複数タイトルにまたがる効率的なフレームワーク管理

– 分散 SCM の活用 –

2014 年 2 月 27 日 初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 2 月 27 日	板垣 衛	（初稿）

■ 目次

■ 概略 1

■ 目的 1

■ 対象 1

■ 本書の内容について 2

▼ 分散 SCM の活用 2

▼ 分散 SCM について 2

■ フレームワークの基本要件 2

■ フレームワークのスタイル 3

▼ 共通ライブラリは基本的にソースコードで提供 3

- プロジェクト主導によるライブラリの改変を許容 3
- ライブラリ改変の安全性 3
- プロジェクト向けブランチの活用 4
- 「亜種」ブランチを許容しない 4
- 改変ではなく追加が必要なら一旦中間ライブラリに 4
- 他のプロジェクトへの反映 5
- 「機能ブランチ」の活用 5
- 根気強く共通ライブラリを育成 6

▼ 機密性の高いライブラリとの区別 6

- 複数のライブラリ 6

▼ 特定プロジェクト向けのライブラリのカスタマイズ 7

- マクロによるコンパイラスイッチの活用 7
- コンパイラスイッチの方針①：プロジェクトの判別を禁止 7
- コンパイラスイッチの方針②：プロジェクトで機能の選択 7
- 所定のコンパイラスイッチ定義ファイル 8
- コンパイラスイッチ用ブランチ 8
- コンパイラスイッチテンプレート 8
- 既定のライブラリ配置 8
- 強制インクルード設定 8
- プリコンパイル済みヘッダー 9

▼ ライブラリのサブモジュール化 9

- 「サブモジュール」について 9

- 容易なプロジェクト配布 9
- 明示的な「サブモジュールの更新」 10

▼ 分散 SCM の注意点 10

- 【扱えないこと】シーケンシャルなリビジョン番号 10
- 【できないこと】ファイルのロック 10
- 【できないこと】アクセス権設定 11

■ 分散 SCM を用いた開発の特徴 11

▼ 安全な途中コミット 12

▼ サーバー障害に強い 12

▼ 障害復旧に強い 12

▼ 手軽なブランチ操作 12

- トピックブランチ 12
- ブランチの切り替え 13
- トピックブランチの提出 13
- 機能ブランチ 13
- 頻繁なブランチ切り替え 14

▼ 止めないコミット 14

▼ 【問題点】複雑な操作 14

- 技術サポート 14
- 技術サポートスタッフに対する要件 14

■ 分散 SCM を用いた開発ワークフロー 16

▼ 共有リポジトリの作成 16

▼ 共有リポジトリへのアクセス設定 16

▼ クローン 17

▼ フェッチ／プル 17

- 「フェッチ」操作 17
- 「プル」操作 17

▼ ブランチの切り替え（チェックアウト） 18

- 「チェックアウト」のイメージ 18

▼ トピックブランチ／機能ブランチの作成 18

- ブランチの命名規則 18

▼ サブモジュールの更新（アップデート） 18

- サブモジュールの更新タイミングについて 18
- サブモジュールの状態について 19

▼ サブモジュールの追加 19

- 追加したサブモジュールのコミット 19

▼ コミット 19

▼ プッシュ 19

- 競合の発生 19
- マージ 20
- トピックブランチのプッシュ 20
- トピックブランチの提出の前に 20

▼ マージリクエスト 20

▼ マージ 20

- 【注意】マージコミット 20
- 【注意】トピックブランチマージ時のファストフォワードマージの禁止 20
- ファストフォワードマージを使用する場面 21
- 【注意】トピックブランチマージ時のスカッシュマージの禁止 21
- スカッシュマージを使用する場面とチェリーピックマージ 21
- トピックブランチの受け入れ拒否 21

▼ 遠隔地での開発 22

■ ホスティングサービスを使ったワークフロー 22

▼ ホスティングサービスについて 22

- 商用ホスティングサービス（クラウド） 22
- エンタープライズ版 23
- 無償版 23

▼ ユーザー管理 23

▼ フォーク 23

▼ プルリクエスト 23

▼ 通信セキュリティ 24

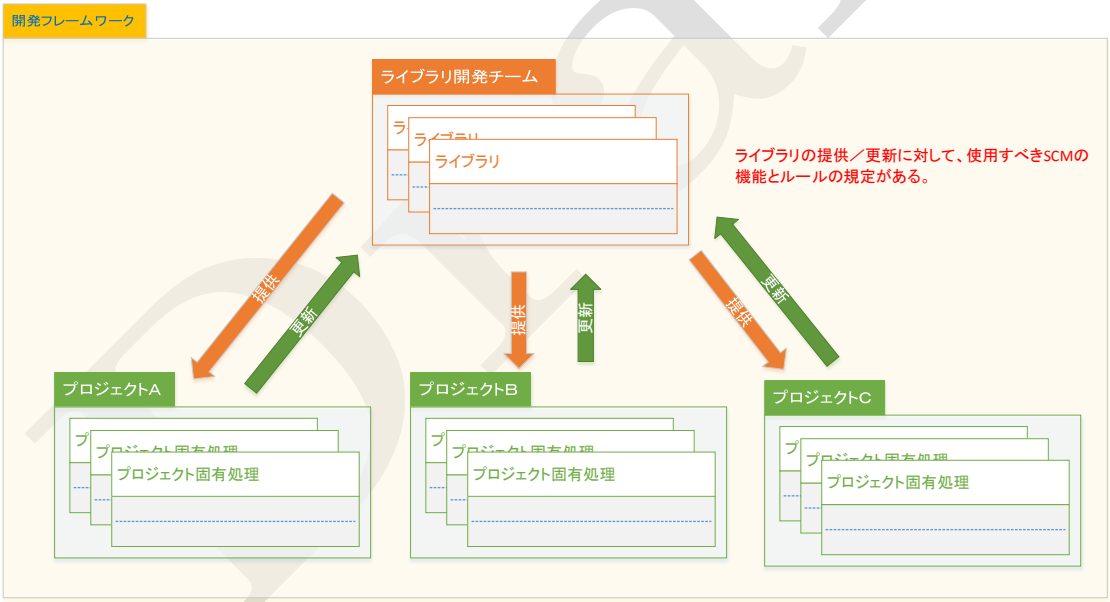
■ 概略

分散 SCM（Software Configuration Management = ソフトウェア構成管理）を活用することによる効率的な開発フレームワークの管理手法について説明する。

