2章

Subversionの差分エディタ：存在論としてのインタフェース

カール・フォーゲル(Karl Fogel)

美しいコードの例とされるものはしばしば、明確な境界を持つ、容易に理解可能な問題の、局所的な解であったりしがちです。そのような例としては、Duff's Device(http://ja.wikipedia.org/wiki/Duff's_device)とか、rsyncのローリングチェックサムアルゴリズム(http://ja.wikipedia.org/wiki/Rsync#Algorithm)などがあります。これは、小さく簡潔な解だけが美しい、という理由からではなく、複雑なコードの良さを理解するには、ナプキンの裏に走り書きできる程度より大きな文脈が必要だからです。

ここでは、本の数ページを費してよいということなので、上記のようなものよりは大きな「美しい」ものを示そうと思います。それは流し読みする読者を瞬時に惹きつけるような美しさではありませんが、そのコードをしょっちゅう扱うプログラマが、その問題領域に対する経験を蓄積するにつれて、高く評価するようになるような美しさだと言えます。

私が提示する例というのは、アルゴリズムではなくインタフェース、具体的にはオープンソースのバージョン管理システムSubversion(http://subversion.tigris.org)の中で使われているプログラミングインタフェースです。このインタフェースは2つのディレクトリツリーの差を表現するためのものですが、一方のツリーを他方のツリーに変換するのにも使われます。Subversionでは、C言語の型`svn_delta_editor_t`がこのインタフェースの公式の名前になっていますが、通称の**差分エディタ**(delta editor)という名前でよく知られています。

Subversionの差分エディタは、プログラマが良いデザインに見出す複数の特性を体現しています。まず、問題を非常に自然な境界に沿って分割しているので、Subversionに追加する新しい機能を設計する人は誰もが、いつ何のためにそれぞれの関数を呼べばよいのか容易に分かります。また、プログラマに効率を最大化する(例えばネットワークを経由した不要なデータ伝送を無くす)機会をごく自然な形で提供しますし、追加作業(例えば処理の進捗率表示)を簡単に組み込むことを可能にしています。そしておそらく最も重要なことですが、その設計は機能強化や改訂に対して非常に弾力的に対応可能です。

そして、よい設計の出どころについて多くの人が薄々気づいている点を裏付けるように、差分エディタは1人の人物によって、ほんの数時間の間に作り出されたものなのです(ただしその人は、対象とする問題やコードについて熟知していました)。

何が差分エディタを美しいものとしているのか理解するために、まずはそれが解こうとしている問題について見てみることにしましょう。

バージョン管理とツリーの変換

Subversionプロジェクトのごく初期の段階で開発チームは、2つの類似した(たいていは関連を持った)ディレクトリツリーの違いを最少量で表現するという一般的なタスクが繰り返し行われるだろうと認識しました。バージョン管理システムであるSubversionの目的の1つは、個々のファイル内容だけでなく、ディレクトリ構造に対する変更も追跡することでした。

実際、Subversionのサーバ側リポジトリは基本的に、ディレクトリのバージョン管理を中心として設計されています。リポジトリは時とともに変更されていくディレクトリツリーに対する単なる一連のスナップショットに過ぎません。リポジトリ(格納庫)にコミット(投入・確定)される変更セットごとに、前のツリーからどことどこだけが違っているかが同定され、新しいツリーが生成されます。新しいツリーのうち、変更されなかった部分については、前のバージョンのツリーと記録内容が共有されます。前のバージョンについても同様ですから、この共有は一連のバージョンを遡って行われます。各バージョンは単調増加する整数のラベルを持ちます。このラベルが、各バージョンを一意に識別する**リビジョン番号**(改訂通番)となります。

ですから、リポジトリはリビジョン番号の順に並んだ無限に延びる配列だと思ってください。慣習として、リビジョン0は常に空っぽのディレクトリを表します。**図2-1**の例では、リビジョン1はそこからツリーがぶら下がっていて(通常、リポジトリに対する最初の記録内容に対応します)、他にはコミットされたリビジョンはありません。四角い箱はリポジトリに格納される仮想ファイルシステムのノードを表しています。ノードはディレクトリ(DIRと表示)またはファイル(FILEと表示)のどちらかです。

