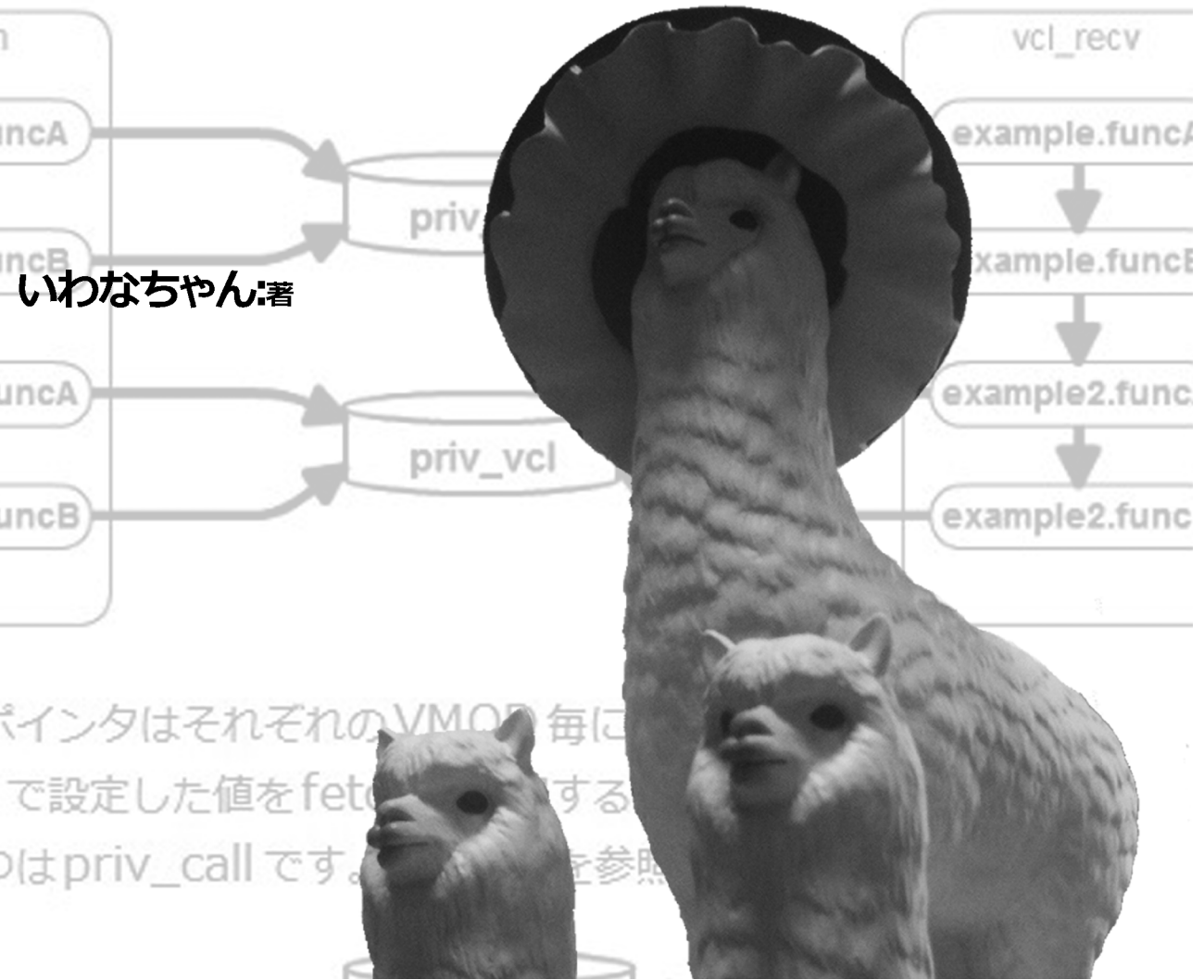
reverseproxy

Varnish Cache

inline-C/VMOD GUIDEBOOK

inline-C/VMOD ガイドブック



いわなちゃん:著

目次

本書の対象者

本書は以下の方を対象とします。

・Varnish Cache(以下 Varnish)を利用したことがある
・各種 Varnish のツール(varnishlog など)の使用の仕方がわかる
・C 言語を触ったことがある

もし Varnish を使用したことがない・まだ不安だという方は、是非当 CD に同梱している「VarnishCache 入門」の一読をおすすめします。
また環境は以下で行なっています

```
OS:       Scientific Linux6
Varnish: 3.0.2
```

Varnish の高度なカスタマイズ

Varnish は DSL（ドメイン固有言語）の VCL が搭載されており、まさにプログラミングを行なっているような非常に自由度が高い設定が可能です。
しかし VCL では変数の新規定義や、一部の変数の型変換が出来なかったりなど、
少しむず痒い制限が存在することも事実です。このような制限を回避するために
Varnish では以下の2つの解決策を用意しています。

・インライン C
・Varnish Module(以下 VMOD)

インライン C は文字通り VCL 中に C 言語を書くことができる方法で、ちょっとした機能追加を気軽に出来る機能です。
例えば503をバックエンドから受け取った際に syslog を出力する場合は以下のように記述できます。

```
//ファイルの先頭
C{
  #include <syslog.h>
}C
//～中略～
sub vcl_fetch{
  if(beresp.status == 503){
    C{
      syslog(LOG_INFO, "Status 503 url = %s" , VRT_r_req_url(sp));
    }C
  }
  return(deliver);
}
```

C{～}C で囲んだ箇所がインライン C です。このように非常に気軽に書けるのがわかります。
しかしここで少し考えてみましょう。もしコードが長い場合はどうでしょうか？
ファイルを外出しして include するのも有りですがあまりスマートではありません。
関数化を行ったとしても、呼び出す際にはインライン C で書く必要があります。
例えば、req.url の値をインライン C で取得する場合は VRT_r_req_url(sp)と、書かなくてはならないなどコードのメンテナンス性にも難があります。
また、他の共有ライブラリにリンクする必要があったりする場合は起動パラメータの cc_command でリンクしたいモジュールを書いたりする必要があったりします。
そこでもうひとつの解決策が VMOD です。
名前の通りモジュールで纏まっており、VCL から使う際もインライン C を書く必要がありません。
例えば標準で配布されている std という VMOD を使い syslog を出力する際は以下のように行います。

```
import std;
sub vcl_fetch{
  if(beresp.status == 503){
    std.syslog(6,"Status 503 url = " + req.url);
  }
  return(deliver);
}
```

非常に分かりやすいです。
ではインライン C と VMOD はどのように使い分ければいいのでしょうか？
これは筆者個人の考えですが、以下にまとめました。

インライン C

- 頻繁に書き換えが発生する場合
- 特定のフローの中でしか使わない場合
- 比較的軽い処理

VMOD

- 関数にまとまる場合
- 外部の共有ライブラリを利用する場合
- 関数内やモジュール内で共有リソースを使いたい場合
- 初期化・終了処理が必要な場合

他にも様々な判断基準はありますが、上記項目は抑えておきたいです。
またどちらを使うにせよ C 言語や Varnish 特有の各種関数の知識は必須です。
次章から解説します。

インラインCを使う際に必要な基礎知識

公式のドキュメントにはインラインCを使う上で詳細なドキュメントは充実しているとは言えません。そのため、基本的に記述法を覚えるには

VCLをダンプして解析する
Varnishのソースを読む

といった覚悟が必要です。
実際にダンプの方法やソースを読む上でのポイントについて解説します。

VCLのダンプ

Varnishのコマンドを以下のようにするとVCLをCに変換した際のコードが出力されます。
出力されたコードはインラインCを書く上で非常に参考になります。もちろんそのまま出力された内容をC{～}Cで囲っても同じ動作を得られます。

コマンド

```
varnishd -d -f [VCLファイル名] -C
```

VCL

```
 1 backend default {.host="192.168.1.199";.port="81";}
 2 backend admin   {.host="192.168.1.199";.port="82";}
 3
 4 sub vcl_recv{                                          ・・・(1)
 5       if(req.url ~ "^/admin/"){                        ・・・(2)
 6             set req.backend=admin;
 7       }else{
 8             set req.backend=default;                   ・・・(3)
 9       }
10       return(lookup);                                  ・・・(4)
11 }
```

変換された VCL（一部抜粋）

```
444 static int
445 VGC_function_vcl_recv (struct sess *sp)                    ・・・(1)
446 {
447   /* ... from ('input' Line 4 Pos 5) */
448   {
449     {
450       VRT_count(sp, 1);
451       if (
452         VRT_re_match(VRT_r_req_url(sp), VGC_re_2)          ・・・(2)
453       )
454         {
455           VRT_count(sp, 2);
456           VRT_l_req_backend(sp,          VGCDIR(_admin)
457           );
458         }
459       else
460         {
461           VRT_count(sp, 3);
462           VRT_l_req_backend(sp,          VGCDIR(_default)  ・・・(3)
463           );
464         }
465       VRT_done(sp, VCL_RET_LOOKUP);                        ・・・(4)
466     }
467   }
468   /* ... from ('Default' Line 40 Pos 5) */
469   {
470     {
471       VRT_count(sp, 4);
472       if (
473         (VRT_r_req_restarts(sp) == 0)
中略

538         )
539       )
540         {
541           VRT_count(sp, 13);
542           VRT_done(sp, VCL_RET_PASS);
543         }
544       VRT_count(sp, 14);
545       VRT_done(sp, VCL_RET_LOOKUP);
546     }
547   }
548 }
549
```

