

バージョン管理システム(GIT)

Ver.201907

作成者：伊賀 将之

第1章 バージョン管理システム

本章の目的

- バージョン管理システムとは何かを知る
- バージョン管理システムの種類を知る
- SubversionとGitの利用イメージを知る

1.1 現在までの開発の問題点

- 以前のバージョンに戻すことが難しい
- 複数人で開発をすることが難しい
- 昔は以下のことを行っていた
 - プロジェクト単位で過去のバージョンのバックアップを取っておく
 - 複数人で開発したものを共用ファイルサーバで管理する
 - ⇒間違いが発生しやすい
 - ⇒ある部分はそのまま、ある一部の変更だけ元に戻すというようなことはできない
- このような背景から「バージョン管理システム」が生まれた

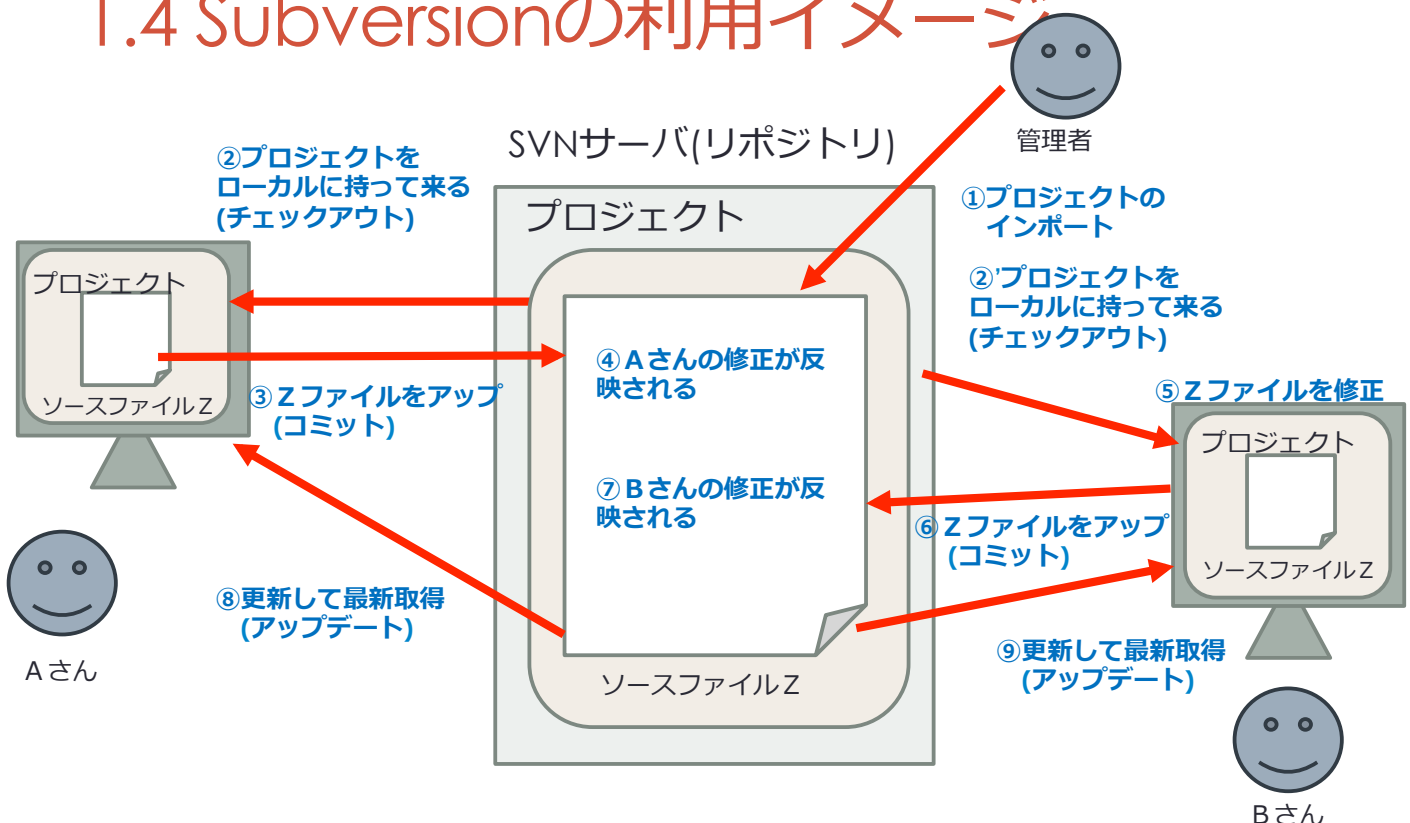
1.2 バージョン管理システム

- 一つのファイル(主にソースコード)やファイルの集合(プロジェクトなど)に対して時間とともに加えられていく変更を記録するシステム
- バージョン管理システムを使うと、以下のことができる
 - ファイルのバージョン管理
 - ファイルやプロジェクトを以前の状態まで戻す
 - 過去の変更履歴を比較する
 - 他の開発者との共同開発
 - 問題が起こっているかもしれないものを誰が最後に修正したか、誰がいつ問題を混入させたかを確認する
 - 複数人が同じファイルを修正した際に自動でマージ(統合)する
 - 複数人が全く同じ個所を修正した際に競合したことを知らせる

1.3 バージョン管理システムの種類(集中型)

- 集中型
 - プロジェクトを1つのサーバーで集中的に管理
 - (メリット)操作は分散型と比べて単純
 - (デメリット)サーバーがつぶれたら復旧が困難
- CVS(Concurrent Versions System)
 - 一昔前のスタンダード
- SVN(Subversion)
 - CVSをより使いやすくしたもの
- TFS(Team FoundationServer)
 - Microsoft系のツールで使われているもの

1.4 Subversionの利用イメージ



1.5 バージョン管理システムの種類(分散型)

・分散型

- ・ 変更履歴を含んだプロジェクトのクローンを作り管理
- ・ (メリット) 1つのPCが故障しても他のPCにクローンがあるため復旧が比較的簡単
- ・ (デメリット) 操作は集中型と比べて複雑、慣れが必要

