SPARTAN VIM 4.0

~Dark side of Vim script~

MURAOKA Taro
(KoRoN, @kaoriya)

スパルタン Vim 4.0

～Dark side of Vim script～

はじめに

先日、私も執筆に参加させていただいた「Vim script テクニックバイブル」という書籍が技術評論社さんから発売されました。この本は Vim script を完全な初心者が学ぶための教科書としての位置づけでした。そのため、どうしても堅実で抑制した執筆作業が求められ、なんともフラストレーションが溜まってしまいました。

そこで今回のスパルタン Vim では Dark side of Vim script と題して、Vim のソースコード(C 言語)のうち Vim script を実装した部分を、Vim を使って鑑賞・解析していきます。その過程を細かく説明することで、Vim の実装と Vim script の表からは見えない性質と、スパルタン Vim 的な操作をすべて学べてしまう、ついでに筆者としての私のフラストレーションを解消してしまおう、という算段です。

どうぞお付き合いいただけたら幸いです。

準備

対象となる Vim のバージョンは 7.4.404 です。文中のファイル名、内容やその行数は、すべてこのバージョンを基準としています。完全なソースコードは以下の URL で閲覧できます。

http://goo.gl/bzJwdv

主に見るべきソースは src/eval.c になりますが、他のファイルを覗き見ることもあるので、ソースコードをチェックアウトし、ctags を用いて tags ファイルを生成しておきましょう。

```
$ git clone https://github.com/vim-jp/vim.git
$ cd vim
$ git checkout -b spartan dce0029fd7264840879a6935565c0e9ece5e2cff
$ cd src
$ ctags -R *.c *.h
```

この状態で Vim を起動したら鑑賞開始です。

```
$ vim
```

もちろん起動するのは gvim でも構いません。

鑑賞

source コマンド

Vim script の実行といえば、やはりファイルから Vim script を読み込んで実行する `:source` コマンドが花形でしょう。

手始めにその実装をみてみましょう。`:tag ex_source` してください。ex_cmds2.c の 2882 行目に ex_source()があります。この関数がコマンドの実体です。スタート地点は eval.c ではありませんが、ここから初めて eval.c に辿り着くまでを、最初の目標としてみます。

```
/*
 * ":source {fname}"
 */
    void
ex_source(eap)
    exarg_T     *eap;
{
(中略)
            cmd_source(fname, eap);
(中略)
        cmd_source(eap->arg, eap);
}
```

このように実体は cmd_source()であることがわかります。`/cmd_<CR>` で `cmd_source` の上に移動して、タグジャンプ `<CTRL-]>` しましょう。この cmd_source()も大それたことはしてなくて、実体は do_source()です。すぐに `/do_<CR><CTRL-]>` で飛んでしまいましょう。

do_source()はいろいろとやっていますが、まず注目すべきは 3101 行目で

す。

```
cookie.fp = mch_fopen((char *)fname_exp, READBIN);
```

ここで実際に読み込むファイルをオープンし、source_cookie 構造体の fp フィールドに保存しています。

次に見るべきは、この cookie をいつ利用するか、です。正解は少し下って 3318 行目にあります。

```
/*
 * Call do_cmdline, which will call getsourceline() to get the lines.
 */
do_cmdline(firstline, getsourceline, (void *)&cookie,
                                DOCMD_VERBOSE|DOCMD_NOWAIT|DOCMD_REPEAT);
```

do_cmdline()はソースから読み込んだテキストを、コマンドライン、つまり Ex モードとして解釈・実行する関数です。第 1 引数の firstline はファイルから先読みした 1 行目の内容で、第 2、第 3 引数はソースの抽象レイヤとなっています。そして第 4 引数は読み込み時のオプションです。

この関数は様々なソースから抽象的にテキストを読み込むために、読み込むための関数(getsourceline)と、その関数に与えるデータ((void *)&cookie)を引数に取っています。

ちょっとここで寄り道して引数になってる getsourceline()を見ておきましょう。`:stag getsourceline`で同ファイルの 3526 行目に飛びましょう。ついでに`<CTRL-W>_`で縦方向に最大化。

```
    char_u *
getsourceline(c, cookie, indent)
    int         c UNUSED;
    void        *cookie;
    int         indent UNUSED;
{
    struct source_cookie *sp = (struct source_cookie *)cookie;
```

