java头的信息分析

首先为什么我要去研究java的对象头呢?

这里截取一张hotspot的源码当中的注释

```
//  64 bits:
//  --------
//  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal object)
//  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased object)
//  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promoted object)
//  size:64 ----------------------------------------------------->| (CMS free block)
//
```

这张图换成人可读的表格如下

Object Header (128 bits)							
Mark Word (64 bits)						Klass Word (64 bits)	
unused:25	identity_hashcode:31	unused:1	age:4	biased_lock:1	lock:2	OOP to metadata object	无锁
thread:54	epoch:2	unused:1	age:4	biased_lock:1	lock:2	OOP to metadata object	偏向锁
ptr_to_lock_record:62					lock:2	OOP to metadata object	轻量锁
ptr_to_heavyweight_monitor:62					lock:2	OOP to metadata object	重量锁
					lock:2	OOP to metadata object	gc

意思是java的对象头在对象的不同状态下会有不同的表现形式，主要有三种状态，无锁状态、加锁状态、gc标记状态。那么我可以理解java当中的取锁其实可以理解是给对象上锁，也就是改变对象头的状态，如果上锁成功则进入同步代码块。但是java当中的锁有分为很多种，从上图可以看出大体分为偏向锁、轻量锁、重量锁三种锁状态。这三种锁的效率完全不同、关于效率的分析会在下文分析，我们只有合理的设计代码，才能合理的利用锁、那么这三种锁的原理是什么?所以我们需要先研究这个对象头。

java对象的布局以及对象头的布局

1、JOL来分析java的对象布局

首先添加JOL的依赖

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

```
 <version>0.9</version>
</dependency>
```

A.java

```
public class A {
 //没有任何字段
}
```

JOLExample1.java

```
package com.luban.layout;
import org.openjdk.jol.info.ClassLayout;
import org.openjdk.jol.vm.VM;
import static java.lang.System.out;

public class JOLExample1 {
 public static void main(String[] args) throws Exception {
 out.println(VM.current().details());
 out.println(ClassLayout.parseClass(A.class).toPrintable());
  }
}
```

运行结果

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 0-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

com.luban.layout.A object internals:
 OFFSET SIZE TYPE DESCRIPTION VALUE
  0 4 (object header) 01 00 00 00 (00000001 00000000 00000000 00000000)
(1)
  4 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000)
(0)
```

```
  8 4 (object header) 82 22 01 20 (10000010 00100010 00000001 00100000) (536945282)
  12 4 (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

分析结果1

```
Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]对应：[Oop(Ordinary Object Pointer), boolean, byte, char, short, int, float, long, double]大小
```

整个对象一共16B，其中对象头（Object header）12B，还有4B是对齐的字节（因为在64位虚拟机上对象的大小必须是8的倍数）,由于这个对象里面没有任何字段，故而对象的实例数据为0B？两个问题

1、什么叫做对象的实例数据呢?

2、那么对象头里面的12B到底存的是什么呢?

首先要明白什么对象的实例数据很简单，我们可以在A当中添加一个boolean的字段，大家都知道boolean字段占1B，然后再看结果

A.java

```
public class A {
 //占一个字节的boolean字段
 boolean flag =false;
}
```

运行结果2

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 0-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

com.luban.layout.A object internals:
 OFFSET SIZE TYPE DESCRIPTION VALUE
  0 4 (object header) 01 00 00 00 (00000001 00000000 00000000 00000000) (1)
```

