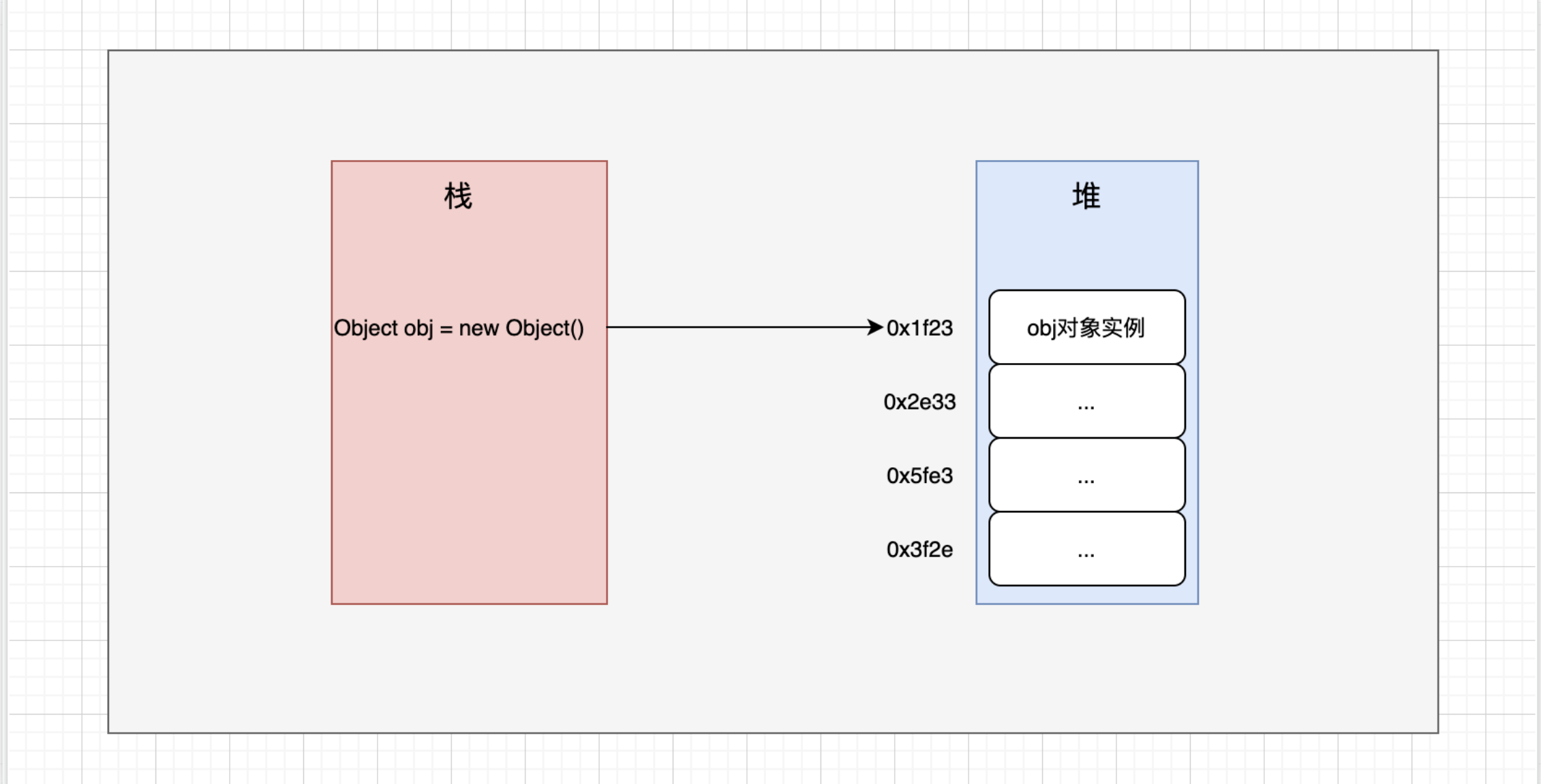
Java对象的内存布局

今天来讲些抽象的东西 -- **对象头**，因为我在学习的过程中发现很多地方都关联到了对象头的知识点，例如JDK中的 synchronized锁优化 和 JVM 中对象年龄升级等等。要深入理解这些知识的原理，了解对象头的概念很有必要，而且可以为后面分享 synchronized 原理和 JVM 知识的时候做准备。

