

git勉強会資料

目標:

svn相当+ α まで使えるようにしましょう

Takuya KOIKE

初版:2012/8/14
細部更新:2014/5/15

GITにはたくさんのコマンドが存在します。

それぞれのコマンドには多くのオプションがあります。

これら全てを覚えることはありません。

多くのコマンドやオプションは、日常的に使うものではないですが、時々必要になることがあります。

忘れてしまうこともあるでしょう。

そんな時のために、
まずはhelpについて知りましょう。

まずはhelpの引き方を覚えましょう

- 個々のコマンドの使い方を忘れてしまった時にはhelpを読みましょう

```
$ git help <command>
```

```
$ git <command> --help
```

※上記はどれでも、インストールフォルダのhtmlファイル(/doc/*)を読みにいきます。

- まずはgitコマンドのmanから見てみましょう

```
$ git help git
```

helpについて

helpの読み方・1

NAME

git - the stupid content tracker

SYNOPSIS

```
git [--version] [--exec-path[=GIT_EXEC_PATH]] [--html-path]
    [-p|--paginate|--no-pager] [--no-replace-objects]
    [--bare] [--git-dir=GIT_DIR] [--work-tree=GIT_WORK_TREE]
    [-c name=value]
    [--help] COMMAND [ARGS]
```

DESCRIPTION

Git is a fast, scalable, distributed revision control system with an unusually rich comm

helpについて

helpの読み方・2

- []で囲まれた引数(オプション)はそれぞれが省略可能であることを示す。

例えば、`[--exec-path[=GIT_EXEC_PATH]]`

はこれ自体が省略可能であり、`--exec-path`だけでもよい。

`=GIT_EXEC_PATH`を書く場合には`-exec-path`を書く

- |(縦棒)で分けられたオプションはいずれかから選択されることを示す。

並列な意味のオプションに対して使われることが多く、

`-p`、`--paginate`、`--no-pager`はいずれか一つが指定されればよい。

※これらは複数を指定されることが想定されない。
(指定するとどれかが優先的に使われたりする)。

参考までに各オプションの説明をする。

`-p`と`--paginate`はコマンド出力にページャプログラムを使うことを指定し、

`--no-pager`はページャを使わないこととする。

- `COMMAND`は最低限必要な引数

※[]で囲まれていないため。

helpについて

用語集 (glossary, terminology)

- ヘルプを読んでいると多くの専門用語を見ます。
 - blob, HEAD, refspec, tree-ish などなど...(※今はこれらの意味を知らなくてよいです)

そんな時は...

- `$ git help gitglossary`

DESCRIPTION

alternate object database

Via the alternates mechanism, a [repository](#) can inherit part of its [object database](#) from database, which is called "alternate".

用語

bare repository

A bare repository is normally an appropriately named [directory](#) with a `.git` suffix tha

説明

ではgitの使い方を学んで行きましょう。

基本的な機能

- (ファイルや変更の)追加
 - `add`
- コミットの作成
 - `commit`
- ログ、差分表示
 - `log`、`diff`
- バージョンの取り出し
 - `checkout`

まずは試してみる・1

レポジトリ作成、ファイルの追加、コミット

- とりあえず何か書きはじめる

```
$ echo 'hi, git!' > scratch
```

(ここではechoコマンドでファイルscratchを作成しました。もちろんエディタで作成してもよいです。以降では簡単のためにechoを使うことがよくあります。)

- gitは急なバージョン管理に応えてくれます。
 - レポジトリを作ります(=gitでバージョン管理を始めるための準備)

```
$ git init
```
 - ファイルを管理対象に追加します

```
$ git add scratch
```
 - コミットしましょう

```
$ git commit
```
 - コミットログを書いたら保存しましょう(utf-8/LFで保存)

- これで一つコミットができました

まずは試してみる・2

変更の追加

- コミットしたファイルを変更することになりました

```
$ echo "what's up?" >> scratch
```

(文字列「what's up?」を追記しました。)

- 変更をコミットしましょう

```
$ git add -u
```

```
$ git commit
```

以上を続けていくのが一番簡単な方法です。

まずは試してみる・3

ログ表示

- ログを確認してみます

```
$ git log
```

```
commit c919fe7e3f5a817b8ae77f1a0c0c88706d0fde97
Author: i_my_me <a@b.c>
Date: Thu May 31 21:59:29 2012 +0900

    変更してみました

commit bbf58bfcc9c4f603bb9931c0c936b3fc17a678e
Author: i_my_me <a@b.c>
Date: Thu May 31 21:58:37 2012 +0900

    最初のコミット
```

コミット名

書いた人

コミット日時

2つ目のコミット

1つ目のコミット

まずは試してみる・4

差分表示

- 1つ目と2つ目のコミットの差分を表示してみます。what's up? が差分になるはずです。

```
$ git diff HEAD~1
```

+から始まる行は
追加されたもの

ちなみに...

- から始まる行は削除行

- 期待通りの差分が表示されました

```
$ git log -p -1
```

とすることもできます。

(diffにコミットログが付加される)

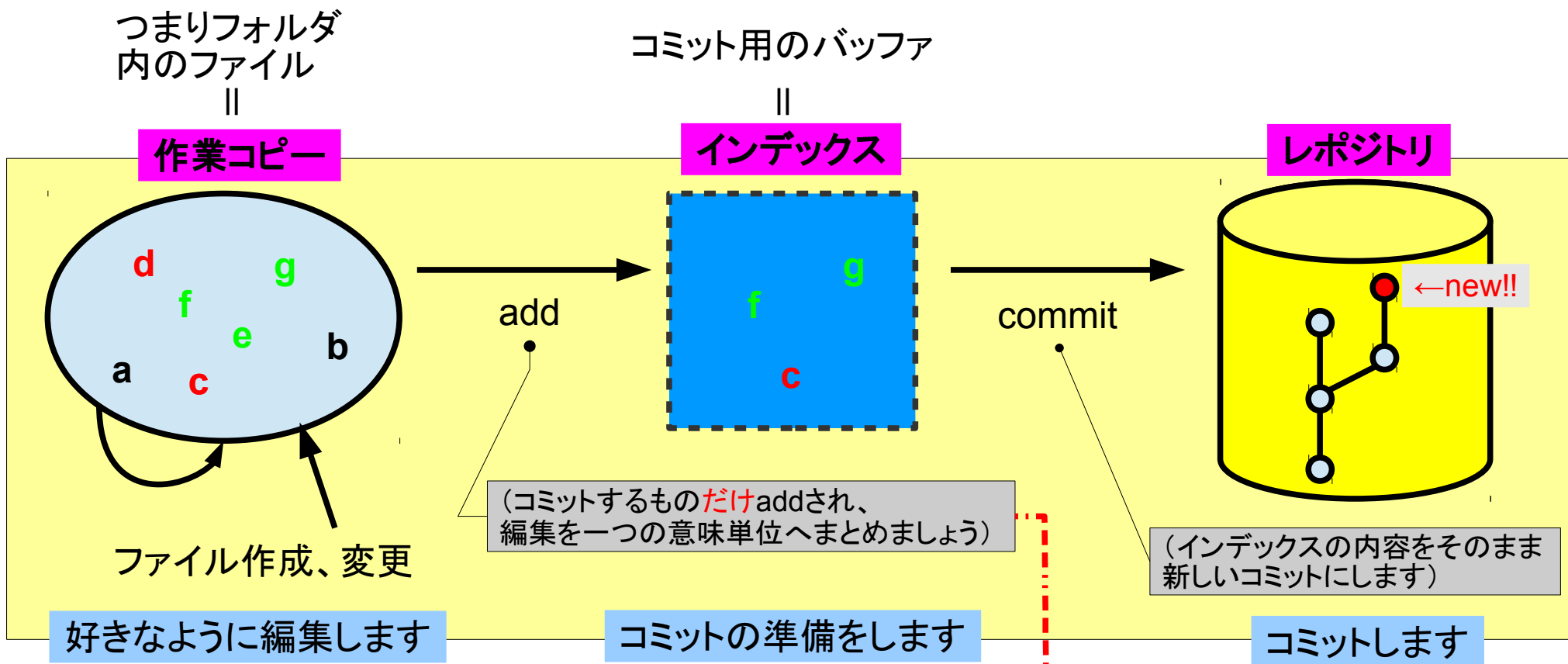
```
Git Bash
$ git diff HEAD~1
diff --git a/scratch b/scratch
index 15ff792..feb7a4b 100644
--- a/scratch
+++ b/scratch
@@ -1,2 @@
 hi, git!
+what's up?
$
```

```
Git Bash
$ git log -p -1
commit c919fe7e3f5a817b8ae77f1a0c0c88706d0fde97
Author: i_my_me <a@b.c>
Date: Thu May 31 21:59:29 2012 +0900

    変更してみました

diff --git a/scratch b/scratch
index 15ff792..feb7a4b 100644
--- a/scratch
+++ b/scratch
@@ -1,2 @@
 hi, git!
+what's up?
$
```

これって何がどうなってこうなるの？・1

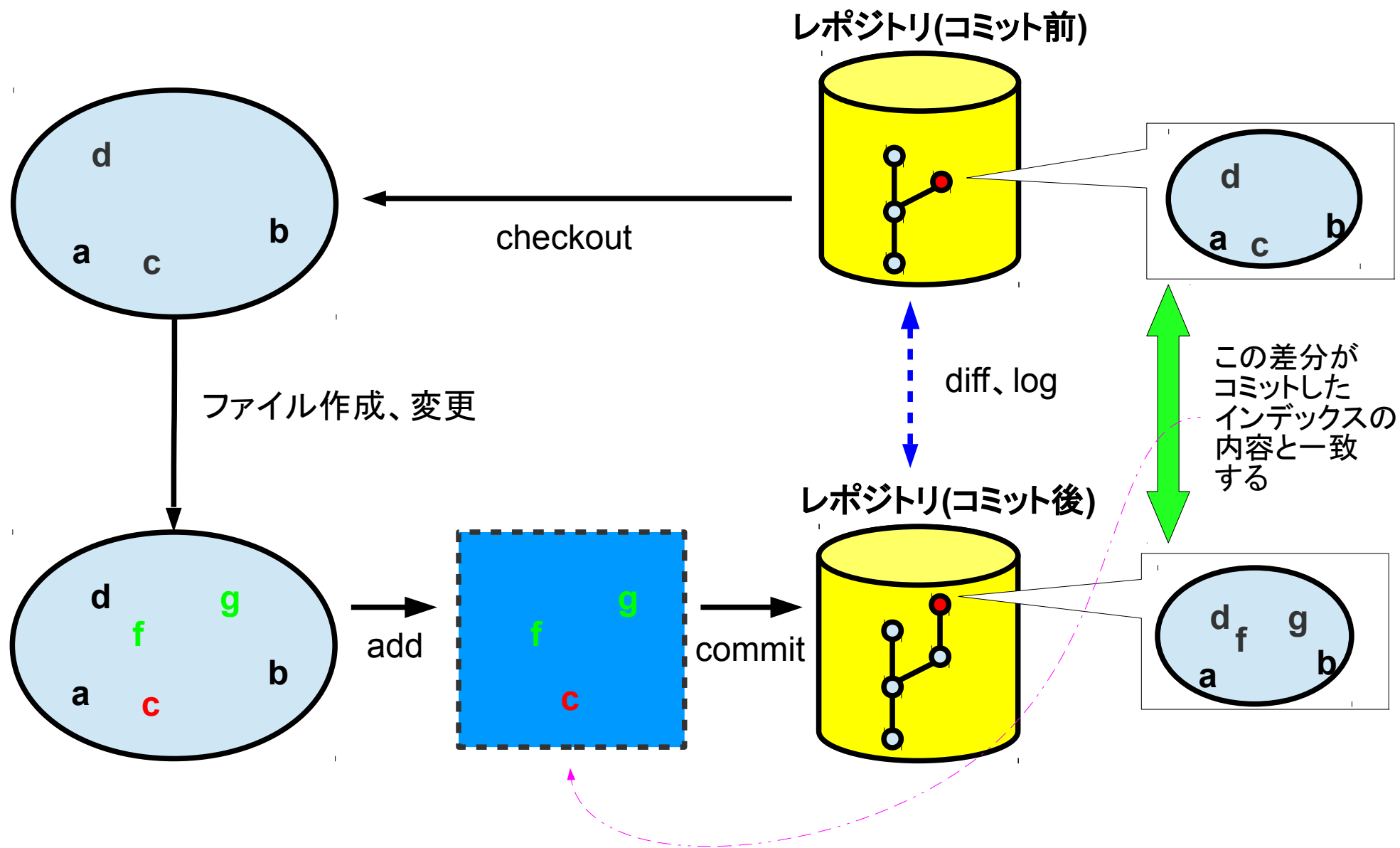


hunkとは差分の部分です。
winmergeを使った時に
表示される1つ1つの差分箇所
のようなものです。
gitが勝手に判別するので
精度が悪い時もありますが
ユーザによる自由選択も可能です。

addする範囲(hunkといいます)を選ぶ場合には、
`git add -p <files>` を使います。

(全ファイルaddしてから `git reset -p <file>` を使って、
不要なhunkを取り除くこともできます。)

これって何がどうなってこうなるの？・2



まずは試してみる・5

バージョンの取り出し・1

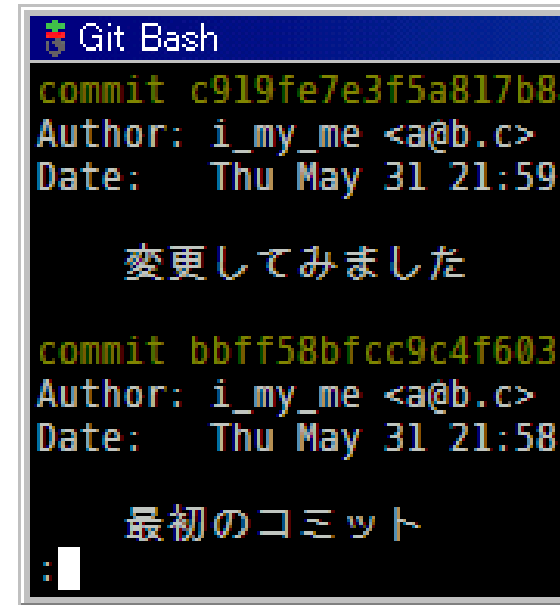
- 最初のコミットを取り出してみます。
 - さっきlogで見たコミット名はbbff58b・・・でした。

```
$ git checkout bbff58b
```

- 取り出すとは作業コピーに反映すること
- scratchの内容を確認してみると
hi, git!になっているはずです。

※もとに戻す場合には・・・？

⇒\$ git checkout master



```
Git Bash
commit c919fe7e3f5a817b8
Author: i_my_me <a@b.c>
Date: Thu May 31 21:59

変更してみました

commit bbff58bfcc9c4f603
Author: i_my_me <a@b.c>
Date: Thu May 31 21:58

最初のコミット
:
```

masterはgitのデフォルトブランチ。svnのtrunkのように、gitのコミットは何かのブランチに属する。これまでの作業は、レポジトリ作成時から存在するmasterブランチ上で作業をしていた。ブランチは履歴の先頭を指すため、c919feを指すmasterを引数に指定することで戻せるのである。
※git checkout c919fe でも可です。

ただし履歴を進める(=コミットする)場合にはブランチをcheckoutしておくこと。(detached HEAD)

まずは試してみる・5

バージョンの取り出し・2

- 作業コピー全体ではなく、ファイルを対象に取り出すこともできます。
次のシナリオを考えてみます。
- まず新しいファイルreadmeをコミットしておく
- 2種類の変更を加える
 - scratchとreadmeに変更を加える

```
$ echo 'modify scratch' >> scratch
```

```
$ echo 'test to checkout' >> readme
```
- readmeだけレポジトリのバージョンに戻してみる

```
$ git checkout readme
```

→scratchは変更されたまま、
readmeはコミットした後の状態になります。

まずは試してみる・5

バージョンの取り出し・3

- 作業コピー(とインデックス)の状態を確認するために `git status` を使うことができます。
- 前ページの①の後では右のようになります。(レポジトリ=作業コピー)

nothing to commit:「コミットするものはありません」

```
Git Bash
$ git status
# On branch topic
nothing to commit (working directory clean)
$
```

- ②の後では
変更したファイルが
表示されます。

```
Git Bash
$ git st
# On branch topic
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme
#       modified:   scratch
#
no changes added to commit (use "git add" and/or "git commit -a")
$
```

ここには、変更されたファイルについての説明が書かれています。

Changed but not updated:「変更されているが、(インデックスが)更新されていないもの」

use **"git add ..."**:「コミットするものを更新するならgit add <file>を使ってください」

use **"git checkout ..."**:「作業コピーの変更を捨てるならgit checkout -- <file>を使ってください」

③ではこれを使ってreadmeの変更を捨てています。
※「捨てる」=「レポジトリの状態へ戻す」

まずは試してみる・5

バージョンの取り出し・4

【補足1】add(インデックスの更新)するとセクションが1つ増えます。

Changed to be committed:「コミット予定の変更」
use "git reset...":
「unstageするならgit reset HEAD <file>を使ってください」

unstageとは、インデックスから作業コピーへ下ろすことを言います。
※stageとはインデックスの別名であり、「stageする」とは「addする」ということです。
「unstageする」はその逆となるわけですね。

```
Git Bash
$ git add scratch
$ git status
# On branch topic
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   scratch
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme
#
$
```

