

付録 Git コマンド解説

# 付録 Git コマンド解説

---

## Git コマンド解説

### init

このコマンドは、リポジトリを新たに作成するためのものです。また、既存のリポジトリを再初期化することもできます。

このコマンドを実行すると「.git」というサブディレクトリが作成され、その中には Git が動作するのに必要な全ての情報が置かれます。このコマンドは、同じリポジトリに対して再度実行されても安全で、上書きされることはありません。

**git init** : Git のリポジトリを新たに作成する

[書式] git init [ディレクトリ] [オプション]

[主なオプション]

--bare            最小限のリポジトリを作成する。  
--shared           リポジトリが複数のユーザで共有されることを指定する。  
                  このオプションは、同じグループに属するユーザがそのリポジトリに  
                  push することを許可する。

実際の使い方を見えます。

### ディレクトリを指定しない

これは、ディレクトリを指定しないで、/c/gitwork ディレクトリに移動してコマンドを実行する例です。

```
$ mkdir /c/gitwork
$ cd /c/gitwork
$ git init
Initialized empty Git repository in C:/gitwork/.git/
$ ls /c/gitwork/.git
config  description  HEAD  hooks/  info/  objects/  refs/
```

ディレクトリを指定しなければ、コマンドを実行したディレクトリの直下にリポジトリのための「.git」というサブディレクトリが作成されます。

## ディレクトリを指定する

ディレクトリを指定してコマンドを実行する例を見えます。ディレクトリを指定する場合は、指定したディレクトリにリポジトリのための「.git」というサブディレクトリが作成されます。

```
$ cd /c/temp
$ mkdir /c/gitwork
$ git init /c/gitwork
Initialized empty Git repository in C:/gitwork/.git/
$ ls /c/gitwork/.git
config  description  HEAD  hooks/  info/  objects/  refs/
```

この例では、/c/temp ディレクトリにて、/c/gitwork ディレクトリを指定することで、リポジトリを/c/temp 下ではなく/c/gitwork 下に作成します。

## 共有リポジトリを作成する

Git サーバの/c/gitrepository/sample.git ディレクトリに共有リポジトリを作成する例です。

```
$ mkdir /c/gitrepository
$ mkdir /c/gitrepository/sample.git
$ cd /c/gitrepository/sample.git
$ git init --bare --shared
Initialized empty Git repository in C:/gitrepository/sample.git
$ ls /c/gitrepository/sample.git
config  description  HEAD  hooks/  info/  objects/  refs/
```

複数のユーザで共有できるリモートリポジトリを作成するために、コマンド `git init` に `--bare` `-share` オプションを指定しています。

## clone

このコマンドは、リポジトリを新しいディレクトリにクローンするためのものです。



クローンとは、指定されたディレクトリに元のリポジトリと同じものを複製することです。

**git clone** : リポジトリを新しいディレクトリに複製する

〔書式〕 `git clone` [リポジトリ] [ディレクトリ]

実際の使い方を見えます。

## リモートリポジトリをクローンする場合

リモートリポジトリとディレクトリを指定してコマンドを実行する場合は、下記ようになります。

```
git clone //KU-HP/sample.git copy
```

これはリポジトリに//KU-HP/sample.git、ディレクトリに copy を指定した例です。

```
$ mkdir copy
$ git clone //KU-HP/sample.git copy
Cloning into 'copy'...
done.
$ cd copy
$ git show-branch
[main] Function-01 has been implemented.
```

指定したディレクトリに移動して git show-branch コマンドを実行すると、リポジトリが作成されていることが確認できます。

## ローカルリポジトリをクローンする場合

ローカルリポジトリを同じローカルのディレクトリにクローンする場合は、下記ようになります。

```
git clone /c/gitwork copy2
```

これはリポジトリに/git/repository を指定し、ディレクトリに copy2 を指定した例です。

```
$ mkdir copy2
$ git clone /c/gitwork copy2
Cloning into 'copy2'...
done.
$ cd copy2
$ git show-branch
[main] Function-01 has been implemented.
```

指定したディレクトリに移動して git show-branch コマンドを実行すると、リポジトリが作成されていることが確認できます。

## ディレクトリを指定しない場合

リポジトリだけ指定してディレクトリを指定しない場合は、現在居るディレクトリの直下にリポジトリがクローンされます。

```
git clone [リポジトリ]
```

```
$ git clone //KU-HP/sample.git
Cloning into 'sample'...
done.
$ cd sample
$ git show-branch
[main] Function-01 has been implemented.
```

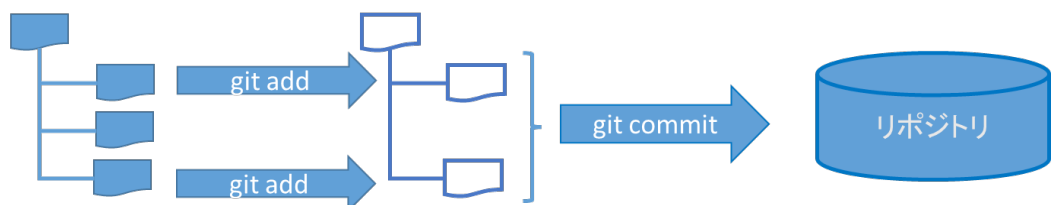
リモートリポジトリと同じ名前のディレクトリに移動して、git show-branch コマンドを実行するとリポジトリが作成されていることが確認できます。

## add

このコマンドは、ワーキングツリーの中でコンテンツ（ファイルなど）をインデックスに追加するためのものです。

ワーキング・ツリー

インデックス  
(ステージ・エリア)



インデックスに追加されたコンテンツは「ステージされた」(Staged) 状態となり、コミットの対象となります。ステージされたコンテンツ以外は、コミット対象となりません。

**git add** : コンテンツをインデックスに追加する

[書式] git add [オプション] [ファイル名またはディレクトリ名]

[主なオプション]

-i / --interactive 対話形式でインデックスにコンテンツを追加する。

実際の使い方を見えます。

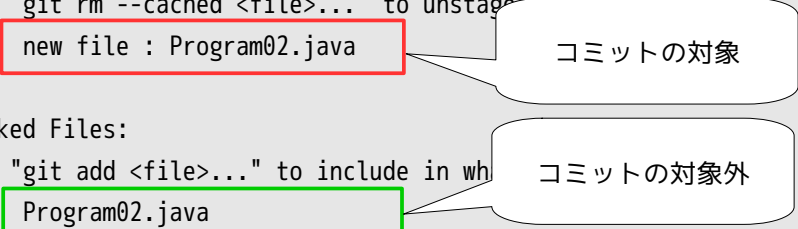
## ファイルをひとつだけ指定する場合

ワーキングツリーにある Program01.java ファイルを指定してコマンドを実行する場合は、下記のようになります。

```
git add Program01.java
```

例えば、ワーキングツリーに Program01.java と Program02.java の 2 つのファイルがあり、Program01.java のみ指定してコマンドを実行します。

```
$ cd /c/gitwork
$ git add Program01.java
$ git status
On branch main
Initial Commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file : Program02.java
Untracked Files:
  (use "git add <file>..." to include in what will be committed)
    Program02.java
```



The diagram illustrates the output of the `git status` command. It shows two files: `Program01.java` and `Program02.java`. `Program01.java` is the file specified in the `git add` command. `Program02.java` is a new file that has been added to the index (staged) but is not yet committed. The output shows `new file : Program02.java` under the 'Changes to be committed' section, which is highlighted with a red box and labeled 'コミットの対象' (Commit target). The 'Untracked Files' section shows `Program02.java` as a file that is not yet tracked by Git, which is highlighted with a green box and labeled 'コミットの対象外' (Not commit target).

