# GLIBC内存管理(ptmalloc2)

## 内存在进程中是如何分段

进程的栈和 mmap 映射区域并不是从一个固定地址开始，并且每次启动时的值都不一样，这是程序在启动时随机改变这些值的设置，使得使用缓冲区溢出进行攻击更加困难。当然也可以让进程的栈和 mmap 映射区域从一个固定位置开始，只需要设置全局变量randomize\_va\_space 值为 0 ， 这 个 变 量 默 认 值 为 1 。 用 户 可 以 通 过 设 置/proc/sys/kernel/randomize\_va\_space来停用该特性，也可以用如下命令：

sudo sysctl -w kernel.randomize\_va\_space=0

一个很重要的概念，内存的延迟分配，只有在真正访问一个地址的时候才建立这个地址的物理映射，这是Linux内存管理的基本思想之一。Linux内核在用户申请内存的时候，只是给它分配了一个线性区（也就是虚拟内存），并没有分配实际物理内存；只有当用户使用这块内存的时候，内核才会分配具体的物理页面给用户，这时候才占用宝贵的物理内存。

Heap segment:

#include <unistd.h>

int brk(void \*end\_data\_segment); /\*end heap addr\*/

void \*sbrk(intptr\_t increment);

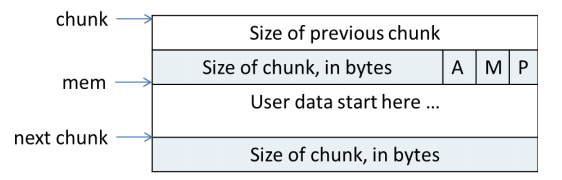
map segment:

#include <sys/mman.h>

void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);

int munmap(void \*start, size\_t length);

## 动态内存是如何管理？



在图中，chunk 指针指向一个 chunk的开始，一个 chunk中包含了用户请求的内存区域和相关的控制信息。图中的 mem 指针才是真正返回给用户的内存指针。

chunk 的第二个域的最低一位为 P，它表示前一个块是否在使用中，P 为 0 则表示前一个 chunk 为空闲，这时 chunk的第一个域prev\_size 才有效，prev\_size 表示前一个 chunk的 size，程序可以使用这个值来找到前一个chunk 的开始地址。当 P为 1时，表示前一个 chunk 正在使用中，prev\_size无效，程序也就不可以得到前一个chunk的大小。 不能对前一个chunk进行任何操作。 ptmalloc分配的第一个块总是将P设为1，以防止程序引用到不存在的区域。

Chunk的第二个域的倒数第二个位为 M，他表示当前 chunk是从哪个内存区域获得的虚拟内存。M为1 表示该 chunk是从mmap映射区域分配的，否则是从heap区域分配的。

Chunk的第二个域倒数第三个位为A，表示该chunk 属于主分配区或者非主分配区，如果属于非主分配区，将该位置为 1，否则置为0。

Malloc分配是上述chunk结构占用2\*size\_t字节。

分配与回收

主分配区，非主分配区

Bins,fast bins, mmap, top chunk

Start brk,brk

**分配流程**

根据用户请求分配的内存的大小，ptmalloc 有可能会在两个地方为用户分配内存空间。在第一次分配内存时，一般情况下只存在一个主分配区，但也有可能从父进程那里继承来了多个非主分配区，在这里主要讨论主分配区的情况，brk 值等于start\_brk，所以实际上 heap 大小为 0，top chunk 大小也是 0。这时，如果不增加 heap大小， 就不能满足任何分配要求。 所以， 若用户的请求的内存大小小于 mmap分配阈值，则 ptmalloc会初始heap。然后在heap中分配空间给用户，以后的分配就基于这个 heap进行。若第一次用户的请求就大于 mmap 分配阈值，则 ptmalloc 直接使用 mmap()分配一块内存给用户，而 heap 也就没有被初始化，直到用户第一次请求小于 mmap 分配阈值的内存分配。 第一次以后的分配就比较复杂了， 简单说来， ptmalloc首先会查找fast bins，如果不能找到匹配的 chunk，则查找small bins。若还是不行，合并 fast bins，把 chunk加入unsorted bin，在 unsorted bin中查找，若还是不行，把 unsorted bin 中的 chunk全加入large bins中，并查找large bins。 在fast bins和small bins中的查找都需要精确匹配，而在 large bins 中查找时，则遵循“smallest-first，best-fit”的原则，不需要精确匹配。

