Emacs 解剖学

Lecture 6

バッファローカル変数(その2)

井田昌之, 榎並嗣智

はじめに

前回では，GNU Emacs での基礎的な変数のつくられ方を，意味について見てきました．今回は，いよいよバッファローカル変数について説明します．それにはまず，その発端の紹介をしなければなりません．Multics Emacs がその発端になります．そこから始めて，前回同様に後半は榎並さんにバトンタッチします．

変数の「有効範囲」

プログラミング言語には，変数の「有効範囲」ということがあります．それには，その変数が物理的に存在している期間（エクステント），そしてその変数が字面上，参照できる区間（スコープ）という二つの性質があります．別の場所で同じ名前の変数が記述されているときに，それらは別のものなのか同一のものなのか，あるいは，一方が他方をシャドウする（隠す）のか，というような議論がそこで出てきます．

伝統的な Lisp では，一般に，

「軌跡の中で，最後に与えられている値が，その変数の値だ．たとえその値がまったく異なるプログラムモジュールで定義されていても．」

という認識が根底にありました．この上に，いろいろなスコーピングルールなどがそれぞれの Lisp によって設定されていくわけです．

このことに関連して Emacs の場合，最も大きなテーマは何かということになってきます．それは，エディタとしての性質を，ベースとなるプログラミング言語の中でどう扱いやすくするかというテーマにもなります．

per-buffer 変数

エディタとしての Emacs の大きな特徴は，まず何と言ってもマルチバッファのエディタであるということです．つまり，Emacs では，バッファ（エディタの作業域）を複数もち，それらを同時に開いて並行して処理を進めることができます．これは，現在ではそれほど珍しくない便利な機能ですが，それを支える裏方の仕組み，しかもプログラミング言語としての仕組みを考えるときには，多少複雑化する要素が出てきます．

たとえば，あるプログラムを同時に複数のバッファで実行させているとします．そのプログラムで扱う変数は，ある種，グローバルにして，同時に動いている複数のコピーがある場合にでも共通して参照させたいものもあるでしょうし，それぞれのバッファごとに独立させたいものも出てくるでしょう．さらに面倒なことには，これらのバッファ間の切替えは，操作者が気ままに勝手にやるので，プログラムの字面として前もって定めるような形で各変数の有効範囲の制御をすることは難しいものになります．

そうすると，自然のなりゆきとして，「このバッフ

ァだけで有効な変数」というものを作れるようにしておいて，それをそれぞれのバッファで独立に管理すればいいというようになってきます．

per-buffer 変数（バッファごとの変数）というのが，Multics Emacs での，こうした変数に対するネーミングでした．

Multics Emacs の中での普通の変数は，その記述言語である Multics MacLisp のルールに従って管理されます．基本的には，起動された Emacs の中で有効なすべての変数は，それが異なる独立したプログラムであっても，同一の，ただ一つの，名前空間の中に置かれます．per-buffer 変数と宣言されたものは，バッファごとに独立した存在をもちます．それらの場合，同じ変数名が別のバッファで動作している別のプログラム中に現われても，その影響を受けません．

前回紹介した Bernie Greenberg は，per-buffer 変数の概念を 1978 年に Multics Emacs に持ち込んだと言っています．その頃，Richard Stallman は，TECO をベースにした ITS Emacs をやっていて，GNU Emacs はまだ誕生していませんでしたし，記述言語として Lisp も採用していませんでした．

per-buffer 変数については，1980 年に出版されたハネウエル社のマニュアル[1]に次のように記述されています．

Multics Emacs での per-buffer 変数の定義

「Emacs のバッファスイッチャは，バッファが切り替えられるときにグローバル（特殊）変数の値をセーブし，リストアする．ある変数を，そのバッファに入るとき，出るときにこの処理の対象とするには，そのことを知らせないといけない．そのような変数は，per-buffer 変数と呼ばれる．バッファスイッチャにそれを教え，現在の値とそのバッファとを関連づけることをその変数を登録する（registering）と言う．いったん，変数があるバッファに登録されるとそれを使用する関数は，そのバッファに入ったときはいつでも，その値はそのバッファでの最後の値であることを前提としてよい．per-buffer 変数は，別の言い方をすれば，local 変数である．local 変数の登録のために，次の二つのプリミティブを用意している．ただし，それらは，実際に値を処理するものではなく，ちゃんと動作できるような仕組みを処理するものである．

