# 内存基础

## 基本概念

### 内存在计算机中的位置

内存是计算机中重要的组成部分，它通过内存总线与其他设备项链，内存本身是整个存储系统的一部分，访问速度比磁盘快几个数量级，可以视为磁盘到CPU之间的缓存盘。

**存储器的层次结构：**



注：在寄存器之上还有运算速度更快的ALU计算单元。

**CPU到内存计算时间：**



**多核CPU缓存：**



注：单核独享L1、L2，多核共享L3，多个槽共享内存。

**超线程：**一个ALU对应多个PC或Register，所谓的四核八线程（比如一个核有一个ALU，但是有两组Register+PC，这样ALU就可以分时计算两个组的数据，这比存储上下文然后恢复现场要快）。

缓存行（cache line，64字节）：



MESI cache一致性协议：



### 物理地址

通常计算机使用大容量的、单位容量价格较为低廉的DRAM作为内存。内存的最小单位为bit，不过访问内存时一般以8bit也就是字节为单位。硬件层会将计算机所有内存条抽象为一个连续的字节数组，内存条上的每一个字节大小的存储位置将对应于这个数组中的一个元素，该元素的数组下标称为这个字节的物理地址（Physical Address，简称PA）。

### 虚拟地址

现代处理器中往往不再使用直接物理寻址的方式访问内存，而通过所谓的虚拟地址VA来访问内存。在CPU芯片中增加了一个叫内存管理单元（MMU=Memory Manager Unit）的部分用于地址翻译。

CPU将要访问的内存的虚拟地址发给MMU，MMU翻译为物理地址后请求内存，内存返回实际数据给CPU。

无论是编译型还是解释型的应用程序，其指令和静态数据都会被编码为一定的内存地址，这些地址在程序运行之前即已确定。当直接应用程序使用物理地址时，多个程序中的相同地址对应相同的内存块，在这些程序交替执行时，将不得不反复的替换物理内存中的内容。有了虚拟地址后，操作系统可以将不同程序的相同的虚拟地址映射到不同的物理地址上，这样大大提升了程序的执行效率。同时操作系统还可以对不同的应用程序进行有效的隔离，提高了安全性。

### 进程虚拟地址空间

每个进程都有自己的虚拟地址空间，如下图所示，两个不同的进程A、B的地址空间内，各自只使用了部分地址，它们的实际地址可以映射到物理内存中的不同的段。注意进程的虚拟地址空间可以比物理内存大，比如1G物理内存可以支持4G的虚拟地址空间，系统可以将暂时不用的内存交换到磁盘上。



用户进程通常情况只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址

内核空间是由内核负责映射，不会跟着进程变化；内核空间地址有自己对应的页表，用户进程各自有不同额页表

### 指令集架构的特权级

现代指令集架构一般都包含多个特权等级，每个等级下可以执行的指令集和范围不同。以IntelX86为例，Ring3为最低特权级，只能执行受限制令，Ring0为最高特权级，可以执行所有指令。



说明：图中的Ring1和Ring2是Intel预期保留给设备驱动程序的，但在Intel系统实现时，由于只使用Ring0和Ring3，所以设备驱动程序实际在Ring0执行。

### 用户态和内核态

操作系统可以基于特权控制应用程序对各种资源包括内存的访问。对IntelX86，Linux只使用Ring0和Ring3两个特权级，Ring0对应内核态，Ring3对应用户态。用户态一般不能直接访问系统的资源，也不能直接访问操作系统内核的代码与数据，而需要通过如系统调用的方式，切换为内核态之后，由操作系统去访问具体的资源或实现某些功能。很多系统调用常由库函数包装后提供给应用程序使用。

### 虚拟地址的划分

虚拟地址空间被划分为内核空间和用户空间。

当进程处于内核态时（通常是发生系统调用时，此时执行的是内核和设备驱动代码）可以访问内核空间和用户空间的所有地址。当进程处于内核态时，只能访问用户空间的内存地址。

下图是32位Linux系统的典型地址划分，用户空间：内核空间=3:1。



对于64位系统，2^64非常巨大，实际用不到那么大的虚拟地址空间。Linux一般使用42位或47位地址空间，而此时内核空间与用户空间也不是连续的，从0x0开始的128T虚拟地址空间为用户空间，从而虚拟地址空间顶部开始向下的128T空间为内核空间。

Linux上用轻量级进程实现线程，因此实际上每个线程都可以各自处于用户态或内核态。

## 页与巨页

### 页

在使用虚拟地址空间时，虚拟地址不等于物理地址，需要两者建立映射关系，而这种映射关系也需要被保存在内存中以供MMU查询。显然，按字节来建立映射关系，整个内存都不够用。因此按照一定大小的内存块管理内存，并以块为单位来建立映射关系，就成为一种合理的选择。硬件系统本身也支持这种管理方式。

页：一定大小的内存块。操作系统以页为单位来管理虚拟地址空间和物理内存，在Linux上一页一般为4K大小。

### 页表

虚拟页与物理页可以通过两个页的首地址的映射关系来关联，这种映射关系的记录保存在所谓的页表中，因此被称为页表项。页表使用树状的结构来保存所有的页表项，以方便页表项的快速查找。页表的详细组织结构以及页表项的查找可以参考操作系统相关书籍。

每个进程都有自己的页表来记录自己的虚拟页的映射关系，页表并不需要记录该进程整个虚拟地址空间的页映射情况，只需要记录已分配的部分即可，这样可以节省大量内存空间。

### TLB（Translation Lookaside Buffer）

TLB：旁路转换缓存，是继承在CPU芯片上的一个内存管理单元，用于缓存页表项。

在引入TLB之前，访问一个内存单元需要两步。首先要从内存中的页表查询页表项，然后根据页表项再访问与虚拟地址对应的物理地址。这样需要访问内存两次，效率比较低。在引入TLB之后，有了页表项缓存，可以提高内存访问效率。



问题：MMU本身是做地址翻译的，那么MMU是怎么知道内存中的页表在哪里呢？

回答：这个地方需要操作系统与硬件相配合。操作系统可以通过指令告知MMU页表的实际物理地址，MMU直接通过物理地址访问页表。

备注：如果MMU在TLB中查到映射关系，还是需要通过内存中的页表去查询的。

### 巨页

引入TLB后，内存访问效率有了提升。但现代的操作系统上运行的进程很多，所有进程的页表项数目非常可观，而TLB本身的空间有限，只能缓存所有页表项中较少的一部分，缓存命中率不高。为了提升TLB命中率，Linux操作系统采用了巨页技术，巨页大小为M量级甚至G量级，典型大小为2M或1G。一个2M的巨页等于512个4K页，原来需要512个页表项记录映射关系，现在仅需1个页表项，页表项总数大大缩减，TLB的缓存数/页表项总数的比值提高了，命中率也就提高了。

### 使用巨页操作系统

巨页可以由用户手动设置或由操作系统自行管理，手动设定可以用如下命令指定巨页文件系统：

mount –t hugetlbfs –o uid=<value>,gid=<value>,mode=<value>,size=<value>,

nr\_inodes=<value> none <mount\_path>

需要注意的是，该文件系统上不能使用常用的read/write访问文件，只能使用mmap即内存映射的模式或共享内存方式（shmget、shmat，最终仍然调用mmap）来访问。

