

第 10 章

类的生命周期

本章摘要

- 类的生命周期
- 类加载的内部实现及触发
- 类的初始化
- 类加载器的本质
- 类实例分配

Java 类的生命周期是一个绕不开的话题，本书将从源代码的角度来讲解 Java 类生命周期的技术实现细节。上一章所描述的 Java 执行引擎，更是与类的生命周期息息相关——Java 执行引擎直接负责和管理 Java 类生命周期的大部分阶段，包括类的加载、初始化、创建与方法调用。

10.1 类的生命周期概述

Java 程序的所有数据结构和算法都封装在类型之中，这也是面向对象编程语言的一大特色。当 JVM 执行一个 Java 类所封装的算法之前，首先要做的一件事便是字节码文件解析，字节码文件解析包含 3 个主要的过程——常量池解析、Java 类字段解析及 Java 方法解析。通过类字段解析，JVM 能够分析出 Java 类所封装的数据结构；通过方法解析，JVM 能够分析出 Java 类所封装的算法逻辑。而无论是数据结构还是方法信息，很多与“字符串”或者大数据（是指占二进位比较多的大数）相关的信息都封装于常量池中，所以 JVM 欲解析字段和方法信息，必先解析常量池。前文详细描述了字节码文件解析的技术实现细节，当常量池、字段和方法信息全部

被解析完，则字节码文件的“精华”便已经被完全消化吸收。但是，这几个过程其实仅仅属于 Java 类“加载”过程中的一个环节，这对于一个 Java 类的整个“漫长”的生命周期而言，仅仅是个开始。在字节码文件的精华被吸收之后还需要经过一系列的“二次”消化处理，方能被 JVM 在运行期“随心所欲”地调用。

按照 JVM 规范，一个 Java 文件从被加载到被卸载的整个生命过程，总共要经历 5 个阶段：加载→链接（验证+准备+解析）→初始化（使用前的准备）→使用→卸载。其中第二个阶段“链接”，对应了 3 个阶段：验证、准备和解析，因此，也有很多典籍说 Java 类的生命周期一共包括 7 个阶段。

前文所讲的常量池解析、Java 字段和方法的解析，其实都属于加载阶段的一部分。所谓加载，简而言之就是将 Java 类的字节码文件加载到机器内存中，并在内存中构建出 Java 类的原型——类模板对象。所谓类模板对象，其实就是 Java 类在 JVM 内存中的一个快照，JVM 将从字节码文件中解析出的常量池、类字段、类方法等信息存储到类模板中，这样 JVM 在运行期便能通过类模板而获取 Java 类中的任意信息，能够对 Java 类的成员变量进行遍历，也能进行 Java 方法的调用。反射的机制即基于这一基础。如果 JVM 没有将 Java 类的声明信息存储起来，则 JVM 在运行期也无法反射。字节码相关的工具类库，例如 asm、cglib 等，都利用了这一机制，在运行期动态修改静态声明的 Java 类所对应的字节码内容，从而在运行期直接改掉 Java 类的定义，甚至直接在运行期创建一个全新的 Java 类。

Java 类是写给人类看的，而 JVM 内存中的类模板快照则是写给机器“看”的。物理机器无法直接执行 Java 类的源代码，所以需要通过类加载这样一个过程将字节码格式的 Java 类转换成机器能够识别的内存类模板快照。

JVM 完成 Java 类加载之后，接着便开始进行链接。所谓链接，虽然与编译原理中的链接不是同一件事，然而本质上是相同的。总体而言，链接的主要作用是将字节码指令中对常量池中的索引用转换为直接引用。链接包含 3 个步骤：验证、准备和解析。其实在类加载阶段（也即类的生命周期的第一个阶段），JVM 会对字节码文件进行验证，只不过该阶段的验证着重于字节码文件格式本身，与“链接”阶段的验证侧重点不同。在链接阶段，着重于由字节码信息出发进行反向验证，例如，验证根据字节码文件中的类名是否能够找到对应的类模板。这些都验证无误之后，JVM 才能放心地加载当前类，也才能放心地将字节码指令中对常量池索引号的引用重写为直接引用。关于链接的具体技术实现，实在是太重要了，因为在 JVM 执行字节码指令的过程中，会依赖于重写机制，例如 invokevirtual 指令。重写本身比较简单，但是与方法调用联系到一起，就比较复杂了，本书暂时不对此进行深入分析，不知各位道友的胃口深浅，如果各位的口味都比较“重”，那么下一版中要好好讲一讲。

在链接阶段，在正式使用 Java 类之前的最后一道工序便是“初始化”。这里所谓的初始化，

并非指对类进行实例化，而是指执行类的<clinit>()方法。Java 类的实例化，对应的乃是 Java 类生命周期中的“使用”阶段。类的<clinit>()方法在前文讲解 JVM 解析 Java 类方法时曾经详细分析过，若有不清楚的小伙伴可以翻看那一章。总体而言，当 Java 类中包含 static 修饰的静态字段，或者有使用 static {} 块包裹的代码段时，编译后便会在字节码文件中包含一个名为<clinit>() 的方法，JVM 在初始化阶段便会调用该方法。需要说明一点，该方法仅能由 Java 编译器生成并由 JVM 调用，程序开发者无法自定义一个同名的方法，更无法直接在 Java 程序中调用该方法，虽然该方法也是由字节码指令所组成。

等 JVM 完成类的初始化之后，便“万事俱备，只欠东风”，就等着开发者使用了。使用方式多种多样，其中最常见的一种方式是通过 new 关键字来实例化一个 Java 类。当然，除了通过 new 关键字使用 Java 类，还可以有多种方式，例如下面这个例子：

清单：/Test.java

功能：演示 Java 类的使用

```
public static void main(String[] args) throws Exception {
    Map map = (Map)Class.forName("java.util.HashMap").newInstance();
    map.put("aaa", "sss");
    System.out.println("map: " + map.get("aaa"));
}
```

该示例使用 Class.forName(String) 接口加载一个类，并通过 Class.newInstance() 接口实例化一个类。

从广义上说，类的加载也可以对应类生命周期的 7 个阶段中的前 5 个阶段，即加载、验证、准备、解析和初始化。当类加载之后，JVM 内部会为 Java 类创建一个对等的类模板，类模板在 JDK 6 时代被存储在所谓的 perm 区，而到了 JDK 8 时代，则被存储在所谓的 metaSpace 区。无论存储在哪里，当存储区即将被打爆而这个类又不再使用时，JVM 的 GC 便有可能将其回收，即释放内存。而当实例化一个 Java 类之后，JVM 内部则会为 Java 类实例对象创建一个对等的实例对象，该实例对象所存储的区域与具体的 GC 策略紧密关联，有可能在新生代，也可能在老年代，当然，更可能在栈上（栈上分配）。当类被使用完毕之后，JVM 必须销毁实例对象，否则 JVM 内存区早晚会被打爆。JVM 对类模板的销毁和类实例对象的销毁，都是卸载。

总体而言，Java 类的生命周期如图 10.1 所示。

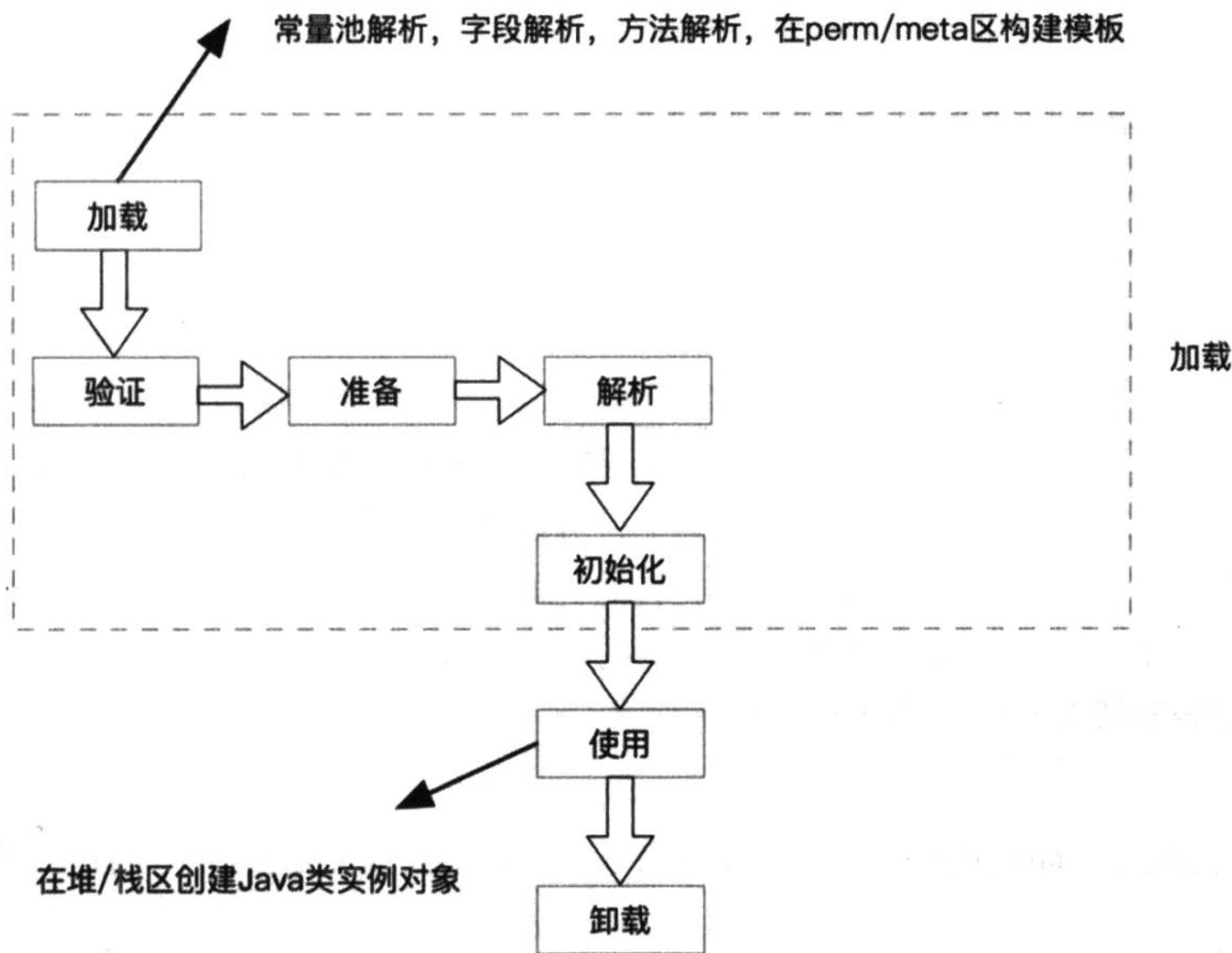


图 10.1 Java 类的生命周期

虽然 Java 类的生命周期包含 7 个阶段，然而这 7 个阶段到底做了些什么事情，内部实现机制如何，我们仍然一无所知。如果不了解这些内部机制，则即使知道这 7 个阶段，怕是也没啥用处。所以下面还是通过 HotSpot 来一窥其中的门道。

10.2 类加载

前文已经描述过 JVM 对字节码文件的精华部分的解析过程，当字节码文件解析完成之后，JVM 便会在内部创建一个与 Java 类对等的类模板对象，说白了该对象其实是 C++类的实例。每一个 Java 类模型，最终在 JVM 内部都会有一个 klassOop 与之对等，Java 类中的字段、方法及常量池等都会保存到 klassOop 实例对象中。要注意，这个实例对象并非 Java 类的实例对象，其仅仅用于表示 Java 类型本身，或者 Java 类的定义。与 Java 类实例对象对等的 JVM 内部对象是 instanceOop 实例。

下面就从 Java 类模板对象——instanceKlass 的创建开始讲起。

前面描述过 Java 字节码文件的常量池解析、字段解析与方法解析，这三部分内容的解析便是 Java 字节码文件的精华所在。这三部分内容的解析都位于 ClassFileParser::parseClassFile()函

数中，对应的接口分别如下：

- ◎ parse_constant_pool(), 解析常量池。
- ◎ parse_fields(), 解析 Java 类字段。
- ◎ parse_methods(), 解析 Java 类方法。

这 3 个接口的源代码在前面详细解读过。当这 3 个接口执行完成之后，Java 字节码文件的精华便被分析完了，至此 JVM 便对 Java 类中所定义的一切数据结构和算法“了如指掌”，为了巩固“胜利成果”，JVM 需要将这些好不容易辛辛苦苦解析出来的结果保存起来。这些解析的结果会存储到 klassOop 这个内部类对象实例中，可以将该对象看作 Java 类在 JVM 内部完全对等的一个镜像——只不过 Java 类是写给人类看的，而内部镜像 klassOop 则是写给机器读的。当成功保存解析结果之后，则 Java 类的生命周期的第一个阶段——加载，便大功告成。不看过程看结果，类加载阶段其实就是为了这一目标而来——在 JVM 内部创建一个与 Java 类结构对等的数据对象。

HotSpot 仍然在 ClassFileParser::parseClassFile() 函数中完成 klassOop 的创建，其主要源码如下：

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

功能：创建 klassOop

```
instanceKlassHandle ClassFileParser::parseClassFile(Symbol* name,
                                                     Handle class_loader,
                                                     Handle protection_domain,
                                                     KlassHandle host_klass,
                                                     GrowableArray<Handle>* cp_patches,
                                                     TempNewSymbol& parsed_name,
                                                     bool verify,
                                                     TRAPS) {
    // ...
    // 解析常量池
    constantPoolHandle cp = parse_constant_pool(CHECK_(nullHandle));

    //.....
    // 解析 Java 类中定义的字段
    typeArrayHandle fields = parse_fields(cp, access_flags.is_interface(),
                                           &fac, &fields_annotations, CHECK_(nullHandle));

    // ...
    // 解析 Java 类中定义的方法
    objArrayHandle methods = parse_methods(cp, access_flags.is_interface(),
                                             &promoted_flags,
                                             &has_final_method,
                                             &methods_annotations_oop,
```

```

        &methods_parameter_annotations_oop,
        &methods_default_annotations_oop,
        CHECK_(nullHandle));

// ...
// 开始创建与 Java 类对等的内部对象
klassOop ik = oopFactory::new_instanceKlass(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");
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);
// ...

// 设置 itable 偏移表
klassItable::setup_itable_offset_table(this_klass);
}

```

在这段源码中，笔者将常量池解析、Java 类字段解析、Java 方法解析的调用同时贴了出来，这样方便各位道友把握字节码文件解析的主脉络。在创建 klassOop 时，首先通过调用 oopFactory::new_instanceKlass() 接口在内存中构建一个 instanceKlass 对象实例，该接口的机制与在前文介绍过的 JVM 构建常量池对象实例的逻辑类似，如果有不熟悉的小伙伴可以专门看看常量池解析的那一章，这里不再赘述。不过与常量池的构建、方法对象 methodOop 的构建一样，构建实例对象绕不开的一个问题是，所创建的对象占多大的内存空间？要弄清楚这个问题，很简单，只需要看下在调用 oopFactory::new_instanceKlass() 接口时所传入的参数，上面的源码包含了该接口调用的部分，从上面的源码可知，在调用该接口时传入了 vtable_size、itable_size、static_field_size 和 total_oop_map_count 这 4 个与大小有关的数据，之所以要传入这 4 个数据，是因为它们与 klassOop 在内存中的实际布局是有关系的。oopFactory::new_instanceKlass() 接口所构建的对象类型是 instanceKlass，该类型继承自 Klass 类，其内部结构如下：

```

instanceKlass//类结构
Klass//结构部分
jint _layout_helper //布局类型

```

```

juint _super_check_offset
Symbol* _name //类名
klassOop _secondary_super_cache
// ...
jint _biased_lock_revocation_count
instanceKlass//结构部分
klassOop _array_klasses
objArrayOop _methods //Java 类中定义的方法信息
typeArrayOop _fields //Java 类中定义的字段信息
oop _class_loader //类加载器
typeArrayOop _inner_classes //内部类
int _nonstatic_field_size //非静态字段的大小
int _static_field_size //静态字段大小
int _vtable_len //虚方法表长度
// ...
volatile u2 _idnum_allocated_count

```

从 instanceKlass 的结构可以看到，其内部定义了若干字段，这些字段足以存储 Java 类规范所支持的一切信息，例如字段、方法、内部类等，因为 instanceKlass 要作为 Java 类在 JVM 内部对等的结构体，所以能够兼容 Java 类中的所有元素是其唯一的设计目标。但是 JVM 在创建 instanceKlass 对象时，为其所申请的内存空间却超过了 instanceKlass 类型本身所需的内存大小，这是因为 JVM 需要在 instanceKlass 内存空间的末尾再预留出足够的空间，存储虚方法表 vtable、接口表 itable 及 JAVA 类中的引用类型表 oopMap。存储虚方法表 vtable，其作用在前文分析 Java 方法的解析机制时详细描述过，这里不再赘述。itable 与 oopMap 也是各有其作用。不过在调用 oopFactory::new_instanceKlass() 接口创建 instanceKlass 对象实例时，还传入了 static_field_size 这个数据，其表示 Java 类中所定义的静态字段所占内存的大小。不过静态字段在不同的 JDK 版本中存放的位置不同，在 JDK 6 中，静态字段会被分配到 instanceKlass 实例对象所申请的内存空间中，而在 JDK 7 和 8 中，静态字段将会被分配到与 instanceKlass 对等的镜像类——java.lang.Class 实例中，关于静态字段的分配及镜像类会在下文详细分析，此处先略过不表。由于在 JDK6 中，静态字段信息也会存放在 instanceKlass 对象的预留内存空间中，因此最终 JVM 为 instanceKlass 申请的内存空间大小实际上是 instanceKlass 类型本身所占的内存大小与 vtable、itable、static fields 及 oopMap 的大小之和，这种逻辑在代码中得到了体现，我们看 JDK 6 中 oopFactory::new_instanceKlass() 接口的实现逻辑：

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

功能：为 instanceKlass 申请内存空间

```

klassOop
instanceKlassKlass::allocate_instance_klass(int vtable_len, int itable_len,
                                             int static_field_size,
                                             unsigned nonstatic_oop_map_count,
                                             ReferenceType rt, TRAPS) {

```

```

const int nonstatic_oop_map_size =
    instanceKlass::nonstatic_oop_map_size(nonstatic_oop_map_count);
int size = instanceKlass::object_size(align_object_offset(vtable_len) +
align_object_offset(itable_len) + static_field_size + nonstatic_oop_map_size);

// Allocation
KlassHandle h_this_klass(THREAD, as_klassOop());
KlassHandle k;
if (rt == REF_NONE) {
    instanceKlass o;
    k = base_create_klass(h_this_klass, size, o.vtbl_value(), CHECK_NULL);
} else {
    instanceRefKlass o;
    k = base_create_klass(h_this_klass, size, o.vtbl_value(), CHECK_NULL);
}
// ...
}

