# 背景

系统的物理内存是有限的，而对内存的需求是变化的, 程序的动态性越强，内存管理就越重要，选择合适的内存管理算法会带来明显的性能提升。

比如nginx，它在每个连接accept后会malloc一块内存，作为整个连接生命周期内的内存池。当HTTP请求到达的时候，又会malloc一块当前请求阶段的内存池, 因此对malloc的分配速度有一定的依赖关系。(而apache的内存池是有父子关系的，请求阶段的内存池会和连接阶段的使用相同的分配器，如果连接内存池释放则请求阶段的子内存池也会自动释放)。

内存管理可以分为三个层次，自底向上分别是：

1、操作系统内核的内存管理。

2、glibc层使用系统调用维护的内存管理算法。

3、应用程序从glibc动态分配内存后，根据应用程序本身的程序特性进行优化，比如使用引用计数std::shared\_ptr，apache的内存池方式等等。当然应用程序也可以直接使用系统调用从内核分配内存，自己根据程序特性来维护内存，但是会大大增加开发成本。

一个优秀的通用内存分配器应具有以下特性:

额外的空间损耗尽量少

分配速度尽可能快

尽量避免内存碎片

缓存本地化友好

通用性，兼容性，可移植性，易调试

目前大部分服务端程序使用glibc提供的malloc/free系列函数，而glibc使用的ptmalloc2在性能上远远弱后于google的tcmalloc和facebook的jemalloc。 而且后两者只需要使用LD\_PRELOAD环境变量启动程序即可，甚至并不需要重新编译。

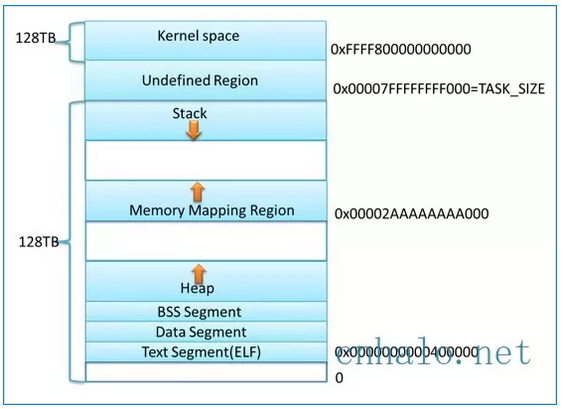
# 分类

## ptmalloc2

ptmalloc2即是我们当前使用的glibc malloc版本。

### 原理

系统调用接口



上图是x86\_64下Linux进程的默认地址空间，对heap的操作，操作系统提供了brk()系统调用，设置了Heap的上边界；对mmap映射区域的操作，操作系统供了mmap()和munmap()函数。

因为系统调用的代价很高，不可能每次申请内存都从内核分配空间，尤其是对于小内存分配。而且因为mmap的区域容易被munmap释放，所以一般大内存采用mmap()，小内存使用brk()。

### 多线程支持

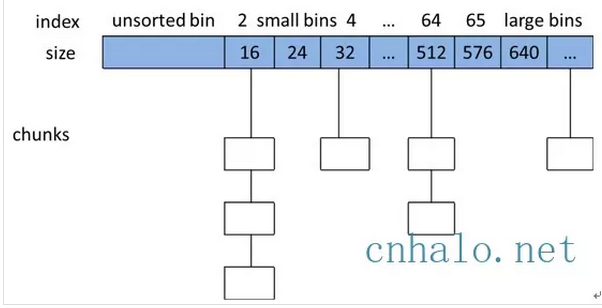
Ptmalloc2有一个主分配区(main arena)，有多个非主分配区。非主分配区只能使用mmap向操作系统批发申请HEAP\_MAX\_SIZE（64位系统为64MB）大小的虚拟内存。当某个线程调用malloc的时候，会先查看线程私有变量中是否已经存在一个分配区，如果存在则尝试加锁，如果加锁失败则遍历arena链表试图获取一个没加锁的arena，如果依然获取不到则创建一个新的非主分配区。

free()的时候也要获取锁。分配小块内存容易产生碎片，ptmalloc在整理合并的时候也要对arena做加锁操作。在线程多的时候，锁的开销就会增大。

### 内存管理

用户请求分配的内存在ptmalloc中使用chunk表示，每个chunk至少需要8个字节额外的开销。用户free掉的内存不会马上归还操作系统，ptmalloc会统一管理heap和mmap区域的空闲chunk，避免了频繁的系统调用。

ptmalloc将相似大小的chunk用双向链表链接起来, 这样的一个链表被称为一个bin。ptmalloc一共维护了128个bin，并使用一个数组来存储这些bin(如下图所示)。