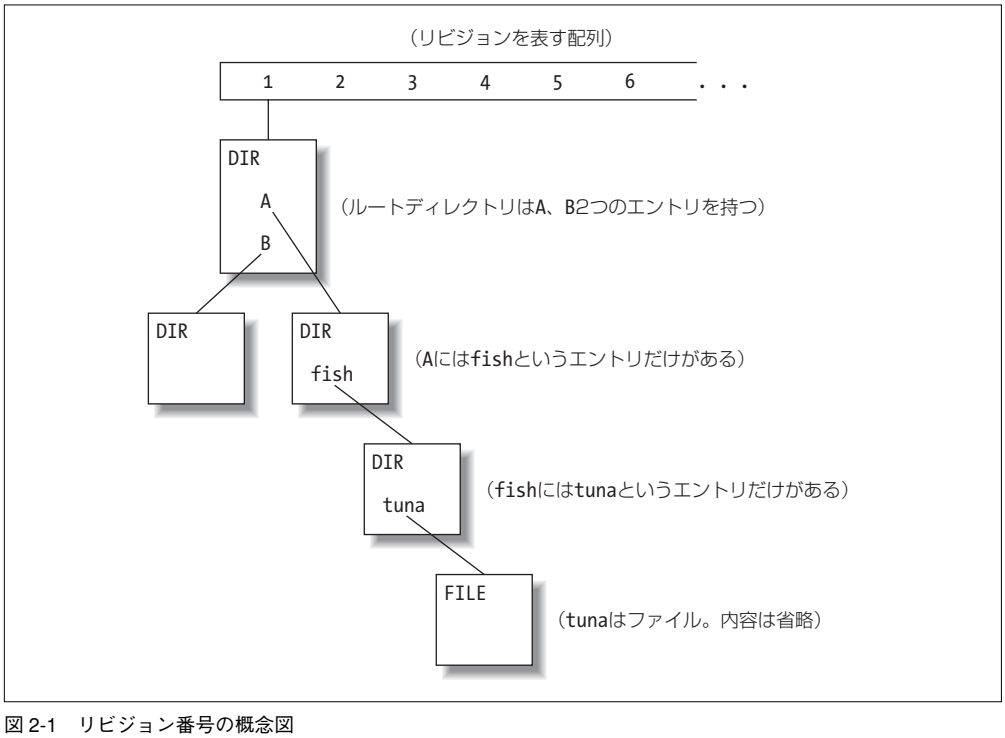

図2-1 リビジョン番号の概念図

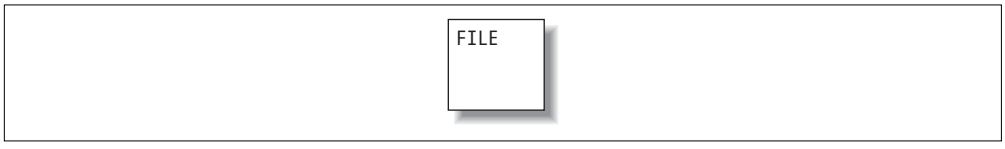

図 2-2　新しいノードが生成されたところ

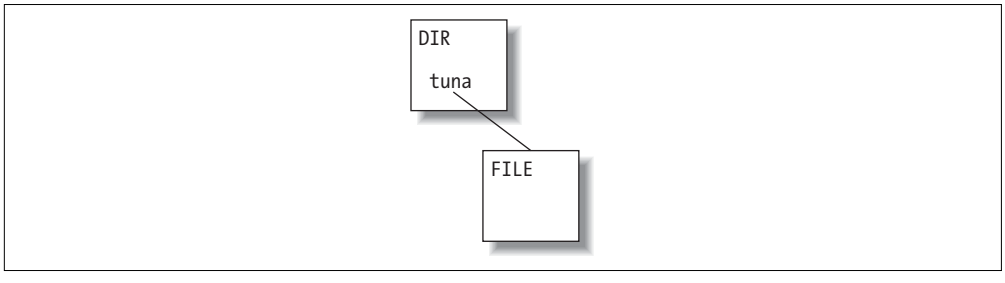

図 2-3　新しい親ディレクトリが生成されたところ

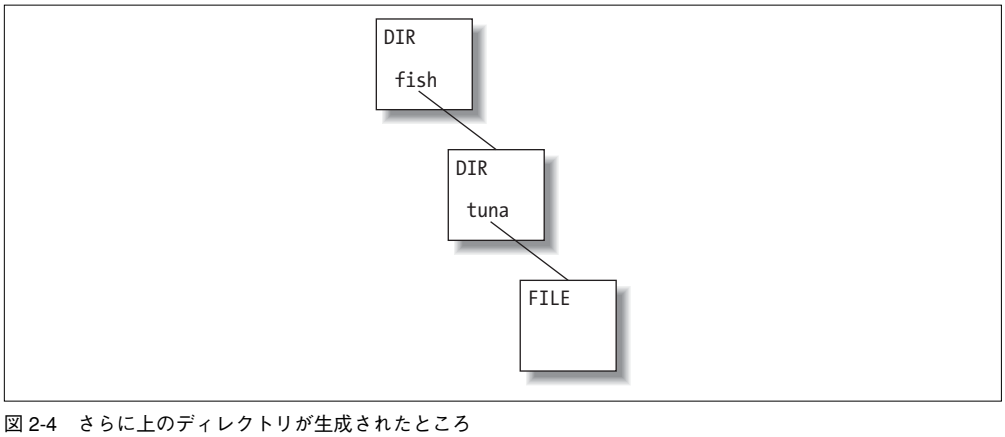

図 2-4　さらに上のディレクトリが生成されたところ

図2-1のツリーにおける、ファイルtunaを変更したとしたら、どうなるでしょうか？ まず、変更後の内容を含んだ新しいファイルノードを作ります。この新しいノードはまだどこにも接続されていません。**図**2-2に示すように、新しいノードはまだ宙ぶらりんです。

次に、変更されたファイルの親ディレクトリに対する新しいリビジョンを作ります。**図**2-3に示すように、親ディレクトリはファイルを参照していますが、これらの部分グラフ全体はまだどこからも指されていません。

さらに上のディレクトリについても新しいリビジョンを作っていきます(**図** 2-4)。

一番上まで来ると、ルートディレクトリの新しいリビジョンができます(**図**2-5)。このルートディレクトリは、エントリAについては新しいディレクトリAを指していますが、エントリBについては**古い**ディレクトリBを指しています。