VCL と変換された VCL の横の数字は対応する行です。

このようにVCLを普通に書いたあとCに変換をかけて、どのように対応するのかを確認します。
変換されたコードは以下のようなブロックで構成されています。
なお行数は、配布されているdefault.vclのbackendの定義のみコメントを外したもので変換した時に対応しています。

構造体・定数・各種変数の定義（行：1～399）

バックエンドやディレクターの構造体の定義などが記述されています。
またreq.urlなど変数を読み書きする際に利用する関数などの定義もあります。

バックエンド・ACLや正規表現の変数などの定義（行：400～424）

定義したバックエンドやACLなどが定義されています。

アクション定義(vcl_recvなど）（行：425～691）

vcl_recvなど定義したアクションが記述されています。
各アクションは更に以下のようにブロックで構成されます

```
static int VGC_function_[アクション名] (struct sess *sp)
{
/* ... from ('input' Line [ライン番号] Pos [ポジション番号]) */
～ユーザが入力したVCLをCに変換した内容～
/* ... from ('Default' Line [ライン番号] Pos [ポジション番号]) */
～デフォルトのVCLをCに変換した内容～
}
```

アクション名はvcl_recvやvcl_fetchなどが入ります。
そしてライン番号・ポジション番号はユーザの入力(input)したVCLとデフォルト(default)のVCLのどの行数からの内容かと行頭からの文字数に一致します。
またここを見ればわかるように、ユーザの入力したVCLの後に必ずデフォルトのVCLが埋め込まれます。そのため各アクションで、明示的にreturn(lookup)などをしないとデフォルトのVCLが動き、想定している動きと変わることがあります。

VRT_count用テーブル（行：692～719）

VarnishはVCLの動作をトレースしています。そのためアクションの開始やif文といった処理の分岐する箇所にVRT_count関数を挿入します。
この番号はVCLのどの場所なのかをテーブル化しています。

VCL のコンストラクタ・デストラクタ（行：720～736）

これは VCL の初期化時・終了時に呼び出されるもので vcl_init/vcl_fini アクションとは関係有りません。
正規表現のコンパイルや VMOD のロードなどを行なっています。

変換された VCL の元（行：737～1022）

ユーザが記述した VCL とデフォルトの VCL が記述されています。

VCL の設定（行：1023～1047）

Varnish が処理の際に使う VCL の設定が入っています。

Varnish のソースを読む

一つ一つの記述を VCL で書いてみて、C に変換して確認するのは非常に手間がかかります。
かと言って Varnish のソースをすべて見て、理解するのは非常に難しいです。
また、あまり望ましくはないですが、内部の関数を巧みに使いトリッキーなことをするなど、高度な処理を行う上で Varnish の動きも最低限把握する必要があります。
そこでポイントとなるソースと読み方について解説します。

/lib/libvcl/generate.py

VCL で利用する各種アクションの戻り値、変数の一覧・型など非常に重要な内容が記述されています。このファイルは以下の内容を含みます。

VCL のトークン一覧

tokens に使用可能な演算子などが定義されています。

アクション(vcl_recv など)で利用可能な戻り値一覧

returns に定義されています。

```
('pipe',  ('error', 'pipe',)),
```

上記が表しているのは、vcl_pipe では return する際に error と pipe を指定可能ということです。

変数の一覧

sp_variables に req.url などの変数の一覧とそれぞれがどこのアクションで
どのように利用可能か・型名などが以下のように定義されています。

```
('bereq.between_bytes_timeout',          //変数名
        'DURATION',                      //変数の型
        ( 'pass', 'miss',),              //変数の読み込みが可能なアクションリスト
        ( 'pass', 'miss',),              //変数の書き込みが可能なアクションリスト
        'struct sess *'                  //読み書きする際の関数の引数の Prefix
),
```

アクションの指定で、全てのアクションで使える all と vcl_ini,vcl_fini を除いた
全てで使える proc があります。

ストレージ変数一覧

stv_variables にストレージの変数が定義されています

VCL での変数の型一覧

vcltypes に使用可能な型が定義されています。それぞれ VCL での型名と
それを C に解釈した際の型がマッピングされています。

また、この generate.py はその名の通りファイルを生成するもので、
以下のファイルを生成します。

```
/lib/libvcl/vcc_token_defs.h
/include/vcl_returns.h
/include/vcl.h
/include/vrt_obj.h
/lib/libvcl/vcc_obj.c
/lib/libvcl/vcc_fixed_token.c
/include/vrt_stv_var.h
```

/lib/libvcl/vcc_obj.c

generate.py から生成された VCL で利用可能な変数の一覧で、以下のように定義
されています。

```
{ "bereq.between_bytes_timeout", DURATION, 27,     //変数名,型名,変数名の長さ
    "VRT_r_bereq_between_bytes_timeout(sp)",       //変数を読む際の関数名
    VCL_MET_PASS | VCL_MET_MISS,                   //変数を読めるアクション
    "VRT_l_bereq_between_bytes_timeout(sp, ",      //変数を書く際の関数名
    VCL_MET_PASS | VCL_MET_MISS,                   //変数を書けるアクション
    0,
},
```

変数を読み/書き込む際の関数名がインラインCで該当の変数を利用する際に使う関数名になります。

ただし変数の型がHEADERの場合はこれに当てはまりません。

```
{ "req.http.", HEADER, 9,
   "VRT_r_req_http_(sp)",
   VCL_MET_RECV | VCL_MET_PIPE | VCL_MET_PASS | VCL_MET_HASH
    | VCL_MET_MISS | VCL_MET_HIT | VCL_MET_FETCH | VCL_MET_DELIVER
    | VCL_MET_ERROR,
   "VRT_l_req_http_(sp, ",
   VCL_MET_RECV | VCL_MET_PIPE | VCL_MET_PASS | VCL_MET_HASH
    | VCL_MET_MISS | VCL_MET_HIT | VCL_MET_FETCH | VCL_MET_DELIVER
    | VCL_MET_ERROR,
   "HDR_REQ",
},
```

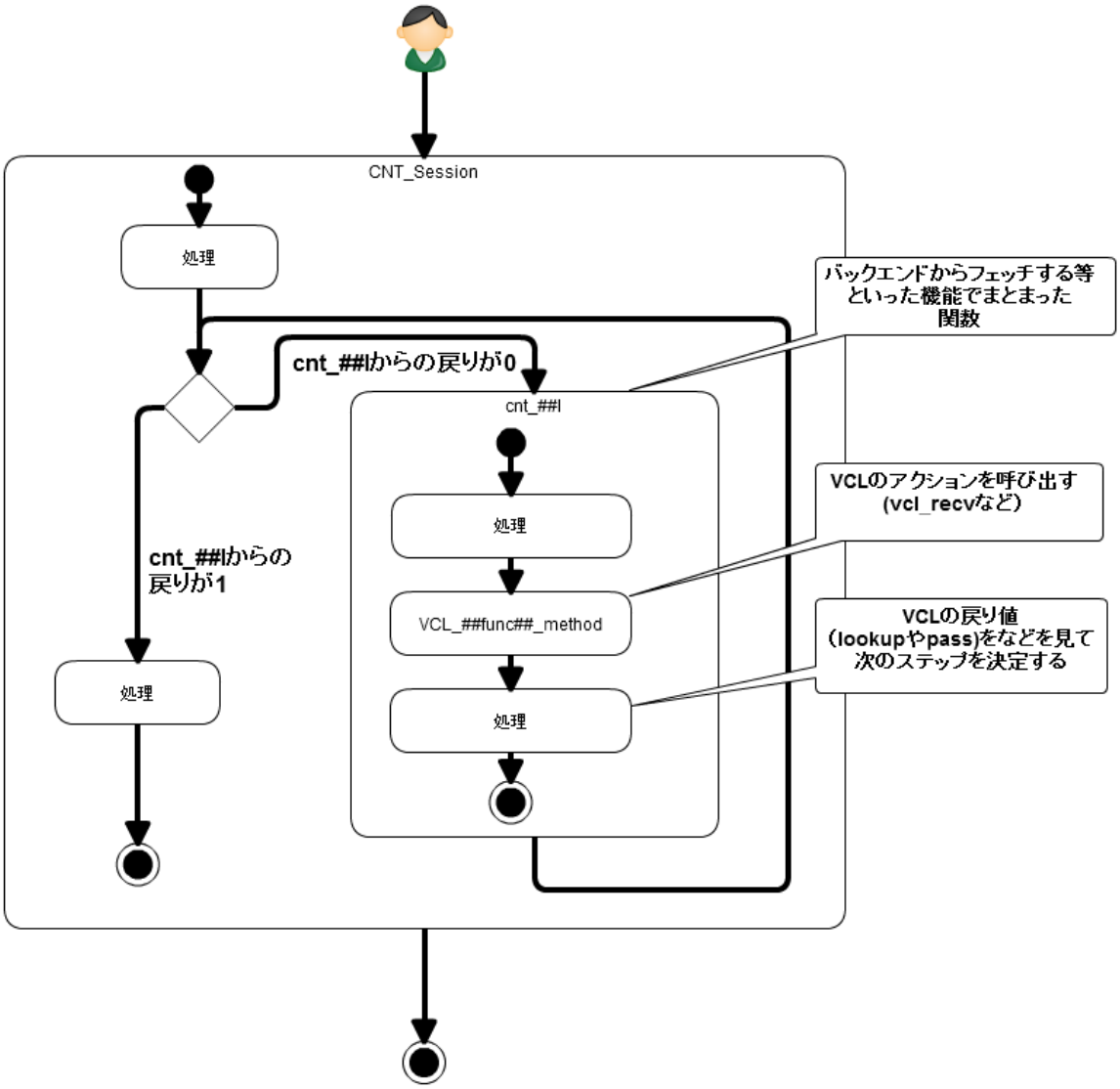
req.httpは実際使う際、req.http.hostといったようにヘッダのフィールド名まで指定します。その際に利用する関数はVRT_GetHdrとVRT_SetHdrで個々では定義されていません。この関数はreq.http・bereq.httpなど全てで共通で利用されます。そのためどのヘッダを読み書きするかを指定する必要があります。
上記で太字にしているHDR_REQがそれに当たり、これを引数に指定します。
詳しくは後述します。

