チュートリアル

2020 年 12 月 8 日

目次

概要

このチュートリアルには 9 つのセクションがある。チュートリアルとしての内容はセクション 1 からセクション 8 までに書かれている。最後のセクション 9 は Buildtools のソースファイルに含まれる Readme.ja.md のコピーである。

1　インストール

1.1　動作条件

Buildtools には次のものが必要である。

1. Linux OS と bash
2. LaTeX システム
3. Make または Rake

1.1.1　Linux OS と bash

Buildtools は Debian と Ubuntu で動作を確認している。おそらく他の linux ディストリビューションでも動作すると思われる。Buildtools のスクリプトでは bash のコマンドが使われているので、bash は必須である。

1.1.2　LaTeX システム

LaTeX のインストールには 2 つの方法がある。

ひとつは、ディストリビューションに含まれている LaTeX のパッケージをインストールする方法である。ubuntu ディストリビューションの場合は、次のようにタイプしてインストールする。

```
$ sudo apt-get install texlive-full
```

他のディストリビューションであれば、そのディストリビューションのドキュメントを参照してインストールしてほしい。

もうひとつの方法は TexLive (https://www.tug.org/texlive) からインストールする方法である。その方法については、ウェブサイトのドキュメントを参照してほしい。

1.1.3　Make または rake

これらのアプリケーションは Buildtools に含まれるツールの実行には直接的には必要ではない。しかしながら、これらのツールは make または rake のコントロールの下で実行することが望ましい。make と rake の両方をインストールする必要はなく、どちらかひとつ、好みのものをインストールすれば十分である。

Make は古くから使われているビルド・ツールで、元々は C コンパイラ用に開発されたものである。ubuntu では、次のようにタイプして make をインストールする。

```
$ sudo apt-get install make
```

Rake は make に似たビルド・ツールであり、ruby アプリケーションのひとつである。rake の良いところは、任意の ruby コードを Rakefile（rake の動作を記述するスクリプト）に書くことができる、ということである。一般的に、Rakefile の方が Makefile よりも読みやすく、理解しやすい。ubuntu では、次のようにタイプして rake をインストールする。

```
$ sudo apt-get install rake
```

もしも、ruby の最新版をインストールしたければ、rbenv と ruby-build を使ってインストールするのが良い。下記の github レポジトリのドキュメントをインストールの参考にしてほしい。

- https://github.com/rbenv/rbenv
- https://github.com/rbenv/ruby-build

1.2 インストール

1.2.1 ダウンロード

まず、次の github リポジトリにブラウザでアクセスする。

- https://github.com/ToshioCP/LaTeX-BuildTools

Code ボタンをクリックするとポップアップ・メニューが現れる。DOWNLOAD ZIP メニューをクリックすると zip ファイルがダウンロードされるので、それを解凍する。

1.2.2 インストール

端末を起動して、カレント・ディレクトリを先程解凍したファイルのディレクトリに移動する。次のようにタイプしてスクリプトをインストールする。

```
$ bash install.sh
```

このスクリプトは実行スクリプトを\$HOME/bin にインストールする。Debian と Ubuntu では、\$HOME/bin ディレクトリがログイン時に存在すれば、それを PATH 環境変数に追加する。install.sh スクリプトは、\$HOME/bin が存在しなければ、それを作成するが、その場合はディレクトリを PATH に追加するために再ログインが必要である。この方法は、ユーザのプライベート・ディレクトリにスクリプトを配置するので、他のユーザはスクリプトにアクセスすることはできない。このインストールをユーザ・レベル・インストールまたはプライベート・インストールという。

もしも、インストール先を/user/local/bin にしたければ、root 権限が必要になる。ubuntu の場合は、

```
$ sudo bash install.sh
```

とタイプすれば、システムレベルのインストールができる。

2 lb を使ったコンパイル

2.1 lb の簡単な使い方

lb は Buildtools の主要なスクリプトである。このセクションでは、短い TeX ファイルを例に用いてその使い方を説明する。

はじめに example という名前のディレクトリを作成し、カレント・ディレクトリをそのディレクトリに移動する。

```
$ makedir example
```

```
$ cd example
```

そして、そのディレクトリに TeX ソースファイルを作成する。エディタを使い、下記の内容をコピーして、main.tex という名前で保存する。

```
\documentclass{article}
\begin{document}
Hello \LaTeX !!
\end{document}
```

それが済んだら、端末から lb とタイプすれば良い。

```
$ lb
```

すると、latexmk と pdflatex が起動され、main.tex がコンパイルされる。メッセージが表示され、コンパイルのプロセスが示される。その中に次のようなメッセージがあれば、コンパイルは正常に終了している。

```
Output written on _build/main.pdf (1 page, 19263 bytes).
```

ディレクトリをチェックしてみよう。

```
$ ls -l
total 8
drwxrwxr-x 2 user user 4096 Dec  6 11:59 _build
-rw-rw-r-- 1 user user   72 Dec  6 11:59 main.tex
 ... ...
```

新しいディレクトリ_build が作られている。そのディレクトリに含まれるファイルを調べてみよう。

```
$ cd _build
$ ls
```

そこには補助ファイルと、ターゲット・ファイル（コンパイルで最終的に生成されたファイル）である main.pdf がある。evince などの pdf ビューワで main.pdf を見てみよう。

```
$ evince main.pdf
```

Hello LaTeX!!