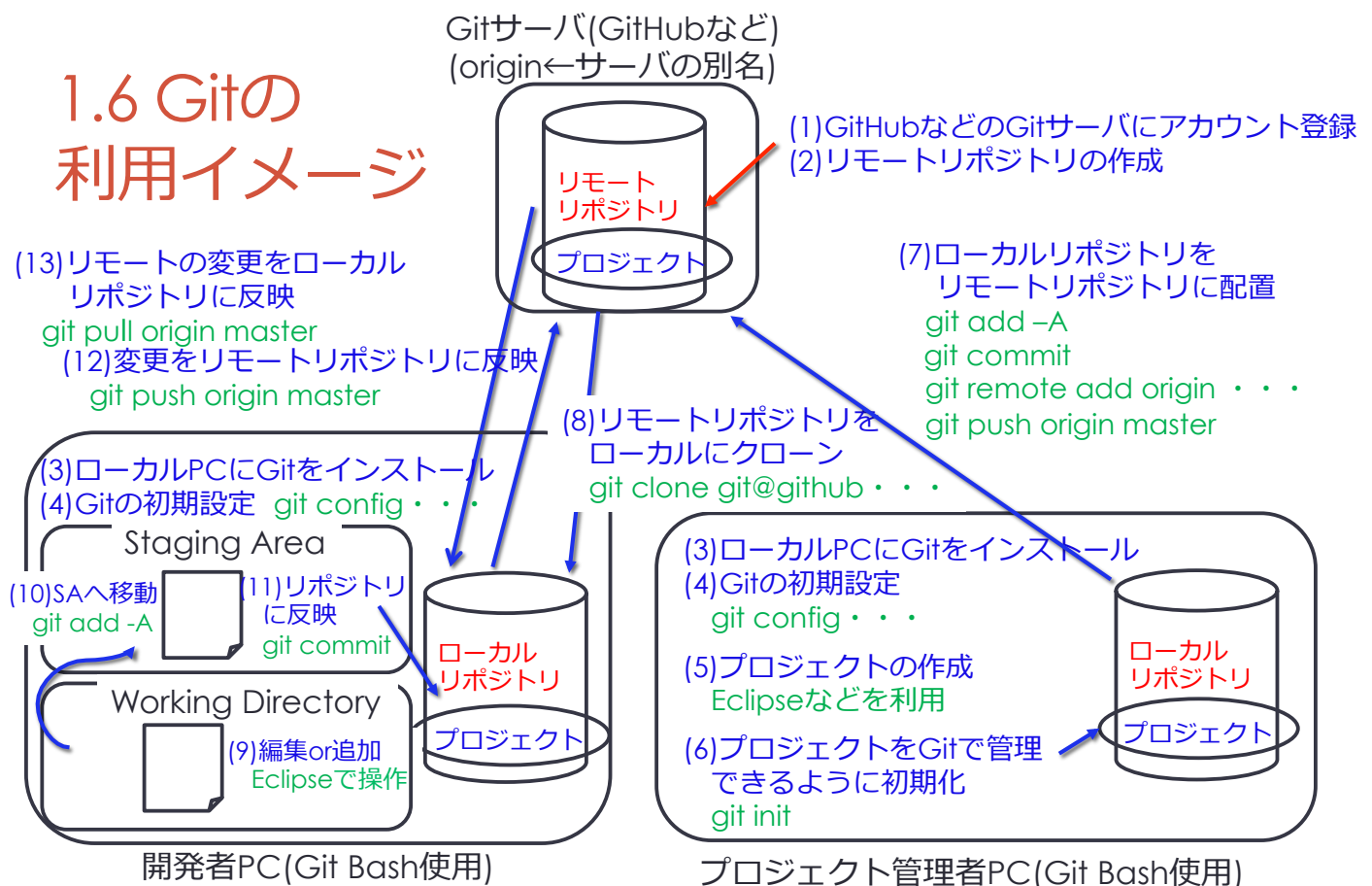
・ Git

- ・ Linuxのソースコードを管理するために作られた
- ・ 最近流行り始め、現場で多く使われている

・ Mercurial

- ・ 考え方使い方はほぼGitと同じ、Pythonで実装されている
- ・ そのためか、Pythonを使うプロジェクトで良く使われる

1.6 Gitの利用イメージ



第2章 GITの基本

本章の目的

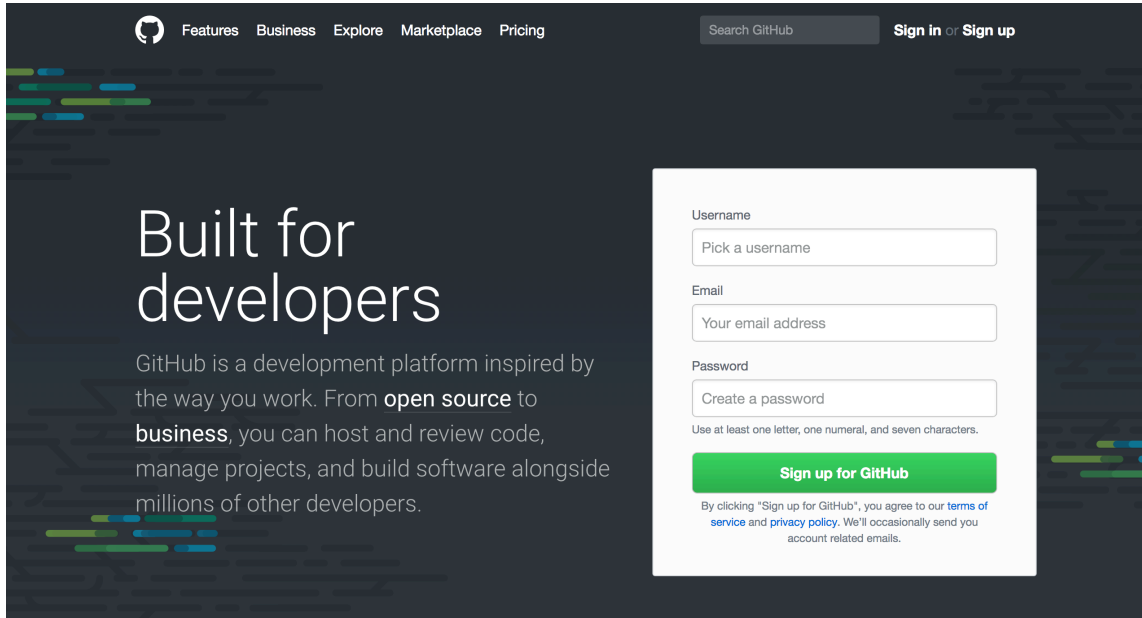
- GitHub(Gitサーバ)にアカウント登録する
- Git Bash(Gitクライアント)をインストールする
- 上記の基本的な使い方を知る