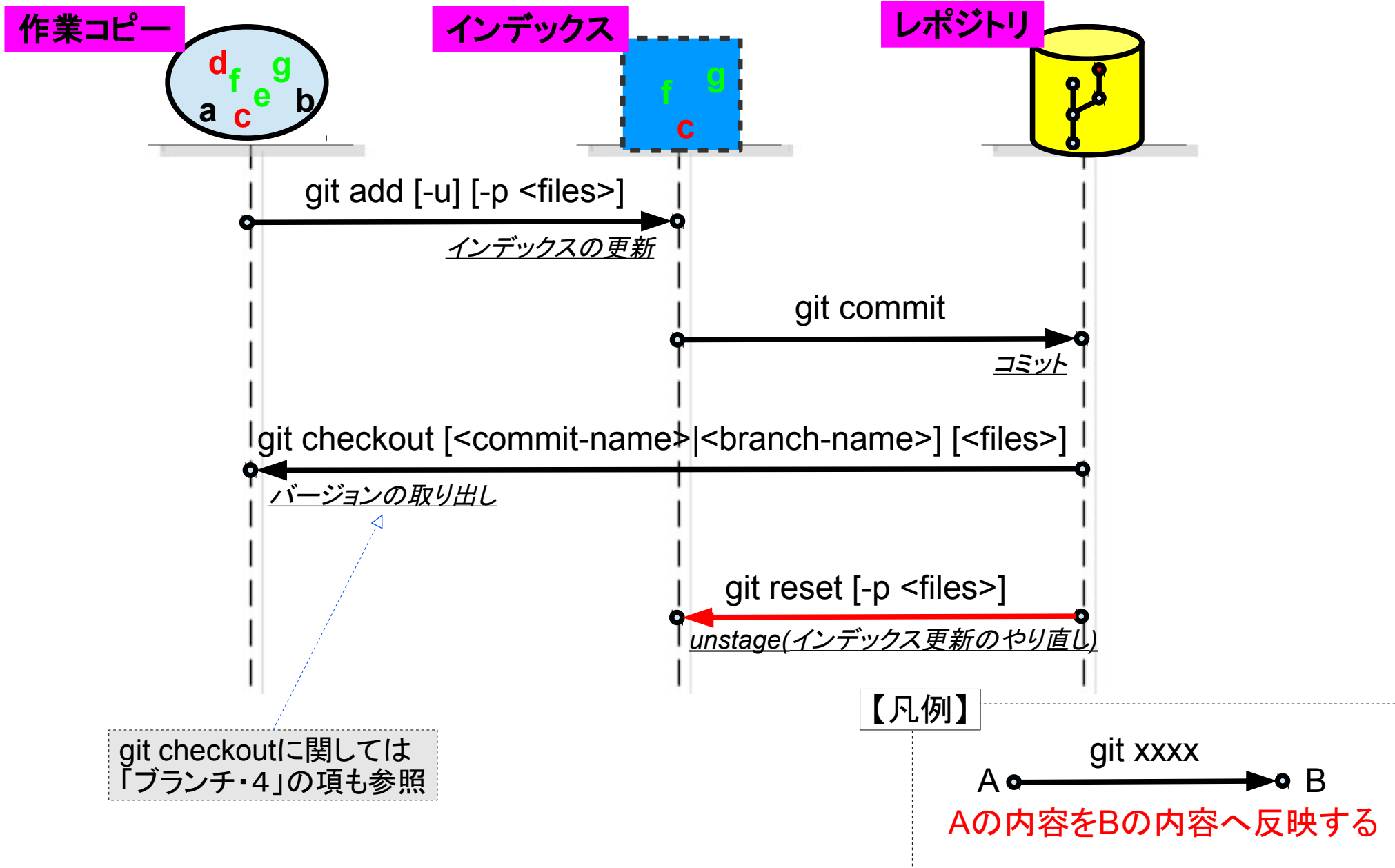
【補足2】版管理されていないファイルがある場合には一番下のセクションに表示されます(ちょうど①の途中でコミットする前)

Untracked files:
「追跡されていないファイル」

```
Git Bash
$ git status
# Not currently on any branch.
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       readme
nothing added to commit but untracked files present (use "git add" to track)
$
```

まとめ①・1

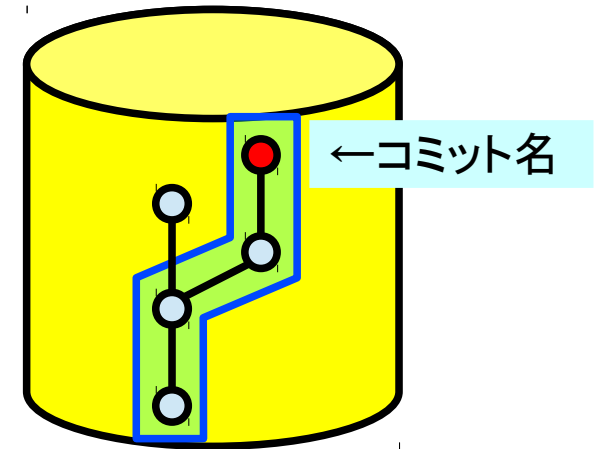
(一回のコミット構築の中でよく使うもの)・基本コマンド



まとめ①・2

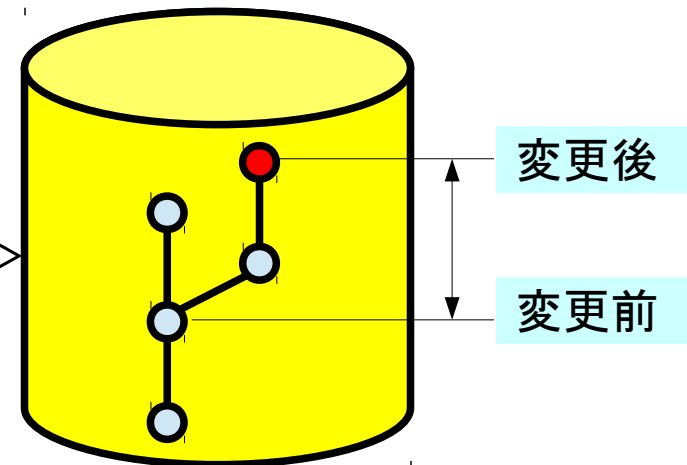
- あるコミットのログを見る

```
$ git log <コミット名>
```



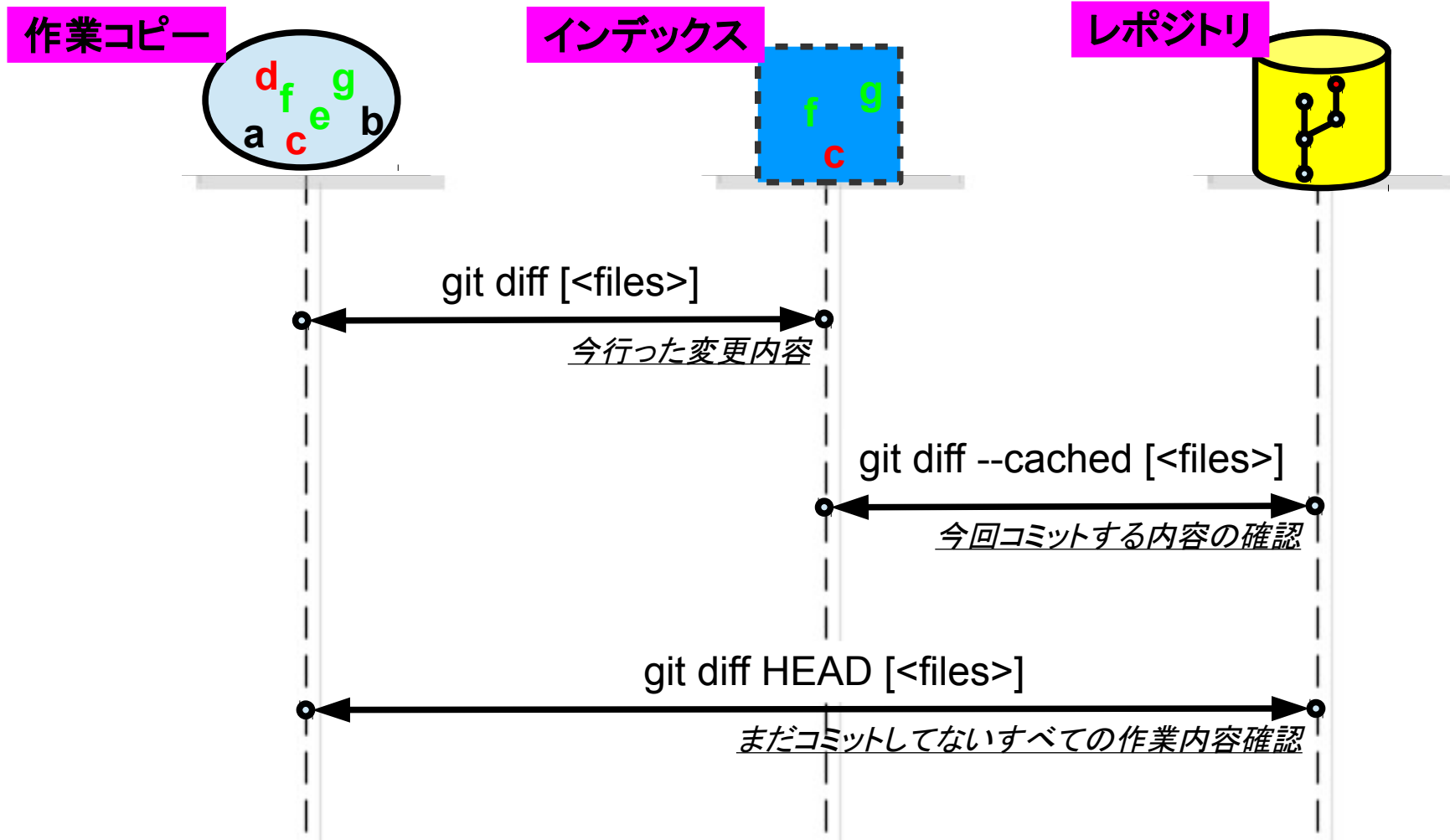
- 2つのコミットの差分を見る

```
$ git diff <変更前> <変更後>
```



まとめ①・3

(一回のコミット構築の中でよく使うもの)・作業確認



まとめ①・(補足)

- これまで説明した3つは、ヘルプではブレがあるので注意。
 - 作業コピー
 - work[ing] tree, working directory, working copy, working repository
 - インデックス
 - index, cache, staging area
 - (ローカル)レポジトリ
 - repository, local repository

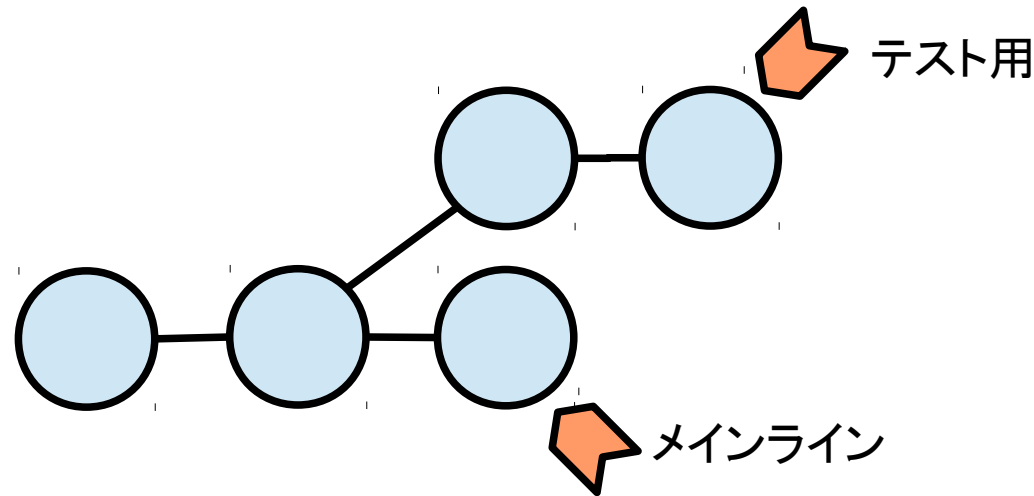
余談・1

コミットログの書き方について

- コードを見れば分かることはできるだけ書かない
- 変更した理由、参照ドキュメントなどを書くとい
- 公式なスタンスとしては下記を参照
[/doc/git/html/user-manual.html#creating-good-commit-messages](https://git-scm.com/doc/git/html/user-manual.html#creating-good-commit-messages)
- 1行目
 - (できれば)50文字以内で要約を書く
 - git logの1行表記に使用される
 - WBSや、バグIDなどを書く
- 2行目
 - **改行のみ** (コミットログを扱う他のコマンドや、外部プログラムが期待します)
- 3行目以降
 - 詳細を書く

ブランチ・1

概要

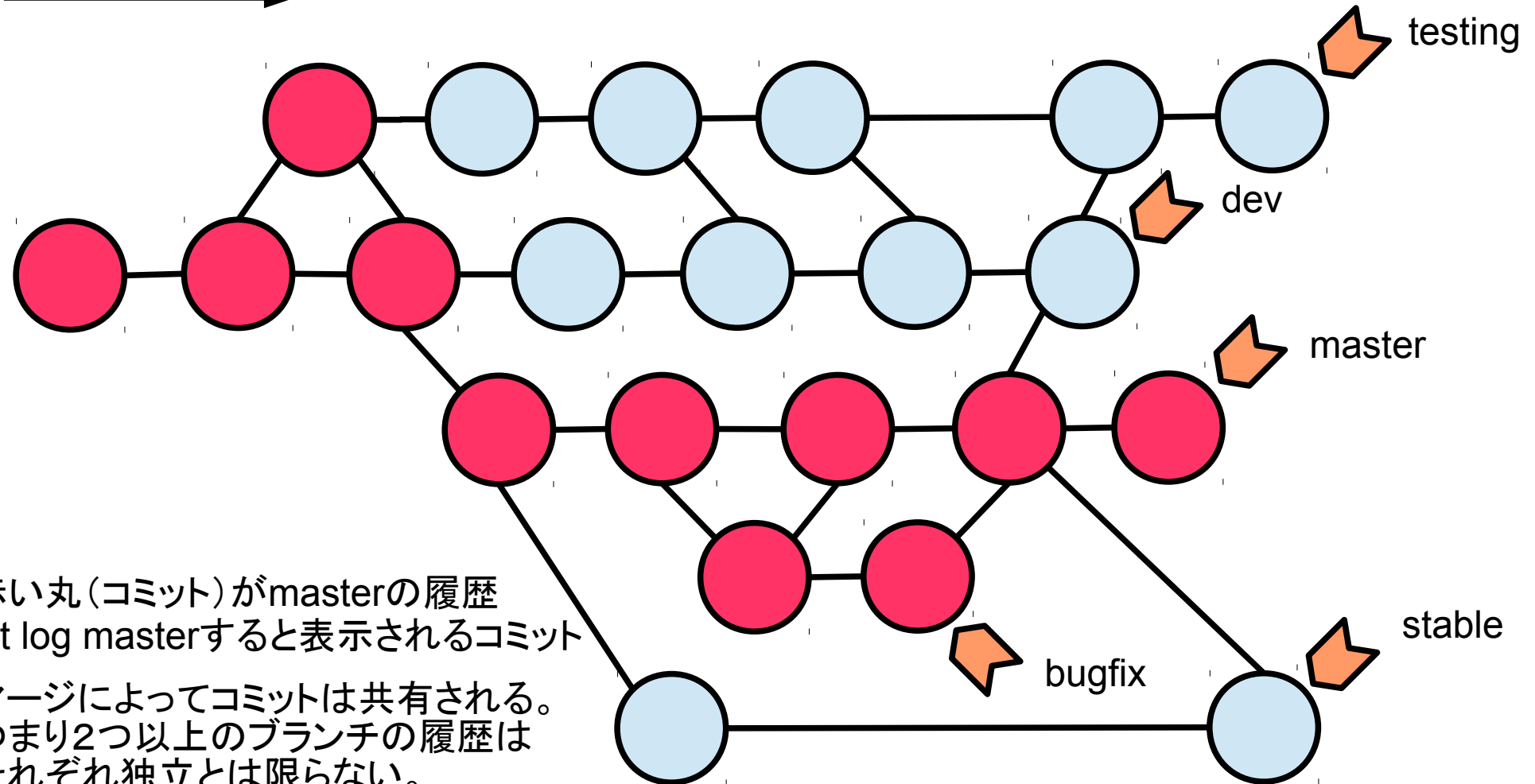


- メインラインとは別の変更を行う場合にブランチを使用する
 - 担当作業毎に分けたい
 - WBSで分けたい
 - ちょっとした動作確認用にテストコードをコミットしておきたい
- メインライン自体もブランチ
 - メインラインとしてmasterという名前がよく使われるが、運用による
(前項ではmasterをデフォルトブランチとして紹介してます
See「まずは試してみる・5」)
- 主な操作
 - ブランチ作成、ブランチ切り替え、マージ

ブランチ・2

運用例・1

時系列順



赤い丸(コミット)がmasterの履歴
git log masterすると表示されるコミット

マージによってコミットは共有される。
つまり2つ以上のブランチの履歴は
それぞれ独立とは限らない。

履歴＝到達可能なコミット群

git flow, GitHub Flow
についても調べてみると
いいです。

運用例としては、下記URLも参照。(リモートレポジトリの知識が必要)
<http://keijinsonyaban.blogspot.jp/2010/10/successful-git-branching-model.html>
ただしより良い運用は、いつも自分が使いやすいものであることを付記しておく。

ブランチ・3

運用例・2

担当機能1

リリース日が決まるまでもう少し寝かせよう

担当1/PG/実装機能1

ブランチ名

担当1/PG/master

担当1のリリースに機能2が必要だからマージしておこう

担当1/PG/実装機能2

ベースのソースファイル

機能確認のために「ベース~1..ベース」の内容が欲しいので、取り入れよう。

担当2/PG/master

担当2/ちょっとしたテスト用

担当機能2

ビルド通らないが担当1の作業を進めたいので一旦コミット

前に使ったテスト用のログとかスタブとか。
担当2のPGでまた使いそうだし残しておこう。
これはリリースするつもりがないので
上位へマージはしないぞ。

ブランチ・4

ブランチの操作・1

- ブランチ作成

```
$ git branch <branchname> <startpoint>
```

※ブランチの切り替えは行われない＝作業コピー変更なし

- ブランチ切り替え

```
$ git checkout <branchname>
```

- branchnameが指すバージョンを取り出している
- checkoutはこれまでコミット名を指定していたが、
実用上はブランチ名を使うことが多い。(detached HEADも参照)

