# 简介：

 Varnish是一款高性能且开源的反向代理服务器和HTTP 加速器，它的作者Poul-Henning Kamp是FreeBSD核心的开发人员之一，Varnish采用全新的软件体系机构，和现在的硬件体系配合紧密。作者认为现在的电脑比起1975年已经 复杂许多。在1975年时，储存媒介只有两种：内存与硬盘。但现在计算机系统的内存除了主存外，还包括CPU内的L1、L2，甚至有L3快取。硬盘上也有 自己的快取装置，因此Squid Cache自行处理物件替换的架构不可能得知这些情况而做到最佳化，但操作系统可以得知这些情况，所以这部份的工作应该交给操作系统处理，这就是 Varnish cache设计架构。

挪威最大的在线报纸 Verdens Gang(vg.no)使用3台Varnish代替了原来的12台squid，性能比以前更好，这是Varnish最成功的应用案例

# **Varnish的结构与特点**

Varnish是一个轻量级的Cache和反向代理软件，先进的设计理念和成熟的设计框架是Varnish的主要特点，现在的Varnish总共代码量不大，功能上虽然在不断改进，但是还需要继续丰富和加强。下面总结了Varnish的一些特点：

（1）是基于内存缓存，重启后数据将消失。  
（2）利用虚拟内存方式，io性能好。  
（3）支持设置0~60秒内的精确缓存时间。  
（4）VCL配置管理比较灵活。  
（5）32位机器上缓存文件大小为最大2G。  
（6）具有强大的管理功能，例如top，stat，admin，list等。  
（7）状态机设计巧妙，结构清晰。  
（8）利用二叉堆管理缓存文件，达到积极删除目的。

# **Varnish与Squid的对比**

说到Varnish，不能不提Squid，Squid是一个高性能的代理缓存服务器，它和varnish之间有诸多的异同点，这里分析如下：  
下面是他们之间的**相同点：**（1）都是一个反向代理服务器，  
（2）都是开源软件，

下面是它们的**不同点**，也是Varnish的优点：  
（1）Varnish的稳定性很高，两者在完成相同负荷的工作时，Squid服务器发生故障的几率要高于Varnish，因为使用Squid要经常重启。  
（2）Varnish访问速度更快，Varnish采用了“Visual Page Cache”技术，所有缓存数据都直接从内存读取，而squid是从硬盘读取，因而Varnish在访问速度方面会更快。  
（3）Varnish可以支持更多的并发连接，因为Varnish的TCP连接释放要比Squid快。因而在高并发连接情况下可以支持更多TCP连接。  
（4）Varnish可以通过管理端口，使用正则表达式批量的清除部分缓存，而Squid是做不到的。

当然，与传统的Squid相比，Varnish也是有**缺点**的，列举如下：  
（1）varnish在高并发状态下CPU、IO、内存等资源开销都高于Squid。  
（2）varnish进程一旦Hang、Crash或者重启，缓存数据都会从内存中完全释放，此时所有请求都会发送到后端服务器，在高并发情况下，会给后端服务器造成很大压力。

# 安装varnish

## 安装环境:

Varnish server 172.16.103.231

**lsb\_release –a**

Distributor ID: RedHatEnterpriseServer

Description: Red Hat Enterprise Linux Server release 6.3 (Santiago)

Release: 6.3

Codename: Santiago

Web Server 172.16.103.131

Distributor ID: RedHatEnterpriseServer

Description: Red Hat Enterprise Linux Server release 5.4 (Tikanga)

Release: 5.4

Codename: Tikanga

建立varnish用户以及用户组,并且创建varnish缓存目录和日志目录

useradd -s /sbin/nologin varnish

mkdir -p /var/varnish/cache

mkdir -p /var/varnish/log

chown -R varnish:varnish /var/varnish/cache

chown -R varnish:varnish /var/varnish/log

## 安装pcre

wget <http://jaist.dl.sourceforge.net/project/pcre/pcre/8.33/pcre-8.33.tar.gz>

tar zxvf pcre-8.33.tar.gz

./configure --prefix=/usr/local/pcre/

Make

make install

## 安装varnish

wget <http://repo.varnish-cache.org/source/varnish-3.0.5.tar.gz>

tar zxvf varnish-3.0.5.tar.gz

cd varnish-3.0.5

export PKG\_CONFIG\_PATH=/usr/local/pcre/lib/pkgconfig

./configure --prefix=/usr/local/varnish

Make

make install

其中，“PKG\_CONFIG\_PATH”是指定varnish查找pcre库的路径，如果pcre安装在了其它路径下，在这里指定相应的路径即可，Varnish默认查找的pcre库路径为/usr/local/lib/ pkgconfig。

# 配置Varnish

## Varnish内置函数

### vcl\_recv函数

用于接收和处理请求，当请求到达并成功接收后被调用，通过判断请求的数据来决定如何处理请求。  
此函数一般以如下几个关键字结束：

pass：表示进入pass模式，把请求控制权交给vcl\_pass函数。

pipe：表示进入pipe模式，把请求控制权交给vcl\_pipe函数。

error code [reason]：表示返回“code”给客户端，并放弃处理该请求，“code”是错误标识，例如200、405等，“reason”是错误提示信息。

### vcl\_pipe函数

此函数在进入pipe模式时被调用，用于将请求直接传递至后端主机，在请求和返回的内容没有改变的情况下，将不变的内容返回给客户端，直到这个链接关闭。  
此函数一般以如下几个关键字结束：

error code [reason]

 pipe

### vcl\_pass函数

此函数在进入pass模式时被调用，用于将请求直接传递至后端主机，后端主机应答数据后送给客户端，但不进行任何缓存，在当前连接下每次都返回最新的内容。  
此函数一般以如下几个关键字结束：

error code [reason]

pass

### lookup

表示在缓存里查找被请求的对象，并且根据查找的结果把控制权交给函数vcl\_hit或者函数vcl\_miss。

### vcl\_hit函数

在执行lookup指令后，如果在缓存中找到请求的内容，将自动调用该函数。  
此函数一般以如下几个关键字结束：

deliver：表示将找到的内容发送给客户端，并把控制权交给函数vcl\_deliver。

error code [reason]

pass

### vcl\_miss函数

在执行lookup指令后，如果没有在缓存中找到请求的内容时自动调用该方法，此函数可以用于判断是否需要从后端服务器取内容。  
此函数一般以如下几个关键字结束：

fetch：表示从后端获取请求的内容，并把控制权交给vcl\_fetch函数。

error code [reason]

