

記事

[Mihoko Iijima](#) · 2020年4月28日 · 9m read

GitLabを使用したInterSystemsソリューションの継続的デリバリー - パート I: Git

誰もがテスト環境を持っています。

本番環境とは完全に独立した実行環境を持てるほど幸運な人もいます。

-- 作者不明

[この連載記事](#)

では、InterSystemsの技術とGitLabを使用したソフトウェア開発に向けて実現可能な複数の手法を紹介し、議論したいと思います。次のようなトピックについて説明します。

- **Git 101**
- Gitフロー（開発プロセス）
- GitLabのインストール
- GitLabワークフロー
- GitLab CI/CD
- CI/CDとコンテナ

この最初のパートでは、最新のソフトウェア開発の基礎であるGitバージョン管理システムとさまざまなGitフローを扱います。

Git 101

これから説明する主なトピックでは、後の概念をより良く理解できるようにするため、ソフトウェア開発全般とGitLabがその取り込みにどのように役立っているのか、Git、あるいはもっと厳密に言えば重要なGit設計の基礎となる複数の[高度な概念](#)を取り上げます。

そして、Gitは以下のようなアイデア（[他にもたくさんありますが](#)、これらが最も重要なアイデアです）に基づいたバージョン管理システムです。

- 直線的でない開発とは、ソフトウェアが結果的にバージョン1から2、3とリリースされている間に、その裏でバージョン1から2への移行が並行して行われることを指しています。実際、さまざまな開発者が多数の機能/バグ修正を同時に開発しています。
- 分散開発とは、開発者が単一の中央サーバーや他の開発者から独立している状態を指しており、自分の環境で簡単に開発できます。
- マージ - 前の2つのアイデアによって多くの別バージョンが同時に存在することになり、それらを統合して1つの完全な状態に戻す必要性が生じます。

ただ、Gitがこれらの概念を発明したと言っているのではありません。違います。むしろGitはこれらの概念を容易化・一般化し、関連する複数のイノベーション（Infrastructure as Code/コンテナ化によるソフトウェア開発の変化など）と結びつけました。

git基本用語集

リポジトリはデータとそのデータに関するメタ情報を保存するプロジェクトです。

- 物理的にはリポジトリはディスク上のディレクトリです。
- リポジトリはファイルとディレクトリを保存します。
- リポジトリには各ファイルの変更の完全な履歴も保存されます。

