7-1 デバッグ手法、テスト手法

ユーザインタフェースに関わるデバッグはどのプログラミング言語においても難しい課題です。本節ではVimスクリプトにおけるデバッグ手法を説明していきます。

:echomsg デバッグ

Vimスクリプトのデバッグの基本は**:echomsg**デバッグ、C言語でいうところのprintfデバッグと言っても過言ではないでしょう。**:echo**も同様の機能を持ちますが、最大の違いは**:echomsg**で出力した内容はVimのログに残るところにあります。

:echomsgの使い方は簡単です。ちょうど、Vimのコンソール出力のような使い方ができます。

```
echomsg "hello, world!"
```

コマンドの実行後にログを参照するには、**:messages**コマンドを使用します。あまりに長すぎるログは途中で切れてしまうことに注意してください。大量のログが発生する場合はログ機構を自作するかファイルに保存すると良いでしょう。

1つ注意事項として**:echomsg**は**:echo**とは異なり、引数に必ず文字列を渡さなければなりません。**:echo**は引数が文字列ではない場合には自動的に文字列に変換しますが、**:echomsg**は引数を変換しないので文字列を以外を渡した場合はエラーとなります。

これに慣れないうちによく犯してしまう間違いが、次のような**:echomsg**の使い方です。

```
" エラー！
echomsg [1, 2, 3]
```

このような場合には、**:echomsg**の引数を組み込み関数**string()**に通しましょう。**string()**は引数を文字列に変換するため、エラーにならなくなります。

```
" OK
echomsg string([1, 2, 3])
```

vim-prettyprint プラグインを用いたデバッグ

:echomsgは非常にシンプルで使いやすいのですが、シンプル過ぎて表示が見苦しくなる場合があります。特にリストや辞書をふんだんに利用した複雑なデータ構造を表示する場合

は、読み解くのが大変です。この欠点を解決するものの1つとして、ここでは**vim-prettyprint**プラグインを取り上げます。

https://github.com/thinca/vim-prettyprint

```
:PP [1, 2, 3]
[1, 2, 3]
```

デフォルトの**:PP**コマンドは**:echo**相当です。**:echomsg**とは違ってログには記録されません。ログに記録したい場合、**:PP!**コマンドを利用しましょう。

```
:PP! [1, 2, 3]
[1, 2, 3]
```

vim-prettyprintが真価を発揮するのは、例えば以下のような複雑なデータ構造の表示です。

```
let a = {}
let a.aaaaaaaaaaaaaa = 'foooooooooooooooooooooooooooooooooooooooo'
let a.bbbbbbbbbbbbbbb = 'barrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr'
let a.ccccccccccccccc = 'bazzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz'
let a.ddddddddddddddd = [1, 2, 3]
```

これを**:echomsg**で出力した場合、画面幅により折り返されますが、改行を含まない直線的なデータとして表示されてしまい、お世辞にも読みやすいとは言えません。

```
:echomsg string(a)
{'aaaaaaaaaaaaaa': 'foooooooooooooooooooooooooooooooooooooooo', 'bbbbbbbbbbbbbbb': 'barrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr', ...(略)
```

一方vim-prettyprintの出力結果と比べてみると、改行やインデントを駆使し、可読性に優れたものになっているのがわかるでしょう。

```
:PP! a
{
  'aaaaaaaaaaaaaa': 'oooooooooooooooooooooooooooooooooooooooo',
  'bbbbbbbbbbbbbbb': 'barrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr',
  'ccccccccccccccc': 'bazzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz',
  'ddddddddddddddd': [1, 2, 3]
}
```

さらにvim-prettyprintは、辞書関数の表示にも対応しています。

```
let f = {}
function! f.func()
  echo 'hello, world!'
  echo 'foo, bar.'
endfunction
verbose PP! f
```

このスクリプトを実行すると次のように表示されます。

```
{
  'func':
   function 634() dict
1    echo 'hello, world!'
2    echo 'foo, bar.'
   endfunction
}
```

辞書関数の内容に加えて、行番号や辞書関数自身の番号が表示されます。これらが役立つのはVimのスタックトレースの解読の場面です。

辞書関数を大量に使っている場合、エラー時に表示される関数呼び出しのバックトレースは、以下のようにエラーが発生した場所がどこなのかまったく追えません。

```
Foo..<SNR>90_bar..22..33..634
```