命令参数中：uid和gid为指定用户的id和组id，mode为读写执行模式，size为要使用的巨页文件大小，nr\_inodes为该文件系统最大inode数，限制文件及文件夹的数目，mount\_path为挂载路径。

### 透明巨页

由操作系统自行管理的巨页称为透明巨页（THP，Transparent Huge Page）。开启了透明巨页后，操作系统会根据内存使用情况判断是都需要将若干较小的页（比如4K的页）合并为较大的页（比如2M的页）进行管理。

使用以下命令可以查询是否启用了THP：

cat /sys/kernel/mm/transparent\_hugepage/enabled

always：尽量使用；never：禁用；madvise：避免内存改变。

启用/禁用透明巨页：

1、开机启动设定

可以通过修改grub.cfg实现，在启动项中增加：

transparent\_hugepage=[always|never|madvise]

2、临时设置，关机或重启后失效

echo [always|never|madvise] > sys/kernel/mm/transparent\_hugepage/enabled

## swap分区

### swap空间

Linux可以借助磁盘空间暂时保存内存中的内容，来实现虚拟地址空间大于物理内存，这部分磁盘空间就称为swap空间。Swap空间可以类比于windows系统上的虚拟内存。

查看swap空间信息的方法：

cat /proc/swaps

swapon –s

### 创建和使用swap

使用swap的相关命令：

mkswap <swap\_path>：在文件和设备上建立swap空间。

swapon/swapoff <swap\_path>：启动/关闭swap空间。

swapon/swapoff –a：启动/关闭所有swap空间。

启动时自动挂载swap空间，需要在/tec/fstab中增加：

<swap\_path> swap swap default 0 0

### 内存换出

Linux通过内核线程kswapd来周期性检测内存使用情况，并执行将内存移出到swap的操作。内存的移出以页为单位，所以也称为内存页换出。选择哪些页被换出基于LRU原则，即最近最少使用的页被优先换出。内核中设定了high、low、min三个内存水位线，低于high表示有一定压力，此时就有可能触发内存换出，低于low表示压力很大，低于min表示内存非常紧张。High和low基于min计算，min水位线可用以下命令查询：

cat /proc/sys/vm/min\_free\_kbytes

### 内存换入

如果进程要访问的内存页已经被移出到swap空间中，而不在物理内存中，那么进程对该页的访问会触发缺页异常。缺页异常会被内存捕获，内核检查发现内存页在swap中，则会发出调页请求，从swap中将页载入到物理内存中，然后进程可继续正常运行。



### Linux释放内存的机制

Linux并非只有一种释放内存的机制。由于Linux倾向于尽量使用内存作为缓存使用，因此实际上内存中还会有文件缓存。在内存不足时，Linux也可以通过清除文件缓存的方式来释放内存。

swappiness值（0~100）指明了系统在释放内存时的倾向，0表示尽量不使用swap方式而通过释放文件缓存来缓解内存压力，100则表示以同样优先级使用两种方式，该值可使用cat /proc/sys/vm/swapiness查看。如果仍无法释放足够的内存，那么系统会选择杀死一些进程来释放内存。

杀死进程指的是发生OMM（Out Of Memory）时，系统按照一定的准则对各个用户进程的内存使用情况计算一个数值，按照这个数值排序，按顺序杀死进程，直至有足够的内存可以使用。如果出现进程莫名其妙退出的情况，可能就是由于系统OOM被杀死了，查看是否发生了OMM，可检查/var/log/message文件或者dmesg命令的输出中是否有Out Of Memery及kill的字样。

## NUMA

### 术语

core：物理核，基本的逻辑执行单元

CPU：指集成了多个core的物理芯片

Socket：主板上的CPU芯片槽位

Thread：这里不是指软件线程而是硬件线程。基于超线程技术，一个core可以支持多个指令并发执行，可认为 core上有多个硬件线程，每个硬件线程在系统中体现为一个逻辑核

Processor：处理器，有时候指core，有时候指CPU，依上下文语境而定

Socket就是常说的“几路几核”服务器找那个的“路”

CPU情况可以查看系统的/proc/cpuinfo信息获取，其中显示的每一个processor一般指thread，即逻辑core。

### UMA

UMA（uniform memory access），即一致性内存访问。在UMA架构下，所有CPU对所有内存的访问时间相同。UMA架构在CPU数量少的时候，性能非常好，但随着CPU数目增加，各CPU访问内存时对内存总线的争用回越来越剧烈，从而导致性能下降。因此，从长期发展来看，UMA架构越来越无法适应多核时代的要求。



### NUMA

NUMA（non-uniform memory access），即非一致性内存访问。在NUMA架构下，每个CPU都有自己的本地内存，如下图所示，紫框内为多个内存条组成的CPU本地内存，红线则为本地内存访问，其访问速度比较快。而如果一个CPU上的核要访问其他CPU的本地内存（远端内存），则需要通过图中的蓝线所示的CPU互联模块来进行，速度会明显慢于本地内存访问。



### 查看NUMA信息

#### numactl

在NUMA架构下，拥有本地内存的CPU称为node。使用numactl –hardware命令可以看到基本的NUMA信息，其中会显示node上的所有逻辑核编号以及本地内存大小、使用量等信息。

最下方的node distances显示了各个node访问本地和远端内存的时间代价的相对值。

#### numastat

numastat –c命令可以查看系统的NUMA使用情况，其中NUMA hit表示访问本地内存的总量，NUMA Miss表示访问远端内存的总量。

numastat –c <pro\_name>可以查看进程内存在node上的分布：

### 启用NUMA自平衡

由于NUMA架构下访问本地内存和远端内存的差异，将进程的内存集中在执行该进程命令的逻辑核所在的node上，显然可以提高进程的性能。系统提供了numad命令来对进程的内存做调解，它会逐步将进程使用的逻辑核与内存集中在相同的node上以提高性能。

numad：系统自动执行NUMA自平衡

numad –p <pid>：对指定的进程执行NUMA自平衡

nomad –r <pid>：对指定的进程取消NUMA自平衡

### NUMA的影响

一般情况下，使用NUMA自平衡可以提高性能，但在特殊情况下，反而会导致性能大幅下降，典型的如数据库场景，数据库进程往往需要大量内存，当启动NUMA自平衡后，这些内存集中在同一个node。当数据库需要更多的内存时，由于Linux的策略原因，会优先将内存置换到swap空间上，而不是在其他node上分配空间，这样就会导致大量的内存换页，造成数据库性能大幅降低甚至卡死。因此在使用NUMA自平衡策略前，需要对进程的内存行为以及系统的策略设置有充分了解。

