Emacs 解剖学

Lecture 5

バッファローカル変数(その1)

井田昌之, 榎並嗣智

はじめに

「Emacs を特徴づける概念は，どんなものなのでしょうか？」

さまざまな角度からこのことに取り組んで紹介しています．コマンドの体系やキーなどに現われる操作的な側面が，まず考えられます．また，プログラミング言語による機能拡張の概念が，次に考えられます．これらについて，TECO，キーボード，Gosling Emacs，Multics Emacs などの話を通して，その一部を紹介してきました．

今回からは，「エディタ向きの Lisp の仕様」という側面で，Emacs らしさを，その発端も含めて紹介します．

今回は，最近では GNU Emacs 19.31 上での mule-2.3 の移植などをやっている榎並嗣智さんに，バッファローカル変数をめぐる GNU Emacs の扱いについての解説をお願いしました．以下の記事は榎並さんによるものです．なお，そのなかでは，Emacs ないし Emacs Lisp という言葉が出てきますが，それらは，すべて GNU Emacs および GNU Emacs での Emacs Lisp を指しています．

（いだ まさゆき　青山学院大学 国際政治経済学部）

変数と，その値

Emacs Lisp におけるバッファローカル変数について，今回と次回の 2 回に分けて考えてみましょう．今回はまず，普通の変数が Emacs Lisp ではどうなっているのか，また，変数と密接な関係のある値がどうなっているのかを把握しておきます．なお，文中実際に Emacs および gdb を使った例が出てきますが，これらは NetBSD/i 386 1.2_BETA 上の Emacs 19.31 をベースとした Mule 2.3 および，NetBSD についてくる gdb 4.11 を使用しました．

gdb は，Free Software Foundation が開発配布している，主として UNIX 上で動くシンボリックデバッガです．Emacs 同様，多くのプラットフォームで動きます．現在の最新版は gdb 4.16 です．文中の例で使っているコマンドについて簡単にまとめておきます（**表**）．

最初に，Lisp の世界でいう値とはどういうものかを，C と対比させて考えてみます．

たとえば C において，

```
int i;
```

という文があった場合，これは i という整数型（int 型）変数を定義します．そして i は，値として整数だけを保持することができます．このように，C の場合，変数には必ず型があり，保持される値もその型に制限されます．また，変数のサイズも，その型に応じて変わります．int が 4 バイトで表現され，double が

表

行頭の(gdb)	gdb からのデフォルトのプロンプト.
p	Print の省略形で，与えられた C の式の値を表示する．/x などの修飾子を与えて，16 進で表示させることなどができる.
x	examine．メモリの内容を表示する．print 同様 /x などの修飾子が使える．/s は 0 terminate された文字列の表示.
run	デバッグされるべき process を起動する.
c	breakpoint その他で中断している process を再開(continue) する.
handle	シグナルの処理方法を指定．handle 2 nostop noprint pass とすることで，SIGINT をそのまま Emacs に渡すようにする．^G のたびに Emacs から gdb に戻らないようにするため.
ptype	引数として与えられたものの定義を表示.

*また，$nn などとして，過去のコマンドの結果を参照できる．$ や $$ で直前あるいはその一つ前の結果が参照できる.

8バイトで表現されている C の処理系の場合，そもそもサイズが違うので，どうやっても int 型変数に double 型の値をもたせることはできません.

一方，Lisp の場合，変数自身には，普通，型はありません．最近は型のある（正確に言えば型の制限された）変数もあります．それらはすべて組込みの型です．たとえば，Emacs 19.31 などでは変数 fill-column に整数以外の値をもたせることはできません.

```
(setq fill-column 'abc)
```

などとすると，“only integers should be stored in the buffer-local variable fill-column” というエラーになります.

一般に，Lisp の変数は，Lisp の世界のもの（Lisp Object）であれば何であれ保持することができます．このことを効率良く実現するために，Lisp では Lisp Object 実体へのポインタを介してそれを操作します．変数も，Lisp Object そのものではなく，Lisp Object へのポインタを保持するようになっています．ポインタのサイズは普通は一定なので，どんな型の値でももてるわけです.

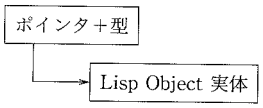
また，ポインタを使うことにより，シンボルの同一性などの表現も簡単に実現できています.