pass

### vcl\_fetch函数

在从后端主机更新缓存并且获取内容后调用该方法，接着，通过判断获取的内容来决定是否将内容放入缓存，还是直接返回给客户端。  
此函数一般以如下几个关键字结束：

error code [reason]

pass

deliver

### vcl\_deliver函数

在缓存中找到请求的内容后，发送给客户端前调用此方法。此函数一般以如下几个关键字结束：

error code [reason]

deliver

### vcl\_timeout 函数

此函数在缓存内容到期前调用。一般以如下几个关键字结束：

discard：表示从缓存中清除该内容。

Fetch

### vcl\_discard函数

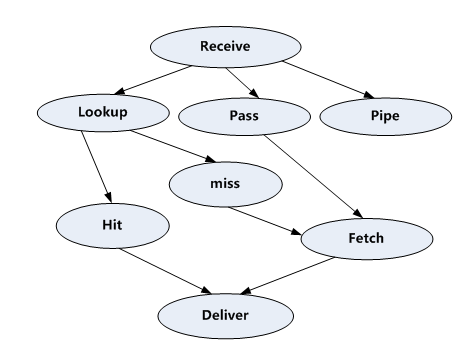
在缓存内容到期后或缓存空间不够时，自动调用该方法，一般以如下几个关键字结束：

 keep：表示将内容继续保留在缓存中。

Discard

## VCL处理流程图

通过上面对VCL函数的介绍，读者对各个函数实现的功能已经有了一个了解，其实每个函数之间都是相互关联的，下图列出了varnish处理HTTP请求的一个运行流程图。



处理过程大致分为如下几个步骤：

1. Receive状态，也就是请求处理的入口状态，根据VCL规则判断该请求应该是Pass或Pipe，或者进入Lookup（本地查询）。
2. Lookup状态，进入此状态后，会在hash表中查找数据，若找到，则进入Hit状态，否则进入miss状态。
3. Pass状态，在此状态下，会进入后端请求，即进入fetch状态。
4. Fetch状态，在Fetch状态下，对请求进行后端的获取，发送请求，获得数据，并进行本地的存储。
5. Deliver状态， 将获取到的数据发送给客户端，然后完成本次请求。

## 内置公用变量

VCL内置的公用变量可以用在不同的VCL函数中，根据这些公用变量使用的不同阶段，下面依次介绍。  
当请求到达后，可以使用的公用变量:

|  |  |
| --- | --- |
| 公用变量名称 | 含义 |
| req.backend | 指定对应的后端主机 |
| server.ip | 表示服务器端IP |
| client.ip | 表示客户端IP |
| req.request | 指定请求的类型，例如GET、HEAD、POST等 |
| req.url | 指定请求的地址 |
| req.proto | 表示客户端发起请求的HTTP协议版本 |
| req.http.header | 表示对应请求中的http头部信息 |
| req. restarts | 表示请求重启的次数，默认最大值为4 |

在向后端主机请求时，可以使用的公用变量

|  |  |
| --- | --- |
| 公用变量名称 | 含义 |
| beresp.request | 指定请求的类型，例如GET、HEAD等 |
| beresp.url | 指定请求的地址 |
| beresp .proto | 表示客户端发起请求的HTTP协议版本 |
| beresp .http.header | 表示对应请求中的http头部信息 |
| beresp .ttl | 表示缓存的生存周期，也就是cache保留多长时间，单位是秒 |

从cache或者后端主机获取内容后，可以使用的公用变量

|  |  |
| --- | --- |
| 公用变量名称 | 含义 |
| obj.status | 表示返回内容的请求状态代码，例如200、302、504等 |
| obj.cacheable | 表示返回的内容是否可以缓存，也就是说，如果HTTP返回是200、203、300、301、302、404、410等，并且有非0的生存期，则可以缓存 |
| obj.valid | 表示是否是有效的HTTP应答 |
| obj.response | 表示返回内容的请求状态信息 |
| obj.proto | 表示返回内容的HTTP协议版本 |
| obj.ttl | 表示返回内容的生存周期，也就是缓存时间，单位是秒 |
| obj.lastuse | 表示返回上一次请求到现在的间隔时间，单位是秒 |

对客户端应答时，可以使用的公用变量

|  |  |
| --- | --- |
| 公用变量名称 | 含义 |
| resp.status | 表示返回给客户端的HTTP状态代码 |
| resp.proto | 表示返回给客户端的HTTP协议版本 |
| resp.http.header | 表示返回给客户端的HTTP头部信息 |
| resp.response | 表示返回给客户端的HTTP状态信息 |

## Varnish配置实例参考版本

backend webserver {  
     .host = "192.168.12.26";  
     .port = "80";  
 }

#调用vcl\_recv开始。  
sub vcl\_recv {  
        if (req.http.x-forwarded-for) {  
                set req.http.X-Forwarded-For =  
                    req.http.X-Forwarded-For ", " client.ip;  
        } else {  
                set req.http.X-Forwarded-For = client.ip;  
        }  
#如果请求的类型不是GET、HEAD、PUT、POST、TRACE、OPTIONS、DELETE时，进入pipe模式。注意这里是“&&”的关系。  
        if (req.request != "GET" &&  
           req.request != "HEAD" &&  
           req.request != "PUT" &&  
           req.request != "POST" &&  
           req.request != "TRACE" &&  
           req.request != "OPTIONS" &&  
           req.request != "DELETE") {  
           return (pipe);  
        }  
#如果请求的类型不是GET与HEAD，则进入pass模式。  
   if (req.request != "GET" && req.request != "HEAD") {  
           return (pass);  
        }

#对ixdba.net或者ixdba.cn两个域名进行缓存加速，这是个泛域名的概念，也就是所有以ixdba.net或者ixdba.cn结尾的域名都进行缓存。  
        if (req.http.host ~ "^(.\*).ixdba.net" || req.http.host ~ "^(.\*).ixdba.cn") {  
           set req.backend = webserver;  
        }

