final_thesis

2018 年 2 月 9 日

Table of Contents

1 はじめに

1.1 目的

西谷研究室で使われているユーザメモソフト，my help の振る舞いを制御しているサブコマンドは，マイナスを付した省略記法が取られている．プログラミング初心者にとってこの省略記法は，覚えにくかったりわかりにくかったりするという問題があり，現在はフルワードを使った自然言語に近い記述法が多く用いられている．そこで，本研究ではコマンドラインツール作成ライブラリを自然言語に近いサブコマンド体系を実装しやすいライブラリである Thor に変更する．ruby の標準ライブラリである optparse で作成されている my help を Thor によって書き直し，異なった 2 つライブラリで作成された my help の使用感を比較検討することを目的とする．

2 基本的事項

2.1 my_help

2.1.1 my_help の目的

my help とは，ユーザー独自のマニュアルを作成することができるユーザメモソフトである．これは，terminal だけを用いて簡単に起動，編集，削除などをすることができるため，非常に便利である．さらに，そのマニュアルは自分ですぐに編集，参照することができるので，メモとしての機能も果たしている．これにより，プログラミング初心者が，頻繁に使うコマンドやキーバインドなどをいちいち web browser を立ち上げて調べるのではなく，terminal 上で即座に取得できるため，プログラム開発を集中することが期待される．

また，正式なマニュアルは英語で書かれていることが多く，初心者には理解するのが困難である．my_help を使用すれば，自分なりのマニュアルを作成することができるので，仕様を噛み砕いて理解することができる．tarminal 上でいつでもメモを参照できるため，どこにメモをしたかを忘れるリスクも軽減される．

2.1.2 使用法

インストールする方法だが，gem の標準とは少し方法が異なっている．まず，github にある my help のリポジトリをフォーク，クローンすることでローカル（ネットワークに繋がれていない環境）でも my help を操作することができるようになる．

```
git clone git@github.com:daddygongon/my_help.git
```

これ以降の作業は bundle にて行っていく．

```
bundle update
```

を実行することで my_help.gemspec に記述されている必要な gems が bundle される．ここで Could not locate Gemfile とエラーが出た場合は、Gemfile のある場所を探し、その配下に移動してから再びコマンドを入力する．

```
bundle exec exe/my_help
```

で my help に用意されているコマンドを参照することができる．

Usage: my_help [options] -v, --version show program Version. -l, --list 個別 (specific) ヘルプの List 表示. -e, --edit NAME NAME(例：test_help) を Edit 編集. -i, --init NAME NAME(例：test_help) の template を作成. -m, --make make executables for all helps. -c, --clean clean up exe dir. --install_local install local after edit helps --delete NAME delete NAME help

デフォルトで emacs help という emacs のヘルプが用意されている．これは emacs help の他に，省略形の e_h でも表示されるようになっている．

次に，独自のヘルプを作成する方法であるが，まず，

bundle exec exe/my_help -i new_help

とすることで new help という名前のヘルプが作成され，そこにテンプレートが格納される．また，

bundle exec exe/my_help -e new_help

で，自分の好きなように編集することができる．ヘルプが完成したら，

bundle exec exe/my_help -m

とすることで exe ディレクトリに new help が追加され，new help，n_h が使用可能になるという手順である．

2.2 optparse

今回の研究対象の my help は，optparse で実装されている．optparse は Ruby でコマンドラインのオプションを操作するためのライブラリである．optparse が操作するオプションは，下記の on メソッドで設定する．

```
def execute
  @argv << ' --help' if @argv.size==0
  command_parser = OptionParser.new do |opt|
    opt.on('-v', '--version','show^^e2^^90^^a3program^^e2^^90^^a3Version.'
) { |v|
      opt.version = MyHelp::VERSION
      puts opt.ver
    }
    opt.on('-l', '--list', 'list^^e2^^90^^a3specific^^e2^^90^^a3helps'){list_helps}
    #中略
  end
  #中略
end

def list_helps
#中略
end
#後略
```

第 1 引数はショートオプションで，-a や-d のような形で設定する．同様にして，第 2 引数はロングオプションを表し，--add や--delete のように，第 3 引数はそのオプションの説明文で，help で表示される説明文を設定する．後ろのブロックには，そのオプションが指定された場合に実行されるコードを記述する []．しかしこのライブラリでは自然言語に近い，ハイフンなしのサブコ

マンドを実装するには相当な書き換えが必要となる．

メソッドの引数でオプションを定義し，引数が指定された時の処理をブロックで記述する．ブロックの引数にはオプションが指定されたことを示す true が渡される．on メソッドが呼ばれた時点ではオプションは実行されず，定義されるだけである．parse が呼ばれた際，コマンドラインにオプションが登録されていれば実行される．