しかし，Lisp においても，整数は整数ですし，リストはリストです．実は，Lisp では，型は変数ではなく値そのものにつけられているのです．そしてそのために，integerp（整数か？）や Listp（リストか？）といった，型を尋ねる述語があるのです.

Lisp Object の実現方法

次に，Emacs Lisp において Lisp Object がどう実現されているのか，型がどう値にもたされているのか，を見てみましょう.

Emacs Lisp では，型は，タグとして付属されています．つまり，Emacs Lisp の Lisp Object は，実体へのポインタ部，さらにそこに含まれる型を表わすタグ部からなっています.



具体的には，

```
((Lisp Object 実体へのポインタ)
 &((1 << VALBITS) - 1))
 |(型 << VALBITS)
```

という式で計算した結果が Lisp_Object という C の型として扱われます．VALBITS はシステムあるいはコンフィグレーションに固有の定数で，以前の Emacs では通常 24 ビット，最近の Emacs では 28 ビットになっています．したがって，ある Lisp_Object obj の型を調べるには，

```
(obj >> VALBITS)
```

とし，Lisp Object 実体へのポインタを得るには

```
(obj & ((1 << VALBITS) - 1)
```

とします（システムによっては，さらに特定のビットを on にしたりする必要がありますが）．また，整数は実体を伴わず，値が直接ポインタ部に格納されます．つまり，123 という整数は

```
(123 & ((1 << VALBITS) - 1))
 | (Lisp_Int << VALBITS)
```

となります．Emacs Lisp における，扱える整数の大きさの制限はここからきています．整数という型を表わす定数 Lisp_Int には実は 0 が使われているので，123 という正の整数は，Lisp_Object として表現される場合も 123 という同じビットパターンになります（負の整数の場合は同じではありません．Lisp_Obejct から取り出すときに符号ビットを復元する必要があり

ます．EmacsのCのコードで使われるFASTINTというマクロがあるのですが，これを負の数の可能性のある場所に使ってしまうのは，Emacs用のCコードを書いた人の多くが一度はする間違いかもしれません）．

この，型を表わすタグ部と実体へのポインタ部によりLisp Objectを表現する方法の利点としては，型を調べるだけならメモリ参照がいらないこと，整数など一部の型を実体を使わずに表現できる（整数はポインタ部に値そのものが入っています）ことなどがあります．欠点としては，実際にメモリ参照する際には実際のポインタを作る手間が必要であること，Lisp Objectの実体をスタックに置けないことなどがあります．また，Emacsのバッファサイズの制限などもここから来ています．

なお，最近のEmacsでは，ポインタ部を広く使うために，一部の型（たとえばmarkerなど）において型情報が実体側に移されています．この変更によって初めて，すべてのプラットフォームにおいて整数が28ビットになったのです．また，将来扱える型を増やす際の制限が緩くなりました．

以上が，Emacs Lispにおいて変数の値となるべきLisp Objectの実現方法です．ということは，Emacs Lispで変数といえば，それはLisp Objectを保持できるものであるはずですね．

では Emacs Lisp の変数とは？

実は，Emacs Lispに限らず，Lispでは一般にシンボルというLisp Objectを変数として使います．シンボルそのものも名前をもったLisp Objectですが，その名前と値に関連をもたせ，参照できるような仕組みが提供されています．そのおかげでシンボルが変数として使えるのです（ただし，すべてのシンボルが変数として使えるわけではありません．たとえばtやnilは定数です）．

その変数と値の対応のことをbinding（束縛）と呼び，変数に値をもたせることを「変数をbindする」と言います．つまり，変数はbindされることによって新しい値をもちます．たとえば set/setq，また，letやcondition-caseや，関数引数などでも新たな束縛は作られます．letは，指定された変数について新たな束縛を作り，その環境のもとで与えらえたformを実行し，そこから抜けるときに元の束縛に戻します．

この束縛を管理するための方法として，たとえば以下のような方法があります．

- deep binding：変数たるシンボルとその値との対を，束縛が作られる度にスタックに積んでいきます．変数の値を参照するには，そのスタックを上から順に調べ，見つかった対の値をその値とします．
- shallow binding：変数にはそれぞれ値を格納する場所があります．ある変数の値は，そこを参照すればわかります．それが有効かどうかは別に用意した手段によって判断します．新たな束縛を作る際には，古い値をスタックに退避し，新たな値をその場所に格納します．

