スパルタンVim 6.0

MURAOKA Taro (KoRoN, @kaoriya)

プロトタイプ(2017/04/09)

モデルを知るべき理由

Vimを使い始めた初心者の中には、他人に勧められるがままにいきなり大量のプラグインを導入する人が結構な数います。この形のVimとのファーストコンタクトは実はやや不幸です。Vimの内部モデルを理解する機会を逸するからです。

Vimは大変に多機能かつ高機能です。そのためVimを活用するには膨大な学習が必要です。オプションやキーバインドにコマンド定義など、自分なりの設定をする必要があるからです。それらを正しく学んで正しく設定するためには、Vimの内部モデルを正しく理解している必要があります。

まったく同じことが優れたプラグインを使う際にも言えます。優れたプラグインというのは、往々にして利用者のニーズに応じて柔軟に設定できるようになっています。しかしプラグインはVimの上で動きますので、Vimの内部モデルをそれなりに知らなければ、自分にあった設定なんてできるはずがありません。

Vim本来の動作をその内部モデルに触れてもらおうというのが本書の趣旨です。取り上げる内部モデルの記述には若干正確性を欠く部分があります。これは単純化したほうが理解には良いだろうとの判断に基づいています。

Vimのモデルを巡る旅

早速Vimの動作と内部モデルを見ていきます。まずはVimで新しいファイル hello.go を開きましょう。そのコマンドは `:e` を使って以下の通りになります。

```
:e hello.go
```

この時Vimの中ではファイル hello.go に相当する バッファ が作成されます。1つのバッファには最大1つのファイルが関連付けられますが、ファイルが1つも関連付けられていないバッファもありえます。詳しくは `:help 'buftype'` などを参照すると良いでしょう。

続けてバッファに対して ウィンドウ が割り当てられます。今回は `:e` コマンドを使ったので、現在のウィンドウ、つまりコマンド実行直前にカーソルがあったウィンドウに割り当てられたバッファに変更がなければ、現在のウィンドウがファイル hello.go のバッファに割り当てられます。仮に既存バッファに既に変更があった場合は、通常は `E37` のエラーにより `:e` コマンドが失敗します。

まとめるとVimでファイルを表示するまでには、ファイルに対応したバッファが作られ、そのバッファがウィンドウに割り当てられる必要があります。実際それに加えて タブ がウィンドウを包含する形で存在するわけですが、説明を単純にするために本書ではこれ以上は触れません。いずれにせよこの機構を知っておくと、似たような操作をした時にVimの内部で起こることを理解・推測するのが容易になります。

やってみましょう。水平分割をしつつファイルを開く以下のコマンドを考えます。

```
:sp world.go
```

`:sp` コマンドはファイルに対応したバッファを作るところまでは `:e` コマンドと一緒です。そこからウィンドウを割り当てるところが違ってきます。 `:sp` では現在のウィンドウのその領域を水平(上下の二段)に等分に分割して新しいウィンドウを作ります。通常ならば、下段のウィンドウには分割前のバッファをそのまま割り当て、上段のウィンドウに先に作った新

しいバッファを割り当てます。以上により水平分割をしつつファイルが開かれます。

ここで1つ応用を考えてみましょう。水平ではなく垂直(左右の二段)に分割できないでしょうか? `:vsp world.go` とすればもちろんできます。

さらにもう1つの応用を考えてみましょう。先程の `:sp` では分割後の上段のウィンドウでファイルが開かれました。つまり新しバッファは上段のウィンドウに割り当てられました。これを下段にできないでしょうか?実は `:bo sp` というコマンドでできます。`:bo` コマンドは動作としては修飾子に近いコマンドで、後ろに指定するコマンドでウィンドウが分割された際に、下もしくは右のウィンドウを優先して割り当てる=使用するよう指示するというものです。詳しくは `:help :botright` を参照してください。

余談ですが `:vsp` は `:vert sp` と記述することができます。 `:vert` は `:bo` 同様に修飾子的なコマンドで、後ろのコマンドで分割されるウィンドウを垂直にする効果があります。