指定した Program01.java だけがインデックスに追加され、コミットの対象になります。

## ファイルをまとめて指定する場合

ワーキングツリーにあるファイルをまとめて指定してコマンドを実行する場合は、下記のようになります。

```
git add .
```

拡張子が「.java」のファイルだけを指定することもできます。

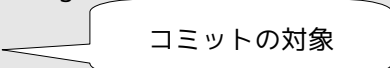
```
git add *.java
```

ディレクトリを指定することもできます。

```
git add dir/*.java
```

ワーキングツリーの src ディレクトリにある program01.c と program02.c をまとめて指定する場合は、下記のようになります。

```
$ git add src/*.c
$ git status
On branch main
Initial Commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file : src/program01.c
    new file : src/program02.c
```



拡張子「.c」を持つ program01.c と program02.c がインデックスに追加され、コミットの対象になります。

## ファイルを対話形式で指定する場合

オプション「-i」または「--interactive」を指定してコマンドを実行すると、対話形式でファイルの選択が可能となります。

オプションに「-i」を指定してコマンドを実行します。

```
$ git add -i
*** Commands ***
    1: status    2: update    3: revert    4: add untracked
    5: patch     6: diff      7: quit      8: help
what now>
```

対話形式なので、表示されたメニューの中から番号、あるいはそれぞれの項目の頭文字（1文字だけ青く表示されている）を入力するとコマンドが実行されます。

この例では、2つのファイルから Program01.java だけを選んでインデックスに追加しています。

```
*** Commands ***
    1: status    2: update    3: revert    4: add untracked
    5: patch     6: diff      7: quit      8: help
what now> 4
    1: Program01.java
    2: Program02.java
Add Untracked>> 1
    * 1: Program01.java
    2: Program02.java
Add Untracked>>
added 1 path

*** Commands ***
    1: status    2: update    3: revert    4: add untracked
    5: patch     6: diff      7: quit      8: help
what now> 1
      staged unstaged path
    1:  +6 / -0 nothing Program01.java

*** Commands ***
    1: status    2: update    3: revert    4: add untracked
    5: patch     6: diff      7: quit      8: help
what now>
```

4: add untracked を選択すると untracked のファイルが表示される

1: Program01.java を選択すると先頭に \* マークが付く

改行を押すとコマンドが実行される

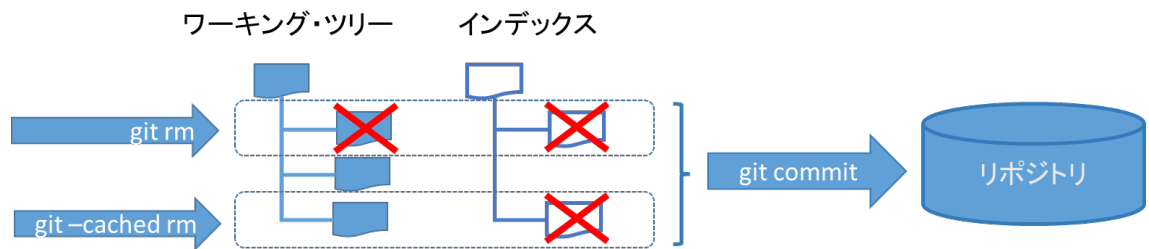
status を実行すると、インデックスに Program01.java が追加されている

対話形式でコマンドを選び、その対象になるファイルを番号で選択するという流れです。



## rm

このコマンドは、ワーキングツリーとインデックスからファイルを削除するためのものです。



このコマンドは、インデックスのみ、または、ワーキングツリーとインデックスの両方からファイルを削除します。ワーキングツリーのファイルを削除しインデックスには残す、ということはありません。

**git rm** : ワーキングツリーとインデックスからファイルを削除する

[書式] `git rm` [オプション] ファイル名あるいはディレクトリ名

[主なオプション]

- `--cached` インデックスからのみ削除する。
- `-r` ディレクトリ名が指定された場合、再帰的に削除する。
- `-f / --force` コミット前やインデックスに追加する前のファイルであっても削除する。

実際の使い方を見えます。

## インデックスのみ削除する場合

Program01.java のインデックスのみ削除するコマンドは、下記のようになります。

```
git rm --cached Program01.java
```


実行例を見てみます。まず、空のリポジトリを作成し、そこに git add コマンドで Program01.java と Program02.java をインデックスに登録します。

```
$ git init
Initialized empty Git repository in C:/gitwork/.git/
$ ls
Program01.java  Program02.java
$ git add *.java
$ git status
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   Program01.java
    new file:   Program02.java
```

この状態で git rm コマンドを実行します。

```
$ git rm --cached Program01.java
rm 'Program01.java'
$ git status
On branch main
Initial Commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   Program01.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  Program01.java
$ ls
Program01.java  Program02.java
```



Program01.java がインデックスから削除されます。しかし、ls コマンドで確認するとファイルは残っています。

## インデックスとファイルの両方を削除する場合

git rm コマンドはインデックスとファイルをまとめて削除することも可能です。

Program01.java がインデックスに追加されていれば、以下のコマンドでインデックスとファイルの両方が削除されます。

```
git rm Program01.java
```

もし、インデックス追加後にファイルの修正があった場合、修正されたファイルは削除できません。コミット前やインデックスに追加する前のファイルを削除する場合は、-f オプションを付けて実行します。

```
git rm -f Program01.java
```

例えば、インデックスに追加された 2 つのファイル Program01.java と Program02.java（まだコミットされていない）を削除する場合は、下記のようになります。

```
$ git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified: Program01.java
    modified: Program02.java
$ git rm -f Program01.java
rm 'Program01.java'
$ git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    deleted: Program01.java
    modified: Program02.java
$ ls
Program02.java
```

インデックスから削除

ワーキングツリーから削除

インデックスとワーキングツリーの両方から Program01.java が削除されています。この削除は、次のコミットを実行した時にリポジトリに反映されます。

## ディレクトリをまとめて削除する場合

git rm コマンドは、ディレクトリをまとめて削除することもできます。

```
git rm -r [ディレクトリ名]
```

例えば、src ディレクトリにある Program01.java と Program02.java がインデックスに追加されている場合を考えます。

```
$ git status
On branch main
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   src/Program01.java
    new file:   src/Program02.java
```

