ミクシィGit研修
（21新卒）

はじめに

はじめに

この講義中は **#21卒git研修** に何か書き込むと、

ニコ動のコメントっぽく画面に流れます。

質問とか感想とか、どんどん実況してください。

講師について

自己紹介

藤田朱門(19 新卒)

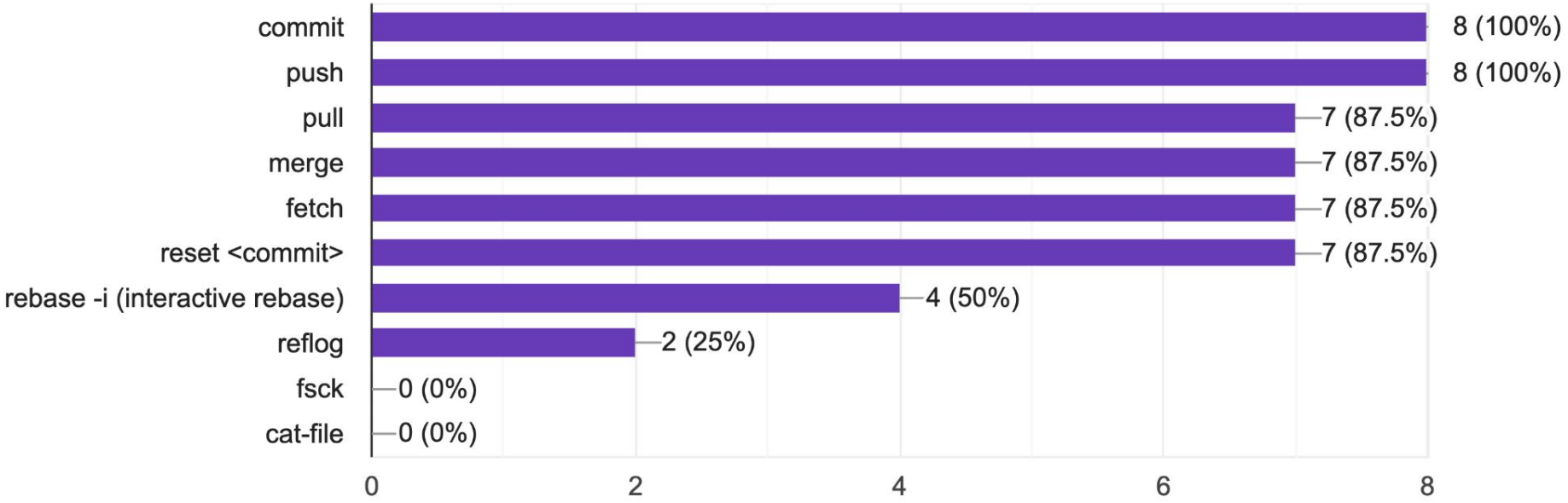
- 経歴
  - 新規事業でサーバ(Go)
  - コトダマンでクライアントとサーバ兼任(Unity, Java)
  - 開発本部インフラ室で映像配信・編集基盤(Go)←イマココ

- その他の活動
  - Git Challenge 問題解説
  - 技術書典
    - mixi tech note は大体寄稿していて、Go か Git の記事を書いてる
    - 『Docker で始めるゲームボーイアドバンス開発入門』著者
  - 最近は趣味で Git のクローンを Go で作ってる
  - Twitter → @shumon_84

事前アンケートについて

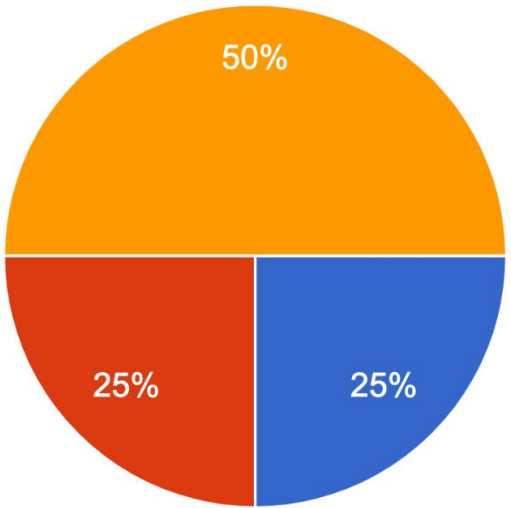
事前アンケートについて

Q. Git のサブコマンドをどれぐらい使ったことある？



事前アンケートについて

Q. Git 研修の内容に求めるものは？



- Git の基本的な使い方・トラブルの対処を身に付けたい
- pull request など、開発プラットフォーム上で Git リポジトリをチーム運用するための機能が知りたい
- Git の内部構造を理解して、より複雑なトラブルからのリカバリなどをできるようになりたい
- その他

事前アンケートについて

というわけで、今日は

- **Git の基礎**
- **Git によるチーム開発のいろは**
- **Git の内部構造**

についてやっていきます。

最後に、今日学んだことを生かして **GitChallenge** に挑戦してもらいます。

というわけで

今日の予定

10:30	Git の基礎
11:30	Git によるチーム開発のいろは
12:00	昼食
13:00	Git の内部構造
15:00	GitChallenge に挑戦
17:30	解説
18:30	終了

Git の基礎

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

卒論.pdf / 卒論(1).pdf / 卒論_最新.pdf / 卒論_修正版.pdf / 卒論_最終稿.pdf / 卒論_提出用.pdf

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

卒論.pdf / 卒論(1).pdf / 卒論_最新.pdf / 卒論_修正版.pdf / 卒論_最終稿.pdf / 卒論_提出用.pdf

→ どれが最新？

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

卒論.pdf / 卒論(1).pdf / 卒論_最新.pdf / 卒論_修正版.pdf / 卒論_最終稿.pdf / 卒論_提出用.pdf

→ どれが最新？ 1 つ前の版は？

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

卒論.pdf / 卒論(1).pdf / 卒論_最新.pdf / 卒論_修正版.pdf / 卒論_最終稿.pdf / 卒論_提出用.pdf

→ どれが最新？ 1 つ前の版は？ 5 つ前の版は？

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

卒論.pdf / 卒論(1).pdf / 卒論_最新.pdf / 卒論_修正版.pdf / 卒論_最終稿.pdf / 卒論_提出用.pdf

→ どれが最新？ 1 つ前の版は？ 5 つ前の版は？

卒論_20210118.pdf / 卒論_20210120.pdf / 卒論20210201.pdf

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

卒論.pdf / 卒論(1).pdf / 卒論_最新.pdf / 卒論_修正版.pdf / 卒論_最終稿.pdf / 卒論_提出用.pdf

→ どれが最新？ 1 つ前の版は？ 5 つ前の版は？

卒論_20210118.pdf / 卒論_20210120.pdf / 卒論20210201.pdf

→ 新しいバージョン保存するたびに容量が 2 倍 3 倍になっていく……

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

卒論.pdf / 卒論(1).pdf / 卒論_最新.pdf / 卒論_修正版.pdf / 卒論_最終稿.pdf / 卒論_提出用.pdf

→ どれが最新？ 1 つ前の版は？ 5 つ前の版は？

卒論_20210118.pdf / 卒論_20210120.pdf / 卒論20210201.pdf

→ 新しいバージョン保存するたびに容量が 2 倍 3 倍になっていく……

1 人で書いている卒論ならこれでもなんとかなるけど、100 人で数年かけて開発するソフトウェアならヤバイ

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

卒論.pdf / 卒論(1).pdf / 卒論_最新.pdf / 卒論_修正版.pdf / 卒論_最終稿.pdf / 卒論_提出用.pdf

→ どれが最新？ 1 つ前の版は？ 5 つ前の版は？

卒論_20210118.pdf / 卒論_20210120.pdf / 卒論20210201.pdf

→ 新しいバージョン保存するたびに容量が 2 倍 3 倍になっていく……

1 人で書いている卒論ならこれでもなんとかなるけど、100 人で数年かけて開発するソフトウェアならヤバイ

Git を始めとする VCS を使えばこんな問題とはおさらば！

Git の基礎

Git は**バージョン管理システム(VCS)**の 1 つ。

バージョン管理とは？

卒論.pdf / 卒論(1).pdf / 卒論_最新.pdf / 卒論_修正版.pdf / 卒論_最終稿.pdf / 卒論_提出用.pdf

→ どれが最新？ 1 つ前の版は？ 5 つ前の版は？

卒論_20210118.pdf / 卒論_20210120.pdf / 卒論20210201.pdf

→ 新しいバージョン保存するたびに容量が 2 倍 3 倍になっていく……

1 人で書いている卒論ならこれでもなんとかなるけど、100 人で数年かけて開発するソフトウェアならヤバイ

Git を始めとする VCS を使えばこんな問題とはおさらば！

簡単に履歴を辿れて、容量も抑えつつ、おまけに改ざんにも強い神ツール！

Git の基礎

と言いつつ、全員 commit と push は使ったことがあるということだったので、これくらいは知っていますね。

Git の基礎

と言いつつ、全員 commit と push は使ったことがあるということだったので、これくらいは知っていますね。

Git を使う上で重要な機能として branch と merge がある。

Git の基礎

と言いつつ、全員 commit と push は使ったことがあるということだったので、これくらいは知っていますね。

Git を使う上で重要な機能として branch と merge がある。

Git は単に時系列順にバージョンを管理するだけでなく、別々の時間軸のバージョンを管理できる。

Git の基礎

と言いつつ、全員 commit と push は使ったことがあるということだったので、これくらいは知っていますね。

Git を使う上で重要な機能として branch と merge がある。

Git は単に時系列順にバージョンを管理するだけでなく、別々の時間軸のバージョンを管理できる。

さらにそれらを統合することができる。

Git の基礎

と言いつつ、全員 commit と push は使ったことがあるということだったので、これくらいは知っていますね。

Git を使う上で重要な機能として branch と merge がある。

Git は単に時系列順にバージョンを管理するだけでなく、別々の時間軸のバージョンを管理できる。

さらにそれらを統合することができる。

別々の時間軸のことを branch と呼び、それらを統合することを merge と言う。

Git の基礎

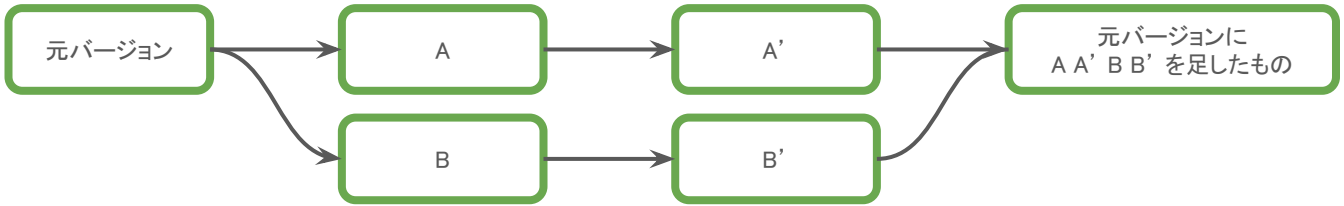
と言いつつ、全員 commit と push は使ったことがあるということだったので、これくらいは知っていますね。

Git を使う上で重要な機能として branch と merge がある。

Git は単に時系列順にバージョンを管理するだけでなく、別々の時間軸のバージョンを管理できる。

さらにそれらを統合することができる。

別々の時間軸のことを branch と呼び、それらを統合することを merge と言う。



Git の基礎

branch と merge を経験してみる。

Git の基礎

branch と merge を経験してみる。

まずは Git リポジトリを作って、適当に commit を作る。

```
$ mkdir hoge && cd hoge

$ git init

$ echo Hello > README.md

$ git add README.md

$ git commit -m "first commit"

$ git log --oneline

45e2f9c (HEAD -> master) first commit
```

Git の基礎

git branch で現在作られている branch 一覧を見たり、新しい branch を作ったりできる。

Git の基礎

git branch で現在作られている branch 一覧を見たり、新しい branch を作ったりできる。

```
$ git branch

* master
```

Git の基礎

git branch で現在作られている branch 一覧を見たり、新しい branch を作ったりできる。

```
$ git bran

* master
```

master はデフォルトで
作られる branch

Git の基礎

git branch で現在作られている branch 一覧を見たり、新しい branch を作ったりできる。

ちなみに Git 2.28 以降なら**git config --global init.defaultBranch**で init 時に作られる branch を変更できる。

```
$ git bran

* master
```

master はデフォルトで
作られる branch

Git の基礎

git branch で現在作られている branch 一覧を見たり、新しい branch を作ったりできる。

ちなみに Git 2.28 以降なら**git config --global init.defaultBranch**で init 時に作られる branch を変更できる。

```
$ git bran

* master

$ git branch develop
```

master はデフォルトで
作られる branch

Git の基礎

git branch で現在作られている branch 一覧を見たり、新しい branch を作ったりできる。

ちなみに Git 2.28 以降なら**git config --global init.defaultBranch**で init 時に作られる branch を変更できる。

```
$ git bran

* master

$ git branch develop

$ git branch

  develop

* master
```

master はデフォルトで作られる branch

Git の基礎

git branch で現在作られている branch 一覧を見たり、新しい branch を作ったりできる。

ちなみに Git 2.28 以降なら**git config --global init.defaultBranch**で init 時に作られる branch を変更できる。

```
$ git bran
* master
$ git branch develop
$ git branc
  develop
* master
```

master はデフォルトで作られる branch

develop という新しい branch が作られている

Git の基礎

git branch で現在作られている branch 一覧を見たり、新しい branch を作ったりできる。

ちなみに Git 2.28 以降なら**git config --global init.defaultBranch**で init 時に作られる branch を変更できる。

```
$ git bran
* master
$ git branch develop
$ git branc
  develop
* master
```

master はデフォルトで作られる branch

develop という新しい branch が作られている

新しい branch を作っただけで、まだ master にいることは注意。

Git の基礎

branch を移動したい場合は、**git checkout** を使う。

Git の基礎

branch を移動したい場合は、**git checkout** を使う。

```
$ git branch

  develop

* master
```

Git の基礎

branch を移動したい場合は、**git checkout** を使う。

```
$ git branch

  develop

* master

$ git checkout develop
```

Git の基礎

branch を移動したい場合は、**git checkout** を使う。

```
$ git branch
  develop
* master
$ git checkout develop
$ git branch
* develop
  master
```

Git の基礎

branch を移動したい場合は、**git checkout** を使う。

```
$ git branch
  develop
* master

$ git checkout develop

* develop
  master
```

＊が現在の branch を表している

Git の基礎

branch を移動したい場合は、**git checkout** を使う。

あとは、それぞれの branch を好きに移動しながら開発を進めていく。

```
$ git branch

  develop

* master

                develop

* develop

  master
```

＊が現在の branch を表している

Git の基礎

master と develop でそれぞれ開発を進める。

Git の基礎

master と develop でそれぞれ開発を進める。

```
$ git log --oneline master

e0d4607 (master) README.md に「master」と追記

76d6092 master.txt を追加

45e2f9c first commit
```

Git の基礎

master と develop でそれぞれ開発を進める。

```
$ git log --oneline master

e0d4607 (master) README.md に「master」と追記

76d6092 master.txt を追加

45e2f9c first commit

$ git log --oneline develop

a81fbd0 (develop) README.md に「develop」と追記

52ad527 develop.txt を追加

45e2f9c first commit
```

Git の基礎

master と develop でそれぞれ開発を進める。

```
$ git log --oneline master

e0d4607 (master) README.md に「master」と追記

76d6092 master.txt を追加

45e2f9c first commit

$ git log --oneline develop

a81fbd0 (develop) README.md に「develop」と追記

52ad527 develop.txt を追加

45e2f9c first commit
```

first commit から、それぞれの branch に 2 コミットずつ積まれている。

Git の基礎

図で表すとこんな感じ。

Git の基礎

図で表すとこんな感じ。次はこれを merge してみる。

Git の基礎

図で表すとこんな感じ。次はこれを merge してみる。

Git では、「merge する branch」=“**theirs**”、「merge される branch」=“**ours**” と呼ぶ。

Git の基礎

図で表すとこんな感じ。次はこれを merge してみる。

Git では、「merge する branch」=**“theirs”**、「merge される branch」=**“ours”** と呼ぶ。

今回は **theirs = develop**, **ours = master** として merge していく。

Git の基礎

merge するには、ours に checkout している状態で、**git merge <theirs>** とする。

```
$ git branch

  develop

* master
```

Git の基礎

merge するには、ours に checkout している状態で、**git merge <theirs>** とする。

```
$ git branch

  develop

* master
```

Git の基礎

merge するには、ours に checkout している状態で、**git merge <theirs>** とする。

```
$ git branch

  develop

* master

$ git merge develop
```

Git の基礎

merge するには、ours に checkout している状態で、**git merge <theirs>** とする。

```
$ git branch

  develop

* master

$ git merge develop

Auto-merging README.md

CONFLICT (content): Merge conflict in README.md

Automatic merge failed; fix conflicts and then commit the result.
```

Git の基礎

merge するには、ours に checkout している状態で、**git merge <theirs>** とする。

```
$ git branch

  develop

* master

$ git merge develop

Auto-merging README.md

CONFLICT (content): Merge conflict in README.md

Automatic merge failed; fix conflicts and then commit the result.
```

Git の基礎

merge するには、ours に checkout している状態で、**git merge <theirs>** とする。

README.md で**コンフリクト(競合)**を起こして、Automatic merge に失敗した。

```
$ git branch

  develop

* master

$ git merge develop

Auto-merging README.md

CONFLICT (content): Merge conflict in README.md

Automatic merge failed; fix conflicts and then commit the result.
```

Git の基礎

merge すると、Git は theirs で開発した差分を自動的に ours に取り込んでくれる。

Git の基礎

merge すると、Git は theirs で開発した差分を自動的に ours に取り込んでくれる。

しかし、ours と theirs で同じ箇所を変更していた場合、どちらの変更を残すべきか自動的に判定できない。

Git の基礎

merge すると、Git は theirs で開発した差分を自動的に ours に取り込んでくれる。

しかし、ours と theirs で同じ箇所を変更していた場合、どちらの変更を残すべきか自動的に判定できない。

→ これがコンフリクト

Git の基礎

merge すると、Git は theirs で開発した差分を自動的に ours に取り込んでくれる。

しかし、ours と theirs で同じ箇所を変更していた場合、どちらの変更を残すべきか自動的に判定できない。

→ これがコンフリクト

コンフリクトが起きたら、人間が手動で差分を取り込んで、コンフリクトを解消させる必要がある。

Git の基礎

merge すると、Git は theirs で開発した差分を自動的に ours に取り込んでくれる。

しかし、ours と theirs で同じ箇所を変更していた場合、どちらの変更を残すべきか自動的に判定できない。

→ これがコンフリクト

コンフリクトが起きたら、人間が手動で差分を取り込んで、コンフリクトを解消させる必要がある。

やってみる。

Git の基礎

git status で、どのファイルでコンフリクトしているか確認できる。

Git の基礎

git status で、どのファイルでコンフリクトしているか確認できる。

```
$ git status

On branch master

-- 省略 --

Unmerged paths:

  (use "git add <file>..." to mark resolution)

    both modified:   README.md
```

Git の基礎

git status で、どのファイルでコンフリクトしているか確認できる。

both modified となっているファイルがコンフリクト中。

```
$ git status

On branch master

-- 省略 --

Unmerged paths:

  (use "git add <file>..." to mark resolution)

    both modified:   README.md
```

Git の基礎

git status で、どのファイルでコンフリクトしているか確認できる。

both modified となっているファイルがコンフリクト中。

```
$ git status

On branch master

-- 省略 --

Unmerged paths:

  (use "git add <file>..." to mark resolution)

    both modified:   README.md
```

今回は README.md だけがコンフリクトしていることが分かる。

Git の基礎

コンフリクト中のファイルを開けば、どの行がどのようにコンフリクトしているのか Git が教えてくれるが、

自動で検出していることもあって、あまり適切でないことが多い。

Git の基礎

コンフリクト中のファイルを開けば、どの行がどのようにコンフリクトしているのか Git が教えてくれるが、

自動で検出していることもあって、あまり適切でないことが多い。

コンフリクト解消の基本は、それぞれの branch がそのファイルに対して**どのような修正を施したのかの、**

意図や内容をきちんと確認してから手を付けること。

Git の基礎

コンフリクト中のファイルを開けば、どの行がどのようにコンフリクトしているのか Git が教えてくれるが、

自動で検出していることもあって、あまり適切でないことが多い。

コンフリクト解消の基本は、それぞれの branch がそのファイルに対して**どのような修正を施したのかの、**

意図や内容をきちんと確認してから手を付けること。

ついついコンフリクト行だけを眺めて、手癖でコンフリクト解消したくなるけど、先に確認するコストと、

不適切な merge になってしまった場合の後始末のコストを比べたら**絶対先に確認するべき。**

Git の基礎

コンフリクト中のファイルを開けば、どの行がどのようにコンフリクトしているのか Git が教えてくれるが、

自動で検出していることもあって、あまり適切でないことが多い。

コンフリクト解消の基本は、それぞれの branch がそのファイルに対して**どのような修正を施したのかの、**

意図や内容をきちんと確認してから手を付けること。

ついついコンフリクト行だけを眺めて、手癖でコンフリクト解消したくなるけど、先に確認するコストと、

不適切な merge になってしまった場合の後始末のコストを比べたら**絶対先に確認するべき。**

→ そもそもコンフリクトしている時点でプチ事故なので、めんどくさがらず慎重に

Git の基礎

コンフリクト中のファイルを開けば、どの行がどのようにコンフリクトしているのか Git が教えてくれるが、

自動で検出していることもあって、あまり適切でないことが多い。

コンフリクト解消の基本は、それぞれの branch がそのファイルに対して**どのような修正を施したのかの、**

意図や内容をきちんと確認してから手を付けること。

ついついコンフリクト行だけを眺めて、手癖でコンフリクト解消したくなるけど、先に確認するコストと、

不適切な merge になってしまった場合の後始末のコストを比べたら**絶対先に確認するべき。**

→ そもそもコンフリクトしている時点でプチ事故なので、めんどくさがらず慎重に

その上で、具体的にどう merge すれば、両方の branch の修正を上手に取り込めるかを考えていく。

Git の基礎

ours と theirs の変更を比較したい場合、まずそれぞれの branch がどこから分岐しているかを調べる。

Git の基礎

ours と theirs の変更を比較したい場合、まずそれぞれの branch がどこから分岐しているかを調べる。

git merge-base <branch1> <branch2>で分岐した commit を調べられる。

Git の基礎

ours と theirs の変更を比較したい場合、まずそれぞれの branch がどこから分岐しているかを調べる。

git merge-base <branch1> <branch2>で分岐した commit を調べられる。

```
$ git merge-base master develop

45e2f9c8827024f53ca982c70676614f781205d7
```

Git の基礎

ours と theirs の変更を比較したい場合、まずそれぞれの branch がどこから分岐しているかを調べる。

git merge-base <branch1> <branch2>で分岐した commit を調べられる。

```
$ git merge-base master develop

45e2f9c8827024f53ca982c70676614f781205d7
```

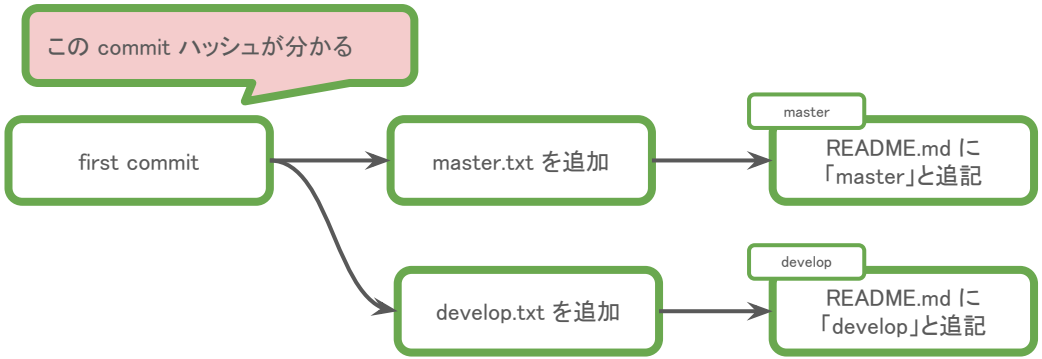
Git の基礎

ours と theirs の変更を比較したい場合、まずそれぞれの branch がどこから分岐しているかを調べる。

git merge-base <branch1> <branch2>で分岐した commit を調べられる。

```
$ git merge-base master develop

45e2f9c8827024f53ca982c70676614f781205d7
```



Git の基礎

git diff で、merge-base から README.md にどんな修正を加えたのか確認する。

Git の基礎

git diff で、merge-base から README.md にどんな修正を加えたのか確認する。

```
$ git diff 45e2f9c8827024f53ca982c70676614f781205d7 develop README.md
```

Git の基礎

git diff で、merge-base から README.md にどんな修正を加えたのか確認する。

```
$ git diff 45e2f9c8827024f53ca982c70676614f781205d7 develop README.md
diff --git a/README.md b/README.md
index e965047..214c073 100644
--- a/README.md
+++ b/README.md
@@ -1 +1 @@
-Hello
+Hello develop
```

Git の基礎

git diff で、merge-base から README.md にどんな修正を加えたのか確認する。

```
$ git diff 45e2f9c8827024f53ca982c70676614f781205d7 develop README.md
diff --git a/README.md b/README.md
index e965047..214c073 100644
--- a/README.md
+++ b/README.md
@@ -1 +1 @@
-Hello
+Hello develop
$ git diff 45e2f9c8827024f53ca982c70676614f781205d7 master README.md
```

Git の基礎

git diff で、merge-base から README.md にどんな修正を加えたのか確認する。

```
$ git diff 45e2f9c8827024f53ca982c70676614f781205d7 develop README.md
diff --git a/README.md b/README.md
index e965047..214c073 100644
--- a/README.md
+++ b/README.md
@@ -1 +1 @@
-Hello
+Hello develop
$ git diff 45e2f9c8827024f53ca982c70676614f781205d7 master README.md
diff --git a/README.md b/README.md
index e965047..1ffbd88 100644
--- a/README.md
+++ b/README.md
@@ -1 +1,2 @@
-Hello
+Hello master
```

Git の基礎

結果、master と develop でやりたかったことは次の通り。

- master　　: Hello → Hello master に変更
- develop　　: Hello → Hello develop に変更

Git の基礎

結果、master と develop でやりたかったことは次の通り。

- master　　: Hello → Hello master に変更
- develop　　: Hello → Hello develop に変更

今回は ours が master なので、Hello master にすることにしたとする。

Git の基礎

結果、master と develop でやりたかったことは次の通り。

- master : Hello → Hello master に変更
- develop : Hello → Hello develop に変更

今回は ours が master なので、Hello master にすることにしたとする。

```
$ emacs README.md # README.md を修正する
```

Git の基礎

結果、master と develop でやりたかったことは次の通り。

- master : Hello → Hello master に変更
- develop : Hello → Hello develop に変更

今回は ours が master なので、Hello master にすることにしたとする。

```
$ emacs README.md # README.md を修正する

$ cat README.md

Hello master
```

Git の基礎

結果、master と develop でやりたかったことは次の通り。

- master : Hello → Hello master に変更
- develop : Hello → Hello develop に変更

今回は ours が master なので、Hello master にすることにしたとする。

```
$ emacs README.md # README.md を修正する

$ cat README.md

Hello master

$ git merge --continue
```

