

Subversionユーザーの為の 分散型SCM「Git」活用の勧め

2013年7月19日 初版

本書は、これまでSubversion(SVN)を使用した事のあるユーザーに対して、Git(ギット)の活用を勧めるものです。

Gitは、オープンなソーシャル開発の場で広く使われていますが、本書は、企業内で閉鎖的な開発を行う際にGitを活用する事を想定しています。

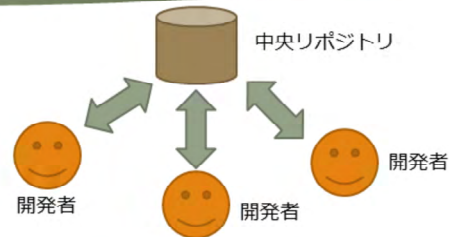
=====

版	リリース	担当	改訂内容
-----	-----	-----	-----
初版	2013年7月19日	板垣 衛	(初版)

基礎知識①：リポジトリの集中型と分散型

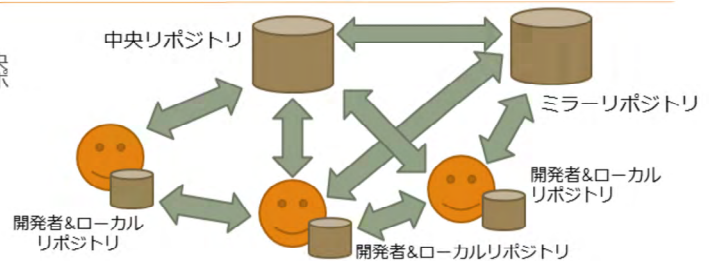
▶ 集中型 (C/S型)

- 開発者チーム全体が一つの中央リポジトリを共通利用するタイプ。
- 主な該当VCS：
SVN(Subversion), CVS(Concurrent Versions System), VSS(Microsoft Visual Source Safe), TFS(Microsoft Visual Studio Team Foundation Server), Perforce, Alien Brain など



▶ 分散型

- 各開発者がそれぞれローカルリポジトリを持ち、中央リポジトリやそのミラーリポジトリなどの多数のリポジトリを同期を取りながら利用するタイプ。
- 主な該当VCS：
Git(ギット), Mercurial(マーキュリアル), Bazaar(バザー), BitKeeper など



集中型と分散型その他、CVSの土台となったRCS (Revision Control System) のように、ローカルで個人利用するのみの、個人型のバージョン管理システムがある。集中型も分散型も、いずれも多人数によるコンピュータ間でのバージョン管理をサポートする。

歴史的には、RCS(1982)→CVS(1990)/VSS(1994)→SVN(2000)といった流れでユーザーが広がっていった。

現在注目を浴びている分散型は、Linuxの開発に使用されていたBitKeeper(1998)が商用ライセンスに切り替わった事を皮切りに、Git(2005)、Mercurial(2005)が開発された。Bazaar(2005)もこの2製品と共に語られる事が多く、これら3製品のユーザーが多い。

SCMホスティングサービス「GitHub」の普及の影響などもあり、Gitユーザーが特に多く、関連ツール類も多い。

「Mercurialの方がSVNに近くわかり易い」、「Bazaarが最も洗練されている」などの意見もあるが、本書は(世の時流に合わせて)Gitのみを扱う。

なお、「リポジトリ」とは、実際のファイルの内容やコミットした人の情報、コミットメッセージなどを含めて、世代管理するデータベースの事。
文中の「VCS」「SCM」という用語については次項で説明。

基礎知識②：VCSとSCM

▶ VCS

- **Version Control System** = バージョン管理システム
 - GitやSVNなどの、バージョン管理を行うシステムそのものを指す。

▶ SCM

- **Software Configuration Management** = ソフトウェア構成管理
 - VCSのみならず、開発プロセスとその為の環境を含んだものを意味する。
 - タスク管理、BTS (Bug Tracking System = バグ追跡システム)、CI (Continuous Integration = 継続的インテグレーション = 自動ビルド、自動テストなど)、プルリクエスト/マージリクエスト、コードレビューといった、VCS以外の機能を含めている。
 - VCSと連携する主な製品としては、Track、Redmine、Jenkins (Hudson) などが、特に有名な物として挙げられる。
 - GitHubのように、VCSと一体となったタスク (Issue) 管理、プルリクエストなどの機能を備えたホスティングサービスもある。