若以上方法都失败了，则ptmalloc会考虑使用top chunk。若top chunk也不能满足分配要求。而且所需 chunk 大小大于 mmap 分配阈值，则使用 mmap 进行分配。否则增加heap，增大top chunk。以满足分配要求。

## 其他内存分配库

Tcmalloc：包含于Google Performance Tools的项目中，它在多线程情况表现优秀，常见于safari，chrome浏览器，加速mysql等。

论文：http://goog-perftools.sourceforge.net/doc/tcmalloc.html

在tcmalloc的论文《TCMalloc : Thread-Caching Malloc》中，提到了ptmalloc2在多线程环境下的一个致命缺陷：

1.TCMalloc要比glibc 2.3的malloc（可以从一个叫作ptmalloc2的独立库获得）和其他我测试过的malloc都快。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。malloc版本的速度是至关重要的，因为如果malloc不够快，应用程序的作者就很有可能在malloc之上写一个自己的自由列表。这就可能导致额外的代码复杂度，以及更多的内存占用――除非作者本身非常仔细地划分自由列表的大小并经常从自由列表中清除空闲的对象。

2.TCMalloc减少了多线程程序中的锁争用情况。对于小对象，几乎已经达到了零争用。对于大对象，TCMalloc尝试使用粒度较好和有效的自旋锁。

3.TCMalloc的另一个好处是小对象的空间最优表现形式。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。而ptmalloc2中每个对象都使用了一个四字节的头，（我认为）并将最终的尺寸规整为8字节的倍数，最后使用了16N字节。

实时计算引擎处理延迟的排查过程

<http://www.cnblogs.com/panfeng412/archive/2012/03/26/real-time-computing-engine-processing-delay-troubleshoot.html>

使用tcmalloc: export LD\_PRELOAD="./libtcmalloc.so"

LD\_PRELOAD这个环境变量很意思：

/\* 文件名：hack.c \*/

#include <stdio.h>

#include <string.h>

int strcmp(const char \*s1, const char \*s2)

{

printf("hack function invoked. s1=<%s> s2=<%s>\n", s1, s2);

/\* 永远返回0，表示两个字符串相等 \*/

return 0;

}

编译程序：

$ gcc -shared -o hack.so hack.c

 $ export LD\_PRELOAD="./hack.so"

如果是密码判断中用到strcmp则直接可以跳过密码检查进入系统。解决办法就是用静态链接库。

缺点：笔者尝试过。tcmalloc不能正确用valgrind,只能用自带gperftools（运行中会core)

jemalloc可以使用valgrind，不过还没完全验证是否都准确。

TCmalloc对squid的性能的提升

<http://www.apparitor.info/2011/03/29/61/>

Jemalloc：freebsd的内存分配库。Redis中默认带有它的源代码。Facebook的C++基础类folly也用它替换了glibc的内存分配模块，firefox也是使用了它的内存分配模块。

它最重要的特性体现在arena和threadcache.

更好的内存管理-jemalloc

<http://wangkaisino.blog.163.com/blog/static/1870444202011431112323846/>

Redis中的一段代码：

/\* Explicitly override malloc/free etc when using tcmalloc. \*/

#if defined(USE\_TCMALLOC)

#define malloc(size) tc\_malloc(size)

#define calloc(count,size) tc\_calloc(count,size)

#define realloc(ptr,size) tc\_realloc(ptr,size)

#define free(ptr) tc\_free(ptr)

#elif defined(USE\_JEMALLOC)

#define malloc(size) je\_malloc(size)

#define calloc(count,size) je\_calloc(count,size)

#define realloc(ptr,size) je\_realloc(ptr,size)

#define free(ptr) je\_free(ptr)

#endif

# 如何设计你的内存池

C语言的内存管理一直是开发者绕不过去的坎，像memcached这种接收大量请求的框架来说，频繁的内存调用和内存泄露是名副其实的性能杀手。malloc/free有三大缺点：1.容易造成内存泄露；2.频繁调用会造成大量内存碎片无法回收重新利用，降低内存利用率；3.作为系统调用，其系统开销远远大于一般函数调用。现在最常用的内存管理方案是使用内存池替代malloc/free调用，内存池方案的基本思想是预先分配一大块内存，此后的内存申请与释放都只操作这块内存。

## 基本知识

Malloc:

在C语言和C++的是有一个小区别：

C语言时不用void类型的强制转换：

Char \* p;