Git の基礎

結果、master と develop でやりたかったことは次の通り。

- master : Hello → Hello master に変更
- develop : Hello → Hello develop に変更

今回は ours が master なので、Hello master にすることにしたとする。

```
$ emacs README.md # README.md を修正する

$ cat README.md

Hello master

$ git merge --continue
```

これで無事コンフリクトを解消して merge できた。

Git の基礎

コンフリクト解消は人間の手が入るため、必要な差分が消えたり、不要な差分が混入したりする可能性がある。

Git の基礎

コンフリクト解消は人間の手が入るため、必要な差分が消えたり、不要な差分が混入したりする可能性がある。

→ 大前提として、コンフリクトは起きないのが 1 番。

Git の基礎

コンフリクト解消は人間の手が入るため、必要な差分が消えたり、不要な差分が混入したりする可能性がある。

→ 大前提として、コンフリクトは起きないのが 1 番。

Git には **fast-forward merge** という絶対にコンフリクトが起きない merge がある。

Git の基礎

コンフリクト解消は人間の手が入るため、必要な差分が消えたり、不要な差分が混入したりする可能性がある。

→ 大前提として、コンフリクトは起きないのが 1 番。

Git には **fast-forward merge** という絶対にコンフリクトが起きない merge がある。

fast-forward merge とは、merge-base が ours のときに起こる merge のこと。

Git の基礎

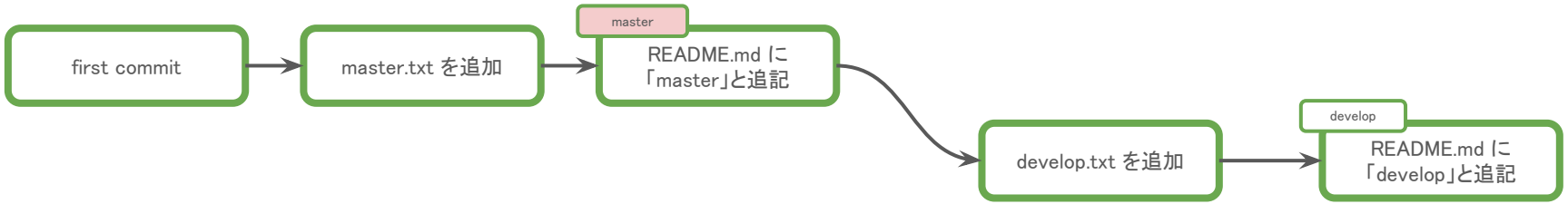
コンフリクト解消は人間の手が入るため、必要な差分が消えたり、不要な差分が混入したりする可能性がある。

→ 大前提として、コンフリクトは起きないのが 1 番。

Git には **fast-forward merge** という絶対にコンフリクトが起きない merge がある。

fast-forward merge とは、merge-base が ours のときに起こる merge のこと。

さっきの例で言うと、↓のような状態で merge すると fast-foward merge が発生する。



Git の基礎

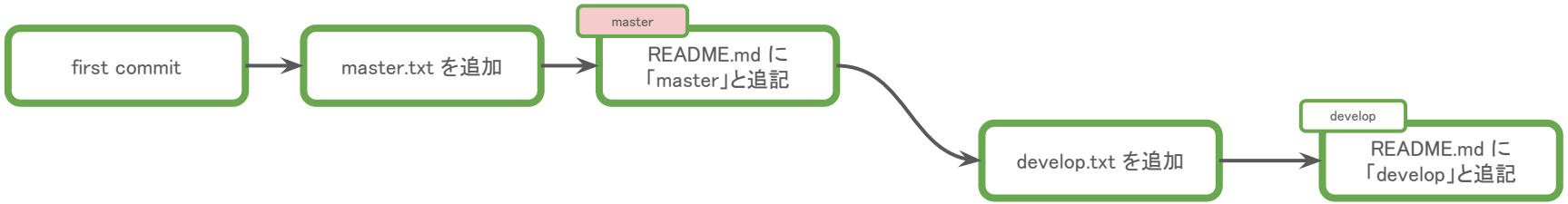
コンフリクト解消は人間の手が入るため、必要な差分が消えたり、不要な差分が混入したりする可能性がある。

→ 大前提として、コンフリクトは起きないのが 1 番。

Git には **fast-forward merge** という絶対にコンフリクトが起きない merge がある。

fast-forward merge とは、merge-base が ours のときに起こる merge のこと。

さっきの例で言うと、↓のような状態で merge すると fast-foward merge が発生する。



この状態で merge すると、Git は merge 作業をサボって、master を develop を同一にするだけになる。

Git の基礎

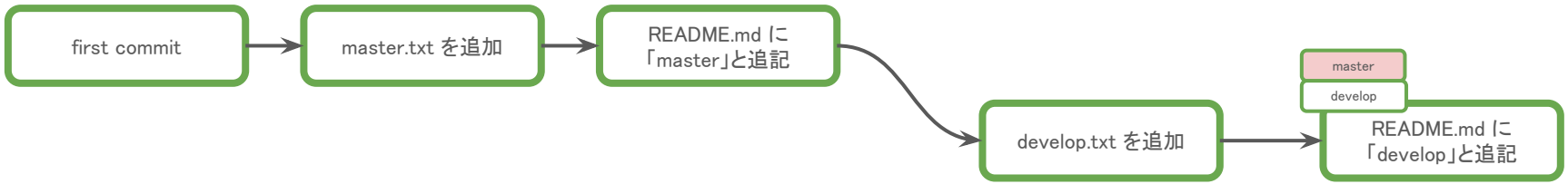
コンフリクト解消は人間の手が入るため、必要な差分が消えたり、不要な差分が混入したりする可能性がある。

→ 大前提として、コンフリクトは起きないのが 1 番。

Git には **fast-forward merge** という絶対にコンフリクトが起きない merge がある。

fast-forward merge とは、merge-base が ours のときに起こる merge のこと。

さっきの例で言うと、↓のような状態で merge すると fast-foward merge が発生する。



この状態で merge すると、Git は merge 作業をサボって、master を develop を同一にするだけになる。

一旦休憩 5 分くらい

休憩中の余談

Git を使う上で、知っておくと便利なちょっとした Tips を紹介

休憩中の余談

Git を使う上で、知っておくと便利なちょっとした Tips を紹介

実は CUI のプロンプトは自分でカスタマイズできる。

休憩中の余談

Git を使う上で、知っておくと便利なちょっとした Tips を紹介

実は CUI のプロンプトは自分でカスタマイズできる。

例えば bash なら、$PS1 という環境変数でプロンプトの表示を制御している。

休憩中の余談

Git を使う上で、知っておくと便利なちょっとした Tips を紹介

実は CUI のプロンプトは自分でカスタマイズできる。

例えば bash なら、**$PS1** という環境変数でプロンプトの表示を制御している。

```
$ PS1="hoge$ "

hoge$
```

休憩中の余談

Git を使う上で、知っておくと便利なちょっとした Tips を紹介

実は CUI のプロンプトは自分でカスタマイズできる。

例えば bash なら、**$PS1** という環境変数でプロンプトの表示を制御している。

```
$ PS1="hoge$ "

hoge$
```

ちなみに PS は Prompt String の略。

休憩中の余談

Git を使う上で、知っておくと便利なちょっとした Tips を紹介

実は CUI のプロンプトは自分でカスタマイズできる。

例えば bash なら、$PS1 という環境変数でプロンプトの表示を制御している。

```
$ PS1="hoge$ "
hoge$
```

ちなみに PS は Prompt String の略。

普段表示されているプロンプトは「プライマリプロンプト」と呼ばれているため、1 が付いている

休憩中の余談

Git を使う上で、知っておくと便利なちょっとした Tips を紹介

実は CUI のプロンプトは自分でカスタマイズできる。

例えば bash なら、**$PS1** という環境変数でプロンプトの表示を制御している。

```
$ PS1="hoge$ "
hoge$
```

ちなみに PS は Prompt String の略。

普段表示されているプロンプトは「プライマリプロンプト」と呼ばれているため、1 が付いている

プロンプトは特殊な物を含めると 4 種類あり、それぞれ $PS1 ~ $PS4 で設定できる。

休憩中の余談

プロンプトは $PS1 は bash が評価した上で画面に表示している。

休憩中の余談

プロンプトは $PS1 は bash が評価した上で画面に表示している。

つまりコマンド置換を使えば、**コマンドの実行結果をプロンプトに含めることができる**。

休憩中の余談

プロンプトは $PS1 は bash が評価した上で画面に表示している。

つまりコマンド置換を使えば、**コマンドの実行結果をプロンプトに含めることができる**。

$PS1 に↓を設定すれば、現在の branch を表示できる。

```
$(git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* ¥(.*¥)/(¥1)/')
```

休憩中の余談

プロンプトは $PS1 は bash が評価した上で画面に表示している。

つまりコマンド置換を使えば、**コマンドの実行結果をプロンプトに含めることができる**。

$PS1 に↓を設定すれば、現在の branch を表示できる。

```
$(git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* ¥(.*¥)/(¥1)/')
```

↓普段使ってる $PS1 晒しておきます。

```
PS1="[$?]¥u¥[¥e[2m¥]$(git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* ¥(.*¥)/(¥1)/')¥[¥e[0m¥]:¥W$ "
```

休憩中の余談

プロンプトは $PS1 は bash が評価した上で画面に表示している。

つまりコマンド置換を使えば、**コマンドの実行結果をプロンプトに含めることができる**。

$PS1 に↓を設定すれば、現在の branch を表示できる。

```
$(git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* ¥(.*¥)/(¥1)/')
```

↓普段使ってる $PS1 晒しておきます。

```
PS1="[$?]¥u¥[¥e[2m¥]$(git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* ¥(.*¥)/(¥1)/')¥[¥e[0m¥]:¥W$ "
```

休憩中の余談

プロンプトは $PS1 は bash が評価した上で画面に表示している。

つまりコマンド置換を使えば、**コマンドの実行結果をプロンプトに含めることができる**。

$PS1 に↓を設定すれば、現在の branch を表示できる。

```
$(git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* ¥(.*¥)/(¥1)/')
```

↓普段使ってる $PS1 晒しておきます。

```
PS1="[$?]¥u¥[¥e[2m¥]$(git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* ¥(.*¥)/(¥1)/')¥[¥e[0m¥]:¥W$ "
```

```
[0]shumon.fujita(master):hoge$
```

プロンプトがカスタマイズできることを知っておくと、例えばコマンドラインのスクショを撮りたいときに、

いちいちユーザ名を隠したりしなくていいので、なにかと便利。

Git によるチーム開発のいろは

Git 誕生秘話

チーム開発の話をする前に、なぜこの世に Git が存在するかご存知ですか

Git 誕生秘話

チーム開発の話をする前に、なぜこの世に Git が存在するかご存知ですか

→ Linux を開発するため

Git 誕生秘話

チーム開発の話をする前に、なぜこの世に Git が存在するかご存知ですか

→ Linux を開発するため

もともと Linux はメーリスに投稿されたパッチを、気合いで適用しながら開発していた

Git 誕生秘話

チーム開発の話をする前に、なぜこの世に Git が存在するかご存知ですか

→ Linux を開発するため

もともと Linux はメーリスに投稿されたパッチを、気合いで適用しながら開発していた

→ 2002 年、「BitKeeper」という有料の VCS を無料で使わせてもらえることになった

Git 誕生秘話

チーム開発の話をする前に、なぜこの世に Git が存在するかご存知ですか

→ Linux を開発するため

もともと Linux はメーリスに投稿されたパッチを、気合いで適用しながら開発していた

→ 2002 年、「BitKeeper」という有料の VCS を無料で使わせてもらえることになった

→ 2005 年、なんやかんやあって、あくまで厚意で無料で使えていただけの BitKeeper を有料化されてしまった

Git 誕生秘話

チーム開発の話をする前に、なぜこの世に Git が存在するかご存知ですか

→ Linux を開発するため

もともと Linux はメーリスに投稿されたパッチを、気合いで適用しながら開発していた

→ 2002 年、「BitKeeper」という有料の VCS を無料で使わせてもらえることになった

→ 2005 年、なんやかんやあって、あくまで厚意で無料で使えていただけの BitKeeper を有料化されてしまった

→ 他の VCS を探す必要がある

Git 誕生秘話

チーム開発の話をする前に、なぜこの世に Git が存在するかご存知ですか

→ Linux を開発するため

もともと Linux はメーリスに投稿されたパッチを、気合いで適用しながら開発していた

→ 2002 年、「BitKeeper」という有料の VCS を無料で使わせてもらえることになった

→ 2005 年、なんやかんやあって、あくまで厚意で無料で使えていただけの BitKeeper を有料化されてしまった

→ 他の VCS を探す必要がある

→ 他の VCS は Linux ぐらいデカいプロジェクトを扱うには遅すぎるし、開発者が大勢いる状態に対応できない

Git 誕生秘話

チーム開発の話をする前に、なぜこの世に Git が存在するかご存知ですか

→ Linux を開発するため

もともと Linux はメーリスに投稿されたパッチを、気合いで適用しながら開発していた

→ 2002 年、「BitKeeper」という有料の VCS を無料で使わせてもらえることになった

→ 2005 年、なんやかんやあって、あくまで厚意で無料で使えていただけの BitKeeper を有料化されてしまった

→ 他の VCS を探す必要がある

→ 他の VCS は Linux ぐらいデカいプロジェクトを扱うには遅すぎるし、開発者が大勢いる状態に対応できない

→ リーナス・トーバルズ「自作するぞ」

Git 誕生秘話

チーム開発の話をする前に、なぜこの世に Git が存在するかご存知ですか

→ Linux を開発するため

もともと Linux はメーリスに投稿されたパッチを、気合いで適用しながら開発していた

→ 2002 年、「BitKeeper」という有料の VCS を無料で使わせてもらえることになった

→ 2005 年、なんやかんやあって、あくまで厚意で無料で使えていただけの BitKeeper を有料化されてしまった

→ 他の VCS を探す必要がある

→ 他の VCS は Linux ぐらいデカいプロジェクトを扱うには遅すぎるし、開発者が大勢いる状態に対応できない

→ リーナス・トーバルズ「自作するぞ」

詳しくは Git の公式ドキュメントにある Git略史 を読んでみてね

Git 誕生秘話

それから 2 週間すぎたころ

BitKeeper の流れを汲んだ高速な VCS

Git

が誕生しました

Git の特徴

というわけで、Git には Linux 並にデカいソフトウェアを大人数で開発するために生まれた。

Git の特徴

というわけで、Git には Linux 並にデカいソフトウェアを大人数で開発するために生まれた。

その結果、Git 以前の VCS に比べて次のような特徴がある。

- 高速な merge と checkout
- 分散型
- branch

高速な merge と checkout

Git の merge と checkout は、実はかなり高速

高速な merge と checkout

Git の merge と checkout は、実はかなり高速

特に、**「履歴の遠さ」**は merge や checkout の時間に**影響を与えない**

高速な merge と checkout

Git の merge と checkout は、実はかなり高速

特に、**「履歴の遠さ」**は merge や checkout の時間に**影響を与えない**

変更されているファイルの数が支配的なのが特徴

高速な merge と checkout

Git の merge と checkout は、実はかなり高速

特に、**「履歴の遠さ」**は merge や checkout の時間に**影響を与えない**

変更されているファイルの数が支配的なのが特徴

→ 理由は後述の Git の内部構造を聞くと分かります

分散型

Git の解説で毎回聞くやつ。

分散型

Git の解説で毎回聞くやつ。

世の中のVCS(バージョン管理システム/Version Control System)には大きく分けて集中型と分散型の2つがある。

分散型

Git の解説で毎回聞くやつ。

世の中のVCS(バージョン管理システム/Version Control System)には大きく分けて集中型と分散型の2つがある。

分散型のVCSは**「リポジトリの全履歴を含めた完全なコピーをローカルに持つ」**という意味。

分散型

Git の解説で毎回聞くやつ。

世の中のVCS(バージョン管理システム/Version Control System)には大きく分けて集中型と分散型の2つがある。

分散型のVCSは**「リポジトリの全履歴を含めた完全なコピーをローカルに持つ」**という意味。

例えば集中型のSubversionは、リポジトリは完全にサーバが管理してる。

分散型

Git の解説で毎回聞くやつ。

世の中のVCS(バージョン管理システム/Version Control System)には大きく分けて集中型と分散型の2つがある。

分散型のVCSは**「リポジトリの全履歴を含めた完全なコピーをローカルに持つ」**という意味。

例えば集中型のSubversionは、リポジトリは完全にサーバが管理してる。

自分が触りたいファイルだけをローカルにコピーし、修正後サーバにpushする 。

分散型

Git の解説で毎回聞くやつ。

世の中のVCS(バージョン管理システム/Version Control System)には大きく分けて集中型と分散型の2つがある。

分散型のVCSは**「リポジトリの全履歴を含めた完全なコピーをローカルに持つ」**という意味。

例えば集中型のSubversionは、リポジトリは完全にサーバが管理してる。

自分が触りたいファイルだけをローカルにコピーし、修正後サーバにpushする 。

その間は他人は自分が使ってるファイルを編集することはできない。

分散型

Git の解説で毎回聞くやつ。

世の中のVCS(バージョン管理システム/Version Control System)には大きく分けて集中型と分散型の2つがある。

分散型のVCSは**「リポジトリの全履歴を含めた完全なコピーをローカルに持つ」**という意味。

例えば集中型のSubversionは、リポジトリは完全にサーバが管理してる。

自分が触りたいファイルだけをローカルにコピーし、修正後サーバにpushする 。

その間は他人は自分が使ってるファイルを編集することはできない。

→ 大人数で開発すると開発速度が落ちる

分散型

Git の解説で毎回聞くやつ。

世の中のVCS(バージョン管理システム/Version Control System)には大きく分けて集中型と分散型の2つがある。

分散型のVCSは**「リポジトリの全履歴を含めた完全なコピーをローカルに持つ」**という意味。

例えば集中型のSubversionは、リポジトリは完全にサーバが管理してる。

自分が触りたいファイルだけをローカルにコピーし、修正後サーバにpushする 。

その間は他人は自分が使ってるファイルを編集することはできない。

→ 大人数で開発すると開発速度が落ちる

→ でも分散型のGitは誰がどんな修正をしていても無視して進めることができる

branch

Git は branch 機能のおかげで、大人数で並行して開発を進めることができる。

branch

Git は branch 機能のおかげで、大人数で並行して開発を進めることができる。

でも無秩序に branch を使うとコンフリクトしまくるし、どこでどの機能が実装されているのか分からない。

branch

Git は branch 機能のおかげで、大人数で並行して開発を進めることができる。

でも無秩序に branch を使うとコンフリクトしまくるし、どこでどの機能が実装されているのか分からない。

→ 複雑なコンフリクトの仕方をすると、解消に失敗してバグが混入するかも。

branch

Git は branch 機能のおかげで、大人数で並行して開発を進めることができる。

でも無秩序に branch を使うとコンフリクトしまくるし、どこでどの機能が実装されているのか分からない。

→ 複雑なコンフリクトの仕方をすると、解消に失敗してバグが混入するかも。

→ どの branch にどの機能が実装されているか分からないので、最新版がどれなのか分からない。

branch

Git は branch 機能のおかげで、大人数で並行して開発を進めることができる。

でも無秩序に branch を使うとコンフリクトしまくるし、どこでどの機能が実装されているのか分からない。

→ 複雑なコンフリクトの仕方をすると、解消に失敗してバグが混入するかも。

→ どの branch にどの機能が実装されているか分からないので、最新版がどれなのか分からない。

→ そういった問題を防ぐため、branch運用には、いくつかの方法論がある。

Git Flow

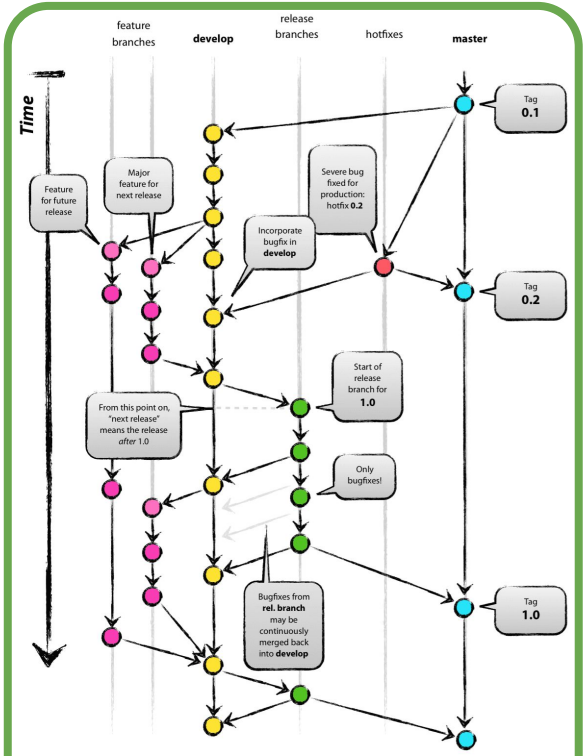
Git Flow

Git では、おそらく最も一般的な branch 運用。

Git Flow

Git では、おそらく最も一般的な branch 運用。

皆でリモートリポジトリを1つに決めて、Git を中央集権的に扱えるようにした。



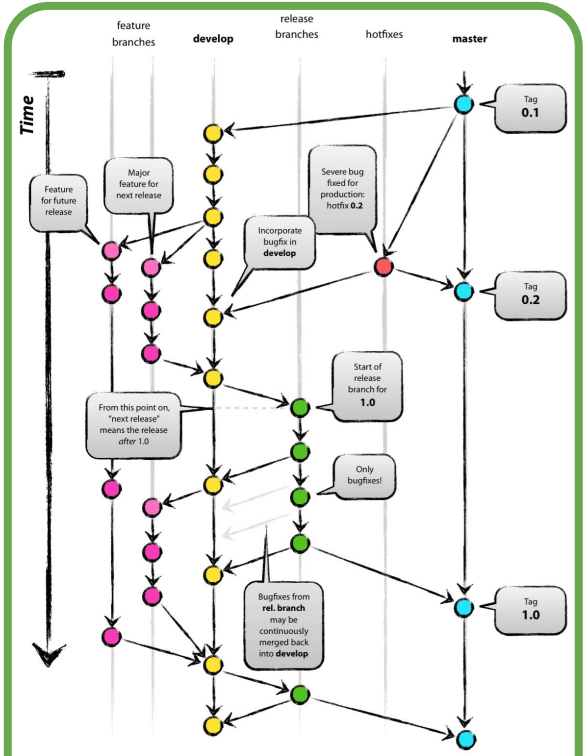
https://nvie.com/posts/a-successful-git-branching-model/

Git Flow

Git では、おそらく最も一般的な branch 運用。

皆でリモートリポジトリを1つに決めて、Git を中央集権的に扱えるようにした。

→ 集中型と分散型の良いとこ取りができる。



https://nvie.com/posts/a-successful-git-branching-model/

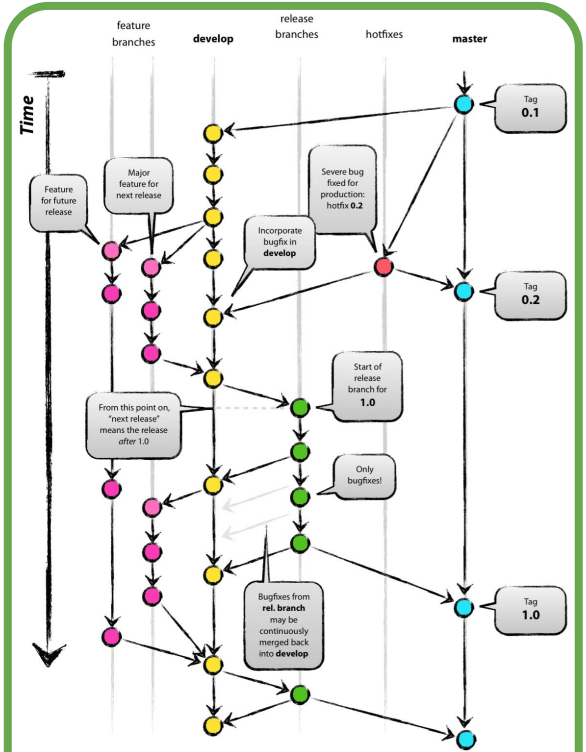
Git Flow

Git では、おそらく最も一般的な branch 運用。

皆でリモートリポジトリを1つに決めて、Git を中央集権的に扱えるようにした。

→ 集中型と分散型の良いとこ取りができる。

大体のチームは、Git Flow をちょっとアレンジして使ってるんじゃないかな。



https://nvie.com/posts/a-successful-git-branching-model/

Git Flow

Git では、おそらく最も一般的な branch 運用。

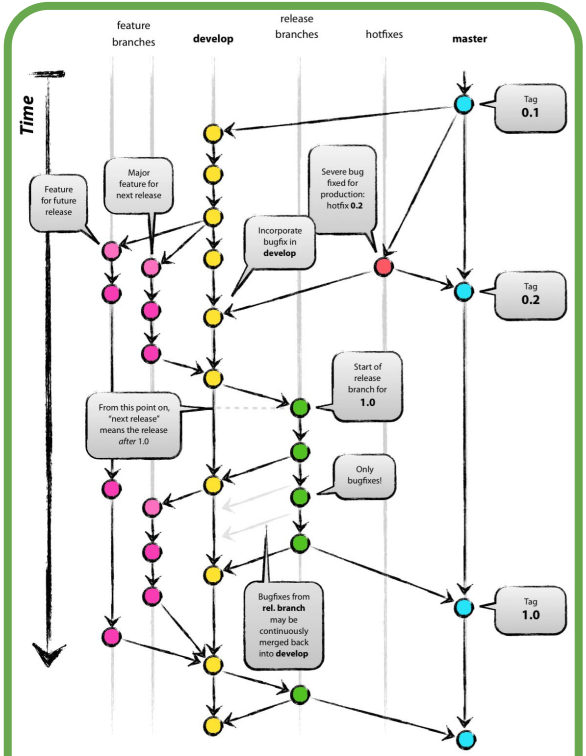
皆でリモートリポジトリを1つに決めて、Git を中央集権的に扱えるようにした。

→ 集中型と分散型の良いとこ取りができる。

大体のチームは、Git Flow をちょっとアレンジして使ってるんじゃないかな。

Git Flow という名前を知らなくても、慣習として Git Flow と同じようなことを

している人も多いと思う。



https://nvie.com/posts/a-successful-git-branching-model/

Git Flow

Git Flow では、まずリモートリポジトリを1つに決める必要がある。

Git Flow

Git Flow では、まずリモートリポジトリを1つに決める必要がある。

GitHub なり GitLab なり BitBucket なりなんでもいい。ちなみに弊社では基本的にGitHub。

Git Flow

Git Flow では、まずリモートリポジトリを1つに決める必要がある。

GitHub なり GitLab なり BitBucket なりなんでもいい。ちなみに弊社では基本的にGitHub。

とにかくそのリモートリポジトリを開発者同士で口裏を合わせて常に正とみなす。

Git Flow

Git Flow では、まずリモートリポジトリを1つに決める必要がある。

GitHub なり GitLab なり BitBucket なりなんでもいい。ちなみに弊社では基本的にGitHub。

とにかくそのリモートリポジトリを開発者同士で口裏を合わせて常に正とみなす。

そんなの当たり前すぎて、逆にリモートリポジトリが複数ある状態なんて、よく分からない人もいるかも。

Git Flow

Git Flow では、まずリモートリポジトリを1つに決める必要がある。

GitHub なり GitLab なり BitBucket なりなんでもいい。ちなみに弊社では基本的にGitHub。

とにかくそのリモートリポジトリを開発者同士で口裏を合わせて常に正とみなす。

そんなの当たり前すぎて、逆にリモートリポジトリが複数ある状態なんて、よく分からない人もいるかも。

でも Git では、複数のリモートリポジトリを持つことができる。

Git Flow

Git Flow では、まずリモートリポジトリを1つに決める必要がある。

GitHub なり GitLab なり BitBucket なりなんでもいい。ちなみに弊社では基本的にGitHub。

とにかくそのリモートリポジトリを開発者同士で口裏を合わせて常に正とみなす。

そんなの当たり前すぎて、逆にリモートリポジトリが複数ある状態なんて、よく分からない人もいるかも。

でも Git では、複数のリモートリポジトリを持つことができる。

- GitHub ができる前から Git 使ってる OSS は自前 Git 鯖と GitHub を両方使ってたりする。

Git Flow

Git Flow では、まずリモートリポジトリを1つに決める必要がある。

GitHub なり GitLab なり BitBucket なりなんでもいい。ちなみに弊社では基本的にGitHub。

とにかくそのリモートリポジトリを開発者同士で口裏を合わせて常に正とみなす。

そんなの当たり前すぎて、逆にリモートリポジトリが複数ある状態なんて、よく分からない人もいるかも。

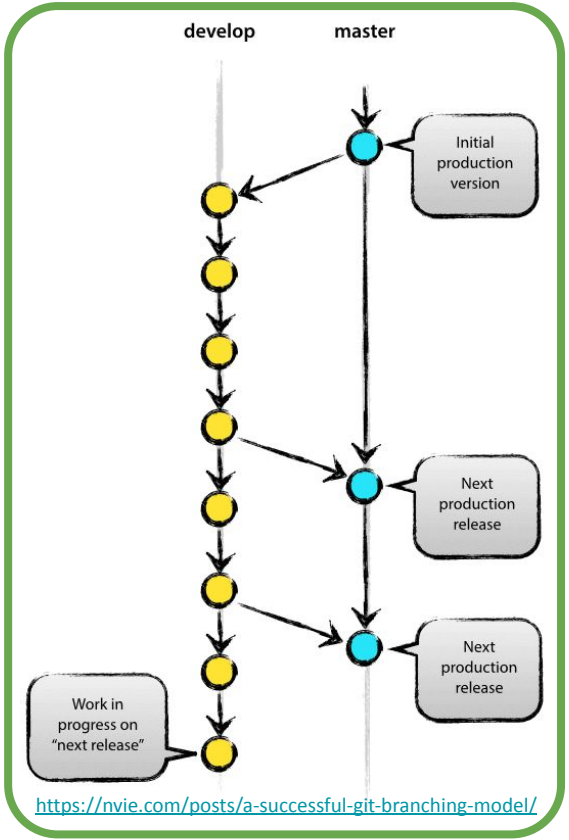
でも Git では、複数のリモートリポジトリを持つことができる。

- GitHub ができる前から Git 使ってる OSS は自前 Git 鯖と GitHub を両方使ってたりする。
- fork したリポジトリを、fork 元のリポジトリに追従したいときも使ったりする。

