

\$ man setup-dev-env

Include: CUI basic, Git, Vagrant, Node.js, Ruby, Package Management, and more...

Web サイト制作の時流に乗り遅れないために、覚えておきたい開発環境の作り方

Development Environments for Web Designers

こもりまさあき

Development Environments for Web Designers

Web サイト制作の時流に乗り遅れないために、覚えておきたい開発環境の作り方

MASAAKI KOMORI

©2014 - 2015 MASAAKI KOMORI

Contents

はじめに	1
いまどきのサイト制作とは	3
変わり続ける Web サイト制作	4
閲覧環境が変わるということ	4
コンテンツの作り方ですら多様化	4
Web が Web サイトでなくなる?	7
最新の制作ツールはコマンドラインから	8
GUI だけではどうにもならない時代に	8
ライブラリのダウンロードもコマンドライン	10
コマンド操作だけができれば解決	10
ローカルで Web サイトを動かすには?	12
OS X なら実は簡単!?	12
MAMP や XAMMP の問題	12
新しい時代に対応するには	14
ターミナルの操作に慣れよう	15
シェル?	16
主なシェル	16
覚えておきたいシェルのコマンド	20
自分の居場所を表示する	20
ディレクトリの内容をリストする	21
コマンドのオプションを指定する	22
作業ディレクトリを移動する	23
入力補完を利用する	25
ヒストリー(入力履歴)を利用する	26
コマンド行のキャレットの移動	26
Finder でディレクトリを開く	28
ファイルの内容を表示する(テキストファイル)	28

CONTENTS

画面の内容をクリアする	29
新規テキストファイルを作成する	30
新規ディレクトリを作成する	30
ファイルやディレクトリを移動する(リネームする)	31
ファイルやディレクトリをコピーする	32
ファイルやディレクトリを消去する	33
複数のファイルやディレクトリをまとめて操作する	35
複数のファイルを結合する	35
ファイルやディレクトリの所有者やパーミッションを変更する	36
管理者としてコマンドを実行する	38
覚えておくと便利なコマンド	40
コマンドの場所を確認する	40
ファイルやディレクトリを圧縮(アーカイブ)・解凍する	41
SSHの鍵の作成	43
SSH(SFTP)でリモートのサーバにログインする	44
シンボリックリンクの作成	47
現在の日時を表示する	48
カレンダーを表示する	48
英語の発音を確認する	48
コマンドの実行結果を他のプログラムに渡す	48
複数のコマンドを一度に実行するには?	49
ターミナルでテキストを編集する	51
Vim(Vi)	51
nano	54
制作環境構築の下準備	56
Xcodeとコマンドラインツールのインストール	57
Xcodeのダウンロード	57
コマンドラインツールのインストール	58
回線環境をシミュレートする設定のインストール	60
JRE(Java Runtime Environment)のインストール	61
Homebrewのインストール	64
Yosemiteにインストール済みのソフトウェア	64
Homebrewとは	65
Homebrewのインストール	66
パスを通す?環境変数に追加する?	72
Homebrewのアンインストール	75
Homebrewによるツールのインストールと管理	76

CONTENTS

tree のインストールと実行	76
OS X のソフトウェアを Homebrew でインストール	77
公式リポジトリ以外からソフトウェアをインストール	80
インストール済みのソフトウェアのアップデート	82
覚えておきたい Homebrew のコマンド	86
Android SDK Tools のインストール	89
Android SDK Tools のダウンロード	89
Android SDK Tools のセットアップ	91
Android SDK Manager の起動	93
HAXM と Apache Ant のインストール	95
Android デバイスのセットアップ	96
Git を導入する	100
Git をインストールする	101
Git について	101
Git の仕組み	103
Git のインストール	104
Git を使う前に覚えておきたいこと	108
Git の初期設定?	108
Windows 環境との共存のために	110
GUI クライアントの Git を変更する	111
Git の基本操作を覚える	115
初めての Git リポジトリ作成	115
Git リポジトリに変更を加える	119
作業履歴をコミットしてみよう	122
プロジェクトごとの設定を作る	126
Git の管理対象をコントロールする	128
リモートの Git リポジトリをクローンする	131
リモートサーバに Git リポジトリを作成する	134
Git を使ってサイトを公開するには?	141
Git と連携してサイトを更新するいくつかの方法	141
git-ftp のインストール	143
git-ftp を使ったファイルの同期	143
SourceTree から git-ftp を実行する	145
タスクランナーから実行する	148
node.js の環境構築	150
node.js のインストール	151

CONTENTS

node.js のバージョンを管理する?	151
公式パッケージをアンインストールする	152
nodebrew のインストール	153
node.js のインストール	154
npm によるツールのインストールと管理	157
グローバルとローカル、インストール先の違い	157
npm を使ったパッケージのインストール	158
package.json ファイルについて	163
覚えておきたい npm コマンド	169
インストールしておきたいツール	174
Bower のインストールと使い方	174
Bower のコマンド	178
npm と Bower を使った自動化	180
パッケージのアップデートを少し便利に	183
Ruby の環境構築	185
Ruby のインストール	186
Ruby のバージョンを管理する?	186
rbenv のインストール	187
rbenv のプラグインの導入	190
Ruby のインストール	191
Gem によるツールのインストールと管理	195
RubyGems とは?	195
Sass、Compass のインストール	196
default-gems を使った自動インストール	197
覚えておきたい gem のコマンド	198
Bundler によるバージョン管理	201
Bundler のインストール	201
Bundler を使った Gems のインストール	201
Bundler でインストールされたツールの実行	203
フロントエンドツールの導入	205
いまどきの制作ツールとは	206
サイト制作を楽にするツールの導入	206
PaaS を使ったテストサイトの起動	206
Vagrant を使った環境構築	207
システム内に別の OS 環境を構築する	208
Vagrant のインストール	208

CONTENTS

仮想マシンを作ってみよう	208
Box ファイルを使ってみよう	208
オリジナルの仮想環境構築	209
Linux のディストリビューション	210
Ubuntu を使った LAMP 環境の構築	210
Node.js、Ruby、その他のインストール	210
Apache を使ったバーチャルホストの作り方	210
ローカルの Git リポジトリの作り方	210

いまどきのサイト制作とは

Web サイト制作、それもフロントエンド側にあたる作業はデバイスの多様化とともに複雑化しています。この数年で制作時に使うツールは、人によって組織によって、また担当する案件やプロジェクトの内容によっても変わってきていることでしょう。これまでのように Dreamweaver やテキストエディタだけがあればどうにかなる、という時代ではなくなってきたことを痛感している人も多いのではないのでしょうか。これからの Web 制作が少し楽になるよう、そして次の時流に乗り遅れないためにも比較的新しめなサイト制作環境を紹介していきます。

変わり続ける **Web** サイト制作

Web サイト制作を取り巻く環境がこの数年で大きく変わり始めています。「世界と日本では Web サイトが全然違う」とよく言われますが、それは単純な見た目だけの話ではありません。コンテンツ配信に対する考え方や作り方も次の時代を見据えてるように思えます。デバイスの多様化にあわせて Web サイト制作を取り巻くいろいろな環境が変わりつつあることを認識しておく方が良いでしょう。

閲覧環境が変わるとのこと

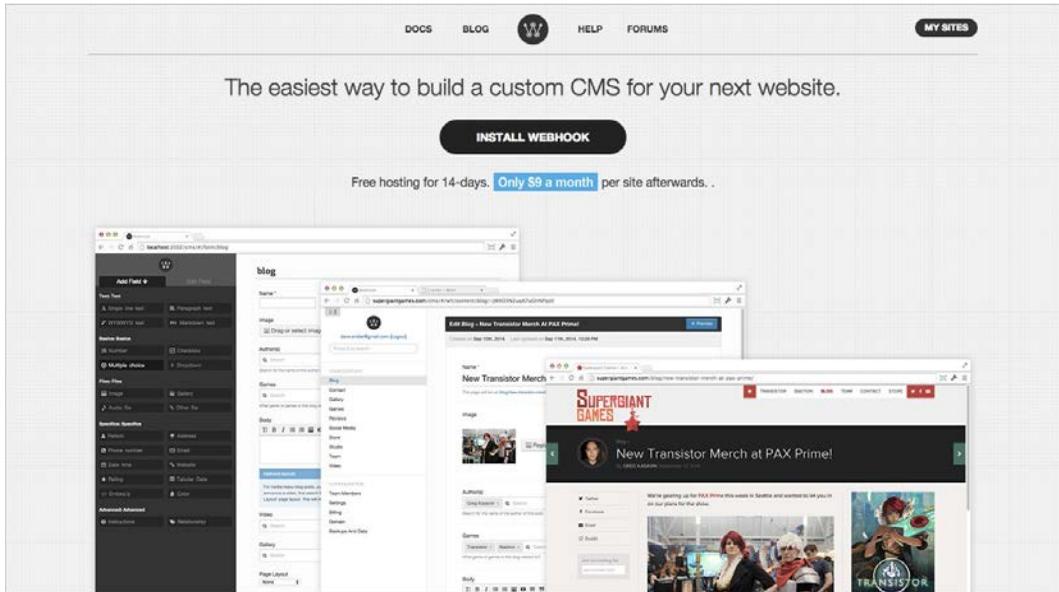
iPhone や Android デバイスの登場以来、これまで主にデスクトップ PC だけを対象に考えていればよかった時代は終わりました。ご存知のようにこの数年のスマートデバイスの普及は目を見張るものがあります。これまでデスクトップ PC だけを対象としてきた制作会社や個人であっても、その普及率にもう目を背けることはできません。これまでデスクトップ PC だけのことを考えていればよかったのに、一気にコンテンツの配信対象が増えたも同然です。

これまでは Dreamweaver やお気に入りのテキストエディタ、それにグラフィックソフトがあればどうにかかりました。しかし、さまざまな閲覧デバイスでコンテンツを使う人たちが増えています。回線環境はバラバラになり、画面サイズもバラバラ、そんな対象に向けていかにストレスなくコンテンツを配信するか。ビジネススピードも早くなっていて、これまでのようなやり方を続けていては到底太刀打ちできないばかりか、時間ばかりが無駄に過ぎていくことにもなりかねません。

コンテンツの作り方ですら多様化

日本においては、MovableType や WordPress などの CMS を使ったサイトの作り方、CSS による見た目の実装をどうすればいいか、jQuery を使った視覚的な表現としてのインタラクションの付け方など、主にフロント側で Web 制作に関わる人たちの興味の対象はまだまだそういったところに重きがあるようにも見えます(ニーズの問題でそれが悪いわけではありません)。

Web サイトのコンテンツが静的なものから動的な要素を含み、デバイスの多様化にあわせるかのように人々の行動までもが変化してくるようになってくると、閲覧する人のコンテンツに合わせた情報配信やリアルタイムコンテンツのニーズも出てくるでしょう。もう jQuery とそのプラグインを使って動きをつける程度の知識では無理があり、より深い JavaScript の知識が必要になっています。コンテンツを構成するのに必要な HTML や CSS ですら素の状態では書かなくなりつつあるのです。



Node.js で動作する Webhook

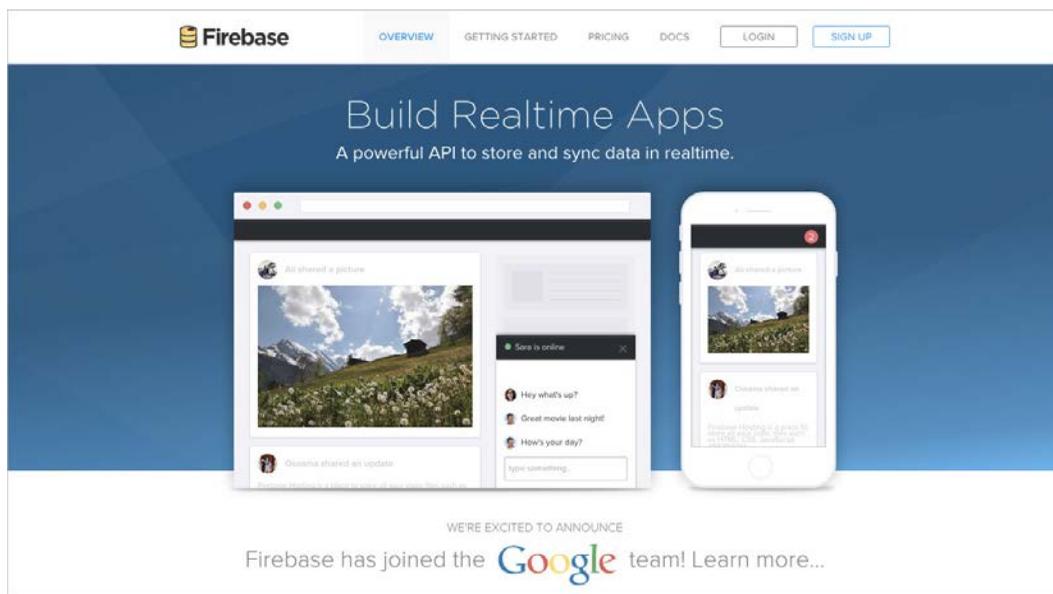


mean.io は、MEAN で作り始めるためのフレームワーク

Web が Web サイトでなくなる？

さらにこの数年で REST (Representational State Transfer) インターフェイスを介して、HTTP の URI ベースでデータをやりとりすることも増えているようです。従来の CMS であってもこの REST を介してデータをやりとりできるようになりつつあります。極端な話をすれば、CMS は完全にコンテンツ管理だけをおこなうだけでよく、URL の書き換えを含めたフロント側は JavaScript のフレームワークでコントロールすることもできます。

REST のような API (Application Programming Interface) によるデータのやりとりが理解できると、いざ自分が何か作る際もバックエンドのシステムを自分で用意する必要もありません。「BaaS (Backend as a Service)」を使えば、データベースや認証の仕組みなどもそれを利用すれば終わりです。ホスティングも含めていろいろな取り巻く環境が大きく変化していることを認識しておいた方が良いでしょう。



Firebase や Parse のような BaaS を使えばバックエンドはおまかせ

いまはまだこれまで通りのやり方で良いでしょうが、今後は主に Web サイトのフロント側を担当すると言ってもさまざまな環境下でのサイト制作をおこなう機会が増えるかもしれません。これまでのやり方を続けていくか、少し先を見ながらのんびりでも対応できる術を身につけるか、それは皆さん次第です。

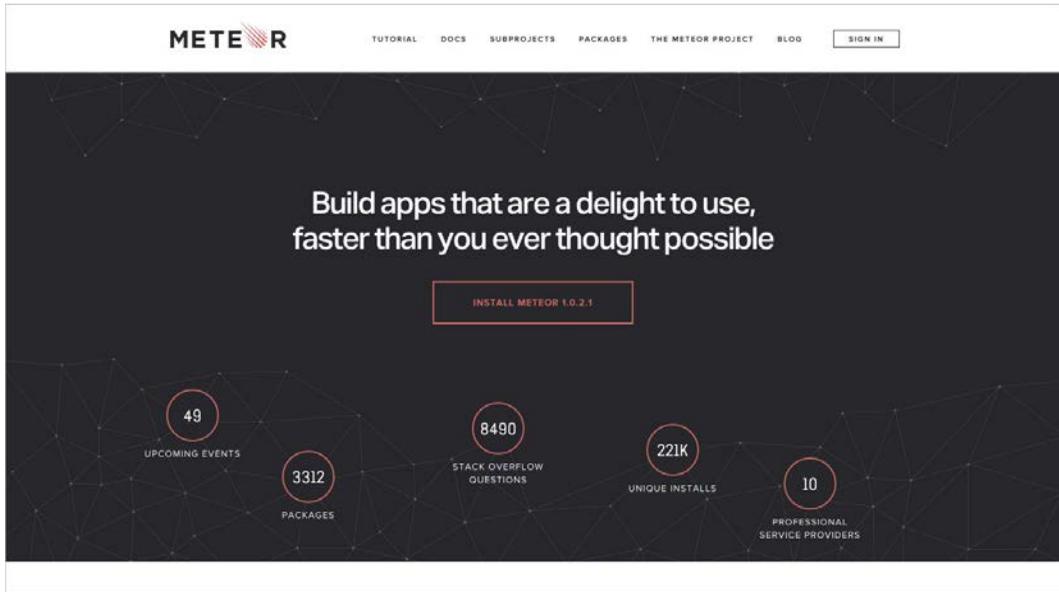
最新の制作ツールはコマンドラインから

いまどきの Web 制作ではフロント側の実装に関係することだけに絞っても、規模の大小にかかわらず実装に関わる負荷が増えています。やるが増えた分、コンピュータが得意なことは任せた方が楽になります。しかし、そういったツールはコマンドラインでしか使えないことが多いのです。

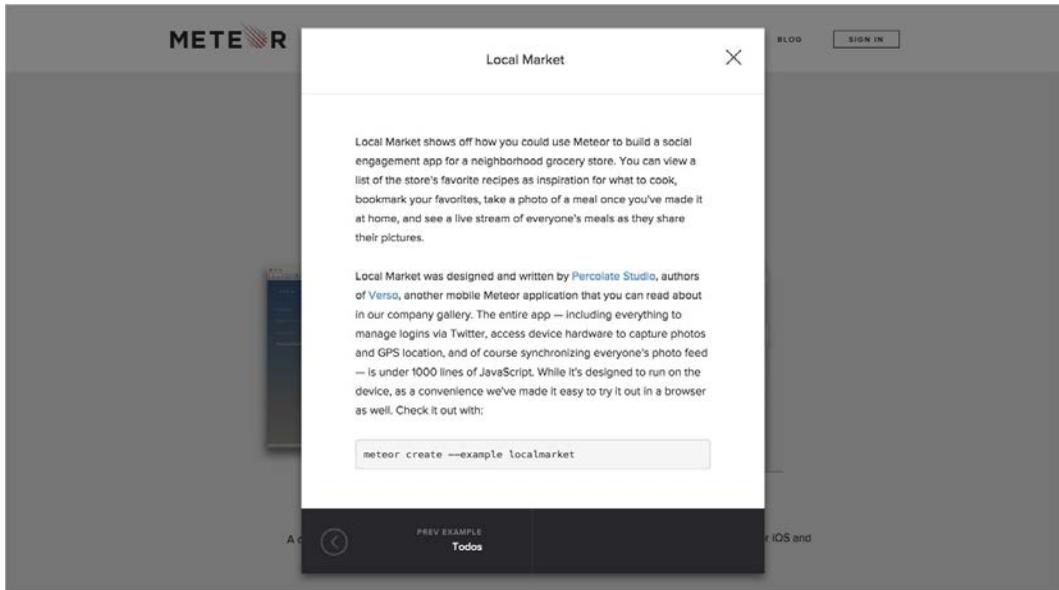
GUI だけではどうにもならない時代に

より直感的に操作できるようにと登場したはずの GUI ですが、今の時代それに逆行するかのように制作ツールに関しては CUI での動作が中心になっています。これまでは GUI の制作ツールの年に一回のアップデートでどうにかなったのかもしれませんが。新しい技術への対応などもそうですが、もはや日進月歩どころの進化ではなく常に新しいバージョンが提供されるような時間軸で動いています。知らなければいつまでもその便利さを享受できません。

まだまだ日本での注目度は低いようですが、「[Meteor](#)」はリアルタイムで動作する Web アプリケーションを簡単に作るためのフレームワークです。コマンド操作で必要な機能やパーツを追加して HTML や CSS、JavaScript を書いていけば、Web とネイティブで動くハイブリッドなアプリケーションをも作ることができます。「そんな便利なものがあるのならこれを使おう」と考えても、これを動作させるにはやはり Node.js の環境が必要となり、デモアプリはおろかサンプルのチュートリアルですら動かすことはできないのです。



リアルタイム Web のためのフレームワーク、Meteor



コマンド操作ができなければデモすら動かせない

ライブラリのダウンロードもコマンドライン

JavaScript のライブラリなどはこれまで.zip などファイル一式が提供されていることが多かったのですが、それをダウンロードしてきて任意の場所に保存すれば使うことができました。しかし、ここ最近ではコマンドラインでのインストール方法や GitHub のリポジトリへのリンクだけしか書いてないこともあります。

いざサイト制作に必要な道具一式を揃えて作り始めようにも、サイトを駆けずり回ってかき集める人とコマンドラインで一瞬で手元に用意できる人ではそのスタート地点からして違ってきています。たとえば、以下はフレームワークの Bootstrap を手元に用意するコマンドです。これを実行すればほんの数秒で Bootstrap だけでなく、その動作に必要な jQuery までも含めてダウンロードが終わります。

```
$ bower install bootstrap
```

次のコマンドは、WordPress の最新版のファイル一式をダウンロードできます。わざわざサイトに行く必要もないだけではなく、設定からデータベースの作成まであとひとつふたつのコマンドを打つだけで WordPress のサイトを立ち上げることができるのです。

```
$ wp core download --locale=ja
```

これだけでは一見ほんの些細なことのようにも思えます。しかし、コマンドで操作できるものは自動化の対象になります。使えるかどうかは仕事全体の生産性にも影響を与えるでしょう。ある人は3日かかってサイトを作る、しかしある人は1日もあればできるとなれば雲泥の差です。仮に個人事業主のような立場であれば時間単価が大きく変わってきます。

コマンド操作だけができれば解決

数年前から「深く使うことはなくても、せめて Node.js と Ruby ぐらいは入れておきましょう」と言い続けてきましたが、いまどきの Web 制作をよりスムーズにおこなうためにはもうコマンドラインの操作は必須です。今さら時代に逆行していると言われればそうですが、便利なツールが GUI では提供されることの方が少ないのですから仕方ありません。

CSS プリプロセッサやタスクランナーのようなものは GUI のソフトウェアから実行することができます。しかし、これらのツールのバージョンアップもまた異様なほどの速さです。CSS プリプロセッサの「Sass」の利用者が多いようですが、Sass はバージョンや変換エンジンに

よって使える機能が異なります。GUI のソフトウェアのバージョンに左右されてしまうと、いつまでも便利な機能が使えないかもしれません。

& SASSSCRIPT

Description

The ability to use `&`, the reference to the current selector, in SassScript. This basically means that `&` can be manipulated, inspected and updated manually.

