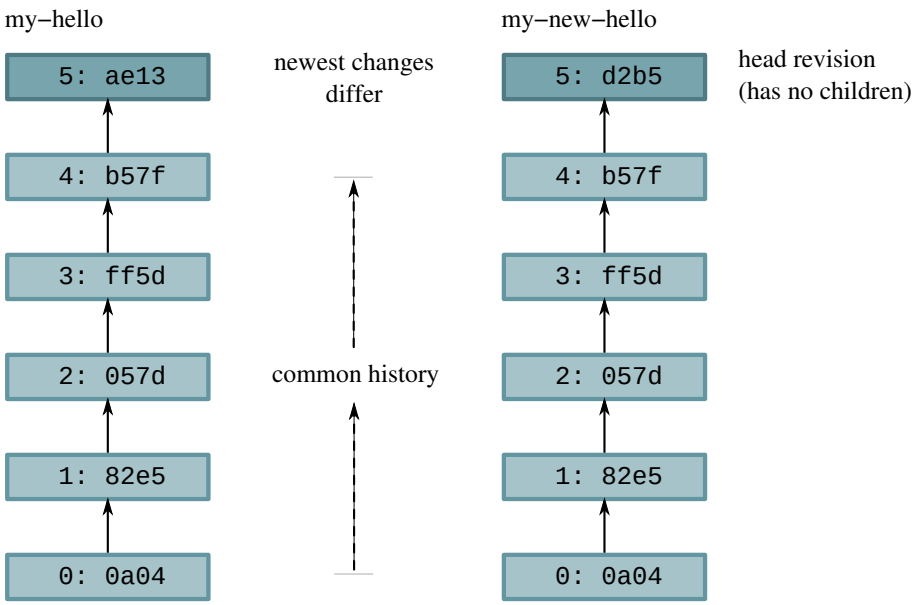

Figure 3.1: my-hello リポジトリと my-new-hello リポジトリの履歴の差異

既に my-hello リポジトリからの pull はワーキングディレクトリに何の影響も与えないことを学んだ.

```
$ 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)
```

しかし “hg pull” コマンドは “heads” についてメッセージを表示する．

3.1.1 Head チェンジセット

head は子やその子孫を持たない変更である．リポジトリの最新のリビジョンは子を持たないため，tip リビジョンはすなわち head である．一方でリポジトリは複数の head を持ち得る．

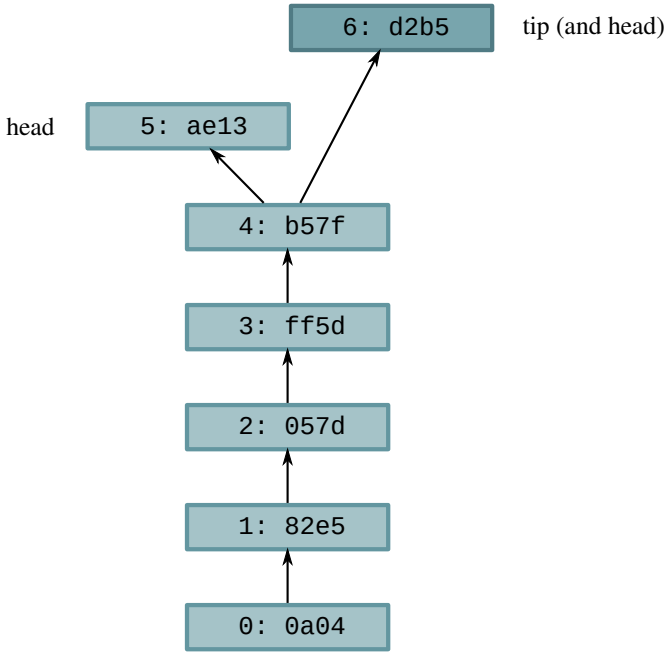

Figure 3.2: my-hello から my-new-hello へ pull したあとのリポジトリの内容

図 3.2に my-hello から my-new-hello へ pull を行った効果を示す． my-new-hello に既に存在した履歴は変更されず，新たにリビジョンが追加される．figure 3.1を見ると，新しいリポジトリにも *changeset ID* が残っていること，**リビジョン番号**が変更されていることが分かる． （これは同時にチェンジセットについて論じる際にリビジョン番号を用いるのが安全でないことの例になっている.）リポジトリ内にある head は “hg heads” によって見ることができる．

```
$ hg heads
changeset:   6:090dbfe691cf
tag:         tip
parent:      4:2278160e78d4
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:26 2009 +0000
summary:     Added an extra line of output

changeset:   5:3e77b31589b3
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:32 2009 +0000
summary:     A new hello for a new day.

```

3.1.2 マージを実行する

新しい tip へ更新するために “hg update” を使った場合何が起きるか？

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

Mercurial は “hg update” コマンドがマージを行わないとメッセージを表示する．“hg update” コマンドは，マージが必要と考えられる場合は，ユーザが（オプションによって）強制しない限りワーキングディレクトリを更新しない．一方，“hg merge” コマンドは 2 つのヘッドのマージを行う．

```
$ hg merge
merging hello.c
merge: warning: conflicts during merge
merging hello.c 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
```

この操作によって “hg parents” と hello.c の出力の双方を反映する**双方の**head からの変更を含むようにワーキングディレクトリが更新される．

3.1.3 マージ結果をコミットする

マージを行うと，マージの結果を “hg commit” するまで，“hg parents” は親 2 つを表示する．

```
$ hg commit -m 'Merged changes'
abort: unresolved merge conflicts (see hg resolve)
```

新しい tip リビジョンは以前のヘッド**両方**を親として持つ．これらは “hg parents” コマンドで表示したのと同じリビジョンである．

```
$ hg tip
changeset:   6:090dbfe691cf
tag:         tip
parent:      4:2278160e78d4
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:26 2009 +0000
summary:     Added an extra line of output

```

図 3.3で，マージの間にワーキングディレクトリに何が起き，コミットした時にリポジトリにどう影響するのかを見ることができる．マージの間，ワーキングディレクトリは 2 つの親チェンジセットを持ち，これらは新しいチェンジセットの両親となる．

マージにより**サイド**ができることがある．左のサイドは “hg parents” の出力の最初の親で，右のサイドが 2 番目の親である．例えば，マージ前にワーキングディレクトリがリビジョン 5 であったとすると，マージの左のサイドがこのリビジョンになる．

3.2 コンフリクトのある変更をマージする

大半のマージはシンプルなものだが，同じファイルの同じ箇所を変更するチェンジセットをマージしなければならないこともある．各々の変更が同じ内容でない限り，結果は *conflict* となるため，変更を整合させるように修正する必要がある．

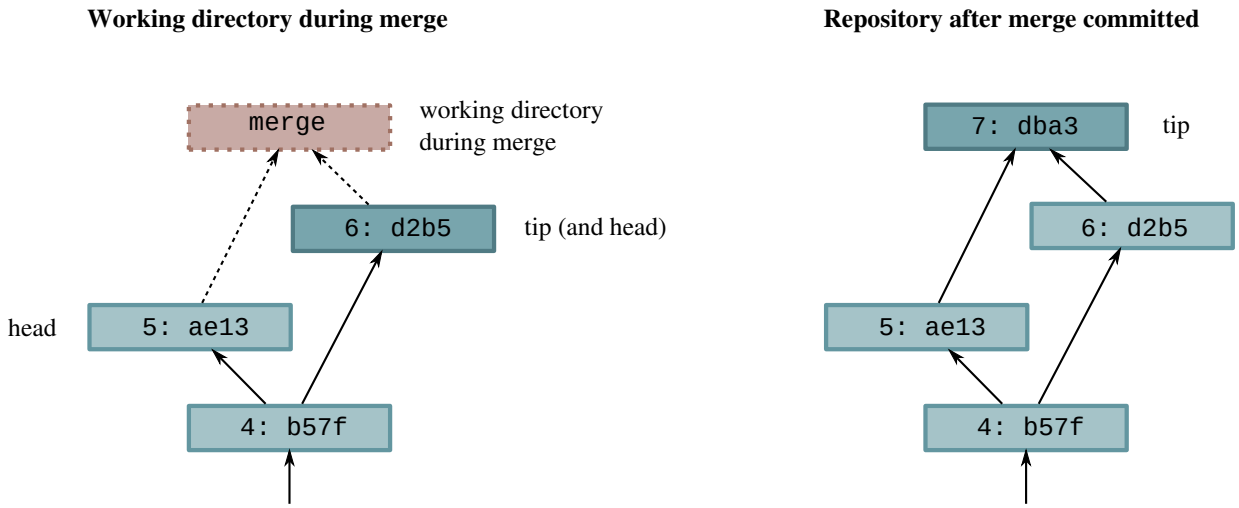

Figure 3.3: マージ中のワーキングディレクトリとリポジトリおよび後続のコミット

1 つのドキュメントに対して 2 つのチェンジセットがコンフリクトした様子を図 3.4に示す．

ここでは，ある 1 つのバージョンのファイルにいくつかの変更を加え，同時に別の人が別の変更を同じテキストに対して加えている．チェンジセットのコンフリクトを解決するため，ファイルがどうあるべきかを明らかにしよう．

Mercurial は内蔵のコンフリクト解決機構を持たない．その代わり，何らかのグラフィカルなコンフリクト解決インタフェースを表示する外部コマンドを起動する．デフォルトで Mercurial はシステムにインストールされている可能性の高いいくつかのマージツールを捜す．Mercurial は最初に 2,3 の全自動のマージツールを試す．これでうまくいかない場合（解決プロセスに人のガイドが必要な場合）やツールが存在しない場合，スクリプトは別の 2,3 のグラフィカルマージツールを試す．

環境変数 `HGMERGE` に好みのプログラム名をを設定することで，Mercurial が `hgmerge` 以外のスクリプトを起動するようにすることも可能である．

3.2.1 グラフィカルマージツールの使用

筆者の好みのグラフィカルマージツールは `kdiff3` なので，これを使ってグラフィカルファイルマージツールの機能を説明する．図 3.5に `kdiff3` の実際の動作のスクリーンショットを示す．今注目しているファイルには 3 つの異なったバージョンがあるため，ここで実行されているのは *3way* **マージ**である．ツールはウィンドウの上部を 3 つのペーンに分割している．

- 左にあるのはファイルの**ベース**バージョンである．これはこれからマージしようとする 2 つのバージョンの親となる最も新しいバージョンである．
- 中央にあるのは我々が変更を加えたバージョンである．
- 右にあるのは他の人が変更を加えたバージョンで，これからマージしようとするものである．

この下のペーンにはマージの**結果**がある．赤字の行はコンフリクトを示しており，我々のバージョンと他の人のバージョンを注意深くマージした結果を用いてこれらをすべて置き換える．

4 つのペーンすべては**同期**しており、どれかを垂直または水平にスクロールすると、他の 3 つも対応する箇所を表示するように更新される。

コンフリクトを解決するために，ファイル中のコンフリクトしているすべての部分に対してベースバージョン，我々のバージョン，別の人々のバージョンのテキストを組み合わせて用いる．さらなる修正が必要な場合は，マージされたファイルを手動で編集することもできる．

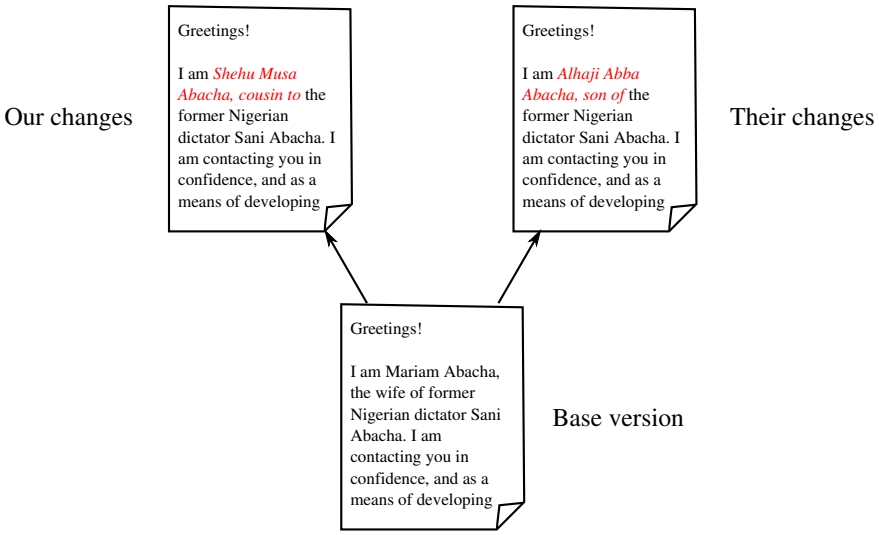

Figure 3.4: コンフリクトしたチェンジセット

数々のマージツールが利用可能であり，それらをカバーすることはできない．それらは，利用可能なプラットフォームが異なり，それぞれ長所と弱点が異なる．ほとんどがプレーンテキストをマージするように作られているが，特別なファイルフォーマット（XML）に特化しているものもある．

3.2.2 実行例

この例では前述の図 3.4での変更履歴を再現する．文書のベースバージョンを含むリポジトリを作成することから始める．

```
$ cat > letter.txt <<EOF
> Greetings!
> I am Mariam Abacha, the wife of former
> Nigerian dictator Sani Abacha.
> EOF
$ hg add letter.txt
$ hg commit -m '419 scam, first draft'
```

リポジトリをコピーし，ファイルに変更を行う．

```
$ 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'
```

別の人が変更を行うのをシミュレートするためにもう一つのクローンを作成する.(別々のリポジトリに隔離して行った作業の結果をマージし，コンフリクトを解消することは少しも珍しいことではないという事がわかるだろう．)

```
$ 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'
```

ファイルに 2 つの別のバージョンを作り，マージを行うのに的下環境を設定する.

```
$ 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)
```

この例では HGMERGE を設定して，Mercurial に非対話的な merge コマンドを起動させる．これは多くの Unix 系システムにバンドルされているコマンドである.(この例を手元のマシンで実行する時には HGMERGE を設定する必要はない.)

Mercurial は merge コマンドで正しくマージできなかった場合，もう一度マージを行うためにどのようにすればいいのかを表示する．これは，グラフィカルマージツールの使用中に混乱を来したか間違いを犯したことに気付いたために終了した場合などに有効であろう.

自動または手動によるマージが失敗した場合，影響を受けるファイルを “修正” してマージの結果をコミットしなければならない.

```
$ cat > letter.txt <<EOF
> Greetings!
> I am Bryan O'Sullivan, no relation of the former
> Nigerian dictator Sani Abacha.
> EOF
$ hg resolve -m letter.txt
$ hg commit -m 'Send me your money'
$ hg tip
changeset:   3:3a540632fa34
tag:         tip
parent:      1:bf66bd12813c
```

```
parent:      2:7f1207026dcf
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:36 2009 +0000
summary:     Send me your money

```

3.3 pull-merge-commit 手順を簡単にする

前節で概要を述べたチェンジセットのマージプロセスは単純なものだったが，3 つのコマンドを順に用いる必要があった．

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

最後のコミットでは，ほとんど退屈なボイラープレートテキストと言ってもよいコミットメッセージの入力も必要であった．

可能であれば必要なステップ数を少なくすることができるとよい．実は Mercurial は `fetch` という，まさにこのことを行うエクステンションを同梱している．

Mercurial はコアを小さく，扱いやすく保ったまま機能を拡張できる柔軟なエクステンションメカニズムを提供している．コマンドラインから利用できる新しいコマンドを追加するようなエクステンションもあれば，目に見えないところで例えばサーバに機能を追加するようなものもある．

`fetch` エクステンションはその名の通りの新しいコマンド “`hg fetch`” を追加する．このエクステンションは `-u` と “`hg merge`” および “`hg commit`” を組み合わせた働きをする．このエクステンションはまず他のリポジトリから変更を pull し，リポジトリに新しいヘッドが追加された場合はマージを行い，(マージが成功した場合) マージ結果を自動的に生成されたコミットメッセージと共にコミットする．新たなヘッドが追加されなかった場合は新たな tip チェンジセットへワーキングディレクトリを更新する．

`fetch` エクステンションは簡単に有効にすることができる．ホームディレクトリの `.hgrc` ファイルを編集し，`[extensions]` セクションに “`fetch `” という行を追加すればよい．

```
[extensions]
fetch =
```

(通常，右辺の “=” はエクステンションの置かれた場所を表すが，`fetch` エクステンションは標準配布物に含まれるため，Mercurial はその所在をすでに知っている．)

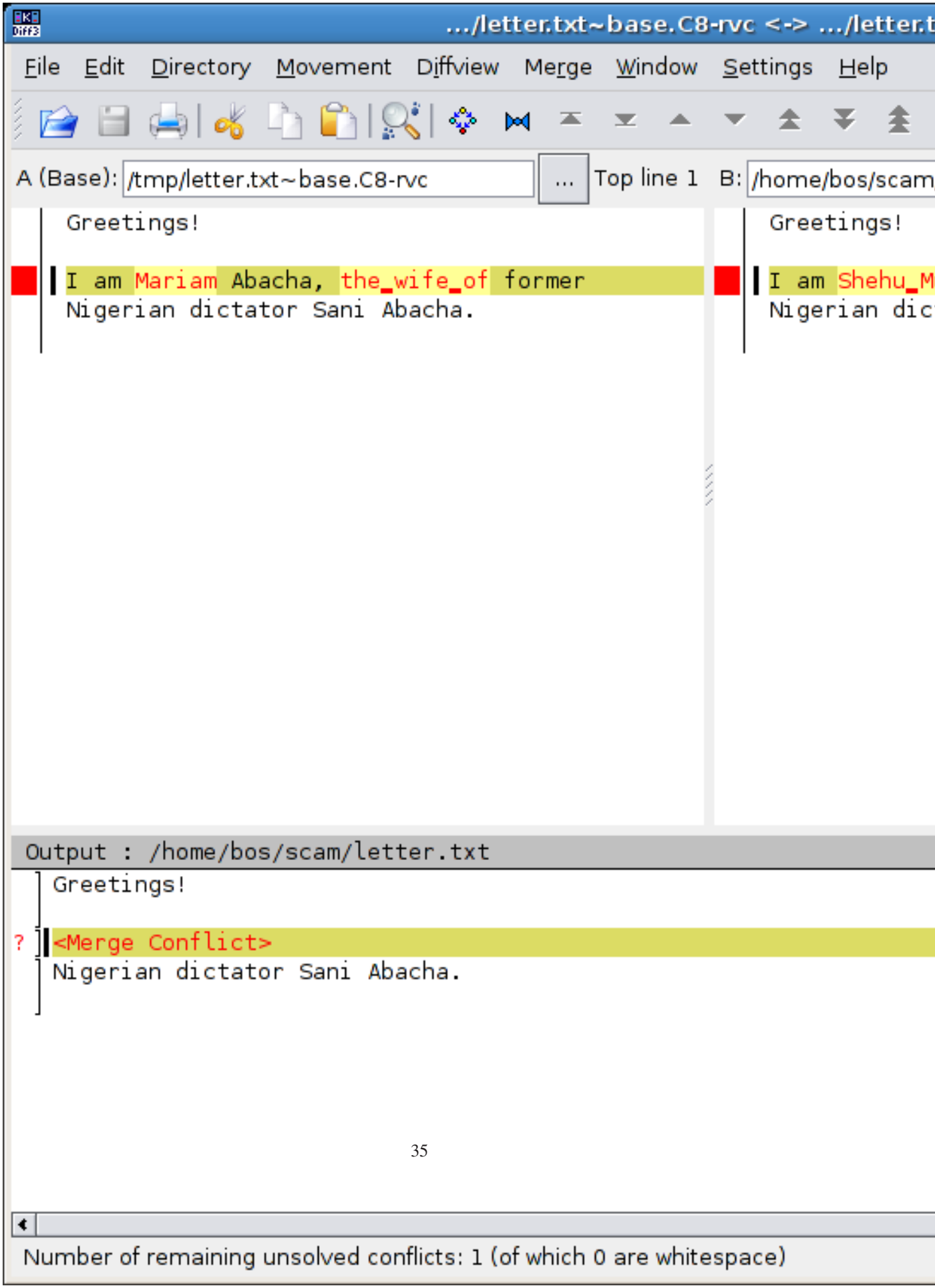

Figure 3.5: ファイルの複数リビジョンを kdiff3 を使ってマージする

Chapter 4

舞台裏

多くのリビジョンコントロールシステムと違って， Mercurial の動作の基本となっている概念を理解することは容易い．これらの詳細を理解することは必ずしも必要ではなく，この章を飛ばしても差し支えない．しかし筆者は，ソフトウェアをよりよく使う上で何が起きているのかについてモデルを意識していることは有用であると考えている．

舞台裏で何が起こっているのか理解できると，筆者は Mercurial が**安全**と**効率**を実現するように注意深く設計されていると確信することができた．また，重要な点として，リビジョンコントロールの操作を行う際にソフトウェアが何をするのかを記憶に留めておくことによって，不意の挙動で驚くことが少なくなった．

この章ではまず Mercurial の設計のコアコンセプトをカバーする．そして実装上のいくつかの興味深い点の詳細について議論する．

4.1 Mercurial の履歴記録

4.1.1 ファイル履歴の追跡

Mercurial はファイルへの変更を追跡する時，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に格納する．ファイルログ内の各々のエントリは，追跡対象のファイルのリビジョンを再建するのに十分な情報を持つ．ファイルログは `.hg/store/data` ディレクトリにファイルとして保存される．ファイルログはリビジョンデータと Mercurial がリビジョンを効率的に見つけられるようにするためのインデックスの 2 種類の情報を持つ．

サイズの大きなファイルや，膨大な履歴を持つファイルは，データが (“`.d`” suffix) およびインデックス (“`.i`” suffix) のファイルに分割された filelog を持つ．サイズが小さく，履歴の大きくないファイルはリビジョンデータとインデックスが 1 つの “`.i`” ファイルに結合されている．ワーキングディレクトリ内のファイルとリポジトリ内の履歴を追跡する filelog との対応を図 4.1に示す.

4.1.2 追跡されているファイルの管理

Mercurial は**マニフェスト**と呼ばれる構造を用いて，追跡すべきファイルの情報を集めている．マニフェスト内の各々のエントリは，単一のチェンジセット内に存在するファイルの情報を持っている．エントリはどのファイルがチェンジセットに存在しているか，それらのリビジョンが何であるのかという情報と，いくつかの他のファイルメタデータを記録している．

4.1.3 チェンジセット情報の記録

changelog は各々のチェンジセットの情報を持つ．各々のリビジョン記録は，誰が変更をコミットしたのか，チェンジセットのコメント，変更に関連した他の情報，使用されるマニフェストのリビジョンを持つ.

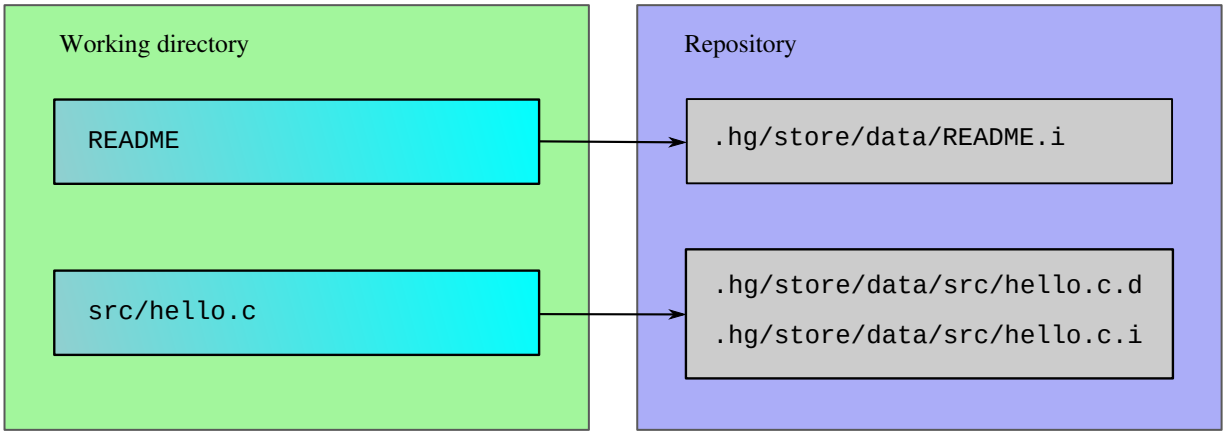

Figure 4.1: ワーキングディレクトリ内のファイルとリポジトリのファイルログの関係

4.1.4 リビジョン間の関係

チェンジログ，マニフェスト，ファイルログ内で，各々のリビジョンは直接の親（あるいはマージの場合はその両親）へのポインタを持つ．すでに述べたように，この構造に現れるリビジョンの間には関係があり，本質的に階層的である．

リポジトリにあるすべてのチェンジセットは，チェンジログ内で正確に 1 つのリビジョンを持つ．チェンジログの各々のリビジョンは，マニフェストの単一のバージョンへのポインタを持つ．マニフェストの各々のリビジョンは，チェンジセットが生成された時のファイルログの各々の単一リビジョンへのポインタを持つ．この関係を図 4.2に示す．

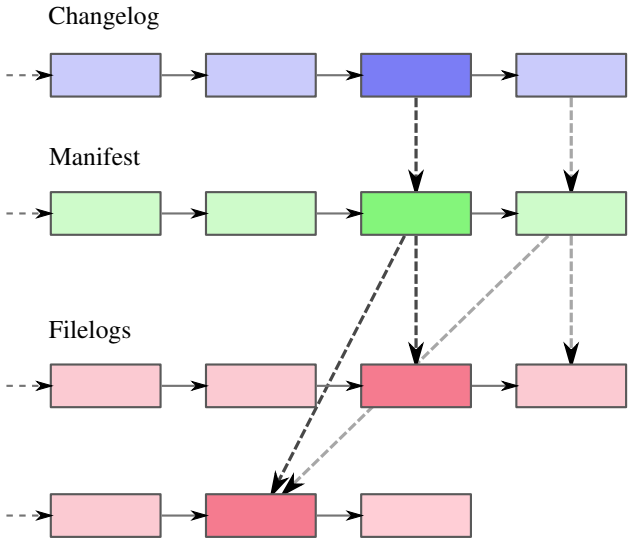

Figure 4.2: メタデータの関係

イラストで示されているように，リビジョンとチェンジログ，マニフェストまたはファイルログの間には 1 対 1 の関係はない．マニフェストが 2 つのチェンジセットの間で変化しなかった場合は，チェンジログ内でこれらのチェンジセットを表すエントリは同じバージョンのマニフェストを示す．Mercurial が追跡するファイルが，2 つのチェンジセット間で変化しなかった場合は，2 つのマニフェストのリビジョンで，そのファイルを示すエントリはファイルログの同じリビジョンを示す．

4.2 安全かつ効率的なストレージ

チェンジログ，マニフェストおよびファイルログの土台に使われているは共通の *revlog* という構造体である．

4.2.1 効率的なストレージ

revlog は *delta* 機構を使ってリビジョンの効率的な記憶を提供する．ファイルの各々のバージョンの完全なコピーを保存するのではなく，古いリビジョンを新しいバージョンへ変換するのに必要な変更を保存する．多くのファイルデータに対して， delta は典型的にはファイルのフルコピーの 1 パーセント未満であある．

古いリビジョンコントロールシステムのいくつかはテキストファイルの delta に対してしか機能しない．それらのシステムではバイナリファイルは完全なスナップショットか，テキスト表現にエンコードされた形式である必要がある．これらは共に無駄の多いアプローチである． Mercurial は任意のバイナリファイルについて，delta を効率的に扱うことができ，テキストを特別扱いする必要がない．

4.2.2 安全な動作

Mercurial は revlog ファイルの末尾にデータの**追加**のみを行う．一度書き込まれた部分は後になって変更されることはない．これはデータの変更や再書き込みを行う方法よりも頑強かつ効率的である．

加えて，Mercurial はあらゆる書き込みを複数のファイルへの**トランザクション**の一部と見なす．トランザクション全体が成功し，読み出し側に一度に結果が見える場合も，そうでない場合もトランザクションは**アトミック**である．このアトミック性の保証は， 2 つの Mercurial を，片方はデータの読み出し，もう一方は書き込みで実行している場合，読み出し側には混乱の原因となる部分的な書き込み結果は見えないことを意味する．

Mercurial はファイルに追記のみをすることで，トランザクションの保証を容易にしている．物事を単純化することによって，処理の正しさを確実にするようにしている．

4.2.3 高速な取得

Mercurial はこれまでのリビジョンコントロールシステムに共通した落とし穴を賢明に避けている．問題だったのは**非効率的な取得**であった．大抵のリビジョンコントロールシステムはリビジョンの内容をスナップショットへの一連の変更の増分として保存している．特定のリビジョンを再現するためにはまずスナップショットを読み込み，しかる後に目的のリビジョンを読む必要があった．ファイルへの履歴が増えれば増えるほど読み込まなければならないリビジョンが多くなり，特定のリビジョンを再現するのに時間がかかるようになる．

Mercurial ではこの問題を単純だが効果的な方法で解決している．前回にスナップショットを作成した時点からの累積的な増分が一定の閾値を越えると，新たな増分ではなく，新たなスナップショットが保存される（このスナップショットは勿論圧縮されている）．これによって**いかなる**リビジョンも迅速に再現される．このアプローチは以後他のリビジョンコントロールシステムにコピーされるほどうまく機能している．

図 4.3に概念を示す．Mercurial は revlog のインデックスファイルのエントリに特定のリビジョンを再現するのに必要なデータファイル内のある範囲のエントリを保存する．

その他: ビデオ圧縮の影響

ビデオ圧縮に慣れていたり，デジタルによるケーブルまたは衛星放送に慣れているのなら，大部分のビデオ圧縮の仕組みでは，ビデオの各フレームが，前のフレームとの差分として保存されていることを知っているだろう．加えて，これらの仕組みは圧縮率を稼ぐために不可逆な圧縮技術を使っており，映像のエラーはフレーム間の差分が増えるに従って蓄積していく．

ビデオストリームでは，信号の不具合によって時折ドロップアウトが出ることがあり，また不可逆圧縮による影響の蓄積を抑えるため，ビデオエンコーダは定期的に（キーフレームと呼ばれる）完全なフレームをストリームに挿入する．次の差分はこのフレームに対して取られる．これによって，ビデオ信号が途絶えても次のキーフレームを受信すれば正常に戻ることができる．またエンコードエラーの蓄積はキーフレームごとに除去される．

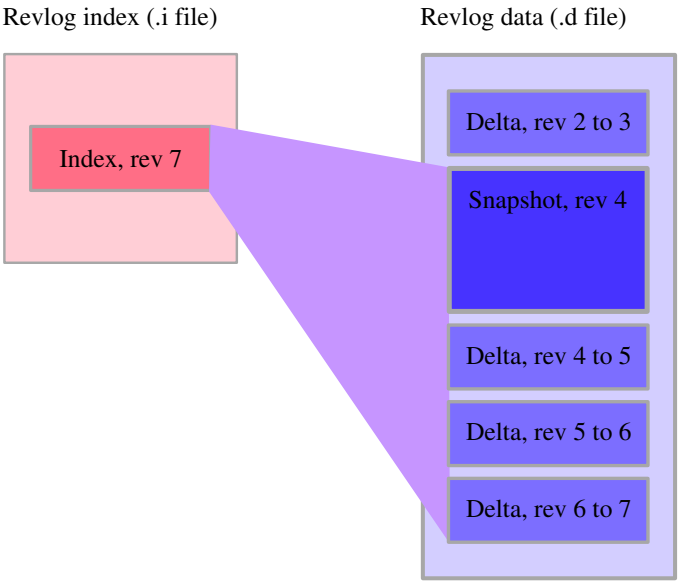

Figure 4.3: 差分を用いた revlog のスナップショット

4.2.4 識別と強い一貫性

差分やスナップショット情報と共に，revlog エントリはデータの暗号ハッシュを持つ．ハッシュにより，リビジョンの内容を偽ることが困難になり，また事故によって内容が破損した場合，発見が容易になる．

ハッシュは単に破損をチェックする以上のことを行う．これらはリビジョンの識別にも用いられる．チェンジセットの識別ハッシュは，エンドユーザからはチェンジログでユーザ名と共に現れる数字として目にすることが多いだろう．ファイルログとマニフェストでもハッシュは使われているが，Mercurial はこれらを背後でのみ用いる．

Mercurial はファイルのリビジョン取得時と変更を別のリポジトリから pull する時にハッシュが正しいことを確認する．一貫性に問題があると，エラーメッセージを出力し，動作を停止する．

取得の効率に加えて，Mercurial は周期的にスナップショットを使うことで，部分的なデータの破損に対して頑強になっている．ハードウェアの問題やシステムのバグで revlog が部分的に破損した場合でも，多くの場合，revlog の破損していない部分から破損した部分の前後でいくつかのリビジョンや大半のリビジョンを再建することができる．差分のみを記録するシステムではこれを実現することはできない．

4.3 リビジョン履歴，ブランチ，マージ

Mercurial revlog のすべてのエントリは，通常 *parent* として参照される直接の祖先のリビジョンを知っている．実際には，リビジョンは 1 つだけでなく 2 つの親を記録することができる．Mercurial は “null ID” という特別なハッシュを使って “ここには親になるリビジョンがない” ということを表現する．このハッシュは単にゼロを並べた文字列である．

revlog の概念的な構造の一例を図 4.4に示す．ファイルログ，マニフェストおよびチェンジログはすべて同じ構造を持つ．これらの違いは，各々の差分やスナップショットに格納するデータの違いだけである．

revlog での最初のリビジョン（図の一番下）は null ID を両親を示すスロットの両方に持ち，このリビジョンが実際にはただ一つの親しか持たないことを表している．同じ親の ID を持つ任意の 2 つのリビジョンはブランチである．ブランチ間のマージを表すリビジョンは，2 つのノーマルリビジョン ID を両親のスロットに持つ．

4.4 ワーキングディレクトリ

Mercurial はリポジトリのある特定のチェンジセットのファイルスナップショットをワーキングディレクトリ内に持つ．

ワーキングディレクトリがどのチェンジセットに更新されているがあるのかは既知である．ワーキングディレクトリが特定のチェンジセットになるように更新する際， Mercurial は適切なリビジョンのマニフェストを検索し，そのチェンジセットがコミットされた時点でどのファイルが追跡されているかを調べ，各々のファイルのリビジョンを特定する．そしてチェンジセットがコミットされた時点の内容をもつファイルのコピーを作成する．

dirstate にはワーキングディレクトリについて Mercurial が把握している情報が格納されている．その内容はワーキングディレクトリがアップデートされているチェンジセットおよびワーキングディレクトリ内で Mercurial が追跡している全てのファイルについての詳細である．

revlog におけるリビジョンが 2 つの親の情報を格納する領域を持ち，通常のリビジョン（親は 1 つ）または 2 つの親のマージを表現できるのと同様に dirstate も 2 つの親のためのスロットを持っている．“`hg update`” コマンドを実行すると，アップデート先のチェンジセットが 1 つ目の親のスロットに記録され，null ID が 2 つ目の親のスロットに記録される．他のチェンジセットと “`hg merge`” を行うと，最初の親はそのままに，2 番目の親はマージするチェンジセットとなる．“`hg parents`” コマンドで dirstate の両親を知ることができる．

4.4.1 コミット時に何が起きるのか

dirstate は管理目的以外の情報も保存している． Mercurial は，コミットの際に dirstate の両親を**新たなチェンジセットの両親**として用いる．

図 4.5にワーキングディレクトリの通常状態を示す．リポジトリの最新のチェンジセットは *tip* で，子を一切持たない．

ワーキングディレクトリが “コミットしようとしているチェンジセット” であると見なすことは役に立つ．追加，削除，リネームまたはコピーしたファイルを Mercurial に認識させると，すでに Mercurial が追跡している任意のファイルへの変更と同様，すべてチェンジセットに反映される．新たなチェンジセットはワーキングディレクトリの両親を両親として持つ．

コミット後，Mercurial はワーキングディレクトリの両親を更新し，1 つ目の親が新たなチェンジセットの ID，2 番目を null ID にする．これを図 4.6に示す．Mercurial はコミット時にワーキングディレクトリ内のファイルには一切触れず，dirstate に新たな両親を記録する．

4.4.2 新たなヘッドを作る

ワーキングディレクトリを現在の tip 以外のチェンジセットに更新することはいささかもおかしなことではない．例えば，この前の火曜日にプロジェクトがどのような状態であったか知りたいと思うかもしれないし，チェンジセットのうちのどれがバグを混入させたか突き止めたいと考えることがあるかもしれない．このような場合，ワーキングディレクトリを興味のあるチェンジセットに更新し，チェンジセットをコミットした時にそれらがどのようであったかを見るためにワーキングディレクトリ内のファイルを調べるのが自然である．この影響は図 4.7に示す．

ワーキングディレクトリを古いチェンジセットに更新し，変更を行ってコミットすると何が起こるだろうか？ Mercurial は前述のように振舞う．ワーキングディレクトリの両親は新たなチェンジセットの両親となる．新たなチェンジセットは子を持たず，従って新たな tip となる．リポジトリには *heads* と呼ばれる 2 つのチェンジセットができる．この時の構造を図 4.8に示す．

Note: Mercurial を使い始めたばかりであれば，よくある “エラー” を覚えておくとよい．それは “`hg pull`” コマンドをオプションなしで実行することである．デフォルトでは “`hg pull`” はワーキングディレクトリの更新を**行わない**．そのため，リポジトリには新しいチェンジセットが到着しているのにワーキングディレクトリは前回 pull したチェンジセットのままである．ここで何らかの変更を行ってコミットしようとすると，ワーキングディレクトリが現在の tip に同期していないため，新たなヘッドを作ることになってしまう．
“エラー” という言葉を引用符で括ったのは，この状態は “`hg merge`” と “`hg commit`” だけで解消できるからだ．言い替えると，この状態はほとんどの場合害をなすものではなく，単に初心者を驚かす程度のものである．この振舞を避ける別の方法や，なぜ Mercurial がこのように驚かせるような方法で動作するのかについては後ほど議論する．

4.4.3 変更のマージ

“`hg merge`” コマンドを実行した時，Mercurial はワーキングディレクトリの最初の親を変更せず残し，2 番目の親を図 4.9のようにマージ先のチェンジセットとする．

Mercurial は 2 つのチェンジセット内で管理されているファイルをマージするためにワーキングディレクトリを変更する必要がある．少し単純化すると，マージプロセスは双方のチェンジセットのマニフェスト内にある全てのファイルに対して次のように行われる．

- どちらのチェンジセットでも変更されていないファイルに対しては何も行わない．
- 変更されたファイルが片方のチェンジセットに含まれ，もう一方には含まれない場合，ワーキングディレクトリに変更されたコピーが作成される．
- 一方のチェンジセットで削除されたファイルがあり，もう一方がそのファイルを含まないか，同様に削除されている場合はワーキングディレクトリからそのファイルを削除する．
- 一方のチェンジセットでファイルが削除されており，もう一方ではそのファイルが変更されている場合は，変更されたファイルを維持するかファイルを消去するかかユーザに尋ねる．
- 両方のチェンジセットでファイルが変更されている場合，マージ後のファイルの内容を選択するために外部のマージプログラムを起動する．これを行うためにはユーザの入力が必要である．
- 一方のチェンジセットでファイルが変更されており，もう一方ではファイルがリネームまたはコピーされている場合，変更は新しい名前のファイルに取り込まれる．

詳しく述べればマージにはいろいろな特殊例がある．しかしここでは最も普通の選択について説明する．これまで見てきたように，大半のケースは完全に自動であり，実際に大半のマージはコンフリクトを解決するために入力を求めることなく自動的に完了する．

マージ後にコミットを行うとき何が起こるかを考える場合はやはりワーキングディレクトリを “これからコミットしようとするチェンジセット” と考えるとよい．“`hg merge`” コマンドが完了した後，ワーキングディレクトリは 2 つの親を持ち，これらは新たなチェンジセットの両親となる．

Mercurial は複数回のマージを促す．ここでそれぞれのマージの結果を自分自身でコミットしなければならない．これは Mercurial がリビジョンとワーキングディレクトリの双方について 2 つの親のみを追跡することによる．複数のチェンジセットを一度にマージすることは技術的には可能だが，マージによる混乱を引き起こし，収拾がつかなくなる見込みが大きい．

4.4.4 マージとリネーム

驚くほど多くのリビジョンコントロールシステムが**ファイル名**の変化に注意を払っていない．例えばマージの際，一方でファイルがリネームされていた場合，もう一方の変更は何の警告も無しに破棄されてしまう．

Mercurial はリネームやコピーを行う時にメタデータを記録し，マージを正しく行うために利用する．例えばあるユーザがファイルをリネームし，別のユーザがリネームせずに同じファイルを編集したとすると，マージの際にファイルはリネームされ，なおかつ編集内容も取り込む．

4.5 設計の他の興味深い点

前節で Mercurial の設計の最も重要な面について取り上げ，信頼性と性能に細心の注意を払っていることを強調した．しかし細部への注意はそれだけに留まらない．Mercurial の構造には，個人的に興味深く感じた点が多々ある．巧妙に設計されたシステムの背後にあるアイディアについて読者が興味を持つならば，より深い理解が出来るよう，前述した大きな括りとは別にその 2,3 について取り上げてみようと思う．

4.5.1 賢い圧縮

適切な場合，Mercurial はスナップショットと差分を圧縮された形式で保存する．Mercurial は常にスナップショットや差分の圧縮を**試みる**が，それを保存するのは圧縮されたバージョンが元のバージョンよりも小さい時のみである．

つまり，Mercurial は `zip` アーカイブや JPEG 画像などのように元々圧縮されているファイルの保存を “正しいやり方” で行う．これらのファイルでは，差分取って圧縮を行っても結果として得られるファイルは通常，元のファイルよりも大きくなる．そこで Mercurial は `zip` や JPEG をそのまま保存する．

通常，圧縮されたファイルのリビジョン間の差分はファイルのスナップショットより大きい．ここでも Mercurial は “正しいやり方” を用いている．このような状況で，差分のサイズがファイルの完全なスナップショットとして保存した方が有利となる閾値を越えると，差分ではなくスナップショットを保存する．このようにして単純に差分のみを記録するアプローチよりも記憶領域を節約している．

ネットワーク再圧縮

ディスクにリビジョンを保存する際，Mercurial は “deflate” 圧縮アルゴリズムを用いる．（これは人気の高い `zip` アーカイブフォーマットと同じアルゴリズムである．）このアルゴリズムは高い圧縮率と高速性をバランスさせている．しかしリビジョンデータをネットワークで伝送する際には Mercurial は圧縮されているリビジョンデータを伸長する．

HTTP による接続の場合，Mercurial はデータストリーム全体をより圧縮率の高いアルゴリズム（広く用いられている圧縮パッケージである `bzip2` で使われている Burrows-Wheeler アルゴリズム）で再圧縮する．このアルゴリズムと，（リビジョン毎でなく）ストリーム全体を圧縮することにより，送信されるバイト数は大きく低減され，あらゆるネットワークでよい性能を得ることができる．

（`ssh` による接続の場合，Mercurial はストリームの再圧縮は行わない．`ssh` コマンド自体が圧縮を行うためである．）

4.5.2 読み書きの順序とアトミック性

ファイルへの追記は読み手が部分ファイルを読まないことの保証としては十分ではない． 図 4.2を思い起こすと，チェンジログ内のリビジョンはマニフェスト内のリビジョンを指し示しており，マニフェスト内のリビジョンはファイルログ内のリビジョンを指し示していた．この階層構造は重要な意味を持っている．

書き手はファイルログとマニフェストを書き込むことでトランザクションを開始するが，これらが完了するまでチェンジログデータの書き込みは行わない．一方，読み手はチェンジログデータ，マニフェストデータ，ファイルログデータの順に読み出しを行う．

書き手は常にファイルログとマニフェストデータをチェンジログの前に書き込み終了しているため，読み手は部分的に書かれたマニフェストリビジョンへのポインタををチェンジログから読み出すことはない．また部分的に書かれたファイルログリビジョンへのポインタをマニフェストから読み出すこともない．

4.5.3 同時アクセス

読み書きの順序とアトミック性の保証により，Mercurial ではデータの読み出し時にリポジトリの**ロック**が不要になっている．読み出し中に書き込みが発生したとしても同様である．このことはスケーラビリティに大きな効果をもたらす．多数の Mercurial プロセスがあっても，リポジトリへの書き込みの有無に関わらず，リポジトリから一度に安全にデータを読み出すことができる．

Mercurial では，ロック無しで読み出しを行うため，マルチユーザシステム上でリポジトリを共有している場合でも clone や pull を行おうとする他のローカルユーザにリポジトリへの**書き込み**許可を与える必要はない．彼らには**読み出し**許可があればよい．（他のリビジョンコントロールシステムではこうはいかず，大半のシステムでは読み手が安全にリポジトリにアクセスするためにロックを取得することを必要とする．これはすなわち読み手が少なくとも 1 つのディレクトリに対して書き込み許可を持っていなければならないことを意味する．このためにセキュリティや管理上の厄介な問題が発生し得る．）

Mercurial は一度に 1 プロセスだけがリポジトリに書き込むことを保証するためにロックを使っている．（Mercurial の用いるロックメカニズムは NFS のようにロックと相性の悪いことで有名なファイルシステム上でも安全である．）リポジトリがロックされると，書き込みプロセスはリポジトリがアンロックされるまでしばらく待つ．しかしリポジトリが長時間にわたってロックされ続ける場合は，書き込みしようとするプロセスはタイムアウトする．これは，例えば，日常用いる自動スクリプトは永遠にスタックするわけではないことを意味する．（もちろんタイムアウトまでの時間はゼロから無限の間で設定可能である）

安全な dirstate アクセス

リビジョンデータの時と同様に，Mercurial は dirstate ファイルを読み出す際にはロックを行わない．ロックを行うのは書き込みの時のみである．一部のみが書き込まれた dirstate ファイルを読み込まないようにするため，Mercurial は同じディレクトリ内に固有な名前で dirstate ファイルを書き，この一時ファイルをアトミックに `dirstate` にリネームする．これにより，`dirstate` ファイルは常に完全であることが保証される．

4.5.4 シークの回避

ディスクヘッドのシークを避けることは Mercurial の性能にとって極めて重要である．シークは大規模な読み出し操作と比較しても非常に高価である．

その理由は，例えば，dirstate は 1 つのファイルに保存されているためで，Mercurial が追跡している dirstate ファイルがディレクトリ毎にあると，Mercurial はディレクトリ毎にシークを行うことになる．実際には Mercurial は全体で一つの dirstate ファイルをワンステップで読む．

