

第1特集

楽しめる
始める

：新規さん歓迎特集！：

Git & GitHub 入門

It's as easy to learn as your



ソフトウェア開発におけるバージョン管理およびリソース管理は、ベストアンサーを探し続ける歴史でした。古くはCVS(Concurrent Versions System)、使いやすさを求めてSubversion。さらにMercurialが分散型バージョン管理システムの可能性をひらき、Linuxの父リーナスが開発したGitに注目が集まるようになりました。そしてGitとGitHubが、現在は主流になっています。

そうした状況のなか、そろそろ新人さんも開発現場へ実戦投入！——の時期ではないでしょうか。ソフトウェア開発において、今までと一番違うのはチームで仕事をすることです。一番大事なのは、ソフトウェアにデグレード(手戻り)を起こさず品質を向上させること。そのために先輩たちは、GitやGitHubを活用して仕事をしています。本特集は初心者を対象に、ちょっと扱いが難しいGitとGitHubをやさしく楽しく解説します。

第1章

基本概念から環境構築、さらに操作方法まで
はじめてのGit入門

岡本 隆史

P.18

第2章

リポジトリ作成からCIツール等との連携まで
GitHub入門

大塚 弘記

P.42

第1章

基本概念から環境構築、
さらに操作方法まで

はじめてのGit入門

Author 岡本 隆史(おかもと たかし) イラスト aico

はじめてGitを使う方の多くは、何をしたらよいのかわからず戸惑うでしょう。Gitの機能の多さ、分散型バージョン管理システムゆえの抽象度が高いシステム構成など、まさにプロの道具です。本章では、新入社員の方、はじめてGitを触る方に向けて、その使い方の概要を解説します。



1-1 新入社員、Gitに出会う

・朝光(よあけ)

新卒でタコ足Web(株)に入社してきたWebプログラマの卵。

・地頭(じとう)

先輩社員。朝光のOJT担当。タコ足Web(株)に服装規定はない。

・大鷹(おおたか)

朝光の同期。ボーダーシャツが好きなようだ。

: だったら、まずは、動いていた状態に戻したら?

: 戻そうと思うんだけど、どこを変更したのか覚えてません(涙)

: 変更する前のファイルのバックアップとか取ってないの?

: 取ってません。私、過去と振られた男は振り返らない主義なんです!

: まあ、頑張って直してくれ……。

このあと、終電まで頑張って直してなんとか乗り越えることができたそうです。

●得られた教訓「朝光メモ」

プログラムが動いたら、まずはそのファイルを保存しておき、誤った修正をした場合でもその状態に戻せるようにしておくこと。

『せっかくの編集内容が他人に上書きされて

さて、このトラブルの5日後、朝光ちゃんは、同じ新人の大鷹君と一緒にチケット購入機能の開発することになりました。2人で開発を行うため、ソースコードを共有しながら開発する必要があります。2人は、ファイルサーバ上でファイルを共有しながらソースコードを編集することにしました。朝光ちゃんはファイルサーバ上のファイルを直接エディタで編集し、大鷹君は

Git導入 Before/After

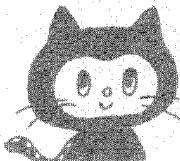
僕は開発経験が少ない

新入社員の朝光ちゃん(朝光と書いて「よあけ」と読むらしいです。ついにキラキラネーム世代の子が入社してくる時代になったんですね……)は、先輩の地頭のもと、研修を兼ねて入社早々、■ーソンのチケット購入システムの開発に携わることになりました。

そこに待ち受けていたのは、早々の苦難でした。

元に戻せないソースコード

: このプログラムさっきまで動いていたんだけど、変更しているうちに動かなくなっちゃいました><



第1章

基本概念から環境構築、さらに操作方法まで

はじめてのGit入門

ファイルサーバ上のファイルをいったん自分のフォルダにコピーして編集しているようです。

:あれ、さっき編集したログイン画面のファイルが元に戻っている……誰か編集した?

:あ、ちょうど僕が編集したファイルをコピーしたところだよ。

:ちょっとお! 私の編集した内容が消えちゃったじゃないの!

:あー、ごめん、ごめん。もう1回編集しなおしてね。

:どこ編集したか覚えてないよお(涙)

●得られた教訓「朝光メモ」

複数人でファイルを編集するときには、編集内容が不用意に上書きされないようにすること。

性格が表れるカオスなバックアップ

さて、ファイルが元に戻せなくなったり、他人にファイルを上書きされたり、トラブルを経験した朝光ちゃん、ファイルのバックアップを取ることにしました。

:間違った編集や不用意なファイルの上書きに備えて、ファイルのバックアップが必要ね。ちゃんとバックアップ取つておこ。

:ちょっと、なんかバックアップファイルみたいなのがたくさんあるんだけど、最新のファイルはどれよ?

```
index.php
index.php.最新
index.php.20150312
index.php.20150314
index.php.20150319
index.php.作業中
```

:「index.php.最新」が最新かな。あれ? 最新でコピーしたあと、index.php.作業中にバグ修正したんだったけ……。

:お前、絶対自分の部屋汚いだろ。洋服とか本とか下着とか床に散乱してそう!

::(ギクッ)そ、そんなことないもん!

●得られた教訓「朝光メモ」

バックアップは最新状態がわかるようにとりましょう。

そしてGitの導入へ

:なかなかファイルのバージョン管理に苦労しているようだね。

:ちょっとずつ工夫はしてみてはいますけど、油断するとファイルを上書きされたり、上書きされたファイルにまた同じ修正を反映させたりと、たいへんです>_<

:実は、Gitを使うとファイルのバージョン管理はもっと簡単にできるんだよ。

:え? そんなに便利なツールがあるんですか? なんでそれをもっと早く言ってくれないんですか?

:うん、身をもって苦労したほうが、バージョン管理の必要性を実感できると思って……。最初から便利な環境を与えられるより、自分で苦労をしたほうが問題意識を持つからね。それでツールを使ってもらえば、より深く理解できると思うんだよね。

:そんなものですか……。

——入社早々朝光さん、いろいろとカオスなことになってきていますが、これらファイルのバージョン管理の問題は、地頭さんのいうとおり、Gitを使えば、簡単に解決できます。Gitは、Linuxカーネルのソースコードの管理のために開発されたツールですが、Linuxカーネルだけでなく、AndroidやRubyをはじめ数多くのプロジェクトで採用されており、バージョン管理を行うための標準的なツールといっても過言ではありません。Gitには、次のような特徴があります。

楽しく
始める、新規さん歓迎特集！

Git & GitHub 入門

It's as easy to learn as your ABC



開発現場のはじめの一歩

1. ファイルの状態を簡単に記録できる

Gitを使えば、ファイルの状態を簡単に記録でき、記録した状態のファイルにいつでも簡単に戻すことが可能。ファイルには新しい記録がどんどん追加されて管理される。したがって古い記録も残る。そのため記録された状態のファイルをいつでも取り出せる

更がされていったのか把握しやすくなる

2. ファイルの同時編集が可能

ファイルを複数の開発者で同時に編集可能。同時に編集した場合でも、ほかの開発者の編集で不用意に上書きされ、自分の作業を消失することはない。また、別々の人が編集した変更点を簡単にマージする機能がある。変更点を、別の人気が編集したファイルに書き写さなくてもよい

——このように、朝光ちゃんが困っていたバージョン管理にまつわるトラブルはGitを使えば解決できます。また、Gitリポジトリを簡単に管理できるサービスとして、GitHub、BitBucketをはじめ、数多くのサービスが提供されていたり、Eclipse、VisualStudioなどのメジャーな開発環境やDreamweaverのようなWebオーサリングツールまでさまざまな開発環境がGitに対応しています。Gitさえ覚えておけば、バージョン管理に困ることはないと言っても過言ではないでしょう。

さて、Gitの重要性について理解できたところで、次の節からは実際にGitを体験していきましょう。

3. ファイルのバックアップが不要

1で言及したが、任意のファイルの状態を記録可能。いつでも記録したファイルを取り出せるので、日付ごとのバックアップファイルなどは不要になる。また、コミットメッセージと呼ばれる編集理由・内容をメモとして残すことができる。コミットメッセージにより、どのような変



1-2

Gitを使ってみよう

—Gitのしくみからリポジトリへのファイル登録まで



Gitの理解を握る鍵「リポジトリ」

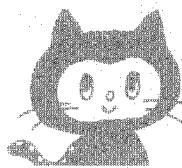
Gitを理解するには、リポジトリの理解が欠かせません。リポジトリは、ファイルの状態を格納するデータベースのようなものです。リポジトリには、ファイルの変更が、変更内容とその変更を補足するメッセージのセットで格納されています。

リポジトリでは、どのようなファイルも管理できますが、Gitはおもにソースコード(HTML、CSS、PHP、JavaScript、C、Java、など)や画

像ファイル、設定ファイルなどを管理します。ソースコードからコンパイルされる生成物、オブジェクトファイル(*.o)やコンパイルにより生成されたクラスファイル(*.class)は一般的に管理しません。ソースコードからコンパイルすればいつでも同じものが得られるので、管理する必要がないからです。

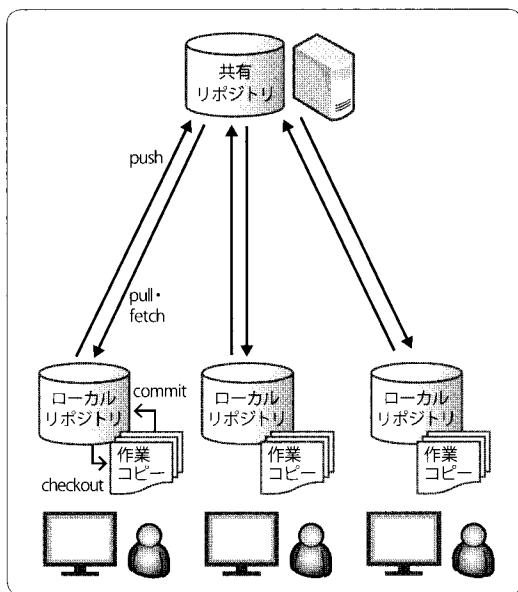
リポジトリの種類には、ローカルリポジトリと共有リポジトリがあります(図1)。

Git自身には、共有リポジトリやローカルリポジトリの概念はなく、どちらも同じリポジトリとして扱われますが、本稿では、Gitを利用する流



はじめてのGit入門

▼図1 共有リポジトリとローカルリポジトリ



これをわかりやすくするために、共有リポジトリ、ローカルリポジトリという言葉を使います。

ローカルリポジトリと共有リポジトリの違いを一言で説明すると、各開発者のPC上にあるリポジトリがローカルリポジトリで、共有サーバ上にある各開発者間の変更内容を共有するリポジトリが共有リポジトリです。

ローカルリポジトリ上のファイルを編集するディレクトリ、ファイルのことを「作業コピー」と呼びます。作業コピーは、エクプローラーやFinder、あるいはシェルからは普通のディレクトリ、ファイルに見えます。しかしながら、それらのファイルはGitにより管理されています。この作業コピー上のファイルの変更をリポジトリに登録する操作をコミット(commit)、リポジトリ上のある状態のファイルを取り出す操作をチェックアウト(checkout)と呼びます。

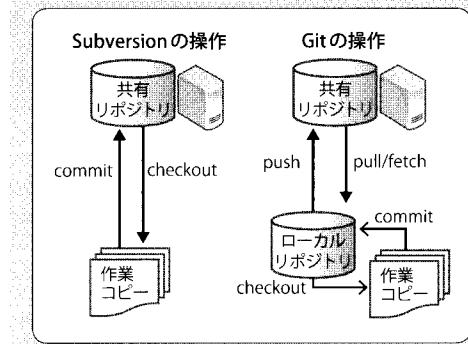
また、ローカルリポジトリに登録された変更点(commit)を共有リポジトリへ反映する操作をpush、共有リポジトリ上の変更点(commit)をローカルリポジトリへ取り込む操作をプル(pull)もしくはフェッチ(fetch)と呼びます。

コラム 「SubversionとGitの違い」

読者のには、すでにバージョン管理にSubversionを使っている方がいるかもしれません。SubversionにもGitと同様にcheckoutやcommitといったコマンドが出てきますが、意味合いが少し違います(図2)。

Subversionはcommit/checkoutにより共有リポジトリへ直接変更点を登録したり、ある時点のファイルを得しますが、Gitでは、ローカルリポジトリに対する操作となります。ほかの開発者が登録した共有リポジトリ上の変更を取り込んだり、自分の変更点をほかの開発者と共有するにはpushやpull/fetchが必要となる点に注意してください。

▼図2 SubversionとGitの違い



No.1 Gitをインストールしてみよう

では、さっそくGitをインストールしてみましょう。ここでは、Windows、Mac OS、LinuxごとのGitのインストール方法を紹介します。

Windows

Gitのサイト(<http://git-scm.com/>)からGitをダウンロードしてインストールします。

インストーラを起動し、ライセンスに同意すると、コンポーネントを選択する画面が表示されます(図3)。

ここでは、Advanced context menu(git-cheetah plugin)を選択します。コンポーネントを選択すると、Gitのパスの設定画面が表示されます(図4)。

それぞれ、次のような意味を持ちます。

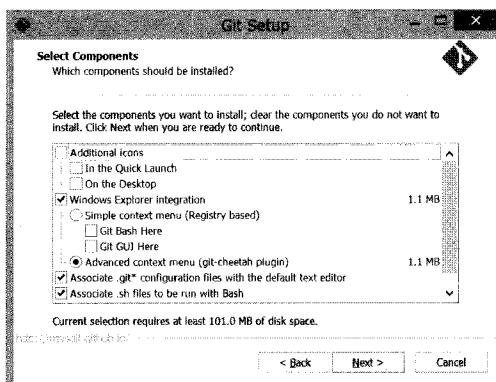
- [Use Git from Git Bash only]

GitをGitで提供しているシェル上だけで利

楽しく
始める！ 新人さん歓迎特集！

Git & GitHub 入門

▼図3 コンポーネント選択画面



用する、デフォルトの設定。ほかの環境に影響を与えないで、基本的には、これを選択する

・ [Use Git from Windows Command Prompt]

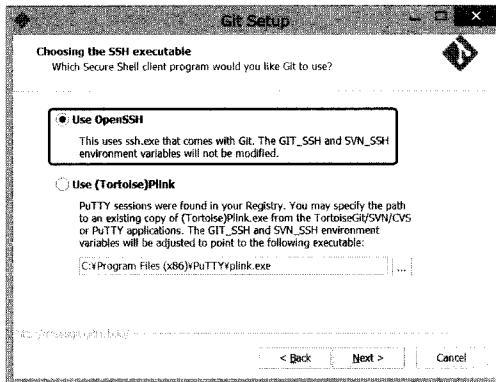
Windows標準のコマンドプロンプト(CMD)からGitを利用できるようする

・ [Use Git and optional Unix tools from the Windows Command Prompt]

上記と同じくWindows標準のコマンドプロンプトでGitを利用できるが、findやsortなどのそのほかのツールも利用できるようにする。Windows標準のツールがGit付属のツールに置き換わって実行されるので、注意が必要

とくにこだわりがなければ、[Use Git from Git Bash only]を選択し、コマンドプロンプト

▼図5 SSHプロトコルの選択画面

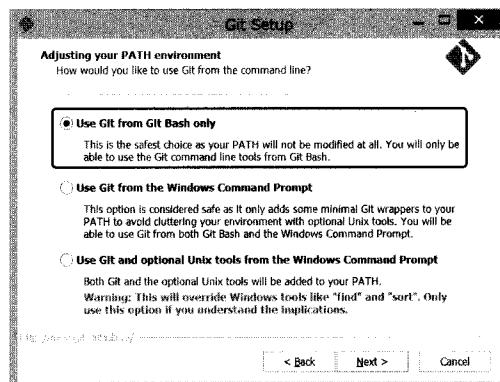


It's as easy to learn as your



開発現場のはじめの一歩

▼図4 Gitのパスの選択画面

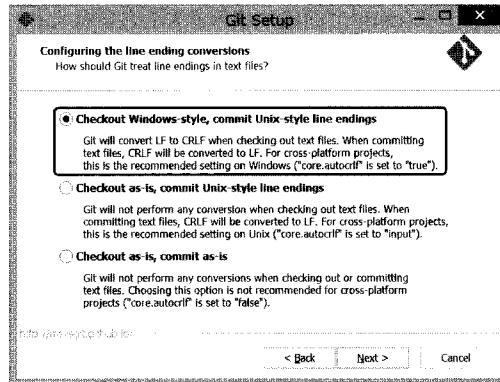


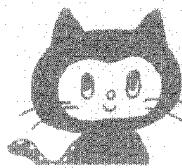
に影響を与えないようにしましょう。

ターミナルエミュレーターとしてPuTTYがすでにインストールされている場合は、SSHで接続するプロトコルを選択する画面が表示されます(図5)。最後に、改行変換のオプションを選択します(図6)。OSによって改行コードが異なりますが、Gitには改行コードを自動的に変換するオプションが提供されています。[Checkout Windows-style, commit Unix-style line endings]を選択し、ソースコードをチェックアウトしたときに、Windowsの改行コード(CR + LF)に変換し、コミットするときに、Unixの改行コード(LF)に変換するように設定します。

