

Varnish 文件缓冲的实现

1、Varnish 文件缓存的整体思路及优点

Varnish 将所有的 HTTP object 存于一个单独的大文件中，而该文件在工作进程初始时就将其整个 mmap 到内存中。Varnish 在该块内存中实现类似于一个简单的“文件系统”，具有分配、释放、修剪、合并等功能。

Varnish 文件缓存的优点，就如其创始人 Poul-Henning Kamp 所说，它是一个具有“现代设计理念”的软件，其整体设计优点可以从[1]得知。我觉得它的优点主要有两点：1、它避免了软件与系统对内存控制的冲突，引入了虚拟内存的概念，将内存与硬盘文件统一，软件只需要注重对内存的操作即可；2、它将所有的 object 存于一个文件中，避免类似 Squid 为每个 object 存放一个小文件的设计，减少文件系统频繁的操作。当然这需要记录每个 object 在内存中的偏移量，并定期检查并释放内存，进而增加编程难度，但同时提高了系统的性能。

2、Varnish 文件缓存的工作流程

Varnish 与一般服务器软件类似，分为 master (management) 进程和 child (worker, 主要做 cache 的工作) 进程。master 进程读入命令，进行一些初始化，然后 fork 并监控 child 进程。child 进程分配若干线程进行工作，主要包括一些管理线程和很多 worker 线程。

针对文件缓存部分，master 读入存储配置 (-s file[,path[,size[,granularity]]])，调用合适的存储类型，然后创建/读入相应大小的缓存大文件（根据其 man 文档，为避免文件出现存储分片[2]影响读写性能，作者建议用 dd(1)命令预先创建大文件）。接着，master 初始化管理该存储空间的结构体。这些变量都是全局变量，在 fork 以后会被 child 进程所继承（包括文件描述符）。

在 child 进程主线程初始化过程中，将前面打开的存储大文件整个 mmap 到内存中（如果超出系统的虚拟内存，mmap 失败，进程会减少原来的配置 mmap 大小，然后继续 mmap），此时创建并初始化空闲存储结构体，挂到存储管理结构体，以待分配。

接着，真正的工作开始，Varnish 的某个负责接受新 HTTP 连接的线程开始等待用户（具体过程可参见[3]），如果有新的 HTTP 连接过来，它总负责接收，然后叫醒某个等待中的线程，并把具体的处理过程交给它。

Worker 线程读入 HTTP 请求的 URI，查找已有的 object，如果命中则直接返回并回复用户。如果没有命中，则需要将所请求的内容，从后端服务器中取过来，存到缓存中，然后再回复。

分配缓存的过程是这样的：它根据所读到 object 的大小，创建相应大小的缓存文件。为了读写方便，程序会把每个 object 的大小变为最接近其大小的内存页面倍数。然后从现有的空闲存储结构体中查找，找到最合适的大小的空闲存储块，分配给它。如果空闲块没有用完，就把多余的内存另外组成一个空闲存储块，挂到管理结构体上。如果缓存已满，就根据 LRU[4]机制，把最旧的 object 释放掉。

释放缓存的过程是这样的：有一个超时线程，检测缓存中所有 object 的生存期，如果超初设定的 TTL (Time To Live) 没有被访问，就删除之，并且释放相应的结构体及存储内存。注意释放时会检查该存储内存块前面或后面的空闲内存块，如果前面或后面的空闲内存和该释放内存是连续的，就将它们合并成更大一块内存。

整个文件缓存的管理，没有考虑文件与内存的关系，实际上是将所有的 object 都考虑是在内存中，如果系统内存不足，系统会自动将其换到 swap 空间，而不需要 varnish 程序去控制。

3、文件缓存的数据结构及其操作

3.1 基本数据结构

尾队列

由于 Poul-Henning Kamp 是 FreeBSD 的内核维护者，Varnish 代码中受 FreeBSD 编程风格的影响较大。程序中使用最多的是名叫尾队列（tail queue）的数据结构，是其四种基本数据结构中的一种：单向列表(single-linked lists)、单向尾队列(single-linked tail queue)、列表(lists)、尾队列(tail queues)。全部的数据结构定义可以在 `varnish-dist/include/vqueue.h` 中找到，附录 1 中有尾队列全部操作的定义。参考文献[5]提到了这些数据结构，并简单提到了这样写的原因，建议一看。

尾队列头部实际上是指向头部成员和尾部成员的结构体，基本定义如下：

```
#define VTAILQ_HEAD(name, type) \
struct name { \
    struct type *vtqh_first; /* first element */ \
    struct type **vtqh_last; /* address of last next element */ \
}
```