Mercurial はリポジトリをローカルストレージにクローンする際に “コピーオンライト” 手法を使っている．元のリポジトリから新しいリポジトリへ全ての revlog ファイルをコピーするのではなく，ハードリンクを作成する．ハードリンクは同一のファイルを指し示す別名を作る簡単な方法である． Mercurial は revlog ファイルに書き込みを行う際にファイルを示す名前が 2 つ以上ある稼働かをチェックする． 2 つ以上の名前がある場合，2 つ以上のリポジトリがファイルを使用しており，Mercurial はファイルのリポジトリに固有な新しいコピーを作成する．

リビジョンコントロールシステムの開発者の幾人かは，ファイルの完全なプライベートコピーを作るというアイディアは，ストレージの利用量の観点からあまり効率的でないと指摘している．これは事実であるが，ストレージは安価であり，この方法は OS 上で差分を取る場合最高の性能をもたらす．その他の手法は性能を損ない，ソフトウェアが複雑になる可能性が高い．これらは日々の使用感よりもずっと重要である．

4.5.5 dirstate の他の内容

Mercurial は，ファイルを変更した場合でも，変更の申告を強制しないため，dirstate に追加の情報を保存することでファイルを変更したかどうか効果的に判別する．ワーキングディレクトリのすべてのファイルについて，最後にファイルが変更された時刻と，その際のファイルサイズを記録している．

明示的にファイルを “`hg add`”, “`hg remove`”, “`hg rename`” または “`hg copy`” した場合，Mercurial は dirstate を更新し，コミットの際にそれらのファイルをどう取り扱うかを判断する．

Mercurial がワーキングディレクトリのファイルの状態をチェックする際は，まずファイルの更新日時を調べる．これが変更されていない場合はファイルは変更がないということがわかる．ファイルサイズが変わって

いる場合は，ファイルが変更されたことがわかる．更新日時が変わっているが，サイズが同じ場合は Mercurial はファイルの内容を直接見て，変更されているかどうかを判断する必要がある．これらのごく僅かの情報を保存することによって，Mercurial はデータの読み取り量を劇的に削減し，他のリビジョンコントロールシステムと比較して大きな性能向上を達成している．

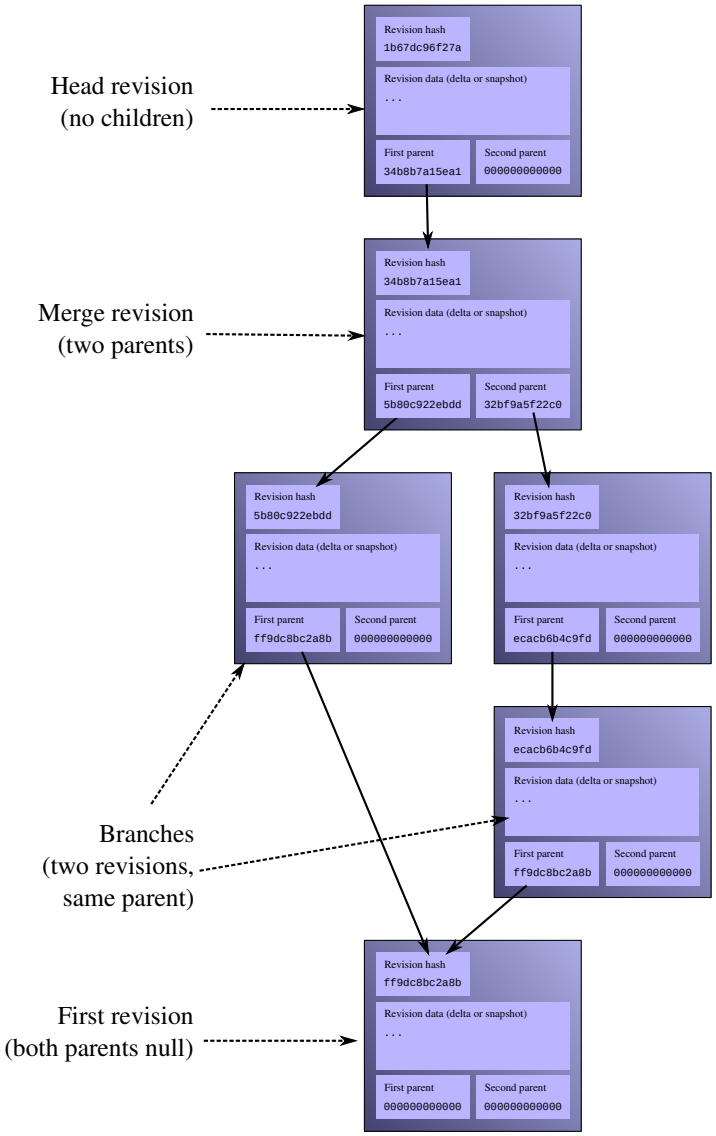
Head revision
(no children)
Merge revision
(two parents)
Branches
(two revisions,
same parent)
First revision
(both parents null)
Revision hash
1b67dc96f27a
Revision data (delta or snapshot)
...
First parent
34b8b7a15ea1
Second parent
000000000000
Revision hash
34b8b7a15ea1
Revision data (delta or snapshot)
...
First parent
5b80c922ebdd
Second parent
32bf9a5f22c0
Revision hash
5b80c922ebdd
Revision data (delta or snapshot)
...
First parent
ff9dc8bc2a8b
Second parent
000000000000
Revision hash
32bf9a5f22c0
Revision data (delta or snapshot)
...
First parent
ecacb6b4c9fd
Second parent
000000000000
Revision hash
ecacb6b4c9fd
Revision data (delta or snapshot)
...
First parent
ff9dc8bc2a8b
Second parent
000000000000
Revision hash
ff9dc8bc2a8b
Revision data (delta or snapshot)
...
First parent
000000000000
Second parent
000000000000

Figure 4.4:

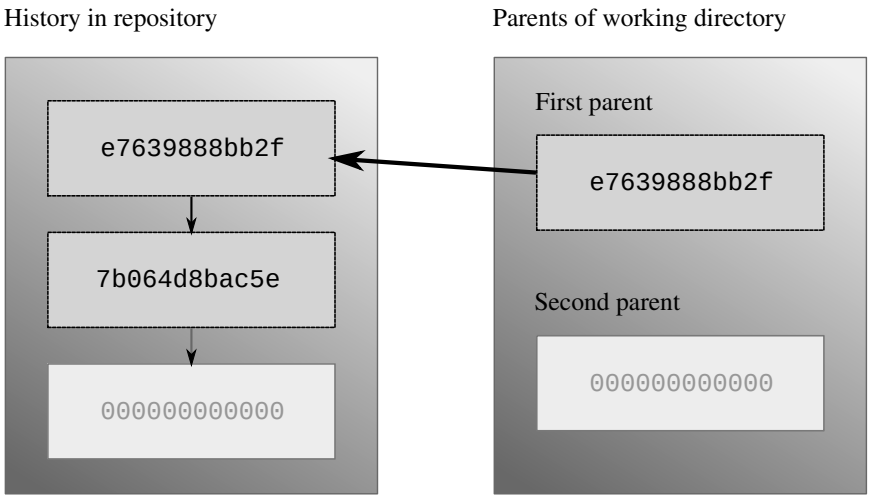

Figure 4.5: ワーキングディレクトリは 2 つの親を持ち得る

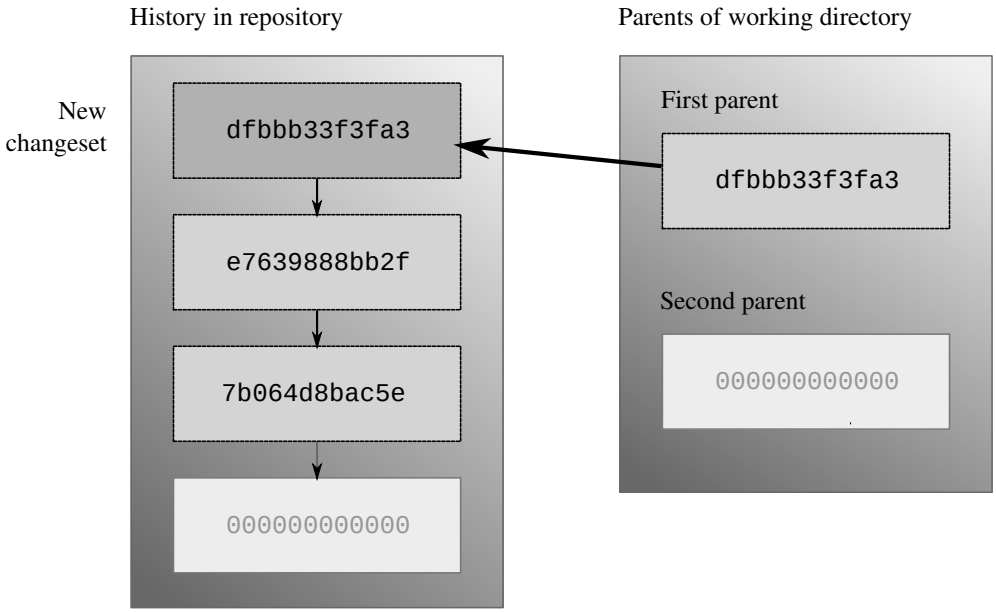

Figure 4.6: コミット後，ワーキングディレクトリは新たな両親を持つ

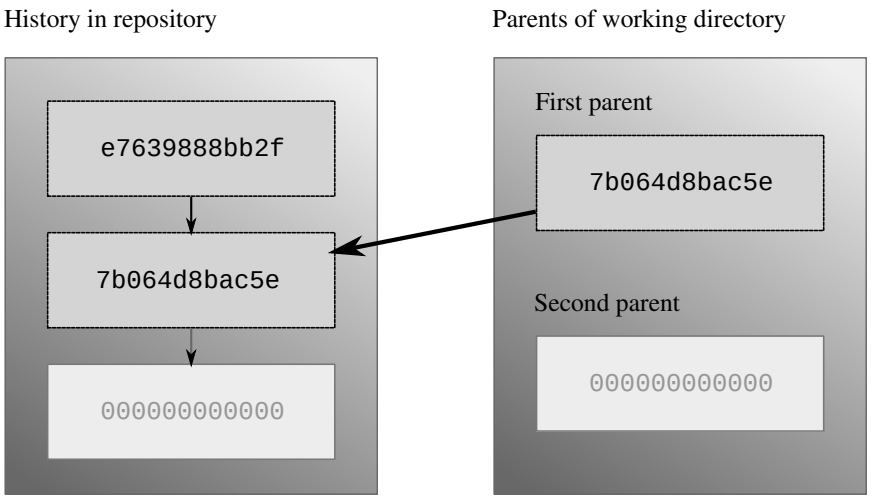

Figure 4.7: 古いチェンジセットへと更新されたワーキングディレクトリ

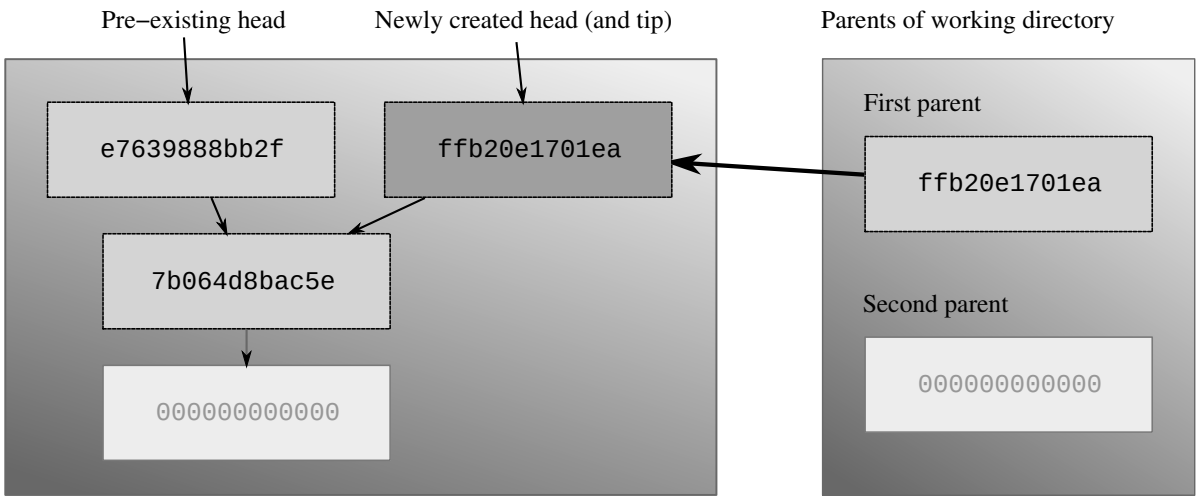

Figure 4.8: 古いチェンジセットに同期中にコミットが行われた場合

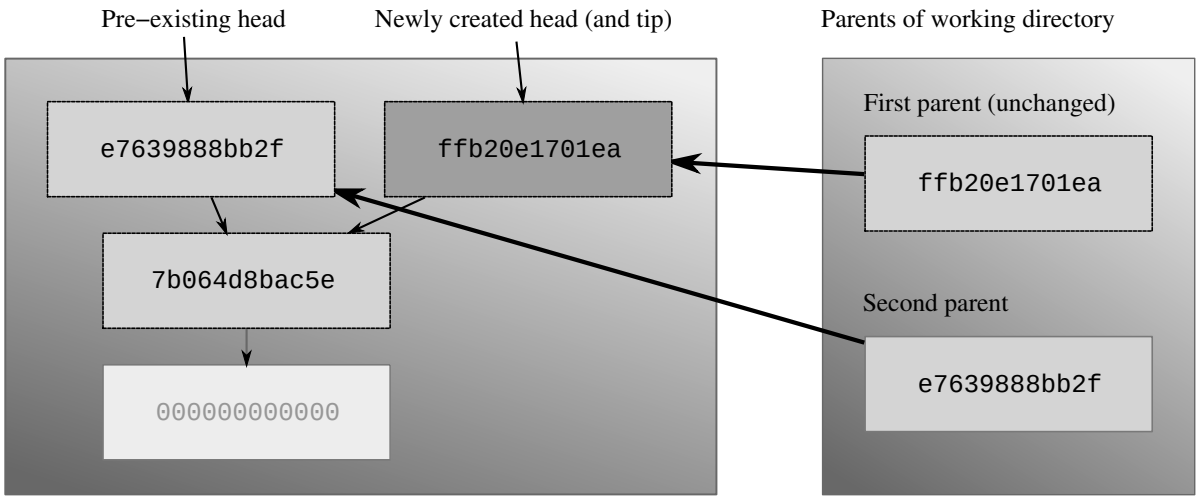

Figure 4.9: 2 つのヘッドのマージ

Chapter 5

Mercurialでの日常作業

5.1 追跡すべきファイルのMercurialへの登録

Mercurialは，ユーザによるファイル管理の指示がない限り，リポジトリ内のファイルであっても管理を行わない．Mercurialが管理しないファイルは“hg status”コマンドを実行すると“?”と表示される．

Mercurialにファイルの追跡をさせるには，“hg add”コマンドを用いる．一度ファイルを追加すると，“hg status”コマンドの出力は“?”から“A”に変わる．

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

“hg commit”コマンドを実行すると，commitの前に追加したファイルは“hg status”の出力に現れなくなる．これは，“hg status”がデフォルトでは，変更，削除，リネームなどを行った“注目すべき”ファイルのみを表示するためである．数千のファイルからなるリポジトリの場合，Mercurialが追跡しているものの，変更の加えられていないファイルについて何かを知りたいということは稀である．（もちろん知りたい場合は情報を得ることもできる．これについては後述する．）

追加したファイルに対してMercurialが直ちに行うことは何もないが，その代わりに次回のコミット時にファイル状態のスナップショットを取る．そしてファイルを削除するまでコミット毎にファイルの変化を追跡する．

5.1.1 明示的なファイル命名対暗黙のファイル命名

Mercurialの全てのコマンドは，引数としてディレクトリ名を渡すと，ディレクトリの内の全てのファイルとサブディレクトリに対する操作であると解釈するため便利である．

```
$ mkdir b
$ echo b > b/somefile.txt
$ echo c > b/source.cpp
$ mkdir b/d
$ echo d > b/d/test.h
$ hg add b
```

```
adding b/d/test.h
adding b/somefile.txt
adding b/source.cpp
$ hg commit -m 'Added all files in subdirectory'
```

この例では Mercurial は追加したファイル名を表示しているが，前の例で `myfile.txt` という名前のファイルを追加した際には表示していなかった点に注意されたい．

前の例では，コマンドラインで追加するファイルを明示的に指定したため，Mercurial はユーザが何をしようとしているのか分かっていると推定し，何も表示しなかった．

しかしディレクトリ名を与えることでファイル名を暗黙的に与えた場合、Mercurial は関連するファイルの名前を 1 つずつ表示する追加のステップを踏む．これによって何が起こっているのか理解しやすくすると共に，沈黙裏に厄介な問題が起こる可能性を減らしている．この挙動は Mercurial の大半のコマンドに共通している．

5.1.2 こぼれ話: Mercurial はディレクトリではなくファイルを追跡する

Mercurial はディレクトリ情報を追跡しない．その代わり，ファイルへのパスを追跡している．ファイルを作成する際には，まずパスのディレクトリ部分を補完する．ファイルを消去した後は，ファイルの含まれていた空のディレクトリを全て消去する．これは当然の挙動のように見えるが，実際上は小さな問題を引き起こす．すなわち，Mercurial は完全に空のディレクトリを表現することができないのである．

空のディレクトリが有用であることは滅多にないが，適当なワークアラウンドとして，リポジトリの動きを妨げない方法が存在する．Mercurial の開発者たちは，空のディレクトリを表現するために加わる複雑さは，その機能に見合わないと考えた．

リポジトリに空のディレクトリが必要な場合，これを実現する方法がいくつかある．1 つ目は，まずディレクトリを作成し，隠しファイルをこのディレクトリ内に “`hg add`” する．Unix 系システムでは，ピリオド (“.”) で始まるファイルは大半のコマンドと GUI ツールで隠しファイルとして取り扱われる．詳細については図 5.1を参照されたい．

```
$ 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
```

Figure 5.1: 隠しファイルを使って空のディレクトリをシミュレートする

空のディレクトリを扱う別の方法には，自動ビルドスクリプトの中で，必要になる前に作成することがある．

5.2 ファイル追跡の停止

ファイルがリポジトリに必要なくなった時は “`hg remove`” コマンドを実行する．このコマンドはファイルを削除した上で Mercurial にファイル追跡の停止を指示する．削除されたファイルは “`hg status`” コマンドの出力

で “R” と表示される．

```
$ 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
```

一度ファイルを “hg remove” した後では，Mercurial はワーキングディレクトリ内にたとえ同じ名前でファイルが再生成されてもファイルへの変更を追跡しない．ファイルを再生成し，Mercurial に追跡させたいのであれば，“hg add” を行う．Mercurial は，新規に追加されたファイルを同名の古いファイルと関係なく扱う．

5.2.1 ファイル削除は履歴に影響を与えない

ファイルの削除がリポジトリに与える影響は 2 つのみである．

- ワーキングディレクトリから原稿のバージョンのファイルを消去する．
- 削除するファイルへの変更を次回のコミットから Mercurial が追跡しないようにする．

ファイルをどのように削除してもファイルの**履歴**は変更**されない**．

ワーキングディレクトリを，消去したファイルが管理されていた頃にコミットされたチェンジセットに更新すると，ファイルはそのチェンジセットのコミット時の内容でワーキングディレクトリに再び現れる．そこでワーキングディレクトリをファイルの消去された後のチェンジセットに更新すると，Mercurial は再びファイルをワーキングディレクトリから消去する．

5.2.2 欠落したファイル

Mercurial は，“hg remove” を使わずに消去されたファイルを**欠落**したファイルとして扱う．欠落ファイルは “hg status” コマンドの出力で “!” と表示される．Mercurial のコマンドは，通常，欠落したファイルには何も行わない．

```
$ 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
```

リポジトリ内で “hg status” コマンドが欠落とリポートするファイルがある場合，“hg remove --after” を実行して，Mercurial に欠落ファイルが消去されたものであることを指示することができる．

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

一方で，誤ってファイルを消去した場合，“hg revert *filename*” を実行することでファイルを修復することができる．ファイルは何も変更されない状態で修復される．

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

5.2.3 こぼれ話: なぜ Mercurial へファイルの削除を明示的に指示しなければならないか

なぜ Mercurial にファイルの削除を明示的に指示しなければならないのだろうか？Mercurial の開発の初期には，ファイルがないことを検知すると，次に “hg commit” が実行される時に自動的にファイルの追跡をやめていた．実際に使ってみるとこの動作ではファイルを知らずのうちに失うことが頻発したのである．

5.2.4 役に立つ簡略法—ファイルの追加と削除を 1 ステップで行う

Mercurial は “hg addremove” という組み合わせコマンドを持っている．このコマンドは追跡されていないファイル群を追加し，欠落しているファイルを消去されたとマークする．

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

“hg commit” コマンドには-A オプションがあり，同じ追加と削除の操作をコミット直後に行う．

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

5.3 ファイルのコピー

Mercurial には，ファイルの新しいコピーを作る “hg copy” コマンドがある．このコマンドを使ってファイルをコピーすると，Mercurial は新しいファイルがオリジナルの複製であることを記録する．複製されたファイルは，別の人による変更をマージする時に特別な取り扱いがなされる．

5.3.1 マージ中のコピーの結果

マージ中には，コピーに “従う” ことが起きる．これの意味するところを示すために，例を挙げることにする．いつものように，ファイル 1 つだけを含む小さなリポジトリを作成する．

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

マージすべき内容を作るために並行していくつかの作業を行う必要があるので，リポジトリのクローンを行う．

```
$ 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って，“hg copy” コマンドを使って，作成したファイルのコピーを行う．

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

その後で “hg status” コマンドの出力を見るとコピーされたファイルは通常通りに追加されたファイルとして見える．

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

しかし “hg status” コマンドに-C オプションを渡すと，新たに追加されたファイルが**どの**ファイルからコピーされたのかを示す行が出力される．

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

ここでクローンしたリポジトリにも変更を加える．作成したファイルに一行を追加する．

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

これでこのリポジトリにも変更された file が存在することになる．ここで最初のリポジトリから変更を pull し，2 つの head をマージすると Mercurial はローカルな file に対する変更をそのコピー new-file に波及させる．

```
$ 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 変更はなぜコピーに従わなければならないか

ファイルへの変更がそのコピーへ波及する動作は難解に思えるかもしないが，多くの場合は非常に望ましい動作である．

まず第一にこの波及はマージを行うときに**のみ**発生する．従ってファイルを "hg copy" する場合や，通常の作業でオリジナルファイルを変更する場合は何も起きない．

知っておくべき点の 2 つ目は，変更は変更のマージ元のチェンジセットがコピーを**関知していない場合**にのみ波及するということである．

Mercurial がこのように動作する理由を説明する．今，私が重要なバグ修正を 1 つのソースファイルについて行い，変更をコミットしたとする．その間，あなたは自分のリポジトリの中で，そのバグあるいは修正を知らず，そのファイルの "hg copy" を行い，ファイルの変更を始めたとする．

あなたが私の行った変更を pull し，merge した場合，Mercurial は変更をコピー間で波及させない．あなたの新しいソースファイルは今バグを持っているかもしれないが，バグ修正を手で波及させなければバグはファイルのコピーに**残る**．

バグを修正する変更を自動的にオリジナルファイルからコピーへ波及させることによって，Mercurial はこの類の問題を避けている．私の知る限り，Mercurial は，このようなコピー間で変更を波及させる唯一のリビジョンコントロールシステムである．

コピーと後続のマージが起こったことが更新履歴に含まれる場合，変更をオリジナルファイルからコピーされたファイルへさらに波及させる必要はない．これが Mercurial が最初のマージの時にコピー間で変更を波及させ，その後では行わない理由である．

5.3.3 変更がコピーに従わないようにする方法

何らかの理由でこのように変更が自動的に波及するやり方が好ましくないと思われる場合は，システム標準のファイルコピーコマンド（UNIX 系システムでは cp）を使ってファイルのコピーを行い，"hg add" で新しいコピーを手動で追加することができる．これを実際に行う前に，5.3.2節を再読し，このやり方の詳細を踏まえた上で，あなたのケースに適しているか判断してほしい．

5.3.4 "hg copy" コマンドの挙動

"hg copy" コマンドを使う時，Mercurial はワーキングディレクトリ内のファイルをその時点の状態でコピーする．すなわち，ファイルに何らかの変更を加え，変更のコミットをせずに "hg copy" を行うと，新たなコピーもその時点までに加えた変更を含んでいる．（筆者はこの挙動はやや直観に反すると考えているため，敢えてここで言及した.）

"hg copy" コマンドは Unix の cp と同様に振舞う.（"hg cp" エイリアスを定義して使うことも可能だ.）引数は 2 つ以上が必要で，最後の引数は**コピー先**で，それ以外の引数は**コピー元**として扱われる．

コピー元として単一のファイルを指定し，コピー先ファイルが存在しない場合，コマンドはコピー先ファイルを新規に作成する．

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

コピー先がディレクトリの場合，Mercurial はコピー元をコピー先ディレクトリの中にコピーする．

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

ディレクトリのコピーは再帰的で，コピー元のディレクトリ構造を保つ．

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

コピー元とコピー先が共にディレクトリの場合，コピー元のツリー構造がコピー先のディレクトリの中に再現される．

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

“hg remove” コマンドと同様に，ファイルを手動でコピーした後で，Mercurial に操作を通知したい場合は，“hg copy” コマンドに--after オプションを付ければよい．

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

5.4 ファイルのリネーム

ファイルをリネームすることはファイルのコピーよりもよく行われる．ファイルのリネームよりも前に “hg copy” について述べた理由は， Mercurial が本質的にリネームをコピーと同様に扱うからである．従って，ファイルをコピーする時に Mercurial が行うことを知ることは，リネームの際に起こることを知ることになる．

“hg rename” コマンドを使う時，Mercurial は各々のソースファイルをコピーし，元のファイルを削除して，消去済みとマークする．

```
$ hg rename a b
```

“hg status” コマンドは新たにコピーされたファイルを追加されたファイルとして表示し，コピーされたファイルを消去されたファイルと表示する．

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

“hg copy” コマンドの場合と同様に， “hg status” コマンドに-C オプションを用いて，追加されたファイルが消去されたオリジナルファイルのコピーとして Mercurial に追跡されているのかを調べることができる．

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

“hg remove” と “hg copy” のように，--after オプションを使うことで，事後にリネームをすることができる．多くの点で “hg rename” コマンドの挙動と，このコマンドが受け入れるオプションは “hg copy” コマンドと類似している．

Unix コマンドに慣れているなら，“hg rename” の代わりに “hg mv” が使えることを知っているとよいだろう．

5.4.1 ファイルのリネームと変更のマージ

Mercurial のリネームはコピーと削除として実装されており，リネーム後にマージを行うのと，コピー後にマージを行うのでは同じ変更の波及が起きる．

私がファイルを変更し，あなたがそのファイルをを新しい名前にリネームした場合，我々がお互いの変更をマージすると元のファイル名に対する私の変更は，新しいファイル名のファイルに波及する．（これができるのは当たり前と思うかもしれないが，実のところ，全てのリビジョンコントロールシステムができるわけではない.）

変更がコピーに従う機能が有用であることは，おそらく容易に同意が得られるところであると思われる．とりわけリネームに追従する機能はきわめて重要であることは明白である．もしこの機能がなければ，ファイルのリネームによって変更はたやすく行き場を失ってしまうだろう．

5.4.2 名前とマージの発散

2 人の開発者のリポジトリ間で 1 つのファイル —foo と呼ぶことにする— について名前の発散が起こった場合について考えてみよう．

```
$ 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
```

アンはファイルを bar と改名した．

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

その間，ボブは同じファイルを quux と改名した．(“hg mv” が “hg rename” のエイリアスであることに注意)

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

各々の開発者はファイルが何と呼ばれるべきか異なった意見を持っており，これは名前のコンフリクトである．

彼らがマージを行った際にどうなればよいだろうか？ Mercurial の実際の挙動は，発散したリネームがあるチェンジセットをマージした場合は常に**両方**の名前を保存する．

```
# 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
```

Mercurial は名前の発散について警告するが，マージ後も名前の発散の解決はユーザに任せる点に注意．

5.4.3 リネームとマージによる収束

2 人のユーザが異なる**ソース**ファイル群を同一の**目的**ファイルにリネームするとコンフリクトが起きる．この場合，Mercurial は通常のマージ機構を起動し，ユーザに適切な解決を促す．

5.4.4 名前に関連したいくつかの問題

Mercurial には，一方を名前を与えたファイル，もう一方を同名のディレクトリとしてマージを行うと失敗するバグが以前からある．これは Mercurial bug no. 29 としてドキュメント化されている．

```
$ 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/issue29PMmG8_/issue29/b
```

5.5 ミスからの回復

Mercurial には一般的なミスからの回復を助けるいくつかの有用なコマンドがある．

“hg revert” コマンドはワーキングディレクトリ内で行った変更を取り消す．たとえば，誤って “hg add” を行った場合，追加したファイル名と共に “hg revert” を実行すればファイルは何も変更されず，Mercurial の管理から外される．また “hg revert” はファイルへの間違った変更を消去するのも使える．

“hg revert” コマンドは未コミットの変更に対して有効であることを覚えておくと良い．変更をコミットしたあとで，これが間違いであったことに気づいた場合は，とれる手段はやや限られる．

“hg revert” コマンドについてのより詳細な情報と，すでにコミットした変更への操作については 9の章を参照されたい．

5.6 複雑なマージの取り扱い

複雑なプロジェクトや大規模なプロジェクトでは，2 つのチェンジセットをマージする時に頭を悩ませることが少なくない．マージ元の双方で盛んに編集されている大きなソースファイルがあるとすると，コンフリクトが生じることはほぼ不可避であり，収拾するためには少々手数がかかる．

このような場合について，簡単な例で解決方法をを見てみよう．まず，ファイル 1 つのみを含むリポジトリを作り，これを二度クローンする．

クローンの一方でファイルを変更する．

もう一方のクローンでも別の変更を行う．

次に元のリポジトリに各々の変更を pull する．

元のリポジトリには今ヘッドが 2 つある．

通常の場合，ここで “hg merge” を実行すると，ファイル myfile.txt のコンフリクトを手動で解決するための GUI が起動する．しかしここでは説明を単純にするために，直ちに失敗するマージをしてみよう．その方法はこのようになる．

自動的に片付かないマージが見つかった場合は，Mercurial のマージメカニズムが false コマンド（これは希望通り直ちに失敗する.）を実行するようにさせる．

ここで “hg merge” を実行すると，実行は停止して，マージの失敗が報告される．

ユーザがマージの失敗に気付かない場合でも，Mercurial は自動的に失敗したマージの結果のコミットを妨げる．

“hg commit” が失敗すると，あまり馴染みのないコマンドである “hg resolve” を使うようにメッセージが表示される．このコマンドの使用法はいつも通り “hg help resolve” を実行することで見ることができる．

5.6.1 ファイル解決状態

マージの際にほとんどのファイルはそのまま残される．Mercurial は操作を行うファイルの状態を管理している．

- **解決済み**ファイルはマージが成功したファイルで，Mercurial によって自動的に行われたか，人がマージ作業を行ったかは問わない．
- **未解決**ファイルはマージが失敗し，なんらかの修正が必要なファイルである．

Mercurial はマージ後に未解決状態のファイルを見つけると，マージが失敗したと判断する．幸いなことにこの場合でもマージを始めからやり直す必要はない．

“hg resolve” コマンドで--list オプションや-l オプションを使うと，マージされた各々のファイルの状態が表示される．

“hg resolve” の出力で，解決済みファイルは R，未解決のファイルは U と表される．U となるファイルがある場合，マージ結果のコミットは失敗する．

5.6.2 ファイルマージの解決

ファイルを未解決状態から解決済み状態へ持っていく方法はいくつかある．最もよく用いられるのが再度 “hg resolve” を実行する方法である．このコマンドは各々のファイルやディレクトリの名前を渡すと，未解決のファイルに対してマージを再度試みる．--all または-a オプションを渡して**全ての**未解決ファイルに対して再マージを行うこともできる．

Mercurial によってファイルの解決状態を直接変更することも可能だ．--mark オプションを使うことで，ファイルを解決済みとマークしたり，逆に--unmark オプションで未解決とマークすることもできる．これらによって混乱したマージを手動で整理して，作業を続けることができる．

Chapter 6

他の人々との共同作業

完全な分散型ツールとして，Mercurial はユーザが他のユーザとどのように作業するかのポリシーを強要することはない．しかし初めて分散リビジョンコントロールツールを使うのであれば，いくつかのツールの使用法と使用例を知ることが取り得るワークフローモデルを考える際に助けとなるであろう．

6.1 Mercurial のウェブインタフェース

Mercurial はいくつかの有用な機能を持つ強力なウェブインタフェースを備えている．

対話的な利用では，ウェブインタフェースにより 1 つのリポジトリまたはいくつかのリポジトリのコレクションを閲覧することができる．リポジトリの履歴を見たり，各々の変更（コメントや差分を含む）を調べたり，ディレクトリやファイルの内容を見ることができる．また，履歴を表示し，個々の変更やマージの関係をグラフィカルに表示することも可能である．

ウェブインタフェースは閲覧用にリポジトリの変更の Atom および RSS フィードを提供する．これを使えば，リポジトリの変化を好みのフィードリーダによって “購読” することができ，リポジトリでの活動が起こるとすぐさま通知を受けられる．この機能は誰がリポジトリのサービスを行っても追加の設定を必要としないため，メーリングリストを購読して通知を受けるモデルよりもずっと便利である．

リモートユーザはウェブインタフェースを用いてリポジトリをクローンすることもできる．変更を pull して，(サーバが許可する設定になっていれば）加えた変更を再び push することもできる．Mercurial の HTTP トンネルプロトコルはデータを積極的に圧縮するため，バンド幅の低いネットワークコネクションでも有効に機能する．

ウェブインタフェースの最も簡単な始め方はウェブブラウザを使って Mercurial のマスタリポジトリ `http://www.selenic.com/repo/hg` のような既存のリポジトリを参照することである．

自分のリポジトリにウェブインタフェースを用意する場合，いくつか良い方法がある．

公式ではない環境で行う最も簡単で早い方法は，“`hg serve`” コマンドを使う方法で，これは短期間の “手軽な” サービスに的している．このコマンドの詳細な使用法については下記の 6.4節を参照のこと．リポジトリを長期間にわたり永続的にサービスしたい場合は，Mercurial に内蔵の CGI(Common Gateway Interface) サポートを利用することができる． CGI の設定については 6.6節を参照のこと．

6.2 共同作業モデル

適切で柔軟なツールをもってしても，ワークフローに関する決定をすることは技術的というよりは社会工学的なチャレンジである． Mercurial がプロジェクトのワークフローに課す制限はほとんどないため，これをいかに構築するかはあなたとその共同作業者に任されており，固有の要求マッチするモデルを作ることができる．

6.2.1 考慮すべき要素

どのようなモデルを使う場合でも，それが作業する人々の要求と能力に適ったものであるかを常に念頭に置くことが最も重要である．これは自明のことのように思えるかもしれないが，片時も忘れてはならない．

自分にとって完全と思えるワークフローモデルを構築したつもりが，共同開発チームにとっては大きな驚きと葛藤を与えてしまったことがある．複雑なブランチの集合がなぜ必要なのか変更がブランチ間でどのように伝播するのかを説明したにもかかわらず，幾人かのチームメンバーは反発した．彼らは聡明であったが，私の拘ったルールが作業に与える制限や，モデルの細部に与える影響に注意を払うことを望まなかった．

将来起こり得る社会的または技術的問題に目を瞑ってはならない．どのような方法をとるにしても，間違いや問題が起きた場合に備えておく必要がある．想像し得るトラブルを防止したり，トラブルから素早く回復させるための自動化された方法を考えておく必要がある．たとえば，リリースに含めない変更を行ったブランチがあるとしたら，誰かが誤ってそこからリリースブランチに変更をマージしてしまう可能性について検討しておくべきである．この場合であれば，不適切なブランチからのマージを禁止するフックを用意することで問題を回避することができる．

6.2.2 非公式な混乱

ここで述べる “なんでもあり” アプローチが持続可能であると言いたいわけではないのだが，このモデルは把握しやすく，いくつかの状況ではうまく機能するものだ．

例えばプロジェクトの多くは実際にはほとんど会うことのない協力者たちの緩やかなグループを持つ．グループのいくつかは機会毎に短い “スプリント” を形成し，離れた場所での隔絶した作業を克服している．スプリントでは,（企業の会議室，ホテルのミーティングルームのような）一カ所に集まって数日間にわたって少数のプロジェクトを集中的にハックする．

スプリントやコーヒーショップでのハッキングセッションは “`hg serve`” コマンドを使うのまさににうってつけの環境である．“`hg serve`” は手の込んだサーバ設備を必要としない．下の 6.4セクションを読んですぐに “`hg serve`” コマンドを使うことができる．サーバを起動していることを隣の開発者に話したり，グループに URL をインスタントメッセージで送れば，すぐに迅速な共同作業ができる．送った URL を他の開発者がブラウザに入力すれば，彼らは簡単にあなたの変更をレビューすることができるし，あなたの行ったバグフィックスを pull して検証することもできる．さらに，新機能の実装されたブランチをクローンして試すこともできる．

アドホックなやり方で行う共同作業の魅力と問題は，あなたの変更を知っていて，場所も分かっている人々しか変更を参照できないことである．このような非公式なアプローチは，各人が n 個の異なったリポジトリのどれから pull を行えばいいか分かっている必要があるため，少人数以上にスケールしない．

6.2.3 1 つの集中リポジトリ

集中型リビジョンコントロールツールを使っていた小規模なプロジェクトが移行する場合には，中央に単一の共有リポジトリを設けて，その中で変更を管理するのが最も簡単な方法であろう．このようなリポジトリはもっと曖昧なワークフロースキームにとっても共通の基盤となる．

協力者はリポジトリのコピーをクローンすることから始める．そして必要な時に変更を pull し，幾人かの（全員かもしれない）開発者は，自分の行った変更を必要な時に push する権限を持つ．

このモデルでも，中央のリポジトリを参照せず他の開発者のリポジトリから直接変更を pull することには意味がある．手元で仮のバグ修正を行ったが，これを中央のリポジトリで公開し，他の開発者達が pull すると各々のソースツリーが壊れるおそれがある場合を考える．ダメージの可能性を下げるために，別の開発者に自分のリポジトリをクローンし，テストすることを依頼することが考えられる．この方法を取ることで，最低限のテストを行うまで，問題のある可能性のある変更を公開することを引き延ばすことができる．

この状況では，開発者たちは通常，中央のリポジトリへ変更をプッシュするために，6.5節で説明したように `ssh` プロトコルを用いる．また 6.6節で述べたように，リポジトリの読み取り専用コピーを HTTP で公開するのも一般的である． HTTP による公開で，プッシュアクセス権を持たない人々やリポジトリの履歴をブラウザで参照したい人のニーズを満たすことができる．

6.2.4 ホスティングによる中央リポジトリサービス

Bitbucket(http://bitbucket.org) のような公共のホスティングサービスでは，ユーザアカウントの設定，認証，セキュアな通信プロトコルなどの面倒なサーバ設定を肩代りしてくれるだけでなく，このモデルがもっと良く機能するためのインフラを提供している．

例えば，うまく構成されたホスティングサービスは，1 クリックでリポジトリのコピーをクローンできるようになっている．これにより，別々の場所で作業を行い，準備が出来次第変更を共有することができるようになる．

また，良いホスティングサービスは，開発者同士が “このツリーにレビューして貰いたい変更がある” などのやりとりをできるような機能も提供している．

6.2.5 複数のブランチでの作業

ある程度の規模のプロジェクトでは，自ずといくつもの前線で開発を進めていくようになる．ソフトウェアの場合，定期的に公式リリースを行うのが通例である．リリースは，公開後暫くして “メンテナンスモード” に移行するかもしれない．メンテナンスリリースではバグの修正だけを行い，新機能の追加は行わないのが通例である．これらのメンテナンスリリースと並行して，将来のリリースが開発される．通常，開発の進行していく若干異なった方向の各々を表すために “ブランチ” という呼称を用いる．

Mercurial は複数ブランチによる同時開発を取り扱うことに特に適している．各々の “開発方向” は中央リポジトリに置くことが可能で，必要になる度にあるブランチから別のブランチへマージできる．リポジトリはそれぞれ独立であるため，開発ブランチでの不安定な変更は，開発者の誰かが明示的にマージを行わない限り安定版ブランチにはなんの影響も与えない．

実際にどのように動作するのかを例で示す．中央のサーバに 1 つの “メインブランチ” があるとしよう．

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

開発者はこれをクローンし，ローカルで変更を加え，テストし，また push するとする．

メインブランチがリリースマイルストーンに到達した際に，“hg tag” で永続的な名前をマイルストーンリビジョンに付けることができる．

```
$ hg tag v1.0
$ hg tip
changeset:   1:5c2316c36766
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:35 2009 +0000
summary:     Added tag v1.0 for changeset 51d81d94bbc6

$ hg tags
tip                                1:5c2316c36766
v1.0                               0:51d81d94bbc6
```

メインブランチで進行中の開発について見てみよう．

```
$ 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!
```

これ以後リポジトリをクローンした人はマイルストーンで記録されたタグと “hg update” コマンドを使って，タグの付けられたリビジョンと全く同じワーキングディレクトリを復元することができる．

```
$ 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.
```

さらに，メインブランチがタグ付けされた直後からサーバ上のメインブランチを新たな “stable” ブランチにクローンできる．これはサーバ上で行うことも可能である．

```
$ 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
```

stable ブランチに変更を加えたい場合，**その**リポジトリをクローンし，変更を行い，コミットした後にその変更をサーバに push することができる．

```
$ 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/branching1MDRLB/stable
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

Mercurial リポジトリは独立で，変更を自動的に波及させることもないので，stable と main ブランチは互いに**隔離されている**．メインブランチに行った変更が stable ブランチに漏れ出したり，その逆になったりすることはない．

多くの場合，stable ブランチに対して行ったバグ修正をメインブランチに対しても取り込みたいと考えるだろう．バグ修正をメインブランチでもう一度行うのではなく，Mercurial を使って変更を stable ブランチから pull して簡単にメインブランチへマージすることができる．