#对以.jsp和.do结尾以及带有?的URL时，直接从后端服务器读取内容。  
        if (req.url ~ "\.(jsp|do)($|\?)") {  
           return (pass);  
        } else {  
        return (lookup);  
        }  
}  
   
sub vcl\_pipe {  
    return (pipe);  
}  
   
sub vcl\_pass {  
    return (pass);  
}  
   
sub vcl\_hash {  
    set req.hash += req.url;  
    if (req.http.host) {  
        set req.hash += req.http.host;  
    } else {  
        set req.hash += server.ip;  
    }  
    return (hash);  
}  
   
sub vcl\_hit {  
    if (!obj.cacheable) {  
        return (pass);  
    }  
    return (deliver);  
}  
   
sub vcl\_miss {  
    return (fetch);  
}  
   
sub vcl\_fetch {  
     if (!beresp.cacheable) {  
         return (pass);  
     }  
     if (beresp.http.Set-Cookie) {  
         return (pass);  
     }

#当url中包含servlet时，不进行缓存。  
    if (req.url ~ "^/servlet/") {  
        return (pass);  
    }

#当url中包含services时，不进行缓存。  
    if (req.url ~ "^/services/") {  
        return (pass);  
    }

#对于请求类型是GET，并且请求的URL中包含upload，那么就进行缓存，缓存的时间是300秒，即5分钟。  
    if (req.request == "GET" && req.url ~ "^/upload(.\*)$") {  
       set beresp.ttl = 300s;  
    }  
#对于请求类型是GET，并且请求的URL以png、xsl、xml、gif、css、js等结尾时，则进行缓存，缓存时间为600秒。  
    if (req.request == "GET" && req.url ~ "\.(png|xsl|xml|pdf|ppt|doc|docx|chm|rar|zip|bmp|jpeg|swf|ico|mp3|mp4|rmvb|ogg|mov|avi|wmv|swf|txt|png|gif|jpg|css|js|html|htm)$") {  
       set beresp.ttl = 600s;  
    }  
    return (deliver);  
}  
  #下面是添加一个Header标识，以判断缓存是否命中。  
sub vcl\_deliver {  
    if (obj.hits > 0) {  
       set resp.http.X-Cache = "HIT from [www.ixdba.net](http://www.ixdba.net)";  
    } else {  
      set resp.http.X-Cache = "MISS from [www.ixdba.net](http://www.ixdba.net)";  
    }  
    return (deliver);  
}

## **Backend servers**

Varnish有后端（或称为源）服务器的概念。后端服务器是指Varnish提供加速服务的那台，通常提供内容。  
第一件要做的事情是告诉Varnish，哪里能找到要加速的内容。打开varnish配置文件，源码包安装的在/usr/local/etc/varnish/default.vcl中，rpm安装的在/etc/varnish/default.vcl中。

backend webserver {

.host = "172.16.103.131";

.port = "80";

}

这段配置，定义了一个Varnish中的一个后端（Backend），叫做webserver。当Varnish需要从这个后端获得内容时，它就会连接到172.16.103.131的80端口上。  
Varnish可以有多个后端，你甚至可以为了负载均衡将几个后端加入到一个集群后端中。  
现在我们完成了Varnish的基本配置，接下来让我们在80端口上把Varnish起起来，对它做些实验性测试。

## Starting varnish

在开始下面的内容之前，你先要确保现在你的varnish没有在运行，如果有，就用pkill varnishd去关闭它。然后到根目录，输入以下代码：

cd /usr/local/varnish/sbin/

./varnishd -f /usr/local/varnish/etc/varnish/default.vcl -s malloc,100M -T 127.0.0.1:2000 -a 0.0.0.0:8080

-f /urs/local/etc/varnish/default.vcl  
-f选项指定了将使用哪个配置文件

-s malloc,1G  
-s选项用于指定varnish使用何种存储类型保存内容。这里我使用malloc，这个代表我只使用内存存储。如果还有其他后端，用:ref:tutorial-storage来表示。1G指定了分配多少内存——这里是一个G。

-T 127.0.0.1:2000  
Varnish含有内置文本管理界面。可以通过它对varnish进行管理，最主要你还不用停掉varnish。你可以给管理界面分配端口。确保你的管理 界面没有暴露给外界，因为通过Varnish管理界面你可以很容易地访问系统根目录。我建议直接绑定在localhost上，就别远程了。如果你的系统上 有不可信任的user，就用防火墙规则只要限制界面访问根目录就行。  
-a 0.0.0.0:8080  
对于进入的HTTP请求，我指定varnish监听8080。对于生产环境，你可能需要让varnish监听在80端口，这个是默认的。（关键看前面有没有负载均衡）  
   
现在我已经启动了Varnish。用浏览器访问下http://varnishServerIP:8080/。你应该会看到你的web应用的运行的。  
Varnish运行后，应用的访问速度更快主要取决于一些因素。如果你的应用为每个session使用cookie的话（很多PHP和Java应用无论是 否需要都会发送一个session cookie），或者应用使用验证的话，这些varnish都不会缓存。现在先放一放，别考虑这些，等到Achieving a high hitrate的时候，我们再来好好谈。  
通过查看日志，我们可以用来确定varnish是不是真的起作用了。

## **Logging in Varnish**

在Varnish中，日志的工作方式，是一个很好特性。Varnish将日志记录到共享内存片段，而不是记录到一个普通文件中。当记录到内存片段的最后处，会再从头开始记，覆写老数据。这个比记录到文件要快的多，而且不需要磁盘空间。

另一方面，如果你没有执行程序去将这些日志写到磁盘中的话，他们是会消失的。  
Varnishlog是一个用来查看Varnish日志的程序。Varnishlog提供给我们原始日志。这里还有其他客户端，之后我们会介绍。  
在启动varnish的终端窗口，我们输入varnishlog，然后按回车。  
你会看到如下内容，使用”.”可以缓缓滚动：

**cd /usr/local/varnish/bin/**

**./varnishlog**

0 CLI - Rd ping

0 CLI - Wr 200 19 PONG 1392605216 1.0

.. 0 CLI - Rd ping

0 CLI - Wr 200 19 PONG 1392605219 1.0

. 0 CLI - Rd ping

0 CLI - Wr 200 19 PONG 1392605222 1.0

0 CLI - Rd ping

这是varnish主进程，检查缓存进程，看是否一切正常。  
现在在浏览器，重新加载页面，显示你的web应用。你会看到如下内容：

CLI - Wr 200 19 PONG 1392605354 1.0

11 SessionOpen c 172.16.100.198 51590 0.0.0.0:8080

11 SessionClose c EOF

11 StatSess c 172.16.100.198 51590 0 1 0 0 0 0 0 0

12 BackendClose - webserver

12 BackendOpen b webserver 172.16.103.145 50808 172.16.103.131 80

12 BackendXID b 1766346205

12 TxRequest b GET

12 TxURL b /

12 TxProtocol b HTTP/1.1

12 TxHeader b Host: 172.16.103.145:8080

12 TxHeader b User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:26.0) Gecko/20100101 Firefox/26.0

12 TxHeader b Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

12 TxHeader b Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,en;q=0.3

12 TxHeader b Accept-Encoding: gzip, deflate

12 TxHeader b Cookie: Hm\_lvt\_b1f097642bd4f6603948b19d9f4560bf=1390275531,1390379376,1392271349,1392602258; Hm\_lvt\_b14dd8757ed9f7dd76e92b0d74271018=1390275531,1390379376,1392271349,1392602258; CNZZDATA1516938=cnzz\_eid%3D2054367192-1389686102-%26ntime%3D1392602285%26cnzz

