Emacs 実践入門

思考を直感的にコード化し開発を加速する

（写経版）

姫 伯邑考

2020 年 04 月 01 日

第 1 章

Emacs の世界へようこそ

1.1　多くの人に愛される歴史あるエディタ

Emacs はテキストエディタである。プログラムを書くためのツールであると思われがちだが、実はそうではない。Emacs の持つポテンシャルをうまく言い表す言葉として「Emacs は環境（OS）である」[*1]という表現が存在する。

1.1.1　ユーザが自由に機能を変更・追加（拡張）することができる

Emacs ではテトリスをプレイすることができる。電卓も利用できる。シェル[*2]も装備している。メールも送受信できるし、ブラウザとしても使える。ゲームからインターネットまで何でもできる、そういったところが「環境」と呼ばれる所以である。

Emacs が他のエディタと最も異なる点は、Emacs Lisp（以下 Elisp と記す）という誰でも気軽に触れることのできる独自のプログラミング言語によって、そのほとんどの機能が作成されている点である。上記の様々な機能も全て、この Elisp によって作成されている。そのため、ユーザの誰もが好きなように機能を変更・追加（拡張）することが可能となっている。故に、Emacs は「Programmable Editor（プログラム可能エディタ）」とも呼ばれる。

1.1.2　使いこなせば強力な味方になる

本稿を閲覧している者は、大きく 2 つのグループに分けられるのではないかと考えている。一方は、まだ Emacs を利用したことはないが、これから利用してみたいと考えている者。もう一方は、Emacs を既に利用しているが、もっと使いこなしたいと考えている者である。

本稿は、その両者を対象として想定している。今まで Emacs に触れたことのない者にとっては、Emacs を活用することができるようになる手助けとなるよう基礎的な利用方法から解説していく。既に利用している者にとっても、最新の Emacs についての話題は勿論、今までコピーで済ませていた設定の意味を正しく理解し、自分の思い通りに設定することができるようになるための足掛かりとなるよう解説していく。

黒い画面、コマンド、キーボードショートカットなどは初学者にとって不安と戸惑いを与えるだろう。しかし、これらはハッカーに強い力を与えてくれる。映画「ソーシャルネットワーク」では、冒頭で主人公の Mark Zuckerberg 氏が軽やかな手つきで Emacs を操り、瞬く間に Web サービスを構築していく姿が描かれている。そういったハッカーと呼ばれる達人たちは、頭から勢い良く溢れ出すコードを即座にプログラムに変えていくのである。

Emacs は古典的でありながらも信じられないほどパワフルな編集機能を与えてくれる。使いこなすことができれば、これほど力強い味方はないだろう。

1.1.3　本当は難しくない Emacs

よく「Emacs の操作は難しい」と言われる。しかし、それは誤解である。タッチタイピングも 1 日で身に付くものではないが、特別な訓練が必要であるわけではなく、ただキーボードをタイプし続けていれば自然に身に付くものである。

[*1] http://c2.com/cgi/wiki?EmacsAsOperatingSystem

[*2] UNIX 系 OS においてコマンドによって OS を操作するためのプログラムのこと

それと同様に、Emacs も何気なく使い続けていると自然と指が慣れてきて、気付けばごく普通に使いこなせるようになるのである。

しかし、自在に操作ができるようになっただけでは、まだ Emacs の本当の実力を引き出せてはいない。柔軟な設定方法を学び、気に入らない部分があれば自分の手に馴染むようにカスタマイズする、そして世界に 1 つだけの自分自身の Emacs を構築することができるようになって初めて一人前となるのである。更に、過去の Emacs とは違い、今日では使いこなせるようになるための近道となるような便利なツールが多数存在する。それらの存在を知ることで、より習得が楽になるだろう。

1.2 本当のエディタにできること

Emacs は非常に多機能である。しかし、応用的な機能を除いて、エディタとして本当に必要な機能にはどのようなものがあるのだろうか？

1.2.1 エディタが本来持つべき機能

テキストデータを編集する際、最もエディタの恩恵を受けていると感じるのは、編集したいと思った箇所にピンポイントで移動することができたり、繰り返しとなる作業を自動化し本人の気付いていないミスを防ぐことだろう。これらを実現するために、エディタとして最低限必要となる機能には次のようなものが挙げられる。

- **検索と置換**
  検索によって、巨大なファイルであっても即座に目的の箇所にカーソルを移動することができる。また、置換により語彙の一括変換や煩雑な入力を手助けしてくれる。
- **入力補助・補完**
  頻繁に入力する定型文、コーディング規約、関数、変数名など、予め定められた入力があれば、それを僅かなタイプで入力できるよう適切に補助してくれる。
- **シンタックスハイライト**
  コードの構文（シンタックス）に従って文字を装飾することで、表示を見やすくしたり、入力ミスを視覚的に示唆してくれる。
- **インデント**
  インデントを整えることでコーディング規約を守り、コードが読み易くなる。自分好みのインデント幅に一括で変更することなども可能である。

これらの機能はタイプミスの削減、可読性や入力速度の向上などに大きく貢献し、テキストエディタには欠かせない重要な機能である。しかし、これらの機能は大抵のエディタに搭載されている機能である。そこで、Emacs ならではの便利機能をいくつか挙げてみる。

- **文法チェック**
  プログラム実行（コンパイル）時のエラーなどを予め検出することで、予期せぬエラーを未然に防ぐことができる。
- **矩形編集**
  複数行に渡って行頭に文字を挿入したいなど、通常の選択範囲では行えない編集を実現する。汎用性があり、複雑な編集も行うことができる。
- **ウィンドウ分割**
  同一、もしくは複数のファイルを同時に参照したいなどに活用される。また、巨大なディスプレイを利用している場合は表示領域を有効活用することにも有用である。
- **履歴**
  編集履歴、クリップボード履歴、ファイルアクセス履歴、カーソル移動履歴など様々な履歴を扱うことができ「間違っても元に戻せる」という安心感を与える。
- **バージョン管理**
  開発の現場では、バージョン管理システムによってソースコードを管理する場合が多い。Emacs では標準で様々なバージョン管理システムに対応しており、Emacs から離れることなく透過的に利用することができる。
- **辞書引き**
  文字を入力する際、日本語・英語を問わず辞書を引きたくなることは頻繁にある。その度にブラウザや辞書アプリ

ケーションを立ち上げるのは億劫であるが、Emacs では様々な辞書データを扱うための拡張機能が色々と用意されている。

- **ドキュメント閲覧・検索**
  辞書と同じく、プログラムのマニュアル・ドキュメントを参照したいことはよくある。Emacs には UNIX マニュアルを読むための機能が用意されている。また、カーソル位置にある単語をドキュメントから検索するなど、エディタならではの使い方が実現できる。

これらの機能はそれぞれ独立しているが、組み合わせることで爆発的な効果が生まれる。全て使いこなせるようになれば、エディタの価値観が大きく変わることになるだろう。

1.3 Emacs が Emacs であるための特徴

前節でまとめたものは「エディタとして」の機能についてであったが、それら個別の機能とは別に「Emacs が Emacs であるための重要な特徴」がある。本稿では、様々な機能を紹介しつつ Emacs の持つ次の 3 つの特徴をテーマに解説を進める。

1.3.1 優れた操作性

まず最初の特徴は、その優れた操作性である。Emacs の操作は主に 2 つに分類される。

1 つ目はカーソル移動である。ファイル内を縦横無尽にカーソル移動させ、コピーやカットなどのエディタらしい機能を使うための操作を指す。これは小さな視点での操作とも言える。

2 つ目は Emacs 全体の操作で、大きな視点での操作と言える。ファイルを開く、保存する、バッファ（これについては第 2 章で解説する）を切り替える、ウィンドウを分割するなど、全てキーボードから行うことができる。勿論、マウスも利用可能だが、キーボードから手を離さず全てが操作可能ということは、非常に大きな意味を持つ。

更に、操作は次に挙げる設定の柔軟性によって自分好みにカスタマイズすることができる。

1.3.2 設定の柔軟性

Emacs と言えば、設定の柔軟性なくして語ることはできない。前述した通り、Emacs のほとんどの機能が Elisp によって実装されている。表示や操作はデフォルトの設定値が与えられているだけなので、その値を変更することによって、自分の好きなようにカスタマイズすることができるのである。

但し、Emacs は様々な機能が組み合わされて構成されているため、どこに記述された設定が反映されているのか、またその優先順位がどの様になっているのかを把握するためには少々経験が必要となる。場合によっては思い通りにならないこともあるだろう。しかし、そういったことも第 3 章で解説するヘルプ機能を利用して調べるスキルを身につけることによって解決することができるようになるだろう。

1.3.3 本体の拡張性

最後に挙げるのは本体の拡張性である。Emacs の機能は本体のバージョンが上がる度に増えている。実は、それらの機能のほとんどは、誰かが Elisp を用いて作成したもので、評判の良いものをメーリングリストなどで議論して本体に組み込んでいるのである。

Emacs はフリーウェアの代表格とも言えるソフトウェアである。非常に多くの開発者が開発を支えている。しかし、何も本体の開発に直接参加している者達だけが貢献者ではない。例えば、小さな機能を考えて Web 上に公開することも重要な素晴らしいコミットである。

自分が欲しいほんの小さな機能、そういったものも自分で作成可能ということは非常に大きな魅力ではないだろうか。

第 2 章

インストール、設定ファイルと画面の構成

2.1　インストール

Emacs のインストールの仕方は幾つか存在するが、大きく分類すると次の 2 つに分けられる。

- ファイルをダウンロードしてコピーするだけで利用可能なバイナリ*1インストール。
- ソースコードをダウンロードし、ビルドしてインストールするソースインストール。

本節では、そのどちらも解説いていく。

2.1.1　Mac へのインストール

Mac には予めターミナルで動作する Emacs 22.1 がインストールされている。しかし、これは 2007 年にリリースされた古いバージョンであるため、本稿で紹介する機能を利用するためには最新バージョンをインストールする必要がある。

Mac 向けの Emacs には、ターミナル上で動作する Emacs 以外にもデスクトップ上で動作する Emacs.app が利用可能である。後者は一般的な macOS アプリケーションと同じルック＆フィールが用いられているため、普段使い慣れたアプリケーションと同じように Emacs を利用することができる。

Emacs.app のインストール方法は、主に既にビルドされた Emacs.app をダウンロードする方法とソースコードからインストールする 2 つの方法が存在する。

Emacs.app を入手する

macOS 向けの公式バイナリは存在しないが「GNU（グヌー） Emacs For Mac OS X」という準公式のバイナリがダウンロードできる Web サイト*2が存在する。

そこにはリリース版、プレテスト版（リリース候補版）、ナイトリー版（毎日ビルドされる開発版）と様々なバイナリが用意されている。ディスクイメージファイル（dmg ファイル）をダウンロードしてマウントし、開いたウィンドウ内に存在する Emacs のアイコンを Applications ディレクトリにドロップするだけでインストールすることが可能である。

Emacs.app を自分でビルドする

macOS 付属の開発環境である Command Line Tools がインストールされていれば、Emacs の本家である「GNU Emacs — GNU Project」*3からソースコードをダウンロードして Emacs.app を作成することもできる。Terminal.app（ターミナル）などの端末から次のコマンドを実行することで Emacs.app を自分で簡単にビルドすることができる（$ はコマンドプロンプト）。

```
$ curl -0 http://ftp.gnu.org/pub/gnu/emacs/emacs-25.2.tar.gz
$ tar xvf emacs-25.2.tar.gz
$ cd emacs-25.2
$ ./configure --with-ns
$ make install
```

*1 既にビルドされたソフトウェアのこと。
*2 http://emacsformacosx.com
*3 http://www.gnu.org/software/emacs

これで「./emacs-25.2/nextstep」ディレクトリ内に Emacs.app が作成されるので、これを Applications ディレクトリにドラッグ＆ドロップしてインストールする。

ビルドに関する詳細は「./emacs-25.2/nextstep/INSTALL」に説明があるので、必要に応じて参照するとよい。

2.1.2 Windows へのインストール

Windows 向けの Emacs は GNU Project が配布する Emacs、Windows Subsystem for Linux（以下、WSL と記す）の Emacs、そして Emacsen（Emacs 派生エディタ）である Meadow など、様々なディストリビューションが存在し、情報が少々錯綜している。特に問題がなければ、公式に配布されている Emacs を利用するとよいだろう。

ただ、Windows 10 から利用可能となった WSL では Linux と同じ環境が実現できるため、grep などの Linux コマンドを Emacs と組み合わせて利用したい者にとっては有用な選択肢となる。そのため、WSL を扱える場合はこちらの Emacs を利用してみるのもよいだろう。

オフィシャルビルドを利用する

Windows に Emacs をインストールするには、バイナリをダウンロードしてインストールするのが一番簡単である。オフィシャルビルドは次のアドレスからダウンロードすることができる。

https://ftp.gnu.org/gnu/emacs/windows/emacs-27/emacs-27.1-x86_64.zip

解凍したフォルダを任意の場所（C:¥emacs など）にコピーしてインストールする。

2.1.3 Linux へのインストール

Linux では既に Emacs がインストールされている場合が殆どだが、ディストリビューションによってはインストールされていなかったり、バージョンが最新ではない場合がある。もし、最新バージョンのパッケージが用意されているようなら、パッケージマネージャーを利用してインストールするのが手軽である。

パッケージに最新の Emacs が存在しない場合は、次のコマンドを利用してインストールする。但し、X Window System（以下、X と記す）を使用せず、ターミナル環境のみで Emacs を利用する場合は、次の項で解説する方法でインストールする必要がある。

```
$ curl -0 http://ftp.gnu.org/pub/gnu/emacs/emacs-27.1.tar.gz
$ tar xvf emacs-27.1.tar.gz
$ cd emacs-27.1
$ ./configure
$ make
$ sudo make install
```

2.1.4 ターミナル環境へのインストール

ターミナル環境への Emacs をビルドする場合、次のような方法が最もシンプルである。このインストール方法は Mac と Linux で共通である。

```
$ curl -0 http://ftp.gnu.org/pub/gnu/emacs/emacs-27.1.tar.gz
$ tar xvf emacs-27.1.tar.gz
$ cd emacs-27.1
$ ./configure --without-x
$ make
$ sudo make install
```

これで「/usr/local/」へ Emacs がインストールされる。

起動方法については第 3 章で解説する。

2.2　ディレクトリと設定ファイルの構成

Emacs はベースとなる部分以外の機能は Elisp によって実装されている。そのため、Emacs をインストールすると大量の Elisp も一緒にインストールされる。OS によって多少インストールされる場所は異なるが、ディレクトリ構造はほぼ共通である。

2.2.1　各ディレクトリの役割

Emacs 本体と一緒にインストールされるディレクトリの中で、重要なものを紹介しておく。

etc ディレクトリ

etc ディレクトリには Emacs の NEWS やライセンスなどのドキュメントが格納されている。NEWS には追加された機能などの情報が掲載されている。

leim ディレクトリ

leim ディレクトリには Emacs 標準の IM（Input Method）が格納されている。特に設定しなくても利用することができるのは魅力的だが、IM としてはあまり強力ではない。Emacs で快適な日本語入力を求めるのであれば、SKK（Simple Kana Kanji conversion program）を推奨する。

lisp ディレクトリ

lisp ディレクトリには Emacs 同梱の Elisp が格納されている。Emacs の全てがこのディレクトリに詰まっていると言っても過言ではないだろう。

site-lisp ディレクトリ

site-lisp ディレクトリは、ユーザが Elisp をインストールするために標準で用意されているディレクトリである。ロードパス（Emacs が Elisp を探すディレクトリ）が通っているため、このディレクトリにインストールされた Elisp は load や require などの関数を利用することで読み込むことができる。

しかし、複数ユーザで同じ Elisp を利用する目的以外では site-lisp に Elisp をインストールすることは推奨しない。.emacs.d ディレクトリの中にインストールする方法を利用すると.emacs.d ディレクトリをコピーするだけで環境がコピーできるため、そちらを推奨する。詳しくは、第 4 章「~/.emacs.d ディレクトリに設定をまとめて管理」で解説する。

bin ディレクトリ

bin ディレクトリは Windows 版の Emacs のみに存在し、Emacs 本体や Emacs から利用するためのコマンドの実行ファイルが格納されている。また、このディレクトリに配置された実行ファイルは Emacs から呼び出し可能となる。

ホームディレクトリに作成される.emacs.d ディレクトリ

Emacs を起動すると、ホームディレクトリに.emacs.d というディレクトリが作成される。このディレクトリは Emacs の設定ファイルを配置するために用いる。

2.2.2　Windows のホームディレクトリ設定

Windows 10 の初期状態では「C:¥Users¥ ユーザ名 ¥AppDate¥Roaming」をホームディレクトリとして利用する。このディレクトリは Windows の環境変数 HOME を設定することで自由に変更することができる。

Windows 10 の場合［コントロールパネル］の［システムとセキュリティ］から［システム］の［システムの詳細設定］を開いて［詳細設定］タブの［環境変数］をクリックし［ユーザー環境変数］に、次のように環境変数を新規追加する。

- 変数名：HOME
- 変数値：値は任意。設定を作成したいフォルダを指定する。例：C:¥Users¥ ユーザ名

この設定は次回ログイン後に反映され、Emacs を起動すると設定したフォルダに.emacs.d ディレクトリが自動的に生成される。

2.2.3　設定ファイルの構成

Emacs の初期化ファイル（設定ファイル）は.emacs と呼ばれる。ドットファイルというのは UNIX 系 OS 特有の設定ファイルのことで、ドットから名前が始まることに由来する。ドットファイルは通常隠しファイルとなっており、意図せず削除してしまったりしないようになっている。

第 1 章において Emacs の特徴として「設定の柔軟性」を挙げたが、その柔軟な設定は設定ファイルに記述することで Emacs を起動する際に一度だけ読み込まれ反映される。設定ファイにルは.emacs や init.el など複数の名前が用意されているが、読み込まれるのは 1 つだけである。その優先順位は次のようになっている。

① ~/.emacs.el（.emacs.elc）
② ~/.emacs
③ ~/.emacs.d/init.el（init.elc）
④ ~/.emacs.d/init

全てのファイルはバイトコンパイル（後述）することで elc ファイルに変換することができる。Emacs では el ファイルと同名の elc ファイルが存在する場合は elc ファイルを読み込み、el ファイルに記述された設定は一切読み込まれないことに注意が必要である。

上記のどのファイルに設定を記述しても Emacs の動作に違いはない。もし、どのファイルに記述すればよいか迷った場合は init.el を推奨する。なぜなら.emacs.d ディレクトリをバックアップするだけで Emacs の設定をバックアップすることができるためである。詳しくは第 4 章で解説する。

2.3　画面の構成

Emacs の画面について解説していく。

2.3.1　フレーム

Windows や Mac でウィンドウと呼ばれるアプリケーションの表示枠のことを、Emacs ではフレーム（*frame*）と呼ぶ。frame と名の付くコマンドや変数（後述）は、このフレームに関する操作を行うものである。

2.3.2　ウィンドウ

Emacs のフレーム上で、ファイル（正確には後述するバッファ）を表示している領域のことをウィンドウと呼ぶ。ウィンドウは分割することが可能で、フレーム内に幾つものウィンドウを並べることができる。また、Emacs は複数のファイルを同時に開くことができるため、異なるファイルや同じファイルの異なる箇所を別のウィンドウに表示しながら編集することができる。

2.3.3　フリンジ

ウィンドウの両端に位置するアイコンを表示可能な細い特殊なウィンドウで、主に行に関する追加情報を視覚的に表示するための場所をフリンジと呼ぶ。例えば、実際には改行されていないが Emacs の画面上では行の折り返しが行われている場合は、折り返し記号を表示する。他にも、EOF（End Of File）などを表示させることも可能である。

2.3.4　バッファ

一般的なコンピュータ用語では、情報を一時的に蓄える記憶領域のことをバッファと呼ぶが、Emacs ではメモリ上に作成されたオブジェクトのことをバッファと呼ぶ。更に直感的に説明するなら、ウィンドウの中に表示されているもの全てがバッファである。Emacs でファイルを開いた場合、そのファイルの内容が全てバッファへと読み込まれてウィンドウに表示される。すなわち、ファイルは Emacs 上では全てバッファと呼ばれることになる。

バッファはあくまでも Emacs 上のオブジェクトなので、ファイルとして保存しない限りその編集はファイルに反映されることはない。また、一度作成されたバッファは明示的に消去しない限り（または Emacs を終了しない限り）Emacs 上に残り続け、バッファを消去することによってメモリから開放される。そして、Elisp によるプログラミングではバッファを変数のように扱うこともできる。

2.3.5 モードライン

モードラインは、ある意味究極的に素っ気ない Emacs において唯一華のある部分である。ウィンドウの下部に存在し、カレントバッファ（編集中のバッファ）に関する様々な情報を表示する。また、バッファに関係のない情報であっても Emacs からアクセス可能であれば何でもモードラインに表示することができる。

2.3.6 ミニバッファ（エコーエリア）

Emacs のフレーム最下部に存在するミニバッファは、Emacs へのコマンドや引数を入力する場所として利用する。ミニバッファでは tab や space による補完機能が利用できることが多い。また、この場所は Emacs からのメッセージを表示するエコーエリアとしても利用される。同じ場所を使用しているが、内部的にはミニバッファとエコーエリアは全くの別物である。

2.4 モード

Emacs を操作する上で必ず理解すべきものがモードという概念である。モードとは一体どういうもので、何を提供してくれるのかを知ることで、Emacs に対する理解が深まるだろう。

2.4.1 メジャーモード

メジャーモードとは、バッファに対して「必ず 1 つ」適用されるモードである。例えば、*scratch* バッファでは lisp-interaction-mode という Lisp システム実行環境用のモードが適用され、rb ファイルを開くとプログラミング言語 Ruby を編集するための ruby-mode が適用される。現在のメジャーモードはモードラインに表示される。

メジャーモードを選択する仕組み

Emacs が自動的にメジャーモードを選択する仕組みは非常に単純で、ファイル名（主に拡張子）とファイルの shebang（シバン）を元にモードを決定する。

auto-mode-alist はファイル名からモードを選択する仕組みである。この変数にはファイル名にマッチさせるための正規表現とマッチした際に選択されるモードのリストが記述されており、先頭から順番にチェックしてマッチした瞬間にそのモードを選択してファイルを開く。

interpreter-mode-alist は shebang からモードを選択する仕組みである。この変数にはファイルの最初の行にある shebang に記述されているインタプリタ名にマッチさせるための正規表現と、マッチした際に選択されるモードのリストが記述されている。Emacs は shebang が存在する場合に限って、このリストを利用してモードを選択する。尚、優先順位は interpreter-mode-alist > auto-mode-alist となっている。また、もし何もマッチしなければデフォルトで text-mode というテキストファイルを編集するためのメジャーモードが選択される。

メジャーモードが提供する機能

メジャーモードが提供する機能には、主に次のようなものがある。

- メジャーモード専用コマンド（例えば、lisp-interaction-mode で式を評価し戻り値を出力する eval-print-last-sexp など）
- 特別なキーバインド（キーマップ）
- シンタックスハイライト
- フック（第 5 章で後述）

2.4.2　マイナーモード

マイナーモードはメジャーモードと異なり、 1 つのバッファに対して複数適用することができる。その最大の特徴は、使いたい際に有効化して必要がなければ無効化することができる点である。マイナーモードが提供する機能は、補完を手助けしてくれたり、表示を大きくしたり実に様々である。現在のマイナーモードもモードラインに表示される。

マイナーモードが提供する機能

マイナーモードが提供する機能には、主に次のようなものがある。

- マイナーモード専用コマンド（例えば、auto-complete-mode で自動補完を行う auto-complete など）
- 特別なキーバインド（キーマップ）
- フック（フックが定義されている場合）

マイナーモードについて詳しくは第 7 章で解説する。

第 3 章

基本的な操作

3.1　コマンド

Emacs はエディタなので文字入力ができるのは当然だが、文字を入力するだけなら単機能なエディタと同じであると言える。Emacs が多くの開発者に愛されているのは、様々なことが行える優秀なコマンドが多く用意されているからである。

3.1.1　入力して実行する

コマンドを実行する方法は 2 通り存在する。

その 1 つがコマンド名を入力して実行する方法である。Emacs 上で Alt+x を押すとミニバッファにフォーカスが移動し、コマンド入力待ちの状態となる。この状態で、例えば about と入力して Tab を押してみる。Tab はミニバッファで入力を補完してくれる非常に便利なキーである。about から始まるコマンドは about-emacs のみなので about-emacs が補完されたはずである。この状態で Enter キーを押すと、環境にインストールされている Emacs のバージョン情報が表示されるバッファが開く。

続いて、同じく Alt+x を押してコマンド入力待ちの状態にし、help-with-tutorial-spec-language と入力してみる。今度はウィンドウが分割され、ミニバッファには Language: と表示されるはずである。これは、更なる入力を求めている状態であり、japa と入力して Tab キーを押す。Japanese が補完されるので Enter を押すと、Emacs チュートリアルの日本語版が開く。このチュートリアルは、読み進めながら指示通りに操作することで Emacs の一連の操作をマスターすることができるようになっている。

3.1.2　キーバインドから実行する

もう 1 つのコマンド実行方法はキーバインドから実行する方法である。キーバインドとは一般的にキーボードショートカットと呼ばれるもので、修飾キーと文字キーを組み合わせた操作のことである。

Emacs には様々なキーバインドが用意されている。このキーバインドを覚えることで、魔法のように速く編集を行うことができるようになる。

何か文字のある行の途中で Ctrl+a を押してみる。すると、カーソルが行頭に移動する。続けて Ctrl+e を押すと、今度はカーソルが行末に移動する。これは、それぞれのキーバインドに beginning-of-line と move-end-line というコマンドが割り当てられ実行されているのである。

Emacs のコマンドはプログラムの関数となっており、それがキーに割り当てられている。キーバインドには自分で好きなコマンドを割り当てることも可能なので、全てのコマンドをキーバインドから実行することも可能である。

コマンドの表記方法

本稿でコマンドを表記する場合は次のように表記する。

```
M-x コマンド名
```

M-x は Alt+x と同じ意味で、この後「キーバインドの表記方法」で詳しく解説する。尚、M-x は「メタエックス」もしくは「エムエックス」と読む。

コマンドに続けて更にミニバッファに入力を求められる場合は、

```
M-x コマンド名 RET 追加入力 RET
```

と表記する。RET は Enter もしくは Ctrl+m を意味する。

キーバインドの表記方法

本稿でキーバインドを表記する場合は表 3.1 のように表記する。

表 3.1: キーバインド早わかり表

キーの種類	名前	表記	キーの種類	名前	表記
装飾キー	Control	C-	特殊キー	←	<left>
	Meta	M-		→	<right>
	Shift	S-		PageUp	<prior>
	Super	s-		PageDown	<next>
文字・記号	a,1,?,···	a,1,?,···		Home	<home>
特殊キー	Tab	TAB		End	<end>
	Space	SPC		Backspace	<backspace>
	Escape	ESC		Delete	<del>
	Return	RET		Shift+TAB	<backtab>
	↑	<up>		F1,F2,···	<f1>,<f2>,···
	↓	<down>			

キーバインドの表記サンプル

キーバインドを説明する際、C-n や M-x、C-x f などの特殊な表記を用いるため、表 3.2 で説明しておく。

表 3.2: キーバインドの表記

キー	説明
C-n	Ctrl を押しながら n を押す。
M-f	Meta を押しながら f を押す。
C-RET	Ctrl を押しながら Enter を押す。
C-x k	Ctrl を押しながら x を押し、Ctrl を離してから k を押す。
C-x C-c	Ctrl を押しながら x を押し、そのまま c を押す。
C-M-S-v	Ctrl と Meta と Shift を押しながら v を押す。
C-/, C-_	Ctrl を押しながら /、もしくは Ctrl を押しながら _ を押す。

また、Emacs の解説では次のようにキーバインドの後にコマンド名を表記する場合がある。

```
C-x C-f (find-file)
```

これは C-x C-f と入力する他に M-x find-file RET としても利用可能である、という意味である。本稿でも、キーバインドと同じ処理を実現するコマンドが存在する場合にはこの形で表記する。

プレフィックスキー（起点キー）

C-x C-f の C-x などは単体ではコマンドが実行されず、その後にキー入力を必要とするキーバインドである。Emacs ではこのようなキーをプレフィックスキーと呼ぶ。プレフィックスキーには、それぞれキーバインド設計のための指針

が用意されているので、表 3.3 で解説しておく。

表 3.3: プレフィックスキーの指針

キー	説明
C-x	システムコマンドが利用する。
C-c	ユーザがキーバインドを定義する。
	拡張機能を作成する場合は C-c で始まるキーバインドは定義しない方がよい。
M-g	行移動に関するキーバインドが定義されている。

3.2 起動と終了

起動と終了を説明しなければならないのは些か時代遅れな気がするが、Emacs は様々な OS や環境で利用できるように一般のアプリケーションより少々方法が多くなっているので、ここで解説しておく。

3.2.1 起動する

最も簡単な Emacs の起動はアイコンをダブルクリックする方法である。Mac では Emacs.app のアイコン、バイナリインストールした Windows では runemacs.exe のアイコンをダブルクリックして起動する。

ターミナルからは emacs、もしくは emacs-27.1 のようにバージョン番号を付けて起動する。もし、X で起動する Emacs をターミナル内で起動したい場合は次のように -nw という引数を付加する。

```
$ emacs -nw もしくは --no-window-system
```

尚、Mac の Emacs.app もターミナル上で次のようにするとターミナル上で起動することができる。

```
$ /Applications/Emacs.app/Contents/MacOS/Emacs -nw
```

Windows でもコマンドプロンプトもしくは PowerShell から bin¥emacs.exe に -nw 引数を付加して起動することで、プロンプト上で Emacs を利用可能である。

Emacs デーモンで起動を高速化する

Emacs は引数を付加して起動することで様々な状態で起動することが可能である。主だった引数はターミナル上でで引数 --help を付けて実行することで確認することができるが、覚えておきたいのが Emaacs デーモンである。

Emacs デーモンは Emacs 23 から導入された機能で、Emacs をデーモンと呼ばれるバックグランドで動作するプログラムとして起動しておくことで、瞬時に Emacs を起動させることができる仕組みである。

デーモンとして起動するには、Emacs に --daemon という引数を付加して実行するだけである。但し、Windows プラットフォームでは Emacs デーモンはサポートされていない。

```
$ emacs --daemon
```

デーモンとして起動しておいた Emacs を利用するには、emacsclient というコマンドを用いる。

```
$ emacsclient -c もしくは --create-frame
```

-c は主に X で Emacs を利用するために用いる。ターミナル上で Emacs デーモンを利用する場合は次の通りである。

```
$ emacsclient -c もしくは -nw
```

尚、Emacs デーモンは普通に終了してもバックグランドで起動し続けている。本当に終了したい場合は、Emacs 上で M-x kill-emacs を実行するか、シェル上で次のコマンドを実行する。

```
$ emacsclient -e '(kill-emacs)'
```

常に Emacs デーモンを利用したいのであれば、シェルにエイリアスを設定しておくとよい。

また、OS の起動スクリプトを利用して OS に起動と同時に自動的に Emacs デーモンを起動する方法が「Emacs Wiki:

Emacs As Daemon」[*1]に紹介されている。

デバックモードで起動する

Emacs は第 4 章で解説する設定ファイル（init.el）を用意することで、起動時にユーザが記述した設定を読み込むようになる。その際、設定に記述ミスがあると途中でエラーが発生し、それ以降の読み込みを中断する。そのような場合、デバックモードで起動することでエラーの原因を調べることができる。

例えば、以下のような設定ファイルを読み込むようにしていたとする。

```
;; cl-lib パッケージを読み込む。
(require ‘cl-lib)
;; スタートアップメッセージを非表示にする。
(setq inhibit-startup-screen t)
p(when window-system  % ← 行頭に不要な p が入っている
  ;; tool-bar を非表示にする。
  (tool-bar-mode 0)
  ;; scroll-bar を非表示にする。
  (scroll-bar-mode 0))
```

すると、Emacs は起動時に以下のようなエラーを出力する。

```
Warning (initilaization): An error occured while loding ‘/Usrs/ユーザ名/.emacs.d/
init.el‘ :

Symbol’s value as variable is boid: p

To ensure normal operation, you should investigate and remove the cause of the
error in your initialization file. Start Emacs with fg4the ‘--debag-init’ option
to view a complete error backtrace.
```