さてウィンドウにはウィンドウ番号が割り振られています。ウィンドウを分割した際には、分割後のウィンドウのうち上もしくは左のウィンドウに元の番号が割り振られ、もう一方のウィンドウにはそれに+1した番号が割り振られます。同時に既存のウィンドウのうち、分割対象よりも大きな番号を持ったウィンドウは全てウィンドウ番号が+1されます。

上で見た `:sp` は、分割したウィンドウのうち番号の小さい方を新しいバッファに割り当てるコマンドだったとわかるでしょう。 `:bo` はそれを大きい方を使うように変更するものなのです。

もう少しだけ先を見てみましょう。このようにして開かれたファイル hello.go にはGo言語の シンタックスハイライト が適用されているはずです。またGo言語用の幾つかの プラグイン も利用できるようになっています。これらはどのようなモデルにより実現されているのでしょう。

バッファには ファイルタイプ が設定される場合があります。ファイルタイプは `'filetype'` オプションによりバッファ毎に設定できます。Vimは

ファイルタイプが設定されると、自動的にそれに関連するシンタックス定義ファイルとファイルタイププラグイン等を読み込みます。もちろんこれらの読み込みをしないようにもできますが、その方法は本書では割愛します。 `:help :fileytpe` や `:help :syntax` を参照してください。

ファイルタイプの設定は自分で明示的に行うこともできますが、通常はVimの 自動コマンド により行われています。基本的には、新しいファイルを開いた時か新しいバッファの編集を始めた時に、そのバッファ名の拡張子からファイルタイプを決定しています。しかし幾つかのファイルタイプについては、もうちょっと複雑な判定方法を用いています。

このようなファイルタイプのための設定は `$VIMRUNTIME/filetype.vim` を中心に行われています。このファイルの中身を精査することで、各種のファイルタイプがどのように判定され適用されるかがわかります。自分でファイルタイプを追加する際 (`:help new-filetype`) には ftdetect というフォルダを作成するのですが、なぜそれが機能するかの答えもこのファイルに存在します。

以上で見てきたようにファイルを開くという人間にとってはシンプルなアクションでも、Vimの内部モデルの視点で見ると、幾つものリソースとそれにまつわる選択肢が機能したものでした。しかしそれらを一旦理解してしまえば、自分が何かを行いたい時にどこにどのように介入すれば良いか、そういうイメージを描きやすくなるはずです。

バッファは複数の 行 の、順序のある集合(リスト)という形で構成されます。

各行は バイト配列 として保持されます。これが文字の配列ではないことには注意が必要です。そうなっている理由は主にメモリ効率からなのですが、UNICODEが主体となった現在においても有効な考え方でしょう。しかしその内部表現、文字エンコードはオプション `'enc'` でユーザーが選択可能なことに特段の注意が必要です。

UNIXであれば `enc=utf-8` であることを前提にして良いかもしれません。しかし少し歴史の長いシステムであれば `enc=euc-jp` のほうが適切である

場合もあるでしょう。またかなり限定的ではありますが、UNIXでも `enc=cp932` を必要とすることがあったりもします。

Windowsでは未だに `enc=cp932` になっていることが少なくありません。Windowsで `enc=utf-8` を利用することに関しては、ここ数年でかなりの改善が進んでいます。しかし既存の資産との整合性ということを考えると `enc=cp932` にしたほうが安全であると言わざるをえません。筆者自身は `enc=utf-8` による利用でほとんど問題ないと考えていますが、そのあたりは察してください。

文字エンコードの違いによる最大の懸念はパフォーマンスです。カーソルを左に動かす=戻すような操作を考えた場合、 `utf-8` であればVimは理論的に O(1) のコストで行えます。しかし `cp932` の場合、文字列の長さ n に依存した O(n) のコストとなります。これはVimがカーソルを左に動かす際に、マルチバイト文字の先頭バイトを見つけるまで1バイトずつ左へ走査しているのと、ほぼ同等であることに基づきます。 `utf-8` は先頭バイトであるかどうかはその値を検査すれば即座にわかるのに対し、 `cp932` では行の先頭から走査することで初めてわかります。