```
$ 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!
```

メインブランチは stable ブランチにあるバグ修正をすべて取り込んでいるだけでなく，stable ブランチにはない変更も持っている．この操作を行っても stable ブランチはこれらの変更による影響を受けない．なぜなら変更は stable ブランチからメインブランチへの一方向のみに波及し，逆向きには波及しないからである．

6.2.6 機能によるブランチ

大規模なプロジェクトで変更を効果的に管理する方法は，チームを小さなグループに分割することである．各々のグループはプロジェクト全体で使われる単一のマスターブランチからクローンした固有の共有ブランチを持つ．各々のブランチで作業する人達は，他のブランチでの開発から隔離されているのが通例である．

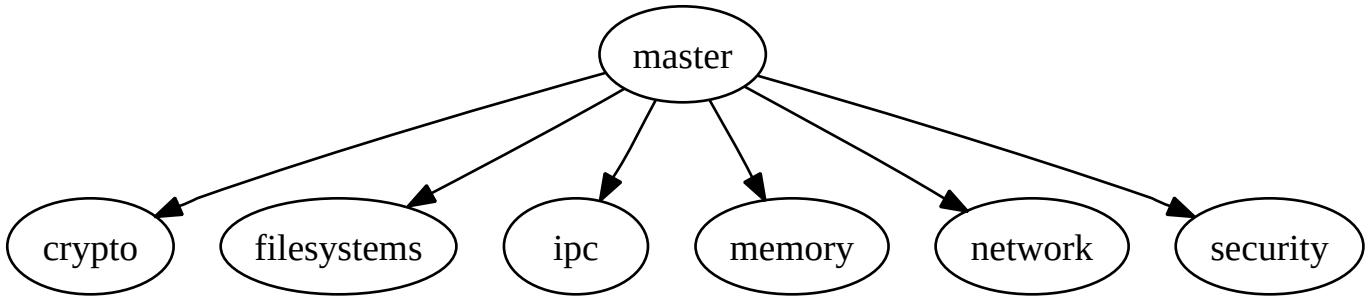

Figure 6.1: 機能によるブランチ

特定の機能が十分な状態になったとき，その機能のチームメンバーはマスターブランチから機能ブランチに pull とマージを行い，マスターブランチに push して戻す．

6.2.7 リリーストレイン

いくつかのプロジェクトはトレイン方式で組織されている．数カ月に一度リリースを行うようにスケジュールされ，トレインが出発できるように新機能が準備される．

このモデルは機能ブランチでの作業と似ている．違いは，機能ブランチはトレインを逃した場合，機能チームのメンバーが機能ブランチのトレインリリースへ行くべき変更を機能リリースに pull およびマージし，開発中の機能が次のリリースに入るようにチームはそのリリースの上で作業を続ける．

6.2.8 Linux カーネルモデル

Linux カーネルの開発では，周りに混沌とした広がりを持つ浅い階層構造が取られている．殆んどの Linux 開発者は Mercurial と同様の機能を持つ `git` という分散リビジョン管理ツールを利用しているため，彼らが取っているワークフローを述べることは我々にも役立つ．彼らのアイディアを気に入れば，そのアプローチはツールの違いを越えて有効である．

開発コミュニティの中心は Linux の創造者である Linus Torvalds である．彼はリポジトリを 1 つ公開しており，それは開発コミュニティ全体の公式カレントツリーと見做されている．誰でも Linus のツリーをクローンできるが，彼自身はどこから pull するかを厳しく選んでいる．

Linus は多くの “信頼できる代行者” を持っている．原則的には彼は彼らが公開したものは何でも pull する．多くの場合，彼らの変更をレビューすることもしない．代行者の何人かはメンテナとなることを同意しており，カーネル内の特定のサブシステムに対して責任を持つ．カーネルハッカーの誰かが Linus のツリーのサブシステムに変更を加えたいと思ったら，サブシステムのメンテナを見つけ出した上で，メンテナに変更を取り込んで貰うように依頼する必要がある．メンテナが変更をレビューし，取り込むことに同意すれば，正式なコースで変更を Linus に手渡す．

それぞれの代理人はレビュー，変更の公開，いつ Linus に提出するかについてそれぞれ独自のアプローチを取っている．また，別の用途向けによく知られたブランチがいくつかあり，例えば幾人かの人はカーネルの過去のバージョンの “stable” リポジトリをメンテナンスし，重要な修正を行っている．別のメンテナは，実験的な変更を取り込むツリー，上流に手渡す寸前の変更を取り込むツリーなどのように複数のツリーを公開している．また 1 つのツリーだけを公開しているメンテナも多い．

このモデルは注目すべき特長を 2 つ持っている．第一に，これらは pull オンリーである．変更を取り込んで貰うためには，他の開発者に依頼したり，説得したり，懇願したりしなければならない．なぜなら 2 人以上がプッシュできるツリーはほとんどなく，別の人がコントロールしているツリーに変更を push する術はないからである．

第二に，これは評判と称賛によるシステムであるということだ．無名の開発者の変更に対しては Linus はおそらく反応することなく無視する．しかしサブシステムメンテナは変更をレビューし，適切であると判断すれば採り入れる．開発者が良い変更を行えば行うほど，メンテナは開発者の判断を信用し，変更を採り入れるようになるだろう．開発者が著名で，Linus がいまだに受け入れていない，長期間にわたるブランチのメンテナであるならば，同じ興味を持つ人々が彼の作業を取り込むために変更を定期的に pull することだろう．

評判と称賛はサブシステムを越えたり，周辺にいる人々を越えることはないだろう．もしあなたがストレージの分野で敬意を集めるハッカーであったとしても，ネットワークのバグを修正しようとしたのなら，行った変更はネットワークのメンテナから完全な部外者と同等の精査を受けることになるだろう．

整然としたプロジェクトから来た開発者にとっては，Linux カーネルの混沌とした開発プロセスはしばしば完全に狂気の沙汰と思えることだろう．Linux カーネルでは，開発プロセスは個々人の気まぐれに依存しており，適切と考えた時は全面的な変更を行い，開発ペースは途方もない．それでもしかし Linux は大いに成功した注目すべきソフトウェアなのである．

6.2.9 Pull のみ vs 共有 push コラボレーション

オープンソースコミュニティでは他の開発者のところから変更を pull するだけの開発モデルが，共有リポジトリに多数の開発者が push できるモデルよりも “優れている” かどうかが常に激しい論争の種になる．

共有プッシュモデルの支持者は，このアプローチを強制するようなツールを使っていることが多い．もし Subversion のような中央集中型のリビジョン管理ツールを使っているのなら，使うモデルを選ぶ余地はそもそも存在しない．これらのツールでは共有 push モデルを使う他なく，それ以外の何かをしたいのであれば,（自力でパッチを当てるなどの方法で）外部で行う必要がある．

良い分散リビジョンコントロールツールは，両方のモデルをサポートする．ユーザや協力者はツールによって強要されるモデルではなく，要求や好みに応じた共同作業の構成を決めることができる．

6.2.10 共同作業がブランチ管理と直面するところ

開発者とそのチームが共有リポジトリをセットアップし，ローカルと共有リポジトリの間で変更をやりとりし始めると，これと関連するが，やや異なった困難に直面するようになるだろう．それは，チームが同時に進め

る多方面への開発をどのように管理するかという問題である．この課題は，チームがどのように共同するかと根源的に関係しており，これだけのために 8という一章を費やす価値があるだろう．

6.3 共有の技術的側面

この章の残りの部分では，協力者と変更を共有する際の疑問点について述べる．

6.4 “hg serve”による非公式な共有

Mercurial の “hg serve” コマンドは，小規模で緊密な開発ペースの速いグループ環境にとても適している．Mercurial コマンドをネットワーク越しに使う素晴らしさを体感できるだろう．

リポジトリ内で “hg serve” コマンドを起動すると，すぐさま特別な HTTP サーバが立ち上げられる．これはあらゆるクライアントからの接続を受け入れ，リポジトリのデータをサービスする．起動したサーバの URL を知っている人やあなたのコンピュータにネットワークを介して接続可能な人は誰でも，ウェブブラウザや Mercurial を使ってリポジトリからデータを読み出すことができる．ラップトップで起動された “hg serve” インスタンスへの URL は http://my-laptop.local:8000/のようになる．

“hg serve” コマンドは一般用途向けのウェブサーバでは**ない**．このコマンドは次の 2 つのことだけを行う：

- 普通のウェブブラウザを使っているユーザに対してはリポジトリの履歴への参照．
- Mercurial を使っているユーザに対しては，“hg clone” または “hg pull” ができるように Mercurial ワイヤプロトコルのサポート．

“hg serve” はリモートユーザにリポジトリの変更を許可しない．このコマンドは読み出しのみの使用を意図している．

Mercurial の “hg serve” コマンドを使って，簡単に手元のコンピュータでリポジトリサービスを行うことができる．遠方にあるサーバとやりとりするのと同様に “hg clone”，“hg incoming” 等のコマンドを使うことができる．これはネットワークでリポジトリを提供する練習になるだろう．

6.4.1 覚えておくべき 2, 3 の点

“hg serve” は，認証なしでアクセスを許すため，ネットワークへのアクセスや，リポジトリからのデータ pull を誰が行っても構わないようなネットワーク環境や，完全な制御が可能なネットワーク環境でのみ使用すべきである．

“hg serve” コマンドは，システムやネットワークにインストールされているファイアウォールソフトウェアについては何も関知しない．このコマンドはファイアウォールの発見や制御はできない．他のユーザが “hg serve” コマンドにアクセスできない場合，まず彼らが正しい URL を使用しているか確認し，その次にすべきことは，ファイアウォールの設定を確認することである．

デフォルトでは “hg serve” は到着する接続をポート 8000 で待つ．使用したいポートをすでに他のプロセスが使用している場合は-p オプションを使って別のポートで待機するように指定することができる．

通常，“hg serve” は始動してもメッセージの出力を行わない．これは多少混乱させるかもしれない．実際に正しく動作しているかどうか確認したい場合や，協力者に教える URL が知りたい場合は-v オプションを指定する．

6.5 Secure Shell (ssh) プロトコルの使用

Secure Shell (ssh) プロトコルを使うことで，変更をネットワーク上で安全に push できる．このプロトコルの利用には，クライアント側かサーバ側に少々設定が必要である．

ssh に馴染みのないユーザのために説明すると，ssh は他のコンピュータと安全に通信を行うためのコマンドおよびネットワークプロトコルの名称である．Mercurial で使うためには，1 つ以上のアカウントをサーバに設定し，リモートユーザがログインし，コマンドを実行できるようにする必要がある．

（ssh に馴染みのあるユーザには以下の話は初歩的に感じられるに違いない．）

6.5.1 ssh の URL をどのように読むか

ssh URL は一般に次のようになる：

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

1. “ssh://” 部は Mercurial に ssh プロトコルを使うことを指示する．
2. “bos@” 部はサーバへのログインに使うユーザ名を指定する．ローカルマシンでのユーザ名と同じものを使用する場合は指定しなくてもよい．
3. “hg.serpentine.com” はログインするサーバのホスト名である．
4. “:22” サーバの接続ポートを制定する．デフォルトポートは 22 なので，22 番以外を使う時のみ指定する必要がある．
5. URL の残りの部分はサーバ上のリポジトリへのローカルパスである．

ssh URL のパス部については，ツール向けに変換する標準的な方法がないため，混乱が多い．いくつかのプログラムと，その他のプログラムではパスを扱う際の挙動が異なっている．この状況は理想とはかけ離れているが，修正するのは困難だと思われる．以下の段落を注意深く読んで欲しい．

Mercurial はリポジトリへのパスをリモートユーザのホームディレクトリからの相対パスとして取り扱う．例えばサーバでユーザ foo はホームディレクトリ/home/foo を持つ．従って bar を含む ssh URL のパス部は /home/foo/bar となる．

他のユーザのホームディレクトリへの相対パスを指定したい場合は，次の例のようにチルダにユーザ名を続けたパスを使うことができる．（ここでは他のユーザのユーザ名を otheruser とする．）

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

サーバ上で絶対パスを指定したい場合は，次の例のようにパス部を 2 つのスラッシュで始める．

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

6.5.2 利用中のシステム向けの ssh client を見つける

Unix 系システムの殆んどは OpenSSH がプリインストールされている．そのようなシステムでは，which ssh を実行して ssh コマンドがインストールされているかどうか調べることができる．（普通は/usr/bin にインストールされているはずだ．）万が一インストールされていなかった場合は，システムのドキュメントを参照してインストール方法を調べて欲しい．

Windows では，TortoiseHg パッケージに Simon Tatham による優れたコマンドである plink が同梱されており，何も設定することなく利用可能である．

6.5.3 鍵ペアの作成

ssh クライアントを使う度に繰返しパスワードを入力するのを避けるために，鍵ペアを作成することを勧める．

> **Note:** 鍵ペアは必須ではない
> Mercurial 自身は ssh での認証や鍵ペアについては一切関知しない．ssh パスワードの入力に倦むことがなければ，この節と後の節を無視しても差し支えない．

- Unix 系システムでは，ssh-keygen コマンドで鍵ペアを作成できる．

- Windows で TortoiseHg を使っているのであれば，PuTTY のウェブサイト http://www.chiark.greenend.org.uk/~sgtatham/putty からダウンロードできる puttygen というコマンドで鍵ペアを作成できる．このコマンドの使用法の詳細については puttygen のドキュメント http://the.earth.li/~sgtatham/putty/0.60/htmldoc/Chapter8.htm を参照されたい．

鍵ペアを作る際にはできるだけパスフレーズで保護することを強く勧める．(安全なネットワークで ssh プロトコルによって自動タスクの実行をする場合にはこうしたくないだろう．)

しかし鍵ペアを作るだけでは十分でない．リモートにログインしたいマシンのユーザの authorised keys として公開鍵を追加する必要がある．OpenSSH を使っているサーバ（大部分が相当する）では.ssh ディレクトリの authorized_keys ファイルへ公開鍵を追加する．

Unix 系システムでは公開鍵は.pub という拡張子を持つ．Windows で puttygen を使用する場合は，ファイルセーブした公開鍵か，鍵を表示しているウィンドウから authorized_keys ファイルにペーストすればよい．

6.5.4 認証エージェントの使用

認証エージェントはパスフレーズをメモリに保存するエージェントである．(ログアウトし，再びログインした時にはパスフレーズは失われている．) ssh クライアントはこのデーモンが動作していることを認識し，パスフレーズの問い合わせを行う．認証エージェントが動作していない場合やエージェントが必要なパスフレーズを保存していない場合は，Mercurial がサーバと（変更を pull や push する）通信の時にパスフレーズを入力する必要がある．

パスフレーズをエージェントに記憶させる弊害は，パワーサイクルを行っても場合によっては周到な手段を用いる攻撃者にパスフレーズのプレーンテキスト情報を取得される可能性があることである．このリスクが許容できるかどうかは自分自身で判断して欲しい．この方法を用いることで，タイプ回数を減らせることは確かである．

Unix 系システムではエージェントは ssh-agent と呼ばれ，ログインすると自動的に起動される．ssh-add コマンドを用いてパスフレーズをエージェントに記憶させる．

Windows で TortoiseHg を使っている場合は，pageant コマンドがエージェントとして動作する． puttygen コマンドの時と同様に pageant コマンドは PuTTY のウェブサイト http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html からダウンロードできる．使用法については http://the.earth.li/~sgtatham/putty/0.60/htmldoc/Chapter9.html\#pageant を参照されたい． pageant コマンドは記憶したパスフレーズを管理するためにシステムトレイにアイコンを追加する．

6.5.5 サーバの正しい設定

ssh は慣れていないと設定が難しいため，新たに使い始める場合，様々な不具合が起きる可能性がある．Mercurial と共に動かす場合，さらに多くが待ち構えている．これらの殆んどがクライアント側ではなくサーバ側で起きる．しかし一度きちんと動作する設定をしてしまえば，動作はずっと続く．

Mercurial から ssh サーバに接続する前に， ssh または putty コマンドを使ってサーバに接続してみることを勧める．これらのコマンドを直接使って問題が起きるようであれば，Mercurial は動作しないはずだ． ssh の上で Mercurial を使うことで，下位の問題が隠れてしまうので，ssh に関連した Mercurial の問題をデバッグする時は，まず ssh クライアントコマンド自体が動作することを確認し，その後に Mercurial の問題を解決すべきである．

サーバ側でまず確認すべきなのは，他のマシンからログインできるかどうかである． ssh または putty コマンドでログインできない場合は，エラーメッセージに何が悪いのか示すヒントがあるかも知れない．最も一般的な問題を以下に列挙する．

- “connection refused” エラーが出る時は，SSH デーモンが動作していないか，ファイアウォール設定のためにマシンへのアクセスが不可能である可能性がある．
- “no route to host” エラーが出る場合は，サーバのアドレスを間違えているか，ファイアウォールがサーバを完全に隠してしまっていることが考えられる．
- “permission denied” エラーが出る場合は，ユーザ名を間違って入力しているか，ログイン用鍵のパスフレーズやユーザパスワードを間違って入力している可能性がある．

まとめると，サーバの ssh デーモンへ接続する際には，まずデーモンが動作しているかを確認すること．デフォルトでインストールされてはいるが，停止されているシステムもある．これを確認した後でサーバのファイアウォールが ssh デーモンの待機してるポート（通常は 22 番）への接続を許可しているか確認する．他の諸々の可能性を考える前にまずこの 2 点を確認すべきである．

クライアント側で鍵のパスフレーズを記憶させるために認証エージェントを動かしているなら，パスフレーズやパスワードの入力を促されることなしにサーバにログインできるはずだ．もしパスフレーズの入力を要求されるなら，いくつかの可能性が考えられる．

- パスフレーズを記憶させるために `ssh-add` または `pageant` を実行していない．
- 別のキーのパスフレーズを記憶させている．

リモートユーザのパスワードを要求される場合は別の問題があるかもしれない．

- ユーザのホームディレクトリまたは.ssh ディレクトリのパーミッションが緩すぎる．このため，ssh デーモンが authorized_keysauthorized_keys ファイルを信頼できないか，あるいは単純に読めない．例えばグループ書き込みパーミッションのあるホームディレクトリまたは.ssh ディレクトリはこの問題をしばしば引き起こす．
- ユーザの `authorized_keys` ファイルに問題がある．ファイルの所有者が別のユーザだったり，他のユーザが書き込みできる場合は ssh デーモンはこのファイルを信頼せず，読み込まない．

理想では，次のコマンドを実行し，1 行で現在の日時を表示できるべきである．

```
ssh myserver date
```

サーバで，バナーやその他の意味のない文字列を表示するようなログインスクリプトを使っている場合，対話的なコマンド以外ではこれらを表示しないようにする．これらの文字列は Mercurial の出力に混入してしまい，Mercurial コマンドをリモート実行する妨げとなる．Mercurial は非対話的な `ssh` ではこれらのバナーを検出し，無視しようと試みるが，これは万全ではない．（サーバのログインスクリプトを編集する場合，ログインスクリプトが対話的シェルで動作しているかを知る方法として，`tty -s` コマンドのリターンコードをチェックする方法がある．）

ssh 単体でサーバに接続できることを確認したら，Mercurial がサーバで動作することを確認する．次のコマンドが動くかどうか調べてみよう：

```
ssh myserver hg version
```

“`hg version`” の出力ではなくエラーメッセージが表示されるときは，`/usr/bin` に Mercurial がインストールされていないことが多い．この場合，`/usr/bin` にインストールし直す必要はない．その代わり，いくつかの問題をチェックすべきである．

- サーバに Mercurial は本当にインストールされているか？ これは下らない問いのように思えるが，確認する価値はある．
- シェルのサーチパスが正しく設定されていない．（通常は環境変数 `PATH` で設定される．）
- 対話的なログインセッションのとき以外は環境変数 `PATH` が `hg` 実行ファイルのあるパスを指していない．`PATH` を不適切なログインスクリプトで設定しているとこの問題が起きる．詳しくはシェルのドキュメントを参照すること．
- 環境変数 `PYTHONPATH` が Mercurial Python モジュールを含む必要がある場合．これが全く設定されていないか，対話的なログインでのみ有効になっている．

ssh 接続で “`hg version`” を実行できたのなら準備は完了だ．サーバとクライアントの設定は正しく行われている．これでサーバ側のユーザ名を使ってホストされているリポジトリに Mercurial を使ってアクセスできるようになった．ここで問題があるのなら，`--debug` オプションを使って何が問題なのかをより明確に把握して欲しい．

6.5.6 ssh での圧縮の利用

Mercurial は，ssh プロトコルを使った場合は，データの圧縮は行わない．ssh プロトコルが透過的にデータの圧縮を行うことができるためである．しかし ssh クライアントのデフォルトの挙動では，圧縮を行わ**ない**．

高速な LAN 以外のネットワーク（ワイヤレスネットワークも含む）では，Mercurial のネットワーク動作を高速化するのに圧縮の使用はとても効果的である．あるユーザの計測によれば，WAN 経由での大規模なリポジトリのクローンは，圧縮を使うことで 51 分から 17 分に短縮することができた．

`ssh` コマンドも `plink` コマンドも圧縮を有効にする`-C` オプションが使える． `hgrc` ファイルを編集して Mercurial が圧縮つきの ssh プロトコルを使用するように設定することができる． Unix 系システムで通常の `ssh` コマンドを設定する例は次のようになる．

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

Unix 系システムで `ssh` を用いてサーバへ接続している場合，サーバへの接続時に常に圧縮を使用するように設定することができる．設定には`.ssh/config` ファイル（存在しない場合は作成する）を次のように編集する．

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

これはホスト名のエイリアス `hg` を定義する．このホスト名を `ssh` のコマンドラインまたは Mercurial `ssh`-プロトコルでの URL で使用すると，`ssh` コマンドは `hg.example.com` への接続に圧縮を用いる．この方法で短縮形のホスト名と圧縮の設定を同時に行うことができる．

6.6 CGI を使用した HTTP によるサービス

一つまたは複数のリポジトリを永続的にサービスする最も簡単な方法は，ウェブサーバで Mercurial の CGI サポートを利用することである．

Mercurial の CGI インタフェースの設定にかかる時間は，目的とするところに応じて数分から数時間程度の幅がある．

ここでは最も単純な例から始めて，より複雑な設定へ進んでいくことにしよう．最も単純なケースでもおそらくウェブサーバの設定を調べて変更する必要があるだろう．

> **Note:** High pain tolerance required
> ウェブサーバの設定は複雑かつ厄介でシステム依存の作業である．読者が遭遇するすべてのケースについて手引きをすることはできないだろう．以下のセクションでは，読みながら読者自身の思慮と判断を働かせて欲しい．たくさんのミスを犯し，サーバのエラーログの解読に多くの時間を割くことになるのを肝に銘じておいて欲しい．
> 延々と設定の手直しを続けることが我慢できなかったり，自前のサービスが必須でなかったりする場合は，前述の公共ホスティングサービスを検討した方が良いだろう．

6.6.1 Web サーバ設定のチェックリスト

ここから先へ進む前に，読者のシステムについて確認をしておこう．

1. ウェブサーバはすでにインストールされているか？ Mac OS X やいくつかの Linux ディストリビューションでは標準で Apache がインストールされているが，ウェブサーバがインストールされていないシステムも多くある．
2. ウェブサーバがインストールされている場合，すでに動作しているか？多くのシステムでは，デフォルトで停止されている．

3. ウェブサーバは CGI プログラムを目的のディレクトリで動作できるように設定されているか？ 大半のサーバではデフォルトで CGI プログラムの動作を明示的に禁じている．

ウェブサーバがインストールされていない場合や，Apach の設定に十分な経験がない場合は，Apache ではなく `lighttpd` ウェブサーバを検討した方がよいだろう．Apache は奇異で混乱する設定で悪名が高い． `lighttpd` は Apache よりもできることが少ないが，それらは Mercurial リポジトリのサービスとは関係がない． `lighttpd` は Apache よりも明らかに簡単に使うことができる．

6.6.2 CGI の基本的な設定

Unix 系システムでは，ユーザのホームディレクトリにウェブページをサービスするための `public_html` というサブディレクトリがあることが普通である．このディレクトリ内の `foo` というファイルは `http://www.example.com/~username/foo` という URL でアクセスできる．

まずインストールされている Mercurial から `hgweb.cgi` スクリプトを見つける．すぐにローカルコピーが見つからなければ，Mercurial のマスターリポジトリ http://www.selenic.com/repo/hg/raw-file/tip/hgweb.cgi からダウンロードする．

このスクリプトを `public_html` ディレクトリにコピーし，ファイルが実行可能であるかをチェックする．

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

`chmod` コマンドへ `755` を渡すと，スクリプトは実行可能よりも若干一般的になる．スクリプトは誰からも実行可能だが，“group” と “other” のユーザからは書き込めなくなる．書き込みパーミッションが有効であると，Apache の `suexec` サブシステムはスクリプトの実行を拒否する可能性が高い．実際のところ，`suexec` はさらにスクリプトの置かれている**ディレクトリ**が他のユーザの書き込みを拒否する設定であることを要求する．

```
chmod 755 ~/public_html
```

どこが問題と成り得るか?

CGI スクリプトを所定の場所にコピーしたら，ウェブブラウザを起動して http://myhostname/~myuser/hgweb.cgi を開く．しかしこの URL にアクセスしてもエラーがでる可能性が高いので，落ち着いて欲しい．エラーには多くの理由が考えられ，実際，そのすべてに引っ掛かっている可能性が高いので，以下の記述を注意深く読んで欲しい．ここに挙げたのは，Fedora 7 で，新規にインストールされた Apache と，この例題のために新規に作成したユーザアカウントで筆者が実際に遭遇した問題である．

ウェブサーバはユーザ毎のディレクトリサービスを禁止されているかもしれない．Apache を使っている場合，設定ファイルの `UserDir` ディレクティブをチェックする．もし存在しなければ，ユーザ毎のディレクトリサービスは禁止されている．存在しても，値が `disabled` に設定されていれば，ユーザ毎のディレクトリサービスは禁止である．また，`UserDir` ディレクティブは Apache がサービス用に探すホームディレクトリ内のサブディレクトリを指定する．典型的な例は `public_html` である．

ファイルアクセスパーミッションがきつすぎる．ウェブサーバはホームディレクトリと `public_html` 内のディレクトリを渡ってファイルを読めなければならない．パーミッションを適切に設定するには例えば次のようにすればよい．

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

パーミッションに関するその他の可能性として，スクリプトをロードしようとすると空のウィンドウが表示される問題がある．この場合は，パーミッション設定が**緩すぎる**可能性が高い．例えば Apache の `suexec` サブシステムは，グループや全世界から書き込みのできるスクリプトを実行しない．

ウェブサーバは CGI プログラムの実行を禁止するように設定されているかもしれない．著者の Fedora システムから，Apache のユーザ毎のデフォルト設定を例として示す．

```
<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>
```

Apache 設定の中に同様の Directory グループがある場合，その中で注目すべきディレクティブは Options である． ExecCGI がリストに無ければ最後に追加し，ウェブサーバを再起動する．

Apache が CGI スクリプトを実行するのではなく，スクリプト自体のテキストを送ってくる場合は，以下のディレクティブを追加する．ディレクティブがすでにあり，コメントアウトされているのなら，有効化する．

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

次の可能性は mercurial に関連した module がインポートできないことを警告するカラフルな Python バックトレースが見えることだ．これは前進と言える．サーバは CGI スクリプトを実行できるようになった．このエラーは Mercurial をシステムワイドにインストールするのではなく，プライベートインストールした場合にのみ起きる．対話的なセッションとは異なり，ウェブサーバは CGI プログラムを環境変数なしで起動する．このエラーが起きた場合は hgweb.cgi を編集し，環境変数 PYTHONPATH がセットされるようにする．

今度は**おそらく**カラフルな Python のバックトレースが見えるはずだ．これは/path/to/repository が見つからないことを警告している．hgweb.cgi スクリプトを編集し，/path/to/repository という文字列を，サービスしたいリポジトリへの完全なパスに置き換える．

ここでページをリロードすると，リポジトリの履歴を表す美しい HTML が見えるはずだ．やった！

lighttpd の設定

筆者の実験の中では，Apache を使ってサービスしたのと同じリポジトリを人気を獲得しつつあるウェブサーバである lighttpd を用いてサービスすることも試みた． Apache に関する全ての問題点を既に解決していたが，その中の多くはサーバ特有というわけではなかった．結果として，ファイルとディレクトリのパーミッション設定が正しいこと， hgweb.cgi スクリプトが正しく設定されていることを確信した．

すでに Apache を使っていたので，lighttpd でリポジトリのサービスを行うことはたやすかった．（これは lighttpd を使ってサービスを試みる場合も Apache のセクションを読む必要があるということでもある．）筆者のシステムでは mod_cgi と mod_userdir が共に無効に設定されていたので，これらを有効にするために，まず設定ファイルの mod_access セクションを編集する必要があった．その後で，下記のモジュールを有効にするために，設定ファイルの末尾に数行を追加した．

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

これらの設定をするだけで lighttpd はすぐに動作した． Apache より前に lighttpd を試していたら，Apache で直面したような様々なシステムレベルの設定問題に遭遇していたに違いないが，初めて使う lighttpd の設定の方が，これまで 10 年以上にわたって使ってきた Apache のそれよりも明らかに簡単であることが分かった．

6.6.3 1 つの CGI スクリプトで複数のリポジトリを共有する

hgweb.cgi スクリプトには，1 つのリポジトリしか公開できないという厄介な制限がある．2 つ以上のリポジトリを公開したい場合は，同じスクリプトを別の名前でいくつも動かすのではなく，hgwebdir.cgi スクリプトを使うのが良いだろう．

hgwebdir.cgi の設定の手順は hgweb.cgi よりもわずかに込み入っているだけだ．手近なところにスクリプトがなければ，Mercurial のマスターリポジトリ http://www.selenic.com/repo/hg/raw-file/tip/hgwebdir.cgi からスクリプトを入手できる．

このスクリプトを public_html ディレクトリにコピーし，実行パーミッションを与える．

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

通常の設定の場合，ブラウザで http://myhostname/~myuser/hgwebdir.cgi を開くと，中身が空のリポジトリを表示するはずだ．ウィンドウ自体が空だったり，エラーメッセージが表示される場合は，セクション 6.6.2 の問題リストを参照してほしい．

hgwebdir.cgi は外部の設定ファイルを使用している．デフォルトでは同じディレクトリ内の hgweb.config というファイルを参照する．このファイルを作成し，全てのユーザから読めるように設定する．このファイルは Python の ConfigParser [?] で処理できるよう Windows の “ini” ファイルと似た形式になっている．

hgwebdir.cgi の最も簡単な設定方法は，collections セクションを編集することである．これは自動的に指定したディレクトリ以下の**全て**のリポジトリを公開する．このセクションは次のようになっている：

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

Mercurial は “=” 記号の**右側**のディレクトリ名を参照して，ディレクトリ階層内のリポジトリを探す．見つかった時は，リポジトリのパスから**左側**の文字列とマッチする部分を削り，実際にウェブインタフェースに表示されるパス文字列を作る．削除後のパスを “仮想パス” と呼ぶ．

上の例で，CGI スクリプトは，ローカルパスが/my/root/this/repo であるリポジトリに対して/my/root を取り除き，this/repo という仮想パスを作成し，公開する．もし CGI スクリプトのベース URL が http://myhostname/~myuser/hgwebdir.cgi だとすると，リポジトリの完全な URL は http://myhostname/~myuser/hgwebdir.cgi/this/repo となる．

この例の左辺の/my/root を/my で置き換えると hgwebdir.cgi は/my だけをリポジトリ名から取り除き，仮想パスとして this/repo ではなく root/this/repo を作る．

hgwebdir.cgi スクリプトは，設定ファイルの collections セクションに書かれたディレクトリを再帰的にサーチする．このスクリプトは見つけたリポジトリの中はサーチしない．

collections メカニズムによって複数のリポジトリを簡単に公開することができる．CGI スクリプトと設定ファイルを編集するのは最初の一回だけでよく，リポジトリを hgwebdir.cgi の探索するディレクトリ階層内に移動すれば公開に，階層内から外せば非公開に設定できる．

どのリポジトリを表示するか明示的に指定する

hgwebdir.cgi スクリプトは collections のメカニズムの他にもリポジトリの特定のリストを公開する方法を用意している．下記のような内容の paths セクションを作る．

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

この場合，各々の定義の左辺に仮想パス（URL に現れる要素）が，リポジトリへのパスが右辺に現れる．選んだ仮想パスとファイルシステム中での位置の間にはいかなる関連性も必要ない．

collections と paths の両方の機構を同一の設定ファイル内で同時に用いることもできる．

Note: 仮想パスの重複に注意
複数のリポジトリが同じ仮想パスを持つ場合でも `hgwebdir.cgi` はエラーを表示しないが，挙動は予測のつかないものとなる．

6.6.4 ソースアーカイブのダウンロード

ユーザは Mercurial のウェブインタフェースからどのリビジョンのアーカイブもダウンロードすることができる．アーカイブは当該リビジョンのワーキングディレクトリのスナップショットを含むが，リポジトリ自体のデータは含まない．

デフォルトではこの機能は無効にされている．有効にするには `allow_archive` 項目を `hgrc` の `[web]` セクションに追加する必要がある．(詳細については下記を参照.)

6.6.5 Web 設定オプション

Mercurial ウェブインタフェース (the "`hg serve`" コマンド，`hgweb.cgi` および `hgwebdir.cgi` スクリプト) には多くの設定オプションがある．これらは `[web]` セクションに含まれる．

`allow_archive` どのアーカイブダウンロードメカニズムを Mercurial がサポートするのかを決定する．この機能を有効にすると，ウェブインタフェースのユーザはリポジトリの任意のリビジョンのアーカイブをダウンロードできるようになる．アーカイブ昨日を有効にするにはこの項目は，下のリストに示す語から構成されなければならない．

- `bz2` bzip2 圧縮された `tar` アーカイブ．圧縮率が最も高いが，サーバの CPU 時間も一番使用する．
- `gz` gzip 圧縮された `tar` アーカイブ．
- `zip` LZW 圧縮された `zip` アーカイブ．この中で圧縮率は最低だが，Windows の環境では広く用いられている．

空のリストを与えるか，`allow_archive` を記述しなければ，この昨日は無効化される．サポートされている 3 つのフォーマット全てを有効にする例を示す．

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

`allowpull` ブール値．リモートユーザにウェブインタフェースを用いた HTTP による "`hg pull`" 及び "`hg clone`" を許可するかどうかを決める．`no` または `false` の場合，ウェブインタフェースはブラウザによる閲覧のみが可能になる．

`contact` 文字列．自由形式（簡潔な表記が好ましい）でリポジトリを担当する人物やグループを記述する．通常は人名と，個人またはメーリングリストのアドレスを含む．多くの場合，`.hg/hgrc` にこの記述を置くのが良い．もし全てのリポジトリを同一人物が管理するのであれば，グローバルな `hgrc` に置くのもよい．

`maxchanges` 整数．1 ページに表示するチェンジセットのデフォルトの最大数．

`maxfiles` 整数．1 ページに表示する更新されたファイルのデフォルトの最大数．

`stripes` 整数．表の表示を行う際に，行を見易くするためウェブインタフェースが交互に "ストライプ" 表示する場合，この値でそれぞれのストライプの行数を設定する．

`style` Mercurial がウェブインターフェースを表示するために使用するテンプレートを制御する．Mercurial にはいくつかのウェブテンプレートが同梱されている．

- `coal` 単色のテンプレート．
- `gitweb` git のウェブインターフェースを模倣したデザインのもの．

- monoblue 単色の青とグレー.
- paper デフォルト
- spartan これまで長らくデフォルトとして使われてきたもの.

独自のテンプレートを指定することも可能である．詳細は 11を参照のこと．ここでは gitweb スタイルを有効にする方法を示す.

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

templates パス．テンプレートファイルを検索するディレクトリ．デフォルトでは Mercurial はインストールされたディレクトリからテンプレートを探す.

hgwebdir.cgi を利用する場合，便利のため，hgrc ファイルではなく hgweb.config ファイルの [web] セクションに motd および style 項目を置くことができる.

個々のリポジトリに特有のオプション

いくつかの [web] 設定項目はユーザやグローバルの hgrc ではなく，リポジトリローカルの.hg/hgrc ファイルに書かれるのが自然である.

description 文字列．リポジトリの内容や目的についての説明．形式は自由だが，簡潔なものが好まれる.

name 文字列．ウェブインタフェースでのリポジトリ名．この名前はリポジトリパス中の最後の要素から作られるデフォルト名をオーバライドする.

“hg serve” コマンド特有のオプション

hgrc ファイルの [web] セクションの項目は “hg serve” コマンドでのみ用いられる.

accesslog パス．アクセスログを出力するファイルの名前．デフォルトでは “hg serve” コマンドはファイルではなく標準出力へ出力を行う．ログエントリはほとんどのウェブサーバで標準のコンバインドファイル形式で行われる.

address 文字列．接続に対してサーバが待機するローカルアドレス．デフォルトでは全てのアドレスに対して待機する.

errorlog パス．エラーを記録するファイルの名前．デフォルトでは “hg serve” コマンドはファイルではなく標準エラー出力へ出力を行う.

ipv6 ブール値．IPv6 プロトコル利用の有無．デフォルトは IPv6 不使用.

port 整数．サーバが待機する TCP ポート番号．デフォルトは 8000 番.

[web] アイテムを追加する正しい hgrc ファイルを選ぶ

Apache や lighttpd のようなウェブサーバは独自のユーザ ID で動作することに留意する必要がある．hgweb.cgi のような CGI スクリプトは通常サーバがサーバのユーザ ID で動作させる.

[web] アイテムをユーザ個人の hgrc ファイルに追加しても CGI スクリプトはその hgrc を参照しない．これらの設定は，ユーザが起動する “hg serve” コマンドにのみ影響を与える．あなたが行った設定を CGI スクリプトから参照させるためには，ウェブサーバを起動するユーザ ID のホームディレクトリに hgrc ファイルを作り，設定をシステムの hgrc ファイルにも追加する必要がある.

6.7 システムワイドの設定

複数のユーザが使用する Unix 系のシステム（ユーザが変更を公開するサーバなど）では，ウェブインタフェースで使用するテーマのように，システム全体でのデフォルトの挙動を定義するとよい場合がある．

`/etc/mercurial/hgrc` というファイルがあると，Mercurial は起動時にこれを読み，全ての設定を適用する．また，`/etc/mercurial/hgrc.d` ディレクトリ内のファイル名が `.rc` で終るファイルを探し，書かれた設定を適用する．

6.7.1 Mercurial の信頼性を上げる

システム全体の `hgrc` ファイルが有用な場合の一例に，他のユーザが所有するリポジトリから pull する場合がある．デフォルトでは Mercurial は別のユーザの所有するリポジトリ内にある `.hg/hgrc` ファイルのほとんどの項目を信頼しない．そのようなリポジトリからクローンや変更の pull を行うと，Mercurial は `.hg/hgrc` を信頼しないという警告を表示する．

Unix で特定のグループに入っているユーザ全てが同じチームに属し，互いに他のユーザの設定を信頼**すべき**場合や，特定のユーザたちの設定を信頼すべき場合は，次のようなシステム全体の `hgrc` ファイルを作成し，Mercurial の懐疑的な設定をオーバライドすることができる．

```
この内容を /etc/mercurial/hgrc.d/trust.rc などとして保存する
rusted]
所有者が "editors" または "www-data" である hgrc ファイルを信頼する
oups = editors, www-data

次のユーザの所有する hgrc ファイルのエントリを信頼する
ers = apache, bobo
```

Chapter 7

ファイル名とパターンマッチング

Mercurial はファイル名について一貫性のあり分かりやすいメカニズムを提供する．

7.1 シンプルなファイル命名

Mercurial はファイル名を取り扱う内部のメカニズムを持っている．あらゆるコマンドはファイル名に対して同一にふるまう．コマンドがファイル名を扱うやり方を以下に示す．

コマンドラインに実際のファイル名を明示的に与えた時は，Mercurial は与えられたファイルだけを扱う．

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

ディレクトリ名を与えた場合は，Mercurial は“このディレクトリとサブディレクトリ内のすべてのファイルに対して処理を行う”と解釈する．Mercurial はディレクトリ内のファイルとサブディレクトリをアルファベット順に渡り歩く．サブディレクトリを見つけると，カレントディレクトリの処理を続けるのではなく，サブディレクトリ内を見に行く．

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

7.2 ファイル名なしでコマンドを実行する

ファイル名を取る Mercurial コマンドはファイル名やパターンを与えないで起動した場合でも有用なデフォルトの振舞を持つ．期待する挙動は，何をするコマンドかによる．コマンドにファイル名等を与えなかった場合にコマンドがどのように動くか推測する大まかなルールをいくつか挙げる．

- 大半のコマンドはワーキングディレクトリ全体に対して働く．例えば“`hg add`”コマンドのふるまいがこれにあたる．
- 復元が困難であったり不可能であるような効果を持つコマンドの場合，最低 1 つの名前やパターンを明示することを求める（下記を参照.）．こうすることで，例えば“`hg remove`”に引数を与えなかったためにファイルすべてを誤って消すことがなくなる．

デフォルトの挙動が気に入らない場合，これを変更するのはたやすい．コマンドが通常ワーキングディレクトリ全体に対して動作するとしよう．これをカレントディレクトリとそのサブディレクトリに対してのみ動作するように変えるには，“.”を渡せば良い．

```
$ 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
```

サブディレクトリから起動してもファイル名をリポジトリのルートからの相対パスで表示するコマンドがある．そのようなコマンドに明示的にサブディレクトリ名を与えると，現在のサブディレクトリからの相対パスが表示される．ここで，サブディレクトリ内から“hg status”コマンドに“hg root”コマンドの出力を引数として与えて起動し，“hg status”コマンドがワーキングディレクトリ内のファイルを現在のサブディレクトリに対する相対パスで表示する様子を見てみよう．

```
$ 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 何が起きているのか

前セクションの“hg add”の例はMercurialコマンドについて別の有用な情報を示している．コマンドラインで明示的に名前を指定しなかったファイルに対してコマンド処理を行う場合，ファイル名を表示し，何が起こっているのか分からなくならないようにしている．

できるだけびっくりさせないというのがここでの原則である．コマンドラインでファイル名を完全に指定した場合，ファイル名が表示されることはない．Mercurial は，名前を与えないか，ディレクトリ名や，以下で説明するようなパターンを与え，ファイルが**暗黙的に**指定された場合は，安全のために現在操作しているファイル名を表示する．

このように振舞うコマンドを，`-q` オプションを与えることで沈黙させることもできる．逆に明示的に指定したファイルへの動作であっても`-v` オプションを与えることですべてのファイル名を表示させることもできる．

7.4 ファイル名識別にパターンを用いる

ファイルやディレクトリ名を使った動作の他に，Mercurial では *patterns* を使ってファイルを識別することもできる．Mercurial のパターン処理は強力である．

Unix 系のシステム (Linux, MacOS 等) ではファイル名とパターンをマッチさせる仕事はシェルに任される．これらのシステムでは，名前がパターンであることを明示的に Mercurial に示さなければならない． Windows ではシェルはパターンを展開しないので，Mercurial は与えられた名前が自動的にパターンであることを認識し，展開する．

コマンドラインで通常の名前の代わりにパターンを渡すためのメカニズムはシンプルである．:

```
syntax:patternbody
```

どのようなパターンなのかを識別するための短いテキストの後ろにコロンを挟んで実際のパターンが続く．

Mercurial は 2 通りのパターン構文をサポートする．最もよく使われるのは `glob`; で，Unix シェルがパターンマッチングに使用しているものと同様のパターンであり，これは Windows のコマンドプロンプトユーザにとっても親しみ深いものである．

Mercurial は Windows では自動的にパターンマッチングを行うときは，`glob` 構文を用いる．従って Windows では “`glob:`” プレフィクスを省略しても安全である．

`re` 構文はより強力で，正規表現を用いることができる．

後で示す例では，Mercurial が見る前にシェルで展開されるのを防ぐためにパターンを引用文字でくるんでいることに注意されたい．

7.4.1 シェル形式の `glob` パターン

glob パターンでマッチングを行う時に使えるパターンの概略を示す．

“`*`” 文字はディレクトリ内の任意の文字列とマッチする．

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

“`**`” パターンはディレクトリを越えて任意の文字列にマッチする．これは Unix 標準の glob トークンではないが，いくつかの人気のある Unix シェルで使うことができ，とても有用である．

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

“`?`” パターンは任意の 1 文字にマッチする．

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

“[” 文字は**文字クラス**を開始する．これはクラス内の任意の 1 文字にマッチする．クラスは “]” 文字で終る．クラスは “a-f” のような範囲を複数持つことができる．この範囲は “abcdef” の短縮形に相当する．

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

文字クラスで “[” の後に “!” が来た場合，これはクラスの否定となり，クラスに含まれない任意の 1 文字とマッチする．

“{” はサブパターンのグループを開始する．グループでは，グループ内の任意のサブパターンがマッチすればグループ全体がマッチしたことになる．“,” 文字はサブパターンを分離し，“}” はグループを終了する．

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

ここに注意！

任意のディレクトリでパターンをマッチさせる場合，“*” を全てとマッチするトークンとして使うことはできない．この文字は 1 つのディレクトリ内でのみマッチする．その代わり，“**” トークンを使う．これらの違いを説明するために例を示す．

```
$ 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 `re` パターンを使った正規表現マッチ

Mercurial は Python 言語と同じ正規表現構文を受け付ける．（Mercurial は内部で Python の正規表現エンジンを使っている．）これは Perl の regexp 構文を元にしている．この構文は最もよく用いられているものであり，たとえば Java でも利用されている．

regexp がそれほど使われていないかのようにここで Mercurial の regexp 構文について議論することは避ける． Perl スタイル regexp はすでに多くのウェブサイトや書籍でで詳細に渡って説明されている．その代わり，ここでは Mercurial で regexp を使う際に知っておくべきいくつかの点に焦点を当てることにする．

regexp はファイル名全体とマッチするが，ファイル名はリポジトリのルートからの相対パスで表される．言い替えれば，すでにサブディレクトリ foo にいるとして，このディレクトリ内のファイルにマッチさせたければ，“foo/” で始まるパターンを渡す必要がある．

Perl 形式の regexp に慣れているのなら，Mercurial の regexp は *root* を持つという点に注意しておくとよい．すなわち，regexp は文字列の始まりからマッチし，文字列の途中からはマッチしない．文字列の途中からマッチさせたい場合は，パターンを “.*” で始める必要がある．

7.5 ファイルをフィルタする

Mercurial はファイルを指定する様々な方法を提供するだけでなく，さらに**フィルタ**によってファイルを選別する方法を提供する．ファイル名を取って動作するコマンドは 2 つのフィルタオプションを受け付ける．

- -I または--include オプションで指定したパターンにマッチしたファイルが処理される．
- -X または--exclude オプションで指定したパターンにマッチしたファイルは処理から除外される．

コマンドラインで複数の-I および-X オプションを指定し，所望の組合せにすることができる． Mercurial はデフォルトでは glob 構文で与えたパターンを解釈するが，正規表現を使うことも可能である．

-I フィルタは“このフィルタにマッチするファイルだけを処理する” と読み替えることができる．

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

-X フィルタは“このパターンにマッチしないファイルだけを処理する” と読み替えるのが最も相応しい．

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

7.6 不必要なファイルやディレクトリを永久的に無視する

新しいリポジトリを作成し，開発作業を続けると，時間の経過とともにビルド生成物などのようにリビジョン管理システムで管理すべきでないファイルがリポジトリに多く含まれるようになる．Mercurial の管理外にあるこれらのファイルは，“hg status” を実行するといちいち画面に表示されてしまう．最も典型的なビルド生成物は，コンパイラなどのツールで生成される出力ファイルである．その他には，多くのテキストエディタがディレクトリに置くロックファイルや，一時的なワーキングファイル，バックアップファイルなどがある．

Mercurial にこれらを永久的に無視させるためには，リポジトリのルートディレクトリに.hgignore という名前のファイルを作成する．このファイルはおそらく他の協力者達にとっても有用であるから，リポジトリの他の残りの部分と同様に管理されるよう，“hg add” すべきである．

デフォルトでは.hgignore ファイルは 1 行に 1 つずつ正規表現のリストを受け付けるようになっている．空行は無視される．殆んどのユーザは無視すべきファイルを前述の “glob” 構文を用いて記述することを好むので，.hgignore の最初は次のようなディレクティブで始める．

```
ntax: glob
```

これは後続の行を正規表現ではなく glob パターンとして解釈するよう Mercurial に指示する．

典型的な.hgignore ファイルの例を示す．

```
ntax: glob
 This line is a comment, and will be skipped.
 Empty lines are skipped too.
この行はコメントで，スキップされる．
空行も同様にスキップされる．

 Backup files left behind by the Emacs editor.
Emacs エディタによって残されるバックアップファイル．


 Lock files used by the Emacs editor.
 Notice that the "#" character is quoted with a backslash.
 This prevents it from being interpreted as starting a comment.
Emacs エディタが使用するロックファイル
"#"文字がバックスラッシュでエスケープされていることに注意．
```