これで、異なるOSでテキストファイルを編集する場合でも、うまく編集できるようになります。ただし、改行コードの自動変換を行うため、この変換がトラブルになるケースも存在し

▼図6 改行コードの変換





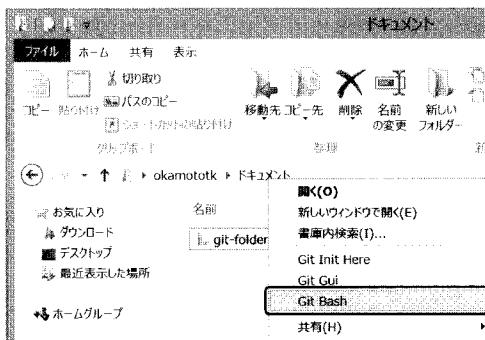
第1章

基本概念から環境構築、さらに操作方法まで はじめてのGit入門

▼図7 スタートメニューのGit Bash



▼図8 エクスプローラーからのGit Bashの起動



ます。その場合は、[Checkout as-is, commit as-is]を選択し、改行コードを変換しないようにしてください。なお、Gitをインストールしたあとで改行コードの変更を無効化するには、Gitのシェルから、

```
$ git config --global core.autocrlf false
```

を実行します。

Windowsでは、デフォルトの設定で日本語のファイル名がlsコマンドで正しく表示されないため、日本語ファイル名を正しく表示できるように設定します。

```
C:/Program Files (x86)/Git/etc/profile
64bit版Windowsの場合のパス
```

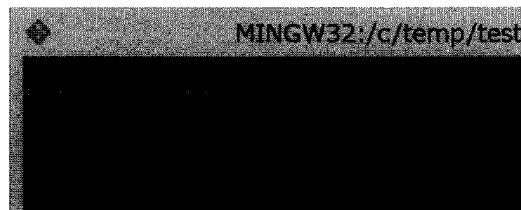
に下記の行を追記します。

```
alias ls='/bin/ls --show-control-chars'
```

インストールが完了したら、プログラムメニューのGit BashからGitが使えるシェルを起動できます。

Gitで利用されるデフォルトのエディタはvim

▼図9 Git Bashの日本語の文字化け



▼図10 フォントサイズの設定



ですが、インストールした状態ではSJISの文字コードでファイルを作成するようになっています。vimを利用するときに文字化けがあるので、utf-8を利用するように変更します。ユーザのホームディレクトリが次の場合、

```
C:/Users/<ユーザ名>/_vimrc
```

このような_vimrcファイルを作成します。

```
set fileencoding=utf-8
set fileencodings=utf-8,sjis
```

Windowsでは、スタートメニューやスタート画面から「Git Bash」を選択し、Gitが使えるシェル(コマンドプロンプト)を起動します(図7)。

また、エクスプローラーからフォルダを右クリックし、メニューから「Git Bash」をクリックすると、選択したフォルダでシェルを起動できます(図8)。

なお、Git Bashで日本語を表示すると、図9のように文字化けすることがあります。

文字化けする場合は、Git Bashのウィンドウ上部のタイトルバーを右クリックしてプロパティを選択し、フォントのサイズを変えると正しく表示されます(図10)。

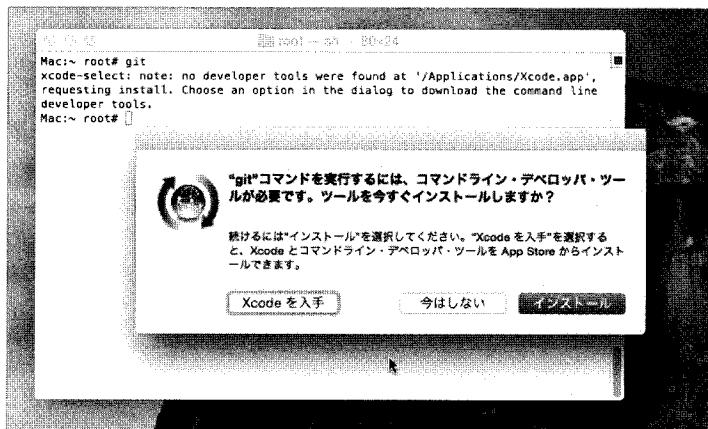
Git & GitHub 入門

It's as easy to learn as your



開発現場の「はじめての一歩」

▼図11 Git(コマンドライン・デベロッパ・ツール)のインストール画面



Mac OS

Mac OS の Yosemite(Mac OS X 10.10)では、ターミナルから git コマンドを実行すると、コマンドライン・デベロッパ・ツールのインストールを促す画面が表示されます(図11)。ここでインストールをクリックし、コマンドライン・デベロッパ・ツールをインストールすると、Git が使えるようになります。統合開発環境の XCode をインストールしても Git が使えるようになりますので、統合開発環境を利用する方は、XCode をインストールするだけで OK です。なお、コマンドライン・デベロッパ・ツールでインストールされる Git はバージョン 1.9.5 になります(2015年4月執筆時点)。最新版を利用したい場合は、Git のサイト (<http://git-scm.com/>) からパッケージをダウンロードしてインストールすることもできます。

Linux

Linux では、ディストリビューションで用意されている Git をインストールするだけで OK です。

Red Hat、CentOS、Fedora など

```
# yum install git
```

Ubuntu、Debian など

```
# apt-get update  
# apt-get install git
```

Git の初期設定

Git のインストールが完了したら、Git を利用するための初期設定を行います。

ユーザ名・日本語ファイル名の文字化け防止の設定

ユーザ名、メールアドレスの設定を行います。

```
$ git config --global user.name "Takashi Okamoto"  
$ git config --global user.email "toraneko@example.com"
```

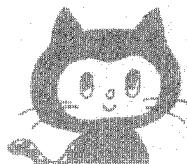
ファイル名のパスに日本語が含まれるとデフォルトの状態ではログメッセージなどを表示させたときに文字化けするので、正しく表示されるように設定します。

```
$ git config --global core.quotePath false
```

プロキシの設定

Git サーバへのアクセスに HTTP プロキシが必要な場合は、次のようにしてプロキシの設定を行います。

```
$ git config --global http.proxy http://<プロキシサーバ名>:<ポート>/
```



第1章

基本概念から環境構築、さらに操作方法まで

はじめてのGit入門

HTTP プロキシ経由で Web にアクセスするオフィスが多いと思いますが、そのような環境で GitHub などの外部の Git リポジトリを利用する場合は、この設定が必要になるので注意してください。



Gitリポジトリの準備

Git の準備ができたら、次にリポジトリを用意します。Git リポジトリを用意するには、新規にリポジトリを作成するか、すでに用意されているリポジトリを複製(clone: クローン)します。

新規リポジトリの作成

Git で管理したいファイルを置いているディレクトリで Git リポジトリ初期化コマンドを実行します。

```
$ git init
```

既存リポジトリのclone

すでに共有リポジトリが用意されている場合は、そのリポジトリを複製することにより、ローカルリポジトリとして取得します。たとえば、GitHub のリポジトリから clone する例は次のようになります。

```
$ git clone https://github.com/okamototk/ test.git
```

共有リポジトリは、GitHub や GitBucket などを利用すると簡単に用意できます。GitHub を利用して共有リポジトリを作成する方法については、38ページの「共有リポジトリを作ってみよう」を参照ください。

次の節では、複数の開発者がリポジトリを同時に編集した場合の競合について解説していますが、競合の解決を試すには、共有リポジトリが必要となりますので、できれば、共有リポジトリを用意してください。

リポジトリを作成、もしくは取得できたら、リポジトリのディレクトリに移動し、ディレク

トリの内容を確認してみましょう。次は作成したばかりの空のリポジトリの例です。

```
$ ls -a  
... .git
```

.git という名前のフォルダが存在するのが確認できます。.git フォルダの下で、Git の設定情報やリポジトリの変更内容が管理されます。



新しいファイルの登録

リポジトリの準備ができたら、さっそくファイルを Git で管理してみましょう。

Git では、Git リポジトリ上で編集するファイルのことを作業コピーと呼びます。作業コピーは、編集するファイルそのもの(テキストファイル、ソースコード、画像など)だと思ってください。

実際に README.txt ファイルを作成し、リポジトリに登録する手順を見ていきましょう。リスト 1 のようなファイルを作成します。

ファイルを作成した時点でファイルの状態は、次のようになっています。

```
$ ls -a  
... .git README.txt
```

リポジトリにファイルを登録するには、まず、インデックスと呼ばれるリポジトリへ登録する内容を一時的に格納する領域に、ファイルを登録します。次にコミット(commit)によりインデックスの内容をリポジトリに登録します。インデックスへの追加は add コマンドを、コミットは commit コマンドを利用します。

```
$ git add README.txt  
$ git commit
```

commit コマンドを実行すると、図 12 のような内容が表示され vi(vim) エディタが起動します。

Git では、ほかの一般的なバージョン管理システムと同じように、ファイルの変更内容を登

▼リスト1 README.txt

こんなものは Git



Git & GitHub 入門

▼図12 新しいファイルをcommitしたときの出力結果

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file: README.txt
#
```

▼図13 新しいファイルに書き込みを加える

```
READMEを追加
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file: README.txt
#
```

登録するときにコミットメッセージを記録して変更内容を登録します。上記のメッセージに表示されているとおり、#で始まる行は無視されるので、メッセージを追加した部分だけコミットメッセージとして登録されます。

ここで、**I**キーを押してインサートモードにして、図13のように「READMEを追加」と入力して、**Esc****:wq**と押して登録内容を保存し終了します。

これで、ファイルのリポジトリへの登録は完了です。ファイルを1つ1つ追加するのが面倒な場合は、addコマンドでフォルダを指定すると、フォルダに含まれるファイルを一括して登録できます。たとえば、srcフォルダとそのサブフォルダの内容をリポジトリに登録するには次のようにします。

```
$ git add src
$ git commit
```



無視ファイルの設定

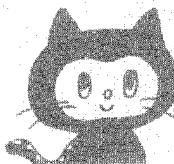
Gitのリポジトリでは、一般的に、ソースコードや画像ファイルなどを管理しますが、ソースコードをコンパイルした生成物(*.o, *.class, *.exeなど)や、エディタのバックアップファイル(*~, *.bakなど)などは、リポジトリで管理は行いません。リポジトリのトップにある.gitignoreファイルに、無視するファイルのパターンを登録しておくと、余計なファイルがリポジトリにcommitされることを防ぐことができます。たとえば、.gitignoreファイルは、リスト2のように記述します。

.gitignoreファイルもGitリポジトリにcommitできます。.gitignoreファイルをcommitし、次に紹介する共有リポジトリへpushすれば、ほかの開発者と.gitignoreの設定を共有できて便利です。GitHubのWebページ注1にさまざまな言語や環境のための.gitignoreファイルのサンプルがあるので、もっと詳しく知りたい方は参考にしてみてください。

▼リスト2 .gitignore

```
*.jar
*~
*#
bin/
*.o
*.obj
*.bak
*.class
*.dll
*.so
*.exe
```

注1) <https://github.com/github/gitignore>



第1章

基本概念から環境構築、さらに操作方法まで

はじめてのGit入門

コラム 「Gitで利用するエディタを変更する」

Gitは標準では、コミットメッセージの編集にvi(vim)を利用するように設定されています。viは操作方法が特殊で慣れない人にとっては、若干使いづらいエディタです。慣れたエディタを利用したい場合は、次のように設定します。

```
$ git config --global core.editor <エディタのパス>
```

なお、エディタで日本語を利用する場合は、BOMなしのUTF-8に対応したエディタを利用する必要があります。Windows標準で用意されているメモ帳(notepad.exe)は、BOMありのUTF-8がデフォルトとなっているため、次のようにコミットメッセージの先頭にBOMが混入してGitでコミットログが正しく表示されません。

```
$ git log
Author: Takashi Okamoto <okamototk@example.com>
Date: Sun Mar 22 16:31:41 2015 +0900
```

<U+FEFF>^M READMEを追加

ほかのエディタでもデフォルトがUTF-8でないエディタの場合では、単にパスを設定するだけでは文字化けしてしまいます。TeraPadとサクラエディタの設定例を紹介しておきます。

・TeraPad

```
$ git config --global core.editor "'C:\Program Files (x86)\TeraPad\TeraPad.exe' //cu8 //el"
```

・サクラエディタ

```
$ git config --global core.editor "'C:\Program Files (x86)\sakura\sakura.exe' -CODE=4"
```

登録が完了したら、logコマンドで登録された内容を確認してみましょう。

```
$ git log
commit feb6e178a0321db73da34d57a5801853b6a426ac
Author: Takashi Okamoto <okamototk@example.com>
Date: Sun Mar 22 16:05:44 2015 +0900

READMEを追加
```

Gitでは、リポジトリへ登録された内容を、各登録ごとにハッシュ値で管理しています。上記のcommitの行にある「feb6e1....」と書かれた16進数で表記された数字です。commitと言うと、このコミットを表すハッシュ値を示すこともあります。commitは、ほかのバージョン管理システムでは、リビジョンと呼ばれ、数字で表されることもあります。

showコマンドにより、実際の変更内容を確認できます。前述のcommitの内容を確認してみましょう。

```
$ git show feb6e1
commit feb6e178a0321db73da34d57a5801853b6a426ac
Author: Takashi Okamoto <okamototk@example.com>
Date: Sun Mar 22 16:17:39 2015 +0900
```

READMEを追加

```
diff --git a/README.txt b/README.txt
new file mode 100644
index 000000..c335f17
--- /dev/null
+++ b/README.txt
@@ -0,0 +1 @@
+こんにちはGit
```

コミットメッセージに続き、変更された内容が表示されました。

```
+++ b/README.txt
+こんにちはGit
```

となっているのは、README.txtに「こんにちはGit」という行が追加されたことを示しています。



編集したファイルの登録

すでにリポジトリで登録されているファイルを変更した場合も、同様の手順で変更をリポジトリへ登録できます。先ほどcommitしたファ

楽しく
始める 新人さん歓迎特集！

Git & GitHub 入門

イルをリスト3のように変更してみましょう。

編集したファイルは、次のようにしてリポジトリへ登録できます。

```
$ git add README.txt  
$ git commit
```

コミットメッセージに「ファイルの編集のテスト」と入力して、ファイルを保存してエディタを終了すればcommitは完了です。

git logで確認すると、先ほどの「README.txtを新規作成」に加えて「ファイルの編集のテスト」でメッセージを入力したcommitが追加されているのがわかります。

```
$ git log  
commit 6c51f8d7a62910a8c6gd78a6ff1618e3683a  
be63  
Author: Takashi Okamoto <okamototk@example.com>  
Date: Sun Mar 22 16:05:44 2015 +0900  
  
ファイルの編集のテスト  
  
commit feb6e178a0321db73da34d57a5801853b6a426ac  
Author: Takashi Okamoto <okamototk@example.com>  
Date: Sun Mar 22 16:05:44 2015 +0900  
  
README.txtを新規作成
```

It's as easy to learn as your



開発現場のはじめの一歩

▼リスト3 README.txt

こんにちはGit

さようならGit

開発が進み履歴が増えてくると、もうちょっとシンプルに履歴を確認したくなります。--onelineオプションを付けることにより、1commit、1行で履歴を表示できます。

```
$ git log --oneline  
6c51f8d7 ファイルの編集のテスト  
feb6e178 README.txtを新規作成
```

ABC ファイルの削除

リポジトリで管理されたファイルをフォルダから削除してもリポジトリからは削除されません。リポジトリからファイルを削除するには、git rmを利用します。

```
$ git rm README.txt  
$ git commit -am "READMEファイルを削除"  
[master bb5a04c] READMEファイルを削除  
1 file changed, 5 deletions(-)  
delete mode 100644 README.txt
```

コラム 「コミットメッセージの書き方」

コミットメッセージは自由にテキストを入力できますが、何もルールを作らないとメンバーごとに統一性の取れないメッセージとなってしまいます。そのため、あらかじめルールを作ておくと良いでしょう。たとえば、次のように記載のルールを決めます。

ログインできない不具合を修正

* データベースからユーザ情報を取得するカラムを誤っていたため修正

1行目にはgit log --onelineコマンドでロガー覧を表示したときに表示されるので、そのcommitの概要がわかるように記載します。2行目は空行にし、3行目以降は・を先頭に、箇条書きで詳細な説明を並べます。・で箇条書きにすると、GitHubやRedmineでコミットログを確認した場合、フォーマットされて表示されます。また、GitHubやRedmineなどでGitリポジトリを利用している場合は、チケットやissue IDを入れることによりコミットメッセージを対応づけることができます。

たとえば、上記のチケットのIDが2番で登録されたら「ログインできない不具合」を修正した場合、コミットメッセージを

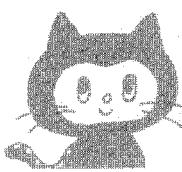
```
(refs #2) ログインできない不具合を修正
```

のように(refs #2)と入力することにより、そのcommitを対応づけることができます^{注A)}。また、GitHubでは次のようにすることでチケットをクローズすることもできます。