この状態で下記のコマンドを実行します。

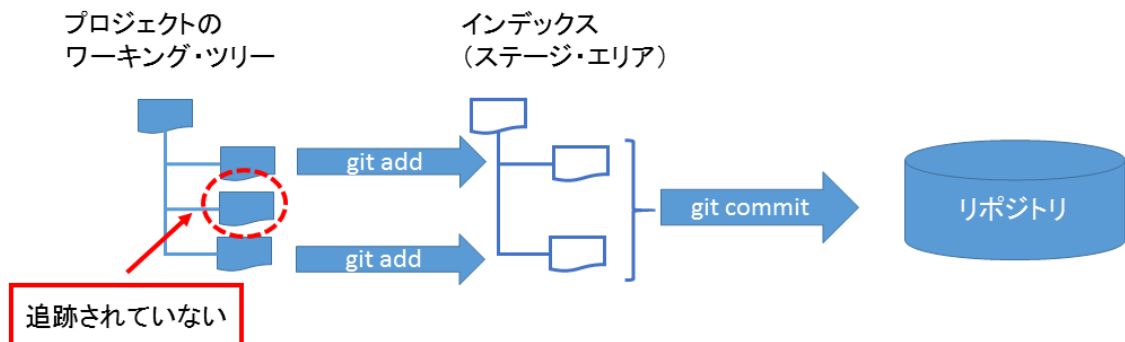
```
$ git rm -r --cached src
rm 'src/Program01.java'
rm 'src/Program02.java'
$ git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    src/
```

指定したディレクトリ内の全ファイルがインデックスから削除されます。

## status

このコマンドは、ワーキングツリーの状態を表示するためのもので、以下に示す条件でコンテンツのパスが表示されます。

- ・ ワーキングツリーに追跡されていないものがある
- ・ ワーキングツリーとインデックスに違いがある
- ・ コミットされたものとインデックスに違いがある



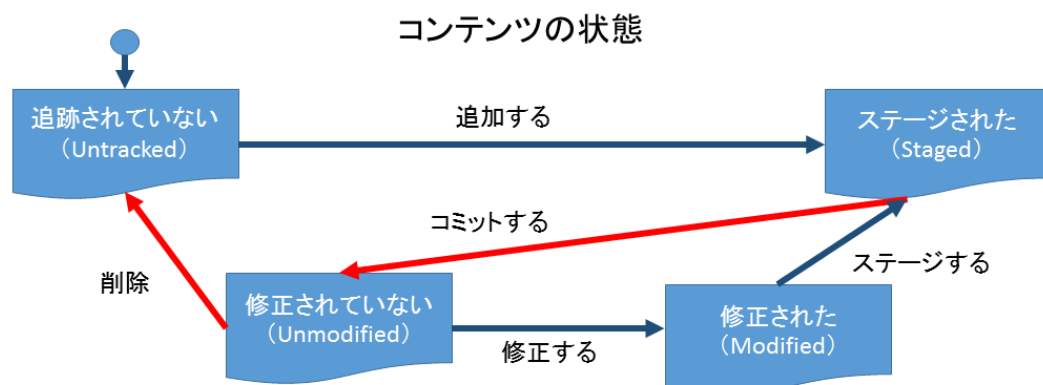
**git status** : ワーキングツリーの状態を表示する

[書式] git status [オプション]

[主なオプション]

-s / --short 短縮した形式で表示する。

このコマンドを実行すると、下記のようなコンテンツの状態が表示されます。



では、実際の使い方とどのように状態が表示されるかを見てみます。

## 追跡されていない場合

新規のファイル Program01.java をワーキングツリーに加えただけの状態で、git status コマンドを実行します。

```
$ ls
Program01.java
$ git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Program01.java

nothing added to commit but untracked files present (use "git add" to track)
```

表示されるのは、「追跡されていないファイル (Untracked Files:)」となります。

## 追加された場合

追跡されていないファイル (Untracked) をインデックスに追加した状態で、git status コマンドを実行します。

```
$ git add Program01.java
$ git status
On branch main
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   Program01.java
```

表示されるのは「コミットされるべき変更 (Changes to be committed)」の「新たなファイル (new file)」となります。

## コミットされた場合

ステージされたファイルをコミットした状態で、git status コマンドを実行します。

```
$ git commit -m "The first version of Program01.java"
[main (root-commit) 77cbc74] The first version of Program01.java
 1 file changed, 5 insertions(+)
 create mode 100644 Program01.java
$ git status
On branch main
nothing to commit, working tree clean
```

状態は「修正されていない (Unmodified)」になります。また、表示されるのは「nothing to commit, working tree clean」というメッセージとなります。つまり、「コミットするものは何もない。ワーキングツリーは修正されていない」ということです。

## 修正された場合

コミットしたファイル (Unmodified) を修正した状態で、git status コマンドを実行します。

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to include in what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified: Program01.java
no changes added to commit (use "git add" and/or "git commit -a")
```

表示されるのは「修正されたファイル (modified)」です。Git は、コミットされた後、ファイルが修正されたかどうかをチェックしています。

## ステージされた場合

修正されたファイル (Modified) がステージされた状態で、git status コマンドを実行します。

```
$ git add Program01.java
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified: Program01.java
```

表示されるのは「コミットされるべき変更 (Changes to be committed)」の「修正された (modified)」となります。

## 削除された場合

修正されていないファイル (Unmodified) を削除した状態で、git status コマンドを実行します。

```
$ git rm Program01.java
rm 'Program01.java'
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted: Program01.java
```

表示されるのは「コミットされるべき変更 (Changes to be committed)」の「削除された (deleted)」となります。次のコミットで、この削除がリポジトリに反映されます。

### 0.1.1 log

このコマンドは、過去のコミット履歴 (ログ) が閲覧できます。ログは膨大なものになりますが、効率よく確認できるように多くのオプションが用意されています。その中でも、使用頻度の高いものについて紹介します。

**git log** : 過去のコミット履歴を閲覧する

[書式] git log [出力対象] [オプション]

[主なオプション]

-n <数字> / --<数字> <数字>件数分を表示する  
--oneline 1行で簡潔に表示する  
--graph ブランチをグラフィカルに表示する  
--since=<date> / --after=<date> <date>以降のログを表示する  
--until=<date> / --before=<date> <date>以前のログを表示する



## 全てのログを表示する

```
git log
```

```
$ git log
commit 068454e0960f4d971d1bc2f69f795552834feb9d (HEAD -> main)
Author: Taro Yamada <email@example.com>
Date:   Fri Feb 14 14:26:24 2020 +0900

    The first version of Program02

commit b2754b7bfc62f5bcf36c9030262d555d030f8303
Merge: c82d29e a169cbf
Author: Taro Yamada <email@example.com>
Date:   Fri Feb 14 13:58:46 2020 +0900

    the conflict has been resolved.

commit c82d29e5817f4481ac17c48541aced94836babae
Author: Taro Yamada <email@example.com>

... 以下略
```

オプションを何も指定しなければ、過去の全てのコミット履歴が表示されます。表示を終了するには「q」キーを押します。

## 件数を指定して表示する

全件表示だと情報が多すぎて見にくい場合は、件数を指定できます。

```
git log -n <数字>
または
git log -<数字>
```

```
$ git log -1
commit 068454e0960f4d971d1bc2f69f795552834feb9d (HEAD -> main)
Author: Taro Yamada <email@example.com>
Date:   Fri Feb 14 14:26:24 2020 +0900

    The first version of Program02
```

## 1行で表示する

主要な情報だけを1行にまとめて表示します。

```
git log --oneline
```

```
$ git log --oneline
068454e (HEAD -> main) The first version of Program02
b2754b7 the conflict has been resolved.
c82d29e Edited by main
a169cbf (feature001) Edited by feature001
39b6dcc (origin/main) Function-01 has been implemented.
d757ba4 The first version of Program01
```

## マージ状況を視覚的に把握する

--graph オプションを付けると、マージの状況までわかりやすく表示します。

