卒業論文

my_help の Thor によるインターフェース改善

関西学院大学理工学部

情報科学科 西谷研究室

27014534 大八木　利治

2018 年 3 月

目 次

表 目 次

第1章　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．

第2章　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．

メモや todo リストの作成が行えることや，保存場所を共通化することでどこでも立ち上げることができることなど，emacs の org-mode と類似している点がいくつか存在する．しかし，明確な相違点も存在する．org-mode は emacs を起動させなければならないが，my_help は emacs を起動させる必要がなく terminal で編集することが可能である．また，org-mode を使用するとなると org-mode 独自のコマンドを学ぶ必要があり，学習コストがかかってしまう．my_help にはその必要がなく，非常に単純な操作でアプリを使用することができるので，org-mode の使い方を理解していない初心者にとって使いやすいものとなっている．

また，アプリやプログラミング言語などの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可能である．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．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 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．

2.2.1 optionの種類

optionの種類については以下のように述べられている．

> 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[1]pp.14-5.

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 [1]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．それがある時，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 [1]p.16.

2.2.2 Command Line Interface

Graphical User Interface（GUI）はコンピュータグラフィックスとポインティングデバイスを用いて操作を行う方法である．それに対し Command Line Interface（CLI）は Character User Interface（CUI）とも呼ばれ，キーボードからの入力と文字による情報の表示だけを用いて操作を行う方法である．

”Build awesome command-line application in ruby 2”の中で，

> グラフィカルユーザーインターフェース (GUI) はいろんな点で素晴らしいものです． 冷たくまたたくカーソルの殺伐とした輝きよりも， GUI はとりわけ初心者にとても優しいものです． でも，それには犠牲が伴います．GUI の熟練者になるには，奥義のようなキーボードショートカットを学ぶ必要があります． そうだとしても，あなたは生産性と効率の限界にぶち当たります． GUI はスクリプトして自動化しにくいことで悪名高いし，それができたとしても，あなたのスクリプトは移植しにくい傾向にあります [1]pp.3-4.

と述べられており，GUI の限界について示唆している．また，

> あなたのアプリをインストールした後，それを用いたユーザーが最初に体験するのは実際のコマンドラインインターフェイスでしょう．もしそのインターフェイスが難しく，直感的でなく，もしくは，とても醜いならば，たくさんの自信を呼び起こすことはできないし，ユーザーはそれを使って明確で簡潔な目的を達成するのに苦労するでしょう．逆に，使いやすければ，あなたのインターフェイスはアプリケーションに観衆と鋭さを与えてくれるでしょう [1]pp.3-4.

とも述べられている．

第3章　研究内容

3.1　optparseとThorの比較

3.1.1　optparse

今回の研究対象のmy_helpは，optparseで実装されている．optparseはrubyの標準ライブラリであり，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[2]．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 program Version.') { |v|
      opt.version = MyHelp::VERSION
      puts opt.ver
    }
    opt.on('-l', '--list', 'list specific helps'){list_helps}
    #中略
  end
  #中略
end

def list_helps
#中略
end
#後略
```

上記のコードはoptparse版my_helpの一部である．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 [3]．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．

3.1.2 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 [4]．Thorには以下のような特徴がある．

- 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 [5].
- 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 [5].
- 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．下記はThor版my_helpの一部である．

```
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
```

表3.1にoptparseとThorの比較をまとめた．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可能である．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，コマンドを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．

表 3.1: optparse と Thor の比較.

	コマンドの記法	特徴	使用頻度
optparse	マイナスを付した省略記法	Rubyの標準ライブラリであり，古くから用いられている	低い
Thor	任意の文字列	サブコマンドを含むコマンドラインツールの作成が容易	高い

3.1.3 比較

書き換えの前に，単純な CLI を例に optparse と Thor の比較を行った．今回はコマンドに続けて name を入力すると，Hello name と出力する CLI を optparse と Thor，それぞれを用いて作成し，コード量や使用感を比較する． 1 つめのソースコードが optparse を用いたもの（optparse.rb），２つめのソースコードが Thor を用いたもの（thor.rb）である．

```
options = {:name => nil}

parser = OptionParser.new do|opts|
  opts.on('-n', '--name name', 'Give your own name') do |name|
    options[:name] = name;
    end

  opts.on('-h', '--help', 'Displays Help') do
    puts opts
    exit
    end
end

parser.parse!