2.1 (1)GitHubなどのGitサーバに アカウント登録

- GitHubとは
 - Gitの仕組みを利用したソフトウェア開発プロジェクトのための共有ウェブサービス
 - 全世界に公開するpublicリポジトリと限られた人しか使えないprivateリポジトリ(有料)を作ることができる
- GitHubの他にも以下のサービスが存在する(2017年時点の情報)
 - BitBucket (privateリポジトリ無制限、5名まで無料)
 - Assembla (privateリポジトリ1つ、3名まで無料)
 - GitLab (MITライセンスのため無料?)
 - GitBucket (インストール型、無料)
 - Phabricator (インストール型、無料)

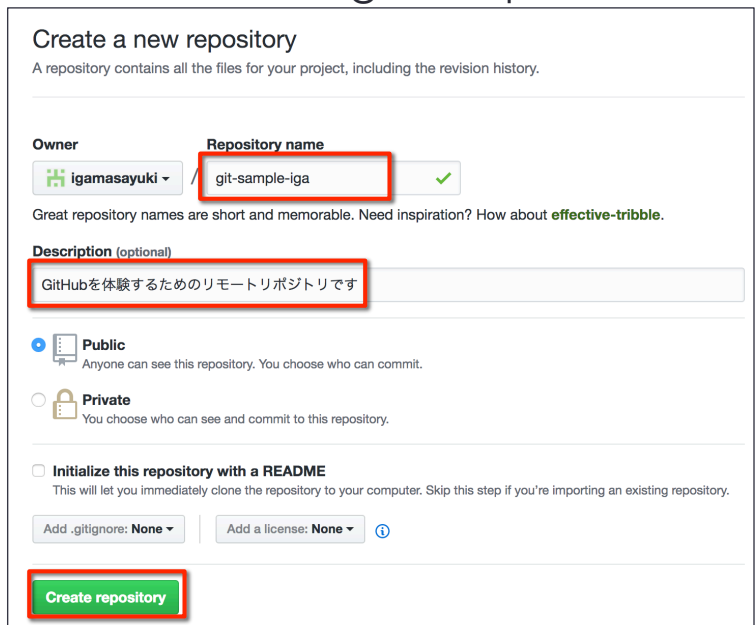
2.1 (1) GitHubなどのGitサーバにアカウント登録

- <https://github.com/> にアクセスし、右上の「Sign up」から会社のメールアドレスでアカウント登録をしてください



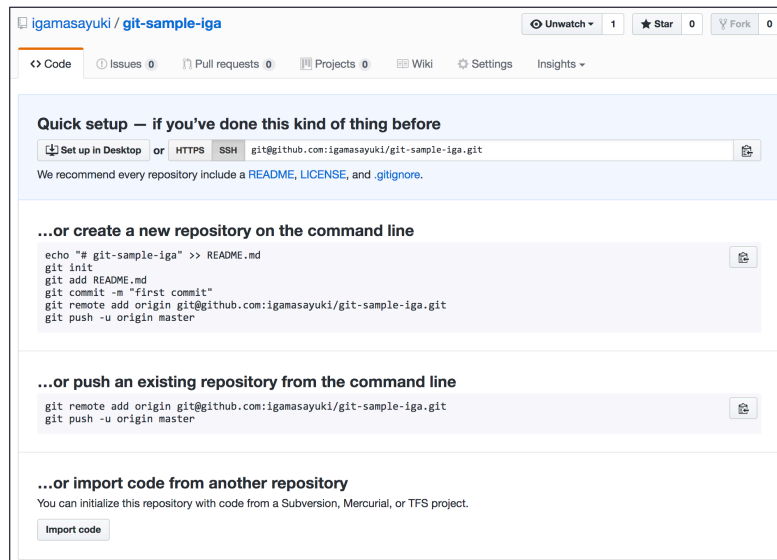
2.2 (2) リモートリポジトリの作成

- GitHubログイン後「Start a project」または「New repository」をクリックし、以下のように「git-sample」リポジトリを作成する
- Repository name に「git-sample-iga」と記載
(igaの部分は自分の名前)
- Descriptionに説明を記載
- 「Create Repository」をクリック