关于NUMA对数据库的影响，参见：

http://cenalulu.github.io/linux/numa

## 内存统计信息

系统中内存使用情况：/proc/meminfo

进程的内存使用情况：/proc/28040/status

查询内存总使用率：free

查询进程 cpu 和内存使用占比：top

虚拟内存统计：vmstat

进程消耗内存占比和排序：ps aux –sort -rss

释放系统内存缓存：

/proc/sys/vm/drop\_caches

To free pagecache, use echo 1 > /proc/sys/vm/drop\_caches

To free dentries and inodes, use echo 2 > /proc/sys/vm/drop\_caches

To free pagecache, dentries and inodes, use echo 3 >/proc/sys/vm/drop\_caches

### 内存指标(/proc/meminfo)

/proc/meminfo包含了系统内存的统计信息，可以使用cat 命令查看其中的内容。

MemTotal：物理内存总量

MemFree：空闲物理内存

MemAvailable：可用物理内存，注意可用物理内存不等于空间物理内存，而是大致上等于MemFree+Cached

Cached：用来缓存文件数据的物理内存大小，内存不足时可以释放这部分来缓解内存压力

Buffer：用来缓存块设备数据占用的物理内存大小，这部分占用一般比较小

SwapCached：内存中和swap空间中都存在的内存大小，在内存紧张时这部分内存无需换出到swap空间，可直接释放

Active：物理内存中较活跃的部分，即最近被使用过的部分

Inactive：物理内存中不活跃的部分，即最近未被访问过的部分

Active+Inactive：一般略小于MemTotal-MemFree

Active(file)：活跃内存中属于文件缓存的部分

Inactive(file)：不活跃内存中术语文件缓存的部分，有内存压力时优先释放的部分

Active(anon)：活跃内存中术语各进程使用的部分

Inactive(anon)：不活跃内存中术语各进程使用的部分

Active=Active(file)+Active(anon)

Inactive=Inactive(file)+Inactive(anon)

Cache大致等于Active(file)+Active(anon)

SwapTotal：swap空间总量

SwapFree：swap空间的空闲大小，前面的SwapCached也算SwapFree

Dirty：脏页大小。Linux处理I/O时一般不直接写入磁盘，而先缓存到内存中，后续再集中写入。这部分缓存可能包括新写入的数据，也有可能包括从磁盘中读入，但已做修改的数据，Linux中称这种数据为“脏”的，而数据又是以页为单位管理的，因而这些数据被称为脏页。如果机器突然断电，那么脏页中的数据将因为未及时写入磁盘而丢失。

/proc/meminfo中各项指标可以用来对系统的内存情况进行判断，比如MemFree很小但MemAvailabel很大，那说明再启动新的进程可能需要释放文件缓存或交换内存到swap空间来空出一部分内存，但如果Inactive(file)也比较小，那么说明文件缓存正在被使用，释放文件缓存可能无法空出足够的内存空间，而需要使用swap空间，则性能明显会受到影响。另外，Active(file)和Dirty两项的值比较大，可以从侧面表明系统的I/O比较活跃。

### 内存相关命令

#### top/free

top命令：显示物理内存、swap空间、Cached、Buffer使用情况，另外还有进程内存使用情况

free命令

free –m/g：以M/G为单位显示内存使用信息

free –h：以可读性最佳的方式显示内存使用信息

free –s <N>：以N秒为间隔不断获取内存使用信息

free –c <N>：显示N次信息后退出

#### vmstat

vmstate –s：显示类似于/proc/meminfo中的主要内存指标，另外还显示一些CPU统计信息

vmstate <N>：以N秒为间隔显示MemFree、SwapFree、Buffers、Cached、si、so信息

vmstate –a <N>：以N秒为间隔显示MemFree、SwapFree、Active、Inactive、si、so信息

其中，si、so分别是每秒从swap换入内存的数据量和每秒从内存换出到swap的数据量

#### sar

sar命令可以显示CPU、内存、网络、磁盘等方面统计信息，是Linux上监控系统使用状况的最强大的命令之一。

sar <N>：以N秒为间隔不断显示统计信息，如果不携带N这个参数，那么会显示从当日0点开始至当前时间以10分钟为粒度的统计信息

sar –r：显示与/proc/meminfo类似的主要内存指标

sar –R：显示物理内存、Buffers、Cached单位时间内变化情况，单位为页/s

sar –B：以页为单位统计了内存与磁盘的交互、缺页异常发生情况、kswapd扫描情况等多种信息

sar –H：显示巨页的统计信息

sar –S：显示swap的使用情况

sar –W：显示单位时间内swap换页情况

另外，系统的/var/log/sa目录下以sar开头的文件记录了每天的系统性能统计信息，每一类统计信息均以10分钟为粒度，可以在排查问题时追溯当时的系统运行状况。

sar命令包含在sysstet工具中。

#### nmon

nmon是一款IBM开发的AIX/Linux系统性能监控工具，它的功能与前面的sar功能上比较接近。

Nmon提供两种监控方式。一种与top类似，动态刷新性能数据，并提供不同的按键以选择具体的监控项，内存项监控为M；一种是通过设置命令行参数，将需要统计的信息输出到文件中。对包含输出信息的文件，可以使用nmon analyser工具将统计数据导入到excel中，以生成形象化的报表。

# 进程内存布局

## 应用程序内存布局

### 内存分段

应用程序的内存分为多段，一般包括：

text段：保存二进制形式的应用程序可执行代码

data段：保存代码中在定义时就被初始化的全局/静态变量

bss段：保存代码中在定义时未被初始化的全局/静态变量

stack段：进程栈区域，代码执行时使用

heap段：进程的堆，动态分配内存时使用

内存映射区：可用于动态库、文件映射以及动态内存分配

### 32位布局

Linux的经典内存布局（早期使用）如下所示，进程的内存地址空间一般为4G，上部为内核空间，一般为1G，下部为用户空间，一般为3G。从0地址开始有一段保留地址，然后依次是text、data、bss段，堆中bss顶部开始，向高地址增长。内存映射区从映射区基地址开始，向高地址增长。栈从用户空间顶部开始，向低地址增长。



现代的32位Linux布局与经典布局不同，主要有如下两点：

内存映射区改为从栈顶部开始向低地址增长，即和堆相向增长，这样可以最大限度的利用中间的空闲地址空间。

在栈、内存映射区和堆的开始处均增加了一段随机大小的偏移，以防御和堆溢出这样的攻击行为。

### 64位局部

64位Linux的布局与32位有些不同。如下图所示，在64位场景下，内核空间与用户空间不再是连续的了，而是由一大块未定义区域隔开，两者各有128T的地址空间。在这种情况下，地址空间一般都很充裕，而且不同段距离很远，无需像现代的32位布局那样保留随机间隔，因而64位用户空间的内存布局实际与32位经典布局类似。



### 申请内存方式

#### brk

进程有一个称为program break的指针指向堆的顶部的地址，通过系统函数brk可以改变这个指针的位置，从而改变堆的大小（堆可以扩大也可以缩小）。这种申请内存的方法存在一个缺陷，即如果堆顶部分的内存一直在使用，就算其他部分未被使用，堆页无法收缩，这可能会造成隐形的内存泄露。