- register-local-variable：これは，1 引数で呼び出される．それは，登録したい local 変数の名前のシンボルである．たとえば，

  ```
  (register-local-variable 'foo)
  ```

  のようにする．もしその名前が以前に登録されていなければ，カレントバッファに対して新たに登録される．その初期値は「グローバルな値」である．登録されても，その値はそのままに残される．もし，グローバル値がなければ（unbound であれば），そして，そのバッファへの最初の登録であれば，値として nil がセットされる．
- establish-local-variable：これは，register-local-variable と同様だが，2 引数をとる．第 2 引数は，グローバル値をもたない変数が，そのバッファに最初に登録されるときに，変数に与える初期値である．

per-buffer 変数のグローバル値は，登録されていないバッファではその変数の値となるものである．登録されていないバッファで値のセットをすると，それはそのグローバル値に対してなされる．local 変数は，それがあるバッファに最初に登録されるときにグローバル値を受け継ぐ．」

（文献 1）pp. 3-16 より）

この仕様を満たすために，バッファごとに per-buffer 変数を管理しているリストがあります．バッファスイッチャが呼び出されると，そのリストが調べられ，その変数の value セルの値は per-buffer リストを使って退避/復旧されます（普通のプッシュダウンスタックやリストは使われません）．

per-buffer からバッファローカルへ

per-buffer 変数が，我々が今使っている GNU Emacs でのバッファローカル（buffer-local）変数へと進んでいくのです．そして，基本的にこの二つは同じものだと言ってかまいません．したがって「（Save-excursion などだけでなく）バッファローカルの概念

も私が考え，それをリチャード（ストールマン）がGUNへ取り込んだんだ。」というBernieの主張は，およそ正しいものだと言えます．

さて，GNU Emacsでのバッファローカル変数はどんな仕組みで，どんな位置づけになっているのでしょうか？

（いだ まさゆき　青山学院大学 国際政治経済学部）

バッファローカル変数の仕組み

前回はEmacs Lispにおける通常の変数とその値について見てきました．Emacs Lispにおいては，変数はシンボルであり，変数の値はその中のvalueスロットに保持されるということがわかりました．今回はいよいよGNU Emacsでのバッファローカル変数ですが，実はこれ，ただのシンボルなのです．つまり，普通の変数となんら変わりがありません．ではどこに区別があるのでしょうか？　それは，値に秘密があるのです．

このEmacs Lispに特有なバッファローカル変数というのは，特殊なLisp_Objectを使って実現されています．set/setqやsymbol-valueという関数は，それぞれ変数の値を変更したり，変数の値を参照する基本的な関数です．これらは，与えられた変数（シンボル）の値の型を調べ，もしバッファローカル変数を表わすLisp Objectであればそれを特別扱いします．実際の変数と値のbindingは，各バッファ構造体がもっています（Cの構造体の，local_var_alistというスロット）．関数buffer-local-variablesによって，そのバッファ構造体がもっているbindingを，連想リストとして得ることもできます．また，効率を上げるために，適宜キャッシュされています．それらの様子を実際に見てみましょう．

```
(setq hoge 123)
=> 123
(make-local-variable 'hoge)
=> hoge
hoge
=> 123
(setq-default hoge 'hoge-default)
=> hoge-default
(default-value 'hoge)
=> hoge-default
```

図1　変数をバッファローカルにする

make-local-variableで変数をバッファローカルにする

まず，前回作った変数hogeをバッファローカルにしてみます（図1）．これで，変数hogeはバッファ*scratch*ではローカルな値123をもち，デフォルトの値としてはシンボルhoge-defaultをもつようになりました．make-local-variableという関数は，与えられた変数をカレントバッファでローカルになるようにします．他のバッファでローカルであるかどうかということには影響を与えません．