これですべての新しいノードが書かれたので、「上昇」プロセスは終わりになり、新しいツリーをリビジョ

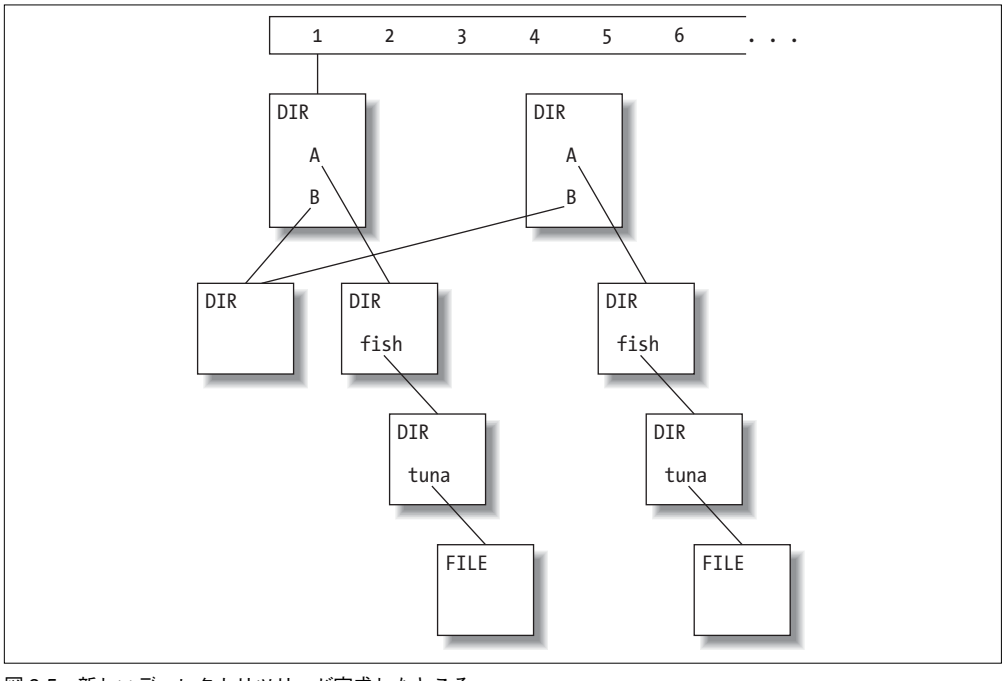

図 2-5 新しいディレクトリツリーが完成したところ

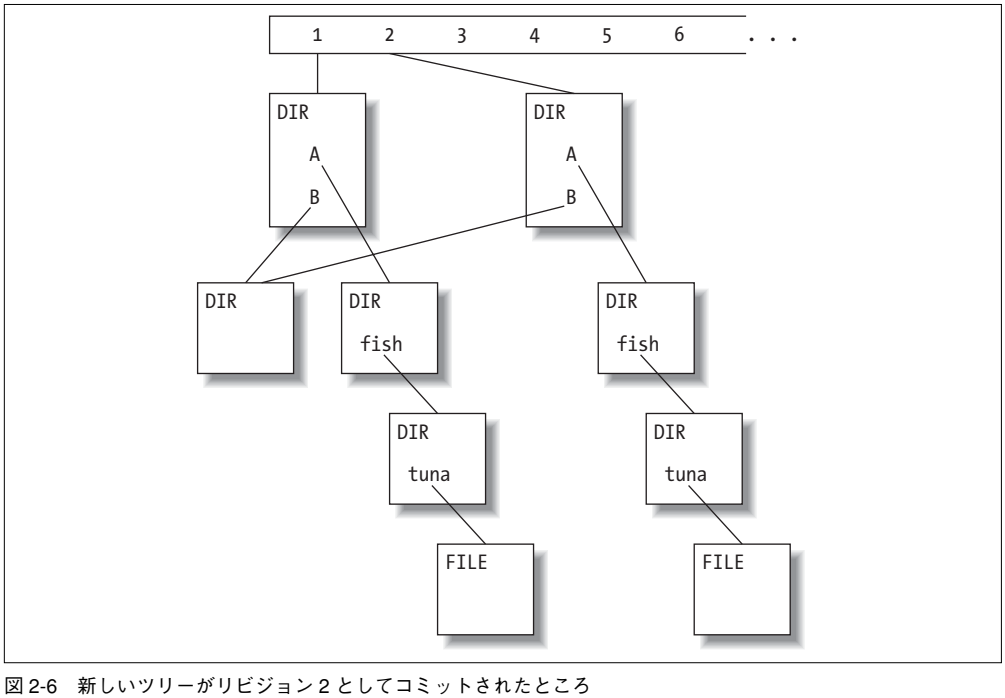

図 2-6 新しいツリーがリビジョン 2 としてコミットされたところ

ン配列の次のリビジョン番号(ここではリビジョン番号2)のところからリンクします(**図2-6**)。これで新しいツリーが他のユーザから見えるようになります。

このようにして、リポジトリ中の各リビジョンはそれぞれ固有のルートノードを指すようになります。そして、新しいリビジョンをコミットする時に書かれた変更部分は、各リビジョンと1つ前のリビジョンとの差分になります。変更を追跡するプログラムは、2つのツリーを並行して降りて行き、どのエントリが違うノードを指しているかを見ればいいのです。(ここでは説明を簡単にするために、いくつかの詳細を省略しました。例えば実際には、記憶領域を節約するために、古いノードは新しいノードを基準とする差分だけ記録するようにして領域を圧縮します。)

このツリーをバージョン管理するモデルは、この章で扱う内容(すぐ次に説明する差分エディタ)の中核になるわけですが、これにはいくつかの優れた特性があります(これをテーマとして、この本の1章分を書くこともできそうです)。優れた特性の代表的なものを挙げておきましょう。

読み込みが容易

ファイル/path/to/foo.txtのリビジョン*n*を取り出すには、まずリビジョン*n*のルートを取り出し、そこからパス名に沿ってたどるだけで済みます。

書き手が読み手と干渉しない

書き手が新しいノードをツリーの下から上向きに作っている間も、それと並行して動作している読み手は進行中の作業を見ることがありません。新しいツリーは、最後にリビジョン配列に新しいリビジョンへのリンクが書かれた時にはじめて、読み手に見えるようになります。

ツリー構造全体がバージョン管理される

ツリーの各リビジョンについて、ツリー構造そのものが保存されます。ファイルやディレクトリの改名、追加、削除などすべてがリポジトリ中の本質的な履歴の一部となります。

もしSubversionが単なるリポジトリであれば、これでお話は終わりになるところです。しかしSubversionでは、(サーバ側にある)リポジトリに加えて、クライアント側に**作業用コピー**があります。ユーザがリビジョンツリーのどれかをチェックアウトし、そこに何らかのローカルな変更を施し、しかしまだそれをコミットはしていない、という状態があり得るわけです。(実際には、作業用コピーは常に1つのリビジョンツリーに対応しているとは限らず、複数の異なるリビジョンから来たノードが混ざっていることも珍しくありません。しかし、そのような状況があるからといってここで扱うツリーの変換について大きな違いが生じるというわけではありません。そういうわけでこの章では、作業用コピーはどれか1つのリビジョンツリーに対応しているものとします。ただしそれが最新のリビジョンであるとは限りません。)

ツリーの差分を表現する

Subversionで最もよくある使い方は、2つのサイト間で変更を伝達することです。これは、他人の変更を受け取る時にはリポジトリから作業コピーへの伝達になりますし、自分が作業した変更を送る時には作業コピーからリポジトリへの伝達になります。2つのツリーの差分を表現することは、他のよく使われる操作の中枢部

分でもあります。例えば、差分のチェック、ある枝への切替え、ある枝に施されている変更を他の枝にマージする、などがその例です。

明らかに、サーバ→クライアントとクライアント→サーバで2つの異なるインタフェースを用意するのは馬鹿げています。どちらの場合でも本質的な作業は同じなのですから。ツリーの差分はツリーの差分であり、どちらのツリーがネットワーク経由で伝送されつつあるのかとか、受け取った側がそれをどう使おうとしているかとかは、作業の本質とは関係ありません。しかし、ツリーの差分を表現する自然な方法を見つけ出そうとしてみると、すごく難しいことが分かりました。さらに事態を複雑にしたのは、Subversionが複数のネットワークプロトコルをサポートし、複数の保管機構をサポートすることでした。そして私たちは、これらのどれを使っている時でも同じに見えるようなインタフェースを必要としていました。

私たちが最初に設計したインタフェースは、不十分であったり、明らかに面倒すぎるものだったりしました。それらをいちいち説明することはしませんが、それらに共通していたこととして、説得力のある答えのない、未解決の問題を残していたという点が挙げられます。

例えば、多くの案では変更のあったパス群をフルパス名やパスの一部分に対応する文字列として伝達していました。では、パス群をどのような順序で伝達するのがいいのでしょう？ 深さ優先？ 幅優先？ ランダム？ 辞書順？ ディレクトリとファイルでコマンドは同じか別か？ もっとも重要なこととして、差分を表す個々のコマンドはどうやってそれが変更全体をまとめた統合的なまとまりの一部だと分かるのでしょうか？ Subversionでは、ツリーに対する操作全体はユーザにとって明白に分かるものなので、プログラミングインタフェースがその概念とマッチしなければ、その尻拭いのためにたくさんの安定性のないコードを書くはめになるだろうと分かっていました。

