## 引言

Java的对象被jvm管理，单个对象如何布局，大小如何，程序员可以不用关心。

但是，掌握一些相关的知识，可以让我们对应用中使用的对象大小有一个大致的估计，做到心中有数，当遇到内存敏感型应用时，可以通过适当的参数调节和应用优化减少内存占用。

另外，just for fun。

## 声明

以下讨论大部分都是基于32bits Java Hotspot VM，关于64bits的会特殊声明。

## Java对象内存组成

对象内存由以下几部分组成：Object header + primitive fields + reference fields + padding

Object header :

对象头。Hotspot中，一般对象(非数组对象)为8bytes，数组对象为12bytes。

primitive fields :

boolean byte 1

char short 2

int float 4

long double 8

reference fields :

4 bytes

padding :

为了内存对齐的填充字节。

Hotspot中，对象占据的内存是8字节的整数倍。如果不是，则加padding到8字节的整数倍。注意，padding不一定是padding到所有field之后，field有可能重新排序，以满足一定的对齐规则，padding也有可能插入到不同的地方，参见下文的例子。

## shallow size和full size

class A{

int id;

String str;

}

我们计算new A()占据的内存时：

对象头， id(primitive)所占据的内存，str的reference内存，padding。为shallow size。

对象头， id(primitive)所占据的内存， str的reference内存，padding，str实例本身所占据的内存(full size)。为full size。

## 简单普通对象例子

一个new Object()占据8bytes，只有对象头。

shallow size=full size=8

class ObjectWithOneBooleanField {

boolean b;

}

一个new ObjectWithOneBooleanField()占据16bytes.

shallow size=full size=8(对象头)+1(boolean)+7(padding)=16

当一个class定义了8个boolean的field。一个对象还是占据16bytes。这种情况下无padding。

shallow size=full size=8(对象头)+8(boolean\*8)=16。

class A {

boolean val0;

long val1;

int val2;

}

一个new A()占据24bytes.

shallow size=full size=8(对象头)+1(boolean)+8(long)+4(int)+3(padding)=24

## 数组及例子

数组是特殊的对象。对象头为12。4bytes保存length。

多维数组不过就是数组的数组。

一维primitive数组

new byte[0]

shallow size=full size=12(对象头)+4(padding)=16

new byte[4]

shallow size=full size=12(对象头)+4(byte\*4)=16

new byte[5]

shallow size=full size=12(对象头)+5(byte\*5)+7(padding)=24

一维reference数组

new Object[] {}

shallow size=full size=12(对象头)+4(padding)=16

new Object[] { new Object(), new Object() }

shallow size=12(对象头)+8(reference\*2)+4(padding)=24

full size=24(shallow size)+16(2个空object的内存，每个8bytes)=40

二维数组

new Object[][] {

new Object[] { new Object() },

new Object[] { new Object(), new Object() }

}

shallow size=12(对象头)+8(reference\*2)+4(padding)=24

对new Object[] { new Object() }

shallow size=12(对象头)+4(reference)=16

full size=16(shallow size)+8(空对象大小)=24

对new Object[] { new Object(), new Object() }

Full size=40(前文计算过)