- ブランチ作成とブランチ切り替えは同時に使われることがよくあります。

結局切り替えるのなら一緒にやっちゃいましょう

```
$ git checkout -b <branchname> <startpoint>
```

└ Bbranch

ブランチ・5

ブランチの操作・2

• ブランチ削除

└ Delete

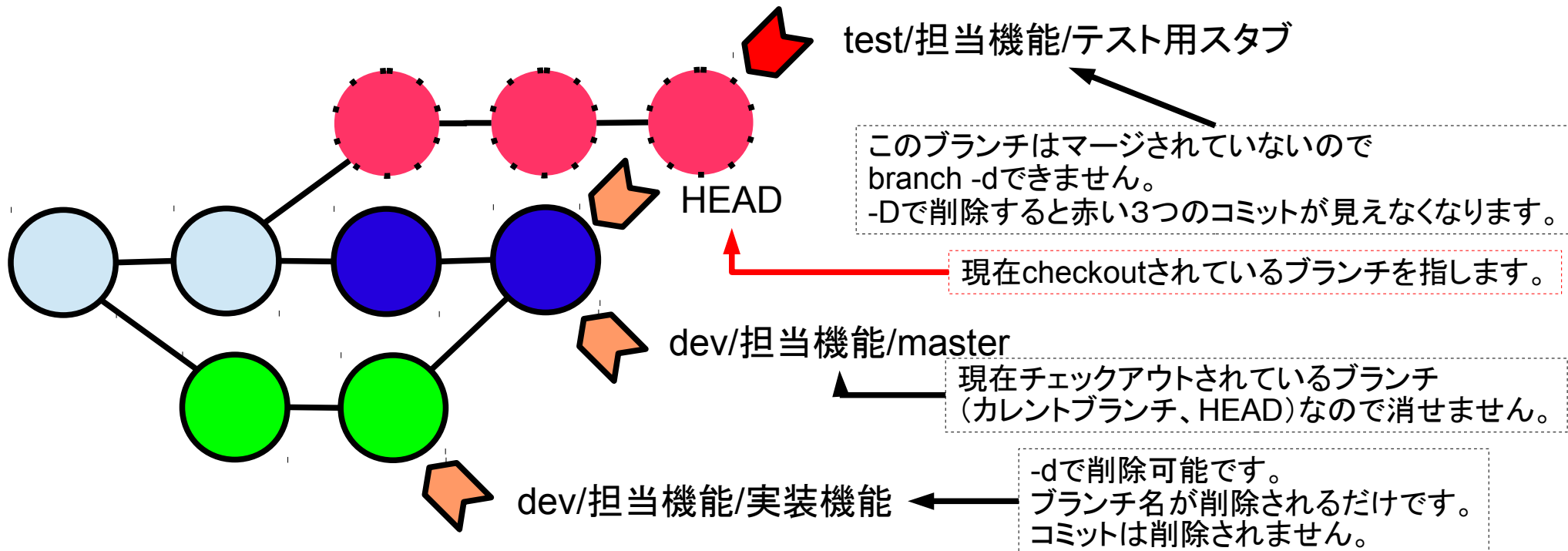
```
$ git branch -d <branchname>
```

- branchnameが現在のブランチにマージされていることが必要(安全策)

```
$ git branch -D <branchname>
```

とするとマージ済みによらず強制削除

どのブランチもそのコミットを参照しない場合には、git logから表示が消える
＝ハンドルできなくなる

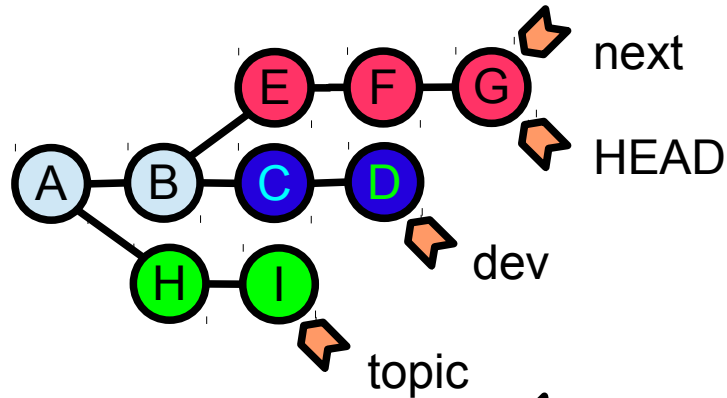


そもそもブランチって？

- ポインタという解釈が適切だと思います。
 - ブランチ名はあるコミットを指しています。
 - 指されているコミットから到達可能なコミット群が履歴を成します。
履歴は副次的なものです。
 - ブランチの実態は下記パスに存在します。
`.git/refs/heads/*`
 - 通常のテキストであり、指しているコミットが書かれています。
 - ブランチ名はファイル名としての使用が前提としてあり、
使用できる文字は限られる。
 - 「/」(スラッシュ)をブランチ名に含めればheads以下にディレクトリが作成される。

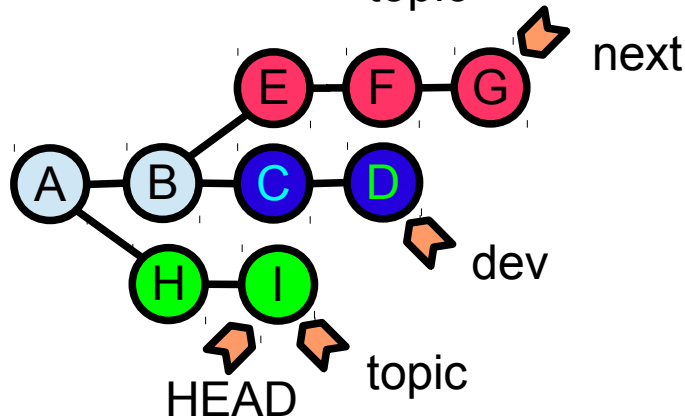
ブランチ・6

HEAD(カレントブランチ)



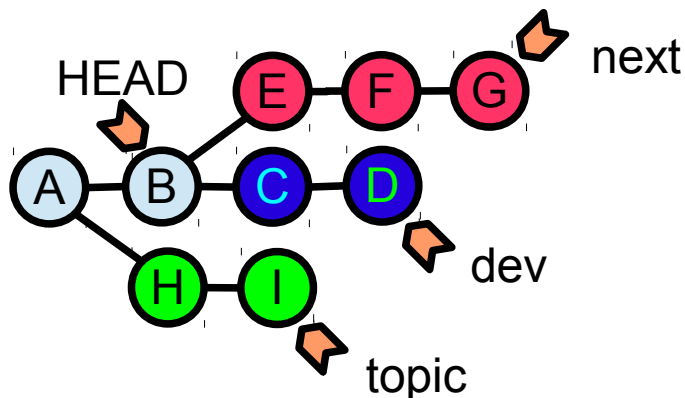
```
$ git checkout next
```

→nextが指すバージョンGを取り出す。
HEADはGを指すようになる。



```
$ git checkout topic
```

→topicが指すバージョンDを取り出す。
HEADはDを指すようになる。



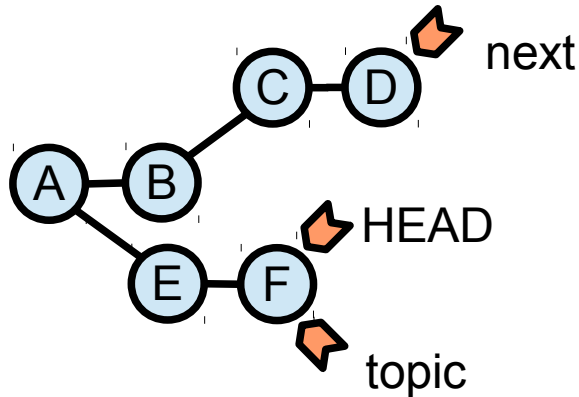
コミット名指定

```
$ git checkout B
```

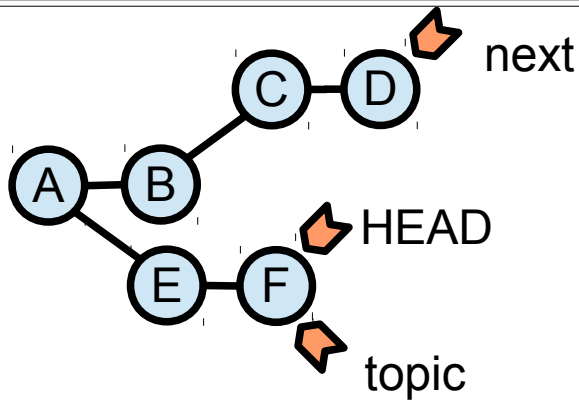
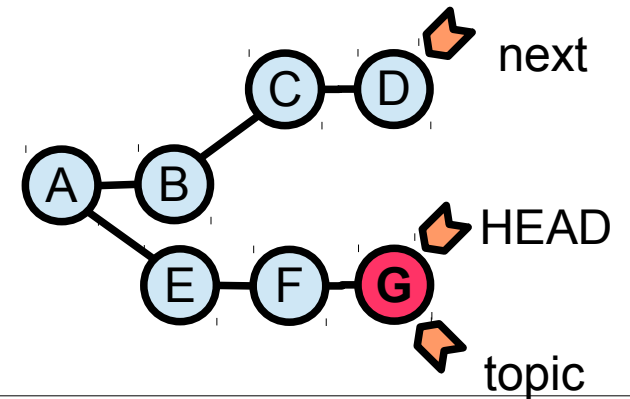
→バージョンB取り出す。
HEADはBを指すようになる。
(detached HEAD)

ブランチ・7

detached HEAD・1



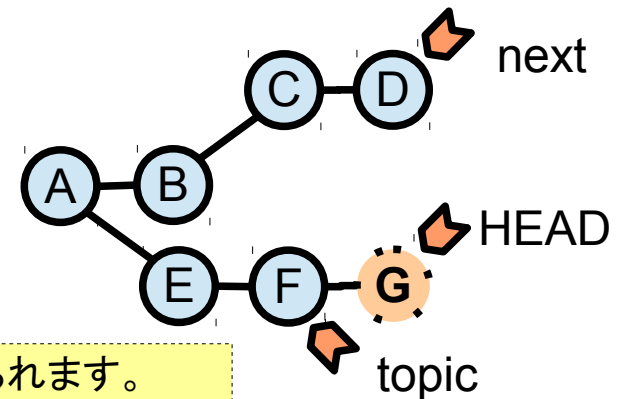
```
$ git checkout topic  
$ edit && git add -u && git commit
```



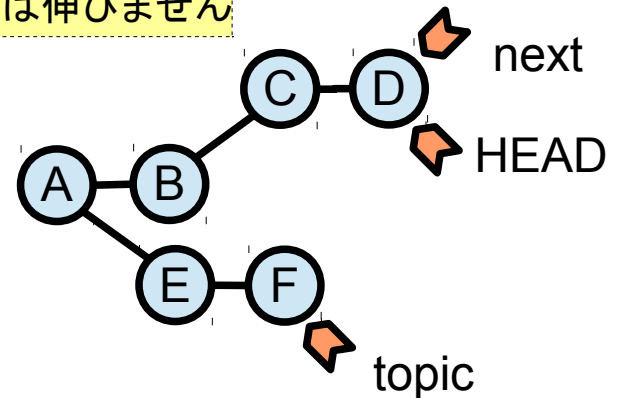
```
$ git checkout F  
$ edit && git add -u && git commit
```

コミット名

Gが作られます。
topicブランチは伸びません



```
$ git checkout next  
Gは見えなくなります
```



ブランチ・8

detached HEAD・2

- 履歴が消えてしまうことから、detached HEADは使いにくいもののよう感じます。
実験的な変更であり、消えても構わないものだけに留めましょう。
(個人的には使ってません。detachedにするくらいなら、適当なブランチを作ってから削除します)

<http://git-scm.com/docs/git-checkout>によると...

--detach

Rather than checking out a branch to work on it, check out a commit for inspection and discardable experiments. This is the default behavior of "git checkout <commit>" when <commit> is not a branch name. See the "DETACHED HEAD" section below for details.

ブランチの利用に関して

ブランチは軽量なものです

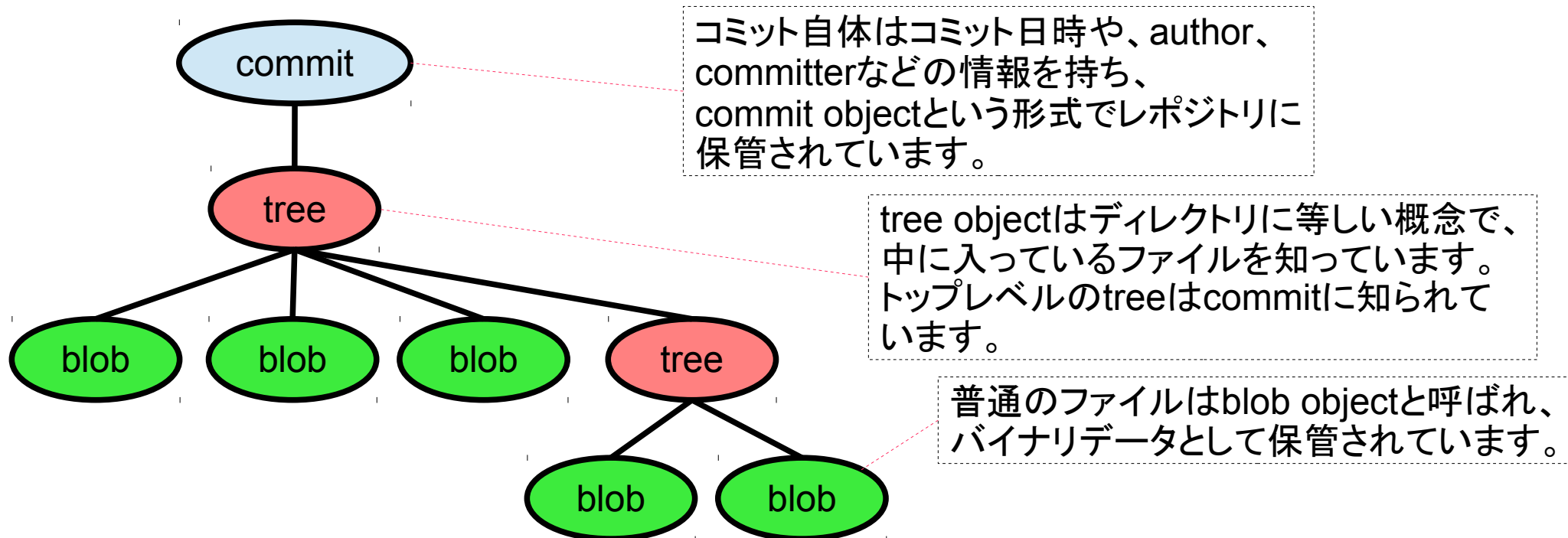
- 躊躇せずどんどん作りましょう。
 - 項「ブランチ・1」ではブランチ利用について、メインラインとは異なる変更を行う場合のように制限されているかのように書いています。
実際はどんな時でもどんどん作って、不要なら削除すればよいだけです。
- ブランチ上で行う変更のイメージが固まっていなくても後で何回もやり直せます。
コミットの分割/圧縮/(他のブランチから)取り出しなんでもできます。
 - # 遅くなればなるほどやり直しは面倒になりますが...
 - (後で述べる) **pushしたブランチはやり直してはダメです!**
- 各ブランチは1つのテーマに関する変更だけをまとめるとマージが容易になります。
 - **topic branch**
 - コード1行が1つの意味をもっていれば、各ブランチの変更は衝突しないでしょう...⇒マージコンフリクトに関係

このページのコマンドは覚える必要はありません

余談・2

gitの内部について・1

- gitは何をどうやって管理しているのでしょうか？
そうするとどんな良いことがあるのでしょうか？
- 1つのコミットは下図の構成になっています。



このページのコマンドは覚える必要はありません

余談・2

gitの内部について・2 (commit object)

- これら3つのobjectは `.git/objects` の中に統一的に管理されています。
- 1つのコミットから管理しているものを調べることができます。
今、調べたいコミット名が `5ab11b` である時、
`commit object` の内容を見ることができます。

```
Git Bash
$ git cat-file -p 5ab11b
tree c3924378d901493df8b5d2820d54c4556044da56
parent f216e87955157229decce2185c92ff2226ad5592
author i_my_me <a@b.c> 1340202997 +0900
committer 1340203005 +0900
encoding sjis

down 霧入 蛭・
$
```

c39243という名前のtree objectを管理していることが分かります。

※到達可能なコミット(parent)も知っているようですね。

このページのコマンドは覚える必要はありません

余談・2

gitの内部について・3 (tree object)

- 5ab11bが管理しているトップレベルのtreeを調べてみましょう。
このtree objectの名前はc39243でした。
(前ページのcommit objectの内容を見てね)

```
Git Bash
$ git cat-file -p c39243
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    readme
100644 blob 6ef41d00c24db486c31357331c242cbd536e9c86    scratch
$
```

2つのblob objectが
管理されています。

- ディレクトリを管理しているときには
下図のようになります。

treeの中に
treeがあります。

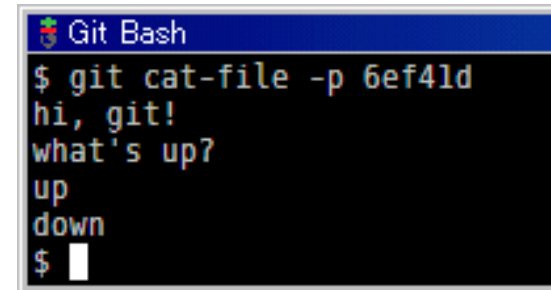
```
Git Bash
$ git cat-file -p 753d
040000 tree 64e88e279ea718b1a964458ff77b7f2dc65b7706    dir
100644 blob 3c32bdc88e7229ebae73bfc79f4a3938b2579ed    readme
100644 blob 6ef41d00c24db486c31357331c242cbd536e9c86    scratch
$ git cat-file -p 64e88e
100644 blob 2e394e7f94196f2d3c0edddcd33ffdd49178338a    tmp
$
```

このページのコマンドは覚える必要はありません

余談・2

gitの内部について・4 (blob object)

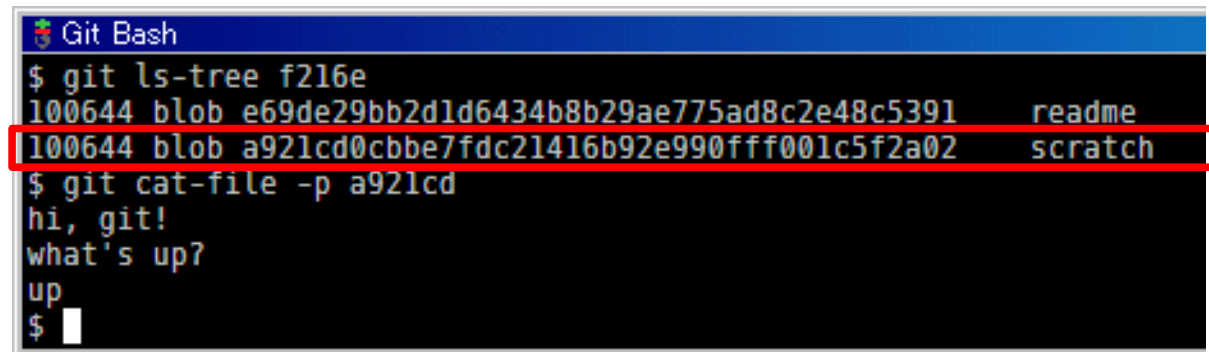
- 最後はblob objectの内容です。



```
Git Bash
$ git cat-file -p 6ef41d
hi, git!
what's up?
up
down
$
```