尾队列成员入口实际上是一个双向链表，但又有不同，基本定义如下：

```
#define VTAILQ_ENTRY(type) \
struct { \
    struct type *vtqe_next; /* next element */ \
    struct type **vtqe_prev; /* address of previous next element */ \
}
```

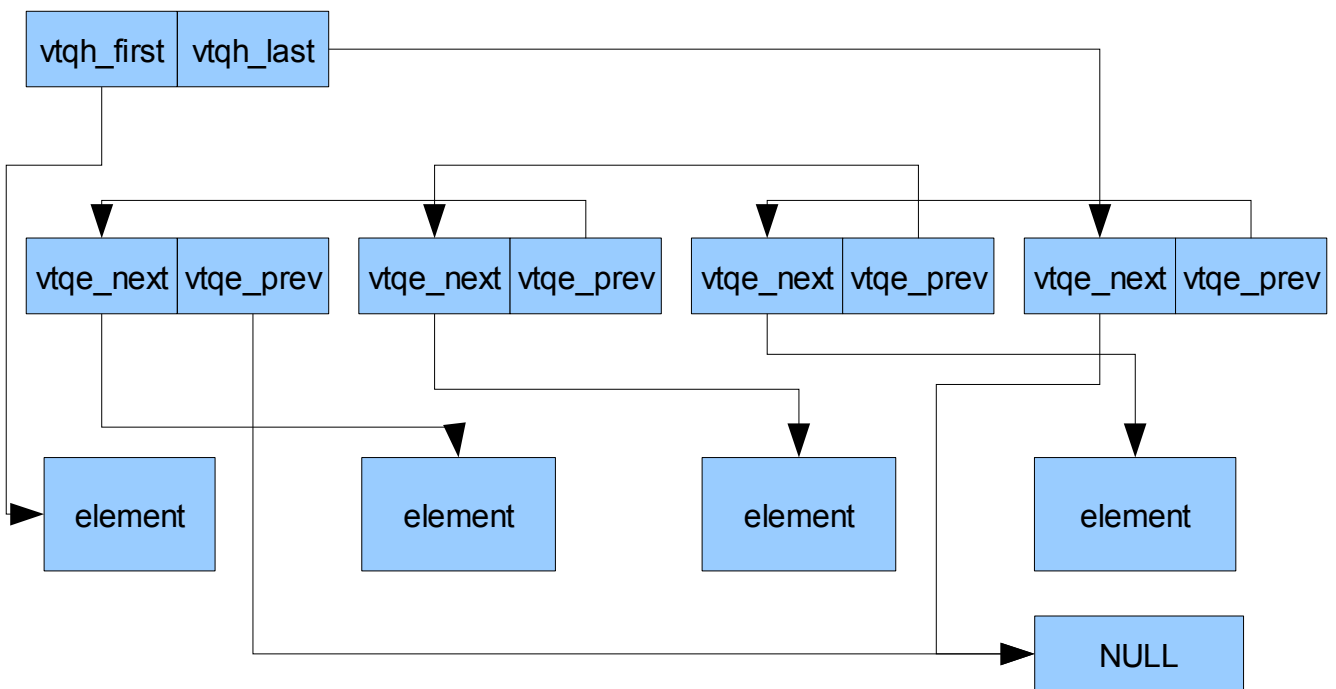


图 1 尾队列示意图

特别要注意的是，尾队列成员入口的 `prev` 是双重指针，指向上一个成员的 `next` 指针。这样做主要为了通用，即使 `element` 成员是不同类型也可以组成链表。这与我们一般教科书上

的 prev 操作不一样。

所以其 last 和 prev 的宏定义也颇为让人费解：

```
#define VTAILQ_LAST(head, headname) \
    (((struct headname *)((head)->vtqh_last))->vtqh_last)

#define VTAILQ_PREV(elm, headname, field) \
    (((struct headname *)((elm)->field.vtqe_prev))->vtqh_last)
```

注意尾队列的 head 和 entry 结构体的定义是一样的，所以在 element 类型是一样的时候，VTAILQ_LAST(head, headname)也等于*(struct type *)head->vtqh_last->vtqe_prev，相应的 VTAILQ_PREV(elm, headname, field)也等于*(struct type *)elm->field.vtqe_prev->vtqe_prev。FreeBSD 写成这样诡异的形式据说是为了可以方便得删除不同类型的 element。

二叉堆

二叉堆[6]由于其查找迅速（查找最值的复杂度为 $O(\log(n))$ ）稳定，在大数据量下的最大/最小值查找运用广泛，特别是游戏中的最短路径搜索中，作为基础存储结构。其插入和删除的复杂度也为 $\log(n)$ ，而且二叉堆的数据可以仅仅存在一个一维的数组中。