困った挙げ句に、私はもう1人の開発者であるベン・コリンズ＝サスマン（Ben Collins-Sussman）と一緒に、シカゴを発ちインディアナ州ブルーミントンに赴きました。そこにいたジム・ブランディ（Jim Blandy）にアドバイスを求めるためです。彼はSubversionのリポジトリモデルを最初に発明した人であり、（はっきり言えば）デザインについて強い意見を持っていました。彼は私たちが木の差分を伝達しようとしたさまざまな試みについて説明する間、静かに聞いていましたが、私たちの説明が進むにつれて、彼の表情は次第に厳しいものとなっていきました。

私たちの説明が終わると、彼はしばらくの沈黙の後、しばらく考えたいので席を外して欲しいと丁寧に言いました。私はジョギングシューズを取り出し、ランニングに行きました。ベンは建物に残って別室で本を読んだりその他のことをしていました。共同でやったことはそれだけです。

私がランニングから戻ってシャワーを浴びてから、ベンと私はジムの部屋に戻り、そこで彼は彼の案を私たちに見せました。それが実質的に、現在のSubversionにあるのと同じものでした。何年も経つ間にさまざまな変更はなされましたが、その基本的な構造を変えることは一切なかったのです。

差分エディタのインタフェース

以下に、要約版の差分エディタのインタフェースを示します。コピーや名前変更を行う部分、Subversionプロパティ（バージョンつきのメタデータであり、本章の内容的には重要ではありません）、その他Subversion固有の管理部分については除いてあります。これらの除いた部分も含め、差分エディタの最新版は http://svn.collab.net/repos/svn/trunk/subversion/include/svn_delta.h にあります。以下に示すコードはr21731（リ

ビジョン 21731）に対応するもので、これは http://svn.collab.net/viewvc/svn/trunk/subversion/include/svn_delta.h?revision=21731 で取れます。

インタフェースを（要約版であっても）理解するためには、以下のSubversion用語を知っておく必要があります。

pool
: 引数poolはメモリプール、つまりメモリ割り付け用バッファを表します。多くのオブジェクトをバッファから割り付け、不要になったらバッファ単位でまとめて解放できるようにしているのです。

svn_error_t
: 関数からの返値の型として使われ、Subversion用のエラーオブジェクトを指すポインタ型となっています。各関数ともエラーが無い場合は null を返すようになっています。

text delta
: ファイルの2つの異なるバージョン間の差分を表します。一方のファイルに差分を適用することで、他方のファイルを得ることができます。Subversionではtextとあっても全部バイナリデータとして扱うので、ファイルの内容はプレーンテキストでも、音声データでも、画像データでも、それ以外のものでも構いません。差分は固定サイズのウィンドウ（範囲）の列として表されます。各ウィンドウには、バイナリの差分データのかたまりが含まれています。このようにすることで、メモリの最大使用量を、パッチの合計サイズ（画像ファイルなどでは極めて大きくなる可能性があります）ではなく、1つのウィンドウのサイズに比例した量に抑えることができます。

window handler
: 1つのウィンドウ分の差分を指定したファイルに適用する関数へのポインタを表します。

baton
: コールバック関数に渡す文脈データを表す void* 型の値です。ライブラリごとに void *ctx、void *userdata、void *closure などと宣言されるものと同様のものです。Subversion でこれを baton と呼んでいるのは、このデータがリレーのバトンのように非常にたくさん受け渡されるためです。

インタフェースは、コードの読み手が適切な心がまえで読み始められるような紹介から始まります。このテキストはジム・ブランディが2000年8月に書いた時からほとんど変わっていませんから、全体の概念は十分枯れていると言えるでしょう。

```
/** ツリーをたどって差分を処理。
 *
 * Subversion には、ツリーの差分を生成したり消費するさまざまなツールが含まれている。
 *
 * commit コマンドの処理の概要は次の通り:
 * - クライアント側は作業用コピーを調べ、コミットすべき変更に対応するツリーの差分を生成する。
 * - クライアント側のネットワークライブラリはそのデータを受け取り、対応する要求をネットワークを通してサーバ側に送る。
```

```
 * - サーバ側は要求を受け取り、クライアント側で生成したものと同一(のはず)の差分を復元する。
 * - Subversion サーバモジュールはその差分を受け取り、ファイルシステムに対する対応するトランザクションをコミットする。
 *
 * update コマンドの場合は処理の向きが逆になる:
 * - Subversion サーバモジュールはファイルシステムとやり取りし、クライアント側の作業用コピーを最新のものにするのに
 *   必要な差分を生成する。
 * - サーバはこの差分をもとに、対応する変更に対応する応答メッセージを組み立てる。
 * - クライアント側のネットワークライブラリはサーバから応答を受け取り、サーバ側と同一(のはず)のツリーの差分を復元。
 * - 作業用コピーライブラリはこの差分を受け取り、作業用コピーに必要な変更を施す。
 *
 * 最も単純なアプローチはツリーの差分を普通のデータ構造として表現することだろう。update の場合、サーバは差分デー
 * タ構造を生成し、作業用コピーライブラリはこのデータ構造で表現された差分を作業用コピーに適用する。この場合、ネット
 * ワークライブラリの仕事はデータ構造をそのまま送ることだけである。
 *
 * しかし我々は、差分データ構造が典型的なワークステーションのスワップ領域に入り切らなくなるほど大きくなる場合も
 * あると考えている。例えば、200Mb のソースツリーをチェックアウトする場合、ソースツリー全体が 1 つの差分で表され
 * ることになる。したがって、差分がスワップ領域に入らないくらい大きい場合をうまく扱うことが重要となる。
 *
 * そこで、ツリーの差分を明示的なデータ構造として表現する代わりに我々は、差分を消費する(受け取る)側が、生成する
 * 側から送られて来たらすぐに処理してしまう標準的な方法を定義する。svn_delta_editor_t データ構造は差分を消
 * 費する側が定義する一群のコールバック関数を格納するもので、それぞれの関数は差分を生成する側が(間接的に)呼び出
 * すことになる。コールバック関数の個々の呼び出しは、それぞれが差分の 1 要素(ファイルの内容変更、ファイルやディレ
 * クトリの名前変更、等)を表現している。
 */
```

