

千葉工業大学 社会システム科学部
プロジェクトマネジメント学科
平成 26 年度 卒業論文

テストを基準にした OSS 開発プロジェクトにおける
タスク処理過程の定量分析

Quantitative analysis of the task processing process in the
OSS development project on the basis of a test

ソフトウェア開発管理コース
矢吹研究室

1142009 安藤 勇樹／Yuki Ando

指導教員印	学科受付印

目次

第 1 章	序論
1.1.	本章の構成 - 1 -
1.2.	研究背景 - 1 -
1.3.	研究目的 - 1 -
1.4.	研究方法 - 1 -
1.5.	プロジェクトマネジメントとの関連性 - 1 -
1.6.	参考文献 - 2 -
第 2 章	ソフトウェア開発について
2.1.	本章の構成 - 3 -
2.2.	チケットとは - 3 -
2.2.1.	チケットの登録 - 6 -
2.2.2.	チケットの一覧表示 - 7 -
2.2.3.	チケットの項目 - 8 -
2.3.	チケット駆動開発とは - 9 -
2.4.	参考文献 - 12 -
第 3 章	Git について
3.1.	本章の構成 - 13 -
3.2.	GitHub とは - 13 -
3.3.	GitHub の用語 - 13 -
3.4.	バージョン管理システムとは - 14 -
3.5.	Git とは - 17 -
3.6.	参考文献 - 33 -
第 1 章	GitHub について
4.1.	未定 エラー! ブックマークが定義されていません。

図目次

図 1	チケットの処理フロー	- 4 -
図 2	チケットの登録画面	- 6 -
図 3	チケットの一覧表示画面	- 7 -
図 4	集中管理方式	- 15 -
図 5	分散管理方式	- 16 -
図 6	git commit --amend.....	- 28 -
図 7	git rebase	- 29 -

表目次

表 1	チケットの項目.....	- 8 -
表 2	GitHub の用語一覧.....	- 13 -

第 1 章

序論

1.1. 本章の構成

本章では、本研究の背景・目的・方法・プロジェクトマネジメントとの関連性を記す。

1.2. 研究背景

ソフトウェア開発のためのホスティングサービスである GitHub では様々なソフトウェアが開発されている。2013 年 12 月には GitHub 上に 1000 万件のリポジトリが作成され、ユーザ数は 400 万人を超えた。数多くのプロジェクトが公開されている GitHub を調査すれば、ソフトウェア開発プロジェクトの分類が可能であると考えられる。

過去に GitHub 上のプロジェクトのチケットを調査し、プロジェクトを分類するという研究があり、プロジェクトの分類が可能であるということが明らかにされていた[2]。しかし、この研究では分類の解釈を人間が主観的に行っており、客観性に欠けているという問題があった。そのため、本研究ではデータマイニング手法を用いて分類を客観的に行う。

GitHub には、リポジトリの人気指標の 1 つにスターが存在する。スターとは、気になるリポジトリをブックマークできる機能である。このスターの数が多いリポジトリは人気が高いことを示している。本研究では、スター数の多いプロジェクトを調査する。

本研究では、プロジェクトを分類するためにチケットを調査する。チケットとは、ソフトウェア開発中に発生した課題やバグの内容を登録する進捗管理ツールである。チケットには、未完了チケットと完了済チケットの 2 種類が存在する。未完了チケットは作業が完了されていないチケットを示し、完了済チケットは作業が完了されているチケットを示す。チケットによって作業の進捗状況を可視化できるため、進捗管理が容易になる。

このチケットを中心に開発する手法をチケット駆動開発という。これは作業を開始する前に必ずチケットを発行することを原則とした開発手法である。この開発手法を運用しているプロジェクトは、未完了チケット数と完了済チケット数の時系列変化から進捗状況を判断できる。

1.3. 研究目的

GitHub 上のプロジェクトを対象とする。チケット数の時系列変化に着目し、データマイニング手法を用いてプロジェクトを分類する。

1.4. 研究方法

Issue (GitHub 上でのチケット) を GitHub 内のスター数ランキング上位 50 件のプロジェクトから API を用いて収集する。収集する Issue は OpenIssue (GitHub 上での未完了チケット) と CloseIssue (GitHub 上での完了済チケット) の 2 種類である。この 2 種類のチケットの時系列変化を調査し、プロジェクトを分類する。

1.5. プロジェクトマネジメントとの関連性

チケットは、ソフトウェア開発中に発生した課題やバグの内容を可視化し、進捗管理を容易にできる。これは PMBOK が提唱するスコープマネジメントに関連する。プロジェクトマネジメントにおいて進捗管理は重要であり、この作業を効率化できるチケットを調査することは有用であると考えられる。

1.6. 参考文献

- [1] Ingrid Lunden. "GitHub Hits The 4M User Mark As It Looks Beyond Developers For Its Next Stage Of Growth". TechCrunch. 2013-9-7.
<http://techcrunch.com/2013/09/11/github-hits-the-4m-user-mark-as-it-looks-beyond-developers-for-its-next-stage-of-growth/>, (参照 2014-9-1).
- [2] Briandoll. "10 Million Repositories". GitHub. 2013-12-23.
<https://github.com/blog/1724-10-million-repositories>, (参照 2014-9-1).
- [3] 小川明彦, 酒井誠. チケット駆動開発. 翔泳社, 2012-8-23.
- [4] 久保孝樹. チケットを活用するオープンソースソフトウェア開発の実態調査. 千葉工業大学, 2013, 卒業論文.
- [5] Project Management Institute, Inc. プロジェクトマネジメント知識体系ガイド(第4版). PMI, 2009-12.

第 2 章

チケットについて

2.1. 本章の構成

本章では，本研究で調査するチケットの基本知識と使用方法について記述する．また，チケットを中心とした開発手法であるチケット駆動開発の基本知識と使用方法についても記す．

2.2. チケットとは

チケットとは，課題管理システム（Issue Tracking System ; ITS）やバグ管理システム（Bug Tracking System ; BTS）で使用される進捗管理ツールである．1つの課題やバグを管理する単位をチケットと呼び，ソフトウェア開発中に発生した課題やバグの内容を記述する．チケットには，未完了チケットと完了済チケットの2種類が存在する．未完了チケットは作業が完了されていないチケットを示し，完了済チケットは作業が完了されているチケットを示す．以下にチケットの処理フローを記載する．

- 検証担当者
アプリケーションの検証を行い，バグを発見した際にチケットを作成します．また，修正結果に対しての再検証を行う．
- 修正担当者
バグの詳細が記述されたチケットを受け，アプリケーションの修正を行う．
- 管理者
担当者のアサイン，検証結果の承認を行う．