在 varnish 中，object 的 TTL 就存储在二叉堆中，以便迅速找到最近超时的结构体。其结构体的定义在 varnish-dist/lib/libvarnish/binary_heap.c 中：

```
struct binheap {
    unsigned          magic;
#define BINHEAP_MAGIC    0xf581581aU    /* from /dev/random */
    void              *priv;    /*NULL*/
    binheap_cmp_t     *cmp;    /*比较函数*/
    binheap_update_t  *update; /*更新函数*/
    void              **array; /*指向保存二叉堆的一维数组*/
    unsigned          length; /*二叉堆的长度，最小为 16*/
    unsigned          next;    /*下一个空闲的二叉堆节点，根的 index 为 1*/
    unsigned          granularity; /*bh->granularity = getpagesize() / sizeof *bh->array*/
};
```

二叉堆的建立（binheap_new）、插入（binheap_insert）、删除（binheap_delete）函数都在该文件中实现，比较简单。

update 和 cmp 函数均在 varnish-dist/bin/varnishd/cache_expire.c 中实现，进行数据的更新和大小的比较。

3.2 文件缓存的物理存储及管理结构体的数据结构

Varnish 的物理存储的管理结构体名称是 struct smf_sc，其定义在 varnish-dist/bin/varnishd/storage_file.c 中：

```
VTAILQ_HEAD(smfhead, smf); /*struct name, type*/
struct smf_sc {
    const char        *filename; /*缓存大文件的名称*/
    int                fd;        /*缓存大文件的文件描述符*/
    unsigned          pagesize; /*内存的页面大小，从系统调用 getpagesize()获得*/
    uintmax_t         filesize; /*缓存大文件的大小*/
};
```

```

struct smfhead  order; /*以内存地址排序的 smf 尾队列头部*/
struct smfhead  free[NBUCKET]; /*空闲的 smf 尾队列头部，NBUCKET 表明的是空
                                闲页面数，大于 NBUCKET-1 空闲页面数的 smf,
                                都挂在 free[NBUCKET-1]这个尾队列上*/

struct smfhead  used; /*正在被使用的 smf*/
MTX             mtx; /*#define MTX pthread_mutex_t*/
};

```

Varnish 的物理存储的基本结构体名称是 struct smf，实际含义是缓存大文件中一小块连续的内存，用以分配给相应大小的 HTTP 对象，其定义也在 varnish-dist/bin/varnishd/storage_file.c 中：

```

struct smf { /*Storage Mmapped File*/
    unsigned      magic; /*魔法数，一般用来检验该结构体是否有效*/
#define SMF_MAGIC      0x0927a8a0

    struct storage s; /*上层逻辑存储结构体*/
    struct smf_sc  *sc; /*smf 管理结构体*/
    int            alloc; /*是否分配完成*/
    off_t          size; /*smf 的大小*/
    off_t          offset; /*已经分配过的内存偏移*/
    unsigned char  *ptr; /*指向可分配内存的起始地址*/
    VTAILQ_ENTRY(smf) order; /*按内存地址排序的 smf 入口*/
    VTAILQ_ENTRY(smf) status; /**/
    struct smfhead *flist; /*指向 smf_sc 中 free [n]相应的空闲尾队列头部*/
};

```

结构体 smf_sc 和 smf 是 Varnish 操作文件缓存的物理接口，主要操作函数有：init、open、alloc、trim 和 free，其函数名称均为 smf_*，函数原型都在 varnish-dist/bin/varnishd/storage_file.c。这些函数都被通过函数指针被挂到逻辑操作结构体 stevedore 中（详见 3.3）。

init 函数在 master 进程初始化时被调用，用以打开相应大小的文件及初始化全局变量 smf_sc 结构体，打开的具体行为为 open(fd, O_RDWR | O_CREAT | O_EXCL, 0600)，init 函数的伪代码如下：

```

{
    从命令行读入缓存文件名称、大小
    获取内存页面大小
    分配并初始化 smf_sc 结构体
    初始化 smf_sc 中的各尾队列成员
    open 缓存文件
    计算命令行需要的文件大小，使用 ftruncate 来进行调整文件
}