#### mmap

mmap函数通常被应用程序用来将文件映射到内存中，其实它也可以用来向系统申请新的内存映射区域作为动态内存来使用。

与堆不同的是，mmap可以在内存映射区中寻找空闲的合适大小的区域来分配，应用程序可以多次调用从而申请多个内存映射区，每一块映射的内存只要不再使用，都可以单独释放掉，不像堆那样受到堆顶内存的影响。

mmap申请的内存大小以页为单位。

## 进程内存段分析

### 内存信息

/proc/<pids>/maps包含进程内存使用信息，每一行包含：

01b68000-01b69000：虚拟地址段的地址范围

rwxp：rwx读、写、执行权限；p/s：私有/共享

01568000：可执行文件/共享库文件不同段的偏移量

fd:00：可执行文件/共享文件主次设备号，其余为00:00

71778881：可执行文件/共享库文件的inode号

/opt/.zxve/\*\*\*：可执行文件名/共享库名/段名

### text段

text段如下所示，text有可执行权限，即x属性：



可以通过objdump命令来查看可执行文件的反编译信息：

objdump –D <exec\_path>

在输出信息中，可以看到<\_FRAME\_END\_>，这个就是text段的结束位置，与maps中的结束位置可以匹配上：



### data段和bss段

类似于text段，可以通过nm命令来获取可执行程序的符号表信息，来确定data段和bss段的一些信息：

nm <exec\_path> | sort

从nm的输出可以看到data段和bss段的起始地址和整个数据段的截止地址，从而确定两段的对应地址范围。

nm命令后面加上sort是为了按照虚拟地址排序，分析起来更加方便。

关于data段和bss段：

bss段的起始位置实际在data段的地址范围内，这个是由于maps中是以页（一般为4K）为单位来显示的。

有的变量已经初始化了，但却在bss段中。

全局/静态变量初始化时如果只由0或NULL或’\0’赋值得到，那么仍然认为是未初始化的，以下全局变量均在bss段：



### libc段

在text段和data段之间还有一小段内存地址是gcc编译时插入的。main函数并不是程序的真正入口，在main函数执行前和执行后libc都做了大量工作，包括程序的初始化与终结，这一小段就是供libc使用的。

nm的输出中，\_FRAME\_END\_和\_GLOBAL\_OFFSET\_TABLE\_之间的就是libc相关的段：



### 内存映射区

在/proc/<pid>/maps里面可以看到类似于如下的段，这些就是进程内存映射区中动态库占用的内存部分。

与应用程序类似，动态库页包含text段、data段、bss段，在下图中，属性为rxp的是动态库的text段，而属性为rwp的则是动态库的data段和bss段，属性为rp的为libc使用的段。



### 动态库

#### 映射方式

动态库一般会被多个进程使用，而且在每个进程中，动态库被加载的虚拟地址一般也不相同。但实际上动态库只需要在物理内存中放置一份，不同进程中不同虚拟地址的动态库都被映射到这同一块物理内存，从而大大减少了内存占用，真正实现共享。



#### 动态库数据修改

动态库的数据段是可能被进程修改的，在这种情况下，系统会使用所谓的COW（copy-on-write）技术，将数据段部分在物理内存中再拷贝一份，然后将进程内该动态库的虚拟地址映射到新拷贝出来的物理内存，然后在新的这块物理内存上进行修改，这样将不影响其他进程使用动态库。



#### 注意点

从前面所述动态库的特点，可以得到如下注意点：

由于修改时新建数据段拷贝，因此不可以通过动态库中的全局变量，在进程间传递数据。因为修改后两个进程中的动态库的全局变量已指向不同的物理地址。

由于代码段被共用，因此如果出现某动态库S正在被进程A使用，然后动态库S的文件被替换为新版本，再启动一个使用S的进程B，那么当A下次调用S中的函数时，就会崩溃，因此此时S已被系统重新加载，原来函数地址的内容已变。

第二种情况是实际发生过的，在某软件升级过程设计中，没有注意另一升级相关的进程引用了被替换的动态库，导致替换动态库后该进程崩溃，软件升级被中断。

### 栈

#### /proc/<pid>/maps

在/proc/<pid>/maps的末尾可以看到标注为[stack]的内存区域，这个与标准的布局一致。但在maps的中间段，还有标记为[stack:<number>]的区域，这是怎么回事？

#### C程序栈

我们在maps底部看到的[stack]实际上只是进程的主线程的栈。进程的其他线程都是POSIX线程，使用pthread\_create创建。POSIX线程栈由glibc中的相应代码来实现，栈空间则通过mmap函数在内存映射区创建，如下图所示：



这就是maps信息中间出现的[stack:<number>]来源。线程栈的默认大小一般为8M或者10M，依Linux发行版不同而有所区别。

较早的Linux不会在maps中显示线程栈信息。

#### 查看C程序栈

pstack命令可以查看进程的栈（包括线程），用法为：

pstack <pid>

另一个可以查看进程堆栈的命令为gstack，用法与pstack一样，实际上Linux的pstack一般为指向gstack的一个软链接。

pstack和gstack对Java程序也有效，只是输出内容包含的是native层的栈信息，不包含Java层的信息。

pstack和gstack在排查进程死锁等问题时非常有效。

有时候，线程本身已经挂死，pstack和gstack查看进程栈时也会挂住，这时可以通过proc文件系统来查看进程栈信息：

cat /proc/<pid>/stack

cat /proc/<pid>/task/\*/stack

task下包含了进程中的所有线程，上面的命令可以查到所有线程的栈信息。不过需要注意的是，这个方法只能显示内核栈，不包含用户栈。此方法在排查进程因系统调用（比如I/O）挂死时很有效。

#### Java程序栈

对于Java程序，除了启动时的主线程之外，其他线程栈都由jvm来实现，而jvm则基于glibc的栈来实现Java线程的栈，所以Java程序的栈也在内部映射区。Linux上Hotspot VM对普通Java程序设置的默认栈大小为1M，如果想要自行指定Java线程栈的大小，可以通过-Xss或-XX:ThreadStackSize参数进行设置。

实际上普通Java线程的栈只有1016K，因为1M的栈空间中会被jvm分出分别称为red page和yellow page的隔离区，这两个区域均为一页大小，所以要扣除8K。另外可以观察到，栈的隔离空间总共为12K，包含了glibc的隔离区和jvm的隔离区。

#### 查看Java程序栈

之前说过pstack命令对Java程序只能获得native层的栈信息，Java层的栈信息要使用jstack命令获取：

jstack <pid>

jstack输出中的nid对应于pstack输出中的LWP（Light Weight Process）值，可以据此将同一个线程的Java栈与native栈联系起来。

按照之前所说的内存布局，堆就在静态区之上。然而事实上，也可能如下图一样，与text段、静态区一起放在内存映射区之上。操作系统的实现实际上是比较灵活的。

另外，还可以使用pmap命令来查看maps中的信息：

pmap <pid>

### 堆

Java程序中有堆和非堆（方法区，又称永久代）的说法，这个说法其实是从jvm的角度来说的。从操作系统进程的角度来说，Java的堆和非堆都在进程的堆上，tomcat的永久代和堆共1.5G（anon），在pmap输出中则为完整的一块内存。