これは短い行に対しての操作と考えれば無視して良いコストなのですが、長い行に繰り返し行うことを考えると膨大なコストになります。またVim全体で考えるとこの種の操作は頻繁に起きますので、パフォーマンスを重視する場合には `enc=utf-8` で利用することが望ましいといえます。

内部の文字エンコードが存在することは紹介しました。ではファイルのエンコードがこの内部のエンコードと異なるときには何が起こるのでしょう?もちろんエンコード変換が起こります。どのエンコードから変換したのかはバッファのオプション `'fenc'` に記録されています。変換元のエンコードはオプション `'fencs'` から選択されます。具体的には `'fencs'` に記載されたエンコードを先頭から順番に試して変換してみるだけのシンプルな仕組みです。

これは比較的うまく機能しますが、一部のケースの自動判定には確実に失敗することがあります。そのような場合にはこの機構を回避して、つまり

エンコードを指定してファイルを開くことができます。それには以下のように ++enc を指定します。

```
:e ++enc=euc-jp hello-old.txt
```

以上をまとめると次のようになります。バッファは行に、行はバイト配列に還元されます。ファイルはバッファとして読み込まれる際に、文字エンコードを変換され内部エンコードのバイト配列として取り扱われます。

Vimに入力されたキーは一旦キューに蓄えられます。キューでは登録されているキーマップに従って、その内容が変換されます。例として以下のようなマップが登録されている際に `<F2>` がキューに積まれると、後半の `a<C-R>…<ESC>` へと変換されます。

```
:map <F2> a<C-R>=strftime("%c")<CR><Esc>
```

この用に `:map` を使っている際には変換後の内容は再度キューに積まれ、キーマップによる再変換の対象となります。 `:noremap` を使っている場合には再変換の対象にはなりません。この2つのコマンドの背景には、このような処理方法の違いが存在します。

キーマップによる変換を経た入力キーのキューは、ストリームとしてVimの入力キー処理部分に引き渡されていきます。その処理部分では入力された内容に基づいて、モード変更を行ったり実際にVimの動作として解釈していきます。もちろんモードに応じて適用するキーマップは異なる場合はありえますが、今は難しく考えるのはやめておきましょう。

ここで以下のような共通部分を持つ複数文字の `{lhs}` からなる2つのキーマップがあったとします。

```
:map <silent> ,h /Header<CR>
:map <silent> ,f /Footer<CR>
```

この時に , だけが入力されたらどうなるでしょう。このケースではVimは摘要すべきキーマップを、次の1文字が入力されるまで確定できません。しかし暫く経つと , :前回の `f` 等の検索を反対方向に繰り返すオプション、として確定することでしょう。これは `'timeoutlen'` などのオプションの効果によります。Vimではキーマップによる入力キューの変換にタイムアウトを設けているのです。

導入のようなあとがき

本書はスパルタンVim6.0のプロトタイプ版です。Vim内部のモデルを理解してもらうことで、Vimそのものだけではなくプラグイン等についてももっとうまく使ってもらえるのではないか、ということをコンセプトにしています。そのため想定している対象読者はVim初級者から中級者です。

そのコンセプトに基づき実際に形にしてはみたものの、できあがった文については消化不良感が否めません。スパルタン成分に欠け、加えて肝心の目的を果たせているかどうかにも確信がもてません。この中途半端・消化不良感は別の機会に改訂版という形で解消を試みたいと考えています。うまくすれば夏くらいには。

今回は語りかけるような流れるような筆致を心がけました。いままでのスパルタンVimとはちょっと違う雰囲気になっているかもしれません。もっとも一貫したスタイルなどないのですけれど。この書き方ももう少し突き詰めてみたいところです。

2017/04/09 村岡太郎 (KoRoN, @kaoriya)