P = malloc(2048);

但在C++中就必须做强制转换：

Char \*p;

P =(char \*)malloc(2048);

C++引用C语言函数，暂时还没有测试。建议是都加上强转。

Malloc可以加一个错误的封闭xmalloc，加一个x做封闭函数，实际项目中建议把日志文件描述符加入其中。

Void \*Xmalloc(size\_t size)

{

Void \*p;

P = malloc(size);

If (p == NULL) {

Perror(“xmalloc”);

Exit(EXIT\_FAILURE);

}

Return p;

}

效率：calloc() > malloc()+memset();

Mmap不用再memset()0了

## 如何调试：

1.编译内存输出的函数：

mem\_dump()

2.通过gdb输出内存数据

x/32b &variable

x/32x & variable

3.加入系统内存信息输出：

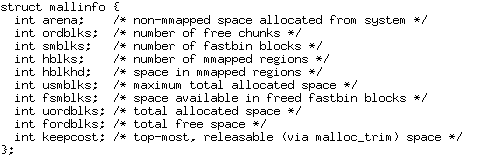
设置环境变量：export MALLOC\_CHECK\_=1;则会自动输出钩子检测。

如：在代码中重复free()某一段内存：





通过结构mallinfo输出内存状态:



通过mallinfo()直接返回当前mallinfo结构体数据。

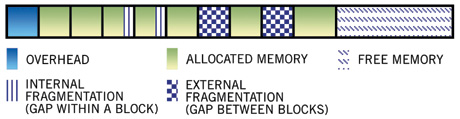
## 内存设计要点

内存泄漏(memory leak)：

分配的内存没有free还给系统，指向内存的指针也消失了，内存再也无法交还给系统。

内存碎片(memory fragmentation)：

系统中所有不可用的空闲内存，内存碎片是因为在分配一个内存块后，使之空闲，但不将空闲内存归还给最大内存块而产生的。通俗的说：内部碎片就是分多了的意思。外部碎片就是小块分存不能合起来用。编写内存池尤其要关注内存碎片问题：

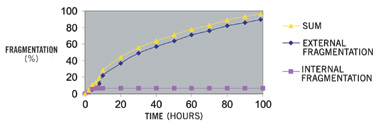


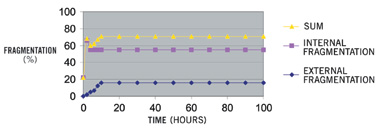
内部碎片：所有的内存分配必须起始于可被4、8 或 16 整除（视[处理](http://www.ednchina.com/word/63897.aspx" \t "_blank)器体系结构而定）的地址。内存分配程序把仅仅预定大小的内存块分配给客户，可能还有其它原因。当某个客户请求一个 43 字节的内存块时，它可能会获得 44字节、48字节 甚至更多的字节。由所需大小四舍五入而产生的多余空间就叫内部碎片。

外部碎片：一个应用程序分配三个连续的内存块，然后使中间的一个内存块空闲。内存分配程序可以重新使用中间内存块供将来进行分配，但不太可能分配的块正好与全部空闲内存一样大。

**碎片=1-当前可分配的最大内存块大小/空闲内存大小**

一个系统有 5M 字节的空闲内存，当它可用来分配的最大内存块为 50k 字节时，其内存碎片为1-50K/5M=99%，如果可分配的最大小内存块为500K，则碎片=1-500K/5120K=90%





## 内存池设计算法

最先适合分配算法：

直接在第一块空闲中分配内存，通常会有4个指针：

m\_start：内存开始端

m\_end：内存结束点

m\_break：已使用内存末端

m\_free：未使用内存开始端

最先适合内存分配算法实现起来简单，而且开始时很好用。但是，经过一段时间后，会出现如下的情况：当系统将内存交给自由表时，它会从自由表的开头部分去掉大内存块，插入剩余的小内存块。最先适合算法实际上成了一个排序算法，即把所有小内存[碎片](http://www.ednchina.com/word/63899.aspx)放在自由表的开头部分。因此，自由表会变得很长，有几百甚至几千个元素。因此，内存分配变得时间很长又无法预测，大内存块分配所花时间要比小内存块分配来得长。另外，内存块的无限制拆分使内存碎片程度很高。