💡 对象内存构成

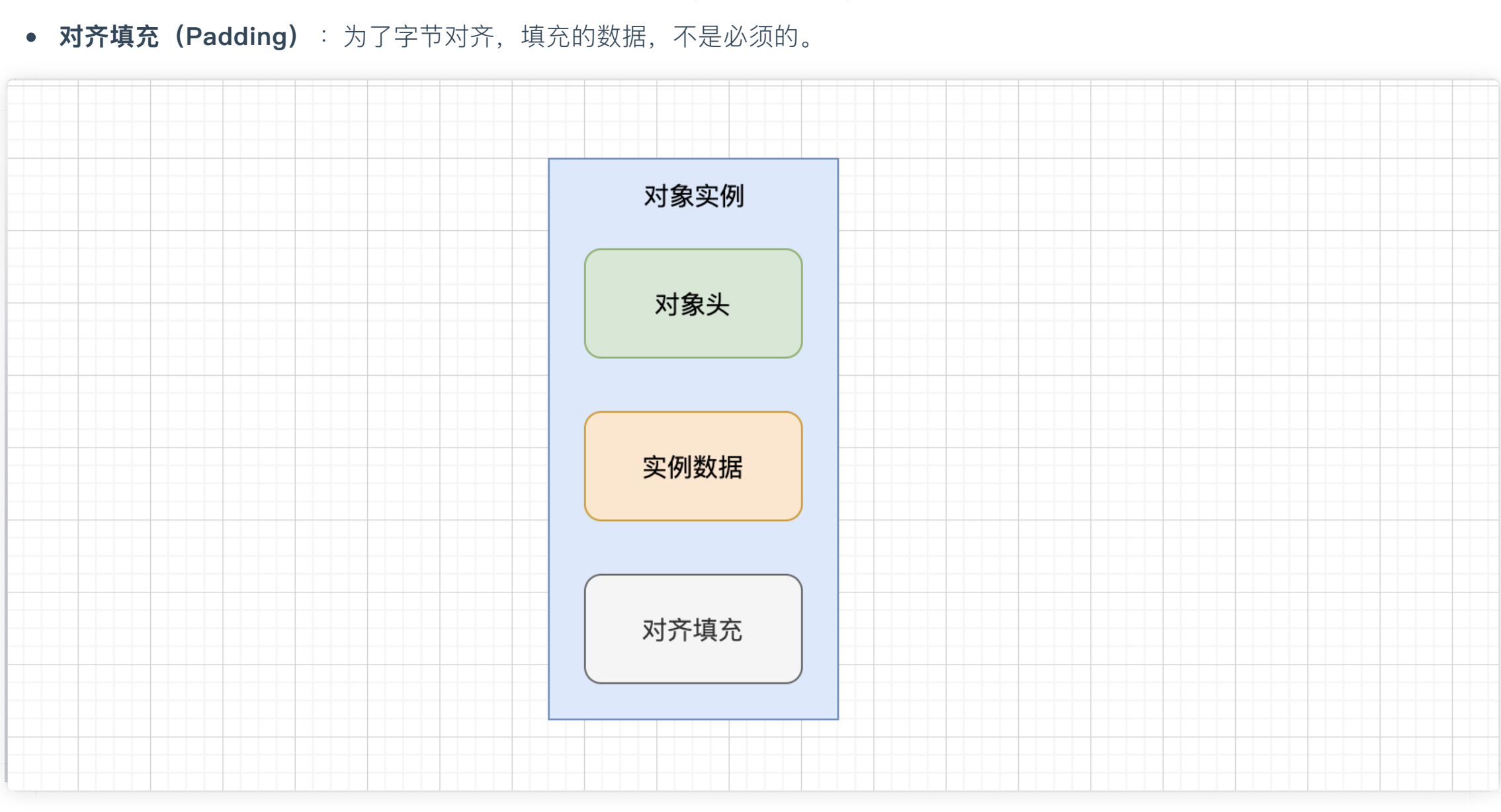
Java 中通过 new 关键字创建一个类的实例对象，对象存于内存的堆中并给其分配一个内存地址，那么是否想过如下这些问题：

- **这个实例对象是以怎样的形态存在内存中的?**
- **一个Object对象在内存中占用多大?**
- **对象中的属性是如何在内存中分配的?**



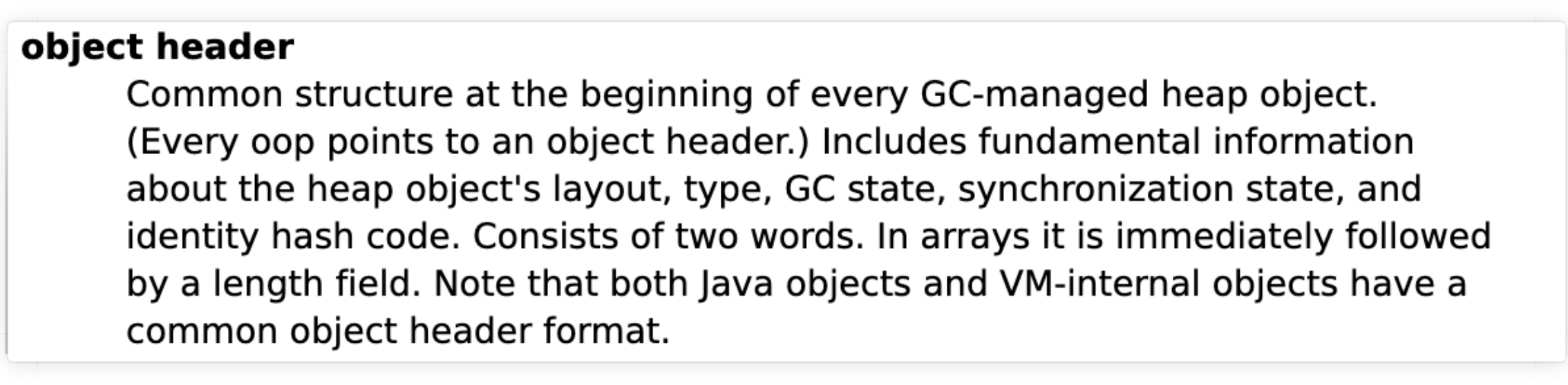
在 JVM 中，Java对象保存在堆中时，由以下三部分组成：

- **对象头（object header）**：包括了关于堆对象的布局、类型、GC状态、同步状态和标识哈希码的基本信息。Java对象和vm内部对象都有一个共同的对象头格式。
- **实例数据（Instance Data）**：主要是存放类的数据信息，父类的信息，对象字段属性信息。
- **对齐填充（Padding）**：为了字节对齐，填充的数据，不是必须的。