12 TxHeader b Pragma: no-cache

12 TxHeader b Cache-Control: no-cache

12 TxHeader b X-Forwarded-For: 172.16.100.198

12 TxHeader b X-Varnish: 1766346205

12 RxProtocol b HTTP/1.1

第一列可以是任意的数字，它代表具体的请求。数字相同，表示他们是同属于一个HTTP事务的。第二列是日志信息的标签。所有的日志条目都是用一个标签去标记，该标签代表何种行为被记录。以Rx开头的标签代表varnish正在接受数据，Tx代表正在发送数据。  
第三列表示数据的是来自或者要发送给客户（c），另外，还有为b的情况，代表数据来自或要发送给后端（b）。第四列是被记录的数据。  
现在，你可以使用varnishlog去过滤下。基本的选项如下：  
-b 只显示varnish和后端服务器之间通信的记录条。当你想优化缓存命中率的时候，非常有用。  
-c 和-b类似，只是针对与客户端的通信情况。  
-i tag 只有显示带有特定标签的行。”varnishlog –I SessionOpen”将只显示新会话的情况。注意标签是大小写敏感的。  
-I 通过正则表达式过滤数据，并显示匹配行。”$varnishlog –c –I RxHeader –I Cookie”，将显示所有来自客户端的cookie头信息。  
-o 根据请求id，将记录条目分组。  
现在Varnish差不多工作正常，现在要将Varnish的端口编程80，进行调优。

## **Sizing your cache**

给Varnish选择多少内存，是个很艰巨的问题。你需要考虑以下事情：  
l  你的热门数据集有多大。对一个门户或者新闻站来说，这个数据集可能就只是首页和它相关内容的大小。这里包括的两部分，一部分是只首页本身的文字图片内容， 另一部分是首页会链接到的页面或对象（比如图片），这个很容易理解，首页的内容是最可能被点击的，命中率也会很高。  
l  产生一个对象的花费有多大？有时候，如果从后端返回并不太消耗资源，同时你的内存又有限的话，我们应该缓存一部分图片，而不是去缓存所有图片。  
l  使用varnishstat或其他工具监控n\_lru\_nuked计数器。如果你有很多LRU活动的话，那么你的缓存正因空间限制在清除对象，此时你就要考虑增加缓存大小了。  
清楚缓存任何对象都会携带保存在实际存储区域之外的开销。所以，即便你指定-s malloc,16G，varnish可能实际使用了两倍。Varnish中每个对象的花销大概是1k。所以，如果在你的缓存中有很多小对象的话，花销是非常大的。

## **Put Varnish on port 80**

直到现在，为了测试，我们都把varnish运行在一个高位端口上。你应该测试你的应用，如果它工作正常，我们就要切换了，Varnish运行在80端口上，你的web服务器运行在高位端口上。  
首先停止varnishd：

pkill varnishd

并停止你的web服务器。修改web服务器的配置，将其绑定到8080端口上，替换掉原来的80。现在打开varnish的default.vcl并且改变default后端的端口到8080。  
启动你的web服务器并且开启varnish：

varnishd -f /usr/local/etc/varnish/default.vcl -s malloc,1G -T 127.0.0.1:2000

注意，我们移除了-a选项。因为，现在我们采用Varnish的默认设置即可，它会自动绑定到http的80端口上。现在每个访问你站点的人，都会通过varnish访问。

## **Varnish Configuration Language – VCL**

Varnish有强大的配置系统。许多其他的系统使用配置指令，基本上就是开或关很多开关。Varnish使用领域专用语言（DSL）作为Varnish配置语言，简写VCL。当请求到达开始执行时，Varnish会将这些配置转换成二进制代码。  
VCL文件被分成多个子程序。不同的子程序在不同时候运行。有的在获得请求时候运行，有的当文件从后端获取后运行。

Varnish将在它工作的不同场景执行这些子程序。因为是代码，所以逐行执行并不是问题。在某些情况你在这个子程序调用一个action，然后该子程序执行结束。  
如果不想在你的子程序中调用一个action，并且到达了最末尾，此时varnish会执行一些VCL的内置代码。在default.vcl中的注释部分，你会看到这些VCL代码。  
99%的情况，你都会要改动两个子程序，vcl\_recv和vcl\_fetch。

### **vcl\_recv**

vcl\_recv是在请求开始时调用的。完成该子程序后，请求就被接收并解析了。用于确定是否需要服务请求，怎么服务，如果可用，使用哪个后端。  
在vcl\_recv中，你也可以修改请求。通常你可以修改cookie，或添加/移除请求头信息。  
注意在vcl\_recv中，只可以使用请求对象req。

### **vcl\_fetch**

vcl\_fetch是在文档从后端被成功接收后调用的。通常用于调整响应头信息，触发ESI处理，万一请求失败就换个后端服务器。  
在vcl\_fecth中，你还可以使用请求对象req。还有个后端响应对象beresp。Beresp包含了后端的HTTP头信息。

### **Actions**

最常用的action如下：  
**pass**：当返回pass的时候，请求和随后的响应都将被传到后端服务器，或从那里传回。不会被缓存。pass可以在vcl\_recv中被返回

**hit\_for\_pass**：类似与pass，但是只有vcl\_fetch可以用。不像pass，hit\_for\_pass将在缓存中创建一个 hitforpass对象。这有个副作用，就是缓存了不像缓存的东西。同时会将未缓存的请求传到后端。在vcl\_recv中这样的逻辑不是必须的，因为它 发生在任何潜在对象队列发生之前

