3. 構成管理とサンドボックス

|  |
| --- |
| 「この機械、こわれる前はこわれてなかったんだろ。」 「あたりまえだ。」 「タイムふろしきで……。」 (ドラえもん のび太の宇宙開拓史より) |

# ソフトウェア構成管理とは

ソフトウェア構成管理とは、SCM (Software Configuration Management) の訳語です[[1]](#footnote-1)。構成管理とは製品やシステムを構成する部品や要素の組み合わせの一貫性を管理することです。ハードウェアやネットワークインフラなどにおいても構成管理の話題があります。ソフトウェアにおける構成管理とは、狭義にはファイルのバージョン管理のこと、またそれを支援するツールのことです[[2]](#footnote-2)。SCMツールのリポジトリ (保管場所) に格納されたファイルは、それぞれに番号が付けられ、その履歴(いつ、だれが、なぜ、どんな変更をしたか)が管理されます。この機能により、リポジトリに保存されている任意のファイルを任意の(過去の)時点に戻すことができます。SCM ツールの歴史は古く、世界最初のSCMツールであるSCCSが登場したのは 1972 年にさかのぼります。それ以後、さまざまなSCMツールが開発され、使われています。無償で利用できるオープンなSCMには、クライアント・サーバ型ではCVS とその後継という位置づけのSubversion[[3]](#footnote-3)が有名です。有償のSCMにはPerforceやMicrosoft Visual SourceSafe, Microsoft Team Foundation Server, Rational ClearCaseなどがあります。このほか、Git[[4]](#footnote-4)やMercurialなどのような分散型のSCMもあります。本章では、これらのSCMツールに共通する概念や、操作の手順、有用なプラクティスなどを示します。本書で説明するのはクライアント・サーバ型のSCMですが、ここに示した概念や操作手順の多くは分散型のSCMにも適用できます。

# リポジトリ

SCMリポジトリは、ファイルサーバのように、その中にファイルを格納して管理するためのものです。しかし、通常のファイルシステムとは違って、SCMリポジトリの中には各ファイルの変更履歴がすべて保存されています。図3 - 1を見てください。



図3 - 1 リポジトリとサンドボックス[[5]](#footnote-5)

SCMはクライアント・サーバ型のソフトウェアです。サーバとはサービスをする人のことで、奉仕者とか給仕という意味です。クライアントはそのサービスを受けるお客さんです[[6]](#footnote-6)。この図では、リポジトリがSCMサーバを表し、サンドボックスとビルドマシンがSCMクライアントを表します。サンドボックスとは開発者用の開発環境 (開発用PC) のことです。また、ビルドマシンとはビルド作業専用のPCのことです。リポジトリの中にソフトウェア開発に必要なファイル (プログラムファイルなど) を入れて、チームメンバーで共有します。

次に、図3 - 2を見てください。ある瞬間のソースファイル一式のことを、スナップショットといいます。この語には、文字通りリポジトリのある瞬間の状態を写真に撮って残すようなニュアンスがあります。リポジトリの中には、スナップショットの履歴がすべて保存されています。このため、通常のファイルシステム内では、絶対パス (c:\directory\_path\file\_name.txtのようなもの) だけでファイルを一意に指定できますが、リポジトリ内のファイルを一意に指定するには絶対パスに加えてリビジョン番号が必要です[[7]](#footnote-7)。



図3 - 2 サンドボックスを  
最新のスナップショットで更新して作業する

リポジトリからは、任意の時点のスナップショットをサンドボックス側に取り出せます。この操作を指して、更新といいます。更新時に取り出したいスナップショットの時点を指定しなければ、最新(枝の先端)のスナップショットが取り出されます。バグの調査などの特別な事情がなければ、チームメンバーが更新の操作をするときは必ず最新のスナップショットを取り出すことになります。

サンドボックスで、意味のある単位 (新機能の追加やバグの修正など) で複数のファイルを修正できたら、それをまとめてリポジトリ側に入れます。この単位をチェンジセット　(差分の集まり)　といいます。また、チェンジセットをリポジトリに入れる操作をコミットといいます。コミットすると、チェンジセットにより追加・変更されたファイルのリビジョン番号がひとつ増え、新しい履歴が保存されます。この操作により、リポジトリ内の枝が伸びていきます。

つまり、複数の開発者はそれぞれのサンドボックス内でコードを修正し、その修正を並行して同じブランチにコミットします。ビルドマシンでは、伸びた枝の先端から定期的にコードを取得 (更新の操作) をしてビルドします。本章では、この枝の上手な伸ばし方や切り分け方を説明しようというわけです。

# サンドボックス

各開発者は、リポジトリ中に管理されているソースコードなどのファイルを取得して、各自のPCに開発環境を構築します。この、SCMクライアントソフトウェアを使って構築した開発環境のことをサンドボックス (お砂場) といいます[[8]](#footnote-8)。開発者は、自分のサンドボックス内で開発作業 (プログラムを書いたり、デバッグしたり) をします。清潔なソースコードを入れておくリポジトリは、汚いものが入り込まないように丁寧に扱わねばなりません。これに対し、サンドボックスではお城を作ってみても良いし、ジョウロで水をかけて川を作っても良いし、なにを試しても良いということが強調されているのです。その代わり、何をしてもいいのはお砂場の中だけであって、その外で遊んではいけません。開発者は、リポジトリを壊すことなく、サンドボックスの中でどろんこになって自由に遊ぶことができます。ここで試行錯誤した結果、上手に新機能を追加したりバグを取り除いたりできたら、その修正に必要となるきれいなファイル(差分)だけをサンドボックスから取り出してリポジトリにコミットします。

# リポジトリにファイルをコミットする

サンドボックスで修正したファイルをリポジトリにコミットする手順を示します。複数人で枝を伸ばしていくイメージをつかんでください。

## 最新のファイルを修正してコミットする例

1. 最新のファイルで 　　　　 ② 更新したファイルを  
   サンドボックスを更新 サンドボックス側で修正

 

1. コミットすると、このファイルの  
   次のリビジョンが作成される



このように、SCMのブランチ (branch; 枝)[[9]](#footnote-9) はその先端にチェンジセットが積み重なるようにして伸びていきます。コミットを過去の時点に挿入することはできないことに注意しましょう。

上図で示したチェンジセットにはひとつのファイルの差分しか含まれていませんが、複数のファイルに対する (差分) をひとつのチェンジセットに含めることもできます。意味のある単位(あるバグの修正のため、ある機能追加のためなど)でチェンジセットを構成することが重要です。特に、そのチェンジセットのコミットにより、ビルドブレークを起こさないように注意します[[10]](#footnote-10)。

## 同じファイルを複数人で同時に修正する例

1. あるファイルを更新して ② ほかの人が、同じファイルを  
   サンドボックスで修正中 修正して、先にコミットした[[11]](#footnote-11)

 

1. ここで再度更新の操作をすると ④ マージされたファイルを  
   修正が自動でマージされる[[12]](#footnote-12) コミットする

 

このようなやり方を、コピー・修正・マージ (copy-modify-merge) 方式といいます。テキストファイル (プログラムファイルなど) を複数人で同時に編集できるので、大変便利です。

このほか、ある人が修正中のファイルは、ほかの人による修正・コミットを許さないやり方もあります。これを、ロック・修正・アンロック (lock-modify-unlock) 方式といいます。これは不便で使いにくいのですが、バイナリファイル (Microsoft Excelの.xlsファイルのようなアプリケーションデータファイル) をリポジトリで管理するときはこの方式を使います。バイナリファイルはマージすると壊れてしまうため、コピー・修正・マージ方式が使えないからです。

# チェンジセット番号

ファイルをコミットすると、優れたSCMツールは、ファイルごとに別の連番をリビジョン番号として振る代わりに、チェンジセットに含まれるチェンジ(差分)の全てに共通の番号を振ります。これをチェンジセット番号といいます。チェンジセット番号は、そのチェンジセットに含まれるファイルセット、もしくはのセットを表します[[13]](#footnote-13)。

また、チェンジセット番号はブランチのある時点も表します。つまり、チェンジセット番号で、そのチェンジセットに含まれない任意のファイルのリビジョンを指定できます。若いチェンジセット番号は古い時点を、大きいチェンジセット番号は新しい時点を意味します。これはチェンジセットが積み重なってブランチが伸びる様子を想起させます。



図3 - 3 チェンジセット番号の意味

SCMツールによっては、チェンジセット番号がこのような通しの番号にならないものもあります。また、これと同じものを指してコミット番号とかコミット名とよぶSCMツールもあります。詳細は、お使いのSCMツールのドキュメントを参照してください。

# チェンジセットを適切に構成する

ひとつのチェンジセットは、意味のある単位で構成することが大切です。

例えば、ある(ひとつの)バグ修正のための変更が複数のファイルにわたるとき、これを複数のチェンジセット(複数のコミット)に分けてリポジトリに入れるべきではありません。これは一時的にもリポジトリの一貫性を壊し、ビルドブレークを引き起こしやすくなるからです。

逆に、ある(ひとつの)チェンジセットに複数のバグ修正や機能追加のための修正を混ぜるべきではありません。その中のどれかが不正だったことが後で判明したとき、意味のある単位でリポジトリを元に戻すのが難しくなるからです。

このように、あるひとつ(だけ)のタスクに関連する複数のファイルをひとつのチェンジセットに含めて、まとめてコミットすることをTask Level Commitといいます。

# コラム アトミックなコミットとは

|  |
| --- |
| **コラム アトミックなコミットとは**  多くのSCMが、チェンジセットをアトミックにコミットできます。アトミックとは、分割できないことを指します。atomは、日本語では原子という意味ですね。その昔、ギリシャの哲学者は、それ以上分割できないとした元素をアトムとよびました。現在では、原子は原子核と電子からなり、さらに原子核は陽子と中性子から、陽子と中性子はクォークからできていることが明らかになり、アトムは「分割できない」という本来の語源とは異なるものになりました。しかし現在でも、哲学的な文脈では、分割できないとしたものをアトムとよぶことがあります。  つまり、アトミックなコミットとは、チェンジセットが最小のコミットの単位であり、これが分割されることはないという意味です。例えば、チェンジセットに5つのファイルの修正を入れてコミットするとき、アトミックなコミットをサポートしないSCMでは、何らかの (例えばネットワークなどの) エラーにより、5つのうちの3つのファイルだけがコミットされ、あとの2つはコミットできなかったということが起こりえます。もちろん、こうなるとリポジトリ内のコードの一貫性は壊れ、ビルドブレークを引き起こします。しかし、アトミックなコミットをサポートするSCMでは、チェンジセットに含まれるすべてのファイルのコミットに成功するか、もしくはすべてのファイルのコミットに失敗するか、必ずそのどちらかになります。データベースシステムなどでも、データの整合性を維持するために、アトミックな操作のサポートが非常に重要です。  CVSは、アトミックなコミットをサポートしません。しかし、Subversionなど、新しいSCMのほとんどがアトミックなコミットをサポートしています。アトミックなコミットは、Task Level Commitのプラクティスを守ることにも役立ちます。 |

# 清潔なサンドボックス

いくらサンドボックスは汚くして良いといっても、ぐちゃぐちゃにして遊ぶのが好ましいわけではありません。それなりに清潔にしておかないと、開発作業に支障が出ます[[14]](#footnote-14)。リポジトリが衛生的なものだとすれば、サンドボックス内の汚れとはリポジトリと違う部分 (差分) であると定義できます。なるべくサンドボックスを清潔に保つべきというポリシーから、次のプラクティス[[15]](#footnote-15)を導きます。

## 頻繁にサンドボックスを更新すべし

ほかの人もリポジトリにファイルをコミットしますから、自分のサンドボックスはどんどん古くなり、放っておいてもリポジトリとの差分が大きくなっていきます。このため、なるべく頻繁にリポジトリから新しいファイルを取得し直す (サンドボックスを更新する) 必要があります。自分のサンドボックスを更新したら、必ずそこでビルドを通しましょう[[16]](#footnote-16)。

## 頻繁にリポジトリにコミットすべし

たとえ、リポジトリにコミットするつもりの変更であっても、それをコミットするまでは、その差分(サンドボックス上で修正したファイル) はただの汚れでしかありません。サンドボックスに汚れがたまると、その中のどれがコミットの必要があるものか、訳がわからなくなってしまいます。なるべくこまめにコミットし、サンドボックスをリポジトリと同期した状態に保ちましょう。

## 頻繁に不要となった修正を破棄すべし

いろいろな修正を試したけどうまくいかなかったファイルは、早めにサンドボックスから除去してください。サンドボックスで編集したコミット前のファイルは、実際に汚いファイル (dirty file) とよばれます[[17]](#footnote-17)。そんな汚れたファイルをサンドボックスに残したままにしておくと、それを誤ってリポジトリにコミットしてしまわないとも限りませんし、汚れたファイルが残っていたためにほかの (サンドボックスで修正した) ファイルが正しく動くように見えることもあります。もちろん、そのような汚いファイルをリポジトリにコミットしてしまうと、ビルドブレークを引き起こし、周りの人に大きな迷惑をかけることになります。汚れたファイルは、単純にサンドボックスから削除し、SCMの更新の操作をすれば破棄できます。あるいは、SCMのリバートという操作を使っても破棄できます。