```
(fix #2) ログインできない不具合を修正
```

Redmineは参照やクローズを行うためには、別途設定が必要です。興味がある方は調べてみてください。

注A) GitHubでは、正確にはrefが不要です。



第1章

基本概念から環境構築、さらに操作方法まで

はじめてのGit入門

ディレクトリを削除する場合は、`-r`オプションを付けて次のようにします。たとえば、`dir`という名前のディレクトリを削除するには、次のようにします。

```
$ git rm -r dir
$ git commit
```



ファイルの名前変更

Gitで管理しているファイルの名前を変更したい場合は、`git mv`を利用します。ファイルの削除と同じく、ディレクトリ上のファイルを単に名前変更しただけでは、ファイル名を変更できないので注意してください。

```
$ git mv README.txt Manual.txt
$ git commit -am "ファイル名を変更"
[master bb5a04c] ファイル名を変更
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename README.txt => Manual.txt (100%)
```



編集したファイルの変更を取り消す

ファイルを編集しているうちに誤って編集してしまったため、ファイルをリポジトリの最新状態に戻したくなることがあります。次のように`reset`コマンドを利用すると最新状態に戻ります。

```
$ git reset --hard HEAD
```



1-3

チームでGitを使うには ——共有リポジトリを使った共同作業



チームで開発する際には、共有リポジトリでファイルを共有しながら開発を進めていきます。ここでは、チーム開発をするときに必要な共有リポジトリでのファイルの共有と、ファイルの編集がほかの開発者と衝突したときの競合の解決方法を紹介します。



共有リポジトリとローカルリポジトリ

今まで紹介してきた`commit`などの操作は開発者のマシン上のローカルリポジトリに対しての操作となります。チームで開発する場合、これらの変更をチームメンバーの間で共有する必要があります。チームで開発を行う場合は、メンバー間でソースコードを共有するために共有リポジトリを利用して変更を共有します。ローカルリポジトリ上の変更を共有リポジトリへ送信する操作を`push`、共有リポジトリ上のほかの開発者の変更を取り込む操作を`pull`と呼びます(図14)。



変更内容の共有リポジトリへのpush

`push`と`pull`の作業を実際に行いながら共有

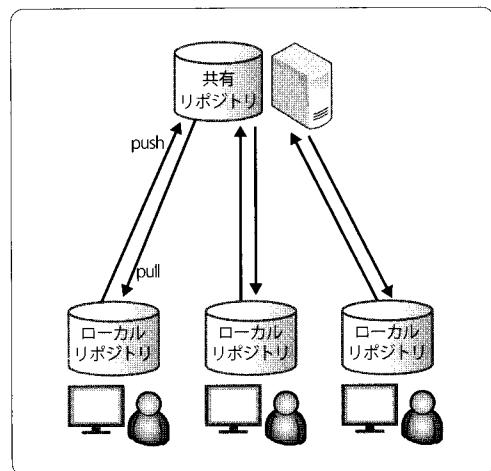
リポジトリで変更を共有してみましょう。

ローカルリポジトリの`commit`内容を共有リポジトリへ送信するには、`push`コマンドを利用します。

```
$ git push
```

空の共有リポジトリを`clone`して初めて`push`する場合は、次のようにリポジトリの送信先と

▼図14 共有リポジトリとpullとpush



楽しく、
始める、 新人さん歓迎特集！

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

▼図15 変更のpushが失敗した例

```
$ git push  
Password for 'https://okamototk@github.com/okamototk/test.git':  
To https://okamototk@github.com/okamototk/test.git  
! [rejected]      master -> master (fetch first)  
error: failed to push some refs to 'https://okamototk@github.com/okamototk/test.git'  
hint: Updates were rejected because the remote contains work that you do  
hint: not have locally. This is usually caused by another repository pushing  
hint: to the same ref. You may want to first integrate the remote changes  
hint: (e.g., 'git pull ...') before pushing again.  
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

送信するブランチを指定します。

```
$ git push origin master
```

この例では、originリポジトリ(clone元のリポジトリ)へmasterブランチ(Gitで利用するデフォルトのブランチ)をpushします。

なお、pushする際にすでに他のメンバによって、共有リポジトリ上に変更がpushされている場合、図15のようなエラーメッセージとともにpushが失敗します。

図15のようなエラーが出力された場合、上記メッセージの「hint: (e.g., 'git pull ...') before pushing again.」に書いてあるとおり、共有リポジトリ上の変更をgit pullで取り込んで、ローカルリポジトリを共有リポジトリと同期を取った状態にしてからpushすることにより、正しく実行できます。具体的には、図16のようにgit pullを実行したあと、git pushを実行します。

ほかの開発者の変更の取り込み

ほかの開発者が変更を共有リポジトリへpushした場合、その変更をローカルリポジトリへ取り込む必要があります。取り込むためにはpullもしくはfetchを行います。ここでは、共有リポジトリへ登録されたほかの開発者の変更を取り込む方法を紹介します。

▼図16 ローカルリポジトリを共有リポジトリと同期しgit pushを実行

```
$ git pull --rebase  
remote: Counting objects: 5, done.  
remote: Total 3 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (3/3), done.  
From https://github.com/okamototk/test.git  
 b6c0efd..b3fd453 master -> origin/master  
First, rewinding head to replay your work on top of it...  
Applying: 他の開発者の変更  
$ git push  
Password for 'https://okamototk@github.com/okamottok/test.git':  
Counting objects: 4, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 275 bytes | 0 bytes/s, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To https://okamototk@github.com/okamototk/test.git  
 b3fd453..92f4e33 master -> master
```

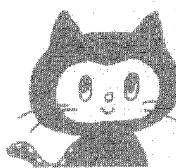
pullとfetch

共有リポジトリ上のcommitを取り込む方法として、pullとfetchの2つの方法があります。pullは、commitを取り込むときに作業ツリーに直接変更内容を反映します。つまり、リポジトリ内のファイルが共有リポジトリ上の最新状態になります。fetchは、共有リポジトリのcommitの内容をローカルリポジトリへ取り込むだけで、ファイルへの反映は行いません(図17)。

pullを実行するには、次のようにします。

```
$ git pull --rebase
```

fetchにより取り込まれたcommitは、FETCH_HEADという名前の特別なcommitで取り込まれます。たとえば、fetchを実行して、fetchで取り込んだ共有リポジトリの変更を確認するには、次のようにgit diffを実行します。



第1章

基本概念から環境構築、さらに操作方法まで

はじめてのGit入門

```
$ git fetch
$ git diff FETCH_HEAD
diff --git a/README.txt b/README.txt
index 14ac5c4..2c97e8b 100644
--- a/README.txt
+++ b/README.txt
@@ -1,3 +1,3 @@
■Git特集企画
```

-本特集は、これからGitを使う人のための面白い入門です。
+本特集は、これからGitを使う人のための真面目な入門です。

差分を確認したら、マージ(merge)、もしくはリベース(rebase)により、共有リポジトリの内容を取り込みます。mergeとrebaseの違いは次に説明しますが、たとえば、rebaseの場合は、次のようにします。

```
$ git rebase FETCH_HEAD
```

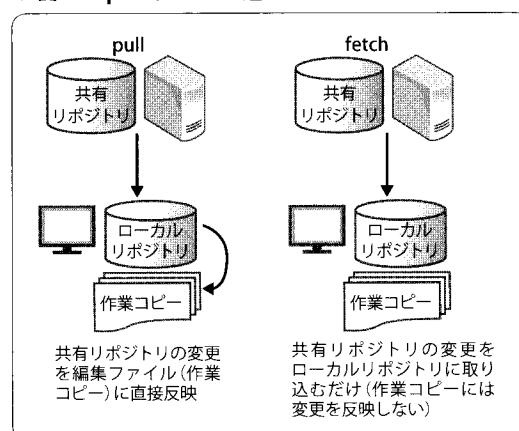
pullのほうが手順は少ないですが、fetchは変更内容を手元で確認してからcommitを取り込むことができます。pullの場合、思わず変更がいつの間に実行されている可能性があるので、できればfetchの利用をお勧めします。

mergeとrebase

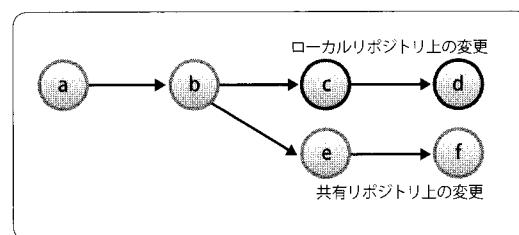
pull・fetchで取り込んだcommitをローカルリポジトリへ統合する方法として、mergeとrebaseという2つの方法があります。ローカルリポジトリ上で新しいcommitがないときは、どちらも同じ動作をしますが、リポジトリ上に新しいcommitがある場合に、mergeとrebaseで共有リポジトリ上のcommitとローカルリポジトリ上のcommitをまとめる方法に違いがあります。Git上のcommitの様子を図18のように表すことがあります。

丸はファイルの状態を表し、矢印はファイルが編集されて内容が変わったことと、commitトを示します。ローカルリポジトリでcommitを行う限り、commitは直線上に伸びていきますが、共有リポジトリにcommitが送信された場合、共通のファイルの状態を起点(ここではb)として、共有リポジトリとローカルリポジトリで別々にcommitが進み枝分かれした状態になります。

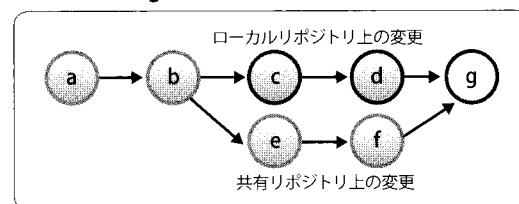
▼図17 pullとfetchの違い



▼図18 commitの表現



▼図19 merge



少しわかりにくいですが、bの状態の共有リポジトリから、朝光さんと大鷹君が同じファイルをclone(もしくは同期)したあと、朝光ちゃんがe, fと2度commitを行い、共有リポジトリへcommitをpushしたあと、大鷹君がc, dと自分のマシン上のローカルリポジトリでcommitした状態だと思ってください。

この枝分かれした状態を統合するやり方がmergeとrebaseで異なります。

mergerは、枝分かれしたcommitはそのままにして、merge commitと呼ばれる2つの枝を統合する特別なcommitを生成して枝分かれを解消します(図19のg)。

一方、rebaseは、共有リポジトリ上の

Git & GitHub 入門

▼表1 merge・rebaseの実行コマンド

	merge	rebase(お勧め)
pull	\$ git pull	\$ git pull --rebase
fetch	\$ git fetch \$ git merge FETCH_HEAD	\$ git fetch \$ git rebase FETCH_HEAD

commitのあとにローカルリポジトリ上のcommitが並ぶようにコミットを並び替えます(図20)。

mergeを利用すると、履歴がごちゃごちゃ枝分かれするので、rebaseのほうが履歴が直線に並ぶのでわかりやすいというメリットがあります。とくにこだわりがなければベースを利用するするのが良いでしょう。

先ほど紹介したpullとfetchを利用して共有リポジトリの変更をmergeとrebaseするコマンド手順を表1に示します。

ABC 競合の解決

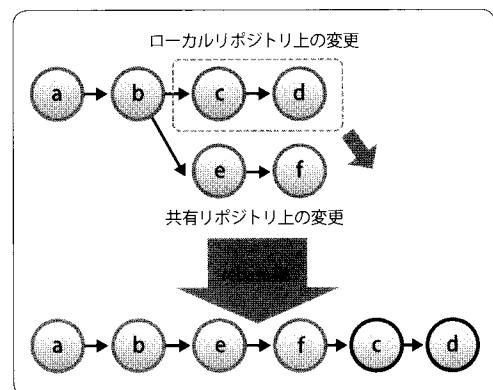
Gitは、複数の編集者が平行して同時にファイルを編集しても、rebaseやmergeで自動的に変更内容を統合してくれます。しかし、自動で変更内容をmergeできないことがあります。その状態を競合もしくはコンフリクトと呼びます。複数の編集者が同じファイルの同じ行を編集(もしくは追加、削除)したときは、Gitはどちらの編集者の変更内容を選択すれば良いのかわからないので、開発者に変更内容を確認するように促します。

たとえば、図21のようにファイルを朝光ちゃんと大鷹君が編集した例を示します。

最初企画書(README.txt)は、「これからGitを使うひとのための入門です」と記載された状態で共有リポジトリに登録されていました。大鷹くんは、面白い企画にしたいので、リポジトリをcloneしたあと、「面白い入門です」と変更・commitして共有リポジトリへpushしました。

大鷹くんと並行して、朝光ちゃんはcloneしたりポジトリで企画を真面目

▼図20 rebase

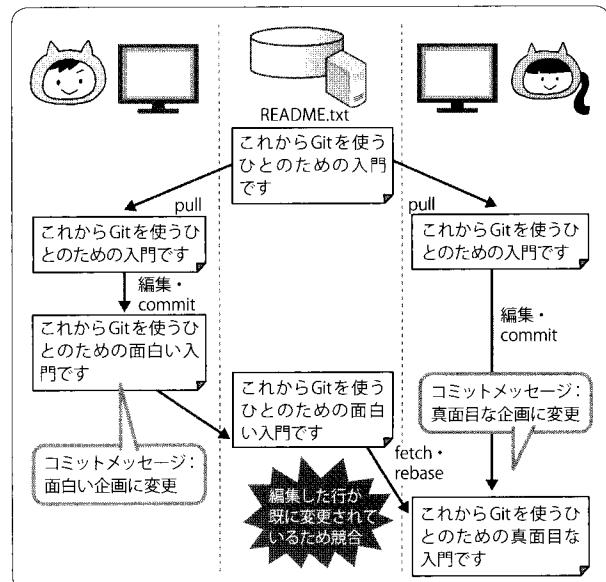


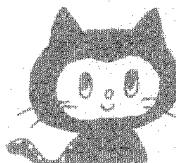
なものにしようと企画書を「真面目な入門です」と変更・commitしました。ここで、変更内容をpushしようとすると、すでに大鷹くんがpushした内容が存在するため、図22のようにエラーとなります。

そこで、共有リポジトリのcommitをローカルリポジトリへ統合します。先ほど紹介したとおり、統合するには、rebaseとmergeの2つの方法があります。それぞれ手順を見ていきましょう。

なお、競合を1人で試してみたい場合は、別々のディレクトリにリポジトリをcloneして、そ

▼図21 競合の例





第1章

基本概念から環境構築、さらに操作方法まで

はじめてのGit入門

▼図22 競合エラーメッセージ表示

```
$ git push
...
Password for 'https://okamototk@github.com/okamototk/test.git':
To https://okamototk@github.com/okamototk/test.git
 ! [rejected]      master -> master (non-fast-forward)
error: failed to push some refs to 'https://okamototk@github.com/okamototk/test.git'
hint: Updates were rejected because a pushed branch tip is behind its remote
hint: counterpart. If you did not intend to push that branch, you may want to
hint: specify branches to push or set the 'push.default' configuration variable
hint: to 'simple', 'current' or 'upstream' to push only the current branch.
```

それぞれ別々の人が変更したと見立てて、変更、commit、pushすると競合した状態を作ることができます。

次のようにclone時にcloneするディレクトリ名を指定すると、異なるローカルリポジトリとしてcloneできますので、それぞれのリポジトリで README.txt の同じ行を変更すれば競合状態を簡単に作り出すことができます。興味がある方は試してみてください。

```
$ git clone https://github.com/okamototk/test.git test-yoake
$ git clone https://github.com/okamototk/test.git test-ootaka
```



rebaseによる共有リポジトリの内容の統合

さて、共有リポジトリ上にcommitが存在するので、まず、fetchで共有リポジトリのcommitを取り込んで、取り込んだ共有リポジトリのcommitの先頭であるFETCH_HEADを指定してreabseを実行します(図23)。

ファイルに競合が存在する場合、

CONFLICT (content): Merge conflict in README.txt

のようなメッセージが出てリベース処理が止まります。競合が発生した README.txt を確認してみましょう(リスト4)。

競合が発生した個所に見慣れない記号

▼図23 rebaseの実行例

```
$ git fetch
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: 面白い企画に変更
Using index info to reconstruct a base tree...
M      README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
CONFLICT (content): Merge conflict in README.txt
Failed to merge in the changes.
Patch failed at 0001 面白い企画に変更
The copy of the patch that failed is found in:
  c:/temp/hage1/test2/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

「<<<<<」、「=====」、「>>>>>」が挿入されているのがわかります。

「=====」を挟んで、上下の部分の意味を見ていきましょう。

<<<<< HEAD
本特集は、これからGitを使う人のための面白い入門です。
=====

これは、現在編集しているローカルリポジトリ(正確にはカレントブランチ)の最新の内容を表しています。

▼リスト4 README.txt

■Git特集企画

<<<<< HEAD
本特集は、これからGitを使う人のための面白い入門です。
=====
本特集は、これからGitを使う人のための面白い入門です。
>>>>> 面白い企画に変更

楽しく
始める 新人さん歓迎特集！

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

本特集は、これからGitを使う人のための面白い入門です。
>>>>>>>>> 面白い企画に変更

これは、共有リポジトリ上の最新状態を表しています。「>>>>> 面白い企画に変更」の部分は、リポジトリの最新のコミットメッセージを示しています。

上記のファイルを編集し、ローカルリポジトリの内容が共有リポジトリの内容を残します。たとえば、ローカルリポジトリの内容を残したい場合は、README.txtファイルを編集し、次のように余分なテキストを取り除きます。

■Git特集企画

本特集は、これからGitを使う人のための面白い入門です。

競合を解決したら、次のように解決したファイルをインデックスに追加し、rebaseを継続します。