```

`open` 函数在 `child` 进程初始化时被调用，其函数主体是 `smf_open_chunk()`，`mmap` 已经打开的文件到内存，然后创建第一个 `smf` 结构体。`mmap` 的函数调用是这样的 `mmap(NULL, sz, PROT_READ|PROT_WRITE, MAP_NOCORE | MAP_NOSYNC | MAP_SHARED, sc->fd, off)`。我发现 `MAP_NOCORE` 和 `MAP_NOSYNC` 这两个标识在 Linux 中是没有的，`MAP_SHARED` 的含义也是不一样的。在 FreeBSD 中，`MAP_NOCORE` 表明映射区域不包括 `core` 文件，`MAP_NOSYNC` 表明虚拟内存中改动过的数据不会自动与文件系统中数据同步（在 FreeBSD 的 `manual` 中说明，如果需要同步，可以使用 `fsync(2)`），而 `MAP_SHARED` 表明不同进程对虚拟内存改动是共享的；相应地，在 Linux 中，`MAP_SHARED` 的含义是 FreeBSD 中 `MAP_NOSYNC` 和 `MAP_SHARED` 标识的联合（在 Linux `manual` 中说明，如果需要同步，可以用 `msync(2)` 和 `munmap(2)` 实现）。`smf_open_chunk` 函数的伪代码为：

```
{
    把缓存文件 mmap 到内存
    IF mmap 成功
        建立一个新的 smf 结构体，指向映射到内存的连续地址
    ELSE
        减小一次 mmap 的大小
        再多次 mmap
    ENDIF
}
```

`alloc` 函数是在创建 `object` 首先需要调用的函数，它主要是用来分配相应的物理内存大小。`alloc` 分配的内存大小必定是页面大小的整数倍，由于系统对内存的操作单位是单个页面大小，所以为了将每个 `object` 的内存操作不会影响到其他 `object`，Varnish 中将每个 `object` 用页面大小来对齐（当然这会浪费一部分的内存），其函数的伪代码为：

```
{
    根据原有的请求大小，找到最接近的整数倍内存页面大小
    从 smf_sc 中的 free 尾队列数组中，找到拥有相应页面数的队列
    从上述尾队列中为 storage 分配分配一个 smf
    把 smf 从 free 队列中删除
    从该 smf 中内存分配相应内存
    IF 该 smf 还有内存剩余
        把这个 smf 分裂出 smf2，smf2 为被使用掉的内存结构体
        在 order 队列中，把 smf2 插入原有的 smf 成员之前
        在 used 队列中，把 smf2 插入到尾部
        在 free 队列中，把 smf 插入到尾部
    ELSE
        smf2=smf
    ENDIF
    将 smf2 分配给 storage 逻辑结构体
    返回 storage
}
```

有时候，调用 `smf_alloc` 的时候并不知道到底需要多少，那就先实现分配一些内存（默认是 128K，也可以设置 `fetch_chunksize` 的值，分配大小为 `fetch_chunksize * 1024`）。当使用完内存后，还有一些内存剩余，这时候，就用到 `trim` 函数来裁剪内存，首先新建一

个 smf2，把剩余的内存从原来已经分配出去的 smf 结构中拿过来，分配给 smf2，smf2 插入到 free 的尾队列中等等。

free 函数在有个名叫 exp_timer 的线程中不断被调用，它会定期查看所有 object 的 TTL (default_ttl，默认是 120 秒)，如果有超时，就删除 object，并释放相应的 smf。free 函数的一个特点是，在删除时，会查看 order 序列上的内存块，如果相邻的内存块是空闲的，就合并成更大的内存块。

上面这些操作是如何进行同步的呢？

因为 Varnish 的操作基本上都是在线程中实现，针对全局变量的一些非原子操作，默认作者通过最简单的 pthread_mutex_lock 操作来进行互斥操作。

3.3 文件缓存的逻辑分配、操作结构体

storage 结构体是 Varnish 中存放每个 object 的地方，可以方便得使用其中分配的内存。其结构体定义的位置为：varnish-dist/bin/varnishd/cache.h

```
struct storage {
    unsigned          magic;
#define STORAGE_MAGIC    0x1a4e51c0
    VTAILQ_ENTRY(storage) list; /*访问入口*/
    struct stevedore *stevedore; /*指向为其分配内存的 stevedore*/
    void             *priv; /*指向其内存对应的 smf*/
    unsigned char *ptr; /*内存的指针*/
    unsigned        len; /*内存已使用的长度*/
    unsigned        space; /*分配的内存大小*/
    int             fd; /*smf->sc->fd*/
    off_t          where; /*smf->offset*/
};
```

stevedore 是操纵 storage 的结构体，它在 master 进程读取命令行的时候就首先建立一个 stevedores 的全局变量，然后根据相应的存储类型进行初始化，以便确定后续内存操作函数的指针。其结构体定义的位置为：varnish-dist/bin/varnishd/stevedore.h

```
struct stevedore {
    unsigned        magic;
#define STEVEDORE_MAGIC    0x4baf43db
    const char     *name;
    storage_init_f  *init; /* 被 master 进程调用*/
    storage_open_f  *open; /* 被 child 进程调用 */
    storage_alloc_f *alloc; /*smf_alloc*/
};
```

```

storage_trim_f      *trim; /*smf_trim*/
storage_free_f      *free; /*smf_free*/
/* private fields */
void                *priv;    /*指向 smf_sc*/
VTAILQ_ENTRY(stevedore) list;
};

