测试链接

by 一本正经说废话

- HAProxy代码分析
  - 1 HAProxy概述
  - 2 HAProxy的进程和资源
  - 3 HAProxy的主要数据结构
  - 4 HAProxy的处理流程
    - 4.1 HAProxy accept连接过程
    - 4.2 HAProxy的数据处理流程
    - 4.3 关于splice
  - 5 HAProxy一些性能参数
  - 6. 一些代码细节Q&A

HAProxy代码分析

1 HAProxy概述

这里是概述

2 HAProxy的进程和资源

HAProxy支持多进程同时运行，但在每一个进程内部，是采用单线程的，事件驱动的，非阻塞模型。

支持多进程时候，需要通过deamon的方式启动。

```
global
    cbproc      4
    maxconn     4000
    cpu-map     all
or
    cpu-map     3  0  2-3
```

上面例子配置文件中global段的cbproc指定需要启动多少个haproxy进程，启动进程的个数一般和cpu core个数有关，按1：1配置即可，另外，haproxy进程组可以绑定CPU core。

多进程的好处，自然是在多核的情况下提升性能。但也有对应的问题：

- 对于backend server，会有多份健康检查
- maxconn这样的设置都是per-process的，注意配置，不要过载
- stick-table，就是记录会话保持的session表，也是per-process的
- 关于peer同步和CLI操作一次只能针对一个进程

综上HAProxy对某些应用给出的建议：

对于ssl和compression这样的CPU大户，提出的分层的概念，CPU大户在第一层，采用多进程结构，处理完毕后交给第二层，第二层采用单进程结构，层之间采用unix socket串联。比如compression卸载，在前端（client端交互那层）可以如下配置：

```
global
 nbproc          1
 cpu-map         1
frontend  www
    bind :80 maxconn 123
    mode http
    maxconn              1234
    default_backend        app
backend app
    mode      http
    balance     roundrobin
    server  app3 unix@/var/run/comp_sock.sock
```

/var/run/comp_sock.sock是后端的unix socket。

第二层做如下配置：

```
global
 chroot      /var/lib/haproxy
    pidfile     /var/run/haproxy.pid
    maxconn     4000
    nbproc      3
    daemon
frontend  unix_sock_comp
    bind /var/run/comp_sock.sock user root mode 600
    mode http
    maxconn                 1234
    compression offload
    compression algo gzip
 compression type text/html text/plain
    default_backend            app
backend app
    mode        http
    balance     roundrobin
    server  app1 192.168.0.3:8080
```

这里和前面的结合起来，就是用unix socket做监听socket，然后和后面的backend server
通讯，这层实现compression的offload，把压缩后的内容送给前端haproxy

Q1：如果只有一个listener，那么多进程怎么分享这listener呢？

A：通过如下几个步骤：

- deamon进程（第一个haproxy进程）创建监听socket，poll(epoll, kqueue......)结构
- fork多个haproxy进程，daemon进程退出
- 每个haproxy只accept到指定个数就暂停，轮给其他进程accept
    1. 连接速度上限（maxconnrate）
    2. 并发连接上限 （maxconn）
    3. 一次最大accept数（tune.maxaccept）
    4. 其他

  note：上面的行为是基于accept和epoll对惊群的解决

Q2：HAProxy支持热重启么？

当HAProxy的配置文件变更时，可以重启HAProxy，启动新进程之前会发信号给老的进程告之停止listen，新启动的HAProxy进程使用新的配置，老HAProxy上的已有连接不受影响，当所有连接关闭后，老HAProxy进程自动退出。

/usr/sbin/haproxy -f /etc/haproxy.cfg -sf pid1 pid2 pid3 ....

等同于如下操作，

killall -SIGUSR1 haproxy

/usr/sbin/haproxy –f /etc/haproxy.cfg

如果想干脆的直接退出

/usr/sbin/haproxy –f /etc/haproxy.cfg –st pid1 pid2 pid3 ....