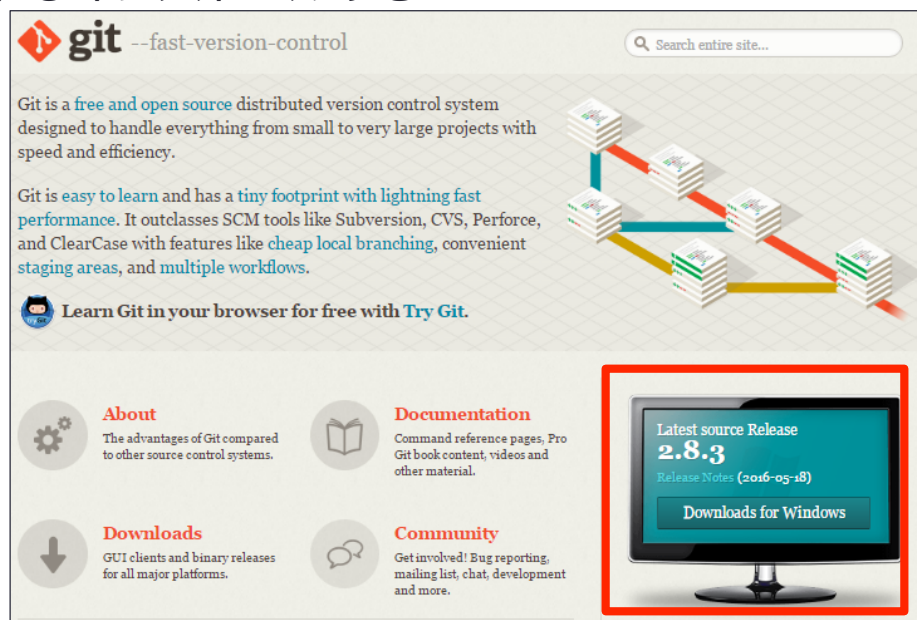
2.2 (2) リモートリポジトリの作成

- publicなリモートリポジトリが作成される
- このリモートリポジトリにプロジェクトをpushする方法が記載されている



2.3 (3) ローカルPCにGitをインストール

- <https://git-scm.com/>へアクセスしWindows版のGitをダウンロードしインストールする



(参考)Gitを学習するためのサイト

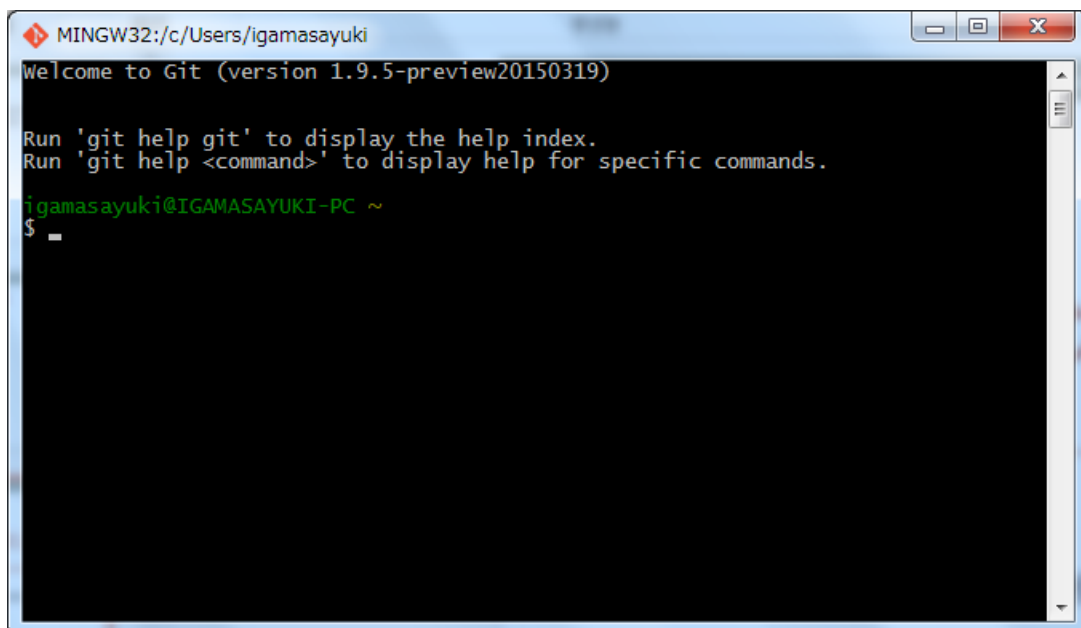
- <https://git-scm.com/> ⇒ Pro Gitリンクをクリック ⇒ 日本語リンクでProGitという書籍の中身を見ることができる



- LearnGitBranching
 - <http://k.swd.cc/learnGitBranching-ja/>
 - 簡単なコマンドをWeb上で練習することが可能なサイト
 - (本テキストでは出てこないコマンドもたくさん学べます)

2.4 (4)Gitの初期設定

- ダウンロードしたGit Bashを起動する



2.4 (4)Gitの初期設定

- 個人識別情報を設定する
 - ユーザ名を設定
 - `$ git config --global user.name "igamasayuki"`
 - ユーザメールアドレスを設定
 - `$ git config --global user.email "iga@example.com"`
- 設定を確認する
 - `$ git config --list`
 - 以下の行が存在していることを確認する
 - `user.name=igamasayuki`
 - `user.email=iga@example.com`
- 以降のスライドはコマンド使用例の紹介

2.4 (4)Gitの初期設定

- GitHubにアクセスするために・・・
- 秘密鍵・公開鍵のペアを生成する
 - `$ ssh-keygen -t rsa -C "igamasayuki@example.com"`
 - 実行すると3回入力を求められるが全てそのままEnterを押す
 - `~/.ssh/id_rsa.pub`という名前のファイルが公開鍵
- 公開鍵情報をGitHubに登録する
 - `$ less ~/.ssh/id_rsa.pub`
 - 上記で表示された以下のような内容をGitHubの「SSH and GPG keys」→「New SSH key」に登録する(Titleは[company pc]などにする)

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQBAQC97wOjfl8XGGnRgApHFb+Gvl2SmHQ8
KgQTMVwTIGOlivr6j395EESeHxU9m+Kyj3vqYqkZocFtikMVcfCX4T2AmWBJIZirYd4UQl
ny017OubWAYCmiA3geLJ99Gh6+5TNQm1RILq49fYqhz1ZVcr1AR5pLTkkfSzAoLytO5y
a1Zjj0hboA8iRZfpDcSYLff/eiTzL21dy6H0wAuKWkmXilfJBPN/
DUOxx6CURnhp8BHiiMmjsvjO8kmcH+uEKIXQEt9LlpL+Vml4TYIOmB5qatjo24mm+Np7j
2rCUAkuyCFDzPL/+PKOnF7ndLIEnRxvN8NNQNHaTBkuYKS7IBwZ
igamasayuki@igamasayuki.local
```

※GitBucket使用の場合は上記手順の代わりに以下を1回だけ実行する

```
git config --global http.sslVerify false
```

↑SSHの鍵がなくてもpushできるように設定変更

2.5 (5)Eclipseプロジェクトを作成

- EclipseまたはSTSで「git-sample-iga」(iga部分は自分の名前)という名前の「Dynamic Web Project」または「Spring starter project」または「Maven project」を作成する

2.6 (6)プロジェクトをGitで管理できるように初期化

- 前ページで作ったEclipseプロジェクトに移動
 - `$ cd /c/env/workspace/git-sample-iga` ←Eclipseの場合
 - `$ cd /c/env/springworkspace/git-sample-iga` ←STSの場合
 - (または、上記のディレクトリをWindowsのエクスプローラで開き、右クリック⇒「Git Bash」でも同じことができる)
- プロジェクトをGitで管理し始める
 - `$ git init`
 - 結果 :
Initialized empty Git repository in c:/env/workspace/git-sample/.git/
 - この操作でプロジェクトが「Working Directory」で管理される

2.6 (6)プロジェクトをGitで管理できるように初期化

- gitで管理しないファイル(個々のPCの設定情報などが書かれているファイル)を登録する
 - \$ vi .gitignore (←viエディタが起動する。使い方は次ページ参照)
 - 以下を記入する