```
$ git add README.txt  
$ git rebase --continue  
Applying: 面白い企画に変更
```

複数の競合が発生した場合は、競合のメッセージがまた表示されるので、同様にして競合を解決していきます。競合を解決したファイルを確認すると、次のようになります。

```
$ git log --oneline --graph  
* 612efa1 面白い企画に変更  
* b141c53 真面目な企画に変更  
* 8417929 README.txtを追加
```

ここでは、--graphオプションを付けて、commitの並びの様子をグラフで確認しています。commitが直線上に並んでいますが、次に紹介するmergeのグラフと比べてみてください。

mergeによる共有リポジトリの内容の統合

mergeで編集の競合が発生した場合の解決手順は、次のようになります。競合は存在する場合は、merge時に次のようなメッセージが表示されます。

```
$ git fetch  
$ git merge FETCH_HEAD  
Auto-merging README.txt  
CONFLICT (content): Merge conflict in README.txt  
Automatic merge failed; fix conflicts and  
then commit the result.
```

CONFLICTの行のメッセージでREADME.txtでコンフリクトが発生したことが確認できます。競合が発生したファイルは、rebaseと同様に競合が発生したマークが挿入され、次のようにになります。

■Git特集企画

<<<<< HEAD

本特集は、これからGitを使う人のための面白い入門です。

<>>>>> FETCH_HEAD

本特集は、これからGitを使う人のための面白い入門です。

rebaseの場合は、共有リポジトリ上のコミットメッセージが「>>>>>」のあとに記載されていましたが、マージの場合は、共有リポジトリから取得したcommitの先頭であるFETCH_HEADが記載されています。rebaseの場合と同様にファイルを編集したら、add/commitして、mergeを終了します。

```
$ git add README.txt  
$ git commit
```

commitする際に、エディタには次のような

コラム 「競合表示の文字化け」

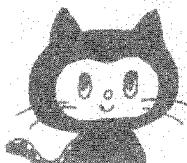
ファイルの文字コードがSJISの場合、競合の表示部分が次のように文字化けすることがあります。

<<<<< HEAD

本特集は、これからGitを使う人のための面白い入門です。

<>>>>> 請+进+變+・・請+%變

最終的に競合を解決するときに文字化けした部分は削除してしまうので、運用上は問題ありませんが、気持ち悪い場合は、ファイルを保存する文字コードをUTF-8にすれば文字化けを防ぐことができます。



第1章

基本概念から環境構築、さらに操作方法まで

はじめてのGit入門

ブランチを merge したことを示すメッセージがすでに入力されています。

```
Merge branch 'master' of https://github.com/okamototk/test.git
Conflicts:
 README.txt
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the
file
```

そのままファイルを保存、エディタを終了して commit します。git log で履歴を確認すると、

次のように merge されていることが確認できます。

```
$ git log --oneline --graph
* a22531f Merge branch 'master' of https://github.com/okamototk/git/test.git
|\ \
| * b141c53 真面目な企画に変更
* | f4fc266 面白い企画に変更
|/
* 8417929 README.txtを追加
```

先ほどの rebasr の場合に比べると、平行して編集した内容がいったん枝分かれして、統合された様子を確認できます。



1-4 ブランチとタグ

——作業ごとにバージョンを分けて管理し目印を付ける

ブランチとは、リポジトリ上のあるバージョンから分岐してファイルを管理する枝(ブランチ)のことです。たとえば、あるソフトウェアのバージョン1.0のソフトウェアをリリースしたあとに、新機能をたくさん盛り込んだ次にリリースするバージョン2.0を開発することになったとします。しかしながら、すでにバージョン1.0はリリースしているので、2.0の開発とは別にバージョン1.0のバグ修正の対応をきちんと管理する必要があります。このようなときには、図24のようにv1.0をリリースした時点のファイルの管理を分け、次のバージョンの開発には影響しないようにします。

Gitでは、リポジトリを作成すると、実は、masterという名前のブランチでバージョンが管理されています。今まで紹介してきたcommitやpush・pullなどの操作はmasterのブランチに対しての操作となっていました。masterは通常、最新版の開発に利用されます。ここで、v1.xという名前のブランチを作成すれば、バージョン1.x系列のメンテナンスができます。



ブランチの作成と確認

では、ブランチを作成してみましょう。ブランチを作成するには、次のようにします。

```
$ git checkout -b new-branch
```

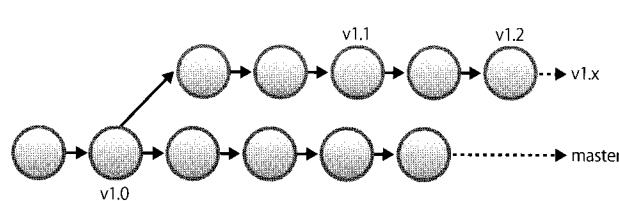
ブランチを作成したら、現在作業しているブランチを確認してみましょう。

```
$ git branch
 master
 * new-branch
```

ブランチ一覧が表示され、new-branchに*マークがついていることから、現在作業しているブランチがnew-branchであることが確認できます。

なお、Windows版のGitでは、Git Bashのコンソールに現在作業中のブランチ名を表示するよう

▼図24 ブランチ



楽しく
始める
新規さん歓迎特集！

Git & GitHub 入門

It's as easy to learn as your



になっていますが、下記のように git checkout で
ブランチが変わったことを確認できます。

```
okamototk@MY-PC /c/Users/okamototk/git/test  
(master)  
$ git checkout -b new-branch  
Switched to a new branch 'new-branch'  
  
okamototk@MY-PC /c/Users/okamototk/git/test  
(new-branch)
```

今までと同じように commitを行っていけば、
作成したブランチに commitできます。また、
このブランチでの作業内容は master ブランチ
には影響を与えません。

ABC ブランチのコミットの 共有リポジトリへの送信

次にブランチを共有リポジトリで共有してみ
ましょう。新しく作成したブランチで push を
行っても、次のように何も起こりません。

```
$ git push  
...  
Password for 'http://admin@192.168.56.101':  
Everything up-to-date
```

新しく作成したブランチを初めて push する
ときは、次のようにブランチを指定して push
します。

```
$ git push -u origin new-branch
```

2回目以降、もしくは一度 push されたブラン
チをほかのメンバが利用する場合は、下記のよ
うに普通に push するだけで OKです。

```
$ git push
```

共有リポジトリで共有されたブランチの情報
をローカルリポジトリに取り込むには、pull・
fetch・cloneなどで共有リポジトリの変更を取り
込む操作をするだけで OKです。

▼図25 ブランチ移動時のエラー

```
$ git checkout master  
error: Your local changes to the following files would be overwritten by checkout:  
      README.txt  
Please, commit your changes or stash them before you can switch branches.  
Aborting
```

ABC ブランチの移動

現在作業しているブランチから別のブランチ
に移動するときは、git checkoutコマンドを利
用します。たとえば、new-branchで作業中のとき、
master ブランチに戻るには次のようにします。

```
$ git checkout master  
Switched to branch 'master'  
Your branch is up-to-date with 'origin/master'.
```

ブランチを移動しようとすると、図25のよ
うにエラーになり失敗することがあります。

このエラーは commitされていない変更があ
る場合に発生します。図25のメッセージでは、
README.txtが変更されていますが、commit
されていないことを示しています。次のように
README.txtを commitすることによって、ブ
ランチを移動できます。

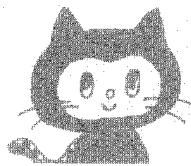
```
$ git add README.txt  
$ git commit  
$ git checkout master
```

編集内容を commit したくない場合は、次
のようにすることにより、編集内容を最後の
commit 状態に戻してブランチを移動できます。

```
$ git reset --hard HEAD  
$ git checkout master
```

ABC ブランチの merge

ブランチの説明を別バージョンの管理の例で
紹介してきましたが、あるまとまった機能開発
やバグ修正などをほかの開発に影響を与えない
ようにブランチを作成して作業を行うことがあ
ります。このようなブランチは作業が完了した
ら最終的に master ブランチやメンテナンスブラ
ンチに作業内容を merge する必要があります。



はじめてのGit入門

mergeを行うには、merge先のブランチにgit checkoutコマンドで移動して、git mergeコマンドでマージしたいブランチを指定します。たとえば、先ほど作成したnew-branchの内容をmasterブランチにmergeするには、次のようにします。

```
$ git checkout master
$ git merge new-branch
```

masterブランチとnew-branchの変更個所が重複し、merge時に競合が発生することがあります。基本的には、33ページで紹介した「rebaseによる共有リポジトリの内容の統合」の要領で競合を解決していきますが、ブランチをmergeした場合は、競合が発生した個所に次のようにブランチ名が記載されます。

```
<<<<< HEAD
本特集は、これからGitを使う人のための真面目な入門です。
=====
本特集は、これからGitを使う人のための面白い入門です。
>>>>> new-branch
```

タグ

Gitのcommitはハッシュ値で表されるので、直感的に扱いにくいものです。v1.1をリリースしたときのソースの状態などをわかりやすくするためにタグを付けることができます。タグを付けるには、タグを付けたいブランチでgit tagコマンドを利用して付与します。

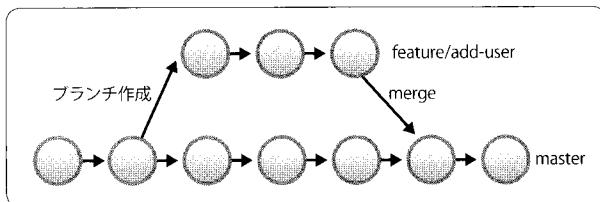
```
$ git branch v1.x
$ git tag v1.1
```

リポジトリで振られているタグの一覧を確認するのは、次のようにします。

```
$ git tag -l
v1.1
```

作成したタグを共有リポジトリに保存する場合は、push時に--tagsオプションを付けます。

▼図26 機能ブランチ



```
$ git push --tags
```

機能ブランチの種類

本節冒頭で、ブランチについてメンテナンスブランチを例に紹介してきました。メンテナンスブランチ以外にも、さまざまな種類のブランチが存在します。ここでは、よく利用する機能ブランチやトピックブランチといったブランチを紹介します。

機能ブランチ

機能ブランチは、あるまとまった大きな機能を実装するときに利用するブランチです(図26)。

大きな機能変更を行う場合、開発途中で思ぬバグを埋め込んでしまうことがあります。そのバグが、ソフトウェア全体に影響してしまう場合、バグのせいではかの開発者の作業も止まってしまうことがあります。そのようなときに機能ブランチとして、新機能を開発すれば、ほかの開発に影響せずに開発を進めることができます。

機能ブランチで作成した機能は十分テストしたあと、開発ブランチにmergeします。

トピックブランチ

機能ブランチと似ていますが、ちょっとした機能追加やバグ修正を行うなど、機能ブランチよりもっと細かい変更を管理するのに利用します。

機能ブランチと同様に、作業が終わったら開発ブランチやメンテナンスブランチなどのほかのブランチにmergeされます。トピックブランチは、共有リポジトリにpushされず、ローカルリポジトリで開発者の作業を行うために利用されることが多いです。

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

なお、ブランチ名を付けるときは、ブランチ種別／ブランチ名という形式でブランチを作成する

と、ブランチの目的・用途が明確になって良いです。

たとえば、機能ブランチは、feature/xxxという名前で、トピックブランチは、topic/xxxという名前でブランチを作成します。



1-5 共有リポジトリを作ってみよう —— GitBucket で楽々リポジトリ構築



Gitリポジトリを簡単に運用できる GitBucket

GitBucketは、Gitのリポジトリサービスとして人気のあるGitHubのクローンです。OSSで開発されており、無償で利用できます。社内LANにGitHubのようなGitリポジトリを簡単に手軽に構築できます。GitHubと同じように、Wikiによる情報整理やチケットによる課題やバグの管理、プルリクエストによるソースコードのレビューも行うことができます。GitHubの詳細については第2章を参照ください。ここでは、GitBucketのインストール、ユーザ作成、リポジトリの作成方法を簡単に紹介します。



インストール・実行

GitBucketの実行には、Javaが必要になります。Windows/Mac OSの場合は、JavaのダウンロードサイトからJavaをダウンロードしてインストールします。

<https://java.com/ja/download/>

Linuxの場合は、Javaパッケージをインストールします。RHEL/CentOS系ではjava-1.7.0-openjdkパッケージ、Debian/Ubuntu系ではdefault-jreパッケージをインストールします。

Javaがインストールできたら、GitBucketの配布サイトからgitbucket.warをダウンロードし、適当なディレクトリに配置します。

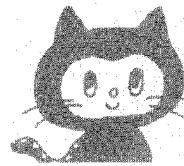
<https://github.com/takezoe/gitbucket/releases>

準備ができたら、コマンドプロンプトやシェルからgitbucket.warを配置したディレクトリに移動し、javaコマンドに-jarオプションを付けてgitbucket.warを実行します(図27)。

正しく動作していれば、<http://localhost:8080/>にWebブラウザでアクセスするとGitBucketのログイン画面が表示されます。ログイン画面で管理者アカウントroot(パスワードroot)でログインしてみましょう。ログインできれば、図28の画面が表示されます。

▼図27 GitBucketのインストール(gitbucket.warの実行)

