reverseproxy

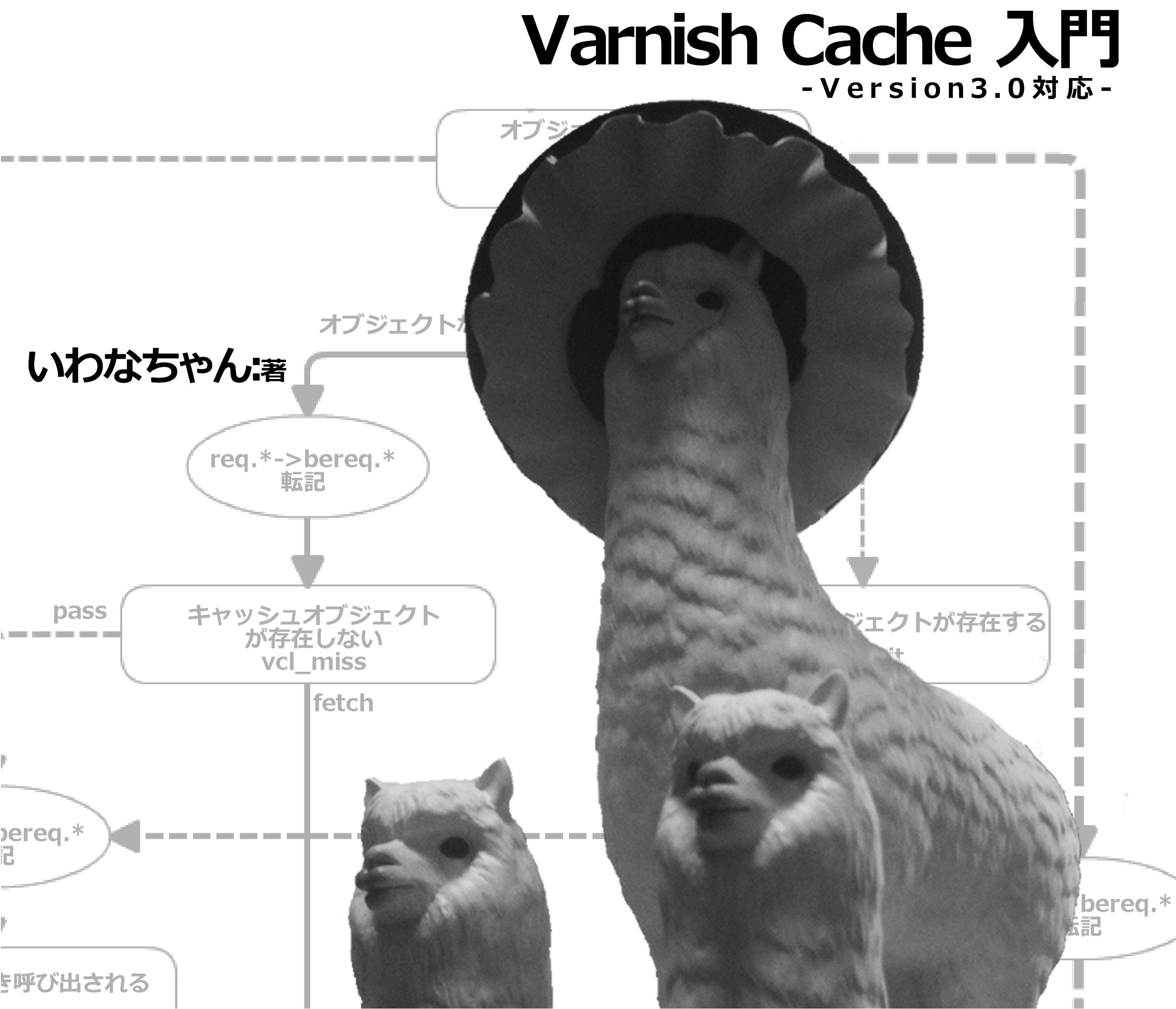
Varnish Cache

Introductory Book

Varnish Cache 入門

-Version3.0対応-

いわなちゃん:著

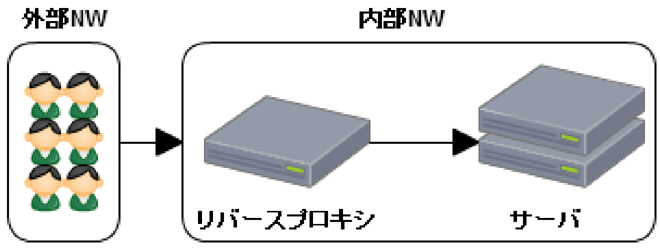


目次

なぜ Varnish Cache なのか..2
どのようなときに使用するか..3
    静的コンテンツの配信..3
    動的コンテンツの配信..5
Varnish を使うための環境..7
Varnish のインストール..7
まずは動かしてみよう..8
Varnish の設定について..9
VCL について..12
    基本的な言語仕様..12
    Varnish の基本動作ステップ..21
    req.*の内容が bereq.*に転記されるタイミング..27
    簡易版アクションフロー図..28
起動時オプションについて..29
Varnish のデバッグ..32
応用的な使い方..36
    管理コンソールの利用方法..36
    インライン C..38
    統計情報の見方..39
    ストレージサイズと TTL の決め方..42
    ヒット率の上げ方..43
    Edge Side Includes (ESI)..44
    gzip の利用..47
    VMOD の追加の仕方..49
    ストレージの高度な制御..50
    バックエンドの高度な利用..51
    varnishtest の使い方..51
    Varnish にやさしい ban の仕方..54
Tip's..55
Appendix..57
あとがき..64

なぜ Varnish Cache なのか

世の中には様々なリバースプロキシが存在します。リバースプロキシとは下図のようにクライアント・サーバ間の中継を行い、セキュリティ向上、リクエスト分散、コンテンツの圧縮・キャシュによる Web アクセラレータなどで利用します。



Web アクセラレータとして使う場合、以下のミドルウェアもしくはモジュールが候補によく上がります。

- プロキシ・リバースプロキシのデファクトスタンダード、多数の実績「**Squid**」
- ミドルウェアも足したくないし手軽に「**mod_cache**(Apache)」
- Inktomi から Yahoo!までの実績「**Apache TrafficServer**」
- Nginx 使ってる時に手軽に「**proxy_cache**(Nginx)」
- 余り表に出ないけど Steam などいろいろ使われている「**Varnish Cache**」

それぞれ利点と欠点が存在しますが、私が VarnishCache(以下 Varnish)を勧める一番大きな点が設定ファイルの柔軟さです。Varnish は VCL という C 言語に似た記述で動作をコントロールします。これを有名な Squid と比較してみてみましょう。

設定内容

- .js .css ファイルを一分キャッシュする。
- /ignore/ディレクトリにある場合はキャッシュしない。

Squid の場合

```
refresh_pattern /ignore/ 0 100% 0
refresh_pattern ¥.(js|css) 1 100% 1
```

Varnish の場合

```
sub vcl_fetch{
        if(req.url ~"/ignore/"){
                beresp.ttl=0s;
        }elsif(req.url ~"¥.(js|css)$"){
                beresp.ttl=1m;
        }
}
```

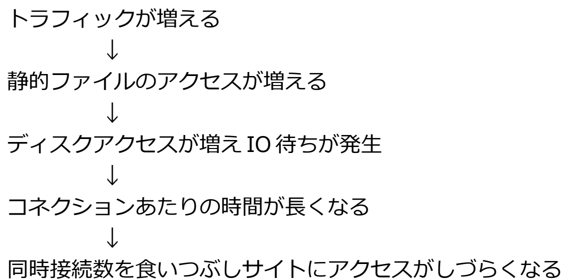
もし、あなたがプログラムの経験があるのであれば Varnish のほうが理解しやすいでしょう。そしてプログラムのように記述できるということは、複雑な条件も比較的頭を悩ませずに記述できることが期待できます。

どのようなときに使用するか

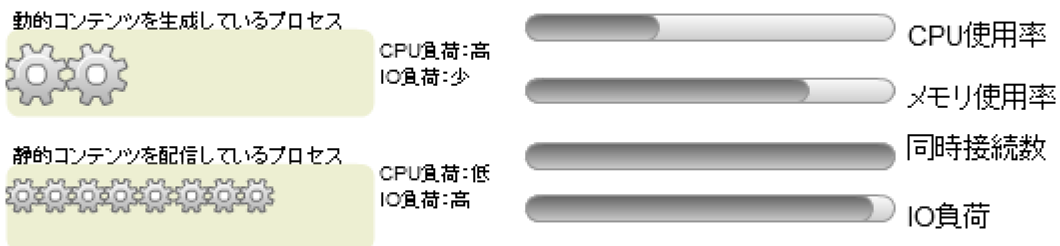
静的コンテンツの配信

Varnish に限らず Web アクセラレータを利用する際に最も一般的な使い方は、静的なコンテンツ（画像や JS や CSS など）を Web サーバとユーザの中間で高速にレスポンスすることです。
なぜそんなことをするのでしょうか？小規模なサイトの場合は PHP などが動いているアプリケーションサーバに静的コンテンツを置いても特に問題は発生しません。
しかしサイトが大規模になりアクセスが増えるとそうも言えなくなってきます。

アプリケーションと配信を同じサーバで行った場合



これはあくまで一例ではありますが起こりうる事態です。
そして下の図はその時のリソースの使われ方です。



Apache がリクエストを処理する体系に大きく2つあります。プロセスで動作する Prefork モデルと、スレッドで動作する Worker モデルです。プロセスは独立性が高い半面、メモリを含む多くのリソースを使用します。スレッドは独立性が低くプロセスに属するスレッド間のリソースの共有も容易でリソースも節約します。しかしそのために、PHP といった動作するモジュールはスレッドセーフでないといけません。
さて、静的コンテンツの場合 Apache の機能だけで配信するので Worker で問題ありません、しかし PHP を使うには一般的に Prefork を利用します。Worker・Prefork の混在はできないので Prefork を利用します。そのためプロセス数が同時接続数となります。プロセスはメモリを多く使うので、同時接続数が Worker よりも少なくなってしまいます。
そして1ページに複数静的コンテンツがあるのが一般的なので CPU・メモリ負荷の

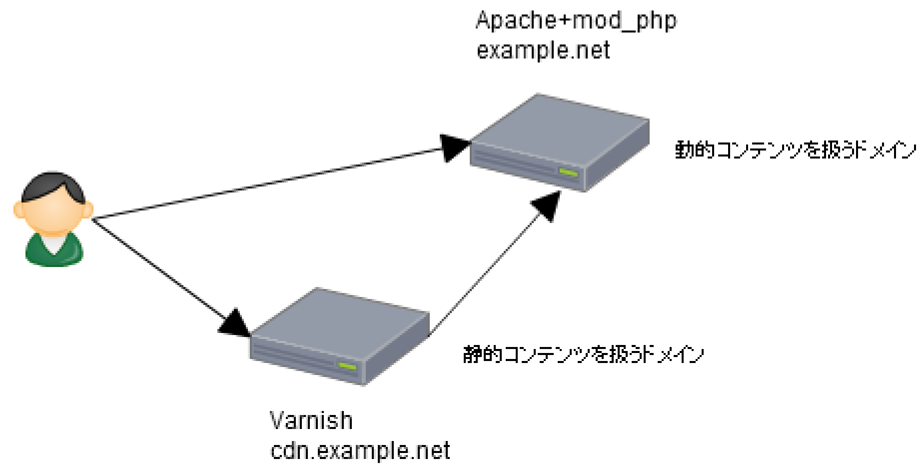
低い静的コンテンツに多くの接続数を利用されます。あまり効率が良くないのがわかります。ここで CPU に余力があるからとスペックを落としてメモリを増やしてバランスを取ろうとしても、一般的に CPU クロックを落とすとページの生成時間も長くなり秒間処理数が低くなってしまいます。また、ここでは IO 負荷も絡めていますが SSD にしたとしてもメモリの制限から同時接続数は大まかに以下になります。

同時接続数＝使えるメモリ/プロセスの平均メモリ使用量

さらに悪いケースについて考えてみましょう。同時接続数が超過した場合に発生する待ち行列についてです。

秒間接続数	秒間処理数	
205	200	
経過秒数	総コネクション数	待機数
1	205	5
10	2050	50
100	20500	500

秒間たった5アクセスが超過しただけで待機数はどんどん増えていくのがわかります。仮に100秒後に秒間接続数が199まで落ちたとしても待機数が0になるのには500秒かかります（実際はタイムアウトもありますが）つまり超過させたら負けなのです。さらに言うならこのサーバは、ページの生成と相乗りなのでページが見えないといった問い合わせが多数来るのが目に見えます。ではサーバを増やせば？もちろんそれも解決方法の一つですが、静的コンテンツが要因で超過したときにページが見えなくなってしまうのは解決しません。コントロールを容易にするるためにも、サーバを分けると同時に Web アクセラレータの導入をおすすめします。

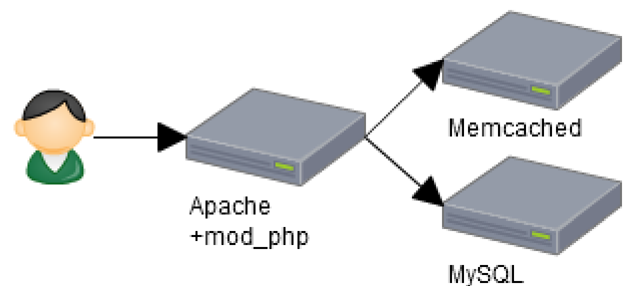


上図のようにまずドメインとサーバををわけます(cdn.example.net/example.net)
静的コンテンツの実体は元々のサーバ(example.net)にあるとします。

Varnish はリクエストを受け取り、キャッシュを保持していればレスポンスします。なければ、Apache に取得しキャッシュしてレスポンスします。つまり Apache への静的コンテンツのアクセスは最低限のレベルに収まり、動的コンテンツの生成に思う存分 CPU を使うことができるので、システム全体で処理できるリクエスト数は大幅に増えることを期待できます。また Varnish は高速なため Apache より多くのリクエストをさばくことができるのでユーザの体感で静的コンテンツの表示が速くなったと感じることもできるでしょう。
状況にもよりますがこれは単純に相乗りで二台に増やすよりシステムとしてのパフォーマンスが良いことが多いです。

動的コンテンツの配信

ある程度大きなサイトの場合、DB の負荷を抑えるために Memcache をはじめとした KVS の導入を行うところが多いです。



上図のような構成で以下のようなページについて考えてみましょう。

（ケース A）毎時更新されるランキングページ
（ケース B）下層コンテンツのタイトルをグループ毎に表示しているページ

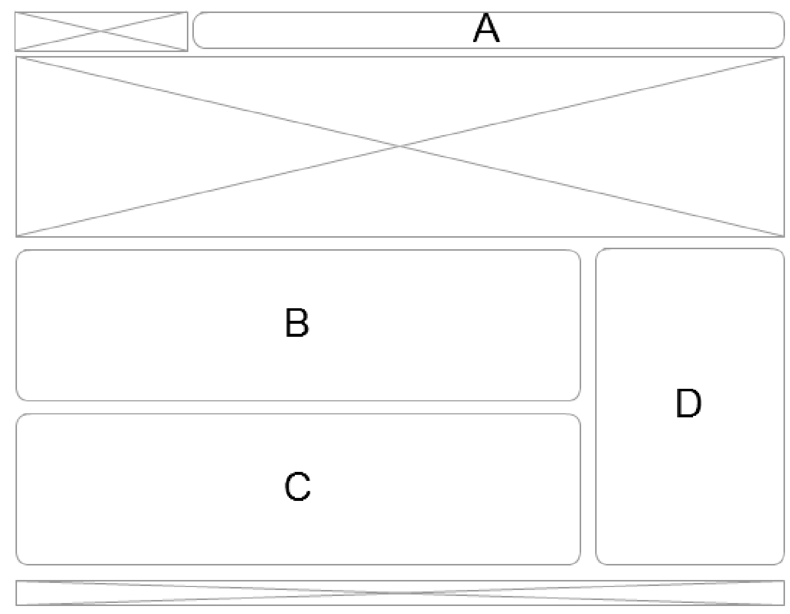
ケース A の場合、毎時でランキングを生成し DB に格納、リクエストのタイミングで Memcache に有効時間を残り時間で値を設定し、負荷を減らすのが一般的です。

その場合このような順序で生成します。

ランキングページのリクエストを受ける
↓
PHP が Memcache に値が存在するかチェックする
↓
値があれば取得し、なければ MySQL から取得して Memcache に格納
↓
取得した値を整形してページを作る
↓
レスポンスする

実にオーソドックスな処理ですが考えてみましょう。このページは動的コンテンツですが 1 時間は静的コンテンツです。つまり Varnish で一時間の間キャッシュしても

問題ありません。乱暴な言い方をすればキャッシュするのが Varnish なのか Memcache かの違いでしかありません。またそうすることで、余計な PHP の処理をすることなくユーザに高速にレスポンス出来ますし、システムの負荷も減ります。



