打造支撑海量用户的

高性能 server 系列之一

高性能 server 的内存管理

By　@codebox-腾讯

http://weibo.com/codebox

引

以此系列文章纪念因 QQ 空间、QQ 农场等互联网业务爆炸式增长，而日夜不眠地为公共组件做架构调整、性能优化的激情岁月。

序

以 QQ 农场、好友买卖为代表的 SNS 网页游戏的突然兴起，全国男女老少一起日夜不眠的在电脑面前奋战，鼠标、键盘还有各路外挂汇成的请求像洪水一般涌来。当流量把机房核心交换机冲得七零八落，当我们把市场上的服务器全部买光都不够时，我终于懂了，这就是 TMD 的所谓的海量。

在这个过程中，从 web 层到逻辑层，再到数据层，每一个组件都经过了一次洗礼，从小男孩成长为真正的男人。

现在我已经投身于云平台的建设了，这个支撑海量用户的高性能 server 系列文章就想为这段激情燃烧的岁月做个迟到的总结。目前打算从以下几个方面展开来：

1) 高性能 server 的内存管理。

2) 大并发下的 server 模型选择。

3) 高效定时器的实践。

4) 异步框架的设计与威力。

5) socket 及 OS 优化。

6) 业务应用的架构调整与思考。

这其中大部分是泥腿子办法，土法炼钢思路，不过却的的确确解决了问题，经受住了考验。

在这儿把这些东东放出来，我按我的想法写，您按您的想法看，然后，然后就没有了。。。

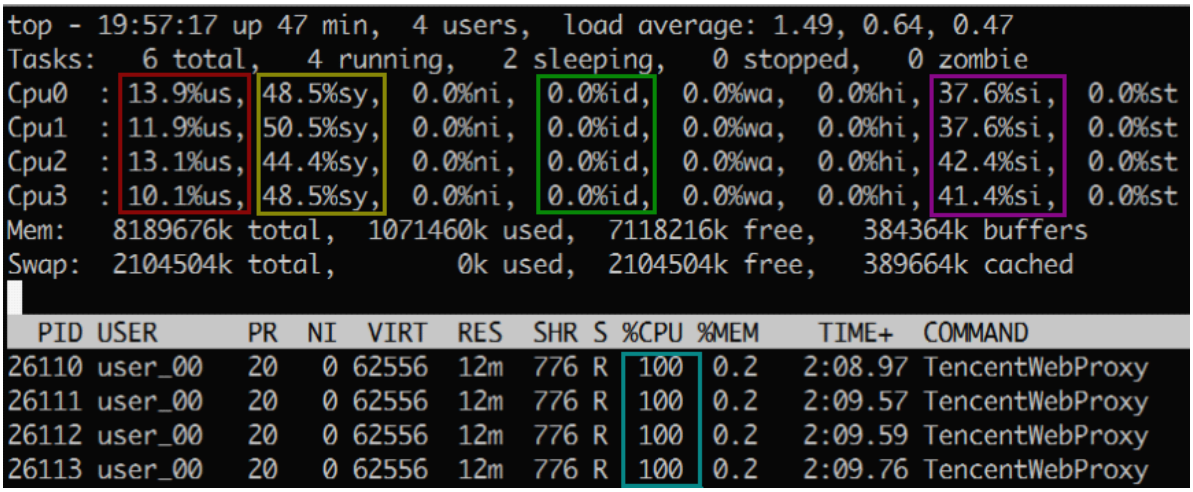
如何衡量好与坏

首先，要做好一件事情，就要知道什么是好与坏。那如何评价一个 server 写得怎么样？我的看法是，**能最有效地榨干系统资源**。

两个关键词，**有效**和**榨干**。

有效表明全在干正事儿，不做无用功；榨干表明充分利用，不浪费。

拿我自己实现的一个 web server 来举例(嘿嘿，我比较得意的一个作品 TencentWebProxy)，在压力测试过程中，表现如下：



i. 充分地利用了 CPU，机器共四个核，TencentWebProxy 相应地启动了四个工作线程，每个线程都完全把系统 CPU 吃尽。每个 CPU 使用都非常均匀，不管是系统态，用户态，还是软中断。当有更多的 CPU 核心时，也有通过启动相匹配的线程数，很好的利用多核。**此为榨干**。

ii. 按 CPU 各个维度的比例来说，系统态和软中断占了近 90%，说明系统大部分资源在处理网络服务，只有 10%多一点的用在用户态，说明对 HTTP 协议解析等操作的资源消耗控制地比较理想，全在干正事儿。**此为有效**。