# サンドボックスの構築手順の確立と文書化

サンドボックス環境の構築は、なかなか大変な作業です。多くの場合、これは必要なツール (コンパイラなど) をインストールし、リポジトリから最新のソースファイルを取得し、その上でビルドを通すといった手順になります。この手順は非常に複雑になりやすいので、チームメンバーのために、必ずサンドボックス構築手順書を用意しましょう。再現可能なビルドを確立するためには、まず再現可能なサンドボックスが必要なのです。ある開発者のPCが壊れたり、新しい開発者がチームに参加したりするなどのタイミングで、新しいサンドボックスを構築する必要が生じます。もし、サンドボックス構築手順書が用意されていなければ、誰も新しいサンドボックスを構築できない、という恐ろしい事態になりかねません。この手順書には、必要なツールやライブラリと、そのバージョンもきちんと記述しておきましょう。よくあるビルドエラーの解決方法をFAQとしてWiki[[18]](#footnote-18)にまとめておくのも良い方法です。

**サンドボックス構築手順書**

1. 以下から Subversion をインストールする。

http://xxx

2. 以下から、JDK x.x をインストールする。

http://xxx

3. 以下から、Eclipse x.x をインストールする。

http://xxx

4. 以下から、Trac をインストールする。

http://xxx

5. 下記のリポジトリから、ソースコードを取得する。

xxx

6. 以上で完了！ビルドを通してみて下さい。

**ビルド手順書**

先に、サンドボックス構築手順書に従ってサンドボックス環境を構築して下さい。

1. cmd.exe を起動して、ビルド用のウィンドウを開く

2. サンドボックスのルートディレクトリに移動

3. ant を実行

…

以上で完了！xxxディレクトリにproduct.msiができるはずです。これを実行すると、インストーラが起動します。

xxxディレクトリにビルドログファイルが作成されます。ビルドに失敗したときはこれを参照して、ビルドエラーを取り除いてください。

**Build Break FAQ**

Q. 次のようなエラーが出る: xxx

A. xxx がインストールされていないか、インストールしたバージョンが間違っています。正しいバージョンは xxx なので、いちどアンインストールして正しいバージョンをインストールし直して下さい。

Q. テスト xxx が失敗する

A. データベース接続設定のためのプロパティファイルは、各サンドボックスごとに修正する必要があります。以下を参考にして下さい。

...

Q. ファイル xxx が見つからない

A. …

# 複数のサンドボックスを切り替えて作業する

ここまでの説明で、開発者は自分のPCにサンドボックスをひとつだけ構築して作業をするというイメージを、読者はお持ちになったかもしれません。しかし、実際には、開発者は自分のPCに複数のサンドボックスを構築することがふつうです。開発者が複数のプロジェクトを兼務したり、ひとつのプロジェクトで複数のブランチが切られたりした場合は、PCの中に複数のサンドボックスを構築する必要が生じるからです。一般に、ひとつのサンドボックスはひとつのブランチに対応します[[19]](#footnote-19)。開発者は、作業対象としたいサンドボックスを適宜切り替えて作業します。また、同じブランチに対して複数のサンドボックスを作っておくこともあります。これについては3-14節で取り上げます。



図3 - 4 ひとつのPC内に、複数のサンドボックスを構築しておく  
各サンドボックスは、それぞれ別のブランチと対応する

# コラム ソフトウェア開発のメタファ③ 音楽

|  |
| --- |
| **コラム ソフトウェア開発のメタファ③ 音楽**  音楽とソフトウェア開発の関係や類似性については、トム・デマルコ先生の著書「ピープルウェア」や「アドレナリンジャンキー」、チャド・ファウラー氏の著書「情熱プログラマー」、ヴェンカット氏とアンディ氏の共著「アジャイルプラクティス[[20]](#footnote-20)」など、複数の書籍で述べられています。中でも、荒井玲子氏の著書「ソフトウェア開発で伸びる人、伸びない人 第二版」では、3つもの章を費やして音楽とソフトウェアの関係が語られています。正確な技術が必要とされること、構造が美しくあるべきこと、国際性があることなどの共通点や、作り出す過程で解釈力・集中力が要求されること、さまざまな役割をもつ人が協調して働くべきことなどが述べられており、大変興味深いです。  私も、初めて海外で仕事をしたときに同じような感覚を味わいました。当時、私はかなり英語が下手でした (今もあまり変わりませんが)。しかし、先方のエンジニアに「いいからコード見せてみろ」と (たぶん) 言われてコードを見せたところ、スムーズな意思疎通ができたようなのです。「おお！自分てば、で会話するミュージッシャンみたいだぜ！」と勘違いしたことでした。実際には、相手がわかってくれただけだったのですが。。プログラミング言語も、日本語や英語のような自然言語と同じく、ことばの一種なのですね。 |

# コラム プログラムファイルは企業の資産

|  |
| --- |
| **コラム プログラムファイルは企業の資産**  SCMリポジトリで管理するファイルは、ソフトウェア開発における重要な成果物です。リポジトリはこれを保護する金庫の役割を果たします。プログラムが大きな価値をもつことをよく理解しているソフトウェア企業は、火事や地震などの災害から守るために、プログラムファイルを実際に本物の金庫の中にしまって管理することもあったようです。その企業の重要な製品のプログラムファイルを失うことがあれば、企業は倒産してしまいかねないからです。現在は、ストレージの値段や保守費用が安くなったこともあり、複数のオフィスに分散してプログラムファイルをバックアップしておくことが一般に行われるため、本物の金庫の中にプログラムをしまう必要性は少なくなりました。  私は、デジカメで撮りためた娘の写真をDVDに焼いて、定期的に自分の親にプレゼントしています。もちろん、親を喜ばせるためもありますが、大事な娘の写真を分散バックアップして、自宅のハードディスクがクラッシュするという大惨事からデータを守るため、というのが真の目的です (親には内緒ですが)。  プログラムファイルも、分散型のSCMツールを使えば、より安く簡単に分散バックアップできます。ただし、大事な資産が社外に漏れないように、セキュリティには十分注意する必要があります。 |

# ビルドマシン

サンドボックスとは別に、ビルド専用のマシンを1台準備します[[21]](#footnote-21)。サンドボックスは汚いですから、きちんとしたビルド (テストチームにリリースし、テスト対象とする統合ビルド) は清潔な専用のマシンでビルドする必要があるからです。このマシンをビルドマシン、もしくはインテグレーションビルドマシンといいます。ビルドマシンにもSCMクライアントソフトウェアをセットアップしてリポジトリと接続し、ビルドのたびに最新のソースファイルを取得できるようにします。規模がそれほど大きくないソフトウェアを開発するときは、SCMサーバと同じPCをビルドマシンとしても使うことがあります。

# リポジトリで管理すべきファイル

リポジトリには、履歴を管理したいファイルであれば何を入れても構いません。プログラムのソースコード以外にも、Microsoft Wordで書いた設計書をリポジトリに入れて管理しても良いでしょう　(あるいは、そうすべきという人もいます)。しかし、実行可能なソフトウェアを上手に構築するためには、SCMで管理すべきでないファイルもあります。本節では、これについてガイドラインを示します。

図3 - 5を見てください。これは、実行可能なソフトウェア (ビルド) を構築するために必要なファイルとその手順の概要を示したものです[[22]](#footnote-22)。



図3 - 5 ビルド時のファイルの生成順と依存関係

ソフトウェアを開発するときは、さまざまなプログラミング言語でプログラムファイルを記述したり、各種設定ファイルを準備したりすることになります。これらのファイルは人間にも読みやすい形で記述されますが、そのままではコンピュータが解釈できないので、コンパイラなどのツールを使ってコンピュータが直接解釈できる形式に変換します。プログラミング言語によっては、さらにリンクなどの手順を経て、最終的に実行可能なソフトウェア (ビルド) が出てきます。本節の要点は、リポジトリに入れて管理すべきファイルはビルドのインプットとする一次ファイル群だけにすべきことです。この一次ファイルのことをソースファイルといいます。ビルドの途中で出てくる中間ファイルや最終的なビルドはリポジトリ内に入れるべきではありません。一次ファイルの一式さえあれば、後日全く同じビルドを再現できるからです。逆にいえば、後日にビルドを再現するために必要なファイルだけをリポジトリで管理すべきということです。もちろん、この中にはビルドスクリプトなどのビルド環境を構築するためのファイルも含まれます。また、サードパーティ製ライブラリなども、それらがビルドに必要な一次ファイルであればリポジトリに入れて管理します。

ビルドは頻繁に作ることになるため、これをすべてリポジトリの中に入れて管理するのはリポジトリのディスク容量を圧迫するので好ましくありません。ビルドは、ファイルサーバに日付フォルダを作って保存する方が便利です。ファイルサーバに保存したファイルは、簡単に削除したりDVDなどの外部メディアに移動したりできますが、リポジトリにコミットしたファイルはそのような管理ができないからです。ただし、マイルストーンビルドやリリースビルドなどの重要なビルドは、リポジトリに入れて管理することもあります。古いビルドを保存しておく価値については、3-15節を参照してください。

表3 - 1 一次ファイルの種類 (例)

|  |  |  |
| --- | --- | --- |
| **拡張子** | **意味** | **コメント** |
| .h, .hpp, .hxx | header file | C/C++ヘッダファイル。複数のC/C++ソースファイルから参照される |
| .c, .cpp, .cxx | C / C plus plus | C/C++ソースファイル |
| .java | Java | Javaソースファイル |
| .jsp | Java server page | Javaサーバページファイル |
| .idl | interface definition language | インターフェイス定義ファイル |
| .rc | resource script | リソーススクリプトファイル |
| .bmp | bitmap image | 画像リソースファイル |
| .htm, .html | hypertext markup language | Htmlファイル |
| .js | Java Script file | Java Scriptソースファイル |
| .properties | properties file | Javaプロパティファイル |
| build.xml | build file | ビルドスクリプト。ビルドツールAntで実行する |
| makefile | make file | メイクファイル。ビルドツールMakeで実行する |

表3 - 2 中間ファイルの種類 (例)

|  |  |  |
| --- | --- | --- |
| **拡張子** | **意味** | **コメント** |
| .obj | object file | C/C++プログラムをコンパイルするとできるバイナリファイル。オブジェクト指向とは関係ない |
| .lib | library file | 複数の.objファイルをひとつにまとめた形式 |
| .pch | precompiled header file | C/C++ヘッダファイルをコンパイルしたもの。C/C++ソースファイルのコンパイル時間を短くする効果がある |
| .map | map file | オブジェクトファイル中のどこに何が配置されているかが記述されたファイル |
| .pdb | program database file | ソースの位置とオブジェクトファイルを関連づけ、デバッグを容易にするためのデータベースファイル |
| .res | compiled resource file | リソーススクリプトをリソースコンパイラでコンパイルしたもの |

表3 - 3 実行可能ファイルの種類 (例)

|  |  |  |
| --- | --- | --- |
| **拡張子** | **意味** | **コメント** |
| .exe | executable | 実行可能ファイル |
| .msi | Microsoft installer | インストーラ。実行するとインストーラが起動する |
| .dll | dynamic link library | ほかの実行可能ファイルから動的に読み込まれて実行される。単体では実行できない |
| .lib | library | 静的にリンクされるライブラリファイル。ライブラリ製品では、このファイルを製品として出荷することがある |
| .class | class | Javaソースファイルをコンパイルしてできるファイル。Java仮想マシン上で実行可能 |
| .jar | Java archive | 複数の.classファイルをアーカイブした (まとめた) .zipファイル。コーヒー豆の品種名であるところのJavaと相性が良いナイスなネーミング (jarとは英語で壺とか瓶という意味。例: 電子ジャー、炊飯ジャーなど) |
| .war | web archive | 中身は.jarと同様だが、webアプリケーションサーバに配置とすると、webアプリケーションとして実行可能な形式 |

# コラム プログラムとは

|  |
| --- |
| **コラム プログラムとは**  プログラムとは、前もって予定された一連の作業手順のことです。運動会のプログラムを想像していただければ良いでしょう。コンピュータの計算処理は非常に高速なので、あらかじめ人間がプログラムを作ってコンピュータに渡しておかないと、コンピュータの能力を十分に活かせません。プログラムは、あいまいさが入り込まないように厳密に文法が定められた言語 (プログラミング言語) でテキストファイルに記述されます。このプログラムファイルをソースファイル、ソースコードなどといいます。ソース (source) とはおおもととか源という意味です。ケチャップのたぐいではありません (それは sauce です)。文字通り、実行可能なソフトウェアの構築に必要な、源 (もと) のファイル、一次のファイルということです。また、コードには文字 (化) とか符号 (化) という意味があります。つまり、コーディングとは、仕様をプログラムという形に符号化して、コンピュータに解釈・実行できるようにするということです。このほか、コードには暗号 (化) という意味もあります。ダビンチ・コードという映画もありましたね。しかし、暗号みたいなものでなく、読みやすいプログラムコードの記述を心がけたいものです。  コンパイルを必要とせず、テキスト形式のソースファイルをそのまま実行できるプログラミング言語もあります。これは、そのプログラムファイルを実行時に動的に翻訳・解釈して実行するソフトウェアがあるからです。このソフトウェアをインタプリタといい、そのような言語をコンパイラ言語に対してインタプリタ言語とかスクリプト言語などといいます。インタプリタ言語で文法的に間違ったプログラムを書くと、実行時に文法エラーとなります。これに対して、コンパイラ言語で文法的に間違ったプログラムを書くと、コンパイル時に文法エラーとなります。プログラムを実行する前に文法エラーを検出できるので、一般にコンパイラ言語はプログラムを書くのは大変ですが、比較的安全で高速なプログラムが書けます。特に、コンパイラ言語の強力な静的型付けの機能は、安全なプログラムの記述を支援するものです。コンパイラ言語にはC++, Java, C#などがあります。これに対し、インタプリタ言語はお手軽にプログラムを書けますが、危なっかしくて比較的遅いプログラムになります。もっとも、最近はハードウェアの進歩もあり、インタプリタ言語の適用範囲が非常に広くなってきています。インタプリタ言語にはRuby, Python, Perl, JavaScriptなどがあります。Rubyは言語仕様も洗練されており、大変優れたインタプリタ言語です。  このほか、手続き型言語や関数型言語など、別の側面でプログラミング言語を分類することもできますが、本コラムでは割愛します。 |