### 其他内存段

在/proc/<pid>/maps的末尾，可以看到名为vsyscall和vdso的两个段。Vsyscall是早期为了提升系统调用的效率而引入的，应用程序可以通过访问vsyscall段的内存地址来触发系统调用，而避免通过软中断触发系统调用的低效率。Vdso则提供了与时间相关的几个系统调用的快速实现。系统将时间信息和相关系统调用映射到vdso的段，应用程序通过C库调用该段中的函数，可以避免一般系统调用中用户态与内核态的切换。



# 进程内存使用情况

## 进程的内存分布

进程虚拟地址空间中的页可能处于以下四种状态：

页已分配，且正处于物理内存中

页已分配，已经交换到swap空间

页已分配，但还未填充，比如文件映射时，未读取的部分对应的虚拟内存页

页尚未分配



根据以上的内存分布类型，一般用以下术语来指称分布在不同部分的内存：

驻留内存：指进程当前处于物理内存中部分的大小

虚拟内存：指虚拟地址空间中所有已分配的页的总大小

另外，还有共享内存，这个指多个进程共享的物理内存大小，包括使用共享内存函数指定创建的部分，以及动态库的部分，一般计算共享内存大小时，会按照进程分摊计算。

## 系统对进程内存资源的限制(ulimit)

操作系统对各进程的内存使用有各种显示，使用ulimit –a命令可以查看相关限制信息，其中与内存相关的有：

max locked memory：进程可锁定的最大内存值

max memory size：进程驻留内存的最大值

stack size：进程栈的最大值

virtual memory：进程虚拟内存的最大值

## 单独指定应用程序内存限制(/proc/<pid>/limits)

进程的资源限制可以在/proc/<pid>/limit中查到，其中hand limits是由超级用户设置并内核强制限制的资源上限，soft limits由不同用户自行设置，但不能超出hard limits的范围。Limits中包含与ulimit类似的内存配置。

limits中的限制可以在运行时动态修改，方法为：

echo –n “<resource\_name>=<soft\_limit\_value><hard\_limits\_value>” >/proc/<pid>/limits

## 进程内存使用情况(top)

监控单个进程资源占用可以使用以下命令：

top –p <pid>

其中与内存相关的为：

VIRT：进程当前虚拟内存的大小

RES：进程驻留内存的大小

SHR：进程共享内存大小

%MEM：RES占系统总内存的百分比

其中，RES最为重要。

## 进程内存使用情况

## /proc/<pid>

除了top命令外，还可以使用proc文件系统下的各种信息查看进程内存使用量：

cat /proc/<pid>/status中vm开头的项为内存使用信息，其中VmRSS为驻留内存大小。

cat /proc/<pid>/statm信息比较简略，其中第二项为驻留内存代销，注意statm统计数值指每部分占用的页数。

cat /proc/<pid>/private\_rss仅包含进程驻留内存信息，以页数来统计。

## /proc/<pid>/smaps

/proc/<pids>/smaps中详细记录了maps中各内存段的使用情况，其中：

Size：该段的虚拟地址长度

Rss：实际驻留内存在物理内存中的内存大小

Shared/Private：共享或私有的

Clean/Dirty：是否有过修改

如果一段内存是文件映射使用的，那么dirty表示该段对应的文件内容从磁盘载入后，已经被修改过。

如果一段内存使用作堆栈内存使用的，那么dirty表示该段内存已经分配且在使用。

## Java程序

Java程序的内存与普通程序不同，Java本身提供了很多查看内存的工具：

jstat主要用来查看垃圾回收相关的信息

jmap可以查看java堆中不同区域的使用情况，堆当中不同对象的数目大小等信息，还可以使用以下命令导出堆内存：

jmap –dump:format=b,file=<outfile> <pid>

导出dump文件可以使用jhat、jprofiler等工具进行分析。

## 读取内存/proc/<pid>/mem

/proc/<pid>/mem将进程内存的地址空间抽象为一个文件，内存地址即相对于文件头部的偏移量，这样就可以通过open、lseek、resd等文件I/O函数去读取内存的内容。

需要注意的是，由于程序运行时内存内容一直在改变，通过mem读取内存前要先通过ptrace暂停进程，读取结束后要恢复进程运行。

读取mem可以作为排查内存泄露等问题的辅助手段。

## 读取内存gcore

gcore是gdb包中提供的命令，它可以生成进程的coredump文件，而且在执行结束后进程可以继续正常运行。gcore相当于提供了进程运行时的一个截面信息，生成coredump后再用gdb分析，可以说是排查复杂问题的首选方法。

gcore的使用方法为：

gcore <pid> -o <outputfile>

不过需要注意的是，gcore导出的部分要比驻留内存大，对内存泄露时，进程内存占用很多的情况，最好不要使用gcore。

## 进程的omm值

在/proc/<pid>/omm\*中查看进程的omm值：

omm=out of memory，当操作系统发现可用内存耗尽时，会杀死一些进程以释放内存，每个进程都有omm值，作为操作系统杀死进程时的排序参考。

omm\_score\_adj：在发生omm时，被系统杀死优先级配置，范围为0~1000,0表示不会被杀，1000表示优先被杀。

omm\_score：表示系统对该进程当前的omm估值，数值越大越容易被杀。

# 内存管理接口

## C

### 内存申请

#### malloc

函数原型：

(void \*)malloc(unsigned size); //字节数

说明：

malloc函数在内存中开辟的是一块连续的空间，size是所需要空间的长度，开辟的大小为size \*参数类型，开辟完之后返回这块空间的首地址。

#### calloc

函数原型：

void \* calloc(size\_t numElements, size\_t sizeOfElements);

//(元素的个数,单个元素的字节数)

说明：

和malloc相似，它也是开辟一块连续的空间，空间大小为：元素的个数\*单个元素的字节数。

#### realloc

函数原型：

void\* realloc(void\* ptr, unsigned newsize); //(地址,字节数)

说明：

给一个已经分配地址的指针重新分配空间，参数ptr为原有的空间指针，newsize为重新申请的地址长度。它与malloc的区别就是如果你给的指针是NULL，那么你使用的就是malloc，如果你给出的指针是一个已经分配了地址的指针（ptr），那么你使用的就是realloc。

上述三种申请内存方式的区别：

1. 函数malloc不能初始化所分配的空间，而函数calloc可以初始化。也就是说，如果由malloc函数分配的空间原来没有被分配过，则其中每一位都可能是0；反之，如果这一块数据块原来被分配过，那里面可能遗留着各种各样的数据。所以，当你在使用malloc开辟一块新空间的时候，要重新初始化那一块空间（一般调用memset函数来初始化空间），否则在多次释放开辟之后，可能会出现使用错误。
2. calloc函数会将所分配的内存空间中的每一位都初始化为0（这也是它和malloc的主要不同之一）。也就是说，如果你是为字符类或者整型类的元素分配空间，那么这些元素会保证被初始化为0；如果你是为指针类函数分配内存，那么这些元素都会被初始化为空指针。
3. malloc向系统申请size个字节的空间，申请完之后返回的是这个空间的首地址，类型为void \*，而void\*表示未确定的类型，在C/C++中void\*被强转成任意类型的指针。
4. realloc可以对给定的指针所指向的空间进行扩大或者缩小，无论是扩大还是缩小，原有内存中的内容将保持不变（如果对于缩小之后的空间，被缩小的那部分空间的数据还是会丢失）。realloc并不保证调整后的内存空间和原来的内存空间保持同一个地址。相反，realloc指针很可能指向一个新的地址。
5. realloc是从堆上分配空间的，但当你进行扩大的时候，realloc会试图从堆上现存的数据后面的那些字节中获取附加的字节，如果能满足，就刚好。但如果后面的字节数不够，其就会使用堆上第一个有足够大小的自由块，然后将现存的数据拷贝到新的位置，将老块放回到堆上。在这个过程中，数据会被移动。也就是说，当你使用realloc的时候，数据可能被移动。

