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)。SCMといえばサプライ・チェーン・マネジメントを想像される方も多いでしょうが、それとは関係ありません。構成管理とはわかりにくい言葉ですが、広義には製品を構成する部品や要素の変更を管理することです。ソフトウェア開発の文脈においては、狭義にはファイルの履歴管理、バージョン管理のことです[[2]](#footnote-2)。SCMのリポジトリ (保管場所) に格納されたファイルは、リビジョン番号 (履歴番号) が付けられ、その履歴が管理されます。この機能により、リポジトリに保存した任意のプログラムを任意の時点に戻すことができます。SCM の歴史は古く、世界最初のSCMソフトウェアである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ソフトウェアに共通する概念や、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。

**これはSCMの一部の一レベル。**

**はじめにのコメントでも書いたように、これはちょっと狭い**

# 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。

**おかしい。**

**ちなみにリポジットリはどちらかというとツール/物理的な観点の用語で、論理的にはライブラリということになります。**



図3 - 1 リポジトリとサンドボックス[[4]](#footnote-4)

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[[5]](#footnote-5)。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。リポジトリの中には、過去のソースファイル一式 (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とリビジョン番号 (履歴番号) が必要です[[6]](#footnote-6)。

**スナップショットの意味が適切に伝わらないのでは？**

**やっぱり前の章で説明しておいた方がいいですよね。**

**といっても、ここで使っているリビジョンは正しくはリビジョンではない。→ こんがらがるから説明しない方がいい？**



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を指定しなければ、最新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、開発者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得して開発作業をします。

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、リポジトリ内にファイルを追加もしくは上書きする操作を「コミット」といいます。コミットすると、当該のファイルのリビジョン番号 (履歴番号) がひとつ増え、新しい履歴が保存されます[[7]](#footnote-7)。この操作により、図中の枝が右側に伸びていきます。

つまり、複数の開発者はそれぞれのサンドボックス上でコードを修正し、その修正を並行してリポジトリにコミットします。これにより、リポジトリ内の枝が伸びていきます。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) をしてビルドをします。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、その外に出てはいけません。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だけをサンドボックスから取り出してSCMの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

サンドボックスでファイルを修正し、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. 最新リビジョンのファイルで 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Rev#33ができる



このように、SCMのブランチ (枝) はコードの差分が積み重なるようにして伸びていきます。この、一度にコミットする単位に含まれる差分の集まりをチェンジセットといいます。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変更 (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。

## 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

 

1. この状態で再度更新すると ④ マージされたファイルを  
   修正が自動でマージされる[[9]](#footnote-9) 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などのアプリケーションデータファイル) にはコピー・修正・マージ方式が使えないので、ロック・修正・アンロック方式を使います。

# 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[[10]](#footnote-10)。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[[11]](#footnote-11)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[[12]](#footnote-12)。

## 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、チームのレビューと承認を受けてリポジトリにコミットする[[13]](#footnote-13)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変更を除去すべし

**これの具体的なやり方のガイドは出さないのですか？**

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[[14]](#footnote-14)。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、そのような汚いファイルを誤ってリポジトリ内にコミットしてしまわないとも限りませんし、そんな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るからです。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、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除去できます。また、SCMツールのリバートという操作により除去することもできます。

# サンドボックスの構築手順の確立と文書化

**多くの場合、開発者はいろいろ兼務しています。**

**対象の作業に応じたワーキングセット/環境をいかにサンドボックスに作るか/切り替えるかは、実践的にはとても重要です。**

**このテーマも扱った方が良いのではないでしょうか？**

サンドボックス環境を構築するのは、なかなか大変な作業です。多くの場合、この手順は、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ものになります。この手順は非常に複雑になることがあるので、必ずサンドボックス構築手順書を記述しておきましょう。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[[15]](#footnote-15)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[[16]](#footnote-16)。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する必要があります。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3-13節で取り上げます。



図3 - 3 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に対応する

# 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[[17]](#footnote-17)」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。  **このコラムは何を言おうとしているのかよくわからない。**  **最後の、言語の一種は、当たり前すぎ。** |

# 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[[18]](#footnote-18)。サンドボックスは汚いですから、きちんとしたビルド (QA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のです。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。ビルドマシンとビルドの手順については、4章で詳しく説明します。

# 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 - 3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[[19]](#footnote-19)。



図3 - 3 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。これらの言語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、コンパイラという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 (コンパイル) 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。