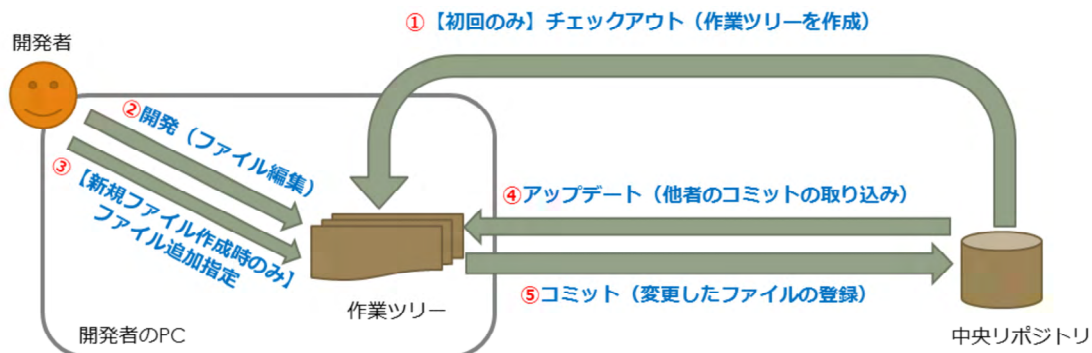
本書で言う「SCM」は、文中にある通り、「Software Configuration Management」の意味で統一する。

ややこしい事に、「SCM」という用語には、「VCS」とほぼ同義語の「Source Code Management」や「Source Control Management」という解釈もあるが、本書ではこの意味で「SCM」という用語を用いる事はない。

また、コンピューターシステムの世界では、「SCM」という用語の意味として、「Supply Chain Management」という、物流系のシステムで用いられる用語があるが、本書の内容とは全く関係がない。