次にケース B について考えてみましょう。若干イメージしづらいので図にしてみました。A にはユーザのアカウント名などの情報が、そして B・C には下層コンテンツのヘッドライン表示、D にはすべてのページで表示されるようなブロックと考えて下さい。それぞれが DB にアクセスして生成する必要があるブロックです。
通常、１ページ内に必ず動的でないといけない情報があるとキャッシュするのは難しいです。ここでいう A です。しかし B・C は下層コンテンツが更新されない限り表示は変わらないし、D に至ってはすべてのページで表示されるようなコンテンツです。ここだけでもキャッシュしてなんとか負荷を下げるにはどうするでしょうか？これを Varnish では ESI という技術で解決しています。ESI はページ中に以下の特殊なタグを埋め込み、タグの場所に他のコンテンツを埋め込むことが出来る技術です。

```
<esi:include src="/esi/block/B.php" />
```

ページ中に上記のように記述した場合 Varnish は src で指定された URL に対して取得を行いタグと置換します。もちろんキャッシュされていれば、それと置換します。つまり B・C・D もキャッシュが可能で DB の負荷を減らすことができます。
そして残った A ですが、一見キャッシュするのは難しそうに見えます。ユーザごとにアカウント名は違うし、別のユーザの情報が表示されては大変な問題です。しかしこれも解決できます。Varnish では同じ URL で複数のコンテンツを格納することが可能です。その際にはユニークな識別子が必要です、多くの会員サイトの場合クッキーにログイン情報をいれています、つまりそのユーザ専用のキャッシュを行うことができます。以降の章でこのような点についても解説します。

Varnish を使うための環境

公式では以下の環境を推奨しています。

・最新64 bit カーネルの Linux・FreeBSD・Solaris
・root 権

なお32 bit でも動作しますが検証ならともかく本運用で使うのはお勧めしません。
使えるメモリが少なすぎます。
また本書を書くに当たって Xen サーバ上で以下の構成で行っています。

OS: Scientific Linux6(以下 SL6)
Kernel: 2.6.32-71.29.1.el6.x86_64
IP: 192.168.1.199
CPU: 仮想 CPU×4
RAM: 2048MB
Storage:HDD100GB HDD5GB SSD5GB
Varnish3.0

Varnish のインストール

ソースビルドもできますが、本誌では簡単に RPM でインストールします。
最初に Varnish のレポジトリを利用するように設定します。

```
rpm --nosignature -i http://repo.varnish-cache.org/redhat/varnish-3.0/el5/noarch/varnish-release-3.0-1.noarch.rpm
```

次に yum でインストールを行います。

```
yum install -y varnish
```

この際以下のような依存関係のエラーが出る場合があります。

```
Error: Package: varnish-3.0.0-2.el5.x86_64 (varnish-3.0)
          Requires: libjemalloc.so.1()(64bit)
```

jemalloc は SL6 では標準にレポジトリには含まれておらず、EPEL を追加する必要があります。その際に単純に EPEL レポジトリを追加するだけではなく SL6 では提供されていないパッケージのみをインストールするよう yum-priorities をインストールします。

```
yum -y install yum-priorities
```

次に/etc/yum.repos.d/sl.repo の内容を変更します。

```
[sl]
name=Scientific Linux $releasever - $basearch
[中略]
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-sl file:///etc/pki/rpm-gpg/RPM-GPG-KEY-dawson
priority=1   ←追加
```

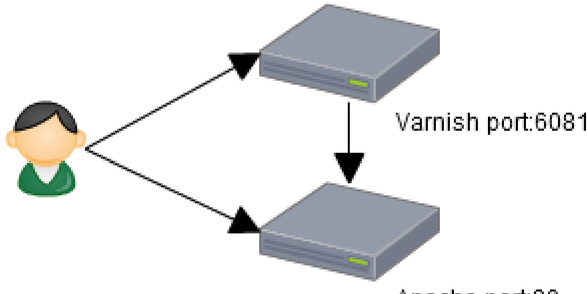
次に EPEL リポジトリを導入します。

```
rpm -ivh http://download.fedora.redhat.com/pub/epel/6/x86_64/epel-release-6-5.noarch.rpm
yum -y update epel-release
```

これで yum install -y varnish で Varnish をインストールできます。

まずは動かしてみよう

何事もまずは触ってみないとわからないので早速動かしましょう。とりあえず以下の構成で動かします。



まずはそのまま Apache を起動します。

```
/etc/init.d/httpd start
```

適当なブラウザでページを見てみましょう。きっと以下の画面が出てきます。

Scientific Linux Test Page

This page is used to test the proper operation of the Apache HTTP server after it has been installed. If you can read this page, it means that the Apache HTTP server installed at this site is working properly.

If you are a member of the general public:

The fact that you are seeing this page indicates that the website you just visited is either experiencing problems, or is undergoing routine maintenance.

If you would like to let the administrators of this website know that you've seen this page instead of the page you expected, you should send them e-mail. In general, mail sent to the name "webmaster" and directed to the website's domain should reach the appropriate person.

For example, if you experienced problems while visiting www.example.com, you should send e-mail to "webmaster@example.com".

For information on Scientific Linux, please visit the Scientific Linux website.

If you are the website administrator:

You may now add content to the directory /var/www/html/. Note that until you do so, people visiting your website will see this page, and not your content. To prevent this page from ever being used, follow the instructions in the file /etc/httpd/conf.d/welcome.conf.

You are free to use the image below on web sites powered by the Apache HTTP Server:

次にVarnishを起動します。

```
/etc/init.d/varnish start
```

そしてブラウザでポートを6081でアクセスします。先ほどと同じ画面がのはずです。さてあまりにもあっさり行きすぎていて本当に動いてるか不安を感じるかもしれません。試しにApacheを止めてみましょう。

```
/etc/init.d/httpd stop
```

まずはポート80にアクセスしてみます。もちろんApacheを落としているので閲覧できません。次に6081でアクセスします、おそらく先ほどと同じ画面が出てくるはずです。これできちんとキャッシュされていることを確認できます。ちなみにしばらく経つとキャッシュの有効期限が切れて見えなくなってしまいます。なので上手く確認できない場合は素早くやってみましょう。
また、本誌で試す上で多くの設定を書き換えます。その際に動的に適用するには

```
/etc/init.d/varnish reload
```

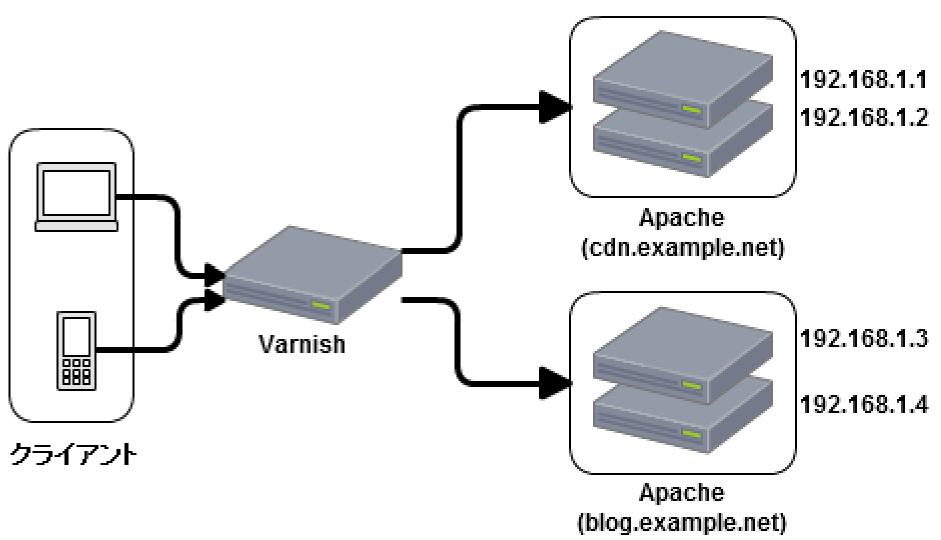
でおこないます。ここではVarnishのデフォルトの設定で動かしてみて、キャッシュの動きを確認できました。次はVarnishの設定の仕方を説明します。

Varnishの設定について

Varnishの設定は大きく以下の二つに分かれます。

VCL（一般的に言う設定ファイルに相当）
起動オプション

VCLはユーザからのリクエストをどのように処理するかを決めるための設定です。設定とは言うものの最初で触れたように一種のプログラミングのように記述します。
起動オプションはVarnishの起動時に指定するオプションです。キャッシュの保存にメモリを使うかなどの指定を行います。まずは下記構成でのVCL例を見てみましょう。



キャッシュ条件

共通

①クエリに purge=1 を含む場合かつ192.168.1.0/24からアクセスの場合キャッシュを削除。
②ヘルスチェク対象は「/」

blog.example.net

③/admin/にマッチしたURLはキャッシュしない。
④30分キャッシュする。
⑤同一URLで携帯とPCにアクセスした場合デザインの違うページをそれぞれキャッシュして出力
⑥ blog.example.net がダウンの際は cdn.example.net/sorry.html を1分キャッシュで出力。

cdn.example.net

⑦ 全て6時間キャッシュ
⑧同一URLで携帯とPCで表示を切り返す。/a.jpg の場合、携帯は/_m/a.jpg PC は/_p/a.jpg

VCL（条件に関わる箇所には該当する番号を書いています）

```
acl local {
  "192.168.1.0"/24;          ・・・①
}

//ヘルスチェック設定
probe healthcheck {
  .url = "/";    ・・・②
  .timeout = 1s; .window = 8; .threshold = 3; .interval = 5s;
}

//バックエンド設定
director cdn random{.retries = 5;
  {.weight = 5;.backend={.host="192.168.1.1";.probe=healthcheck;}}
  {.weight = 5;.backend={.host="192.168.1.2";.probe=healthcheck;}}
}

director blog random{.retries = 5;
  {.weight = 5;.backend={.host="192.168.1.3";.probe=healthcheck;}}
  {.weight = 5;.backend={.host="192.168.1.4";.probe=healthcheck;}}
}

sub vcl_recv{  //クライアントからのレスポンスを受け取る
  if(req.url ~ "(¥?|&)purge=1" && client.ip ~ local){//キャッシュ削除  ・・・①
    ban("obj.http.X-HOST ~ "+ req.http.host + " && obj.http.X-URL ~ " + regsub(req.url,"¥?.*$",""));
    error 200 "banned.";
  }
  //admin/以外のクッキー削除
  if(req.http.Cookie && !(req.http.host ~ "^blog.example.net" && req.url ~ "^/admin/")){
      unset req.http.Cookie;
  }
  if(req.http.host ~ "^blog.example.net"){
    set req.backend = blog;
    if( ! req.backend.healthy){       //blog 死亡              ・・・⑥
      set req.backend = cdn;          //blog 死亡のため cdn/sorry.html に向ける
      set req.url      = "/sorry.html";
    }elseif(req.url ~ "^/admin/"){ //admin はキャッシュしない
      return(pass);  ・・・③
    }
  }elseif(req.http.host ~ "^cdn.example.net"){
```

```
   set req.backend = cdn;
 }else{//不明なホストは403
   error 403 "Forbidden";
 }
 return(lookup);
}

sub vcl_hash {  //キャッシュ格納・特定に使うハッシュキーを作成
 hash_data(req.url);
 if (req.http.host) {
   hash_data(req.http.host);
 } else {
   hash_data(server.ip);
 }
 if(req.http.User-Agent ~"(DoCoMo|UP¥.Browser|KDDI|SoftBank|J-PHONE|Vodafone)"){
   hash_data("mobile");          //携帯の場合は同一 URL で別キャッシュ  ・・・⑤
   if(req.http.host ~ "^cdn.example.net"){
     set req.url = "/_m" + req.url;   ・・・⑧
   }
 }elseif(req.http.host ~ "^cdn.example.net"){
   set req.url = "/_p" + req.url;       ・・・⑧
 }
 return (hash);
}

sub vcl_fetch{  //バックエンドからのレスポンスヘッダを受信
 set beresp.http.X-URL = req.url;
 set beresp.http.X-HOST = req.http.host;
 if(beresp.status >= 400){    //キャッシュ時間設定
   set beresp.ttl = 1m;
 }elseif(bereq.url == "/sorry.html"){
   set beresp.ttl = 1m;    ・・・⑥
 }elseif(bereq.http.host ~ "^blog.example.net"){
   set beresp.ttl = 30m;   ・・・④
 }elseif(bereq.http.host ~ "^cdn.example.net"){
   set beresp.ttl = 6h;    ・・・⑦
 }
 return(deliver);
}

sub vcl_deliver{  //クライアントにレスポンスする直前
 if(obj.hits > 0){
   set resp.http.X-Cache = "HIT";
 }else{
   set resp.http.X-Cache = "MISS";
 }
}
```

ページの都合上若干圧縮した記述で申し訳ありあせん。Varnish は上記のような設定ファイル（VCL)で動作します。これはわざと色々組み込んで複雑にしていますが、本誌を全て読み終えた際には理解できると思います。

次ページから VCL の基本文法について解説します。

VCLについて

VCLはVarnishでの処理に特化したC言語に近い言語です。実際のところVCLは以下のように利用されます。

1. VCLファイルを読み込む
2. C言語に解釈する
3. コンパイルして共有ライブラリを作る
4. Varnishがそれをリンクする

見てわかるとおり、通常のミドルウェアがテキストベースの設定を読んで動くのとは全く違います。本当に言語です、またC言語の記述も書くことが可能です。ではどのような言語なのでしょうか？

基本的な言語仕様

型の種類

数値

12345（整数）123.45(小数点付)

注：0 xC0のような16進数での指定は出来ません。

時間とサイズ

数値の後に単位をつけることが可能

単位	説明	変換
ms	ミリ秒	
s	秒	1000ms = 1s
m	分	60s = 1m
h	時間	60m = 1h
d	日	24h = 1d
w	週	7d = 1w

単位	説明
B	バイト
KB	キロバイト
MB	メガバイト
GB	ギガバイト
TB	テラバイト

BOOL

trueとfalse

IP範囲

“192.168.1.0”/24

文字列

単行の場合　“text”

複数行の場合　{“longtext”}

演算子

比較演算子

符号	説明
==	等しい
!=	等しくない
<	小さい
>	大きい
<=	以下
>=	以上
~	正規表現/ACLマッチする
!~	正規表現/ACLマッチしない

代入演算子

符号	説明
=	代入
+=	加算して代入
-=	減算して代入
*=	乗算して代入
/=	除算して代入

算術演算子

符号	説明
*	乗算
+	加算
－	減算
/	除算

<table>
<tr><th colspan="2">論理演算子</th></tr>
<tr><td>&&</td><td>論理積</td></tr>
<tr><td>||</td><td>論理和</td></tr>
<tr><td>!</td><td>否定</td></tr>
</table>

文字列の結合には+を利用する。
その他に( )を利用することができる。

コメント

```
# コメント行
// コメント行
/* コメント */
```

条件文

```
if(式){処理}
elseif(式){処理}
else{処理}
```

※elseif は elsif とも表記可能です。

識別子

サブルーチン・バックエンド名などで使える文字です。以下の正規表現にマッチする必要があります。

[a-zA-Z][a-zA-Z0-9_-]*

正規表現

Varnish では Perl 互換の PCRE を利用しています。
正規表現にオプションをつけたい場合は**(?**の後にオプションを記述し**)**で閉じます。

```
if(req.http.host ~ "(?i)example¥.net$"){～}
```

ACL

クライアントのIPアドレスと照合するためのリストです。

```
acl ACL名 {
        "ホスト名｜IPアドレス｜サブネット";
}

実際の定義
acl local {
        "localhost";       // 自分自身
        "192.0.2.0"/24;    // ローカルネットワークは許可
        ! "192.0.2.23";    // ルータは除外
}
```

ホスト名を使い、起動時に名前解決できない場合は意図しない動作になります。

また、ACLでクライアントIPと比較する場合は以下のように行います。

```
If (client.ip ~ local) {～マッチする}
else{～マッチしない}
```

バックエンド

Varnishはリバースプロキシとして動作しますので、クライアントからのリクエストをどこのサーバに中継するかの定義が必要です。そのためバックエンドは最低一つ指定しないといけません。以下のように定義します。

```
backend [バックエンド名] {
        .host = "[ホスト名もしくはIPアドレス]";
        .port = "[ポート番号かサービス名]";
}

デフォルトの定義
backend default {
        .host = "127.0.0.1";
        .port = "80";
}
```

これは最低限のオプションで、以下の追加指定も可能です。

.connect_timeout = **3s**;

接続する際のタイムアウト時間

.host_header = **“example.net”**;

bereq.http.hostに指定がない場合のデフォルトのHostヘッダ

.first_byte_timeout = **3s**;

最初のバイトを受信するまでのタイムアウト時間

.between_bytes_timeout = **10s**;

通信中に一時的に受信が待ったときのタイムアウト時間

.max_connections = **50**;