/bin/varnishd/mgt_param.c

正確にはインラインCとはあまり関係ないのですが、非常に重要なファイルの一つなので解説します。
このファイルは、Varnishの起動パラメータのデフォルト値・最大最小値や説明が格納されています。
基本的に、パラメータの一覧を知りたければ管理コンソールにつないで
「param.show -l」を行えばよいです。が、このファイルの使い道はは
バージョンが上がった際に、このファイルをdiffをすることで起動パラメータの変更を調べるのに利用しています。
また同じ理由でgenerate.pyもdiffすることで変数の変更がわかります。

/bin/varnishd/cache_center.c

セッションを開始してから、レスポンスするまでの一連の流れが記述されています。
このファイルを見れば、たいていの Varnish の動きは把握できます。
単純なインライン C を扱う場合はあまり意識をする必要はありませんが、Varnish をより深く知るためには避けられないファイルです。
例えば vcl_hash はどのタイミングで呼び出されるでしょうか？バックエンドへのフェッチはどのタイミングで？そのような処理が全てまとまっています。
下図を参照してください。



おおまかに言うと、Varnish がリクエストを処理する際 CNT_Session を起点として、機能でまとまったステップを呼んでいき処理していきます。

例えばVCLで一番最初に処理されるvcl_recvに行き着くまでは次のような経路を辿ります。

1. CNT_Session
2. cnt_wait
3. cnt_start
4. cnt_recv
    1. VCL_recv_method

特に重要なのがcnt_recvやcnt_fetchなどのVCLの各アクションが呼び出される関数です。例えばcnt_fetchを見てみましょう。

```
static int
cnt_fetch(struct sess sp)
{
        /*中略*/
        http_Setup(sp->wrk->beresp, sp->wrk->ws);

        i = FetchHdr(sp);
        /*中略*/
        if (i == 1) {
                VSC_C_main->backend_retry++;
                i = FetchHdr(sp);
        }

        if (i) {
                sp->handling = VCL_RET_ERROR;
                sp->err_code = 503;
        } else {
                /*中略*/
                VCL_fetch_method(sp);

                switch (sp->handling) {
                case VCL_RET_HIT_FOR_PASS:
                        if (sp->objcore != NULL)
                                sp->objcore->flags |= OC_F_PASS;
                        sp->step = STP_FETCHBODY;
                        return (0);
                case VCL_RET_DELIVER:
                        AssertObjCorePassOrBusy(sp->objcore);
                        sp->step = STP_FETCHBODY;
                        return (0);
                default:
                        break;
                }
                /*中略*/
        }

        /*後略*/
```

例えば、バックエンドからヘッダを取得しているFetchHdrですが、取得に失敗した場合一回だけリトライしています。
リトライも失敗した場合、ステータス503としてVCL_RET_ERRORを返します。
これはVCL内でreturn(error)とした際と同じ値です。
ここで気づいた方もいるかも知れませんが、サーバが明示的に503を返すのとサーバ自体に接続できない場合は動きが違うことがわかります。
vcl_fetchの呼び出し関数、VCL_fetch_methodはヘッダの取得に成功した時しか呼ばれないためサーバに接続できない場合はvcl_fetchは呼び出されません。
このように細かい動きを知るためにcache_center.cを読むのは必要です。

処理を追っかけていく場合、VCL_recv_methodのように各VCLのアクションが呼び出される前後を見ると追いやすいかと思います。

以上がインラインCを行う上で是非見て欲しいファイルです。
もちろんインラインCに限りません。これらのファイルを起点に調べていけばバージョンが上がった場合でも、すぐに把握できるかと思います。
次では、実際にインラインCを使う上で必要最低限な関数や注意事項を説明します。

インラインCで変数操作を行う方法

インラインCでVCLでの変数(req.urlなど)を読み書きするには、ほんの少し工夫が必要です。
各変数に、getter/setterが用意されており、その関数を利用して値の取得・設定を行います。それぞれのやり方について解説します。

各変数に読み込みする方法

基本的にはソースを読むで解説したvcc_obj.cに書いているのですが、すべて覚えるのは難しいです。しかし、法則性がありますので解説します。

HEADER型を除く読み込み

変数名	beresp.backend.ip
C関数名	VRT_r_***beresp_backend_ip***(sp);

インラインCでVCLの変数を読み込む場合は全て関数を使います。
「VRT_r_」をVCLの変数名の頭につけて「.」を「_」に置換すると読み込み関数名になります。なお、第一引数のspは後で説明しますが、今後現れる関数も含めてspはそのまま指定してください。

戻り値は各変数の型によって違います。以下がそのリストです。

VCLでの型名	Cでの型名
BACKEND	struct director *
BOOL	unsigned
DURATION	double
INT	int
IP	struct sockaddr_storage *
STRING	const char *
TIME	double

それぞれの方について解説します。

BACKEND / *struct director* *

バックエンドの情報を格納している型です。
ただし、この構造体のメンバにアクセスするには様々なヘッダを include する必要があります。そのためインライン C では、どのバックエンドが選択されているかを文字列で取得するのに使うのが一般的と思われます。

```
■req.backend に設定されているバックエンドの名前を取得する。
const char * c = VRT_backend_string(sp,VRT_r_req_backend(sp));
```

BOOL / *unsigned*

真偽が入っている型です。

DURATION / *double*

時間を浮動小数点型で格納している型です。
格納の単位は秒です。例として beresp.ttl で見てみましょう。

```
sub vcl_fetch{
    set beresp.ttl = 60m;
    C{
        char str[64];
        snprintf(str,64,"beresp.ttl=%.3f",VRT_r_beresp_ttl(sp));
        syslog(LOG_INFO,str);
    }C
}
```

とした場合

```
beresp.ttl=3600.000
```

といった出力が得られます。

INT / *int*

整数が入っている型です。

IP / *struct sockaddr_storage **

IP アドレスを格納している型です。
BACKEND 型と同様にメンバにアクセスするには様々なヘッダを include する必要があります。そのためインライン C では IP アドレスを文字列で取得するのが一般的と思われます。

■client.ip に設定されている IP アドレスを取得

```
const char *ip = VRT_IP_string(sp,VRT_r_client_ip(sp));
```

STRING / *const char **

文字列を格納しています。

TIME / *double*

時間を格納しています。
double ですが、実際のところは time_t なので以下のような使い方が可能です。
now 変数で試して見ました。

```
C{
    char str[64];
    time_t t=(time_t)VRT_r_now(sp);
    struct tm *ptime = localtime( &t );
    snprintf(str,64,"year=%d",ptime->tm_year+1900);
    syslog(LOG_INFO,str);
}C
```

とした場合

```
year=2011
```

といった出力が得られます。
また簡単に時間の文字列を取得したい場合は VRT_time_string という関数が用意されています。

```
C{
    syslog(LOG_INFO,VRT_time_string(sp,VRT_r_now(sp)));
}C
```