管理しているファイル(scratch; 6ef41d)を読むことができました。

- 少し古いコミットで管理されているscratchを見えます。



```
Git Bash
$ git ls-tree f216e
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    readme
100644 blob a921cd0cbb7fd021416b92e990fff001c5f2a02    scratch
$ git cat-file -p a921cd
hi, git!
what's up?
up
$
```

- 対して、readmeの方は名前がe69de2のまま変わりません。(2つのコミットの間で変更していません)

余談・2

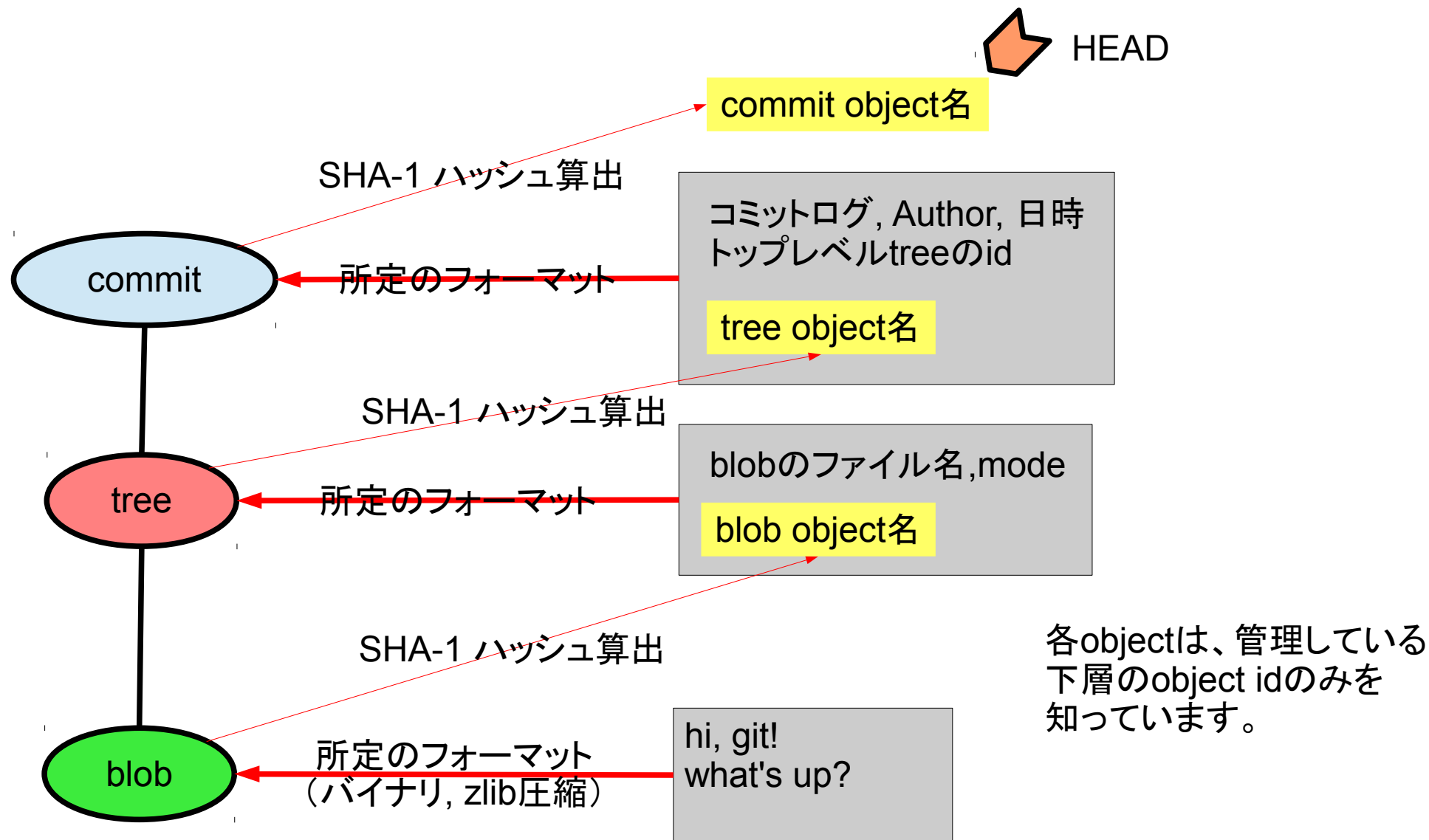
gitの内部について・5(内容追跡・1)

- 変更されたファイルはobject名が変わり、変更されていないものは同じobject名が使われます。object名とは何なのでしょう？
⇒object名は内容の要約(digest)です。
 - gitは各コミットで管理する内容を扱うために、object名を使います。
 - ファイルの内容をSHA-1で要約したものがobject名になります。
 - SHA-1とはハッシュ関数で、ある長さのバイト列を固定バイト列(160bit; 40文字)へ圧縮します。
SHA-1は衝突する可能性が低いハッシュ関数として知られています。gitでは、ファイル内容をobject名へ圧縮するために利用されています。

このページのコマンドは覚える必要はありません

余談・2

gitの内部について・6(内容追跡・2)

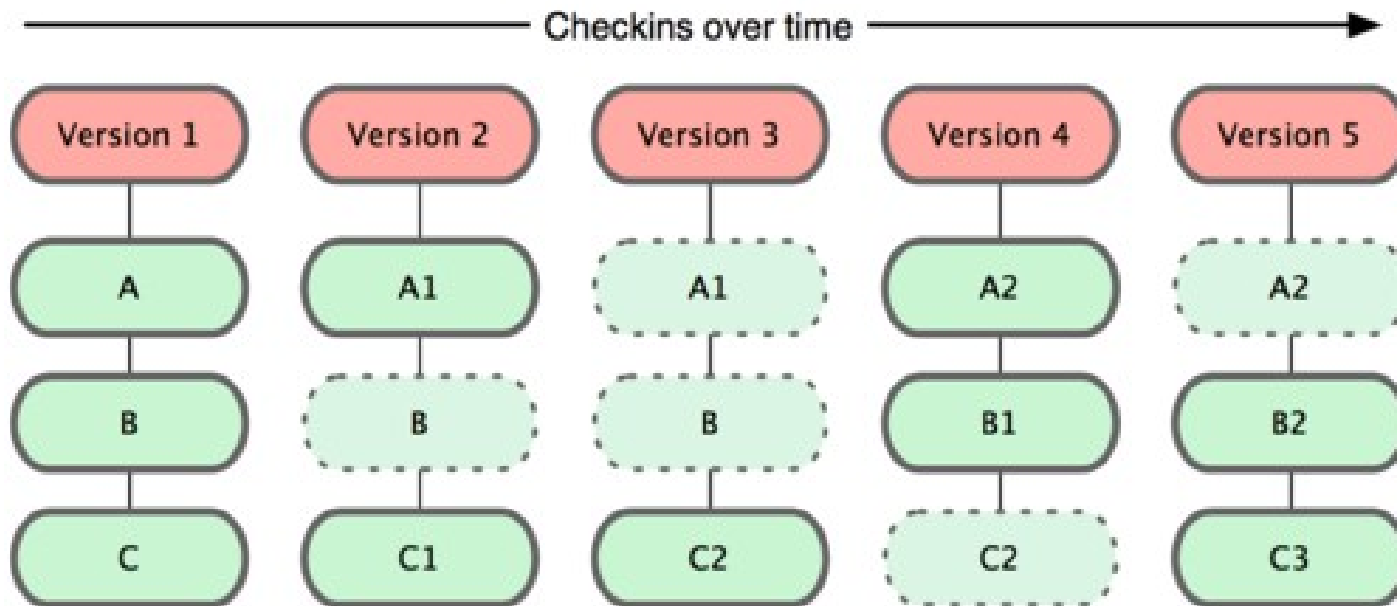


このページのコマンドは覚える必要はありません

余談・2

gitの内部について・7 (内容追跡・3)

- 各コミットが管理するtreeやblobは、内容がSHA-1ハッシュとして間接的に管理されます。コミット間で変更がなければ、実際のblobを使いまわします。(変更されたファイルだけobject名が変わった理由です)



余談・2

gitの内部について・8(内容追跡・4)

- object名は内容の概要なので、
内容の比較にはSHA-1ハッシュの比較だけで
十分です。
- 同じ内容なら同じobject名が算出されるため、
ファイル名変更の検知を行うことができます。
- (サイズが40文字しかないのにファイル内容を示してい
る)SHA-1ハッシュを使うことで、diffやlogなどで、
必要なファイルに効率よく、変更やファイルを見つけたり
することが可能です。
これがgitが速いと言われる所以かもしれませんね。

ファイル名変更の様子

```
Git Bash
$ git status
# On branch topic
nothing to commit (working directory clean)
$ mv readme ChangeLog
$ git status
# On branch topic
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    readme
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       ChangeLog
no changes added to commit (use "git add" and/or "git commit -a")
$ git add ChangeLog && git add -u
$ git status
# On branch topic
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    readme -> ChangeLog
#
$
```

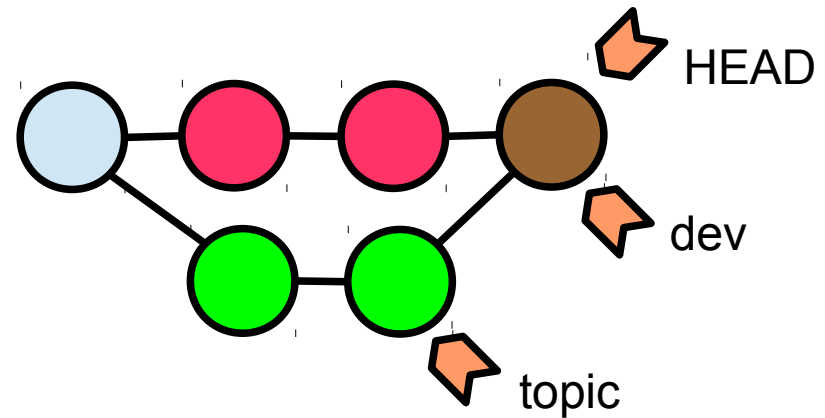
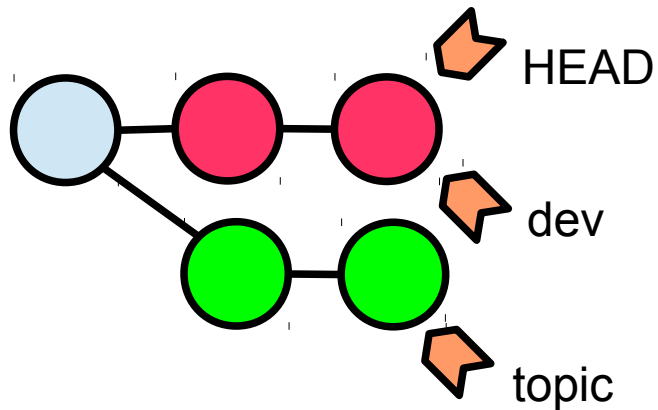
上記をコミットしてから、
コミット前との比較をしてみると
右図となります。

-Mオプション:(変更検知)

```
Git Bash
$ git diff -M HEAD^
diff --git a/readme b/ChangeLog
similarity index 100%
rename from readme
rename to ChangeLog
$
```

マージ・1

概要

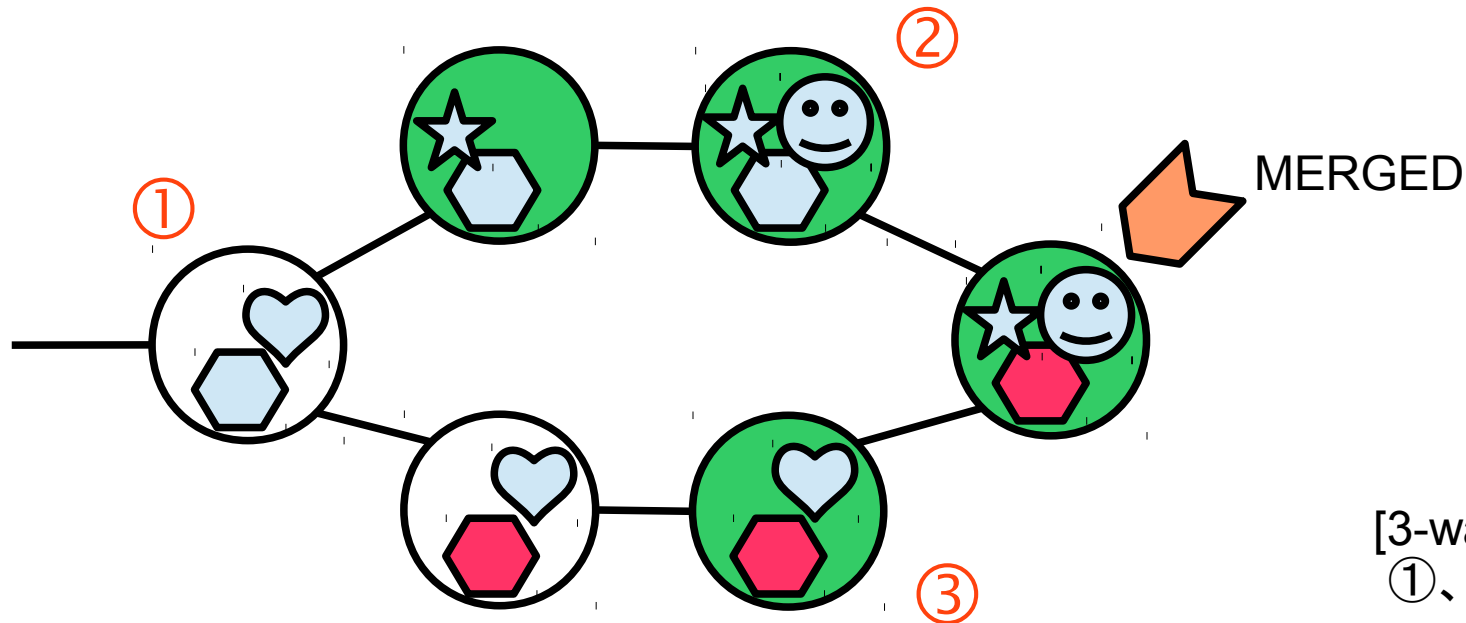


- 複数の変更を統合する操作のこと
 - マージする側のブランチの変更内容をマージされる側へ取り入れる
 - 各ブランチのトピックを統合する
- 関係する話題
 - コンフリクト、merge-base
- HEADがdevの時にtopicをマージ

```
$ git merge topic
```

マージ・2

何がマージされるの？・1

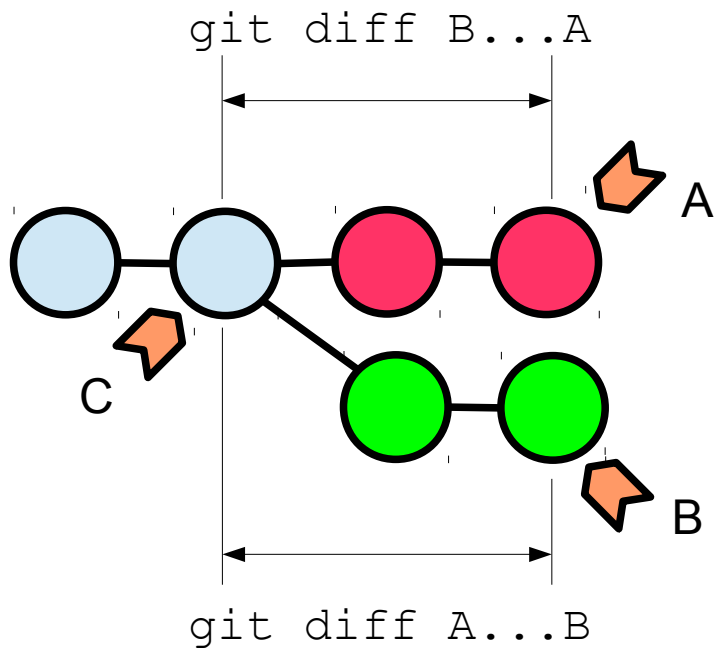


[3-way merge]
①、②、③の内容のみ使う

- 各ブランチの変更を尊重します
 - 2つのブランチで共に同じ変更⇒採用
 - 一方は変更、他方は変更していない⇒異論なし、変更を採用
 - 2つのブランチで意見が衝突⇒コンフリクト

マージ・3

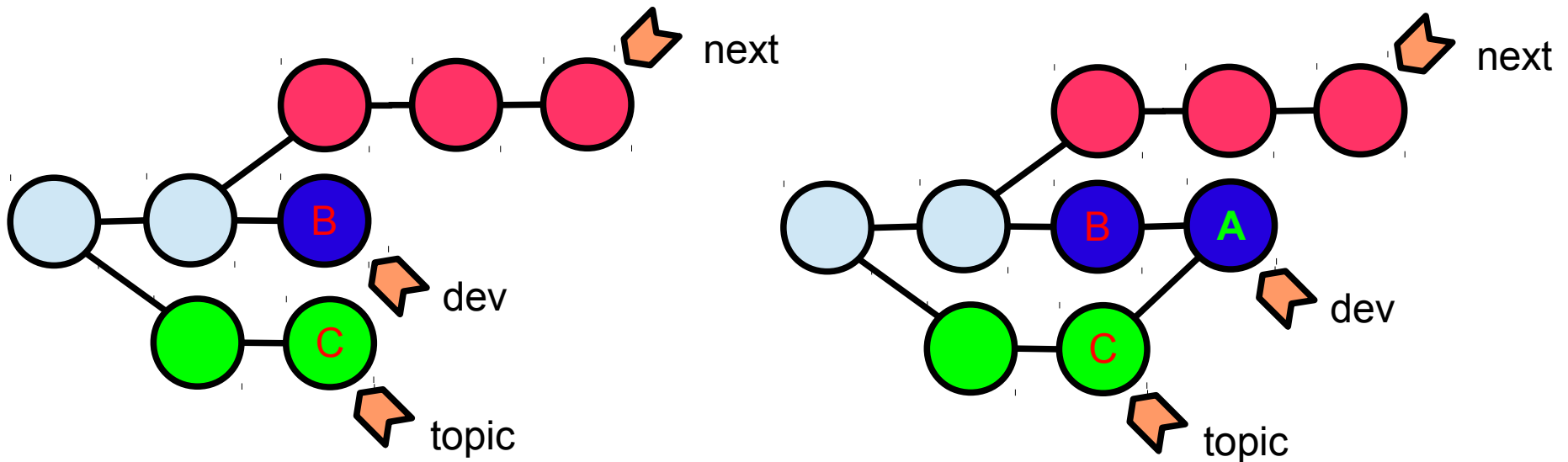
何がマージされるの？・2



- C : 共通祖先(Common ancestor)
- `git diff A...B`
 - ⇔ `git diff $(git merge-base A B) B`
 - ⇔ `git diff C B`