```
git log --graph
```

--oneline オプションと組み合わせると、より見やすくなります。

```
$ git log --oneline --graph
* 068454e (HEAD -> main) The first version of Program02
* b2754b7 the conflict has been resolved.
|\
| * a169cbf (feature001) Edited by feature001
* | c82d29e Edited by main
|/
* 39b6dcc (origin/main) Function-01 has been implemented.
* d757ba4 The first version of Program01
```

## 特定のファイルのログのみを表示する

```
git log ファイル名
```

```
$ git log Program02.java
commit 068454e0960f4d971d1bc2f69f795552834feb9d (HEAD -> main)
Author: Taro Yamada <email@example.com>
Date: Fri Feb 14 14:26:24 2020 +0900
```

```
The first version of Program02
```

## 特定のブランチのログのみを表示する

```
git log ブランチ名
```

```
$ git log feature001 --oneline
a169cbf (feature001) Edited by feature001
39b6dcc (origin/main) Function-01 has been implemented.
d757ba4 The first version of Program01
```

## diff

このコマンドは、ワーキングツリーとインデックスの差分を調べたり、2つのコミットの変更の違いを比べるなど、様々な対象を比較できます。

**git diff** : ワーキングツリーに行われた変更を表示する

[書式] git diff [比較元] [比較先] [オプション]

[主なオプション]

--cached / --staged	インデックスへの変更を表示する
--name-only	変更のあったファイル名のみ表示する
--name-status	変更のあったファイル名とステータスを表示する
--stat	変更の統計情報を表示する

## ワーキングツリーとインデックスを比較する

```
git diff
```

インデックスに追加する前に、変更点を確認できます。

```
$ git diff
diff --git a/Program02.java b/Program02.java
index e69de29..9449d59 100644
--- a/Program02.java
+++ b/Program02.java
@@ -0,0 +1,5 @@
+public class Program02 {
+    public static void main(String[] args) {
+        System.out.println("Program02");
+    }
+}
```

変更点として、Program02.java に 5 行のコードが書かれたことを表示しています。この変更点は、git add するとインデックスに反映されます。

実際に git add を実行すると、ワーキングツリーとインデックスの差はなくなるので、次に git diff を実行しても何も表示されません。

```
$ git add Program02.java
$ git diff
$
```

## インデックスと最新コミットを比較する

インデックスと最新コミットとの差分を表示するために、--cached オプションを指定します。コミットする前に変更点を確認するのに役立ちます。

```
$ git diff --cached
diff --git a/Program02.java b/Program02.java
new file mode 100644
index 0000000..9449d59
--- /dev/null
+++ b/Program02.java
@@ -0,0 +1,5 @@
+public class Program02 {
+    public static void main(String[] args) {
+        System.out.println("Program02");
+    }
+}
```

## 2つのコミットの差分を比較する

```
git diff コミットA コミットB
```

どのコミットを指定するかは、`git log --oneline` コマンドなどでコミットのリビジョン番号を調べます。

```
$ git log --oneline
```

```
068454e (HEAD -> main) The first version of Program02  
b2754b7 the conflict has been resolved.  
c82d29e Edited by main  
a169cbf (feature001) Edited by feature001  
39b6dcc (origin/main) Function-01 has been implemented.  
d757ba4 The first version of Program01
```

```
$ git diff b2754b7 068454e
```

```
diff --git a/Program02.java b/Program02.java  
new file mode 100644  
index 00000000..9449d59  
--- /dev/null  
+++ b/Program02.java  
@@ -0,0 +1,5 @@  
+public class Program02 {  
+    public static void main(String[] args) {  
+        System.out.println("Program02");  
+    }  
+}
```

## ブランチ同士を比較する

ブランチ間の差分も比較できます。

```
git diff ブランチ A ブランチ B
```

```
$ git diff feature001 main
diff --git a/Program01.java b/Program01.java
index a4108d6..cfb428a 100644
--- a/Program01.java
+++ b/Program01.java
@@ -2,6 +2,7 @@ public class Program01 {
        public static void main(String[] args) {
            System.out.println("Program01");
            System.out.println("Function-01");
+           System.out.println("Edited by main");
            System.out.println("Edited by feature001");
        }
    }
diff --git a/Program02.java b/Program02.java
new file mode 100644
index 0000000..9449d59
--- /dev/null
+++ b/Program02.java
@@ -0,0 +1,5 @@
+public class Program02 {
+    public static void main(String[] args) {
+        System.out.println("Program02");
+    }
+}
```

## リモートリポジトリとの変更点を見る

ローカルリポジトリとリモートリポジトリの差分を調べることもできます。

git pull すると、ローカルリポジトリにどのような変更が反映されるかを事前に調べる場合は、以下のようになります。

```
git diff ローカルブランチ名 リモートリポジトリ名/リモートブランチ名
```

また、git push すると、リモートリポジトリにどのような変更が反映されるかを調べる場合は、以下のようになります。

```
git diff リモートリポジトリ名/リモートブランチ名 ローカルブランチ名
```

【例】ローカルリポジトリとリモートリポジトリの main ブランチの比較

```
$ git diff main origin/main
diff --git a/Program01.java b/Program01.java
index cfb428a..8612bc1 100644
--- a/Program01.java
+++ b/Program01.java
@@ -2,7 +2,5 @@ public class Program01 {
        public static void main(String[] args) {
            System.out.println("Program01");
            System.out.println("Function-01");
-           System.out.println("Edited by main");
-           System.out.println("Edited by feature001");
        }
    }
diff --git a/Program02.java b/Program02.java
deleted file mode 100644
index 9449d59..0000000
--- a/Program02.java
+++ /dev/null
@@ -1,5 +0,0 @@
-public class Program02 {
-    public static void main(String[] args) {
-        System.out.println("Program02");
-    }
-}
```

## reset

このコマンドは、コミットを打ち消したり、インデックスにステージした内容を元に戻すことができます。「コミットをどこまで戻すか」と「インデックスやワーキングツリーの状態も戻すかどうか」の指定を組み合わせで使います。

**git reset** : リポジトリの状態を前に戻す

[書式] git reset [コミット] [オプション]

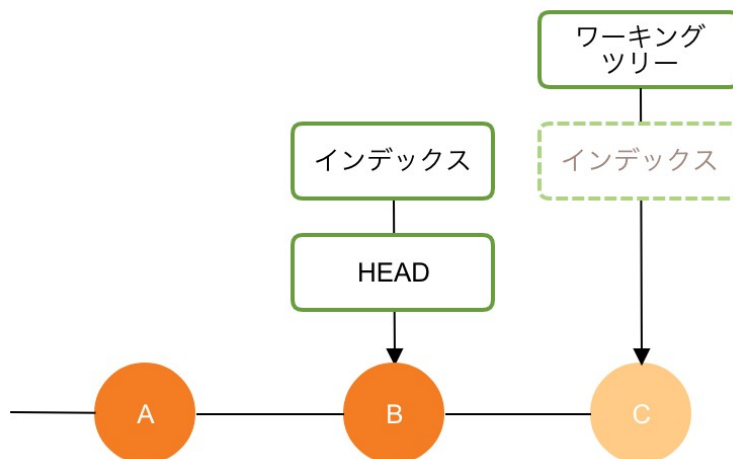
[主なオプション]

--soft	HEAD <sup>1</sup> の位置のみを変更する
--mixed	HEADの位置とインデックスを変更する（デフォルト）
--hard	HEADの位置、インデックス、ワーキングツリーの全てを変更する

### git add を取り消す

git reset

コミット（どこまで戻すか）を省略した場合は、HEADの位置は変わりません。また、オプションを指定しなければ、HEADの位置とインデックスが操作対象となります。つまり、このコマンドでは、git add でステージングした内容だけが取り消されることになります。



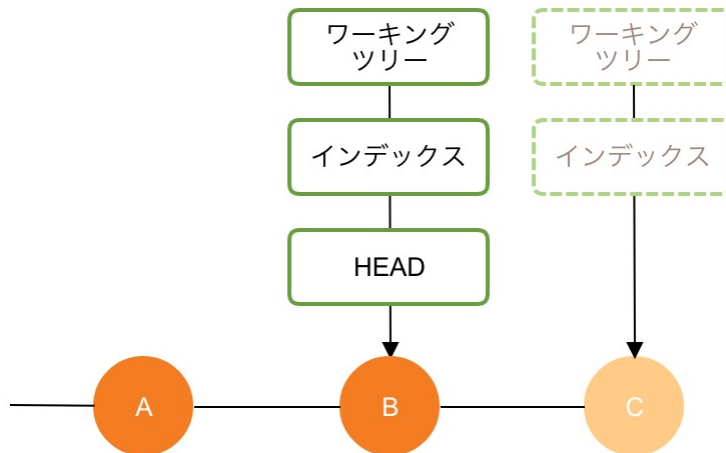
1 HEADとは、現在のブランチで最後に実施したコミットを表します。HEADの詳細は後述します。



## ファイルの変更内容も取り消す