ネットで検索すると、この用語が多くヒットするので注意。

Subversionのワークフロー



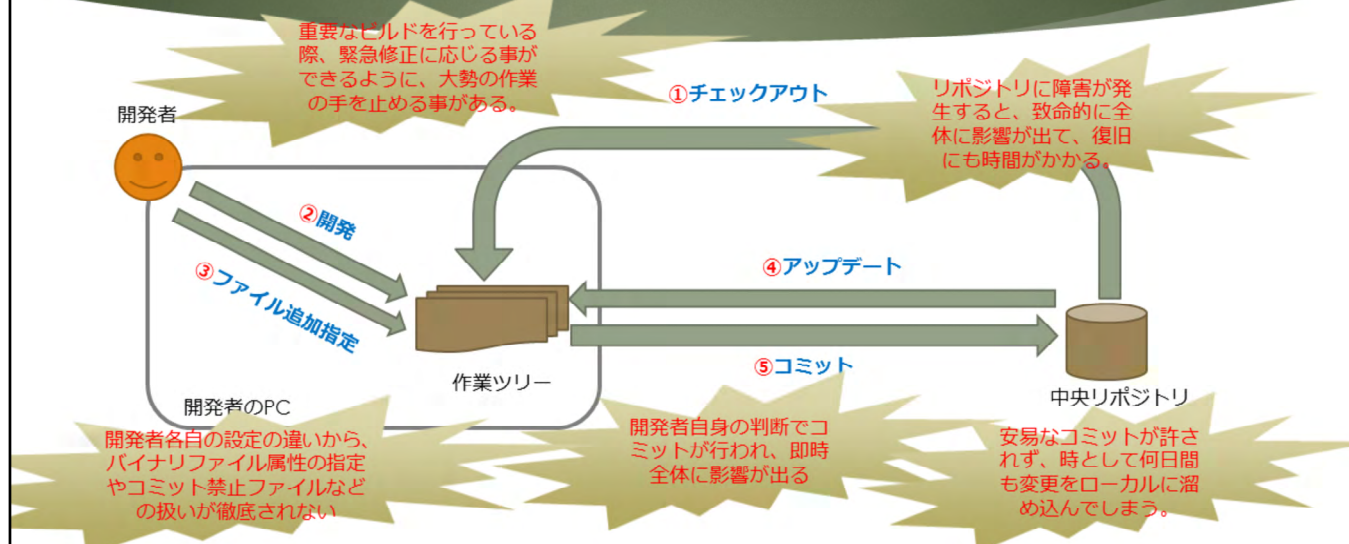
Subversionでは、「中央リポジトリ」を唯一の共有リポジトリとして扱う。

「作業ツリー」とは、実際の開発作業として直接編集するフォルダやファイルの事。

③の「ファイル追加指定」操作は、リポジトリに登録するファイルとそうではないファイルを区別する為に必要。

通常、例えばコンパイラが生成する中間ファイルなどはリポジトリに登録せず、ソースコードのみをリポジトリに登録する。

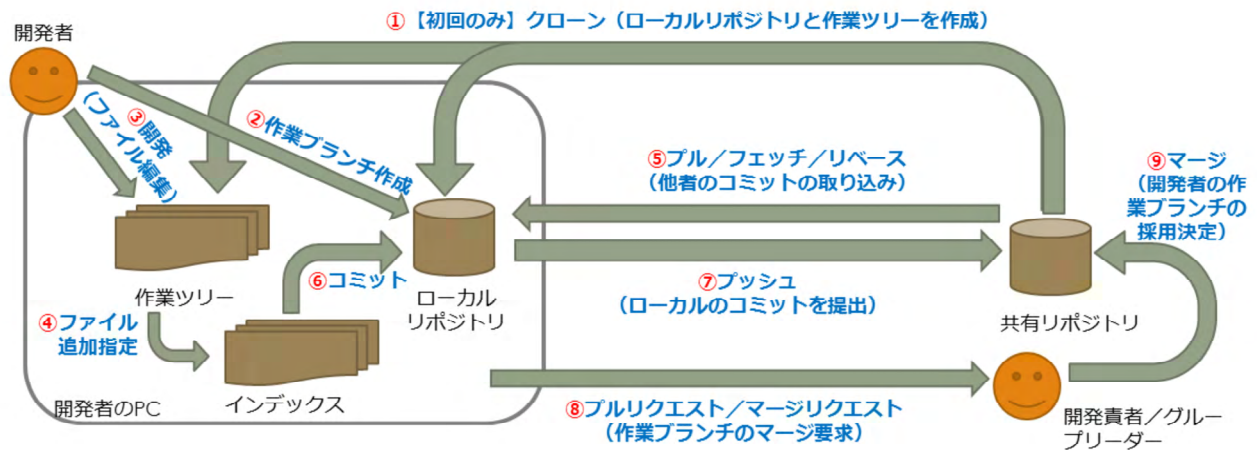
Subversionの問題点



リポジトリに障害が発生すると、多くの場合、開発は全員の作業の手を止めてしまう。

リポジトリの障害対策として、フックスクリプトを活用してリポジトリのレプリケーション(ミラーリング)を行ったり、毎日定時にリポジトリをダンプ出力するようにはしたりといった工夫が必要。

Gitのワークフロー



初回に行う「クローン」は、先に共有リポジトリ(中央リポジトリ)が存在している必要がある。

ただし、Gitの場合、Subversionとは異なり、共有リポジトリが無い状態で、ローカルにリポジトリを新規作成する所から始め、後から作成した共有リポジトリにpushする事が可能。

逆にクローンを行う場合は、ローカルリポジトリを先に作っておく必要はない。

なお、共有リポジトリの作成時は、作業ツリーを持たないBase(裸)リポジトリとして作成し、ローカルリポジトリの作成時は、作業ツリーを伴うリポジトリとして作成する。

TortoiseGitの「ここにリポジトリを作成」コマンドのデフォルトは後者。

ファイル追加時に使用される「インデックス」とは、コミット時にリポジトリに記録される、作業ツリーの変更内容を一時記憶する領域の事。

実際にはローカルリポジトリ内で管理される情報ではあるが、あくまでもコミット前の状態が記録される為、表面上リポジトリにあるように見えない。

(他者がこのローカルリポジトリをクローンしても、コミット前のインデックスのファイルは得られない。)

なお、インデックスへのファイルの追加は、新規ファイルに限らず、変更したファイル全般に対して行う必要がある。

ただし、TortoiseGitのコミットコマンドや、git commit コマンドのオプションでは、既存の

ファイルの変更の場合に限り、インデックスへの追加とコミットをまとめて行う事ができる。(Subversionと同様の動作になる。)