ここでgdbに戻って，バッファローカルにした変数hogeの値には，何が入っているのかを見てみましょう（図2）．

valueスロットの値の型がLisp_Misc_Some_Buffer_Local_Valueになっています．このLisp_

```
Program received signal SIGTSTP (18), Suspended
0x1011e7ab in kill ()
(gdb) p intern ("hoge")                    # 今度は hoge はバッファローカル
$26 = 270336384
(gdb) xsymbol
$27 = (struct Lisp_Symbol *) 0x1d0180
0x1d6cd4 "hoge"
(gdb) p $27->value
$28 = 538816644
(gdb) xtype                                # value スロットには見なれないものが…
Lisp_Misc
Lisp_Misc_Some_Buffer_Local_Value
```

図2　バッファローカルな変数の値

Misc_Some_Buffer_Local_Value という型の Lisp Object の実体は,

(REALVALUE BUFFER CURRENT-ALIST-ELEMENT. DEFAULT-VALUE)

というリストによく似た構造をしています（実は BUFFER の部分から後ろは完全にリストそのものです）（図3）. それぞれの意味を説明しましょう. REALVALUE が, 変数の本当の値です. BUFFER は, 現在の REALVALUE がどのバッファでのものかを示しています. CURRENT-ALIST-ELEMENT というのは, バッファ構造体のバッファローカル変数とその値の ALIST の中の, 対応する対です. DEFAULT-VALUE は, デフォルト値です. これらが実際にどうなっているかを, gdb で見てみましょう（図4）.

図の $30->car が, 上の説明での REALVALUE にあたります. 確かにバッファ *scratch* での変数 hoge の値である 123 になっています. $30 自体は struct Lisp_Buffer_Local_Value という構造体ですが, その cdr から後は Lisp_Cons ですから, これ以降はまさにリストなわけです. この $30->cdr 部以降は通常の cons によるリストなので pr で見てみます.

先の 123 という値がバッファ *scratch* のものであることがわかります. 次の CURRENT-ALIST-ELEMENT が, バッファ構造体の中の対応する対である, つまり current_buffer->local_var_alist の一部であることは gdb で追っていけば容易に確かめられますが, 長くなるのでここでは省略します. DEFAULT-VALUE には hoge-default というシンボルが確かに格納されているようですね. バッファロ

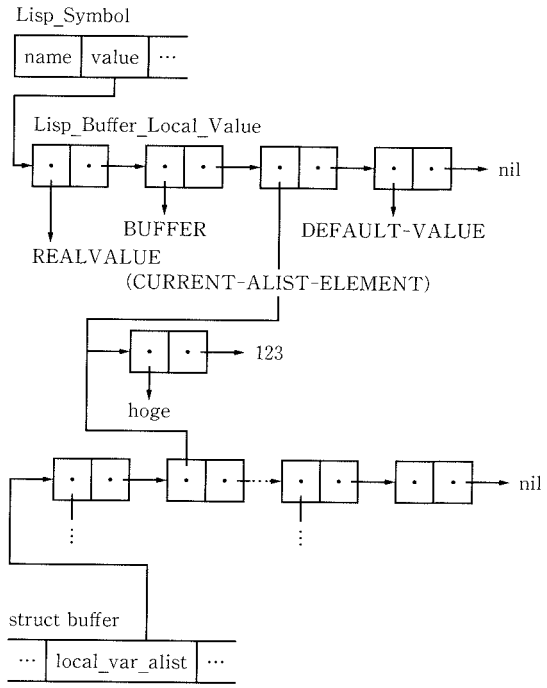

図3 バッファローカル変数の構造1
（バッファローカルな値をもつ場合）

```
(gdb) xbuflocal
$29 = (struct Lisp_Buffer_Local_Value *) 0x1db084
(gdb) p *$29                                  # バッファローカルな変数の value
$30 = {                                       # スロットには cons のようなものが…
  type = 24242,
  spacer = 0,
  car = 123,                                  # これが REALVALUE
  cdr = 1343816260
}
(gdb) p $30->cdr
$31 = 1343816260
(gdb) pr                                      # cdr から後はまさにリストだ
(#<buffer *scratch*> (hoge . 123) . hoge-default)
(gdb) p $27->value
$32 = 538816644
(gdb) pr                                      # hoge はバッファ *scratch* でローカル
#<some_buffer_local_value [realvalue] 123[buffer] #<buffer *scratch*>
[alist-elt] (hoge . 123) [default-value] hoge-default>
(gdb) c
Continuing.
```