### 内存释放

#### free

有分配就一定有释放。malloc对应的就是free函数，free函数只有一个参数，就是要释放的内存块的首地址。

free函数做的事情：斩断指针变量与这块 内存的关系。就像上面的例子中malloc开辟的这一个数据块空间是属于p，你只能通过p来访问这个数据块空间，而free函数做的事情就是斩断malloc和p之间的联系。但是p指针本身存放的地址并没有发生变化，只有它对指针指向的那块内存已经没有所有权了，不能对内存块进行操作。而那块内存块里面的数据也没有被改变，只有你没有办法去访问或者修改那块数据块中的内容中。

malloc和free是一一对应的，如果malloc两次但是只free一次就会存在内存泄露，如果malloc一次但是free两次，就会出错（第一次使用free的时候，malloc所开辟的空间就已经被释放，第二次使用free就无内存空间可以释放了，这种对内存的误操作就有可能会导致程序的崩溃）。

### 内存拷贝

#### memcpy

函数原型：

void\* memcpy(void\* dest, const void\*src,size\_t n);

说明：

从源src所指的内存地址的起始位置开始拷贝n个字节到目标dest所指的内存地址的起始位置中。

这里需要对memcpy这个函数做一些说明，memcpy函数后经常还会有一个malloc函数。

#### memmove

函数原型：

void\* memmove(void\* dest, const void\* src, size\_t count);

说明：

由src所指内存区域复制count个字节到dest所指内存区域。

### 字符串拷贝

#### bcopy

主要作用是将字符串src的前n个字节复制到dest中

### 内存清零

#### memset

函数原型：

void\* memset(void\* s ,int ch, size\_t n);

说明：

该函数是将s中当前位置后面的n个字节（typedef unsigned intsize\_t）的ch替换并返回s。作用是在一段内存块中填充某个指定的值，它是对较大的结构体或数组进行清零的一种最快方法。

## C++

### 内存申请

#### new

我们知道C++是兼容C语言的，那么已经有malloc和free来进行动态内存管理，为什么还需要C++定义new和delete运算符来动态管理内存？

它们的区别和联系：

1. 都是动态管理内存的入口
2. malloc/free是C/C++标准库的函数，new/delete是C++操作符
3. malloc/free只是动态分配/释放内存空间，而new/delete出来分配空间还会调用构造函数和析构函数进行初始化和清理
4. malloc/free需要手动计算类型大小且会返回void\*，new/delett可以自己计算类型的大小，返回对应类型的指针

我们在C++中运行重载的，那么我们也可以重载new和delete（其实new和delete是不同重载了，即使你进行了重载，也只是重载了operator new和operator delete）。

### 内存释放

#### delete

# 零拷贝

## 背景

在写一个服务端程序时（Web Server或者文件服务器），文件下载是一个基本功能。这时候服务端的任务是：将服务端主机磁盘中的文件不做修改地从已连接的socket发出去，我们通常用下面的代码完成：

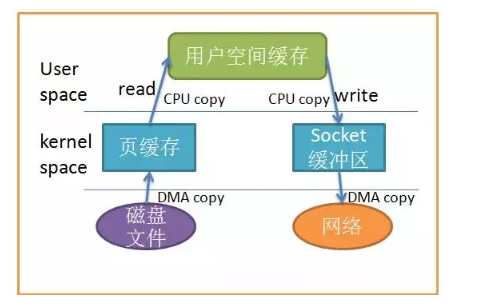
while((n = read(diskfd, buf, BUF\_SIZE)) > 0)

write(sockfd, buf , n);

基本操作就是循环的从磁盘读入文件内容到缓冲区，再将缓冲区的内容发送到socket。但是由于Linux的I/O操作默认是缓冲I/O。这里面主要使用的也就是read和write两个系统调用，我们并不知道操作系统在其中做了什么。实际上在以上I/O操作中，发生了多次的数据拷贝。

当应用程序访问某块数据时，操作系统首先会检查，是不是最近访问过此文件，文件内容是否缓存在内核缓冲区，如果是，操作系统则直接根据read系统调用提供的buf地址，将内核缓冲区的内容拷贝到buf所指定的用户空间缓冲区中去。如果不是，操作系统则首先将磁盘上的数据拷贝的内核缓冲区，这一步目前主要依靠DMA来传输，然后再把内核缓冲区上的内容拷贝到用户缓冲区中。

接下来，write系统调用再把用户缓冲区的内容拷贝到网络堆栈相关的内核缓冲区中，最后socket再把内核缓冲区的内容发送到网卡上。



数据拷贝

从上图中可以看出，共产生了四次数据拷贝，即使使用了DMA来处理了与硬件的通讯，CPU仍然需要处理两次数据拷贝，与此同时，在用户态与内核态也发生了多次上下文切换，无疑也加重了CPU负担。

在此过程中，我们没有对文件内容做任何修改，那么在内核空间和用户空间来回拷贝数据无疑就是一种浪费，而零拷贝主要就是为了解决这种低效性。

## 概述

零拷贝主要的任务就是避免CPU将数据从一块存储拷贝到另外一块存储，主要就是利用各种零拷贝技术，避免让CPU做大量的数据拷贝任务，减少不必要的拷贝，或者让别的组件来做这一类简单的数据传输任务，让CPU解脱出来专注于别的任务。这样就可以让系统资源的利用更加有效。

我们如何减少数据拷贝的次数呢？一个很明显的着力点就是减少数据在内核空间和用户空间来回拷贝，这也引入了零拷贝的一个类型：让数据传输不需要经过 user space。

## 实现

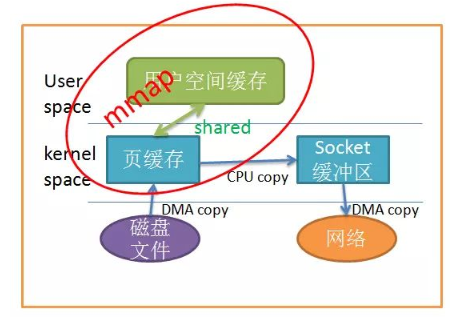
### mmap

我们减少拷贝次数的一种方法是调用mmap()来代替read调用：

buf = mmap(diskfd, len);

write(sockfd, buf, len);

应用程序调用mmap()，磁盘上的数据会通过DMA被拷贝的内核缓冲区，接着操作系统会把这段内核缓冲区与应用程序共享，这样就不需要把内核缓冲区的内容往用户空间拷贝。应用程序再调用write(),操作系统直接将内核缓冲区的内容拷贝到socket缓冲区中，这一切都发生在内核态，最后，socket缓冲区再把数据发到网卡去。同样的，看图很简单：