マージ・4

(no-fast-forward)



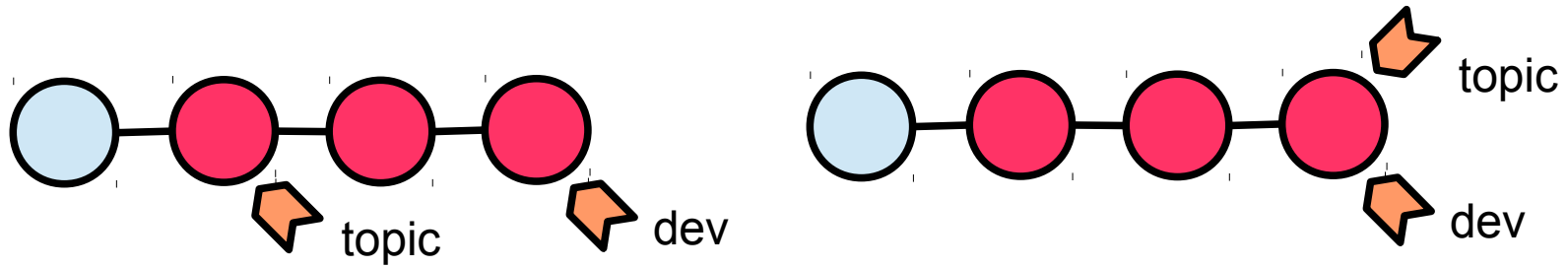
```
$ git checkout dev && git merge topic
```

マージコミットAがコミットされました

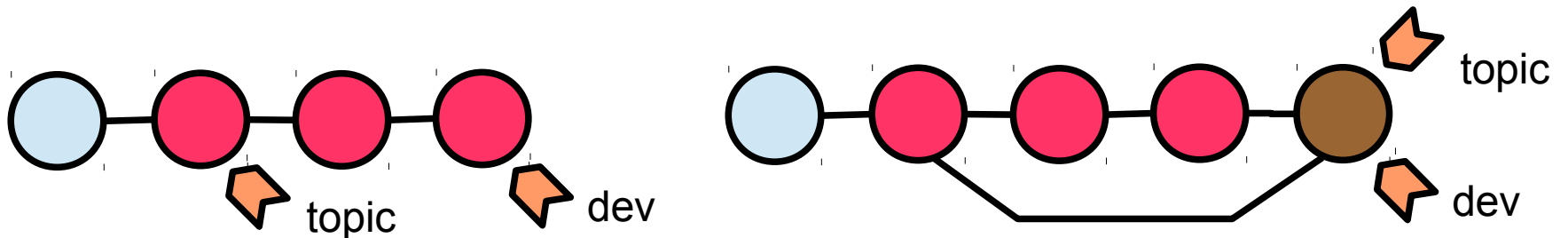
- マージされた後の内容は、
BやCとは異なるためAは存在する
- これはno-fast-forward

マージ・5

fast-forward



- 履歴の共有
 - fast-forward(, already up to date)
- 強制的にno-fast-forwardにすることもできます

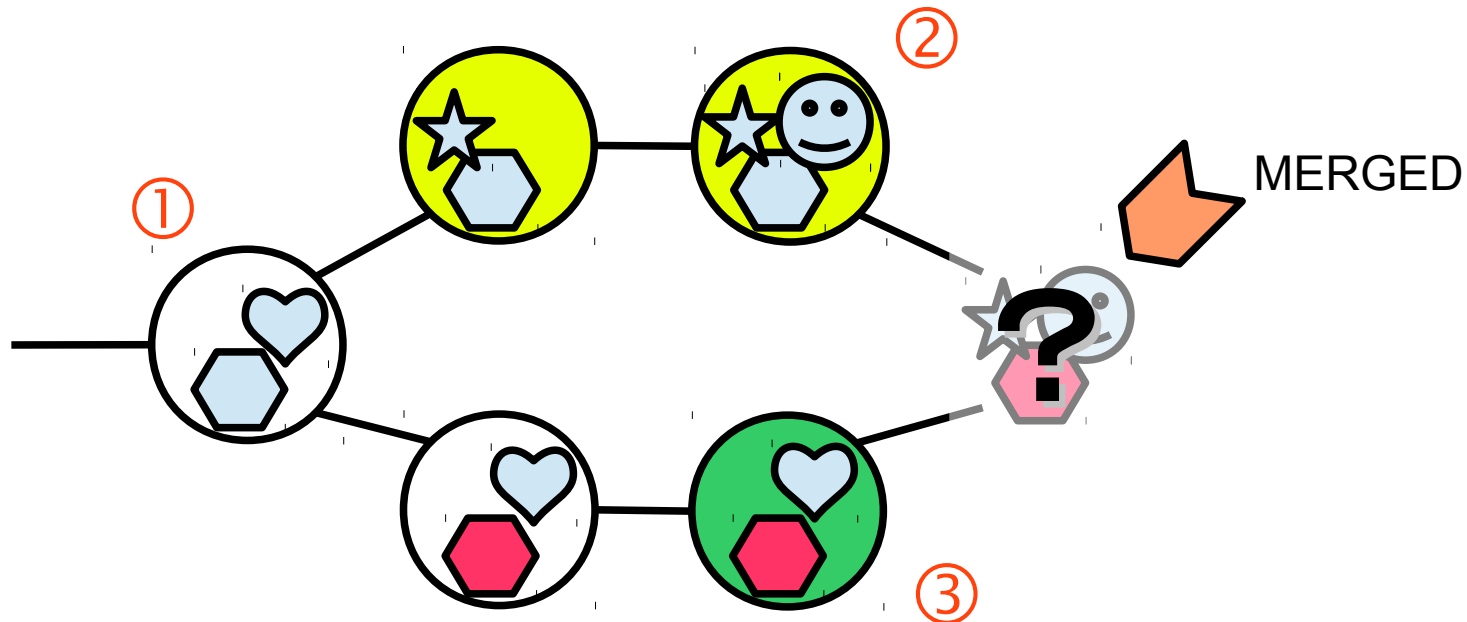


```
$ git merge --no-ff <commit>...
```

- 使い分けは・・・？

マージが失敗した！ どうしよう！ ・1

- どのような時に起きるのか
 - どちらの変更を採用していいかわからない。
同じ箇所に別の変更をしている。



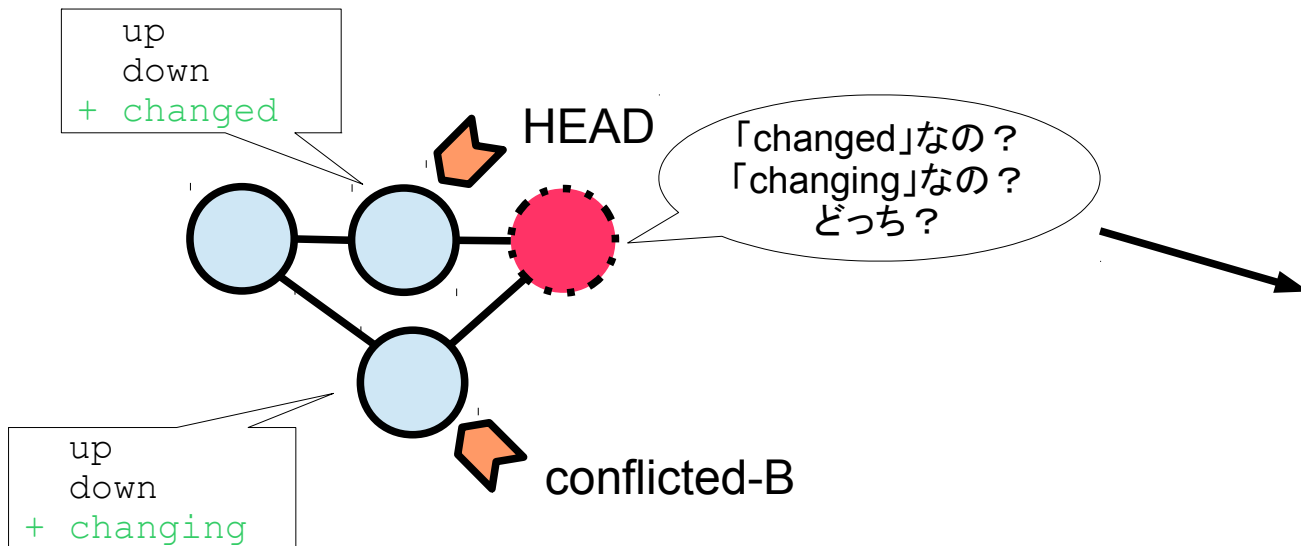
マージが失敗した！ どうしよう！ ・2

- コンフリクト解決の流れ
 - コンフリクトした内容を確認する。
 - コンフリクトマーカ
 - `git diff`(コンフリクト中は表示が変わる)
 - コンフリクト解決
 - マージ後の内容を決定する
 - マージ終了
 - マージコミットのために`add`、`commit`する
 - やっぱりマージやめよう
 - `git merge --abort` か `git reset --merge`

マージが失敗した！ どうしよう！ ・3

コンフリクトマーカ

- コンフリクトしている場所の検出



ファイルscratchの内容

```
hi, git!  
what's up?  
up  
down  
<<<<<< HEAD  
changed  
=====  
changing  
>>>>>> conflicted-B
```

Unmerged paths....:

「未マージのファイル:
(マージが)解決したらgit add(もしくはrm) <file>
を使ってください。」

```
Git Bash  
$ git status  
# On branch conflicted-A  
# Unmerged paths:  
#   (use "git add/rm <file>..." as appropriate to mark resolution)  
#  
#       both modified:   scratch  
#  
no changes added to commit (use "git add" and/or "git commit -a")  
$
```

diffの表示が変わる

```
Git Bash  
$ git diff  
diff --cc scratch  
index 384e504,2388418..0000000  
--- a/scratch  
+++ b/scratch  
@@@ -2,4 -2,4 +2,8 @@@ hi, git  
what's up?  
up  
down  
++<<<<<< HEAD  
++changed  
++=====  
++changing  
++>>>>>> conflicted-B  
$
```

マージが失敗した！ どうしよう！ ・4

コンフリクトの解決、マージ終了

- コンフリクトマーカの位置を参考にマージ後の内容を作ります。
- コンフリクト解決したらadd

```
Git Bash
$ git add -u && git status
# On branch conflicted-A
# Changes to be committed:
#
#       modified:   scratch
#
$
```

ファイルscratchの内容

```
hi, git!
what's up?
up
down
changed
changing
coffee
```

- ログを見てみるとコンフリクトしたことが書かれます。
(git commit時に最初から入っています。消してもよいです)

```
Git Bash
commit 48a45a781ba735fd563cc22c1aa430fdd582e309
Merge: d267498 7694848
Author:
Date:   Sun Jul 8 13:24:37 2012 +0900

    Merge branch 'conflicted-B' into conflicted-A

    Conflicts:
        scratch
:
```

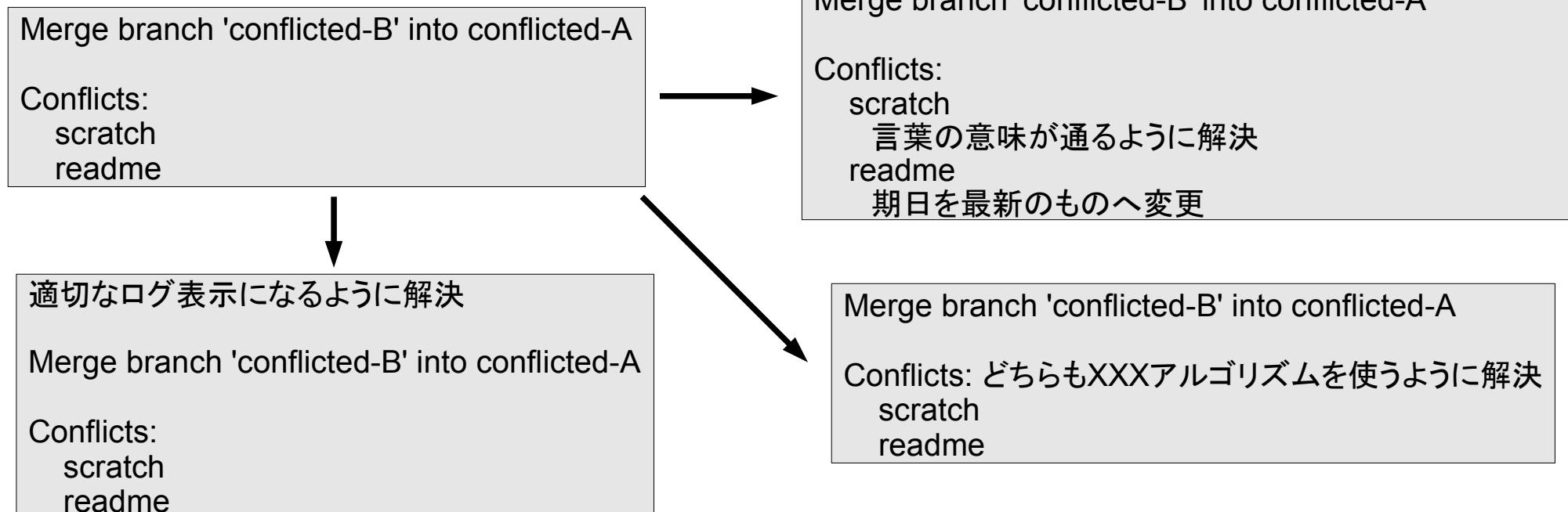
```
Git Bash
$ git diff
diff --cc scratch
index 384e504,2388418..0000000
--- a/scratch
+++ b/scratch
@@@ -2,4 -2,4 +2,6 @@@ hi, git
    what's up?
    up
    down
+changed
+ changing
++coffee
$
```

マージが失敗した！ どうしよう！ ・5

解決時のコミットログの書き方

- コミットログ（デフォルトを使わなくてもよい）
 - 運用が決まっていればその通りにすればよい。
（例えば1行目にはWBSを入れるなど。22ページの余談・1も参照）
 - オススメのログ
 - どのようにコンフリクト解消したかを明記するといいかも。

デフォルトで用意される元のコミットメッセージ



余談・3

gitrevisions

```
$ git help gitrevisions
```

- `<name>~n`

n個前のコミット(親コミット)

「`<name>~`」 ⇔ 「`<name>~1`」、 「`<name>~~`」 ⇔ 「`<name>~2`」

- `<name>^n`

n個目の親(マージで親が複数ある場合の選択方法)

「`<name>^`」 ⇔ 「`<name>^1`」、 「`<name>^^`」 ⇔ 「`<name>~2`」

※親の順番を決めるルールはよく分かりません...