```
C:\Users\okamototk>java -jar gitbucket.war
2015-03-31 16:46:46.168:INFO:oejs.Server:jetty-8.y.z-SNAPSHOT
2015-03-31 16:46:46.199:INFO:oejw.WebInfConfiguration:Extract jar:file:/C:/Users/okamototk/gitbucket.war!/ to C:\Users\okamototk\.gitbucket\${tmp}\webapp
2015-03-31 16:46:55.196:INFO:oejw.StandardDescriptorProcessor:NO JSP Support for
/, did not find org.apache.jasper.servlet.JspServlet
3 31, 2015 4:46:56 午後 grizzled.slf4j.Logger info
情報: The cycle class name from the config: ScalatraBootstrap
3 31, 2015 4:46:56 午後 grizzled.slf4j.Logger info
情報: Initializing life cycle class: ScalatraBootstrap
2015-03-31 16:46:56.639:INFO:oejs.AbstractConnector:Started SelectChannelConnect
or@0.0.0.0:8080
```

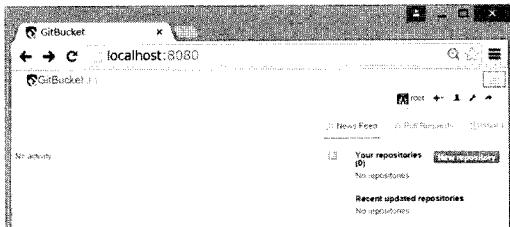


第1章

基本概念から環境構築、さらに操作方法まで

はじめてのGit入門

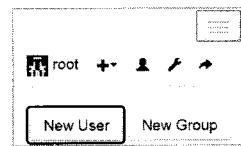
▼図28 GitBucketのログイン画面



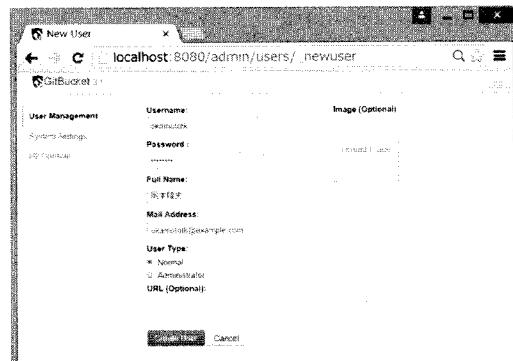
◀図29 メニューアイコン、ツールアイコンのクリック



▶図30 ユーザ作成



▼図31 ユーザ情報入力画面



ABC ユーザとリポジトリの作成

GitBucketのGitリポジトリはrootユーザで作成することもできますが、通常はGitBucketを利用するユーザを作成して、ユーザアカウントでリポジトリを作成します。各ユーザのアカウントを作成することにより、ユーザが自由にリポジトリを作成できるようになっています。ユーザを作成して、そのユーザでログインしてリポジトリを作成してみましょう。rootユーザでログインした状態で、右上のメニューアイコン、ツールアイコンを順番にクリックします(図29)。

ユーザ管理の画面になるので、ここで[New User]を選択します(図30)。

ユーザ情報入力画面(図31)になるので、ユーザ情報を入力して[Create User]ボタンをクリックすれば、ユーザの作成は完了です。

ユーザを作成し終わったら、rootユーザからいったんログアウトします。右上のメニューから矢印のアイコンをクリックします(図32)。



リポジトリ作成

作成したユーザでログインし、画面右にある[New repository]ボタンをクリックします(図33)。

リポジトリ情報入力画面(図34)になるので、リポジトリ名や説明文などを入力します。デフォルトでは、リポジトリをほかのユーザと共有する設定となります。[Private]を選択するとほかのユーザと共有しない設定になります。

[Create repository]ボタンを

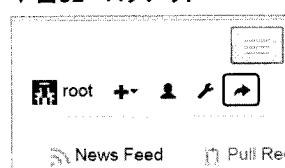
クリックすれば、リポジトリが作成され、リポジトリへアクセスする情報が表示されます(図35)。

たとえば、okamototkという名前のユーザでtestという名前のリポジトリを作成すると、次のURLでGitリポジトリにアクセスできるようになります。

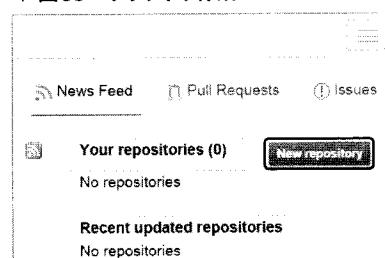
```
$ git clone http://localhost:8080/git/ okamototk/test.git
```

なお、作成されたリポジトリは、gitbucketを実行したユーザのホームディレクトリに作成される「.gitbucket」ディレクトリにあります。バックアップなどを取得する場合は、.gitbucketディレクトリをバックアップするようにすると良いでしょう。

▼図32 ログアウト



▼図33 リポジトリ作成



楽しく
始める
新規入力

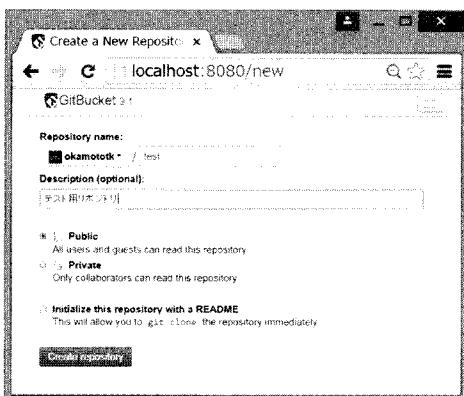
It's as easy to learn as your



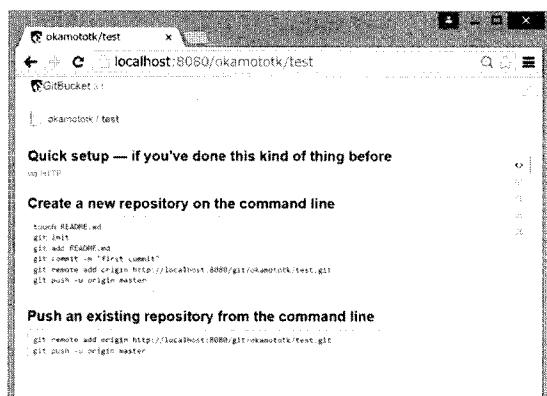
開発現場のはじめの一歩

Git & GitHub 入門

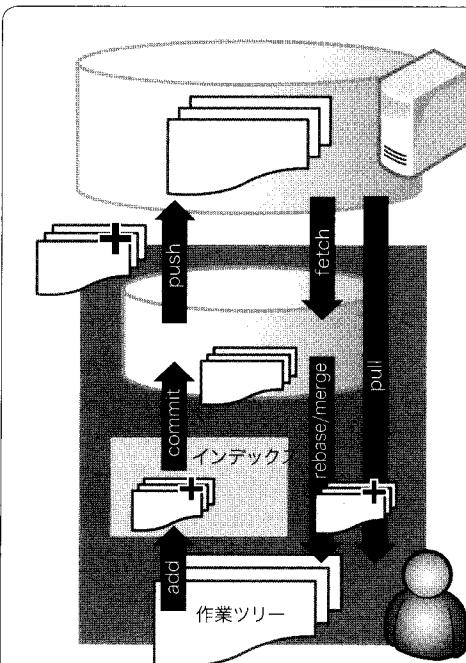
▼図34 リポジトリ情報入力画面



▼図35 リポジトリ作成完了画面



▼図36 Git基本コマンドの使い方



共有リポジトリ

共有リポジトリは、チームメンバーで共有するリポジトリで、ソースコードのメインバージョンが格納されています。cloneでローカルリポジトリに複製し、pull/fetchで変更をローカルリポジトリに取り込みます。

ローカルリポジトリ

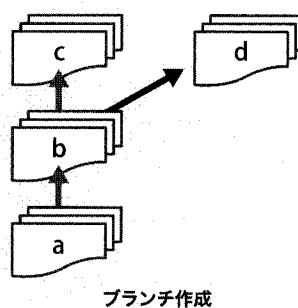
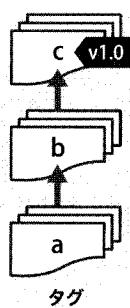
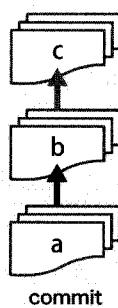
作業者のマシンにあるリポジトリです。ローカルマシンでの作業内容が保存されます。pushにより共有リポジトリに変更内容を反映します。

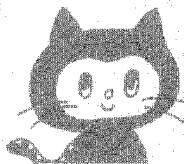
インデックス

ローカルリポジトリへ反映する変更を一時的にためておく場所です。インデックスの内容は、commitによりローカルリポジトリへ反映されます。

作業ツリー

ローカルリポジトリ上にある現在の作業ファイルです。作業ツリーの変更点はaddによりインデックスに追加されます。





第1章

基本概念から環境構築、さらに操作方法まで

はじめてのGit入門



おわりに



以上、Gitのさわりについて紹介してきました。おまけとしてGitの基本的なコマンドの使い方が一目でわかるチートシートを掲載します(図36)。Gitの使い方に慣れない人は、このチートシートを手元においてGitに向き合うはどうでしょうか。なお、このチートシートは筆者

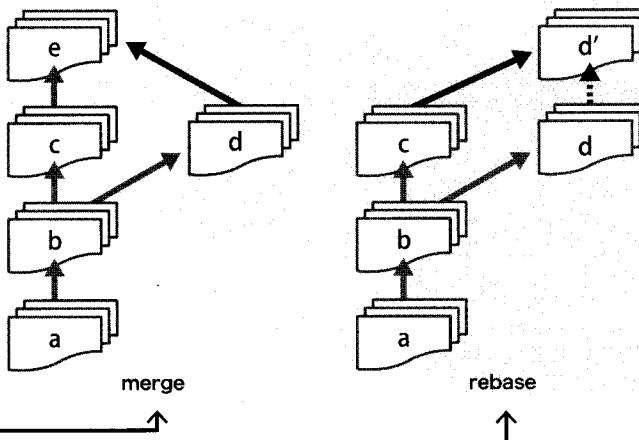
の著書『Gitポケットリファレンス』から抜粋したもので、Gitには本稿で紹介しきれなかつたさまざまなコマンドと使い方があります。本書では網羅的に解説しています。もっと知りたい方は、ぜひ『Gitポケットリファレンス』(技術評論社刊行)を! SD

基本

リポジトリを作成する
\$ git init
もしくは
\$ git clone https://github.com/okamototk/test
ユーザ名、メールアドレスを設定する
\$ git config user.name "Takashi Okamoto"
\$ git config user.email okamototk@example.com
ファイルを管理する
\$ git add FILE...
\$ git rm FILE...
\$ git mv OLD NEW
変更を確認する
\$ git status
\$ git diff BRANCH_NAME
\$ git show 8dc2334
変更をcommitする
\$ git add <変更したファイル>
\$ git commit
もしくは
\$ git commit -a
変更を共有リポジトリへ送信する
\$ git push

チーム開発

公開リポジトリ上の変更を取り込む
\$ git fetch
\$ git log HEAD..FETCH_HEAD
(取り込んだ変更履歴の確認)
\$ git diff HEAD..FETCH_HEAD
(取り込んだ変更内容の確認)
\$ git rebase FETCH_HEAD
(変更をカレントブランチに取り込む)
もしくは
\$ git pull --rebase
変更をmergeする
\$ git merge BRANCH_NAME
(コンフリクトした場合、修正後commit)
\$ git add <競合したファイル>
\$ git commit
変更をrebaseする
\$ git rebase BRANCH_NAME
\$ git rebase -continue
特定のcommitをブランチに取り込む
\$ git cherry-pick 8cab345a
(別ブランチの特定のバグ修正や機能変更だけを取り込んで、そのほかは取り込またくないときに利用)



その他

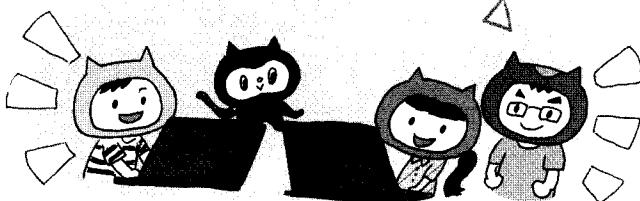
ブランチ・タグに切り替える
\$ git checkout FOO
リビジョンにタグを付ける
\$ git tag v1.0
タグを公開リポジトリへ反映させる
\$ git push --tags
変更履歴を表示する
\$ git log
現在の作業を一時保存する
\$ git stash
(一時保存)
\$ git stash list
(一時保存したリストの確認)
\$ git stash show
(一時保存した内容の確認)
\$ git stash pop
(一時保存した内容の取り出し)

第2章

リポジトリ作成からCIツール等
との連携まで

GitHub入門

Author 大塚 弘記(おおつかひろき) イラスト aico



GitHubは、Gitリポジトリを置く場所をインターネット上で提供するサービスです。バージョン管理の機能に加え、Pull Requestをはじめとした開発者同士がコミュニケーションしながら開発を行える機能を備えています。本章では、GitHubの基本的な利用手順とチームでGitHubを利用する流れについて解説します。また、今後のさらなる活用の足がかりとして、チャットやCIツールなどとの連携事例を紹介します。



2-1 GitHubとは？



「GitHub」という言葉を最近、耳にすることはないでしょうか？ 本節ではGitHubはどのようなものなのか、なぜ世界中の開発者が利用しているのかを解説します。



GitHubを提供するGitHub社

GitHubはサンフランシスコに本社を置くGitHub社によって提供されているサービスです。同社が提供しているのはGitHubがメインとなりますが、「Speaker Deck」^{注1)}のようなほかのサービスも提供企業を買収して提供しています。

GitHubにはOctocatと呼ばれる、タコと猫を合わせたようなマスコットキャラクターもあります(図1)。プログラマ向けのイベントや勉強会で、このマスコットのステッカーなどを目にしたことのある人は多いのではないでしょうか？

GitHub Octodex^{注2)}では、さまざまなバリエーションのOctocatを見ることができます。GitHub Shop^{注3)}では、Octocatのグッズだけでなく、GitHubに関連するグッズも購入できます。GitHubを気に入った人は、GitHub関連グッズを身の回りに置いてみてはいかがでしょうか？

GitHub社の現在の情報は公式ホームページ^{注4)}で確認できます。詳細はそちらをご覧ください。

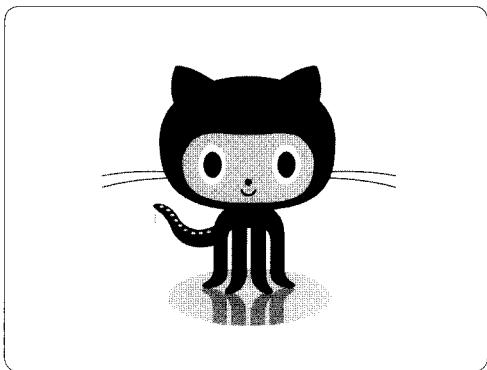


GitHubの利用状況

GitHubは2015年3月現在^{注5)}で、900万人の利用者がおり、約2,110万リポジトリがあり、世界中の開発者が日夜利用しています。

GitHubとは直接関係はありませんが、2015年3月12日に「Google Code」のサービス終了のアナウンス^{注6)}がありました。Googleは同サービスを終了する理由の1つとして、多くのOSSプロジェクトがGitHubなどへ移行していることを

▼図1 Octocat



注1) プрезентーション資料を共有するサービス。https://speakerdeck.com/

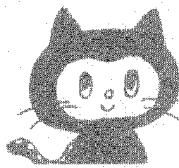
注2) https://octodex.github.com/

注3) http://github.myshopify.com/

注4) https://github.com/about

注5) https://github.com/about/press

注6) http://google-opensource.blogspot.jp/2015/03/farewell-to-google-code.html



第2章

リポジトリ作成からCIツール等との連携まで GitHub入門

挙げています。Google Codeの移行先としてGitHubを挙げて、移行ツールを提供しました。

このようなことからも世界中の開発者がGitHubを利用していることがうかがい知れます。



GitとGitHubの違い

ここではまずGit^{注7}とGitHub^{注8}の違いについて解説します。GitとGitHubはまったくの別物です。本特集ではGitとGitHubとは一貫して区別して表記します。

第1章で解説したバージョン管理システムソフトウェア「Git」では、リポジトリというデータの貯蔵庫にソースコードなどを格納して利用します。このリポジトリを置く場所をインターネット上に提供しているのがGitHubというサービスです。そのため、GitHubで公開されているソフトウェアのソースコードは、すべてGitで管理されています。よって、Gitについて理解しておくことは、GitHubを使いこなすうえでとても重要なことです。Gitの詳細は第1章を参照してください。



利用料金

GitHubは公開リポジトリであれば無料でいくつも作成できます。非公開のリポジトリを作成したければ、個人向けの一番安いMicroプランで月額7ドルを支払うと非公開リポジトリを5個まで作成できます。会社など複数人で

GitHubを利用するのであれば、Organization plansを利用してください。詳しくはGitHubの料金ページ^{注9}を参照してください。

GitHubは教育機関への支援も積極的であり、学生であればGitHub Education^{注10}のページから学生であることを申請すればMicroプランに相当する月額7ドル分を無料で利用させてもらいます。GitHub Educationのページには、Student Developer Packと題したソフトウェア開発を行う際に活用できるクラウドサービスやツールなどの紹介ページがあります。ここでは、「通常は有料で提供されているが、学生であれば一部無料で利用できるもの」が紹介されています。最小限のWebサービスは無料で開発して提供できる時代となりました。研究や勉強にも利用できるので、これを使わない手はありません。



GitHubを利用しているプロジェクト

実際にGitHubがどのようなプロジェクトで利用されているのか、代表的なものを表1にまとめてみました。みなさんも一度は聞いたことや使ったことがあるのではないでしょうか。

Explore GitHub^{注11}を覗けば、GitHubで人気のあるプロジェクトや話題のプロジェクトを見つけられます。驚くようなソフトウェアは一斉に注目を浴びて、Trending repositories^{注12}などに顔を出しますので、定期的にチェックすればおもしろいものにめぐり逢えると思います。

▼表1 GitHubを利用しているプロジェクト

名前	概要	URL
Ruby on Rails	Rubyの代表的なオープンソースWebフレームワーク	https://github.com/rails/rails
Bootstrap	Twitter社の開発するモダンなUIを提供するフレームワーク	https://github.com/twbs/bootstrap
Docker	Docker社の開発するアプリケーションコンテナエンジン	https://github.com/docker/docker
Fluentd	Google Cloud Platformで採用されたログコレクター	https://github.com/fluent/fluentd
Go	Google社で開発されたプログラミング言語	https://github.com/golang/go

注7) <http://git-scm.com/>

注8) <https://github.com/>

注9) <https://github.com/pricing>

注10) <https://education.github.com/>

注11) <https://github.com/explore>

注12) <https://github.com/trending>

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

また、GitHubを使って今までとはちょっと違った試みをされている、ソフトウェア開発以外のプロジェクトがいくつかあります。たとえば、ドイツの法律をGitHubで管理し、変更内容を追えるようにしているプロジェクト^{注13)}があります。日本では和歌山県がオープンデータ^{注14)}として、県内の避難先情報や道路規制情報などを公開しています。

筆者自身も書籍の原稿を書いていくのに編集者とともにGitHubを利用するなど、とくにコンピュータ関連の雑誌や書籍の作成にも利用されている事例が多数あります。

このようにソフトウェア開発だけではなく、さまざまな場面でGitHubが利用されています。



GitHubが提供するおもな機能

GitHubには、開発者が良いコードを効率的にアウトプットするための機能が豊富にあります。ここでは、それらの機能の概要を説明していきます。

Gitリポジトリ

GitHubで提供するGitリポジトリは、基本的には無料で何個でも作成できます。ですが、限られた人や自分だけに公開を制限したいようなプライベートリポジトリを作成したい場合は、毎月7ドルからのプランに応じた金額を支払うことで利用できます。

Organization

通常、個人であれば個人アカウントを使えば良いのですが、会社で使うような場合にはOrganizationアカウントの利用をお勧めします。アカウントや権限の管理を一括して行える、支払いを統一できるなどのメリットがあります。

公開リポジトリしか使わないのであれば無料でOrganizationアカウントを作成できるので、勉強会やIT系のコミュニティでソフトウェアを開発するときには活用してみてはいかがでしょうか。

ビューア

GitHubはリポジトリに格納されたコードを閲覧するためのビューア機能を提供しています。各種言語のシンタックスハイライトにも対応し、快適にコードを閲覧、検索できるように作られています。

また、CSVやTSVデータを表形式に表示したり^{注15)}、画像ファイルの差分を表示したり^{注16)}、3Dファイルをレンダリングして表示したり^{注17)}など、データそのものを人間が認識しやすい形で表示してくれる機能もあります。

Issue

Issue機能とは、1つのタスクや問題を1つのIssueに割り当てて、トラッキングや管理を行えるようにするための機能です。バグ管理システムのような使い方やチケット駆動開発のチケットのような使い方ができます。GitHubでは後述するPull Requestが行われた際も、同時にIssueが1つ発行されます。

1つの機能変更や修正などに対して1つのIssueが割り当てられ、論議や修正などはそのIssueを中心として行われます。Issueを見ればその変更に関することがすべてわかるよう管理ができるのです。

Gitのコミットメッセージに「#7」のようにIssueの発行IDを書き加えると、GitHubでは自動的にIssueからcommitに対してリンクが張られます。また特定のフォーマットに基づいて

注13) <https://github.com/bundestag/gesetze>

注14) 国や地方公共団体が公開する行政情報などのうち、コンピュータで扱いやすいデータ形式で、かつ二次利用が可能なルールで提供されるデータ。

注15) <https://help.github.com/articles/rendering-csv-and-tsv-data/>

注16) <https://help.github.com/articles/rendering-and-diffing-images/>

注17) <https://help.github.com/articles/3d-file-viewer/>



第2章

リポジトリ作成からCIツール等との連携まで

GitHub入門

コミットメッセージを記述すれば、IssueをCloseすることもできます。非常に便利な機能ですので、ぜひ実践してください。

Wiki

Wiki機能は、いつでも誰でも文章を書き換えて保存できるため、共同で文章を作成できます。開発ドキュメントやマニュアルなどの記載に使われていることが多いです。記法はGFM(GitHub Flavored Markdown)^{注18)}で記述できます。

WikiページもGitリポジトリとして管理されており、改版履歴がしっかりと残るので、安心して書き換えを行えます。cloneして編集もできるため、プログラマがプラウザを立ち上げずに利用することも可能です。

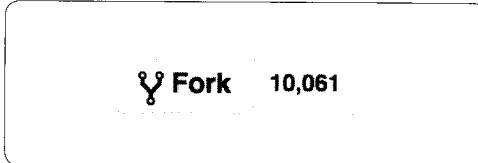
Fork

ForkはGitHub内においてリポジトリを自分の権限のあるリポジトリとして複製する機能です。

たとえばRuby on Railsのプロジェクトは、GitHubではrails Organizationのrailsリポジトリですので、GitHubの「rails/rails」で開発されています。このリポジトリへ変更を加える権限はコミッターにしかありません。それ以外の人は直接の変更を加えることはできません。このリポジトリのコードに修正を加えたければ、リポジトリ右上の「Fork」ボタン(図2)を押せば、同じリポジトリが「ユーザ名/rails」で複製されます。複製されたリポジトリは自分のリポジトリですので、自由に変更できます(図3)。

バグなどを修正するのであれば、Forkした自分のリポジトリでコードを修正して、Fork

▼図2 Forkボタン



注18) <https://help.github.com/articles/github-flavored-markdown/>

元のリポジトリに対して修正した内容のPull requestを作成する形になります。GitHubで公開されているOSSへの貢献は、多くがこのような形で行われます。

Pull Request

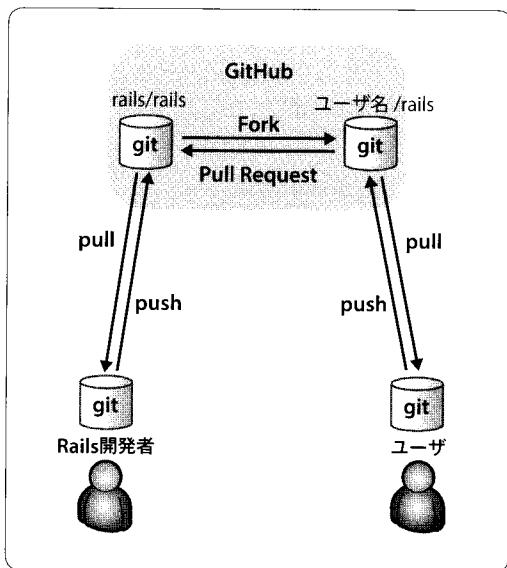
Pull Requestは、あなたがGitHubのリポジトリにpushした変更や機能追加をほかの人のリポジトリに取り込んでもらうための要求を出す機能です。

Pull Requestが送信されると、送信先のリポジトリの管理者などは送られてきたPull Requestの内容や含まれているコードの変更などを確認できます。そこでは、Pull Requestやソースコードの差分などについてレビューや論議をするための機能があります。ソースコードの行単位でコメントを付けたりできるので、プログラマ同士で効率的なコミュニケーションを取ることができます。

GitHubを使うとチームはどうなる?

GitHubを利用するとなると、Pull Request

▼図3 Forkのイメージ図



楽しく
始める、新規さん歓迎特集！

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

を作成してソフトウェアを開発していくスタイルになるかと思います。こうした1つの単純なワークフローを中心にGitHubとほかのさまざまなツールやサービスを連携させることによって、GitHubを中心としてソフトウェア開発が可視化されていきます。

GitHubを利用するだけでも、Pull Requestによりソフトウェアがどのように変化していくのかが目に見え、差分をレビューしながら開発していくことが、とてもやりやすくなります。複数のプロジェクトのリポジトリが更新されていく様子をNews Feed(図4)やメールなどさまざまな形で知ることができ、情報を追っていくことが容易になります。開発者自身が関連して

いる重要な情報はNotificationsでもれなく確認できます(図5)。

また、のちほど紹介するチャットサービスやCIのツールを利用することによって、GitHubを中心として、Pull Requestへのコメントなどをほぼリアルタイムに把握することや、自動テストを実施した結果などを労力少なく把握・管理していくことを、チーム全員ができるようになります。

GitHubによってソフトウェアの状態が可視化されるようになると、チームでソフトウェアを開発している場合、多くの人の目に触れる事になるので、人の目の数によって質を上げられるようになります。また、GitHubが文字ど

おりHubとなってほかのツールやサービスと連携するので、目の数で質を向上させるだけでなく、ツールやサービスの恩恵によってソフトウェアの質が上がる、開発者の労力が削減されるなど、多くの恩恵を受けられます。

本章の後半では、世界中で利用されているGitHubを開発者がどのような組み合わせで利用しているのかについても紹介していきます。読者のみなさんの開発現場の参考になれば幸いです。

▼図4 News Feed

The screenshot shows the GitHub interface with the 'News Feed' tab selected. It displays a list of recent events:

- A user closed an issue in rails/rails#13887.
- A user commented on an issue in rails/rails#13887, mentioning database adapter replacement.
- A user opened an issue in rails/rails#13887, asking about binding replacement.
- A user commented on a pull request in rails/rails#13885, discussing association selection.
- A user commented on a pull request in rails/rails#13886, pointing out test cases returning nil.

On the right sidebar, there are sections for 'Better Organizations' (with a note about managing organizations), 'Repositories you contribute to' (github-book, github-fizzbuzz), and 'Your repositories' (gh-prb).

▼図5 Notificationsの画面

The screenshot shows the GitHub interface with the 'Notifications' tab selected. It displays a list of unread notifications:

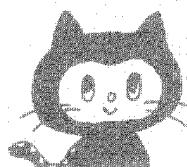
- Participating: 0 notifications.
- All notifications: 16 notifications.
- github-book/fizzbuzz: 16 notifications, all marked as read.

The notifications for the 'github-book/fizzbuzz' repository are listed:

- [WIP] Add output GitHub
- 7 case output GitHub
- Add output GitHub
- Add output GitHub
- Add output GitHub
- Add output GitHub when the number includes 7
- Add output GitHub
- Add output GitHub
- Add output GitHub
- Aug 28, 2014

まとめ

本節では「GitHubとは何なのか？」から始まり、世界中で利用されている現在の状況、機能について解説しました。開発者にとっては見過ごせないサービスであることを実感できたのではないかでしょうか？ 次節からGitHubを本格的に利用していきます。



第2章

リポジトリ作成からCIツール等との連携まで

GitHub入門



2-2 GitHubを利用するための最初の一歩☆△

本節では、GitHubを利用するためには必要な事前準備や、コードなどを公開する最低限の操作について解説します。



利用前の準備

GitHubのサービスを本格的に利用する前に最低限の準備の設定を行います。

アカウントの作成

さっそく、GitHubのアカウントを作成しましょう。GitHubのトップページからアカウントを作成できるようになっています(図6)。お使いのブラウザでGitHubのトップページ^{注19}へアクセスしてください。

①の「Pick a username」には希望するIDを英数字で入力してください。これは、あなたの公開ページのURLである「<http://github.com/○○>」の部分としても使われます。そのほかの項目も画面の指示どおりに入力してください。

すべての項目を入力後、②の「Sign up for GitHub」をクリックします。入力項目に問題がなければ、プランの選択画面に遷移します(図7)。とりあえずアカウントを作成するだけであれば、③の無料のFreeプランを選択してください。有料プランへの変更はあとからでも実施できます。希望するプランを選択して④の「Finish sign up」をクリックすればアカウント作成は終了です。そのあとは作成したアカウントでサインインした状態になり、ダッシュボードが表示されているはずです(図8)。この状態でサービスを利用できるようになりました。ログイン中はページ右上にユーザー名が表示された状態になります。

▼図6 GitHubのトップページ

The screenshot shows the GitHub homepage with a large banner reading "Build software better, together." Below it, a sub-banner says "Introducing a faster, more flexible GitHub Enterprise." A sign-up form is visible with fields for "Pick a username" (marked ①), "Your email" (marked ②), and "Create a password". Below the form, there's a note about powerful collaboration, code review, and code management for open source and private projects.

▼図7 プランの選択画面

The screenshot shows the "Welcome to GitHub" page after sign-up. It displays a "Completed Step 1: Set up a personal account" message. The "Step 2: Choose your plan" section is shown, featuring a table of personal plan options: Large (\$50/month), Medium (\$22/month), Small (\$12/month), Micro (\$7/month), and Free (\$0/month). To the right, a sidebar lists "Each plan includes:" features like unlimited collaborators, SSL Protection, and email support. At the bottom, there's information about organization plans and a "Finish sign up" button (marked ④).

▼図8 ダッシュボードの様子

The screenshot shows the GitHub dashboard titled "GitHub Bootcamp". It features several sections: "Set up Git" (with a quick guide), "Create repositories" (describing how to fork), "For repositories" (about creating unique projects), and "Work together" (about sending pull requests). On the right, there's a "Your repositories" section with a note about creating first repositories and a "Pro Tip!" for editing the feed. The top navigation bar includes "Explore", "List", "Blog", "Help", "News Feed", "Pull Requests", and "Issues".

注19) <https://github.com/>

楽しく
始める、新規さん歓迎特集！

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

▼図9 Settingsのアイコンは歯車



アイコンの設定

GitHub上の随所で表示されるアバター(アカウントごとのアイコン)は、GitHubを利用するうえで必須のものではありません。ですが、コードを書いた人の顔やアイコンが見えたほうが安心できますし、その人に興味を持つきっかけになるかもしれません。人にフォーカスできるGitHubだからこそ、積極的にアバターを設定することを推奨します。

設定する方法は、右上のメニューbaruにあるSettingsの歯車アイコンをクリックしてください(図9)。Settingsの最初のページの「Public profile」の項目に「Profile picture」があります(図10)。「Upload new picture」ボタンをクリックして、利用する画像をアップロードして設定してください。以上の操作で、あなたのアイコンが各所に表示されるようになります、ほかの人からも認識しやすくなります。

SSH Keyの設定

GitHubでは、作成したリポジトリへのアク

▼図11 公開鍵と秘密鍵の作成