**lookup**：当在vcl\_recv中返回lookup时，就等于你告诉varnish发送缓存中的内容，即使该请求应该是被pass的。在vcl\_fetch中不能使用lookup。

**pipe**：pipe也可以在vcl\_recv中返回。pipe缩短了客户和后端的环路链接，并且varnish将只是待在哪里，来回偏移字节。 varnish不会在意来回发送的数据，所以你的日志是不完整的。注意一个客户会基于相同链接发送几个请求，当使用HTTP 1.1时。所以在实际返回pipe之前，你需要让varnish添加”Connection:close”的头信息。

**deliver**：投递缓存对象给客户。经常在vcl\_fetch中使用。

### **请求，响应和对象**

在VCL中，有三种重要的数据结构。请求：来自客户端；响应：来自后端服务器；对象：存储在缓存中。  
在VCL中你应该知道以下结构。  
req：请求对象。当varnish接受了请求，req就会创建并生产。许多在vcl\_recv中要做的工作都需要用到req。  
beresp：后端响应对象。包含了从后端返回的对象的头信息。vcl\_fetch中，你会使用beresp对象。  
obj：缓存了的对象。大多数是驻留在内存中的只读对象。obj.ttl是可以写的，剩下的都是只读的。

### **操作符**

VCL中可用的操作符如下，稍后可以看例子：  
=：赋值  
==：比较  
~：匹配。可使用正则表达式或ACLs  
!：取反  
&&：逻辑与  
||：逻辑或

**例1 – 操作头信息**  
移除所有在web服务器的/images目录中的对象的cookie：

sub vcl\_recv {

  if (req.url ~ "^/images") {

    unset req.http.cookie;

  }

}

现在，当请求传到后端服务器时，他是不带有cookie头信息的。需要在意的是if语句，它根据正则表达式匹配了URL（属于请求对象）。注意匹配操作符。如果它匹配，请求的cookie头信息就会被删除。

**例2 – 操作beresp**

这里如果匹配某种条件，我们就重写beresp的TTL属性

sub vcl\_fetch {

   if (req.url ~ "\.(png|gif|jpg)$") {

     unset beresp.http.set-cookie;

     set beresp.ttl = 1h;

  }

}

**例3 – ACLs**

你创建了一个使用acl关键字的访问控制列表。你可以使用匹配操作符去判断客户端的IP地址是否与一个ACL匹配。

# Who is allowed to purge....

acl local {

    "localhost";

    "192.168.1.0"/24; /\* and everyone on the local network \*/

    ! "192.168.1.23"; /\* except for the dialin router \*/

}

sub vcl\_recv {

  if (req.request == "PURGE") {

    if (client.ip ~ local) {

       return(lookup);

    }

  }

}

sub vcl\_hit {

   if (req.request == "PURGE") {

     set obj.ttl = 0s;

     error 200 "Purged.";

    }

}

sub vcl\_miss {

  if (req.request == "PURGE") {

    error 404 "Not in cache.";

  }

}

## **Statistics**

现在你的varnish已经运行了，现在让我们看看它是如何工作的吧。有几个工具可以帮助你

### Varnishtop

varnishtop工具读取共享内存日志，并且显示一个持续更新的最常见的记录条的列表。  
通过使用-I,-i,-X和-x选项进行适当过滤，可以用于显示排名，有请求文档、客户端、用户代理（浏览器）或其他记录在日志中的信息。  
varnish –i rxurl将显示客户端请求的URL。varnishtop –i txurl 将显示你的后端被什么请求最多。varnishtop –i RxHeader –I Accept-Encoding将显示最常见的客户端发来的Accept-Encoding header

### varnishhist

varnishhist工具读取varnishd(1)共享内存日志，并且提供不断更新的直方图，用以展示它们处理的最近的N个请求的分布。N值和垂直刻度在左上角显示。水平刻度是对数的。命中用管道符号（“|”）标记，未命中使用哈希符号（“#”）标记。

### Varnishsizes

varnishsizes和varnishhist相似，但它会显示对象的大小，不显示完成请求消耗的时间。可以很直观的告诉你，你正在处理的对象有多大。

### Varnishstat

varnish有很多计数器。统计丢失数，命中数，存储信息数，创建了的线程数，删除的对象数，几乎一切。varnishstat将转存这些计数器。当对varnish进行调优时，这就很有用了。  
这里有一些程序可以定期获取varnishstat，很好地绘出这些计数器的图形。Munin是其中一个。你可以在http://munin-monitoring.org找到。在varnish的源码中有一个munin的插件

## **Achieving a high hitrate—高命中率**

现在Varnish已经运行，并且你可以通过Varnish访问你的站点。除非你的应用是专门为在一个web加速器后工作而写的，否则为了在Varnish中获得高命中率，你可能需要在配置或应用上做一些修改。  
除非varnish完全确定缓存你的数据室安全，否则varnish是不会缓存的。所以，为了让你明白varnish是如何确定的，并使如何缓存页面的，我将通过一些很有用工具去引导你：

注意你需要一个工具去观察传输于你和web服务器之间的HTTP头信息。在varnish服务器上，首先是使用varnishlog和varnishtop，但有时候需要客户端工具去搞清楚。下面就是我们用到的。

### 工具：varnishtop

你可以使用varnishtop确定出后端命中最多的URL。varnishtop –i txurl是必须的命令。你可以在前一节Statistics中，看到一些其他的varnishtop的例子。

### 工具：varnishlog

当你已经确定了最常发送给后端的URL是多少时，你可以使用varnishlog去查看完整的请求。varnishlog –c –o /foo/bar将给你来自客户端（-c）的完整（-o）的匹配/foo/bar的请求。  
对于扩展诊断头信息，可以参看：<http://www.varnish-cache.org/trac/wiki/VCLExampleHitMissHeader>

### 工具：lwp-request

lwp-request是perl的World-Wide Web library中的一部分。它是一些基本的程序，这些程序可以处理HTTP请求，并且给你结果。我通常使用两个程序，GET和HEAD。  
vg.no是第一个使用varnish的站点，站点的创建者很明白varnish。所以，让我们看看他们的HTTP头信息。我们对他们的首页发一个GET请求：

$ GET -H 'Host: www.vg.no' -Used <http://vg.no/>

GET <http://vg.no/>

Host: www.vg.no

User-Agent: lwp-request/5.834 libwww-perl/5.834

200 OK

Cache-Control: must-revalidate

Refresh: 600

Title: VG Nett - Forsiden - VG Nett

X-Age: 463

X-Cache: HIT

X-Rick-Would-Never: Let you down

X-VG-Jobb: <http://www.finn.no/finn/job/fulltime/result?keyword=vg>+multimedia Merk:HeaderNinja

