10 | Java对象的内存布局

2018-08-13 郑雨迪

10 | Java对象的内存布局
朗读人：郑雨迪　11′19″ | 5.19M　0:00 / 0:00

在 Java 程序中，我们拥有多种新建对象的方式。除了最为常见的 new 语句之外，我们还可以通过反射机制、Object.clone 方法、反序列化以及 Unsafe.allocateInstance 方法来新建对象。

其中，Object.clone 方法和反序列化通过直接复制已有的数据，来初始化新建对象的实例字段。Unsafe.allocateInstance 方法则没有初始化实例字段，而 new 语句和反射机制，则是通过调用构造器来初始化实例字段。

以 new 语句为例，它编译而成的字节码将包含用来请求内存的 new 指令，以及用来调用构造器的 invokespecial 指令。

```
1// Foo foo = new Foo(); 编译而成的字节码
20 new Foo
33 dup
44 invokespecial Foo()
57 astore_1
```

□ 复制代码

提到构造器，就不得不提到 Java 对构造器的诸多约束。首先，如果一个类没有定义任何构造器的话， Java 编译器会自动添加一个无参数的构造器。

```
1// Foo 类构造器会调用其父类 Object 的构造器
2public Foo();
30 aload_0 [this]
41 invokespecial java.lang.Object() [8]
54 return
```

□ 复制代码

然后，子类的构造器需要调用父类的构造器。如果父类存在无参数构造器的话，该调用可以是隐式的，也就是说 Java 编译器会自动添加对父类构造器的调用。但是，如果父类没有无参数构造器，那么子类的构造器则需要显式地调用父类带参数的构造器。

显式调用又可分为两种，一是直接使用“super”关键字调用父类构造器，二是使用“this”关键字调用同一个类中的其他构造器。无论是直接的显式调用，还是间接的显式调用，都需要作为构造器的第一条语句，以便优先初始化继承而来的父类字段。（不过这可以通过调用其他生成参数的方法，或者字节码注入来绕开。）

总而言之，当我们调用一个构造器时，它将优先调用父类的构造器，直至 Object 类。这些构造器的调用者皆为同一对象，也就是通过 new 指令新建而来的对象。

你应该已经发现了其中的玄机：通过 new 指令新建出来的对象，它的内存其实涵盖了所有父类中的实例字段。也就是说，虽然子类无法访问父类的私有实例字段，或者子类的实例字段隐

藏了父类的同名实例字段，但是子类的实例还是会为这些父类实例字段分配内存的。

这些字段在内存中的具体分布是怎么样的呢？今天我们就来看看对象的内存布局。

压缩指针

在 Java 虚拟机中，每个 Java 对象都有一个对象头（object header），这个由标记字段和类型指针所构成。其中，标记字段用以存储 Java 虚拟机有关该对象的运行数据，如哈希码、GC 信息以及锁信息，而类型指针则指向该对象的类。

在 64 位的 Java 虚拟机中，对象头的标记字段占 64 位，而类型指针又占了 64 位。也就是说，每一个 Java 对象在内存中的额外开销就是 16 个字节。以 Integer 类为例，它仅有一个 int 类型的私有字段，占 4 个字节。因此，每一个 Integer 对象的额外内存开销至少是 400%。这也是为什么 Java 要引入基本类型的原因之一。

为了尽量较少对象的内存使用量，64 位 Java 虚拟机引入了压缩指针 [1] 的概念（对应虚拟机选项 -XX:+UseCompressedOops，默认开启），将堆中原本 64 位的 Java 对象指针压缩成 32 位的。

这样一来，对象头中的类型指针也会被压缩成 32 位，使得对象头的大小从 16 字节降至 12 字节。当然，压缩指针不仅可以作用于对象头的类型指针，还可以作用于引用类型的字段，以及引用类型数组。

那么压缩指针是什么原理呢？

打个比方，路上停着的全是房车，而且每辆房车恰好占据两个停车位。现在，我们按照顺序给它们编号。也就是说，停在 0 号和 1 号停车位上的叫 0 号车，停在 2 号和 3 号停车位上的叫 1 号车，依次类推。

原本的内存寻址用的是车位号。比如说我有一个值为 6 的指针，代表第 6 个车位，那么沿着这个指针可以找到 3 号车。现在我们规定指针里存的值是车号，比如 3 指代 3 号车。当需要查找 3 号车时，我便可以将该指针的值乘以 2，再沿着 6 号车位找到 3 号车。

