第 6 章

类变量解析

本章摘要

◎ Java 类变量解析的原理
◎ 计算机基础——偏移量与内存对齐
◎ Java 类与字段的对齐与补白
◎ Java 字段的继承机制
◎ 使用 HSDB 查看运行时的 Java 类结构

JVM 从 Java 类的字节码文件中解析出常量池信息后，便将 Java 类的变量和方法基本信息读取进内存，保存在 constantPoolOop 的 tag 和数据区。但是 tag 与数据区中的数据仍然是非结构化的数据，根据这些数据并不能直观地描述 Java 类结构，因此 JVM 需要进一步解析。这便是本章要讲的主要内容。按照本书的“惯例”，为了加强前后思路的连贯性，防止思路被分支所干扰，下面给出类解析的总体地图，如图 6.1 所示。

本章开始讲解 JVM 解析 Java 类中所定义的变量的技术实现细节。

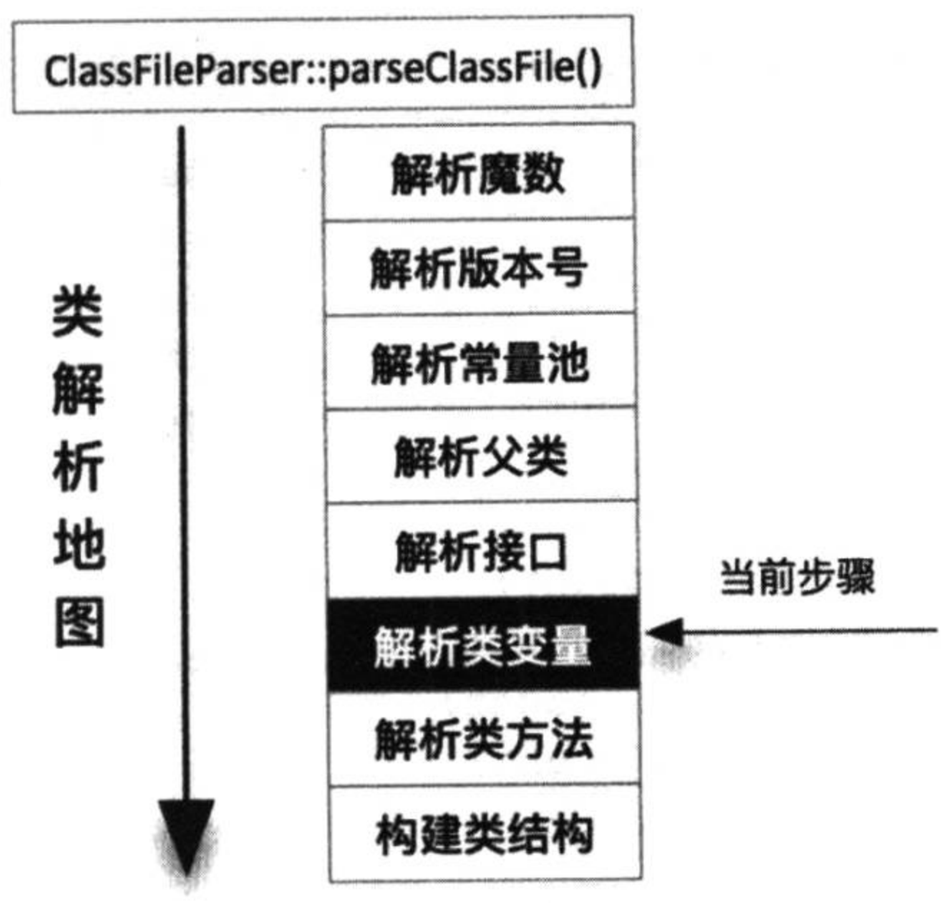

图 6.1　类解析的步骤

6.1　类变量解析

在 ClassFileParser::parseClassFile()函数中，解析完常量池、父类和接口后，接着便调用 parse_fields()函数解析类变量信息：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：解析类变量

```
void ClassFileParser::parseClassFile(constantPoolHandle cp, int length,
TRAPS) {
    // …
    struct FieldAllocationCount fac = {0,0,0,0,0,0,0,0,0,0};
    objArrayHandle fields_annotations;
    typeArrayHandle fields = parse_fields(cp, access_flags.is_interface(),
&fac, &fields_annotations,
        CHECK_(nullHandle));
    //…
}
```

在调用 parse_fields()函数之前，先定义了一个变量 fac，这里的 FieldAllocationCount 是一个结构体类型，声明如下：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：FieldAllocationCount 结构体

```
struct FieldAllocationCount {
  unsigned int static_oop_count;
  unsigned int static_byte_count;
  unsigned int static_short_count;
  unsigned int static_word_count;
  unsigned int static_double_count;
  unsigned int nonstatic_oop_count;
  unsigned int nonstatic_byte_count;
  unsigned int nonstatic_short_count;
  unsigned int nonstatic_word_count;
  unsigned int nonstatic_double_count;
};
```

通过声明可知，FieldAllocationCount 结构体类型的变量实例将会记录 5 种静态（static）类型变量的数量和 5 种对应的非静态类型的变量的数量，这 5 种变量类型分别是：

- Oop，引用类型
- Byte，字节类型
- Short，短整型
- Word，双字类型
- Double，浮点型

在 parse_fields()函数里面，会分别统计 static 和非 static 的这 5 种变量的数量，后面 JVM 为 Java 类分配内存空间时，会根据这些变量的数量计算所占内存大小。可能有小伙伴会问：Java 语言所支持的实际数据类型远不止这 5 种呀，例如 boolean、float、int，等等，为什么 JVM 只统计这 5 种呢？这是因为在 JVM 内部，除了引用类型，所有内置的基本类型都使用剩余的 4 种类型来表示，因此想知道一个 Java 类的域变量需要占用多少内存，只需要分别统计这几种类型的数量即可。

下面一起看看 JVM 对 Java 类域变量的具体解析逻辑：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：ClassFileParser::parse_fields()

```
typeArrayHandle ClassFileParser::parse_fields(constantPoolHandle cp, bool
is_interface,
                                              struct FieldAllocationCount *fac,
                                              objArrayHandle*  fields_annotations,
TRAPS) {
    //…
    // 获取 Java 类域变量的数量
```

```
u2 length = cfs->get_u2_fast();

int index = 0;
typeArrayHandle field_annotations;
for (int n = 0; n < length; n++) {
  //读取变量的访问标识，例如private|public等
  jint flags = cfs->get_u2_fast() & JVM_RECOGNIZED_FIELD_MODIFIERS;
  //…

  //读取变量名称索引
  u2 name_index = cfs->get_u2_fast();
  //…

  //读取类型索引
  u2 signature_index = cfs->get_u2_fast();
  //…

  //读取变量属性
  u2 attributes_count = cfs->get_u2_fast();
  //…

  fields->short_at_put(index++, access_flags.as_short());
  fields->short_at_put(index++, name_index);
  fields->short_at_put(index++, signature_index);
  fields->short_at_put(index++, constantvalue_index);

  //判断变量类型
  BasicType type = cp->basic_type_for_signature_at(signature_index);
  FieldAllocationType atype;
  if ( is_static ) {
    switch ( type ) {
      case  T_BOOLEAN:
      case  T_BYTE:
        fac->static_byte_count++;
        atype = STATIC_BYTE;
        break;
      case  T_LONG:

      //…

      default:
        assert(0, "bad field type");
    }
  }

  fields->short_at_put(index++, atype);
  fields->short_at_put(index++, 0);
```

```
        fields->short_at_put(index++, generic_signature_index);
    }
```

上面对 ClassFileParser::parse_fields()的主要逻辑进行了注释，通过注释可以知道，JVM 解析 Java 类域变量的逻辑是：

- ◎ 获取 Java 类中的变量数量。
- ◎ 读取变量的访问标识。
- ◎ 读取变量名称索引。
- ◎ 读取变量类型索引。
- ◎ 读取变量属性。
- ◎ 判断变量类型。
- ◎ 统计各类型数量。

在 Java 类所对应的字节码文件中，有专门的一块区域保存 Java 类的变量信息（在前文详细讲解过），字节码文件会依次描述各个变量的访问标识、名称索引、类型索引和属性。由于每个变量的访问标识、名称索引、类型索引和属性数量都占用 2 字节（u2），因此在这段逻辑中，只需依次调用 cfs->get_u2_fast()即可。

解析完一个变量的属性后，调用 fields->short_at_put()函数将属性保存到 fields 所代表的内存区域中。fields 是在 ClassFileParser::parse_fields()方法一开始就申请的内存，如下：

```
u2 length = cfs->get_u2_fast();
typeArrayOop new_fields =
    oopFactory::new_permanent_shortArray(
        length*instanceKlass::next_offset, CHECK_(nullHandle));
```

如果你非常认真地研究了前文所讲的常量池对象 constantPoolOop 的内存申请与分配机制，那么这里对于 typeArrayOop 的内存分配机制自然一看就明白，因此这里不关注其实现的具体过程。但是，有一个问题却非常值得关注：这里到底申请了多大的内存？

oopFactory::new_permanent_shortArray()函数的第一个入参决定了最终所分配的内存大小，而该入参并没有直接给出，而是一个表达式：

```
length * instanceKlass::next_offset
```

length 自然是指 Java 类中的变量数量，而 instanceKlass::next_offset 在 instanceKlass.hpp 文件中定义，是一个枚举：

清单：/src/share/vm/oops/instanceKlass.hpp
功能：FieldOffset 枚举定义

```
enum FieldOffset {
```

```
    access_flags_offset    = 0,
    name_index_offset      = 1,
    signature_index_offset = 2,
    initval_index_offset   = 3,
    low_offset             = 4,
    high_offset            = 5,
    generic_signature_offset = 6,
    next_offset            = 7
  };
```

由该定义可知，instanceKlass::next_offset 的值为 7。

由此也可知，最终所申请的内存大小是 7 * length，由于 length 类型是 u2，占 2 字节，因此所申请的内存大小实际上相当于 14 个变量所占用的总字节。假设 Java 类中一共有 5 个变量，则 JVM 需要为其申请 14 * 5 = 70 字节的内存空间，来存放变量的属性。

这里仍然有一点需要注意，字节码文件对 Java 类变量的描述维度与 JVM 内存中的映像并不相同，字节码文件仅仅描述了变量的访问标识、名称索引、类型索引、属性，而 JVM 内存映像除此之外，还描述了每一个变量的偏移量和泛型索引。而关于偏移量的话题，正是下一节的主题。

在遍历解析各个变量属性的循环后面，JVM 会取出每一个变量的类型进行判断，并根据类型来统计上文所提到的 5 大类型（静态和非静态各 5 种类型）的总数量。

6.2　偏移量

解析完字节码文件中 Java 类的全部域变量信息后，JVM 计算出 5 种数据类型各自的数量，并据此计算各个变量的偏移量。

6.2.1　静态变量偏移量

先看静态类型变量，其逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：静态变量类型定义

```
    typeArrayHandle ClassFileParser::parseClassFile(constantPoolHandle cp, bool
is_interface,
                                                    struct FieldAllocationCount *fac,
                                                    objArrayHandle*  fields_annotations,
TRAPS) {
```

```
    //…
    //定义各种偏移量
    int static_field_size = 0;
    int next_static_oop_offset;
    int next_static_double_offset;
    int next_static_word_offset;
    int next_static_short_offset;
    int next_static_byte_offset;
    int next_static_type_offset;

    //…

    //获取起始偏移量
    next_static_oop_offset     = instanceMirrorKlass::offset_of_static_fields();

    //计算 5 大类型的静态变量的偏移量
    next_static_double_offset   = next_static_oop_offset +
                                (fac.static_oop_count * heapOopSize);
    /…
    next_static_word_offset     = next_static_double_offset +
                                (fac.static_double_count * BytesPerLong);
    next_static_short_offset    = next_static_word_offset +
                                (fac.static_word_count * BytesPerInt);
    next_static_byte_offset     = next_static_short_offset +
                                (fac.static_short_count * BytesPerShort);
    next_static_type_offset     = align_size_up((next_static_byte_offset +
                                fac.static_byte_count ), wordSize );

    //计算全部静态变量所占的字节数
    static_field_size           = (next_static_type_offset -
                                next_static_oop_offset) / wordSize;

    //…
}
```

这段逻辑很简单，先拿到起始偏移量，接着分别根据各个静态类型变量的偏移量计算总偏移量。计算顺序是：

- static_oop
- static_double
- static_word
- static_short
- static_byte

每一种“下游”数据类型的偏移量都依赖其“上游”数据类型所占的字节宽度及数量。

JVM 为何要记录每一个变量的偏移量呢？其实，这与静态变量的存储机制和访问机制有关。对于 JDK 1.6 而言，静态变量存储在 Java 类在 JVM 中所对应的镜像类 Mirror 中，当 Java 代码访问静态变量时，最终 JVM 也是通过设置偏移量进行访问。

6.2.2　非静态变量偏移量

相比于静态变量偏移量，非静态变量的偏移量计算稍显复杂。逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：ClassFileParser::parse_fields()

```
    typeArrayHandle ClassFileParser::parseClassFile(constantPoolHandle cp, bool
is_interface,
                                                    struct FieldAllocationCount *fac,
                                                    objArrayHandle*  fields_annotations,
TRAPS) {
      //…
      //获取父类非静态字段大小（以字节为单位）
      int nonstatic_field_size = super_klass() == NULL ? 0 :
super_klass->nonstatic_field_size();

      //…

      //…
      //①.计算非静态字段起始偏移量
      first_nonstatic_field_offset = instanceOopDesc::base_offset_in_bytes() +
                                     nonstatic_field_size * heapOopSize;
      next_nonstatic_field_offset = first_nonstatic_field_offset;

      //…
      //②.计算 nonstatic_double_offset 和 nonstatic_oop_offset 这 2 种非静态类型变量的
起始偏移量
      //分配策略不同，这 2 种变量的起始偏移量也不同
      if( allocation_style == 0 ) {
         // Fields order: oops, longs/doubles, ints, shorts/chars, bytes
         next_nonstatic_oop_offset    = next_nonstatic_field_offset;
         //…
      }

      //…
      //处理压缩类型
      if( nonstatic_double_count > 0 ) {
         int offset = next_nonstatic_double_offset;
         next_nonstatic_double_offset = align_size_up(offset, BytesPerLong);
         if( compact_fields && offset != next_nonstatic_double_offset ) {
```

```
            int length = next_nonstatic_double_offset - offset;
            //…
        }

        //…
        //③.计算剩余 3 种类型变量的起始偏移量：nonstatic_word_offset,
nonstatic_short_offset, nonstatic_byte_offset
        next_nonstatic_word_offset  = next_nonstatic_double_offset +
                                      (nonstatic_double_count * BytesPerLong);
        next_nonstatic_short_offset = next_nonstatic_word_offset +
                                      (nonstatic_word_count * BytesPerInt);
        next_nonstatic_byte_offset  = next_nonstatic_short_offset +
                                      (nonstatic_short_count * BytesPerShort);

        //④.计算对齐补白空间
        int notaligned_offset;
          if( allocation_style == 0 ) {
            notaligned_offset = next_nonstatic_byte_offset + nonstatic_byte_count;
          } else { // allocation_style == 1
            next_nonstatic_oop_offset = next_nonstatic_byte_offset +
nonstatic_byte_count;
            if( nonstatic_oop_count > 0 ) {
              next_nonstatic_oop_offset = align_size_up(next_nonstatic_oop_offset,
heapOopSize);
            }
            notaligned_offset = next_nonstatic_oop_offset + (nonstatic_oop_count *
heapOopSize);
          }
          next_nonstatic_type_offset = align_size_up(notaligned_offset,
heapOopSize );

        //⑤.计算补白后非静态变量所需要的内存空间总大小
        nonstatic_field_size = nonstatic_field_size + ((next_nonstatic_type_offset
                                - first_nonstatic_field_offset)/heapOopSize);

        //…
        //5 大类型变量的偏移量都计算完，现在开始遍历所有字段，分别计算各类型具体变量的偏移值，这
里包括静态的和非静态的
        int len = fields->length();
          for (int i = 0; i < len; i += instanceKlass::next_offset) {
            int real_offset;
            FieldAllocationType atype = (FieldAllocationType) fields->ushort_at(i +
                          instanceKlass::low_offset);
            switch (atype) {
              case STATIC_OOP:
                real_offset = next_static_oop_offset;
                next_static_oop_offset += heapOopSize;
```

```
        //…
      }

      //保存偏移量
      fields->short_at_put(i + instanceKlass::low_offset,
extract_low_short_from_int(real_offset));
      fields->short_at_put(i + instanceKlass::high_offset,
                                    extract_high_short_from_int(real_offset));
    }

  //…  //计算 oop_map_count
  const unsigned int total_oop_map_count =
     compute_oop_map_count(super_klass, nonstatic_oop_map_count,
                           first_nonstatic_oop_offset);

  //…
}
```

如上面代码所注释的那样，非静态类型变量的偏移量计算逻辑可以分为 5 个步骤：

（1）计算非静态变量起始偏移量。

（2）计算 nonstatic_double_offset 和 nonstatic_oop_offset 这 2 种非静态类型变量的起始偏移量。

（3）计算剩余 3 种类型变量的起始偏移量：nonstatic_word_offset、nonstatic_short_offset 和 nonstatic_byte_offset。

（4）计算对齐补白空间。

（5）计算补白后非静态变量所需要的内存空间总大小。

JVM 的内存管理模型在所有编程语言中属于比较复杂的一种，而对于一个给定的 Java 类，其主要的内存空间便是其非静态类型的变量所占用的空间（另一部分是虚拟方法分发表），因此理解了这部分内存分配策略，JVM 的内存管理模型便理解了一半，故这部分内容十分重要。下面分析非静态类型变量偏移量的计算逻辑。

1. 计算非静态变量起始偏移量

要计算非静态变量的偏移量，首先需要知道从哪里开始偏移。

本节多次讲到了偏移量，那么什么是偏移量呢？对于非静态类型的变量，其偏移量是相对于未来即将 new 出来的 Java 对象实例在 JVM 内部所对应的 instanceOop 对象实例首地址的偏移位置。

前面描述常量池对象时提到过，在 JVM 内部，使用 oop-klass 这种一分为二的模型去描述

一个对象，常量池对象本身便是这种模型。对于 Java 类，JVM 内部同样使用这种一分为二的模型去描述，对应的 oop 类是 instanceOopDesc，对应的 klass 类是 instanceKlass。在 oop-klass 模型中，oop 模型主要存储对象实例的实际数据，而 klass 模型则主要存储对象的结构信息和虚函数方法表，说白了就是描述类的结构和行为。

当 JVM 加载一个 Java 类时，会首先构建对应的 instanceKlass 对象，而当 new 一个 Java 对象实例时，则会构建出对应的 instanceOop 对象。instanceOop 对象主要由 Java 类的成员变量组成，而这些成员变量在 instanceOop 中的排列顺序，便由各种变量类型的偏移量决定。

在 HotSpot 内部，任何 oop 对象都包含对象头，因此实际上非静态变量的偏移量要从对象头的末尾开始计算。Java 类实例在堆内存中的起始偏移量如图 6.2 所示。

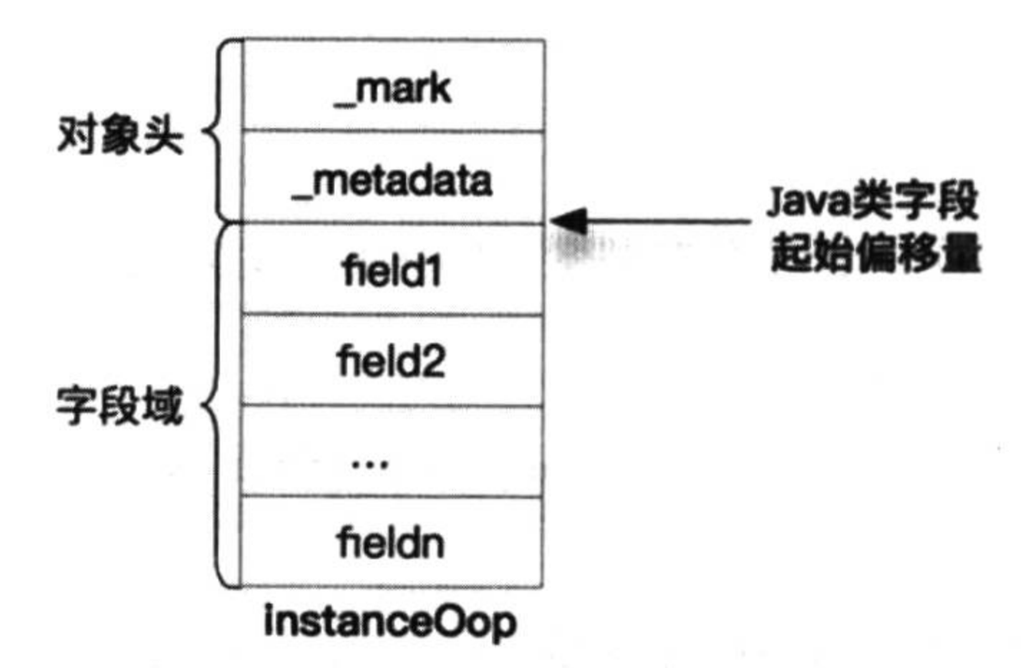

图 6.2　Java 类字段的起始偏移量

知道了偏移量的含义，现在来看看 JVM 的实现。源代码里有如下逻辑：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：计算非静态变量起始偏移量

```
 typeArrayHandle ClassFileParser::parseClassFile(constantPoolHandle cp, bool
is_interface,
                                                struct FieldAllocationCount *fac,
                                                objArrayHandle*  fields_annotations,
TRAPS) {
    //…
    //获取父类非静态字段大小（以字节为单位）
    int nonstatic_field_size = super_klass() == NULL ? 0 :
super_klass->nonstatic_field_size();

    //…
    //计算非静态字段起始偏移量
    first_nonstatic_field_offset = instanceOopDesc::base_offset_in_bytes() +
                                   nonstatic_field_size * heapOopSize;
     //…
```

```
}
```