3 つの引数のうち大事なのは cookie だけです。この関数が呼び出されるときには、先ほどの do_cmdline()の第 3 引数の値が渡されます。それにより読み込みソースとなるファイルがこの関数に伝わります。

ファイルから読み込む処理は、 3576 行目に見られる get_one_sourceline()に委ねられます。

```
sp->nextline = get_one_sourceline(sp);
```

さらに読み進めると物理的なファイルの読み込みには、 3678 行目近辺の fgets()が使われることがわかります。

```
if (fgets((char *)buf + ga.ga_len, ga.ga_maxlen - ga.ga_len,
                                                sp->fp) == NULL)
        break;
```

さぁ、これで物理ファイルと do_cmdline()が繋がりました。ではウィンドウを `<CTRL-W>c` で閉じて 3318 行目、do_cmdline()の呼び出しへ戻りましょう。

その勢いで `^<CTRL-]>` で関数 do_cmdline()の内容に移動します。do_cmdline()は ex_docmd.c の 755 行目で、シグネチャが以下のように定められています。

```
    int
do_cmdline(cmdline, fgetline, cookie, flags)
    char_u  *cmdline;
    char_u  *(*fgetline) __ARGS((int, void *, int));
    void  *cookie;    /* argument for fgetline() */
    int   flags;
```

既にざっくり解説しましたが、とりあえず引数 fgetline がどう使われるかを `/fget<CR>*` で追ってみます。

次に注目すべきは 1008 行目です。

```
cmd_getline = fgetline;
cmd_cookie = cookie;
```

おっと、ここで cmd_getline への乗り換えが起こっていますね。ここの直上に次のような不穏なコメントがありますが、本書では無視します。コマンドの実行関数が、その読み込み方をめっちゃ制御してます。

```
if (cstack.cs_looplevel > 0)
{
    /* Inside a while/for loop we need to store the lines and use them
     * again.  Pass a different "fgetline" function to do_one_cmd()
     * below, so that it stores lines in or reads them from
     * "lines_ga".  Makes it possible to define a function inside a
     * while/for loop. */
```

1 つのコマンドの実行

気を取り直して cmd_getline を追いましょう。すると使っている場所は 1130 行目しかありません。簡単です。

```
next_cmdline = do_one_cmd(&cmdline_copy, flags & DOCMD_VERBOSE,
                          &cstack,
                          cmd_getline, cmd_cookie);
```

このシグネチャから do_one_cmd()の動作は、1 つコマンドを実行して次に実行すべき行を返す、だと推測できます。1701 行目を見てみましょう。

この do_one_cmd()は、実に 1100 行にも及ぶ巨大な関数です。普通に読んでいては時間がいくらあっても足りません。まずは関数のコメントを確認します。

```
/*
 * Execute one Ex command.
 *
 * If 'sourcing' is TRUE, the command will be included in the error mes
sage.
 *
 * 1. skip comment lines and leading space
 * 2. handle command modifiers
 * 3. parse range
 * 4. parse command
 * 5. parse arguments
 * 6. switch on command name
```

ここでは `6. switch on command name` にだけ着目し、`/switch on` で検索してみましょう。2675 行目に行き着くはずです。そこからソースコードを少し先、2701 行目まで読んでみましょう。

```
/*
 * Call the function to execute the command.
 */
ea.errmsg = NULL;
(cmdnames[ea.cmdidx].cmd_func)(&ea);
```

コマンドの実行部分がありました。cmd_func フィールドの関数ポインタの指す先がコマンドの本体となります。ちょっと`:stag cmd_func`でその実体を覗いてみましょう。ex_cmds.h の 81 行目付近に行きつきます。

```
typedef void (*ex_func_T) __ARGS((exarg_T *eap));

static struct cmdname
{
    char_u      *cmd_name;      /* name of the command */
    ex_func_T   cmd_func;       /* function for this command */
    long_u      cmd_argt;       /* flags declared above */
}
```