对象头

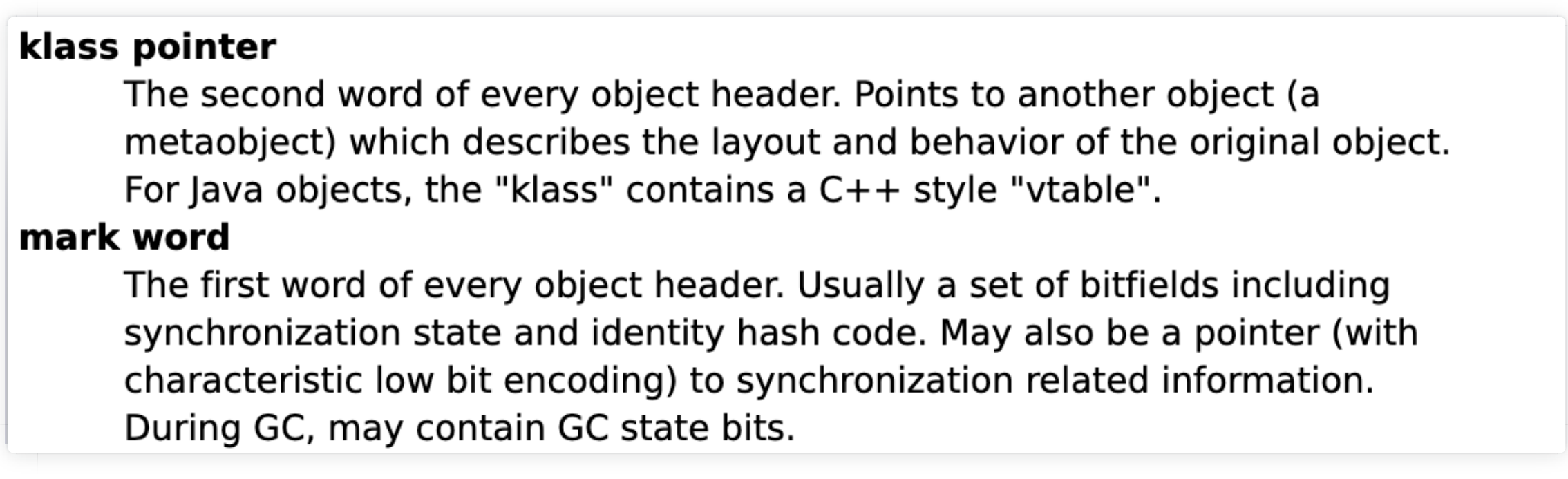
我们可以在Hotspot官方文档中找到它的描述(下图)。从中可以发现，它是Java对象和虚拟机内部对象都有的共同格式，由两个 **字**(计算机术语)组成。另外，如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通Java对象的元数据信息确定Java对象的大小，但是从数组的元数据中无法确定数组的大小。



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.

它里面提到了对象头由两个 **字** 组成，这两个 **字** 是什么呢？我们还是在上面的那个Hotspot官方文档中往上看，可以发现还有另外两个名词的定义解释，分别是 **mark word** 和 **klass pointer**。



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".

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.