引き続いて、公式のドキュメンテーションとしてのコメントと、インタフェース自体の定義が来ます。インタフェースはコールバック関数群を格納した表です。コールバック関数の呼ばれる順序には一定の制約が課せられます。

```
/** コールバック関数群を格納した構造。これらの関数は差分の生成側によって呼び出される。
 *
 * 関数の使用法
 * ==============
 *
 * ツリーの差分を表すためにこれらの関数を使う方法を以下で説明する。
 *
 * 差分の消費側はこのデータ構造に記されているコールバック関数群を実装し、差分の生成側がそれらを呼び出す。したがっ
 * て、呼び側(生成側)がツリーの差分を呼ばれ側(消費側)に向かってプッシュする(押し込む)ことになる。
 *
 * ツリーをたどり始める時に、消費側は差分エディタ全体が使うグローバルなバトン edit_baton を生成側に渡す。
 *
 * 次に、ツリーの差分がある(何らかの変更がある)場合は、生成側はこの edit_baton の値を関数 open_root に渡し、
 * 返値として変更のルートディレクトリを表すバトンを受け取る。
 *
```

```
* ほとんどのコールバック関数はその名前通りの動作をする:
*
*    delete_entry
*    add_file
*    add_directory
*    open_file
*    open_directory
*
* これらはいずれも、どのディレクトリに対して変更が適用されるかを表すバトンと、変更するファイルやサブディレクトリやディレ
* クトリエントリを表すパス名(変更中のツリーのルートを起点とする相対パス)を引数pathとして受け取る。エディタは通常、この
* 相対パスを編集バトンに対応付けられている起点(URLとか、OSのファイルシステム上の場所を表す絶対パスなど)と組み合わせて
* 使うことになる。
*
* add_directory や open_directory を含めて、すべての呼び出しは親ディレクトリに対応付けられたバトンを引数として
* 要求するので、開始時にどこから最初のディレクトリのバトンを貰って来るのかという問題がある。その答えは、関数open_root
* を呼ぶと変更のトップディレクトリに対応するバトンが返されるということである。一般に、生成側は実質的な作業に入る前に必ず、
* エディタの関数 open_root を呼ぶ必要があるだろう。
*
* open_root が変更されつつあるツリーのルートに対応するバトンを返すのに対し、add_directory や open_directory は追
* 加されたり変更開始されたディレクトリのバトンを返す。これらは他のコールバック関数と同様に、parent_batonと相対パスを
* 表す path を引数として受け取り、これらをもとにして追加されたり変更開始されたディレクトリのバトン child_baton を作っ
* て返す。生成側はサブディレクトリに対する変更を指示するときに、これらの child_baton の値を指定する。
*
* したがって、既に foo と foo/bar というサブディレクトリがあったとして、新しいファイル foo/bar/baz.c を作り出すには、
* 生成側は次のような呼び出しを行えばよい:
*
*    - open_root() --- トップディレクトリに対応するバトン root を返す
*
*    - open_directory(root, "foo") --- foo に対応するバトンを返す
*
*    - open_directory(f, "foo/bar") --- foo/bar に対応するバトンを返す
*
*    - add_file(b, "foo/bar/baz.c") --- ファイルを追加
*
* 生成側がディレクトリに対する変更を終えたら、関数close_directoryを呼ばなければならない。これに対応して、消費側は必
* 要な後始末を行い、バトンのために使っていた領域を解放することができる。
*
* add_file や open_file は生成されたり変更されつつあるファイルに対応するバトンを返す。このバトンを指定して
* 関数 apply_textdelta を呼ぶことで、ファイルの内容を変更できる。生成側がファイルに対する変更を終えたら、関数
* close_file を呼ばなければならない。これによって消費側は必要な後始末を行い、バトンのために使っていた領域を
* 解放することができる。
*
* 関数の呼び出し順
* ====================
```

```
 *
 * 生成側がバトンを使うに当たって、次の 5 つの制約がある。
 *
 * 1. 生成側はopen_directory、add_directory、open_file、add_fileを1つのディレクトリエントリに対して最大1回
 *    しか呼び出してはならない。delete_entry は 1 つのディレクトリエントリに対して最大 1 回しか呼び出してはならない(そ
 *    の後で同じ名前のディレクトリエントリを指定した add_directory や add_file を呼ぶことは許される)。
 *    open_directory、add_directory、open_file、add_fileをあるディレクトリエントリに対して呼び出した後で、同
 *    じディレクトリエントリを指定した delete_entry を呼び出してはならない。
 *
 * 2. 生成側があるディレクトリに対応するバトンをcloseするのは、そのすべてのサブディレクトリに対するcloseを実行した後
 *    でなければならない。
 *
 * 3. 生成側が open_directory や add_directory を呼ぶ時には、現在 open されているディレクトリのバトンのうちで、最も
 *    最近にopenされたものを渡さなければならない。言い方を変えれば、生成側は兄弟関係にある2つのディレクトリのバトンを
 *    同時に open しておくことはできない。
 *
 * 4. 生成側が open_file や add_file を呼んだら、それに引き続いてファイルへの変更(apply_textdelta 呼び出し)を行い、
 *    close_file を呼び出さなければならない。これらを行う前に他のファイルやディレクトリを操作することはできない。
 *
 * 5. 生成側がapply_textdeltaを呼んだら、それに引き続いてすべての必要なウィンドウハンドラ呼び出し(最後のNULLウィン
 *    ドウに対するものも含む)を行ったあとでなければ、他の svn_delta_editor_t の呼び出しを行うことはできない。
 *
 * そういうわけで、生成側はディレクトリやファイルのバトンをちょうど1つの深さ優先たどりを行っているのと同様の順序で使わな
 * ければならない。
 *
 * プールの使用法
 * ==========
 *
 * エディタの多くの関数は複数回呼び出され、その呼び出し順序はエディタの「ドライバ」により制御される。ドライバはエディタの
 * 関数が各周回で使うプールを生成し、関数の呼び出し時に適切なプールをパラメタとして渡す。
 *
 * エディタの各関数は深さ優先順で呼ぶ必要があるので、ドライバもこれに合わせてプールを入れ子構造に構成することが多い。しか
 * しこれは、トップレベルの要素に対応するプールが消去されるとツリーの下の方にあるプールもまた消去されることが保証されると
 * いう予防安全的な特性に過ぎず、インタフェース自体としては各関数呼び出しに渡されるプールの構成について特定の構造を仮定し
 * たり要求するものではない。
 */
typedef struct svn_delta_editor_t
{
  /** *root_baton は変更されるディレクトリのトップを指す(変更されつつあるサブツリーのトップであり、ファイルシステムの
   * トップである必要はない)。他のディレクトリバトンと同様、生成側は作業がすべて終わったら root_baton についても
   * close_directory を呼ばなければならない。
   */
  svn_error_t *(*open_root)(void *edit_baton,
                            apr_pool_t *dir_pool,
```

```
                          void **root_baton);

/** parent_baton が指定するディレクトリから path で指定された名前のディレクトリエントリを削除する。
 */
svn_error_t *(*delete_entry)(const char *path,
                             void *parent_baton,
                             apr_pool_t *pool);

/** pathという名前の新しいサブディレクトリを作る。 この関数が*child_batonに格納する値が、新しいサブディレクトリに
 * 変更を加える呼び出しにおいて parent_baton として渡される。
 */
svn_error_t *(*add_directory)(const char *path,
                              void *parent_baton,
                              apr_pool_t *dir_pool,
                              void **child_baton);

/** parent_batonで指定されたディレクトリ中のpathで指定された名前のサブディレクトリに対する変更を開始する。サブディ
 * レクトリの名前はpathで指定する。この関数が*child_batonに格納する値が、以後サブディレクトリに変更を施す関数にお
 * いて parent_baton として渡される。
 */
svn_error_t *(*open_directory)(const char *path,
                               void *parent_baton,
                               apr_pool_t *dir_pool,
                               void **child_baton);

/** dir_baton(add_directoryやopen_directoryで設定される)で指定されるサブディレクトリに対する処理を終わったの
 * で、以後このバトンは使用しない。したがって、このバトンに関連付けられた資源は解放してよい。
 */
svn_error_t *(*close_directory)(void *dir_baton,
                                apr_pool_t *pool);

/** path という名前の新しいファイルを作る。この関数が **file_baton に格納する値が、作成されたファイルに変更を施す
 * apply_textdelta の呼び出しにおいて渡される。
 */
svn_error_t *(*add_file)(const char *path,
                         void *parent_baton,
                         apr_pool_t *file_pool,
                         void **file_baton);

/** parent_baton で指定されたディレクトリ中の path で指定された名前のファイルに対する変更を開始する。この関数が
 * **file_baton に格納する値が、指定されたファイルに変更を施す apply_textdelta の呼び出しにおいて渡される。
 */
svn_error_t *(*open_file)(const char *path,
                          void *parent_baton,
```

```
                                apr_pool_t *file_pool,
                                void **file_baton);

  /** ファイル中のテキストに差分を適用し、ファイルの新しい版を作る。file_baton が作成ないし変更を行うファイルを表す。
   * file_baton の値はこれ以前に add_file や open_file から返されたものである。
   *
   * この関数は*handleにテキスト差分ウィンドウハンドラ関数を設定する必要がある。ハンドラ関数は、以後テキストの差分を受
   * け取るごとに繰り返し呼び出される。この関数が *handler_baton に設定した値が、ハンドラ関数にバトンとして渡される。
   */
  svn_error_t *(*apply_textdelta)(void *file_baton,
                                  apr_pool_t *pool,
                                  svn_txdelta_window_handler_t *handler,
                                  void **handler_baton);

  /** file_baton(add_directoryやopen_directoryで設定される)で表されるファイルに対する処理が終わったので、以後
   * このバトンは使用しない。したがって、このバトンに関連付けられた資源は解放してよい。
   */
  svn_error_t *(*close_file)(void *file_baton,
                             apr_pool_t *pool);

  /** 1つの差分に対する処理が終わったことを示す。edit_baton は編集全体に対するバトンである。
   */
  svn_error_t *(*close_edit)(void *edit_baton,
                             apr_pool_t *pool);

  /** エディタドライバが処理中止を決めた時に呼び出す。これによって、差分エディタは必要な後始末を行いお行儀よく終了できる。
   */
  svn_error_t *(*abort_edit)(void *edit_baton,
                             apr_pool_t *pool);

} svn_delta_editor_t;
```