```
これによってコメントの開始文字として解釈されるのを防いでいる．
*

 Temporary files used by the vim editor.
vim エディタが使用する一時ファイル
.swp

 A hidden file created by the Mac OS X Finder.
Mac OS X のファインダーによって作られる隠しファイル
S_Store
```

7.7 大文字小文字の影響

Linux をはじめとする Unix や Mac，Windows が混在する開発環境で作業をしているのなら，各々のシステムはファイル名の (“N” と “n”) のような大小文字を異なったやり方で取り扱うことに留意すべきである．これが問題になることは滅多になく，問題になる場合でも簡単に解決できるが，システム間での取り扱いの違いを知らなければ驚くことになるだろう．

オペレーティングシステムとファイルシステムは，ファイル名とディレクトリ名の**大小文字**の扱いで異なっている．大小文字を扱う方法は 3 通りある．

- 完全に大小文字を区別しない．大文字と小文字は，ファイルの作成とその後のアクセスで同一に扱われる．これは古い DOS ベースのシステムで用いられていた．
- 大小文字の違いは保存されるが区別されない．ファイルやディレクトリが作られると，オペレーティングシステムは名前の大小文字を保存し，表示などに使用する．既存のファイルが参照される時は，大小文字は無視される．これは Windows と MacOS で標準的な取り扱いである．`foo` と `FoO` は同一のファイルと認識される．このような大小文字の取り扱いは *case folding* と呼ばれることもある．
- 大小文字を区別する．常に大小文字で名前を区別する．`foo` と FoO は別のファイルと認識される．これは Linux や Unix システムの通常の取り扱いである．

Unix 系のシステムでは同時に上記の方法を利用することが可能である．例えば USB メモリを Linux で FAT32 ファイルシステムとしてフォーマットした場合，Linux はそのファイルシステム上のファイル名を大小文字保存だが区別せずに取り扱う．

7.7.1 安全で可搬なリポジトリストレージ

Mercurial のリポジトリ格納メカニズムは**大小文字セーフ**である．Mercurial は大小文字の区別をするシステムでもしないシステムでも安全に保存できるようにファイル名を変換する．このため，通常のファイルコピーツールを使って Mercurial リポジトリを USB メモリに転送し，Mac，Windows の動く PC，Linux マシンの間で移動しても安全である．

7.7.2 大文字小文字の衝突を検出する

ワーキングディレクトリでの動作の際に， Mercurial はワーキングディレクトリのあるファイルシステムのネーミングポリシーを尊重する．大小文字を保存するが区別しないファイルシステムの場合，Mercurial は名前の大小だけが異なっているファイルを同じファイルとして取り扱う．

このアプローチで重要なのは，大小を区別するファイルシステム（典型的には Linux および Unix）で大小を区別しないシステム（通常，Windows または MacOS）のユーザにとってトラブルとなる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という点である．もし Linux ユーザが `myfile.c` というファイルと `MyFile.C` というファイルに対するチェンジセットをコミットした場合，それらはリポジトリに正しく保存され，他の Linux ユーザのワーキングディレクトリでも正しく別々のファイルとして現れる．

ここで Windows または Mac のユーザがこのチェンジセットを pull すると，Mercurial のリポジトリ格納メカニズムは大小文字に対して安全なため，最初のうちは問題とならない．しかしワーキングディレクトリをそのチェンジセットに“hg update”しようとしたり，そのチェンジセットと“hg merge”しようとすると，Mercurial は， 2 つのファイル名をファイルシステムが同じ名前として扱うために生じるコンフリクトを検出し，“hg update”や“hg merge”を許さない．

7.7.3 大文字小文字の衝突を解決する

もしあなたが Windows や Mac を使っていて，何人かの協力者が Linux または Unix を使っている混合環境で開発をしており，“hg update”または“hg merge”で Mercurial が大小文字のコンフリクトを検出するのであれば，解決方法はシンプルである．

手近な Linux または Unix マシンの上に，問題のあるリポジトリをクローンし，Mercurial の“hg rename”を実行して問題のあるファイルやディレクトリの名前を変更し，コンフリクトを解決する．この変更をコミットし，“hg pull”または“hg push”コマンドであなたの Windows または MacOS システムに転送し，“hg update”でコンフリクトのないリビジョンへ更新すればよい．

大小文字のファイル名コンフリクトがあるチェンジセットはプロジェクトの履歴に残っており， Windows や MacOS システムではそのチェンジセットへ“hg update”することはできないものの，問題なく開発を続けることができる．

Chapter 8

リリースとブランチ開発の管理

Mercurial は同時に複数の局面で進行していくプロジェクトを管理するのに役立つ機能を持っている．これらの機能を理解するため，まず通常のソフトウェアプロジェクトの構造を考える．

多くのソフトウェアプロジェクトでは，新機能を持つメジャーリリースを定期的にリリースする．並行して多数のマイナーリリースも行われる．これらはメジャーリリースのバグを修正したものである．

この章ではまずリリースのようなプロジェクトのマイルストーンに言及する．次いでプロジェクトの各フェーズでの流れを説明し，Mercurial でどのように区分けし，管理できるのかを説明する．

8.1 リビジョンに永続的な名前を付ける

あるリビジョンをリリースと決めた時に，これを記録しておくのは良い考えである．これは後日，バグの再現やソフトウェアの移植などの目的でリリースを再現するのに役立つ．

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

特定のリビジョンに Mercurial で永続的な名前を付けることができる．この名前はタグと呼ばれる．

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

タグは実のところリビジョンに付けられたシンボル名に他ならない．タグは単にユーザの便宜のために付けられる．タグは特定のリビジョンを参照するための手軽で永続的な方法で，Mercurial はタグを翻訳しない．タグの名前には，明確にパースするために必要ないくつかのもの以外に制限はない．タグネームは以下の文字を含むことはできない．

- コロン (ASCII 58, ":")
- 復帰文字 (ASCII 13, "\r")
- 改行文字 (ASCII 10, "\n")

"hg tags" コマンドでリポジトリに存在するタグを表示することができる．出力では，タグ付けされたリビジョンは名前，リビジョン番号，固有のハッシュ値の順に区別される．

```
$ hg tags
tip                                1:f94804064dd0
v1.0                               0:cc7bf5e7c1d6
```

“hg tags” の出力に tip が含まれていることに注意．tip タグは，リポジトリの中で常に最新のリビジョンに付けられている特別なフローティングタグである．

“hg tags” コマンドの出力において，タグはリビジョン番号によって逆順で表示されている．これによって，通常，新しいタグが古いタグよりも前に表示される．tip タグは “hg tags” コマンドの出力の一番先頭に表示される．

“hg log” コマンドはタグの結びつけられたリビジョンを表示する際にタグも表示する．

```
$ hg log
changeset:   1:f94804064dd0
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:15 2009 +0000
summary:     Added tag v1.0 for changeset cc7bf5e7c1d6

changeset:   0:cc7bf5e7c1d6
tag:         v1.0
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:15 2009 +0000
summary:     Initial commit

```

Mecurial コマンドにリビジョン番号を渡す必要がある場合，常にタグネームを使うことができる．内部的には Mercurial はタグネームを対応するリビジョン ID に変換して使用している．

```
$ echo goodbye > myfile2
$ hg commit -A -m 'Second commit'
adding myfile2
$ hg log -r v1.0
changeset:   0:cc7bf5e7c1d6
tag:         v1.0
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:15 2009 +0000
summary:     Initial commit

```

リポジトリの中で使えるタグ数，一つのリビジョンに付けられるタグ数に上限はない．事情はプロジェクトによって異なるだろうが，実用的にはタグを多く付けすぎることはあまり良い考えとは言えない．タグとはリビジョンを見つけやすくするために使うものだからだ．タグを多く付けすぎると，タグによってリビジョンの区別をすることがとたんに難しくなる．

例えばあなたのプロジェクトが数日毎にマイルストーンを迎えている場合，それぞれにタグを付けることは理に適っている．しかし，それぞれのリビジョンがクリーンにビルドできるか検証するためのビルドシステムを持っているような場合，それぞれのクリーンビルドにタグを付けていたら収拾がつかなくなるだろう．むしろビルドが失敗することが少ないのであれば，失敗したリビジョンにタグをつけた方が良いし，単にビルドが通ったことを示すのにタグを用いるべきではないかも知れない．

タグが必要なくなった時は “hg tag --remove” コマンドで消すことができる．

```
$ hg tag --remove v1.0
$ hg tags
tip                                3:982e9eceb3af
```

タグはいつでも変更できるので，あるタグを他のリビジョンに付け替えるようなこともできる．タグを**本当**に更新したい場合は-f オプションを指定する必要がある．

```
$ hg tag -r 1 v1.1
$ hg tags
tip                                5:0e804f04bfea
v1.1                               1:f94804064dd0
$ 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                                5:3fb75c429bd4
v1.1                               2:b746f7790c50
```

タグの以前のアイデンティティーの永続的な記録は残っているが，Mercurial はもはやこれを利用しない．よって，間違ったリビジョンにタグを付けることでペナルティが課せられるということはない．間違いを見つけた時は単にやり直して正しいリビジョンにタグを付ければよい．

Mercurial はタグをリポジトリの中の通常のリビジョン管理ファイルに保存する．タグを作成した時，リポジトリのルートにある.hgtags というファイルにタグが保存されているのが分かるだろう．“hg tag” コマンドを実行すると Mercurial はこのファイルを変更し，変更をこのファイルにコミットする．つまり “hg tag” を実行するといつも “hg log” の出力の中に対応するチェンジセットを見ることになる．

```
$ hg tip
changeset:   5:3fb75c429bd4
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:17 2009 +0000
summary:     Added tag v1.1 for changeset b746f7790c50

```

8.1.1 マージの際にタグのコンフリクトを解決する

.hgtags を気にしなければならないことは多くないが，マージの際には存在を明らかにする．ファイルフォーマットはシンプルで，一連の行を含むだけである．それぞれの行はチェンジセットのハッシュで始まり，1 つのスペースが続き，タグの名称が続く．

マージ中に.hgtags ファイル内のコンフリクトを解決している場合，.hgtags を変更する一捻りがある．Mercurial がリポジトリ中のタグをパースする時，Mercurial は.hgtags のワーキングコピーを**決して**読まない．その代わり**最も新しくコミットされた**リビジョンを読む．

この設計の残念な結果は，変更をコミットした**後**でないとマージした.hgtags ファイルを実際にはベリファイできないことである．そのため，マージ中に.hgtags のコンフリクト解決をしている場合は，コミット後に忘れず “hg tags” を実行する必要がある．.hgtags にエラーが見つかった場合，エラーのある場所を報告する．それを見て修正し，コミットすることができる．そこで “hg tags” を再び走らせ，修正が正しいことを確認すべきである．

8.1.2 タグとクローン

“hg clone” コマンドが-r オプションを持っていることにすでに気づいているかも知れない．このオプションで特定のチェンジセットのコピーをクローンすることができるが、クローンしたコピーは，指定したリビジョン後の履歴を持たないため、不用心なユーザはしばしば驚くことになる。

タグは.hgtags ファイル内のリビジョンとして記録されていることを思い出して欲しい．このため，タグ作成時に，タグが記録されているチェンジセットから古いチェンジセットへの参照が生じる．タグ foo が付いているリポジトリをクローンするために “hg clone -r foo” 実行する時，新しいクローンはタグを生成した履歴を**含まない**．結果として，新しいリポジトリに含まれる履歴は，プロジェクト履歴のサブセットになり，タグは含まれない．

8.1.3 永久タグが必要でない場合

Mercurial のタグはリビジョンコントロールされ，プロジェクトの履歴に付随しているため，同じプロジェクトで働く人は皆タグを知ることになる．リビジョンに名前を付けることは，単にリビジョン `4237e45506ee` がバージョン `v2.0.2` であると記述する以上の意味を持つ．もしバグを追跡しているなら，“Anne saw the symptoms with this revision”（アンはこのリビジョンで症状を見た）などのタグを付けたくなるだろう．

このような場合，**ローカル**タグを使いたくなるはずだ．ローカルタグは “`hg tag`” コマンドを `-l` オプション付きで使うことで作成できる．このオプションを使うと，タグは `.hg/localtags` というファイルに保存される．`.hgtags` と違って，`.hg/localtags` はリビジョンコントロールされない．`-l` オプションで作成したあらゆるタグは厳密にローカルに管理され，今作業しているろポジ取りには一切反映されない．

8.2 更新の流れ—大局的 vs. 局所的

この章の最初で示したアウトラインに戻るために，プロジェクトが開発中，一度に複数の並行した部分を持つと考えよう．

新しい “main” をリリースする準備にかかっている状態で，最新のメインリリースに対する新しい小規模なバグフィックスリリースと，既にメンテナンスモードになっている古いリリース向けの計画外の “hot fix” をリリースする状況を考える．

人々が，異なった並行的な開発の方向性について触れる時は，“ブランチ” として言及する．しかし，既に幾度となく Mercurial が**全ての履歴**を一連のブランチとマージとして取り扱っているのを見てきた．ここでは，わずかに関連しているが名前を共有している 2 つのアイディアを考える．

- “Big picture” ブランチはプロジェクトの進化を表す．開発者はこれらに名前を与え，会話で用いる．
- “Little picture” ブランチは日々の開発とマージの所産で，コードがどのように開発されたかを示すものである．

8.3 リポジトリ間での大局的ブランチの管理

Mercurial で “big picture” を隔離する最も簡単な方法は，専用のリポジトリを用意することである．もし一つの共有リポジトリを持っている場合，これを `myproject` と呼ぶことにする．これが “1.0” マイルストーンに到達したら，1.0 に 1.0 リリースとタグを付け，将来のメンテナンスリリースに備えることができる．

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

そして新たに `myproject-1.0.1` とタグを付けてリポジトリをクローンする．

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

以後，バグフィックスをしたい人は 1.0.1 マイナーリリースへいくべきである．彼らは `myproject-1.0.1` リポジトリをクローンし，変更を加え，プッシュする．

```
$ 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-repoNCePuF/myproject-1.0.1
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

一方で次のメジャーリリースに向けての開発は，myproject リポジトリの中で隔離され，滞る事なく継続することが可能である．

```
$ 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-repoNCePuF/myproject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

8.4 手で繰り返さないこと：ブランチ間でのマージ

メンテナンスブランチで修正すべきバグがあるとき，多くの場合，メインブランチにそのバグがある可能性は高い．（さらに他のメンテナンスブランチにおいても．）バグの修正を何度も繰り返したいと思う開発者はまずいない．そこでバグ修正を繰り返す代わりのではなく，Mercurial で解決できるいくつかの方法を見てみることにしよう．

最も簡単なのは，メンテナンスブランチから変更をターゲットブランチのローカルクローンに pull する方法である．

```
$ cd ..
$ hg clone myproject myproject-merge
updating working directory
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd myproject-merge
$ 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)
```

その後 2 つのブランチのヘッド同士をマージし，メインブランチへプッシュする．

```
$ 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-repoNCePuF/myproject
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 1 changes to 1 files
```

8.5 1つのリポジトリ内でのブランチの命名

多くの場合，ブランチ毎に複数のリポジトリを用意し隔離するのは正しいアプローチである．これは単純なため把握が容易であり，失敗を犯す可能性が低い．作業しているブランチとディレクトリの間には 1 対 1 の関係がある．Mercurial を考慮しない通常のツールをブランチ／リポジトリ内のファイルに対して使うことも可能である．

あなたと（あなたの協力者が**共に**）“パワーユーザ” 以上のカテゴリに属すなら，ブランチを取り扱う別の方法も考えられる．すでに “局所的なモデル” と “大局的なモデル” という，区別について触れた．Mercurial は，複数の “局所的な” ブランチ（たとえば変更を pull してマージしていない場合など）用いることができる一方で，複数の “大局的な” ブランチを用いることも**も**できる．

この方法を用いる際の鍵は，Mercurial によってブランチに永続的な**名前**を付けることである．ブランチに名前を付ける前でも，`default` ブランチのトレースを見ることができる．

例として，“`hg commit`” コマンドを実行し，コミットメッセージを書くためにエディタが起動された時，最下部の “`HG: branch default`” という行を見てほしい．この行はコミットが `default` という名前のブランチに対して行われることを示している．

名前付きブランチを使うにあたって，まず “`hg branches`” を使い，リポジトリ内にすでに存在する名前付きブランチを列挙することから始める．このコマンドによってそれぞれのブランチの tip になっているチェンジセットがわかる．

```
$ hg tip
changeset:   0:81b3558d971c
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:37 2009 +0000
summary:     Initial commit

$ hg branches
default                        0:81b3558d971c
```

ここではまだ名前付きブランチを作っていないので，`default` ブランチだけが存在する．

現在のブランチが何なのかを知るためには，“`hg branch`” コマンドを引数なしで実行する．このコマンドで現在のチェンジセットの親ブランチがわかる．

```
$ hg branch
default
```

新しいブランチを作るのにも “`hg branch`” コマンドを使う．この場合は，作成したいブランチの名前を引数として渡す．

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

ブランチを作った後で “hg branch” コマンドがどのような効果を持っているのか分からないかもしれない．“hg status” コマンドと “hg tip” コマンドはそれぞれ何を表示するだろうか？

```
$ hg status
$ hg tip
changeset:   0:81b3558d971c
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:37 2009 +0000
summary:     Initial commit

```

ワーキングディレクトリ内では何も変化は起きず，何の履歴も生成されていない．これから分かるように，“hg branch” コマンドを実行しても，永続的な効果は何も起きない．このコマンドは**次の**チェンジセットのコミットがどのブランチに対して行われるかを Mercurial コマンドに示すのに使われる．

変更をコミットする時，Mercurial はコミットするの対象になるブランチの名前を記録する．ひとたび default から他へ変更すれば，新しいブランチ名が “hg log” や “hg tip” の出力に含まれるのが見て取れるだろう．

```
$ echo 'hello again' >> myfile
$ hg commit -m 'Second commit'
$ hg tip
changeset:   1:2c47a266518e
branch:      foo
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:38 2009 +0000
summary:     Second commit

```

“hg log” のようなコマンドは，default ブランチ以外に属するすべてのチェンジセットに対してブランチ名を表示する．ブランチに名前を付けていない場合はこの出力を目にすることはない．

ブランチに名前を付け，その名前を使って変更のコミットを行うと，以後に続く全てのコミットは同じ名前を持つ．名前の変更は “hg branch” コマンドを使うことでいつでも可能だ．

```
$ 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:b318fc0f3801
branch:      bar
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
```

```
date:        Tue Jun 09 06:06:38 2009 +0000
summary:     Third commit

```

実用においては，ブランチ名はかなり長い期間使われる傾向があり，変更は頻繁には行われない.（これは法則と言えるようなものではなく，単なる観測結果である.）

8.6 リポジトリ内で複数の名前の付いたブランチの取り扱い

1 つのリポジトリ内に 2 つ以上のブランチを持っている場合，Mercurial は，“hg update” や “hg pull -u” のようなコマンドを立ち上げるときに，ワーキングディレクトリがどのブランチであるか記憶している．これらのコマンドは，リポジトリ全体での tip が何であるかには関係なく，ワーキングディレクトリを現在のブランチの tip に更新する．他の名前つきブランチに属すリビジョンへ更新するには，-C オプション付きで “hg update” コマンドを実行する必要があるかもしれない.

この振舞いはやや奇異かもしれない．実際の動作を見てみることにする．まず我々がどのブランチにいるかを調べ，リポジトリにどんなブランチがあるか見てみよう.

```
$ hg parents
changeset:   2:b318fc0f3801
branch:      bar
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:38 2009 +0000
summary:     Third commit

$ hg branches
bar                            2:b318fc0f3801
foo                            1:2c47a266518e (inactive)
default                        0:81b3558d971c (inactive)
```

今いるのは bar ブランチで，foo ブランチも存在する.

foo と bar の tip の間を “hg update” コマンドで行き来することができる．この操作は更新履歴の中を線形に移動するだけなので，-C オプションは必要ない.

```
$ hg update foo
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg parents
changeset:   1:2c47a266518e
branch:      foo
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:38 2009 +0000
summary:     Second commit

$ hg update bar
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg parents
changeset:   2:b318fc0f3801
branch:      bar
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:38 2009 +0000
```

```
summary:     Third commit
```

foo ブランチへ戻り，“hg update” を実行しても，bar の tip には移動せず，foo のままである．

```
$ 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
```

新しい変更を foo にコミットすると，新しい head が作られる．

```
$ echo something > somefile
$ hg commit -A -m 'New file'
adding somefile
created new head
$ hg heads
changeset:   3:842993b7e874
branch:      foo
tag:         tip
parent:      1:2c47a266518e
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:39 2009 +0000
summary:     New file

changeset:   2:b318fc0f3801
branch:      bar
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:38 2009 +0000
summary:     Third commit
```

8.7 ブランチ名とマージ

おそらく気づいていることと思うが，Mercurial でのマージは対称的ではない．今，リポジトリが 17 と 23 という 2 つの head を持つとしよう．ここで “hg update” で 17 に更新し，“hg merge” によって 23 とマージすると，Mercurial は 17 をマージの最初の親，23 を 2 番目の親として記録する．逆に “hg update” で 23 に更新し，“hg merge” で 17 とマージすれば，23 を最初の親，17 を 2 番目の親と記録する．

これはマージの際に Mercurial がブランチ名をどのように選ぶかに影響を与える．マージ後、マージの結果をコミットする際に Mercurial は 1 番目の親のブランチ名を用いる．1 番目の親のブランチ名が foo で，bar とマージを行ったとすると，マージ後のブランチ名は foo となる．

1 つのリポジトリが同じブランチ名を持ついくつもの head を持っていることは珍しいことではない．今，私とあなたが同じ foo ブランチで作業をしており，それぞれ異なった変更をコミットするとしよう．あなたの行った変更を私が pull すると，私は foo ブランチに 2 つの head を持つことになる．マージの結果は，あなたが期待するように foo ブランチ上で 1 つの head になる．

しかし，私が bar ブランチで作業をしていて，foo ブランチからマージを行うと，成果物は bar ブランチに残ることになる．

```
$ hg branch
bar
$ hg merge foo
```

```
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg commit -m 'Merge'
$ hg tip
changeset:   4:437661ca9cf6
branch:      bar
tag:         tip
parent:      2:b318fc0f3801
parent:      3:842993b7e874
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:40 2009 +0000
summary:     Merge

```

もっと具体的な例を挙げると，もし私が bleeding-edge ブランチで作業をしていて，stable ブランチの新しい修正を取り込むために stable から pull と merge を行うと，Mercurial は“正しい” ブランチ名 (bleeding-edge) を選ぶ.

8.8 ブランチに名前を付けることは役に立つ

名前付きブランチを 1 つのリポジトリの中に長命なブランチを複数持つ状況にのみ用いるべきだと考えるべきではない．名前付きブランチはブランチ毎にリポジトリを持つ場合でも極めて有用である．

最も単純な例は，それぞれのブランチに名前を与えることで，チェンジセットがどのブランチを起源に持つのか永続的に記録することができる．これにより，長命なブランチを持つプロジェクトの歴史を追いかける時に，前後関係が掴みやすくなる．

共有リポジトリを使って作業している場合，pretxnchangegroup フックを設定することで間違ったブランチ名を持つ更新をブロックすることができる．これは例えば“最先端” ブランチから“安定” ブランチへ変更を push するような間違いを防ぐのにシンプルかつ効果的な方法である．フックは，共有リポジトリの hgrc に

```
1 [hooks]
2 pretxnchangegroup.branch = hg heads --template 'branches ' | grep mybranch
```

のように記述される.

Chapter 9

ミスの発見と修正

リビジョンコントロールシステムには人の犯した間違いをうまく処理する機能が求められている．この章では，プロジェクトに起こり得る問題の解決に利用可能なテクニックについて述べる．Mercurial には問題のあるソースを切り分け，処理する強力な機能がある．

9.1 ローカルヒストリーを消去する

9.1.1 アクシデントによるコミット

筆者にはタイピング中に考えるよりも先に指が動いてしまう癖が以前からあり，たまにこれが起きると，不完全だったり間違った内容のチェンジセットをコミットしてしまう．筆者の場合，不完全なチェンジセットの典型は新しいソースファイルを作成したのに “`hg add`” を忘れることで，“間違った内容” のコミットはあまり起きないが，やはり同じように厄介である．

9.1.2 トランザクションのロールバック

4.2.2節で，Mercurial はリポジトリへの個々の変更を**トランザクション**として扱うということを述べた．Mercurial は，別のリポジトリへのチェンジセットのコミットや変更の pull を記憶している．ユーザは undo したり，一回に限りアクションを “`hg rollback`” コマンドで**ロールバック**することができる．(このコマンドの重要な制約については 9.1.4を参照のこと．)

筆者が良く起こすミスは，新しいファイルを作成したチェンジセットをコミットする時に “`hg add`” の実行を忘れるものである．

```
$ hg status
M a
$ echo b > b
$ hg commit -m 'Add file b'
```

コミット後の “`hg status`” の出力を見ると，直ちにエラーを表示していることがわかる．

```
$ hg status
? b
$ hg tip
changeset:   1:df535de6f7bd
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:13 2009 +0000
```

```
summary:     Add file b

```

コミットは a への変更を含んでいるが，b への変更は含んでいない．ここで私が同僚と共有しているリポジトリへチェンジセットのプッシュを行ったとしたら，彼らが変更をプルした時，a の中の何かが彼らのリポジトリに含まれない b への参照を行う可能性は高い．そうなったら筆者は同僚の怒りを買うことになるだろう．

しかし好運にもチェンジセットをプッシュする前にエラーに気づいた場合，“hg rollback” コマンドを使用することで Mercurial から最後の更新を取り除くことができる．

```
$ hg rollback
rolling back last transaction
$ hg tip
changeset:   0:95dd9b5b55ee
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:13 2009 +0000
summary:     First commit

$ hg status
M a
? b
```

チェンジセットはリポジトリの履歴にもはや存在せず，ワーキングディレクトリの a は変更されたと認識される．コミットしてロールバックすると，ワーキングディレクトリは完全にコミット前の状態になり，チェンジセットは完全に消去される．この状態で安全に “hg add” b し，もう一度 commit することができる．

```
$ hg rollback
rolling back last transaction
$ hg tip
changeset:   0:95dd9b5b55ee
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:13 2009 +0000
summary:     First commit

$ hg status
M a
? b
```

9.1.3 誤ったプル

Mercurial を使って別々の開発ブランチを別々のリポジトリで管理することがよくある．あなたの開発チームはプロジェクトのリリース 0.9 のために 1 つの共有リポジトリを持ち，リリース 1.0 のために異なる変更を持ったもう一つ別のリポジトリをもっている．

ここであなたは誤ってローカルの 0.9 リポジトリに共有 1.0 リポジトリからプルしたとする．最悪の場合，注意を怠って，これを共有の 0.9 リポジトリに書き戻してしまい，開発チーム全体を混乱させてしまうことが有り得る．（この場合どうすればいいのかについては後述する．）実際には，Mercurial は pull 元の URL を表示するし，間違ったリポジトリから pull すれば大量のチェンジセットが表示されるため，即座に何かがおかしいと気づくことだろう．

“hg rollback” コマンドを実行すれば，今プルしたばかりのチェンジセットを全て消去することができる．Mercurial は一回 “hg pull” によるトランザクションでもたらされた全てのチェンジセットをグループ化しているので，“hg rollback” を一度実行するだけで，ミスを全てやり直すことができる．

9.1.4 一度プッシュした後ではロールバックできない

他のリポジトリに変更をプッシュした後では“hg rollback”の価値はゼロである．変更をロールバックすることによって変更は完全に消滅するが，それはあなたが“hg rollback”を実行したリポジトリに限ってのことである．ロールバックによって（変更の）履歴自体がなかったことになるので，変更の消滅を他のリポジトリに波及させる手段はない．

もしあなたが他のリポジトリ（特に共有リポジトリ）に変更をプッシュしているのなら，困った事態が起こっており，別の方法でミスから復旧する必要がある．チェンジセットをどこかへプッシュした後でロールバックし，プッシュ先のリポジトリから再びプルした場合，チェンジセットがローカルリポジトリに再び現れる．

（もしロールバックしたいチェンジセットがプッシュ先のリポジトリで最新であり，かつ，誰もそれをプルしていないことが確実である場合は，プッシュ先のリポジトリでロールバックすることが可能だが，この方法は確実ではないということを肝に命じておくべきである．この方法は直接コントロールできなかったり，その方法を忘れてしまったリポジトリへの変更の修正には使えない.）

9.1.5 ロールバックは一回のみ

Mercurial はトランザクションログにそのリポジトリに起こった最も新しいトランザクション一回分のみを記録している．ロールバック一回毎に一つ前のリビジョンに戻るわけではない．

```
$ hg rollback
rolling back last transaction
$ hg rollback
no rollback information available
```

リポジトリ内で一度ロールバックしたら，別のコミットをするかプルをするまでロールバックはできない．

9.2 間違った変更を元に戻す

ファイルに変更を加えた後で，変更が必要でないと分かり，まだコミットされていない時は“hg revert”コマンドを使うことができる．このコマンドはワーキングディレクトリの親チェンジセットを参照し，ファイルの内容をチェンジセットの状態に戻す.（これはあなたが加えた変更をくどくどしく述べたものである.）

“hg revert”コマンドがどのように動作するか，別の小さな例で説明する．すでに Mercurial が管理しているファイルを変更したところから始める．

```
$ cat file
original content
$ echo unwanted change >> file
$ hg diff file
diff -r b80b0a5057fc file
--- a/file        Tue Jun 09 06:06:53 2009 +0000
+++ b/file        Tue Jun 09 06:06:53 2009 +0000
@@ -1,1 +1,2 @@
 original content
+unwanted change
```

この変更が必要でない場合，単にファイルに“hg revert”を実行すればよい．

```
$ hg status
M file
$ hg revert file
$ cat file
original content
```

“hg revert” コマンドは安全のため.orig というファイル名で変更をセーブする．

```
$ hg status
? file.orig
$ cat file.orig
original content
unwanted change
```

“hg revert” コマンドが扱えるケースについてまとめる．より詳しい説明は，後の節で行う．

- ファイルを変更した場合，“hg revert” はファイルを変更される前の状態に戻す．
- “hg add” を実行した場合，“hg revert” は add を取り消すが，ファイル自体はそのまま手を触れずに残す．
- Mercurial を操作せずにファイルを消去していた場合，“hg revert” はファイルを変更前の状態で復元する
- “hg remove” コマンドでファイルを消去していた場合，変更前の状態でファイルを復元する．

9.2.1 ファイル管理のミス

“hg revert” コマンドは単に変更したファイルを戻すだけでなく，“hg add”，“hg remove” といった Mercurial のファイル操作コマンドの結果を取り消すことができる．

“hg add” でファイルを追加した後で，Mercurial に追跡させる必要がないと分かった時，“hg revert” で add を取り消すことができる．“hg revert” はファイルのマークを消すだけで，ファイルの中身は一切変更しないので心配する必要はない．

```
$ echo oops > oops
$ hg add oops
$ hg status oops
A oops
$ hg revert oops
$ hg status
? oops
```

同様に，“hg remove” コマンドでファイルを消去した時，“hg revert” コマンドでワーキングディレクトリの親の内容にファイルを復元することができる．

```
$ hg remove file
$ hg status
R file
$ hg revert file
$ hg status
$ ls file
file
```

また Mercurial を使わずに手で消したファイルについて同じ操作で復元することができる．（Mercurial の用語でこれを missing と呼んでいたことを思い出して欲しい．）

```
$ rm file
$ hg status
! file
$ hg revert file
$ ls file
file
```

“hg copy” コマンドを取り消した場合，コピー先のファイルはワーキングディレクトリに残るが Mercurial からは管理されない．コピー操作自体コピー元ファイルに影響を与えないため，取り消しによってコピー元ファイルが影響を受けることもない．

やや特殊なケース：リネームの取り消し

“hg rename” した後では，少し留意しておくべき点がある．リネーム後に “hg revert” した場合，ここで説明するように，リネームしたファイルの名前を指定するだけでは不十分である．

```
$ hg rename file new-file
$ hg revert new-file
$ hg status
? new-file
```

“hg status” の出力から分かるように，リネーム先のファイルはもはや add された扱いになっていない．しかしリネーム**元**のファイルはまだ消去されたままになっている．これは（少なくとも筆者にとって）非直感的だが，取り扱いは簡単である．

```
$ hg revert file
no changes needed to file
$ hg status
? new-file
```

“hg rename” を取り消す際には，元のファイルとリネーム後のファイル**両方**を指定する必要がある．

（一方，リネームした後にリネーム先のファイルを編集し，リネームの前後のファイル名を指定して取り消しを行い，Mercurial がリネームの際に消去されたファイルを修復すると，このファイルは編集前の状態になっている．リネーム後のファイルへの変更がリネーム前のファイルに残るようにするには，コピーを作っておく必要がある．）

リネームを復元する時に起こる厄介ごとはおそらく Mercurial のバグと言えるかもしれない．

9.3 コミットされた変更の扱い

a をコミットした後で別の *b* をコミットし，ここで *a* は誤りであることに気がついた場合を考える．Mercurial はチェンジセット全体とチェンジセットの一部分を手でバックアウトするように促す．

この節を読む前に，“hg backout” コマンドは履歴に**追加**することで取り消し操作を行うことを覚えておいて欲しい．変更や消去ではない．バグの修正のために役立つツールだが，変更の取り消しのために用いると破滅的な結果をもたらす．後者の目的のためには 9.4を参照のこと．

9.3.1 チェンジセットのバックアウト

“hg backout” コマンドはチェンジセット全体の作用を打ち消す．Mercurial の履歴は不変なので，このコマンドは取り消したいチェンジセットを取り除くものでは**ない**．その代わり，取り除きたいチェンジセットの**逆**の働きの新たなチェンジセットを生成する．

“hg backout” コマンドの動作はやや複雑なので，例を挙げて説明することにする．まずいくつかの単純な変更のあるリポジトリを考える．

```
$ 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” コマンドはバックアウトすべきチェンジセットの ID を一つ引数に取る．“hg backout” はデフォルトでコミットメッセージ入力のためにテキストエディタを起動するので．バックアウトの理由を記録しておく．この例では-m オプションを使ってコミットメッセージを記録している．

9.3.2 tip チェンジセットをバックアウトする

最後にコミットしたチェンジセットをバックアウトすることから始める．

```
$ hg backout -m 'back out second change' tip
reverting myfile
changeset 2:3c02b57bf61e backs out changeset 1:350b17a64f14
$ cat myfile
first change
```

myfile の 2 行目がなくなっているのが分かると思う．“hg log” を見てみると，“hg backout” が何をしたのかが分かる．

```
$ hg log --style compact
2[tip]   3c02b57bf61e   2009-06-09 06:06 +0000   bos
  back out second change

1   350b17a64f14   2009-06-09 06:06 +0000   bos
  second change

0   6fdb7a4fcc6c   2009-06-09 06:06 +0000   bos
  first change

```

“hg backout” が生成した新しいチェンジセットは，バックアウトしたチェンジセットの子になっている．9.1は更新履歴を図示したもので，理解の助けになるはずだ．図から分かるように履歴は線形で整合が取れている．

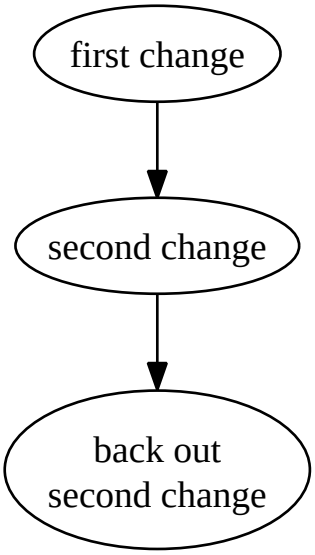

Figure 9.1: “hg backout” コマンドを使って更新をバックアウト

9.3.3 tip でない変更をバックアウトする

最後のコミット以外の変更をバックアウトしたい時は，“hg backout” に--merge オプションを付ける．

```
$ 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
```

このオプションはどんなチェンジセットでも一回の動作で行うことができ，手早く簡単である．

```
$ echo third change >> myfile
$ hg commit -m 'third change'
$ hg backout --merge -m 'back out second change' 1
reverting myfile
created new head
changeset 3:3c02b57bf61e backs out changeset 1:350b17a64f14
merging with changeset 3:3c02b57bf61e
merging myfile
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

バックアウトが終ったあとで myfile の中身を見ると，1 番目と 3 番目の変更だけが残っており，2 番目の変更が消えていることがわかる．

```
$ cat myfile
first change
third change
```

図 9.2で示された履歴で，Mercurial は 2 つのコミットを行っている．(図中で箱で示された節点は Mercurial が自動的にコミットした変更である．) バックアウトプロセスの前に Mercurial は，現在のワーキングディレクトリの親が何であるかを記憶する．そしてターゲットのチェンジセットを取り除き，これをチェンジセットとしてコミットする．最後にワーキングディレクトリの前の親へマージを行い，マージの結果をコミットする．

最終的に，いくらかの余計な履歴を残しつつ，取り除きたかったチェンジセットの影響を除去して，望む状態に戻すことができている．

常に--merge オプションを使う

バックアウトしようとするチェンジセットがチップかチップでないかにか変わらず，“hg backout” コマンドを使おうとする時は**常に**--merge オプションを使うべきである．対象がチップである場合は不要なマージを試みることはないため，常にこのオプションを指定して問題ない．

9.3.4 バックアウトプロセスをより細かく制御する

変更をバックアウトする際，常に--merge オプションを使うことを勧めたが，“hg backout” コマンドでは，バックアウトするチェンジセットをどのようにマージするか指定することができる．バックアウトプロセスを手でコントロールする必要はほとんどないはずだが，“hg backout” コマンドが自動でどのように動作しているの

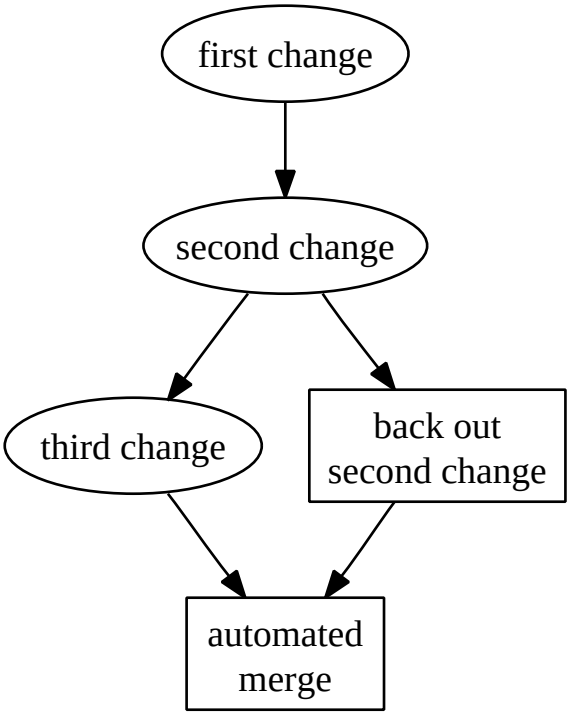

Figure 9.2: tip でない変更を “hg backout” コマンドで自動的にバックアウトする

か理解するのには役立つかも知れない．動作を説明するために最初のリポジトリのクローンを，バックアウトを除いて作って始めることにしよう．

前に使った例のように，3 番目のチェンジセットをコミットした後で，その親をバックアウトして何が起きるか見てみよう．

```
$ echo third change >> myfile
$ hg commit -m 'third change'
$ hg backout -m 'back out second change' 1
reverting myfile
created new head
changeset 3:f98b30967148 backs out changeset 1:350b17a64f14
the backout changeset is a new head - do not forget to merge
(use "backout --merge" if you want to auto-merge)
```

前の例と同様に，新しいチェンジセットはバックアウトしたチェンジセットの子にあたる．したがって新しいヘッドはチップの子では**ない**．“hg backout” コマンドはこの親子関係についてかなりはっきりと表示する．

```
$ hg log --style compact
3[tip]:1   f98b30967148   2009-06-09 06:06 +0000   bos
  back out second change

2   e7dadd84a9d7   2009-06-09 06:06 +0000   bos
  third change

1   350b17a64f14   2009-06-09 06:06 +0000   bos
  second change
```

```
0   6fdb7a4fcc6c   2009-06-09 06:06 +0000   bos
  first change

```

```
$ cd ..
$ hg clone -r1 myrepo newrepo
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 newrepo
```

ここでもリビジョン履歴のグラフ 9.3を見れば，何が起きているのか理解しやすい．チップ以外の変更をバックアウトするために“hg backout”を使った時，Mercurial が新しいヘッドをリポジトリに追加するのがわかる．（四角で示されたコミット.）

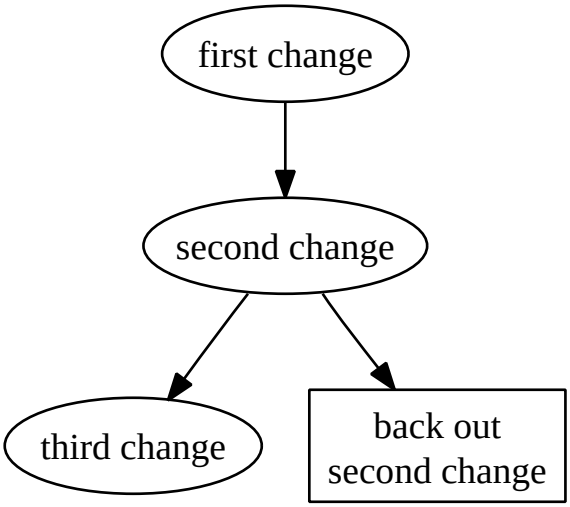

Figure 9.3: “hg backout”コマンドによる変更のバックアウト

“hg backout”コマンドは，完了した後，バックアウトチェンジセットをワーキングディレクトリの親として残す.

```
$ hg parents
changeset:   2:e7dadd84a9d7
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:27 2009 +0000
summary:     third change

```

ここで 2 つの隔離された変更のセットが存在することになる.

```
$ hg heads
changeset:   3:f98b30967148
tag:         tip
parent:      1:350b17a64f14
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:27 2009 +0000
summary:     back out second change

changeset:   2:e7dadd84a9d7
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:27 2009 +0000
summary:     third change

```

今ここで myfile の内容を見たいとする．最初の変更はバックアウトしていないので存在しているはずである．2 番目の変更はバックアウトしたので消えて無くなっているはずだ．3 番目の変更は履歴グラフで分離したヘッドとして表示されるため，3 番目の変更が myfile にあるとは期待していない．

```
$ cat myfile
first change
second change
third change
```

3 番目の変更をファイルに反映させるためには，2 つのヘッドをマージしてやればよい．

```
$ 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
```

その後ではリポジトリの履歴のグラフは図 9.4のようになる

9.3.5 なぜ "hg backout" はこのように動作するのか

"hg backout" がどのように動作するか完結に説明すると以下のようになる．

1. ワーキングディレクトリがクリーンであることを確認する．i.e."hg status" が空であることを確認する
2. 現在のワーキングディレクトリの親を記憶する．このチェンジセットを orig と呼ぶことにする．
3. ワーキングディレクトリを，バックアウトしたチェンジセットの内容にするために "hg update" と等価な動作を行う．このチェンジセットを backout と呼ぶ．
4. そのチェンジセットの親を見つけ出す．このチェンジセットを parent と呼ぶことにする．
5. backout チェンジセットが影響する各々のファイルに対して，"hg revert -r parent" と等価な操作を行い，チェンジセットがコミットされる前の状態に戻す．
6. 新しいチェンジセットの結果をコミットする．このチェンジセットは backout を親に持つ．

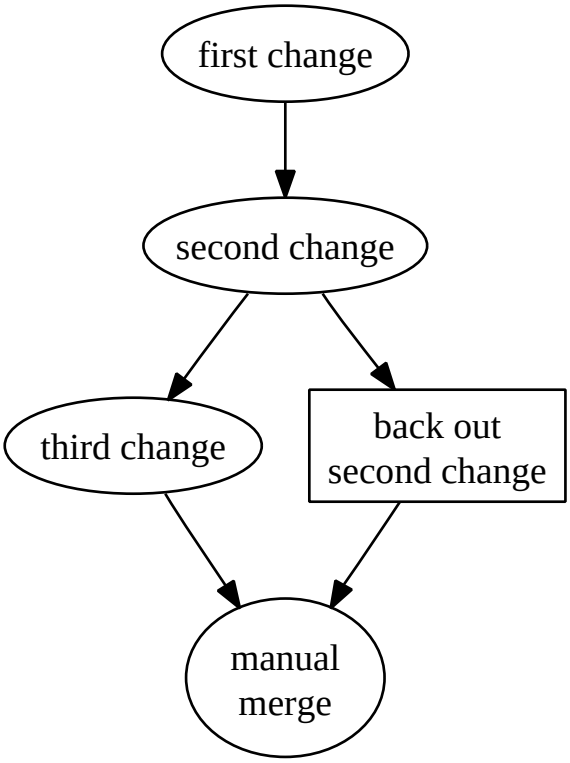

Figure 9.4: バックアウトチェンジの手動によるマージ

7. `--merge` をコマンドラインから入力した場合，`orig` とのマージを行い，結果をコミットする．

“`hg backout`” コマンドを実装する別の方法として，“`hg export`” でバックアウトされるべきチェンジセットを差分として出力し，`patch` を`--reverse` オプション付きで呼び，ワーキングディレクトリの操作を省略してリバースパッチする方法が考えられる．この方法はずっと単純だが，ほとんどうまく動かない．

“`hg backout`” がアップデート，コミット，マージ，コミットを行う理由は，バックアウトすべき変更と現在のチップの間で，マージ機構が最も良い結果を得られるようにするためである．

もしプロジェクトの履歴の中で 100 リビジョンも遡るようなチェンジセットをバックアウトするとすれば，`patch` が diff を適用できる可能性は明らかに低い．間にある変更が，`patch` コマンドがパッチ適用の可否を判断するための文脈を壊している可能性が高いからである．(この話が分からない場合は 12.4の `patch` コマンドに関する議論を参照して欲しい.）また，Mercurial のマージ機構は，`patch` コマンドが扱えないファイルとディレクトリのリネーム，パーミッション変更，バイナリファイルへの変更などを扱うことができる．

9.4　存在すべきでない変更

大抵の場合，“`hg backout`” はある変更を取り消そうとする際に思ったように機能するはずである．元々のコミットやそれを取り除いた時に何をしたのか永続的な記憶が残される．

それでもたまにリポジトリに全く残したくない変更をコミットしてしまうことがある．たとえば，非常に変なコミットや，通常ミスと考えられるようなコミット，ソースだけではなくプロジェクトのオブジェクトファイルもコミットしてしまうなどが有り得る．オブジェクトファイルは固有の値を持ち得ず，かつ**大きい**．そのため，リポジトリのサイズを増やし，クローンやプルに余計な時間が掛かるようになる．

茶色の紙袋コミット（茶色の紙袋を頭に被りたいぐらいバツの悪いコミット）に使えるオプションについて議論する前に，いくつかのうまく行かないアプローチを述べたい．

Mercurial は履歴を累積的なものとして扱う．全ての変更はそれに先立つ全ての変更の上になりたっている．一般的に言って破壊的な変更を回避することはできない．たった一つの例外はコミットを行った直後で，他のリポジトリにプッシュもプルもされていない場合である．その場合，9.1.2のセクションで詳しく触れたように，安全に “hg rollback” を実行することができる．

間違った変更を他のリポジトリにプッシュした後でもローカルコピーの変更を取り消すために依然として “hg rollback” を使うことが**できる**が，これは意図したような結果にはならない．変更は依然としてリモートのリポジトリには存在しており，次にプルした時にはローカルリポジトリにも現れる．

このような状況になった時，もしどのリポジトリにこの間違った変更が波及しているのかが明らかであれば，それらのリポジトリの一つ一つから，変更を取り除くことを試みることができる．これはもちろん満足のいく解決法ではない．もし一つでも消去を忘れれば，変更は野放しになっており，さらに広がりうる．

もし 1 つ以上の新たな変更を，消したいと思っている変更の後にコミットしていたとすれば，使えるオプションはさらに少なくなる．Mercurial はチェンジセットをそのままに履歴に穴を開けるような方法を提供していない．

XXX 追記の必要性あり．examples ディレクトリ内の hg-replay スクリプトは機能するが，チェンジセットのマージを扱わない．重要な制限である．

9.4.1 逸脱した変更から自分自身を守る

ローカルリポジトリにいくつかの変更をコミットし，それらが別の所にプッシュまたはプルされていて，必ずしも大災害とは言えない．あなたはいくつかのクラスの間違った変更から自分自身で身を守ることができる．これはあなたのチームが中央のリポジトリから変更をプルしている場合は特に簡単である．

中央リポジトリの上で，変更到着にフックを設定する（10を参照）ことで，ある種の誤ったチェンジセットが中央リポジトリにコミットされるのを自動的に防ぐことができる．そのような設定を行うことで，ある種の誤ったチェンジセットは中央リポジトリに波及することができず，死滅していく傾向がある．好都合なことに，死滅は陽に介入する必要なく起きる。

例えば，チェンジセットを実際にコンパイルする変更到着フックは，不注意によるビルド不能を防ぐことができる．

9.5 バグの原因を見つける

バグを発生させたチェンジセットをバックアウトするためには，どのチェンジセットでバグの混入が起きたのかを知らねばならない．Mercurial は “hg bisect” という有用なコマンドを提供しており，これによってチェンジセットの特定を自動化し，バックアウトを極めて効果的に行うことができる．

“hg bisect” コマンドの背後には，チェンジセットによって単純なバイナリテストで十分識別可能な挙動の変化が生れるという考え方がある．あなたはどのコードが変化を引き起こしたのかはわからないが，バグが起きているかをテストする方法は知っている．“hg bisect” コマンドは，テストによってバグを引き起こしたチェンジセットを特定する．

以下はこのコマンドをどのように適用できるのか理解を助けるシナリオである．

- バグが起きていなかった最も新しいバージョンを覚えているが，どのバージョンでバグが混入したか分からない．ここでバイナリテストを行い，バグの存在を調べる．
- バグを大急ぎで修正し，あなたのチームのバグデータベースをクローズする．バグデータベースはどこでバグが修正されたかのチェンジセットの ID を必要とするが，あなたはどのチェンジセットで修正されたか記憶していない．ここでまたバグの存在をバイナリテストする．
- あなたのソフトウェアは正しく動いたが，以前測定した時よりも 15%遅くなっていた．あなたはどのチェンジセットがこの性能劣化をもたらしたのか知りたい．この場合，バイナリテストで速度を測定する．
- 出荷するプロジェクトのコンポーネントサイズが最近爆発的に増えた．あなたはプロジェクトのビルドの方法になんらかの変化が起きたのではないかと疑っている．

この例から，“hg bisect” コマンドは単にバグの所在を探すのに役立つだけではなく、バイナリテストを用意できるのであれば、リポジトリで起きたあらゆる変化（単にツリーをテキストサーチしたのでは発見できない変化）を知るのに使えることがわかる。

ここでサーチのどのパートがあなたの責任に属し，どのパートが Mercurial に属するのか明確にするために少しばかり用語を導入する．*test* は**あなた**が “hg bisect” を実行する時に選んだチェンジセットである．*probe* は “hg bisect” がリビジョンが良いか判定するリビジョンである．“bisect” という単語を ““hg bisect” コマンドを使ってサーチする” ということの同義語として名詞と動詞両方で用いる．

サーチプロセスを自動化する単純な方法は，全てのチェンジセットを probe することである．しかしこれは殆んどスケールしない．一つのチェンジセットのチェックに 10 分かかり，リポジトリにチェンジセットが 10000 あったとしたら，虱潰しのアプローチはバグを発生させたチェンジセットの特定に平均で 35 日かかる．バグが最後の 500 チェンジセットで発生したことが分かっていて，サーチをそれらのチェンジセットに限定したとしても，バグを引き起こしたチェンジセットの特定に 40 時間以上かかる．

“hg bisect” はあなたのプロジェクトのリビジョン履歴の “シェイプ” の知識を，サーチすべきチェンジセット数の**対数**に比例した時間でサーチするために使う．（二分探索を行う．）このアプローチによって 10000 チェンジセットのサーチは，各々のサーチが 10 分かかったとしても，テスト数は 14 で 3 時間以下で終了する．最後の 100 のチェンジセットに限って行ったとすると，およそ 7 回のテストで 1 時間程度で終了する．

“hg bisect” コマンドは，Mercurial プロジェクトのリビジョン履歴が枝分かれしがちな性質を持つことを念頭において設計されており，リポジトリにブランチ，マージ，複数のヘッドがあっても問題なく取り扱える．一回の probe で履歴の中のある分枝全てを刈ることができるため，極めて効率的に動作する．

9.5.1 “hg bisect” コマンドを使う

“hg bisect” の動作を例で示す．

> **Note:** バージョン 0.9.5 以前の Mercurial では，“hg bisect” はコアコマンドではなく，extension として Mercurial に同梱されていた．この節は古い extension についてではなく，ビルトインコマンドとしての “hg bisect” について述べている．

独立して “hg bisect” を試せるようにリポジトリを作成する．

```
$ hg init mybug
$ cd mybug
```

単純なバグのあるプロジェクトをシミュレートする．バグは繰返し操作の間に自明な変更を行い，特定の 1 つの変更をバグを持つと指名する．このループは 35 のチェンジセットを作り，各々，1 つのファイルをリポジトリに追加する．バグは “i have a gub” という文字列を持つファイルで表現する．

```
$ 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
```

次に “hg bisect” の使い方を説明したい．Mercurial のビルトインのヘルプが使える．

```
$ 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 --noupdate option is specified).
    Once you have performed tests, mark the working directory as bad
    or good 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.