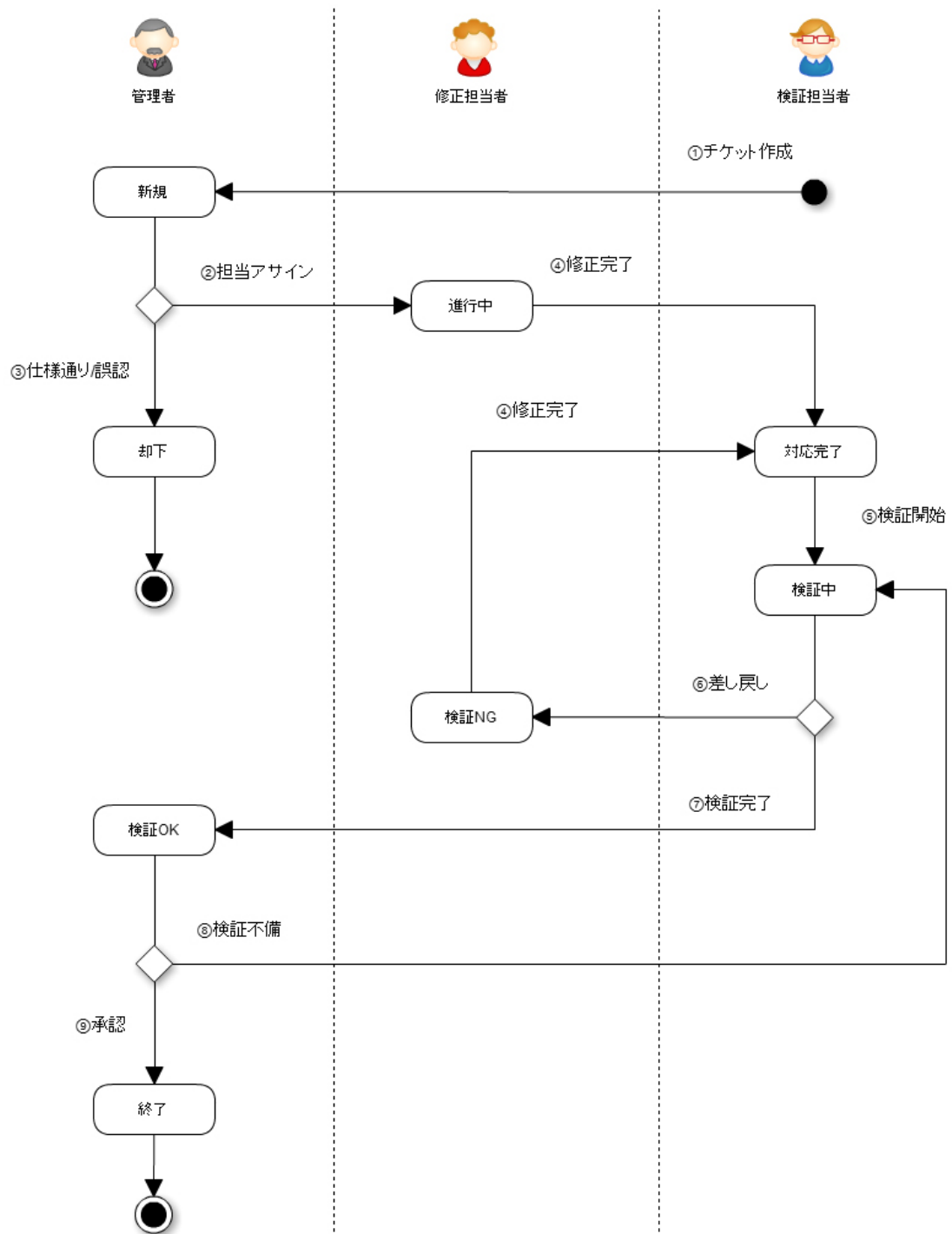


図 1 チケットの処理フロー

1) チケット作成 [→ 新規]

検証担当者はバグを発見すると、バグの詳細を記述したチケットを新規に作成します。作成の際に担当者を管理者にすることで、バグが発見された通知が管理者に送られます。

2) 担当アサイン [新規 → 進行中]

管理者はチケットに記述されたバグ情報を確認し、修正担当者をアサインします。

3) 仕様通り／誤認 [新規 → 却下]

チケットに記述されたバグ情報を確認し、そもそも仕様通りの場合や検証担当者の誤認だった場合は対応不要のため、理由を記述したうえでチケットのステータスを"却下"に変更し、終了します。

4) 修正完了 [進行中 → 対応完了] [検証 NG → 対応完了]

修正担当者はバグの修正対応を行い、その対応の詳細をチケットに記述します。再度検証を行うため、担当者をテスト担当者に変更します。

5) 検証開始 [対応完了 → 検証中]

対応完了になっているチケットから、検証を開始するチケットを選択し「検証中」に変更します。

6) 差し戻し [検証中 → 検証 NG]

バグの修正対応が不十分であった場合、「検証 NG」として差し戻しを行います。

7) 検証完了 [検証中 → 検証 OK]

バグの修正対応に問題がないと確認できた場合「検証 OK」とし、管理者に最終承認を行ってもらいます。

8) 検証不備 [検証 OK → 検証中]

検証結果を確認し、検証内容に不備があった場合や、追加で検証を行ってほしい項目などがあれば、検証担当者に追加の検証を行ってもらいます。

9) 承認 [検証 OK → 終了]

検証結果に問題がなければ承認とし、チケットのステータスを「終了」にします
[1].

2.2.1. チケットの登録

チケットは、ソフトウェア開発中に課題やバグが発生した場合に登録される。チケットの登録はプロジェクトのメンバが行え、登録されたチケットはメンバ間で閲覧できる。チケットの項目には、題名・作業内容・ステータス・優先度・担当者・期日などがあり、チケットを登録する際に記述できる。また、Label を設定することで、チケットがどのような種類であるのかを指定できる。以下の図は、チケットと同じ役割を持つ GitHub の Issue である。

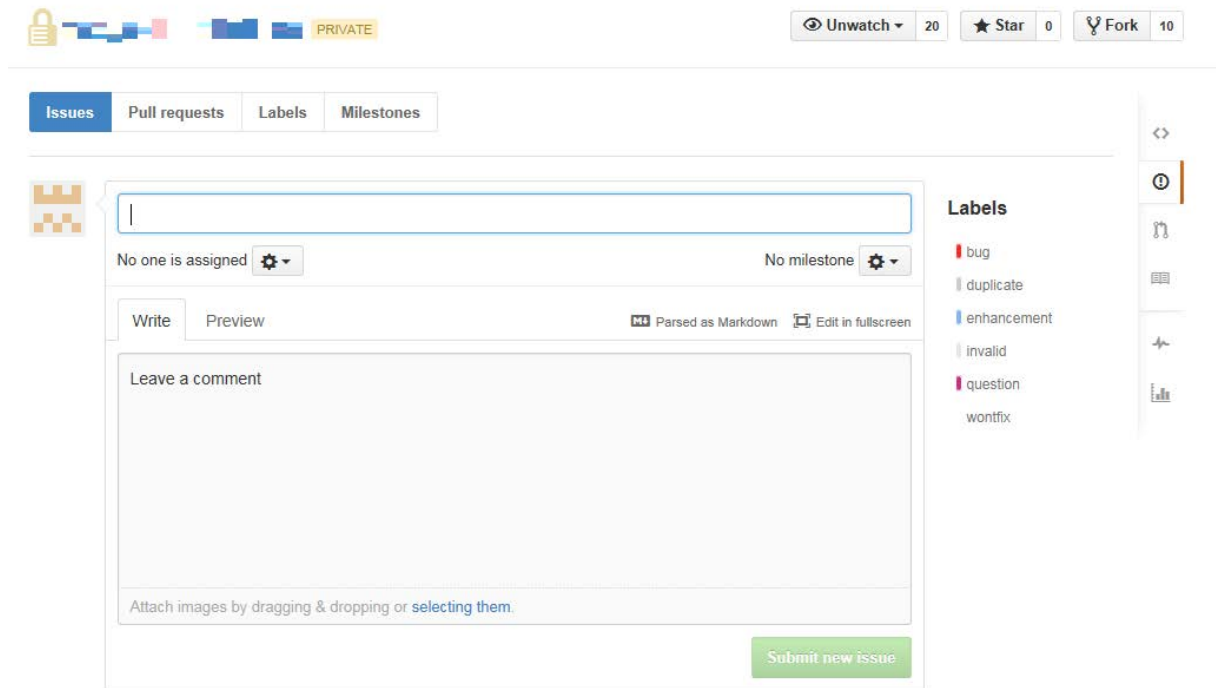


図 2 チケットの登録画面

2.2.2. チケットの一覧表示

登録されたチケットは、検索機能やソート機能によって検索・並び替え・一覧表示が可能である。例として、Label を利用した種類ごとの分類表示やキーワード検索、優先度順に並び替えるなどが可能である。一覧表示では、Open と Close を別々に表示できるため、作業の進捗状況を把握できる。以下の図は、チケットと同じ役割を持つ GitHub の Issue の一覧表示画面である。