从中可以发现对象头中那两个字：第一个字就是 **mark word**，第二个就是 **klass pointer**。

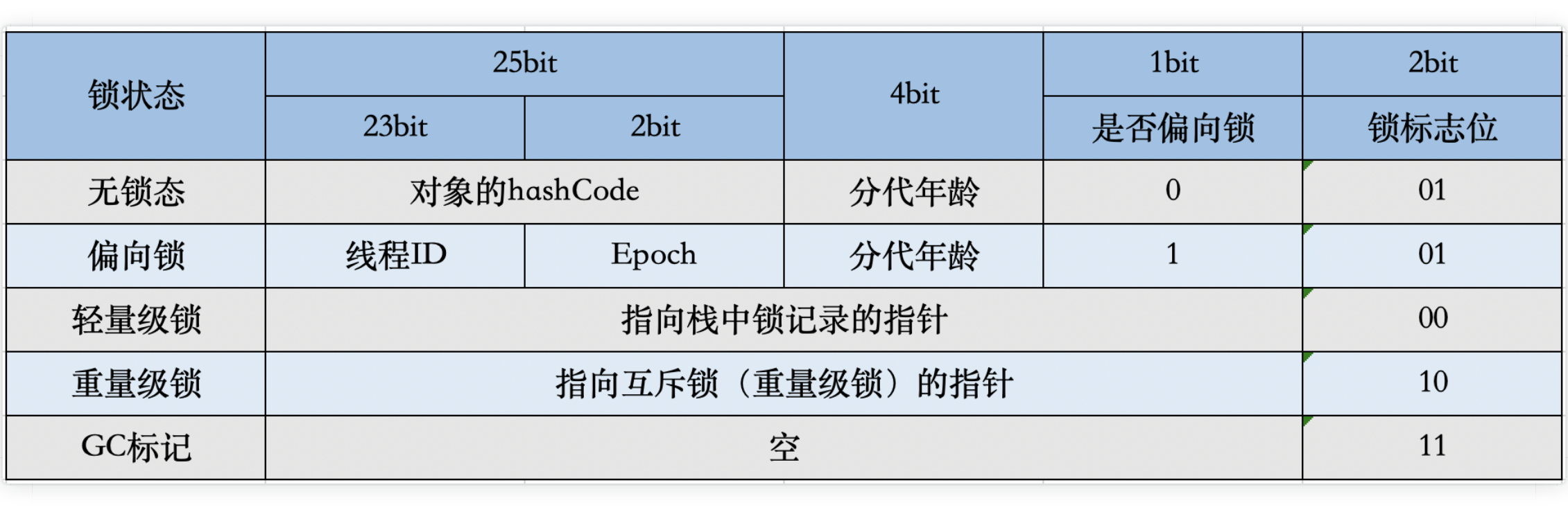
Mark Word

用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等等。

Mark Word在32位JVM中的长度是32bit，在64位JVM中长度是64bit。我们打开openjdk的源码包，对应路径 `/openjdk/hotspot/src/share/vm/oops`，Mark Word对应到C++的代码 `markOop.hpp`，可以从注释中看到它们的组成，本文所有代码是基于Jdk1.8。

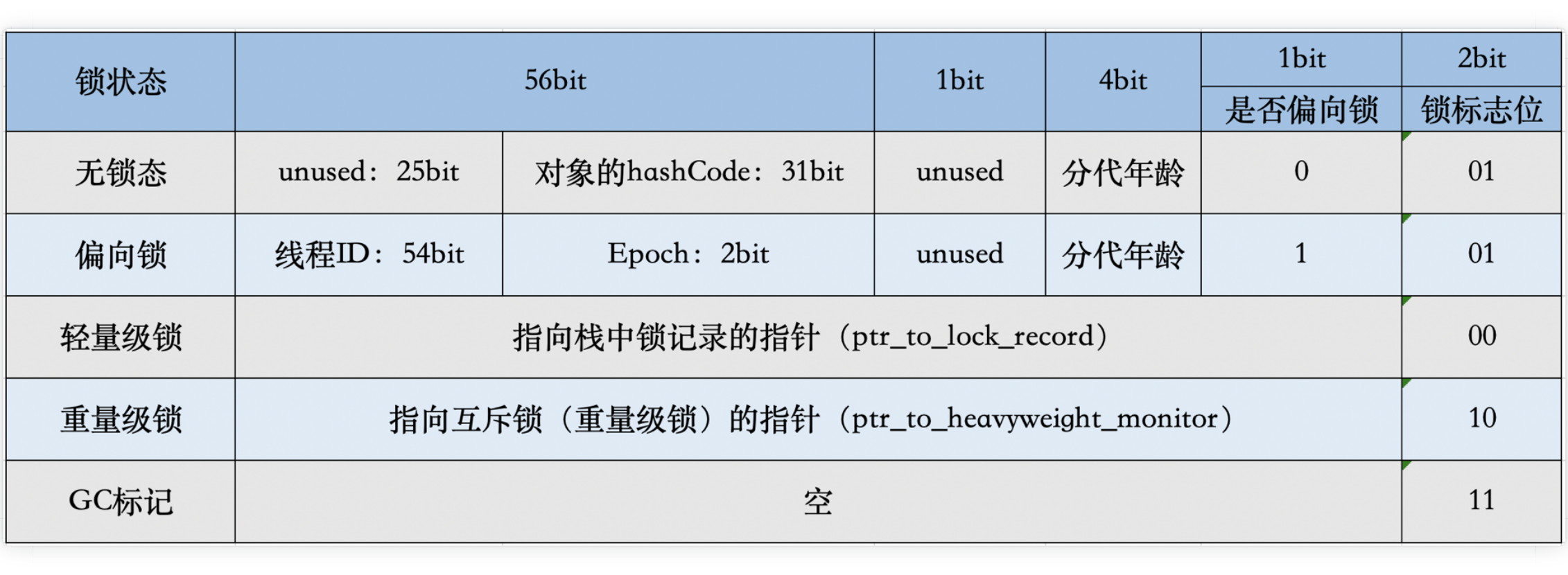
```
// Bit-format of an object header (most significant first, big endian layout below):
//
//  32 bits:
//  --------
//             hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal object)
//             JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object)
//             size:32 ------------------------------------------>| (CMS free block)
//             PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted object)
//
//  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)
//
```

Mark Word在不同的锁状态下存储的内容不同，在32位JVM中是这么存的



锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否偏向锁	锁标志位
无锁态	对象的hashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥锁（重量级锁）的指针				10
GC标记	空				11

在64位JVM中是这么存的



锁状态	56bit		1bit	4bit	1bit	2bit
					是否偏向锁	锁标志位
无锁态	unused：25bit	对象的hashCode：31bit	unused	分代年龄	0	01
偏向锁	线程ID：54bit	Epoch：2bit	unused	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针（ptr_to_lock_record）					00
重量级锁	指向互斥锁（重量级锁）的指针（ptr_to_heavyweight_monitor）					10
GC标记	空					11