しかし、vim-prettyprintが表示してくれるこれらの数字を用いれば、デバッグの手間は格段に軽減されます。

:echomsg、vim-prettyprintのどちらをデバッグに用いる場合も、デバッグメッセージの消し忘れに注意してください。コミットやpushする前にdiffをよく確認すると良いでしょう。

Vimのデバッガ機能を用いたデバッグ

Vimには簡易なデバッガ機能も搭載されています。

このデバッガ機能を始めるには 明示的にデバッグモードでコマンドを実行するか、ブレークポイントをセットする必要があります。

明示的にデバッグモードでコマンドを実効するには、**:debug**コマンドを使います。

書式

```
:debug {command}
```

このようにするとExコマンド**{command}**を実行する前にデバッグモードに入ります。

ブレークポイントはVimスクリプト上の場所を指定し、スクリプト実行中にその場所へ到達した際に自動的にデバッグモードに入る機能です。

ブレークポイントを指定・設置するには、**:breakadd**コマンドを使います。

書式

```
:breakadd func [lnum] {name}
```

この例では**{name}**で指定した関数の**[lnum]**行目にブレ

ークポイントを設置しています。

書式

```
:breakadd file [lnum] {name}
```

こちらの例では**{name}**で指定したファイルの**[lnum]**行目にブレークポイントを設置します。

:breakadd hereを用いると、Vimのカレントバッファの現在行に対してブレークポイントを設置できます。このコマンドはスクリプト編集しながらデバッグする際に便利でしょう。

:debugコマンドを使ってもブレークポイントを使っても、デバッグモードに入るとVimスクリプトの実行は停止しされ、任意のVimのコマンドを実行して変数の値などを検査することができます。

```
:echo g:some_variable
0
```

さらに、デバッグモードでは、表7.1に示すデバッグ用コマンドを使用できます。一度実行したデバッグ用コマンドは**<CR>**で繰り返すことができます。

表7.1 デバッグ用コマンド

デバッグ用コマンド	説明
cont または c	次のブレークポイントまで実行する
next または n	次のコマンドを実行し、デバッグモードに戻る。ユーザ関数やソースファイルの読み込み時にはその中には入らずスキップする
step または s	コマンドを実行しそれが終わるとデバッグモードに戻る。ユーザ関数やソースファイルの呼び出し時には呼び出し先の行に進む
finish または f	実行中のスクリプトやユーザ関数を終了し呼び出し元でデバッグモードに戻る
quit	異常停止するが、次のブレークポイントで停止する
interrupt	例外割り込みについて:finallyや:catchをテストするのに使用する

より詳しい解説は**:help debug-scripts**を参照してください。

Vim を最小構成で起動する

Vimスクリプトの動作はホストとなるVimの設定に左右されるため、極めて環境依存の強い言語と言えます。そのため、仮に不具合の報告を受けたとしても報告者の設定があなたのものとは異なる場合、再現ができず原因がわからず解決も困難となる場合があります。

そこで環境になるべく依存させないよう、Vimを最小構成で起動するコマンドはぜひ覚えておきましょう。下記がVimを最小構成で起動するコマンドです。

```
vim -N -u NONE -U NONE -i NONE --noplugin
```

Vimスクリプトをデバッグする場合、最小構成にテスト用の.vimrcを加えた構成で行うのが普通ですので、.vimrcを指定する方法もマスターしておきましょう。

```
vim -N -u ~/test.vimrc -U NONE -i NONE --noplugin
```

-uオプションの引数に読み込む.vimrcへのパスを記述します。

よくある最小構成の.vimrcは、例えば次のようなものとなります。

```
filetype off

" プラグインのパスをruntimepathに追加
set rtp+=~/foo/bar

filetype plugin indent on

" その他必要な設定があれば
```

起動オプション**-u**で.vimrcを指定すると**nocompatible**オプションは自動的にはセットされなくなります。

しかし、Vimの起動時に**-N**オプションを指定しているので、**set nocompatible**を.vimrcに書く必要はありません。

7-2 スクリプトに関わるオプションの紹介

VimにはVimスクリプトの動作に影響を与えるオプションがいくつかあります。Vimスクリプトを書くときに、これらのオプションのことを考慮していないと、開発環境では動くのに実際のユーザの環境では動かないスクリプトができあがってしまいます。本節ではそのようなオプションと対処方法の紹介をします。

'ignorecase' オプション

'ignorecase'オプションは、Vimにおける文字列比較全般と広範に影響を与えるオプションです。特に理由がない限りは、このオプションの影響を受けない適切な比較演算子を用いるようにしてください。