# コミットするときによくある失敗

リポジトリにファイルをコミットする操作は、開発者にとって非常に日常的で、頻繁に行う機会があります。それだけに、失敗することも多いものです。コミットするときにありがちな失敗には、次のようなものがあります。

## 必要なファイルのコミットを忘れてしまう

あるバグを治すのに複数のファイルを修正してコミットする必要があるとき、そのうちのいくつかのコミットを忘れてしまいます。

## 不要なファイルをコミットしてしまう

あるバグを治すのに複数のファイルを修正してコミットする必要があるとき、そのバグ修正には不要な関係ないファイルも一緒にコミットしてしまいます。

## 汚いファイルをコミットしてしまう

自分のサンドボックス環境が古くなっていたため、実はこのサンドボックスでしかビルドが通らないファイルをコミットしてしまいます。あるいは、自分のサンドボックスでもビルドが通らないファイルをコミットしてしまいます。

## 間違ったブランチにコミットしてしまう

ポート作業[[23]](#footnote-23)をしているときに、ブランチ名を間違えて作業してしまうなどです。

# コミットの手順

前節に挙げたようなコミットの失敗を避けるために、コミッタ (ファイルをコミットする人) が従うべきプラクティスを紹介しましょう。

## 単体テストをする

単体テストは、本書の範囲に含まれませんので詳細は説明しませんが、ソースコードをコミットする前に、まず単体テストを動かしてパスすることを確認すべきです。もし、単体テストがまだなければ、先に単体テストを書いてコミットしておかねばなりません。単体テストは、JavaであればJUnit、C/C++であればCppUnitなどの単体テストフレームワークを使って記述します。

## プライベートビルドをする

サンドボックスでビルドすることをプライベートビルドといいます。ソースファイルをコミットする直前に、コミッタは必ずサンドボックス環境を更新してプライベートビルドし、ビルドが通ることを確認します。プライベートビルドは、そのファイルをコミットしても安全である(ビルドブレークを引き起こさない)ことの確認として、最低限必要な作業です。必ず実施しましょう。

## プライベートビルドをスモークテストする

スモークテストとは、ハードウェアのテストに由来します。ハードウェアを組み上げた直後にスイッチを入れてみて、煙がもくもくと出ないかどうか検査することをスモークテストといいます。同じように、ソフトウェアを直後に実行してみて、煙がもくもくと出ないかどうか検査するのがソフトウェアのスモークテストです。重要なシナリオや機能を短時間で検査できるテストケースをスモークテストとして選択しておくと良いでしょう。スモークテストは、自動化しておければ便利です。

## コードレビューを依頼する

プライベートビルドがスモークテストにパスしたら、修正したファイルをコミットする前に同僚の開発者にレビューしてもらいます。この、レビューをしてくれる人のことをレビューア (reviewer) といいます[[24]](#footnote-24)。コミッタは、レビューしてほしいファイル(チェンジセット)をメールに添付して、レビューアにレビューを依頼します。

開発チーム全員にレビューを依頼するのは良くありません。それでは、みなが「誰かがレビューするだろう」と考えるため、誰もレビューしてくれないからです。そこで、あらかじめ機能ごとにレビューアを決めておきます。一般に、その機能を実装するドライバ (フィーチャーオーナー) をレビューアとします[[25]](#footnote-25)。フィーチャーオーナーがコミッタのときにもレビューしてもらえるように、機能ごとに複数人をレビューアに設定しておくのも良いでしょう。例えば、デブリードやブランチオーナーにもレビューを担当してもらいます。レビュー依頼メールをレビューアに送るとき、開発チーム全員にccするのも良い習慣です。

もし、レビューの結果がNGだったら、レビューアは具体的にどこがまずいのか、どのように修正すべきかをアドバイスします。コミッタは、それに従ってコードを修正したら、再度レビューアにレビューを依頼します。このレビューサイクルを、レビューアからOKがもらえるまで繰り返します。

コミットしたいソースファイルに追加したロジックが正しいかどうかはもちろんですが、コーディング規約を順守しているかどうかもレビューの対象とします。利用可能なプログラム整形ツールがあれば、レビューを依頼する前に通しておきましょう。最近のIDEのほとんどが、プログラム整形ツールを同梱しています。

コードレビューを依頼するときのメールの書式の例を表3 - 4にまとめましたので、参考にしてください[[26]](#footnote-26)。レビューを依頼したメールにOKとかLooks good.などと返事をもらえたら、コミッタは修正したファイルをコミットできます。もし、レビューのサイクルに時間がかかったときは、念のためもう一度サンドボックスを更新して、ビルドが通るのを再確認してからコミットしましょう。

表3 - 4 コードレビュー依頼票 (例)

|  |
| --- |
| Summary: バグをひと言でわかりやすく (バグ報告票のタイトルと同じでよい)  Details: バグの詳細や、Root Cause (根本原因) など  Bug#: 対応するバグ報告票の番号  Related Bug#: このバグに関連するバグ報告票の番号  Steps: バグを再現できる手順  Observed: バグが再現したときの振る舞い  Expected: 本来あるべき振る舞い  Modified files: 修正したソースファイル一覧  How to build: 修正したソースファイルをビルドする手順  Modifier: コードを修正した人  Contributor: コードの修正に協力した人  Reviewer: コードの修正をレビューする人  Affected area: 修正の影響が及ぶ範囲  Test impact: テストに対する影響度をLowもしくはHighなどと記入  Regression risk: リグレッションが発生する可能性をLowもしくはHighなどと記入 |

規模が大きく、安定した品質のソフトウェアでは、このようなコミット前のコードレビューを義務付けることがよくあります。ただし、コミットの頻度が多すぎるとレビューし切れなくなってしまうため、柔軟な運用が必要になることもあります。次のような文言をレビュー依頼メールの先頭に書くなどすると良いでしょう。

* この変更は正しいかどうか不安なので、ぜひレビューして下さい。
* この変更は特に問題がないと思いますので、いついつまでに返答がいただけない場合は、問題ないものとしてコミットさせて下さい。
* この変更はすぐにコミットします。ご容赦ください。

また、Review BoardやCodeStriker, Crucible, CodeCollaboratorなど、このようなコミット前のレビューを支援するツールもあります。これらは、コミット前のファイルのほか、コミット済みのファイルをレビュープロセスにのせることもできます。また、ソースコード以外の任意のファイル (Microsoft Officeのファイルや画像ファイルなど) をレビューできるものもあります。

あるいは、メールでコードレビューをするよりも、ペアプログラミングの方が効果的かもしれません[[27]](#footnote-27)。プロジェクトの状況に応じて、適切なコードレビューを使いましょう。

## バディビルドを依頼する

さらに、今回コミットしたいファイル以外のファイルも修正しているなどで、自分のサンドボックスが少し汚れているかもという不安があるときは、コードレビューと一緒にバディビルドもレビューアにお願いします。バディという言葉は、スキューバダイビングをする人には馴染みがあるでしょう。自分の命を守るために、バディ (相棒) と一緒に潜ることが義務づけられることがありますね。バディビルドとは、リポジトリ中のファイルを守るために、バディ (同僚) にもビルドしてもらうことです。バディのサンドボックスでも問題なくビルドでき、そのスモークテストでも問題が出なければ、その変更は安全にコミットできると考えられます。レビューアは、レビューとスモークテストの結果を合わせてコミッタに返します。

また、開発に使っているサンドボックスとは別に、同じブランチのきれいなサンドボックスを自分で持っているなら、そこにコミットしたいチェンジセットを適用して、自分でバディビルドをすることもできます。コードを修正する作業をしたサンドボックスでビルドするのは、バディビルドではないことに注意しましょう。

最近は、バディビルドをしてくれるビルドサーバ[[28]](#footnote-28)もあります。コミットしたいファイルを専用のwebサーバにアップロードするなどしてビルドサーバにバディビルドを依頼すると、ビルドサーバがそのファイルをきれいなサンドボックス(ビルドサーバ上のビルド環境)に適用し、ビルドしてビルドエラーの有無を知らせてくれます。自動化されたテストがあれば実行して、それで問題が見つからなければ、そのままコミットまでしてくれます。このようなビルドサーバの機能をテスト済みのコミット(pre-tested commit)とかゲートチェックイン(gated check-in)などといいます。便利な世の中になったものです。この場合は、ビルドサーバからビルドとコミット完了の通知を受けて、サンドボックスを更新します。自分の修正が正しくコミットされていることを確認して、コミットの手続きが完了します。SCMツールによっては、サンドボックスでリバート(破棄)の操作が必要になる場合もあります。

## コミットコメントを書く

コミッタは、コミットの際に、そのチェンジセットの内容 (なぜそのような修正をしたのか、どのような修正をしたのか) を説明するコメントを書きます。このコメントをコミットコメントといいます。ほとんどのSCMは、コミットの際にコミットコメントの入力を促し、それを当該のチェンジセットと関連づけてリポジトリに記録する機能があります。また、コミットコメントを書かないとコミットできないように設定できるSCMもあります。コミットコメントは、コードの変更を追跡するための重要なドキュメントなのです。

コミットコメントは、「修正しました」のようなものではいけません。そのようなコメントには、何の情報も含まれていません。そこで、コミットコメントには、コードレビュー依頼票と同じものをコピペして記入することを義務付ける組織もあります。それが煩雑で難しければ、その修正に関連するバグ報告票の番号を記入すると良いでしょう。バグ報告票には、そのコミットに関するより詳細な情報(バグの再現手順やどのように修正したかなど)が書いてあるからです。

チェンジセットをコミットして、対応するバグ報告票を「解決済み」にするとき、そのチェンジセット番号も合わせてバグ報告票に記入しておきましょう。これにより、コミットしたチェンジセットと、バグ報告票との間で双方向にリンクをつけておくことができ、構成管理におけるトレーサビリティを確保できます。また、コミット時にバグ番号を指定すると、SCMとバグ追跡システム (BTS) を連携させて、対応するバグ報告票の状態を自動的に解決済みに変更してくれるツールもあります。

以上のようなプラクティスが、不注意なコミットを避けるために大変有効です。開発プロジェクトの段階や製品の規模に応じて、上記のプラクティスのいくつか(もしくは全て)をコミット前に義務づけると良いでしょう。このようなルールをコミットポリシー(commit policy)とかチェックインポリシー(check-in policy)などといいます。表3 - 5に、コミットポリシーの一例を示します。みなさんが開発するソフトウェアのドメインや、チームメンバーの文化や習熟度などに合わせて、適切なコミットポリシーを準備してください。一般に、ソフトウェア開発の段階が進むにつれ、コミットしてよい修正のバーが上がります。いくつかのイベント(最初のビルドが出たとか、コードコンプリートした、ZBBにヒットしたなど) ごとにコミットポリシーを追加していくと良いでしょう。また、このようなコミットポリシーはぎりぎりになってから開発者に示しても受け入れられません。あらかじめ提示し、チーム内で合意を得ておきましょう。

表3 - 5 コミットポリシーの一例

|  |
| --- |
| コミットポリシー |
| コンパイラがエラーを出力するソースをコミットしてはいけない |
| コンパイラが警告を出力するソースをコミットしてはいけない |
| コミットコメントを入れなければコミットしてはいけない |
| コードレビューアがサインオフするまではコミットしてはいけない |
| 対応するタスクもしくはバグがBTSに登録されていなければコミットしてはいけない |
| 対応するタスクもしくはバグの番号がコミットコメントに記入されていなければコミットしてはいけない |
| 対応する単体テストがコミットされていないソースファイルをコミットしてはいけない |
| 対応する単体テストがパスしないソースをコミットしてはいけない |
| バディビルドがビルドできるまでコミットしてはいけない |
| バディビルドがスモークテストにパスしなければコミットしてはいけない |
| バグの修正以外(新機能の追加など)はコミットしてはいけない |
| 優先度1のバグの修正以外はコミットしてはいけない |
| トリアージの結果、許可が得られない限りはコミットしてはいけない |



図3 - 6 コミットの手順

# コラム 適切なコミットの頻度は？

|  |
| --- |
| **コラム 適切なコミットの頻度は？**  3-6節では、サンドボックスを清潔に保つ指針として、コミットは頻繁に行うべきことを紹介しました。SCMツールを使うときのプラクティスとして、多くの人が一日に何度もコミットすることを勧めています。しかし、場合によってはこれが正しくない状況もあるようです。それは、コミットに必要なコスト(時間や手間)がとても大きいときです。特に、製品の規模が大きく、またコミットにビルドサーバによるバディビルドを義務づけた場合です。コミットのたびにビルドが動くので、1回のコミットに半日かかったりしますから、とても1日に何度もという頻度ではコミットできません。また、同一のソースファイルを同時に修正する開発者が少ないときも、頻繁なコミットによるメリットは比較的に少なくなります。  一方で、更新の操作はほとんどの場合で頻繁に行うべきです。もし、みなさんが頻繁な更新をすることに精神的な負担を感じるなら、それは更新の操作によりサンドボックスが壊れてしまう経験を多くしているからでしょう。しかし、それは更新に問題があるのではなく、コミットに問題があります。ひとつのファイルを修正するたびにいい加減なコミットを繰り返す習慣はやめて、リポジトリ内のコードの一貫性を壊さないチェンジセットを構成してからコミットすることを習慣にしましょう。清潔なサンドボックスよりも清潔なリポジトリが、そして頻繁なコミットよりも一貫性のあるコミットと頻繁な更新が大切なのです。 |