```

上面对于 Varnish 如何分配使用内存应该可以知道个大概了。

3.4 HTTP 对象、hash 存储等

Varnish 的核心作用就是缓存 HTTP object，而 object 的查找是通过 hash 的方法来实现的，其内存的组织形式也是挂在相应桶的 hash 结构体下面（默认桶有 16383 个）。

hash 结构体的入口是全局变量 hcl_head（classic 是默认的 hash 方法，当然你也可以选择 simple_list），在子进程初始化时分配内存，hash 结构体的头部和入口定义均在 varnish-dist/bin/varnishd/hash_classic.c:

```

struct hcl_entry {
    unsigned        magic;
#define HCL_ENTRY_MAGIC      0x0ba707bf
    VTAILQ_ENTRY(hcl_entry) list;
    struct hcl_hd    *head; /*指向 hash 桶的头部*/
    struct objhead   *oh; /*objhead*/
    unsigned        refcnt; /*引用次数，应该也是该 hash 桶上的成员数*/
    unsigned        digest; /*计算机出来的 hash 值*/
    unsigned        hash; /*hash 桶数，digest % hcl_nhash*/
};
struct hcl_hd {
    unsigned        magic;
#define HCL_HEAD_MAGIC      0x0f327016
    VTAILQ_HEAD(, hcl_entry) head;
    MTX             mtx; /*线程锁，在访问 hash 队列时的互斥*/
};
static unsigned    hcl_nhash = 16383;
static struct hcl_hd *hcl_head;

```

HTTP object 在被创建并从后台服务器取到内容以后，是被挂到 hash 中各桶的 objhead 上去的，object 和 objhead 的结构体定义在 varnish-dist/bin/varnishd/cache.h:

```

struct object {

```

```

    unsigned    magic;
#define OBJECT_MAGIC    0x32851d42
    unsigned    refcnt; //被 hash 结构体引用次数
    unsigned    xid;
    struct objhead    *objhead; /*存储该 object 的 objhead*/
    struct storage    *objstore; /*存储的 st 队列*/
    struct objexp    *objexp;    /*存储该 object 超时结构体*/
    struct ws        ws_o[1];    /*存储空间指向 storage 的内存*/
    unsigned char    *vary;
    struct ban        *ban;    /*用于禁止访问某些 object 的结构体*/
    unsigned    pass;
    unsigned    response;
    unsigned    cacheable;    /*已经 cache? */
    unsigned    busy;    /*忙否? */
    unsigned    len;    /*所占内存长度*/
    double    age;    /*存在时间*/
    double    entered;    /*? */
    double    ttl;    /*生存时间*/
    double    grace;    /*优雅时间, ttl = ttl-grace*/
    double    prefetch;
    double    last_modified; /*最后获取过来的时间*/
    struct http    http[1];    /*新的 http 结构体*/
    VTAILQ_ENTRY(object)    list; /*入口地址*/
    VTAILQ_HEAD(, storage)    store; /*object 所对应的 storage 队列, 可能不止一个*/
    VTAILQ_HEAD(, esi_bit)    esibits;
    double    last_use;
    /* Prefetch */
    struct object    *parent;
    struct object    *child;
    int    hits;    /*命中次数*/
};

struct objhead {
    unsigned    magic;
#define OBJHEAD_MAGIC    0x1b96615d

```

```

void          *hashpriv;          /*指向某个 hash 入口*/
pthread_mutex_t  mtx;
VTAILQ_HEAD(object) objects;
char          *hash;              /*指向产生 hash 的字符串*/
unsigned      hashlen;           /*hash 长度, 等于 sp->lhashptr*/
VTAILQ_HEAD(, sess) waitinglist; /* There are one or more busy objects, wait for them */
};