Git Flow – developとmaster

Git Flowで最も重要なのが、masterとdevelopの関係。

（去年 master じゃなくて main を使おうという動きがありましたが、 原典 は master を使っているのでこのまま行きます）

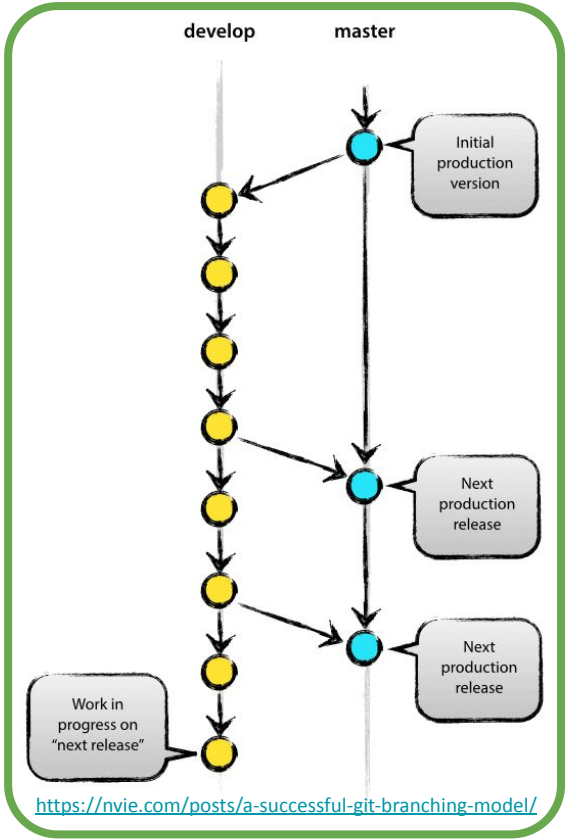


Git Flow – developとmaster

Git Flowで最も重要なのが、masterとdevelopの関係。

（去年 master じゃなくて main を使おうという動きがありましたが、 原典 は master を使っているのでこのまま行きます）

開発は develop で行い、master はリリース時に初めて commit される。



https://nvie.com/posts/a-successful-git-branching-model/

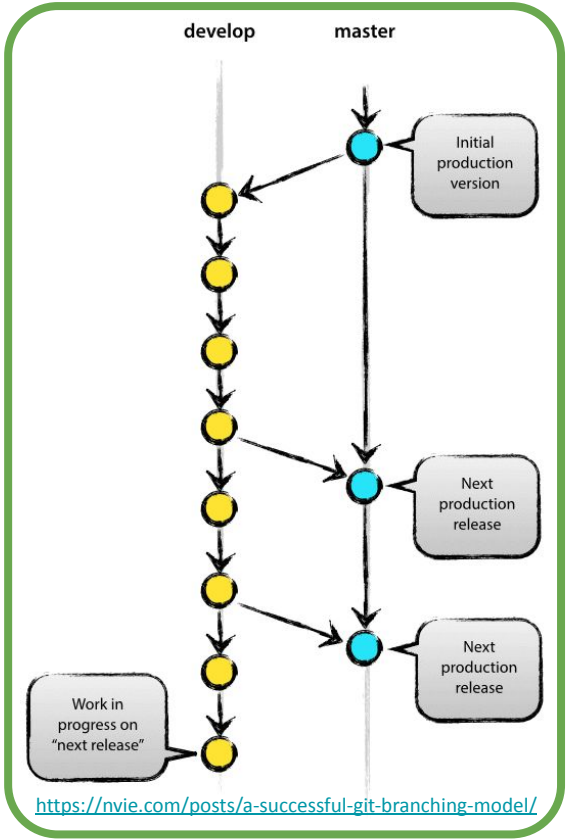
Git Flow – developとmaster

Git Flowで最も重要なのが、masterとdevelopの関係。

（去年 master じゃなくて main を使おうという動きがありましたが、 原典 は master を使っているのでこのまま行きます）

開発は develop で行い、master はリリース時に初めて commit される。

master の commit にはリリースごとに tag を打つ。



Git Flow – developとmaster

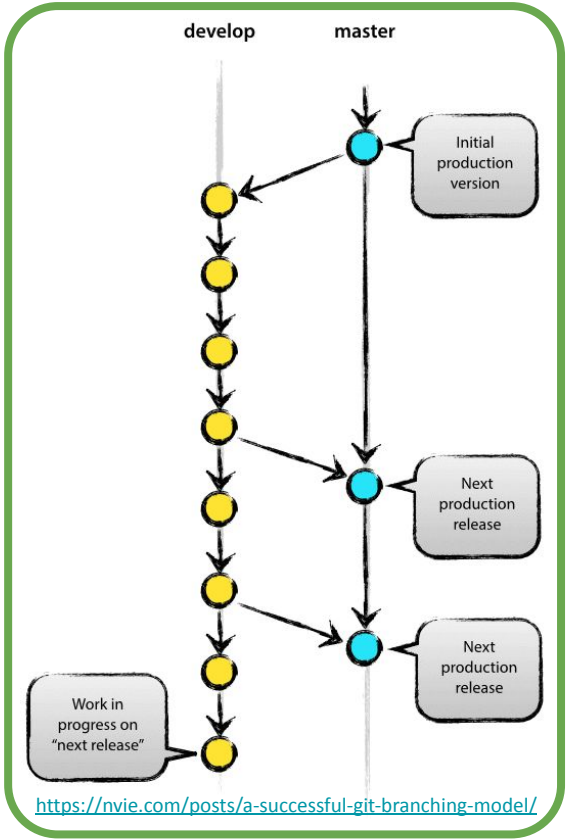
Git Flowで最も重要なのが、masterとdevelopの関係。

（去年 master じゃなくて main を使おうという動きがありましたが、 原典 は master を使っているのでこのまま行きます）

開発は develop で行い、master はリリース時に初めて commit される。

master の commit にはリリースごとに tag を打つ。

masterの先頭が常にプロダクトの最新のリリースになる。



https://nvie.com/posts/a-successful-git-branching-model/

Git Flow – developとmaster

Git Flowで最も重要なのが、masterとdevelopの関係。

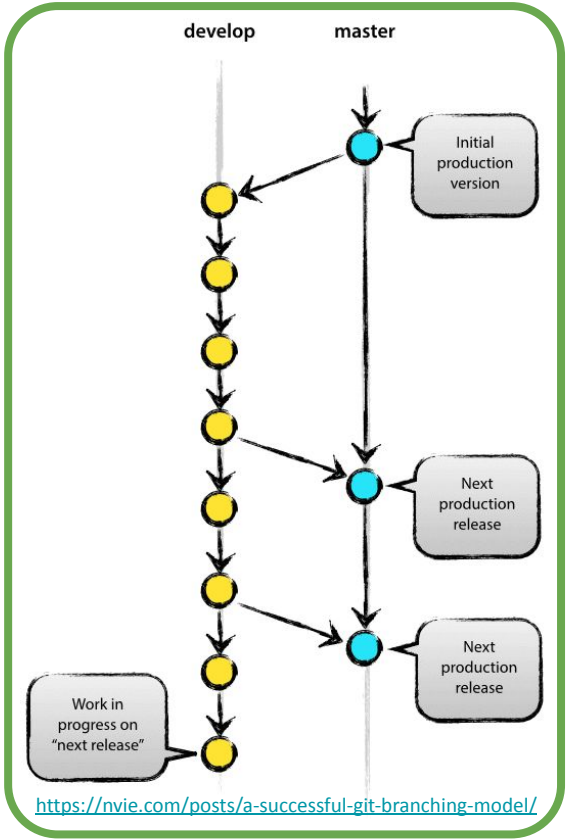
（去年 master じゃなくて main を使おうという動きがありましたが、 原典 は master を使っているのでこのまま行きます）

開発は develop で行い、master はリリース時に初めて commit される。

master の commit にはリリースごとに tag を打つ。

masterの先頭が常にプロダクトの最新のリリースになる。

（個人的には master への commit は squash が好き）



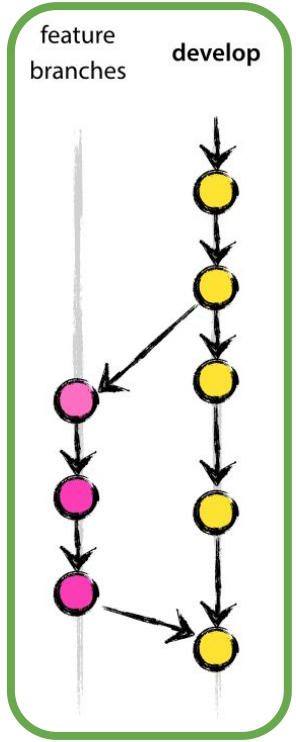
Git Flow – feature

ここから先は、全部 develop と master をサポートするための branch 。

Git Flow – feature

ここから先は、全部 develop と master をサポートするための branch 。

feature は複数人で develop を開発するときに使う。



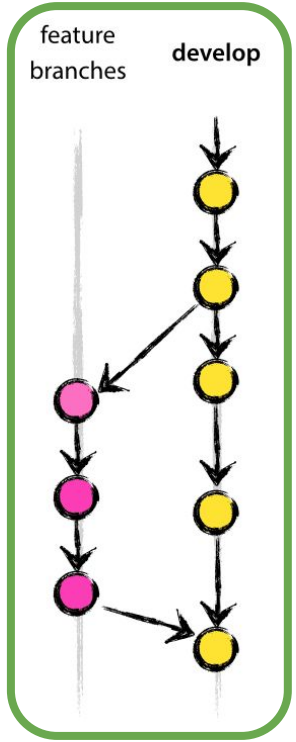
https://nvie.com/posts/a-successful-git-branching-model/

Git Flow – feature

ここから先は、全部 develop と master をサポートするための branch 。

feature は複数人で develop を開発するときに使う。

1機能ごとに develop から branch を切る。機能の開発が終われば develop に merge する。



https://nvie.com/posts/a-successful-git-branching-model/

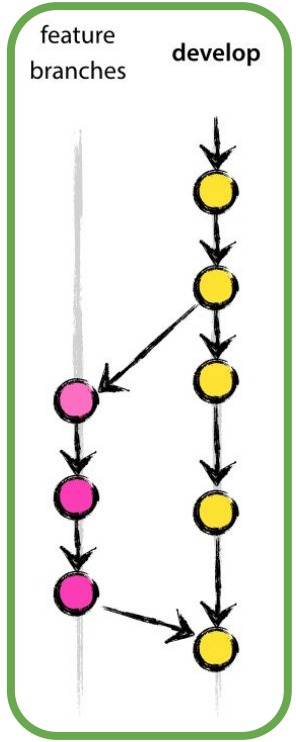
Git Flow – feature

ここから先は、全部 develop と master をサポートするための branch 。

feature は複数人で develop を開発するときに使う。

1機能ごとに develop から branch を切る。機能の開発が終われば develop に merge する。

「1 機能 = 1 feature」ルールは守ろう。複数の機能にまたがる feature はデメリットが多い。



https://nvie.com/posts/a-successful-git-branching-model/

Git Flow – feature

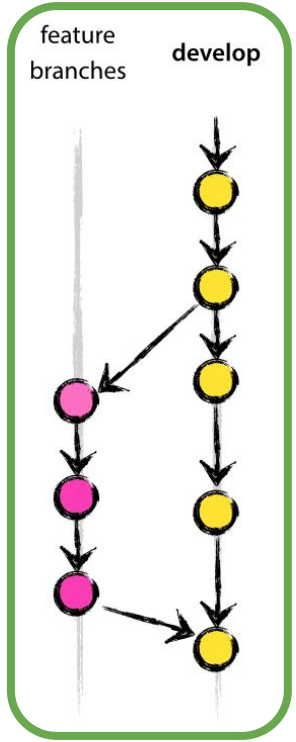
ここから先は、全部 develop と master をサポートするための branch 。

feature は複数人で develop を開発するときに使う。

1機能ごとに develop から branch を切る。機能の開発が終われば develop に merge する。

「1 機能 = 1 feature」ルールは守ろう。複数の機能にまたがる feature はデメリットが多い。

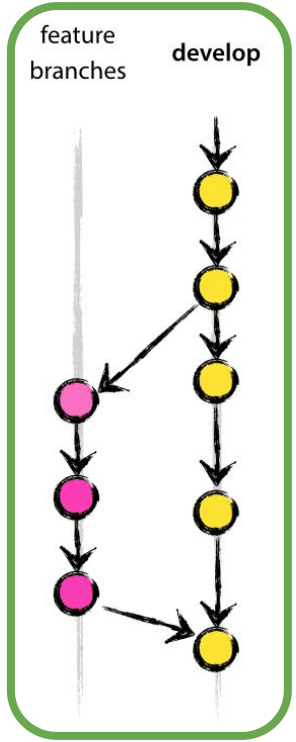
- どの機能がどの branch に入っているのか分かりにくい。



https://nvie.com/posts/a-successful-git-branching-model/

Git Flow – feature

ここから先は、全部 develop と master をサポートするための branch 。



feature は複数人で develop を開発するときに使う。

1機能ごとに develop から branch を切る。機能の開発が終われば develop に merge する。

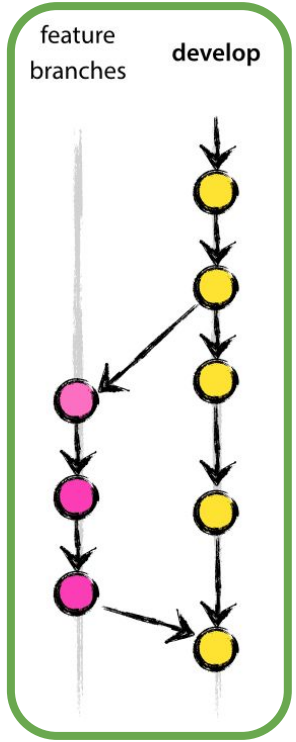
「1 機能 = 1 feature」ルールは守ろう。複数の機能にまたがる feature はデメリットが多い。

- どの機能がどの branch に入っているのか分かりにくい。
- 差分が大きくなってコンフリクトしやすい。

https://nvie.com/posts/a-successful-git-branching-model/

Git Flow – feature

ここから先は、全部 develop と master をサポートするための branch 。



https://nvie.com/posts/a-successful-git-branching-model/

feature は複数人で develop を開発するときに使う。

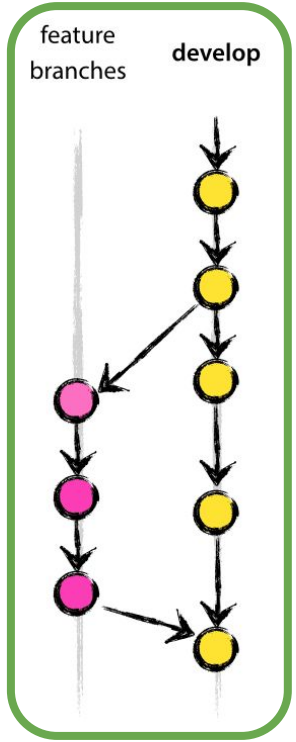
1機能ごとに develop から branch を切る。機能の開発が終われば develop に merge する。

「1 機能 = 1 feature」ルールは守ろう。複数の機能にまたがる feature はデメリットが多い。

- どの機能がどの branch に入っているのか分かりにくい。
- 差分が大きくなってコンフリクトしやすい。
- revert する際は、機能単位で revert したいことがほとんど。

Git Flow – feature

ここから先は、全部 develop と master をサポートするための branch 。



https://nvie.com/posts/a-successful-git-branching-model/

feature は複数人で develop を開発するときに使う。

1機能ごとに develop から branch を切る。機能の開発が終われば develop に merge する。

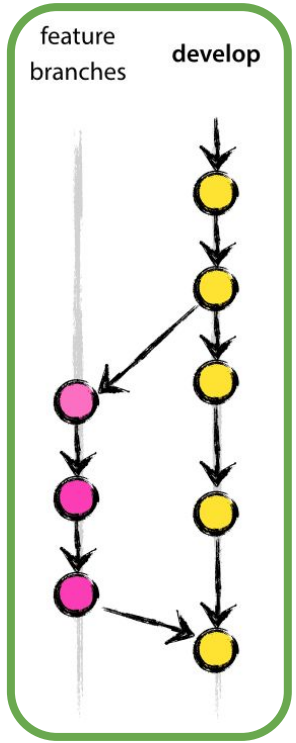
「1 機能 = 1 feature」ルールは守ろう。複数の機能にまたがる feature はデメリットが多い。

- どの機能がどの branch に入っているのか分かりにくい。
- 差分が大きくなってコンフリクトしやすい。
- revert する際は、機能単位で revert したいことがほとんど。

大きい機能の場合は feature からさらに feature を切ったりもする。

Git Flow – feature

ここから先は、全部 develop と master をサポートするための branch 。



feature は複数人で develop を開発するときに使う。

1機能ごとに develop から branch を切る。機能の開発が終われば develop に merge する。

「1 機能 = 1 feature」ルールは守ろう。複数の機能にまたがる feature はデメリットが多い。

- どの機能がどの branch に入っているのか分かりにくい。
- 差分が大きくなってコンフリクトしやすい。
- revert する際は、機能単位で revert したいことがほとんど。

大きい機能の場合は feature からさらに feature を切ったりもする。

概念実証のような、最終的に製品に入れるか分からないものもここ。

https://nvie.com/posts/a-successful-git-branching-model/

Git Flow – release

release は develop から master へ merge するための準備をする branch 。

Git Flow – release

release は develop から master へ merge するための準備をする branch 。

チームによっては、staging という名前で呼んでるかも。

Git Flow – release

release は develop から master へ merge するための準備をする branch 。

チームによっては、staging という名前で呼んでるかも。

具体的にどんな「準備」をするかは、プロジェクトによってまちまち

Git Flow – release

release は develop から master へ merge するための準備をする branch 。

チームによっては、staging という名前で呼んでるかも。

具体的にどんな「準備」をするかは、プロジェクトによってまちまち

- 新規実装を凍結し、bugfix のみに使う

Git Flow – release

release は develop から master へ merge するための準備をする branch 。

チームによっては、staging という名前で呼んでるかも。

具体的にどんな「準備」をするかは、プロジェクトによってまちまち

- 新規実装を凍結し、bugfix のみに使う
- version 表記やタイムスタンプを更新する

Git Flow – release

release は develop から master へ merge するための準備をする branch 。

チームによっては、staging という名前で呼んでるかも。

具体的にどんな「準備」をするかは、プロジェクトによってまちまち

- 新規実装を凍結し、bugfix のみに使う
- version 表記やタイムスタンプを更新する
- Apple や Google の審査のために使う

Git Flow – release

release は develop から master へ merge するための準備をする branch 。

チームによっては、staging という名前で呼んでるかも。

具体的にどんな「準備」をするかは、プロジェクトによってまちまち

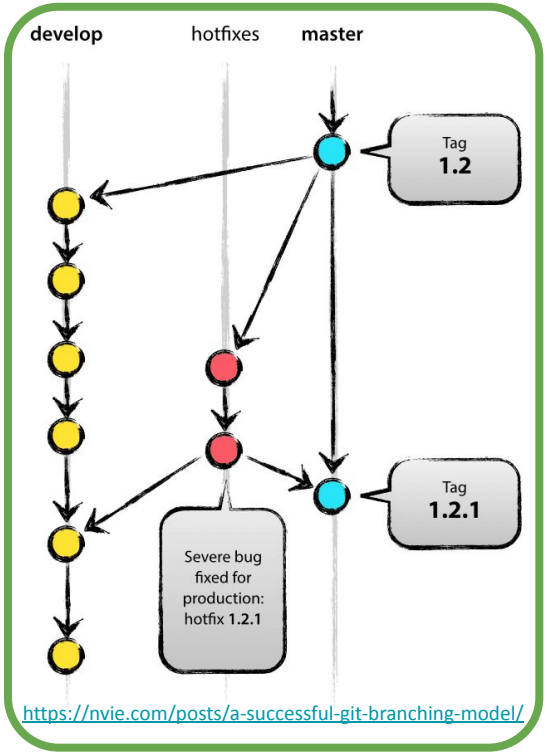
- 新規実装を凍結し、bugfix のみに使う
- version 表記やタイムスタンプを更新する
- Apple や Google の審査のために使う

いずれにしても、release は master へ commit する(= ユーザの手に渡る)までの、最終段階にあたるので、

release で新機能を追加するのはご法度。

Git Flow – hotfix

hotfix は本番環境で発生したバグの内、緊急性の高いものを修正するための branch。

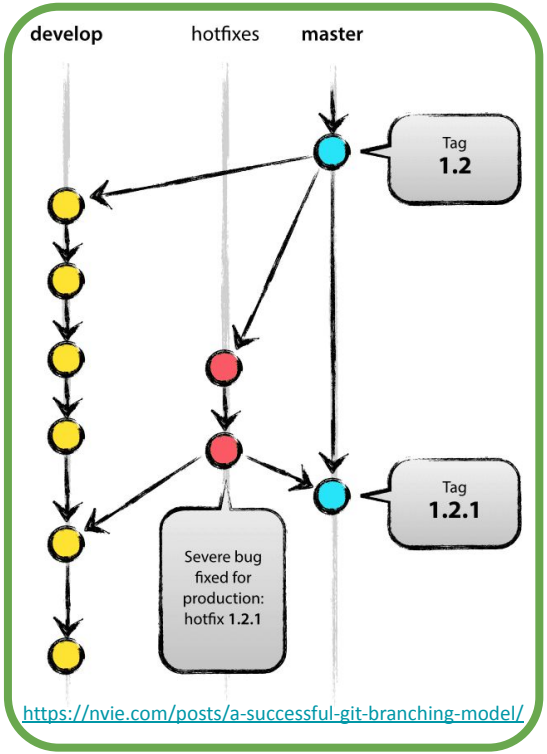


https://nvie.com/posts/a-successful-git-branching-model/

Git Flow – hotfix

hotfix は本番環境で発生したバグの内、緊急性の高いものを修正するための branch。

develop 以外で、唯一 master から切られる branch 。



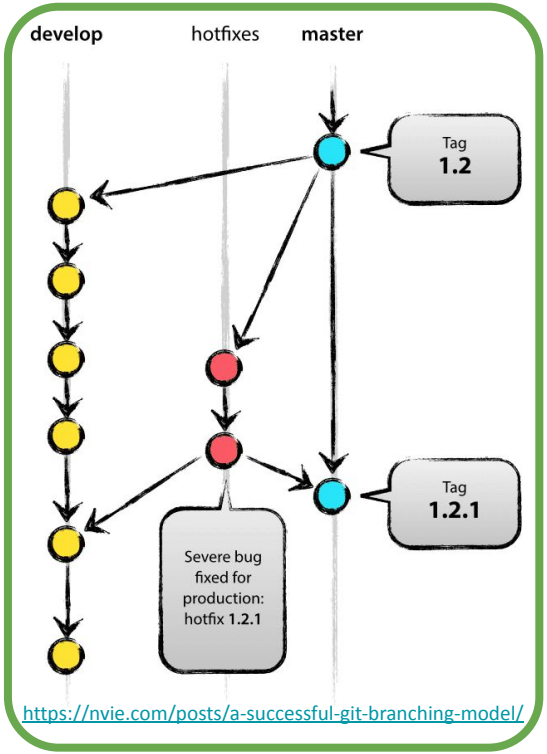
https://nvie.com/posts/a-successful-git-branching-model/

