バージョン管理システム入門

（初心者向け）

Gitの基礎勉強
～Gitによるバージョン管理を使う～

はじめに

tracpath

最近、Git（ギット）が多くの開発現場で利用されています。これまでは、Subversionを利用していたところも分散バージョン管理であるgitに移行しています。

> **ノート ： tracpath は2011年6月のバージョンアップで分散バージョン管理の2製品「git / mercurial」に対応しました。**

はじめてバージョン管理システムを利用する人、初学の人から、すでに開発に使っている人までこれまでのバージョン管理に比べ圧倒的に便利だ。という反面、「バージョン管理の利点と使い方はなんとなくわかる。が、分散バージョン管理はよくわからない」という声があります。私自身もgitの利用を開始した当初は、「Index? HEAD? なにそれ? 」状態でした。流行のオープンソースの場合、インターネットにたくさんの情報があり、有用な日本語訳も提供されています。
でも、バージョン管理という新しい概念を学ぶときは初めての用語が多く、そもそもどのような検索キーワードで検索すればよいのか、公式マニュアルも専門用語の羅列でなんかよくわからないという人も多いのではないでしょうか。分散バージョン管理も同じように新しい用語や概念が存在しており、それが「なんとなくわかっているけど、まだよくわからない」に繋がっていると思います。このチュートリアルで分散バージョン管理のモヤモヤを解消できるように解説したいと思います。

gitの基礎勉強ではつまずきやすい分散バージョン管理の考え方とgitの基本的な使い方を解説します。考え方さえ理解すればあとは実践あるのみです。とてもパワフルな分散バージョン管理「git」の世界にようこそ。

達成できること

tracpath

このコンテンツを読むことで

クローンからプルしてマージ、コミットしてプッシュする

の意味が理解できるようになります。

分散バージョン管理の考え方

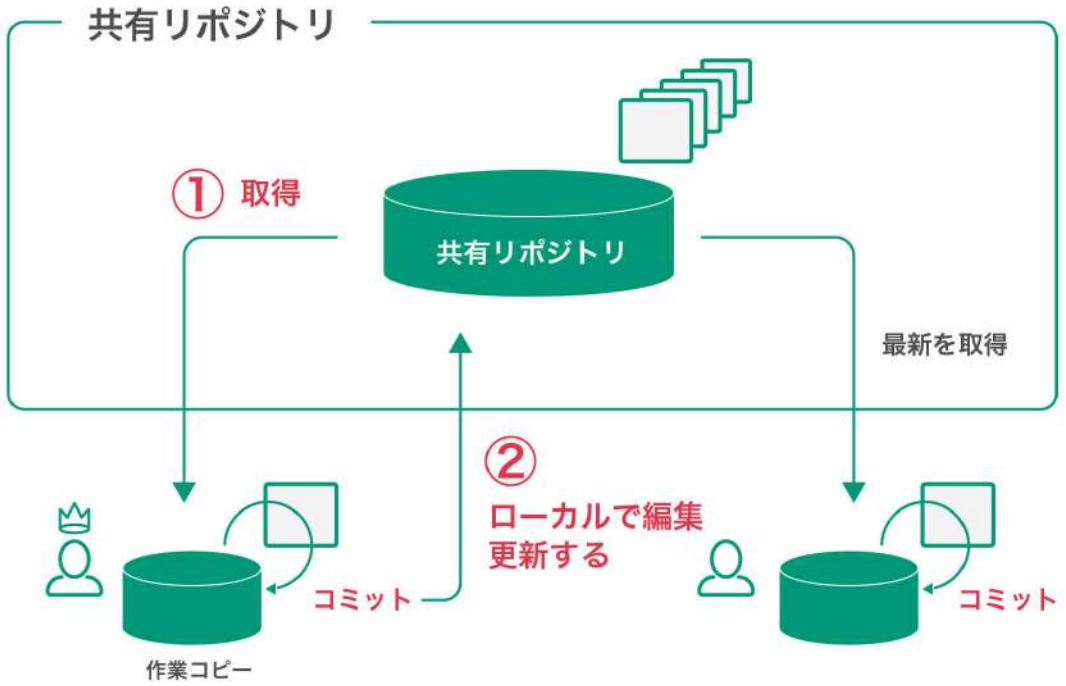
tracpath

git 初めの一歩

初めの一歩としてgitを運用するとき共有リポジトリという考え方を理解しておく必要があります。共有リポジトリとは、複数人でリポジトリを共有して開発を行うときに、開発者全員が参照し、開発の元になるリポジトリです。中央リポジトリともいいます。

ここで、初めて出てきたキーワード「リポジトリ」について解説すると、リポジトリとはファイルの貯蔵庫のことです。バージョン管理したいファイル群を格納しておく箱となります。

ノート ：
企業では部門などで共有ファイルサーバをつかってオフィス文書を共有することがあります。この共有サーバでみんなで共有されるファイルと同じようなものと考えて下さい。



分散バージョン管理の考え方

tracpath

共有リポジトリの絵を見て、説明が必要なキーワードがあります。

作業ディレクトリ

作業ディレクトリとは、共有リポジトリから取得したローカル環境のリポジトリです。Working Directory とも言われますがここでは「作業ディレクトリ」とします。作業ディレクトリは共有リポジトリと同じ構成のあなた専用のファイル群です。自由に追加・編集しても良いファイルであり、git をつかった開発ではこの作業ディレクトリ上で開発を進めることになります。

コミット

「コミット」は新規作成したファイルや編集したファイルを保存することを意味しています。新しい機能を追加したときやバグを修正したとき、作業ディレクトリ上で一区切りついた時にコミットしてそれまでの作業を一旦保存します。ファイルの変更後、ファイルを保存した上でコミットを行います。

ここまで

・「共有リポジトリはみんなで共有する大切な箱」です。開発を進めるときは「作業ディレクトリ」をつかって開発していきます。

・作業ディレクトリで開発を進めて、区切り毎に保存する操作を「コミット」といいます。

分散バージョン管理の考え方

tracpath

git をつかった作業の流れ

gitを使った開発を進めるために説明しておく必要がある3つの重要なキーワードがあります。
通常の開発は作業ディレクトリ上で行うことは説明しました。
作業ディレクトリで行った機能追加やバグ修正を共有リポジトリに反映する必要があります。この反映する作業のためにいくつかの段階と特殊な用語があるためgit:分散バージョン管理の分かりにくい点があります。