リポジトリは以下の場所に保存できます。

- お使いのコンピューター（ローカル）
- リモートサーバー（リモート）

しかし、gitの観点からするとローカルリポジトリとリモートリポジトリには特に違いはありません。

コミット

はリポジトリのある決まった状態を表しています。コミットするたびにリポジトリの完全な状態を保存して

は、

リポジト

リが急速に肥大化

してしまうのは明らかです。そのた

め、コミットでは現在のコミットとその**親コミット**との差分である**diff**を保存しています。

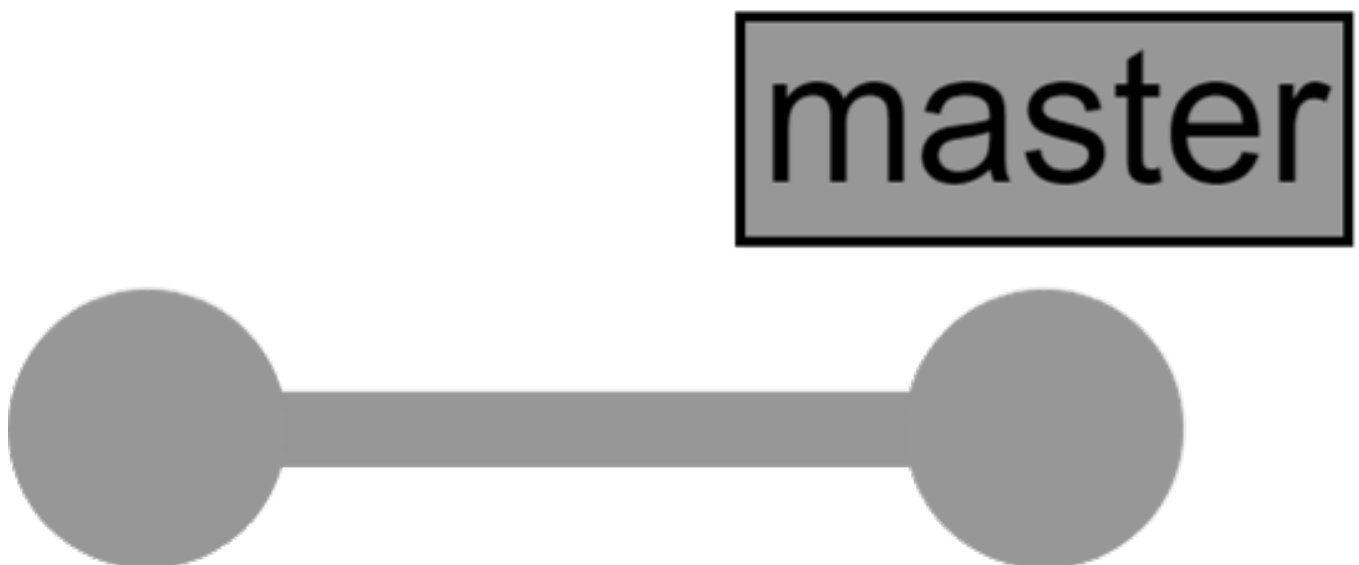
異なるコミットは、次のように異なる数の親を持つことができます。

- 0 - リポジトリ内の最初のコミットには親がありません。
- 1 - これは通常の場合です。コミットが親コミットと同じようにリポジトリ内の何かを変更するようになります。
- 2 - 2種類のリポジトリの状態がある場合、それらを1つの新しい状態に統合できます。また、その状態とそのコミットには2つの親があります。
- 3以上 - 2種類を超えるリポジトリの状態を1つの新しい状態に統合した場合に発生し得ます。これは私たちの議論には特に関係ないと思いますが、このような場合もあります。

親の場合、その差分となる各コミットは**子コミット**

と呼ばれます。各親コミットには、任意の数の子コミットを含めることができます。

ブランチはコミットへの参照（またはポインタ）です。次のようなイメージです。



この画像には2つのコミット（灰色の円）を持つリポジトリがあります。2番目のコミットはmasterブランチのHEADです。コミットを追加すると、リポジトリは次のようになります。