2.2 lb.conf の使い方

前のサブセクションでは lb は pdflatex を起動した。lb が pdflatex を選択した理由は、ドキュメントクラスが「article」だったからである。article のコンパイルには lualatex や xelatex も可能だが、長い間 pdflatex がスタンダードであった。

例えば、lualatex でコンパイルしたい、というようなとき、それは lb.conf しなければならない。こお初期

化ファイルには 6 つの項目がある。

rootfile　ルートファイルとは、メインの latex ファイルである。それは通常、\begin{document}と\end{document}を含んでいる。他のファイルはサブファイルという。

builddir　ビルド・ディレクトリは一時的なディレクトリともいわれ、すべての補助ファイルとターゲット・ファイル（通常は pdf）が置かれる。

engine　これは latex エンジン（ソースファイルをコンパイルするプログラム）を指定する。pdflatex、xelatex、lualatex、latex、platex を指定することができる。

latex_option　これは `latexmk` に与えるオプションである。lb.conf が無くても lb は '-halt-on-error' のオプションを自動的に与える。

dvipdf　これは dvi ファイルを pdf ファイルに変換するコンバータである。これは latex または platex がエンジンに指定されたときのみ必要であり、他のエンジンでは必要ない。dvipdfmx が最も良いとされている。

preview　Pdf ビューワである。lb が引数にサブファイルを与えられたときにコンパイル後にこれを起動する。

エディタに下記の内容を打ち込み `lb.conf` という名前で保存する。

```
rootfile=main
builddir=_build
engine=lualatex
latex_option=-halt-on-error
dvipdf=
preview=evince
```

保存したら、lb とタイプしてみよう。

```
$ lb
```

すると、今度は lualatex を使ってコンパイルが実行されている。

もしも、ソースファイルの名前を 'example.tex' とシた場合にあ、lb.conf の 1 行目を変更する。

```
rootfile=example
```

または

```
rootfile=example.tex
```

このように、拡張子は省略できる。

更に、すべての補助ファイルとターゲット・ファイルをソースディレクトリに起きたいときは 2 行目を変更する。Then change the second line in lb.conf to

```
builddir=
```

これで builddir には空文字列が指定される。それはビルド・ディレクトリを作成しない、ということを意味する。

`lb` を下記の `lb.conf` の下で実行してみよう。

```
rootfile=example
builddir=
engine=latex
latex_option=-halt-on-error
dvipdf=dvipdfmx
preview=evince
```

ここでは、エンジンは latex で dvipdf コンバータは dvipdfmx になっている。

```
$ rm -r _build
$ mv main.tex example.tex
$ lb
```

次のような内容を含むメッセージが表示される。

```
This is pdfTeX, Version 3.14159265-2.6-1.40.21 (TeX Live 2020)
 (preloaded format=latex)
  ... ...
  ... ...
Output written on example.dvi (1 page, 332 bytes).
Transcript written on example.log.
Latexmk: Examining 'example.log'
=== TeX engine is 'pdfTeX'
Latexmk: Log file says output to 'example.dvi'
Latexmk: All targets (example.dvi) are up-to-date
example.dvi -> example.pdf
[1]
3662 bytes written
```

これはエンジンが latex[*1] であることを示している。そして、pdf ではなく、dvi ファイルが生成される。その後、dvipdfmx が latexmk に呼び出され、dvi ファイルは pdf ファイルに変換される。メッセージには dvipdfmx の文字は現れないが、'example.dvi -> example.pdf' は dvipdfmx によって出力されたものである。したがって、ビルドの間に latexmk によって dvipdfmx が呼び出されたことが確認できる。

```
$ ls
example.aux  example.fdb_latexmk  example.log  example.tex
example.dvi  example.fls          example.pdf  lb.conf
```

ディレクトリのリストから、_build のような一時ディレクトリが生成されていないことがわかる。これは lb.conf の中で builddir に空文字列を指定したことによるものである。

lb の重要な特長にサブファイルを単独でコンパイルできることがある。これは、後ほどセクション**??** テス

[*1] Texlive2020 では、latex コマンドは、オリジナルの tex ではなく、pdftex をその代わりに用いている。

ト・コンパイル (p. ??) で説明される。

3　テンプレートの生成

3.1　newtex.conf

newtex スクリプトはディレクトリを作成し、その中にテンプレートファイルを生成する。これはコンパイルに先がけて最初に実行される。

はじめに、コンフィギュレーション・ファイルの newtex.conf を作成する必要がある。Buildtools の（ダウンロードした）ソースファイルの中に、newtex.ja.conf がある。このチュートリアルの実行用のディレクトリを作って、その中にこのファイルをコピーする。

```
$ cp newtex.ja.conf（作成したディレクトリ)/newtex.conf
```

コピーが完了したら、ファイルの内容を確認しよう。

```
# This is a configuration file for newtex.
# The name of this file is newtex.conf
# A string between # and new line is a commnet and it is ignored by newtex.
# Empty line is also ignored.

title="チュートリアル"
# document name

# lb.conf
# Lb.conf has six lines.
# The following six lines are copied to lb.conf.
rootfile=main.tex
builddir=_build
engine=lualatex
latex_option=-halt-on-error
dvipdf=
preview=evince

# documentclass
documentclass=ltjsarticle

# chapters/sections and subfile names
#   Chapters/sections and subfile names must be surrounded by double quotes.
#   Subfile names have no suffix or ".tex" suffix.
# If your LaTeX file is not big and no subfile is necessary, then leave out
 the following lines.
```

```
section="インストール" "installation"
section="lb を使ったコンパイル" "lb"
section="テンプレートの生成" "generate_templates"   # Subfiles are NOT allowed
 to include space characters. Use underscore instead of space.
section="TeX ソースファイルの編集" "edit_tex_files"
section="テスト・コンパイル" "test_compile"
section="プリプロセッシング" "preprocessing"
section="rake による自動化" "rake"
section="アーカイブの作成" "tarball"
```

ここではこのチュートリアルの文書自身をどのように作成するかを説明する。上記のファイルはこの作成時に用いた newtex.conf と全く同一の内容である。

ハッシュマーク（#）から改行まではコメントで、newtex はこの部分を無視する。空行も同様に無視される。残りの行が newtex への指示が書かれたものである。

それぞれの行は「キー=値」の形式になっている。使われているキーは、以下の通りである。

title　作成する文書の表題
rootfile　ルートファイルの名前
builddir　ビルド・ディレクトリの名前
engine　ソースファイルをコンパイルする latex エンジン
latex_option　latex エンジンに与えるオプション
dvipdf　dvi を pdf に変換するプログラム
preview　pdf ビューワ
documentclass　ドキュメントクラスの名前（\documentclass の引数）
chapter　章とそれに対応するサブファイル
section　セクションとそれに対応するサブファイル

ltjsbook のようなドキュメントクラスを使い、書籍のような大きな文書を作成する場合は、「chapter」と「section」キーを用いる。ltjsarticle のようなドキュメントクラスを使い、小さな（あるいはさほど大きくない）文書を作る場合は、「section」キーのみを用いる。

