第 5 章

常量池解析

本章摘要

◎ Java 字节码常量池的内存分配链路
◎ oop-klass 模型
◎ 常量池的解析原理

前文讲述了 JVM 执行引擎的核心机制，以及 Java class 字节码文件的格式。从本章开始，继续剖析 JVM 内部的源码。JVM 要完成 Java 逻辑的执行，必须能够“读懂”Java 字节码文件。而 Java 字节码文件从总体上而言，其实主要包含三部分：常量池、字段信息和方法信息。其中常量池存储了字段和方法的相关符号信息，因此对常量池的解读便成为解读 Java 字节码文件的基础。本章主要分析 JVM 解析 Java 字节码文件中常量池的逻辑。本章内容难度属于中等，只要你稍微具备一点 C/C++基础便能读懂，当然，即使完全不具备 C/C++基础，读懂本章也不是难事，大家都是写程序的，而写程序的小伙伴们，智商都是蛮高的^_^。

常量池是 Java 字节码文件的核心，因此也是 JVM 解析字节码的重头戏。要注意的是，这里所说的常量池并不等同于 JVM 内存模型中的常量区，这里的常量池仅仅是文件中的一堆字节码而已。但是字节码文件中的常量池与 JVM 内存区的常量区之间却有着千丝万缕的联系，因为 JVM 最终会将字节码文件中的常量池信息进行解析，并存储到 JVM 内存模型中的常量区。字节码文件中的常量池是 Java 编译器对 Java 源代码进行语法解析后的产物，只是这种初步解析产生的结果比较粗糙，里面包含了各种引用，信息不够直观。而 JVM 根据字节码文件中的常量池信息再进行二次解析，这种解析目标清晰，直奔终点，会还原出所有常量元素的全部信息，让内存与你所编写的 Java 源代码保持一致。

在字节码文件中，用于描述常量池结构的字节码流所在的块区紧跟在魔数和版本号之后，因此 JVM 在解析完魔数与版本号后，接着便开始解析常量池。JVM 解析 Java 类字节码文件的接口是 ClassFileParser::parseClassFile()，该接口内部解析的总体步骤如图 5.1 所示。

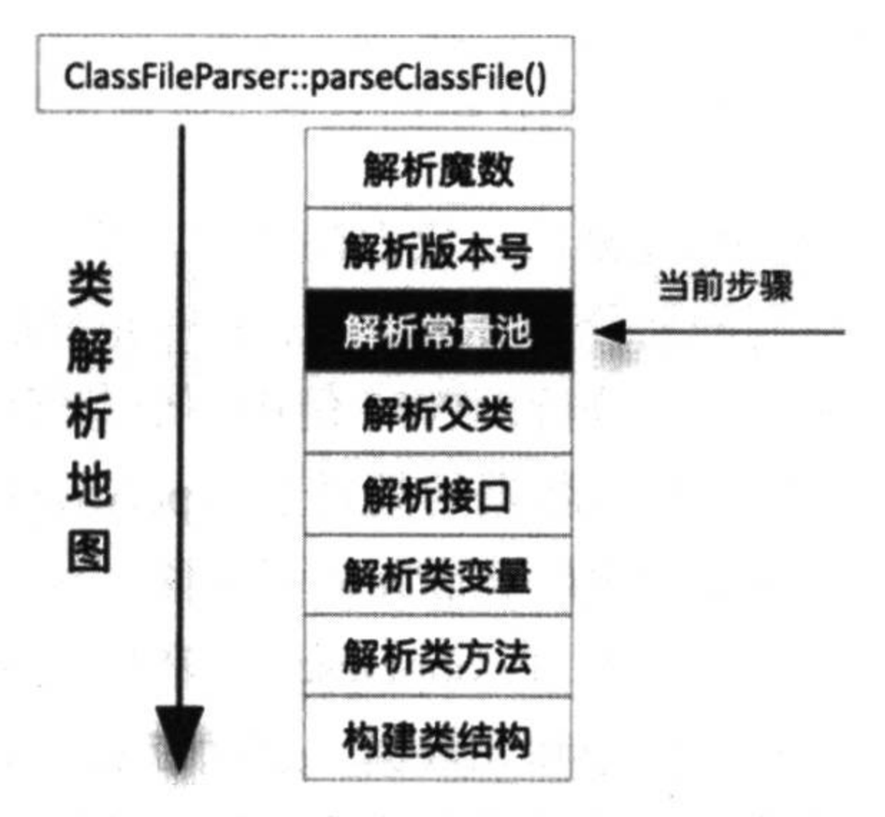

图 5.1　JVM 解析 Java 字节码文件的总体步骤

本章主要讲解 JVM 对 Java 字节码文件中常量池信息的解析。本章以及后续几个章节会详细讲解 JVM 解析常量池、类变量和类方法的过程，为了使思路上保持连贯性，后续章节也会贴出该示意图。

JVM 中对字节码常量池信息的解析的主要链路如图 5.2 所示。

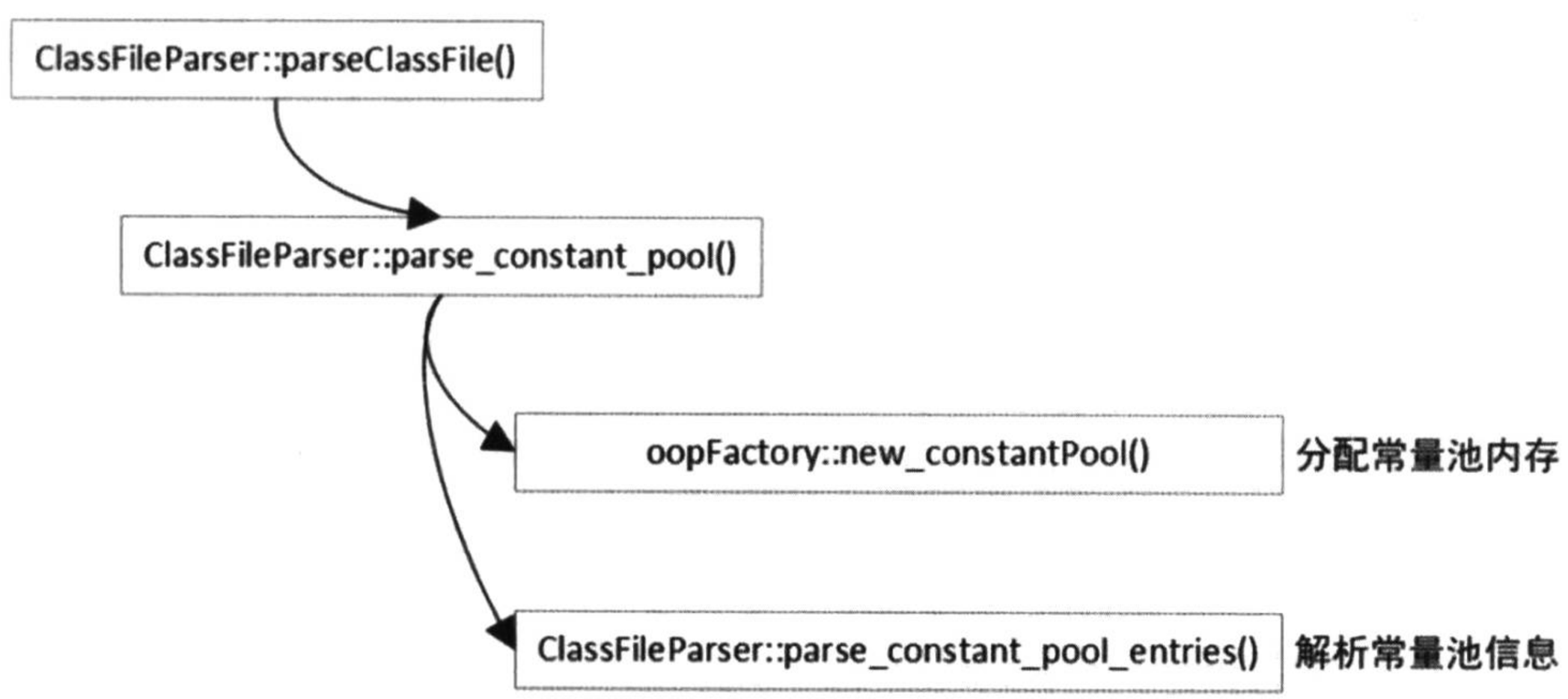

图 5.2　解析常量池的主要链路

由图 5.2 可以看出，JVM 对常量池的解析主要分为两步：第一步是为常量池分配内存，第二步是解析常量池信息。下面就从这两个部分分析常量池解析的过程。

5.1 常量池内存分配

JVM 欲解析常量池信息，就必须先划出一块内存空间，将常量池的结构信息加载进来，然后才能进行进一步分析。内存空间的划分主要考虑两点：一是分配多大的内存；二是分配在哪里。关于常量池的数据结构在前文已有分析，JVM 使用一个专门的 C++类 constantPoolOop 来保存常量池的信息，而该类里面实际保存数据的是属于 typeArrayOop 类型的_tags 实例对象。typeArrayOop 类继承于 oopDesc 顶级结构，oopDesc 里面除了标记和元数据，就啥都没有了。而令人绝望的是，typeArrayOop 并没有在 oopDesc 的基础上多增加一些字段，换言之，typeArrayOop 这种类型也是仅仅只有标记和元数据。这就带来一个问题，typeArrayOop 究竟如何对复杂的常量池信息进行描述的呢？其所带来的关联问题就是，JVM 怎样根据 typeArrayOop 去分配到合适大小的内存呢？

对于天生就适合领域建模的 Java 语言而言，如果要描述某类对象，一般会为其建立对应的数据结构模型。拿大伙儿喜闻乐见的学生为例，如果使用 Java 来建模，则至少会是这个样子：

清单：Student.Java
作用：演示 Java 类对事物的定义

```
// 学生类型
class Student{
    /**
     * 年龄
     */
    private Integer age;

    /**
     * 身高
     */
    private Integer height;

    /**
     * 名字
     */
    private String name;

    /**
     * 年级
     */
```

```
    private Integer grade;
}
```

可以看出，Java 语言所定义的这种类型能够对“学生”这种“事物”进行精确描述，同时，数据结构一旦定义好，则意味着其所占用的内存空间也已经确定。

可是如果将学生类型替换成常量池，并且使用 typeArrayOop 这种类型来描述，则其所描述的学生类型是这样的：

清单：Student.Java
作用：演示 Java 类对事物的定义

```
class oopDesc {
    private:
        volatile markOop  _mark;//线程锁等标记
        union _metadata {
            wideKlassOop    _klass;
            narrowOop       _compressed_klass;
        } _metadata;//元数据，自引用
}
```

很显然，直接根据 typeArrayOop 这种类型，根本就看不出其所描述的对象究竟包含哪些属性，更无法据此计算出其所描述的事物应该占用多大内存。看来这个类真的如同其名字一样，只描述数组信息。而事实上，常量池也的确类似于数组，不同的 Java 类所生成的常量池信息是不同的，因此也无法使用一种预先定义好的数据模型去描述。但是，常量池又不是严格意义上的数组，因为其每种元素成员的类型并不相同。那么 typeArrayOop 如何描述常量池所需内存空间和常量池结构信息呢？

饭是一口一口吃的，问题也得一个一个解决。我们先将目光回归到本节主题：常量池内存分配。

5.1.1　常量池内存分配总体链路

在 ClassFileParser::parse_constant_pool()函数中，通过下面这行代码实现常量池内存分配：

```
onstantPoolOop constant_pool =
                    oopFactory::new_constantPool(length,
                                                 oopDesc::IsSafeConc,
                                                    CHECK_(nullHandle));
```

oopFactory::new_constantPool() 的第一个入参是 length，其在 ClassFileParser::parse_constant_pool()函数中实现了对常量池大小的解析。length 值代表当前字节码文件的常量池中一共包含多少个常量池元素，该数值由 Java 编译器在编译期间通过分析计算得出，最终将其保存

在字节码文件中，所以 JVM 在解析常量池时可以直接拿来使用，这里的 length 便是 JVM 直接从字节码文件中读取出来的。在前面章节对 Java 字节码文件结构的分析中，讲到了常量池的大小的计算，有不清楚的小伙伴可以先温习前面的章节。不过要注意的是，length 并不表示常量池需要占用多大的内存空间，而是代表一共有多少个常量池元素。有了这个值，JVM 就能对常量池中的常量池元素进行逐个遍历处理。

oopFactory::new_constantPool 链路比较长，下面先给出总体调用路径（如图 5.3 所示）。

由图 5.3 可以看出，常量池内存分配的链路比较长，下面我们先对这条链路所涉及的类型进行一个简单说明：

◎ **oopFactory**。顾名思义，就是 oop 的工厂类。工厂类设计模式主要负责生产专门的对象，与具体的编程语言无关。前文讲过，Java 语言是一门面向对象的语言，所有一切皆对象，这种面向对象的特性不仅在语法层面贯彻得很彻底，而且在 JVM 内部实现层面也得到彻底的实现。在 JVM 内部，常量池、字段、符号、方法等一切都被对象包装起来，所有这一切对象在内存中都通过 oop 这种指针进行跟踪（指向），JVM 根据 oop 所指向的实际内存位置便可获取到对象的具体信息。而在 JVM 内部，所有这些对象的内存分配、对象创建与初始化（清零）工作都通过 oopFactory 这个入口得以统一实现。本节所讲解的常量池也不例外。

◎ **constantPoolKlass**。常量池类型对象。对于每一种对象，JVM 内部都通过 oop 指针指向到某个内存位置，这个内存位置往往便是与该 oop 相对应的 klass 类型。klass 用于描述 JVM 内部一个具体对象的结构信息，例如，一个 Java 类中包含哪些字段、哪些方法、哪些常量。同理，常量池在 JVM 内部也被表示为一种对象，虽然开发者并不能在源程序中通过 Java 程序来表示它。不同的 Java 类被编译后所生成的字节码文件中的常量池大小、元素顺序和结构等都不相同，因此 JVM 内部必须要预留一段内存区块来描述常量池的结构信息，这便是 constantPoolKlass 的意义所在。后文还会对其进行专门讲解，尤其是最重要的与源码层面的 Java 类相关联的相关 klass 对象。

◎ **collectedHeap**。闻声辨人，顾名思义，这表示 JVM 内部的堆内存区，可被垃圾收集器回收并反复利用。其代表 JVM 内部广义的堆内存区域，在 JDK 6 时代，这个内存区域会包含用于分配 Java 类对象实例的堆内存区、常量池区和 perm 区（方法区）。在 JVM 内部，除了堆栈变量之外的一切内存分配，都需要经过本区域。如果 JVM 内部的一个实例对象不在这一区域申请内存空间，那么只能跑到 JVM 堆外内存去申请了。

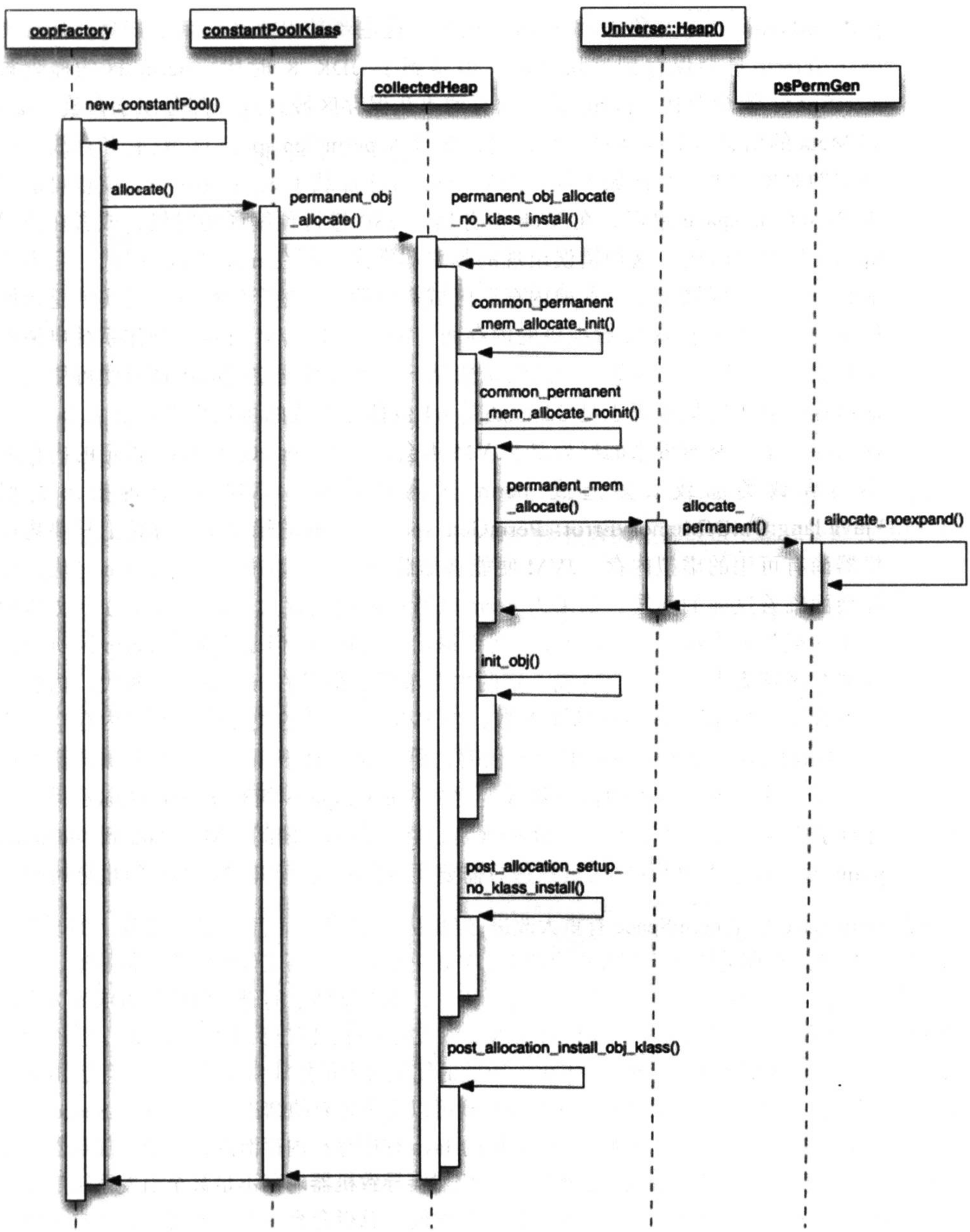

图 5.3　常量池内存分配调用链路

◎ **psPermGen**。这个对象表示 perm 区内存，这主要是 JDK 6 时代的产物，Java 类的字节码信息会保存到这块区域。但是到了 JDK 8 时代，perm 区的概念被 metaSpace 概念取代。perm 是指内存的永久保存区域，这一部分用于存放 Class 和 Meta 的信息，Class 在被加载的时候被放入 permGen space 区域，它和存放 Java 类实例对象的堆内存区域不同，如果 Java 程序加载了太多 Java 类，就很可能出现 PermGen space 错误，在 Web 服务器对 JSP 进行预编译的时候，或者在使用 Spring 框架的时候，这种错误出现的频率非常高。而在 metaSpace 时代，官方的解释是：这块区域属于“本地内存”区域。所谓“本地”的概念，其实就是指操作系统，是相对于 JVM 虚拟机的内存而言的。由于 JVM 直接向操作系统申请内存存储 Java 类的元信息，因此默认情况下，类元数据空间的申请只受可用的本地内存限制（容量取决于是 32 位还是 64 位操作系统的可用虚拟内存大小）。这种内存分配策略所带来的好处是，JVM 不会由于为 permGen space 指定的内存太小而导致类加载太多造成 perm 区内存耗尽，JVM 不会再报出类似“java.lang.OutOfMemoryError: PermGen space”这样的错误了，理论上只要物理机器尚有可用的虚拟内存，JVM 便能够加载新的类元数据。而在 perm 时代，往往会面临着这样的问题：如果为 perm 区所指定的内存太小，则很容易导致 JVM 由于加载类太多耗尽 perm 内存空间而退出正常运行。但是如果为 perm 区所指定的内存区域太大，则又会造成一定的内存浪费。但是在 metaSpace 时代，仍然应该设置 MaxMetaspaceSize 这个参数，当 JVM 所加载的类元信息所占内存空间达到 MaxMetaspaceSize 参数的设定值时，将会触发对于僵死的类及类加载器的垃圾回收。除了内存分配的位置被改变之外，metaSpace 相对于 permSpace 时代，在性能上也做了一点调整，具体表现是，在 Full GC 期间，Metadata 到 Metadata pointers 之间不需要扫描了，虽然这只需要几纳秒时间，但几纳秒的时间也是时间。

虽然 metaSpace 相比 permSpace 有重大改进，但是从 JVM 角度而言，并没有本质上的改变，JVM 的类结构在运行期的描述机制并没有改变，JVM 仍然从字节码文件中解析出常量池、字段、方法等类元信息，然后保存到内存中某个位置。因此，本书整体上的研究仍然以 JDK 6 为主，对 JDK8 不做过多深究。而事实上，如果对 JDK 6 深入研究过，回过头来再看 JDK8，已经没有什么技术上的障碍或者挑战了。况且，JDK8 所做的优化努力虽然非常值得肯定，但是 JDK 6 并不会因此就可以被贬抑得一无是处，并因此得到可以完全舍弃的结论。虽然 permSpace 有其缺点，但是 metaSpace 也并非只有优点。如果你的 Java 程序存在内存泄露，导致类加载器不停地重复加载类，不停地扩展 metaspace 的空间，仍然会导致机器内存不足甚至引发系统崩溃。这些都是自动内存管理的通病，在自动化和稳定性之间，总得舍弃一个，或者两者各自舍弃一点以达到某种均衡。

虽然常量池内存分配的链路很长，但是从宏观层面来看，常量池内存分配大体上可以分为下面 3 个步骤。

1）在堆区分配内存空间

这一步最终在 psOldGen.hpp 中通过 object_space()->allocate(word_size)实现。具体实现的思路下文会进行详细描述。

2）初始化对象

主要通过在 collectedHeap.inline.hpp 中调用 init_obj()进行对象初始化。所谓的对象初始化，其实仅仅是清零。

3）初始化 oop

在 collectedHeap.inline.hpp 中调用 post_allocation_install_obj_klass()完成 oop 初始化并赋值。JVM 内部通过 oop-klass 来描述一个 Java 类。一个 Java 类的实例数据会被存放在堆中，而为了支持运行期反射、虚函数分发等高级操作，Java 类实例指针 oop 会保存一个指针，用于指向 Java 类的类描述对象，类描述对象中保存一个 Java 类中所包含的全部成员变量和全部方法信息。本步骤便是为这一目标而设计的。

这 3 步所对应的链路部分如图 5.4 所示，分别对应从上至下的 3 个被框定的流程。

执行完上面这 3 步，一个常量池对象便完成了内存分配和部分初始化，但是此时常量池对象是空对象，其内存区还没有填充具体的值，parseClassFile()函数下一步会做这个事情。

这里额外插一句，Java 字节码中的一切皆对象，无论是常量池、成员变量、方法还是数组等，在 JVM 虚拟机内部都被识别为“对象”，而为类对象分配内存空间的操作都封装在 oopFactory 中。oopFactory 为各种 JVM 内建对象分配内存空间并初始化类型实例的机制还是一样的，都是先获取对应的 klass 类描述对象，然后为 oop 分配内存空间。JVM 为一个 Java 类分配内存空间的机制，与 JVM 为一个常量池分配内存空间的机制在本质上是一样的，因此在这里，我们将一起对 constantPool 的内存分配机制进行认真和细致的研究，这一块研究透彻了，到了后面看到 JVM 为其他各种对象分配内存时就不会觉得难以理解了，主要的原理都是类似的，只有部分细节不同。同时，将这些对象分配机制研究透了之后，可以反过来促进对 JVM 的执行引擎的研究，其实执行引擎的很多伏笔就埋在了 JVM 类对象分配流程之中。

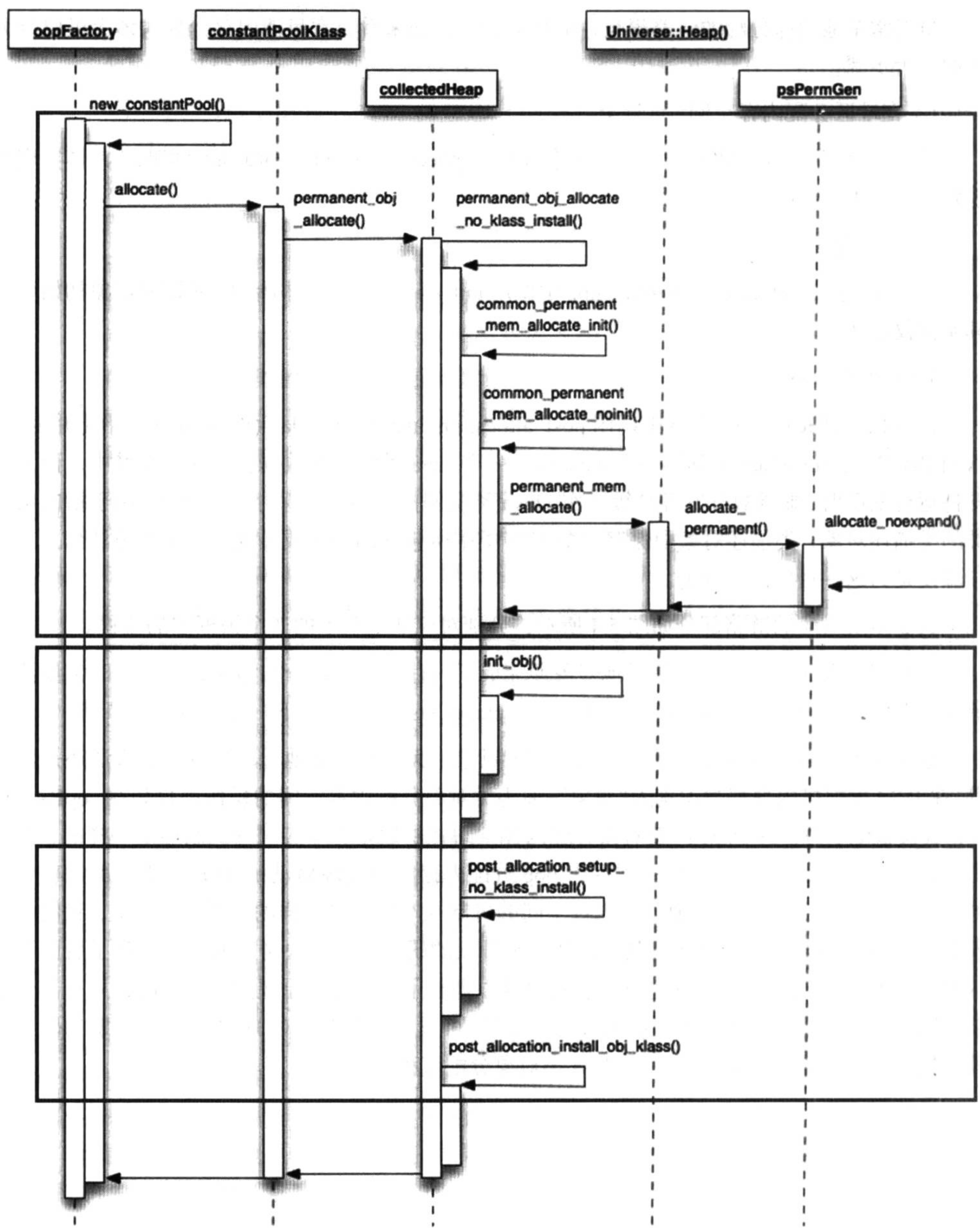

图 5.4　常量池内存分配三大步

下面分别对 constantPool 对象初始化的各主要步骤进行详细讲解。

5.1.2　内存分配

从 oopFactory::new_constantPool()调用开始一直到 mutableSpace::allocate()，这个过程可以认为是第一个阶段，即为 constantPool 申请内存。

前面讲过，内存申请主要关注两点：一是应该申请多大的内存；二是在哪里申请内存。在图 5.5 所示的链路中，从一开始便知道要申请多大的内存，内存的大小就是 length 变量所代表的值。因此，常量池有多少个常量池元素，最终便会分配多大的内存，至于为何是这样，下面会详细讲解。下面先分析 JVM 为 constantPool 申请内存的机制。