これは最も単純なケースです。1人の開発者が同時に1つの変更を手がけます。

しかし

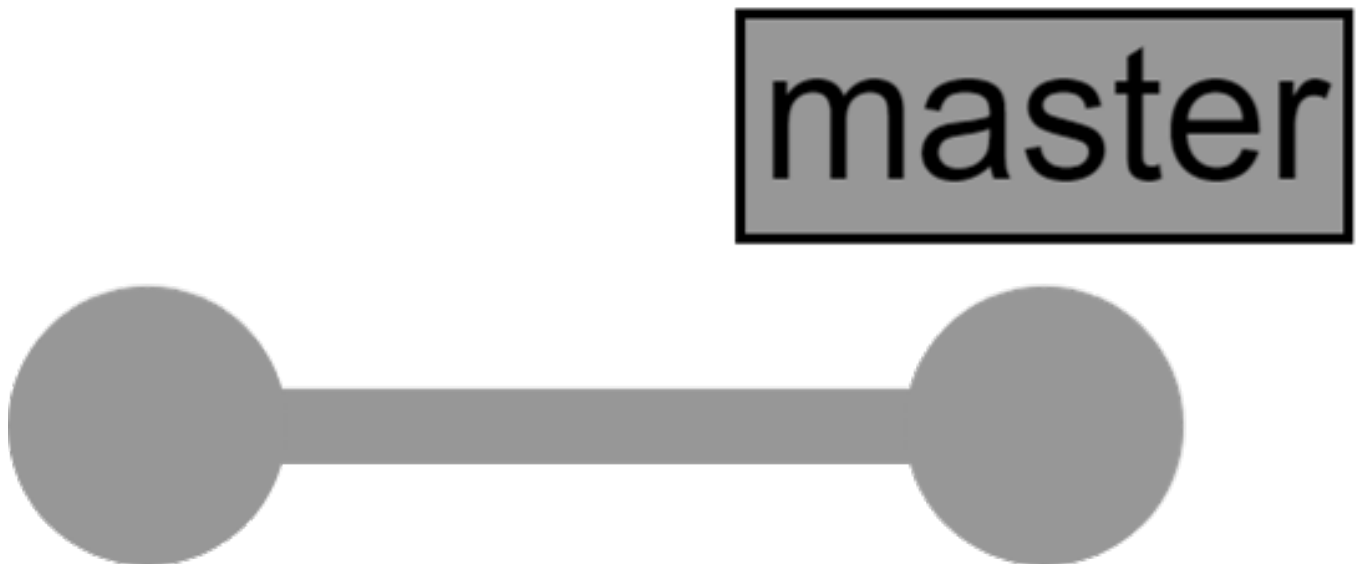
、通常は多く

の開発者が同時に別々の機

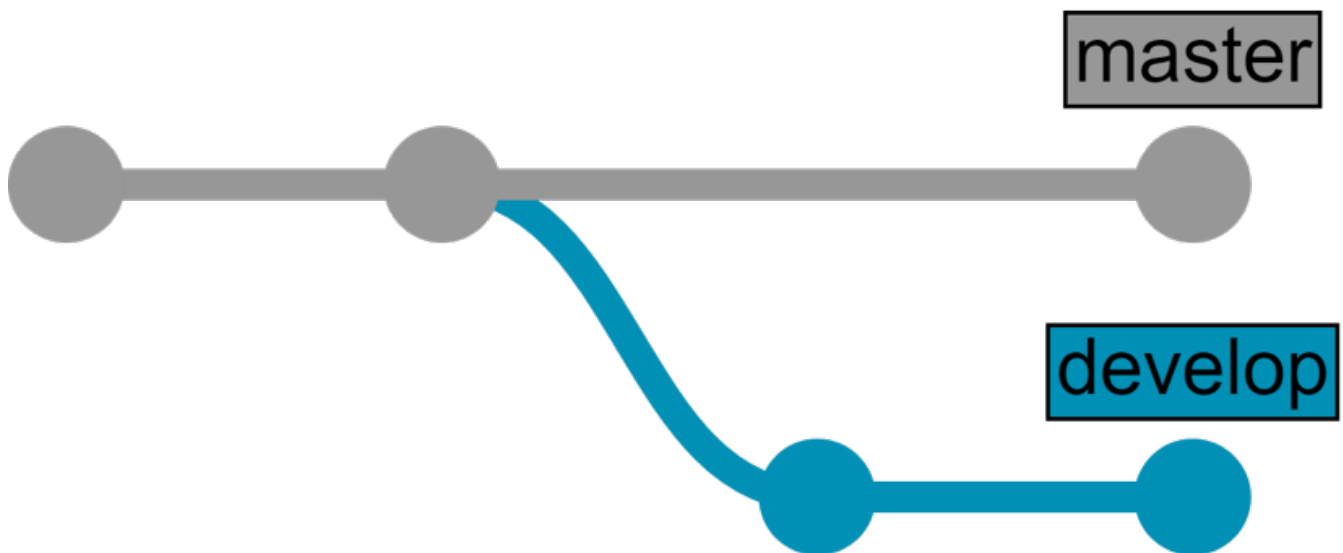
能を手がけているため、リポジトリの状況を表示するための**コミットツリー**が必要になります。