下面就进入这篇的主题，如果构建高性能 server 的第一项，内存管理。

内存管理

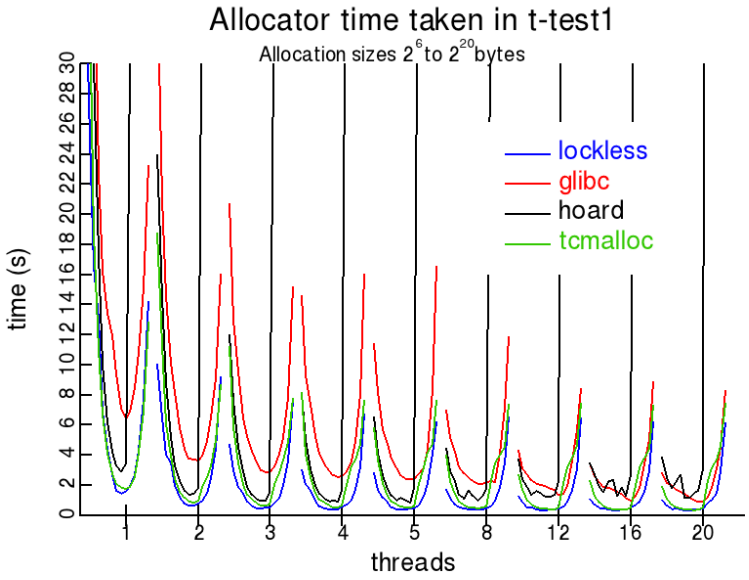
最省力的方法--PRELOAD 高效内存分配库

tcmalloc

业界最有名的内存分配库，当数 google 的 tcmalloc。Tcmalloc 在管理小内存块时非常有效，而且能够避免在大内存分配时的 mmap()系统调用。它在多线程中的表现也不错能很好的减少锁碰撞(glibc 致命的问题)。Tcmalloc 现在基本上成了 mysql DBA 的标配了。

lockless

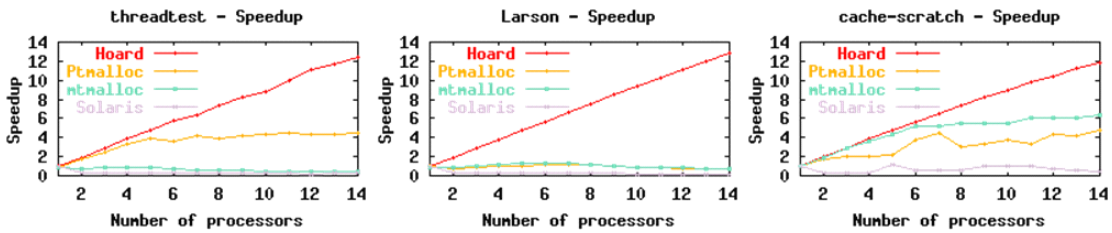
另一个比较常用的内存分配器是 lockless，着眼点是使用远锁技术来提升多线程内存分配性能。官方测试数据显示，在多线程环境下表现比 tcmalloc 还好。当然，官方数据也只能当参考，信一半就差不多了，呵呵。



（*The Lockless memory allocator can take about half the time as other allocators to complete the benchmarks. The Lockless memory allocator does not suffer from slowdowns at larger allocation sizes.*）

hoard

最后一个是 fory 最近向我推荐的 hoard 库 ,他在 UMass 的老师写的 ,hoard 最主要的着眼点是在 SMP 系统上，当核心越多的时候它的优势相当明显。用它自已经的话来说就是“*dramatically improve application performance, especially for multithreaded programs running on multiprocessors and multicore CPUs*”



我还没用过这个 hoard 库，后面有时间试试。

我不爽 glibc 的内存分配算法主要是两点：一是大内存的 mmap，其实这也不是很大的问题，凡是逼得 libc 调用 mmap，一般都是用法太不讲究了；其二，这才是最大的问题，多线程分配内存时的锁碰撞，会造成 server 性能直线下降，直到不可忍受的程度！

用 preload 高效内存库的好处就是立杆见影，不用改代码，不用重新编译。把 LD_PRELOAD 环境变量导出来就好了。如果线上系统出现内存管理的问题，我一般都会把 tcmalloc 加载上去看看效果。

