Git (ギット) によるソースファイル管理

視覚メディア研究室

1　はじめに

1.1　バージョン管理の必要性

1.1.1　更新履歴の記録

ソフトウェアを開発していると，以前は動いていたのに変更したら動かなくなったということが起こる．また，トラブルやミスにより成果物を失ってしまうこともある．ソースコードの更新履歴を保存しておくことにより，過去のバージョンのソースコードを閲覧したり，開発中のソフトウェアをそのバージョンに戻したりできる．

1.1.2　ソフトウェアの共同開発

一般にソフトウェア開発は，複数の開発者が参加するプロジェクトとして行われる．この場合，異なる開発者が同じソースコードを編集するということが起こり得る．同一のファイルを複数の人が同時に編集すると，編集内容に矛盾が発生したり，最新バージョンがどれだかわからなくなったりすることがよくある．

ソースコードの更新履歴を管理することにより，常に最新バージョンが明らかになり，複数の開発者の編集内容の統合（マージ）や競合（コンフリクト）の調停を行うことができる．

1.2　VCS (Version Control System)

ソースコードの履歴管理を行うソフトウェアを VCS (Version Control System) - バージョン管理システムと呼ぶ．VCS において，更新履歴は以前のバージョンとの差分（変更点）によって管理される．この差分を保持するディレクトリをリポジトリ (Repository, Repo) と呼ぶ．

VCS として，古くから SCCS (Source Code Control System) や RCS (Revision Control System) が使われてきた．リポジトリを持つものとしては，CVS (Concurrent Versions System) や SVN (Subversion) がある．また，Microsoft Visual Studio にも VSS (Visual Source Safe) という VCS が用意されている．これらはソースコードを単一のリポジトリで管理する集中型の **VCS** である．

これに対し分散型の **VCS**，いわゆる **DVCS (Distributed VCS)** も使われている．これは開発者間で共有するリポジトリの他に，個々の開発者も専用のリポジトリを持つ．この場合，開発者が個々に履歴管理を行い，結果を共有のリポジトリに反映することになる．これには Git や Mercurial などがある．

VCS や DVCS によりバージョン管理を行う場合，開発者は最初にリポジトリからソースファイルを取り出して，手元（のマシンやディレクトリ）に保存する．この操作をチェックアウトという．また，手元のファイルを編集してリポジトリを更新する操作を，チェックインという．なお Git などの DVCS では，この操作は手元のリポジトリに対するものと共有のリポジトリに対するものの二段階になる．

手元にファイルを置いている間に他の開発者がチェックインする可能性があるので，ファイルの編集を始める前にはリポジトリから最新のソースコードを取り出して，手元のファイルを更新すべきである．この操作をアップデートという．また，自分がチェックインする前にも適宜アップデートを行い，他の開発者の編集内容と自分の編集内容を統合しておく．

1.3 Git

Linux のカーネル開発で使われている DVCS である．Windows でも使える．GitHub (`http://github.com/`) のような公開リポジトリ（非公開にすることもできるけど有料）もある．

2 Git のインストール

たいていの Linux ディトリビューションは Git のパッケージを用意している．Mac OS X にも `git` コマンドは最初から入っている（Xcode 4 が対応している）．Windows については `git` コマンドやそのためのシェルあるいは GUI が入手可能である．詳しくは Git のホームページ `http://git-scm.com/` を参照のこと．

3 ローカルリポジトリの作成

3.1 初期設定

最初に開発者の名前とメールアドレスを登録する．これは `git` コマンドを使うすべてのマシンで最初に一度だけやっておく必要がある．“Your name” には開発者の名前，user@center.wakayama-u.ac.jp には開発者のメールアドレスを設定する．これらは commit（後述）するたびに履歴情報に埋め込まれる．

```
git config --global user.name "Your name"
git config --global user.email user@center.wakayama-u.ac.jp
```