1．作業ディレクトリ（Working Directory）

共有リポジトリと同じファイルがローカルに展開され、ファイルの追加修正を行うディレクトリ

2．索引（Index）

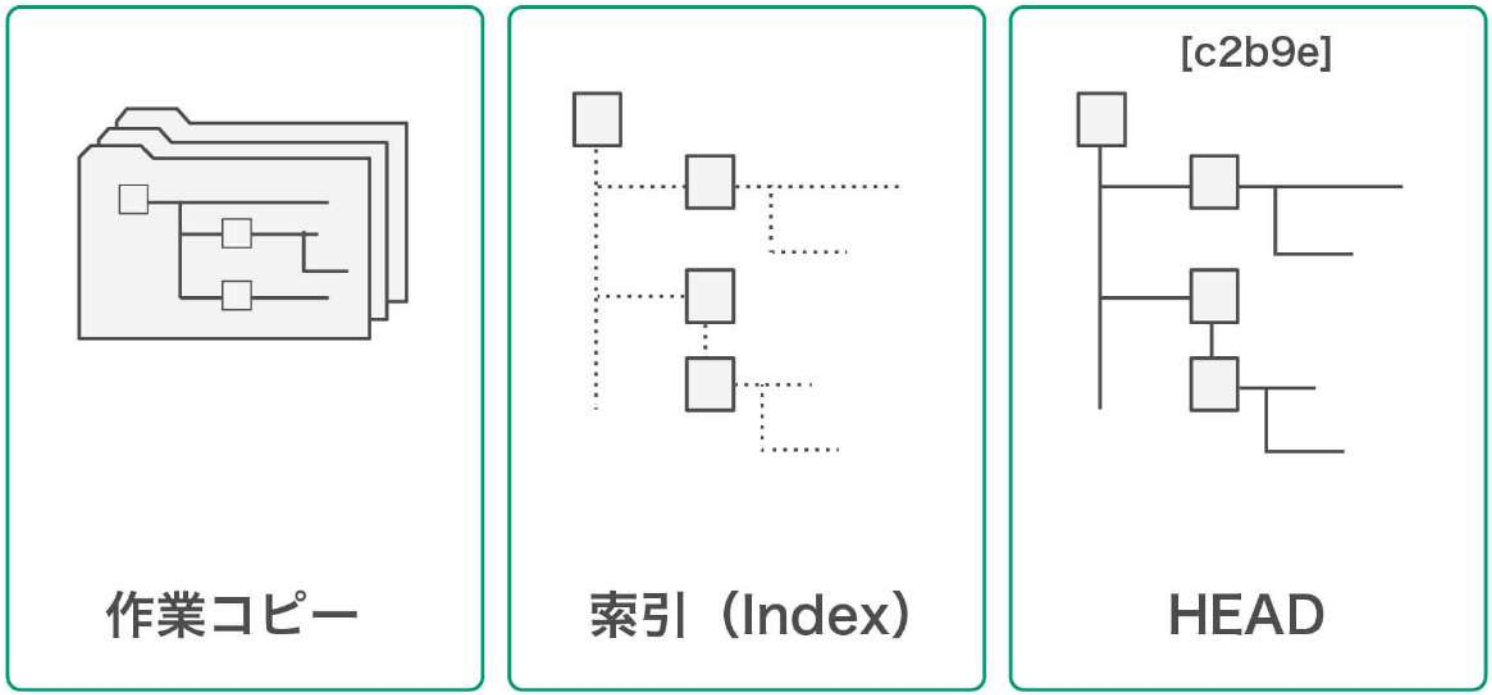
コミットするためにファイルを索引に追加し、コミット予定のファイル群を記録します。
このコミット対象のファイルを索引に追加する操作は、「ステージング」「コミット予定」「管理対象」と言われます。

3．作業最後のコミットを指すHEAD

git は作業ディレクトリでコミットが行われるとmasterブランチと呼ばれる作業ディレクトリを作ります。masterブランチは直近のコミットを指し示す意味を持っています。この後コミットを繰り返す度にmasterブランチの指し示す場所は進んでいきます。HEADとは作業ディレクトリでコミットされた最後のコミットを指すことを意味すると覚えておいて下さい。

分散バージョン管理の考え方

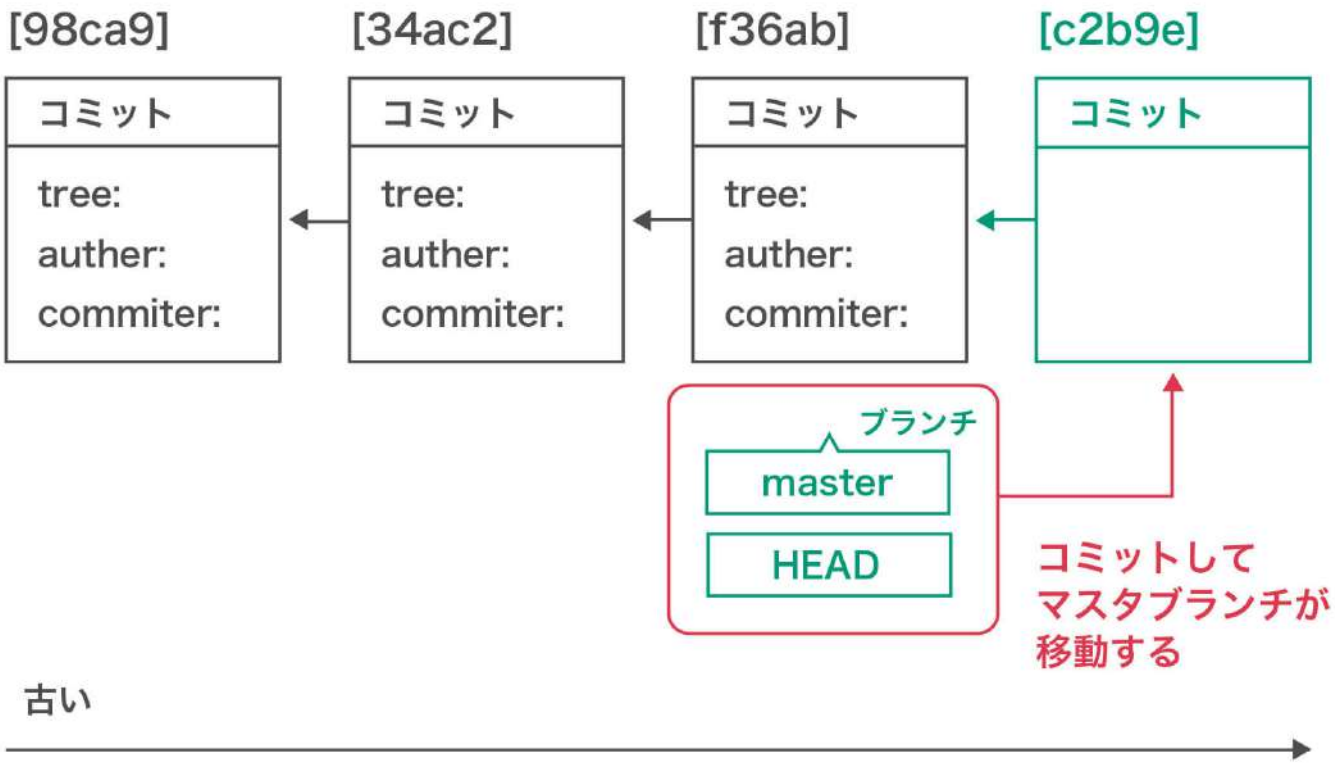
tracpath



分散バージョン管理の考え方

tracpath

コミットを作業ディレクトリで実行し、[master]ブランチが新しいコミットに移動します。



ここまでの流れはすべて作業ディレクトリで行います。この時点では共有リポジトリには反映されていません。古いコミット[98ca9]から HEAD として記録されたコミット[c2b9e]までを共有リポジトリに反映するための操作が必要です。

分散バージョン管理の考え方

tracpath

リポジトリの作成とクローン

これまでのgitをつかった作業の流れをコマンドで見てみましょう。
共有リポジトリがない場合、リポジトリを作成することができます。リポジトリの作成は

```
git init
```

を実行します。これで初期リポジトリが作成出来ます。

通常は開発メンバーで一緒に開発を進めることが想定されるため、共有リポジトリを利用する場合が多いでしょう。共有リポジトリを利用して開発を進める場合、クローン（clone）して作業ディレクトリをローカルに作成します。

Subversionなどは「チェックアウト（checkout）」を実行していましたが、分散バージョン管理「Git」はクローン（clone）します。このクローンはサーバが保持しているデータをほぼすべてローカルにコピーします。これはプロジェクトのすべてのファイルのすべての履歴が手元にコピーされることを意味しています。他の開発者に影響を与えずにブランチを作成したりできる分散管理バージョンのメリットです。

作業ディレクトリを作成するコマンドは

```
git clone https://username@domain/path/to/repository
```

を実行します。これで共有リポジトリからローカルに作業ディレクトリを作成します。

分散バージョン管理の考え方

tracpath

ファイルの追加&コミット

作業ディレクトリで変更したファイルを索引に追加します。（コミット準備、管理対象にする）

```
git add <filename>
または
git add *             # 変更があったファイルをすべて索引に追加します
```