```
git reset --hard
```

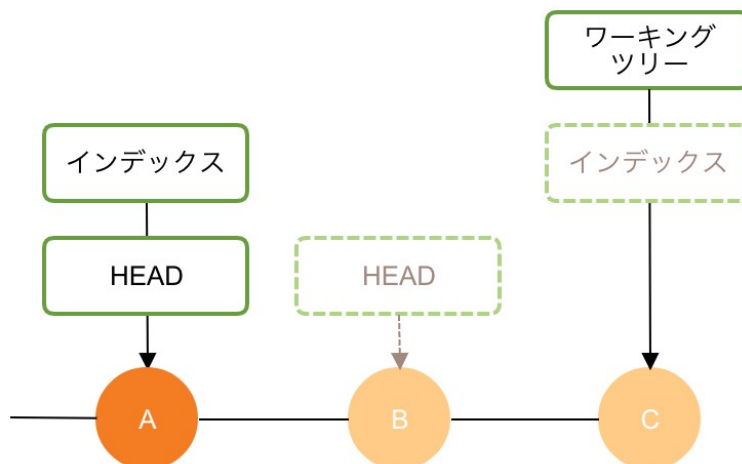
--hard オプションを指定すると、ワーキングツリーの状態も一緒に変更します。上記コマンドでは、コミットを省略しているため、インデックスとワーキングツリーが最新のコミット（HEAD）と同じ状態まで戻ります。



## コミットを1つ前に戻す

```
git reset HEAD^
```

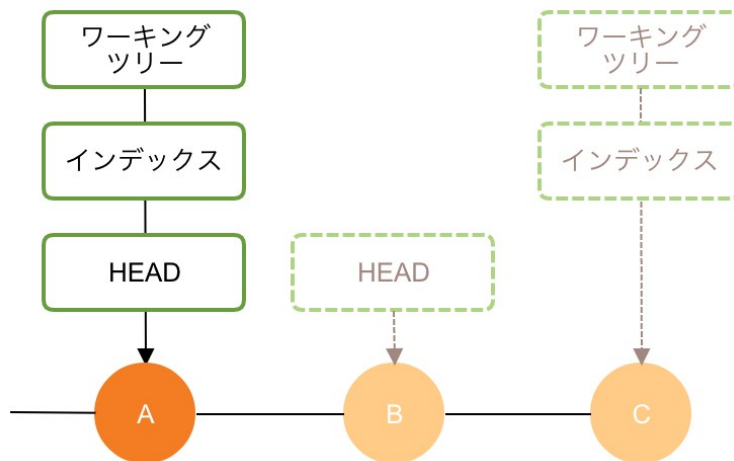
HEAD の後ろに「^」（キャレット）を1つ指定すると、「HEAD の1つ前のコミット」を意味します。上記コマンドでは、オプションを指定していないので、ワーキングツリーはそのまま、インデックスと HEAD が1つ前のコミットである「A」まで移動します。



## 全ての変更を任意のコミットまで戻す

```
git reset --hard コミットA
```

git log --oneline コマンドなどで確認できるリビジョン番号を使って、任意のコミットの状態まで戻ることもできます。上記コマンドのように--hard オプションをつければ、ワーキングツリーも含めた全ての状態が戻ります。手元のソースファイルが失われたり、重要な変更が取り消されることになりますので、--hard オプションの使用はくれぐれも慎重に行ってください。



## HEAD とは何か

HEAD とは、現在のブランチの先頭を表す名前です。例えば、main ブランチにいる時、main の最新コミットが HEAD です。

```
$ git log --oneline
068454e (HEAD -> main) The first version of Program02
b2754b7 the conflict has been resolved.
c82d29e Edited by main
a169cbf (feature001) Edited by feature001
39b6dcc (origin/main) Function-01 has been implemented.
d757ba4 The first version of Program01
```

ブランチが切り替われば、HEAD もまたそちらのブランチへと移動します。

```
$ git checkout feature001
Switched to branch 'feature001'
$ git log --oneline
a169cbf (HEAD -> feature001) Edited by feature001
39b6dcc (origin/main) Function-01 has been implemented.
d757ba4 The first version of Program01
```

各種 git コマンドを使用する際、コミットを指定できる箇所でそれを省略すると、原則として HEAD が指定されたことになります。

【例】

```
git reset
↓
git reset HEAD
```

また、「^」を後ろにつけると「1つ前のコミット」を意味します。「HEAD^^」なら2つ前です。チルダと数字を使って「HEAD~2」とも表せます。

【例】3つ前のコミットと現在のコミットを比較する