```
.settings
/build/
.classpath
.project
```

←Dynamic Web Projectの場合

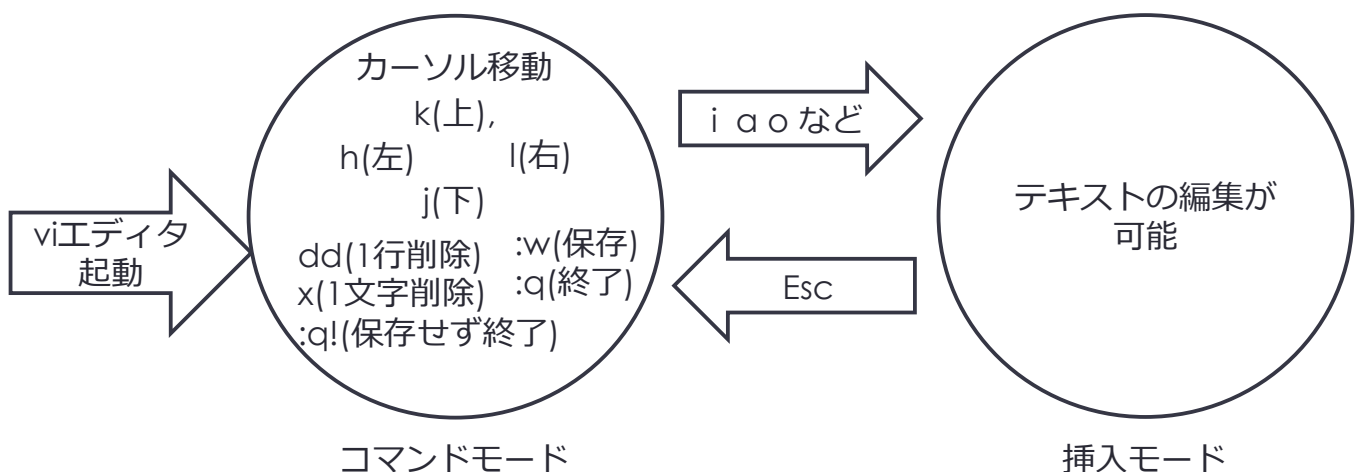
Maven Projectおよび
Spring Starter Projectの場合→
(既に存在している場合は
書き換える)