とした場合

```
Sun, 11 Dec 2011 16:37:21 GMT
```

と出力されます。フォーマットは「%a, %d %b %Y %T GMT」です。

以上が VCL で利用している変数の型一覧です。

また本文中で紹介した以外に各型から文字列に変換する関数があるので一覧で紹介します。

変換元の型	関数名	出力例
IP	VRT_IP_string	192.168.1.1
INT	VRT_int_string	123
BYTES	VRT_double_string	3600.00
DURATION		
TIME	VRT_time_string	Sun, 11 Dec 2011 16:37:21 GMT
BOOL	VRT_bool_string	true
BACKEND	VRT_backend_string	default

HEADER 型の読み込み

変数名　　resp.http.Expires
C 関数名　　VRT_GetHdr(sp,***HDR_RESP,"¥010Expires:"***);

HEADER 型は要素の数が可変なため、それぞれの要素ごとに今までの INT 型などのように、決まった関数は存在しません。全て VRT_GetHdr を利用します。
第二引数にはどこのヘッダを参照するかを定数で指定します。以下が一覧です。

定数名	対応するHEADER型変数	説明
HDR_REQ	req.http.*	クライアントからのリクエストヘッダを格納
HDR_RESP	resp.http.*	クライアントへのレスポンスヘッダを格納
HDR_OBJ	obj.http.*	キャッシュオブジェクトのヘッダを格納
HDR_BEREQ	bereq.http.*	バックエンドへのリクエストヘッダを格納
HDR_BERESP	beresp.http.*	バックエンドからのレスポンスヘッダを格納

第三引数にはフィールド名を指定します。この際の指定方法は注意が必要です。

null 文字 + フィールド名の長さを8進数で2桁 + フィールド名(末尾に:)

特に文字列長の指定が**8進数で2桁**というのに注意が必要です。
もし例えば req.http.X に対してアクセスする場合の指定は以下のようになります。

VRT_GetHdr(sp,***HDR_REQ,"¥002X:"***);

アクセスしたいフィールド名は「X」で一文字ですが、「:」が追加されるため実質2文字なのに注意しましょう。

各変数に書き込みする方法

基本的には読み込みと似たような感じですが、文字列の扱いに注意が必要です。
なお IP と TIME 型は書き込み可能な変数が存在しません。

HEADER・STRING 型を除く書き込み

VCL	set bereq.connect_timeout = 1m;
C 関数名	VRT_l_***bereq_connect_timeout***(sp,***60***);

関数名は「VRT_l_」から始まり、読み込みと同様に変数名の「.」を「_」にして結合したものになります。当然ですが第二引数の部分は操作しようとしている変数の型によって変わります。
それぞれについて解説します。

BACKEND / *struct director* *

バックエンドの指定です。文字列で「client」のように指定できればいいのですが、そのようにはできません。以下のように指定します。

■バックエンドの定義
backend **client**{.host="192.168.1.199";.port="81";}

■req.backend に client を指定
VRT_l_req_backend(sp, VGCDIR(**_client**));

VGCDIR はマクロです。バックエンド名を「client」にした場合は「_」をつけて「_client」と指定します。

BOOL / *unsigned*

真偽値を指定します。好みの問題かもしれませんが、VCL がコンパイルする際は以下のように指定しています。

■true
VRT_l_req_esi(sp, **(0==0)**);

■false
VRT_l_req_esi(sp, **(0==1)**);

DURATION / *double*

時間の指定です。
全て秒単位です。

INT / *int*
整数の指定です。

STRING 型の書き込み

VCL	set resp.response = “A” + “B”;
C 関数名	VRT_l_***resp_response***(sp,***”A”,”B” ,vrt_magic_string_end***);

第2引数以降は可変長となっており、複数の文字列を指定すると順番に結合されます。
また必ず最後に **vrt_magic_string_end** を指定します。忘れてもエラーにならない上に挙動がおかしくなるので絶対忘れないで下さい。

HEADER 型の書き込み

VCL	set resp.http.X = “A”+”B”;
C 関数名	VRT_SetHdr(sp,***HDR_RESP,”¥002X:”, ”A”,”B”,vrt_magic_string_end***);

第3引数までは読み込みの際と同じで、残りは STRING 型の書き込みの仕方と同様です。複数指定した文字列は結合されます。最後に **vrt_magic_string_end** を指定してください。
またフィールド自体を削除したい場合は0のみを指定します。

VCL	remove resp.http.X;
C 関数名	VRT_SetHdr(sp,*HDR_RESP,”¥002X:”,**0***);

削除する場合は **vrt_magic_string_end は必要ありません。**

struct sess *sp について

変数の読み書きを行う関数の第一引数には、必ず「sp」を指定されています。
この変数は、セッションの状態を保持しています。
例えば、現在実行中の VCL のメソッド(fetch など）やオブジェクトの格納場所など様々な情報が格納されています。
そのため頑張れば、オブジェクトの Body 部に対してへのアクセスなど、
通常できない操作が可能です。しかし、ヘッダの include などが非常に煩雑なため操作をする場合は VMOD で行うべきでしょう。
定義は/bin/varnishd/cache.h に存在します。

組み込み関数を使う

VCL には ban や hash_data のような様々な組み込み関数が存在します。
以下に一覧とインライン C で呼び出す際の方法を説明します。

ban

指定された正規表現を Ban リストに追加します。

```
VCL          ban("req.http.host == " + req.http.host
                 + "&& req.url == " + req.url);
インラインC  VRT_ban_string(sp, VRT_WrkString(sp,
                 "req.http.host == ",
                 VRT_GetHdr(sp, HDR_REQ, "¥005host:"),
                 "&& req.url == ",
                 VRT_r_req_url(sp),
                 vrt_magic_string_end));
```

ban は VRT_ban_string となりますが、一点注意が必要です。この関数自体が複数のテキストを許容しないことです。そのため事前にテキストを組み立てる必要があります。
その際利用するのが VRT_WrkString です。この関数はワークスペースを利用して（後述します）テキストを組み立てます。これも今まで複数のテキストを扱っていた時と同様、必ず末尾に vrt_magic_string_end を指定します。

ban_url

指定されたURLをBanリストに追加します。

```
VCL           ban_url(req.url);
インラインC  VRT_ban(sp, "req.url", "~",
                    VRT_r_req_url(sp), 0);
```

この関数の引数は可変長ですが、コードを見るかぎり実質次の通りになります。

```
VRT_ban(sp, "評価対象", "演算子", “評価対象”, 0);
```

またこの関数の最後の引数はvrt_magic_string_endではなく0なので注意しましょう。

call

ユーザが定義したサブ関数を呼びます

```
VCL           call inlineTest;
インラインC  if (VGC_function_inlineTest(sp))
                    return (1);
```

定義した関数は、VGC_function_##定義名##となります。

hash_data

オブジェクトの特定・格納に利用するハッシュの定義に追加します。

```
VCL           hash_data(req.url+"_pc");
インラインC  VRT_hashdata(sp,VRT_r_req_url(sp),"_pc",
                    vrt_magic_string_end);
```

この関数も可変長の引数なので、末尾に vrt_magic_string_end を指定します。

panic

指定されたメッセージを出力して、現在の子プロセスを殺します。

```
VCL          panic("ng" + req.url);
インラインC  VRT_panic(sp,(
                        VRT_WrkString(sp,
                        "ng",VRT_r_req_url(sp),
                        vrt_magic_string_end)
                    ),vrt_magic_string_end);
```

この関数の引数も可変長です。が、実際内部で利用する引数は可変長部の一つ目だけなので、VRT_WrkString で結合が必要です。
もちろん VRT_WrkString・VRT_panic 両方共 vrt_magic_string_end が必要なことに注意してください。

purge

現在選択されているオブジェクトを即座に削除します。

```
VCL          purge;
インラインC  VRT_purge(sp,0,0);
```

return

関数をリターンします。

```
VCL          return(deliver);
インラインC  VRT_done(sp, VCL_RET_DELIVER);
```

引数で指定する deliver などは全て大文字にした後 Prefix に「VCL_RET_」をつけます。

synthetic

vcl_error などで使うレスポンスボディを作成します。

```
VCL             synthetic "url"+req.url;
インラインＣ  VRT_synth_page(sp, 0,"url",VRT_r_req_url(sp),
                        vrt_magic_string_end);
```

この関数も可変長の引数なので末尾に vrt_magic_string_end を指定します。
また VCL の関数は vcl_error でのみ有効ですが、インラインＣの場合
vcl_deliver でも動作することを確認しています。

rollback

req.*変数を初期化します。

```
VCL             rollback;
インラインＣ  VRT_Rollback(sp);
```

error

ステータスコードとメッセージを指定して、vcl_error に遷移します。

```
VCL             error(404 ,"NotFound.");
インラインＣ  VRT_error(sp, 404, "NotFound.");
```