Git Flow – hotfix

hotfix は本番環境で発生したバグの内、緊急性の高いものを修正するための branch。

develop 以外で、唯一 master から切られる branch 。

修正が完了したら master と develop (or release)に merge する。



https://nvie.com/posts/a-successful-git-branching-model/

Git Flow – hotfix

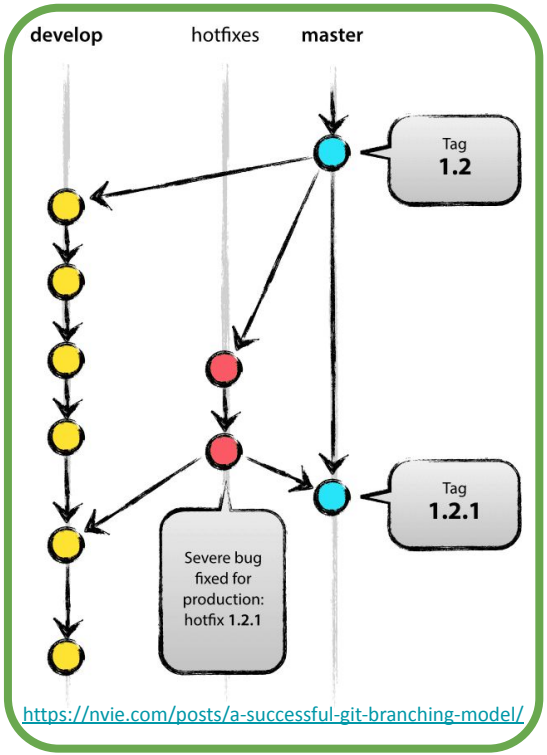
hotfix は本番環境で発生したバグの内、緊急性の高いものを修正するための branch。

develop 以外で、唯一 master から切られる branch 。

修正が完了したら master と develop (or release)に merge する。

hotfix から master を更新した場合は、patch バージョンを上げることが多い。

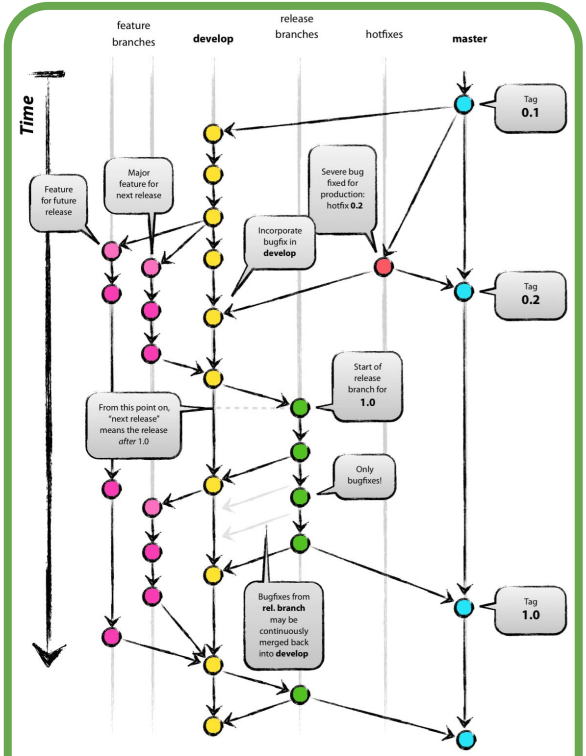
(1.2 → 1.3 ではなく、1.2 → 1.2.1)



https://nvie.com/posts/a-successful-git-branching-model/

Git Flow

ここまでの話を全部まとめると、右の図になる。

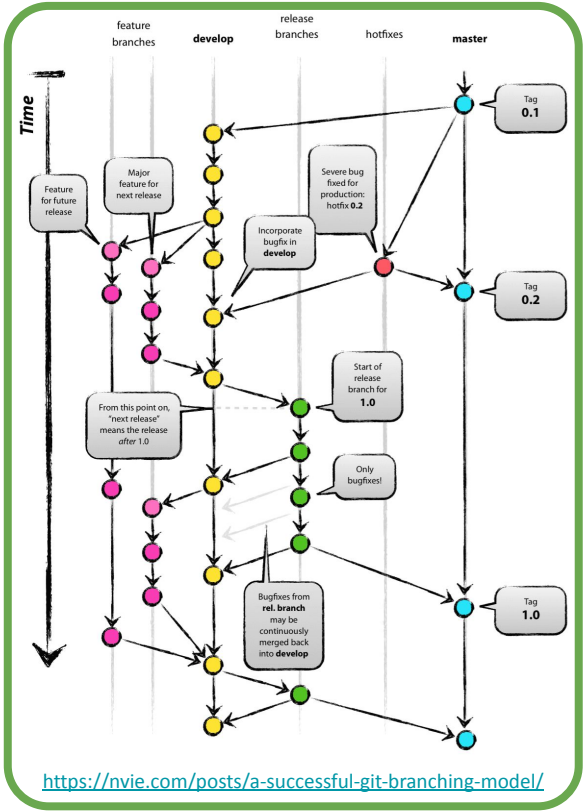


https://nvie.com/posts/a-successful-git-branching-model/

Git Flow

ここまでの話を全部まとめると、右の図になる。

前述の通り、大体のチームでは Git Flowをそのままではなく、



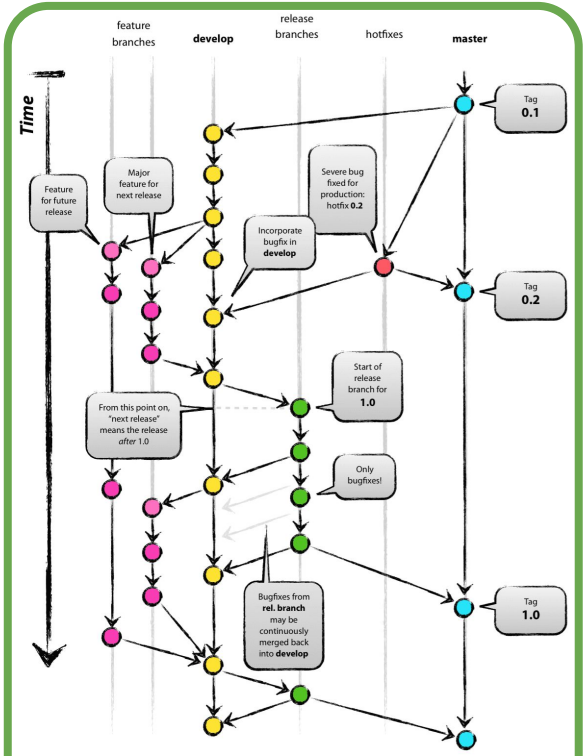
https://nvie.com/posts/a-successful-git-branching-model/

Git Flow

ここまでの話を全部まとめると、右の図になる。

前述の通り、大体のチームでは Git Flowをそのままではなく、

ちょっとアレンジして使っていると思う。



https://nvie.com/posts/a-successful-git-branching-model/

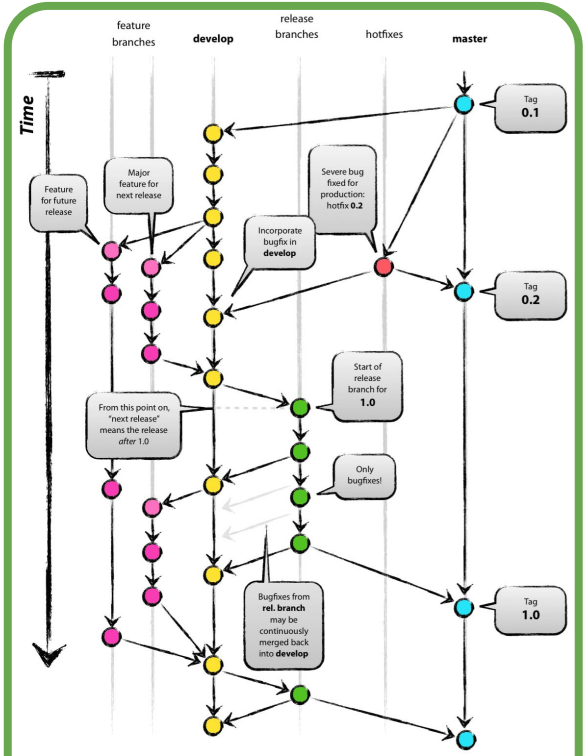
Git Flow

ここまでの話を全部まとめると、右の図になる。

前述の通り、大体のチームでは Git Flowをそのままではなく、

ちょっとアレンジして使っていると思う。

細かいところは配属されてからのお楽しみ。



https://nvie.com/posts/a-successful-git-branching-model/

Git Flow

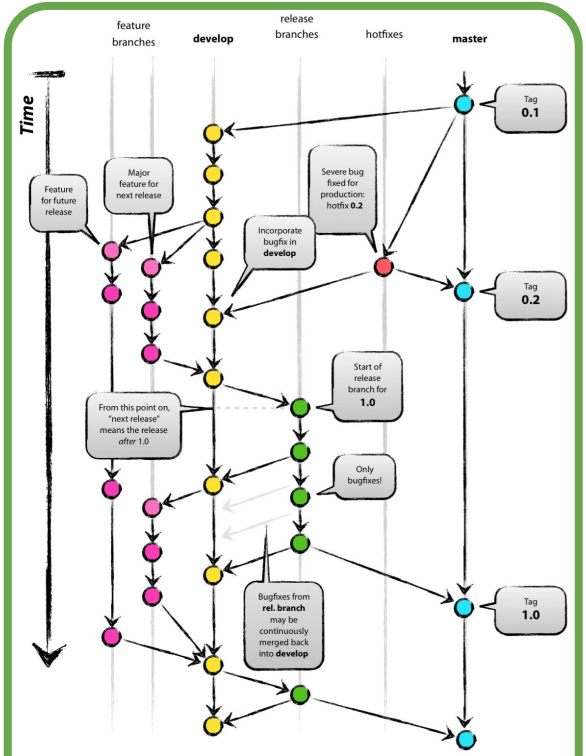
ここまでの話を全部まとめると、右の図になる。

前述の通り、大体のチームでは Git Flowをそのままではなく、

ちょっとアレンジして使っていると思う。

細かいところは配属されてからのお楽しみ。

原典→ [A successful Git branching model]



https://nvie.com/posts/a-successful-git-branching-model/

Git Flow

ここまでの話を全部まとめると、右の図になる。

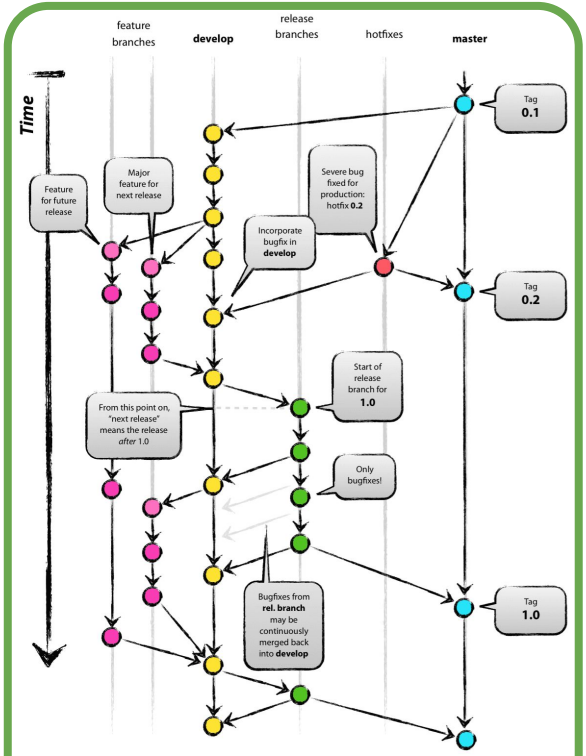
前述の通り、大体のチームでは Git Flowをそのままではなく、

ちょっとアレンジして使っていると思う。

細かいところは配属されてからのお楽しみ。

原典→ [A successful Git branching model]

英語だけど、難しいことは書いていないのでぜひ読んでみてね。



https://nvie.com/posts/a-successful-git-branching-model/

GitHub について

GitHub について

Git 単体で使うのではなく、リポジトリホスティングサービスと併用することで Git は真価を発揮する。

GitHub について

Git 単体で使うのではなく、リポジトリホスティングサービスと併用することで Git は真価を発揮する。

弊社では GitHub を使っているので、GitHub の機能について話します。

GitHub について

Git 単体で使うのではなく、リポジトリホスティングサービスと併用することで Git は真価を発揮する。

弊社では GitHub を使っているので、GitHub の機能について話します。

GitHub は Git をチームで使うにあたって便利な機能がいっぱい入っている

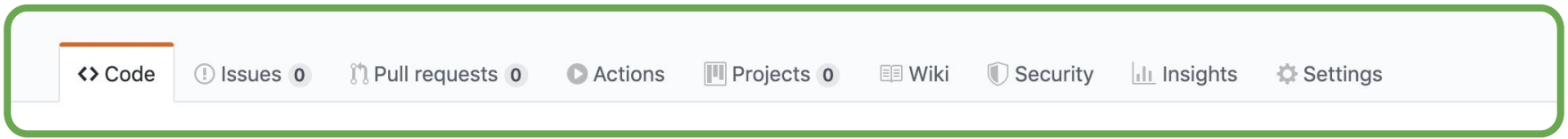
GitHub について

Git 単体で使うのではなく、リポジトリホスティングサービスと併用することで Git は真価を発揮する。

弊社では GitHub を使っているので、GitHub の機能について話します。

GitHub は Git をチームで使うにあたって便利な機能がいっぱい入っている

基本的な機能はリポジトリのトップページのタブでまとめられている。



GitHub について

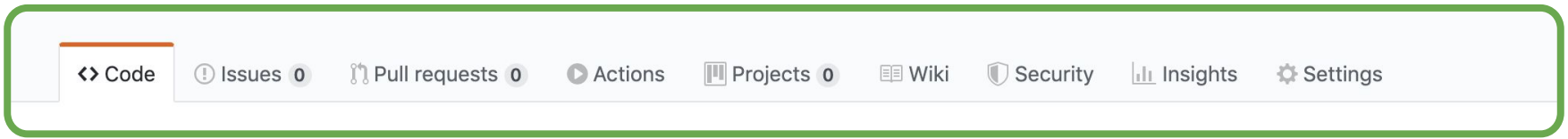
Git 単体で使うのではなく、リポジトリホスティングサービスと併用することで Git は真価を発揮する。

弊社では GitHub を使っているので、GitHub の機能について話します。

GitHub は Git をチームで使うにあたって便利な機能がいっぱい入っている

基本的な機能はリポジトリのトップページのタブでまとめられている。

でも意外と全部のタブ開いたことない人もいるんじゃないかな。



GitHub について

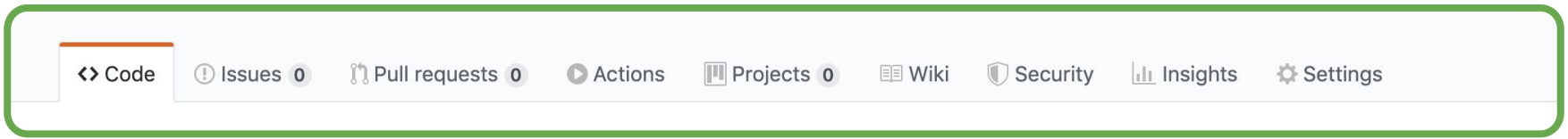
Git 単体で使うのではなく、リポジトリホスティングサービスと併用することで Git は真価を発揮する。

弊社では GitHub を使っているので、GitHub の機能について話します。

GitHub は Git をチームで使うにあたって便利な機能がいっぱい入っている

基本的な機能はリポジトリのトップページのタブでまとめられている。

でも意外と全部のタブ開いたことない人もいるんじゃないかな。



まずはIssuesの機能から解説していく。

Issue について

気になったことをなんでもMarkdownで書いておくところ。

Issue について

気になったことをなんでもMarkdownで書いておくところ。

（実際の運用はチームによるけど大体のところは何書いても良いと思う）

Issue について

気になったことをなんでもMarkdownで書いておくところ。

（実際の運用はチームによるけど大体のところは何書いても良いと思う）

ここに実タスクを並べて、後述の**Projectsで管理**するチームもあれば、

Issue について

気になったことをなんでもMarkdownで書いておくところ。

(実際の運用はチームによるけど大体のところは何書いても良いと思う)

ここに実タスクを並べて、後述の**Projectsで管理**するチームもあれば、

「テストが遅いから改善したい」とか「あの負債をどうやって解消しよう」とかの

相談事を書いて**議論の土台にする**チームもある。

Issue について

気になったことをなんでもMarkdownで書いておくところ。

(実際の運用はチームによるけど大体のところは何書いても良いと思う)

ここに実タスクを並べて、後述の**Projectsで管理**するチームもあれば、

「テストが遅いから改善したい」とか「あの負債をどうやって解消しよう」とかの

相談事を書いて**議論の土台にする**チームもある。

ラベルをつけて、**種類別に整理**しやすくしたり、

Issue について

気になったことをなんでもMarkdownで書いておくところ。

(実際の運用はチームによるけど大体のところは何書いても良いと思う)

ここに実タスクを並べて、後述の**Projectsで管理**するチームもあれば、

「テストが遅いから改善したい」とか「あの負債をどうやって解消しよう」とかの

相談事を書いて**議論の土台にする**チームもある。

ラベルをつけて、**種類別に整理**しやすくしたり、

マイルストーンをつけて、**いつまでに対応するべきなのかを明確化**したり、

Issue について

気になったことをなんでもMarkdownで書いておくところ。

(実際の運用はチームによるけど大体のところは何書いても良いと思う)

ここに実タスクを並べて、後述の**Projectsで管理**するチームもあれば、

「テストが遅いから改善したい」とか「あの負債をどうやって解消しよう」とかの

相談事を書いて**議論の土台にする**チームもある。

ラベルをつけて、**種類別に整理**しやすくしたり、

マイルストーンをつけて、**いつまでに対応するべきなのかを明確化**したり、

特定のissueに**誰かをassign**して対応を任せたり。

Issue について

気になったことをなんでもMarkdownで書いておくところ。

(実際の運用はチームによるけど大体のところは何書いても良いと思う)

ここに実タスクを並べて、後述の**Projectsで管理**するチームもあれば、

「テストが遅いから改善したい」とか「あの負債をどうやって解消しよう」とかの

相談事を書いて**議論の土台にする**チームもある。

ラベルをつけて、**種類別に整理**しやすくしたり、

マイルストーンをつけて、**いつまでに対応するべきなのかを明確化**したり、

特定のissueに**誰かをassign**して対応を任せたり。

まあいっぱいできる。

Pull Request について

GitHub 関連の機能で一番重要。

Pull Request について

GitHub 関連の機能で一番重要。

特定の branch や、fork 元のリポジトリに、「私がやった作業を取り込んでくれ」とお願いを出せる機能。

Pull Request について

GitHub 関連の機能で一番重要。

特定の branch や、fork 元のリポジトリに、「私がやった作業を取り込んでくれ」とお願いを出せる機能。

→ 元々は fork 元のリポジトリにお願いする機能なので“Merge Request” ではなく “Pull Request”

Pull Request について

GitHub 関連の機能で一番重要。

特定の branch や、fork 元のリポジトリに、「私がやった作業を取り込んでくれ」とお願いを出せる機能。

→ 元々は fork 元のリポジトリにお願いする機能なので“Merge Request” ではなく “Pull Request”

お願いされた側は取り込む前に差分をよく吟味し、大丈夫そうなら merge して差分を取り込む。

Pull Request について

GitHub 関連の機能で一番重要。

特定の branch や、fork 元のリポジトリに、「私がやった作業を取り込んでくれ」とお願いを出せる機能。

→ 元々は fork 元のリポジトリにお願いする機能なので“Merge Request” ではなく “Pull Request”

お願いされた側は取り込む前に差分をよく吟味し、大丈夫そうなら merge して差分を取り込む。

ダメそうならダメなポイントを指摘してあげる。

Pull Request について

GitHub 関連の機能で一番重要。

特定の branch や、fork 元のリポジトリに、「私がやった作業を取り込んでくれ」とお願いを出せる機能。

→ 元々は fork 元のリポジトリにお願いする機能なので“Merge Request” ではなく “Pull Request”

お願いされた側は取り込む前に差分をよく吟味し、大丈夫そうなら merge して差分を取り込む。

ダメそうならダメなポイントを指摘してあげる。

↑いわゆるコードレビュー

PR を出す側の注意点

PR を出す側の注意点

- できるだけ細かい単位で PR を出すよう心がける

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく
        - その結果コンフリクトする

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく
        - その結果コンフリクトする
          - さらにコンフリクトの解消に失敗してバグるかも

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく
        - その結果コンフリクトする
          - さらにコンフリクトの解消に失敗してバグるかも
- **巨大にならざるをえないときは、途中でレビューしてもらう**

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく
        - その結果コンフリクトする
          - さらにコンフリクトの解消に失敗してバグるかも
- **巨大にならざるをえないときは、途中でレビューしてもらう**
  - Git Flow で少し話をした、feature から feature を切るテク

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく
        - その結果コンフリクトする
          - さらにコンフリクトの解消に失敗してバグるかも
- **巨大にならざるをえないときは、途中でレビューしてもらう**
  - Git Flow で少し話をした、feature から feature を切るテク
  - 最初の feature には WIP をタイトルに付けたり、draft PR として出したりすると ○

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく
        - その結果コンフリクトする
          - さらにコンフリクトの解消に失敗してバグるかも
- **巨大にならざるをえないときは、途中でレビューしてもらう**
  - Git Flow で少し話をした、feature から feature を切るテク
  - 最初の feature には WIP をタイトルに付けたり、draft PR として出したりすると ○
- **コミットログは綺麗にしておこう**

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく
        - その結果コンフリクトする
          - さらにコンフリクトの解消に失敗してバグるかも
- **巨大にならざるをえないときは、途中でレビューしてもらう**
  - Git Flow で少し話をした、feature から feature を切るテク
  - 最初の feature には WIP をタイトルに付けたり、draft PR として出したりすると ○
- **コミットログは綺麗にしておこう**
  - こまめにコミットし、できれば最後に `git rebase -i` で体裁を整えよう

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく
        - その結果コンフリクトする
          - さらにコンフリクトの解消に失敗してバグるかも
- **巨大にならざるをえないときは、途中でレビューしてもらう**
  - Git Flow で少し話をした、feature から feature を切るテク
  - 最初の feature には WIP をタイトルに付けたり、draft PR として出したりすると ○
- **コミットログは綺麗にしておこう**
  - こまめにコミットし、できれば最後に `git rebase -i` で体裁を整えよう
  - 厳しめの OSS だと fast-forward merge できないだけでもダメって言われたりする

PR を出す側の注意点

- **できるだけ細かい単位で PR を出すよう心がける**
  - 巨大 PR はレビューに時間がかかる
    - レビューに時間がかかると merge までの時間も当然伸びる
      - 別の PR がどんどん先に merge されていく
        - その結果コンフリクトする
          - さらにコンフリクトの解消に失敗してバグるかも
- **巨大にならざるをえないときは、途中でレビューしてもらう**
  - Git Flow で少し話をした、feature から feature を切るテク
  - 最初の feature には WIP をタイトルに付けたり、draft PR として出したりすると ○
- **コミットログは綺麗にしておこう**
  - こまめにコミットし、できれば最後に `git rebase -i` で体裁を整えよう
  - 厳しめの OSS だと fast-forward merge できないだけでもダメって言われたりする
    - fast-forward merge できるにこしたことはないので、意識できるなら意識した方がいい

PR を見る側の注意点

PR を見る側の注意点

- 機械的にチェックできる部分は CI に任せる

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**
  - 仕様の穴、セキュリティ/法的にヤバそうなところ

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**
  - 仕様の穴、セキュリティ/法的にヤバそうなところ
  - 負債になりそうなところ、設計的にまずいところ、計算量・レイテンシ的に厳しいところ

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**
  - 仕様の穴、セキュリティ/法的にヤバそうなところ
  - 負債になりそうなところ、設計的にまずいところ、計算量・レイテンシ的に厳しいところ
  - 意外と実装中に気づかなかった問題点が見つかることも多いので、自分のPRもレビューしておくと吉

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**
  - 仕様の穴、セキュリティ/法的にヤバそうなところ
  - 負債になりそうなところ、設計的にまずいところ、計算量・レイテンシ的に厳しいところ
  - 意外と実装中に気づかなかった問題点が見つかることも多いので、自分のPRもレビューしておくと吉
  - なかなか人間の目ではバグは見つかりません！高品質なコードは高品質なテストから

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**
  - 仕様の穴、セキュリティ/法的にヤバそうなところ
  - 負債になりそうなところ、設計的にまずいところ、計算量・レイテンシ的に厳しいところ
  - 意外と実装中に気づかなかった問題点が見つかることも多いので、自分のPRもレビューしておくと吉
  - なかなか人間の目ではバグは見つかりません！高品質なコードは高品質なテストから
- **実装者のことを思いやる**

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**
  - 仕様の穴、セキュリティ/法的にヤバそうなところ
  - 負債になりそうなところ、設計的にまずいところ、計算量・レイテンシ的に厳しいところ
  - 意外と実装中に気づかなかった問題点が見つかることも多いので、自分のPRもレビューしておくと吉
  - なかなか人間の目ではバグは見つかりません！高品質なコードは高品質なテストから
- **実装者のことを思いやる**
  - 「このコードは良くない」のような抽象的な言い方はやめて、どこがどういう理由で良くないのか書く

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**
  - 仕様の穴、セキュリティ/法的にヤバそうなところ
  - 負債になりそうなところ、設計的にまずいところ、計算量・レイテンシ的に厳しいところ
  - 意外と実装中に気づかなかった問題点が見つかることも多いので、自分のPRもレビューしておくと吉
  - なかなか人間の目ではバグは見つかりません！高品質なコードは高品質なテストから
- **実装者のことを思いやる**
  - 「このコードは良くない」のような抽象的な言い方はやめて、どこがどういう理由で良くないのか書く
    - 合わせて、どういう方法なら良いのかも書けると議論もしやすい

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**
  - 仕様の穴、セキュリティ/法的にヤバそうなところ
  - 負債になりそうなところ、設計的にまずいところ、計算量・レイテンシ的に厳しいところ
  - 意外と実装中に気づかなかった問題点が見つかることも多いので、自分のPRもレビューしておくと吉
  - なかなか人間の目ではバグは見つかりません！高品質なコードは高品質なテストから
- **実装者のことを思いやる**
  - 「このコードは良くない」のような抽象的な言い方はやめて、どこがどういう理由で良くないのか書く
    - 合わせて、どういう方法なら良いのかも書けると議論もしやすい
  - テキストでの指摘は、結構キツく感じてしまいがちなので、普段の1.5倍やさしい言葉使いをしよう

PR を見る側の注意点

- **機械的にチェックできる部分は CI に任せる**
  - テストはしっかり書こうね
  - コードフォーマッタを使うと、細かいスペースの差分とかが生まれなくて便利
- **人間が見るべきポイントにしっかり集中する**
  - 仕様の穴、セキュリティ/法的にヤバそうなところ
  - 負債になりそうなところ、設計的にまずいところ、計算量・レイテンシ的に厳しいところ
  - 意外と実装中に気づかなかった問題点が見つかることも多いので、自分のPRもレビューしておくと吉
  - なかなか人間の目ではバグは見つかりません！高品質なコードは高品質なテストから
- **実装者のことを思いやる**
  - 「このコードは良くない」のような抽象的な言い方はやめて、どこがどういう理由で良くないのか書く
    - 合わせて、どういう方法なら良いのかも書けると議論もしやすい
  - テキストでの指摘は、結構キツく感じてしまいがちなので、普段の1.5倍やさしい言葉使いをしよう
    - HRT(謙虚・尊敬・信頼)の原則