このように、「インデックス」にコミット候補の記録を行うのは、Mercurialなどの他のVCSにはない、Gitの特徴の一つ。
Gitのリポジトリの管理構造に依拠する仕組み。

Gitでは、Subversionと比べてブランチを積極的に活用する開発スタイルとなる。
この図では、基本的な流れとして、開発者が毎回作業ブランチを作成して、リーダーの判断でマージする事を示しているが、開発者が直接マスター(本流)にソースを登録する事も可能。

「作業ブランチ」と表記しているのは、Subversionにおけるブランチよりもかなり意味が軽い。
一つの作業タスクや一つのバグ修正にあたって毎回ブランチを作成する、いわゆる「トピックブランチ」と呼ばれるブランチを指す。

また、数名で一つの機能を作成している間などは、もう少し大掛かりなブランチとして、「機能ブランチ」と呼ばれるブランチを作成して、その一部のスタッフ間だけで共有して作業する。

もちろん、Subversionと同じく、過去のリリースを保持しつつ、新しいリリースを開発する為の「開発ブランチ」としても用いられる。
長期的なブランチとしては、「開発ブランチ」、「安定ブランチ」、「メンテナンスブランチ」のように細かく分けて扱う場合もある。

Subversionと同じく、特定のリリース状態を「タグ」として記録する事も可能。

いずれのブランチも、システムの機能としての違いはなく、あくまでも作業を進行する上での約束事として使い分ける。

ブランチの名称には、ブランチの種類を示す接頭辞を付ける事や、作業タスクのチケット番号／BTSの番号を入れる事、個人の名前を入れる事など、命名規則を定めて開発する。
(例: issue/m_itagaki/123, bug/m_itagaki/456, feature/23, devel/1.0, master, maint/1.1 など)

上記のような細かいブランチを活用する事で、例えば、何かしらの新しい開発を行っている時に、数日前にコミットしたソースの修正が求められた場合などにも混乱なく対応ができる。

その時作業中だったものをローカルにコミットした上で(他者に影響が出ないので中途半端な状態でコミットして良い)、修正対象のブランチに切り替えて作業を行い、終わったら中断していた作業に戻る事ができる。
ブランチを切り替えると、Gitはソースコードの状態を適切に切り替えてくれるのが利点。
しかもそれはローカルで行われるので早い。

これにより、重要なビルド工程の際に、万が一に備えて大勢の開発者にビルドの結果を待って作業の手を止めてもらうといった、Subversion使用時に起こっていたワークフロー上の問題を改善できる。

「プルリクエスト」や「マージリクエスト」は、直接的な声掛けで運用する事も可能だが（「一通り出来たのでマージして下さい」と依頼する）、GitHubやGitLab、RhodeCodeといった、いわゆる「ホスティングサービス」を使用すると、そのような通達をシステム的な機能として実現できる。

このようなコミュニケーションの為の機能は、Git自身は有していない。

図には描いていないが、共有リポジトリとして、Subversionのリポジトリを使用する事もできる。

Gitの利点①

▶ 頻繁なコミットが可能

- 不安定なソースコードをコミットしても他者に影響が出ないので、ビルドが通らないような状態でも、小まめにコミットできる。
- 開発ブランチを用いる事により、共有リポジトリにプッシュしても安全。

▶ 管理者を通したワークフローの健全化

- 開発ブランチのマージを管理者に委ねる事により、開発者自身以外の判断で正規のシステムへの組み込みを判断する事や、事前のコードレビューなどを行えるようにする。

▶ 開発者の作業の手を止めない／問題発生時はすぐに対処

- 作業ブランチにより、本流に影響を与えないワークフローの為、重要なビルドを行っている最中も個人の作業を継続できる。
- 問題が発生したら、その時点の作業ブランチを一旦コミットし、本流に切り替えて、問題点の対処を行える。

開発ブランチでのコミットを小まめに共有リポジトリにプッシュしていると、万が一開発者不在時に途中状態のソースが必要になる事があっても、第3者が参照する事ができる。

管理者や機能ブランチの開発責任者(サブグループのリーダー)に本流へのマージを依頼する体制は、頻繁にリポジトリを作成する作業ルールとする事により、システム的に実践し易くなる。