这里首先调用 instanceOopDesc::base_offset_in_bytes()方法，该方法实现如下：

清单：/src/share/vm/oops/instanceOop.hpp
功能：计算对象头大小

```
static int base_offset_in_bytes() {
    return UseCompressedOops ?
            klass_gap_offset_in_bytes() :
            sizeof(instanceOopDesc);
}
```

如果没有启用压缩策略，则最终返回 instanceOopDesc 类型大小。instanceOopDesc 继承自 oopDesc，而 oopDesc 类型大小在前文已经计算出来了，是两个指针宽度。假设 JVM 运行在 64 位架构上，则这个值是 16。而如果启用了压缩策略，则在 64 位架构上，该值为 12，这是因为无论是否开启压缩策略，oop._mark 作为一个指针是不会被压缩的，任何时候都会占用 8 字节，而 oop._klass 则会受压缩策略是否开启的影响，若开启压缩策略，则_klass 仅会占用 4 字节，所以在 64 位架构上开启压缩策略的情况下，oop 对象头总共占用 12 字节的内存空间。

Java 类是面向对象的，继承是面向对象编程的 3 大特性之一，除了 java.lang.Object 类以外，所有的 Java 类都显式或隐式地继承了某个父类，而字段继承和方法继承则构成了继承的全部内涵。

如果说继承是目标，那么字段在子类中的内存占用则是技术手段。子类必须将父类中所定义的非静态字段信息全部复制一遍，才能实现字段继承的目标。因此，在计算子类非静态字段的起始偏移量时，必须将父类可被继承的字段的内存大小考虑在内。具体而言，子类的非静态字段起始偏移量，在计算完 oop 对象头的大小后，还需要为父类的可被继承的字段预留空间。

Hotspot 将父类可被继承的字段的内存空间安排在子类所对应的 oop 对象头的后面，因此最终一个 Java 类中非静态字段的起始偏移位置在父类被继承的字段域的末尾，如图 6.3 所示。

2. 内存对齐与字段重排

在上面第一步计算出 Java 类字段的起始偏移量后，接下来就能基于这个起始偏移量计算出 Java 类中所有字段的偏移量。不过在这之前，需要先研究这样一个问题——内存对齐。

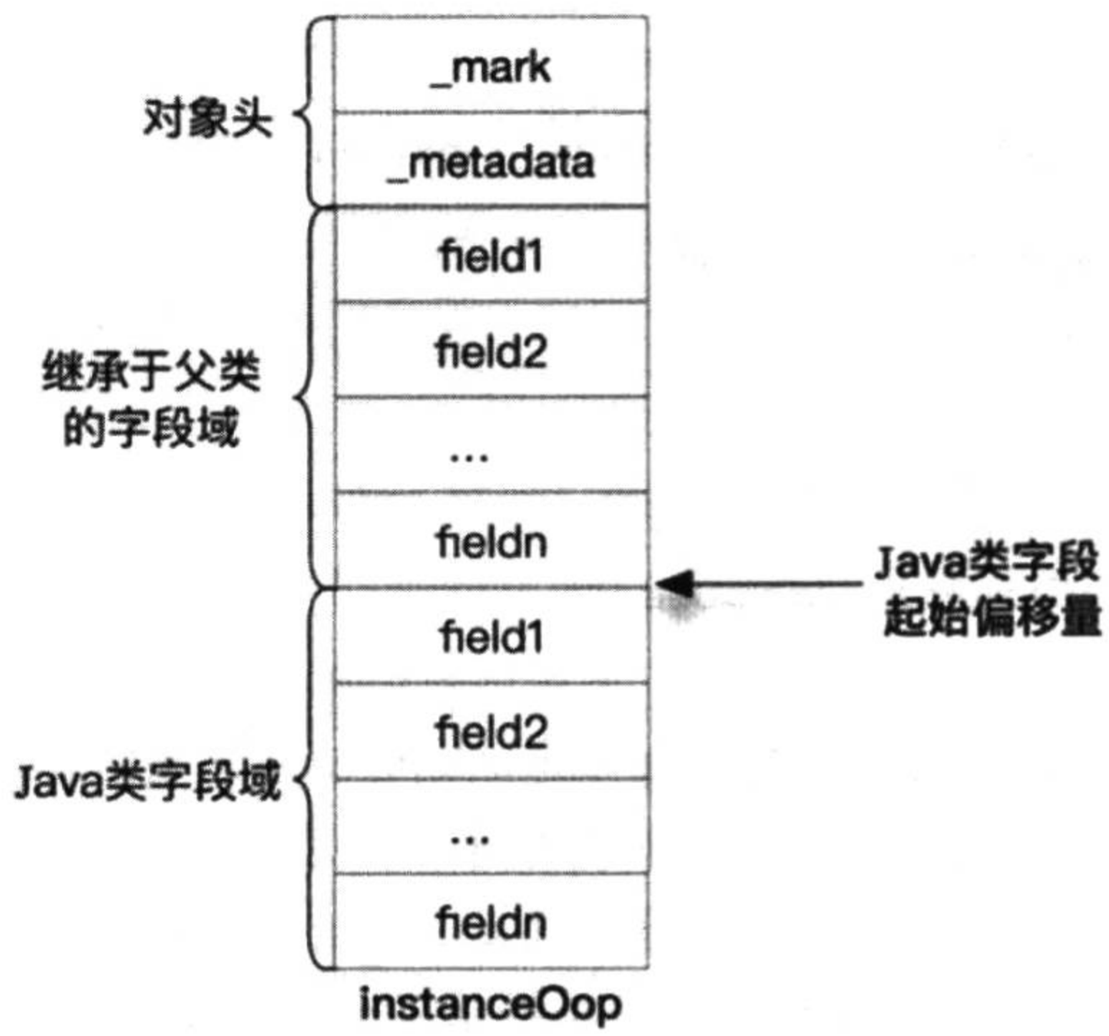

图 6.3　Java 类字段的起始偏移量

因为 Java 类字段的偏移地址与内存对齐有脱不开的关系，JVM 为了处理内存对齐，颇费了一番心思，甚至不惜将字段进行重排。

1）什么是内存对齐

内存对齐与数据在内存中的位置有关。如果一个变量的内存起始地址正好等于其长度的整数倍，则这种内存分配就被称作自然对齐。

在 32 位 x86 平台上，基本的对齐规则如表 6.1 所示。

表 6.1　32 位 x86 平台上的内存对齐规则

变量类型	长　度	对齐规则
char	1 字节	按 1 字节对齐
short	2 字节	按 2 字节对齐
int	4 字节	按 4 字节对齐

举个例子，在 32 位 CPU 下，假设一个 int 整型变量的内存地址为 0x00000008，那么该变量就是自然对齐的。

2）为什么要字节对齐

现代计算机中内存空间都是按照字节划分的，也即内存的粒度细分到存储单元，而一个存储单元恰恰包含 8 个比特，正好是 1 字节(byte)，因此从理论上讲似乎对任何类型的变量的访问可以从任何地址开始。

但实际情况恰恰相反，各个硬件平台在对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。例如有些架构的 CPU 在访问一个没有进行对齐的变量时会发生错误，那么在这种架构下进行编程就必须保证字节对齐。

例如，在某些硬件架构上，如果一个 int 整型变量的内存地址为 0x00000003，那么当在程序中对该变量进行读写时，硬件就会报错。要验证 CPU 硬件平台是否支持非对齐状态的变量读写，只需通过如下一个 C 程序进行测试：

清单：示例程序
作用：验证 CPU 硬件平台对内存对齐的支持

```
char ch[2];

char *p = &ch[0];
int i = *(int *)p;

p = &ch[1];
i = *(int *)p;
```

这段程序很简单，先声明一个大小为 2 的 char 类型数组，接着将该数组的第 1 个元素（从 1 开始计数）的内存地址传递给指针 p，再通过变量 i 从数组的第 1 个元素所在的内存位置开始连续读取 4 字节（即一个 int 类型的宽度）。接着再从数组的第 2 个元素所在的内存位置开始连续读取 4 字节数据。

如果 ch 数组的首地址恰好是 4 字节的整数倍，则第一次读取 4 字节能够成功，但是第二次一定会失败，因为第二次读取时，起始地址一定不是 4 字节的整数倍。

某些平台会支持非对齐内存位置的数据读写，硬件不会抛出异常，但是最常见的是，如果不按照其平台要求对数据存放进行对齐，会在存取效率上有所损失。比如有些平台每次读都是从偶地址开始，如果一个 int 型（假设为 32 位系统）存放在偶地址开始的地方，那么一个读周期就可以读出这 32 比特，而如果存放在奇地址开始的地方，就需要 2 个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该 32 比特数据。显然在读取效率上下降很多。

例如，假设一个整型变量的内存地址不是 4 字节对齐，其内存位置位于 0x00000002，则 CPU 需要进行两次内存访问才能读取到该变量的值：

- 第一次从 0x00000002 和 0x00000003 这两个连续的内存存储单元中读取一个 short 类型宽度的数据。
- 第二次则从 0x00000004 和 0x00000005 这两个连续的内存单元中再读取一个 short 类型宽度的数据。

将这 2 次读取的数据进行组合之后才能得到所要的数据。

如果该变量的内存位置是 0x00000003，则 CPU 可能需要 2 次甚至 3 次内存访问才能将该变量的值读取出来。CPU 访问两次的方案是：

◎ 第一次从 0x00000001~0x00000004 这 4 个连续的内存单元中读取一个 int 类型宽度的数据。

◎ 第二次则从 0x00000005~0x00000008 这 4 个连续的内存单元中读取一个 int 类型宽度的数据。

将两次内存访问的结果剔除首尾多出来的字节，拼凑出目标结果。

而 CPU 访问三次的方案是：

◎ 第一次从 0x00000003 这个内存单元上读取一个 char 类型宽度的数据。

◎ 第二次从 0x00000004 和 0x00000005 这两个连续的内存单元中读取一个 short 类型宽度的数据。

◎ 第三次则从 0x00000006 这个内存单元上读取一个 char 类型宽度的数据。

三次都读取完成后，再将这 3 次读取的结果进行组合得到整型数据。

如果这样的程序工作在多线程环境中，则还需要进行总线级别的原子操作，以防止出现脏数据，从而造成程序的严重错误。

而如果这个整型变量的内存地址是自然对齐的，则 CPU 只需要一次内存访问就能读取数据，并且还是原子性的，效率和安全性无疑是最高的。

所以，由于以上 2 种情况，一种是程序健壮性，一种是高性能，均需要各种类型数据按照一定的规则在空间上排列，而不是顺序地一个一个地排放，从而对数据进行对齐。

3）什么时候需要考虑对齐

无论是 C/C++这样的编译性语言，还是 Java/C#这样的高级语言，一般情况下都不需要开发者去考虑对齐的问题，因为编译器或者虚拟机会自动将数据进行对齐补白。不过如果你是在设计底层协议或者开发硬件驱动程序，这时候就必须要考虑对齐的事情了。

下面这个简单的例子能够验证 C/C++的编译器 gcc 对内存对齐的支持：

清单：示例程序
作用：gcc 编译器自动处理数据对齐示例

```
#include <stdio.h>

    void test();

    int main(){
```

```
 test(6);
}

void test(int x){
 char c = 65;
 short s = 16;
 int i = 8;
 printf("%d\n", x + i);
}
```

本例很简单，在 test()函数里声明和定义了 3 种不同类型的变量，之所以要声明这几个不同类型的变量，就是为了测试 gcc 编译器的对齐处理能力。

使用 gcc 编译器将这段 C 程序编译成汇编程序，如下所示：

清单：示例程序
作用：gcc 编译器自动处理数据对齐示例

```
main:
    pushq   %rbp
    movl$6, %edi
    callq   _test
test:
    pushq   %rbp
    subq$32, %rsp

    movl%edi, -4(%rbp)//获取入参
    movb$65, -5(%rbp)//为变量 c 赋值
    movw$16, -8(%rbp)//为变量 s 赋值
    movl$8, -12(%rbp)//为变量 i 赋值

    movl-4(%rbp), %edi
    addl-12(%rbp), %edi
    movl%edi, -16(%rbp)         ## 4-byte Spill
    movq%rax, %rdi
    movl-16(%rbp), %esi         ## 4-byte Reload
    movb$0, %al
    callq   _printf
    movl%eax, -20(%rbp)         ## 4-byte Spill
    addq$32, %rsp
    popq%rbp
    retq
```

在这段汇编程序的 test 段中，中间有几条分别为 test()函数中的局部变量进行赋值的指令，其中变量 c 的堆栈位置是–5(%rbp)，因为 c 是字节类型，因此只占 1 个内存存储单元。变量 c 之后的局部变量是 short 类型的 s，由于 short 类型占 2 个内存存储单元，因此按常理 s 的堆栈位

置应该是–7(%rbp)，但是编译器却将变量分配在了–8(%rbp)这个位置，其实在这里，编译器便是对变量进行了对齐处理。由于 short 类型的宽度是 2 字节，因此其内存首地址必须是 2 字节的整数倍，如果将变量 s 的偏移地址放在–7(%rbp)这个位置，很显然不符合自然对齐的原则。

下面这个结构体的例子则很经典，能够使你在不了解汇编语言的情况上，直观地观察到编译器对内存对齐处理的支持：

清单：示例程序
作用：C 语言的 struct 内存对齐

```
#include <stdio.h>

struct A
    {
     int a;
        char b;
        short c;
};
struct B
    {
        char b;
        int a;
        short c;
};

int main(){
     int a = sizeof(struct A);
     int b = sizeof(struct B);
     printf("%d\n", a);
     printf("%d\n", b);
}
```

在这段程序中，声明了 2 个结构体类型 A 和 B，注意，这两个结构体中所包含的变量数量和变量类型完全一致，唯一不一致的是变量的声明顺序。

结构体所占的内存大小，往往是其中所有变量所占内存的总和，按照这个逻辑，则本例中的两个结构体的大小应该都是一样大小。在 32 位机器上，以上几种数据类型的长度如下：

◎ Char，1（有符号无符号都相同）
◎ Short，2（有符号无符号都相同）
◎ Int，4（有符号无符号都相同）

a 是 int 类型，宽度是 4 字节；b 是 char 类型，宽度是 1 字节；c 是 short 类型，宽度是 2 字节。所以这两个结构体所占的内存大小应该是 7 字节。然而结果并不是这样。运行 main()函数，

打印的结果如下：

sizeof(strcut A)的值为 8。

sizeof(struct B)的值却是 12。

很奇怪，两个结构体的大小竟然没有一个是 7。这是因为 gcc 编译器对变量进行了内存对齐。

对于结构体 A，其变量类型的顺序是 int->char->short，在内存分配时，先为 int 类型的变量 a 分配 4 字节内存。接着为 char 类型的 b 变量分配内存。由于 char 仅占 1 字节内存，因此其实并无内存对齐要求，所以变量 b 的内存可以直接跟在变量 a 之后。假设变量 a 的内存起始地址标记为 4x（即 4 字节的整数倍），则 b 的内存地址为 4(x+1)（为了方便描述问题，暂时忘记堆栈空间的增长方向到底是往高低址还是往低地址吧）。

接着为 short 类型的 c 变量分配内存。如果不考虑内存对齐，则变量 c 应当位于变量 b 的后面，则变量 c 的内存地址应该为 4(x+1)+1。但是由于 short 类型的数据的对齐原则是按 2 字节对齐，而 4(x+1)+1 这样的内存地址很显然并不能被 2 整除，不符合按 2 字节对齐的要求，所以编译器就自作主张，将其又往后移动了一个字节，将变量 c 分配在 4(x+1)+2 的内存地址。这样一来，给人的感觉是 char 类型的变量 b 似乎占据了 2 个内存单元。其实变量 b 并没有占据 2 个存储单元，仍然只需要一个存储单元的内存空间，而其后面的另一个多出来的存储单元是用于对齐的补白空间。因此，最终 struct A 需要占用的内存空间总大小就是 8 字节。其内存空间布局如图 6.4 所示。

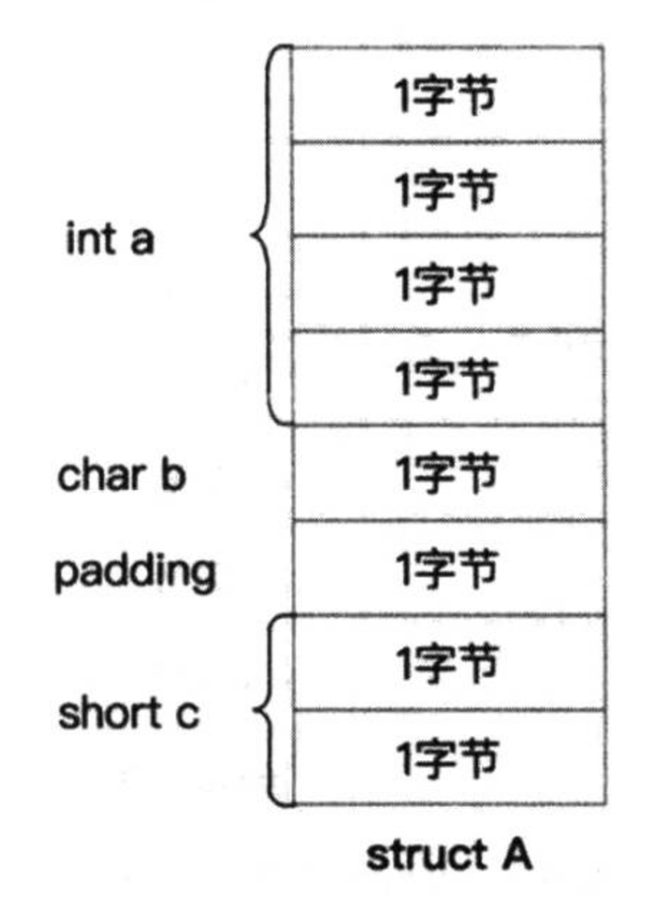

图 6.4　struct A 的内存布局

再来分析 struct B。其变量类型的顺序为 char->int->short。分配内存时，先为 char 类型的 b 变量分配内存空间。b 变量只需要一个存储单元便足够了。

接着为 int 类型的变量 a 分配内存。假设变量 b 的内存地址为 4x（即 4 字节的整数倍），这里解释一下，变量 b 的内存地址之所以也是 4 字节的整数倍，这是由于编译器不仅保证各种变量、结构、复合结构的数据类型是内存对齐的，还会保证堆、堆栈的内存地址也是自然对齐的。所以无论结构体 B 被分配在哪里，其内存起始地址一定是 4 的整数倍。假设变量 a 的内存位置紧跟在变量 b 的后面，则 a 的内存地址应该是 4x+1（为了方便描述问题，暂时忘记堆栈空间的增长方向到底是往高地址还是往低地址吧），因为变量 b 只需要 1 个内存单元即可。但是由于内存对齐的需要，变量 a 需要按 4 字节对齐，所以肯定不能分配在 4x+1 这个内存位置，那怎么办呢？很简单，往后移动 3 个字节，补齐至 4 字节即可。移动 3 字节后的内存位置是 4(x+1)，这下能被 4 整除了，所以就分配在这里了。

接着为 short 类型的变量 c 分配内存。由于变量 a 占 4 字节内存空间，因此变量 c 的起始内存地址自然就是相对于变量 a 的起始地址 4(x+1)再往后移动 4 个字节，所以变量 c 的起始内存地址变成了 4(x+2)。

现在，内存布局是如图 6.5 所示的样子。

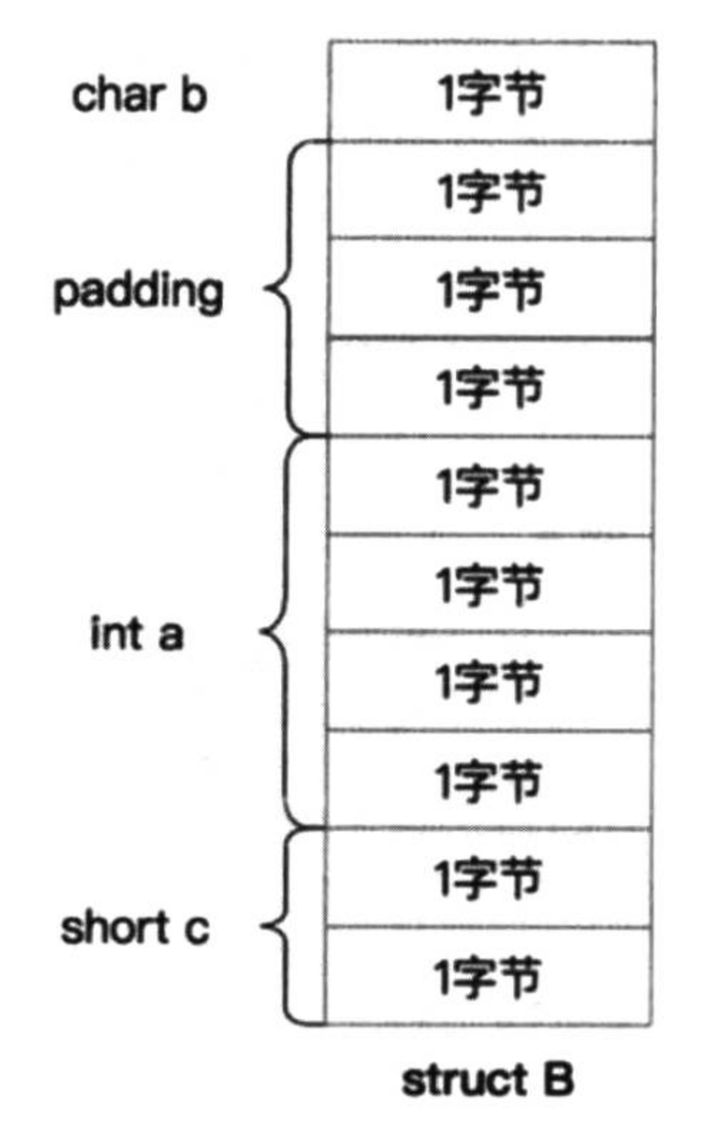

图 6.5　struct B 的内存布局

在这种内存布局下，struct B 总共占有 10 字节内存空间。但是在此刻，内存对齐的规则又发挥作用，具体发挥的对象就是 struct B 类型本身。

在编译原理中，不仅是基本的数据类型要求做到自然对齐，连结构体这样的复合结构类型也需要整体进行自然对齐。其实，数据类型的对齐并不是为了方便自己，而是为了方便别人。例如，struct B 中的 char 类型的变量 b 后面补白了 3 字节，是为了让紧跟其后的 int 类型的 a 变量能够自然对齐。同理，struct B 也需要考虑让其后面的变量能够自然对齐。由于 struct B 也不知道其后面会跟什么类型的变量，因此便按照默认的 4 字节进行对齐。这么做使得后续在处理数据对齐时的逻辑变得简单。