Actions について

GitHub が提供している CI/CD 環境。

Actions について

GitHub が提供している CI/CD 環境。

他のサービスと連携させる必要がないので、簡単に使える + パブリックリポジトリでは無料で使える点がメリット。

Actions について

GitHub が提供している CI/CD 環境。

他のサービスと連携させる必要がないので、簡単に使える + パブリックリポジトリでは無料で使える点がメリット。

また、汎用的な部分は GitHub で公開されている Action を利用することができることもありがたい。

Actions について

GitHub が提供している CI/CD 環境。

他のサービスと連携させる必要がないので、簡単に使える + パブリックリポジトリでは無料で使える点がメリット。

また、汎用的な部分は GitHub で公開されている Action を利用することができることもありがたい。

hook したいイベントとワークフローを書いた yaml を.github/workflows/ に配置すると、勝手に走ってくれる。

Actions について

GitHub が提供している CI/CD 環境。

他のサービスと連携させる必要がないので、簡単に使える + パブリックリポジトリでは無料で使える点がメリット。

また、汎用的な部分は GitHub で公開されている Action を利用することができることもありがたい。

hook したいイベントとワークフローを書いた yaml を.github/workflows/ に配置すると、勝手に走ってくれる。

→ 具体的な書き方は 公式ドキュメント を読んでください。

Actions について

GitHub が提供している CI/CD 環境。

他のサービスと連携させる必要がないので、簡単に使える + パブリックリポジトリでは無料で使える点がメリット。

また、汎用的な部分は GitHub で公開されている Action を利用することができることもありがたい。

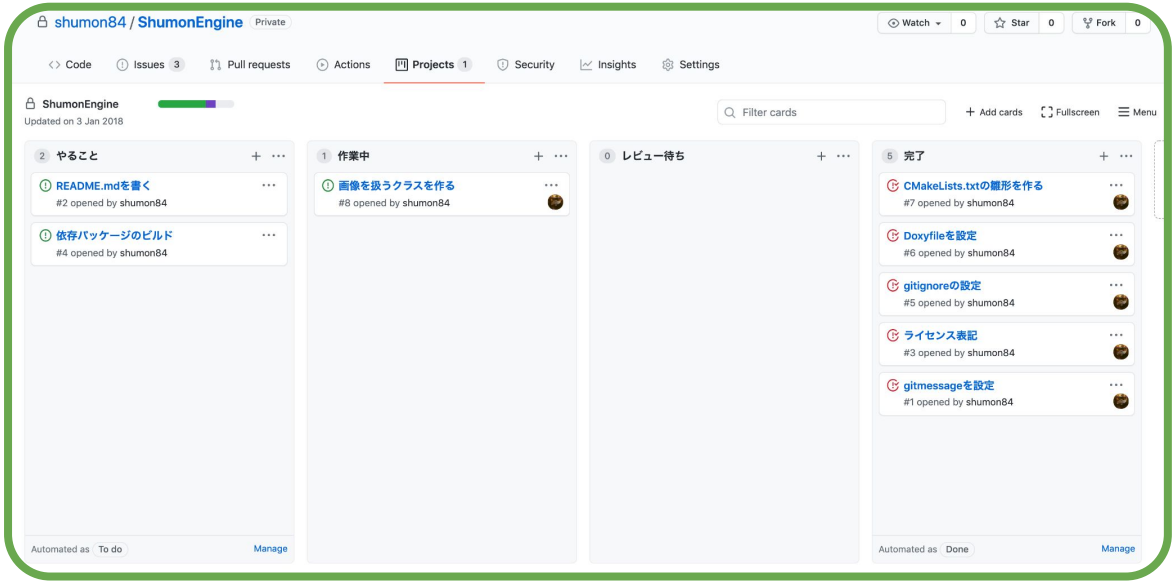
hook したいイベントとワークフローを書いた yaml を.github/workflows/ に配置すると、勝手に走ってくれる。

→ 具体的な書き方は 公式ドキュメント を読んでください。

ただし比較的新しめのサービスなので、まだまだ機能面では発展途上。

Projects について

カンバンスタイルのタスク管理ツール。

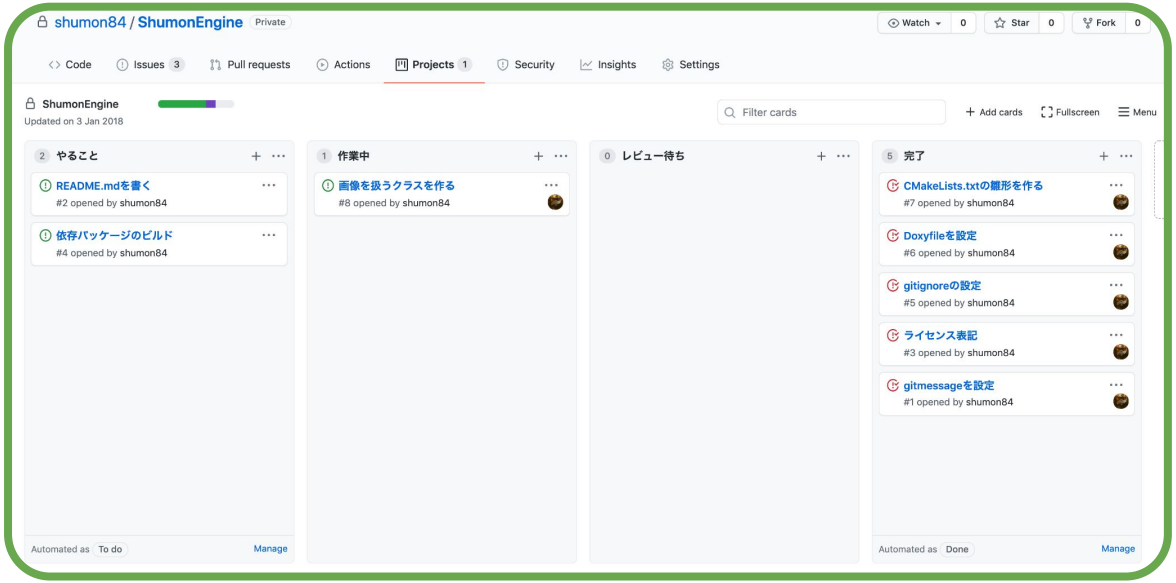


Projects について

カンバンスタイルのタスク管理ツール。

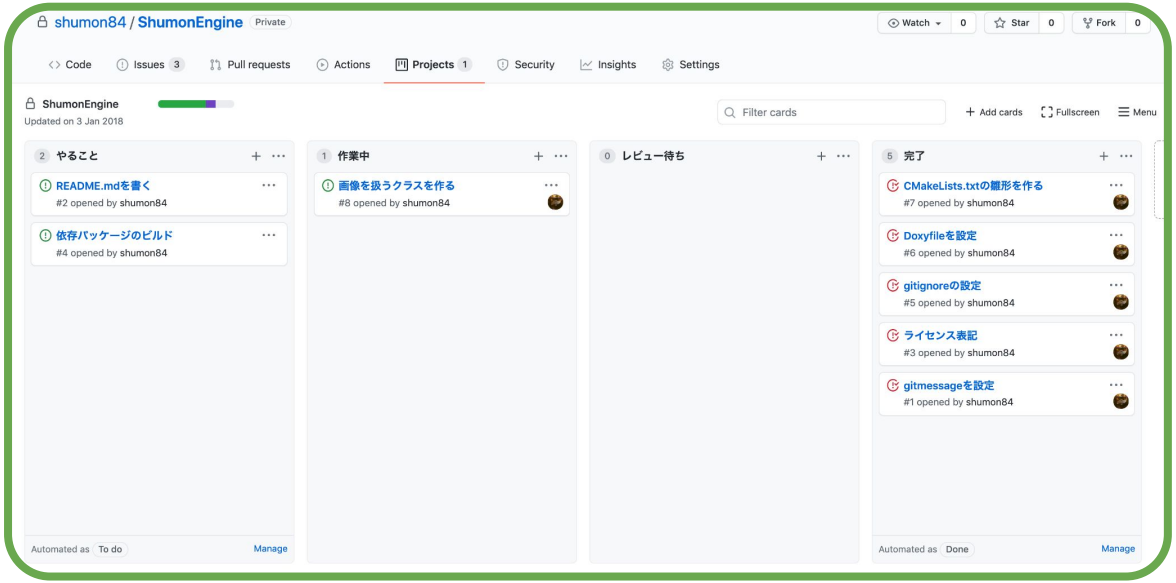
issue や PR を付箋のようにタブを移動

させて進捗状況を可視化できる。



Projects について

カンバンスタイルのタスク管理ツール。



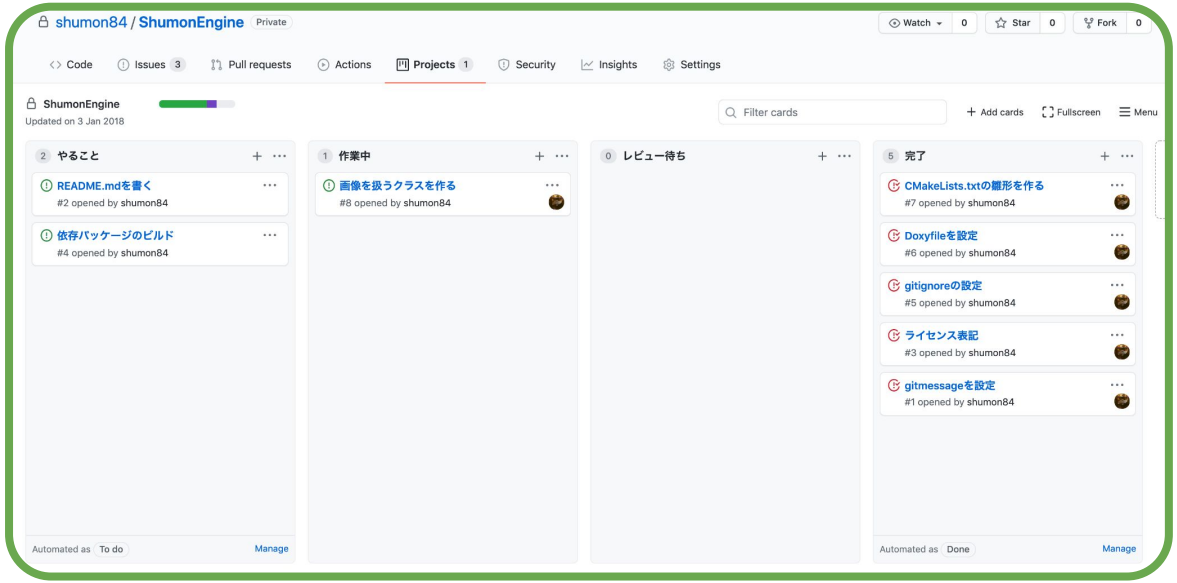
issue や PR を付箋のようにタブを移動

させて進捗状況を可視化できる。

これは、簡単なタブしか作っていないけど、実際のカンバンはもっと奥が深い(ちゃんとした人に見せたら怒られる)

Projects について

カンバンスタイルのタスク管理ツール。



issue や PR を付箋のようにタブを移動

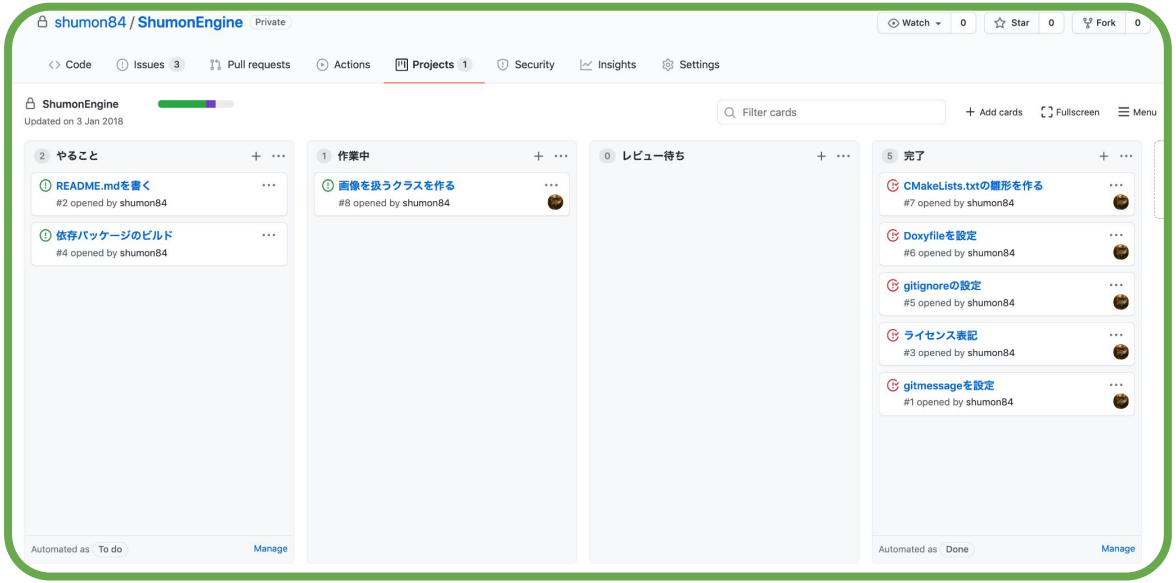
させて進捗状況を可視化できる。

これは、簡単なタブしか作っていないけど、実際のカンバンはもっと奥が深い(ちゃんとした人に見せたら怒られる)

アジャイルとか、スクラムとかと関係の深い話なので、興味がある人は調べてみてください。

Projects について

カンバンスタイルのタスク管理ツール。



issue や PR を付箋のようにタブを移動

させて進捗状況を可視化できる。

これは、簡単なタブしか作っていないけど、実際のカンバンはもっと奥が深い(ちゃんとした人に見せたら怒られる)

アジャイルとか、スクラムとかと関係の深い話なので、興味がある人は調べてみてください。

→ 個人的には カイゼン・ジャーニー と SCRUM MASTER THE BOOK が読みやすくてオススメ。

Wiki について

いわゆる Wiki を究極にシンプルにしたやつ。

Wiki について

いわゆる Wiki を究極にシンプルにしたやつ。

基本的にチームメンバーが自由に記事を追加・編集・削除できるだけの機能。

Wiki について

いわゆる Wiki を究極にシンプルにしたやつ。

基本的にチームメンバーが自由に記事を追加・編集・削除できるだけの機能。

個人的にドキュメント類はバージョン管理したいので、Git に乗せるべきで、

ちょっとした FAQ や、チーム文化のような常に流動的なことがらを書くべき場所と思ってる。

Wiki について

いわゆる Wiki を究極にシンプルにしたやつ。

基本的にチームメンバーが自由に記事を追加・編集・削除できるだけの機能。

個人的にドキュメント類はバージョン管理したいので、Git に乗せるべきで、

ちょっとした FAQ や、チーム文化のような常に流動的なことがらを書くべき場所と思ってる。

この辺は色々流派があると思うので、正解はないです。

Wiki について

いわゆる Wiki を究極にシンプルにしたやつ。

基本的にチームメンバーが自由に記事を追加・編集・削除できるだけの機能。

個人的にドキュメント類はバージョン管理したいので、Git に乗せるべきで、

ちょっとした FAQ や、チーム文化のような常に流動的なことがらを書くべき場所と思ってる。

この辺は色々流派があると思うので、正解はないです。

機能がかなり貧弱なので、ちゃんと運用してるチームは多分あんまりないんじゃないかなと思う。

Wiki について

いわゆる Wiki を究極にシンプルにしたやつ。

基本的にチームメンバーが自由に記事を追加・編集・削除できるだけの機能。

個人的にドキュメント類はバージョン管理したいので、Git に乗せるべきで、

ちょっとした FAQ や、チーム文化のような常に流動的なことがらを書くべき場所と思ってる。

この辺は色々流派があると思うので、正解はないです。

機能がかなり貧弱なので、ちゃんと運用してるチームは多分あんまりないんじゃないかなと思う。

せっかく docbase 契約してるし、docbase で良いのでは……?

Security について

主に↓の 3 つの機能についてのあれこれをするタブ。

- Security Policy
- Security Advisories
- Dependabot

Security について

主に↓の 3 つの機能についてのあれこれをするタブ。

- Security Policy
- Security Advisories
- Dependabot

Security Policy は脆弱性報告の手順を Markdown で書いておく。

Security について

主に↓の 3 つの機能についてのあれこれをするタブ。

- Security Policy
- Security Advisories
- Dependabot

Security Policy は脆弱性報告の手順を Markdown で書いておく。

Security Advisories は非公開の issue のようなもので、脆弱性が対策されるまでは秘匿した状態で議論し、

対応パッチがリリースされると、その議論を公開できる。

Security について

主に↓の 3 つの機能についてのあれこれをするタブ。

- Security Policy
- Security Advisories
- Dependabot

Security Policy は脆弱性報告の手順を Markdown で書いておく。

Security Advisories は非公開の issue のようなもので、脆弱性が対策されるまでは秘匿した状態で議論し、

対応パッチがリリースされると、その議論を公開できる。

Dependabot はコードから依存パッケージのバージョンを解析して、脆弱性があればアラートを飛ばしてくれる。

Security について

主に↓の 3 つの機能についてのあれこれをするタブ。

- Security Policy
- Security Advisories
- Dependabot

Security Policy は脆弱性報告の手順を Markdown で書いておく。

Security Advisories は非公開の issue のようなもので、脆弱性が対策されるまでは秘匿した状態で議論し、

対応パッチがリリースされると、その議論を公開できる。

Dependabot はコードから依存パッケージのバージョンを解析して、脆弱性があればアラートを飛ばしてくれる。

→ Dependabot 以外はパブリックリポジトリでしか使い道がないので、基本的にはOSSのための機能。

Insights について

リポジトリの活発度を色々な指標で確認できる場所。

Insights について

リポジトリの活発度を色々な指標で確認できる場所。

時系列でコミット数や差分の量がグラフ化されていたり、リポジトリの色々な統計情報が見られる。

Insights について

リポジトリの活発度を色々な指標で確認できる場所。

時系列でコミット数や差分の量がグラフ化されていたり、リポジトリの色々な統計情報が見られる。

自分のリポジトリでこのタブを使うことはあんまりない。

Insights について

リポジトリの活発度を色々な指標で確認できる場所。

時系列でコミット数や差分の量がグラフ化されていたり、リポジトリの色々な統計情報が見られる。

自分のリポジトリでこのタブを使うことはあんまりない。

新しいライブラリやツールの導入を検討しているとき、それが GitHub でホストされているならこのタブを見ると、

どれくらいメンテされているのかが一目瞭然なので、ぜひ使ってみてね。

Settings について

リポジトリの設定を変えるタブ(それはそう)

Settings について

リポジトリの設定を変えるタブ(それはそう)

各種機能の on/off や、デフォルトブランチの変更、PRのマージに関するルールの設定など色々できる。

Settings について

リポジトリの設定を変えるタブ(それはそう)

各種機能の on/off や、デフォルトブランチの変更、PRのマージに関するルールの設定など色々できる。

おそらく開発中に最もよく使うのは**Branch protection rules**。

Settings について

リポジトリの設定を変えるタブ(それはそう)

各種機能の on/off や、デフォルトブランチの変更、PRのマージに関するルールの設定など色々できる。

おそらく開発中に最もよく使うのは**Branch protection rules**。

指定した正規表現にマッチした branch に対して、PR を経ない直接の push や force push を禁止したり...etc

Settings について

リポジトリの設定を変えるタブ(それはそう)

各種機能の on/off や、デフォルトブランチの変更、PRのマージに関するルールの設定など色々できる。

おそらく開発中に最もよく使うのは**Branch protection rules**。

指定した正規表現にマッチした branch に対して、PR を経ない直接の push や force push を禁止したり...etc

master(main) と develop には、大体なんらかのルールが設定されていると思う。

Settings について

リポジトリの設定を変えるタブ(それはそう)

各種機能の on/off や、デフォルトブランチの変更、PRのマージに関するルールの設定など色々できる。

おそらく開発中に最もよく使うのは**Branch protection rules**。

指定した正規表現にマッチした branch に対して、PR を経ない直接の push や force push を禁止したり...etc

master(main) と develop には、大体なんらかのルールが設定されていると思う。

private → public の変更、オーナー権限の委譲、リポジトリの削除などの**危険な操作もあるので注意**。

Git の内部構造

Git の内部構造

ついにきました本日のメインです。

ここからの話が理解できれば Git の 5 割ぐらいは自作できます。

もう 5 割を作りたい人は個人的に聞きにきてください。

目指せ Git マスター。

Git の内部構造

Git をよく使っていても、実は全然 Git のことを理解できていない

- commit って親 commit との**差分を保存**してるんでしょ
- branch って**分かれた枝のこと**でしょ
- reset って commit を**なかったことにする**コマンドでしょ

Git の内部構造

Git をよく使っていても、実は全然 Git のことを理解できていない

- commit って親 commit との**差分を保存**してるんでしょ
- branch って**分かれた枝のこと**でしょ
- reset って commit を**なかったことにする**コマンドでしょ

↑は**全部間違い**

Git の内部構造

Git をよく使っていても、実は全然 Git のことを理解できていない

- commit って親 commit との**差分を保存**してるんでしょ
- branch って**分かれた枝のこと**でしょ
- reset って commit を**なかったことにする**コマンドでしょ

↑は**全部間違い**

自分も学生時代は勘違いしたまま使ってた

Git の内部構造

Git をよく使っていても、実は全然 Git のことを理解できていない

- commit って親 commit との**差分を保存**してるんでしょ
- branch って**分かれた枝のこと**でしょ
- reset って commit を**なかったことにする**コマンドでしょ

↑は**全部間違い**

自分も学生時代は勘違いしたまま使ってた

ここからの話を聞けば「**reset や rebase 叩くときは毎回検索してコピペしてる**」みたいな状況を脱せるはず

Git の内部構造

特に↓について詳しく処理を追っていく。

- コミットの仕組み
- checkout, reset の仕組み

コミットの仕組み

コミットの仕組み

Git を使っていると、大きく3つの段階を踏んで commit する

コミットの仕組み

Git を使っていると、大きく3つの段階を踏んで commit する

1. コードを編集する
2. 編集したコードを add する
3. commit する

コミットの仕組み

Git を使っていると、大きく3つの段階を踏んで commit する

1. コードを編集する
2. 編集したコードを add する
3. commit する

2 と 3 のときに、Git 内部で一体何が起こっているのかを追っていく

コミットの仕組み

こんなリポジトリを用意して実験してみる。

```
$ tree

.

└── README.md


0 directories, 1 file
```

ルートに README.md があるだけ。

コミットの仕組み

編集したコードを add すると、内部的には何が起こっているのか

コミットの仕組み

編集したコードを add すると、内部的には何が起こっているのか

→ 教科書的な回答は「コミットに含めたいファイルを index に登録している」

コミットの仕組み

編集したコードを add すると、内部的には何が起こっているのか

→ 教科書的な回答は「コミットに含めたいファイルを index に登録している」

具体的に index はどこにあるかというと、.git/index に保存されている。

コミットの仕組み

編集したコードを add すると、内部的には何が起こっているのか

→ 教科書的な回答は「コミットに含めたいファイルを index に登録している」

具体的に index はどこにあるかというと、.git/index に保存されている。

とりあえず cat してみる。

```
$ cat .git/index

DIRC`m��  \D�`m��   \D����_A�</�S]:7��U��W���V�9�    README.mdTREE1 0

Z��  ̀c𝆹𝅥c$-Y T1`^:qî r g p;TT
```

コミットの仕組み

編集したコードを add すると、内部的には何が起こっているのか

→ 教科書的な回答は「コミットに含めたいファイルを index に登録している」

具体的に index はどこにあるかというと、.git/index に保存されている。

とりあえず cat してみる。バイナリファイルで文字化けしてるけど、それっぽい文字列が見える。

```
$ cat .git/index

DIRC`m��  \D�`m��   \D����_A�</�S]:7��U��W���V�9�    README.mdTREE1 0

Z��  c$-Y1`^:qr  T
```

コミットの仕組み

index の中身を確認するコマンドがあるので、それで確認してみる。

コミットの仕組み

index の中身を確認するコマンドがあるので、それで確認してみる。

```
$ git ls-files --stage
```

コミットの仕組み

index の中身を確認するコマンドがあるので、それで確認してみる。

```
$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md
```

コミットの仕組み

index の中身を確認するコマンドがあるので、それで確認してみる。

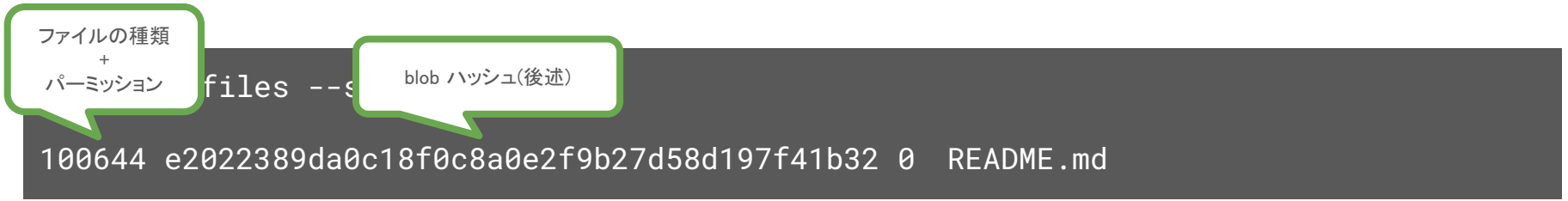
ファイルの種類
+
パーミッション

```
files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md
```