```
$ git diff HEAD~3 HEAD
diff --git a/Program01.java b/Program01.java
index 8612bc1..cfb428a 100644
... (以下略)
```

## fetch

このコマンドは、他のリポジトリ（リモートリポジトリなど）のデータを取得するものです。ただし、fetch コマンドは他のリポジトリのデータを取得するだけで、ローカルで作業しているファイルを書き換えたりマージしたりするようなことはありません。

**git fetch** : 他のリポジトリのデータを取得する

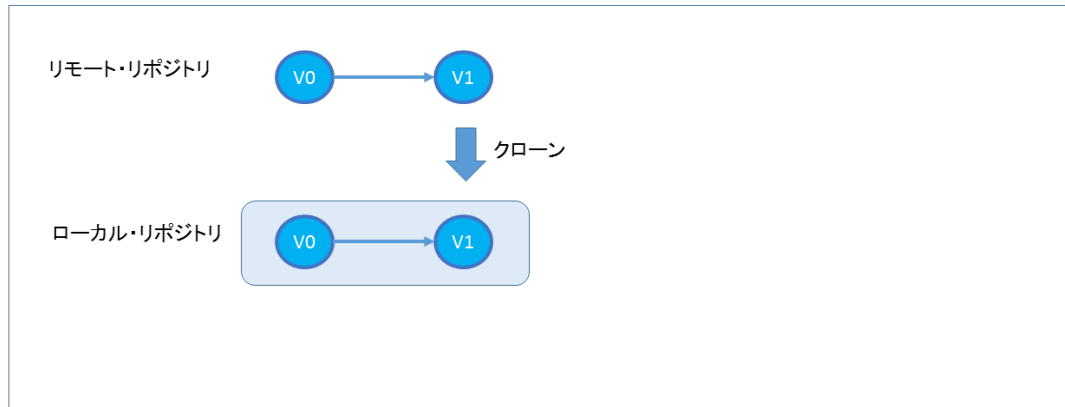
〔書式〕 git fetch [リポジトリ名] [ブランチ名]

実際の使い方を見えます。

## リモートリポジトリを fetch する場合

例えば、リモートリポジトリをクローンしたローカルリポジトリがあったとします。クローンした時点では、リモートリポジトリがローカルリポジトリにコピーされます。

以下の図は、V0 から始まったリモートリポジトリが、V1 の時点でクローンされた時の様子を表しています。



この時点までの変更の記録は、ローカルリポジトリもリモートリポジトリも同じなので、ローカルリポジトリの変更履歴がリモートリポジトリの変更履歴でもあります。

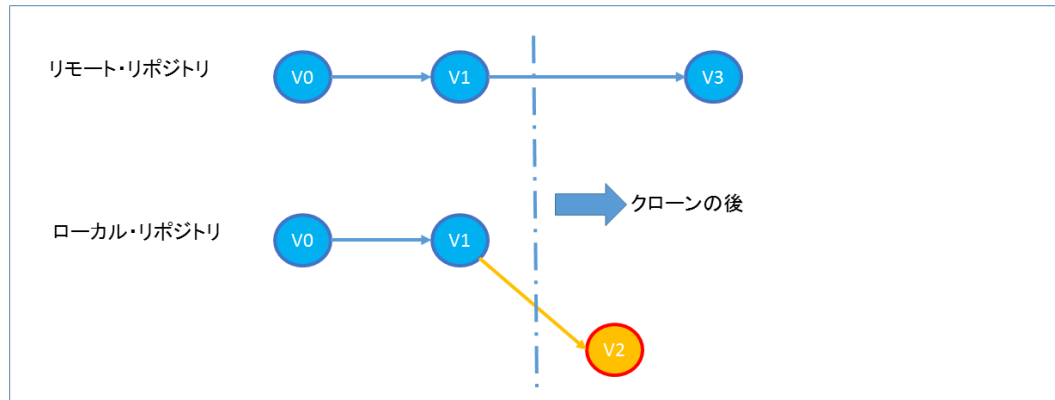
```
$ git log
commit 0361fcf20db006213ee655b04e062319066a32c0 (HEAD -> main , origin/main,
origin/HEAD)
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:38:45 2017 +0900

    Version V1

commit 298074bf414b33041546bfa96c7c78373495a0a2
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:38:03 2017 +0900

    Version V0
```

しかし、ローカルリポジトリで V1 に変更を加えた V2 が記録され、続いてリモートでも V1 が変更された V3 が記録されるとどうなるでしょうか。クローンされた後は、変更の記録が独立して行われることになります。



ローカルリポジトリの変更履歴を確認すると、V3 がないことが分かります。

```
$ git log
commit 121d8ea6ab44eddedb1ffbec410a99504c447943 (HEAD -> main)
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:49:05 2017 +0900

    Version-V2

commit e80541dd2883a69f1d43f687c9051ff57620a91a (HEAD -> main , origin/main,
origin/HEAD)
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:38:45 2017 +0900

    Version-V1

commit e80541dd2883a69f1d43f687c9051ff57620a91a
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:38:03 2017 +0900

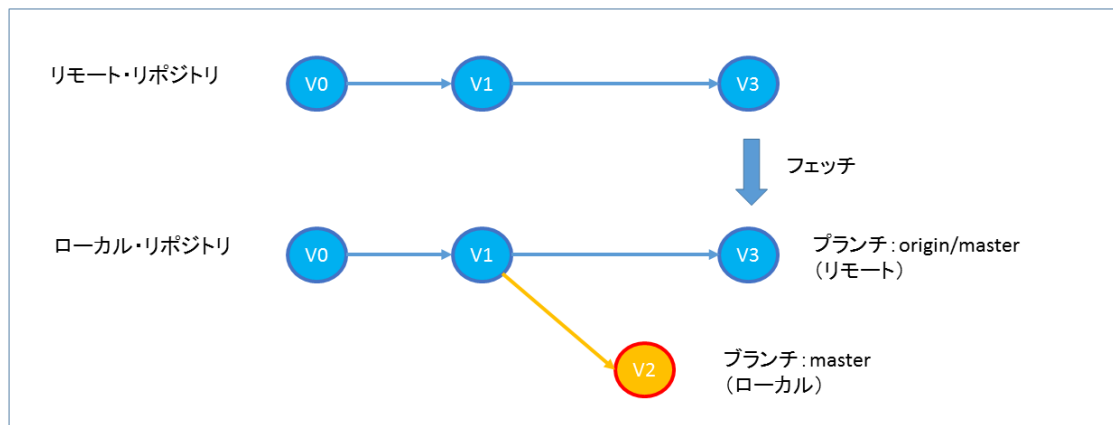
    Version-V0
```

つまり、ローカルからリモートリポジトリは見えません。そこで fetch コマンドを使用します。

リモートリポジトリの名前は、クローンした時点で自動的に origin という名前になっています。そこで、origin という名前を指定して git fetch コマンドを実行します。

```
$ git fetch origin
remote: Counting objects : 3 , done.
remote: Compressing objects : 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From C:/gitworkclone/. . /gitwork
    0361fcf..1262d1d      main    -> origin/main
```

コマンドを実行すると、ローカルリポジトリにリモートリポジトリ追跡のためのブランチが作られます。このブランチの名前を使って、リモートにどんな変更が記録されているかを見られます。



では、ローカルでリモートリポジトリとローカルリポジトリの変更の履歴を見えます。

リモートリポジトリの変更履歴は、リモートリポジトリ追跡のためのブランチ名 origin/main を指定してコマンドを実行することで表示されます。

```
git log origin/main
```

```
$ git log origin/main
commit 1262d1d3e83c0e3f36b840a844357a5255800a46 (origin/main, origin/HEAD)
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:49:05 2017 +0900

    Version-V3

commit 0361fcf20db006213ee655b04e062319066a32c0
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:38:45 2017 +0900

    Version-V1

commit 298074bf414b33041546bfa96c7c78373495a0a2
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:38:03 2017 +0900

