スパルタンVim 6.1

MURAOKA Taro (KoRoN, @kaoriya)

α版(2017/08/11)

モデルを知るべき理由

Vimを使い始めたばかりの初心者の中には、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相当数いるようです。しかしこの形でのVimとのファーストコンタクトは、Vimの内部モデルを理解する機会を逸するという意味でやや不幸です。

Vimはとても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になります。オプションやキーバインドにコマンド定義など、自分好みの設定をするには多くの知識が必要になります。それらを正しく学んで設定するためには、Vimの内部モデルを少なからず理解している必要があるのです。

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、自分にあった設定をするのは不可能と言っても過言ではありません。

Vimの動作をその内部モデルにまで踏み込みながら理解し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説明には若干正確性を欠く部分がありますが、それらは単純化したほうが理解しやすいだろうとの配慮に基づくものです。

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。

ファイルを開く時のモデルの振る舞い

まずはVimで新しいファイル hello.go を開いてみましょう。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(図1)。詳しくは :help 'buftype' を参照してください。

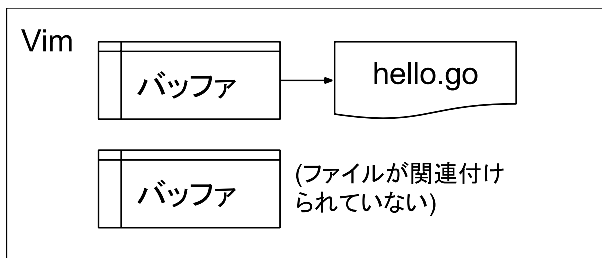

図 *1.* ファイルとバッファの関係

続けてバッファに対して ウィンドウ が割り当てられ、開いたファイルがユーザーから操作可能な状態になります。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へ割り当てられている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(図2)。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。

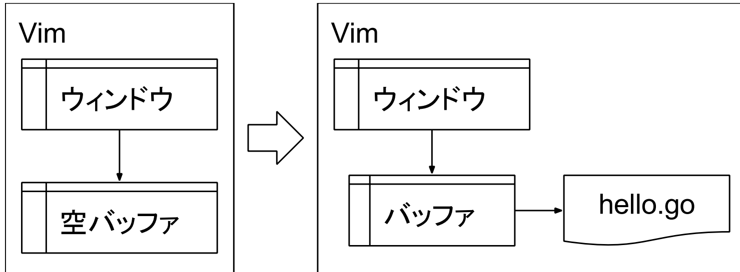

図 *2.* ウィンドウとバッファの関係

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推測し理解するのが容易となります。

では実際にウィンドウの水平分割をする際にVim内部で起こることを見てみましょう。次のコマンドはウィンドウを水平分割しつつ別のファイルを開くコマンドです。

```
:sp world.go
```

このコマンドを実行する前のVimは 図3 のような状態になっているとしましょう。

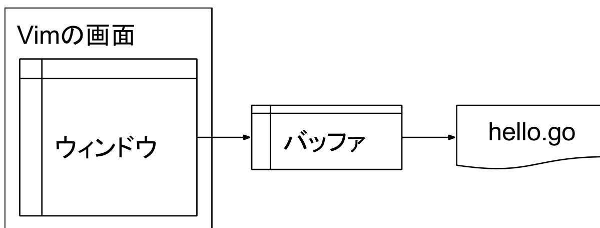

図 *3. split*による分割前

`:sp` コマンドはまずウィンドウを水平に分割します。分割後のウィンドウは分割直前のバッファを指しています。つまり hello.go が表示されています(図4)。 `:sp` コマンドに引数を渡していなければ話はココで終わります。

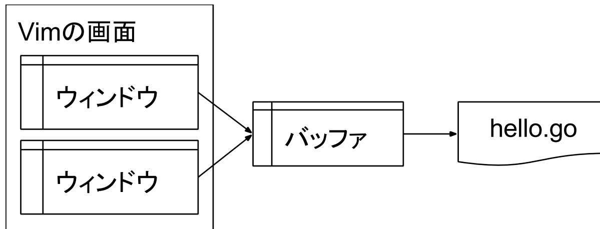

図 4. *split*による分割直後

しかし今回は world.go を開くことを引数で指定していますので、分割後のカレントウィンドウ、すなわち上のウィンドウで world.go を開きます。ここから先にに起こることは `:e` コマンドとほぼ一緒です。ファイル world.go を開いて新しいバッファを関連付け、さらに上のウィンドウに関連付けます(図5)。以上により水平分割をしつつファイルが開かれます。