虽然它们在不同位数的JVM中长度不一样，但是基本组成内容是一致的。

- **锁标志位（lock）**：区分锁状态，11时表示对象待GC回收状态，只有最后2位锁标识(11)有效。
- **biased_lock**：是否偏向锁，由于无锁和偏向锁的锁标识都是 01，没办法区分，这里引入一位的偏向锁标识位。
- **分代年龄（age）**：表示对象被GC的次数，当该次数到达阈值的时候，对象就会转移到老年代。
- **对象的hashcode（hash）**：运行期间调用System.identityHashCode()来计算，延迟计算，并把结果赋值到这里。当对象加锁后，计算的结果31位不够表示，在偏向锁，轻量锁，重量锁，hashcode会被转移到Monitor中。
- **偏向锁的线程ID（JavaThread）**：偏向模式的时候，当某个线程持有对象的时候，对象这里就会被置为该线程的ID。 在后面的操作中，就无需再进行尝试获取锁的动作。
- **epoch**：偏向锁在CAS锁操作过程中，偏向性标识，表示对象更偏向哪个锁。
- **ptr_to_lock_record**：轻量级锁状态下，指向栈中锁记录的指针。当锁获取是无竞争的时，JVM使用原子操作而不是OS互斥。这种技术称为轻量级锁定。在轻量级锁定的情况下，JVM通过CAS操作在对象的标题字中设置指向锁记录的指针。
- **ptr_to_heavyweight_monitor**：重量级锁状态下，指向对象监视器Monitor的指针。如果两个不同的线程同时在同一个对象上竞争，则必须将轻量级锁定升级到Monitor以管理等待的线程。在重量级锁定的情况下，JVM在对象的ptr_to_heavyweight_monitor设置指向Monitor的指针。

Klass Pointer

即类型指针，是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

实例数据

如果对象有属性字段，则这里会有数据信息。如果对象无属性字段，则这里就不会有数据。根据字段类型的不同占不同的字节，例如boolean类型占1个字节，int类型占4个字节等等；

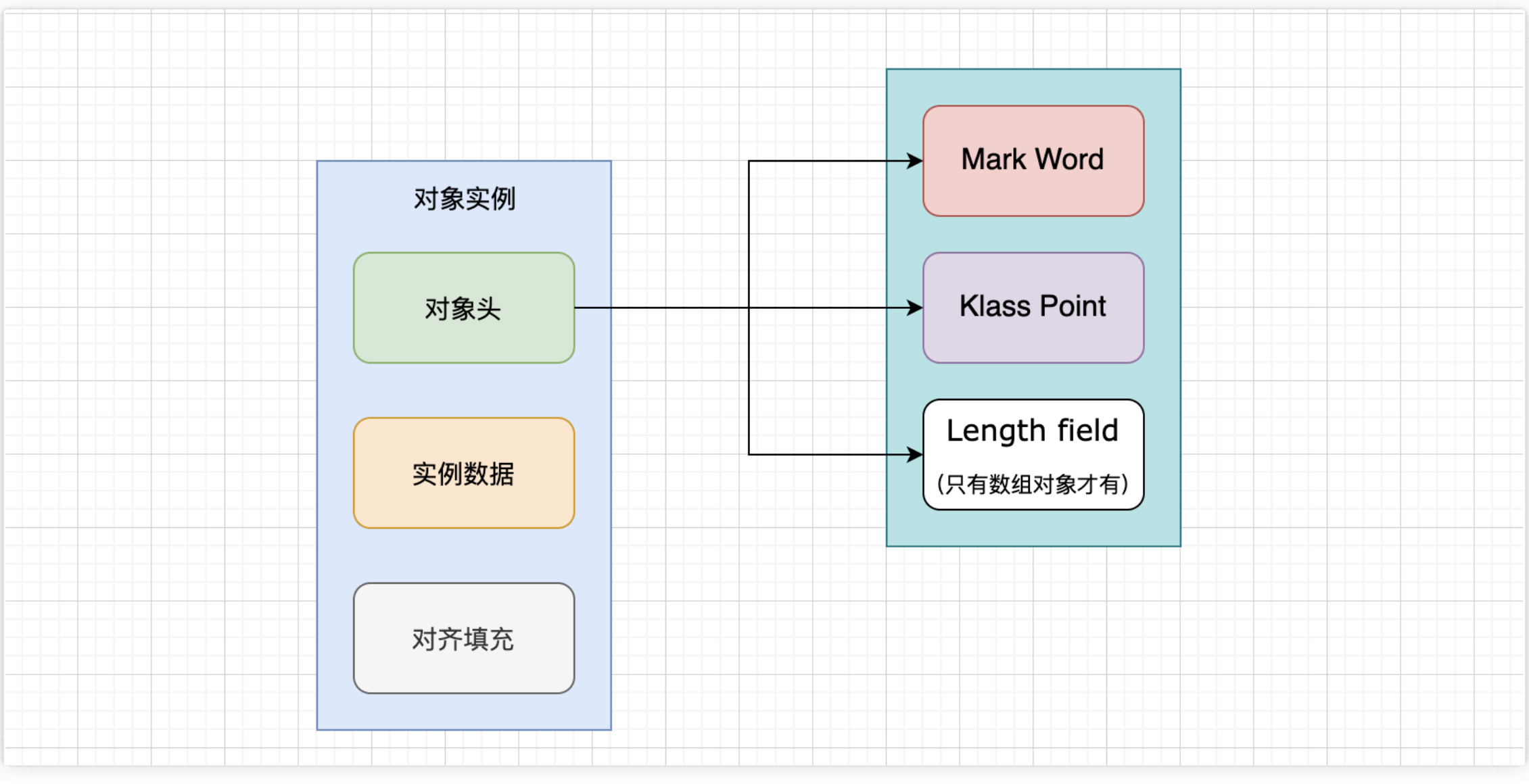
对齐数据

对象可以有对齐数据也可以没有。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个字节则需要对其填充，以此来补齐对象头和实例数据占用内存之后剩余的空间大小。如果对象头和实例数据已经占满了JVM所分配的内存空间，那么就不用再进行对齐填充了。