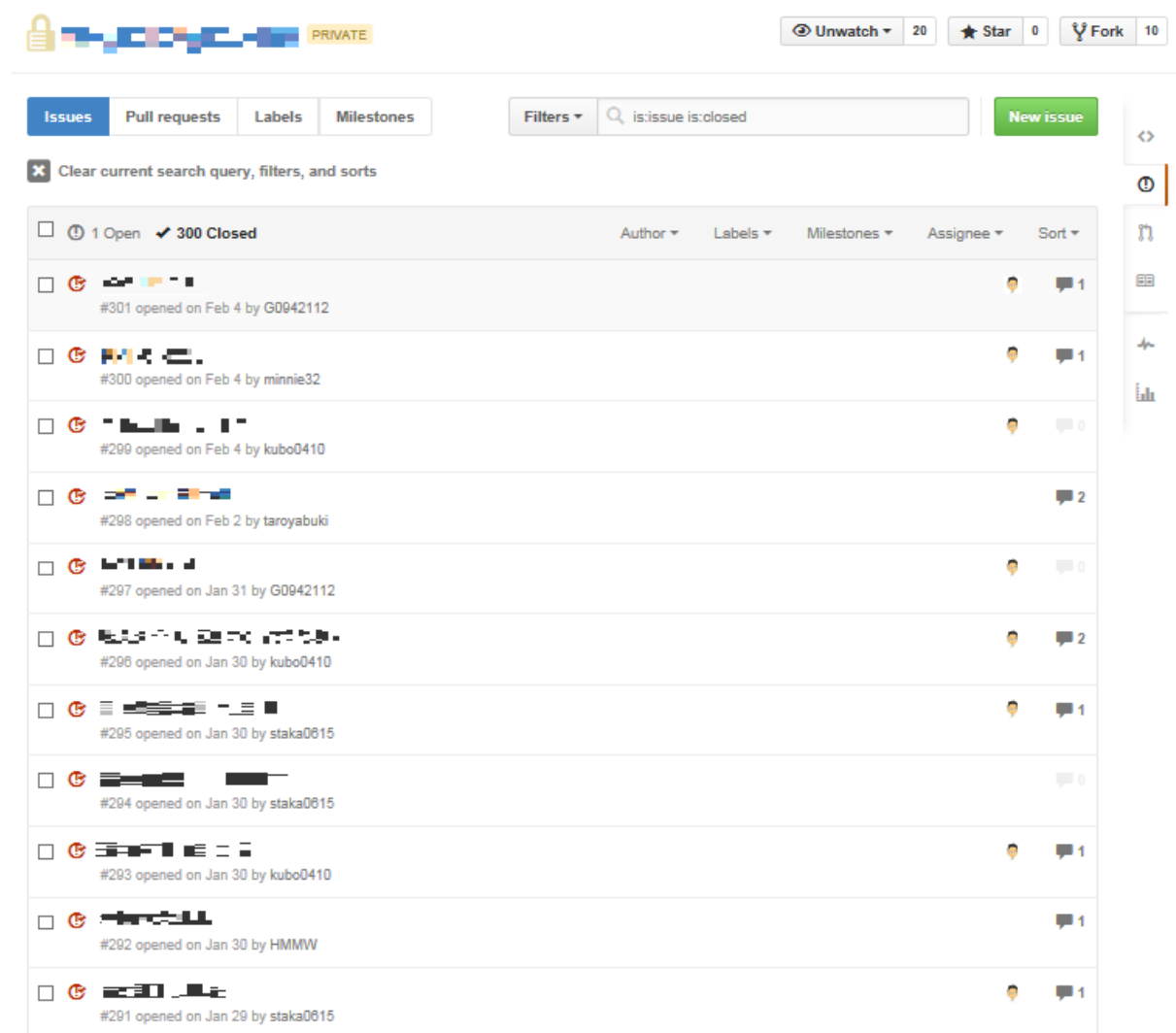


図 3 チケットの一覧表示画面

2.2.3. チケットの項目

チケットには様々な項目が存在し、登録する際に記述できる。これらの項目により、チケットがどのような理由から登録されたのかを把握できる。以下の表は、チケットの項目である。

表 1 チケットの項目

項目名	内容
Label	チケットがどのような種類であるのかを表す。種類には、バグ・重複・機能追加・質問・対応無しなどがある。
題名	チケットの一覧表示画面などに表示される。チケットの内容を端的に表すものを記述する。
作業内容	題名だけでは書ききれない詳細な作業内容を記述する。
ステータス	チケットの進捗状況を表し、Open であるか、Close であるかを記述する。
優先度	どのチケットから優先的に着手すべきかを明示できる。チケットの優先度を記述する。
登録者	チケットを登録したメンバの名前を記述する。
担当者	チケットを処理すべき担当者の名前を記述する。
チケット番号	チケットが登録された際に割り振られる番号である。何番目に登録されたチケットなのかを表す。
対象バージョン	登録されたチケットをどのバージョンに関連付けるのかを指定する。
進捗率	作業の進捗率を記述する。
開始日	作業を開始すべき日付を記述する。
期日	作業の期日を記述する。

2.3. チケット駆動開発とは

チケット駆動開発 (Ticket-Driven Development ; TiDD) とは、チケット管理から生まれたプロジェクト管理技法の 1 つである。ITS/BTS のチケットを用いて作業を管理し、作業を開始する前に必ずチケットの登録をすることを原則とした開発手法である。作業管理にチケットを用いることで、作業漏れの防止や頻繁な作業の変更に対応できるようになる。チケット駆動開発には、以下の様な規律が存在する。

- プラクティス 1. チケット無しのコミット不可 (No Ticket, No Commit)

プログラムなどの成果物を変更する場合、必ずチケットに変更履歴を残します。バージョン管理のコミットフック機能(post-commit-hook)を利用して、プログラムをコミットする時、必ずコミットログにチケット番号を記録する運用から生まれた基本ルールです。作業の変更管理を強化し、プログラムの変更履歴をチケット経由で追跡できる利点(traceability)があります。また、開発者は、1 日の作業の開始前にチケットを登録(No Ticket, No Work)して、チケットに基づいてプログラム修正を行い、プログラムのコミットと同時にチケットを完了するという開発のリズムが生まれます。

- プラクティス 2. チケット無し作業不可(No Ticket, No Work)

作業を開始する前に、必ずチケットに作業内容を登録してから作業を開始します。そして、チケットは仕様書ではなく、作業指示書になります。作業履歴がチケットに必ず残るため、他の開発者が参照できたり、収集したチケットから予定・実績の工数集計、進捗・障害の集計結果が得られるため、原因分析や是正対策作りに役立てることができます。

- プラクティス 3. イテレーションはリリースバージョンである (Iteration is Version)

イテレーションはソフトウェアのリリースバージョン、プロジェクトのマイルストーンと一致させます。XP のイテレーション(Scrum のスプリント)をバージョン管理の配下にあるブランチのリリースバージョンと同一視することで、イテレーション終了時には必ずソフトウェアをリリースできるというリリース完了条件が導かれます。また、開発チームは、イテレーション単位に定期的に小規模リリースしていくことによって、システムを漸進的に開発していくリズムが生まれます。

- プラクティス 4. チケットは製品に従う

ITS プロジェクトというチケット管理の集合は、リリースされる製品の構成(アーキテクチャ)に従属するように対応付けます。機能追加や障害修正のチケットはソースに対して管理する(No Ticket, No Commit)ため、製品の構成に基づくバージョン管理リポジトリと対応付けるように、ITS プロジェクトを作る方が管理が楽になります。そして、Conway's Law(組織はアーキテクチャに従う)に従えば、開発組織はチケット管理の階層構造に組み込まれて、チケット駆動で開発しやすい組織体制の変化が促されます。

パッケージ製品を顧客ごとにカスタマイズする派生開発や、複数の製品系列の開発であるソフトウェアプロダクトライン(SPLE)にも適用できます。

- プラクティス 5. タスクはチケットで分割統治(Divide and Conquer)

チケットの粒度が大きい場合、タスクの分割という観点から、開発者が作業しやすいようにチケットを分割します。チケット駆動で運用するといつも迷ってしまうのは、チケットの粒度です。基本的な運用方針としてはチケットの粒度は 1 ～ 5 人日程度まで細分化します。なぜならば、1 日の開発リズムが生まれやすいため、開発者は作業しやすいからです。従って、チケットの作業内容が大きすぎると気づいたら、たとえチケットが登録された後でも、作業しやすい粒度までチケットを分割します。

- プラクティス 6. チケットの棚卸し

定期的に、放置された未完了チケットを精査して最新化します。チケット駆動を中心にプロジェクトを運営すると、チケットはどんどん登録され更新されるため、何らかの規律がなければ、乱発・放置されるチケットの重みで開発速度が遅くなってしまいます。その状況を打開するために必要な作業である「チケットの棚卸し」には以下の 4 つの注意点があります。

- 1) 役割分担

開発チームのリーダーがチケット管理の最終責任者であると認識し、リーダーは定期的に、誰も手を付けない作業やチームの開発の支障となる課題を優先順位付けしたり、ステータスを最新化したりします。

- 2) 棚卸しのタイミング

例えば、毎日の朝会やリリース後のふりかえりミーティングのように、棚卸しのタイミングを故意に作ります。

- 3) チームの開発速度(Velocity)を超えるチケットは後回し

現状のメンバーの技術力やチームの成熟度の観点では、どう考えても実施不可能なほど大量のチケットであふれている状況があります。その場合、チームの開発速度 (Velocity)を超えるチケットはイテレーションから削減し、チームが消化できるレベルのチケットの枚数になるように調整します。

- 4) 作業不要のチケットは、リリース未定の特別なイテレーションへ移す

作業不要のチケットは、別のイテレーションへ延期したり、リリース期限が未定の特別なイテレーションへ移動して除去します。この特別なイテレーションは、「Unplanned」「バックログ」「Icebox」などとも呼ばれ、次のイテレーション計画作成中に必要なチケットがあると判断されれば、そのチケットを取り込む運用になります。

- プラクティス 7. ペア作業(Pair Work)

一つのチケットを二人以上で連携する作業です。XP のペアプログラミングでは必ず二人が同時時間帯に一つの机で作業しますが、ペア作業では一つのチケットの作業結果を受けて、チケットのステータスを更新しながら非同期にペアで作業します。例えば、障害修正において開発者とテスト担当者が交互に修正と検証を行ったり、コードレビューをレビューアと開発者(レビューイ)が交互にレビューと指摘事項の反映を行ったりする時に使われます。一つのチケットを二人以上の目を通してチェックし、成果物の品質を向上できる利点があります[2]。