这条链路所涉及的接口调用及其所在的文件位置如图 5.5 所示。

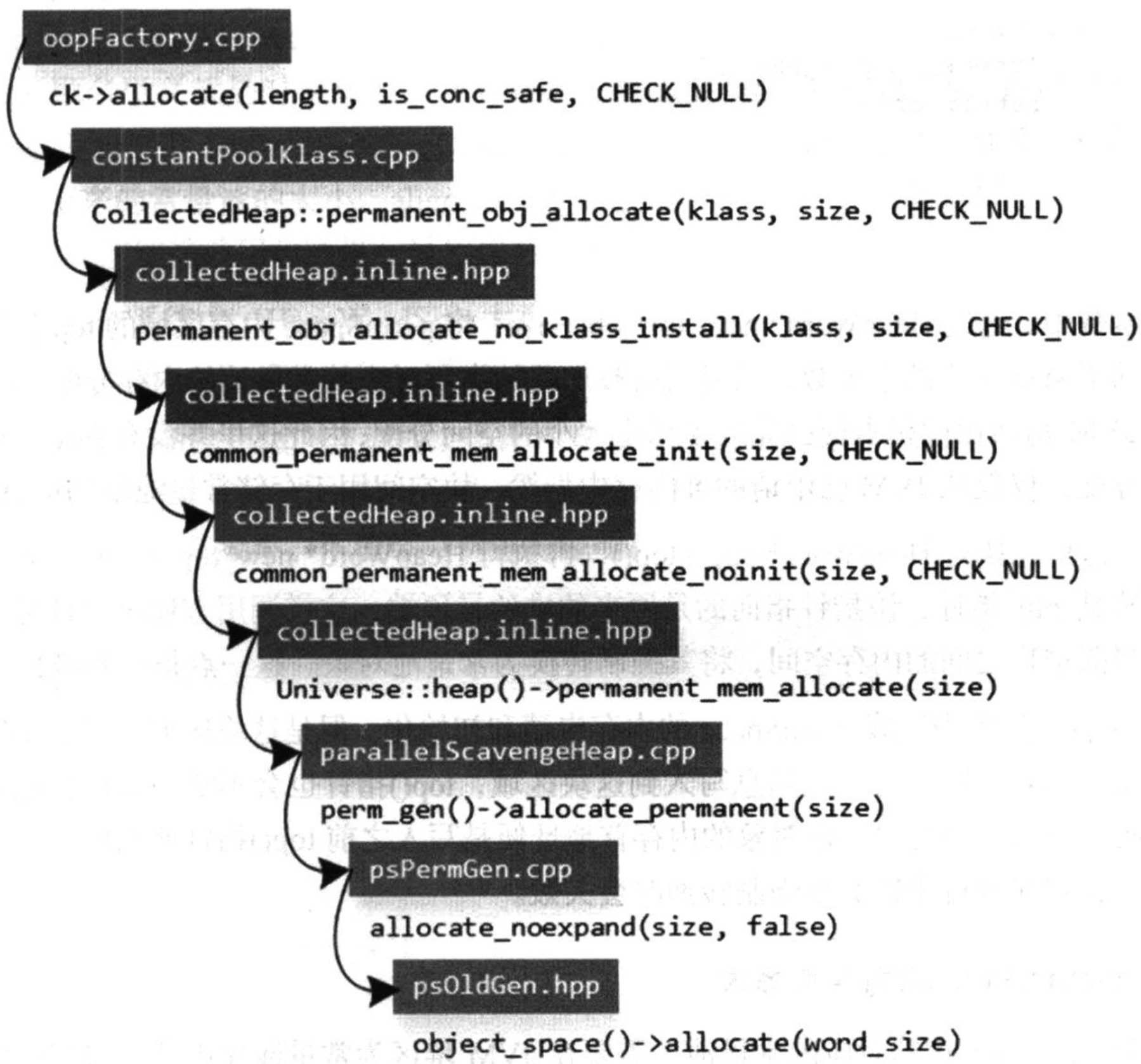

图 5.5　JVM 为常量池对象 constantPool 分配内存的链路

内存申请最终通过 object_space()->allocate()实现，其上游一系列链路主要实现方法的逐级调用，为最后调用 allocate()作铺垫。allocate()函数定义在 mutableSpace.cpp 中，定义如下：

清单：src/share/vm/gc_implemention/shared/mutableSpace.cpp
作用：在 JVM 堆区分配内存

```
// This version requires locking. */
HeapWord* MutableSpace::allocate(size_t size) {
    assert(Heap_lock->owned_by_self() ||
        (SafepointSynchronize::is_at_safepoint() &&
        Thread::current()->is_VM_thread()),
        "not locked");
    HeapWord* obj = top();
    if (pointer_delta(end(), obj) >= size) {
        HeapWord* new_top = obj + size;
        set_top(new_top);
        assert(is_object_aligned((intptr_t)obj) && is_object_aligned((intp
tr_t)new_top),
        "checking alignment");
        return obj;
    } else {
        return NULL;
    }
}
```

在该函数中，通过 HeapWord* new_top=obj+size，将 permSpace 内存区域的 top 指针往高地址方向移动了 size 大小的字节数，完成当前被加载的类所对应的常量池的内存分配。由于 JVM 的堆区在 JVM 初始化过程中便已完成了指定大小的空间分配，因此这里并没有真正向操作系统申请内存分配，仅仅从 JVM 已申请的堆内存中划拨一块空间用于存储常量池结构信息。

注意，这里先执行 HeapWord* obj = top()，再执行 HeapWord* new_top = obj + size，而最终返回的仍然是 obj 指针，该指针指向的是原来的堆的最顶端，这样调用方通过指针可以还原从原堆顶到当前堆顶之间的内存空间，将其强制转换为常量池对象。这一点很重要哦！

JVM 在启动过程中完成 permSpace 的内存申请和初始化，但是这块区域一开始是空的，没有任何数据。随着不断有常量池信息写入到这块区域，top()指针也会不断往高内存地址方向移动，而每次新写入一个对象，该对象的内存首地址便是写入之前 top()指针所指的位置，这一点需要留心，否则在进行指针类型强制转换时会失败。

1．constantPool 的内存有多大

为一个 Java 类创建其对应的常量池，需要在 JVM 堆区为常量池先申请一块连续的内存空间。所申请的内存空间的大小取决于一个 Java 类在编译时所确定的常量池大小，更直白地说，

就是取决于你所定义的类的大小。在上面所描绘的常量池初始化调用链路中，可以看到 size 这个变量一直从链路前端被透传到最末端，这个 size 变量便是 JVM 为当前常量池所计算出来的内存大小。

在常量池初始化链路中会调用 constantPoolKlass::allocate()方法，该方法会调用 constantPoolOopDesc::object_size(length)方法来获取常量池大小，该方法的原型如下：

清单：src/share/vm/oops/constantPoolOop.hpp
作用：计算 Java 类所对应的常量池的内存大小

```
static int object_size(int length)   {
    return align_object_size(header_size() + length);
}
static int header_size() {
    return sizeof(constantPoolOopDesc)/HeapWordSize;
}
```

object_size()的实现逻辑比较简单，仅仅是将 header_size 与 length 相加后再进行内存对齐。header_size 顾名思义就是对象头大小，对象头的概念会在后面进行描述，在这里只需要知道 header_size()返回的是 constantPoolOopDesc 类型的大小，而 length 则为 Java 类在编译期间由编译器所计算出来的常量池的大小。align_object_size()函数的作用是实现内存对齐，这样便于在 GC 进行工作时能够高效回收垃圾，虽然这会造成一定的空间浪费。

在 32 位操作系统上，HeapWordSize 大小为 4，为一个指针型变量的长度。因此，在 32 位平台上，sizeof(constantPoolOoopDesc)返回 40。

关于 sizeof()函数这里做一点说明。当计算 C++类时，该函数返回的是其所有变量的大小加上虚函数指针的大小。若在类中定义了普通的函数，无论是公有还是私有，也无论是静态还是非静态，都不会计算其大小。下面举一例进行说明：

清单：测试
作用：演示 sizeof()函数的功能

```
#include<stdio.h>

class A{
private:
    char *a;
public:
    int getA(){ return 1; }
};

int main(){
    printf( "sizeof(A)=%d\n" , sizeof(A));
```

```
    return 0;
}
```

在 32 位平台上运行该程序，最终打印出来的值是 4，因为 A 类中只包含一个指针型变量，而 32 位平台上一个指针的数据宽度是 4，所以最终打印出来 4。

根据这个演示程序，可以推算 sizeof(constantPoolOoopDesc) 返回值的大小。constantPoolOoopDesc 本身包含 8 个字段，如下：

```
typeArrayOop          _tags;
constantPoolCacheOop _cache;
klassOop              _pool_holder;
typeArrayOop          _operands;
int                   _flags;
int                   _length;
volatile bool         _is_conc_safe;
int                   _orig_length;
```

由于 constantPoolOoopDesc 继承自 oopDesc 父类，因此 constantPoolOoopDesc 类还会包含来自父类的 2 个成员变量：

```
volatile markOop  _mark;
union  metadata {
    wideKlassOop    _klass;
    narrowOop       _compressed_klass;
} _metadata;
```

注意：markOop 的类型原型是 markOopDesc*指针类型，因此_mark 的类型是指针，在 32 位平台上占 4 字节的内存。_metadata 是联合体，联合体内部是指针类型，因此也占 4 字节空间。

虽然 oopDesc 类内部包含 static BarrierSet* bs 这样一个变量，但是这是静态类型的变量，在 JVM 启动之初其会被操作系统直接分配到 JVM 程序的数据段内存区，在 JVM 为 Java 程序分配内存时，不会为_bs 这样的全局变量在 JVM 堆内存或者 permSpace 内存区另外分配空间。

如此看来，constantPoolOoopDesc 最终实际包含 10 个字段，因此在 32 位平台上，sizeof(constantPoolOoopDesc)将返回 40。各位小伙伴大可以使用 GDB 在 32 位平台上断点调试，在断点时直接打印表达式 sizeof(constantPoolOoopDesc)的值进行验证。

搞清楚了 sizeof(constantPoolOopDesc)，还需要研究下 HeapWordSize，因为通过 header_size() 计算 constantPoolOopDesc 类大小时使用到了 HeapWordSize（header_size()的计算公式是 sizeof(constantPoolOopDesc)/HeapWordSize）。

HeapWordSize 定义如下：

清单：src/share/vm/utilities/globalDefinitions.hpp
作用：计算 HeapWordSize 大小

```
const int HeapWordSize          = sizeof(HeapWord);

class HeapWord {
     friend class VMStructs;
private:
     char* i;
#ifndef PRODUCT
public:
     char* value() { return i; }
#endif
};
```

可以看到，HeapWordSize 是 HeapWord 类的大小，而该类只包含一个 char*指针型变量，因此在 32 位平台上，sizeof(HeapWordSize)返回 4，即其大小是 4 字节大小。如果在 64 位平台上，sizeof(HeapWordSize)返回 8，即其大小是 8 字节大小。所以，HeapWord 的大小其实就是一个指针的大小，不同平台的指针所占的内存大小是不同的。同理，header_size()函数返回的也是当前平台上的指针大小。

至此，header_size()函数的作用已经十分清楚了，sizeof(constantPoolOopDesc)返回 constantPoolOopDesc 类型本身所占内存的总字节数，而 HeapWordSize 则返回对应平台上的一个指针宽度。在 32 位平台上，一个指针宽度为 4 字节，即双字（双字占 4 字节），因此 header_size()计算出的结果表示 constantPoolOopDesc 这个类型实例在内存中所占用的双字数。

理解了上面所讲的 header_size()函数的含义之后，再回过头来看 object_size(int length) { return align_object_size(header_size() + length); }这个函数，便很容易理解该函数的思路了，其实 object_size()函数最终返回的值代表 constantPoolOopDesc 类型本身的大小加上常量池的大小 length，constantPoolKlass::allocate()便根据这个结果，从 JVM 的 perm 区申请所需的内存空间大小。

最终，constantPoolKlass::allocate()从 JVM 堆内存中所申请的内存空间大小包含如下所示的两部分：

constantPoolOopDesc 大小	Java 类常量池元素数量

代码总是枯燥的，因此我们还是以前文所举的 Iphone6s.Java 为例。该类定义如下：

清单：Iphone6s.java
作用：Iphone6s 类型结构

```
public class Iphone6s {
     int length = 138;                    //长度（毫米）
     int width = 67;                      //宽度（毫米）
```

```
        int height = 7;                    //高度（毫米）
        int weight = 143;                  //重量（克）
        int ram = 2;                       //ram 容量（G）
        int rom = 16;                      //rom 容量（G）
        int pixel = 1200;                  //摄像头像素（万）
    }
```

使用十六进制编辑器打开编译后生成的字节码文件，得到常量池的长度 length，如图 5.6 所示。

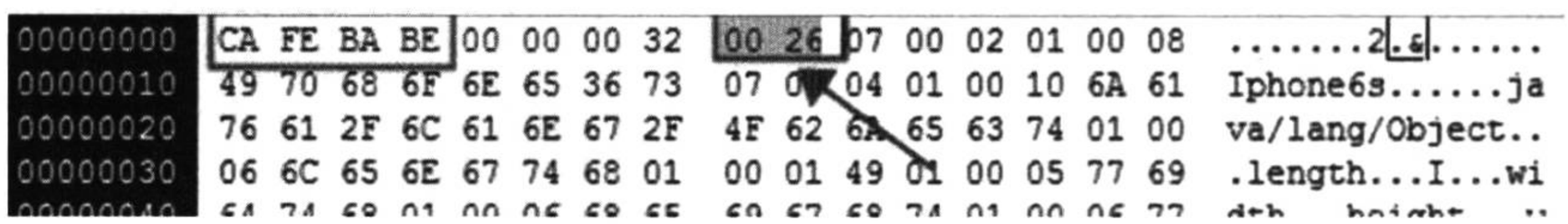

图 5.6　示例 Java 程序字节码文件中的常量池大小

图 5.6 中第一个方框内容是魔数，值为 0xCAFEBABE，因此图中箭头所指的字节便代表常量池的元素数量。图中所示的常量池的元素数量为 0x26，换算成十进制为 38。由此可知，Iphone6s.class 字节码文件中的常量池一共包含 38 个元素。使用 javap -verbose 命令进行分析的结果如下：

```
>javap -verbose Iphone6s
Compiled from "Iphone6s.java"
public class Iphone6s extends java.lang.Object
  SourceFile: "Iphone6s.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = class        #2;    //  Iphone6s
const #2 = Asciz        Iphone6s;
const #3 = class        #4;    //  java/lang/Object
const #4 = Asciz        java/lang/Object;
const #5 = Asciz        length;
const #6 = Asciz        I;
//......
const #34 = Asciz       this;
const #35 = Asciz       LIphone6s;;
const #36 = Asciz       SourceFile;
const #37 = Asciz       Iphone6s.java;
```

javap 命令所分析出的结果一共有 37 个常量池元素，之所以比字节码文件中的少一个，是因为 JVM 会保留第 0 号常量池位置，因此仅从第 1 个开始计算。如此便能保持一致。

如果 JVM 当前正在解析 Iphone6s.class 字节码文件的常量池信息，那么按照上面的分析，最终 JVM 会从 permSpace 中划分出(40 + 38) * 4 字节的内存大小（32 位平台）。

现在问题就来了，为什么 JVM 为常量池对象分配这么大的内存呢?

要解答这个疑问，需要知道 JVM 是如何解析字节码的常量池信息的，后文会专门讲解细节，因此这里先将问题放下，先讨论一个重要的问题：常量池的内存布局。

2. 内存空间布局

在为 constantPoolOop 常量池对象分配内存时，需要分析 JVM 为常量池所申请的内存空间布局的模型。刚才提到，JVM 为 constantPoolOop 实例对象所分配的内存空间大小是（headSize + length）个指针宽度（或者双字宽度）。JVM 为常量池对象申请的内存位于 perm 区，perm 区本身是一片连续的内存区域，而 JVM 为常量池申请内存时也是整片区域连续划分，因此每一个 constantPoolOop 对象实例在 perm 区中都是连续分布的，不会存在碎片化。

最终申请好的内存空间布局如图 5.7 所示。

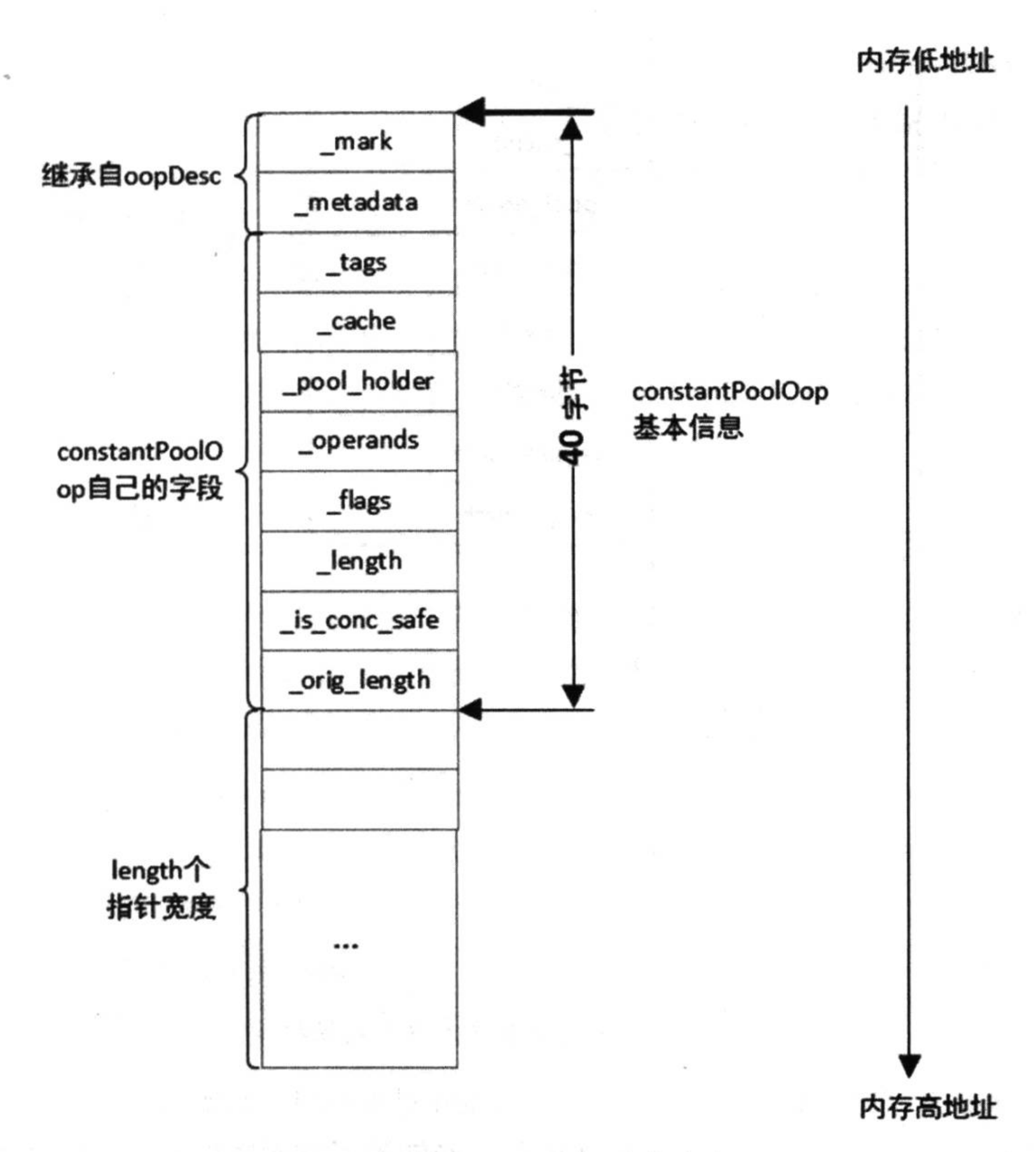

图 5.7　constantPoolOop 的内存布局

注：该图假设 JVM 运行于 Linux 32 位平台上，因此指针宽度为 4 字节，并且常量池分配方向为从低地址内存往高地址内存方向。

图 5.7 看起来虽然简单，但是在 JVM 内部，几乎所有的对象都是这种布局。总体而言，JVM 内部为对象分配内存时，会先分配对象头，然后分配对象的实例数据，不管字段对象还是方法对象，抑或是数组，莫不如是。再强调一遍，JVM 内部为对象实例分配内存空间的模型都是“对象头+实例数据”的结构哟！先用这个模型对图 5.7 所示的 constantPoolOop 的内存布局进行简化，得到如图 5.8 所示的布局。

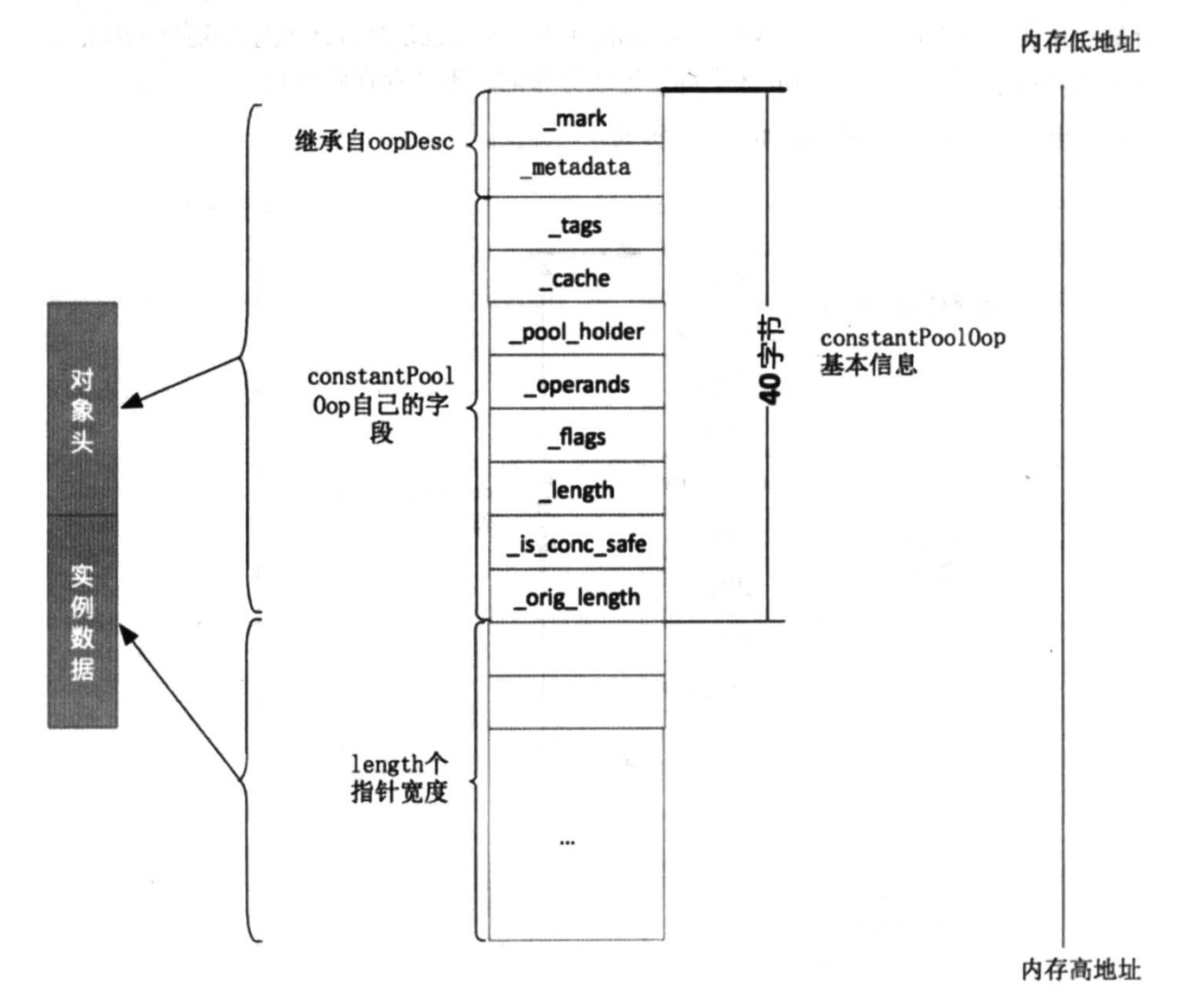

图 5.8　简化的常量池内存布局结构

对于常量池而言，其对象头就是 constantPoolOop 对象本身，而实例数据究竟放啥暂时还不知道（原谅我无法预先进行剧透），不过聪明的你已经知道了实例数据所占空间的大小既然等于

Java 字节码文件中所有常量池元素所占空间的大小，那么可以大胆猜测，实例数据应该就是保存 Java 字节码文件中的常量池元素的某些特殊信息，但是常量池信息量往往比其数量要大得多，因此紧跟在 constantPoolOop 之后的这一块实例数据区不可能用于存储常量池的全部信息。事实上，这一块的存储机制是比较复杂的，后续的 2 个小节将会专门讲述这个问题。目前只需要知道常量池的内存布局总体上也是由对象头和实例数据这两块组成的。

有句话是怎么说的？知其然，更要知其所以然。虽然如此浓墨重彩地深入详细地分析区区一个常量池的内存模型是很必要的，但是本书并不打算仅仅讲解 what 和 how，后面会分析为何 JVM 要将常量池分配在 perm 区（JDK 1.6），同时需要考虑一个最核心的问题：JVM 为何要设计常量池这样一种机制。这两个问题是 why，研究 what 和 how 通常仍然属于技术的范畴，而研究 why 则意味着超脱技术而上升到哲学的高度，虽然这里所谓的哲学也许被称为“技术哲学”要显得更加合理。作为技术爱好者，知其然固然好，能够做到知其所以然更是难能可贵，但是一旦做到了，所得到的回报往往会超出想象。以 JVM 为例，虽然绝大多数开发者也许一辈子都没有开发虚拟机的机会，从这个角度而言，如果单纯为了研究 JVM 而研究 JVM，则一定是在浪费生命和时间。但是如果能够研究 JVM 种种技术设计的背后原因，了解 JVM 的各个模块之所以这样设计的道理，这样就能超脱 JVM 本身，不被局限于这一个领域之内，而能达到“天高任鸟飞，海阔凭鱼跃”之境界。在未来，计算机技术发展的方向一定比现如今还要更为宽广，机器人领域、量子通信与计算、虚拟现实、神经网络、3D 打印等各方面都需要较为扎实的基本功，一座座未来技术大厦的建设都需要多样化的技术支撑，为了迎接未来有可能出现的技术潮流，现在深入研究一些核心的技术，一点都不算浪费。各位有志向的小伙伴们共勉！

5.1.3　初始化内存

JVM 为 Java 类所对应的常量池分配好内存空间后，接着需要执行这段内存空间的初始化。所谓的初始化其实就是清零操作。由于在申请内存空间时执行了内存对齐，同时由于 JVM 堆区会反复加载新类和擦除旧类，因此如果不执行清零，则会影响后续对 Java 类的解析。

在 CollectedHeap::common_permanent_mem_allocate_init() 中调用 init_obj(obj, size)，在 init_obj(obj, size)中调用 Copy::fill_to_aligned_words(obj + hs, size - hs)，在 x86 Linux 平台上，该函数最终调用 pd_fill_to_words()函数，此函数声明如下：

清单：/src/cpu/x86/vm/copy.x86.hpp
作用：JVM 内部对象清零

```
static void pd_fill_to_words(HeapWord* tohw, size_t count, juint value) {
#ifdef AMD64
    julong* to = (julong*) tohw;
```

```
        julong  v  = ((julong) value << 32) | value;
        while (count-- > 0) {
          *to++ = v;
        }
    #else
        juint* to = (juint*)tohw;
        count *= HeapWordSize / BytesPerInt;
        while (count-- > 0) {
          *to++ = value;
        }
    #endif // AMD64
    }
```

在 pd_fill_to_words()函数中，会将指定内存区的内存数据全部清空为零值，也即该函数的第 3 个入参 value 值。由于在 CollectedHeap:: init_obj()中调用了 Copy::fill_to_aligned_words(obj + hs, size – hs 函数，)而 Copy::fill_to_aligned_words()函数实际有 3 个入参，声明如下：

清单：/src/share/vm/utilities/copy.hpp

作用：清零

```
static void fill_to_aligned_words(HeapWord* to, size_t count, juint value =
 0) {
    assert_params_aligned(to);
    pd_fill_to_aligned_words(to, count, value);
}
```

注意，该函数第 3 个参数默认为 0，因此最终在执行 pd_fill_to_words()函数时，指定的内存区会全部被抹为零值。

5.2 oop-klass 模型

现在，JVM 已经为 constantPoolOop 分配好内存并进行清零，接下来会进行非常重要的一步：填充内存。JVM 为 constantPoolOop 申请内存，随后会解析 Java 字节码中的常量池信息，并将解析的中间结果暂存到所申请的内存中去。但是在讲解 constantPoolOop 的解析过程之前，必须要先弄清楚一个概念——oop-klass 一分为二的内存模型。上文讲过，JVM 为常量池所分配的内存的布局包含两部分，分别是对象头和实例数据。当年詹爷使用了一种特殊的模型来表示类型，这种模型就是 oop-klass。因此，不把这种特殊的模型研究清楚，很难理解 JVM 为对象头分配内存的机制，接下来的源码阅读也一定会很吃力。

事实上，即使弄懂了 JVM 的类模型表示机制，这部分源代码阅读起来仍然十分吃力-_-。吃力的一个重要原因是指针以及基于指针的各种数据类型之间的强制转换，可能当年詹爷也是

深受指针之苦，因此才下定决心要创造出一个没有指针的编程世界。为了达到消灭“指针”的目的，JVM 本身的类模型变得格外复杂，这给源码解读带来相当大的困难。并且这种复杂性和基本的类模型从 JVM 发布以来一直没有经过大的变更，即使从 JDK6 到 JDK8，也仅仅是对 JVM 内部的几种具体的类型进行了重组和去繁就简，并没有进行根本上的变革。因此如果对 JDK6 的类模型研究通透，则对 JDK8 的类模型自然会触类旁通，一看就懂。不过最为关键的是，只要 Java 语言本身对外所提供的功能特性不发生巨大变化，则 JVM 内部的类模型也不会发生巨大的质变，这与本书一开始的章节描述 JVM 执行引擎的技术选择一样，都有其技术上的必然性，虽然真正实现上的细节可能会千差万别并会得到不断改进和优化，但是主要的算法与策略不会变化，一切都是由技术本身所决定的。