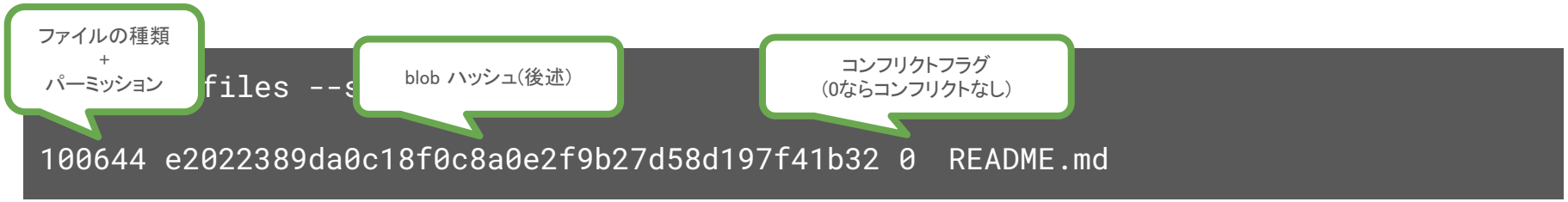
コミットの仕組み

index の中身を確認するコマンドがあるので、それで確認してみる。



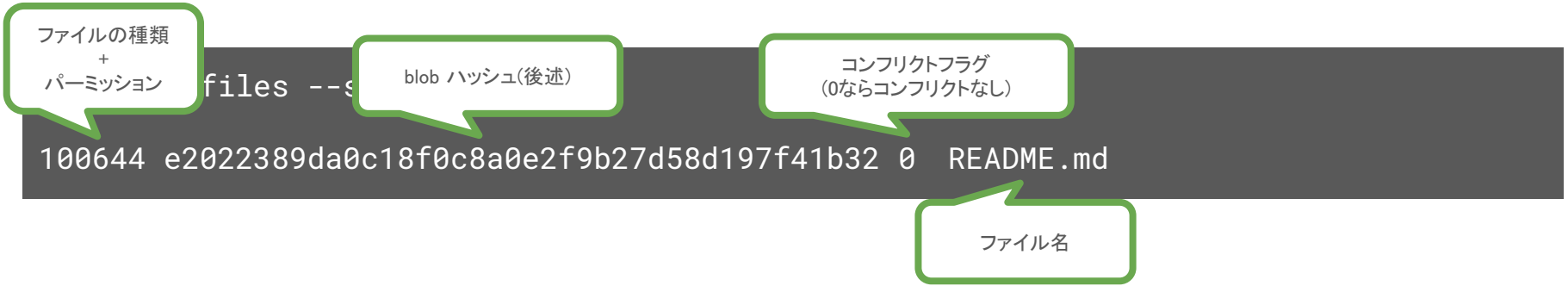
コミットの仕組み

index の中身を確認するコマンドがあるので、それで確認してみる。



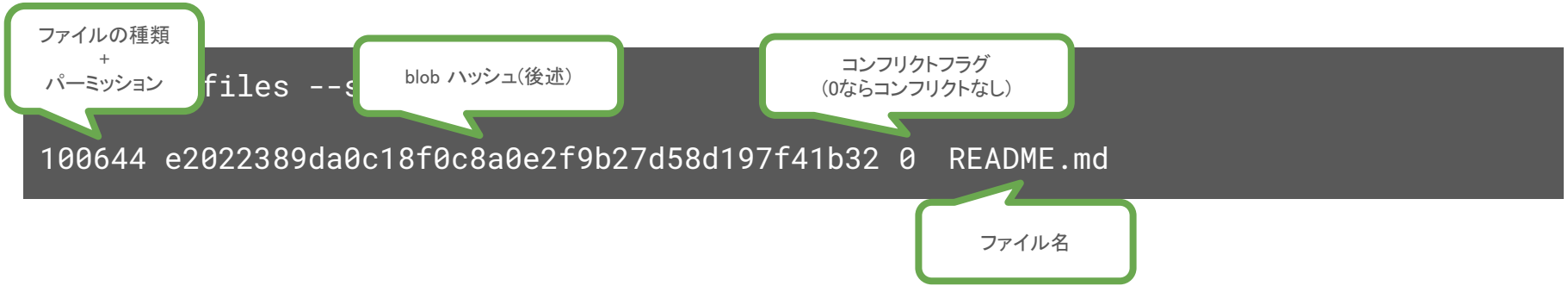
コミットの仕組み

index の中身を確認するコマンドがあるので、それで確認してみる。



コミットの仕組み

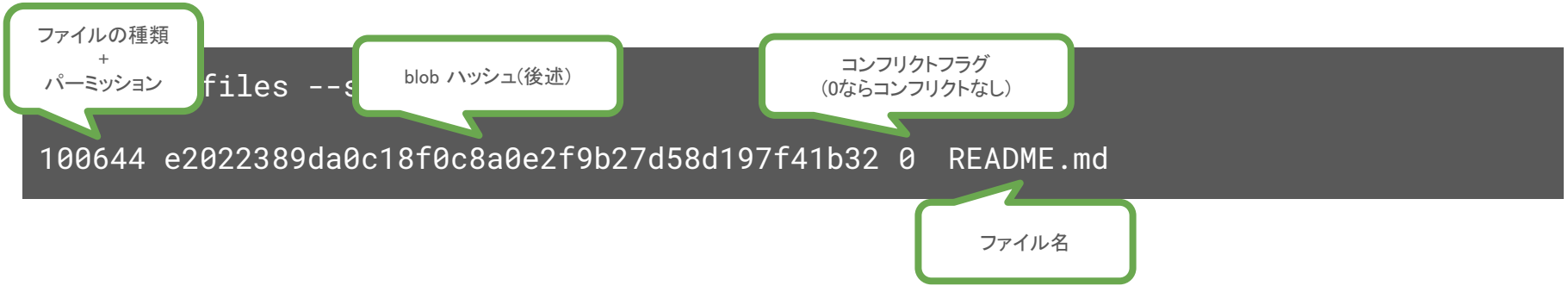
index の中身を確認するコマンドがあるので、それで確認してみる。



ファイル名と一緒に何やら色んな情報が出てくる。

コミットの仕組み

index の中身を確認するコマンドがあるので、それで確認してみる。

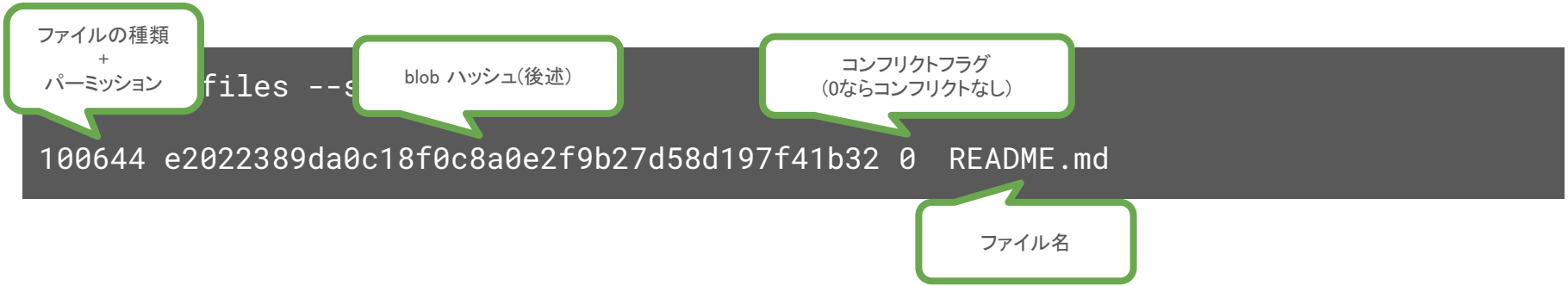


ファイル名と一緒に何やら色んな情報が出てくる。

ちなみに index は、ここに表示されていない情報も持っている。

コミットの仕組み

index の中身を確認するコマンドがあるので、それで確認してみる。



ファイル名と一緒に何やら色んな情報が出てくる。

ちなみに index は、ここに表示されていない情報も持っている。

気になる人は --debug を付けると index が持ってる全情報を確認できる(詳しくは index-format.txt 参照)

コミットの仕組み

blob ハッシュってなに？

コミットの仕組み

blob ハッシュってなに？→ Git に管理されている**オブジェクトの 1 種(の key)**のこと。

コミットの仕組み

blob ハッシュってなに？→ Git に管理されている**オブジェクトの 1 種(の key)**のこと。

Git には様々なデータを、「オブジェクト」と呼ばれる概念で表現している。

コミットの仕組み

blob ハッシュってなに？→ Git に管理されている**オブジェクトの 1 種(の key)**のこと。

Git には様々なデータを、「オブジェクト」と呼ばれる概念で表現している。

オブジェクトには以下の 4 種類がある。

- commit オブジェクト
- tree オブジェクト
- blob オブジェクト
- tag オブジェクト

コミットの仕組み

blob ハッシュってなに？→ Git に管理されている**オブジェクトの 1 種(の key)**のこと。

Git には様々なデータを、「オブジェクト」と呼ばれる概念で表現している。

オブジェクトには以下の 4 種類がある。

- commit オブジェクト
  - コミットの情報が入っているオブジェクト
- tree オブジェクト
- blob オブジェクト
- tag オブジェクト

コミットの仕組み

blob ハッシュってなに？→ Git に管理されている**オブジェクトの 1 種(の key)**のこと。

Git には様々なデータを、「オブジェクト」と呼ばれる概念で表現している。

オブジェクトには以下の 4 種類がある。

- commit オブジェクト
  - コミットの情報が入っているオブジェクト
- tree オブジェクト
  - ディレクトリの情報が入っているオブジェクト
- blob オブジェクト

- tag オブジェクト

コミットの仕組み

blob ハッシュってなに？→ Git に管理されている**オブジェクトの 1 種(の key)**のこと。

Git には様々なデータを、「オブジェクト」と呼ばれる概念で表現している。

オブジェクトには以下の 4 種類がある。

- commit オブジェクト
  - コミットの情報が入っているオブジェクト
- tree オブジェクト
  - ディレクトリの情報が入っているオブジェクト
- blob オブジェクト
  - ファイルの情報が入っているオブジェクト
- tag オブジェクト

コミットの仕組み

blob ハッシュってなに？→ Git に管理されている**オブジェクトの 1 種(の key)**のこと。

Git には様々なデータを、「オブジェクト」と呼ばれる概念で表現している。

オブジェクトには以下の 4 種類がある。

- commit オブジェクト
  - コミットの情報が入っているオブジェクト
- tree オブジェクト
  - ディレクトリの情報が入っているオブジェクト
- blob オブジェクト
  - ファイルの情報が入っているオブジェクト
- tag オブジェクト
  - annotated tag の情報が入っているオブジェクト

コミットの仕組み

blob ハッシュってなに？→ Git に管理されている**オブジェクトの 1 種(の key)**のこと。

Git には様々なデータを、「オブジェクト」と呼ばれる概念で表現している。

オブジェクトには以下の 4 種類がある。

- commit オブジェクト
  - コミットの情報が入っているオブジェクト
- tree オブジェクト
  - ディレクトリの情報が入っているオブジェクト
- blob オブジェクト
  - ファイルの情報が入っているオブジェクト
- tag オブジェクト
  - annotated tag の情報が入っているオブジェクト

ただしGit で扱うデータが全てオブジェクトなわけではない。例えば branch はオブジェクトで管理していない。

コミットの仕組み

オブジェクトの実体は .git/objects の中に zlib で圧縮して保存されている。

コミットの仕組み

オブジェクトの実体は .git/objects の中に zlib で圧縮して保存されている。

例えば、key が e2022389da0c18f0c8a0e2f9b27d58d197f41b32 のオブジェクトのパスは、

.git/objects/e2/022389da0c18f0c8a0e2f9b27d58d197f41b32になる。

コミットの仕組み

オブジェクトの実体は .git/objects の中に zlib で圧縮して保存されている。

例えば、key が e2022389da0c18f0c8a0e2f9b27d58d197f41b32 のオブジェクトのパスは、

.git/objects/**e2**/**022389da0c18f0c8a0e2f9b27d58d197f41b32**になる。

このように、Git は一種の **Key-Value Store** としてオブジェクトを管理している。

コミットの仕組み

README.md の blob (e2022389da0c18f0c8a0e2f9b27d58d197f41b32) の中身を確認してみる。

コミットの仕組み

README.md の blob (e2022389da0c18f0c8a0e2f9b27d58d197f41b32) の中身を確認してみる。

```
$ cat README.md

# 21 卒 Git 研修用リポジトリ

$ cat .git/objects/e2/022389da0c18f0c8a0e2f9b27d58d197f41b32 | zlib d

blob 38# 21 卒 Git 研修用リポジトリ
```

コミットの仕組み

README.md の blob (e2022389da0c18f0c8a0e2f9b27d58d197f41b32) の中身を確認してみる。

```
$ cat README.md

# 21 卒 Git 研修用リポジトリ

$ cat .git/objects/e2/022389da0c18f0c8a0e2f9b27d58d197f41b32 | zlib d

blob 38# 21 卒 Git 研修用リポジトリ
```

ファイルの先頭に、オブジェクトの種類とファイルサイズを付けたものを、zlib で圧縮している事が分かる。

コミットの仕組み

README.md の blob (e2022389da0c18f0c8a0e2f9b27d58d197f41b32) の中身を確認してみる。

```
$ cat README.md

# 21 卒 Git 研修用リポジトリ

$ cat .git/objects/e2/022389da0c18f0c8a0e2f9b27d58d197f41b32 | zlib d

blob 38# 21 卒 Git 研修用リポジトリ
```

ファイルの先頭に、オブジェクトの種類とファイルサイズを付けたものを、zlib で圧縮している事が分かる。

ちなみに、圧縮前の状態の SHA-1 が、この blob を指す key になっている。

```
$ cat .git/objects/e2/022389da0c18f0c8a0e2f9b27d58d197f41b32 | zlib d | shasum

e2022389da0c18f0c8a0e2f9b27d58d197f41b32  -
```

コミットの仕組み

README.md の blob (e2022389da0c18f0c8a0e2f9b27d58d197f41b32) の中身を確認してみる。

```
$ cat README.md

# 21 卒 Git 研修用リポジトリ

$ cat .git/objects/e2/022389da0c18f0c8a0e2f9b27d58d197f41b32 | zlib d

blob 38# 21 卒 Git 研修用リポジトリ
```

ファイルの先頭に、オブジェクトの種類とファイルサイズを付けたものを、zlib で圧縮している事が分かる。

ちなみに、圧縮前の状態の SHA-1 が、この blob を指す key になっている。

```
$ cat .git/objects/e2/022389da0c18f0c8a0e2f9b27d58d197f41b32 | zlib d | shasum

e2022389da0c18f0c8a0e2f9b27d58d197f41b32  -
```

コミットの仕組み

各オブジェクトの中身を確認するには、自分で zlib で伸長してもいいけど、git cat-file を使えば簡単にできる。

コミットの仕組み

各オブジェクトの中身を確認するには、自分で zlib で伸長してもいいけど、git cat-file を使えば簡単にできる。

```
$ git cat-file -p <オブジェクトハッシュ>
```

コミットの仕組み

各オブジェクトの中身を確認するには、自分で zlib で伸長してもいいけど、git cat-file を使えば簡単にできる。

```
$ git cat-file -p <オブジェクトハッシュ>
```

-p オプションを付けると、自動でオブジェクトの種類を判別して読みやすく整形して表示してくれる。

コミットの仕組み

各オブジェクトの中身を確認するには、自分で zlib で伸長してもいいけど、git cat-file を使えば簡単にできる。

$ git cat-file -p <オブジェクトハッシュ>

-p オプションを付けると、自動でオブジェクトの種類を判別して読みやすく整形して表示してくれる。

```
$ git cat-file -p e2022389da0c18f0c8a0e2f9b27d58d197f41b32

# 21 卒 Git 研修用リポジトリ
```

コミットの仕組み

本題に戻って、編集したコードを add すると何が起こるのか。

```
$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md
```

コミットの仕組み

本題に戻って、編集したコードを add すると何が起こるのか。

```
$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

$ echo hoge > hoge.txt
```

コミットの仕組み

本題に戻って、編集したコードを add すると何が起こるのか。

```
$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

$ echo hoge > hoge.txt

$ git add hoge.txt
```

コミットの仕組み

本題に戻って、編集したコードを add すると何が起こるのか。

```
$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

$ echo hoge > hoge.txt

$ git add hoge.txt

$ git ls-files --stage
```

コミットの仕組み

本題に戻って、編集したコードを add すると何が起こるのか。

```
$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

$ echo hoge > hoge.txt

$ git add hoge.txt

$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

100644 2262de0c121f22df8e78f5a37d6e114fd322c0b0 0  hoge.txt
```

コミットの仕組み

本題に戻って、編集したコードを add すると何が起こるのか。

```
$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

$ echo hoge > hoge.txt

$ git add hoge.txt

$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

100644 2262de0c121f22df8e78f5a37d6e114fd322c0b0 0  hoge.txt
```

コミットの仕組み

本題に戻って、編集したコードを add すると何が起こるのか。

```
$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

$ echo hoge > hoge.txt

$ git add hoge.txt

$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

100644 2262de0c121f22df8e78f5a37d6e114fd322c0b0 0  hoge.txt
```

index が更新されて、新しいエントリが追加されていることが分かる(このとき blob も一緒に生成される)

コミットの仕組み

add は基本的に index の更新 + blob オブジェクトの生成しかしていない。

```
$ mkdir fuga

$ echo fuga > fuga/fuga.txt
```

コミットの仕組み

add は基本的に index の更新 + blob オブジェクトの生成しかしていない。

```
$ mkdir fuga

$ echo fuga > fuga/fuga.txt

$ git add fuga/fuga.txt
```

コミットの仕組み

add は基本的に index の更新 + blob オブジェクトの生成しかしていない。

```
$ mkdir fuga

$ echo fuga > fuga/fuga.txt

$ git add fuga/fuga.txt

$ git ls-files --stage
```

コミットの仕組み

add は基本的に index の更新 + blob オブジェクトの生成しかしていない。

```
$ mkdir fuga

$ echo fuga > fuga/fuga.txt

$ git add fuga/fuga.txt

$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

100644 9128c3eb56a3547e2cca631042366bf0f37abe67 0  fuga/fuga.txt

100644 2262de0c121f22df8e78f5a37d6e114fd322c0b0 0  hoge.txt
```

コミットの仕組み

add は基本的に index の更新 + blob オブジェクトの生成しかしていない。

```
$ mkdir fuga

$ echo fuga > fuga/fuga.txt

$ git add fuga/fuga.txt

$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

100644 9128c3eb56a3547e2cca631042366bf0f37abe67 0  fuga/fuga.txt

100644 2262de0c121f22df8e78f5a37d6e114fd322c0b0 0  hoge.txt
```

コミットの仕組み

add は基本的に index の更新 + blob オブジェクトの生成しかしていない。

```
$ mkdir fuga

$ echo fuga > fuga/fuga.txt

$ git add fuga/fuga.txt

$ git ls-files --stage

100644 e2022389da0c18f0c8a0e2f9b27d58d197f41b32 0  README.md

100644 9128c3eb56a3547e2cca631042366bf0f37abe67 0  fuga/fuga.txt

100644 2262de0c121f22df8e78f5a37d6e114fd322c0b0 0  hoge.txt
```

ディレクトリが追加されても tree オブジェクトは作らず、blob オブジェクトしか作らない。

コミットの仕組み

blob オブジェクトは add 時に、tree オブジェクトは commit 時に生成する。

コミットの仕組み

blob オブジェクトは add 時に、tree オブジェクトは commit 時に生成する。

commit すると内部的に実行している処理は大まかには次の通り(本当はもうちょっと色々やってるけど割愛)

1. index から tree オブジェクトを生成
2. commit オブジェクトを生成
3. HEAD を新しい commit ハッシュに書き換え

コミットの仕組み

blob オブジェクトは add 時に、tree オブジェクトは commit 時に生成する。

commit すると内部的に実行している処理は大まかには次の通り(本当はもうちょっと色々やってるけど割愛)

1. index から tree オブジェクトを生成
2. commit オブジェクトを生成
3. HEAD を新しい commit ハッシュに書き換え

それぞれ細かく見ていく。

コミットの仕組み

commit オブジェクトを作る前に、index から tree オブジェクトを生成する。

コミットの仕組み

commit オブジェクトを作る前に、index から tree オブジェクトを生成する。

tree オブジェクトの構造はこんな感じ。

```
$ cat .git/objects/18/4135bd0b06f80372a29a35c32ba2e8a5609dc6 | zlib d | hexdump -C

00000000  74 72 65 65 20 33 37 00  31 30 30 36 34 34 20 52  |tree 37.100644 R|

00000010  45 41 44 4d 45 2e 6d 64  00 e2 02 23 89 da 0c 18  |EADME.md...#....|

00000020  f0 c8 a0 e2 f9 b2 7d 58  d1 97 f4 1b 32           |......}X....2|
```

コミットの仕組み

commit オブジェクトを作る前に、index から tree オブジェクトを生成する。

tree オブジェクトの構造はこんな感じ。

```
$ cat .git/objects/18/4135bd0b06f80372a29a35c32ba2e8a5609dc6 | zlib d | hexdump -C

00000000  74 72 65 65 20 33 37 00  31 30 30 36 34 34 20 52  |tree 37.100644 R|

00000010  45 41 44 4d 45 2e 6d 64  00 e2 02 23 89 da 0c 18  |EADME.md...#....|

00000020  f0 c8 a0 e2 f9 b2 7d 58  d1 97 f4 1b 32           |......}X....2|
```

コミットの仕組み

commit オブジェクトを作る前に、index から tree オブジェクトを生成する。

tree オブジェクトの構造はこんな感じ。

```
$ cat .git/objects/18/4135bd0b06f80372a29a35c32ba2e8a5609dc6 | zlib d | hexdump -C

00000000  74 72 65 65 20 33 37 00  31 30 30 36 34 34 20 52  |tree 37.100644 R|

00000010  45 41 44 4d 45 2e 6d 64  00 e2 02 23 89 da 0c 18  |EADME.md...#....|

00000020  f0 c8 a0 e2 f9 b2 7d 58  d1 97 f4 1b 32           |......}X....2|
```

コミットの仕組み

commit オブジェクトを作る前に、index から tree オブジェクトを生成する。

tree オブジェクトの構造はこんな感じ。

```
$ cat .git/objects/18/4135bd0b06f80372a29a35c32ba2e8a5609dc6 | zlib d | hexdump -C

00000000  74 72 65 65 20 33 37 00  31 30 30 36 34 34 20 52  |tree 37.100644 R|

00000010  45 41 44 4d 45 2e 6d 64  00 e2 02 23 89 da 0c 18  |EADME.md...#....|

00000020  f0 c8 a0 e2 f9 b2 7d 58  d1 97 f4 1b 32           |......}X....2|
```

コミットの仕組み

commit オブジェクトを作る前に、index から tree オブジェクトを生成する。

tree オブジェクトの構造はこんな感じ。

```
$ cat .git/objects/18/4135bd0b06f80372a29a35c32ba2e8a5609dc6 | zlib d | hexdump -C

00000000  74 72 65 65 20 33 37 00  31 30 30 36 34 34 20 52  |tree 37.100644 R|

00000010  45 41 44 4d 45 2e 6d 64  00 e2 02 23 89 da 0c 18  |EADME.md...#....|

00000020  f0 c8 a0 e2 f9 b2 7d 58  d1 97 f4 1b 32           |......}X....2|
```

コミットの仕組み

commit オブジェクトを作る前に、index から tree オブジェクトを生成する。

tree オブジェクトの構造はこんな感じ。

```
$ cat .git/objects/18/4135bd0b06f80372a29a35c32ba2e8a5609dc6 | zlib d | hexdump -C

00000000  74 72 65 65 20 33 37 00  31 30 30 36 34 34 20 52  |tree 37.100644 R|

00000010  45 41 44 4d 45 2e 6d 64  00 e2 02 23 89 da 0c 18  |EADME.md...#....|

00000020  f0 c8 a0 e2 f9 b2 7d 58  d1 97 f4 1b 32           |......}X....2|
```

ファイルの種類とパーミッションのフラグとファイル名、blob ハッシュが 0x00 区切りで書き込まれている。

コミットの仕組み

commit オブジェクトを作る前に、index から tree オブジェクトを生成する。

tree オブジェクトの構造はこんな感じ。

```
$ cat .git/objects/18/4135bd0b06f80372a29a35c32ba2e8a5609dc6 | zlib d | hexdump -C

00000000  74 72 65 65 20 33 37 00  31 30 30 36 34 34 20 52  |tree 37.100644 R|

00000010  45 41 44 4d 45 2e 6d 64  00 e2 02 23 89 da 0c 18  |EADME.md...#....|

00000020  f0 c8 a0 e2 f9 b2 7d 58  d1 97 f4 1b 32           |......}X....2|
```

ファイルの種類とパーミッションのフラグとファイル名、blob ハッシュが 0x00 区切りで書き込まれている。

→ 全部 index が持っていた情報

コミットの仕組み

commit オブジェクトを作る前に、index から tree オブジェクトを生成する。

tree オブジェクトの構造はこんな感じ。

```
$ cat .git/objects/18/4135bd0b06f80372a29a35c32ba2e8a5609dc6 | zlib d | hexdump -C

00000000  74 72 65 65 20 33 37 00  31 30 30 36 34 34 20 52  |tree 37.100644 R|

00000010  45 41 44 4d 45 2e 6d 64  00 e2 02 23 89 da 0c 18  |EADME.md...#....|

00000020  f0 c8 a0 e2 f9 b2 7d 58  d1 97 f4 1b 32           |......}X....2|
```

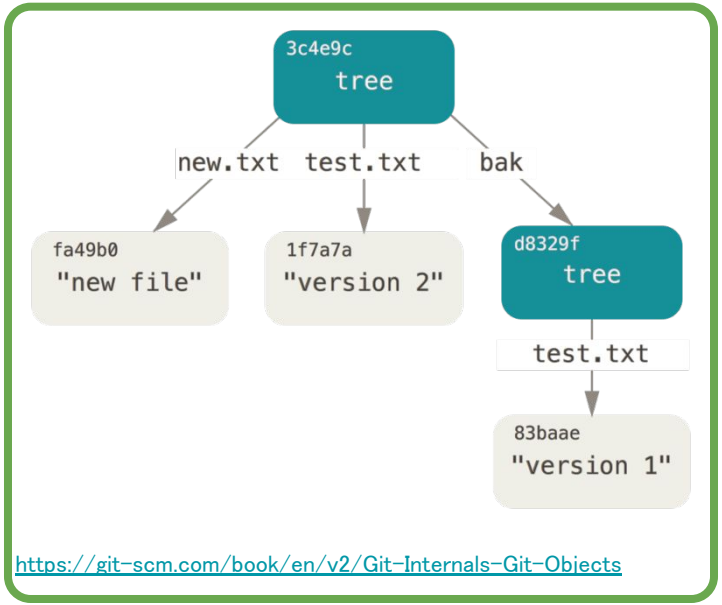
ファイルの種類とパーミッションのフラグとファイル名、blob ハッシュが 0x00 区切りで書き込まれている。