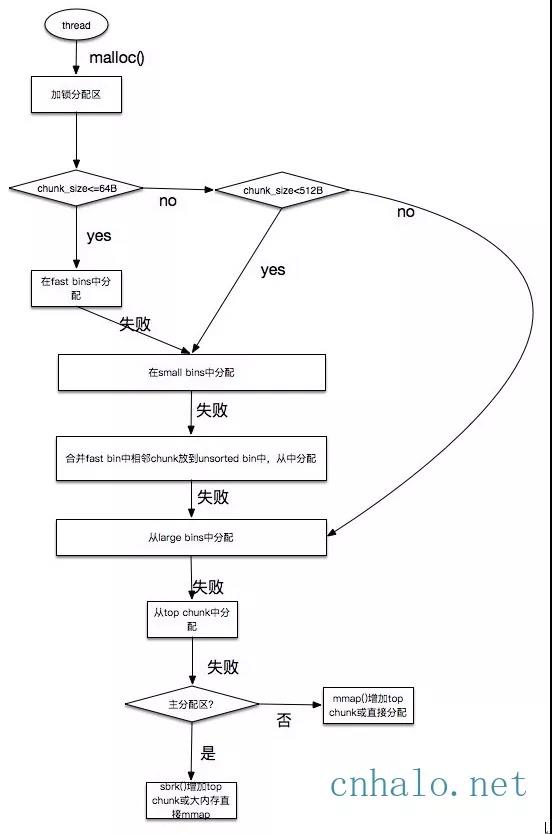
数组中的第一个为unsorted bin, 数组中从2开始编号的前64个bin称为 small bins，同一个small bin中的chunk具有相同的大小。small bins后面的bin被称作large bins。

当free一个chunk并放入bin的时候， ptmalloc还会检查它前后的chunk是否也是空闲的，如果是的话，ptmalloc会首先把它们合并为一个大的chunk，然后将合并后的chunk放到unstored bin中。另外ptmalloc为了提高分配的速度,会把一些小的(不大于64B) chunk先放到一个叫做fast bins的容器内。

在fast bins和bins都不能满足需求后，ptmalloc会设法在一个叫做top chunk的空间分配内存。 对于非主分配区会预先通过mmap分配一大块内存作为top chunk， 当bins和fast bins都不能满足分配需要的时候, ptmalloc会设法在top chunk中分出一块内存给用户, 如果top chunk本身不够大, 分配程序会重新mmap分配一块内存chunk, 并将 top chunk 迁移到新的chunk上，并用单链表链接起来。如果free()的chunk恰好 与 top chunk 相邻,那么这两个 chunk 就会合并成新的 top chunk，如果top chunk大小大于某个阈值才还给操作系统。主分配区类似，不过通过sbrk()分配和调整top chunk的大小，只有heap顶部连续内存空闲超过阈值的时候才能回收内存。

需要分配的 chunk 足够大,而且 fast bins 和 bins 都不能满足要求,甚至 top chunk 本身也不能满足分配需求时,ptmalloc 会使用 mmap 来直接使用内存映射来将页映射到进程空间。

### 分配流程



### 缺陷

后分配的内存先释放，因为ptmalloc收缩内存是从top chunk开始，如果与top chunk相邻的chunk不能释放，top chunk以下的chunk都无法释放。

多线程锁开销大，需要避免多线程频繁分配释放。

内存从thread的areana中分配， 内存不能从一个arena移动到另一个arena， 就是说如果多线程使用内存不均衡，容易导致内存的浪费。比如说线程1使用了300M内存，完成任务后glibc没有释放给操作系统，线程2开始创建了一个新的arena，但是线程1的300M却不能用了。

每个chunk至少8字节的开销很大。

不定期分配长生命周期的内存容易造成内存碎片，不利于回收。64位系统最好分配32M以上内存，这是使用mmap的阈值。

## tcmalloc

### 概述

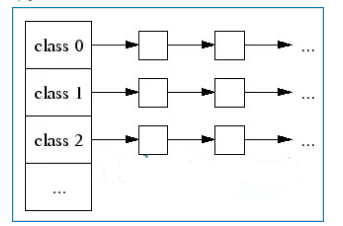
tcmalloc是Google开源的一个内存管理库，作为glibc malloc的替代品。目前已经在chrome、safari等知名软件中运用。