```
$ ssh-keygen -t rsa -C "your_email@example.com"
Generating public/private rsa key pair.
Enter file in which to save the key
(/Users/your_user_directory/.ssh/id_rsa): ↪[Enter]キーを押す
Enter passphrase (empty for no passphrase): ↪パスフレーズを入力
Enter same passphrase again: ↪再度パスフレーズを入力
```

▼図12 公開鍵と秘密鍵の作成(実行結果)

```
Your identification has been saved in /Users/your_user_directory/.ssh/id_rsa.
Your public key has been saved in /Users/your_user_directory/.ssh/id_rsa.pub.
The key fingerprint is:
[Redacted] your_email@example.com
The key's randomart image is:
+--[ RSA 2048]----+
| .+ + |
| = o 0 . |
(...略...)
```

▼図10 アイコン用の画像の登録



セス認証をSSHの公開鍵認証で行います。公開鍵認証に必要なSSH Keyの作成と、GitHubへの公開鍵登録を行います。すでに作成している人は既存の鍵を利用および設定してください。

お手元のCLI環境で図11のように実行して、SSH Keyを作成します。「your_email@example.com」の部分はGitHubに登録した自分のメールアドレスに変えてください。パスフレーズは認証の際に入力します。覚えやすく、かつ複雑なものを推奨します。

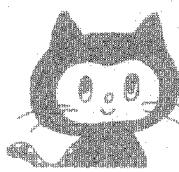
パスフレーズを入力すると、図12のように出力されます。id_rsaというファイルが秘密鍵で、id_rsa.pubが公開鍵です。

公開鍵の登録

GitHubに公開鍵を登録して、秘密鍵を用いてGitHubと認証できるようにします。右上のSettingsボタンを押し、「SSH Keys」のメニューを選択してください。「Add SSH key」を押すと、

図13のような入力欄が表示されます。

⑤のTitleには適当な鍵の名前を入力してください。⑥のKeyにはid_rsa.pubの内容をコピーして貼り付けます。id_rsa.pubの内容は図14のようにして参照してください。



第2章

リポジトリ作成からCIツール等との連携まで GitHub入門

▼図13 Add SSH keyを押すと表示される入力画面

SSH keys
There are no SSH keys with access to your account.
Add an SSH key
Title
Key
5
6

登録が無事に済めば、登録したメールアドレスに公開鍵登録完了のメールが届くはずです。

以上の設定が終了すれば、手元の秘密鍵を利用してGitHubとの認証と通信ができるようになります。実際に動作確認をしてください(図15)。図16のように表示されれば成功です。これでリポジトリをpush、pullする際に認証される公開鍵の設定ができました。



初めての公開リポジトリ

ここでは、公開リポジトリを作成する流れを解説します。まず右上のツールバーにある「New repository」(図17)というアイコンをクリックして、リポジトリを新たに作ります。

Repository name/Description

リポジトリを新規作成するための画面(図18)では、⑦の「Repository name」にリポジトリの名前を入力します。今回は「hello-github」と入力してください。⑧の「Description」にリポジトリの説明を設定できます。必須ではありませんから空白でも大丈夫です。

Public/Private

⑨で「Public」と「Private」が選

▼図14 公開鍵(id_rsa.pub)の内容を表示

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa 公開鍵の内容 your_email@example.com
```

▼図15 GitHubとの認証の確認

```
$ ssh -T git@github.com
The authenticity of host 'github.com (207.97.227.239)' can't be established.
RSA key fingerprint is フィンガープリント.
Are you sure you want to continue? [yes/no]? yesと入力
```

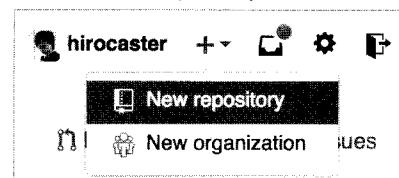
▼図16 認証時に表示されるメッセージ

```
Hi hirocastest! You've successfully authenticated, but GitHub does not provide shell access.
```

択できるようになっています。今回は「Public」を選択してください。公開リポジトリとして作成され、リポジトリの内容はすべて公開されます。

「Private」を選択するとアクセス制限を設定できる非公開リポジトリを作成できますが、有料プランの契約が必要となります。

▼図17 New repository



▼図18 New repositoryをクリックしたあとの画面

Owner: hirocaster
Repository name: 7
Great repository names are short and memorable. Need inspiration? How about shiny-dangerzone.
Description (optional):
⑨ Public
Anyone can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
⑩ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.
⑪ Add .gitignore: None
Add a license: None
⑫
⑬

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

Initialize this repository with a README

⑩の「Initialize this repository with a README」にチェックを入れてください。GitHubのリポジトリの初期化とREADMEファイルの設置を自動的に行ってくれるため、作成直後からこのリポジトリをcloneできるようになります。

に含めるコードのライセンスを決めている場合は選択してください。ライセンスの内容が書かれたLICENSEファイルがリポジトリに作成されます。このリポジトリに含まれるものライセンスを表明することになります。

入力や選択を完了したら、⑪の「Create repository」をクリックしてください。これでリポジトリが完成しました。

Add .gitignore

⑫のプルダウンメニューが便利ですので覚えておいてください。これは.gitignore^{注20}ファイルを初期化時に作成してくれます。この設定をしておけば、一般的にGitリポジトリに含めてバージョン管理しなくても良いファイルをあらかじめ.gitignoreファイルに記述してくれるので、毎回フレームワークに合わせて設定する作業から解放されます。プルダウンメニューには主要な言語やフレームワークがあるので、利用予定のものを選択してください。今回はフレームワークなどは利用しないので選択しません。

リポジトリにアクセス

「Create repository」をクリックすると、すでに作成されたリポジトリにアクセスしていると思います(図19)。次のURLが、今回作成したリポジトリのページです。

<https://github.com/ユーザー名/hello-github>

現時点からこのリポジトリは公開された状態になっています。リポジトリに入っているファイルはREADME.mdファイルだけです。先ほどチェックマークをしたことによって作成されたファイルです。

Add a license

⑬のプルダウンメニューは、追加するライセンスファイルを選択できます。このリポジトリ

README.md

README.mdファイルはリポジトリのトップページに自動的に中身が表示されます。したがって、このリポジトリに入っているソフトウェアの概要、利用するための手順、ライセンスなどが明記されているのが一般的です。Markdown記法で記述すれば、とても読みやすくマークアップしてくれます。



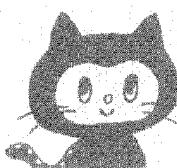
リポジトリに変更を
加え、反映する

ここまでで作業でリポジトリを作成できました。通常のソフトウェア開発であればこのリポ

▼図19 作成されたリポジトリの画面

The screenshot shows the GitHub repository page for 'hirocaster/hello-github'. The repository has 1 star, 1 branch (master), and 1 commit from hirocaster. The commit message is 'Initial commit'. The repository was created a minute ago at 64fac998cb. The README.md file contains the text 'hello-github'.