→ 全部 index が持っていた情報

commit すると、リポジトリのルートディレクトリを含む全ディレクトリ分の tree を自動で作る。

コミットの仕組み

tree オブジェクトは簡易的なファイルシステムのようになっていて、図のような構造を持っている。

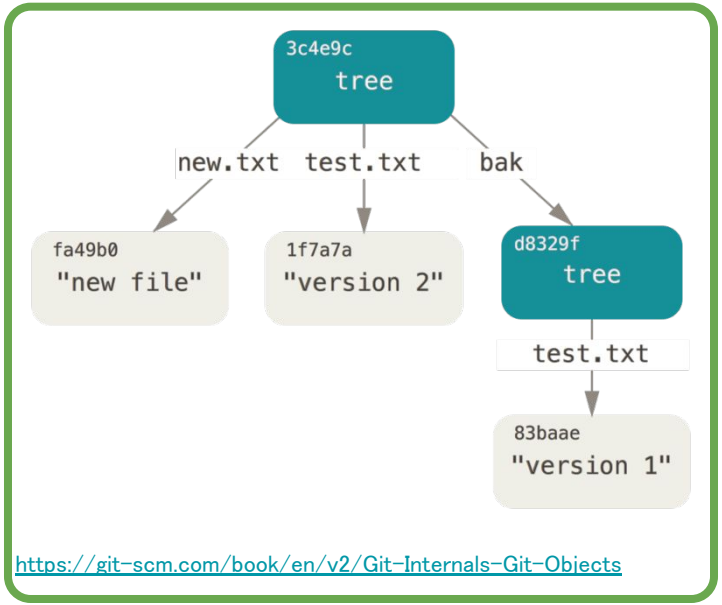


https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

コミットの仕組み

tree オブジェクトは簡易的なファイルシステムのようになっていて、図のような構造を持っている。

白い四角が blob オブジェクト、青い四角が tree オブジェクトで、それぞれファイルとディレクトリに相当する。



https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

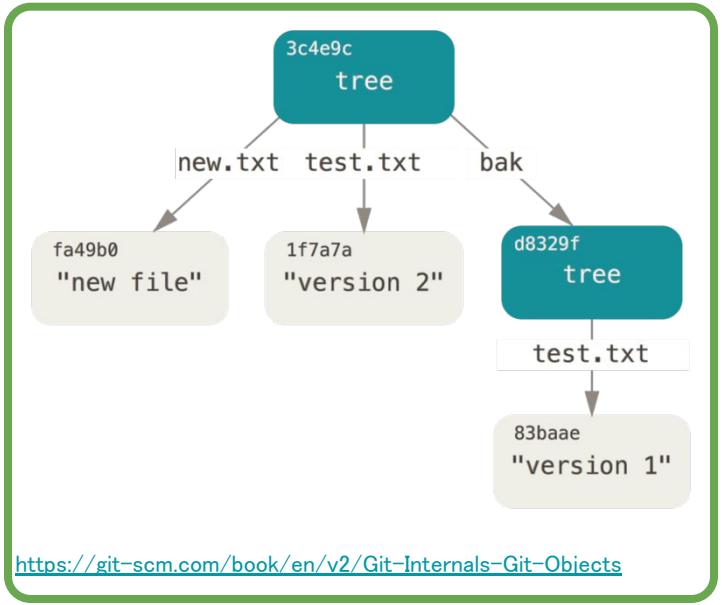
コミットの仕組み

tree オブジェクトは簡易的なファイルシステムのようになっていて、図のような構造を持っている。

白い四角が blob オブジェクト、青い四角が tree オブジェクトで、それぞれファイルとディレクトリに相当する。

blob オブジェクトはファイルの中身しか保存せず、

ファイル名の様なメタデータは tree オブジェクトが管理する。



https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

コミットの仕組み

tree オブジェクトは簡易的なファイルシステムのようになっていて、図のような構造を持っている。

白い四角が blob オブジェクト、青い四角が tree オブジェクトで、それぞれファイルとディレクトリに相当する。

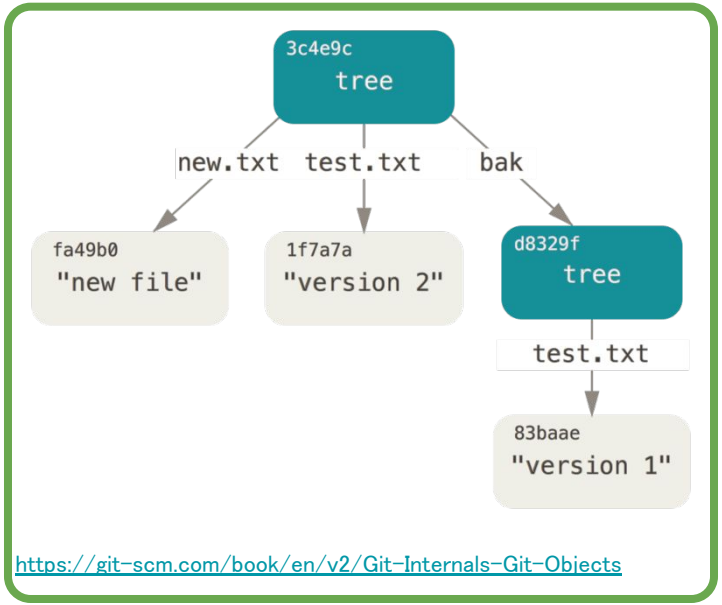
blob オブジェクトはファイルの中身しか保存せず、

ファイル名の様なメタデータは tree オブジェクトが管理する。

後述する commit オブジェクトは、リポジトリのルートにあたる

tree オブジェクトを参照していて、commit 時のリポジトリの

状態を再現できるようになっている。



https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

コミットの仕組み

tree オブジェクトは簡易的なファイルシステムのようになっていて、図のような構造を持っている。

白い四角が blob オブジェクト、青い四角が tree オブジェクトで、それぞれファイルとディレクトリに相当する。

blob オブジェクトはファイルの中身しか保存せず、

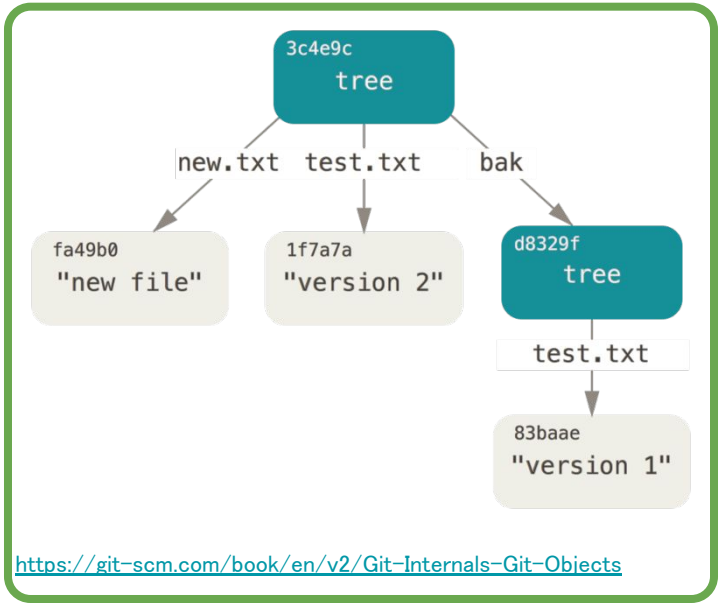
ファイル名の様なメタデータは tree オブジェクトが管理する。

後述する commit オブジェクトは、リポジトリのルートにあたる

tree オブジェクトを参照していて、commit 時のリポジトリの

状態を再現できるようになっている。

ちなみにこれは Merkle-tree というデータ構造。



https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

コミットの仕組み

ちなみに tree オブジェクトも cat-file で確認できる。

コミットの仕組み

ちなみに tree オブジェクトも cat-file で確認できる。

```
$ git cat-file -p 184135

100644 blob e2022389da0c18f0c8a0e2f9b27d58d197f41b32    README.md
```

コミットの仕組み

ちなみに tree オブジェクトも cat-file で確認できる。

16進数でダンプするよりも、もっと人間にやさしく表示してくれる。

```
$ git cat-file -p 184135

100644 blob e2022389da0c18f0c8a0e2f9b27d58d197f41b32    README.md
```

コミットの仕組み

ちなみに tree オブジェクトも cat-file で確認できる。

16進数でダンプするよりも、もっと人間にやさしく表示してくれる。

```
$ git cat-file -p 184135

100644 blob e2022389da0c18f0c8a0e2f9b27d58d197f41b32    README.md
```

tree オブジェクトを作ったら、次は commit オブジェクトを作る。

コミットの仕組み

commit オブジェクトの構造はこんな感じ。

```
$ cat .git/objects/56/4d5425c6eff4fab2be1cec9562003b6b64eea0 | zlib d

commit 232tree 184135bd0b06f80372a29a35c32ba2e8a5609dc6

parent dd08f09944c8c97a7185480308865l4d8ef4b887

author shumon84 <cameremon84@gmail.com> 1617805187 +0900

committer shumon84 <cameremon84@gmail.com> 1617805187 +0900


README.md を追加
```

コミットの仕組み

commit オブジェクトの構造はこんな感じ。

他のオブジェクトと同様に、これの SHA-1 ハッシュが commit ハッシュになる。

```
$ cat .git/objects/56/4d5425c6eff4fab2be1cec9562003b6b64eea0 | zlib d

commit 232tree 184135bd0b06f80372a29a35c32ba2e8a5609dc6

parent dd08f09944c8c97a7185480308865l4d8ef4b887

author shumon84 <cameremon84@gmail.com> 1617805187 +0900

committer shumon84 <cameremon84@gmail.com> 1617805187 +0900


README.md を追加
```

コミットの仕組み

commit オブジェクトの構造はこんな感じ。

他のオブジェクトと同様に、これの SHA-1 ハッシュが commit ハッシュになる。

```
$ cat .git/objects/56/4d5425c6eff4fab2be1cec9562003b6b64eea0 | zlib d

commit 232tree 184135bd0b06f80372a29a35c32ba2e8a5609dc6

parent dd08f09944c8c97a7185480308865l4d8ef4b887

author shumon84 <cameremon84@gmail.com> 1617805187 +0900

committer shumon84 <cameremon84@gmail.com> 1617805187 +0900


README.md を追加
```

コミットの仕組み

commit オブジェクトの構造はこんな感じ。

他のオブジェクトと同様に、これの SHA-1 ハッシュが commit ハッシュになる。

```
$ cat .git/objects/56/4d5425c6eff4fab2be1cec9562003b6b64eea0 | zlib d

commit 232tree 184135bd0b06f80372a29a35c32ba2e8a5609dc6

parent dd08f09944c8c97a7185480308865l4d8ef4b887

author shumon84 <cameremon84@gmail.com> 1617805187 +0900

committer shumon84 <cameremon84@gmail.com> 1617805187 +0900


README.md を追加
```

コミットの仕組み

commit オブジェクトの構造はこんな感じ。

他のオブジェクトと同様に、これの SHA-1 ハッシュが commit ハッシュになる。

```
$ cat .git/objects/56/4d5425c6eff4fab2be1cec9562003b6b64eea0 | zlib d

commit 232tree 184135bd0b06f80372a29a35c32ba2e8a5609dc6

parent dd08f09944c8c97a7185480308865l4d8ef4b887

author shumon84 <cameremon84@gmail.com> 1617805187 +0900

committer shumon84 <cameremon84@gmail.com> 1617805187 +0900


README.md を追加
```

コミットの仕組み

commit オブジェクトの構造はこんな感じ。

他のオブジェクトと同様に、これの SHA-1 ハッシュが commit ハッシュになる。

```
$ cat .git/objects/56/4d5425c6eff4fab2be1cec9562003b6b64eea0 | zlib d

commit 232tree 184135bd0b06f80372a29a35c32ba2e8a5609dc6

parent dd08f09944c8c97a7185480308865l4d8ef4b887

author shumon84 <cameremon84@gmail.com> 1617805187 +0900

committer shumon84 <cameremon84@gmail.com> 1617805187 +0900


README.md を追加
```

コミットの仕組み

commit オブジェクトの構造はこんな感じ。

他のオブジェクトと同様に、これの SHA-1 ハッシュが commit ハッシュになる。

```
$ cat .git/objects/56/4d5425c6eff4fab2be1cec9562003b6b64eea0 | zlib d

commit 232tree 184135bd0b06f80372a29a35c32ba2e8a5609dc6

parent dd08f09944c8c97a7185480308865l4d8ef4b887

author shumon84 <cameremon84@gmail.com> 1617805187 +0900

committer shumon84 <cameremon84@gmail.com> 1617805187 +0900


README.md を追加
```

コミットの仕組み

commit オブジェクトの構造はこんな感じ。

他のオブジェクトと同様に、これの SHA-1 ハッシュが commit ハッシュになる。

```
$ cat .git/objects/56/4d5425c6eff4fab2be1cec9562003b6b64eea0 | zlib d

commit 232tree 184135bd0b06f80372a29a35c32ba2e8a5609dc6

parent dd08f09944c8c97a7185480308865l4d8ef4b887

author shumon84 <cameremon84@gmail.com> 1617805187 +0900

committer shumon84 <cameremon84@gmail.com> 1617805187 +0900


README.md を追加
```

コミットの仕組み

commit オブジェクトに含まれている情報は↓の通り。

- リポジトリのルートディレクトリの tree ハッシュ
- 親 commit ハッシュ
- committer と author のタイムスタンプ・名前・メアド
- コミットメッセージ

コミットの仕組み

commit オブジェクトに含まれている情報は↓の通り。

- リポジトリのルートディレクトリの tree ハッシュ
- 親 commit ハッシュ
- committer と author のタイムスタンプ・名前・メアド
- コミットメッセージ

このうち、どれか 1 つでも変わると、(SHA-1 が衝突しない限り)別の commit ハッシュ になる。

コミットの仕組み

commit オブジェクトに含まれている情報は↓の通り。

- リポジトリのルートディレクトリの tree ハッシュ
- 親 commit ハッシュ
- committer と author のタイムスタンプ・名前・メアド
- コミットメッセージ

このうち、どれか 1 つでも変わると、(SHA-1 が衝突しない限り)別の commit ハッシュ になる。

ちなみに、commit ハッシュがどれくらい衝突しないのかというと、

「1000 人で 1 日 10 回、40 京年間 commit し続けても、衝突する可能性は 50 %」

と言われている。

コミットの仕組み

commit オブジェクトに含まれている情報は、↓の通り

- リポジトリのルートディレクト
- **親 commit ハッシュ**
- committer と author のタイムスタンプ・名前・メアド
- コミットメッセージ

親 commit のハッシュを
持っていることが超重要

このうち、どれか 1 つでも変わると、(SHA-1 が衝突しない限り)別の commit ハッシュ になる。

ちなみに、commit ハッシュがどれくらい衝突しないのかというと、

「1000 人で 1 日 10 回、40 京年間 commit し続けても、衝突する可能性は 50 %」

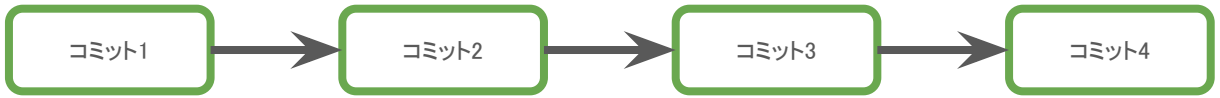
と言われている。

コミットの仕組み

親 commit ハッシュを commit オブジェクトの一部に含めることで、改ざんされていないことが保証できる。

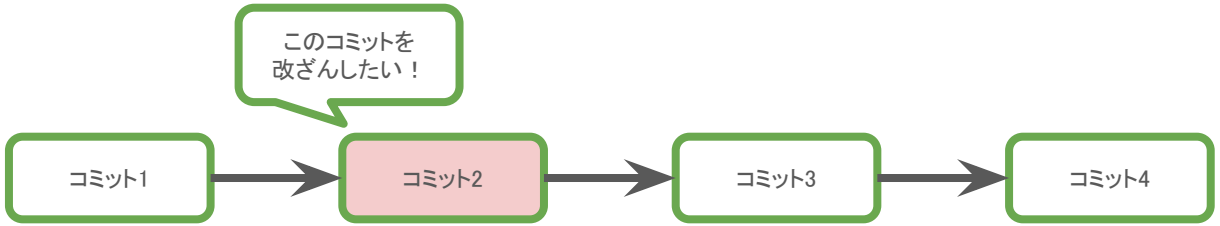
コミットの仕組み

親 commit ハッシュを commit オブジェクトの一部に含めることで、改ざんされていないことが保証できる。



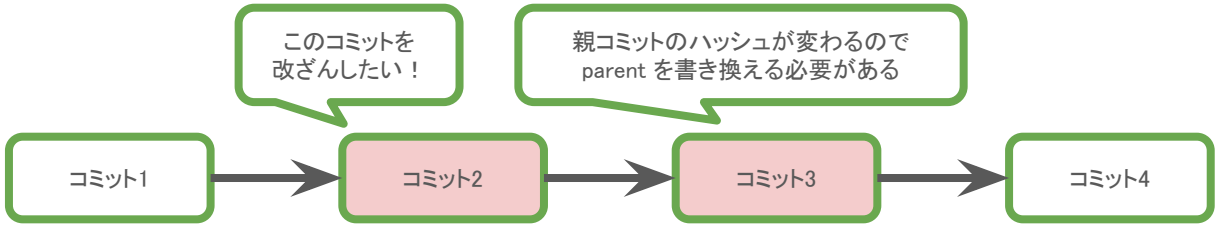
コミットの仕組み

親 commit ハッシュを commit オブジェクトの一部に含めることで、改ざんされていないことが保証できる。



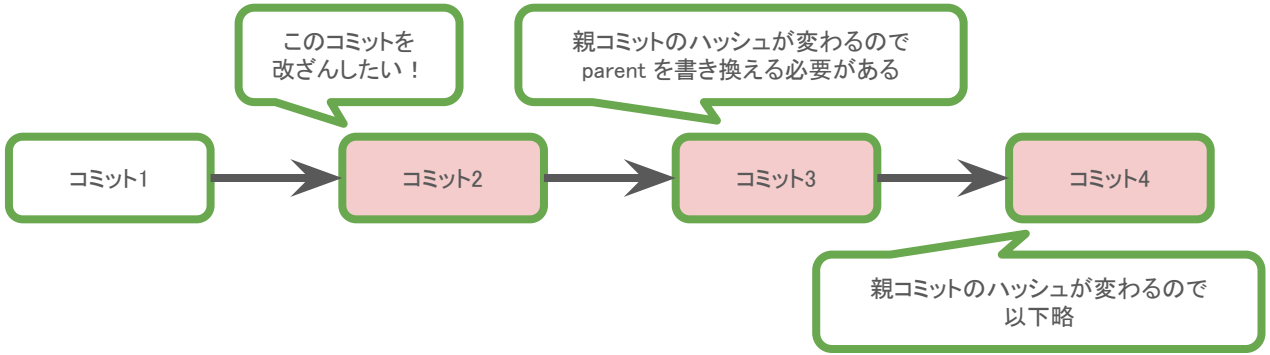
コミットの仕組み

親 commit ハッシュを commit オブジェクトの一部に含めることで、改ざんされていないことが保証できる。



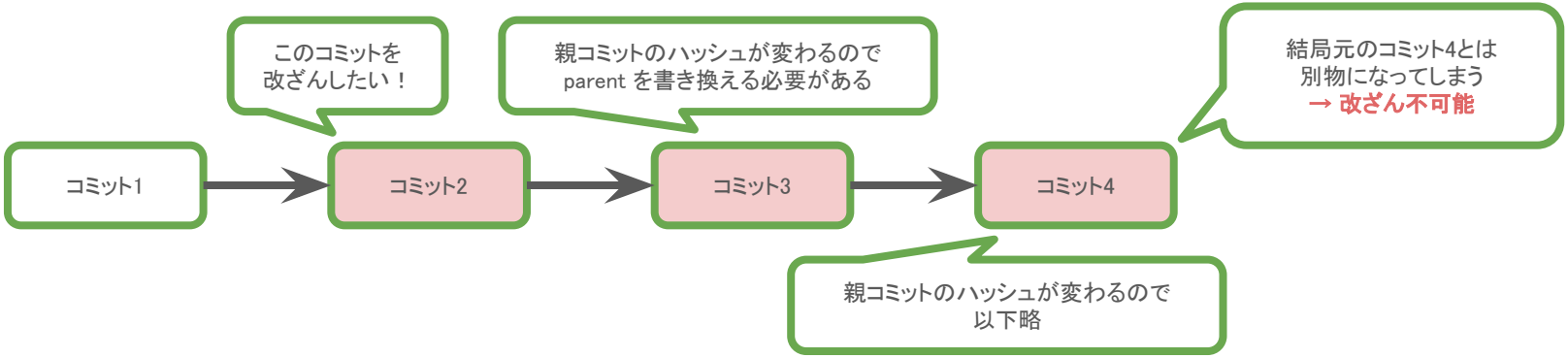
コミットの仕組み

親 commit ハッシュを commit オブジェクトの一部に含めることで、改ざんされていないことが保証できる。



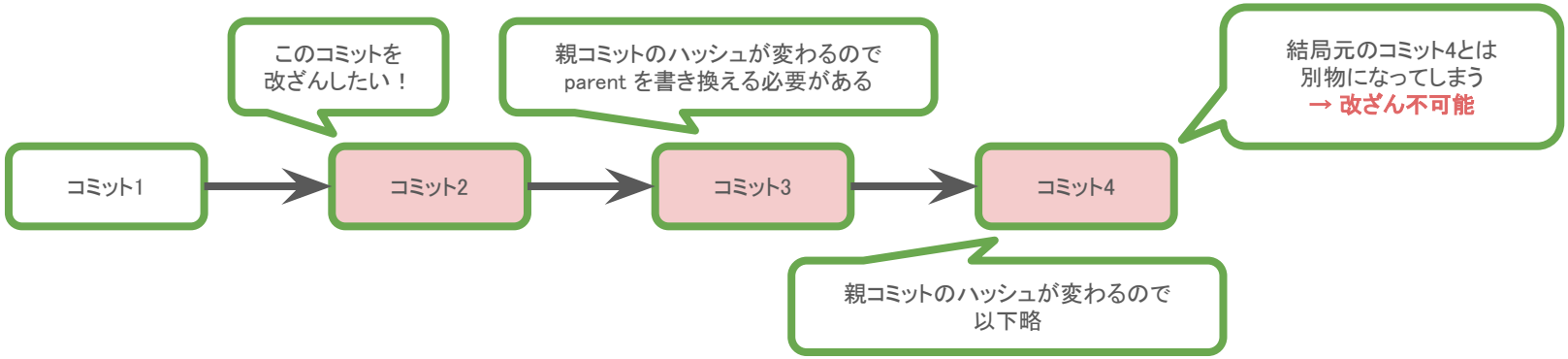
コミットの仕組み

親 commit ハッシュを commit オブジェクトの一部に含めることで、改ざんされていないことが保証できる。



コミットの仕組み

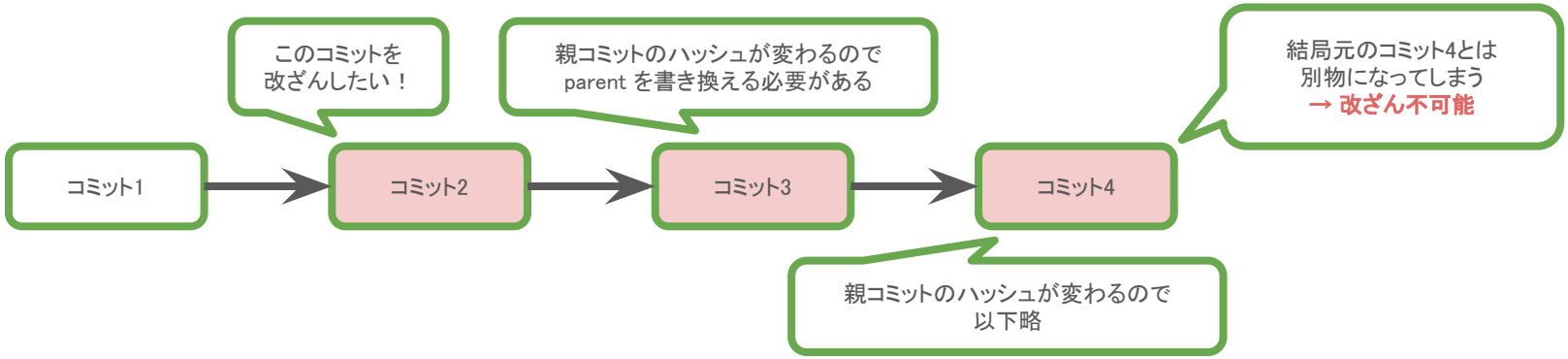
親 commit ハッシュを commit オブジェクトの一部に含めることで、改ざんされていないことが保証できる。



過去の commit を改ざんするには、それ以降全ての commit を改ざんする必要があり、最新の commit も別物に。

コミットの仕組み

親 commit ハッシュを commit オブジェクトの一部に含めることで、改ざんされていないことが保証できる。

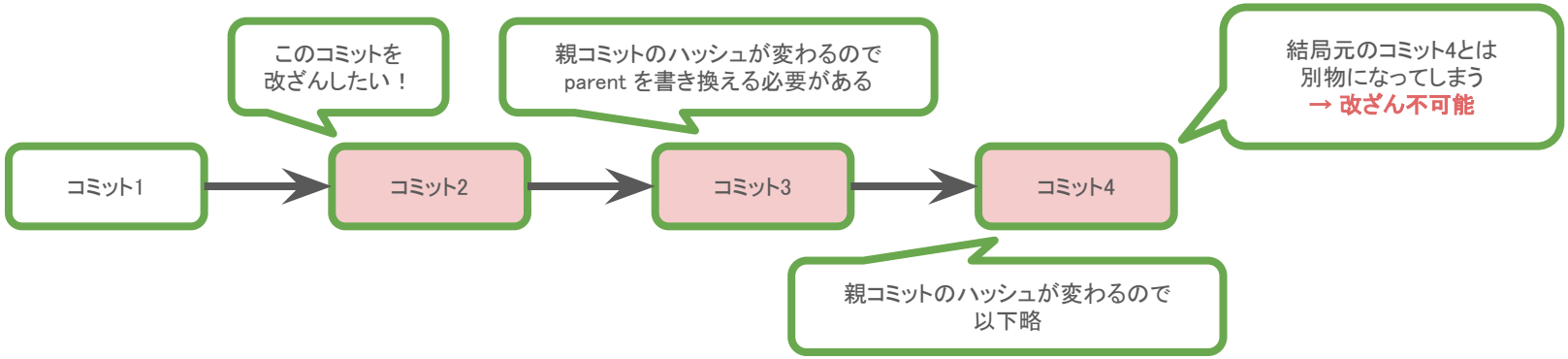


過去の commit を改ざんするには、それ以降全ての commit を改ざんする必要があり、最新の commit も別物に。

つまり最新の commit ハッシュを計算するだけで、過去の履歴すべてが valid であることを証明できる。

コミットの仕組み

親 commit ハッシュを commit オブジェクトの一部に含めることで、改ざんされていないことが保証できる。



過去の commit を改ざんするには、それ以降全ての commit を改ざんする必要があり、最新の commit も別物に。

つまり最新の commit ハッシュを計算するだけで、過去の履歴すべてが valid であることを証明できる。

→ **Git はブロックチェーン！！**

コミットの仕組み

blob オブジェクトは add 時に、tree オブジェクトは commit 時に生成する。

commit すると内部的に実行している処理は大まかには次の通り(本当はもうちょっと色々やってるけど割愛)

1. index から tree オブジェクトを生成
2. commit オブジェクトを生成
3. **HEAD を新しい commit ハッシュに書き換え**

それぞれ細かく見ていく。

コミットの仕組み

commit オブジェクトを作ったら、Git は最後に HEAD を書き換える。

コミットの仕組み

commit オブジェクトを作ったら、Git は最後に HEAD を書き換える。

→ HEAD とは？

コミットの仕組み

commit オブジェクトを作ったら、Git は最後に HEAD を書き換える。

→ HEAD とは？

HEAD の話をする前に、**refs** の話をする必要がある。(HEAD は refs の一種)