この関数は複数のテキストを許容しないため、複数の文字列を組み立てたい場合は
VRT_WrkString を利用します。

以上です。
正規表現(regsub,regsuball)はインラインＣで使うことが事実上難しいため省略します。VMOD の箇所で解説しています。

インライン C で外部の共有ライブラリを利用する場合

libmemcached や libxml2 といった共有ライブラリを利用する場合、本来は VMOD を利用すべきです。しかしどうしてもインライン C で利用したい場合、起動パラメータの cc_command を変更することで共有ライブラリを呼び出すことが可能になります。
cc_command は Varnish が VCL をコンパイルする際に利用するコマンドになります。今回は libmemcached を例に説明します。

まずは現在のパラメータを確認します。

```
[root@localhost ~]# varnishadm  param.show cc_command
cc_command              "exec gcc -std=gnu99 -O2 -g -pipe -Wall -Wp,-
D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector --param=ssp-buffer-
size=4 -m64 -mtune=generic -pthread -fpic -shared -Wl,-x -o %o %s"
～後略～
```

デフォルトのパラメータは環境によって違うため必ず確認してください。
確認ができたら起動パラメータに**-lmemcached** 追加します。

```
DAEMON_OPTS="-a ${VARNISH_LISTEN_ADDRESS}:${VARNISH_LISTEN_PORT}
¥
-i testsv ¥
        -f ${VARNISH_VCL_CONF} ¥
        -T ${VARNISH_ADMIN_LISTEN_ADDRESS}:$
{VARNISH_ADMIN_LISTEN_PORT} ¥
        -t ${VARNISH_TTL} ¥
        -w ${VARNISH_MIN_THREADS},${VARNISH_MAX_THREADS},$
{VARNISH_THREAD_TIMEOUT} ¥
        -u varnish -g varnish ¥
-p cc_command='exec gcc -std=gnu99 -O2 -g -pipe -Wall -Wp,-
D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector --param=ssp-buffer-size=4
-m64 -mtune=generic -pthread -fpic -shared -Wl,-x -lmemcached -o %o %s' ¥
"
```

スペースが入っているため「'」などで囲むのを忘れないで下さい。

今回は、サブ関数の mcSet を呼び出すと req.http.X-mck に格納された文字列をキーとして扱い req.http.X-mcv を値として memcache に格納するコードを記述します。

```
C{
#include <stdlib.h>
#include <stdio.h>
#include <libmemcached/memcached.h>
void mctest(char *k ,char *v){
        struct memcached_st             *mmc            = NULL;
        struct memcached_server_st   *servers       = NULL;
        memcached_return                rc;
        mmc      = memcached_create(NULL);
         servers = memcached_server_list_append(servers,"localhost", 11211, &rc);
        rc          = memcached_server_push(mmc, servers);
        memcached_server_list_free(servers);
        rc         = memcached_set(mmc, k, strlen(k), v, strlen(v), 600, 0);
        memcached_free(mmc);
}

}C

sub mcSet{
            if(req.http.X-mck && req.http.X-mcv){
                        C{
                                    char *key=VRT_GetHdr(sp,HDR_REQ,"¥006X-mck:");
                                    char *value=VRT_GetHdr(sp,HDR_REQ,"¥006X-mcv:");
                                    mctest(key,value);
                        }C
            }
            remove req.http.X-mck;
            remove req.http.X-mcv;
}
sub vcl_recv{
            set req.http.X-mck = "Last:req.xid";
            set req.http.X-mcv = req.xid;
            call mcSet;
～後略～
```

実際に telnet で memcache に接続し値を取得してみました。

```
[root@localhost libmemcached-1.0.2]# telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
get Last:req.xid
VALUE Last:req.xid 0 10
1938831702
END
```

実際に値が設定されていることを確認できます。

インライン C で共有ライブラリを利用する際に注意が必要なのが、デバッグする際も cc_command を指定する必要が有ることです。

指定しない場合、当然ながら undefined symbol が出て実行できません。

VMOD を使う際に必要な基礎知識

インライン C で大規模なコードを書こうとすると以下のような悩みがでてきます。

・VCL 中にインライン C が混在して見づらい
・変数の受け渡しに HEADER 変数を使ったりと変則的

他にも様々な悩みが出てくるでしょう。
コードの書き方にもよりますが、インライン C で書いたコードは多少再利用しづらいと考えています。
そこで出てくるのが VMOD です。
VMOD は Apache や Nginx のモジュールのように配布しやすく使いやすいです。
まずは配布されている VMOD を入れてみて感覚を掴みましょう。

外部の *VMOD* を使ってみる*(vmod_example)*

まずは Varnish 公式が配布している vmod_example をダウンロードして使ってみましょう。

```
https://github.com/varnish/libvmod-example
```

このモジュールは HelloWorld を出力するだけの単純なものです。
筆者の環境では以下のようにして導入しました。

```
[root@localhost example]# wget http://repo.varnish-cache.org/source/varnish-
3.0.2.tar.gz
[root@localhost example]# tar zxf varnish-3.0.2.tar.gz
[root@localhost example]# cd varnish-3.0.2
[root@localhost varnish-3.0.2]# ./configure
[root@localhost varnish-3.0.2]# make
[root@localhost varnish-3.0.2]# cd ..
[root@localhost example]# git clone https://github.com/varnish/libvmod-
example.git
[root@localhost example]# cd libvmod-example/
[root@localhost libvmod-example]# ./autogen.sh
[root@localhost libvmod-example]# ./configure
VARNISHSRC=~/example/varnish-3.0.2                    ・・・(1)
[root@localhost libvmod-example]# make
[root@localhost libvmod-example]# make check          ・・・(2)
[root@localhost libvmod-example]# make install        ・・・(3)
```

注意が必要な箇所はまず configure で Varnish のソースディレクトリを指定しなくてはいけません。

単純にソースだけで良ければ、varnish-debuginfo をインストールしてその箇所を指定すれば問題ありません。しかし、コンパイルされている varnishtest が必要なため、同じバージョンの varnish のソースを make しています。
make install までは必要ありません。
また make 後のテストは make check で行います。
make install を行うと、デフォルトの VMOD のインストール場所にコピーされます。筆者の環境では /usr/lib64/varnish/vmods/ でした。

次は早速 VCL から使ってみましょう。
以下のような VCL を書きます。

```
import example;
sub vcl_deliver{
        set resp.http.hello = example.hello("World");
}
```

この状態でリクエストしてみるとレスポンスヘッダに

```
hello: Hello, World
```

が付与されます。

vmod に関数を追加してみる

先ほどの vmod_example の構造を見ていきます。以下がファイルツリーです。

```
.
├── autogen.sh
├── configure.ac
├── LICENSE
├── m4
│   └── PLACEHOLDER
├── Makefile.am
├── README.rst
└── src
    ├── Makefile.am
    ├── tests
    │   └── test01.vtc
    ├── vmod_example.c
    └── vmod_example.vcc

3 directories, 10 files
```

1から作るのもいいのですが、手間なので vmod_example をベースに編集していきます。

その際に必ず編集が必要なファイルは以下です。

```
src/vmod_example.c
src/vmod_example.vcc
```

まずは簡単な関数を一つ追加してみましょう。
名前は len で、文字列の長さを返却します。
最初に vmod_example.vcc を以下のように修正します。

```
～中略～
Function STRING hello(STRING)
Function INT len(STRING)
```

次に vmod_example.c を以下のように修正します。

```
～中略～
int vmod_len(struct sess *sp, const char *p)
{
        return(strlen(p));
}
```

早速 make して使ってみましょう。VCL は以下です

```
set resp.http.len = example.len("Hello World!!");
```

この状態でリクエストしてみるとレスポンスヘッダに

```
len: 13
```

が付与されます。

非常に簡単に関数を追加できることがわかった思います。
次からはより実際に使う内容について解説します。

vmod_example.vcc (vmod.vcc)

VMOD を VCL コンパイラと VCL から呼び出すためのインタフェースを定義します。
以下の三要素があります

Module [module 名]	この VMOD の名前空間を示します。
Init [関数名]	VMOD の初期化関数です。
Function [返却型] [関数名](引数の型)	VCL から呼び出す関数です。

また先頭が「#」の場合コメントとして扱われます。「//」や「/*～*/」はエラーとなりますので注意してください。
それぞれについて解説します。

Module [module名]

Module名を定義します。この名前は他のモジュールと重複してはいけません。
以下のように定義します。

```
Module example
```

Init [関数名]

VCLが読み込まれる際に呼ばれるVMODの初期化関数です。
予め初期化しておく必要があるテーブル等を初期化するのに使用します。
以下のように定義します。

```
Init init_function
```

Initに対する解放処理がありませんが、これは後述するVMODで有効なワークスペース・プライベートポインタを利用することで解決できます。

Function [返却型] [関数名](引数の型)

VCLから呼ばれる関数です。各型はC言語の型ではなくVCLでの型となります。
以下のように定義します。関数名は半角英数字小文字のみ許容されます。

```
■戻り値有り
Function STRING hogehoge(INT,STRING)

■戻り値無し
Function VOID hogehoge(INT,STRING)
```

変数の型は後述します。

vmod_example.c (vmod.c)