Work-around

There is no known polyfill or work-around for this.

Tests and support SHOW DETAILS

	RUBY SASS 3.2	RUBY SASS 3.3	RUBY SASS 3.4	LIBSASS
SUPPORT	✗	✗	✓	✗

§ ANGLE CONVERSION

Description

Angles can be emitted in four different units:

- Degrees: `deg`
- Radians: `rad`
- Gradians: `grad`
- Turns: `turn`

Work-around

```
@function convert-angle($value, $unit) {
  $convertable-units: deg grad turn rad;
  $conversion-factors: 1 10grad/9deg 1turn/360deg 3.1415926rad/180deg;
  @if index($convertable-units, unit($value)) and index($convertable-
  @return $value
  / nth($conversion-factors, index($convertable-units, uni
  • nth($conversion-factors, index($convertable-units, $unit
```

Sass はバージョンやエンジンで使える機能が違う

今後何かを始めるにあたっては、コマンドラインからしかインストールできないソフトウェアを使わなければならないこともあるでしょう。フロントエンドとバックエンドの境界線があいまいになりつつある中、その基本操作と簡単な仕組みさえ覚えておけばいろいろなシーンで応用が効きます。「それ知らないです」「使っていません」では、仕事そのものがなくなるかもしれません。仕事はできる人の方に流れるものです。

ローカルで **Web** サイトを動かすには？

HTML と CSS、jQuery でのインタラクションが付与された静的なページであれば、ローカルで制作中の「～.html」のファイルを Web ブラウザで開けば動作しているかどうかを確認できます。しかし、動的に外部のリソースを取得するタイプのサイトを作るとなるとそうはいきません。時には「file://～」ではじまるアドレスでは表示確認ができないこともあるでしょう。当然、Ruby や PHP のようなプログラムが介在する Web サイトでも同様です。

OS X なら実は簡単!?

たとえば、CMS を使おうとすればそれを動作させるためのサーバ環境が必要です。OS X にはあらかじめ Web (HTTPD) サーバとして有名な「Apache」、プログラムの実行に必要な「PHP」「Ruby」などが含まれており、それらを有効化するだけで新たにテスト環境を用意せずに利用できます。しかし、それらを動かしたくても仕組みがわからないことにはできません。わからないからといって、簡単にローカルのテスト環境が用意できるオールインワンパッケージになったソフトウェアが良いかということそうでもないのです。

MAMP や **XAMMP** の問題

バックエンドの仕組みに詳しくない方は「**MAMP**」や「**XAMPP**」のように、あらかじめ Web サーバと Perl、PHP、MySQL などがひとつのパッケージとなったソフトウェアを使っているでしょう。必要なものがパッケージされたこれらは簡単に導入できて大変便利な反面、実際に動作するサーバの構成とは異なるがゆえの問題も出てきます。

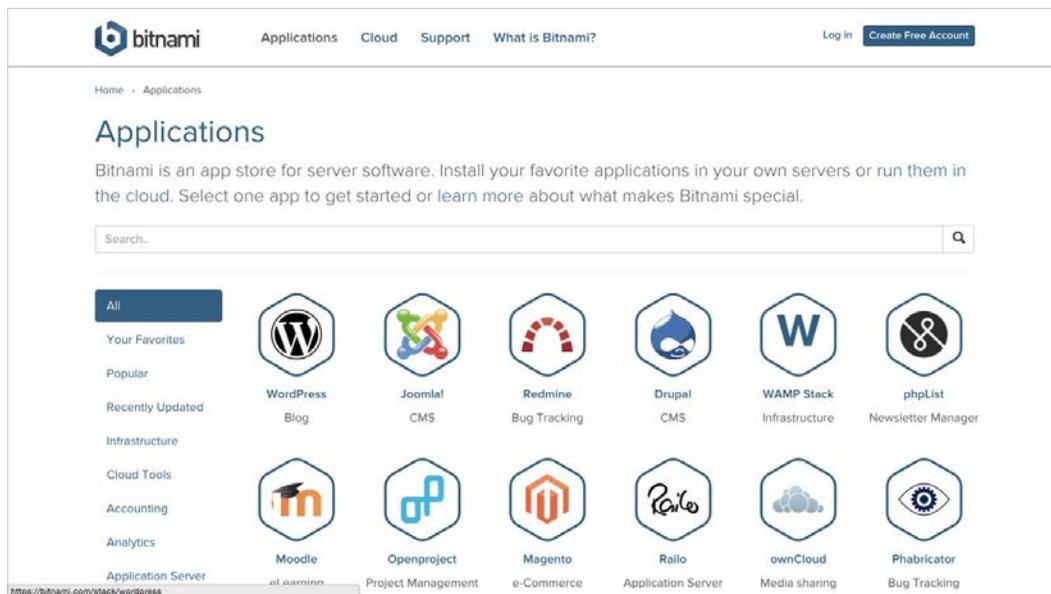
Web サーバにも PHP にも MySQL にもそれぞれ設定があります。エラーが出た場合、OS 側の問題なのかアプリケーションの問題なのか切り分けができなくなり問題の解決に時間がかかってしまう事にもなりかねません。MAMP は OS X Yosemite へのアップグレードによって正常に起動できないという問題が起きた方もいらっしゃるようですが、特殊な環境であることがむしろ困る場合もあるのです。

最近では Web サーバも「nginx(エンジンエックス)」の人气が高まっています。こういった Web サーバの特徴や仕様・設定方法を知らないままでは、ホスティングされたサイトの動作設定の変更はもちろんのこと、テスト環境をあわせて作ることもできません。組織の一員であれば誰かがやってくれるでしょうが、個人事業主などで受託をしている場合は困ります。HTML5 のアプリケーションなどバックエンドのシステムがローカルに不要なサイトは、JavaScript のツールを使ってローカルサーバをコマンドひとつ入力して起動する方が簡単です(いざとなったらそのまま外部公開もできます)。

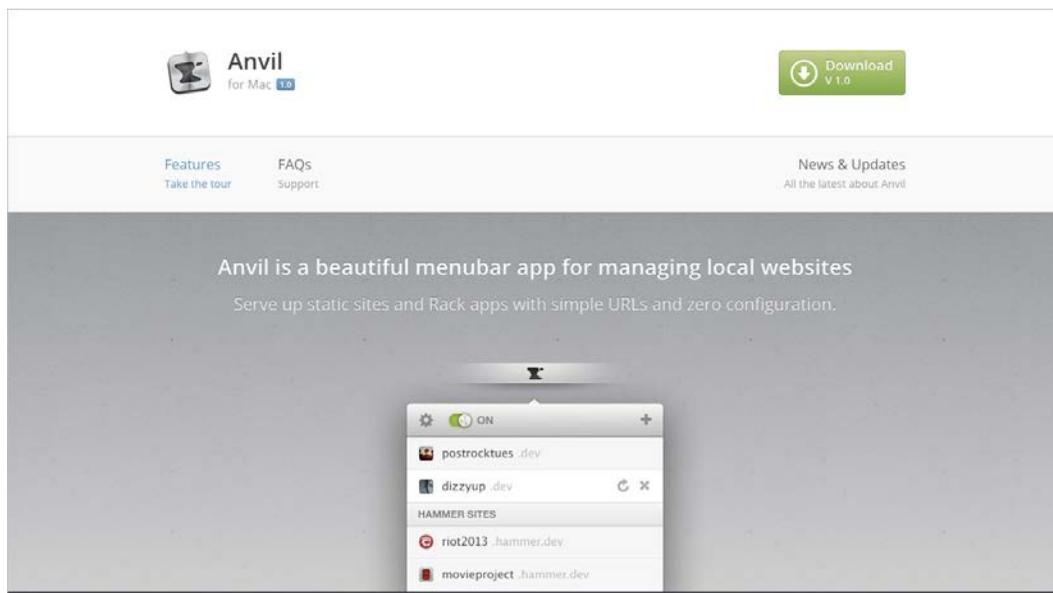
\$ ss

このたった 2 文字のコマンドを入力するだけでいいのです。

新しい技術に対応した環境を GUI で用意するなら「[Bitnami Stacks](#)」を使えば簡単です。静的なサイトの確認程度なら「[Anvil](#)」を使えば、ディレクトリがどこだろうがそこをルートディレクトリとしてローカルサーバを起動できます。



Bitnami であれば最新の環境も GUI で



任意のディレクトリをルートにするなら Anvil

新しい時代に対応するには

このように本書の Chapter 07 では、フロントエンド側の制作に役立つツールだけでなく、JavaScript を使ったローカルサーバの起動方法、ローカルマシンの制作サイトの状態を一時的に外部に公開する方法や PaaS (Platform as a Service) を使ったテスト環境の公開も紹介します。本書の後半では仮想環境をサーバの OS から選択して構築できる「Vagrant」というソフトウェアを使い、OS の中を極力汚さないで済むような開発・テスト環境を動かしてみましよう。

仮にきっちりと分業体制が敷かれていたとしても、制作するコンテンツによってはフロントエンドとバックエンドの境界線は曖昧になっています。そういった時流に逆らうことなく新たな潮流にも少しずつでも対応できるよう、いまどきのサイト制作をおこなうための環境を少しは理解しておいた方が良いでしょう。

ターミナルの操作に慣れよう

CUI(Character User Interface)/ CLI(Command Line Interface)は、テキストでコマンドの文字列を入力して実行します。一見難しそうに感じますが、日常的に使うコマンドは限られています。GUI の画面でボタンを押す代わりに、テキストで命令を与えてるだけだと考えれば決して怖いものではありません。いまどきの Web 制作のワークフローを効率化するには、この CUI でしか使えないソフトウェアが必要なのです。いまのうちからターミナル(Terminal.app)の操作に慣れておきましょう。

シェル？

ターミナルを起動すると OS X では「bash(バッシュ)」と呼ばれるシェルプログラムが起動します。このシェル(Shell)の基本的な仕事は、他のプログラムを起動することです。

主なシェル

UNIX 系の OS の標準のシェル(ログインしたときに起動されるシェル)は bash であることがほとんどのようです。実はシェルにはいくつかの種類があり、自分の好きなシェルを指定して使うことができます。それぞれのシェルは独自に備えた機能であったりコマンドの補完のような使い勝手が異なるため、エディタと同じように人によって使うシェルの好みがあります。

- sh(Bourne shell)
- bash(Bourne-Again shell)
- csh(C Shell)
- tcsh(TENEX C shell)
- zsh(Z Shell)

コマンドの説明は後ほどおこないますので、まずは Finder からターミナル(Terminal.app)を起動し、表示されてる「\$」のあとに下記のコマンドを入力してリターンキーを入力してみましょう。「cat」と「/etc/shells」の間には「(半角スペース)」を入れてください。

> cat コマンドを実行

```
$ cat /etc/shells (リターンキー)
```

```
Last login: Sun Jan  4 14:28:36 on tty??  
cipMBA:~ cipher$ cat /etc/shells █
```

ターミナルを起動してコマンドを入力

コマンドを実行すると以下の内容が表示されるでしょう。

> cat コマンドの実行結果

```
/bin/bash  
/bin/csh  
/bin/ksh  
/bin/sh  
/bin/tcsh  
/bin/zsh
```

コマンドを実行した結果として、ディスクの「/etc」の中にある「shells」ファイルの内容が画面上に出力されたのです。ここにリストされたシェルは、あらかじめ OS X にインストール済みのものです。シェルは「他のプログラムを起動すること」が仕事ですから、標準のシェルである bash からこれらを起動することもできます。bash から「zsh」と打ってリターンキーを押すと zsh が起動します。

> zsh を起動

```
$ zsh (リターンキー)
```

コマンドを実行すると、zsh が起動して以下のようにターミナルの表示が変わります。

> zsh のプロンプトの表示

```
マシン名%
```

簡単にいえば「bash が起動しているうえにさらに zsh が起動している」状態になったということです。

```
[zsh]
[bash]
```

続けて「ps」コマンドを実行すると、自分自身が起動しているプロセスが表示されて「zsh」が起動していることがわかります。

> ps コマンドの実行

```
% ps
```

> 起動中のプロセス

PID	TTY	TIME	CMD
9782	ttys000	0:00.01	-bash
9867	ttys000	0:00.02	zsh

起動した zsh を終了するには「control+d」のショートカットを入力します。ターミナルでプログラムを実行したり、リモートのサーバに SSH(Secure SHell)でログインしたりするようになると、いたるところで以下の 2 つのショートカットを使います。いまのうちに覚えておきましょう。

- **control+D**: ログアウト(シェルから抜け出す、リモートサーバからログアウトする)
- **control+C**: プログラムを終了する

では、ここからは OS X の標準シェルである「bash」を使って解説を進めます。

シェルの概念や機能について深く知りたい場合は、以下の記事をご覧ください。
(参考資料)[シェルの概念と機能](#)

覚えておきたいシェルのコマンド

ここからは実際にコマンドを入力・実行して、ターミナルの操作に慣れていきましょう。各コマンドを入力した後は「リターンキー」で実行します。

自分の居場所を表示する

まずは基本中の基本のコマンドを使って、自分の現在の居場所(現在の作業ディレクトリ)を表示してみましょう。

> 現在のディレクトリを表示

```
$ pwd
```

ターミナルの起動直後は自分のホームディレクトリにいますので、コマンドを実行するとターミナルの画面には自分のホームディレクトリのパス(ローカルマシン内での位置)が表示されます。

> pwd の実行結果

```
/Users/自分のアカウント名
```

「pwd」は「Print Working Directory」の略です。ターミナルで実行するコマンドの名前は、これからおこないたい処理内容の頭文字を取った略語がほとんどです。グラフィカル・ユーザー・インターフェイスの画面でいえば「pwd」と書かれたボタンを押してると考えてください。

ディレクトリの内容をリストする

自分がいるディレクトリにあるディレクトリやファイルをリストしてみましょう。現在の作業ディレクトリは、さきほど「pwd」で表示された自分のアカウントのホームディレクトリです。「ls(小文字のlとs)」と入力してください。

> ls コマンドの実行

```
$ ls
```

ルートディレクトリにあるディレクトリやファイルがリストされましたか？

> ls の実行結果

```
Applications  Downloads  Music
Dropbox       Pictures  Desktop
Public Library Documents  Movies
```

任意の場所をリストするには、以下のように ls の後に半角スペースをいれて場所(パス)を入力します。

> 場所を指定してリスト

```
$ ls 表示したい場所
```

たとえば、システムのルートディレクトリ「/」にあるものを見る場合はこうなります。

> ‘/’ の内容をリスト

```
$ ls /
```

OS X でルートディレクトリをリスト表示するとこのような感じになっているのではないでしょうか。

> ‘ls /’ の実行結果

Applications	Users	dev	private
Library	Volumes	etc	sbin
Network	bin	home	tmp
System	cores	net	usr
TSSleepHandlerHelp	opt	var	

日頃使っている GUI の Finder からは見えてない領域が多いことがわかります。このディレクトリ構成は、UNIX 系のマシンでは大体似たり寄ったりでおなじみのものです。この先リモートにある Linux サーバなどにログインする機会もあるでしょうから、これらのディレクトリがどのような役割になっているか簡単ですが紹介しておきます。

- /bin (基本コマンドが入っている)
- /etc (設定ファイルの多くは基本的にここに)
- /home (ユーザーのホーム。OS X では /Users なのでここは空)
- /sbin (システムの管理コマンドなど)
- /tmp (一時領域)
- /usr (各種コマンド、プログラムなどが入っている)
- /var (ログや Web サイトのデータなど、変更されるデータの格納領域)

コマンドのオプションを指定する

ターミナルから実行できる各種コマンドは、「-(ハイフン)」や「--(ハイフン ×2)」で始まるオプション指定が用意されているものがあります。以下のように半角スペースに続けて「-l(小文字の L)」を加えて「ls」コマンドを実行してみましょう。

> '-l' を付けて ls コマンドを実行

```
$ ls -l
```

先ほどのように単純に「ls」を実行しただけではディレクトリやファイル名が羅列されただけだったものが、この「ls -l」のように「-l」オプションを付けることでパーミッションや所有者とグループも一緒に画面内にリスト表示することができます。

「-a」オプションを付ければ、不可視ファイルも表示します。

> '-a' を付けて ls コマンドを実行

```
$ ls -a
```

オプションはまとめてすることもできます(ls -l -a でも可)。

> 複数のオプションを指定して実行

```
$ ls -la
```

冒頭で紹介した「ps」コマンドもオプションを付けて実行すれば、起動中のプロセスをさまざまな形で閲覧できます。次のコマンドを実行すると、システムが実行しているプロセスをはじめとして起動中のプロセスがすべて確認できるでしょう。

> プロセスの表示

```
$ ps -ax
```

どうでしょうか？GUIでPCを操作していると表向きには見えないだけで、実は裏側ではこのように多くのプロセスが起動して動いているのです。本書を読み進めていく過程では、コマンドラインを使って制作ツールを起動することもあるでしょう。動いてるのかわからない時などは、このようにして調べることができると覚えておきましょう。

コマンドの使い方やオプションが知りたいときは「man」コマンドを実行するか、コマンドによっては「-h」や「--help」「help」のオプションと一緒にコマンド実行すればヘルプ画面を表示することができます。

> ls コマンドのマニュアルを表示

```
$ man ls
```

マニュアルが表示されたら、「リターンキー」「矢印キーの上下」「Fキー(forward)」「Bキー(backward)」を使って表示位置を進んだり戻したりすることができます。変なキーを押してしまったら「escキー」を。マニュアル画面を終了するには「Qキー」を入力します。

作業ディレクトリを移動する

ターミナルでは現在自分がいるディレクトリを基準にして各種コマンドを実行します。現在の作業ディレクトリを変えてどこか別のディレクトリに移動するには、「cd」コマンドを実行します。言うまでもなく「Change Directory」の略です。

> デスクトップに移動する

```
$ cd ~/Desktop
```

上記コマンドでデスクトップに移動します。「~(チルダ)」は、自分のホームディレクトリ(\$HOME)を表しますので覚えておきましょう。「\$HOME」のように\$ではじまる文字列は、環境変数として登録されています。次のようにコマンドに組み合わせて使うことも可能です。

