链滴

Java 高并发与多线程（二）——Java 普通对象的对象头及组成部分

作者：xiaoyao2102

原文链接：https://ld246.com/article/1582177721141

来源网站：链滴

Java 中的对象除了我们可见的属性部分，其实还有另外不可见的内容，这部分就是**对象头（Object Header）**。本文讲探究一下对象头里具体都有哪些细节，为将来 synchronized 关键字的讨论打一个基。

1. 工具准备

笔者本地使用 64 位的 JDK8，然后引入一个叫**JOL**的依赖包，全称 Java Object Layout。截止到本写作时，最高版本为 0.10，这里选用了非最新的使用较多的一个版本，GAV 如下：

```
<dependency>
    <groupId>org.openjdk.jol</groupId>
    <artifactId>jol-core</artifactId>
    <version>0.9</version>
</dependency>
```

这个工具本身功能很丰富，可以从各个角度对内存中的对象进行查看，我们这里只使用查看对象内部构的功能，代码如下。（有兴趣的同学可以去OpenJDK官网参考更多样例）

```
public class Test {

    public static void main(String[] args) {
        Object o = new Object();

        ClassLayout classLayout = ClassLayout.parseInstance(o);

        System.out.println(classLayout.toPrintable());
    }
}
```

执行输出如下：

```
java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 00 00 00 (00000001 00000000 00000000 00
00000) (1)
      4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00
00000) (0)
      8     4        (object header)                           e5 01 00 20 (11100101 00000001 00000000 00
00000) (536871397)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

这里需要特别说明的是，由于存在**大端小端**的问题，我们看上面 bit 值的顺序应该是**每 8 个 bit 一组组内从左往右看，组间从后往前看。**即，第二行第四组bit值为整个Object Header的开头，其中第四从左开始的第一个bit也是整个对象头的第一个bit；第一行第一组的bit值为整个对象头的末尾，其中为1的那个bit就是整个对象头最后一个bit。

2. 对象头的具体内容

上一节中我们用 JOL 输出了一个对象在内存中的内容，这一节我们就对这个结果进行分析。

2.1 普通对象的内存组成

普通对象的内存占用由以下四个部分组成：

1. Mark Word 占 8 字节
2. Class Pointer 占 4 字节
3. 实例数据
4. Padding 对齐

其中 Mark Word 和 Class Pointer 合起来称为**Object Header（对象头）**。Padding 部分并不携带何信息，它存在的原因是当前 64 位处理器每次读取内存都是 8 个字节为一组，因此如果一个对象的小可以正好被 8 整除，那么每次读取的数据块都可以保证在同一个对象内，省去了判断对象结尾位置样的操作。

2.2 Mark Word 部分

对象头里面主要的信息都存在 Mark Word 部分，具体内容可以看下面这个表格。

<table>
<tr style="background-color:#bfbfbf">
<td>锁状态</td>
<td colspan="2" >56 bit</td>
<td>1 bit</td>
<td>4 bit</td>
<td>1 bit</td>
<td>2 bit</td>
</tr>
<tr>
<td>无锁</td>
<td>31 bit 未用</td>
<td>25 bit hash</td>
<td>未用</td>
<td>分代年龄</td>
<td>是否偏向锁</td>
<td>01</td>
</tr>
<tr>
<td>偏向锁</td>
<td>54 bit 线程ID</td>
<td>2 bit Epoch</td>
<td>未用</td>
<td>分代年龄</td>
<td>是否偏向锁</td>
<td>01</td>
</tr>
<tr>
<td>轻量级锁</td>
<td colspan="5">指向栈中锁记录的指针</td>
<td>00</td>
</tr>
<tr>
<td>重量级锁</td>
<td colspan="5">指向重量级锁的指针</td>
<td>10</td>

```
        </tr>
        <tr>
            <td>GC标记</td>
            <td colspan="5">空</td>
            <td>11</td>
        </tr>
</table>
```

从这个表格中我们可以看出，Mark Word 中的内容是根据对象的锁状态来决定的，不同的锁状态 Mark Word 会存储不同的内容。涉及锁相关的内容我们会在下一篇讨论 synchronized 关键字的文章中解，本文会讨论一些其它部分的内容，目前读者只要知道**synchronized 关键字会修改对象的 Mark Word**就好。

2.2.1 hash code

在无锁状态下，Mark Word 会用前 56 个 bit 中的后 25 个 bit 记录 hash code。我们看代码如下：