この大まかな意味は「init.el ファイルを読み込み時にエラーが発生しました。p というシンボル変数は存在しません」である。そして最後に「--debag-init オプションを利用することで、完全なエラーのバックトレースを見ることができます」という説明が書かれている。実際に --debag-init を付加して起動してみると、Emacs 上で次のように表示される。

```
Debugger entered--Lisp error: (void-variable p)
  eval-buffer(#<buffer *load> nil "/Users/ユーザ名/.emacs.d/init.el" nil t) ;
Readnig at buffer position 92
  load-with-code-conversion("/Users/ユーザ名/.emacs.d/init.el" "/Users/ユーザ名/
.emacs.d/init.el/" t t)
  load("/Users/ユーザ名/.emacs.d/init" t t)
  #[0 "H\205\266^Q \306=/203^Q^Q\307^H\310Q\202?^Q
   （中略）
  command-line()
  normal-top-level()
```

注目してほしいのは「Reading at buffer position 92」の部分である。92 という数値は Emacs 内部の文字の場所（ポイントと呼ぶ）の数値である。従って、この場所では init.el ファイルを開いて `M-x goto-char RET 92 RET` といコマンドを実行すると p という文字にカーソルが移動し、ここに p という文字が紛れ込んでいるのを発見することができる。

3.2.2 終了する

Emacs を終了させるためには、通常 `C-c C-x` というキーバインドを利用する。まだ保存されていないファイルが存在する場合は、ミニバッファに、

[*1] http://www.emacswiki.org/emacs/EmacsAsDaeom

```
Save file /Users/ユーザ名/.emacs.d/init.el? (y, n, !, ., q, C-r, d or C-h)
```

という質問が表示される。この質問に対するそれぞれの指示は表 3.4 の通りである。

表 3.4: 未保存のファイルが存在する場合の対応

キー	説明
y	このファイルを保存する。
n	このファイルを保存しない。
!	全てを保存する（未保存のファイルが複数存在する場合）。
.	このファイルを保存して Emacs を終了する。
q	全てのファイルを保存しない。
C-r	ファイルを表示する（フレーム上に表示されていない場合）。
d	保存されているファイルと編集中のファイルの差分を表示する。
C-h	選択肢のヘルプを表示する。
C-g	操作をキャンセルする（何もしない）。

表 3.4 のキーを押すことで対応する操作が実行される。未保存のファイルが複数存在する場合は全てのファイルに対して同じ質問が繰り返される。保存を拒否したファイルが存在する場合、最後に次のように質問される。

```
Modified buffers exist anyway? (yes or no)
```

これは「変更がまだ残っていますが、本当に終了しますか？」という意味なので、yes RET とタイプすると変更を破棄して終了し、no RET で終了をキャンセルする。C-g でもキャンセル可能である。

3.3 ファイル（バッファ）を開く、保存する

Emacs はコマンドラインの時代から利用されているアプリケーションであるため、ファイルを開く、保存するなど非常に一般的な操作も通常はキーボードから操作する。また、ここで解説するファイルの開き方はあくまで基本となる操作であり、第 6 章で解説する Helm という拡張機能を利用することで簡単にファイルを開くことが可能となる。

3.3.1 ファイル（バッファ）を開く：C-x C-f

C-x C-f (find-file) というコマンドは、ファイルを開くための基本コマンドである。カレントバッファのディレクトリを起点とし、ミニバッファにファイル名を入力する。ミニバッファでは TAB による補完が利用できるため、慣れてくるとマウスでファイルを見つけてダブルクリックするよりも速くファイルを開くことができるようになる。

また、存在しないファイル名を入力するとバッファが作成され、保存時にディスクにファイルとして書き込まれる。

3.3.2 ファイル（バッファ）を保存する：C-x C-s

ファイルを開くという動作は、実際にはファイルを Emacs のバッファに読み込むという操作のことである。従って、編集したバッファは保存するまでは実際のファイルに反映されない。バッファを保存するには C-x C-s (save-buffer) というキーバインドを用いる。

尚、標準の設定では一番初めにファイルを保存する際、ファイルを開いた時の状態をバックアップファイル（後述）として保存する。この機能を利用することで、何度保存しても最初に開いた時の状態に復元することができる。

3.3.3 全てのファイル（バッファ）を保存する：C-x s

C-x s (save-some-buffers) というコマンドも用意されている。こちらは「Emacs の終了」で解説したものと同じで、Emacs 上で開いている全てのファイルに対して保存するか否かそれぞれ確認する。

3.3.4 バックアップファイル

Emacs が作成するバックアップファイルは、ファイル名の末尾に ~（チルダ）を付けた名前となっている。例えば「init.el」というファイルであれば「init.el~」というファイルがバックアップファイルである。

オートセーブファイル

前述のバックアップファイルは Emacs が終了しても残り続けるが、これ以外にも Emacs には編集中のファイルのバックアップを随時作成するオートセーブという仕組みが用意されている。これはアイドルタイム（Emacs を操作していない時間）を利用してファイル名の前と後ろに # マークが付いたオートセーブファイルを自動的に作成してくれる。尚、このオートセーブファイルはバッファをファイルに保存すると自動的に削除される。このオートセーブに関する詳しい設定については第 5 章で解説する。

3.3.5 別名で保存する：C-x C-w

ファイルを別名で保存するには `C-x C-w (write-file)` というキーバインドを用いる。コマンドを入力すると `C-x C-f (find-file)` と同じようにファイル名（パスも含めて）を聞かれるので、入力して RET とすると別名で保存される。

3.3.6 バッファに別ファイルを挿入する：C-x i

現在開いているバッファに別のファイルを挿入することも可能である。`C-x i (insert-file)` を実行すると、ミニバッファで「Insert file:」とファイル名を聞かれるので、ファイルを開く時と同様にファイル名を入力するだけで簡単にファイルを挿入することができる。また、`M-x insert-buffer` でバッファを挿入することも可能である。すなわち、既に Emacs で開いているファイルまたは Emacs が自動生成するバッファの内容を挿入することができる。

3.3.7 文字コード・改行コードを変換する：C-x RET f

最近は意識する必要が少なくなったとはいえ、テキストファイルを扱う上で忘れてならないのが文字コードと改行コードである。

Emacs はエディタの中でも最も多くの文字コードを扱うことができるエディタの 1 つである。基本的には自動判別して適切な文字コードでファイルを開いてくれる。尚、Emacs でファイルを作成した際の標準文字コードは Unicode（UTF-8）、改行コードは UNIX（LF）となっている。

現在編集中のバッファの文字コードを変更したい場合は、`C-x RET f (set-buffer-file-coding-system)` というキーバインドを用いる。実行するとミニバッファで「Coding system for saving file (default nil):」と問われるので、変更したい文字コードの名前（コーディングシステム名）を入力すると文字コードが変更される。例えば、sjis-dos とすると Shift-JIS で CR+LF、sjis であれば Shift-JIS で改行コードは変更しないという意味となる。

指定可能な文字コードを表 3.5、改行コードを表 3.6 に示す。

表 3.5: 文字コード

モードライン表記	文字コード名	Emacs 上の呼称
U	Unicode	utf-8、utf-16、utf-7 など
S	Shift_JIS	sjis（shift_jis）
J	JIS コード	iso-2022-jp など
E	日本語 EUC	euc-jp、euc-jis-2004 など
1	Lattin-1	latin-1（iso−8859−1）
M	emacs-mule	emacs-mule

表 3.6: 改行コード

モードライン表記	改行コード名	Emacs 上の呼称	説明
:	LF	unix	UNIX 系 OS で主に利用される。
(DOS)	CR+LF	dos	Windows で主に利用される。
(Mac)	CR	mac	Mac OS 9 まで利用されていた。

文字コードの選択でも TAB による補完と候補一覧を利用することができる。

3.3.8 文字コード・改行コードを変換して開き直す：C-x RET r

次は、途中変更ではなく文字コードを指定して開き直す方法である。`C-x RET r (revert-buffer-with-coding-system)` というキーバインドで、前述と同様に文字コードを問われるので入力するとファイル名を指定した文字コードで開き直してくれる。

3.3.9 バッファを切り替える：C-x b

今までも何度か登場したが、Emacs ではファイルを開くとバッファを作成し、消去しない限り Emacs 上に保持し続ける。これはブラウザで言うところのタブのようなもので、タブを切り替えるようにバッファを切り替えることで複数のファイルを 1 つの Emacs で編集することができる。

バッファを切り替えるには `C-x b (switch-to-buffer)` というキーバインドを用いる。するとミニバッファでバッファ名を問われるので入力して RET を押す。この時、勿論 TAB による補完が利用可能である。いちいちバッファ名を入力するのが煩わしい場合は、`C-x <right> (next-buffer)`、`C-x <left> (previous-buffer)` というキーバインドを利用するとよいだろう。これは Emacs 内で管理しているバッファリストに従って、バッファを順番に切り替えてくれる。バッファリストの確認は `C-x C-b (list-buffers)` というキーバインドで可能である。

3.3.10 バッファを消去する：C-x k

バッファを消去するには `C-x k (kill-buffer)` というキーバインドを用いる。すると消去するバッファを問われるので、そのまま RET するとカレントバッファが消去される。

もし、そのバッファがまだ保存されていない場合は「buffer バッファ名 modified; kill anyway? (yes or no)」と問われるので、yes と入力するとバッファを保存せずに消去、すなわち編集を破棄する。no または `C-g` でバッファの消去をキャンセルする。

3.4 カーソル移動

次はカーソル移動である。第 1 章でも述べたが、Emacs を使うメリットとしてキーボードで思い通りのカーソル移動ができるという点が挙げられる。その優れた操作性を身に付けることで、とても快適な編集を実現してくれる。

3.4.1 キーバインド一覧

カーソル移動については、Emacs チュートリアルで一通り学ぶことができるが、改めて主要なキーバインド一覧を表 3.7 にまとめておく。

表 3.7: カーソル移動系の主要キーバインド

キー	コマンド名	説明
`C-l`	`recenter-top-bottom`	カーソル位置を起点にウィンドウの表示をリフレッシュする。
`C-p`	`backward-line`	1 つ上の行に移動する。
`C-n`	`next-line`	1 つ下の行に移動する。

C-f	previous-line	1 文字前に移動する。
C-b	previous-line	1 文字後に移動する。
C-a	move-beginning-of-line	行頭に移動する。
C-e	move-end-of-line	行末に移動する。
C-v	scroll-up-command	1 画面下にスクロールする。
M-v	scroll-down-command	1 画面上にスクロールする。
C-M-v	scroll-other-window	ウィンドウ分割時に他のウィンドウに対して C-v を実行する。
C-M-S-v	scroll-other-window-down	ウィンドウ分割時に他のウィンドウに対して M-v を実行する。
M-<	beginning-of-buffer	バッファの先頭へ移動する。
M->	end-of-buffer	バッファの終端へ移動する。
M-g g	goto-line	ミニバッファで入力した行番号へ移動する。

3.5 文字の入力や文字列の操作

続いては入力に関する操作である。通常の文字タイピングに関しては特筆すべき事はないが、コピーやカット、ペーストなどの入力補助については大いに有用である。

3.5.1 マークとリージョン：C-SPC

コピーやカットを行う為には、まず範囲選択を行う。この範囲選択はメモ帳などのエディタでは Shift+↑ ↓ ← → を用いるのが一般的だが、Emacs では一風変わった範囲選択方法が用意されている。それがマークとリージョンという概念である。

マークは言葉の通りカーソル位置にマーキング（印付け）を行う操作である。キーバインドは C-SPC (mark-set-command) もしくは C-@ である。マークしてからカーソル移動すると、マークから現在位置までが選択範囲となる。これがリージョンである。Emacs はリージョンを用いてコピーやカットなどの一般的な操作から整形、変換など高度な処理を行うことができる。

3.5.2 コピーとカット：M-w、C-w

リージョンを学んだ上で、コピーとカットの方法を解説する。まずはコピーだが、リージョンによる範囲選択を行ってから M-w (kill-ring-save) というキーバインドを用いるとリージョンをコピーしてくれる。

コマンド名に kill-ring（キルリング）という言葉が入っているが、これは一般的にクリップボードと呼ばれるような機構で、消去（キル）したテキストを記録しておく場所となっている。

カットはコピーと同じくリージョンを作成してから C-w (kill-region) というキーバインドを用いる。コマンド名は kill-region となっているが、このキルは消去という意味であり、キルされたテキストは全てキルリングに記録される。尚、キルリングに記録しない文字の消去は削除（delete）と表現し、コマンド名も delete から始まるものが用いられる。

3.5.3 行を消去する：C-k

少し特殊な操作として行の消去がある。C-k (kill-line) はカーソル位置より右にあるテキストを行末まで（改行は含まない）を消去する。消去なのでテキストはキルリングに記録される。非常によく用いられる操作としては、行頭へ移動して行を消去する C-a C-k が挙げられる。

C-k はカーソルが改行に位置する場合、改行のみを消去する。また、連続して C-k を用いて消去した内容は 1 回のペーストで貼り付けることができるようになっている。

3.5.4 ペーストする：C-y、C-y M-y

コピーやカットなどで消去され、キルリングに記憶されたテキストは C-y (yank) を用いていつでもヤンク（ペースト）可能である。このヤンクは直前に消去された内容を貼り付けるコマンドである。

以前にキルリングに記録した内容を遡ってヤンクするには、C-y に続けて M-y (yank-pop) を入力する。M-y は直前のコマンドが C-y だった場合にのみ利用可能なコマンドで、キルリングの内容を遡ってヤンクする。M-y を続けて入力することでキルリングを遡り続けるのだが、インタフェースとしては少々使いづらいものであるため、第 6 章で紹介する helm-show-kill-ring という拡張機能を利用する方がよいかもしれない。

3.5.5 コメントする、コメントを解除する：M-;

コメントとはコードや設定ファイルにおいて処理に含めない部分のことであり、読む人間のためのメモ書きやコード処理させないようにする（コメントアウトする）などに利用する。Emacs には M-; (comment-dwin) という言語と状況によってコメントを挿入・解除してくれるコマンドが備わっている。

- リージョン選択中に M-; するとコメントアウト、もしくはコメント解除する。
- リージョン選択中に C-u 数値 M-; するとコメント文字列を数値分にする。
- 何も書かれていない行（空行）で M-; した場合、コメント文字列を挿入する。
- 何か書かれている行で M-; した場合、行末にコメント文字列を挿入する。
- コメントが存在する行で M-; した場合、コメント本文までジャンプする。
- コメント行で引数を与えて M-; した（例えば C-u M-;）場合、コメント行であれば削除する。

編集中のファイルが特有のコメント開始文字を持たない場合はミニバッファで「No comment syntax is defined. Use:」と問われるので、例えば # として RET すると # をコメント開始記号として利用してくれる。

3.5.6 特殊文字を入力する：C-q

テキストには、例えば改行文字やタブ文字など通常の文字とは異なる文字（制御文字など）が存在する。これらを入力する方法を覚えておくと、思わぬところで役立つ可能性がある。C-q (quoted-insert) は、次に入力したキーの制御文字を挿入してくれる。例えば、リターンではなく改行文字そのものを挿入したい場合は C-q C-j を実行する。同じくインデントではなくタブ文字を挿入したい場合は C-q TAB を実行し、Ctrl+c の制御文字を挿入したい場合は C-q C-c を実行する。この C-q による特殊文字の入力を覚えておくと、例えば一括変換で改行を消去したり、カンマ（,）を改行に変換したりすることなどが簡単に行えるようになる。

3.5.7 アンドゥ：C-/、C-_、C-x u

入力中に編集内容を 1 つ前の状態に戻したい場合はアンドゥを利用する。C-/ (undo) を実行すると、直前の変更を元の状態に戻すことが可能である。

アンドゥとは逆の操作を実現するリドゥは Emacs 標準には用意されておらず、C-g を実行してからアンドゥを行うことでリドゥとの機能を実現する。この操作に馴染めない場合は、第 6 章で紹介する拡張機能 undo-tree を導入するとよいだろう。

3.6 Emacs の正規表現

Emacs では他のエディタ同様、検索や置換などに正規表現を利用することができる。正規表現については本節ではあまり深く語らないが、Emacs の正規表現は文字列で記述するため、一般的な正規表現リテラルによる記述とは少々異なり注意が必要な部分が存在する。

3.6.1 特別な文字

特別な文字とは一般的にメタキャラクタと呼ばれるもので、普通の文字とは異なり特殊な意味を持つ文字のことである。表 3.8 にメタキャラクタを示す。

表 3.8: 正規表現で使うことのできる特別な文字（メタキャラクタ）

文字	説明	利用例
.	改行以外の任意の文字に一致する。	.macs（Emacs, imacs などに一致）
*	直前の正規表現を可能な限り反復する後置演算子。	E*macs（macs, Emacs, EEEmacs などに一致）
+	直前の正規表現に 1 回以上一致する。	E+macs（Emacs, EEmacs, EEEmacs などに一致）
?	直前の正規表現に 1 回以上一致するか、あるいは 1 回も一致しない。	E?macs（macs, Emacs に一致）
\{n}\	直前の正規表現がｎ回の場合のみ一致する後置演算子。	E\{2\}macs（EEmacs に一致）
\{n, m\}	直前の正規表現がｎ回からｍ回まで一致する後置演算子。	E\{1, 3\}macs（Emacs, EEmacs, EEEmacs に一致）
[...]	この間にある文字集合に一致する。	[Ee]macs（Emacs, emacs に一致）
[^...]	この間にある文字集合以外に一致する。	[^Ee]macs（Emacs, emacs 以外に一致）
^	行頭に一致する。	^Emacs（行頭の Emacs に一致）
$	行末に一致する。	Emacs$（行末の Emacs に一致）
\	特別な文字をクオート（エスケープ）する。\. は文字. に一致する。	\.emacs（.emacs に一致）
|	両端の正規表現のどちらかに一致する。通常はグループ化と組合わせて使用する。	a|b（a か b に一致する）
\(... \)	グループ化。囲まれた正規表現を 1 つの文字のように扱う。	\(Emacs|emacs\)（Emacs か emacs に一致）
\1,\2,...,\9	グループ化した文字を引用する。	\(Emacs\).*?\1（Emacs から同じ行に登場する次の Emacs まで一致）

3.7 検索と置換

検索と置換はエディタの中心機能の 1 つである。そもそも、検索は見つけるという操作以外にも移動したり思い出すなど様々な役割があり、OS を含む全てのアプリケーションにおいて検索を使いこなせるか否かで作業効率が大幅に変化する。

3.7.1 grep による検索

grep とは UNIX 系 OS における検索の代名詞とも言えるコマンドであり、指定されたファイルの中から正規表現を用いて検索して一致する行を一覧表示してくれるものである。Emacs には `M-x grep` というコマンドが用意されており、Emacs から直接 grep を実行することができる。`M-x grep RET` と入力すると「Run grep (like this): grep -nH -e」という内容がミニバッファに表示される。これはターミナルにおいて grep を実行する場合と同じ要領でコマンドを入力せよ、という要求である。検索対象となるディレクトリはカレントディレクトリとなる。

例えば、emacs という単語を全てのファイルから検索したい場合、`grep -nH -e "emacs" * RET` と入力する。すると *grep* というバッファが開かれ、通常の grep と同様に一致した行が一覧表示される。この一覧表からファイルを直接開くことが可能である。

通常の grep 以外にも対話式 grep コマンドである lgrep、再帰的に grep を行う rgrep というコマンドも用意されており、両方とも有用である。

3.7.2　インクリメンタル検索：C-s、C-r、C-M-s、C-M-r

C-s (isearch-forward) は Emacs で非常によく利用されるキーバインドの 1 つである。ミニバッファで「I-search:」と問われるので、文字を入力するとインクリメンタル検索が開始され、カレントバッファ上のカーソル以降にあるマッチ（一致）する文字の場所へカーソルがジャンプする。尚、C-r (isearch-backward) は C-s とは逆にカーソル以前のマッチする文字へジャンプする。

C-s や C-r は検索している際に連続で入力すると、次にマッチする場所までカーソルが移動する。これによって本文検索とカーソル移動を同時に実現することができる。

C-M-s (isearch-forward-regexp) は regexp という名前の付く通り isearch の正規表現版である。

3.7.3　対話置換、一括置換：M-%、C-M-%

文字列の置換は正しく利用することで確実にテキストを置き換えてくれる。Emacs には正規表現を利用するかどうか、置換時に確認するかどうかという異なる条件の置換コマンドが 4 つ用意されている。尚、置換は全てカーソル以降に対してのみ行われるようになっており、カーソル以前は対象外となる。

M-%(query-replace) は対話型の置換コマンドで、ミニバッファに検索する文字列、次に置換する文字列を入力することで置換が開始される。検索にマッチした文字列を見つけるとカーソルが移動し「Query replacing 検索文字列 with 置換文字列: (? for help)」と問われる。? を押すとヘルプが表示され、y で置換、n でスキップする。

C-M-%(query-replace-regexp) は同じく対話型置換コマンドの正規表現利用版である。M-x replace-string は対話無しで一括置換を行うコマンドであり、M-x replace-string-regexp は同じく一括置換の正規表現利用版である。

ナローイングを用いてバッファの一部のみを編集する

ナローイング（narrowing）は普段あまり利用する機会のあるものではないが、意図しないキー操作によって実行される場合がある為、記憶に留めておいた方がよい仕組みである。

範囲を制限するという意味であるナローイングは、Emacs では編集可能範囲を制限するという意味で用いられる。ある範囲だけを置換対象にしたい場合にも利用することができる。

リージョン選択時に C-x n n (narrow-to-region) というコマンドを実行すると、バッファからリージョンのみを残して他の部分が消え去る。

このコマンドは初心者にとって混乱の原因となる為、標準では利用できないようになっている。そのため、初めて実行した際に本当に実行するかどうかを確認する問いが表示される。

```
You have typed C-x n n, invoking disable command narrow-to-region.
It is disabled because new users often find it confusing.
Here's the first part of its description:

(中略)

You can now type
y   to try it and enable it (no question if you use it again).
n   to cancel--don't try the command, and it remains disabled.
SPC to try the command just this once, but leave it disabled.
!   to try it, and enable all disable commands for this session only.
```

そして、ミニバッファでは「Type y,n,! or SPC (the space bar):」という問いが表示され入力待ちとなる。y を押すと narrow-to-regin が実行され、 2 度と同じ質問がされなくなる。それぞの選択結果については表 3.9 にまとめる。

表 3.9: narrow-to-region の問いに対する選択結果

キー	説明
y	narrow-to-region を実行。(put 'narrow-to-region 'disable nil) を設定ファイルに追加し、2 度と質問しない。
n	実行をキャンセルする。次に実行する際は再度質問する。
SPC	今回は実行するが、次に実行する際は再度質問する。
!	実行する。Emacs を終了するまで質問しない。

narrow-to-region を実行すると、モードラインに「Narrow」と表示され、リージョン以外が消えてしまった様な表示となるが、編集範囲を制限（隠した）だけである。ナローイングを解除するワイドニングは C-x n w (widen) である。ワイドニングを実行すると再び全体が表示される。

3.8 ウィンドウ操作

Emacs を使っていると自動的にウィンドウ分割するコマンドが多々存在する為、本人が望まなくともウィンドウ操作を余儀なくされる。最初の内は少々ややこしいかもしれないが、実際にウィンドウ操作で覚えなければならないものは最低でも以下の 3 つだけである。

3.8.1 ウィンドウを分割する：C-x 2、C-x 3

ウィンドウを分割するコマンドは 2 つのみである。横に 2 分割する C-x 2(split-window-vertically) と、縦に分割する C-x 3 (split-window-horizontally) である。

vertically（垂直）と horizontally（水平）が、日本語の感覚としては逆であることが少々混乱を与えるが、横ではなく上下に分割するという意味合いから vertically と名付けられたのだろう。覚え方としては、漢字の「二」のように分割するのが C-x 2 であると捉えると記憶し易いかもしれない。

3.8.2 ウィンドウを移動する：C-x o

C-x o (other-window) は、ウィンドウが分割されている際に、カレントウィンドウを他のウィンドウへ移動するコマンドである。

3.8.3 分割したウィンドウを閉じる：C-x 1、C-x 0

ウィンドウを閉じるコマンドは 2 つ用意されている。

1 つ目は C-x 1 (delete-other-window) で、実行するとカレントウィンドウ以外の全てのウィンドウを閉じる。

2 つ目は C-x 0 (delete-window) で、実行するとカレントウィンドウのみを閉じる。状況に応じて使い分けると良いだろう。因みに、ウィンドウを閉じてもバッファが削除されることはない。

3.9 ディレクトリ操作（Dired）

Emacs はテキストエディタだが、標準でディレクトリ操作が可能である。

それが Dired という機能である。C-x d (dired) から C-x C-f と同じように開きたいディレクトリを指定して RET する。すると、Dired というディレクトリエディタの為のバッファが開く。

Dired ではディレクトリやファイルのコピー、リネーム、作成、パーミッション変更など、標準的なファイラの機能は全て利用可能である。基本操作を表 3.10 にまとめる。

表 3.10: Dired 上での代表的なコマンド一覧

キー	説明	キー	説明
n、SPC	次の行へ移動する。	u	現在行のマークを外す。
p	前の行へ移動する。	*!	マークを全て外す。
RET、f	現在行のファイルを開く。	+	ディレクトリを作成する。
d	削除候補としてマークする。	C_	操作を 1 つ戻す。
x	マークしたファイルを削除する。	D	指定したファイルを削除する。
m	マークする。	R	指定したファイルの名前を変更する。
*%	正規表現でマークする。	C	指定したファイルをコピーする。
<backspace>	1 行上のマークを外す。	q	ウィンドウを閉じる。

3.9.1 ファイル名の一括変更：wdired-change-to-wdired-mode

Dired を利用してバッファに表示されているファイル名をテキストファイルを編集するように一括変換する機能が備わっている。M-x wdired-to-change-wdired-mode というコマンドを実行すると、Dired 画面に表示されているディレクトリ名を含む全てのファイル名がテキストとして編集可能となる。すなわち、第 7 節の「検索と置換」や第 6 章で解説する「矩形編集」などの Emacs の持つエディタとしての編集機能を活用することでファイル名を一括変換することができる。

3.10 キーボードマクロによる繰り返し操作

キーボードマクロは Excel などのマクロの様に、使用頻度の高い操作を記録して再利用可能とする機能である。

3.10.1 基本的な使い方

キーボードマクロの基本的な利用方法は C-x ((start-kbd-macro) を実行し、繰り返したい操作を行う。操作が終わってから C-x) (end-kbd-macro) でその操作が記録される。すると、C-x e (call-last-kbd-macro) で先程記録したキーボードマクロを呼び出すことができる。もし、10 回実行したければ C-u 10 C-x e の様に前置引数を付加して実行する。

例えば、行頭に移動して-と入力して次の行に移動するというマクロを作成したければ、C-x (を実行した後に C-a-C-n とし、最後に C-x) を実行することになる。この状態で C-e を実行すると行頭に「-」が挿入される。

3.10.2 名前をつける

M-x name-last-kbd-macro というコマンドを用いて、直近に記録したキーボードマクロに名前を付けることができる。M-x name-last-kbd-macro RET insert- RET と入力すると M-x insert- というコマンドが用意され、別のキーボードマクロを作成しても利用し続けることが可能となる。しかし、名前を付けても設定ファイルに保存しなければ Emacs を終了すると記録は消されてしまう。

3.10.3 再利用するために保存する

定義したキーボードマクロを Emacs を終了しても利用可能とするには、設定ファイルに保存する必要がある。名前を付けた後に設定ファイルを開いて、M-x insert-kbd-macro RET insert- RET と入力する。すると、次の S 式が挿入される。

```
(fset 'insert-
   "\C-a- \C-n")
```

この式を読み込むことで M-x insert- というコマンドを常に利用することができる。

3.11　表示の変更

本節では、少々毛色の変わった表示に関する操作を解説する。

3.11.1　文字サイズをすぐに変更する：C-x C-+、C-x C-=、C-x C--、C-x C-0

C-x C-+ もしくは C-x C-= は文字サイズを大きくするするコマンドである。逆に、小さくするコマンドが C-x C-- である。C-x の後のキーを連続でタイプすることで段階的にサイズ変更することが可能である。そして、C-x C-0 で元のサイズに戻すことができる。

これらは M-x text-scale-adjust というコマンドを利用しており、起点となるサイズを 0 として +1 または -1 と表示サイズレベルを変化させる（モードラインに現在のサイズレベルが表示される）。尚、このレベルは text-scale-mode-step という変数の値（初期値は 1.2）を現在の文字の高さに乗算したものとなっている。

C-u 5 M-x text-scale-adjust というコマンドで直接サイズレベルを指定することも可能であり、この場合は +5 のレベルを指定しているので 1.2 の 5 乗となり、デフォルトのサイズが 12pt である場合には約 29pt となる。

3.11.2　行の折り返し表示を変更する：M-x toggle-truncate-lines

テキストの 1 行当たりの文字数に制限はないが、当然ながら Emacs の表示幅は有限である。その為、1 行が表示幅を超える様な場合には折り返す／折り返さないという 2 つの表示方法を選択することが可能である。因みに、標準では折り返す設定になっている。

行の折り返し設定を切り替えるには M-x toggle-truncate-line というコマンドを用いる。因みに、toggle というのはオンオフを両方兼ね備えたスイッチという意味であり、toggle- から始まるコマンドは 1 つのコマンドによってオン／オフを切り替えることができる。

3.12　ヘルプの利用

Emacs は「Emacs is an extensible self-documenting editor」[*2]と称されることもあるくらい、ドキュメントが充実している（但し、英文である）。これには Elisp 自体に説明文を埋め込めるようになっていたり、またそれらを検索する仕組みがあったり、更にはコミュニティの協力があったりなど様々な要因がある。

総じて重要なことは「Emacs で分からない事があれば Emacs から直接教えてもらう事ができる」ということである。

3.12.1　info：M-x info

info とは説明書の事である。ターミナルからも info コマンドを利用することができるが、Emacs からも利用可能である。

3.12.2　ヘルプコマンド：C-h、<f1>

Emacs を自分好みにカスタマイズできるようになるためには、Emacs の機能や関数を自分で調べることができるようになる必要がある。標準のキーバインドであれば C-h はヘルプコマンドを呼び出すための接頭辞キーとなっている。C-h に続いてキーをタイプすることで、Emacs の様々な情報を調べることができる。C-h C-h を実行すると、どういったヘルプコマンドが用意されているのかを確認することができる。

3.12.3　よく利用されるヘルプコマンド

よく利用されるヘルプコマンドを紹介しておく。尚、where-is と describe-function と describe-variable については、実行した時のカーソル位置の文字を自動的に拾ってくれる。その場合はコマンド名などを入力せずに RET でヘルプを参

[*2] http://www.emacswiki.org/emacs/SelfDocumentation

照することができる。

C-h a 文字列 RET
入力した文字列が含まれるコマンドのリストを表示する。

C-h b (M-x describe-bindings)
現在の割り当てキー表を表示する。

C-h k キーバインド
キーバインドが実行するコマンド（関数）とそのドキュメントを表示する。

C-h w コマンド名 RET (M-x where-is)
入力したコマンドを実行するキーを表示する。

C-h f 関数名 RET (M-x describe-function)
入力した関数の説明を表示する。

C-h v 変数名 RET (M-x describe-variable)
入力した変数の説明を表示する。

3.12.4 日本語ドキュメント

Emacs 本体に同梱されているドキュメントは全て英語で書かれている。但し、バージョンは古いが過去に様々な人達によって翻訳されたドキュメントが Web 上に存在し「EmacsWiki: Emacs Lisp リファレンス」[*3]にまとめられている。

[*3] EmacsWiki: Emacs Lisp リファレンス：https://www.emacswiki.org/emacs?interface=ja

第 4 章

設定ファイルの管理方法

4.1　効率的な設定ファイルの作成方法と管理方法

Emacs 操作の基礎を一通り学んだところで、次は Emacs の最も魅力的な部分であるカスタマイズに進みたい。本章では Emacs の設定を学ぶ前段階として、設定ファイルの管理方法を解説していく。

Emacs を本格的に使用していくと様々な設定を追加することになる。自分だけの Emacs を築くための設定ファイルはかけがえのない資産である。もし、何かの拍子に設定ファイルを失うような事があれば、きっと大きなショックを受けることになるだろう。