```

与为常量池和方法对象申请内存空间的实现逻辑一样，在为 instanceKlass 申请内存空间时，oopFactory 也是先计算所要申请的内存大小，然后调用相应的接口进行申请。在这段代码中可以看到，所要申请的预留内存空间大小 size 的计算逻辑是 vtable、itable、static fields 及 oopmap 之和，因此最终静态字段一定在这个预留空间中。

在 JDK 6 中，JVM 在 instanceKlass 的内存空间末尾预留出足够的空间，存放虚方法表 vtable、接口表 itable、静态字段信息表及 Java 类中的引用类型表 oopMap，这 4 张表存放的顺序依次是 vtable、itable、static fields 和 oopMap，这是由 JVM 的源码所规定的，源码如下：

清单：/src/share/vm/oops/instanceKlass.hpp

功能：vtable、itable 和 oopMap 的存放顺序

```

// 获取 instanceKlass 对象实例在内存中所占用的空间大小
static int header_size() {
    return align_object_offset(oopDesc::header_size() +
sizeof(instanceKlass)/HeapWordSize);
}

// 获取 vtable 的起始偏移量，该偏移量正是 instanceKlass 类型本身的大小
static int vtable_start_offset() {
    return header_size();
}

// 获取 vtable 的内存地址
// 该地址值为 instanceKlass 实例对象的内存首地址 + instanceKlass 对象实例的内存大小
intptr_t* start_of_vtable() const {
    return ((intptr_t*)as_klassOop()) + vtable_start_offset();
}

```

```

// 获取 itable 的内存地址
// 该地址为 vtable 的内存首地址 + vtable 的长度，该长度即为 itable 相对于 vtable 的偏移量
intptr_t* start_of_itable() const {
    return start_of_vtable() + align_object_offset(vtable_length());
}

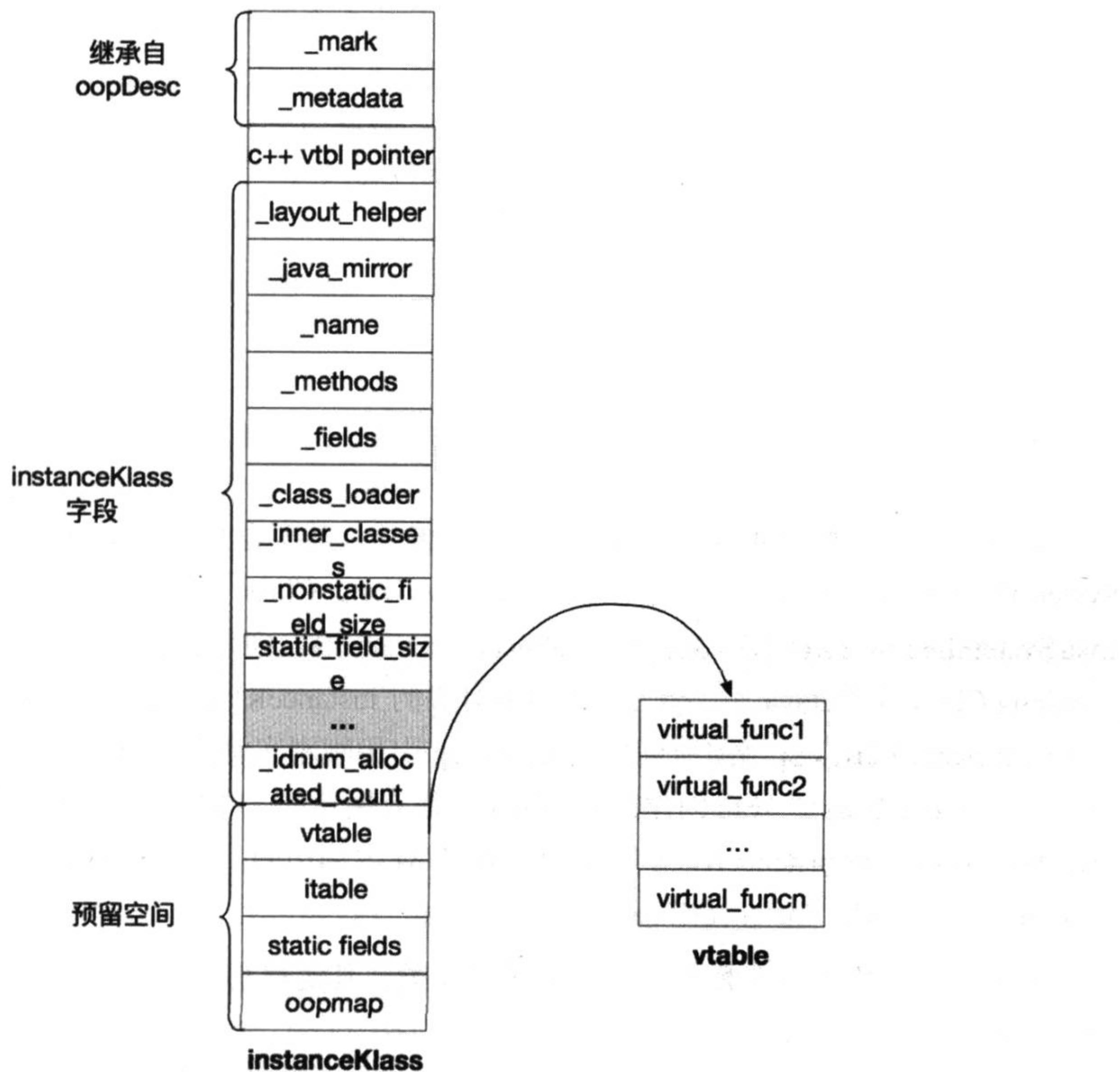
// 静态字段的起始偏移量
HeapWord* start_of_static_fields() const {
    return (HeapWord*)(start_of_itable() + align_object_offset(itable_length()));
}

// 获取 oopMap 的内存地址
// 该地址为 itable 的内存首地址 + itable 的长度，该长度即为 oopMap 相对于 itable 的偏移量
OopMapBlock* start_of_nonstatic_oop_maps() const {
    return (OopMapBlock*)(start_of_itable() +
align_object_offset(itable_length()));
}

```

在 instanceKlass.hpp 文件中定义了 3 种接口，分别用于获取 vtable、itable 和 oopMap 这 3 种数据在内存中的地址，通过上面这段逻辑分析可知，这 3 部分数据的存储顺序依次是 vtable、itable 和 oopMap。由此可知，JVM 调用 oopFactory::new_instanceKlass() 接口所创建的数据结构实际如图 10.2 所示。

图 10.2 中的这个内存数据结构，便是 Java 类加载的最终产物，也是 Java 类在内存中的对等体。JVM 根据这个数据结构，能够获取 Java 类中所定义的一切元素。仔细观察图 10.2，会发现其中有一项数据是 c++ vtbl pointer，这是何物？其实顾名思义，这便是 C++ 类型对象中的虚方法表指针。由于 HotSpot 内部的 klass 都继承自基类 Klass，而 Klass 类又继承自 Klass_vtbl，但是 Klass_vtbl 中却有一个虚方法 unused_initial_virtual()。前面在讲解 Java 的虚方法表 vtable 的实现机制时曾经讲过，如果 C++ 类中也包含虚方法，则编译器会在 C++ 实例对象头部插入虚方法表的指针，其道理与 Java 的虚方法表一样，都是为了实现多态。不过在 JDK 6 里面让所有的 klass 类型都继承自 Klass_vtbl，是为了更好地管理 vtable，方便方法区的内存回收，但是到了 JDK 8 就没有这玩意儿了，因为所有的 klass 都继承了 Metadata 类。

图 10.2 `instanceKlass` 的内存布局结构图

10.2.1 类加载——镜像类与静态字段

类加载的最终结果便是在JVM的方法区创建一个与Java类对等的`instanceKlass`实例对象，但是在JVM创建完`instanceKlass`之后，又创建了与之对等的另一个镜像类——`java.lang.Class`。在JDK 6中，创建镜像类的逻辑被包含在`instanceKlass::allocate_instance_klass()`函数中，在该函数的末尾执行`java_lang_Class::create_mirror()`调用，该接口实现逻辑如下：

清单：/src/share/vm/classfile/javaClasses

功能：创建镜像类

```
oop java_lang_Class::create_mirror(KlassHandle k, TRAPS) {
    int computed_modifiers = k->compute_modifier_flags(CHECK_0);
    k->set_modifier_flags(computed_modifiers);
```

```

if (SystemDictionary::Class_klass_loaded()) {
    Handle mirror =
instanceKlass::cast(SystemDictionary::Class_klass())->allocate_permanent_instance(CHECK_0);
    mirror->obj_field_put(klass_offset, k());
    k->set_java_mirror(mirror());

    // ...
    return mirror();
} else {
    return NULL;
}
}

```

通过观察这段源码可知，所谓的 mirror 镜像类，其实也是 instanceKlass 的一个实例对象，SystemDictionary::Class_klass() 返回的便是 java_lang_Class 类型，因此 instanceMirrorKlass::cast(SystemDictionary::Class_klass())->allocate_instance(k, CHECK_0) 这行代码就是用来创建 java.lang.Class 这个 Java 类型在 JVM 内部对等的 instanceKlass 实例的。接着通过 k->set_java_mirror(mirror()) 调用，让当前所创建的 klassOop 引用刚刚实例化的 java.lang.Class 对象。JVM 之所以在 instanceKlass 之外再创建一个 mirror，是有用意的，总体而言，java.lang.Class 是为了被 Java 程序调用，而 instanceKlass 则是为了被 JVM 内部访问。所以，JVM 直接暴露给 Java 的是 java_mirror，而不是 InstanceKlass。

事实上，JDK 类库中所提供的反射等工具类，其实都基于 java.lang.Class 这个内部镜像实现。例如下面这个 Java 程序：

清单：/Test.java

功能：演示 Java 的反射功能

```

public class Test {
    public Integer i;
    public String s;

    public int add(int a, int b){
        return a + b;
    }

    public static void main(String[] args) throws Exception {
        Class klass = Class.forName("Test");
        System.out.println(klass.getFields().length);
        for(int i = 0; i < klass.getFields().length; i++){
            System.out.println(klass.getFields()[i]);
        }
    }
}

```

该示例 Java 类很简单，Test 类中包含 2 个公开的字段和一个公开的方法，在 main()方法中通过 java.lang.Class.forName(String)接口反射获取 Test 类型，反射之后通过 java.lang.Class.getFields()接口获取 Test 类中所包含的全部公开字段数组，并遍历字段数组，打印出字段名。运行该程序，输出如下：

```
2
public java.lang.Integer Test.i
public java.lang.String Test.s
```

打印结果显示 Test 类中一共包含 2 个公开字段，与定义的完全一致。在这里，重点研究的是，java.lang.Class.getFields()接口究竟如何知道 Test 类中有两个公开的字段。源码面前无秘密。首先看 java.lang.Class.getFields()接口，该接口最终会调用 java.lang.Class.getDeclaredFields0(boolean publicOnly)接口，该接口是一个 native 接口，其最终调用的接口位于 HotSpot 内部的函数中，该函数如下：

清单：/src/share/vm/prims/jvm.cpp

功能：获取类中声明的字段

```
JVM_ENTRY(jobjectArray, JVM_GetClassDeclaredFields(JNIEnv *env, jclass
ofClass, jboolean publicOnly))
{
    JVMWrapper("JVM_GetClassDeclaredFields");
    JvmtiVMObjectAllocEventCollector oam;

    instanceKlassHandle k(THREAD,
java_lang_Class::as_klassOop(JNIEnvHandles::resolve_non_null(ofClass)));
    constantPoolHandle cp(THREAD, k->constants());

    // 执行类的链接阶段
    k->link_class(CHECK_NULL);

    // 通过 k->fields() 获取类中的全部字段
    typeArrayHandle fields(THREAD, k->fields());
    int fields_len = fields->length();

    // ...

    return (jobjectArray) JNIHandles::make_local(env, result());
}
JVM_END
```

上面这个 JVM_GetClassDeclaredFields()函数便是 java.lang.Class.getDeclaredFields0(boolean publicOnly)这个 Java 类方法所对应的内部实现。由于 java.lang.Class.getDeclaredFields0(boolean publicOnly)方法是类的成员方法，因此该方法包含一个隐藏的入参 this，this 指向 java.lang.Class

类型实例自己，所以调用的 `JVM_GetClassDeclaredFields()` 函数的第 2 个人参 `ofClass` 便是 `java.lang.Class` 类型实例。同时，在执行上面这个 `JVM_GetClassDeclaredFields()` 函数调用时，说明其前面的一个步骤——`Class klass = Class.forName(“Test”)` 已经执行完了，此时在 JVM 内部的 `klass` 实例，实际上是 `Test` 类型在 JVM 内部的镜像类，虽然 `java.lang.Class` 仅仅是一个镜像类，但是也保存了 `Test` 这个 Java 类中的全部信息，所以在 `JVM_GetClassDeclaredFields()` 函数中能够获取 `Test` 类中的全部字段。这便是 Java 反射的原理。通过本示例也可以知道，Java 的反射是离不开 `java.lang.Class` 这个镜像类的。

如果思维再放得开阔一点，可以这样认为，即使 JVM 内部没有安排 `java.lang.Class` 这么一个媒介作为面向对象反射的基础，那么 JVM 也必然要定义另外类，假设这个类就叫作 `Reflection`，这个类能够直接被 Java 程序开发者使用，那么 `Reflection` 这个类也必然需要在 JVM 内部与所要反射的目标 Java 类所对应的 `instanceKlass` 之间建立联系，能够让 Java 开发者通过这个 `Reflection` 类反射出目标 Java 类的字段、方法等全部信息。从这个意义上而言，`java.lang.Class` 并非是偶然有的，而是必然，是 Java 这种面向对象的语言与虚拟机实现机制这两种规范下的必然技术实现，如果说有巧合的话，那便是恰好叫了“`java.lang.Class`”这个类名。

既然 `java.lang.Class` 是一个必然的存在，所以每次 JVM 在内部为 Java 类创建一个对等的 `instanceKlass` 时，都要再创建一个对应的 `Class` 镜像类，作为反射的基础。

刚才讲过，在 JDK 6 中，静态字段会存储在 `instanceKlass` 的预留空间里，在 JVM 为 `instanceKlass` 申请内存空间时已经为静态字段预留了空间，而在创建完 `instanceKlass` 之后，JVM 在 `ClassFileParser::parseClassFile()` 函数中调用 `this_klass->do_local_static_fields(&initialize_static_field, CHECK_(nullHandle))` 对这部分内存空间进行初始化，`do_local_static_fields()` 函数的实现如下：

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

功能：为 Java 类中静态字段分配空间

```
void instanceKlass::do_local_static_fields(void f(fieldDescriptor*, TRAPS),
TRAPS) {
    instanceKlassHandle h_this(THREAD, as_klassOop());
    do_local_static_fields_impl(h_this, f, CHECK);
}

void instanceKlass::do_local_static_fields_impl(instanceKlassHandle
this_oop, void f(fieldDescriptor* fd, TRAPS), TRAPS) {
    fieldDescriptor fd;
    int length = this_oop->fields()->length();
    for (int i = 0; i < length; i += next_offset) {
        fd.initialize(this_oop(), i);
        if (fd.is_static()) { f(&fd, CHECK); } // Do NOT remove {}! (CHECK macro
```

```

expands into several statements)
}
}
}

```

这段逻辑遍历 Java 类中的全部静态字段并逐个将其塞进 instanceKlass 的预留空间中。在这段逻辑中,需要注意, instanceKlass::do_local_static_fields(void f(fieldDescriptor*, TRAPS), TRAPS) 函数的第一个入参是函数指针,看上面这段逻辑, instanceKlass::do_local_static_fields(void f(fieldDescriptor*, TRAPS), TRAPS) 内部调用了 instanceKlass::do_local_static_fields_impl(instanceKlassHandle this_oop, void f(fieldDescriptor* fd, TRAPS), TRAPS),而在后者内部则通过函数指针 f 调用其指向的函数。那么指针 f 指向哪个函数呢?