ここで標準的に使用するテキストエディタを指定することもできる．指定しなければ vim が起動する[1]．emacs を使う場合は，下記のコマンドを実行する．

```
git config --global core.editor emacs
```

3.2 git init - リポジトリの初期化

履歴管理をおこなうプロジェクトのディレクトリ（フォルダ）に cd する．Mac では cdto というツールをインストールしておくと便利である．Windows では，Git をインストールしていればフォルダの右クリックで “Git Bash Here” というメニューが現れる．そこで以下のコマンドを実行する．

```
git init
```

3.3 .gitignore ファイルの作成 - 履歴管理しないファイルの指定

プロジェクトのディレクトリの中にある履歴の管理を行わないファイル[2]のファイル名を .gitignore というファイルに列挙して，プロジェクトのディレクトリに置く．ファイル名の指定にはワイルドカードが使用できる．

```
a.out
core
*.o
*~
```

このファイルの書式などの詳細は自分で調べること．

[1] vim のもととなった vi は emacs と双璧をなす Unix / Linux 伝統のテキストエディタなので，使い方は覚えておいて損はない．
[2] テキストファイルでないファイルやサイズが巨大なファイル，他人には見られたくないファイルなど．

3.4 git add - 履歴管理するファイルの追加

履歴管理を行うファイルをリポジトリに追加（登録）するには `git add` コマンドを実行する．

```
git add ファイル名 …
```

そのディレクトリ以下のすべての（`.gitignore` に登録していない）ファイルを追加するには，カレントディレクトリを表す `.`（ピリオド）を指定すればよい．

```
git add .
```

ファイルの追加を取り消すには `git rm` コマンドを実行する．`--cached` というオプションを付けないとファイル自体も削除されてしまうので注意すること．

```
git rm --cached ファイル名 …
```

3.5 git rm / git mv - 履歴管理しているファイルの削除と移動

`git add` コマンドで追加したファイルは Git の管理下にあるので，不用意に削除したり移動したりしてはいけない[3]．ファイルの削除や移動は `git rm` コマンドや `git mv` コマンドを使って行う．

```
git mv 元のファイル名 変更後のファイル名／移動先
git rm ファイル名 …
```

3.6 git commit - 履歴の登録

ファイルの追加が終わったら，commit してローカルリポジトリに登録する．`-m` オプションで指定している “first commit” はこの commit に付けるコメントである．

```
git commit -m "first commit"
```

4 リモートリポジトリの作成

VCS は複数の開発者による共同開発を支援する．Git の場合，同一のコンピュータなら，前章で作成したリポジトリを `git clone` コマンド（後述）でコピーして共同開発を行う．Git はさらに，ネットワークを介して異なるマシン間で共同開発する場合にも使用することができる．その場合はリポジトリだけを特定のコンピュータ上に作成し（リモートリポジトリ），それを共有する方法を採ることもできる．

4.1 ssh クライアント

4.1.1 遠隔ログイン

Windows 版の Git は MinGW や Cygwin で動いているので，その端末ウィンドウ（シェル）で `ssh` コマンドが使える．サーバ host.wakayama-u.ac.jp にログインするには，下記のようにする．

```
ssh user@host.wakayama-u.ac.jp
```

このほかのクライアントとして，Windows には Tera Term や PuTTY がある．Tera Term は余計なものをインストールしない「コンパクトインストール」を個人的におすすめする．PuTTY には日本語化された「ごった煮版」がある．

[3] でも多分大丈夫．

4.1.2 ファイル転送

これも Git をインストールしていれば，シェルから scp コマンドや sftp コマンドが使える．このほかのクライアントとして，Windows では WinSCP が使える．Mac にはもともと scp コマンドや sftp コマンドが入っているが，Cyberduck も使いやすい．また Linux / Windows / Mac で使える FileZilla というクライアントもある．