図4 バッファローカルな変数の値の実体

ーカル変数自体も pr でも見ることができます．

バッファローカルな変数の値を変更してみる

ここで変数の値を変更してみましょう．Emacs に戻って，図5のように操作してから gdb に戻り，図6のように変数 hoge の value スロットを見てみると，REALVALUE だけが新しい値に変更されています．CURRENT-ALIST-ELEMENT がいつ変更されるのかは後述します．とりあえずは Emacs に戻って，別のバッファに行ってみましょう．

```
C-x 4 b piyo RET
M-x lisp-interaction-mode RET
```

```
(setq hoge 'hoge-hoge)
=> hoge-hoge
hoge
=> hoge-hoge
```

図5　バッファローカルな変数の値を変更する

としてバッファ piyo に移ってモードを *scratch* と同じモードにします．ここで変数 hoge の値を参照してみると，シンボル hoge-default つまりデフォルト値が返ってきます．このバッファでは変数 hoge はローカルではないからです．ここで gdb に戻り，value スロットを確認してみましょう（図7）．

いったん Emacs に戻って値を変えてみましょう．図8のように，バッファ piyo での変数 hoge の値をシンボル hoge-piyo にしてみます．その上で図9のようにしてみると，やはり REALVALUE スロットの値だけが変更されています．

このあたりの細かい話は次でしますが，とりあえず

```
(setq hoge 'hoge-piyo)
=> hoge-piyo
hoge
=> hoge-piyo
```

図8　バッファローカルでないバッファで変数の値を変更

```
Program received signal SIGTSTP (18), Suspended
0x1011e7ab in kill ()
(gdb) p $27->value
$33 = 538816644
(gdb) pr                              # 新しい hoge の値が REALVALUE のところに．
#<some_buffer_local_value [realvalue] hoge-hoge[buffer] #<buffer *scratch*
>
[alist-elt] (hoge . 123) [default-value] hoge-default>
(gdb) c
Continuing.
```

図6　バッファローカル変数の値の変更

```
Program received signal SIGTSTP (18), Suspended
0x1011e7ab in kill ()
(gdb) p intern ("hoge")
$35 = 270336384
(gdb) xsymbol
$36 = (struct Lisp_Symbol *) 0x1d0180
0x1d6cd4 "hoge"
(gdb) p $36->value
$37 = 538816644
(gdb) pr                              # バッファローカルでないので REALVALUE
                                      # は DEFAULT-VALUE と同じ．alist-elt の
                                      # #1 は循環リストを表わす．
#<some_buffer_local_value [realvalue] hoge-default[buffer] #<buffer piyo>
[alist-elt] (#1 . hoge-default) [default-value] hoge-default>
(gdb) c
Continuing.
```

図7　別のバッファでのバッファローカル変数の値

```
Program received signal SIGTSTP (18), Suspended
0x1011e7ab in kill ()
(gdb) p $36->value
$38 = 538816644
(gdb) pr                                    # やはり REALVALUE だけが変わっている
#<some_buffer_local_value [realvalue] hoge-piyo[buffer] #<buffer piyo>
[alist-elt] (#1 . hoge-default) [default-value] hoge-default>
(gdb) c
Continuing.
```

図9　バッファローカルでない変数の変更後の値

バッファローカル変数の値が特別な Lisp Object（およびバッファ構造体そのもの）に格納されていることがわかっていただけたでしょうか．

バッファローカル変数の更新

さて，先ほど make-local-variable をした後，あるいはカレントバッファを変更した後，いったん変数 hoge の値を参照してから gdb に戻りました．これには理由があります．