2.3.1. チケット駆動開発の開発サイクル

チケット駆動開発では、概ね次のような PDCA サイクルを繰り返して開発が行われる。

- 1) 大まかなリリース計画を作る。
- 2) 仕事を細かいタスクに分割し、タスクを書き出す。(チケットの発行)
- 3) イテレーション単位でタスクをまとめて、イテレーション計画を作る。
- 4) タスクを一つ選び、実装する。
- 5) 差分をコミットし、完了する。(チケットのクローズ)
- 6) イテレーションに紐づくタスクがすべて終了ステータスになるとリリースする。
- 7) リリース後、開発チームで作業をふりかえる。
- 8) 次のイテレーション計画へ顧客の要望やふりかえりの内容を反映する[3]。

2.3.2. チケット駆動開発のメリット

- 作業管理が容易

作業の全てをチケット経由で行うため、作業状況を可視化でき、明確に作業管理ができる。これにより、作業漏れや頻繁な作業の変更に対応できる。また、各担当者がメンバの作業を閲覧できるため、メンバ間での情報共有が容易に行える。

- 進捗管理が容易

チケットに進捗率や期日を記述できるので、作業の進捗状況を把握しやすい。また、優先度を記述することにより、優先的に着手しなければならない作業を明示できる。

- 変更履歴の管理が可能

ドキュメントやソースコードの変更履歴をチケットに連動できるため、変更履歴を管理できる。これにより、変更後のデータに不具合が見つかった場合でも、履歴をたどることで変更前のデータに戻すことができる。

- マネジメントが容易

作業を割り振られているメンバごとにチケットを閲覧できるので、リソースの空き状況を把握しやすい。

2.3.3. チケット駆動開発のデメリット

- チケットの乱立

登録されたチケットが完了されずに未完了チケットばかりが増加すると、管理するチケット数が膨大な量になり、マネジメントや進捗管理に支障をきたす。

- チケットの登録作業

作業を開始する前に必ずチケットを登録しなければならないため、進捗状況を文章化する作業が増える。

- チケット駆動開発の理解

チケット駆動開発についての知識がない場合、開発作業とは別にチケットの登録方法などを学習する必要がある。また、混乱を防ぐためにもチケットの粒度や必須項目についてメンバ間でルールを決める必要がある。

2.4. 参考文献

- [1] 天沼健仁, TIS 株式会社. “ガチで 5 分で分かる ITS/BTS&使えるツール 6 選(3/7)”. @IT. 2013-06-27. http://www.atmarkit.co.jp/ait/articles/1306/26/news012_3.html, (参照 2014-09-26).
- [2] Sean Osawa. “チケット駆動開発を上手に運用するためのプラクティス (ゲストブログ)”. Atlassian Blogs. 2012-10-02. <http://japan.blogs.atlassian.com/2012/10/tidd-part2/>, (参照 2014-09-26).
- [3] “チケット駆動開発 - Wikipedia”. ウィキペディア(Wikipedia):フリー百科事典. 2012-10-19. <http://ja.wikipedia.org/wiki/%E3%83%81%E3%82%B1%E3%83%83%E3%83%88%E9%A7%86%E5%8B%95%E9%96%8B%E7%99%BA>, (参照 2014-9-26).
- [4] 小川明彦, 酒井誠. チケット駆動開発. 翔泳社, 2012-8-23.

第 3 章

GitHub について

3.1. 本章の構成

本章では、GitHub と Git の基本知識や使用方法について記す。また、GitHub の API についても記す。

3.2. GitHub とは

バージョン管理システム Git のプロジェクトをインターネット上で共有・公開することができるネットサービスの一つ。同名の企業(GitHub 社)が運営している。オープンソースソフトウェアの開発プロジェクトなどでソースコードの管理や公開によく用いられ、最も人気の高い Git ホスティングサービスの一つである。

Git で保管・管理するデータの集積(リポジトリ)を GitHub の運用するサーバ上に集積し、組織内や複数人で共有したり、広く一般に公開したりすることができる。リポジトリは Git を用いて操作できるほか、Web サイト上にも情報が公開され、Web ブラウザを通じて閲覧や操作ができる。サイト上では利用者(開発者)間のコミュニケーションが可能で、一種の SNS としても機能している。

パブリックリポジトリ(一般に公開されるリポジトリ)は無料で作成することができるが、プライベートリポジトリ(企業内プロジェクトなどで利用するための非公開のリポジトリ)の作成・利用には月額料金がかかる[1]。

3.3. GitHub の用語

GitHub には開発を効率的に行うための機能が多く存在する。その中で使われる用語を一部抜粋し、以下に記述する。

表 2 GitHub の用語一覧

用語名	内容
Repository (リポジトリ)	アプリケーションやシステム情報などのデータが保持されている貯蔵庫のようなもの。ファイルやディレクトリなどをオブジェクトとして表現する。
Clone (クローン)	サーバに存在するリモートリポジトリを手元の PC (ローカル) にコピーする。
Fork (フォーク)	GitHub 上で公開されているリポジトリを自分のリポジトリとして複製する。
Origin (オリジン)	Clone 元のリモートリポジトリを示す。
Current directory (作業ディレクトリ)	ユーザが作業しているディレクトリを示す。
Index (ステージング) (ステージングエリア)	ステージングとは、変更したデータのうち、コミットする対象を選別する作業である。ステージングエリアに登録することを「ステージングする」と呼ぶ。 ステージングエリアとは、Git ディレクトリに含まれる、次のコミットに関しての情報が保持された 1 つのファイルである。

Git directory (Git ディレクトリ)	Git リポジトリの全ての変更履歴を保持しているディレクトリを示す。
Revert (リバート)	ステージングエリアに追加した変更を戻す。
Commit (コミット)	ステージングエリアに追加した変更を Git ディレクトリ上に変更履歴として反映する。
Branch (ブランチ)	メインの流れとは異なる流れで作業をする仕組みである。メインの流れでバグ修正や機能改善を行い、異なる流れで新機能の開発を行う。
Merge (マージ)	異なるブランチを 1 つのブランチに統合する。
Push (プッシュ)	ローカルリポジトリに追加した変更履歴をリモートリポジトリに反映する。
Pull (プル)	リモートリポジトリに追加された変更履歴をローカルリポジトリに反映する。
Pull Request (プルリクエスト)	Fork したリポジトリに対して変更を加え、その変更を Fork 元のリポジトリに反映してもらうようにリクエストを送る作業である。
Issue (イシュー)	開発中に発生したバグや課題を記述し、登録する機能。1 つのバグや課題に 1 つの Issue が割り当てられる。プロジェクトメンバー間での共有が可能であり、コメント、削除などの操作ができる。チケット駆動開発のチケットと同様な役割を持つ。
Star (スター)	ソーシャルブックマークと同様な機能であり、スターを付けたリポジトリを即座に閲覧できるようになる。

3.4. バージョン管理システムとは

バージョン管理システムとは、ファイルに対して「誰が」「いつ」「何を変更したか」というような情報を記録することで、過去のある時点の状態を復元したり変更内容の差分を表示できるようにするシステムのことです。バージョン管理システムは大きく 2 つに分けると、「集中管理方式」「分散管理方式」があります。

過去には集中管理方式の「CVS」「Subversion」が多く利用されていましたが、複数人での分散開発の容易さやパフォーマンスに優れた分散管理方式の「Git」「Mercurial」などがスタンダードになりつつあります[2]。

3.4.1. 集中管理方式「Subversion(SVN)」

SVN は Git が登場する前から使われている集中型のバージョン管理システムです。SVN よりも先に公開されたバージョン管理システムに「CVS」がありますが、SVN は CVS を参考にして開発されました。まずは SVN について見ていきます。