在 ClassFileParser::parseClassFile() 函数中调用 instanceKlass::do_local_static_fields(void f(fieldDescriptor*, TRAPS), TRAPS) 时,所传入的函数指针是&initialize_static_field,所以该指针指向的函数如下:

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

功能: 初始化静态字段

```

static void initialize_static_field(fieldDescriptor* fd, TRAPS) {
    KlassHandle h_k (THREAD, fd->field_holder());
    if (fd->has_initial_value()) {
        BasicType t = fd->field_type();
        switch (t) {
            case T_BYTE:
                h_k()->byte_field_put(fd->offset(), fd->int_initial_value());
                break;
            case T_BOOLEAN:
                h_k()->bool_field_put(fd->offset(), fd->int_initial_value());
                break;
            case // ...
        }
    }
}

```

在该函数中,通过调用 h_k()->**_field_put() 系列接口,将不同类型的静态字段存储到 instanceKlass 对象实例的预留内存空间中,如此便完成了 Java 类中静态字段的存储。而在 JDK 8 中,静态字段不再存储于 instanceKlass 预留空间,而是转移到 instanceKlass 的镜像类——java.lang.Class 的预留空间里去,因此在 JDK 8 的源码中,上面的这个 initialize_static_field() 函数定义到 javaClasses.cpp 中了。同时,创建 mirror 镜像类的接口也不再在 java_lang_Class::create_mirror() 函数中调用,而是在 ClassFileParser::parseClassFile() 函数中调用。虽然调用的地方不同了,但是函数实现的内部机制并没有从根本上发生变化,因此从这一点上看, JDK 6 和 JDK 8 并没有做很大的变更。JDK 8 之所以要将静态字段从 instanceKlass 迁移到 mirror 中,也不是没有道理,

毕竟静态字段并非 Java 类的成员变量，如果从数据结构这个角度看，静态字段不能算作 Java 类这个数据结构的一部分，因此 JDK 8 将静态字段转移到 mirror 中。从反射的角度看，静态字段放在 mirror 中是合理的，毕竟在进行反射时，需要给出 Java 类中所定义的全部字段，无论字段是不是静态类型。例如，将上面的 Test 类做个修改，在里面增加一个 static 类型的公开字段，则最终的打印结果会包含该字段。

综上所述，对于 JDK 6 而言，类加载阶段所产出的最终结果便是如图 10.3 所示的这两个实例对象。

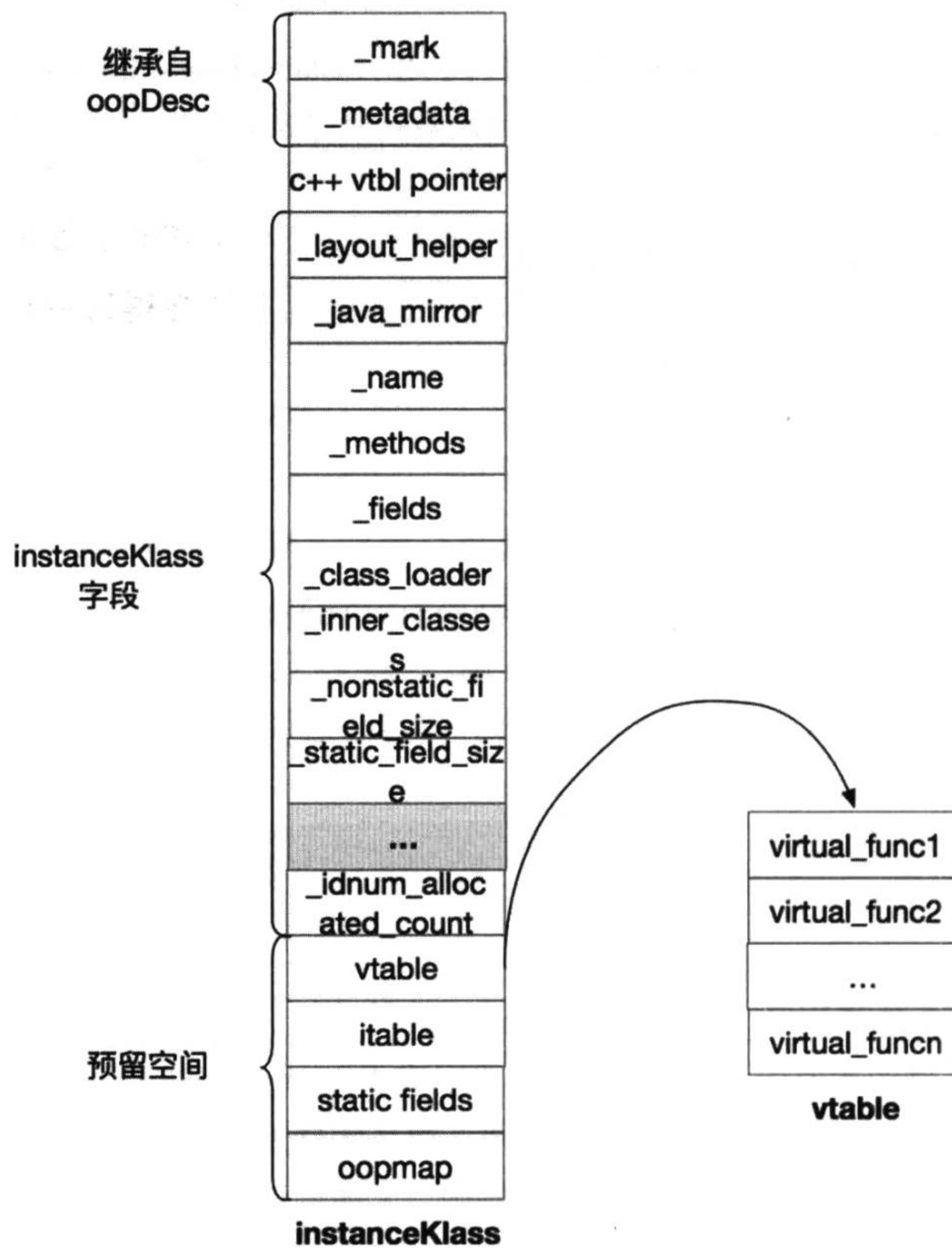


图 10.3 java 类加载阶段所产生的结果

在 JDK 6 中，由于 mirror 也是一个 instanceKlass，因此其包含了 instanceKlass 所包含的一切字段。

10.2.2 Java 主类加载机制

到上一节为止，Java 类加载的过程终于全部讲完了。在前面章节详细讲解了常量池解析、字段解析、方法解析、instanceKlass 创建及镜像类的创建。之所以要逐个详细讲解，一方面是因为 JVM 使用 C/C++ 编写而成，而 C/C++ 语言本身就比 Java 语言更具难度，相信只要不是直接从事 JVM 开发的道友，阅读起来都会比较吃力，里面有太多的内存分配、回收、指针、类型转换的内容，笔者作为 Java 开发者，阅读过程中也费了无数脑筋，相当不轻松，因此笔者感同身受，将一些比较关键的源代码和算法详细描述出来，这是自己辛苦阅读的一种沉淀，相信也会帮助很多对 C/C++ 语言不够熟悉的道友。另一方面是因为 JVM 作为虚拟机，里面涉及的计算机基础知识多而杂，几乎覆盖了方方面面，其实现也复杂，然而其过程也精彩，所以虽然阅读的过程痛苦，但是结果却是快乐的，理解了原理之后再次面对 Java 程序，会有一种“一览众山小”之快感，你就是 JVM 世界里的神，做神的感觉，其美妙不足为外人道也，而这种享受也是支持笔者这两年里一直坚持写下去的最大动力。有苦有乐，生活才能丰富多彩。

牛皮吹完，我们应该总结一下类加载的整体过程了。虚拟机在得到一个 Java class 文件流之后，接下来要完成的主要步骤如下：

- (1) 读取魔数与版本号。
- (2) 解析常量池，parse_constant_pool()。
- (3) 解析字段信息，parse_fields()。
- (4) 解析方法，parse_methods()。
- (5) 创建与 Java 类对等的内部对象 instanceKlass，new_instanceKlass()。
- (6) 创建 Java 镜像类，create_mirror()。

以上便是一个 Java 类加载的核心流程。了解了类加载的核心流程之后，也许聪明的你会忍不住想，Java 类的加载到底何时才会被触发呢？Java 类加载的触发条件比较多，其中比较特殊的便是 Java 程序中包含 main() 主函数的类——这种类一般也被称作 Java 程序的主类。Java 主类的加载由 JVM 自动触发——JVM 执行完自身的若干初始化逻辑之后，第一个加载的便是 Java 程序的主类。总体上而言，Java 主类加载的链路如下：

```

java.c::JavaMain(): 执行 mainClass = LoadClass(env, classname)
java.c::LoadClass(): 执行 cls = (*env)->FindClass(env, buf)
jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name)):
执行 loader = Handle(THREAD, SystemDictionary::java_system_loader())
jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name)):
执行 result = find_class_from_class_loader(env, sym, true, loader,
protection_domain, true, thread) 加载主类

```

```
jvm.cpp::find_class_from_class_loader(): 执行 klassOop klass =
SystemDictionary::resolve_or_fail(name, loader, protection_domain, throwError != 0, CHECK_NULL)
    SystemDictionary::resolve_or_fail()
        SystemDictionary::resolve_or_null()
            SystemDictionary::resolve_instance_class_or_null(): 执行 k =
load_instance_class(name, class_loader, THREAD) (Do actual loading)
                SystemDictionary::load_instance_class()
                    JavaCalls::call_virtual();
                    java.lang.ClassLoader.loadClass(String)
                        sun.misc.AppClassLoader.loadClass(String, boolean)
                        java.lang.ClassLoader.loadClass(String, boolean)
                        java.net.URLClassLoader.findClass(final String)
                        java.net.URLClassLoader.defineClass(String, Resource)
                        java.lang.ClassLoader.defineClass(String,
java.io.ByteBuffer, ProtectionDomain)
                            native java.lang.ClassLoader.defineClass0()
                                ClassLoader.c::Java_java_lang_ClassLoader_defineClass1()
                                    jvm.cpp::JVM_DefineClassWithSource()
                                    jvm.cpp::jvm_define_class_common()
                                    SystemDictionary.cpp::resolve_from_stream()
                                    ClassFileParser.cpp::parseClassFile()
```

上面是 Java 程序 main 主类加载的整体链路，该调用链路的核心逻辑如下：

(1) JVM 启动后，操作系统会调用 java.c::main() 主函数，从而进入 JVM 的世界。java.c::main() 方法调用 java.c::JavaMain() 方法，java.c::JavaMain() 方法主要执行 JVM 的初始化逻辑，初始化完毕之后，便会搜索 Java 程序的 main() 主函数所在的类，也即“主类”，找到主类的类名之后，便会调用 mainClass = LoadClass(env, classname) 对主类进行加载。

(2) LoadClass(env, classname) 方法是 java.c::LoadClass() 方法，而后者执行 cls = (*env)->FindClass(env, buf) 来寻找主类。

(3) (*env)->FindClass(env, buf) 函数首先跳转到 jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name))，JNI_ENTRY 是一个宏，在预编译阶段便已展开，这个宏作用的结果是：(*env)->FindClass(env, buf) 最终会调用 jni.cpp::jni_FindClass(JNIEnv *env, const char *name) 函数。

jni.cpp::jni_FindClass(JNIEnv *env, const char *name) 函数先调用 loader = Handle(THREAD, SystemDictionary::java_system_loader()) 获取类加载器。Java 程序主类的类加载器默认是系统加载器，该加载器是 JDK 类库中定义的 sun.misc.AppClassLoader，关于该加载器的细节会在后文详述。JVM 体系中加载器的继承关系如图 10.4 所示。

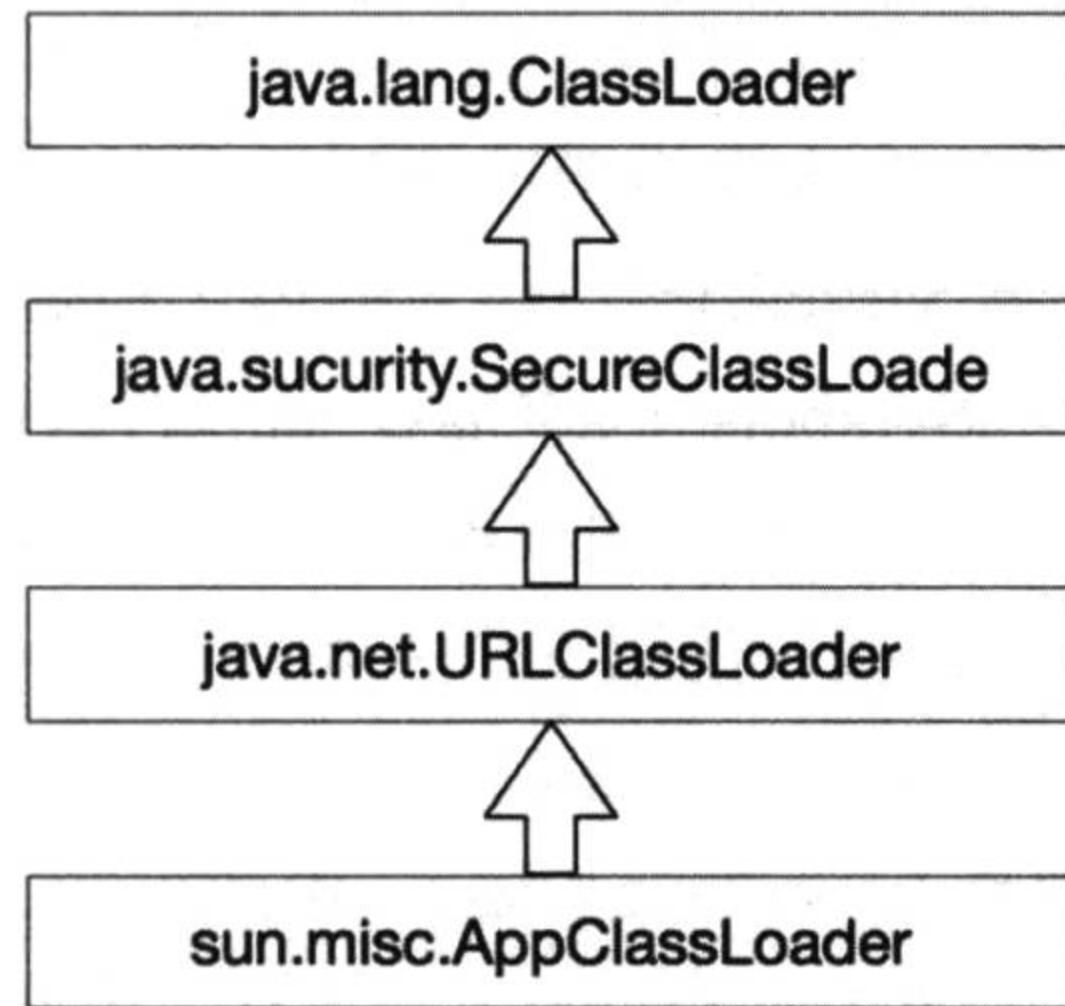


图 10.4 JVM 系统加载器的继承关系

由图 10.4 可知，系统加载器所继承的顶级父类是 `java.lang.ClassLoader`，这是 JDK 类库所提供的核心加载器。事实上，无论 Java 程序内部有没有自定义类加载器，最终都会调用 `java.lang.ClassLoader` 所提供的几个 native 接口完成类的加载，这些接口主要包括如下 3 种：

```

private native Class<?> defineClass0(String name, byte[] b, int off, int len,
ProtectionDomain pd);
private native Class<?> defineClass1(String name, byte[] b, int off, int len,
ProtectionDomain pd, String source);
private native Class<?> defineClass2(String name, java.nio.ByteBuffer b, int
off, int len, ProtectionDomain pd, String source);
  
```

Java 主类的加载也无法绕过这 3 个接口。

`jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name))` 函数内部获取到系统加载器之后，接着便开始调用 `find_class_from_class_loader()` 接口加载主类，而后者则调用 `SystemDictionary::resolve_or_fail()` 接口。

(4) `SystemDictionary::resolve_or_fail()` 接口经过一系列调用，最终调用 `SystemDictionary::resolve_instance_class_or_null()` 接口，该接口内部逻辑比较冗长，会经过层层判断，确认同一个加载器没有别的线程在加载同一个类，则最终会执行真正的加载，调用 `SystemDictionary::load_instance_class()` 接口，该接口内部执行如下调用：

```

JavaCalls::call_virtual(&result,
                        class_loader,
                        spec_klass,
                        vmSymbols::loadClass_name(),
                        vmSymbols::string_class_signature(),
                        string,
                        CHECK_(nh));
  
```

`JavaCalls::call_virtual()` 接口的主要功能是根据输入的参数，调用指定的 Java 类中的指定方

法。该接口的第 2 个人参（人参从位置 1 开始计数）指明所调用的 Java 类对应的 instance，第 4 个人参指明所调用的特定方法，第 5 个人参指明所调用的 Java 类的签名信息。当 JVM 执行 Java 程序主类加载时，向 JavaCalls::call_virtual() 接口传入的第 2 和第 4 个人参分别是 class_loader 和 vmSymbols::loadClass_name()，vmSymbols::loadClass_name() 返回的方法名是 loadClass()，而 class_loader 则是前置流程中实例化好的系统加载器——AppClassLoader，在 JVM 内部对等的实例对象。同时，JavaCalls::call_virtual() 接口的第 5 个人参是 vmSymbols::string_class_signature()，其返回的字符串是(Ljava/lang/String;)Ljava/lang/Class，该字符串表示所调用的 Java 方法的人参是 Ljava/lang/String，而返回值则是 Ljava/lang/Class。由此可知，当 JVM 加载 Java 程序的主类时，最终会调用 AppClassLoader.loadClass(String) 这个方法。由此，JVM 的流程便转移到了 Java 的世界，进入到了 Java 类的逻辑流之中。

JavaCalls::call_virtual() 接口的第 6 个人参则包含所调用的 Java 方法所需要的全部人参信息，在 JVM 加载 Java 应用程序主类时，向 JavaCalls::call_virtual() 接口所传入的第 6 个人参是 string，在 SystemDictionary::load_instance_class() 函数中，该人参封装了所需要加载的 Java 类的全限定名称，最终这个全限定名称将作为 java.lang.AppClassLoader.loadClass(String) 接口的人参，系统加载器据此加载目标 Java 类。

JavaCalls::call_virtual() 接口最终会调用 JavaCalls::call() 接口，JavaCalls::call() 接口调用 JavaCalls::call_helper()，而后者则会调用 StubRoutines::call_stub() 例程，对于该例程，阅读过全书的小伙伴一定不会陌生，该例程在本书前面专门花了一章去讲解，有不清楚的小伙伴可以回过去仔细阅读。总体而言，该例程在运行期对应着一段机器码，其作用是辅佐 JVM 执行 Java 类方法。这里不得不提一句，JVM 作为一款虚拟机，其本身由 C/C++ 语言写成，但是 JVM 是为执行 Java 字节码文件而生的，因此 JVM 内部必然有一套机制能够从 C/C++ 程序调用 Java 类中的方法，这套机制便通过 JavaCalls 类来实现，该类中定义了各种 call_*() 接口，这些接口最终都要调用 StubRoutines::call_stub() 例程，从而辅佐 JVM 执行 Java 方法。

事实上，JavaCalls::call_virtual() 接口在 JVM 内部是一个很常用的接口，大凡涉及 Java 类成员方法的调用，最终都会经过该接口。

(5) 经过上一个步骤，JVM 最终会调用 sun.misc.AppClassLoader.loadClass(String) 接口加载 Java 应用程序的主类。AppClassLoader 继承自 java.lang.ClassLoader 这个基类，java.lang.ClassLoader.loadClass(String) 方法调用 loadClass(String, boolean) 方法，由于继承的关系，实际调用的是 sun.misc.AppClassLoader.loadClass(String, boolean) 方法，该方法的实现逻辑如下：

清单：/src/sun/misc/Launcher.java

功能：系统加载器加载类的逻辑

```
public Class loadClass(String name, boolean resolve) throws
```

```

ClassNotFoundException{
    int i = name.lastIndexOf('.');
    if (i != -1) {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPackageAccess(name.substring(0, i));
        }
    }
    return (super.loadClass(name, resolve));
}

```

这段代码逻辑是，先判断所加载的类名中是否包含点号“.”，如果包含则说明传入的一定是类的全限定名，包含了包名，则 JVM 调用 SecurityManager 模块检查包的访问权限。通过访问权限验证之后，则调用 super.loadClass(name, resolve)方法。由于继承关系，super.loadClass(name, resolve)方法其实调用的是 java.lang.ClassLoader.loadClass(String name, boolean resolve)方法，该方法的主要逻辑如下：