    If you supply a command it will be used for automatic bisection. Its exit
    status will be used as flag to mark revision as bad or good. In case exit
    status is 0 the revision is marked as good, 125 - skipped, 127 (command not
    found) - bisection will be aborted; any other status bigger than 0 will
    mark revision as 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” コマンドは段階を踏んで動作する．各段階は以下のように進む．

1. バイナリテストを実行する
    - テストが成功した場合，“hg bisect” を引数 “hg bisect good” を付けて実行する．
    - テストが失敗した場合は “hg bisect --bad” コマンドを実行する．
2. コマンドは引数で渡された情報を使って次にどのチェンジセットをテストすべきか決定する．
3. コマンドはワーキングディレクトリをそのチェンジセットに更新する．プロセスを再開する．

“hg bisect” のプロセスは，テストが成功から失敗へ変化したチェンジセットを特定できると終了する．
サーチを始めるには “hg bisect --reset” を実行する．

```
$ hg bisect init
(use of 'hg bisect <cmd>' is deprecated)
```

このケースでは，バイナリテストは単純で，リポジトリ内に “i have a gub” を含むファイルがあるかどうかをチェックするだけである．もしこれがあれば，そのチェンジセットがバグを引き起こしている変更を含んでいるということになる．慣例に従って，探しているチェンジセットを “bad”，それ以外を “good” と呼ぶことにする．

大抵の場合，ワーキングディレクトリが同期しているリビジョン（通常は tip のはずだ）は，バグのある変更によって引き起こされた問題を示しているので，このリビジョンを “bad” とマークする．

```
$ hg bisect bad
(use of 'hg bisect <cmd>' is deprecated)
```

次にバグを**含まない**リビジョンを指名する．“hg bisect” コマンドは最初の “good” チェンジセットと “bad” チェンジセットをブラケットする．この例では，リビジョン 10 はバグを持たないと分かっている．(最初の “good” チェンジセットの選び方については後述する．)

```
$ hg bisect good 10
(use of 'hg bisect <cmd>' is deprecated)
Testing changeset 22:0a1e0af7d61f (24 changesets remaining, ~4 tests)
0 files updated, 0 files merged, 12 files removed, 0 files unresolved
```

コマンドから出力があることに注意．

- バグの入ったチェンジセットを特定するまでに考慮すべきチェンジセットの個数と，必要な検査の回数を表示する．
- ワーキングディレクトリを次のテストすべきチェンジセットに更新し，どのチェンジセットをテストしているのかを表示する．

ここでワーキングディレクトリでテストを行うことができる．grep コマンドを使って “bad” ファイルがワーキングディレクトリにあるかどうかをチェックする．もし存在すれば，リビジョンは bad で，存在しなければリビジョンは good ということになる．

```
$ 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
Testing changeset 16:1c6866353ff9 (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る．シェル関数にしてみよう．

```
$ mytest() {
>   if grep -q 'i have a gub' *
>   then
>     result=bad
>   else
>     result=good
>   fi
>   echo this revision is $result
>   hg bisect $result
> }
```

テスト全体を一つの mytest コマンドとして実行できるようになった．

```
$ mytest
this revision is good
(use of 'hg bisect <cmd>' is deprecated)
Testing changeset 19:a53789dfb9fb (6 changesets remaining, ~2 tests)
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

このパックされたテストコマンドを起動するだけでテストが完了する．

```
$ mytest
this revision is good
(use of 'hg bisect <cmd>' is deprecated)
Testing changeset 20:d702d251b2a5 (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:c17bdbe80350 (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:   22:0a1e0af7d61f
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:06:31 2009 +0000
summary:     buggy changeset

```

40 個のチェンジセットがあるにもかかわらず，“hg bisect” コマンドはバグをもたらしたチェンジセットを 5 回のテストで発見することができた．“hg bisect” のテスト回数は探索すべきチェンジセット数が増えるに従って，対数的に増えるが，虱潰し探索に対する優位性は，チェンジセットが増加するに従って増加する．

9.5.2 サーチ後のクリーンアップ

リポジトリ内で “hg bisect” コマンドを使い終った後に “hg bisect reset” コマンドで探索に使った情報を消去することができる．コマンドは大きなスペースを使うわけではないので，このコマンドを実行するのを忘れても特に問題はない．しかし “hg bisect” は “hg bisect reset” が実行されるまで，そのリポジトリ内で新たな探索を始めることはできない．

```
$ hg bisect reset
(use of 'hg bisect <cmd>' is deprecated)
```

9.6 効率的なバグの発見法

9.6.1 正しい入力を行う

“hg bisect” コマンドは各テストで正しく結果を入力することが必要である．もしテストに成功したのに失敗と入力すれば，矛盾が生じうる．入力の中に矛盾があることが分かれば，コマンドは特定のチェンジセットを good かつ bad と表示する．しかし矛盾を完璧に見つけ出すことは困難で，誤ったチェンジセットをバグの起源と報告する可能性が高い．

9.6.2 できる限り自動化する

“hg bisect” コマンドを使いはじめた時，数回，コマンドラインから手でテストを実行した．この方法は適した方法ではない．数回のテストの後で，最終的な結果を得るためにサーチをやりなおさなければならないことに気づくことが何度かあった．

“hg bisect” コマンドを手で実行していた時の最初の問題は，小さなリポジトリでの単純なサーチでも起こった．もし問題がもっと劣悪で，“hg bisect” が実行しなければならないテストの数が増えれば，オペレータのエラーが起こる可能性はずっと高くなる．一度テストを自動化してからは，ずっとよい結果が得られるようになった．

テストを自動化するキーは 2 つある．

- 常に同じ兆候をテストする．そして
- 常に “hg bisect” コマンドに一貫した入力を行う．

上のチュートリアル例では grep コマンドが兆候をテストしており，if 文がこのチェックの結果を取り，常に同じ入力が “hg bisect” コマンドになされるようになっていた．mytest 関数は，再現可能な方法で，これらを結びつけ，ここのテストが一様で一貫するようになっていた．

9.6.3 結果をチェックする

“hg bisect” サーチの出力は，ユーザの入力が正しい場合に限って正しいので，コマンドが報告するチェンジセットを絶対に正しいと考えてはいけない．報告のクロスチェックを行う単純な方法は，以下のチェンジセットに対して，手動でテストを行うことである．

- 最初の bad リビジョンとして報告されるチェンジセット．これについてテストは bad となる筈である
- そのチェンジセットの親（マージされているなら両方の親）．テストは good になる筈である
- 最初の bad リビジョンとして報告されるチェンジセットの子．テストは bad となる筈である

9.6.4 バグ同士の相互干渉に留意する

一つのバグに対するサーチがその他のバグの存在で混乱させられることが有り得る．例えばソフトウェアがリビジョン 100 でクラッシュし，リビジョン 50 で動作したとする．あなたには未知だが，誰か別の人がリビジョン 60 で他のクラッシュするバグを持ち込み，リビジョン 80 で修正したとする．これは結果を複数の方向へ捩曲げ得る．

この別のバグが追いかけているバグを完全にマスクしてしまうことも有り得る．追いかけているバグが明らかになる前にこれが起きることも有り得る．他のバグが防ぎ切れないなら，バグが特定のチェンジセットに含まれると言うこともできない．“hg bisect” コマンドは直接あなたを助けることができない．その代わり，あるチェンジセットを “hg bisect --skip” によって未テストとマークすることができる．

バグのテストが十分に詳細でなかった場合，別の問題が起こり得る．プログラムがクラッシュするかどうかをチェックしている時，追跡しているクラッシュバグと，それとは無関係なクラッシュバグが混在しているとこれらは同じに見えてしまい，結果的にチェックをマスクしてしまい，“hg bisect” をミスリードする．

“hg bisect --skip” が有効な別の例として，ビルド不能になる変更がチェックインされたリビジョンがあり，これを飛ばさなければテストができないような状況がある．

9.6.5 探索を怠惰にブラケットする

最初の good と bad チェンジセットの組を選び，サーチ範囲を指定するのは多くの場合簡単だが，若干の議論の余地がある．“hg bisect” から見た時，最新のチェンジセットは bad で，古いチェンジセットは good となる．

適切な good チェンジセットを覚えていない時，“hg bisect” にそれを指定すると，ランダムに選ぶよりも悪い探索を行う可能性がある．バグを示さない（おそらくバグのある機能が実装されていない）チェンジセットと，他の問題が追跡中のバグをマスクしてしまうようなチェンジセットのみを消去することを覚えておくべきだ．

“hg bisect” が行わなければならないテストの回数は，対象とするチェンジセットの対数に比例するため，数ヵ月に及ぶ履歴の中の数千のチェンジセットの中に対しても少数のテストを追加するだけで済む．

Chapter 10

リポジトリイベントをフックで取り扱う

Mercurial は，リポジトリに起こるイベントに対して自動化されたアクションを行う強力なメカニズムを持っている．いくつかのイベントに対しては，Mercurial の反応を制御することもできる．

Mercurial は**フック**と呼ばれる機構を使ってアクションを行う．フックは他のリビジョンコントロールシステムでは “トリガ” と呼ばれることもあるが，これらは実際は同じものである．

10.1 Mercurial でのフックの概要

Mercurial がサポートするフックの一部を列挙する．個々のフックについては後ほど section 10.8で詳細に触れる．

`changegroup` チェンジセットグループが外部からリポジトリに加わった後で呼び出される．

`commit` ローカルリポジトリで新しいチェンジセットが作成された後に呼び出される．

`incoming` 新しいチェンジセットが外部からリポジトリに加わる毎に呼び出される．*group* が加わる毎に呼び出されるフックに `changegroup` フックがある．

`outgoing` リポジトリからチェンジセットグループが外部に送信された後に呼び出される．

`prechangegroup` チェンジセットグループをリポジトリから外部に持ち出す前に呼び出される．

`precommit` Controlling. コミットが始まる前に呼び出される．

`preoutgoing` Controlling. チェンジセットグループをリポジトリから外部に送信する前に呼び出される．

`pretag` Controlling. タグが作成される前に呼び出される．

`pretxnchangegroup` Controlling. チェンジセットグループが外部からローカルリポジトリに加わった後，永続的に記録するためのトランザクションが完了する前に呼び出される．

`pretxncommit` Controlling. 新しいチェンジセットがローカルリポジトリで作成された後，永続的に記録するためのトランザクションが完了する前に呼び出される．

`preupdate` Controlling. ワーキングディレクトリの更新またはマージが行われる前に呼び出される．

`tag` タグが作成された後で呼び出される．

`update` ワーキングディレクトリの更新かマージが完了した時に呼び出される．

フックのうち，説明が “Controlling” という語で始まるものは，フックの成否を判定し，以後の動作を行うかどうか決定する機能がある．フックが成功すると動作は先へ進む．フックが成功しなかった場合は以後の動作は実行が許可されないか，動作が完了せず終る．

10.2 フックとセキュリティ

10.2.1 フックはユーザの権限で動作する

リポジトリ内で Mercurial コマンドを実行し，コマンドがフックを呼び出した場合，フックは**操作者**のシステムで，**操作者**の特権レベルで動作する．フックは任意の実行コードなので，相応の留意が必要である．フックを作ったのが誰で，そのフックが何をするのかきちんと理解することなしにフックをインストールしてはならない．

自分でインストールしたのではないフックを実行させてしまうことも有り得る．他所のシステムの上で Mercurial を使う場合，Mercurial がシステムワイドの `hgrc` ファイルで定義されたフックを起動する可能性がある．

他のユーザが所有するリポジトリで作業する場合，Mercurial は所有者のリポジトリ内で定義されたフックを実行し得るが，実行権限はあなたのもので行われる．例えばそのリポジトリから "`hg pull`" を行うとき，リポジトリ内の `.hg/hgrc` ファイルでローカルな `outgoing` フックが定義されていると，あなたはそのリポジトリを所有していないにもかかわらずそのフックはあなたの権限で実行される．

> **Note:** これはローカルなリポジトリや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る．http または ssh を使って pull した場合，すべての `outgoing` フックはサーバ上でサーバプロセスを実行しているアカウントの権限で実行される．

XXX リポジトリ内で定義されているフックを見るには "`hg config hooks`" コマンドを使う．あるリポジトリで作業していて，新しいコードを他のリポジトリから入手するために通信を行う場合（例えば "`hg pull`" か "`hg incoming`" を実行する），チェックすべきフックは手元のリポジトリのものではなく，リモートのリポジトリのフックである．

10.2.2 フックは伝播しない

Mercurial では，フックはリビジョン管理されず，リポジトリからクローンしたり pull しても伝播しない．こうなっている理由は単純で，フックは任意の実行可能コードであるからである．フックはあなたのマシンの上であなたのユーザー ID と特権レベルで動作で動く．

分散リビジョンコントロールシステムにリビジョン管理されたフックを実装するのはとても無謀である．このような機構があるとリビジョンコントロールシステムのユーザアカウントを偽装するような仕掛けを簡単に実現できてしまう．

Mercurial はフックを伝播させないので，共通のプロジェクトで他の開発者と協力している場合，彼らはあなたが使ってるのと同じ Mercurial フックを使っていると仮定したり，彼らのリポジトリでもフックが同様に正しく設定されていると仮定すべきではない．彼らがフックを使うように望むのであれば，使用法についてドキュメントを作成すべきである．

企業内のイントラネットではコントロールは幾分簡単である． NFS ファイルシステム上で Mercurial の "標準" インストレーション例を提供し，すべてのユーザが参照するサイトワイドの `hgrc` ファイルを用意し，その中でフックを定義することができる．しかしこの方法にも下記のような欠点がある．

10.2.3 フックはオーバライド可能である

Mercurial では，フックを再定義することによって，既存のフック定義をオーバライドすることができる．これは変数を空の文字列にしたり，挙動を変更することで禁止することができる．

フックを定義したシステムワイドまたはサイトワイドの `hgrc` ファイルを使用しても，リポジトリのユーザがフックを禁止したりオーバライドできてしまうのに気付くだろう．

10.2.4 クリティカルなフックが確実に実行されるようにする

時には他の開発者が回避策をとれないようなポリシーを要請したい場合があるだろう．例えばすべてのチェンジセットが厳格なテストセットにパスすることを必要とする場合が考えられる．これをサイトワイドの `hgrc`

でフックとして定義した場合，これはラップトップを使っているリモートユーザには機能しないし，フックをオーバライドできるため，ローカルユーザにとっても無視できてしまう．

その代わり，Mercurial の使用に関するポリシーを設定し，適切に設定され，厳重に管理された既知の “カノニカル” サーバを通じて変更が波及するようにすることができる．

これを実現する一法はソーシャルエンジニアリングと技術の組み合わせで行う方法である．限定アクセスアカウントを設定し，このアカウントで管理されているリポジトリに対してユーザは変更をネットワークを介して push できる．しかしユーザはこ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することはできない．このシナリオでは，ユーザはどんな古いゴミを含んだチェンジセットでもコミットすることができる．

多くのユーザが pull するサーバに誰かがチェンジセットを push した場合，サーバはチェンジセットを恒久的に受け入れる前にチェックを行い，一連のテストをパスできなければリジェクトを行う．ユーザがこのフィルタサーバからのみ pull を行うのであれば，全ての変更は自動的に全て検査されていることになる．

10.3 共有アクセスリポジトリで `pretxn` フックを使う

複数のユーザが共有アクセスを行うリポジトリで，自動化された作業を行うためにフックを使用したいなら，どのように行うか注意深く考える必要がある．

Mercurial はリポジトリに書き込みを行うときにだけリポジトリをロックする．また Mercurial の書き込みを行う部分のみがロックを考慮する．書き込みロックは，複数の同時書き込みが他の変更を上書きし，リポジトリを破損するのを防ぐ．

Mercurial はデータの読み書きの順序を注意深く行うため，リポジトリからのデータ読み出しの際にロックを取得する必要がない． Mercurial のリポジトリから読み出しを行う部分は，ロックを全く気にする必要がない．このロックなし読み出しによって，同時実行性と性能を大幅に高めている．

しかしながらこの高性能はそれを知らなければ問題を引き起こすあるトレードオフをももたらす．これを説明するために，Mercurial がどのようにチェンジセットをリポジトリに追加し，それらを読み出すかの詳細に触れなければならない．

Mercurial はメタデータを**書き込む**とき，直接目的のファイルに書き込みする．Mercurial はまずファイルデータを書き込み，次いで（新しいファイルデータの場所を示すポインタを含む）マニフェストデータを書き込む．そして（新しいマニフェストデータの場所を示すポインタを含む）チェンジログデータを書き込む．各々のファイルへの最初の書き込みの前に，ファイルの末尾のレコードをトランザクションログに保存する．トランザクションがロールバックされる場合は，Mercurial は各々のファイルをトランザクションが始まる前のサイズに切り詰める．

Mercurial はメタデータを**読む**時にまずチェンジログを読み，次いで残りの部分を読む．リーダはチェンジログに現れるマニフェストの一部またはファイルメタデータの一部にのみアクセスするため，部分的に書かれたデータを見ることはできない．

いくつかの制御フック（`pretxncommit` と `pretxnchangegroup`）はトランザクションがほぼ完了した時に実行される．すべてのメタデータが書き込まれるが，この時点でも Mercurial はトランザクションを元に戻すことができ，その場合は新しく書かれたデータは消失する．

これらのフックのうち 1 つが長時間にわたって実行されていると，リーダがチェンジセットのメタデータを読むことのできるタイムウィンドウが開く．このチェンジセットはまだ永続的なものになっておらず，従って実在すると考えるべきではないものである．フックが実行されている時間が長くなればなるほど，タイムウィンドウが開く時間も長くなる．

10.3.1 問題の詳細

実用における `pretxnchangegroup` フックの良い使用法としては，到着した変更が中央のリポジトリに取り込まれる前に自動でビルドとテストを行うことが考えられる．これにより，ビルドを妨げる変更は誰もリポジトリに push できないことが確実になる．クライアントがテスト中に変更を pull することができれば，このテストの有用性はゼロになってしまう．疑いを持たずに誰かがテストされていない変更を pull できるのであれば，彼らのビルドは失敗する可能性がある．

この問題への技術的に最も安全な回答は，“門番” リポジトリを**一方向**に設定することである．リポジトリを外部から push された変更を受け取るが，誰も pull できないように設定する（`preoutgoing` フックを使って

リポジトリをロックする）．changegroup フックを設定し，ビルドやテストが成功したときに限って，フックが新たな変更をユーザの pull **できる**別のリポジトリに push するようにする．

実際上は，このように集中したボトルネックを置くことは良い考えとは言えず，トランザクションの可視性は全くない．プロジェクトのサイズおよびビルドとテストに要する時間が増加するに従って，このような“事前に試す”手法は壁に突き当たる．テストに使える時間で捌き切れないほどのチェンジセットを試さなければならなくなるからである．フラストレーションが貯るのは避けられないだろう．

よりスケールする手法は，開発者に push 前のビルドとテストをさせることである．中央で自動によるビルドとテストを行うのは，push **後**に，全てに問題がないことを確認するために行う．このアプローチの利点はリポジトリが変更を受け入れるペースに何も制限を課さないことである．

10.4 フックの使用法

Mercurial フックを書くのは容易い．ここでは“hg commit”コマンドが終了した際に，作成されたチェンジセットのハッシュ値を表示するくフックを書いてみよう．このフックを commit と呼ぶことにする．

```
$ 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 ab436e617bb8aa592d079b34b90f9144d289d53f
```

Figure 10.1: チェンジセットがコミットされた時に動作する単純なフック

全てのフックは例 10.1と同じパターンになる． hgrc ファイルの [hooks] セクションにエントリを追加する．左辺にトリガーとなるベントを記述し，右辺に対応するアクションを記述する．例に示した通り，フックには任意のシェルコマンドを書くことができる．環境変数を設定することで Mercurial からフックに追加の情報を渡すことができる．（例の HG_NODE を参照．）

10.4.1 1 つのイベントに複数のアクションを行う

例 10.2に示したように，特定のイベントに 2 つ以上のフックを定義することが必要になることが多いだろう．Mercurial では，フック名の最後に**拡張子**を追加することでこれが可能になる．拡張子を付けるには，“.” 文字と，これに続く何文字かからなる名前をフックにつければよい．例えば Mercurial は commit イベントが起きた時に commit.foo と commit.bar フックの両方を呼び出す．

```
$ echo 'commit.when = echo -n "date of commit: "; date' >> .hg/hgrc
$ echo a >> a
$ hg commit -m 'i have two hooks'
committed 82e997376f919642c4f9978bc9f62da8b21e0bbd
date of commit: Tue Jun  9 06:06:59 GMT 2009
```

Figure 10.2: 2 番目の commit フックを定義する

1 つのイベントに複数のフックが定義されている時，Mercurial はフックを拡張子でソートし，ソートされた順序に従ってフックを実行する．上記の例では `commit`，`commit.bar`，`commit.foo` の順に実行する．

新しいフックを定義する時に，内容を説明するような拡張子を付けるのはよい考えである．こうすることによって，フックの目的が何なのかを覚えやすくなる．フックが失敗した場合，エラーメッセージにはフック名と拡張子が含まれる．説明的な拡張子はフックが失敗した理由を知るよい手がかりとなる．(例は 10.4.2を参照のこと.)

10.4.2 動作が進行できるかどうか制御する

前の例ではコミットが完了した後に実行される `commit` フックを用いた．これは Mercurial フックのうち，動作が終了した後に実行されるもののうちの 1 つである．このようなフックは，Mercurial の動作そのものに影響を及ぼさない．

Mercurial は動作の開始前や，開始後終了するまでの間に発生するイベントを多数定義している．これらのイベントでトリガーされるフックは，動作を続行するか中断するか決めることができる．

`pretxncommit` フックはコミット後，コミットの完了前に呼び出される．言い替えると，チェンジセットを示すメタデータがディスクに書き込まれた後で，トランザクションが完了する前に呼び出される．`pretxncommit` フックはトランザクションを完了するか，ロールバックするかを決定する機能がある．

`pretxncommit` フックがステータスコード 0 で終了するとトランザクションは完了することができ，コミットが終了し，`commit` フックが呼び出される．`pretxncommit` フックが非 0 のステータスコードで終了すると，トランザクションはロールバックされ，チェンジセットを示すメタデータは消去される．この場合 `commit` フックは呼び出されない．

```
$ 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 b24eda6993d093e5c5a7d432a1856f75fa8e373e
date of commit: Tue Jun  9 06:06:59 GMT 2009
```

Figure 10.3: コミットを制御するために `pretxncommit` フックを使用する

10.3はコミットコメントがバグ ID を含むかをチェックする．もし含まれればコミットは完了する．含まない場合はコミットはロールバックされる．

10.5 オリジナルのフックを書く

フックを書く場合，Mercurial を`-v` オプションを付けたり `verbose` 設定項目を “true” に設定することは役に立つ．この場合，Mercurial は各々のフックを呼ぶ前にメッセージを表示する．

10.5.1 フックがの動作方法を選ぶ

フックはシェルスクリプトのような通常のプログラムとして書くこともできるし，Mercurial プロセスの内部で呼び出される Python 関数として書くこともできる．

フックを外部プログラムとして書く利点はMercurialの内部動作を知らなくてもフックが書けることである．必要な追加の情報を得るために通常のMercurialコマンドを呼び出すことができる．外部フックはプロセス内フックよりも低速なのがトレードオフである．

Pythonプロセス内フックはMercurial APIへの完全なアクセスが可能である．他のプロセスを生成することがないため，本質的に外部フックよりも高速である．フックが必要とする情報の大半は，Mercurialコマンドを実行するよりもMercurial APIを使って集める方が容易い．

Pythonを使い慣れていたり，高い性能が必要ならば，フックをPythonで書くのがよいだろう．しかし書こうとするのが単純明解なフックで,（多くのフックがそうであるように）性能には特に関心がない場合，シェルスクリプトとして書いても全く問題ない．

10.5.2 フックパラメータ

Mercurialは各々のフックの呼出しの際にきちんと定義されたパラメータを渡す．Pythonでは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る．外部プログラムには，パラメータは環境変数として渡される．

フックがPythonで書かれているか，シェルスクリプトして書かれているかに関わらず，フック固有のパラメータ名と値は同じものが用いられる．ブール型パラメータはPythonではブール値として扱われるが，外部フックのための環境変数では，1（“真”）または0（“偽”）という数値として扱われる．フックパラメータに`foo`という名前が付けられているとき，Pythonフックへのキーワード引数も同じ`foo`という名前になる．一方，外部フックのための環境変数では`HG_FOO`という名前になる．

10.5.3 フックの戻り値と動作の制御

正しく動作した外部フックはステータスとしてゼロを，プロセス内フックの場合はブール値“false”を返さなければならない．なんらかの失敗があった場合は，外部フックは非ゼロの終了ステータスを，プロセス内フックはブール値“true”を返す．もしプロセス内フックが例外を発生した場合は，フックは失敗したと見なされる．

動作の継続を制御するフックの場合は，ゼロ／falseは継続の許可を，非ゼロ／true／例外の場合は禁止を意味する．

10.5.4 外部フックを作成する

外部フックを`hgrc`で定義し，実行する場合，フックはシェルに渡される値を変換すため，フックの本体部分でシェルスクリプトを記述することができる．

外部フックは常にリポジトリのルートディレクトリをカレントディレクトリとして実行される．

各々のフックパラメータは環境変数として渡される．すなわち，名前は全て大文字になり，“`HG_`”という接頭辞が付けられる．

フックパラメータを例外として，Mercurialはフックを実行する際に環境変数の定義や変更を行わない．多くのユーザが実行するサイトワイドのフックを作成する場合，ユーザが設定している環境変数は異なることを記憶しておくべきである．マルチユーザ環境の場合，フックを試す際にあなたが設定している環境変数に依存すべきではない．

10.5.5 Mercurialにプロセス内フックを使うように指示する

プロセスないフックを定義する`hgrc`構文は実行可能フックとは僅かに異なっている．フックの値は必ず“`python:`”で始まり，後にフック値として用いられる呼出し可能オブジェクトの完全な名前が続かなければならない．

フックが含まれるモジュールはフックが実行される時に自動的に読み込まれる．モジュール名および`PYTHONPATH`が正しいかぎり必ず動作するはずである．

今説明した構文と概念を説明する断片的な例を以下の`hgrc`に示す．

```
[hooks]
commit.example = python:mymodule.submodule.myhook
```

Mercurialが`commit.example`フックを実行する時，`mymodule.submodule`を読み込み，呼出し可能オブジェクト`myhook`を探し，実行する．

10.5.6 プロセス内フックを作成する

実際には何も行わない最も単純なプロセス内フックをフック API の基本的な使い方を説明するために示す.

```
def myhook(ui, repo, **kwargs):
    pass
```

Python フックへの最初の引数は常に mercurial.ui.ui オブジェクトである．2 番目の引数はリポジトリオブジェクトで，現在のところ常に mercurial.localrepo.localrepository のインスタンスである．他のキーワード引数はこれらの 2 つの引数に続く．これらは呼び出されているフックに依存するが，フックはキーワードの辞書に**kwargs と記述することで，必要のない引数を無視することもできる.

10.6 フックの例

10.6.1 意味のあるコミットメッセージを出力する

非常に短いコミットメッセージが有用であることはまずない．図 10.4に示す pretxncommit という単純なフックは 10 バイト以下の長さのコミットメッセージでコミットを行うことを禁止する.

```
$ 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'
```

Figure 10.4: 極端に短いコミットメッセージを禁止するフック

10.6.2 ぶら下がった空白をチェックする

コミットに関連したフックの興味深い使用法の一つにより奇麗なコードを書く手助けがある．“奇麗なコード”のごく単純な例は，例えば，新たなぶら下がった空白を含む行を含まないものである．ぶら下がった空白とは，行末の一連のスペースやタブである．ほとんどの場合，ぶら下がった空白は不必要な見えないノイズであるが，問題を引き起こす場合もあり，除去したがる人が多い.

ぶら下がり空白の問題を直すために precommit フックや pretxncommit フックを用いることができる．precommit フックを使う場合，どのファイルをコミットするのかフックは知ることがない．そのため，リポジトリで変更されたファイルすべてについてぶら下がり空白をチェックする必要がある． foo というファイルをコミットしたいが，bar ファイルがぶら下がり空白を含んでいる場合，precommit フックでチェックを行うと，ファイル bar の問題のために foo のコミットができなくなる．これは正しい挙動とは言えない.

pretxncommit フックを選び，チェックがコミット完了トランザクションの直前まで起きないようにすべきである．これにより，コミットされようとするファイルだけをチェックすることができるようになる．しかしながら，コミットメッセージを対話的に入力し，コミットが失敗するとトランザクションはロールバックするため，ぶら下がり空白を修正し，“hg commit” を再び実行し，コミットする際にコミットメッセージを再入力しなければならない.

図 10.5は pretxncommit というぶら下がり空白をチェックする単純なフックを導入している．このフックは短いが，それほど有用ではない．更新がぶら下がり空白を含む行をどのファイルに追加しても，このフックは

```
$ 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'
```

Figure 10.5: ぶら下がった空白をチェックする単純なフック

エラーステータスと共に終了するが，問題のあるファイルや行を特定するための情報は一切表示しない．このフックは変更のない行には全く関心を持たないという良い性質も持っている．新規にぶら下がり空白を加える問題のある行のみを検査する．

```
$ cat .hg/hgrc
[hooks]
pretxncommit.whitespace = .hg/check_whitespace.py
$ echo 'a ' >> a
$ hg commit -A -m 'add new line with trailing whitespace'
sh: .hg/check_whitespace.py: /usr/bin/python: bad interpreter: No such file or directory
transaction abort!
rollback completed
abort: pretxncommit.whitespace hook exited with status 126
$ sed -i 's, *$,,' a
$ hg commit -A -m 'trimmed trailing whitespace'
sh: .hg/check_whitespace.py: /usr/bin/python: bad interpreter: No such file or directory
transaction abort!
rollback completed
abort: pretxncommit.whitespace hook exited with status 126
```

Figure 10.6: ぶら下がり空白をチェックするフックの改良版

図 10.6はかなり複雑だが，より有用である．このフックは統合形式の diff をパースし，ぶら下がり空白を含むラインを探す．そして見つけるとファイル名と行番号を表示する．さらに変更がぶら下がり空白を追加すると，このフックはコミットコメントを保存し，終了前に保存ファイルの名前を表示する．そして Mercurial にトランザクションをロールバックすることを指示する．問題を修正した後で保存されたコミットメッセージを使うには “`hg commit -l` *`filename`*” とすればよい．

最後に，図 10.6のようにファイルからぶら下がり空白を除去するために `perl` コマンドのインプレイス編集機能を用いる．これはここで再現するに当たって十分短くかつ有効である．

```
perl -pi -e 's,                                s+$,,' filename
```

10.7 組み合わせフック

Mercurial にはいくつかのフックが同梱されている．フックは Mercurial ソースツリーの `hgext` ディレクトリにある． Mercurial のバイナリパッケージを使用しているのであれば，フックはパッケージインストーラが Mercurial をインストールしたディレクトリ内の `hgext` ディレクトリにあるはずだ．

10.7.1 `acl`—リポジトリの部分に対するアクセスコントロール

`acl` エクステンションは，リモートユーザにチェンジセットをネットワーク接続されたサーバにプッシュできるようにする．特定のリモートユーザが保護された以外の部分の変更をプッシュできるようにリポジトリの任意の部分（全体をも含む）を保護することができる．

このエクステンションはユーザがプッシュを行う際に，誰がチェンジセットをコミット**できない**ようにすることでアクセスコントロールを実装している．このフックは，リモートユーザの認証を行う保護されたサーバ環境で，指定されたユーザだけが変更をサーバにプッシュできるようにしたい場合にのみ意味をなす．

`acl` フックの設定

`acl` フックを，到着するチェンジセットを管理するために使うためには，`pretxnchangegroup` フックとして用いる必要がある．このフックは，各々のチェンジセットでどのファイルが変更されたのかをチェックし，変更禁止のファイルへ変更があった場合はチェンジセットをロールバックする．例：

```
[hooks]
pretxnchangegroup.acl = python:hgext.acl.hook
```

`acl` エクステンションは 3 つのセクションで設定される．

`[acl]` セクションは，`sources` というエントリ 1 つを持つ．このエントリでフックが監視すべき到着チェンジセット内ソースを列挙する．

`serve` リモートリポジトリから http または ssh を使って到着するチェンジセットを制御する．`sources` のデフォルト値で，通常はこの設定内で唯一設定する必要のある項目である．

`pull` ローカルリポジトリから pull したチェンジセットを制御する．

`push` ローカルリポジトリから push したチェンジセットを制御する．

`bundle` 別のリポジトリからバンドルによって到着したチェンジセットを制御する．

`[acl.allow]` セクションはチェンジセットのリポジトリへの追加を許可されているユーザを設定する．このセクションが存在しない場合，明示的に拒否されていないすべてのユーザは許可される．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すべてのユーザは拒否される．（すなわち，空のセクションはすべてのユーザの拒否という意味になる．）

`[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拒否するユーザを設定する．このセクションが存在しないか，空の場合はどのユーザも拒否されない．

`[acl.allow]` と `[acl.deny]` セクションの構文は同一である．各々のエントリの左辺はファイルまたはディレクトリにマッチする glob パターンで，リポジトリルートからの相対パスである．右辺はユーザ名である．

以下の例では，`docwriter` というユーザは，リポジトリの `docs` サブツリーにしか push できない．また `intern` は `source/sensitive` 以外ならばどのディレクトリのどのファイルにも変更を push することができる．

```
[acl.allow]
docs/** = docwriter

[acl.deny]
source/sensitive/** = intern
```

テストと問題解決

acl フックをテストしたい場合は，Mercurial のデバッグ出力を有効にして実行すると良い．サーバ上で実行するのであれば、--debug オプションを渡すのは不便であったり、不可能であったりすることがある。このため、デバッグ出力は hgrc でも有効にできることを記憶しておくべきである．

```
[ui]
debug = true
```

デバッグ出力が有効の場合，acl フックは，特定のユーザを許可あるいは拒否した理由を知るのに十分な情報を出力する．

10.7.2 bugzilla—Bugzilla との結合

bugzilla エクステンションは，Bugzilla で管理されているバグへ，コミットコメントの中でバグ ID を参照している場合コメントを追加する．このフックは共有サーバにもインストールできるので，リモートユーザが変更を push した場合にも動作する．

このフックは次のようにバグへコメントを追加する．（コメントの内容は設定可能である．これについては下記を参照.）

```
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
```

このフックは，チェンジセットがバグを参照した場合，バグの更新を自動化できるところに価値がある．このフックをきちんと設定すれば，Bugzilla バグを閲覧しているユーザが，バグから直ちに関係するチェンジセットを参照できるようになる．

このフックのコードを出発点として，別の Bugzilla との結合を行うことも可能である．いくつかの例を挙げる：

- サーバにプッシュされたチェンジセットすべてにコミットコメントに有効なバグ ID があることを要求する．この場合，フックを pretxncommit フックとして設定する必要がある．これによって，バグ ID を含まない変更を拒否することができる．
- 到着するチェンジセットに，コメント追加に加えて，バグの**ステート**を自動的に変更するようにできる．例えば，フックが “fixed bug 31337” のような文字列を認識して，bug 31337 のステートを “requires testing” に変更するなどが考えられる．

bugzilla フックの設定

このフックは次の例のようにサーバ上の hgrc で incoming フックと設定すべきである．

```
[hooks]
incoming.bugzilla = python:hgext.bugzilla.hook
```

このフックの特別な性質，Bugzilla がこのような結合を念頭に書かれていないことによって，このフックの設定は複雑なプロセスとなる．

設定を開始する前に，フックを実行するホスト上で Python 用の MySQL バインディングをインストールする必要がある．実行環境用にバイナリパッケージが用意されていなければ，ソースファイルを [?] からダウンロードすることができる．

このフックの設定情報は，hgrc ファイルの [bugzilla] セクションにある．

version サーバへインストールされた Bugzilla のバージョン．Bugzilla の使用するデータベースのスキーマは時に変更されるため，フックはどのスキーマが使用されるのかを知る必要がある．現時点では Bugzilla2.16 だけがサポートされている．

host Bugzilla データを格納している MySQL サーバのホストネーム．データベースは bugzilla フックを実行するホストから接続可能に設定されていなければならない．

user MySQL サーバに接続するユーザ名．データベースは bugzilla フックを実行するホスト上からこのユーザの接続を許可するように設定されていなければならない．このユーザは Bugzilla テーブルにアクセスし，変更できる権限がなければならない．この項目のデフォルト値は MySQL データベースでの Bugzilla ユーザの標準名 bugs である．

password 上記のユーザの MySQL パスワード．平文で保存されるため，権限のないユーザがこの hgrc ファイルを確実に読めないようにしておく必要がある．

db MySQL サーバ上の Bugzilla データベースの名前．この項目のデフォルト値は bugs この項目のデフォルト値は Bugzilla がデータを保存する MySQL データベースの標準名 bugs である．

notify フックからバグにコメントが追加された時，Bugzilla から講読者に通知メールが送られるようにしたい場合は，フックがデータベースを更新した場合は常にコマンドが実行されるように設定しなければならない．実行するコマンドは，Bugzilla をどこにインストールしたかに依存する．Bugzilla を /var/www/html/bugzilla にインストールした場合，典型的なコマンドは次のようになる：

```
1 cd /var/www/html/bugzilla && ./processmail %s nobody@nowhere.com
```

Bugzilla processmail プログラムは bug ID (フックが “%s” を bug ID に置換する．) と email アドレスを取る．このプログラムはまた実行されるディレクトリ内でいくつかのファイルへ書き込みを必要とする．Bugzilla とフックが同じマシン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たサーバ上で processmail を実行する方法を見つける必要がある．

コミット者の名前を Bugzilla のユーザ名へマップする

デフォルトでは bugzilla フックはバグを更新する Bugzilla ユーザ名としてチェンジセットのコミッタの email アドレスを使おうとする．この挙動が望ましくない場合は，[usermap] セクションを設定することでコミッタの email アドレスを Bugzilla のユーザ名にマップすることができる．

[usermap] セクションの各々の項目は，左辺に email アドレス，右辺に Bugzilla ユーザ名を持つ．

```
1 [usermap]
2 jane.user@example.com = jane
```

[usermap] データを通常の hgrc ファイルに保存することも bugzilla フックに外部の usermap ファイルを読むように指示することもできる．後者の場合，例えば usermap データをユーザが変更可能なリポジトリに置くことも可能である．これはユーザに usermap エントリの管理を任せることになる．メインの hgrc ファイルは次のようになる．：

```
1 # regular hgrc file refers to external usermap file
2 [bugzilla]
3 usermap = /home/hg/repos/userdata/bugzilla-usermap.conf
```

バグに追加された文字列を設定する

フックが追加するコメント文字列は Mercurial テンプレートとして指定することができる．hgrc のエントリ ([bugzilla] セクションに含まれているものを含む) で挙動をコントロールできる．

strip URL 生成用の部分パス名を作るためにリポジトリのパス名から取り除かれる先行要素の数．サーバ上の/home/hg/repos にリポジトリ群があり，/home/hg/repos/app/tests というリポジトリを持っている場合，strip を 4 にすると部分パス名 app/tests が得られる．フックはこの部分パスをテンプレートの webroot に適用する．

template テンプレートに使用されるテキスト．通常のチェンジセット関連の変数に加えて，hgweb（上記の hgweb の設定値）と webroot（上記の strip を使って作ったパス）が利用できる．

さらに，baseurl 項目を hgrc の [web] セクションに追加できる．これはテンプレートの拡張時に bugzilla フックにより Bugzilla コメントからチェンジセットを参照する際の URL 生成のベース文字列として利用される．

```
[web]
baseurl = http://hg.domain.com/
```

bugzilla フックの設定情報の例を示す．

```
[bugzilla]
host = bugzilla.example.com
password = mypassword
version = 2.16
# server-side repos live in /home/hg/repos, so strip 4 leading
# separators
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}
```

テストと問題解決

bugzilla フックを設定する際に Bugzilla の processmail スクリプトの実行とコミッター名のユーザ名へのマッピングがよく問題になる．

10.7.2節で，サーバ上で Mercurial を動作させているユーザと processmail スクリプトを動かしているユーザが同じだと述べた．processmail スクリプトは Bugzilla に設定ディレクトリ内のファイルに書き込ませることがある． Bugzilla の設定ファイルは通常，ウェブサーバを起動しているユーザの所有である．

sudo コマンドを使うことで processmail を適切なユーザの権限で実行することができる．sudoers ファイルの記述例を示す．

```
hg_user = (httpd_user) NOPASSWD: /var/www/html/bugzilla/processmail-wrapper %s
```

この例では，ユーザ hg_user が processmail-wrapper プログラムをユーザ httpd_user の権限の元で実行することができる．

このラッパースクリプトから実行する間接実行は必要である．processmail は Bugzilla をどこにインストールしてもカレントディレクトリから実行されるためである．実行制限の種類を sudoers ファイルに記述する．ラッパースクリプトの内容はシンプルなものである．

```
#!/bin/sh
cd `dirname $0` && ./processmail "$1" nobody@example.com
```

processmail に渡す email アドレスはどんなものでもよい．

[usermap] が正しく設定されていない場合，変更をサーバにプッシュする際に bugzilla フックからのエラーメッセージがユーザに送られる．エラーメッセージは

```
cannot find bugzilla user id for john.q.public@example.com
```

のような内容である．これは，コミッタのアドレス john.q.public@example.com が有効な Bugzilla ユーザネームでなく，また有効な Bugzilla ユーザ名へのマップファイルである [usermap] にも記述がないという意味である．

10.7.3 notify—メールで通知を行う

Mercurial の組み込みウェブサーバは全てのリポジトリの変更の RSS フィードを提供するが，変更の通知をメールで受け取ることを好むユーザも多い．notify notify フックは，チェンジセットが到着した時に関心を持つユーザのメールアドレスに通知を送る．

bugzilla と同様に notify フックもテンプレートを使って送信される通知メッセージの内容をカスタマイズすることができる．

デフォルトでは notify フックは送信される全てのチェンジセットの diff を含む．この diff のサイズの上限を設定したり，送信自体を停止することができる．diff を送信すると，講読者が URL をクリックすることなく変更をすぐにレビューできる利点がある．

notify フックの設定

notify フックを設定し，到着したチェンジセット毎や（一度の pull や push で到着した）一連のチェンジセット毎に email を送信することができる．

```
[hooks]
# 一連の変更毎にメールを送信する
changegroup.notify = python:hgext.notify.hook
# 変更一つ毎にメールを送信する
incoming.notify = python:hgext.notify.hook
```