```
  4 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000) (0)
  8 4 (object header) 82 22 01 20 (10000010 00100010 00000001 00100000) (536945282)
  12 1 boolean A.flag false
  13 3 (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 3 bytes external = 3 bytes total
```

分析结果2

整个对象的大小还是没有改变一共16B，其中对象头（Object header）12B，boolean字段flag（对象的实例数据）占1B、剩下的3B就是对齐字节。由此我们可以认为一个对象的布局大体分为三个部分分别是对象头（Object header）、对象的实例数据字节对齐

接下来讨论第二个问题，对象头为什么是12B？这个12B当中分别存储的是什么呢？（不同位数的VM对象头的长度不一样，这里指的是64bit的vm）

关于java对象头的一些专业术语

http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html

首先引用openjdk文档当中对对象头的解释

> **object header**
> Common structure at the beginning of every GC-managed heap object. (Every oop points to an object header.) Includes fundamental information about the heap object's layout, type, GC state, synchronization state, and identity hash code. Consists of two words. In arrays it is immediately followed by a length field. Note that both Java objects and VM-internal objects have a common object header format

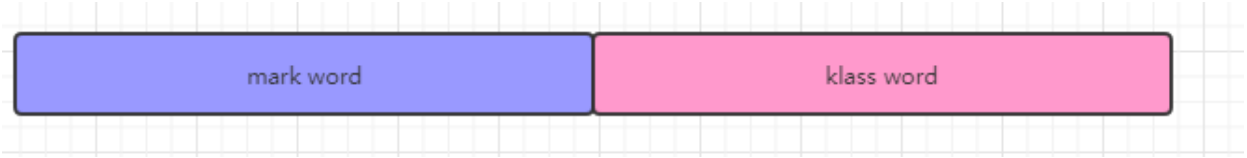
上述引用中提到一个java对象头包含了2个word，并且好包含了堆对象的布局、类型、GC状态、同步状态和标识哈希码，具体怎么包含的呢？又是哪两个word呢？

> **mark word**
> The first word of every object header. Usually a set of bitfields including synchronization state and identity hash code. May also be a pointer (with characteristic low bit encoding) to synchronization related information. During GC, may contain GC state bits.

mark word为第一个word根据文档可以知他里面包含了锁的信息，hashcode，gc信息等等，第二个word是什么呢？

> **klass pointer**
> The second word of every object header. Points to another object (a metaobject) which describes the layout and behavior of the original object. For Java objects, the "klass" contains a C++ style "vtable".

klass word为对象头的第二个word主要指向对象的元数据。



假设我们理解一个对象头主要上图两部分组成（数组对象除外，数组对象的对象头还包含一个数组长度），那么一个java的对象头多大呢？我们从JVM的源码注释中得知到一个mark word一个是64bit，那么klass的长度是多少呢？

所以我们需要想办法来获得java对象头的详细信息，验证一下他的大小，验证一下里面包含的信息是否正确。

根据上述利用JOL打印的对象头信息可以知道一个对象头是12B，其中8B是mark word 那么剩下的4B就是klass word了，和锁相关的就是mark word了，那么接下来重点分析mark word里面信息

在无锁的情况下markword当中的前56bit存的是对象的hashcode，那么来验证一下

java代码和运行结果：

```
1  public static void main(String[] args) throws Exception {
2   A a= new A();
3   out.println("befor hash");
4   //没有计算HASHCODE之前的对象头
5   out.println(ClassLayout.parseInstance(a).toPrintable());
6   //JVM 计算的hashcode
7   out.println("jvm------------"+Integer.toHexString(a.hashCode()));
8   HashUtil.countHash(a);
9   //当计算完hashcode之后，我们可以查看对象头的信息变化
10   out.println("after hash");
11   out.println(ClassLayout.parseInstance(a).toPrintable());
12
13  }
```

```
1  befor hash
2  com.luban.layout.A object internals:
3   OFFSET SIZE TYPE DESCRIPTION VALUE
4   0 4 (object header) 01 00 00 00 (00000001 00000000 00000000 00000000)
(1)
5   4 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000)
(0)
```

```
  8 4 (object header) 43 c1 00 20 (01000011 11000001 00000000 00100000) (536920387)
  12 1 boolean A.flag false
  13 3 (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 3 bytes external = 3 bytes total

jvm------------0x6ae40994
util-----------0x6ae40994
after hash
com.luban.layout.A object internals:
 OFFSET SIZE TYPE DESCRIPTION VALUE
  0 4 (object header) 01 94 09 e4 (00000001 10010100 00001001 11100100) (-469134335)
  4 4 (object header) 6a 00 00 00 (01101010 00000000 00000000 00000000) (106)
  8 4 (object header) 43 c1 00 20 (01000011 11000001 00000000 00100000) (536920387)
  12 1 boolean A.flag false
  13 3 (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 3 bytes external = 3 bytes total