等同于

killall –SIGTERM(15) haproxy

/usr/sbin/haproxy –f /etc/haproxy.cfg

当前haproxy对如下信号感知：

- SIGQUIT　　内存pool回收
- SIGHUP　　dump内存信息
- SIGUSER1　相当于sf参数，gracefully退出，就是停止监听，老的连接还能work
- SIGTTOUT　旧的进程暂停监听，保留listener对象，但是shutdown fd
- SIGTTIN　　旧的进程恢复监听，重新调用listen

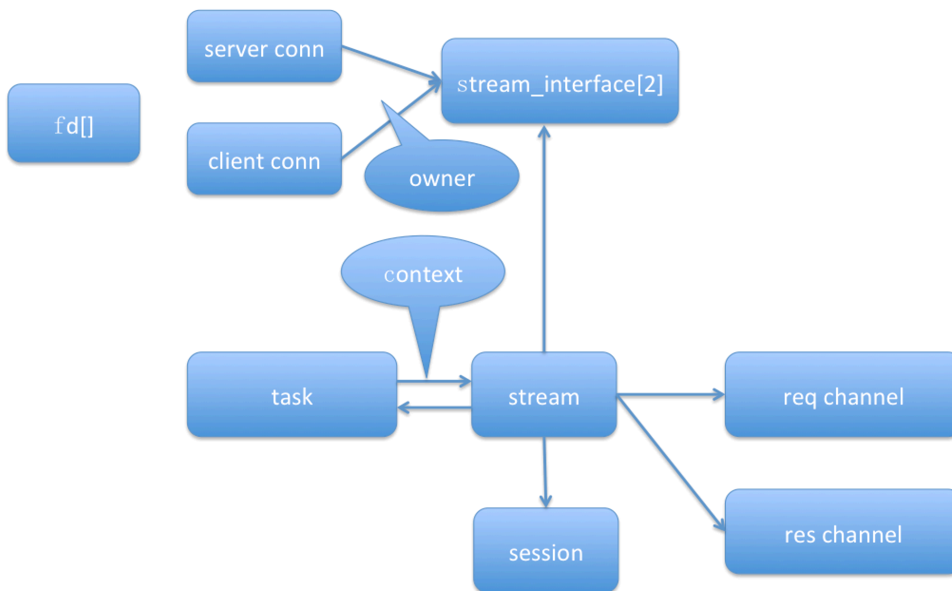
后三的个协作是这样的，在新的进程里先让旧的进程暂停，然后自己各种启动，如果启动失败就恢复旧的，最后成功后停止旧的。

Q3：怎么知道当前HAProxy的CPU使用情况？

除了TOP/ps外，HAProxy提供了一种自己的时间统计叫idle或者Idle_pct。他是根据poll执行的时间 v.s.
总时间，如果比较闲，大部分时间都在poll等待，所以大约idle值为100，如果一直忙于处理，则idle值为0。那么这个和top/ps有什么区别呢，这个是per–process计算，TOP是per–CPU计算。我们可以依据这个做监控，如果idle值比较低，有可能是不合理的情况，可以从以下几方面去着手解决：

1. 一种可能是系统抢走比较多的资源，比如softirq占用从50%到80%，导致haproxy进程可用CPU资源太少，从而导致idle少，这样需要分析系统层面，比如：
   a. 是不是防火墙占用太多资源
   b. 是不是interrupt/softirq是不是绑定在固定cpu上了，碰巧和haproxy在一个core上。
2. 设置nbproc参数，开启多进程处理模式

3 HAProxy的主要数据结构



- proxy：一个proxy可以认为是一个客户，通过一个proxy的流量有着相同的转发规则。haproxy进程可以容纳多个proxy，对应于配置中的listener或frontend
- listener：一个监听fd的封装，一个proxy可以有多个listener，对应于配置中的bind，每个listener可以有自己的最大连接数。

```
frontend  www
    bind :80 maxconn 123
    bind :81
    bind :8001-8003
    mode http
    default_backend          app
```