3.2　newtex の実行

newtex.conf の編集が終わり、保存したら、以下のようにタイプする。

```
$ newtex
```

すると、newtex は「チュートリアル」という newtex.conf に記された表題と同じ名前のディレクトリを作る。おしも、表題に半角の空白文字が含まれている場合は、それらはアンダースコア（_）に変換される。例えば、英語の表題で「A tutorial for beginners」は、「A_tutorial_for_beginners」に変換され、それがディレクトリ名となる。これは、空白文字を含むファイル名が問題を引き起こす場合があるための処理である。newtex はさらに雛形となるファイルをそのディレクトリの下に生成する。

```
$ cd チュートリアル
$ ls
Makefile            generate_templates.tex  preprocessing.tex
Rakefile            helper.tex              rake.tex
_build              installation.tex        tarball.tex
cover.tex           lb.conf                 test_compile.tex
edit_tex_files.tex  lb.tex
gecko.png           main.tex
```

いくつか重要なファイルを見てみよう。

```
$ cat lb.conf
rootfile=main
builddir=_build
engine=lualatex
latex_option=-halt-on-error
dvipdf=
preview=evince
```

このファイルの内容は、newtex.conf の一部のコピーである。

```
$ cat main.tex
\documentclass{ltjsarticle}
\input{helper.tex}
\title{チュートリアル}
\author{} % Write your name if necessary.
\begin{document}
\maketitle
% If you want table of contents here, uncomment the following line.
%\tableofcontents

\section{インストール}
  \input{installation.tex}
\section{lb を使ったコンパイル}
  \input{lb.tex}
\section{テンプレートの生成}
  \input{generate_templates.tex}
\section{TeX ソースファイルの編集}
  \input{edit_tex_files.tex}
\section{テスト・コンパイル}
  \input{test_compile.tex}
\section{プリプロセッシング}
```

```
  \input{preprocessing.tex}
\section{rake による自動化}
  \input{rake.tex}
\section{アーカイブの作成}
  \input{tarball.tex}
\end{document}
```

最初の行はドキュメントクラスの指定で、newtex.conf の dovumentclass キーの値が引数になっている。2 番めの行では helper.tex を取り込む input コマンドが書かれている。helper.tex は\usepackage コマンドでパッケージを取り込んだり、\newcommand コマンドでマクロを定義するなどの役割がある。プリアンブルの記述のほとんどは helper.tex に書かれている。ユーザが自分専用の helper.tex を作っておくのは、良い考えである。というのは、同じプリアンブルをいろいろな文書に使い回すことが多いからである。自分専用 helper.tex を持っているユーザはこのテンプレートを上書きすると良い。

4 行目は書き直しが必要である。例えば、

```
\author{関谷　敏雄}
```

のように。'\date'、'\thanks'、abstract 環境などを必要に応じて付け加えて良い。また、目次が必要であれば、8 行目のハッシュマークを外して、アンコメントする。残りの行は、セクションとそのセクションの内容を記述したサブファイルを取り込むための\input コマンドである。

Rakefile は rake に対する指示を書いたファイルである。当面はこれを書き換える必要はない。テンプレートをそのまま使い、rake を実行してみよう。

```
$ rake
 ... ...
 ... ...
$ ls
Makefile            generate_templates.tex  preprocessing.tex
Rakefile            helper.tex              rake.tex
_build              installation.tex        tarball.tex
cover.tex           lb.conf                 test_compile.tex
edit_tex_files.tex  lb.tex                  チュートリアル.pdf
gecko.png           main.tex
$ ls _build
main.aux          main.fls  main.out
main.fdb_latexmk  main.log  main.pdf
$ evince Tutorial.pdf
```

rake は lb を起動して main.tex をコンパイルし、その後「_build/main.pdf」を「チュートリアル.pdf」にコピーする。Evince は「チュートリアル.pdf」を次のように表示してくれる。

チュートリアル

2020 年 12 月 8 日

4 TeX ソースファイルの編集

main.tex には 8 つのセクションとそれに対応するサブファイルの取り込みが書かれている。対応するそれぞれのサブファイルは生成された直後は空のファイルである。そのサブファイルの編集が文書作成の主要な作業になり、作成にかかる時間の大部分がそれに当てられる。

最初のセクションとサブファイルは、それぞれ「インストール」と「installation.tex」である。おそらく、セクションの一部を編集した段階で、pdf ファイルがどのように出来ているかを見るためにテスト・コンパイルすることがあるだろう。多くの場合、編集とテスト・コンパイルの間を何回も行ったり来たりするものである。

もしも、文書がさほど大きくないならば、rake を用いるのがテスト・コンパイルには最も良い。というのは、さほどコンパイル時間もかからず、文書全体の pdf を見ることができるからである。このチュートリアルは、どちらかといえば小さい文書であるから、rake をテスト・コンパイルに用いるのが良い。

```
\subsection{動作条件}
Buildtools には次のものが必要である。
\begin{enumerate}
\item Linux OS と bash
\item LaTeX システム
\item Make または Rake
\end{enumerate}
 ... ...
 ... ...
```

編集が終わったら（もちろん、チュートリアルであるから、ソースファイルの installation.tex をコピーしても何ら差し支えない)、下記のようにタイプして pdf を見てみよう。

```
$ rake
$ evince チュートリアル.pdf
```

チュートリアル

2020 年 12 月 8 日

1　インストール

1.1　動作条件

Buildtools には次の各項目が必要である。

1. Linux OS と bash
2. LaTeX システム
3. Make または Rake

1.1.1　Linux OS と bash

Buildtools は Debian と Ubuntu で動作を確認している。おそらく他の linux ディストリビューションでも動作すると思われる。Buildtools のスクリプトでは bash のコマンドが使われているので、bash は必須である。