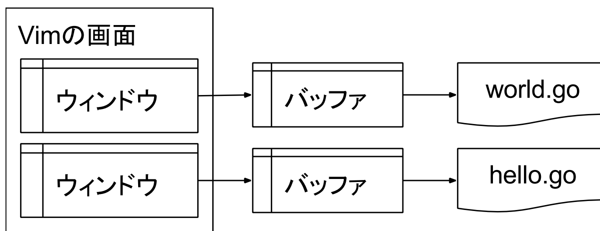

図 5. *:sp world.go* の実行後

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? `:sp world.go` の代わりに `:vsp world.go` とすればもちろんできます(図6)。

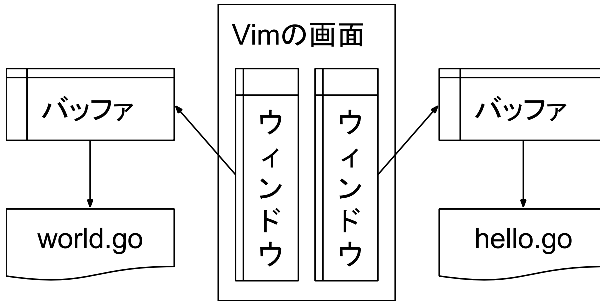

図 *6. :vsp world.go* の実行後

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使用するよう指示するコマンドです。詳しくは `:help :botright` を参照してください。

余談ですが `:vsp` は `:vert sp` と記述することもできます。 `: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(図7)。

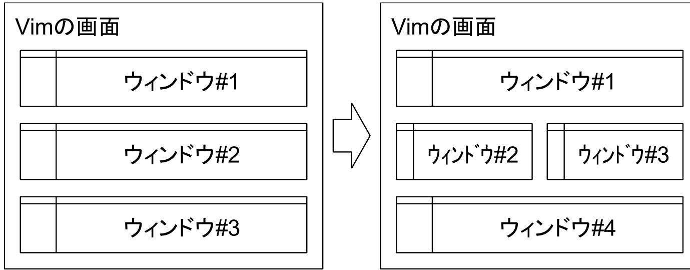

図 7. *:vsplit* 後のウィンドウ番号

図7 はウィンドウ#2で `:vsplit` した際にウィンドウ番号がどのように変わるかを示しています。ウィンドウ#1は変わらず、#2は#2と#3に分割され、#3は#4に変わります。なおカレントウィンドウの番号は `:echo winnr()` で調べられます。

この動作を知っていると、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使用す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(図8)。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。

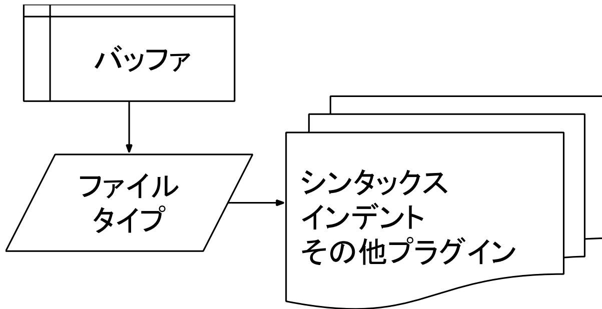

図 *8.* バッファに設定されたファイルタイプとその他のリソース

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の中に存在します。

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関与したものでした。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。

バッファの構成モデル

次はバッファの中身を見てみましょう。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(図9)。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。

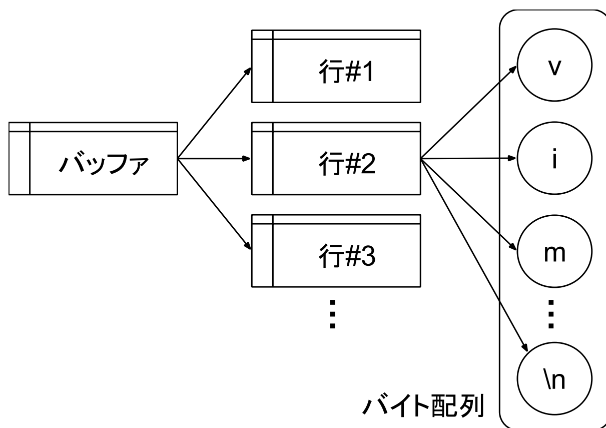

図 *9.* バッファと行とバイト配列

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` を必要とする環境があったりもします。

Windows では 未だに `enc=cp932` で 使うことが 推奨 されています。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` で利用しておりほとんど問題なく使えていますが、 `ecn=utf-8` を推奨できない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` においてはあるバイトが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程度のコストなのですが、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相当に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` で利用するほうが望ましい場合が多いと言えます。