一般に、ビルドをリポジトリの中に入れて管理することはしません。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るのは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しておくので十分です。ただし、マイルストーンビルドやリリースビルドなど、重要なビルドについては、リポジトリに入れて管理しておくと良いでしょう。古いビルドを保存しておく価値については、3-14節を参照してください。

**ビルドも入れるべきなんじゃないの？**

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 Servlet Page | Javaサーブレットファイル |
| .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 | Hyper Text Markup Language | Htmlファイル |
| .js | Java Script file | Java Scriptソースファイル |
| .properties | Properties file | Javaのプロパティ (設定) 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 Hedar 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を記述したファイル |
| .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 |
| .class | Class | Javaソースファイルを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 | 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 (とは英語で壺とか瓶という意味) |
| .war | Web Archive | Webアプリケーションサーバに配置すると、Webアプリケーションとして実行可能 |

# 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 ですね)。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)、符号 (化) という意味のほかにも暗号 (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。  **これはこれでいいのですが、プログラムと、ここでいっているコードとの関係をもう少しコメントしてあげると良いかと思います。**  コードを実行するには、これをPCが直接実行できる形に変換する必要があります。この変換を行うソフトウェアをコンパイラといい、変換する処理を指してコンパイルといいます。もしプログラムが文法的に間違っていれば、コンパイル時にエラーとなりコンパイルできません。  **本文の説明と重複している**  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、実行時にエラーとなります。これに対して、コンパイラ言語はコンパイル時に文法エラーを検出できます。そのため、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、安全で高速なプログラムが書けます。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、本書では割愛します。 |

# 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[[20]](#footnote-20)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。

## 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。これにより、そのファイルをコミットしてもビルドマシンでビルドブレークしないことを担保します。

## 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手順をスモークテストとして準備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。

**？？　コンパイル → レビュー → ビルド → スモークテスト → 修正 → レビュー　・・・ じゃないの？**

## 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[[21]](#footnote-21)。機能ごとの担当者 (フィーチャーオーナー) や、ブランチオーナーをレビューアとして指名します。コミッタは、レビューしてほしいファイルをメールに添付するなどして、レビューアに渡します。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。この手順を、レビューアからOKがもらえるまで繰り返します。

**通常、こんなことしない**

**やっぱり基本はFaceToFace**

コミットしたい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、コーディング規約に合致しているかどうかもレビューの対象になります。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。もし、コンパイルが通らないソースファイルをレビュー依頼すると、レビューアにがっかりされてしまうので気をつけましょう。

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。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、念のためコミットの直前に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ましょう。

**前に示した手順と矛盾している**

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)  **？？　コードレビューはバグFix単位？**  **基本、プログラム単位じゃないの？**  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 (した人)  Affected area: 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 |

また、コミット前のレビューを支援するツールを活用することもできます。このようなツールには、Review BoardやCodeStriker, Crucible, CodeCollaboratorなどがあります。コミット前のレビューをチームに義務付けたいなら、導入を検討しましょう。これらのツールは、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) のレビューに使えるものもあります。

## 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、スモークテストでも問題が出なければ、その変更は安心してコミットできると考えて良いでしょう。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。

最近は、バディビルドをしてくれるビルドサーバ[[22]](#footnote-22)もあります。コミットしたいファイルを添付してビルドサーバにバディビルドを依頼すると、ビルドサーバがクリーンなサンドボックスにその差分を適用し、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pre-tested commitなどといいます。便利な世の中になったものです。

## 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) 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。それだけ、コミットコメントは、コードの変更を追跡するために重要なドキュメントなのです。

**プログラムに対する変更履歴のコメントに関するガイドは？ ... 後から出てくるのかもしれませんが**

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勧めます。特に、コミットコメントにはバグ報告票の番号を含めておくことが重要です。バグ報告票には、より詳細なバグの再現手順やバグの原因などが書いてあるからです。

**これに関連して、ある段階からは、バグとか明示的に指示された修正を除いては、コードは自由に変更できなることは説明しておいた方が良いと思います。**

修正をコミットすると、SCMによりそのチェンジセットには番号がつけられ、「コミットに成功しました。チェンジセット#は314です。」のようなメッセージが表示されます。対応するバグ報告票を「解決済み」とするとき、この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) が連携して、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ものもあります。

多くの SCMは複数のファイルにまたがった修正をまとめて、アトミックにコミットできます。このように、タスクに関連づけて複数のファイルを意味のある単位でまとめてコミットすることをTask Level Commitといいます。