このバックエンドの最大コネクション数

.saintmode_threshold = **10**;

saintmodeに入る際の閾値

またVarnishはヘルスチェック機能を持っています。この機能はバックエンドが、何らかの原因でダウンしているかどうかを判定し、状況によってはリクエストをしないことが可能です。その際に以下のような指定を行います。

```
.probe = {
        .url = [ヘルスチェックファイル];
        .timeout = [タイムアウト時間];
        .window = [直近 n アクセスを評価対象とするか];
        .threshold = [何回失敗したら異常とするか];
}
```

以下の場合/chk.jpgに対してタイムアウト0.3秒で直近5回のうち2回失敗したら異常と判定します。

```
.probe = {
        .url = "/chk.jpg";
        .timeout = 0.3s;
        .window = 5;
        .threshold = 2;
}
```

若干windowとthresholdの関係が分かりづらいので解説します。
例えばwindow=5 threshold=2とした場合で考えてみましょう。

1	2	3	4	5	6	7	8	9	直近チェックの順番	Healthy
○	×	○	○	×	○	○	○	×	ヘルスチェックの成否	
				5	4	3	2	1	チェックする範囲（window=5)	
					3	2	1		成否の数チェック(threshold=2)	

直近5回のチェック中3回成功しているのでHealthy(正常)です。

1	2	3	4	5	6	7	8	9	直近チェックの順番	Sick
○	×	○	○	×	○	×	×	×	ヘルスチェックの成否	
				5	4	3	2	1	チェックする範囲（window=5)	
					1				成否の数チェック(threshold=2)	

直近5回のチェック中1回しか成功していないのでSick（異常）と判定され正常になるまで切り離されます。ここで少し考えてみましょう起動時にはヘルスチェックをまだ行っていないので全て異常になりそうですが、それを防ぐためにinitialという指定があります。この指定回数成功しているとみなしています。
他には以下の指定が存在します。

```
.initial = 3;
        起動時に既に何回成功しているとみなすか
.expected_response = 200;
        どのステータスを成功とみなすか
.interval = 5s;
        何秒毎にヘルスチェックを行うか
.request = "GET / HTTP/1.1"
          "Host: www.foo.bar"
          "Connection: close";
        ヘルスチェックする際のHTTPリクエスト、.urlと同時指定はできない
```

組み合わせると以下の記述になります。

```
backend default {
        .host = "127.0.0.1";
        .port = "80";
        .probe = {
                .url = "/chk.jpg";
                .timeout = 0.3s;
                .window = 5;
                .threshold = 2;
        }
}
```

また probe は、以下のように外出ししての記述も可能です。

```
probe healthcheck {
        .url = "/chk.jpg";
        .timeout = 0.3s;
        .window = 5;
        .threshold = 2;
}
backend default {
        .host = "127.0.0.1";
        .port = "80";
        .probe = healthcheck;
}
```

複数のバックエンドで同じヘルスチェックを行う場合は楽なのでおすすめです。

ディレクター

ディレクターは複数のバックエンドを、クラスタ化したグループです。Probe を指定してヘルスチェックを行っている場合、異常なバックエンドを切り離してクラスタ内の他バックエンドにリクエストするなどが可能です。基本的な指定は以下です。

```
director [ディレクター名] [ディレクタータイプ] {
        {
                .backend = [バックエンド名];
        }
        {
                .backend = {
                        .host = "127.0.0.1";
                        ～インラインでバックエンド記述も可能～
                }
        }
}
```

ディレクタータイプとは、バックエンド群からどれを使うかを決める際の方式です。次のタイプが存在し、追加のパラメータが存在したりします。

random

バックエンド群からランダムに選択します。

```
director d1 random{
        .retries = 5; //バックエンドを選択する際に最大何回リトライを行うか
        {
                .backend=b1;
                .weight = 7; //バックエンドを使う際の重み付け指定
        }
        {
                .backend=b2;
                .weight = 3; //バックエンドを使う際の重み付け指定
        }
}
```

round-robin

バックエンド群から順番に選択します。特に追加パラメータはありません。

DNS

バックエンドを選ぶ際に DNS を利用して定義済みの IP リストとマッチするものをバックエンドとして選択します。以下のように記述します。

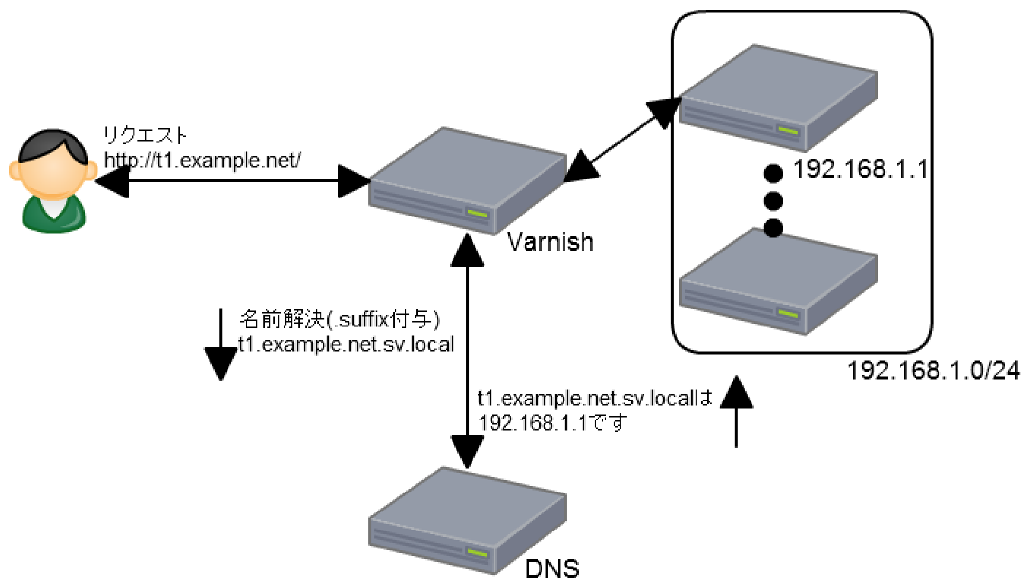
```
director d1 dns{
        {
                .backend = b1; //バックエンドも指定可能
        }
        .list = {
                .port = "80";
                "192.168.15.0"/24; //IP 帯域を指定する
                "192.168.16.128"/25; //複数指定も可能
        }
        .ttl = 5s; //DNS の結果をどの程度キャッシュするか（任意）
        .suffix = ".sv.local"; //DNS 解決する際のホスト名の末尾に付ける文字列（任意）
}
```

.list 内には.port 以外に以下も指定可能です。

.host_header
.connect_timeout
.first_byte_timeout
.between_bytes_timeout
.max_connections
.saintmode_threshold

オプションの意味は先程のバックエンドのオプションと同じです。

これだけだと DNS ディレクターの動きが分かりづらいので詳しく解説します。



上記図でのディレクターの設定は以下です。

```
director d1 dns{
        .list = {
                .port = "80";
                "192.168.1.0"/24;
        }
        .ttl = 5s;
        .suffix = ".sv.local";
}
```

流れは以下です。

1. クライアントが http://t1.example.net/を要求
2. Varnish が t1.example.net を解決するため DNS に.suffix を付与して問い合わせする (t1.example.net.sv.local)
    1. DNS が t1.example.net.sv.local は192.168.1.1と返却
    2. DNS の結果を5秒キャッシュする(.ttl)
3. 192.168.1.1に対して Host を t1.example.net でリクエストする
    1. 結果を受け取る
4. クライアントにレスポンスする

ここで注意が必要です。名前解決する際は.suffix が付与されますが、バックエンドにリクエストするときは付与されません。公式を見ると sv.local の指定だけで動きそうなのですが単純に結合しているので「.」を付けないと t1.example.**netsv**.local で名前解決しようとするので注意が必要です。

また、DNS が複数の IP を返してきた場合はそれでラウンドロビンします。

なお、.list で範囲指定を行う場合はヘルスチェックができません。

そのため backend で範囲分のサーバを、インラインで次のように全て定義してしまうのもいい手です。

```
probe healthcheck {
            .url = "/chk.jpg";
            .timeout = 0.3s;
            .window = 5;
            .threshold = 2;
            .interval = 1m;
}
director d1 dns{
            {.backend={.host="192.168.1.1";.probe=healthcheck;}}
            {.backend={.host="192.168.1.2";.probe=healthcheck;}}
            ～
            {.backend={.host="192.168.1.254";.probe=healthcheck;}}
            .ttl = 5s;
            .suffix = "sv.local";
}
```

.list 表記でも Varnish 内部で最初に.backend に変換しているので、ヘルスチェック以外のオーバーヘッドの心配はほぼないと考えています。ただ台数が多いので
ヘルスチェックの間隔はできるだけ長めにしたほうが良いです。

client

クライアントの識別情報(client.identify)に基づいてバックエンドを選択します。
簡単にいえば、クライアント A はバックエンド A を常に見に行き、またクライアント B はバックエンド B を見に行く、といった制御が可能です。client.identity に
クッキーのような識別子を指定すればセッションパーシステンス機能のように動作し、リクエスト URL を指定すればコンテンツスイッチング機能のように動作します。
追加パラメータは random ディレクターと同様に.retries と.weight が存在します。

hash

リクエストされた URL のハッシュ値に基づいてバックエンドを選択します。
これは多段で構築している場合、そのバックエンド群で重複したキャッシュを保持しないため便利です。
追加パラメータは random ディレクターと同様に.retries と.weight が存在します。

fallback

Version3.0.1 で追加されたディレクターで、その名の通り fallback(縮退）します。
最初に定義されたバックエンドが正常な限り常に選択しますが、異常になった場合
2番目に定義されたバックエンドを選択します。
特に追加パラメータはありません。

```
director b3 fallback {
  { .backend = www1; }
  { .backend = www2; } // www1 が異常状態の時に使用される
  { .backend = www3; } // www1 と www2 の両方が異常状態の時に使用される
}
```

変数操作

変数に対し値を設定は以下のように行います。

set **[変数] [代入演算子] [値]**;

```
set req.http.test = “testtest”;
```

削除は以下のように行います。

unset **[変数]**;

```
unset req.http.test;
```

その他記法

インライン C

C{ C で書かれたコード }C

別ファイルの読み込み

include “filename”;
※フルパスが望ましい

サブルーチン

sub サブルーチン名{処理}

サブルーチン呼び出し

call サブルーチン名;

以上が基本的な言語仕様です。次に Varnish がどのようにリクエストを受けて
キャッシュしそれをレスポンスするかについて解説します。

Varnish の基本動作ステップ

1. ユーザからのリクエストを受ける
2. キャッシュの有無をチェック
3. キャッシュがなければバックエンドに取得しに行く
4. キャッシュしても良いデータならキャッシュする
5. ユーザにレスポンスする

かなり端折って書いていますが、大まかな流れはこのような形です。Varnish では各ステップで呼ばれるアクションが存在し、そこで様々な条件分岐を行います。
例えば特定の URL はキャッシュしなかったり、他より長めにキャッシュしたり、
同じ URL だけど UserAgent で複数のキャッシュをもったり様々なことができます。
また、制御を行うための判断材料となる様々な変数は、アクションによってアクセスが制限されています。ちょっと考えてみましょう、バックエンドからのレスポンスをユーザからのリクエストを受けたときに把握することは不可能です。変数の一覧及び各アクションでのアクセス権、指定できる return 値は Appendix を参照してください。さて、それぞれのアクションについて代表的な使い方と一緒に解説します。

vcl_recv リクエストを受けた時に呼ばれる

ユーザからリクエストがあった際に一番最初に呼ばれるアクションです。
おそらく一番処理を書くことが多い箇所です。
ここでよく行う処理は以下です。それぞれ解説します。

1. どのバックエンドを使うか指定する
2. キャッシュしやすいようにホストなどを正規化する
3. 特定の IP からのアクセスを除外
4. キャッシュの消去を行う(ban)
5. オブジェクトをどのように返却するか

どのバックエンドを使うか指定する

バックエンドとは Varnish がコンテンツを取得しに行くサーバのことです。
例えば t1.example.net の時は予め定義した b1 バックエンドで t2.example.net の時は b2 とした場合は次のような記述となります。

```
if      (req.http.host ~”t1¥.example¥.net$”){
            set req.backend = b1;
}elseif(req.http.host ~”t2¥.example¥.net$”){
            set req.backend = b2;
}
```

なお、指定がない場合は default というバックエンドを使用します。

キャッシュしやすいようにホストなどを正規化する

例えば、あなたが運営しているサイトが以下のドメインを持っているとします。

- www.example.net
- example.net

ドメインは違いますが、返す内容が同じ場合とします。しかし Varnish はホストが違うためそれぞれでキャッシュを保持します。
このような場合無駄なので以下のように正規化します。

```
if(req.http.host ~ "(?i)(www¥.)?example¥.net"){
        set req.http.host = "example.net";
}
```

このようにできるだけ同じ内容で微妙にドメインや URL が違ったりした場合は正規化することにより、ヒット率が向上します。

特定の IP からのアクセスを除外

例えば/_s/で始まる URL は、LAN 内のみの許可の場合は以下のように記述します。

```
acl local {
        "localhost";       // 自分自身
        "192.0.2.0"/24;   // ローカルネットワークは許可
        ! "192.0.2.23";    // ルータは除外
}
sub vcl_recv{
        ～
        If (req.url ~"^/_s/" && client.ip !~ local) {
                error 403 "Forbidden"; //403 を返却
        }
        ～
}
```

キャッシュの消去を行う(ban)

運用するに当たって、しばしばキャッシュを削除したいことがあります。

```
PURGE / HTTP/1.0
Host: www.example.com
```

上記のようなリクエストで/を削除するには以下のように記述します。

```
if (req.request == "PURGE") {
        ban("req.http.host == " + req.http.host + "&& req.url == "+ req.url);
        error 200 "banned.";
}
```

若干分かりづらいので変数部を展開してみましょう。

```
req.http.host == "example.net" && req.url == "/"
```

という文字列が ban 関数に渡されます。見てわかるとおり式なので例えば

```
req.http.host == "example.net" && req.url ~ "^/image/"
```

のように image ディレクトリ配下の全てを対象とする事も可能です。

しかし、ban は実行してすぐにキャッシュが消えるわけではありません。内部で ban リストを作っており、それを lookup もしくは定期的に評価しマッチするものがあれば消去します。評価する際にはオブジェクトがキャッシュされた時間より後にリストされたものと照合しますが、全体的にオブジェクトが長い TTL を持っていて大量に ban している場合はパフォーマンスに影響を与える可能性もあるため注意してください。

また他に、即キャッシュから消去する purge というものもありますが後述します。

オブジェクトをどのように返却するか

例えば会員ページなど、キャッシュしてはいけないページや不明なメソッドを制御することができます。

```
if(req.request !~ "^(GET|HEAD|PUT|POST|TRACE|OPTIONS|DELETE)$"){
        return(pipe); //不明なメソッドの通信はパイプする
}elseif(req.url ~ “^/admin/”){
        return(pass); //admin ページはキャッシュしない
}else{
        return(lookup); //キャッシュの動作をする
}
```

ここで出てきた pipe,pass,lookup ですが次の違いがあります。

pipe

クライアントとバックエンド間のコネクションが閉じられるまで、継続的にスルーします。Varnish は内容に対する操作は行いませんし、キャッシュも行いません。単純に右から左へ通信を中継するだけです。

pass

Varnish はキャッシュしません。しかし、返却時にレスポンスヘッダに追加で項目を加えるなどの操作は可能です。

lookup

キャッシュを保持している場合はキャッシュを返却し、保持していなければキャッシュしようとします。

vcl_pipe pipe 動作をするときに呼び出される

主に行うことは、バックエンドへのコネクションタイムアウト値を変更と pipe 時に Varnish が bereq.*に転記しないヘッダの再設定をする程度であまり使いません。

vcl_pass pass 動作をするときに呼び出される

pass 動作時に、バックエンドへリクエストヘッダを改変する必要がある場合などに使います。

vcl_hash オブジェクトを特定するハッシュを作成する

たとえば会員ページをどうしてもキャッシュしたい場合どうすればいいでしょうか？
他にも同じ URL だけど、携帯のキャリアごとにキャッシュする内容を変えたいと
いったこともあるかもしれません。通常 Varnish は URL やホストなどからハッシュ
値を作成しこの値でキャッシュを格納しています。

```
sub vcl_hash {
        hash_data(req.url);
        if (req.http.host) {
                hash_data(req.http.host);
        } else {
                hash_data(server.ip);
        }
        //ここに追加で増やす
        return (hash);
}
```

そして上記で示した「**ここに追加で増やす**」に追記していってハッシュを作成する
ときに使う値を追加で増やすことができます。
もし、vcl_hash の定義を消してみたらどうなるでしょうか？

```
sub vcl_hash {
        //ハッシュ値がいつも同じになるのでいつも同じキャッシュを返す
        return (hash);
}
```

ハッシュ値が同じになるため、最初にキャッシュしたオブジェクトを他の URL に
アクセスしても返します。なので元の記述に対して修正する際は注意してください。
さて先程に例で出した会員ページをキャッシュしたい、と携帯キャリアごとに内容を
変えたいを書いてみます。**元の記述を消さず**に追加する場所に追記してください。