由于编译器要将 struct 结构体按照 4 字节对齐，因此最终 struct B 的内存空间会被扩展到 12 字节，最终的内存布局如图 6.6 所示。

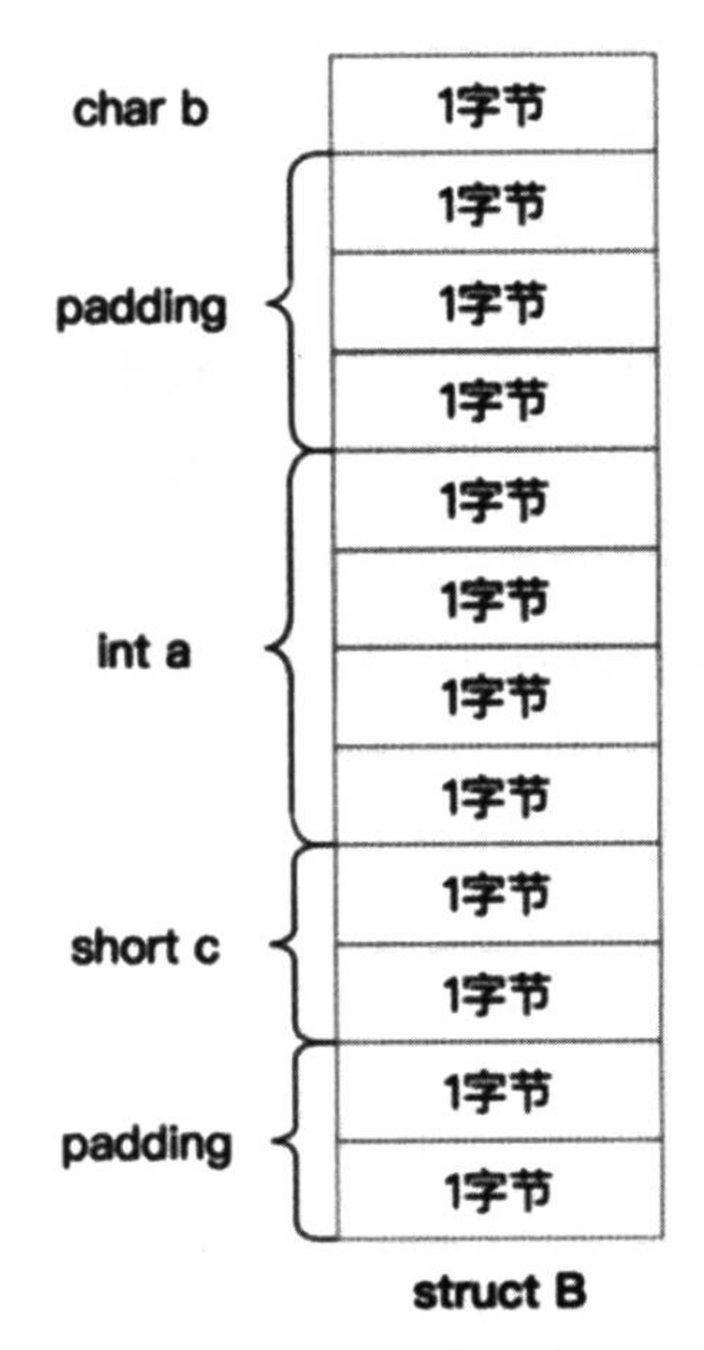

图 6.6　struct B 整体对齐后的内存布局

4）人工优化结构体内存空间

通过上面的两个例子可以看到，即使是如 c/c++这样接近底层的编程语言，平时在开发中也不需要关注内存对齐的问题，因为编译器可以帮我们搞定一切。但是上面那 2 个结构体的例子却向我们透漏出一个强烈的信号：虽然编译器会帮我们处理好内存对齐的问题，但是编译器并不是万能的，并不会使用除了内存对齐以外的其他优化技巧。上面例子中的 A 和 B 两个结构体所包含的成员项是完全相同的，所不同的仅仅是成员项的声明顺序不同而已，结果就造成两者在内存空间利用率上的巨大差异。

假设结构体的成员项数量增多，并且声明完全是无序的，那么内存的利用率将更加糟糕，例如下面这个结构体：

清单：示例程序
作用：C 语言的 struct 内存对齐

```
struct A
    {
        char b;
        int a;

        char b2;
        int a2;

        char b3;
        int a3;
};
```

像这种结构体，将会占用 24 个内存单元。对于这种糟糕的内存使用率，程序员完全可以主动优化，例如，更改下结构体成员项的声明顺序，变成如下：

清单：示例程序
作用：C 语言的 struct 内存对齐

```
struct A
    {
        char b;
        char b2;
        char b3

        int a;
        int a2;
        int a3;
    };
```

修改后的结构体只需要占用 16 个内存单元，比优化之前的结构体一下子少占用 8 字节的内存空间，这种内存空间的节省量相当可观。

请记住这个示例，下面讲解 JVM 的内存分配策略时，与此会有很大关系。

所以，在进行 C/C++程序开发时，一个优秀的程序员还是应当主动关注内存对齐的问题。在这方面，一种可以遵循的设计技巧是：在定义结构体类型中的成员项时，按照类型大小从小到大依次声明，如此便能够尽量减少中间的填补空间。

除了这种设计技巧，还有一种主动的以空间换时间的策略，显式地声明填补空间，例如对上面例子中的 A 结构体进行如下处理：

```
struct A
{
    int a;
   char b;
   char reserved;//声明一个补白字节
   short c;
};
```

在变量 b 后面多声明一个成员项，其类型也是 char。这个成员对程序没有明显的意义，仅仅起到填补空间以达到字节对齐的目的。当然，即使不加这个成员项，编译器也会自动填补对齐，我们自己加上它只是起到显式的提醒作用。

假设 A 结构体的结构是这样的：

```
struct A
{
   char a;
   int b;
};
```

由于变量 b 是 int 类型，需要按 4 字节进行对齐，因此编译器会自动在变量 a 的后面填补 3 字节以对齐。如果你不够信赖编译器，可以自行添加 3 字节的补白空间，添加方式就是声明一个元素数量为 3 的 char 类型的数组。修改后的 A 结构体如下：

```
struct A
{
   char a;
   char reserved[3];
   int b;
};
```

5）Java 类字段对对齐的要求

Java 类的字段最终也是要保存到堆内存中的，也需要被 CPU 频繁地读写，并且 Java 类的变量类型最终也会映射成 CPU 硬件平台架构所能支持的数据类型，因此 Java 类字段在内存中的位置也必须满足自然对齐的要求。尤其是 Java 语言作为一门跨平台的编程语言，可以运行在众多的硬件平台上，更应该兼容各种异构平台对数据对齐的要求，否则万一运行在某个硬件平台上，而该平台压根儿不支持非对齐内存的访问，那么 Java 虚拟机很可能就会可怜地因为这个低级的原因而直接崩溃。

因此，JVM 在设计上就必须使其内部 5 大类型的数据都满足内存对齐的原则。Java 类中包含八大基本数据类型，每种数据类型所占用的内存空间大小总体上与 C/C++中的基本数据类型的宽度相等，如表 6.2 所示。

表 6.2　Java 类基本数据类型所占用的内存大小

Java 基本数据类型	占用空间(bit)	占用空间(byte)
boolean	8	1
byte	8	1
char	16	2
short	16	2
int	32	4
float	32	4
long	64	8
double	64	8

除了这 8 种基本数据类型，还有另外一种类型，就是引用。引用数据类型所占用的内存大小与不同的硬件平台以及是否开启压缩策略有关。在 32 位平台上，引用数据类型占用 4 字节内存空间，而在 64 位平台上，如果开启了压缩策略，则占用 4 字节内存空间，否则便占用 8 字节内存空间。

Java 语言中的 8 种基本数据类型与引用类型对应 JVM 内部所定义的 5 大数据类型。它们的对应关系如表 6.3 所示。

表 6.3　Java 数据类型与 JVM 内部 5 种数据类型的对应关系

Java 基本数据类型	JVM 内部数据类型	数 据 宽 度
引用类型	oop	4 字节/8 字节
boolean/byte	byte	1 字节
char/short	short	2 字节
int/float	word	4 字节
long/double	double	8 字节

JVM 必须确保其内部这 5 种基本数据类型都能够自然对齐，即确保其内存地址能够被其所占用的字节宽度所整除。解决数据类型自然对齐的手段无非是内存补白，JVM 自然也少不了这一手（事实上除了这法子也没别的法子了），但是 JVM 却技高一筹，不仅使用了补白，还祭出了另一件法宝——字段重排。

JVM 的字段重排策略主要包括下面 2 点：

◎ 将相同类型的字段组合在一起。

◎ 按照 double->word->short->byte-oop 的顺序依次分配。

第一点，将相同类型的字段组合在一起，究其原因，是因为这样更能节省内存空间。还记得上面用 C 语言写的那个结构体的例子吗？使用 C 语言编写一个结构体时，结构体中成员项声明的顺序不同，则结构体的大小也随之不同。这种规律对 Java 字段同样适用。看下面这个简单的 Java 类：

清单：示例程序

作用：Java 语言的内存对齐

```
class A{
    byte b;
    long l;
    byte b2;
    int i;
}
```

如果没有字段重排，则 JVM 为了让各种类型的字段做到自然对齐，最终只能按照如图 6.7 所示这种补白的方式来分配内存（省略 oop 对象头）。

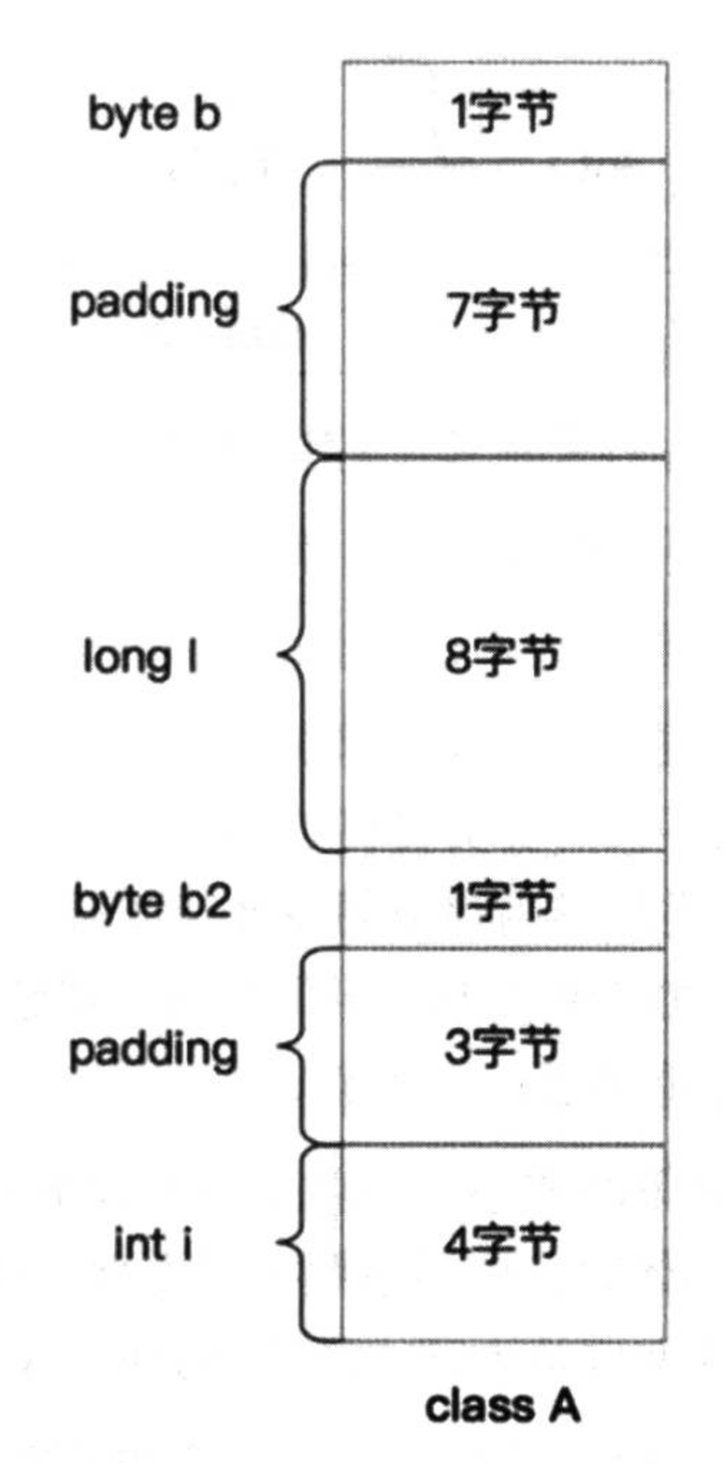

图 6.7　未进行字段重排时类 A 的内存布局

这样的内存布局需要占用 24 字节的内存空间。

如果将相同类型的字段组合在一起进行内存分配，并且假设按照 byte -> long -> int 的顺序分配内存，则最终所分配的内存空间布局如图 6.8 所示。

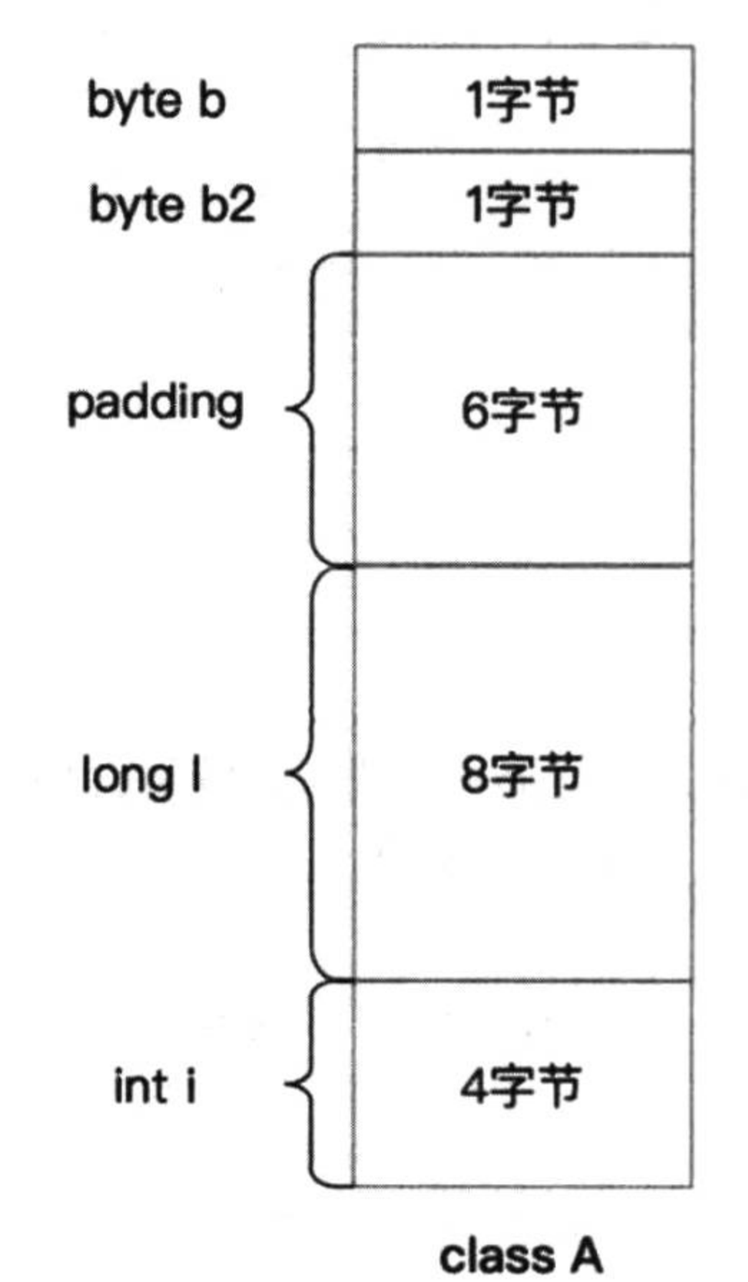

图 6.8　对类 A 进行字段重排后的内存布局

经过字段重排后，现在只需要补白 6 字节，类 A 总共只占 20 字节内存空间，比重排之前节省宝贵的 4 字节。

但是，与 C 语言一样，gcc 编译器不仅要求对结构体里的每个成员进行对齐补白，还要求对整个结构体进行对齐，JVM 也不仅需要负责将 Java 类里面的每个字段进行对齐，还需要对整个类进行对齐。JVM 规范要求对类的堆内存空间按 8 字节对齐，因此刚才对类 A 进行字段重排后，虽然全部字段加起来只需要 20 个内存单元，但是为了对整个类进行对齐，最终仍需补白至 24 字节，这样一来，反倒与没有重排之前一样了。

因此，这时候 JVM 字段重排策略的第 2 点就显得十分重要了，那就是重排后字段的顺序。前面讲过，在定义 C 语言的结构体时，应当按照字段类型的宽度从小到大的顺序声明成员项。而 JVM 里面则是按照字段类型的宽度从大到小的顺序分配字段。

现在按照 long -> int -> byte 的顺序对类 A 的字段空间重新排列，重排后的内存空间布局如图 6.9 所示。

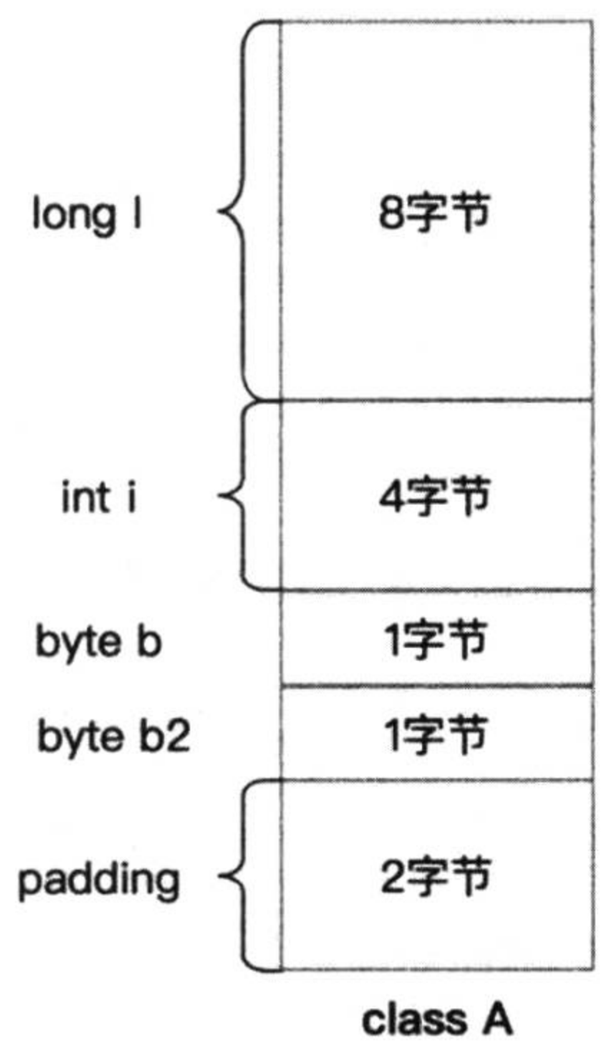

图 6.9　class A 按指定顺序重排后的内存布局

现在按照新的顺序对数据类型进行排序，不仅对各个字段进行了内存对齐，同时对整个类按 8 字节进行对齐（暂时没考虑 oop 对象头），重排后一共仅占 16 个内存单元，比优化之前的 24 字节节省了宝贵的 8 字节内存空间。别小看这 8 字节的内存空间，对于一个生产环境中的 JVM，其需要加载成千上万个类实例对象，这成千上万个 8 字节加起来的内存空间便十分可观。

通过这个例子也可以知道，JVM 要按照 long -> int -> short -> byte 的顺序对字段进行排列，实在是有其道理的。

当然，JVM 对字段重排的优化还不止如此，下文会在分析源码时再次讲到优化策略。

3．计算变量偏移量

本书虽然研究理论，但更关注 JVM 内部的实现细节，这是本书的宗旨。上面关于 JVM 内存分配的原理讲了那么多，但终究要窥一窥 HotSpot 内部究竟是如何实现的。

其实通过前面所阐述的 Hotspot 对字段内存分配的策略，不难推导出一种计算 Java 类各字段偏移量的方法。既然 Hotspot 将字段进行了重排，将相同类型的字段存储在一起，那么便可以先计算出其内部 5 大类型字段的起始偏移量。每一种类型都包含零或多个 Java 类字段，基于该类型的起始偏移量，便可逐个计算出该类型所对应的每一个具体的 Java 类字段的偏移量。事实上，Hotspot 也就是这么实现的。看源码：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：ClassFileParser::parseClassFile()

```
int next_nonstatic_oop_offset;
```

```
    int next_nonstatic_double_offset;
    int next_nonstatic_word_offset;
    int next_nonstatic_short_offset;
    int next_nonstatic_byte_offset;
    int next_nonstatic_type_offset;
    int first_nonstatic_oop_offset;
    int first_nonstatic_field_offset;
    int next_nonstatic_field_offset;

    //first_nonstatic_field_offset 是 Java 类字段的起始偏移量，这里将 next_nonstatic_
field_offset 也指向起始偏移量
    next_nonstatic_field_offset = first_nonstatic_field_offset;

    //前置流程解析 Java 类常量池时，解析出 Java 类中各种类型的字段数量，这里分别获取这 5 种类
型的字段的数量
    unsigned int nonstatic_double_count = fac.nonstatic_double_count;
    unsigned int nonstatic_word_count   = fac.nonstatic_word_count;
    unsigned int nonstatic_short_count  = fac.nonstatic_short_count;
    unsigned int nonstatic_byte_count   = fac.nonstatic_byte_count;
    unsigned int nonstatic_oop_count    = fac.nonstatic_oop_count;

    //根据不同的顺序策略，计算 oop 或者 long 的起始偏移量
    if( allocation_style == 0 ) {
      // Fields order: oops, longs/doubles, ints, shorts/chars, bytes
      next_nonstatic_oop_offset    = next_nonstatic_field_offset;
      next_nonstatic_double_offset = next_nonstatic_oop_offset +
                                     (nonstatic_oop_count * heapOopSize);
    } else if( allocation_style == 1 ) {
      //如果 allocation_style 是 1,则字段分配顺序是 longs/doubles、ints、shorts/chars、
bytes、 oops
      //由于先分配 long 类型字段所以 long 类型的字段的起始偏移量自然就是整个 Java 类字段的起始
偏移量
      next_nonstatic_double_offset = next_nonstatic_field_offset;
    }
```