X-VG-Korken: <http://www.youtube.com/watch?v=Fcj8CnD5188>

X-VG-WebCache: joanie

X-VG-WebServer: leon

OK。我们来解释一下。GET通常发送HTTP0.9请求，其缺少Host头信息。所以我通过-H添加了一个Host头信息。-U打印请求头信息，-s打印响应状态，-e打印响应头信息，-d丢弃实际内容。我们并不关心内容，只要头信息。

你可以发现，VG在他们的头信息中添加了一些信息。像X-Rick-Would-Never表示了vg.no的某种奇怪的幽默。其他，像X-VG-Webcache是用来调试的。  
所以，你可以检查一个站点的特定URL是否设置了cookie，只需要：

GET -Used <http://example.com/> |grep ^Set-Cookie

### 工具：实时HTTP头信息

Firefox也有个插件。Live HTTP Headers可以显示你发送和接受到的头信息。你可以通过google找到“Live HTTP Header”，或者到https://addons.mozilla.org/en-US/firefox/addon/3829/可以找到。  
HTTP头信息的角色  
随着每个HTTP请求和响应变成一群携带原数据的头信息。varnish将查看这些头信息，以确定这里的内容是否适合缓存，以及缓存多久。  
请注意，当考虑这些头信息时候，实际上varnish只考虑真实web服务器中varnish自己的那部分。考虑的理论依据都在于你的控制。  
术语surrogate origin cache没有在IETF so RFC2616中很好的定义。所以varnish不同的工作方式可能会和你的预期不同。  
让我们看看你应该知道的重要的头信息：

#### **Cache-Control**

Cache-Control指定了缓存如何处理内容。varnish关心max-age参数，并用它来计算对象的TTL。  
“Cache-Control:nocache”是被忽略的，如果你需要，你也可以方便的增加支持。  
所以，确保你发出的Cache-Control头信息具有max-age。你可以看看Varnish软件的联盟服务器发出了什么：

|  |  |
| --- | --- |
| Cache-Control | public, max-age=21600 |

#### **Age**

|  |  |
| --- | --- |
| Age | 6590 |

varnish添加了一个Age头信息，以指示在Varnish中该对象被保持了多久。你可以通过varnishlog像下面那样抓出Age

varnishlog -i TxHeader -I ^Age

#### **Pragma**

一个HTTP 1.0服务器可能会发送”Pragma:nocache”。Varnish忽略这种头信息。在VCL中你可以很方便的增加对这种头信息的支持。  
在vcl\_fetch中：

if (beresp.http.Pragma ~ "nocache") {

   pass;

}

#### **Authorization**

如果varnish看到授权头信息时，它会pass该请求。如果这不是你希望的，你可以unset这个头信息。

#### Overriding the time-to-live(ttl)

有时你的后端会误操作。根据你的安装，在varnish中覆写ttl会比修复你某些麻烦的后端要简单的多。  
你需要VCL去确定你想要的对象，然后你将beresp.ttl的值设置成你想设置的值。

sub vcl\_fetch {

    if (req.url ~ "^/legacy\_broken\_cms/") {

        set beresp.ttl = 5d;

    }

}

该例会为你网站上过去的遗留物，将TTL设置为5天

#### Forcing caching for certain requests and certain responses

由于你还在使用那些麻烦的不能很好工作的后端，你可能会想再varnish中覆写更多的内容。我们推荐你尽可能多的使用默认缓存规则。尽管强制varnish在缓存中查找一个对象很简单，但我们还是不推荐

#### Normalizing your namespace

一些站点可以通过很多主机名访问。比如， http://www.varnish-software.com/ , http://varnish-software.com/和http://varnishsoftware.com/。这些都指向同一个站点。由于 varnish不知道他们的区别，varnish会为每个主机名的每个页面做缓存。你可以通过在你的web服务器配置中设置跳转或者使用VCL来缓解这个 情况：

if (req.http.host ~ "(?i)^(www.)?varnish-?software.com") {

  set req.http.host = "varnish-software.com";

}

## **Cookies**

varnish不会缓存来自后端的具有Set-Cookie头信息的对象。同样，如果客户端发送了一个Cookie头信息，varnish将绕过缓存，直接发给后端。  
这可能太过保守。很多站点使用Google Analytics(GA)去分析他们的流量。GA通过设置cookie去跟踪。这个cookie是供客户端的JavaScript程序使用的，服务器是不需要的。  
对很多web应用，它会完全忽视cookies，除非你正在访问网站的特定部分。在vcl\_recv中的下面的VCL代码块将忽略cookies，除非你访问/admin/：

if ( !( req.url ~ ^/admin/) ) {

  unset req.http.Cookie;

}

非常简单。然后，如果你需要做更复杂的处理，像在几个cookie中移除其中一个，事情就麻烦了。遗憾的是，varnish没有很好的工具去操作 cookie。我们不得不使用正则表达式去做这件事情。如果你熟悉正则表达式，你就明白怎么办了。如果你不熟悉，我建议你找本书好好学习下，或者通过 pcrepattern手册页面或其他的在线教程。  
让我展示给你看看，varnish软件是用什么的。我们使用一些GA的跟踪cookie或其他类似工具的cookie。这些cookie都是供 JavaScript使用的。varnish和其联盟站不需要这些cookie，并且因为varnish会因为这些cookie不缓存页面，所以当客户端 发送cookie时，我们将在VCL中丢弃这些非必要的cookie。  
下面的VCL中，我们丢弃了所有的以“\_”开头的cookie：

/ Remove has\_js and Google Analytics \_\_\* cookies.

set req.http.Cookie = regsuball(req.http.Cookie, "(^|;\s\*)(\_[\_a-z]+|has\_js)=[^;]\*", "");

// Remove a ";" prefix, if present.

set req.http.Cookie = regsub(req.http.Cookie, "^;\s\*", "");

## **Vary**

Vary头信息是web服务器发送的，代表什么引起了HTTP对象的变化。可以通过Accept-Encoding这样的头信息弄明白。当服务器发 出”Vary:Accept-Encoding”，它等于告诉varnish，需要对每个来自客户端的不同的Accept-Encoding缓存不同的版 本。所以，如果客户端只接收gzip编码。varnish就不会提供deflate编码的页面版本。  
如果Accept-Encoding字段含有很多不同的编码，比如浏览器这样发送：

Accept-Encodign: gzip,deflate

另一个这样发送:

Accept-Encoding: deflate,gzip

因为Accept-Encoding头信息不通，varnish将保存两种不同的请求页面。规范Accept-Encoding头信息将确保你的不同尽可能的少。下面的VCL代码将规范Accept-Encoding的头信息：

if (req.http.Accept-Encoding) {

    if (req.url ~ "\.(jpg|png|gif|gz|tgz|bz2|tbz|mp3|ogg)$") {

        # No point in compressing these

        remove req.http.Accept-Encoding;

    } elsif (req.http.Accept-Encoding ~ "gzip") {

        set req.http.Accept-Encoding = "gzip";

    } elsif (req.http.Accept-Encoding ~ "deflate") {

        set req.http.Accept-Encoding = "deflate";

    } else {

        # unkown algorithm

        remove req.http.Accept-Encoding;

    }

}

该代码设置了来自客户端的Accept-Encoding头信息，gzip具有更高优先级。

## **Pitfall – Vary:User-Agent**

一些应用或应用服务器，会随它们的内容发送”Vary:User-Agent”。这指示Varnish对每个不同的User-Agent缓存不同的副本。 这非常的多。甚至相同浏览器的一个补丁都至少会产生10中不同的User-Agent头信息，这个产生的不同是和浏览器所运行的操作系统有关。  
所以，如果你真的需要基于User-Agent变化，要确保规范头信息，否则你的命中率会非常的差。可以利用上面的代码作为模板。

## **Purging and banning**

最有效提升命中率的方法是增加你对象的ttl(time-to-live存活时间)。但是，你要知道，在微博时代，提供过时的内容是很不利于业务的。  
解决方案是，当有新内容时，就通知varnish。这可以通过两个机制实现。HTTP清理(PURGE，以下称PURGE)和禁止(BAN，以下简称BAN)。首先让我们解释下HTTP PURGE

### HTTP PURGE

PURGE(清理)是指当你选出一个缓存对象时，根据其变化的内容进行丢弃。通常PURGE是通过HTTP的PURGE方法进行调用的（即method=purge，这个purge是http协议中没有预定义的，应该是varnish中扩展的）。  
HTTP PURGE类似于HTTP GET请求，只是method是PURGE。事实上，你可以调用任何你希望的method，不过许多人都倾向于使用PURGE。Squid支持相同的机制。为了在varnish支持PURGE，你需要以下代码：

acl purge {

        "localhost";

        "192.168.55.0/24";

}

sub vcl\_recv {

        # allow PURGE from localhost and 192.168.55...

        if (req.request == "PURGE") {

                if (!client.ip ~ purge) {

                        error 405 "Not allowed.";

                }

                return (lookup);

        }

}

sub vcl\_hit {

        if (req.request == "PURGE") {

                purge;

                error 200 "Purged.";

        }

}

sub vcl\_miss {

        if (req.request == "PURGE") {

                purge;

                error 200 "Purged.";

        }

}

正如你看到的。我们使用了新的VCL子程序，vcl\_hit和vcl\_miss。当我们调用lookup时，varnish将尝试在缓存中查找对象。要么命中，要么丢失，然后调用相应的子程序。在vcl\_hit中，我们可以获得存于缓存中的对象，并且可以设置它的TTL。  
所以，对于example.com，要让它的首页失效（表示要拿新的内容），可以这样请求varnish

PURGE / HTTP/1.0

Host: example.com

之后，Varnish就会丢弃主页。这会移除所有变量，如vary所定义的。

### Ban

这是另一个让内容失效的方法。禁止（BAN），你可以将其认为是一种过滤器。你禁止你的缓存提供某些内容。你可以根据我们有的元数据，进行禁止。  
Varnish支持禁止功能，并且可以再CLI接口中获得。对于VG，如果想禁止属于example.com的png对象，他们可以分发以下内容：

ban req.http.host == "example.com" && req.http.url ~ "\.png$"

真的很强大。  
当在缓存中命中对象时且在投递之前，就会检查其是否BAN。一个对象只会被较新的BAN检查。  
只对beresp.\*起作用的BAN，由背景工作线程运行着，称为ban lurker。ban lurker将检查堆，看看是否匹配对象，并且去除匹配对象。ban lurker的频度（活跃度），可以通过ban\_lurker\_sleep参数控制。  
禁止那些较老的，对于缓存中最老的对象不经验证就直接丢弃。如果你有很多具有长TTL对象，这些对象很少被访问，那么你会累积大量的禁止。这会影响CPU的利用率和性能。  
你可以通过HTTP向varnish添加BAN。这样做需要一些VCL：

sub vcl\_recv {

        if (req.request == "BAN") {

                # Same ACL check as above:

                if (!client.ip ~ purge) {

                        error 405 "Not allowed.";

                }

                ban("req.http.host == " + req.http.host +

                      "&& req.url == " + req.url);

                # Throw a synthetic page so the

                # request won't go to the backend.

                error 200 "Ban added";

        }

}

该VCL代码段启用varnish，去处理一个HTTP BAN method，对URL添加禁止，包括host部分。

## **Edge Side Includes**

Edge Side Includes（边界情况包含）是一种语言，用来包含在其他web页面中的web页面片断。可以认为他是一个通过HTTP实现的HTML包含语句。  
在许多web站点，许多内容是各页面间共享的。为每个页面重新生成这些内容是很浪费的，并且ESI（Edge Side Includes的缩写）致力于让你为每个片断单独决定缓存策略。  
在varnish中，我们只实现了ESI的一个小的子集。自2.1起，我们就有三个ESI语句：

* esi:include
* esi:remove
* <!–esi …–>

基于变量和cookie的内容替换还没有实现，但是已经在计划中了。  
例子：esi include  
让我们看看如何使用它。这段简单的cgi脚本，输出了日期：

#!/bin/sh

echo 'Content-type: text/html'

echo ''

date "+%Y-%m-%d %H:%M"

现在，让我们做个包含ESI include语句的HTML文件：

<HTML>

<BODY>

The time is: <esi:include src="/cgi-bin/date.cgi"/>

at this very moment.