上面的配置文件就有会80，81，8001，8002，8003这5个listener。

- connection：一个具体连接fd的封装，可以通过fdtab[fd].owner找到。
- task：haproxy的一个执行调度单位，想执行点东西一般先激活一个task去执行，比如当socket有事件的时候，把task加入run queue，然后执行task。
  task有两类，一类是等待执行的task，比如3秒后做health check，挂在wait queue上，3秒后执行，另一类是马上需要执行的task，挂在run queue上，按『顺序』执行，决定task执行顺序的是task的nice值，类似于UniX进程调度的nice值。
- stream：stream用于表示一个转发流，可以认为是一个http的transaction，包含前端连接和后端两部分，stream和task是1：1的关系。
- stream_interface：stream的成员，一个stream有2个stream_interface，分别表示前端和后端，可以认为是connection的一种中转形式，因为haproxy都是异步操作，如果想发数据，不能直接发，需要调用stream_interface的发送，实际上并没有发送，只是注册了写事件，等poll触发后才可以通过connection发送。
- session：以前版本里的session相当于现在的stream，现在的session已经简化了，只记录了该stream的一些基本属性信息，比如属于哪个frontend，listener，accept时间等，用于记录统计。
- channel：对数据通道的封装，比如超时，状态，标志，统计等，都在这里设置，一个stream里有2个channel，分别对应请求和响应，channel并不负责发送和接收，只是维护数据及其状态，另外一个channel的作用就是封装http和tcp的channel为一个统一的接口，因为tcp和http不一样，需要统一接口。
- buffer：channel操作的对象，真正存储数据的地方。

4 HAProxy的处理流程

4.1 HAProxy accept连接过程

当一个listener fd有read event的时候，会调用void listener_accept(int fd)函数，在其内部过程如下：

```
/* This function is called on a read event from a listening socket, corresponding
 * to an accept. It tries to accept as many connections as possible, and for each
 * calls the listener's accept handler (generally the frontend's accept handler).
 */
void listener_accept(int fd)
{
 /*0.根据global.tune.maxaccept设置一次accept多少个，但还要受限后面的cps,sps等*/
 /*1.检查listener并发连接数*/
 if(unlikely(l->nbconn >= l->maxconn))
 {
  /*暂停对该fd poll*/
  listener_full(l);
  return
 }
 /*2.检查global的session rate*/
 if(!(l->options & LI_O_UNLIMITED) && global.sps_lim)
 {
  if(/*到了上限*/)
  {
   expire=###;
   goto wait_expire;
  }
  max_accept = 可再接收的值;
 }
 /*3.检查global的conn rate*/
 if(!(l->options & LI_O_UNLIMITED) && global.cps_lim)
 {
  if(/*到了上限*/)
  {
   expire=###;
   goto wait_expire;
  }
  max_accept = 可再接收的值;
 }
 /*4.检查该front end(proxy)上的session rate*/
 /*......*/
 /*5.检查全局并发连接是否超过限制*/
  if (unlikely(actconn >= global.maxconn) && !(l->options & LI_O_UNLIMITED)) {
   limit_listener(l, &global_listener_queue);
   task_schedule(global_listener_queue_task, tick_add(now_ms, 1000)); /* try again in 1 second */
   return;
 }
 /*6.检查该front end(proxy)是否超过限制*/
 if (unlikely(p && p->feconn >= p->maxconn)) {
  limit_listener(l, &p->listener_queue);
  return;
 }
 /*开始接收过程....*/
wait_expire:
 limit_listener(l, &global_listener_queue);
 task_schedule(global_listener_queue_task, tick_first(expire, global_listener_queue_task->expire));
 return;
}
```

根据限定范围，限制分三类，一类是全局性质的限制，如global.maxconn，一类是per-proxy的限制，比如proxy->maxconn，还有一类是per-listener的，比如l->maxconn。不同层次都有自己的限制，代码看起来很乱，因为hit到不同的限制处理结果不一样。

从分支上看，有这么几个方式：

1. 到达listener并发处理能力上限，不能继续了，如步骤1，比如只能支持并发1000，现在第1001个连接来了，不再接收。处理方式为暂停对listener 的fd进行poll。
2. 到达全局速度处理能力上限，不能再继续了，如步骤2，3。conn rate或者session rate到上限了，比如cps是100个/秒，用了0.8秒接收到了100个了，那么就将listener对应的任务执行时间设为等0.2秒后，暂停对listener进程poll，并加入global listener wait queue。
3. per-proxy的session rate到上限，如步骤4，处理方式类似2，但这里加入的是per-proxy的listener wait queue
4. 到达全局并发连接数上限，如步骤5，暂停对fd进行poll，加入global listener wait queue，如果其他极致没有在1秒内抢先唤醒，则靠这个1秒后再来。（代码里没有说为什么1s）
5. 到达proxy并发连接数上限，如步骤6，暂停对fd进行poll，不加入任何listener queue。当时没看懂为什么，还写邮件给haproxy mail list。答复如下：

   > No because if we're limited by the frontend itself, after we disable the
   > listener, we will automatically be woken up once a connection is released
   > there. It's when the global maxconn is reached that we want to reschedule
   > because there are some situations where we cannot reliably detect if
   > certain connections impacting global.maxconn have been released (eg:
   > outgoing peers connections and Lua cosockets count here).