> 環境変数を使った移動

```
$ cd $HOME/Desktop
```

「cd」だけを実行する、または「cd ~」を実行すれば自分のホームディレクトリに戻ります。

> ‘~’ を付けてホームに移動

```
$ cd ~
```

一つ上の階層に移動したい場合は、「cd ..(ピリオド×2)」を入力しましょう。

> ひとつ上の階層に移動

```
$ cd ..
```

Web サイト制作で相対パスを指定するのと同じですね。このようにコマンド操作だけで場所を移動したり、ディレクトリの中身を確認したりするのは簡単なことです。シェルの扱いに慣れてくると、Finder を使って場所を移動しディレクトリを開くといった操作すらもどかしく感じるかもしれません。

ターミナルでは日本語の入力もできますが、移動などをよりスムーズにおこなうためにも英数字を使ったディレクトリ名にしておく方が良いでしょう。日本語の OS X の Finder 上で「デスクトップ」「書類」と表示される OS の標準ディレクトリは、それぞれ「Desktop」と「Documents」でアクセスできます。

入力補完を利用する

「`cd ~/Desktop`」のように入力する文字列が長くなればなるほど自分で入力すれば記述ミスが発生しますし、ディレクトリが深くなればなるほど(ファイル名が長くなればなるほど)入力に時間がかかってしまうでしょう。そこで時間がかかってしまうのであれば、GUIで操作した方が早いということになってしまいます。

コマンドの入力途中で「tab キー」を押せば、その先の内容を補完する機能がほとんどのシェルに用意されています。

> 「`cd ~/D`」を入力

```
$ cd ~/D (tab キー)
```

「`cd ~/D`」まで入力して tab キーを 2 回押せば、bash の場合は「`Desktop/ Documents/ Downloads/`」のようにルートディレクトリ以下にある「D」ではじまるディレクトリが表示されます。

> 入力内容を追加

```
$ cd ~/Des (tab キー)
```

デスクトップに移動したい場合は、その後続く「`es`」ぐらいまで入力して再度「tab キー」を入力するとその先の入力が補完されて自分で入力する必要はありません。デスクトップにあるディレクトリ名やファイル名がわからなくても、補完された後に再度 tab キーを 2 回押せばその次の候補が表示されます。

> ディレクトリ内をさらに調べる

```
$ cd ~/Desktop/ (tab キー)
```

大量にリストされた場合は「`:more`」が画面内に表示されます。この表示を終了するには「`Q` キー」を 1 度入力しましょう。ファイル名の冒頭の文字を数文字入れれば、それが含まれるものに絞り込まれるので、また tab キーを使って補完していけば入力は最低限で終わります。

Finder の「ディレクトリへ移動」メニューでも「tab キー」による補完が可能です。

ヒストリー（入力履歴）を利用する

ターミナルでの操作は、何度も何度も同じコマンドを入力することもあるでしょう。入力補完があるとはいえ、同じ内容を何度も何度も入力するほど面倒なことはありません。bash（やそれ以外のシェルも含む）は、一度実行した結果をヒストリーとして保存しています。

> ヒストリーの表示

\$ （矢印キーの上下）

上矢印のキーを押していけば、過去にそのシェルで入力したコマンドを遡ることができます。行き過ぎたら、下矢印キーを入力しましょう。このコマンドの実行履歴は、自分のホームディレクトリの直下に「.bash_history」という不可視ファイルで保存されます。

bash では矢印キーの上下以外での選択以外に、「control+R」キーを押してから任意の文字列を入力し、ヒストリーをインクリメンタルサーチすることもできます。

コマンド行のキャレットの移動

ターミナルでは、ここまで紹介したようにして任意のコマンドをテキストで入力します。実行するコマンドにはオプションだけではなく、ディレクトリ名、ファイル名などを付け加えるので、実行コマンドに打ち間違えなどがあると修正箇所まで戻るのが大変です。

たとえば、自分のホームディレクトリ以外で下記のコマンドを行末まで入力してから間違いに気付くことがあります。

\$ cd Desktop/folder-name/filename

正確には「~/Desktop/folder-name/filename」のようにチルダが必要です。ここで一文字ずつ入力内容を消去したり、「矢印キー(←→)」や「control+F」「control+B」で一文字ずつ移動するのは面倒です。ターミナル操作に慣れていたとしてもそんなことは多々ありますので、コマンド行の中のキャレットの位置を移動するショートカットを覚えておくとういでしょう。

bash では、「control+A(頭で覚える)」でコマンド行の先頭、「control+E(Endで覚える)」でコマンド行の最後に移動できます。ターミナルの環境設定で「メタキーとして Option キーを使用」のチェックボックスを入れておくと、「option+左矢印(または option+F)」で単語をひとつずつ飛ばしてキャレットを後ろに移動、「option+右矢印(option+B)」で逆に前方に単語をひとつずつ飛ばしてキャレットを移動できます。

入力しかけたコマンドを一旦キャンセルするには「control+C」を入力します。

> 入力をキャンセルする

```
$ cd Desktop/folder-nam (control + C キーを実行)
```

これで入力した内容が消去されます。間違えてるからと「delete」キーなどで消すより簡単です。

Finder でディレクトリを開く

任意のディレクトリを OS X の Finder 上に開くには「open」コマンドが使えます。「.(ピリオド)」は現在のディレクトリを表しますので、下記のコマンドを実行すると Finder で現在の作業ディレクトリが開きます。

> 現在のディレクトリを Finder で表示

```
$ open .
```

OS X には Finder からは直接開けないディレクトリもあります。開きたい場合は、Finder の「ディレクトリへ移動」メニューを使わなくてもターミナルから直接開くことができます。

> 不可視のディレクトリを表示する

```
$ open ~/Library/Application\ Support/
```

ディレクトリ名やファイル名に半角スペースが含まれるものは「\ 」という風に「バックslash+半角スペース」の入力が必要です。これもタブキーによる入力補完を利用すれば入力する必要はありませんね。

ファイルの内容を表示する（テキストファイル）

テキストファイルの中身などを確認したい場合は、「cat」や「more」「less」といったコマンドが使えます。

「cat」コマンドでファイル名を指定すれば、そのファイルの内容がすべて画面内に出力されます。「cat」コマンドの本来の用途は複数ファイルを繋げる(concatenate)コマンドですが、単一のファイルの全体をパッと見たいといった用途にも使えます。上記コマンドを実行すれば、画面内にファイル内容のすべてが出力されて終了します。長いテキストファイルが出力されたとしてもターミナルの画面をマウスカーソルで遡るなどして内容は確認できるでしょう。

> cat コマンドでファイルを表示

```
$ cat ファイル名
```

長いテキストファイルの内容を順追って見たい時は、「more」または「less」コマンドが便利です。

> more コマンドでファイルを表示

```
$ more ファイル名
```

> less コマンドでファイルを表示

```
$ less ファイル名
```

いずれもファイルを開いて画面をスクロールさせながら内容を確認できます。「more」や「less」は、「/」や「?」を入力してから単語を入力し「return」キーを押して検索したり、ハイライトするといった使い方もできます（more と less で挙動は違います）。これらのコマンドはそのままでは起動したままになるので、終了するには「Q キー」を入力しましょう。

ファイルの拡張子によっては「open」コマンドを使って直接 GUI のアプリケーションで開くこともできます。

> ファイルを任意のアプリケーションで開く

```
$ open ファイル名
```

ログファイルなど長いファイルの最後の方だけ見たい場合は「tail」を使うと便利です。

> tail コマンドでファイルを表示

```
$ tail ファイル名
```

「tail」コマンドはファイルの最後の行から 1 画面分を画面上に出力します。

画面の内容をクリアする

コマンド操作を続けていくと、ターミナルの画面内にはどんどん文字列が表示されていきます。現在表示されている画面を綺麗にしたい場合は「clear」コマンドを使うとスクリーンが一画面分スクロールして真っ新の画面になります。

> clear コマンドを実行

```
$ clear
```

「clear」コマンドにはショートカットが割り当てられているので「control + L」キーを実行しても同様の結果が得られます。

スクロールしているだけなので、画面をスクロールすれば直前までの画面を遡ることは可能です。

新規テキストファイルを作成する

空の新規ファイルを作るには「touch」コマンドを使います。本来の用途は既存のファイルのタイムスタンプを変更するものですが、ファイルが存在しない場合は新規の空ファイルが作成されます。

> 'sample.txt' ファイルを作成

```
$ touch sample.txt
```

コマンドを実行すると現在の作業ディレクトリに、真っさらの「sample.txt」というテキストファイルができるでしょう。

新規ディレクトリを作成する

新規でディレクトリを作成する場合は「mkdir」コマンドを使います。言うまでもなく「MaKe DIRectories」の略です。

> 新規ディレクトリの作成

```
$ mkdir ディレクトリ名
```

任意のディレクトリ名を付けて実行すれば、現在の作業ディレクトリに新しいディレクトリが作成されます。場所を指定してディレクトリを作りたい場合は、そこまでのパスを入力してディレクトリ名を指定します。

> 場所を指定してディレクトリを作成

```
$ mkdir ~/Desktop/ディレクトリ名
```

「mkdir」コマンドでは、半角スペースを空けて作りたいディレクトリ名を列挙すれば複数のディレクトリを一度に作成可能です。

> 複数のディレクトリの作成

```
$ mkdir ディレクトリ A ディレクトリ B
```

「-p」オプションを付け加えれば、階層構造をもったディレクトリも直接作れて便利です。

> 階層付きでディレクトリを作成

```
$ mkdir -p 親ディレクトリ名/子ディレクトリ名
```

このような操作を覚えておけば、以下のように一回のコマンドで「projects」ディレクトリ内に「images」「css」「js」のディレクトリをあらかじめ用意するといったことができますね。

> 階層付きのディレクトリを複数作成

```
$ mkdir -p projects/images projects/css projects/js
```

ファイルやディレクトリを移動する（リネームする）

ファイルやディレクトリを移動する場合は、「mv」コマンドを使います。言わずもがな「MoVe files」です。移動する対象と移動する場所を対で指定します。

> mv コマンドを実行

```
$ mv ファイル名 移動する場所
```

ファイルをデスクトップに移動する場合は以下ようになります。移動先がディレクトリであれば特にファイル名を指定する必要はありません。

> デスクトップにファイルを移動

```
$ mv ファイル名 ~/Desktop/
```

ファイル名も指定したい場合は、移動先でのファイル名を指定しましょう。

> ファイル名を指定して移動

```
$ mv ファイル名 ~/Desktop/新しいファイル名
```

「mv」コマンドは、ファイルの移動だけでなくファイル名を変更することもできます。同一階層で「mv」コマンドを実行すれば、単純にファイル名がリネームされます。以下の例は、「a.txt」が「b.txt」に変更されます。

> ファイル名の変更

```
$ mv a.txt b.txt
```

ファイルと同様にディレクトリを移動する場合も「mv」コマンドを使います。

> ディレクトリを移動

```
$ mv ディレクトリ名 移動する場所
```

簡単ですね。ファイル同様、ディレクトリ名前を変えたい場合も「mv」コマンドを使いますので覚えておきましょう。

ファイルやディレクトリをコピーする

ファイルをどこか別の場所にコピーしたり、別名で保存しておく場合は「cp」コマンドを使います。言うまでもなく…、です。

> ファイルをコピーする

```
$ cp ファイル名 コピーする場所
```

「mv」コマンド同様に移動先でのファイル名を指定することもできます。

```
$ cp ファイル名 ~/Desktop/新しいファイル名
```

同じ階層で「cp」コマンドを実行すれば、同じものが別名で複製されます。テキストファイルを編集する際は、あらかじめファイルを別名で複製しておいて作業するのが安全ですね。

> ファイルの複製

```
$ cp ファイル名 新しいファイル名
```

「cp」コマンドはファイルだけでなくディレクトリのコピーも可能ですが、ディレクトリの場合には「cp」コマンドをそのまま実行してもエラーになります。ディレクトリの場合には「-R(-r)」オプションが必要です。

> ディレクトリのコピー

```
$ cp -R ディレクトリ名 移動先/ディレクトリ名
```

OS X にはファイルやディレクトリのコピーする「ditto」という別のコマンドが用意されています。「ditto」コマンドは、ディレクトリをコピーする時もオプション指定が不要ですのでこちらの方が簡単ですね。

> ditto コマンドによるコピー

```
$ ditto ディレクトリ名 移動先/ディレクトリ名
```

ファイルやディレクトリを消去する

ファイルやディレクトリを削除する場合は「rm」コマンドを使います。「ReMove」で覚えると良いでしょう。

「rm」コマンドを実行すると Finder のゴミ箱に入るわけではありません。実行すれば問答無用で、何事もなかったかのようにファイルやディレクトリは消え去りますので注意が必要です。

> rm コマンドでファイルを消去

```
$ rm ファイル名
```

中身のあるディレクトリの場合は「-R(-r)」オプションを付けて実行しましょう。

> ディレクトリを消去

```
$ rm -R ディレクトリ名
```

いきなり消されるのが怖い場合は、「-i」オプションを付けて実行することで確認のダイアログを出すことができます。

> '-i' オプション付きで rm コマンドを実行

```
$ rm -i ファイル名
```

コマンドを実行すると「Remove ファイル名? 」と聞かれますので、消去する場合は「Y」キー、キャンセルする場合は「N」キーを入力します。うっかり「return」キーを押しても大丈夫です。キャンセルされます。

> 'rm -i' による確認

```
Remove ファイル名? (y か n を入力)
```

rm コマンドの利用は慣れるまで慎重に実行しましょう。

「rm」を使うといきなり消されてしまうので、それを予防する「trash」のようなプログラムもあります(標準ではインストールされません)。こういった別のコマンドを別途インストールすることでファイルやディレクトリを一旦ゴミ箱に移動することができます。

「rm」コマンドを必ず「-i」オプション付きで実行するためには、コマンドにいちいちオプションを付けずシェルの alias 機能を使ってあらかじめコマンドを登録しておくこともできます。alias とは任意のコマンドを別名で置き換えることができるもので、ホームディレクトリ直下にあるシェルの設定ファイル「.profile」「.bash_profile」「.bashrc」などに記述します。シェルの設定ファイルや alias については後の章で解説します。

複数のファイルやディレクトリをまとめて操作する

複数のファイルやディレクトリをまとめて移動したり消去したい場合は、ファイル名やディレクトリ名の代わりに「* (アスタリスク) 」を使ってワイルドカード指定をします。Web 制作の作業中は、任意の拡張子が含まれているものだけを移動したり消したいことは良くあることでしょう。

たとえばこのように「*.php」とすれば、themes ディレクトリ内の拡張子「.php」を持つファイルがすべて消去されます。

> ワイルドカードを使ったコマンドの実行

```
$ rm ./themes/*.php
```

以下のコマンドを実行すれば「images」ディレクトリ内の「common-」で始まるファイルがすべて、「images/common/」ディレクトリに移動します。

> ワイルドカードを使ったファイルの移動

```
$ mv ./images/common-* ./images/common/
```

ワイルドカードで移動や消去、コピーする対象を指定する場合で中にディレクトリが含まれる時は「-R」や「-f」オプションが必要です。「-f」オプションは「強制的に」という意味ですね。

複数のファイルを結合する

最初の方で紹介した「cat」コマンドを使えば、複数のファイルを結合してひとつのファイルにすることができます。

> ファイルの結合

```
$ cat a.txt b.txt > concat.txt
```

このように結合したい複数のファイル名を「>」で渡してあげることによって、この場合は「a.txt」と「b.txt」の内容が結合されて新しく「concat.txt」が生成されます。この「>」を「**リダイレクト**」と呼んでいます。ファイルが既に存在している場合は、内容が消去され（上書きされ）「a.txt」「b.txt」の内容が変わるので注意が必要です。

ファイルが既に存在していて、そこに新たに「c.txt」などの内容を追加したい場合は「>>」を使います。

> 既存のファイルに追加

```
$ cat c.txt >> cancat.txt
```

「cat」コマンドを利用すれば、「JavaScript のライブラリなどを使用する際にライセンステキストを付与する」といった作業もわざわざファイルを開いてコピー&ペーストすることなくコマンドから一発で付け足すことができます。

ファイルやディレクトリの所有者やパーミッションを変更する

UNIX 系の OS では、ファイルやディレクトリには所有者とグループが設定されています。OS X の場合はシステムが利用するものは、所有者が「root」でグループが「wheel(または admin)」になっています。以下のコマンドを実行するとリストの中の項目の所有者とグループが「root:wheel」になっているでしょう。

> '/usr/bin' をリスト

```
$ ls -l /usr/bin
```

一方、デスクトップや自分が作成したディレクトリなどをリストすると「自分のアカウント名:staff」になっているはずです。

> デスクトップをリスト

```
$ ls -l ~/Desktop
```

このように UNIX 系の OS は複数のアカウントで使うことを前提としているため、所有者と所属するグループが厳密に分けられて権限が設定されています。

Web サイト制作の現場では、CMS を設置する時などリモートのサーバにファイルをアップロードして実行ファイルに実行権限を与えたり、ディレクトリのパーミッションを変更することがよくあります。もちろんそれはローカルの OS X での作業中でもたびたび発生することでしょう。そんな時は「chmod(CHange MODE)」や「chown(CHange OWNer)」「chgrp(CHange GRoup)」のコマンドを使います。

ファイルやディレクトリのパーミッションを変更するには「chmod」コマンドを使います。

> ファイルのパーミッションの変更

```
$ chmod 755 example.cgi
```

このように「chmod」コマンドの後にファイルやディレクトリのパーミッションを指定してファイル名やディレクトリ名を指定します。もちろんワイルドカード指定もできますので、任意の拡張子を持つファイルやディレクトリ丸ごとパーミッションを変えるといったことも簡単です。

> ワイルドカードによる変更

```
$ chmod 644 *.php
```

> ワイルドカードによる変更

```
$ chmod 777 ./htdocs/uploads/*
```

任意のファイルやディレクトリの所有者とグループを変更する場合は、それぞれ「chown」コマンド、「chgrp」コマンドを使います。一度に所有者とグループを変えたい場合は、以下のように「所有者: グループ名」と所有者とグループ名を「:(コロン)」で区切って一緒に適用すると簡単です(-R は繰り返し処理オプション)。

※PDF 版では「所有者」と「グループ名」を区切る「:」の後に半角スペースが入ってるように見えますが、この位置の半角スペースは不要です。

> 所有者とグループを一度に変更

```
$ chown -R 所有者: グループ名 ファイル名やディレクトリ名
```

いずれのコマンドにしても所有者が「root」である場合は、実行してもパーミッションエラーになります。その場合は「sudo」コマンドと併せて以下のように実行します(sudo コマンドについては後述)。

> 'sudo' 付きでコマンドを実行

```
$ sudo chown -R 所有者名: グループ名 ファイル名やディレクトリ名
```

パスワードが求められたら、自身の OS X パスワードを入力します。

もちろん「*」を使って任意のディレクトリ以下のファイルをすべて変更することも可能です。

```
$ sudo chown -R 所有者名: グループ名 /usr/local/bin/*
```

Web 制作系のツールではインストール方法によって、ルートユーザーの権限が必要な状態でインストールされるものがあるので覚えておきましょう。時にはパーミッションの変更を促される場合があります。

管理者としてコマンドを実行する

OS X ではシステムまわりのコマンドを実行したり、管理者 (root) でないと開けないファイルがあります。そのようなディレクトリやファイルを扱う場合は、「sudo」をコマンドの先頭に付けて実行します。「sudo」で実行する場合は、管理者パスワードの入力を求められますので、自身の OS X パスワードを入力します。

> 'sudo' 付きでコマンドを実行

```
$ sudo コマンド
```

特に最近の Web 制作では「npm」や「gem」などを使う作業も増えています。インストール方法によってはシステム内に直接コマンドがインストールされてしまい、コマンドの実行に「sudo」を付けている人も多いのではないのでしょうか (sudo なしで実行するインストール方法については 5 章以降で解説します)。指定されたコマンドを実行してもパーミッションエラーが出るようであれば管理者権限が必要なんだと考えて、再度「sudo」を付けて実行してみましょう。

sudo コマンドは一度パスワードを入力すると初期設定された時間だけパスワード入力が省略されますが、一定時間が経過したあとは再度パスワード入力が求められます。管理者権限が必要な作業をしばらく続ける場合など、いちいちパスワードを入力するのがわずらわしく感じるでしょう。そのような場合は sudo に「-s」を付けて、一時的に root ユーザーになってしまうこともできます。