本章では、その様な事にならないように次の項目を中心に、より良い設定ファイルの管理方法を解説していく。

- バックアップし易い管理体制の構築
- 異なる環境下でも共通の設定ファイルを利用する方法
- 設定ミスによる起動エラーの回避

4.1.1　~/.emacs.d ディレクトリに設定をまとめて管理する

設定ファイルやインストールした拡張機能ファイルなどが様々な場所に分散していると、全体の把握や管理が困難になってしまう。そこで Emacs の設定にまつわる全てのファイルを効率良く管理すべきである。

第 2 章では、Emacs の設定ファイルは「.emacs」「.emacs.el」「init.el」の 3 種類が存在することを説明した上で、最後の init.el の利用を推奨したが、その理由は.emacs.d ディレクトリに Emacs の設定や拡張機能を集約するためである。そうすることで.emacs.d ディレクトリ 1 つをバックアップするだけで全ての機能を保護し、また別のマシンにコピーするだけで同じ Emacs 環境が復元可能となる。

サブディレクトリの構成

基本方針として.emacs.d ディレクトリ以下に必要なサブディレクトリを作成していく。拡張機能によってファイルを追加する必要がある場合は.emacs.d ディレクトリ直下もしくはサブディレクトリを追加することになる。

最近の拡張機能の多くでは、標準で.emacs.d 以下にファイルまたはディレクトリを作成して管理してくれる。

Elisp をどのように配置すべきか

Emacs は Elisp を読み込むことで様々な拡張機能を追加することができるエディタである。この Elisp はプログラミングコードが記述された el という拡張子の（スクリプト）ファイルに過ぎない。

Elisp をインストール（ロードパスに追加済みのディレクトリにファイルを配置する）する場所は、UNIX 系 OS で標準的にインストールされた場合は、

- /usr/local/share/emacs/site-lisp
- /use/local/share/emacs/27.1/site-lisp

などのディレクトリが用意されている。この中に配置された Elisp を読み込むことが可能となるのだが、このディレクトリへのインストールは良い点とあまり良くない点が混在しているため、一考の余地がある。

良い点

・全ユーザに対してがインストールされた拡張機能が利用可能となる。
・サブディレクトリ以下も自動的に追加してくれる。
・バージョン別ディレクトリが存在するため、拡張機能が要求するバージョンを切り分けることができる。

あまり良くない点

・OS やインストール方法によって異なるディレクトリ配置となる可能性がある。
・バックアップを取るのが少々面倒である。

拡張機能をインストールするディレクトリは、設定によって別途ユーザが任意で追加することが可能であり、上記の良い点も後述する分割管理と設定の分岐によって改善することができることから、本稿では.emacs.d 以下にインストールすることを推奨する。

基本方針で示した構造に従う場合は ~/.emacs.d/elisp ディレクトリに Elisp をインストールしていくことになる。複数のファイルが存在する大きめの拡張機能の場合は、elisp ディレクトリ中にサブディレクトリを作成し、それも自動的に読み込むようにすると綺麗に整理することができる。また、Git や Subversion などのバージョン管理システムを利用してリポジトリをチェックアウトしてインストールする場合は ~/.emacs.d/public_repos 以下へチェックアウトするようにする。これはリポジトリからチェックアウトしている拡張機能を判別し易くするためである。

Elisp 配置用のディレクトリを作成する

前項の基本方針に従った環境を構築するために、まずディレクトリを作成しておく。ターミナルからは次のようにしてディレクトリを作成する。

```
$ cd ~/.emacs.d
$ mkdir elisp public_repos conf elpa
```

本稿では、これらのディレクトリを作成し、これから解説する設定を記述していく前提で解説を進める。ディレクトリ名や場所を変更する場合は適宜読み替えること。

ロードパスを追加する

続いて、ロードパスという拡張機能や設定ファイルを検索するためのディレクトリを設定する。ロードパスは load-path 変数にリストとして登録されており、任意のディレクトリを追加する場合はこの変数に追加する。

```
;; ~/.emacs.d/elisp ディレクトリをロードパスに追加
;; 但し、add-to-load-path 関数を作成した場合は不要
(add-to-list 'load-path "~/.emacs.d/elisp")
```

基本的には、このようにしてディレクトリをロードパスへ追加していくが、この方法ではサブディレクトリは自動的に追加されない。サブディレクトリも毎回 add-to-list 関数を用いてロードパスへ追加するのは少々面倒なので、次の関数を用いて簡略化する。

```
;; ロードパスをサブディレクトリも含めて追加する関数の定義
(defun add-to-load-path (&rest paths)
  (let (path)
    (dolist (path paths paths)
      (let ((default-directory
              (expand-file-name (concat user-emacs-directory path))))
        (add-to-list 'load-path default-directory)
        (if (fboundp 'normal-top-level-add-subdirs-to-load-path)
            (normal-top-level-add-subdirs-to-load-path))))))
;; 引数のディレクトリとそのサブディレクトリをロードパスに追加
(add-to-load-path "elisp" "conf" "public_repos")
```

この新しく作成した add-to-load-path 関数を利用すると.emacs.d ディレクトリ直下のディレクトリ名を渡すだけで、サブディレクトリが存在すれば自動的にロードパスへ追加してくれるようになる。本稿では、この add-to-load-path 関数

によってロードパスを追加する設定を行ったという前提で以後解説を進める。また、この設定は init.el のできるだけ最初の部分に記述しておくと良い。パスの設定については、第 5 章「パスの設定」において更に詳しく解説する。

設定を反映する方法

上述の設定を反映するには、次のいずれかを行う。

① Emacs を再起動する。
② この S 式の末尾で `C-x C-e` を実行する。

② を行うとコードが評価（Lisp ではコードを実行することを評価すると言う）され、設定を即時に反映することができる。この評価については第 5 章「設定を反映する方法」で詳しく解説する。

4.1.2 設定を分割して管理する

Emacs は何でも設定することが可能であるため、使い込めば使い込むほど設定ファイルの行数が膨らんでしまう。ある程度 Emacs の設定に慣れてきて行数が増えてきた時、init.el に全て記述するのではなく、環境変数に関する設定や各種プログラミング言語のための設定などをファイル単位で分割してジャンル毎に管理したいと思うかもしれない。本節では、そんな設定の分割管理について解説していく。

但し、設定を分割すると読み込みの順番によってはエラーが生じる可能性があるため、本稿における解説は分割せずに全て init.el に設定を記述している前提で進める。

ファイルを分ける

設定ファイルも拡張機能と同じ Elisp ファイルなので、ファイルに書き出して読み込むだけで簡単に分割することができる。前節で ~/.emacs.d/conf というディレクトリを作成してロードパスに追加したが、この conf というディレクトリに分割した設定ファイルを追加し、init.el から読み込むようにする。例えば、conf 以下に init-perl.el というファイルが存在し、これを読み込みたい場合は init.el に次のように記述するだけでよい。

```
(load "init-perl") ; 拡張子は不要
```

また、自身で分割するのではなく Emacs が自動的に init.el に書き込む設定を別のファイルに保存させ、それを init.el から読み込むことも可能である。

```
;; カスタムファイルを別ファイルにする
(setq custom-file (locate-user-emacs-file "custom.el"))
;; カスタムファイルが存在しない場合は作成する
(unless (file-exists-p custom-file)
  (write-region "" nil custom-file))
;; カスタムファイルを読み込む
(load custom-file)
```

この設定を利用すると第 6 章で解説する ELPA やテーマを利用した際に自動的に追加される設定が ~/.emacs.d/custom.el ファイルに書き込まれるようになる。

init-loader.el を利用する

init-loader.el[*1]は、IMAKADO 氏が作成した分割した設定ファイルを自動的に読み込むための拡張機能である。

これを利用するには init-loader.el ファイルをダウンロードして ~/.emacs.d/elisp ディレクトリにインストールすればよい。init-loader.el が設定を読み込むディレクトリを指定する必要があるため、次の設定を init.el に追記する。

```
(require 'init-loader)
(init-loader-load "~/.emacs.d/conf") ; 設定ファイルがあるディレクトリを指定する
```

これにより、conf ディレクトリ以下に存在する設定ファイルが init-loader の次の規則に従って読み込まれるようになる。

[*1] https://github.com/emacs-jp/init-loader

① 2 桁の数字から始まる設定ファイルを数字の順番に読み込む。（例：00_eval.el → 01_perl.el）
② Meadow の場合「meadow」から始まる名前の設定ファイルを読み込む。（例：meadow-emacs-config.el）
③ CarbonEmacs の場合「carbon-emacs」から始まる名前の設定ファイルを読み込む。（例：carbon-emacs-config.el）
④ CocoaEmacs の場合「cocoa-emacs」から始まる名前の設定ファイルを読み込む。（例：cocoa-emacs-config.el）
⑤ ターミナルの場合「nw」から始まる名前の設定ファイルを読み込む。（例：nw-config.el）

init-loader.el には分割された設定ファイルの記述に何らかのエラーが含まれている場合でも、そこで設定ファイルの読み込みを中断せずにスキップし、次のファイルの読み込みを進行するというメリットがある。読み込み状況は *Messages* という Emacs のログが蓄積されるバッファへと記録されるので、エラーが発生している場合はそこから確認する。

4.2 環境に応じた設定の分岐

前述した init-loader.el にも Meadow や CarbonEmacs などの OS 固有の Emacs に対してのみ設定を読み込ませる仕組みが存在したが、どういった仕組みで分岐しているのかを解説しておく。

Emacs には自分がどの様な環境で利用されているのかを識別するための変数が用意されている。この値を利用することで OS やターミナルか GUI かという環境に応じた柔軟な分岐を行えるようになっている。

4.2.1 OS の違いによる分岐

Emacs がどの OS 上で動作しているかどうかを知るには system-type という変数の値を調べればよい。

- Windows の場合：windows-nt あるいは cygwin
- Mac の場合：darwin
- Linux の場合：gnu/linux

もし、Mac だけに読み込ませたい設定がある場合は、次のように記述することになる。

```
(when (eq system-type 'darwin)
  (require 'ucs-normalize)
  (setq file-name-coding-system 'utf-8-hfs)
  (setq local-coding-system 'utf-8-hfs))
```

4.2.2 CLI と GUI による分岐

Windows や Mac などの GUI が当たり前となった OS を利用している者には、ウィンドウシステムという言葉に馴染みがないかもしれない。ウィンドウシステムとは GUI 操作を実現するための画面を描画するシステム、またはそのソフトウェアの事を指す。

window-system という変数に Emacs が起動しているウィンドウシステム名がセットされている。

- Windows の場合：w32 あるいは pc
- Mac の場合：ns
- Linux の場合：x
- ターミナルの場合：nil

これを利用して、次のような設定を行うことができる。

```
;; ターミナル以外はツールバー、スクロールバーを非表示にする
(when window-system
  (tool-bar-mode 0)
  (scroll-bar-mode 0))
;; CocoaEmacs 以外はメニューバーを非表示にする
(unless (eq windowsystem 'ns)
  (menu-bar-mode 0))
```

4.2.3　Emacs のバージョンによる分岐

昔から Emacs を利用している者にとっては、Emacs のバージョンも非常に重要な情報である。Emacs がメジャーバージョンアップする際、内部で使用されている関数が変更される場合がある。

そういった内部変更によって、メンテナンスが追い付いていない拡張機能は最新バージョンの Emacs でうまく動作しない場合がある。また、自身で書いた設定が特定のバージョンに依存するケースもあるだろう。特定のバージョンのみに設定を反映させるために、emacs-version もしくは emacs-major-version 変数の値を利用して設定を分岐させることができる。両者の違いは emacs-version が完全なバージョン番号を格納する変数であるのに対して、emacs-major-version はメジャーバージョン番号のみを格納する変数であることである。

```
;; Emacs 23 より以前のバージョンを利用している場合、
;; user-emacs-directory 変数が未定義であるため、次の設定を追加する
(when (< emacs-major-version 23)
  (defvar user-emacs-directory "~/.emacs.d/"))
```

4.3　拡張機能の読み込み方

ディレクトリにインストールされた拡張機能は、Emacs に読み込むように設定ファイルに記述して初めて利用可能となる。逆に言えば、読み込みを行わない限り、拡張機能をインストールしても機能を利用することはできない。

4.3.1　require と autoload の違い

拡張機能の読み込み方式には、最初から全て読み込む方式とコマンドを登録だけしておき実行時に読み込む方式の 2 種類が用意されており、拡張機能によって使い分けられる。これらは前者が require 方式、後者が autoload 方式と呼ばれ、本項ではそれぞれの違いについて解説する。

require

require 関数は最も標準的な拡張機能の読み込み方式で、もしインストールした拡張機能が require を用いて読み込めるのであれば、通常はこれを用いて読み込む。

```
(require 機能名 ファイル名 エラーの制御)
```

拡張機能側で用意されている「機能（FEATURE）」を用いて読み込む。一度読み込んだ拡張機能は別の場所で require されても繰り返し読み込まれることはない。拡張機能で機能名が用意されていない場合は require で読み込むことができない可能性があるが、実際に機能名が用意されていない拡張機能はほとんど存在しない。尚、第 2 引数の「ファイル名」以下の引数は省略可能となっている。

例えば、cl-lib という Emacs 上で Common Lisp というプログラミング言語の関数を利用可能とするための拡張機能を読み込む場合は、次のようにする。

```
;; cl-lib パッケージを読み込む
(require 'cl-lib)
```

require は読み込みに失敗するとエラーを出力し、それ以降の読み込みを停止するが、第 3 引数の「エラー制御」に対して non-nil（nil ではない）な値を渡すと、require 時にエラーが発生しても処理を停止せず、nil を返すだけとなる（読み込みに成功した場合は機能名を返す）。これを利用して読み込みに成功した場合のみ、その拡張機能を有効化する設定などを作成することが可能となる。

```
;; php-mode を読み込む
(when (require 'php-mode nil t)
  ;; 読み込みに成功した場合のみ、拡張子 ctp を php-mode で実行する
  (add-to-list 'auto-mode-alist '("\\.ctp$" . php-mode)))
```

php-mode は Emacs 25 現在、標準で同梱されていないため、各自でインストールする必要がある。インストール方法

などの詳細は、第 7 章で解説する。尚、require は拡張機能の使用の有無に関わらず全てを読み込む。そのため、読み込む拡張機能が大量である場合 Emacs の起動時間が長くなり、メモリの使用量も増加するという欠点がある。

autoload

autoload は require とは異なり、登録した関数（コマンド）を実行した際に指定した拡張機能ファイルを読み込む仕組みである。起動時に全てを読み込まないため、require に比べて起動時間が早くなりメモリも節約することができる。しかし、拡張機能ファイル内で使用されている変数もコマンド実行時に定義されるため、幾つかの面で注意が必要となる。

autoload の書式は次の通りである。

```
(autoload 関数名 ファイル名 説明文 対話判定 関数の種類)
```

autoload は関数名とファイル名（拡張子は不要）を記述して利用する。require では機能名を指定したが、autoload ではファイル名を指定することに注意する。また、対話判定に non-nil な値を与えるとコマンドとして実行可能（すなわち M-x から実行できる）となり、関数の種類の値を keymap もしくは macro と指定することで、それらをロードすることも可能である。尚、説明文以下の引数は省略可能となっている。

autoload を利用した読み込みの設定例として、第 7 章で解説する Ruby 用の拡張機能を読み込むために、次のように設定ファイルに記述している。

```
;; run-ruby 関数の初呼び出し時に inf-ruby.el を読み込む
(autoload 'run-ruby "inf-ruby"
  "Run an inferior Puby process")
```

4.3.2 コマンドが存在する場合のみ読み込む

基本的に自分自身が利用しているメイン環境であれば、外部コマンドの有無も把握済みで、足りないコマンドがあれば直ちにインストールすることができる。しかし、メイン以外の環境で Emacs を利用する場合、コマンドのインストールができないなどの理由で同じ環境を構築することが難しいこともある。

そのような場合、少々気を付けて設定ファイルを記述することで、インストールされていないコマンドに依存する機能に関する設定のみを無視して、それ以外はメイン環境と同じ状態の Emacs の設定を利用可能である。

コマンドがインストールされているかどうかを確認するには excutable-find 関数を用いる。もし、引数として渡されたコマンドを Emacs から見つけることができればコマンドのパスを返し、見つからなければ nil を返す。これにより、今まで通り when と組み合わせることで、コマンドが見つかった場合のみ評価される設定を作成することができる。

```
(when (excutable-find "git")
  (require 'magit nil t))
```

この設定は git コマンドが見つかった場合のみ Magit（Git を Emacs から使い易くするための拡張機能）を読み込む。Magit の導入、利用方法については第 7 章「Git フロントエンド」で詳しく解説する。

4.4 Web サービスを用いたバックアップ

本節では Web サービスを用いたバックアップ方法を紹介する。

既に、本章を読み終えて ~/.emacs.d ディレクトリに設定ファイルや拡張機能ファイルを集約する方法を覚えたなら、この 1 つのディレクトリを定期的にバックアップすることで Emacs に関する設定をバックアップすることができるようになっている。

但し、ローカルのみにバックアップを配置しておくとハードウェアごと故障した場合は対処しきれない。近年では便利な Web サービスが存在するため、サーバを持たない者にも簡単に Web 上にバックアップを配置することができるようになっている。

4.4.1 GitHub

Web サービスを用いたバックアップの内、 1 つ目は GitHub[*2] や Bitbucket[*3] などのコードホスティングサービスを利用したバックアップが挙げられる。GitHub では様々な者が dotfiles というプロジェクト名で設定ファイルを公開している。一度ホスティングサービスへプッシュ（送信）した設定ファイルは、チェックアウトすることで容易に他の環境へコピーすることも可能となる。

GitHub では非公開リポジトリの作成は有用プランとなっているが、Bitbucket では無料で非公開リポジトリを作成することが可能なので、無料で非公開にしておきたい場合は BitBucket を利用するとよいだろう。

~/projects/dotfiles ディレクトリに設定ファイルを移動する

GitHub は勿論、Git や BitBucket は Mercurial か Git をバージョン管理システムとして利用可能である。ここでは、両者で利用することができる Git を用いて解説する。

Git のインストールについては、Mac の場合は最新の Command Line Tools をインストールするだけで利用可能となる。UNIX 系 OS の場合は各種パッケージ管理ツール、もしくはソースコードからインストールする。Windows の場合は Git for Windows[*4] をインストールする。

著者の環境を例に挙げながら解説すると、まずホームディレクトリに ~/projects/dotfiles というディレクトリを作成し、その中に.emacs.d ディレクトリを移動する。これはホームディレクトリを Git で管理するわけではないためである。dotfiles ディレクトリの中のファイルを Git で管理した上で、シンボリックリンクを利用してホームディレクトリに.emacs.d を配置する。この方法を利用すると.emacs.d 以外の設定ファイル（例えば.bashrc など）も 1 つのリポジトリで管理することが可能となる。

```
$ mkdir -p ~/projects/dotfiles
$ cd ~/projects/dotfiles
$ mv ~/.emacs.d ./
```

シンボリックリンクを利用する

dotfiles ディレクトリへ移動した設定ファイルは、シンボリックリンクを利用することで今まで通り利用可能となる。シンボリックリンクは UNIX 系 OS で利用することが可能（Windows でも Vista 以降では利用可能）な、ファイルやディレクトリを別の場所から参照することができる仕組みである。シンボリックリンクは ln -s source_file target_file というコマンド（Windows の場合は mklink コマンド）で作成することができるので、次のようにして作成する。

```
Mac、Linux の場合：
$ cd ~/
$ ln -s /Users/ユーザ名/projects/dotfiles/.emacs.d .emacs.d
Windows の場合：
> %HOME%\mklink /D .emacs.d %HOME%\projects\dotfiles\.emacs.d
```

Git にコミットする

ここまで準備してから Git による管理を開始する。dotfiles ディレクトリでリポジトリを初期化し、ファイルをリポジトリへと追加しコミットする。

```
$ cd ~/projects/dotfiles
$ git init
$ git add .
$ git commit -m "First commit"
```

これで Git による dotfiles の管理が開始される。全てのファイルを追加したくない場合は、ファイルを追加（git add）する前に dotfiles 直下に.gitignore ファイルを作成し、除外したいファイルを指定する。

[*2] https://github.com
[*3] https://bitbucket.org
[*4] https://git-for-windows.github.io

```
;; Git で管理しないファイル
.*~
*~
/.emacs.d/*
;; Git で管理したいファイル（行頭に ! を付加する）
!/.emacs.d/init.el
!/.emacs.d/conf
```

GitHub へプッシュする

Git による管理を始めた後は GitHub でリポジトリを作成し、そのリポジトリをリモート先として登録してプッシュすることで GitHub によるバックアップを開始することができる。

```
GitHub で作成したリポジトリをリモートとして登録する。
$ git remote add origin git@github.com:ユーザ名/dotfiles.git
リモートリポジトリにファイルをプッシュする。
$ git push -u origin master
```

因みに、GitHub Desktop[*5] という GUI から簡単に Git を利用することができ、GitHub にもプッシュすることができるツールも存在する。他にも GUI で Git を扱うことができるソフトウェアは幾つか存在する[*6]ので、Git の使い方がまだよく分からないという者は試してみるとよいだろう。

4.4.2 Dropbox

もっと簡単にバックアップを取りたい場合は、オンラインストレージサービスの Dropbox[*7] を利用するのも 1 つの手である。Dropbox はインストールすると自動的に Windows であればドキュメントフォルダ内の My Dropbox 以下を、Mac であればユーザディレクトリ内の Dropbox 以下をオンライン上に同期するサービスである。

利用方法としては、前節で解説した GitHub を利用する方法とほぼ同じであり、Dropbox で管理されているディレクトリに設定ファイルを移動し、ホームディレクトリにはそのシンボリックリンクを張るだけである。

勿論、そのディレクトリを Git で管理することも可能なので、Dropbox による自動バックアップと Git によるバージョン管理という二段構えの体制でバックアップをることも可能である。

[*5] https://desktop.github.com
[*6] https://git-scm.com/downloads/guis
[*7] https://www.dropbox.com

第 5 章

本体の設定

5.1　設定を反映する方法

本章からは Emacs の真骨頂とも言える設定について本格的に解説していく。これまで何度も述べてきたが、Emacs は非常にパワフルでハッカーフレンドリーなツールである。その理由は、全て設定の柔軟性に集約されている。

Emacs の設定は init.el に Elisp を記述することで行っていくが、ファイルに設定を書き込むだけで即座に設定が反映されるわけではない。Emacs への設定反映は、変数の値を変更したり関数を実行したりすることで行われるが、その方法は大きく分けて 2 通り用意されている。

1 つ目は init.el を保存して Emacs を再起動する方法である。これは確実だが、少しの設定変更で毎回再起動するのはさすがに面倒である。そこで、主に 2 つ目の方法を利用することになる。それが評価（*eval, evaluation*）である。

5.1.1　C-x C-e と C-j による評価

必ず覚えておきたい評価方法は `eval-last-sexp (C-x C-e)` である。sexp とは S-expression の略で、日本語では S 式と呼ばれる。S 式 `(function arg)` の閉じ括弧の後ろにカーソルを移動した状態で `C-x C-e` を実行することで、その S 式を評価することができる。

Emacs では `C-x C-e` を用いることで、いつでもどこでも S 式を評価して Elisp を実行する（戻り値がある場合はエコーエリアに表示される）ことができる。これが Emacs における設定の反映方法である。これから紹介する設定を直ちに反映させたい場合は、init.el へ記述した後、その S 式の後ろで `C-x C-e` を実行すればよい。再起動することなく設定が Emacs に反映される。

また、*scratch* バッファでは `C-x C-e` の代わりに `C-j` によって S 式を評価することができ、実行すると戻り値が次の行に出力される。

5.1.2　その他の評価

`eval-last-sexp` 以外にも S 式を評価するコマンドは幾つか用意されているが、そのうち利用頻度の高いコマンドを 2 つ紹介しておく。他のコマンドに興味がある者は `M-x apropos-command RET eval- RET` から調べてみるとよいだろう。

- `M-x eval-buffer RET`：カレントバッファの S 式を全て評価する。
- `M-x eval-region RET`：リージョン選択範囲の S 式を全て評価する。

5.2　キーバインドの設定

キーバインドの設定は Emacs の設定の中でも基本中の基本と言える。キーボードから全てを操作可能なエディタとして、キーボード操作を自由に設定することができることは非常に意義があることである。

Emacs の設定は Elisp によって記述されるため、初級者には自由自在に設定をカスタマイズすることは難しいかもしれない。しかし、要点さえ掴むことができればキーバインドの設定はそこまで難しいものではない。

キーバインドの設定には、既に登録されているキーバインドを上書きする場合と未定義のキーバインドに新しくコマンドを割り当てる場合の 2 通りが考えられるが、その方法はどちらも同じである。

以下で解説するキーマップへ登録するだけで、上書きも新規追加も行われる。

5.2.1 キーマップ

キーマップとは Emacs がキーバインドを管理するためのデータ構造である。少々ややこしいが、難しく考えることはない。Emacs がキーバインドを登録しておく対応表（ルックアップテーブル）がキーマップなのである。Emacs はキーボードから入力を受けるとキーマップを検索し、入力に該当するキーバインドが見つかると、そこに登録されているコマンドを実行する。

キーマップは大きく分けて 3 つに分類される。

- **グローバルマップ**（global-map）
  全てのバッファで有効となるキーバインドを登録するためのキーマップ。
- **カレントバッファローカルマップ**（current-local-map）
  特定のバッファのみで有効となるキーバインドを登録するためのキーマップ。通常はフック（後述）と組み合わせて利用する。
- **各モードのキーマップ**
  メジャーモード、そして一部のマイナーモードでそれぞれ独自に利用可能なキーバインドを登録するためのキーマップ。「モード名-mode-map」という名前が付けられることが多い。

キーバインドの優先順位
もし重複して同じキーバインドが登録されている場合でも、次のようなルールに従って先に見つかったキーバインドが優先される。

① マイナーモードのキーマップを検索する。
② カレンドバッファローカルマップを検索する。
③ メジャーモードのキーマップを検索する。
④ グローバルマップを検索する。
⑤ 以上で見つからない場合はエラーメッセージを返す。

もし登録したキーバインドが意図通りに利用できない場合は、この優先順位を確認するとよいだろう。

5.2.2 キーバインドの割り当て

実際にキーバインドを割り当ててみる。キーバインドをキーマップに登録するには define-key 関数を用いる。

```
(define-key キーマップ キーバインド 関数のシンボル)
```

例えば、次のように利用する。

```
;; C-m に newline-and-indent を割り当てる。初期値は newline となっている。
(define-key global-map (kbd "C-m") 'newline-and-indent)
```

この例では、グローバルマップに C-m を入力した際に改行と同時にインデント（字下げ）を行う newline-and-indent コマンドを実行するようにキーバインドをグローバルマップに登録している（この設定の意味は後述する）。

第 2 引数にあたるキーバインドには文字列かキーシーケンスを渡す。上の例ではキーシーケンス (kbd "C-m") を渡しているが、文字列の場合は特殊なエスケープが必要となる。例えば、C-m を \C-m と表記する必要がある。文字列による記述は短くて済む反面、間違いやすくミスが起こりやすくなる。

kbd 関数はエスケープなしでキーバインドの文字列を渡すと、キーシーケンスを返してくれる便利な関数である。そのため、他の記述スタイルに比べ視認性に優れるというメリットがある。本稿では一貫して kbd 関数を用いて記述する方法で解説していく。

5.2.3 キーバインド例

ここでは便利なキーバインド設定例を紹介する。しかし、キーバインドの好みは人それぞれなので、ここで挙げる例はあくまでも参考として閲覧してもらいたい。

改行と同時にインデントする

先の例にも挙げたが、C-m には通常 newline、すなわち改行コードが割り当てられており、C-j には newline-and-indent という改行コードを入力してインデントも行うコマンドが割り当てられている。自然な流れでコーディングを行っている場合は、改行した際にそのままインデントも行ってほしいので、これを C-m に割り当てるのが次の設定である。

```
;; C-m に newline-and-indent を割り当てる。
;; 先程とは異なり global-set-key 関数を利用している。
(global-set-key (kbd "C-m") 'newline-and-indent)
```

今回は global-set-key という新たな関数が登場した。この関数は (define-key global-map ~) の短縮表示のようなものである。どちらを用いても結果は同じなので、好みに応じて選択するとよい。

C-h をバックスペースにする

第 3 章にて C-h はヘルプコマンドであると紹介したが、C-h を <backspace> として利用したい者もいるだろう。Emacs において <backspace> は内部的に <del> として処理されている。そこで、Emacs 内部で C-h を <del> に置き換えることによって C-h を <backspace> として利用することができる。その設定は次のようになる。

```
;; C-h をバックスペースに置き換える。
(define-key key-translation-map (kbd "C-h") (kbd "<DEL>"))
```

行の折返し表示を切り替える

Emacs には 1 行の文字数が画面幅を超える場合の処理として、画面幅で折り返す／折り返さないという 2 つの選択肢が用意されている。この表示の切り替えは M-x toggle-truncate-lines というコマンドによって行うことができるのだが、この切り替えを頻繁に行う場合はキーバインドに登録しておくと便利である。頻繁に実行するコマンドは、どんどんキーバインドに登録することで作業効率が飛躍的にアップする。

```
;; 折り返しトグルコマンドを定義する。
(define-key global-map (kbd "C-c l") 'toggle-truncate-lines)
```

C-c から始まるキーバインドは、拡張機能製作の指針としてユーザの為に開けておくというルール（マナーとも言える）が存在する。従って、ここでは C-c l にコマンドを割り当てた。

簡単にウィンドウを切り替える

昔に比べてモニタサイズが大きくなった昨今、Emacs のウィンドウを分割して作業するというスタイルが標準的になりつつある。そんな状況の中、ウィンドウを切り替えるキーバインドが標準の C-x o では多用するのに不便である。そこで、C-t を C-x o と同じウィンドウを切り替えるキーバインドに変更してみる。

```
;; C-t でウィンドウを切り替える（初期値は transpose-chars）。
(define-key global-map (kbd "C-t") 'other-window)
```

これで C-t で分割されたウィンドウ間を移動できるようになる。

5.3 環境変数の設定

ターミナル環境を普段利用しない Windows、Mac ユーザにとっては、環境変数という言葉はあまり耳馴染みがないかもしれない。環境変数とは OS が提供する非常に大事な機能であり、プログラムは環境変数に設定された値を参照して振る舞いを変化させる。

ターミナル環境ではコマンドの名前のみを入力して実行するが、これは環境変数にパス（PATH）というコマンドを検

索するディレクトリが設定されているからである。パスに登録されていない（パスの通っていない）ディレクトリに配置されているコマンドは、コマンドの存在するディレクトリに移動しない限りコマンド名のみでは実行することができない。

このような環境変数だが、ターミナル環境ではシェルから引き継がれるため、Emacs 上で別段設定する必要はないはずである。また、Windows の場合もシステムのプロパティにある環境変数の値が自動的に利用される。ただ、Mac の Emacs.app は基本的に環境変数を引き継がないため、Emacs の中で設定する必要がある。

5.3.1 パスの設定

前述の通り、パスとはコマンドを検索するためのディレクトリのことである。Emacs が起動する際、OS の環境変数 PATH を読み取ってパスの値が設定される。この値を調べたい場合は `M-x getenv RET PATH RET` というコマンドを利用する。

しかし、Emacs が実際にコマンドを実行する際は、この PATH の値を直接利用するのではなく PATH の値に基づいて Emacs 規定のコマンド検索パスを追加してリスト化した exec-path の値からコマンドを検索する。従って、もし Emacs 起動後にパスを追加する必要がある場合には次のようにして exec-path のリストにパスを追加する必要がある。

```
(add-to-list 'exec-path "/usr/local/bin")
(add-to-list 'exec-path "/opt/local/bin")
```

PATH 以外の環境変数についても `M-x getenv RET 環境変数名 RET` から調べることができる。尚、環境変数は process-environment 変数に格納されているので、`C-h v process-environment RET` から全ての値を確認することもできる。

5.3.2 文字コードの設定

2000 年くらいまで、テキストファイルにおいて文字コードは常に意識しておくべき最重要項目の 1 つであった。

今日でもその重要性に変わりはないのだが、近年は OS 標準のソフトウェアが様々な文字コードを扱うことができること、自動判別の制度が上がったことに加えて、UTF-8 が文字コードの業界標準になりつつあるため、2000 年以降に PC を使い始めた者がその重要性に気づかないとしても無理もないだろう。