对不同情况处理方式不一样可以归结到一下两类：

- 以什么样的方式暂停对fd的poll操作，一类是limit_listener,一类是listener_full。
  区别是什么呢？full暂停fd的poll操作后是通过有资源来重新激活的，比如一个连接释放了，不如加入人和队列，而limit，则干了2件事，一时个暂停poll，另一个是加入到一个listener wait queue，要么是全局的，要么是per-proxy的，可以通过2种方式救活，一个是有资源释放了，一个是时间到了，本wait queue的全部激活。
- 是否需要把task加入listener queue等待

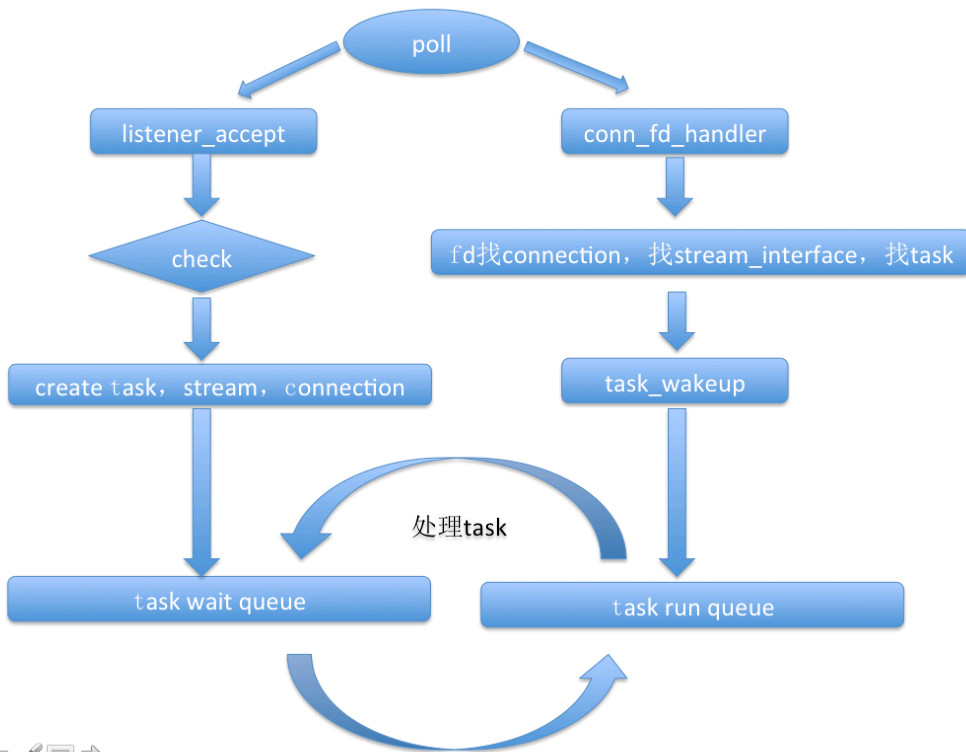
  速度方面的必须加入队列，因为本秒的配额已经用完，必须等到下一秒，另一个是全局连接数到上限了，需要等1秒，但可以提前唤醒。

最后在看一下当 有连接释放，资源 可用的时候怎么唤醒listener的：

```
if (sess->listener) {
  if (!(sess->listener->options & LI_O_UNLIMITED))
   actconn--;
  sess->listener->nbconn--;
  if (sess->listener->state == LI_FULL)
   resume_listener(sess->listener);  <------唤醒通过lisenter_full()暂停的
  /* Dequeues all of the listeners waiting for a resource */
  if (!LIST_ISEMPTY(&global_listener_queue))
   dequeue_all_listeners(&global_listener_queue);   <-----因为全局连接数到上限而暂停的
  if (!LIST_ISEMPTY(&sess->fe->listener_queue) &&
      (!sess->fe->fe_sps_lim || freq_ctr_remain(&sess->fe->fe_sess_per_sec, sess->fe->fe_sps_lim, 0) > 0))
   dequeue_all_listeners(&sess->fe->listener_queue);
<-----因为proxy的连接数到上限而暂停的，这里恶心的，因为速度受限的不能唤醒。
 }
```