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。ただし、これらを厳格に運用し過ぎると、開発者は頻繁なコミットを嫌がるようになります。特に、開発の初期には表3 - 4に示したコードレビュー依頼票は煩雑すぎます。これよりも簡潔な書式を使った方が、頻繁にコミットしやすい雰囲気を作りやすく、使いやすいでしょう。また、コミット前のコードレビューを義務付ける代わりに、ペアプログラミングをする方がいい場合もあります[[23]](#footnote-23)。逆に、安定したリリース済みの製品に対しては、気軽にコミットできない方が安全ですから、表3 - 4のようなレビュー依頼票をきちんと適用すべきです。プロジェクトの状況に応じて、最適な運用を心がけましょう。ただし、必要なときは、決めたプロセスにこだわりすぎないようにしてください。プロセスに従うよりも、結果を出すことの方が大事です。

**だったら現実的な例も示した方がいいのでは？**

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。ソフトウェア開発がある段階を過ぎ、タスクレベルコミットができる状態にまでビルドが育ってきたら、コードを自由に変更してコミットすることは禁止すべきです。それ以降は、バグの修正や機能の追加などの作業を指示された担当者だけが、コードの修正とコミットを許されます。この作業は、バグ追跡システム (BTS) のバグ報告票が担当者に割り当てられることで指示されます[[24]](#footnote-24)。また、そのような段階になったら、すべてのコミットに上記で示したようなコードレビューを義務づけると良いでしょう。もし、コミットが多すぎてすべてをレビューし切れないときは、次のような文言のいずれかをレビュー依頼票の先頭に書くなどして、運用を工夫します。

* 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。

開発が初期のころ (最初のいくつかのビルドをQAリリースするまで) は、タスクレベルコミットや、コミット前のコードレビューをチームに義務づけるのは難しいものです。このようなときは、上記に示したようにコミットの単位をレビューするよりも、プログラムを紙に印刷して、プログラム単位をでレビューすると良いでしょう。あるいは、ペアプログラミングをした方がいいかもしれません[[25]](#footnote-25)。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。ただし、必要なときは、決めたプロセスにこだわりすぎないようにしてください。プロセスに従うよりも、結果を出すことの方が大切です。



図3 - 4 コミットの手順

# 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**  アトミックとは、分割できないということです。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分割できない最小のコミットの単位です。例えば、あるバグを治すために、5つのファイルを同時に修正してコミットしたいとします。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、間違って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しているSCMなら、何らかの原因でいくつかのファイルがコミットできなかったときは、すべてのファイルのコミットに失敗するようになっています。つまり、必ずすべてのファイルのコミットが成功するか、すべてのファイルのコミットが失敗するか、どちらかになります。SCMソフトウェア以外にも、データベースシステムなどでは、アトミックな操作のサポートが、データの整合性を守るために非常に重要です。  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。 |

# 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、単純な作業ミスにより、時には不正なコミットをしてしまうことがあります。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、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。文字通り、ビルドが壊れるという意味ですね[[26]](#footnote-26)。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。

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、あなたがリポジトリの汚染に気づいたら、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などでこれを開発チームに通知する。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する。このとき、ビルドエラー (コンパイルエラー) をメールに添付して、心当たりのある人 (犯人) に出頭を促す。

**原因/心あたりを特定する役割の人は？　またその作業**

**主語は？**

2. ビルドブレークの知らせを受けた開発者らは、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する。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コンパイルエラーが出てビルドができなくなり、開発作業がストップしてしまう。

**文章！**

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する。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する。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よい。

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



図3 - 5 ビルドブレークを解消する手順

比較的大勢の開発者で並行開発しているときや、リポジトリの衛生状態がやや悪くなっているとき、このようなことが起こりやすくなります。ここでは、やらかした人を犯人と呼びましたが、悪気があってしたのではないのですから、あまり犯人を責めるべきではないでしょう。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[[27]](#footnote-27)。また、最初にビルドブレークに気づいた人は、1.のステップでチームに怒りのメールを出すこともありますが、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。気を付けましょう。

|  |
| --- |
| 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。リポジトリが汚染されていることを知らずに更新の操作をして、それを自分のサンドボックスに導入してしまうと面倒やっかいです。サンドボックスでビルドが通らなくなると、開発作業ができなくなってしまうからです。しかし、3-4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、時にはリポジトリの汚染をサンドボックスにうつしてしまうこと (そのリポジトリの汚染に、チーム内であなたが最初に気づくこと) もあるでしょう。これを予防するには、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、まずサンドボックスBで更新をし、ビルドを通します。ビルドが通れば、作業をサンドボックスBに移すことができます。もしビルドブレークしたら、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、リポジトリが回復するのを待てます。