■ 目的

本書は、開発プロジェクトの進行をスムーズにし、かつ、高品質な開発を行うための環境を構築することを目的とする。

複数プロジェクトにまたがって共通利用するライブラリの管理手法を確立し、それを利用するプロジェクト全体を一つのフレームワークとみなし、共通ルールのもとに安全に相互発展する開発環境を構築する。



■ 対象

本件は、「プログラム」の開発に特化した内容である。
グラフィックやサウンドなどのコンテンツの作成は対象に含まない。

■ 本書の内容について

本書は、分散 SCM の「Git」の機能に依存した環境を提案するが、具体的な Git のコマンド操作まで説明するものではなく、要件を満たすために活用する機能を説明する程度にとどめるものとする。

▼ 分散 SCM の活用

本書は、分散 SCM の強力な「ブランチ」機能や、複数のリポジトリを統合管理する「サブモジュール」機能などの活用を前提する。

開発作業の際には頻繁なブランチ切り替えが必要となる。Subversion で同様のブランチ活用を行おうとすると非常に手間となることが、分散 SCM では手軽なものとして扱うことができる。

▼ 分散 SCM について

本書は、Git の使用を前提とした管理手法を示すが、類似の分散 SCM である「Mercurial」、「Bazaar」でもおそらく同様の管理が可能である。

■ フレームワークの基本要件

まず、開発効率を最適化するためのフレームワークの基本要件を明確にする。

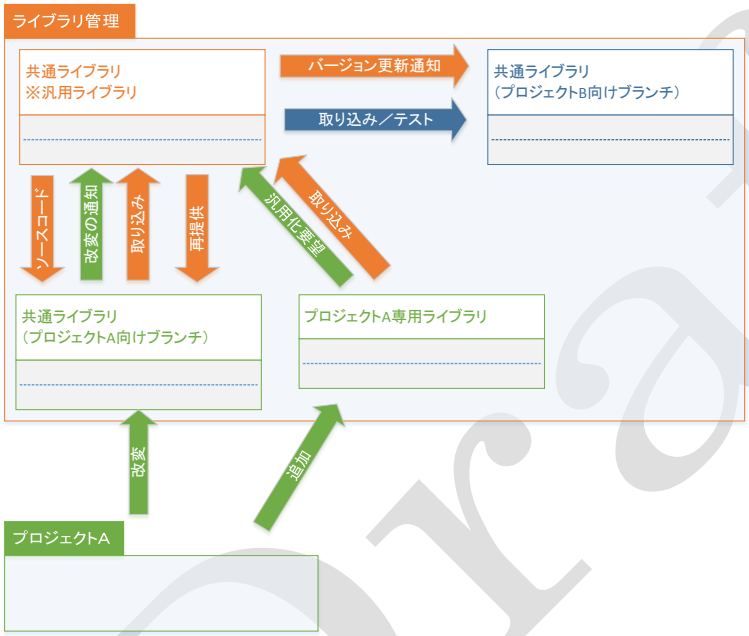
- 複数プロジェクトにまたがる共通ライブラリを扱うものとする。
- 共通ライブラリは、ライブラリ開発者からの一方的な提供ではなく、特定のプロジェクトで発生した要件を適切に取り込んで相互発展し、他のプロジェクトもその恩恵を受ける事ができるものとする。
- 特定のプロジェクトからの要件に基づく改変が、他のプロジェクトに悪影響を及ぼさないことを保証するものとする。
- 上記の要件を満たしつつ、必要に応じて、特定のプロジェクト主導でライブラリに改変を加えることを可能とする。
  - 「ライブラリ修正待ち」のような停滞期間を作らず、プロジェクトの開発をスムーズに進行できるものとする。

■ フレームワークのスタイル

基本要件を踏まえて、フレームワークのスタイルを確立する。

▼ 共通ライブラリは基本的にソースコードで提供

共通ライブラリは、ライブラリ形式（.lib / .a）ではなく、ソースコードで各プロジェクトに配布することを基本とする。



● プロジェクト主導によるライブラリの改変を許容

特定プロジェクトの開発中に、ライブラリの問題によって開発に支障をきたした場合、先にプロジェクト側で修正を行って開発を進め、修正内容の事後報告を許容するスタイルとする。

● ライブラリ改変の安全性

ライブラリの改変は、即時他のプロジェクトにまで波及するものではない。

改変を行ったプロジェクト内はコミットも済ませ、チーム内で共有している状態であっても、ライブラリチームが改変内容を受け入れなければ正式に共通ライブラリに反映されないものとする。

- **プロジェクト向けブランチの活用**

特定プロジェクトによるライブラリ改変の影響を局所的なものにするために、ライブラリは各プロジェクト向けのブランチを作成して提供するものとする。

- **「亜種」ブランチを許容しない**

「プロジェクト向けブランチ」は、あくあまでも一方で行われた改変の影響から他方を保護することが目的である。そのままプロジェクトごとにライブラリの内容がバラバラになることは決して許容しないものとする。

ライブラリの変更は必ず全プロジェクトに反映

ライブラリチームが認めた改変は、必ず全プロジェクトに反映するものとする。

汎用性のない改変は却下

ライブラリチームの判断により、汎用性を失うような改変は認められず、ライブラリに取り込まれないものとする。

この時、ライブラリチームとプロジェクトのチームはよく相談し、両者の落としどころを決めて、双方にとって最善の対応を取らなければならない。場合によってはプロジェクトが行った改変と違う形で、要望を満たす機能の提供を行う。結果としてライブラリが改変されたら、全プロジェクトに必ず反映する。

【例外】亜種ブランチを認めるケース

ポストプロダクションフェーズに入ったプロジェクトは、使用するライブラリのバージョンをフィックスするため、以後発生するライブラリの改変は亜種ブランチとして許容されるものとする。