4.2 HAProxy的数据处理流程

先看图：



- 对于listener fd和数据fd，有着不同的回调函数，当新建连接的时候，调用listener_accept函数，然后检查能否接收，具体检查流程参考5.1
- 检查通过后创建task，stream，session，和connection对象，然后task放入wait queue等超时（以防只建连不发包的情况）
- 当client端来request后，调用回调conn_fd_handler，然后根据指来指去的指针，找到task，把task放入run queue
- 处理到时间的task或者run queue上的task
- 在住循环run_poll_loop里，处理queue上的task，统一回调process_stream函数
- 回调函数里分几部分

- 如果client连接okay，有请求，解析请求
- 如果server连接okay，有响应，解析响应
- splice的透传处理（需开通splice）
- 没有backend connection时，LB选择server与建连
- 更新channel，触发双方连接的poll读写

但这几部分是选择执行的，比如一个transaction的流程可能如下：

- client请求到来会依次执行1，4。
- LB选server以及和server建连，由于是非阻塞的，不知道是否成功。
- server建好连接回调的时候走5，注册写事件
- poll回调server fd，写请求到server
- server请求回来，走2，5，注册client写事件
- poll回调client fd，写响应回去。

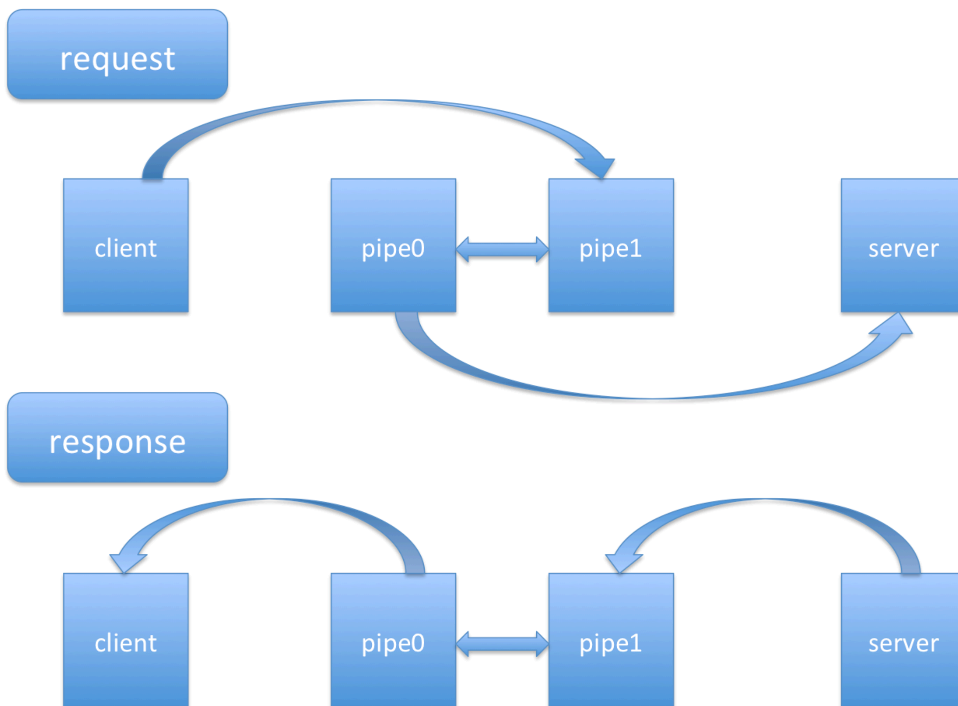
4.3 关于splice

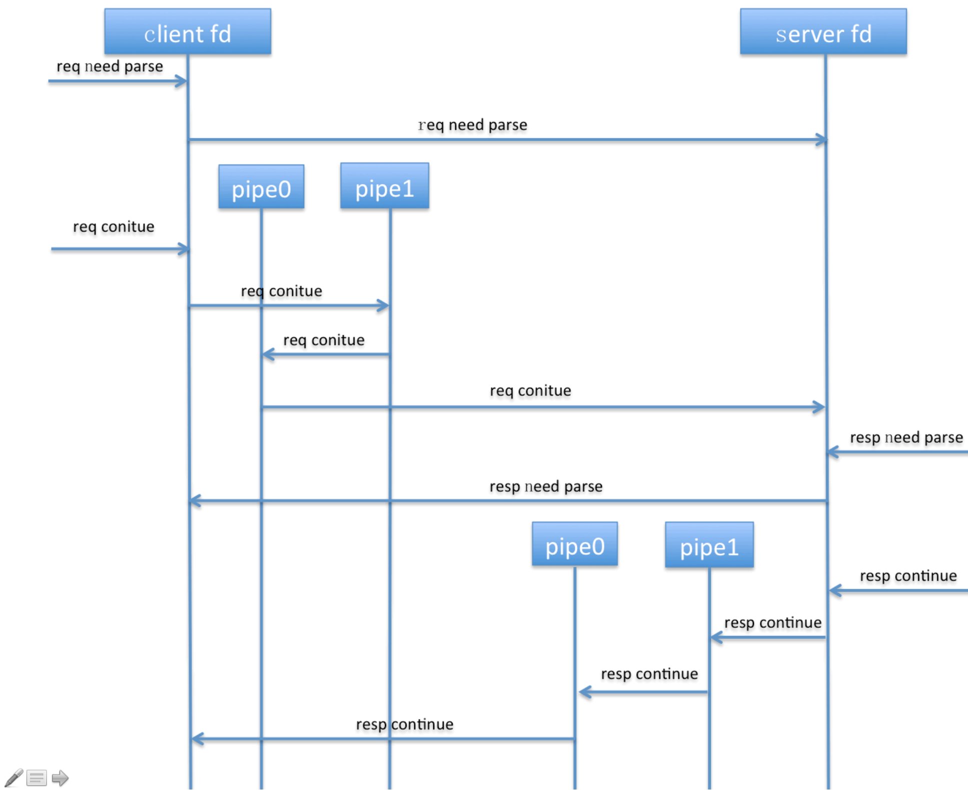
haproxy提供了一种通过减少用户态和内核态拷贝而提升性能的方案。就是splice，需要2.6.28以上的内核版本才能支持。默认情况并不开通需要开启下面3个中的选项，可多选：

- option splice-request
- option splice-response
- option splice-auto