```

object 对象被缓存到内存以后, 一般有个 TTL (生存期)。有个名叫 exp_timer 的线程会检查最老 object 的 TTL, 如果已经超时, 就删除之。那么它是如何来迅速知道哪个 object 是最老的呢? 已经删除最老的那个 object, 又如何把次老的 object 提取出来呢? Varnish 用到的主要是二叉堆的结构体 (见 3.1)。该结构体的定义在 Varnish-dist/bin/varnishd/cache_expire.c:

```

static struct binheap *exp_heap;
static MTX exp_mtx;
struct objexp {
    unsigned      magic;
#define OBJEXP_MAGIC      0x4d301302
    struct object *obj;
    double        timer_when;     /*超时时间点, 注意 Varnish 是用一个 deouble 的浮点
                                   来保存时间的 (秒数+微秒)*/
    const char    *timer_what;    /*对上述超时器的描述*/
    unsigned      timer_idx;      /*在二叉堆中的序号*/
    VTAILQ_ENTRY(objexp) list; /*访问入口*/
    int           on_lru;         /*是否处于 lru 队列中*/
    double        lru_stamp;
};
static VTAILQ_HEAD(objexp) lru = VTAILQ_HEAD_INITIALIZER(lru);

```

此外, 有时内存不足时, 就不得不将一些未超时的 object 删除, 删除的策略主要是通过一个 LRU[4]队列来实现。它是一个存储 objexp 的尾队列, 每次某个 object 被访问以后, 该 object 的超时结构体即被从上述的尾队列中移到末尾。所以, 每次需要腾空内存时, 该尾队列的头部 object 总是最久未被访问的对象。

3.5 session 及一些工作结构体

每次 Varnish 接受一个 HTTP 请求连接以后, 就创建了一个 session 对象, 它负责保存有关这个请求所有内容, 它是 Varnish 工作的核心结构体。此外 ws (workspace) 和 seemem 结构体是为 session 预分配内存的结构体。其结构体定义在 varnish-dist/bin/varnishd/cache.h:

```

struct sess {

```

```

    unsigned    magic;
#define SESS_MAGIC    0x2c2f9c5a
    int         fd; /*socket fd*/
    int         id; /*= fd*/
    unsigned    xid; /*状态机处理的序号*/
    int         restarts;
    int         esis;
    struct worker    *wrk; /*指向当前工作的线程*/
    socklen_t     sockaddrlen; /*sizeof(sm->sockaddr[0])*/
    socklen_t     mysockaddrlen; /*sizeof(sm->sockaddr[1])*/
    struct sockaddr    *sockaddr; /*(void*) (&sm->sockaddr[0])*/
    struct sockaddr    *mysockaddr; /*(void*) (&sm->sockaddr[1])*/
    struct listen_sock    *mylsock; /*指向本 session 的监听 socket*/
    /* formatted ascii client address */
    char         *addr;
    char         *port;
    struct srcaddr    *srcaddr;
    /* HTTP request */
    const char    *doclose; /*redo with http_GetHdrField()*/
    struct http    *http; /*&sm->http[0]*/
    struct http    *http0; /*&sm->http[1]*/
    struct ws      ws[1]; /*sessmem 内存后面的一段空间*/
    char         *ws_ses; /* WS above session data */
    char         *ws_req; /* WS above request data */
    struct http_conn    htc[1];
    /* Timestamps, all on TIM_real() timescale */
    double        t_open;
    double        t_req; /*请求进入时间*/
    double        t_resp; /*回复时间*/
    double        t_end;
    /* Acceptable grace period */
    double        grace;
    enum step      step; /*状态机字段*/
    unsigned      cur_method;

```

```

unsigned    handling; /*处理结果: VCL_RET_LOOKUP*/
unsigned char    sendbody;
unsigned char    wantbody;
int         err_code;
const char    *err_reason;
VTAILQ_ENTRY(sess)    list;
struct director    *director; /* sp->vcl->director[0]*/
struct vbe_conn    *vbe; /*sp->director->getfd(sp)*/
struct bereq    *bereq; /*指向针对后台服务器的请求*/
struct object    *obj;    /**/
struct objhead    *objhead;
struct VCL_conf    *vcl; /*处理的 vcl 配置*/
/* Various internal stuff */
struct sessmem    *mem; /*指向其分配其空间的 sessmem*/
struct workreq    workreq;
struct acct    acct;
/* pointers to hash string components */
unsigned    nhashptr; /* sp->vcl->nhashcount * 2*/
unsigned    ihashptr; /* 需要 hash 的字符串长度 ,制作 hash 时有用*/
unsigned    lhashptr; /*hash length, sp->lhashptr = 1; */
const char    **hashptr; /*WS_Alloc(sp->http->ws, sizeof(const char *) *
(sp->nhashptr + 1))*/
};

struct ws {
    unsigned    magic;
#define WS_MAGIC    0x35fac554
    const char    *id;    /* identity : “sess” , “obj” */
    char    *s;    /* (S)tart of buffer */
    char    *f;    /* (F)ree pointer */
    char    *r;    /* (R)eserved length */
    char    *e;    /* (E)nd of buffer */
    int    overflow; /* workspace overflowed */
};