清单：/src/java/lang/ClassLoader

功能：java.lang.ClassLoader.loadClass(String name, boolean resolve)方法逻辑

```

protected synchronized Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {
    Class c = this.findLoadedClass(name);
    if(c == null) {
        try {
            if(this.parent != null) {
                c = this.parent.loadClass(name, false);
            } else {
                c = this.findBootstrapClassOrNull(name);
            }
        } catch (ClassNotFoundException var5) { ; }

        if(c == null) {
            c = this.findClass(name);
        }
    }

    if(resolve) {
        this.resolveClass(c);
    }

    return c;
}

```

在 java.lang.ClassLoader.loadClass(String name, boolean resolve) 方法中，首先通过 findLoadedClass(name)方法判断当前加载器是否加载过指定的类，如果没有加载，则判断当前加

载器的 parent 是否为 null，如果不为 null，则调用 parent.loadClass(name, false)方法，通过父加载器加载指定的 Java 类。AppClassLoader 的父加载器是 ExtClassLoader，这是扩展类加载器，用于加载 JDK 中指定路径下的扩展类，这种加载器不会加载 Java 应用程序的主类，所以程序流会进入 if(this.parent != null){} 代码块，但是 parent.loadClass(name, false) 返回 null。接着 java.lang.ClassLoader.loadClass(String name, boolean resolve) 方法只能通过调用 this.findClass(name) 来加载 Java 主类。

java.lang.ClassLoader.findClass(String) 方法直接抛出异常，因此该类注定要由子类来实现。对于系统类加载器 AppClassLoader，其继承自 URLClassLoader，因此 java.lang.ClassLoader.findClass(String) 方法实际指向 java.net.URLClassLoader.findClass(String)。java.net.URLClassLoader.findClass(String) 方法最终调用 java.lang.ClassLoader.defineClass1() 这一 native 接口，这是一个本地接口，由本地类库实现。openjdk 项目包含了 JDK 核心 Java 类库中的全部本地实现，java.lang.ClassLoader.defineClass1() 所对应的本地实现是 ClassLoader.c::Java_lang_ClassLoader_defineClass1()，有兴趣的道友可自行查看下其实现，这里就不贴代码了，以免占用过多篇幅。通过调用 java.lang.ClassLoader.defineClass1() 接口，Java 程序流又转移到 JVM 内部，因此 Java 类的加载最终仍然是通过 JVM 本地类库得以实现。

ClassLoader.c::Java_lang_ClassLoader_defineClass1() 调用 jvm.cpp::JVM_DefineClassWithSource(), jvm.cpp::JVM_DefineClassWithSource() 调用 jvm.cpp::jvm_define_class_common()，而后者则调用 SystemDictionary.cpp::resolve_from_stream() 接口来加载 Java 主类。在 SystemDictionary.cpp::resolve_from_stream() 接口中，终于开始调用 ClassFileParser.cpp::parseClassFile() 这个函数来解析 Java 主类，并最终创建 Java 主类在 JVM 内部的对等体——klassInstance，由此完成 Java 主类的加载。

10.2.3 类加载器的加载机制

体现 Java 语言强大生命力和巨大魅力的关键因素之一便是，Java 开发者可以自定义类加载器来实现类库的动态加载，加载源可以是本地的 JAR 包，也可以是网络上的远程资源。通过类加载器可以实现非常绝妙的插件机制，这方面的实际应用案例举不胜举，例如，著名的 OSGI 组件框架，再如 Eclipse 的插件机制。类加载器为应用程序提供了一种动态增加新功能的机制，这种机制无须重新打包发布应用程序就能实现。同时，自定义加载器能够实现应用隔离，例如 Tomcat、Spring 等中间件和组件框架都在内部实现了自定义的加载器，并通过自定义加载器隔离不同的组件模块。这种机制比 C/C++ 程序要好太多，想不修改 C/C++ 程序就能为其新增功能，几乎是不可能的，仅仅一个兼容性便能阻挡住所有美好的设想。

Java 应用程序自定义类加载器很简单，只需要继承 `java.lang.ClassLoader` 类，然后覆盖它的 `findClass(String name)` 方法即可。使用自定义类加载器来加载 Java 类的使用方式通常如下：

```
MyClassLoader mcl = new MyClassLoader();
Class klass = mcl.loadClass("com.***.Test");
Test t = (Test)klass.newInstance();
```

通过这段示例程序可以看出，使用自定义类加载器进行 Java 类加载时，首先需要调用加载器的 `loadClass()` 接口完成 `java.lang.Class` 类的加载，然后才能实例化所要加载的类型实例。自定义类加载器的 `loadClass()` 方法所完成的类加载，便是 Java 类生命周期 7 个阶段（加载→验证→准备→解析→初始化→使用→卸载）中的类加载阶段。

有很多自定义的类加载器会重写 `java.lang.ClassLoader` 中的很多接口，实现起来非常复杂，但是无论多么复杂的自定义类加载器，最终都会调用 `java.lang.ClassLoader.defineClass*()` 系列本地接口，最终仍然会由 JVM 内部的本地实现来完成实际的加载工作，在 JVM 内部创建与所要加载的目标 Java 类对等的 `klassInstance` 对象实例，从而完成 Java 类型的加载。

关于 `java.lang.ClassLoader.defineClass*()` 的本地接口实现机制，在前面讲述 Java 应用程序主类加载时已经讲解过，该接口最终会调用 `ClassFileParser.cpp::parseClassFile()` 这个 JVM 内部的函数完成 Java 类的解析，并创建内部对应的对象实例，将 Java 类从字节码文件格式完全转换成内存格式。

10.2.4 反射加载机制

除了通过自定义类加载器完成类的加载，也可以通过 `java.lang.Class.forName(String)` 反射接口完成类加载，不过该接口所完成的加载，包含了 Java 类生命周期 7 个阶段的前面 5 个——加载、验证、准备、解析和初始化。

`java.lang.Class.forName(String)` 最终会调用 `private static native Class<?> forName0()` 这个本地接口完成类加载。`openjdk` 提供了该方法的本地接口实现，如下（仅摘录核心逻辑）：

清单：/src/java/lang/Class.c

功能：通过反射加载类

```
JNIEXPORT jclass JNICALL
Java_java_lang_Class_forName0(JNIEnv *env, jclass this, jstring classname,
                                jboolean initialize, jobject loader)
{
    char *cname;
    jclass cls = 0;
    // ...
```

```

    cls = JVM_FindClassFromClassLoader(env, cname, initialize,
                                       loader, JNI_FALSE);
    return cls;
}

```

java.lang.Class.forName()的本地接口调用 jvm.cpp::JVM_FindClassFromClassLoader()进行类加载，而后者会调用 jvm.cpp::find_class_from_class_loader()接口，前面讲述 Java 主类加载时绘制了 Java 主类的加载链路，Java 主类加载也会经过 jvm.cpp::find_class_from_class_loader()接口，因此两者的后续流程都相同，最终仍然会从 JVM 内部发起对 java.lang.ClassLoader.loadClass(String)接口的调用，从而完成类的真正加载。由此可知，在使用反射机制加载类时，最终仍然走了 java.lang.ClassLoader.loadClass(String)这个方法。其中的技术实现细节前面都已详细阐述过，这里不再赘述。

10.2.5 import 与 new 指令

在硬编码阶段，如果要想使用某个类，必须先 import 进来。但是纵观 Java 的字节码指令，以及编译后的 Java class 字节码文件，里面其实并没有任何关于 import 关键字的解释或痕迹，到了编译阶段，import 关键字就这样悄无声息地消失了。

事实上，import 语句仅仅是个语法糖，是为了不写那一长串的全限定名。import 关键字并没有任何关联的运行时行为，更不会导致类的加载。它的存在纯粹是为了方便写代码，让大家可以把别的 package 的“名字”引入到当前源码文件里直接用。否则，每次在 new 一个 Java 类时，都必须在类名前面补全完整的 package 包名。而 Java 类的加载，则使用了“延迟加载”机制，仅在第一次被使用时才会发生真正的加载，而与是否使用了 import 关键字无关。延迟加载的机制在后文会详细讲解。

虽然 import 语句没有对应的指令，也不会导致类的加载，但是所 import 进来的包名则会被写入 Java class 文件的常量池中，作为字符串储存起来，以用于类加载时的验证和解析。

至于 import 进来的类究竟啥时候会被加载，有好几种情况。例如，使用 new 关键字，或者读写类的静态变量，或者通过反射加载类。本节便来看看使用 new 关键字对类进行实例化的时候，JVM 内部是如何完成类的加载的。

在 32 位 Intel 处理器上，new 指令对应的实现是在 templateTable_x86_32.cpp::TemplateTable::_new()函数中。使用 new 指令加载类时，如果一个类尚未被加载或者未被链接过，则会进入慢分配流程（后文会讲解类的分配机制），而慢分配主要通过调用 InterpreterRuntime::_new()函数完成。InterpreterRuntime::_new()函数会调用 constantPool->klass_at()接口来获取 new 指令后面所对应的 Java 类模板，该接口最终调用 constantPoolOopDesc::klass_at_impl()函数，后

者实现如下（下面仅摘录主要逻辑）：

```
清单: /src/share/vm/oops/constantPoolOop.cpp
功能: new 指令加载类

klassOop constantPoolOopDesc::klass_at_impl(constantPoolHandle this_oop, int
which, TRAPS) {
    // .....
    { ObjectLocker ol(this_oop, THREAD);
        // 获取类加载器
        loader = Handle(THREAD, instanceKlass::cast(this_oop->pool_holder())->
class_loader());
    } // unlocking constantPool

    // ...
    if (do_resolve) {
        // 执行类加载
        klassOop k_oop = SystemDictionary::resolve_or_fail(name, loader, h_prot,
true, THREAD);
    }

    return (klassOop)entry.get_oop();
}
```

如果在应用程序中第一次使用某个类，且此时类尚未被加载进 JVM 内部，会走上面这条链路，先获取类加载器，最终调用 SystemDictionary::resolve_or_fail()接口进行类加载。而 Java 主类的加载也会走到这个接口中，因此后续链路与 Java 主类加载的链路完全相同，最终仍然会调用 java.lang.ClassLoader.loadClass(String)这个 java 接口去执行类加载。本章前面已经详细讲解过 Java 主类的加载过程，因此这里不再赘述。由此可知，在第一次对某个 Java 类使用 new 关键字创建其实例对象时，如果类尚未加载过，则会进入上述流程先完成加载，再进行实例化。

10.3 类的初始化

完成类的加载后，经过链接，便会进入类的初始化阶段。所谓初始化，其实说白了就是调用 java 类的<clinit>()方法。前文已经讲过，该方法是编译器在编译期间自动生成的，当 Java 类中出现静态字段或者包含 static{}块逻辑时，所编译出来的 Java 字节码文件中便会自动包含一个名为<clinit>的方法。该方法不能由程序员在 Java 程序中调用，只能由 JVM 在运行期调用，这个调用的过程便是 Java 类的初始化。注意，<clinit>()方法并非类的构造函数。

JVM 规范规定，当遇到 new、getstatic、invokestatic 等字节码指令或者加载 Java 应用程序

主类或者其他一些情况时，会执行类的初始化逻辑。下面以 new 指令为例，说明 JVM 内部是如何一步一步调用<clinit>()方法的。

当使用 new 关键字来实例化一个 Java 类时，如果该 Java 类是第一次被使用，则必定会先执行加载→链接→初始化逻辑，然后才能创建类实例对象。使用 new 关键字时，在 32 位 Intel 处理器上，new 指令对应的实现在 templateTable_x86_32.cpp::TemplateTable::_new() 函数中。使用 new 指令加载类时，如果一个类尚未被加载和解析过，则会进入慢分配流程，慢分配流程调用 InterpreterRuntime::_new() 函数，而 InterpreterRuntime::_new() 函数会调用 klass->initialize(CHECK) 接口，该接口的实现在 instanceKlass.cpp 中，该接口内部调用 instanceKlass::initialize_impl()，后者调用 instanceKlass::call_class_initializer()，instanceKlass::call_class_initializer() 调用 instanceKlass::call_class_initializer_impl() 接口。该接口实现逻辑如下：

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

功能：Java 类的初始化逻辑

```
void instanceKlass::call_class_initializer_impl(instanceKlassHandle
this_oop, TRAPS) {
    // 获取 Java 类的<clinit>()函数所对应的 method 对象
    methodHandle h_method(THREAD, this_oop->class_initializer());

    // 如果 Java 类没有<clinit>()函数，则不执行初始化逻辑，跳过
    if (h_method() != NULL) {
        JavaCallArguments args; // No arguments
        JavaValue result(T_VOID);
        JavaCalls::call(&result, h_method, &args, CHECK); // 调用<clinit>()函数
    }
}
```

这段逻辑其实比较简单，先获取 Java 类的<clinit>()函数所对应的 method 对象，接着通过 JavaCalls::call() 接口执行初始化方法。如此便完成类的初始化。当然，如果 Java 类中没有定义任何静态字段，也没有 static{} 逻辑块，则编译后的字节码文件中自然不会包含<clinit>()方法，则上面这段逻辑不会执行，直接跳过。

前文讲过，每一个类都有一个加载器，并且不同加载器加载的同一个类无法相互转换。换言之，如果先后使用 2 个不同的类加载器去加载同一个类，则该类必定会先后被加载两次。同理，该类的初始化逻辑也会先后被执行两次。看下面这段示例：

清单：/Test.java

功能：使用不同的类加载器加载同一个类

```
public class Test {
    static {
        System.out.println("static logic. classLoader=" + Test.class.
```

```

getClassLoader());
}

public static void main(String[] args) throws Exception {
    ClassLoader loader = new ClassLoader() {
        @Override
        public Class<?> loadClass(String name) throws ClassNotFoundException {
            try {
                String className = name.substring(name.lastIndexOf(".") + 1) + ".class";
                InputStream is = getClass().getResourceAsStream(className);
                if (is == null) {
                    return super.loadClass(name);
                }
                byte[] buffer = new byte[is.available()];
                is.read(buffer);
                return defineClass(name, buffer, 0, buffer.length);
            } catch (Exception e) {
                throw new ClassNotFoundException(name);
            }
        }
    };
}

System.out.println("start to load Test with custom classLoader");
Object test = loader.loadClass("Test").newInstance();
System.out.println("loaded Test with custom classLoader");

Test t = new Test();
}
}

```

在本示例程序中，为 Test 测试类加入了 static{} 逻辑块，并在里面打印一行文字。这样如果调用类的初始化逻辑，则可以通过打印结果观察到。在本示例程序的主函数中，自定义了一个类加载器，并使用该类加载器加载 Test 测试类。运行该程序，打印结果如下：

```

static logic. classLoader=sun.misc.Launcher$AppClassLoader@2d3fcdbd
start to load Test with custom classLoader
static logic. classLoader=Test$1@36f6e879
loaded Test with custom classLoader

```

通过打印结果可以观察到，在自定义的类加载器准备加载测试类之前，Test 类的 static{} 逻辑便被执行过一次。这是因为 Test 类包含主函数，因此其属于测试程序的主类，在 JVM 启动完成之后便会首先加载该类。使用自定义的类加载器再次加载 Test 类时，由于虽然测试类已经被加载过，但是由于这一次使用的类加载器发生了变化，因此 JVM 便又加载了它一次，因此 Test 的 static{} 块逻辑再次被调用。

10.4 类加载器

要想在 JVM 内部创建一个与 Java 类完全对等的结构模型，必须经过类加载器。类加载器的好处自不必多言，本节开始一起探讨类加载器的本质，类加载器到底是啥，与 JVM 内部究竟有哪些联系，所谓的双亲委派到底是怎么回事，JDK 的核心类库究竟是啥时候加载的……。

10.4.1 类加载器的定义

Java 体系中定义了 3 种类加载器，分别如下：

- ◎ Bootstrap ClassLoader(引导类加载器，缩写为 BCL)。加载指定的 JDK 核心类库，无法由 Java 应用程序直接引用。负责加载下述 3 种情况下所指定的核心类库：
 - %JAVA_HOME%/jre/lib 目录
 - -Xbootclasspath 参数所指定的目录
 - 系统属性 sun.boot.class.path 指定的目录中特定名称的 jar 包
- ◎ Extension ClassLoader (扩展类加载器，缩写为 ECL)。加载扩展类，拓展 JVM 的类库。该加载器加载下述两种情况下所指定的类库：
 - %JAVA_HOME%/jre/lib/ext 目录
 - 系统属性 java.ext.dirs 所指定的目录中的所有类库
- ◎ System ClassLoader (系统类加载器，缩写为 SCL)。加载 Java 应用程序类库，加载类库的路径由系统环境变量 ClassPath 、 -cp 或 系统属性 java.class.path 指定。

除了这 3 种加载器之外，Java 还能支持开发者自定义加载器，自定义的加载器大大丰富了 Java 中间件，在若干 Java 框架和组件中得到极其广泛的应用。通过下面这个测试程序，可以获取运行时 JVM 的各类加载器所加载的类路径：

清单： /Test.java

功能：获取 JVM 各种类加载器的类路径

```
public class Test {
    public static void main(String[] args) {
        System.out.println("引导类加载器加载路径：" + System.getProperty("sun.boot.class.path"));
        System.out.println("扩展类加载器加载路径：" + System.getProperty("java.ext.dirs"));
        System.out.println("系统类加载器加载路径" + System.getProperty("java.class.path"));
    }
}
```

在下文讲述系统类加载器和扩展类加载器的初始化机制时，会看到 JDK 核心类库是通过直接读取这几个系统属性来获取对应加载器的路径的。

从使用的角度看，虽然 JVM 提供了多种多样的类加载器，并且开发者可以自定义若干加载器，但是站在程序的角度看，其实 Java 体系一共只定义了两种类加载器，一种使用 C++ 语言定义，另一种则使用 Java 语言定义。使用 C++ 语言定义的类加载器如下：

清单： /src/share/vm/classfile/classLoader.hpp

功能：使用 C++ 定义的类加载器

```
class ClassLoader: AllStatic {  
  
private:  
    friend class LazyClassPathEntry;  
    static ClassPathEntry* _first_entry;  
    static ClassPathEntry* _last_entry;  
    static PackageHashtable* _package_hash_table;  
    static const char* _shared_archive;  
  
    // 根据 Java 类名加载 Java 类  
    static instanceKlassHandle load_classfile(Symbol* h_name, TRAPS);  
  
    // 设置加载器所加载的类路径  
    static void setup_bootstrap_search_path();  
  
    // ...  
public:  
    // 初始化类加载器  
    static void initialize();  
  
    // ...  
};
```

JVM 内部使用 C++ 定义的 ClassLoader，其实这便是传说中的 bootstrap class loader，即引导类加载器。该加载器内部所有的字段和函数都使用 static 修饰，因此该加载器并不需要实例化，当需要加载 Java 类时，直接调用静态函数。该加载器提供 `setup_bootstrap_search_path()` 接口用于设置加载器所要搜索的类路径，同时提供了一个最重要的方法——`load_classfile()`，来加载指定的 Java 类。该接口的实现如下：