最为常见

最佳/最差适合分配算法：

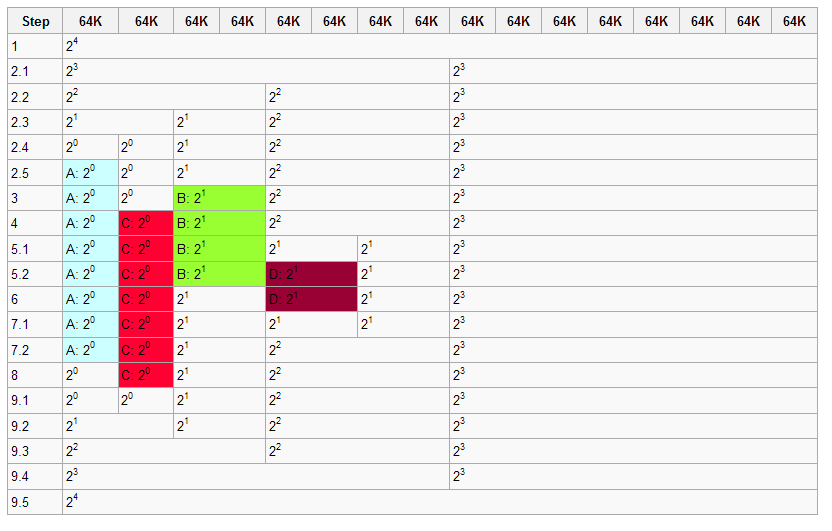
系统在分配一个内存块时，要搜索整个自由表，寻找最接近请求存储量的内存块。这种搜索所花的时间要比最先适合算法长得多，但不存在分配大小内存块所需时间的差异。最佳适合算法产生的内存碎片要比最先适合算法多，因为将小而不能使用的碎片放在自由表开头部分的排序趋势更为强烈。而最差适合分配算法，始终选择最大空闲内存块做内存分配。

很少使用

伙伴分配算法(buddy)：

每个块都有个朋友，或叫“伙伴”，既可与之分开，又可与之结合。伙伴分配程序把内存块存放在比链接表更先进的数据结构中。这些结构常常是桶型、树型和堆型的组合或变种。伙伴分配程序通常在某种程度上限制内存碎片。

应用不少



1 初始化时，系统拥有1M的连续内存，允许的最小的内存块为64K，图中白色的部分为空闲的内存块，着色的代表分配出去了得内存块。

2 程序A申请一块大小为34K的内存，对应的order为0，即2^0=1个最小内存块

   2.1 系统中不存在order 0(64K)的内存块，因此order 4(1M)的内存块分裂成两个order 3的内存块(512K)

   2.2 仍然没有order 0的内存块，因此order 3的内存块分裂成两个order 2的内存块(256K)

   2.3 仍然没有order 0的内存块，因此order 2的内存块分裂成两个order 1的内存块(128K)

   2.4 仍然没有order 0的内存块，因此order 1的内存块分裂成两个order 0的内存块(64K)

   2.5 找到了order 0的内存块，将其中的一个分配给程序A，现在伙伴系统的内存为一个order 0的内存块，一个order1的内存块，一个order 2的内存块以及一个order 3的内存块