```

/*使用两组 seemem 来分配 session，这样是为了减少加锁的时间长度*/

```
struct sessmem {
    unsigned    magic;
#define SESSMEM_MAGIC    0x555859c5
    struct sess    sess;    /*存储 session 的地方*/
    struct http    http[2];
    unsigned    workspace;    /*params->sess_workspace*/
    VTAILQ_ENTRY(sessmem)    list; /*指向空闲出来的 session*/
    struct sockaddr_storage    sockaddr[2];
};
```

处理 session 的都是一些 worker 线程，描述这些 worker 的结构体定义也在 varnish-dist/bin/varnishd/cache.h:

```
struct workreq {
    VTAILQ_ENTRY(workreq)    list;
    workfunc    *func; /* wrk_do_cnt_sess()*/
    void    *priv; /*指向某个 session*/
};

struct worker {
    unsigned    magic;
#define WORKER_MAGIC    0x6391adcf
    struct objhead    *objhead; /*指向一个 objhead*/
    struct object    *nobj; /*w->nobj = (void *)st->ptr*/
    double    lastused; /*最后的使用时间*/
    pthread_cond_t    cond; /*线程锁*/
    VTAILQ_ENTRY(worker)    list; /*入口*/
    struct workreq    *wrq; /*指向当前工作的 work 请求*/
    int    *wfd;
    unsigned    werr; /* valid after WRK_Flush() */
    struct iovec    iov[MAX_IOVS];
    int    niov;
    ssize_t    liov;
```

```

struct VCL_conf      *vcl;
struct srcaddr      *srcaddr;
struct acct         acct;
unsigned char       *wlb, *wlp, *wle;
unsigned            wlr;
};
struct wq {
    unsigned         magic;
#define WQ_MAGIC     0x606658fa
    MTX              mtx;
    struct workerhead idle; /*空闲的线程队列*/
    VTAILQ_HEAD(, workreq) overflow; /*无法处理 workreq 的数目*/
    unsigned         nthr;
    unsigned         nqueue; /*排队数目*/
    unsigned         lqueue;
    uintmax_t        ndrop;
    uintmax_t        noverflow; /*无法处理 workreq 的数目*/
};

```

Varnish 使用的是每个 HTTP 请求对应一个线程的方式，所以采取的是线程队列的方式。

参考文献:

- 1、<http://varnish.projects.linpro.no/wiki/ArchitectNotes>
该文献较为重要，作者阐明其软件的设计思路，建议一读，相应的中文翻译：
<http://yaoweibin2008.blog.163.com/blog/static/11031392008101132330325/>
- 2、http://en.wikipedia.org/wiki/File_system_fragmentation
- 3、<http://varnish.projects.linpro.no/wiki/VarnishInternals>
- 4、http://en.wikipedia.org/wiki/Cache_algorithms
- 5、<http://freebsdchina.org/forum/viewtopic.php?t=37913>
- 6、http://en.wikipedia.org/wiki/Binary_heap

附录:

1、FreeBSD 中 tail queue 的定义:

```
/*
 * Tail queue declarations.
 */
#define VTAILQ_HEAD(name, type) \
struct name { \
    struct type *vtqh_first; /* first element */ \
    struct type **vtqh_last; /* addr of last next element */ \
}

#define VTAILQ_HEAD_INITIALIZER(head) \
    { NULL, &(head).vtqh_first }

#define VTAILQ_ENTRY(type) \
struct { \
    struct type *vtqe_next; /* next element */ \
    struct type **vtqe_prev; /* address of previous next element */ \
}

/*
 * Tail queue functions.
 */
#define VTAILQ_CONCAT(head1, head2, field) do { \
    if (!VTAILQ_EMPTY(head2)) { \
        *(head1)->vtqh_last = (head2)->vtqh_first; \
        (head2)->vtqh_first->field.vtqe_prev = (head1)->vtqh_last; \
    } \
}
```

```

        (head1)->vtqh_last = (head2)->vtqh_last;    \
        VTAILQ_INIT((head2));                      \
    }                                               \
} while (0)

#define VTAILQ_EMPTY(head) ((head)->vtqh_first == NULL)

#define VTAILQ_FIRST(head) ((head)->vtqh_first)

#define VTAILQ_FOREACH(var, head, field)           \
    for ((var) = VTAILQ_FIRST((head));           \
         (var);                                   \
         (var) = VTAILQ_NEXT((var), field))

#define VTAILQ_FOREACH_SAFE(var, head, field, tvar) \
    for ((var) = VTAILQ_FIRST((head));           \
         (var) && ((tvar) = VTAILQ_NEXT((var), field), 1); \
         (var) = (tvar))

#define VTAILQ_FOREACH_REVERSE(var, head, headname, field) \
    for ((var) = VTAILQ_LAST((head), headname);           \
         (var);                                           \
         (var) = VTAILQ_PREV((var), headname, field))

#define VTAILQ_FOREACH_REVERSE_SAFE(var, head, headname, field, tvar) \
    for ((var) = VTAILQ_LAST((head), headname);           \
         (var) && ((tvar) = VTAILQ_PREV((var), headname, field), 1); \
         (var) = (tvar))

#define VTAILQ_INIT(head) do {                     \
    VTAILQ_FIRST((head)) = NULL;                   \
    (head)->vtqh_last = &VTAILQ_FIRST((head));    \
} while (0)

#define VTAILQ_INSERT_AFTER(head, listelm, elm, field) do { \
    if ((VTAILQ_NEXT((elm), field) = VTAILQ_NEXT((listelm), field)) != NULL) \
        VTAILQ_NEXT((elm), field)->field.vtqe_prev = \