実はバッファローカル変数の内容は，単にカレントバッファを変更しただけでは変更されず，実際に参照されたときに初めて更新されるからです．先ほど述べた，CURRENT-ALIST-ELEMENT も，その際同時に更新されます．バッファローカル変数を参照するときの流れは次のとおりです．

1. カレントバッファが BUFFER と同じかどうか調べます．もし同じなら REALVALUE が求める値です．
2. そうでないなら，CURRENT-ALIST-ELEMENT の cdr 部に REALVALUE の値を退避します．今の時点では，CURRENT-ALIST-ELEMENT は BUFFER の local_var_alist の中の対応する部分を指していることに注意してください．
3. カレントバッファの local_var_alist から，対応する対を探します．もし見つかれば，それを新しい CURRENT-ALIST-ELEMENT とします．見つからなかった場合はカレントバッファではその変数はローカルではないのですが，この場合 CURRENT-ALIST-ELEMENT 自身を新しい CURRENT-ALIST-ELEMENT とします（図10）．厳密に書くと，CURRENT-ALIST-ELEMENT はある cons の car 部であり，DEFAULT はその cons の cdr 部なのですが，その cons の car 部をその cons 自身にする，ということです（循環リストになります）．これにより新しい CURRENT-ALIST-ELEMENT の cdr 部は実は DEFAULT ということになり，このため，2.のステップにおいてその変数が BUFFER ではローカルでなかった場合には，REALVALUE は DEFAULT の部分に退避されることになります．
4. カレントバッファを新しい BUFFER とし，さらに REALVALUE を更新します．

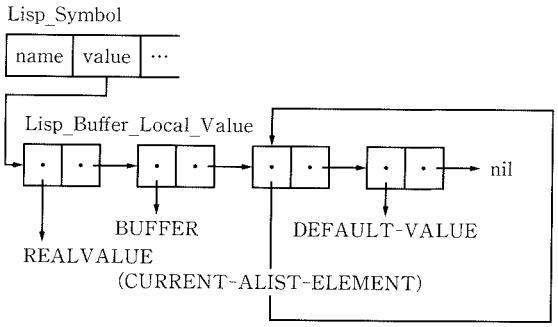

図10　バッファローカル変数の構造2
（バッファローカルな値をもたない場合）

たとえば先ほどのバッファ piyo の例では，変数 hoge はバッファローカルではないのですが，REALVALUE の値だけが変わっていることに注意してください．DEFAULT-VALUE の部分は，他のバッファで変数 hoge の値が参照されたとき，あるいはバッファ piyo でも変数 hoge がバッファローカルになったときに更新されます．この時点で，バッファ *scratch* に戻り，変数 hoge の値を参照してから gdb に戻ると，DEFAULT-VALUE の部分が更新されて

```
Program received signal SIGTSTP (18), Suspended
0x1011e7ab in kill ()
(gdb) p intern ("hoge")
$39 = 270336384
(gdb) xsymbol
$40 = (struct Lisp_Symbol *) 0x1d0180
0x1d6cd4 "hoge"
(gdb) p $40->value
$41 = 538816644
(gdb) pr                    # バッファ piyo での REALVALUE が
                            # DEFAULT-VALUE スロットに入っている
#<some_buffer_local_value [realvalue] hoge-hoge[buffer] #<buffer *scratch*
>
[alist-elt] (hoge . hoge-hoge) [default-value] hoge-piyo>
(gdb) c
Continuing.
```

図11 DEFAULT-VALUE の更新

いることがわかります（図11）.

以上が，バッファローカル変数を実現している仕組みの，大雑把な解説です．バッファごとの変数と値の束縛がいかに格納され，そして使われるかが，わかったかと思います．

次はバッファローカル変数に関連したいくつかの話題に触れてみます．

バッファローカル変数に関連した，その他のこと

make-local-variable と make-variable-buffer-local との違い

まず，make-local-variable によく似た make-variable-buffer-local について見てみましょう．

make-variable-buffer-local で作られるバッファローカル変数（より正確には，その変数の value スロットに格納される Lisp Object）と，make-local-variable によって作られるものとは，その型（Lisp_Misc_Buffer_Local_Value と Lisp_Misc_Some_Buffer_Local_Value）を除いてまったく同一です．つまり，キャッシュの仕組みやデフォルト値のもち方はまったく同一であり，単にその型に応じて，内部での扱いが多少違うだけです．たとえば，値を更新する際，その変数に対応する対をカレントバッファが local_var_alist にもっていなければ，どちらの場合も自動的に作られます．しかし，make-local-variable は即座にローカルな束縛を作りますが，make-variable-buffer-local では次に set されたときに初めてローカルな束縛が作られます．さらに，次節で述べるように kill-local-variable に対する挙動はずいぶん違います．