根据官方测试报告，ptmalloc在一台2.8GHz的P4机器上（对于小对象）执行一次malloc及free大约需要300纳秒，而tcmalloc的版本同样的操作大约只需要50纳秒。

### 小对象分配

tcmalloc为每个线程分配了一个线程本地ThreadCache，小内存从ThreadCache分配，此外还有个中央堆（CentralCache），ThreadCache不够用的时候，会从CentralCache中获取空间放到ThreadCache中。

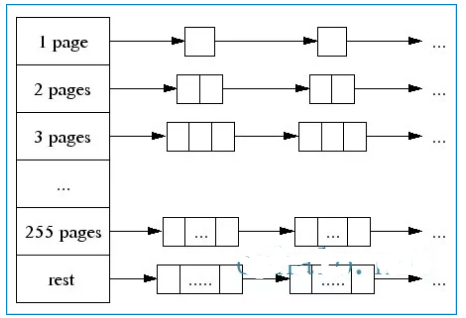
小对象（<=32K）从ThreadCache分配，大对象从CentralCache分配。大对象分配的空间都是4k页面对齐的，多个pages也能切割成多个小对象划分到ThreadCache中。



小对象有将近170个不同的大小分类(class)，每个class有个该大小内存块的FreeList单链表，分配的时候先找到best fit的class，然后无锁的获取该链表首元素返回。如果链表中无空间了，则到CentralCache中划分几个页面并切割成该class的大小，放入链表中。

### CentralCache分配管理

大对象(>32K)先4k对齐后，从CentralCache中分配。CentralCache维护的PageHeap如下图所示，数组中第256个元素是所有大于255个页面都挂到该链表中。



当best fit的页面链表中没有空闲空间时，则一直往更大的页面空间则，如果所有256个链表遍历后依然没有成功分配。则使用sbrk, mmap, /dev/mem从系统中分配。

tcmalloc PageHeap管理的连续的页面被称为span。

如果span未分配，则span是PageHeap中的一个链表元素。

如果span已经分配，它可能是返回给应用程序的大对象，或者已经被切割成多小对象，该小对象的size-class会被记录在span中

在32位系统中，使用一个中央数组(central array)映射了页面和span对应关系， 数组索引号是页面号，数组元素是页面所在的span。在64位系统中，使用一个3-level radix tree记录了该映射关系。

### 回收

当一个object free的时候，会根据地址对齐计算所在的页面号，然后通过central array找到对应的span。

如果是小对象，span会告诉我们他的size class，然后把该对象插入当前线程的ThreadCache中。如果此时ThreadCache超过一个预算的值（默认2MB），则会使用垃圾回收机制把未使用的object从ThreadCache移动到CentralCache的central free lists中。

如果是大对象，span会告诉我们对象锁在的页面号范围。 假设这个范围是[p,q]， 先查找页面p-1和q+1所在的span，如果这些临近的span也是free的，则合并到[p,q]所在的span， 然后把这个span回收到PageHeap中。

CentralCache的central free lists类似ThreadCache的FreeList，不过它增加了一级结构，先根据size-class关联到spans的集合， 然后是对应span的object链表。如果span的链表中所有object已经free， 则span回收到PageHeap中。

### 改进

ThreadCache会阶段性的回收内存到CentralCache里。解决了ptmalloc2中arena之间不能迁移的问题。

Tcmalloc占用更少的额外空间。例如，分配N个8字节对象可能要使用大约8N \* 1.01字节的空间。即，多用百分之一的空间。Ptmalloc2使用最少8字节描述一个chunk。

更快。小对象几乎无锁， >32KB的对象从CentralCache中分配使用自旋锁。 并且>32KB对象都是页面对齐分配，多线程的时候应尽量避免频繁分配，否则也会造成自旋锁的竞争和页面对齐造成的浪费。

## jemalloc

jemalloc是facebook推出的，最早的时候是freebsd的libc malloc实现。 目前在firefox、facebook服务器各种组件中大量使用。

