卒業論文

コマンドラインツール作成ライブラリ Thor による hikiutils の書き換え

関西学院大学　理工学部　情報科学科

27013554 山根亮太

2017 年 3 月

指導教員　　西谷　滋人　教授

目次

1　概要

研究室内の内部文書，あるいは外部への宣伝資料，さらに wikipedia のように重要な研究成果の発信などに西谷研では hiki system を利用しています．これは初心者にも覚えやすい直感的な操作であるが，慣れてくるとテキスト編集や画面更新にいちいち web 画面へ移行せねばならず，編集の思考が停止する．そこで，編集操作が CUI で完結させるためにテキスト編集に優れた editor との連携や，terminal 上の shell command と連携しやすい hikiutils が開発された．しかし，そのユーザーインターフェースにはコマンドが直感的でないという問題がある．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変更することでコマンドを実装し直し直感的なコマンドにすることを目的とした．optparse で作成されている hikiutils を thor で作成し，そして 2 つのコマンドラインツール作成ライブラリで作成された hikiutils を比較する．研究結果は，thor のほうがコマンドを簡単に定義することができ，またコードも短くできた．

2 序論

hiki は，hiki 記法を用いた wiki clone です．wiki はウォード・カニンガムが作った wikiwikiweb を源流とする home page 制作を容易にするシステムで，hiki も wiki の基本要求仕様を満足するシステムを提供しています．wiki の特徴である web 上で編集する機能を提供しています．これを便宜上 hiki web system と呼びます．図にある通り，一般的な表示画面の他に，編集画面が提供されており，ユーザーはこの編集画面からコンテンツを編集することが可能です．リンクやヘッダー，リスト，引用，表，図の表示などの基本テキストフォーマットが用意されています．

hiki web system の実際の基本動作は，hiki.cgi プログラムを介して行われています．こちらを便宜上 hiki system と呼びます．図に従って hiki system の動作概要を説明します．hiki system は，data/text に置かれた書かれたプレーンテキストを html へ変換します．この変換は hikidoc[1-1] という hiki フォーマット converter を使っています．また，添付書類は cache/attach に，一度フォーマットした html は parser に置かれており，それらを参照して html を表示する画面を hiki.cgi は作っています．さらに hiki system では検索機能，自動リンク作成などが提供されています

図 1　hiki web system と hiki system の対応関係.

研究室内の内部文書，あるいは外部への宣伝資料，さらに wikipedia のように重要な研究成果の発信などに西谷研ではこの hiki system を利用しています．初心者にも覚え

やすい直感的な操作です．しかし，慣れてくるとテキスト編集や画面更新にいちいち web 画面へ移行せねばならず，編集の思考が停止します．そこで，テキスト編集に優れた editor との連携や，terminal 上の shell command と連携しやすいように hikiutils という cli(command line interface) を作成して運用しています．しかし，そのユーザインタフェースにはコマンドが直感的でないという問題点がある．そこで，Thor というコマンドラインツール作成ライブラリを用いる．hikiutils では，optparse というコマンドライン解析ライブラリを使用しているが，新たなライブラリ Thor を使用してコマンドを書き換え，より直感的なコマンドに変更する．

3　方法

3.1 optparse と thor の比較

ここでは，FizzBuzz という単純なコードを例に optparse と CLI のコードの比較を行う．

3.1.1 optparse

optparse とは，getopt よりも簡便で，柔軟性に富み，かつ強力なコマンドライン解析ライブラリである．optparse では，より宣言的なスタイルのコマンドライン解析手法，すなわち OptionParser のインスタンスでコマンドラインを解析するという手法をとっている．これを使うと，GNU/POSIX 構文でオプションを指定できるだけでなく，使用法やヘルプメッセージの生成も行える [1-3]．利用頻度はあまり高くないが古くから開発され，使用例が広く紹介されている．

optparse の基本的な流れとしては

1. OptionParser オブジェクト opt を生成する
2. オプションを取り扱うブロックを opt.on に登録する
3. opt.parse(ARGV) でコマンドラインを実際に parse する

である．

OptionParser はコマンドラインのオプション取り扱うためのクラスであるためオブジェクト opt を生成され opt.on にコマンドを登録することができる．しかし，OptionParser#on にはコマンドが登録されているだけであるため，OptionParser#parse が呼ばれた時，コマンドラインにオプションが指定されていれば実行される．optparse にはデフォルトとして–help と–version オプションを認識する [1-4].

