二

# 10 Java对象的内存布局

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。

```
// Foo foo = new Foo(); 编译而成的字节码
  0 new Foo
  3 dup
  4 invokespecial Foo()
  7 astore_1
```

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。

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

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。

```
class A {
  long l;
  int i;
}

class B extends A {
  long l;
  int i;
}
```

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。

```
# 启用压缩指针时，B 类的字段分布
B object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0     4        (object header)
      4     4        (object header)
      8     4        (object header)
     12     4    int A.i                                          0
     16     8   long A.l                                          0
     24     8   long B.l                                          0
     32     4    int B.i                                          0
     36     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。

```
# 关闭压缩指针时，B 类的字段分布
B object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0     4        (object header)
      4     4        (object header)
      8     4        (object header)
     12     4        (object header)
     16     8   long A.l
     24     4    int A.i
     28     4        (alignment/padding gap)
     32     8   long B.l
     40     4    int B.i
     44     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。

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

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

上一页 下一页