このフックの設定は hgrc ファイルの [notify] セクションに書く．

test デフォルトではこのフックはメールを全く送信しない．その代わり，送信されのと同じ内容を表示する．この項目を false に設定するとメールが送信される．デフォルトでメール送信がオフにされている理由は，意図する通りにこの設定を行うためには数回の試行が必要なためである．デバッグ中に講読者に壊れた通知を送るのはスパムまがいで好ましくない．

config 講読者情報を含む設定ファイルへのパス．メインの hgrc と分離することで，リポジトリ毎にリポジトリ内で管理できる．協力者はリポジトリをクローンし，講読者をアップデートして変更をサーバに push できる．

strip リポジトリに講読者がいるかどうか判定する際に，リポジトリのパスから除去する先行部分をパス区切り文字で示した数．例えば，サーバでリポジトリ群が/home/hg/repos に置かれており， notify が /home/hg/repos/shared/test というリポジトリを対象とする時，strip を 4 に設定すると notify はパスを shared/test に短縮し，これを用いて講読者のマッチを行う．

template メッセージを送信する時に使われるテキストのテンプレート．メッセージのヘッダとボディ両方を設定することができる．

maxdiff メッセージの末尾に添付される diff データの最大行数．diff がこの値より大きい場合は切り詰められる．デフォルトでは 300 行．この値を 0 にすることで通知 email への diff の添付を抑制することができる．

sources 対象チェンジセットのソースのリスト．例えば，このリストでリモートユーザがサーバ経由でリポジトリへプッシュした変更に関してのみ通知を行うように制限することができる．ここで指定できるソースについては 10.8.3 節を参照のこと．

[web] セクションの baseurl 項目を設定しているなら，これを webroot としてテンプレート内で使うことができる．

notify 設定の例を示す．

```
[notify]
# 実際にメールを送信するか
test = false
# 講読者データが通知リポジトリ内にある
config = /home/hg/repos/notify/notify.conf
# リポジトリ群はサーバの /home/hg/repos にあるので 4 つの "/" 文字をストリップする
strip = 4
template = X-Hg-Repo: {webroot}
  Subject: {webroot}: {desc|firstline|strip}
  From: {author}

  changeset {node|short} in {root}
  details: {baseurl}{webroot}?cmd=changeset;node={node|short}
  description:
    {desc|tabindent|strip}

[web]
baseurl = http://hg.example.com/
```

生成されるメッセージは次のようになる：

```
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...]
```

テストと問題解決

デフォルトでは notify エクステンションは**メールを送信しない**ことに留意すること．実際に送信させるためには明示的に test を false に設定しなければならない．この設定をしなければ，送信するのと同じメッセージを表示するだけである．

10.8 フック作製者への情報

10.8.1 プロセス内フックの実行

プロセス内フックは次のような形式の引数を伴って呼び出される：

```
def myhook(ui, repo, **kwargs):
    pass
```

ui パラメータは mercurial.ui.ui のオブジェクトである．repo パラメータは mercurial.localrepo.localrepository のオブジェクトである．**kwargs パラメータの名前と値は，呼び出されるフックに依存し，次のような共通した特徴を持つ．

- パラメータが node または parent*N* と名付けられた場合，16 進数のチェンジセット ID を持つ．“ヌルチェンジセット ID” を表すために 0 ではなく空の文字列が用いられる．
- パラメータが url と名付けられた場合，リモートリポジトリが特定されるならばその URL を持つ．
- ブール値を持つパラメータは Python の bool オブジェクトとして表現される．

プロセス内フックは，リポジトリのルートで実行される外部フックと異なりプロセスのワーキングディレクトリの変更なしに呼び出される．プロセス内フックはプロセスワーキングディレクトリを変更してはならない．さもなければ，Mercurial API への呼び出しは全て失敗する．

フックがブール値 “false” を返した場合，フックの実行は成功したと見なされる．ブール値 “true” を返した場合および例外を発生させた場合は失敗したと見なされる．呼び出し既約は “失敗した時は教える” と覚えるとよい．

Python フックに渡されるチェンジセット ID は，Mercurial API が通常用いるバイナリハッシュ値ではなく 16 進数文字列であることに注意．ハッシュ値を 16 進数文字列からバイナリに変換するには mercurial.node.bin 関数を用いる．

10.8.2 フックの外部実行

外部フックは Mercurial を実行しているシェルで実行される．変数の置換やコマンドのリダイレクトなどのシェルの機能が利用できる．外部フックは，Mercurial の実行と同じディレクトリ内で実行されるプロセス内フックとは異なり，リポジトリのルートディレクトリで実行される．

フックパラメータはフックに環境変数として渡される．各々の環境変数名は大文字に変換され，接頭辞 “HG_” が付けられる．例えば “node” というパラメータが使われたとすると，このパラメータを表す環境変数名は “HG_NODE” となる．

ブール値パラメータについては，真が “1”，偽が “0” と表現される．HG_NODE，HG_PARENT1，HG_PARENT2 という環境変数が定義されているとき，これらは 16 進数の文字列で表されたチェンジセット ID を含む．“ヌルチェンジセット ID” を表現するためにゼロでなく空の文字列が用いられる．環境変数が HG_URL であるとき，リモートリポジトリが特定できれば，その URL が入る．

フックがステータス 0 で終了した場合は成功と見なされる．0 以外のステータスで終了した場合は実行が失敗したと見なされる．

10.8.3 チェンジセットのソースを調べる

ローカルなリポジトリ・他のリポジトリ間のチェンジセットの転送に関連したフックは，遠隔地の情報を得ることができる．Mercurial は**どのように**変更が転送されるか，また多くの場合**どこへ**，あるいは**どこから**転送されるのかも知ることができる．

チェンジセットのソース

Mercurial はフックにチェンジセットの内容，リポジトリ間の転送方法などを通知する． Mercurial は通知のために `source` という Python パラメータまたは `HG_SOURCE` という環境変数を用いる．

`serve` チェンジセットはリモートリポジトリから http または ssh で転送された．

`pull` チェンジセットは 2 つのリポジトリ間を pull によって転送された．

`push` チェンジセットは 2 つのリポジトリ間を push によって転送された．

`bundle` チェンジセットはバンドルによって転送された．

変更の行き先—リモートリポジトリの URL

Mercurial は，可能であるならばリポジトリ間のチェンジセットデータの転送の相手の情報をフックに通知する．Mercurial は `url` という Python パラメータまたは `HG_URL` という環境変数を使って通知を行う．

この情報はいつも既知である．http または ssh でサービスされているリポジトリからフックが起動された場合，Mercurial はリモートリポジトリがどこにあるのかを通知することはできないが，クライアントがどこから接続しているのかは知ることができる．

その場合，URL は次のいずれかの形式を取る：

- `remote:ssh:`*`ip-address`*—ある IP アドレス上のリモート ssh クライアント
- `remote:http:`*`ip-address`*—ある IP アドレス上の http クライアント．もしクライアントが SSL を使用している場合，`remote:https:`*`ip-address`* のような形式になる．
- Empty—リモートクライアントに関する情報がない．
- `remote:ssh:`*`ip-address`*—与えられた IP アドレスのリモート ssh クライアント．
- `remote:http:`*`ip-address`*—与えられた IP アドレスのリモート http クライアント．クライアントが SSL を使っている場合は `remote:https:`*`ip-address`* の形式になる．
- 空白—リモートクライアントについての情報は得られなかった．

10.9 フック参照

10.9.1 `changegroup`—リモートチェンジセットが追加された後

このフックは，“`hg pull`” または “`hg unbundle`” などで既存のチェンジセットグループがリポジトリに追加された後に実行される．このフックは，1 つ以上のチェンジセットを追加するオペレーションにつき一度実行される．グループに関係なくチェンジセット 1 つ毎に実行される `incoming` フックとは対照的である．

このフックを使って追加されたチェンジセットに対する自動ビルドやテストを起動することや，新たな変更を持つリポジトリの講読者に通知することができる．

このフックへのパラメータ：

`node` チェンジセット ID．追加されたグループの最初のチェンジセットの ID．この ID を含み `tip` までの間のチェンジセット全ては一回の “`hg pull`”，“`hg push`” または “`hg unbundle`” で追加された．

`source` 文字列．これらの変更のソース．詳細については 10.8.3を参照のこと．

`url` URL．リモートリポジトリが特定できる場合はそのアドレス．詳細については 10.8.3を参照のこと．

参考：`incoming` (10.9.3節), `prechangegroup` (10.9.5節), `pretxnchangegroup` (10.9.9節)

10.9.2 commit—新しいチェンジセットが作成された後

このフックは新たなチェンジセットが作られた後に呼ばれる．
このフックへのパラメータ：

node チェンジセット ID．新たに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．新たにコミットされたチェンジセットの 1 つ目の親のチェンジセット ID.

parent2 チェンジセット ID．新たにコミットされたチェンジセットの 2 つ目の親のチェンジセット ID.

参考：precommit (10.9.6節), pretxncommit (10.9.10節)

10.9.3 incoming—リモートチェンジセットが追加された後

このフックは “hg push” コマンドなどで既存のチェンジセットがリポジトリに追加された後で実行される．チェンジセットのグループが一度に追加された場合，各々のチェンジセットについて一度ずつこのフックが呼び出される．
このフックは changegroup(10.9.1節) と同じ目的で使うことができる．違いは changegroup がチェンジセットグループ全体に対して一度呼ばれる点である．
このフックへのパラメータ：

node チェンジセット ID．新たに追加されるチェンジセットの ID.

source 文字列．これらの変更のソース．詳細については 10.8.3を参照のこと．

url URL．リモートリポジトリが特定できる場合にはそのアドレス．詳細については 10.8.3を参照のこと．

参考：changegroup (10.9.1節) prechangegroup (10.9.5節), pretxnchangegroup (10.9.9節)

10.9.4 outgoing—チェンジセットが波及した後

このフックはチェンジセットのグループが “hg push” や “hg bundle” コマンドなどによって外部に波及した後で実行される．
このフックを使って管理者は pull された変更を知ることができる．
このフックへのパラメータ：

node チェンジセット ID．送信されたチェンジセットグループの最初のチェンジセット ID.

source 文字列．操作元 (10.8.3節を参照)．リモートクライアントが変更をリポジトリから pull すると source が serve となる．リポジトリから変更を入手したクライアントがローカルであれば，source はクライアントの行った動作に応じて bundle，pull，push のいずれかとなる．

url URL．リモートリポジトリが特定できる場合はそのアドレス．詳細については 10.8.3節を参照のこと．

参考： preoutgoing (10.9.7節)

10.9.5 prechangegroup—リモートチェンジセットがが追加される前

この制御フックは Mercurial が一連のチェンジセットを別のリポジトリに追加する前に実行される．
このフックは追加されるチェンジセットの情報は何も持たない．なぜならこのフックはチェンジセットの送信が始まる前に実行されるからだ．フックの実行が失敗した場合はチェンジセットは送信されない．
このフックを使って，外部の変更がリポジトリに追加されないようにすることもできる．例えば，このフックをサーバで提供されるブランチを一時的または永続的に “凍結” し，ローカルな管理者はリポジトリを変更できるが，リモートユーザはそのリポジトリに push できないようにできる．
このフックへのパラメータ：

`source` 文字列．変更のソース．詳細については 10.8.3を参照．

`url` URL．リモートリポジトリが特定される場合はそのアドレス．詳細については 10.8.3を参照．

参考： `changegroup` (10.9.1節)， `incoming` (10.9.3節)， `pretxnchangegroup` (10.9.9節)

10.9.6 `precommit`—チェンジセットをコミットする前

このフックは Mercurial が新しいチェンジセットをコミットする前に実行される．実行は Mercurial がコミットされるファイル名，コミットメッセージ，コミット日時のようなコミットのためのメタデータを持つ前である．

このフックを使って，外部からのチェンジセットの取り込みを許可する一方で，新しいチェンジセットをコミットできないようにすることもできる．また，ビルドやテストを実行したり，それが成功した時のみコミットを行うようにすることもできる．

このフックへのパラメータ：

`parent1` チェンジセット ID．ワーキングディレクトリの 1 つ目の親のチェンジセット ID．

`parent2` チェンジセット ID．ワーキングディレクトリの 2 つ目の親のチェンジセット ID．

コミットが進行するとワーキングディレクトリの親は新しいチェンジセットの親となる．

参考：`commit` (10.9.2節)， `pretxncommit` (10.9.10節)

10.9.7 `preoutgoing`—チェンジセットを波及させる前に

このフックは Mercurial が送信するチェンジセットを識別する前に実行される．

このフックを用いて，他のリポジトリへ変更を送信しないようにすることもできる．

このフックへのパラメータ：

`source` 文字列．このリポジトリから変更を取得しようとする動作のソース（10.8.3を参照）．このパラーメタが取り得る値については， 10.9.4節の `outgoing` フックへの `source` パラメータの項目を参照のこと．

`url` URL．リモートリポジトリが特定できる場合はそのアドレス．詳細については 10.8.3を参照のこと．

参考： `outgoing` (10.9.4節)

10.9.8 `pretag`—チェンジセットにタグをつける前に

この制御フックはタグが作成される前に実行される．フックの実行が成功した場合はタグが作成される．フックの実行が失敗した場合はタグは作成されない．

このフックへのパラメータ：

`local` ブール値．新しいタグが現在のリポジトリにローカルなもの (`.hg/localtags` に保存される) か Mercurial でグローバルに管理されるもの (`.hgtags` に保存される) かを示す．

`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．生成されたタグの名前．

生成されたタグがリビジョンコントロールされている場合，`precommit` フックと `pretxncommit` フック (10.9.2節および 10.9.10) 節) の両方が実行される．

参考： `tag` (10.9.12節)

10.9.9 `pretxnchangegroup`—リモートチェンジセットの追加を完了する前に

この制御フックは，外部から新しいチェンジセットを追加するトランザクションが完了する前に実行される．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はリポジトリ内で永続的になる．フックの実行が失敗した場合，トランザクションはロールバックされ，チェンジセットのデータは消去される．

このフックはトランザクション中のチェンジセットのメタデータにアクセスすることができるが，このデータを使って永続的なことは一切してはならない．ワーキングディレクトリの変更もしてはならない．

このフックの実行中，他の Mercurial プロセスがリポジトリにアクセスすると，トランザクション中のチェンジセットを永続的なものと見なす可能性がある．適切な回避策を講じなければ，これが競合状態を引き起こす可能性がある．

このフックは一連のチェンジセットを自動的に排除するために使うこともできる．このフックの実行が失敗した場合，トランザクションがロールバックされる時にチェンジセット全体がリジェクトされる．

このフックへのパラメータ：

`node` チェンジセット ID．追加される一連のチェンジセットのうち，最初のチェンジセット ID．これと `tip` の間の全てのチェンジセットが一度の “`hg pull`”, “`hg push`” または

`source` 文字列．これらの変更のソース．詳細については 10.8.3節を参照のこと．

`url` URL．既知のリモートリポジトリの場所．詳細については 10.8.3節を参照のこと．

参考： `changegroup` (10.9.1節), `incoming` (10.9.3節), `prechangegroup` (10.9.5節)

10.9.10 `pretxncommit`—新しいチェンジセットのコミットを完了する前に

この制御フックは新たなコミットのためのトランザクションの完了前に実行される．このフックの実行が成功した場合，トランザクションが完了され，チェンジセットはリポジトリ内で永続的になる．フックの実行が失敗した場合はトランザクションはロールバックされ，コミットデータは消去される．

このフックはトランザクション中のチェンジセットのメタデータにアクセスすることができるが，このデータを使って永続的なことは一切してはならない．ワーキングディレクトリの変更もしてはならない．

このフックの実行中，他の Mercurial プロセスがリポジトリにアクセスすると，トランザクション中のチェンジセットを永続的なものと見なす可能性がある．適切な回避策を講じなければ，これが競合状態を引き起こす可能性がある．

このフックへのパラメータ：

`node` チェンジセット ID．新たにこ見んとされたチェンジセットの ID.

`parent1` チェンジセット ID．新たにコミットされたチェンジセットの 1 つ目の親のチェンジセット ID.

`parent2` チェンジセット ID．新たにコミットされたチェンジセットの 2 つ目の親のチェンジセット ID.

参考：`precommit` (10.9.6節)

10.9.11 `preupdate`—ワーキングディレクトリのアップデートまたはマージの前に

この制御フックはワーキングディレクトリのアップデートまたはマージが始まる前に実行される． Mercurial の通常のアップデート前チェックがアップデートまたはマージを実行できると判断した場合にのみ実行される．フックの実行が成功した場合，アップデートまたはマージが行われる．失敗した場合はアップデートまたはマージは開始されない．

このフックへのパラメータ：

`parent1` チェンジセット ID．ワーキングディレクトリがアップデートされる親 ID. ワーキングディレクトリがマージされる場合，親 ID は変更されない．

`parent2` チェンジセット ID．ワーキングディレクトリがマージされる場合のみセットされる．ワーキングディレクトリがマージされるリビジョン ID.

参考： `update` (10.9.13節)

10.9.12 tag—チェンジセットにタグ付けした後に

このフックはタグが生成された後に実行される．
このフックへのパラメータ：

`local` ブール値．新しいタグが現在のリポジトリにローカルなもの (`.hg/localtags` に保存される) か Mercurial にグローバルに管理されるもの (`.hgtags` に保存される) かを示す．

`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．生成されたタグの名前．

生成されたタグがリビジョンコントロールされている場合，`commit` フック (10.9.2節) はこのフックの前に実行される．
参考： `pretag` (10.9.8節)

10.9.13 update—ワーキングディレクトリを更新またはマージした後に

このフックはワーキングディレクトリのアップデートまたはマージが完了した後に実行される．マージは失敗することもある（外部の `hgmerge` コマンドはファイル内のコンフリクトを解決できないことがある）ので，このフックはアップデートまたはマージが正常に完了したかどうかを問い合わせる．

`error` ブール値．アップデートまたはマージが正常に完了したかどうかを示す．

`parent1` チェンジセット ID．ワーキングディレクトリがアップデートされる親 ID．ワーキングディレクトリがマージされる場合，親 ID は変化しない．xxx

`parent2` チェンジセット ID．ワーキングディレクトリがマージされる時のみセットされる．ワーキングディレクトリがマージされるリビジョン ID．

参考: `preupdate` (10.9.11節)

Chapter 11

Mercurialの出力のカスタマイズ

Mercurial には情報の表示をコントロールするための強力な機構がある．この機構はテンプレートをベースとしている．テンプレートによってあるコマンドから特別の出力を行ったり，内蔵のウェブインタフェースの見た目をカスタマイズすることができる．

11.1 用意された出力スタイルの利用

すぐに使うことのできるいくつかのスタイルが Mercurial に同梱されている．スタイルとは缶詰されたテンプレートであり， Mercurial のインストールされたどこかのマシンで誰かが書いてインストールしたものである．

Mercurial 同梱のスタイルを見る前に，通常の出力を見てみよう．

```
$ hg log -r1
changeset:   1:7f36ee516f32
tag:         mytag
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:17 2009 +0000
summary:     added line to end of <<hello>> file.

```

この出力には有益な情報が含まれているが，1 つのチェンジセット毎に 5 行を使うなど，多くのスペースを費やす．`compact` スタイルは疎らな方法を使うことでこれを 3 行に減らす．

```
$ hg log --style compact
3[tip]   3d826c485bdf   2009-06-09 06:07 +0000   bos
  Added tag v0.1 for changeset a51282aec6e8

2[v0.1]   a51282aec6e8   2009-06-09 06:07 +0000   bos
  Added tag mytag for changeset 7f36ee516f32

1[mytag]   7f36ee516f32   2009-06-09 06:07 +0000   bos
  added line to end of <<hello>> file.

0   5785e6d3a00e   2009-06-09 06:07 +0000   bos
  added hello

```

`changelog` スタイルは Mercurial のテンプレートエンジンの威力を知るいい例である．このスタイルは GNU プロジェクトの changelog ガイドライン [**?**] に従おうとする．

```
$ hg log --style changelog
2009-06-09  Bryan O'Sullivan  <bos@serpentine.com>

        * .hgtags:
        Added tag v0.1 for changeset a51282aec6e8
        [3d826c485bdf] [tip]

        * .hgtags:
        Added tag mytag for changeset 7f36ee516f32
        [a51282aec6e8] [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.
        [7f36ee516f32] [mytag]

        * hello:
        added hello
        [5785e6d3a00e]

```

Mercurial のデフォルト出力スタイルが `default` と名付けられているのは驚くに値しない.

11.1.1 デフォルトスタイルの設定

Mercurial の全てのコマンドで用いられる出力スタイルは `hgrc` ファイルを編集することで設定でき，好きな名前を付けることができる.

```
[ui]
style = compact
```

自分でスタイルを書いたときは，スタイルファイルのパスを追加したり，ファイルを Mercurial が発見できる場所 (典型的には，Mercurial のインストールディレクトリの `templates` サブディレクトリ) にコピーすることで利用可能になる.

11.2 スタイルとテンプレートをサポートするコマンド

Mercurial の "`log` 系" の全てのコマンド: "`hg incoming`", "`hg log`", "`hg outgoing`", および "`hg tip`" はスタイルとテンプレートを利用している.

このマニュアルで書いているように，これまでのところ，これらのコマンドだけがスタイルとテンプレートをサポートしている．これらがカスタマイズ可能な出力が必要な最も重要なコマンドであるため，Mercurial のユーザコミュニティから他のコマンドにスタイルとテンプレートサポートを適用可能にせよというプレッシャーはほとんどない.

11.3 テンプレートの基本

最も単純な Mercurial テンプレートはテキストの断片である．テキストの一部は必要に応じて**展開**されたり新しいテキストに置換され，別の部分は不変である.

さらに続ける前に Mercurial 標準出力の例をもう一度見てみよう．

```
$ hg log -r1
changeset:   1:7f36ee516f32
tag:         mytag
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:17 2009 +0000
summary:     added line to end of <<hello>> file.

```

ここで同じコマンドを出力を変化させるためにテンプレートを使ってみよう．

```
$ hg log -r1 --template 'i saw a changeset\n'
i saw a changeset
```

上の例は最も単純なテンプレートを示した; 静的なテキストだけからなり，各チェンジセットについて一度だけ出力を行う．“hg log” コマンドに--template オプションを使うと Mercurial は各々のチェンジセットを出力するときに，与えられたテキストをテンプレートとして用いる．

テンプレート文字列は “\n” で終ることに注意．これは Mercurial にテンプレートの各要素の末尾に改行を出力するよう指示する**エスケープシーケンス**である．この改行を省略すると Mercurial は各要素を繋げて出力する．エスケープシーケンスのより詳細な説明については 11.5節を参照のこと．

常に固定文字列を出力するテンプレートはあまり有用であるとは言えない．もう少し込み入ったテンプレートを試そう．

```
$ hg log --template 'i saw a changeset: {desc}\n'
i saw a changeset: Added tag v0.1 for changeset a51282aec6e8
i saw a changeset: Added tag mytag for changeset 7f36ee516f32
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
```

テンプレート内の “{desc}” という文字列は出力では，各々のチェンジセットの説明に置換される．Mercurial は中括弧 (“{” と “}”) で囲まれたテキストを見つけると中括弧とテキストを内部のテキストを展開したものに置換しようと試みる．文字として中括弧を印字したい場合は 11.5節で示すようにエスケープする必要がある．

11.4 テンプレートの共通キーワード

単純なテンプレートは以下のようなキーワードを使って直ちに書くことができる．

author 文字列．チェンジセットの著者．

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 の場合は空欄となる．

date 日時情報．チェンジセットがコミットされた日時．これは人間の可読な形式では**ない**．必ず適切なフィルタを呼び出して変換する必要がある．フィルタに関するより詳細な説明は 11.6 節を参照のこと．日時は 2 つの数字の組み合わせで表される．最初の数字は UTC での Unix タイムスタンプ (1970 年 1 月 1 日からの秒数) で，2 番目の数字はコミッタのタイムゾーンの UTC からの時差を秒数で表したものである．

desc 文字列．チェンジセットの説明文．

files 文字列のリスト．このチェンジセットで変更，追加，削除された全てのファイルの名前．

```
$ 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: 7f36ee516f32d4bb403be52cab4f15f4cb884222
$ 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
```

Figure 11.1: テンプレートキーワードの使用

`file_adds` 文字列のリスト．このチェンジセットで追加されたファイルの名前．

`file_dels` 文字列のリスト．このチェンジセットで削除されたファイルの名前．

`node` 文字列．チェンジセットの識別ハッシュを 40 文字の 16 進数で示したもの．

`parents` 文字列のリスト．チェンジセットの親．

`rev` 整数．リポジトリローカルのチェンジセットリビジョン番号．

`tags` 文字列のリスト．チェンジセットに関連づけられた任意のタグ．

いくつか試してみればこれらのキーワードにどのような効果があるのかを知ることができる．結果を図 11.1に示す．

既に述べたように，date キーワードは可読な出力を行わないため，特別な取り扱いが必要である．その目的で *filter* を使用するが，より詳細には 11.6節で取り扱う．

```
$ hg log -r1 --template 'date: {date}\n'
date: 1244527637.00
$ hg log -r1 --template 'date: {date|isodate}\n'
date: 2009-06-09 06:07 +0000
```

11.5 エスケープシーケンス

Mercurial のテンプレートエンジンは文字列で最もよく使用されるエスケープシーケンスを認識する．エンジンはバックスラッシュ(“\”) を見つけると後続の文字を含む 2 文字を以下のような 1 文字で置換する．

\\ バックスラッシュ, “\”, ASCII 134.

\n 改行, ASCII 12.

\r 復帰, ASCII 15.

\t タブ, ASCII 11.

\v 垂直タブ, ASCII 13.

\{ 開き中括弧, “{”, ASCII 173.

\} 閉じ中括弧, “}”, ASCII 175.

上に示したように，テンプレートの展開の際に文字 “\”, “{”, または “{” を入れたい場合はエスケープすることが必要である．

11.6 結果を改変するフィルタキーワード

テンプレート展開の結果には，そのままでは利用しづらいものもある．Mercurial が提供する *filters* オプションを用いて，展開されたキーワードを読みやすく加工することができる．日時を可読にするためによく用いられる isodate フィルタの動作例については既に見てきた．

以下に示すのは Mercurial がサポートするフィルタの中で最もよく使われるものである．いくつかのフィルタは任意の文字列に対して適用可能な一方，他のものは特定の状況においてのみ利用可能である．各々のフィルタの名前は，利用可能な状況を示す表示で始まり，得られる効果の説明が続く．

addbreaks 任意のテキスト．XHTML タグ “
” を最終行以外の各行の末尾に追加する．例えば “foo\nbar” は “foo
\nbar” となる．

age date キーワード．日時の経過時間を現在を起点に整形する．結果は “10 minutes” のようになる

basename 任意のテキストだが，files キーワードやその関連キーワードに対して最も有用である．テキストをパスとして扱い，ベースネームを返す．例えば “foo/bar/baz” は “baz” となる．

date date キーワード．日時を Unix の date コマンドの出力にタイムゾーン情報を追加した形式に整形する．結果は “Mon Sep 04 15:13:13 2006 -0700” のようになる．

domain 任意のテキスト．author キーワードでの使用に最も有用．最初に現れる email アドレス形式の文字列を見つけ，ドメイン部分だけを抜き出す．例えば “Bryan O’Sullivan <bos@serpentine.com>” は “serpentine.com” となる．

email 任意のテキスト．author キーワードで最も有用．最初に現れる email アドレス形式の文字列を抜き出す．例えば “Bryan O’Sullivan <bos@serpentine.com>” は “bos@serpentine.com” となる．

escape 任意のテキスト．XML/XHTML 文字 “&”, “<” および “>” を XML エンティティで置換する．

fill68 任意のテキスト．テキストを 68 桁に収まるように整形する．これは 80 桁に固定された端末での表示用に tabindent フィルタを使う際に便利である．

fill76 任意のテキスト．76 桁に収まるように整形する．

firstline 任意のテキスト最初の行だけを出力し，後続の行は一切出力しない．

hgdate date キーワード．日時を可読な数字のペアに整形する．“1157407993252000” のような文字列を出力する．

isodate date キーワード．日時を ISO 8601 フォーマットに整形する．出力は “2006-09-04 15:13:13 -0700” のようになる．

obfuscate 任意のテキスト．author キーワードと併せて利用する場合最も有用．入力されたテキストを XML エンティティのシーケンスとして出力する．これは画面をスクレイピングする間抜けなスパムボットを避ける働きがある．

person 任意のテキスト．author キーワードと合わせて使う場合もっとも有用．email アドレスの前のテキストを抽出する．例えば “Bryan O’Sullivan <bos@serpentine.com>” は “Bryan O’Sullivan” となる．

rfc822date date キーワード．日時を email ヘッダと同じ形式で整形する．“Mon, 04 Sep 2006 15:13:13 -0700” のような文字列を生成する．

short チェンジセットハッシュ．短形式のチェンジセットハッシュ，すなわち 12 バイトの 16 進数文字列を生成する．

shortdate date キーワード．年月日を整形する．“2006-09-04” のような文字列を生成する．

strip 任意のテキスト．文字列の前後の空白部分を除去する．

tabindent 任意のテキスト．タブ文字で始まらない行すべてを出力する．

urlescape 任意のテキスト．URL パーサから見て “特別” な文字のエスケープを行う．例えば foo bar は foo\%20bar となる．

user 任意のテキスト．author キーワードと共に使った場合に最も有用．email アドレスからユーザ名の部分を抜き出す．例えば “Bryan O’Sullivan <bos@serpentine.com>” は “bos” となる．

> **Note:** フィルタを適用不能なデータに対して使おうとすると Mercurial はエラーを起こし，Python からの例外を出力する．例えば desc キーワードの出力に isodate フィルタを適用するのはいい考えとは言えない．

11.6.1 組み合わせフィルタ

複数のフィルタを組み合わせ，望みの出力を簡単に作ることができる．ここでは例として，説明文を整理し，きれいに 68 桁に収まるように整形し，8 文字のインデントを行うフィルタチェーンを示す．(UNIX システムの習慣では，タブ幅は 8 桁である．)

```
$ 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.
```

テンプレートで最初の行をインデントするためには “\t” (タブ文字) を使う必要がある．なぜなら tabindent は最初の行**以外**のインデントを行うからである．

フィルタを組み合わせる時フィルタの順序が重要であることを頭に置いておく必要がある．最初のフィルタはキーワードの結果に対して適用され，2 番目のフィルタは最初のフィルタの結果に対して適用される．例を挙げると，fill68|tabindent と tabindent|fill68 の結果は全く違うものである．

11.7 テンプレートからスタイルへ

コマンドラインテンプレートは出力をフォーマットするための素早く単純な方法を提供する．テンプレートはメッセージを多く出力するようにもできるが，テンプレートに名前を付けるのは有益である．スタイルファイルは，ファイルに保存された名前を付けられたテンプレートである．

さらに，スタイルファイルを使うことで，Mercurial のテンプレートエンジンのコマンドライン--template オプションからでは利用できない力を最大限に引き出すことができる．

```
$ 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: 1244527637.00
$ hg log -r1 --template '{date|age}\n'
3 seconds
$ hg log -r1 --template '{date|date}\n'
Tue Jun 09 06:07:17 2009 +0000
$ hg log -r1 --template '{date|hgdate}\n'
1244527637 0
$ hg log -r1 --template '{date|isodate}\n'
2009-06-09 06:07 +0000
$ hg log -r1 --template '{date|rfc822date}\n'
Tue, 09 Jun 2009 06:07:17 +0000
$ hg log -r1 --template '{date|shortdate}\n'
2009-06-09
$ 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'
7f36ee516f32d4bb403be52cab4f15f4cb884222
$ hg log -r1 --template '{node|short}\n'
7f36ee516f32
```

11.7.1 最も単純なスタイルファイル

サンプルスタイルファイルはただの 1 行からなる:

```
$ echo 'changeset = "rev: {rev}\n"' > rev
$ hg log -l1 --style ./rev
rev: 3
```

これは Mercurial に “チェンジセットを出力する時は右のテキストをテンプレートとして使え” という指示を与える.

11.7.2 スタイルファイルの文法

スタイルファイルの文法規則は単純である.

- ファイルは一度に 1 行ずつ処理される
- 前後の空白は無視される
- 空行はスキップされる
- 行が “#” または “;” で始まる場合は，行全体がコメントとして扱われ，空白と同様にスキップされる.
- 行はキーワードで始まる．必ずアルファベットかアンダースコアで始まる必要があり，後続の文字はアルファベット，数字，アンダースコアのいずれかであってよい.（正規表現では `[A-Za-z_][A-Za-z0-9_]*` とマッチする）
- 次の要素は文字 “=” で，前後に空白が入っても良い.
- 数の釣り合ったシングルクォートまたはダブルクォートで括られる部分が続く場合はこれがテンプレートボディとして扱われる.
- 続く部分がクォート文字で始まらない場合はファイル名として扱われ，ファイルの内容がテンプレートボディとして使われる.

11.8 スタイルファイルの例

スタイルファイルの書き方を説明するためにいくつかの例題を挙げてみる．完全なスタイルファイルを用意してこれを説明するのではなく，とても単純な例から始めて，だんだんと複雑な例に進むことでスタイルファイルの通常の開発プロセスを模倣してみる.

11.8.1 スタイルファイルでの誤りを特定する

Mercurial がスタイルファイルで問題に直面すると，簡潔なエラーメッセージを表示する．このメッセージはその意味するところが分かっていればとても有益である.

```
$ cat broken.style
changeset =
```

`broken.style` で `changeset` キーワードを定義しようとしているが，定義の内容が全く与えられていない．このスタイルファイルを使おうとすると，Mercurial はすぐさまエラーを表示する.

```
$ hg log -r1 --style broken.style
abort: broken.style:1: parse error
```

このエラーメッセージは恐ろしげだが，対応するのはさほど難しくない．

- 最初の部分は “ギブアップ” の Mercurial 的な表現である．

```
abort: broken.style:1: parse error
```

- 次にエラーのあるスタイルファイル名が表示される

```
abort: broken.style:1: parse error
```

- ファイル名の後にエラーの起きた行番号が続く．

```
abort: broken.style:1: parse error
```

- 最後に問題点の説明が表示される

```
abort: broken.style:1: parse error
```

問題の説明は常に明快であるとは限らない．(この例のように．) しかし謎めいていたとしても，殆んどの場合，スタイルファイルの問題のある行を見ると，ごく些細な問題にすぎないものである．

11.8.2 リポジトリの特定

短い文字列を識別子に使って Mercurial リポジトリを特定したい場合はリポジトリ内の最初のリビジョンを使うことができる．

```
$ hg log -r0 --template '{node}'
65c9ab67c326e8d216dedbf0c379e5817b0ffab3
```

これはユニークであることが保証されているわけではないが，多くの場合に有用であることはいうまでもない．

- 完全に空のリポジトリではこの方法は使えない．空のリポジトリにはリビジョン ゼロすら存在しないためである．
- 独立した複数のリポジトリをマージしてリポジトリを作成し，これらのリポジトリが依然として存在するなどの（きわめて稀な）場合はこの方法は使えない．

この識別子の使用例を示す:

- サーバ上でリポジトリを管理するデータベース用のテーブルへのキーとして．
- {*repository ID*, *revision ID*} タプルの半分として．自動ビルドやその他の操作を行うときにこの情報をセーブしておき，後でビルドを再現することが必要となったら “リプレイ” を行えるようにする．

11.8.3 Subversion 出力の模倣

別のバージョン管理ツールである Subversion の出力をエミュレートしてみよう．

```
$ 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.
```

```
7
8  Signed-off-by: Sean Hefty <sean.hefty@intel.com>
9
10 ---------------------------------------------------------------------
```

Subversion の出力スタイルはかなり単純なので，出力からファイルへ hunk をコピー&ペーストし，Subversion が生成したテキストをテンプレートによって置換するのは簡単である．

```
1  $ cat svn.template
2  r{rev} | {author|user} | {date|isodate} ({date|rfc822date})
3
4  {desc|strip|fill76}
5
6  ---------------------------------------------------------------------
```

テンプレートによる出力が Subversion が生成する出力と異なる 2, 3 の例がある．

- Subversion は中括弧で囲まれた “可読な” 日時を出力する．（上の例では “Wed，27 Sep 2006”.） Mercurial のテンプレートエンジンはこのフォーマットで時刻とタイムゾーンを含まない形で日時を出力することができない．
- Subversion の “セパレータ” 行の出力をテンプレートを行幅一杯の “-” 文字で終えることでエミュレートしている．テンプレートエンジンの header キーワードを使ってセパレータ行を出力の最初の行として出力する．（下記を参照.）これによって Subversion と似通った出力を行っている．
- Subversion の出力はヘッダ内にコミットメッセージの行数のカウントを含んでいる．Mercurial ではこれを再現することはできない．テンプレートエンジンは現状では渡されたアイテムを数えるフィルタを作ることができない．

Subversion の出力を参考に，テキストを置換するキーワード，テンプレートの入ったフィルタを準備するのにはほんの１，２分しかかからなかった．スタイルファイルは単純にテンプレートを参照するのみである

```
1  $ cat svn.style
2  header = '---------------------------------------------------------------------\n\n'
3  changeset = svn.template
```

テンプレートファイルのテキストをクォートで括り，改行を “
n” でエスケープすることでスタイルファイルに直接追加することも可能ではある．しかしそうした場合，スタイルファイルがほとんど読めないほど複雑になってしまうだろう．テキストをスタイルファイルに入れるか，スタイルファイルから呼ばれるテンプレートファイルに入れるかは，可読性を考慮して決定すべきである．もしテキストを追加したことでスタイルファイルが大きくなり過ぎたり，乱雑になったりした場合は，テンプレートへの移動を考慮するとよい．

Chapter 12

Mercurial Queuesで変更を管理する

12.1 パッチ管理の問題

よくあるシナリオ：ソースからソフトウェアパッケージをインストールする必要がある．しかしパッケージを使う前にソースから直すべきバグを見つけた．変更を加えて，暫くパッケージのことは忘れている．数ヶ月後，新しいバージョンをインストールすることになった．もし新しいバージョンが，依然としてそのバグを持っていた場合，修正を古いソースから抽出して，新しいバージョンに適用しなければならない．これは退屈でしかも間違いやすい仕事だ．

これがパッチ管理問題のシンプルな例である．もしあなたが変更できない上流のソースツリーがあれば，アップストリームのツリーの上でローカルな変更をしなければならない．あなたはアップストリームソースに適用できるようにきっとこの変更を分離しておきたいと思うはずだ．

パッチ管理問題はいろいろな状況で起こる．おそらく最も明らかなのは，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にバグフィックスや新機能をパッチの形で貢献することであろう．

オープン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が彼らの環境で正しくビルドできるようにパッケージへ変更を加えることが多い．

もしメンテナンスしている変更がごくわずかなら，標準の `diff` コマンドと `patch` コマンドを使って単一のパッチを管理するのが簡単だ．（これらのツールについては 12.4の節を参照.）変更の数が増えてくると，単一のバグフィックスを含む個々のパッチを一塊の集合として管理することが現実味を帯びてくる．（各々のパッチは複数のファイルを変更するかもしれないが，目的は 1 つきりである.）修正したいバグがいくつもあったり，ローカルでの様々な変更の必要のため，数多くのそのようなパッチを持つことになるかもしれない．この状況で，アップストリームのメンテナにバグ修正パッチを送り，彼らがあなたの修正を後のリリースに取り込めば，手元で新しいリリースに切替えた時には単純にそのパッチを破棄すれば良い．

単一のパッチをアップストリームのツリーに対してメンテナンスすることはやや面倒で，間違いの元になりがちであるが，難しくはない．しかし，メンテナンスするパッチの数が増えるに従って問題の複雑さが急速に増していく．パッチの数がある程度以上多くなると，どのパッチを適用したか，どのパッチを管理しているのかの理解が，厄介という状態から圧倒されるている状態になる．

幸いにも，Mercurial は Mercurial Queues（あるいは単に MQ）という強力なエクステンションをもっており，パッチ管理の問題を大幅に単純化する．

12.2 Mercurial Queues 前史

1990 年代の終わり頃，Linux カーネルの開発者達は，Linux カーネルを改善する一連のパッチの管理を始めた．それらのうち，あるものは安定性に，別のものは特定の機能に，また別のものは野心的な内容に特化していた．

これらのパッチシリーズのサイズははすぐに膨れ上がった．2002 年に Andrew Morton は彼のパッチキューの管理を自動化するいくつかのシェルスクリプトを公表した．Andrew は数百から数千の Linux カーネルパッチをこれらのスクリプトで管理することができていた．

12.2.1 patchwork quilt

2003 年の始め，Andreas Gruenbacher と Martin Quinson は Andrew のスクリプトのやり方を真似て，“patchwork quilt” [**?**] あるいは単に “quilt” と呼ばれるツールをリリースした．（詳しくは論文 [**?**] を参照のこと．）quilt は十分に自動化されたパッチ管理を提供していたので，ソフトウェア開発者の大きな支持を急速に獲得していった．

quilt はディレクトリツリー上で**パッチのスタック**を管理する．パッチの管理を始めるには，quilt に管理すべきディレクトリツリーと，管理対象のファイルを指定する．quilt はこれらのファイルの名前と中身を保存する．バグを修正する場合は，まず新しいパッチを（コマンド 1 つで）作成し，必要なファイルに修正を加えた後，パッチを “リフレッシュ” すればよい．

リフレッシュでは，quilt はディレクトリツリーをスキャンし，変更全てを反映させてパッチを更新する．“あるパッチが適用された状態のツリー” から “2 つのパッチが適用されたツリー” へ更新するのに必要な変更を追跡し，別のパッチを作ることもできる．

ユーザはどのパッチがツリーに適用されるかを**変更**できる．あるパッチをポップすると，このパッチによる変更はツリーから消滅する．quilt はどのパッチをポップしたか記憶しており，一度ポップしたファイルを再びプッシュし，ディレクトリツリーをパッチによる変更がなされた状態に戻すことができる．最も重要なことは，いつでも “refresh” コマンドを実行して一番新しく適用されたパッチを更新できることである．これはすなわち，パッチの適用された状態と，パッチそれ自体をいつでも更新できるということである．

quilt はリビジョン管理ツールについては全く関知しないため，展開された tar アーカイブ上でも，Subversion のワーキングコピー上でも同様に動作する．

12.2.2 patchwork quilt から Mercurial Queues へ

2005 年の中頃 Chris Mason は，quilt の機能を取り入れて，Mercurial に quilt のような動作を追加する Mercurial Queues というエクステンションを書いた．

quilt と MQ の違いは，quilt はリビジョン管理システムについて何も関知しないのに対して，MQ は Mercurial に統合されていることである．プッシュした個々のパッチは Mercurial のチェンジセットとして表現される．パッチをポップすると，チェンジセットは消えてなくなる．

quilt はリビジョン管理ツールと無関係に利用可能なため，Mercurial と MQ が使えない状況では依然として非常に有益なツールであることは記憶に留めておくべきである．

12.3 MQ の大きな利点

MQ がパッチとリビジョンコントロールの統合によってもたらす価値を誇張するわけにはいかない．

時を追う毎にリビジョンコントロールツールの利用が広がっているにもかかわらず，フリーソフトとオープンソースの世界にパッチが存在する大きな理由はその**機敏さ**にある．

伝統的なリビジョンコントロールツールは，行った操作の永久的で不可逆的な記録を残す．これには大きな価値がある一方で，窮屈に感じることもある．もし過激な実験をするのであれば，どのように進めるか慎重にする必要がある．さもなければ，不要な，あるいは誤解を招いたり，安定性を損なうトレースとエラーを永久的なリビジョン履歴に残すことになる．

対称的に，MQ による分散リビジョンコントロールとパッチの結合は，作業を隔離することを遥かに容易にする．パッチは通常のリビジョン履歴の上に乗っており，望むように消滅させたり再現させることができる．パッチを気に入らなければ，これを棄却することもできる．パッチがあなたの望むようなものでなければ，希望通りになるまで何度でも簡単に修正することができる．

例えば，リビジョンコントロールへのパッチの統合は，パッチを理解し，その影響，依って立つベースコードとの相互作用をデバッグすることを可能にする．適用されたパッチは，関連づけられたチェンジセットを持つため，“`hg log filename`” によってどのチェンジセットとパッチがファイルに影響を与えているか調べることができる．`bisect` コマンドで全てのチェンジセットと適用されたパッチに対してバイナリサーチを行い，どこでバグが混入したか，あるいは修正されたかを調べることができる．“`hg annotate`” コマンドでどのチェンジセットかパッチがソースファイルの特定の行を変更したか見ることができる．

12.4 パッチとは何か

MQ はパッチ指向である性質を隠蔽していないので，パッチが何であるか，ツールがパッチをどう扱うかを理解するのはたやすい．

Unix の伝統的な `diff` コマンドは 2 つのファイルを比較し，差分のリストを出力する．`patch` コマンドはこの差分を変更と解釈し，ファイルに及ぼす．図 12.1にこれらのコマンドの動作を示す．

```
$ echo 'this is my original thought' > oldfile
$ echo 'i have changed my mind' > newfile
$ diff -u oldfile newfile > tiny.patch
$ cat tiny.patch
--- oldfile        Tue Jun  9 06:07:04 2009
+++ newfile        Tue Jun  9 06:07:04 2009
@@ -1 +1 @@
-this is my original thought
+i have changed my mind
$ patch < tiny.patch
patching file oldfile
$ cat oldfile
i have changed my mind
```

Figure 12.1: `diff` コマンドと `patch` コマンドの単純な使用例

`diff` が生成する（また `patch` が入力に取る）ファイルの種類は “patch” または “diff” と呼ばれ，これらの間には違いはない．(以下では，より多く使われている “patch” という言葉を使うことにする．)

パッチファイルは任意のテキストで始まる．`patch` コマンドはこのテキストを無視するが，MQ はチェンジセットを作る際のコミットメッセージとして利用する．パッチ内容の先頭を見つけるために `patch` は “`diff -`” で始まる最初の行をサーチする．

MQ*unified* diffs 形式を使う．(`patch` コマンドはこれ以外に何種類かの diff フォーマットを受け付けるが，MQ では利用できない．) unified diff は 2 種類のヘッダを持つ．*file* **ヘッダ**は変更されているファイルを記述する．`patch` コマンドは，新しいファイルヘッダを見つけると，そこに記述されているファイル名を持つファイルを探す．

ファイルヘッダの後には一連の *hunks* が続く．各々の hunk はヘッダで始まる．このヘッダはファイル内で hunk が改変すべき行番号の範囲を識別するのに使われる．ヘッダに続き，hunk は数行（通常 3）の改変されていないファイルからの行がある．これらは hunk の**コンテキスト**と呼ばれる．もし連続する hunk の間にコンテキストが少ししかなければ，`diff` は新たな hunk ヘッダを出力せず，各々の hunk と間のコンテキストを合わせて出力する．