```

```

        &VTAILQ_NEXT((elm), field); \
    else { \
        (head)->vtqh_last = &VTAILQ_NEXT((elm), field); \
    } \
    VTAILQ_NEXT((listelm), field) = (elm); \
    (elm)->field.vtqe_prev = &VTAILQ_NEXT((listelm), field);\
} while (0)

#define VTAILQ_INSERT_BEFORE(listelm, elm, field) do { \
    (elm)->field.vtqe_prev = (listelm)->field.vtqe_prev; \
    VTAILQ_NEXT((elm), field) = (listelm); \
    *(listelm)->field.vtqe_prev = (elm); \
    (listelm)->field.vtqe_prev = &VTAILQ_NEXT((elm), field);\
} while (0)

#define VTAILQ_INSERT_HEAD(head, elm, field) do { \
    if ((VTAILQ_NEXT((elm), field) = VTAILQ_FIRST((head))) != NULL) \
        VTAILQ_FIRST((head))->field.vtqe_prev = \
            &VTAILQ_NEXT((elm), field); \
    else \
        (head)->vtqh_last = &VTAILQ_NEXT((elm), field); \
    VTAILQ_FIRST((head)) = (elm); \
    (elm)->field.vtqe_prev = &VTAILQ_FIRST((head)); \
} while (0)

#define VTAILQ_INSERT_TAIL(head, elm, field) do { \
    VTAILQ_NEXT((elm), field) = NULL; \
    (elm)->field.vtqe_prev = (head)->vtqh_last; \
    *(head)->vtqh_last = (elm); \
    (head)->vtqh_last = &VTAILQ_NEXT((elm), field); \
} while (0)

#define VTAILQ_LAST(head, headname) \
    (((struct headname *)((head)->vtqh_last))->vtqh_last)

#define VTAILQ_NEXT(elm, field) ((elm)->field.vtqe_next)

```

```

#define VTAILQ_PREV(elm, headname, field)          \
    (((struct headname *)((elm)->field.vtqe_prev))->vtqh_last))

#define VTAILQ_REMOVE(head, elm, field) do {      \
    if ((VTAILQ_NEXT((elm), field)) != NULL)     \
        VTAILQ_NEXT((elm), field)->field.vtqe_prev = \
            (elm)->field.vtqe_prev;              \
    else {                                         \
        (head)->vtqh_last = (elm)->field.vtqe_prev; \
    }                                             \
    *(elm)->field.vtqe_prev = VTAILQ_NEXT((elm), field); \
} while (0)

```

2、Varnish 源代码中一些特定的缩写字母含义：

BH:Binary Heap

CLI:Command-Line Interface,part of the published Varnish-API,see "cli.h"

CNT:CeNter

EVB:Event Variable Base

EXP:EXpire

HCL:Hash CLassic

HTC:HTtp Connection

MCF:Main ConFigure

mgt:managment

PAN:PANic

PF:PoolFD, Poll File Description

SES:SESSion

SMF:Storage Mmapped File

SMS:Storage Mutex Synth

STV:STeVedore

TMO:TiMe Out

VBE:Varnish BackEnd

VBM:Varnish Bit Map

VBP:Varnish Backend Polling

VCA:Varnish Cache Acceptor

VCC:Varnish Configure Compile?

VCL:Varnish Configure Language

VCT:Varnish Character Type

VDI:Varnish DIrector

VEV:Varnish Event Variable

VLU:Varnish Line Up

VPF:Varnish PID File

VS:Varnish Storage Buffer

VSS:Varnish adreSS?

VSL:Varnish Share-memory Log

VTAILQ:Varnish Tail Queue,

WQ:Work Queue

WRK:WoRKer

WSL: Worker Share-emory Log