1.1.2　LaTeX システム

LaTeX のインストールには 2 つの方法がある。

ひとつは、ディストリビューションに含まれている LaTeX のパッケージをインストールする方法である。ubuntu ディストリビューションの場合は、次のようにタイプしてインストールする。

5　テスト・コンパイル

もし大きな文書を作成しているのであれば、例えば 100 ページを越える書籍を作るような場合、テスト・コンパイルに rake を使う（rake で文書全体をコンパイルするという意味）のは良い方法ではない。ドキュメントが大きくなればなるほど、コンパイル時間が長くなるからである。

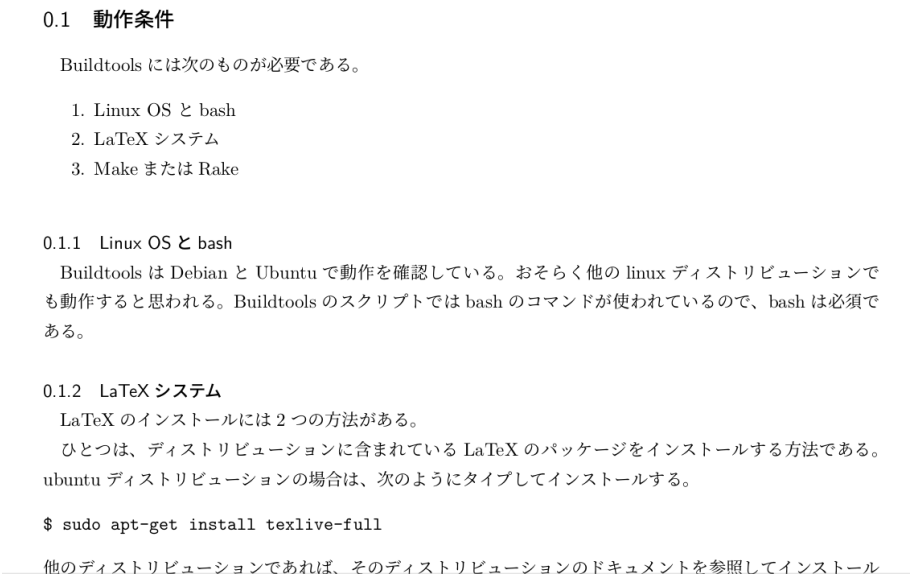
それよりは、lb を使ってサブファイルを単独でコンパイルするほうがより良い方法である。lb は一時的なルートファイル（テンポラリ・ルートファイルという）を作り、その中にサブファイルを取り込む命令を記述し、1 度だけコンパイルする。この方法の良い点は短時間でコンパイルできることである。しかし、良くない点もある。サブファイルのみのコンパイルであるから、文書全体の pdf を見ることはできない。また、1 度だけのコンパイルなので、相互参照は反映されない。どちらが良いのかは一概に行くことは出来ない。それは作成している文書によるが、もしそれが非常に大きな文書であれば、テスト・コンパイルを lb を使って部分的にやるほうがより良いといえる。

次のようにタイプしてほしい。

```
$ lb installation
```

この引数はサブファイル名である。拡張子は省略することもできる。

すると、lb はテンポラリ・ルートファイル「_build/test_installation.tex'」を作る。そのプリアンブルは元のルートファイルのプリアンブルのコピーである。そして、\input コマンドでサブファイルの「installation.tex」を取り込むようになっている。lb は「_build/test_installation.tex」をコンパイルし、lb.conf で指定された pdf ビューワを起動して pdf ファイルを表示する。



0.1 動作条件

Buildtools には次のものが必要である。

1. Linux OS と bash
2. LaTeX システム
3. Make または Rake

0.1.1 Linux OS と bash

Buildtools は Debian と Ubuntu で動作を確認している。おそらく他の linux ディストリビューションでも動作すると思われる。Buildtools のスクリプトでは bash のコマンドが使われているので、bash は必須である。

0.1.2 LaTeX システム

LaTeX のインストールには 2 つの方法がある。

ひとつは、ディストリビューションに含まれている LaTeX のパッケージをインストールする方法である。ubuntu ディストリビューションの場合は、次のようにタイプしてインストールする。

```
$ sudo apt-get install texlive-full
```

他のディストリビューションであれば、そのディストリビューションのドキュメントを参照してインストール

lb はコンパイルするときに synctex のオプションをオンにする。したがって、ソースファイルの「_build/test_installation.tex」と「installation.tex」をエディタで開いておけば、ソースと pdf の間で前方参照と後方参照をすることができる。もしも gedit をエディタに、evince をビューワに使っていれば、コントロール・キーを押したまま左クリックすれば後方参照（pdf からソースへの参照）が可能である。しかし、「installation.tex」から pdf への前方参照は働かない。それを機能させるためには、サブファイルの先頭に次の 1 行を加えなければならない。

```
% mainfile: _build/test_installation.tex
```

しかし、前方参照を使うことは後方参照に比べそう多くはない。上記の 1 行を加えるのは、たいていは必要ないだろう。

6 プリプロセッシング

ときには latex ソースファイルをコンパイルする前に何かしておきたい、ということがあるかもしれない。例えば、

- gnuplot などのプログラムを使って画像ファイルを生成したい
- pandoc を使って latex ソースファイルを自動生成したい

このチュートリアルではプリプロセッサとして pandoc を使う方法を紹介したい。pandoc は文書コンバータである。マークダウン、latex、html、pdf など多くの種類の文書を変換することができる。このチュートリ

アルでは、Buildtools のソースファイルの中にある「Readme.ja.md」が「readme.tex」という latex ソースファイルに変換される。

たいていのディストリビューションでは pandoc パッケージが備わっているので、インストールは簡単である。例えば ubuntu では

```
$ sudo apt-get install pandoc
```