HotSpot 提供了好几种重排顺序选项。如果 allocation_style 的值是 0，则按照 oops -> longs /doubles -> ints -> shorts/chars -> bytes 的顺序为字段分配内存空间；如果 allocation_style 的值是 1，则最先分配 longs/doubles，最后分配 oops。这里仅讨论后一种情况。

根据上面的源码，此时其实 longs/doubles 类型的起始偏移量已经计算出来了，这个偏移量就是整个 Java 类的起始偏移量。

分配完 longs/doubles 类型之后接着分配 ints 类型，说白了就是计算 ints 的起始偏移量。ints 的起始偏移量一定位于 longs 内存空间的末尾，所以 ints 的起始偏移量的计算方法是：

longs/doubles 的起始偏移量 + longs 的宽度 * longs 的数量。

ints 之后的各种数据类型的起始偏移量的计算方法也类似，我们看 HotSpot 的实现：

```
next_nonstatic_word_offset  = next_nonstatic_double_offset +
                              (nonstatic_double_count * BytesPerLong);
next_nonstatic_short_offset = next_nonstatic_word_offset +
                               (nonstatic_word_count * BytesPerInt);
next_nonstatic_byte_offset  = next_nonstatic_short_offset +
                               (nonstatic_short_count * BytesPerShort);
```

通过这段代码，HotSpot 将 ints、shorts/chars、bytes 这 3 种字段类型的起始偏移量也计算出来。

这里要注意两点：

- 无论字段重排是哪种顺序，longs/doubles 后面跟的一定是 ints，而 ints 后面所跟的一定是 shorts/chars，并且 shorts/chars 后面跟的一定是 bytes。
- 由于 longs/doubles 字段进行了对齐处理，所以其末尾的下一个内存地址一定是 8 字节的整数倍。对于这样的内存地址，其也一定是 4 字节的整数倍，所以将 ints 紧跟在 longs/doubles 字段后面，ints 的字段也就天然是对齐的。同理，ints 后面的 shorts/chars 以及 shorts/chars 后面的 bytes 字段也一定是天然对齐的。

此时的内存布局如图 6.10 所示。

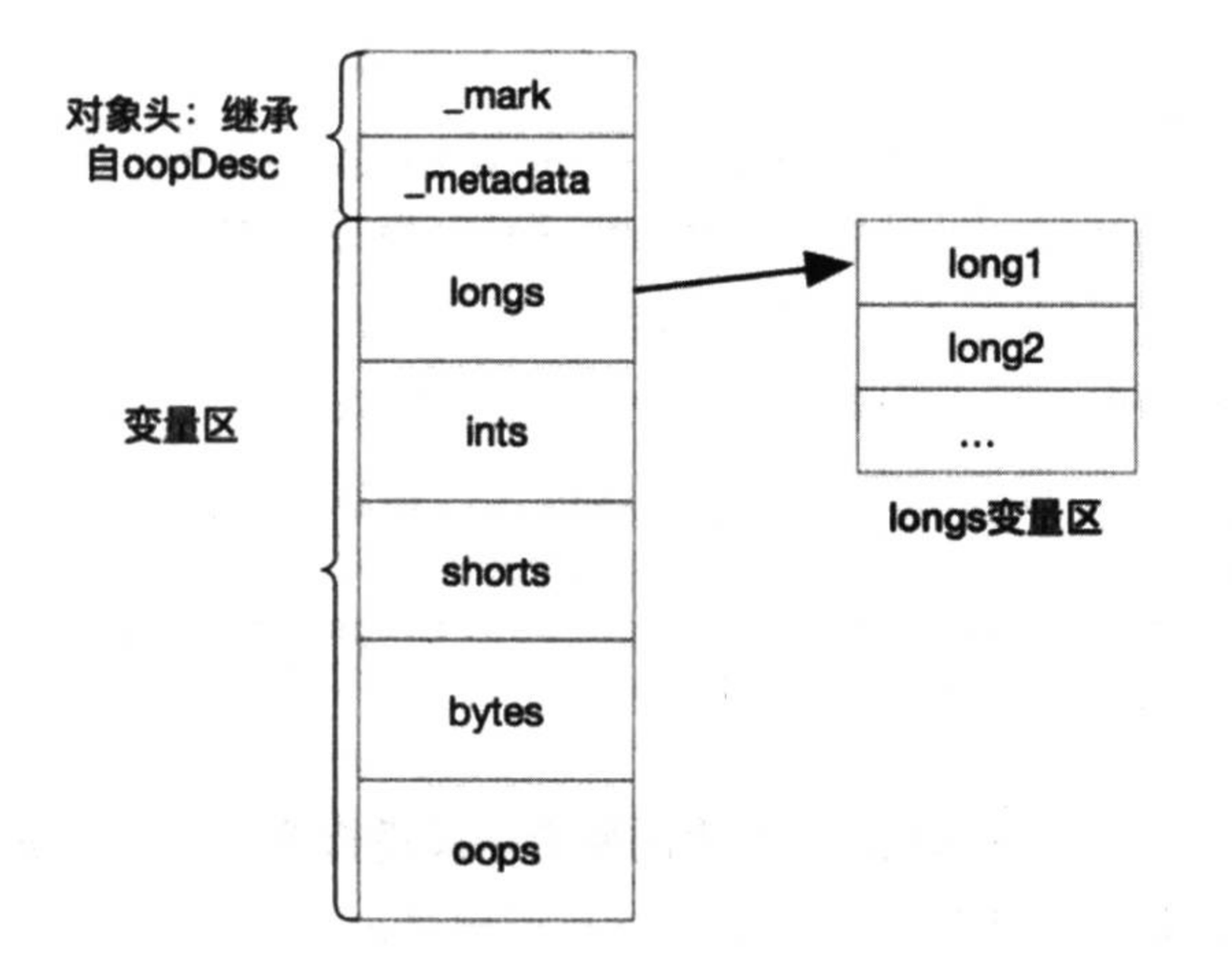

图 6.10　内部 5 大类型的起始偏移量

完成了 JVM 内部 5 大类型数据的起始偏移量计算之后，接着就可以计算每种类型所对应的 Java 类中的字段的具体偏移量了。计算方法很简单，将 Java 类中的字段按照其所属的 5 大类型的起始偏移量进行顺序排列即可。看 HotSpot 的源码实现：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：ClassFileParser::parseClassFile()

```
//获取 Java 类中非静态字段的数量
int len = fields->length();
for (int i = 0; i < len; i += instanceKlass::next_offset) {
  int real_offset;
  FieldAllocationType atype = (FieldAllocationType) fields->ushort_at(i +
instanceKlass::low_offset);
      switch (atype) {
        //========静态字段偏移量计算，这里略过
        case STATIC_OOP:
        //…

        //========非静态字段偏移量计算
        case NONSTATIC_OOP:
          if( nonstatic_oop_space_count > 0 ) {
            real_offset = nonstatic_oop_space_offset;
            nonstatic_oop_space_offset += heapOopSize;
            nonstatic_oop_space_count  -= 1;
          } else {
            real_offset = next_nonstatic_oop_offset;
            next_nonstatic_oop_offset += heapOopSize;
          }
          // …

        case NONSTATIC_DOUBLE:
          real_offset = next_nonstatic_double_offset;
          next_nonstatic_double_offset += BytesPerLong;
          break;
        default:
          ShouldNotReachHere();
      }
      fields->short_at_put(i + instanceKlass::low_offset,
extract_low_short_from_int(real_offset));
      fields->short_at_put(i + instanceKlass::high_offset,
extract_high_short_from_int(real_offset));
  }
```

在这段逻辑里面，HotSpot 对 Java 类中的所有字段进行遍历，并分别计算其各个字段的偏移量。以 Java 类中的 long 类型字段为例，其计算逻辑是：

```
case NONSTATIC_DOUBLE:
  real_offset = next_nonstatic_double_offset;
  next_nonstatic_double_offset += BytesPerLong;
  break;
```

real_offset 就是当前字段的真实偏移量。假设 Java 类中包含 2 个 long 类型的字段，并假设 Hotspot 当前遍历到第一个 long 字段，则该 long 字段的偏移量就是 next_nonstatic_double_offset。

计算完第一个 long 字段的偏移量之后，Hotspot 执行 next_nonstatic_double_offset += BytesPerLong，将 next_nonstatic_double_offset 地址往后偏移 8 字节，这样当 Hotspot 遍历到 Java 类中第 2 个 long 类型的字段时，通过 real_offset = next_nonstatic_double_offset 就能直接计算出第二个 long 字段的偏移量了。

4．gap 填充

也许有细心的小伙伴可能发现上面那段代码中有部分逻辑比较不同寻常，那就是 case NONSTATIC_DOUBLE 分支里的逻辑与其他 case 分支的逻辑都不一样，其他 case 分支的逻辑明显比 case NONSTATIC_DOUBLE 这个条件分支的逻辑复杂一些。这是为什么呢?

要解决这个问题，不得不再去关注 JVM 内部的 oop 对象头的事儿。前文也讲到过，每一个 Java 类在堆内存中，都是从 oop 头开始的，而这个 oop 头所占的内存空间与 JVM 是否开启了指针压缩策略有关。在 64 位平台上，如果开启了指针压缩策略，则对象头仅会占用 12 个内存单元，如果没有开启，则会占用 16 个内存单元。

如果 oop 对象头只占用了 12 个内存单元，就会带来一个问题：对象头作为一个整体，不是 8 字节对齐的。对象头作为一个整体是否按 8 字节对齐，与对象头本身没有关系，但是却影响其后面的 Java 类字段的自然对齐效果。由于 JVM 按照 longs/doubles -> ints -> shorts/chars -> bytes 的顺序分配内存空间，因此紧跟在 oop 对象头之后的就是 longs/doubles 类型的数据。

如果 oop 对象头只占用了 12 字节，那么其后面第一个 long 类型的字段的起始偏移量按照常理应该是 12，但这不符合 long 类型数据的自然对齐原则（12 不能被 8 整除），所以只能在 oop 对象头后面连续填充 4 个空字节，然后从第 16 个偏移位置处为第一个 long 类型的字段分配内存。这样一来就造成从 oop 对象头到第一个 long 类型字段之间浪费了 4 字节的内存空间。虽然 4 字节看起来微不足道，但是这对于一个高性能的虚拟机而言是绝对不能接受的。

对内存吝啬到前无古人的地步的 JVM 来说，即使这么一点内存也必须要充分利用起来，而利用的方式便是，将 int、short、byte 等宽度小于等于 4 字节的字段往这个内存间隙里插入，虽然这会破坏 Hotspot 对不同类型字段重排的顺序策略。

我们来看 HotSpot 的源码：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：ClassFileParser::parseClassFile()

```
//填充 gap 间隙。如果对象头+父类非静态字段的末尾不是 8 字节(long)对齐，则在这中间填充 Java
类中非 long/double 类型的字段
```

```
//long/double 字段的起始位置还是 8 字节对齐
//gap 填充的顺序：int, short, byte, oopmap
if( nonstatic_double_count > 0 ) {
  int offset = next_nonstatic_double_offset;
  next_nonstatic_double_offset = align_size_up(offset, BytesPerLong);
  if( compact_fields && offset != next_nonstatic_double_offset ) {
    // Allocate available fields into the gap before double field.
    int length = next_nonstatic_double_offset - offset;
    assert(length == BytesPerInt, "");

    //先将 int 型字段填充进 gap，理论上只能填充 1 个 int，因为 gap 的总大小最大只能是 7
    nonstatic_word_space_offset = offset;
    if( nonstatic_word_count > 0 ) {
      nonstatic_word_count      -= 1;
      nonstatic_word_space_count = 1; // Only one will fit
      length -= BytesPerInt;
      offset += BytesPerInt;
    }

    //如果填充了 1 个 int 型字段，gap 还没填充完，则接着填充 short。由于 Java 类中可能并没
有定义 int 类型的字段，因此可以填充多个 short
    nonstatic_short_space_offset = offset;
    while( length >= BytesPerShort && nonstatic_short_count > 0 ) {
      nonstatic_short_count      -= 1;
      nonstatic_short_space_count += 1;
      length -= BytesPerShort;
      offset += BytesPerShort;
    }

    //如果填充完 short 字段之后，还有 gap 空间，则继续填充 byte 类型的字段
    nonstatic_byte_space_offset = offset;
    while( length > 0 && nonstatic_byte_count > 0 ) {
      nonstatic_byte_count      -= 1;
      nonstatic_byte_space_count += 1;
      length -= 1;
    }

    //如果 byte 类型的字段填充完了还有 gap 空间，则继续填充 oopmap
    // Allocate oop field in the gap if there are no other fields for that.
    nonstatic_oop_space_offset = offset;
    if( length >= heapOopSize && nonstatic_oop_count > 0 &&
       allocation_style != 0 ) { // when oop fields not first
      nonstatic_oop_count      -= 1;
      nonstatic_oop_space_count = 1; // Only one will fit
      length -= heapOopSize;
      offset += heapOopSize;
    }
```

```
    }
}
```

事实上，如果一个 Java 类显式继承了父类，那么如果父类字段的末尾不是按 8 字节对齐的，则父类字段末尾与子类第一个 long/double 字段之间也会形成补白空隙，则这段空隙也会参与上述逻辑计算，被安插 int、short、byte 等宽度比较小的字段。下文会继续讲解。

5. Java 语言与其他语言处理内存对齐的差异

纵观 HotSpot 对 Java 类字段的堆内存分配算法，可以看出 HotSpot 对内存的利用几乎已经到了极致，虽然存在模仿，但几乎已经无法再被超越了。

做高手的感觉真的好寂寞啊！

我仿佛听到了 Hotspot 内心深处的一声叹息。

别的就不说了，就拿经典的 gcc 编译器来说，最多也只做到了自动将数据类型进行自然对齐，基本不需要开发者再去为这件事情而烦恼，但是也仅此而已。例如前面所列举过的结构体的例子：

清单：示例程序
作用：C 语言的 struct 内存对齐

```
struct A
    {
     int a;
        char b;
        short c;
};
```

struct A 结构体由于内存对齐的需要，最终需要分配 8 字节的内存空间。如果将 A 结构体内部的成员项的顺序变动一下，变成如下：

清单：示例程序
作用：C 语言的 struct 内存对齐

```
struct B
    {
        char b;
        int a;
        short c;
};
```

则内存占用空间立马变大。

这种情况在 Java 中是不存在的。与 struct A 所对等的 Java 类如下：

清单：示例程序
作用：Java 语言的内存对齐

```
class A{
    int a;
      byte b;
      short c;
};
```

除去 Java 类在 JVM 内部所对应的 oop 的对象头部分的内存空间，最终类 A 字段在堆内存中也会占用 8 字节。

对于 Java 类，不管其内部字段的声明顺序如何变化，都不会影响其内存占用，这主要得益于 JVM 的字段重排算法。

其实经典的编译器诸如 gcc 等，也是可以像 Hotspot 那样，对局部变量进行字段重排的，这在算法层面完全是可行的，只是囿于当时的算法技术而未如此优化。不过事物发展的规律总是长江后浪推前浪，短暂的几十年的历史罅隙，对算法而言可谓是悠久长远的历史长河，再过几十年，说不定会有新的算法横空出世，睥睨天下。不过不管未来怎样，至少从目前来看，JVM 的这种内存分配算法几乎是已经到了无可再优化的境界。

6.2.3 Java 字段内存分配总结

前面浓墨重彩地详细剖析了 Hotspot 对 Java 类字段的内存分配原理，并举了若干例子。这部分内容实在是重要之极，也是理解 JVM 内存模型的最基础、最核心的一步。便连作者本人以前也对 JVM 的内存模型存在诸多误会，以为全是面向对象，以为面向对象的东西必然会浪费非常多的内存空间。直到将 Hotspot 的类字段分配模型剖析完，才发现压根儿不是那么回事，JVM 对内存空间利用率的要求是非常苛刻的，如果去掉占用 12 字节或 16 字节的对象头，其内存使用效率一定超过绝大多数的编程语言了。

本节再对 JVM 的类字段分配策略进行梳理归纳。虽然上面所分析的源码皆基于 Hotspot，但是 HotSpot 所努力实现的目标，事实上正是 JVM 的规范要求。

- 规则 1：任何对象都是以 8 字节为粒度进行对齐的。
- 规则 2：类属性按照如下优先级进行排列：长整型和双精度类型；整型和浮点型；字符和短整型；字节类型和布尔类型；最后是引用类型。这些属性都按照各自类型宽度对齐。
- 规则 3：不同类继承关系中的成员不能混合排列。首先按照规则 2 处理父类中的成员，接着才是子类的成员。

◎ 规则 4：当父类最后一个属性和子类第一个属性之间间隔不足 4 字节时，必须扩展到 4 字节的基本单位。

◎ 规则 5：如果子类第一个成员是一个双精度或长整型，并且父类没有用完 8 字节（没有显式的父类，并且 JVM 开启了指针压缩策略，oop 对象头只占用 12 字节时），JVM 会破坏规则 2，按整型（int）、短整型（short）、字节型（byte）、引用类型（reference）的顺序向未填满的空间填充。

对于规则 1，没啥好说的，与 C 语言中对结构体整体对齐的约束一样，JVM 也需要使 Java 类在 JVM 内部的堆内存映像从整体上做到对齐，这并不是为了方便 Java 类自己，而是为了方便其后续的其他类的内存分配。

对于规则 2，是内存分配时最基本的要求，自然对齐，否则 JVM 运行在某些不支持非对齐内存访问的 CPU 硬件上时会因此而崩溃。

对于规则 3，其实于内存的利用率而言，并不是一个必须要遵守的原则，甚至反而因为遵守了这个原则而导致内存利用率降低。这个原则存在的目的主要是为了内存分析时方便，尤其是当一个类的继承体系比较深的时候，如果若干父类与子类的字段都混合组合在一起，那内存分析人员的情绪一定是崩溃的。

对于规则 4，也是为了让父类属性集合从整体上做到对齐，从而方便其后续子类字段在处理对齐时能够尽可能地简单。Hotspot 从源码级别保证了规则 4 的履行：

```
first_nonstatic_field_offset = instanceOopDesc::base_offset_in_bytes() +
                               nonstatic_field_size * heapOopSize;
```

这行代码上文讲过，意在计算 Java 类字段的整体起始偏移量。这个偏移量也要算上 oop 对象头和父类字段，而父类字段是按照 heapOopSize 对齐的，heapOopSize 的定义如下：

清单：share/vm/utilities/globalDefinitions.cpp
作用：heapOopSize 定义

```
int heapOopSize        = 0;
if (UseCompressedOops) {
 // Size info for oops within java objects is fixed
 heapOopSize        = jintSize;
 LogBytesPerHeapOop = LogBytesPerInt;
 LogBitsPerHeapOop  = LogBitsPerInt;
 BytesPerHeapOop    = BytesPerInt;
 BitsPerHeapOop     = BitsPerInt;
} else {
 heapOopSize        = oopSize;
 LogBytesPerHeapOop = LogBytesPerWord;
 LogBitsPerHeapOop  = LogBitsPerWord;
```

```
        BytesPerHeapOop     = BytesPerWord;
        BitsPerHeapOop      = BitsPerWord;
    }
```

如果开启了指针压缩策略，则其大小是 jintSize，而 jintSize 的值是 4；如果没有开启指针压缩策略，则其大小是 oopSize， oopSize 的值是 8。

父类的字段大小是 nonstatic_field_size，该值的算法如下：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：ClassFileParser::parseClassFile()

```
    int notaligned_offset;
    if( allocation_style == 0 ) {
      notaligned_offset = next_nonstatic_byte_offset + nonstatic_byte_count;
    } else { // allocation_style == 1
      next_nonstatic_oop_offset = next_nonstatic_byte_offset +
nonstatic_byte_count;
      if( nonstatic_oop_count > 0 ) {
        next_nonstatic_oop_offset = align_size_up(next_nonstatic_oop_offset,
heapOopSize);
      }
      notaligned_offset = next_nonstatic_oop_offset + (nonstatic_oop_count *
heapOopSize);
    }
    next_nonstatic_type_offset = align_size_up(notaligned_offset,
heapOopSize );

    //这里计算的是 field 所占用的字节数，而非 field 的数量。如果是数量，会命名为 count
    //本字段大小由父类的该字段大小加上本类的字段大小
    nonstatic_field_size = nonstatic_field_size +
((next_nonstatic_type_offset
                                - first_nonstatic_field_offset)/heapOopSize);
```

在这段逻辑中，先对 Java 类中的 byte 类型字段进行补白，对齐至 4 字节或 8 字节。由于在 JVM 分配内存时，排在最末尾的是 byte 类型，而前面的 long、int、short 这几种类型的字段之间一定不会存在任何补白（因为 long 末尾后面的内存位置一定能够保证 int 类型字段是自然对齐的，而 int 末尾后面的内存位置也一定能够保证 short 类型字段是自然对齐的），所以只要确保最末尾的 byte 类型的字段能够按照 4 字节或者 8 字节对齐，则整个 Java 所对应的堆内存也一定是按照 4 字节或者 8 字节对齐的。