> 一時的に root ユーザーに変更

```
$ sudo -s
```

コマンドを実行すれば、以後 root ユーザーとしてすべてのコマンドが実行可能です (あまりお薦めはしませんが)。作業が終わったら「control + D」キーを押して root ユーザーからログアウトしましょう。

root ユーザーになるコマンドは他にもいくつかありますが、一時的に変わるだけなら上記の「`sudo -s`」を覚えておけば良いでしょう。一般的には `sudo` コマンドが実行できるユーザーは限定されます。

覚えておくと便利なコマンド

日常の作業では使う頻度は少ないかもしれませんが、覚えておくと便利なコマンドもあります。

コマンドの場所を確認する

ターミナルで作業をしたり、CLI で実行されるツールなどを使うようになると、実際のコマンドがどこにあるかを調べたいときも出てきます。そのような場合は「whereis」「which」のコマンドが使えます。たとえば、OS X には標準で PHP が入っていますが、コマンドの場所を確認するには以下のように「whereis」または「which」にコマンド名を付けて実行します。

> whereis コマンドの実行

```
$ whereis php
```

> which コマンドの実行

```
$ which php
```

コマンドを実行すれば場所を教えてください。

> 実行結果

```
/usr/bin/php
```

既に Homebrew やその他のツールで PHP が別にインストール済みで、それを使っている場合は「/usr/local/bin/php」などになるでしょう。

ファイルやディレクトリを圧縮（アーカイブ）・解凍する

手元のディレクトリをまるごと zip 圧縮したり、アーカイブとして tar.gz といった形式でまとめるには、OS の圧縮プログラムを利用しましょう。リモートのサーバにディレクトリの中身をちまちまと FTP でアップロードしたり、また逆にちまちまとダウンロードするようなことはできれば避けたいものです（予期せぬ事態でネットワークが切断されてやり直しになるかもしれません）。

手元のファイルを zip 圧縮するには「zip」コマンドを使います。

> zip でファイルを圧縮する

```
$ zip 保存するファイル名.zip 圧縮するファイル
```

ここでは保存するファイル名が先になることに注意してください。ディレクトリを丸ごと圧縮する場合は「-r」オプションを付けます。「-r」オプションを付けないと空の zip ファイルの完成です。

> ディレクトリを zip 圧縮する

```
$ zip -r 保存するファイル名.zip 保存するディレクトリ
```

zip コマンドは多様なオプションを備えています。「-1」～「-9」までで圧縮率を変えたり、「-f」オプションで変更したファイルだけ圧縮するようなことも可能です。

では、今度は zip を解凍してみましょう。

> zip ファイルを解凍する

```
$ unzip 解凍したいファイル
```

ディレクトリに入っているものはそのまま解凍されます。オプションで「-Z」を付けると、.zip の中身を表示することができます。

> zip ファイルの中身を確認する

```
$ unzip -Z 中身をみたい zip ファイル
```

インターネット上で公開されているライブラリやフレームワークなどは「tar.gz」形式（tar 形式のアーカイブ+gunzip）でまとめられているものがあります。これらのファイルもコマンドラインから元のファイルに戻すことが可能です。

> tar.gz 形式を解凍する

```
$ tar -xvzf 解凍したいファイル.tar.gz
```

「tar」コマンドは、tar 形式のファイル进行操作するものです。オプションの指定はいろいろありますが、よくオプションとして使われる「-xvzf」は「eXtract/Verbose/gunZip/Filename」の意味があります。「解凍する/冗長に/gunzip 形式も/次のファイル名を」といったところでしようか。

逆に tar.gz 形式でアーカイブ+gunzip 圧縮したい場合は、「x」を「c(Create)」に変えるだけです。

> tar.gz 形式で圧縮する

```
$ tar -cvzf 圧縮するファイル名.tar.gz 圧縮する対象のファイルやディレクトリ
```

これで対象となるファイルやディレクトリなどが、tar.gz 形式でひとかたまりのファイルになります。ここでは「-xvzf」「-cvzf」とハイフン付きにしましたが、「xvzf」「cvzf」でも大丈夫です。Gunzip による圧縮をせず、tar 形式のアーカイブだけ作りたい場合は「z」オプションを抜きましょう。

> tar 形式のアーカイブの作成

```
$ tar cvf 圧縮するファイル名.tar 圧縮する対象のファイルやディレクトリ
```

このように手元のファイルやリモートのファイルは、インターネット経由で FTP などでのダウンロード/アップロードの前に一度丸ごとまとめてから作業する方がスムーズです。

リモートサーバへの SSH によるログインが可能なら、ローカルやリモートで圧縮・解凍操作をすることにおけば、FTP でバラバラと大量のディレクトリやファイルを扱うことはないでしょう。

SSH の鍵の作成

SSH でリモートのサーバにログインしたり GitHub などの Git リポジトリにアクセスするなど、従来の FTP に変わって SSH と(その鍵)を使ってサーバとアクセスすることが増えています。環境によってはパスワード認証は許可されず、SSH の鍵認証などでのログインしか許可されないこともあります。SSH 鍵をローカルのマシンで生成するには、以下のコマンドを実行します(オプションで鍵の種別を指定することもできます)。

> SSH の鍵を作成

```
$ ssh-keygen
```

コマンドをそのまま実行すると下記のメッセージが表示されます。「id_rsa」はデフォルトの鍵の名前です。マシンをアップデートしている場合などは既に作っていることも多いでしょうから、新規で作る場合は重複しないようにどこで使う鍵なのかわかりやすい名前を付けて鍵を作成しましょう。

> 生成中の画面

```
Generating public/private rsa key pair.
```

```
Enter file in which to save the key (/Users/username/.ssh/id_rsa): (ここに鍵の保存場所を記述)
```

任意で鍵の名前を決めてリターンを入力します(この場合は、keyname)。

> 鍵の保存場所と名前を入力

```
Enter file in which to save the key (/Users/username/.ssh/id_rsa): /Users/username/.ssh/keyname (リターン)
```

鍵にアクセスするためのパスフレーズの入力が求められますので入力します。パスフレーズを忘れると開くことができなくなるので、頭の中をしっかり覚えておくかどこかにこっそりメモしておきましょう。

これで生成される「keyname」というのが秘密鍵、「keyname.pub」と'.pub' が付いている方が公開鍵です。サードパーティのサービスなどに登録する時は公開鍵を登録します。必要になったらテキストエディタなどで開いて内容をコピーするか、後ほど紹介する「pbcopy」コマンドでコピーして登録しましょう。

SSH は「~/ssh/config」でホスト毎の設定が可能です。

(参考資料)[ssh-keygen\(1\) Mac OS X Manual Page](#)

制作環境構築の下準備

実際の制作環境を作り始める前に下準備をしておきましょう。コマンドラインを使ったツールが増えているいまどきの Web 制作では、それらをすんなりと扱うためにもあらかじめ入れておいた方がよいソフトウェアがあります。本章で紹介する各種ツールはなくてもどうにかはありますが、いずれどこかでこれらのお世話になることになるかもしれません(ないと始まらないこともあります)。ここで前もってインストールだけ済ませておきましょう。

Xcode とコマンドラインツールのインストール

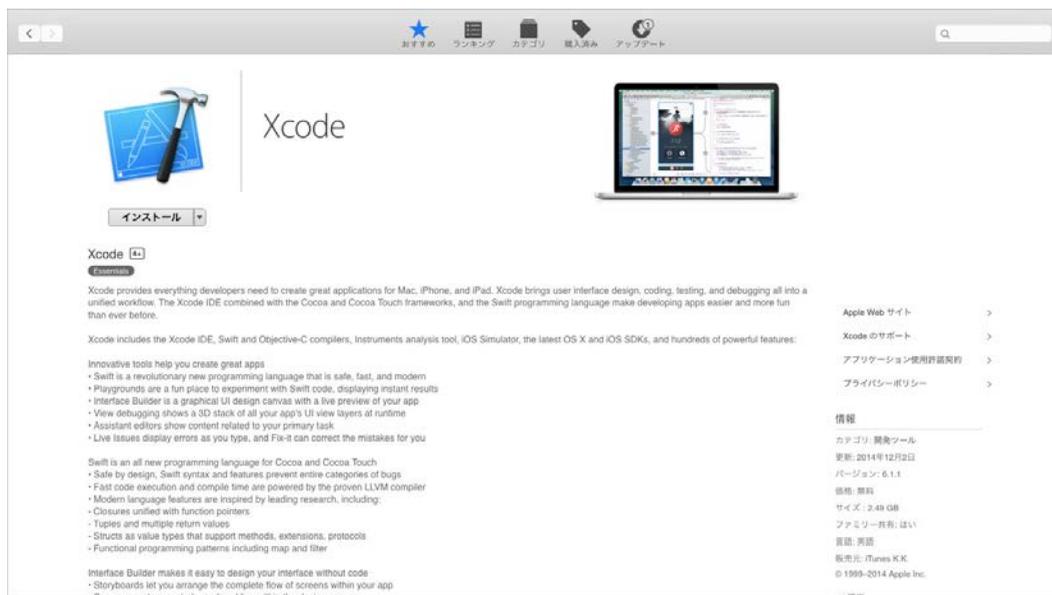
Xcodeは、OS X や iOS のアプリケーションを作成するための統合開発環境です。Web 制作の現場では iOS シミュレータなどで既にお世話になっている方もいらっしゃるかもしれませんが、最新の Web 制作ツールを利用するにあたって Xcode 本体は必要ありませんが、Apple の Developer サイトで公開されている「[Command Line Tools for Xcode](#)」だけはインストールしておきましょう。

注意

本ページ以降、ソースコードやコマンドの実行文が多く含まれます。「\ (バックスラッシュ)」は、通常記述できない文字を表記する際に用いられます(参考: [バックスラッシュ](#))。本文のコードやコマンド中でバックスラッシュが含まれている場合は、その直後の半角スペースなどをエスケープしているか、折り返しの含まれる長い行のコマンドをその部分で改行していることを表します。本書は Leanpub の自動変換処理のため、電子書籍のフォーマットによって任意の場所でエスケープされています。

Xcode のダウンロード

Xcode は、[App Store](#) からダウンロードしてインストールすることができます。前述のように本体そのものは iOS アプリなどの開発をしない限りは必要ではありません。Web とネイティブで動作するハイブリッドなアプリを開発する時がきたら、「[Cordova](#)」のようなツールでコンパイル・シミュレートをする時には必要なので入れておくと良いでしょう(ディスク容量が切迫している方は、次の Command Line Tools だけでも構いません)。



Xcode は App Store からダウンロード

Xcode がインストールできているかを確認する場合は、以下のコマンドをターミナルから実行します。

> Xcode の有無を確認

```
$ xcode-select -p
```

既にインストールされている場合は、以下のようにインストール先が表示されます。

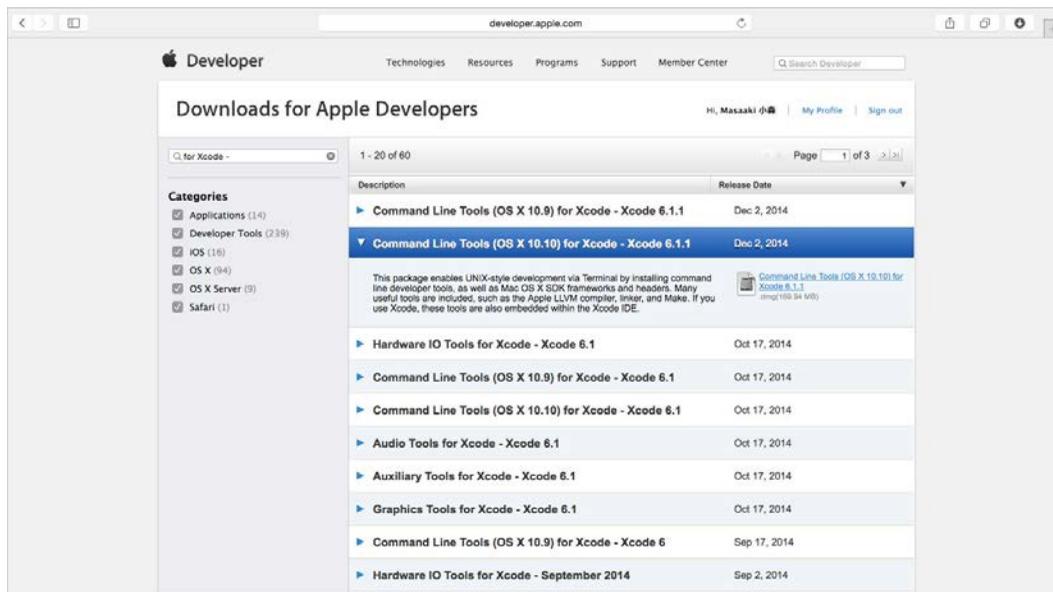
> コマンドの実行結果

```
$ /Applications/Xcode.app/Contents/Developer
```

コマンドラインツールのインストール

Apple が提供するソフトウェアのコンパイルに必要なコマンドラインツール一式は、「[Command Line Tools for Xcode](#)」として、Xcode とは別に配布されています。このツール群は、Apple の Developer サイトからダウンロードするか(要無償の会員登録)、ターミナルからインストールすることもできます。

Command Line Tools は OS X のバージョン毎に分かれて配布されています。Web サイトからダウンロードする際は、使用中の OS のバージョンにあわせてダウンロードしてください。このファイルは Xcode のアップデートの時などにあわせてバージョンアップされますので、たまには最新版がないか覗いてみると良いでしょう。ダウンロードが終わったらインストールしておきます。



Command Line Tools は OS のバージョン別に配布



インストーラを起動して画面の指示に従ってインストール

既に Command Line Tools がインストール済みかはどうかわからない場合は、以下のコマンドを実行してみると良いでしょう。「gcc」の実行には、Command Line Tools が必要になります。

> インストールされているかの確認

```
$ gcc
```

ダイアログが表示されたらインストールを実行するか、あらかじめターミナルからインストールする場合は以下のコマンドを入力します。

> Command Line Tools をターミナルからインストール

```
$ xcode-select --install
```

インストールが終わったら確認してみましょう。

> gcc のバージョンを確認

```
$ gcc --version
```

インストールされていれば「gcc」コマンドのバージョンが表示されます。

> gcc のバージョン表示

```
gcc --version
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --w\
ith-gxx-include-dir=/usr/include/c++/4.2.1
Apple LLVM version 6.0 (clang-600.0.56) (based on LLVM 3.5svn)
Target: x86_64-apple-darwin14.0.0
Thread model: posix
```

回線環境をシミュレートする設定のインストール

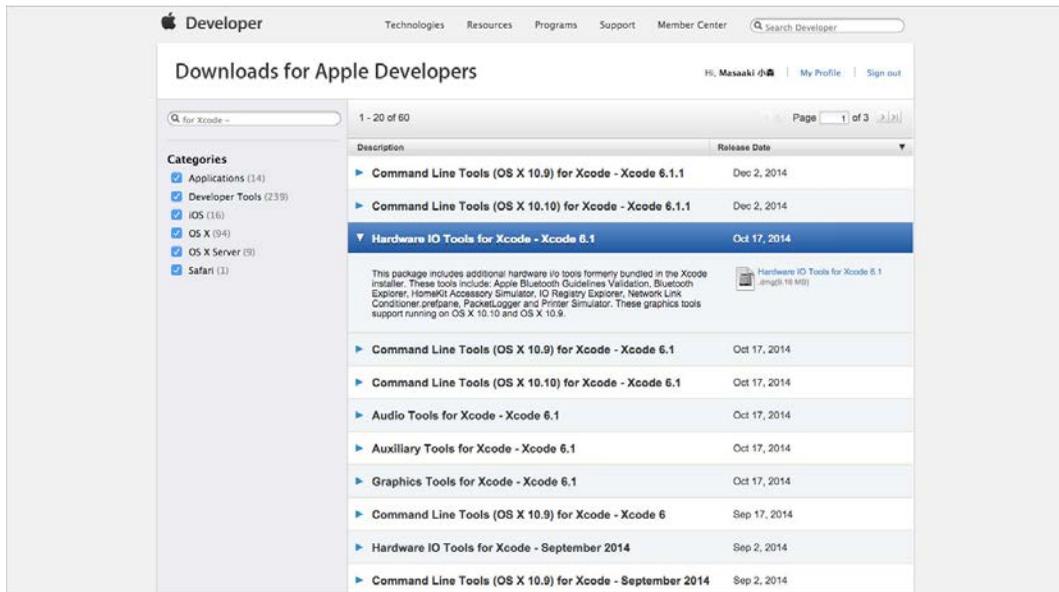
Apple の Developer サイトには、Web 制作の現場で役に立つ「[Hardware IO Tools for Xcode](#)」も公開されています。こちらをあわせてダウンロードしておくとも便利です。このツール群には「[Network Link Conditioner.prefpane](#)」が含まれています。このファイルをダブルクリックしてインストールすると、環境設定のパネルからネットワーク速度に制限をかけることができます。

これがあれば回線環境をシミュレートしながら、制作中もしくは公開中の Web サイトの表示速度チェックが可能です。普段高速な回線で作業していると、遅い回線のことは自分で気にしない限りは気付かれませんからインストールしておきたいものです。同様のことは「[Charles](#)」のような GUI アプリケーションでも可能です。

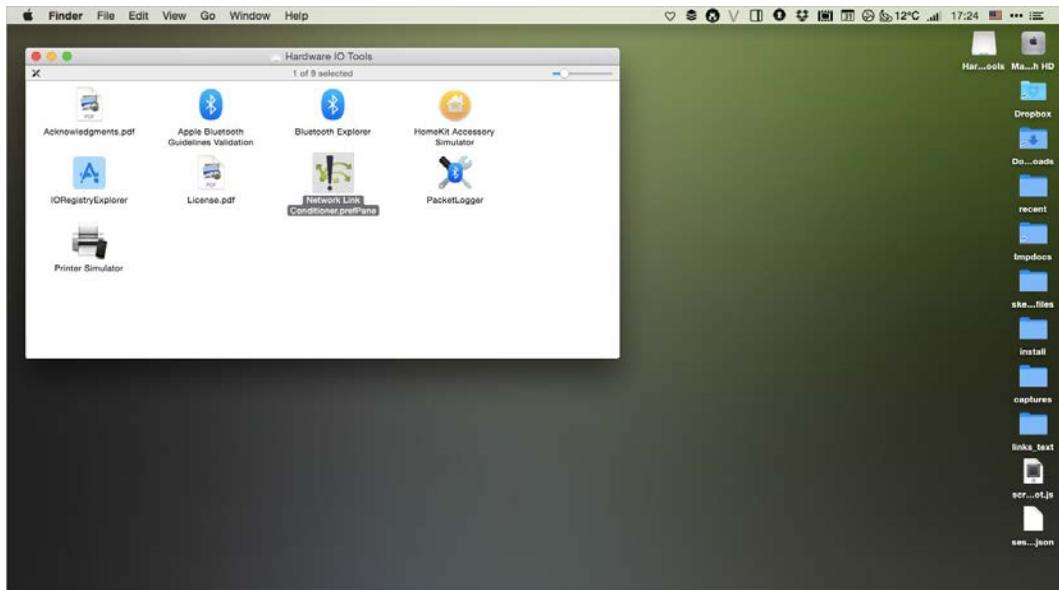
JRE (Java Runtime Environment) のインストール

開発ツールの中には、Java を必要とするものがありますので「JRE (Java Runtime Environment)」をインストールします。Java のパッケージの最新版は Oracle のサイトで公開されていますが、中には Java 6 を要求するものがあり、Apple 社が公開している「[Java for OS X 2014-001](#)」が必要なことがあります。あらかじめこちらをインストールしておきましょう。

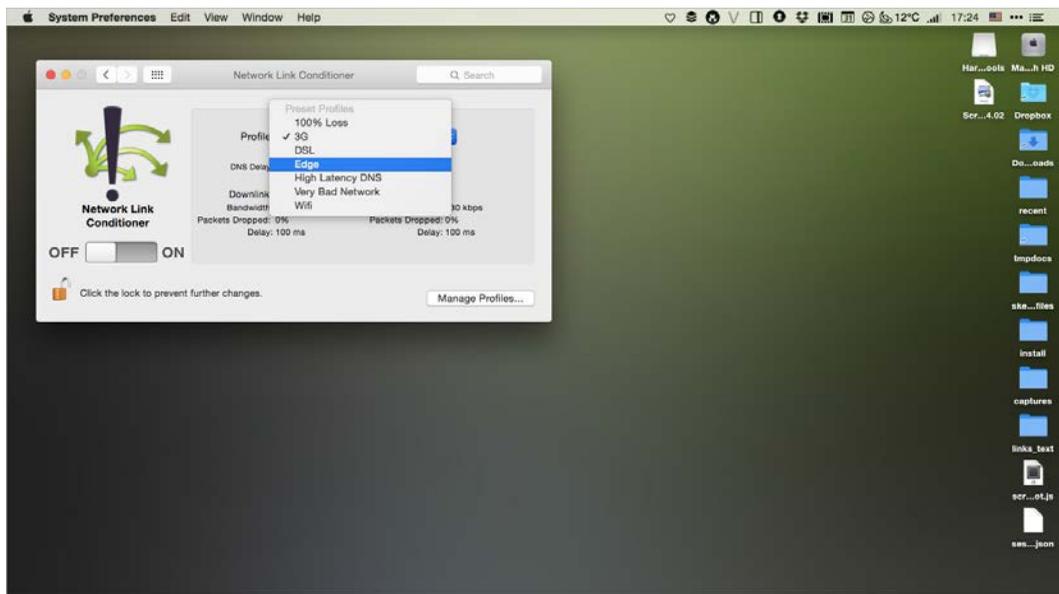
ここまで紹介したもろもろのソフトウェアは、インストールが終わったら特に何もする必要はありません。



Hardware IO Tools も Apple の Developer サイトから



ダウンロードファイルを開いてダブルクリックでインストール



システム環境設定からパネルを開いて通信速度を制限可能

A screenshot of the Apple Store page for "Java for OS X 2014-001". The page header includes the Apple logo and navigation links for "Store", "Mac", "iPhone", "Watch", "iPad", "iPod", "iTunes", "Support", and a search icon. The product name "Java for OS X 2014-001" is displayed in a large font, with a "Download" button below it. A language selector shows "日本語". The main content area contains several paragraphs of Japanese text. The first paragraph states that the update includes improvements for installation and that it is a security update for Java for OS X 2013-005. The second paragraph explains that the update is required for systems with Java for OS X 2012-006 or earlier, and that it will update the Java version to 6. The third paragraph advises users to close Java applications before installing. The fourth paragraph provides a link to the support page for more details. The fifth paragraph provides a link to the security content page. At the bottom, there is a "Post Date: 2013/10/15" and "File Size: 63.98 MB". A "System Requirements" link is also present.