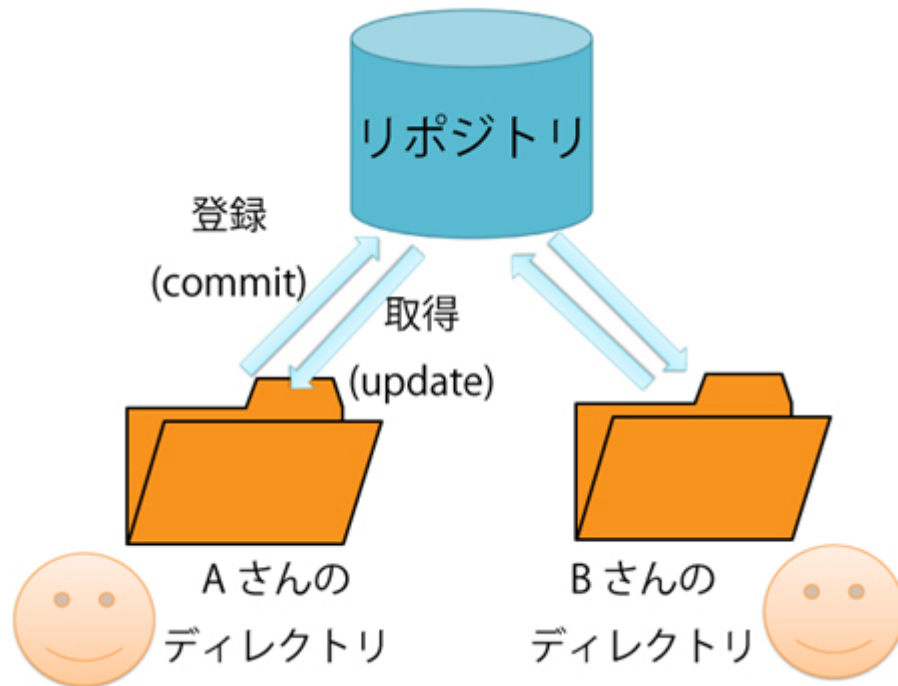


図 4 集中管理方式

SVN は集中型(クライアント・サーバ型)のバージョン管理システムです。ファイルそのものや変更の履歴などを保存する場所を「リポジトリ」(貯蔵庫)と呼びますが、SVN の場合は(ソフトウェア 1 つにつき)1 つのリポジトリを使います。ソフトウェア開発に参加するメンバーは、中央リポジトリ(プロジェクトメンバー間で共有するリポジトリ)からソースコードを持ってきて編集し、編集が終わったら中央リポジトリに直接反映します。SVN は集中型のバージョン管理システムなので、リポジトリが置かれたサーバに接続できない環境の場合、最新のソースコードを取得やファイル編集の反映ができません [3]。

3.4.2. 分散管理方式「Git」

集中型バージョン管理システムの SVN に対し、Git は分散型のバージョン管理システムです。リポジトリを複数持つことができ、開発の形態や規模に合わせてソースコードの管理ができます。リポジトリを複数用意できるので「分散型」と呼ばれています。

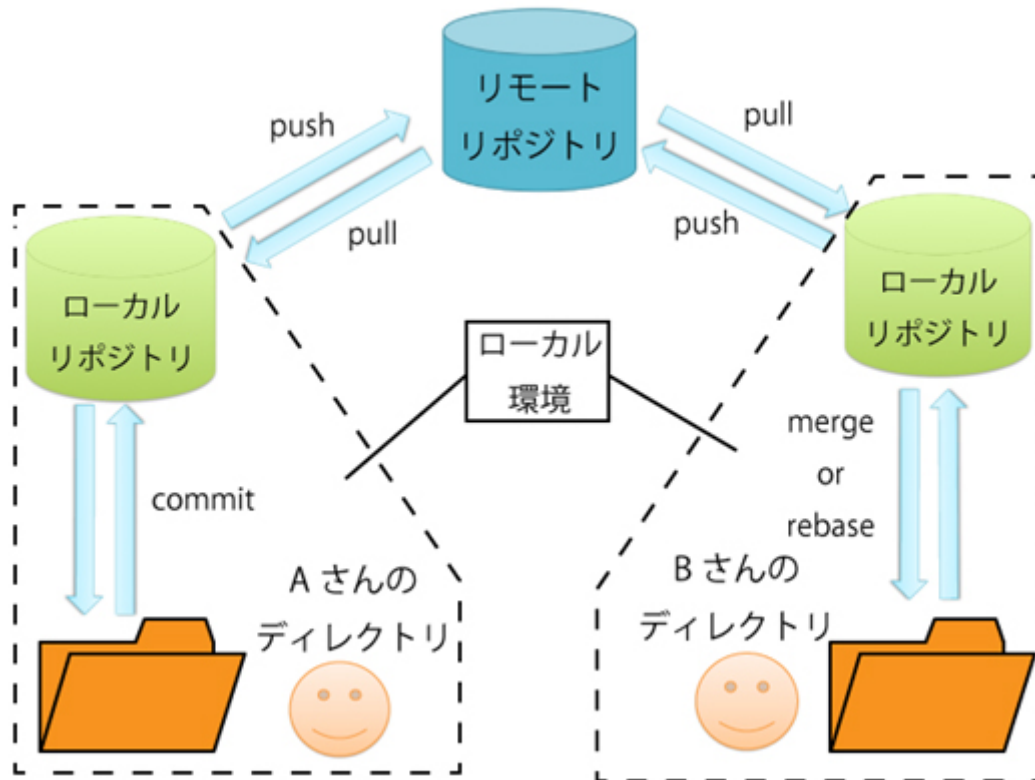


図 5 分散管理方式

例えば上の図のように、リモートリポジトリをサーバ上に置き、開発者それぞれがローカルにリポジトリを持つという構成が考えられます。この場合、普段はローカルリポジトリを使って作業し、ある程度作業できたらリモートリポジトリに反映するといった使い方ができます。リモートリポジトリにアクセスできない環境でも作業を進めることができます。また、大きな規模のソフトウェアの場合、個人や小さなチームで試験的に実装を進めて、ある程度進んだ時点で親のリポジトリや他の開発者のリポジトリに反映するといった使い方ができます。開発者同士の作業を柔軟に進めることができ、オープンソースのプロジェクトなどにも適しています[3]。

3.5. Git とは

オープンソースの分散バージョン管理システムの一つ。複数の開発者が共同で一つのソフトウェアを開発する際などに、ソースコードやドキュメントなどの編集履歴を管理するのによく用いられる。

バージョン管理システムの基本的な機能として、個々のファイルにいつ誰がどのような変更を行ったかを記録しており、必要に応じて特定の日時の版を参照したり、その状態に戻したりすることができる。また、プロジェクトの時系列を分岐(ブランチ)して派生プロジェクトを作成したり、それを再び元の系列に融合(マージ)したりすることができる。

Git の特徴として、管理するデータの集合体であるリポジトリを利用者の手元のコンピュータに丸ごとコピーし、必要に応じて中央の管理サーバと同期する方式を取っている。これにより、常にサーバと通信可能でなくても編集が可能で、検索や参照なども高速に行なうことができる。

GitHub のようにインターネット上のサーバに Git の中央リポジトリを作成・運用できるサービスもあり、GitHub に作成したリポジトリに各開発者が Git でアクセスして開発を進めるというスタイルがオープンソースソフトウェアなどで人気となっている [4]。

3.5.1. Git コマンド

GitHub は Git ホスティングサービスであるため、リポジトリの作成・運用を Git コマンドで行う。以下に、リポジトリを操作するうえで重要である Git コマンドを記載する。

- git init

git init は Git リポジトリを新たに作成するコマンドです。このコマンドは、バージョン管理を行っていない既存のプロジェクトを Git リポジトリに変換する場合や、空の新規リポジトリを作成して初期化する場合に使用します。このコマンドを除く他のコマンドはほとんどすべて初期化されたリポジトリ以外には適用することができないため、このコマンドは新規プロジェクトを開始する場合に通常最初に実行するコマンドです。

git init コマンドを実行すると、リポジトリに付随するすべてのメタデータを有する .git サブディレクトリがプロジェクトのルートに作成されます。この .git ディレクトリを除けば、既存プロジェクトには何の改変も行われません(SVN とは異なり、Git では各々のサブディレクトリに .git フォルダが作成されることはありません)。

git init

カレントディレクトリを Git リポジトリに変換します。このコマンドを実行するとカレントディレクトリに .git フォルダが作成され、プロジェクトのバージョンの管理を開始することができます。

```
git init <directory>
```

指定したディレクトリに空の Git リポジトリを作成します。このコマンドを実行すると、.git サブディレクトリのみを含む<directory>という名称の新規フォルダーが作成されます。

```
git init --bare <directory>
```

作業ディレクトリを持たない空の Git リポジトリを作成して初期化します。共有リポジトリは必ず--bare フラグを指定して作成しなければなりません(下の補足説明参照)。--bare フラグを指定して作成したリポジトリの名称の最後には習慣的に.git を付加します。例えば、my-project という名称のリポジトリのベアバージョンは、my-project.git という名称のフォルダーに格納します。

- git clone

git clone は、既存の Git リポジトリのコピーを作成するコマンドです。このコマンドは svn checkout と似ていますが、作業コピーがそれ自身で完全な Git リポジトリを構成する点が異なっていて、即ち作業コピーは自分自身の履歴を持ち、自分自身でファイルを管理し、元のリポジトリとは完全に独立した環境を提供します。

利用者の便宜のため、クローンを行うと、元のリポジトリをポイントする origin という名称のリモート接続を自動的に作成します。これにより、極めて簡単に中央リポジトリとの通信を行うことができます。

```
git clone <repo>
```

<repo>にあるリポジトリをローカルマシンにクローンするコマンドです。元のリポジトリはローカルマシンに存在しても HTTP や SSH を用いてアクセスするリモートマシンに存在しても構いません。

```
git clone <repo> <directory>
```

<repo>にあるリポジトリをローカルマシン上の<directory>という名称のフォルダーにクローンするコマンドです。

- git config

git config は、インストールした Git(または個々のリポジトリ)に対してコマンドラインから設定を行うコマンドです。このコマンドを使用すると、ユーザー情報からリポジトリ動作の初期設定にいたるまであらゆる項目の設定が可能です。通常使用される設定オプションのいくつかを以下に示します。

```
git config user.name <name>
```