    Version-V0
```

今度は、変更履歴に V3 が表示されています。

## pull

このコマンドは、他のリポジトリ（リモートリポジトリなど）のデータを取得し、ローカルリポジトリのブランチに統合するものです。このコマンドを実行すると、リモートリポジトリでの変更をローカルリポジトリのブランチに組み込みます。

**git pull** : 他のリポジトリのデータを取得する

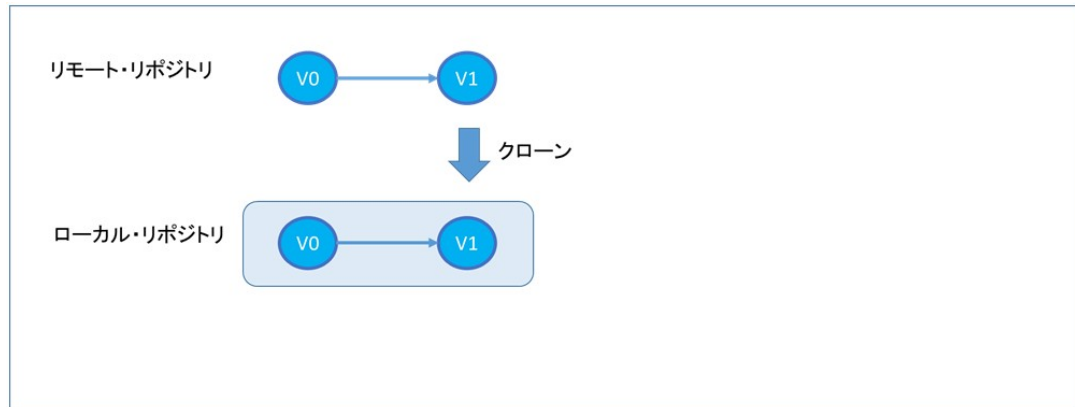
【書式】 git pull [リポジトリ名] [ブランチ名]

実際の使い方を見えます。

## リモートリポジトリを pull する場合

例えば、リモートリポジトリをクローンしたローカルリポジトリがあったとします。クローンした時点では、リモートリポジトリがローカルリポジトリにコピーされます。

以下の図は、V0 から始まったリモートリポジトリが、V1 の時点でクローンされた時の様子を表しています。



この時点までの変更の記録は、ローカルリポジトリもリモートリポジトリも同じなので、ローカルリポジトリの変更履歴がリモートリポジトリの変更履歴でもあります。

```
$ git log
commit 0361fcf20db006213ee655b04e062319066a32c0 (HEAD -> main , origin/main,
origin/HEAD)
Author : Taro Yamada <email@example.com>
Date : Fri Aug 25 15:38:45 2017 +0900

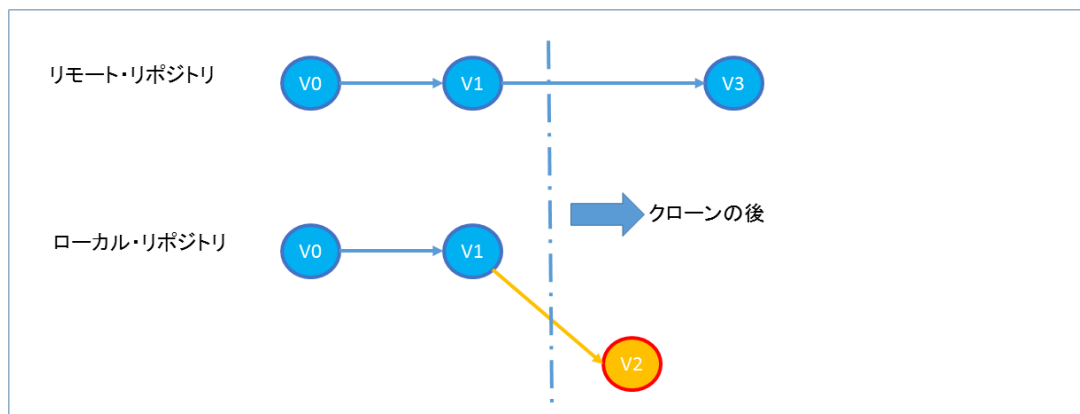
    Version V1

commit 298074bf414b33041546bfa96c7c78373495a0a2
Author : Taro Yamada <email@example.com>
Date : Fri Aug 25 15:38:03 2017 +0900

    Version V0
```



しかし、ローカルリポジトリで V1 に変更を加えた V2 が記録され、続いてリモートでも V1 が変更された V3 が記録されるとどうなるでしょうか。クローンされた後は、変更の記録が独立して行われることになります。



ローカルリポジトリの変更履歴を確認すると、V3 がないことが分かります。

```
$ git log
commit 121d8ea6ab44eddedb1ffbec410a99504c447943 (HEAD -> main)
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:49:05 2017 +0900

    Version-V2

commit 0361fcf20db006213ee655b04e062319066a32c0 (origin/main, origin/HEAD)
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:38:45 2017 +0900

    Version-V1

commit 298074bf414b33041546bfa96c7c78373495a0a2
Author : Taro Yamada <email@example.com>
Date :   Fri Aug 25 15:38:03 2017 +0900

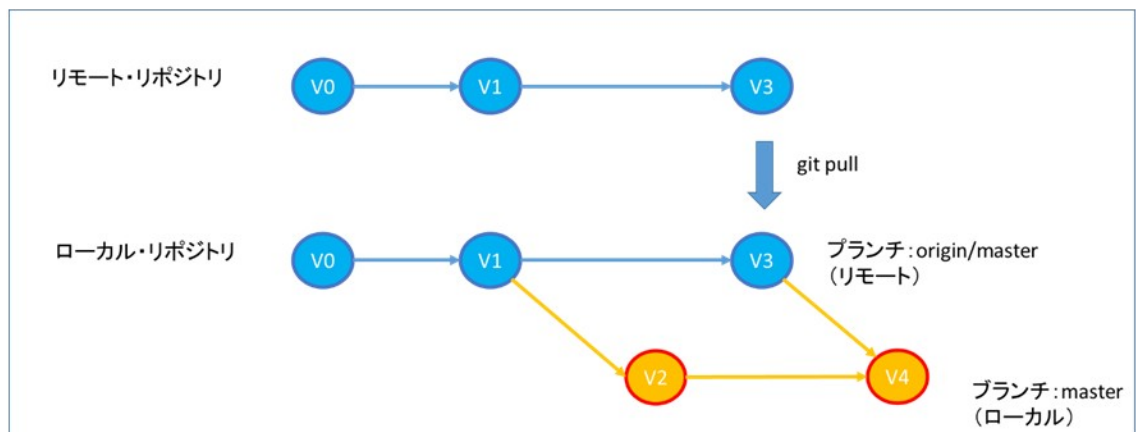
    Version-V0
```

つまり、ローカルリポジトリとリモートリポジトリには違いがあるということです。そこで、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させるために、pull コマンドを使用します。

リモートリポジトリの名前は、クローンした時点で自動的に origin という名前になっています。そこで、origin という名前を指定して git pull コマンドを実行します。

```
$ git pull origin main
remote: Counting objects : 3 , done.
remote: Compressing objects : 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From C:/gitworkclone/. . /gitwork
*   branch              main    -> FETCH_HEAD
    d3dbe0..31cf6bd      main    -> origin/main
Merge made by the 'recursive' strategy.
 Program01.java | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