所以上面这段逻辑能够确保 next_nonstatic_type_offset 最终一定也是按照 4 字节或 8 字节对齐。这直接影响到最终所计算出来的父类的 nonstatic_field_size 值。

在这段代码的最后一行表达式中，为了让问题简化，假设父类没有父类，所以最后一行表

达式中的 nonstatic_field_size 一开始是 0，此时最后一行表达式退化成下面这行表达式：

```
nonstatic_field_size = ((next_nonstatic_type_offset - first_nonstatic_field_
offset)/heapOopSize);
```

若 JVM 没有开启指针压缩选项，则 heapOopSize 的值为 8，而 next_nonstatic_type_offset 经过补白，已经是按照 8 字节对齐的。没有开启指针压缩选项，则 oop 对象头占用 16 字节，first_nonstatic_field_offset 的值就是 16，所以这行表达式，所有的 3 个变量值都是 8 的整数倍，所以最终计算出来的也必定是 8 的整数倍。

若 JVM 开启了指针压缩选项，则 heapOopSize 的值为 4，next_nonstatic_type_offset 经过补白，也按照 4 字节对齐。开启指针压缩选项后，oop 对象占用 12 字节，则 first_nonstatic_field_offset 的值为 12，所以这行表达式中的 3 个变量都是 4 的整数倍，则最终所计算出来的结果也必定是 4 的整数倍。

所以，无论 JVM 是否开启指针压缩选项，则父类字段所需要占用的内存空间必定是 4 的整数倍。这句话换个说法，就是规则 4。

下面这个例子对于本规则有极强的说服力：

清单：Father.java
功能：测试指针压缩

```
public class Father {
    private byte b1;
}

class Son extends Father{
    byte b2;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Son();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

对于规则 5，其实上文已经详细分析过 Hotspot 的源码实现。当开启了指针压缩选项，oop 对象头只需要 12 字节的内存空间，如果 Java 类中定义了 long 或 double 类型的字段，则 oop 对象头与第一个 long/double 字段之间会有 4 字节的补白空间。JVM 为了充分使用内存，坚决不浪费宝贵的内存空间，就按照整型（int）、短整型（short）、字节型（byte）、引用类型（reference）的顺序往这段空隙中填充。

下面这个 Java 类可以验证这一规则：

清单：Father.Java
功能：测试 gap 机制

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

import java.util.*;

public class Father {
    private long l;
    private byte b1;
    private byte b2;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

开启指针压缩选项，假设 HotSpot 没有填充 oop 对象头后面的空隙 gap，则此时 Father 类的内存布局如图 6.11 所示。

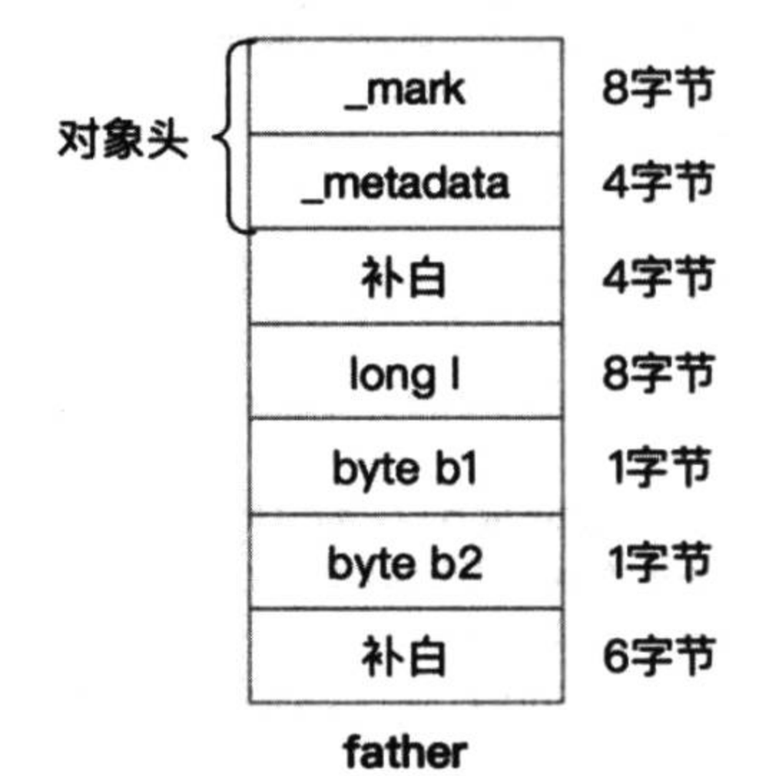

图 6.11　没有 gap 机制时的 Father 类实例内存布局

最终 Father 类在内存中占用 32 字节。而 HotSpot 填充 gap 后，内存布局如图 6.12 所示。

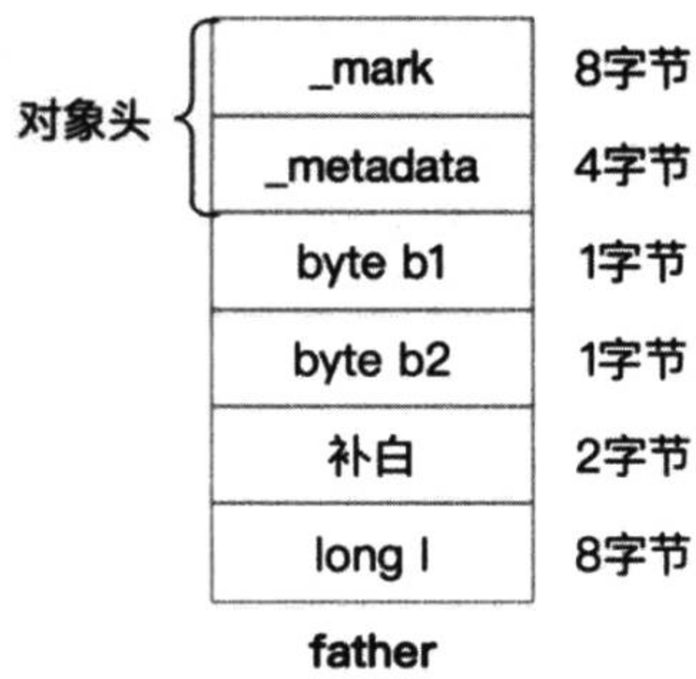

图 6.12　使用 gap 机制后的 Father 类实例内存布局

此时 Father 类在内存中仅占用 24 字节。

而当有父类参与时，则需要同时满足规则 4 与规则 5。下面这个例子正好能够验证这种情况：

清单：Father.java
功能：测试 Java 字段内存对齐

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

import java.util.*;

public class Father {
    int i;
    short s;
}

class Son extends Father{
    byte b;
    long l;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Son();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

父类为了满足规则 4，做到按 4 字节对齐，所以父类的末尾补白了 2 个字节，这段空间硬生生被浪费掉了。如果在父类中再定义一个占 2 字节的变量，则 JVM 会用这个字段填充被补白的 2 字节空间。如下面的程序：

清单：Father.java
功能：测试 Java 字段内存对齐

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

import java.util.*;

public class Father {
    int i;
    short s;
    short s2;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}

class Son extends Father{
    byte b;
    long l;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Son();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

而即使这样，父类字段的末尾与子类的第一个 long 类型字段之间仍然会有 4 字节的补白。由于子类中包含了 1 个 byte 类型的字段，所以 JVM 违反了规则 2，将这个字段插在了这个间隙里，现在还剩下 3 个补白字节。所以如果子类中继续声明 3 个 byte 类型的字段，则 JVM 会将这 3 个字段插入到剩下的 3 个补白字节中，从而使得 Son 类的堆内存大小依然保持不变。

清单：Father.java
功能：测试 Java 字段内存对齐

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

import java.util.*;

public class Father {
    int i;
    short s;
```

```
    short s2;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}

class Son extends Father{
    byte b;
    long l;
    byte b2;
    byte b3;
    byte b4;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Son();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

6.3　从源码看字段继承

在 Java 类的继承关系与细节上，很多小伙伴可能对有些概念存在疑问或误解，下面就从源码的角度逐一分析。

6.3.1　字段重排与补白

分析问题总是免不了要做实验，阅读源码使人明白细节，而实验则使人能够从结果直接验证细节。为了分析 Java 类的一些继承问题，需要通过做实验进行验证，但是在做这个实验之前需要做个实验来先验证字段重排与补白。之所以要验证这个课题，是因为在继承的实验中，基本通过观察父类与子类所占用的内存大小来验证字段是否被继承，而父类与子类所占用的内存大小并不严格等价于其所声明的各个成员本身所占用内存的总和，这还受到内存对齐补白的机制制约。

先从一个最简单的实验用例开始，这是一个空的 Java 类：

清单：/Father.java
功能：一个空的 Java 类

```
public class Father {

}
```

这个空的 Java 类到底占用多大内存空间呢？由于 Java 编程语言并没有提供类似于 C/C++ 语言的 sizeof 这种可以获取变量或结构体或类型大小的关键字，因此只能借用第三方类库。这里借用 Ehcache 所提供的 SizeOf 工具类。Ehcache 作为广泛使用的分布式缓存中间件，对空间管理必然十分精细，对每一个 Java 类所占用的内存空间大小计算必须十分准确，这是实现内存精细化管理的关键前提。Ehcache 里提供了实现 SizeOf 的多种工具版本，这里选择 AgentSizeOf。

准备就绪，实验开始，在 Father 类中编写 main()函数计算该类所占用的内存大小：

清单：/Father.java
功能：一个空的 Java 类

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }

}
```

运行 main()函数后打印如下结果（在 64 位平台上，后面的例子都基于 64 位）：

```
father's size: 16
```

这个结果可能出乎很多小伙伴的预料，但是这个结果是十分“伟光正”的。

前面讲到，Hotspot 内部会使用 oop 来表示每一个 Java 类的实例，Java 类实例在堆内存中的布局如图 6.13 所示。

Father 类没有显式的父类，并且本身是个空类，没有任何私有域，因此该类在堆内存中只有对象头。不过前文讲过，在 64 位平台上，如果开启压缩选项，则对象头占 12 字节的内存空间，否则占用 16 字节。

图 6.13　instanceOop 的内存布局

为了确认是否开启压缩选项，使用如下命令查看 JVM 启动的参数：

```
java -XX:+PrintCommandLineFlags
-XX:MaxNewSize=174485504
-XX:MaxTenuringThreshold=4
-XX:NewRatio=7
-XX:NewSize=21811200
-XX:OldPLABSize=16
-XX:OldSize=65433600
-XX:+PrintCommandLineFlags
-XX:+UseCompressedOops
-XX:+UseConcMarkSweepGC
-XX:+UseParNewGC
```

可以看到，JVM 默认开启了压缩参数-XX:+UseCompressedOops。既然使用了压缩算法，则对象头只应该占用 12 字节的内存空间呀，为何 Father 类却占用了 16 字节呢？

这主要是由对齐引起的。JVM 在 64 位平台上为 Java 类对象实例分配内存时，会基于 8 字节的整数倍数进行对齐。如果内存空间不足 8 字节的整数倍，则会将其补白到 8 字节的整数倍。这就是 Father 这个空类占用 16 字节内存的原因。其内存布局如图 6.14 所示。

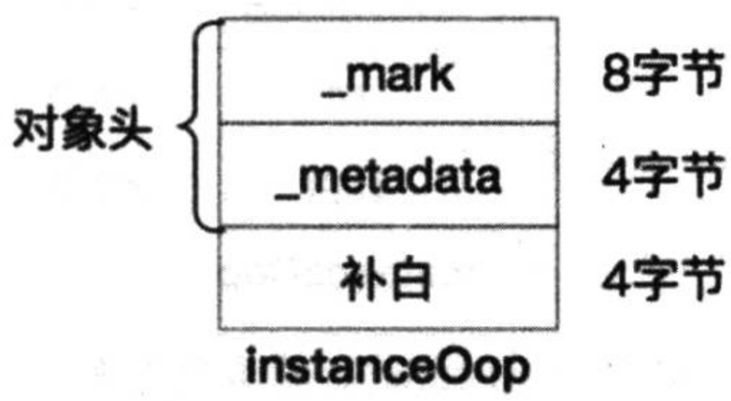

图 6.14　Father 类的堆内存布局

为了验证这一点，对 Father 类进行改造。先增加一个字节类型的变量：

清单：/Father.java
功能：包含 1 个 byte 类型字段的 Java 类

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {
    private byte b1;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }

}
```

现在增加了一个字节类型的变量 b1。在 JVM 规范中，一个字节类型的变量仅占 1 字节空间大小。

由于 Father 类没有显式继承父类，因此在堆中，其对象头之后应该紧跟着 b1 变量。当 JVM 最终为对象头和 b1 变量分配内存空间时，由于对象头加上 b1 所占用的内存空间总和只有 13 字节，因此 JVM 会将其补白到 16 字节，所以 Father 类最终应该仍然占用 16 字节的内存空间。

运行 main()函数，得到结果的确是 16：

```
father's size: 16
```

Father 类的堆内存布局如图 6.15 所示。

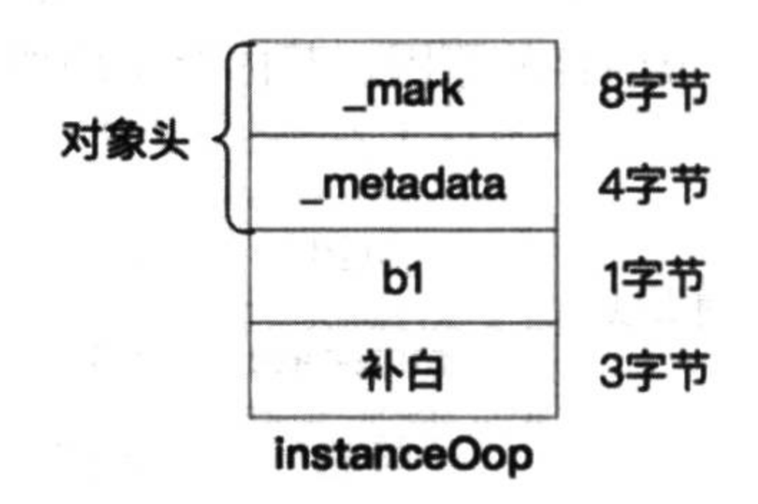

图 6.15　Father 类的堆内存布局

为了继续验证，在 Father 类中继续定义 3 个 byte 类型的变量，如下：

清单：/Father.java
功能：包含 4 个 byte 类型字段的 Java 类

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;
```

```
public class Father {
    private byte b1;
    private byte b2;
    private byte b3;
    private byte b4;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }

}
```

运行 main()函数，得到如下结果：

```
father's size: 16
```

由于现在对象头加上 4 个 byte 类型的变量的内存总和正好等于 16 字节，正好是 8 的整数倍，因此 JVM 不会对其进行补白。

此时 Father 类的堆内存布局如图 6.16 所示。

图 6.16　Father 类的堆内存布局

接着见证奇迹的时候到了，再增加一个 byte 类型的字节，如下：

清单：/Father.java
功能：包含 5 个 byte 类型字段的 Java 类

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {
    private byte b1;
    private byte b2;
```

```
        private byte b3;
        private byte b4;
        private byte b5;

        public static void main(String[] args){
            SizeOf sizeOf = new AgentSizeOf();
            Father father = new Father();
            System.out.println("father's size: " + sizeOf.sizeOf(father));
        }

    }
```

现在执行 main()函数，得到结果如下：

```
father's size: 24
```

此时 Father 类的堆内存布局如图 6.17 所示。

图 6.17　Father 类的堆内存布局

这几个例子已经能够验证 Java 类在内存分配时的对齐机制了。不过 byte 类型的变量不够通用，因此再对 Father 类进行微小的变动，相信小伙伴们能够准确计算出下面这个 Java 类所占的内存空间大小：

清单：/Father.java

功能：包含 2 个 int 类型的 Java 类

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {
    private int i1;
    private int i2;
```

```
    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }

}
```

现在在 Father 类里面定义了 2 个 int 类型的基本类型变量，JVM 标准规定，一个 int 类型的变量占用 4 字节内存大小。因此 Father 类需要的内存空间大小为：

12 字节(对象头大小) + 4 字节 * 2(2 个 int 类型变量) = 20 字节

由于 JVM 的内存补白机制，因此最终为其分配 24 字节的内存空间，后面的 4 字节通过补白进行对齐。其内存布局如图 6.18 所示。

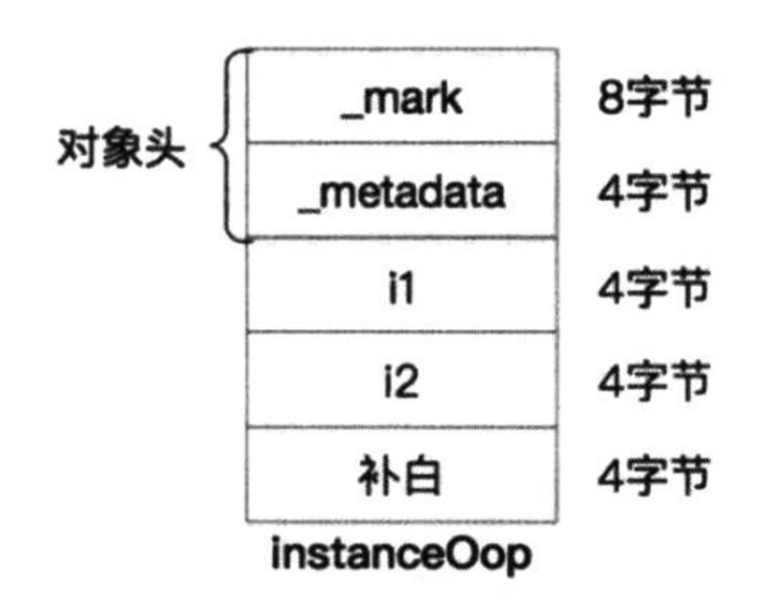

图 6.18　Father 类的堆内存布局

运行 main()函数，的确打印出 24。

关于 JVM 内存对齐的机制就讲到这里，这块理清楚了，有助于接下来要讲的继承机制。

6.3.2　private 字段可被继承吗

有一种说法是，凡是父类中被定义成 private 的字段，都是“老子”的私有财产，即便是“儿子”，也继承不了。

可是 JVM 的世界真的如此无情吗？

看下面这个例子：

清单：/Father.java
功能：演示被 private 关键字修饰的字段的继承机制

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {

}

class Son extends Father{
    public static void main(String[] args){
        Father father = new Father();
        Son son = new Son();

        SizeOf sizeOf = new AgentSizeOf();

        System.out.println("father's size: " + sizeOf.sizeOf(father));
        System.out.println("son's size: " + sizeOf.sizeOf(son));
    }
}
```

运行 main()函数，打印如下结果：

```
father's size: 16
son's size: 16
```

本例中的父类与子类都是空类，因此这两个对象实例在内存中其实都是只有对象头，只需要 12 字节的内存空间，但是为了对齐，最终 JVM 为其分配了 16 字节的内存大小。此时父类与子类的内存布局如图 6.19 所示。

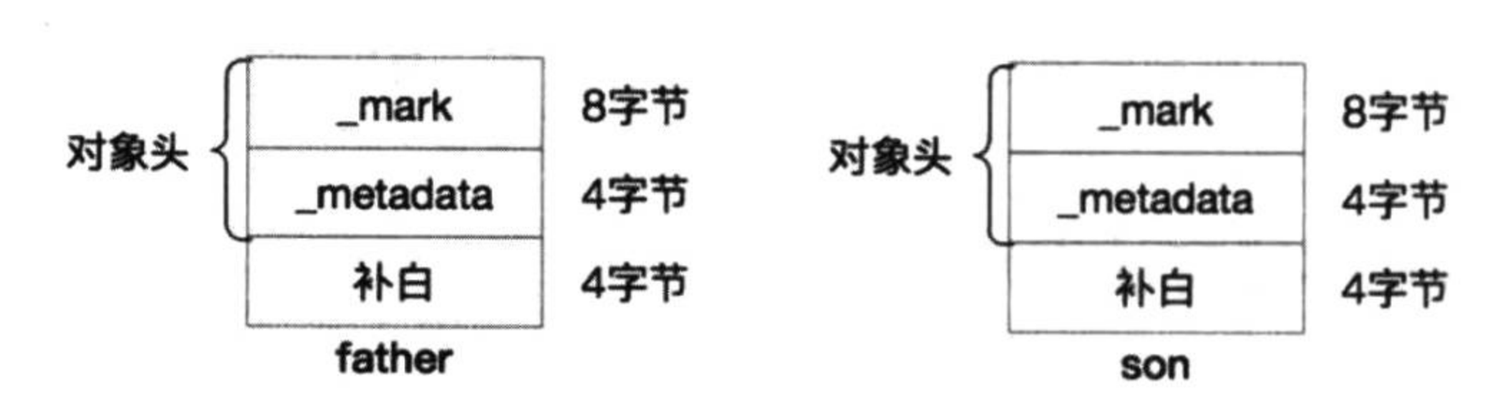

图 6.19　Father 与 Son 实例的堆内存布局

虽然通过上面的例子可以证明，Java 父类中的 private 字段的确被子类继承了，证明在 JVM 的世界里其实也是有爱的。可是很多小伙伴对一件事情很是耿耿于怀，因为从编程的角度看，子类并不能直接访问父类中的 private 字段。例如下面的例子：

清单：/Father.java
功能：演示被 private 关键字修饰的字段的继承机制

```
public class Father {
    private int b1;
    private int b2;
}

class Son extends Father{

    public Son(){
        this.b1;
        super.b1;
    }

}
```

在本例中，父类中定义了变量 b1，并使用 private 关键字进行修饰。在子类 Son 的构造函数中，无论使用 this.b1，还是 super.b1，均会报编译错误。

前面不是证明了“老子”的私有财产是可以被儿子继承的吗，可是儿子为何却偏偏使用不了老子的私有财产呢？会不会是前面的证明有问题？

为了弄清楚这个问题，下面对示例进行改造：

清单：/Father.java
功能：演示被 private 关键字修饰的字段的继承机制

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {
    private int b1;
    private int b2;

    public Father(){
        this.b1 = 1;
        this.b2 = 2;
    }

    public int getB2() {
        return b2;
    }
    public void setB2(int b2) {
        this.b2 = b2;
    }

    public int getB1() {
```

```
        return b1;
    }
    public void setB1(int b1) {
        this.b1 = b1;
    }
}

class Son extends Father{
    public Son(){
        System.out.println("b1 = " + this.getB1());
        System.out.println("b2 = " + this.getB2());
    }

    public static void main(String[] args){
        Son son = new Son();
    }
}
```