マージ先(元々カレントブランチだった方)が1番目、

マージ元(`git merge`に指定したブランチ側)が2番目になる気がします。

`<name>`

- コミット名、ブランチ名、タグ名などのコミットを指すことができるもの

gitって何がいいの？

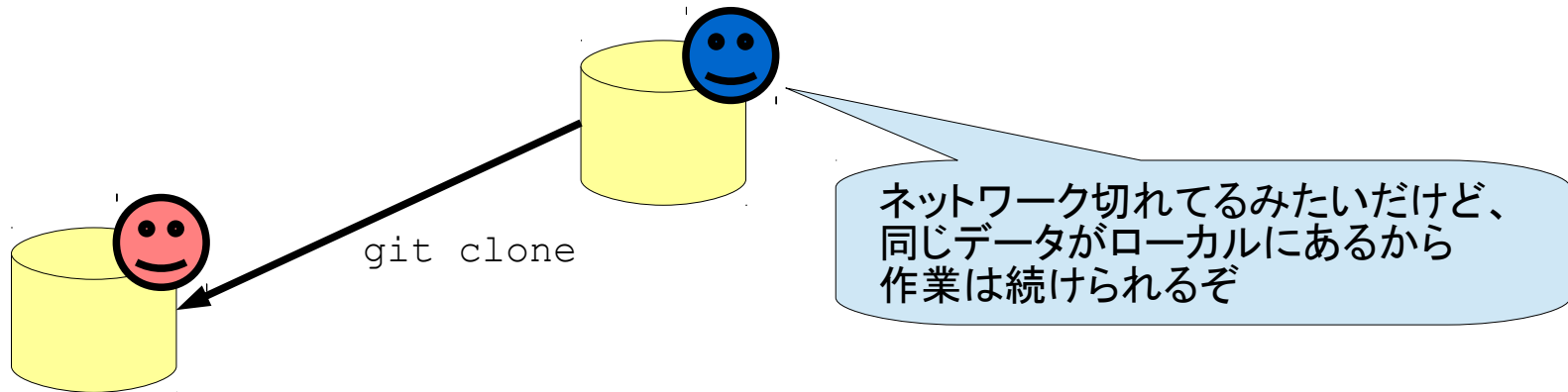
- 流行にのりたい
- 速いらしい
- マージ戦略がシャープらしい
- 簡単に色々やり直せるらしい
- 急な要求にも応えてくれるらしい
- サーバが落ちてても使えるらしい
 - 現状の構成ではこれが大事な点

なぜサーバが落ちてても使えるのか

- 個々の開発者がレポジトリをそれぞれ持つため。
 - 開発者は、svn等の他VCSで行うチェックアウトのタイミングで、レポジトリを複製(クローン)する

```
$ git clone <url> <path>
```

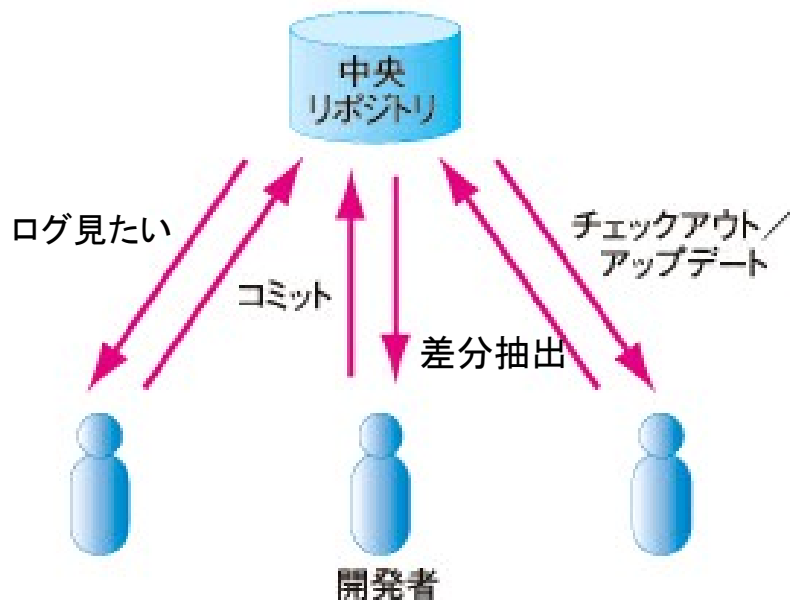
⇒レポジトリ<url>を<path>として複製する



個々の開発者がレポジトリをそれぞれ持つ方式を**分散型**と言います。

集中型と分散型

集中型



【メリット】

- ・分散型に比べると管理者側にメリットがある。
(開発者のパーミッションなど)
- ・履歴の流れが読みやすい。
- ・開発者は同期が取れる。
(取り消しされた履歴に強い)

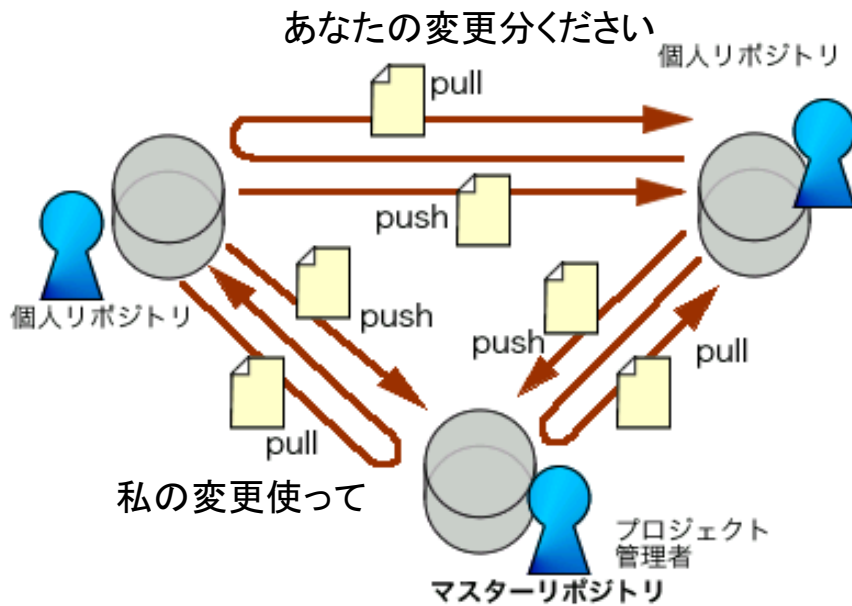
【デメリット】(ソフトウェアによるかもしれない)

- ・中央レポジトリがないと、ファイルの修正しかできない。
- ・他の人が何をやっているか分からない
(一旦コミットしないとテストに展開できない)
- ・コミットして初めてコンフリクトに気づく
(ロックなんて使う人いない)

※このページの「コミット」はsvnを想像してください

集中型と分散型

分散型



gitは非常に柔軟で、レポジトリの性質は運用によって決められる。

【メリット】

- ・開発者が自主的に作業可能
(ネットワークで隔てられていたり、**ダウンしていても作業可能**)
- ・必要に応じて集中型として使える
(柔軟性が高い)
- ・開発工程によって作業レポジトリを分離させられる
- ・(gitは)やり直しが可能
pushしなければ不要なコミットを削げる

【デメリット】

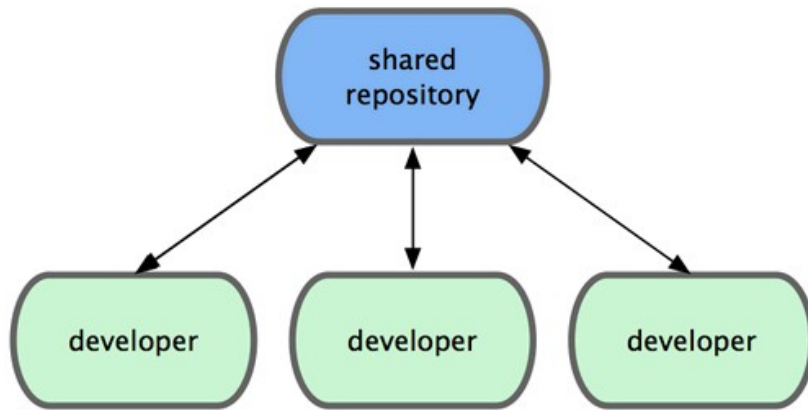
- ・個々のホストが複製されたレポジトリを持つため、
リソースの消費が激しい
- ・運用の手間があるかもしれない
- ・自由すぎて運用を決めないとカオス

・・・あまり思いつかない

分散型の柔軟性

種々の運用方式

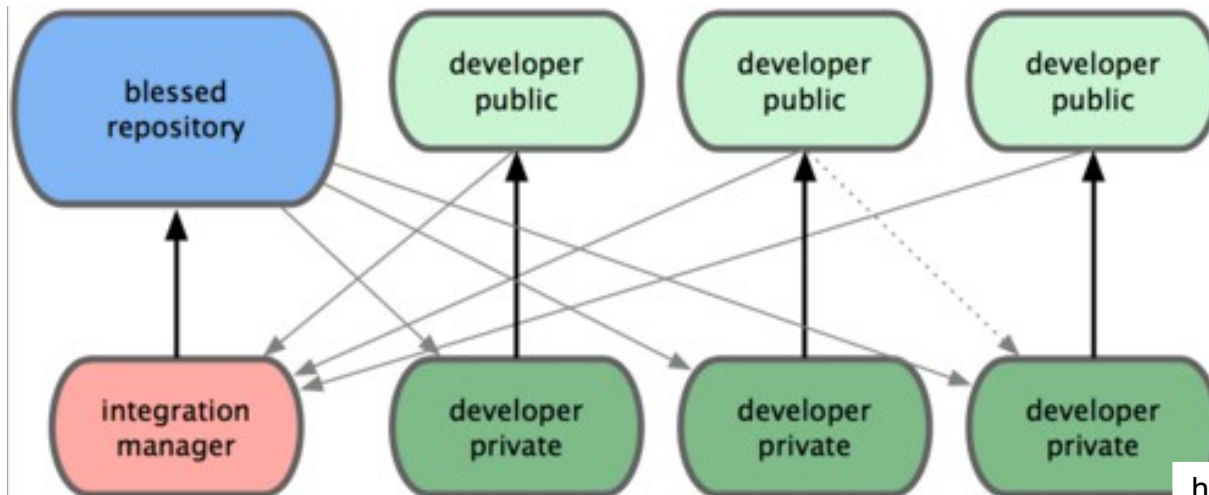
集中型の運用



svnと同じように使える

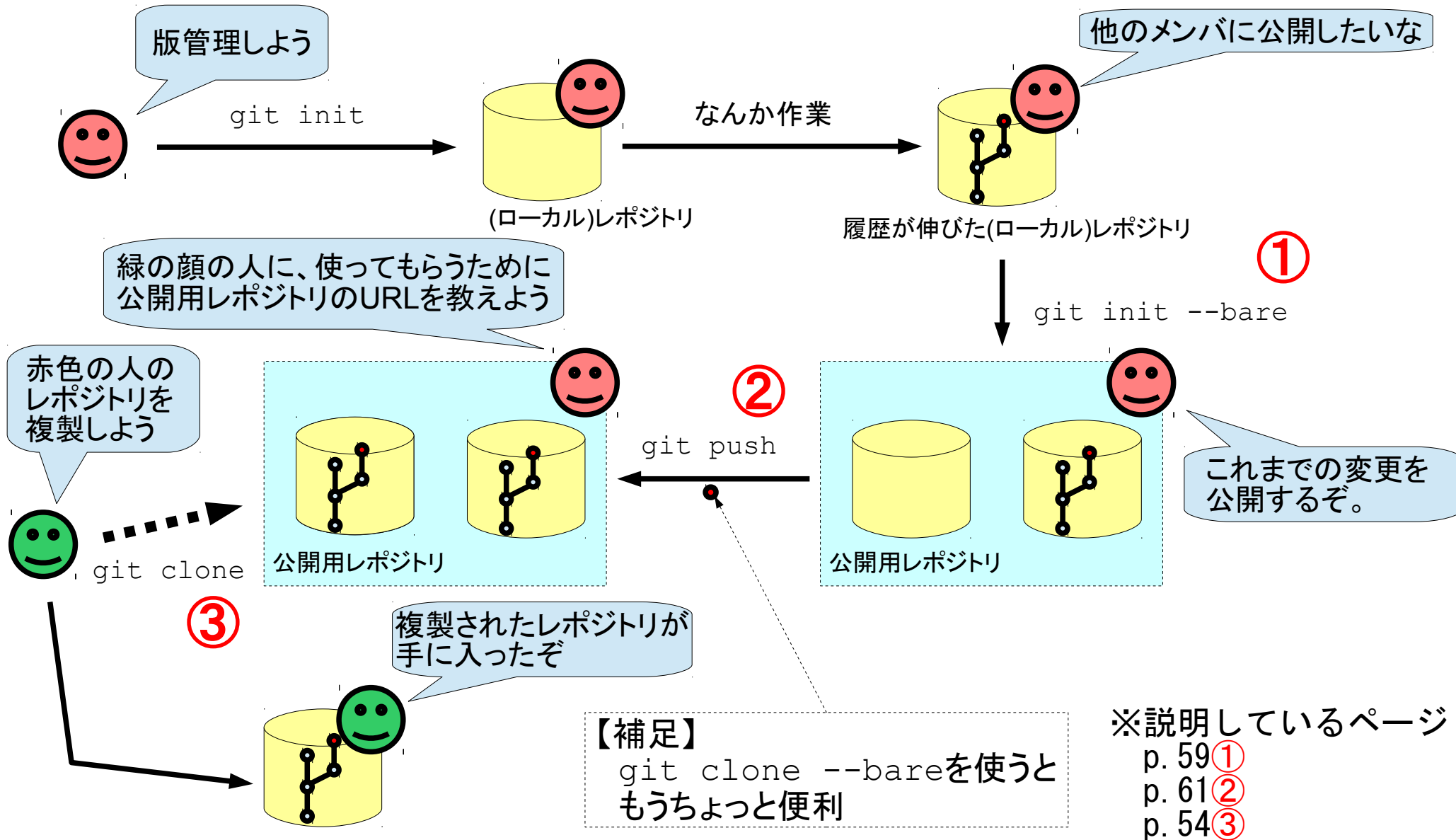
1. プロジェクトのメンテナーが公開リポジトリにプッシュする
2. 開発者がそのリポジトリをクローンし、変更を加える
3. 開発者が各自の公開リポジトリにプッシュする
4. 開発者がメンテナーに「変更を取り込んでほしい」というメールを送る
5. メンテナーが開発者のリポジトリをリモートに追加し、それをマージする
6. マージした結果をメンテナーがメインリポジトリにプッシュする

統合マネージャ型の運用



リモートレポジトリの使い方・1

リモートレポジトリのユースケース・1



リモートレポジトリの使い方・2

リモートレポジトリの作成

- リモートレポジトリを作りたいフォルダへ
移動してから、レポジトリを作成

```
$ git init --bare
```

- 公開用に作業コピーの不要なレポジトリを作ります。
作業自体はこれまでの(ローカル)レポジトリで行うことが普通です。
--bareをつけると.git/内のファイルをむき出し(=bare)にしたようなファイルツリーが作られます。作業コピーが不要なら、これまでと違って.git/内に入れておかなくても構いませんね。
- リモート、ローカルの名前は相対的なものです。
作業しているレポジトリがローカルなら、そのレポジトリの履歴がpushされるレポジトリがリモートになります。
普通はローカルレポジトリを砂場的に使用し、
公開(リリースなど)する履歴だけをリモートに送信(push)します。

cloneした時はこのページは不要です。最初からclone先がremote addされています。

リモートレポジトリの使い方・3

リモートレポジトリの登録

- これまで作業していたフォルダに、
今作ったリモートレポジトリを登録しましょう。

```
$ git remote add <alias> <remote-repos>
```

– 例えば・・・

```
$ git remote add origin /c/Users/hoge/remote.git
```

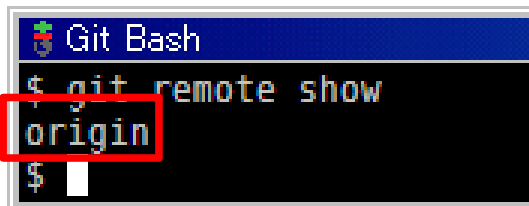
```
$ git remote add origin http://foo/remote.git
```

```
$ git remote add origin ../remote.git
```

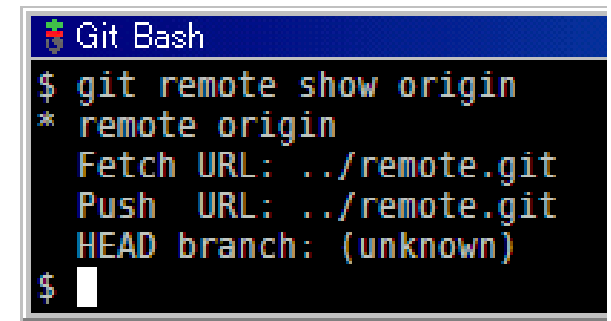
リモートレポジトリのbasenameには慣習的に「.git」を末尾へ付加します。

- 登録できたか確認してみます

– \$ git remote show



```
Git Bash
$ git remote show
origin
$
```



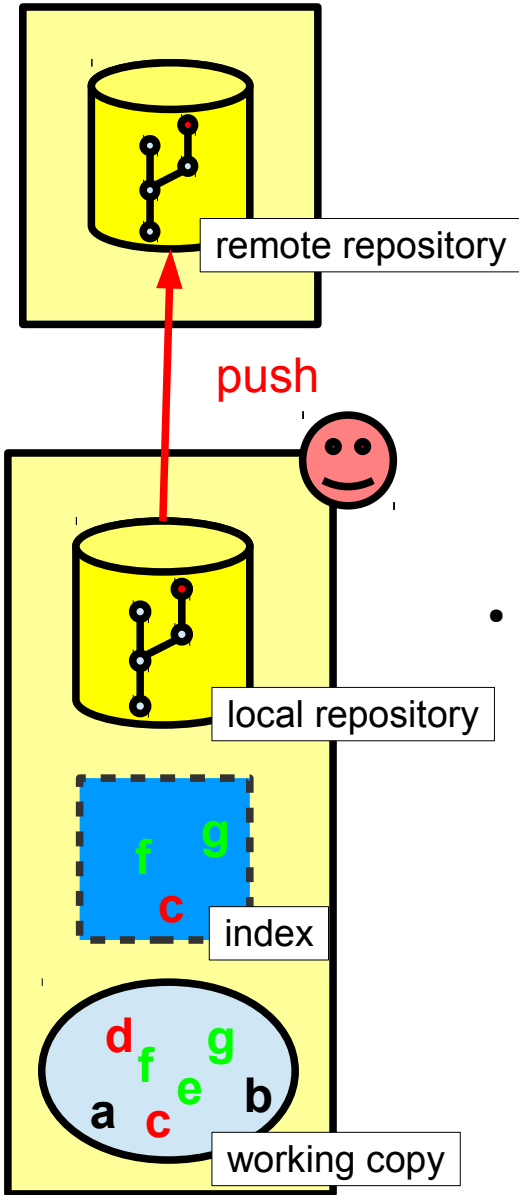
```
Git Bash
$ git remote show origin
* remote origin
Fetch URL: ../remote.git
Push URL: ../remote.git
HEAD branch: (unknown)
$
```

詳細な情報を表示

– originが登録されたのが確認できます。

リモートレポジトリの使い方・4

履歴の送信(push)



```
$ git push origin <branch>
```

- ローカルレポジトリの<branch>をリモートレポジトリに<branch>として作成する。(存在すれば履歴を更新する)(63ページのpush.defaultも参照)

- 例えば...

```
$ git push origin master
```

- 別のブランチ名でpushしたい場合には次ページのrefspecを使う。

push masterすると、ローカルのmasterとリモートのmasterが関連付けられました。

```
Git Bash
$ git remote show origin
* remote origin
Fetch URL: ../remote.git
Push URL: ../remote.git
HEAD branch: master
Remote branch:
  master tracked
Local ref configured for 'git push':
  master pushes to master (up to date)
```

push master後の詳細情報

refspec

- pushする時の参照(ブランチやタグ)指定方法
ローカルの参照とリモートの参照を紐付ける表記
`<source>:<destination>`
 - `<source>`、`<destination>`はブランチやタグなどのコミットを参照するもの。
相対表記を使って名前が付いていないコミットも指定できる。
 - 例. 3コミット前のbrをBRへpush
⇒ `git push origin br~3:BR`
- 前ページの最後のコマンドは
`$ git push origin master:master`
の略記。
- 別のブランチ名 (例としてanother) としてpushする場合は
`$ git push origin master:another`
とする。

push.default

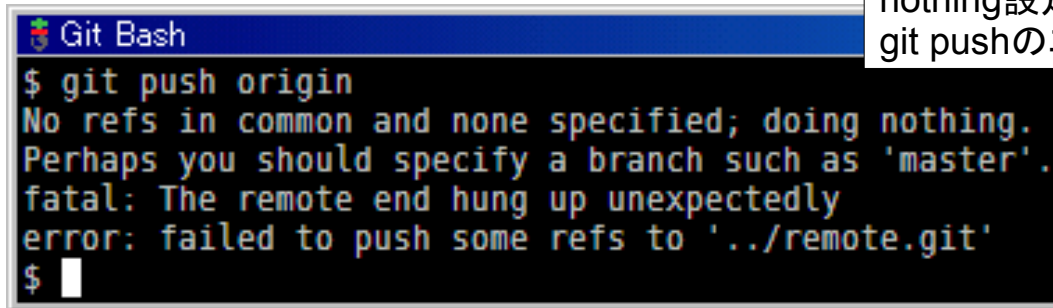
- push時にrefspecを省略した場合の扱いは変数`push.default`の値による。