# ビルドブレークに対処する手順

前節のようなプラクティスを用いて慎重に作業しても、時には不正なコミットをしてしまうこともあるでしょう。また、作業ミスがなくても、複数人からのコミットが競合してしまうこともあります。そのようなコミットは、リポジトリ内での一貫したプログラムの構造を壊し、ビルドブレークを引き起こします。

ビルドブレークとは、文字通りビルドが壊れるという意味で、コンパイルエラーや自動テストが失敗するなどでビルドが通らなくなってしまうことをいいます[[29]](#footnote-29)。開発者がサンドボックス上で開発作業をしているときは、コンパイルエラーを頻繁に見ることになりますが、これは一般にビルドブレークとはいいません。サンドボックスはビルドを壊して作り直すための場所ですから、それはそれで良いのです[[30]](#footnote-30)。

しかし、ビルドマシン上でビルドブレークしたり、きれいな (リポジトリと同期しているはずの) サンドボックスでビルドブレークしたら、それはリポジトリが汚染されてしまったことを意味します。もし、あなたがそれに気づいたら、早急にこれを除去しなければなりません。リポジトリが清潔であるべきなのは最優先事項なのです。次のような手順になるでしょう。

1. ビルドマシン上での統合ビルドがブレークし、ビルドマスターもしくはビルドシステムが、メールもしくはソフトウェアあんどん[[31]](#footnote-31)などでこれを開発チームに通知します。あるいは、サンドボックスを更新してビルドした開発者がビルドブレークに気づき、開発チームにメールで通知します。このとき、ビルドエラーをメールに添付して、心当たりのある人 (変なコミットをしてビルドを壊した犯人) に出頭を促します。

2. ビルドブレークの知らせを受けた開発チームのメンバーは、この問題が解決されるまで更新の操作を行わないように注意します。もしサンドボックスを更新したら、そのサンドボックスでもビルドブレークするようになり、開発作業ができなくなってしまうからです。

3. リポジトリの履歴から最近にコミットされたファイルを調べ、早急にリポジトリから汚染を除去します。例えば、不正なコミットを取り消したり、コミットし忘れたファイルをコミットしたりして、ビルドが通るようにします。この作業は、犯人が判明すれば、その人が中心となって行うと良いでしょう。

4. 修正できたら、その旨をメールで開発チームに報告します。サンドボックスを更新しても安全な状態に戻ります。



図3 - 7 ビルドブレークを解消する手順

ここでは、やらかした人を犯人と呼びましたが、罪を憎んで人を憎まず、悪気があってしたのではないのですから、犯人を責めるべきではありません。しかし、不注意により開発中のソフトウェアを一時的にも危地に立たせたのなら、犯人はチームにひと言謝罪すべきです[[32]](#footnote-32)。また、1.のステップでチームに怒りのメールを出す人をまれに見ることがありますが、それは良くありません。落ち着いて穏やかにメールしましょう。実は自分のサンドボックス環境に原因があった、なんてことになったら、恥をかくことになりますからね。

また、開発者の人数が多いと、ビルドブレークの通知があっても、「誰かが対応してくれるだろう」と全員が考えるため、対処が遅れることになりがちです。気をつけましょう。

|  |
| --- |
| To: 開発チーム  From: 中原 慶  Date: 2010/12/16 14:15  Subject: ビルドブレークしました  次のようなエラーが出ています。  SomeClass.java:114: cannot find symbol  symbol : class NewClass  location: class SomeClass  new NewClass();  ^  しばらく、更新は控えて下さい。 |

|  |
| --- |
| To: 開発チーム  From: 津田 義史  Date: 2010/12/16 14:45  Subject: Re: ビルドブレークしました  すみません、私がやらかしたようです。  すでにサンドボックスを更新してしまった人は、  ファイルSomeClass.javaのリビジョンを#18に戻すことで  ビルドブレークを回避できます。  現在、鋭意修正作業中です。  よろしくお願いします。 |

# リポジトリの汚染に備える

前述のように、いくら注意を払っても、コミットの失敗により一時的にリポジトリが汚染されてしまうことはあります。そのようなときに更新の操作をすると、自分のサンドボックスでもビルドブレークが発生するようになり、それが解消されるまでは自分の作業ができなくなってしまいます。しかし、3-6節で紹介した「頻繁にサンドボックスを更新すべし」というプラクティスに従えば、時にはそういうこともあるでしょう。

そうした状況に対処するには、あらかじめ自分のPCの中に複数のサンドボックスを作っておくことが有効です。例えば、同じブランチ上の同じ製品に対して、サンドボックスを2つ作っておき、サンドボックスAで開発作業を行います。ここで、そろそろサンドボックスを更新したくなったら、先にサンドボックスBを更新してビルドを通します。ビルドが通れば、サンドボックスAを更新しても安全です。あるいは、汚いサンドボックスAから、更新したばかりの清潔なサンドボックスBに作業を移しても良いでしょう。もしビルドが通らなければ、リポジトリが一時的に汚れていると考えられますので、「変なコミットした人、このエラー治して！」と呼ばわり、そのままサンドボックスAで開発作業を続けながら、リポジトリの回復を待てます。

# タグを付ける

SCMツールを使う大きな目的のひとつは、過去のビルドを再現可能にすることです。そのためには、ビルドのインプットとしたソースファイルの履歴をすべて保存するだけでは足りません。どのビルドが、どの時点 (スナップショット) のソースからビルドされたのかを記録しておかねばなりません。例えば、build#39のビルドはチェンジセット#何番のスナップショットを使ってビルドした、もしくは何年何月何日の何時何分何秒のスナップショットを使ってビルドした、などとメモを取っておきます。しかし、もっと良い方法があります。SCMには、ある時点のスナップショットに任意の名前をつける機能があります。この名前をタグといい、この操作を指してタグを付けるとかタグを打つといいます。例えば、build#39をビルドした時点のソースファイル一式に、次のような名前のタグを付けておきます[[33]](#footnote-33)。

|  |
| --- |
| ProductName\_mainline\_20110216\_build#39 |

上記では、タグ名の書式を<製品名>\_<ブランチ名>\_<日付>\_<ビルド番号>としています。タグ名はリポジトリ内でユニークとすべきなので、上記のように書式を統一しておくと便利です。

タグを付けておけば、後でこのタグを指定してソースファイル一式を取得 (更新) できます。これまでに付けたタグの一覧を取得することもできます。

タグを付けたいファイルの一式(ファイルセット)を指定する方法はいくつかあります。SCMツールによって、次のような方法のいくつか(あるいはすべて)が用意されています。

* 最新のリビジョンにタグ付けすることを指定
* 日時で指定
* チェンジセット番号で指定
* 現在のサンドボックスに取得したファイルのリビジョンを使って指定

これらのいずれかを、SCMツールのタグ付けコマンドに渡します。最後の方法を使うと、それぞれ違う時点における別のファイルを集めてファイルセットを作り、これに同じタグをつけることができます。

ただし、SCMツールによってはタグ名の変更や削除を追跡できないので注意しましょう。ずいぶん昔のことですが、筆者も誤って大事なタグを削除してしまい、これを自分の胸だけにこっそりしまっておいたことがありました。そのようなSCMツールでは、タグの代わりにブランチを使う方が安全です。実際、タグとブランチを区別しないSCMツールもあります。

図3 - 8 タグを付けておけば、後で  
そのタグがついたファイル一式を簡単に取り出せる[[34]](#footnote-34)

# バグが混入したタイミングを調べる

古いビルドをある程度の期間まで保存しておけば、あるバグがどのビルドから混入したかを簡単に調べられます。例えば、最新ビルド#32で、あるバグを発見したとしましょう。そうしたら、昔のビルド#16を取り出して、そのバグが再現するかテストします。再現しなければ、そのバグは#16から#32の間のどこかで混入したことがわかります。そこで、#16と#32の真ん中のビルド#24を取り出して、バグが再現するかテストします。再現したら、このバグは#16から#24のどこかで混入したとわかります。このような手順を繰り返していけば、どのビルドからそのバグが混入したかを突き止められます。このように、探す範囲を半分ずつにしながら探す方法を二分探索といいます。



図3 - 9 二分探索法で、どのビルド#からバグが混入したか調べる

図3 - 9は、この手順を最後まで繰り返した様子です。この結果をビルド番号の若い順に並べ直すと、次のようになりました(丸数字は、テストした順です)。

② build#16 再現しない

⑤ build#18 再現しない

⑥ build#19 再現しない

④ build#20 再現した

③ build#24 再現した

* 1. build#32 再現した

build#19まで再現せず、build#20以降で再現することが分かります。つまり、このバグはbuild#20で混入したのです。さらに、タグを使ってbuild#19からbuild#20の間にコミットしたチェンジセット一覧を取り出して調べれば、その中のどれがこのバグの原因なのか、突き止めることができます。このように、これまでのビルドすべて(上記の例では32個のビルド)をテストしなくても、二分探索を使えば、バグが最初に混入したビルドを探せます。

もし古いビルドを保存していなければ、昔のソースファイルをリポジトリから取り出して、必要なビルドを再現する (ビルドする) 手間が必要になりますから、たとえSCMリポジトリでソースファイルを管理していたとしても、これまでのビルドをある程度の期間、そのまま保存しておくことには価値があります。

ただし、原則としては、バグが入り込んだらそれをすぐに発見できなければなりません。この例では、build#20で混入したバグが、build#32をリリースするまで発見できなかったのですから、問題があります。その原因を追究しておきましょう。

# ブランチを切る

ある時点からサブブランチを作ることを、ブランチを分ける、あるいはブランチを切るなどと言います。これは、概念的にはリポジトリ内のある時点のファイルセット (スナップショット) を、リポジトリ内の別のディレクトリにコピーしてコミットすることです。すると、コピーされた2つのファイルは同じ履歴を共有する別のファイルになります。タグと同様に、任意のファイルの任意の時点からブランチを分けることができます。

優れたSCMツールは、サブブランチを作成してもリポジトリ内で元ブランチに含まれる全てのファイルを物理的にはコピーしません。そのため、この操作によりディスク容量を無駄に消費することはありません。ブランチを分けたファイルが修正されてコミットされたら、その差分だけがリポジトリに格納され、サブブランチの枝が伸びていきます。





図3 - 10 サブブランチを切ったときのディレクトリ構成の例

SCMツールのブランチング機能により、ある時点でのMainlineディレクトリからBranches\1.1ディレクトリを分ける例を図3 - 10に示します。リポジトリ内では物理的にコピーされていませんが、これをサンドボックス側に取り出すと、図3 - 10のように別のディレクトリに別のファイルとして取り出されます。ブランチを分けた直後は、どちらも全く同じ内容になっていますが、それぞれに別の変更をコミットすることができ、それぞれ違う枝に進化していきます。もちろん、メインブランチのディレクトリツリーだけをサンドボックス側に取り出すことも、あるいはサブブランチのディレクトリツリーだけを取り出すこともできます。

# ブランチとビルド番号

ブランチを分けたとき、各ブランチからリリースするビルドには、それぞれのブランチに通しの番号をつけることを勧めます。同じ製品だからといって、図3 - 11のように複数のブランチ間で通しの番号をつけると、手間がかかるし、間違いも起きやすくなってしまいます。



図3 - 11 間違いやすいビルド番号の付け方

ただし、ブランチごとの通し番号としたビルド番号では、どのブランチのビルドかわかりません。そこで、何らかの方法でブランチの識別子をビルド番号に含めます。

図3 - 12は、ビルド番号の百の位にブランチ識別番号を割り当てた例です。ビルド番号だけで、それぞれのビルドを区別できます。しかし、ビルド番号が大きくなるとビルド番号がかぶってしまうので、そうなる前にバージョン番号を増やすか、十分に大きい桁 (百の位でなく千の位など) をブランチ番号にする必要があります。



図3 - 12 百の位にブランチ識別番号を割り当てた例

図3 - 13は、マイナーバージョン番号でブランチを識別する例です[[35]](#footnote-35)。この例では、各サブブランチで元ブランチのビルド番号をそのまま使っていますが、1から数え直すのも良いでしょう。どちらにしても、ビルド番号のほかに、各ビルドを識別するためにバージョン番号が必要になります。



図3 - 13 マイナーバージョン番号でブランチを識別する例

このほか、ブランチ名もしくはその略称をビルド番号の先頭に付加するなどの方法もあります。運用しやすく便利な書式ポリシーをあらかじめ決めておきましょう。

# ブランチ間で差分をやりとりする

前述の通り、分けたばかりのサブブランチは、概念的には元のブランチのコピーです。このため、もし元のブランチ上のプログラムファイルにバグがあれば、サブブランチの同じファイルにも同じバグがあります。そこで、元のブランチでバグを修正したら、忘れずにサブブランチでも同じ作業をして、同じバグを修正しなければなりません。このように、あるブランチにコミットしたチェンジセットを、共通の履歴をもつ別のブランチにも適用してコミットする操作をポートといいます。