kill-local-variable は何をするのか

kill-local-variable という関数があります．これは，機能的に make-local-variable と対を成すと思っていいでしょう．この関数のドキュメントは

> Make VARIABLE no longer have a separate value in the current buffer.
> From now on the default value will apply in this buffer.

となっています．ここに書いてあるように，この関数はその変数が現在のカレントバッファでローカルな値をもたないようにします．つまりローカルな値，あるいはローカルな束縛がなくなるだけで，その変数がバッファローカル変数であるということには変わりがありません．したがって make-variable-buffer-local を使って，set/setq されれば常にローカルになるようにした変数は，依然としてそうあり続けます．つまり，次に setq した場合はその値は単にそのバッファでローカルな値になります．この意味では，この関数は kill-local-value あるいは kill-local-binding と名づけられたほうがよかったのかもしれません．make-local-variable もしかりです．

その make-local-variable を使って，あるバッファでのみローカルになるようにされた（あるいは，そのバッファでローカルな束縛が作られた）変数は，

kill-local-variable でローカルな束縛が解消されれば，それはバッファローカルでない変数と区別はつきません．

依然としてその変数の value スロットには型 SOME_BUFFER_LOCAL_VALUE である Lisp Object が格納されているのですが．

では，いったんバッファローカルにした変数をそうでなくすことはできるのでしょうか？ たとえば，バッファローカル変数として使われているシンボルを一度 unintern してから intern しなおし，すべての古いシンボルへの参照を新しいものへ置き換えることができれば可能でしょう．でもそれより，新しい C のプリミティブを作るほうが簡単でしょう．

hook 変数をバッファローカルにするには？

hook 変数というのは，hook として使われる変数のことで，それ以上の意味はありません．

したがって，make-local-variable などを使ってもバッファローカルにすることはできます．しかし，make-local-hook という専用の関数もあります．この関数を使うと，run-hooks が，バッファローカルな値だけでなくグローバルな値も hook として実行してくれる，という点が違います．このあたりは，両者の挙動を把握した上で使い分ければいいのではないかと思います．たとえば make-local-variable を使っておき，条件によってグローバルな hook は実行したりしなかったりさせることなども可能かと思います．

let とバッファローカル変数との関係

先ほど，バッファローカル変数というのは，シンボルの value スロットに特別な Lisp Object をもたせ，set/setq と symbol-value がそれを特別扱いすることによって実現されている，と書きました．

一方 let は，あるシンボルの古い値を symbol-value 使って参照し，その値をそのシンボルとともにスタックに積み，新しい値を set を使ってそのシンボルの値とします．束縛を解く際にも，やはり set を使って，退避してあった値を復元します．

つまり，バッファローカル変数の仕掛けは，set/setq と symbol-value の中に隠され，let からは見えません．これがどういうことかというと，つまり，もし let でバッファローカル変数が bind されていると，その中で実行される form がカレントバッファを変更したままですと，間違ったバッファでの値が復元されてしまうということです．これを防ぐためには，バッファローカル変数を bind するような let の中でカレントバッファを変更したら，save-excursion などで必ず元に戻るようにしておく必要があります．それでも，ユーザが勝手に，あるいは間違ってプログラマの予想しない変数をバッファローカルにしてしまったら…．このあたりは効率を取るか安全性を取るかの，設計のバランスなのでしょうか．

文献ガイド

1) Multics Emacs Extension Writers' Guide, Order # CJ 52, Honeywell Information Systems, January 1980.

（えなみ つぐとも）