要注意的是，所有的动态内存分配算法，肯定是在众多方面做了 trade-off，用之前要理解清楚了，方能起到预期效果。

通用内存池

内存池是常见的一种自己管理内存的方式，比如说 nginx 内部的所有内存分配都是基于 ngx_palloc 和 ngx_pfree 及其家族函数进行管理的。

Nginx 分配内存过程一般如下:

1. 如果分配大内存，则直接调 malloc 分配，用完后挂到 large 链表中，下次分配大内存可从 large 链表中分配。
2. 如果分配的是小内存，则从 current 所指的链表中寻找大小合适的内存，找到就返回，找不到就新分配。

腾讯使用的三网代理 qhttpd 也使用了内存池技术，不同的是 qhttpd 的内存是预分配的，启动的时候就分配了一大块内存，运行过程中不再分配。

内存池的缓存级别还是比较低，对应的是内存 buffer 级别抽象。但由于高性能内存分配库的不断改时，这些自己实现的内存池，不一定就更有效，并且是各种 bug 的温床。我对这种方式持比较保留的意见。

通用对象池

对象池最有名的当数 linux 内核中的 slab 高速缓冲区，为内核中各种常见数据结构做缓存，如 skb 等。

在用户空间里一般这样实现：把空闲对象挂到一个链表上，分配的时候就可以从链表上取下来，释放的时候再挂回去。这都是 O(1)的操作。这样做的好处是，每种对象基本上都是固定大小的，所以可以直接从空闲链表上拿下来用。

不过这儿还有一个问题，就是多线程操作同一个链表的锁问题，如果不做特别设计，会出现类似 malloc 在多线程环境下的锁碰撞问题。

这个问题的解决思路一般有两个：

i. 对象池局部化。局部化后，每个线程都有自己独立的对象池，这样可以完全避免锁，但是资源不能全局调度。

ii. 对象池半局部化。这是一种折中的办法，指主线程控制所有的资源，线程来取对象的时候采用批发的方式要一大批对象过去，退还的时候也批量退还，这样基本上就有效地解决了锁碰撞的问题。

对象池可以很好的解决固定大小的对象问题，但是对于变长类型，如字符串就不大起作用了，而这也是 web 服务器常常消耗非常大的一些方面。

高性能 Server 专用对象池！！

这才是我真正想要说的终极方案！！！

我写的所有 server 几乎无一例外的使用了这一方式的对象缓存，极其简单，极其有效，完全解决了内存分配的所有问题……

事情是这样的：

Server 服务器的设计都是围绕着 socket 展开的，通常会把 socket fd 包装成一个对象，TencentWebProxy 使用的是 epoll，每个 fd 包装成一个 poller 对象。而 fd 是天然的数组下标！！！并且天然地具有唯一性！！！

所以，直接静态分配 poller 对象数组，数组大小为 ulimit 中 fd 的上限。当 accept 到一个 fd 时，直接到 poller[fd]就找到唯一属于它的 poller 对象了。对，数组可以随机访问！

那字符串类的对象呢。。。

由于整个 server 是以 poller 为中心设计的，poller 对象的内存 buffer 都是跟 poller 有关联的，比如说 recv_buff, send_buff, tmp_buff 等等，都会被 poller 用指针或都内嵌 buffer 对象引用到(TencentWebProxy 中使用 xbuffer_t 来管理字符串和变长 buffer)。这些内存被使用到的时候，直接分配，fd 关闭时，调用 poller 的 reset 函数，将 poller 的对象里面的所有内

容重置，以供下次使用。对于已经分配的 buffer 不释放，只要把标记 buffer 里数据长度(len)的变量置为 0 即可。

```
 8 #define REALLOC_EXTRA_SIZE 128
 9
10 typedef struct {
11     char    *buf;
12     uint32_t  size;
13     uint32_t  len;
14 } xbuffer_t;
```

这样一来，只有 server 一启动的时候会分配内存，当收到更大的数据包的时候 xbuffer_t 的 buf 会 remalloc 一下，相应的 size 也会变大。当运行一段时间后，server 几乎没有内存分配操作了。