Java for OS X のダウンロード

Homebrew のインストール

OS X は UNIX ベースの OS ですが、UNIX 系の OS でよく利用されるツールがすべて入っているわけではありません。また、ひとつのバージョンとしてパッケージングされて提供される OS という性質上、システムの出荷時にあらかじめインストール済みの各種ソフトウェアのバージョンがどうしても古くなってしまふことが起こります。「Homebrew」は、そういった問題を解決してくれるパッケージマネージャと呼ばれるソフトウェアです。

Yosemite にインストール済みのソフトウェア

Homebrew をインストールする前に、OS X Yosemite にインストール済みで Web 制作の現場で使いそうなソフトウェアのバージョンを確認してみましょう。各種ソフトウェアのバージョンもコマンドラインから確認できます。

実は OS X には、Web(HTTPD)サーバである Apache、そして PHP もインストールされています。Apache は初期状態でただ起動されてないだけです。Apache のバージョンを確認するには、以下のように「-v」オプションを付けて下記のコマンドを実行します(ソフトウェアによっては、--version オプションで表示)。

> Apache のバージョン確認

```
$ httpd -v
```

コマンドを実行すると以下のように表示されます。

> Apache のバージョン表示結果

```
Server version: Apache/2.4.9 (Unix)
Server built:   Sep  9 2014 14:48:20
```

Apache の執筆時点(2014 年 12 月末)での最新版は「2.4.10」で、これは 2014 年の 7 月にリリースされているものです。PHP は複数のバージョンがありますが、OS X にインストール済みの 5.5.x 系の最新版は「5.5.20」です。このようにちょっとだけ古いバージョンになってしまうのです。以下、OS X Yosemite にインストール済みのソフトウェアのバージョンです。

- Apache: Apache/2.4.9 (Unix)
- PHP: PHP 5.5.14 (cli) (built: Sep 9 2014 19:09:25)

- Ruby: ruby 2.0.0p481 (2014-05-08 revision 45883)
- Python: Python 2.7.6
- Git: git version 1.9.3 (Apple Git-50)

次に執筆時点(2014年12月末)での各ソフトウェアの最新バージョンです。

- Apache: 2.4.10
- PHP: 5.4.36/5.5.20/5.6.4
- Ruby: 2.2.0
- Python: 2.7.9/3.4.2
- Git: 2.2.1

ご覧のようにやはり最新 OS であるとはいえ、バージョンには違いがあるものなのです。

決して古いのが悪いというわけではありませんが(重大なセキュリティ上の問題があるものはアップデートされます)、最新の OS であればまだしも数年にわたって使っていくとどうしても古いものを使ってる状態になります。Git については 2014 年 12 月にセキュリティホールが見つかっていますが、2014 年 12 月末の執筆時点では Xcode の β 版をインストールしないと更新されないようです。

Homebrew とは

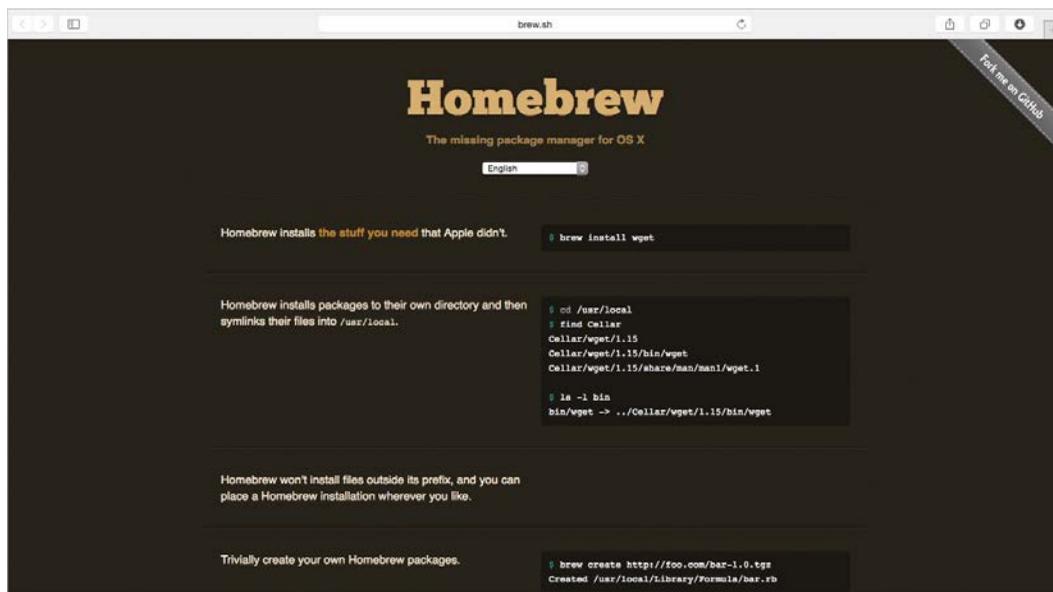
Homebrewは「The missing package manager for OS X」というタグラインが示すように、OS X にはないパッケージを管理するためのソフトウェアです。Linux など UNIX 系の OS に触れる機会がある方には、おなじみのパッケージマネージャーです。

そもそもソフトウェアは、ソースコードがあってそれをコンパイルすることで動作するものです。公開されたソースコード一式をコンパイルするといっても、その他にも必要なソフトウェアが存在します。これらを手動で用意してコンパイルして実行するという作業は、OS に詳しくない人、慣れない人には苦行以外の何ものでもありません。できれば、インストールもアップデートもアンインストールも簡単であるに超したことはありません。

そういった手間を軽減するためにもソフトウェアをすぐに利用できるようにパッケージ化して、その実行に必要な他のソフトウェアとの依存関係までを丸っと管理してしまおうというのがパッケージマネージャーです。各種 Linux 系の OS をはじめとして UNIX 系の OS で

は、使う仕組みこそ異なりますがそれぞれにこのようなパッケージマネージャーが存在しています。この考え方は、いまどきの Web 制作でもよく利用するツールにも採り入れられています。Node.js の「npm」や Ruby の「RubyGems(gem)」は、まさにそれ専用のパッケージマネージャーに他なりません。

Homebrew の役割は、「システム内のファイルを直接変更することなく、比較的最新版のソフトウェアを使えるようにする」「インストールされてない UNIX 系のソフトウェアを簡単に使えるようにして、ソフトウェアのバージョンを含めて全体を管理する」ことだと思ってもらえば良いでしょう。

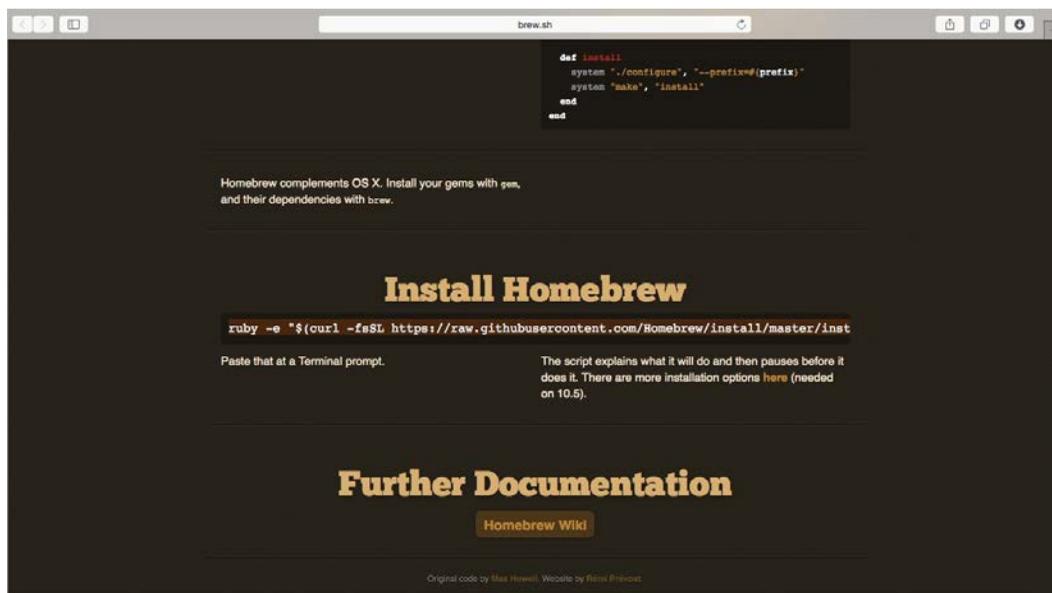


Homebrew の公式サイト

Homebrew のインストール

Homebrew のインストールは簡単です。

公式サイトに記載されたインストールコマンドをターミナルから実行しましょう。下記のコマンドが変わることはないと思いますが、公式サイトトップページ下にあるコマンドをコピーしてターミナルの画面にペーストして「return キー」を押して実行してください。



Homebrew のインストールコマンドはページの下に

> Homebrew のインストールコマンド

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

コマンドを実行すると以下のようにインストールプロセスが始まります。途中パスワードを入力する必要がありますので、ほったらかしにせずしばらく眺めていてください。

```
cipMBA:~ cipher$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
==> This script will install:
/usr/local/bin/brew
/usr/local/Library/...
/usr/local/share/man/man1/brew.1

Press RETURN to continue or any other key to abort
█
```

コマンドをコピー&ペーストして実行。一度リターンキーを入力する

```
cipMBA:~ cipher$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
==> This script will install:
/usr/local/bin/brew
/usr/local/Library/...
/usr/local/share/man/man1/brew.1

Press RETURN to continue or any other key to abort
==> /usr/bin/sudo /bin/mkdir /usr/local

WARNING: Improper use of the sudo command could lead to data loss
or the deletion of important system files. Please double-check your
typing when using sudo. Type "man sudo" for more information.

To proceed, enter your password, or type Ctrl-C to abort.

Password:█
```

パスワードの入力を求められたら入力

> インストール中の出力

```
==> This script will install:
```

```
/usr/local/bin/brew
/usr/local/Library/...
/usr/local/share/man/man1/brew.1
```

Press RETURN to **continue** or any other key to abort (リターンキーを入力)

```
==> /usr/bin/sudo /bin/mkdir /usr/local
```

WARNING: Improper use of the sudo **command** could lead to data loss or the deletion of important system files. Please double-check your typing when using sudo. Type "**man sudo**" **for** more information.

To proceed, enter your password, or **type** Ctrl-C to abort.

Password: (パスワードを入れてリターンキーを入力)

```
==> /usr/bin/sudo /bin/chmod g+rwx /usr/local
==> /usr/bin/sudo /usr/bin/chgrp admin /usr/local
==> /usr/bin/sudo /bin/mkdir /Library/Caches/Homebrew
==> /usr/bin/sudo /bin/chmod g+rwx /Library/Caches/Homebrew
==> Downloading and installing Homebrew...
remote: Counting objects: 218996, done.
remote: Compressing objects: 100% (57781/57781), done.
remote: Total 218996 (delta 160034), reused 218919 (delta 159981)
Receiving objects: 100% (218996/218996), 48.94 MiB | 4.39 MiB/s, done.
Resolving deltas: 100% (160034/160034), done.
From https://github.com/Homebrew/homebrew
 * [new branch]      master      -> origin/master
HEAD is now at 3bdab72 libmagic: update 5.21 bottle.
==> Installation successful!
==> Next steps
Run `brew doctor` before you install anything
Run `brew help` to get started
```

ここまで表示がでたらインストール完了です。最後に記載されているように、まずは「brew doctor」コマンドを実行しましょう(brew doctor コマンドについては後述)。

> インストール後にセルフチェック

```
$ brew doctor
```

Homebrew のセルフチェックが実行され、問題がなければ以下のような表示が出力されるでしょう。

> 'brew doctor' の実行後

```
Your system is ready to brew.
```

Homebrew の各種ファイルは「/usr/local」ディレクトリ以下にインストールされます。Homebrew を使ってインストールされたソフトウェアは、「/usr/local/Cellar」ディレクトリに格納されて、それぞれが「/usr/local/bin」にシンボリックリンクが追加されて使える状態になります。

brew コマンドが実行されない場合は、以下のコマンドを実行してみましょう。

> 環境変数の確認

```
$ echo $PATH
```

表示結果に「/usr/local/bin」があるか確認してください。

> 環境変数の表示結果

```
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

「/usr/local/bin」が含まれていない場合は、以下のコマンドを実行して環境変数のパスに追加します。パスの解説は後述。

> 環境変数にパスを追加

```
$ echo export PATH='/usr/local/bin:$PATH' >> ~/.bash_profile
```

追加した設定を再読み込みします。

> Bash の設定を再読み込み

```
$ source ~/.bash_profile
```

もう一度「brew」コマンドが実行できるか確認しましょう(bre まで入力して tab キーで候補に「brew」が見えれば大丈夫です)。

以下、「brew doctor」を実行して起こりがちなエラーとその対策です。

書き込み権限がないと言われる

Warning: Some directories in /usr/local/share/man aren't writable.
This can happen if you “sudo make install” software that isn't managed by Homebrew. If a brew tries to add locale information to one of these directories, then the install will fail during the link step. You should probably chown them:

```
/usr/local/share/man/man3  
/usr/local/share/man/man5  
/usr/local/share/man/man7
```

「/usr/local」ディレクトリ以下に書き込み権限がないと言われて処理が続けられない場合は、「chown」コマンドを使って所有者とグループを変えましょう。所有者は「自分のアカウント名」、グループは「admin」にすれば大丈夫でしょう。

> 所有者とグループを変更する

```
$ sudo chown -R username:admin /usr/local/*
```

node ディレクトリのヘッダファイルが列挙される

Warning: Unbrewed header files were found in /usr/local/include.
If you didn't put them there on purpose they could cause problems when building Homebrew formulae, and may need to be deleted.

```
Unexpected header files:
/usr/local/include/node/ares.h
/usr/local/include/node/ares_version.h
/usr/local/include/node/ares.h
/usr/local/include/node/node.h
...以下ファイル名が続く
```

Node.js を既にインストールしていてこのエラーが出る人は非常に多いのですが、「/usr/local/include/node」ディレクトリ内にある Node.js のヘッダファイルを消しましょう。

> /usr/local/include/node 内のヘッダを消す

```
$ sudo rm -f /usr/local/include/node/*.h
$ sudo rm -f /usr/local/include/node/openssl/*.h
$ sudo rm -f /usr/local/include/node/uv-private/*.h
```

Node.js を公式のインストーラからでなくインストールする方法は後の章で紹介します。そうすればこのエラーとは無縁になるでしょう。

パスを通す？環境変数に追加する？

エンジニアさんのブログ記事などを読んでみるとよく「環境変数に追加する」「パスを通す」という表現がされています。プログラミングやバックエンドのことに不慣れだと、これらの言い回しでまず躓くのではないのでしょうか？

通常、システムが利用するコマンドは「/bin」や「/usr/bin」といったディレクトリに実行ファイルを置くようになっています。すべてがシステムに関係するソフトウェアであれば良いのですが、そういうものばかりでもありません。時には今回の Homebrew のように標準ではインストールされておらず、後から自分が追加して実行したいソフトウェアも出てくるでしょう。

システムのコマンドがインストールされているディレクトリに自分が追加する実行ファイルを直接入れても良いのですが、それだとシステムのものなのかどうなのかかわからなくなり管理もしにくくなります。そこで、一般的にユーザーが追加するソフトウェアは「/usr/local/bin」や自分のホームディレクトリ直下の任意のディレクトリなどに実行ファイルをインストールします。

しかし、システムがチェックするディレクトリは「/etc/paths」に記述されたものだけなので、それではシェルがそのディレクトリを見つけることができません。そこで必要なのが「パスを通す」という作業です。このパスを通すというのは環境変数の「\$PATH」に対して自分のシェルから実行可能なディレクトリを登録する・追加することを意味します。一般的には Bash の設定ファイルである「~/ .bash_profile」などにその場所を書いておきます。

> .bash_profile を編集

```
$ nano -w ~/.bash_profile
```

追加したいパスをファイルに追記します。

> .bash_profile を編集

```
export PATH=/usr/local/bin:$PATH
```



```
GNU nano 2.0.6 File: /Users/cipher/.bash_profile
export PATH=/usr/local/bin:$PATH
[ Read 1 line ]
^G Get Help  ^O WriteOut  ^R Read File  ^W Where Is  ^Y Prev Page  ^V Next Page  ^K Cut Text   ^U UnCut Text
^X Exit      ^J Justify   ^_          ^C Cur Pos   ^T To Spell
```

パスを追加する

ただ追加しただけではすぐには反映されないなので、「source ~/.bash_profile」を実行して再読み込みします。

> Bash の設定を再読み込み

```
$ source ~/.bash_profile
```

新しくソフトウェアをインストールした際に、パスの追加を促されたらこの作業をおこないましょう。

自分しか使わないマシンなら「/etc/paths」に直接書くという方法もありますが、一般的な方法をお勧めします。パスを追加したのにコマンドが呼び出せない(反映されない)場合は、source コマンドを実行し忘れていないか、パスの順番がおかしくないか、などを確認しましょう。

パスは個別に指定しても良いですし、1行でまとめても構いませんが、その記述順が関係します。同一コマンドが複数の場所にある場合は、先に見つけて欲しいパスを先に記述します。パスの区切りは「:(コロン)」でディレクトリ最後の「/(スラッシュ)」は不要です。

> 1行でパスを追加する