- **改変ではなく追加が必要なら一旦中間ライブラリに**

プロジェクトが必要とするものがライブラリの改変ではなく、全く新しい機能であった場合、プロジェクト専用の共通ライブラリに実装した上で、正式にライブラリに組み込むこととを要望してライブラリチームに通知するものとする。

プロジェクト専用ライブラリの規定

プロジェクトは必ず専用ライブラリを持つものとする。

プロジェクト内の共通処理を扱う目的のほか、汎用ライブラリの候補としてもみなし、中間ライブラリとして位置付ける。

ライブラリチームの判断

ライブラリチームが認めた追加機能は、共通ライブラリに組み込まれ、全プロジェクトに反映されるものとする。

認められなかったものはそのままプロジェクト専用処理として扱われる。

● 他のプロジェクトへの反映

ライブラリチームが認めた改変は、即時他のプロジェクトにまで波及するものではない。

ライブラリのバージョン更新がアナウンスされたあと、各プロジェクトのライブラリ担当者（主にリードプログラマ）が、都合のよいタイミングで、変更内容の確認とテストを行った上で、プロジェクト向けブランチに反映させるものとする。

受け入れの拒絶

最新ライブラリの変更内容が、あるプロジェクトにとって受け入れがたいものである可能性がある。

この場合、ライブラリチームとプロジェクトのチームはよく相談し、両者の落としどころを決めて、双方にとって最善の対応を取らなければならない。場合によっては共通ライブラリの再修正を行わなければならず、その場合、あらためて全プロジェクトに反映し直すものとする。

● 「機能ブランチ」の活用

プロジェクトからライブラリチームへの改変の提出、および、ライブラリチームからプロジェクトへの最新版の反映は、「機能ブランチ」を使用し、他のスタッフに影響を及ぼさない手法で扱うものとする。

機能ブランチについての詳しい説明は後述する。要するに一時作業用のブランチのことである。

● 根気強く共通ライブラリを育成

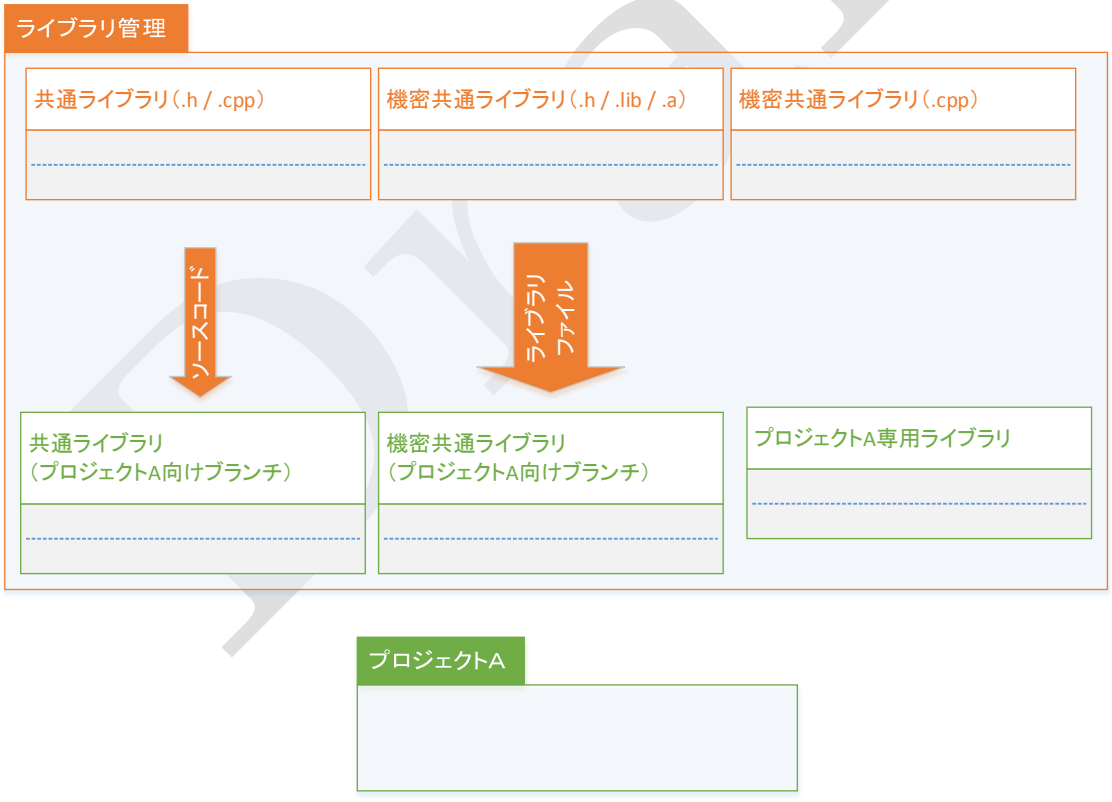
場合によってはプロジェクト間の相性が悪く、双方に適合する形に落ち着かない可能性がある。

その場合、最悪別クラス／関数に分けてしまうといった思い切った対応を選択することも認め、とにかく、安易に「亜種ブランチ」を認めてしまうことだけは絶対にしないものとする。