Process finished with exit code 0
```

http://tool.oschina.net/hexconvert/ 进制转换

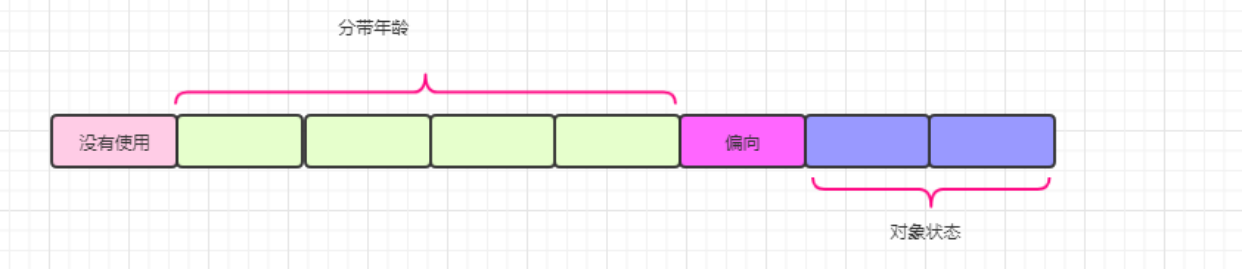
```
package com.luban.layout;
import sun.misc.Unsafe;
import java.lang.reflect.Field;

public class HashUtil {
 public static void countHash(Object object) throws NoSuchFieldException, IllegalAccessException {
  // 手动计算HashCode
  Field field = Unsafe.class.getDeclaredField("theUnsafe");
  field.setAccessible(true);
   Unsafe unsafe = (Unsafe) field.get(null);
   long hashCode = 0;
   for (long index = 7; index > 0; index--) {
   // 取Mark Word中的每一个Byte进行计算
   hashCode |= (unsafe.getByte(object, index) & 0xFF) << ((index - 1) * 8);
```

```
15    }
16    String code = Long.toHexString(hashCode);
17    System.out.println("util-----------0x"+code);
18
19    }
20  }
```

分析结果3：

1-----9行是没有进行hashcode之前的对象头信息，可以看到1-7B的56bit没有值，打印完hashcode之后16----21行就有值了，为什么是1-7B，不是0-6B呢？因为是小端存储。其中12行是我们通过hashcode方法打印的结果，13行是我根据1-7B的信息计算出来的hashcode，所以可以确定java对象头当中的mark work里面的后七个字节存储的是hashcode信息，那么第一个字节当中的八位分别存的就是分带年龄、偏向锁信息，和对象状态，这个8bit分别表示的信息如下图（其实上图也有信息），这个图会随着对象状态改变而改变，下图是无锁状态下



关于对象状态一共分为五种状态，分别是无锁、偏向锁、轻量锁、重量锁、GC标记，那么2bit，如何能表示五种状态（2bit最多只能表示4中状态分别是：00,01,10,11），jvm做的比较好的是把偏向锁和无锁状态表示为同一个状态，然后根据图中偏向锁的标识再去标识是无锁还是偏向锁状态。什么意思呢？写个代码分析一下，在写代码之前我们先记得无锁状态下的信息**00000001，然后写一个偏向锁的例子看看结果**

Java代码和输出结果：

```
1  package com.luban.layout;
2  import org.openjdk.jol.info.ClassLayout;
3  import static java.lang.System.out;
4
5  public class JOLExample2 {
```

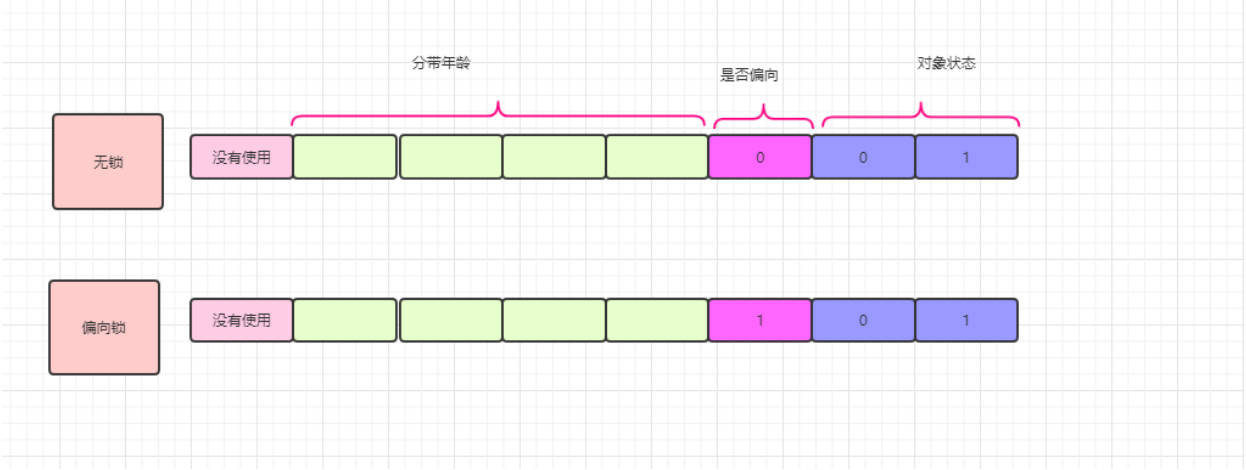
```
6   static A a;
7   public static void main(String[] args) throws Exception {
8   //Thread.sleep(5000);
9   a = new A();
10   out.println("befre lock");
11   out.println(ClassLayout.parseInstance(a).toPrintable());
12   sync();
13   out.println("after lock");
14   out.println(ClassLayout.parseInstance(a).toPrintable());
15   }
16
17   public static void sync() throws InterruptedException {
18   synchronized (a){
19   System.out.println("我也不知道要打印什么");
20   }
21
22   }
23  }
```