3.1.2 Thor

Thor とは，コマンドラインツールの作成を支援するライブラリのことである．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 [1-2].

Thor の基本的な流れとしては

1. Thor を継承したクラスのパブリックメソッドがコマンドになる
2. クラス.start(ARGV) でコマンドラインの処理をスタートする

である [1-2].

start に渡す引数が空の場合，Thor はクラスのヘルプリストを出力する．また，Thor はサブコマンドやサブサブコマンドも容易に作ることができる．

3.2 既存の hikiutils のコマンド解説

既存の hikiutils はコマンド解析ライブラリの optparse を用いて，コマンドの処理を行っている．optparse の特徴は，「コマンドの登録，実行 method」に分けて記述することが期待されている．また，CLI の起動の仕方が特徴的である．この二つを取り出して，動作とコードを説明する．

3.2.1 コマンドの登録と実行メソッド

図 2　コマンドの登録と実行メソッドの対応

optparse では以下の通り，コマンドの登録と実行が行われる．

1. OptionParser オブジェクト opt を生成
2. opt にコマンドを登録
3. 入力されたコマンドの処理のメソッドへ移動

optparse では OptionParser オブジェクト opt の生成を行い，コマンドを opt に登録することでコマンドを作成することができる．しかし，これはコマンドを登録しているだけでコマンドの一覧ではこれを表示することができるが，コマンドの実行を行うためには実行を行うためのメソッドを作成する必要がある．optparse でのコマンドの実行は opt で登録されたコマンドが入力されることでそれぞれのコマンドの処理を行うメソッドに移動

し処理を行う．しかし，このコマンド登録はハイフンを付けたコマンドしか登録ができず，ハイフンなしのコマンド登録はまた別の手段でやらなくてはいけない．

```
1     def execute
2       @argv << '--help' if @argv.size==0
3       command_parser = OptionParser.new do |opt|
4         opt.on('-v', '--version','show␣program␣Version.') {
              |v|
5           opt.version = HikiUtils::VERSION
6           puts opt.ver
7         }
8         opt.on('-s', '--show','show␣sources') {show_sources}
9         opt.on('-a', '--add','add␣sources␣info') {
              add_sources }
10        opt.on('-t', '--target␣VAL','set␣target␣id') {|val|
              set_target(val)}
11        opt.on('-e', '--edit␣FILE','open␣file') {|file|
              edit_file(file) }省
              略
12
13        ......
14
15      end
16      begin
17        command_parser.parse!(@argv)
18      rescue=> eval
19        p eval
20      end
21      dump_sources
22      exit
23    end
24
25    def show_sources()
26      printf("target_no:%i\n",@src[:target])
27      printf("editor_command:%s\n",@src[:editor_command])省略
28
29      ......
30
31    end以下略
```

3.2.2 CLI の実行プロセス

optparse を用いた場合の CLI の実行プロセスは次の通りとなる．

1. Hiki の HikiUtils::Command.run(ARGV) で hikiutils.rb の run メソッドを呼ぶ
2. new(argv).execute で execute メソッドが実行される

図 3　CLI の実行プロセス.

optparse では Hikiutils::Command.run(ARGV) を実行される．require で呼び出された hikiutils.rb で run メソッドが実行される．そこでコマンドを登録している execute メソッドへ移動し入力したコマンドと対応させる．そして，対応したコマンドの処理が行われるメソッドに移動することで実行される．このように optparse では実行を行うためのメソッドが必要であるが,