現在のリポジトリにおけるすべてのコミットに使われるオーサー名を設定します。このコマンドでは通常、--global フラグを指定して現在のユーザーを対象としたオーサー名設定を行ないます。


```
git config --global user.name <name>
```

現在のユーザーが行うすべてのコミットのオーサー名を設定します。

```
git config --global user.email <email>
```

現在のユーザーが行うすべてのコミットに関してそのオーサーE メールアドレスを設定します。

```
git config --global alias.<alias-name> <git-command>
```

Git コマンドのショートカットを作成します。

```
git config --system core.editor <editor>
```

現在使用しているマシンにおいて `git commit` のようなコマンドを実行する際に使用するエディターを指定します。引数<editor>は、該当のエディターを起動するコマンドです(例えば、`vi` エディターなど)。

```
git config --global -edit
```

グローバルな設定ファイルをテキストエディターで開くコマンドで、これを使用して手動で設定ファイルを編集することができます。

- `git add`

`git add` は、作業ディレクトリ内の変更をステージングエリアに追加するコマンドです。このコマンドは、個々のファイルのアップデート内容を次回コミットの対象とすることを `Git` に指示します。ただし、`git add` コマンドだけでは実際にはローカルリポジトリに何の影響も与えず、`git commit` コマンドを実行するまでは変更が実際に記録されることはありません。

これらのコマンドと関連して、作業ディレクトリおよびステージングエリアの状態を確認するために、`git status` コマンドが用いられます。

```
git add <file>
```

<file>に加えられたすべての変更をステージして次回のコミットの対象とします。

```
git add <directory>
```

<directory>内のすべての変更をステージして次回のコミットの対象とします。

```
git add -p
```

インタラクティブなステージングセッションを開始します。インタラクティブなステージングセッションでは、ファイルの一部を選択してステージし、次のコミットの対象とすることができます。このコマンドを実行すると、変更部分が表示され、それに対する次のコマンド入力を要求されます。y を入力するとその部分がステージされ、n を入力するとその部分は無視され、s を入力するとその部分はより小さい部分に分割され、e を入力するとその部分を手作業で編集することが可能となり、q を入力するとインタラクティブなセッションを終了します。

- **git commit**

git commit は、ステージされたスナップショットをローカルリポジトリにコミットするコマンドです。コミットされたスナップショットはプロジェクトの「安全に保存された」バージョンであると解釈でき、明示的に変更指示が行われない限り **Git** がそれを変更することはありません。これは **git add** と共に **Git** における最も重要な種類のコマンドです。

名称は同じですが、このコマンドは **svn commit** コマンドとは全く異なるものです。スナップショットはローカルリポジトリにコミットされるため、他の **Git** リポジトリには全く影響を与えません。

```
git commit
```

ステージされたスナップショットをコミットするコマンドです。このコマンドを実行するとテキストエディターが起動され、コミットメッセージの入力を求められます。メッセージの入力後、ファイルを保存してエディターを終了するとコミットが実行されます。

```
git commit -m "<message>"
```

テキストエディターを起動することなく、<message> をコミットメッセージとして、ステージされたスナップショットをコミットします。

```
git commit -a
```

作業ディレクトリにおけるすべての変更のスナップショットをコミットします。これには追跡対象ファイル(過去に **git add** コマンドによってステージングエリアに追加されたことのあるファイル)の修正のみが含まれます。

- **git status**

git status は、作業ディレクトリの状態とステージされたスナップショットの状態を表示するコマンドです。このコマンドを実行すると、ステージされた変更内容、されていない変更内容、Git による追跡の対象外となっているファイルが表示されます。このステータス情報出力には、コミット済みの変更履歴に関する情報は含まれません。コミット済みの変更履歴に関する情報を取得する場合は、**git log** コマンドを使用します。

git status

ステージされたファイル、ステージされていないファイル、追跡対象外のファイルを一覧表示します。

- **git log**

git log は、コミット済みのスナップショットを表示するコマンドです。このコマンドを使用することにより、コミット済み変更履歴の一覧表示、それに対するフィルター処理、特定の変更内容の検索を行うことができます。**git status** コマンドは作業ディレクトリとステージングエリアの状態を確認するためのものであるのに対し、**git log** コマンドはコミット済みの履歴(コミット履歴)のみが対象です。

git log

コミット履歴全体をデフォルトの形式で表示します。出力表示が 2 ページ以上にわたる場合は **Space** キーでスクロールが可能です。終了する場合は **q** を入力します。

git log -n <limit>

表示するコミット数を<limit> に制限します。例えば **git log -n 3** とすると、表示するコミット数は 3 です。

git log --oneline

各々のコミットの内容を 1 行に圧縮して表示するコマンドです。このコマンドは、コミット履歴を概観する目的に適しています。

git log --stat

通常の **git log** 情報に加えて、改変されたファイルおよびその中の追加行数と削除行数を増減数で表示します。

git log -p

各々のコミットに対応するパッチを表示します。このコマンドを実行すると、コミット履歴から取得できる最も詳細な情報である各々のコミットの完全な差分情報が表示されます。

```
git log --author="<pattern>"
```

特定のオーサーが行なったコミットを検索します。引数<pattern>にはプレーンテキストまたは正規表現を用いることができます。

```
git log --grep="<pattern>"
```

コミットメッセージが<pattern>(プレーンテキストまたは正規表現)と一致するコミットを検索します。

```
git log <since>..<until>
```

<since>と<until>の間に位置するコミットのみを表示します。2個の引数には、コミット ID、ブランチ名、HEAD、その他任意のリビジョンリファレンスを用いることができます。

```
git log <file>
```

指定したファイルを含むコミットのみを表示します。このコマンドは、特定のファイルの履歴を調べる目的に便利です。

```
git log --graph --decorate --oneline
```

ここにはいくつかの便利なオプションが示されています。--graph フラグを指定すると、コミットメッセージの左側にテキストベースでコミット履歴をグラフ化したものの描画します。--decorate フラグを指定すると、表示されているコミットのブランチ名やタグ名を追加して表示します。--oneline フラグを指定するとコミット情報を圧縮して1行に表示するためコミット履歴の概観に便利です。

- git checkout

git checkout は、ファイルのチェックアウト、コミットのチェックアウト、ブランチのチェックアウトの3つの異なる機能を有するコマンドです。この章では最初の二つの機能について説明します。

コミットのチェックアウトを行うと、作業ディレクトリがそのコミットと完全に一致した状態になります。このコマンドは、プロジェクトの現在の状態を一切変更することなく過去の状態を確認する場合に使用します。ファイルのチェックアウトを行うと、作業ディレクトリの他の部分に一切影響を与えることなくそのファイルの過去のリビジョンを確認することができます。

```
git checkout master
```

master ブランチに戻るコマンドです。ブランチについては次の章で詳しく説明しますが、ここではとりあえず master ブランチはプロジェクトの「現在の」状態に戻る手段だと考えてください。

```
git checkout <commit> <file>
```

ファイルの過去のリビジョンをチェックアウトするコマンドです。このコマンドを実行すると、作業ディレクトリに存在する指定した<file>が、指定した<commit>に含まれそのファイルの完全なコピーとなり、さらにそれをステージングエリアに追加します。

```
git checkout <commit>
```

作業ディレクトリ内のすべてのファイルを指定したコミットと同一の状態に更新するコマンドです。引数<commit>にはコミットハッシュまたはタグを使用することができます。このコマンドを実行すると、” detached HEAD” 状態になります。

- **git revert**

git revert は、コミットされたスナップショットを元に戻すコマンドです。ただし、プロジェクト履歴においてそのコミットがなかったものとするのではなくそのコミットによって加えられた変更を元に戻す方法を見出してその結果を新しいコミットとして追加するものです。これは、Git の履歴を保全するためであり、このことはバージョン履歴の完全性の維持とコラボレーションの信頼性の確保には必須です。

```
git revert <commit>
```

<commit>によって加えられたすべての変更を元に戻す新しいコミットを生成し、それを現在のブランチに適用するコマンドです。

- **git reset**

git revert コマンドが変更を元に戻す「安全な」方法であるとするならば、**git reset** コマンドは「危険な」方法とすることができます。**git reset** コマンドを使用して元に戻すと（そして **ref** や **reflog** によるリファレンスが不可能になっている場合）、元の状態を復元する方法はありません。この「元に戻す」操作を取り消すことは不可能なのです。このコマンドは Git において作業結果を失う可能性のある数少ないコマンドのひとつであり、このコマンドを使用する場合は注意が必要です。

git checkout コマンド同様、**git reset** も様々な設定項目のある応用範囲の広いコマンドです。コミット済みのスナップショットを削除する目的にも使用されますが、ステージングエリアや作業ディレクトリにおける変更を元に戻す場合により多く使われます。いずれにしてもこのコマンドの使用はローカルな変更を元に戻す場合に限るべきであり、他の開発者に公開されているコミットの取り消しは決して行ってはなりません。

```
git reset <file>
```