これは芸術だろうか？

私には、このインタフェースの美しさがすぐ分かったとは主張できません。ジムにとってもそうだったと思います。たぶん彼は、私とベンを自分の家から追い払うためにこれを書いたのだとは思います。しかし彼もまたこの問題について長い間熟考し続けて来たのであり、ツリー構造がどのように振る舞うべきかという彼の直観にしたがって答えを出したわけです。

差分エディタインタフェースが見た人を驚愕させる最初の点は、それが制約を**選んでいる**ということでしょう。ツリーの編集を深さ優先順(ないしその他の順序)で行うといった要求はどこにも書かれていませんが、にもかかわらず、インタフェースはバトンの使用関係を通じて深さ優先順を強制しているのです。そしてこのことが、インタフェースの使用方法や振る舞いをより分かりやすくさせているのです。

2番目の点として、編集操作全体が目立たない形で、再びバトンを通じて、コンテキストを受け渡している点です。ファイルバトンには**その**親ディレクトリのバトンが格納でき、ディレクトリバトンにはその親ディレクトリのバトンが格納でき(編集全体のルートについては親はnullです)、そして編集全体を表すバトンはどこからでも参照されてよいのです。個々のバトンは消去され得るオブジェクトですが(例えば、ファイルをcloseするとそのファイルのバトンは消去されます)、どのバトンからでも全体の編集コンテキスト(例えばクライアント側が現在更新されつつあるのはどのバージョンに対してかといった情報を含んでいるかもしれません)が参照できます。ですから、バトンには複数の意味が重ねられています。バトンは編集の一部分に対するスコープ(ないし寿命、なぜならバトンはそれが割り付けられているプールが存在している間しか存在し続けられませんから)を表すと同時に、全体のコンテキストも表しているのです。

3つ目の重要な特徴は、インタフェースがツリーに対する変更に関与するさまざまな部分操作の境界を明示しているという点です。例えば、ファイルを open するというのはそのファイルに何らかの変更が(2 つのツリーの間で)施されていることだけを示しており、その詳細については述べていません。`apply_textdelta`を呼べば変更の詳細を示すことができますが、その必要がないなら`apply_textdelta`を呼ばなくてもいいのです。同様に、ディレクトリをopenするということはそのディレクトリやそれ以下の何かが変わったことを示しますが、それ以上詳しく示す必要がないのなら、すぐディレクトリを close して先に進んでもいいのです。

これらの境界は、インタフェースの**ストリーム指向**の結果であり、そのことは冒頭のコメントにも次のように語られています。「そこで、ツリーの差分を明示的なデータ構造として表現する代わりに我々は、差分を消費する(受け取る)側が、生成する側から送られて来たらすぐに処理してしまう標準的な方法を定義する。」大きいデータのかたまり(つまりファイルの差分)だけをストリームで扱うというのが魅力的な選択だったかもしれませんが、差分エディタのインタフェースではそれにとどまらず、ツリーの差分全体をストリーム化し、その結果として生成側も消費側もメモリ使用量、進捗報告、中断などにおける詳細な制御を可能にしたのです。