これで編集したファイルが索引に追加されました。
それでは変更内容をコミットします。

```
git commit -m "初めてのコミット"
```

変更内容が索引からコミットされ、HEADに格納されました。まだ共有リポジトリには反映されていないことに注意して下さい。

分散バージョン管理の考え方

tracpath

共有リポジトリにプッシュする

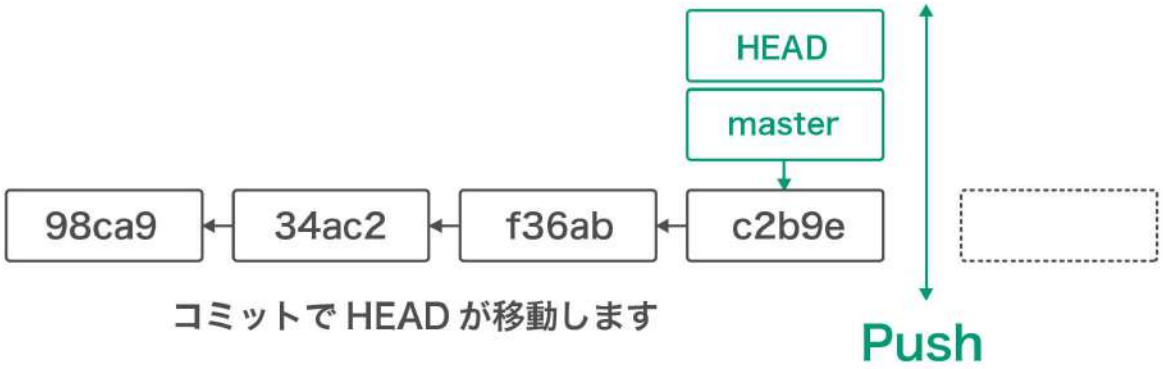
編集内容がローカルの作業ディレクトリ内でHEADに格納されています。
あなたの変更を共有リポジトリに反映させる必要があります。反映させるために利用するコマンドはpush（プッシュ）です。

```
git push origin master
```

originは共有リポジトリを指し示す名前です。コマンドの解釈として、「マスタブランチにコミットされたHEADの内容を共有リポジトリに送信する」となります。

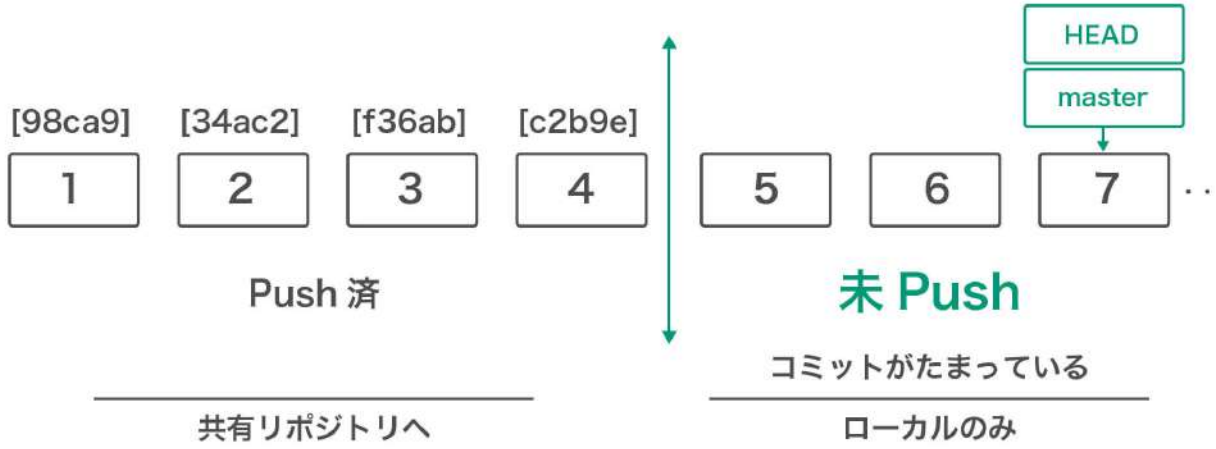
プッシュ（push）して共有リポジトリに作業ディレクトリのコミットをします。HEADの解説で使った図をもう一度見てみます。プッシュ（push）を実行します。

```
git push origin
```



分散バージョン管理の考え方

プッシュ（push）した所までのコミット[98ca9][34ac2][f30ab][c2b9e]が共有リポジトリに反映されます。
プッシュ（push）を実行した後も開発は継続します。さらにコミットを繰り返した場合作業ディレクトリにまだプッシュ（push）していないコミットが溜まっていくことになります。溜まったコミットを共有リポジトリに反映されるためにプッシュ（push）を行います。



ローカルでリポジトリを作成（git init）や共有リポジトリからクローン（clone）していない場合、共有リポジトリを登録することができます。

```
git remote add origin <server>
```

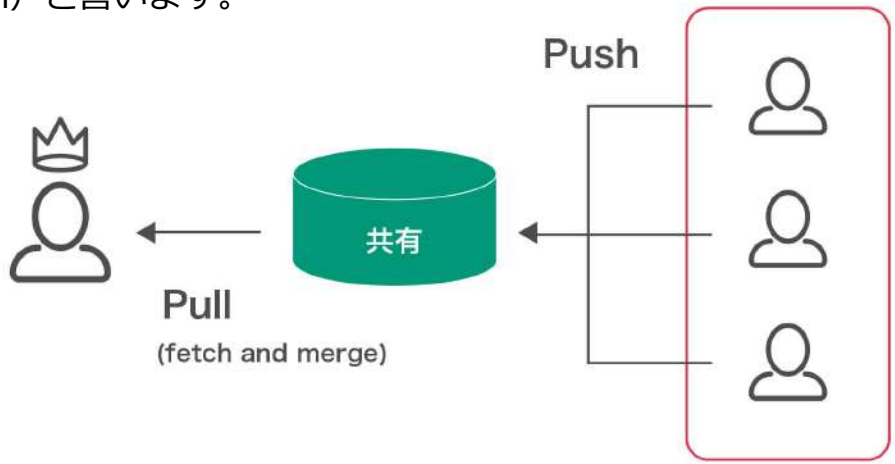
登録後に共有リポジトリにプッシュ（push）してください。

分散バージョン管理の考え方

tracpath

共有リポジトリから最新をプルし、マージする

共有リポジトリはあなた以外の開発メンバーもコミット・プルします。
他のメンバーが追加した機能や修正したファイルを自分の作業ディレクトリに取り込むことができます。
この操作をプル（pull）と言います。



プル(pull) とは、他のメンバーがコミットした内容は共有リポジトリに反映されます。共有リポジトリの最新情報を取得し（fetch）自分の作業ディレクトリと統合（merge）することを意味しています。

あなたの作業ディレクトリを最新のコミットに更新するには

```
git pull
```

を作業ディレクトリで実行します。この操作は共有リポジトリの最新情報を取得し、作業ディレクトリの状態に統合（merge）します。

分散バージョン管理の考え方

tracpath

分散バージョン管理によるチーム開発の流れ

ここまでがgitによるバージョン管理の基本的な考え方です。
分散バージョン管理は複数メンバーでの開発をしやすく、効率的にバージョン管理するための優れた機能を持っています。

ブランチの考え方

基本的な流れでマスタブランチを説明しました。gitをつかった開発ではブランチを必ず利用します。このブランチを理解することが、gitによる分散バージョン管理の特長です。

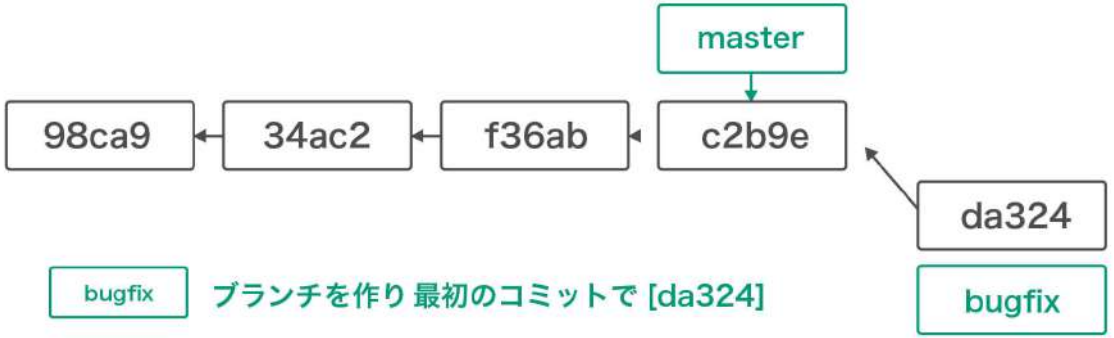
ブランチを利用するのはどのようなときでしょうか。通常は最初のコミットでgitが自動的に作成するマスタブランチ（master branch）を利用します。このブランチが標準のブランチとして機能しています。

ブランチを切り替えて利用するタイミングは、マスタブランチと関連のない機能を実装するときに利用されるのが一般的です。
ブランチはあなたの好きな名前を付けることができます。これはマスタブランチと関連のない機能を開発する場合に何のためのブランチか分かりやすいような名前を付けることができます。