### 原理

与tcmalloc类似，每个线程同样在<32KB的时候无锁使用线程本地cache。

Jemalloc在64bits系统上使用下面的size-class分类：

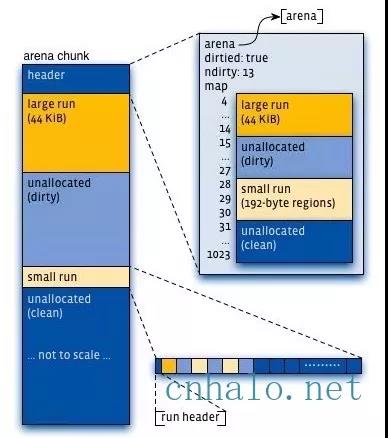
Small: [8], [16, 32, 48, …, 128], [192, 256, 320, …, 512], [768, 1024, 1280, …, 3840]

Large: [4 KiB, 8 KiB, 12 KiB, …, 4072 KiB]

Huge: [4 MiB, 8 MiB, 12 MiB, …]

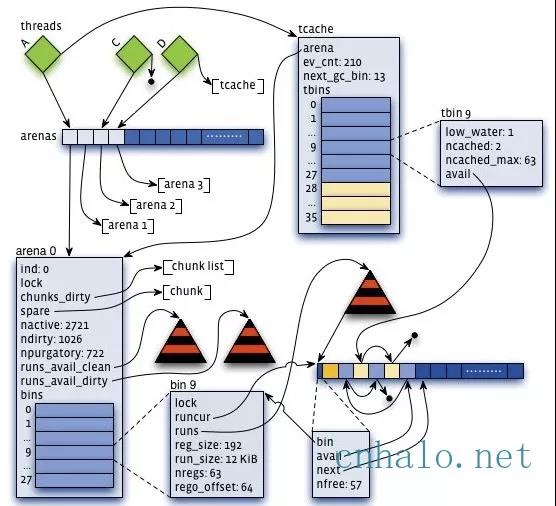
small/large对象查找metadata需要常量时间， huge对象通过全局红黑树在对数时间内查找。

虚拟内存被逻辑上分割成chunks（默认是4MB，1024个4k页），应用线程通过round-robin算法在第一次malloc的时候分配arena， 每个arena都是相互独立的，维护自己的chunks， chunk切割pages到small/large对象。free()的内存总是返回到所属的arena中，而不管是哪个线程调用free()。



上图可以看到每个arena管理的arena chunk结构，开始的header主要是维护了一个page map（1024个页面关联的对象状态），header下方就是它的页面空间。 Small对象被分到一起， metadata信息存放在起始位置。 large chunk相互独立，它的metadata信息存放在chunk header map中。

通过arena分配的时候需要对arena bin（每个small size-class一个，细粒度）加锁，或arena本身加锁。并且线程cache对象也会通过垃圾回收指数退让算法返回到arena中。



### 使用

1. 编译
2. 使用jemalloc分析heap文件：

格式：jeprof –functions –svg/pdf 待分析的可执行文件 heap文件 > 结果存储

./jemalloc-5.2.0/bin/jeprof --functions --svg mysql-proxy jeprof.32728.2.i2.heap > xxxx.svg

注：如果想要显示函数名称不要增加格式控制符—show-bytes。

### 优化

Jmalloc小对象也根据size-class，但是它使用了低地址优先的策略，来降低内存碎片化。

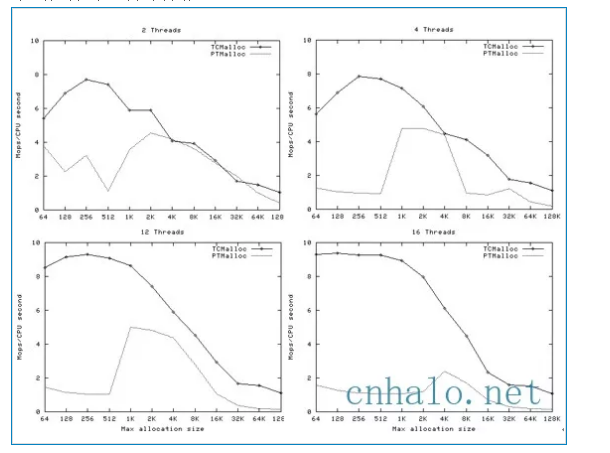
Jemalloc大概需要2%的额外开销。（tcmalloc 1%，ptmalloc最少8B）。