sayHello = 'Hello ' + options[:name]

puts sayHello
```

```
require 'thor'

class MyCLI < Thor
  desc "hello NAME", "say hello to NAME"
  def hello(name)
    puts "Hello #{name}"
  end
end

MyCLI.start(ARGV)
```

optparse.rb は，on メソッドで，入力された名前を受け取る-n というコマンドの定義を行い，parser.parse!以降で Hello name と出力するように記述を行った．また，-h のコマン

ドでヘルプを表示できるようにコマンドを登録した．本来ならば，-n ではなく-h（hello の頭文字）で登録したかったのだが，ヘルプを表示するコマンドが-h であり，重複してしまうためやむを得ず-n とした．一方 thor.rb では，Thor を継承したクラスの中で name の受け取りと Hello name の表示を行うように hello コマンドを設定した．optparse と違い，Thor では自動でヘルプが作成されるため，help コマンドは作成していない．optparse.rb はサブコマンドを書かずに実行すると，エラーになってしまうが，thor.rb ではコマンド一覧を表示してくれる．

コード量は，help を自動で生成する分，thor.rb の方が短くなっており，desc の中で一つのコマンドが完結しているということもあって optparse.rb よりも書きやすく感じられた．また，実際にコマンドを入力する際，thor.rb の方が自然言語に近いコマンドのため直感的に入力することができた．さらに，thor.rb で hello の後に name（引数）を入力せずに実行すると hello コマンドの usage が表示されるなど，使用感は thor.rb の方がよく感じられた．

3.2 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．

3.3 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．

3.4 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)
```

3.5　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．

3.6　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(file)}
        opt.on('-i NAME', '--init NAME',
         'initialize NAME help(eg test_help).'){|file| init_help(file)}
        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すことが可能．
def init(file)
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
```

3.7　変更後のオプション

オプションを省略記法から自然言語に近い記法に変更したことでより直感的なコマンド入力が可能になると思われる．以下に Thor で書き換えた後の my_help のオプション一覧を示す．

```
Commands:
my\_help clean, -\/-clean clean up exe dir.
my\_help delete NAME, -\/-delete NAME delete NAME help
my\_help edit NAME, -\/-edit NAME edit NAME help(eg test\_help)
my\_help help {[}COMMAND{]} Describe available commands or one specific
command
my\_help init NAME, -\/-init NAME initialize NAME help(eg test\_help).
my\_help install\_local, -\/-install\_local install local after edit
helps
my\_help list, -\/-list list specific helps
my\_help make, -\/-make make executables for all helps.
my\_help version, -v show program version
```

第4章　総括

今回の研究では，ユーザメモソフトである my_help の Thor による書き換えを行った．この書き換えによる成果は以下の通りである．

1. オプションの記法を-（ハイフン）を用いた省略記法からフルワードを用いた自然言語に近い記法に変更した．これにより，初心者でも直感的にオプションを使用することができ，学習コストの削減，学習効率の向上に繋がると考えられる．

2. Thor を用いることによって optparse を用いた時よりもソースコードを短くすることに成功した．

謝辞

本研究を行うにあたって，終始多大なるご指導，ご鞭撻を頂いた関西学院大学理工学部情報科学科西谷滋人教授に対し，深く御礼申し上げます．また，本研究の進行に伴い，様々な助力，協力をして頂きました西谷研究室の先輩方，同輩達に心から感謝いたします．西谷研究室の益々のご発展，ご多幸を心よりお祈り申し上げます．

参考文献

[1] David Bryant Copeland, *Build Awesome Command-Line Applications in Ruby 2*, (Pragmatic Bookshelf, 2013).

[2] library optparse, erikhuda, https://docs.ruby-lang.org/ja/latest/library/optparse.html, accessed 2018.2.9.

[3] コマンドライン引数によるオプションに対応する (optparse), http://maku77.github.io/ruby/io/optparse.html, accessed 2018.2.1.

[4] S.Koichiro, Thor の使い方まとめ, http://qiita.com/succi0303/items/32560103190436c9435b, accessed 2018.2.9.

[5] Hibariya, Thor で簡単にコマンドラインアプリをつくる, http://note.hibariya.org/articles/20111025/a0.html, accessed 2018.2.9.