ブランチを作成する例

マスタブランチで開発をしていますが、過去リリースしたコミット済のある機能について、性能向上が期待できるロジックを思いつきました。しかし、ほんとうにうまくいくのか実際に試してみるまでわかりません。試験的に試すだけなので、間違っても共有リポジトリに反映させることは検証が終わってからになります。

このようなマスタブランチの開発とは関係ない機能を試す場合に利用するのが一般的なブランチの利用方法です。



ブランチの考え方

tracpath

ブランチの基本とマージ

開発の現場で発生する状況としては、すでに本番稼働しているサービスがあり最初のバージョン（1.0）がエンドユーザに提供されています。開発チームは新しい機能（バージョン 1.1）を開発している状況を考えてみます。

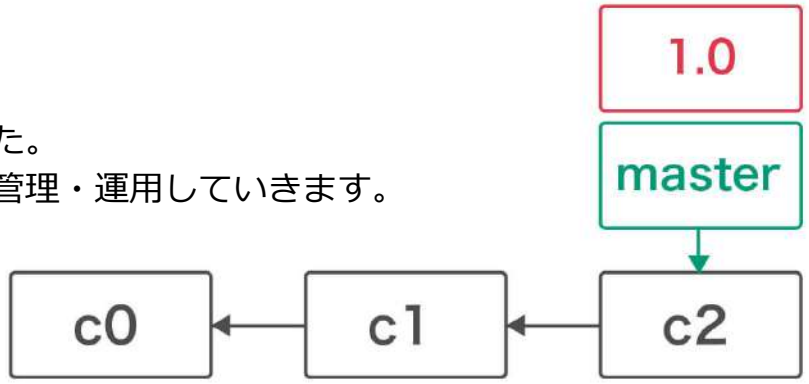
1. リリースされたバージョン1.0が稼働
2. 開発チームは新しいブランチを作成し、バージョン2.0の開発を実施
3. すでにバージョン2.0の開発は進んでいる時に、本番稼働しているバージョン1.0に重大な障害がみつかり、早急に対応が必要となった
4. バージョン2.0のブランチをバージョン1.0のブランチに変更する
5. 修正適用のためのブランチを1.0から作成する
6. 修正用ブランチで問題を解決し、修正用ブランチをバージョン1.0のブランチにマージする。新しいバージョン1.1ができる
7. マージしたバージョン1.1を共有リポジトリにプッシュ（push）する
8. 問題の解決後、作業途中のバージョン2.0 ブランチに戻り開発を継続する

ブランチの考え方

tracpath

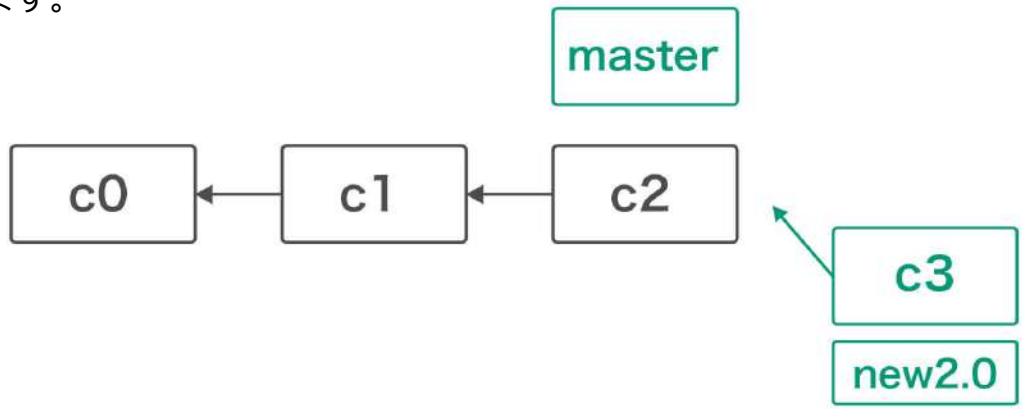
1. サンプルシステム 1.0 が稼働

あなたはサンプルプロジェクトの開発チームです。
無事、サンプルシステム 1.0 がリリースされました。
これから、運用チームがサンプルシステム 1.0 を管理・運用していきます。



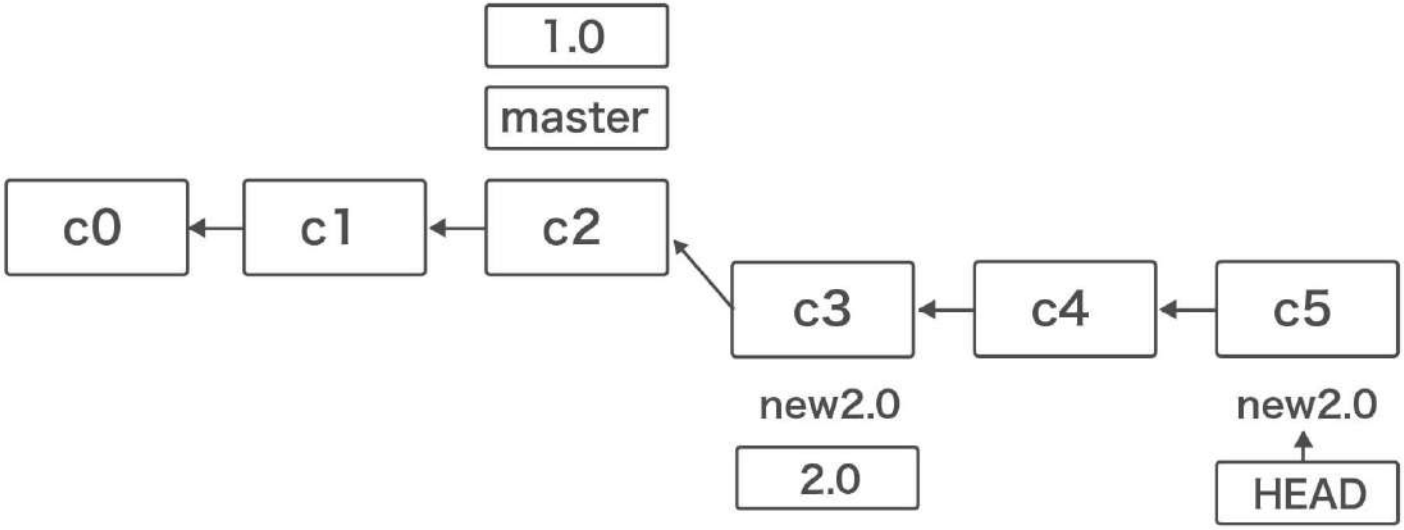
2. 新プロジェクトのスタート

早速、エンドユーザから要望が挙がってきました。この要望を機能仕様としてまとめ新プロジェクト「サンプルシステム 2.0」の開発がスタートします。2.0 のために 1.0 から新しいブランチ「new-2.0」を作成し、開発を進めていきます。



3. 運用チームより重大な障害報告

サンプルシステム2.0の開発は順調に進んでいますが、運用チームから重大な障害報告がバグ管理システムに挙がってきました。早急に対処する必要があります。（開発がスタートした2.0のブランチに対して、メンバーから新しい機能が追加され、何度もコミット/プッシュされている状態をイメージしてください。）