```
export PATH=/usr/local/bin:/bin:/sbin:$PATH
```

シェルの設定ファイルは「.bash_profile」以外に「.bashrc」や「.profile」などいくつかあり、自分のホームディレクトリ直下に配置されますが、それぞれのファイルは読み込まれるタイミングが異なります。詳しいことは以下の記事が参考になるでしょう。

(参考記事)[bash の便利な機能を使いこなそう \(2/2\)](#)

Homebrew のアンインストール

「インストールがうまくいかない」「もう一度入れ直したい」ということもあるかと思いますが、一度インストールした Homebrew をアンインストールする場合は、[公式の GitHub のページ](#)にアンインストールの方法の説明と、アンインストールスクリプトへのリンクがあります。

スクリプトを実行してもうまくいかない場合は、[こちらのコメント欄のスクリプトを保存して実行する](#)と良いかもしれません。手動で構わなければ、以下のディレクトリやファイルをコマンドラインから `rm` コマンドで消去しましょう。

> Homebrew がインストールするファイル

```
"/usr/local/.git"  
"/usr/local/.gitignore"  
"/usr/local/Cellar"  
"/usr/local/Library"  
"/usr/local/CODEOFCONDUCT.md"  
"/usr/local/CONTRIBUTING.md"  
"/usr/local/LICENSE.txt"  
"/usr/local/README.md"  
"/usr/local/SUPPORTERS.md"  
"/Library/Caches/Homebrew"  
"~/Library/Caches/Homebrew"  
"~/Library/Logs/Homebrew"  
"/usr/local/bin/brew"  
"/usr/local/share/man/man1/brew"
```

ブログの記事などでは「`/usr/local/bin` ディレクトリを丸ごと消す」といった記載もあつたりしますがやめておきましょう。そのディレクトリは、他のツールでも使うディレクトリなのでまるごと消すのはお薦めしません。

インストール済みのパッケージがある場合は、あらかじめ「`brew list`」を実行し、再度インストールし直すパッケージをメモしておきましょう。リストには、パッケージの追加時に依存関係でインストールされたものも含まれます（使ってるツールだけ入れ直せば依存しているものは自動で入ります）。ブログなどの記事にならって新しく Homebrew で

インストールする際は、「何を追加したか」をどこかにメモしておくのがポイントです。

Homebrew によるツールのインストールと管理

Homebrew のインストールが終わったら、さっそく Homebrew を使ってあると便利なソフトウェアをいくつかインストールしつつ、Homebrew の使い方に慣れていきましょう。

tree のインストールと実行

「tree」は、任意のディレクトリ以下に含まれるファイルやディレクトリをツリー形式のフォーマットで書き出せるソフトウェアです。Web 制作の作業をしていると、ディレクトリ構造をツリー上に表したいこともあります(何が何でも Excel というわけでもないでしょう)。

> ~/Music 以下を tree で表示した結果

```
├── iTunes
│   ├── Album\ Artwork
│   ├── iTunes\ Library\ Extras.itdb
│   ├── iTunes\ Library\ Genius.itdb
│   ├── iTunes\ Library.itl
│   ├── iTunes\ Media
│   └── iTunes\ Music\ Library.xml
└── sentinel
```

では、さっそく Homebrew でインストールしましょう。Homebrew の公式リポジトリ(登録先)にあるソフトウェアをインストールするには「brew install」コマンドを実行します。「install」の後には半角スペースを入れて、インストールしたいパッケージ名を入力します(Homebrew の各種コマンドについては後述します)。

> tree のインストール

```
$ brew install tree
```

コマンドを実行するとインストールプロセスが表示されます。

Git を導入する

この 1 年あまりでフロントエンド側の制作者の間でも Git を使う人が増えてきました。Git はコマンドラインから使うだけでなく、GUI のクライアントも各種揃っています。普段の作業では GUI のクライアントを使う方が簡単です。Git を使って制作データの変更履歴を管理するだけであれば良いのですが、もう少し踏み込んで Git を使えば同時にサイトを公開する(デプロイする)といったこともできます。ここでは、Git そのものの使い方は深くは取りあげませんが、Git を使ってできる作業フローの改善を紹介します。

Git をインストールする

Git は、プログラマやエンジニアの方の間では以前からおなじみの存在ですが、この 1 年あまりでこれまで馴染みのなかったであろう Web のフロント側を担当する方たちの間でも利用者が増えているようです。まずは、簡単な Git の概要とインストールの方法から解説していきます。

Git について

「Git」は、バージョン管理システム (Version Control System) のひとつです。バージョン管理システムには、他にも CVS や Subversion、Mercurial などがあります。Git はソースコードの変更履歴などを記録・追跡するための分散型のバージョン管理システムで、複数人が同時に作業するチームや組織体で特に有用なものですが、個人レベルでも作業履歴を保持しつつも以前の状態に戻せるなど、非常に利用価値の高いものです。

まだまだ「Git = GitHub」と勘違いされている方も多いようですが、Git はソフトウェアを含むシステムの総称であり、「GitHub」は複数人での作業やオープンソースでの開発をしやすくするための Git リポジトリ (簡単に言えば保管庫) を提供するサービスのことです。このような Git ホスティングサービスには「Bitbucket」をはじめとして無償・有償さまざまなものがあります。Git、GitHub、それぞれの詳しい使い方については 2014 年に出版各社から書籍がいくつか発売されているのでそちらをご覧ください。

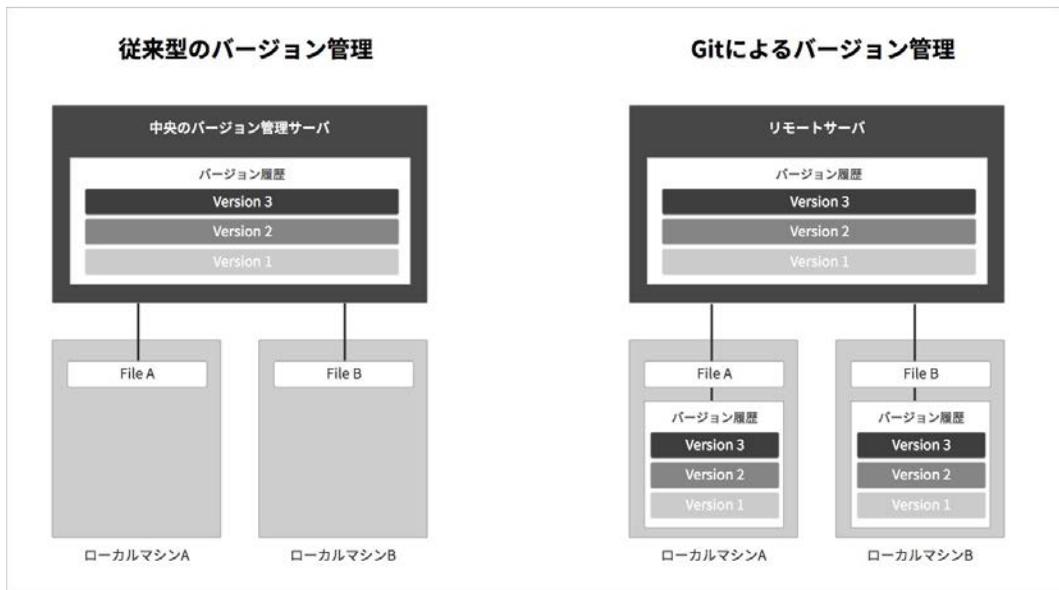
(参考記事) [12 Git Hosting Services Compared - Tower Blog](#)

Git の仕組み

Git は分散型のバージョン管理システムだと紹介しました。バージョン管理システムはその名の通りファイルなどデータのバージョンを管理するもので、ソースコードの管理だけでなく古くからいろいろな形でその仕組みは利用されていたことでしょう。では、従来型のバージョン管理システムと Git の違いから見ていくことにしましょう。

一番単純なバージョン管理は、ファイルやディレクトリを複製して別名で保存することです。立派なバージョン管理ですが、これはマシンが壊れたりしたら終わりです。それ以外に、中央の管理サーバを中心にしたバージョン管理システムもあります。この場合は、管理サーバのデータの一部を取り出して編集して終わったらまたサーバに戻す、といった仕組みです。この仕組みでは中央のサーバに何かしらのトラブルが発生してアクセスできなくなると、変更を戻すどころか他の作業を続けることができません。

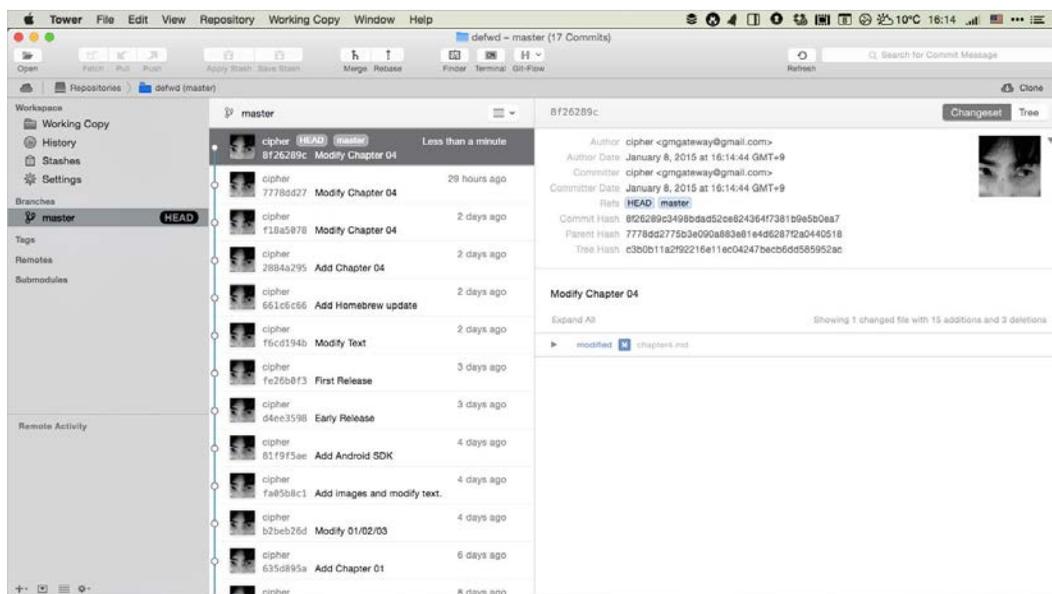
では、Git はそれらと何が違うのでしょうか。Git も他のバージョン管理システム同様、リモートサーバにデータを管理するリポジトリを用意して変更履歴を集中管理することもできます。従来型のシステムとの大きな違いは、リモートサーバのデータをローカルのマシンに変更履歴も含め丸ごとコピー（ミラーリング）してしまうという点にあります。こうすればオフラインであっても編集できるだけでなく、仮に管理サーバが壊れたとしてもどれかひとつのデータを戻せば復元可能です。



従来型と Git のバージョン管理の違い

チームなど複数人での作業となると中央でデータを集中的に管理できた方が便利ですが、Git は必ずしもリモートサーバが必要ではありません。ローカルマシンに Git をインストールしておけば、任意のディレクトリを Git の監視対象とするだけでディレクトリやファイルの変更履歴を記録することができます。Git はリモートだけでなく、ローカルだけで完結するバージョン管理システムとしても動作するのです。

個人レベルに狭めてもバージョン管理ができるとなれば、Web サイト制作や文書作成などいろいろな場面で利用できます。Git で管理さえされていれば、任意のタイミングで変更履歴を保存しながら作業をし、編集時に何か困ったことがあればいつでも任意の時点に巻き戻すことも可能になります。仮に誤ってファイルを削除したとしても、変更履歴のデータさえ消さなければ元に戻るのですから使わない手はありません。

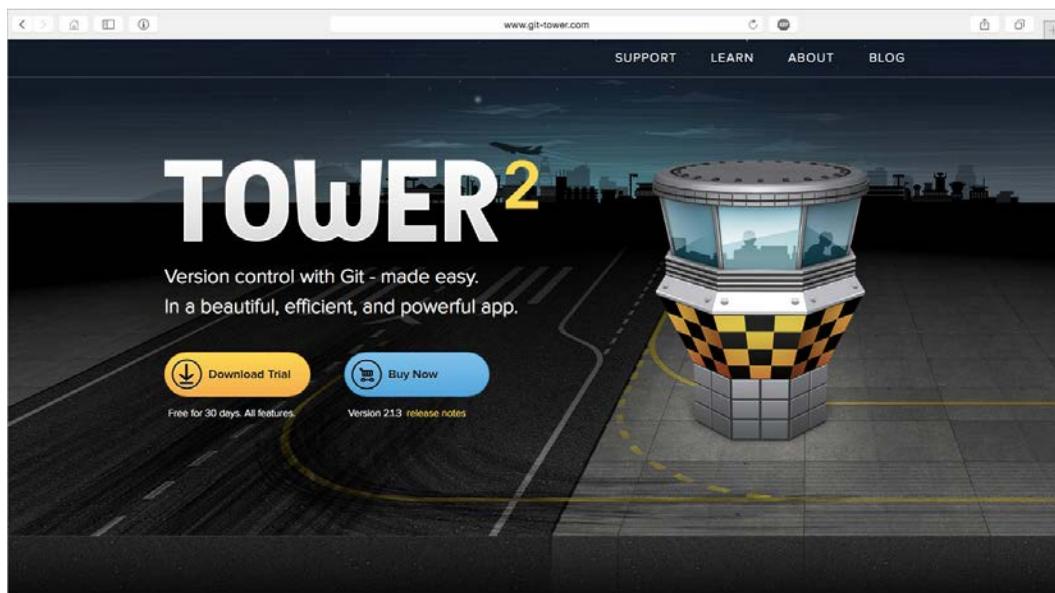


言うまでもなく、この書籍の原稿も丸ごと Git で管理

Git のインストール

OS X では、以下のいずれかで Git を使うことができます。

- [公式サイト](#)で提供されているインストーラを使う
- 「[SourceTree](#)」や「[Tower](#)」といった GUI のソフトウェアを利用する
- Xcode と Command Line Tools をインストールする (Chapter 03 参照)



こちらは有償の Tower

コマンドラインで Git を使うには、公式のインストーラでインストールするのが王道です。しかし、本書では既に Command Line Tools のインストールをおこなっていますので、ここまで順番に読んできた方は既に Git が使える状態になってます（ないと Homebrew が動きません…）。以下のコマンドをターミナルから実行してみましょう。

> Git のバージョンの確認

```
$ git version
```

> Git のバージョン表示

```
git version 1.9.3 (Apple Git-50)
```

Git は現在バージョン 2.2.1 が最新バージョンです（2014 年 12 月末）。2014 年 12 月に重大な脆弱性が発見されており、本来であれば早急にアップデートした方が良いものですが、残念ながら Xcode の β 版をインストールしなければ更新されません。Homebrew で最新版をインストールしてシステムの Git を置き換えてしまいましょう（Homebrew の項で解説したように置き換えてもシステムの Git は残ります）。

> Homebrew で Git をインストール

```
$ brew install git
```

Homebrew のインストールコマンドを実行すれば「/usr/local/bin/git」として最新版がインストールされます。正常にインストールが終わっていれば、バージョンを確認すると最新版の Git が実行されるでしょう。

> Git のパスを確認

```
$ which git
```

> Homebrew の Git のパス

```
/usr/local/bin/git
```

> 再度 Git のバージョンを確認

```
$ git version
```

> 入れ替わった Git のバージョン

```
git version 2.2.1
```

ここまで進んでいけばすんなり切り替わると考えられますが、最新バージョンにならない場合はやはりパスを確認しましょう。OS をバージョンアップしていたり、以前にパスを追加した記憶があれば、設定ファイルの「.bashrc」「.bash_profile」などに記述が分散しがちです。どれかひとつの設定ファイルにひとまとめしておく方が管理も楽でしょう。

GUI クライアントの SourceTree は、最新版の 2.2.1 に変更されたアップデートが出ています。Tower は Git のバージョンが対象外なのか、公式の Twitter などを見てもアップデートが出そうにありません。Homebrew で Git の最新版をインストールして設定から使用する Git を変更しましょう(変更方法については後述)。

Git を使う前に覚えておきたいこと

Git のインストールも終わって本格的に Git を使いたいところですが、Git の仕組みや使い方を解説する前に先に初期設定ファイルの話をしておきましょう。ここからはコマンドラインと GUI クライアントの双方を使いながら話を進めていきます。

Git の初期設定？

Git にも他のソフトウェア同様に設定ファイルが存在します。SourceTree や Tower といった GUI のクライアントで既に Git を使い始めた方は、おそらくその存在を知らないまま作業をされているかもしれません。

UNIX 系の OS におけるソフトウェアの設定は、一般的に「/etc」ディレクトリに保存されます（OS X の場合は「/private/etc」ディレクトリへのリンクになっています）。OS X では直接編集することはほぼありませんが、多くの場合システムにインストールされるさまざまなソフトウェアの設定は「/etc」にあることを覚えておきましょう。では、いつもの `ls` コマンドでちょっと一覧を表示します。

> OS X の `/private/etc` ディレクトリ

```
$ ls -l /private/etc
```

何やら見慣れないファイルやディレクトリが表示されます。これがシステムにインストールされたソフトウェアの設定です。しかし、Command Line Tools でインストールした Git を含め、先ほどのコマンドでリストされた内容を遡ってみてもこの中には Git の設定らしきものはどこにもありません。

通常、全ユーザーに共通のシステム全体で有効な設定は「/etc/gitconfig」ファイルに記述します。しかし、Command Line Tools でインストールされた Git、Homebrew でインストールした Git のいずれもシステム設定としての「gitconfig」ファイルは作りません。

Git の動作設定は、ホームディレクトリの直下に「.gitconfig」という不可視ファイルをグローバル設定として保存しそれを読み込むようになっています。既に Git を使ったことがある方、SourceTree などの GUI クライアントを一度でも実行したことのある方は以下のコマンドを実行してみましょう。「~/ .gitconfig」があれば設定済みの内容が表示されます。コマンド中の「--global」はユーザーのグローバル設定を指示するオプションです。

> Git の設定を表示

```
$ git config --list --global
```

ここでは「git config」コマンドを使って設定内容を読み出しましたが、実体はただのテキストファイルです。初めて GUI のクライアントを起動するとユーザー名とメールアドレスの入力を促されるのは、このグローバル設定を作るためでしょう。Git は「~/ .gitconfig」があれば、それをグローバルな設定として「user.name」と「user.email」に指定されたユーザー情報を使い変更履歴を管理する仕組みになっています（プロジェクト毎に切り替える方法は後述します）。

> ~/.gitconfig ファイルの例

```
[user]
  name = 名前
  email = メールアドレス
```

今回はじめて Git をインストールした方は、Git 使い始める前のお作法として名前とメールアドレスを以下のコマンドで登録してみましょう。2 つのコマンドの実行後ファイルが生成されたか確認します。

> 名前の登録

```
$ git config --global user.name "Masaaki Komori"
```

> メールアドレスの登録

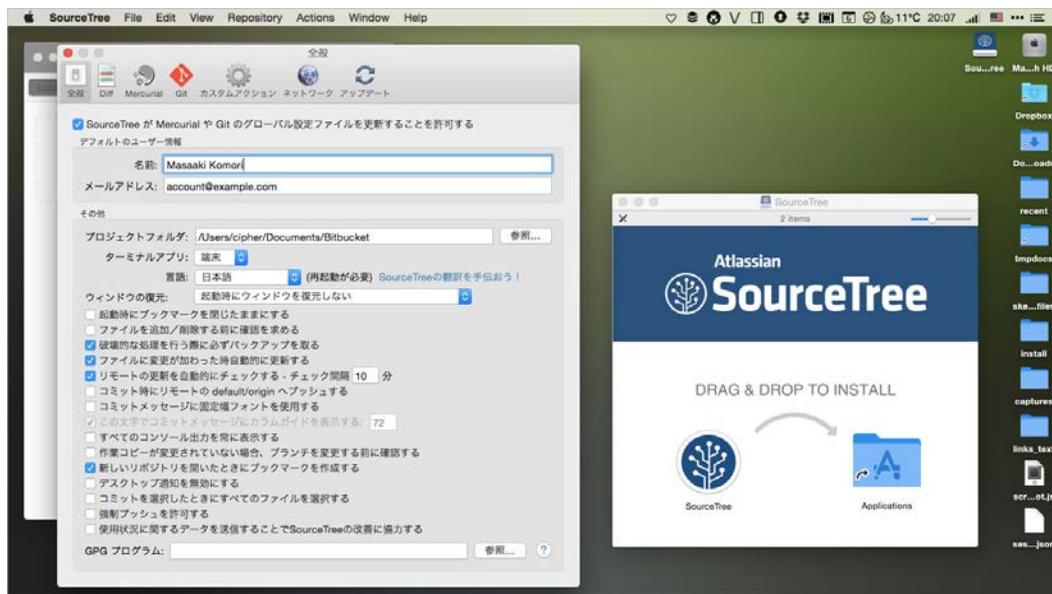
```
$ git config --global user.email account@example.com
```

> Git の設定を表示