Java 是面向对象的编程语言，这种面向对象不仅体现在语法层面，也不仅仅体现在外在的 Java 类层面。JVM 在内部使用 C++类去描述 Java 类的面向对象特性，JVM 的奇特之处就在于，连同这些内部描述的类也被设计成面向对象机制，毫不夸张地说，JVM 内部的面向对象机制比外在的 Java 类语法层面的面向对象机制实现得更加彻底和更加纯粹，Java 语言所表现出来的面向对象特性与 JVM 内部的面向对象特性相比，简直有点小儿科。在 JVM 内部，不仅用于描述 Java 类的 C++对象被赋予相比于 C++语言本身所具备的面向对象特性更深一层的对象机制，而且其还用于描述相对于 Java 类外在对象而言属于 JVM 内部的不能由开发者控制的类型。JVM 内部用于描述 Java 字节码文件中的常量池类型不能被 Java 开发者访问和操作，但是即使是这种内部类，虽然本身使用 C++这种面向对象的语言描述，但是仍然被表示成 oop-klass 这种二分模型。关于 JVM 内部的这种一分为二的模型，完全可以专门开辟一个章节来详细阐述，但是如果仅仅阐述理论未免会流于空洞和枯燥，因此便将其安插在本章中，在描述 JVM 内部常量池的 oop-klass 表示机制的过程中，顺便对这种机制进行阐述。

5.2.1　两模型三维度

前文讲过，JVM 内部基于 oop-klass 模型描述一个 Java 类，将一个 Java 类一拆为二分别描述，第一个模型是 oop，第二个模型是 klass。所谓 oop，并不是 object-oriented programming（面向对象编程），而是 ordinary object pointer（普通对象指针），它用来表示对象的实例信息，看起来像个指针，而实际上对象实例数据都藏在指针所指向的内存首地址后面的一片内存区域中。而 klass 则包含元数据和方法信息，用来描述 Java 类或者 JVM 内部自带的 C++类型信息。其实，klass 便是前文一直在讲的数据结构，Java 类的继承信息、成员变量、静态变量、成员方法、构造函数等信息都在 klass 中保存，JVM 据此便可以在运行期反射出 Java 类的全部结构信息。当然，JVM 本身所定义的用于描述 Java 类的 C++类也使用 klass 去描述，这相当于使用另一种面向对象的机制去描述 C++类这种本身便是面向对象的数据。

JVM 使用 oop-klass 这种一分为二的模型描述一个 Java 类，虽然模型只有两种，但是其实从 3 个不同的维度对一个 Java 类进行了描述。侧重于描述 Java 类的实例数据的第一种模型 oop 主要为 Java 类生成一张“实例数据视图”，从数据维度描述一个 Java 类实例对象中各个属性在运行期的值。而第二种模型 klass 则又分别从两个维度去描述一个 Java 类，第一个维度是 Java 类的“元信息视图”，另一个维度则是虚函数列表，或者叫作方法分发规则。元信息视图为 JVM 在运行期呈现 Java 类的“全息”数据结构信息，这是 JVM 在运行期得以动态反射出类信息的基础。

下面的图 5.9 描述了 JVM 内部对 Java 类的“两模型三维度”的映射。

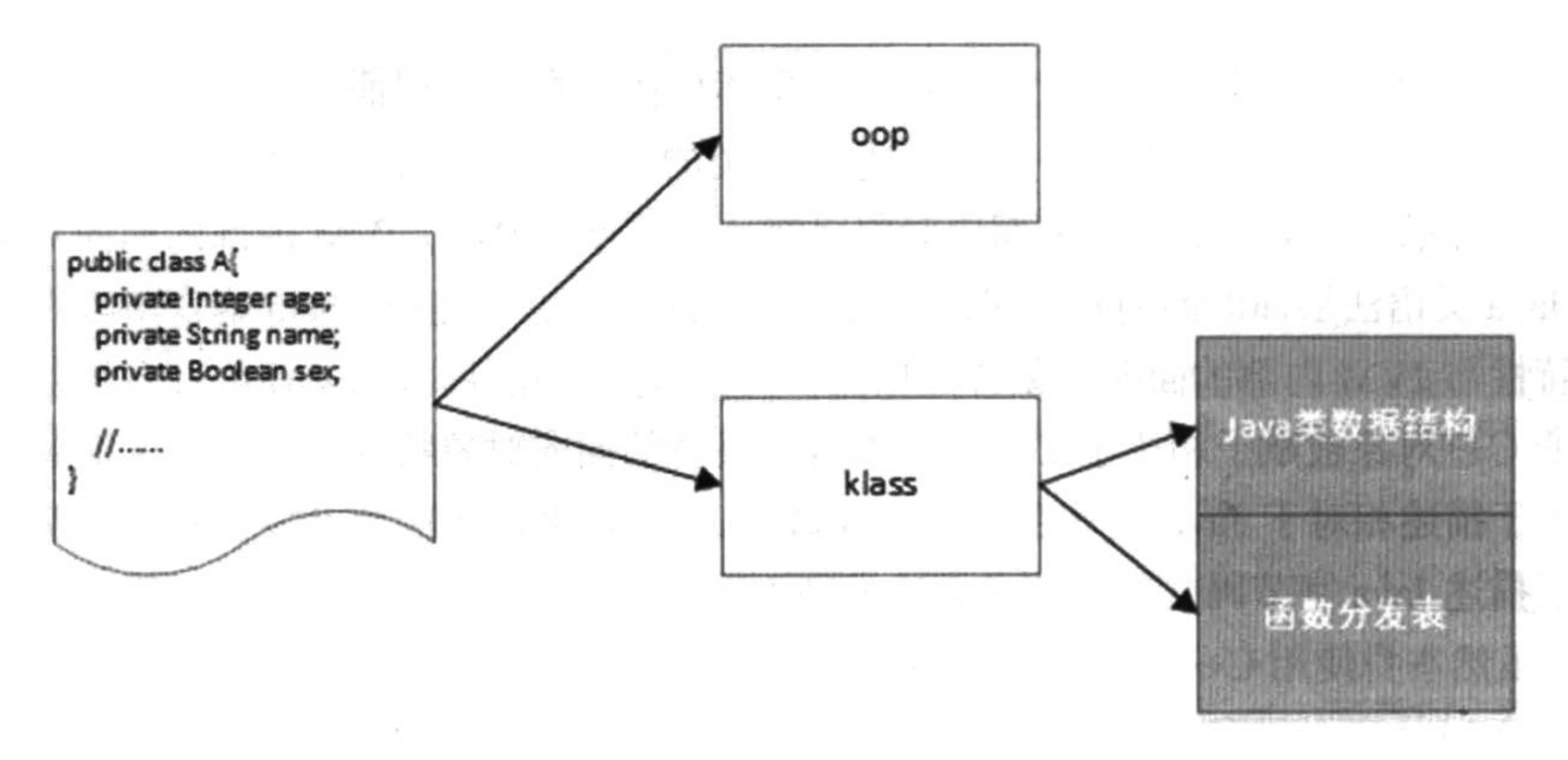

图 5.9　Java 类的两模型三维度

在 Java 编程语言中，并没有“虚函数”的概念，具体来说就是你不能在 Java 源代码中使用“virtual”这个关键字去修饰一个 Java 方法。而有过 C++编程经验的小伙伴都知道，C++实现面向对象多态性的关键字主要就是 virtual。这本来与 Java 毫无关系，毕竟是两种不同的语言，谁也无权强制要求别人为了支持多态就一定要通过“virtual”这个关键字。但是不巧的是，JVM 在内部使用 C++类所定义的一套对象机制去表达 Java 类的面向对象机制，这一表达顺手就把 Java 类的多态机制也包含在内了，毕竟 Java 号称是比 C++更加纯粹的面向对象的语言，结果总不能连个多态都不支持吧。但是詹爷非但不走寻常路，而且还把正常的路径给堵住了不让走，就是不让 Java 支持 virtual 的概念。但是这样就会带来一个问题，Java 类最终被表达成了 JVM 内部的 C++类，并且 Java 类方法的调用最终要通过对应的 C++类，但是 Java 语言是面向对象的，多态性是其基本特性，这意味着 JVM 内部的 C++类要能够支持 Java 语言的多态性，可是 Java 方法并不支持使用 virtual 这个关键字来修饰，这样问题就来了，C++层面怎样才能知道 Java 类中的哪个方法是虚函数，哪个方法不是虚函数呢？换言之，当面对一个多重继承的 Java 类体系时，JVM 内部的 C++类怎么才能将这种多态性表达出来呢？詹爷的做法很简单粗暴，那就是

将 Java 类的所有函数都视为是“virtual”的，这样 Java 类中的每个方法都可以直接被其子类、子子类覆盖而不需要增加任何关键字作为修饰符。正因为如此，Java 类中的每个方法都可以晚绑定，只不过对于一些确定的调用，在编译期便能实现早绑定。

正是因为JVM将Java类中的每一个函数都视为虚函数，所以最终在JVM内部的C++层面，就必须维护一套函数分发表。关于函数分发表，没有 C++编程经验的小伙伴肯定不知其所云，不过不用着急，我们这里慢慢道来。

5.2.2　体系总览

在 JVM 内部定义了 3 种结构去描述一种类型：oop、klass 和 handle 类。注意，这 3 种数据结构不仅能够描述外在的 Java 类，也能够描述 JVM 内在的 C++类型对象。

前面讲过，klass 主要描述 Java 类和 JVM 内部 C++类型的元信息和虚函数，这些元信息的实际值就保存在 oop 里面。oop 中保存一个指针指向 klass，这样在运行期 JVM 便能够知道每一个实例的数据结构和实际类型。handle 是对 oop 的行为的封装，在访问 Java 类时一定是通过 handle 内部指针得到 oop 实例的，再通过 oop 就能拿到 klass，如此 handle 最终便能操纵 oop 的行为了（注意，如果是调用 JVM 内部 C++类型所对应的 oop 的函数，则不需要通过 handle 来中转，直接通过 oop 拿到指定的 klass 便能实现）。klass 不仅包含自己所固有的行为接口，而且也能够操作 Java 类的函数。由于 Java 函数在 JVM 内部都被表示成虚函数，因此 handle 模型其实就是 Java 类行为的表达。

先上一张图说明这种三角关系（如图 5.10 所示）。

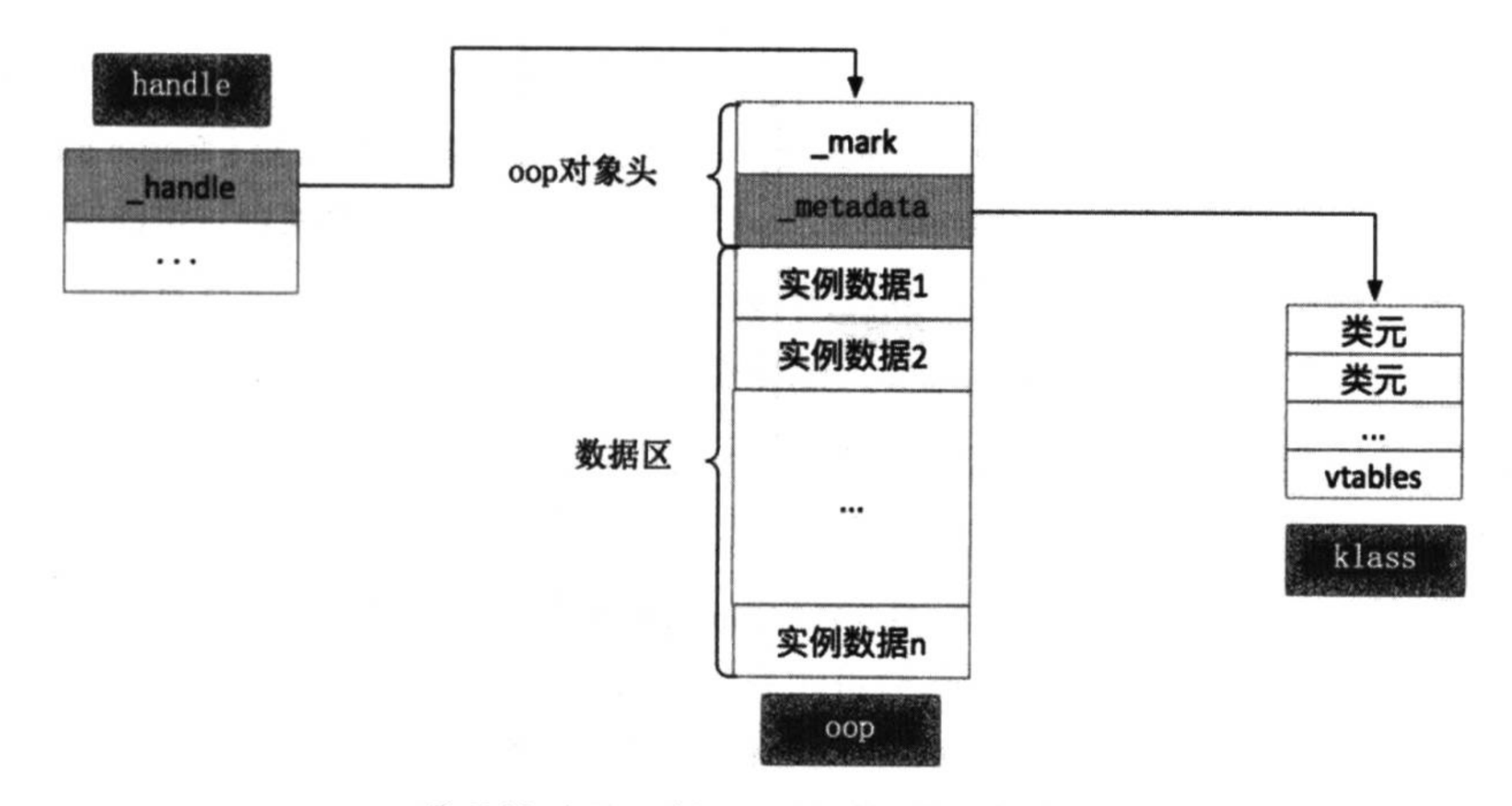

图 5.10　oop、klass、handle 的三角关系

这种三角关系最终在数据结构中得以落地，在 handles.hpp 文件中定义了 Handle 类的基本结构：

清单：/src/share/vm/runtime/handles.hpp

作用：Handle 类的数据结构

```
class Handle VALUE_OBJ_CLASS_SPEC {
 private:
  oop* _handle;

  //……
};
```

可以看到，Handle 类内部只有一个成员变量_handle，该变量类型是 oop*，因此该变量最终指向的就是一个 oop 的首地址。换言之，只要能够拿到 Handle 对象，便能据此得到其所指向的 oop 对象实例，而通过 oop 对象实例又能进一步获取其所关联的 klass 实例，而获取到 klass 对象实例后，便能实现对 oop 对象方法的调用。因此，虽然从表面上看，handle 体系貌似是对 oop 的一种封装，但是实际上其醉翁之意在于最终的 klass 体系。

oop 一般由对象头、对象专有属性和数据体这 3 部分构成。其一般结构如图 5.11 所示。

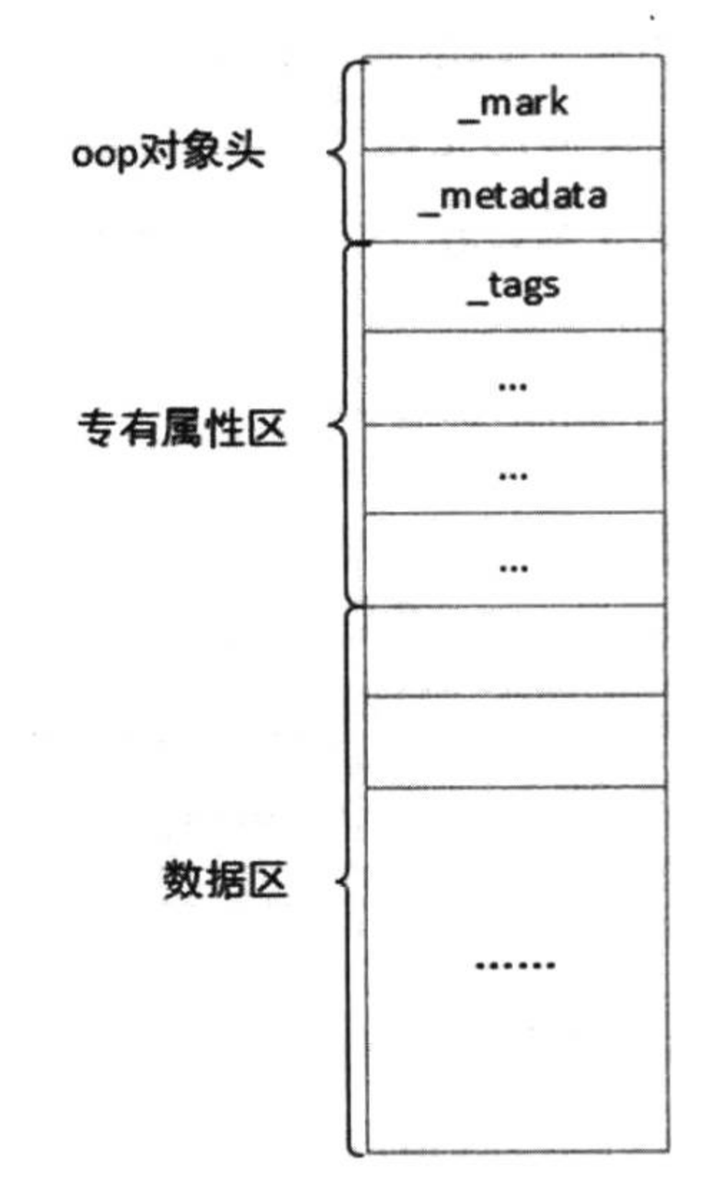

图 5.11　oop 的模型

例如，本节讲解的 JVM 内部对象 constantPool 常量池对象，其 oop 对象是 constantPoolOop，该对象的结构与上面的模型完全符合。上面这张 oop 模型图的中间布局是 oop 类型的专有属性区，JVM 内部定义了若干 oop 类型，每一种 oop 类型都有自己特有的数据结构，oop 的专有属性区便是用于存放各个 oop 所特有的数据结构的地方。

5.2.3　oop 体系

虽然前面我们已经接触过 constantPoolOop，也讲了很多 oop 的东西，但是 oop 究竟是啥？为什么要有这种模型？我们依然疑惑。

所谓 oop，就是 ordinary object pointer，也即普通对象指针。但是究竟什么才是普通对象指针呢？要搞清楚何谓 oop，要问 2 个问题：

1）HotSpot 里的 oop 指啥

Hotspot 里的 oop 其实就是 GC 所托管的指针，每一个 oop 都是一种 xxxOopDesc*类型的指针。所有 oopDesc 及其子类（除神奇的 markOopDesc 外）的实例都由 GC 所管理，这才是最最重要的，是 oop 区分 Hotspot 里所使用的其他指针类型的地方。

2）对象指针之前为何要冠以“普通”二字

对象指针从本质上而言就是一个指针，指向 xxxOopDesc 的指针也是普通得不能再普通的指针，可是为何在 Hotspot 领域还要加一个“普通”来修饰？要回答这个问题，需要追溯到 OOP（这里的 OOP 是指面向对象编程）的鼻祖——SmallTalk 语言。

SmallTalk 语言里的对象也由 GC 来管理，但是 SmallTalk 里面的一些简单的值类型对象都会使用所谓的“直接对象”的机制来实现，例如 SmallTalk 里面的整数类型。所谓“直接对象”（immediate object）就是并不在 GC 堆上分配对象实例，而是直接将实例内容存在对象指针里的对象。这样的指针也叫做“带标记的指针”（tagged pointer）。

这一点倒是与 markOopDesc 类型如出一辙，因为 markOopDesc 也是将整数值直接存储在指针里面，这个指针实际上并无“指向”内存的功能（markOopDesc 会在下一节中讲解）。

所以在 SmallTalk 的运行期，每当拿到一个对象指针时，都得先校验这个对象指针是一个直接对象还是一个真的指针？如果是真的指针，它就是一个“普通”的对象指针了。这样对象指针就有了“普通”与“不普通”之分。

所以，在 HotSpot 里面，oop 就是指一个真的指针，而 markOop 则是一个看起来像指针但实际上是藏在指针里的对象（数据）。这也正是 markOop 实例不受 GC 托管的原因，因为只要出了函数作用域，指针变量就会直接被从堆栈上释放掉了，不需要垃圾回收了。

JVM 内部并不是 oop 一个人在战斗，而是整个一套继承体系在运作。在 oopsHierarchy.hpp 中定义了这个体系家族的所有成员：

清单：/src/share/vm/oops/oopsHierarchy.hpp
作用：oop 的继承体系

```
typedef class    oopDesc*                    oop;
typedef class    instanceOopDesc*            instanceOop;
typedef class    methodOopDesc*              methodOop;
typedef class    constMethodOopDesc*         constMethodOop;
typedef class    methodDataOopDesc*          methodDataOop;
typedef class    arrayOopDesc*               arrayOop;
typedef class    objArrayOopDesc*            objArrayOop;
typedef class    typeArrayOopDesc*           typeArrayOop;
typedef class    constantPoolOopDesc*        constantPoolOop;
typedef class    constantPoolCacheOopDesc*   constantPoolCacheOop;
typedef class    klassOopDesc*               klassOop;
typedef class    markOopDesc*                markOop;
typedef class    compiledICHolderOopDesc*    compiledICHolderOop;
```

这些不同的 oop 类型能够分别描述不同的对象，具体作用见如表 5.1 所示。

表 5.1　opp 类型描述的对象

oop	所有 oop 的顶级父类
instanceOop	表示 Java 类实例
methodOop	表示 Java 方法
constMethodOop	表示 Java 方法中的只读信息（其实就是字节码指令）
methodDataOop	表示性能统计的相关数据
arrayOop	数组对象
objArrayOop	表示引用类型数组对象
typeArrayOop	表示基本类型数组对象
constantPoolOop	表示 Java 字节码文件中的常量池
constantPoolCacheOop	与 constanttPoolOop 相伴生，是后者的缓存对象
klassOop	指向 JVM 内部的 klass 实例的对象
markOop	oop 的标记对象

面对这十几种不同的 oop 大可不必惊慌，事实上只需要关注两种最常用的即可：constantPoolOop 和 instanceOop。其中，constantPoolOop 已经在上一节中详细描述过，而 instanceOop 将会在后面的章节详细描述。作为 Java 程序的解释器和虚拟运行介质，JVM 将 Java 实例映射成 instanceOop，这注定 instanceOop 是一个无法跳过的坎，当然也注定其过程一定很

精彩。

虽然 oop 类型比较多，但是只要深入研究一番 constantPoolOop 和 instanceOop 这两个最重要的 oop，其他的 oop 自会触类旁通。

在这里有必要再次对 JDK 的版本问题做个说明。可能很多 JVM 发烧友都知道，JDK 8 中的 oop 与 JDK 6 相比变化很大，整个继承体系都发生了变化，但是笔者认为本质上并没有发生多大变化。JDK 8 中仍然有描述 constantPool 常量池的 oop，仍然有描述 Java 类实例对象的 oop，JDK 8 并没有抛弃 Java 字节码文件中的常量池，没有对 Java 字节码文件结构进行大刀阔斧的调整（事实上这已经成为 Java 的一种标准，想改也改不了），并没有对描述 Java 类的 oop-klass 这种二分模型进行根本上的改变。只要这些机制或标准不发生彻底的变化，那么相应的实现机制便注定不可能有本质上的不同。因此 JDK 8 虽然修改了继承关系，对一些 oop 进行了删减，但是在本质上与 JDK 6 仍然保持一致。正是由于这种原因，对 JDK 6 的研究并不会变得过时，更不会徒劳无功。

所以，本书基本以 JDK 6 源码为主，分析 JVM 的内存模型。如果小伙伴对 JDK 8 情有独钟，也完全不用担心辛辛苦苦做的研究会白费掉。

还有一点需要注意，由于 oopDesc 不同的子类类型名称都以 OopDesc 结尾，因此为了简化，JVM 为其取了别名，统一以 oop 结尾。

5.2.4　klass 体系

oop 的讲述先告一段落，再来看看 klass 部分。按照 JVM 的官方解释，klass 主要提供下面 2 种能力：

◎　klass 提供一个与 Java 类对等的 C++类型描述。

◎　klass 提供虚拟机内部的函数分发机制。

其实这种说法与上文所说的 2 种维度的含义是相同的。klass 分别从类结构和类行为这两方面去描述一个 Java 类（当然也包含 JVM 内部非开放的 C++类）。

与 oop 相同，在 JVM 内部也不是 klass 一个人在战斗，而是一个家族。klass 家族体系如下：

清单：/src/share/vm/klass/klassHierarchy.hpp
功能：klass 的继承体系

```
class    Klass;
class    instanceKlass;
class    instanceMirrorKlass;
class    instanceRefKlass;
```

```
class    methodKlass;
class    constMethodKlass;
class    methodDataKlass;
class    klassKlass;
class    instanceKlassKlass;
class    arrayKlassKlass;
class    objArrayKlassKlass;
class    typeArrayKlassKlass;
class    arrayKlass;
class    objArrayKlass;
class    typeArrayKlass;
class    constantPoolKlass;
class    constantPoolCacheKlass;
class    compiledICHolderKlass;
```

klass 家族一看就比 oop 家族要人丁兴旺，让人望而生畏。但是不要紧，只要深入研究了其中两个最重要的 klass，其他的便都会是浮云。

在 klass 家族里，除去与 constantPoolOop 相对应的 constantPoolKlass 之外，最重要的莫过于 instanceKlass 和 klassKlass 了。这两个在后面的章节会陆续介绍，其中，instanceKlass，顾名思义，其实就是专门用于描述 Java 类的。名字取得好就这点好处，往往只需看个名字便能知其八分，绝对比看相的和算命的要准很多，所以本书中会使用很多次“顾名思义”，因为很多事情的真相其实就隐藏在变量名称或者类型的名称里面，无须太多解释。

klass 家族的基类是 Klass 类，而 Klass 其实并不是顶级父类，Klass 继承了一个名叫 Klass_vtbl 的类，后者才是整个 klass 家族的“元老”。当然，顾名思义，后者貌似是一个描述 vtbl（即虚函数表）的类，而事实上该类的确当此大任。

klass 家族的各个成员的定位是不同的，具体如表 5.2 所示（由于 JDK 6 中的很多 klass 在 JDK 8 中已经被删减了，因此这里仅挑重要的几个进行描述，有兴趣的小伙伴可以自行查阅 JDK 6 源码）。

表 5.2　klass 家族成员

类	定　位
Klass	klass 家族的基类
instanceKlass	虚拟机层面与 Java 类对等的数据结构
instanceMirrorKlass	描述 java.lang.Class 的实例
instanceRefKlass	描述 java.lang.ref.Reference 的子类
methodKlass	表示 Java 类的方法
constMethodKlass	描述 Java 类方法所对应的字节码指令信息的固有属性

续表

类	定　位
klassKlass	klass 链路的末端。 该类型在 JDK 8 中已经不复存在，但是由于其比较重要，因此本书依然会介绍这个特殊的 klass
arrayKlass	描述 Java 数组的信息，是个抽象基类
typeArrayKlass	描述 Java 中基本类型数组的数据结构
objArrayKlass	描述 Java 中引用类型数组的数据结构
constPoolKlass	描述 Java 字节码文件中的常量池的数据结构

由于 JDK 8 并没有对 JDK 6 进行本质上的模型变革，因此 JDK 6 中的这些重要的 klass 模型依然能够在 JDK 8 中寻找到，除了 klassKlass 这个特殊的类型。

既然 klass 能够描述 Java 类的数据结构（即元数据），那么一起来看看 klass 类里面究竟有些什么。基类 Klass 的定义如下：

清单：/src/share/vm/oops/klass.hpp
功能：Klass 的数据结构

```
class Klass : public Klass_vtbl {
protected:
    jint        _layout_helper;

    juint       _super_check_offset;

    Symbol*     _name;

public:
    oop* oop_block_beg() const { return adr_secondary_super_cache(); }
    oop* oop_block_end() const { return adr_next_sibling() + 1; }

protected:

    //========================oop block START========================//
    klassOop    _secondary_super_cache;

    objArrayOop _secondary_supers;

    klassOop    _primary_supers[_primary_super_limit];

    oop       _java_mirror;

    klassOop  _super;
```

```
    klassOop _subklass;

    klassOop _next_sibling;
    //=========================oop block END=========================//

    jint        _modifier_flags;
    AccessFlags _access_flags;

    juint    _alloc_count;

    jlong    _last_biased_lock_bulk_revocation_time;
    markOop  _prototype_header;
    jint     _biased_lock_revocation_count;
}
```

该类包含的字段比较多，相关字段含义或作用如表 5.3 所示。

表 5.3　klass 类相关字段

字段名	含义/作用
_layout_helper	对象布局的综合描述符
_name	类名。例如，java.lang.String 类的_name 属性值会是 java/lang/String
_java_mirror	类的镜像类
_super	父类
_subklass	指向第一个子类，若无，则为 NULL
_next_sibling	指向下一个兄弟节点，若无，则为 NULL
_modifier_flags	修饰符标识，例如 static
_access_flags	访问权限标识，例如 public