実際のVMODのコードです。
必ず以下のヘッダをincludeする必要があります。

```
#include "vcc_if.h"
```

また関数名はvccで定義した名前の頭にvmod_がつきます。

```
■VCCでの名前
hello

■Cでの名前
vmod_hello
```

またインラインCで扱った関数と同様に、必ず第一引数はspとなります。

```
int vmod_len(struct sess *sp, const char *p){
        return(strlen(p));
}
```

第二引数以降は受け取る変数によって変わってきます。
init_functionについては後述します。

*VMOD*で使用可能な変数の型

VMODで利用可能な型はほぼVCLと同じです。しかし一部非推奨だっり特殊な型があったりします。

VCLでの型名	Cでの型名	戻値	引数	備考
BACKEND	struct director *	○	○	記述なし
BOOL	unsigned	○	○	非推奨
DURATION	double	○	○	
INT	int	○	○	
IP	struct sockaddr_storage *	△	○	非推奨
STRING	const char *	○	○	
TIME	double	○	○	
HEADER	enum gethdr_e, const char *	×	○	非推奨
REAL	double	○	○	
STRING_LIST	const char *, ...	×	○	
PRIV_VCL	struct vmod_priv *	△	○	
PRIV_CALL	struct vmod_priv *	△	○	
VOID	void	○	×	

戻値が△なのは、書き込みできる変数が存在せず、使い道がさほど思いつかなかった為です。また非推奨のものは公式のドキュメントに記載されていたものです。
それぞれの変数を引数・戻値に持つ簡単な関数を作って解説します。

BACKEND

バックエンドの情報を格納しています。引数、戻り値共に指定可能です。

■vcc

```
Function BACKEND tbackend(BACKEND)
```

■c

```
struct director * vmod_tbackend(struct sess *sp, struct director *p){
        return p;
}
```

■VCL

```
set req.backend = example.tbackend(req.backend);
```

もしdirectorのメンバーにアクセスする場合は以下のヘッダをincludeする必要があります。

```
#include "bin/varnishd/cache.h"
#include "bin/varnishd/cache_backend.h"
```

例として指定したバックエンドが正常の場合はそれを返却し、そうでない場合は現在選択されているバックエンドを返却します。

```
■vcc
Function BACKEND gethealthydirector(BACKEND)

■c
struct director * vmod_gethealthydirector(
struct sess *sp, struct director *p){
        if(VDI_Healthy(p, sp)){
                return p;
        }
        return sp->director;
}

■VCL
set req.backend = example.gethealthydirector(client_2);
```

VDI_Healthyは、バックエンドの状態を返却します。
利用するには以下のヘッダをincludeする必要があります。

```
#include "bin/varnishd/cache.h"
```

BOOL

真偽が入っている型です。

```
■vcc
Function BOOL tbool(BOOL)

■c
unsigned vmod_tbool(struct sess *sp, unsigned p){
        return p;
}

■VCL
set req.esi = example.tbool(req.esi);
```

DURARATION

時間を浮動小数点型で格納している型です。

■vcc

```
Function DURATION tduration(DURATION)
```

■c

```
double vmod_tduration(struct sess *sp, double p){
        return p;
}
```

■VCL

```
set beresp.ttl = example.tduration(10m);
```

INT

整数が格納されている型です。

■vcc

```
Function INT tint(INT)
```

■c

```
int vmod_tint(struct sess *sp, int p){
        return p;
}
```

■VCL

```
set beresp.status = example.tint(200);
```

IP

IP アドレスを格納されている型です。

■vcc

```
Function INT tip(IP)
```

■c

```
int vmod_tip(struct sess *sp, struct sockaddr_storage * p){
        if (p->ss_family == AF_INET)  {return 4;}
        if (p->ss_family == AF_INET6) {return 6;}
        return 0;
}
```

■VCL

```
set resp.http.iptype = example.tip(client.ip);
```

sockeaddr_storage の要素にアクセスするには、以下のヘッダを include する必要があります。

```
#include "sys/socket.h"
```

STRING

文字列が格納されている型です。

```
■vcc
Function STRING tstring(STRING)

■c
const char* vmod_tstring(struct sess *sp, const char* p){
        return p;
}

■VCL
set resp.http.str = example.tstring(“abc”);
```

VCL で文字列の結合が必要になった場合 VRT_WrkString を補完します。

STRING_LIST

引数でのみ利用可能な、複数の文字列が指定可能な型です。

```
■vcc
Function STRING tstring_list(STRING_LIST)

■c
const char* vmod_tstring_list(struct sess *sp, const char*p,...){
        va_list ap; char *b;
        va_start(ap, p);
        b = VRT_String(sp->wrk->ws, NULL, p, ap);
        va_end(ap);
        return(b);
}

■VCL
set resp.http.str = example.tstring_list(“abc”,”aaa”);
```

ここで利用している VRT_String は、ワークスペースを利用して文字を結合して一つにまとめる関数です。利用するには以下の include する必要があります。

```
#include "bin/varnishd/cache.h"
```

ワークスペースについては後述します。

HEADER

ヘッダーが格納されている型です。

■VCC

```
Function STRING theader(HEADER)
```

■C

```
const char* vmod_theader(struct sess *sp, enum gethdr_e e, const char *p){
        switch(e){
                case HDR_REQ:
                        return(“req”);
                        break;
                case HDR_RESP:
                        return(“resp”);
                        break;
                case HDR_OBJ:
                        return(“obj”);
                        break;
                case HDR_BEREQ:
                        return(“bereq”);
                        break;
                case HDR_BERESP:
                        return(“beresp”);
                        break;
        }
        return “”;
}
```

■VCL

```
set resp.http.test = example.theader(req.http.x);
```

「enum gethdr_e e」にはどこのヘッダかが含まれます。

また「const char *p」はフィールド名が「:」付きで入っています。

REAL

浮動小数点が格納されている型です。

DURARATION は時間を表すのに対し、REAL は単純に浮動小数点数を表します。

```
■vcc
Function REAL treal(REAL)

■c
double vmod_treal(struct sess *sp, double p){
        return p+0.1;
}

■VCL
if(example.treal(0.5)>0.5)
```

TIME

時間が格納されている型です。

```
■vcc
Function TIME ttime(TIME)

■c
double vmod_ttime(struct sess *sp, double p){
        return p;
}

■VCL
set resp.http.time = example.ttime(now);
```

例として指定された時間を加える関数を作ってみます。

```
■vcc
Function TIME timeoffset(TIME,DURATION,BOOL)

■c
double vmod_timeoffset(struct sess *sp, double time ,double os ,unsigned rev){
        if(rev){ os*=-1; }
        return time+os;
}

■VCL
set resp.http.time = example.timeoffset(now,1h,false);
```

第三引数が true になると、時間に対してマイナスします。

VOID

戻値がない場合に指定する型です。

```
■vcc
Function VOID tvoid()

■c
void vmod_tvoid(struct sess *sp){
        return;
}

■VCL
example.tvoid();
```

PRIV_VCL

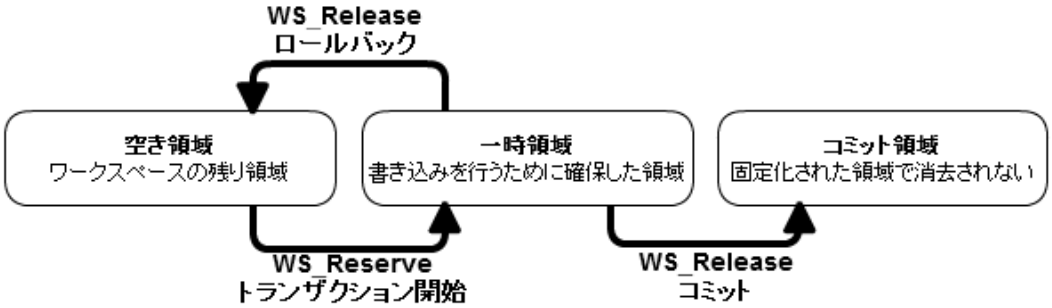
VMOD 内で有効な、プライベートポインタを示す特殊な型です。
後述します

PRIV_CALL

VMOD 内の呼び出し関数で有効なプライベートポインタを示す特殊な型です。
後述します。

セッションワークスペースの使い方

Varnish では、セッション毎にワークスペースを持っています。VMOD では主に
戻り値が文字列で、メモリを確保する必要がある場合に使用します。
ここからメモリを確保すれば、Varnish 側で制御してくれるのでメモリリークの
心配がありません。デフォルトは64 KB が確保されており、起動パラメータの
sess_workspace でサイズを変更できます。
64 KB と聞くと十分なサイズと思われます。しかし、例えばクライアントが
リクエストしてきた際の生データも格納されていたり、と他でも使用されているため
不必要なら確保したものも開放しましょう。
ワークスペースの領域の状態は3つあります。