```
$ git config --list --global
```

もちろん GUI のクライアントを使っても設定可能です。SourceTree は設定画面を開いて名前とメールアドレスを入力し、一番上にある「SourceTree が Mercurial や Git のグローバル設定ファイルを更新することを許可する」にチェックを入れます（その他の設定を

含め上書きされたくない場合は、先にコマンドラインで.gitconfigを作ってチェックを外します)。Towerは設定画面の「Git」タブに同様の入力欄があります。



SourceTree の設定画面

OS X で Git のシステム設定は以下に保存されるようです(初期設定はなし)

- OS X の Git: /Applications/Xcode.app/Contents/Developer/usr/etc/gitconfig
- Homebrew の Git: /usr/local/etc/gitconfig

Windows 環境との共存のために

Git を OS X だけで使う分には気にすることはありませんが、会社やチームで Git を使う場合には OS 環境は混在しがちです。特に Git の便利さを知って会社でも導入しようと考えた場合など知らないとぶつかってしまう壁もあります。

Git の公式サイトにも記載されているように、Windows 環境と OS X/Linux 環境が混在していると改行コードの違いが問題になります。以下、公式マニュアルの該当箇所からの引用です。

Git を使ってサイトを公開するには？

Git でバージョンを管理しはじめると、「FTP とかもう使いたくない、Git からリモートのサーバのデータを更新できたら…」そんな欲望がフツフツとわきあがるでしょう。リモートの Web サイトにデータを反映させる方法はいくつかあります。

Git と連携してサイトを更新するいくつかの方法

「先方のサーバで Git が利用できない」という声をよく聞きますが、必ずしも外部のサーバに Git がインストールされている必要はありません。「[Beanstalkapp](#)」のような Git リポジトリサービス、「[dploy](#)」や「[deploy](#)」のようなデプロイサービスを利用すれば、サーバの種類やプロトコルも特に問わずいろんなサーバへデータを転送することができます。

リモートのサーバで Git が使える場合

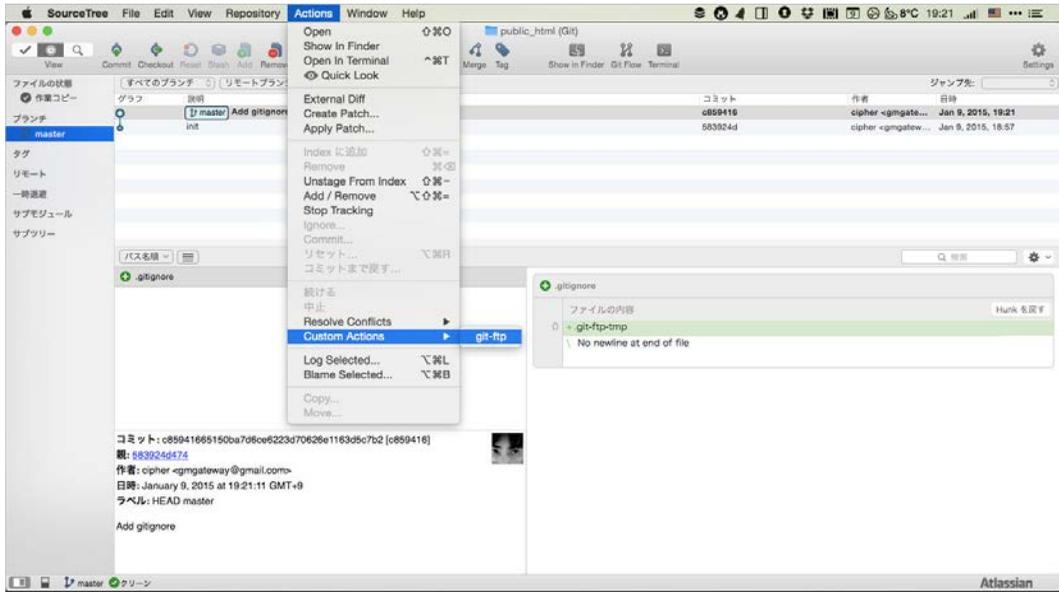
- リモートサーバ内に Git リポジトリを作って更新する

リモートのサーバで Git が使えない場合

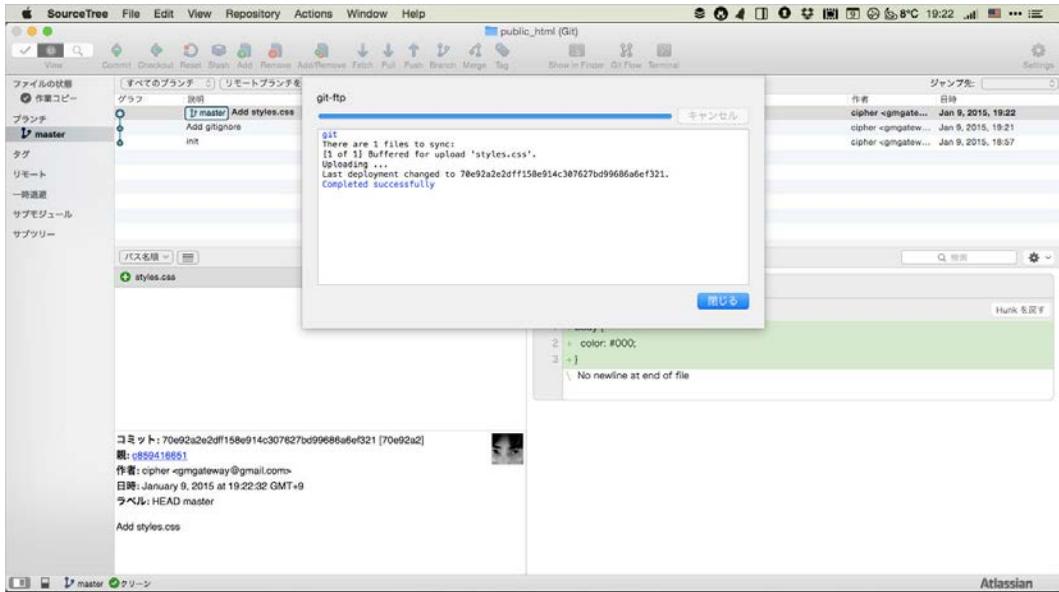
- 外部のデプロイサービスを使って、公開中のサーバに転送する
- PHP などのフックスクリプトを使って、リポジトリのデータを取得する
- ローカルの Git で履歴を管理しつつ、`git-ftp` を使って転送する

各種サービスは無償で使える枠もありますが、仕事となればいくつものクライアントのデータを抱えていることもあるでしょう。管理するリポジトリが増えれば増えるほど費用もかかります。ここではリモートのリポジトリも不要で、予算をかけずに実行できる「ローカルの Git で履歴を管理しつつ `git-ftp` を使って転送する」方法を解説していきましょう。

Chapter 09 ではリモートのサーバに Git をインストールし、リモートリポジトリを作って Push と同時にデータを同期させる方法を紹介します。



リポジトリウィンドウを開いて、登録したアクションを実行



初回だけはコマンド操作だが 2 回目以降は GUI で更新ができる

コミットログをもとにしているため、変更分だけが反映されます。もちろん追加だけでなく削除も可能です。環境によってはエラーメッセージが出る場合もありますが、実際にはファイルは削除されるでしょう。

タスクランナーから実行する

GUIのクライアントではなく、Chapter 07で紹介予定のJavaScriptのタスクランナーを使っても良いでしょう。タスクとして「git ftp init」や「git ftp push」を登録すれば、一連のタスクのついでにでも実行することもできます。[こちらのGitHubのリポジトリ](#)でgulpjsを使った簡単な実行例のコードを公開しています。

設定の「host」の部分は、「ftp://example.com/public_html」のようにアップロードしたいホスト名とパスを指定します。「root」の部分は、リモートのアップロード先と同期させるローカルのディレクトリを指示します。gulpfile.jsからのパスになるため、「./public_html」以下を同期させたい場合はそれを設定しましょう。初回の接続時のみ「gulp ftp-init」を実行します。

> gulpfile.js ファイル

```
var gulp = require('gulp');
var shell = require('gulp-shell');

var config = {
  'username': 'FTP アカウント',
  'passwd': 'FTP パスワード',
  'host': 'FTP のホストとパス',
  'root': '同期させるローカルディレクトリ'
}

gulp.task('ftp-init', shell.task('git ftp init -u ' + config.username + ' -p\
' + config.passwd + ' - ftp://' + config.host + ' --syncroot ' + config.roo\
t))

gulp.task('ftp-push', shell.task('git ftp push -u ' + config.username + ' -p\
' + config.passwd + ' - ftp://' + config.host + ' --syncroot ' + config.roo\
t))
```


node.js の環境構築

JavaScript のランタイムエンジンである node.js は、いまや JavaScript によるサイト制作をおこなわない人でも入れておかねば話にならないもののひとつになりました。node.js はクライアントサイドとサーバサイドの双方で動作するもので、それを利用したサイトを作る以外ではあまり用事はないようにも思われます。しかし、いまどきの制作ツールの多くがそのパッケージマネージャである npm を通じて配布されていること、またそれらを実行するためには node.js が必要なのです。ここでは、node.js を少し簡単に管理する方法とパッケージマネージャの基本操作、そしていくつかの入れておきたいツールを紹介しましょう。

node.js のインストール

node.js は、[公式サイト](#)においてOS 別にインストーラが配布されています。インストーラパッケージでインストールする方法が一番簡単ですが、ここではもう少し取り扱いがしやすいように別の方法でインストールしてみましょ

node.js のバージョンを管理する？

node.js 自体は、機能追加やバグフィックスを含め頻繁にアップデートされます。Web 制作ツールのインストールや実行程度の用途であれば、よほどのことがない限りは本体のアップデートを頻繁にする必要はありません。しかし、node.js にはバージョンの系統があるため、安定版だけではなく場合によっては特定バージョンの node.js にする必要があることも出てくるかもしれません。

公式インストーラは、「/usr/local」以下のシステムの内部にさまざまなファイルが組み込まれる、npm を使ってツールをインストールすると同じくそのディレクトリ内にファイルが入る、モジュールのインストール時にいちいち「sudo」コマンドが必要になるなど、インストールした後に多少不便を感じる場合があります。そこで、ここでは node.js そのもののバージョンを管理できるようにインストールしていきましょう。

node.js 自体のバージョンを管理するツールには、「n」「nvm」「ndenv」「nodebrew」など有名なものがいくつかあります。どれを使うかは好みでもありますが、本書では nodebrew を用いて node.js のバージョンを管理してみましょ。nodebrew は、株式会社ピクセルグリップの外村氏によって開発・公開されており、何か不具合があった場合の対処もきちんとされていて安心です。

nodebrew は、自分のホームディレクトリ以下に「.nodebrew」というディレクトリを作って node.js を管理します。そのため「/usr/local」ディレクトリ以下に見知らぬファイルが追加されることもなければ、ツールのインストールに「sudo」コマンドも要りません。また、ホームディレクトリ直下にあるので必要に応じて環境一式をバックアップするのも簡単です。

「/usr/local」ディレクトリ以下にツールがインストールされるのは構いません。しかし、OS X Yosemite へアップデートする際に「/usr/local」以下がバックアップされて復元されるフローになり、インストールが進んでるのか終わってるのかわからないという状況になった方も多そうです。Homebrew を使う場合も同じですが、システム側にインストールするツールは必要最低限に留めておくのが良いかもしれません。

公式パッケージをアンインストールする

node.js の公式のパッケージのアンインストールは、インストールされた各種ファイル、これまでに追加している npm のパッケージ (node_modules) をアンインストールします。公式パッケージにはアンインストーラーは付いてないので、以下のコマンドを実行してインストールされたファイルを確認して削除します (リダイレクトやパイプを使えば長いリストの確認は楽勝ですね。Chapter 02 参照)。

> パッケージインストールされたファイルの表示

```
$ lsbom /var/db/receipts/org.nodejs.pkg.bom
```

公式インストーラでは以下のファイル、ディレクトリが追加されるようです。ご参考まで。これらを削除すれば良いと考えられますが、実際にインストールされたファイルは念のため上記「lsbom」コマンドで確認してください (バージョンによって大きく異なるとは考えにくいですが…)。

- /usr/local/bin/node
- /usr/local/bin/npm
- /usr/local/lib/node
- /usr/local/lib/node_modules
- /usr/local/include/node_modules
- /usr/local/share/man/man1/node.1
- /usr/local/lib/dtrace/node.d
- /var/db/receipts/org.nodejs.*
- ~/.npm

OS X のパッケージでインストールされたものは、「/var/db/receipts」ディレクトリ内にある「.bom」ファイルを確認します。「.plist」ファイルもありますので、アンインストールする際はこれらも削除しましょう。

> 関連ファイルの削除

```
$ sudo rm -rf /usr/local/bin/node
$ sudo rm -rf /usr/local/bin/npm
$ sudo rm -rf /usr/local/lib/node*
$ sudo rm -rf /usr/local/include/node_modules
$ sudo rm -rf /usr/local/share/man/man1/node.1
$ sudo rm -rf /usr/local/lib/dtrace/node.d
$ sudo rm -rf /var/db/receipts/org.nodejs.*
$ rm -rf ~/.npm
```

既に自分自身で npm を使っていてインストールしているパッケージの一覧は、「`npm list -g --depth=0`」コマンドで確認できます (npm の各種コマンドの解説は後ほど)。

> インストールされているパッケージの表示

```
$ npm list -g --depth=0
```

npm list コマンドを実行して、「ERR! Maximum call stack size exceeded」とエラーが表示されるようであれば、以下のようにスタックサイズを増やして実行してみましょう。

```
node -stack_size=100000 /usr/local/bin/npm list -g --depth=0
```

nodebrew のインストール

nodebrew は、[GitHub の公式ページ](#)で公開されています。インストールはそこに記載された手順通りにやれば問題ありませんが、英語による解説なのでここで簡単にインストール手順を紹介しましょう。

ターミナルを起動して公式ページに記載されたコマンドをコピーして実行します。

npm によるツールのインストールと管理

「**npm**」は、node.js(JavaScript)のパッケージ・マネージャーです。Homebrew などと同じように公開されているツールなどのパッケージを管理するためのソフトウェアにあたります。最近の Web 制作ツールの多くはこの npm、もしくは次章で紹介する gem で管理するものが増えています。

グローバルとローカル、インストール先の違い

npm は本書で既に紹介した Homebrew とは異なり、そのインストール先を「グローバル」と「ローカル」で切り替えることができます。グローバルにインストールした場合はどこからでもコマンドを実行することができるため、「頻繁に使うツールはグローバルにインストール」「コマンドラインから直接呼び出す必要がない、特定のプロジェクトだけに必要なソフトウェアはローカル」といった感じで使い分けると良いでしょう。

nodebrew でインストールされた node.js では npm を使ってパッケージを管理する際に「sudo」コマンドは不要です。グローバルインストールとローカルインストールのコマンドの違いは、「-g(または --global)」のオプションがついているかどうかだけです。

> グローバルにインストールする

```
$ npm install -g package-name (npm i -g package-name でも可)
```

> ローカルにインストールする

```
$ npm install package-name (npm i package-name でも可)
```

nodebrew でインストールされた node.js の場合、グローバルにインストールするパッケージは「~/nodebrew/current/lib/node_modules/」ディレクトリ内にそれぞれ保存され、各パッケージの実行ファイルは「~/nodebrew/current/bin/」内にシンボリックリンクが貼られます。ここは nodebrew のインストール時にシェルのパスを通していているため実行できるのです。

一方のローカルインストールの場合は、コマンドを実行したディレクトリの直下に「node_modules」ディレクトリが生成され、その中にパッケージがインストールされます。ローカルにインストールした場合は、言うまでもなくシェルに対してパスは通っていない状態ですのでコマンドを打っても実行されません。一般的にそのようにしてインストールされたパッ

ページを実行するには「./node_modules/package-name/bin/command」のようにする必要があります（npm からコマンドを呼び出すことは可能）。

たとえば、「npm」コマンドは node.js のインストール時にグローバルに既に追加されています。このように「npm」はパスが通っている場所（システムから見える位置）にあるのでコマンド名を直接書いて実行することができるというわけです。

> npm の場所を調べる

```
$ which npm
```

> npm の場所が表示される

```
/Users/●●●●/.nodebrew/current/bin/npm
```

node.js をインストールした直後の npm のバージョンは「v1.4.x」ですが、npm の最新版は「v2.1.x」になっています。npm 自体をアップデートする場合は「npm i -g npm」を実行しましょう。仮に特定バージョンにあげたあとうまく動かないようであれば、npm のバージョンを下げてみると良いでしょう（コマンドの詳細は後述）。

> npm のアップデートコマンド

```
$ npm i -g npm
```

nodebrew 以外、たとえば公式のインストーラなどでインストールした場合は、npm を用いてパッケージをインストールすると「/usr/local/lib/node_modules」や「/usr/local/include/node_modules」などにファイルが保存され、「/usr/local/bin」ディレクトリ内に実行ファイルのリンクが張られるようです。

npm を使ったパッケージのインストール

ここからは、npm を使って実際にパッケージをインストールしながらその操作に慣れてみましょう。任意の名前で新規のディレクトリを作って移動します。

> --save-dev オプション付きでインストール

```
$ npm i gulp --save-dev
```

こちら先ほどと同様にインストールが終わると、package.json に依存するパッケージが追加されますが、「--save-dev」オプションを付けているので開発時だけ必要なパッケージとしての依存をしめす「devDependencies」の項目にパッケージが追加されました。

> インストール後の package.json

```
{  
  "name": "pkgjson",  
  (中略)  
  "dependencies": {  
    "express": "^4.10.7"  
  },  
  "devDependencies": {  
    "gulp": "^3.8.10"  
  }  
}
```

では、package.json ファイルを別のディレクトリにコピーしてインストールを実行します。もう一度新しくディレクトリを作って、作成した package.json をコピーしてみましょう。

> 新しいディレクトリを作成して移動

```
$ mkdir ~/Desktop/pkgjson2; cd ~/Desktop/pkgjson2
```

> package.json ファイルを複製

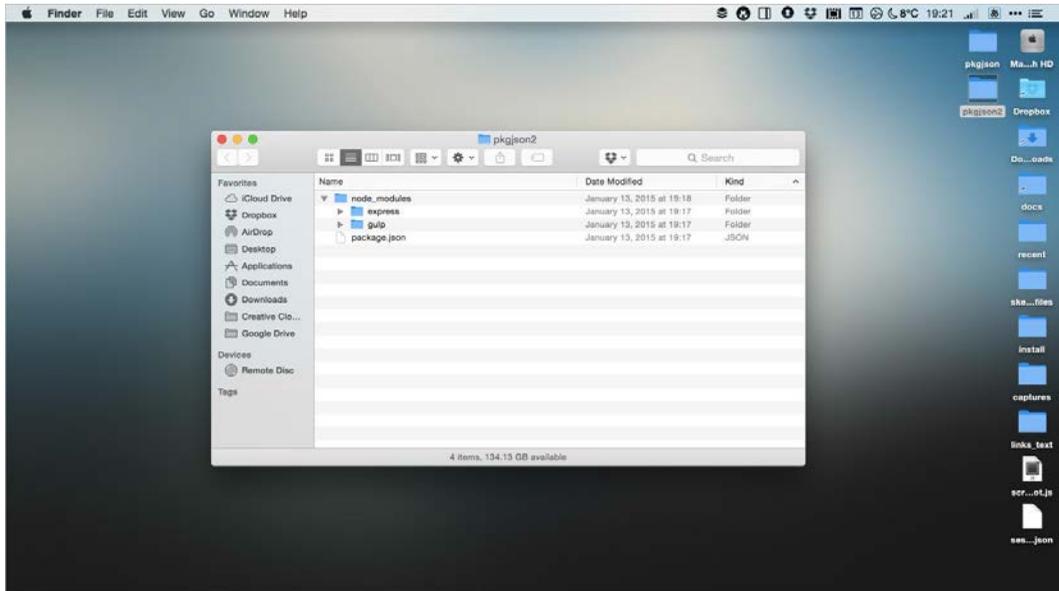
```
$ cp ~/Desktop/pkgjson/package.json . (ドット)
```

package.json がコピーできたらインストールコマンドを実行します。今回は既に内容が記述された package.json をそのまま使うので、「npm install」コマンドだけを実行するだけです。

> npm install コマンドの実行