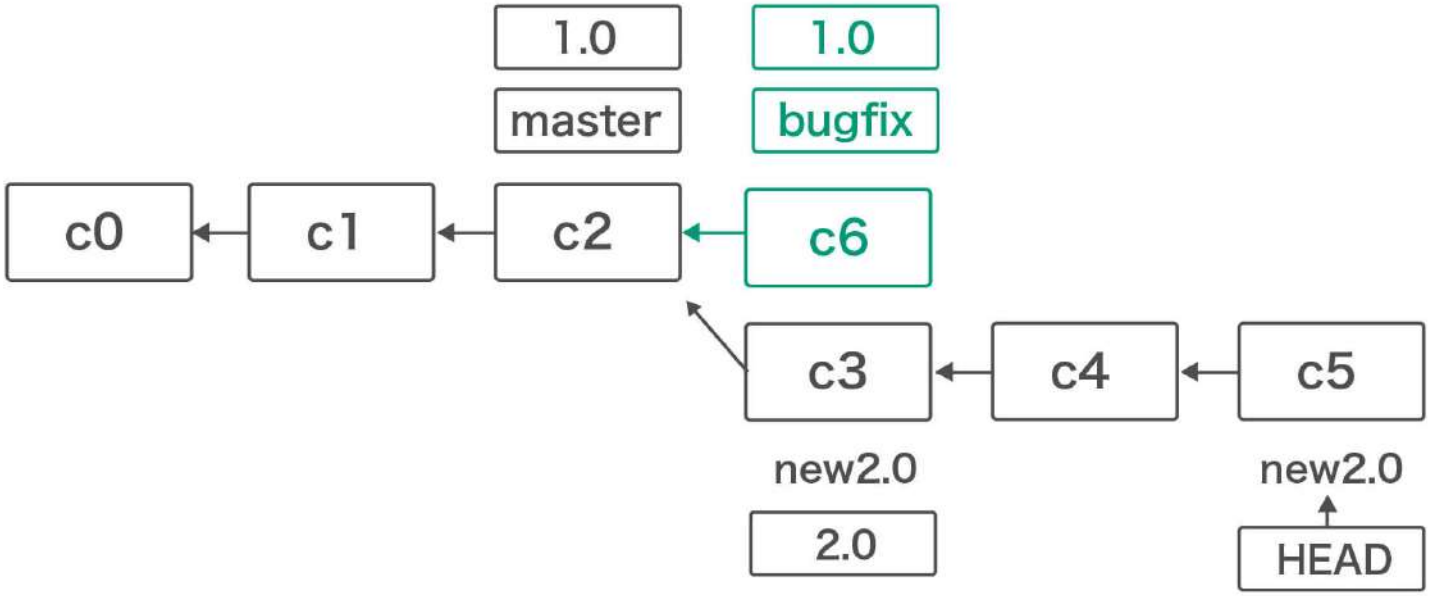


ブランチの考え方

tracpath

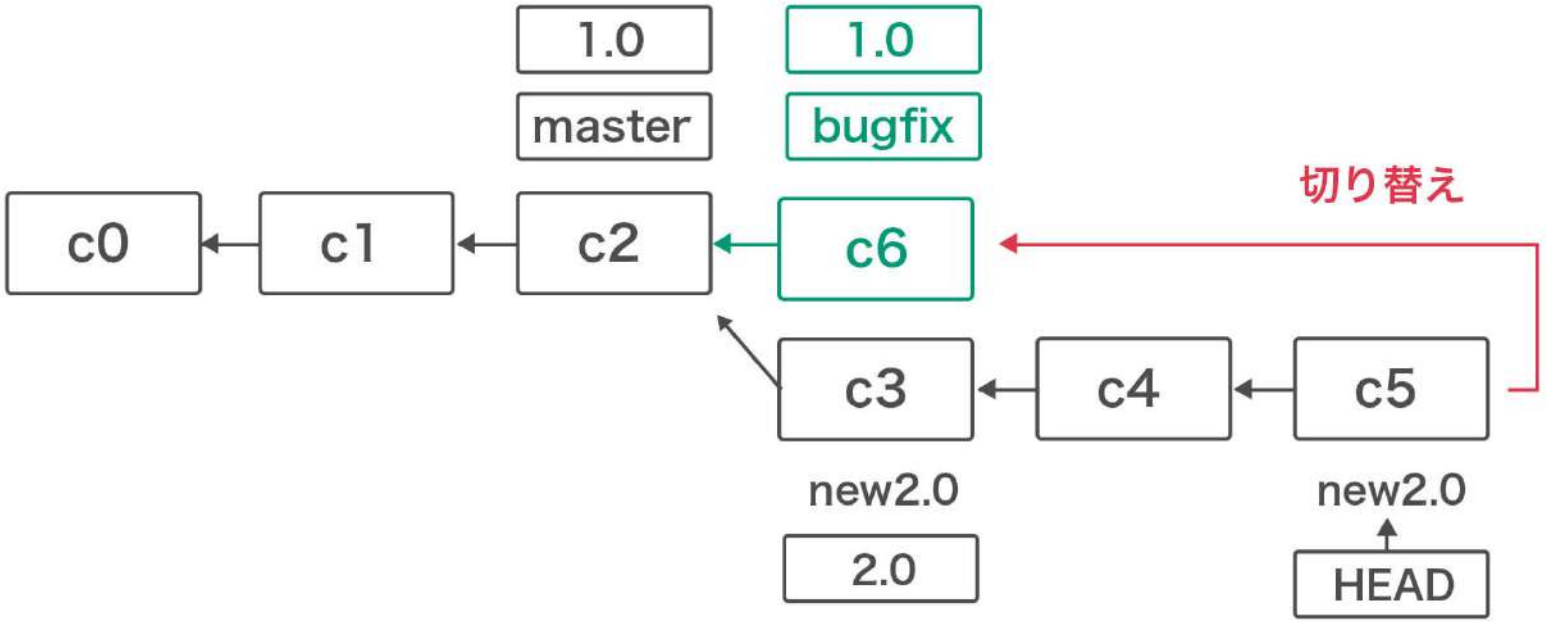
4. サンプルプロジェクト2.0の開発を一旦中断して、バグ対応

サンプルプロジェクト2.0の開発は途中ですが一旦ストップして、早急にバグに対処する必要があります。
バグの対処のためサンプルシステム1.0のmasterブランチから障害対応のためのbugfixブランチを作成します。



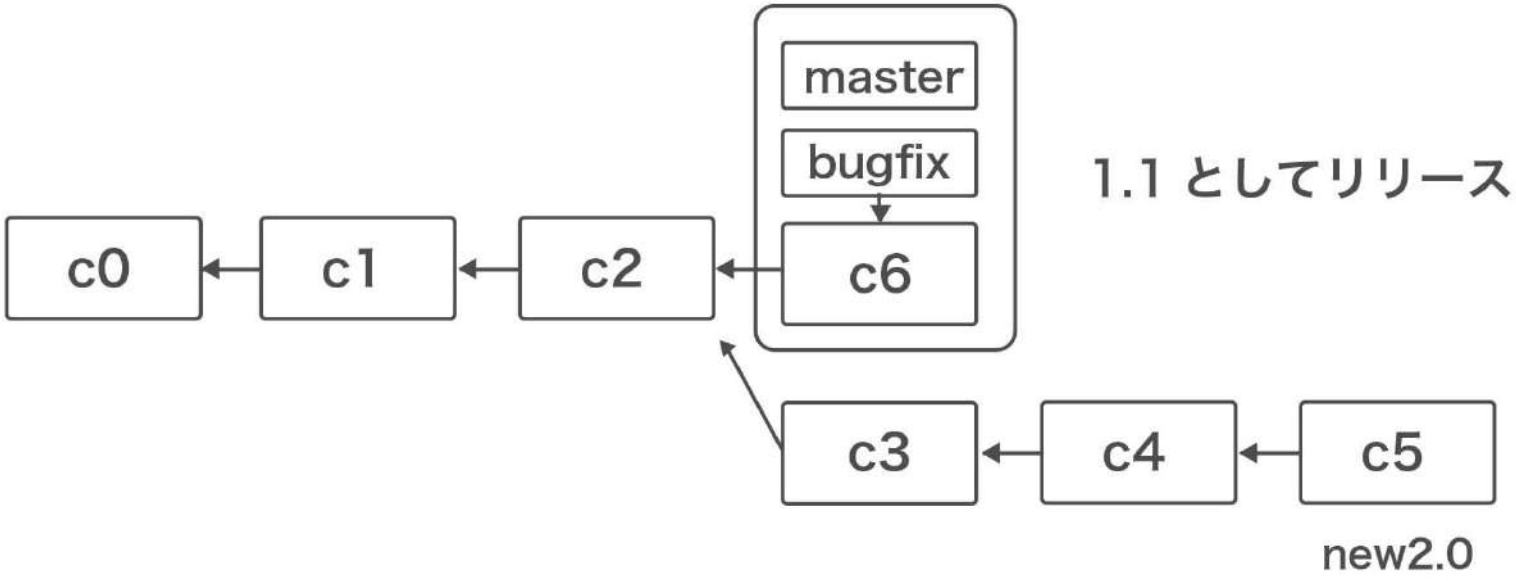
5. バグ対応のためのブランチ「bugfix」

バグ対応を実施します。これまで作業していたnew-2.0ブランチからbugfixブランチに切り替えます。
この切り替え作業はローカルの作業ディレクトリで実施します。