- nothing

- 省略を許さない(Tabキーで補完が効くこともあるしオススメ！)

`$ git push origin master:master`

と書かなければならない。



```
Git Bash
$ git push origin
No refs in common and none specified; doing nothing.
Perhaps you should specify a branch such as 'master'.
fatal: The remote end hung up unexpectedly
error: failed to push some refs to '../remote.git'
$
```

nothing設定にしている時の
git pushのエラー表示

- matching (1.x系のデフォルトだったと思う・・・)

- `$ git push origin master`
とするとリモートにmasterを作成/更新

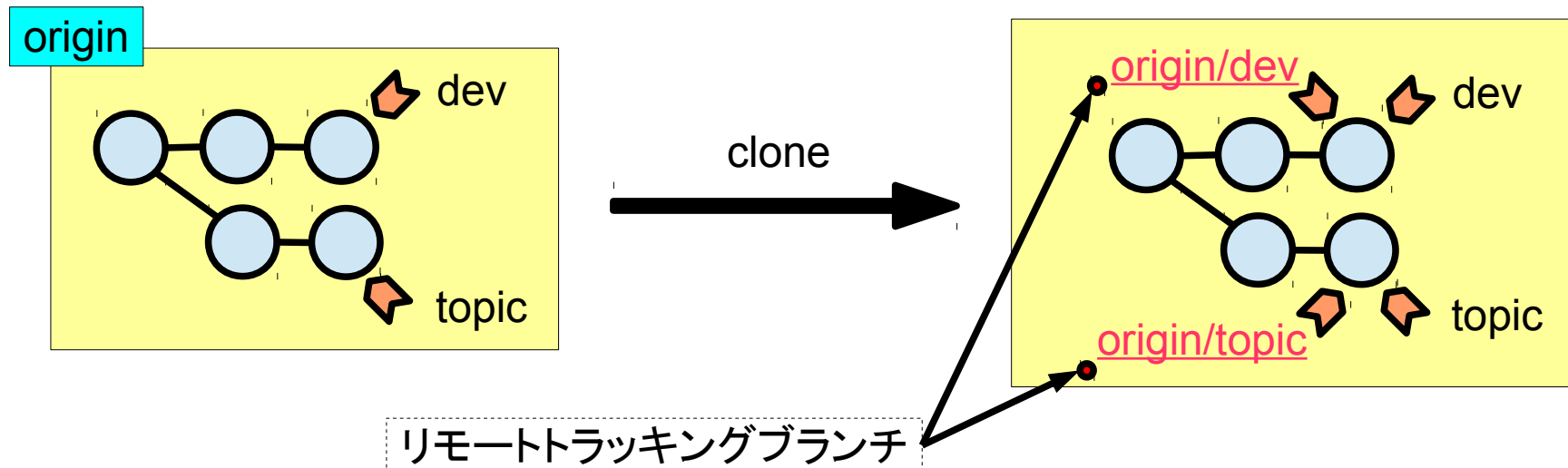
- 他にも値が存在するが詳細はヘルプ(`git config -help`)

- (最近はsimpleというのがあるみたいです)

リモートレポジトリの使い方・5

リモートトラッキングブランチ

- `clone`や`fetch`(後述)などでリモートレポジトリとのブランチを取得すると、ローカルレポジトリにその情報が取り込まれます。
 - リモートトラッキングブランチは、あるレポジトリのあるブランチがどのコミットを指しているかの情報を持っており、ローカルレポジトリのブランチと同一に扱うことが可能です。(例えば`git log`など)ただし、リモートトラッキングブランチを変更することはできません。リモートレポジトリにあるブランチを示しているからです。
 - ローカルレポジトリにコミットするのと同じようにコミットする操作のためには`push`を使うことになります。



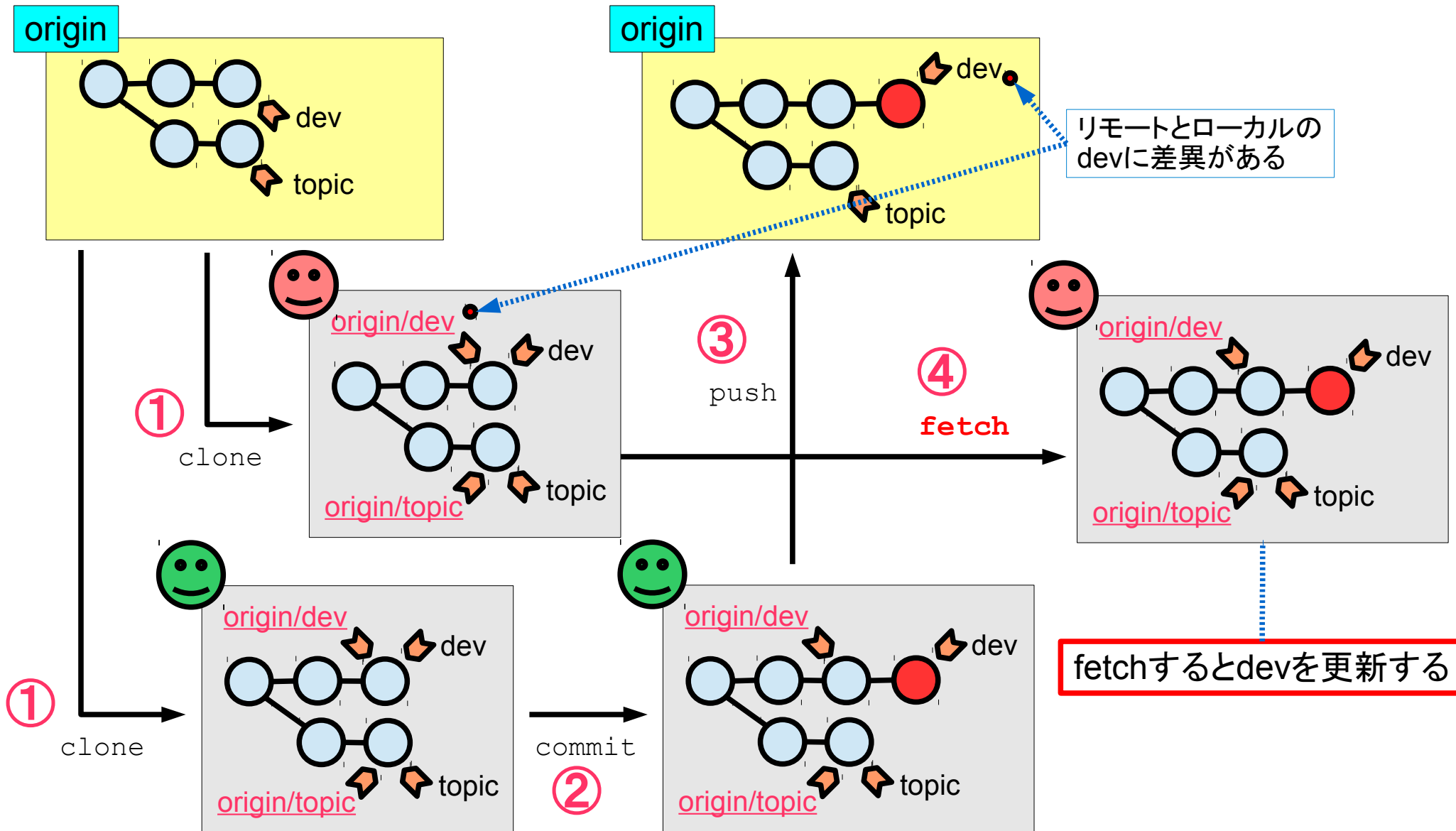
リモートレポジトリの使い方・6

複数人で開発する場合・1

- pushはfetchしてから。
 - clone後に他の開発者がpushすると、リモートレポジトリにあるブランチとローカルレポジトリにあるリモートトラッキングブランチとの間で差異が出る。
 - pushはこの違いを無くしてからすることになっている。
(fetchなしでpushが可能な場合に、どうなるかについては67ページを参照)
- fetch
 - リモートレポジトリのブランチを更新する
 - 自分がリモートレポジトリを取得した時点から他の開発者がpushすると、リモートレポジトリのブランチに差異が生まれます。
git fetchはブランチを取得し、ローカルにあるリモートトラッキングブランチを更新します。

リモートレポジトリの使い方・6

git fetch



リモートレポジトリの使い方・6

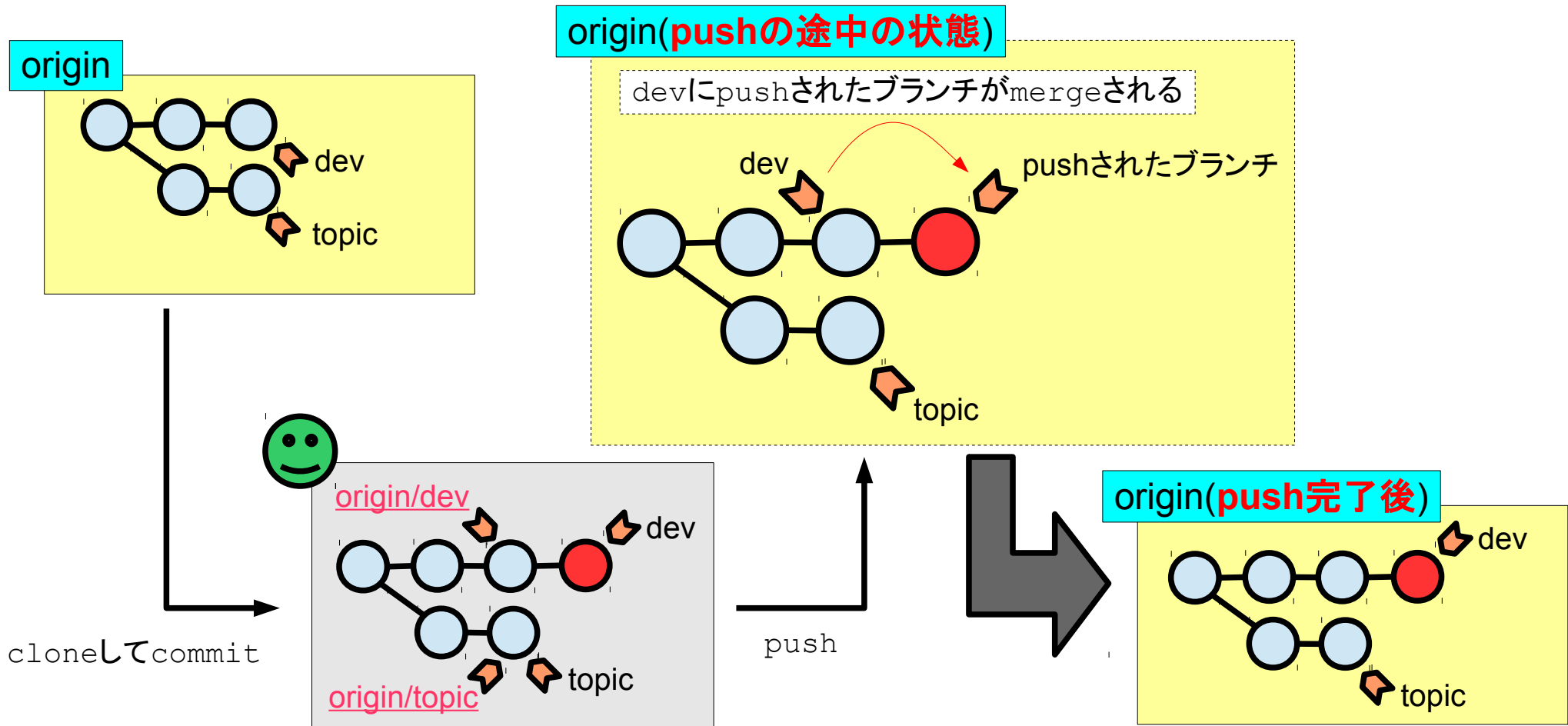
複数人で開発する場合・2

- pushは基本的にリモートレポジトリのブランチがfast-forwardで更新されるようにすること。
 - no-fast-forwardなpushを許してしまうと開発者間でブランチは無関係なものになってしまう。
(次々ページ)
- そもそもpushとは？
 - pushされるとリモートレポジトリでは送信された履歴を使ってmergeが行われる。
(次ページ)

リモートレポジトリの使い方・6

複数人で開発する場合・3

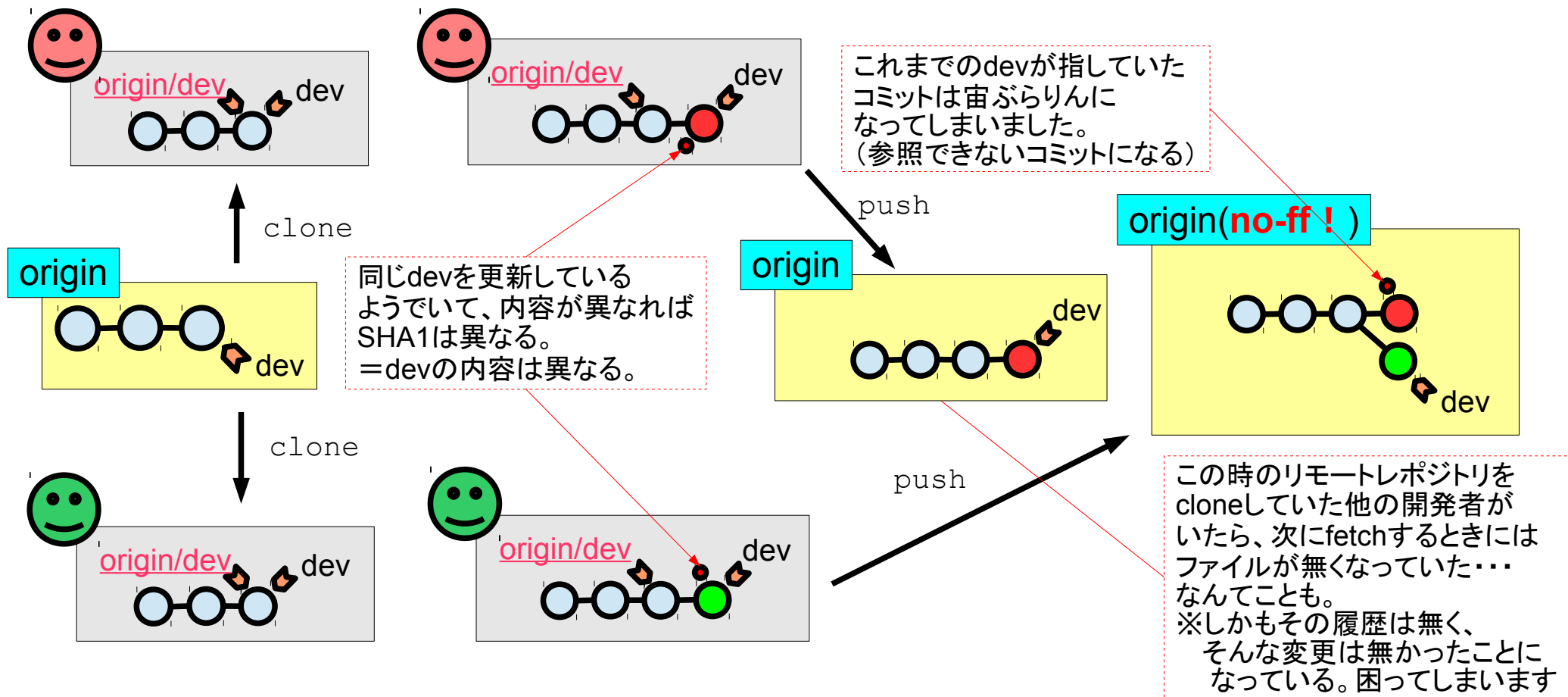
- pushされるとリモートレポジトリ側では送信された履歴を使ってmergeが行われる。
 - そのmergeがfast-forwardになるようにする。