コミットの仕組み

commit オブジェクトを作ったら、Git は最後に HEAD を書き換える。

→ HEAD とは？

HEAD の話をする前に、**refs** の話をする必要がある。(HEAD は refs の一種)

refs は特定の commit を指すポインタのようなもの。commit ハッシュのエイリアスとも言える。

コミットの仕組み

commit オブジェクトを作ったら、Git は最後に HEAD を書き換える。

→ HEAD とは？

HEAD の話をする前に、**refs** の話をする必要がある。(HEAD は refs の一種)

refs は特定の commit を指すポインタのようなもの。commit ハッシュのエイリアスとも言える。

具体的には↓などが refs にあたる。

- tag
- branch
- HEAD

refs について

まずは refs の中でも、最も単純な light weight tag を見ていく。

refs について

まずは refs の中でも、最も単純な light weight tag を見ていく。

light weight tag は本当にただ特定の commit を指しているだけ。

refs について

まずは refs の中でも、最も単純な light weight tag を見ていく。

light weight tag は本当にただ特定の commit を指しているだけ。

```
$ git tag light-weight # オプションなしでタグを作ると light weight tag になる
```

refs について

まずは refs の中でも、最も単純な light weight tag を見ていく。

light weight tag は本当にただ特定の commit を指しているだけ。

```
$ git tag light-weight # オプションなしでタグを作ると light weight tag になる

$ git log -n 1 --oneline

7c4c08d (HEAD -> master, tag: light-weight) Merge branch 'develop'
```

refs について

まずは refs の中でも、最も単純な light weight tag を見ていく。

light weight tag は本当にただ特定の commit を指しているだけ。

タグを作ると .git/refs/tags/ 以下に保存される。

```
$ git tag light-weight # オプションなしでタグを作ると light weight tag になる

$ git log -n 1 --oneline

7c4c08d (HEAD -> master, tag: light-weight) Merge branch 'develop'
```

refs について

まずは refs の中でも、最も単純な light weight tag を見ていく。

light weight tag は本当にただ特定の commit を指しているだけ。

タグを作ると .git/refs/tags/ 以下に保存される。

```
$ git tag light-weight # オプションなしでタグを作ると light weight tag になる

$ git log -n 1 --oneline

7c4c08d (HEAD -> master, tag: light-weight) Merge branch 'develop'

$ cat .git/refs/tags/light-weight

7c4c08d68be5a53406af4582a961a460b0db83cd
```

refs について

次に annotated tag の挙動を見ていく。

refs について

次に annotated tag の挙動を見ていく。

annotated tag とはコメントが付けられるタグのこと。

-a オプションで作ることができる。

refs について

次に annotated tag の挙動を見ていく。

annotated tag とはコメントが付けられるタグのこと。

-a オプションで作ることができる。

```
$ git tag -a annotated -m "メッセージ"

$ git log -n 1 --oneline

7c4c08d (HEAD -> master, tag: annotated) Merge branch 'develop'
```

refs について

次に annotated tag の挙動を見ていく。

annotated tag とはコメントが付けられるタグのこと。

-a オプションで作ることができる。

```
$ git tag -a annotated -m "メッセージ"

$ git log -n 1 --oneline

7c4c08d (HEAD -> master, tag: annotated) Merge branch 'develop'

$ cat .git/refs/tags/annotated

e0114d0446b25c2c653fa5dd28c678602033c48b
```

refs について

次に annotated tag の挙動を見ていく。

annotated tag とはコメントが付けられるタグのこと。

-a オプションで作ることができる。

```
$ git tag -a annotated -m "メッセージ"

$ git log -n 1 --oneline

7c4c08d (HEAD -> master, tag: annotated) Merge branch 'develop'

$ cat .git/refs/tags/annotated

e0114d0446b25c2c653fa5dd28c678602033c48b
```

light weight tag と違って、直接 commit ハッシュが書かれているわけではない。

refs について

次に annotated tag の挙動を見ていく。

annotated tag とはコメントが付けられるタグのこと。

-a オプションで作ることができる。

```
$ git tag -a annotated -m "メッセージ"

$ git log -n 1 --oneline

7c4c08d (HEAD -> master, tag: annotated) Merge branch 'develop'

$ cat .git/refs/tags/annotated

e0114d0446b25c2c653fa5dd28c678602033c48b
```

light weight tag と違って、直接 commit ハッシュが書かれているわけではない。

→ じゃあ何なの？

コミットの仕組み

blob ハッシュってなに？→ Git に管理されている オブジェクトの 1 種(の key)のこと。

Git には様々なデータを、「オブジェクト」と呼ばれる概念で表現している。

オブジェクトには以下の 4 種類がある。

- commit オブジェクト
  - コミットの情報が入っているオブジェクト
- tree オブジェクト
  - ディレクトリの情報が入っているオブジェクト
- blob オブジェクト
  - ファイルの情報が入っているオブジェクト
- **tag オブジェクト**
  - **annotated tag の情報が入っているオブジェクト**

ただしGit で扱うデータが全てオブジェクトなわけではない。例えば branch はオブジェクトで管理していない。

refs について

オブジェクトなので、例によって cat-file で覗いてみる。

もちろん実体は .git/objects/ 以下に zlib で圧縮して保存されている。

refs について

オブジェクトなので、例によって cat-file で覗いてみる。

もちろん実体は .git/objects/ 以下に zlib で圧縮して保存されている。

```
$ git cat-file -p e0114d0446b25c2c653fa5dd28c678602033c48b

object 7c4c08d68be5a53406af4582a961a460b0db83cd

type commit

tag annotated

tagger shumon84 <cameremon84@gmail.com> 1618330965 +0900



メッセージ
```

refs について

オブジェクトなので、例によって cat-file で覗いてみる。

もちろん実体は .git/objects/ 以下に zlib で圧縮して保存されている。

```
$ git cat-file -p e0114d0446b25c2c653fa5dd28c678602033c48b

object 7c4c08d68be5a53406af4582a961a460b0db83cd

type commit

tag annotated

tagger shumon84 <cameremon84@gmail.com> 1618330965 +0900



メッセージ
```

object の欄に、tag が指している
commit のハッシュが書かれている

refs について

オブジェクトなので、例によって cat-file で覗いてみる。

もちろん実体は .git/objects/ 以下に zlib で圧縮して保存されている。

```
$ git cat-file -p e0114d0446b25c2c653fa5dd28c678602033c48b

object 7c4c08d68be5a53406af4582a961a460b0db83cd

type commit

tag annotated

tagger shumon84 <cameremon84@gmail.com> 1618330965 +0900



メッセージ
```

object の欄に、tag が指している
commit のハッシュが書かれている

commit オブジェクトにちょっと似てる。

refs について

次に branch の解説。

refs について

次に branch の解説。

branch は基本的に light-weight tag とほぼ変わらない。

refs について

次に branch の解説。

branch は基本的に light-weight tag とほぼ変わらない。

```
$ cat .git/refs/heads/master

7c4c08d68be5a53406af4582a961a460b0db83cd
```

refs について

次に branch の解説。

branch は基本的に light-weight tag とほぼ変わらない。

```
$ cat .git/refs/heads/master

7c4c08d68be5a53406af4582a961a460b0db83cd
```

保存場所が .git/refs/heads/ 以下になっているだけで、書かれている内容は同じ。

refs について

次に branch の解説。

branch は基本的に light-weight tag とほぼ変わらない。

```
$ cat .git/refs/heads/master

7c4c08d68be5a53406af4582a961a460b0db83cd
```

保存場所が .git/refs/heads/ 以下になっているだけで、書かれている内容は同じ。

指している commit ハッシュが直接書かれているだけ。

refs について

次に branch の解説。

branch は基本的に light-weight tag とほぼ変わらない。

```
$ cat .git/refs/heads/master

7c4c08d68be5a53406af4582a961a460b0db83cd
```

保存場所が .git/refs/heads/ 以下になっているだけで、書かれている内容は同じ。

指している commit ハッシュが直接書かれているだけ。

→ tag との違いは、tag は基本的に書き換えないのに対して、branch はどんどん書き換わっていくところ

refs について

次に branch の解説。

branch は基本的に light-weight tag とほぼ変わらない。

```
$ cat .git/refs/heads/master

7c4c08d68be5a53406af4582a961a460b0db83cd
```

保存場所が .git/refs/heads/ 以下になっているだけで、書かれている内容は同じ。

指している commit ハッシュが直接書かれているだけ。

→ tag との違いは、tag は基本的に書き換えないのに対して、branch はどんどん書き換わっていくところ

ちなみに、branch 名がそのままファイルパスになるため、「feature/hoge」という branch を作ると、

feature/ というディレクトリが作られてしまうため、それ以降「feature」という branch は作れなくなる。

refs について

話を戻して、HEAD について。

refs について

話を戻して、HEAD について。HEAD は現在の commit を指す refs。

refs について

話を戻して、HEAD について。HEAD は現在の commit を指す refs。

checkout したときは HEAD が書き換わっている。

refs について

話を戻して、HEAD について。HEAD は現在の commit を指す refs。

checkout したときは HEAD が書き換わっている。

.git/HEAD に保存されている。

refs について

話を戻して、HEAD について。HEAD は現在の commit を指す refs。

checkout したときは HEAD が書き換わっている。

.git/HEAD に保存されている。

```
$ cat .git/HEAD

ref: refs/heads/master
```

refs について

話を戻して、HEAD について。HEAD は現在の commit を指す refs。

checkout したときは HEAD が書き換わっている。

.git/HEAD に保存されている。

```
$ cat .git/HEAD

ref: refs/heads/master
```

branch 名で checkout すると、commit ハッシュではなく、branch の場所が書き込まれる。

refs について

話を戻して、HEAD について。HEAD は現在の commit を指す refs。

checkout したときは HEAD が書き換わっている。

.git/HEAD に保存されている。

```
$ cat .git/HEAD

ref: refs/heads/master

$ cat .git/refs/heads/master

7c4c08d68be5a53406af4582a961a460b0db83cd
```

branch 名で checkout すると、commit ハッシュではなく、branch の場所が書き込まれる。

refs について

話を戻して、HEAD について。HEAD は現在の commit を指す refs。

checkout したときは HEAD が書き換わっている。

.git/HEAD に保存されている。

```
$ cat .git/HEAD

ref: refs/heads/master

$ cat .git/refs/heads/master

7c4c08d68be5a53406af4582a961a460b0db83cd
```

branch 名で checkout すると、commit ハッシュではなく、branch の場所が書き込まれる。

参照を辿って、現在の commit は推移的に解決される。

コミットの仕組み

ようやく commit の内部処理の話に戻ります。

コミットの仕組み

ようやく commit の内部処理の話に戻ります。

commit オブジェクトを作ったら、Git は最後に HEAD を書き換える。

コミットの仕組み

ようやく commit の内部処理の話に戻ります。

commit オブジェクトを作ったら、Git は最後に HEAD を書き換える。

HEAD が直接 commit ハッシュを参照している場合

HEAD が branch を参照している場合

コミットの仕組み

ようやく commit の内部処理の話に戻ります。

commit オブジェクトを作ったら、Git は最後に HEAD を書き換える。

HEAD が直接 commit ハッシュを参照している場合

→ HEAD の commit ハッシュを書き換える

HEAD が branch を参照している場合

コミットの仕組み

ようやく commit の内部処理の話に戻ります。

commit オブジェクトを作ったら、Git は最後に HEAD を書き換える。

HEAD が直接 commit ハッシュを参照している場合

→ HEAD の commit ハッシュを書き換える

HEAD が branch を参照している場合

→ HEAD が参照している branch の commit ハッシュを書き換える

コミットの仕組み まとめ

Git では、1回コミットするまでに次のような処理をしている。

コミットの仕組み まとめ

Git では、1回コミットするまでに次のような処理をしている。

1. コードを編集する

コミットの仕組み まとめ

Git では、1回コミットするまでに次のような処理をしている。

1. コードを編集する
2. 編集したコードを add する
   - index の更新
   - blob オブジェクトの生成

コミットの仕組み まとめ

Git では、1回コミットするまでに次のような処理をしている。

1. コードを編集する
2. 編集したコードを add する
   - index の更新
   - blob オブジェクトの生成
3. commit する
   - tree オブジェクトの生成
   - commit オブジェクトの生成
   - HEAD の書き換え

コミットの仕組み まとめ

Git では、1回コミットするまでに次のような処理をしている。

1. コードを編集する
2. 編集したコードを add する
   - index の更新
   - blob オブジェクトの生成
3. commit する
   - tree オブジェクトの生成
   - commit オブジェクトの生成
   - HEAD の書き換え

もうちょっと色々やってるけど、大まかには「コミットする」というとこんな感じの処理のことを指す。

checkout, reset の仕組み

checkout, reset の仕組み

すでに講義中に何度も登場している、みんなお馴染み checkout。

checkout, reset の仕組み

すでに講義中に何度も登場している、みんなお馴染み checkout。

checkout に比べると、複雑で上級者向けのコマンドと思われている reset。

checkout, reset の仕組み

すでに講義中に何度も登場している、みんなお馴染み checkout。

checkout に比べると、複雑で上級者向けのコマンドと思われている reset。

実は、この 2 つのコマンドは結構似たようなことをしている。

checkout, reset の仕組み

すでに講義中に何度も登場している、みんなお馴染み checkout。

checkout に比べると、複雑で上級者向けのコマンドと思われている reset。

実は、この 2 つのコマンドは結構似たようなことをしている。

この 2 つは、基本的に↓の 3 つを書き換えるコマンド。

- ワークツリー(作業ディレクトリ)
- index
- HEAD

checkout, reset の仕組み

すでに講義中に何度も登場している、みんなお馴染み checkout。

checkout に比べると、複雑で上級者向けのコマンドと思われている reset。

実は、この 2 つのコマンドは結構似たようなことをしている。

この 2 つは、基本的に↓の 3 つを書き換えるコマンド。

- ワークツリー(作業ディレクトリ)
- index
- HEAD

それぞれ見ていく。

checkout の仕組み

まず checkout は、指定した commit にワークツリーも index も HEAD も全部向けるコマンド。

checkout の仕組み

まず checkout は、指定した commit にワークツリーも index も HEAD も全部向けるコマンド。

1. 指定した commit が参照している tree をワークツリーに展開する。

checkout の仕組み

まず checkout は、指定した commit にワークツリーも index も HEAD も全部向けるコマンド。

1. 指定した commit が参照している tree をワークツリーに展開する。
2. index をワークツリーと同期する(= tree と同じ状態にする)。

checkout の仕組み

まず checkout は、指定した commit にワークツリーも index も HEAD も全部向けるコマンド。

1. 指定した commit が参照している tree をワークツリーに展開する。
2. index をワークツリーと同期する(= tree と同じ状態にする)。
3. HEAD を指定した commit に変更する。

checkout の仕組み

まず checkout は、指定した commit にワークツリーも index も HEAD も全部向けるコマンド。

1. 指定した commit が参照している tree をワークツリーに展開する。
2. index をワークツリーと同期する(= tree と同じ状態にする)。
3. HEAD を指定した commit に変更する。

refs を指定していた場合、HEAD にはその参照が書き込まれる。

reset の仕組み

reset は、オプションによって挙動が変わる。

reset の仕組み

reset は、オプションによって挙動が変わる。

おおまかには soft, mixed, hard の 3 種類のオプションがある。

reset の仕組み

reset は、オプションによって挙動が変わる。

おおまかには soft, mixed, hard の 3 種類のオプションがある。

指定した commit ハッシュに向けて、それぞれの内容を書き換える。

reset の仕組み

reset は、オプションによって挙動が変わる。

おおまかには soft, mixed, hard の 3 種類のオプションがある。

指定した commit ハッシュに向けて、それぞれの内容を書き換える。

	HEAD	index	ワークツリー
`git reset --soft <commit ハッシュ>`	書き換える		
`git reset --mixed <commit ハッシュ>`	書き換える	書き換える	
`git reset --hard <commit ハッシュ>`	書き換える	書き換える	書き換える

reset の仕組み

reset は、オプションによって挙動が変わる。

おおまかには soft, mixed, hard の 3 種類のオプションがある。

指定した commit ハッシュに向けて、それぞれの内容を書き換える。

	HEAD	index	ワークツリー
`git reset --soft <commit ハッシュ>`	書き換える		
`git reset --mixed <commit ハッシュ>`	書き換える	書き換える	
`git reset --hard <commit ハッシュ>`	書き換える	書き換える	書き換える

--hard は全部書き換えるし、checkout と違いはあるの？

reset の仕組み

reset は、オプションによって挙動が変わる。

おおまかには soft, mixed, hard の 3 種類のオプションがある。

指定した commit ハッシュに向けて、それぞれの内容を書き換える。

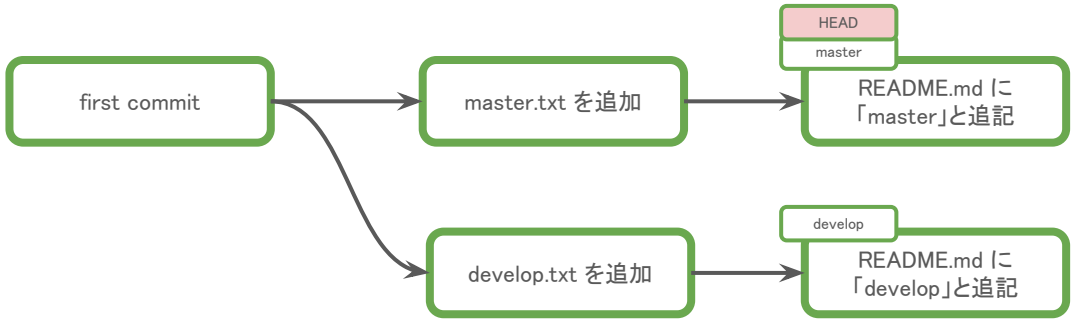
	HEAD	index	ワークツリー
`git reset --soft <commit ハッシュ>`	書き換える		
`git reset --mixed <commit ハッシュ>`	書き換える	書き換える	
`git reset --hard <commit ハッシュ>`	書き換える	書き換える	書き換える

--hard は全部書き換えるし、checkout と違いはあるの？

reset と checkout の最も大きな違いは、HEAD が branch を参照していた場合の挙動。

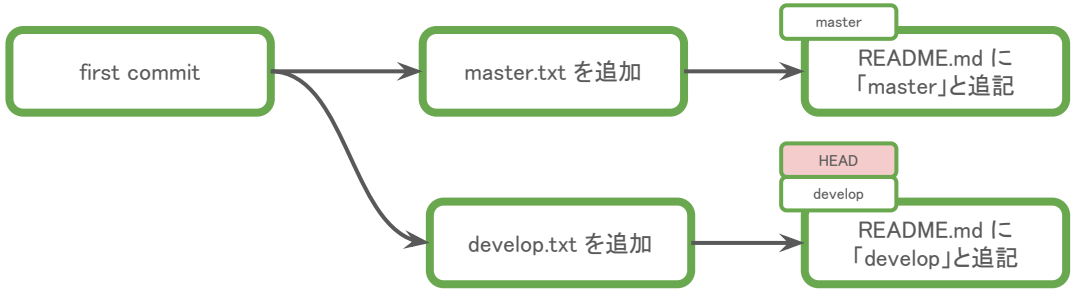
reset の仕組み

checkout は HEAD が branch を参照していても、書き換えるのは HEAD のみ。



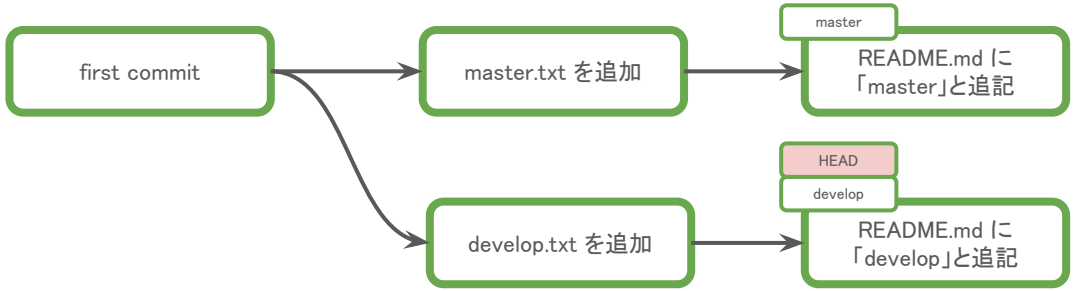
reset の仕組み

checkout は HEAD が branch を参照していても、書き換えるのは HEAD のみ。

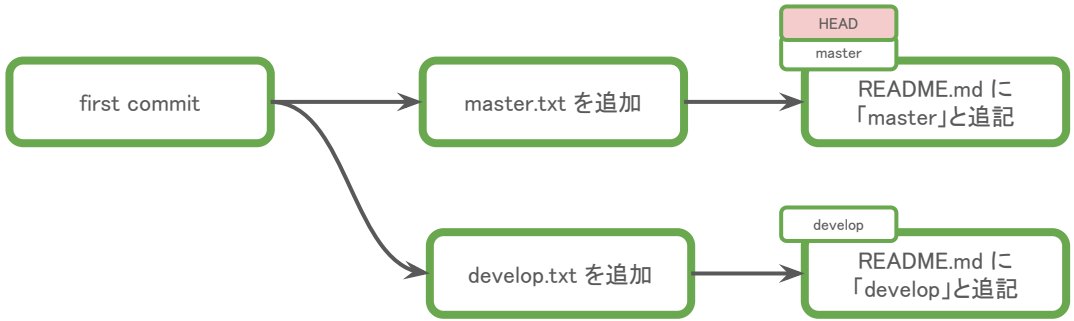


reset の仕組み

checkout は HEAD が branch を参照していても、書き換えるのは HEAD のみ。

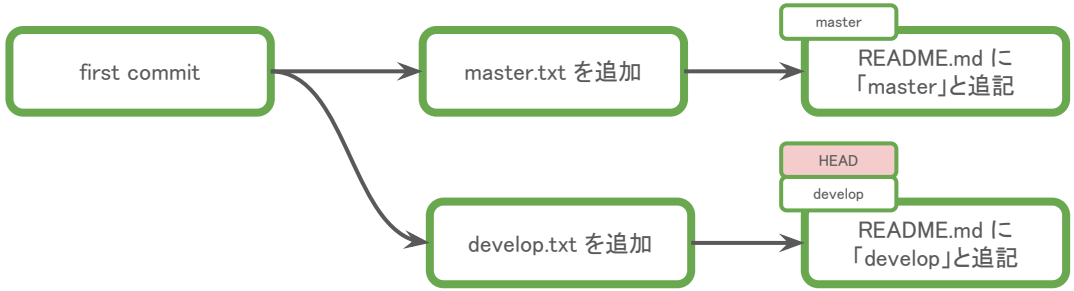


reset は HEAD が branch を参照していた場合、branch の参照先も書き換える。

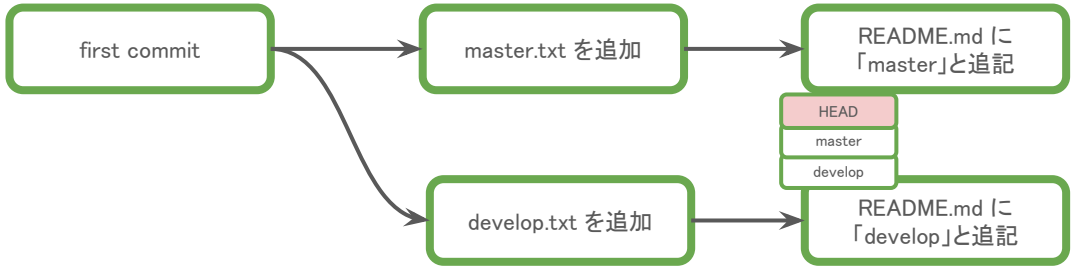


reset の仕組み

checkout は HEAD が branch を参照していても、書き換えるのは HEAD のみ。



reset は HEAD が branch を参照していた場合、branch の参照先も書き換える。



reset の仕組み

branch の参照先を変えられる機能を応用すると、reset は色々な使い方ができる。

reset の仕組み

branch の参照先を変えられる機能を応用すると、reset は色々な使い方ができる。

① コミットを無かったことにする

reset の仕組み

branch の参照先を変えられる機能を応用すると、reset は色々な使い方ができる。

① コミットを無かったことにする

1 つ前の commit ハッシュに branch を向けることで、最新のコミットをなかったことにできる。

reset の仕組み

branch の参照先を変えられる機能を応用すると、reset は色々な使い方ができる。

① コミットを無かったことにする

1 つ前の commit ハッシュに branch を向けることで、最新のコミットをなかったことにできる。

② add の取り消し

reset の仕組み

branch の参照先を変えられる機能を応用すると、reset は色々な使い方ができる。

① コミットを無かったことにする

1 つ前の commit ハッシュに branch を向けることで、最新のコミットをなかったことにできる。

② add の取り消し

git reset --mixed HEAD とすると、index を HEAD の状況に戻せるので、add を取り消すことができる。

reset の仕組み

branch の参照先を変えられる機能を応用すると、reset は色々な使い方ができる。

① コミットを無かったことにする

1 つ前の commit ハッシュに branch を向けることで、最新のコミットをなかったことにできる。

② add の取り消し

git reset --mixed HEAD とすると、index を HEAD の状況に戻せるので、add を取り消すことができる。

git 2.23 で追加された、restore というサブコマンドを使っても同じことができる※experimental

reset の仕組み

branch の参照先を変えられる機能を応用すると、reset は色々な使い方ができる。

① コミットを無かったことにする

1 つ前の commit ハッシュに branch を向けることで、最新のコミットをなかったことにできる。

② add の取り消し

git reset --mixed HEAD とすると、index を HEAD の状況に戻せるので、add を取り消すことができる。

git 2.23 で追加された、restore というサブコマンドを使っても同じことができる※experimental

使い方は無限大。

Git Challenge に挑戦

の前に休憩 10分くらい

休憩中の余談 ①

一般的なファイルシステムで採用されている、パスなど「ストレージ内のどこに保存されているか」という情報から

保存されているデータにアクセスするストレージは Location Addressable Storage (LAS) と呼ばれる。

階層構造を持つのではなく、Git のようにデータの内容(のハッシュ)そのものによってデータにアクセスする

ストレージは Content Addressable Storage (CAS) と呼ばれる。

CAS は、データの更新頻度が少ないとき効率がよく、保存してからデータが変更されていないことを保証できる。

長期間保存する + 改変してはならないことが、法令で定められているような情報の保存によく使われるらしい。

休憩中の余談 ②

index や tree オブジェクトでファイルに付いていた 100644 という謎のフラグの話。

```
$ git ls-files --stage

100644 e965047ad7c57865823c7d992b1d046ea66edf78 0  README.md
```

上 3 桁がファイルの種類、下 3 桁が UNIX 形式のファイルパーミッションを表している。

ファイルの種類は次の通り。

- 040 : ディレクトリ(tree)
- 100 : 通常のファイル
- 120 : シンボリックリンク
- 160 : submodule

下 3 桁は、owner に実行権限があるなら 755、ないなら 644 になる。

(つまり 644 と 755 しか Git は管理できない)

休憩中の余談 ③

割愛していた commit 時に「もうちょっと色々やってる」って話を詳しく。

- 各種 Hooks の起動
  - .git/hooks/ 以下にスクリプトを置いておくと、対応するイベントが発火したときに実行してくれる。
  - どんなイベントを hook できるのかは公式ドキュメント参照
- reflog の更新
  - .git/logs/HEAD に HEAD の向き先の移動履歴を書き込む。
  - HEAD が branch を参照していた場合、.git/log/refs/<branch 名> にも移動履歴を書き込む。
    - ちなみにこれらの移動履歴は git reflog で確認できる。
- COMMIT_EDITMSG の編集
  - コミットメッセージを設定せずに commit すると自動でエディタが起動するが、そのエディタが開いているファイルが COMMIT_EDITMSG
  - ここに書き込まれている内容が、コミットメッセージとして commit オブジェクトに取り込まれる。

休憩中の余談 ④

reset を使っても、本当に commit がなかったことにできるわけではない。

reset はあくまでも、HEAD・index・ワークツリーを書き換えるコマンドのため、当然 commit オブジェクトを

削除したりはできない。

そうすると、どこからも参照されていない commit が出来上がることがある。

→ そういうオブジェクトは一定期間経つと、**git gc** というコマンドが走って消される。

→ ちなみにどこからも参照されていないオブジェクトは**git fsck** で探せるよ

Git Challenge に挑戦