ポートを手作業で行うのは非常に間違えやすく、神経を使う作業です。ポートしたいチェンジセットに複数のファイルが含まれていれば、ポート先のブランチの同じ複数のファイルをすべて同じように修正しなければなりません。同じファイルといっても、別ブランチ上のファイルは少しずつ違っているでしょうから、正しく差分を適用するのもややこしいですし、作業漏れも発生しやすくなります。しかも、元のブランチから複数のサブブランチを分けていれば、そのすべてのサブブランチに対して同じ作業を繰り返さなければなりません。

ありがたいことに、多くのSCMツールはチェンジセットを自動でポートする機能を備えています。ポートを自動で行うには、一般にポート先のブランチと対応するサンドボックスを作成し、そのサンドボックス上で次の2つのパラメータをSCMのポートコマンドに渡します[[36]](#footnote-36)。

* 元ブランチの名前
* ポートしたいチェンジセットの番号

このほか、日時やタグで範囲を指定することにより、複数のチェンジセットを一度にポートすることもできます。ポートの操作をすると、サンドボックス上のファイルに、当該のチェンジセットがマージされた状態になります。念のため、ビルドとスモークテストをして、問題が見つからなければそれをまとめてリポジトリにコミットします。これで、指定したチェンジセットをサンドボックス経由でポートできたことになります。サンドボックス内にマージしたファイルは、コミットせずに破棄できることにも注意しましょう。逆方向のポートも同様に行えます。

1. ポート先のブランチで ② ポートの操作で、指定した  
   サンドボックスを更新 差分を自動的にマージ

 

1. これをコミットすれば

ポートが完了する



手作業によるポートとは、図中の②の操作を手作業で行うことですが、SCMツールはこれを自動化して支援します。

ポート元とポート先のファイルの内容が大きく違っていると、マージを自動で完了できず、「競合」という状態になります。このときは、サンドボックス上のファイルの競合した部分に、編集前の内容、自分が編集した後の内容、ほかの人が編集してコミット済みの内容が並んで表示されます。これを手作業で編集し、競合状態を解決 (適切にマージ) してからコミットします。SCMツールにより自動でマージできたファイルは、コミット前のコードレビューを義務づける必要はないと思います。しかし、手動でマージが必要となったファイルについては、コミット前のコードレビューをすべきかもしれません。

# フォワードポートとバックポート

**エラー! 参照元が見つかりません。**に示すように、元のブランチからサブブランチからへポートするとき、これをフォワードポートといいます。この逆向きの方向で行うポートは、バックポートといいます。

|  |  |
| --- | --- |
| C:\idd\doc\forwardport.png  図3 - 14 フォワードポート | C:\idd\doc\backport.png  図3 - 15 バックポート |

図中には、コミットをひとつだけ示していますが、実際にはどちらのブランチにもたくさんのコミットがされています。コミットしたときにブランチが伸びることを思い出してください。複数のコミットをまとめてポートすることをバルクポートといいます。



図3 - 16 バルクポート

また、フォワードポートの方向にバルクポートすることを(Forward Integration; 順方向の統合)、バックポートの方向にバルクポートすることを(Reverse Integration; 逆方向の統合)などといいます。バルクポートの対象となる複数のチェンジセットをまとめて、ペイロード(積み荷)などといいます。ペイロードに含まれるチェンジセットはまとめられ、ひとつのチェンジセットに入れられて、ポート先ブランチに一度にコミットされます。ペイロードに含まれるチェンジセットが非常に多いとき、これをビッグバン・インテグレーションといいます。競合の危険が大きくなりますから、ビッグバン・インテグレーションは避けるべきです。親子関係にあるブランチでは、ペイロードが大きくなりすぎない頻度で、定期的なバルクポートを計画すると良いでしょう。

あるブランチから別のブランチへポートするとき、あるチェンジセットだけをポートして、ほかのチェンジセットはポートせずにおくこともできます。また、チェンジセットに含まれる複数のファイルのうち、一部だけを選んでポートすることもできます。このように、必要なチェンジ(セット)だけを選んでポートすることを、バルクポートに対してさくらんぼ摘みポート("cherry-picking” port)といいます。

# マージの追跡

ふたつのブランチ間でポートを繰り返すことを双方向マージ (bi-directional merging) とか、はしごマージ (ladder merging)、階段マージ (stairway merging) などといいます[[37]](#footnote-37)。



図3 - 17 はしごマージ

しかし、このようなポートが苦手なSCMツールもあります。そのようなSCMは、バルクポートをしたときにポート済みの差分をもう一度ポートしようとしたり、ポートしてきた差分を元のブランチにポートして戻そうとしたりしてしまいます。こうなると、しなくていい競合が多く発生し、とても面倒です。ブランチを扱うのが上手なSCMツールは、どのチェンジセットがポート済みなのか、いつ誰がどこからどこにポートしたのかなどを覚えておいてくれるので、そのような問題は生じません。この機能をマージトラッキング (マージの追跡) などといいます。

もし、みなさんがブランチを活用したソフトウェア開発をしたいなら、GitやMercurialなどのブランチングに強いSCMツールを使うことを勧めます。Gitは、もともとひとつだったふたつのブランチを、ファスナーを閉じるようにポートして、枝を分けた時点を過去から未来に動かすリベースという機能があります。このほか、PerforceやMicrosoft Team Foundation Serverなどは有償のソフトウェアですが、やはりブランチングに強く、とても信頼できるSCMです。Perforceは2ユーザーまでなら無料で試用できるので、評価してみても良いでしょう[[38]](#footnote-38)。

SubversionもVer.1.5からマージトラッキングの機能がサポートされましたが、兄弟ブランチ間でのポートや、親から子へ、子からその妹へ、妹から親へといった間接的なポートはまだうまくできないようです。このようなSCMツールで双方向や間接的なポートを行うと、大量の競合が発生することがあります。また、直接の親子関係がないブランチ間では、自動的なマージができないSCMツールもあります。

# 誰がブランチを切る？

古典的ななぞなぞに、「東大寺は誰が建てたでしょうか？」というのがあります。そう、答えは聖武天皇ではなく大工さんです。そこで問題です。誰がブランチを切るのでしょうか？答えは、プロジェクトマネージャではなく構成管理者 (Software Configuration Manager) です。

ブランチを切り分けることには高度な判断が必要なため、プロジェクトマネージャやプロジェクトリード、デブリードらが事前に戦略的に計画します。その指示を受けて、SCMの操作に熟練した構成管理者がブランチを切る操作をします。

ブランチを切る機会それ自体がそれほど多くないこともあり[[39]](#footnote-39)、新人エンジニアのみなさんは、「今の自分は聖武天皇でも大工でもないし、ブランチの話題はとりあえず自分には関係ないな」と考えるかもしれません。しかし、そんなことはありません。いちどサブブランチを切れば、頻繁にポート作業を行う必要が生じるからです[[40]](#footnote-40)。あるブランチから3本のサブブランチが出ているとします。あなたがそのブランチでひとつバグを治したら、そのたびごとに3回のフォワードポートの作業が発生します[[41]](#footnote-41)。ブランチを切る機会は比較的少ないのに比べて、ポートの作業はとても日常的なものです。

ですから、ポートの操作そのものに熟達する必要があるのはもちろんですが、各ブランチがなぜ切られたのか、その背景をよく理解できるようにしておきましょう。次節では、ブランチングの戦略をいくつか紹介します。

# コラム 開発中の催し③ ブラウンバッグミーティング

|  |  |
| --- | --- |
| **コラム 開発中の催し③ ブラウンバッグミーティング**  ブラウンバッグとは、ハンバーガーやサンドイッチ屋さんで買い物をすると、買ったものを入れてくれる茶色の紙袋のことです。転じて、この茶袋を持ち込んでよいカジュアルなランチミーティングのことをブラウンバッグといいます。もちろん、ブラウンバッグだからといって茶色の紙袋を持ってこなければいけない、という決まりはありません。ブラウンバッグを定期的に催すことで、チーム内で技術情報を共有するよい機会を作れます。曜日を決めてブラウンバッグを週次で実施し、チームメンバーが持ち回りで一人ずつ何かを話してもらうようにすると、メンバーのプレゼンテーション技術を磨くこともできます。例えば、各人が関心を持っているテクノロジーや、プロジェクトで担当しているサブシステムの技術的な背景などが議題として適切です。   |  | | --- | | To: 開発プロジェクトチーム  From: デブリード  Date: 2010/09/21 15:28:03  Subject: ブラウンバッグしましょう  来週の火曜日のお昼から、毎週ブラウンバッグをしませんか。  メンバーが持ち回りで、それぞれ得意な技術分野について話しましょう。  プレゼン資料はあってもなくても結構です。  私が期待する話題としては、  ・各自が担当するサブシステムのアーキテクチャについて  ・Antの独自タスクの開発方法について  ・Mercurialのブランチの上手な使い方について  ・メールのMimeフォーマットについて  などです。また、ソフトウェアや仕事に関する話題以外でも結構です。  スケジュールしますので、各自話したいトピックと準備に必要な時間を知らせてください。  来週の火曜日までに準備できる人がいなければ、この日は私がジャグリングの魅力について語ろうと思います。  よろしくお願いします。 | |

# ブランチを切りたいとき

どのようなときにブランチを切りたくなるのか、考えてみましょう。さまざまな状況があります。

## 複数のバージョンを並行してサポートしたいとき

あなたは、ある製品をバージョン1としてリリースしました。さらに強力な機能を追加しながら、バージョン2、バージョン3をリリースしていきたいとします。このとき、バージョンごとにブランチを分けておくと、過去のバージョンもうまく保守できます。このようなブランチをリリースライン (release line) といいます。また、リリースごとにブランチを切るパターンをbranch per releaseといいます。

問題: メインラインだけでは  
複数のバージョンを並行して  
サポートできない

解決: バージョンごとにブランチを切れば、複数のバージョンを並行してサポートできる

 

Ver.1のラインにコミットしたバグ修正のためのチェンジセットは、忘れずメインラインにもバックポートしましょう。さらに、メインラインからVer2とVer3のラインへのフォワードポートも必要に応じて行います。

## 複数のユーザーに、それぞれ別にカスタマイズした製品をリリースしたいとき

あなたは、webで会議室予約ができるという実に画期的な製品を開発しました。これを複数のユーザーが購入してくれたとします。しかし、ユーザーAは会議室以外にも給湯室を予約できるようにしてほしいと言いました。また、ユーザーB はオフィスグリコを装備した会議室を区別して表示できるようにしてほしいと言いました。それぞれのカスタマイズ案件は各ユーザーに特殊すぎて、メインラインにコミットしたくありません。このようなときは、ユーザーAとユーザーBそれぞれに専用のブランチを切り、おのおののブランチにだけカスタマイズされた機能をコミットして管理します。このようなブランチを指してパーソナルコードライン (personal codeline) あるいはユーザーブランチ (user branch) などといいます。これはリリースラインの亜種と考えられます。

問題: 複数のユーザーに、それぞれ固有のカスタマイズを製品に施したい

解決: ユーザーごとに専用のブランチを切れば、ユーザーごとに固有の修正をコミットできる

 

## 過去のリリースを修正して再リリースしたいとき

あなたは、ある製品をユーザーにリリースしました。その後、さらにその製品を強力なものにするべく、このブランチ上で開発を続けています。ところが、製品のユーザーから致命的なバグの報告がありました。調査したところ、幸いにも簡単に修正でき、その影響範囲も限定的なものでした。しかし、このブランチ上では、次のリリースのために開発した機能がすでに多くコミットされています。テストも十分でなく、このブランチの最新のソースをビルドしてリリースすることはできません。このようなときは、ユーザーにリリースした時点から新しくブランチを切り、そのサブブランチ側に必要な修正を入れてビルドしてリリースします。これにより、現在ユーザーのところで動いているビルドとは、そのバグを修正した部分だけが異なるビルドを作れます。このようなサブブランチを指して保守ブランチ (maintenance branch) といいます。やはりリリースラインの応用といえますが、保守ブランチでは新規機能の開発はしないことが強調されます。

問題: 過去にリリースした製品を少し修正したいが、すでに多くの修正がコミットされている

解決: 過去のリリース時点から保守ブランチを切り、こっちに修正をコミットして再リリース

 

## リリースの準備に入ったバージョンと、次バージョンの開発を並行して作業したいとき

前節で紹介したのはリリース後にブランチを切るパターンですが、リリース前にもブランチを切ってしまうことがあります。例えば、ある製品のあるバージョンの開発がほぼ完了したと考えてください。しかし、ヘルプドキュメントなど、リリース前にいくつか追加のファイルをコミットしなければなりません。しかし、その作業の完了を待たずに次のバージョンの開発をすぐに始めたいとします。このようなときは、リリースの前にブランチを切って、ふたつの作業を並行して進められるようにします。このようなブランチをリリース準備ライン (release-prep line) といいます。リリース後は、そのままリリースラインとして運用することができます。

問題: リリースのためのコミットが終わっていないが、次バージョンの開発もすぐに始めたい

解決: リリース前にリリース用のブランチを切れば、次バージョンの開発を並行して行える

 