# 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突き止めることができます。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。

**前提として、こういった状態にならないようにしなければならない**



図3 - 6 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 - 6は、この手順を最後まで繰り返した様子です。この結果は次のようになりました。

build#16 再現しない

build#18 再現しない

build#19 再現しない

build#20 再現した

build#24 再現した

build#32 再現した

つまり、このバグはbuild#20で混入したことが分かります。さらに、build#20に入れた修正の一覧をSCMから取り出して調べれば、その中のどれが当該のバグの原因なのか、突き止められることでしょう。二分探索を使えば、これまでのビルド(上記の例では32個のビルド) すべてをテストしなくても、そのバグがどのビルドから混入したかを見つけ出せます。

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、前述のように、ビルドは一般にリポジトリの中に入れて管理することはしません。古いビルドは、ファイルサーバに日付フォルダを作って保存しておくので十分だからです。

ただし、前提として、バグが入り込んだらすぐにそれに気づけるようになっているべきです。もし、ずいぶん前に入ったバグにしばらく気が付かなかったとしたら、それは何かの危険信号です。その原因を追究しておきましょう。

# タグを付ける

SCMソフトウェア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、ビルドのインプットとなるソースファイルの履歴を管理するわけですが、それだけでは過去のビルドを再現することはできません。どのビルドが、どの時点 (リビジョン#) のソースからビルドされたのかを記録しておく必要があります。例えば、build#39のビルドは何年何月何日の何時何分何秒のソースコード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、ある時点のソースファイル一式に任意の印をつける機能があります。この印をタグといい、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[[28]](#footnote-28)。

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

上記では、タグの書式を<製品名>\_<日付>\_<ビルド番号>としています。タグにはどんな名前を付けても良いのですが、書式は上記のようにブランチを通じて統一しておくと使いやすくなります。

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。これまでに付けられたタグの一覧を取得することもできます。

ブランチと、ある日時を指定すると、その時点におけるブランチ上のファイルセットにタグを付けることができます。あるいは、現在サンドボックスに取得しているファイルセットを使って、タグを付けたいファイルセットを指定することもできます。このように、タグは任意の時点のファイルに対して付けることができます。これに対し、ファイルのコミットは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。ファイルをコミットすると、それは必ずそのファイルの最新リビジョンの次の時点にコミットされます。



図3 - 7 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[[29]](#footnote-29)

# 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。ファイル一式をディレクトリごとリポジトリ内でコピーすれば、製品を枝分かれさせることができるわけです。タグと同様に、任意の時点からブランチを分けることができます。

ただし、優れたSCMソフトウェアは、サブブランチを作成してもリポジトリ内でファイルセットを物理的にコピーすることはしないので、ディスク容量を無駄に消費することはありません。





図3 - 8 サブブランチを切ったときのディレクトリ構成の例

SCMソフトウェア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 - 6に示します。リポジトリ内では物理的にコピーされてはいませんが、これをサンドボックス側に取り出すと、図3 - 6のように別のディレクトリツリーとして扱えます。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、それにつれて異なる内容に変化していきます。

# ブランチとビルド番号

ブランチを分けた場合、それぞれのブランチからリリースするビルドには、どのようなビルド番号とバージョン番号をつけるべきでしょうか。それぞれのビルドを区別できれば、どのようにしても構わないのですが、ビルド番号は、それぞれのブランチごとに通しの番号をつけることを勧めます。図3 - 7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るからです。



図3 - 9 間違いやすいビルド番号の付け方

それぞれのブランチごとの通し番号をつけると、別のブランチのビルドに同じビルド番号がついてしまいます。そこで、何らかの工夫をして、別ブランチのビルドを区別します。

図3 - 10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るのでとても便利です。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 - 10 百の位にブランチ識別番号を割り当てた例

図3 - 11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。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るというデメリットがあります。



図3 - 11 マイナーバージョン番号でブランチを識別する例

このように、各ブランチにバージョン情報を付与する方法はいろいろあり、それぞれメリットとデメリットがあります。どの方法を選択するにせよ、バージョン情報の書式ポリシーを決めたら、それを一貫して使い続けることが大事です。逆にいえば、一貫して使い続けることのできる書式ポリシーを最初に選択しておく必要があるということです。