运行 main()函数，打印如下结果：

```
b1 = 1
b2 = 2
```

通过本例可以看出，虽然子类不能直接访问父类的私有成员变量，但是却可以通过调用父类为私有成员变量所提供的公开接口访问父类的私有字段。

在 main()函数中执行 Son 的构造函数时，首先要执行父类的构造函数，这是众所周知的。当父类的构造函数执行完成之后，父类的 2 个字段便有了值，所以在子类中能够将父类的 2 个变量值打印出来。但是所谓父类的 2 个字段，其实已经在子类的内存空间里，因此执行父类的构造函数，其实是在初始化子类中所继承的父类部分的字段。

最终所分配的内存模型如图 6.20 所示。

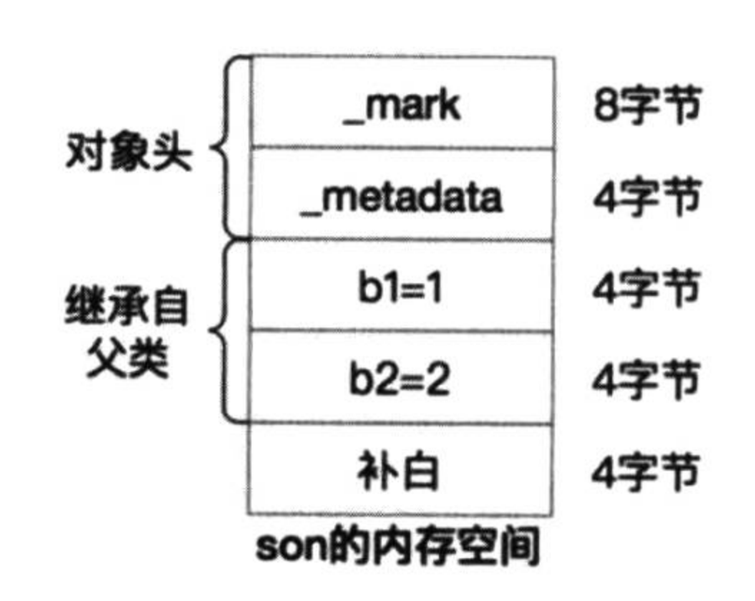

图 6.20　son 的内存空间布局

所以，关于 private 关键字的继承问题，应该可以使用如下 2 句话加以概括：

◎ “老子”的私有财产（特指私有成员变量）的确是被儿子继承的

◎ “儿子”虽然能够继承老子的私有财产，但是却没有权利直接支配，除非老子给儿子开放了接口，否则儿子不能动老子的私有财产。

6.3.3 使用 HSDB 验证字段分配与继承

使用 HSDB 验证 Java 字段的继承：父类 private 字段是否继承，父类 final 字段是否继承，父类 private/public/protected 字段是否被覆盖（引用类型与基本类型）。

前面讲了很多关于一个 Java 类字段大小、字段排序以及在继承的情况下字段覆盖的问题，并且使用工具类来测试了类的大小，从侧面验证了相关理论。但是 JDK 为大家提供了一个神器，其能够直接观察处于运行时的一个 Java 类真实的内存布局以及父类字段的继承与覆盖。

下面就与各位道友一起，通过 HSDB 来直接观察 Java 类在运行期的内存布局。

首先要有测试程序，示例如下，先定义一个父类 MyClass：

```
public abstract class MyClass {
    private Integer i = 1;

    protected long plong = 12L;

    protected final short s = 6;

    public char c = 'A';
}
```

接着定义一个子类继承自 MyClass：

```
public class Test extends MyClass{
    private long l;
    private Integer i = 3;
    private long plong = 18L;
    public char c = 'B';

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
```

```
        test.add(2, 3);
    }
}
```

验证的过程主要使用 HSDB 工具，该工具为 JDK 自带，是一套视窗系统，能够在视窗里面通过界面或者命令行查看运行过程中的若干数据，包含堆栈、堆、perm（JDK8 已经没有 perm 区的概念）、实例数据、常量池，以及与实例所对应的 JVM 内部类 instanceKlass 等。工具的使用也很简单，启动 Java 程序，设置断点后，使用 HSDB 连上 Java 进程即可进行观察。

对于本示例，在 add()函数的 Test test = this 这一句代码上打上断点（可以使用 jdb 命令打断点调试，也可以使用 IDE 集成化的工具），然后使用 HSDB 连接。

本示例使用 jdb 进行调试，首先编译示例程序，得到 Test.class，接着进入 Test.class 所在目录，运行下面命令：

```
jdb -XX:+UseSerialGC -Xmx10m -XX:-UseCompressedOops
```

若在 64 位机器上运行 jdb，为了能够正常使用 HSDB，必须加上-XX:-UseCompressdOops 命令，该命令取消 JVM 的指针压缩功能。如果不加上该选项，则 HSDB 无法正确获取运行期各种数据的内存地址。

执行 jdb 命令后，接着执行下面几个命令：

```
$ javac Test.java
$ jdb -XX:+UseSerialGC -Xmx10m -XX:-UseCompressedOops
正在初始化 jdb...
> stop in Test.add
正在延迟断点 Test.add。
将在加载类后设置。
> run Test
运行 Test
设置未捕获的 java.lang.Throwable
设置延迟的未捕获的 java.lang.Throwable
>
VM 已启动：设置延迟的断点 Test.add

断点命中："线程=main", Test.add(), 行=16 bci=0
16              Test test = this;

main[1] next
>
已完成的步骤："线程=main", Test.add(), 行=17 bci=2
17              int z = a + b;

main[1]
```

上述过程中主要执行了 stop in Test.add、run Test 和 next 这 3 条 jdb 命令。Stop in Test.add 表示在 Test 类的 add()方法处设断点；接着执行 run Test，表示开始启动 Test 类的 main()函数，Test 的 main()函数运行后，jdb 会在 Test.add()方法的第一行代码上暂停，此时执行 next 命令，于是程序进入 Test.add()方法的第 2 句代码。

Test 类的断点调试就到这里，接着执行 jps 命令，得到 Test 这个 Java 进程的进程 ID，然后启动 HSDB，单击 HSDB 的 File->Attach to hotspot process 菜单选项，输入 Test 进程号，即可进入 HSDB 界面。

刚进入 HSDB 时的界面如图 6.21 所示。

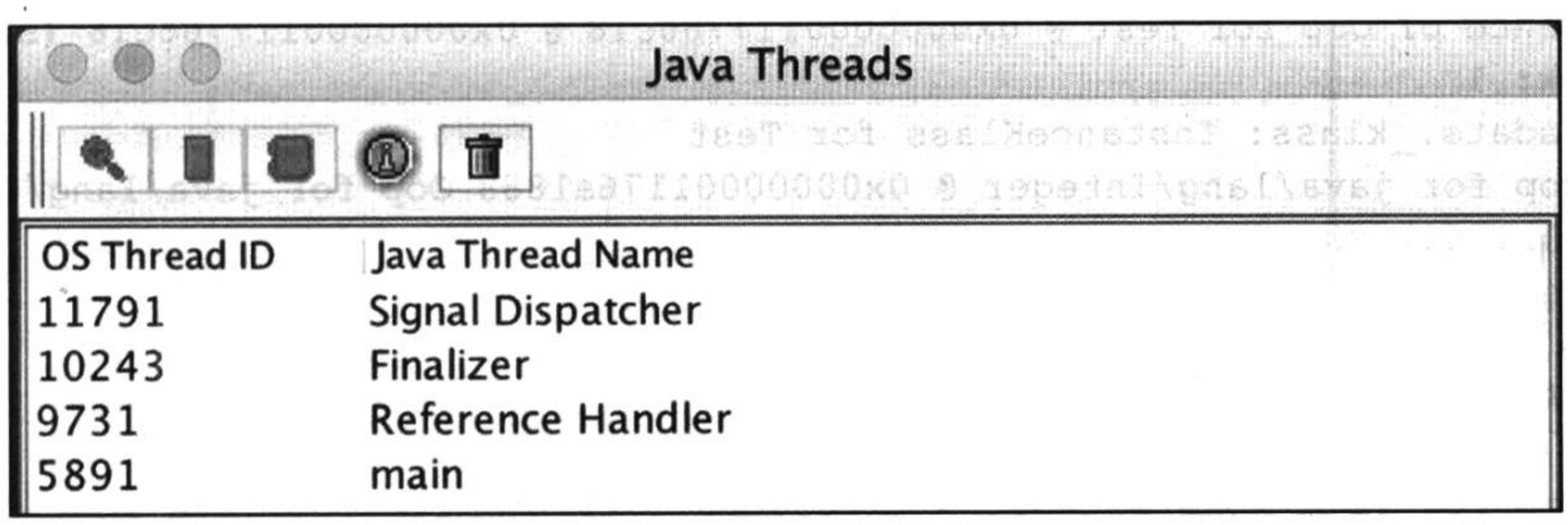

图 6.21　HSDB 连接上 Java 进程后的主窗口

接着单击 HSDB 的 Windows->Console 菜单选项，打开 HSDB 的控制台，刚打开的控制台一片空白，按下回车键，就会出现 hsdb>提示符。接着输入 universe 命令，查看当前 Java 进程的堆内存，如下所示：

```
hsdb> universe
Heap Parameters:
Gen 0:   eden [0x0000000117600000,0x0000000117770a90,0x00000001178b0000) space
capacity = 2818048, 53.58432503633721 used
     from [0x00000001178b0000,0x00000001178b0000,0x0000000117900000) space
capacity = 327680, 0.0 used
     to   [0x0000000117900000,0x0000000117900000,0x0000000117950000) space
capacity = 327680, 0.0 usedInvocations: 0

Gen 1:   old  [0x0000000117950000,0x0000000117950000,0x0000000118000000) space
capacity = 7012352, 0.0 usedInvocations: 0
```

由于该程序运行于 JDK 8 之上，因此并没有显示 perm space。如果是 JDK 6，则会显示。这里需要注意，虽然 JDK 6 与 JDK8 的内存模型有一些重大不同，但是内部类模型并没有根本上的改变，因此使用 JDK 8 来运行程序，一样能够完成演示和验证。

接着输入 scanoops 命令，在内存中搜索 Test 类实例，命令如下：

```
hsdb> scanoops 0x0000000117600000 0x0000000118000000 Test
0x0000000117766c18  Test
```

scanoops 命令接受 3 个参数，分别是搜索的起始地址、终止地址及要搜索的类实例名称。这里所输入的搜索起始地址与终止地址，分别是上面使用 universe 命令所计算出的 Gen 0 的起始地址和 Gen 1 的结束地址。

如果能够搜索到类实例对象，scanoops 命令会显示该实例在 JVM 内部对应的 instanceOop 的内存首地址。果然这里搜索到了一个 Test 类的实例地址，该地址是 0x0000000117766c18。可以使用 inspect 命令查看这个地址处的 oop 的全部数据，如下：

```
hsdb> inspect 0x0000000117766c18
instance of Oop for Test @ 0x0000000117766c18 @ 0x0000000117766c18 (size = 72)
_mark: 1
_metadata._klass: InstanceKlass for Test
i: Oop for java/lang/Integer @ 0x00000001176a1888 Oop for java/lang/Integer @
0x00000001176a1888
plong: 12
s: 6
c: 'A'
l: 0
i: Oop for java/lang/Integer @ 0x00000001176a18b8 Oop for java/lang/Integer @
0x00000001176a18b8
plong: 18
c: 'B'
```

在 JDK 8 中，Java 类在 JVM 内部所对应的 oop 对象的结构仍然是对象头后面跟着一群类成员变量。除了可以在 HSDB 的 console 命令行中使用 inspect 命令查看 oop 对象外，也可以使用图形化的 Inspect 界面来观察。单击 HSDB 的 Tools->Inspector 菜单选项，输入地址即可，如图 6.22 所示。

图 6.22 所展示的内存布局与前面使用 inspect 命令所得到的内存布局是完全相同的。从图中可以看到，Test 类所对应的 oop 对象头后面一共跟了 8 个字段，其中对象头与这 8 个字段如图 6.23 所示。

Test 类中其实仅定义了 4 个字段，但是其对应的 JVM 内部 oop 对象头后面却跟了 8 个字段，很显然，另外 4 个就是 Test 类的父类 MyClass 中的字段。父类与子类字段的划分如图 6.24 所示。

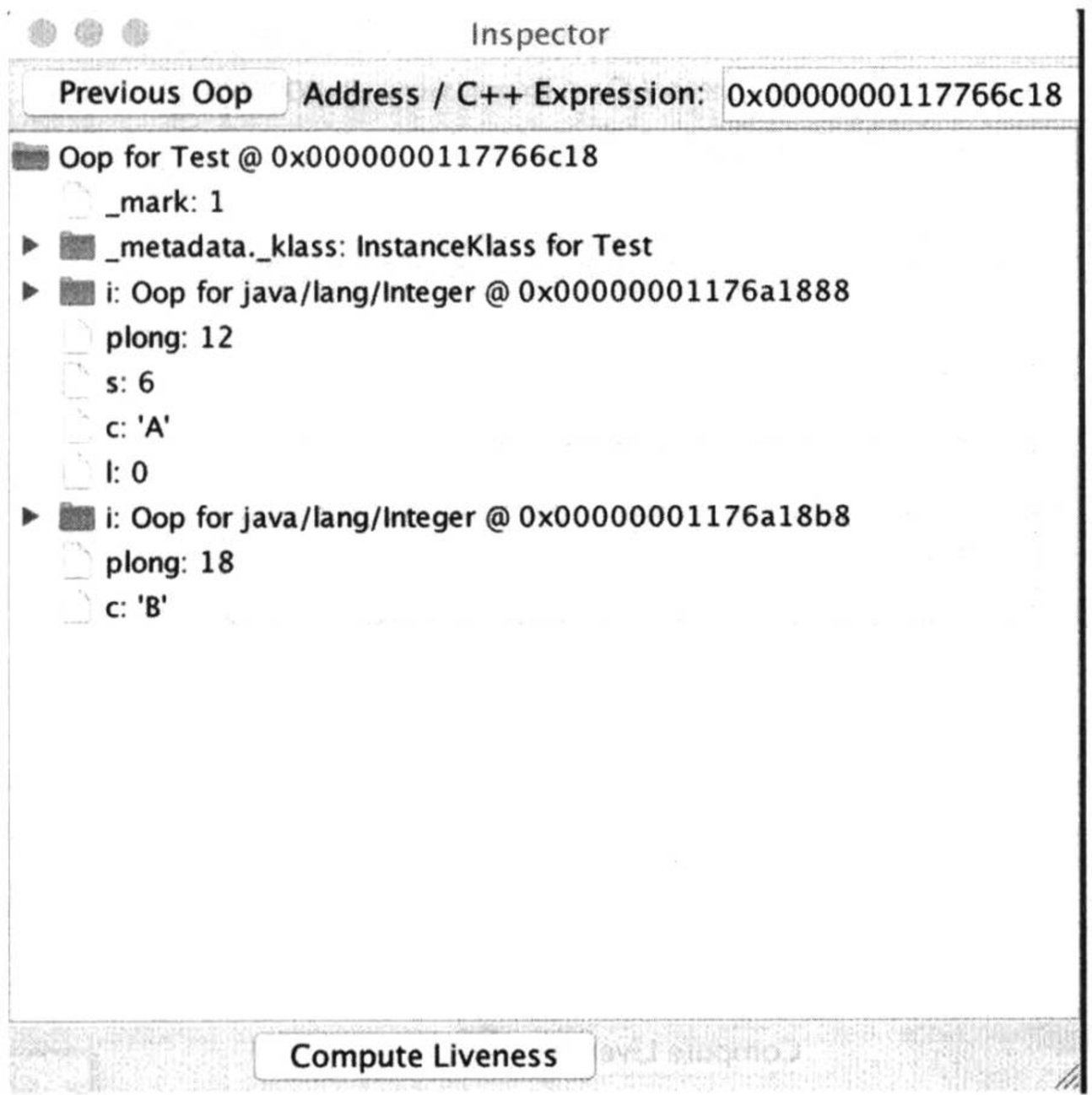

图 6.22　使用 HSDB 的 Inspector 工具查看 instanceKlassOop 结构

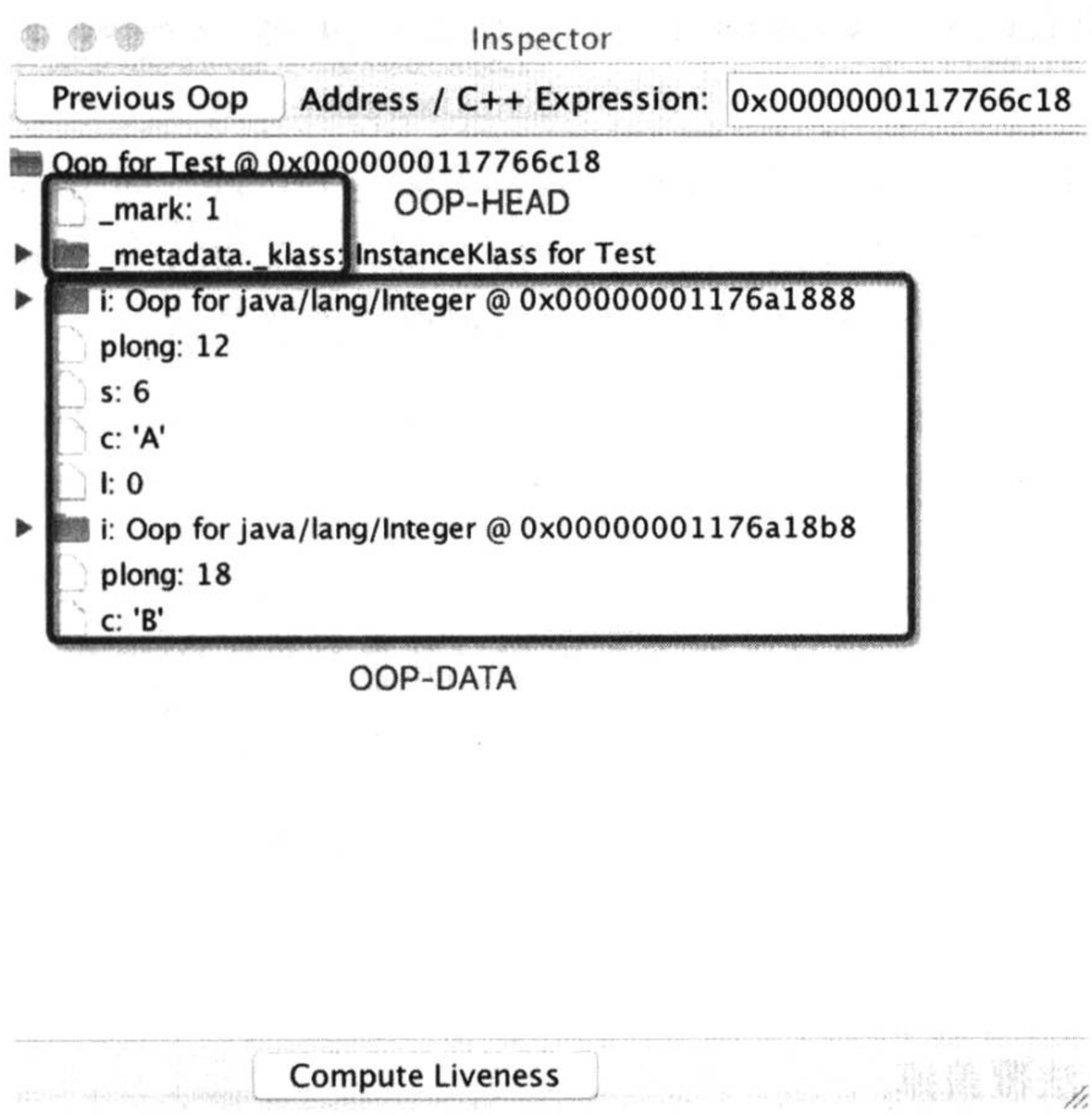

图 6.23　oop 对象头与对象头后面的字段

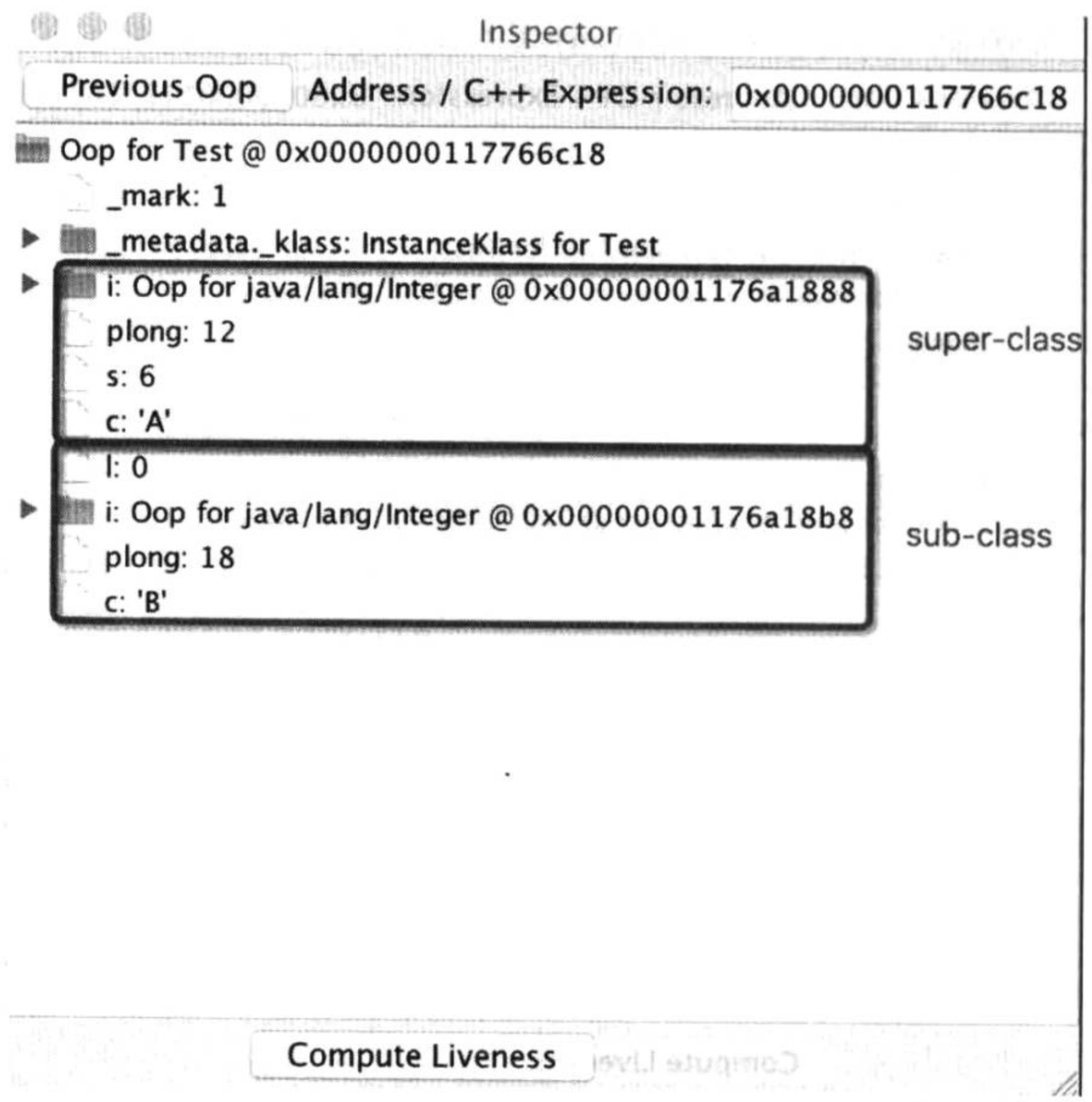

图 6.24　Test 类所对应的 oop 中父类与子类字段的划分

从图 6.24 中可以看出来，父类的 4 个字段分别是 i、plong、s 和 c，子类的 4 个字段分别是 l、i、plong 和 c。

父类 MyClass 中的字段 i 类型是 Integer，访问权限是 private。而子类 Test 也定义了一个同名、同类型的字段 i，从图 6.24 可以看出来，JVM 内部在 Test 这个子类的实例对象中，同时为父类和子类的变量 i 都开辟了内存空间，由此可以得出下面这条结论：