使用mmap替代read很明显减少了一次拷贝，当拷贝数据量很大时，无疑提升了效率。但是使用mmap是有代价的。当你使用mmap时，你可能会遇到一些隐藏的陷阱。例如，当你的程序map了一个文件，但是当这个文件被另一个进程截断(truncate)时, write系统调用会因为访问非法地址而被SIGBUS信号终止。SIGBUS信号默认会杀死你的进程并产生一个coredump,如果你的服务器这样被中止了，那会产生一笔损失。

通常我们使用以下解决方案避免这种问题：

1、为SIGBUS信号建立信号处理程序

当遇到SIGBUS信号时，信号处理程序简单地返回，write系统调用在被中断之前会返回已经写入的字节数，并且errno会被设置成success,但是这是一种糟糕的处理办法，因为你并没有解决问题的实质核心。

2、使用文件租借锁

通常我们使用这种方法，在文件描述符上使用租借锁，我们为文件向内核申请一个租借锁，当其它进程想要截断这个文件时，内核会向我们发送一个实时的RTSIGNALLEASE信号，告诉我们内核正在破坏你加持在文件上的读写锁。这样在程序访问非法内存并且被SIGBUS杀死之前，你的write系统调用会被中断。write会返回已经写入的字节数，并且置errno为success。

我们应该在mmap文件之前加锁，并且在操作完文件后解锁：

if(fcntl(diskfd, F\_SETSIG, RT\_SIGNAL\_LEASE) == -1) {

perror("kernel lease set signal");

return -1;

}

/\* l\_type can be F\_RDLCK F\_WRLCK 加锁\*/

/\* l\_type can be F\_UNLCK 解锁\*/

if(fcntl(diskfd, F\_SETLEASE, l\_type)){

perror("kernel lease set type");

return -1;

}

### sendfile

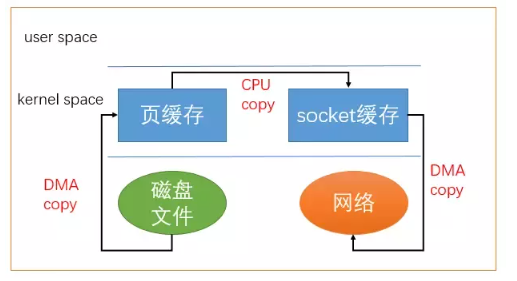
从2.1版内核开始，Linux引入了sendfile来简化操作:

#include <sys/sendfile.h>

ssize\_t sendfile(int out\_fd, int in\_fd, off\_t \*offset, size\_t count);

系统调用sendfile()在代表输入文件的描述符infd和代表输出文件的描述符outfd之间传送文件内容（字节）。描述符outfd必须指向一个套接字，而infd指向的文件必须是可以mmap的。这些局限限制了sendfile的使用，使sendfile只能将数据从文件传递到套接字上，反之则不行。

使用sendfile不仅减少了数据拷贝的次数，还减少了上下文切换，数据传送始终只发生在kernel space。

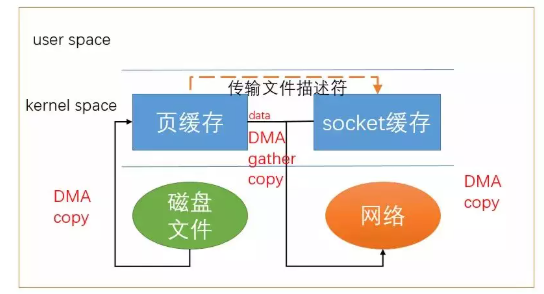


在我们调用sendfile时，如果有其它进程截断了文件会发生什么呢？假设我们没有设置任何信号处理程序，sendfile调用仅仅返回它在被中断之前已经传输的字节数，errno会被置为success。如果我们在调用sendfile之前给文件加了锁，sendfile的行为仍然和之前相同，我们还会收到RTSIGNALLEASE的信号。

目前为止，我们已经减少了数据拷贝的次数了，但是仍然存在一次拷贝，就是页缓存到socket缓存的拷贝。那么能不能把这个拷贝也省略呢？

借助于硬件上的帮助，我们是可以办到的。之前我们是把页缓存的数据拷贝到socket缓存中，实际上，我们仅仅需要把缓冲区描述符传到socket缓冲区，再把数据长度传过去，这样DMA控制器直接将页缓存中的数据打包发送到网络中就可以了。

总结一下，sendfile系统调用利用DMA引擎将文件内容拷贝到内核缓冲区去，然后将带有文件位置和长度信息的缓冲区描述符添加socket缓冲区去，这一步不会将内核中的数据拷贝到socket缓冲区中，DMA引擎会将内核缓冲区的数据拷贝到协议引擎中去，避免了最后一次拷贝。



带DMA的sendfile

不过这一种收集拷贝功能是需要硬件以及驱动程序支持的。

使用splice

sendfile只适用于将数据从文件拷贝到套接字上，限定了它的使用范围。Linux在2.6.17版本引入splice系统调用，用于在两个文件描述符中移动数据：

#define \_GNU\_SOURCE

/\* See feature\_test\_macros(7) \*/

#include <fcntl.h>

ssize\_t splice(int fd\_in, loff\_t \*off\_in, int fd\_out, loff\_t \*off\_out, size\_t len, unsignedint flags);

splice调用在两个文件描述符之间移动数据，而不需要数据在内核空间和用户空间来回拷贝。他从fdin拷贝len长度的数据到fdout，但是有一方必须是管道设备，这也是目前splice的一些局限性。flags参数有以下几种取值：

SPLICEFMOVE ：尝试去移动数据而不是拷贝数据。这仅仅是对内核的一个小提示：如果内核不能从pipe移动数据或者pipe的缓存不是一个整页面，仍然需要拷贝数据。Linux最初的实现有些问题，所以从2.6.21开始这个选项不起作用，后面的Linux版本应该会实现。

SPLICEFNONBLOCK ：splice 操作不会被阻塞。然而，如果文件描述符没有被设置为不可被阻塞方式的 I/O ，那么调用 splice 有可能仍然被阻塞。

SPLICEFMORE：后面的splice调用会有更多的数据。

splice调用利用了Linux提出的管道缓冲区机制， 所以至少一个描述符要为管道。

以上几种零拷贝技术都是减少数据在用户空间和内核空间拷贝技术实现的，但是有些时候，数据必须在用户空间和内核空间之间拷贝。这时候，我们只能针对数据在用户空间和内核空间拷贝的时机上下功夫了。Linux通常利用写时复制(copy on write)来减少系统开销，这个技术又时常称作COW。

## 总结

如果多个程序同时访问同一块数据，那么每个程序都拥有指向这块数据的指针，在每个程序看来，自己都是独立拥有这块数据的，只有当程序需要对数据内容进行修改时，才会把数据内容拷贝到程序自己的应用空间里去，这时候，数据才成为该程序的私有数据。如果程序不需要对数据进行修改，那么永远都不需要拷贝数据到自己的应用空间里。这样就减少了数据的拷贝。