# ブランチ間で差分をやりとりする

前述の通り、サブブランチは元のブランチにあるファイル一式をコピーすることで作成されます。このため、もし元のブランチ上のプログラムにバグがあれば、サブブランチの同じプログラムにも同じバグがあることになります。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、同じバグを修正する必要があります。このように、あるブランチにコミットした差分を、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。あるバグを修正するために複数のファイルを修正したら、サブブランチ上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、そのすべてのブランチに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ソフトウェアは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をサンドボックスに更新した上で、次の2つのパラメータをSCMのポート命令に渡します[[30]](#footnote-30)。

* 元ブランチの名前
* 元ブランチ上にある、ポートしたい差分のセット (チェンジセット)

チェンジセットとは、コミットの単位となる差分の集まりのことです。チェンジセットは、それがコミットされた日時の範囲や、コミットの単位ごとにつく番号 (チェンジセット番号) などで指定できます。ポートの操作をすると、サンドボックス上にあるポート先ブランチのファイルに、当該の差分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リポジトリにコミットします。下記に、サブブランチから元ブランチへポート (バックポート) する手順を示します。

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. コミットすれば、ポートが完了



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、マージを自動で行うことができず、「競合」という状態になります。このときは、人が手作業でサンドボックス上のファイルを編集し、競合状態を解決 (適切にマージ) してからコミットする必要があります。また、プロジェクトによっては、自動でマージされたファイルにも、コミットの前にコードレビューを義務付ける場合があります。

# バックポートとフォワードポート

図3 - 12に示すように、サブブランチから元のブランチへポートするとき、これをバックポートといいます。この逆向きの方向で行うポートは、フォワードポートといいます。

 

図3 - 12 バックポート 図 3 – 12’ フォワードポート

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。また、複数のコミットをまとめてバックポートすることをリバースインテグレーション、複数のコミットをまとめてフォワードポートすることをフォワードインテグレーションなどということもあります。このほか、親子ブランチの間だけでなく、兄弟ブランチ (共通の親ブランチをもつ別のブランチ) の間でもポートできるSCMもあります。



図3 - 13 バルクポート

ブランチは、コードの差分が積み重なるようにして伸びていきますが、多くの場合、この積み重なる順序は重要ではありません。このため、あるブランチから別のブランチへポートするとき、ある差分 (バグ修正など) だけをポートして、ほかの差分 (機能追加など) はポートせずにおくことができます。また、一度のコミットに含まれる差分 (チェンジセット) には、複数のファイルの変更が含まれることがありますが、これを全てポートすることも、そのチェンジセットに含まれる一部のファイルの差分だけをポートすることもできます。

**？？**

# 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[[31]](#footnote-31)。



図3 - 14 はしごマージ

しかし、このようなポートが苦手なSCMソフトウェアもあります。そのようなSCMは、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ソフトウェアは、どの差分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、上記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ソフトウェア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は商用のソフトウェアですが、十分に枯れており、また大規模なファイルを大量にリポジトリに入れてもきびきびと動作する信頼性の高いSCMです。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[[32]](#footnote-32)。

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ソフトウェアを使うときは、双方向や間接的なポートは避けた方が安全です。すべてのブランチに必要な修正は、必ずまずメインラインに入れ、それを各サブブランチへフォワードポートするようにします。

# 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、答えは聖徳太子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担当者 (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、デブリードらが話し合って決めます事前に戦略的に計画します。その判断と指示を受けて、SCMの操作に熟練した構成管理担当者がブランチを切る操作をします。

**骨格は戦略的・計画的に事前に決めておくべき**

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[[33]](#footnote-33)、新人エンジニアのみなさんは、「今の自分は聖徳太子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[[34]](#footnote-34)。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作業が発生します。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。

**ポート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なんじゃないですか？**

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。なっておくべきです。以下にブランチングの戦略を解説しますので、これをよく読み、ぜひ熟練の大工さん、さらには聖徳太子になるための資質を養ってください。

# 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です。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ることができます。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  ・Git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

では、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コードは、メインラインにバックポートしておくことを忘れないでください。

## 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たくさん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わけにはいきません。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、リリースします。こうすれば、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ることができます。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、大事な製品を不安定にしてしまいます。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ポートします。多くのSCMは、複数の差分をすべて一度にポートし、コミットする機能を備えています。このようにすれば、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ポートする