でインストールできる。

readme.tex を生成するには次のようにタイプする。

```
$ pandoc -o readme.tex ../Readme.md
```

この他に 2 つほど main.tex と helper.tex に変更を加える必要がある。まず、\input コマンドを用いて readme.tex を取り込む命令を main.tex の最後に記述する。

```
\documentclass{article}
\input{helper.tex}
 ... ...
 ... ...
\section{Make tarball}
  \input{tarball.tex}
\input{readme.tex}
\end{document}
```

更に、8 行目の\tableofcontents をアンコメントしてコマンドが利くようにしよう。

```
 ... ...
\maketitle
% If you want a table of contents here, uncomment the following
 line.
\tableofcontents
 ... ...
```

2 番めは、\tightlist を万度を定義することが必要である。Helper.tex がその定義を書くのに最も適した場所である。

```
 ... ...
 ... ...
\providecommand{\tightlist}{%
  \setlength{\itemsep}{0pt}\setlength{\parskip}{0pt}}
 ... ...
 ... ...
```

このコードは https://github.com/jgm/pandoc-templates/blob/master/default.latex から引用したものである。

rake を使ってコンパイルする。

```
$ rake
```

目次

これで、目次にセクション 9、そして「Readme.ja.md」の内容が表示された。

このセクションでは pandoc を手動で走らせた。もしも、Readme.ja.md がアップグレードされたならば、再び pandoc を実行しなければならない。それは面倒なことであり、本来自動化されるべきことである。ひとつの方法は Rakefile を変更して rake がコンパイル前に自動的にプリプロセッシングするようにすることである。次のセクションでその方法を説明する。

7 rake による自動化

rake は make に似たビルド・ツールである。Rakefile には、rake がソースファイルをビルドする手順が書かれている。Rakefile には任意の ruby コマンドを記述することができる。したがって、そのビルドの過程が、たとえ複雑であってもそれを記述することが可能である。

newtex は自動的に Rakefile を生成するが、プリプロセッシングが無ければ、そのままで十分使うことができる。前セクションでは、readme.tex を生成するために pandoc を用いたプリプロセッシングが行われた。したがって、Rakefile を変更して pandoc をその中に記述する必要がある。以下のように Rakefile を変更する。
In the previous section, we used pandoc to generate readme.tex. So, we need to modify Rakefile to put in pandoc. Modify the Rakefile as follows.

```
require 'rake/clean'
```

```
# if readme.tex doesn't exist, generate it first.
# This is necessary because readme.tex is accessed by gfiles in
#  line 12.
if File.exist?("readme.tex") == false
  sh "pandoc -o readme.tex ../Readme.md"
end
# use Latex-BuildTools
@tex_files = (`tfiles -a` + `tfiles -p`).split("\n")
@tex_files <<= "readme.tex"
@graphic_files = []
@tex_files.each do |file|
  @graphic_files += `gfiles #{file}`.split("\n")
end

task default: "チュートリアル.pdf"

file "チュートリアル.pdf" => "_build/main.pdf" do
  sh "cp _build/main.pdf チュートリアル.pdf"
end

file "_build/main.pdf" => (@tex_files+@graphic_files) do
  sh "lb main.tex"
end

file "readme.tex" => "../Readme.ja.md" do
  sh "pandoc -o readme.tex ../Readme.ja.md"
end

CLEAN << "_build"
task :clean

task :ar do
  sh "arl main.tex"
  sh "tar -rf main.tar Rakefile"
  sh "gzip main.tar"
  sh "mv main.tar.gz チュートリアル.tar.gz"
end

task :zip do
```

```
  sh "arl -z main.tex"
  sh "zip main.zip Rakefile"
  sh "mv main.zip チュートリアル.zip"
end
```

この変更のおかげで、pandoc を手動で走らせる必要はなくなった。やらなければいけないのは、ただ単に「rake」とタイプするだけである。

ruby と rake に関するウェブサイトはいくつかある。例えば、

- https://www.ruby-lang.org/en/
- http://rubylearning.com/
- https://ruby.github.io/rake/

がその主なもので、参照してほしい。

8　アーカイブの作成

ソースファイルを配布したい、という場合もあるかもしれない。そのようなときには、それをアーカイブすることが必要になる。Buildtools に含まれるスクリプトの arl は、ルートファイルが取り込むサブファイルや画像ファイルを検索し、それらアーカイブする。このアーカイブされたファイルのうち、tar というコマンドで作られたものを tarball という。

- -g オプションが与えられると、gzip で圧縮された tarball を作る。
- -b オプションが与えられると、bzip2 で圧縮された tarball を作る。
- -z オプションが与えられると、zip ファイルを作る。
- オプションが与えられなければ、非圧縮の tarball を作る。

もし、プリプロセッシングで生成される latex ソースファイルなどがある場合は、arl の実行前にそれらを生成しておかなければならない。

```
$ arl
$ tar -tf main.tar
main.tex
edit_tex_files.tex
generate_templates.tex
installation.tex
lb.tex
preprocessing.tex
rake.tex
readme.tex
tarball.tex
test_compile.tex
helper.tex
```

```
Tutorial_1.png
Tutorial_2.png
hellolatex.png
tableofcontents.png
test_installation.png
```

Rakefile も当然ながら、tarball に含めなければならない。

```
$ tar -rf main.tar Rakefile
```

そして、gzip などに圧縮して tarball が完成する。

```
$ gzip main.tar
```

以上の手続きは、実はすでに Rakefile に記述されている。「rake ar」とタイプすることにより、rake が tarball を自動生成してくれる。

```
$ rm main.tar.gz
$ rake ar
arl main.tex
tar -rf main.tar Rakefile
gzip main.tar
mv main.tar.gz チュートリアル.tar.gz
```

最後にできあがる tarball の名前は「チュートリアル.tar.gz」である。

もしも、zip ファイルを作りたければ、「rake zip」とタイプする。

このセクションで、チュートリアルは終わりである。次のセクションは Buildtools のソースファイルに含まれる Readme.ja.md のコピーである。この文書には、Buildtools を作成した背景や個々のスクリプトの特長と使い方が書かれている。