注20) Gitリポジトリでの管理対象外のファイル・ディレクトリを記述するファイルのことです。



第2章

リポジトリ作成からCIツール等との連携まで GitHub入門

▼図20 SSH clone URL

```
SSH clone URL
git@github.com:hir
You can clone with HTTPS, SSH, or Subversion. ?
```

▼図21 リポジトリをcloneする

```
$ git clone git@github.com:hirocaster/hello-github.git
Cloning into 'hello-github'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
```

▼リスト1 README.mdファイルの変更例

```
# hello-github
ソフトウェアデザイン（2015年6月号）を参考にGitHubへの練習をしているリポジトリです。
```

ジトリに対して機能の追加などをていきます。ここでは実際にリポジトリを手元の環境に持ってきて、変更を加えて、GitHubのリポジトリに反映させる一連の流れを解説します。

Gitの操作を行いますが、Gitを扱える環境は構築されているものとします。環境の構築のしかたについては第1章を参照してください。

リポジトリのURL

先ほど作成したGitHubのリポジトリのURLをブラウザで開いてみてください。右のサイドバーにこのリポジトリを取得するためのURLが掲載されています(図20)。「SSH」というリンクをクリックするとSSH経由でGitリポジトリへアクセスするURLがテキストボックスに表示されます。「HTTPS」を押すとHTTPSで参照するためのURLが表示されます。「Subversion」をクリックするとSubversionで利用するためのURLが表示されます。

「SSH」をクリックして、テキストボックスの右隣にある「Copy to clipboard」をクリックしてください。URLがClipboardへコピーされます。コピーされたURLは

```
git@github.com:ユーザ名/hello-github.git
```

となるはずです。これをgit cloneコマンドを利用して、手元の環境へリポジトリをcloneしましょう(図21)。

▼図22 変更をcommitする

```
$ git add README.md
$ git commit -m "Add description"
[master f3caacd] Add description
 1 file changed, 2 insertions(+)
```

現在のディレクトリの直下にhello-githubというディレクトリが作成されました。これが取得したリポジトリです。中にはREADME.mdファイルがあることが確認できます。

```
$ ls
README.md
```

ここまでで、GitHubのリモートリポジトリをcloneして手元の環境へリポジトリを複製することができました。

リポジトリに変更を加える

通常のソフトウェア開発であれば、機能の追加など、ファイルに変更を加えるのですが今回はREADME.mdファイルに変更を加えることとします。

お使いのエディタでREADME.mdファイルを開き、リスト1のようにリポジトリの説明を記載してみてください。

内容を記述したらファイルを閉じてしまつかまいません。この変更をcommitするところまで実施します(図22)。一連の流れの詳細については第1章を参照してください。

以上でリポジトリのファイルに変更を加えることができました。通常のソフトウェア開発であれば、いくつかのcommitに分けて変更作業をしていってください。

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

リポジトリをpushする

ここまでで手元の環境ではファイルを変更できました。ですが、GitHub側のリモートリポジトリへは反映されません。手元のリポジトリをGitHubのリモートリポジトリに対してpushすることで、変更した内容を反映しましょう。

pushをするためにはgit pushコマンドを利用します。何も指定しなければcloneしてきたリポジトリに対して、pushしてくれます(図23)。

▼図23 リモートリポジトリに変更内容をpushする

```
$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 380 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:hirocaster/hello-github.git
  64f6c99..f3caacd master -> master
```

▼図24 pushした後の様子

hirocaster / hello-github

Description: README, license, ready to deploy

Website: https://github.com/hirocaster/hello-github

Save or Cancel

2 commits | 1 branch | 0 releases | 1 stargazer

f3caacd · master · hello-github · +

Add description: hirocaster (modified 4 minutes ago)

README.md Add description

latest commit f3caacd@4 minutes ago

4 minutes ago

hello-github

ソフトウェアデザイン（2015年6月号）を参考にGitHubの練習をしているリポジトリです。

git pushの際にSSH Keyのパスフレーズを聞かれた場合は、入力して認証してください。このような表示にならず、うまくいかなかった場合はSSHの鍵まわりの設定や認証まわりを疑ってください。

無事にpushできていれば、再度ブラウザでGitHubにアクセスしてみると README.md ファイルの内容が更新されていることを確認できるかと思います(図24)。リポジトリのcommit 数なども増えているのが同時に確認できます。

このように自分が作成したリポジトリや書き込み権限があるリポジトリに対しては、直接pushできます。直接pushできない他人のリポジトリに関しては、ForkしてPull Requestを投げるなどをします。



本節ではGitHubにアカウントを作成し、鍵などの最低限の設定をしました。秘密鍵についてはリポジトリの認証に利用されるものですので、厳重に管理してください。また、GitHubにリポジトリを作成して、リポジトリの内容に変更を加え、反映しました。これでコードなどを公開する最低限の操作を一通りやってみた形になります。

2-3 GitHub Flowを利用した開発の流れ

GitHub Flowを利用した開発の流れ

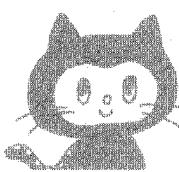
この節では、GitHubを利用した開発フローである「GitHub Flow」について解説します。このワークフローはGitHub社が実践しているとてもシンプルなワークフローです^{注21}。筆者自

身もGitHub Flowを数多くのプロジェクトで利用しています。

このワークフローは、デプロイ^{注22}を中心としたワークフローです。実際の開発の現場では

注21) <https://speakerdeck.com/holman/how-github-uses-github-to-build-github>

注22) ソースコードを本番環境に配備し稼働させること。



第2章

リポジトリ作成からCIツール等との連携まで

GitHub入門

1日に何十回とデプロイを実施します。それを支えるのは、今回解説するシンプルなワークフローと徹底的に自動化された環境です。シンプルにするためにForkなどの機能は利用しません。関係する開発者全員を信頼してGitHubの1つのリポジトリに対して権限を与えます。そうすることで、あらゆることに柔軟に対応できます。GitHubを利用するのであれば、ぜひこのワークフローを採用してください。

このワークフローは小さなチームでも、大きなチームでも効果的に機能します。GitHub社では、このフローを利用して15~20人が同じプロジェクトで作業しているようです^{注23}。筆者の経験からも、同じプロジェクトで20名程度ぐらいまでは、このワークフローを利用して大きな問題が発生したことはありません。



GitHub Flowの流れ

ワークフローの全体は次のようにになります。

1. master ブランチは常にデプロイできる状態とする
2. 新しい作業をするときは master ブランチから記述的な名前(後述)のブランチを作成する
3. 作成したローカルリポジトリのブランチに commit する
4. 同名のブランチを GitHub のリポジトリに作成し、定期的に push する
5. 助けてほしいときやフィードバックがほしいときは Pull Request を作成し、Pull Request でやりとりする
6. ほかの開発者がレビューし、作業終了を確認したら master ブランチにマージする
7. master ブランチへマージしたら、ただちにデプロイする

基本的には特定の作業をするブランチを作成するだけですので、作業を始めてデプロイする

までの過程がとてもシンプルです。これはワークフローを実施するまでの学習コストを抑えられるという利点があります。それよりも大きな利点として、シンプルであるからこそ、多くの開発者がすばやく行えることを可能にします。そして、小さな変更などにも柔軟に対処できるようになります。

これから上記フローのひとつひとつの項目について順番を追って解説していきます。

➡ master ブランチは常にデプロイできる状態

このワークフローで絶対に守らなければならないルールは「master ブランチは常にデプロイできる状態にする」ということです。通常の Web サービスの場合、数時間ごとに常にデプロイをすれば、大きなバグが複数入り込むことはまずありません。小さなバグが入り込むことはあるかもしれません、該当の commit を revert する(変更前の状態にする)か、修正されたコードを commit するなどして、即座に対応可能です。このように数分から数時間単位で継続的にデプロイが実施されるため、リリースという概念がこのワークフローには存在しません。よって、作業内容を取り消すために HEAD を古い commit に差し戻す^{注24}ことはありません。

master ブランチは常にデプロイ可能であるため、新たにブランチを作成することも常に可能となります。

基本的なルールとして、テストが書かれていないコードや、テストを落としてしまうようなコードを絶対に master ブランチに入れてはいけません。そのため、継続的インテグレーションなどのアプローチは必須となります。誌面の都合上、これについての解説は行いません。

➡ master ブランチから新ブランチを作成

新しい作業をするときは、master ブランチ

注23) <http://scottchacon.com/2011/08/31/github-flow.html>

注24) Git でいう git reset コマンドに該当する作業。

Git & GitHub 入門



から新たにブランチを作成します。新機能の追加でもバグの修正でも、新しくブランチを作るという作業は変わりません。ブランチ名には記述的な名前を付けてください。

記述的な名前とは、そのブランチの特質をそのまま正確に表したような名前です。たとえば、

- ・ user-content-cache-key
- ・ submodules-init-task
- ・ redis2-transition

のように、ほかの開発者が何をしているのかを想像できる具体的な名前が望ましいです。

これにより、リモートリポジトリのブランチ名一覧を確認すれば、チームがどんなタスクを実施しているのかが一目でわかります。また、何をするべきブランチなのかが明確なため、いつたんほかの作業をして戻ってきてもすぐに思い出せます。GitHubのブランチリストのページ^{注25}を見れば、それぞれのブランチと master ブランチの差分なども容易に確認できます。

新ブランチにcommitする

ここまでこの状態で、開発者が変更を加えるべき新しいブランチが作成できたと思います。そして、そのブランチでは何をするかが明確になっているはずです。そのため、トピック以外の作業の commit などはこのブランチにしないようにコードに変更を加え、commitしてください。

ここでポイントなのは、Pull Request をレビューする開発者のためにも、意図が伝わる commit の粒度を心がけてください。そのため commit ひとつひとつのサイズは小さくすることを意識してください。

たとえば、メソッド1つを追加する作業だけでも、追加すべき個所やクラスを特定したあとに、開発者は次のようなことを日ごろから行っているのではないでしょうか。

- ・近くのコードのインデントが崩れていたので適切に修正

- ・変数の単語の間違いを見つけたので正しい単語に修正

- ・今回の作業として追加すべきメソッドを追加

これらの一連の流れを1つの commit とする1つの差分に3つの意味が含まれてしまうため、望ましい commit の粒度とは言えません。それぞれ3つの commit として分けたほうが、それぞれの差分の意図が伝わるはずです。

このようなポイントを踏まえ、通常の開発どおりこのブランチにコードの変更を commit していってください。

pushする

このワークフローでは master ブランチ以外は作業中のブランチとなるため、気軽に作業中のブランチを push できます。ローカルリポジトリで作成したブランチと同じ名前のブランチで、GitHubのリモートリポジトリに定期的に push してください。

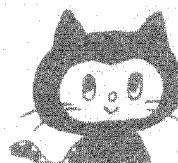
この際に Pull Request を作成してしまうことを筆者は推奨します。この Pull Request はまだ作業中であることを表明するために、Pull Request のタイトル冒頭に “[WIP]”^{注26} と付けるようにしましょう。こうすることにより、ほかの開発者が誤って作業途中の Pull Request をマージしてしまうことを防ぎます。これは GitHub 利用者のプラクティスです。

これにより、コードのバックアップにもなりますし、チームの開発者が定期的にコミュニケーションをする機会を与えられます。ほかの開発者が何を実施しているのか、助けを必要としているいかなど、GitHub の Pull Request とブランチリストのページを利用して全員が確認できます。

自分が書いたコードをほかの開発者が見られ

注25) <https://github.com/ユーザ名/リポジトリ名/branches>

注26) “Work in Progress”的略。



第2章

リポジトリ作成からCIツール等との連携まで

GitHub入門

る状態にし、また積極的にほかの開発者のコードを見る習慣のあるチームになります。コードでコミュニケーションを取れるのは開発者に許された特権です。この権利を活用しない手はありません。



Requestを使う・活用する

Pull Requestは、master ブランチにマージしてほしいときだけに作成するものではありません。チームで開発を行うのであれば、master ブランチへマージするタイミングよりもずっと前から Pull Requestを作成し、レビューをしてもらうなどして、フィードバックを得ながら開発するべきです。

Pull Requestには、差分を閲覧する機能やコードの行にコメントを入れられる機能があります。これを活用してコミュニケーションをしてください。また、特定の開発者からフィードバックやアドバイスを得たいのであれば、コメントの中に「@ユーザ名」と入力すれば、そのユーザに Notifications が飛びます。それに気づいて、何らかのフィードバックをしてくれるはずです。

開発者からレビューを受ける

ブランチでの作業が完了したら、完了したこととを表明（“[WIP]”を削除して、完了した旨を伝えるなど）して、ほかの開発者によるレビューを受けてください。ほかの開発者の目を通すことによって、思い込みやミスを防ぐことができます。コードを書いた人以外がレビューしてください。問題があれば指摘してもらい、修正を行ってください。もちろん、自動テストは全部パスしていることが大前提です。

レビューは master ブランチにマージして問題ないと思うのであれば、その意思を表明しましょ

▼図25 +1の絵文字

hirocaster commented on Feb 17, 2013

う。「:+1:」や「:shipit:」などの記法で絵文字を利用するのも GitHub の文化です（図25）。LGTM という「Looks good to me」の略語をコメントするケースもあります。

複数人の同意を得られたら、適切なタイミングでほかの開発者が master ブランチにマージしてください。

マージ後はすぐにデプロイ

master ブランチにコードがマージされ、自動テストがすべてパスしたら、ただちにデプロイしてください。デプロイして、マージしたコードが問題ないことを確認してください。

まとめ

GitHub を複数人のチームで利用していく流れが見えたのではないですか？ GitHub Flow は非常にシンプルなワークフローですので、すぐに理解して実践できると思います。実践する中でチーム独自のルールを加えていき、よりチームに最適化されたワークフローへと進化させてください。

Pull Request 上でのやりとりに慣れれば、今まで以上にコードの意図を伝えやすくなり、レビューが捲るはずです。Pull Request をうまく使うことがこのワークフローを成功させるポイントです。あなたの現場のチームでこのワークフローを採用し、より GitHub を活用していただければ幸いです。



2-4 GitHubを中心として利用されるサービス・ツール

GitHubを積極的に活用している開発の現場では、GitHubを利用してPull Requestのスタイルで開発することは当たり前になってきています。このGitHubを利用した開発に対して、ほかのサービスやツールをどのように組み合わせて、何を実現しているのかといったことに各組織やチームで特色が出てきています。その中からいくつか紹介したいと思います。これを参考に、開発の現場で採用できるものがないか、ご検討ください。

誌面の都合上、詳細な手順の解説などができません。できる限り、参考になるURLなどを記載しましたので、実際の導入時にはそちらを

▼図26 数多くのサービスと連携できる

Available Services
Add service ▾
Filter services
AWS CodeDeploy
AWS OpsWorks
ActiveCollab
Acunote
AgileBench
AgileZen
AmazonSNS
Apiary
Apoio
AppHarbor
Apropos
Asana