コミットツリー

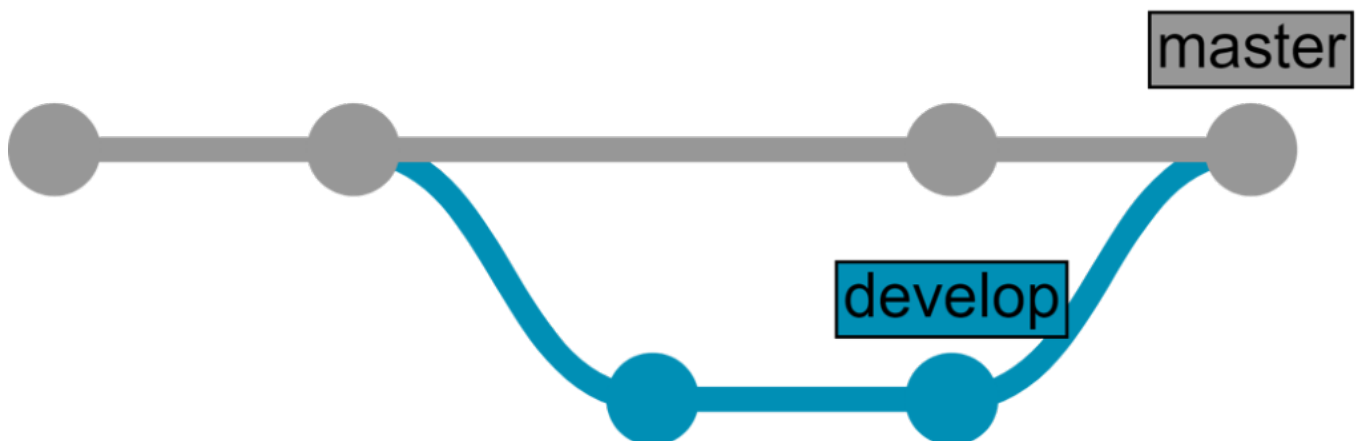
同じ出発点から始めましょう。2つのコミットを含むリポジトリを次に示します。



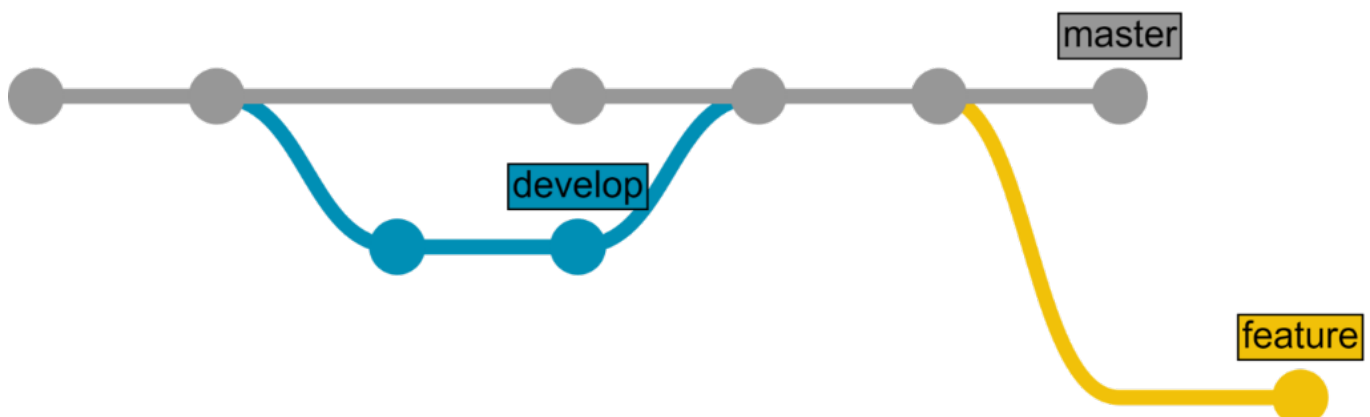
ただし、今度は2人の開発者が同時に作業しており、互いに干渉しないように別々のブランチで作業しています。



この二人はしばらく経ってからブランチへの変更内容を統合する必要がある、そのために**マージリクエスト**（**プルリクエスト**とも呼ばれています）を作成します。これはまさに文字通り2つの異なるリポジトリの状態を1つの新しい状態に統合する（この場合はdevelopブランチをmasterブランチに統合する）ためのリクエストです。適切なレビューと承認が行われた後、リポジトリは次のようになります。



その後も開発は継続されます。



Git 101 - 概要

主な概念：

- **Git**は直線的でない分散バージョン管理システムです。
- **リポジトリ**はデータとそのデータに関するメタ情報を保存します。
- **コミット**はリポジトリのある決まった状態を表しています。
- **ブランチ**はコミットへの参照です。
- **マージリクエスト（プルリクエスト**
とも呼ばれています）は、2つの異なるリポジトリの状態を1つの新しい状態に統合するためのリクエストです。

Gitについてさらに詳しく知りたい場合は、[こちらから書籍を入手できます](#)。

Gitフロー

Gitの基本的な用語と概念は理解していただけたかと思いますので、今度はGitを使用してソフトウェアライフサイクルの開発部分を管理する方法について説明しましょう。Gitを使用した開発プロセスを説明する多くのプラクティス（フローと呼ばれる）がありますが、ここではそのうち以下の2つについて説明します。