コマンドが実行されると、ローカルリポジトリにリモートリポジトリの変更内容がマージされ、V4 が追加されます。



では、ローカルリポジトリにリモートリポジトリの変更が反映された履歴を見えます。

```
$ git log
commit d967b0b06757c2acb97d6060d5653b783e672d1a (HEAD -> main)
Merge : cb39bee 31cf6db
Author : Taro Yamada <email@example.com>
Date :   Mon Sep 4 11:01:13 2017 +0900

    Version-V4

commit 31cf6db65b13a160672d9874e50ffaed7ac7d887 (origin/main, origin/HEAD)
Author : Taro Yamada <email@example.com>
Date :   Thu Aug 24 20:24:35 2017 +0900

    Version-V3

commit cb39bee8fd226701eb5b6c82623dde7c7cd06641
Author : Taro Yamada <email@example.com>
Date :   Thu Aug 24 20:23:28 2017 +0900

    Version-V2

commit d13dbe005a65f01852aa47ceb75fff03b9c8fa2c
Author : Taro Yamada <email@example.com>
Date :   Thu Aug 24 20:20:24 2017 +0900

    Version-V1

commit 7915704bf9ee4d5fd4e2735e82bee922724a1ef9
Author : Taro Yamada <email@example.com>
Date :   Thu Aug 24 20:19:32 2017 +0900

    Version-V0
```

今度は、ローカルリポジトリの履歴に、リモートリポジトリの V3 とローカルリポジトリの新たなタグ V4 の変更履歴が追加されています。

## tag

このコマンドは、コミットに名前を付けることができます。名前を付けることで参照しやすくなります。また、バージョンを付けて管理したい場合（例：ver1.0）などにも便利です。タグには、簡易的な「注釈なし」タグと、詳細な「注釈あり」タグがあります。

### git tag : コミットに名前を付ける

[書式] git tag [タグ名] [オプション]

#### [主なオプション]

- a 注釈ありでタグをつける
- m 注釈ありタグの場合、コメントをつける
- d タグを削除する

### 注釈なしでタグをつける

```
git tag タグ名
```

「first-tag」という名前で注釈なしタグを付けます。git log コマンドに--decorate オプションを付けて、タグの情報も一緒に表示して結果を確認します。

```
$ git tag first-tag
$ git log --oneline --decorate
068454e (HEAD -> main, tag: first-tag) The first version of Program02
b2754b7 the conflict has been resolved.
c82d29e Edited by main
a169cbf (feature001) Edited by feature001
39b6dcc (origin/main) Function-01 has been implemented.
d757ba4 The first version of Program01
```

## 注釈ありでタグをつける

注釈付きのタグを付けます。-a オプションにタグ名、-m オプションにコメントを指定します。

```
git tag -a タグ名 -m コメント
```

「second-tag」という名前でタグを付けます。

```
$ git tag -a second-tag -m "注釈ありのタグです"
```

ログを確認すると、先程の「first-tag」に続いて「second-tag」が追加されています。1つのコミットにタグは複数付けられます。

```
$ git log --oneline --decorate
068454e (HEAD -> main, tag: first-tag, tag: second-tag) The first version of
Program02
b2754b7 the conflict has been resolved.
c82d29e Edited by main
a169cbf (feature001) Edited by feature001
39b6dcc (origin/main) Function-01 has been implemented.
d757ba4 The first version of Program01
```

## タグの内容を確認する

タグの情報は git show コマンドで確認できます。

```
git show タグ名
```

```
$ git show second-tag
tag second-tag
Tagger: Taro Yamada <email@example.com>
Date:   Mon Feb 17 13:33:47 2020 +0900

"注釈ありのタグです"

commit 068454e0960f4d971d1bc2f69f795552834feb9d (HEAD -> main, tag: first-tag,
tag: detail-tag)
... (以下略)
```

## タグを削除する

タグを削除するには、`-d` オプションを指定します。

```
git tag -d タグ名
```

```
$ git tag -d first-tag
Deleted tag 'first-tag' (was 068454e)
$ git log --oneline --decorate
068454e (HEAD -> main, tag: second-tag) The first version of Program02
b2754b7 the conflict has been resolved.
c82d29e Edited by main
a169cbf (feature001) Edited by feature001
39b6dcc (origin/main) Function-01 has been implemented.
d757ba4 The first version of Program01
```

## 任意のコミットにタグをつける

これまでは、最新のコミットにのみタグを付ける例でしたが、過去のコミットにタグを付けることも可能です。

```
git tag タグ名 コミット
```

【例】2つ前のコミットにタグをつける

```
$ git tag third-tag HEAD^^
$ git log --oneline --decorate
068454e (HEAD -> main, tag: second-tag) The first version of Program02
b2754b7 the conflict has been resolved.
c82d29e (tag: third-tag) Edited by main
a169cbf (feature001) Edited by feature001
39b6dcc (origin/main) Function-01 has been implemented.
d757ba4 The first version of Program01
```

## タグをリモートリポジトリに反映させる

タグは、作成しただけではリモートリポジトリには反映されません。git push コマンドでタグ名を指定します。

```
git push origin タグ名
```

全てのタグをまとめて push する場合は、--tags オプションを指定します。

```
git push --tags
```

## タグの一覧を確認する

全てのタグを一覧表示したい場合は、次のコマンドを使います。

```
git tag
```

## push

このコマンドは、ローカルリポジトリの変更を他のリポジトリ（リモートリポジトリなど）に反映します。

**git push** : ローカルリポジトリの変更を他のリポジトリに反映する

[書式] git push [オプション] [リポジトリ名] [ブランチ名]

[主なオプション]

--all	全てのブランチを反映する
--tag	全てのタグを反映する
--delete	指定したリポジトリのブランチを削除する

実際の使い方を見えます。

## 現在のブランチを他のリポジトリのブランチに反映する

```
git push リポジトリ名 ブランチ名
```

現在のブランチを、リモートリポジトリの main ブランチに反映する例になります。

```
$ git push origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 327 bytes | 327.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To C:/gitrepository/sample.git/
* [new branch]      main -> main
```

## 指定したリポジトリのブランチを削除する

```
git push --delete リポジトリ名 ブランチ名
```

リモートリポジトリの feature-change-textfile ブランチを削除する例になります。

```
$ git push --delete origin feature-change-textfile
To https://github.com/tech-student01/gitflowtest.git
- [deleted]      feature-change-textfile
```

また、--delete オプションを使わずに、「:」を指定する方法もあります。

```
git push リポジトリ名 :ブランチ名
```



## commit

このコマンドは、インデックスに追加（git add）した内容をリポジトリに記録します。

**git commit** : インデックスに追加した内容をリポジトリに記録する

[書式] git commit [オプション] [ファイル名]

[主なオプション]

-m コメント      コメントを指定する

-a                  変更済みの全ファイルをリポジトリに記録する

実際の使い方を見えます。

### インデックスに追加した内容をリポジトリに記録する

```
git commit -m コメント
```

インデックスに追加したファイルをローカルリポジトリに記録する例になります。

```
$ git commit -m "The first version of Program01.java"
[main (root-commit) 14ce97b] The first version of Program01.java
1 file changed, 5 insertions(+)
create mode 100644 Program01.java
```