しかし、今後文字コードを意識しなかったために起こるかもしれないミスを未然に防ぐためにも、文字コードは意識しておくべきである。

尚、カレントバッファの文字コードは第 3 章で解説した通りモードラインに表示されている。

現在の文字コード設定を調べる

Emacs の文字コードの設定はかなり詳細に分かれている。現在の Emacs の文字コードに関する設定がどの様になっているかを確認するためには、`M-x describe-current-coding-system RET` を実行する。すると、*Help* バッファに次のように表示される。

```
Coding system for saving this buffer:
  Not set locally, use the default.
Default coding system (for new files):
  U -- utf-8-unix (alias: mule-utf-8-unix)
Coding system for keyboad input:
  U -- utf-8-unix (alias: mule-utf-8-unix)
(以下略)
```

上から順に、カレントバッファのファイル保存時の文字コード、新規ファイル作成時の文字コード、キーボード入力のための文字コードといった具合に設定が続く。

```
Priority order for recognizing coding system when reading files:
  1, utf-8 (alias: mule-utf-8)
  2, iso-202207bit
  3, iso-latin-1 (alias: iso-8859-1 latin-1)
```

次に「Priority order for」から続く行はファイル読み込み時の自動判別の優先順位である。様々なソフトウェアで起こる（最近は滅多に起こらないが）文字化けは、文字コードの自動判別に失敗することによって起こる現象である。

Emacs では限りなくこの可能性をゼロにするため、利用者の環境で開きやすい文字コードを優先的に上位に配置し、文字化けを防いる。

最後に「Particular coding system」から続く行を見てみる。

```
Particular coding systems specified for certain file names:

OPERATION         TARGET PATTERN     CODING SYSTEM(s)
---------         --------------     ----------------
File I/O          "\\.dz\\'"         (no-conversion . no-conversion)
                  "\\.txz\\'"        (no-conversion . no-conversion)
```

最初の値は、File I/O を介した dz 拡張子を持つファイルを扱う場合の文字コードは no-conversion となりバイナリファイルと同じ扱いをする、という意味である。文字コードの指定は (decoding-system . encoding-system) という形式、もしくは定義済みのコーディングシステムとなっている。

尚、Emacs で扱うことができる文字コード（定義済み）の一覧は `M-x list-coding-system RET` から確認することができる。

文字コードを指定する

ここでは文字コードの具体的な設定方法を解説する。最近の Emacs は特に指定せずともユーザのロケール（言語設定）を読み取って適切に設定してくれるようになっている。そのため、次の設定は不要かもしれない。

```
(set-language-environment "Japanese")
(prefer-coding-system 'utf-8)
```

1 行目の set-language-environment 関数ではユーザのロケールを指定する。これによって設定される主な値を表 5.1 にまとめる。

表 5.1: set-language-environment で設定される値

名前	個別に設定する際のコマンド
標準の文字コード	set-file-name-coding-system
文字コードの優先順位	set-coding-system-priority
インプットメソッド	set-input-method
文字セットの優先順位	set-charset-priority

set-language-environment で指定する Japanese では環境設定としては曖昧であり、実際にどの変数にどのような値が設定されるのか分からない。Japanese と指定してどの変数にどのような値が設定されるのかを確認するためには、`M-x describe-variable RET language-info-alist RET` を実行する。

5.4 フレームに関する設定

本節ではフレームに関する設定を解説する。フレームは Emacs がバッファを描画している画面全体のことなので、ここからの設定は Emacs の見た目に関する設定ということになる。

5.4.1　モードラインに関する設定

まずは簡単に変更可能なところから設定してみることにする。モードラインは標準で様々な情報を追加表示することが可能である。

行番号／カラム番号を表示する

標準で用意されている設定の中で、モードラインにカーソル位置の行番号とカラム番号を表示させるというものがある。行番号は標準で表示されているはずなので、カラム番号の表示を加えてみることにする。そのためには、設定ファイル（init.el）に以下の行を追記して評価する。

```
;; モードラインにカラム番号も表示する。
(column-number-mode t)
```

すると、今まで行番号が表示されていた場所の表示が（行番号, カラム番号）という形式の表示に切り替わる。この設定は `M-x column-number-mode RET` でトグル切り替えを行うことができる。

行番号を表示させたくない場合は、次のような設定を記述する。

```
;; モードラインに行番号を表示させない。
(line-number-mode 0)
```

この関数の引数に着目すると、オンの場合に t を、オフの場合に 0 を指定しているが、これは Emacs におけるスイッチオン／オフの慣例的な指定方法である。Elisp では真偽値の真は nil 以外の全ての値だが、真を表すための特別なシンボルとして t を用いる。

オフは真偽値の偽を与えればよさそうなものだが、トグル切り替えを行う関数に nil を与えると、オンであればオフ、オフであればオンといったトグルを逆転する特別な操作になる。明示的にオフにしたい場合は 0 以下の整数値を与える仕組みになっているが、慣例的には 0 を用いることになっている。

ファイルサイズ、時計、バッテリー残量を表示する

ファイルサイズ、時計、バッテリー残量を表示する設定をまとめて以下に記す。

```
;; ファイルサイズを表示する。
(size-indication-mode t)
;; 時計を表示する（好みに応じてフォーマットを変更可能）。
(setq display-time-day-and-date t) ; 曜日・月・日を表示
(setq display-time-24hr-format  t) ; 24 時表示
(display-time-mode t)
;; バッテリー残量を表示する。
(display-battery-mode t)
```

標準で用意されているのは以上だけなのだが、拡張機能を追加したり自分で Elisp を記述すれば柔軟に表示を追加／変更することが可能である。例えば、リージョンで範囲選択している際にリージョン内の行数と文字数を表示するという設定を紹介しておく。

```
;; リージョン内の行数と文字数をモードラインに表示する。
(defun count-lines-and-chars ()
  (if mark-active
      (format "(%dlinrs,%dchars) "
              (count-lines (region-beginning) (region-end))
              (- (region-end) (region-beginning))) ""))
(add-to-list 'default-mode-line-format
             '(:eval (count-lines-and-chars)))
```

5.4.2 タイトルバーにファイルのパス名を表示する

Emacs ではタイトルバーに表示される内容も自由に変更することができる。例えば、タイトルバーに編集中のファイルのパスを表示したい場合には、次の行を設定ファイルに追記して評価する。

```
;; タイトルバーにファイルのフルパスを表示する。
(setq frame-title-format "%f")
```

この設定では frame-title-format という変数に `%f` というフォーマットテンプレート（表 5.2 にモードライン用に定義された％記法の一覧を記す）を設定している。

表 5.2: モードライン用フォーマットテンプレート（％記法）

名前	説明
%b	カレントバッファ名を表示する。
%f	ファイルのフルパスを表示する。
%F	選択中のフレーム名を表示する。
%*	バッファが読み出し専用であれば \%、変更されていれば *、それ以外は - を表示する。
%+	バッファが読み出し専用であっても変更されていれば *、変更されていなければ %、それ以外は - を表示。
%&	バッファが読み出し専用であっても変更されていれば *、それ以外は - を表示する。
%s	サブプロセスのステータスを表示する。
%l	カーソル位置の行番号を表示する。
%c	カーソル位置のカラム番号を表示する。
%i	バッファのファイルサイズを表示する（バイト単位）。
%I	ファイルサイズを表示する（k、M、G バイト単位）。
%p	ウィンドウ上端より上にあるバッファの量の割合（パーセンテージ）を表示する。
%P	ウィンドウ下端より上にあるバッファの量の割合（パーセンテージ）を表示する。
%n	ナローイング中であれば Narrow を表示する。
%t	ファイルがテキストであれば T、バイナリであれば B を表示する（但し、OS に依る）。
%z	キーボード、端末、ファイルの文字コードを表示する（キーボードと端末はターミナル環境のみ）。
%Z	文字コードと改行コードを表示する。
%e	エラーメッセージを表示する。
%@	ファイルがローカルホストであれば –、リモートホストであれば @ を表示する。
%[	再帰編集（*recursive ediiting*）のレベルを [の数で表示する。%] も同様。
%%	% 自身を表示する。
%-	無限に - を表示する。

5.4.3 ウィンドウ左に行番号を表示する

常に行番号をウィンドウの左側に表示させておきたい場合は `M-x linum-mode` を用いることで表示させることができる。linum-mode、global-linum-mode の 2 種類がコマンドとして用意されており、linum-mode はバッファローカル、global-linum-mode は全てのバッファに対して反映される。起動時に有効化したければ、設定ファイルに次の行を追記する。

```
;; 行番号を常に表示する。
(global-linum-mode t)
```

5.5 インデントの設定

インデントを適切に利用することは、読み易く綺麗なコードを書き、長く保守管理可能とするための重要な習慣である。共同開発の現場では、インデントに関するルールが定められていることも珍しくない。Emacs は標準で十分読み易いインデントを提供してくれるが、ここでは一歩進んで自分でインデントの設定を行えるよう解説しておく。

5.5.1 タブ文字の表示幅

タブ文字は文字としては 1 文字だが、表示される幅は環境によって異なる。Emacs では tab-width の値を参照してタブ文字の表示幅を決定しているため、この値を変更することでタブ文字の表示幅を変更することができる。

```
;; TAB の表示幅（初期値は 8）
(setq-default tab-width 4)
```

ここでは今までの setq ではなく setq-default という新しい関数を用いて設定を行っている。tab-width はバッファローカルな変数であるため、setq を用いるとカレントバッファのみに変更が反映される。その為、全てのバッファで設定を変更したい場合は setq-default を用いる必要がある。

5.5.2 タブ文字の利用

Emacs 標準のインデント設定では、タブ文字と空白文字の両方を使用する。インデントの際、タブ文字を使用したくなければ indent-tabs-mode 変数に nil を設定することで、その後はインデントにタブ文字を一切使用しなくなる。

```
;; インデントにタブ文字を使用しない。
(setq-default indent-tabs-mode nil)
```

indent-tabs-mode もバッファローカル変数であるため、全ての環境で有効化するには setq-default 関数を用いる必要がある。もし、特定のモードのみで有効化したい場合は後述するフックを利用して次のように設定を記述する。

```
;; php-mode のみタブ文字を利用しない。
(add-hook 'php-mode-hook
          (lambda ()
            (setq indent-tabs-mode nil)))
```

5.5.3 C、C++、Java、PHP などのインデント

多くの言語では c-mode で定義されているインデント設定を流用している。c-mode ではインデント設定をスタイルとして定義し、複数用意している。このスタイルを切り替えるのは `M-x c-set-style RET スタイル名 RET` で常に可能だが、スタイルを変更しただけではインデントは反映されない。バッファ全体のインデントを変更したい場合は、`C-x h TAB`（バッファ全体をリージョン選択してタブによるインデント）を実行する。

デフォルトのインデントスタイルを変更するには c-mode-common-hook に c-set-style のスタイル名を設定する。

```
(add-hook 'c-mode-common-hook
          (lambda ()
            (c-set-style "bsd")))
```

スタイルには表 5.3 のようなものが用意されている。

表 5.3: スタイル名

スタイル名	インデントのカラム数
gnu	2
k&r	5
bsd	8

stroustrup	4
whitesmith	4
ellemtel	3
linux	8
python	8
java	4
awk	4

尚、スタイルは c-style-alist 変数にリストとして定義されており（すなわち、独自のスタイルを追加することも可能）、この変数の値を確認（`M-x describe-variable RET c-style-alist RET`）することで詳細を知ることができる。

この c-mode 標準のインデントは開き括弧のインデント、閉じ括弧のインデント、switch 文のインデント、引数リストのインデントなど様々な項目のインデントを細かく指定することが可能となっている。例えば、PHP モードは独自に php というインデントスタイルを定義して利用しており、switch 文の case ではインデントが行われなくなっている。これを変更したい場合、次のように指定する。

```
(add-hook 'php-mode-hook
          (lambda ()
            (c-set-offset 'case-label '+)))
```

case-label は switch 文の case のことをを指している。ここで指定した '+ は標準のインデントのカラム数（c-basic-offset 変数の値）を利用するという意味のシンボルである。その他の言語のインデントについては、第 7 章「各種言語の開発環境」にて詳しく解説する。

5.6 表示・装飾に関する設定

本節では表示・装飾に関する設定を解説する。これは主にソースコードの色分け、すなわちシンタックスハイライトに関する部分である。シンタックスハイライトはソースコードの見易さは勿論のこと、記述ミスなどの構文チェックにも有用である。

5.6.1 フェイス

Emacs のシンタックスハイライトはフェイスと呼ばれる仕組みによって実装されている。このフェイスは特定のキーワードに対して、文字色、背景色、下線、斜体、太字などの文字修飾を設定し、キーワードにマッチする部分にその装飾を施す。フェイスの指定方法は表 5.4 のようになっている。

表 5.4: フェイス設定関数一覧

関数	説明
`(set-face-foreground フェイス名 カラー)`	文字色を指定する。
`(set-face-background フェイス名 カラー)`	背景色を指定する。
`(set-face-background フェイス名 nil)`	背景色を無しにする。
`(set-face-bold-p フェイス名 t)`	太字にする。
`(set-face-bold-p フェイス名 nil)`	太字をやめる。
`(set-face-italic-p フェイス名 t)`	斜体にする。
`(set-face-italic-p フェイス名 nil)`	斜体をやめる。
`(set-face-underline-p フェイス名 t)`	下線を表示する。
`(set-face-underline-p フェイス名 nil)`	下線の表示をやめる。

現在のカーソル位置のフェイスは、M-x describe-face RET によって確認することができる。また、Emacs 上で定義されている全てのフェイス一覧を確認するために list-faces-display というコマンドも用意されている。フェイスを設定するには、例えば次のように設定を記述する。

```
;; リージョンの背景色を変更する。
(set-face-background 'region "darkgreen")
```

尚、Emacs で指定可能な色の一覧は、M-x list-colors-display RET で確認することができる。

5.6.2 フォントの設定

Emacs 23.1 から内部の文字コード処理が洗練され、1 文字単位でフォントを指定することができるようになった。しかし、そのためのインタフェースは用意されていないため設定は難しいと言われているが、ポイントさえ掴めば意外と簡単である。

scratch バッファで (prin1 (font-family-list)) を評価すると、Emacs で利用可能なフォント一覧が出力される。フォント名には和名も利用可能である。

英語フォントを指定する

実際にフォントを設定するためには、設定ファイルに次の行を追記する。

```
;; Ascii フォントを Menlo に設定する。
(set-face-attribute 'default nil
                    :family "Menlo"
                    :height 120)
```

set-face-attribute 関数を用いて利用可能なフォントから指定する。:height はフォントサイズである。

日本語フォントを指定する

set-fontset-font 関数と font-spec 関数を用いて日本語フォントを指定する。ここでは例として、Google が提供している Noto フォント[*1]を利用する。実際に試す場合には、予めダウンロードしてインストールしておくこと。

```
;; 日本語フォントを Noto Serif CJK JP に設定する。
(set-fontset-font nil 'japanese-jisx0208
                  (font-spec :family "Noto Serif CJK JP"))
```

また、漢字以外の全角文字だけフォント変更することも可能である。設定方法は、日本語を変更した後に平仮名だけ別のフォントで上書きする形となる。指定には Unicode の符号を利用する。

```
;; 平仮名とカタカナを Noto Sans CJK JP に変更する。
;; U+3000-303F CJK の記号および句読点
;; U+3040-309F ひらがな
;; U+30A0-30FF カタカナ
(set-fontset-font nil '(#x3040 . #x30ff)
                  (font-spec :family "Noto Sans CJK JP"))
```

フォントの横幅を調節する

set-fontset-font 関数で設定したフォントのサイズ調整は face-font-rescale-alist から行う。

```
;; Noto フォントの横幅を調節する。
(add-to-list 'face-font-rescale-alist '(".*Noto*" . 1.2))
```

(フォント名の正規表現 . 横幅の倍率) という書式で指定する。うまく調節することができれば、全角文字と半角文字の比率を 2 : 1 にすることが可能である。

[*1] https://www.google.com/get/noto

5.7　ハイライトの設定

シンタックスハイライトやフォント以外にも、編集時の見易さを支援する仕組みが用意されている。例えば、現在行の表示を目立たせることで直ちに見つけることが可能となる。また、括弧がネスト（入れ子）になっている場合、対応関係を視覚化することで編集ミスを防ぐことが可能となる。

5.7.1　現在行のハイライト

現在行をハイライトして目立たせたい場合、標準で含まれる hl-line-mode を用いる。表示をカスタマイズしたい場合はフェイスを調節することで自由に変更可能である。

```
(defface my-hl-line-face
  ;; 背景が dark ならば背景色を NavyBlue にする。
  '((((class color) (background dark))
     (:background "NavyBlue" t))
    ;; 背景が light ならば背景色を LightSkyBlue にする。
    (((class color) (background light))
     (:background "LightSkyBlue" t))
    (t (:bold t)))
  "hl-line's my face")
(setq hl-line-face 'my-hl-line-face)
(global-hl-line-mode t)
```

5.7.2　対応する括弧のハイライト

カーソル位置の括弧と対応する括弧を強調表示することが可能である。Elisp は括弧が多く登場する言語なので、この設定を行うことで Elisp の記述が非常に楽になる。

```
;; paren-mode：対応する括弧を強調して表示する。
(setq show-paren-delay 0) ; 表示までの秒数。初期値は 0.125 秒。
(show-paren-mode t)       ; 有効化する。
;; paren のスタイルを設定：expression は括弧内も強調表示する。
(setq show-paren-style 'expression)
;; フェイスを変更する。
(set-face-background 'show-paren-match-face nil)
(set-face-underline-p 'show-paren-match-face "darkgreen")
```

5.8　バックアップとオートセーブ

コンピュータ上の作業では、時として異常事態が起こりアプリケーションや OS が突然終了することがある。その際、まだ保存されていないファイルはその作業が失われることになる。

しかし、優秀なアプリケーションはこのような自体に備えて最悪の事態を回避する努力をしている。Emacs においては、第 3 章で解説したオートセーブがそれに当たる。Emacs では標準で 300 回のタイプ、あるいは 30 秒間何もしないと編集中のファイル名の前後に # を付けたオートセーブファイルを作成するようになっている。

5.8.1　オートセーブからの復元

オートセーブファイルからの復元はとても簡単である。例えば、~/.emacs.d/init.el ファイルを編集中に何らかの事情で Emacs が突然終了したとする。しかし、幸いにも ~/.emacs.d/#init.el# ファイルが存在していた。

オートセーブファイルから復元したい場合には、次のコマンドを実行する。# 付きの #init.el# ではないことに注意すること。

```
M-x recover-file RET ~/.emacs.d/init.el RET
```

次のような問いがミニバッファに表示されるので yes と答える。

```
Recover auto save file /Users/ユーザ名/.emacs.d/#init.el#?  (yes or no)
```

すると、オートセーブファイルの内容が反映された init.el が開かれる。この時点ではまだファイルの保存がされていないため、問題がなければ保存しておく。

5.8.2 バックアップとオートセーブの設定

バックアップとオートセーブをより便利に利用するためにの設定を紹介しておく。

バックアップとオートセーブファイルの作成先を変更する

Web 上に公開されている多くの者の設定ファイルを参照すると、以外にもバックアップファイルとオートセーブファイルを利用しない設定をしている者が一定数いるようである。

```
;; バックアップファイルを作成しない。
(setq make-backup-files nil) ; 初期値は t
;; オートセーブファイルｗｐ作成しない。
(setq auto-save-default nil) ; 初期値は t
```

もしもの為のせっかくの機能が用意されているにも関わらず、利用しないのは勿体ない話である。恐らく、オートセーブを利用しない者の心理として、邪魔なファイルを自動的に作成されるのが嫌なのであろう。そんな者のために、編集ファイルと同じ場所ではなく特定のディレクトリに集めてしまう方法を紹介する。

```
;; バックアップファイルの作成場所をシステムの Temp ディレクトリに変更する。
(setq backup-directory-alist
      '((".*" . ,temporary-file-directory)))
;; オートセーブファイルの作成場所をシステムの Temp ディレクトリに変更する。
(setq auto-save-file-name-transforms
      '((".*" ,temporary-file-directory t)))
```

この設定の Elisp 中にある ' はバッククォート構文と呼ばれるもので、リスト中の , に続く引数を評価する（すなわち、評価後の値が利用される）。尚、この temporary-file-directory はシステムの temp ディレクトリとなっているが、例えば次のように ~/.emacs.d 以下に作成した backups というディレクトリをしている者も多い。

```
;; バックアップとオートセーブファイルを ~/.emacs.d/backups/ へ集める。
(add-to-list 'backup-directory-alist
             (cons "." "~/.emacs.d/backups/"))
(setq auto-save-file-name-transforms
      '((".*" ,(expand-file-name "~/.emacs.d/backups/") t)))
```

オートセーブの間隔を変更する

オートセーブファイルを標準の間隔よりも早く作成してほしい場合は、次の設定を追記する。

```
;; オートセーブファイル作成までの秒間隔を設定する。
(setq auto-save-timeout 15)
;; オートセーブファイル作成までのタイプ間隔を設定する。
(setq auto-save-interval 60)
```

5.9 変更されたファイルの自動更新

Emacs でファイルを編集中に別のプログラムからファイルが更新された場合、Emacs 上のファイル（バッファ）はまだ変更されておらず、ファイルを編集しようとした際に変更を反映するかを確認される。この場合、y を押すと編集を続行し、n を押すとファイルの変更を中断、r を押すとファイルを読み込み直す。

この動作は非常に安全な仕組みだが、Git で別のブランチにチェックアウトするなど、大量のファイルが更新されるような場合は、わざわざ全てのファイルに変更を反映させるかどうかを確認しなければならず、逆に面倒に感じることがある。

5.9.1 ファイルの自動更新

ファイルに変更があった場合、確認を行わずに即剤に変更を反映してほしいのであれば、次の設定を追記する。

```
;; 更新されたファイルを自動的に読み直す。
(global-auto-revert-mode t)
```

すると、Emacs は全てのファイルについて確認を行わずに自動的にファイルを読み込み直すようになる。

5.10 フック

本節ではフックという設定のテクニックを解説する。フック（*hook*）とは、引っ掛けるフックと同じニュアンスであり、特定のイベント（ファイルの保存やメジャーモードの変更など）に予め関数をセットして（引っ掛けて）おき、イベント発生時にセットした関数を自動的に実行する仕組みである。

これを利用することで、例えばメジャーモードだけで常に有効化したい機能をフックに設定しておくことで自動的に実行されたかのように動作する。

5.10.1 自動化の仕組み

実際の例を見ながら解説する。

```
;; ファイルが #! から始まる場合、+x 属性を付加して保存する。
(add-hook 'after-save-hook
          'executable-make-buffer-file-executable-if-script-p)
```

after-save-hook は Emacs がバッファをファイルとして保存する際に実行されるフックである。フックとして登録している executable-make-buffer-file-executable-if-script-p という関数はファイルをチェックして、もし 1 行目が「#!」から始まるならば「+x（実行権）」を与えて保存する関数である。

このように、特定の動作に対して予め実行する関数をセットしておくことで、毎回行うような作業を自動化することが可能となる。

5.10.2 利用方法

フックの設定は、これまでも何度かサンプルが登場しているが、次のように記述する。

```
(add-hook フック名 実行する関数)
```

第 2 引数の「実行する関数」の部分は、関数を 1 つだけ追加する場合はそのシンボルを渡せばよいのだが、複数の関数を追加したい場合は lambda 式を利用する方法とフック用の関数を定義してそのシンボルを渡す方法の 2 通りの方法がある。尚、次の例で登場する emacs-lisp-mode-hook など特定のモード用のフックに記述した設定は、その設定を評価した際ではなく、それ以降に emacs-lisp-mode が選択された際に実行される。

無名関数 lambda を利用する場合

無名関数 lambda を利用する場合は次のようになる。

```
;; emacs-lisp-mode のフックをセットする。
(add-hook 'emacs-lisp-mode-hook
          '(lambda ()
             (when (require 'eldoc nil t)
```

```
    (setq eldoc-idle-delay 0.2)
    (setq eldoc-echo-area-use-multiline-p t)
    (turn-on-eldoc-mode))))
```

この設定はカーソル位置にある Elisp 関数や変数の情報をエコーエリアへ表示させる有用な設定である。この設定を init.el へ追記すると init.el の編集が非常に楽になる。

関数を定義する場合

続いて、関数を定義する場合は次のようになる。

```
;; emacs-lisp-mode-hook 用の関数を定義する。
(defun elisp-mode-hooks ()
  "lisp-mode-hooks"
  (when (require 'eldoc nil t)
    (setq eldoc-idle-delay 0.2)
    (setq eldoc-echo-area-use-multiline-p t)
    (turn-on-eldoc-mode)))
;; emacs-lisp-mode のフックをセットする。
(add-hook 'emacs-lisp-mode-hook 'elisp-mode-hooks)
```

一見すると、lambda 式を利用した方が楽なように見えるが、実は大きな欠点が存在する。それは、もし Emacs 起動中に lambda 式の中身を変更した場合、その内容だけを評価することができないため、その変更内容は基本的に Emacs の再起動時にしか設定が反映されないという点である。

一方、専用の関数を定義した場合はフック実行時に関数が呼び出されるため、中身を変更してもその関数自体を評価するだけで設定が反映される。従って、できる限り定義済みの関数をフックに追加することを推奨する。

5.10.3 代表的なフック一覧

特に頻繁に利用されるフックは表 5.5 の通りである。

表 5.5: イベント用フック一覧

名前	説明
after-save-hook	ファイル保存後に実行される。
fine-file-hook	`C-x C-f (find-file)` でファイルを開いた際に実行される。
emacs-startup-hook	Emacs 起動時に設定ファイルを読み終えてから一度だけ実行される。
kill-emacs-hook	Emacs 終了時に実行される。
c-mode-hook	c-mode を起動した後に実行される。
php-mode-hook	php-mode を起動した後に実行される。
cperl-mode-hook	cperl-mode を起動した後に実行される。
emacs-lisp-mode-hook	emacs-lisp-mode ｗｐ起動した後に実行される。
lisp-interaction-mode-hook	lisp-intraction-mode を起動した後に実行される。
js-mode-hook	js-mode を起動した後に実行される。
css-mode-hook	css-mode を起動した後に実行される。
nxml-mode-hook	nxml-mode を起動した後に実行される。

例えば、特定のメジャーモードにおいて after-save-hook を利用したい場合は、次のように設定ファイルに追記することで実現することができる。

```
(defun php-after-save-hooks ()
  ;; メジャーモードが php-mode と一致する場合は中身を実行する
  (when (eq major-mode 'php-mode)
    実行したい処理
    ))
;; after-save-hook に関数をセットする。
(add-hook 'after-save-hook 'php-after-save-hooks)
```

第 6 章

テキスト編集を更に効率化する拡張機能

6.1　Elisp のインストール

これまで何度も触れてきたが、Emacs は Elisp によって自由自在に拡張することが可能となっている。自分で Elisp を記述して拡張するのもよいが、まずは世界中のハッカー達によって書かれた Elisp をインストールすることで Emacs を別のアプリケーションように進化させてみよう。

本章では、Elisp のインストール方法について解説した後、代表的な拡張機能のインストールと設定について詳述する。

6.1.1　インストール方法の種類

Emacs 24 以降から Elisp のインストールは基本的に package.el というパッケージマネージャを用いて行う。この機能が搭載されるまでは、手動で Elisp をダウンロードしてロードパスに追加されているディレクトリに配置する必要があった。本節では、それぞれのインストール方法の違いについて解説した後、手動インストールと package.el によるインストールの両方を解説する。

手動インストール

手動インストールはロードパスに追加されている（パスの通った）ディレクトリに el ファイルを配置するというシンプルで簡単な方法である。現在、多くの Elisp が ELPA からインストール可能となっているが、もしインストールしたい Elisp が ELPA に対応していない場合はこちらの方法を用いてインストールすることになる。package.el が使えれば自動的にインストール可能なのだが、覚えておいて損はない。

package.el によるインストール

package.el は Red Hat Linux の yum や Mac の Homebrew または Node.js の npm（*node Package Manager*）などのように拡張機能のインストールや削除などを行ってくれるツールである。package.el を用いた Elisp のインストールは ELPA（*Emacs Lisp Package Archive*：エルパと読む）という専用のパッケージリポジトリを通じて行われる。必要に応じてリポジトリの追加を行うことで、多くのパッケージをインストールすることが可能である。また、インストールしたい Elisp が他の Elisp の機能に依存している場合は依存元の Elisp も合わせてインストールしてくれる。

Elisp を最新バージョンに更新したい場合も package.el によって行うことが可能である。リポジトリをチェックしてインストールされているパッケージからの更新があれば簡単にアップデートが行える。とても便利な package.el なのだが、1 つだけ大きな欠点が存在する。それはバージョンを管理する機能が用意されていないという点である。すなわち、ELPA を通じてインストールされる Elisp は常に最新バージョンとなってしまうため、もし最新バージョンの Elisp に何らかの問題があってもダウングレードすることができないという点である。頻繁に発生する問題ではないと思われるが、こういった事態に対処したい場合は自分自身で Git などを利用して ELPA からインストールされる ELisp をバージョン管理する必要がある。

6.1.2　手動インストール

本項では手動インストールについて解説する。まずは Elisp ファイルの配置について解説した後、配置されたファイルのバイトコンパイルについて解説する。

Elisp を配置する

インストールしたい Elisp をロードパスに追加されているディレクトリに配置する。ここでは例としてアンドゥを強化する undo-tree.el をインストールしてみることにする。この拡張機能は ELPA からインストール可能となっているが、ここでは解説のために手動でインストールしてみることにする。

Linux や Mac の場合、ターミナルから次のようにコマンドを実行するのが簡単である。

```
$ cd ~/.emacs.d/elisp
$ curl -0 http://www.dr-qubit.org/undo-tree/undo-tree.el
```

~/.emacs.d/elisp を設定によってロードパスへ追加している場合、これだけで拡張機能のインストールは完了である。Windows の場合はブラウザなどからファイルをダウンロードして ~/.emacs.d/elisp フォルダへコピーする。後は、第 4 章で解説した設定を init.el へ追記し、起動と同時に利用可能としておく。

バイトコンパイルする

バイトコンパイルとは Elisp の読み込みを速くするために el ファイルをバイトコードと呼ばれる特別な形式に変換して elc ファイルというファイルを出力する処理のことである。拡張機能の中にはバイトコンパイルが必須のものも存在し、インストールした拡張機能はバイトコンパイルしておくのが一般的である。package.el を用いてインストールされた Elisp は自動的にバイトコンパイルされるため、この処理は不要となる。

elc ファイルが作成されると、同名の el ファイルが存在したとしても load 関数も require 関数も elc ファイルのみを読み込む。従って、バイトコンパイル済みの el ファイルを編集した場合は必ず再度バイトコンパイルする必要がある。

注意点として、Emacs 22 と 23 では内部文字コードが大きく変更されたため、23 でバイトコンパイルしたファイルは 22 以前では読み込むことができない。逆に 22 でバイトコンパイルしたファイルは 23 以降でも読み込み可能となっている。Emacs 22 と 23 を併用する場合は、とりあえず 22 でバイトコンパイルしておくか、拡張機能のインストール先を分ける必要がある。

byte-compile-file コマンド

今回手動でインストールした undo-tree.el はまだバイトコンパイルされていないため、これをバイトコンパイルしてみる。Emacs を起動した状態で `M-x byte-compile-file RET` を実行するとミニバッファでファイル名を問われるので ~/.emacs.d/elisp/undo-tree.el を指定する。すると、undo-tree.el ファイルがバイトコンパイルされ同じディレクトリに undo-tree.elc ファイルが生成される。

シェルからバイトコンパイルする

Emacs にはエディタとしての利用法以外に、フレームを描画せず Perl や Ruby のように Elisp を処理するためのバッチモードという機能が備わっている。これを利用してシェルからコマンドラインによって Elisp ファイルをバイトコンパイルすることも可能である。

```
バージョンを指定する方法
$ emacs-25.2 -batch -f batch-byte-compile ~/.emacs.d/elisp/undo-tree.el
Emacs.app を利用する方法（Mac 専用）
$ /Application/Emacs.app/Contents/MacOS/
   Emacs -batch -f batch-byte-compile ~/.emacs.d/elisp/undo-tree.el
emacs.exe を利用する方法（Windows 専用）
> \emacs\bin\emacs.exe -batch -f atch-byte-compile ~/.emacs.d/elisp/undo-tree.el
```