- GitHubフロー
- GitLabフロー

GitHubフロー

GitHubフローは簡単です。以下のとおりです。

1. リポジトリからブランチを作成します。
2. 自分の変更を新しいブランチにコミットします。
3. 自分が提案した変更を含むブランチからプルリクエストを送信して、ディスカッションを開始します。
4. 必要に応じてブランチでさらに変更をコミットします。プルリクエストは自動的に更新されます。
5. ブランチをマージする準備ができたなら、プルリクエストをマージします。

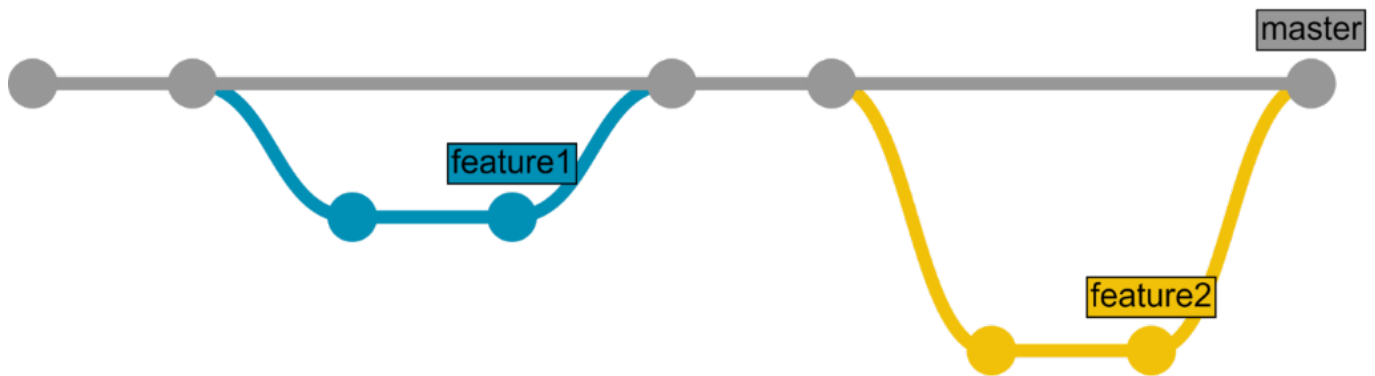
また、以下のようないくつかのルールに従う必要があります。

- masterブランチが常にデプロイ可能であること（また、機能すること！）
- masterブランチで直接開発が行われていないこと
- フィーチャーブランチで開発が進行中であること
- master はプロダクション（Production）* 環境** であること
- できるだけ頻繁に本番環境にデプロイすること

* 「Ensemble Productions」と混同しないでください。ここでは、「Production」とは本番を意味します。

** 環境とは、コードが実行される構成済みの場所です。サーバー、VM、コンテナなどが考えられます。

以下にそのイメージを掲載します。



GitHubフローの詳細については、[こちら](#)で確認できます。また、[図解ガイド](#)もあります。

Gitフローを使い始めるのであれば、GitHubフローを小規模なプロジェクトで試すのが最適です。GitHubもこれを使用していますが、大規模なプロジェクトでも実行可能です。

GitLabフロー

本番環境にすぐにデプロイする準備ができていない場合、GitLabフローはGitHubフローと環境を提供します。ここではその動作を説明します。上記と同じフィーチャーブランチで開発し、上記と同じマスターにマージとしましょう。しかし、その場合はmasterはテスト環境にのみ等しくなるという点が異なります。また、さまざまな他の環境にリンクされている「環境ブランチ」が存在する場合があります。