オプション定義の際，スペースの後に任意の文字を追加すると，そのオプションは引数を受け取るオプションになる．その文字に [] をつけることで引数は必須でなくなる．また引数がハイフンで始まる場合，オプションとの間にハイフンを 2 つ挟むことで引数として認識される．

help と version のサブコマンドはデフォルトで作成される．

2.3 thor

本研究では optparse の代わりのライブラリとして Thor の採用を検討する．Thor は，コマンドラインツールの作成を支援するライブラリであり，git や bundler のようにサブコマンドを含むコマンドラインツールを簡単に作成することができる [3]．Thor には以下のような特徴がある． 1. コマンドラインオプションのパーズやサブコマンドごとのヘルプを作るなどの面倒な作業を簡単にこなすことができ，手早くビルドツールや実行可能なコマンドを作成できる []． 1. 特殊な DSL(Domain Specific Language) を使わずにメソッドを定義することで処理を記述するため，テストを行いやすい []． 1. optparse では作成することが困難な，マイナスを伴わない (自然言語に近い) サブコマンドを実装することが可能である．

```
desc 'list, --list', 'list specific helps'
    map "--list" => "list"
    def list
      print "Specific help file:\n"
      local_help_entries.each{|file|
        file_path=File.join(@local_help_dir,file)
        help = YAML.load(File.read(file_path))
        print "  #{file}\t:#{help[:head][0]}\n"
      }
    end
end
```

optparse では on メソッドでコマンドの登録を行い，その後の def でコマンドの振る舞いを定義している．それに対して Thor は登録と定義を同時に行うことができるのでコードが書きやすい．また，Thor を継承したクラスのパブリックメソッドがそのままコマンドになるので非常に簡単にコマンドを作成することが可能である．Thor はコマンドを作成した時点で，自動的にヘルプを生成してくれるため，楽であり，コマンドの指定せずにコマンドラインアプリを実行するとヘルプを表示する．

2.4 option

今日，複雑な機能を持つコマンドが増加している．そのようなコマンドは，オプション（サブコマンド）を使用することで適切な動作を実行することが可能になる．例えば git コマンドはオプションなしでは意味をなさない．オプションでどのような動作をするかが決まるので，オプションを入力することで正常に動作するのである．

そもそも，オプションにはショートオプションとロングオプションの 2 種類がある．ショートオプションはハイフンの後に英字 1 字を付けた形式のもので，-a や-v などといったものがショートオプションである．また，ショートオプションは 2 つ以上のオプションを 1 つにまとめて実行することもできる．例えば，-l，-a，-t の 3 つのオプションを 1 つにまとめて-lat として実行することが可能である．それに対してロングオプションは，ハイフン 2 つの後に英字 2 字以上を付けることができる形式である．例えば--all や--version などである．ロングオプションは英字を 2 字以上使用することができるので，どのようなオプションであるかを明確にするために一般的にフルワードが採用されている．--no-の形にすることで否定形のオプションを作成することも可能である．ロングオプションはショートオプションのように複数のオプションを 1 つにまとめることは不可能であり，1 つ 1 つをスペースで区切る必要がある．

ショートオプションを設定する際，英字 1 文字しか使用することができないので基本的には対応づけられたロングオプション（そのオプションがどのような動作を行うのかを表す単語）の頭文字であることが多く，慣れている人であればショートオプションを使うことで素早くアプリケーションを動かすことが可能である．しかし，用いるアプリが 1 つであるとは限らないし，全てのアプリのショートオプションを統一するのも困難である．また initial と install など，頭文字が重複してしまう 2 つのオプションがある場合，-i というショートオプションが initial を意味するのか install を意味するのかを判断するのは，初心者には容易ではなく，混乱を引き起こしかねない．そのため，ヒューマンエラーを引き起こしてしまったり，学習コストがかかってしまったりすることがある．

ショートオプションの場合，複数のオプションを一つにまとめることができると記述したが，これは引数を必要としないオプションの場合である．引数を必要とするオプションの場合，2 文字目以降の英字は引数扱いになってしまう．例えば上に示した-lat において，-l が引数を必要とするオプションであれば，-lat は at という引数が与えられた-l という風に解釈されてしまうので注意が必要である．そういった点においてもショートオプションは初心者にとって扱いにくい形式であると言える．

Command line application のオプションの記述方法には幾つもの流儀があるようで何らかの標準があるわけではない．しかし，それら全てに対応することはできず，なんらかの基準に従ってオプション記法を解釈する必要がある．

ここでは， "Build awesome command-line application in ruby 2"に従ってオプション記法と用語をまとめておく．

コマンドラインの基本形は，

```
ls -lat dir_name
```

というように

executabel options arguments

での形であった．これは，GNU 標準に基本構造が記載されている．

その後，幾つかの switch や flag を組み合わせて，複雑な命令を解釈できるようにするに従って，

grep --igonre-case -C 4 "some string" /tmp

などとほぼ呪文のような形態となってきた．

その後，command line suite と呼ばれる一群の command line application が登場した．典型的なのは git である [awesome, pp.14-5].

git は linux のバックアップを分散処理するために， Linus Torvalds が開発したものであるが，いくつもの機能に従ってそれぞれ個別のコマンドが用意されていた．それぞれ，git-commit とか git-fetch などであった．それが，あるとき，すべてをまとめて suite としてパッケージし直され，1 つのまとまった command として提供された．すなわち，git commit や git fetch などである [awesome, p.16]