6.1.3 package.el によるインストール

次に package.el によるインストール方法を解説する。まずパッケージリポジトリの追加について解説した後、実際のインストール方法を解説する。初回のみ少々設定が必要となるが、その後は基本的に設定なしで利用可能である。

ELPA リポジトリを追加する

package.el は ELPA リポジトリを通じて Elisp のインストールを行うが、この ELPA は 1 つではない。代表的な ELPA リポジトリは以下の通りである。

- GNU ELPA： http://elpa.gnu.org/
- marmalade： https://marmalade-repo.org/
- MELPA： https://melpa.org/

GNU ELPA は Emacs の開発元である GNU プロジェクトが管理しているリポジトリであり、本家本元の ELPA である。FSF（Free Software Foundation）が著作権を保護している Elisp のみが登録されている。

他の 2 つは ELPA 互換リポジトリである。GNU ELPA は著作権を FSF へ譲渡する必要があるが、こちらのリポジトリではその必要はない。誰もが気軽に Elisp の登録が行えるようになっており、GNU ELPA よりもかなり多くの Elisp が登録されている。その為、package.el を利用して Elisp をインストールする場合には、このどちらかのリポジトリを追加することになる。

尚、これ以外にも ELPA 互換リポジトリ（以下、特に区別を必要としない場合は単に ELPA と記述する）は存在するので、詳しく知りたい者は Emacs Wiki の ELPA のページ[*1]を参照すること。

それでは ELPA を追加してみる。初期状態では GNU ELPA のみがリポジトリとして登録されているため、それ以外の ELPA を追加する場合は次の設定を追記する。

```
(require 'package) ; package.el を有効化する。
;; パッケージリポジトリに Marmalade と MELPA を追加する。
(add-to-list
 'package-archives
 '("marmalade" . "https://marmalade.org/packages/"))
(add-to-list
 'package-archives
 '("melpa" . "https://melpa.org/packages/#/"))
(package-initialize) ; インストール済みの Elisp を読み込む。
```

最後の package-initialize 関数は package.el によってインストールされた Elisp を読み込むためのコマンドである。この設定を追記しておくことで、今後インストールした Elisp のほとんどが特に設定を必要とすることなく利用可能となる。

これで ELPA から Elisp をインストールする準備が整ったので、実際のインストールへ進む。

パッケージ一覧を取得する

それでは ELPA に登録されているパッケージを実際にインストールしてみる。初めに、パッケージ一覧を取得するコマンドを実行する。

```
M-x list-packages
```

表示に少々時間がかかるかもしれないが、ELPA に登録されているパッケージ一覧が表示される。パッケージ一覧が表示されているバッファは package-menu-mode というメジャーモードになっており、操作は表 6.1 の通りである。

表 6.1: package-mode-menu の操作一覧

キー	説明
h	ミニバッファに操作ヘルプを表示する。
p	前の行へ。
n	次の行へ。
?, RET	パッケージの説明を取得する。

[*1] https://www.emacswiki.org/emacs/ELPA

i	インストール候補としてマークする。
U	アップデート可能なパッケージを全てマークする。
d	削除候補としてマークする。
DEL	1 行上のマークを外す。
u	現在行のマークを外す。
x	マークしたパッケージをインストール／削除する。
r	パッケージ一覧をリフレッシュする。
q	ウィンドウを閉じる。
f	フィルタで絞り込む（g または q で解除）。

パッケージ一覧から選択してインストールする

例として Emacs のバッファを表示通りに HTML 化する htmlize という拡張機能をインストールしたい場合、htmlize の行で i を押す。すると htmlize がインストール候補としてマークされる。そして x を押すと、ミニバッファに実行確認の yes/no が問われるので、yes RET と答えるとパッケージのインストールが開始される。

インストール先はデフォルトで ~/.emacs.d/elpa 以下となっている。パッケージのダウンロードからバイトコンパイルの順序で進行していき、バイトコンパイルが開始される際、Emacs 上で開いている全ての未保存ファイルを保存するか問われる。バイトコンパイルが終了するとインストール完了である。

尚、インストールされた拡張機能は手動インストールの際とは異なり、直ちに利用可能にはなっていない。利用するためには ELPA からインストールした拡張機能を読み込むためのコマンドを実行する必要がある。そのコマンドが package-initialize なのである。init.el への記述は Emacs 起動時に一度だけ実行されるだけなので、Emacs の再起動をすることなく拡張機能を読み込むためには次のコマンドを実行する。

```
M-x package-initialize
```

これで先程インストールした htmlize の機能が利用可能となる。

package.el を用いてインストールした拡張機能には初期設定ファイルが同梱されており、基本的には init.el に設定を記述することなく利用可能となっている（但し、全てではない）。詳しくは後述の「インストール先のディレクトリ構成と設定の仕組み」で解説する。

コマンドから直接インストールする

次はコマンドからインストールする方法を解説する。例として、Emacs をターミナルとして利用可能とする multi-term という拡張機能をインストールしてみる。M-x list-packages によって一度パッケージリストを取得している場合は次のコマンドによってインストールすることが可能である。

```
M-x package-install RET multi-term RET
```

これは次の S 式を評価するのと同じである。

```
(package-install 'multi-term)
```

先程と同様に M-x Package-initiliza を実行すると、直ちに multi-term の機能が利用可能となる。multi-term については第 7 章で解説する。

ファイルやバッファからインストールする

package.el には、ダウンロードしたファイルやバッファに記述されているコードからインストールする機能も用意されている。次のコマンドを実行すると、指定した Elisp ファイルをインストールする。

```
M-x package-install-file RET Elisp ファイルのパス名 RET
```

また、インストールしたい Elisp コードを表示した状態で M-x package-install-from-buffer を実行すると、バッファから Elisp をインストールすることが可能である。

インストール先のディレクトリ構成と設定の仕組み

手動インストールの場合とは異なり、Elisp パッケージマネージャを利用してインストールした拡張機能は、基本的に init.el に設定を記述する必要はない。package-initialize を実行するだけで設定が済むようになっている。どのような仕組みで実現されているか理解するために、以下で解説する。

package.el によってインストールされる拡張機能は ~/.emacs.d/elpa 以下へとインストールされ、ELPA からインストールされる拡張機能は「パッケージ名-バージョン」という形式のディレクトリ以下にインストールされる。そして、拡張機能本体以外に「パッケージ名-pkg.el」と「パッケージ名-autoloads.el」という 2 つのファイルも併せてインストールされる。設定の自動化は「パッケージ名-autoloads.el」によって実現されている。「パッケージ名-autoloads.el」の正体は、その名の通りパッケージのコマンドを自動的に読み込む設定であり、autoload 関数を利用する方法で記述されている。「パッケージ名-pkg.el」はパッケージのバージョンが定義されているファイルで、package.el がパッケージのバージョンを管理するために利用される。

6.2 テーマ

第 5 章において装飾を変更するフェイスについて解説したが、Emacs には背景色や文字色などの装飾を設定したテーマと呼ばれる仕組みが用意されている。

6.2.1 テーマの変更：load-theme

テーマを変更するには、`M-x load-theme RET` を実行する。すると、ミニバッファで「Load custom theme:」と問われるので、利用したいテーマ名を入力する。勿論、TAB による補完入力と候補一覧も利用可能である。テーマ名を入力して RET を押すと、直ちにテーマが適用され Emacs の見た目が変化する。テーマを解除したい場合には `M-x disable-theme RET 解除したいテーマ名 RET` を実行する。解除したいテーマ名の入力において TAB を押すと現在適用されているテーマが補完される。

インストールする

デフォルトで用意されている意外のテーマを利用したい場合には、ELPA からインストールすることが可能である。テーマを探す際は「Emacs Themes」[*2]を参考にするとよいだろう。

例えば、Emacs で人気のある zenburn-theme[*3] をインストールするには次のコマンドを利用する。

```
M-x package-install RET zenburn-theme RET
```

外部からインストールしたテーマを load-theme コマンドから適用しようとした場合、読み込み確認と次回から確認なしに読み込みを行うかどうか問われる。特に問題がなければ、両方とも yes と答えればよいだろう。但し、2 回目の問いで yes と答えても次回のコマンド実行時に読み込みの確認が行われないだけで、Emacs を再起動するとテーマが適用されていない状態で起動する。

テーマ選択を保存する

次回からも同じテーマを利用したい場合には、次の設定を init.el に追記する。

```
;; zenburn テーマを利用する。
(load-theme ‘zenburn t)
```

これで Emacs の起動時に自動的にテーマが適用されるようになる。

6.3 統一したインタフェースでの操作

Emacs は様々な操作が可能であり、その全てを覚えるには相当な学習コストが必要となる。ファイルを開く、バッファを選択するなど似たようなアクションを同じ操作で実現することができれば、学習コストを下げることが可能となる。最近の Emacs では Helm を利用することによって統一されたインタフェースによる操作が行えるようになっている。

[*2] https://emacsthemes.com
[*3] https://spacemacs.org

6.3.1 候補選択型インタフェース：Helm

Helm はさまざまな操作を 1 つのインタフェースで実現してしまう便利な拡張機能で、その操作性も極めてシンプルである。Helm が提供する機能は次の通りである。

① ソースと呼ばれる情報源に基づいた候補のリストアップ
② インクリメンタルな候補の絞り込み
③ アクションの実行

これらの一連の操作は、さまざまな Emacs の操作に応用が効く。例えば、バッファを切り替える際、コマンドを実行する際、ドキュメントを検索する際など、通常の操作に比べて視認性に優れ、また多くのケースでタイプ数を節約することが可能となる。

インストールする

MELPA リポジトリを追加している状態の Emacs であれば、Helm のインストールは非常に簡単である。次のようにコマンドを実行すると Helm がインストールされる。

```
M-x package-install RET helm RET
```

Helm は Helm 本体以外に Helm extensions（以下 Helm 拡張と記す）と呼ばれるパッケージによって機能追加を行うことが可能となっている。`M-x list-packages` から興味のある Helm 拡張が存在すればインストールしてみるとよいだろう。

利用可能にする

Helm を利用するための設定は非常にシンプルである。

```
;; Helm を利用可能にする。
(require 'helm-config)
```

基本的には、これのみで Helm の利用を開始することができる。

ファイルを開く・バッファを切り替える

Emacs 上でファイルとバッファの違いは、現実的には Emacs で開いているかどうかでしかなく（開いているものがバッファ、開いていないものがファイル）ファイルを開くことも、バッファを切り替えることも表示をアクティブにしたいという意味では同じである。

まずは、次のコマンドを実行してみる。

```
M-x helm-for-files
```

すると、バッファのリストと最近開いたファイルのリスト、そしてカレントディレクトリのファイルリストが表示される。helm-for-files を実行した状態で文字をタイプすると、その文字にマッチするファイルとバッファのみが絞り込まれる。<up>、<down> もしくは `C-n`、`C-p` によって行を移動することができ、RET を押すと選択中のファイルもしくはバッファがカレントバッファとなる。

Helm 実行中に TAB もしくは `C-i` を実行すると、アクションというコンテキストメニュー（右クリックメニュー）のようなメニュー選択でき、デフォルト以外の処理（例えば、バッファを閉じる）を選択できるようになる。また、Helm 実行中に `C-SPC` を実行すると現在行をマークすることができる。これを活用することで、一度に複数の項目を処理の対象にすることが可能となる。

キーバインドを一覧表示する：helm-descbinds

ここからは、バッファやファイルを操作する以外の便利な Helm 拡張を紹介していく。

helm-descbinds をインストールした上で (helm-descbinds-mode) という設定を追加していると、`C-h b` によるキーバインド一覧表示が Helm インタフェースに置き換わる。関数名やキーバインドから絞り込むことができ、表示がとても見易いのが特徴である。helm-descbinds を起動して、例えば「mark」と入力すると、コマンド名に「mark」を含むキーバ

インドだけを見つけることができる。そのまま選択してコマンドを実行することも可能である。

過去の履歴からペーストする：helm-show-kill-ring

Emacs は過去に C-k や C-w で消去した文字をキルリングと呼ばれる場所に保存している。M-x helm-show-kill-ring を利用すると、キルリングを一覧表示することができるため、標準の C-y、M-y よりも格段に使い勝手に優れる。

helm-show-kill-ring が気に入ったのであれば、次の設定を追記ればよい。

```
;; M-y に helm-show-kill-ring を割り当てる。
(define-key global-map (kbd "M-y") 'helm-show-kill-ring)
```

moccur を利用する：helm-c-moccur

helm-c-moccur をインストールして、以下のように設定すると C-M-o を実行することで helm-c-moccur-by-moccur というコマンドが実行される。

```
(when (require 'helm-c-moccur nil t)
  (setq
   helm-idle-delay 0.1
   ;; helm-c-moccur 用 'helm-idle-delay'
   helm-c-moccur-helm-idle-delay 0.1
   ;; バッファ情報をハイライトする。
   helm-c-moccur-highlight-info-line-flag t
   ;; 現在選択中の候補の位置を他の Window に表示する。
   helm-c-moccur-enable-auto-look-flag t
   ;; 起動時にポイント位置の単語を初期パターンにする。
   helm-c-moccur-enable-initial-pattern t)
   ;; C-M-o に helm-c-moccur-occur-by-moccur を割り当てる。
   (global-set-key (kbd "C-M-o") 'helm-c-moccur-occur-by-moccur))
```

コマンドを実行するとミニバッファに入力を求められるので、検索したい文字列をタイプすることでマッチする行がリストアップされる。尚、この検索には正規表現も用いることができる。

6.4 入力の効率化

昨今ではブラウザ上でテキスト入力を行うことが増えてきたが、最もテキストを入力するソフトウェアはテキストエディタであろう。だからこそ、入力を効率化するための準備はしておきたい。

6.4.1 補完入力の強化：Auto Complete Mode

エディタを使いこなす上で検索・置換と双璧を成す機能と言えば、補完機能であろう。Emacs の補完機能には以前から略語展開という機能が備わっていたが、ここ数年で IDE を超える高機能な補完機能を追加する Auto Complete Mode（以下、auto-complete と記す）という拡張機能が登場した。

インストールする

これも ELPA からインストール可能である。M-x package-install RET auto-complete RET でインストールする。

利用可能にする

auto-complete はマイナーモードをオンにして使用するため、ELPA でインストールした場合も利用には次の設定が必要となる。

```
;; auto-complete の設定
(when (require 'auto-complete-config nil t)
  (define-key ac-mode-map (kbd "M-TAB") 'auto-complete)
```

```
  (ac-config-default)
  (setq ac-use-menu-map t)
  (setq ac-ignore-case nil))
```

auto-complete-config.el を読み込むことで auto-complete も併せて読み込まれる。また、ac-config-default 関数を実行することによってサンプル設定が有効化される。

補完候補をポップアップ＆絞り込む

auto-complete は特殊な操作を必要としない。文字を入力すると自動的に補完候補が表示されるので、自然な流れで利用することができる。補完候補が表示されている状態で C-s を押すと、Emacs のインクリメンタル検索による候補の絞り込みが利用可能となる。この機能は候補が非常に多い場合に非常に有用である。

6.5 検索と置換の拡張

どんなエディタでも検索は頻繁に利用する機能である。Emacs には標準の iserach とは異なった形の便利な検索機能を提供する Elisp が多数存在する。

6.5.1 検索結果のリストアップ：color-moccur

Emacs には標準で occur という検索にマッチした行を一覧表示するコマンドが用意されている。Emacs では検索に grep も利用可能だが、grep はファイルを検索するのに対して occur はファイルとして保存されていないバッファに対しても検索を行うことが可能である。その occur をマルチバッファ対応、操作性や可読性の向上などあらゆる面で利用し易くなるよう開発されたのが color-moccur[*4] である。

インストールする

ELPA から M-x package-install RET color-moccur RET でインストール可能である。

利用可能にする

color-moccur を require 関数で読み込む設定を init.el に追記する。

```
;; color-mocuur を利用するための設定
(when (require 'color-moccur nil t)
  ;; M-o に occur-by-moccur を割り当てる。
  (define-key global-map (kbd "M-o") 'occur-by-moccur)
  ;; スペース区切りで AND 検索する。
  (setq moccur-split-word t)
  ;; ディレクトリ検索のときに除外するファイルを指定する。
  (add-to-list 'dmoccur-exclusion-mask "\\.DS_Store")
  (add-to-list 'dmoccur-exclusion-mask "^#.+#$"))
```

代表的なコマンド

color-moccur に関する代表的なコマンドは表 6.2 の通りである。

表 6.2: color-moccur に関する代表的なコマンド

名前	説明
occur-by-moccur	カレントバッファで moccur を実行する。
moccur	全てのバッファで moccur を実行する。
dmoccur	指定ディレクトリに対して moccur を実行する。
moccur-grep	moccur を用いた grep 検索を行う。
moccur-grep-find	moccur を用いた grep-find を実行する。

[*4] http://www.emacswiki.org/emacs/color-moccur.el

マルチバッファを検索し結果をリストアップする

M-o に割り当てた occur-by-moccur はカレントバッファのみを検索対象とする occur を color-moccur の機能で実現する機能である。*Moccur* 上で行移動すると、分割されたバッファに対応する行の周辺が表示され、RET を押すとその行へジャンプする。moccur を中断するには q を押す。

6.5.2 moccur の結果を直接編集する：moccur-edit

moccur は検索にマッチした行のみをバッファにリストアップする。その結果を直接編集することが可能なら、大量の置換も面倒ではなくなるだろう。そんな要望に応える機能が moccur-edit である。

インストールする

ELPA から次のようにしてインストールする。

```
M-x package-install RET moccur-edit RET
```

利用可能にする

color-moccur をインストールした状態で mocccur-edit を読み込む。

```
;; moccur-edit を利用するための設定
(require 'moccur-edit nil t)
```

moccur の結果を編集する

moccur による検索を実行した後、*Moccur* バッファ上で r を押すことで検索結果を直接編集可能となる moccur-edit モードとなる。moccur-edit モードでは C-c C-u、C-x k、C-c C-k (moccur-edit-kill-all-change) を用いて編集を破棄することが可能である。

望みの場所を編集した後に C-c C-c もしくは C-x C-s (moccur-edit-finish-edit) を押すと、編集箇所がバッファへと反映される。この状態ではまだ編集されたファイルは保存されていないので、編集をファイルに反映させるために C-x s (save-some-buffers) などを用いて保存する。

編集を終了した際にファイルに自動保存する

次の設定を init.el へ追記することで編集の終了と同時にファイルへ保存することが可能となる。

```
;; moccur-edit-finsh-edit と同時にバッファをファイルに保存する
(defadvice moccur-edit-change-file
    (after save-after-moccur-edit-buffer activate)
  (save-buffer))
```

6.5.3 grep の結果を直接編集する：wgrep

大量のファイルを検索する場合、moccur よりも高速に検索を行うことが可能な grep を利用する。こちらの検索結果も moccur-edit と同様に編集したいと思うかもしれない。そのような場合は wgrep[*5] を用いる。

インストールする

ELPA から次のようにしてインストールする。

```
M-x package-install RET wgrep RET
```

利用可能にする

package-initialize では wgrep を読み込まないので、wgrep を読み込む設定を init.el へ追記する必要がある。

```
;; wgrep を利用するための設定
(require 'wgrep nil t)
```

[*5] http://www.emacswiki.org/emacs/wgrep/el

grep の結果を直接編集する

M-x grep、M-x rgrep、M-x lgrep などの grep コマンドから検索すると *grep* バッファに検索結果が表示される。そこで C-c C-p (wgrep-change-to-wgrep-mode) を実行すると、*grep* バッファを編集可能な wgrep-mode になる。後の操作は moccur-edit とほぼ同様となっており、C-c C-k (wgrep-abort-changes) で編集を破棄、C-c C-c (wgrep-finish-edit) で編集をバッファへ反映させる。moccur-edit と同様に編集内容はファイルに保存されないため、別途保存する必要があるが、M-x wgrep-save-all-buffers RET で wgrep によって編集したファイルを一括保存するコマンドが用意されているので、これを利用する。

6.6 さまざまな履歴管理

PC を使って作業する際、前の状態に戻る処理で最もよく利用されるのはアンドゥ（Undo）であろう。アンドゥはアプリケーションの世界で標準的な機能として広く普及しており、なくてはならない機能である。本節では、履歴をより効率的に活用することが可能な拡張機能を幾つか紹介する。

6.6.1 編集履歴の記憶：undohist

エディタでアンドゥを利用すると編集履歴を遡って編集を取り消すことができるのだが、大抵の場合、ファイルを閉じたりエディタを終了してしまうと編集履歴はリセットされてしまう。すなわち、もはや元に戻したいと思ったファイルが既に閉じられてしまった後では、もはやアンドゥを用いても元に戻せなくなる。この問題を解消する拡張機能が undohist である。undohist を導入すると編集履歴をバッファ保存のタイミングで ~/.emacs.d/undohist ディレクトリ以下に保存し続ける。そしてファイルを開く際に編集履歴を読み込み、ファイルを閉じても Emacs を再起動してもアンドゥを利用し続けることが可能となる。

インストールする

ELPA から次のようにしてインストールする。

```
M-x package-install RET undohist RET
```

利用可能にする

undohist を読み込んだ後に undohist-initialize 関数で呼び出す設定を init.el に追記する。

```
;; undohist を利用するための設定
(when (require 'undohist nil t)
  (undohist-initialize))
```

6.6.2 アンドゥの分岐履歴：undo-tree

例えば、編集の途中でアンドゥして編集を再開したが、やはりアンドゥする前の状態に戻りたくなった場合、通常のアンドゥでは戻ることはできない。undo-tree はアンドゥの分岐管情報を記憶し、樹形図で視覚化することにより解り易いアンドゥを提供し、リドゥを実現可能とする。

インストールする

ELPA から次のようにしてインストールする。

```
M-x package-install RET undo-tree RET
```

利用可能にする

undo-tree を読み込んだ後に global-undo-tree-mode によって undo-tree-mode を有効化する。また、リドゥをキーバインドで割り当てたい場合は undo-tree-redo をキーバインドに登録する。

```
(when (require 'undo-tree nil t)
  ;; C-. にリドゥを割り当てる。
  ;; (define-key global-map (kbd "C-.") 'undo-tree-redo)
  (global-undo-tree-mode))
```

樹形図を見ながらアンドゥする

C-x u (undo-tree-visualize) コマンドを実行すると、アンドゥの履歴を視覚化したバッファが表示される。使い方も直感的で、樹形図を移動するとバッファがアンドゥされていくので、戻りたいポイントに来た際に q で抜ければよい。t で樹形図を時間表示と切り替えることも可能である。

6.6.3 カーソルの移動履歴：point-undo

エディタの操作は文字の入力だけではない。カーソル移動も歴（れっき）とした操作である。行数の多いファイルを編集している際、検索や誤ってバッファ先頭や終端まで移動してしまったりとカーソルが自分の意図しない場所へ行ってしまい、先程まで編集していた場所を探すのに苦労したことはないだろうか？ そういったエディタ内迷子も point-undo を利用することで解決することが可能である。point-undo を利用すると、編集中のバッファ内で行われたカーソル移動をメモリに記録し、編集と同じ感覚でカーソル移動をアンドゥ、リドゥすることが可能となる。

インストールする

ELPA から次のようにしてインストールする。

```
M-x package-install RET point-undo RET
```

利用可能にする

point-undo を読み込んだ後、キーバインドを割り当てる。

```
;; point-undo を利用するための設定
(when (require 'point-undo nil t)
  (define-key global-map (kbd "M-[") 'point-undo)
  (define-key global-map (kbd "M-]") 'point-redo))
```

カーソル位置を戻す／やり直す

point-undo を割り当てたキーバインドを実行すると、カーソル位置を 1 つ前の場所に戻すことができる。また、point-redo によってその操作をやり直すことも可能である。

6.7 ウィンドウ管理

第 3 章で解説したが、Emacs には 1 つの画面（フレーム）を横や縦に分割してフレーム中に複数のウィンドウを作成することができる。それによって画面を切り替えることなく、ファイルを見比べたりすることが可能となっていた。この分割状態を維持したまま別の作業をしたいと思った際、新たにフレームを作成することで一応実現可能だが、フレームの数が増えていくと切り替えに時間がかかり、1 つ 1 つのフレームが小さくなり作業効率が低下する。

6.7.1 分割状態を維持する：ElScreen

ウィンドウの分割をしたいがフレーム数は増やしたくないといった要望や、同じバッファの 10 行目付近を表示しているウィンドウと 1,000 行目付近を表示しているウィンドウを交互に切り替えたいなどといった要望を叶えるためには、ウィンドウの状態を保持しておく拡張機能があれば実現できそうである。

ターミナル環境に慣れている者にとって、そういった画面の状態を保持したい場合に真っ先に思いつくのが GNU Screen という仮想端末ソフトウェアであろう。この Screen のような機能を提供してくれる拡張機能が ElScreen である。ElScreen を導入するとウィンドウとフレームの間にスクリーンと呼ばれる仮想フレームのようなものが作成可能となり、ウィンドウ状態を保持したまま切り替えることが可能となる。

インストールする

ELPA から次のようにしてインストールする。

```
M-x package-install RET elscreen RET
```

利用可能にする

ElScreen を読み込むことで利用可能となる。標準では C-z にプレフィックスキーが割り当てられているため、変更したい場合は elscreen-prefix-key に別のキーを割り当てる。

```
;; ElScreen のプレフィックスキーを変更する（初期値は C-z）
;; (setq elscreen-prefix-key (kbd "C-t"))
(when (require 'elscreen nil t)
  (elscreen-start)
  ;; C-z C-z をタイプした場合にはデフォルトの C-z を利用する
  (if window-system
      (define-key elscreen-map (kbd "C-z") 'iconfig-or-deiconfy-frame)
    (defun-key elscreen-map (kbd "C-z") 'suspend-emacs)))
```

スクリーンを作成する

ElScreen という名前の通り Screen の動作を参考にして作成されたこの拡張機能は、操作方法も Screen ライクになっている。Screen を利用したことがある者は違和感なく利用できるようになっている。

標準の状態では C-z が ElScreen の各種コマンドを利用するためのプレフィックスキーとなっている。ElScreen を読み込んだ段階で、既に 0 番目のスクリーンが 1 枚作成されている状態となっている。C-z c (elscreen-create) を実行すると、番号 1 番の新しいスクリーンが作成されて新しいスクリーンに移動する。もう 1 度 C-z c を実行すると番号 2 番のスクリーンが作成され移動する。

スクリーンを移動する

番号 2 番のスクリーンにいる状態で C-z p (elscreen-previous) を実行すると、番号 1 番のスクリーンへ移動する。C-z n (elscreen-next) を実行すると今度は番号 2 番のスクリーンへ移動する。もう 1 度 C-z n を実行すると、まだ番号 3 番のスクリーンが存在しないため番号 0 番のスクリーンへ移動する。ElScreen における代表的なキーバインドを表 6.3 にまとめておく。

表 6.3: Elscreen に関する代表的なコマンド

キー	説明
C-z c	新規スクリーンを作成して移動する。
C-z k	現在のスクリーンを閉じる。
C-z p	前のスクリーンへ移動する。
C-z n	次のスクリーンへ移動する。
C-z a	前と次のスクリーンをトグルする。
C-z 0, C-z 1,...,C-z 9	番号のスクリーンへ移動する。
C-z ?	ヘルプを表示する。

6.8 メモ・情報整理

Emacs はコードを書くためだけのツールではない。テキスト編集全般を強力にサポートするための機能が幾つも備わっている。さて、プログラミング以外で PC に向かって文章を書くのはどういった場合だろうか？ メモを取るため、ブログの記事を書くため、スケジュール管理をするため、簡単な計算をするためなどが考えられる。これらの文書を書く際、導入しておくと便利なのが（ファイル名や記述スタイルなどの）フォーマットである。予め決められたフォーマットに従ってマークアップ（意味付け）をしておくだけで、様々な恩恵を受けることができる。

6.8.1 メモ書き・TODO 管理：howm

howm (*Hitori Otegaru Wiki Modoki*) は作者の紹介によると次のようなツールである。

> Emacs で断片的なメモをどんどんとるための環境です。分類機能はあえてつけていません。かわりに、全文検索とメモ間リンクが手軽にできるようにしました。自由形式なので改宗も不要 :-)
> ——「howm:Hitori Otegaru Wiki Modoki（http://howm.sourceforge.jp/index-j.html）」

howm を利用すると通常のテキストに次の機能を追加することが可能となる。

- Wiki のようなリンク機能
- TODO リスト機能

主な機能としてはこれだが、これらの機能を組み合わせて便利にメモを取ることが可能となる。

インストールする

ELPA から次のようにしてインストールする。

```
M-x package-install RET howm RET
```

利用可能とする

howm を使い始めるための設定は次のとおりである。

```
;; hown メモ保存の場所を指定する。
(setq hown-directory (concat user-emacs-directory "howm"))
;; howm-menu の言語に日本語を設定する。
(setq howm-menu-lang 'ja)
;; howm メモを 1 日 1 ファイルにする。
(setq howm-file-name-format "%Y/%m/%Y-%m-%d.howm")
;; howm-mode を読み込む。
(when (require 'howm-mode nil t)
  ;; C-c,, で howm-menu を起動する。
  (define-key global-map (kbd "C-c ,,") 'howm-menu))
```

howm-directory は howm がメモを保存するディレクトリである。標準では ~/howm となっているが ~/.emacs.d 以下に集約した方がバックアップしやすいだろう。

howm-menu を起動する

howm の利用方法は極めてシンプルである。メモを取りたい際に howm-menu を起動して、好きなだけメモを取るというスタイルである。M-x howm-menu もしくは C-c , , を実行すると howm-menu と呼ばれるバッファが表示される。メニューの最上段にはショートカットキーが表示されているので、すぐに使い方を覚えることができるだろう。c を押すと新規のメモを作成することができる。

メモを書く

例えば、init.el を編集中にメモファイルを作成すると、次のようなテキストが自動的に挿入される。

```
=
[2020-2-17 21:13] >>> ~/.emacs.d/init.el
```

= から始まる行は howm 内でメモのタイトルとして使用される。>>> ~/.emacs.d/init.el の部分はファイルへのリンクとなっており、RET を押すことで *howmS* バッファが開き、移動可能なファイル一覧が表示される。この場合は、>>> ~/.emacs.d/init.el を含むメモと init.el が対象となる。

実際にメモを書いてみると以下のようになる。

```
= howm に関する設定を追加してみた
[2020-2-17 21-26] >>> ~/.emacs.d/init.el

http://www.bookshelf.jp/soft/meadow_38.html#SEC563
こちらの記事が大変参考になりました。
```

という形でメモを取って保存した後に、再び C-c , , で howm-menu を起動すると「最近のメモ」という項目に先頭のメモが追加される。

保存と同時に閉じる

頻繁にメモを取るようになると「作業→メモを取る→作業に戻る」という具合に、流れるようにメモを取りたくなる。メモを取った後、保存と同時にメモを閉じられると便利である。次の設定を追記することで、それを実現することが可能となる。

```
;; howm メモを保存と同時に閉じる。
(defun howm-save-buffer-and-kill ()
  ; howm メモを保存と同時に閉じる。
  (interactive)
  (when (and (buffer-file-name)
             (howm-buffer-p))
  (save-buffer)
  (kill-buffer nil)))
;; C-c C-c でメモの保存と同時にバッファを閉じる。
(define-key howm-mode-map (kbd "C-c C-c") 'howm-save-buffer-and-kill)
```

howm には通常のメモ以外にも予定表や TODO 管理機能などさまざまな機能が用意されているので、より詳しく知りたい者は公式サイト*6のドキュメントを参照すること。

6.8.2 アウトラインエディタ：org-mode

org-mode は Emacs 23 から同梱されたアウトラインエディタである。簡単な入力規則に従って記述することで、豊富な機能を提供してくれる。org-mode は Emacs に標準で同梱されているため、インストールの必要はない。拡張子 org のファイルを開くと自動的に org-mode となる。

アウトライン編集機能

まずは memo.org のようなファイルを作成してみる。すると Emacs は自動的に org-mode となる。org-mode 上では、次のような様々なアウトライン編集機能が利用可能である。

```
* 見出し 1
** 見出し 2
*** 見出し 3
**** 見出し 4
### 箇条書きなどを入力する
```