```
1  befre lock
2  com.luban.layout.A object internals:
3   OFFSET SIZE TYPE DESCRIPTION VALUE
4   0 4 (object header) 01 00 00 00 (00000001 00000000 00000000 00000000)
(1)
5   4 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000)
(0)
6   8 4 (object header) 43 c1 00 20 (01000011 11000001 00000000 00100000) (5
36920387)
7   12 1 boolean A.flag false
8   13 3 (loss due to the next object alignment)
9  Instance size: 16 bytes
10  Space losses: 0 bytes internal + 3 bytes external = 3 bytes total
11
12  我也不知道要打印什么
13  after lock
14  com.luban.layout.A object internals:
15   OFFSET SIZE TYPE DESCRIPTION VALUE
16   0 4 (object header) 01 00 00 00 (00000001 00000000 00000000 00000000)
(1)
17   4 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000)
(0)
```

```
  8 4 (object header) 43 c1 00 20 (01000011 11000001 00000000 00100000) (536920387)
  12 1 boolean A.flag false
  13 3 (loss due to the next object alignment)
```

分析结果4

上面这个程序只有一个线程去调用sync方法，故而讲道理应该是偏向锁，但是你会发现输出的结果（第一个字节）依然是**00000001**和无锁的时候一模一样，其实这是因为虚拟机在启动的时候对于偏向锁有延迟，比如把上述代码当中的睡眠注释掉结果就会不一样了，结果会变成**00000101**当然为了方便测试我们可以直接通过JVM的参数来禁用延迟-XX:+UseBiasedLocking -XX:BiasedLockingStartupDelay=0，想想为什么偏向锁会延迟？（除了这8bit，其他bit存储了线程信息和epoch，这里不截图了），需要注意的after lock，退出同步后依然保持了偏向信息



性能对比偏向锁和轻量级锁：

```
package com.luban.layout;
public class A {
 int i;
 public synchronized void parse(){
 i++;
 }
}
```

```
package com.luban.layout;
import org.openjdk.jol.info.ClassLayout;
```

```
import static java.lang.System.out;
//-XX:BiasedLockingStartupDelay=20000 -XX:BiasedLockingStartupDelay=0
public class JOLExample3 {
 public static void main(String[] args) throws Exception {
 A a = new A();
 long start = System.currentTimeMillis();
 //调用同步方法1000000000L 来计算1000000000L的++，对比偏向锁和轻量级锁的性能
  //如果不出意外，结果灰常明显
  for(int i=0;i<1000000000L;i++){
  a.parse();
  }
  long end = System.currentTimeMillis();
  System.out.println(String.format("%sms", end - start));

  }

}
```

那么问题来了，什么是轻量级锁呢？工作原理是什么呢？为什么比偏向锁慢？轻量级锁尝试在应用层面解决线程同步问题，而不触发操作系统的互斥操作，轻量级锁减少多线程进入互斥的几率，不能代替互斥

首先看一下轻量级锁的对象头

```
package com.luban.layout;
import org.openjdk.jol.info.ClassLayout;
import static java.lang.System.out;

public class JOLExample5 {
 static A a;
 public static void main(String[] args) throws Exception {
 a = new A();
 out.println("befre lock");
  out.println(ClassLayout.parseInstance(a).toPrintable());
  sync();
  out.println("after lock");
  out.println(ClassLayout.parseInstance(a).toPrintable());
  }

  public static void sync() throws InterruptedException {
```

```
 synchronized (a){
 out.println("lock ing");
 out.println(ClassLayout.parseInstance(a).toPrintable());
 }
 }
}
```

性能对比轻量对比重量：