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

功能：引导类加载器加载逻辑

```
instanceKlassHandle ClassLoader::load_classfile(Symbol* h_name, TRAPS) {
    // ...

    instanceKlassHandle h(THREAD, klassOop(NULL));

    ClassFileParser parser(stream);
    Handle class_loader;
    Handle protection_domain;
    TempNewSymbol parsed_name = NULL;
    instanceKlassHandle result = parser.parseClassFile(h_name,
                                                       class_loader,
                                                       protection_domain,
                                                       parsed_name,
                                                       false,
                                                       CHECK_(h));

    // add to package table
    if (add_package(name, classpath_index, THREAD)) {
        h = result;
    }
}

return h;
}
```

如果对前文讲解的 Java 字节码文件的常量池解析比较熟悉，则你对这段代码也不会感到陌生，没错，这个 C++ 加载器的加载接口直接调用了 ClassFileParser::parseClassFile() 接口来解析并加载 Java 类，并最终在 JVM 内部创建一个与 Java 类完全对等的 C++ 对象实例，完成类的加载。如果对 ClassFileParser::parseClassFile() 接口不够熟悉，建议翻看前面的章节，前文对该接口的讲解很详细。JVM 内部所定义的引导类加载器的实现十分干脆纯净，不像使用 Java 所定义的类加载器那么“九弯十八绕”。不过，并不是所有的 Java 虚拟机都会使用 C++ 专门定义一个类加载器，有些 JVM 也使用 Java 语言来定义引导类加载器，只不过其实现仍然要依赖于 JVM 内部所提供的本地接口。是否使用 C++ 来定义引导类加载器并不重要，重要的是，无论是用 Java 编写的加载器，还是用其他语言编写的加载器，其最终目的都是要在 JVM 内部创建一个与 Java 类完全对等的结构体。只要能够实现这个目标，技术上怎么玩都是可以的。

了解了 C++ 定义的类加载器，再看 Java 定义的加载器。使用 Java 语言定义的类加载器，便是 JDK 核心类库中的 java.lang.ClassLoader 类。这里就不贴出该类的主要逻辑了，打开 IDE 便能查看。在 HotSpot 中，除引导类加载器 BCL 外，其余所有的类加载器——无论是 Java 体系所

提供的，还是开发者自定义的，都继承自 `java.lang.ClassLoader`，扩展类加载器与系统类加载器也不例外。扩展类加载器与系统类加载器都定义在 `sun.misc.Launcher` 类中，类名分别是 `ExtClassLoader` 和 `AppClassLoader`，这两个类加载器都继承自 `URLClassLoader`，而 `URLClassLoader` 则继承自 `java.lang.ClassLoader`。事实上，`java.lang.ClassLoader` 提供了绝大多数类加载功能，同时提供了最重要的 `define*` 系列的 `native` 接口，扩展类加载器与系统类加载器最终也是依靠 `java.lang.ClassLoader` 的本地接口方能完成 Java 类的加载。所以，可以说，扩展类加载器与系统类加载器仅仅是张皮而已，`java.lang.ClassLoader` 才是真正的“幕后主事”。

事实上，从 JDK 研发者的角度看，最初的 JDK 根本就没有所谓的类加载器这个概念，JDK 的核心类库直接通过调用 `ClassFileParser::parseClassFile()` 接口完成加载，而对于 Java 应用程序中的类库，则绕个弯子，通过调用 `native` 接口从而间接调用 `ClassFileParser::parseClassFile()` 接口完成加载。只不过为了 Java applet 而专门开发了类加载器。只可惜 Java applet 没有生根发芽，但是类加载器倒是遍地开花，生活得很好。看来技术也有“听天由命”的一面，有心栽花花不开，无心插柳柳成荫。

Java 体系所定义的 3 种类加载器——引导类加载器、扩展类加载器和系统类加载器，与开发者自定义的类加载器，它们之间存在一定的关系，这种关系如图 10.5 所示。

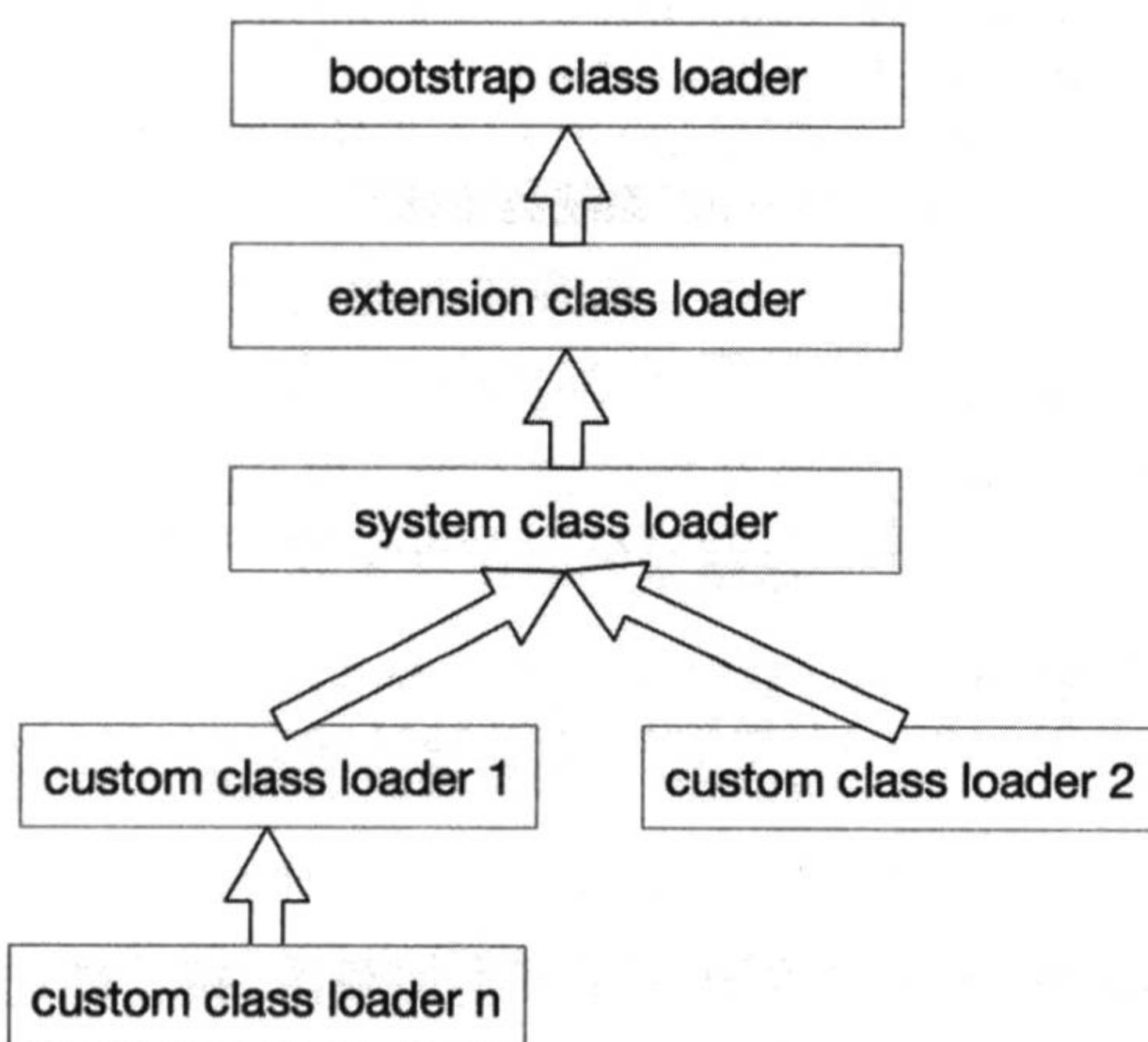


图 10.5 Java 类加载器之间的委派关系

图 10.5 所示的各种类加载器之间的关系，并非 Java 类面向对象的三大特性之一的那种“继承”关系，毕竟引导类加载器是使用 C++ 语言编写而成的，Java 类编写的类加载器想去继承也无法继承。图 10.5 所表达的联系，仅仅是类的委托加载机制，尤其是其中著名的“双亲委派”机制，为世人所乐道，这种机制将在下文描述。虽然这几种类加载器之间并没有直接的继承关

系，但是扩展类加载器、系统类加载器及用户自定义的加载器，却都继承自 `java.lang.ClassLoader` 这个基类。前文讲解过该类，当 JVM 加载 Java 主类时，最终也是通过 `java.lang.ClassLoader.loadClass(String)` 这一接口完成的。

图 10.5 所表达的委托加载关系，本质上通过 `java.lang.ClassLoader.parent` 字段实现，该字段表示父加载器。对于引导类加载器，并没有所谓的父加载器的概念，因为引导类加载器本身是随 JVM 的启动而初始化，并且该类中的字段和方法全部是静态字段和方法，并不需要实例化便能使用。而除了引导类加载器之外的所有类加载器，由于都继承自 `java.lang.ClassLoader` 这个基类，因此便都拥有 `parent` 字段，由于该字段被 `final private` 修饰，因此子类只能通过调用 `java.lang.ClassLoader` 的构造函数才能初始化该字段。对于扩展类加载器和系统类加载器，在 JVM 第一次加载 Java 类时会被创建，并完成其父加载器的设定。扩展类加载器与系统类加载器在 `sun.misc.Launcher` 类的构造函数中完成初始化，`sun.misc.Launcher` 类的构造函数逻辑如下（仅摘录主要逻辑）：

清单：/src/sun/misc/Launcher.java

功能：构造函数

```
public class Launcher {
    private static URLStreamHandlerFactory factory = new Factory();
    private static Launcher launcher = new Launcher(); // 启动器
    private static String bootClassPath =
        System.getProperty("sun.boot.class.path"); // 引导类加载器路径
    private ClassLoader loader; // 系统类加载器

    public Launcher() {
        ClassLoader extcl;
        // 创建扩展类加载器
        extcl = ExtClassLoader.getExtClassLoader();

        // 创建系统类加载器
        loader = AppClassLoader.getAppClassLoader(extcl);
    }
}
```

在 `Launcher` 类的构造函数中，首先创建了扩展类加载器，接着创建了系统类加载器。注意，扩展类加载器是在构造函数中定义的局部变量 `extcl`，而系统类加载器则是 `Launcher` 类的成员变量 `loader`。为何扩展类加载器不是一个类成员变量，而是一个局部变量呢？当构造函数执行完毕，这个扩展类实例变量不就被销毁了吗，那么 JVM 还如何使用该扩展类去加载扩展包呢？其实，当 `Launcher` 类的构造函数执行完之后，扩展类加载器实例对象并不会被 GC 回收，因为在创建系统类加载器的时候，扩展类加载器被设置为系统类加载器的 `parent`，因此当 JVM 想加载扩展类时，总是能够通过系统类加载器的 `parent` 属性获取到扩展类加载器，从而使用扩展类加

载器去加载相应的类库。

在 Launcher 类的构造函数中，调用 ExtClassLoader.getExtClassLoader()接口创建扩展类加载器，该接口实现如下（仅摘录主要逻辑）：

清单：/src/sun/misc/Launcher.java

功能：创建扩展类加载器

```
static class ExtClassLoader extends URLClassLoader {
    // 创建扩展类加载器
    public static ExtClassLoader getExtClassLoader() throws IOException
    {
        final File[] dirs = getExtDirs(); // 获取扩展类加载路径

        return new ExtClassLoader(dirs); // 实例化扩展类加载器
    }

    // 获取扩展类加载文件
    private static File[] getExtDirs() {
        String s = System.getProperty("java.ext.dirs");
        File[] dirs;
        // ...
        return dirs;
    }
}
```

在 ExtClassLoader.getExtClassLoader()接口中先获取扩展类加载路径，最后直接通过 new ExtClassLoader(dirs)来实例化一个扩展类加载器。在 ExtClassLoader(File[])构造函数中，调用 super(getExtURLs(dirs), null, factory)父类构造函数来实例化一个加载器，注意，在调用父类构造函数时，所传入的第 2 个人参是 null，该人参最终会被赋值给 parent 属性。由此可见，扩展类加载器的 parent 父加载器是 null，并不是引导类加载器。

分析完扩展类加载器，再看系统类加载器的创建，其原理大同小异，需要注意的是，在 Launcher 类构造函数中实例化系统类加载器时，将刚刚创建的扩展类加载器作为人参传递给了 AppClassLoader.getAppClassLoader(final ClassLoader)接口，该接口最终会将系统类加载器的 parent 属性设置为扩展类加载器。因此系统类加载器的父加载器是扩展类加载器，但是扩展类加载器的父加载器是 null。

既然系统类和扩展类加载器都是在 Launcher 类的构造函数中才得以创建，那么 Launcher 类是在什么时间点被实例化呢？这是一个问题，该问题后文会讲。

JVM 在启动过程中，除了会进行扩展类加载器与系统类加载器的实例化，也会进行引导类加载器的初始化。引导类加载器便是上文所展示的使用 C++语言编写的 ClassLoader 类，该类提

供了 initialize() 接口，该接口在 JVM 的 init() 初始化链路中被调用，该接口会读取引导类路径，定位到相关的 jar 文件，为加载核心类库做准备。

10.4.2 系统类加载器与扩展类加载器创建

上文讲过，系统类加载器与扩展类加载器在 sun.misc.Launcher 的构造函数中被创建，但是 sun.misc.Launcher 类又在何时被创建呢？这要从 Java 主类的加载说起。前面讲过，JVM 加载 Java 主类时，会走下面这条链路：

```
java.c::JavaMain()
java.c::LoadClass()
jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name))
jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name))
jvm.cpp::find_class_from_class_loader()
SystemDictionary::resolve_or_fail()
// .....
jvm.cpp::jvm_define_class_common()
SystemDictionary.cpp::resolve_from_stream()
ClassFileParser.cpp::parseClassFile()
```

这条链路的第 3 步会进入 jni_FindClass() 入口，当 JVM 加载 Java 应用程序主类时，该入口最终会调用 loader = Handle(THREAD, SystemDictionary::java_system_loader()) 来获取类加载器，sun.misc.Launcher 类的实例便会在 SystemDictionary::java_system_loader() 链路里创建。

这里的 SystemDictionary::java_system_loader() 返回 SystemDictionary._java_system_loader，这便是传说中的“系统类加载器”。SystemDictionary._java_system_loader 成员变量在 JVM 启动期间通过调用 SystemDictionary::compute_java_system_loader() 函数进行初始化，该函数实现如下：

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

功能：compute_java_system_loader() 函数实现

```
void SystemDictionary::compute_java_system_loader(TRAPS) {
    KlassHandle system_klass(THREAD, WK_KLASS(ClassLoader_klass));
    JavaValue result(T_OBJECT);
    JavaCalls::call_static(&result,
        KlassHandle(THREAD, WK_KLASS(ClassLoader_klass)),
        vmSymbols::getSystemClassLoader_name(),
        vmSymbols::void_classloader_signature(),
        CHECK);
    _java_system_loader = (oop) result.get jobject();
}
```

该函数通过调用 JavaCalls::call_static() 函数来完成 _java_system_loader 变量的初始化。

JavaCalls::call_static()函数在 JVM 内部也是调用频率很高的一个接口，大凡涉及 Java 类静态方法调用时，最终都会经过本接口调用。该接口顾名思义，主要作用是调用 Java 类的静态方法，该接口的第 1~4 个人参含义分别如下：

- ◎ JavaValue* result，储存调用 Java 类静态方法后的返回值。
- ◎ KlassHandle klass，所调用的目标 Java 类在 JVM 内部的对等结构体。
- ◎ Symbol* name，所调用的目标 Java 类静态方法的名称。
- ◎ Symbol* signature，目标 Java 类静态方法的签名。

众所周知，Java 程序的方法一定是被封装在 Java 类中，所以要调用一个 Java 方法，必定要知道类对象、类名和方法签名，静态方法也不例外，由此便能理解为何调用 JavaCalls::call_static() 函数需要传递上面这几个人参。

SystemDictionary::compute_java_system_loader()函数调用 JavaCalls::call_static()函数时，所传入的第 2 个参数是 KlassHandle(THREAD, WK_KLASS(ClassLoader_klass))，其中，ClassLoader_klass 在 systemDictionary.hpp 中通过宏定义，如下：

```
template(ClassLoader_klass, java_lang_ClassLoader, Pre)
```

其实这里所定义的宏比较长，上面这一行仅仅是其中一行，这一点在后面讲解“预加载”的时候再描述。总之，通过这个宏可以知道，ClassLoader_klass 最终映射到的 Java 类是 java.lang.ClassLoader 类。由此可知，JavaCalls::call_static()函数最终要调用的目标方法所在的类是 java.lang.ClassLoader 类。

SystemDictionary::compute_java_system_loader()函数调用 JavaCalls::call_static()函数时，所传入的第 3 个参数是 vmSymbols::getSystemClassLoader_name()，在 vmSymbols.hpp 中，通过下面这个宏定义了 getSystemClassLoader_name()函数的返回值：

```
template(getSystemClassLoader_name, "getSystemClassLoader")
```

由此可知，getSystemClassLoader_name()函数返回“getSystemClassLoader”这个字符串标识。而该标识将作为 JavaCalls::call_static()函数的第 3 个人参，该人参正是 JavaCalls::call_static() 函数最终要调用的目标方法。综合上面第 2 和第 3 这两个人参可知，SystemDictionary::compute_java_system_loader() 将通过调用 java.lang.ClassLoader 类的静态方法 getSystemClassLoader() 来获取类加载器，最终 JVM 将使用这个加载器去加载 Java 程序的主类。

java.lang.ClassLoader.getSystemClassLoader() 函数主要通过调用 java.lang.ClassLoader.initSystemClassLoader() 接口来初始化系统类加载器，initSystemClassLoader() 接口的主要逻辑如下：

清单：/src/java/lang/ClassLoader

功能：系统类加载器初始化

```
private static synchronized void initSystemClassLoader() {
    if (!sclSet) {
        if (scl != null)
            throw new IllegalStateException("recursive invocation");
        sun.misc.Launcher l = sun.misc.Launcher.getLauncher();
        if (l != null) {
            Throwable oops = null;
            scl = l.getClassLoader();
            try {
                scl = AccessController.doPrivileged(
                    new SystemClassLoaderAction(scl));
            } catch (PrivilegedActionException pae) {
                // ...
            }
            if (oops != null)
                // ...
        }
    }
    sclSet = true;
}
```