如果子类定中义了与父类同名的字段，并且当父类中的字段访问权限是 private 时，子类不会覆盖父类的字段，JVM 会在内存中同时为父类和子类的该相同字段各分配一段内存空间。

同样，各位道友其实也早都看出来了，父类 MyClass 中另外定义的 2 个变量——plong 和 c，子类 Test 中也都定义了一模一样的字段名和类型，并且父类中这两个字段的访问权限分别是 protected 与 public，但是子类依然没有覆盖。由此可以得出下面这条结论：

当子类中定义了与父类相同名称、相同类型的字段时，无论父类中字段的访问权限是什么，也无论父类字段是否被 final 修饰，子类都不会覆盖父类字段。

这一点与 Java 类继承概念中的方法继承大不相同，如果子类重写(override)了父类的方法，则子类会将父类的方法覆盖掉。

虽然子类的字段不会覆盖父类字段，这意味着“儿子”会全盘接纳“老子”的全部财产，

但是并不等于儿子就有权直接支配老子的财产。前面也举例分析过，对于父类中被定义为 private 的字段，这部分字段属于“老子”的私有财产，儿子不能直接访问（例如，儿子不能直接通过 super.xxx 来使用），除非“老子”开放了 getXXX()这样的公共接口，否则儿子无论如何都访问不了老子的私有字段。这一点倒是与方法的继承如出一辙。

类成员变量的偏移量

在上面使用 inspect 命令查看 Test 类实例所对应的 oop 的内存分布时（注意，不是 Inspect 视窗），显示了 oop 的内存大小为 72 字节。而当使用 Inspect 窗口查看这个 Test 类的实例时，字段的显示顺序却与父类中各个字段所定义的顺序相同，如果按照这种顺序来分配内存，那么 Test 类的实例大小可能会大于 72 字节（考虑字段对齐的情况）。

实际上，无论是 inspect 命令还是 Inspect 视窗，所显示的字段顺序都不是内存中真正分配的顺序。但是 HSDB 提供了其他窗口可以查看各个字段的位置，单击 HSDB 工具栏的 Tools-> Class Browser 命令，输入 Test，便能看到 Test 类中所定义的各个字段的偏移量，如图 6.25 所示。

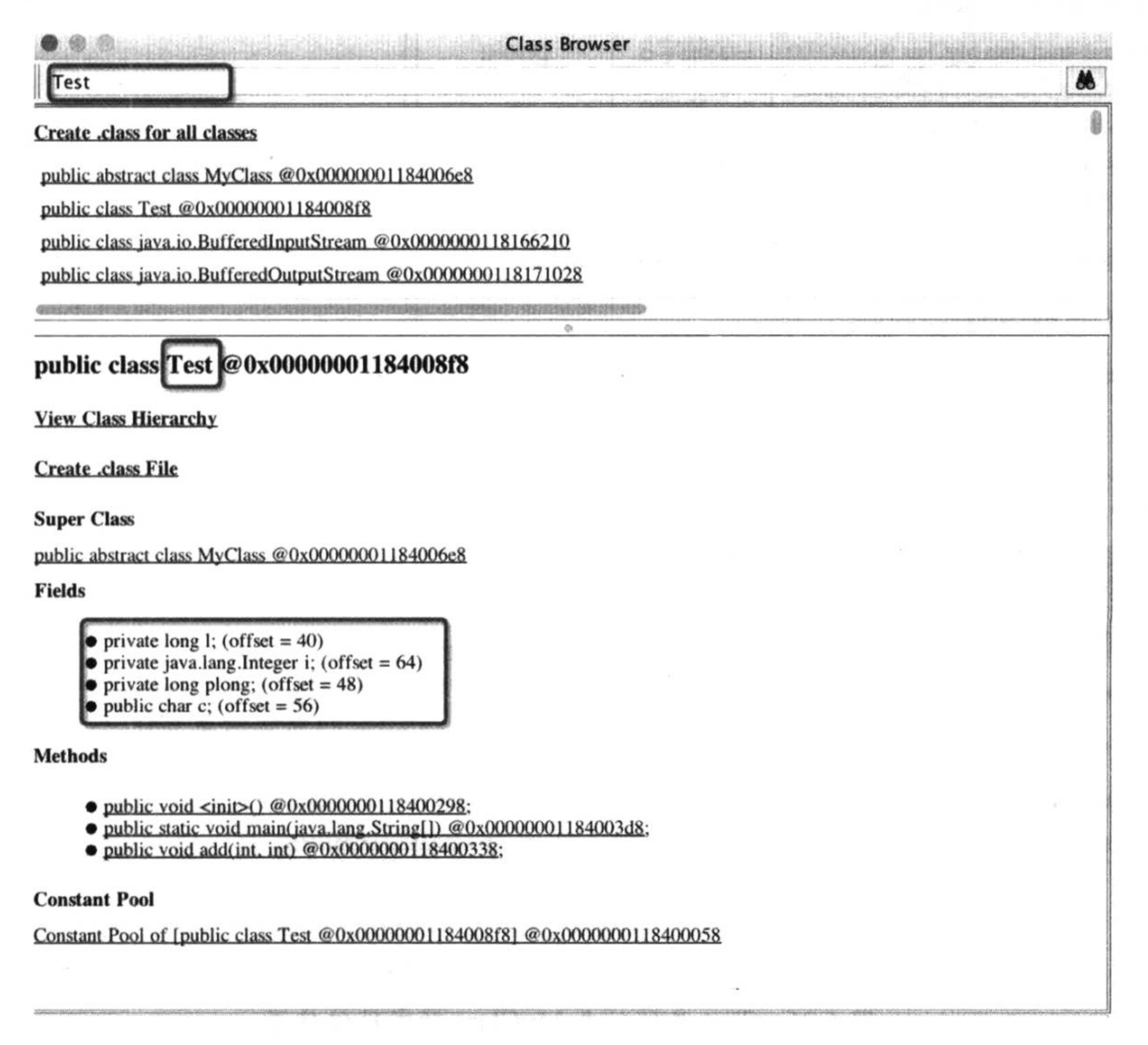

图 6.25　Test 类的字段偏移

图 6.25 显示出 Test 类中所定义的 4 个字段，并且显示出每个字段的偏移量。各个字段的偏移量如下（按照由小到大排列）：

```
private long l:     offset = 40
private long plong:     offset = 48
public char c:     offset = 56
private java.lang.Integer i:     offset = 64
```

可以看出，偏移量最小的字段 l，其偏移量也为 40，而 Test 在 JVM 内部所对应的 oop 的对象头在 64 位机器上最大也仅占用 16 字节，很显然，这是因为在 Test 类自己的字段域与 oop 对象头之间还存在其他数据，而这些数据正是 Test 父类 MyClass 的字段域。

图 6.25 显示了 Test 类的 Super Class，单击 Test 类的父类 MyClass，就显示出 MyClass 类的全部信息，如图 6.26 所示。

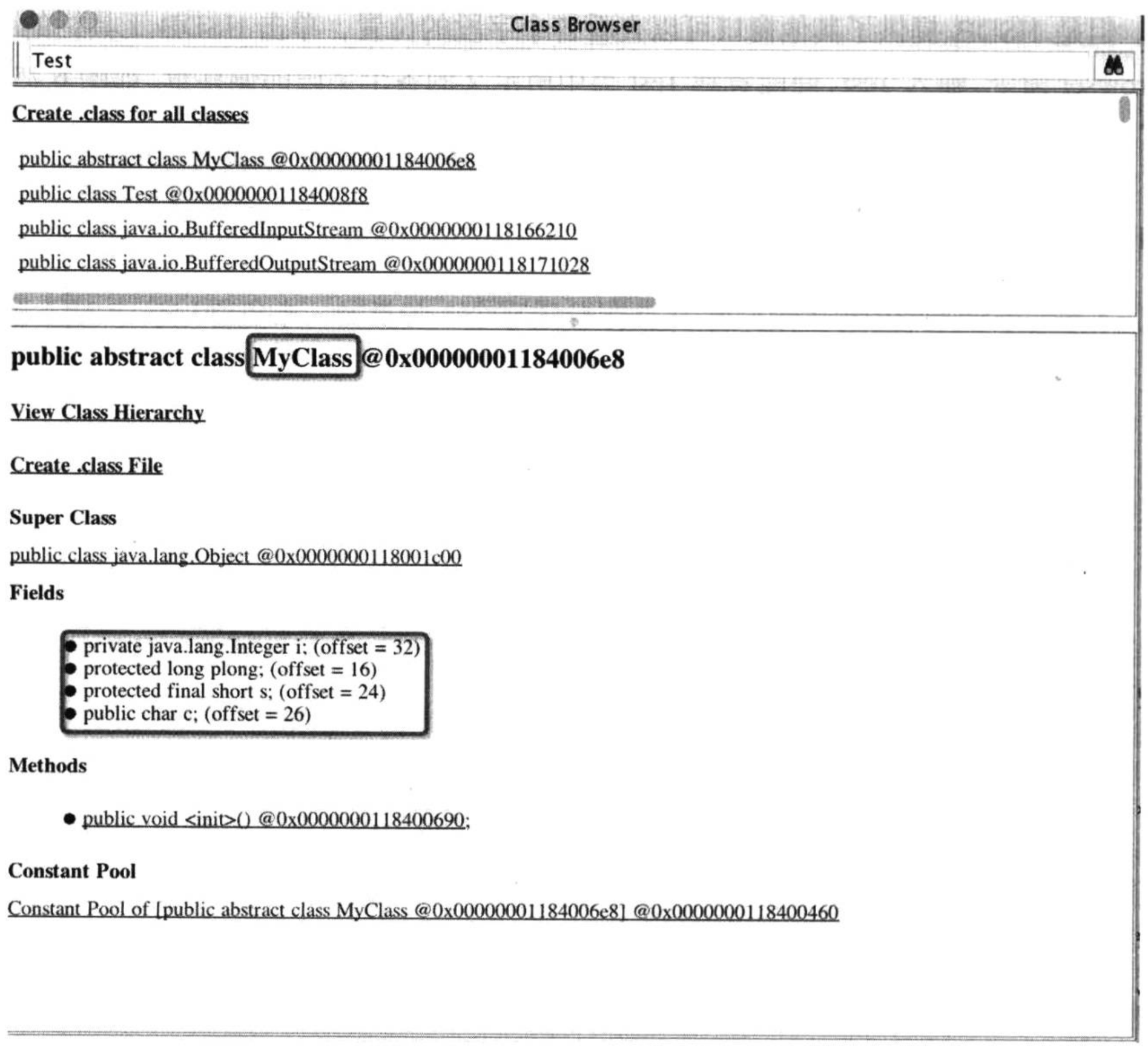

图 6.26　Test 类的父类 MyClass 的字段偏移量

图 6.26 显示了父类 MyClass 中的全部字段及各个字段的偏移量。各个字段的偏移量按照由小到大的顺序分别如下：

```
protected long plong:    offset = 16
protected final short s:    offset = 24
public char c:    offset = 26
private java.lang.Integer i:    offset = 32
```

偏移量最小的字段是 plong，其偏移量是 16，这正好位于 Test 类在 JVM 内部的 oop 对象的对象头之后。而偏移量最大的字段是变量 i，其偏移量是 32。Test 类在 JVM 内部的字段域被分配在其父类字段域之后，而父类 MyClass 的字段域的最后一个字段 i 的偏移量是 32，因此 Test 类的第一个字段 l 的偏移量是 40。

Test 类所对应的 oop 的完整内存布局如图 6.27 所示。

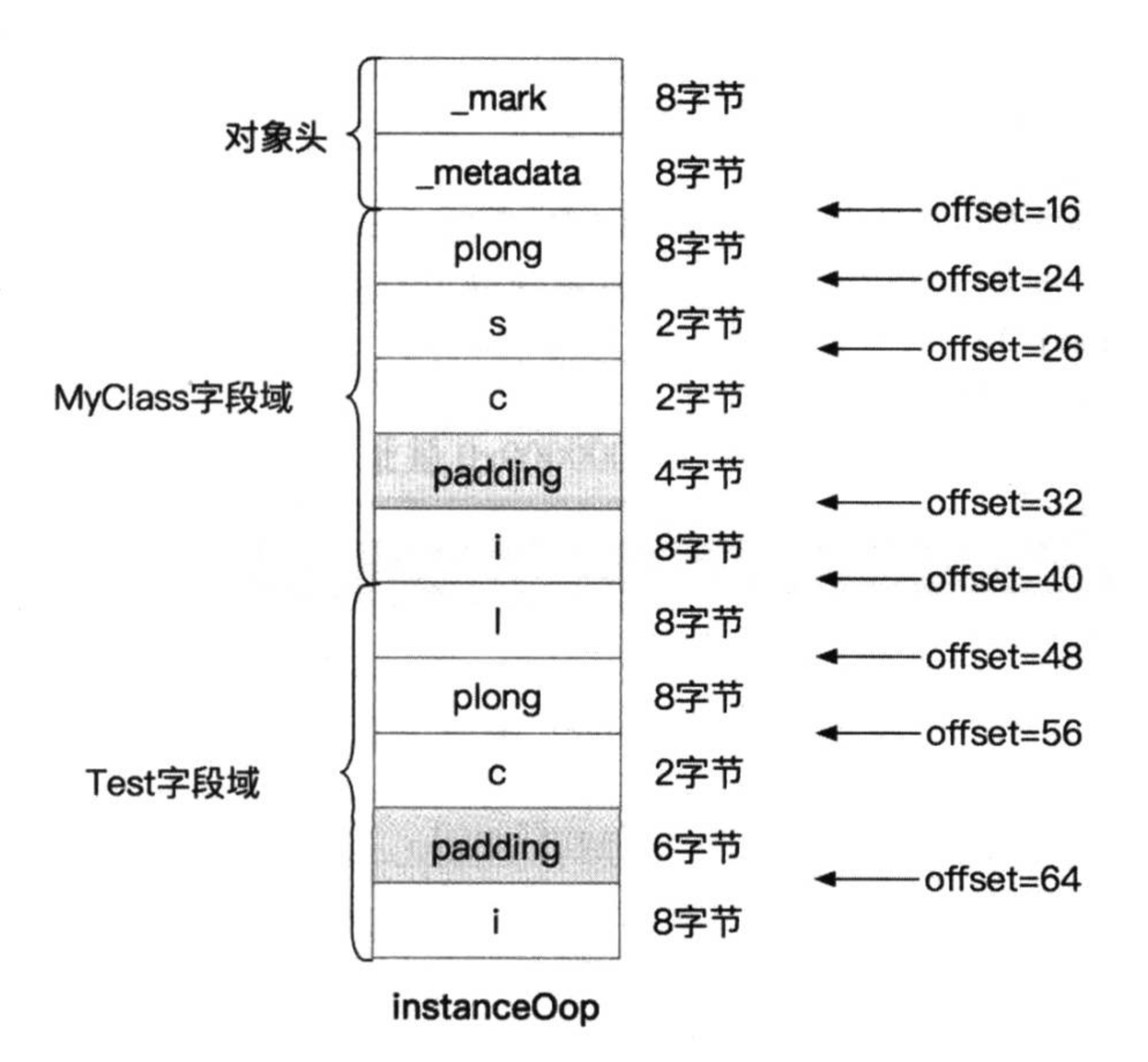

图 6.27　Test 类所对应的 instanceOop 的内存布局（64 位平台，关闭指针压缩选项）

图 6.27 显示出 Test 类所对应的 instanceOop 的完整内存布局，并且显示出各个字段所占的内存大小、起始偏移量。注意，JVM 为了字段对齐，在 instanceOop 里面有两处做了对齐补白。

各位道友可以对照图 6.27 所示内存布局，与 HSDB 中所显示的父类与子类中各个字段的偏移量做一比较，再次感受下 JVM 分配字段的机制。

6.3.4 引用类型变量内存分配

在上面 Test 类的示例中，在 Test 类中定义了引用类型的类成员变量，即 private Integer i = 3。同时，在 main()主函数中实例化了 Test 类，得到实例 test。变量 i 与 main()中的局部变量 test，一个是类的成员变量，一个是 Java 方法内的局部变量，但是两者都是引用类型。一起来围观引用类型在两种不同场景下的内存分配吧。

首先看类成员变量 i。变量 i 既然是 Test 类的成员变量，其应该也在 Test 类实例所对应的 instanceOop 的字段域之中，使用 HSDB 的 Inspect 视窗查看在 main()主函数中所创建的 Test 类实例，如图 6.28 所示。

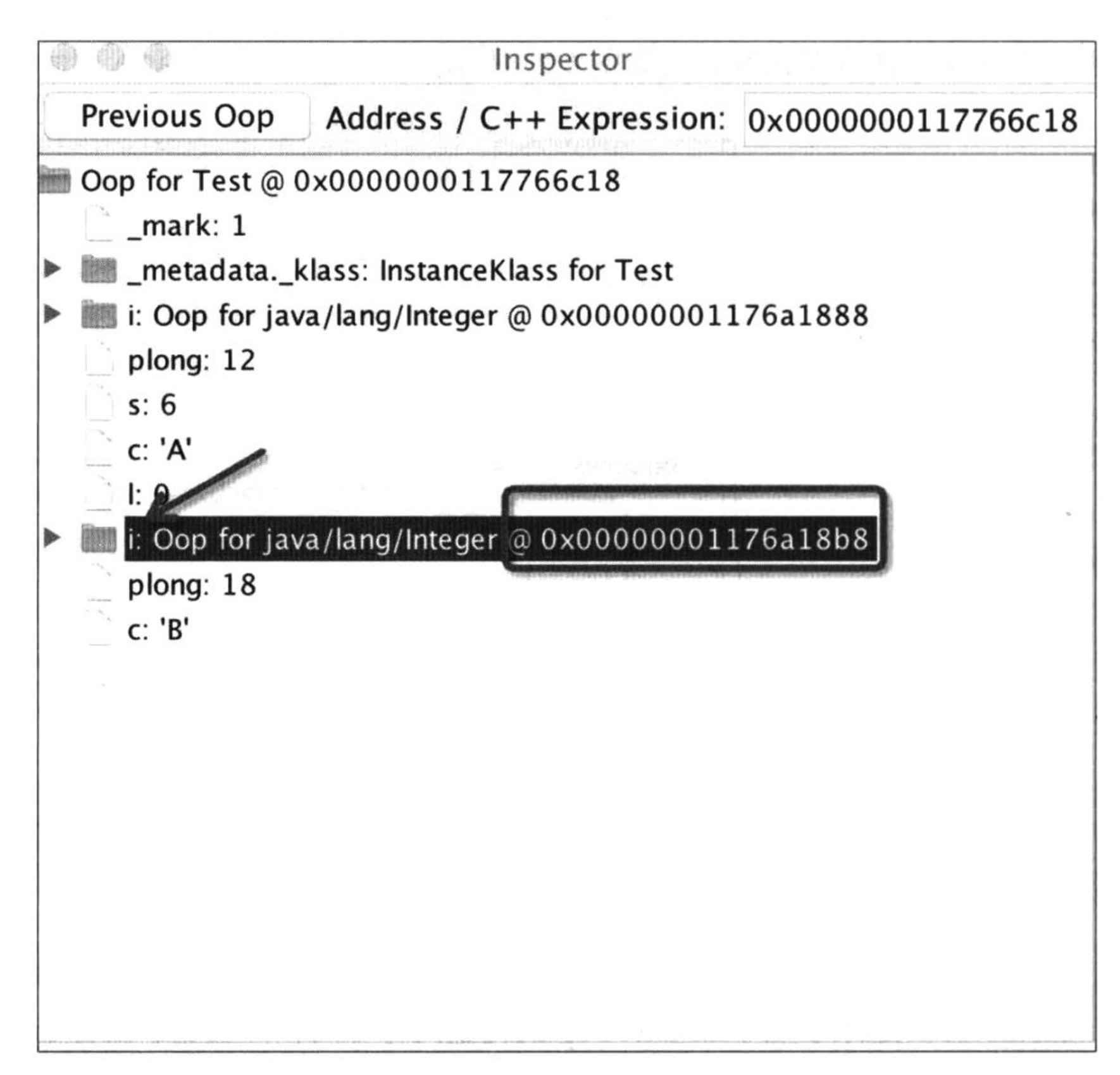

图 6.28 Test 类实例中的成员变量 i

图 6.28 显示出 Test 类的成员变量 i 的位置和值，其值指向一个 java/lang/Integer 类型所对应的 oop，该 oop 地址是 0x00000001176a18b8。再使用 HSDB 的 Inspect 视窗查看 Integer 的这个实例地址，如图 6.29 所示。