这些选项对于HTTP和TCP都是生效的，splice-request只把client到backend server之间的数据透传，不需要拷贝到用户态，同理splice-response。splice-auto我想不到它使用的场合，他本意是数据来的太快了，每当读的时候就导致读buffer满，有点处理不过来的意思就会触发splice。

splice和我以前理解的不一样，以为把2个fd splice到一起就不用管了，实际上还是一样复杂，只是没有是数据copy而已,实际上关于读多少，写多少仍然需要自己控制，只是看不到书局而已了。





Q：什么情况splice，什么情况不能splice?

TCP模式下，只要打开开关了，就可以splice。HTTP，一般是传输boby的时候才可以splice，因为header这些是需要parse和改写的，所以一定需要header解析完成后才能splice，但也不是所有的body都呢能或者需要splice，比如读header的时候，一不小心把body都读完了（响应body很小），就不需要splice。

Q：splice有什么优点和缺点?

优点：就是减少数据拷贝，提升性能。

缺点：splice的问题是占用比较多的fd，一个http transaction，除了前后端2个连接外，还需要1组pipe，所以需要4个fd。可以通过maxpipe设置最大pipe数，默认情况是1/4的maxconn。

5 HAProxy一些性能参数

- 连接相关
  - tune.maxaccept <number>
    设置切换到其他工作的时候，最多accept多少个client连接。对于单进程来说，大值会有一定性能提升，但对于多进程来说，为了更好的进程间任务均衡，设置小值更何时。官方建议单进程的时候是64，多进程的时候是64/2/绑定的CPU个数，比如用了2个core，就是64/2/2=16个。
  - maxconn <number>
    global或者per-proxy（per-listener）,设置能服务的并发连接数的上限。默认值是2000。当到达上限的时候，停止poll检查这个listener fd。

```
if (unlikely(l->nbconn >= l->maxconn)) {
  listener_full(l);
  return;
 }
```