見出しの上で TAB を押すと、見出しの中を折り畳むことができる。- や + そして 1. などの番号を行頭に付加するとリストとなる。リストを続けて入力する際、M-RET (org-insert-heading) という便利なキーバインドが用意されている。これは現在行と同じリスト項目を次の行に挿入してくれる。

表を入力する

テキストで表を入力することは多くないかもしれないが、メールなどで簡単な表を用いて説明したい場合などがある。そのような場合は org-mode の表入力機能を用いる。|A|B|C| と入力して TAB を押すと次のように展開される。

```
| A | B | C |
| _ |   |   |
```

_ の位置がカーソル位置である。org-mode では | で開始された行は自動的に表として扱われ、表編集機能になる。この中で TAB は次のフィールドへ移動するためのキーとなっている。勿論 <backspace> で前のフィールドへ移動することができる。RET は行を追加する。

*6 http://howm.sourceforge.jp/index-j.html

次は、|- と入力して TAB を押してみる。

```
| A | B | C |
|- ← ここで TAB を押す。
```

すると、水平線が挿入される。

```
| A | B | C |
|---+---+---|
|   |   |   |
```

org-mode には、この他にも HTML や TEX などに出力する機能など、多くの機能が備わっている。興味のあるものはドキュメント[*7]を参照すること。

6.9 特殊な範囲の編集

テキスト編集の際に複数行に渡ってカラム単位の編集を加えたい場合がある。しかし、通常の範囲選択は行単位で行われるため、一部のカラムのみを選択することはできない。だが、Emacs には表計算ソフトのようにカラムのみを選択し編集するための機能が備わっている。それが矩形編集機能である。

6.9.1 矩形編集：cua-mode

矩形（*rectangle*）編集は Emacs に限らず大抵のエディタに備わっている機能だが、知らない者はどのエディタを使っていてもこの機能を知らない。しかし、知っている者にとってはなくてはならない機能の 1 つである。

勿論 Emacs にも標準で矩形編集機能が備わっているが、少々キーバインドが特殊で覚えるのに苦労する。そこで、矩形編集機能を更に強化してキーバインドも簡略化した cua-mode を利用してみる。これで誰でも矩形編集を自由自在に扱うことができるようになるだろう。

利用可能にする

cua-mode は本来 Emacs 上で CUA（*Common User Access*）キーバインド（C-z でアンドゥ、C-c でコピーなど）を利用するためのマイナーモードであるが、実際にはこれらのキーバインドは Emacs のキーバインドに慣れ親しんだ者には邪魔であることが多い。しかし、cua-mode には使い勝手の良い矩形編集モードが備わっている。そこで、cua-mode を利用しつつ CUA キーバインドをオフにする設定を施す。

```
;; cua-mode の設定
(cua-mode t) ; cua-mode を有効化する。
(setq cua-enable-cua-keys nil) ; CUA キーバインドを無効化する。
```

各行頭に文字を追加する

cua-mode が有効化された状態で用いるキーバインドは C-RET (cua-set-rectangle-mark) である。例えば、次のようなテキストがあったとする。

```
List1
List2
List3
```

この行頭に - を追加したい場合、List1 の行頭にカーソルを置いた状態で C-RET をタイプする。するとピンク色のカーソルが表示され、矩形編集モードとなる。この状態で List3 の行頭までカーソルを移動する。そして - を入力すると 3 行全ての行頭に - が挿入される。

```
- List1
- List2
- List3
```

[*7] https://github.com/org-mode-doc-ja/org-ja/tree/master/work

C-g をタイプして矩形編集モードを終了する。因みに、Emacs 標準の矩形編集機能を用いて同じことを実現する場合は、List1 の行頭でマークして List3 の行頭へカーソル移動し、C-x r t (string-rectangle) とタイプしてからミニバッファに - を入力して RET する。

各行頭の文字列を削除する

続いて List の文字を削除する。これも矩形編集機能の出番である。List1 の L にカーソルを置いて、再び C-RET をタイプする。そして、List3 の t にカーソル移動する。ピンク色のエリアに List の文字が囲まれていることを確認して C-w をタイプする。すると、List の文字が全て切り取られる。

```
- 1
- 2
- 3
```

尚、これを標準の矩形編集機能で実現するには List1 の L の位置でマークして List3 の 3 の位置までカーソル移動し、C-x r d (delete-rectangle) をタイプする。

連番を入力する

ここまで挿入と切り取りの 2 つの矩形編集を紹介したが、次に cua-mode ならではの機能を紹介する。ここでは M-o と M-n を覚えておく。M-o (cua-open-rectangle) は選択範囲の右に向かって半角スペースを 1 つ追加するコマンドである。また、M-n (cua-sequence-rectangle) は連番入力を行うためのコマンドである。これらを組み合わせて、次の行頭に 1. 2. のようなナンバリングを付加してみる。

```
ListA
ListB
ListC
ListD
```

まず、ListA の行頭でマークして、次に ListD の行頭にカーソル移動して C-RET をタイプする。cua-mode の矩形編集機能は、先に通常のマークをしてから C-RET を実行してもマーク範囲を編集することができるようになっている。

それでは M-o をタイプする。すると行頭にスペースに 1 文字追加される。そして M-n をタイプすると、ミニバッファに「Start value: (0)」という文字が表示され、入力を求められる。これは連番入力の初期値である。デフォルトは 0 になっているが、今回は 1 からなので 1 を入力して RET する。次は「Increment: (1)」と表示され入力を求められる。これは 1 行毎に加算される値である。例えば 3 と入力すれば 1 行毎に 3 が加算されるが、今回は 1 でよいので何も入力せず RET を押す。最後に「Format: (%d)」と表示され入力を求められる。ここで挿入される文字列フォーマットを決めることが可能である。%d は数値を出力するので、今回のように「1.」と出力したければ %d.と入力して RET を押す。すると、全ての行が順番にナンバリングされる。

```
1. ListA
2. ListB
3. ListC
4. ListD
```

最初に M-o によってスペースを 1 文字追加したのは、M-n が挿入ではなく、現在選択中の部分を置換するからである。

第 7 章

開発を更に効率化する拡張機能

7.1　各種言語の開発環境

Emacs はテキストエディタなので、種類を問わずテキストを書く全ての者のためのソフトウェアである。しかし、Emacs というソフトウェアに巡り合った多くの者は開発のための機能を Emacs に求めているのではないだろうか？ そこで、締めくくりとなる本章では開発に主眼を置いた Emacs の利用方法を紹介する。

まずは各種言語に関する設定である。Emacs には様々な言語を扱うためのメジャーモードとマイナーモードが用意されている。Emacs 本体に同梱されているものからインストールが必要なものまで様々存在し、また同一言語でも複数のメジャーモードが用意されている。本稿では多くの者に支持されている機能を中心に紹介していく。

7.1.1　Web 開発

Web 開発は HTML、CSS、JavaScript などのフロントエンド技術や PHP や Ruby などのサーバサイド技術などが複合していることが一般的である。そのため、1 つのファイル内で複数の言語が入り乱れることもある。

Emacs は第 2 章で解説した通り、バッファに対して 1 つのメジャーモードが適用される仕組みとなっているため、1 つのファイルの中に複数の言語が書かれている場合、シンタックスハイライトやインデントなどがうまく機能しない問題が存在していた。

しかし、ここで紹介する web-mode は 1 つのメジャーモードで複数の言語をうまく扱う仕組みが用意されているため、web-mode を利用することで、これまで解決が難しかったこれらの問題を回避することができる。

勿論、それぞれのメジャーモードには専用のコマンドが用意されているため、全てのケースにおいて必ずしも web-mode が最適の選択であるとは限らないが、とりあえずファイルを開いて編集するだけの目的であれば最適なメジャーモードであろう。そのため、まず web-mode を紹介した後に各言語別のメジャーモードを紹介していくので、自分の利用環境に応じて適切な選択を行うこと。

web-mode

web-mode が提供する機能は主に次のようになっており、HTML や JavaScript、PHP や Ruby などの言語が混在する環境でも適切に動作するのが最大の特徴となっている。

- 複数言語の同時編集
- 自動インデント
- HTML タグの自動挿入（主に閉じタグ）
- シンタックスハイライト
- コメントアウト／アンコメント
- コードの折り畳み

web-mode は ELPA から次のようにしてインストールする。

```
M-x package-install RET web-mode RET
```

基本となる編集については特に意識することなく使えるようになっているが、表 7.1 のキー操作を覚えることで、より編集が楽になる。

表 7.1: web-mode における代表的なコマンド一覧

キー	説明
M-;	カーソル行およびブロックをコメントアウト／アンコメントする。
C-c C-n	開始タグおよび閉じタグへカーソルを移動する。
C-c C-f	タグ内の文字列を折り畳む／折り畳みを解除する。
C-c C-i	カーソル行もしくはリージョンをインデントする。
C-c C-e r	カーソル付近の HTML 開始タグと閉じタグを一括リネームする。
C-c C-e /	閉じタグを挿入する。
C-c C-s	スニペットを挿入する。
C-c C-d d	HTML タグの文法をチェックする。

web-mode の基本設定

特定のファイルを自動的に web-mode で開きたい場合、次の設定を init.el に追記する。尚、インデントは標準でスペース 2 つ分に設定されている。

```
(when (require 'web-mode nil t)
  ;; 自動的に web-mode を起動したい拡張子を追加する。
  (add-to-list 'add-mode-alist '("\\.html\\'" . web-mode))
  (add-to-list 'add-mode-alist '("\\.css\\'" . web-mode))
  (add-to-list 'add-mode-alist '("\\.js\\'"  . web-mode))
  (add-to-list 'add-mode-alist '("\\.jsx\\'" . web-mode))
  (add-to-list 'add-mode-alist '("\\.tpl\\'" . web-mode))
  (add-to-list 'add-mode-alist '("\\.php\\'" . web-mode))
  (add-to-list 'add-mode-alist '("\\.ctp\\'" . web-mode))
  (add-to-list 'add-mode-alits '("\\.jsp\\'" . web-mode))
  (add-to-list 'add-mode-alist '("\\.asx\\'" . web-mode))
  (add-to-list 'add-mode-alist '("\\.cpx\\'" . web-mode))
  (add-to-list 'add-mode-alist '("\\.erb\\'" . web-mode))
;; web-mode のインデント設定用フックの設定
  (defun web-mode-hook ()
    "Hooks for Web mode."
     (setq web-mode-markup-indent-offset 2)  ; HTML 用のインデント
     (setq web-mode-css-indent-offset    2)  ; CSS  用のインデント
     (setq web-mode-code-indent-offset   2)  ; JS, PHP, Ruby など用のインデント
     (setq web-mode-comment-style        2)  ; web-mode 内のコメントのインデント
     (setq web-mode-style-padding        1)  ; <style>  内のインデント開始レベル
     (setq web-mode-script-padding       1)) ; <script> 内のインデント開始レベル
  (add-hook 'web-mode-hook 'web-mode-hook))
```

7.1.2 HTML

HTML（*HyperText Markup Language*）は文章をコンピュータ処理しやすくするために、タグと呼ばれるマークアップ（意味付け）を行うことによって見出し・箇条書きといった各要素をコンピュータに理解できるようにした言語である。WWW はインターネットによる文書の恒久的なアーカイブを目的として誕生したため、Web 標準に準拠した妥当（Valid）な HTML で記述する限り、例えば Google Chrome がバージョン 100 になり、HTML 10 が誕生したとしても、将来も閲覧することができることが保証されている。マークアップ検証サービスも提供されている。

但し、HTML ソースは人間にとって読み書きしにくいものである。そのような HTML を可能な限り記述しやすく補助してくれるモードを紹介する。

html-mode

Emacs で HTML ファイルを開くと標準で選択されるのが html-mode である。この html-mode は sgml-mode をベースに HTML 特有のタグをサポートしてくれる。基本的な機能としてはタグの入力補助、シンタックスハイライト、インデント補助のみの非常にシンプルな設計となってる。

html-mode を用いてタグを入力する

html-mode ではキーバインドに用意されている HTML タグの入力サポートを利用することができる。基本的なキーバインドを表 7.2 に示す。詳しくは html-mode を選択中に `M-x describe-bindings (<f1>,C-h b)` からメジャーモードのキーバインドを参照すること。

表 7.2: html-mode における基本キーバインド

キー	コマンド名	説明
C-c 1	html-headline-1	h1 要素を挿入する。
C-c 2	html-headline-2	h2 要素を挿入する。
C-c 3	html-headline-3	h3 要素を挿入する。
C-c RET	html-paragragh	p 要素を挿入する。
C-c C-j	html-line	br 要素を挿入する。
C-c C-c o	html-ordered-list	ol 要素と li 要素を挿入する。
C-c C-c u	html-unordered-list	ul 要素と li 要素を挿入する。
C-c C-c l	html-list-item	li 要素を挿入する。
C-c C-c i	html-image	画像のあるアドレスを聞いて img 要素を挿入する。
C-c /, C-c]	sgml-close-tag	カーソルの位置するタグ要素の閉じタグを挿入する。
C-c C-d, C-c DEL	sgml-delete-tag	カーソル位置かその後ろにあるタグを削除する。
C-c C-t	sgml-tag	ミニバッファから聞かれたタグを挿入する。
C-c C-n	sgml-name-char	入力した文字の文字実体参照を挿入する。

html-mode が自動選択される仕組み

標準で html-mode が選択される仕組みは、第 2 章で解説した auto-mode-alist 変数に、

```
("\\.[sx]?html?\\(\\.[a-zA-Z_]+\\)?\\'" . html-mode)
```

という指定があるためである。もし HTML を編集する際に違うモードを利用したければ、init.el に設定を追加して auto-mode-alist 変数の値を書き換える必要がある。例として、nxml-mode を Emacs で標準利用する場合の設定を後述する。

nxml-mode

nxml-mode（*New XML editing mode*）は XML 文章を編集する際に Emacs で標準選択されるモードである[*1]。HTML には XHTML という HTML を XML で記述するマークアップ言語が存在し、XHTML を編集する場合には nxml-mode を利用することができる。

nxml-mode には html-mode のような豊富なタグ入力補助機能は用意されていないのだが、スキーマ（Schema）を利用した構文チェック機能を利用することができる。

通常 XML は XML Schema と呼ばれるスキーマ言語によって論理構造が定義されているが、nxml-mode では RELAX NG[*2] というスキーマ言語を利用して XML 文書構造の定義を持っている。そして、スキーマを利用してリアルタイムに構文をチェックしてくれる。

[*1] xml-mode は nxml-mode のエイリアスとなっている。

[*2] http://relaxng.org

nxml-mode を HTML 編集のデフォルトモードにする

nxml-mode を Emacs で HTML を編集する際のデフォルトモードをしたい場合は、次の設定を init.el に追記する。

```
(add-to-list 'auto-mode-alist
             '("\\.[sx]?html?\\(\\.[a-zA-Z]+\\)0+\\)?\\'" . nxml-mode))
```

これで html-mode が選択されるはずだったファイルは全て nxml-mode が選択されるようになる。但し、全ての HTML ファイルが nxml-mode を利用するようになるため、XML でない HTML を編集する場合には注意が必要となる。その場合は、`M-x html-mode` で個別に切り替えるとよい。もし XHTML しか編集せず、構文チェックを利用したければ nxml-mode を、そうでなければ入力補助機能を利用可能な html-mode を利用するのがよいだろう。

nxml-mode で構文チェックを利用する

nxml-mode では標準で rng-validate-mode（`C-c C-v`でオン／オフが切り替え可能）というリアルタイム構文チェック機能がオンになっている。この構文チェックはロードされたスキーマを利用して行われる。

XML 文書と利用するスキーマの対応は rng-schema-locating-file-alist 変数によって管理されており、XHTML の場合は xhtml.rnc というファイルを利用する。XHTML の場合は xhtmk.rnc というファイルを利用している。スキーマ（RDF、XHTML、XSLT など）は `C-c C-s (rng-set-document-type-and-validate)` からスキーマ名を入力（`TAB` による補完が可能）して切り替えることができる。

rng-validate-mode がオンの状態では、XHTML が妥当であればモードラインに「Valid」妥当でなければ「Invalid」と表示される。また、`C-c C-n (rng-next-error)` というキーバインドが用意されており、エラー箇所にジャンプすることが可能である。エラー箇所にカーソルを乗せると、ミニバッファにエラー内容が表示される（例えば、スキーマに定義されていないタグを利用する「Unkown element」と表示される）。

HTML5 を nxml-mode で編集する

nxml-mode には HTML5 のためのスキーマが標準では用意されていないため、HTML5 を nxml-mode で編集すると section などの HTML5 から登場した要素は「Unknown element」としてエラーとなってしまう。しかし、nxml-mode ではスキーマを追加し独自の XML 文書にも対応させることが可能となっており、HTML5 もスキーマを導入することで構文をチェックすることができる。

HTML5 のスキーマは ELPA から次のようにしてインストールすることが可能である。

```
M-x package-install RET html5-schema RET
```

これで nxml-mode によって HTML ファイルを編集する場合、HTML5 のスキーマが自動選択されるようになる。

nxml-mode の基本設定

nxml-mode では html-mode のようなタグ入力のキーバインドは用意されていない。しかし、タグ補完などの機能が用意されているので、次の設定によってそれらを活用することが可能である。

```
;; </ を入力すると自動的にタグを閉じる。
(setq nxml-slash-auto-complete-flag t)
;; M-TAB でタグを補完する。
(setq nxml-bind-meta-tab-to-complete-flag t)
;; nxml-mode で auto-complete-mode を利用する。
(add-to-list 'ac-modes 'nxml-mode)
```

また、インデントの幅も調節可能である。インデント幅を変更したい場合は次の設定を追記する。

```
;; 子要素のインデント幅を設定する（初期値は 2 ）。
(setq nxml-child-indent 0)
;; 属性値のインデント幅を設定する（初期値は 4 ）。
(setq nxml-attribute-indent 0)
```

7.1.3 CSS

HTML が文書を構造化するための言語であるのに対して、CSS（*Cascading Style Sheets*）は文書に表現（レイアウトや文字修飾）を与えるために誕生した言語である。

Emacs にはプロパティ名などを入力ミスしないよう手助けしてくれる機能が用意されている。また、CSS には拡張機能が施された LESS[*3] と Sass (SCSS)[*4] という 2 つの代表的な言語が存在する。Emacs には標準で css-mode というメジャーモードが用意されており、CSS と SCSS に対応している。最初に紹介した web-mode も CSS と SCSS に対応してるため、この 2 つに関しては基本的にどちらかを利用するとよいだろう。css-mode と web-mode が対応していない LESS と Sass を編集する場合には次に紹介するメジャーモードをインストールしておく。

less-css-mode

LESS を編集する場合は less-css-mode を利用する。ELPA から次のようにしてインストールする。

```
M-x package-install RET less-css-mode RET
```

ELPA からインストールすると、特に設定する必要がなく less ファイルを開くと less-css-mode が選択され、シンタックスハイライトやインデントが適用される。lessc コマンドがインストールされている環境であれば、less-css-compile コマンドによって LESS ファイルを CSS にコンパイルすることも可能である。

sass-mode

Sass を編集する場合は sass-mode を利用する。こちらも ELPA から次のようにしてインストールする。

```
M-x package-install RET sass-mode RET
```

こちらもインストールするだけで sass ファイルを開くと sass-mode が選択され、シンタックスハイライトなどが適用される。

7.1.4 JavaScript

JavaScript は近年の Web 開発において最重要言語と言ってよいだろう。HTML、CSS によって作成された Web ページのインタラクティブな操作や使いやすい UI を提供し、非同期通信による動的な処理を実現するなど、今や Web 制作を行う上で決して避けては通れない言語となっている。そんな JavaScript を Emacs で快適に編集するための設定を紹介する。

標準の js-mode

Emacs で JavaScript を編集するためにはには、どのメジャーモードが適切だろうか？ 数年前まではこの問いに答えるのは困難であった。しかし、Emacs 23.2 から js-mode が本体に同梱され、JavaScript ファイル（jsファイル）を開くとメジャーモードとして選択される。js-mode はそれまで espresso-mode と呼ばれていた拡張機能で、数ある JaveScript のメジャーモードの中でも最も構文解析に優れていた。

js-mode で追加で設定することは殆どないが、もし、コーディング規約などによってスペース 2 つのインデントに変更したい場合は以下の設定を init.el に追記すればよい。

```
(defun js-indent-hook ()
  ;; インデント幅を 4 にする。
  (setq js-indent-level 2
        js-expr-indent-offset 2
        indent-tabs-mode nil)
  ;; switch 文の case ラベルをインデントする関数を定義する。
  (defun my-js-indent-line ()
    (interactive)
```

[*3] http://lesscss.org

[*4] http://sass-lang.com

```
    (let* ((parse-status (save-excursion (syntax-ppss (point-at-bol))))
           (offset (- (current-column) (current-indentation)))
           (indentation (js--proper-indentation parse-status)))
      (back-to-indentation)
      (if (looking-at "case\\s-")
          (indent-line-to (+ indantation 2))
        (js-indent-line))
      (when (> offset 0) (forward-char offset))))
  ;; case ラベルのインデント処理を設定する。
  (set (make-local-variable 'indent-line-function) 'my-js-indent-line)
  ;; ここまで case ラベルを調整する設定。
  )
;; js-mode の起動時に hook を追加する。
(add-hook 'js-mode-hook 'js-indent-hook)
```

これで switch 文の case ラベルも 2 スペースのインデントになるはずである。もし、case ラベルのインデントが不要であれば my-js-indent-line 関数の定義以下の設定は必要ない。

構文チェック機能を備えた js2-mode

js-mode が標準で同梱される前から人気の高かったメジャーモードに js2-mode がある。js2-mode は Google の Steve Yagge 氏が開発したメジャーモードで、その最大の特徴として Rhino[*5] という Java で記述された Javascript 実装を Elisp に移植することで独自の構文チェックを備えている。

js2-mode をインストールする

js2-mode は ELPA からインストールすることが可能である。

```
M-x package-install RET js2-mode RET
```

JavaScript ファイルを開いた際に自動的に js2-mode を適用するためには、init.el に次の設定を追記する。

```
(add-to-list 'auto-mode-alist '("\\.js\\'" . js2-mode))
```

7.1.5 PHP

ここまで紹介してきた HTML、CSS、JavaScript は主に Web 開発の中でもフロントエンド周りを開発するための言語である。次はサーバサイドに移ることにする。サーバでは Web サーバやデータベースサーバなどが動作しており、それらとフロントエンドを繋ぐのが Perl、Ruby、Python、PHP などのスクリプト言語である。それぞれに特徴のある言語であり、どれを学べばいいのかは本稿の趣旨とは異なるため語ることはしないが、どの言語も Emacs だけで開発が行うことができるという点は共通している。

本項では PHP について紹介していく。PHP は別名テンプレート言語と呼ばれるほど HTML と親和性が高く、<?php ?> という開始タグと終了タグに囲まれた部分以外は HTML をそのまま記述することができる。正確には、PHP は開始タグと終了タグを用いることで、どんなドキュメントにも埋め込むことが可能である。

php-mode

Emacs 25.2 では標準で PHP 用のメジャーモードが同梱されていない。そのため、自身でインストールすることが必要となる。ELPA から次のようにしてインストールする。

```
M-x package-install RET php-mode RET
```

MELPA からインストールすると、php ファイルを開いた際に自動的に php-mode が適用される。また、設定次第で関数などの補完機能も利用可能となるが、第 6 章で紹介した auto-complete を導入していれば自動補完が適用され、こちらの方が php-mode に付属する補完機能よりも有用であるため、ここではあえて紹介することはしない。

[*5] http://www.mozilla.org/rhino

オンラインドキュメントを利用する

php-mode にはシームレスにオンラインドキュメントを利用する機能が備わっている。標準では英語のドキュメントを利用するようになっているが、php-manual-url 変数にシンボル 'ja を設定すると日本語のドキュメントを利用するようになる。本稿執筆時点ではベースとなる PHP サイトの URL が古いようなので https を用いる URL に変更しておくとよいだろう。

```
;; 日本語ドキュメントを利用するための設定
(when (require 'php-mode nil t)
  (setq php-site-url "https://secure.php.net/"
        php-manual-url 'ja))
```

関数のリファレンスを調べたい場合は、調べたい関数にカーソルを合わせて C-c C-f (php-search-documentation) を実行するだけで、ブラウザでオンラインリファレンスを参照することができる。ドキュメントを開きたい場合は C-c RET (php-browse-manual) で直ちにブラウザでドキュメントのページを開くことができる。

php-mode のインデントを調整する

第 5 章でも少々触れたが php-mode のインデントは c-mode のインデントを利用している。従って、PHP によくあるコーディング規約に従う必要がある場合は、次の設定を追加することでインデントをそろえることができる。

```
;; php-mode のインデントの設定
(defun php-indent-hook ()
  (setq indent-tabs-mode nil)
  (setq c-basic-offset 4)
  (c-set-offset 'case-label    '+)  ; switch 文の case ラベル
  (c-set-offset 'arglist-intro '+)  ; 配列の最初の要素が改行した場合
  (c-set-offset 'arglist-close  0)) ; 配列の閉じ括弧
(add-hook 'php-mode-hook 'php-indent-hook)
```

c-basic-offset 変数がインデントを行う際の基本文字数となっているので、もし 2 文字にしたい場合は、

(setq c-basic-offset 4) の部分を (setq c-basic-offset 2)

とすればよい。

7.1.6 Perl

Perl は現在のようなフレームワークによる Web アプリケーション開発が一般的になるずっと前、CGI と呼ばれる掲示板やメールフォームしかなかった時代から Web 開発の第一線で使用されてきたスクリプト言語である。

様々なプログラミング言語の中でも、特に Perl 開発者には Emacs を利用している者が多く存在する。その理由としては Perl には TMTOWTDI（ティムトゥディ）（*There's More Than One Way To Do It*：やり方は色々ある）というポリシーがあり、非常に柔軟な書き方が可能な言語となっているためである。そのため、インデントやシンタックスハイライトを行うための構文解析も複雑で、それらに早くから対応していた Emacs を選択する者が多かったのではないかと考えられる。

cperl-mode

Emacs には Perl を編集するためのメジャーモードが最初から 2 つ用意されている。1 つは perl-mode であり、もう 1 つは構文解析を C 言語に似せた cperl-mode である。どちらを利用すればよいか悩むところだが、cperl-mode を利用している者が多いようである。本稿でも cperl-mode について解説していく。

cperl-mode は最初から Emacs に同梱されているためインストールは不要である。但し、標準では pl ファイルを開くと perl-mode が自動選択されるため、cperl-mode を利用するなら cperl-mode が自動選択されるように設定する必要がある。cperl-mode を必ず利用するというのであれば、エイリアス（*Alias*）という仕組みを利用するとよいだろう。コンピュータにはエイリアスというファイルやコマンドなどに別名を付けて呼び出す仕組みが備わっているが、Elisp でも関数や変数にエイリアス（別名）を付けることができる。

これを利用することで、perl-mode を cperl-mode の別名にしてしまうことができる。

設定するためのコードは非常に簡単で、defalias 関数に別名と実際に実行する関数の 2 つを引数として渡すだけでよい。

```
;; perl-mode を cperl-mode のエイリアスに設定する。
(defalias 'perl-mode 'cperl-mode)
```

この設定を init.el に追記して読み込むと、以後 perl-mode が実行されると全て cperl-mode が呼び出されるようになる。モードラインに「CPerl」と表示されていれば、cperl-mode が正しく呼び出されていることを確認するすることができる。

cperl-mode のインデントを調整する

cperl-mode においても細かくインデントを調整することができる。標準のインデントが気に入らない場合は、次のサンプルを参考にして自身に合ったインデントを設定すればよい。

```
;; cperl-mode のインデント設定
(setq cperl-indent-level               4 ; インデント幅を 4 にする
      cperl-continued-statement-offset 4 ; 継続する文のオフセット
      cperl-brace-offset              -4 ; ブレースのオフセット
      cperl-label-offset              -4 ; ラベルのオフセット
      cperl-indent-parens-as-block     t ; 括弧もブロックとしてインデントする
      cperl-close-paren-offset        -4 ; 閉じ括弧のオフセット
      cperl-tab-always-indent          t ; TAB をインデントにする
      cperl-highlight-variables-indiscriminately t) ; スカラを常にハイライトする
```

yaml-mode

プログラミングの世界では様々なデータを扱うが、その 1 つに YAML（*YAML Ain't Markup Language*）というテキスト形式のデータシリアライズ言語が存在する。YAML は Perl だけでなく様々な言語で標準利用可能だが、その昔は Perl でよく扱われていたため、ここで紹介しておく。

yaml-mode は ELPA からインストールすることが可能である。

```
M-x package-install RET yaml-mode RET
```

ELPA からインストールすると、yml ファイルを開くと自動的に yaml-mode が選択され、モードラインに「YAML」と表示される。

7.1.7 Ruby

今日（2017 年 10 月現在）において、Ruby は最もプログラマに愛されている言語と言っても過言ではないだろう。プログラマがプログラムのことだけを考えて楽しく書くことを目標に設計された Ruby は、Ruby on Rails（以下 Rails と記す）というフレームワークの登場によって一躍世界中で注目を浴びた。

因みに、Ruby の生みの親である まつもとゆきひろ 氏は、Emacs から利用可能なメーラ「morq」と「cmail」を開発する程の Emacs ユーザとしても知られている。

Emacs で Ruby を編集するためのメジャーモードである ruby-mode は、Emacs 23 から本体に同梱されている。作者はまつもと氏本人であるため、安心して利用することができるだろう。

ruby-mode のインデントを調整する

rb ファイルを開くと自動的に ruby-mode が選択される。ruby-mode も各種プログラミングモードと同様にインデントの設定が可能となっている。インデントの初期設定はスペース 2 つ分で、タブ文字は使用しないようになっている。これを変更したい場合は、次のように init.el に設定を追記することで変更することが可能である。

```
;; ruby-mode のインデントを調整する
(setq ruby-indent-level 3              ; インデント幅を 3 にする（初期値は 2 ）
      ruby-deep-indent-paren-style nil ; 改行時のインデントを調整する
      ;; ruby-mode 実行時に indent-tabs-mode を初期値に変更する
      ruby-indent-tabs-mode t)         ; タブ文字を使用する（初期値は nil）
```

Ruby 編集用の便利なマイナーモードを利用する

Ruby には Ruby のソースコードにも標準添付されている便利なマイナーモードが 2 つ用意されている。1 つは括弧などを自動挿入してくれる ruby-electric、もう 1 つが対話的に Ruby を実行可能にするインタラクティブシェルである irb を Emacs 上から利用可能とする inf-ruby である。

2 つとも ELPA からインストールすることが可能である。

```
M-x package-install RET ruby-electric RET
M-x package-install RET inf-ruby RET
```

インストール後、次の設定を init.el に追記する。

```
;; ruby-mode-hook に ruby-electric-mode を追加する
(add-hook 'ruby-mode-hook #'ruby-electric-mode)
```

これで次回 ruby-mode を起動した際に ruby-electric-mode が利用可能となる。inf-ruby は自動的に ruby-mode-hook に追加されるため、特に設定しなくても利用可能となっている。ruby-electric は特に使い方を覚える必要なく利用することができるため、以降は inf-ruby の使用方法を解説する。

Emacs から irb を利用する── inf-ruby

inf-ruby を利用するためには `C-c C-s (inf-ruby)` を実行する。実行するとウィンドウが分割され irb が起動した *ruby* バッファが開く。勿論、このまま irb を利用することも可能だが、編集中のソースコードのウィンドウで次の表 7.3 のコマンドが利用可能となる。

表 7.3: inf-ruby のキーバインド

キー	コマンド名	説明
C-c C-s	inf-ruby	irb を起動する。
C-c C-b	ruby-send-block	ブロックを irb へ送る。
C-c C-x	ruby-send-definition	メソッド定義を irb へ送る。
C-c C-r	ruby-send-region	リージョンを irb へ送る。
C-c C-z	ruby-switch-to-inf	*ruby* バッファへ移動する。
C-c C-l	ruby-load-file	ファイルを読み込んで irb へ送る。