## メインラインを安定した状態のままにしたいとき

これまで述べてきたように、メインラインは大事に育てる必要があります。うかつなコードをメインラインにコミットして、すぐにそれに気づけず、さらにほかの変更をコミットしていけば、メインラインのビルドは不安定になってしまいます。そこで、メインラインには直接コミットせず、メインラインと並行したブランチで開発を行います。このブランチが安定したことを確認した段階で、それまでの修正をまとめてメインラインにバルクポートします。これにより、2-12のStable buildの節で説明したように、最新の機能を装備しているが不安定なLatest build (最新ビルド)と、少し古いが安心して使えるStable build (安定ビルド) のブランチを同時に並行して維持できます。このようなブランチを開発用ライン (active development line) もしくは開発ブランチ (development branch) といいます。

 

問題: メインラインを安定に保ったまま、開発作業を続けたい

解決: 別のブランチで開発し、安定した段階でメインラインにバルクポートする

## 長期の作業を、メインラインから隔離した場所で行いたいとき

例えば製品に対して大掛かりなリファクタリングをしたいとか、共通インターフェイスを変更するなどのような影響範囲が非常に広い修正をするときにも、ブランチの作成が有効です。このような修正は、一時的にもビルドを通らなくしてしまうため、ほかの開発者の作業をストップさせてしまいます。このため、メインラインとは別のブランチで作業してビルドを通し、テストでポートしても良い修正になったことを確認した上で、このブランチに入れたすべてのチェンジセットをメインラインにポートし、このブランチを破棄します[[42]](#footnote-42)。このように、ある程度長期にわたってメインラインと隔離した場所で作業をするためのブランチをアクティビティライン (activity line) もしくはタスクブランチ (task branch) といいます。また、短期でマージすることを意図した小さなタスクブランチをトピックブランチ(topic branch)といいます。ある機能を実装するために分けたブランチを機能ブランチ (functional branch) ということもあります。このように、タスクごとにブランチを切るパターンをbranch per taskといいます。この場合は、並行して元のラインにも多くのコミットがされるでしょう。適宜、それらを元のラインからタスクブランチにフォワードポートしておけば、最後にまとめてバックポートするときに競合が発生しにくくなります。

問題: ほかの開発者に影響を与えることなく、製品に一定の規模の変更を加えたい

解決:このタスクに関連する作業はタスクブランチで行い、ほかのタスクはメインラインで行う

 

# コラム リファクタリングとは

|  |
| --- |
| **コラム リファクタリングとは**  リファクタリングとは、ソフトウェアの振る舞いをまったく変更することなく、その内部構造をきれいに書き直すことです。ひと昔前は、動作しているソフトウェアには手を入れるべきではないと言われていました。ソフトウェアは非常に壊れやすいため、ソフトウェアの振舞いを変更しない修正をわざわざ入れてリグレッションの危険を冒すなど、以前は考えられなかったのです。しかし、「正しく動いているソフトウェアに手を入れることは禁止」というルールに従い続けていると、どうしてもそのソフトウェアの構造は壊れていき、保守ができなくなってしまいます。そこで、たとえ動いているソフトウェアであっても、書き直す勇気をもとう、というのがリファクタリングです。  もちろん、単に動いているソフトウェアに手を入れるというだけでは蛮勇というものです。リファクタリングは、テストによって支えられねばなりません。つまり、テストによって、リファクタリングの前後でそのソフトウェアの振る舞いが変わらないことを担保します。もし、大量のテストが自動化されていれば、リファクタリングをしようという勇気が得られます。テストによって振る舞いの変化を検出できるため、不適切なリファクタリングによる修正をコミットせずに済むからです。逆に、大規模かつ自動化されていないテストをすでに実施済みであるようなときは、コードを修正したいという誘惑には勝たねばなりません。このようなときは、リグレッションリスクが非常に大きくなるからです。ソフトウェアの内部構造を大きく変更するリファクタリングは、大量のテストが自動化されていない限り、リリース済みのソフトウェアに対して安全に行うのは困難です。 |

## 外部のライブラリやコンポーネントを管理したいとき

あなたは、自社製品の開発に、テスト済みで品質が良いことがわかっている外部のコンポーネント製品を利用することにしました。ところが、半年から数年程度経つと、以前のコンポーネント製品はサポートが切れて、入手することもできなくなってしまいました。これでは、以前の自社製品をビルドできません。あるいは、入手はできても、以前の自社製品をビルドするときにどのバージョンのコンポーネントを使ったのかを失念してしまうことがあります。この問題を解決するには、外部のライブラリやコンポーネントもリポジトリに入れておき、自社のソースコードと同じようにタグを打って管理することが有効です。このようなブランチをサードパーティコードライン (third party codeline) もしくはベンダーブランチ (vendor branch)といいます[[43]](#footnote-43)。このブランチはメインラインから分けて作るのではなく、最初から独立したラインになります。

もちろん、複数のプロジェクトにまたがって利用されている自社開発のコンポーネントについても同様の管理が可能です。これをサブシステムライン (subsystem line) もしくはコンポーネントブランチ (component branch) などといいます。

問題: 外部コンポーネントを使っている自社製品の、過去のビルドの再現性を確保したい

解決: コンポーネント製品を別のブランチに保存し、ビルドと同じタグを打っておく

 

## 大量のポートを安全に行いたいとき

複数のサブブランチに入れたチェンジセットをすべてメインラインに統合し、サブブランチは破棄したいとします。しかし、このバルクポートのペイロードが非常に大きいため、ポートの操作により多くのファイルがサンドボックス上で競合してしまいました。手作業でマージを完了できたとしても、これをいきなりメインラインにコミットするのは危険です。

このようなときは、ポート先のブランチの先端から新しくブランチを切り、まずこちらにバルクポートします。このブランチのビルドを安定させてから、このブランチに入れたチェンジセットをまとめて、ポート先のブランチにポートします。このようなブランチを統合ブランチ (integration branch) もしくは受け取りブランチ (receiving branch) などといいます。しかし、こまめにフォワードポートをすることで、統合ブランチを使わずにすむようにしたいものです。

問題: 複数のブランチをまとめて安全にメインラインにマージしたい

解決: 先に統合ブランチにポートし、このビルドを安定させてからメインラインへポート

 

## 複数のチームでひとつの製品を開発したいとき

規模の大きいソフトウェア製品を開発するときは、機能ごとにチームを編成し、それぞれが責任をもってその機能を作りこむことがあります。しかし、大勢が同じブランチで作業すると、簡単に競合が発生してビルドブレークしてしまいます。このようなときは、各チーム用のサブブランチを切り、まずはそのブランチにコミットする運用をすることがあります。機能ブランチと同様ですが、各チームの拠点が世界中に散らばっているとき、これをリモートライン (remote line) といいます。リモートラインは、図のように階層化して運用されることもあります。このとき、親ブランチをL1ブランチ、子ブランチをL2ブランチ、孫ブランチをL3ブランチなどとよんで区別します[[44]](#footnote-44)。

問題: 地理的に離れた複数の開発チームが同時に作業するブランチは壊れやすくなる

解決: 各開発チームのサブブランチから、定期的に差分をすぐ上のレベルにプロモートする

 

タスクブランチなどでは必要な最小限のファイルやディレクトリだけをブランチすることもありますが、リモートラインではメインラインに含まれるすべてのファイルをブランチすることがふつうです。リモートの開発支部でも、ビルドができるようにするためです。

表3 - 6 よく使われるコードラインの例

|  |  |  |
| --- | --- | --- |
| 名前 | 意味 | 別名 |
| Mainline | その製品を保持する一番重要なライン | Mainstream, Trunk, Top-level branch |
| Release line | あるリリースに対応するライン |  |
| Release-prep codeline | あるリリースを準備 (prepare) するためのライン。リリース後は、そのまま Release line もしくは Maintenance line となる。 |  |
| Maintenance line | 厳しくコミットを制限してリグレッションを防止し、本当に必要な修正だけを入れるライン |  |
| Active development line | 新しいバージョンを開発するためのライン | Development branch, Master branch |
| Task branch | ほかの開発者に影響を及ぼすことなく、ある大きな修正 (タスク) を完了させてから、それをまとめて元のラインにポートするためのライン | Activity branch, Topic branch |
| Functional branch | ある機能を実装するためのライン。タスクブランチの一種 |  |
| Experimental line | 破棄を前提とした実験的なプロトタイプのためのライン | Prototype line, Proto-line |
| Integration line | ほかの複数のブランチの修正を統合するためのライン | Receiving Line, Docking line, Consolidate branch |
| Project branch | 複数のタスクを同時に作業するためのライン |  |
| Private line | ブランチオーナーだけがコミットを許されるライン | Private branch |
| Component branch | 自社製のライブラリやコンポーネントを格納するライン | Module line, Subsystem line |
| Third party codeline | 他社(第三者)製のライブラリやコンポーネントを格納するライン | Vendor line |
| Platform line | 複数のプラットフォーム (OSなど) に対応するためのライン。プラットフォームに依存するソースファイルについてのみ、プラットフォームごとにサブブランチを切る[[45]](#footnote-45) | Variant line |
| Remote line | 離れた地域で並行して同じ製品を開発するとき、その地域でテストしてからメインラインにポートしてもらうためのライン | Site development line, Remote development line |
| Staging branch | チェンジセットを段階的にプロモートするためのブランチ |  |
| Frozen branch | コミットができないように凍結したブランチ | Snapshot branch |

# コードのプロモーション

Promotionとは昇格とか昇進という意味です。コードのプロモーションとは、あるブランチにコミットしたコード(実際にはチェンジセット)を、より安定したブランチにポートすることです。これは、開発ブランチを運用するときにとても役に立つ考え方です。



一般に、機能ブランチなどの開発ブランチを運用するときは、直接の親ブランチへの定期的なRIを計画しますが、これを指してコードのプロモーションというわけです。この用語は、より安定したブランチへのポートが丁寧に行われるべきことを示唆します。上図では機能ブランチAからのRIだけを表示していますが、機能ブランチBからメインラインへのRIも同様に定期的に行います。

このようなRIを実施する直前には、まず子ブランチでビルドを作ってテストし、コードをプロモートしても問題ないか確認します。ここで問題が見つかれば、単純にこのRIをキャンセルします。ただし、RIのキャンセルが何度か続くと、その次のRIがビッグバン・インテグレーションになってしまうので注意が必要です。また、RIを実施した直後に、ポート先のブランチでビルドを作ってテストする運用をすることもあります。

さらに、親ブランチにRIしたペイロード(チェンジセットの集合)は、すぐにほかの全ての子ブランチへFIします。安定したコードを、ほかの子ブランチに配るようなイメージです。一般に、FIのときにはRIと同様の丁寧なテストは不要です。

このような定期的なRIを計画するときは、そのカットオフ日時を明確に決めましょう。カットオフまでにコミットされたチェンジセットだけが、RIの対象となります。カットオフまでにコミットできなかったコードは、次回にスケジュールしたRIでプロモートします。

コードのプロモーションは、機能ブランチやリモートラインなどでブランチを親・子・孫のように複数の階層で構成したときも有用です。複数の階層構造でブランチを構成し、直接の親子関係にあるブランチ間で定期的なコードのプロモーションを実施すると、プロモート先のブランチをより安定させ、不正なコミットの影響を局所的にする効果があります。ただし、その代償としてビルドとテストの工数が増えますし、開発のペースも遅くなります。例えば、孫ブランチにコミットしたチェンジセットが子ブランチを経由して親ブランチにポートされるまでには、2回のRIが必要です。

# ブランチのプロモーション

ブランチのプロモーションとは、サブブランチだったブランチをメインラインに格上げすることです。例えば、あなたは今、メインラインと保守ブランチの両方をもっているとします。リリースした後にもそのユーザーからの要求が多く、保守ブランチで多くの修正や機能追加を行いました。ふと気づくと、最近はメインラインにほとんど手を入れておらず、ビルドもしていません。元々メインラインだったはずのブランチは実質破棄されており、保守ブランチだったはずのブランチがメインラインに変化してしまっているのです。そういえば、このメインラインだったはずのブランチも、もっと以前はほかのブランチから分けたサブブランチでした。このように、メインラインとなっているブランチを次々に乗り換えながら製品の開発・保守を続けることを、ブランチのプロモーションモデルといいます。



図3 - 18 ブランチのプロモーションモデル

ここに示したシナリオのような、なんちゃってプロモーションモデルはお勧めできません。これでは、メインラインとしてワークしているブランチはどれなのか、また現在保守を続けているブランチはどれとどれなのか、訳がわからなくなってしまいます。一般に、サブブランチでバグを修正したとき、その差分をメインラインにポートする手間を惜しむと、このようなことが起こります。ずいぶん前のことですが、筆者もポート作業を怠ったために、複数のブランチのうちどれが最新の修正を含むべきブランチなのか、訳が分からなくなりかけたことがありました。みなさんも気をつけましょう。

これに対し、元々メインラインだったブランチをそのままメインラインとして堅持し続けるモデルをメインラインモデルといいます。プロモーションモデルよりも、メインラインモデルを使うことを勧めます。



図3 - 19 メインラインモデル

メインラインモデルにより、製品の進化のための最上のブランチを1本残すことができます。メインラインモデルを維持するには、

* 保守ブランチからは次のメジャーバージョンの製品をリリースしない
* 保守ブランチにコミットしたチェンジセットは必ずメインラインにバックポートする
* 以前のバージョンをリタイアしたら、その保守ブランチは凍結して明確に破棄する
* サブブランチは、可能な限りメインラインから切る (なるべくサブブランチからは切らない、ブランチの階層を深くしすぎない)