3 程序B申请一块大小为66K的内存，对应的order为1，即2^1=2个最小内存块，由于系统中正好存在一个order 1的内存块，所以直接用来分配

4 程序C申请一块大小为35K的内存，对应的order为0，同样由于系统中正好存在一个order 0的内存块，直接用来分配

5 程序D申请一块大小为67K的内存，对应的order为1

   5.1 系统中不存在order 1的内存块，于是将order 2的内存块分裂成两块order 1的内存块

   5.2 找到order 1的内存块，进行分配

6 程序B释放了它申请的内存，即一个order 1的内存块

7 程序D释放了它申请的内存

   7.1 一个order 1的内存块回收到内存当中

   7.2由于该内存块的伙伴也是空闲的，因此两个order 1的内存块合并成一个order 2的内存块

8 程序A释放了它申请的内存，即一个order 0的内存块

9 程序C释放了它申请的内存

   9.1 一个order 0的内存块被释放

   9.2 两个order 0伙伴块都是空闲的，进行合并，生成一个order 1的内存块m

   9.3 两个order 1伙伴块都是空闲的，进行合并，生成一个order 2的内存块

   9.4 两个order 2伙伴块都是空闲的，进行合并，生成一个order 3的内存块

   9.5 两个order 3伙伴块都是空闲的，进行合并，生成一个order 4的内存块

**尽可能多次反复使用内存块，而不要每次都对内存块进行分割，以正好符合请求的存储量。分割内存块会产生大量的小内存碎片，犹如一堆散沙。以后很难把这些散沙与其余内存结合起来。比较好的办法是让每个内存块中都留有一些未用的字节。**

**Linux采用伙伴系统解决外部碎片的问题，采用slab解决内部碎片的问题**

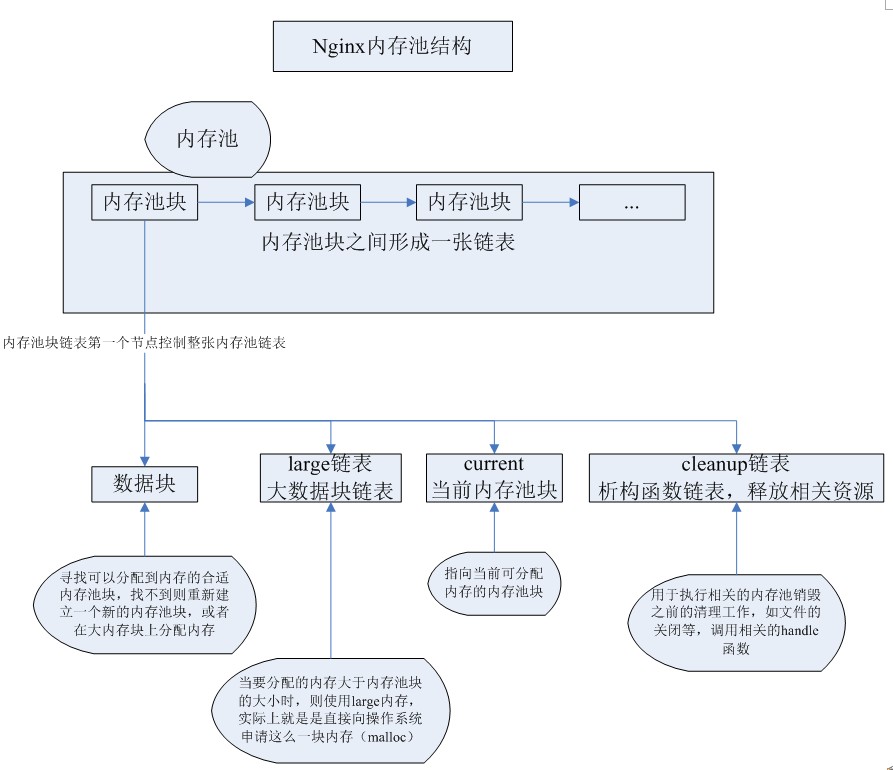
slab的核心思想是以对象的观点来管理内存

Linux 内核在运行过程中，常常会需要经常使用一些内核的数据结构（对象）。例如，当进程的某个线程第一次打开一个文件的时候，内核需要为该文件分配一个称为 file 的数据结构；当该文件被最终关闭的时候，内核必须释放此文件所关联的 file 数据结构。这些小块存储空间并不只在某个内核函数的内部使用，否则就可以使用当前线程的内核栈空间。同时，这些小块存储空间又是动态变化的，不可能像物理内存页面管理使用的 page 结构那样，有多大内存就有多少个 page 结构，形成一个静态长度的队列。而且由于内核无法预测运行中各种不同的内核对象对缓冲区的需求，因此不适合为每一种可能用到的对象建立一个“缓冲池”，因为那样的话很可能出现有些缓冲池已经耗尽而有些缓冲池中却又大量空闲缓冲区的现象。因此，内核只能采取更全局性的方法。  
我们可以看出，内核对象的管理与用户进程中的堆管理比较相似，核心问题均是：如何高效地管理内存空间，使得可以快速地进行对象的分配和回收并减少内存碎片。但是内核不能简单地采用用户进程的基于堆的内存分配算法，这是因为内核对其对象的使用具有以下特殊性：  
1.        内核使用的对象种类繁多，应该采用一种统一的高效管理方法。  
2.        内核对某些对象（如 task\_struct）的使用是非常频繁的，所以用户进程堆管理常用的基于搜索的分配算法比如First-Fit（在堆中搜索到的第一个满足请求的内存块）和 Best-Fit（使用堆中满足请求的最合适的内存块）并不直接适用，而应该采用某种缓冲区的机制。  
3.        内核对象中相当一部分成员需要某些特殊的初始化（例如队列头部）而并非简单地清成全 0。如果能充分重用已被释放的对象使得下次分配时无需初始化，那么可以提高内核的运行效率。  
4.        分配器对内核对象缓冲区的组织和管理必须充分考虑对硬件高速缓存的影响。  
5.        随着共享内存的多处理器系统的普及，多处理器同时分配某种类型对象的现象时常发生，因此分配器应该尽量避免处理器间同步的开销，应采用某种 Lock-Free 的算法。