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(図10)。

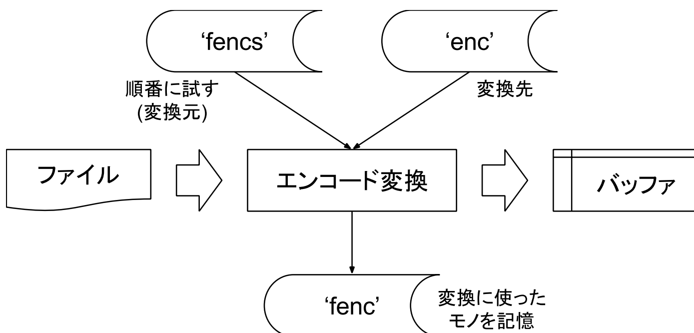

図 *10.* ファイル読み込み時のエンコード変換

逆にバッファの内容を保存する場合はオプション `'fenc'` のエンコードで変換してファイルへ書き込みます(図11)。

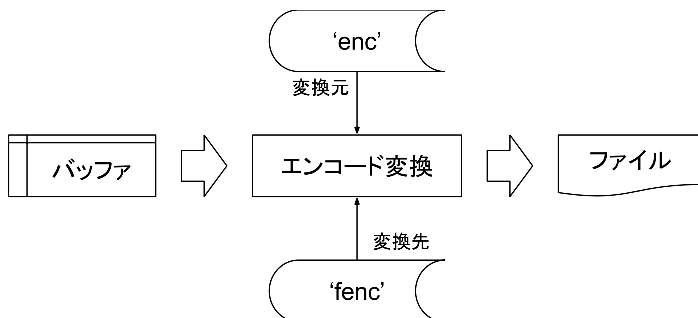

図 *11.* ファイル書き込み時のエンコード変換

これは比較的うまく機能しますが、ファイルの内容次第では自動判定に確実に失敗する場合があります。そのようなファイルはこの機構を回避して、つまりエンコードを直接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` を使用します。

```
: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。

キー入力モデル

最後にキーの入力モデルを簡単に解説します。

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たのち、Vimのコマンドとして解釈されます(図12)。

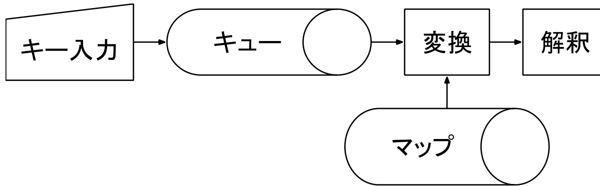

図 *12.* キー入力のパイプライン

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(図13)。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。

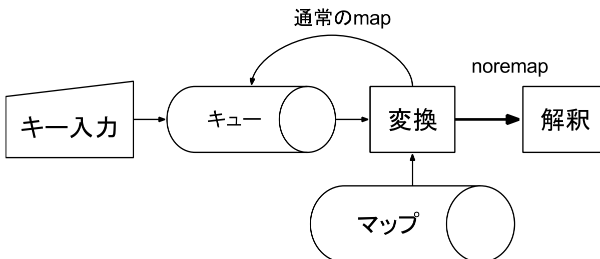

図 *13.* キー入力のマップと再変換

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が異なる場合はありますが、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(図14)。

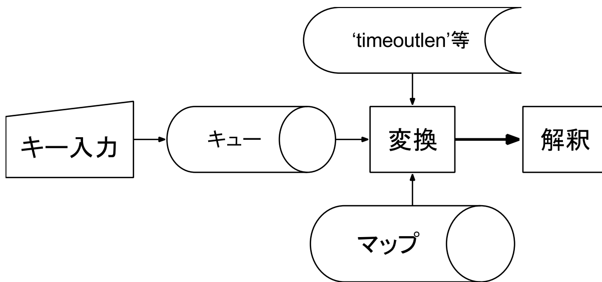

図 *14.* 未決定なキーマップのタイムアウト

あとがき

本書はスパルタンVim6.xのα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、というの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。

本書は2017年4月の技術書典で出版したものの改訂版です。説明のための図を大幅に追加し、それに伴い細かく文章を修正しています。結果として思い描いていた形に随分と近づいてきましたが、まだ足りない部分(主にスパルタン成分)も多く残ってしまいました。

またリベンジしたいなと考えています。うまくすれば2017年10月の技術書典でしょうか。

2017/08/11 村岡太郎 (KoRoN, @kaoriya)