GitHubやRhodeCodeなどのホスティングサービスを利用すると、この依頼を出す際に、システムの機能として、「プルリクエスト」「マージリクエスト」を用いる事ができる。

開発者は、緊急で問題に対処する際、そこまでの作業状態が中途半端でもコミットできるので、ブランチを切り替える際に面倒なマージが発生しない。

Gitの利点②

▶ コミットのキャンセル・再コミット

- Subversionなどと異なり、一度コミットした内容を修正して再コミットできる。

▶ オフライン開発

- ローカルリポジトリへのコミットは、共有リポジトリがなくても可能な為、オフライン状態でも作業を継続できる。

▶ 遠隔地開発

- 遠隔地の拠点に専用の共有リポジトリ（ミラーリポジトリ）を設置して、拠点内のチームで完結する作業体制を取る事ができる。

Subversionのようにコミットを絶対視するシステムと異なり、一度行ったコミットをなかった事にしたり、追加・修正したファイルのコミットを漏れやコミットメッセージ・作者の間違いを後から修正したりといった事ができる。

これにより、不用意にコミットログが長くなる事を防ぎ、常に重要なコミットがわかり易い状態を保つ。

ただし、共有リポジトリへのプッシュ前のローカル作業に限定して許可するなどの運用ルールは設けた方が良いかもしれない。

なお、Subversionでもフックスクリプトを適切に設定する事により、コミットメッセージの変更は可能になる。

オフライン状態とは、例えば、ネットワーク障害やサーバー障害が発生している時や、また、移動時にノートPCを使って開発している時など。

完全にオフラインの状態でも作業を継続してコミットし、後々オンラインになった時に、共有リポジトリにプッシュする事ができる。

遠隔地開発では、拠点内のチームリーダーが代表して、実際の共有リポジトリに変更をpushするような体制を取る。

Gitの利点③

▶ リポジトリのバックアップ

- バックアップの為の特別な仕組みを用意しなくても、リポジトリのコピーがあちこちのローカルにあるので、普及しやすい。

▶ コミット除外ファイル、ファイル属性をリポジトリに記録して共有

- リポジトリの先頭ディレクトリに特定のファイル名でそれらを定義できる為、ユーザーごとにファイルの扱いが異なるといった問題を防ぐ事ができる。

リポジトリのバックアップとしては、例えば、実際の共有リポジトリをLinux上で扱い、Windowsサーバーにバックアップを置きたいような場合は、Linux上のリポジトリでフックスクリプトを利用し(もしくは短い間隔の定期実行で)、Windowsサーバー上のミラーリポジトリに逐一変更を反映するようなやり方が考えられる。

除外ファイル、ファイル属性の指定は、Subversionでは、各ユーザーのローカル設定になっていた。

除外ファイルとは、例えば .o ファイルや .suo ファイルなどの中間ファイルを指定する。ファイル属性とは、例えば .xlsx や .docx をバイナリファイルとして扱うといった指定。これにより、改行コードの変換が行われてバイナリファイルが崩れるといった問題を防ぐ事ができる。

Gitでは、これらの設定を、リポジトリの先頭ディレクトリに、.gitignore(除外ファイル設定)、.gitattributes(ファイル属性設定)を置く事により、開発者全体で設定を共有できる。リポジトリ毎にファイルを置き直す必要があるが、個人個人に設定を委ねるよりは遥かに管理し易い。

なお、Subversionでもファイルコミット時(コミット後)に属性を与えてコミットする事で属性がリポジトリに保存される。

また、ディレクトリに対して無視ファイルを設定してコミットする事が可能。

とはいえ、管理者が後追いで開発者が正しい属性でコミットしたかを確認する必要がある為、管理の手間がかかる。

Gitの方式の方が管理し易い。

Gitの問題点

- ▶ 操作が複雑で覚えるべき事も多く、敷居が高い
- ▶ Subversionのように、自動で採番される一貫したシンプルなりビジョン番号がない
- ▶ RCS, CVS, Subversionのようなキーワード展開が出来ない
- ▶ ファイルロックで編集の競合を防ぐ仕組みがない
- ▶ 空のフォルダを扱えない
- ▶ ユーザーのアクセス権制御が難しい