一時領域は、使い終わった時点で必ずその領域をロールバックかコミットを行う必要があります。一時領域を作れるのは一つまでのため、どちらもせずに再度
トランザクション開始するとエラーとなります。
では実際に使ってみましょう。
include が必要なヘッダと使用する関数は以下です

```
#include "bin/varnishd/cache.h"
```

トランザクションの開始（ワークスペースの空き領域予約）

■関数

unsigned WS_Reserve(struct ws *ws, unsigned bytes);

■引数

struct ws *ws	ワークスペースを指定
unsigned bytes	確保したいバイトを指定、基本的に0指定（＝残り全て）

■戻値

確保できたバイト数

領域のコミット・ロールバック処理(ワークスペースの使用領域を確定)

■関数

void WS_Release(struct ws *ws, unsigned bytes);

■引数

struct ws *ws	ワークスペースを指定
unsigned bytes	**コミットする**バイト数

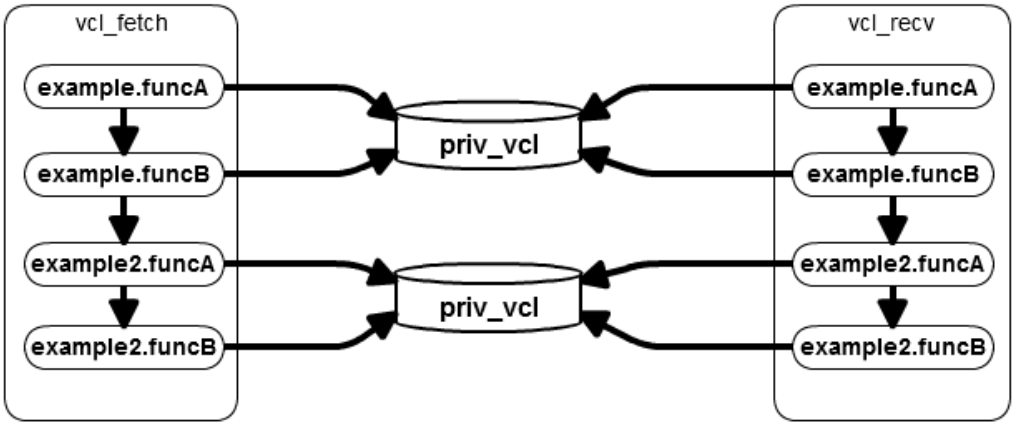
例として10バイト確保するコードを記述します。
仮に10バイト確保できない場合、開放して NULL を返します。

```
u = WS_Reserve(sp->wrk->ws, 0);
if(u<10){
  WS_Release(sp->wrk->ws,0); //確保可能な領域が10バイト以下なので処理を終了
  return NULL;
}
char * str = (char*)sp->wrk->ws->f; //空き領域のポインタを指定
～処理～
WS_Release(sp->wrk->ws,10); //10 バイトコミットする
```

プライベートポインタの使い方

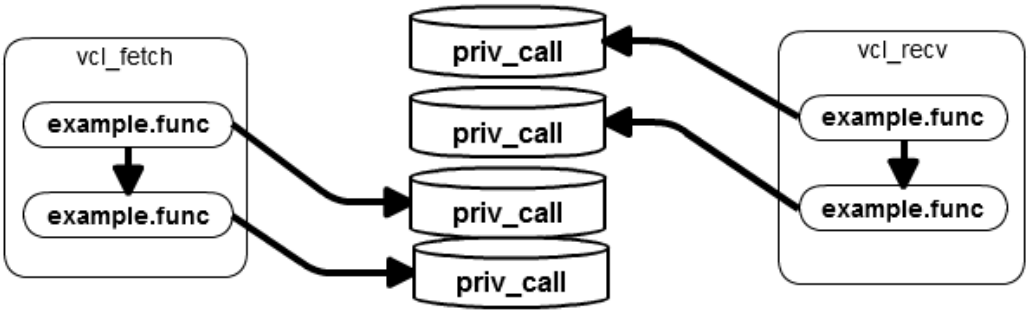
先ほどのセッションワークスペースは、セッションが開始されるたびにクリアされてしまいます。たとえば、正規表現のコンパイル等、高コストの処理を一度だけ呼び出して、そのあとは使いまわしたいときはどうすればよいでしょうか？
Varnish ではプライベートポインタというものを用意しています。
これは違うセッションでも設定したテーブルなどを保持できる機構です。

プライベートポインタには二種類存在します。
VMOD 内で有効な priv_vcl です。以下の図を参照してください。



プライベートポインタはそれぞれの VMOD 毎に割り当てられます。
そのため recv で設定した値を fetch で参照することも可能です。

そしてもう一つは priv_call です。以下の図を参照してください。

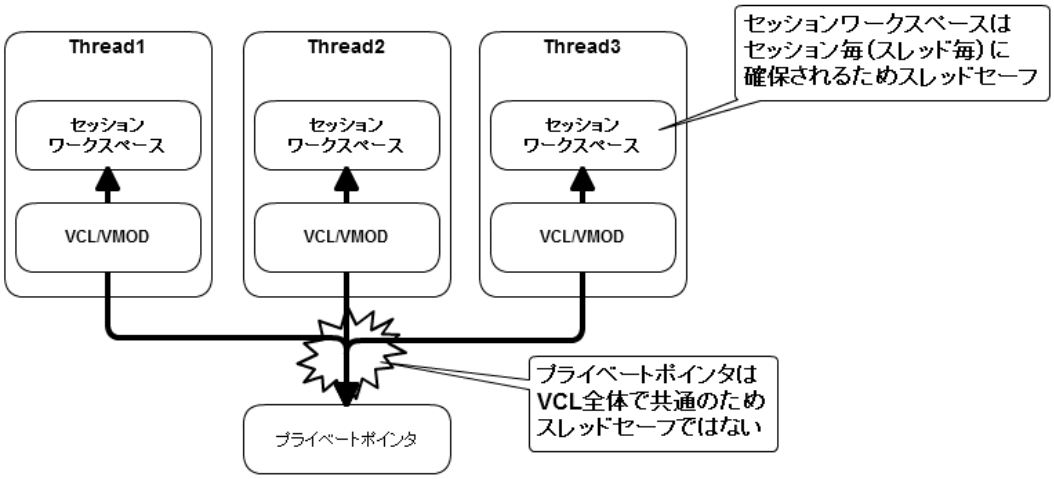


これは関数の呼び出し毎にプライベートポインタを割り当てるものです。
同じ関数でも、別々のポインタが割り当てられる事に注意してください。
設定した値は次のセッションで参照可能です。

次に実際に使う場合のコードを説明します。

スレッドセーフであることの必要性

Varnish は多数のスレッドが同時に動きます。セッションワークスペースの場合はスレッド毎に確保されるため、特にマルチスレッドで動いていることを意識せずとも問題有りません。
しかしプライベートポインタを使う場合は別です。以下の図を参照してください。



もしアクセス毎に共通のカウンタをインクリメントするプログラムを書いた場合はどうでしょうか？以下のような現象が起きます。

1.スレッドＡがプライベートポインタから**「1」**を取得
2.スレッドＢがプライベートポインタから**「1」**を取得
3.スレッドＡがプライベートポインタに　**「2」**を書き込み
4.スレッドＢがプライベートポインタに　**「2」**を書き込み

二回書き込んだのにポインタの中身は**「3」**にならずに**「2」**になります。
そのためプライベートポインタを使う場合、スレッドセーフであることを意識する必要があります。
スレッドセーフにするためには2つ方法があります。

ロック

特定のリソースに複数のスレッドが操作を行うと
破綻をきたす処理（＝クリティカルセクション）に対して1つの
スレッドのみ処理ができるようにする仕組み。

ロックフリー

特定のリソースに複数のスレッドが操作を行なっても
破綻しないようにすることを可能にする仕組み。

ローカルのファイルに対してアクセスした内容をプライベートポインタに保持する場合はロックをお勧めします。

```
■静的変数宣言
static pthread_mutex_t   tmutex = PTHREAD_MUTEX_INITIALIZER;

■ロック処理
AZ(pthread_mutex_lock(&tmutex));
〜クリティカルセクション〜
AZ(pthread_mutex_unlock(&tmutex));
```

AZ は関数が0以外の時にエラーとする Varnish のマクロで定義は以下です。

```
#define AZ(foo)         do { assert((foo) == 0); } while (0)
```

しかし、単純なインクリメントの場合はできるだけロックしないことが望ましいです。数百以上のスレッドで動作するプログラムにおいて、クリティカルセクションが多いことは望ましくありません。一瞬で終わる処理も大量のスレッドが1つのリソースを奪い合う状況の場合多くの「待ち」が発生する可能性があります。
この本では紙面の都合上、具体的な方法は説明しませんが、以下の資料が大変参考になるため一読をお勧めします。

冬の Lock-Free 祭り (@kumagi さん)
http://www.slideboom.com/presentations/460931/冬の Lock-Free 祭り_safe