在这个函数中，终于看到了 `sun.misc.Launcher` 类的实例化，其实例化通过调用 `sun.misc.Launcher.getLauncher()` 得以完成，该函数直接返回 `Launcher.launcher` 静态变量，而该静态变量在定义时便实例化了，如下：

```
private static Launcher launcher = new Launcher();
```

而系统类加载器和扩展类加载器都是在 `Launcher` 类的构造函数中被创建的，由此 JVM 便完成了系统类加载器和扩展类加载器的创建。

10.4.3 双亲委派机制与破坏

凡是接触过 Java 类加载器的小伙伴，必定知道 JVM 中的双亲委派机制。该机制在 `java.lang.ClassLoader.loadClass(String, boolean)` 接口中，该接口实现在前文讲解 JVM 加载 Java 主类时贴出来过，为了节省篇幅，这里不再贴出了，有不清楚的小伙伴可回头找码看。该接口的逻辑如下：

(1) 先在当前加载器的缓存中查找有无目标类，如果有，直接返回。

(2) 判断当前加载器的父加载器是否为空，如果不为空，则调用 `parent.loadClass(name, false)`

接口进行加载。

(3) 反之，如果当前加载器的父类加载器为空，则调用 `findBootstrapClassOrNull(name)` 接口，让引导类加载器进行加载。

(4) 如果通过以上 3 条路径都没能成功加载，则调用 `findClass(name)` 接口进行加载。该接口最终会调用 `java.lang.ClassLoader` 接口的 `define*` 系列的 native 接口加载目标 Java 类。

双亲委派的模型就隐藏在这第 2 和第 3 步中。在理解双亲委派机制之前，有一点需要先说明，那就是 JVM 中的所有类加载都会通过 `java.lang.ClassLoader.loadClass(String)` 接口（自定义类加载器并重写 `java.lang.ClassLoader.loadClass(String)` 接口的除外），连 JDK 的核心类库也不能例外，虽然核心类库会存在“预加载”的行为，这一点下文会详细分析。知道了这一点，方能理解双亲委派模型。假设当前加载的是 `java.lang.Object` 这个类，很显然，该类属于 JDK 中核心得不能再核心的一个类，因此一定只能由引导类加载器进行加载。当 JVM 准备加载 `java.lang.Object` 时，JVM 默认会使用系统类加载器去加载，按照上面 4 步加载的逻辑，在第 1 步从系统类的缓存中肯定查找不到该类，于是进入第 2 步。由于从系统类加载器的父加载器是扩展类加载器，于是扩展类加载器继续从第 1 步开始重复。由于扩展类加载器的缓存中也一定查找不到该类，因此进入第 2 步。扩展类的父加载器是 null(前文刚讲过)，因此系统调用 `findClass(String)`，最终通过引导类加载器进行加载。这便是双亲委派机制，这种机制保证核心类库一定是由引导类加载器进行加载，而不会被多种加载器加载，否则每个加载器都会加载一遍核心类库，世界要大乱了，同时也会存在安全隐患。

双亲委派从本质上而言，其实规定了类加载的顺序是：引导类加载器先加载，若加载不到，由扩展类加载器加载，若还加载不到，才会由系统类加载器或自定义的类加载器进行加载。

不过，可能聪明的你会想到，如果在自定义的类加载器中重写 `java.lang.ClassLoader.loadClass(String)` 或 `java.lang.ClassLoader.loadClass(String, boolean)` 方法，抹去其中的双亲委派机制，仅保留上面这 4 步中的第 1 步与第 4 步，那么是不是就能够加载核心类库了呢？这也不行！因为 JDK 还为核心类库提供了一层保护机制。不管是自定义的类加载器，还是系统类加载器抑或扩展类加载器，最终都必须调用 `java.lang.ClassLoader.defineClass(String, byte[], int, int, ProtectionDomain)` 方法，而该方法会执行 `preDefineClass()` 接口，该接口中提供了对 JDK 核心类库的保护，其实现如下：

清单：/src/java/lang/ClassLoader

功能：核心类库保护

```
private ProtectionDomain preDefineClass(String name, ProtectionDomain pd) {
    if (!checkName(name))
        throw new NoClassDefFoundError("IllegalName: " + name);
```

```

    if ((name != null) && name.startsWith("java.")) {
        throw new SecurityException
            ("Prohibited package name: " + name.substring(0,
name.lastIndexOf('.')));
    }
    if (pd == null) {
        pd = defaultDomain;
    }

    if (name != null) checkCerts(name, pd.getCodeSource());
}

return pd;
}

```

在该接口中会判断所要加载的目标类的全限定名是否以“java.”开始，如果是，则直接抛出异常。这便是所有的自定义类加载器都只能重写 `java.lang.ClassLoader.findClass(String)` 接口的原因。

10.4.4 预加载

JVM 启动期间，会先加载一部分核心类库，这部分核心类库包括：

清单：/src/share/vm/classfile/systemDictionary.hpp

功能：定义预加载的类

```

#define WK_KLASSES_DO(template)
/* well-known classes */
template(Object_klass,
         java_lang_Object,           \
         Pre) \
template(String_klass,
         java_lang_String,          \
         Pre) \
template(Class_klass,
         java_lang_Class,           \
         Pre) \
template(Cloneable_klass,
         java_lang_Cloneable,        \
         Pre) \
template(ClassLoader_klass,
         java_lang_ClassLoader,      \
         Pre) \
template(Serializable_klass,
         java_io_Serializable,       \
         Pre) \
template(Thread_klass,
         java_lang_Thread,           \
         Pre) \
template(ThreadGroup_klass,
         java_lang_ThreadGroup,       \
         Pre) \
template(Properties_klass,
         java_util_Properties,       \
         Pre) \
template(reflect_AccessibleObject_klass,
         java_lang_reflect_AccessibleObject, \
         Pre) \
template(reflect_Field_klass,
         java_lang_reflect_Field,       \
         Pre) \
template(reflect_Method_klass,
         java_lang_reflect_Method,       \
         Pre) \
template(Float_klass,
         java_lang_Float,             \
         Pre) \
template(Double_klass,
         java_lang_Double,            \
         Pre) \
template(Byte_klass,
         java_lang_Byte,              \
         Pre) \
template(Short_klass,
         java_lang_Short,             \
         Pre) \
template(Integer_klass,
         java_lang_Integer,            \
         Pre) \
template(Long_klass,
         java_lang_Long,              \
         Pre) \

```

```
// ...
/*end*/
```

可以看到，平时开发中最常用的基础类库，基本都在这里面了，例如 Object、Long、String、Serializable 及 Thread 等。也正是因为这些类使用频率非常高，即使是一个非常简单的 Java 程序都可能用到这些类中的大部分类，因此不如预先将其加载到内存。但是同时也应该看到，放眼 JDK 的整个核心类库，这几个类也是凤毛麟角，这是因为预加载的类越多，则 JVM 的启动过程将越慢，反而不美。

核心类库的加载是可以被观察到的，只需要在 Java 命令行上加上-XX:+TraceClassLoading 选项，JVM 启动时便会打印类似下面这样的跟踪类加载的日志：

```
[Opened
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Object from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.io.Serializable from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Comparable from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.CharSequence from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.String from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
// ...
```

JDK 核心类库随 JVM 启动而进行预加载的链路如下：

```
java.c::main()
    java_md.c::LoadJavaVM()
        jni.cpp::JNI_CreateJavaVM()
            Threads::create_vm()
                init.cpp::init_globals()
                    Universe.cpp::universe2_init()
                        Universe::genesis()
                            SystemDictionary::initialize()
                                SystemDictionary::initialize_preloaded_classes()
                                    SystemDictionary::initialize_wk_klasses_until()
```

在 SystemDictionary::initialize_wk_klasses_until() 函数中，遍历 WK_KLASSES_DO 宏中所定义的全部预加载的类，并调用 SystemDictionary::initialize_wk_klass() 函数对这些类进行逐个加载。SystemDictionary::initialize_wk_klass() 函数实现如下：

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

功能：JDK 预加载核心类库

```
bool SystemDictionary::initialize_wk_klass(WKID id, int init_opt, TRAPS) {
```

```

// ...
if ((*klassp) == NULL && try_load) {
    if (must_load) {
        (*klassp) = resolve_or_fail(symbol, true, CHECK_0); // load required class
    } else { // ... }
}
return ((*klassp) != NULL);
}

```

在该函数中，最终调用 SystemDictionary::resolve_or_fail()接口去加载核心类库。在前文分析 JVM 加载 JAVA 主类时，其链路也会经过这里，该函数会一路调用，最终调用到 SystemDictionary::load_instance_class()函数。到了这个函数中，开始体现出“双亲委派”的机制。该函数根据类加载器变量 class_loader 是否为 null，被分成了 if{} 和 else{} 这两大分支，如果是 null 则进入 if{} 逻辑块，否则进入 else{} 逻辑块。由于 JVM 在启动期间加载核心类库时，核心类库压根儿没有所谓的“类加载器”概念，因此核心类库的预加载流程直接进入 if{} 逻辑块，而前文在分析 JVM 加载 Java 应用程序主类时，其链路进入了 else{} 逻辑块。在 if{} 逻辑块中，JVM 会调用内部的引导类加载器来加载核心类库，而在 else{} 逻辑块中，则会通过 JavaCalls::call_virtual()接口来加载类，而这个接口最终会通过 java.lang.ClassLoader.loadClass(String)接口执行类的加载。这便是 jvm 内部的“双亲委派”机制，一边是直接使用引导类加载器，一边则是使用系统类加载器。在 if{} 语句块中，可以看到 JVM 调用了 ClassLoader::load_classfile()接口，这是引导类加载器的加载接口。

由于 JDK 核心类库的预加载直接使用了引导类加载器，不经过 java.lang.ClassLoader.loadClass(String)接口，因此如果在这个接口内部打断点，是无法看到核心类库的加载过程的。

正是由于 JVM 启动期间会对部分核心类库进行预加载，因此 JVM 在加载 Java 应用程序主类的链路中对 sun.misc.Launcher 类进行初始化时，才能直接获取到 java.lang.ClassLoader 类在 JVM 内部对应的实例对象，并调用其静态方法 getSystemClassLoader() 来初始化系统类加载器和扩展类加载器。

10.4.5 引导类加载

JVM 使用“双亲委派”机制加载类，如果所加载的类属于 JDK 核心类库中所定义的类，则 java.lang.ClassLoader.loadClass(String, boolean) 会进入 findBootstrapClassOrNull() 方法，通过引导类加载核心类库。该方法最终会调用 private native Class<?> findBootstrapClass(String name) 这个本地接口执行核心类库加载。这个本地接口最终会调用引导类加载器执行加载。开放的 JDK 源

码中包含这个本地接口的实现，如下（仅摘录核心逻辑）：

```
清单: /src/share/native/java/lang/ClassLoader.c
功能: JNI 本地接口加载核心类库接口
JNIEXPORT jclass JNICALL
Java_java_lang_ClassLoader_findBootstrapClass(JNIEnv *env, jobject loader,
                                                jstring classname)
{
    char *clname;
    jclass cls = 0;
    char buf[128];

    clname = getUTF(env, classname, buf, sizeof(buf));
    cls = JVM_FindClassFromBootLoader(env, clname);
    return cls;
}
```

在 JNI 实现中，调用 `JVM_FindClassFromBootLoader()` 来加载核心类库。而在 `JVM_FindClassFromBootLoader()` 接口中，则直接调用 `SystemDictionary::resolve_or_null()` 函数，该函数最终也会走到 `SystemDictionary::load_instance_class()` 函数中。关于该函数，前面多次提到过，无论是 JVM 在启动期间预加载部分核心类库还是 JVM 加载 Java 应用程序主类，只要涉及类的加载，最终都会经过该函数。而在上文也提到过，该函数内部也使用了“双亲委派”机制，如果 `class_loader` 为空，则直接调用引导类加载器的 `ClassLoader::load_classfile()` 接口进行类加载，否则，仍然要通过所指定的 Java 类加载器去加载。由于在加载核心类库时，类加载器为空，因此核心类库的加载最终都会通过调用引导类加载器接口进行加载。

10.4.6 加载、链接与延迟加载

看下面这个测试程序：

```
清单: /Test.java
功能: 核心类库加载调试
public class Test {
    public static void main(String[] args) {
        java.lang.Long l1 = 3000L;
    }
}
```

在 `java.lang.ClassLoader.findBootstrapClassOrNull(String)` 中打上断点，调试时会发现 `java.lang.Long` 这个核心类仍然会经过该断点，这说明 `java.lang.Long` 核心类仍然通过 `java.lang.ClassLoader` 进行加载。但是前面讲过，JVM 在启动时会预加载一部分核心类库，包括

java 基本类型的包装类，因此 `java.lang.Long` 这种核心类也一定早就预加载好了，可是为何在测试程序中实例化 `Long` 类型时，仍然要再次加载一遍呢？其实，之所以会再次走加载流程，并不是因为碰到了类加载的指令，而是因为测试类所对应的常量池索引尚未转换成直接对象引用，更准确地说，是因为预加载的核心类库虽然已经完成了类的加载，但是由于尚未在应用程序中使用到，因此并未经链接，所以在 `new` 字节码指令的流程中便走了“慢分配”流程，这在后文讲解类实例化机制时会讲到。

众所周知，Java 类的生命周期一共分为 7 个阶段，其中加载阶段仅仅是其中第一步，加载完成之后需要进行链接和初始化。在链接阶段，字节码指令会被重写，将其所引用的常量池的索引号转换为直接引用。例如，在实例化一个类时，编译后所生成的字节码指令如下：

```
new #2
```

`new` 指令后面跟随的#2 表示引用常量池中索引号为 2 的元素，该元素一定指向某个 Java 类的全限定名。如果是实例化 `Long`，则常量池 2 号索引指向的字符串一定如下：

```
class java/lang/Long
```

在 Java 类经编译后所得到的字节码文件中的原始常量池中，`class java/lang/Long` 并不是存放于常量池的一个元素中而是分开存放于好几个元素中，例如，`class` 本身是常量池中的一种固定的类型，而 `java/lang/Long` 在常量池中则是一种字符串类型，UTF-8 格式的字符串。当 JVM 加载完测试类 `Test` 之后，会对其字节码进行重写，重写后的 `new` 字节码指令，其后面所跟随的便是直接指向“`java/lang/Long`”这个字符串的内存地址，这个重写便是整个 Java 类生命周期中“链接”阶段所做的最重要的事情。其实重写字节码的过程，可以认为是做了一次缓存的优化，通过重写，避免运行期再去将多个不同的常量池元素一步步拼装出 `new` 指令实际要指向的类型。

等到 JVM 真正运行到 `new` 这条字节码指令时，JVM 为了加快指令执行速度，又做了一次缓存。这一次缓存的是啥呢？众所周知，在 Java 语言中，`new` 指令的作用很简单也很唯一，就是实例化一个类型，或者数组。当使用 `new` 实例化一个 Java 类型时，JVM 必须要知道其所实例化的是何种类型，这个问题在 Java 类链接阶段便已解决了，通过字节码指令重写，JVM 知道所要实例化的 Java 类的全限定名。但是仅仅这样仍然不够，放眼 JVM 执行 `new` 指令的过程，JVM 需要根据 Java 类的全限定名称，在内存 perm 区（JDK 8 中是 metaspace 区）定位到这个 Java 类在内存中的对等体——`instanceKlass`，`instanceKlass` 作为 Java 类在内存中的对等体，包含了原始 Java 类中的一切信息，JVM 只有找到了 `instanceKlass`，才能根据这个类模板创建出 Java 类的实例对象。在 JVM 开始执行 `new` 指令所对应的实例对象创建过程之前，一定会先完成类的加载、链接和初始化，`instanceKlass` 便在类的加载阶段完成构建，因此 JVM 根据 Java 类的全限定名称一定能够定位到 `instanceKlass` 这个类模板。但是如果每次执行 `new` 指令都要根据 Java 类的全限定名称去定位到 `instanceKlass` 这个内存对象，未免会做重复的事情，浪费机器性能，

为了避免这种情况，JVM 在第一次执行 new 指令时，便会将定位到的 instanceKlass 缓存起来，这样如果程序后续需要再次实例化同样的 Java 类对象时，便直接从缓存中读取 instanceKlass，直接据此创建 Java 类实例对象，从而提升效率。前文曾提及，使用 new 指令加载类时，如果类尚未被加载或者未被链接过，则会进入慢分配流程（后文会细讲类分配机制），从而会进入 InterpreterRuntime::new() 函数，而 InterpreterRuntime::new() 函数则会调用 constantPool->klass_at() 接口来获取 new 指令后面的 Java 类模板，该接口最终调用 constantPoolOopDesc::klass_at_impl() 函数，该函数实现如下（仅摘录主要逻辑）：

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

功能：Java 类模板缓存

```

klassOop constantPoolOopDesc::klass_at_impl(constantPoolHandle this_oop, int
which, TRAPS) {
    // 从常量池中取指定位置的元素
    CPSlot entry = this_oop->slot_at(which);

    // 第一次执行 new 指令时，不会进入该分支
    if (entry.is_oop()) {
        return (klassOop)entry.get_oop();
    }

    // ...
    { ObjectLocker ol(this_oop, THREAD);
        // 获取类加载器
        loader = Handle(THREAD, instanceKlass::cast(this_oop->pool_holder())->
class_loader());
    } // unlocking constantPool

    // ...
    if (do_resolve) {
        // 执行类加载
        klassOop k_oop = SystemDictionary::resolve_or_fail(name, loader, h_prot,
true, THREAD);

        // ...
        if (TraceClassResolution && !k()->klass_part()->oop_is_array()) {
            // ...本逻辑块处理 new 数组的情况
        } else { // 如果不是实例化数组，则一定是 Java 类

            // 再次判断目标类是否已缓存，可能其他 Java 线程会先于本线程执行 new 操作
            do_resolve = this_oop->tag_at(which).is_unresolved_klass();

            if (do_resolve) {
                // 将目标类模板缓存到常量池中，下次执行 new 指令时直接从这里取
        }
    }
}

```

```

        this_oop->klass_at_put(which, k());
    }
}

// 从常量池指定位置的元素中获取类模板
return (klassOop) entry.get_oop();
}

```

上面这段代码的主要思路是，先从缓存中读取 Java 类模板，如果读取不到，则执行类加载，加载之后，将类模板写入缓存，最终仍然从缓存中读取类模板并返回。这与 java 应用程序中的缓存读写逻辑何其相似！