スタッフは、広範囲に適用可能なライブラリを育成することを常に意識しなければならない。

▼ 機密性の高いライブラリとの区別

暗号化処理など、ソースコードの提供が望ましくない気密性の高いライブラリは、別ライブラリとして扱い、ヘッダーとライブラリファイル（.lib / .a）で提供する。

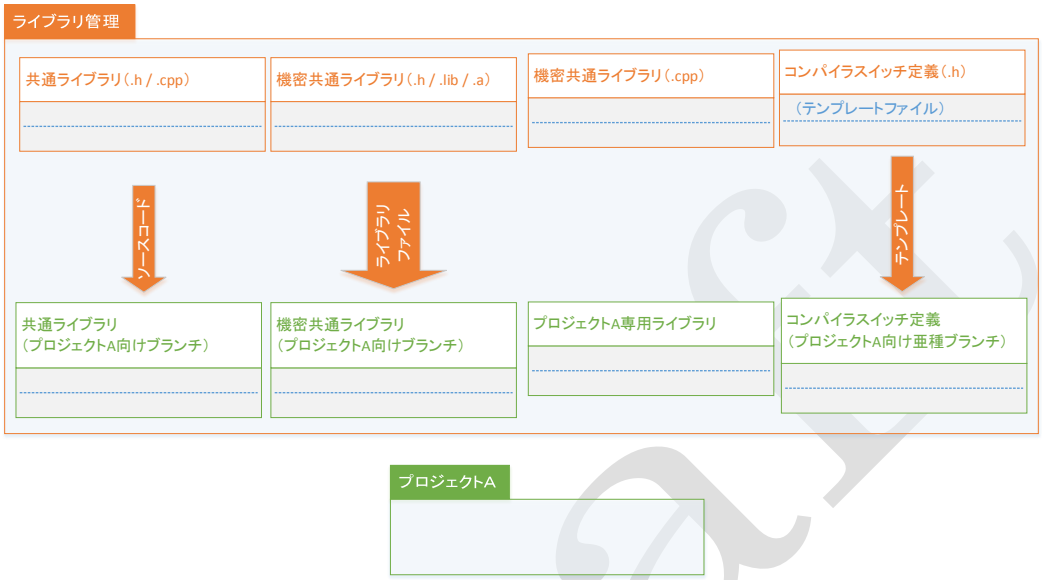


● 複数のライブラリ

つまり、開発プロジェクトは、二つ（以上）のライブラリを扱うものとする。

▼ 特定プロジェクト向けのライブラリのカスタマイズ

共通ライブラリは、多数のプロジェクトおよびプラットフォームにまたがって共通利用するため、特定プロジェクト向けに、プロジェクト主導でライブラリのカスタマイズが可能なものとする。



● マクロによるコンパイラスイッチの活用

プラットフォームに特化した処理やオーバーヘッドの大きい処理などは、ライブラリのソースコード中に「#ifdef（マクロ）～ #nedif」を仕込んで、マクロの定義状態に応じて挙動を変えるものとする。

ごく一般的な手法である。

● コンパイラスイッチの方針①：プロジェクトの判別を禁止

共通ライブラリのソースコード中には、「#ifdef IS_PROJ_A」のような、特定のプロジェクトを識別するようなコードは一切含んではいけないものとする。

要するに、ライブラリがプロジェクトを判別するようなことは、決して行ってはいけない。

● コンパイラスイッチの方針②：プロジェクトで機能の選択

共通ライブラリの挙動・有効化する機能の選択は、全てプロジェクト側から指定するものとする。

プロジェクト固有のコンパイラスイッチ定義ファイルにより、「#define FUNCTION_X_ENABLED」のようにして、ライブラリの挙動を決定づけるためのマクロを個々に定義するものとする。

● 所定のコンパイラスイッチ定義ファイル

対象プラットフォームや、使用する機能の選択といった、各プロジェクトの要件に見合った挙動を決定づける、コンパイラスイッチとしてのマクロの定義ファイルを一つだけ用意するものとする。

そして、そのファイルは全てのプロジェクトで共通して、所定の場所の所定の名前のファイルで扱うものとする。

● コンパイラスイッチ用ブランチ

コンパイラスイッチ定義ファイルは、必ず全プロジェクトで異なる内容となる。そのため、コンパイラスイッチ定義ファイル一つだけを持つリポジトリを設けるものとする。

なお、このリポジトリは、亜種ブランチとして扱う。

● コンパイラスイッチテンプレート

共通ライブラリへの機能追加に伴い、コンパイラスイッチの追加も発生する。そのため、ライブラリの提供とともに、コンパイラスイッチの説明を記したファイルも合わせて提供するものとする。

このファイルを「コンパイラスイッチテンプレート」と呼ぶ。(C++のテンプレートクラス／テンプレート関数とは全く無関係)

● 既定のライブラリ配置

コンパイラスイッチ定義ファイルとともに、共通ライブラリは全てのプロジェクトで同じ配置にするものとする。

● 強制インクルード設定

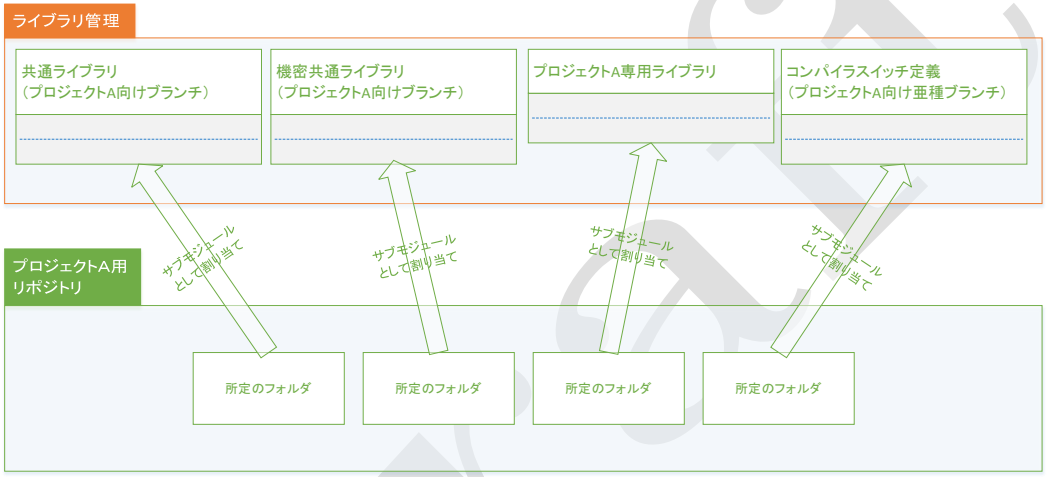
コンパイラスイッチ定義ファイルは、各プロジェクトで「強制インクルードファイル」(「必ず使用されるインクルードファイル」) に設定することを推奨する。