## メインラインを汚さずに、新しい意欲的な機能を試験的に実装したいとき

あなたは、大事に育ててきた製品をリポジトリ上に持っています。ここで、その製品のアーキテクチャをアグレッシブに作り変え、より強力にバージョンアップする計画が持ち上がりました。しかし、その変更はあまりにアグレッシブなので、失敗すると大事な製品を死滅においやりかねません。そこで、安定した製品をメインラインに残したまま、開発用のブランチを切ってそちらで試験的に実装します。ドラスティックな変更がうまくいかなかったときは、単純にこのブランチを破棄します。うまく開発でき、このラインで作ったビルドが安定した暁には、開発ブランチの根元からコミットしたすべてのコードの差分をメインラインにポートしてから、このブランチを破棄します[[35]](#footnote-35)。このようなブランチを実験用ライン (experimental line) といいます。実験用ラインは、開発用ラインとは異なり、実験が終われば破棄されます。

問題: メインラインを危険にさらすことなく、実験的な実装を試したい

解決: 別のブランチで実験的な実装を試し、うまくいったらこれをメインラインにポートする

 

## 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、このブランチで行ったすべての修正をメインラインにマージし、このブランチを放棄します。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。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。また、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と思います。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、製品に大掛かりな変更を加えたい

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

 

## 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わからなくなっ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[[36]](#footnote-36)。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。

**もっと前のところになると思いますが、ベースラインの概念や、ベースラインと構成の関係とかも、CMであれば、本来は説明しておくべきだとは思います。**

**＋　コードに特化しすぎ/コードしか見ていないような感じ**

**これはブランチというより、ベースラインを分ける/リリース単位の分解としてとらえた方が分かりやすいんじゃないの？**

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

ここまでに紹介したさまざまな事情で、複数のサブブランチに入れた複数のコミットをまとめて元のブランチにポートし、サブブランチは破棄したいとしましょう。しかし、このサブブランチを元のブランチから分けたのは、けっこう前のことです。そのため、サブブランチを分けた後に、メインラインにも多くのコミットがされています。サンドボックス上でポートの操作をすると、かなりの数のファイルが競合してしまいました。手作業でマージしましたが、これらのファイルをいきなり元のブランチにコミットするのは危険です。このようなときは、ポート先の最新のリビジョンからブランチを切り、このブランチにポートをします。追加で必要な修正を入れ、このブランチのビルドの動作が安定したところで、元のブランチに修正をポートします。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。

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: 統合ブランチにポートして、ビルドとテストをしてからメインラインへマージ

 

|  |
| --- |
| **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手を入れるべきではないと言われていました。というのは、ソフトウェアは非常に壊れやすいからです。せっかく動いているのに、これに手を入れてしまえば、動いているものを動かなくしてしまうことにもなりかねません。しかし、「動いているソフトウェアにわざわざ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、きれいに書き直す機会を得ることはありません。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。1章の★図1-2における「安定化」に相当する作業であると考えられます。  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をもつことができます。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。リファクタリングはもちろん、バグの修正も我慢すべきです。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。特に、リファクタリングのようなソフトウェアの内部構造の変更は、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非常に難しいと考えるべきでしょう。 |

# メインラインモデルとプロモーションモデル

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、プロモーションモデルといいます。



図3 - 15 プロモーションモデル

Promotionとは昇格とか昇進という意味で、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ということです。このモデルを使うのはお勧めしません。もしこのモデルを使うなら、現在のメインラインがどのブランチなのか、いつも明確に意識して作業してください。さもなければ、ここで示したシナリオのように、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いくので要注意です。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、私もブランチを多く切ったあげく、ポート作業を怠って、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 - 16 メインラインモデル

メインラインモデルでは、保守のためにブランチを切っても、保守ブランチにコミットした修正で必要なものは必ずメインラインにバックポートし、次のバージョンの製品はメインラインもしくは新しくメインラインから切ったサブブランチからリリースします。このとき、以前のバージョンの保守ブランチは凍結して誰もコミットできないようにし、これを明確に破棄します。これにより、製品の進化のための最上のブランチを一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、次の2つのプラクティスに従ってください。

* サブブランチでした修正で必要なものは、必ずメインラインにバックポートすること。
* ブランチを切るときは、必ずメインラインからにすること。

この2つのプラクティスの間には、深い関係があります。なぜそうなのか、考えてみてください。ブランチに対する理解が深まるはずです。

# 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活用すべきかを紹介してきました。本節では、ブランチを使うための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コードラインポリシー (codeline policy) といいます。

## 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つけてブランチを切りましょう。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ね。

## ブランチングを遅らせる

例えば3-21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、必要な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

前の段落と相反するようですが、例えば3-21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、次のバージョンのための作業がすぐに始められるようになります。

## フリーズよりもブランチを使う

SCM ソフトウェア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、タグを打ち、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