如果一个 Klass 既不是 instance 也不是 array，则其_layout_helper 被设置为 0。如果 Klass 标识一个 instance，则其_layout_helper 为正数，其值表示 instance 的大小。如果 Klass 代表的是一个数组，则_layout_helper 为负数。

5.2.5　handle 体系

前面讲过，handle 封装了 oop，由于通过 oop 可以拿到 klass，而 klass 是对 Java 类数据结构和方法的描述，因此 handle 间接封装了 klass。JVM 内部使用一个 table 来存储 oop 指针。

如果说 oop 是对普通对象的直接引用，那么 handle 就是对普通对象的一种间接引用，中间

隔了一层。但是 JVM 内部为何要使用这种间接引用呢？答案是，这完全是为 GC 考虑。具体表现在 2 个地方：

◎ 通过 handle，能够让 GC 知道其内部代码都有哪些地方持有 GC 所管理的对象的引用，这只需要扫描 handle 所对应的 table，这样 JVM 便无须关注其内部到底哪些地方持有对普通对象的引用。

◎ 在 GC 过程中，如果发生了对象移动（例如从新生代移到了老一代），那么 JVM 的内部引用无须跟着更改为被移动对象的新地址，JVM 只需要更改 handle table 里对应的指针即可。

当然实际的 handle 作为对 Java 类方法的访问的包装，远不止上面所描述的这么简单。这里涉及 Java 类的类继承和接口继承的话题，在 C++领域，类的继承和多态性最终通过 vptr（虚函数表）来实现。在 klass 内部，记录了每一个类的 vptr 信息，具体而言分为两部分来描述。

1．vtable 虚函数表

vtable 中存放 Java 类中非静态和非 private 的方法入口，JVM 调用 Java 类的方法（非静态和非 private）时，最终会访问 vtable，找到对应的方法入口。

2．itable 接口函数表

Itable 中存放 Java 类所实现的接口类方法。同样，JVM 调用接口方法时，最终会访问 itable，找到对应的接口方法入口。

不过要注意，vtable 和 itable 里面存放的并不是 Java 类方法和接口方法的直接入口，而是指向了 Method 对象入口，JVM 会通过 Method 最终拿到真正的 Java 类方法入口，得到方法所对应的字节码/二进制机器码并执行。当然，对于被 JIT 进行动态编译后的方法，JVM 最终拿到的是其对应的被编译后的本地方法的入口。

关于 vtable 和 itable 的话题先讲到这里，后面会单独开辟一个章节来详细讲解。

与 oop 和 klass 一样，在 JVM 内部，handle 也不是一个人在战斗，而是有一个庞大的家族。在 handles.hpp 文件中定义了 handle 体系的家族：

清单：/src/share/vm/runtime/handles.hpp
功能：handle 家族

```
//handle体系基类
class Handle VALUE_OBJ_CLASS_SPEC {
 //...
};
```

```
//KlassHandle基类
class KlassHandle: public Handle {
//...
};

//-----------------------------------------------------------------------
-----------------------------------------------
//oop家族所对应的handle定义宏
#define DEF_HANDLE(type, is_a)                                  \
  class type##Handle;                                          \
  class type##Handle: public Handle {                           \
   protected:                                                  \
    type##Oop  obj() const            { return (type##Oop)Handle::obj(); } \
    type##Oop non_null_obj() const { return (type##Oop)Handle::non_null_obj(); } \
                                                               \
   public:                                                     \
    /* Constructors */                                          \
    type##Handle ()                                            : Handle()            {} \
    type##Handle (type##Oop obj) : Handle((oop)obj) {                  \
      assert(SharedSkipVerify || is_null() || ((oop)obj)->is_a(),      \
             "illegal type");                                          \
    }                                                                  \
    type##Handle (Thread* thread, type##Oop obj) : Handle(thread, (oop)obj) { \
      assert(SharedSkipVerify || is_null() || ((oop)obj)->is_a(), "illegal type"); \
    }                                                                  \
    \
    /* Special constructor, use sparingly */ \
    type##Handle (type##Oop *handle, bool dummy) : Handle((oop*)handle, dummy) {} \
                                                               \
    /* Operators for ease of use */                             \
    type##Oop    operator () () const          { return obj(); } \
    type##Oop    operator -> () const          { return non_null_obj(); } \
  };

//定义oop家族的handle体系
DEF_HANDLE(instance              , is_instance          )
DEF_HANDLE(method                , is_method            )
DEF_HANDLE(constMethod           , is_constMethod       )
DEF_HANDLE(methodData            , is_methodData        )
DEF_HANDLE(array                 , is_array             )
DEF_HANDLE(constantPool          , is_constantPool      )
DEF_HANDLE(constantPoolCache     , is_constantPoolCache)
DEF_HANDLE(objArray              , is_objArray          )
DEF_HANDLE(typeArray             , is_typeArray         )
```

```
//-----------------------------------------------------------------------
-----------------------------------------------
//klass家族所对应的handle宏定义
#define DEF_KLASS_HANDLE(type, is_a)                    \
  class type##Handle : public KlassHandle {             \
   public:                                              \
    /* Constructors */                                  \
  type##Handle ()                                       : KlassHandle()          {} \
  type##Handle (klassOop obj) : KlassHandle(obj) {                                  \
    assert(SharedSkipVerify || is_null() || obj->klass_part()->is_a(),  \
           "illegal type");                                                          \
  }                                                                                  \
  type##Handle (Thread* thread, klassOop obj) : KlassHandle(thread, obj) {  \
    assert(SharedSkipVerify || is_null() || obj->klass_part()->is_a(), \
           "illegal type");                                                          \
  }                                                                                  \
                                                        \
  /* Access to klass part */                                \
  type*       operator -> () const  { return (type*)obj()->klass_part(); } \
                                                        \
  static type##Handle cast(KlassHandle h) { return type##Handle(h()); } \
                                                        \
  };

//定义klass家族的handle体系
DEF_KLASS_HANDLE(instanceKlass              , oop_is_instance_slow )
DEF_KLASS_HANDLE(methodKlass                , oop_is_method        )
DEF_KLASS_HANDLE(constMethodKlass           , oop_is_constMethod   )
DEF_KLASS_HANDLE(klassKlass                 , oop_is_klass         )
DEF_KLASS_HANDLE(arrayKlassKlass            , oop_is_arrayKlass    )
DEF_KLASS_HANDLE(objArrayKlassKlass         , oop_is_objArrayKlass )
DEF_KLASS_HANDLE(typeArrayKlassKlass        , oop_is_typeArrayKlass)
DEF_KLASS_HANDLE(arrayKlass                 , oop_is_array         )
DEF_KLASS_HANDLE(typeArrayKlass             , oop_is_typeArray_slow)
DEF_KLASS_HANDLE(objArrayKlass              , oop_is_objArray_slow )
DEF_KLASS_HANDLE(constantPoolKlass          , oop_is_constantPool  )
DEF_KLASS_HANDLE(constantPoolCacheKlass , oop_is_constantPool  )
```

可以看到，在 handles.hpp 中，通过 2 个宏分别批量声明了 oop 和 klass 家族的各个类所对应的 handle 类型。

在编译期，宏被替换后，便出现了如下 handle 体系（见表 5.4 和表 5.5）。

表 5.4　oop 体系所对应的 handle 体系

类	定　位
instanceHandle	类实例 handle

续表

类	定　位
methodHandle	方法实例 handle
constMethodHandle	方法字节码实例 handle
methodDataHandle	性能统计 handle
arrayHandle	数组 handle
constantPoolHandle	常量池 handle
constantPoolCacheHandle	常量池缓存 handle
objArrayHandle	引用类型数组 handle
typeArrayHandle	基本类型数组 handle

表 5.5　klass 体系所对应的 handle 体系

类	定　位
instanceKlassHandle	类元结构 handle
methodKlassHandle	方法元结构 handle
constMethodKlassHandle	方法固定类元结构 handle
klassKlassHandle	klass 链路末端类 handle
arrayKlassKlassHandle	描述基类数组元结构 handle
objArrayKlassKlassv	描述引用类型数组的类元结构 handle
typeArrayKlassKlasHandle	描述基本类型数组的类元结构 handle
arrayKlassHandle	基类数组元结构 handle
typeArrayKlassHandle	基本类型数组的类元结构 handle
objArrayKlassHandle	引用类型数组的类元结构 handle
constantPoolKlassHandle	常量池类元结构 handle
constantPoolKlassHandle	常量池缓存类元结构 handle

这里有个问题，前面不是一直在说 handle 是对 oop 的直接封装和对 klass 的间接封装吗，为什么这里却分别给 oop 和 klass 定义了 2 套不同的 handle 体系呢？这给人的感觉好像是，封装 oop 的 handle 和封装 klass 的 handle 并不是同一个 handle，既然不是同一个 handle，那么通过封装 oop 的 handle 还怎么去得到所对应的 klass 信息呢？

其实这正是 JVM 内部常常容易使人迷惑的地方。在 JVM 中，使用 oop-klass 这种一分为二的模型去描述 Java 类以及 JVM 内部的特殊类群体，为此 JVM 内部特定义了各种 oop 和 klass

类型。但是，对于每一个 oop，其实都是一个 C++类型，也即 klass；而对于每一个 klass 所对应的 class，在 JVM 内部又都会被封装成 oop。JVM 在具体描述一个类型时，会使用 oop 去存储这个类型的实例数据，并使用 klass 去存储这个类型的元数据和虚方法表。而当一个类型完成其生命周期后，JVM 会触发 GC 去回收，在回收时，既要回收一个类实例所对应的实例数据 oop，也要回收其所对应的元数据和虚方法表（当然，两者并不是同时回收，一个是堆区的垃圾回收，一个是永久区的垃圾回收）。为了让 GC 既能回收 oop 也能回收 klass，因此 oop 本身被封装成了 oop，而 klass 也被封装成 oop。而 JVM 内部恰好将描述类实例的 oop 全都定义成类名以 oop 结尾的类，并将描述类结构和方法信息的 klass 全都定义成类名以 klass 结尾的类，而 JVM 内部描述类信息的模型恰巧也叫作 oop-klass，与类名存在重合，这就导致了很多人的疑惑，这些疑惑完全是因为叫法上的重合而产生。

因此为了进一步解开疑惑，我们不妨换个叫法，不再将 JVM 内部描述类信息的模型叫作 oop-klass，而是叫作 data-meta 模型（瞎取的，名字没啥特殊含义）。然后将 JVM 内部的 oop 体系的类名全都改成以 Data 结尾，例如，methodData、instanceData、constantPoolData 等，同时将 klass 体系的类名也全都改成以 Meta 结尾，例如，methodMeta、instanceMeta、constantPoolMeta 等。JVM 在进行 GC 时，既要回收 Data 类实例，也要回收 Meta 类实例，为了让 GC 便于回收，因此对于每一个 Data 类和每一个 Meta 类，JVM 在内部都将其封装成了 oop 模型。对于 Data 类，其内存布局是前面为 oop 对象头，后面紧跟实例数据；而对 Meta 类，其内存布局是前面为 oop 对象头，后面紧跟实例数据和虚方法表。封装成 oop 之后，再进一步使用 handle 来封装，于是便有利于 GC 内存回收。

在这种新的模型中，不管是 Data 类还是 Meta 类，都是一种普通的 C++类型，只不过它们从不同的角度对 Java 类进行了描述。不管是 Data 类还是 Meta 类，当其所在的 JVM 的内存区域爆满后，都会触发 GC，为了方便回收，因此就需要将其封装成 oop。这样说，小伙伴们都懂了吗？总之，原来的说法里，对 Java 类的描述模型里有一个 oop 的概念，存储 Java 类实例数据的类也都是以 oop 结尾，为了方便 GC 回收而使用的封装策略也叫作 oop，多种概念混在一起，安能不晕？

5.2.6　oop、klass、handle 的相互转换

oop、klass 和 handle 三者之间是可以相互转换的。

1．从 oop 和 klass 到 handle

handle 主要用于封装 oop 和 klass，因此往往在声明 handle 类实例的时候，直接将 oop 或者 klass 传递进去，便完成了这种封装。同时，当 JVM 执行 Java 类的方法时，最终也是通过 handle

拿到对应的 oop 和 klass。而为了支持高效快速的调用，JVM 重载了类方法访问操作符->。

oop 类型所对应的 handle 的基类是 Handle，Handle 有如下几种构造函数：

清单：/src/share/vm/runtime/handles.inline.hpp
功能：Handle 类的构造函数

```
inline Handle::Handle(oop obj) {
  if (obj == NULL) {
    _handle = NULL;
  } else {
    _handle = Thread::current()->handle_area()->allocate_handle(obj);
  }
}
```

这个构造函数接受 oop 类型的入参，并将其保存到当前线程在堆区所申请的 handleArea 表中。注意，由于 oop 仅仅是一种指针，因此表中存储的实际上也是指针。

除了这种带有入参的构造函数，还有默认构造函数：

清单：/src/share/vm/runtime/handles.hpp
功能：Handle 类的构造函数

```
Handle() {  _handle = NULL; }
```

oop 和 klass 被 handle 封装之后，JVM 内部大部分对 oop 和 klass 的函数调用都要经过 Handle 类。而从 Handle 类到 oop 或者 klass，都必须经过至少一次中间过渡性的寻址，因此为了减少寻址次数，Handle 重载了操作符->：

清单：/src/share/vm/runtime/handles.hpp
功能：Handle 类的构造函数

```
oop  non_null_obj() const  {
    assert(_handle != NULL, "resolving NULL handle");
    return *_handle;
}

oop  operator -> () const  {
    return non_null_obj();
}
```

这里定义了 oop operate ->()操作符重载函数，返回 non_null_obj()，而后者则直接返回 oop 类型的*_handle 指针，因此如果 JVM 想要调用 oop 的某个函数，可以直接通过 handle。例如，在常量池类型 constantPoolOopDesc 中定义了 void field_at_put(int which, int class_index, int name_and_type_index)函数，该函数将解析出的常量池元素保存进 constantPoolOop 对象头后面的数据实例区中，在解析常量池的过程中，JVM 并没有直接通过 constantPoolOop 指针来调用

这个函数，而是通过 constantPoolHandle。

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

功能：通过 handle 调用 oop 函数举例

```
void ClassFileParser::parse_constant_pool_entries(constantPoolHandle
cp, int length, TRAPS) {

  // 这里省略若干代码

  for (int index = 1; index < length; index++) {
    u1 tag = cfs->get_u1_fast();
    switch (tag) {
     //...
      case JVM_CONSTANT_Fieldref :
        {
          cfs->guarantee_more(5, CHECK);  // class_index,
name_and_type_index, tag/access_flags
          u2 class_index = cfs->get_u2_fast();
          u2 name_and_type_index = cfs->get_u2_fast();
          cp->field_at_put(index, class_index, name_and_type_index);
        }
        break;
      //…
      }
    }
  }
}
```

这里面执行了 cp->field_at_put()函数，而 cp 则是 constantPoolHandle 类型。

klass 体系的 Handle 类全都继承于 KlassHandle 这个基类，与 oop 体系类似，KlassHandle 中也定义了多种构造函数用来实现对 klass 或 oop 的封装，并且重载了 operate ->()函数实现从 handle 直接调用 klass 的函数。JVM 创建 Java 类所对应的类模型时便使用了这种方式：

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

功能：通过 handle 调用 oop 函数举例

```
instanceKlassHandle ClassFileParser::parseClassFile(Symbol* name,
                                                    Handle class_loader,
                                                    Handle
protection_domain,
                                                    KlassHandle host_klass,
                                                    GrowableArray<Handle>*
cp_patches,
                                                    TempNewSymbol&
parsed_name,
                                                    bool verify,
                                                    TRAPS) {
```

```
    // 这里省略若干代码

    klassOop ik = oopFactory::new_instanceKlass(name, vtable_size,
itable_size,
                                                static_field_size,
                                                total_oop_map_count,
                                                rt, CHECK_(nullHandle));
    instanceKlassHandle this_klass (THREAD, ik);

    // Fill in information already parsed
    this_klass->set_access_flags(access_flags);
    this_klass->set_should_verify_class(verify);
    jint lh = Klass::instance_layout_helper(instance_size, false);
    this_klass->set_layout_helper(lh);
    assert(this_klass->oop_is_instance(), "layout is correct");
    assert(this_klass->size_helper() == instance_size, "correct
size_helper");
    // Not yet: supers are done below to support the new subtype-checking
fields
    //this_klass->set_super(super_klass());
    this_klass->set_class_loader(class_loader());
    this_klass->set_nonstatic_field_size(nonstatic_field_size);
    this_klass->set_has_nonstatic_fields(has_nonstatic_fields);

    //…

}
```

在该函数中，先通过 klassOop ik = oopFactory::new_instanceKlass()创建了一个 klassOopDesc 实例，接着通过 instanceKlassHandle this_klass (THREAD, ik)将 oop 封装到了 instantceKlassHandle 中，接下来便通过 this_klass 指针来直接调用 instanceKlass 中的各种函数。

2．klass 与 oop 的相互转换

为了便于 GC 回收，每一种 klass 实例最终都要被封装成对应的 oop，具体操作时，先分配对应的 oop 实例，接着将 klass 实例分配到 oop 对象头的后面，从而实现 oop+klass 这种内存布局结构。对于任何一种给定的 oop 和其对应的 klass，oop 对象首地址到其对应的 klass 对象首地址的距离都是固定的，因此只要得到了 oop 对象首地址，便能通过偏移固定的距离得到 klass 对象的首地址。反之，得到 klass 对象的首地址后，也能通过偏移固定的距离得到 oop 对象的首地址。通过内存偏移，便能实现 oop 和 klass 的相互转换。对于每一种 oop，都提供了 klass_part()这样的函数，通过本函数可以直接由 oop 得到对应的 klass 实例。例如 klassOop 便提供了这种函数：

清单：/src/share/vm/oops/klassOop.hpp
功能：oop 转换为 klass

```
Klass* klass_part() const  {
   return (Klass*)((address)this + klass_part_offset_in_bytes());
}
```

由于 klass 在内存上相对于 oop 实例位于高地址方向，因此从 oop 转换到 klass 只需要增加 oop 的首地址。同理，如果将 klass 转换为 oop，则只需要对 klass 首地址做减法。例如：

清单：/src/share/vm/oops/klass.hpp
功能：klass 转换为 oop

```
klassOop as_klassOop() const {
    return (klassOop) (((char*) this) - sizeof(klassOopDesc));
}
```

下面还是以前文所举的学生类 Student 为例，假设在某个 Java 方法中连续实例化了 3 个 Student 类，那么最终在 JVM 内存中将会出现如图 5.12 所示这种布局。

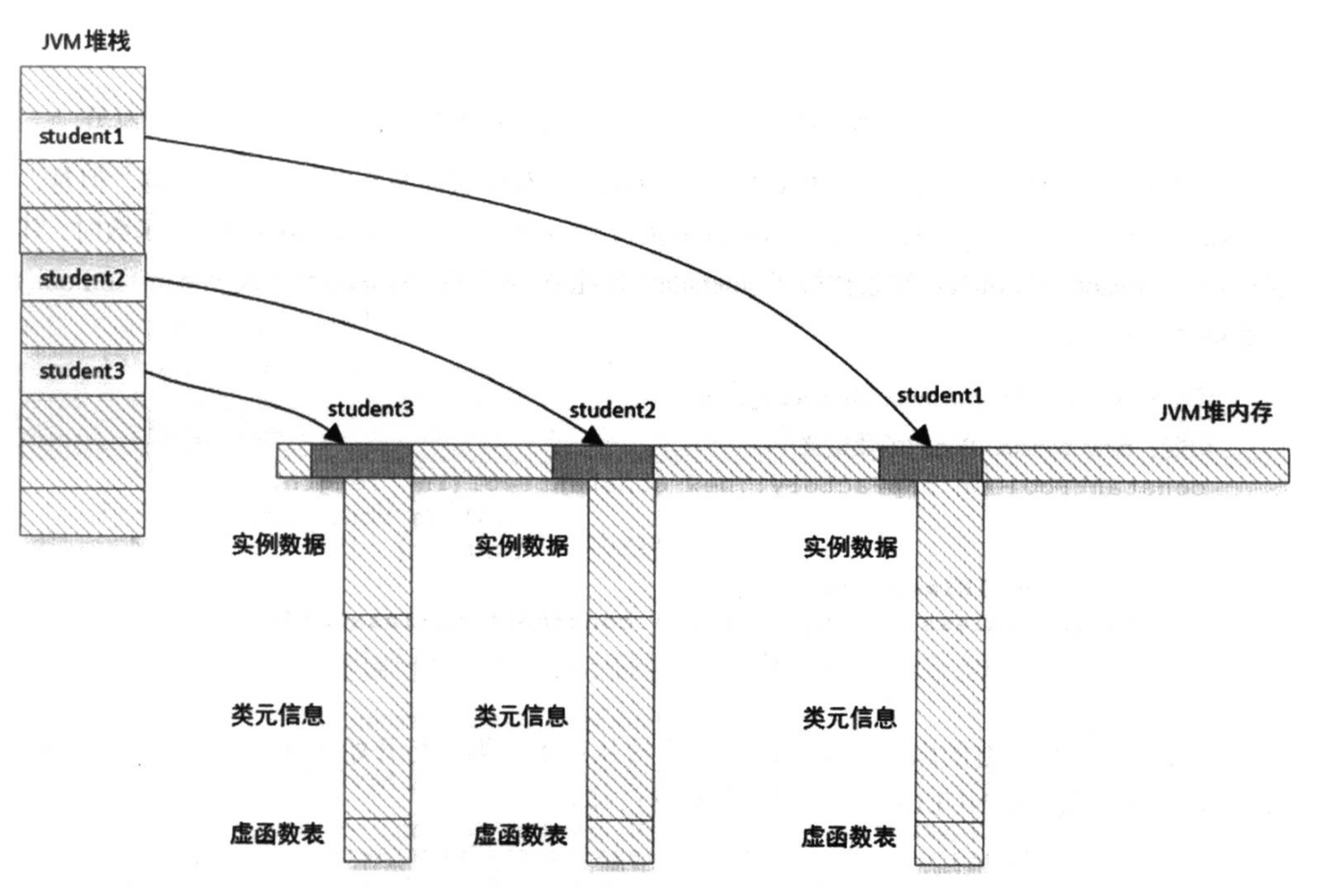

图 5.12　以 Student 类演示 Java 类内存布局

5.3 常量池 klass 模型（1）

既然在 JVM 内部，每一个对象都会表示为 oop-klass 这种一分为二的模型，那么常量池也不例外。在前面的讲解中，已经分析了 JVM 为 constantPoolOop 所分配的内存大小和内存布局，具体的内存布局如图 5.13 所示。

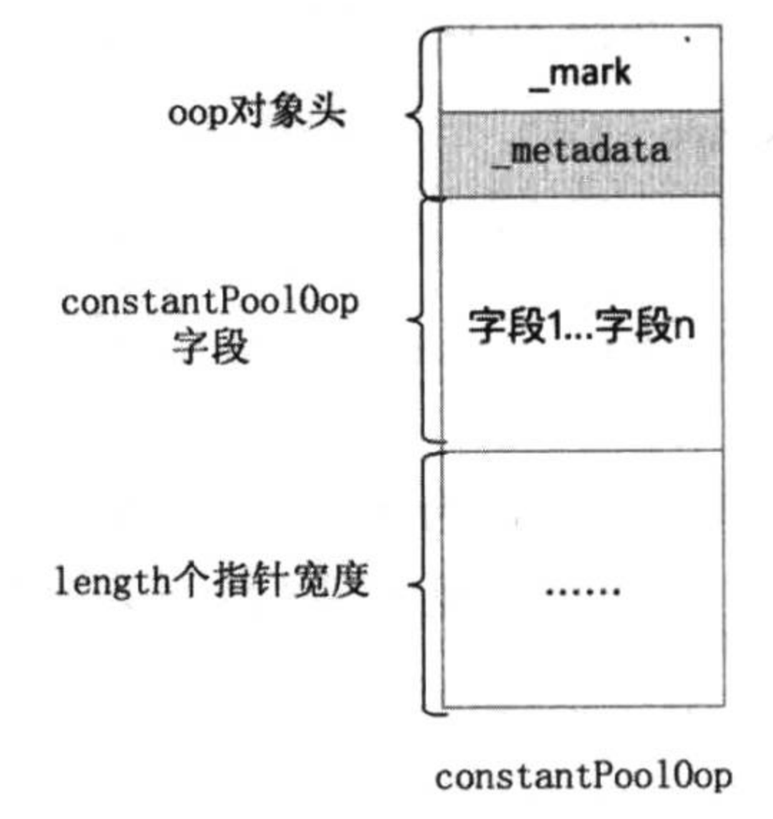

图 5.13 constantPoolOop 的内存布局

到目前为止，JVM 为 constantPoolOop 所分配的内存区域还是空的，还没有数据。但是，在 ClassFileParser::parse_constant_pool()函数中执行完 oopFactory::new_constantPool()函数时，其实已经为 consntantPoolOop 初始化好了_metadata 所指向的实例。oopFactory::new_constantPool()函数的逻辑如下：

清单：/src/share/vm/memory/oopFactory.cpp
功能：new_constantPool()函数定义

```
constantPoolOop oopFactory::new_constantPool(int length,
                                             bool is_conc_safe,
                                             TRAPS) {
  constantPoolKlass* ck =
  constantPoolKlass::cast(Universe::constantPoolKlassObj());
    return ck->allocate(length, is_conc_safe, CHECK_NULL);
  }
```

前文在分析常量池的内存分配链路时，直接从这里的第二行开始（即 ck->allocate(length, is_conc_safe, CHECK_NULL)），而忽略了第一行，即

```
constantPoolKlass*ck= constantPoolKlass::cast(Universe::constantPoolKlassObj
());
```

这行代码调用全局对象 Universe 的静态函数 constantPoolKlassObj()来获取 constantPoolKlass 实例指针，Universe::constantPoolKlassObj()函数逻辑如下：

```
static klassOop constantPoolKlassObj() {
    return _constantPoolKlassObj;
}
```

这个函数直接返回了全局静态变量_constantPoolKlassObj。该变量在 JVM 启动过程中被实例化，在 JVM 初始化过程中，会调用如下逻辑：

清单：/src/share/vm/oops/constantPoolKlass.cpp
功能：create_klass()函数定义

```
klassOop constantPoolKlass::create_klass(TRAPS) {
  constantPoolKlass o;
  KlassHandle h_this_klass(THREAD, Universe::klassKlassObj());
  KlassHandle k = base_create_klass(h_this_klass, header_size(),
o.vtbl_value(), CHECK_NULL);
  java_lang_Class::create_mirror(k, CHECK_NULL); // Allocate mirror
  return k();
}
```

在这个函数中，先调用 KlassHandle h_this_klass(THREAD, Universe::klassKlassObj())函数获取 klassKlass 实例。这个实例究竟是啥姑且不管（后面会讲到），在 JVM 初始化过程中，会执行如下逻辑：

清单：/src/share/vm/oops/klassKlass.cpp
功能：create_klass()函数定义

```
klassOop klassKlass::create_klass(TRAPS) {
  KlassHandle h_this_klass;
  klassKlass o;
  klassOop k = base_create_klass_oop(h_this_klass, header_size(),
o.vtbl_value(), CHECK_NULL);
  k->set_klass(k); // point to thyself
  return k;
}
```

在这个逻辑中，最终调用 base_create_klass_oop()函数创建 klassKlass 的实例，该实例是全局性的，可以通过调用 Universe::klassKlassObj()函数获取到。

至此我们知道，在 JVM 启动过程中，先创建了 klassKlass 实例，再根据该实例，创建了常量池所对应的 Klass 类——constantPoolKlass。因此，欲分析 constantPoolKlass 实例的构建机制，首先就要分析 klassKlass 实例的构建。下面就先从 klassKlass 实例的构建原理说起。