● プリコンパイル済みヘッダー

コンパイラスイッチ定義ファイルは、各プロジェクトで「プリコンパイル済みヘッダー」に含めた上で、プリコンパイル済みヘッダーを強制インクルード設定することを推奨する。

▼ ライブラリのサブモジュール化

各開発プロジェクトでは、多数のライブラリをひとまとめに管理するために、Git の「サブモジュール」の機能を活用するものとする。



● 「サブモジュール」について

サブモジュールは、リポジトリ内の所定フォルダに他のリポジトリ（とその特定のリビジョン）を割り当てる、Git 特有の機能である。

なお、Mercurial では「subrepo」という同様の機能で提供されているとのこと。

● 容易なプロジェクト配布

サブモジュールにより、主体となるリポジトリをチェックアウト（Git では「クローン」）するだけで、関係するリポジトリを全てまとめて取得できる。

これが Subversion なら一つずつのチェックアウトが必要で、個人個人がリビジョン（ブランチ）を合わせて管理しなければならず、手間となる。それ以前に、リポジトリ内のサブフォルダに別のリポジトリを展開することが危険。

● 明示的な「サブモジュールの更新」

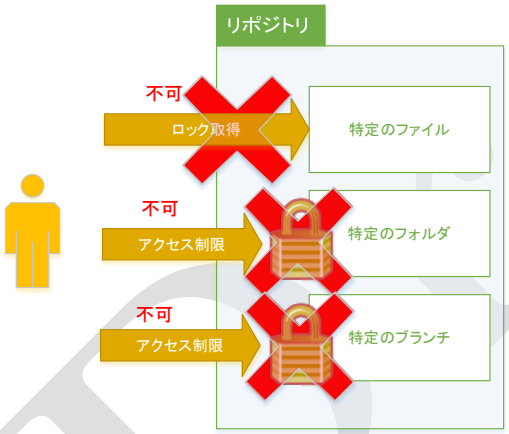
サブモジュールは、リポジトリを更新（Git では「プル／フェッチ」と「チェックアウト」）するだけでは最新の状態に反映されないので注意が必要である。

リポジトリの更新によって、サブモジュールに割り当てた「リビジョン」のみが更新されるため、その後明示的に「サブモジュールの更新」という操作を行う必要がある。

サブモジュールが別サーバーに配置されている可能性もあるため、これは必然的な操作でもある。

▼ 分散 SCM の注意点

分散 SCM による開発環境の改善を目指すが、分散 SCM によってできなくなることもあるので、よく注意した上でフレームワークを構成しなければならない。



● 【扱えないこと】シーケンシャルなリビジョン番号

Git の場合、Subversion などのようなシーケンシャルなリビジョン番号はなく、160 ビット（20 バイト）の SHA-1 ハッシュ値でコミットを識別する。

● 【できないこと】ファイルのロック

分散 SCM では、ローカル作業が基本となるため、ファイルのロックを取得して排他的に編集することができない。

このため、マージが不可能なファイル（バイナリファイル）を多数のスタッフで共同編集するようなことはできない。

- **【できないこと】アクセス権設定**

Subversion には基本機能としてユーザー管理とフォルダごとのアクセス権設定があるが、Git にはそれがない。

【解決策】ホスティングサービスのユーザー管理

Git 本体ではなく、「ホスティングサービス」（例えば GitHub など）によって、ユーザー管理とリポジトリに対するアクセス権を管理することは可能。

【やはりできないこと】フォルダ／ブランチに対するアクセス権

ホスティングサービスを使っても、フォルダやブランチに対するアクセス権までは設定できない。つまり、リポジトリを共有している以上は、他のプロジェクトが行ったコミットがまるわかりになる。

【解決策】ホスティングサービスのフォーク

ホスティングサービスの「フォーク」を用いる事で、共通ライブラリに直接コミットされることを防ぐことが可能。これにより、他のプロジェクトによるコミットが直接見えるようなこともない。

ただし、特定のフォルダやブランチを、特定プロジェクトに対して隠すようなことはやはりでいない。

アクセス権の問題

常にリポジトリ全体が公開されることが、本書が提案するフレームワークを実践する上で大きな問題となる可能性がある。ただし、本書内においてはこの問題を不問とする。

■ 分散 SCM を用いた開発の特徴

分散 SCM を用いた開発の特徴を示す。

Subversion などの旧来のバージョン管理システム（VCS）を用いた場合との対比も説明する。

▼ 安全な途中コミット

分散 SCM はローカルリポジトリ（自分の PC 上のリポジトリ）を扱うため、コミットしただけでは他者に反映されることがない。長い作業の途中で、動作が不安定な状態のコミットを行ってもよい。

Subversion などでは、途中コミットを行うと他者の作業に影響を及ぼすため、安定した状態になるまでコミットしてはいけないルールにするのが通常。長い作業の際には、別途ファイルサーバーにファイルを退避するなどの工夫が必要であった。

▼ サーバー障害に強い

ローカルリポジトリで動作するため、共有リポジトリのサーバーがダウンしても作業が止まることがない。

▼ 障害復旧に強い

時には共有サーバーのファイルが破損するような事態も起こり得る。基本的にローカルリポジトリは共有リポジトリの複製であるため、スタッフの誰かのローカルリポジトリのファイルを一式コピーすれば、即座に共有サーバーを復旧できる。

Subversion などでは、レプリケーションのためのやや大掛かりな仕組みを必要としていた。

▼ 手軽なブランチ操作

分散 SCM の開発スタイルは、ブランチを積極的に活用することである。

Subversion などでは、海外版やパッチ版などを開発する時に初めてブランチを作成するが、そうした大局的な節目でのブランチだけではなく、日常業務で一時作業用のブランチを作成する。

● トピックブランチ

個人個人のスタッフが作業を開始する際には、必ず自分の作業用の一時ブランチを作成する。これを「トピックブランチ」と呼ぶ。

なお、これは分散 SCM が提供する機能のことではなく、ブランチ機能を活用した

運用上のルールである。

● ブランチの切り替え

分散 SCM は標準的にブランチをサポートしており、その切り替えは非常に高速で安全である。

例えば、提出ビルド作成中に問題が発覚し、緊急でその修正が求められるようなことがある。この時、スタッフは現在の作業の手を止めて緊急対応にあたるために、途中状態をトピックブランチにコミットした上で、本流にブランチを切り替えて作業する。対応が済んだらまたトピックブランチに切り替えて作業を継続する。