9 Buildtools

9.1 Buildtools 作成の背景

■Latextools と Buildtools の関係　大きな文書、例えば 100 ページを越える本などを LaTeX で作る場合は、小さい文書の作成と異なる様々な問題を考える必要がある。

- ソースファイルの分割
- 分割コンパイル
- 文書の一括置換
- 前処理（LaTeX コンパイル前に処理する作業）

これらを支援するツール群が Latextools であり、2 つのグループに別れている。

- Buildtools。ソースファイルの新規作成、ビルド、分割コンパイルを支援するツール群

- Substools。ソースファイルに対する一括置換をするツール群

このように、Buildtools は Latextools を構成する一部であるが、同時にその中核をなすツール群である。このドキュメントでは Buildtools を解説する。

■ソースファイルの分割　LaTeX ソースファイルを単にここではソースファイルと呼ぶ。大きな文書を 1 つのソースファイルに記述するのは適切でない。なぜなら、ファイルが大きくなると、エディタで編集するのが極めて困難になるからである。そこで、文書を分割することになる。通常は`\begin{document}`と`\end{document}`を含む 1 つのファイルと、そのファイルから`\include` または`\input` で呼び出される複数のファイルに分割する。前者をルートファイル、後者をサブファイルという。`\include` と`\input` はどちらもサブファイルの取り込みのコマンドだが、違いがある。

- `\include` はネストはできない。`\include` はボディ（`\begin{document}`と`\end{document}`の間の部分）にのみ書くことができる。`\includeonly` でファイルのリストが指定されている場合は、そのリストに書かれているファイルのみが`\include` で取り込め、リストにないファイルの`\include` は無視される。`\includeonly` はプリアンブルのみに書くことができる。`\include` はファイルを取り込む前に`\clearpage` をする。
- `\input` は単にファイルを取り込むだけで`\clearpage` はしない。このコマンドはネストできる。

文書をコンパイルによって作成することをビルドと呼ぶ。ビルドは LaTeX のコンパイルだけでなく、前処理（例えば gnuplot による画像生成だったり、データから tikz のグラフを生成したりなど）も含める。ビルドは最終的にルートファイルをコンパイルすることによって完了する。LaTeX ソースファイルをコンパイルするプログラムにはいくつかあり、それをエンジンと呼んでいる。Buildtools でサポートしているエンジンには、latex、platex、pdflatex、xelatex、lualatex がある。コンパイルは文書が大きくなればなるほど時間がかかるようになる。1 箇所だけを変更する場合も同じだけ時間がかかる。文書の編集の途中で出来栄えをチェックするためにコンパイルすること（これをテストするなどということがある）は頻繁に起こるが、そのたびに長い時間がかかる。文書が大きくなればなるほどこの問題は深刻になる。そこで、サブファイルのみをコンパイルできるようにする方法がいろいろ考えられた。

- `\includeonly` コマンドの引数（サブファイルのリスト）からコンパイルしたくないファイルをコメントアウトする
- subfiles パッケージを用いる

とくに subfiles パッケージはよくできており、推薦する方も多い。ただ、パッケージを取り込み、そのコマンドを適切に使うことが必要である。もちろん、上記のコンパイル時間の問題からすれば、それくらいは何ともないことであるが。

これ以外の方法としては、サブファイル単独のコンパイル用に特別のプリアンブル部などを付け加える方法がある。具体的には、サブファイルを「`\documentclass` から`\begin{document}`まで」と「`\end{document}`」でサンドイッチにする。そのときにサブファイル自体を変更するのではなく、プリアンブルなどを記述したファイルからサブファイルを`\input` で取り込むようにする。そのサブファイルを取り込むファイルを「仮のルートファイル」と呼んだりする。それに対して元々のルートファイルを「オリジナルのルートファイル」と呼ぶこともある。仮のルートファイルのプリアンブル部は、オリジナルのルートファイルのコピーである。この方法の良い点は

- ソースファイルにパッケージの取り込みや特別な命令を書き込む必要がない。
- したがって、ソースファイルを配布するにあたって、被配布者に特定のパッケージをインストールさせる必要がない。

つまり、ソースファイルをなんら変更することなく、サブファイルのみのコンパイルが可能だというのが長所である。ただ、そのためには仮のルートファイルを生成するプログラムが必要である。Buildtools では、`ttex` というシェルスクリプトでそれを行っている。

■コンパイルのリピート（繰り返し） 最終的に文書をコンパイルするときには、相互参照などを実現するためにコンパイルを複数回行わなければならない。その回数は 2 回であったり、3 回であったりするらしいが、筆者はその事情をよく知らない。が、ここに優れたプログラムがあり、必要回数を判断し、必要なだけコンパイルしてくれる。latexmk というプログラムである。latexmk を用いることによって、ビルドは非常に楽になる。Buildtools ではルートファイルのコンパイルに latexmk を使用する。

■作業用ディレクトリの設置 LaTeX でコンパイルすると、様々な補助ファイルやログファイルが生成されるので、きれい好きの人はそれについて不満を感じることがあると思う。それで、作業用ディレクトリを設置して、そういうファイルの一切合財を入れてしまうとソースディレクトリをきれいに保つことができる。そういうソフトに cluttex があり、きれい好きな人にはお勧めである。Buildtools では作業用ディレクトリ `_build`（名前は変更も可能）を設置し、補助ファイルなどを格納する。このことにより、ソースディレクトリを汚さずに済む。また、コンパイルのログや補助ファイルを見たいときは `_build` の中を見れば良い。非常に単純なのである。蛇足になるが、最近 C のビルドツールとして人気のある meson も作業ディレクトリを使う。これは、一般に作業用ディレクトリをソースディレクトリと区別することが人間にとって非常に分かりやすくなるということの表れである。