6. 問題を修正しmasterブランチにbugfixブランチをマージ

バグ対応が完了し、テストも完了しました。
masterブランチとバグ対応した（C6）bugfixブランチをマージします。
マージした新しい図は以下のようになります。



ブランチの考え方

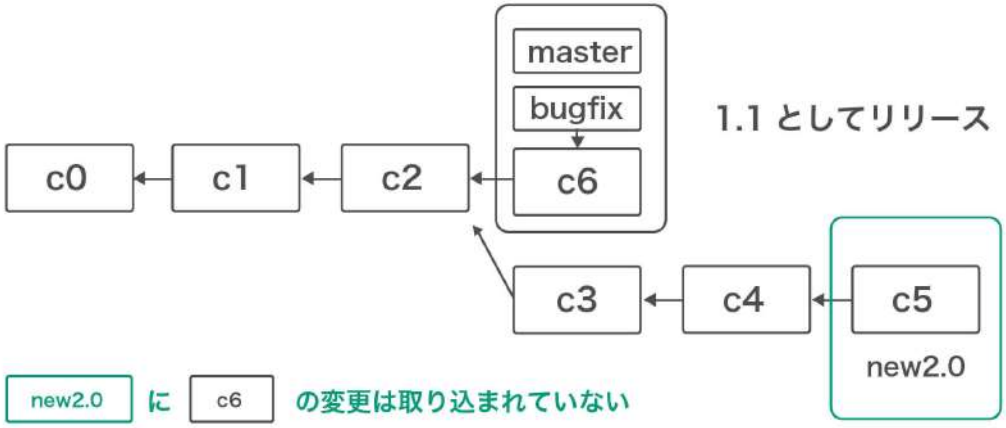
tracpath

7. マージした新しいバージョンをリリース

それではバグ対応版（1.1）をリリースします。
これで、運用チームから緊急対応依頼された問題は解決しました。お疲れ様でした。
今回のマージは「fast foward（ファストフォワード）」と言います。これはbugfixブランチがmasterブランチの親にあたるため、masterブランチの指し示す場所を前（C6）に進めただけです。
2つのブランチ、masterとbugfixは同じバージョンを指しています。

ノート：バグ対応が完了したbugfixブランチは削除することをオススメします。

警告：masterブランチ（bugfixブランチと同じ）で行ったバグ対応の修正は、new-2.0ブランチ（新バージョンの開発プロジェクト）には含まれていません。この状態のままnew-2.0ブランチの開発を継続しても良いですが、いつか new-2.0ブランチの内容をmasterにマージする必要があります。もう一つの方法として、masterブランチの内容をnew-2.0ブランチにマージする方法もあります。

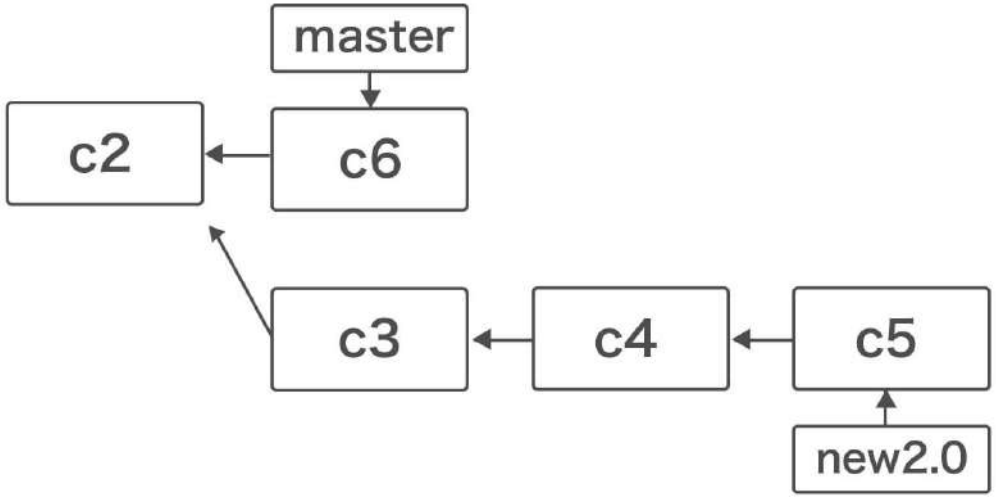


ブランチの考え方

tracpath

8. new-2.0ブランチに戻り、新プロジェクトの開発を継続

1.1版としてリリースしました。
ローカルの作業ディレクトリのブランチを切り替えます。masterブランチからnew-2.0ブランチに切り替え、新プロジェクトの開発を進めます。



ブランチの利用方法の一例を説明しました。
gitはCUIでの操作が多く実際のブランチ作成、切り替え、マージ処理は図で見ることができません。
視覚的にバージョンの遷移を見ることができればブランチ機能の理解を助けてくれます。
最近は、Windows 向け、Mac向けに優れたgitクライアントが数多くリリースされています。
CUIが苦手な方はGUIのgitクライアントを利用することをお勧めします。

よく利用されるgitクライアントツールは別のコンテンツで基本的な使い方を解説する予定です。

ブランチの考え方

tracpath

マージの基本

もの凄く奥が深いマージ。すべての機能をいきなり理解することはとても無理と思えるくらいのボリュームがあります。マージ（merge/rebase）はよく理解していない人が操作したことによって履歴の喪失などのリポジトリを破壊してしまうことがあります。

操作を間違えてしまいリポジトリを破壊してしまった。gitでトラブルが発生して開発が停止してしまった。これは、リベース（rebase）コマンドをつかったときに起こりやすいトラブルで、gitを使っている人は経験したことがあるのではないでしょうか。

そのため、マージ/リベース操作を実行するユーザを特定の人のみに制限し、他の開発者は マージ/リベース をしない。という方針をとっている開発チームも多いと思います。
規模が大きい開発チームの場合、技術者のスキルに大きな差があるため、ログの喪失、リポジトリの破壊につながる操作は許可しない方が賢明です。

マージ（merge）の基本について説明します。マージは枝分かれしたブランチを1つに統合することを言います。コミットされる度にブランチは1つずつ版を重ねていくことになりますが、コミットは1ファイルだけではなく複数のファイルが含まれています。そのため、マージ処理とは

1. 同一ファイルに行われた変更を統合する
2. コミットに含まれるファイル全体を統合する

ことを意味しています。

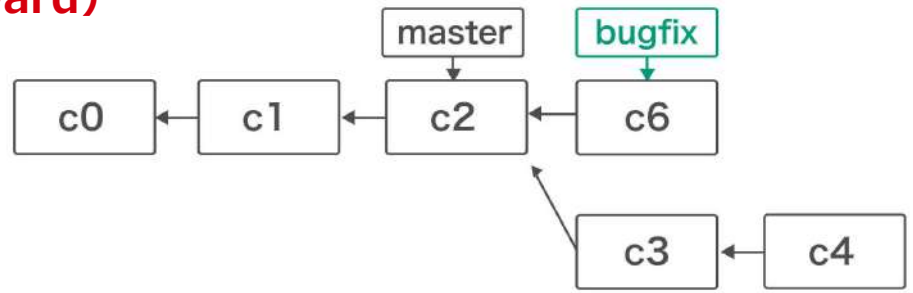
ブランチの説明で「ファストフォワード（fast forward）= 早送り」がありますが、これはマージ処理方法の説明です。マージ処理は
1．ファストフォワード（Fast-Forward）
2．ファストフォワードしないマージ（Non Fast-Forward）
3．リベース（rebase）
大きく3パターンのマージ処理があります。