実は上記のような利点が分かったのは、差分エディタをさまざまな問題に適用し始めた後だったりします。例えば、私たちが実装したかったことの1つとして、変更のサマリー、つまり2つのツリーどうしの、詳細には立ち入らないおおまかな違いの表示を作ることがありました。そのようなものは、ユーザが自分の作業コピーのうちどれとどれをチェックアウト後に修正したか知りたいが、具体的にどこを変えたかまでは知らなくていい、という場合に役立ちます。

そのような機能の実装を単純に言えば、次のようになります。クライアントはサーバに現在の作業用コピーがどのリビジョンツリーに基づくものかを通知し、次にサーバがクライアントに、差分エディタを使って、リビジョンツリーと作業用コピーの違いを通知して来ます。つまり、サーバが生成側、クライアントが消費側になります。

この章の始めの方で図示した例では、`/A/fish/tuna` に修正を加えてリビジョン 2 を作成しました。これを使って、クライアント側のリビジョンがまだ1である場合に、一連の差分エディタの呼び出しがどのようになるかを見てみましょう。コードの2/3くらいの位置にあるifブロックのところが、サマリーの編集なのか全部の詳細を受け取る編集なのかの枝分かれになっています。

```
svn_delta_editor_t *editor
void *edit_baton;

/* もちろん実際には、この変数はパラメタとして渡される方がよい */
int summarize_only = TRUE;

/* 実際には、これらの変数はサブルーチン内で宣言されていて、その寿命はスタックフレームと一致し、それらが
   指すオブジェクトの寿命やツリーの編集フレームの寿命とも一致する。 */
void *root_baton;
void *dir_baton;
void *subdir_baton;
void *file_baton;

/* 同様に、本来はトップレベルのプールだけではなくサブプールを使う。 */
apr_pool_t *pool = svn_pool_create();

/* 差分エディタインタフェースを使う時にはまず、自分が使いたい機能を実装したエディタ要求する。例えば、編集をネットワーク経
   由で順次受け取るとか、その編集内容を作業用コピーに適用するとかいった具合に。*/
Get_Update_Editor(&editor, &eb,
                  some_repository,
                  1, /* ソースのリビジョン番号 */
                  2, /* ターゲットのリビジョン番号 */
                  pool);

/* 続いて、エディタの各関数を呼び出す。実際には、これらの呼び出し列は直接書かれるのではなく、2つのリポジトリのツリーを両
   側でたどりながら適切なeditor->foo()を呼び出すという形で動的に生成される。 */

editor->open_root(edit_baton, pool, &root_baton);
editor->open_directory("A", root_baton, pool, &dir_baton);
editor->open_directory("A/fish", dir_baton, pool, &subdir_baton);
editor->open_file("A/fish/tuna", subdir_baton, pool, &file_baton);

if (! summarize_only)
  {
    svn_txdelta_window_handler_t window_handler;
    void *window_handler_baton;
    svn_txdelta_window_t *window;

    editor->apply_textdelta(file_baton, pool
                            apr_pool_t *pool,
                            &window_handler,
                            &window_handler_baton);

    do {
      window = Get_Next_TextDelta_Window(...);
```

```
      window_handler(window, window_handler_baton);
    } while (window);
  }

editor->close_file(file_baton, pool);
editor->close_directory(subdir_baton, pool);
editor->close_directory(dir_baton, pool);
editor->close_directory(root_baton, pool);
editor->close_edit(edit_baton, pool);
```

この例から分かるように、変更のサマリーを作る場合と変更をすべて適用する場合の違いは、差分エディタのインタフェースが提供する境界に自然な形であてはまるため、両方の場合についてコードの大部分が共有できます。この例の場合は2つのリビジョンツリーは隣接したバージョン(リビジョン1と2)でしたが、隣接していなくても全く構いません。同じコードが任意の２つのツリー対して使えますから、例えば長い間作業用コピーを更新していなくて、間に多数のリビジョンがはさまった状態であっても大丈夫です。また、2つのツリーの新旧が逆で、作業用コピー側が新しくても構いません(変更してしまったものを元の状態に復元する場合などがこれに相当します)。

監視点としての抽象化

差分エディタの柔軟性を示す次の例は、1つのツリーを編集して2つ以上の異なる変更を行う必要がある場合などについてのものです。最初期からあったそのような状況の１つとして、キャンセルの扱いがありました。ユーザが更新を中断した場合、シグナルハンドラが要求に割り込んでフラグを立てます。続いて操作のあちこちの箇所でそのフラグを調べ、もしフラグが立っていたら適切な中止処理を行います。実際にやってみると、たいていの場合、中断するのに最も安全な場所は単純に次のエディタ関数の入口か出口の時点でした。これはクライアント側でI/Oを行わない操作(変更のサマリーを作ったり差分を取ったりするなど)では明らかにそうだと言えますが、ファイルを触るような多くの操作についても同様でした。結局、更新時の作業の大部分は単にデータを書き出すことであり、ユーザが更新を中断して割り込みが検出された時に何のファイルが処理中であったとしても、現在書き出し中のものを全部書き終わるか、または綺麗さっぱり書かなかったことにするのが最善でした。

しかし、フラグのチェックはどこに入れたらいいでしょうか？ Get_Update_Editor()で参照が返される差分エディタの中に直接そういうコードを含めることもできます。しかしそれは明らかにいまいちな選択です。というのは、差分エディタはライブラリ関数としてさまざまなコードから呼ばれ、呼び出し側のコードによっては全く違う方式のキャンセルチェックを採用していたり、全くチェックしなかったりするかもしれないからです。

もう少しだけましな解としては、キャンセルチェック用関数とそれに対応するバトンをGet_Update_Editor()呼び出し時に引数として渡すことかもしれません。これによって作成される差分エディタは、そのバトンを引数としてチェック関数を定期的に呼び出し、その返値によって動作を続けるか、途中で中止するかを選択します(キャンセルチェック関数としてnullを渡した場合は呼び出しも行わないものとします)。しかし、このような設計も理想的とは言えません。キャンセルをチェックするということ自体が、副次的な目的です。あ

なたは更新の途中でチェックをしたいと思ったりしたくないと思ったりするでしょうが、いずれにせよその動作は更新自体の過程には影響しないのです。理想的に言えば、この2つのことがらはコード中で一緒くたになるべきでないのです。そういうわけで結局、私たちの結論は、差分エディタの操作においてキャンセルのチェックに細かい制御が必要となることはほぼ皆無であり、エディタの関数の出入口でチェックするだけで十分というものでした。