Emacs Lispは，後者の方法を使っています．それを具体的に確認してみましょう．

シンボルの構造を覗く

Emacs Lispでは，シンボルは内部では次のようなCの構造体として実現されています．

```
struct Lisp_Symbol
  {
    struct Lisp_String *name;
    Lisp_Object value;
    Lisp_Object function;
    Lisp_Object plist;
    struct Lisp_Symbol *next;
  };
```

このうちvalueというスロットが，シンボルが変数として値を保持するための場所です．処理系によっては，このvalueスロットは存在せず，代わりにplistの中に変数としての値をもっている処理系もあるそうです．

シンボルが変数として値をもっている，ということを，実際にgdbというシンボリックデバッガを使って確認してみましょう．以下の例（図）では，(gdb) がgdbからのプロンプトです．また，pは

```
enami@pavlov% gdb ./emacs
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.11 (i386-netbsd), Copyright 1993 Free Software Foundation, Inc...
Environment variable "DISPLAY" not defined.
TERM = kterm
Function "abort" not defined.
(gdb) handle 2 nostop noprint pass
SIGINT is used by the debugger.
Are you sure you want to change it ? (y or n) y
Signal          Stop    Print   Pass to program Description
SIGINT(2)       No      No      Yes             Interrupt
(gdb) run
```

図1　gdb の元での Emacs の起動

gdb コマンドの print の省略形で，与えられた C の式の値を表示します．

まず図1のように，コマンドラインから gdb を起動し，さらにそのなかから Emacs を立ち上げます．次にバッファ *scratch* で変数 hoge に整数 123 を代入してみます（図2）．記号 => で始まる行は，Emacs からの出力を意図しています（実際のバッファには記号 ⇒ は現われません）．

これで hoge という変数の値が 123 になったはずです．Emacs を suspend して gdb に戻ってみましょう（図3）．

current_buffer という C のグローバル変数は，現在カレントバッファとなっている buffer 構造体へのポインタです．バッファ *scratch* の内容が，current_buffer->own_text.beg の中身として見えています．hoge というシンボルへのポインタを得るには，たとえば

```
p intern("hoge")
```

とします．Emacs の内部関数 intern は，hoge という名前のシンボルを探してきて，Lisp_Object としての値を返してくれます．今の場合，270336384 あるいは 0x101d0180 というのが Lisp_Object，つまり型の埋め込まれたポインタです．これから，実際に型とポインタを抜き出してみましょう．型は Lisp_Symbol で，ポインタの値は 0x1d0180 だということがわかります．

実は，Lisp_Object から型やポインタを取り出すためのマクロが，Emacs の配布ディレクトリの下，

```
(setq hoge 123)
=> 123
hoge
=> 123
```

図2　変数 hoge に整数 123 を代入する

src/.gdbinit の中に定義されています（これは Emacs をコンパイルしたディレクトリから gdb を起動すると自動的に読み込まれて使えるようになります．あるいは，後から source コマンドで読み込むこともできます．コンフィグレーションによって異なる定数などは，enum を使って実行ファイルのデバッグ情報の中に埋め込まれていたりもします）．型を取り出すマクロは xtype，struct Lisp_Symbol へのポインタを取り出すマクロは xsymbol で，たとえば図4のように使います．また，Emacs が process として動いていれば，pr というマクロで pretty print することもできます

xsymbol の出力の1行目は，$9 にはシンボルの実体へのポインタが格納されていることを示し，2行目はシンボル名前が hoge であることを示しています．シンボルの実体の中身を見てみましょう（図5）．先にも説明したように整数の型は 0 なので，整数 123 を保持してるはずのシンボル hoge の value スロットは，確かに 123 になっています．