この一連の操作を、極めて高速かつ安全に行うことができる。

Subversion などの場合、本質的にブランチはサポートしておらず、その切り替えは非常に手間である。

スタッシュ

本書では解説しないが、Git では一時的な作業の退避のために「スタッシュ」という仕組みが用意されている。イメージとしてはプッシュ／ポップ型の一時ブランチである。

この機能を用いるよりも、トピックブランチの作成を徹底した方が、作業が安全かつシンプルになる。

● トピックブランチの提出

トピックブランチの作業が済んだら、そのまま本流に反映させず、トピックブランチをグループリーダー（リードプログラマーなど）に提出する。

グループリーダーは内容を確認した上で、問題がなければ本流に反映する。これにより、各スタッフの無造作なコミットが他のスタッフに影響を与えることを防ぐ。

● 機能ブランチ

一人で完結できない機能追加要件の場合、複数名で共有するブランチを作成する。これを「機能ブランチ」と呼ぶ。

扱い方は「トピックブランチ」とほぼ同様であるが、やはり各スタッフはその中でさらにトピックブランチを作成して作業する。機能ブランチのリーダーを設定し、機能全体がまとまったところでリーダーが機能ブランチの提出を行う。

● 頻繁なブランチ切り替え

機能ブランチに関わっていると、機能の完成までに自分の作業を終えて他の作業に着手することがある。また、他者が機能ブランチにコミットしたら、それを反映させて動作を確認する必要も生じる。

このように、頻繁なブランチ切り替えで作業を進めるスタイルとなるが、分散 SCM の強力なブランチ機能により、このような操作を高速かつ安全に行ことができる。

▼ 止めないコミット

ローカルコミットとトピックブランチにより、「提出ビルド作成」のようなクリティカルな業務の最中にも、問題なくコミットを継続できる。

▼ 【問題点】複雑な操作

分散 SCM は、その恩恵が大きい反面、操作が複雑で習得が難しい。

少なくとも、ローカルリポジトリと共有リポジトリの同期を取る操作が必要であり、Subversion などよりも手間がかかる。加えて、トピックブランチを扱うとなると、スタッフに要求する操作の幅はさらに広がる。

● 技術サポート

行うべき操作が多くなると、その分問題が発生する機会も増す。そのため、分散 SCM 導入の際は、技術サポートを行うスタッフを 1～数名擁立することが不可欠である。各開発プロジェクトに、必ずこの役割のスタッフを設定する。

● 技術サポートスタッフに対する要件

技術サポートを行うスタッフは、問題解決のための知識も豊富に習得しておく必要がある。

下記の Git の操作は、できればスタッフ全員が知っておくべきことであるため、技術サポートスタッフは最低限抑えておく必要がある。

- リベース
- リセット
  - ブランチのバージョンを戻したり進めたりする

- ファストフォワードマージ
- チェリーピックマージ
- リバート

スタッフがあまり意識することがない、下記の操作や知識についても技術サポートスタッフはできるだけ抑えておくほうがよい。

【操作】

- ブランチの削除
- Reflog の確認と削除
- スタッシュ
- サブモジュールの割り当て
- サブモジュールの変更とコミット
  - 特殊な操作があるわけではないが、幾つかの点で注意が必要。
    - ・ サブモジュールは常に特定の「バージョン」が割り当てられた状態になっていること。
    - ・ 変更する際には、最初にサブモジュールのフォルダ（リポジトリ）上で、明示的なチェックアウト（ブランチの切り替え）とコミットが必要なこと。
    - ・ 親リポジトリ上で、サブモジュールのフォルダをコミットが必要なこと。
    - ・ TortoiseGit の変更マークが付かず、サブモジュールのフォルダをコミットする必要があることがわかりにくいこと。

【知識】

- リモート追跡ブランチ
  - ローカルリポジトリ上でのリモートリポジトリの管理状態
- HEAD
- インデックス
- 内部管理構造
  - blob, tree, commit および、それらのハッシュ値

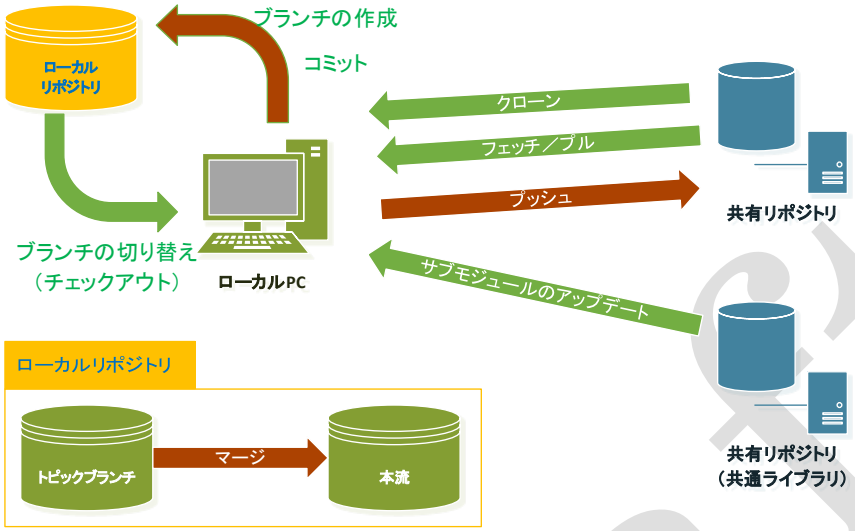
更には、ホスティングサービスの管理機能と下記の操作について十分な知識が必要である。

- フォーク
- プルリクエスト

こうした技術習得が欠かせないことが、分散 SCM 導入の障害となる。

■ 分散 SCM を用いた開発ワークフロー

分散 SCM を用いた開発のワークフローを示す。



▼ 共有リポジトリの作成

プロジェクトの開発を始める際、まずは Subversion などと同様に、サーバー上に共有ポジトリを作成する。

Git の場合、共有リポジトリは通常「Bare リポジトリ」（裸のリポジトリ）として作成する。これは、ワーキングツリー（作業フォルダ）を持たないリポジトリのことである。