这样一来，32 位压缩指针最多可以标记 2 的 32 次方辆车，对应着 2 的 33 次方个车位。当然，房车也有大小之分。大房车占据的车位可能是三个甚至是更多。不过这并不会影响我们的寻址算法：我们只需跳过部分车号，便可以保持原本车号 *2 的寻址系统。

上述模型有一个前提，你应该已经想到了，就是每辆车都从偶数号车位停起。这个概念我们称之为内存对齐（对应虚拟机选项 -XX:ObjectAlignmentInBytes，默认值为 8）。

默认情况下，Java 虚拟机堆中对象的起始地址需要对齐至 8 的倍数。如果一个对象用不到 8N 个字节，那么空白的那部分空间就浪费掉了。这些浪费掉的空间我们称之为对象间的填充（padding）。

在默认情况下，Java 虚拟机中的 32 位压缩指针可以寻址到 2 的 35 次方个字节，也就是 32GB 的地址空间（超过 32GB 则会关闭压缩指针）。

在对压缩指针解引用时，我们需要将其左移 3 位，再加上一个固定偏移量，便可以得到能够寻址 32GB 地址空间的伪 64 位指针了。

此外，我们可以通过配置刚刚提到的内存对齐选项（-XX:ObjectAlignmentInBytes）来进一步提升寻址范围。但是，这同时也可能增加对象间填充，导致压缩指针没有达到原本节省空间

的效果。

举例来说，如果规定每辆车都需要从偶数车位号停起，那么对于占据两个车位的小房车来说刚刚好，而对于需要三个车位的大房车来说，也仅是浪费一个车位。

但是如果规定需要从 4 的倍数号车位停起，那么小房车则会浪费两个车位，而大房车至多可能浪费三个车位。

当然，就算是关闭了压缩指针，Java 虚拟机还是会进行内存对齐。此外，内存对齐不仅存在于对象与对象之间，也存在于对象中的字段之间。比如说，Java 虚拟机要求 long 字段、double 字段，以及非压缩指针状态下的引用字段地址为 8 的倍数。

字段内存对齐的其中一个原因，是让字段只出现在同一 CPU 的缓存行中。如果字段不是对齐的，那么就有可能出现跨缓存行的字段。也就是说，该字段的读取可能需要替换两个缓存行，而该字段的存储也会同时污染两个缓存行。这两种情况对程序的执行效率而言都是不利的。

下面我来介绍一下对象内存布局另一个有趣的特性：字段重排列。

字段重排列

字段重排列，顾名思义，就是 Java 虚拟机重新分配字段的先后顺序，以达到内存对齐的目的。Java 虚拟机中有三种排列方法（对应 Java 虚拟机选项 -XX:FieldsAllocationStyle，默认值为 1），但都会遵循如下两个规则。

其一，如果一个字段占据 C 个字节，那么该字段的偏移量需要对齐至 NC。这里偏移量指的是字段地址与对象的起始地址差值。

以 long 类为例，它仅有一个 long 类型的实例字段。在使用了压缩指针的 64 位虚拟机中，尽管对象头的大小为 12 个字节，该 long 类型字段的偏移量也只能是 16，而中间空着的 4 个字节便会被浪费掉。

其二，子类所继承字段的偏移量，需要与父类对应字段的偏移量保持一致。

在具体实现中，Java 虚拟机还会对齐子类字段的起始位置。对于使用了压缩指针的 64 位虚拟机，子类第一个字段需要对齐至 4N；而对于关闭了压缩指针的 64 位虚拟机，子类第一个字段则需要对齐至 8N。

```
1class A {
2long l;
3int i;
4}
5
6class B extends A {
7long l;
8int i;
9}
```

□ 复制代码

我在文中贴了一段代码，里边定义了两个类 A 和 B，其中 B 继承 A。A 和 B 各自定义了一个 long 类型的实例字段和一个 int 类型的实例字段。下面我分别打印了 B 类在启用压缩指针和未启用压缩指针时，各个字段的偏移量。

```
1 # 启用压缩指针时，B 类的字段分布
2 B object internals:
3 OFFSET SIZE TYPE DESCRIPTION
4 0 4 (object header)
5 4 4 (object header)
```

□ 复制代码

```
6 8 4 (object header)
7 12 4 int A.i 0
8 16 8 long A.l 0
9 24 8 long B.l 0
1032 4 int B.i 0
1136 4 (loss due to the next object alignment)
```