変数の値を変えてみる

次に，変数の値を変えてみます．いったん Emacs

```
Program received signal SIGTSTP (18), Suspended
0x1011e7ab in kill ()
(gdb) p current_buffer                  # 変数 current_buffer を表示
$1 = (struct buffer *) 0x149c00
(gdb) p *current_buffer                 # 変数 current_buffer の中身を表示
$2 = {
  size = 537002060,
  next = 0x149e00,
  own_text = {
    beg = 0x1f8800 " (setq hoge 123) \n123\nhoge\n123\n",
    gpt = 30,
        :
       (中略)
        :
  extra2 = 269745156,
  extra3 = 269745156,
  mc_flag = 0
}
(gdb) p intern ("hoge")                 # 名前が hoge であるシンボル
$3 = 270336384                          # を Lisp_Object として得る
(gdb) p/x intern ("hoge")
$4 = 0x101d0180
(gdb) p ($4 >> 28)
$5 = 1
(gdb) p (enum Lisp_Type)  ($4 >> 28)
$6 = Lisp_Symbol
(gdb) p/x ($4 & ((1 << 28) - 1))    # 下位 28 ビット を取り出す
                                    # /x は 16 進で表示するため
$7 = 0x1d0180
```

図3　通常の変数の内部構造

```
(gdb) p intern ("hoge")
$8 = 270336384
(gdb) xtype                             # xtype は直前の値 ($) に対して
Lisp_Symbol                             # 実行される．$ は変更されない
0
(gdb) xsymbol                           # xsymbol も同じ．$ が変更される
$9 = (struct Lisp_Symbol *) 0x1d0180
0x1d6cd4 "hoge"
(gdb) p intern ("hoge")
$10 = 270336384
(gdb0) pr                               # pr も $ に対して実行される
hoge
```

図4　macro xtype/xsymbol と pr

を resume し（図6），hoge の値を t にしてみましょう（図7）．

再度 Emacs を suspend して gdb に戻り，シンボルの実体を表示し直してみましょう（図8）．value スロットの値が変わっていますね．これはシンボル t のはずですが，それを確認するため，value の値を見てみます．確かに t のようです．ちょっとシンボル t 自身の値も見てみましょう．これは変数 hoge の現在の value スロットの値と同じ，ということは t ですね．t は自分自身を値としてもっていたのです．これは

```
(gdb) p *$9                                  # 変数 hoge に対応する struct
$11 = {                                      # Lisp_Symbol * の中身.
  name = 0x1d6ccc,
  value = 123,
  function = 269745176,
  plist = 269745156,
  next = 0x190684
}
```

図5　シンボルの実体

```
(gdb) c
Continuing.
```

図6　gdb から Emacs に戻る

```
(setq hoge t)
=> t
```

図7　変数 hoge の値を t にする

```
Program received signal SIGTSTP (18), Suspended
0x1011e7ab in kill ()
(gdb) p *$9                                  # 以前とは value スロットの値が違う
$12 = {
  name = 0x1d6ccc,
  value = 269745196,
  function = 269745176,
  plist = 269745156,
  next = 0x190684
}
(gdb) p $9->value                            # 変数 hoge の value スロットには…
$13 = 269745196
(gdb) xtype                                  # シンボルが入っていた
Lisp_Symbol
0
(gdb) xsymbol                                # そのシンボルは…
$14 = (struct Lisp_Symbol *) 0x13fc2c
0xb46c0 "t"                                  # t という名前だった
(gdb) p $14->value                           # t の value スロットは t 自身
$15 = 269745196
```

図8　変更後の変数の値を調べる

nil についても同じです.

ついでに，シンボルの名前も見てみましょうか. 図9のような手順でシンボル hoge の名前が最終的には hoge という C の文字列に行き着くことが確認できます.

struct Lisp_String の data というスロットは，unsigned char data [1]; という定義なので，普通に表示させただけでは最初の1文字しか表示されません. 実際には，(sizeof (struct Lisp_String)＋実際の data の長さ) 分の大きさの領域が確保され，data の後ろに2文字目以降が続いているのです. さらに data は0で terminate されているので x/s として文字列としてメモリダンプしてやれば内容を確認することができます. あるいは .gdbinit のマクロ xstring のように，陽に長さを指定して表示させることもできます.