▼ 共有リポジトリへのアクセス設定

次に、各スタッフが共有リポジトリにアクセスするための設定を行う。
そのためには、まず共有リポジトリをどのような方法で公開するか決定する必要がある。
共有リポジトリの主な公開方法を以下に列挙する。

- ホスティングサービス上にリポジトリを作成し、http/https/ssh 通信のいずれかで公開する。
- ファイル共有でリポジトリを公開する。
- SSH 通信でリポジトリを公開する。
- Git 専用通信サービス（Git プロトコル）を使用してリポジトリを公開する。

推奨はホスティングサービス＋SSH。

公開鍵と秘密鍵を扱う手間がかかるが、インターネット経由の遠隔地開発でも比較的安心できる。

なお、Git 通信サービスは、セキュリティが皆無のアクセス手段であり、ユーザー間のコミットの受け渡しで一時的に用いるのが通常である。本書では解説しないが、「チェリーピックマージ」という、他者の任意のコミットを反映する操作を行う際に用いる。

▼ クローン

分散 SCM の大きな特徴の一つは、必ずローカル（自分の PC 上）にリポジトリを持つことである。

各スタッフがリポジトリにアクセスする際は、最初に「クローン」という操作を行い、共有リポジトリの複製をローカルに作成する。

このリポジトリはそのまま開発作業に用いるので、ワーキングツリー（作業フォルダ）が必要となる。そのためには、「Bare リポジトリの指定をせずに」クローンしなければならない。

▼ フェッチ／プル

クローン直後はともかく、作業を開始する際は、まず、共有リポジトリ（サーバー）から最新のファイルを取得し、ローカルリポジトリの複製状態を同じ内容にする。

この操作を「フェッチ」と呼ぶ。

● 「フェッチ」操作

「フェッチ」操作はローカルリポジトリの更新を行うだけなので、ワーキングツリーには一切影響を与えない。

フェッチ後、ワーキングツリーを最新の状態にするには、「リベース」か「マージ」操作が必要。

● 「プル」操作

「プル」操作は、フェッチとリベース／マージを同時に行う操作。

▼ ブランチの切り替え（チェックアウト）

作業を開始する際は、まずはその基本となるブランチに切り替えを行う。
この操作を「ブランチの切り替え」もしくは「チェックアウト」と呼ぶ。

● 「チェックアウト」のイメージ

Subversion などのイメージだと、チェックアウトとは、クローンかフェッチのように、共有リポジトリからのファイルの取得のように思えるが、そうではない。ブランチを切り替えて、ワーキングツリーの状態を変えるのがチェックアウトである。

この操作による共有リポジトリへのアクセスは発生しない。

▼ トピックブランチ／機能ブランチの作成

各スタッフが機能追加、機能変更、バグ修正といった作業を行う際は、必ずトピックブランチを作成する。
場合によっては機能ブランチを作成する。

● ブランチの命名規則

ブランチの命名規則を確立すべきである。以下のことがブランチ名から判別できるようになっているとよい。

- ブランチの種類（バージョン分岐ブランチ／機能ブランチ／トピックブランチ）
- （トピックブランチの場合）ブランチ作成者
- ブランチの機能 ID（タスクチケット番号など）
- バグ ID（BTS のチケット番号など）

▼ サブモジュールの更新（アップデート）

共通ライブラリがサブモジュールとしてフォルダに割り当てられている時、それを更新して最新の状態にする必要がある。
この操作を「サブモジュールの更新（アップデート）」と呼ぶ。

● サブモジュールの更新タイミングについて

サブモジュールの更新タイミングは、手動で更新を行った時のみである。
フェッチやリベース、ブランチの切り替えだけでは更新されず、内容がそのままで

あることに注意。

● サブモジュールの状態について

サブモジュールは、通常「no attach」という状態で管理されている点に注意。これは、カレントのブランチがない状態を意味する。

サブモジュールに変更を加える際は、必ず最初にサブモジュールのブランチを切り替えるか作成する必要がある。

▼ サブモジュールの追加

共通ライブラリを特定のフォルダに割り当てる際は、「サブモジュールの追加」という操作を行う。

● 追加したサブモジュールのコミット

Git では、通常コミット対象になるのはファイルのみであるが、サブモジュールに関しては、それを設定した「フォルダ」がコミットの対象になる。

サブモジュール内で変更したファイルをコミットする際は、サブモジュールのフォルダ上でコミットを行ったのち、親リポジトリ上でフォルダをコミットする必要があることに注意。

▼ コミット

分散 SCM における「コミット」操作は、ローカルリポジトリへのコミットのみを意味する。

つまり、コミットしただけでは他のスタッフに変更が伝わらない。

▼ プッシュ

ローカルリポジトリにコミットした内容を共有リポジトリに反映させるには「プッシュ」操作を行う。

● 競合の発生

前回フェッチしてからプッシュまでの間に、他者が同じブランチに対してコミット

している場合、プッシュは競合して失敗する。

● **マージ**

競合が発生した場合、フェッチし直してマージしなければならない。

● **トピックブランチのプッシュ**

トピックブランチを使用している限りは、競合が発生することはまずない。ローカル PC の障害発生に備えて、時折コミットを時折プッシュしたほうがよい。

● **トピックブランチの提出の前に**

トピックブランチが完成し、提出する際には、最新の本流のファイルをフェッチし、ローカルでマージして動作確認してからプッシュするべきである。

▼ マージリクエスト

トピックブランチが完成してプッシュしたら、リーダーに確認とマージを要請する。

この要請は SCM の機能で行うものではなく、メール、口頭、タスク管理システムなどを利用して行う。

▼ マージ

マージリクエストを受けたリーダーは、ローカルリポジトリ上でトピックブランチを本流にマージして内容確認する。

● **【注意】マージコミット**

通常、マージを行うと自動的にコミットされる点に注意。

トピックブランチの内容に問題がないか確認が取れるまで、このコミットをプッシュしないように、さらに注意。

● **【注意】トピックブランチマージ時のファストフォワードマージの禁止**

TortoiseGit のマージは「ファストフォワードマージ」がデフォルトになっているが、トピックブランチのマージ時は、その指定を必ず外すこと。

ファストフォワードマージは、ブランチに対して行われた一連のコミットを、（可能な限り）本流に対するコミットとして展開してしまう。この状態は、本流をロールバックしたい時に問題となるため、ファストフォワードマージは使用禁止とする。