所有的对象分配的字节总SIZE需要是8的倍数，如果前面的对象头和实例数据占用的总SIZE不满足要求，则通过对齐数据来填满。

为什么要对齐数据？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。其实对其填充的最终目的是为了计算机高效寻址。

至此，我们已经了解了对象在堆内存中的整体结构布局，如下图所示



💡 Talk is cheap, show me code

概念的东西是抽象的，你说它是这样组成的，就真的是吗？学习是需要持怀疑的态度的，任何理论和概念只有自己证实和实践之后才能接受它。还好 openjdk 给我们提供了一个工具包，可以用来获取对象的信息和虚拟机的信息，我们只需引入 jol-core 依赖，如下

```
<dependency>
  <groupId>org.openjdk.jol</groupId>
  <artifactId>jol-core</artifactId>
  <version>0.8</version>
</dependency>
```

jol-core 常用的三个方法

- `ClassLayout.parseInstance(object).toPrintable()`：查看对象内部信息.
- `GraphLayout.parseInstance(object).toPrintable()`：查看对象外部信息，包括引用的对象.
- `GraphLayout.parseInstance(object).totalSize()`：查看对象总大小.

普通对象

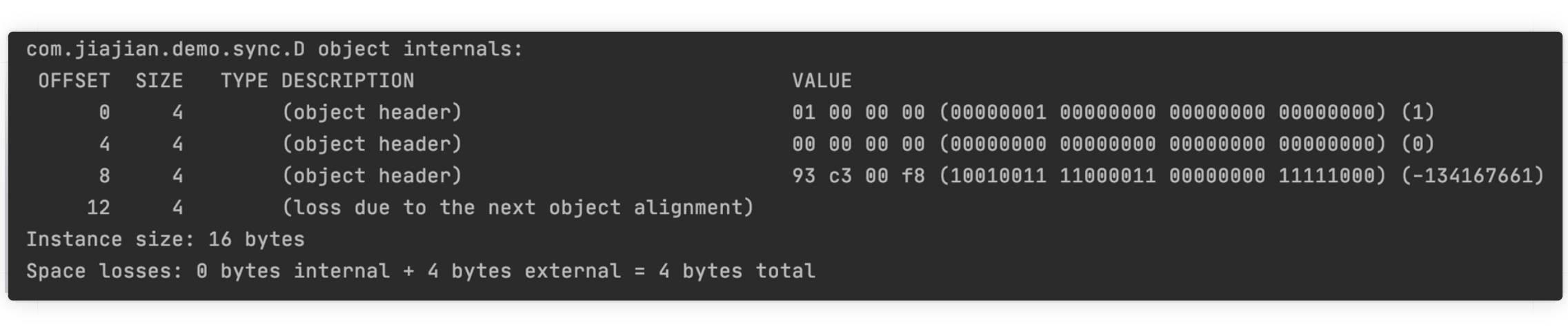
为了简单化，我们不用复杂的对象，自己创建一个类 D，先看无属性字段的时候

```
public class D {
}
```

通过 jol-core 的 api，我们将对象的内部信息打印出来

```
public static void main(String[] args) {
    D d = new D();
    System.out.println(ClassLayout.parseInstance(d).toPrintable());
}
```