ブランチの考え方

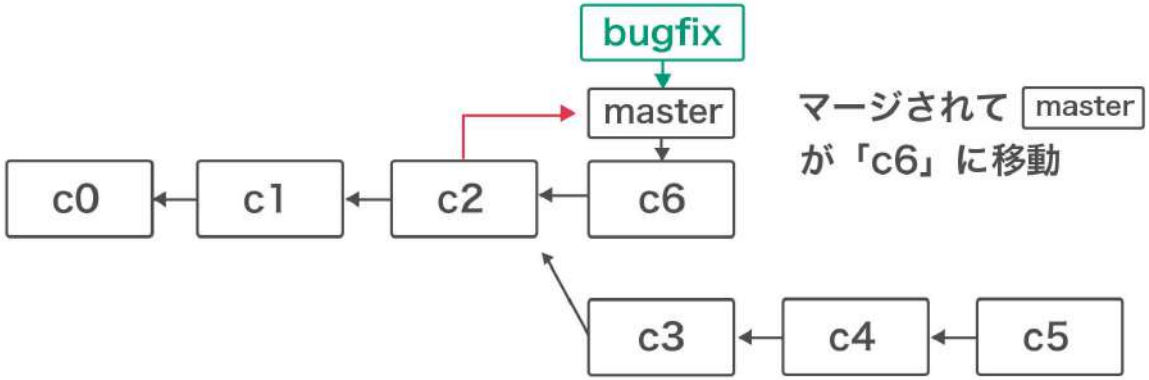
1. ファストフォワード（Fast-Forward）

ファストフォワード（Fast-Forward）は「早送り」という意味です。どういうことか？
これまでの例を参考に見てみます。

masterをチェックアウトしている状態です。
この状態でbugfixをマージします。



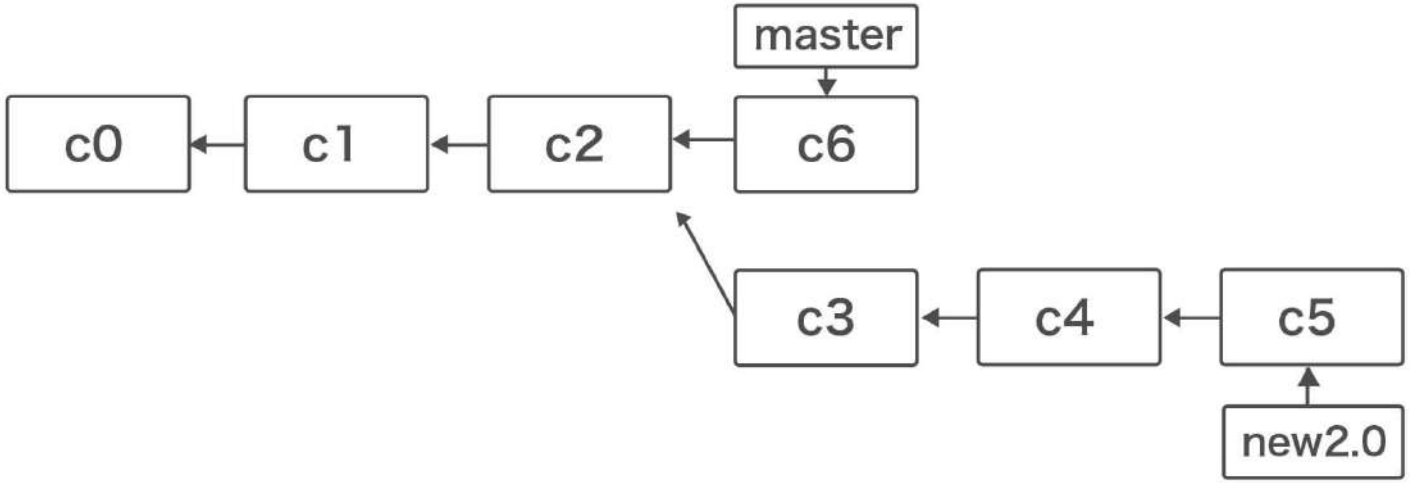

```
git merge bugfix
```



「c6」が「master」にマージされて、「master」が「c6」に移動しました。マージしたが新しいコミットは作られずブランチの位置が移動しただけです。だから「早送り」といいます。

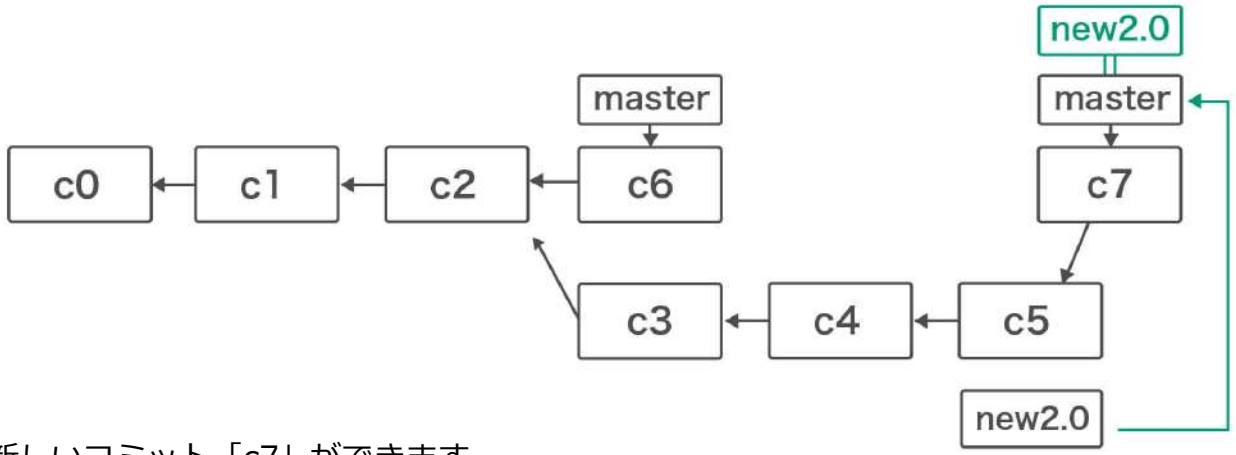
2. ファストフォワードしないマージ（Non Fast-Forward）

ファストフォワードしないマージ（Non Fast-Forward）とはどのような状態のときに利用するのでしょうか。



1. git merge new-2.0

ブランチの考え方



マージによって新しいコミット「c7」ができます。
ファストフォワードしないマージとは早送りと違って新しいコミットを作成し枝分かれした履歴を保持した状態で「c7」を作成します。

同じコマンドを実行しても ファストフォワードとファストフォワードしないマージと異なる動作をしていますがこのあたりは gitが賢く動作してくれます。

```
git merge <branch-name>
早送り（Fast-Forward）できればする、無理なら普通（Non Fast-Forward）のマージ
```