● ファストフォワードマージを使用する場面

逆に本流の内容をトピックブランチに反映させる場合は、ファストフォワードマージを行うべきである。

● 【注意】トピックブランチマージ時のスカッシュマージの禁止

トピックブランチのマージはとにかくマージコミットのみとする。TortoiseGit のマージにある「融合」がスカッシュマージの指定である。

スカッシュマージを行うと、ブランチのコミット内容を全部一つにまとめた新しいコミットを本流に対して行う。この時、本流とブランチが統合されたものとして扱われず、ブランチが破棄されたように見えてしまうので、スカッシュマージは原則使用しないものとする。

● スカッシュマージを使用する場面とチェリーピックマージ

チェリーピックマージで複数のコミットを取り込むときにはスカッシュマージにしてもよい。

チェリーピックマージは、他のブランチで行われたコミットを別のブランチに取り込む場合に使用する。作業の途中で他者のコミットを反映する必要が生じた時に用いる。

● トピックブランチの受け入れ拒否

トピックブランチの内容を許容できない場合、「リセット」操作を使用し、マージ前の状態に戻す。

ローカルリポジトリの状態を元に戻せば、以後のプッシュにも影響が出ない。

トピックブランチの提出者には、結果を告げて再提出を求める。

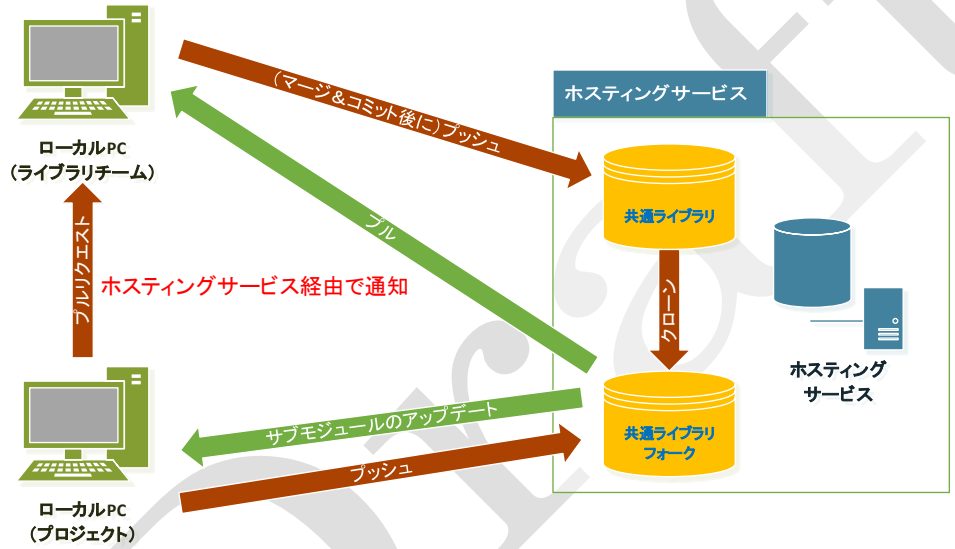
▼ 遠隔地での開発

遠隔地での開発は、拠点内専用の共有リポジトリを設けることで開発を効率化できる。

インターネット経由で中央リポジトリにアクセスする役割のスタッフを数名に絞ることで、セキュリティと通信速度の問題を解消する。

■ ホスティングサービスを使ったワークフロー

ホスティングサービスを使ったワークフローを示す。



▼ ホスティングサービスについて

ホスティングサービスとは、GitHub に代表されるような、共有リポジトリにアクセスするためのサービスである。

ユーザー管理やアクセス権設定など、Git 自体が備えない機能を提供する。

● 商用ホスティングサービス（クラウド）

GitHub などの商用ホスティングサービスをそのまま企業で利用することもできる。

プライベートリポジトリの数に応じた課金が必要になる。

● エンタープライズ版

GitHub などの商用サービスを自社サーバーにセットアップして使用するために、パッケージの販売も行われている。

● 無償版

GitLab や Rhodecode などの優れた無償ホスティングサービスパッケージは、機能強化と共に、相次いで有償化していっている。

無償のサーバーを利用することは、実質的にほとんどできない考えたほうがよい。

▼ ユーザー管理

ユーザー管理のスタイルはホスティングサービスによってさまざまであるが、下記のような要素が扱える。GitHub が対応しているものにはマークを付けて示す。

- ユーザー【GitHub】
- 組織【GitHub】
- ユーザーグループ
- リポジトリグループ
- 組織用プライベートリポジトリ【GitHub】
- リポジトリごとの読み取り／書き込みアクセス権設定

▼ フォーク

共通ライブラリの提供は、ホスティングサービスを通じてフォークで提供することを想定する。

フォークとは、ミラーリポジトリのことである。

中央のリポジトリに対してはコミットする権限を与えず、フォークリポジトリに対してのみコミットを許可することで、共通ライブラリを安全に共有する。

▼ プルリクエスト

フォークに対するコミットは、最終的に中央のリポジトリに受け入れてもらう必要がある。

ホスティングサービスはそのための機能として「プルリクエスト」を提供している。

プルリクエストを受けたライブラリチームは、フォークリポジトリからコミットを受け取ってマージする。

ここでやるべきことは、前述のマージ操作と同じである。

▼ 通信セキュリティ

ホスティングサービスとの通信は、遠隔地からのアクセスを考慮し、SSH 通信が望ましい。

■■以上■■

■ 索引

索引項目が見つかりません。

複数タイトルにまたがる効率的なフレームワーク管理

以　上