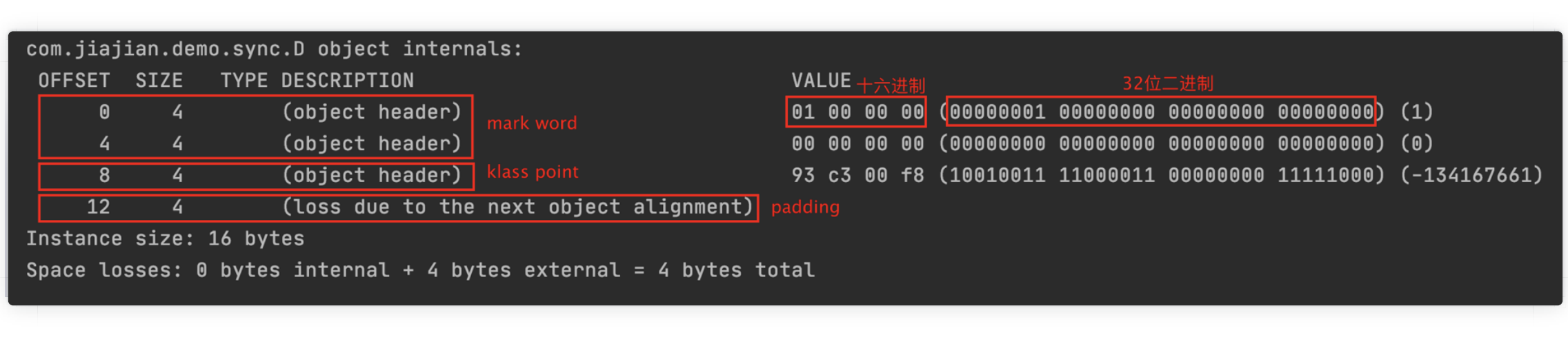
最后的打印结果为



```
com.jiajian.demo.sync.D 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)                           93 c3 00 f8 (10010011 11000011 00000000 11111000) (-134167661)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

可以看到有 OFFSET、SIZE、TYPE DESCRIPTION、VALUE 这几个名词头，它们的含义分别是

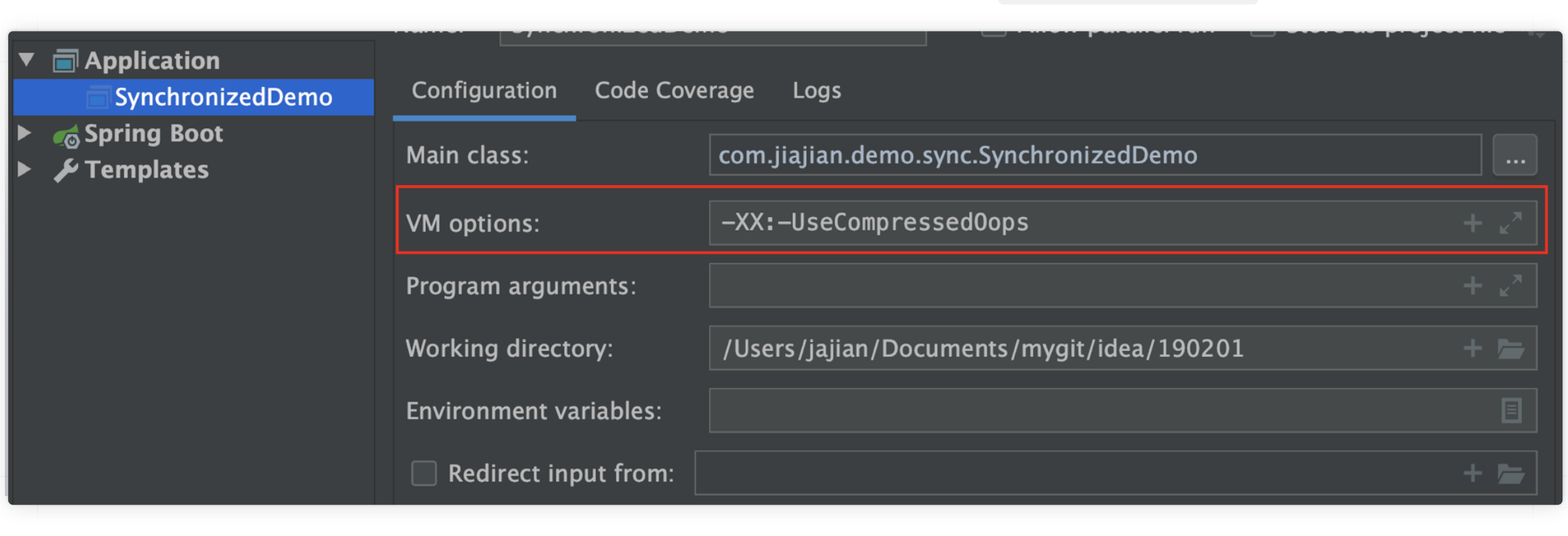
- OFFSET：偏移地址，单位字节；
- SIZE：占用的内存大小，单位为字节；
- TYPE DESCRIPTION：类型描述，其中object header为对象头；
- VALUE：对应内存中当前存储的值，二进制32位；



```
com.jiajian.demo.sync.D 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)                           93 c3 00 f8 (10010011 11000011 00000000 11111000) (-134167661)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

可以看到，d对象实例共占据16byte，对象头（object header）占据12byte（96bit），其中 mark word占8byte（64bit），klass pointe 占4byte，另外剩余4byte是填充对齐的。

这里由于默认开启了 **指针压缩**，所以对象头占了12byte，具体的指针压缩的概念这里就不再阐述了，感兴趣的读者可以自己查阅下官方文档。jdk8版本是默认开启指针压缩的，可以通过配置vm参数开启关闭指针压缩，`-XX:-UseCompressedOops`。



Main class: com.jiajian.demo.sync.SynchronizedDemo

VM options: -XX:-UseCompressedOops

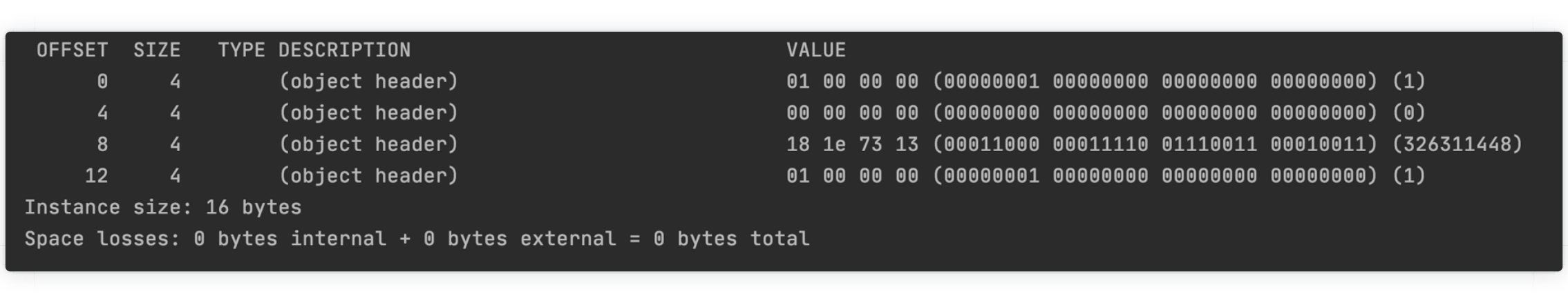
Program arguments:

Working directory: /Users/jajian/Documents/mygit/idea/190201

Environment variables:

☐ Redirect input from:

如果关闭指针压缩重新打印对象的内存布局，可以发现总SIZE变大了，从下图中可以看到，对象头所占用的内存大小变为16byte（128bit），其中 mark word占8byte，klass pointe 占8byte，无对齐填充。



```
 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)                           18 1e 73 13 (00011000 00011110 01110011 00010011) (326311448)
     12     4        (object header)                           01 00 00 00 (00000001 00000000 00000000 00000000) (1)
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

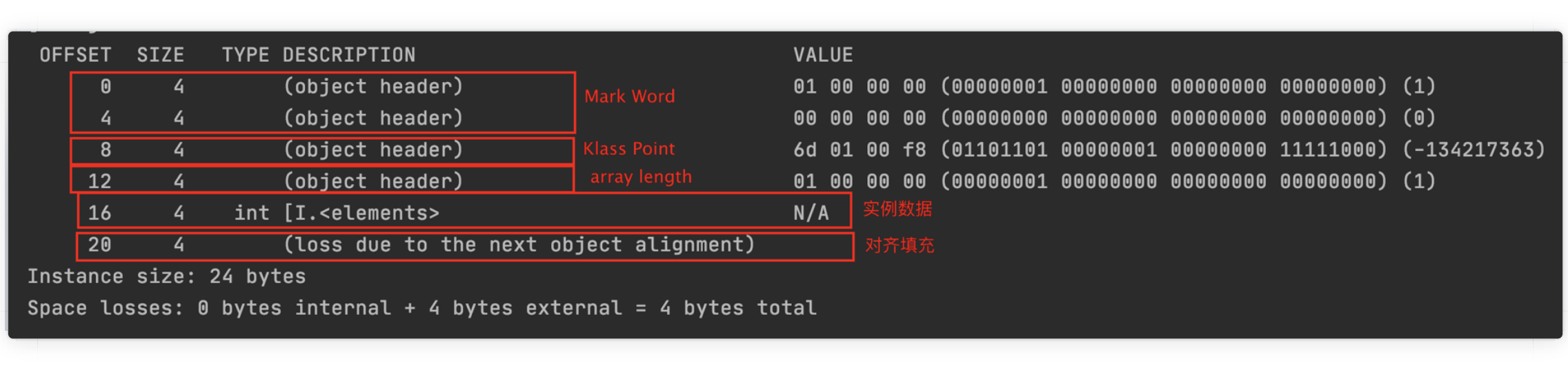
开启指针压缩可以减少对象的内存使用。从两次打印的D对象布局信息来看，关闭指针压缩时，对象头的SIZE增加了4byte，这里由于D对象是无属性的，读者可以试试增加几个属性字段来看下，这样会明显的发现SIZE增长。因此开启指针压缩，理论上来讲，大约能节省百分之五十的内存。jdk8及以后版本已经默认开启指针压缩，无需配置。

数组对象

上面使用的是普通对象，我们来看下数组对象的内存布局，比较下有什么异同

```
public static void main(String[] args) {
    int[] a = {1};
    System.out.println(ClassLayout.parseInstance(a).toPrintable());
}
```

打印的内存布局信息，如下



```
 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)                           6d 01 00 f8 (01101101 00000001 00000000 11111000) (-134217363)
     12     4        (object header)                           01 00 00 00 (00000001 00000000 00000000 00000000) (1)
     16     4    int [I.<elements>                             N/A
     20     4        (loss due to the next object alignment)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

可以看到这时总SIZE为共24byte，对象头占16byte，其中Mark Work占8byte，Klass Point 占4byte，array length 占4byte，因为里面只有一个int 类型的1，所以数组对象的实例数据占据4byte，剩余对齐填充占据4byte。

💡 结尾

经过以上的内容我们了解了对象在内存中的布局，了解对象的内存布局和对象头的概念，特别是对象头的Mark Word的内容，在我们后续分析 synchronize 锁优化 和 JVM 垃圾回收年龄代的时候会有很大作用。

JVM中大家是否还记得对象在Suvivor中每熬过一次MinorGC，年龄就增加1，当它的年龄增加到一定程度后就会被晋升到老年代中，这个次数默认是15岁，有想过为什么是15吗？在Mark Word中可以发现标记对象分代年龄的分配的空间是4bit，而4bit能表示的最大数就是2^4-1 = 15。

文章持续更新中，可以微信关注公众号『小菜亦牛』，第一时间阅读。点滴积累，厚积薄发，小菜鸟也能成为大牛。

分类：[01].Java基础

标签：Java基础

« 上一篇：线程池其实看懂了也很简单
» 下一篇：MapStruct 解了对象映射的毒

posted @ 2020-09-21 08:47 JaJian 阅读(11015) 评论(2) 编辑 收藏 举报