内存碎片处理技术

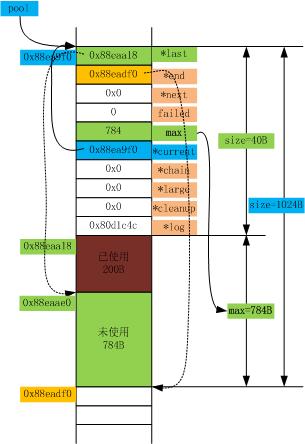
<http://www.52rd.com/Blog/Detail_RD.Blog_imjacob_16498.html>

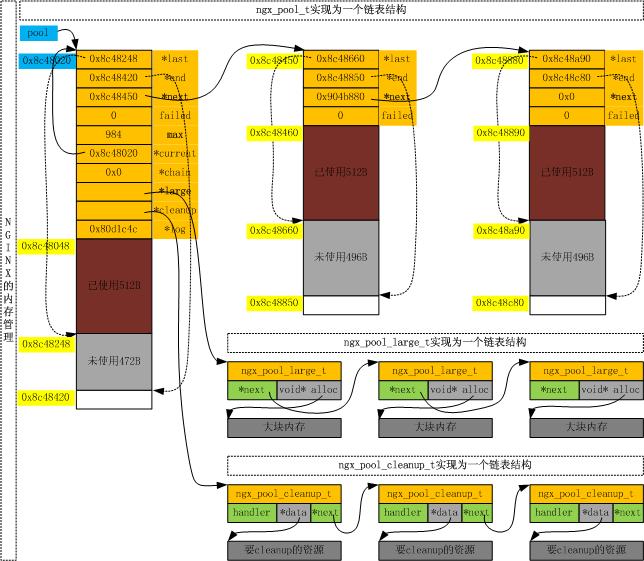
# nginx内存池

[](http://tomzhou-wordpress.stor.sinaapp.com/uploads/2012/01/%E5%86%85%E5%AD%98%E6%B1%A0%E7%BB%93%E6%9E%84.jpg)

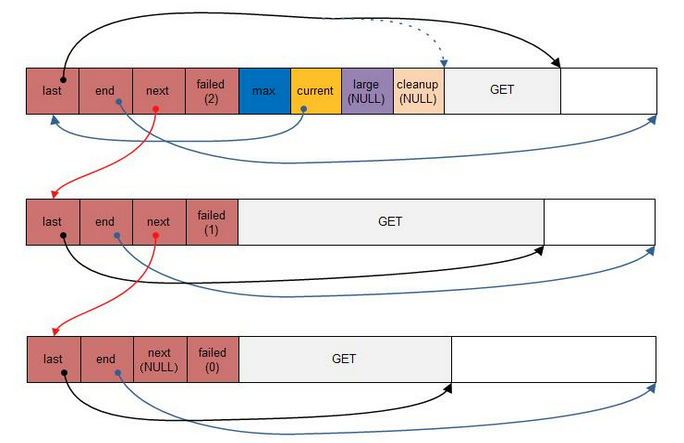
nginx的内存池操作也比较简单，下面是几个主要接口：  
1、创建内存池  ngx\_create\_pool  
2、销毁内存池  ngx\_destroy\_pool  
3、内存分配    ngx\_palloc  
4、释放large链表内存 ngx\_pfree  
5、直接调用palloc分配好内存，然后进行一次0初始化操作 ngx\_pcalloc  
6、注册cleanup  ngx\_pool\_cleanup\_add  
具体接口的算法可以看源码，其实都是比较简单的实现，但是却很高效和实用。  
**nginx的内存释放是利用了web server应用的特殊场景来完成；一个web server总是不停的接受connection 和request，所以nginx就将内存池分了不同的等级，有进程级的内存池、connection级的内存池、request级的内存池。也就是说，创建好一个worker进程的时候，同时为这个worker进程创建一个内存池，待有新的连接到来后，就在worker进程的内存池上为该连接创建起一个内存池；连接上到来一个request后，又在连接的内存池上为request创建起一个内存池。这样，在request被处理完后，就会释放request的整个内存池，连接断开后，就会释放连接的内存池。因而，就保证了内存有分配也有释放。**内存池的接口设计，开发人员很少需要关注内存的释放，只在确定不使用的时候调用destroy接口一次性释放内存池的所有内存资源，其余场合可以 利用alloc分配任意大小的内存，可以reset内存池避免重新分配和初始化，如果想减少内存碎片，可以在内存池之上实现自己的slab预分配策略，扩 展性很好