4.2 鍵作成（この作業は飛ばしても構わない）

鍵は ssh-keygen コマンドで作成できる．公開鍵と秘密鍵を作成し，公開鍵をログイン先のコンピュータに保存する．保存の仕方はログイン先に依存するが，システム情報学センターのサーバの場合は，ホームディレクトリにある .ssh という（不可視）ディレクトリ内に置いた authorized_keys というファイルに追加する[4]．これは単に次のコマンドを実行すればよい．

```
cat 公開鍵ファイル >> ~/.ssh/authorized_keys
```

なお，鍵を作成する際にパスフレーズ（鍵を開くためのパスワード）を付けないでおくと，遠隔ログインの際にパスフレーズの入力を求められなくなる．秘密鍵を持ったクライアントからしかログインできないので他人が勝手にログインできるわけではないが，秘密鍵が漏れて（他人に知られて）しまうと危険なので，パスフレーズは付けておくべきである．

4.3 空のリポジトリの作成

まず ssh クライアントを使ってサーバ host.wakayama-u.ac.jp に遠隔ログインする．リポジトリを user というユーザのホームディレクトリの下の git というディレクトリに作るなら，次のコマンドを実行する．myproject は自分が作成するプロジェクトの名前である．

```
mkdir ~user/git/myproject.git
cd ~user/git/myproject.git
git --bare init
```

リポジトリだけを置く（ソースファイルなどは置かない）ので，この git init には --bare オプションを付ける．リポジトリを自分のホームディレクトリ直下に作るなら，“ user/git” は “ ” だけでよい．
この後ログアウトする．以後はリポジトリを削除するときくらいしかログインすることはないと思う．

4.4 git remote add - リモートリポジトリの登録

ローカルリポジトリに git commit した内容をリモートリポジトリに反映するには，最初に手元のプロジェクトのディレクトリで git remote add コマンドを実行して，リモートリポジトリを登録する．

```
git remote add origin ssh://user@host.wakayama-u.ac.jp/~user/git/myproject.git
```

このコマンドは host.wakayama-u.ac.jp の user/git/myproject.git というディレクトリにあるリポジトリを，origin という名前のリモートリポジトリとしてローカルリポジトリに登録する．
ローカルリポジトリの内容をリモートリポジトリに反映するには，git push コマンドを使う．

```
git push -u origin master
```

[4] このファイルが無ければ新規作成する．

これは origin で表されるリモートリポジトリにローカルリポジトリの master というブランチ（後述）の内容を反映する．一度 git push コマンドに -u オプションを付けて実行すれば，以後 git push コマンドを実行するときに引数の “origin master” を省略できる．

```
git push
```

5 日常の手順

5.1 git clone - リポジトリのコピー

別のコンピュータで作業をする場合や，他人のプロジェクトを取り出す場合など，既にリモートリポジトリに反映されている内容を手元にコピーするには，git clone コマンドを用いる．これは最初に一度だけやればよい．

```
git clone ssh://user@host.wakayama-u.ac.jp/~user/git/myproject.git
```

これでプロジェクトのディレクトリが手元に丸々コピーされる．

5.2 git pull - アップデート

手元にプロジェクトのコピーが既にある場合は，そのファイルを編集する前に，git pull コマンドを使ってリモートリポジトリの内容をローカルリポジトリに反映しておく．

```
git pull
```

このとき競合（コンフリクト）が発生していたら，他の開発者と相談の上，どう対応するか考える．詳細は git log コマンドや git show コマンドで調べることができる．

5.3 git commit - 履歴の登録

ファイルの編集が終わったら commit してローカルリポジトリに履歴を登録する．-a オプションを付ければ，変更されたファイルに対して git add も自動的に実行してくれる．

```
git commit -a -m "コメント"
```

-m オプションで指定する “コメント” には，どういう編集や変更をおこなったかを記録しておく．-m オプションを省略した場合は，初期設定でテキストエディタを指定していなければ，vim が起動する．なお，コメントを指定しないと commit されない．

5.4 commit の取り消し

commit を取り消す場合は git reset コマンドを使う．このとき，commit したことだけを取り消してファイル自体はそのままにしておくなら--soft オプションを指定する．

```
git reset --soft HEAD^
```