Buildtools では、生成された最終文書（pdf ファイル）も作業用ディレクトリにできあがる。それをソースファイルディレクトリに置きたいというのは自然は発想だが、それには make または rake を使うと良い。rake はスクリプト言語 ruby で書かれた make とでもいうべきものであるが、特長は Rakefile（rake のスクリプト）に ruby 言語を使うことができることである。そのことによって、make よりもはるかに強力で分かりやすい記述ができる。話を元に戻すが、最終文書をソースディレクトリに置くには、Makefile または Rakefile に、作業用ディレクトリからソースディレクトリに最終文書をコピーするコマンドを書いておくのである。また、make や rake を使うことの利点は、前処理を記述できることである。前処理はその文書やユーザの使うツールに依存するので、Buildtools でカバーするのは難しい。それに比べて、Makefile や Rakefile はフレキシブルなので、前処理を記述するのに適しているのである。

Buildtools では、make または rake を併用することを推奨している。

■Texworks との連携 Buildtools では、`lb` というコマンドでルートファイルのビルドもサブファイルのテストコンパイルも行うことができる。それを Texworks のタイプセッティングに登録すると、Texworks から起動できて大変便利である。「設定」ー>「タイプセッティング」タブで設定する。+ をクリックして新たなコマンドを設定する。名前=>`lb`、コマンド=>`lb`、引数=>`$fullname` で良い。設定後はルートファイルで全体のコンパイル、サブファイルは単独のテスト・コンパイルがワンクリックでできるようになる。

9.2 Buildtools の構成

Buildtools は大きく分けて次のような部分から構成されている。

- newtex: 新規ソースファイルの作成を支援するツール。
- lb: Latex Build。ルートファイルをコンパイル、または ttex を使ってサブファイルのテストコンパイルをする。
- アーカイブ作成を支援するツール。
- インストーラ
- ユーティリティ群。上記のプログラムを下支えする。

文書作成は、次の手順で行われる。

1. 文書全体の構成を決める。章立てを決める。
2. newtex を使ってフォルダとソースファイルの雛形を作成する。
3. Makefile、Rakefile、表紙（cover.tex）、プリアンブル部（helper.tex）の雛形を必要に応じて書き換える。
4. 本文の作成と、テストコンパイル。
5. 前処理の作成。
6. 最終ビルド。

上記の 3 から 5 は行ったり来たりすることになり、必ずしもこの順に作業が進むわけではない。

9.3 主要なツールについて

それぞれのツールの簡単なヘルプは--help オプションをつけて実行することにより表示される。例えば、newtex に--help オプションをつけて実行すると、次のようなメッセージが表示される。

```
$ newtex --help
Usage:
  newtex --help
    Show this message.
  Newtex.conf needs to be edited before running newtex.
  newtex
    A directory is made and some template files are generated under the directory.
```

各ツールを説明する文書は

- 各ツールの--help オプションによるヘルプ・メッセージ
- このドキュメントにおける以下の記述

だけである。より詳細を知りたい場合はソースコードを見ていただきたい。Buildtools のすべてのツールはシェル・スクリプトで書かれている。それぞれのスクリプトは短く、シェル・スクリプトをご存知の方であれ

ば、比較的簡単にソースコードを理解できる。

■newtex

```
$ newtex
```

新規に LaTeX の文書を作るときに使うスクリプト。newtex を使う前に全体の構成と章立てを決めておき、それを事前に newtex.conf に書いておく。このスクリプトは、newtex.conf に書かれた指示に従って、新しくディレクトリを作り、テンプレート・ファイルを生成する。

このプログラムは 2 回に分けて使う。

1. Buildtools のソースファイルの中に newtex.conf ファイルがある。これを書き直して、ユーザの環境やこれから作る LaTeX ファイルに合うようにする。
2. newtex を実行する。このスクリプトは、newtex.conf の中で指定されたタイトル名と同じ名前のディレクトリを新たに作成する。ただし、タイトル中の空白文字はアンダースコアに変換されてディレクトリ名となる。スクリプトは、そのディレクトリの下にテンプレート・ファイルを生成する。

■lb

```
$ lb [LaTeXfile]
```

引数省略の場合は main.tex が引数で与えられた場合と同じ動作をする。lb は引数の LaTeX ファイルをビルドするスクリプトであり、これだけで足りることが多い。

- 引数がルートファイルの場合はそれを latexmk を使ってビルドする。サブファイルの場合は ttex でビルドする。
- 引数がルートファイルの場合は、synctex を使わない。
- 引数がサブファイルの場合は、synctex を使い、コンパイル後に lb.conf で指定されたプリビューワを起動する。
- カレント・ディレクトリ（通常はルートファイルのあるディレクトリになる）に lb.conf があれば、それを読み込んで変数の初期化をする
- lb.conf でエンジン指定を省略すると lb が自分でエンジンを予測する。しかし、lb.conf でエンジンを指定するほうが好ましい。

lb.conf で初期値の設定ができる。

```
rootfile=main.tex
builddir=_build
engine=
latex_option=-halt-on-error
preview=texworks
```

- rootfile はルートファイルの名前。ただし、lb の引数でルートファイルを指定した場合は、引数を優先する。
- builddir は作業ディレクトリを指定する。そのディレクトリには補助ファイルや出力ファイル、対象

がサブファイルの場合は仮のルートファイルが出力される。空文字列を指定すると、作業ディレクトリは生成されず、ソースファイルの置かれているディレクトリが作業ディレクトリになる。

- `engine` は LaTeX エンジンを指定する。latex、platex、pdflatex、xelatex、lualatex を指定することができる。その他のエンジンはサポートしていない。
- `latex_option` は `latexmk` を通じてエンジンに与えるオプション。`lb.conf` が存在しない場合でも、`-output-directory` は `lb` が自動的にエンジンに与える。
- `preview` はできあがった pdf を見るためのプリビューワ。ただし、サブファイルのときのみ動作する。

■arl　$ arl [-b|-g|-z] [rootfile]

`arl` という名前は、ARchive LaTeX files から。ルートファイルの関連ファイル（下記参照）を検索してアーカイブを作る。ルートファイルが省略された場合は、`main.tex` を指定されたものとして処理する。