そのすぐ下には、次のような cmdnames の定義が存在します。なおこのソースはわかりやすさのために一部編集しています。

```
cmdnames[] =
{
EX(CMD_append,          "append",       ex_append,
                        BANG|RANGE|ZEROR|TRLBAR|CMDWIN|MODIFY),
EX(CMD_abbreviate,      "abbreviate",   ex_abbreviate,
                        EXTRA|TRLBAR|NOTRLCOM|USECTRLV|CMDWIN),
EX(CMD_abclear,         "abclear",      ex_abclear,
                        EXTRA|TRLBAR|CMDWIN),
(以下略)
```

つまりコマンドは cmdname 構造体の配列として定義され、その構造体は名前、関数ポインタ、フラグから構成されていることがわかります。そのため先に見た

```
(cmdnames[ea.cmdidx].cmd_func)(&ea);
```

は、たくさんあるコマンド中から 1 つを選択して、実行することに他なりません。

さぁ、そろそろ eval.c に到達しそうです。ここでは調べるコマンドを:echo と:call に限定します。「Vim script テクニックバイブル」にも記載しましたが、Vim script は echo の引数や、call の引数によって評価・実行されているからです。

まずは:echo を見てみましょう。ex_cmds.h で/echo<CR>すると、340 行目に以下が見つかります。

```
EX(CMD_echo,            "echo",         ex_echo,
                        EXTRA|NOTRLCOM|SBOXOK|CMDWIN),
```

これは関数 ex_echo が本体であることを示していますので、早速 ex_echo までカーソルを動かして<CTRL-W><CTRL-]>で、本体を見てみましょう。

ついに eval.c に行き着きました。しかも 21538 行目です。1 つのファイルがこのサイズであることから、もう魔境であることがうかがい知れるでしょう。少しだけ読み進めると、21558 行目に以下の関数呼び出しが見つかります。

```
if (eval1(&arg, &rettv, !eap->skip) == FAIL)
```

この eval1()こそが Vim script の本丸、評価機能の関数です。ここで:call についても同様に:tag ex_call して少し先を見ておきましょう。3428 行目です。

```
if (eval0(eap->arg, &rettv, &eap->nextcmd, FALSE) != FAIL)
    clear_tv(&rettv);
```

また同様に eval0()の中身の、4055 から 4065 行目をざっくりと眺めておきましょう。

```
    static int
eval0(arg, rettv, nextcmd, evaluate)

(中略)
    ret = eval1(&p, rettv, evaluate);
```

こちらも eval1()を呼び出しています。やはり「eval1()こそが Vim script である」と言って良いようです。

eval1

ついに Vim script の正体である eval1()に到達しました。まずは手始めに関数の説明を見てみましょう。

```
/*
 * Handle top level expression:
 *      expr2 ? expr1 : expr1
```

ここで「あれ?」と気がついた方は、なかなかの勉強家、かつ大した記憶力の持ち主と言えます。実は、これは`:help expr1`の以下の記述と同じです。

```
expr1                                                   *expr1* *E109*
-----

expr2 ? expr1 : expr1
```

つまり eval1()は expr1 の内容、三項演算子を実装した関数です。コノように eval.c には幾つか evalX()という関数が定義されていますが、それぞれが`:help exprX`を実装しています。(X は数字に置き換えられます)

では eval1()の中身を順番に見てみましょう。まずは 4089 行目、関数のコメントの続きです。

```
 * "arg" must point to the first non-white of the expression.
 * "arg" is advanced to the next non-white after the recognized express
ion.
```

引数 arg には、Vim scrip として次に解釈すべき非空白な文字を指すポインタが入っています。また関数を抜ける際には、次に解釈すべき文字を指すようポインタを更新しなければなりません。これはすべての evalX()関数に共通したインターフェースとなっています。

次に 4105 行目を見てみましょう。

```
/*
 * Get the first variable.
 */
if (eval2(arg, rettv, evaluate) == FAIL)
    return FAIL;

if ((*arg)[0] == '?')
{
```

コメントやヘルプに書かれた通り、`expr2`を評価・実行するために eval2()関数を呼び出しています。結果は rettv に格納されます。また arg は次の解釈文字を指すように更新されています。