```
$ npm i
```

インストールするパッケージを指定していないのに「node_modules」ディレクトリが作成され、先ほどインストールした「express」と「gulp」がインストールされました。



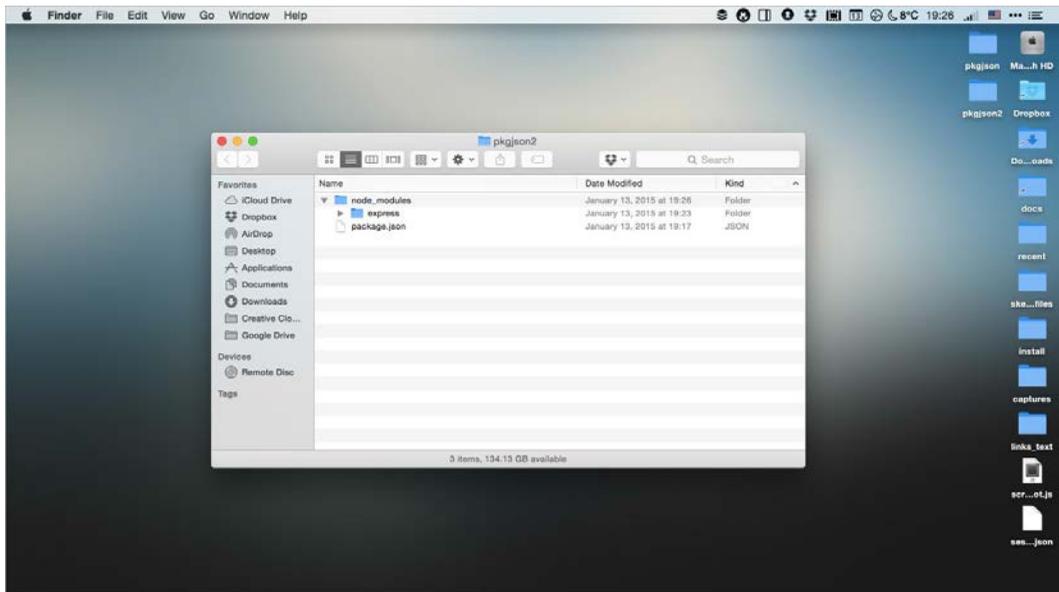
インストールされた express と gulp

では、一度「node_modules」ディレクトリをゴミ箱に移動し、package.json だけにした後、今度は以下のコマンドを実行してみましょう。

> --production オプションを付けてインストール

```
$ npm i --production
```

ディレクトリ内を確認すると「node_modules」の中には「dependencies」に指定した「express」だけがインストールされています。このように「実行時に必ず必要なもの (dependencies)」「開発時だけに必要なもの (devDependencies)」とその用途を分けて登録することで、サイトを公開する本番環境では実行時に必要なものだけをインストールすることもできるようになっています。



オプション付きで実行に必要なものだけでも可能

package.json はインストールだけが簡単になるわけではありません。「scripts」の項目にはあらかじめ「test」というコマンドが登録されています。この「scripts」には「start」や「postinstall」をはじめあらかじめ予約済みのスクリプトを使ってコマンドを実行するだけでなく、自分自身で登録したスクリプト名と実行コマンドの組み合わせを実行することもできます。たとえば以下の「ls」スクリプトを追加してコマンドを実行してみましょう(※ 行末のカンマを忘れずに)。

> scripts に ls を追加

```
{  
  "name": "pkgjson",  
  (中略)  
  "scripts": {  
    "ls": "ls",  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  (以下略)
```

インストールしておきたいツール

ここまでで npm を使ったパッケージのインストールや管理ができるようになりました。Web 制作の現場で使うであろうツールは、人や組織、取り扱う案件の内容などでそれぞれになってしまいますが、ここからは多くの人が使いそうな最低限入れておきたい npm のパッケージとその使い方を取りあげて解説してみましょう。

Bower のインストールと使い方

「**Bower**」は「A package manager for the web」というテキストが示すとおり、Web サイト制作に必要なライブラリやフレームワーク用のパッケージマネージャーです。本書の冒頭でも書いたように、最近のライブラリなどは.zip の圧縮ファイルなどが公開されず、Bower のインストールコマンドと GitHub のサイトへのリンクのみというのも珍しくありません。

Bower は、GitHub などの公開サイトからソースファイルを拾ってくるというのが基本的な仕事です。Bower のサイトに登録されてるものだけを拾ってくるというわけではなく、設定ファイルである「bower.json」に独自の URL を書いてそれを使うといったこともできます。Bower さえ入れておけば、サイト制作の初期段階でありがちなあっち行ってこっち行ってファイルをかき集めるという作業からは解放されます。

Bower はその性質上どこからでも実行できるよう、グローバルにインストールしておく方が便利です。以下のコマンドでインストールしましょう。

> Bower のインストール

```
$ npm i -g bower
```

インストールが終わったら、簡単な動作確認の意味で以下のコマンドを実行しましょう。

> 新規ディレクトリを作成し移動

```
$ mkdir ~/Desktop/bowerdemo; cd ~/Desktop/bowerdemo
```

> ui-route のインストール

```
$ bower install ui-route
```

このコマンドはAngularJSのモジュールである「UI-Router」をダウンロードします。実際にこれを使うには AngularJS の本体が必要になりますが、コマンドを実行するとその動作に依存する本体の AngularJS のファイルもダウンロードされます。また、一度ダウンロードしたファイルは、npm 同様「~/ .cache/bower」ディレクトリにキャッシュされます。「-o」オプションを付ければキャッシュされたソースを使ってオフラインインストールをすることも可能になっています。

> ダウンロードプロセス

```
bower not-cached  git://github.com/angular-ui/...
bower resolve     git://github.com/angular-ui/...
bower download    https://github.com/angular-u...
bower extract     ui-route** archive.tar.gz
bower resolved    git://github.com/angular-ui/...
bower not-cached  git://github.com/angular/bow...
bower resolve     git://github.com/angular/bow...
bower download    https://github.com/angular/b...
bower extract     angular#>= 1.0.8 archive.tar...
bower resolved    git://github.com/angular/bow...
bower install     ui-route#0.2.13
bower install     angular#1.3.8
```

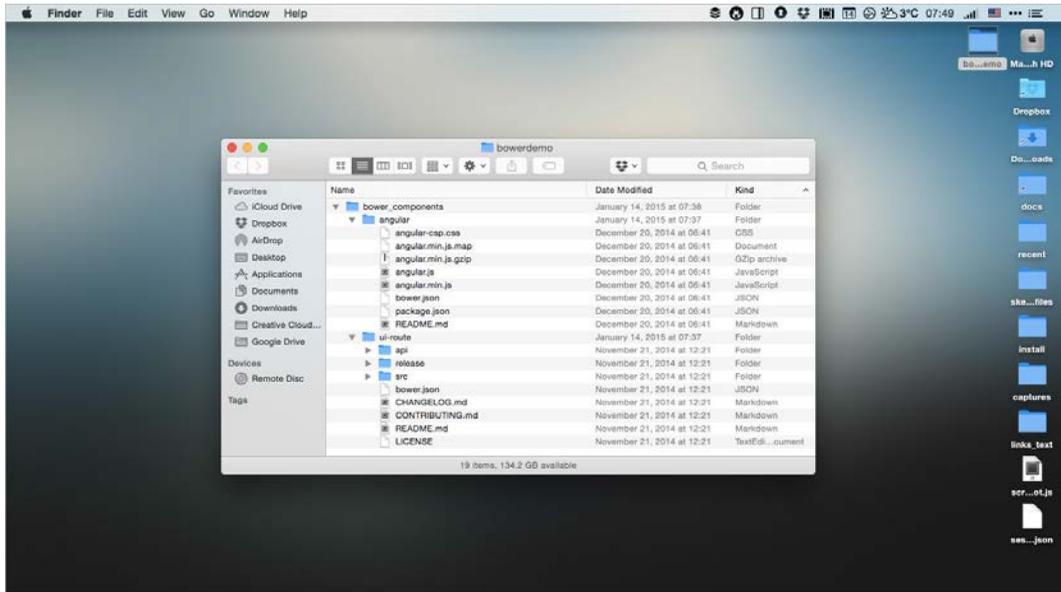
```
ui-route#0.2.13 bower_components/ui-route
└── angular#1.3.8
```

```
angular#1.3.8 bower_components/angular
```

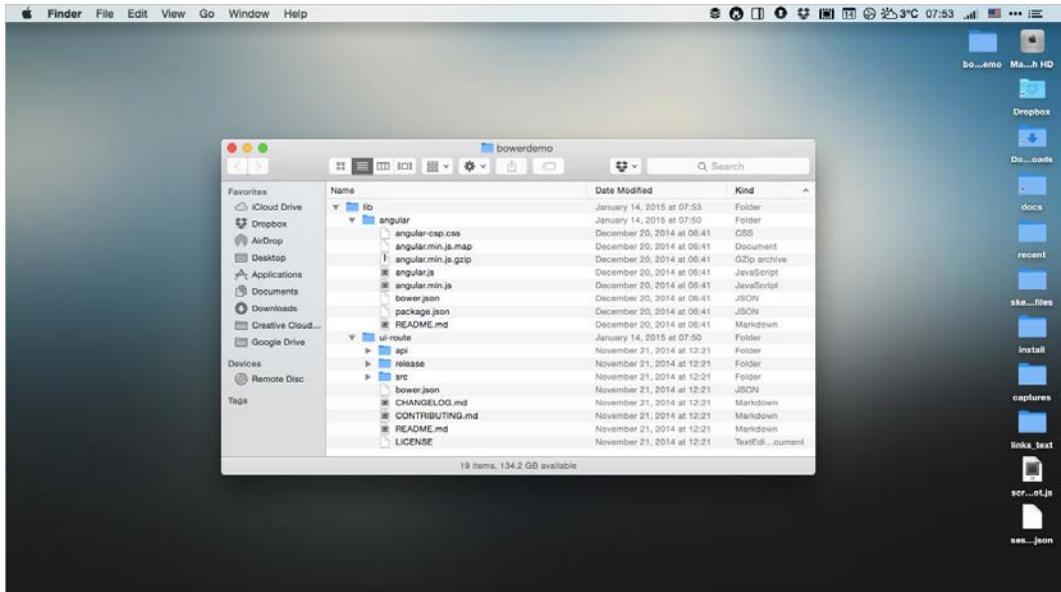
Bower のインストールディレクトリの初期設定は「bower_components」です。これはプロジェクトディレクトリ内の「.bowerrc」というファイルで変更可能です(グローバルに変更したければ、自分のホームディレクトリに)。以下の例のように「directory」に保存したいディレクトリの名称を与えてダウンロード先を変更することができます。

> .bowerrc の記述例

```
{
  "directory": "lib"
}
```



Bower でインストールされた UI-Router と AngularJS



インストールディレクトリの変更は .bowerrc で

Bower も npm 同様にインストールするソフトウェアの依存関係やバージョンを管理する

> インストールされたパッケージをアンインストール

```
$ bower uninstall package-name
```

「~/ .cache/bower」ディレクトリにキャッシュ済みのパッケージをリストします。

> キャッシュ済みのパッケージをリスト

```
$ bower cache list
```

キャッシュの中身をクリアするには「bower cache clean」コマンドを使いましょう。

> キャッシュの中身をクリア(パッケージを指定することも可)

```
$ bower cache clean
```

```
$ bower cache clean package-name
```

npm や Bower などのパッケージを簡単に検索するには、OS X の方はランチャーアプリの「Alfred」を使って npm や Bower のディレクトリを検索するワークフローを入れておくと便利です。

npm と Bower を使った自動化

npm や Bower を使えるようになると、npm の scripts を使って Bower と連携させてサイトの初期準備を自動化するといったこともできます。

Bower でダウンロードされるファイルは GitHub などの公開ディレクトリのデータ一式になっていることも多く、実際に使うファイル以外は不要であることも多いものです。ここではフレームワークの Bootstrap をダウンロードして、ダウンロードしたファイルから必要なソースだけにするような作業を自動化してみましよう。今回使うのは、npm の「postinstall」スクリプトです。

初期設定として以下の「package.json」と「bower.json」を用意します。下記のサンプルはそれぞれ init コマンドを実行して、必要なパッケージとライブラリを登録した状態です。

npm でインストールするのは、Bower でインストールしたライブラリから指定したディレクトリ以外を除去する「Preen」というパッケージです。また、「scripts」に「postinstall」を追加しているところに注目してください。

Ruby の環境構築

Ruby はプログラミング言語のひとつでご存知の方も多いことでしょう。OS X ではシステムにあらかじめインストールされており、その Ruby を使用することは特に問題はありません。本書ではできるだけ OS そのものを汚さずに環境を作るという方向で、OS 側の Ruby をそのままにして別の Ruby をインストールします。また、最近では Sass や Compass を既に導入し実務に活用している方も増えているようですがこれらもバージョンが関係するものです。併せて効率的に管理する方法を紹介しましょう。

Ruby のインストール

OS X の環境では Ruby が既にインストール済みなのでいつでも使える状態です。Windows 環境では Ruby のインストーラーなどを用いれば、動作環境そのものは比較的簡単に用意することができるでしょう。ただ、node.js と同じように制作時にはバージョンの問題も出る可能性もあります。ここからはシステムそのものに手を入れないようにし、複数バージョンを共存可能な方法で新しい Ruby をインストールしてみましょう。

Ruby のバージョンを管理する？

OS にインストールされた Ruby のバージョンはターミナルから以下のコマンドを入力して確認しましょう。インストールされている Ruby のバージョンは OS X のシステムバージョンによって異なりますが、OS X Yosemite の場合は「2.0.0p481」になっています。

> Ruby のバージョン確認

```
$ ruby -v
```

> Yosemite にインストール済みの Ruby

```
ruby 2.0.0p481 (2014-05-08 revision 45883) [universal.x86_64-darwin14]
```

主にフロントエンドを担当するのであれば、RubyGems (gem) を使ってツールをインストールしたり、それらを利用する時だけ Ruby を使っているかもしれません。そのような用途であれば、インストール済みの Ruby をそのまま使ってもさほど問題にはなりにくいと考えられます。しかし、Ruby On Rails をはじめとしてさまざまなフレームワークを使ったサイト開発もおこなわれています。場合によっては特定バージョンの Ruby で固定する必要があるかもしれません。Ruby 本体のバージョンを切り替えるソフトウェアは、「[RVM](#)」や「[chruby](#)」「[rbenv](#)」が有名です。ここでは利用者も多い「[rbenv](#)」を使って Ruby をバージョン管理する方法を解説します。

この数年で CSS プリプロセッサの利用者も増えているようで、特に日本では「[Sass](#)」や「[Compass](#)」が人気です。それらを導入する際には言われるがまま「`gem install sass`」のインストールコマンドで実行している人も多いでしょう。Sass は、バージョンによって使える機能が異なるだけでなくアップデートの頻度も高いものです。それをグローバルのバージョンだけでコントロールしては、長期のプロジェクトの運用には向かないことが容易に想像できます。本章の後半では「[Bundler](#)」を使って、プロジェクトごとにツールのバージョンを管理する方法もあわせて紹介します。

Gem によるツールのインストールと管理

Ruby のインストールが終わったので、フロントエンド側の仕事でよく使いそうなツールをインストールしながら RubyGems(gem)の基本的な扱い方を覚えていきましょう。

RubyGems とは？

RubyGemsは、Ruby で書かれたプログラムやライブラリ(gem)などを管理するパッケージ管理システムです。ここまで解説してきた Homebrew や node.js の npm と同じものだと考えておけば良いでしょう。

他のパッケージマネージャー同様、プログラムやライブラリの依存関係を解決しながら必要なソフトウェアを簡単に導入できるだけでなく、アップデートやアンインストールなどの操作もコマンドから簡単に実行できます。Ruby をインストールすると RubyGems が既に組み込まれている状態ですので、「gem」コマンドを使ってすぐにも使い始めることができます。

> バージョンの確認

```
$ gem --version
```

RubyGems 自体もソフトウェアですので定期的にアップデートしましょう。RubyGems の最新バージョンは「2.4.5」です(執筆時 2015 年 1 月現在)。アップデートコマンドは「gem update --system」です(gem コマンドについては後述)。

> RubyGems のアップデート

```
$ gem update --system
```

> 最新版であれば以下の表示に

```
Latest version currently installed. Aborting.
```

では、さっそくツールのインストールに…といきたいところですが、その前に。他のツール同様に RubyGems にも設定ファイルがあり、ホームディレクトリ直下の「`~/.gemrc`」ファイルで gem コマンドの動作を管理することが可能です。ツールやライブラリのインストールをスムーズにするために、先に「`~/.gemrc`」ファイルを作って以下の内容を追加しましょう。

> インストールされたバージョンの確認

```
$ sass -v  
$ compass -v
```

「gem install」コマンドでインストールするものは、Homebrew でインストールするソフトウェアと同じ状態、node.js の npm で言うところの「グローバル」にインストールされた状態になります。頻繁に使うコマンドで、かつバージョンアップ頻度が高くないもの(バージョンの相違がさほど影響を与えないもの)であれば、次に紹介する「gem update」コマンドでアップデートをかけても特に問題はありません。

しかし、Sass をグローバルにインストールするにはちょっと不安も残ります。Sass の利用者が増え始めた頃のバージョンは「3.3.x」、現在のバージョンは「3.4.x」で当然のように使える機能も異なります。グローバルにインストールしてそれを常に実行するようになって、最新版にアップデートした後に以前のプロジェクトのファイルをコンパイルするとエラーが出てしまう可能性もゼロではないでしょう。

Sass や Compass をグローバルにインストールしたままにしておくのは構いませんが、Sass をヘビーに使う制作環境を構築するのであれば、本章の後半で紹介する「Bundler」もあわせて使う方がプロジェクトごとバージョン管理がしやすくなるのでお勧めです。

default-gems を使った自動インストール

Ruby のバージョンに関係なく必ずインストールする Gems があるなら、rbenv のプラグインとしてインストールしておいた「rbenv-default-gems」を使い Ruby のインストールと同時に自動でインストールされるようにしておくのが楽です。プラグイン自体は最初にインストールしているので、「~/ .rbenv/default-gems」を用意して自動的にインストールしたい Gems の名前を書いておきましょう。

> ~/ .rbenv/default-gems を編集

```
$ nano ~/ .rbenv/default-gems
```

Bundler によるバージョン管理

Sass のように頻繁にアップデートがあるツールやバージョンごとに機能が大きく違うツールなどでは、ひとつのプロジェクトを長く運用する際に以前作ったときにインストールしたバージョンではなくなってしまうこともあるでしょう。そういった問題が起こらないよう、プロジェクトごとに Gems のバージョンなどを管理できるツールを導入しておくといいでしょう。

Bundler のインストール

「Bundler」は、そのサイトに「Bundler provides a consistent environment for Ruby projects by tracking and installing the exact gems and versions that are needed.」と書かれているように、継続的に Ruby のプロジェクトを運用できるように Gems などのバージョンを管理するツールです。

インストールは簡単で「gem install」コマンドで追加します。

> Bundler のインストール

```
$ gem install bundler
```

インストールが終わると「bundle」もしくは「bundler」のいずれかのコマンドで利用できるようになります。

Bundler を使った Gems のインストール

Bundler を使ってプロジェクトごとのツールのバージョンを管理するのは簡単です。まずは、サンプルのディレクトリとして「~/Desktop/sassdemo」を作って移動してコマンドを実行してみます。

> ディレクトリを作成して移動

```
$ mkdir ~/Desktop/sassdemo; cd ~/Desktop/sassdemo
```

Bundler を使って Gems などを管理する場合は「Gemfile」というファイルにその設定を記述します。ディレクトリに移動したら「bundle」コマンドでファイルを作成しましょう。