キャンセルはツリーの差分エディタに関連する付帯作業の1つの例に過ぎません。私たちは、更新やコミットの進捗を表示したり、それと類似の場面において、クライアントからサーバに変更を伝達中にコミット対象を追跡管理するという、キャンセルの場合に類似した問題に直面しました(と、思いました)。当然ながら、私たちはこれらの付帯的な動作を抽象化して、コアの作業がごちゃごちゃしないようにする方法を探求しました。実際のところ、私たちは探求しすぎて、過度に抽象化してしまいました。

```
/** editor_1 および対応するバトンを editor_2 および対応するバトンと統合する。
 *
 * new_editorに新しいエディタ(プール中に割り付けられる)を返す。その各関数呼び出しはeditor_1->funとeditor_2->fun
 * をこの順で、対応するバトンを指定して呼び出す。
 *
 * もし editor_1->fun がエラーを返したらそのエラーは new_editor->fun から返され、editor_2->fun は呼ばれない。
 * そうでない場合、new_editor->fun の返値は editor_2->fun の返値と同一である。

 * もしエディタ関数が null であれば、単にそれの呼び出しは行われない(これはエラーではない)。
 */
void
svn_delta_compose_editors(const svn_delta_editor_t **new_editor,
                          void **new_edit_baton,
                          const svn_delta_editor_t *editor_1,
                          void *edit_baton_1,
                          const svn_delta_editor_t *editor_2,
                          void *edit_baton_2,
                          apr_pool_t *pool);
```

これはやりすぎだったと分かったわけですが(なぜかはすぐ後で)、私は今でもこれはエディタインタフェースの美しさの証拠だと思っています。合成されたエディタは予想される通りの振る舞いを持ち、コードはきれいであり(というのは、それぞれのエディタ関数はどこかで自分と並行して呼び出されている関数があるなどと心配する必要がありませんから)、結合則のテストをパスします。つまり、合成した結果のエディタをさらに他のエディタと合成してもちゃんと動きます。なぜ動くかというと、すべてのエディタはそれぞれが実行する操作の形について互いに整合性を持っていて、それぞれがデータに対して全く違う操作を行うとしてもこの点は変わらないからです。

お分かり頂けると思いますが、私はこのエディタ合成機能について、エレガンスさゆえに、あっても良かったと今でも思っています。しかし結局のところ、それは私たちが必要とする以上の抽象化でした。私たちが当初エディタの合成を用いて実装した機能の大部分は、通常のエディタ生成関数に専用のコールバックを渡す形で書き直されました。エディタ関数の呼び出し時に付帯的な動作が必要になることは確かにあるのです

が、**すべての**呼び出し時にというわけではなかったのです(多くの、でさえありませんでした)。そのため、エディタ合成機能を使った場合、作業自体に対する骨組み設定機能の比重が高くなりすぎました。並行エディタを設定する一連のコードを見せることで、私たちはユーザに、そのコードにおける付帯的作業の呼び出し頻度を実際よりもずっと高いものと勘違いさせてしまいがちだったのです。

エディタ合成機能について行き着くところまで行ってから退却しましたが、それでも私たちは自分達が欲しかったものを手で実装することができます。今日のSubversionでは、キャンセル動作は手で組んだエディタ合成で実装されています。キャンセルチェックつきエディタのコンストラクタは、他のエディタ(本来の操作を行うエディタ)をパラメタとして受け取るようになっています。

```
/** *editorと*edit_batonにwrapped_editorとwrapped_batonを包含したキャンセル機能つきエディタとそのバトンを
 * 設定する。
 *
 * キャンセル機能つきエディタでは、その関数のどれかがキャンセルされようとしたとき、cancel_batonをパラメタとして
 * cancel_funcが呼び出される。この関数がSVN_NO_ERRORを返した場合には、キャンセルは行われず対応する関数の呼び出しが
 * 行われる。
 *
 * もしcancel_funcがNULLであれば、単に*editorには*wrapped_editor、*edit_batonには*wrapped_batonが
 * 設定される。
 */
svn_error_t *
svn_delta_get_cancellation_editor(svn_cancel_func_t cancel_func,
                                  void *cancel_baton,
                                  const svn_delta_editor_t *wrapped_editor,
                                  void *wrapped_baton,
                                  const svn_delta_editor_t **editor,
                                  void **edit_baton,
                                  apr_pool_t *pool);
```

私たちは同様に手で合成する方法で、条件つきデバッグトレース機能を実装しました。他の付帯的な動作である、本体動作の進捗報告、イベント通知、ターゲットの計数などは、エディタのコンストラクタにコールバック関数を渡して登録し、それらがnullでなければ決められた適切な箇所でエディタがそれらを呼び出すという形で実装しています。

差分エディタのインタフェースはSubversionのコードに対して強い統合力を発揮し続けています。ユーザを抽象化しすぎに誘惑したAPIを称賛するというのも変に思えるかもしれませんが、その誘惑もAPIがストリーム指向でツリーの差分を伝達するという問題をあまりにもうまく克服したことの副作用と言えるでしょう。つまり、元の問題をあまりにもうまく手なずけたので、別種の問題まで元の問題と同種の問題にしたくなってしまった、というわけです！ その別種の問題については退却しましたが、それでも差分エディタのコンストラクタはそのような問題向けのコールバックを設定する適切な場所ですし、エディタ内部の動作の境界は私たちがいつコールバックを呼ぶべきか考える上での助けとなっています。

結論

このAPIの真の威力は、ひいては、すべての優れたAPIの真の威力は、それが人の考え方を導いてくれるということだと思います。Subversionにおいてツリーを変更するすべての操作は、今やほぼ共通のやり方に統一されています。これによって、新人が既存のコードを学ぶために費す時間が節約されるのみならず、新しいコードを書く時にも手本とすべき明快なモデルが与えられることとなり、開発者がヒントを得ることができます。例えば、1つのリポジトリへの変更を直接他のリポジトリに反映するsvnsync機能はSubversionに2006年（差分エディタが出現してから6年も経っています）に追加されたものですが、これも差分エディタのインタフェースを使って変更を伝達しています。この機能の開発者は変更を伝達する機構を設計する必要がなかっただけでなく、変更を伝達する機構を設計する必要があるかどうか**考慮する**必要すらなかったのです。そして、このコードをハックしている他の人たちも、そのコードを最初に見た瞬間から、おおむね馴染みのあるコードだと感じているのです。

これらの事柄は極めて大きな利点です。正しいAPIを持つことで、学ぶ時間を節約できるだけでなく、開発コミュニティが設計に関する議論を始めてしまい、メーリングリストが延々と物議をかもすスレッドで占拠されてしまうこともなくなるのです。これは純粋に技術的な美しさとか倫理的な美しさとは言えないかもしれませんが、多くの参加者を持ち絶えず展開していくプロジェクトにおいては、極めて役に立つ美しさなのです。