そして直後で`(*arg)[0] == '?'`により、文字「?」が存在するか確認して、存在すれば if 文の中で三項演算子を実行し、なければそのまま eval2()の実行結果を eval1()自身の結果として終了しています。

続いて三項演算子の実行内容を確認しましょう。4113 行目から。

```
result = FALSE;
if (evaluate)
{
    int         error = FALSE;

    if (get_tv_number_chk(rettv, &error) != 0)
        result = TRUE;
```

eval2()の戻り値を、数値として解釈して 0 でなければ、result を TRUE にしています。変数 evaluate については TRUE であるものとして、いったん解説を保留し、続き(4129 行目)を読み進めてしまいましょう。

```
if (eval1(arg, rettv, evaluate && result) == FAIL) /* recursive! */
    return FAIL;
```

これは三項演算子の第 2 項の expr1 に相当し、自分自身の再帰呼び出しになっています。ただし第 3 引数は evaluate と result ともに TRUE の時にだけ TRUE になっています。このことから第 3 引数である evaluate は、実際に評価・実行するかどうかを制御するフラグであることが明白です。つまり evaluate が FALSE の時は、evalX()系の関数は、Vim script を解釈するが実行はしない、ということです。

三項演算子の動作としてこの場所をもう一度考えなおすと、eval2()の結果が FALSE ならば、解釈は行うが実行はしない、また arg は更新するということがわかります。逆に TRUE ならば、もちろん解釈と実行を伴うわけですね。

続く 4132 行目は、三項演算子の一部「:」のチェックです。

```
/*
 * Check for the ":".
 */
if ((*arg)[0] != ':')
{
    EMSG(_("E109: Missing ':' after '?'"));
```

「:」がなければ当然エラーとなります。

そして 4147 行目からは第 3 項の評価となります。

```
if (eval1(arg, &var2, evaluate && !result) == FAIL) /* recursive! */
{
    if (evaluate && result)
        clear_tv(rettv);
    return FAIL;
}
if (evaluate && !result)
    *rettv = var2;
```

第 2 項の時とは、第 3 引数における result の判定が、逆になっていることがポイントです。

これで eval1()の実装は、ひと通り見終わりました。Vim script においては解釈と実行が同時に行われている、ということが理解できたことでしょう。他の evalX()系の実装も、同様の構造になっています。

まとめ

Vim において Vim script がどのように実装・実行されているのかを、ファイルからの読み込みと最初の構文(expr1)をたどることで、詳細に解説しました。これにより、最新の言語実装系であれば字句解析器(lexer)、構文解析器(parser)、実行器(evaluator)を分けるところを、Vim script ではそれらを一緒にやっていることがわかりました。

このすべてを一緒に実装してしまう方式は、言語の実装手段としてはとても古いものであり、実行速度が遅くなったり、言語的な拡張性を犠牲にしたりと、さまざまなデメリットがあります。しかし本書で解説したように、どうやって動作しているのか理解しやすいというメリットもあります。

つまり Vim script を Vim のソースコードから見るということは、言語の実装手段の入門として、非常に良い教科書となる可能性があります。みなさんも Vim の eval.c を読むことで、新たなプログラミング言語を生み出すことができるかもしれません。

その言語でのプログラミングには、Vim が使用されることを願って止みません。

あとがき

スパルタン Vim として 4 作目、ついに Vim のソースコードを読むことに突入しました。エディタを使うのに、ソースコードまで読ませる...なんていうスパルタン。

例によって半日で書き上げた文章ではありますが、「Vim のソースコードを読ませたい」という構想は、スパルタン Vim の企画当初から温めていたという意味で、数年越しとなります。

折しも Vim script の本(略称:Vim スク本)の発売に合わせ、Vim の中でも比較的読みやすい Vim script の実装部分(eval.c)への入門書として、それなりの役割は果たせたのではないかと感じています。

読者の皆さまにおかれましては少しでも楽しんで、そして 1 人でも多くの方に Vim のソースへ踏み入れてもらえれば、と願う次第です。

2014/08/17 村岡 太郎 (KoRoN, @kaoriya)