作業ディレクトリに何の変更も加えずに、指定したファイルをステージングエリアから削除するコマンドです。このコマンドを実行すると、変更を書き込むことなく指定したファイルをアンステージします。

```
git reset
```

作業ディレクトリに何の変更も加えることなくステージングエリアをリセットして直前のコミット時の状態と一致させるコマンドです。このコマンドを実行すると、変更を書き込むことなくすべてのファイルをアンステージし、一度ステージされたスナップショットを初めから再構築することができますようになります。

```
git reset --hard
```

ステージングエリアと作業ディレクトリをリセットして直前のコミット時の状態と一致させるコマンドです。--hard は、変更をアンステージした上でさらに作業ディレクトリ内のすべての変更を元に戻すことを Git に指示するフラグです。言い換えると、これはコミット前のすべての変更を全くなかったものとするコマンドであり、これを使用する場合はローカルマシーン上で行った開発作業を本当に破棄していいのか否かを確認する必要があります。

```
git reset <commit>
```

現在のブランチの先端を<commit>の位置に戻した上でステージングエリアをその状態と一致するように元に戻しますが、作業ディレクトリのみはそのままにしておきます<commit>の実行後に行われた変更は作業ディレクトリに保存されており、より変更規模が小さくて整理されたスナップショットを作成してローカルリポジトリへの再コミットを行うことができます。

```
git reset --hard <commit>
```

現在のブランチの先端を<commit>の位置に戻した上でステージングエリアおよび作業ディレクトリをその状態と一致するように元に戻します。このコマンドを実行すると、コミット前の変更に加えて<commit>の後に行われたすべてのコミットも全くなかったものとなります。

- **git clean**

git clean は、作業ディレクトリから追跡対象外のファイルを削除するコマンドです。 **git status** コマンドを使用して追跡対象外のファイルを確認して手作業でそれらを削除することは簡単ですので、このコマンドはどちらかといえば便宜性を高めるために用意されたものです。通常の **rm** コマンド同様 **git clean** コマンドも元に戻すことはできないため、このコマンドを実行する際にはその追跡対象外ファイルを本当に削除していいか否かを再確認してください。

git clean は、**git reset --hard** コマンドとよく併用されます。既に説明したように **reset** コマンドが作用するのは追跡対象となっているファイルのみであるため、追跡対象外のファイルをクリーンアップするためには別のコマンドが必要になるのです。この2つのコマンドを併用することにより、作業ディレクトリをある特定のコミットの時点と完全に同一の状態に戻すことができます。

git clean -n

git clean の「予行演習」を行うコマンドです。このコマンドを実行すると削除されるファイルを表示しますが、実際の削除は行われません。

git clean -f

追跡対象外のファイルをカレントディレクトリから削除するコマンドです。設定オプション **clean.requireForce** が **false** にセットされていない場合(このオプションはデフォルトでは **true** です)は、**-f (force)** フラグは必須です。このコマンドでは、追跡対象外のフォルダーやファイルであっても **.gitignore** で指定したものは削除しません。

git clean -f <path>

追跡対象外のファイルを削除しますが、その対象範囲は指定したパスに限定するコマンドです。

git clean -df

カレントディレクトリ内の追跡対象外ファイルおよび追跡対象外ディレクトリを削除します。

git clean -xf

カレントディレクトリ内の追跡対象外ファイルおよび **Git** では通常無視されるファイルを削除します。

- **git branch**

ブランチとは独立な開発ラインを意味します。ブランチは、このチュートリアルシリーズの最初の章である **Git の基本** で説明した編集/ステージ/コミットプロセスに対する抽象概念です。これは、作業ディレクトリやステージングエリア、プロジェクト履歴を全く新しく作成する手段であるとも考えられます。新規のコミットは、現在のブランチの履歴に記録され、プロジェクト履歴における分岐を形成します。

git branch は、ブランチの作成、一覧表示、リネーム、削除を行うコマンドです。このコマンドにはブランチの切り替えを行う機能も、分岐した履歴を統合して元に戻す機能もありません。このため、**git branch** コマンドは多くの場合 **git checkout** コマンドおよび **git merge** コマンドと併用されます。

```
git branch
```

リポジトリ内のすべてのブランチを一覧表示します。

```
git branch <branch>
```

<branch>という名称の新規ブランチを作成します。新たに作成されたブランチのチェックアウトは行われません。

```
git branch -d <branch>
```

指定したブランチを削除します。ブランチにマージされていない変更が残っている場合は **Git** が削除を拒否するため、このコマンドは「安全な」操作です。

```
git branch -D <branch>
```

指定したブランチにマージされていない変更が残っていたとしてもそれを強制的に削除するコマンドです。このコマンドは、特定の開発ラインで行われたすべてのコミットを完全に破棄する場合に使用します。

```
git branch -m <branch>
```

現在のブランチの名前を<branch>に変更します。

- **git checkout**

git checkout は, **git branch** コマンドによって作成されたブランチ間を移動するコマンドです. ブランチのチェックアウトを行うことにより, 作業ディレクトリ内のファイルがそのブランチに保存されているリビジョンに更新され, その後すべての新規コミットはそのブランチに記録されます. このコマンドは, 作業を行う開発ラインを選択する手段であると考えられます.

前の章において過去のコミットを閲覧する場合の **git checkout** コマンドの使用法を説明しました. ブランチのチェックアウトは, 指定されたブランチあるいはバージョンに一致するように作業ディレクトリが更新されるという点では似ていますが, 作業ディレクトリに加えられた変更が残っている場合はそれがプロジェクト履歴に保存されるという点が異なります. 即ちこのコマンドはリードオンリーの操作ではないのです.

```
git checkout <existing-branch>
```

git branch コマンドを使用して作成したブランチのチェックアウトを行うコマンドです. このコマンドを実行すると, **<existing-branch>**が現在のブランチとなり, それと一致するように作業ディレクトリが更新されます.

```
git checkout -b <new-branch>
```

新規ブランチ**<new-branch>**を作成して即時チェックアウトするコマンドです. **-b** フラグは, **git branch <new-branch>**コマンドを実行し, 続いて **git checkout <new-branch>**を実行する便利なフラグです.

```
git checkout -b <new-branch> <existing-branch>
```

前記のコマンドと同じ機能ですが, ただし現在のブランチではなく **<existing-branch>**を新規ブランチの基点とします.

- **git merge**

マージは, Git において分岐した履歴を戻して統合する手段です. **git merge** は, **git branch** コマンドを使用して作成された独立な複数の開発ラインをひとつのブランチに統合するコマンドです.

ここで, 次に説明するすべてのコマンドは現在のブランチへのマージを行うものであることに留意してください. 現在のブランチはマージの結果更新されますが, ターゲットブランチ(引数で指定したブランチ)はそのまま残ります. 従って, **git merge** コマンドは通常現在のブランチを選択する **git checkout** コマンドおよび不要になったターゲットブランチを削除する **git branch -d** コマンドと併用されます.

```
git merge <branch>
```

指定したブランチを現在のブランチにマージするコマンドです。Git では、マージアルゴリズムは自動的に選択されます(下で説明します)。

```
git merge --no-ff <branch>
```

指定したブランチを現在のブランチにマージしますが、その際に常に(たとえそれが「早送り」可能であっても)マージコミットを作成してマージします。このコマンドは、リポジトリにおいて発生したすべてのマージを記録する場合に有用です。

- **git commit --amend**

`git commit --amend` は、直前のコミットを修正する場合に便利なコマンドです。このコマンドを実行すると、全く新たなスナップショットをコミットするのではなく、ステージされた変更内容と直前のコミットとの結合が行なわれます。また、スナップショットに変更を加えずに単に直前のコミットメッセージを編集する場合にも有用です。

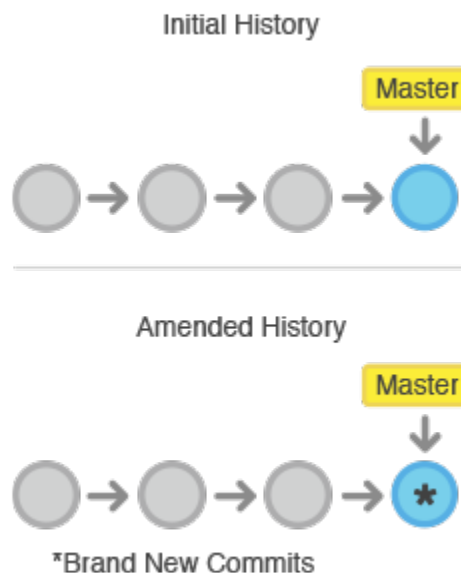


図 6 `git commit --amend`

ただし直前のコミットの修正とは、それを上書きするのではなく、全く別のコミットで置き換えることを意味します。Git ではそれは上図において星印(*)に示すように全く新しいコミットのように見えます。公開リポジトリに対する作業を行う場合はこのことを覚えておく必要があります。

```
git commit --amend
```