内存池第一块内存前40字节为ngx\_pool\_t结构，后续加入的内存块前16个字节为ngx\_pool\_data\_t结构，这两个结构之后便是真正可以分配内存区域。



Nginx源码剖析之内存池，与内存管理

<http://blog.csdn.net/v_july_v/article/details/7040425>

# MEMCACHE内存池

**Page 是memcached内存分配的单位**

Memcached的内存分配以page为单位，默认情况下一个page是1M，可以通过-I参数修改，最小1K，最大128M。如果需要申请内存时，memcached会划分出一个新的page并分配给需要的slab区域。page一旦被分配,在memcached重启前不会被回收或者重新分配

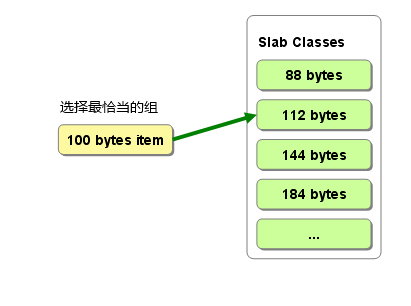
**Slab是memcached划分的数据空间**

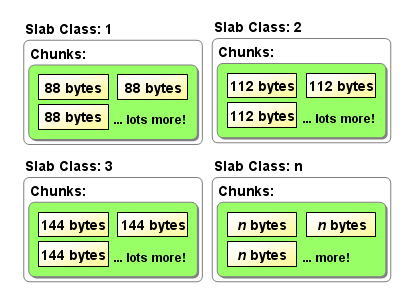
Slab可以理解为一个内存块，一个slab是memcached一次申请内存的最小单位，在memcached中，一个slab的大小默认为1048576字节（1MB），也就是1Page的单位。每一个slab被划分为若干个相等的chunk，每个chunk里保存一个item，memcached默认情况下下一个slab的chunk值为前一个slab的chunk值的1.25倍，这个可以通过修改-f参数来修改增长比例。

Memcached并不是将所有大小的数据都放在一起的，而是预先将一个个数据空间划分为一系列的slabs，每个slab只负责一定大小范围内的数据存储。每个slab只存储大于其上一个slab的size并小于或者等于自己最大size的数据。例如：slab 3只存储大小介于137 到 224 bytes的数据。如果一个数据大小为230byte的数据进行存储，它将被分配到slab 4中。

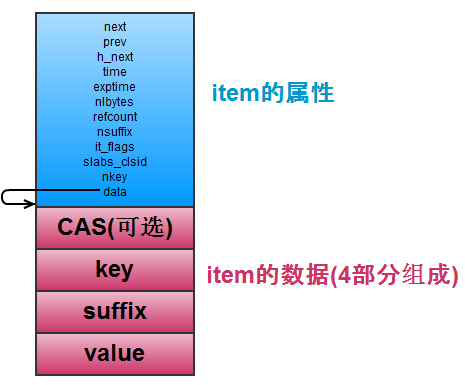
**Chunk是实际用来存储数据的内存空间**

Chunk是一系列固定的内存空间，这个大小就是管理它的slab的最大存放大小。例如：slab 1的所有chunk都是104byte，而slab 4的所有chunk都是280byte。因为chunk的大小固定为slab能够存放的数据(iterm)的最大值，所以所有满足分配给当前slab的数据都可以被chunk存下。举例来说，如果chunk size是128byte，而存储的数据只有100byte，剩下的28byte将被闲置。此外，memcached允许配置的最小的chunk空间为48个字节（key+value+flags），通过-n参数可以调节这个数值。









Memcached源码分析之内存管理篇

<http://kenby.iteye.com/blog/1423989>

memcached源码剖析之内存管理

<http://www.cozyshu.com/memcached-slabs/>

简单的实现内存池管理器

<http://blog.chinaunix.net/uid-21457204-id-2183215.html>