在该逻辑中，如果目标 Java 类模板尚未被解析，则 JVM 会调用 SystemDictionary::resolve_or_fail() 接口先执行类加载。前文已经讲过该接口所经过的链路，该接口最终会调用 SystemDictionary::load_instance_class() 函数执行类加载，对于 SystemDictionary::load_instance_class() 函数，前面分析过其实现机制，其内部也使用了一个类似于“双亲委托”的机制。当类加载器为 null 时，则直接加载核心类库，否则最终仍然会调用 java.lang.ClassLoader.loadClass(String) 这个 Java 接口去执行类加载。由于在本示例程序中，通过执行 new 指令进入了 SystemDictionary::load_instance_class() 函数，而 Java 程序的默认类加载器便是系统类加载器，因此 SystemDictionary::load_instance_class() 函数最终仍然通过 java.lang.ClassLoader.loadClass(String) 这个 Java 接口去执行类加载。这是为何在 java.lang.ClassLoader.findBootstrapClassOrNull(String) 中打上断点，而调试时会发现 java.lang.Long 这个核心类仍然会经过该断点进行加载的原因。

不过前面也讲过，java.lang.Long 在 JVM 启动期间便被预加载，因此最终通过 java.lang.ClassLoader.findBootstrapClassOrNull() 接口调用引导类加载器去加载核心类库时，JVM 内部并未真的重新将 java.lang.Long 加载一遍，而是直接从缓存中读取。

将上面的测试示例稍微修改一下，变成如下：

清单：/Test.java

功能：核心类库加载调试

```

public class Test {
    public static void main(String[] args) {
        java.lang.Long l1 = 3000L;
        java.lang.Long l11 = 5113L;
    }
}

```

在本示例程序中，连续两次实例化了 Long 类型，在 java.lang.ClassLoader.

findBootstrapClassOrNull(String)中打上断点，调试时会发现 `java.lang.Long` 这个核心类的加载仅在执行本示例程序的第一个 `Long` 实例化代码时才会经过该断点，而第二次实例化 `Long` 类型时则不会再经过该断点，道理其实很简单，因为第一次实例化之后，`java.lang.Long` 这个类模板已经完成解析，所以第二次再次实例化时，JVM 便会尝试走“快速分配”流程进行无锁分配，如果该类不满足快速分配的条件（例如类太大），虽然仍然会进入慢分配流程，但是由于已经解析过，因此在慢分配阶段并不会经历类加载过程。

为了证明在对 JDK 核心类库中被预加载的类执行 `new` 实例化操作时，JVM 没有重复加载类，在断点测试时开启-XX:+TraceClassLoading 选项，查看本示例程序中的 `Long` 类型是何时被加载的。

事实上，不仅 JDK 核心类的加载如此，所有的 Java 类的实例化都是如此，各位道友可以自由编写测试程序进行断点调试。很多书籍和网络博客都说 Java 类的加载、链接和初始化并没有严格的顺序，可以混着来，其实道理便在这里。

从更广义的角度看，其实这也体现了类的延迟加载机制——一个类，只有当真正被使用时才会被加载，即使程序中 `import` 了某个类，但是如果不用，则 JVM 在整个运行期都不会加载它。甚至，即使在程序中使用到了某个类，但是如果使用条件一直不符合，导致 JVM 一直没有进入使用的分支流程中，则 JVM 也自始至终都不会加载这个类。例如下面这段示例代码：

```
if(...){  
    new Test();  
}
```

如果整个程序只有这一处地方用到了 `Test` 类，但是程序在运行期从来都没有进入过这里的 `if` 分支流，则 `Test` 类不会被加载。不用说，延迟加载机制避免了系统无谓的开销。

10.4.7 父加载器

前面讲过，JVM 默认提供 3 种类加载器：引导类加载器、扩展类加载器和系统类加载器。其中，系统类加载器的父加载器是扩展类加载器，而扩展类加载器与引导类加载器的父加载器都是 `null`，这是 JVM 中固化下来的关系设定。默认情况下，Java 应用程序的类加载器是系统类加载器。

下面这个测试程序可以验证父加载器的相关理论：

清单：/Test.java

功能：父加载器

```
public class Test {  
    public static void main(String[] args) throws Exception {
```

```
Test t = new Test();

System.out.println("Test.classLoader is " + t.getClass().getClassLoader());
System.out.println("Test.classLoader.parentClassLoader is "
    + t.getClass().getClassLoader().getParent());

System.out.println("Test.classLoader.parentClassLoader.parentClassLoader is "
    + t.getClass().getClassLoader().getParent().getParent());
}

}
```

运行后，打印结果如下：

```
Test.classLoader is sun.misc.Launcher$AppClassLoader@2d3fcdbd
Test.classLoader.parentClassLoader is sun.misc.Launcher$ExtClassLoader@7225790e
Test.classLoader.parentClassLoader.parentClassLoader is null
```

根据该结果可知，Test 测试类的加载器是 Launcher\$AppClassLoader，而这正是系统类加载器。同理，Test 测试类加载器的父加载器是 Launcher\$ExtClassLoader，这是扩展类加载器。通过这里的打印结果可知，扩展类的父加载器的确是 null。而至于 Test 类的加载器为何是系统类加载器，前文从源码级别进行了分析。

前面讲过，JDK 核心类库中的类由引导类加载器负责加载，那么如果在测试程序中加载一个核心类，然后打印其加载器，会打印出啥呢？测试程序很简单，如下：

清单： /Test.java

功能：父加载器

```
public class Test {  
    public static void main(String[] args) throws Exception {  
        Object obj = new Object();  
        System.out.println("Object.classLoader is " +  
obj.getClass().getClassLoader());  
    }  
}
```

该测试程序十分简单，仅仅实例化了一个 `java.lang.Object` 类。该类绝对是 JDK 核心类库中的核心类。

打印结果是.

Object.classLoader is null

不是说好核心类库的加载器是引导类加载器吗，为何这里却打印出一个空值呢？道理其实很简单，站在程序的角度看，引导类加载器与另外两种类加载器——系统类加载器和扩展类加载器，并不是同一个层次意义上的加载器，引导类加载器是使用 C++语言编写而成的，而另外

两种类加载器则是使用 Java 语言编写而成的。由于引导类加载器压根儿就不是一个 Java 类，因此在 Java 程序中只能打印出空值。

JVM 提供自定义类加载器的接口，开发者只需继承 `java.lang.ClassLoader` 或者其子类（例如 `java.net.URLClassLoader`），重写相关接口，便能自定义类加载器。例如下面的示例：

清单：/Test.java

功能：自定义类加载器

```

public static void main(String[] args) throws Exception {
    // 自定义类加载器
    ClassLoader loader = new ClassLoader() {
        @Override
        public Class<?> loadClass(String name) throws ClassNotFoundException {
            try {
                String className =
name.substring(name.lastIndexOf(".") + 1) + ".class";
                InputStream is = getClass().getResourceAsStream(className);
                if(is == null){
                    return super.loadClass(name);
                }
                byte[] buffer = new byte[is.available()];
                is.read(buffer);
                return defineClass(name, buffer, 0, buffer.length);
            } catch (Exception e) {
                throw new ClassNotFoundException(name);
            }
        }
    };
    // 使用自定义类加载器加载 Test 测试类
    Object test = loader.loadClass("Test").newInstance();
    System.out.println("Test.classLoader is " + test.getClass().
getClassLoader());
    System.out.println("Test.classLoader.parentClassLoader is " +
test.getClass().getClassLoader().getParent());
    System.out.println("Test.classLoader.parentClassLoader.parentClassLoader is " +
test.getClass().getClassLoader().getParent().getParent());
}

```

运行该测试程序，打印结果如下：

```

Test.classLoader is Test$1@36f6e879
Test.classLoader.parentClassLoader is sun.misc.Launcher$AppClassLoader@2d3fcdbd
Test.classLoader.parentClassLoader.parentClassLoader is
sun.misc.Launcher$ExtClassLoader@531be3c5

```

通过打印结果可知，Test 类的加载器是 Test\$1@36f6e879，这里的 Test\$1 表示是 main 主函数中的局部变量，因为在本示例程序中，自定义的加载器被定义成局部变量。接着看这个自定义类加载器的父加载器，是 Launcher\$AppClassLoader，即系统类加载器。这里比较奇怪，明明自定义的类加载器直接继承了 java.lang.ClassLoader，并没有继承 Launcher\$AppClassLoader，但是为何自定义的类加载器的父加载器变成了系统类加载器呢？原因其实很简单，这个秘密隐藏在 java.lang.ClassLoader 的默认构造函数中，其默认的无参构造函数是：

清单：/src/java/lang/ClassLoader.java

功能：java.lang.ClassLoader 的默认构造函数

```
protected ClassLoader() {
    this(checkCreateClassLoader(), getSystemClassLoader());
}
```

通过这个默认构造函数可知，如果自定义的类加载器没有重写构造函数，则该默认构造函数会将 getSystemClassLoader() 返回的结果作为自定义类加载器的父加载器。而 getSystemClassLoader() 函数在前文已经讲过，最终返回的便是系统类加载器——Launcher\$AppClassLoader。

10.4.8 加载器与类型转换

经过上述分析可知，每个 Java 类都必定有一个类加载器，JDK 核心类库的加载器是引导类加载器，应用程序中的类加载器默认是系统类加载器，除此以外，开发者还可以为应用程序开发自定义的加载器。例如 Tomcat 这类 Web 应用服务器，内部自定义了好几种类加载器，用于隔离同一个 Web 应用服务上的不同应用程序。

在一般情况下，使用不同的类加载器去加载不同的功能模块，会提高应用程序的安全性。但是，如果涉及 Java 类型转换，则加载器反而容易产生不美好的事情。在做 Java 类型转换时，只有两个类型都是由同一个加载器所加载，才能进行类型转换，否则转换时会发生异常。将上面自定义类加载器的那个示例稍微修改一下，变成如下：

清单：/Test.java

功能：自定义类加载器与类型转换

```
public static void main(String[] args) throws Exception {
    // 自定义类加载器
    ClassLoader loader = new ClassLoader() {
        @Override
        public Class<?> loadClass(String name) throws ClassNotFoundException {
            try {
                // ...
            } catch (Exception e) {
```

```

        throw new ClassNotFoundException(name);
    }
}

// 使用自定义类加载器加载 Test 测试类
Test test = (Test)loader.loadClass("Test").newInstance();
System.out.println("Test.classLoader is " + test.getClass().
getClassLoader());
System.out.println("Test.classLoader.parentClassLoader is " +
test.getClass().getClassLoader().getParent());
System.out.println("Test.classLoader.parentClassLoader.parentClassLoader is " +
test.getClass().getClassLoader().getParent().getParent());
}

```

这里修改了一行代码，原本在加载 Test 实例时，没有做类型转换，直接返回 Object 类型，而这里进行了类型转换。运行程序，最终会抛出下面这条异常：

```
Exception in thread "main" java.lang.ClassCastException: Test cannot be cast to Test
```

异常信息提示得很清楚：Test 类无法转换成 Test 类。乍一看，这个异常抛得莫名其妙，同一个类型竟然无法转换。但是实际上 JVM 并没有错，错的是你。在 Test test = (Test)loader.loadClass("Test").newInstance() 这句代码中，等号左边所声明的 Test 类型，由于测试程序并没有明确为其指定类加载器，因此 JVM 会使用系统类加载器加载 Test 类。而等号右边则明确使用了自定义的类加载器加载 Test 类型。因此等号左边与右边的两个 Test 类型的加载器并不是同一个，所以程序便抛出异常了。

ClassCastException 这类异常在使用 Spring 框架的 Java 应用程序中比较常见，相信很多道友都遇到过。究其原因，还是因为很多中间件内部都有自定义的类加载器，因此被内存加载器所加载的类型，无法直接转换为使用默认加载器加载的类型。

10.5 类实例分配

编写 Java 程序，使用最频繁的指令几乎就是 new 了，因为 Java 所有的数据和行为都封装在类中，想要读写数据或者触发某种行为，必须先实例化 Java 类。实例化 Java 类的方式有很多，例如，调用 `java.lang.Class.newInstance()` 或者直接使用 `new` 指令。前文在讲解类加载过程时对 `new` 的内部机制进行了局部介绍，而事实上，`new` 的实现机制相当精彩，对内存、线程并发控制等几乎都达到了登峰造极的地步，利用了软件和硬件所能利用的一切优化手段。并且，对象

实例化涉及内存分配，而内存分配又与垃圾收集器紧密耦合在一块，所以可以这么说，这部分的技术实现实在是精华中的精华。相比于执行引擎中对硬件指令的封装、运行期多层动态编译和指令实时优化的精微和深奥，内存分配则显得相当简单——什么技术够高大上，就直接拿来使用，为了性能和内存开销用尽一切谋略。

HotSpot 提供了 new 字节码指令的机器码实现，在 Intel 32 位平台上，其实现在 templateTable_x86_32.cpp::new() 函数中。该函数中的代码都是为了在运行期生成硬件相关的机器码。好在 HotSpot 保留了字节码解释器的实现，并且字节码解释器中的实现逻辑与机器码逻辑基本一致，所以为了方便，可以直接参考字节码解释器的逻辑来分析实例化的机制。字节码解释器的实现如下：

清单：/src/share/vm/interpreter/bytcodeInterpreter.cpp

功能：new 指令的实现机制

```
void BytecodeInterpreter::run(interpreterState istate) {
//...
run:
//...

CASE(_new): {
    u2 index = Bytes::get_Java_u2(pc+1);
    constantPoolOop constants = istate->method()->constants();

    // 如果目标 Java 类已经解析
    if (!constants->tag_at(index).is_unresolved_klass()) {
        oop entry = constants->slot_at(index).get_oop();
        klassOop k_entry = (klassOop) entry;
        instanceKlass* ik = (instanceKlass*) k_entry->klass_part();

        // 如果符合快速分配场景
        if (ik->is_initialized() && ik->can_be_fastpath_allocated() ) {
            size_t obj_size = ik->size_helper();
            oop result = NULL;
            bool need_zero = !ZeroTLAB;
            if (UseTLAB) {
                // 先通过 tlab 进行分配
                result = (oop) THREAD->tlab().allocate(obj_size);
            }

            // 如果 tlab 分配失败，则在 eden 区分配
            if (result == NULL) {
                need_zero = true;
                // Try allocate in shared eden
                retry:
                HeapWord* compare_to = *Universe::heap()->top_addr();
            }
        }
    }
}
```

```

// 指针碰撞分配法(bump -the-pointer)
HeapWord* new_top = compare_to + obj_size;
if (new_top <= *Universe::heap()->end_addr()) {
    if (Atomic::cmpxchg_ptr(new_top, Universe::heap()->top_addr(),
compare_to) != compare_to) {
        goto retry;
    }
    result = (oop) compare_to;
}
if (result != NULL) {

    // tlab 区清零
    if (need_zero) {
        HeapWord* to_zero = (HeapWord*) result + sizeof(oopDesc) / oopSize;
        obj_size -= sizeof(oopDesc) / oopSize;
        if (obj_size > 0) {
            memset(to_zero, 0, obj_size * HeapWordSize);
        }
    }
    if (UseBiasedLocking) {
        result->set_mark(ik->prototype_header());
    } else {
        result->set_mark(markOopDesc::prototype());
    }
    result->set_klass_gap(0);
    result->set_klass(k_entry);

    // 将对象地址压入操作数栈栈顶
    SET_STACK_OBJECT(result, 0);

    // 更新程序计数器 pc, 取下一条字节码指令, 继续处理
    UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1);
}

}

// 慢分配
CALL_VM(InterpreterRuntime::_new(THREAD, METHOD->constants(), index),
handle_exception);
SET_STACK_OBJECT(THREAD->vm_result(), 0);
THREAD->set_vm_result(NULL);
UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1);
}

//...
}

```

这段逻辑从宏观上分为两部分：一部分是快速分配，一部分则是慢分配。如果所要 new 的 Java 类型尚未被解析过（即使已经被加载也不算），则直接进入慢分配，这便是前文所讲述的 JVM 延迟加载的基础所在。快速分配的流程比较复杂，而慢分配则直接调用 InterpreterRuntime::_new() 接口。前文在讲解 Java 类加载时多次提到这个接口。

为了尽可能地加快内存分配速度，并减少并发操作带来的性能损失，JVM 在分配内存时，总是优先使用快速分配策略，当快速分配失败时，才会启用慢分配策略，这便是上面这段源码的总体逻辑，这段逻辑可以概括为如下几点：

- (1) 若 Java 类尚未被解析，则直接进入慢分配，不会使用快速分配策略。
- (2) 快速分配。如果没有开启栈上分配或不符合条件则会进行 TLAB 分配。
- (3) 快速分配。如果 TLAB 分配不成功，则尝试在 eden 区分配。
- (4) 如果 eden 区分配失败，则会进入慢分配流程。
- (5) 如果对象满足了直接进入老年代的条件，那就直接分配在老年代。
- (6) 快速分配。对于热点代码，如果开启逃逸分析，JVM 则会执行栈上分配或标量替换等优化方案。

第 6 点中的热点代码的逃逸分析并不包含在本逻辑中，下文会详解。

10.5.1 栈上分配与逃逸分析

前文在讲解 JVM 的 JIT 即时编译器时曾提到过逃逸分析。即时编译(just-in-time compilation, JIT) 是一种通过在运行时将字节码翻译为机器码，从而改善字节码编译语言性能的技术。在 HotSpot 中有多种实现：C1、C2 和 C1+C2，分别对应 Client、Server 和分层编译。未来究竟还会有何种更加精妙的实现谁也说不准。

而所谓逃逸，是指一个在方法内部被创建的对象不仅在方法内部被引用，还在方法外部被其他变量引用，这带来的后果是：在该方法执行完毕之后，该方法中创建的对象无法被 GC 回收，因为对象在方法外部还被引用着，这便是逃逸的含义。JVM 所进行的逃逸分析是确定方法内部所创建的对象会不会逃逸出方法体外部，如果确定不会逃逸出去，那么就能对该对象采用多种优化措施，这些优化措施主要围绕两大方面进行：内存分配和线程同步。

逃逸分析的算法主要基于连通图，通过引入连通图来构建对象和对象引用之间的可达性关系，并在此基础上，提出一种组合数据流分析法。该算法是上下文相关和流敏感的，同时模拟了对象任意层次的嵌套关系，所以运行时间比较长和内存消耗比较大(要感知上下文和程序流)，但是分析精度比较高。然而其并不能确保百分百的准确性，因为 Java 语言拥有许多动态特性，

例如动态生成字节码、调用本地函数、反射、方法拦截等，这些语言特性导致逃逸分析的算法不能作为编译期间的静态优化措施，而只能是基于运行时的动态分析，而这正是逃逸分析为何需要感知运行时的上下文和程序流的原因。一个对象在编译期可能并没有发生逃逸，但是可能被一个 AOP 框架拦截，结果就不好说了，很可能在运行期就会发生方法逃逸甚至线程逃逸。