Subversionのようなコミット毎の(チェンジセット毎の)一貫したリビジョン番号は存在せず、ファイル名やファイル内容に基づいて算出された160ビットのSHA-1ハッシュ値がそのファイルのリビジョンとして扱われる。
その為、ある時点のリポジトリの状態を、Subversionのようにリビジョン番号一つで表現する事はできない。

Gitにキーワード展開の機能がないのは、上記のリビジョン番号の採番の仕組みと密接に絡んでる。

ファイル内容に基づいてハッシュ値を得てリビジョン番号とする都合から、コミット時やpush時にファイルの内容を変更する事ができない。

しかし、インデックスへの変更内容登録時(git add 時)と、ブランチの切り替え時(git checkout時)に、キーワード展開を行うようにする事は可能。

それには、そのタイミングで実行されるフィルターの指定と、そこに与える変換用のスクリプトを用意する必要があり、面倒が伴う。

全開発者のローカルリポジトリに徹底して導入する必要などもあり、基本的にキーワード展開は使えないものとした方が無難。

ファイルロックはGitの管理構造上、共有リポジトリの状態をコミット時に確認したりしないので、基本的に不可能。

ファイルロックが必要なものはSubversionで管理するか、別途ロック状態を確認しあえるような仕組みを導入する必要がある。

Excelファイルのように、複数のスタッフが編集する可能性のあるバイナリファイルの扱いに注意。

Gitは基本的にファイルしか管理しない為、空のフォルダは扱えない。

Gitのリポジトリに対して、ユーザー毎に読み取り専用などのアクセス権を設定する場合、gitoliteやGitLab、RhodeCodeのような管理ツール(ホスティングサービス)を別途導入する必要がある。

なお、Subversionは標準でリポジトリやブランチ／タグに対するユーザーアクセス権限を管理する仕組みを備えている。

Gitの場合、そもそもユーザー管理がない為、それを含めて管理ツールに委ねるものとなる。

Gitを活用すべきケース

- ▶ プログラム開発（ソースコード管理）全般
- ▶ 遠隔地（拠点間）での共同開発を行う場合
- ▶ ソーシャル開発を行う場合

ただし、Git利用者は全員がきちんとGitの作法を学ぶ必要があり、習得の敷居が高い。ケースに応じたマニュアルを用意して、なるべく単純化できるようにすべき。

このような開発体制で少しでも問題を起こさないようにする為に、不用意な競合が発生しないように注意すべき。

.o ファイルや .suo ファイルといった中間ファイルをリポジトリに含んでしまわないようによく注意する。

開発者の多くが在宅勤務であったり各社に散らばっているような状況ではGitが扱い易い。（遠隔地／ソーシャル開発）

Gitを活用すべきでないケース

- ▶ Excelなどのバイナリファイルを複数の開発者が変更しなければならない場合
- ▶ 既にSubversion等を使用していて、前述のような問題が出ていない現場の場合
- ▶ 多くのスタッフに対して、なるべく簡潔化した操作マニュアルを提示できない場合
- ▶ 問題発生時に十分スタッフのサポートができる体制が取れない場合

バイナリファイルなど、ロック必須のファイルを扱う場合は、やはりSubversionの方が扱い易い。

こういったファイルをどうしてもGit上で扱う場合は何かしらの工夫が必要。

考えられる工夫としては、下記のようなもの。


- ①遠隔地の拠点間で共通して変更しなければならないようなファイルは持たない。
- ②各拠点毎に、変更可能なファイルを定める。(フォルダで決めるなど、なるべく単純なルールに)
- ③【愚策】ロックの代わりに、現在編集集中のファイルを、ファイルサーバー上の共有Excelに記入して宣言し、スタッフ間で編集集中のファイルかどうかを確認し合うようにする。

やはりSubversionに比べてGitは操作が複雑で、その分オペレーション上のトラブルが発生する懸念も高い。

その為、問題なく十分うまくいっている現場では、Git習得などの目的が無い限り、むやむにやり方を変える事はない。

また、スタッフ全体に十分なGitの習得スキルを求めるのは容易ではないので、導入の際は簡潔なマニュアルを用意し、問題発生時のサポートができる体制を取るべき。

スタッフのサポートについては、本来Gitに限った話ではないが、本格運用に先立って、十分にGitを習得したスタッフが、全体の規模に合わせて数名確保できた状態にすべき。



以上

SUBVERSIONユーザーの為の
分散型SCM「GIT」活用の勧め