なお次ページ以降の例では紙面の都合上、**特にスレッドセーフは意識していません。**

PRIV_VCL

VMOD 全体で共通のプライベートポインタです

```
■vcc
Function INT tpriv_vcl(PRIV_VCL)

■c
int vmod_tpriv_vcl(struct sess *sp,struct vmod_priv *priv){
        int *i;
        if (priv->priv == NULL) {
                priv->priv = malloc(sizeof(int));
                i =(int*)priv->priv;
                *i=0;
                priv->free = free;
        }else{
                i =(int*)priv->priv;
        }
        *i=*i+1;
        return *i;
}

■VCL
set resp.http.test = example.tpriv_vcl();
```

PRIV_VCL は、VCL から呼び出す際に別途引数で指定する必要はありません。
VMOD 関数を呼ぶ際に Varnish が補完します。
使用するには以下のヘッダを include する必要があります。

```
#include "vrt.h"
```

stuct vmod_priv 周りの構造は以下です。

```
struct vmod_priv;
typedef void vmod_priv_free_f(void *);
struct vmod_priv {
        void                    *priv;          //プライベートポインタポインタ
        vmod_priv_free_f        *free;          //解放する際に呼び出す関数
};
```

ここで注意が必要なのが、**解放する際に呼び出す関数を指定する**ことです。
VCL 終了時に priv と free が定義されている場合、Varnish は定義された関数を呼び出して開放を行います。
例ではメモリを解放する free()を指定しています。独自に実装する場合は static 関数にしてください。

PRIV_CALL

呼び出し毎で利用できるプライベートポインタです。
基本的に PRIV_VCL と同じ定義で、変わるのは vcc の引数の型指定のみです。

■vcc

```
Function INT tpriv_call(PRIV_CALL)
```

■c

```
int vmod_tpriv_call(struct sess *sp,struct vmod_priv *priv){
        int *i;
        if (priv->priv == NULL) {
                priv->priv = malloc(sizeof(int));
                i =(int*)priv->priv;
                *i=0;
                priv->free = free;
        }else{
                i =(int*)priv->priv;
        }
        *i=*i+1;
        return *i;
}
```

■VCL

```
set resp.http.test = example.tpriv_call();
```

init_function

先ほど飛ばしていた vmod の初期化関数の init_function です。
これも PRIV_VCL が含まれています。

```
int init_function(struct vmod_priv *priv, const struct VCL_conf *conf){
        return(0);
}
```

PRIV_VCL の使い方は同様のため省略します。また VCL 自体の設定
（ファイル名や VCL のアクションへのポインタ等）を格納した*conf もあります。
これを利用するには以下のヘッダを include する必要があります。

```
#include "vcl.h"
```

VMOD で外部の共有ライブラリを使う

VMOD で外部の共有ライブラリを使うのは非常に簡単です。
インライン C で行ったように libmemcached を使ってみましょう。

まず最初に/src/Makefile.am を変更してます。

```
libvmod_example_la_LDFLAGS = -module -export-dynamic
 -avoid-version -lmemcached
```

LDFLAGS に-lmemcached を追加します。
次は vcc と c を記述します。

■vcc

```
Function VOID mcset(STRING,STRING)
```

■c

```
#include <stdlib.h>
#include <stdio.h>
#include <libmemcached/memcached.h>
void vmod_mcset(struct sess *sp, const char *k ,const char *v){
        struct memcached_st             *mmc            = NULL;
        struct memcached_server_st      *servers        = NULL;
        memcached_return                rc;

        mmc     = memcached_create(NULL);
                servers = memcached_server_list_append(servers,
                                "localhost", 11211, &rc);
        rc      = memcached_server_push(mmc, servers);
        memcached_server_list_free(servers);
                rc      = memcached_set(mmc, k, strlen(k),
                        v, strlen(v), 600, 0);

        memcached_free(mmc);
}
```

■VCL

```
example.mcset("Last:req.xid",req.xid);
```

インライン C と比較して、VCL からの呼び出しが非常に簡単なのがわかります。
また VMOD の場合、インライン C で設定したような起動パラメータの
cc_command の変更は必要ありません。
この点からも、筆者は外部のライブラリを使う際は VMOD をお勧めします。

VMOD で正規表現を使う

インラインＣの解説の際に、正規表現は事実上難しいため省略すると書きました。
理由として、Varnish が正規表現のコンパイルを最初に行い、その名前が「VGC_re_[数値]」と言ったおおよそ元の正規表現を想像しづらい物になる為です。
さらにインラインＣでは、プライベートポインタのようにセッションを超えた格納方法がありません。そのため、正規表現を使うには毎回コンパイルを行いすぐに開放せざるを得ないため実際使うのには不適切と考えています。
しかし、VMOD にはプライベートポインタが存在します。そのためコンパイルした正規表現を使いまわすことが可能です。以下例です。

■vcc

```
Function BOOL regex(PRIV_CALL,STRING,STRING)
```

■c

```
struct regex{ char *pat; void *regex; };
static void regexfini(void *d){
        struct regex *r=(struct regex*)d;
        free(r->pat);
        VRT_re_fini(r->regex);
}
unsigned vmod_regex(struct sess *sp, struct vmod_priv *priv,const char *pat
,const char *tg){
        struct regex *regex; int flag=0;
        if (priv->priv == NULL) {
                flag=1;
        }else{
                regex =(struct regex*)priv->priv;
                if(strcmp(regex->pat,pat)!=0){
                        regexfini(regex);
                        flag=1;
                }
        }
        if(flag){
                priv->priv = malloc(sizeof(struct regex));
                regex =(struct regex*)priv->priv;
                regex->pat = (char*)malloc(strlen(pat)+1);
                strcpy(regex->pat,pat);
                VRT_re_init(&regex->regex, pat);
                priv->free = regexfini;
        }
        return VRT_re_match(tg,regex->regex);
}
```

■VCL

```
example.regex(req.http.regex, req.url);
```

例では同じ正規表現が来る間、プライベートポインタに格納した物を使い、格納しているものと違う正規表現が来た場合は開放してコンパイルを行い続行するものです。
なお正規表現に関する関数は以下のとおりです。

```
VRT_re_init([正規表現を格納するポインタ],[正規表現])
        正規表現をコンパイルする
VRT_re_fini([正規表現を格納するポインタ])
        コンパイルした正規表現を解放する
VRT_re_match([評価文字列],[正規表現を格納するポインタ])
        正規表現でマッチを行う
VRT_regsub(sp,[置換フラグ],[評価文字列],[正規表現を格納するポインタ],[置換文字列])
        正規表現で置換する 置換フラグが0の場合は最初に一致したもの、1の場合は全て置換
```

VMOD のデバッグの仕方(varnishtest)

VMOD でデバッグを行うには幾つかの方法があります。最も良いのは varnishtest を使うことです。
通常の varnishtest での vtc の定義は変わりませんが、1つだけ忘れてはいけないことがあります。
VCL の定義で当然 vmod の import を行いますが、テストを行なっている vmod のため場所の指定をする必要があります。
以下のように行います。

```
varnish v1 -vcl+backend {
        import example from
                "${vmod_topbuild}/src/.libs/libvmod_example.so";
        ～VCL 中略～
} -start
```

また vtc は/src/tests/に置いておけば、特に Makefile.am に追加せずともテストします（vmod_example をベースとした場合）

おしらせ

達人出版会様(http://tatsu-zine.com/)から来年の初頭に Varnish 本を出すことを予定しています。近くなりましたらまたブログや Twitter で告知するとは思いますが、今まで時間の関係で書けなかった内容や様々な事を詰め込みますのでよろしくお願いします。

あとがき

初めましてな方ははじめまして！いわなましまろのいわなちゃんです。
夏コミに続いて、Varnish 本を作りました。今回はインライン C・VMOD と銘打っていますが、裏の目的としバージョンアップが怖くない様になればというのもあります。
御存知の通り、Varnish はかなり非互換の変更を行い、その際の移行ドキュメントも正直充実してるとは言いづらいです。しかし特定のソースを見る・diff すると大体変更内容がわかったりします。この本を読んで感覚を掴んで頂ければ幸いです。
また、今回とりあえず書き終えた（校正なし）タイミングがなんとコミケ4日前と恐ろしく落としかねないギリギリのスケジュールでした。そのため涙を飲んで端折った記述も多々あります（varnishtest とか）・・・。これから校正を行いますが、多分に読みづらい点があるかと思います。本当に申し訳ありません。

それでは機会があればまた！

奥付

Varnish Cache
inline-C/VMOD ガイドブック

ーー発行日ーー
2011－12－31（初版）
2012－01－26（2版）
2012－02－15（3版）

ーー発行ーー
いわなましまろ

ーー発行人ーー
いわなちゃん(@xcir)

ーー連絡先ーー
http://xcir.net/

ーー スペシャルサンクス（敬称略）ーー
@dai_yamashita
@W53SA
そして
Varnish Software

inline-C/VMOD ガイドブック