5.3.1 klassKlass 实例构建总链路

先上一张 klassKlass 实例构建的总体链路图，如图 5.14 所示。

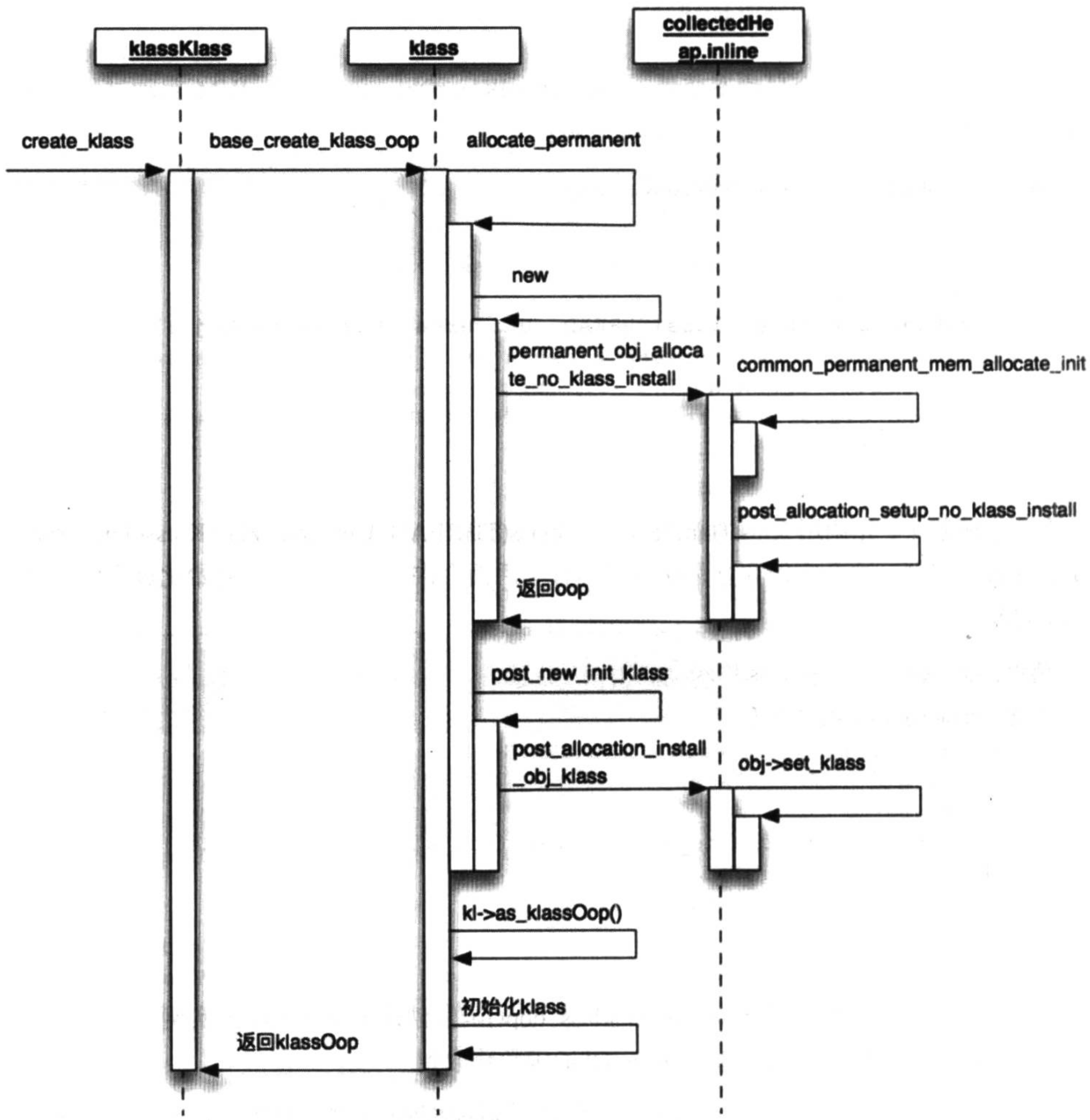

图 5.14 klassKlass 实例构建的总体链路

klassKlass.cpp::create_klass()函数位于非常核心的位置，该位置也是理解 JVM 内存模型的关键，将这个理解透了，JVM 内存模型便明白了一大半。由于这个函数实在太重要了，因此十分

有必要对其进行详细讲解。

图 5.14 所示的调用链路虽然相比于 JVM 内部其他链路已经显得很简单了，但是这些方法分布在不同的源代码文件中，阅读时需要在不同的源代码文件中反复切换，因此下面通过将函数展开并将源码贴在一起（大家千万不要以为这是在浪费纸张，整本书里并没有几处会大肆粘贴源代码来占篇幅，实在是这里的代码太核心了），让大家更容易理清头绪：

```
//---------------begin klassKlass.cpp create_klass()--------------------------------
KlassHandle h_this_klass;
klassKlass o;
klassOop k = base_create_klass_oop(h_this_klass, header_size(), o.vtbl_value(), CHECK_NULL);

    //---------------begin klass.cpp base_create_klass_oop()-------------------------
    size = align_object_size(size);
     Klass*   kl = (Klass*) vtbl.allocate_permanent(klass, size, CHECK_NULL);

        //---------------begin klass.hpp allocate_permanent()-----------------------
        void* result = new(klass_klass, size, THREAD) thisKlass();

            //---------------begin klass.hpp allocate_permanent()------------------
            klassOop k =
                (klassOop)
CollectedHeap::permanent_obj_allocate_no_klass_install(klass,size, CHECK_NULL);


                //---------------begin collectedHeap.inline.hpp
permanent_obj_allocate_no_klass_install()--
                HeapWord* obj = common_permanent_mem_allocate_init(size, CHECK_NULL);

                    //---------------begin collectedHeap.inline.hpp
common_permanent_mem_allocate_init()--
                    // ① 为 klassOop 申请内存
                    HeapWord* obj = common_permanent_mem_allocate_noinit(size,
CHECK_NULL);
                    // ② klassOop 内存清零
                    init_obj(obj, size);
                    return obj;
                    //---------------end collectedHeap.inline.hpp
common_permanent_mem_allocate_init()----


                // ③ 初始化 klassOop._mark 标识
                post_allocation_setup_no_klass_install(klass, obj, size);

                    //---begin collectedHeap.inline.hpp
post_allocation_setup_no_klass_install()
```

```
                    oop obj = (oop)objPtr;
                    if (UseBiasedLocking && (klass() != NULL)) {
                         obj->set_mark(klass->prototype_header());
                    } else {
                         obj->set_mark(markOopDesc::prototype());
                    }
                    //---end collectedHeap.inline.hpp
post_allocation_setup_no_klass_install()

                    return (oop)obj;
                    //---------------end collectedHeap.inline.hpp
permanent_obj_allocate_no_klass_install()----


                    return k->klass_part();
                    //---------------end klass.hpp allocate_permanent()--------------------

          if (HAS_PENDING_EXCEPTION) return NULL;
          klassOop new_klass = ((Klass*) result)->as_klassOop();
          OrderAccess::storestore();
          // ④ 初始化 klassOop._metadata
          post_new_init_klass(klass_klass, new_klass, size);
          return result;
          //---------------end klass.hpp allocate_permanent()--------------------------


klassOop k  = kl->as_klassOop();

// 这里省略部分逻辑

// ⑤ 初始化 klass
kl->set_java_mirror(NULL);
kl->set_modifier_flags(0);
kl->set_layout_helper(Klass::_lh_neutral_value);
kl->set_name(NULL);
AccessFlags af;
af.set_flags(0);
kl->set_access_flags(af);
kl->set_subklass(NULL);
kl->set_next_sibling(NULL);
kl->set_alloc_count(0);
kl->set_alloc_size(0);

kl->set_prototype_header(markOopDesc::prototype());
kl->set_biased_lock_revocation_count(0);
kl->set_last_biased_lock_bulk_revocation_time(0);
```

```
    return k;
    //---------------end klass.cpp base_create_klass_oop()--------------------------

// ⑥ 自指
k->set_klass(k); // point to thyself
return k;
//---------------end klassKlass.cpp create_klass()-----------------------------------
```

上面就是整合之后的源代码，其实代码量并不是很大（当然里面还是省略了部分函数展开）。

这部分代码逻辑大体上可以划分为 6 个步骤，在上面的这段整合好的代码中分别使用①、②、③等这样的编号进行标注，这 6 个步骤如图 5.15 所示。

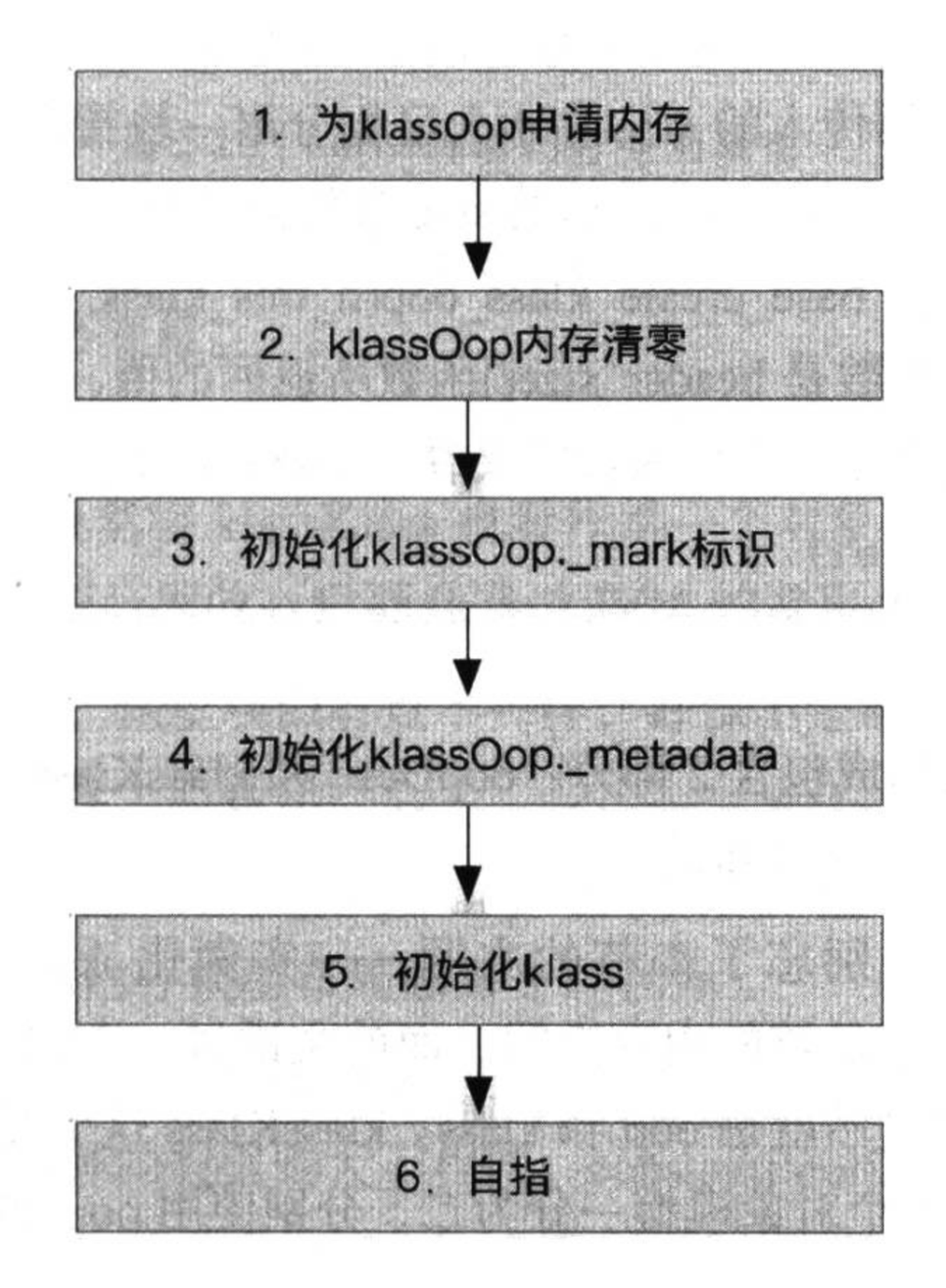

图 5.15　klassOop 实例化的 6 个步骤

下面将从这 6 个方面按照程序主线依次讲解。

5.3.2　为 klassOop 申请内存

从整合之后的源代码可以看出，从进入 klassKlass.cpp::create_klass()之后，就开始一路调用被层层封装起来的函数，除了函数调用之外并没有其他复杂的逻辑，一直调用到 CollectedHeap::permanent_obj_allocate_no_klass_install()，才终于消停，开始做正事了。函数名

不小心暴露了这件正事，顾名思义，这里开始在永久区为 obj 对象分配内存了。这个函数的实现逻辑是：

清单：/src/share/vm/gc_interface/collectedHeap.inline.hpp
作用：permanent_obj_allocate_no_klass_install()函数

```
//为 oop 申请内存
HeapWord* obj = common_permanent_mem_allocate_init(size, CHECK_NULL);

//初始化 oop 标识
post_allocation_setup_no_klass_install(klass, obj, size);
```

这个函数主要干了两件正事：内存申请和标识初始化。申请内存调用的是 common_permanent_mem_allocate_init()函数，这个函数在前文讲解 ConstantPoolOop 的内存初始化时详细讲解过，其主要功能就是根据传入的 size 在永久区划分出一块指定大小的内存区域。函数的实现不再赘述，但是 size 的大小需要关注。这里的 size 参数值的源头在 klassKlass.cpp::create_klass()函数中，该函数调用 base_create_klass_oop(h_this_klass, header_size(), o.vtbl_value(), CHECK_NULL)时，第 2 个参数是 header_size()函数所返回的值，要知道 size 参数的值，只需要看看 header_size()函数的实现即可。该函数定义在 klassKlass.hpp 文件中，定义如下：

```
static int header_size() {
    return oopDesc::header_size() + sizeof(klassKlass)/HeapWordSize;
}
```

这个定义表明，size 的构成包含 2 部分：oopDesc 和 klassKlass 的类型所占内存空间。这两个类型本身的大小在编译期间便可知道。

为什么会是这么大呢？别忘了本节的主题——实例化 klassKlassOop。既然要实例化 klassKlassOop，就得为其分配足够的内存空间，而前文讲过，JVM 内部通过“两模型三维度”去描述一个对象，两个模型自然就是 oop 和 klass。klassKlass 这个类实例在 JVM 内部也是被描述的对象，因此 JVM 也将这个对象模型一分为二，分别使用 oop 和 klass 这两个拆分的模型来描述。

当 common_permanent_mem_allocate_init()函数执行完之后，JVM 的永久区中便多了一个对象内存布局，该对象是 klassKlassOop，其内存布局模型如图 5.16 所示。

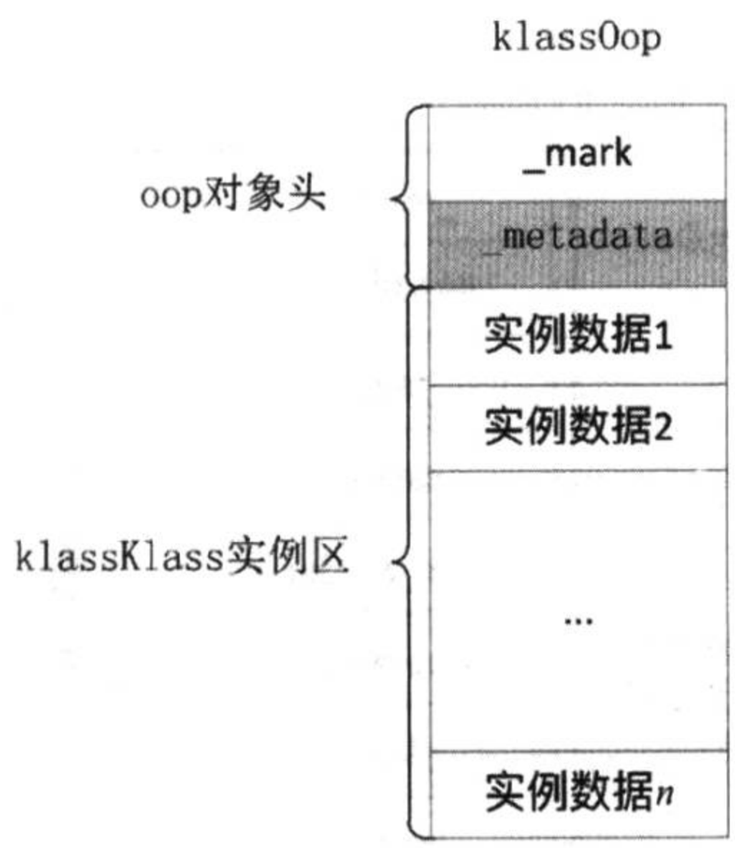

图 5.16　klassKlass 内存布局模型

图 5.16 中的 klassKlass 实例区其实就是 klassKlass 的实例对象所在的地方，该区域中的实例数据都是 klassKlass 类的成员变量。

虽然直到这里，并没有明确提出这块内存区域就隶属 klassKlassOop 这个对象，但是并不妨碍我们大胆猜测，因为 JVM 既然将 klassKlass 也看作是一个对象，那么 JVM 一定会将其包装成一个 oop，这样才方便垃圾统一回收。后面会通过 JVM 的源码来进行证实。

但是等等，等等，等一下，这里貌似有点不一样，好像不是那么回事，似乎哪里不一致。具体是哪里呢？你稍等，让我理下思路先。

前文讲过 JVM 的常量池 constantPoolOop 的内存分配，对于常量池对象，对于 constantPoolOop 对象，JVM 的内存模型布局如图 5.17 所示。

对比下 klassKlassOop 和 constantPoolOop 这两个对象的内存布局，其对象头都是一致的，但是紧跟在对象头后面的数据区的数据则大不相同。constantPoolOop 的数据区由 2 部分组成：constantPoolOop 自己的字段和长度等同于一个 Java 类编译后得到的常量池所有元素所占内存区域。而到了 klassKlassOop 对象，其数据区则直接变成了 klassKlass 类型实例。这里有一个点疑问：klassKlassOop 的对象头后面直接跟了 klass 模型实例，klass 模型实例一般用于描述对象的元数据，但是 oop 本身含有一个指针_metadata 指向这个元数据区域，那么 oop 的_metadata 指针指向哪里，还有用吗？或者难道是直接指向其自身？

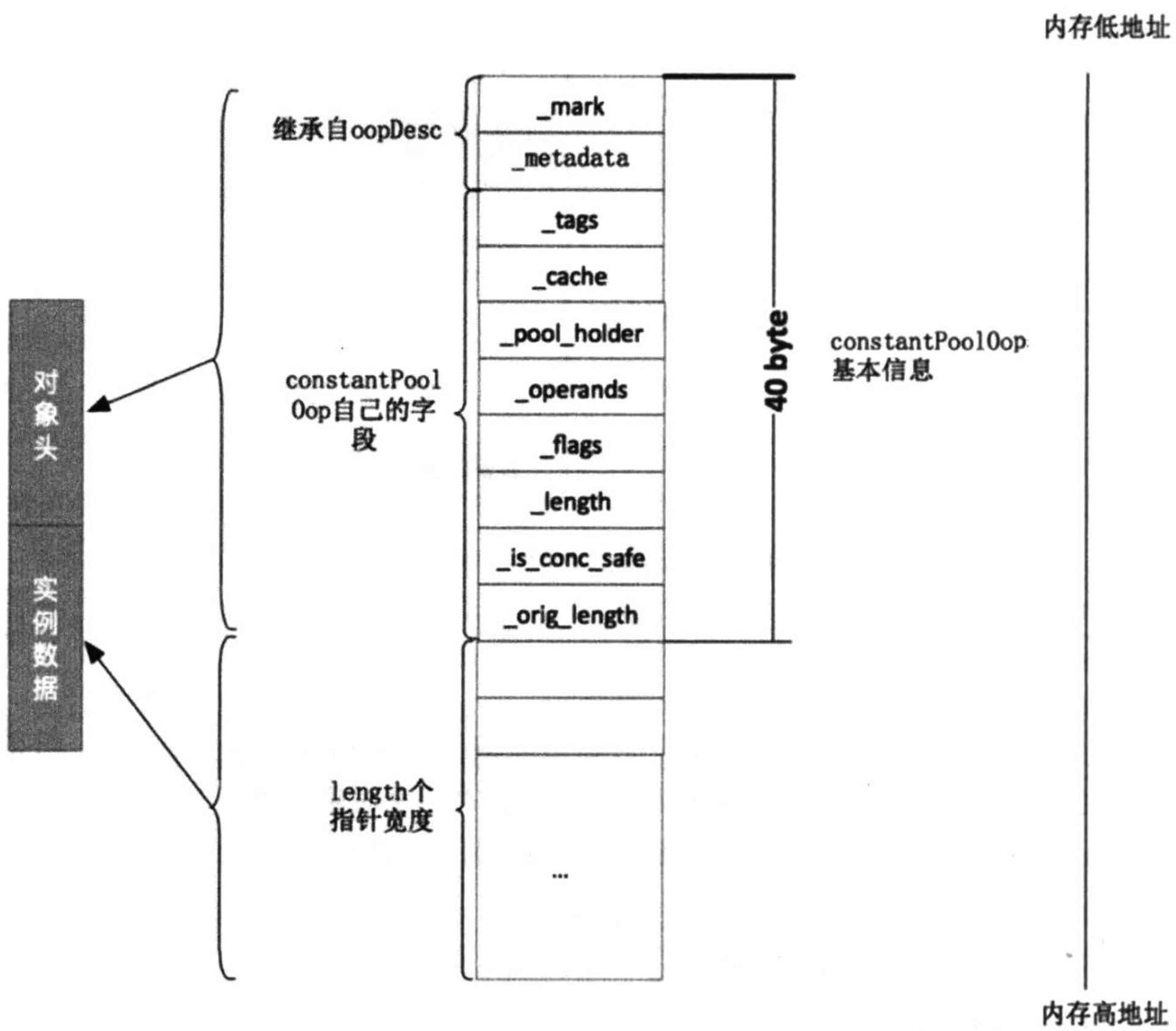

图 5.17　constantPoolOop 对象内存布局模型

如果你还不明白，可以这样思考，在 JVM 内部按照两模型三维度去描述一个对象，那么如果按照这种规范来描述 klassKlassOop，则 klassKlassOoo 的内存布局应该是如图 5.18 所示的样子。

在图 5.18 中，klassKlassOop._metadata 指向了一个专门的元数据区，这样 Java 类通过对类实例进行反射便能在运行期动态获取到类型的结构信息。但是很明显，真实的情况是，klassKlassOop 的类元信息直接跟在了 oop 对象头后面，oop 后面跟的并不是 oop 本身的实例数据。既然这样，那么_metadata 指针最终会指向哪里呢？要想知道答案，还得继续往下看代码。

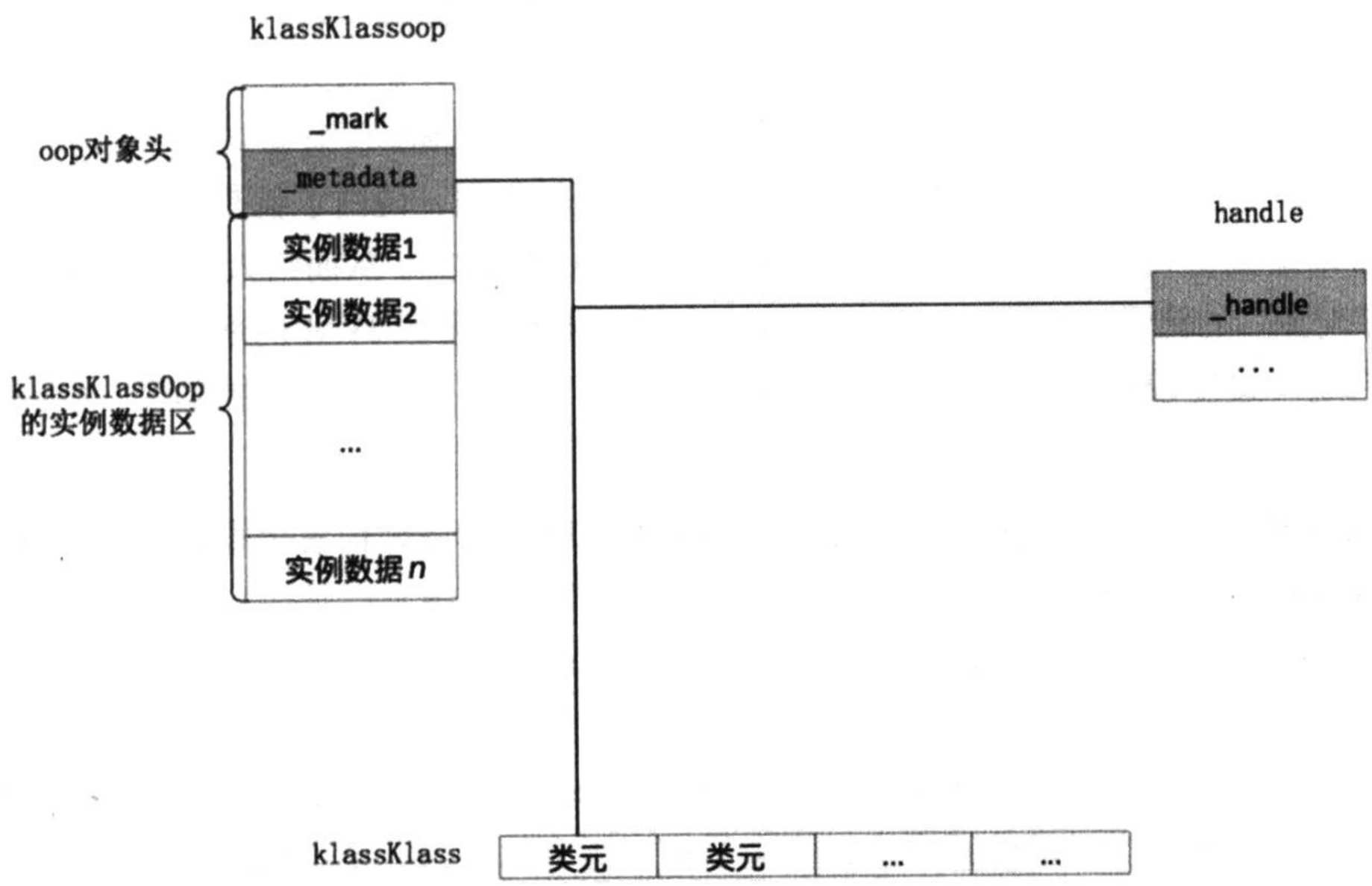

图 5.18　使用“三维度”模型描述 constantPoolOop

5.3.3　klassOop 内存清零

为 klassOop 分配完内存后，接下来开始将这段内存清零。清零调用函数 collectedHeap.init_obj()，该函数的实现思路在前文讲解 constantPoolOop 的内存清零时已经讲过。之所以要清零，是因为 JVM 的永久区是一个可重复使用的内存区域，会被 GC 反复回收，因此 JVM 为 oop 刚申请的内存区域里可能还保存着原来已经被清理的对象的数据。

5.3.4　初始化 mark

通过前面两个步骤，已经成功地为 klassOop 申请了内存并实现清零，那么接下来的逻辑自然是往内存里填充数据。oop 类里面一共包含两个成员变量：_mark 和_metadata，这里先填充_mark 变量。_mark 变量的填充主要通过调用 post_allocation_setup_no_klass_install()函数实现。在上面整合的源码中贴出了该函数的实现，其中主要调用了 obj->set_mark (markOopDesc::prototype())函数。

这里有必要对 markOopDesc 这个 C++类做一个简单的说明，因为该类在整个 JVM 的源代码中都显得十分特别，甚至可以说非常诡异（哈哈，笔者一开始研究的时候，每每看到这里就读不下去了，面对这个类有着说不清的无力感，诸多疑惑萦绕在心头）。之所以说其诡异，是因为其实 markOopDesc 类型在 JVM 内部就是起个标记的作用，说白了就是一个整型数字，但是其设计者却偏偏将其设计成一个类，并且还继承了 oopDesc 类，从而给人一种好像 markOop 也是一个“对象”的错觉，好像这个类纯粹就是为了迎合或者遵循 JVM 内部面向对象的设计风格的，说它是“过度设计”貌似也没有太冤枉它（哈哈，也可能是笔者眼拙脑笨，没能窥透这里面的道理）。

markOopDesc 的开发者倒也坦诚，在 markOopDesc 的注释信息里直接敞开天窗说亮话，markOopDesc 类注释的开头两句赫然写着：

```
Note that the mark is not a real oop but just a word.
It is placed in the oop hierarchy for historical reasons.
```

看来作者还是很直白的，直接说出了 markOopDesc 并不是一个真正的 oop 对象指针，之所以也将其放在 oop 的类继承体系之中，完全是由于历史的原因。至于到底是何历史原因，个人并没有继续深究，大家有兴趣可以去找找这方面的资料。

总之，这个类非常容易使人困惑，如果顺着 oop 这套模型去分析，反而会走上“邪道”，并且最终会走进死胡同。

markOopDesc 类除了从 oopDesc 类继承而来的两个成员变量，并没有自己的成员变量，只是里面定义了不少枚举。这些枚举类型本身并不占用内存空间。继续上面的话题，在为 klassOop 填充_mark 成员变量时，调用了 obj->set_mark(markOopDesc::prototype())函数，其入参是 markOopDesc::prototype()，该函数声明如下：

清单：/src/share/vm/oops/markOopDesc.hpp
作用：prototype()函数

```
static markOop prototype() {
    return markOop( no_hash_in_place | no_lock_in_place );
}
```

在 prototype()函数里貌似返回了 markOop 的构造函数，这个构造函数的入参是一个整型变量。先从 markOopDesc.hpp 文件中搜索这样的构造函数，结果很遗憾搜不到。由于 markOopDesc 继承了 oopDesc 类，但是 oopDesc 类中也没有定义任何一个类似的构造函数。事实上，这里并不是调用了 markOop 的构造函数，而是 C 语言的内建类型直接赋初值的写法。在 C 语言中，对于变量初始化，有一种快速赋初值的写法，例如：

```
int x=3;
```

可以直接写成：

```
int x(3);
```

同样，指针类型也是基本类型，自然也支持快速赋初值的写法，例如：

```
int x=3;
int *p=&x;
```

可以直接写成：

```
int x=3;
int *p(&x);
```

本来这种写法一般情况下是简单易懂的，即使不知道有这种写法的人看了这种写法之后，也能猜出个大概意思。但是当这种赋初值的写法与自定义类型结合起来时，就容易误导人了，例如这里的 markOop(no_hash_in_place | no_lock_in_place)，就容易使人误以为在调用 markOop 的构造函数。markOop 是一种自定义的数据类型，在 hierarchy.hpp 中的定义是：

```
typedef class markOopDesc* markOop;
```