後でマージ (ポート) をしなければならないことがわかっている修正は、なるべくすぐ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、ポートしたかった修正が競合してマージできなくなることがあります。

## 複数ブランチに共通の修正は、枝の根元に近いブランチから

複数のブランチにまたがる同じバグを修正したいときは、一般に、枝分かれした時点からのコミットが最も少ないブランチを最初に修正すべきです。その方が、その差分をもう一方のブランチにポートするとき、競合してしまう危険が小さくなります。例えば、3-21節の保守ブランチの例では、先に保守ブランチを修正し、それをメインラインにポートする方が安全です[[37]](#footnote-37)。

## 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とっておいてくれる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。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、専用のリポジトリを準備しておくと役立ちます。SCMソフトウェアの操作に慣れたり、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[[38]](#footnote-38)) 開発チームからしばらく抜けた際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[[39]](#footnote-39)。

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、この人が責任を担うとする方が良いでしょう。ブランチオーナーは、コードレビューを担当したり、トリアージに参加したりするなどして、当該のブランチが壊れないようにするゲートキーパー (門番) の役割を果たします。

**微妙**

## 作業を分けて、人は分けない

ブランチを分けることで、別の作業を独立して行えるようになります。しかし、これは人と人とのコミュニケーションを妨げるものでも、不要にするものでもありません。ブランチを分けても、人と人は分けないようにしましょう。

表3 - 5 よく使われるコードラインポリシー

|  |  |  |
| --- | --- | --- |
| 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 |
| 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 |
| 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 |
| 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 | 実験的な実装を試すためのライン | 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 |
| 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 | variant line |
| Remote development 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 |

# ポートを忘れないようにする

保守ブランチにバグ修正しかコミットしていないときは、この保守ブランチからメインラインへのポートを忘れてしまうことはあまりありません。その保守ブランチに入れたバグ修正は、後でまとめて一度に全部メインラインにポートすれば良いからです。しかし、ユーザーブランチが複数あり、それぞれに各ブランチ固有の修正とバグ修正の両方がコミットされているときは、必要なポートを忘れてしまうことがあります。バグ修正の差分だけを取り出して、必要なブランチにポートしなければならないからです。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、どこで治したっけ？」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。

**それ以前に計画的にやらないといけないんじゃないの？**

そこで、ポートが必要と思われる修正をしたら、そのポート先のブランチと同じ数だけ、バグ報告票をBTSに登録しておきましょう。そ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、閉じてくれるはずです。

# リポジトリへのアクセスを制御する

あるブランチに対して、ソースファイルの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です。もし、コードを修正してしまうと、また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、コードフリーズをした後になって細かいバグを見つけて、こっそり治してコミットしてしまう開発者がいたりします。もちろん、リグレッションの危険があるため、許されることではありません。もし深刻なバグが見つかれば、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[[40]](#footnote-40)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、ビルドマシンでビルド中にコミットされると正しくビルドできなくなる場合があるので、一時的にビルド中だけコミットを禁止したいこともあります。

このほか、開発チームにはソースファイルの変更 (コミット) を許すが、QAチームには閲覧しか許さないとか[[41]](#footnote-41)、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。この機能を使えば、あるブランチに対してコミットを物理的に禁止できます。また、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、といった機能をもつものもあります。お使いのSCMソフトウェア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ソフトウェアを検討する必要があるかもしれません。

# SCM サーバの管理

SCMサーバ (リポジトリ) は、各プロジェクトにおいてSCMエンジニアのロールを担当する人が管理・運用します。また、各ブランチはSCMエンジニア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チームの誰かが兼務します)。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。

**何を管理するのか？　どう管理するのか？　は言わないのですか？**



図3 - 17 社内ITが提供するサービスの例

# まとめ

表3 - 6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ソフトウェアで使われるコマンドも、各項目に添付しました。SCMソフトウェアによっては、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ものもあります。

表3 - 6 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, client workspace, view, 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# |
| 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 |
| タグ | スナップショットにつけることのできる目印 (文字列) 。 | 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 | 同じリビジョン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ソフトウェア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 |
| 追加 | リポジトリにファイルを追加する。 | 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。 | update, sync, checkout, refresh, get |
| 編集 | サンドボックス内のあるファイルに、修正中であると印をつける。このような機能がないと、サンドボックスでどのファルを修正したか、分からなくなってしまう。 | 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する。 | commit, submit, checkin, put, keep |
| リバート | サンドボックスでの修正を取り消し、リポジトリの最新版の状態に戻す。リポジトリでの修正をリバートできるSCMソフトウェアもある。 | 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。 | 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/>