会員ページをキャッシュしたい

認証情報にクッキーを使っていると仮定して/member/のページでユーザ毎にキャッ
シュを作ります。

```
if(req.url ~ "^/member/"){
        hash_data(req.http.Cookie);
}
```

クッキーのデータに認証以外のデータが入っている場合、それも含めてハッシュ値作
成に使われますので、認証データだけに正規化するとより効率が良くなります。

携帯キャリアごとに内容を変えたい

ip アドレスから判定してもいいのですがここは手軽に UserAgent で判定します。

```
if(req.http.User-Agent ~"DoCoMo"){
          hash_data("mobile-docomo");
}elseif(req.http.User-Agent ~"(UP¥.Browser|KDDI)"){
          hash_data("mobile-kddi");
}elseif(req.http.User-Agent ~"(SoftBank|J-PHONE|Vodafone)"){
          hash_data("mobile-softbank");
}
```

vcl_hit キャッシュオブジェクトが存在する

キャッシュオブジェクトが存在するときに呼ばれます。
主に後述の vcl_miss と合わせて特定のキャッシュの即消去（purge）に使われます。

```
sub vcl_hit {
  if (req.request == "PURGE") {
    purge;
    error 200 "Purged.";
  }
}
```

通常 GET や HEAD が入っている req.request に PURGE が入っていた場合、
選択されたオブジェクトを即時に消去します。

vcl_miss キャッシュオブジェクトが存在しない

キャッシュオブジェクトが存在しないときに呼ばれます。
vcl_miss 時はキャッシュしていないので必要ないと思われますが、purge リクエストに対しコマンドが正しく受け付けられたことを示すためにも、同じ記述をするのが一般的です。

vcl_fetch バックエンドからレスポンスヘッダを受信

ファイルの有無(404)や、特殊なヘッダで動作を制御したい際に使います。これは pipe を除いたバックエンドとのやりとりをする際は必ず呼ばれます。様々な使い方がありますが、基本的な使い方はキャッシュの保持時間の設定です。
例えば画像は12時間、他は3時間キャッシュする場合以下のように記述します。

```
if(req.url ~ "¥.(jpe?g|gif|png)(¥?.*)?$"){
          beresp.ttl = 12h;
}else{
          beresp.ttl = 3h;
}
```

他の使い方として Squid でいう negative_ttl のように404の際に短期間キャッシュすることもよく行います。

```
if(beresp.status == 404){
        beresp.ttl = 5m;
}
```

また、何らかの条件でそもそもキャッシュしたくないこともあります。
Varnish は Pragma を無視しますが HTTP1.0 なサーバはこれをレスポンスしてくることがあります。

```
if(beresp.http.Pragma ~ "no-?cache"){
        return(hit_for_pass);
}
```

なお、ここで TTL の制御が可能なのは vcl_miss を通過するパタンのみ、つまり pass を指定していないことが必須です。既に pass されていて TTL を指定してもエラーにはなりませんが、設定してもキャッシュはされませんので注意してください。

vcl_deliver クライアントにレスポンスする直前

vcl_deliver は pipe 動作を除きクライアントにレスポンスする際に必ず呼ばれます。何らかのヘッダを付与したり、または削除したりします。ここでよく行うのはどちらかというとデバッグのためにオブジェクトが HIT したのか MISS したのかを判定するヘッダを付与することです。

```
if(obj.hits > 0){
        set resp.http.X-Cache = "HIT";
}else{
        set resp.http.X-Cache = "MISS";
}
```

以上で VCL の基本的な記述は網羅しました。
補足する点がありますので次ページ以降で記述します。

*req.*の内容が bereq.*に転記されるタイミング*

いろんな場所で読み書きすることが可能な req.*ですが、バックエンドにリクエストする際に利用する bereq.*は req.*から生成されています。これが転記されるタイミングは3つあります。

1. vcl_recv で pipe を行い vcl_pipe が呼び出される前
2. vcl_recv/vcl_miss/vcl_hit で pass を行い vcl_pass が呼び出される前
3. vcl_hash を通過してキャッシュの有無判定を行いキャッシュがなく vcl_miss が呼び出される前

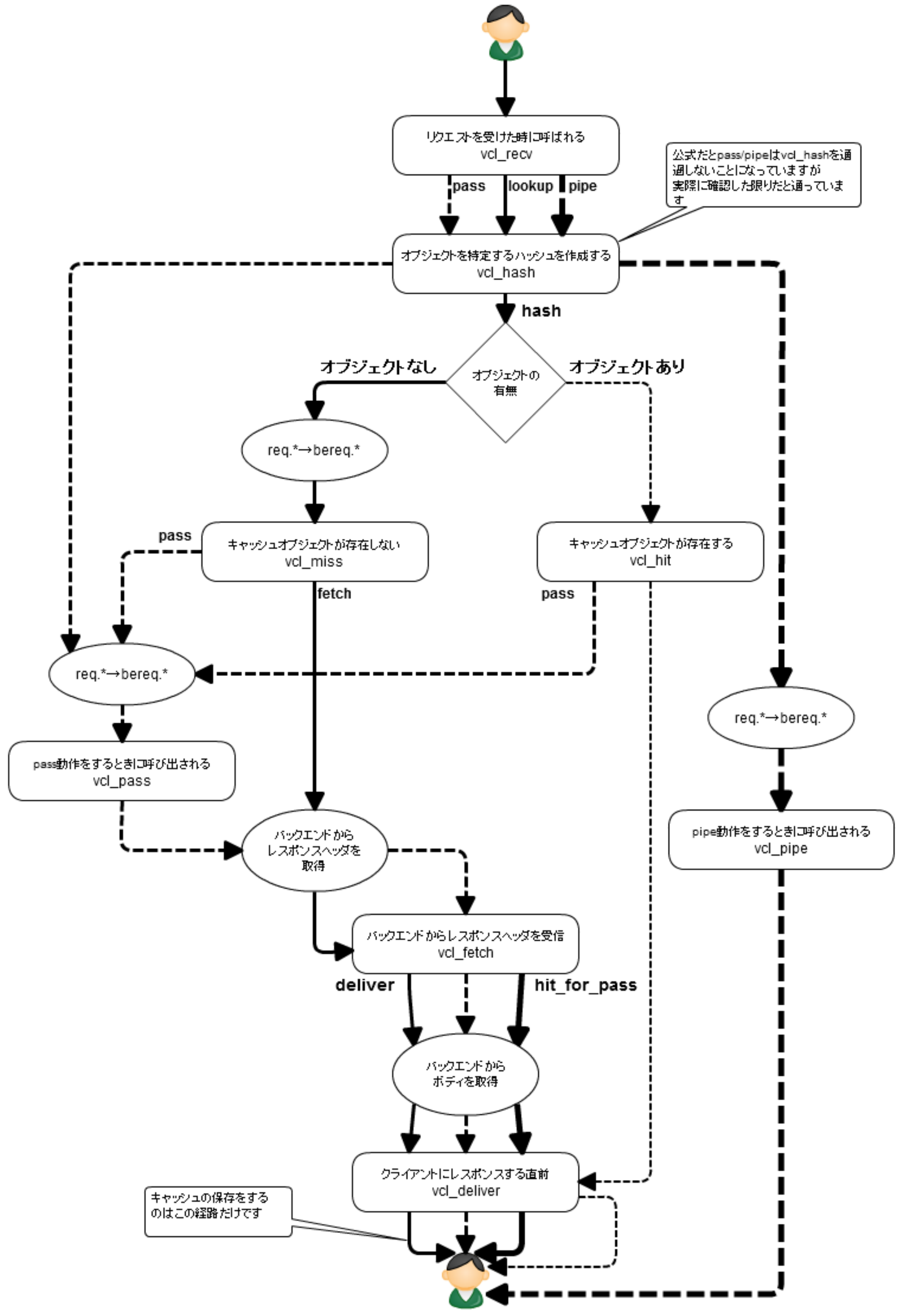
です。つまり例えば vcl_miss で req.url を書き換えたて vcl_miss で pass を行わない場合はバックエンドに問い合わせに使う bereq.url には影響しません。
また、複数回 req.*から bereq.*に転記される場合既にあったものは破棄されます。
以下のルートを通った場合について考えましょう。

1. vcl_recv で return(lookup)
2. キャッシュが見つからず vcl_miss に突入以下の値を設定して return(pass)
    1. bereq.http.X-TEST1 = 1 を設定
    2. req.http.X-TEST2 = 1 を設定

この場合、最終的にバックエンドへのリクエストに含まれるのは X-TEST2 です。
逆に pass せず fetch した場合は X-TEST1 が含まれます。

簡易版アクションフロー図

詳細図はAppendixを参照してください。



起動時オプションについて

Varnish は、ほとんどの処理は今までの多くのミドルウェアのような設定 ON/OFF では行わず、先ほど説明した VCL で行います。しかし、いくつかのオプションは存在します。例えばどのポートで受付を行うかなどの設定などです。
以下は本誌を記述している際に ps でコマンドを確認、整形したものです。

```
/usr/sbin/varnishd ¥
-P /var/run/varnish.pid ¥
-a :6081 ¥
-f /etc/varnish/default.vcl ¥
-T :6082 ¥
-t 120 ¥
-w 1,1000,120 ¥
-u varnish ¥
-g varnish ¥
-p vcl_trace=on ¥
-p max_esi_depth=2 ¥
-p esi_syntax=0x00000001 ¥
-s Transient=malloc,100M ¥
-s st1=file,/tmp/varnish_file2,200M ¥
-s st2=malloc,200M
```

オプションの動作チェックで指定しているものがあるのでそのまま使用しないでください。また/etc/init.d/varnish で起動するときのオプションの指定は

```
/etc/sysconfig/varnish
```

に記述されています。通常の指定はこちらで行います。
さて様々な指定がありますが、この章では最低限の起動オプションについて説明します。sysconfig の該当箇所を書き換える際の参考にしてください。

-a address[:port][,address[:port][...]

HTTP 要求を受け付けるインタフェース指定です。デフォルトでは:6081ですが本番で使う場合は:80を指定することが多いです。

-T address[:port]

Varnish の管理インタフェースにつなぐ指定です。非常に強力な機能（VCL の書き換えや Varnish の停止など）があるので絶対にインターネットに公開しないように注意して設定してください。

-f config

読み込む VCL ファイルを指定します。

-s [name=]type[,options]

キャッシュオブジェクトを保存するストレージを指定します。複数定義可能です。
Name は全体でユニークである必要があります。応用的な使い方で説明しますが、
指定するとオブジェクトを保存するストレージを指定できたり便利です。
指定しない場合 s0,s1,s2・・・という風に内部で名前がつけられます。
名前は最大15文字です。それ以上を指定すると16文字目以降は切り捨てられます。
また Transient という名前のストレージを Varnish がつくるので名前を指定する場合は s [数値]と Transient という名前を避けたほうが無難でしょう。
Transient の使い方については Tip's の「よくわからないけどスワップする」を参照してください。

各タイプで説明します。

file[,path[,size[,granularity]]]

ファイルにオブジェクトを保存します。ただし再起動すると消えてしまいます。
ストレージにはアクセスしますがメモリに余裕があれば OS でキャッシュされているので高速です。ただメモリサイズより大きくに指定する場合は SSD などの高速ストレージをおすすめします。
また起動時に指定サイズを確保するわけではないので事前に dd コマンドなどで初期化しておくと断片化などの影響を受けずに済みパフォーマンスに好影響を与えます。

path 保存パス
size サイズ 単位として k M G T と空き容量の割合で変化する%がある 1GB を指定する場合は1G
granularity 粒度 オブジェクトが指定サイズ(kGMT)より小さい場合は指定サイズまで切り上げます

ちなみにサイズの単位を指定しない場合はバイト単位になります。

malloc[,size]

メモリにオブジェクトを保存します。再起動すると消えます。
基本的にこれを多く利用します。

size サイズ 単位として k M G T と空き容量の割合で変化する%がある 1GB を指定する場合は1G

persistent[,path[,size]]

条件によっては再起動してもキャッシュが消えないストレージです。
実際に試してみたところ、Varnish が panic を起こして自動再起動した際や管理インタフェースから stop/start をしたときにキャッシュが消えないだけです。

path 保存パス
size サイズ 単位として k M G T と空き容量の割合で変化する%がある 1GB を指定する場合は1G

-d

デバッグモードで Varnish を起動し、そのまま管理コンソールに入った状態になります。またリクエストを受け付ける子プロセスは起動していないのでコンソール上で start とタイプして動かさないとリクエストを処理することができません。

-C

-d と-f オプションと組み合わせて指定した VCL ファイルを C 言語に解釈したときのコードを出力します。

```
varnishd -d -f /etc/varnish/default.vcl -C
```

といったように指定します。

-p param=value

様々なパラメータを指定できます。パフォーマンスに影響するチューニングやエラーを起こしたときに自動再起動するかなど様々な値を設定できます。結構危ないパラメータがあるので注意して変更する必要があります。また、複数指定も可能です。

ここではデバッグする際に便利なパラメータを紹介します。

vcl_trace on/off

VCL の実行トレースします。膨大なログがでるのでデフォルトで off です。

VCL_trace [VRT_count の第二引数] [VCL の行数],[行頭から何文字目か]

この VRT_count の第二引数ですが VCL を C に解釈した際に以下のようなコードが出力されるます。

VCL

```
行数
145     if (req.http.host) {
146         hash_data(req.http.host);
147      } else {
```

C 言語に変換時

```
738      if (
739         (VRT_GetHdr(sp, HDR_REQ, "¥005host:") != 0)
740       )
741         {
742           VRT_count(sp, 13);
743           VRT_hashdata(sp,
744             VRT_GetHdr(sp, HDR_REQ, "¥005host:"),
745             vrt_magic_string_end
746 );
```

これを実行してログを取得すると以下の出力を得られます。

```
13 VCL_trace    c 13 146.9
```

C 変換時の VRT_count の13と VCL の146行が実行されたということがわかります。また hash_data は行頭から9文字目に存在することもわかります。

また VCL_call も通常

```
12 VCL_call     c hash
```

という出力なのですがこのオプションを有効にすると

```
12 VCL_call     c hash 12 142.1
```

に変化します。付与された数値は同じ見方ができます。

vcc_err_unref on/off
定義したバックエンドや ACL を使用しないと Varnish は起動しません。しかしこのオプションを off に設定すると、定義したけど使っていないときに起こるエラーは警告となり起動できます。

Varnish のデバッグ

何事もデバッグをするにはログを見ることが重要です。しかし Varnish は、ほかのミドルウェアで見るようなファイルには出力しません。一般的にファイルに書き出す処理は非常に重いため高速化のために Varnish では共有メモリに出力しています。
そしてその共有メモリの内容をログ参照用のコマンドを使い見たり、必要に応じてファイルに保存したりします。まずはデバッグの時によく使う varnishlog について解説します。その他については後述します。

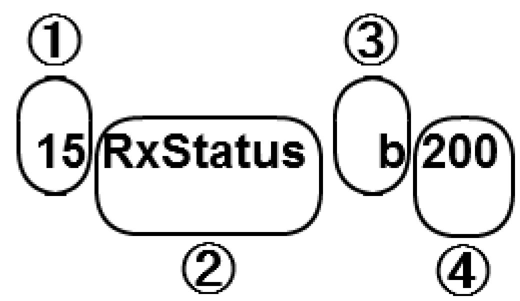
まず Varnish を起動したあと varnishlog を起動します

```
varnishlog
```

特にオプションは必要ありません。そしてブラウザで Varnish にアクセスしてみましょう。そうすると以下のようなログが出力されます。

```
（一部抜粋）
15 TxHeader     b X-Varnish: 1829024782
15 TxHeader     b Accept-Encoding: gzip
15 RxProtocol   b HTTP/1.1
15 RxStatus     b 200
```

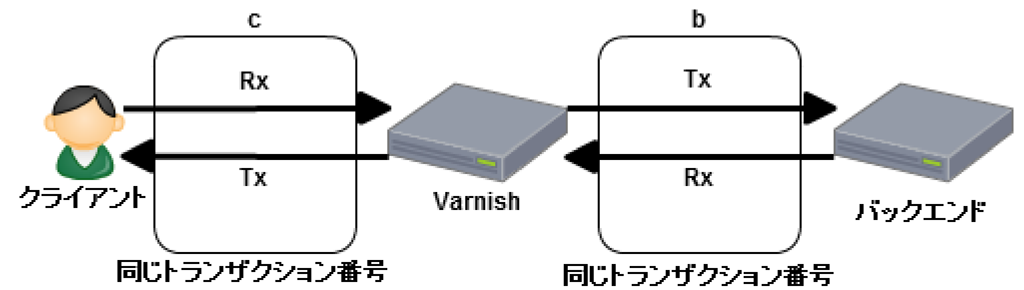
パッと見なんとなくわかるようで微妙にわからないので次ページで解説します。
基本的にスペース区切りで4ブロックに分かれています。



1. **トランザクショングループ**　**15** RxStatus　 b 200
   HTTP トランザクションのグループです。
   同じ番号は同じ HTTP のトランザクションに属します。
   あくまで HTTP なのでクライアント・Varnish 間と Varnish・バックエンド間は別の番号が振られます。