当系统中的内存吃紧的时候，可以沿着 poller[]统一回收一下内存。不过在实际运营过程中发现，这个设计其实没大有必要。

！！！使用以 fd 为下标的对象数据，一定要记住对象 reset 完成之后，才能关闭 fd。提前关闭 fd 会造成其它线程重新使用了这个 fd,进而两个线程同时操作同一个 poller 对象，造成内存错误。本人消耗了一下午的时候来解决这个 bug。。。这儿的时序是一定要保障的，为了安全，我还在这儿加了内存屏障。

相关代码：

分配 poller 静态数组

```
79      NEW_ARRAY(g_max_fd,Connection,g_cnn_array);
80
81      if(g_cnn_array == NULL){
82          log_error("New Connection array failed");
83          exit(-1);
84      }
```

Conncetion 是 poller 的子类。

NEW_ARRAY 是分配数组对象的宏，定义如下：

```
82 #define NEW_ARRAY(n, type, pointer) \
83      do \
84      { \
85          try \
86          { \
87              pointer = 0; \
88              pointer = new type[n]; \
89          } \
90          catch (...) \
91          { \
92              pointer = 0; \
93          } \
94      } while(0)
```

poller 对象时序保证的内存屏障：

```
41 #if __GNUC__ < 4
42 static inline void barrier(void) { __asm__ volatile("":::"memory"); }
43 #else
44 static inline void barrier(void) { __sync_synchronize (); }
45 #endif
```

字符串方面操作的优化：

1. 用数字比较代替字符串比较，nginx 中也大量使用了这个技巧。这个技巧我之前在微博上提过。

2. 不用 memset 将 buffer 置 0，只要将第一个字节置 0，然后使用安全的字符串操作函数就不可能出错。
3. 当 xbuffer_t 在扩展 buf 大小的时候，每次以 128 的粒度增加，避免频繁扩展。(128 是经验值)
4. 将 memcpy,strlen 等函数自己实现或将代码 copy 过来然后 inline 掉，将会极大减少函数调用开销。

效果

经过以上方面的优化，我们在使用 perf 来看 TWP 的运行情况来看，已经非常理想了，占用率高的，不管是内核态还是用户态的，都是有效操作。

```
------------------------------------------------------------------------------------------------------------------------
  3989 irqs/sec  kernel:87.2%  exact:  0.0% [1000Hz cycles],  (target_pid: 26109)
------------------------------------------------------------------------------------------------------------------------

samples  pcnt function                                                DSO
_______ _____ ________________________________________________________ ____________________________________________________

2298.00  3.9% _spin_lock                                              [kernel.kallsyms]
2124.00  3.6% _spin_lock_bh                                           [kernel.kallsyms]
2084.00  3.6% tcp_sendmsg                                            [kernel.kallsyms]
2057.00  3.5% igb_poll                                               [kernel.kallsyms]
1841.00  3.1% tcp_packet                                             [kernel.kallsyms]
1598.00  2.7% ipt_do_table                                           [kernel.kallsyms]
1435.00  2.4% igb_xmit_frame_ring_adv                                 [kernel.kallsyms]
1414.00  2.4% skb_release_data                                       [kernel.kallsyms]
1227.00  2.1% nf_conntrack_find_get                                   [kernel.kallsyms]
1143.00  2.0% tcp_ack                                                [kernel.kallsyms]
 976.00  1.7% tcp_recvmsg                                            [kernel.kallsyms]
 928.00  1.6% _spin_lock_irqsave                                     [kernel.kallsyms]
 902.00  1.5% Connection::http_state_proc(bool)                      /home/user_00/twp/twp/src/TencentWebF
 840.00  1.4% nf_conntrack_in                                        [kernel.kallsyms]
 835.00  1.4% http_header_parse(http_header*, xbuffer_t*, http_code_t*) /home/user_00/twp/twp/src/TencentWebF
 819.00  1.4% sk_run_filter                                          [kernel.kallsyms]
 794.00  1.4% tcp_rcv_established                                    [kernel.kallsyms]
 788.00  1.3% copy_user_generic_string                               [kernel.kallsyms]
 775.00  1.3% tcp_poll                                               [kernel.kallsyms]
 708.00  1.2% ip_route_input                                         [kernel.kallsyms]
 674.00  1.2% sock_wfree                                             [kernel.kallsyms]
```

呵呵，效果还不错吧。。

跋

昨天晚上写到 3 点多，今天又起了个大早起写完了这篇文章，内心肯定是狂野地期待各位网友的反馈和意见。

这也是我把这个系列文章写下去的所有动力。。

我在 **http://weibo.com/codebox** 期待大家的交流。