由此可见，markOop 的实际类型是 markOopDesc*指针类型，而指针是一种基本的数据类型，因此自然支持赋初值的写法。下面举例说明：

清单：test.cpp
作用：演示指针赋初值

```
#include <stdio.h>

class A{
public:
    enum{
        a=1,
        b=2
    };
};

//自定义一种数据类型 AT
typedef class A* AT;

AT test(){
    //为指针赋初值
    return AT(3);
}

int main(){
    printf("yy=%p\n", test());
}
```

本例中，先定义类型 A，接着声明一种自定义的数据类型 AT，AT 的类型原型是 A*，其实是一种指针。在 test()函数中，采用赋初值的写法直接返回类型 AT 的变量。最终打印出来的结果是 3，因为 test()返回了一个指针，而指针里所存储的值就是 3。

其实 C++编译器在解释 test()函数时，会将语法最终展开为下面这种形式：

```
A* test(){
    //为指针赋初值
    A* a=(A*)3;
     return a;
    }
```

这里需要注意的是，由于指针类型变量本质上存储的也是整型数据（因为内存地址一定是整数），因此可以将一个整数直接强制转换成一个指针类型。

基于这里的例子，回头看 markOop::prototype()函数，一样的道理，该函数最终会在编译期展开为下面这种逻辑：

```
static markOopDesc* prototype() {
    markOopDesc* mark=(markOopDesc*)(no_hash_in_place | no_lock_in_place );
    return mark;
}
```

这样一来，prototype()函数的逻辑就再清晰不过了，该函数最终将返回一个指针，这个指针里存储了一个整数数字。由于 oop._mark 成员变量用于存储 JVM 内部对象的哈希值、线程锁等信息，而 prototype()函数所返回的 mark 标识则标记这个 oop 当前尚未建立唯一哈希值，并且也没有加任何锁。哈希值相对于 JVM，就类似于人的身份证之于社会，此时的这个 oop 对象就像一个刚出生的婴儿一样，尚依偎在母亲“JVM”的襁褓中，甚至在这个世界上连唯一的身份证都还没有。

JVM 内部每次读取 oop 的 mark 标识时，会调用 markOopDesc 的 value()函数，该函数定义如下：

```
uintptr_t value() const {
    return (uintptr_t) this;
}
```

value()函数直接返回 mark 指针的值，由此可见，markOopDesc 内部其实是将 this 指针当作它的值来使用的，而并没有将 markOop 指针还原成 markOopDesc 对象来使用。所以，markOop 虽然看起来是指针但其实并不是真的指向 markOopDesc 实例的指针，这正是 markOopDesc 这个类型神奇的地方。

如果你认为 oop 的 mark 标识是一个指向 markOopDesc 实例的指针，那么在分析 klassOop 时，你可能会认为 JVM 执行完 markOopDesc::prototype()函数，初始化完 klassOop._mark 成员变

量之后的内存布局如图 5.19 所示。

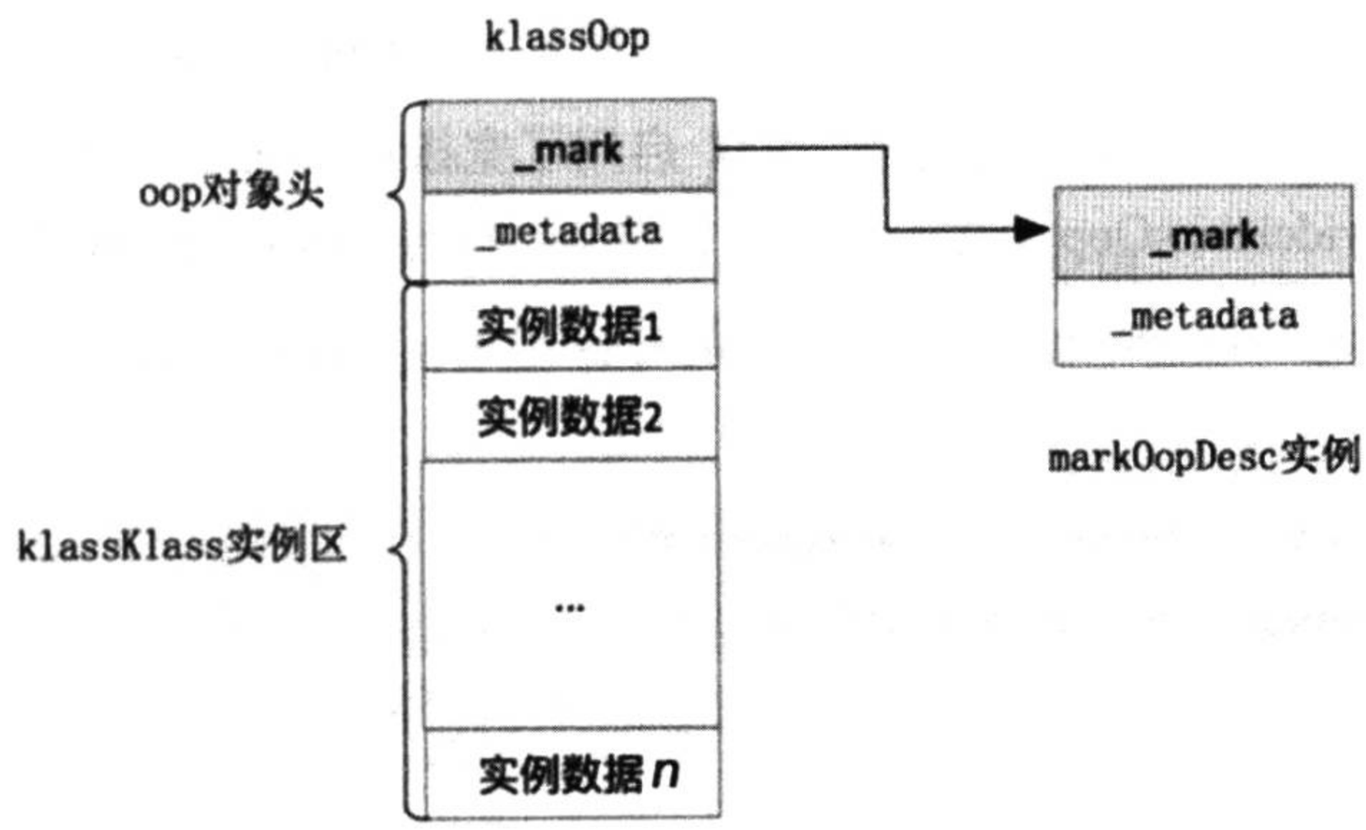

图 5.19　一种假设的内存布局图

如果真是这样就陷入了死胡同，因为 JVM 内部根本就没有在任何地方构建 markOopDesc 实例对象。

正是由于 markOop 指针其实并不是一个真正的 oopDesc 类型，仅仅是一个指针，并且其作用仅仅用于存储 JVM 内部对象的哈希值、锁状态标识等信息，因此 JVM 执行完 markOopDesc::prototype()函数之后，正确的内存布局应该如图 5.20 所示。

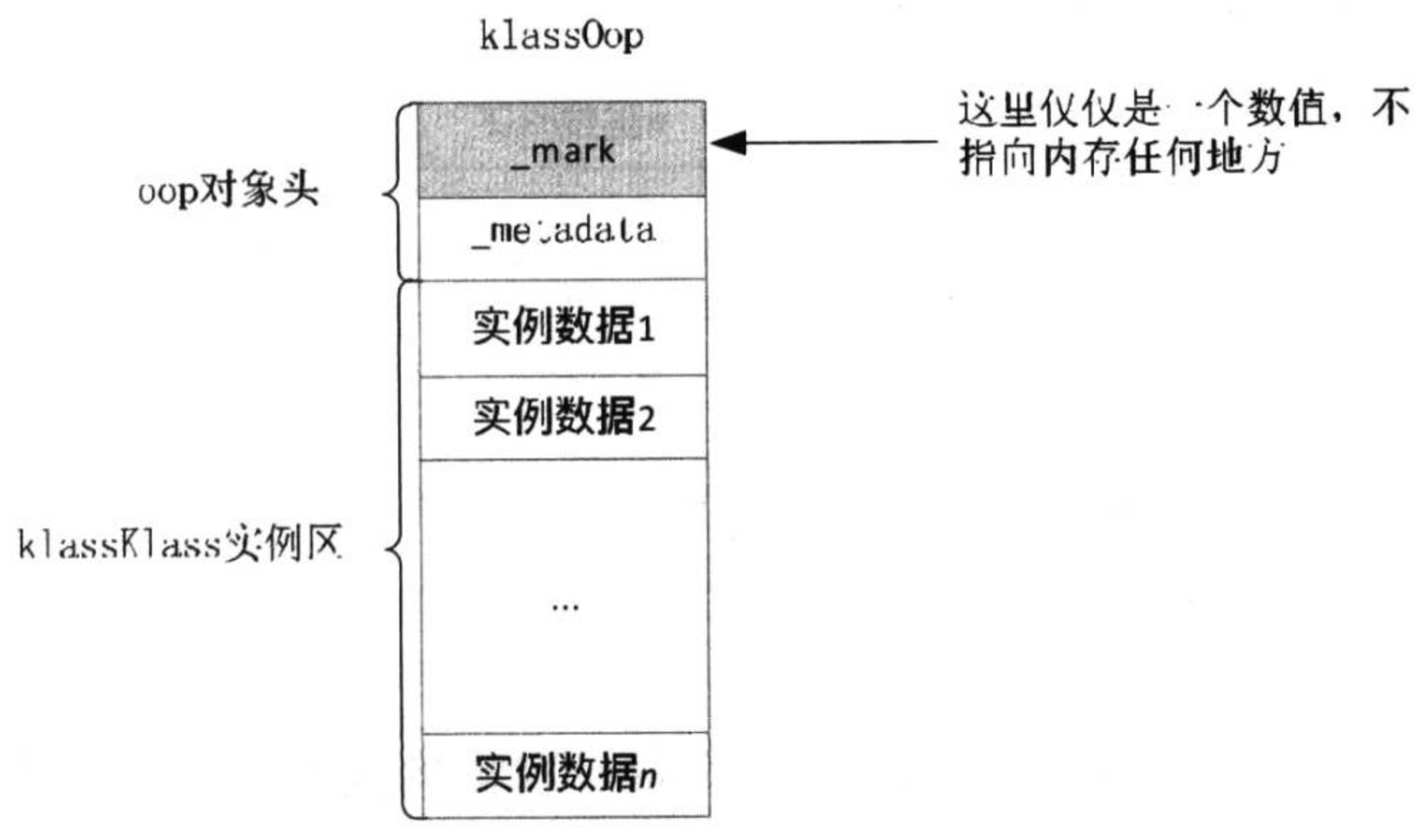

图 5.20　mark 字段的含义

5.3.5 初始化 klassOop._metadata

初始化完 klassOop._mark 标识后，接着开始初始化 klassOop._metadata 成员。

oop 的_metadata 是一个联合体，当 JVM 启用了宽指针选项后，_metadata 的类型便是 wideKlassOop，而 wideKlassOop 仍然是一个指针类型，指向 klassOopDesc 类型实例。

在 klass.hpp 的 allocate_permanent()函数中调用 post_new_init_klass()函数，而该函数最终调用的方法如下：

清单：/src/share/vm/gc_interface/collectedHeap.inline.hpp
作用：post_allocation_install_obj_klass()函数

```
void CollectedHeap::post_allocation_install_obj_klass(KlassHandle klass,
                                                      oop obj,
                                                      int size) {
    obj->set_klass(klass());
}
```

该函数通过调用obj->set_klass()来完成_metadata成员变量的赋值，该函数的入参是klass()，这并不是在调用 KlassHandle 的构造函数，因为在 Handle 类中，()操作符被重载，因此这里的 klass()其实是在调用普通函数。在 Handle 类中有如下定义：

清单：/src/share/vm/runtime/handles.hpp
作用：post_allocation_install_obj_klass()函数

```
class Handle VALUE_OBJ_CLASS_SPEC {
 private:
  oop* _handle;

 protected:
  oop     obj() const  { return _handle == NULL ? (oop)NULL : *_handle; }
  //…


  oop     operator () () const  { return obj(); }
  //…
};
```

在 Handle 类里面定义了 oop operator()() const 这样的操作符重载函数，很显然，在 CollectedHeap::post_allocation_install_obj_klass()函数中调用obj->set_klass(klass())时，入参klass()最终调用了 Handle 类的 obj()函数，而该函数直接返回 Handle 类里面的_handle 变量。

于是问题变成了分析_handle 指针到底指向哪里，这需要从源头上分析。将目光重新回到整条链路的源头——klassKlass::create_klass()函数。CollectedHeap::post_allocation_install_obj_klass()

函数中所调用的 obj->set_klass(klass())的入参的 klass 变量便是在该函数中定义的，定义方式是：KlassHandle h_this_klass，未使用 new 关键字，因此这会调用 Handle 类的默认构造函数，而 Handle 类的默认构造函数定义如下：

```
Handle() { _handle = NULL; }
```

在这个默认构造函数里面，将成员变量_handle 设置成了空值。

在 klassKlass::create_klass()函数中定义了 KlassHandle 类型变量后，便一路透传，直到透传到 CollectedHeap::post_allocation_install_obj_klass()函数中的 obj->set_klass(klass())函数的入参，并且在整个过程中，KlassHandle 类型变量都未做任何修改，因此其内部的_handle 成员变量一直都是空值，于是在 obj->set_klass(klass())中调用 klass()后所返回的值便也是空值。所以，当这一步执行完之后，klassOop 类实例的内存空间布局如图 5.21 所示。

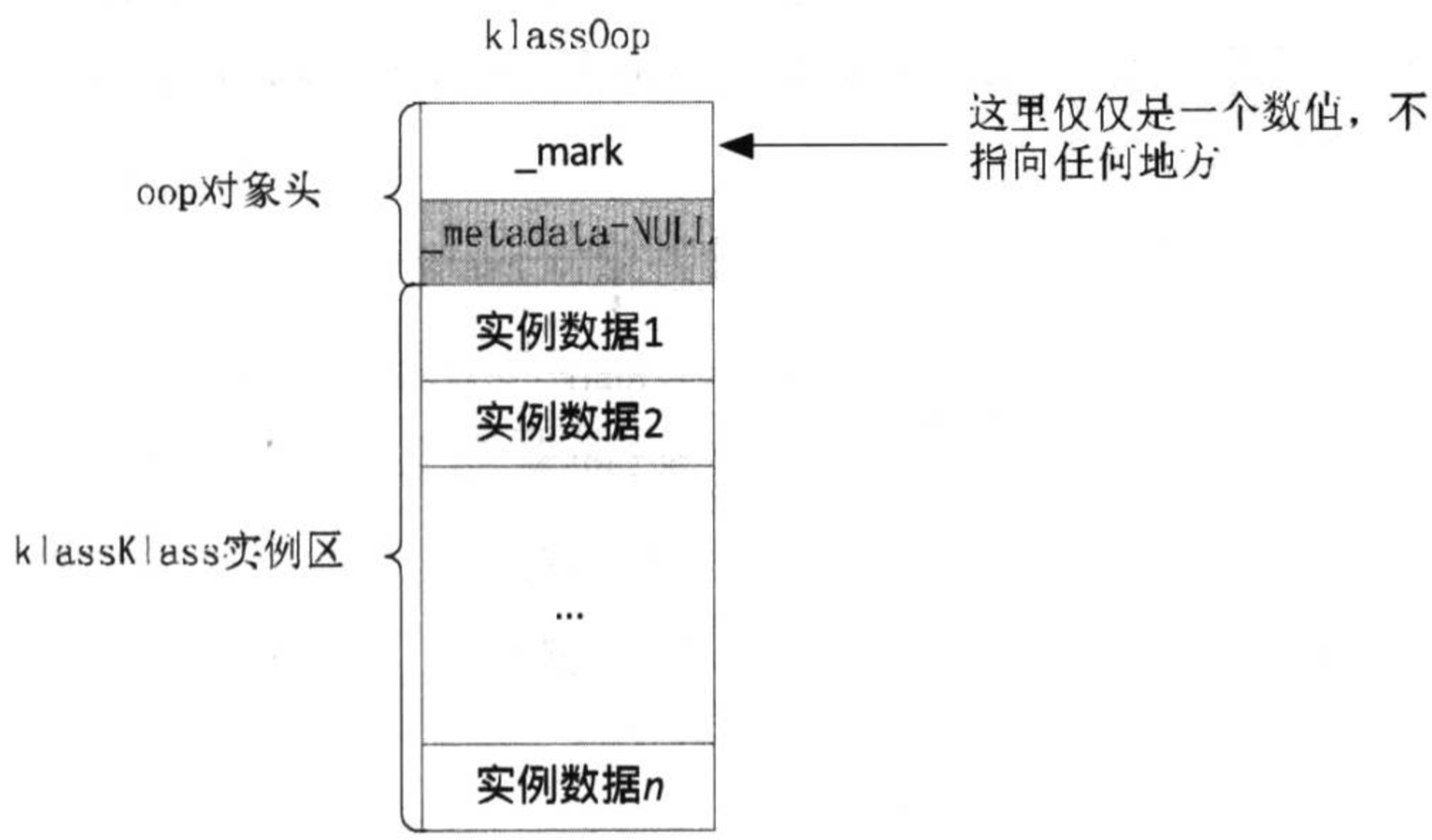

图 5.21　执行完 post_new_init_klass()函数之后的 klassOop 内存布局

5.3.6　初始化 klass

经过前面几个步骤，klassOop 对象实例已经成功完成 oop 对象头的初始化。由于 klassOop 也是一个 oop，因此对象头的内存后面也会接一段数据区，并且这段数据区正是 klassKlass 类型实例存放地。因此完成 oop 对象头的初始化之后，接着便开始初始化 klassKlass 类型实例。

在 Klass::base_create_klass_oop()函数中，执行完 Klass* kl = (Klass*) vtbl.allocate_permanent (klass, size, CHECK_NULL) 函数之后，便得到 klass 对象实例的指针，Klass::base_create_klass_oop()函数中后续的逻辑便都是在处理 klass 对象实例的初始化，具体代码在上面的整合代码中已经贴出，可以看到此时大部分属性都被赋为空值了。注意这里也调用

了 kl->set_prototype_header(markOopDesc::prototype())函数来设置对象标，这说明 JVM 内部虽然将 klass 也封装成了 oop，但是 klass 毕竟是独立的一种类型，因此也需要记录线程锁等相关标识。

5.3.7 自指

当执行完 Klass::base_create_klass_oop()函数之后，CPU 回到了 klassKlass::create_klass()函数中，开始执行 k->set_klass(k)这个逻辑。这里的 k 是 klassOop，是 klassOopDesc*类型的指针。

k->set_klass(k)其实是在设置 klassOopDesc 类型实例的_metadata 成员变量，虽然在前面的步骤中，_metadata 成员变量已经被设置过一回，但是当时被设置成了 NULL 空值，这一次再次设置，则是专门为 klassOop 量身定做了。在本步骤中，klassOop 的_metadata 成员变量被设置为指向其自身。为什么要指向自己呢？这个问题暂时不回答，等后面讲完了 constantPoolOop 的完整内存布局后再来讲。这里先看下 Klass::base_create_klass_oop()函数执行完之后的内存布局，如图 5.22 所示。

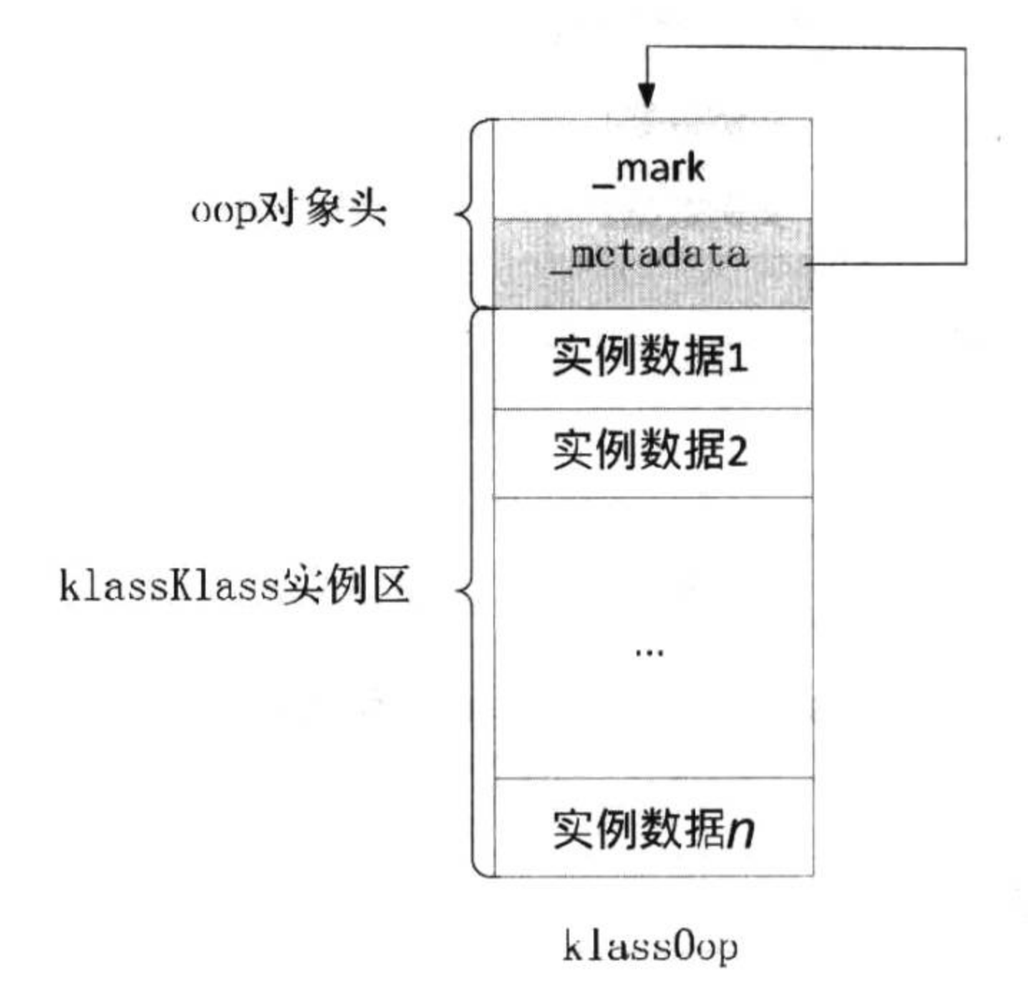

图 5.22 执行完 Klass::base_create_klass_oop()函数之后的 klassOop 内存布局

至此，klassOop 的实例化便完成了，最终得到的 klassOop 的内存模型便是如图 5.22 所示的样子。虽然最终只得到了 oop，但是其实内存里是有两个类型实例的，一个是 oop 对象头实例，紧接其后的则是 klass 类型实例。通过 oop 对象头的内存位置可以得到 klass 类型实例的首地址，进行转换后拿到 klass 实例。在这整个过程中，笔者想讲解一下这里面所使用到的一些 C++特性技巧。

5.4　常量池 klass 模型（2）

5.4.1　constantPoolKlass 模型构建

上文分析了 klassKlass 对象实例构建的原理，而且提到，之所以要分析 klassKlass 的构建，是因为这是研究常量池 constantPoolKlass 构建的基础，因为在 JVM 启动过程中，JVM 需要执行 constantPoolKlass::create_klass()函数来构建 constantPoolKlass 实例，而在该函数中调用了 KlassHandle h_this_klass(THREAD, Universe::klassKlassObj())函数来获取 klassKlass 所对应的 handle，JVM 以此为基础来构建常量池所对应的 klass。

在 constantPoolKlass::create_klass() 函数中执行完 KlassHandle h_this_klass(THREAD, Universe::klassKlassObj())函数之后，便开始执行 base_create_klass(h_this_klass, header_size(), o.vtbl_value(), CHECK_NULL)函数，由于 constantPoolKlass 继承自 klass，因此这里的 base_create_klass()函数消息最终也由 klass 对象接收和执行。整条链路时序图如图 5.23 所示。

此图与上文 klassKlass 实例构建的时序图基本相同，通过调用 klass::base_create_klass_oop() 入口进入，之后分别经过下面 5 个步骤：

（1）内存申请

（2）内存清零

（3）初始化_mark 成员变量

（4）初始化_metadata 成员变量

（5）初始化 klass

最终返回包装好的 klassOop。

由于 constantPoolKlass 实例构建的过程与 klassKlass 实例构建的过程完全一致，因此这里不再进行详细讲解，只需要关注最后的结果即可。很显然，JVM 最终在永久区所创建的对象是 constantPoolKlass，但是由于每一个 klass 最终都会被包装成 oop，因此最终在内存中所构建的模型如图 5.24 所示。

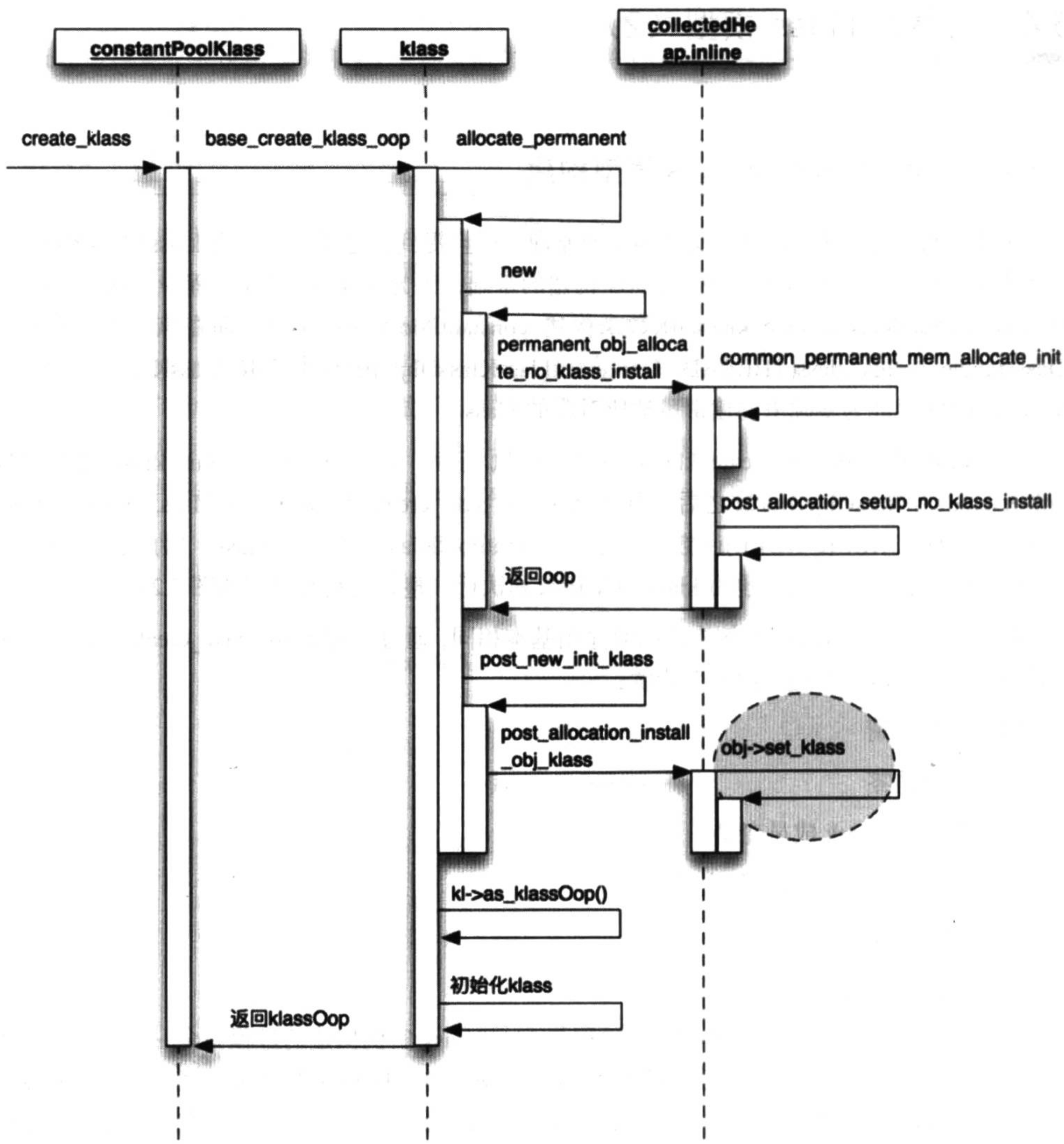

图 5.23　constantPoolKlass 实例构建时序图

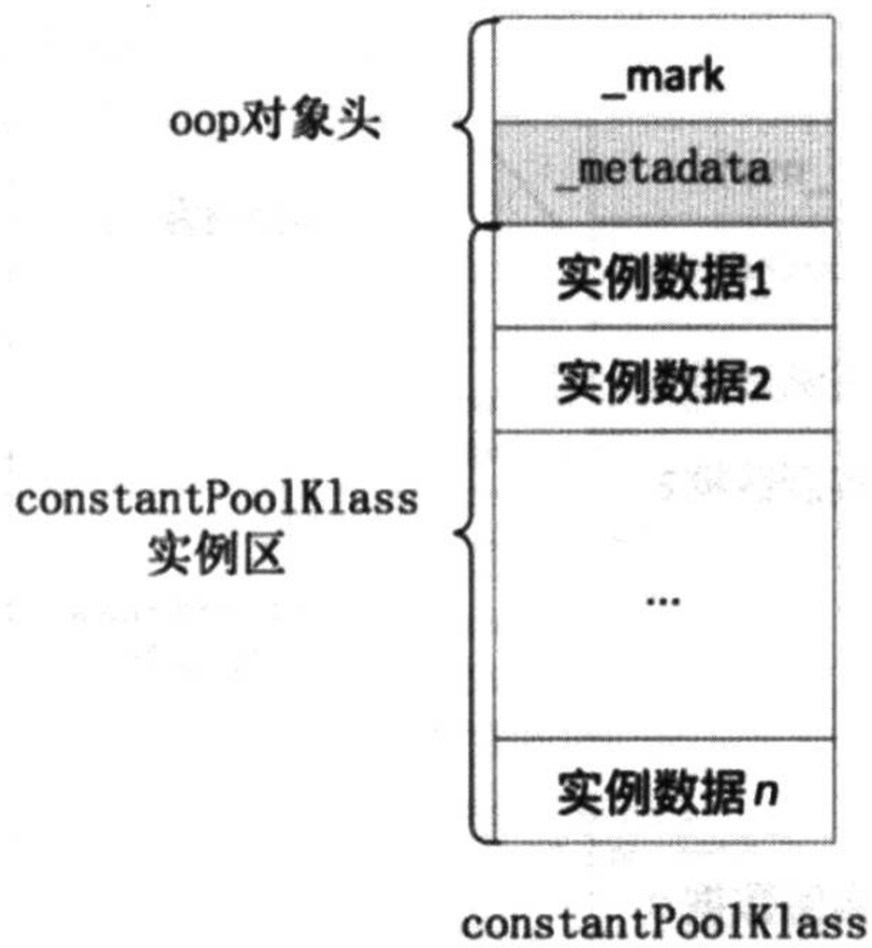

图 5.24　constantPoolKlass 内存布局模型

当然，为了构建这么一个模型，从一开始就要准确计算出其所占内存大小，计算出的大小将通过 klass::base_create_klass_oop()函数的 size 入参传递给后续流程。在构建 constantPoolKlass 时，这个大小由 constantPoolKlass::header_size()函数所确定：

```
static int header_size() {
    return oopDesc::header_size() + sizeof(constantPoolKlass)/HeapWordSize;
}
```