ここまでで，Emacs Lisp では Lisp の変数としてシンボルを使い，個々のシンボルに value というスロットを設けて値をもたせているのがわかったと思います. ここでちょっと寄り道をして void な変数というものについて見てみます.

```
(gdb) p $9->name
$16 = (struct Lisp_String *) 0x1d6ccc
(gdb) p *$16                              # name の中身
$17 = {
  size = 4,
  intervals = 0x0,
  data = "h"
}
(gdb) ptype                               # struct Lisp_String の定義は ?
type = struct Lisp_String {
    int size;
    struct interval *intervals;
    unsigned char data[1];
}
(gdb) x/s $16->data                       # data を文字列として dump する
0x1d6cd4 <end+650076>:   "hoge"
(gdb) c
Continuing.
```

図9　シンボルの名前の構造

```
(makunbound 'hoge)
=> hoge
hoge
=> error ! Symbol's value as variable is void: hoge
```

図10　変数 hoge を void にする

Symbol's value as variable is void!

"Symbol's value as variable is void" というエラーメッセージは，GNU Emacs を使っている人なら一度は見たことがあると思いますが，これは bind（束縛）されてない変数の値を求めようとした，というエラーです．bind されていない変数は値をもっていません．したがって，そのような変数の値を求めようとすることはエラーになります．

ある変数が bind されていないということと，その変数の値が nil であるということは別のことです．前者はその変数の値は存在しませんが，後者では nil という値が存在しています．変数の値が存在しないことを，その変数は void であるといいます．このことを gdb を使って確認してみましょう．

先ほど使った変数 hoge はすでに値をもっています．そのような変数を再び void にするには，関数 makunbound を使います．図10のようにして hoge の値を確認しようとすると，エラーになると思います．さて，このとき，シンボル hoge の value スロットには何が入っているでしょうか？　先ほどと同じように，gdb を使って見てみます（図11）．

おや？　シンボル hoge の value スロットには unbound という名前のシンボルが入っているようですね．ということは Emacs Lisp では unbound というシンボルは，変数の値として使えないのでしょうか？ Emacs を resume して，変数 hoge の値を unbound というシンボルにしてみましょう．

```
(setq hoge 'unbound)
```

と入力します．値を確認するために

```
hoge
```

とすると，今度はエラーにはならずにシンボル unbound を返してきます．

もう一度 gdb に戻って，シンボル hoge の value スロットを確認してみましょう（図12）．確かに，unbound というシンボルが値となっていますが，よく見ると void な変数の value スロットにあるシンボル unbound は，Lisp_Object としては 269745176 であるのに対し，今のシンボル unbound は 270336444 です．

```
Program received signal SIGTSTP (18), Suspended
0x1011e7ab in kill ()
(gdb) p intern ("hoge")
$18 = 270336384
(gdb) xsymbol
$19 = (struct Lisp_Symbol *) 0x1d0180
0x1d6cd4 "hoge"
(gdb) p $19->value                    # void な変数の value スロットは…
$20 = 269745176
(gdb) xtype                           # シンボルだった !
Lisp_Symbol
0
(gdb) xsymbol                         # 名前は unbound
$21 = (struct Lisp_Symbol *) 0x13fc18
0xb46b0 "unbound"
(gdb) c
Continuing.
```

図 11　void な変数の場合

```
Program received signal SIGTSTP (18), Suspended
0x1011e7ab in kill ()
(gdb) p intern ("hoge")               # 変数 hoge にシンボル unbound を
$22 = 270336384                       # もたせた場合には…
(gdb) xsymbol
$23 = (struct Lisp_Symbol *) 0x1d0180
0x1d6cd4 "hoge"
(gdb) p $23->value                    # void なときとは value スロットの
$24 = 270336444                       # 値が違う
(gdb) xsymbol
$25 = (struct Lisp_Symbol *) 0x1d01bc
0x1d6df4 "unbound"
(gdb) c
Continuing.
```

図 12　unbound というシンボルを値にもつ変数の場合

つまり，名前は同じなのですが実体としては別のシンボルだったのです．Lisp 的な言い方をすると，後者は obarray に属すシンボルであるが，前者は前もって作られた，どこにも属してないシンボルなのです．前もって作られたものであるのでユーザが後から作ったものと同じになることはありません．また通常の手段でユーザがアクセスすることもできません（もちろん，C のレベルでそのための関数を書けばできますが）．したがって，void であることを示すために十分有効に使えるわけです．

以上，バッファローカル変数について見る前に，通常の変数とその値について見てきました．次回はいよいよバッファローカル変数の仕組みにせまってみたいと思います．

（えなみ つぐとも）