例えば、実行したい部分をリージョンで囲って `C-c C-r` をタイプすると、実行結果が *ruby* バッファに表示される。

7.1.8 Python

Python はシンプルな文法を持ち、標準ライブラリが非常に充実したスクリプト言語である。Mac や Linux などで標準ツールの作成言語として活用されていたり、最近（2017 年 10 月現在）では機械学習方面でも人気を博していて、Google や Dropbox などの Web サービスでも積極的に活用されている（Python の開発者である Guido van Rossum 氏は 2013 年まで Google に在籍し、2017 年からは Dropbox に在籍している）。Python のインデントはブロック構造を表現するための文法として機能する。そのため、誰が書いてもほぼ統一された見た目となり、その読み易さや効率の良さからプログラミング言語教育向きであるとも言われている。

python-mode

py ファイルを開くと自動的に python-mode が選択される。

構文をチェックする

python-mode では C-c C-v (python-check) を実行することで構文チェックを行うことができる。標準では pyflacks を利用するように設定されているが、次の設定によって変更が可能である。

```
(setq pyhon-check-command "flake8")
```

Python には様々な構文チェックツールが用意されているので、必要に応じてインストールすればよい。

7.1.9 C/C++

プログラミング言語の中で最も有名なのはやはり C であろう。C はチューリングマシン（イギリスの数学者 Alan Mathison Turing 氏の論文「計算可能数について――決定問題への応用」で発表された仮想マシン）を具体化したノイマン型アーキテクチャに特化した言語であり、コンピュータが今の形を続ける限り滅びることはない。但し、C は Ruby などの新しいスクリプト言語に比べて読み書きが難しく、学習コストや生産性、メンテナンス性などを考えると簡単な処理に向いているとは言えない。だが、多くのプログラミング言語や OS が C で書かれている。そのため、Emacs には C を書くためのノウハウが凝縮されており、C でコーディングしたい者は Emacs の利用を推奨する。

cc-mode

Emacs で C を編集する際は、本体に同梱されている cc-mode というメジャーモードを利用するのが一般的である。しかし、実際には cc-mode というメジャーモードは存在しない。cc-mode は「major-mode for editing C and similar language」という位置付けで、C を中心とした類似言語である C、C++、Objective-C、Java、AWK などをサポートしている。c-mode、c++-mode、objc-mode、java-mode、awk-mode などが cc-mode 上に定義されており、特に設定することなく、それぞれの拡張子のファイルを開くと自動的にメジャーモードが選択されるようになっている。

尚、これまでも何度か解説してきたインデントについては、多くのプログラミング言語用のメジャーモードで c-mode のインデントスタイルを利用している。

cc-mode 付属のマイナーモード

cc-mode には幾つかのマイナーモードが付属している。例えば、c-mode を実行するとモードラインに「C/l」と表示され、c++-mode を実行すると「C++/l」と表示される。この「C」は c-mode の「C」であり「l」は electric mode というマイナーモードがオンになっているという印である。cc-mode には編集を手助けしてくれる機能が標準でマイナーモードとして搭載されており、利用者の好みで容易にオン／オフを切り替えることができるようになっている（表 7.4）。

表 7.4: cc-mode 付属のマイナーモード

マイナーモード名	説明	切り替えコマンド	モードラインの表示
syntactic-indentation-mode	標準でオン。オフにすると C-j でインデントしない。	c-toggle-syntactic-indentation	なし
electric-mode	特定の文字（; や : など）を入力すると自動的にインデントする。	c-toggle-electric-state (C-c C-l)	|
auto-newline-mode	特定の文字（{ や ; など）を入力すると自動的に改行を挿入する。	c-toggle-auto-newline (C-c C-a)	a
hungry-delete-mode	<del> や <backspace> で空白文字を削除する際、次の文字が現れるまでの空白文字を全て削除する（改行も含む）。	c-toggle-hungry-state	h

subword-mode	Emacs 上のワードの単位を CamelCase（キャメルケース）に対応させる。	subword-mode (C-c C-w)	w

※ CamelCase（または camelCase）とは、単語の境界を _ や - などで区切らず大文字にすることで表現する記法である。先頭の文字を大文字から始めるアッパーキャメルケース（UCC）と小文字から始めるローワーキャメルケース（LCC）が存在する。プログラミングの世界では LCC が一般的に利用されている。

表 7.4 最後の subword-mode だけは cc-mode 以外でも利用可能である（但し、キーバインドによオン／オフは cc-mode のみ標準で利用可能）。標準でオンにしたい／オフにしたい場合は、次に紹介するフックから設定が可能である。

c-mode-common-hook とそれぞれのフック

cc-mode には cc-mode 全体で共通利用する c-mode-common-hook と、c-mode-hook や php-mode-hook などのようなそれぞれのメジャーモード専用のフックの 2 種類が用意されている。実行される順番は、先ず c-mode-common-hook が実行され、その後にそれぞれのメジャーモード専用のフックが実行されるようになっている。

7.2 Flycheck による文法チェック

括弧の対応関係などは、メジャーモード標準のインデントやシンタックスハイライトの機能などによって視覚的に直ちに気付くことが可能だが、例えば文末のセミコロンが抜けていてもプログラムを実行する際まで気付くことは難しい。このような簡単な文法ミスについても Emacs 上の編集段階で気付くことが可能である。この文法チェックの機能を提供してくれるのが Flycheck である。

Flycheck の仕組みとしては、現在編集中のバッファのコピーを作成して外部の文法解析ツールを定期的に実行さる。各種文法解析ツールはフォーマットこそ多少異なるが、検知した文法ミスをエラー内容と行番号で出力するという形式はほぼ同じである。その結果を解析して間違いのレベルによって、警告やエラーなどを適切に色分けして Emacs 上に反映するという仕組みとなっている。そのため、特定の言語に限定することなく利用可能となっている。

7.2.1 Flymake との違い

同様の仕組みとして Emacs に標準で Flymake という機能が備わっている。しかし、Flymake はその名の通り本来は文法チェックではなく make コマンドを実行するための仕組みとして用意されたものであるため、文法チェック機能に限れば Flycheck の方が多くの点で優れている。

Flymake と Flycheck の違いについては「Flycheck versus Flymake」[*6]にまとめられているが、最大の違いは Flycheck では殆どの言語で設定を必要としない点であろう。

7.2.2 Flycheck を利用可能な言語

Supported Language[*7] では次の通り 50 以上の言語が挙げられている。

> Ada, AsciiDoc, C/C++, CFEngine, Chef, Coffeescript, Coq, CSS, D, Dockerfile, Elixir, Emacs Lisp, Erlang, ERuby, Fortran, Go, Groovy, Haml, Handlebars, Haskell, HTML, JavaScript, JSON, Less, Lua, Markdown, Perl, PHP, Processing, Protobuf, Pug, Puppet, Python, R, Racket, RPM Spec, reStructuredText, Ruby, Rust, Sass/SCSS, Scala, Scheme, Shell Scripting Language, Slim, SQL, systemd Unit Configuration, TeX/LaTeX, Texinfo, TypeScript, Verilog, XML, YAML

また、Python で紹介したように言語によっては複数の文法チェッカが存在する場合があるが、Flycheck は複数の文法チェッカを利用することも可能となっている。

7.2.3 Flycheck の利用

それでは Flycheck を実際に導入してみる。

[*6] http://www.flycheck.org/en/latest/user/flycheck-versus-flymake.html
[*7] http://www.flycheck.org/en/latest/languages.html

インストールする

Flycheck は ELPA からインストールすることが可能である。

```
M-x package-install RET flycheck RET
```

文法チェックを実行する

Flycheck による文法チェックは基本的には自動的に実行される。利用可能とするには init.el に次の設定を追記する。

```
(add-hook 'after-init-hook \#'global-flycheck-mode)
```

設定を追加すると Flycheck がサポートしていて、かつ文法チェッカが存在している言語のファイルを開くと自動的に文法チェックが開始され、エラー箇所に下線が引かれるようになる。

機能を追加する

Flycheck は拡張機能をインストールすることで対応する言語が追加可能となっている。追加したい言語がある場合は package.el のパッケージ一覧から探してインストールすればよい。

言語追加以外にも、エラー表示をよりわかりやすくしてくれるパッケージも存在する。例えば、flycheck-pos-tip はエラーをツールチップとしてカーソル付近に表示してくれるパッケージである。

これも ELPA からインストールすることが可能である。

```
M-x package-install RET flycheck-popup-tip RET
```

インストールした後、次の設定を init.el に追記することで、エラー表示がツールチップ表示されるようになる。

```
;; Flycheck の追加機能を利用するための設定
(with-eval-after-load 'flycheck
  (flycheck-popup-tip-mode))
```

7.3 コードの実行

Emacs 上でプログラムを書いている際、現在のコードを実行してみたいと思うことはないだろうか？ そんな場合に便利なのが syohex 氏が開発した quickrun である。

7.3.1 quickrun によるコード実行

quickrun は Emacs のバッファを様々なプログラムで実行可能な拡張機能である。殆どの代表的な言語に対応しており、バッファやリージョンのコードを実行して実行結果を *quickrun* バッファに表示する。

インストールする

quickrun は ELPA からインストールすることが可能である。

```
M-x package-install RET quickrun RET
```

インストールが完了すると、直ちに quickrun が提供するコマンドが利用可能となる。

コードを実行する

quickrun が提供する機能を表 7.6 に示す。実際によく利用するコマンドは quickrun と quickrun-region になるだろう。

表 7.6: quickrun が提供するコマンド一覧

コマンド名	説明
quickrun	現在のバッファを実行する。
quickrun-region	選択中のリージョンのみ実行する。
qucikrun-shell	シェルから対話的に実行する。

quickrun-compile-only	コンパイルのみ実行する。
quickrun-replace-region	リージョンを実行結果で置換する。
quickrun-with-arg	ミニバッファから引数をとって実行する。

quickrun の使い方は非常に簡単である。プログラムを書いて実行したくなった際に M-x quickrun を実行するだけである。quickrun コマンドを実行すると、現在のメジャーモードからコンパイラやインタプリタを自動的に判別し、カレントバッファに記述されているコードの実行結果が *quickrun* バッファに表示される。

quickrun を利用すると、これまでターミナルからプログラムを実行していた操作を Emacs 内で行えるようになり、より開発者に集中することが可能となる。

7.4 タグによるコードリーディング

プログラムを読む際、ある関数、ある変数がどのように定義されているかを調べることが多々ある。1 つのファイルに書かれた短いスクリプトであれば検索ですぐに発見することができるかもしれないが、複数のファイルに渡る規模の大きなプログラムの場合は定義元を見つけるのもひと苦労である。そんな場合に役立つのがタグによるジャンプである。

タグとはソースコードにあるオブジェクト（関数やクラスや変数）を参照しやすいようにインデックスを付したファイルのことで、Emacs では標準で Etags と呼ばれるタグを生成するプログラムと etags.el というタグを利用したジャンプ機能が備わっている。

タグファイルを作成することで、Emacs からオブジェクト一覧を表示し、その定義元に一発でジャンプできるようになる。これによって初めて読むソースコードも素早く理解することが可能となる。

7.4.1 Etags 以外のタグ作成プログラム

Emacs には標準で備わっている Etags 以外にも有名なタグ作成プログラムが 2 つ存在する。1 つは GNU GLOBAL[*8]（以下 gtags と記す）、もう 1 つは Exuberant Ctags[*9]（以下 ctags と記す）である。

gtags は対応言語こそ少ないものの構文解析に優れ、解析結果の HTML 出力が可能であるなどの高性能さが特徴である。ctags は豊富な言語対応と etags 互換タグファイルの作成が可能（すなわち etags.el から利用可能）となっている。

どのタグシステムもそれぞれ利点があり、言語や環境または好みに応じて使い分けることが望ましかったのだが、gtags は 6.2.7 から ctags と連携可能となり、6.3.2 からは Python 製のシンタックスハイライター Pygments とも連携可能となった。そのため、gtags を利用すると現存する殆ど全ての言語でタグを利用することが可能となったので、特にこだわりがない場合は gtags を利用するとよいだろう。

7.4.2 gtags と Emacs の連携

gtags は Homebrew が利用可能な環境であれば簡単にインストールすることが可能である。ctags と Pygments と連携する場合は、次のコマンドでインストールする。

```
$ brew install global --with-exuberant-ctags --with-pygments
```

連携機能を利用したい場合は、更に利用しているシェルに環境変数を設定する。

```
# bash や Zsh での設定
export GTASCONF=/usr/local/share/gtags/gtags.conf
export GTAGSLABEL=pygments
```

gtags をインストールすると gtags コマンドが利用可能となる。タグを作成したいディレクトリで gtags -v コマンドを実行することで、ソースコードの解析とタグの作成の両方が行われる。

[*8] http://www.gnu.org/software/global
[*9] http://ctags.sourceforge.net

gtags.el をインストールする

次に Emacs から gtags を利用するために gtags.el をインストールする。gtags.el は gtags のソースコードを展開したディレクトリか /usr/local/share/gtags/ 内に配置されているので package.el を用いてインストールする。

```
M-x package-install-file RET /usr/local/share/gtags/gtags.el RET
```

インストール完了後、次の設定を init.el に追記することで Emacs から GTAGS ファイルを利用したタグジャンプ機能を利用することが可能となる。

```
;; gtags-mode のキーバインドを有効化する。
(setq gtags-suggested-key-mapping t)
;; ファイル保存時に自動的にタグをアップデートする。
(setq gtags-auto-update t)
```

標準ではキーバインドは設定されていないが、読み込み時に gtags-suggested-key-mapping 変数に nil 以外の値をセットすると gtags-mode のキーバインドが有効化される。但し、gtags-mode のキーバインドは標準のキーバインドを一部上書きしてしまうため、一度試した上で好みに応じて設定を追加するとよいだろう。

gtags-mode の使い方

gtags.el の機能は gtags-mode というマイナーモードとして設計されているが、マイナーモードがオンになっていなくても読み込んだ時点で各種コマンドが利用可能となっている。基本となるコマンドは、解析されたタグファイルの中から関数やクラスなどの定義元へジャンプする `M-x gtags-find-tag` とジャンプする前の場所へ戻る `M-x gtags-pop-stack` である。基本的には、この 2 つのコマンドを使いこなすだけで十分なのだが、それ以外の代表的なコマンドも簡単に紹介しておく。

表 7.7: gtags.el のコマンド

キー	コマンド名	説明
C-c t	gtags-find-tag	関数の定義元へ移動する。
C-c r	gtags-find-rtag	関数の参照元へ移動する。
C-c s	gtags-find-symbol	変数の定義元と参照元へ移動する。
C-c P	gtags-find-file	タグで解析されているファイルへ移動する。
C-t	gtags-pop-stack	移動前の場所へ戻る。

Helm と gtags の連携

helm-gtags を利用することで Helm インタフェースを用いて gtags を利用することが可能となる。helm-gtags は ELPA からインストールすることが可能である。

```
M-x package-install RET helm-gtags RET
```

helm-gtags は gtags.el の機能を基本的に踏襲している。そのため、helm-gtags を利用するのであれば gtags.el は必要ない。gtags.el のコマンドとの対応は表 7.8 にまとめておく。

表 7.8: helm-gtags コマンドと gtags コマンドの対応

コマンド名	対応コマンド
helm-gtags-find-tag	gtags-find-tag
helm-gtags-find-rtag	gtags-find-frag
helm-gtags-find-symbol	gtags-find-symbol
helm-gtags-find-files	gtags-find-files
helm-gtags-pop-stack	gtags-pop-stack

設定は次のようになる。

```
(custom-set-variables
 '(helm-gtags-suggested-key-mapping t)
 '(helm-gtags-auto-update t))
```

helm-gtags を利用するとタグの検索が更に便利になるため、Helm を利用している者には利用を推奨する。

7.5 プロジェクトベースの編集：projectile

例えば、Rails のようなフレームワークを用いた開発プロジェクトを複数抱えている場合、同名のファイルが幾つも存在するため、Helm などを利用してファイルを検索すると複数のファイルがヒットする。しかし、Emacs がプロジェクト毎にスコープを持っていれば、現在開発を行っているプロジェクトのみを対象にしてファイルの検索などを行うことが可能となる。

7.5.1 projectile によるプロジェクト管理

Emacs では projectile という拡張機能によって、主にリポジトリを対象としたプロジェクトスコープを作成することが可能である。Emacs 内でプロジェクトを切り替えることが可能で、同名のファイルが存在していても現在編集中のプロジェクトに対してのみにファイル検索対象を絞り込むことが可能となる。

インストールする

projectile は ELPA からインストールすることが可能である。

```
M-x package-install RET projectile RET
```

次の設定を init.el に追記すると Emacs でファイルを開いた際、自動的に projectile によるプロジェクト管理が開始される。

```
;; projectile を利用するための設定
(when (require 'projectile nil t)
  ;; 自動的にプロジェクト管理を開始する
  (projectile-mode)
  ;; プロジェクト管理から除外するディレクトリを指定する
  (add-to-list
   'projectile-globally-ignored-directories
   "node_modules")
  ;; プロジェクト情報をキャッシュする
  (setq projectile-enable-caching t))
```

projectile を利用する

projectile はプロジェクトというスコープを用いて、これまで紹介してきた Emacs の標準的な機能である「検索や置換」「ファイルの切り替え」「コマンドの実行」など、様々な操作が行うことが可能となっている。代表的な機能を表 7.9 に示す。

表 7.9: projectile による代表的なコマンド一覧

キー	Helm	コマンド名	説明
C-c p f	○	projectile-find-file	編集中のプロジェクトのファイル一覧を表示する。
C-c p F	○	projectile-find-file-in-known-projects	開いたことがあるプロジェクトのファイル一覧を表示する。
C-c p d	○	projectile-find-dir	編集中のプロジェクトのディレクトリ一覧を表示する。
C-c p s g	○	projectile-grep	プロジェクト内で grep を実行する。
C-c p v	−	projectile-vc	vc-dir コマンドをプロジェクトルートで実行する。

C-c p b	○	projectile-switch-to-buffer	プロジェクトで現在開いているファイル一覧を表示する。
C-c p r	−	projectile-replace	プロジェクトに M-%(query-replace) を実行する。
C-c p R	−	projectile-regenerate-tags	プロジェクトのタグファイルを再生成する。
C-c p j	−	projectile-find-tag	カーソル位置の関数の定義元へ移動する。
C-c p k	−	projectile-kill-buffers	プロジェクトで開いているファイルを全て閉じる。
C-c p D	−	projectile-dired	プロジェクトルートで Dired を開く。
C-c p e	○	projectile-recentf	プロジェクトで最近開いたファイル一覧を表示する。
C-c p !	−	projectile-run-shell-command-in-root	プロジェクトルートでシェルコマンドを実行する。
C-c p c	−	projectile-compile-project	プロジェクトの標準ビルドコマンドを実行する。
C-c p P	−	projectile-test-project	プロジェクトの標準テストコマンドを実行する。
C-c p t	−	projectile-toggle-between-implementation-and-test	対応するソースファイルとテストファイルを切り替える。
C-c p p	○	projectile-switch-project	開いたことのあるプロジェクトを切り替える。
C-c p ESC	−	projectile-project-buffers-other-buffer	プロジェクトで最も開いたファイルに切り替える。

尚、projectile のキーバインドは C-c p というプレフィックスキーが設定されている。このプレフィックスキーは次の設定から変更可能になっているので、好みに応じて変更可能である。

```
;; projectile のプレフィックスキーを s-p に変更する。
(define-key projectile-mode-map (kbd "s-p") 'projectile-mode-map)
```

Helm を用いて利用する

helm-projectile を利用することで、Helm インタフェースを用いて projectile の一部のコマンドを利用することができる。helm-projectile は ELPA からインストールすることが可能である。

```
M-x package-install RET helm-projectile RET
```

次の設定を init.el に追記すると projectile-find-file など一部のコマンドのキーバインドが helm-projectile-find-file などの Helm インタフェースを用いたものに置き換えられる。

```
;; Fuzzy マッチを無効化する。
(setq helm-project-fuzzy-match nil)
(when (require 'helm-projectile nil t)
  (setq projectile-completion-system 'helm))
```

helm-projectile は標準で、完全一致しなくても曖昧検索を行う Fuzzy マッチを利用する。この機能が不要な場合は、helm-projectile を読み込む前に helm-projectile-fuzzy-match に nil を設定して無効化すればよい。

Rails サポートを利用して編集するファイルを切り替える

projectile-rails は Rails サポートを強化する。これによって複数の Rails プロジェクトの開発を行っている場合でも、他のプロジェクトのファイルと区別した上で、モデルやコントローラなど特定の種類のファイルに絞って切り替えること

が可能となる。projectile-rails は ELPA からインストールすることが可能である。

```
M-x package-install RET projectile-rails RET
```

利用するためには、次の設定を init.el に追記する。

```
;; projectile-rails のプレフィックスキーを s-r に変更する
(setq projectile-rails-keymap-prefix (kbd "s-r"))
(when (require 'projectile-rails nil t)
  (projectile-rails-global-mode))
```

代表的なコマンドを表 7.10 に記す。

表 7.10: projectile-rails による代表的なコマンド一覧

キー	コマンド名	説明
C-c r m	projectile-rails-find-model	モデル一覧を表示する。
C-c r M	projectile-rails-find-current-model	対応するモデルに移動する。
C-c r c	projectile-rails-find-controller	コントローラ一覧を表示する。
C-c r C	projectile-rails-find-current-controller	対応するコントローラに移動する。
C-c r v	projectile-rails-find-view	ビュー一覧を開く。
C-c r V	projectile-rails-find-current-view	対応するビューに移動する。
C-c r p	projectile-rails-find-spec	スペック一覧を表示する。
C-c r P	projectile-rails-find-current-spec	対応するスペックに移動する。
C-c r g r	projectile-rails-goto-routes	config/routes.rb ファイルを開く。
C-c r g s	projectile-rails-goto-seeds	db/seeds.rb ファイルを開く。

helm-projectile を導入していて init.el に設定を追記している場合は、こちらも Helm インタフェースを利用する。また、プレフィックスキーも projectile 同様に変更可能となっている。

7.6 特殊な文字の入力補助

この世には様々な文字が存在する。身近なものであればアルファベット／記号／ひらがな／カタカナ／漢字などがあるが、携帯電話（スマートフォン）の普及によって一般化した絵文字は Emacs ではどのように入力すればよいのだろうか？ 本節では、このような特殊文字の入力を助けてくれる機能を紹介する。

7.6.1 絵文字の入力補助：ac-emoji

代表的な文字コードである Unicode のバージョン 6 から導入された絵文字は、今や日本だけでなく世界中の Web サービスで広く利用されている。その中でも GitHub で採用されている `:smile:` のように記述する絵文字のマークアップは Slack や Qiita などでも採用されているため、多くのエンジニアに馴染みがある。そんな絵文字マークアップの入力を auto-complete を利用して手助けしてくれるのが ac-emoji である。

インストールする

ac-emoji は ELPA からインストールすることが可能である。

```
M-x package-install RET ac-emoji RET
```

ac-emoji は auto-complete が利用可能な状態で `M-x ac-emoji-setup` を実行することで利用可能となる。そのため、text-mode や markdown-mode において ac-emoji を利用したい場合は、次のような設定を init.el に追記する。

```
(when (require 'ac-emoji nil t)
  ;; text-mode と markdown-mode で auto-complete を有効化する
  (add-to-list 'ac-modes 'text-mode)
```

```
(add-to-list 'ac-modes 'markdown-mode)
;; text-mode と markdown-mode で ac-emoji を有効化する
(add-hook 'text-mode-hook 'ac-emoji-setup)
(add-hook 'markdown-mode-hook 'ac-emoji-setup))
```

絵文字を入力する

入力は通常の auto-complete によって補完されるため :（コロン）から絵文字コードの一部を入力すると、補完候補が表示される。

7.7 差分とマージ

コマンドラインツールに慣れ親しんだ UNIX 系 OS ユーザであれば diff/patch コマンドは必須コマンドであろう。diff コマンドを利用して 2 つのファイルを比較したり、パッチを作成したり、patch コマンドで差分を適用する際に差分を確認しながら編集したい場合は Emacs の diff 機能を利用することを推奨する。

Emacs には diff コマンドを Emacs 上で実行するだけでなく、異なっている部分を自由自在にジャンプする機能や、2 つのファイルを比較しながらマージする機能など様々な機能が備わっている。

Emacs 標準の diff 機能はシンプルな diff と高機能な Ediff の 2 つが存在する。

7.7.1 diff による差分表示：M-x diff

`M-x diff` は diff コマンドを Emacs バッファに表示させるためのコマンドである。

```
M-x diff ~/tmp/example1.txt RET ~/tmp/sample2.txt RET
```

という形で 2 つのファイルを指定すると、コマンドラインで、

```
$ diff -u ~/tmp/example2.txt ~/tmp/sample1.txt
```

と実行した場合と同じ結果が *Diff* バッファに出力される。修正後ファイルと修正前ファイルを指定すると順番がコマンドラインと逆で、diff コマンドでは修正後ファイルの方が先に表示される点に注意すること。

Diff を `C-x C-w (write-file)` コマンドで名前を付けて保存するとパッチとして利用することも可能だが、パッチを作成する場合は次に紹介する `M-x ediff` を利用する方がよいだろう。

7.7.2 Ediff による差分表示：M-x ediff

Ediff では単に diff コマンドの結果を出力するのではなく、ウィンドウを分割して差分を視覚的に比較することができる。また、Ediff は 3 つファイルを同時に比較することも可能である。

ファイルを比較する

Ediff の使い方は diff と似ており、2 つのファイルを比較する場合、

```
M-x ediff RET ~/tmp/example2.txt RET ~/tmp/sample1.txt RET
```

のように用いる。ファイルの指定順が `M-x diff` とは逆順で diff コマンドと同じである。この順番は、ファイルを比較するだけの場合は関係ないのだが、パッチを出力する際に意味を持ってくる。Ediff を実行するとウィンドウが分割され、ファイルを指定した順番に A、B とバッファに対して識別子が割り振られてモードラインの一番左端に表示される。これは後述するキー操作によって操作する際に利用する。

ターミナル以外の環境の Emacs の場合は標準では小さなフレームが右上に表示され、ターミナルの場合はモードラインのすぐ上に *Ediff Control Panel* というコントロールパネルフレームが表示される。Ediff では、このバッファがアクティブ（すなわちカレントバッファ）になっている状態で操作を行う。「Type ? for help」と表示されている通り ? を押すとキーバインド一覧が説明付きで表示される。

同一フレーム内にコントロールパネルを表示する

右上にコントロールパネルフレームが表示される形式が好みでない場合、次の設定を init.el に追記しておくとターミナルと同じようにモードラインの上にコントロールパネルが表示されるようになる。

```
;; ediff コントロールパネルを別フレームにしない
(setq ediff-window-setup-function 'ediff-setup-windows-plain)
```

キーバインドの一覧

Ediff の操作はコマンドが多く一見難しそうに見えるが、実際に利用するのは一部なのでそれほど難しくはない。主に n、p で差分を移動、|| でウィンドウ分割の縦と横を切り替え、wd で差分ファイルに出力（パッチを作成）くらいを覚えておけば十分活用することが可能である。主な操作を表 7.11 にまとめておく。

表 7.11: Ediff の操作一覧

キー	説明
?	ヘルプの表示／非表示を切り替える。
p, DEL	前の差分へ移動する。
n, SPC	次の差分へ移動する。
|	ウィンドウ分割の縦横を切り替える。
A, B	指定した識別子のバッファを読み取り専用にする／または解除する。読み取り専用になるとモードラインの改行コードの表示付近に「%%」が表示される。
a, b	カーソル位置の差分を他方へコピーする。
ra, rb	r に続いてタイプした識別子のバッファを元の状態に戻す。
wa, wb	w に続いてタイプした識別子のバッファを保存する。
wd	差分を保存する。
q	Ediff を終了する。

7.7.3 Ediff によるパッチの適用：M-x epatch

Ediff を用いて差分をファイルに保存することでパッチを作成することができる。先程と同じように、

```
M-x ediff RET ~/tmp/example2.txt RET ~/tmp/sample1.txt RET
```

を実行し Ediff を起動する。そして wd とタイプするとミニバッファに「Saving diff output . . . 」と表示された後「File to save in:」と問われるので、ここでは example.patch と入力してパッチを作成する。

このようにして作成したパッチの適用も、Emacs 上で行うことができる。例えば、example2.txt に適用するとする。パッチを適用するコマンドは M-x epatch である。このコマンドを実行すると、先ずミニバッファで「Is the patch already in a buffer? (y or n)」と問われる。パッチはまだ Emacs のバッファ上には存在せずファイルを指定する必要があるので n とタイプする。すると今度は「Patch is in File:」と問われるので、ここで先程作成したパッチファイル ~/tmp/example.patch と入力する。ウィンドウが分割されパッチファイルの内容が表示され「File to patch (directory, if multifile patch):」とパッチを適用したいファイルを問われるので ~/tmp/example2.txt と入力すると example2.txt にパッチが適用される。パッチが適用されると自動的にファイルの末尾に.orig が付けられたパッチアップファイルが作成され、Ediff による差分比較になる。ここで再びパッチを適用する前のファイルの内容に戻すことも可能となるが、その必要がなければ q で Ediff を終了する。

7.7.4 Ediff によるマージ：M-x ediff-merge

最後に Ediff の機能を用いて 2 つのファイルの差分から新しいファイルを作成する機能を紹介する。M-x ediff-merge コマンドを実行すると Ediff と同じく比較する 2 つのファイルを問われるが、ウィンドウが 3 分割される。3 つ目のウィンドウは *ediff-merge* バッファで、2 つのファイルのどちらか指定した方の差分を採用しマージして、最後に保

存することで新しいファイルを作成することができる。

ediff-merge の操作は基本的に Ediff と同じである。n、p で差分を移動し a、b によって採用する差分を選択する。また、選択後に r をタイプすることで元に戻すことも可能である。マージが完了したら q をタイプして ediff-merge を終了し、C-x C-w によって名前を付けて保存するか *ediff-merge* バッファは C という識別子が与えられているで ediff-merge 終了前に wc とタイプしてそのまま保存することも可能である。

7.8 Emacs からデータベースを操作する

データベース操作には、MySQL であれば phpMyAdmin といった便利なインタフェースが存在する。これらを利用すれば SQL を覚えていなくてもデータベースを操作することができる。また、最近のフレームワークも標準で用意されたモデルやメソッドを活用することによって 1 行も SQL 文を記述することなくデータベースを利用可能となっている。

しかし実際には、SQL の知識を持たずに開発を行うと不要なクエリを発行して重くて使い物にならないアプリケーションを作ってしまうことがあるため、必要最低限の SQL は学習しておくべきである。SQL を学ぶには自身で SQL を書いて実行するのが一番である。通常ターミナルから対話的に操作を行うが、SQL 文を頻繁に書かない者にとって SQL を覚えるのはなかなか苦労する。

そこで Emacs の登場である。Emacs は代表的なデータベースサーバに接続し、操作する機能を標準で備えている。Emacs はエディタなので SQL 文もターミナルよりも楽に記述することができる。また、SQL 文をファイルに保存することで一度利用した SQL 文を繰り返し利用することが可能となる。生の SQL に触れながら楽に SQL を覚えることができ、実用性にも富む Emacs からデータベース操作を覚えておくとよいだろう。

7.8.1 データベースへの接続

Emacs からデータベースへ接続するには、M-x sql-mysql のように「M-x sql-データベース名」というコマンドを用いる。次の通り、標準で多くのデータベースに対応している。

- MySQL
- PostgreSQL
- SQLite
- Oralce
- Microsoft SQL Server
- DB2

MySQL へ接続する：sql-interactive-mode

例として MySQL へ接続してみることにする。M-x sql-mysql コマンドを実行するとミニバッファで「ユーザ名」「パスワード」「データベース名」「ホスト名」を問われるので、それぞれ適切に入力する。すると、ターミナルから SQL シェルを実行した場合と同じ画面が Emacs の *SQL* バッファに表示される。

この *SQL* バッファは sql-interactive-mode という Emacs 上から SQL シェルを操作するメジャーモードである。毎回同じデータベースに接続する場合は init.el に初期設定を記述しておくことで入力を省略することができる。

```
;; SQL サーバへ接続するためのデフォルト情報
(setq sql-user     "root"          ; デフォルトユーザ名
      sql-database "database_name" ; データベース名
      sql-server   "localhost"     ; ホスト名
      sql-product  'mysql)         ; データベースの種類
```

sql-interactive-mode によりターミナルのシェルから SQL を発行する必要はなくなったが、これだけでは Emacs から SQL を利用可能となっただけに過ぎない。