コンテキストの各行は，空白文字で始まる．hunk 内では行は “-” で始まる．これは “この行を削除せよ” を意味する．一方，“+” で始まる行は，“この行を挿入せよ” を意味する．例を挙げると，変更された行は 1 つの削除と 1 つの挿入で表される．

パッチの些細な点については後で（ 12.6）また触れるが，これまでで，MQ を使うのに十分な情報は述べた．

12.5 Mercurial Queues を使ってみる

MQ はエクステンションとして実装されているので，使う前に明示的に有効にする必要がある．(何もダウンロードする必要はない．MQ は通常の Mercurial に同梱されている）MQ を有効にするには `~/.hgrc` ファイルを編集し，図 12.5のように設定を追加する．

エクステンションが有効化されると一連のコマンドが利用可能になる．エクステンションが有効であることを確認するためには，“`hg help`” で “`hg qinit`” があることを確認する．図 12.3の例を参照されたい．

```
[extensions]
hgext.mq =
```

Figure 12.2: MQ エクステンションを有効にするために~/.hgrc に追加する内容

```
$ hg help qinit
hg qinit [-c]

init a new queue repository

    The queue repository is unversioned by default. If -c 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
```

Figure 12.3: MQ が有効であることの確認法

MQ は**あらゆる**Mercurial リポジトリで使うことができ，そのコマンドは各々のリポジトリの中にだけ作用する．始めるために“hg qinit”コマンドでリポジトリを用意する．(図 12.4を参照) このコマンドは MQ がメタデータを保存するための.hg/patches という名前の空のディレクトリを作成する．他の Mercurial コマンドと同様に，“hg qinit”コマンドも成功すると何も表示しない．

```
$ 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
```

Figure 12.4: MQ を使うためにリポジトリを準備する

12.5.1 新しいパッチの作成

新しいパッチを扱う時は“hg qnew”コマンドを使う．このコマンドは作成されるパッチの名前として引数を 1 つ取る．図 12.5のように，MQ はこれを.hg/patches ディレクトリ内の実際のファイル名として利用する．

.hg/patches ディレクトリの中には，series と status という 2 つのファイルも新しく作られる．series ファイルは MQ が関知する，このリポジトリ内の全てのパッチのリストがあり，1 行に 1 つのパッチが記述されている．Mercurial は status ファイルを内部の管理に用いる．このファイルはリポジトリ内で MQ が適用し

```
$ hg tip
changeset:   0:18c40a119a78
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:08 2009 +0000
summary:     first change

$ hg qnew first.patch
$ hg tip
changeset:   1:941d7db9e8a7
tag:         qtip
tag:         first.patch
tag:         tip
tag:         qbase
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:09 2009 +0000
summary:     [mq]: first.patch

$ ls .hg/patches
first.patch  series  status
```

Figure 12.5: 新しいパッチの作成

た全てのパッチが記録されている．

> **Note:** 例えば，複数のパッチが適用される手順を変更するなどの目的で `series` ファイルを手で編集したくなることがあるかもしれない．しかし `status` を手で編集するのはほとんどの場合良くない考えで，MQ の状態追跡を簡単に狂わせてしまう．

新しいパッチを作った後，ワーキングディレクトリ内のファイルを通常通り編集することができる．“`hg diff`” や “`hg annotate`” のような全ての通常の Mercurial コマンドが全く同じように使える．

12.5.2 パッチのリフレッシュ

作業内容をセーブするポイントに差しかかったら，“`hg qrefresh`” コマンドを使って（図 12.5）作業中のパッチを更新する．このコマンドはワーキングディレクトリの差分をパッチに取り込み，チェンジセットが変更を含むように更新する．

“`hg qrefresh`” はいつでも好きな時に実行でき，作業の状態保存をするのにも使える．パッチのリフレッシュは都合のよい時に行えばよい．実験的なコードを動かす場合，事前にリフレッシュをしておけば，実験の結果コードが動かなくても，“`hg revert`” すれば，変更は元に戻る．

12.5.3 パッチのスタックと追跡

パッチへの作業が終了したり，他の作業ををする必要がある場合，“`hg qnew`” コマンドを実行し，新しいパッチを作ることができる．Mercurial はこのパッチを既存のパッチの上から適用する．図 12.8を参照のこと．パッチは先行するパッチの変更をコンテキストとして持つ．(“`hg annotate`” の出力でより明瞭に確認することができる．)

これまでは “`hg qnew`” と “`hg qrefresh`” を除いて，通常の Mercurial コマンドのみを使うように注意してきた．しかし MQ はパッチを扱う場合，図 12.9に示すようなより簡単なコマンドを用意している．

```
$ echo 'line 2' >> file1
$ hg diff
diff -r 941d7db9e8a7 file1
--- a/file1        Tue Jun 09 06:07:09 2009 +0000
+++ b/file1        Tue Jun 09 06:07:09 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]   0a131332f94e   2009-06-09 06:07 +0000   bos
  [mq]: first.patch

diff -r 18c40a119a78 -r 0a131332f94e file1
--- a/file1        Tue Jun 09 06:07:08 2009 +0000
+++ b/file1        Tue Jun 09 06:07:09 2009 +0000
@@ -1,1 +1,2 @@
 line 1
+line 2

```

Figure 12.6: パッチのリフレッシュ

- “hg qseries” コマンドは MQ が関知するリポジトリ内のパッチ全てを古いものから新しく**作成された**ものの順にリスト表示する．
- “hg qapplied” コマンドは MQ が**適用した**リポジトリ内のパッチ全てをやはり古いものから新しく**作成された**ものの順にリスト表示する．

12.5.4 パッチスタックの操作

これまでの議論は “known” と “applied” パッチに違いがあることを暗黙に仮定していた．MQ は差異のないパッチをリポジトリに対して適用することもできる．

適用されたパッチはリポジトリ内に対応するチェンジセットを持ち，パッチの効果とチェンジセットはワーキングディレクトリ内で見ることができる．パッチの適用は “hg qpop” コマンドで取り消すことができる．ポップされたパッチは対応するチェンジセットを持たず，変更もワーキングディレクトリに残っていないが，MQ はポップされたパッチを依然として記憶し，管理している．図 12.10は適用されたパッチと管理されているパッチの違いを示す．

適用を外したパッチやポップしたパッチを “hg qpush” コマンドで再適用することができる．このコマンドは，パッチに対応した新しいチェンジセットを作り，パッチによる変更はもう一度ワーキングディレクトリに現れる．“hg qpop” と “hg qpush” の使用例を図 12.11に示す．1 つまたは複数のパッチをポップすると “hg qapplied” の出力は変化するが，“hg qseries” の出力は同じまま残るのに注意されたい．

12.5.5 パッチのプッシュとポップ

“hg qpush” と “hg qpop” は一度に各々のパッチ一つずつを操作する（これはデフォルト動作である）一方，複数のパッチを一度にポップ・プッシュする操作もある．“hg qpush” コマンドの-a オプションは，未適用のパッチ全体をプッシュし，“hg qpop” コマンドの-a オプションは適用済みのパッチ全体をポップする．(他の複数のパッチを同時にプッシュ・ポップする方法については下の 12.7を参照のこと)

```
$ echo 'line 3' >> file1
$ hg status
M file1
$ hg qrefresh
$ hg tip --style=compact --patch
1[qtip,first.patch,tip,qbase]   d81af87521ed   2009-06-09 06:07 +0000   bos
  [mq]: first.patch

diff -r 18c40a119a78 -r d81af87521ed file1
--- a/file1        Tue Jun 09 06:07:08 2009 +0000
+++ b/file1        Tue Jun 09 06:07:09 2009 +0000
@@ -1,1 +1,3 @@
 line 1
+line 2
+line 3

```

Figure 12.7: パッチのリフレッシュで変更を蓄積する

12.5.6 安全性チェックとオーバライド

いくつかの MQ コマンドは，実行時にまずワーキングディレクトリをチェックし，変更がなされていたら終了するようになっている．これは，ユーザのまだパッチに取り込まれていない変更が失われないようにするためである．図 12.13はこれを説明したもので，“hg qnew” コマンドは孤立した変更がある時に新たなパッチを生成しない．この例では，変更は “hg add” による file3 の追加によって生まれている．

ワーキングディレクトリのチェックを行うコマンドは全て，上級者向けの強制オプション-f を持っている．-f の厳密な意味は，コマンドによって異なる．例えば “hg qnew -f” は孤立した変更を新しいパッチに取り込むという意味だが，“hg qpop -f” は，ポップするパッチによるあらゆるファイルへの変更を取り消すという意味になる．-f オプションを使う時は必ず事前にコマンドのドキュメントを読んで確認するようにして欲しい．

12.5.7 複数のパッチを一度に扱う

“hg qrefresh” コマンドは**最も上**に適用されたパッチに対してリフレッシュを行う．これはリフレッシュによって 1 つのパッチに現在の作業をサスペンドしたり，別のトップパッチや一時的に作業するためパッチを作るためにポップやプッシュができるということを意味する．

この機能の使い方を示す例を示す．2 つのパッチとして新しい機能を開発しているとする．最初のものはあなたのソフトウェアのコアを変更し，2 番目のものは 1 番目のものの上に展開する，今追加したコアのコードを使ったユーザインタフェースだとする．UI パッチで作業中にコアのバグに気付いたとすると，簡単にコアの修正は行える．単純に “hg qrefresh” によって進行中の UI パッチをセーブし，“hg qpop” によってコアパッチへ遡る．コアのバグを修正した後に “hg qrefresh” をコアパッチについて行い，“hg qpush” で UI パッチに戻って作業を続ける．

12.6 さらにパッチについて

MQ はパッチの適用に GNU の patch コマンドを用いる．そのため，patch の動作とパッチ自体についてより詳しく知っておくことは有用である．

```
$ hg qnew second.patch
$ hg log --style=compact --limit=2
2[qtip,second.patch,tip]   7ec2570ed8d3   2009-06-09 06:07 +0000   bos
  [mq]: second.patch

1[first.patch,qbase]   d81af87521ed   2009-06-09 06:07 +0000   bos
  [mq]: first.patch

$ echo 'line 4' >> file1
$ hg qrefresh
$ hg tip --style=compact --patch
2[qtip,second.patch,tip]   5224ca7cbc2d   2009-06-09 06:07 +0000   bos
  [mq]: second.patch

diff -r d81af87521ed -r 5224ca7cbc2d file1
--- a/file1        Tue Jun 09 06:07:09 2009 +0000
+++ b/file1        Tue Jun 09 06:07:10 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
```

Figure 12.8: 最初のパッチの上に 2 番目のパッチをスタックする

12.6.1 ストリップカウント

パッチのファイルヘッダを見ると，パスネームの最初に追加の部分がついているのに気づくはずだ．これは実際のパスネームには存在しないもので，パッチを生成する時の習慣からの遺物である．(多くの人がいまだこの方法を使っているが，近代的なリビジョンコントロールツールではほとんど見られない．)

アリスが tar アーカイブを展開し，ファイルを編集し，パッチを作りたいと思っているとする．彼女は差分を取るためにまず作業しているディレクトリをリネームし，tar アーカイブをもう一度展開する．そしてオリジナルのままのディレクトリと，変更を行なったディレクトリの間で，そして `diff` に`-r` と`-N` オプションを付けて，再帰的な差分を取り，パッチを生成するだろう．生成されたパッチファイルを見ると，各ファイルのヘッダ部分で左辺値の先頭にオリジナルの方のディレクトリ名が現れ，右辺値の先頭に変更した方のディレクトリ名が現れているはずである．

ネット上の多数のアリスたちからパッチを受け取った人は，変更なしのディレクトリと変更されたディレクトリの両方をアリスと同じ名前で持っている可能性はまずない．`patch` コマンドには`-p` オプションがあり，パッチを適用する時にパスの要素を先頭からいくつ削るかを指定できる．この数は**ストリップカウント**と呼ばれる．

オプション "`-p1`" は "ストリップカウント 1 を使う" という意味である．もし `patch` がファイルヘッダに `foo/bar/baz` を見つけた時は，`foo` を落して `bar/baz` という名前のファイルにパッチを試みる．(厳密に言うとストリップカウントは**パスセパレータ**の数を参照している（結果として要素も参照される）ストリップカウント 1 は `foo/bar` を `bar` に変える．また`/foo/bar`（前に置かれたスラッシュに注意）は `foo/bar` となる．)

```
$ hg qseries
first.patch
second.patch
$ hg qapplied
first.patch
second.patch
```

Figure 12.9: “hg qseries” と “hg qapplied” でパッチスタックを調べる

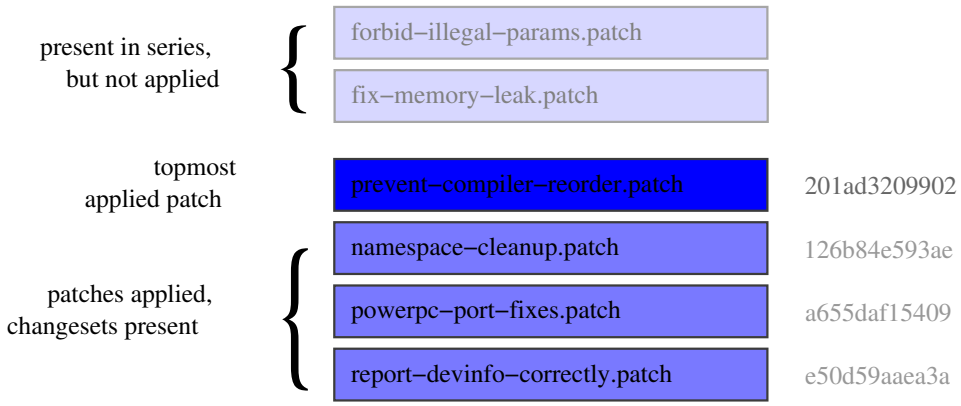

Figure 12.10: MQ パッチスタックの中の適用されたパッチと適用されないパッチ

パッチへの “標準的な” ストリップカウントは 1 である．ほとんど全てのパッチは取り除くべきパス要素を一つ持っている．Mercurial の “hg diff” コマンドはパス名をこの形式で生成し，“hg import” コマンドと MQ はストリップカウント 1 を前提としている．

誰かからパッチを受け取り，パッチキューに追加するとき，ストリップカウントが 1 以外であれば単に “hg qimport” することはできない．これは “hg qimport” コマンドがまだ-p オプションを持っていないためだ（Mercurial bug no. 311 を参照）．最も上手くいきそうな方法は，“hg qnew” で新たにパッチを作り，“patch -p*N*” を実行して受け取ったパッチを適用した後，パッチによって追加されたり削除されたファイルを “hg addremove” によってピックアップし，“hg qrefresh” を実行することである．この複雑な手順は将来的には必要なくなるはずだ（Mercurial bug no. 311 を参照）．

12.6.2 パッチ適用のための戦略

patch が hunk を適用するとき，いくつかのあまり正確でない戦略を続けて用いる．このフォールバック手法によって，古いバージョンのファイルに対して作成したパッチを新しいファイルに適用することさえしばしば可能となる．

最初に，patch は行番号，コンテキスト，変更されるテキストの正確なマッチを試みる．正確なマッチができない場合，行番号を無視してコンテキストの正確なマッチを試みる．これが成功すれば，hunk が適用されたが，**オフセット**が生じたというメッセージを表示する．

コンテキストのみのマッチが失敗すると，patch はコンテキストの最初と最後の行を削り，コンテキストの**縮小**マッチを試みる．縮小コンテキストによる hunk が成功すると，hunk が適用されたことと *fuzz* **ファクター**を含むメッセージを表示する．(fuzz ファクターの後ろの数字は，patch コマンドが，パッチを適用する前にコンテキストの何行を削除する必要があったのかを示す.)

どちらも上手くいかなかった場合，patch は hunk がリジェクトされたというメッセージを表示する．コマンドはリジェクトされた hunk（あるいは単に “rejects”）を同名で拡張子.rej のファイルにセーブする．コマンドは同時に無変更のファイルを.orig という拡張子でセーブする。拡張子のないファイルのコピーは，hunk

```
$ 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
```

Figure 12.11: 適用されたパッチのスタックを変更する

```
$ hg qpush -a
applying second.patch
now at: second.patch
$ cat file1
line 1
line 2
line 3
line 4
```

Figure 12.12: 適用されていないパッチを全てプッシュする

からクリーンに適用された変更を全て含む．foo というファイルを変更する 6 つの hunk を含むパッチファイルがあり，その内の 1 つが適用できなかったとすると，無変更のファイル foo.orig，hunk1 つを含むファイル foo.rej，正しく適用された 5 つの hunk を含むファイル foo が生成される．

12.6.3 パッチ表現の奇妙な点

patch がファイルにどのように作用するか，有用な点をいくつか述べる．

- すでに明白だが，patch はバイナリファイルを扱うことはできない．
- 実行ビットを考慮しないだけでなく，新しいファイルを読み取り可能属性で作成し，実行可能属性は付けない．
- patch コマンドはファイルの削除を削除されるファイルと中身が空のファイルとの差分として扱う．従って，ファイルを削除するということは，パッチファイルの中ではファイルの中の全ての行を削除することと等価である．
- patch コマンドはファイルの追加を空のファイルと追加されるファイルの差分として扱う．従ってファイルの追加はパッチファイルの中ではファイルに全ての行を追加することと等価である．
- patch コマンドは名前の変更されたファイルを旧名のファイルの削除と新しいファイルの追加として扱う．従って名前の変更されたファイルはパッチ内で大きなフットプリントを占める．(Mercurial は現在のところ，パッチ内でファイルがリネームされたのか，コピーされたのかを関知しない)

```
$ echo 'file 3, line 1' >> file3
$ hg qnew add-file3.patch
$ hg qnew -f add-file3.patch
abort: patch "add-file3.patch" already exists
```

Figure 12.13: パッチの強制的な生成

- patch コマンドは中身の空のファイルを表現できない．従ってパッチファイルを「空のファイルをツリーに追加する」という用途に使うことはできない．

12.6.4 曖昧な点について

あるオフセットや fuzz factor の位置への hunk の適用は，多くの場合問題なく成功するが，不正確なマッチを用いた場合，パッチを適用したファイルを壊している可能性が残る．最もよくあるのは，パッチを二度適用したり，ファイルの間違った場所に適用してしまうことだ．patch または "hg qpush" がオフセットか fuzz ファクターを表示している合，変更されたファイルが正常であるか確認する必要がある．

オフセットや fuzz factor の出たパッチをリフレッシュするのは多くの場合良い考えである．パッチをリフレッシュすると，新しいコンテキスト情報が生成され，パッチがクリーンに適用できるようになる．「常に」ではなく「多くの場合」と断ったのは，パッチをリフレッシュすることで，元のファイルの異なったリビジョンでパッチが適用できなくなることがあるからだ．複数のバージョンのソースツリーの上で適用可能なパッチを管理しなければならないような場合などは，パッチの結果を検証してあるのであれば fuzz の出るパッチも許容され得る．

12.6.5 リジェクトの取り扱い

"hg qpush" はパッチの適用に失敗した場合，エラーメッセージを表示して終了する．.rej ファイルが作られている場合は，通常，さらなるパッチの追加や新しい作業の前にリジェクトされた hunk について修正を行うべきである．

それまでパッチがクリーンに適用されており，パッチの対象になっているコードに変更を加えたために適用できなくなったのなら，Mercurial Queues の支援を受けることができる．詳しい情報はセクション 12.8を見て欲しい．

残念なことに，リジェクトされた hunk を取り扱う決まった良い方法は存在しない．大抵の場合，.rej ファイルを見てターゲットファイルを編集し，リジェクトされた hunk を手で適用することになる．

冒険が好きなら，Linux kerner ハッカーの Neil Brown が書いた wiggle [**?**] を試してみると良い．このコマンドは patch よりも精力的にパッチの適用を試みる．

別の Linux kerner ハッカー Chris Mason（Mercurial Queues の作者でもある．）は mpatch [**?**] というツールを書いた．このコマンドは patch コマンドでリジェクトされた hunk の適用を自動化する．mpatch コマンドは，hunk がリジェクトされる主な原因 4 つに対応する:

- hunk の中のコンテキストが変更された
- hunk の開始部・終端部でコンテキストが見つけられない
- 大きな hunk はもっと上手に適用できる筈だ—hunk を小さな hunk に分割して適用する
- hunk 内の行が現在ファイルにある行と若干違っている場合，その行を削除する

wiggle または mpatch を使用した場合は，結果に細心の注意が必要である．実際には mpatch はツールの出力で二重チェックを強制し，動作が終ると自動的にマージプログラムを起動する．これによって作業を確認し，マージを完了することができる．

12.7 MQ を最大限に活用する

MQ はとても効率的に大量のパッチを処理することができる．筆者は 2006 年の中頃，2006 EuroPython conference [**?**] での講演のために性能測定を行った．測定に用いたデータは Linux 2.6.17-mm1 のパッチシリーズで，1,738 のパッチを含む．このパッチを Linux kernel リポジトリに適用した．リポジトリは Linux 2.6.12-rc2 から Linux 2.6.17 までの 27,472 のリビジョンを含んでいる．

筆者の古く遅いラップトップで，“`hg qpush -a`” で 1738 のパッチすべてを処理するのに 3.5 分，“`hg qpop -a`” を行うのに 30 秒を要した．(新しいラップトップではプッシュの所要時間は 2 分であった．) 最大のパッチ (22.778 行で，287 のファイルを変更する) に “`hg qrefresh`” を行ったところ，6.6 秒を要した．

MQ が巨大なツリーでの作業に適しているのは明らかだが，さらに最高の性能を引き出すためにいくつかのトリックが使える．

まず第一に，“batch” オペレーションを併用することができる．“`hg qpush`” または “`hg qpop`” は，ワーキングディレクトリに変更を加えたのに “`hg qrefresh`” の実行を忘れていないかどうか調べるため，実行時に常にワーキングディレクトリをスキャンする．小規模なツリーでは気づかない程度の時間しかかからないが，(数万のファイルを持つような) 中規模のでは数秒の時間を要する．

“`hg qpush`” コマンドと “`hg qpop`” コマンドは複数のファイルを同時に push または pop することができる．最終目的のパッチがあるならば，“`hg qpush`” に目的のパッチを指定して実行することで，指定したパッチが再上位になるまでパッチを push する．同様に “`hg qpop`” に目的のパッチを指定すれば，指定したパッチが再上位になるまでパッチを pop する．

目的のパッチは名前でも番号でも指定可能である．番号が用いられる場合は，パッチは 0 からカウントされる．つまり最初のパッチは 0 となり，2 番目は 1，という風になる．

12.8 対象コードの変化に合わせてパッチを更新する

直接変更しないリポジトリの上にパッチのスタックを管理することはよく行われる．サードパーティのコードを変更する作業をしている場合や，元になるコードの変更頻度に比べて長い時間がかかる機能を開発している場合，しばしば元のコードの同期を行い，パッチの中のもはや適用できなくなった hunk を修正することになるだろう．これはパッチの *rebasing* と呼ばれる．

これを行う最も簡単な方法は，パッチに対して “`hg qpop -a`” を行い，下位のリポジトリに “`hg pull`” を行う．そして最後にパッチに再び “`hg qpush -a`” を行う． MQ はパッチ適用中にコンフリクトがあると，いつでもプッシュを停止し，“`hg qrefresh`” によってコンフリクトを修正する機会を与える．その後にすべてのパッチスタックを適用するまでプッシュを続ける．

このアプローチは簡単で，パッチの適用される下位のコードへの変更がなければうまく働く．しかし，パッチスタックが，頻繁に更新されたり，下位リポジトリへ侵入的に更新されたりするコードに触れている場合はリジェクトされたコードの修正は面倒なものになる．

rebase プロセスを部分的に自動化することは可能である．パッチが下位のリポジトリのいずれかのバージョンにクリーンに適用できるのであれば，MQ はこの情報を使ってパッチとその他のリビジョンとの間のコンフリクトを解消するのを援助する．

このプロセスはやや込み入っている．

1. 最初に，パッチがクリーンに適用できるリビジョンの上で，すべてのパッチに対して “`hg qpush -a`” を行う．

2. パッチディレクトリをセーブを “`hg qsave -e -c`” を用いて行う．パッチをセーブしたディレクトリ名が表示される．このコマンドは，`.hg/patches.`*N* というディレクトリにセーブを行う．ここで *N* は小さい整数である．このコマンドは “セーブチェンジセット” を適用されたパッチの上にコミットする．これは内部的な管理と，`series` ファイル及び `status` ファイルの状態を記録するためである．

3. “`hg pull`” コマンドを使って新たな変更を下位のリポジトリに取り込む．(“`hg pull -u`” を実行してはいけない．理由については下記を参照．)

4. “`hg update -C`” を用いて，push したパッチをオーバライドして新たな tip リビジョンへのアップデートを行う．

5. “hg qpush -m -a” を用いてすべてのパッチのマージを行う．-m オプションを “hg qpush” に付けると，MQ はパッチの適用に失敗した場合，3 ウェイマージを行う．

“hg qpush -m” コマンドの実行中，series 内のパッチは通常通り適用される．もしパッチが fuzz や reject を出した場合，MQ は “hg qsave” したキューを参照し，対応するチェンジセットとの間で 3 ウェイマージを行う．マージは Mercurial の通常の機構を使って行われるため，設定によって問題を解決するための GUI マージツールなどが起動する．

パッチの影響の解決を終えた時，MQ はマージの結果を踏まえてパッチのリフレッシュを行う．

このプロセスの最後にリポジトリには古いパッチキューに由来する一つ余分な head ができ，古いパッチキューが.hg/patches.*N* にコピーされる．この head は “hg qpop -a -n patches.*N*” または “hg strip” によって消去できる．バックアップが不要なことが分かれば，.hg/patches.*N* を消去しても構わない．

12.9 パッチの識別

パッチを扱う MQ コマンドは，パッチをパッチ名または番号で参照する．名前の場合は例えばファイル名 foo.patch を “hg qpush” に渡す．この場合，コマンドは foo.patch までのファイルをプッシュする．

ショートカットとして，パッチに名前と数字によるオフセット両方を用いて参照することができる．foo.patch-2 は “foo.patch の 2 つ前のパッチ” と解釈される．また，bar.patch+4 は “bar.patch の 4 つ後のパッチ” と解釈される．

インデックスがあまり違わないパッチの参照．“hg qseries” で最初に表示されるパッチはパッチ 0(これも 0 から数え始める) で，2 番目はパッチ 1 のように続く．

MQ は，通常の Mercurial コマンドを使う時のも簡単にしている．全てのコマンドはチェンジセット ID と適用されたパッチの番号を受け付ける．MQ は，リポジトリに適用されたパッチの名前のついたタグを受け付ける．さらに，最下位の適用されたパッチを表す特別なタグ qbase と，最上位を表す特別なタグ qtip も受け付ける．

Mercurial の通常のタグ機能へのこれらの追加は，パッチの取り扱いにおいて大きな意味を持つ．

- 一連の最新の変更をパッチ爆弾としてメーリングリストへに投げ込みたいだろうか？

```
hg email qbase:qtip
```

(“patchbombing” が何か分からなければ section 14.4を参照のこと．)

- ツリー中のサブディレクトリに含まれるファイルに触れた foo.patch 以降のパッチを全て見たいか？

```
hg log -r foo.patch:qtip subdir
```

MQ でのパッチの名称は，内部のタグ機構を用いているため，Mercurial の他の部分でも利用可能である．名称を指定する時，名称全てを入力する必要はない．

パッチ名をタグとして取り扱うことのもう一つの利点は，“hg log” コマンドを実行した時，通常出力の一部にパッチ名をタグとして表示する点である．このため，適用したパッチは下位の “通常の” リビジョンと視覚的に区別し易くなる．図 12.14にいくつかの通常の Mercurial コマンドを適用済みパッチと共に使用した場合を示す．

12.10 知っておくべきいくつかの点

MQ の使用法には個別に取り上げるほどではないが，知っておくと良いいくつかの点がある．ここではそれらをまとめて取り上げる．

- パッチを “hg qpop” した後で再び “hg qpush” すると，pop/push した後のチェンジセットは，以前のチェンジセットと異なるアイデンティティを持ち，ハッシュ値が異なる．この理由についてはセクション B.1.13を参照されたい．

```
$ hg qapplied
first.patch
second.patch
$ hg log -r qbase:qtip
changeset:   1:2e300002d528
tag:         first.patch
tag:         qbase
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:05 2009 +0000
summary:     [mq]: first.patch

changeset:   2:804dc37536fb
tag:         qtip
tag:         second.patch
tag:         tip
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:05 2009 +0000
summary:     [mq]: second.patch

$ hg export second.patch
# HG changeset patch
# User Bryan O'Sullivan <bos@serpentine.com>
# Date 1244527625 0
# Node ID 804dc37536fb16ee409c274d6ae05072b127528f
# Parent  2e300002d5289ee7820d569af559f4611db81fb5
[mq]: second.patch

diff -r 2e300002d528 -r 804dc37536fb other.c
--- /dev/null        Thu Jan 01 00:00:00 1970 +0000
+++ b/other.c        Tue Jun 09 06:07:05 2009 +0000
@@ -0,0 +1,1 @@
+double u;
```

Figure 12.14: パッチを扱うため MQ のタグ機能を利用する

- "`hg merge`" が他のブランチのパッチチェンジセットをマージすることは，そのチェンジセットとパッチスタックに積み込まれた他のチェンジセットとの間で一貫性を維持しようとするのであれば避けるべきである．これを試みた場合，一見成功したように見えても MQ は混乱状態に陥ってしまう．

12.11 リポジトリ内でのパッチの管理

MQ の`.hg/patches` ディレクトリは Mercurial リポジトリのワーキングディレクトリの外にあるので，"下位の"Mercurial リポジトリはパッチの管理や，その存在自体について何も知らない．

このことから，パッチディレクトリの管理を独立した Mercurial リポジトリとして行うことができるのではないかという興味が生まれる．実際この方法は有効な方法となり得る．例えば，しばらくの間 1 つのパッチについて作業を行った後で，"`hg qrefresh`" を行い，パッチの現在の状態を "`hg commit`" することができる．これによって，後でこのパッチを "ロールバック" することが可能となる．

複数の下位リポジトリの間で，同じパッチスタックの異なるバージョンを共有することができる．筆者はこれを Linux カーネルの機能を開発する時に利用している．いくつかの CPU アーキテクチャの各々に対して

まっさらなカーネルソースのリポジトリを用意し，作業中のパッチを含むリポジトリをそれらの間でクローンしている．変更を異なるアーキテクチャでテストしたい時は，現在のパッチをそのアーキテクチャ用のカーネルツリーと結び付いたパッチのリポジトリへプッシュし，すべてのパッチを pop，push してカーネルのビルドとテストを行っている．

リポジトリ内でパッチを管理すると，複数の開発者が同じパッチシリーズの上でコンフリクトすることなく作業できるようになる．これは彼らが下位のソースベースをコントロールできるか否かに関わりがない．

12.11.1 MQ によるパッチリポジトリサポート

MQ は.hg/patches ディレクトリをリポジトリとして使いながら作業することをサポートしている．パッチを扱う作業用のリポジトリを “hg qinit” で作成する時，-c を渡して.hg/patches ディレクトリを Mercurial リポジトリにすることができる．

> **Note:**
> -c オプションを忘れた場合はいつでも.hg/patches に入って “hg init” を実行すればよい．status ファイルを.hgignore ファイルに追加するのを忘れないこと．(“hg qinit -c” はこれを自動的に行う.）status をバージョン管理する必要は**全くない**.

.hg/patches がリポジトリの時は，MQ は便利のためにこれまで作成し，インポートしたすべてのパッチを自動的に “hg add” する．

MQ はショートカットコマンド “hg qcommit” を持つ．このコマンドは “hg commit” コマンドを.hg/patches ディレクトリの中で実行する．これによって繁雑な入力を省くことができる．

パッチディレクトリの管理のために UNIX 環境なら mq というエイリアスを作っておくと便利だ．Linux では bash シェルの設定ファイル~/.bashrc にこれを定義しておく．

```
1 alias mq='hg -R $(hg root)/.hg/patches'
```

メインリポジトリから “mq pull” を実行することができる

12.11.2 注意しておくべきいくつかの点

MQ が大量のパッチのあるリポジトリを扱う際に，いくつかの小さな点から制限がある．

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ない．パッチファイルや series に対して “hg pull” を行ったり，手で編集したり，“hg update” を行った場合は，下位のリポジトリに対して “hg qpop -a” と “hg qpush -a” を行い，変更を知らせる必要がある．これを忘れると MQ はどのパッチが適用されたのか把握できなくなってしまう．

12.12 パッチ向けサードパーティツール

しばらくパッチを使った作業をしていると，パッチを取り扱う助けとなるようなツールが欲しくなるに違いない．

diffstat コマンド [**?**] はパッチ内のそれぞれのファイルへの変更のヒストグラムを生成する．これは，パッチが影響を及ぼすファイルを把握したり，全体でそれぞれのファイルへどの程度変更を加えるのか把握するのに役立つ．（diffstat コマンドの-p オプションを用いるのは勿論良い考えである．さもなければコマンドはファイル名の prefix に対して “賢い” ことをしでかすため，少なくとも筆者は混乱を避けられなかった.）

patchutils パッケージ [**?**] は非常に有用である．このパッケージは “UNIX 哲学” に従った小さなユーティリティからなり，各々のユーティリティはパッチに対して有用な単機能を提供する．patchutils の中で筆者が最も良く使うコマンドは filterdiff で，これはパッチファイルからパッチの部分集合を取り出す．例えば，10 以上のディレクトリに跨って数百のファイルに変更を行うようなパッチがあるとして，ただ一回の filterdiff の実行で特定のパターンにマッチしたファイルのみを変更するより小さなパッチを生成することができる．そのほかの例は 13.9.2を見よ．

```
$ diffstat -p1 remove-redundant-null-checks.patch
bash: diffstat: command not found
$ filterdiff -i '*/video/*' remove-redundant-null-checks.patch
bash: filterdiff: command not found
```

Figure 12.15: diffstat, filterdiff, および lsdiff コマンド

12.13 好ましいパッチ取り扱い方法

フリーソフトやオープンソースプロジェクトに提出するためのパッチシリーズに対して作業しているときや，パッチとしての作業が終った際に通常のチェンジセットとして取り込む予定のパッチシリーズに対して作業しているときは，作業をうまく行なうためのシンプルなテクニックがある．

パッチに記述的な名前を与える．パッチに与えるべき良い名前は rework-device-alloc.patch のようなものである．理由は即座にパッチの目的のヒントを与えるからである．長い名前は問題にはならない．なぜなら名前をタイプすることは滅多になく，代わりに “hg qapplied” や “hg qtop” のようなコマンドを繰り返し実行することになるからである．多数のパッチに対して作業をしているときは，良い名前を付けることは特に重要となる．そうしないと，もしあなたがたくさんの別のタスクを抱えているならば，パッチの区別はつかなくなってしまう．

どのパッチについて作業しているのか把握しておくべきである．“hg qtop” コマンドを使い，パッチを頻繁にチェックすべきである．また，例えば “hg tip -p” を使って，いまどこにいるのかを調べることも必要である．意図しているパッチ以外のパッチについて “hg qrefresh” を行い，間違ったパッチに加えられた変更を正しいパッチに移動させた．

このため，セクション 12.12で述べたサードパーティ製のいくつかのツール，特に diffstat と filterdiff の使い方を学ぶのはとても意味がある．前者は，あなたのパッチがどのような変更を加えるのか即座に知ることができ，一方で後者はパッチの中の hunk を別のパッチへ挿入するのを容易にする．

12.14 MQ クックブック

12.14.1 “トリビアル” なパッチの管理

ファイルを Mercurial リポジトリに追加するオーバヘッドは小さい．このため，たとえソースの tar アーカイブへ少数の変更を加えるだけだとしても，パッチをリポジトリで管理するのは良い方法である．

ソース tarball をダウンロードして展開し，これを Mercurial リポジトリに変換する．

```
$ 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
```

新たなパッチスタックを作り，変更を行う．

```
$ 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:af97734f7610
tag:         qtip
tag:         build-fix.patch
tag:         tip
tag:         qbase
user:        Bryan O'Sullivan <bos@serpentine.com>
date:        Tue Jun 09 06:07:06 2009 +0000
summary:     fix build problem with gcc 4

diff -r bb7b49bf7b49 -r af97734f7610 netlink.c
--- a/netlink.c        Tue Jun 09 06:07:06 2009 +0000
+++ b/netlink.c        Tue Jun 09 06:07:06 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");

```

数週間から数カ月経った時，パッケージの作者が新しいバージョンをリリースしたとしよう．その場合，まず新しいバージョンとの差分をリポジトリに取り込む．

```
$ 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
```

“hg locate” の出力をパイプで繋いでワーキングディレクトリのすべてのファイルを消去し，“hg commit” コマンドの--addremove オプションを使って実際にどのファイルが新しいバージョンのソースで削除されたのか指定する．

最後にパッチを新しいツリーに適用する．

```
$ 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 tip)
applying build-fix.patch
now at: build-fix.patch
```

12.14.2 パッチ同士を結合する

MQ にはパッチ全体を結合する “hg qfold” というコマンドがある．このコマンドは名前をつけたパッチを名付けた順番で再上位の適用済みパッチに畳み込む．各々のパッチの説明は結合されてパッチの説明の末尾に追加される．畳み込むパッチはすでに適用されていてはならない．

パッチを畳み込む順番は重要である．今，再上位の適用済みパッチが foo だとして，“hg qfold” bar と quux を行うと，パッチは foo, then bar, quux の順番でパッチを適用したのと同じ効果を持つことになる．

12.14.3 パッチの一部分を別のパッチへマージする

パッチの**一部**を別のパッチへマージすることは，パッチを結合するよりは難しい．

変更をファイルの集合全体に移動する場合，filterdiff に-i と-x を付け，一つのパッチから取る去る変更，パッチの末尾に結合する変更を選ぶ．通常の場合，変更を取り出したパッチを変更する必要はない．その代わり，MQ は (他のパッチに移動した hunk から)“hg qpush” した時にリジェクトされた hunk を報告する．

パッチに一つのファイルを変更する複数の hunk があり，その中の一部を移動したい場合は，作業はもっとややこしいものになるが，依然として一部は自動化することができる．“lsdiff -nvv” コマンドでパッチのメタデータを表示する．

```
$ lsdiff -nvv remove-redundant-null-checks.patch
bash: lsdiff: command not found
```

このコマンドは 3 つの異なる番号を出力する:

- (最初の列) このパッチで変更されるファイルを識別するための**ファイル番号**
- (次の行でインデントされている) 変更されたファイルの中で hun が始まる行番号
- (それと同じ行) 当該 hunk を識別する *hunk* **番号**

必要なファイルと hunk 番号を見つけるために，パッチを読むなどして目視で確認する必要がある．その番号は，extract するファイルと hunk を選ぶために filterdiff コマンドの--files と--hunks オプションで使える．

この hunk ができたら，目的のパッチの末尾に結合し，セクションの残りの部分を続けることができる 12.14.2.

12.15 quilt と MQ の違い

すでに quilt に馴染んでいるユーザのために，MQ は同様のコマンドセットを用意している．コマンドの挙動には何点か違いがある．

すでにほとんどの quilt コマンドには，対応する “q” で始まる MQ コマンドがあることに気づいていることと思う．例外は，quilt の add コマンドと remove コマンドで，これらにはそれぞれ通常の Mercurial コマンドの “hg add” と “hg remove” が対応する．また，quilt の edit コマンドに対応する MQ コマンドはない．

Chapter 13

Mercurial Queuesの高度な使い方

Mercurial Queues の単純な使い方を取り上げた一方，少々の統制と MQ のあまり用いられない機能を用いることで複雑な開発環境での作業が可能となる．

この章では，Linux カーネル用 Infiniband デバイスドライバの開発を管理するのに用いたテクニックを例として用いる．このドライバは（ドライバとしては）大規模で，35 のソースファイルにわたる 25000 行からなる．このソースは小規模な開発者のチームによって維持管理されている．

この章での素材は Linux に特化しているが，同じ原則はいかなるコードベースに対しても適用可能で，自分自身で所有していないが，多くの開発を行う必要のあるコードなどにも適用できる．

13.1 ターゲットが複数あるという問題

Linux カーネルは急速に変化し，その内部は決して一定ではない．開発者たちは頻繁にリリース間で劇的な変更を行う．このため，特定のリリースバージョンで良く動いたドライバが，例えば別のバージョンでは正しくコンパイルすることすらできなくなるということを意味する．

ドライバを維持するために，開発者はいくつもの別のバージョンの Linux を想定しなければならない．

- 一つのターゲットは Linux カーネルの開発ツリーである．この場合，コードの維持管理はカーネルコミュニティの他の開発者（これはカーネルのサブシステムを改良するためにドライバを “追い立てる” 変更を行う開発者である）と部分的に共有されている．
- 開発者は我々のドライバが組み込まれていない古い Linux ディストリビューションを使っている顧客をサポートするために，古いバージョンの Linux カーネルへの “バックポート” を抱えている．コードの一部を**バックポート**することは，ドライバが開発されたバージョンよりも古いバージョンの環境で動くようにすることを意味する．
- 最終的に Linux ディストリビュータやカーネル開発者を待たせることなくスケジュール通りにソフトウェアをリリースすることが必要であり，これによって顧客にカーネルやディストリビューション全体をアップグレードさせることなく新機能を顧客に届けられる．

13.1.1 やってしまいがちな間違った方法

さまざまな環境をターゲットにしたソフトウェアを管理する 2 つの標準的な方法がある．

第 1 の方法は，単一のターゲット向けの複数のブランチを維持することである．この方法の問題点は，リポジトリ間での変更の流れについて厳格な規律を維持しなければならないことである．新機能やバグ修正は，きれいな状態のリポジトリで開始し，バックポートリポジトリに浸透する必要がある．バックポート変更は波及すべきブランチ内に限定されていなければならない．必要のないブランチへのバックポートの波及はおそらくドライバをコンパイル不能にしてしまうだろう．

第 2 の方法は，チャンクやコードを，目的とするターゲット毎に on/off する条件文を追加した単一のソースツリーを維持する方法である．これらの “ifdef” は linux カーネルツリーでは許されていないため，手動また

は自動でこれらのコードを除去し，クリーンなツリーを作るプロセスが必要になる．このようなやり方で管理されたコードベースはすぐに条件節の巣窟と化し、理解や管理の妨げとなる。

あなたが正式なソースツリーを所有しているのでなければ、これらのやり方のどちらもそぐわないだろう。標準の linux カーネルに同梱されているドライバの場合，Linus のツリーは世界中で正式なコードとして扱われるコピーを含んでいる．“私” のドライバの上流版は，知らない人によって変更され得るし，自分の変更が Linus のツリーに入ったあとでさえも変更され得る．

これらのアプローチは，うまく書かれたパッチを上流へ提出することを困難にしてしまう．

Mercurial Queues は上記のような開発シナリオを管理するよい候補の一つであるといえる．このような場合のために MQ には作業をより快適にするいくつかの機能がある．

13.2 ガードを使ったパッチの条件的な適用

多くのターゲットを対象としながら正常に保つ一番良い方法は，状況に応じて特定のパッチを選べるようにすることである．MQ はこの目的のために quilt の `guards` コマンドに類似した “guards” と呼ばれる機能を提供している．開始に当たって，実験のために単純なリポジトリを作成する．

```
$ 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
```

小さなリポジトリを作り，別々のファイルを操作する互いに依存性のない 2 つのパッチを置く．

条件付きでパッチの適用を制御する方法では，*guard* によってパッチに付けられた “タグ” を活用する．タグは自由に選んだ単純なテキストで，MQ がパッチを適用するときに用いる特定のガードを指定する．MQ はガードパッチの適用またはスキップを，指定されたガードによって決定する．

パッチは任意の数のガードを持つことができる．各々のガードは，*positive*（このガードが選択されているときにこのパッチを適用）または *negative*（このガードが選択されているときにこのパッチをスキップ）である．ガードを持たないパッチは常に適用される．

13.3 パッチ内のガードを操作する

“`hg qguard`” コマンドでどのガードがパッチに適用されるかを指定したり，すでに有効になっているガードを表示させることができる．引数なしでこのコマンドを使うと，現在の最上位のパッチのガードを表示する．

```
$ hg qguard
goodbye.patch: unguarded
```

パッチにポジティブガードを設定するには，ガード名の前に “+” をつける．

```
$ hg qguard +foo
$ hg qguard
goodbye.patch: +foo
```

パッチにネガティブガードを設定するには，ガード名の前に “-” をつける．

```
$ 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
```

Note: "hg qguard" コマンドはパッチにガードを**設定する**がパッチ自体を**変更しない**点に注意されたい．パッチに対して "hg qguard +a +b" を実行し，同じパッチに "hg qguard +c" を実行すると，このパッチへのガードは+c のみになる．

Mercurial はガードを series ファイルに保存する．書式は理解しやすく，手で変更するのも簡単である．(つまり，"hg qguard" コマンドを使いたくなければ，直接 series を編集して済ますこともできる．)

```
$ cat .hg/patches/series
hello.patch
goodbye.patch #+foo
```

13.4 使用するガードを選ぶ

"hg qselect" コマンドを実行することで，その時点でアクティブなガードを指定することができる．このコマンドによって，次に "hg qpush" が実行されたときにどのパッチが MQ によって適用されるかが指定される．それ以外の効果はなく，特にすでに適用されているパッチに対しては何も行わない．

引数なしで "hg qselect" コマンドを実行すると，現在有効なガードのリストを 1 行に 1 つずつ表示する．各々の引数は適用されるガードの名前として解釈される．

```
$ 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に興味がある場合，これは guards ファイルに保存されている．

```
$ cat .hg/patches/guards
foo
```

選択されたガードの効果は “hg qpush” の実行時に見ることができる．

```
$ hg qpush -a
applying hello.patch
applying goodbye.patch
now at: goodbye.patch
```

ガードを “+” や “-” の文字で始めることはできない．ガード名は空白を含んではならないが，その他の殆んどの文字を含むことができる．もし使用不可の文字を含む場合は、MQ が警告を表示する。

```
$ hg qselect +foo
abort: guard '+foo' starts with invalid character: '+'
```

選択されたガードを変更することは、適用されるパッチの変更を意味する。

```
$ 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
```

下の例でネガティブガードがポジティブガードよりも優先度を持つことがわかる。

```
$ 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

MQ が適用するパッチを決定する規則は下記のとおりである。

- ガードのないパッチは常に適用する

- パッチに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れば、そのパッチをスキップする
- パッチに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れば、そのパッチを適用する
- パッチがポジティブ・ネガティブいずれかのガードを持つが、現在選択されているガードとマッチするものがなければ、そのパッチをスキップする