Jemalloc和tcmalloc类似的线程本地缓存，避免锁的竞争。

相对未使用的页面，优先使用dirty page，提升缓存命中。

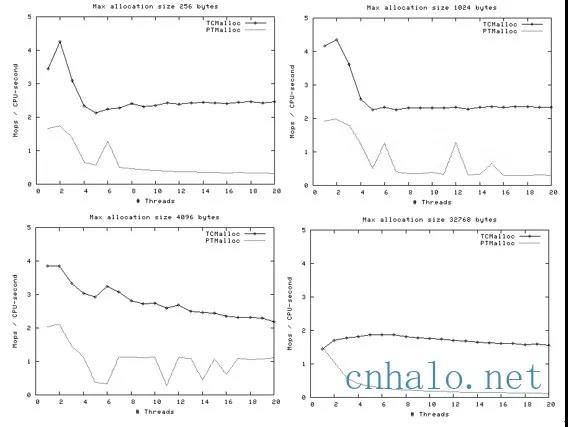
# 性能对比

测试环境是2.4GHz dual Xeon，开启超线程，redhat9，glibc-2.3.2, 每个线程测试100万个操作。



tcmalloc测试1

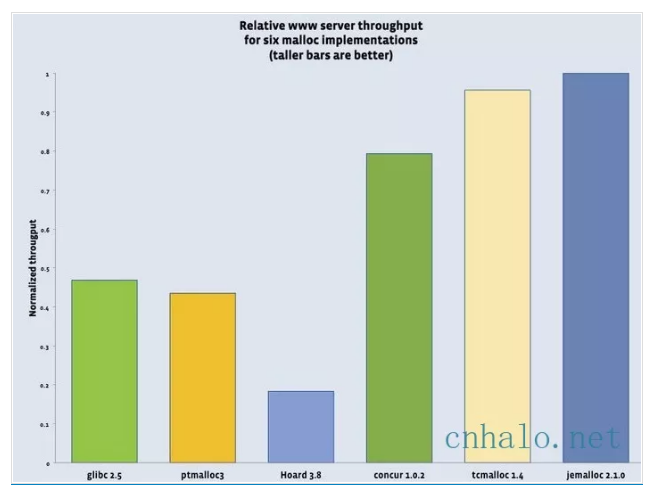
上图中可以看到尤其是对于小内存的分配， tcmalloc有非常明显性能优势。



tcmalloc测试2

上图可以看到随着线程数的增加，tcmalloc性能上也有明显的优势，并且相对平稳。

jemalloc官方测试报告

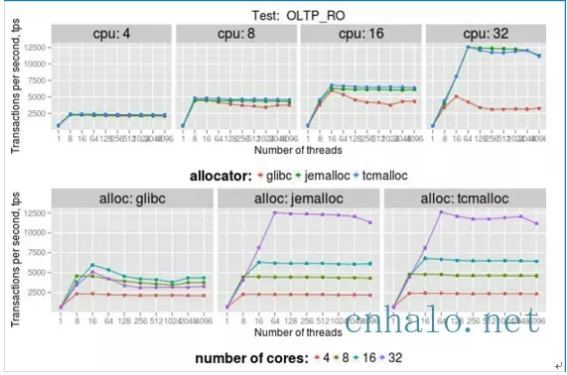


上图是服务器吞吐量分别用6个malloc实现的对比数据，可以看到tcmalloc和jemalloc最好(facebook在2011年的测试结果，tcmalloc这里版本较旧)。

**mysql优化：**

测试环境：2x Intel E5/2.2Ghz with 8 real cores per socket，16 real cores， 开启hyper-threading，总共32个vcpu。16个table，每个5M row。

OLTP\_RO测试包含5个select查询：select\_ranges, select\_order\_ranges, select\_distinct\_ranges, select\_sum\_ranges,



可以看到在多核心或者多线程的场景下， jemalloc和tcmalloc带来的tps增加非常明显。

**参考资料：**

glibc内存管理ptmalloc源代码分析

Inside jemalloc

tcmalloc浅析

tcmalloc官方文档

Scalable memory allocation using jemalloc

mysql-performance-impact-of-memory-allocators-part-2

ptmalloc,tcmalloc和jemalloc内存分配策略研究

Tick Tock, malloc Needs a Clock

# 总结

在多线程环境使用tcmalloc和jemalloc效果非常明显。

当线程数量固定，不会频繁创建退出的时候，可以使用jemalloc；反之使用tcmalloc可能是更好的选择。