2. **メッセージタグ**　15 **RxStatus**　 b 200
   アクティビティの種別でタグが付きます。
   Prefix に Rx,Tx が付いている場合は意味があります。
   Rx・・・Varnish が受け取ったデータ(Receive)
   Tx・・・Varnish が送信したデータ(Transfer)
3. **データの流れ**　15 RxStatus　 **b** 200
   どこからデータが来たかを示します。
   c・・・クライアントからのデータ
   b・・・バックエンドからのデータ
4. **データ**　15 RxStatus　 b **200**
   実際のデータです。

図解すると以下です。



例を挙げると

12 **Rx**Header **c** Host: example.net

クライアント(c)から送出された Header を Varnish が受け取っています(Rx)

13 **Tx**Header **b** Host: example.net

Varnish がバックエンド(b)に対して Header を 送出しています(Tx)

です。

しかしこのコマンド膨大なログが出ます。1アクセスずつ確認する場合はそこまで気になりませんが運用中に確認するのはまず無理でしょう。
そのため絞り込みのオプションがあります。主に使うオプションを紹介します。

-b

Varnish とバックエンド間のトラフィックのみ出力します。
例えば、キャッシュできるはずなのに何故かキャッシュできていない際の調査に役に立ちます。

-c

Varnish とクライアント間のトラフィックのみを出力します。

-m tag:Regex

指定タグとメッセージが正規表現とマッチするトランザクションを出力します。また複数指定可能です。（V3.0.0 では二個までしか動作確認できませんでした）
例えば、/test で始まるクライアントからのリクエストをステータス200で返せたトランザクションを出力するとこうなります。

```
varnishlog -m 'RxUrl:^/test.*$' -m 'TxStatus:200'

13 SessionOpen  c 192.168.1.123 37131 :6081
～中略～
13 RxURL        c /test
13 RxProtocol   c HTTP/1.0
13 RxHeader     c User-Agent: Wget/1.11.4 Red Hat modified
～中略～
13 TxStatus     c 200
～中略～
13 ReqEnd       c 1266506662 1311094899.230543137 1311094904.424874067
0.000063658 0.788361788 4.405969143
```

-i tag (-x tag)

-i は特定のタグのみを出力します。複数指定可能です。-m と違いトランザクションまるごとではないので注意してください。
また、-x はその逆でマッチするものを除外します。

-I Regex (-X Regex)

-I は正規表現でマッチしたデータの行のみ出力します。一つのみ指定可能です。例えばクライアントから送出されるクッキーのヘッダのすべてを出力したい場合は以下のようなコマンドになります。

```
varnishlog -c -i RxHeader -I Cookie
```

-d

通常 varnishlog は起動してからの情報しか出力しませんが、このオプションを指定すると共有メモリに残っているログをすべて表示します。何かしら問題が起きてすぐであれば運良くログが残っているかもしれません。

```
varnishlog -d -c -m TxStatus:503
```

上記の例では過去ログから503を出力したリクエストを取得します。

さて、デバッグをする際にもう一つよくやることがあります。
コード中にログを出力する関数を書いて、その時点で操作している変数がどのような値を示しているかなどを確認することです。行い方を解説します。

しかしVarnishは標準ではログ出力を行うことはできません。行うには別途
モジュールを読み込む必要があります。このモジュールの仕組みはVMODと呼ばれておりバージョン3より実装されました。
そしてVarnish本体と一緒に配布されているstdというVMODを使うことでlog出力が可能です。一緒に配布されているので特にインストールの際に別の手順は必要ありません。早速使ってみましょう。

```
import std; //ファイルの先頭
～中略～
sub vcl_recv{
        std.log("RequestUrl:" + req.url);
        return(lookup);
}
```

これをvarnishlogでログを取得してみると以下の出力を得られます。

```
13 VCL_Log      c RequestUrl:/b
```

非常に簡単です。
また他にデバッグの際によく行うのがどのルートで実行されたかの確認ですが。
もちろんlog出力で行っても構いませんがスマートではないので先ほど紹介した
起動オプションで

```
-p vcl_trace=on
```

を使い確認するのが良いでしょう。

また、-Xはその逆でマッチするものを除外します。

以上で基本的なVarnishの使い方を説明しました。
次ページから運用時に実際に使ったり、知っておくと便利な応用的な記法などを説明します。

応用的な使い方

管理コンソールの利用方法

Varnish を CLI で管理するツールに varnishadm があります。同一のサーバの場合はオプション無しで実行することで、そのままコンソールに繋ぐことができます。
まずは help と打ってみましょう。使えるコマンドの一覧が出力されます。

```
help
200
help [command]
ping [timestamp]
～中略～
```

コマンドの後に表示される数字（上記では200)はコマンドを実行した成否です。
200の場合は受付され処理されます。それ以外は何らかのエラーがあります。
また、よくわからないコマンドについては以下のようにすることで説明が見れます。

help **<コマンド名>**

```
help vcl.list
200
vcl.list
        List all loaded configuration.
```

管理コンソールは非常に強力なツールです。よく使うコマンドについて解説します。

param.show 現在のパラメーター覧を表示する

現在のパラメータの設定値を見ることができます。また以下のように指定することでそのパラメータの説明も含めてみることができます。

param.show **<パラメータ名>**

```
param.show vmod_dir
200
vmod_dir                /usr/lib64/varnish/vmods
                        Default is /usr/lib64/varnish/vmods
                        Directory where VCL modules are to be found.
```

param.set パラメータの設定を行う

パラメータを動的に設定することができます。一部パラメータについては後述する子プロセスの再起動が必要だったりしますが、ほとんどが即時反映します。

param.set **<パラメータ名> <値>**

```
param.set shortlived 5s
200
```

stop 子プロセスの停止を行う

Varnish は管理する親プロセスとリクエストを受け付ける子プロセスが存在します。その子プロセスの停止を行います。子プロセスの再起動が必要なパラメータを変更した際などに使います。

start 子プロセスの開始する

先程の stop で停止した、もしくは何らかの原因などで止まってる子プロセスを開始します。

panic.show 最後に子プロセスがパニックを起こした際のログを出力

Varnish の子プロセスが何らかの原因でパニックを起こして再起動を起こした際に追跡を行うためのログを採取します。原因となったリクエストなども採取できます。

```
panic.show
200
Last panic at: Wed, 20 Jul 2011 16:42:38 GMT
Panic from VCL:
  PANIC:  panictest
thread = (cache-worker)
～中略～
```

ban キャッシュの ban を行う

VCL の時の Varnish と同じように ban が行えます

ban **<ban を行う際の評価式>**

```
ban req.http.host == "example.net" && req.url ~ "^/image/"
```

ban.list アクティブな ban のリストを表示する

ban は即時に消去せずにリストに貯めておきリクエストが来た際に評価しきゃキャッシュの消去を行います。そのリストの確認ができます。大量の ban がエントリされているとパフォーマンスに若干の影響を与えることがあるので、大量に ban を行うことが多い場合はたまに確認するとよいでしょう。

```
ban.list
200
0x2acd1b11f280 1311182960.361679     0  req.http.host == example.net &&
req.url ~ ^/image/
```

なお RPM で配布されている varnishadm は readline が使えません。
（入力済み文字列を矢印キーで再度出す機能）
そのため rlwrap というコマンドを同時に使うことで解決できます。

```
rlwrap varnishadm
```

rlwrap が見つからない場合は yum でインストールすることができます。

インライン C

VCL の説明をしたときに C 言語を中に埋め込むことができると紹介しましたが、その方法について解説します。
C のコードを挿入するには VCL 中の任意の箇所に以下のように記述します。

```
C{
任意のコード
}C
```

非常に簡単です。
実際に行いたい処理はそれぞれなので、例よりもポイントを説明します。

基本的に習熟は VCL をダンプして

公式のドキュメントにはインライン C の詳細なドキュメントはありません。ではどのように使える関数などを調べるのでしょうか？
基本的に VCL を C 言語にダンプして覚えていくのが速いです。先程の起動オプションで解説したとおり。

```
varnishd -d -f [VCL ファイル] -C
```

で C 言語をダンプして書いてある内容で調べていきましょう。また巻末の Appendix において以下を紹介しています。

VCL で利用する各型を STRING に変換する関数
　　内部では IP 型や TIME 型等様々ありますがそれの全変換関数です。
インライン C 利用時にヘッダ操作をする関数
　　req.http.*を操作するときの関数と操作の仕方です。
VCL 変数一覧
　　req.url など全ての変数の使えるアクションとインライン C 時の関数一覧です。

include は忘れずに

VCL と混在なので忘れがちなのが include です。利用する際は VCL の先頭で

```
C{
#include <stdlib.h>
#include <stdio.h>
}C
```

きちんと書きましょう。

libmemcachedやlibxml2など共有ライブラリを使う方法

インラインCで書くだけで済む処理なら問題ないのですが、様々な共有ライブラリを使いたいことがあるかと思います。その際はcc_commandという起動パラメータを使用します。この起動パラメータは、VCLをコンパイルする際に利用するコマンドとなります。libmemcachedを利用する場合はここに-lmemcachedと加えます。デフォルト値は環境によって異なりますので、先ほど解説した管理コンソールからparam.showでデフォルトを取得してから行ってください。筆者の環境では以下のように指定しました。

```
-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' ¥
```

後はインラインCで呼び出すためのコードを書くことで利用することができます。
また当然ですが、デバッグなどでコマンドで叩く際はこのオプションを忘れずに指定してください。共有ライブラリが読み込めず、undefined symbolがでてハマることがあります。

統計情報の見方

Varnishで実運用をするうえで統計情報を見るのは欠かせません。その際に使うツールを簡単に解説します。

varnishtop

varnishlogでの一行ごとにカウントをとっていきランキングを作ります。

```
list length 146

    4.96 VCL_return      deliver
    4.79 CLI             Rd ping
    4.62 VCL_Log         reqlocalhost.localdomain
    3.71 TxProtocol      HTTP/1.1
```

よく使うオプションの組み合わせは以下です。

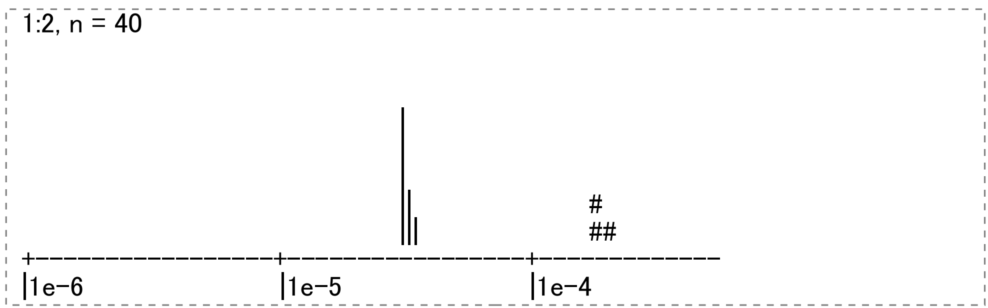
varnishtop -i rxurl
　クライアントから多くリクエストされたURLがわかります。

varnishtop -i txurl
　バックエンドに多く要求しているURLがわかります。キャッシュ対象にもかかわらず重複が多い場合上手くキャッシュ出来ていないので確認が必要です。

Varnishtop -i txstatus
　クライアントに返却したステータスコードがわかります。400以上のステータスが多い場合は異常が起きていないか確認が必要です。

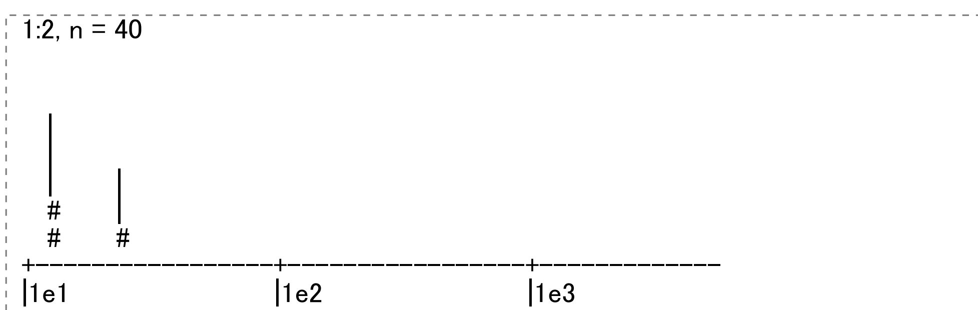
varnishhist

直近ｎアクセスの応答時間を図で表示したものです。



[|]はヒットしたアクセス[#]はミスヒットしたアクセスになります。
縦軸はアクセス数で比率は左隅に出力されています。
（図では1:3なので1キャラクタが3アクセス）
横軸は応答時間です。1 e-6 は10^{-6}～10^{-5}つまり1μ秒～10μ秒を表します。
全体的にレスポンスが遅いようであればチューニングなどの検討の対象となります。

varnishsizes



先程の varnishhist はレスポンスタイムでしたが varnishsizes はレスポンスサイズです。
横軸はサイズとなり、 1 e2 は10^{2}～10^{3}つまり1 kB～10kB を表します。

varnishstat

Varnish は様々なカウンタを保持しています。ミスヒットした件数、バックエンドへのリクエスト数、今までの転送サイズなどいろいろです。実運用を行っている際にチューニングを行うかの検討材料に使うことが多いです。

```
0+00:30:35 ←アップタイム
Hitrate ratio:      10      100      145      ←下のヒット率を計算した秒数範囲
Hitrate avg:    0.8663   0.8590   0.8490   ←秒数毎のヒット率
           ↓生データ        ↓起動してからの秒数での平均
           64        0.00     0.03     client_conn - Client connections accepted
           64        0.00     0.03     client_req - Client requests received
           63        0.00     0.03     cache_hit - Cache hits
           6         0.00     0.00     cache_miss - Cache misses
           6         0.00     0.00     backend_conn - Backend conn. success
           6         0.00     0.00     fetch_length - Fetch with Length
           10        .        .        n_sess_mem - N struct sess_mem
           1         .        .        n_object - N struct object
           3         .        .        n_objectcore - N struct objectcore
           3         .        .        n_objecthead - N struct objecthead
           2         .        .        n_waitinglist - N struct waitinglist
                     ↑直近一秒での値
                            ～中略～
```

ヒット率は以下の場所を見ます。

```
Hitrate ratio:      10      100      145
Hitrate avg:    0.8663   0.8590   0.8490
```

直近10秒の時は86.6%、100秒は85.9%、145秒は84.9%を示します。
他に、代表的な見ておくと良い数値を解説します。

N worker threads not created
　スレッドを新規につくろうとしたが、作れなかった回数。出ないのが望ましい。
N worker threads limited
　スレッドプールの最大値に引っかかって新規に作れない。出ないのが非常に望ましい。
N dropped work requests
　処理を諦めたリクエスト数

これらの値はスレッドに関する値で、だいたいプール数の調整を行うことで改善します。初期プール数を大きめにするのも有効です。

ヒット率
　高いほどよいです。
N LRU nuked objects
　オブジェクトが TTL を迎える前に削除された件数。ストレージサイズが小さいと起きる。

これらの値はキャッシュポリシーを考える際に利用します。
あくまで一例ですが、例を上げます。

ヒット率は高く nuked は少ない

理想の状態です。もし少し nuked が出ているのであればキャッシュするオブジェクトが増えるとバックエンドへの問い合わせが増える可能性があるので注視しましょう。

ヒット率も nuked も高い

ストレージサイズが足りない状態です。キャッシュオブジェクトが今後増える場合ヒット率も下がりやすいのでサイズを増やすことを検討しましょう。

ヒット率も nuked も低い

そもそもキャッシュするべきオブジェクトなのかの検討が必要です。またキャッシュ出来るはずなのに起きている場合は VCL の見直しが必要です。例えば正規化が可能な余計なクエリが付いていたりする可能性があります。

他にも様々なカウンタがあるのでうまく使いチューニングを行いましょう。

ストレージサイズと *TTL* の決め方

ストレージサイズと TTL を決めるのはなかなか難しいです。
エイヤと128 GB のメモリを搭載しているサーバで全て割り当てて満足、というのもコスト度外視でよければ悪くはないのですが、実際はなかなか難しいです。
サイズを決める際のポイントを説明します。

良くアクセスされるオブジェクトはどの程度か？

例えば各ページに埋め込まれている JS や CSS、ロゴ画像はよくアクセスされるので最低限それ以上のサイズは指定しましょう。またポータルサイトのような形態であれば、TOP や各メインコンテンツからリンクされているページを全て格納できるサイズを用意しておくとよいでしょう。

どのようなアクセスパタンか？

前項と少しかぶるのですが、例えばニュースサイトにおいては最新の記事は非常にアクセスが多いことが想定されます。しかし一ヶ月すぎたニュースはさほどアクセスはされないでしょう。アクセス解析などでどの程度で PV が減っていくかを見てアクセスの多い期間はキャッシュできるように TTL を設定するとよいでしょう。