合起来 full size=24+24+40=88

## Field重排序

我们来看一个例子。

**class** ClassWithManyFields {

**boolean** val0;

**long** val1;

**int** val2;

**long** val3;

**boolean** val4;

**long** val5;

}

ClassWithManyFields

----------------------------------------------------------------

object class info : allen.memoryutil.dirver.ClassWithManyFields

object identityHashCode : 10851992

in parent info : root object

shallow size = 40

full size = 40

-----------shallow size detail.-----------------

headerType = NormalHeader size = 8

offset : 8 size = 8 long allen.memoryutil.dirver.ClassWithManyFields.val1

offset : 16 size = 8 long allen.memoryutil.dirver.ClassWithManyFields.val3

offset : 24 size = 8 long allen.memoryutil.dirver.ClassWithManyFields.val5

offset : 32 size = 4 int allen.memoryutil.dirver.ClassWithManyFields.val2

offset : 36 size = 1 boolean allen.memoryutil.dirver.ClassWithManyFields.val0

offset : 37 size = 1 boolean allen.memoryutil.dirver.ClassWithManyFields.val4

padding size = 2

-----------end of shallow size detail.----------

----------------------------------------------------------------

可以看出，field被重新排序，以保证K长度的数据类型地址都是K的整数倍。

## 类继承关系对内存布局的影响

**class** Father {

**byte** father\_byte\_0;

**long** father\_long;

**byte** father\_byte\_1;

}

**class** Child **extends** Father {

**byte** child\_byte\_0;

**long** child\_long;

**byte** child\_byte\_1;

}

Child

----------------------------------------------------------------

object class info : allen.memoryutil.dirver.Child

object identityHashCode : 15244180

in parent info : root object

shallow size = 32

full size = 32

-----------shallow size detail.-----------------

headerType = NormalHeader size = 8

offset : 8 size = 8 long allen.memoryutil.dirver.Father.father\_long

offset : 16 size = 1 byte allen.memoryutil.dirver.Father.father\_byte\_0

offset : 17 size = 1 byte allen.memoryutil.dirver.Father.father\_byte\_1

offset : 20 size = 1 byte allen.memoryutil.dirver.Child.child\_byte\_0

offset : 21 size = 1 byte allen.memoryutil.dirver.Child.child\_byte\_1

offset : 24 size = 8 long allen.memoryutil.dirver.Child.child\_long

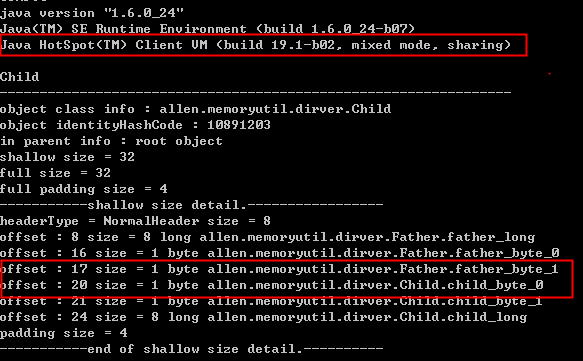
padding size = 4

-----------end of shallow size detail.----------

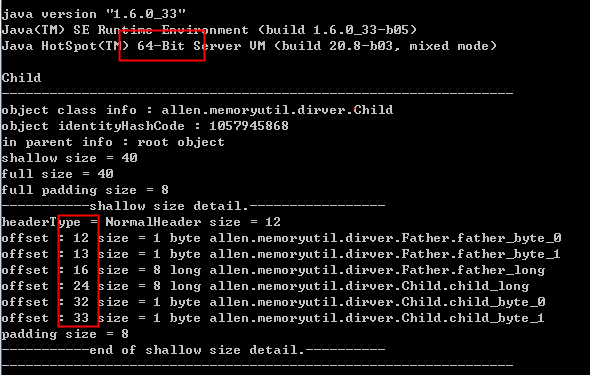
----------------------------------------------------------------