除此之外，还有一些零拷贝技术，比如传统的Linux I/O中加上O\_DIRECT标记可以直接I/O，避免了自动缓存，还有尚未成熟的fbufs技术，本文尚未覆盖所有零拷贝技术，只是介绍常见的一些，如有兴趣，可以自行研究，一般成熟的服务端项目也会自己改造内核中有关I/O的部分，提高自己的数据传输速率。

# 内存故障

## 内存泄漏

### 原因

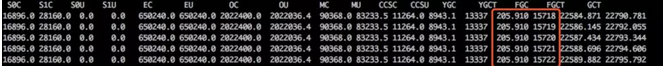
1. 在类的构造函数和析构函数中没有匹配地调用new和delete函数
2. 没有正确地清除嵌套的对象指针
3. 没有将基类的析构函数定义为虚函数
4. 当基类的指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有得到正确释放，因此造成内存泄露
5. 缺少拷贝构造函数，按值传递会调用（拷贝）构造函数，引用传递不会调用
6. 指向对象的指针数组不等同于对象数组，数组中存放的是指向对象的指针，不仅要释放每个对象的空间，还要释放每个指针的空间
7. 缺少重载赋值运算符，也是逐个成员拷贝的方式复制对象，如果这个类的大小是可变的，那么结果就是造成内存泄露

### 工具

首先是 top free df 三连，结果还真发现了些异常：

我们的探测进程CPU占用率特别高，达到了900%。

我们的Java进程，并不做大量CPU运算，正常情况下，CPU应该在100~200% 之间，出现这种CPU飙升的情况，要么走到了死循环，要么就是在做大量的GC。

使用 jstat -gc pid [interval]命令查看了Java进程的GC状态，果然，FULL GC 达到了每秒一次：

这么多的 FULL GC，应该是内存泄漏没跑了，于是使用 jstack pid > jstack.log 保存了线程栈的现场。

使用 jmap -dump:format=b,file=heap.log pid 保存了堆现场，然后重启了探测服务，报警邮件终于停止了。

jstat

jstat 是一个非常强大的 JVM 监控工具，一般用法是：

jstat [-options] pid interval

它支持的查看项有：

class 查看类加载信息。

compile 编译统计信息。

gc 垃圾回收信息。

gcXXX 各区域 GC 的详细信息，如 -gcold。

使用它，对定位 JVM 的内存问题很有帮助。

### 排查问题

#### 分析栈

栈的分析很简单，看一下线程数是不是过多，多数栈都在干嘛：

> grep 'java.lang.Thread.State' jstack.log | wc -l

> 464

才四百多线程，并无异常：

> grep -A 1 'java.lang.Thread.State' jstack.log | grep -v 'java.lang.Thread.State' | sort | uniq -c |sort -n

10 at java.lang.Class.forName0(Native Method)

10 at java.lang.Object.wait(Native Method)

16 at java.lang.ClassLoader.loadClass(ClassLoader.java:404)

44 at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)

344 at sun.misc.Unsafe.park(Native Method)

线程状态好像也无异常，接下来分析堆文件。

#### 下载堆dump文件

堆文件都是一些二进制数据，在命令行查看非常麻烦，Java 为我们提供的工具都是可视化的，Linux 服务器上又没法查看，那么首先要把文件下载到本地。

由于我们设置的堆内存为 4G，所以 dump 出来的堆文件也很大，下载它确实非常费事，不过我们可以先对它进行一次压缩。

gzip 是个功能很强大的压缩命令，特别是我们可以设置 -1~-9 来指定它的压缩级别。

数据越大压缩比率越大，耗时也就越长，推荐使用 -6~7，-9 实在是太慢了，且收益不大，有这个压缩的时间，多出来的文件也下载好了。

#### 使用 MAT 分析 jvm heap

MAT 是分析 Java 堆内存的利器，使用它打开我们的堆文件（将文件后缀改为 .hprof）, 它会提示我们要分析的种类。

对于这次分析，果断选择 memory leak suspect：

从上面的饼图中可以看出，绝大多数堆内存都被同一个内存占用了，再查看堆内存详情，向上层追溯，很快就发现了罪魁祸首。

#### 分析代码

找到内存泄漏的对象了，在项目里全局搜索对象名，它是一个 Bean 对象，然后定位到它的一个类型为 Map 的属性。

这个 Map 根据类型用 ArrayList 存储了每次探测接口响应的结果，每次探测完都塞到 ArrayList 里去分析。

由于 Bean 对象不会被回收，这个属性又没有清除逻辑，所以在服务十来天没有上线重启的情况下，这个 Map 越来越大，直至将内存占满。

内存满了之后，无法再给 HTTP 响应结果分配内存了，所以一直卡在 readLine 那里。而我们那个大量 I/O 的接口报警次数特别多，估计跟响应太大需要更多内存有关。

## C野指针

1. 指针变量没有初始化
2. 指针被 free 或 delete 后，没有设置为 NULL
3. 指针操作超越了变量的作用范围，比如返回指向栈内存的指针就是野指针
4. 访问空指针（需要做空判断）
5. sizeof 无法获取数组的大小
6. 试图修改常量，如：char p='1234';p=\'1\';

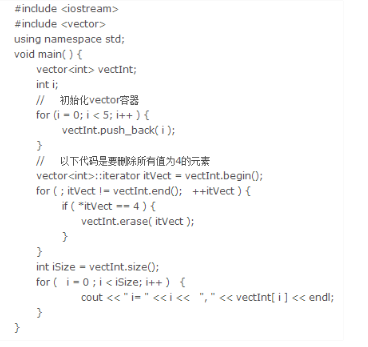
## C资源访问冲突

1. 多线程共享变量没有用 valotile 修饰
2. 多线程访问全局变量未加锁
3. 全局变量仅对单进程有效
4. 多进程写共享内存数据，未做同步处理
5. mmap 内存映射，多进程不安全

## STL迭代器失效

1. 被删除的迭代器失效
2. 添加元素（insert/push\_back 等）、删除元素导致顺序容器迭代器失效

错误示例：删除当前迭代器，迭代器会失效



正确示例：迭代器 erase 时，需保存下一个迭代器



## C++ 11智能指针

1. auto\_ptr 替换为 unique\_ptr
2. 使用 make\_shared 初始化一个 shared\_ptr

weak\_ptr 智能指针助手

（1）原理分析：

（2）数据结构：

（3）使用方法：

a.lock() 获取所管理的对象的强引用指针

b. expired() 检测所管理的对象是否已经释放

c. get() 访问智能指针对象

## C++ 11更小更快更安全

std::atomic 原子数据类型 多线程安全

std::array 定长数组开销比 array 小和 std::vector 不同的是 array 的长度是固定的，不能动态拓展

std::vector vector 瘦身 shrink\_to\_fit()：将 capacity 减少为于 size() 相同的大小

td::forward\_list

forward\_list 是单链表（std::list 是双链表），只需要顺序遍历的场合，forward\_list 能更加节省内存，插入和删除的性能高于 list

std::unordered\_map、std::unordered\_set用 hash 实现的无序的容器，插入、删除和查找的时间复杂度都是 O(1)，在不关注容器内元素顺序的场合，使用 unordered 的容器能获得更高的性能