そのオブジェクトを作成するコストはどれだけか？

例えば秒間10回と1回、アクセスされるオブジェクトがそれぞれあるとしましょう。単純に考えれば10回アクセスされるものをキャッシュするべきです。しかし秒間1回アクセスを受けるオブジェクトがプログラムで作成し CPU を相当使い、秒間10回アクセスを受けるオブジェクトは静的コンテンツの場合、秒間1回のアクセスしかなくても意識してキャッシュするべきでしょう。

ヒット率の上げ方

ちょっとした工夫でヒット率は上がり、ストレージサイズも節約できますので紹介します。

何がなんでも正規化

これに尽きます。正規化を行う際の見ておきたいポイントを解説します。

ドメイン名の正規化

VCL の説明の箇所でも記述しましたが、重要なので再度説明します。
www.example.net と example.net が同じ内容を返すのであればどちらかに寄せるべきです。

```
if(req.http.host ~ "(?i)(www¥.)?example¥.net"){
        set req.http.host = “example.net”;
}
```

QueryString の正規化 1

例えば URL 中に特にコンテンツの内容にかかわらないパラメータが存在する場合は消してしまうのも手です。

```
sub vcl_recv{
        set req.url = regsub(req.url, "(¥?|&)xxx=[^&]+&?(.+)?$", "¥1¥2");
        set req.url = regsub(req.url, "(¥?|&)$", "");
        return(lookup);
}
```

ここでは xxx=のパラメータが不要として削除するようにしています。
また、ここで利用した regsub ですが正規表現を利用した置換関数です。
以下のように利用します。

```
regsub(置換対象,正規表現,置換式)
```

正規表現でキャプチャしたものは¥1,¥2といったように指定します。また regsub は最初のマッチしたものだけの置換ですが全て置き換える regsuball もあります。

QueryString の正規化 2

Varnish では以下の場合でも異なるオブジェクトとします。

```
/Test?aaa=1&bbb=1
/Test?bbb=1&aaa=1
```

aaa と bbb の順番が違うだけです。インライン C を使ってソートすることも可能ですが、そもそも HTML やテンプレート側でリンクの際の QueryString はソートしてから渡すことで解決します。

クッキーの正規化

クッキーについてもQueryStringと同様の理由で正規化する必要があります。
もしくは必要ないのであれば削除するべきでしょう。

Varyに注意する

Varyヘッダは同じURLで異なる内容を返す際に、何を判断すればいいのかをプロキシに伝えるものです。

```
Vary: User-Agent
```

上記の場合User-Agentが違う場合異なるキャッシュで保存しようとします。おそらくこれの期待はIE・FireFox・Chromeなどのブラウザの種類で切り替えることですが、IEでどれだけUser-Agentがあるのでしょうか？OSのバージョンや.NETのバージョン、ツールバーが情報を付与したり同じバージョンのIEだとしても10じゃきかないでしょう。そのため注意して正規化をする必要があります。

また正規化以外の方法としては、TTLを引き上げるなどの方法、次に説明するESIとgzipがあります。

Edge Side Includes (ESI)

本誌の最初の動的コンテンツの配信のところで触れているとおり、非常に強力な機能です。ページ自体の高速化・システム全体の負荷の軽減が期待できます。
ただ、高速化にすごい期待しないでください。いれれば即100倍になった！という単純なものではないのです。どのようなものでもそうなのですが、システムは一番遅いところの速度に落ち着きます。ESIでは、以下の場合苦手です。

- ページを生成する際に絶対キャッシュできないものがある
- そのコンテンツの取得が遅い
- コンテンツの取得がページ全体の処理時間の大半を占めている

具体的には当てはまるのは何でしょうか？一例として携帯サイトでの広告です。携帯サイトの場合、基本的にJSは使えないのでどうしても広告を取得してページに埋め込む必要があります。その処理は毎回行わないといけないので、その遅さに引きづられます。もちろんその場合でもシステムの負荷の軽減は期待できるので検討するのもよいでしょうが、何がなんでもESIで解決ではないということを意識してください。

さて、実際の使い方です。vcl_fetch内でberesp.do_esiをtrueにします。

```
sub vcl_fetch{
        set beresp.do_esi = true;
}
```

これだけで有効になります。
そしてコンテンツ側の記述ですが以下のように記述します。

<esi:include src="*path*" />

指定したパスを展開します。

```
/index.html
<html>
<body>
↓<br />
<esi:include src="/esi/1.html" />
↑<br />
</body>
</html>

/esi/1.html
Hello ESI!<br />
```

上記を実際にみると

```
<html>
<body>
↓<br />
Hello ESI!<br />
↑<br />
</body>
</html>
```

と返却されます。

<esi:remove>～</esi:remove>

ESI 実行時に削除される範囲です。タグで囲まれた範囲を削除します。
つまり ESI が有効でない場合にそのまま表示されます。<esi:XXXX>といった記述は特に HTML の表示に悪影響はないので一時的に ESI を止めてる時にメッセージを出すときに利用します。

```
記述
→<esi:remove>hoge</esi:remove>←

ESI 有効時
→←

ESI 無効時
→<esi:remove>hoge</esi:remove>←
```

<!--esi ～ -->

ESI 処理が行われた際に表示される項目です。無効時はコメントのままです。

```
記述
→<!--esi hoge -->←

ESI 有効時
→ hoge ←

ESI 無効時
→<!--esi hoge -->←
```

ESI の規格では他のタグも存在するのですが Varnish がサポートしているのは以上の記述のみです。

また、ESI 利用時の注意事項について説明します。

デフォルトでは HTML/XML タグ形式ではないと解釈されない

include の例できちんと<HTML>～と記述しましたが例えば内容を include のみにした場合は動作しません。どうしても動作させたい場合は起動パラメータで設定を行います

```
-p esi_syntax=0x00000001 ¥
```

このオプションを指定するとタグが存在しなくても ESI の構文を探します。

ネストするページで<html>や<body>を出力するとそのまま出力される

ESI では特に中身について変更を行わず単純に埋込みを行います。つまり include 元と先で両方共<html>や<body>タグがあると両方共出力されてしまいます。

ネスト数は制限がある

ESI した先でさらに include の記述があってもネストすることはできますが
デフォルトでは最大ネスト数は5です。変更したい場合は起動パラメータで以下のように指定します。

```
-p max_esi_depth [最大ネスト数] ¥
```

なおネスト制限に引っかかって展開されない場合でも include はそのまま残るわけではなく削除されます。

ページの ESI 化を行う際は再起動かページの ban を行うのが望ましい

詳しくは gzip で解説しますが、Varnish が ESI のタグを解釈するのは fetch した時なので既にキャッシュされている場合はキャッシュが消えるまではそのオブジェクトを ESI せずにそのまま返却します。

do_esi を外すだけでは ESI 処理を OFF に出来ない

有効化の時と同様の理由です。ESI 処理が入った場合、単純に

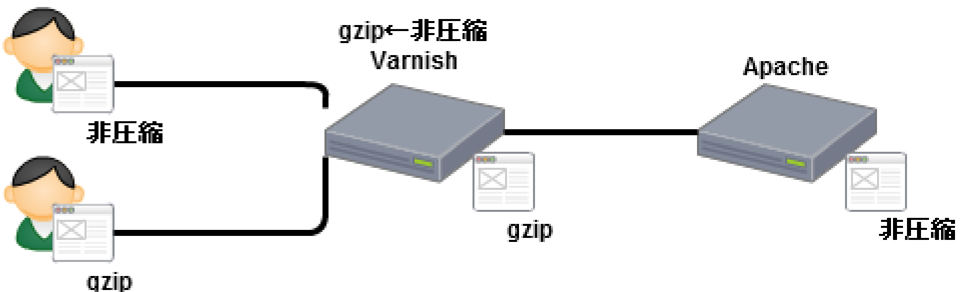
```
beresp.do_esi = false
```

では止めることはできません。フロー図を思い出してください、ヒットしたオブジェクトは fetch を通過しません。オフにする場合は vcl_recv で以下を指定してください。

```
set req.esi = false;
```

gzip の利用

Varnish3 では gzip をサポートしています。これはどういう事かというと、サーバから非圧縮のオブジェクトを取得して、gzip 圧縮を行いキャッシュし、そしてクライアントが Accept-Encoding で gzip をうけとれるのであれば gzip を送信し、そうでなければキャッシュ内の gzip を解凍して送信することができます。



これには3つの利点があります。

まず単純にキャッシュの効率的な利用です。Varnish2.1 までは同じ内容を gzip で要求するクライアント、非圧縮を要求するクライアントが混在で両方対応する必要がある場合、Varnish は両方キャッシュする必要がありました。しかし、Varnish3 ではレスポンス生成時の gzip 圧縮・解凍が可能になりました。これによりキャッシュされるオブジェクト数は半減する上、キャッシュも圧縮された状態で保存されるのでより多くのオブジェクトを保存できます。

次に、帯域の節約です。画像がメインの場合はそうでもありませんが、CSS や JS、HTML を大量にキャッシュするようなサイトの場合はかなり効いてくるはずです。

最後に ESI です。2.1までは include する要素が全て非圧縮な必要がありました。解凍できなかったためです。そうすると ESI で組み立てを行ったページは自ずと非圧縮になってしまい、クライアントが gzip の受付ができるにもかかわらず非圧縮で返さざるを得ませんでした。しかし3.0では圧縮も解凍もできるため、2.1までの問題は起きません。

では使い方を説明します。

```
sub vcl_fetch{
        set beresp.do_gzip = true; //圧縮されていない場合圧縮する
        //set beresp.do_gunzip = true; //圧縮されている場合解凍する
}
```

これで全ての非圧縮オブジェクトはキャッシュへ保存時に gzip されます。
非常に便利な gzip ですが2点注意があります。

CPU を使います
当然ですが圧縮・解凍には CPU を使います。おそらく気にならないレベルですが環境によっては注視する必要があるでしょう。

ESI を利用する際は Varnish・バックエンド間は非圧縮で
必須ではありませんがバックエンドからのレスポンスは非圧縮が望ましいです。
順に考えてみましょう。
Varnish が ESI のタグを解釈するタイミングは fetch した時です。解釈を行い、234 バイトスキップしてその後234バイトはそのまま、次にファイルを include して・・・といった特殊な情報を付与します。このプロセスは当然ながら gzip 圧縮された状態では動作しないため解凍して行われます。

レスポンスが圧縮状態の場合
レスポンス→**解凍**→ESI 解釈→その後の処理

レスポンスが非圧縮状態の場合
レスポンス→ESI 解釈→その後の処理

そのためこのような記述を行ってください。

```
sub vcl_miss{
        if (ESI 処理を必要とするオブジェクト){
                unset bereq.http.accept-encoding;
        }
}
sub vcl_fetch {
        if (ESI 処理を必要とするオブジェクト){
                set beresp.do_esi = true;
                set beresp.do_gzip = true;
        }
}
```

これによりバックエンドから取得する際は非圧縮で、キャッシュに保存する際は gzip されている状態になります。

VMOD の追加の仕方

Varnish には VMOD という機能拡張のためのモジュールが存在します。標準では std というモジュールが存在しログ出力などが可能です。
しかし、他に配布されているモジュールを実際に使うにはどうすればいいでしょうか？行い方を解説します。
今回は Kristian Lyngstøl(@kristianlyng)さんの、クッキーなどのヘッダ操作を行う vmod_header で試してみます。VMOD のコンパイルには Varnish のソースが必要です。インストールしているバージョンの Varnish のソースと VMOD のソースをダウンロードします。

github

```
https://github.com/KristianLyng/libvmod-header
```

解凍～インストール

```
tar zxf KristianLyng-libvmod-header-5b6fe2b.tar.gz
tar zxf varnish-3.0.0.tar.gz
cd KristianLyng-libvmod-header-5b6fe2b
./configure VARNISHSRC=/root/temp/varnish-3.0.0 ¥
            VMODDIR=/usr/lib64/varnish/vmods
make
make install
```

オプションの VARNISHSRC は必須です、ソースを解凍した場所を指定してください。VMODDIR は必須ではありませんが指定しておくといいです。
場所は以下で調べられます。

```
[root@localhost ~]# varnishadm param.show vmod_dir
CLI connected to 0.0.0.0 6082
vmod_dir                 /usr/lib64/varnish/vmods
                         Default is /usr/lib64/varnish/vmods
                         Directory where VCL modules are to be found.
```

ちなみに上記のように varnishadm は引数にコマンドを入れることも可能です。
また make 時に以下のようなメッセージが出力されエラーになることがあります。

```
============================================
You need rst2man installed to make dist
============================================
make[2]: *** [vmod_header.3] Error 1
```

その場合は python-docutils を入れる必要があります。

```
yum install -y python-docutils
```

その後は make clean を行い再度 configure から行ってください。

次は実際にVCLを記述します。

```
//先頭付近に
import header;
～中略～
sub vcl_fetch {
        header.append(beresp.http.Set-Cookie,"foo=bar");
        header.remove(beresp.http.Set-Cookie,"dontneedthiscookie");
}
```

この後varnishをリロードか再起動を行い試してみてください。

```
---response begin---
HTTP/1.1 200 OK
～中略～
Set-Cookie: foo=bar
～後略～
```

きちんと追加されていることが確認できます。

またこのモジュールですが、クッキーの操作に便利なのでぜひmanで使い方を見てみてください。

```
man vmod_header
```

なおVarnishのバージョンが上がるとリビルドが必要なことがあります。注意してください。

ストレージの高度な制御

Varnishのキャッシュストレージは複数指定することができますが、そのままだとどこのストレージに保存するかは不定です。しかし、あんまりヒットしないものはfileに凄いアクセスされるのはmallocに・・・としたいこともあるかもしれません。

ここではストレージの制御の方法を解説します。

まず最初に、起動オプションのストレージ指定においてユニークな名前をつけます。

```
-s st1=malloc,1G ¥
-s st2=file,/tmp/varnish_file,1G ¥
```

太字の箇所がユニークな名前です。

次にURLのパターンによって格納先を変えます。

```
sub vcl_fetch{
        if(req.url ~"^/s1/")     { set beresp.storage = "st1";}
        else                     { set beresp.storage = "st2";}
}
```

パスが/s1/の時はst1、それ以外はst2に保存します。

次は残り容量をチェックして保存先を切り替えます。

```
sub vcl_fetch{
        if(storage.st1.free_space > 100MB)      { set beresp.storage = “st1”;}
        else                                    { set beresp.storage = “st2”;}
}
```

上記では、ストレージの残り容量が100 MB を越えている場合は st1、それ以外は st2 となります。また現在 free_space が使えるのは malloc だけのようで file,persistent で指定しても有効な値は帰ってきません。

バックエンドの高度な利用

特定のバックエンドにアクセス出来ないときにのみ別のバックエンドを使いたいときはどうすればいいでしょうか？以下解説します。

```
sub vcl_recv{
        set req.backend = b1;
        if( ! req.backend.healthy ){
                set req.backend = b2;
        }
}
```

ヘルスチェクをしていないといけませんが req.backend.healthy を使うとバックエンドの状態がわかります。
この例では、b1 が異常の場合は b2 を利用します。

varnishtest の使い方

Varnish の VCL は非常に複雑な処理も可能な言語です。そのためテストが重要になってきます。varnishtest はテストの定義を予め作って置くことで、VCL 変更時のデグレードテストも手軽に行えます。

基本的な定義

```
# バックエンドサーバを定義する　名前は s1 とする
server s1 {
   # リクエストを受ける
   rxreq
   # レスポンスする
   txresp -hdr "Connection: close" -body "012345¥n"
}
# バックエンドサーバを起動
server s1 -start

# varnish の定義　名前は v1 とする
varnish v1 -arg "-b localhost:${s1_port}"
```

```
# クライアントを定義する　名前はc1とする
client c1 {
    # リクエストする
    txreq -url "/"
    # レスポンスを待つ
    rxresp
# ステータスを評価する 200ならOK
expect resp.status == 200
}

# varnishを起動
varnish v1 -start
# クライアントを動かす
client c1 -run
# リクエストを待つ
server s1 -wait
```

実行は

varnishtest -v ***[filename]***

です。-v トレース結果を出力するオプションで必須ではありません。

流れとしては

1. Varnish とバックエンドとクライアントの定義をする
2. Varnish とバックエンドを起動する
3. クライアントを Run してテスト実行
4. 後処理

です。最低限覚えておきたい記述について解説します。

${*xxx*_port} ポート指定（xxx はサーバ名）

基本的にサーバのポートは自動的に付けられます。
そのため複数のバックエンドサーバを定義したりVarnishを定義した場合に指定がしづらいといったことがあります。その際にこの変数を使うことで解決できます。
またアドレス版の${xxx_addr}もあります。

expect *[変数] [比較演算子] [変数]*

様々な比較を行い一つでも失敗したらfailになります。
トレース中はこのような表記で出力されます。

```
成功 expect resp.status == 200
**** c1    0.3 EXPECT resp.status (200) == 200 (200) match
失敗 expect resp.status == 202
---- c1    0.3 EXPECT resp.status (200) == 202 (202) failed
```

rxreq
リクエストを受けるのを待ちます。

txresp -hdr *"[ヘッダー]"* -body *"[内容]"*
サーバがレスポンスする際に利用します。オプションはこれだけではありませんが主に使うのは hdr と body です。

txreq -req *[メソッド]* -url *"url"* -hdr *"[ヘッダー]"*
クライアントがリクエストする際に利用します。メソッドは GET や POST を指定します。

rxresp
レスポンスを待ちます。

delay *[秒数]*
指定秒数次の処理を待ちます。
秒数指定は小数点が指定可能です

server *[name] {振る舞い方} [option]*
バックエンドサーバの定義でリクエストを受け取った際にどのような振る舞いをするか定義します。オプションには以下があります。