次に紹介する sql-mode と連携することで、格段に便利で学習効率の良い SQL 操作を実現することが可能となる。

7.8.2 sql-mode との連携

次に C-x C-f example.sql としてファイルを作成する。拡張子が sql のファイルを開くと自動的に sql-mode というメジャーモードが選択されるので、再び sql-interactive-mode を利用してデータベースに接続する。そして example.sql ファイルに show-tables; と記述して C-c C-c を実行すると、example.sql に記述した SQL 文が *SQL* バッファ上で実行される。

sql-mode では C-c C-c に sql-send-paragraph というコマンドが割り当てられている。sql-send-paragraph はその名の通り、段落文を Emacs 上の SQL シェルへと送信する。すなわち、sql-mode と sql-interactive-mode を組み合わせることで、テキストファイルから直接 SQL サーバへクエリを送信し処理することが可能となる。

SQL シェルは手軽な反面、複雑な SQL 文を実行するのは少々面倒である。しかし、Emacs のテキスト編集機能がフル活用できていれば、長い SQL 文でも楽に編集することができるはずである。そして、SQL 文を記述しているのはシェル上ではなくテキストファイルである。そのため、ファイルに保存しておくことで一度記述した SQL 文を複製することも可能となる。

データベース毎に SQL ファイルを作成し、よく利用する SQL 文を保存しておくことでデータベースのメンテナンスや集計など全てサーバ上の Emacs から行えるようになるのが理想である。

7.9 バージョン管理

開発の現場において、今や必須なのがバージョン管理システムである。作業履歴を管理し、過去のコードを遡ることを可能にしてくれるバージョン管理システムは、コードを書く全ての者が使うべきツールである。勿論、全ての機能を使いこなすためにはそれ相応の学習コストがかかるが、それ以上の恩恵を受けることができる。バージョン管理システム本体はコマンドラインツールだが、敷居を下げるために様々なフロントエンドツールが存在する。例えば、ファイラに統合されたフロントエンドツールや専用のアプリケーションも多数存在する。

Emacs は標準で殆どのバージョン管理システムに対応している。また、拡張機能を導入することで更に簡単にバージョン管理システムの機能を利用することが可能となる。まずは Emacs 標準で搭載されているバージョン管理システムの機能を解説し、その後で代表的なバージョン管理システム向けの拡張機能を紹介することにする。

7.9.1 Emacs 標準のバージョン管理機能

Emacs が標準で搭載するバージョン管理システム用のインタフェースは VC と呼ばれる。この VC は vc- から始まるコマンド群で構成されている。コードがどのバージョン管理システムで管理されているのかを自動判別し、違いを意識する必要がないように設計されている。尚、Emacs から Git などを利用するにはシステムにインストールした上で第 5 章「パスの設定」で解説した exec-path 変数に各種バージョン管理ツールのコマンドパスを設定しておく必要がある。

バージョン管理システムで管理されているファイルを開くと、モードラインに各バージョン管理システム固有のインジケータが表示され、VC コマンドが利用可能となる。VC の代表的なコマンドは表 7.12 の通りである。

表 7.12: 代表的な VC コマンド

キー	コマンド名	説明
C-x v v	vc-next-action	次に行うべき操作を実行する。
C-x v i	vc-register	ファイルを追加する。
C-x v m	vc-merge	ファイルをマージする。
C-x v =	vc-diff	最新のチェックインとの差分を表示する。
C-x v l	vc-print-log	ログを表示する。
C-x v d	vc-directory	VC 用ディレクトリエディタ。

C-x v g	vc-annotate	チェンジログを表示する。
C-x v u	vc-revert	編集中のファイルを差し戻す。
C-x v ~	vc-revision-other-window	編集中のファイルの指定バージョンを表示する。

覚えておきたいコマンドは、次に行うべき操作を自動的に判断して実行してくれる C-x v v (vc-next-action) である。ファイルの追加、コミットというバージョン管理システムを利用する際のフローを自動化してくれる。

標準の VC も十分に便利なのだが、各バージョン管理システム専用に開発されたフロントエンド拡張機能を導入することで Emacs からバージョン管理システムが更に利用し易くなる。本稿では、広く普及している Subversion と、最近最も人気の高いバージョン管理システムである Git のフロントエンド拡張機能を紹介する。

7.9.2 Subversion フロントエンド：psvn

Git などの分散型バージョン管理システムが誕生する以前は、バージョン管理システムのスタンダードと言えば Subversion であった。Subversion は集中型バージョン管理システムであり、開発者はソースコードを中央リポジトリに対してコミット／チェックアウトすることによって開発を進めて行く。

最近ではあまり見かけなくなってきたが、古くから存在するプロジェクトで Subversion を採用している場合もある。また、Subversion は十分に枯れたソフトウェアであるため、Subversion を便利に扱う安定したソフトウェアが多く存在する。その中でも Emacs から Subversion を便利に利用するための拡張機能として人気の高い psvn を紹介する。

インストールする

psvn は ELPA からインストールすることが可能である。

```
M-x package-install RET psvn RET
```

インストールが完了すると、直ちに psvn が提供するコマンドが利用可能となる。

操作する

Subversion で管理されているファイルを編集し M-x svn-status を実行すると、ミニバッファでステータスを確認したいディレクトリを問われる。デフォルトで編集中のファイルのディレクトリが設定されているので、そのまま RET する。

すると、指定したディレクトリ以下に存在するファイル一覧が *svn-status* バッファに表示される。psvn の操作は Dired ライクになっており、n、p でファイルを移動、a でファイルを追加、m でマーク、u でアンマーク、そして c でコミットメッセージ入力のバッファとなり、C-c C-c でコミットを完了する。

U で svn up を実行することも可能であり、およそ Subversion で必要な機能は全てそろっている。詳しい操作については表 7.13 に記す。

表 7.13: psvn のコマンド一覧

キー	コマンド名	説明
g	svn-status-update	状態を更新する。
U	svn-status-update-cmd	ファイルを更新する。
a	svn-status-add-file	ファイルを追加する。
m	svn-status-set-user-mark	ファイルをマークする。
u	svn-status-unset-user-mark	ファイルをアンマークする。
c	svn-status-commit	マークしたファイルをコミットする。
r	svn-status-revert	マークしたファイルを取り消す。

=	svn-status-show-svn-diff	差分を見る。
E	svn-status-ediff-with-revision	ediff で差分を見る。
l	svn-status-show-svn-log	ログを見る。
?	svn-status-toggle-hide-unknown	管理外のファイルを表示をトグルする。

7.9.3 Git フロントエンド：Magit

Git は CVS、Subversion などの集中型とは異なり、中央サーバ不要の分散型バージョン管理システムである。Linux カーネルの開発や GitHub などの Web サービスに牽引され、現在最も普及しているバージョン管理システムと言える。

Git は使い方が難しいバージョン管理システムと言われることがあるが、それは恐らく Subversion などから移行した者が設計哲学が全く異なるために苦戦を強いられたり、Git があらゆる自体に対応可能であるように柔軟に設計されているため、できることの幅が広過ぎるためだと思われる。基本概念さえしっかり理解してしまえば、とてもパワフルで使いやすく信頼性の高い大変優れたツールである。Git を Emacs から快適に利用可能にする拡張機能の 1 つが Magit である。

インストールする

Magit は ELPA からインストールすることが可能である。

```
M-x package-install RET magit RET
```

インストールが完了したら、直ちに Magit が提供するコマンドが利用可能となる。

操作する

Magit からは Git が提供する殆ど全ての機能を利用可能だが、全てを覚える必要はない。まずは表 7.14 の基本コマンドを覚えればよい。

表 7.14: Magit の基本コマンド一覧

コマンド名	説明
magit-status	git-status 相当の情報を表示する。
magit-diff	git-diff 相当の情報を表示する。
magit-log	git-log 相当の情報を表示する。

これから解説する操作は主にこれら基本コマンドから操作することになるが、この基本コマンドを実行すると Magit による Git 操作専用のモード（以下、magit-mode と記す）が起動される。Magit は、この magit-mode からの対話的操作とミニバッファからコマンドを直接実行する直接的操作の 2 つに分かれている。

magit-mode の操作は（当然ながら）全てキーボードから行うことになるが、機能が多いため覚えるのが大変である。そこで表 7.14 で挙げた基本コマンドを実行中にメニューを表示する表 7.15 のコマンドを覚えておくとよい。

尚、Magit による操作中は q を押すことで操作をキャンセルすることができるので、こちらも合わせて覚えておくとよいだろう。

表 7.15: Magit ポップアップメニュー

キー	説明
c	コミットポップアップメニューを表示する。
l	ログポップアップメニューを表示する。
f	フェッチポップアップメニューを表示する。
F	プルポップアップメニューを表示する。
b	ブランチポップアップメニューを表示する。

P	プッシュポップアップメニューを表示する。
m	マージポップアップメニューを表示する。
z	スタッシュポップアップメニューを表示する。
M	リモートポップアップメニューを表示する。

コミットする

Magit でコミットするには、まずコミット対象となるファイルをステージングするため M-x magit-status を実行する。すると git status を実行した際に表示される画面に似た画面が表示され、そこから対話的にステージングを行うことができる。この画面では、p、n で上下に移動可能となっている。ステージングしたいファイルの行で s を入力すると、そのファイルがステージングされる。また、ステージングされたファイルの行で u を入力するとアンステージすることができる。全てのファイルを一括でステージングしたい場合は、S を入力すると現在アンステージとなっているファイル全てがステージングされる。

Git ではファイルではなく特定の編集箇所だけをステージング（すなわちコミット）することができる。これを部分ステージングと呼ぶが、これも Magit から行うことが可能である。部分的にステージングしたいファイルで TAB を押すとファイル名の表示から変更差分が展開表示される。この表示でも、p、n によって差分ブロックを移動することが可能であり、先程のステージングを行う際と同様に変更ブロックで s を入力するとファイルの一部分だけがステージングされる。ステージングされた箇所の表示も TAB を押すことで展開して確認することが可能である。

ステージングが完了し、コミットする場合は c を入力する。するとコミットポップアップメニューが開くので、更に c を入力する。次にコミットログを入力する画面が表示されるので、これにコミットログを記述して C-c C-c を実行する。これで Git リポジトリへのコミットが完了する。この際、C-c C-k を実行するとコミットをキャンセルすることができる。

7.9.4 差分の表示：git-gutter

Git で管理しているファイルを編集している際、変更状態を確認したくなることがある。そのような場合は基本的には git diff コマンドを用いるが、毎回実行するのは少々面倒である。git-gutter は Git と連携して変更差分を各行に表示してくれる拡張機能である。これを利用することで git diff を実行せずとも変更されているかどうかが直ちに分かるようになる。

インストールする

git-gutter は ELPA からインストールすることが可能である。

```
M-x package-install RET git-gutter RET
```

次の設定を init.el に追記することで、自動的に git-gutter が有効化される。

```
(when (require 'git-gutter nil t)
  (global-git-gutter-mode t)
  ;; linum-mode を利用している場合は次の設定も追加する
  (git-gutter:linum-setup))
```

差分を視覚的に表示する

git-gutter が有効な状態で Git 管理されているファイルを編集すると、画面左端に変更状態を示す記号が表示されるようになる。初期状態では変更された行には「=」、追加された行には「+」、削除された行には「-」が表示されるようになっている。これらの記号は次のような設定によって変更可能である。

```
(custoum-set-variables
 '(git-gutter:modified-sign "*")
 '(git-gutter:added-sign    ">")
 '(git-gutter:deleted-sign  "x"))
```

また、変更行へと移動する M-x git-gutter:next-hunk や M-x git-gutter:previous-hunk などのコマンドも用意されているため、頻繁に利用したい者は次のような設定でキーバインドに登録しておくとよいだろう。

```
(global-set-key (kbd "C-x p") 'git-gutter:previous-hunk)
(global-set-key (kbd "C-x n") 'git-gutter:next-hunk)
```

7.10　シェルの利用

開発者にとってシェルはなくてはならない大事なツールである。GUI が全盛の現在においてもその事実は変わらない。ターミナルを利用したことのない者からすれば、難しそうなシェルが使われ続ける理由がよく分からないかもしれないが、ターミナルが使われ続ける理由の 1 つに GUI よりもできることの幅が広いことが挙げられる。

アプリケーションは小さなコマンドの集合で構成されており、その小さなコマンドを組み合わせて無数の機能を提供している。その組み合わせをアプリケーションの垣根で限定せず、ほぼ全てのコマンドを組み合わせて利用可能なのがシェルである。自身の目的に合うアプリケーションが存在していなくても、シェルを用いてコマンドを組み合わせることで実現可能となることが多くある。

このような素晴らしいシェルを利用するためには通常ターミナルを利用する必要があるのだが、ターミナルと Emacs を行ったり来たりするのは面倒であるし、思考の中断にも繋がる。実は、Emacs にはシェルを利用する方法が多数用意されている。本節では、簡単なコマンドを実行したい場合から、ガッツリとターミナルとにらめっこしたい場合などのケースに合わせて Emacs のシェル機能の使い方を紹介する。

7.10.1　シェルコマンドの実行：M-!

Emacs には M-! (shell-command) という機能が用意されている。これはミニバッファからシェルコマンドを実行するためのコマンドで、カレントバッファが保存されるディレクトリ（存在しなければ Emacs を起動したディレクトリ）をカレントディレクトリとしてシェルコマンドを実行する。例えば、M-! date RET とタイプすると現在時刻がミニバッファに表示される。勿論、TAB による補完もできて簡単なコマンドを手軽に実行したい場合に最適である。

尚、実行結果の標準出力は *Shell Command Output* バッファへも出力されており、ミニバッファの表示を見逃してもバッファを切り替えて確認することができる。しかし、このバッファはコマンド実行の度に上書きされてしまうため、ログを確認する用途には向かない。

カレントファイルへ結果を出力する：C-u M-!

C-u M-! のように前置引数を付けると、出力先が *Shell Command Output* バッファからカレントバッファ（カーソル位置）に変更される。コマンドの出力結果をファイルに挿入したい場合などに非常に便利なので、覚えておくとよいだろう。

バッファの内容を標準入力として利用する：M-|

バッファ上のテキストをコマンドの標準入力にしたい場合、M-| (shell-command-on-region) を利用する。リージョン選択範囲を標準入力に使用してコマンドを実行する。また、こちらも C-u M-| のように前置引数を付けて実行するとリージョンが出力結果で置換される。一時ファイルを作成したければ Dired やファイラで操作するよりも Emacs から cp コマンドを実行した方が確実に速い。ちょっとしたコマンドを 1 回実行して結果を利用したい場合は M-! と M-| で殆ど実現可能なので、うまく活用するとよいだろう。

7.10.2　ターミナルの利用：multi-term

Emacs はエディタではなく環境、あるいは OS であると言われている。それは Emacs からテキストエディタの枠を超えた機能を利用できるからに他ならない。そして、Emacs 内蔵のターミナルエミュレータもその一例として挙げられる機能である。Emacs では単にコマンドを実行するだけではなく、本格的なターミナルとしてシェルを利用することも可能なのである。

Emacs のターミナルエミュレータは M-x shell、M-x term (M-x ansi-term)、M-x eshell の 3 つが代表的であり、全て標準で本体に同梱されている。この中でも本物のターミナルとの違いが少ない ansi-term が人気でよく名前が挙げられるが、ansi-term は Emacs 標準のキーバインドを奪ってしまうため、ansi-term をベースにチューニングを施した multi-term の導入を推奨する。

インストールする

multi-term は ELPA からインストールすることが可能である。

```
M-x package-install RET multi-term RET
```

インストールが済んだら、require と使用するシェルを init.el に記述するだけで設定が完了する。

```
;; multi-term の設定
(when (require 'multi-term nil t)
  ;; 使用するシェルを指定する
  (setq multi-term-program "/usr/local/bin/bash"))
```

ターミナルを起動する

M-x multi-term を実行すると *terminal<1>* と表示されたバッファが表示され、Emacs がターミナルに変身する。multi-term はその名の通り Emacs 上に幾つものターミナルを作成することも可能である。GNU Screen を利用している者であれば、Emacs が GNU Screen になったかのように思えるかもしれない。ターミナルを閉じたければ exit コマンドで終了することができる。

7.11 TRAMP によるサーバ接続

サーバ上のファイルを編集するには幾つかの方法が考えられるが、基本的な手段としては「ローカル環境へダウンロードして編集する」または「サーバへログインして編集する」の 2 通りであろう。前者の場合、自身の Emacs で編集することが可能であるため作業効率は上がるが、ファイルをアップロードし直すのが面倒である。後者の場合、サーバのファイルを取得する手間は省けるが、自身の環境とは異なる環境で作業することになるため作業効率が低下する。

そこで第 3 の選択肢として覚えておきたいのが、Emacs でサーバへログインして直接ファイルを編集する方法である。標準で本体に同梱されている TRAMP という拡張機能を用いることでサーバ上のファイルを直接を編集することが可能となる。方法は簡単で、例えば SSH が利用可能な環境であれば C-x C-f (find-file) から「/sshx:ユーザ名@ホスト名:」と入力していくと、そのままサーバへ接続してローカルファイルのように扱うことができる。

7.11.1 sudo、su による編集

TRAMP の機能を利用すると、Emacs から直接 su、sudo を用いてファイルを開くことができる。利用方法は ssh の場合と同様で C-x C-f (find-file) から「/su:」もしくは「/sudo:」でファイルを開くだけである。

7.11.2 バックアップファイルを作成しない

TRAMP を利用して SSH や su、sudo を用いたファイル編集を行っている際はバックアップファイルを作成したくないと考えるかもしれない。TRAMP はそのような要望に応えられるように設計されている。次の設定を追記すればよい。

```
;; TRAMP でバックアップファイルを作成しないための設定
(add-to-list 'backup-directory-alist
             (cons tramp-file-name-regexp nil))
```

7.12 ドキュメント閲覧・検索

開発者が困った際に頼りになるのは man などのドキュメントである。本節では Emacs から man などのドキュメントを閲覧・検索するための方法を解説する。

ドキュメントの多くは英語で記述されているため読むのに抵抗を感じるかもしれないが、ドキュメントは先人の叡智の結晶であり検索エンジンで探すよりも素早く問題を解決してくれる。もし、まだドキュメントを読むことに抵抗感がある者は、これを機会にドキュメントを読むことに挑戦することを推奨する。

7.12.1 Emacs 版 man ビューア（WoMan）の利用

man は非常によく利用されるコマンドだが、ターミナル上で man を読むのは意外と不便である。Emacs を利用しているのであれば Emacs から man を読む癖を付けるとよいだろう。man 間の移動も容易であるし、いちいち man を閉じる必要もなくなりコマンドのコピーなども簡単に行うことが可能である。

Emacs には man コマンドの結果を Emacs 上に読み込む `M-x man` と Emacs 独自の man ビューアを利用する WoMan（Without Man）の 2 種類が存在する。`M-x man` も十分便利なのだが、WoMan の方が文字コードの扱いやキー操作の面で優れているため WoMan を推奨する。

WoMan を利用するには `M-x woman RET コマンド名 RET` という形式でコマンドを実行する（man は `M-x man RET コマンド名 RET` である）。WoMan の場合は、環境に同一コマンドの man ファイルが複数存在する場合、どちらを利用するかを追加で問われる。Woman を利用する際に覚えておきたいキーバインドは `M-p (WoMan-previous-manpage)` と `M-n (WoMan-next-manpage)` である。この 2 つのキーバインドによって、ブラウザの戻る／進むのように移動することが可能となる。

利用可能にする

WoMan で設定しなければならない項目は特にないのだが、知っておきたい設定を紹介しておく。1 つ目はキャッシュの設定である。woman-cache-filename 変数にキャッシュファイル名を設定しておくと、man のパスと man で検索可能なコマンド名をキャッシュとして保存し、次回起動時の動作を高速化することができる。

```
;; WoMan でキャッシュを作成するための設定
(setq woman-cache-filename "~/.emacs.d/.wmncash.el")
```

このキャッシュは `C-u M-x woman RET` という形で前置引数を付加して起動することで更新することができる。

2 つ目は man のパス設定である。正しく man が検索できない際はパスの設定がうまくいっていない可能性が考えられるため、woman-manpath 変数に man のパスのリストを設定する。

```
;; man のパスを設定する
(setq woman-manpath '("/usr/share/man"
                      "/usr/local/share/man"
                      "/usr/local/share/mam/ja"))
```

7.12.2 Helm による man 検索

検索と言えばやはり Helm である。Helm には標準で `M-x helm-man-woman` というコマンドが定義されている。これは helm-source-man-pages という man を一覧表示するためのソースを利用している。これを用いて Emacs 上で様々なドキュメントを串刺し検索するコマンドを作成してみる。

```
;; 既存のソースを読み込む
(require 'helm-elisp)
(require 'helm-man)
;; 基本となるソースを定義する
(setq helm-for-document-sources
      '(helm-source-info-elisp
        helm-source-info-cl
        helm-source-info-pages
```

```
        helm-source-man-pages))
;; helm-for-document コマンドを定義する
(defun helm-for-document ()
  "Preconfigured ‘helm’ for helm-for-document."
  (interactive)
  (let ((default (thing-at-point 'symbol)))
    (helm :sources
          (nconc
           (mapcar (lambda (func)
                     (funcall func default))
                   helm-apropos-function-list)
           helm-for-document-sources)
          :buffer "*helm for document*")))
```

この新しく定義された `M-x helm-for-document` というコマンドは man や info、apropos の結果を同時に検索して一覧にしてくれる。コマンドを実行した時のカーソル位置にあるワードを拾い上げるようになっているため、検索したいワードの上で実行し RET するだけでドキュメントを閲覧することができる。

7.13 テスティングフレームワークとの連携

最後は、Emacs とテスティングフレームワークとの連携について解説する。最近では TDD（*Test Driven Development*、**テスト駆動開発**）もかなり浸透してきているが、テストコードを記述する者にとって開発中にテストを実行する回数は、コミットを行うより多いことだろう。そこで Emacs からテストを実行してその結果を得られるようになれば、開発がより効率的に行えるようになるだろう。Emacs は各言語のテスティングフレームワークと連携することが可能だが、本稿では PHP と Ruby の例を紹介する。

7.13.1 phpunit.el

phpunit.el は PHPUnit を Emacs から利用するための拡張機能である。プロジェクト全体やファイル単体だけでなく、カーソル付近のクラスのみを実行することも可能である。

インストールする

phpunit.el は ELPA からインストールすることが可能である。

```
M-x package-install RET phpunit RET
```

インストールが完了すると、直ちに phpunit.el が提供するコマンドが利用可能となる。

テストを実行する

代表的なコマンドを表 7.16 にまとめる。

表 7.16: 代表的な phpunit.el コマンド

コマンド名	説明
`phpunit-current-project`	全てのユニットテストを実行する。
`phpunit-current-test`	カレントバッファのユニットを実行する。
`phpunit-current-class`	カーソル行付近のクラスのユニットテストを実行する。

PHPUnit が利用可能な環境で `M-x phpunit-current-project` を実行すると *compilation* バッファにテスト結果が表示される。もし、これらのコマンドをキーバインドから実行したい場合は、次のような設定を init.el に追記すればよい。

```
;; web-mode にキーバインドを追加する。
(define-key php-mode-map (kbd "C-t t") 'phpunit-current-test)
```

```
(define-key php-mode-map (kbd "C-t c") 'phpunit-current-class)
(define-key php-mode-map (kbd "C-t p") 'phpunit-current-project)
```

7.13.2 rspec-mode

rspec-mode は Rails などでよく利用される RSpec を Emacs から利用するための拡張機能である。プロジェクト全体やファイル単体だけでなく、カーソル行付近の spec（RSpec ではテストコードのことを spec と呼ぶ）のみを実行することも可能である。

インストールする

rspec-mode は ELPA からインストールすることが可能である。

```
M-x package-install RET rspec-mode RET
```

インストールが完了すると、直ちに rspec-mode が提供するコマンドが利用可能となる。環境によっては、direnv などを利用して.envrc から環境変数を読み込まなければアプリケーションが起動できないようになっている場合がある。そのような場合は、direnv をインストールして設定を追記しておくことで対応する。

```
M-x package-install RET direnv RET
```

設定は次のようなものになる。

```
;; direnv の設定
(when (require 'direnv nil t)
  (setq direnv-always-show-summry t)
  (direnv-mode))
```

尚、rspec-mode にはテストの実行に spring を利用したり Docker を利用したりなど、環境に合わせた設定が用意されている。必要があれば、環境に応じて設定を追記することが必要となる。

テストを実行する

代表的なコマンドを表 7.17 にまとめておく。恐らく、開発中に主に利用するコマンドは `M-x rspec-verify` か `M-x rspec-verify-single` になるだろう。

表 7.17: rspec-mode による代表的なコマンド一覧

キー	コマンド名	説明
C-c , a	rspec-verify-all	プロジェクト全体の spec を実行する。
C-c , v	rspec-verify	カレントバッファの spec を実行する。
C-c , s	rspec-verify-single	カーソル行付近の spec を実行する。
C-c , r	rspec-rerun	最後に実行した spec を再実行する。
C-c , f	rspec-run-last-failed	最後に失敗した spec を実行する。
C-c , y	rspec-yank-last-command	最後に spec を実行したコマンドをコピーする。

spec を実行するとウィンドウが分割され、テスト結果が *rspec-completion* バッファに表示される。

付録 A

最新情報を入手するには

A.1　開発状況

Emacs は GNU プロジェクト[*1]というフリーウェア開発の総本山で開発されている。

A.1.1　フリーソフトウェアとしての Emacs

ここで言うフリーソフトウェアとは当然ながら価格が無料という意味ではない。また、オープンソースソフトウェアとも異なる。Richard M.Stallman 氏を中心とするフリーソフトウェア運動の歴史は、自由を守るための戦いであり、全てのソフトウェア開発者が多かれ少なかれ恩恵を享受している。

従って、その賛否に関わらずフリーソフトウェアの思想に一度は触れておくべきだと考える。開発者の生活において無くてはならないフリーソフトウェアが、どのような歴史的経緯で誕生したのかを知ることは決して無駄にはならないはずである。

このような話に興味を持った者は「GNU プロジェクトの理念」[*2]や「GNU 宣言」[*3]または Mikiya Okuno 氏の「なぜリチャード・ストールマンはオープンソースを支持しないか」[*4]などの記事を参照して、自分自身の頭で咀嚼してみるとよいだろう。とにかく、Emacs は Linux カーネルと並んで巨大で歴史あるフリーソフトウェアプロジェクトであり、誰でもソースコードを入手して開発に参加することができる。Emacs のメインコードは C 言語で書かれているが、Emacs のソースコードは C 言語で書かれたプログラムの中でも綺麗だと言われている。C 言語プログラミングに興味のある者は、一度コードリーディングをしてみるのもいいだろう。

A.1.2　ソースコードリポジトリ

Emacs のコードリポジトリは Savannah[*5] という開発ホスティングサイトで管理されている[*6]。40 年以上の歴史を持つ Emacs はバージョン管理システムも RCS（*Revision Control System*）、CVS（*Concurrent Versions System*）、Bazaar、そして Git と移り変わってきたが、Eric S.Raymond 氏の尽力により過去の記録をそのまま現在にまで引き継いでいる。

Git を利用してソースコードを取得する

Git が利用可能な環境であれば、プロジェクトディレクトリから `git clone git://git.sv.gnu.org/emacs.git` でソースコードを取得することが可能である。

A.2　開発版の試用

ソフトウェアの開発はリリース版（安定版）と開発版に分かれていることが多く、Emacs にもリリース版と開発版が存在する。一般的なソフトウェアはリリースまでにアルファ版、ベータ版を経て、RC（*Release Candidate*）版で問題がなければ安定版としてリリースすることが多い。Emacs の場合はこういった区切りとなるリリースは行われず、ある日

[*1] https://www.gnu.org/software/emacs
[*2] https://www.gnu.org/philosophy/philosophy.ja.html
[*3] https://www.gnu.org/gnu/manifesto.html
[*4] https://nippondanji.blogspot.jp/2011/06/blog-post_17.html
[*5] https://savannah.gnu.org
[*6] https://savannah.gnu.org/projects/emacs

を境にプレテスト版という形でテストリリースが行われ、この時点で新規機能の追加は終了してバグフィックスを繰り返して安定版がリリースされる流れとなっている。

A.2.1　開発版のビルド

開発版の Emacs とは、基本的にはソースコードをチェックアウトしてから自らビルドしたものを指す。一昔前では、自身でソフトウェアをビルドするためには様々な知識が必要であったが、最近では要領さえ理解すれば誰でも簡単にビルドすることができる。開発版の Emacs を利用すると次に搭載される新機能などをいち早く試すことができるので、興味のあるものは是非ビルドして利用してみるとよいだろう。

README を読む

ソフトウェアをビルドするための情報は、一般的にソースコードに同梱されている README ファイルに詳しく記載されている。何か問題が生じた場合は、Web で検索する前にまず README ファイルなどオフィシャルのドキュメントを読む癖を付けるべきである。

Emacs の README は各種ディレクトリとファイルの説明が記載されている。これによると、インストールに関する説明は INSTALL ファイルに記述されていることが分かる。但し、Git からソースコードを取得してインストールする場合は INSTALL ファイルだけではなく INSTALL.REPO ファイルにも目を通しておく必要があるので注意が必要である。

configure ファイルを作成する

Git リポジトリから取得したソースコードには configure ファイルが含まれていない場合がある。その場合、autogen.sh スクリプトを実行して作成する必要がある。これには autoconf 2.65 以上のバージョンが要求される（2017 年 10 月現在）。無事に configure スクリプトが作成することができたら、後は第 2 章で解説した通りの方法でビルドすればよい。

A.3　新しい情報に触れるには

もし好きな物事に対して人よりも一歩でも先に進みたければ、敏感にアンテナを張り巡らせて、重要であり、ホットであり、そして正しい情報をいち早くキャッチしなければならない。但し、闇雲に Web 上に溢れる情報を漁るばかりでは情報の波に飲み込まれてるばかりで、なかなか先に先に進むことができないだろう。

人よりも敏感にアンテナを張るためには、より質の高い情報が集まる場所を見つける必要がある。質の高い情報が集まる場所には必ずレベルの高い者たちが集まっており、そこで活動することによって最前線で活動することが可能となる。本節ではそのような場所を紹介していく。

A.3.1　メーリングリスト

Emacs 開発の議論は、主にメーリングリスト[*7]の emacs-devel で行われている。現在、Emacs の開発がどのように進んでいるかをチェックしたいと思ったら、メーリングリストを読んでみるとよい。また、メーリングリストへのポストはアーカイブ[*8]が用意されているので、メーリングリストに参加しなくても確認可能である。

A.3.2　EmacsWiki：世界最大の Emacs コミュニティ

メーリングリストに次いで Emacs 開発者のコミュニティとして規模の大きなものは EmacsWiki[*9] である。本稿でも数々の拡張機能を EmacsWiki から紹介してきた。EmacsWiki は Emacs 開発者にとってノウハウ共有の場であり、作品を公開する場ともなっている。EmacsWiki の RecentChanges[*10] の RSS リーダに登録し、更新をチェックしておくと世界の誰かが作成した拡張機能をいち早く知ることができる。更新頻度は多くても 1 日 10 ページ程度となっており、読むのに疲れることもない。

[*7] http://savannah.gnu.org/mail/?group=emacs
[*8] http://lists.gnu.org/archive/html/emacs-devel
[*9] https://www.emacswiki.org
[*10] https://www.emacswiki.org/emacs/RecentChanges

A.3.3　メーリングリスト、Wiki、その先へ

メーリングリストや EmacsWiki はオフィシャルな情報が集まる場所だが、実際にはそこに集まる情報だけでなく、世界中の開発者が日夜 Emacs に関する Tips を公開して拡張機能を開発している。それらの多くは開発者個人のブログや Web サイトで公開されることが多い。

Twitter などで開発者同士のコミュニティを楽しむのもよいが、やはりまとまった情報はブログを読むのが一番であろう。開発者たちのブログを紹介してもよいのだが、国内外を問わず非常に多く存在するため、ここでは紹介しきることはできない。そこで、簡単に見つける方法を一部紹介しておく。

GitHub には言語別の検索機能やトレンドを見つける機能が備わっており、Elisp のリポジトリを検索したり、活発なリポジトリを調べることができる。ここから自身にとって有益な拡張機能を作成している作者を見つけて、その作者をアクティビティをチェックするというのがその方法である。

また、はてなブックマークのタグ「emacs」を含む新着エントリの RSS をチェックするのもよいだろう。日本人向けの情報としては EmacsJP と Slack チャット[*11]も存在するので、興味のあるものは参加してみるとよいだろう。

[*11] http://emacs-jp.slack.com