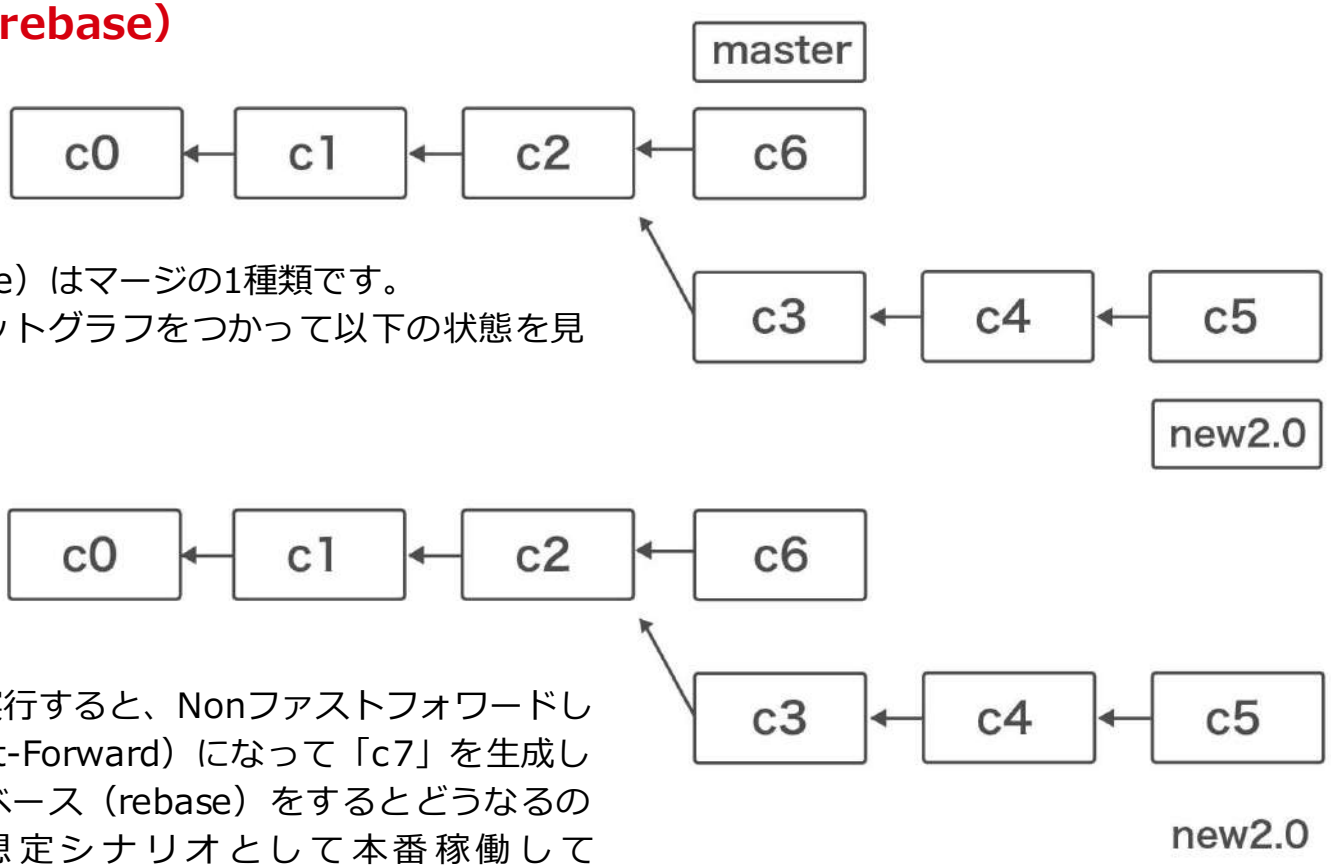
```
git merge --no-ff <branch-name>
普通の(Non Fast-Forward)マージ
```

```
git merge --ff-only <branch-name>
早送り(Fast-Forward)マージ
```

ブランチの考え方

tracpath

3. リベース（rebase）



リベース（rebase）はマージの1種類です。
これまでのコミットグラフをつかって以下の状態を見てください。

ここでマージを実行すると、Nonファストフォワードしないマージ（Fast-Forward）になって「c7」を生成します。では、リベース（rebase）をするとどうなるのでしょうか。想定シナリオとして本番稼働して「master」ブランチのバグ対応版である「c6」を取り込んだ上で、「new-2.0」ブランチの開発を継続するです。

ブランチの考え方

tracpath

リベースによってどうなったか動作を見てます。

1. 「c3」の差分をパッチ（3）にする
2. 「c4」の差分をパッチ（4）にする
3. 「c5」の差分をパッチ（5）にする
4. 「new-2.0」ブランチを「c2」から「c6」に移動
5. パッチ（3）を適用して、「c3′」
6. パッチ（4）を適用して、「c4′」
7. パッチ（5）を提供して、「c5′」

という処理を行います。

枝分かれしていた「c2」から新しい「c6」から枝がでてきました。
枝分かれしている元を再指定しているのでリベース（rebase）となります。

リベース（rebase）はローカルにのみ存在するブランチに対して操作するほうがよいです。共有リポジトリを使っている場合、ブランチ「new-2.0」は複数メンバーで共有されている可能性が高いため、リベース（rebase）してしまうと共有リポジトリにpushできなくなります。強制的に上書きすることはできますがバージョン管理システムを利用している開発では版の履歴がとても重要です。共有リポジトリではリベース（rebase）しないというルールを作ることをオススメします。

マージとリベースはこちらの情報を参考にしています。

- こわくない Git

ブランチの考え方

tracpath

コンフリクト

マージをおこなうと、コンフリクト（conflict） = 衝突が発生する場合があります。
衝突とは、マージを実行したとき「同じファイルの同じ行に違う変更をしていた」ブランチとブランチをマージした場合を言います。これはどちらの変更（ブランチ）を採用するか gitはわからないため利用者の判断が必要になるためです。

pullを実行したときもコンフリクトは起こる可能性があります。プル（pull）はフェッチ（fetch）とマージ（merge） を行っているためです。

コンフリクトが起こった場合、どうすればよいのでしょうか?
まずは、焦らず落ち着きましょう。gitはコンフリクトが発生したときコンフリクトが解決するまで処理を停止します。（マージは正常に終わっていないことになります。）

gitはコンフリクトが発生したとき、どこのソースコードで発生したのか、詳細な情報を提示してくれます。提示された内容のどちらを採用するのか判断するだけでコンフリクトは解消します。

1. git status

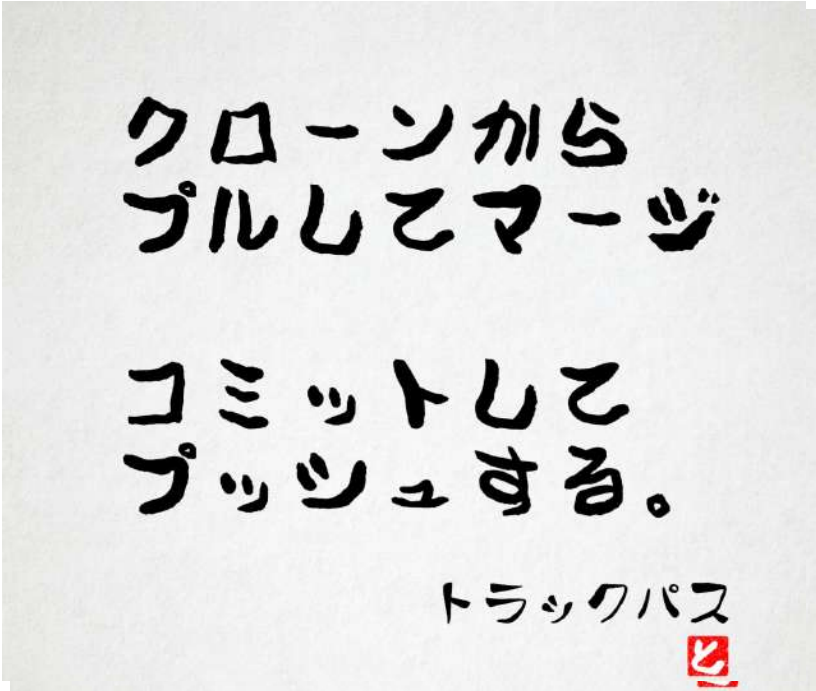
git status コマンドでマージ失敗したソースコードを確認し、手作業で修正しましょう。
この作業を繰り返し、コンフリクトが解決するまで直す必要があります。

コンフリクトを解決して一安心しても作業終了ではありません。
修正したソースコードに対して索引（index）に追加する必要があります。（git add）indexに追加してgitにコンフリクトを修正したことを通知します。最後にコミットを実行してコンフリクトは解決です。

最後に

tracpath

分散バージョン管理の概要が理解できたでしょうか。
クローンからプルしてマージ、コミットしてプッシュする



これは、gitの基礎勉強として分散バージョン管理の考え方を説明しました。
gitには、ブランチ、タブ、マージなど開発を強力にサポートしてくれる強力な機能をたくさん持っています。ぜひgitをあなたの開発に取り入れて下さい。

リンクと資料

tracpath

次に読むことをオススメ

- Pro git
- git–簡単ガイド
- こわくないGit
- リモートリポジトリを使うなら、tracpath（トラックパス）が便利です！
  下記記事をぜひご参照下さい。
  ■tracpath（トラックパス）を使って、安全に複数名でバージョン管理を行う

学習サイト

- LearnGitBranching

tracpath（トラックパス）のご紹介

tracpath

社内サーバにリモートリポジトリを作るのも一つですが、「開発にまつわる面倒事」をこの際全部、tracpath（トラックパス）に任せてみませんか？
バージョン管理サービス・プロジェクト管理サービスの「tracpath（トラックパス）」では、
ユーザー５名、リポジトリ数３つまで、永久無料で利用可能です。
さっそく実務でも使って見ましょう。
自らも開発を行う会社が作ったからこそ、開発チームの「作る情熱」を支える、やるべきことに集中出来るサービスになっています。
エンタープライズ利用が前提のASPサービスなので、セキュリティも強固です。