-start サーバを起動
-wait リクエストを待機

client *[name] {振る舞い方} [-run]*
バックエンドサーバの定義で、リクエストを受け取った際にどのような振る舞いをするか定義します。-run を指定するとクライアントを動作させます。

varnish *[name]* -arg *"起動オプション" {振る舞い方} [-option]*
varnish を定義します。arg で起動オプションを指定します。その際にテストしたい VCL を指定すると良いでしょう。オプションには以下があります。

-start サーバを起動
-stop サーバを停止
-wait リクエストを待機

さらなる使い方については varnish のソース中に含まれる*.vtc ファイルを参照してください。
例えば rxresp,rxreq を上手く使うことでサーバが複数回レスポンスを返したり、クライアントが複数回リクエストするような記述も可能です。

Varnish にやさしい *ban* の仕方

ban は実行されたタイミングで即削除を行わず、リストに ban の条件式を登録します。そして lookup、もしくは実際の削除を行っているスレッド(ban lurker thread)が定期的に見て回り、式が True になったものがあれば削除を行います。
ここで以下の ban について考えてみましょう。

```
ban req.http.host == "example.net" && req.url ~ "^/image/"
```

リクエストの host と url を評価しています。リクエストの変数(req.*)が生成されるのはリクエストを受けたときだけです。つまりこの ban が実行されるのは lookup されるときにしか行われません。条件に合うリクエストが来るまでリストに残ります。ほんの数件であれば問題ないのですが大量にリストされていると、
パフォーマンスに影響が出てきます。何とかして 削除しているスレッドに回収させたいのですが、そのためには req を使わない方法でないといけません。
そこで以下の方法でキャッシュを表す obj 変数を使うことで回避可能です。

```
vcl
sub vcl_fetch{
        set beresp.http.X-URL = req.url;
        set beresp.http.X-HOST = req.http.host;
}

ban
ban obj.http.X-HOST == "example.net" && obj.http.X-URL ~ "^/image/"
```

vcl_fetch で beresp に.X-URL/HOST を入れているのは、obj 変数が beresp の情報を維持する為です。beresp には.url は存在しませんし、レスポンスには通常は Host ヘッダ含まないため、beresp.http.host もありません。しかし req 変数を使わないためにはその上方が必要なため.X-URL と.X-HOST に退避しています。こうすることで obj で url と host を指定できます。実際に実行してみた結果です。

```
  PID USER      PR  NI  VIRT  RES  SHR  COMMAND
15734 varnish   20   0  440m  18m  988  varnishd     //起動状態
15734 varnish   20   0  504m  78m 1208  varnishd     //56M ファイルをキャッシュ
15734 varnish   20   0  504m  78m 1244  varnishd     //req.*で ban して数秒後
15734 varnish   20   0  444m  20m 1300  varnishd     //obj.*で ban して数秒後
```

最初の56 MB のファイルをキャッシュの際しかアクセスしていませんが obj で ban すると回収されるのがわかります。
なお、レスポンスヘッダには.X-URL と.X-HOST が含まれます。必要ない場合は vcl_deliver で unset してください。
また、obj.*変数の詳細については Appendix を参照ください

Tip's

X-Varnish ヘッダってなに？

Varnish が返却するヘッダに X-Varnish というものがあるがこれはなんでしょうか？
出力パタンは2つあります。

- X-Varnish: 1393335468
- X-Varnish: 1393335498 1393335468

実は数字の部分がひとつの場合はバックエンドに取得しに行った物で、2つある場合はキャッシュから返却したものです。数値はトランザクション ID です。

- X-Varnish: [トランザクション ID]
- X-Varnish: [今回のトランザクションの ID] [キャッシュする際に使ったトランザクション ID]

この ID はバックエンドにリクエストする際も付与されており、バックエンド側で X-Varnish をログしている場合どのリクエストがキャッシュされたかが分かるためデバッグがしやすくなります。

リダイレクトはどうやるの？

Varnish でリダイレクトを行うのは少し厄介です。基本は error で処理を vcl_error に飛ばしそこで Location を設定する形で行います。例えば存在しない HTML ファイルへのアクセスをトップページに飛ばす場合はこのように行います。

```
sub vcl_fetch {
	if (beresp.status == 404 && req.url ~ "¥.html?$") {
		error 750 "http://example.net/";
	}
}
sub vcl_error {
	if (obj.status == 750) {
		set obj.http.Location = obj.response;
		set obj.response = "Fonund";
		set obj.status = 302;
		return(deliver);
	}
}
```

ポイントはエラーに飛ばす際にステータスを実際には使われないコードを指定する事と（この場合750）レスポンスにリダイレクト先を含めることです。
もちろん vcl_fetch 内だけではなく vcl_recv などの他のアクションでも使用できます。

SSL について

Varnish は SSL に対応していないため前段に Nginx などを置くなどする必要があります。

画像が404の際にデフォルトの画像を出す方法

vcl_fetch において該当ファイルが404の存在有無がわかるので、実行すると vcl_recv に戻る事が可能な restart を行います、req.*の引継ぎが可能なので以下のように記述します。

```
sub vcl_fetch {
        if(beresp.status == 404 && req.restarts == 0){
                set req.url = "/noimage.jpg";
                return(restart);
        }
}
```

大きなファイルの時に503になる

ストレージのサイズがオブジェクトのサイズを超える場合 Varnish は503を返します。例えば640 MB の ISO イメージがたくさんあるサーバでストレージのサイズが 200 MB しかない場合 ISO イメージをリクエストしたら503が起きます。想定される転送サイズより大きめのストレージサイズを指定しましょう。

よくわからないけどスワップする

スワップする原因は幾つかありますが、Varnish3 で起こりなかなかハマるものについて紹介します。Varnish はごく短期間の TTL の場合別の一時的なストレージ(Transient)に保存します。これは初期状態だと制限がなく（=搭載メモリ）しかも malloc ストレージのため最悪の場合スワップしてしまいます。基本的に短期間しか保存しないので問題ないのですが、制限がないのも気持ち悪いので以下の方法で制限がかけられます。

```
-s Transient=malloc,100M
```

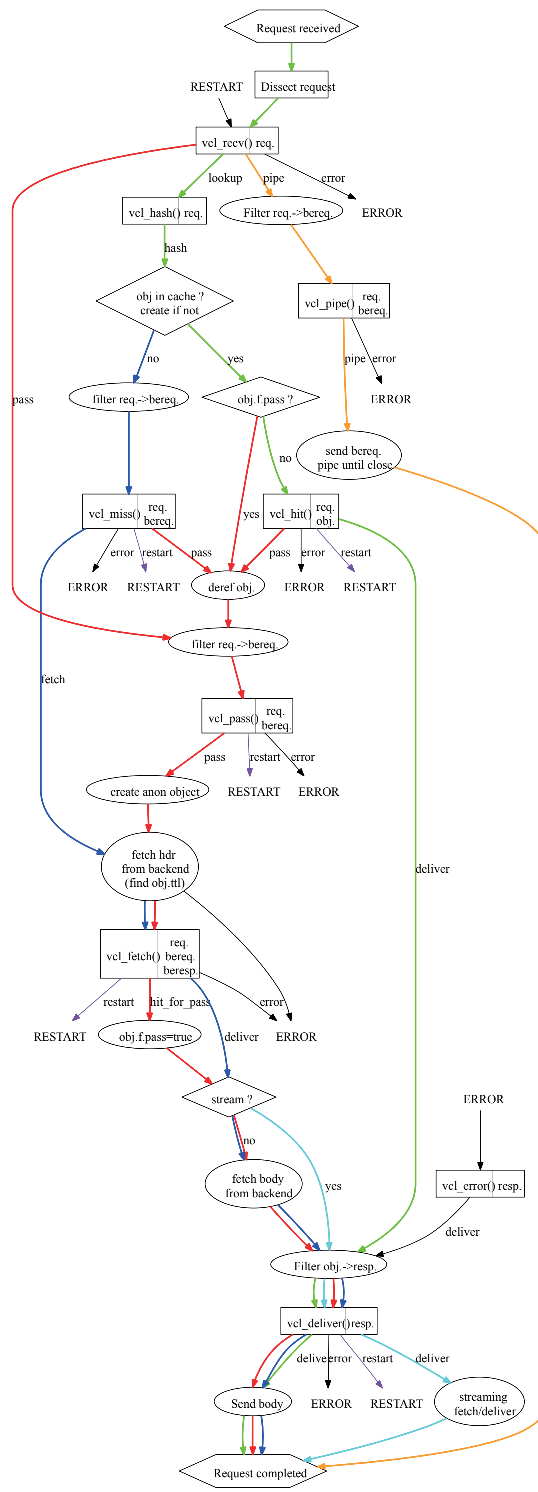
を起動オプションに追加します。ここでは100 MB と指定していますがそれぞれの環境に応じて調整してください。オブジェクトのサイズがストレージを超えると、やはり503になります。またストレージの指定は Transient だけではなく他のストレージも必ず指定して2個以上ストレージがあるようにしてください。場合により segfault する場合があります。

.ttl と.grace と.keep について

ttl はオブジェクトの有効期間です。期間中はキャッシュからレスポンスします。
grace が有効になるのは TTL が切れて再キャッシュしようとした場合でバックエンドが異常に陥りキャッシュできない場合に暫定で TTL が切れていてもレスポンスする猶予期間です。
また keep ですが V3.0.0 では特に有効に動きません。将来の予約のためだそうです。ちなみにキャッシュの実際の保持期間は ttl+grace+keep です。もちろん溢れた場合は消されます。

Appendix

アクションフロー図詳細



各アクションの際に指定できるリターン値

	アクション(vcl_*)										
return値	recv	pipe	pass	hash	miss	hit	fetch	deliver	error	init	fini
error	x	x	x		x	x	x				
pass	x		x		x	x					
hit_for_pass							x				
pipe	x	x									
lookup	x										
restart			x		x	x	x	x	x		
hash				x							
fetch					x						
deliver						x	x	x	x		
ok										x	x

vcl_init/fini の動作条件

vcl_init/fini は本文中で解説していませんが VCL がロード・アンロードの際に呼び出されます。Varnish の再起動などを行い実際に呼ばれるパタンを調べたのが以下です。

	vcl_init	vcl_fini
/etc/init.d/varnish start	x	
/etc/init.d/varnish restart	x	
/etc/init.d/varnish reload	x	
/etc/init.d/varnish stop		
start(cli)	x	
stop(cli)		
vcl.load(cli)	x	
vcl.discard(cli)		x
vcl.use(cli)		

vcl_fini が呼び出される条件は明示的に VCL が破棄されるタイミングのみです。

VCL で利用する各型を STRING に変換する関数

<table>
<tr><th>型名</th><th>関数名</th></tr>
<tr><td>IP</td><td>VRT_IP_string</td></tr>
<tr><td>INT</td><td>VRT_int_string</td></tr>
<tr><td>BYTES</td><td rowspan="2">VRT_double_string</td></tr>
<tr><td>DURATION</td></tr>
<tr><td>TIME</td><td>VRT_time_string</td></tr>
<tr><td>BOOL</td><td>VRT_bool_string</td></tr>
<tr><td>BACKEND</td><td>VRT_backend_string</td></tr>
</table>

第一引数に sp 指定、第二引数に変換したい値を指定する。

インライン C 利用時にヘッダを操作する関数

値の設定

void VRT_SetHdr(const struct sess *sp , enum gethdr_e where, const char *hdr, const char *p, …)		
項番	引数名	説明
1	*sp	spを指定
2	where	操作する対象を指定（HDR_REQ, HDR_RESP, HDR_OBJ, HDR_BEREQ, HDR_BERESP）
3	*hdr	操作するフィールドを指定
4～	*p	挿入する文字列を指定、複数指定の場合は結合する。最後にvrt_magic_string_endの指定必須
	戻り値	なし

値の取得

char * VRT_GetHdr(const struct sess *sp, enum gethdr_e where, const char *n)		
項番	引数名	説明
1	*sp	spを指定
2	where	操作する対象を指定（HDR_REQ, HDR_RESP, HDR_OBJ, HDR_BEREQ, HDR_BERESP）
3	*n	操作するフィールドを指定
	戻り値	指定したフィールドの値

*hdr 及び*n の指定方法は以下の通り

フィールド長のサイズ(1バイト) + フィールド名(末尾に:)

VRT_SetHdr(sp, HDR_RESP, "¥010Expires:", “XXXXXX”, vrt_magic_string_end);

VRT_GetHdr(sp, HDR_RESP, "¥010Expires:");

組み込み関数一覧

関数	説明	利用可能なアクション
ban(式)	指定条件でbanする	
ban_url(正規表現)	指定正規表現と一致するreq.urlの場合banする	
call サブルーチン名	別のサブルーチンを呼び出す	
hash_data(式)	オブジェクトを格納する際のハッシュに追加する	hashのみ
panic(式)	指定メッセージを出力して子プロセスを殺す	
purge	即時にハッシュを削除する	miss,hitのみ
return(アクション名)	returnする	
rollback	変数を初期化してvcl_recvに移動	
set 変数 代入演算子 式	変数に値を設定する	
synthetic 式	レスポンスボディを合成する	errorのみ
remove 変数	unsetと同じ（過去の互換性維持のため存在）	
unset 変数	変数を削除する	
error ステータスコード レスポンス	obj.status,obj.responseを指定してvcl_errorに行く	
regsub(置換対象,正規表現,置換式)	正規表現で最初にヒットしたものの置換を行う	
regsuball(置換対象,正規表現,置換式)	正規表現で最初にヒットしたもの全ての置換を行う	

VCL 変数の説明

変数	説明
client.*	クライアントの情報（IPなど）
server.*	サーバ情報（IPなど）
req.*	クライアントからのリクエスト（URLやヘッダ）
bereq.*	バックエンドへのリクエストする際に利用する（URLやヘッダ）
beresp.*	バックエンドからのレスポンスヘッダ
obj.*	キャッシュされたオブジェクトの情報（ヒット回数やヘッダ）
resp.*	クライアントにレスポンスするヘッダ
storage.*	ストレージ（使用サイズなど）