```
.settings
/target/
.classpath
.project
```

ディレクトリ名 or ファイル名	ディレクトリやファイルの説明
.settings	Eclipseの設定情報が入っているディレクトリ
/build/ or /target/	コンパイル後のclassファイルが格納されるディレクトリ
.classpath	Buildパス情報が書かれているファイル
.project	Eclipseプロジェクトの設定情報が書かれているファイル

(参考)viエディタ

- UNIXやLinux と呼ばれるOSに標準で入っているエディタソフトウェア
- Windowsのエディタとは操作方法が大きく異なる
- コマンドモードと挿入モードの2種類のモードからなる



2.7 (7)ローカルリポジトリを リモートリポジトリに配置

- 今作成したローカルリポジトリをリモートに配置する
- 現在の編集状態を確認
 - `$ git status`
- 編集ファイルをStaging Areaへ移動
 - `$ git add -A`
- 編集ファイルをローカルリポジトリへ反映
 - `$ git commit`
- viエディタが起動するためコミットコメントの入力
 - 次スライド参照

(参考)viエディタでコミットコメント記入

- コミットコメントにはわかりやすく、「新規にプロジェクトを作成」と書く
- 次回以降はどのような修正を行ったのかを詳しく書く(詳細は章末に記載)
- 記入後、viエディタで保存して(`:w;q`)、コミット(ローカルリポジトリへ反映)完了
- これらの作業で作業履歴が記録される

2.7 (7)ローカルリポジトリを リモートリポジトリに配置

- GitHubで作成したリモートリポジトリまでのURLをローカルリポジトリに登録
 - `$ git remote add origin git@github.com:igamasayuki/git-sample-iga.git`
 - originとは今登録したリモートリポジトリの別名で慣例的な名前(わかりやすくgithubという別名にしても良い)
 - URLはGitHubからコピーしてくる
- 登録したリモートリポジトリと別名を確認
 - `$ git remote -v`
- ローカルリポジトリをリモートリポジトリに配置
 - `$ git push -u origin master`
 - ※-uは次回からoriginやmasterを省略できるようにするオプション

2.8 (8)リモートリポジトリを ローカルにクローン

- 今作成したEclipseまたはSTSのプロジェクトを削除する
 - リモートリポジトリにあるプロジェクトをクローンするため
- Git Bashを起動し、上記のワークスペースへ移動する
 - `$ cd /c/env/workspace/` ←Eclipseの場合
 - `$ cd /c/env/springworkspace/` ←STSの場合
 - (または、上記のディレクトリをWindowsのエクスプローラで開き、右クリック⇒「Git Bash」でも同じことができる)
- リモートリポジトリをクローンして持ってくる
 - `$ git clone git@github.com:igamasayuki/git-sample-iga.git`
 - ※URLはGitHubからコピーする
- クローンして持ってきたプロジェクトをインポートする

(参考)コミットコメントの書き方

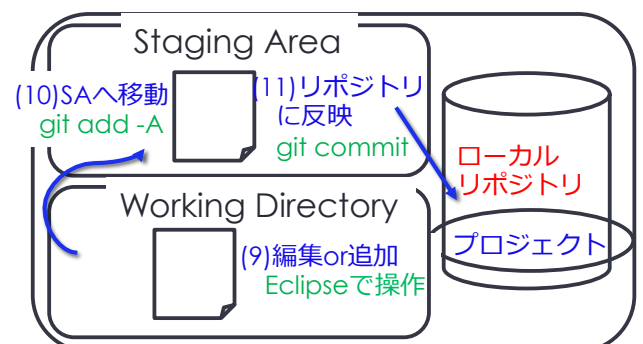
- コミットコメントは非常に大切。開発を早く進めようとしてコメントを疎かにしてしまうエンジニアが多い
- コミットメッセージはわかりやすく
 - コードを見ただけではわからない意図を書く
 - 5W1Hの、Whyを書く
 - Gitの機能でWho Whenは自動的に書き込まれる
 - What Where Howは変更内容から確認できる
 - 残るWhyは自動的に書いてくれないのでWhyを主軸に書く
 - 書きにくい場合は変更前の挙動と変更後の挙動をBefore afterで書いても良いし、必要であれば参考にしたURLを記載しても良い
- 悪い例
 - 「Second commit」「Third commit」「〇〇クラスを変更した」「〇〇メソッドを使うようにした」「とりあえずコミットしておく」「ユーザー登録機能」「今日の分をコミット」
- 良い例
 - 「〇〇機能を実装した」「××機能を削除した」「書き間違いを修正」「〇〇だったのを元に戻した」「〇〇の不具合に対応した」

2.9 (9)Eclipseで編集or追加

2.10 (10)Staging Areaへ移動

2.11 (11)ローカルリポジトリに反映

- Eclipseで編集or追加
 - 今まで通りEclipseを使用して開発を進める
- Staging Area へ移動
 - `git add -A`
- ローカルリポジトリに反映
 - `git commit`
 - ※コミットコメントをきちんと書く



2.12 (12)変更をリモートリポジトリに反映

- 変更した内容がローカルリポジトリに反映されているため、その差分をリモートリポジトリに反映させる
 - `git push origin master`

2.13 (13)リモートリポジトリの変更をローカルリポジトリに反映

- 変更をpushしたリモートリポジトリの内容の差分をローカルリポジトリに取り込む
 - `git pull origin master`

(参考)その後の開発 その他のコマンド

- 以降、IDEで開発を行い、ひとつのことが完了したら「`git add -A`」「`git commit`」を繰り返し、編集履歴をつけていく
- そして1機能完成程度の粒度で「`git push`」コマンドを使い、リモートリポジトリに反映させる
- Gitのその他の良く使うコマンド
 - `git diff` (前回のコミットとの差分を確認)
 - `git log` (コミット履歴閲覧)
 - `git help <verb>` (gitコマンドのヘルプを表示)
 - `git <verb> -help` (gitコマンドのヘルプを表示)
- その他、必要な時にマニュアルを見ながら操作を行う

(参考)コミットタイミング

- だいたいひとつのことが終わったらコメントつけてコミットしておく。
- 悪い例
 - 一機能終わったらcommit ⇒一機能終わったらやることはpush
 - コンパイルエラーや環境エラーがある状態でcommitやpushはもっと×
- 良い例
 - 入力値チェック終了後にcommit
 - モックから1つの画面を作ったらcommit
 - コントローラーと画面の疎通確認ができればcommit
 - データベース接続確認ができればcommit
 - 開発中に他の開発依頼が来て、他のブランチに切り替えるために今のコードは途中だが、保存のために一時的にcommit

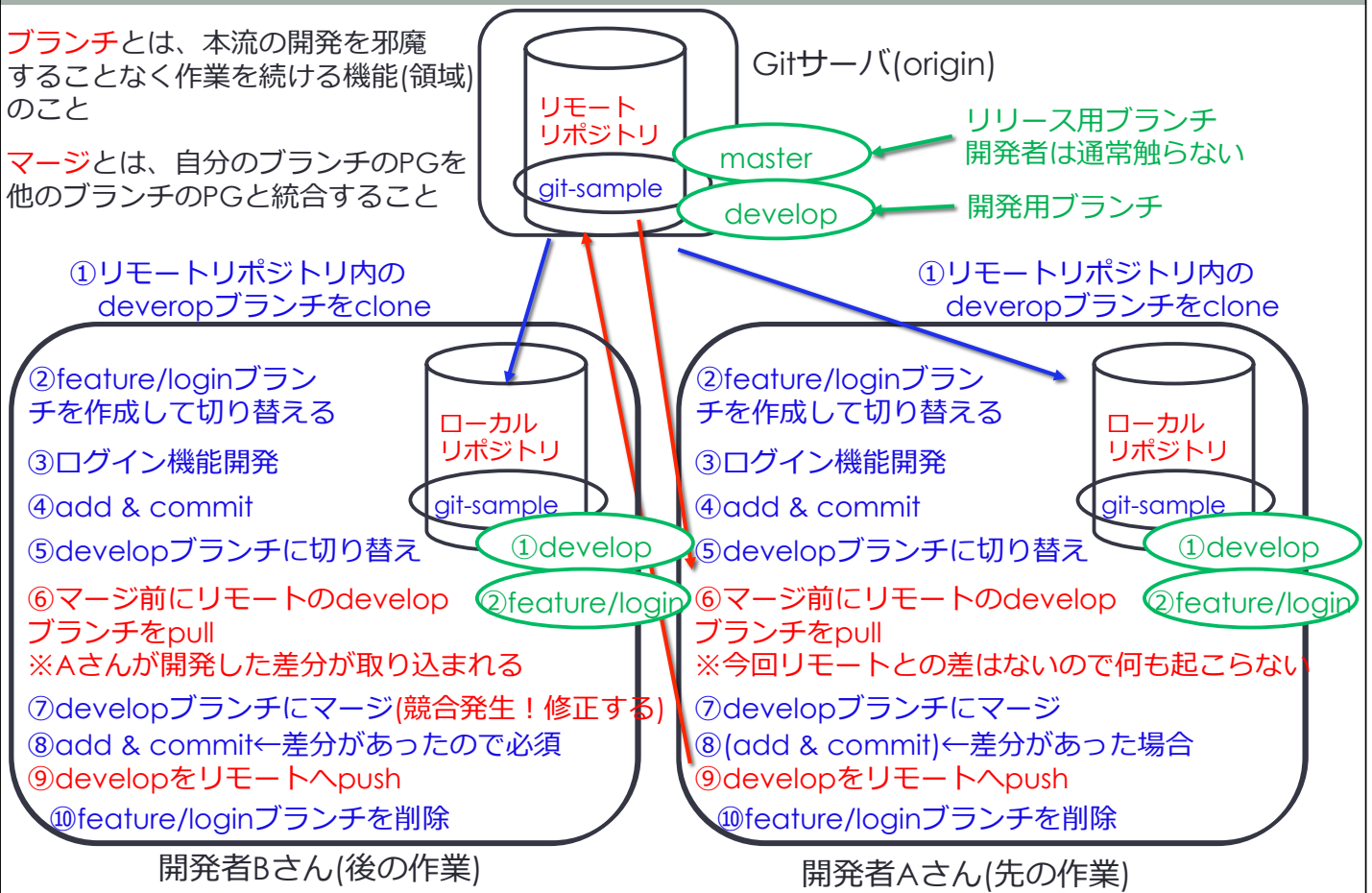
第3章 GITシェアプログラミング

本章の目的

- チーム開発を行う際のGitの使い方を理解する