由这个逻辑可知，JVM 在构建 constantPoolKlass 时，所分配的内存大小为 oopDesc 的大小与 constantPoolKlass 的大小之和，即 JVM 将封装 constantPoolKlass 的 oop 的大小已经计算进去了。

JVM 构建 constantPoolKlass 的逻辑与构建 klassKlass 的逻辑基本是一致的，但入参 size 不同，且在调用 klass::base_create_klass()函数时所传入的 KlassHandle 入参也不相同，KlassHandle 入参的不同将决定最终所构建出来的 klassOop 的_metadata 参数不同。在 klass::base_create_klass()入口后面的链路中，有一步调用了 obj->set_klass()，这一步便是在设置 oop 的_metadata 成员变量。前面分析过，在构建 klassKlass 时，所传入的 KlassHandle 其实是 NULL，因此在 klassKlass::create_klass()中又调用了 k->set_klass(k)，最终将 klassOop 的_metadata 指向了自己。但是在构建 constantPoolKlass 时，所传入的 KlassHandle 则是封装了 klassKlass 的 handle，因此最终所构建出来的 constantPoolKlass 所对应的 oop 的_metadata 便指向前面所构建的 klass。最终 constantPoolKlass 内存布局如图 5.25 所示。

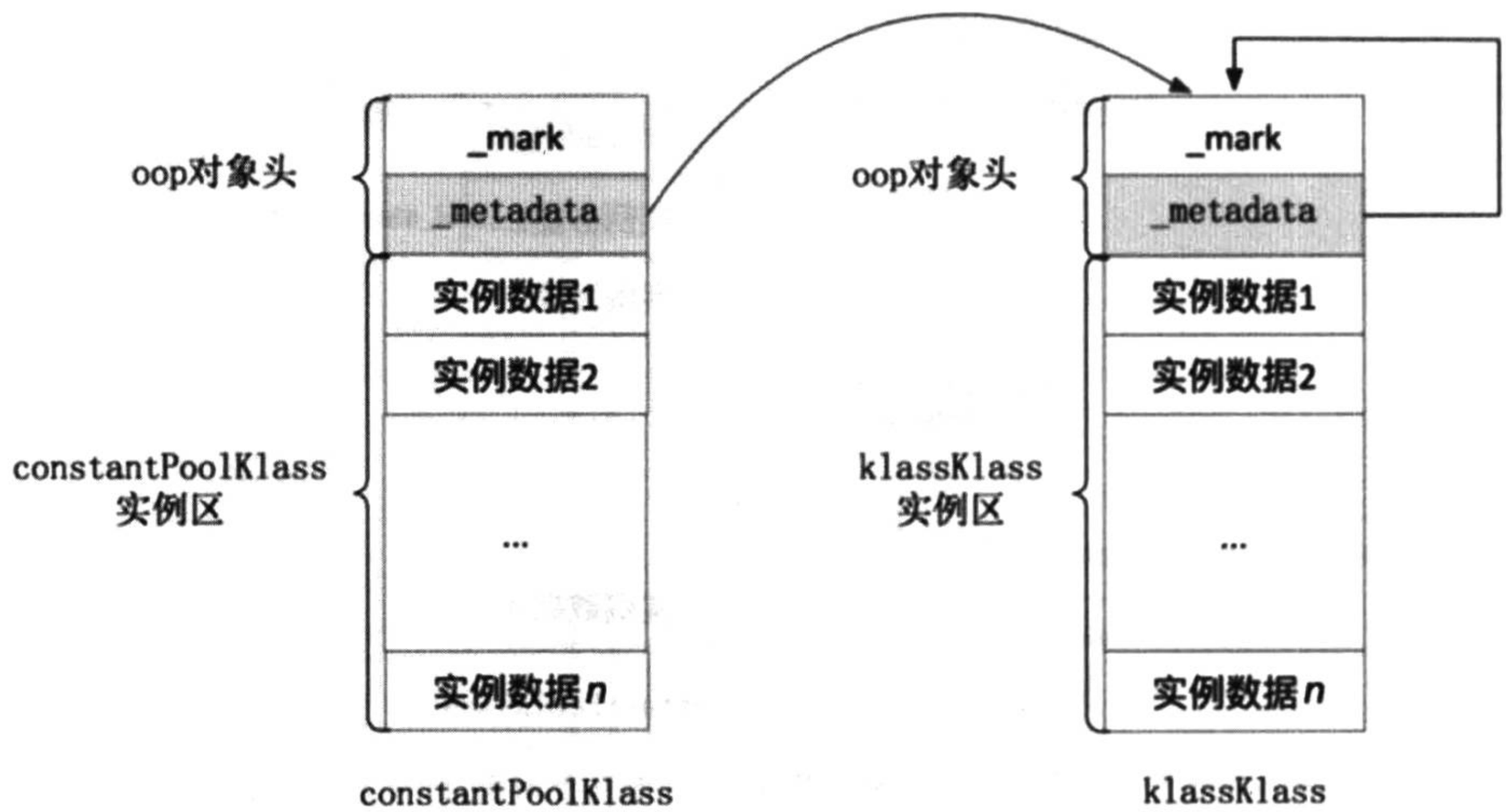

图 5.25　constantPoolKlass 内存布局

5.4.2　constantPoolOop 与 klass

刚刚分析了最终所构建出来的常量池 constantPoolKlass 的内存模型，其实 constantPoolKlass 正是 constantPoolOop 的类元信息，即 constantPoolOop 的 _metadata 指针应该指向 constantPoolKlass 对象实例。

而事实上，当 constantPoolOop 实例在构建过程中时，JVM 的确进行了这一步操作，这一步便是在图 5.26 中被椭圆框所框住的一步——post_allocation_install_obj_klass()。

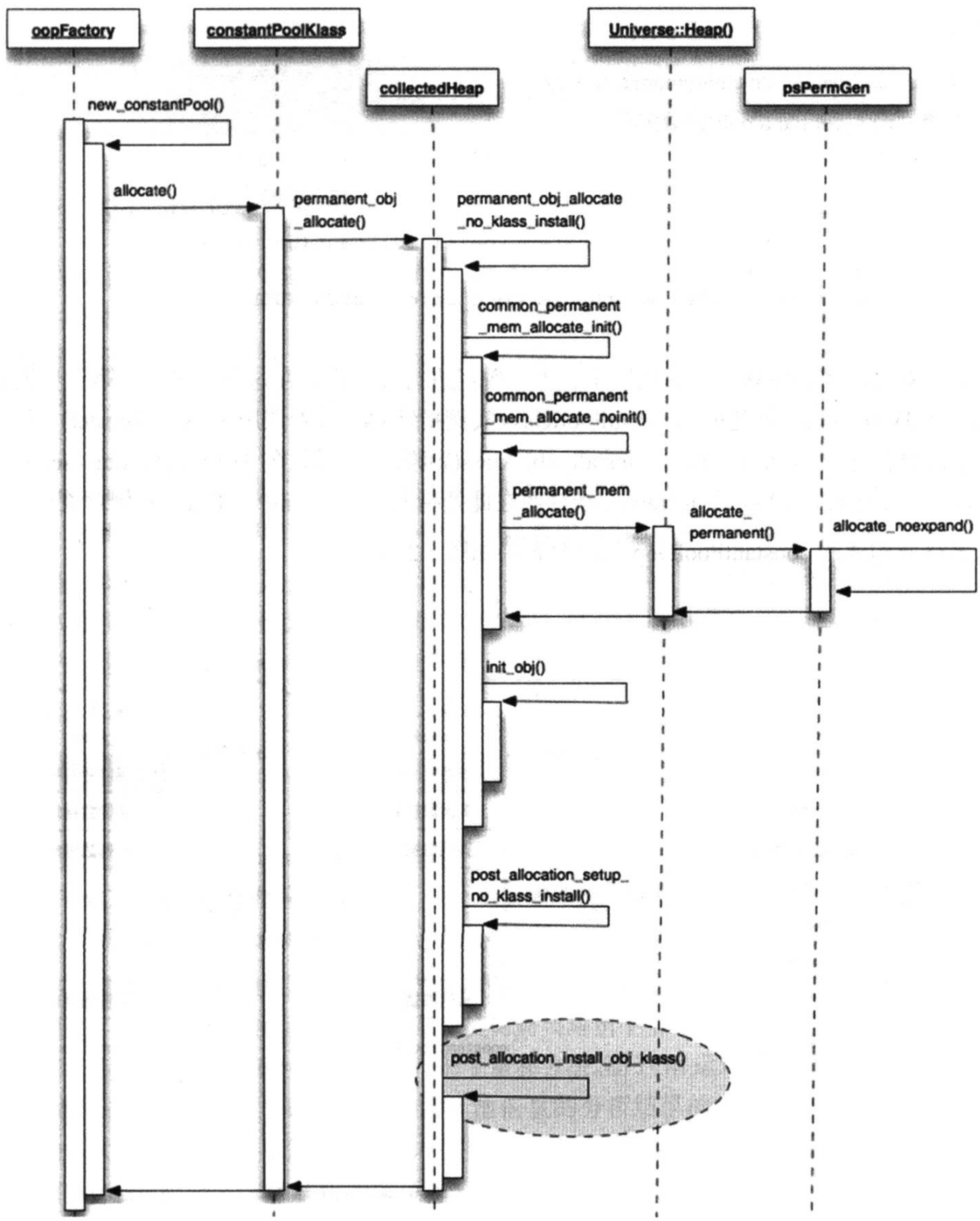

图 5.26　constantPoolOop 实例构建过程中设置_metadata

如果对前面构建 klassKlass 和构建 constantOopKlass 的过程很熟悉的话，应该清楚这一步实际上调用了 obj->set_klass()这个函数，而该函数会将_metadata 指针指向其对应的 klass 实例。JVM

创建 constantPoolOop 的过程中执行了如下逻辑：

清单：/src/share/vm/memory/oopFactory.cpp
功能：new_constantPool()函数定义

```
constantPoolOop oopFactory::new_constantPool(int length,
                                             bool is_conc_safe,
                                             TRAPS) {
    constantPoolKlass* ck = constantPoolKlass::cast(Universe::
constantPoolKlassObj());
    return ck->allocate(length, is_conc_safe, CHECK_NULL);
  }
```

大家对这段逻辑应该不会感到陌生，上文曾经进行过引用。在这段逻辑中，第一步先是获取到了随 JVM 启动而创建的 constantPoolKlass 实例指针 ck，接着调用了 ck->allocate()方法。ck 指针最终被传递到 post_allocation_install_obj_klass()函数中，因此在 JVM 执行 obj->set_klass()方法时，constantPoolOop 的_metadata 指针便在此处指向了 constantPoolKlass 实例对象。

最终所构建的 constantPoolOop 的内存布局如图 5.27 所示。

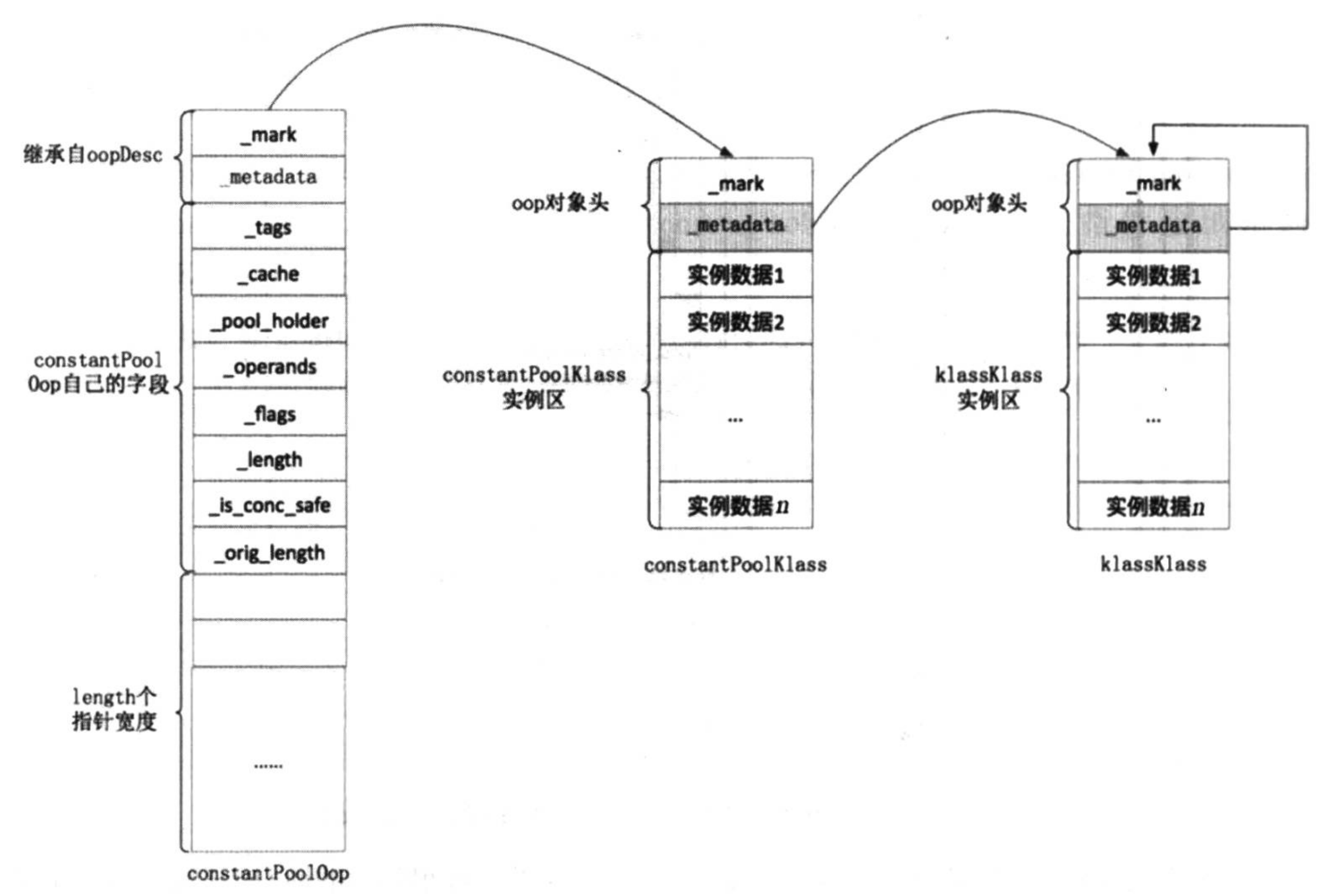

图 5.27　constantPoolOop 与 constantPoolKlass

至此，constantPoolOop 实例的构建便结束了。建议小伙伴们将这段逻辑熟记于心，因为 JVM 内部其他 oop 对象实例的构建都大同小异，明白 constantPoolOop 的构建原理，便意味着明白了所有。

5.4.3　klassKlass 终结符

在 JVM 构建 constantPoolOop 的过程中，由于 constantPoolOop 本身的大小不固定，这种不确定性主要体现在“length 个指针宽度”这块区域，因为不同的 Java class 被编译后，常量池元素的数量会不同。因此 JVM 需要使用 constantPoolKlass 来描述这些不固定的信息，这样最终 GC 在回收垃圾时才能准确地知道到底要回收多大内存空间。这便是 constantPoolKlass 的意义所在。

而 constantPoolKlass 的实例大小也是不确定的，因此 constantPoolKlass 本身也需要其他 klass 来描述，这便是 JVM 在构建 constantPoolOop 的过程中会引用 klassKlass 的原因。

但是，如果 klassKlass 又去使用一个 klass 来描述自身，那么 klass 链路将会无限引用下去，因此 JVM 便将 klassKlass 作为整个引用链的终结符，到这里就结束了，并且让 klassKlass 自己指向自己。这便是 klassKlass 实例为何最终要“自指”的原因。

5.5　常量池解析

前面详细分析了 constantPoolOop 的内存分配和 klass 模型构建，现在 JVM 完成了 constantPoolOop 全部内存布局的大手笔，接下来要做的便是往里面填充实际数据。在这里不妨回到初心，问问 JVM 构建 constantPoolOop 的意义所在。

Java 类源代码被编译成字节码，字节码中使用常量池来描述 Java 类中的结构信息——包括 Java 类中的一切变量和方法。JVM 加载某个类时需要解析字节码文件中的常量池信息，从字节码文件中还原出 Java 源代码中所定义的全部变量和方法。而 JVM 运行时的对象 constantPoolOop 便是用来保存 JVM 对字节码文件中的常量池信息的分析结果的。因此 JVM 需要先构建出 constantPoolOop 的实例，并分配足够的内存空间来保存字节码文件中的常量池信息。从本节开始，我们将会详细讲解 JVM 解析和保存常量池字节码的过程。

5.5.1 constantPoolOop 域初始化

在构建 constantPoolOop 的过程中，会执行 constantPoolKlass::allocate()函数，该函数主要干了 3 件事：

◎ 构建 constantPoolOop 对象实例。
◎ 初始化 constantPoolOop 实例域变量。
◎ 初始化 tag。

本节之前都在讲第一件事，这里讲第二件事。先看源码：

清单：/src/share/vm/oops/constantPoolKlass.cpp
功能：constantPoolKlass 域变量初始化

```
constantPoolOop constantPoolKlass::allocate(int length, bool is_conc_safe,
TRAPS) {
  //...

  pool->set_length(length);
  pool->set_tags(NULL);
  pool->set_cache(NULL);
  pool->set_operands(NULL);
  pool->set_pool_holder(NULL);
  pool->set_flags(0);
  pool->set_orig_length(0);
  pool->set_is_conc_safe(is_conc_safe);
 //…
}
```

这部分逻辑执行完之后，constantPoolOop 的内存布局如图 5.28 所示。

注意：此时 constantPoolOop 域的_tags 是 NULL，但是这个域变量对于 constantPoolOop 而言其实是十分重要的一个数据载体，_tags 实际上将会存放字节码常量池中的全部元素的标记。因此，下一步 JVM 便会对_tags 进行一系列处理。

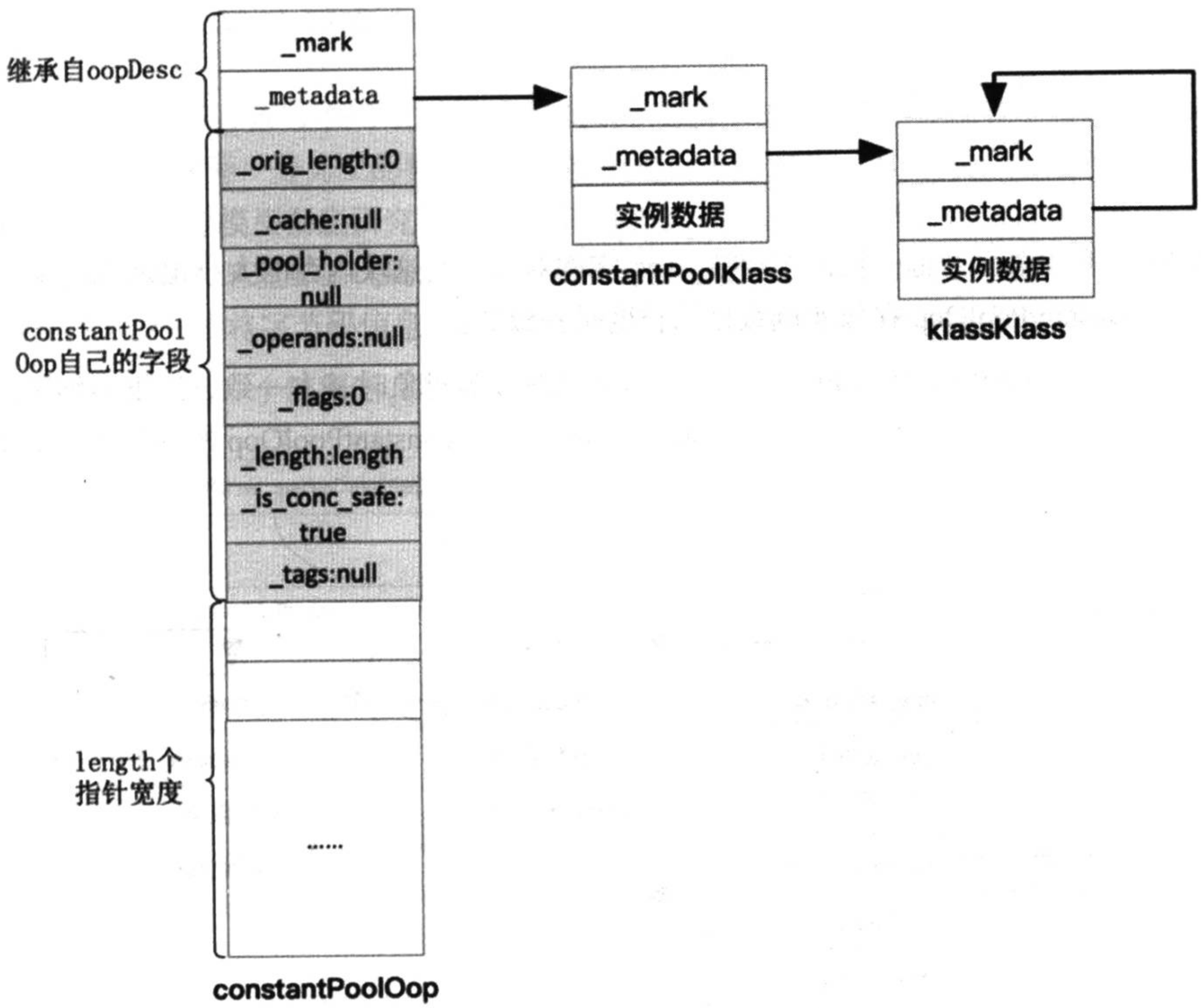

图 5.28　初始化 constantPoolOop 实例域变量

5.5.2　初始化 tag

在创建 constantPoolOop 的过程中，会为其_tags 域申请内存空间，这段逻辑还是在 constantPoolKlass::allocate()函数中实现，具体逻辑如下：

清单：/src/share/vm/oops/constantPoolKlass.cpp

功能：constantPoolKlass 域变量初始化

```
constantPoolOop constantPoolKlass::allocate(int length, bool is_conc_safe,
TRAPS) {
    //…

    typeArrayOop t_oop = oopFactory::new_permanent_byteArray(length, CHECK_NULL);
    typeArrayHandle tags (THREAD, t_oop);
    for (int index = 0; index < length; index++) {
```

```
    tags()->byte_at_put(index, JVM_CONSTANT_Invalid);
  }
  pool->set_tags(tags());

 //…
}
```

JVM 会在永久区开辟一块内存存放_tags 实例对象，而这块内存的大小也是 length 个指针宽度，与 constantPoolOop 存放实例数据的长度是一致的。

构建其内存布局的具体逻辑与前面构建 klassKlass 等对象时基本一致，这里不细讲，有兴趣的小伙伴们可以自行分析研究。完成_tags 的构建之后，constantPoolOop 的内存布局如图 5.29 所示。

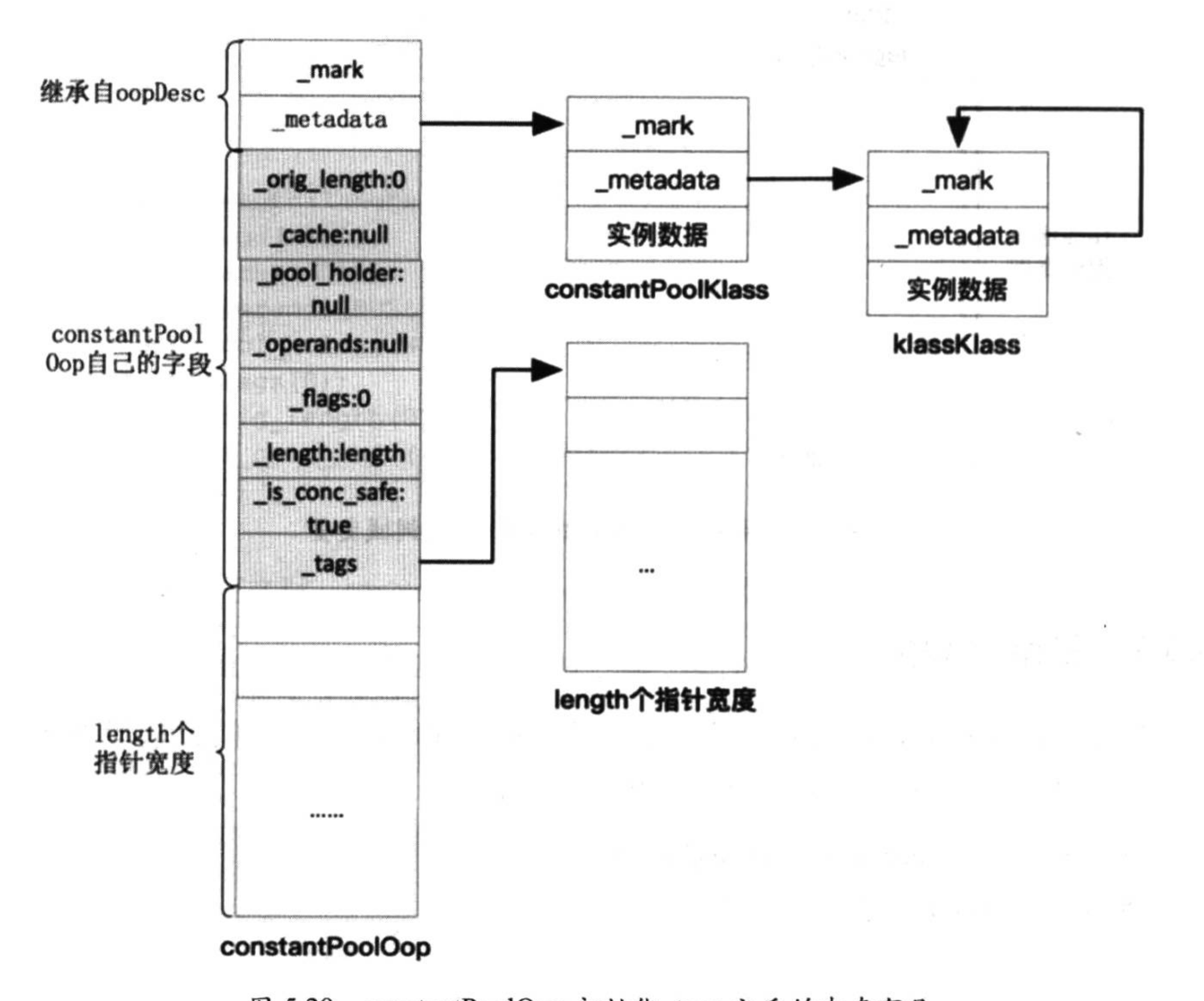

图 5.29　constantPoolOop 初始化_tags 之后的内存布局

由于_tags 所指向的实例对象刚刚构建，因此此时内存中没有实际数据。接下来 JVM 开始解析字节码文件的常量池元素，并逐个填充到这块内存区域。

5.5.3　解析常量池元素

前面经过“千辛万苦”，终于完成了常量池 oop 的基本构建工作。之前花了很多篇幅讲解常量池的 oop-klass 这种一分为二的模型，到这里便可以告一段落。不过付出这个代价是值得的，将常量池 oop 的内存模型彻底研究清楚，便基本扫清了 JVM 内存分配的大部分障碍，之后我们可以更快速、更深入地理解源代码。接下来需要将目光重新投向 JVM 类加载的“舞台”，研究 JVM 是如何一步一步将字节码信息翻译成可以被物理机器识别的动态的数据结构的。