```
package com.luban.layout;

import java.util.concurrent.CountDownLatch;

public class JOLExample4 {
 static CountDownLatch countDownLatch = new CountDownLatch(1000000000);
 public static void main(String[] args) throws Exception {
 final A a = new A();

 long start = System.currentTimeMillis();

 //调用同步方法1000000000L 来计算1000000000L的++，对比偏向锁和轻量级锁的性能
 //如果不出意外，结果灰常明显
 for(int i=0;i<2;i++){
 new Thread(){
 @Override
 public void run() {
 while (countDownLatch.getCount() > 0) {
 a.parse();
 }
 }
 }.start();
 }
 countDownLatch.await();
 long end = System.currentTimeMillis();
 System.out.println(String.format("%sms", end - start));

 }

}
```

偏向	轻量	重量锁
3553	29075	61359

关于重量锁首先看对象头

```
package com.luban.layout;
import org.openjdk.jol.info.ClassLayout;
import static java.lang.System.out;

public class JOLExample6 {
 static A a;
 public static void main(String[] args) throws Exception {
 //Thread.sleep(5000);
 a = new A();
  out.println("befre lock");
  out.println(ClassLayout.parseInstance(a).toPrintable());

  Thread t1= new Thread(){
  public void run() {
  synchronized (a){
  try {
  Thread.sleep(5000);
  System.out.println("t1 release");
  } catch (InterruptedException e) {
  e.printStackTrace();
  }
  }
  }
  };
  t1.start();
  Thread.sleep(1000);
  out.println("t1 lock ing");
  out.println(ClassLayout.parseInstance(a).toPrintable());
  sync();
  out.println("after lock");
  out.println(ClassLayout.parseInstance(a).toPrintable());

```

```
33   System.gc();
34   out.println("after gc()");
35   out.println(ClassLayout.parseInstance(a).toPrintable());
36   }
37
38   public static void sync() throws InterruptedException {
39   synchronized (a){
40   System.out.println("t1 main lock");
41   out.println(ClassLayout.parseInstance(a).toPrintable());
42   }
43   }
44 }
```

如果调用wait方法则立刻变成重量锁

```
1  package com.luban.layout;
2
3  import org.openjdk.jol.info.ClassLayout;
4
5  import static java.lang.System.out;
6
7  public class JOLExample7 {
8   static A a;
9   public static void main(String[] args) throws Exception {
10   //Thread.sleep(5000);
11   a = new A();
12   out.println("befre lock");
13   out.println(ClassLayout.parseInstance(a).toPrintable());
14
15   Thread t1= new Thread(){
16   public void run() {
17   synchronized (a){
18   try {
19   synchronized (a) {
20   System.out.println("before wait");
21   out.println(ClassLayout.parseInstance(a).toPrintable());
22   a.wait();
23   System.out.println(" after wait");
24   out.println(ClassLayout.parseInstance(a).toPrintable());
25   }
26   } catch (InterruptedException e) {
```

```
27   e.printStackTrace();
28   }
29   }
30   }
31   };
32   t1.start();
33   Thread.sleep(5000);
34   synchronized (a) {
35   a.notifyAll();
36   }
37   }
38  }
```

需要注意的是如果对象已经计算了hashcode就不能偏向了

```
1  package com.luban.layout;
2  import org.openjdk.jol.info.ClassLayout;
3  import static java.lang.System.out;
4
5  public class JOLExample8 {
6   static A a;
7   public static void main(String[] args) throws Exception {
8
9   Thread.sleep(5000);
10   a = new A();
11   a.hashCode();
12   out.println("befre lock");
13   out.println(ClassLayout.parseInstance(a).toPrintable());
14
15   Thread t1= new Thread(){
16   public void run() {
17   synchronized (a){
18   try {
19   synchronized (a) {
20   System.out.println("lock ed");
21   out.println(ClassLayout.parseInstance(a).toPrintable());
22   }
```

```
23   } catch (Exception e) {
24   e.printStackTrace();
25   }
26   }
27   }
28   };
29   t1.start();
30
31   }
32 }
```