ブランチとは、本流の開発を邪魔することなく作業を続ける機能(領域)のこと

マージとは、自分のブランチのPGを他のブランチのPGと統合すること



最初の手順(2人1ペアになります)

※サポートされたい人同士のペアは避けます

- Aさん(管理者の役割)が新規にリモートリポジトリを作成する
- AさんがEclipse(またはSTS)にてプロジェクトを作成する
- 今作成したプロジェクト内でgit bashを起動し、git initの実行や.gitignoreなどを作成する
- プロジェクトに1つのクラスを作成する(Carクラスなど)
- 今作成したプロジェクトをリモートリポジトリにpushする(add & commitも忘れずに!)
- その後、リモートリポジトリでdevelopブランチを作成する
- Aさんは、今作成したローカルのプロジェクトをワークスペース上から削除する
 - あとでもう一度cloneするため1度削除します
- AさんがGitHub上でBさんがpushできるようにCollaboratorsにBさんを追加する
「settings」タブ→「Collaborators」リンク
Bさんに承認依頼メールがくるのでBさんは承認する
- 2人ともc:\env\springworkspaceをエクスプローラで開き、Git Bashを起動する

AさんBさんが並行してやる作業

- ①リモートリポジトリのdevelopブランチをclone
 - `$ git clone -b develop git@github.com:igamasayuki/git-sample-iga.git`
 - (URLはGitHubからコピー)
 - `$ cd git-sample-iga` ←cloneしてきたgit-sampleの中へ入る
 - `$ git branch` ←branchの確認
 - Eclipse(STS)からプロジェクトをインポートする
- ②branchの作成と切り替え
 - `$ git branch feature/login` ←branchの作成
 - `$ git branch` ←branchの確認
 - `$ git checkout feature/login` ←branchの切り替え
 - `$ git branch` ←branchの確認
 - ※ブランチの作成と切替を一気に行う方法
 - `$ git checkout -b feature/login`
- ③ログイン機能開発
 - 実際に割り当てられた担当分の機能開発を行う
 - ※この際、AさんとBさんは同じ行を修正すること。例えばCarクラスのprivate int speed 部分など)
 - `$ git status` ←ファイルのステータスを確認
- ④add & commit
 - `$ git add -A`
 - `$ git commit`
- ⑤developブランチに切り替え
 - `$ git checkout develop`
 - `$ git branch` ←切り替わったことを確認

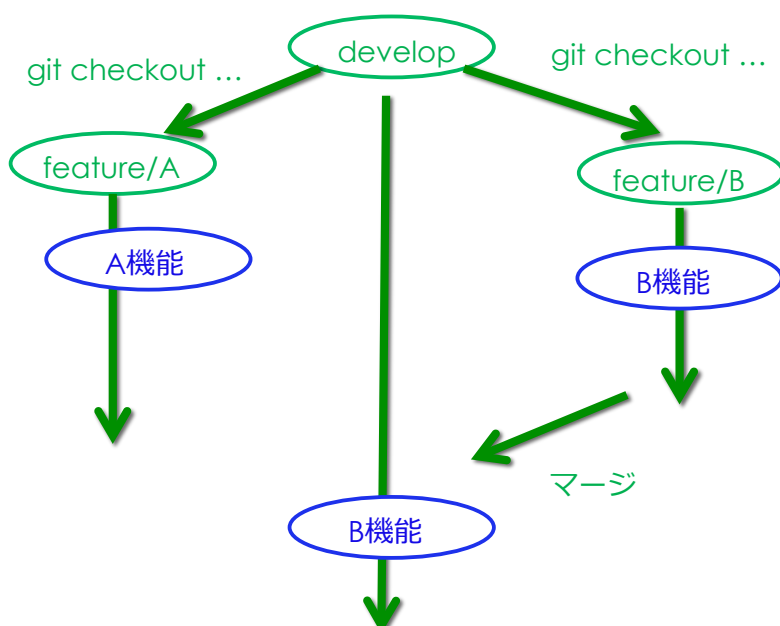
Aさんの作業(先にやる)

- Bさんは作業をストップしてください
- ⑥マージ前にリモートのdevelopをpull
 - `$ git pull origin develop` ←git pull リモート名 ブランチ名
 - ※今回はリモートの差分が無いため競合が起こらない
- ⑦developにマージ
 - `$ git merge feature/login` ←マージするdevelopブランチで行う
- ⑧(add & commit)←今回は競合が発生しないため必要ない
- ⑨developをリモートへpush
 - `$ git push origin develop`
 - ※他の人が先にpushしていたらもう一度pullしてからpush
 - Github上にきちんと追加(or修正)したものが上がっているか確認すること！
- ⑩feature/loginブランチを削除
 - `$ git branch -d feature/login`
 - `$ git branch` ←削除されたか確認

Bさんの作業(後にやる)

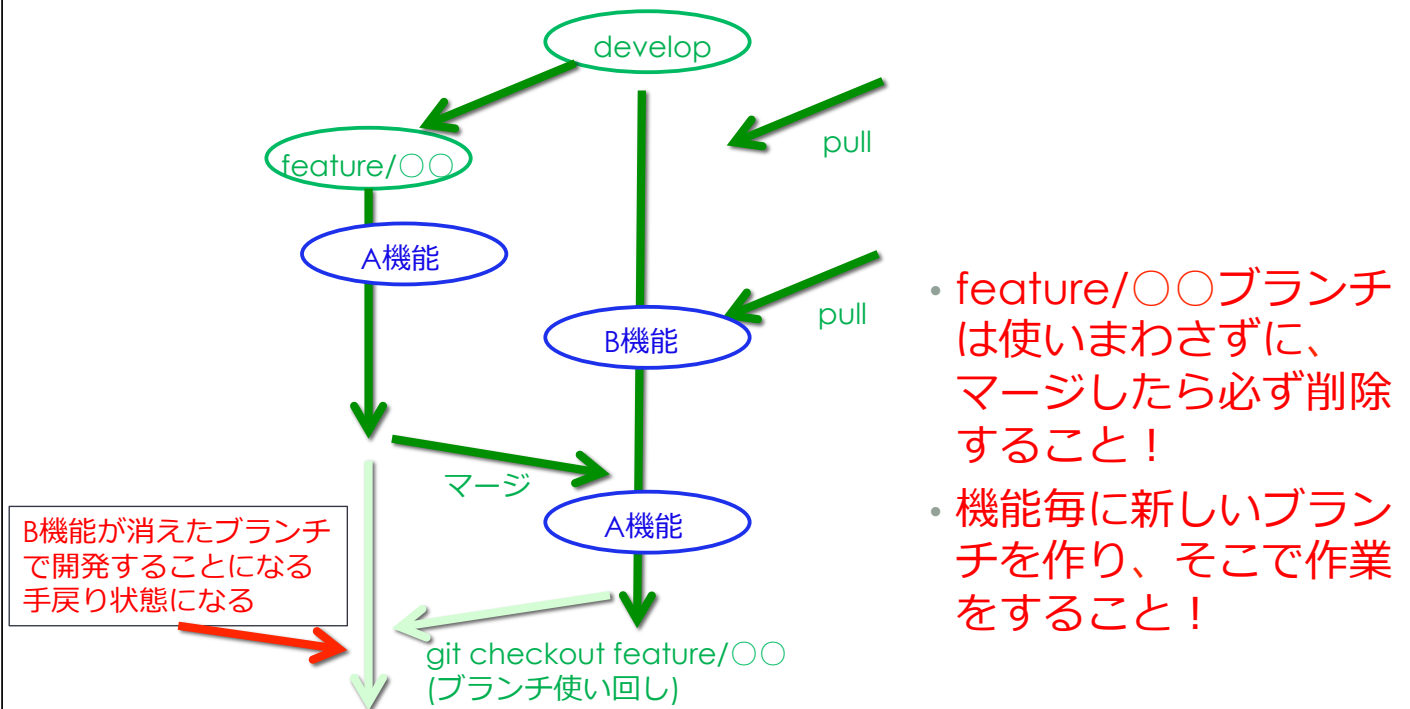
- Aさんの後にBさんが行う手順
- ⑥マージ前にリモートのdevelopをpull
 - \$ git pull origin develop ←git pull リモート名 ブランチ名
 - ※Aさんの開発した差分が取り込まれる
- ⑦developにマージ
 - \$ git merge feature/login ←マージするdevelopブランチで行う
 - ※ここで競合が発生するのでBさんが競合を修正する
- ⑧～⑩までは同じ手順
- 後は本日やった内容を2人で一緒に何回も行いGitでのチーム開発に慣れる

(参考)なぜブランチを切るの？



- A機能開発途中にB機能開発の依頼が来た！
- developブランチで開発しているとA機能の開発とB機能の開発を同時にやることになるのでごちゃごちゃになってしまう
- ブランチを切っておけばA機能の開発を中断しB機能のみの開発に専念できる

(参考)feature/〇〇ブランチは 使い回さないこと！



(参考)競合を起こす確率を減らすために

- チームのメンバーが近くにいる時は、pushしたい人は「今からpull/pushします！」と声をかけて、自分がpushするまで他の人にpushするのを待ってもらう
- コントローラやサービスをユースケース毎に作成し、一つのクラスの中を書くプログラムを1つの役割のみにする
 - ※ドメインやリポジトリは今まで通り1テーブルにつき1クラス

総合演習

- 2人ペアで前回作成した掲示板を作成してください
- この際、掲示板課題資料にある「サブレットで設計したクラス図(ユースケースごとに1つのコントローラ)」が競合が起これにくい設計になっているのでそれを参考にします
- コーディング技術の向上ではなく、gitに慣れることが目的のため、一番複雑なRepository内のコードやapplication.ymlは以前作成したものからコピーしてもってきてOKです

(参考)git add / git commitについて

- git addの様々な方法
 - 1ファイルのみaddする方法
 - `$ git add filename.txt`
 - 全てのaddされていないファイルをaddする方法
 - `$ git add *`
 - `$ git add .`
- git commitの方法
 - 1ファイルのみcommitする方法
 - `$ git commit filename.txt`
 - 全てのcommitされていないファイルをcommitする方法
 - `$ git commit`
 - commitコメントをviエディタを使わずに書く方法
 - `$ git commit -m "ここにコミットコメントを書く"`

(参考)作業のやり直し方法

- add前(ステージングされる前)の取り消し方法
 - ファイル修正後、git statusするとやり方が書かれている