当启用压缩指针时，可以看到 Java 虚拟机将 A 类的 int 字段放置于 long 字段之前，以填充因为 long 字段对齐造成的 4 字节缺口。由于对象整体大小需要对齐至 8N，因此对象的最后会有 4 字节的空白填充。

```
1 # 关闭压缩指针时，B 类的字段分布                                    □ 复制代码
2 B object internals:
3 OFFSET SIZE TYPE DESCRIPTION
4 0 4 (object header)
5 4 4 (object header)
6 8 4 (object header)
7 12 4 (object header)
8 16 8 long A.l
9 24 4 int A.i
1028 4 (alignment/padding gap)
1132 8 long B.l
1240 4 int B.i
1344 4 (loss due to the next object alignment)
```

当关闭压缩指针时，B 类字段的起始位置需对齐至 8N。这么一来，B 类字段的前后各有 4 字节的空白。那么我们可不可以将 B 类的 int 字段移至前面的空白中，从而节省这 8 字节呢？

我认为是可以的，并且我修改过后的 Java 虚拟机也没有跑崩。由于 HotSpot 中的这块代码年久失修，公司的同事也已经记不得是什么原因了，那么姑且先认为是一些历史遗留问题吧。

Java 8 还引入了一个新的注释 @Contended，用来解决对象字段之间的虚共享（false sharing）问题 [2]。这个注释也会影响到字段的排列。

虚共享是怎么回事呢？假设两个线程分别访问同一对象中不同的 volatile 字段，逻辑上它们并没有共享内容，因此不需要同步。

然而，如果这两个字段恰好在同一个缓存行中，那么对这些字段的写操作会导致缓存行的写回，也就造成了实质上的共享。（volatile 字段和缓存行的故事我会在之后的篇章中详细介绍。）

Java 虚拟机会让不同的 @Contended 字段处于独立的缓存行中，因此你会看到大量的空间被浪费掉。具体的分布算法属于实现细节，随着 Java 版本的变动也比较大，因此这里就不做阐述了。

如果你感兴趣，可以利用实践环节的工具，来查阅 Contended 字段的内存布局。注意使用虚拟机选项 -XX:-RestrictContended。如果你在 Java 9 以上版本试验的话，在使用 javac 编译时需要添加 --add-exports java.base/jdk.internal.vm.annotation=ALL-UNNAME

总结和实践

今天我介绍了 Java 虚拟机构造对象的方式，所构造对象的大小，以及对象的内存布局。

常见的 new 语句会被编译为 new 指令，以及对构造器的调用。每个类的构造器皆会直接或者间接调用父类的构造器，并且在同一个实例中初始化相应的字段。

Java 虚拟机引入了压缩指针的概念，将原本的 64 位指针压缩成 32 位。压缩指针要求 Java 虚拟机堆中对象的起始地址要对齐至 8 的倍数。Java 虚拟机还会对每个类的字段进行重排列，使得字段也能够内存对齐。

今天的实践环节比较简单，你可以使用我在工具篇中介绍过的 JOL 工具，来打印你工程中的类的字段分布情况。

```
1curl -L -O http://central.maven.org/maven2/org/openjdk/jol/jol-cli/0.9/jol-cli-0.9-full.jar
2java -cp jol-cli-0.9-full.jar org.openjdk.jol.Main internals java.lang.String
```

复制代码

[1] https://wiki.openjdk.java.net/display/HotSpot/CompressedOops
[2] http://openjdk.java.net/jeps/142

写留言

精选留言

godtrue 15

小结

1:Java中创建对象的方式

1-1:new -通过调用构造器来初始化实例字段

1-2:反射-通过调用构造器来初始化实例字段

1-3:Object.clone-通过直接复制已有的数据，来初始化新建对象的实例字段

1-4:反序列化-通过直接复制已有的数据，来初始化新建对象的实例字段

1-5:Unsafe.allocateInstance-没有初始化对象的实例字段

2:Java对象的空间占用

2-1:通过new创建的对象，涵盖了它所有父类中的对象实例的字段

2-2:对象头，由标记字段和类型指针构成

2-3:标记字段，用于存储Java虚拟机有关该对象的运行数据，比如：哈希码、GC信息、锁信息等

2-4:类型指针，用于指向该对象的类

2-5:此对象的实例字段对应的内存空间

3:压缩指针

JVM的内存空间有限且昂贵，所以，能缩减的就缩减，通过一定的算法改进压缩类型指针的空间后仍可以寻址到对象的实例对应的类，所以，就采用了

4:字段重排
意思是JVM会重新分配字段的位置，和我们Java源码中属性声明的位置存在差异，猜想Java编译器编译后的字节码是没有改变源码中字段声明的位置的，这样做是为了更好的实现内存对齐，内存对齐本质上会浪费一定的内存空间，不过可以减少内存行的读取次数，通过一消一涨的比对发现这样对于JVM的性能有一定的提高，所以，也就使用了这种方式，浪费点空间能提高性能也是值得的