リモートレポジトリの使い方・6

複数人で開発する場合・4

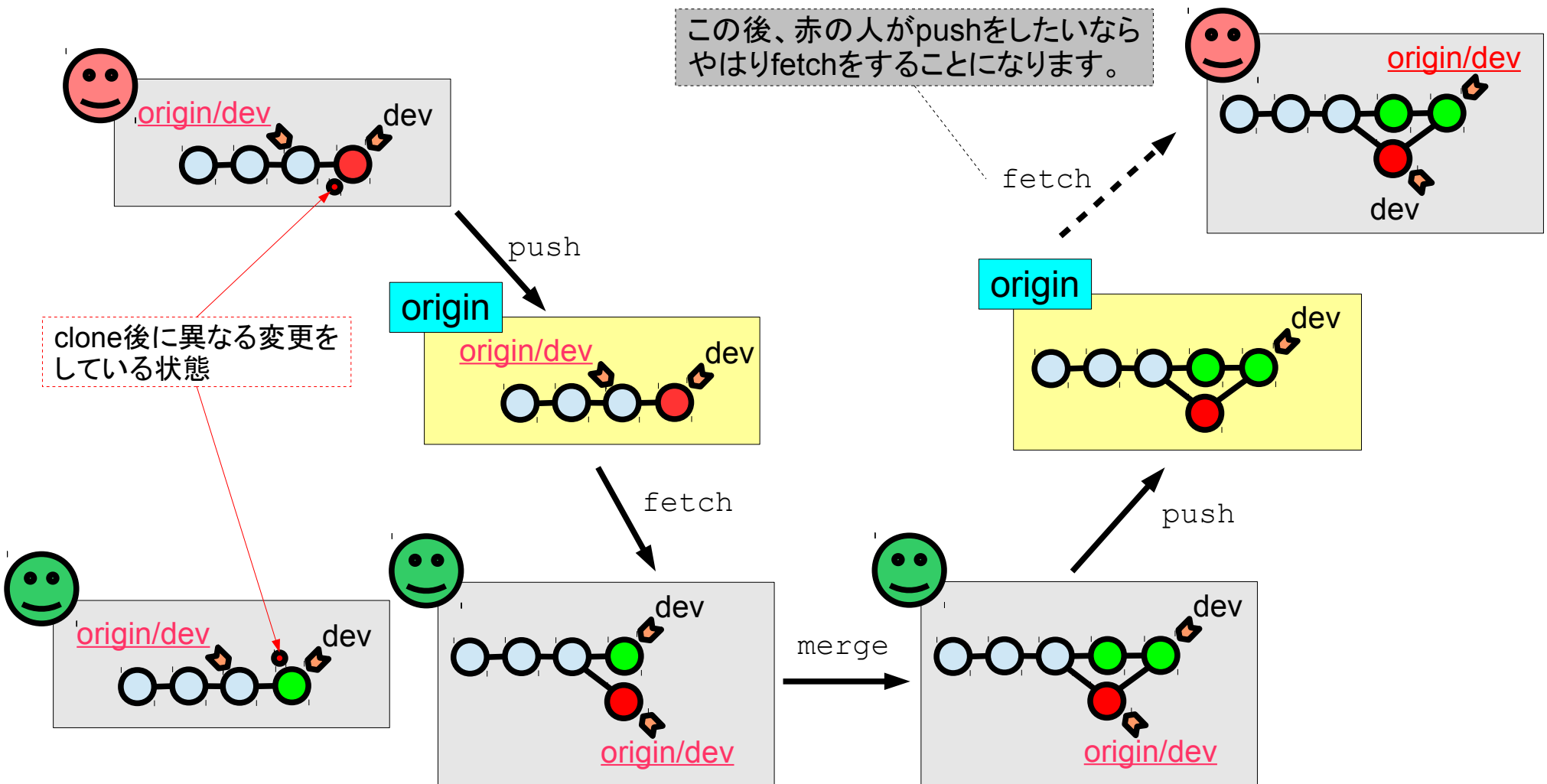
- no fast-forwardを許してしまうと・・・
⇒ブランチの内容に信ぴょう性が無くなってしまいます。



リモートレポジトリの使い方・6

複数人で開発する場合・5

- という訳で・・・pushはfetchしてから行いましょう。



余談・4

- リモートレポジトリに関するヘルプでの表記について
 - upstream

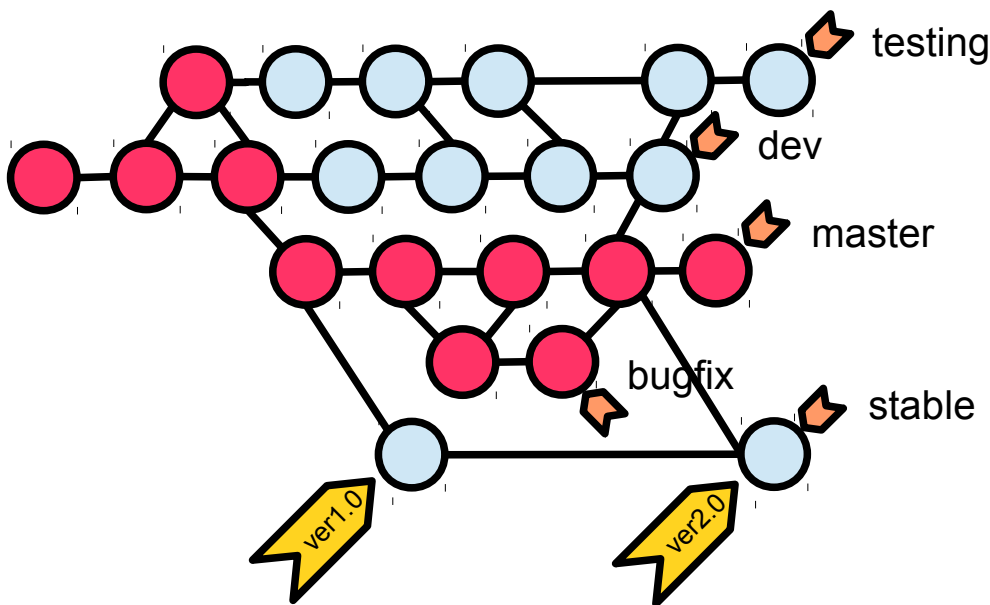
＝上流。p.57の統合マネージャ型運用のケースで使われるのが、集中型での使用に比べて、よりマッチする。
 - remote [repository]

一般的な用語。push先、という意味合いで。
必ずしもpush元とpush先のホストのネットワークが離れていることを指さない。

タグ・1

```
$ git tag <name> <commit-ish>
```

- <commit-ish>が参照しているコミットに<name>という名前のタグを付けます。
<commit-ish>はコミットを参照することができるもので、コミット名、ブランチ、タグのことです。
- ブランチと異なり、定点を指すのに使います。
 - ブランチはpushやcommitをすると指し示すコミットが変わってしまいますがタグは動きません。
 - 「ver0.01」などの内容が変わらないコミットを指すときに使うとよいです。
 - 公開後の変更は許されません。
(no fast-forwardなpushが許されないのと同じ問題です。)



【使い方例】

- 安定版にバージョン番号をつける。
- レビューに使用したコミットに
後で分かりやすい名前をつけておく。
(SHA1をメモしておくより良さそうです。)
- メモ的に後でチェックするものに付けておく
(この場合はpushしない方がいいですね)

タグ・2

- タグの種類

- 軽量(前ページのもの; `git tag <tagname> <コミット名やブランチやタグ>`)
- annotate(軽量と同じですがオプションaを使います)
 - 注釈を付加できます。`git commit`と同じようにエディタが開きます。
(「何月何日にレビューしてもらったコミット」とか「このコミットを後で●●について調べる」とか書いておくと便利かも。)

- タグに関するコマンドオプション

- pushするためにはtagsオプションを使います。

```
$ git push --tags origin master:master
```
- fetchするためにもtagsオプションを使います。
- **tagsオプションで注意するのは、全タグが対象になることです。**
ローカルにタグを作っていると、それが不要であってもpushされてしまいます。
pushするタグの指定にはrefspecを使うことになります。

```
$ git push origin refs/tags/tagname
```

※この際、tagsオプションは不要で、refspecのリモート側の指定も不要です。

余談・5

- タグは参照を保持します。
 - detached HEADで見られたように、コミットへの参照がなくなった時点で無くなってしまいうようなものを抑えることができます。



⇒ ブランチを削除すると、タグが消えてしまうことは無いということです。

- 上の図で `git branch -D topic` した時、`ver1.0` のタグと E～G のコミットは消えません。
- リモートへの余計なブランチ作成を抑えられるかもしれません。

管理対象から無視するファイル・1

- そもそも版管理に含めなければよいです。
 - でもgit statusに毎回表示されて、本当に必要な情報が見つけづらくなったり...
 - 管理しないという意味を表明することでgit cleanすることもできます。
- 無視パターンを記述するファイル(3種類)
- プロジェクト単位: `.gitignore`を版管理に含める
 - ビルドすると作成されるオブジェクトファイルなどを記述するとよいです。
開発者全員で共有されます。
- レポジトリ単位: `.git/info/exclude`
- 自分の環境(PC単位): 変数`core.excludefile`で指定したファイル
 - 使っているエディタなどのバックアップファイルなど、他の開発者に使ってもらう必要がないものを書くといでしょう。
- 無視パターンについて(パターンマッチについては次ページも参照)
 - パターンマッチは、原則として無視ファイルが存在するパス以下に適用される。
 - 無視の否定もできる。
 - 上位パスのgitignoreでオブジェクトファイルを無視しているが、ある下位パス以下では無視したくない時。
 - 先頭に「！」をつける

管理対象から無視するファイル・2

- パターンマッチの詳細

- 詳細はgit help gitignore の“PATTERN FORMAT”を参照。
各パターンは原則として、無視ファイルが存在するパス以下に適用される。
(下記項番6は例外である)
- 1. 空行はマッチ対象として使用されない。
gitignoreファイルを読みやすくするのに使うとよい。
- 2. 「#」で始まる行はコメントとして扱われる。
- 3. 「!」が頭につくと、その後に続くパターンを否定する。
- 4. 「/」で終わるパターンはディレクトリにマッチし、
普通のファイルにはマッチしない。
- 5. ワイルドカード“*”は、パス区切り「/」にマッチしない。
例えば、“Docs/*.html”は“Docs/git.html”にはマッチするが、
“Docs/ppc/git.html”にはマッチしない。
- 6. 「/」で始まるパターンは、パス名先頭にマッチする。
例えば“/*.*”は“cat-file.c”にマッチするが、
“foo/bar/cat-file.c”にはマッチしない。

やり直し・1

- amend

```
$ git commit --amend
```

- HEADをcommitし直す。
HEADのSHA1が変わることに注意。

- reset

- コミットに対して

- コミットを書き換える。
親へ巻き戻す。巻き戻し地点までのコミットを参照するブランチ等がなければ履歴から消える。
オプションによっては作業コピーは保ちつつコミットだけ戻せる。

```
$ git reset <rewind-point>
```

- インデックス、ファイルに対して
⇒18ページ

- revert

```
$ git revert <revert-commit>
```

<revert-commit>を打ち消すようなコミットを行う。

- 履歴は進む。既存の履歴を改変しないため、公開した履歴に対して有効。

- その他もろもろ

- 下記URLがまとまっててよいと思います。

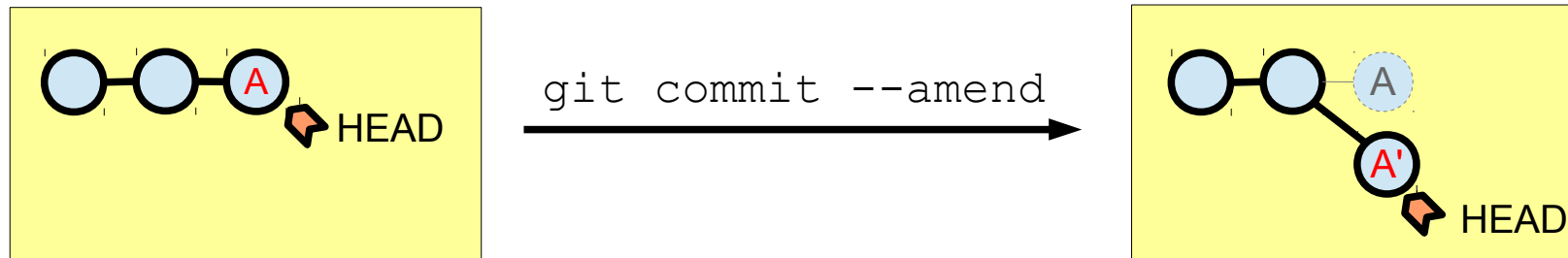
<http://labs.timedia.co.jp/2011/08/git-undo-999.html>

※多少、難易度が高いです。

本稿では説明していないstash、reflog、filter-branchなどの知識が必要となります。

やり直し・2

amend



- コミット後、ファイルのコミットし忘れに気づいた時などの直前のコミットをやり直したいときに用いる。
\$ git add <コミットし忘れたファイル>
\$ git commit --amend
通常のコミット手順にオプションを付けて使う。
- amend時のコミットメッセージは、最初のコミットメッセージを再利用できる(=エディタに最初から書かれている)
※新しいコミットメッセージにしたいなら、最初から書かれているメッセージを上書きして書き直せばよい。

やり直し・3

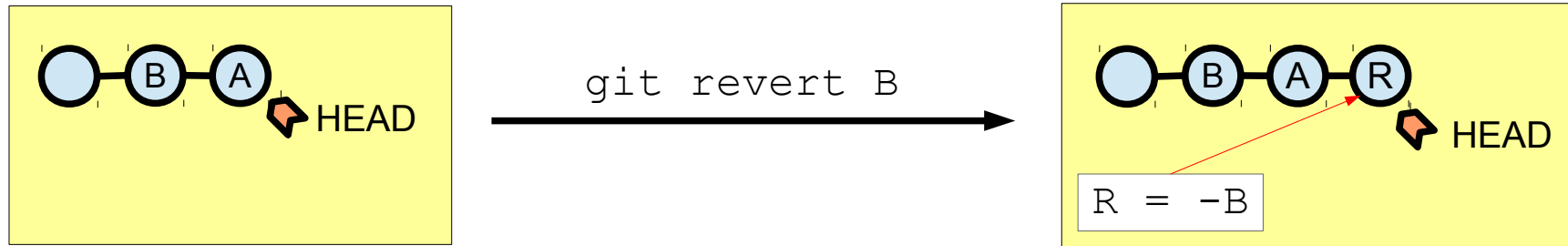
reset



- `$ git reset <rewind-point>`
resetすると、HEADを<rewind-point>へ戻す。
「<rewind-point>..HEAD」の変更分は作業コピーに残ったまま。
- 上記の変更分「<rewind-point>..HEAD」の扱いは、
オプション`soft`、`hard`、`mixed`で決めることができるが、デフォルトで困ることは無い。
※デフォルトオプションは`mixed`。
詳細を知りたい場合は、`git help reset`の"DISCUSSION"を参照。

やり直し・4

revert



- `$ git revert <revert-commit>`
revertでコミットされる内容は、
<revert-commit>をdiff表示で見たときの、
+と-を逆にしたものに対応する。
- <revert-commit>にBを指定すると、B~1..Bの内容が打ち消される。

tips・1

ログ表示あれこれ

- グラフィカルなlogはありませんか？
 - `git log --graph --decorate --oneline --all`
 - CUI。--allを付けなければHEADに到達可能な履歴を表示します。
 - `gitk [<branchname>...] [--all]`
 - GUI。これもオプションを付けなければHEADを表示。
 - TortoiseGitの”Show log”

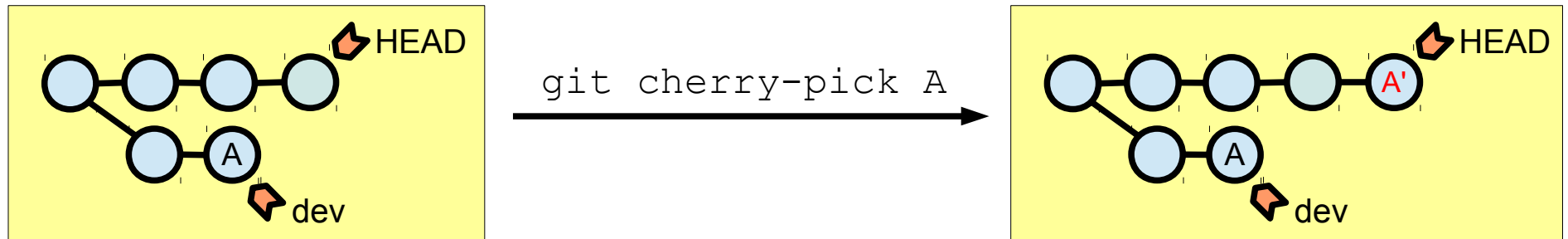
他にも何かフリーソフトを探してみるとか・・・

<https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>

tips・2

- cherry-pick

- あるコミットでの変更内容だけ欲しい。そんな時に。
- git-mergeを使うと、欲しい変更内容以外でコンフリクトが起きるので面倒だ。そんな時に。
(欲しい変更部分に相違があればコンフリクトは起きます)
- そもそもマージの必要がない。そんな時に。



※チェリーピックされたコミットA'はAとは親コミットが異なるためSHA1は変わる。

説明しないコマンド

- archive (svn exportに相当。いわゆるmake distの代わりになる。あまり使わないかも。)
 - show (GUIerには不向き。複数の文字エンコーディングがあると最悪。標準入力を使えるエディタがあれば、使い勝手がいいかも。)
 - rebase (運用によるがmergeで対応可能。必要ならbranch+cherry-pickで代替可)
 - submodule (管理が煩雑)
 - stash (branchで代替可能)
 - reflog (時間的な参照(.git/refs/*)の推移。失敗の復旧に便利ではある。)
 - bisect (テストコードがあればhook機構と合わせて使えるかも。)
 - notes (有用だが、push/fetch時の使い勝手が悪い所為か有名ではない。)
 - 細かいログ系のコマンド (whatchanged, shortlog⇒git log --format=***で十分)
 - メール関係: format-patch, am (パッチファイルとして渡したい時は良いかも?)
 - describe (リビジョン番号みたいなものを生成してくれます)
 - rerere (何回も同じコンフリクト解決するの面倒・・・→それ、rerereならできるよ)
 - grep (版管理してるobjectの範囲内でgrep)
-
- 他にも、git helpのGIT COMMANDSにはたくさんコマンドがあるが、よく使うのは一部。他のコマンドで代用可能だったり、あれば便利、程度のコマンドも多い。

などなど・・・

gitでできないこと

- gitはソースコードを書いてくれません
- ファイル毎のエクスポートができないと思う
 - svn exportみたいな機能
 - ※git archiveはファイル単位でできないと思う。
 - 差分があったファイルだけをエクスポートすることもできないんじゃないだろうか
 - TortoiseGitにはあるらしい？
- ディレクトリ単位のclone
 - svnではディレクトリ単位でサーバから取得(checkout)できる。

参考文献 ・ 1

- gitのヘルプ(git version 1.7.2.3.msysgit.0 同梱)
 - git tutorial / git tutorial-2
 - Everyday Git
- 他の公式ドキュメント
 - <http://git-scm.com/doc>
- Pro Git
 - <http://git-scm.com/book/ja/>
- Git Cheat Sheet
 - ⇒google検索
 - ※たくさんあります。使いやすいものを使いましょう。
- gitの内部の話
 - </doc/git/html/user-manual.html#hacking-git>
 - </doc/git/html/user-manual.html#object-details>

- 書籍

- 入門Git(Junio C Hamano, 秀和システム)
 - メンテナの著書。gitのスタンスを知るにはよい。
 - 雑多。
- 入門git(Travis Swicegood, オーム社)
 - シンプルなドキュメントで導入にはよい。
 - コマンドリファレンス的な使い方
- 実用Git(Jon Loeliger, O'Reilly)
 - ボリュームあり。
 - 体系的で深く知るためにはよいヒントになる。
- Gitによるバージョン管理(岩松信洋・上川純一・まえだこうへい・小川伸一郎 共著)
 - # ちゃんと読んでない・・・