可以看到，父类的field先单独排列(有重排序)，然后才是子类的field排列(有重排序)。

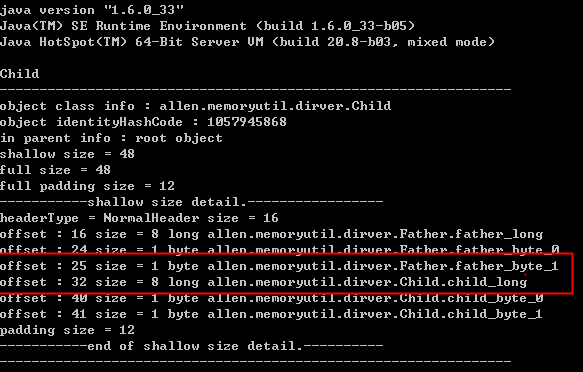
32bits VM

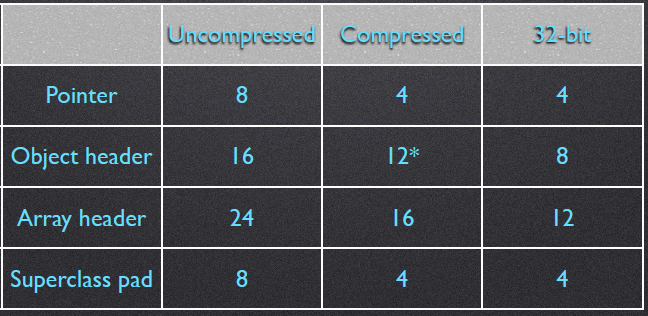


64bits VM开压缩指针



64bits VM关压缩指针





## 如何合法浪费内存

理解了内存对齐和父类的field单独排列，我们可以合法的构造浪费内存的对象。

**class** C\_A {

**byte** a\_byte;

}

**class** C\_B **extends** C\_A {

**long** b\_long;

}

**class** C\_C **extends** C\_B {

**byte** c\_byte;

}

**class** C\_D **extends** C\_C {

**long** d\_long;

}

C\_D

----------------------------------------------------------------

object class info : allen.memoryutil.dirver.C\_D

object identityHashCode : 30377347

in parent info : root object

shallow size = 40

full size = 40

-----------shallow size detail.-----------------

headerType = NormalHeader size = 8

offset : 8 size = 1 byte allen.memoryutil.dirver.C\_A.a\_byte

offset : 16 size = 8 long allen.memoryutil.dirver.C\_B.b\_long

offset : 24 size = 1 byte allen.memoryutil.dirver.C\_C.c\_byte

offset : 32 size = 8 long allen.memoryutil.dirver.C\_D.d\_long

padding size = 14

-----------end of shallow size detail.----------

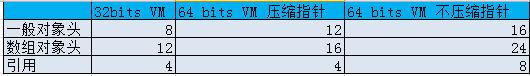
----------------------------------------------------------------

可以看到，由于类型在继承体系中，field是byte和long间隔定义，导致padding比较大。

当类的继承链比较长时，则最多可以浪费7/16=43%的内存。

## 32位jvm和64位jvm

64bits JVM和32bits相比变化如下：



对象占用的内存大小受到VM参数UseCompressedOops的影响。

开启(-XX:+UseCompressedOops) 可以压缩指针。

关闭(-XX:-UseCompressedOops) 可以关闭压缩指针。

原理：

4bytes引用(地址编码)，当寻址单位为1 byte时，寻址空间为4G空间。

由于java对象都是8bytes对齐的，因此，所有java对象地址的低3bits都是0，有点浪费。

如果寻址单位从1byte改为8bytes，则可以去除该浪费。

4bytes引用(地址编码)，当寻址单位为8 byte时，寻址空间为32G空间。

可以看到，压缩指针技术只适用于java堆大小<=32G的情况。

UseCompressedOops默认开启规则：

Compressed oops is supported and enabled by default in Java SE 6u23 and later. In Java SE 7, use of compressed oops is the default for 64-bit JVM processes when -Xmx isn't specified and for values of -Xmx less than 32 gigabytes. For JDK 6 before the 6u23 release, use the -XX:+UseCompressedOops flag with the java command to enable the feature.

## memoryutil详情输出简要说明

"abcd"字符串的详情例子

----------------------------------------------------------------

object class info : java.lang.String //对象类型

object identityHashCode : 4875744 //对象identityHashCode

in parent info : root object

shallow size = 24 //对象shallow size

full size = 48 //对象full size

full padding size = 4 //对象full padding size

-----------shallow size detail.----------------- //对象shallow size的详情信息

headerType = NormalHeader size = 8 //普通对象头，大小为8

offset : 8 size = 4 private final char[] java.lang.String.value //value field的offset为8,大小为4。

offset : 12 size = 4 private final int java.lang.String.offset //offset field的offset为12,大小为4。

offset : 16 size = 4 private final int java.lang.String.count //count field的offset为16,大小为4。

offset : 20 size = 4 private int java.lang.String.hash //hash field的offset为20,大小为4。

padding size = 0 //padding为0。

-----------end of shallow size detail.----------

//该字符串引用的char[]详情信息

object class info : char[]

object identityHashCode : 15672056

in parent info : private final char[] java.lang.String.value in parent. //对象在被引用对象中的位置信息。

shallow size = 24

full size = 24

full padding size = 4

-----------shallow size detail.-----------------

headerType = ArrayHeader size = 12 //数组对象头，大小为12

compType = char arrayLength = 4 size = ( 2 \* 4 ) = 8 //数组的类型为char，长度为4，大小为8。

padding size = 4 //padding为4。

-----------end of shallow size detail.----------

----------------------------------------------------------------

## 参考

<http://www.javamex.com/tutorials/memory/index.shtml>

<http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#compressedOop>