```
import org.openjdk.jol.info.ClassLayout;

public class Test {

    public static void main(String[] args) {
        Object o = new Object();

        o.hashCode();

        ClassLayout classLayout = ClassLayout.parseInstance(o);

        System.out.println(classLayout.toPrintable());
    }
}
```

输出是：

```
java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 a5 e5 4a (00000001 10100101 11100101 01
01010) (1256563969)
      4     4        (object header)                           01 00 00 00 (00000001 00000000 00000000 00
00000) (1)
      8     4        (object header)                           e5 01 00 20 (11100101 00000001 00000000 00
00000) (536871397)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

根据本节最开始提到的阅读顺序，可以看出，有区别的部分是第一行后三组 bit 和第二行第一组的最一个 bit，一共是 3 * 8 + 1 = 25 个 bit，这部分就是 hash code。第二行第一组的前 7 个 bit 和后组 bit，一共 7 + 3 * 8 = 31 个 bit，就是未使用的部分。

这个 hash code 只有在对象第一次调用 hashCode()方法时生成，后面再需要 hash code 的地方就直接从对象头里面读取。这里我们可以得到一个额外的结论：**最好不要使用可变的属性来生成 hash code。**原因是如果这个可变属性发生了变化，在类似 HashMap 和 HashSet 这种依赖于 hash code

场景中，可能会有意外的问题发生。

2.2.2 分代年龄

我们知道，堆内存是划分为新生代和老年代区域的。一个对象在每经历一次 GC 后如果还存活，那么的年龄就会 +1。这个年龄就是记录在对象头中，占用 4 个 bit，因此一个对象的年龄最大就到 15，超过时，对象就会从新生代晋升到老年代。当然这个值可以通过--XX:MaxTenuringThreshold 参数调整，但也不能超过 15。

我们还是来看代码：

```
import org.openjdk.jol.info.ClassLayout;

public class Test {

    public static void main(String[] args) {
        Object o = new Object();

        ClassLayout classLayout = ClassLayout.parseInstance(o);

        System.gc(); // 向jvm发送gc信号，但不保证一定gc

        System.out.println(classLayout.toPrintable());
    }
}
```

输出为：

```
java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           09 00 00 00 (00001001 00000000 00000000 00
00000) (9)
      4     4        (object header)                           00 00 00 00 (00000000 00000000 00000000 00
00000) (0)
      8     4        (object header)                           e5 01 00 20 (11100101 00000001 00000000 00
00000) (536871397)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

根据表格，第一个 bit 未用，从第二到第五个 bit 为分代年龄，我们可以看到第五个 bit 值加了 1。

2.3 Class Pointer

JVM 的内存区域中，有一个叫**方法区**的部分，这部分主要存储的内容是关于类的元数据。我们在堆内中的对象，如果需要该类型的元数据，就是通过这个 Class Pointer 来找到的。

有一些 C 基础的同学可能会发现，64 位机器中的指针应该占用 8 个 byte，可是这里只占用了 4 个其实这里是 JVM 的一个优化内容，叫作**指针压缩。**

我们可以在命令行里输入：

```
java -XX:+PrintCommandLineFlags -version
```

可以看到输出为：

```
-XX:InitialHeapSize=132730432 -XX:MaxHeapSize=2123686912 -XX:+PrintCommandLineFlags
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:-UseLargePagesIndividual
llocation -XX:+UseParallelGC
java version "1.8.0_171"
Java(TM) SE Runtime Environment (build 1.8.0_171-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.171-b11, mixed mode)
```

其中有两项-XX:+UseCompressedClassPointers -XX:+UseCompressedOops 就是控制指针压缩的经笔者测试，这两个参数只要关闭其中一个，Class Pointer 就会变为 8 字节大小。

3. 下集预告

这篇博文主要是介绍了 Object Header 的大概内容，在下篇 synchronized 关键字的探究中会对 Obj ct Header 中锁的部分进行详细介绍。