3.2.3　コード

- optparse

```
1 #!/usr/bin/env ruby
2
3 require "hikiutils"
4
5 HikiUtils::Command.run(ARGV)
```

```
1     def self.run(argv=[])
2       print "hikiutils: provide utilities for helping hiki
            editing.\n"
3       new(argv).execute
4     end
5
6     def execute
7       @argv << '--help' if @argv.size==0
8       command_parser = OptionParser.new do |opt|
```

```
9           opt.on('-v', '--version','show program Version.') {
                |v|
10            opt.version = HikiUtils::VERSION
11            puts opt.ver
12          }
13          opt.on('-s', '--show','show sources') {show_sources}
14          opt.on('-a', '--add','add sources info') {
                add_sources }
15          opt.on('-t', '--target VAL','set target id') {|val|
                set_target(val) }
16          opt.on('-e', '--edit FILE','open file') {|file|
                edit_file(file) }
17          opt.on('-l', '--list [FILE]','list files') {|file|
                list_files(file) }
18          opt.on('-u', '--update FILE','update file') {|file|
                update_file(file) }
19          opt.on('-r', '--rsync','rsync files') {rsync_files}
20          opt.on('--database FILE','read database file') {|
                file| db_file(file)}
21          opt.on('--display FILE','display converted hikifile'
                ) {|file| display(f\
22  ile)}
23          opt.on('-c', '--checkdb','check database file') {
                check_db}
24          opt.on('--remove FILE','remove file') {|file|
                remove_file(file)}
25          opt.on('--move FILES','move file1,file2',Array) {|
                files| move_file(file\
26  s)}
27          opt.on('--euc FILE','translate file to euc') {|file|
                 euc_file(file) }
28          opt.on('--initialize','initialize source directory')
                 {dir_init() }
29        end
30        begin
31          command_parser.parse!(@argv)
32        rescue=> eval
33          p eval
34        end
35        dump_sources
36        exit
37      end
```

4　結果

4.1 コマンドの命名原則

機能ごとの動作はコマンドのオプションによって指定されます．このオプションにどのような名前をつけるかは，どれだけコマンドを覚えやすいかという意味で重要です．コマンドの振る舞いを的確に表す名称をつける必要があります．

この振る舞いとしてもっとも受け入れやすいのが shell で用意されているコマンドです．pwd, ls, rm, touch, open などはもっとも直感的に動作がわかるコマンドです．hikiutils の振る舞いを予測できるシェルコマンドと同じ名前でオプションを提供するようにします．

4.1.1 hikiutils の想定利用形態

ここで hikiutils があらかじめ想定している利用形態を解説しておきます．

図 4　hikiutils があらかじめ想定している利用形態．

hikiutils は，

- local PC と global server とが用意されており，
- それらのデータを rsync で同期する

ことで動作することを想定しています．これは，ネットに繋がっていないオフラインの状況でもテキストなどの編集が可能で，さらに不用意な書き換えを防ぐための機構です．さ

らに，どちらもが何かあった時のバックアップともなって，ミスによる手戻りを防いでいます．

これらの設定は， /.hikirc に yaml 形式で記述・保存されています．

```
bob% cat ~/.hikirc
:srcs:
- :nick_name: new_ist
  :local_dir: "/Users/bob/Sites/new_ist_data/ist_data"
  :local_uri: http://localhost/ist
  :global_dir: nishitani@ist.ksc.kwansei.ac.jp:/home/
      nishitani/new_ist_data/ist_data
  :global_uri: http://ist.ksc.kwansei.ac.jp/~nishitani/
- :nick_name: dmz0
  :local_dir: "/Users/bob/Sites/nishitani0/Internal/data"
  :local_uri: http://localhost/~bob/nishitani0/Internal
  :global_dir: bob@dmz0:/Users/bob/Sites/nishitani0/
      Internal/data
  :global_uri: http://nishitani0.kwansei.ac.jp/~bob/
      nishitani0/Internal
```

また，一般的に一人のユーザがいくつものまとまりとしての local-global ペアを保持して管理することが普通です．それぞれに nicke_name をつけて管理しています．

```
bob% hiki -s
hikiutils: provide utilities for helping hiki editing.
"open -a mi"
target_no:1
editor_command:open -a mi
 id | name       | local directory
                                        | global uri
----------------------------------------------------------------------------------------------------

  0 | new_ist    | /Users/bob/Sites/new_ist_data/ist_data
                | http://ist.ksc.k
 *1 | dmz0       | /Users/bob/Sites/nishitani0/Internal/
    data       | http://nishitani
  2 | ist        | /Users/bob/Sites/hiki-data/data
                   | http://ist.ksc.k
  3 | new_maple | /Users/bob/Sites/new_ist_data/
```

maple_hiki_data | http://ist.ksc.k

とすると，それらの一覧と，いま target にしている nick_name ディレクリが表示されます.