3 研究内容

3.1 thor 化に伴う書き換え（具体的な作業）

optparse から thor への書き換えの際，第一に必要となるのは，~.gemspec ファイル内に Thor がインストールされるように変更を加えることである．

```
spec.add_development_dependency "thor"
```

この記述は，作成しているコマンドラインツールが Thor に依存性を持つことを意味している．optparse で書いた場合は，Thor に依存性を持つ必要がなく，当然このような記述はされていないので，上の一文を書き加える必要がある．また，bundle update をターミナル上で実行することで，ローカルでのコマンドラインツールを実行した際に Thor が適用されるようにアップデートされる．そうすることで，Thor の記述が実際にソースファイルで使用できるようになる．

3.2 thor によるサブコマンド

次に，コマンドが呼ばれる流れについて説明する．

1. コマンドを実行する．
2. コマンドを実行すると，exe ディレクトリの中にあるコマンド名と同じ名前のファイル（以降コマンドファイルと呼ぶ）が実行される．
3. コマンドファイル内で lib ディレクトリ内のソースファイルを require しておき，クラス内のコマンドを解析する関数を呼び出す．

4. ソースファイル内に書かれた処理が実行される．

このような流れでコマンドが呼び出され，処理が行われる．Thor と optpase の差異は 4. におけるコマンドの解析による処理関数への中継の方法である．

3.3 exe ディレクトリの書き換え

exe ディレクトリの中にあるコマンドファイルについて書き換えを行う．まずは optparse を使用した際のコマンドファイルの記述を記載する．

```
#!/usr/bin/env ruby
require "my_help_opt"

MyHelp::Command.run(ARGV)
```

optparse の場合，MyHelp::Command.run(ARGV) というクラス関数を実行することでコマンドを呼び出している．しかし, Thor は optparse のように run(ARGV) を用いず, start(ARGV) というクラス関数を実行してコマンドを呼び出す．よって，run を start に書き換える作業が必要になる．以下に Thor を使用した際のコマンドファイルの記述を記載する．

```
#!/usr/bin/env ruby
require "my_help_thor"

MyHelp::Command.start(ARGV)
```

3.4 lib ディレクトリ

まず，self.run についてであるが，Thor では使用されないのでこの一文は削除する．次に initialize であるが，こちらについては変更が必要である．そもそも initialize とは my_help を動かすのに必要なディレクトリがあるかどうかを調べるメソッドであり，なければここで作ることができる．optparse の initialize では argv=[] となっているところを Thor では（アスタリスク）argv とする必要がある．また，大きな違いは super の有無である．optparse では super は必要ないのだが，Thor では必要になる．これは，

3.5 その他オプション

optparse ではコマンド実行の際，引数の解析を行い，その引数に合わせた関数を呼び出す，という手順で動作している．その手順を記述した関数が execute である．

```
def execute
      @argv << '--help' if @argv.size==0
      command_parser = OptionParser.new do |opt|
```

```
        opt.on('-v', '--version','show program Version.') { |v|
          opt.version = MyHelp::VERSION
          puts opt.ver
        }

        opt.on('-l', '--list', 'list specific helps'){list_helps}
        opt.on('-e NAME', '--edit NAME', 'edit NAME help(eg test_help)'){|file| edit_help(
        opt.on('-i NAME', '--init NAME', 'initialize NAME help(eg test_help).'){|file| ini
        opt.on('-m', '--make', 'make executables for all helps.'){make_help}
        opt.on('-c', '--clean', 'clean up exe dir.'){clean_exe}
        opt.on('--install_local','install local after edit helps'){install_local}
        opt.on('--delete NAME','delete NAME help'){|file| delete_help(file)}
      end

      begin
        command_parser.parse!(@argv)
      rescue=> eval
        p eval
      end
      exit
end
```

しかし Thor の場合，execute のような間を取り持つ関数を用意する必要がなく，関数自体をコマンドとして登録していく形をとっているので，この関数は不必要である．以下に Thor での関数宣言を記載する．optparseno 場合，-i というコマンドで呼び出されている処理を init というコマンドで呼び出されるように書き換えたのが以下の関数である．

```
desc 'init NAME, --init NAME', 'initialize NAME help(eg test_help).'
# 関数についての説明，ここがヘルプで表示される．
map "--init" => "init"
# --オプションでも呼び出せるようにしてある．
def init(file)# 上にずらずらと書いているが，実際にオプション名として参照しているのは
関数名らしい．
#以下は変更なし
    p target_help=File.join(@local_help_dir,file)
    if File::exists?(target_help)
        puts "File exists. rm it first to initialize it."
        exit
    end
    p template = File.join(@default_help_dir,'template_help')
```

```
    FileUtils::Verbose.cp(template,target_help)
end
```

このように書き換えることで，Thor を使用したオプションの設定を行うことができるのだが，実際に動かしてみるとエラーが表示されることがある．ここで表示されるエラーは，コマンドとして設定されていない関数があることについての警告文である．その関数はコマンドとして使用しないということを明確に記述することでこの警告を消すことが可能である．

```
no_commands do

    #この間にコマンド設定しない関数を記述

end
```

4 比較

ここで optparse 版と thor 版の比較をする．

5 総括