疑问❓
1:为什么一个子类即使无法访问父类的私有实例字段，或者子类实例字段隐藏了父类的同名实例字段，子类的实例还是会为这些父类实例字段分配内存呢?
另外，如果采用指针指向的方式定位父类实例的内容是否能更节省内存空间?

2:五种创建对象的方式，通过new指令新建出来的对象，他的内存其实涵盖了所有父类中的实例字段，其他的方式是怎样的哪?

2018-08-13

三木子 4
有一个小白问题，new一个对象(继承一个类)会调用父类构造器，这个可以理解，因为对象可能调用父类方法。那么为什么new对象会调用到object呢？这有什么用意吗?
2018-08-13

amourling 3
作者大大辛苦了，货很干，搭配《深入理解java虚拟机》会很香
2018-08-13

life is short, enjoy mor... 1
对象头

每个对象都有一个对象头，对象头包括两部分，标记信息和类型指针。

标记信息包括哈希值，锁信息，GC信息。类型指针指向这个对象的class。

两个信息分别占用8个字节，所以每个对象的额外内存为16个字节。很消耗内存。

压缩指针

为了减少类型指针的内存占用，将64位指针压缩至32位，进而节约内存。之前64位寻址，寻的是字节。现在32位寻址，寻的是变量。再加上内存对齐(补齐为8的倍数)，可以每次寻变量都以一定的规则寻找，并且一定可以找得到。

内存对齐

内存对齐的另一个好处是，使得CPU缓存行可以更好的实施。保证每个变量都只出现在一条缓存行中，不会出现跨行缓存。提高程序的执行效率。

字段重排序

其实就是更好的执行内存对齐标准，会调整字段在内存中的分布，达到方便寻址和节省空间的目的。

虚共享

当两个线程分别访问一个对象中的不同volatile字段，理论上是不涉及变量共享和同步要求的。但是如果两个volatile字段处于同一个CPU缓存行中，对其中一个volatile字段的写操作，会导致整个缓存行的写回和读取操作，进而影响到了另一个volatile变量，也就是实际上的共享问题。

@Contented注解

该注解就是用来解决虚共享问题的，被该注解标识的变量，会独占一个CPU缓存行。但也因此浪费了大量的内存空间。

2018-10-11
作者回复

赞总结！

2018-10-12

everyok22 1
你文章里说： 64位的JVM中，不采用压缩指针的方式，标记字段与类型指针分别占用8个字节，而采用了压缩指针标记字段与类型指针都会压成32位（8字节）那对象头不是只占用8个字节么，为什么你说是12个字节
2018-08-22
作者回复

标记字段没有被压缩。

2018-08-22

倔强 1
也就是说默认情况下，小于32G的堆内存中的对象引用为4个字节，一旦堆内存大于32G，对象引用为8个字节
2018-08-14

Mr.钧 0
想请教老师大大几个问题：
1、什么是CUP缓存行?
2、如果跨缓存行的字段，为什么会降低执行效率？是因为某些读取程序，一行一行

的读效率较高？还是因为以行分割呢？
3、明显启用压缩指针，性能更高，但是为什么还会在64位情况下，不启用压缩指针的情况呢？ 是因为CPU运行速度更快，可以护士不压缩指针导致的内存浪费吗？
2018-10-16

xlogic 0
字段重排列

其一，如果一个字段占据 C 个字节， 那么该字段的偏移量需要对齐至 NC。这里的偏移量指的是字段地址与对象的起始地址差值

以 long 类为例，它仅有一个 long 类型的实例字段。在使用了压缩指针的 64 位虚拟机中，尽管对象头的大小为 12 个字节，该 long 类型字段的偏移量也只能是 16，而中间空着的 4 个字节便会被浪费掉。