```
igamasayuki:git-test igamasayuki$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

       modified:   src/test/Test.java

no changes added to commit (use "git add" and/or "git commit -a")
```

- 1ファイルのみ取り消し
 - \$ git checkout filename.txt
- 全てのファイル
 - \$ git checkout .

(参考)作業のやり直し方法

- add(コミット前)の取り消し方法
 - git add後、git statusするとやり方が書かれている

```
igamasayuki:git-test igamasayuki$ git add -A
igamasayuki:git-test igamasayuki$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

       modified:   src/test/Test.java
```

- 1ファイルのみ
 - \$ git reset HEAD filename.txt
- 全てのファイル
 - \$ git reset HEAD .
 - \$ git reset HEAD *

(参考)作業のやり直し方法

- commitのやり直し方法
 - 直近のコミットコメントを修正する方法
 - `$ git commit --amend` ←コミットのやり直し&コメント上書き修正
 - 直近commitにファイルを追加する方法
 - `$ git commit -m 'initial commit'` ←コミット
 - `$ git add forgotten_file.txt` ←コミットに含むのを忘れてたファイルをadd
 - `$ git commit --amend` ←コミットのやり直し
 - ※ 2 番目のコミットが最初のコミットの結果を上書きする

(参考)作業のやり直し方法

- `git log` ←コミット履歴を表示する

```
igamasayuki:git-test igamasayuki$ git log
commit 44de9ee0a62e1453ea1826b16c849d643aadf688 ←3番目のコミットID
Author: Masayuki IGA <igamasayuki@gmail.com>
Date: Fri Jun 15 08:25:39 2018 +0900
```

test追加

```
commit d18930a648837705c0e6e8299ccb3743a178c61b ←2番目のコミットID
Author: Masayuki IGA <igamasayuki@gmail.com>
Date: Fri Jun 15 08:05:29 2018 +0900
```

新規追加 コメント変更

```
commit 3062fb018aee6a825aa562e34723cf6755f5d0db ←最初のコミットID(今回ここに戻したい)
Author: Masayuki IGA <igamasayuki@gmail.com>
Date: Fri Jun 15 07:47:12 2018 +0900
```

新規プロジェクト

(参考)作業のやり直し方法

- 特定のcommitまで戻す方法
 - `git reset --hard` 戻したいコミットID

```
igamasayuki:git-test igamasayuki$ git reset --hard 3062fb018aee6a825aa562e34723cf6755f5d0db
HEAD is now at 3062fb0 新規プロジェクト
```

```
igamasayuki:git-test igamasayuki$ git log
commit 3062fb018aee6a825aa562e34723cf6755f5d0db ←最初のコミット段階に戻っている
Author: Masayuki IGA <igamasayuki@gmail.com>
Date: Fri Jun 15 07:47:12 2018 +0900
```

新規プロジェクト