在讲解之前先看一个链路图，如图 5.30 所示。

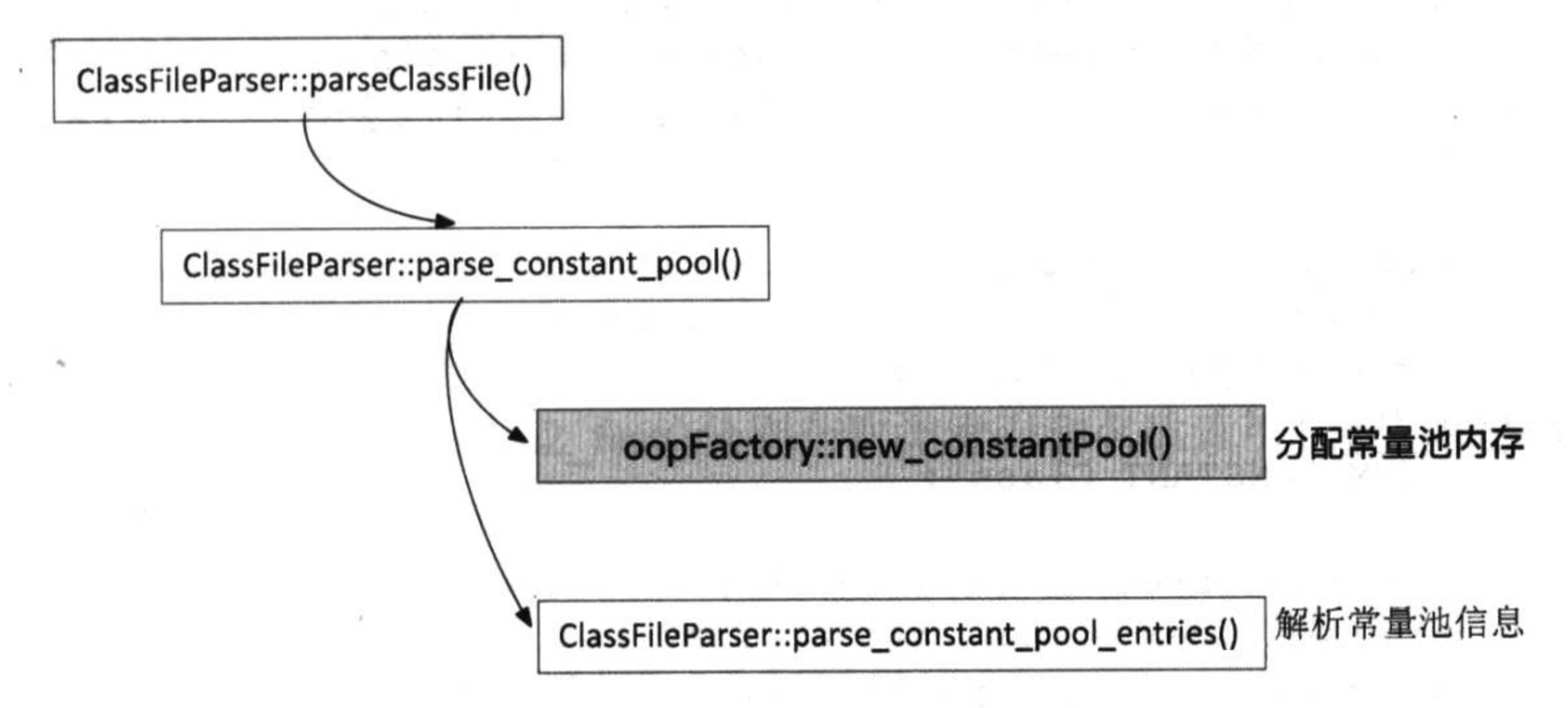

图 5.30　构建 constantPoolOop 的链路位置

在 classFileParser 中通过执行语句 constantPoolOop constant_pool = oopFactory::new_constantPool()完成 constantPoolOop 的构建，前面都是在分析这部分逻辑。

接下来便开始解析常量池元素，为此 classFileParse 专门定义了一个函数：parse_constant_pool_entries()，其在链路图中的位置如图 5.31 所示。

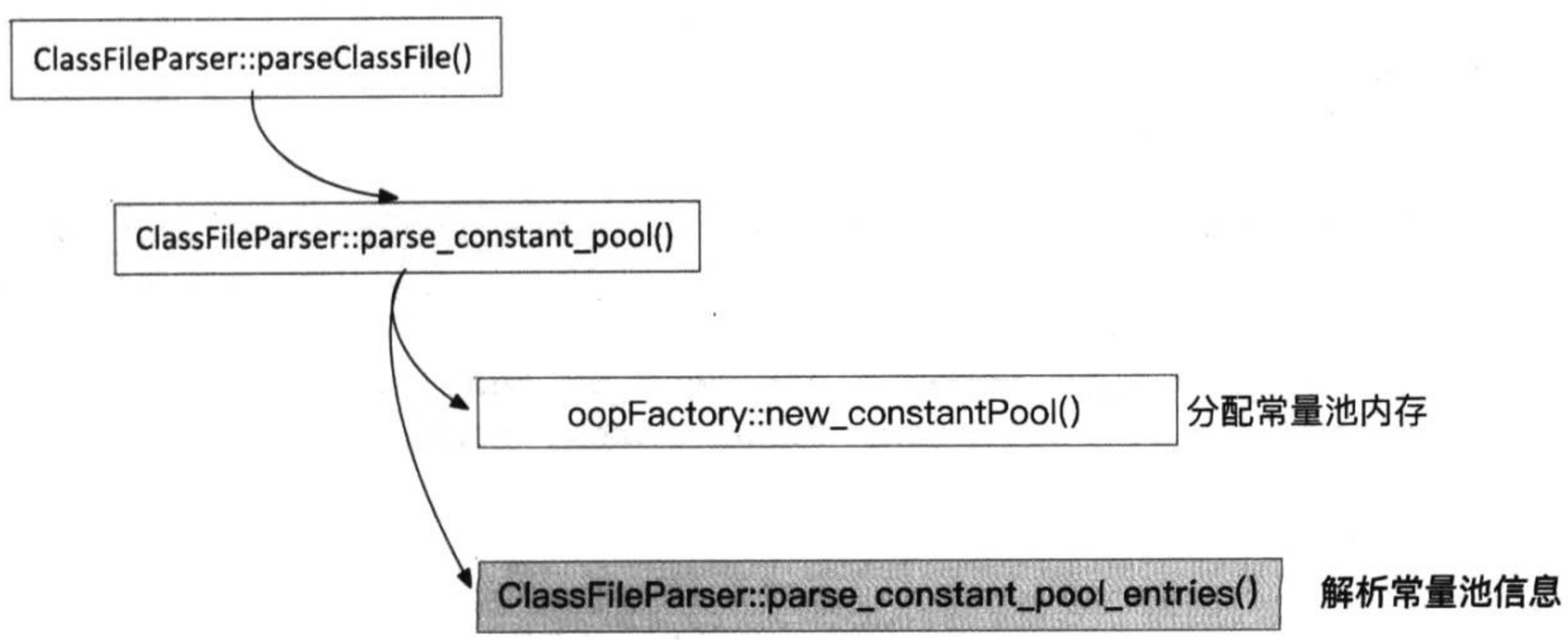

图 5.31　解析常量池元素的链路位置

先一睹 parse_constant_pool_entries()函数的“芳容”：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：常量池元素解析函数

```
void ClassFileParser::parse_constant_pool_entries(constantPoolHandle cp, int length, TRAPS) {
    ClassFileStream* cfs0 = stream();
    ClassFileStream cfs1 = *cfs0;
    ClassFileStream* cfs = &cfs1;

    const char* names[SymbolTable::symbol_alloc_batch_size];
    int lengths[SymbolTable::symbol_alloc_batch_size];
    int indices[SymbolTable::symbol_alloc_batch_size];
    unsigned int hashValues[SymbolTable::symbol_alloc_batch_size];
    int names_count = 0;

    // parsing  Index 0 is unused
    for (int index = 1; index < length; index++) {
      u1 tag = cfs->get_u1_fast();
      switch (tag) {
        case JVM_CONSTANT_Class :
          {
            cfs->guarantee_more(3, CHECK);  // name_index, tag/access_flags
            u2 name_index = cfs->get_u2_fast();
            cp->klass_index_at_put(index, name_index);
          }
          break;
        case JVM_CONSTANT_Fieldref :
          {
            cfs->guarantee_more(5, CHECK);  // class_index, name_and_type_index, tag/access_flags
            u2 class_index = cfs->get_u2_fast();
            u2 name_and_type_index = cfs->get_u2_fast();
            cp->field_at_put(index, class_index, name_and_type_index);
          }
          break;
        case JVM_CONSTANT_Methodref :
          {
            cfs->guarantee_more(5, CHECK);  // class_index, name_and_type_index, tag/access_flags
            u2 class_index = cfs->get_u2_fast();
            u2 name_and_type_index = cfs->get_u2_fast();
            cp->method_at_put(index, class_index, name_and_type_index);
          }
          break;
        //...

    }
```

本函数主要通过一个 for 循环处理所有的常量池元素，每次循环开始的时候，先执行语句 u1 tag = cfs->get_u1_fast()，从字节码文件中读取占 1 字节宽度的字节流。之所以这样，是因为在字节码文件中的常量池元素区，每一个常量池元素起始的 1 字节都用于描述常量池元素类型（这在前文分析过），JVM 解析常量池元素的第一步就是需要知道每个常量池元素的类型。

获取到常量池元素类型之后，通过 switch 条件表达式，对不同类型的常量池元素进行处理。这里以第一个 case JVM_CONSTANT_Class 语句为例进行说明。首先通过 u2 name_index = cfs->get_u2_fast()获取类的名称索引，接着通过 cp->klass_index_at_put(index, name_index)将当前常量池元素的类型和名称索引分别保存到 constantPoolOop 的 tag 数组和数据区。

这里需要注意的是，不同类型的常量池元素在字节码文件中的组成结构也不同，例如 JVM_CONSTANT_Class 类型的常量池元素，其组成结构是：u1 宽度的类型标识 + u2 宽度的名称索引，因此 JVM 只需要先调用 cfs->get_u1_fast()获取类型标识，再调用 cfs->get_u2_fast()获取其索引。再如 JVM_CONSTANT_Fieldref 类型的常量池元素，其组成结构是：u1 宽度的类型标识 + u2 宽度的类索引 + u2 宽度的名称索引，因此 JVM 需要先调用 cfs->get_u1_fast()获取类型标识，再连续调用两次 cfs->get_u2_fast()分别获取类索引和名称索引，代码的逻辑如下：

```
case JVM_CONSTANT_Fieldref :
{
    //获取类索引
    u2 class_index = cfs->get_u2_fast();

    //获取名称索引
    u2 name_and_type_index = cfs->get_u2_fast();

    //保存到 constantPoolOop 的 tag 和数据区中
    cp->field_at_put(index, class_index, name_and_type_index);
}
```

继续刚才类型为 JVM_CONSTANT_Class 的常量池元素的解析，获取到名称索引后，接着执行 cp->klass_index_at_put(index, name_index)将当前常量池元素信息保存到 constantPoolOop 中。先看看其逻辑实现方式：

清单：/src/share/vm/oops/constantPoolOop.hpp
功能：解析常量池元素

```
void klass_index_at_put(int which, int name_index) {
    tag_at_put(which, JVM_CONSTANT_ClassIndex);
    *int_at_addr(which) = name_index;
}

void tag_at_put(int which, jbyte t) {
    tags()->byte_at_put(which, t);
}
```

在这个逻辑中，通过 tag_at_put(which, JVM_CONSTANT_ClassIndex)将当前常量池元素的类型保存到 constantPoolOop 所指向的 tag 的对应位置的数组中，并通过*int_at_addr(which) = name_index 将当前常量池元素的名称索引保存到 constantPoolOop 的数据区中对应的位置。

姑且将 tag 理解为数组（事实上也是，只不过被 oop 对象头包住了），当前常量池元素本身在字节码文件常量区中的位置索引将决定该常量池元素最终在 tag 数组中的位置，也决定该常量池元素的索引值最终在 constantPoolOop 的数据区的位置。

1．类元素解析

为了说明概念，还是举个例子比较好，这里再次“祭出”前面所举的 Iphone6s.java 这个类，使用 javap 命令查看该类的结构如下：

清单：Iphone6s.java
作用：使用 Java 类描述 iPhone 6S 手机参数

```
Compiled from "Iphone6s.java"
   public class Iphone6s extends java.lang.Object
     SourceFile: "Iphone6s.java"
     minor version: 0
     major version: 50
     Constant pool:
   const #1 = class        #2;    //  Iphone6s
   const #2 = Asciz        Iphone6s;
   const #3 = class        #4;    //  java/lang/Object
   const #4 = Asciz        java/lang/Object;
   const #5 = Asciz        length;
   const #6 = Asciz        I;
   const #7 = Asciz        width;
   const #8 = Asciz        height;
   const #9 = Asciz        weight;
   const #10 = Asciz       ram;
   const #11 = Asciz       rom;
   const #12 = Asciz       pixel;
   const #13 = Asciz       <init>;
   const #14 = Asciz       ()V;
   const #15 = Asciz       Code;
   const #16 = Method      #3.#17; //  java/lang/Object."<init>":()V
   const #17 = NameAndType #13:#14;//  "<init>":()V
   const #18 = Field       #1.#19; //  Iphone6s.length:I
   const #19 = NameAndType #5:#6;//  length:I
   const #20 = Field       #1.#21; //  Iphone6s.width:I
   const #21 = NameAndType #7:#6;//  width:I
   const #22 = Field       #1.#23; //  Iphone6s.height:I
   const #23 = NameAndType #8:#6;//  height:I
```

```
const #24 = Field      #1.#25; //  Iphone6s.weight:I
const #25 = NameAndType #9:#6;//  weight:I
const #26 = Field      #1.#27; //  Iphone6s.ram:I
const #27 = NameAndType #10:#6;//  ram:I
const #28 = Field      #1.#29; //  Iphone6s.rom:I
const #29 = NameAndType #11:#6;//  rom:I
const #30 = Field      #1.#31; //  Iphone6s.pixel:I
const #31 = NameAndType #12:#6;//  pixel:I
const #32 = Asciz      LineNumberTable;
const #33 = Asciz      LocalVariableTable;
const #34 = Asciz      this;
const #35 = Asciz      LIphone6s;;
const #36 = Asciz      SourceFile;
const #37 = Asciz      Iphone6s.java;
```

由于第一号常量池元素的类型是 JVM_CONSTANT_Class，还由于其在常量区中的位置是 1，因此其最终在 tag 和 constantPoolOop 数据区中的位置也是 1。当 JVM 解析完这个常量池元素后，constantPoolOop 的内存布局如图 5.32 所示。

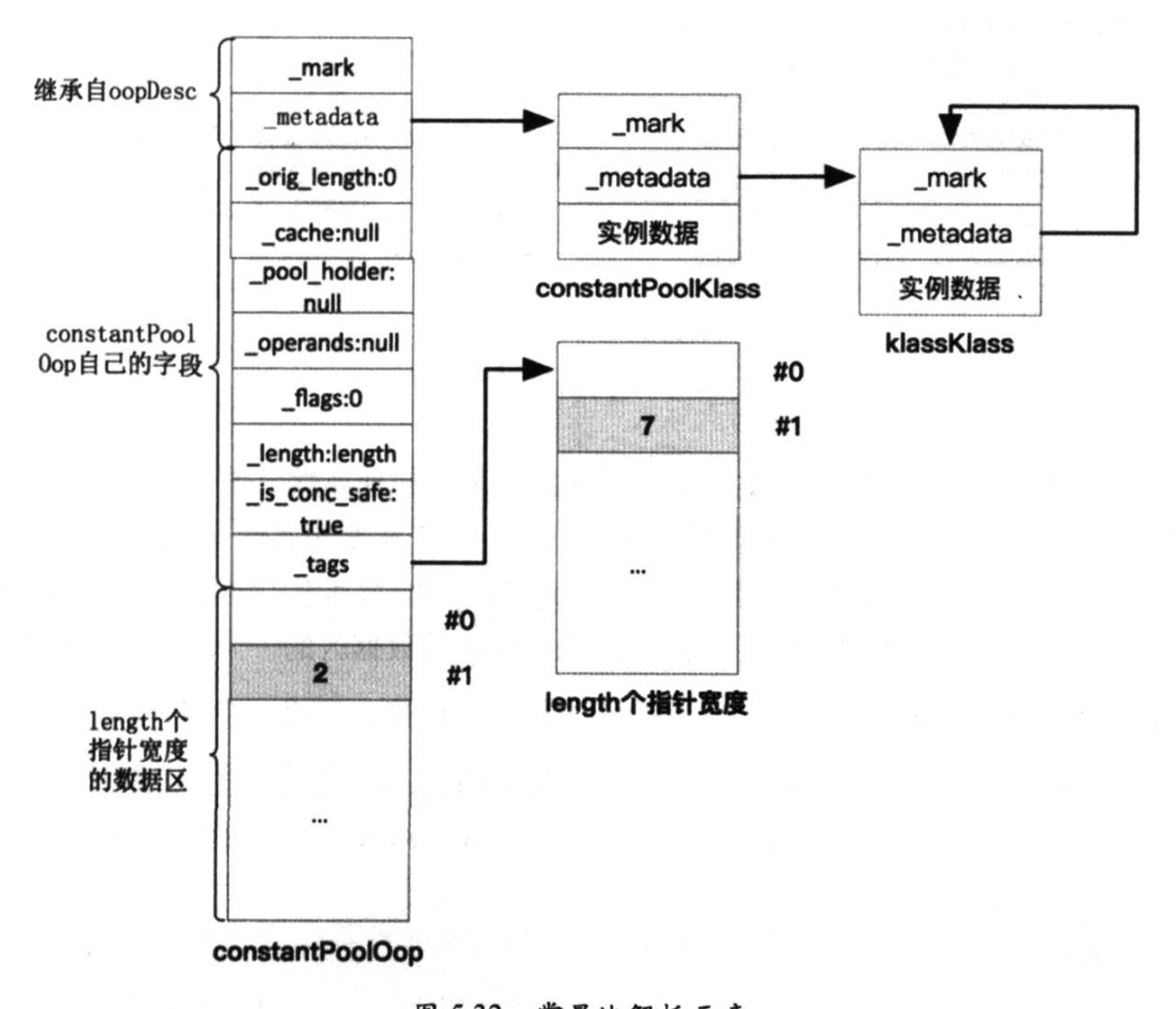

图 5.32　常量池解析示意

图 5.32 中的#0 和#1 表示数组下标位置，其中在 constantPoolOop 的数据区的#1 号位置所存储的值为 2，因为当前常量池元素（第 1 号常量池元素）的名称索引为 2。而 tag 的#1 号位置所存储的值为 7，因为当前常量池元素的类型是 JVM_CONSTANT_Class，该枚举值为 7。

这里贴出常量池元素类型枚举的定义：

清单：/src/share/vm/prims/jvm.h
功能：常量池元素类型枚举

```
enum {
    JVM_CONSTANT_Utf8 = 1,
    JVM_CONSTANT_Unicode,                /* unused */
    JVM_CONSTANT_Integer,
    JVM_CONSTANT_Float,
    JVM_CONSTANT_Long,
    JVM_CONSTANT_Double,
    JVM_CONSTANT_Class,
    JVM_CONSTANT_String,
    JVM_CONSTANT_Fieldref,
    JVM_CONSTANT_Methodref,
    JVM_CONSTANT_InterfaceMethodref,
    JVM_CONSTANT_NameAndType,
    JVM_CONSTANT_MethodHandle          = 15,  // JSR 292
    JVM_CONSTANT_MethodType            = 16,  // JSR 292
    //JVM_CONSTANT_(unused)            = 17,  // JSR 292 early drafts only
    JVM_CONSTANT_InvokeDynamic         = 18,  // JSR 292
    JVM_CONSTANT_ExternalMax           = 18   // Last tag found in classfiles
};
```

2. 方法元素解析

有些常量池元素比较特殊，其类型标识后面跟了不止一种属性，方法元素就是其中一种。在方法常量池元素的占 u1 宽度的标识后面，跟了 2 个占 u2 宽度的属性：类索引和方法名索引。这便带来一个问题：常量池元素的类型会保存在 tag 数组中，而其索引会保存在 constantPoolOop 的数据区，但是这里有 2 个索引，如何在 constantPoolOop 的数据区的一个位置上保存 2 个值呢？

JVM 给出的方案很简单，就是将这两个索引值进行拼接，变成一个值，然后再保存。

JVM 解析类型为 JVM_CONSTANT_Methodref 的常量池元素时，会调用 cp->method_at_put(index, class_index, name_and_type_index)函数，我们看一下该函数的实现：

清单：/src/share/vm/oops/constantPoolOop.hpp
功能：方法元素解析

```
void method_at_put(int which, int class_index, int name_and_type_index) {
    tag_at_put(which, JVM_CONSTANT_Methodref);
```

```
    *int_at_addr(which) = ((jint) name_and_type_index<<16) | class_index;
}
```

看到了没？这里将 name_and_type_index 左移了 2 字节，通过位或操作符与 class_index 完成两个值的拼接。在这里，位或操作符|之所以能够实现两个数值的拼接，是因为这 2 个数值都只占 2 字节。

还是以 Iphone6s.java 为例，其常量池中第 16 号元素是 Method，其属性如下：

```
const #16 = Method     #3.#17; //  java/lang/Object."<init>":()V
```

这表示其 class_index 是 3，而方法名索引是 17，当 JVM 将其解析完成后，constantPoolOop 的内存布局如图 5.33 所示。

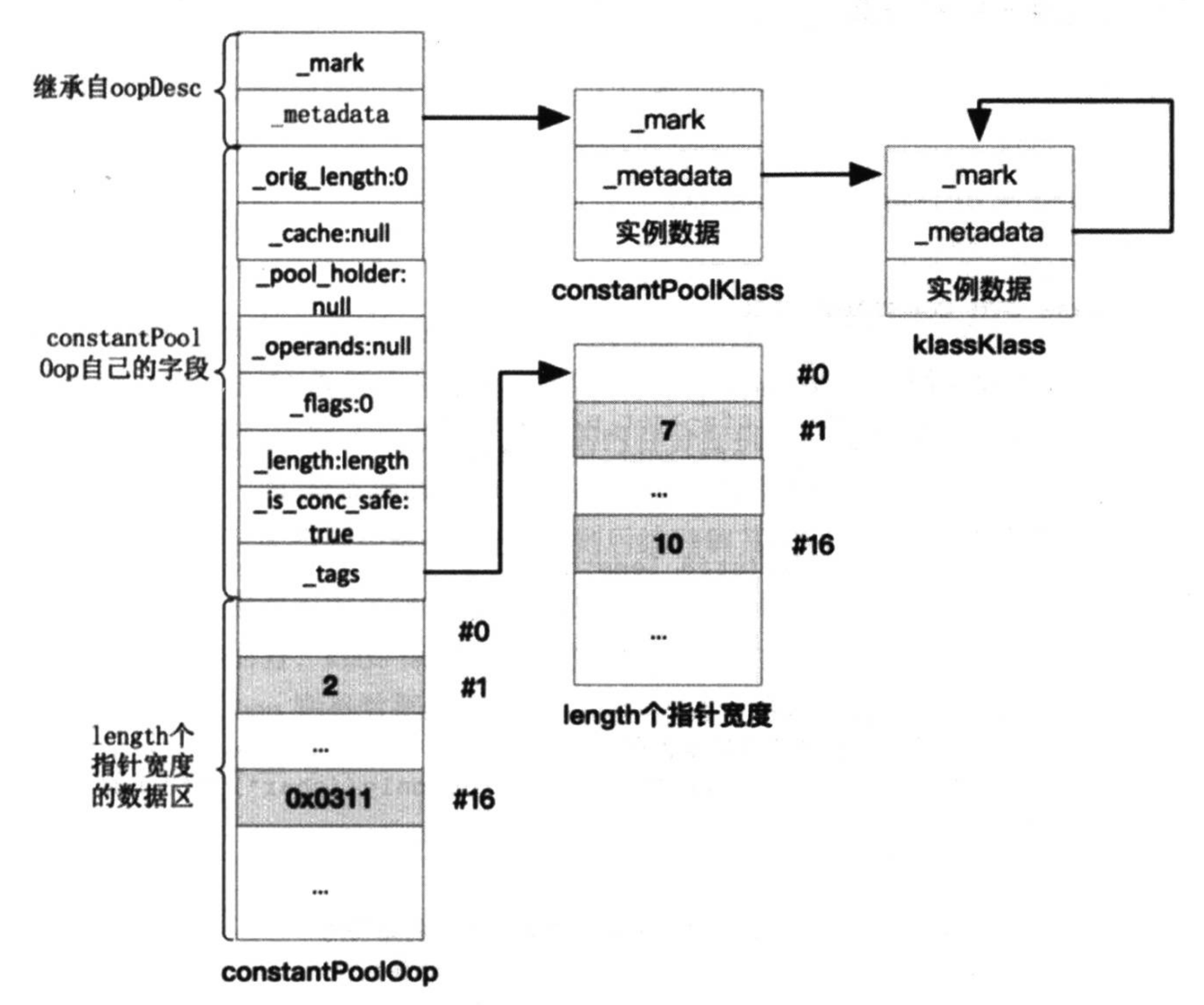

图 5.33　解析方法常量池元素后的内存布局

在 constantPoolOop 的数据区的第 16 号位置所保存的值是 0x0311，这正是该方法所对应的类索引值 3 和方法名索引 17 这两个数值拼接后的值。

3．字符串元数据解析

对于常量池而言，字符串的概念比较广泛，并不单指字符串变量。类名、方法名、类型、this 指针名，等等，都可以看作是字符串，最终都会被 JVM 当作字符串处理，存储到符号区。

由于无论是 tag 还是 constantPoolOop 的数据区，一个存储位置只能存放一个指针宽度的数据，而字符串往往很大，因此 JVM 专门设计了一个“符号表”的内存区，tag 和 constantPoolOop 数据区内仅保存指针指向符号区。

JVM 对字符串的处理如下：

清单：/src/share/vm/classfile/classFileParser.cpp
功能：常量池元素解析函数

```
void ClassFileParser::parse_constant_pool_entries(constantPoolHandle cp, int
length, TRAPS) {
    // …
    for (int index = 1; index < length; index++) {
      u1 tag = cfs->get_u1_fast();
      switch (tag) {
        //...
        case JVM_CONSTANT_Utf8 :
          {
            // 读取字符串长度
            u2  utf8_length = cfs->get_u2_fast();
            u1* utf8_buffer = cfs->get_u1_buffer();

            //…
            cfs->skip_u1_fast(utf8_length);

            //…

            unsigned int hash;
            //从符号表中查找该字符串是否存在
            Symbol* result = SymbolTable::lookup_only((char*)utf8_buffer,
utf8_length, hash);
            if (result == NULL) {

              //如果符号表中没有，就往符号表中新增一个字符串符号
              names[names_count] = (char*)utf8_buffer;
              lengths[names_count] = utf8_length;
              indices[names_count] = index;
              hashValues[names_count++] = hash;
              if (names_count == SymbolTable::symbol_alloc_batch_size) {
                SymbolTable::new_symbols(cp, names_count, names, lengths,
indices, hashValues, CHECK);
```

```
        names_count = 0;
      }
    } else {
      cp->symbol_at_put(index, result);
    }
  }
  break;
  //…

}
```

以上代码给出了一个基本思路，即字节码文件中的字符串常量池元素最终都会被保存到符号表中。为了节省内存，JVM 会先判断符号表中是否存在相同的字符串，如果已经存在，则不会申请内存。这就是如果你在一个类中定义了两个字符串，但是这两个字符串的值相同，最终这两个字符串变量都会同时指向常量池中同一个位置的原因。

5.6　本章总结

本章详细分析了解析 Java 字节码常量池的原理，描述了 JVM 内部常量池对象的内存分配机制，并由此阐述了 JVM 内部的对象表示机制——oop-klass 模型。想要研究 JVM 内核的道友应当认真阅读本章，最好能够非常熟悉常量池的内存分配机制，因为后续章节会讲解 Java 字段、方法的内存分配机制，而 Java 字段和方法的内存分配机制基本与常量池的内存分配机制类似，因此对常量池的内存分配机制理解得越透彻，则后续理解其他对象的内存分配机制便会越轻松。

当然，在理解常量池的内存分配机制之前，必须先搞清楚 JVM 内部对象的表示机制——oop-klass 机制。总体而言，klass 保存一个类型的“元”信息，说白了，klass 便是 JVM 内部对数据结构的实现，而 oop 则是数据结构的实例表示方式。各位道友务必弄清楚此机制，否则越到后面越感到“云里雾里”。