个人理解：1. 应该是 Long 类型；2. 因为 long 字段的占 8 个字节，所以偏移量是N8，比12大的最接近的数就是 16，所以偏移量就是16，也就是说字段与对象的起始位置差是16。
2018-09-29

Geek_987169 0
老师，请教您几个问题
1：每个类都有一个对应的class对象，那么这class对象是什么时候生成的，存储jvm的哪个区域？
2：类实例对象object header中的类型指针其实就是指向该类所属class的对象的指针吗？
3：class对象的内存结构又是什么样子的呢？类似于普通Java实例对象吗？
ps：这个指针压缩的原理有些困扰到我了。。。求解惑！！！
2018-09-22

Geek_987169 0
老师，“堆中原本 64 位的 Java 对象指针压缩成 32 位”这几句话中的“64位java对象指针”是个啥？在哪里？为什么会影响想到对象头中类型指针？
2018-09-21

哈哈哈哈 0
老师你好，为了确定数组的对象头的大小，看了网上各种描述很困惑，我去找了openjdk8的源码：
在openjdk-8/hotspot/src/share/vm/oops/arrayOop.hpp里面的class arrayOopDesc
找到size_t hs = align_size_up(length_offset_in_bytes() + sizeof(int), HeapWordSize);
想确认一下64位虚拟机环境下，不开启指针压缩，数组的对象头应该是8+8+4=20字节对吗
求回复啊
2018-09-11

一个坏人 0

老师好，请教一下：“自动内存管理系统为什么要求对象的大小必须是8字节的整数倍？”，即内存对齐的根本原因在于？

2018-08-25

作者回复

在某些体系架构上，不对齐的话内存读写会报错。

在X86_64上，一个是为了让字段也能对齐，这样就不会出现字段横跨两个缓存行的情况，另一个原因更像个副作用，就是对象地址最后三位一直是0，JVM利用这个特性来实现压缩指针，也可以用这三位来记录一些额外信息

2018-08-30

周仕林 0

对象头的组成如果阅读过周志明的JVM虚拟机会发现作者说的有一些有失偏颇，对象头的组成是对象运行信息，类型指针（如果对象访问采用直接指针），数组长度（如果对象是数组）

2018-08-21

作者回复

周老师书上的是提炼之后的抽象说法。

HotSpot的对象头一直是mark word(标记字段)和类型指针。如果对源代码有兴趣可以查看OpenJDK源代码目录下，src/hotspot/share/oops/oop.hpp里的class oopDesc。数组对象头是同目录下的arrayOop.hpp里的class arrayOopDesc

2018-08-22

小白猪 0

引用类型是4个字节吗，还是压缩开启后是4个字节

2018-08-15

贾智文 0

默认情况下32位可以寻址2的35次，应该是因为地址是32位乘以2的三次（默认对其为8），那么如果不采用压缩指针，能够寻址的范围应该是2的64次对吧。然后之前模糊的地方就是觉得两个寻址范围并不一致，是不是可以这么理解并没有通过压缩指针让两个寻址范围一致，而是通过压缩指针放大了32位的寻址空间使它够用了

2018-08-15

作者回复

对的

2018-08-15

贾智文 0

有一点想不明白，既然内存对齐是八位而不是举例的两位为什么空间只是从64位变成32而不是从64变成8

2018-08-15

作者回复

不是很理解你的问题。

对象间需要内存对齐至8字节。64位和32位对应8字节和4字节。

2018-08-15

大能猫 0

最近研究String时遇到一个跟Java内存相关的问题：常量池里到底有没有存放对象？常量池主要存放两大类常量：字面量（Literal）和符号引用（Symbolic Reference）；

如果常量池里有一个“hello”的字面量，这个字面量算是一个对象吗？如果不算对象，那么它所指向的对象又存放在哪里呢

2018-08-14

作者回复

String literal指向的对象存放在JVM的String pool里。

2018-08-15

熊猫酒仙 0

接触过C/C++的内存字节对齐，就比较好理解本章内容了。希望老师后面讲讲java内存在并发上的相关机制，譬如搞不懂的内存障是怎么实现的！

2018-08-14

作者回复

底层的内存屏障，比如说mfence，lock指令，是用来防止指令重排布的。Java里的内存屏障除了生成上述底层指令外，还会限制即时编译器对内存访问的重排序。之后我会讲Java内存模型。

2018-08-15

xianhai 0

压缩指针指向的类对象是存在heap中，还是heap外？

2018-08-14