ステージされた変更を直前のコミットと結合し、その結果生成されるスナップショットで直前のコミットを置き換えるコマンドです。ステージエリアに何も無い状態でこのコマンドを実行すると、スナップショットを書き換えることなく直前のコミットメッセージの編集を行うことができます。

- **git rebase**

リベースは、ブランチの基点となるコミットを別のコミットに移動する操作です。一般的な動作を次の図に示します:

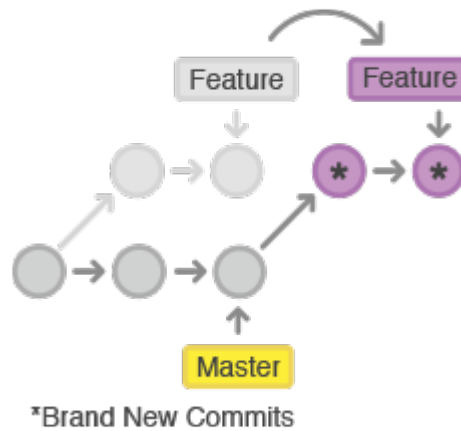


図 7 git rebase

見かけ上は、リベースはあるコミットから他のコミットにブランチを移動する手段に過ぎません。しかし Git の内部では、新たなコミットを生成してそれを移動先のベースコミットに適用することによってこれを行っており、これは即ち文字通りにプロジェクト履歴の書き換えをしていることになります。ここでは、ブランチそのものは同じものに見えていても、それを構成するコミットは全く異なることを理解することが重要です。

```
git rebase <base>
```

現在のブランチを<base>にリベースするコマンドで、リベース先としてはすべての種類のコミット参照(コミット ID, ブランチ名, タグ, HEAD への相対参照)を使用することができます。

- **git rebase -i**

git rebase を-i フラグを指定して実行するとインタラクティブなリベースセッションが開始されます。インタラクティブなリベースでは、すべてのコミットをそのまま新しいベースに移動するのではなく、対象となる個々のコミットの改変が可能です。これを使用して、既存の一連のコミットの削除、分割、改変を行って履歴を整理することができます。これはちょうど **git commit --amend** コマンドの強化版と言えます。

```
git rebase -i <base>
```

インタラクティブなリベースセッションを使用して現在のブランチを <base> にリベースするコマンドです。このコマンドを実行すると、エディターが開き、リベースする個々のコミットに対するコマンド(下で説明します)の入力が可能となります。ここでのコマンドは、個々のコミットを新しい基点に移動する方法を指定します。また、エディターにおけるコミットの並びを直接編集することによりコミットの順番を並び替えることもできます。

- **git reflog**

Git では、**reflog** と呼ばれる機能が働いて、ブランチの先端に対する更新の追跡が行われています。これにより、いかなるブランチからいかなるタグからも参照されていない更新内容であってもその時点に戻ることができます。履歴を書き換えた後であっても **reflog** にはブランチの過去の状態が記録されており、必要な場合にはそこに戻ることができます。

```
git reflog
```

ローカルリポジトリの **reflog** を表示するコマンドです。

```
git reflog --relative-date
```

相対形式の日付(例: 2 週間前)で **reflog** を表示するコマンドです。

- **git remote**

git remote は、他のリポジトリとの接続の作成、内容確認、削除を行うコマンドです。リモート接続とは、他のリポジトリへのダイレクトリンクではなく、ブックマークのようなものです。他のリポジトリにリアルタイムアクセスを行うのではなく、非短縮 URL への参照として使用可能な短縮名称として機能します。

例えば、次の図はローカルリポジトリと中央リポジトリとの間およびローカルリポジトリと他の開発者のリポジトリとの間の 2 つのリモート接続を示したものです。それらをフル URL を用いて参照するのではなく、他の Git コマンドに “**origin**” および “**john**” という名称のショートカットを引き渡すことが可能になります。

```
git remote
```

他のリポジトリへのリモート接続の一覧を表示するコマンドです。

```
git remote -v
```

上のコマンドと同様ですが、ただし各々の接続の URL が表示されます。

```
git remote add <name> <url>
```

リモートリポジトリに対する新規接続を作成するコマンドです。作成されると他の Git コマンドにおいて<url>の代わりに<name>を短縮ショートカットとして使用することができます。

```
git remote rm <name>
```

<name>という名称のリモートリポジトリへの接続を削除するコマンドです。

```
git remote rename <old-name> <new-name>
```

リモート接続を<old-name>から<new-name>にリネームするコマンドです。

- **git fetch**

git fetch は、リモートリポジトリからローカルリポジトリにブランチをインポートするコマンドです。インポートされたブランチは、これまで学習してきた通常のローカルブランチとしてではなく、リモートブランチとして保存されます。この機能により、それらをローカルリポジトリに統合する前に変更内容を確認することができます。

```
git fetch <remote>
```

リポジトリからすべてのブランチをフェッチするコマンドです。このコマンドを実行すると、付随するすべてのコミットおよびファイルもそのリポジトリからダウンロードされます。

```
git fetch <remote> <branch>
```

上のコマンドと同様の機能を有するコマンドですが、ただしフェッチする対象は指定したブランチのみです。

- **git pull**

中央リポジトリでの変更のローカルリポジトリへのマージは、Git ベースのコラボレーションワークフローにおいてはよく行われるタスクです。**git fetch** コマンドとそれに続く **git merge** コマンドを使用してこの操作を行う方法は既に説明しましたが、**git pull** はこの二つのコマンドをひとつにまとめたコマンドです。

```
git pull <remote>
```

現在のブランチの指定したリモートにおけるコピーをフェッチして、それを現在のブランチに即時マージします。これは、**git fetch <remote>** コマンドを実行し、続いて **git merge origin/<current-branch>** コマンドを実行するのと同様です。

```
git pull --rebase <remote>
```

上のコマンドと同様ですが、リモートブランチを現在のブランチにマージする際に `git rebase` コマンドを使用します。

- **git push**

プッシュとは、ブランチをローカルリポジトリからリモートリポジトリに送る操作を意味します。このコマンドは `git fetch` と対をなすものですが、フェッチはブランチをローカルリポジトリにインポートする操作であるのに対し、プッシュはブランチをリモートリポジトリにエクスポートする操作です。このコマンドは変更の誤書き込みを起こす可能性があるため、使用の際は注意が必要です。この問題は下で説明します。

```
git push <remote> <branch>
```

付随するすべてのコミットおよび内部オブジェクトと共に指定したブランチを `<remote>` にプッシュするコマンドです。このコマンドを実行すると、プッシュ先のリポジトリにローカルブランチが作成されます。変更の誤書き込みを防止するために、Git ではプッシュ先リポジトリにおける統合処理が早送りマージ以外である場合にはプッシュが拒否されます。

```
git push <remote> --force
```

上のコマンドと同様ですが、早送りマージ以外の場合にも強制的にプッシュが実行されます。プッシュ操作によって何が起こるかを完全に理解している場合以外は `--force` フラグを使用してはなりません。

```
git push <remote> --all
```

すべてのローカルブランチを指定したリモートリポジトリにプッシュするコマンドです。

```
git push <remote> --tags
```

ブランチをプッシュしただけでは、例え `--all` フラグが指定されていても、タグは自動的にプッシュされません。 `--tags` フラグを指定することにより、すべてのローカルタグをリモートリポジトリに送ることができます。[5]

3.6. 参考文献

- [1] Incept Inc.. “GitHub”. IT用語辞典 e-Words. 2014-09-28. <http://e-words.jp/w/GitHub.html>, (参照 2014-09-29).
- [2] 江口和宏, 山本竜三, 株式会社ヌーラボ. “知らないと現場で困るバージョン管理システムの基礎知識 (1/3)”. @IT. 2013-05-20. <http://www.atmarkit.co.jp/ait/articles/1305/20/news015.html>, (参照 2014-09-29).
- [3] 平屋真吾, クラスメソッド株式会社. “ガチで 5 分で分かる分散型バージョン管理システム Git (3/6)”. @IT. 2013-07-05. http://www.atmarkit.co.jp/ait/articles/1307/05/news028_3.html, (参照 2014-09-29).
- [4] Incept Inc.. “Git”. IT用語辞典 e-Words. 2014-09-28. <http://e-words.jp/w/Git.html>, (参照 2014-09-29).
- [5] Atlassian. “Git チュートリアル”. Atlassian. <https://www.atlassian.com/ja/git/tutorial>, (参照 2014-10-09).
- [6] 大塚弘記. GitHub 実践入門. 技術評論社, 2014-04-25.