jvm压缩指针原理以及32g内存压缩指针失效详解

为什么要引入压缩指针(明白的跳过)

先要明白：
32位操作系统可以寻址到多大内存 答：4g 因为 2^32=4 * 1024 * 1024=4g
64位呢？答：近似无穷大

为什么要用64位操作系统 答：因为连你家电脑的内存都不止4g了吧，你用8g的内存在32位电脑上是只有4g有效的，而4g满足不了我们的需求

可是用64位有些那些问题？
答：64位过长，给我们寻址带宽和对象内引用造成了负担

什么负担？往下看！

同一个对象存在堆里会花费更多的空间！！！！

口说无凭，首先我们计算下同一个对象在不同操作系统的堆中存放的大小

下面的东西是一个对象占用的字节数，

对象头
32位系统，占用 8 字节(markWord4字节+kclass4字节)
64位系统，开启 UseCompressedOops(压缩指针)时，占用 12 字节，否则是16字节(markWord8字节+kclass8字节，开启时markWord8字节+kclass4字节)
实例数据
boolean 1
byte 1
short 2
char 2
int 4
float 4
long 8
double 8
引用类型
32位系统占4字节 (因为此引用类型要去方法区中找类信息,所以地址为32位即4字节同理64位是8字节)
64位系统，开启 UseCompressedOops时，占用4字节，否则是8字节
对齐填充
如果对象头+实例数据的值不是8的倍数，那么会补上一些，补够8的倍数

好了开始举例

假设有一个对象

```
class A{
    int a;//基本类型
    B b;//引用类型
}
```

32位操作系统 花费的内存空间为
对象头-8字节 + 实例数据 int类型-4字节 + 引用类型-4字节+补充0字节(16是8的倍数) 16个字节

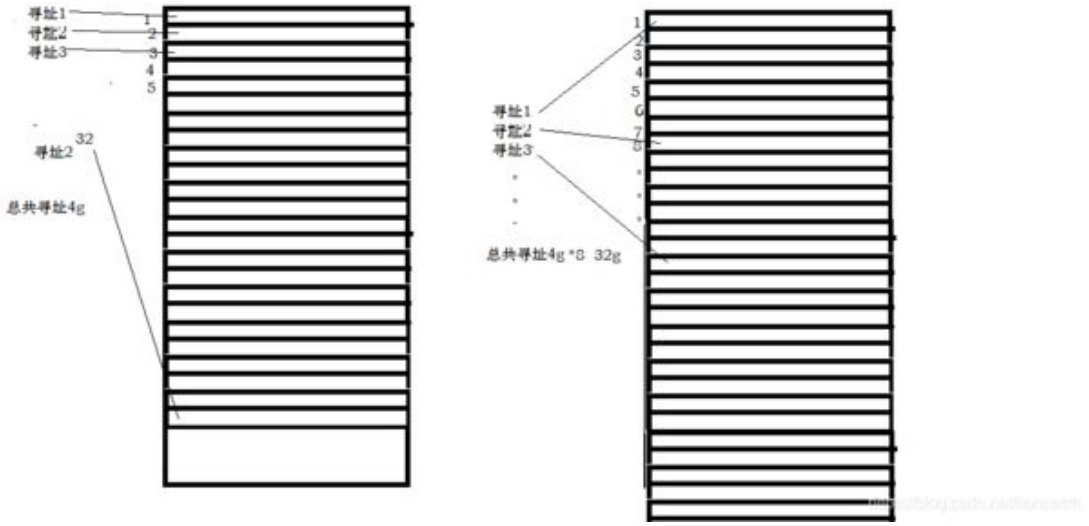
64位操作系统
对象头-16字节 + 实例数据 int类型-4字节 + 引用类型-8字节+补充4字节(28不是8的倍数补充4字节到达32字节) 32个字节

同样的对象需要将近两倍的容量,(实际平均1.5倍),所以需要开启压缩指针：

64位开启压缩指针 **对象头-12字节 + 实例数据 int类型-4字节 + 引用类型-4字节+补充0字节=24个字节**
开启后可以减缓堆空间的压力(同样的内存更不容易发生oom)

压缩指针是怎么实现的

JVM的实现方式是
不再保存所有引用，而是每隔8个字节保存一个引用。例如，原来保存每个引用0、1、2...，现在只保存0、8、16...。因此，指针压缩后，并不是所有引用都保存在堆中，而是以8个字节为间隔保存引用。
在实现上，堆中的引用其实还是按照0x0、0x1、0x2...进行存储。只不过当引用被存入64位的寄存器时，JVM将其左移3位（相当于末尾添加3个0），例如0x0、0x1、0x2...分别被转换为0x0、0x8、0x10。而当从寄存器读出时，JVM又可以右移3位，丢弃末尾的0。（oop在堆中是32位，在寄存器中是35位，2的35次方=32G。也就是说，使用32位，来达到35位oop所能引用的堆内存空间）
仔细看图~ 仔细看图 ~仔细看图



哪些信息会被压缩?
1.对象的全局静态变量(即类属性)
2.对象头信息:64位平台下，原生对象头大小为16字节，压缩后为12字节
3.对象的引用类型:64位平台下，引用类型本身大小为8字节，压缩后为4字节
4.对象数组类型:64位平台下，数组类型本身大小为24字节，压缩后16字节

哪些信息不会被压缩?
1.指向非Heap的对象指针
2.局部变量、传参、返回值、NULL指针

总结:

在JVM中（不管是32位还是64位），对象已经按8字节边界对齐了。对于大部分处理器，这种对齐方案都是最优的。所以，使用压缩的oop并不会带来什么损失，反而提升了性能。

压缩指针32g指针失效问题

讲到这应该很明了了，因为寄存器中3的35次方只能寻址到32g左右(不是准确的32g，有可能在31g就发生指压缩失效)，所以当你的内存超过32g时，jvm就默认停用压缩指针，用64位寻址来操作，这样可以保证能寻址到你的所有内存，但这样所有的对象都会变大，实际上未开启开启后的比较，40g的对象存储个数比不上30g的存储个数