## 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としました。 [↑](#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. 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-4)
5. 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。 [↑](#footnote-ref-5)
6. サンドボックスは、必ず特定のブランチに関連づけられます。最初にサンドボックスを作成するときに、作業対象とするブランチを指定します。 [↑](#footnote-ref-6)
7. SCMソフトウェアによって、リビジョン番号の増え方が違うので注意して下さい。CVSのように、ファイルごとにそれぞれ別のリビジョン番号がひとつ増えるものもあれば、Subversionのように、コミットの操作によりリポジトリ内の全てのファイルの共通のリビジョン番号がひとつずつ増えるものもあります。一般に、後者の方がリビジョン番号でスナップショットを指定できるので便利です。 [↑](#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ね。 [↑](#footnote-ref-8)
9. 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-9)
10. 例えば、ねこちゃんがお砂箱にウンコしたら、支障が出ますよね。 [↑](#footnote-ref-10)
11. プラクティスとは、守るべき習慣や作法のことです。 [↑](#footnote-ref-11)
12. 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。エラーを除去して、ビルドができるようにすることを「ビルドを通す」といいます。 [↑](#footnote-ref-12)
13. 修正したファイルをコミットするには、チームのレビューと承認が必要です。これについては後述します。 [↑](#footnote-ref-13)
14. このほか、SCMとは関係なく、エディタで編集中だがまだ保存していないファイルを指してもdirtyということがあります。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。あんまりです。 [↑](#footnote-ref-14)
15. 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。XP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。氏は、統合テストを支援するフレームワークFITの開発者としても知られています。 [↑](#footnote-ref-15)
16. 複数のブランチを、ひとつのサンドボックスに対応づけることのできるSCMソフトウェアもあります。また、あるサンドボックスを、後で別のブランチ (親子関係や兄弟間系にあるブランチ) に対応づけることができるSCMソフトウェアもあります。 [↑](#footnote-ref-16)
17. 私が一番好きなギタリスト、Pat Methenyの言葉が引用されています。 [↑](#footnote-ref-17)
18. 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章で扱います。 [↑](#footnote-ref-18)
19. 詳細なビルドの手順は4章で説明します。 [↑](#footnote-ref-19)
20. 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適用することです。ポートについては後述します。 [↑](#footnote-ref-20)
21. 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、コミッタ (コミットをする人) で統一しました。 [↑](#footnote-ref-21)
22. 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-22)
23. ペアプログラミングとは、XP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-23)
24. BTSについては、5章で詳しく扱います。 [↑](#footnote-ref-24)
25. ペアプログラミングとは、XP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-25)
26. ★4-5節では、このほかのビルドの壊れ方を紹介します。 [↑](#footnote-ref-26)
27. ビルドビ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いいます。 [↑](#footnote-ref-27)
28. 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-28)
29. @マークはatの略記です。日付や時刻、場所などの記入に日常的に使われる記号です。@マークについてもっと知りたい方は、Wikipedia日本語版の「単価記号」をどうぞ。 [↑](#footnote-ref-29)
30. コマンドの詳細は、SCMソフトウェア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-30)
31. 私は、これを「あみだマージ」と呼んでいます。 [↑](#footnote-ref-31)
32. May the perForce be with you. (ぱフォースと共にあらんことを。) ― unknown [↑](#footnote-ref-32)
33. 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) ブランチを切ることがあります。 [↑](#footnote-ref-33)
34. もちろん、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-34)
35. 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異なる操作なので、区別して覚えましょう。 [↑](#footnote-ref-35)
36. 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-36)
37. ただし、SCMソフトウェアによっては、最適なポートの戦略が異なることがあります。詳細は、お使いのSCMのドキュメントを参照してください。3-19節も参考にしてください。 [↑](#footnote-ref-37)
38. メンバーがバスに轢かれたためにプロジェクトを続行できなくなるとき、そのバスに轢かれた人数をバスナンバー (bus number) といいます。また、メンバーがトラックに轢かれる数をトラックナンバー (truck number)、新婚旅行にでかけていなくなる人数をハネムーンナンバー (honeymoon number) といいます。もちろん、これらの数は大きいほど好ましいものです。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。 [↑](#footnote-ref-38)
39. XP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-39)
40. 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-40)
41. もちろん、QAチームが統合ビルドのテストオートメーションの開発を担当するときは、QAにもソースをコミットする権限が必要です。 [↑](#footnote-ref-41)