</BODY>

</HTML>

要让esi工作，你需要在VCL中激活ESI，比如像下面那样：

sub vcl\_fetch {

    if (req.url == "/test.html") {

       set beresp.do\_esi = true; /\* Do ESI processing               \*/

       set beresp.ttl = 24 h;    /\* Sets the TTL on the HTML above  \*/

    } elseif (req.url == "/cgi-bin/date.cgi") {

       set beresp.ttl = 1m;      /\* Sets a one minute TTL on        \*/

                                 /\*  the included object            \*/

    }

}

例子：esi remove  
该remove关键字，允许你remove输出。当ESI无法获得时，你可以使用此，做各种各样的回退，代码如下：

<esi:include src="<http://www.example.com/ad.html>"/>

<esi:remove>

  <a href="<http://www.example.com>">www.example.com</a>

</esi:remove>

例子：<!—esi…–>  
这是一个特殊的构造，允许ESI标记的HTML呈现，而无需处理。当处理页面时，ESI处理器将移除开始（<–esi）和结尾(–>)，然而仍然会处理其内容。如果页面没有被处理，它将会留下，编程HTML/XML的注释标签。例如：

<!--esi

<p>Warning: ESI Disabled!</p>

</p>  -->

这保证了如果没有处理ESI标记，它也不会影响最后HTML的呈现。

## **Running inside a virtual machine(VM)**

虽然可以将varnish运行在虚拟的硬件上，但是出于高性能，我们不建议这样。

OpenVz

如果你运行在64位OpenVz（或并行VPS），你必须在启动varnish前减少最大栈尺寸。默认分配给每个线程的内存有点多，这会导致varnish随着线程数（==流量）增加而down掉。  
在启动脚本中，运行以下，降低最大栈尺寸：

ulimit -s 256

## **Advanced Backend Configuration**

某些情况，你可能需要让varnish缓存几个服务器的内容。你可能希望varnish映射所有URL到一个或多个主机。这里有许多选项。

比如，我们需要引入一个Java应用到PHP网站。我们的Java应用会处理以/java/开头的URL。  
我们将东西起起来，运行在8000端口。现在来看看default.vcl

backend default {

    .host = "127.0.0.1";

    .port = "8080";

}

我们添加新的后端：

backend java {

    .host = "127.0.0.1";

    .port = "8000";

}

现在我们要指示，发送不同URL的规则。看看vcl\_recv：

sub vcl\_recv {

    if (req.url ~ "^/java/") {

        set req.backend = java;

    } else {

        set req.backend = default.

    }

}

非常简单。现在先让我们停一下，考虑一下这里的情况。如你所见，你可以根据任意情况定义如何选择后端。如果你发送移动设备的请求到不同的后端，可以做类似的操作，if(req.User-agent ~ /mobile/)

## **Directors**

你也可以将几个后端分组为一组后端。这些组称为directors。这可以提高性能和灵活度。你可以定义几个后端，并把它们归到一个director中

backend server1 {

    .host = "192.168.0.10";

}

backend server2{

    .host = "192.168.0.10";

}

现在创建一个director：

director example\_director round-robin {

{

        .backend = server1;

}

# server2

{

        .backend = server2;

}

# foo

}

该director是一个循环director。这代表该director将根据循环基础分发进入的请求。这也是一个随机director，它以随机风格分发请求。  
但是，如果你的一个服务器down了怎么办？varnish可以将所有的请求，指向一个健康的服务器吗？当然了。这就是下面要说的健康检查。

## **Health checks**

让我们建立一个director，该director具有两个后端并带健康检查。首先让我们定义后端

backend server1 {

  .host = "server1.example.com";

  .probe = {

         .url = "/";

         .interval = 5s;

         .timeout = 1 s;

         .window = 5;

         .threshold = 3;

    }

  }

backend server2 {

   .host = "server2.example.com";

   .probe = {

         .url = "/";

         .interval = 5s;

         .timeout = 1 s;

         .window = 5;

         .threshold = 3;

   }

 }

probe是新的内容。varnish将使用probe对每个后端进行健康检查。其选项有：  
url：定义什么样的URL需要varnish（感觉这里应该是做处理的意思）请求。  
interval：查询的间隔时长  
timeout：probe的超时时间  
window：varnish将保持一个结果的滑动窗（该滑动窗不是实际的窗体，是一种流量控制方法，允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。）。这里窗口有5个确认点。  
threshold：上次查询的.window数量为多少时，就代表后端是健康的。  
initial：当varnish启动时候，用多少个probe去探测健康情况——默认情况，此数量与threshold的数量一致。  
现在我们定义director：

director example\_director round-robin {

      {

              .backend = server1;

      }

      # server2

      {

              .backend = server2;

      }

}

使用这个director，就通你使用其他director或后端一样。varnish不会发送流量到那些标记为不健康主机。  
如果所有的后端都down掉了，varnish也会提供老的内容。关于如何启用这个功能，可以参看Misbehaving servers。  
请注意，varnish将为所有已加载的VCL，持续探测是否活动（active）。varnish将合并相同的probe——所以，注意，如果你做了很多VCL加载，就不要改变probe的配置。卸载VCL，将丢弃probe。

## **Misbehaving servers**

varnish有个重要的特性，它可以保护你免受web-和应用服务器的不良行为。

### Grace mode

当几个客户端正访问相同页面时，varnish会发送一个请求到后端，并且让其他请求等待，当从后端取回一个副本时。在某些产品中，这称为请求黑那个，varnish会自动做这个。  
如果你每秒需要相应成千上万的点击，等待的请求队列就会很巨大。这里有两个潜在问题，一个是thundering herd problem（这个无法翻译。。。wiki有专门的对应解释），突然增加一千个线程去提供内容，会让负载变得很高。第二个，没有人喜欢等。为了解决这个 问题，我们指示varnish去保持缓存的对象超过他们的TTL（就是该过期的，不让它过期），并且去提供旧的内容给正在等待的请求。  
所以，为了提供旧的内容，首先我们必须有内容去提供。所以，我们使用以下VCL，以使varnish保持所有对象超出了他们的TTL30分钟。

sub vcl\_fetch {

  set beresp.grace = 30m;

}

这样，varnish还不会提供旧对象。为了启用varnish去提供旧对象，我们必须在请求上开启它。下面表示，我们接收15s的旧对象：

sub vcl\_recv {

  set req.grace = 15s;

}

你可能想知道，为什么，如果我们无法提供这些对象，我们在缓存中保持这些对象30分钟？如果你开启健康检查，你可以检查后端是否出问题。如果出问题了，我们可以提供长点时间的旧内容

if (! req.backend.healthy) {

   set req.grace = 5m;

} else {

   set req.grace = 15s;

}

所以，总结下，优雅模式解决了两个问题：  
1、         通过提供旧的内容，避免请求扎堆。  
2、         如果后端坏了，提供旧的内容。