通常、以下の3つの環境が存在します（必要に応じてさらに作成できます）。

- テスト環境 == masterブランチ
- プリプロダクション環境 == preprodブランチ
- プロダクション環境 == prodブランチ

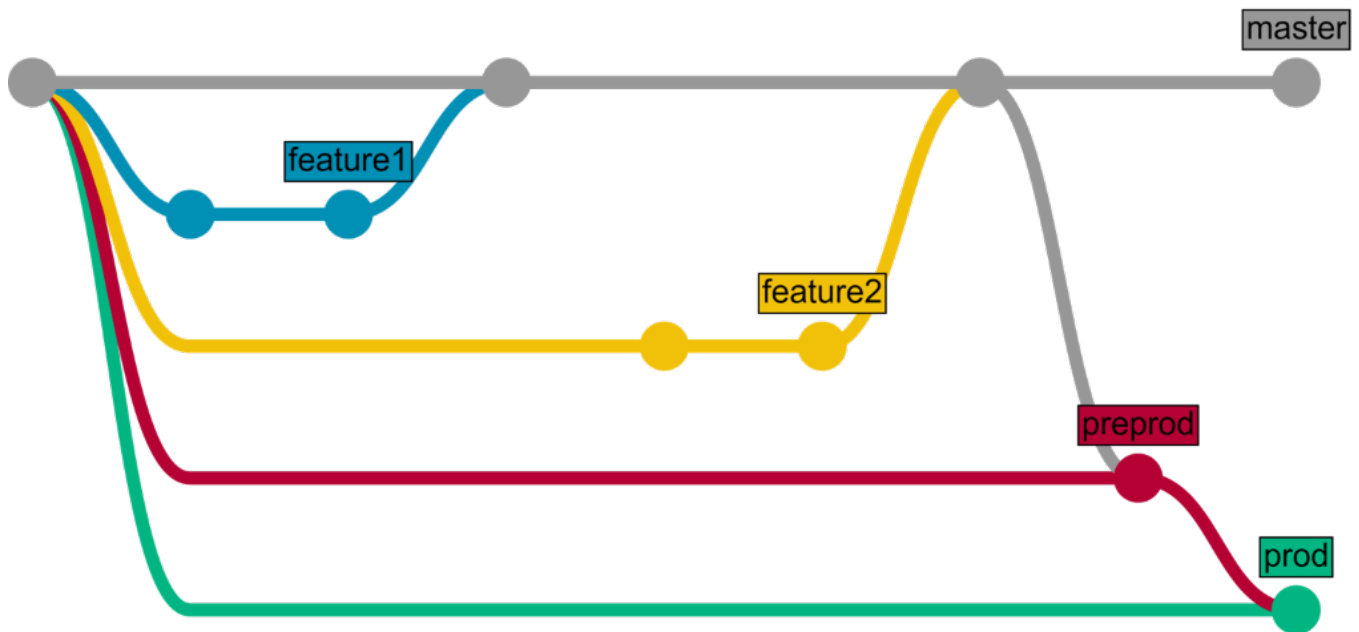
いずれかの環境ブランチに届いたコードは、対応する環境にすぐに移動する必要があります。以下のいずれかで実行できます。

- 自動（パート2およびパート3で対応します）
- 半自動（デプロイを許可するボタンを押すこと以外は自動と同じ）
- 手動

プロセス全体は次のようになります。

1. 機能はフィーチャーブランチで開発されます。
2. フィーチャーブランチがレビューされ、masterブランチにマージされます。
3. しばらくすると（いくつかのフィーチャーブランチがマージされた）マスターがpreprodにマージされます。
4. しばらくすると（ユーザーテストなどの後に）preprodがprodにマージされます。
5. マージおよびテスト中にいくつかの新機能が開発され、マスターにマージされます。そして3に戻ります。

これは次のようになります。



GitLabフローの詳細については、[こちら](#)で確認できます。

まとめ

- Gitは直線的でない分散バージョン管理システムです。
- Gitフローはソフトウェア開発サイクルのガイドラインとして使用できます。また、その中にはいくつかの選択肢があります。

リンク

- [Gitの書籍](#)
- [GitHubフロー](#)
- [GitLabフロー](#)
- [Driessenフロー](#)（より包括的なフロー、比較用）
- [この記事のコード](#)

ディスカッションの質問

- gitフローを使用していますか？ どのフローですか？
- 平均的なプロジェクトにはいくつの環境がありますか？

次の内容

次のパートの実施内容：

- GitLabをインストールします。
- いくつかの推奨される調整事項を紹介します。
- GitLabワークフローについて議論します（GitLabフローと混同しないこと）。

ご期待ください。

[#Git](#) [#継続的デリバリー](#) [#その他](#)

ソースURL:

<https://jp.community.intersystems.com/post/gitlab%E3%82%92%E4%BD%BF%E7%94%A8%E3%81%97%E3%81%9Fintersystems%E3%82%BD%E3%83%AA%E3%83%A5%E3%83%BC%E3%82%B7%E3%83%A7%E3%83%B3%E3%81%AE%E7%B6%99%E7%B6%9A%E7%9A%84%E3%83%87%E3%83%AA%E3%83%90%E3%83%AA%E3%83%BC-%E3%83%91%E3%83%BC%E3%83%88-i%E7%BC%9Agit>