--hard オプションを指定すれば，編集したファイルも一緒に commit 前に戻す．

```
git reset --hard HEAD^
```

なお，直前の commit 内容を捨てて現在の編集内容で commit を新たにやり直すには，git commit に --amend オプションを追加する．

```
git commit --amend -a -m "やり直した内容に対するコメント"
```

5.5 git push - リモートリポジトリへの反映

commit した内容をリモートリポジトリに反映して他の人と共有するには，次のコマンドを用いる．

```
git push
```

6 ブランチ

プロジェクトに対して大幅に変更を加える場合には，一旦そのリポジトリを枝分かれさせて，それに変更を加えた後，変更点をもとのリポジトリに統合（マージ）するという手順を取る．

6.1 ブランチの作成

リポジトリを枝分かれさせるには，`git branch` コマンドを使う．枝分かれさせたリポジトリをブランチと呼ぶ．newbranch は枝分かれさせたブランチに付ける名前である．

```
git branch newbranch
```

ブランチ名（ここでは newbranch）を省略すると，現在のリポジトリにあるブランチの一覧が表示される．もともとあったブランチは master になっているはず．

6.2 ブランチの切り替え

あるブランチに対して変更を加える時は，`git checkout` コマンドを使ってそのブランチに切り替える．

```
git checkout newbranch
```

このブランチに加えた変更は，他のブランチには反映されない．例えば，ここで `git checkout master` コマンドによって master ブランチに切り替えると，変更したはずのファイルが元に戻っている．
なお，`git checkout` コマンドに `-b` オプションを付ければ，新しいブランチを作ってそのブランチに切り替える操作を一度にできる．

```
git checkout -b newbranch
```

このコマンドは，次の二つのコマンドと同じである．

```
git branch newbranch
git checkout newbranch
```

6.3 リモートリポジトリにブランチを作成

るブランチで行った変更は，`git push` によって他のブランチや最初のブランチである master には反映できない．ブランチはリモートリポジトリの対応するブランチに `git push` する．

```
git commit -a -m "コメント"
git push origin newbranch
```

この `git push` を `-u` オプションを付けて実行すれば，以後 `git push` コマンドを実行するときに引数の “origin newbranch” を省略できる．

6.4 ブランチのマージ

現在のブランチで行った変更を master ブランチに反映するには，master ブランチに切り替えてからそのブランチをマージする．リモートブランチの変更内容をマージする場合は，この前に `git pull` する．

```
git checkout master
git merge newbranch
```

これをリモートリポジトリにも反映するには，`git push` コマンドを使う．

```
git push
```

7 こういうときは

7.1 関係ないファイルを git add してしまった

```
git rm --cached ファイル …
```

7.2 git add 自体を取り消したい

```
git reset HEAD
```

7.3 関係ない（.gitignore に登録されていない）ファイルを削除したい

```
git clean -f
```

このとき必要なファイルまで消してしまう可能性があるので，事前に次のコマンドでチェックする．

```
git clean --dry-run
```

7.4 前の commit を捨ててもう一度 commit し直したい

```
git commit --amend -a -m "コメント"
```

7.5 ファイルを編集前に戻したい

```
rm ファイル
git checkout ファイル
```

7.6 全部のファイルを直前に commit したときに戻したい

```
git reset --hard HEAD
```

`--hard` オプションを付けると変更内容を取り戻すことはできないので注意すること．

7.7 現在の編集内容をそのままにして直前の commit を取り消したい

```
git reset --soft HEAD^
```

7.8 直前の commit 自体を取り消して現在の編集内容をその前に戻したい

```
git reset --hard HEAD^
```

7.9 直前の commit からの変更内容を見る

```
git diff
```

7.10 直前に commit した内容を見る

```
git show
```

7.11 履歴を見る

```
git log
```

-p オプションを付けると変更内容も表示する.