参考にしていただけすると幸いです。

いずれもGitHubのリポジトリの「Settings」→「Webhook & Services」から連携の設定ができます。各種環境を構築後、もしくはサービスの契約設定後にこの画面からWebhookやServiceとの連携を設定します。今回紹介できていないサービスもたくさんあるので、一度目を通してみるとお勧めします(図26)。



チャットサービスとの連携

近年、開発のコミュニケーションをチャットで行う現場が増えてきています。今までメールを使って行っていたコミュニケーションなども、チャットにシフトしてきているのが現状です。よって、GitHubを含めたチャットサービスの利用のされ方について解説したいと思います。

開発と相性の良いチャットサービス

開発の現場でよく使われるサービスとして「Slack」注27や「HipChat」注28などが挙げられます。これらはAPIが公開されていることやスマートフォンから利用できることはもちろんですが、GitHubをはじめとしたほかのサービスへの連携が気軽に設定できるように設計されています。そのため、開発者が何かしようとしたときに、やりやすい環境が整っているので、よく使われています。

なぜチャットを利用するのか？

GitHubと連携するチャットサービスを利用すれば、「GitHubで新しくPull Requestを作成した」「コメントをした」「差分をpushした」などの各種通知を簡単にチャットに流せます(図

注27) <https://slack.com/>

注28) <https://www.atlassian.com/ja/software/hipchat>



第2章

リポジトリ作成からCIツール等との連携まで GitHub入門

27)。これによって、チームメンバー全員でリアルタイム(もしくは非同期)に情報を共有できるようになります。通知から話題を広げて、ちょっととしたコミュニケーションをすることもできます。

また、スマートフォンに対応したチャットサービスであれば、外出先で情報の確認もできますし、急用があればユーザ間でNotificationを送ることによってスマートフォンにpush通知を送って相手を呼び出すことも気軽にできます。

このスピード感をメールで実現するのは、なかなか難しいものがあります。そのことからメールやほかのツールで行われていたコミュニケーションがチャットに移ってきている組織が増えています。

◆ チャットbot、ChatOps

各種サービスの通知をチャットサービスに連携するだけでなく、チャットbotと呼ばれるプログラムをロボットアカウントとしてチャットに常駐させ、チャット利用者が入力した文字やサービスの通知に反応して、一定の動作をさせるプラクティスがあります。

SlackやHipChatなど主要なチャットサービスに対応しているチャットbotとして、GitHub社で開発された「Hubot」などが挙げられます。豊富なプラグインから機能を柔軟に追加することができます。サーバを用意せよともHerokuなどで動作させられるので、気軽に試してみてください^{注29)}。

このようなチャットbotで自動化された運用オペレーションを実施させることをChatOpsと呼びます。ChatOpsの代表的なオペレーションとしては、デプロイが挙げられます。チャットで「hubot deploy」といった特定の文字を入力

▼図27 SlackにGitHubの内容が通知されている様子

```

github [001 457154]
[hello-github] New branch "sample-pr" was pushed by hirocaster

[hirocaster/hello-github] Pull request submitted by hirocaster
#1 Sample pr
サンプルのPull requestです。

[hirocaster/hello-github] New comment on pull request #1: Sample pr
Comment by hirocaster
テストコメント

```

することにより、チャットbotがデプロイの一連の処理を実施するのです。

チャットをトリガーにしてデプロイを実施することにより、チャットにいるメンバー全員に「いつ、何を、どのバージョンをデプロイした」といった情報がデプロイ作業と一緒に共有できるのです。このような操作はチャットで文字入力さえできれば実現できるので、外出先でスマートフォンからのデプロイも可能になります。

現在のシステムはさまざまなコンポーネントで構成されているため、システムとして稼働させるまでの手順が複雑になります。こういったチャットに特定の文字を入力するだけの単純なオペレーションを前提にシステムを構築していくれば、デプロイ作業は自動化せざるをえません。これにより、チームに入ったばかりのあまり慣れていないメンバーでも、気軽に安全にデプロイできるようになります。

◆ Pull Requestを起点とした連携

GitHubをチーム開発に導入すると、Pull Requestを作成してレビューを繰り返しソフトウェアを成長させていくことになります。このPull Requestを起点としてさまざまなツールと連携されることにより、効率的かつ安全にソフトウェアに変更を加え続けていけるようになります。そのような例をいくつか解説します。

^{注29)} Qiita「YoemanでHubotを作成してHerokuへデプロイしSlackと連携する」
<http://qiita.com/hksusu/items/1dc7db9607ab8cb35150>

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

コードィングスタイルチェック

プログラミング言語のほとんどは、同じ処理を記述するのにも複数の書き方ができます。そこで、可読性を上げるためにある一定のルールに沿ったコードの書き方を定義したものをコードィングスタイルと呼びます。

ソースコードは書かれることよりも、読まれることのほうがはるかに頻度が多くなります。そのため、読みにくいコードや変則的な記述のされ方をしたコードは、誤解やバグなどを生じさせてしまうリスクを抱えています。このリスクを回避するのがコードィングスタイルを導入する目的です。プログラマが一定のコードィングスタイルに沿って記述することによって、コードの読みやすさを向上させ、長期にわたって一貫性を保つことができます。

このような背景から複数人のチームでソフトウェアを開発する場合ははらかのコードィングスタイルに沿ってソースコードの書き方を統一するのが一般的です。

代表的なコードィングスタイル

これはオープンソースの世界でも同じことが言えます。各種言語やソフトウェアにはコードィングスタイルやルールが存在します。有名なところだとPythonのPEP8^{注30}、GitHub社で使われているRuby Style Guide^{注31}があります。国内企業だとクックパッド株がRuby^{注32}、Objective-C^{注33}、Java^{注34}のスタイルガイドを公開しています。

スタイルチェック

コードを書くためのルールを厳密に決めても、人がコードを書いている限りミスをしてしまうはある程度避けられません。GitHubの

Pull Requestをレビューする際に、こういったミスを見つけることはできますが、すべてを人間が検出するのは難しく、この行為自体があまり効率的ではありません。そこでツールを使って検知するアプローチが存在します。たとえばRubyではRuboCop^{注35}というツールがよく使われます。Rubyのソースコードに対して実行すれば設定された記述ルールに沿っているかどうかのチェックが行われて、間違っている部分をCLIやHTMLなどの形式で表示して指摘をしてくれます。

こういったツールをプログラマ各人の環境で設定しなくとも、Pull Requestを作成した時点でソースコードの差分に対してチェックを行い、違反している個所があればPull Requestにコメントを付けてくれるようなサービスが出てきました。具体的にはthoughtbot社が提供している「HOUND」^{注36}などです。

Pull Requestのレビューの一貫としてコードィングスタイルのチェックが必ず行われて、誰もが見える形でコメントとして指摘がされるので、スタイルチェックとしては効果的です。人間の心理的に、指摘されれば直さざるをえないでしょうし、マージするほうも指摘されているものをマージしづらいという意識が働きます。

通常の開発の流れである「Pull Requestを作成し、レビューして、マージする」の中に、機械的に誰もが理解しやすく、すぐに直せる形で組み込むことがポイントです。

継続的インテグレーション(CI)／自動テスト

継続的インテグレーションはアジャイルソフトウェア開発手法の1つであるエクストリーム・プログラミング(XP)で実施されるプラクティスの

注30) <http://legacy.python.org/dev/peps/pep-0008/>

注31) <https://github.com/bbatsov/ruby-style-guide>

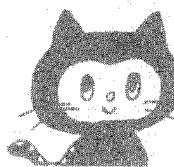
注32) <https://github.com/cookpad/styleguide/blob/master/ruby.ja.md>

注33) <https://github.com/cookpad/styleguide/blob/master/objective-c.ja.md>

注34) <https://github.com/cookpad/styleguide/blob/master/java.ja.md>

注35) <https://github.com/bbatsov/rubocop>

注36) <https://houndci.com/>



GitHub入門

1つです。狭義にはビルド、テスト、インスペクションなどを継続的に実施することです。

代表的なツールだと「Jenkins」^{注37)}、「Drone」^{注38)}などが挙げられます。JenkinsではGitHub pull request builder plugin^{注39)}などを利用してください。最近ではこういったCI環境を提供するサービスも増えてきています。代表的なものだと「Travis CI」^{注40)}、「CircleCI」^{注41)}、「wercker」^{注42)}などが挙げられます。

Pull Requestが作成された段階で、CIのツールやサービスによって自動テストを実施し、追加・変更・削除したコードの差分が既存の機能を壊していないかをチェックします。これで問題のあるPull Requestをマージしなくて済むようになります。

CIツールやサービスで実施した自動テストの結果などはPull Requestに反映され、ユーザに通知することができます(図28、29)。これは連携したCIツールやサービスがGitHubのAPIを通して結果を通知しているためです。

CIツールと連携させることにより、自動テストの恩恵を継続的に確実に受けることができるようになります。Pull Requestのレビューをする人は、自動テストでカバーされている範囲には意識を向ける必要があるため、人間が注目すべき設計や質の高いレビューに集中していくようになります。

カバレッジ

自動テストを継続的に実施するのであれば、コードカバレッジにも注目すべき

です。コードカバレッジとはコード網羅率と呼ばれることもあり、テストによりソフトウェアのソースコードがどの程度網羅されているかの割合を意味する言葉です。基本的にはテストコードでカバーされているソースコードの割合が多いほど、自動テストによって不具合を検知する確率は上がります。

これをレポートして開発者が見やすい形で提供するサービスなども登場してきています。代表的なものだと「Coveralls」^{注43)}などのサービスが挙げられます。サポートされている言語も徐々に増えており、一部の例を挙げるとC/C++、Go、Haskell、Java、JavaScript、Lua、.Net、Perl、PHP、Python、Ruby、Scala、Objective-Cなど広く普及している言語はサポートされています。オープンソースのプロジェクトによってはこのカバレッジ率を表示しているようなプロジェクトもあります(図30)。

新しい機能を増やしてカバレッジ率が下がれば、「テストコードを書いてね」とも伝えやすくなります。また、開発しているソフトウェアにおいて、自動テストがカバーしているコードの部分を把握できていると、Pull Requestでのレビュー時に「どこをとくに気をつけて見なけれ

▼図28 テストが成功した様子



▼図29 テストが失敗した様子



注37) <https://jenkins-ci.org/>

注38) <https://github.com/drone/drone>

注39) <https://wiki.jenkins-ci.org/display/JENKINS/GitHub+pull+request+builder+plugin>

注40) <https://travis-ci.com/>

注41) <https://circleci.com/>

注42) <http://wercker.com/>

注43) <https://coveralls.io/>

楽しく
始める、新規さん歓迎特集！

Git & GitHub 入門

It's as easy to learn as your



開発現場のはじめの一歩

▼図30 CucumberのREADME.mdに表示されているカバレッジ

README.md

build passing code climate 3.6 coverage 91% dependencies up-to-date

Cucumber

ばならないのか」という判断をするのにとても役立ちます。また、新たにテストを増やしていく際にも、現在の状態を把握することは考慮すべき内容です。

デプロイ

Pull Requestをマージした時点で、自動的にデプロイをしてしまう考え方もあります。ここでも活躍するのはCI関連のツールです。Travis CIでは「Heroku」^{注44}や「AWS CodeDeploy」など数多くのサービスと連携できるようになっています^{注45}。CircleCIやwerckerなどでも数多くのサービスとの連携を実現しています。自分で構築したければJenkinsや「RUNDECK」^{注46}などを利用すると構築できます。

また、特定のコマンドを実施するような設定もできます。既存システムでデプロイに「Capistrano」^{注47}などを利用していれば、Capistranoのデプロイコマンドを設定することによって、既存システムに組みやすくなっています。

各種サービスともに特定のブランチがマージされたらデプロイすることなども設定できます。いきなり本番環境で自動的にデプロイする環境を構築するのではなく、まずは開発している環境への導入を検討してみてください。developmentなどの特定のブランチへマージされたら自動的にデプロイされるしくみを作つてみてはどうでしょう

う。リリース前の確認・テスト・QAのために毎回デプロイしている作業を自動化するところから始められます。

Pull Requestスタイルで開発してマージのタイミングでデプロイまでされるようになると、どの環境で何がデプロイされているのかなどの情報共有がGitHubに集約されてくるので、今まで「どの環境に何をデプロイします」といったやりとりが減り、確実な記録を残しながら進めていくようになります。チャットシステムとの連携もしていれば、より速やかな情報共有となるはずです。

確認するための環境構築の自動化

Pull Requestをレビューする際に、UIなどビジュアライズするためのコードはコードを見ただけではレビューがやりにくいものです。Pull Requestにスクリーンショットの画像ファイルをドロップすれば画像がPull Requestに挿入できますので、レビューする側の人間はコードだけのレビューよりもわかりやすくなります。

これをさらに発展させたのが、Pull Requestごとに自動的に確認できる環境を作ってしまうことです。Herokuではベータ機能ながらPR apps^{注48}という機能を提供しています。これはPull Requestを作成すると、そのブランチの環境をHeroku側に自動的に作成してくれる機能です。自動的に作成した環境のURLをPull Requestに通知してくれるので、Pull RequestからURLをクリックするだけでブラウザ上で動作確認が可能になります。詳しくはHerokuのブログ記事「GitHub Integration: Pull Request Deploys」^{注49}を参照してください。

注44) <https://www.heroku.com/>

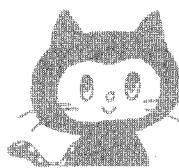
注45) <http://docs.travis-ci.com/user/deployment/codedeploy/> <http://docs.travis-ci.com/user/deployment/heroku/>

注46) <http://rundeck.org/>

注47) <http://capistranorb.com/>

注48) <https://devcenter.heroku.com/articles/github-integration#pr-apps>

注49) <https://devcenter.heroku.com/articles/github-integration-pull-request-deploys>



第2章

リポジトリ作成からCIツール等との連携まで

GitHub入門

これと似たような機能を自分で構築するのに、mod_mrubyとDockerを利用して環境を作る「pool」^{注50)}といったツールなどもあります。詳しくはQiitaで公開されている作者の記事^{注51)}をご覧ください。

このようなツールやサービスを組み合わせることにより、開発者も確認しやすく、非開発者も確認が容易にできる環境を整えられます。



Pull Request駆動でライブラリをアップデート

多くの開発者はライブラリのアップデートは後回しにしがちです。古いバージョンのライブラリを利用し続けていると、セキュリティ上のバグを抱えたままだったり、新機能の恩恵を受けられなかつたりします。また、システム全体の大きなバージョンアップの際には足を引っ張る原因にもなりかねません。

Pull Requestスタイルの開発方法をしているのであれば、ライブラリのアップデート用のPull Requestを自動的に作成してもらうという方法があります。「Tachikoma」^{注52)}は、bundler(Ruby)、carton(Perl)、david(Node.js)、cocoapods(Objective-C)、composer(PHP)などのライブラリをアップデートしたPull Requestを自動的に作成してくれるツールです。

Pull Requestを作成してくれれば、ここまで紹介したCIツールなどによって、システムのテストは自動的に行えますので、ライブラリのアップデートによりシステムが正常に稼働しているのか判断できるはずです。開発者はテストが無事に通っていることをレビューして、マージするだけです。これだけで最新ライブラリへのアップデート対応ができるようになります。

実際にはライブラリのアップデート内容などを把握することが必要になるかと思います。し

かし、機械的にアップデートするタイミングを作成してくれるだけで、確認するライブラリは都度最小のものになるので確認しやすく、アップデートしやすくなるはずです。

日々の開発と同じようにPull Requestを作成してもらい、レビューしてマージするという形になるだけでもカジュアルにライブラリのアップデートをしていけるようになります。今回紹介したTachikomaには「Tachikoma.io」^{注53)}という非公開リポジトリ向けの有料機能があります。こちらはTachikomaの環境を構築せずともGitHubのリポジトリと連携するだけで利用できるので、導入を検討してみてください。



まとめ

本節では、GitHubのさまざまな機能を中心としてほかのサービスと連携することにより、開発者がより快適に開発できるようになるプラクティスを数多く紹介しました。GitHub登場以降、多くのサービスがGitHubとの連携をサポートしています。

多くのさまざまなサービスの登場によって、CIやチャットシステムなどの開発環境全般を自分で構築・保守・運用し続けるのが必ずしも最適とは言えなくなりました。筆者はいくつかのサービスを組み合わせて利用し、自分たちが提供するプロダクトの開発に専念するというのも賢い選択の1つになってきていると感じています。

今回紹介した中で、組織やチームの問題を解決してくれそうなものがあれば、1つずつ導入して徐々に変化させていくことをお勧めします。



注50) <https://github.com/mookjp/pool>

注51) Qiita「mod_mrubyとDockerを使ってレビュー環境を作成するプロキシサーバを作った」
<http://qiita.com/mookjp/items/ed5961589428238d610b>

注52) <https://github.com/sanemat/tachikoma>

注53) <http://tachikoma.io/?setLang=ja-JP>