13.6 作業環境を縮小する

以前言及したデバイスドライバでの作業では、Linux カーネルツリーにパッチを適用していなかった．その代わり，Infiniband ドライバの開発に関連したソースとヘッダファイルだけを持つリポジトリを用いた．このリポジトリはカーネルリポジトリの 1%ほどのサイズで，作業が容易である．

ここでパッチを適用するための “base” バージョンを選ぶ．これは任意に選んだ Linux カーネルツリーのスナップショットで，作成する際，カーネルリポジトリのチェンジセット ID をコミットメッセージに記録しておく．スナップショットはカーネルツリーの関連する部分の原型を保っているため，自分のパッチを開発用の個別のリポジトリに適用するのと同様に通常のカーネルツリーに適用することができる．

通常，パッチが適用されるベースツリーは上流のツリーのごく最近のスナップショットであるべきだ．こうすることで，開発したパッチを修正することなく，あるいはごく僅かな修正のみで，上流へ提出することが可能になる．

13.7 `series` ファイルを分割する

筆者は `series` ファイルのパッチをいくつかの論理的なグループにカテゴライズしている．パッチのそれぞれのセクションはあとに続くパッチが何であるか説明するコメントブロックで始まっている．

その後にメンテナンスしているパッチグループが連なる．グループを整理することは重要である．まずグループについて説明した後，この理由に触れる．

- “accepted” グループ．開発チームが Infiniband サブシステムのメンテナに提出し，受理されたが，小リポジトリのベースとなっているスナップショットにはまだ取り込まれていないもの．これらはツリーを上流メンテナのリポジトリと同様の状態にするために存在する “読み出しのみ” のパッチである．
- “rework” グループ．提出したが上流メンテナが受理に先だって変更を要求したもの．
- “pending” グループ．上流メンテナにはまだ提出していないが，手元での作業は完了したもの．これらはしばらくの間 “read only” になっており，上流のメンテナへ提出して受理された時は “accepted” グループへ移動される．メンテナが変更を要求した場合は “rework” グループへ移動される．
- “in progress” グループ．このグループには，アクティブに開発されており，まだどこへも提出されるべきではないパッチが含まれる．
- “backport” グループ．ソースを古いバージョンのカーネルツリーへ適合させるパッチからなる．
- “do not ship” グループ．なんらかの理由によって決して上流へ提出されるべきでないパッチからなる．例えば，ツリー外のドライバとベンダによって配布されているドライバを識別しやすくするためにドライバの識別文字列を変更するようなパッチが該当する．

パッチをこのような方法で整理する理由の説明へ戻ろう．一般に，パッチスタックの底にあるパッチはなるべく安定であって欲しいと思うものだ．そうであれば，パッチのコンテキストで変更による再作業をせずに済む．`series` に最初に追加するパッチは決して変更されないものにするとよい．

また，適用のために変更しなければならないパッチについては，できるだけ上流のツリーに近いソースツリーに適用できるようになっていて欲しいと考える．これが受理されたパッチをしばらくの間保持している理由である．

“backport” と “do not ship” パッチは `series` ファイルの末尾に置かれる．“backport” パッチは他のパッチすべての上から適用されなければならないし，“do not ship” パッチは安全なところに置かれていなければならない．

13.8 パッチシリーズを管理する

この作業では，どのパッチが適用されるか制御するためにいくつかのガードを用いている．

- “Accepted” パッチは `accepted` によってガードされている．筆者はこのガードをほぼいつも有効にしている．すでにパッチが存在しているツリーの上に新たなパッチを適用する時は，このパッチをオフにして，後続のパッチがクリーンに適用されるようにすることができる．
- 完成されているものの，まだ提出されていないパッチはガードを持たない．このパッチスタックを上流ツリーのコピーに適用する場合，ガードを全く使わずに十分安全なソースツリーを得ることができる．
- 再提出のために作業が必要なパッチには `rework` ガードを用いる．
- 開発中のパッチには `devel` ガードを用いている．
- バックポートパッチはパッチが適用されるカーネルのバージョンそれぞれに対してガードを持つことがある．例えば，あるコードを 2.6.9 にバックポートするパッチは `2.6.9` というガードを持つ．

多くのガードを用いることによって，柔軟に作業対象のソースツリーを選ぶことができる．ビルド処理中に正しいガードを自動的に選択できることがほとんどだが，特殊な状況のために手で選ぶことも可能である．

13.8.1 バックポートパッチを書く技術

MQ を使ってバックポートパッチを書くのは単純な作業である．そのようなパッチがすべきことは，古いバージョンのカーネルに備わっていない機能を使うコードを変更し，古いバージョンのカーネルでも正しく機能し続けるようにすることだけである．

よいバックポートパッチとは，そのパッチによってあたかもコードがターゲットの古いバージョンのカーネルに向けて書かれたような状態になるものである．無理のないパッチは理解しやすく，メンテナンスも容易である．バックポートパッチをいくつも書いているならば，コード内の大量の `#ifdef` で条件的コンパイルされるコードによる混乱を避けるべきである．そのためにバージョンに依存した`#ifdef` ではなく，ガードによって制御される条件コンパイルによらない変更を加えるべきである．

バックポートパッチを通常の変更を加えるパッチと別のグループに分割する理由が 2 つある．第一の理由は，2 つを混ぜ合わせることで，上流のメンテナへパッチを提出するために `patchbomb` エクステンションのようなツールを使うのが難しくなることである．第二の理由は，バックポートパッチは，後続の通常パッチの適用されるコンテキストを乱すことがあり，そのような場合，通常パッチはバックポートパッチなしに正しく適用することが不可能になってしまう．

13.9 MQ による開発の tips

13.9.1 ディレクトリ内でパッチを管理する

MQ を用いて比較的大規模なプロジェクトで作業している場合，大量のパッチが蓄積することが少なくない．例えば筆者のところには，250 以上のパッチを含むパッチリポジトリがある．

もしこれらのパッチを論理的なカテゴリに分割することができ，別々のディレクトリに収容したいと考えるならば，パッチ名にパスセパレータを含めても MQ は問題なく取り扱うことができる．

13.9.2 パッチのヒストリを見る

長い期間にわたって多くのパッチを作ってきたのなら， 12.11節で議論してきたようにそれらをリポジトリで管理するのは良い考えである．そうした場合，すぐに “`hg diff`” コマンドではパッチのヒストリを見ることに使えないことに気づくだろう．これは実際のコードの二次的な派生物（diff の diff）を見ていることと，MQ がパッチを更新する時にタイムスタンプとディレクトリ名を変更することで，プロセスにノイズを加えているためでである．

しかし Mercurial に同梱されている extdiff エクステンションを使い，2 つのバージョンのパッチ間で意味のある差分を取ることができる．このために patchutils [?] を使う必要がある．このパッケージは 2 つの diff ファイル間の差分を diff として表示する interdiff というコマンドを提供している．同じ diff の異なるバージョン間で使用すれば，前後の diff の間の diff を生成する．

hgrc ファイルの [extensions] セクションに次の行を追加する通常の方法で extdiff を有効にできる．

```
[extensions]
extdiff =
```

interdiff コマンドは 2 つのファイル名を渡されることを想定しているが，extdiff 拡張は，2 つのディレクトリを引数として取るコマンドを起動する．従ってこれらの 2 つのディレクトリの各々のファイルに対して interdiff を起動する小さなプログラムが必要になる．この本のソースコードリポジトリディレクトリ内の examples ディレクトリにある hg-interdiff というプログラムがそれだ．

```
#!/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

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
    else:
        if os.path.isfile(base):
            yield '', base

# 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):
    if f:
        path = os.path.join(base, f)
    else:
        path = base
    # interdiff requires two files; use /dev/null if one is missing.
    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)
```

シェルサーチパスの中に hg-interdiff があれば，MQ パッチディレクトリの中から次のように起動できる．

```
hg extdiff -p hg-interdiff -r A:B my-change.patch
```

この長いコマンドは頻繁に使うことになるだろう．そこで hgrc を編集し，hgext で通常の Mercurial コマンドのように使えるようにする．

```
[extdiff]
cmd.interdiff = hg-interdiff
```

これは hgext に interdiff コマンドが利用可能であることを指示するため，以前の “hg extdiff” の起動を若干簡単にできる．

```
hg interdiff -r A:B my-change.patch
```

Note:
interdiff コマンドは，パッチのバージョンが生成された元のファイルが同一に保たれている場合に限って正しく動く．パッチを作成してから元のファイルを変更し，パッチを再生成した場合は interdiff は有用な差分を出力しないかもしれない．

extdiff エクステンションは MQ パッチの表現を改善する以上のことをする．このエクステンションについて更に知りたい時は 14.2セクションを参照されたい．

Chapter 14

拡張による機能の追加

機能の観点から見ると Mercurial はかなり完備しているが，派手な機能については意図的に排除している．Mercurial のメンテナとユーザの双方にとって単純さを保つためにこのアプローチを取っている．

しかしながら，Mercurial は融通の利かないコマンドセットを提供しているのではない．*extensions*（あるいは *plugins* と呼ばれることもある）によって機能を追加することができる．これらのいくつかについては，以前の章で見ている．

- セクション 3.3は `fetch` エクステンションをカバーしている．これは新しい変更を pull し，ローカルな変更とマージを単一のコマンド "`hg fetch`" で実行する．
- 10では，フックに関連したいくつかの拡張について扱う．`acl` はアクセス制御リストを追加する．`bugzilla` は Bugzilla 機能の統合機能を提供する．`notify` は，新たな変更の際に通知電子メールを送る機能を提供する．
- Mercurial Queue というパッチマネージメント拡張は，非常に重要なので 2 章と付録 1 章を費やして説明する．Chapter 12は基本的な機能を説明する．chapter 13では高度な機能について説明し，appendix Bでは各コマンドの詳細を説明する．

この章では Mercurial で利用可能なその他の拡張について取り扱い，また自分で Mercurial 拡張を書く時に役立つ内部の機構についても説明する．

- 14.1節では `inotify` 拡張を用いることで得られる**大きな**性能向上について述べる．

14.1 `inotify` 拡張による性能向上

Mercurial の最も多用されるコマンドのいくつかが数百倍速くなることに興味があるならばぜひ読んで欲しい！

通常の条件下で Mercurial は高い性能を持っているが，"`hg status`" コマンドを実行した時，Mercurial はほぼ全てのディレクトリとファイルをスキャンすることになる．他の多くの Mercurial コマンドは，このような操作を意識させないようになっている．例えば "`hg diff`" はステータス機構を用いて，明らかに変更されていないファイルの比較を避けている．

良い性能を得るためには，ファイルステータスの取得が重要な関心事となるため，Mercurial の作者たちはこれをぎりぎりのところまで最適化している．しかし "`hg status`" コマンドではこれを避ける手立てがない．Mercurial は，管理しているファイルが最後にチェックした時から変更されているか調べるために，少なくとも一つの高価なシステムコールをする必要がある．ある程度以上大きなリポジトリでは，この操作には長い時間を要する．

この影響について調べるために 150,000 のファイルを擁するリポジトリを作成した．**まったく**変更がない場合でも "`hg status`" コマンドの実行には 10 秒を要した．

近年のオペレーティングシステムは，ファイル通知の機構を備えている．プログラムが適切なサービスに登録すると，オペレーティングシステムは対象となるファイルの作成，変更，削除をプログラムに通知する．Linux システムではこれを行うカーネルコンポーネントは `inotify` と呼ばれる．

Mercurial の inotify 拡張は，“hg status” コマンドを最適化するために，カーネルの inotify コンポーネントへアクセスする．この拡張は 2 つのコンポーネントからなる． inotify サブシステムから通知を受け取るためのデーモンがバックグランドで動作する．このデーモンは Mercurial の他のコマンドからの接続も受け付ける．この拡張は Mercurial の挙動を変更し，ファイルシステムをスキャンするのではなく，デーモンへの問い合わせを行うようにする．デーモンはリポジトリの状態を完全に把握しているので，直ちに問い合わせに返答することができ，リポジトリのディレクトリとファイルのスキャンを避けることができる．

プレーンな Mercurial では “hg status” コマンドが 150,000 ファイルのリポジトリに対して 10 秒を要していたことを思い出して欲しい。inotify 拡張を使った場合、所要時間は 0.1 秒に下がり，*100* 倍速くなっていることが分かる．

さらに進む前に，注意点を挙げる．

- inotify 拡張は Linux 特有のものである．この機能拡張は Linux の inotify サブシステムに直接アクセスするため，他のオペレーティングシステムでは動作しない．
- 2005 年初め以降にリリースされたどのような Linux ディストリビューションでも動作するはずだが，古いディストリビューションでは inotify を欠いていたり，必要なインターフェースサポートを glibc が提供していなかったりする可能性がある．
- 全てのファイルシステムが inotify 拡張で利用可能なわけではない．例えば Mercurial をいくつかのシステムで動作させている場合，同一のネットワークファイルシステムを各々のシステムでマウントしていることが多いが， NFS などのネットワークファイルシステムは考慮されていない．

2007 年 5 月までは inotify 拡張は Mercurial に同梱されていなかった．そのため，セットアップは他の拡張に比べてやや複雑だが，得られる性能向上にはそれだけの価値がある．

現在，拡張は 2 つのパートに分かれている．Mercurial ソースコードへのパッチと inotify サブシステムへの Python バインディングライブラリである．

> **Note:**
> Python の inotify バインディングライブラリは *2* **つ**ある．1 つは pyinotify であり，いくつかの Linux ディストリビューションでは python-inotify としてパッケージ化されている．これはバグが非常に多く，実用するには非効率的であり，使うべきで**ない**.

先へ進むに当って，正しく機能する Mercurial がインストールされていることが望ましい．

> **Note:** 以下の指示に従ってインストールした Mercurial を，最新の Mercurial コードで**置き換える**ことができる．(警告されなかったとは言わないこと)

1. Python inotify バインディングのリポジトリをクローンし，ビルドとインストールを行う．

```
hg clone http://hg.kublai.com/python/inotify
cd inotify
python setup.py build --force
sudo python setup.py install --skip-build
```

2. Merecurial の crew リポジトリをクローンする．inotify パッチリポジトリをクローンし，Mercurial Queues が crew リポジトリにパッチを当てられるようにする．

```
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 拡張（mq）が有効になっていることを確認する．MQ を使ったことがなければ，12.5を一読することをおすすめする．
4. inotify リポジトリへ行き，“hg qpush” コマンドに-a オプションを使って inotify のパッチをすべて適用する．

```
cd inotify
hg qpush -a
```

“hg qpush” でエラーが起きた場合，先へ進まずに助言を求めて欲しい．

5. パッチを当てた Mercurial をビルドしてインストールする．

```
python setup.py build --force
sudo python setup.py install --skip-build
```

適切なパッチの当った Mercurial をビルドすれば，後は inotify 拡張を利用するように hgrc を設定するだけである．

```
[extensions]
inotify =
```

inotify が有効になっていると，Mercurial はリポジトリ内でコマンドが最初に実行された時に自動的かつ透過的にステータスデーモンを起動する．リポジトリ 1 つごとに 1 つのステータスデーモンが起動される．

ステータスデーモンは何も出力を行わず，バックグラウンドで動作する．inotify 拡張を有効にした後で別のリポジトリ内でいくつかコマンドを実行し，実行中のプロセスのリストを見ると，いくつかの hg プロセスがカーネルからのアップデートと Mercurial からの問い合わせを待っているのが見られるだろう．

inotify 拡張を有効にし，リポジトリ内で Mercurial コマンドを最初に実行した時は通常の Mercurial コマンドと同様の性能で動く．ステータスデーモンは通常のステータススキャンを行い，カーネルから更新の通知を受けるためのベースラインを取得しておかなければならないからである．一方で，後続のステータスチェックを行う**すべての**コマンドは，そこそこの大きさのリポジトリに対しても目に見えて高速化される．さらにリポジトリが大きくなるに連れて性能の向上は大きくなる．inotify デーモンは，リポジトリのサイズに関わらず，ステータス取得をほぼ瞬時に行うことができる．

“hg inserve” コマンドを使ってステータスデーモンを手動で実行することも可能である．これにより，デーモンがどのように動作するのかをやや細かくコントロールすることができる．当然ながら，このコマンドは inotify が有効の場合のみ利用可能である．

inotify 拡張を使っている時は Mercurial の**全く変化なし**という動作に気付くだろう．ステータスに関係するコマンドは以前よりずっと高速になっているという一つの例外を除いてコマンドは特別の出力も結果も出力しないことに留意されたい．何か特別なことが起きたらバグとして報告して欲しい．

14.2 extdiff 拡張による柔軟な diff サポート

Mercurial のビルトインコマンド “hg diff” は unified 形式の diff をプレーンテキストで出力する．

```
$ hg diff
diff -r dcb3f0d99557 myfile
--- a/myfile        Tue Jun 09 06:06:55 2009 +0000
+++ b/myfile        Tue Jun 09 06:06:56 2009 +0000
@@ -1,1 +1,2 @@
 The first line.
+The second line.
```

変更を外部ツールを使って表示したいならば，extdiff 拡張を使うと良い．この拡張は例えばグラフィカルな diff 表示を行う．

extdiff 拡張は Mercurial に同梱されているので，セットアップは容易である．hgrc の中の [extensions] セクションに 1 行の設定を追加するだけで良い．

```
[extensions]
extdiff =
```

この拡張で “hg extdiff” というコマンドが使えるようになる．このコマンドはシステムの diff を使って組み込みの “hg diff” コマンドと同様の unified 形式の diff を生成する．

```
$ hg extdiff
--- a.dcb3f0d99557/myfile        Tue Jun  9 06:06:56 2009
+++ /tmp/extdiffg47cbx/a/myfile        Tue Jun  9 06:06:55 2009
@@ -1 +1,2 @@
 The first line.
+The second line.
```

得られる成果物は組み込みの “hg diff” と全く同じにはならないだろう．理由は diff コマンドはシステム毎に異なり，同じオプションを渡しても同じ出力をするとは限らないからだ．

出力の “making snapshot” 行が主張するように，“hg extdiff” コマンドはソースツリーの 2 つのスナップショットを作るように働く．1 つ目のスナップショットはソースリビジョンで，2 つ目はワーキングディレクトリのターゲットリビジョンである．“hg extdiff” コマンドはこれらのスナップショットをテンポラリディレクトリに作り，外部 diff ビューアにそれぞれのディレクトリの名前を渡す．その後，テンポラリディレクトリを消去する．効率のために 2 つのリビジョン間で変更のあったディレクトリとファイルのスナップショットだけを取る．

スナップショットディレクトリの名前はリポジトリと同じベースネームを持つ．リポジトリのパスが/quux/bar/foo ならば，foo がスナップショットディレクトリのベースネームになり，ここにチェンジセット ID が付加される．スナップショットのリビジョンが a631aca1083f なら，ディレクトリは foo.a631aca1083f となる．ワーキングディレクトリのスナップショットはチェンジセット ID を持たないので，この例では foo という名前になる．実際の動作を見るには，上記の “hg extdiff” を参照されたい． diff のヘッダにはスナップショットディレクトリ名が埋め込まれている点に留意されたい．

“hg extdiff” コマンドには重要なオプションが 2 つある．-p オプションで差分を取るコマンドとして diff 以外のコマンドを指定できる． -o オプションで “hg extdiff” が外部プログラムに渡すオプションを変更することができる．（デフォルトでは “-Npru” が渡される．これは diff が起動される時のみ意味を持つ．）その他の点では “hg extdiff” コマンドはビルトインの “hg diff” と同様の動作をする．すなわち，同じオプション名，文法，リビジョンやファイルを指定する書式などを持つ．

ここでシステムの通常コマンドの diff を，unified diff 形式ではなく，（-c オプションを使って）context diff 形式を出力させる例を見てみよう．context diff に含まれるコンテキストの行数もデフォルトの 3 行ではなく，（-C へ 5 オプションを渡して）5 行出力させるようにする．

ビジュアル diff ツールの起動はたやすい．ここでは kdiff3 ビューアの起動方法を示す．

```
hg extdiff -p kdiff3 -o ''
```

利用しようとする diff ビューコマンドがディレクトリを扱えない場合，わずかなスクリプトを書くことで簡単にこの問題を回避することができる．実際に使われているスクリプト例としては，mq 拡張と interdiff コマンドの間のものがある．これについては 13.9.2を参照されたい．

14.2.1 コマンドのエイリアスを作る

“hg extdiff” コマンドと diff ビューア双方のオプションを覚えておくのは厄介なため，extdiff 拡張で diff ビューアを正しいオプションで起動する**新しい**コマンドを定義することができる．

このためには hgrc を編集し [extdiff] という名前の新しいセクションを追加すればよい．このセクション内では複数のコマンドを定義することができる．kdiff3 コマンドを追加する方法を例として示す．一度定義を行えば，“hg kdiff3” とタイプすることで extdiff 拡張が kdiff3 を実行する．

```
[extdiff]
cmd.kdiff3 =
```

上の例のように右辺を空にしておくと，extdiff 拡張は外部で起動すべきコマンド名として定義を用いる．この名前は重複してはならない．ここでは “hg wibble” という名前でコマンド kdiff3 を呼び出すを定義している．

```
[extdiff]
cmd.wibble = kdiff3
```

diff ビューアを起動する際のデフォルトオプションも定義できる．定義ではオプションを定義したいコマンドの前に “opts.” という接頭辞を付ける．この例では vim エディタを DirDiff オプション付きで起動する “hg vimdiff” コマンドを定義している．

```
[extdiff]
cmd.vimdiff = vim
opts.vimdiff = -f '+next' '+execute "DirDiff" argv(0) argv(1)'
```

14.3 transplant 拡張を用いたチェリーピッキング更新

（Brendan とよく話をする必要がある．）

14.4 patchbomb 拡張によって変更をメールする

多くのプロジェクトが “更新のレビュー” の文化を持っている．開発者たちは最終バージョンを共有リポジトリにコミットする前に，パッチを査読者達がいるメーリングリストへ変更を送り，チェックや受けたりコメントを貰ったりする．いくつかのプロジェクトでは門番のような人々を持っていたりさえする．彼らの仕事は，人々から送られた変更を，彼ら以外にアクセス権のないリポジトリへ適用することである．

Mercurial では patchbomb 拡張を用いるとレビューや適用のために変更をメールで送信することが容易になる．この拡張の名前は変更をパッチとしてフォーマットし，1 つのチェンジセット毎にメール 1 通を送信することに由来する．一連の変更を連続してメールで送信することを “爆撃” に見立てている．従って受信者が受け取るのは “パッチ爆弾” である．

通常，patchbomb 拡張の基本設定は hgrc の中で 1，2 行ほどである．

```
[extensions]
patchbomb =
```

拡張を有効にすると，“hg email” コマンドが使えるようになる．

“hg email” を起動するのに最も安全で良い方法は，まず最初に-n オプションを付けて実行してみることだ．これにより，実際の送信は行わずに何が起きるのかを見ることができる．変更に目を通して正しい変更を送信することを確認できたら，同じコマンドを-n オプションなしで実行する．

“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構文を受け付ける．例えばリビジョン 7 から tip までを含む変更を送信するなどのように指定が可能である．

```
hg email -n 7:tip
```

比較対象として**リポジトリ**を指定することもできる．リポジトリを指定し，リビジョンを指定しない場合，“hg email” はローカルリポジトリにあってリモートリポジトリにない全てのバージョンを送信する．リビジョンまたはブランチ名を指定すると（後者は-b を用いる），送信されるリビジョンに制限が加わる．

送信したい人達の名前を指定せずに “hg email” コマンドを使っても全く安全である．この場合，対話的に入力を求める．Linux または Unix 系のシステムを使っていれば，ヘッダの入力に使いやすい readline スタイルの編集機能を利用できる

ただ 1 つのリビジョンを送る場合，“hg email” コマンドはデフォルトでチェンジセットの説明の最初の行をメールの件名として用いる．

複数のリビジョンを送る場合，“hg email” コマンドは 1 つのチェンジセット毎に 1 通のメールを送信する．開始のメールには一連の変更の目的を記述する説明文を付ける．

14.4.1 patchbombs の挙動を変更する

全てのプロジェクトが同じようなメールによる変更の送付の習慣を持っているわけではない．patchbomb 拡張はコマンドラインオプションで様々な方法に対応できるようになっている．

- コマンドラインから-s オプションを使って，説明文になる件名を入力することができる．このオプションは件名として使われるテキストを引数として取る．
- メッセージの送信元のアドレスを変更するには-f オプションを用いる．このオプションはメールアドレスを引数として取る．
- デフォルトでは 1 メッセージ毎に unified 形式 diff を送信する．（このフォーマットについては 12.4を参照.）-b オプションを使えば，バイナリ形式のデータを添付することもできる．
- unified 形式 diff は通常メタデータのヘッダを持つ．--plain オプションを使うことでこれを割愛することができる．
- 通常，差分はパッチの説明と同じボディの中に “インライン” で書き込まれる．いくつかのメールクライアントでは最初の MIME パートのみから引用が可能なため，こうすることが最も多くの人にとってパッチの特定の部分にコメントしたりするのに好都合である．説明文と diff を別のボディパートにして送信したい場合は-a オプションを使う．
- メールでメッセージを送信する変わりに，-m オプションを使って mbox フォーマットで出力させることも可能だ．このオプションは出力ファイル名を一つオプションとして取る．
- diffstat フォーマットのサマリを各々のパッチに付け，説明文を付けたい場合は，-d オプションを使うと良い．diffstat コマンドはパッチされた各々のファイル名，影響を受けた行数，各ファイルがどれだけ変更されたかを示すヒストグラムをを含むテーブルを表示する．この情報は，パッチの複雑さについて定性的な理解を得るのに役立つ．

Appendix A

コマンドリファレンス

A.1 “hg add”—次回のコミットでファイルを追加

--include, also -I

--exclude, also -X

--dry-run, also -n

A.2 “hg diff”—ヒストリまたはワーキングディレクトリ内の変更を表示

指定したファイル、ディレクトリについてリビジョン間での差分を表示する。表示には unified diff フォーマットが用いられる。unified diff フォーマットについての説明は 12.4を参照のこと。

デフォルトではバイナリデータと考えられるファイルの差分は出力しない．この挙動は-a と --git によって変更できる．

A.2.1 オプション

--nodates option

diff ヘッダから日付と時間情報を省略する．

--空行を無視する, also -B

空行の挿入または削除だけの変更を表示しない．空白文字が含まれる行は空行とは見なされない．

--include, also -I

指定したパターンとマッチするファイルまたはディレクトリを対象に加える

--exclude, also -X

指定したパターンとマッチするファイルまたはディレクトリを対象から除外する

--text, also -a

このオプションが指定されなければ “hg diff” はバイナリと判定されたファイルに対する diff の生成を行わない．-a を指定すると “hg diff” は全てのファイルをテキストとして扱い，全てのファイルに対して diff を生成する．

このオプションは，ほぼ全てがテキストだが一部にヌル文字を含んでいるようなファイルに対して有用である．このオプションをバイナリデータが多く含まれるファイルに適用すると無意味な出力になるだろう．

--ignore-space-change, also -b

空白の変更のみの行について出力しない．

--git, also -g

git 互換の diff を出力する．

--show-function, also -p

ハンクヘッダの中に含まれている関数の名前を表示する．検索は単純な発見的方法で行う．この機能はデフォルトで有効にされており，-p オプションは下の例のように showfunc 設定を変更されるまで意味をなさない．

```
$ echo '[diff]' >> $HGRC
$ echo 'showfunc = False' >> $HGRC
$ hg diff
diff -r c4b31f0820e4 myfile.c
--- a/myfile.c        Tue Jun 09 06:06:45 2009 +0000
+++ b/myfile.c        Tue Jun 09 06:06:45 2009 +0000
@@ -1,4 +1,4 @@
 int myfunc()
 {
-    return 1;
+    return 10;
 }
$ hg diff -p
diff -r c4b31f0820e4 myfile.c
--- a/myfile.c        Tue Jun 09 06:06:45 2009 +0000
+++ b/myfile.c        Tue Jun 09 06:06:45 2009 +0000
@@ -1,4 +1,4 @@
 int myfunc()
 {
-    return 1;
+    return 10;
 }
```

--rev, also -r

比較する対象のリビジョンを一つ以上指定する．“hg diff” コマンドは，比較するリビジョンを指定するために-r オプションを 2 つまで受け付ける．

1. 親リビジョンのワーキングディレクトリと，現在のワーキングディレクトリの差分を表示．
2. 指定されたチェンジセットと現在のワーキングディレクトリの差分を表示．
3. 指定された 2 つのチェンジセット間の差分を表示．

2 つのリビジョンを指定する方法として，-r を 2 つ使ってもいいし，リビジョンレンジ記法を使っても良い．例えば，下の 2 つのリビジョン指定は等価である．

```
hg diff -r 10 -r 20
hg diff -r10:20
```

2 つのリビジョンを指定する時，リビジョン指定の順序には意味がある．“hg diff -r10:20” はファイルの内容がリビジョン 10 からリビジョン 20 に変わったとして差分を作成する．“hg diff -r20:10” であれば逆の意味になる．ワーキングディレクトリの差分を取る場合にはこのようにリビジョンの順序を逆にすることはできない．

--ignore-all-space, also -w

A.3 “hg version”—バージョン情報とコピーライト情報を表示する

このコマンドは現在動作中の Mercurial のバージョンとコピーライトライセンスを表示する．メッセージは 4 種類ある．

- 文字列 “unknown”． このバージョンの Mercurial は Mercurial リポジトリの中でビルドされたのではないため，自分自身のリビジョンは分からない
- “1.1” のような短い数値文字列．これはビルドリポジトリ内でタグ付けされた特定のリビジョンの Mercurial であることを示す．（これは正式リリース版であるということを必ずしも意味しない．Mercurial をビルドするリポジトリでタグを付けるのは誰でもどのリビジョンに対しても可能である）
- “875489e31abe” のような 16 進数文字列．これは Mercurial が表示されたリビジョンのビルドであることを示す．
- “875489e31abe+20070205” のような 16 進文字列＋日付．これはそのリビジョンに対してコミットされていないローカルな変更を加えたソースからビルドされた Mercurial であることを示す．

A.3.1 Tips and tricks

Why do the results of “hg diff” and “hg status” differ?

“hg status” を実行した時，Mercurial は次回のコミットで記録する変更のリストを表示する．“hg diff” を実行すると，“hg status” の**一部の**diff が表示されることに気付くだろう．これには二つの理由が考えられる．

1 つ目は “hg status” が “hg diff” が通常表示しない何種類かの変更を表示するためである．“hg diff” は通常 unified diff 形式で出力するが，これは Mercurial が追跡できる変更のいくつかを追跡できない．特に，古い diff ではファイルが実行かのうかどうかを表現できないが，Mercurial はこの情報を記録する．

“hg diff” コマンドで--git オプションを使っている場合，この情報を表示できる git 互換の diff を出力する．

2 つ目の考えられる理由は，“hg diff” を引数なしで呼んでいるため，“hg diff” がワーキングディレクトリの直接の親との差分を取っているためである．チェンジセットをマージするために “hg merge” を実行すると，ワーキングディレクトリは 2 つの親を持つことになる．（親を表示するには “hg parents” を使う．）“hg status” はコミットされていないマージの**両方**の親との差分を取るのに対して，“hg diff” は順序的に先の親との差分を取る．後の親との差分を取るためには-r オプションを使う．両方の親との差分を取る方法は存在しない．

バイナリの差分を安全に取得する

大半がテキストであるファイル同士や，多くのバイナリデータが含まれるファイル同士の差分を取るために-a を指定する場合，生成された差分は Mercurial の “hg import” コマンドやシステムの patch に用いることができない．

“hg import” で利用できるバイナリファイルの差分を生成するには，パッチ生成に “hg diff”–git オプションを指定すればよい．システムの patch コマンドからはこの差分は利用できない．

Appendix B

Mercurial Queues リファレンス

B.1 MQ コマンドリファレンス

MQ によって提供されるコマンドの概要については “hg help mq” を利用されたい.

B.1.1 “hg qapplied”—適用されたパッチの表示

“hg qapplied” コマンドは，適用されたパッチの現在のスタックを表示する．パッチは古いものから新しいものの順に表示される．そのため，リストの最後のパッチは “top” パッチとなる．

B.1.2 “hg qcommit”—キューの中の変更をコミットする

“hg qcommit” コマンドは，.hg/patches ディレクトリにある突出した変更をコミットする．このコマンドは.hg/patches ディレクトリがリポジトリの場合にのみ動作する．例えば “hg qinit -c” コマンドによってディレクトリを作ったり，“hg qinit” コマンドの後に “hg init” を実行した場合が相当する．

このコマンドは “hg commit --cwd .hg/patches” の短縮形である．

B.1.3 “hg qdelete”—series ファイルからパッチを消去する

“hg qdelete” コマンドは.hg/patches ディレクトリの series ファイルからパッチを消去する．パッチが既に適用されている場合はパッチをポップしない．デフォルトではパッチファイルを消去しないため，その用途では-f オプションを使う．

オプション:

-f パッチを消去する．

B.1.4 “hg qdiff”—最上位の適用されたパッチの diff を出力する

“hg qdiff” コマンドは最上位の適用されたパッチの diff を出力する．これは “hg diff -r-2:-1” と等価である．

B.1.5 “hg qfold”—いくつかのパッチを一つにマージ (または “fold”) する

“hg qfold” コマンドは複数のパッチを適用された再上位のパッチにマージする．再上位のパッチは関心のあるパッチ全ての変更の集合になる．

fold するパッチは適用されていてはならない．“hg qfold” は，どれかが適用されている場合はエラーを返して終了する．fold されるパッチの順序は重要で，“hg qfold a b” は，“a, b が続いている現在の再上位のパッチを適用する” という意味になる．

フォールドされたパッチのコメントは，目的のパッチのコメントに追加される．各々のコメントブロックは 3 つのアスタリスクによって分離されている．-e オプションによって，フォールドが完了した後に結合したパッチ／チェンジセットのコミットメッセージを編集することができる．

オプション:

-e 新たにフォールドされるパッチのコミットメッセージとパッチの説明を編集する．

-l フォールドされたパッチの新しいコミットメッセージ及びパッチの説明として，与えられたファイルを使用する．

-m フォールドされたパッチの新しいコミットメッセージ及びパッチの説明として，与えられたテキストを用いる．

B.1.6 “hg qheader”—パッチのヘッダ／説明を表示

“hg qheader” コマンドはパッチのヘッダまたは説明を表示する．デフォルトでは再上位に適用されたパッチのヘッダを表示する．引数が渡されると，指定されたパッチのヘッダを表示する

B.1.7 “hg qimport”—サードパーティのパッチをキューへインポートする

“hg qimport” コマンドは，series ファイルに外部のパッチのためのエントリを追加し，パッチを.hg/patches ディレクトリにコピーする．追加は再上位の適用済みパッチの直後に行われ，パッチのプッシュは行わない．

.hg/patches ディレクトリがリポジトリの場合，“hg qimport” は自動的にインポートされたパッチに対して “hg add” を行う．

B.1.8 “hg qinit”—MQ で使用するリポジトリを用意する

“hg qinit” コマンドは MQ で使用するリポジトリを用意する．.hg/patches というディレクトリが作られる．

オプション:

-c .hg/patches をリポジトリとして，コマンド実行時の権限で作成する．同時に status ファイルを無視するために.hgignore ファイルを作成する．

.hg/patches ディレクトリがリポジトリの場合，“hg qimport” コマンドと “hg qnew” コマンドは新しいパッチを自動的に “hg add” する．

B.1.9 “hg qnew”—新しいパッチを作成する

“hg qnew” コマンドは新しいパッチを作成する．このコマンドは必須の引数としてパッチファイルとして使用する名前を取る．新規に作成されたパッチは，デフォルトでは空であり，series ファイルに，現在の再上位の適用されたパッチの直後に追加され，直ちにそのパッチの上にプッシュされる．

“hg qnew” は，ワーキングディレクトリから変更されたファイルを見つけると，-f オプション（下記を参照）が使われない限り新しいパッチの作成を拒否する．この挙動のため，再上位の適用されたパッチの上に新たなパッチを適用する前に，“hg qrefresh” することができる．

オプション:

-f カレントディレクトリの内容が更新されている場合，新しいパッチを作成する．孤立した変更は新規に作成したパッチに追加され，このコマンドが終了するとワーキングディレクトリは変更なしの状態になる．

-m 与えられたテキストをコミットメッセージとして用いる．このテキストはパッチファイルの先頭でデータの前に記録される．

B.1.10 “hg qnext”—次のパッチの名前を表示する

“hg qnext” コマンドは series ファイルの次のパッチの名前を表示する．このパッチは，“hg qpush” を実行すると最上位の適用済みパッチとなる

B.1.11 “hg qpop”—スタックからパッチをポップする

“hg qpop” コマンドは適用されたパッチのスタックのトップからパッチを除去する．デフォルトではパッチを 1 つ除去する．

このコマンドはリポジトリからポップされたパッチを表すチェンジセットを除去し，ワーキングディレクトリをパッチの効果を除去するように更新する．

このコマンドはポップするパッチの名前やインデックスとして使うために，オプションの引数を取る．このコマンドは，名前が与えられると名前の付けられたパッチが最上位の適用済みパッチとなるまでパッチをポップする．番号が与えられた場合，“hg qpop” は番号を一連のファイルの中のエントリへのゼロから数え始める（空行とコメント行は数えない）インデックスとして取り扱う．このコマンドは与えたインデックスのパッチが最上位の適用済みパッチとなるまでパッチをポップし続ける．

“hg qpop” コマンドはパッチやシリーズファイルを読み書きしない．そのためすでに series ファイルから削除したパッチや，完全に消去したパッチを “hg qpop” しても安全である．後から述べた 2 つのケースでは，パッチを適用した時の名前使用する．

デフォルトでは “hg qpop” コマンドは，ワーキングディレクトリが変更されている場合はいかなるパッチもポップしない．この挙動は-f オプションによってオーバライド可能で，これにより，ワーキングディレクトリのすべての変更が取り消される．

オプション:

-a 適用されたすべてのパッチをポップする．このコマンドはパッチを適用する前の状態へリポジトリを戻す.

-f ポップ時にワーキングディレクトリへのあらゆる更新を強制的に戻す.

-n 名前付けされたキューからパッチを 1 つポップする.

“hg qpop” コマンドは status ファイルの末尾から，ポップされたパッチに対応する行を 1 行取り除く.

B.1.12 “hg qprev”—以前のパッチの名前を表示する

“hg qprev” コマンドは，series ファイル内にある，最上位の適用済みパッチの前パッチの名前を表示する．このパッチは “hg qpop” を実行すると，最上位のパッチとなる.

B.1.13 “hg qpush”—パッチをスタックにプッシュする

“hg qpush” コマンドはパッチを適用済みスタックの上に追加する．デフォルトでは，このコマンドはパッチを一つだけ追加する.

このコマンドは適用されたパッチひとつひとつについて新たなチェンジセットを作成し，ワーキングディレクトリにパッチの影響を適用するよう変更を加える.

チェンジセットを作成する時に用いられるデフォルトデータは次の通りである:

- コミットの日時とタイムゾーンには，現在の日時とタイムゾーンが用いられる．これらのデータはチェンジセットのアイデンティティを計算するのに用いられるため，パッチを “hg qpop” した後，再び “hg qpush” すると，push によるチェンジセットは pop した時と異なるアイデンティティを持つ.
- オーサーは “hg commit” コマンド用いられるデフォルト値が用いられる.
- コミットメッセージは，パッチファイルの最初の diff ヘッダの前のあらゆるテキストである．そのようなテキストが存在しない場合，パッチの名前を識別するのにデフォルトのコミットメッセージが使われる.

パッチが Mercurial のパッチヘッダ (XXX add link) を含む場合，パッチヘッダの情報がこれらのデフォルト値をオーバライドする.

オプション:

-a series ファイルのすべての未適用パッチをすべてプッシュする

-l パッチの名前をコミットメッセージの最後に追加する.

-m 正常にパッチが適用できなかった場合，パッチを 3 ウェイマージするためのパラメータを，キューにセーブされた他のエントリーから計算し，通常の Mercurial のマージ機構を用いて 3 ウェイマージを行う．

-n プッシュ中のマージに名前つきキューを用いる．

“hg qpush” コマンドは series ファイルを読むが変更は行わない．このコマンドはプッシュする各々のパッチを表す行を “hg status” ファイルに追加する．

B.1.14 “hg qrefresh”—再上位の適用済みパッチを更新する

“hg qrefresh” コマンドは最上位の適用済みパッチを更新する．このコマンドはパッチを変更し，パッチを表す古いチェンジセットを除去し，変更されたパッチを表す新たなチェンジセットを生成する．

“hg qrefresh” コマンドは，以下のような変更を探す．

- コミットメッセージへの変更．例えば パッチファイルの中の最初の diff ヘッダの前のテキストは，パッチを表す新しいチェンジセットに反映される．
- ワーキングディレクトリの中の管理されているファイルへの変更はパッチへ追加される．
- コマンド “hg add”, “hg copy”, “hg remove”, “hg rename” で管理されているファイルへの変更．追加されたファイル，コピー・リネーム先の名前はパッチに追加され，削除されたファイルとリネーム元の名前はパッチから削除される．

“hg qrefresh” は，変更を検知しなかったとしても，パッチを表す新たなチェンジセットを再生成する．これにより，チェンジセットのアイデンティティは，パッチを表していた以前のチェンジセットのものとは別のものになる．

オプション：

-e コミットとパッチの説明を好みのエディタで変更する．

-m コミットメッセージとパッチの説明を，与えられたテキストで行う．

-l コミットメッセージとパッチの説明を与えられたファイルによって行う．

B.1.15 “hg qrename”—パッチのリネーム

“hg qrename” コマンドはパッチをリネームし，series ファイル中のこのパッチのエントリを変更する．

引数 1 つを与えた場合，“hg qrename” は最上位の適用済みパッチをリネームする．引数 2 つの場合，1 番目の引数のパッチを 2 番目の引数の名前にリネームする．

B.1.16 “hg qrestore”—セーブされたキュー状態に復元する

XXX このコマンドが何をするか不明

B.1.17 “hg qsave”—現在のキュー状態をセーブする

XXX 同上

B.1.18 “hg qseries”—パッチ系列を全て表示

“hg qseries” コマンドは，series ファイルに含まれるパッチ系列全てを表示する．このコマンドはパッチ名だけを表示し，空行やコメントは表示しない．最初に適用されたパッチから最後に適用されたパッチの順に表示する．

B.1.19 “hg qtop”—現在のパッチの名前を表示

“hg qtop” コマンドは現在の最上位の適用済みパッチの名前を表示する．

B.1.20 “hg qunapplied”—未適用のパッチを表示

“hg qunapplied” コマンドは series ファイルに含まれるすべての未適用のパッチの名前を表示する．このコマンドは次に末尾にプッシュされるパッチから順に表示する．

B.1.21 “hg strip”—リビジョンとその子孫を削除

“hg strip” コマンドはリポジトリから 1 つのリビジョンとその子孫を削除する．このコマンドは削除されたリビジョンの影響をリポジトリから取り除き，ワーキングディレクトリを削除されたリビジョンの親の状態に更新する．

“hg strip” コマンドは，削除されたチェンジセットのバックアップ一式を保存するので，誤って削除した場合などには再適用することができる．

オプション：

-b 除去したチェンジセットと混交した無関係のチェンジセットをバックアップバンドルに保存する

-f ブランチが複数のヘッドを持っている場合，すべてのヘッドを消去する．XXX このオプションはリネームされるべきで，-f はペンディング状態の変更がある場合にリビジョンを除去するのに用いられるべきである．

-n バックアップバンドルを保存しない．

B.2 MQ ファイルリファレンス

B.2.1 series ファイル

series ファイルは MQ が適用できるすべてのパッチの名前を保持している．これは名前のリストとして表現されており，1 行に 1 つずつパッチ名を含む．前後の空白は無視される．

行にはコメントを含めても良い．コメントは “#” 文字で始まり，行末まで続く．空行とコメントのみの行は無視される．

コメントと空行のために series を手で編集する必要が生じることがある．例えば，あるパッチを一時的にコメントアウトして，“hg qpush” がパッチ適用時にそのパッチをスキップするようにするなどが考えられる．また，パッチの適用される順番を series を編集することによって変更することもできる．

series ファイルをリビジョンコントロールの下に置くこともサポートされている．このファイルが参照するすべてのパッチをリビジョンコントロール下におくことは良い考えである．“hg qinit” コマンドに-c オプションを渡して使い，直接パッチを生成した場合は，自動的にこのような状態になる．

B.2.2 status ファイル

status ファイルは MQ が現在適用しているすべてのパッチの名前とチェンジセットハッシュを持つ．series ファイルと違って，このファイルは編集されることを意図していない．このファイルはリビジョン管理したり，編集したりすべきではない．これは MQ によって内部の管理に用いられるべきものである．

Appendix C

ソースからMercurialをインストールする

C.1 Unix系システムでのインストール

Python (2.3 移行) のインストールされている，十分新しい Unix 系のシステムでは，Mercurial のソースからのインストールは簡単である．

1. http://www.selenic.com/mercurial/download から新しいソースの tarball をダウンロードする
2. tarball を解凍する:

```
gzip -dc mercurial-version.tar.gz | tar xf -
```

3. ソースディレクトリに入って，インストーラスクリプトを実行する．Mercurial がビルドされ，ユーザホームディレクトリにインストールされる．

```
cd mercurial-version
python setup.py install --force --home=$HOME
```

インストールを行うと，Mercurial はユーザホームディレクトリ内の bin サブディレクトリに収められる．このディレクトリがシェルのコマンドサーチパスにあるのを確認すること．

Mercurial の実行ファイルが他の Mercurial パッケージを発見できるように，おそらく環境変数 PYTHONPATH を設定することが必要になる．例えば，筆者のラップトップではこれを/home/bos/lib/python に設定している．実際の正しいパスはどのように Python をビルドしたかによるが，これを見つけ出すのは簡単な筈だ．もし不明であれば，前述のインストーラスクリプトの出力から，mercurial ディレクトリの中身がどこにインストールされたかを見れば良い．

C.2 Windowsでのインストール

Mercurial を Windows 上でソースからビルドしインストールするためには，多くのツールが必要で，それなりの技術的知識が必要であり，困難である．もしカジュアルなユーザであれば，このやり方はおすすめしない．Mercurial 自体をハックしたいのでなければ，バイナリパッケージをおすすめする．

もし Mercurial を Windows 上でソースからビルドしたいのであれば，Mercurial wiki http://www.selenic.com/mercurial/wiki/index.cgi/WindowsInstall の “hard way” の説明に従って行う．この方法では多くの面倒な作業が待ち構えていることを覚悟する必要がある．

Appendix D

Open Publication License

Version 1.0, 8 June 1999

D.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 D.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.

D.2 Copyright

The copyright to each Open Publication is owned by its author(s) or designee.

D.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.

D.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.

D.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).

D.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.

A 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.

B 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.