などのプラクティスが有効です。

適切なプロモーションモデルの運用もあります。図3 - 20は、次のメジャーバージョンのための開発ブランチを、そのままそのバージョンのメインラインにプロモートした例です。ただし、この場合でも、各バージョンのメインラインは堅持すべきです。さもないと、各メインラインからマイナーリリースのための保守ブランチを切るなどした場合に混乱してしまいます。各バージョンのサポート終了に伴い、それぞれのメインラインは破棄します。



図3 - 20 プロモーションモデルによる  
branch per major releaseパターン

# コラム 複数のチームメンバーがいろんな事情で長期休暇

|  |
| --- |
| **コラム 複数のチームメンバーがいろんな事情で長期休暇**  複数のチームメンバーが、バスに轢かれて入院しました。もし、このためにプロジェクトを続行できなくなったとき、その入院した人数をバスナンバー (bus number) といいます。同様に、複数のメンバーがトラックに轢かれて入院しました。このためにプロジェクトを続行できなくなったとき、その入院した人数をトラックナンバー (truck number)といいます。同様に、複数のメンバーが新婚旅行にでかけたためにプロジェクトを続行できなくなったとき、その数をハネムーンナンバー (honeymoon number) といいます。呼びかたは違いますが、どれも同じ意味です。もちろん、これらの数は、大きいほどチームが頑健であることを示すので、好ましいものです。まあ、大きすぎても、それはチームに無駄な人員を抱えているということかもしれませんが。。バスナンバーを小さくするには、特定の人に依存した作業を作らないことです。あなたのプロジェクトのバスナンバーを数えてみましょう。 |

# 必要なさくらんぼ摘みポートを追跡する

保守ブランチに入れたコミットを、メインラインにポートするのを忘れルことはあまりありません。保守ブランチにはバグの修正しかコミットされていませんから、後でまとめてメインラインにバルクポートできるからです。しかし、ユーザーブランチには、バグ修正と各ブランチ固有の修正の両方がコミットされますから、必要なチェンジセットだけを拾ってポートする必要があります。このようなさくらんぼ摘みポートは、どうしても忘れがちです。もしポートを忘れると、「あれ、このバグどこかで見たような。。修正したような気がするけど、どのブランチで治したっけ？」と、バグ追跡システム (BTS) に登録されたバグ報告票を探し回ることになります。

そこで、後でポートが必要になる修正をコミットしたら、そのポート作業を指示するバグ報告票をBTSに登録しておきましょう。後で摘みたいさくらんぼ(ポートしたいチェンジセット)に、ポート先のブランチの数と同じ枚数のバグ報告票を貼り付けておけば、後でそのポート作業を忘れずに済むというわけです。

このバグ報告票は、元の修正を指示したバグ報告票をコピーし、作業対象のブランチを適切に書き換えてファイルします。これらのバグ報告票の間にはリンクを張り、こっちのブランチのチェンジセット#XXで修正済み、と記入しておくのも忘れないようにしましょう。ポート先のブランチからビルドをリリースするとき、そのブランチのオーナーがそのバグ報告票を適切に処理してくれるはずです。このバグ報告票は、ポートが完了したら、通常のバグ報告票と同じプロセスを経てクローズします。

# ブランチへのアクセスを制御する

あるブランチに対して、コミットを許したくない時期があります。例えば、ある製品がリリースに近づき、フルリグレッションテストを開始した後などです。もし、コードを修正してしまうと、またフルリグレッションテストを最初からやり直さなくてはいけません。そこで、コードフリーズを宣言してコミットを禁止します。ところが、単にコミットの禁止を宣言しただけでは、開発者の誰かが細かいバグをこっそり修正してコミットしてしまうかもしれません。もちろん、リグレッションが発生する危険がありますから、許されることではありません。もしもバグが見つかれば、これを治すべきかどうか慎重に判断しなければいけません[[46]](#footnote-46)が、開発者が勝手にコミットしてしまうと、製品をコントロールできなくなってしまいます。

あるいは、ビルドマシンでのビルドが失敗することを避けるために、ビルド中はコミットを一時的に禁止したいこともあります。このほか、開発チームにはコミットを許すが、テストチームには閲覧(更新の操作)しか許さないとか[[47]](#footnote-47)、プロジェクトの外部の人には閲覧も禁止したい、というように、SCMへのアクセスレベルを細かく調整したいことがあります。

このような要求に応えるために、多くのSCMはブランチごと、あるいはファイルごとにきめ細かくアクセス権を設定する機能を備えています。ブランチを切ったら、すぐに適切なアクセス権を設定しましょう。

このほか、あるブランチを監視し、何かがコミットされたらブランチオーナーにメールで通知するなどの機能をもつものもあります。お使いのSCMツールのマニュアルをよく読み、活用できる機能を探すと良いでしょう。もし必要とする機能がなければ、別のSCMツールを探す必要があるかもしれません。

# ブランチを廃棄する

さまざまな目的のために切ったブランチは、その目的を達すれば不要になり、廃棄されます。前節で説明したSCMの機能を使って、廃棄したブランチは誰もコミットできないようにしておくことを勧めます。これにより、誰かが誤って廃棄したブランチに何かをコミットしてしまう失敗を防ぎます。

また、ユーザーにリリースした製品のブランチを破棄 (リタイア)　するときには、一般に次のことを決める必要があります。

* どのバージョンの製品のサポートを打ち切るのか。
* ユーザーには、それをどのように知らせるか。また、サポートを継続できる新しいバージョンの製品にどのように乗り換えてもらうのか。

# ブランチを上手に使うためのプラクティス

ここまで、さまざまな状況でどのようにブランチを切るべきかを紹介してきました。本節では、ブランチを上手に使うための一般的なプラクティスを示し、ブランチの話題をまとめます。

## ブランチを切らない

ブランチを覚えると、つい多く切ってみたくなるものです。しかし、ブランチが増えると、ポートやリリースの手間 (コスト) も増えてしまいます。そこで、コミットについて別のポリシーが必要になったときだけ、ブランチを切りましょう。例えば、開発ブランチと保守ブランチでは、コミットしてよい修正の種類が違います。このように、ブランチごとに適用する別のポリシーをブランチポリシーといいます。必要なブランチポリシーが見つかるまでは、開発者全員がメインライン上で作業します。可能な限り、メインライン・セントリックにいきましょう。

## ブランチに良い名前をつける

新しく必要なブランチポリシーを見つけたら、それを連想できる適切な名前を新しいブランチに付けましょう。プログラムを書くときは意味のある関数名や変数名をつけるべきことと同じです。適切な名前やポリシーをもたないブランチは、意味不明ブランチ(unknown branch)とか謎めきブランチ(mystery branch)などとよばれて忌み嫌われます。

## ブランチングを遅らせる

例えば3-24節で紹介したユーザーブランチのように、別ユーザーのためのカスタマイズ用ブランチを切る必要が生じたとしましょう。このようなときは、なるべくブランチを切るタイミングを遅らせるのが良い方法です。あまり早くにブランチを切ると、その後で発生したバグの修正などのポート作業が多く発生してしまうからです。

## ブランチングを早める

前節と相反するようですが、例えば3-24節で紹介したリリース準備ラインを検討しましょう。リグレッションテストを終え、コードフリーズが宣言されたなら、リリースを待たずにそのタイミングでブランチをさっさと切ってしまいましょう。これにより、次のバージョンの作業にすぐ着手できるようになります。

## フリーズよりもブランチを使う

SCMツールのアクセス制御機能により、あるブランチに対して任意のコミットを禁止することをブランチフリーズといいます。ブランチフリーズにより、確実にコードフリーズを行えます。しかし、開発ブランチを長期間フリーズすることは開発者の作業を滞らせてしまうので避けましょう。ブランチをフリーズする代わりにタグを打っておけば、後で必要に応じてそこからブランチを切ることができます。

## マージは早めに頻繁に

後でポートしなければならないことがわかっているチェンジセットは、なるべく早めにポートしましょう。時間が経つと、ふたつのブランチに含まれるソースファイルは少しずつ違うものに変化していくため、競合が発生しやすく、またこれを解決するのも難しくなりやすいものです。

## さくらんぼ摘みポートよりもバルクポートを使う

さくらんぼ摘みポートは、追跡するのに余計な手間がかかりますし、必要なポートを忘れたりするので損です。可能な限り、バルクポートで事が足りるようにブランチを構成しましょう。

## ポートを計画する

必要なポート作業は、きちんと開発計画や見積もりに盛り込みましょう。さくらんぼ摘みポートをするなら、BTSで必要なチェンジセットを追跡して適宜ポートします。バルクポートをするなら定期的なインテグレーションを計画します。ブランチごとに、ポートポリシーを決めておくと良いでしょう。

## 各チームメンバー専用のリポジトリを用意する

ファイルのすべての履歴を保存できるSCMリポジトリは、ファイルをバックアップする場所として非常に便利です。そこで、すべてのチームメンバーに、それぞれの一時的な作業場として、専用のリポジトリ(プライベートライン)を準備しておくと役立ちます。SCMツールの操作に慣れたり、実験したりする場所としても重宝します。このようなリポジトリは、プロジェクトに関係なく、各メンバーに恒久的に用意されていればより快適です。ただし、各開発者用に製品ブランチを別に切って用意すべきではないことに注意して下さい。これでは、ソースコードを共有するためにSCMを使う意味がなくなってしまいます。

## コードは共同所有、ブランチは担当者所有

以前は、コードは書いた人が責任をもち、それを書いた人だけがそのソースファイルを修正すべきと言われることもありました。しかし現在では、コードの設計情報を共有することの有用性や、コードの所有者が (交通事故や新婚旅行などで) いなくなったときに作業が滞ってしまうリスクを避けるといった側面から、コードは開発者全員で共同所有した方がメリットが大きいことが広く認知されています[[48]](#footnote-48)。

しかし、コードは共同所有であっても、ブランチについてはブランチオーナーを決めておくと良いでしょう。ブランチオーナーは、そのブランチに管理されているファイルの一貫性に責任をもちます。コードレビューやトリアージを担当するなどして、このブランチへのコミットについての役割を果たします。また、必要に応じてそのブランチを凍結したり、凍結解除します。例えば、フィーチャーブランチ(機能ブランチ)については、そのフィーチャーオーナーがブランチオーナーを兼務します。このほか、ブランチオーナーがビルドマスターを兼務し、そのブランチからのビルドとリリースに責任を持つこともあります。

# ポリシーを文書化する

SCMのリポジトリを運用するときは、そのポリシーを決めておくと役に立ちます。次のようなものを用意しておくと良いでしょう。

* **ブランチポリシー**
  + このブランチの目的
  + このブランチのブランチオーナーの名前
  + このブランチへのアクセス制御リスト
  + ビルドの頻度とタグの書式、バージョン情報の書式
  + このブランチからのブランチングポリシー
  + このブランチへのコミットポリシー
  + サブブランチ間とのポートポリシー
  + このブランチでのビルドのバグ起票ポリシー

など

* **ブランチングポリシー**
  + いつブランチを切るか
  + 誰がブランチを切れるか
  + ブランチの命名ルール
  + ポリシーの文書化と更新時の周知ルール

など

* **コミットポリシー**
  + コミットできる修正の種類 (新機能の追加、些末なバグの修正、深刻なバグの修正、バグの優先度など)
  + コミット前に義務付けられる手順のリスト (★3-13節を参照)

など

* **ポートポリシー**
  + 誰が責任をもってポートするか
  + さくらんぼ摘みポートとバルクポートのどちらをするか
  + どのテストにpassすればポート可能とするか
  + ポートの頻度や曜日

など

* **バグの起票ポリシー**
  + 起票したバグのアサインポリシー
  + 親子・兄弟ブランチで、同じバグに対しひとつのバグを起票するか、それぞれ別のバグを起票するか
  + 構成やプラットフォームで、同じバグに対しひとつのバグを起票するか、それぞれ別のバグを起票するか
  + その他、ブランチごとにバグ報告票のテンプレート(項目ごとのデフォルト値)の準備

など

ポリシーを文書化したら、これをそのブランチのルートにコミットしておくのも良い習慣です。開発の途中でコミットポリシーを変更したら、漏れなくチームメンバーに周知しましょう。

また、フィーチャーオーナーリストを作っておくと、コードレビュー依頼時やバグ報告票のアサイン時に役立ちます。フィーチャーオーナーとは機能ごとのリストには、機能ごとの開発担当者とテスト担当者の名前を記入します。★プログラムマネージャとフィーチャークルーについても言及？

# SCM サーバの管理

SCMサーバ (リポジトリ) は、各プロジェクトの構成管理者が管理・運用します。その任務の中でも、SCMサーバが使うディスク装置の容量と、パフォーマンスの状態(SCMツールのコマンドが遅滞なく完了するか)の監視が特に重要です。このほか、定期的にリポジトリをバックアップしたり、必要に応じて各メンバーにリポジトリやブランチを作成してアクセス権限を与えるなどの業務があります。

各ブランチは、構成管理者から権限の委譲を受けてブランチオーナーが管理します (通常、ブランチオーナーは開発者が兼務します)。ただし、SCMがソフトウェア開発に必須のインフラであることをよく理解している企業では、メールサーバと同じように、SCMサーバのサービスを全社に提供するIT部門を設ける場合があります。このサービスを利用したいプロジェクトは、この部門に申請して、これから開発する製品のためのリポジトリとブランチを作ってもらいます。また、リポジトリの定期的なバックアップもこの部門が行います。このような部門があると、個々のプロジェクトチームが負担するコストは大きく削減されます。