  - maxconnrate <number>
    每秒最多accept多少连接，默认无限制，到达上限后，会进入wait queue等下一个周期开始才接收新连接。
  - tune.maxpollevents <number>
    一次epoll_wait最多能获取多少个事件来处理，默认值是200，根据测试结果，少于200是拿吞吐换时延，多于200是拿时延换吞吐。也就是说设置的小一点，可以减少延迟，但是吞吐会下降，多余200会提升吞吐，但时延会提升。
- 内存相关
  - tune.buffers.limit <number>

- tune.buffers.reserve <number>
- tune.bufsize <number>

上面3个内存相关的设置，bufsize是处理数据时候的缓冲区的大小，默认是16k，强烈不建议修改。buffers.reserve是当内存短缺时候，至少有这么多个buffer才处理该请求，否则加入等待队列，这是为了避免处理一半就buffer不够的情况，比如请求来了，处理完了，等收响应的时候，没有内存了，会导致transaction不完整。buffers.limit就是per-process的buffer上限，同时也限定了该实例的内存大小，因为主要是buffer来用内存，haproxy的建议是设置为global.maxconn的1/10。从测试上看也不是越多越好，较少会加大buffer复用，也会提升L2/L3 cache的命中率。

- CPU相关
    - cpu-map <"all"|"odd"|"even"|process_num> <cpu-set>...
- 压缩相关
    - maxcomprate <number>
    - maxcompcpuusage <number>

  这2个功能差不多，就是压缩的每秒吞吐到达一定程度或者CPU使用率到一定程度的化就降低压缩级别，如果还不行就自动停止压缩。默认情况没有开启压缩。
- SSL相关
    - maxsslconn
    - maxsslrate
    - tune.ssl.maxrecord <number>
    - tune.ssl.cachesize
    - tune.ssl.lifetime
    - tune.ssl.force-private-cache
    - tune.ssl.maxrecord
    - tune.ssl.default-dh-param
    - tune.ssl.ssl-ctx-cache-size

6. 一些代码细节Q&A

Q：socket的回调函数是什么？

监听socket：
- fdtab[fd].iocb=listener->proto->accept; 即listener_accept

通讯socket：
- fdtab[fd].iocb=conn_fd_handler;

Q：连接上的事件怎么转化为task执行？

A：每一个socket都有一个callback函数，当有读事件来的时候，触发callback函数，

Q：Task的nice值是怎么回事？

任务可以设置自己的nice值，值越小（可以负值）优先级越高，nice值可以改变node在ebtree的位置，然后先被便利到。

Q：如果开启了4个进程，设置全局的max conn是100，每个进程是25么？

不是，每个进程仍然是100，所以要考虑好per-process资源的分配。

Q：在restart过程中，老进程组stop listen，新的开始listen，中间的时间差如果来连接怎么办？

是有损的，大约1/10000，在HA官方测试中，30000qps在切换的时候会失败3个。失败的原因有如下2方面：

1. 在因为其他原因绑定失败的的时候，对老进程监听socket进行了暂停，继续的操作，在这2个操作的时间差里，会失败。在BSD系统里，和3.9以上的linux内核支持多个进程绑定相同IP和端口，所以不是问题，但是我们当前是2.6的，没有办法避免
2. 在旧的进程租暂停监听后，带来的问题。有2种方式，1、SYN过来的时候就暂停监了，有时候防火墙会发现没有监听会drop syn包，然后client重传，这样就没有问题了，但我们的系统没有防火墙，会直接reset。2、三次握手最后一个ACK回来之前停止监听，这个铁定没戏。

Q：listener，proxy，global并发连接数的隐含关联

listener有配置就用自己的，没有就用proxy的，proxy有配置就用自己的，没有用default里的，default里没有就用默认值2000。global配置就用自己的否则默认值2000。proxy没有cps，只有session rate，这个默认用default字段配置的。cps只有global一份。

持续丰富中。。。。。