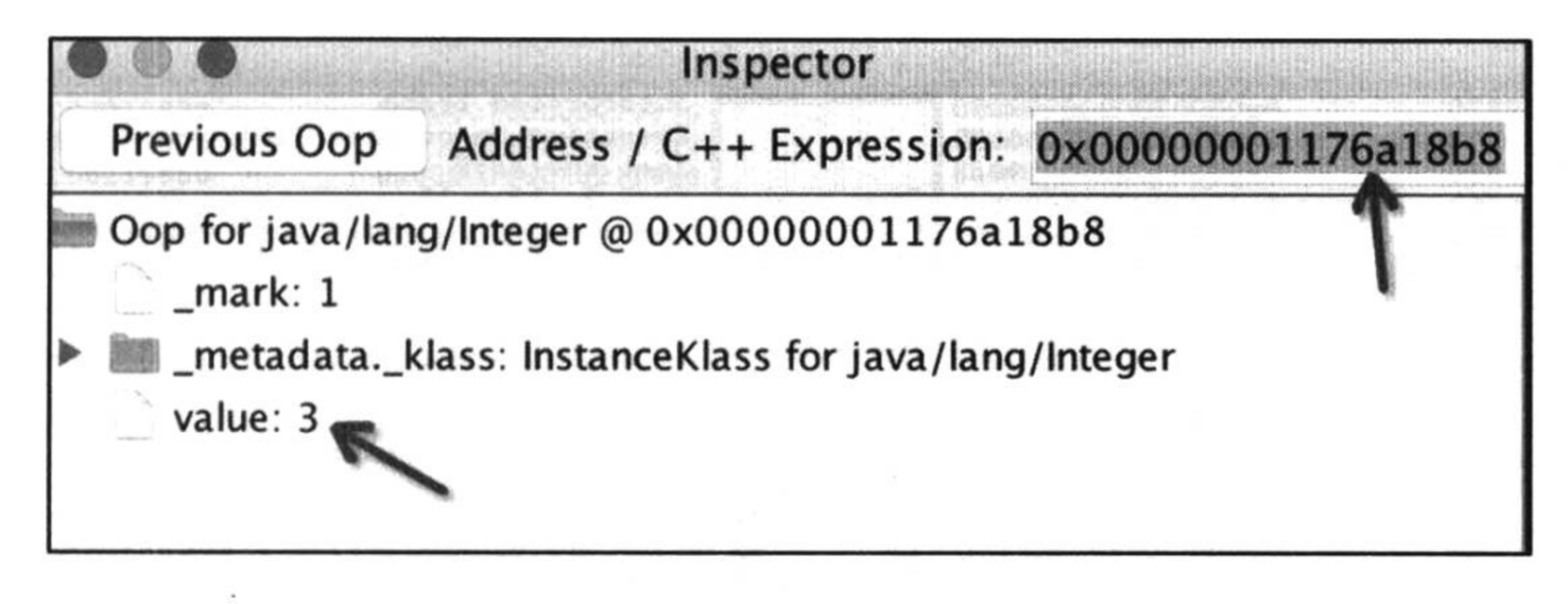

图 6.29　Integer 实例的内存分布

由此可知，Test 类中的成员变量 i 的实际数值被存储在 java.lang.Integer 类型实例在 JVM 内部所对应的 instanceOop 的字段域之中，而在 Test 类实例所对应的 instanceOop 中的成员变量 i 所存储的仅仅是一个指针引用。Test 类的成员变量 i 的指针引用存储在 Test 类实例所对应的 instanceOop 中，i 指针指向 java.lang.Integer 的实例 instanceOop。无论是 Test 类的实例 instanceOop 还是 java.lang.Integer 类的实例 instanceOop，都位于 JVM 的堆内存中。所以可以得出结论：

类的成员变量的引用（指针）和类成员变量的实例都分配在 JVM 的堆内存中，类的成员变量的引用分配在所在类的实例 instanceOop 的字段域之中。

接着看 Java 方法内引用类型的局部变量的内存分配位置。在 main()主方法里实例化了一个 Test 类的对象 test，test 对象的内存地址是 0x0000000117766c18，由于 test 属于 main()方法的局部变量，因此在 main()方法的栈帧里一定存在对这个地址的引用。HSDB 支持查看一个线程的整体堆栈内存，使用 HSDB 刚刚连接上 Java 进程时，会显示如图 6.30 所示的 Java Thread 窗口，该窗口主要显示当前 Java 进程的所有线程，其中最下面可以看到 main 主线程，选中该线程，单击该窗口上方一排工具按钮中的第 2 个按钮，如图 6.30 所示。

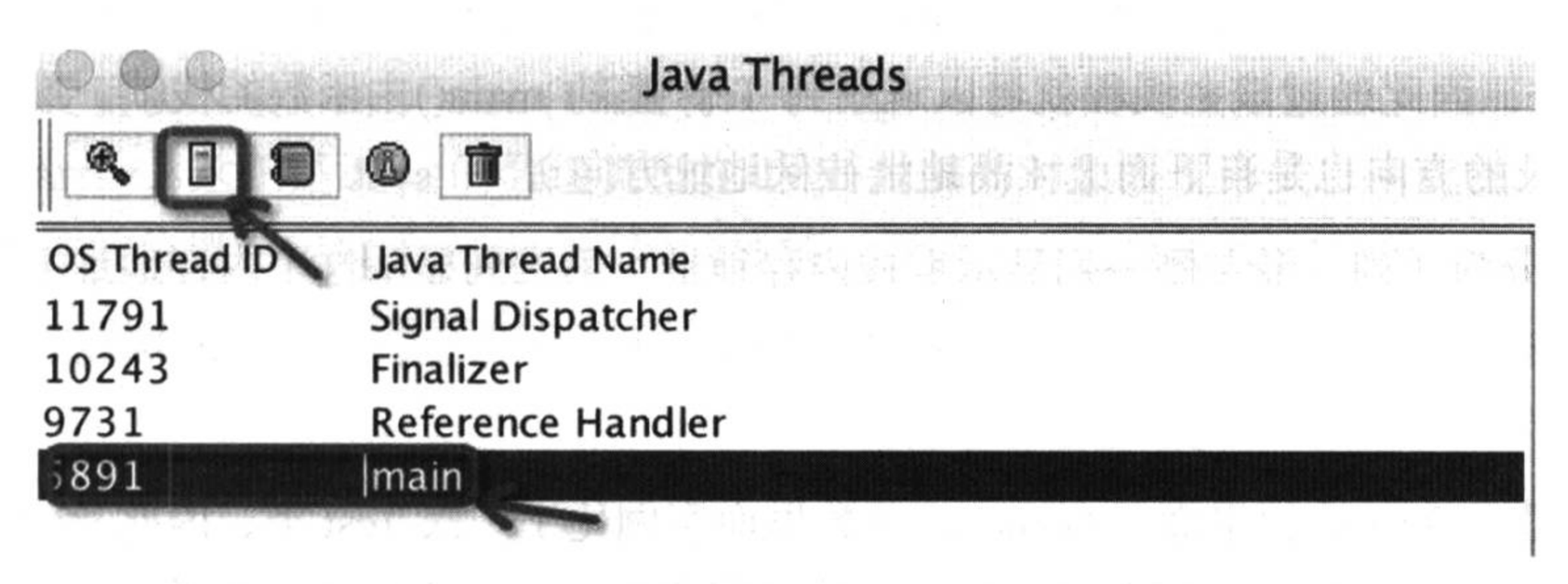

图 6.30　选中 Java 主线程并查看线程堆栈

单击第 2 个工具按钮后，会弹出新的窗口显示该线程详细的堆栈内存，如图 6.31 所示。

Stack Memory for main

0x0000700000219870	0x0000700000219870	Interpreted frame
0x0000700000219878	0x0000000118400322	Executing in codelet "safepoint entry points" at PC
0x0000700000219880	0x00007000002198e8	Test.add(II)V
0x0000700000219888	0x0000000118400be8	@bci 2, line 17
0x0000700000219890	0x0000000000000000	Interpreter constant pool cache
0x0000700000219898	0x0000000118400338	Interpreter frame Method*
0x00007000002198a0	0x0000000000000000	
0x00007000002198a8	0x00007000002198d8	
0x00007000002198b0	0x0000700000219930	Interpreter locals area for frame with SP = 0x00007
0x00007000002198b8	0x0000000108190ca4	
0x00007000002198c0	0x0000000000000000	Interpreted frame
0x00007000002198c8	0x0000000000000000	Executing in codelet "invoke return entry points" a
0x00007000002198d0	0x0000000117766c18	Test.main([Ljava/lang/String;)V
0x00007000002198d8	0x0000000000000003	@bci 11, line 29
0x00007000002198e0	0x0000000000000002	Interpreter expression stack
0x00007000002198e8	0x0000000117766c18	NewGen Test
0x00007000002198f0	0x00007000002198f0	NewGen Test
0x00007000002198f8	0x00000001184003cb	
0x0000700000219900	0x0000700000219948	
0x0000700000219908	0x0000000118400be8	Interpreter constant pool cache
0x0000700000219910	0x0000000000000000	
0x0000700000219918	0x00000001184003d8	Interpreter frame Method*
0x0000700000219920	0x00007000002198d8	
0x0000700000219928	0x0000700000219948	
0x0000700000219930	0x00007000002199b0	Interpreter locals area for frame with SP = 0x00007
0x0000700000219938	0x00000001081897a0	
0x0000700000219940	0x0000000117766c18	NewGen Test
0x0000700000219948	0x0000000117766c00	NewGen ObjArray @ 0x0000000117766c00
0x0000700000219950	0x0000000100001fa0	
0x0000700000219958	0x000000000000000a	
0x0000700000219960	0x00007fd6cc003000	
0x0000700000219968	0x0000000000000001	
0x0000700000219970	0x000000010818971f	
0x0000700000219978	0x00000001184003d8	
0x0000700000219980	0x0000700000219a48	
0x0000700000219988	0x0000700000219dd0	
0x0000700000219990	0x000000010000000a	
0x0000700000219998	0x00000001184003d8	
0x00007000002199a0	0x000000010819ec20	
0x00007000002199a8	0x0000700000219b88	
0x00007000002199b0	0x0000700000219b60	
0x00007000002199b8	0x00000001074eab0e	
0x00007000002199c0	0xa000000000000001	
0x00007000002199c8	0x00007fd6cc003000	
0x00007000002199d0	0xa000000000000000	
0x00007000002199d8	0x0000700000219dd0	
0x00007000002199e0	0x0000000000000000	
0x00007000002199e8	0x000000010819ec20	

图 6.31　HSDB 显示线程堆栈详情

图 6.31 里显示出当前 Test 进程的 main 主线程，由于 Java 的主线程最终会调用 Java 程序的 main()主函数，因此通过该主线程就可以观察到 Test 类的 main()主函数的栈帧。堆栈由下往上看，堆栈增长的方向也是自下而上（高地址往低地址方向）。

图 6.31 分为 3 列，最左侧一列显示堆栈内存地址，第二列显示这个内存地址上的数据，最右侧显示关键内存区域的属性。由于 Test 类运行于 64 位平台上，因此最左侧相邻行之间的地址相差 8 字节。

由于在整个 Test 进程里面只有 main()函数里面实例化了一次 Test 类，因此 main()函数的堆栈必然会第一个引用 Test 类实例的地址，因此只需要在 main()的栈帧里搜索 test 这个实例的地址。仔细观察这张 main 主线程的堆栈图，自下而上搜索 Test 类实例的地址，该地址是 0x0000000117766c18。果然能够找到，如图 6.32 所示。

Stack Memory for main

0x0000700000219870	0x0000700000219870	Interpreted frame
0x0000700000219878	0x0000000118400322	Executing in codelet "safepoint entry points" at PC
0x0000700000219880	0x00007000002198e8	Test.add(II)V
0x0000700000219888	0x0000000118400be8	@bci 2, line 17
0x0000700000219890	0x0000000000000000	Interpreter constant pool cache
0x0000700000219898	0x0000000118400338	Interpreter frame Method*
0x00007000002198a0	0x0000000000000000	
0x00007000002198a8	0x00007000002198d8	
0x00007000002198b0	0x0000700000219930	Interpreter locals area for frame with SP = 0x00007
0x00007000002198b8	0x0000000108190ca4	
0x00007000002198c0	0x0000000000000000	Interpreted frame
0x00007000002198c8	0x0000000000000000	Executing in codelet "invoke return entry points" a
0x00007000002198d0	0x0000000117766c18	Test.main([Ljava/lang/String;)V
0x00007000002198d8	0x0000000000000003	@bci 11, line 29
0x00007000002198e0	0x0000000000000002	Interpreter expression stack
0x00007000002198e8	0x0000000117766c18	NewGen Test
0x00007000002198f0	0x00007000002198f0	NewGen Test
0x00007000002198f8	0x00000001184003cb	
0x0000700000219900	0x0000700000219948	
0x0000700000219908	0x0000000118400be8	Interpreter constant pool cache
0x0000700000219910	0x0000000000000000	
0x0000700000219918	0x00000001184003d8	Interpreter frame Method*
0x0000700000219920	0x00007000002198d8	
0x0000700000219928	0x0000700000219948	
0x0000700000219930	0x00007000002199b0	Interpreter locals area for frame with SP = 0x00007
0x0000700000219938	0x00000001081897a0	
0x0000700000219940	0x0000000117766c18	NewGen Test
0x0000700000219948	0x0000000117766c00	NewGen ObjArray @ 0x0000000117766c00
0x0000700000219950	0x0000000100001fa0	
0x0000700000219958	0x000000000000000a	
0x0000700000219960	0x00007fd6cc003000	
0x0000700000219968	0x0000000000000001	
0x0000700000219970	0x000000010818971f	
0x0000700000219978	0x00000001184003d8	
0x0000700000219980	0x0000700000219a48	
0x0000700000219988	0x0000700000219dd0	
0x0000700000219990	0x000000010000000a	
0x0000700000219998	0x00000001184003d8	
0x00007000002199a0	0x000000010819ec20	
0x00007000002199a8	0x0000700000219b88	
0x00007000002199b0	0x0000700000219b60	
0x00007000002199b8	0x00000001074eab0e	
0x00007000002199c0	0xa000000000000001	
0x00007000002199c8	0x00007fd6cc003000	
0x00007000002199d0	0xa000000000000000	
0x00007000002199d8	0x0000700000219dd0	
0x00007000002199e0	0x0000000000000000	
0x00007000002199e8	0x000000010819ec20	

图 6.32　寻找 main 主线程堆栈所引用的 Test 类实例

由于这个位置是第一个对 Test 实例对象地址的引用位置，因此该位置一定属于 main()主函数的局部变量表区域。由于 main()函数有一个 args 入参，因此该位置的下面那个位置的数据类型是 ObjArray，这正是 Java 的数组类型在 JVM 内部的对象表现形式，由此更加确定图 6.32 中方框所框住的位置一定属于 main()函数的栈帧，而这一点也能够反向证明 test 这个局部变量的引用位于其所在方法的栈帧之中。其内存布局如图 6.33 所示。

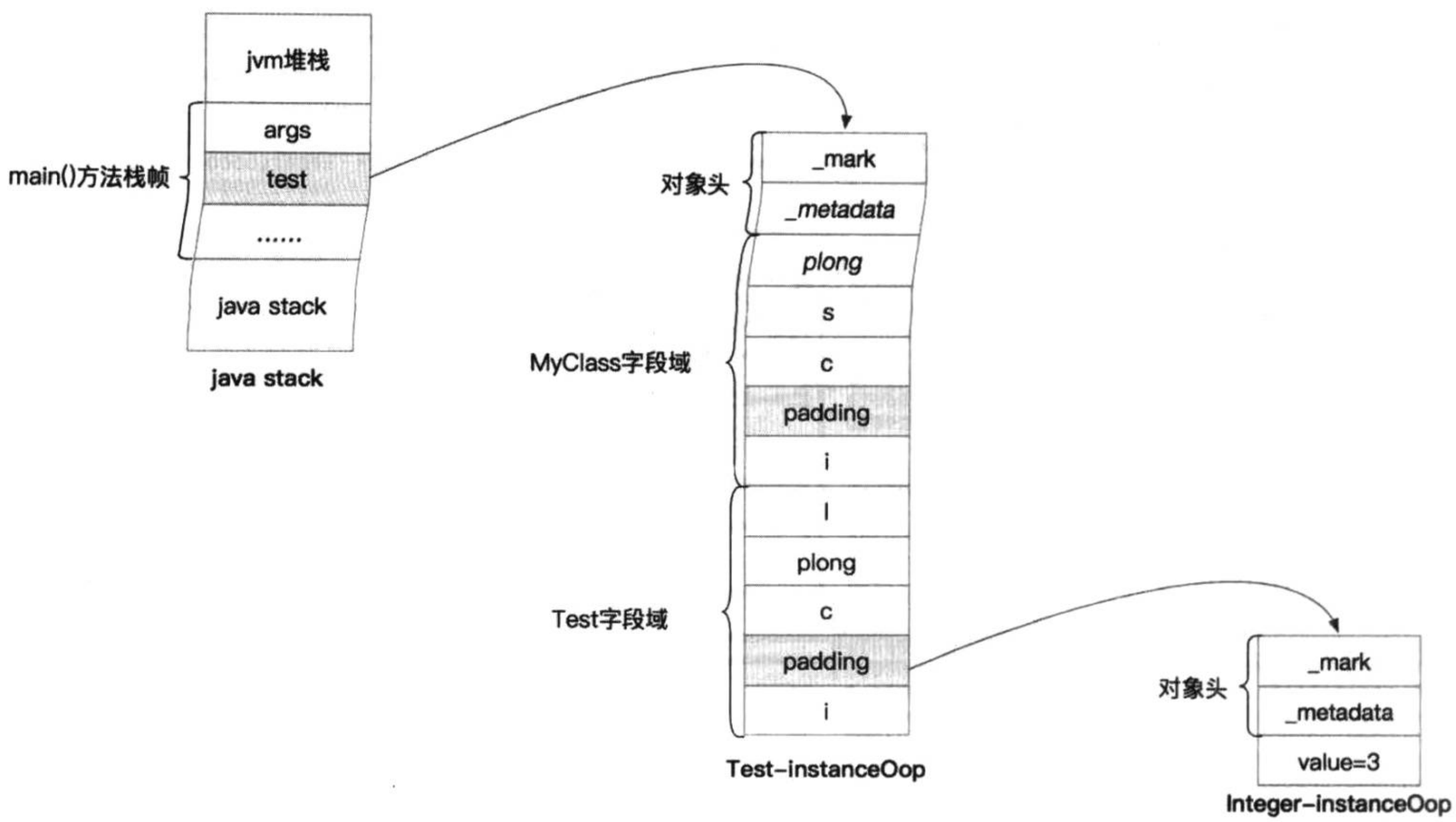

图 6.33　main()函数中的 Test 实例及 Test 类引用成员变量的内存布局

由此可以确定，类成员变量的引用都被分配在堆内存中。

6.4　本章总结

总体而言，HotSpot 解析 Java 类变量的脉络比较清晰，但是也可以看出花了很多心思，这导致 JVM 虽然在执行引擎上相比于那些直接编译成本地机器码的编程语言可能要稍逊一筹，但是在对象的内存分配上，并不比这些编程语言多浪费一点空间（除了每个 Java 类对象必须保留一个对象头），甚至由于字段重排的优化策略，对内存的利用率还要高于这些编程语言的编译器的分配算法。假如分别使用 C++类和 Java 类去描述一个客观事物，当类中包含各种类型的字段，从字节到双精度类型全有，并且字段被乱序声明时，则 Java 类所占用的堆内存空间经过 JVM 的优化之后，甚至会比 C++类所占用的堆内存空间要少。

Java 类在堆内存中的内存空间，主要由 Java 类非静态字段占据。HotSpot 解析 Java 类非静态字段和分配堆内存空间的主要逻辑总结为如下几步：

（1）解析常量池，统计 Java 类中非静态字段的总数量，按照 5 大类型（oops、longs/doubles、ints、 shorts/chars、 bytes）分别统计。

（2）计算 Java 类字段的起始偏移量，起始偏移位置从父类继承的字段域的末尾开始。

（3）按照分配策略，计算 5 大类型中的每一个类型的起始偏移量。

（4）以 5 大类型各个类型的起始偏移量为基准，计算每一个大类型下各个具体字段的偏移量。

（5）计算 Java 类在堆内存中所需要的内存空间。

经过上面 5 步，HotSpot 便能确定一个 Java 类所需要的堆内存空间。当全部解析完 Java 类之后，Java 类的全部字段信息及其偏移量将会保存到 HotSpot 所构建出来的 instanceKlass 中，至此，一个 Java 类的字段结构信息便全部解析完成。当 Java 程序中使用 new 关键字创建 Java 类的实例对象时，HotSpot 便会从 instanceKlass 中读取 Java 类所需要的堆内存大小并分配对应的内存空间。