図3 - 21 社内ITが提供するサービスの例[[49]](#footnote-49)

# まとめ

本章では、サンドボックスの扱い方と、さまざまなブランチングの戦略を紹介しました。これらのパターンをうまく組み合わせて、みなさんのソフトウェア開発に適切なブランチングを計画してください。

表3 - 8に、SCMの概念や操作の名前とその意味を示し、本章のまとめとします。各SCMツールで使われるありがちなコマンド名も、各項目に添付しました。SCMツールによっては、同じ概念を指すのに違う用語を使ったり、違う概念を指すのに同じ用語を使ったりすることがあるので注意してください。また、ここに示した概念や操作のいくつかをもたないSCMもあります。

表3 - 8 SCMの概念や操作を表す言葉

|  |  |  |
| --- | --- | --- |
| 用語 | 意味 | 別名 |
| リポジトリ | SCMサーバが管理するファイル置き場の領域 | repository, depot, archive, VOB, database |
| サンドボックス | SCMクライアントとなっている、開発者の作業環境 | sandbox, view,  client workspace, work area, project, enlistment |
| リビジョン番号 | 各ファイルにつけられる履歴の番号 | revision#, changeset#, changelist# |
| 最新リビジョン | 各ファイルもしくはスナップショットの一番新しいリビジョン | latest revision, head,  tip revision |
| スナップショット | 任意のブランチの、任意の時点に含まれるファイル一式 | snapshot |
| ファイルセット | ファイル一式。正確な定義があるわけではないが、スナップショットといえばビルドを構成するソース全体、ファイルセットといえば任意のファイルの集まりというニュアンスがある | fileset |
| タグ | スナップショットにつけることのできる名前 | tag, label |
| ブランチ | 履歴が管理されたファイル一式を複製したもの | branch, codeline, stream, tree |
| メインライン | 開発作業の根幹となる、一番大事なブランチ。親ブランチをもたないブランチ | mainline, trunk, main branch, codebase |
| 差分 | あるファイルの別のリビジョン間の違い | diff, patch |
| チェンジセット | 複数のファイルにまたがって同時にコミットされる単位。差分のセット | changeset, changelist, transaction, patch |
| チェンジセット番号 | コミットの単位に付加される番号。また、ブランチ上の一点を示す番号 | changeset#, changelist#, commit# |
| 競合 | 同じの同じファイルを複数人が同時に編集することにより、マージが必要になった状態 | conflict |
| マージ | 競合したファイルを併合し、コミットできるようにすること。競合の状態によっては、SCMツールによる自動的なマージが可能。また、ブランチを併合することを指してもいう | merge, integrate, resolve |
| 解決 | 競合したファイルを手動でマージすること。また、マージが完了し、そのファイルをコミットしても安全な状態になったことをSCMツールに知らせること | resolve |
| 追加 | リポジトリに新しいファイルを追加すること | add, checkin |
| 更新 | リポジトリからサンドボックスにファイルを取得すること | update, sync, checkout, refresh, get |
| 編集 | サンドボックス内のあるファイルに、修正中であるという印(ダーティフラグ)をつけること。 | edit, checkout |
| コミット | サンドボックス中のファイルを、リポジトリに追加もしくは上書きして格納すること | commit, submit, checkin, put, keep |
| リバート | サンドボックスでの修正を破棄すること。リポジトリにコミット済みの修正をリバートできるSCMツールもある | revert, undo |
| ポート | あるブランチ上のある時点からある時点までの差分を、同じ履歴を共有する別のブランチにも適用すること | port, integrate, merge |

# 参考文献

## Sandbox (software development)

<http://en.wikipedia.org/wiki/Sandbox_(software_development)>

## Sandbox (computer security)

<http://en.wikipedia.org/wiki/Sandbox_(computer_security)>

## Streamed Lines: Branching Patterns for Parallel Software Development

Brad Appleton, Stephen P. Berczuk, Ralph Cabrera, Robert Orenstein

<http://www.cmcrossroads.com/bradapp/acme/branching/>

## Microsoft Team Foundation Server Branching Guidance

<http://branchingguidance.codeplex.com/>

## CVS II: Parallelizing Software Development

Brian Berliner

<http://docs.freebsd.org/44doc/psd/28.cvs/paper.pdf>

## High-level Best Practices in Software Configuration Management

Laura Wingerd & Christopher Seiwald

<http://www.perforce.com/perforce/papers/bestpractices.html>

## Smoke testing

<http://en.wikipedia.org/wiki/Smoke_testing>

## 反復型開発のエコノミクス

Walker Royce, Kurt Bittner, Mike Perrow, 藤井 拓 (監訳)

<http://www.amazon.co.jp/dp/4894715449>

1. SCMを、Source Code Managementの略とする人もいます。また、ソフトウェアの構成管理を支援するツールをRC (Revision Control), VC (Version Control), SC (Source Control) などということもありますが、本書ではSCMで統一しました。 [↑](#footnote-ref-1)
2. 広義には、このほかにもソースファイルを格納するディレクトリ構造はどうあるべきかや、さまざまな要求や設計のための文書管理についてなどの話題も含みますが、本書では扱いません。 [↑](#footnote-ref-2)
3. subversionという語には、転覆とか破滅という意味があります。 [↑](#footnote-ref-3)
4. gitという語は、イギリスのスラング英語で役立たずの間抜け、という意味です。 [↑](#footnote-ref-4)
5. ビルドマシンでは開発作業をすべきではありませんが、バージョン情報ファイルなどはビルドマシン上で修正してコミットすることがあります。 [↑](#footnote-ref-5)
6. 本書の文脈では、サーバとは人ではなくソフトウェアです。もちろん、テニスのサーブをする人ではありません。クライアントも同様です。先進的なサーバの例については、RFC2324を参照して下さい。 [↑](#footnote-ref-6)
7. ブランチ名も必要ですが、一般にサンドボックスは必ず特定のブランチに関連づけられているため、サンドボックスで作業するときにはブランチ名を明示的に指定する必要はありません。ブランチ名は、最初にサンドボックスを作成するときに指定します。 [↑](#footnote-ref-7)
8. このような開発環境のほか、コンピュータウィルスのワクチンソフトの開発のために、ウィルスを安全に閉じ込めてテストできる環境や実験場などもサンドボックスといいます。本文中で紹介したサンドボックスと同じく、外界と隔絶した環境という意味です。 [↑](#footnote-ref-8)
9. breakfastみたいなlunchはbrunchです。 [↑](#footnote-ref-9)
10. ビルドブレークについては4章で扱います。 [↑](#footnote-ref-10)
11. この後にコミットしようとすると、実際には「先に更新しないとコミットできません」というエラーになるので、ほかの人の修正を上書きして消してしまう心配はありません。 [↑](#footnote-ref-11)
12. 別の人が同じファイルの同じ場所を修正した場合は、自動でマージできません。この場合は、手作業でマージする必要があります。このような状態を「競合」といいます。 [↑](#footnote-ref-12)
13. このほかにも、そのチェンジセットをコミットした人の名前や、コミットの日時、コミットの際に入力したコメントコメントなどが、チェンジセット番号に関連付けられてリポジトリ内に記録されます。 [↑](#footnote-ref-13)
14. 例えば、ねこちゃんがお砂場の中にウンコしたら、支障が出ますよね。 [↑](#footnote-ref-14)
15. プラクティスとは、守るべき習慣や作法のことです。 [↑](#footnote-ref-15)
16. コンパイルエラーなどでビルドができない状態を「ビルドが通らない」といいます。また、問題なくビルドが完了することを確認すること、もしくはビルドエラーを除去してビルドできるようにすることを「ビルドを通す」といいます。 [↑](#footnote-ref-16)
17. このほか、SCM以外の文脈でも、編集中だがまだ保存していないデータ(ファイル)を指してdirtyということがあります。例えば、あるデータが未保存かどうかを検査する関数にはIsDirty()という名前がよく使われます。ところで、クリント・イーストウッドの映画「ダーティハリー」が日本で初公開された当時、邦題は「汚いハリー」になっていたそうですよ。 [↑](#footnote-ref-17)
18. とは、WebページをWebブラウザで作成・編集できる情報共有システムで、インターネットのであるウィキペディアの構築にも使われています。エクストリーム・プログラミングの創始者のひとりであるウォード・カニンガム氏が開発しました。 [↑](#footnote-ref-18)
19. あるサンドボックスを、後で別のブランチ (親子関係や兄弟関係にあるブランチ) に対応づけ直すことができるSCMツールもあります。ただし、このようなツールの支援がない限り、ひとつのサンドボックスを複数のブランチに使いまわすのは避けるべきです。 [↑](#footnote-ref-19)
20. 私が一番好きなギタリスト、パット・メセニーの言葉が引用されています。 [↑](#footnote-ref-20)
21. 分散ビルドのテクノロジーを使えば、複数台のPCをビルドマシンとして活用できます。分散ビルドについては4章で紹介します。 [↑](#footnote-ref-21)
22. 詳細なビルドの手順は4章で説明します。 [↑](#footnote-ref-22)
23. ポートとは、あるブランチにコミット済みの差分を別のブランチにも適用してコミットすることです。ポートについては後述します。 [↑](#footnote-ref-23)
24. レビューを受ける人のことを、レビューイ (reviewee) といいます。本文中では、レビューイとコミッタ (コミットをする人) は同一人物なので、コミッタで統一しました。 [↑](#footnote-ref-24)
25. ドライブする人 (ここでは、その機能の実装に責任をもつ担当者) をドライバといいます。機能ごとに、その実装をドライブするフィーチャーオーナーと、そのテストをドライブするテストオーナーを決めておくと良いでしょう。 [↑](#footnote-ref-25)
26. ここに示した書式は、少々詳細すぎるかもしれません。詳細すぎると、開発者は頻繁なコミットを嫌がるようになります。不要な項目を除いて、あなたの組織にちょうど良い書式を探してください。 [↑](#footnote-ref-26)
27. ペアプログラミングとはエクストリーム・プログラミングのプラクティスのひとつで、2人が1台のPCを共有して一緒にプログラミングをすることです。 [↑](#footnote-ref-27)
28. ビルドサーバとは、より高度なビルドを自動化するためのソフトウェアです。4章で紹介します。 [↑](#footnote-ref-28)
29. このほかのビルドの壊れ方については、4-5節で紹介します。 [↑](#footnote-ref-29)
30. ただし、サンドボックス自体が壊れることもあります。これは大変良くありません。変なビルドエラーがなかなか治らないときは、サンドボックスが壊れていることを疑いましょう。 [↑](#footnote-ref-30)
31. ソフトウェアあんどんは、4-5節で紹介します。 [↑](#footnote-ref-31)
32. ビルドブレークを発生させた犯人は、次に誰かがビルドブレークを起こすまでの間、ビルドサーバのお守りをしなければならないという過酷な懲罰:)を課す組織もあると聞きます。 [↑](#footnote-ref-32)
33. タグは、ひとつのファイルだけに付けることもできます。しかし、関連するファイルすべて (ファイルセット) に同じタグを付けるのが一般的な使い方です。 [↑](#footnote-ref-33)
34. @マークはatの略記記号で、日本語では単価記号とよばれます。ソフトウェア以外の文脈でも、日付や時刻、場所などの記入に日常的に使われます。 [↑](#footnote-ref-34)
35. これをbranch per minor releaseパターンといいます。 [↑](#footnote-ref-35)
36. コマンドの詳細は、SCMツールによって異なります。お使いのSCMのマニュアルを参照してください。 [↑](#footnote-ref-36)
37. 私は、これを「あみだマージ」と呼んでいます。 [↑](#footnote-ref-37)
38. May the perForce be with you. (ぱフォースと共にあらんことを。) ― unknown [↑](#footnote-ref-38)
39. 強力なブランチの機能をもつGitのようなSCMを活用するときは、とても頻繁に (1日に何度も) 個人の作業用ブランチを切ることがあります。 [↑](#footnote-ref-39)
40. もちろん、まずポートの作業が最小限になるように、開発全体を計画しておくことが重要です。 [↑](#footnote-ref-40)
41. 定期的なバルクポートを計画できれば、何かをコミットするたびにポートの作業は発生しません。バルクポートを計画するためのヒントは3-24節で示します。 [↑](#footnote-ref-41)
42. このような操作をブランチのマージといいます。ファイルのマージとは概念的に異なる操作なので、区別して覚えましょう。 [↑](#footnote-ref-42)
43. ベンダーブランチを使うときは、外部の製品の著作権に注意してください。 [↑](#footnote-ref-43)
44. レベルもしくはレイヤのLです。 [↑](#footnote-ref-44)
45. 2章のコラムで紹介したように、C/C++ではブランチでなくプリプロセッサを使って複数のプラットフォームに対応することもよくあります。 [↑](#footnote-ref-45)
46. このような判断をトリアージといいます。トリアージは5章であつかいます。 [↑](#footnote-ref-46)
47. もちろん、テストチームがオートメーションの開発を担当するときは、テスターにもコミットできる権限が必要です。 [↑](#footnote-ref-47)
48. エクストリーム・プログラミングには、コードは共同所有すべきというプラクティスがあります。 [↑](#footnote-ref-48)
49. ビルドサーバは、一般にIT部門で管理されることはありません。ただし、プロジェクト内に専任のビルドチームを配し、ビルドを担当してもらうことはあります。 [↑](#footnote-ref-49)