- 前処理プログラムがある場合、そのプログラムを実行してから `arl` を起動する必要がある。
- `arl` がアーカイブするのは、LaTeX ソースファイルと、`includegraphics` される画像ファイルのみ。
- したがって `Makefile` や、前処理のソースファイル（例えば gnuplot のソース）などはアーカイブされない。

Makefile にターゲットを作り（例えば `ar` という名前のターゲット）、`arl` で作ったアーカイブに tar で Makefile や前処理ソースファイルを追加するスクリプトを書いておくと便利である。同様のことは Rakefile でもできる。

アーカイブを圧縮するオプション`-g`、`-b`、`-z` で、それぞれ、tar.gz, tar.bz2, zip をサポート。オプションが与えられなかった場合は、圧縮なしの tarball を作る。

■ユーティリティ群　この項はスクリプトをメンテナンスするのでなければ読む必要はない。

```
$ srf subfile
```

`subfile` からルートファイルを探し、その結果（絶対パス）を出力する。`srf` は「Search Root File」の意味。

```
$ tfiles [-p|-a|-i] [rootfile]
```

rootfile のサブファイルの一覧を取得する。引数のルートファイルが省略された場合は、`main.tex` が指定されたものとして処理する。

- オプション無し => ルートファイルが取り込むサブファイル（`\begin{document}`から `\end{document}`までの `include` または `input` コマンドで指定されたファイル）のリストを標準出力に出力する
- `-p` プリアンブルで取り込まれるサブファイルのリストを標準出力に出力する
- `-a` オプション無しのリストにルートファイルを加えて標準出力に出力する
- `-i include` コマンドで取り込まれるファイルのみを標準出力に出力する。ただし、`includeonly` で指定されなかったファイルは除かれる。

注意：出力されるファイルのリストは改行で区切られている。

```
$ tftype [-r|-s|-q] files ...
```

LaTeX のソースファイルの種類を調べるスクリプト。

- -r（デフォルト）引数のファイルの中からルートファイルのみを抽出して出力する
- -s 引数のファイルの中からサブファイルのみを抽出して出力する
- -q（quiet）上記の出力を抑制する。引数は 1 つのファイルのみで、そのファイルタイプを exit ステータスで返す。exit ステータスが 0 はルートファイル、1 はサブファイル、エラーが生じた場合は 2 となる。

-q オプションを使うことが最も多い。

```
$ gfiles files ...
```

引数は、latex のソースファイル（の列）である。与えられたファイルの中で\includegraphics によって取り込まれる画像ファイルの一覧を返す。

```
$ ltxengine rootfile
```

コンパイルを行う LaTeX エンジンを予想する（本来ユーザが明示すべきだが・・・）例えば、

```
\usepackage[luatex]{graphicx}
```

というコマンドがプリアンブルにあれば、エンジンは lualatex と予想がつく。

```
$ ttex [-b builddir] -e latex_engine [-p dvipdf] [-v previewr] -r rootfile subfile
```

サブファイルに仮ルートファイルをつけてコンパイルする。コンパイルは 1 回だけ。そのため相互参照は反映されない。（これはテストのためのスクリプトであって、最終仕上げではないから相互参照はさほど重要ではない、という考えに基いている）。また、該当のサブファイル以外のファイルにあるラベルを参照することはできない。単独で使うことも可能だが lb を通して呼び出すのが普通の使い方。オプションについては下記の通り。

- -b 作業ディレクトリを指定する。デフォルトは_build である。
- -e latex エンジンを指定する。エンジンの種類についての制限はないが、latex、platex、pdflatex、xelatex、lualatex のいずれかが指定されることを想定している。
- -p エンジンが latex または platex である場合は、dvi ファイルが出力される。その dvi から pdf を出力するためのアプリケーションを指定する。デフォルトは dvipdfmx である。その他に、dvipdfm や dvipdf を指定することができる。
- -v プリビューアを指定する。evince など、pdf を表示できるアプリケーションを指定する。ソースファイルを texworks で編集している場合は、ここに texworks を指定するのが良い。
- -r ルートファイルを指定する。

9.4 インストールとアンインストール

■インストールに必要な環境

- Linux と bash。Debian と Ubuntu ではテストされているが、おそらくその他の linux ディストリビューションでも動作すると思われる。Bash コマンドを用いてスクリプトが記述されているので、bash は必要である。
- LaTeX システム。LaTeX のインストールには 2 つのオプションがある。1 つはディストリビューションに付属のシステムをインストールすることである。他方は TexLive をインストールすることである。
- make または rake。これらのツールは Buildtools にとって、必ずしも必要というわけではない。しかし、make または rake のもとで、Buildtools を実行することが望ましい。この 2 つに両方をインストールする必要はない。どちらか 1 つを選んでインストールすれば良い。make は長く使われているビルド・ツールで、元々は C コンパイラの制御に用いられてきた。rake はこれに似たツールで、ruby で書かれたアプリケーションである。rake を使うことの利点は、そのスクリプトである Rakefile の中で任意の ruby コードを記述できることである。一般に、Rakefile は Makefile よりも読みやすく、理解しやすい。

■インストール　インストール用のスクリプト `install.sh` を使う。

```
$ bash install.sh
```

シェルスクリプトなどの実行ファイルは`$HOME/bin` に保存される。debian や ubuntu では、ログイン時に `$HOME/bin` があれば、bash の実行ディレクトリのパスを表す環境変数 `PATH` に追加される。インストール時に新規に `$HOME/bin` を作成した場合には、再ログインしないと、それが実行ディレクトリに追加されないので注意が必要。root になってインストールすると`/usr/local/bin` に実行ファイルをインストール。debian の場合は、

```
$ su -
# bash install.sh
```

ubuntu の場合は

```
$ sudo bash install.sh
```

■アンインストール　アンインストールは `uninstall.sh` で行う。一般ユーザで実行すれば、`$HOME` 以下のインストールファイルが削除される。

```
$ bash uninstall.sh
```

root で実行すれば、`/usr/local` 以下のインストールファイルが削除される。debian の場合は、

```
$ su -
# bash uninstall.sh
```

ubuntu の場合は、

```
$ sudo bash uninstall.sh
```