4.1.2 コメンド名と振る舞いの詳細

検討の結果コマンドを以下のように書き換えることとします．上部に記した，特によく使うコマンドに関しては，shell でよく使われるコマンド名と一致するようにしました.

表 1

変更前	変更後	動作の解説
edit FILE	open	open file
list [FILE]	ls	list files
rsync	rsync	rsync files
update FILE	touch	update file
show	pwd	show nick_names
target VAL	cd	target を変える，change directory とのメタファ
move [FILE]	mv	move file
remove [FILE]	rm	remove files
add		add sources info
checkdb		check database file
datebase FILE		read datebase file
display FILE		display converted hikifile
euc FILE		translate file to euc
help [COMMAND]		Describe available commands or one specific command
version		show program version

それぞれの意図を動作の解説として記述しています.

■open FILE　ファイルを編集のために editor で open．Editor は /.hikirc に

```
:editor_command: open -a mi
```

として保存されている．open -a mi を emacs などに適宜変更して使用．

■ls [FILE]　local_dir にあるファイル名を [FILE*] として表示．例えば，hikiutils_yamane 以下の拡張子がついたファイルを表示．hiki システムでは text ディレクトリーは階層構造を取ることができない．西谷研では directory の代わりにスネーク表記で階層構造を表している．

■rsync　local_dir の内容を global_dir に rsync する．逆方向は同期に誤差が生じたり，permission がおかしくなるので，現在のところ一方向の同期のみとしている．したがって，作業手順としてはテキストの変更は local_dir で読み行うようにしている．

■touch FILE　loccal_dir で書き換えた FILE の内容を local_uri に反映させ，ブラウザで表示．シェルコマンドの touch によって，変更時間を現在に変え，最新状態とするのに似せてコマンド名を touch としている．

■pwd　nick_name の一覧と target を表示，current target を current dir とみなして，コマンド名を pwd とした．

■cd VAL　target を変える，change directory とのメタファ．ただし，いまのところ nick_name では対応しておらず，nick_name の番号を VAL 入力することで変更する．

4.2 Thor による実装

手法のところで概観した通り，thor を用いることで記述の簡略化が期待できる．ここでは，実際に書き換える前後，すなわち optparse 版と thor 版の対応するコードを比較することで，以下の具体的な違い

- クラス初期化
- コマンド定義
- CLI の実行プロセス

について詳しく検討を行う．

4.2.1 クラス初期化

図 5　Thor の initialize でのコード

Thor の initialize でのコードはつぎの通りである．

1. Hikithor::CLI.start(ARGV) が呼ばれる
2. initialize メソッドが呼ばれる
3. これでは Thor の initialize メソッドが呼ばれない

4. super を書くことで Thor の initialize メソッドが呼ばれる

optparse では require で optparse を呼び optparse の initialize を定義する必要はないが，Thor は initialize を定義する必要がある．Thor の定義方法は require で Thor を呼び CLI クラスで継承し，initialize メソッドに super を書くことで Thor の initialize が呼ばれる．initialize メソッド内では Thor の初期設定がされていないため，スーパークラスのメソッドを読み出してくれる super を書き加えることで図のように initialize メソッド内で Thor の initilalize メソッドが呼ばれ定義される．

```
1
2  module Hikithor
3
4    DATA_FILE=File.join(ENV['HOME'],'.hikirc')
5    attr_accessor :src, :target, :editor_command, :browser, :
         data_name, :l_dir
6
7    class CLI < Thor
8     def initialize(*args)
9         super
10        @data_name=['nick_name','local_dir','local_uri','
              global_dir','global_uri']
11        data_path = File.join(ENV['HOME'], '.hikirc')
12        DataFiles.prepare(data_path)以下略
13
14        ......
15    end
```