アクション遷移中に req.*は bereq.*にコピーされ、beresp.*はキャッシュ格納時に obj.*にレスポンス時には resp.*にコピーされます。

VCL 変数一覧

	インラインC利用時の関数	
変数名	読み込み時	書き込み時
client.ip	VRT_r_client_ip(sp)	
client.identity	VRT_r_client_identity(sp)	VRT_l_client_identity(sp,
server.ip	VRT_r_server_ip(sp)	
server.hostname	VRT_r_server_hostname(sp)	
server.identity	VRT_r_server_identity(sp)	
server.port	VRT_r_server_port(sp)	
req.request	VRT_r_req_request(sp)	VRT_l_req_request(sp,
req.url	VRT_r_req_url(sp)	VRT_l_req_url(sp,
req.proto	VRT_r_req_proto(sp)	VRT_l_req_proto(sp,
req.http.*	VRT_GetHdr(sp, HDR_REQ,	VRT_SetHdr(sp, HDR_REQ,
req.backend	VRT_r_req_backend(sp)	VRT_l_req_backend(sp,
req.restarts	VRT_r_req_restarts(sp)	
req.esi_level	VRT_r_req_esi_level(sp)	
req.ttl	VRT_r_req_ttl(sp)	VRT_l_req_ttl(sp,
req.grace	VRT_r_req_grace(sp)	VRT_l_req_grace(sp,
req.keep	VRT_r_req_keep(sp)	VRT_l_req_keep(sp,
req.xid	VRT_r_req_xid(sp)	
req.esi	VRT_r_req_esi(sp)	VRT_l_req_esi(sp,
req.can_gzip	VRT_r_req_can_gzip(sp)	
req.backend.healthy	VRT_r_req_backend_healthy(sp)	
req.hash_ignore_busy	VRT_r_req_hash_ignore_busy(sp)	VRT_l_req_hash_ignore_busy(sp,
req.hash_always_miss	VRT_r_req_hash_always_miss(sp)	VRT_l_req_hash_always_miss(sp,
bereq.request	VRT_r_bereq_request(sp)	VRT_l_bereq_request(sp,
bereq.url	VRT_r_bereq_url(sp)	VRT_l_bereq_url(sp,
bereq.proto	VRT_r_bereq_proto(sp)	VRT_l_bereq_proto(sp,
bereq.http.*	VRT_GetHdr(sp, HDR_BEREQ,	VRT_SetHdr(sp, HDR_BEREQ,
bereq.connect_timeout	VRT_r_bereq_connect_timeout(sp)	VRT_l_bereq_connect_timeout(sp,
bereq.first_byte_timeout	VRT_r_bereq_first_byte_timeout(sp)	VRT_l_bereq_first_byte_timeout(sp,
bereq.between_bytes_timeout	VRT_r_bereq_between_bytes_timeout(sp)	VRT_l_bereq_between_bytes_timeout(sp,
beresp.proto	VRT_r_beresp_proto(sp)	VRT_l_beresp_proto(sp,
beresp.saintmode		VRT_l_beresp_saintmode(sp,
beresp.status	VRT_r_beresp_status(sp)	VRT_l_beresp_status(sp,
beresp.response	VRT_r_beresp_response(sp)	VRT_l_beresp_response(sp,
beresp.http.*	VRT_GetHdr(sp, HDR_BERESP,	VRT_SetHdr(sp, HDR_BERESP,
beresp.do_esi	VRT_r_beresp_do_esi(sp)	VRT_l_beresp_do_esi(sp,
beresp.do_stream	VRT_r_beresp_do_stream(sp)	VRT_l_beresp_do_stream(sp,
beresp.do_gzip	VRT_r_beresp_do_gzip(sp)	VRT_l_beresp_do_gzip(sp,
beresp.do_gunzip	VRT_r_beresp_do_gunzip(sp)	VRT_l_beresp_do_gunzip(sp,
beresp.ttl	VRT_r_beresp_ttl(sp)	VRT_l_beresp_ttl(sp,
beresp.grace	VRT_r_beresp_grace(sp)	VRT_l_beresp_grace(sp,
beresp.keep	VRT_r_beresp_keep(sp)	VRT_l_beresp_keep(sp,
beresp.backend.name	VRT_r_beresp_backend_name(sp)	
beresp.backend.ip	VRT_r_beresp_backend_ip(sp)	
beresp.backend.port	VRT_r_beresp_backend_port(sp)	
beresp.storage	VRT_r_beresp_storage(sp)	VRT_l_beresp_storage(sp,
obj.proto	VRT_r_obj_proto(sp)	VRT_l_obj_proto(sp,
obj.status	VRT_r_obj_status(sp)	VRT_l_obj_status(sp,
obj.response	VRT_r_obj_response(sp)	VRT_l_obj_response(sp,
obj.hits	VRT_r_obj_hits(sp)	
obj.http.*	VRT_GetHdr(sp, HDR_OBJ,	VRT_SetHdr(sp, HDR_OBJ,
obj.ttl	VRT_r_obj_ttl(sp)	VRT_l_obj_ttl(sp,
obj.grace	VRT_r_obj_grace(sp)	VRT_l_obj_grace(sp,
obj.keep	VRT_r_obj_keep(sp)	VRT_l_obj_keep(sp,
obj.lastuse	VRT_r_obj_lastuse(sp)	
resp.proto	VRT_r_resp_proto(sp)	VRT_l_resp_proto(sp,
resp.status	VRT_r_resp_status(sp)	VRT_l_resp_status(sp,
resp.response	VRT_r_resp_response(sp)	VRT_l_resp_response(sp,
resp.http.*	VRT_GetHdr(sp, HDR_RESP,	VRT_SetHdr(sp, HDR_RESP,
now	VRT_r_now(sp)	
storage.[storagename].free_space	VRT_Stv_free_space([storagename])	
storage.[storagename].used_space	VRT_Stv_used_space([storagename])	
storage.[storagename].happy	VRT_Stv_happy([storagename])	

	アクション											
型名	recv	pipe	pass	hash	miss	hit	fetch	deliver	error	init	fini	説明
IP	R	R	R	R	R	R	R	R	R	R	R	クライアントのIPアドレス
STRING	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	クライアントの識別子client directorで使う
IP	R	R	R	R	R	R	R	R	R	R	R	サーバのIPアドレス
STRING	R	R	R	R	R	R	R	R	R	R	R	サーバのホスト名
STRING	R	R	R	R	R	R	R	R	R	R	R	サーバの識別子（起動オプション-iで指定）
INT	R	R	R	R	R	R	R	R	R	R	R	サーバのLISTENポート
STRING	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	リクエストメソッド
STRING	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	リクエストURL
STRING	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	HTTPプロトコルバージョン
HEADER	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	HTTPヘッダ
BACKEND	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	このリクエストを処理するバックエンドを指定
INT	R	R	R	R	R	R	R	R	R	R	R	このリクエストを処理するときに何回restartしたか
INT	R	R	R	R	R	R	R	R	R	R	R	ESI処理時の階層
DURATION	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	有効期間を上書きする
DURATION	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	猶予時間を上書きする
DURATION	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	保持時間を上書きする
STRING	R	R	R	R	R	R	R	R	R	R	R	リクエストのユニークなID
BOOL	R W						R W	R W	R W			ESI機能の有効・無効化
BOOL	R	R	R	R	R	R	R	R	R	R	R	クライアントがgzip転送を許可しているか
BOOL	R	R	R	R	R	R	R	R	R	R	R	現在選択されているバックエンドが正常かどうか
BOOL	R W											キャッシュ検索しているビジーなオブジェクトを無視するか
BOOL	R W											強制的にこのリクエストをキャッシュミスにするかどうか
STRING		R W	R W		R W		R W					リクエストメソッド
STRING		R W	R W		R W		R W					リクエストURL
STRING		R W	R W		R W		R W					HTTPプロトコルバージョン
HEADER		R W	R W		R W		R W					HTTPヘッダ
DURATION		R W	R W		R W							バックエンドに接続する際のタイムアウト時間
DURATION		R W			R W							バックエンドから最初のバイトが転送される際のタイムアウト時間
DURATION		R W			R W							通信中に一時的に受信が待ったときのタイムアウト時間
STRING							R W					HTTPプロトコルバージョン
DURATION							W					セイントモードを有効にするか
INT							R W					HTTPステータスコード
STRING							R W					HTTPステータスコード文字列
HEADER							R W					HTTPヘッダ
BOOL							R W					ESI処理を行うか
BOOL							R W					ストリームを有効にするか
BOOL							R W					gzip圧縮するか
BOOL							R W					gzip解凍するか
DURATION							R W					オブジェクトの有効期間
DURATION							R W					オブジェクトの追加の猶予時間
DURATION							R W					オブジェクトの追加の保持時間
STRING							R W					使用したバックエンドの名前
IP							R					使用したバックエンドのIPアドレス
INT							R					使用したバックエンドのポート
STRING							R W					キャッシュを格納するストレージの名前を指定
STRING						R W			R W			HTTPプロトコルバージョン
INT									R W			HTTPステータスコード
STRING									R W			HTTPステータスコード文字列
INT						R		R				オブジェクトをレスポンスした回数、0はキャッシュミスを表す
HEADER						R			R W			HTTPヘッダ
DURATION						R W			R W			オブジェクトの有効期間
DURATION						R W			R W			オブジェクトの追加の猶予時間
DURATION						R W			R W			オブジェクトの追加の保持時間
DURATION						R		R	R			オブジェクトが最後に使われてからの秒数
STRING								R W				HTTPプロトコルバージョン
INT								R W				HTTPステータスコード
STRING								R W				HTTPステータスコード文字列
HEADER								R W				HTTPヘッダ
TIME	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	現在時間(epoch)
BYTES	R	R	R	R	R	R	R	R	R	R	R	指定ストレージの空き容量
BYTES	R	R	R	R	R	R	R	R	R	R	R	指定ストレージの使用容量
BOOL	R	R	R	R	R	R	R	R	R	R	R	おそらく正常かどうかだと思うが不明

灰色部の init/fini において、各変数は定義上利用可能でしたが、

確認した限り、指定すると Varnish がクラッシュしました。

必ず確認してから使って下さい。

起動パラメーター覧

パラメータ名	型	最小値	最大値	デフォルト値	属性
acceptor_sleep_decay	DOUBLE	0	1	0.9	実験的
acceptor_sleep_incr	時間(小数あり)	0	1	0.001	実験的
acceptor_sleep_max	時間(小数あり)	0	10	0.05	実験的
auto_restart	BOOL値	–	–	on	
ban_dups	BOOL値	–	–	on	
ban_lurker_sleep	時間(小数あり)	0	UINT_MAX	0.01	
between_bytes_timeout	時間(小数あり)	0	UINT_MAX	60	
cc_command	文字列	–	–	VCC_CC	
cli_buffer	数値	4096	UINT_MAX	8192	
cli_timeout	時間(整数)	0	0	10	
clock_skew	数値	0	UINT_MAX	10	
connect_timeout	時間(小数あり)	0	UINT_MAX	0.7	
critbit_cooloff	時間(小数あり)	60	254	180	開発向け
default_grace	時間(小数あり)	0	UINT_MAX	10	
default_keep	時間(小数あり)	0	UINT_MAX	0	
default_ttl	時間(小数あり)	0	UINT_MAX	120	
diag_bitmap	BITMAP	–	–	0	
esi_syntax	BITMAP	–	–	0	
expiry_sleep	時間(小数あり)	0	60	1	
fetch_chunksize	数値	4	UINT_MAX / 1024	128	実験的
fetch_maxchunksize	数値	64	UINT_MAX / 1024	262144	実験的
first_byte_timeout	時間(小数あり)	0	UINT_MAX	60	
group	グループ名	–	–	MAGIC_INIT_STRING	
gzip_level	数値	0	9	6	
gzip_stack_buffer	数値	2048	UINT_MAX	32768	実験的
gzip_tmp_space	数値	0	2	0	実験的
http_gzip_support	BOOL値	–	–	on	実験的
http_max_hdr	数値	32	UINT_MAX	64	
http_range_support	BOOL値	–	–	on	実験的
http_req_hdr_len	数値	40	UINT_MAX	2048	
http_req_size	数値	256	UINT_MAX	32768	
http_resp_hdr_len	数値	40	UINT_MAX	2048	
http_resp_size	数値	256	UINT_MAX	32768	
listen_address	アドレス	–	–	:80	
listen_depth	数値	0	UINT_MAX	1024	
log_hashstring	BOOL値	–	–	on	
log_local_address	BOOL値	–	–	off	
lru_interval	時間(整数)	0	0	2	実験的
max_esi_depth	数値	0	UINT_MAX	5	
max_restarts	数値	0	UINT_MAX	4	
ping_interval	数値	0	UINT_MAX	3	
pipe_timeout	時間(整数)	0	0	60	
prefer_ipv6	BOOL値	–	–	off	
saintmode_threshold	数値	0	UINT_MAX	10	実験的
send_timeout	時間(整数)	0	0	60	
sess_timeout	時間(整数)	0	0	5	
sess_workspace	数値	1024	UINT_MAX	65536	
session_linger	数値	0	UINT_MAX	50	実験的
session_max	数値	1000	UINT_MAX	100000	
shm_reclen	数値	16	65535	255	
shm_workspace	数値	4096	UINT_MAX	8192	
shortlived	時間(小数あり)	0	UINT_MAX	10	
syslog_cli_traffic	BOOL値	–	–	on	
user	ユーザ名	–	–	MAGIC_INIT_STRING	
vcc_err_unref	BOOL値	–	–	on	
vcl_dir	文字列	–	–	VARNISH_VCL_DIR	
vcl_trace	BOOL値	–	–	off	
vmod_dir	文字列	–	–	VARNISH_VMOD_DIR	
waiter	WAITER	–	–	default	実験的

適用タイミング	単位	説明
	-	アクセプターが成功するたびに受け入れを行うための待機秒数を乗数的に減らす時の乗数
	秒	アクセプターがリソースを使い果たした際に再試行する際の待機秒数
	秒	アクセプターがリソースを使い果たした際の最大待機秒数
	-	子プロセス死亡時に自動再起動を行うか
	-	重複しているbanを排除
	秒	banリストが追加された際にban lurkerスレッドが待機する秒数
	秒	通信中に一時的に受信が待ったときのタイムアウト時間
要リロード	-	VCLをコンパイルする際に利用するコマンドライン
	バイト	CLIで入力時のバッファサイズ
	秒	CLIで親プロセスから子プロセスにリクエストする際のタイムアウト秒
	秒	バックエンドとVarnishでのクロックのずれの許容秒
	秒	バックエンドに接続する際のデフォルトの接続タイムアウト秒
	秒	critbitが削除されたオブジェクトのヘッダをガベージするまでにcooloffリストで保持する時間
遅延適用	秒	デフォルトのgrace時間を適用
遅延適用	秒	デフォルトのkeep時間を適用
	秒	デフォルトのTTLを指定 既にキャッシュされているオブジェクトには影響しない
	-	診断を行う際のオプション
	-	ESIのパースを行う際のチェックオプション
	秒	スレッドをスリープするまでの待機秒数
	キロバイト	フェッチで利用するチャンクサイズ
	キロバイト	ストレージから割り当てる最大チャンクサイズ
	秒	バックエンドから最初のバイトを受け取るまでのタイムアウト秒数
要再起動	-	子プロセスのグループ名を設定
	-	gzipの圧縮率(0=debug,1=高速～9=高圧縮)
	バイト	gzipに利用するスタックバッファサイズ
	-	gzip/gunzipで利用する一時領域指定
	-	gzipをサポート
	ヘッダ行数	クライアントのリクエストとバックエンドのレスポンスの最大ヘッダ行数
	-	Rangeヘッダを有効にする
	バイト	クライアントのHTTPリクエストヘッダの最大バイト数
	バイト	クライアントのHTTPリクエストの最大バイト数
	バイト	バックエンドのHTTPレスポンスヘッダの最大バイト数
	バイト	バックエンドのHTTPレスポンスの最大バイト数
要再起動	-	Listenアドレス
要再起動	コネクション数	Listenキュー数
	-	ログにハッシュコンポーネントを出力する
	-	ローカルアドレスのSessionOpenをログに記録する
	秒	LRUリストを移動する際間隔
	ネスト数	esi:includeの最大ネスト数
	restarts	1リクエストあたりの最大リスタート数
要再起動	秒	親プロセスから子プロセスへのping間隔(0は無効化)
	秒	Pipeのアイドル状態のセッションタイムアウト時間
	-	IPv4/v6両方を持っているバックエンドに接続する際にIPv6を利用
	objects	Saintmodeに入る際の閾値(0は無効化)
遅延適用	秒	クライアント接続の送信タイムアウト時間
	秒	永続的セッションのタイムアウト値
遅延適用	バイト	HTTPプロトコルのワークスペースサイズ
	ミリ秒	ワーカースレッドが次のリクエストを取得しに行くまでの待機時間
	セッション数	セッション最大数
	バイト	共有メモリのログレコードの最大バイト数
遅延適用	バイト	ワーカースレッドに割り当てる共有メモリワークスペースサイズ
	秒	指定TTL以下のオブジェクトを一時的なストレージに格納します
	-	syslogのCLIの実行ログを記録(LOG_INFO)
要再起動	-	子プロセスのユーザ名を指定
	-	VCLコンパイル時の未使用定義エラーを出力
	-	VCLディレクトリを指定
	-	VCLのトレースログを出力
	-	VMODディレクトリを指定
要再起動	-	イベント待ちのカーネルインタフェースを設定

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。
もともと、勉強会やブログなどで Varnish の情報を発信していたのですが、自分の頭の整理を兼ねて Varnish の本を出してみたいなぁと思い書いてみました。
のんびり書いていたら Varnish3 のリリースがあったり、仕事が忙しくなったり、コミケに落ちて油断していたら委託していただけるということで慌てたり、ソースコードを読みながら検証していたら時間が足りなくなったり様々なことがありました。
他にもいろいろ書きたかったこと、サラッと流したけどもっと詳しく書きたかったことといろいろありますが時間の許す限り詰め込みました。あなたが Varnish を今使っている、使っていないにかかわらず参考になれば幸いです。

それでは機会があればまた！

奥付

Varnish Cache 入門

ーー発行日ーー
2011－08－13（Rev.1）, 2011－09－12（Rev.2）
2011－09－13（Rev.2.1）, 2012－06－01（Rev.3）

ーー印刷ーー
サンライズ様

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
校正を手伝って下りました
@ishikawa84g
@saeoshi
@tmae
@W53SA

そして
Varnish Software

Varnish Cache 入門

なぜVarnish Cacheなのか
どのようなときに使用するか
　静的コンテンツの配信
　動的コンテンツの配信
Varnishを使うための環境
Varnishのインストール
まずは動かしてみよう
Varnishの設定について
VCLについて
　基本的な言語仕様
　Varnishの基本動作ステップ
　req.*の内容がbereq.*に転記されるタイミング
　簡易版アクションフロー図
起動時オプションについて
Varnishのデバッグ
応用的な使い方
　管理コンソールの利用方法
　インラインC
　統計情報の見方
　ストレージサイズとTTLの決め方
　ヒット率の上げ方
　Edge Side Includes (ESI)
　gzipの利用
　VMODの追加の仕方
　ストレージの高度な制御
　バックエンドの高度な利用
　varnishtestの使い方
　Varnishにやさしいbanの仕方
Tip's
Appendix
あとがき