4.2.2 コマンド定義

thor では optparse のような登録処理はない．図にある通りにコマンドが記述され，それらは以下のように構成される．

1. desc 以降にコマンド名と，その説明が記述される．これらはコマンド help で一覧として表示させる
2. map によって別のコマンド名でも実行できるように定義される．
3. def で定義されたメソッドの実行コード

Thor では desc で一覧を表示されるコマンド名，コマンドの説明を登録する．しかし，ここで記述したコマンドは単に一覧で表示させるためのものであり，実際に実行される時に呼び出すコマンド名は，def で定義された名前である．Thor では処理実行を行うメソッド名がコマンド名となり，コマンド名 1 つが対応する．

図 6　thor におけるコマンド記述のひな形.

これに別名を与えるために利用されるキーワードが map である.

```
map A => B
```

map とは B と呼ばれるメソッドを A でも呼べるようにしてくれるものである．よって，これを使うことでコマンドの別名を指定することができる.

```
1     desc 'show,--show', 'show␣sources'
2     map "--show" => "show"
3     def show
4       printf("target_no:%i\n",@src[:target])
5       printf("editor_command:%s\n",@src[:editor_command])以下
            略
6       ,,,...
7     end
```

以上より，Thor ではコマンドの指定と処理には desc,map, 処理メソッドだけで済む．optparse ではコマンドを登録するためのメソッドと処理メソッドの両方が必要になっていた．一方 Thor では，処理メソッドが直接コマンド名となるため記述が簡潔になる.

4.2.3　CLI の実行プロセス

Thor における cli の実行プロセスは次の通りである.

1. hiki_thor の Hikithor::CLI.start(ARGV) で hikiutils_thor.rb の CLI クラスを呼ぶ

図 7　CLI の実行プロセス.

2. hikiutils_thor.rb の CLI クラスのメソッドを順に実行していく

Thor では start(ARGV) を呼び出すことで CLI を開始する．Hikithor::CLI.start(ARGV) を実行されることにより require で呼ばれている hikiutils_thor.rb の CLI コマンドを順に実行する．そして，入力されたコマンドと一致するメソッドを探し，そのコマンドの処理が実行される.

```
1 #!/usr/bin/env ruby
2
3 require "hikiutils_thor"
4
5 Hikithor::CLI.start(ARGV)
```

```
module Hikithor

  DATA_FILE=File.join(ENV['HOME'],'.hikirc')
  attr_accessor :src, :target, :editor_command, :browser, :data_name, :l_dir

  class CLI < Thor
   def initialize(*args)
      super
```

```
@data_name=['nick_name','local_dir','local_uri','global_dir','global_uri
data_path = File.join(ENV['HOME'], '.hikirc')
DataFiles.prepare(data_path)
... 以下略...
```

Thor ではクラスのメソッドを順に実行していくため run メソッドと execute メソッドは必要ない．また，optparse での実行手順はメソッドの移動回数が多く複雑であるが，Thor は単純で分かりやすいものとなっている．

4.2.4 optparse との全体的な比較

コードからも Thor のほうが短くなっていることが分かる．よって，Thor と optparse でのコードの違いは以上の部分になるが全体的にも Thor のほうがコードが短くなり，コマンドの定義も簡単に行うことができる．また，実行手順も分かりやすくコードが読みやすいため書き換えもすぐ行うことができるので，より直感的なコマンドを実装することも可能となった．

5　参考文献

[1-1] hikidoc, https://rubygems.org/gems/hikidoc/versions/0.1.0, https://github.com/hi
アクセス.

[1-2]「Thor の使い方まとめ」, http://qiita.com/succi0303/items/32560103190436c9435b
,2017/1/30 アクセス.

[1-3]「15.5. optparse - コマンドラインオプション解析器」, http://docs.python.jp/2/library/opt
,2017/1/30 アクセス.

[1-4]「library optparse」, https://docs.ruby-lang.org/ja/latest/library/optparse.html
,2017/1/30 アクセス.