由于逃逸分析是在运行期进行的，并且很耗费内存和 CPU 资源，因此 JVM 不可能对每一个方法里的变量都进行逃逸分析，所以其只能作为 JIT 的一项优化措施，即只有 JVM 触发 JIT 编译时才会进行逃逸分析。这也是为何在上面所贴出来的 BytecodeInterpreter::run() 函数源码中并没有看到丝毫与所谓栈上分配相关的实现的原因。在逃逸分析完成之后，JIT 编译器会基于逃逸分析的结果，直接基于 Java 字节码指令，生成优化后的本地机器码指令，所生成的本地机器指令会直接将 Java 对象分配在栈甚至硬件寄存器中。这些优化的本地机器指令已经再也看不到 Java 的 new 字节码指令了，而这也是并不能在 Java 的 new 字节码指令的直接实现里看到栈上分配相关的实现的原因。

清单：Test.java

功能：方法逃逸

```
public class Test {
    int k;

    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = new Test();
        t2.foo(t1);
    }

    public void foo(Test test) {
        Test t = test;
    }

    public void doSomething() {
    }
}
```

在该示例的 main() 主函数中，创建了 Test 类的两个实例 t1 和 t2，其中 t2 并没有在主函数之外被引用，因此不会发生逃逸。而 t1 被传递到了 foo() 方法，不过由于 t2 没有逃逸，因此 JVM 认为 t1 也没有逃逸。

JIT 基于逃逸分析的结果，可以使用不同的策略，为对象实例分配内存，在不同的策略下所生成的本地机器码自然不同。目前主要的优化技术包括标量替换和栈上分配。这两种优化技术都不会将对象实例直接分配在堆上。

1. 标量替换

所谓标量，是指不可分割的量，Java 中的基本数据类型和 reference 类型都属于标量，其中 reference 类型在 JVM 内部其实就是一个指针，因此也属于不可分割的量。如果一个数据类型可以继续分解，则称为聚合量。如果将一个对象拆散，将其成员变量恢复到基本类型以用于访问，这个过程就叫作标量替换。由此可知，标量替换不仅仅是替换那么单纯，替换后还要修改类型字段的读写指令。如果标量替换的优化比较激进，甚至可以直接将一个类的所有字段都打散分配到硬件寄存器中，当然前提是这个类型中包含的字段不能太多，毕竟寄存器数量是有限的。

2. 栈上分配

如果一个类实例引用变量没有发生逃逸，则将实例对象直接分配在方法栈上。在栈上分配的对象实例，会随着 Java 方法执行结束后方法栈空间的被回收而被回收，因此不需要 GC 来回收。这种方式所带来的好处是不言而喻的，毕竟 GC 时间越少，则 JVM 留给用户线程执行的时间片段就越多。

不过栈上分配也有其硬伤，那就是 Java 类型不能太大，包含的字段不能太多，毕竟堆栈空间是有限的，容纳不下几个大型的 Java 类。如果不是因为有这个限制，JVM 也不必建立所谓的堆内存空间。

逃逸分析的算法比较复杂，并且与本地机器指令有关，这里就不贴了，毕竟绝大多数开发者都不需要关心其具体实现。但是逃逸分析和基于此所进行的内存分配优化可以通过 jmap -histo 命令观察开启和关闭逃逸分析选项这两种情况下的实例总数来进行验证。除了这种办法，也可以通过观察 GC 日志来间接分析。例如下面这个示例：

清单：Test.java

功能：验证逃逸分析及基于此的内存分配优化

```
public class Test {
    int k = 3;
    long l = 300L;
    int m = 10;

    public static void main(String[] args) {
        for(int i = 0; i < 500000; i++) {
            Test t = new Test();
        }
    }

    // 可以在这里打断点，通过 jmap -histo 统计开启和关闭
    // 逃逸分析两种情况下的 Test 实例总数
    System.out.println("==1");
}
```

本示例程序中通过一个 50 万次循环，不断创建 Test 类实例对象。之所以要循环，是因为只有达到一定的循环次数才能成为热点代码，才会触发 JIT 编译优化，也才会启用逃逸分析。使用-server -Xmx5m -Xms5m -XX:-DoEscapeAnalysis -XX:+PrintGC 选项运行该测试程序，结果输出如下：

```
[GC (Allocation Failure) 1024K->715K(5632K), 0.0012355 secs]
[GC (Allocation Failure) 1739K->1015K(5632K), 0.0009311 secs]
[GC (Allocation Failure) 2039K->1253K(5632K), 0.0007820 secs]
[GC (Allocation Failure) 2277K->1293K(5632K), 0.0006812 secs]
[GC (Allocation Failure) 2317K->1309K(5632K), 0.0006202 secs]
[GC (Allocation Failure) 2333K->1325K(5632K), 0.0006383 secs]
[GC (Allocation Failure) 2349K->1413K(5632K), 0.0009138 secs]
[GC (Allocation Failure) 2437K->1477K(5632K), 0.0004426 secs]
[GC (Allocation Failure) 2501K->1445K(5632K), 0.0003631 secs]
[GC (Allocation Failure) 2469K->1477K(5632K), 0.0003411 secs]
[GC (Allocation Failure) 2501K->1397K(5632K), 0.0003140 secs]
// ...
```

-XX:-DoEscapeAnalysis 选项表示关闭逃逸分析。接着使用-server -Xmx5m -Xms5m -XX:+DoEscapeAnalysis -XX:+PrintGC 选项运行该测试程序，结果输出如下：

```
[GC (Allocation Failure) 1024K->716K(5632K), 0.0011911 secs]
[GC (Allocation Failure) 1740K->962K(5632K), 0.0039492 secs]
[GC (Allocation Failure) 1986K->1240K(5632K), 0.0020249 secs]
[GC (Allocation Failure) 2264K->1280K(5632K), 0.0007983 secs]
[GC (Allocation Failure) 2304K->1288K(5632K), 0.0009195 secs]
[GC (Allocation Failure) 2312K->1320K(5632K), 0.0016358 secs]
[GC (Allocation Failure) 2344K->1417K(5632K), 0.0015501 secs]
[GC (Allocation Failure) 2441K->1353K(5632K), 0.0008891 secs]
[GC (Allocation Failure) 2377K->1449K(5632K), 0.0005889 secs]
```

这一次使用-XX:+DoEscapeAnalysis 选项开启了逃逸分析（其实该选项在 JDK 8 中默认是开启的，因此不需设置该选项），JVM 一共只进行了 8 次 GC 回收，相比关闭逃逸分析时，次数少了很多。开启逃逸分析之后，JVM 之所以仍然存在 GC，是因为一开始运行时的循环次数不够，未触发 JIT，因此 Test 类实例对象仍然分配在堆空间，而当触发 JIT，JVM 使用 JIT 编译后的本地机器指令来实现类对象实例分配时，实例对象不再分配在堆上，因此不再有 GC 输出日志。

10.5.2 TLAB

相信很多研究 JVM 的道友对 TLAB 不会陌生，TLAB 的全称是 Thread Local Allocation Buffer，即线程本地分配缓存区，这是一个线程专用的内存分配区域。

TLAB 的出现能够解决直接在堆上安全分配所带来的线程同步性能消耗问题。堆内存是全

局的，任何线程都能够在堆上申请空间，因此每次申请堆内存空间时都必须进行同步处理。对于一个生产环境下的 Java Web 应用程序而言，拥有一两千、两三千个线程是很常见的事情，这么多线程同时在堆上申请内存，竞争十分激烈，必然会出现线程阻塞。而 TLAB 则是线程私有的一块内存空间，这块空间位于 eden 区。由于各个线程所拥有的 TLAB 区域彼此不重复，因此线程在各自的 TLAB 内存区域申请空间，无须加锁，这样内存申请的效率便会得到极大提升。其实说白了，这就是一种典型的“空间换时间”的策略。

参数-XX:+UseTLAB 可以开启 TLAB，默认是开启的。TLAB 的内存空间非常小，默认情况下仅占整个 eden 空间的 1%，每个线程所能拥有的 TLAB 空间非常少，因此 JVM 必然会限制 Java 类对象实例的大小。如果对象实例超过一定的阈值，便属于大对象，JVM 会将大对象直接分配在堆上，不再分配在 TLAB 区，否则一下子可能将 TLAB 区“打爆”。

在上面贴出来的 BytecodeInterpreter::run() 函数源码中，可以看到 TLAB 的逻辑，其逻辑是直接调用 THREAD->tlab().allocate(obj_size) 来完成 TLAB 内存分配。tlab 实际上是 JVM 内部 Thread 类的一个成员变量，类型是 ThreadLocalAllocBuffer，该类内部主要通过 3 个字段维护 TLAB 区域的范围，这 3 个字段分别是：

- ◎ _start，TLAB 区域的内存首地址。
- ◎ _top，最近一次 TLAB 分配内存后所指向的地址。
- ◎ _end，TLAB 区域的终止位置（内存对齐后的位置）。

通过这 3 个变量，TLAB 便能完成内存申请与释放。在 BytecodeInterpreter::run() 函数中调用 tlab->allocate(size_t) 来申请内存，实现如下：

清单：/src/share/vm/memory/ThreadLocalAllocBuffer.inline.hpp

功能：TLAB 内存分配

```
inline HeapWord* ThreadLocalAllocBuffer::allocate(size_t size) {
    invariants();

    // 获取当前 top
    HeapWord* obj = top();
    if (pointer_delta(end(), obj) >= size) {

        // 重置 top
        set_top(obj + size);

        invariants();
        return obj;
    }
    return NULL;
}
```

这里的实现极为简单，如果 TLAB 剩余的空间足够容纳 Java 类对象实例，则重置 TLAB 的 top 属性，如此便完成内存分配。虽然 TLAB 分配的逻辑很简单，但是 TLAB 内存空间的维护稍具复杂性，需要随着运行时的执行流而随时变化。如果 TLAB 剩余空间不够，则 TLAB 内存分配必定会失败，此时 JVM 便会向 eden 区申请内存空间。如果 JVM 再次申请分配一个比较小的 Java 对象实例，该对象大小小于 TLAB 区的剩余空间，则 JVM 会继续将小对象优先填充进 TLAB 区域中。

如果 TLAB 区域都用完了，如何处理呢？别忘了，TLAB 区域本身被分配在 eden 区，属于 eden 区的一部分，因此当 eden 区也快要用完的时候，会触发 GC 垃圾回收，在垃圾回收期间，TLAB 的内存空间会随着 eden 区的回收一起被回收掉，如此实现 TLAB 的循环利用。

10.5.3 指针碰撞与 eden 区分配

如果 JVM 向 TLAB 申请内存失败，则会转而向 eden 区申请内存。在这个过程中，使用了 bump-the-pointer 技术，也即指针碰撞。其逻辑实现其实比较简单，就在上文贴出来的 BytecodeInterpreter::run() 函数源码中，其实现如下：

清单：BytecodeInterpreter::run()

作用：指针碰撞

```
HeapWord* compare_to = *Universe::heap()->top_addr();
HeapWord* new_top = compare_to + obj_size;
if (new_top <= *Universe::heap()->end_addr()) {
    if (Atomic::cmpxchg_ptr(new_top, Universe::heap()->top_addr(),
compare_to) != compare_to) {
        goto retry;
    }
    result = (oop) compare_to;
}
```

Universe::heap()返回 JVM 内部所使用的 CollectedHeap 堆对象，top_addr()指向 eden 区空闲块的起始地址，end_addr()指向 eden 区空闲块的结束地址。JVM 首先通过 compare_to 保存 eden 区空闲块的起始地址，接着使用 new_top 保存分配内存后新的空闲块的起始地址。指针碰撞技术的关键在于 CAS 操作，这里通过基于 CPU 硬件的 CAS 原子指令进行空闲块的同步操作，比较_top 的预期值与 compare_to 是否相同，若相同则表明没有其他线程操作该变量，若没有其他线程操作该变量，则会更新 eden 区的_top 属性，并返回原来的_top 作为 Java 类实例对象的内存首地址。这便是所谓的“指针碰撞”技术，其实就是判断预期的 top 与原来的 top 是否相等。而指针碰撞的关键就是 CAS 原语，JVM 通过 CAS 避免了多线程之间的锁竞争，这是实现内存快速分配的技术保障。

10.5.4 清零

前面两个快速分配策略——先在 TLAB 上分配，如果分配失败，则再向 eden 区申请内存空间，如果这两种分配策略中有一个成功，则 Java 类实例对象将会在 TLAB 区或者 eden 区占有席之地。但是别忘了，无论是 TLAB 区还是 eden 区，都会不断地被 GC，因此 Java 对象实例所分配到的内存空间有可能仍残留着那些已经被回收或者被转移到其他堆内存区域的对象的信息片段，如果是这样，则需要将这段内存空间进行清零。在 BytecodeInterpreter::run() 函数源码中实现了清零逻辑：

清单：BytecodeInterpreter::run()

作用：清零

```
bool need_zero = !ZeroTLAB;
if (UseTLAB) {
    result = (oop) THREAD->tlab().allocate(obj_size);
}
if (result == NULL) {
    need_zero = true;

    // 向 eden 区申请内存
    // ....
}
if (result != NULL) {
    if (need_zero) {
        HeapWord* to_zero = (HeapWord*) result + sizeof(oopDesc) / oopSize;
        obj_size -= sizeof(oopDesc) / oopSize;
        if (obj_size > 0) {
            memset(to_zero, 0, obj_size * HeapWordSize);
        }
    }
}
```

在这段逻辑中，主要关注 need_zero，如果在 TLAB 区中成功分配内存，并且 TLAB 区本身已清零，则表达式 !ZeroTLAB 返回 false，于是 if(need_zero) 不满足条件，无需清零。否则，即使在 TLAB 区成功申请内存，最终仍需清零。如果 TLAB 区申请内存失败，则向 eden 区分配内存之前先将 need_zero 设置为 true，这意味着只要是从 eden 区分配的内存，最终都需要清零。清零的方式很简单，将指定的内存区域全部设置为 0 即可，所有的二进制位都是 0。

10.5.5 偏向锁

如果快速分配策略成功实施并完成清零，接着会设置偏向锁。所谓设置偏向锁，其实是设置对象头，即 oop 的 mark 标记，其逻辑如下：

清单：BytecodeInterpreter::run()

作用：设置偏向锁

```
if (UseBiasedLocking) {
    result->set_mark(ik->prototype_header());
} else {
    result->set_mark(markOopDesc::prototype());
}
```

不是说好设置偏向锁的吗？可是这里的源码看起来像是在设置 prototype。其实 prototype 的类型便是 mark，而每一个 Java 类实例都有一个 mark 标记，在前文讲解 oop-klass 这种面向对象的表达机制时曾提到，所谓的 mark，看起来像个 C++ 对象，但实际上在 JVM 内部是被当作一个指针使用的，在 32 位平台上，指针就是一个 32 位的正整数，同理，64 位上的指针便是一个 64 位的正整数。而 JVM 会将 Java 类对象的 GC 分代年龄、哈希码、锁标志位等信息存放到这个 mark 上，其实说白了就是二进制打标。其中，偏向锁的标识也会打在这个 mark 指针上。

看上面这段源码，如果 JVM 开启偏向锁，则将 ik->prototype_header() 设置为新创建的 Java 类实例对象的标记。ik->prototype_header() 返回的标记中的偏向锁，其实指向当前线程 ID，而当前线程便是正在执行 new 指令、创建目标 Java 类对象实例的线程，因此通过 result->set_mark(ik->prototype_header()) 便将该新创建的对象的偏向锁偏向于当前创建它的线程，如果在接下来的执行过程中，该锁没有被其他的线程获取，则持有偏向锁的线程将永远不需要再进行同步。

而如果 JVM 没有开启偏向锁，则将 markOopDesc::prototype() 设置为新创建的 Java 类实例对象的标记。markOopDesc::prototype() 在前文也分析过，其返回一个没有哈希码、没有偏向锁的标记。

关于偏向锁，其概念往往与轻量级锁、重量级锁紧密关联在一起，这几种锁可以相互转化。很多书籍与网络博客都有详细介绍，感兴趣的道友可以深入研究。关于多线程同步控制是一个很复杂的话题，从硬件到软件有各种各样的解决方案，想要描述清楚，本身便可以写一本书。如果感兴趣的道友非常多，则笔者会在后续版本中详细讲解，配合着 JVM 的 GC 机制一起深入分析。

10.5.6 压栈与取指

在快速分配流程走完偏向锁设置之后，Java 类实例对象的内存空间已经分配完成，接着 JVM 将 Java 对象实例的内存首地址压入操作数栈栈顶，完成压栈之后则开始取指——读取下一条字节码指令，在 BytecodeInterpreter::run() 函数中实现了这种逻辑：

```
SET_STACK_OBJECT(result, 0); //压栈
UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1); //取指
```

不过按道理，完成对象内存分配之后，不是应该将对象的内存首地址存储到局部变量表中对应的位置吗？这里为何只进行了压栈？道理很简单，Java 源码中的 new 语句通常会被编译为几条字节码指令，其中第一条字节码指令便是 new 指令，上面的逻辑都是 new 指令的内部实现机制。完成 new 指令之后，JVM 接着会调用 Java 类的构造函数，通过构造函数才真正返回一个完成原始构建的内部对象。构造函数运行之后，则会生成一条字节码指令，将对象的内存首地址存储到局部变量表中。例如下面的示例：

清单：Test.java

功能：new 指令

```
public class Test {
    public void test() {
        Test t = new Test();
    }
}
```

该示例特别简单，编译后，使用 javap 命令查看其字节码指令，如下：

```
public void test();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=2, locals=2, args_size=1
  0: new           #2                      // class Test
  3: dup
  4: invokespecial #3                    // Method "<init>":()V
  7: astore_1
  8: return
```

可以看到，源码中的 new 语句对应了 4 条字节码指令，首先执行 new 指令，接着执行 invokespecial 指令调用类的默认无参构造函数<init>(), 最后再通过 astore_1 将构造好的实例对象内存首地址保存进局部变量表中。由此可以看出，完成 new 指令后，之所以不立即将对象内存地址写入局部变量表中，是因为接下来就会调用方法，而 JVM 每次调用 Java 方法之前，都必须要将入参压入操作数栈栈顶。如果执行完 new 指令之后就立即将对象内存地址写入局部变量表中，那么接下来调用类的构造函数时就需要再次将对象内存地址从局部变量表中读取出来并压入操作数栈栈顶，这样多了几次内存读写，所以 JVM 干脆就在执行完 new 指令后，直接将内存地址压入栈顶，提升性能。

上面分析了 Java 类对象内存的快速分配的技术实现机制，总体而言，快速分配的策略示意图如图 10.6 所示。

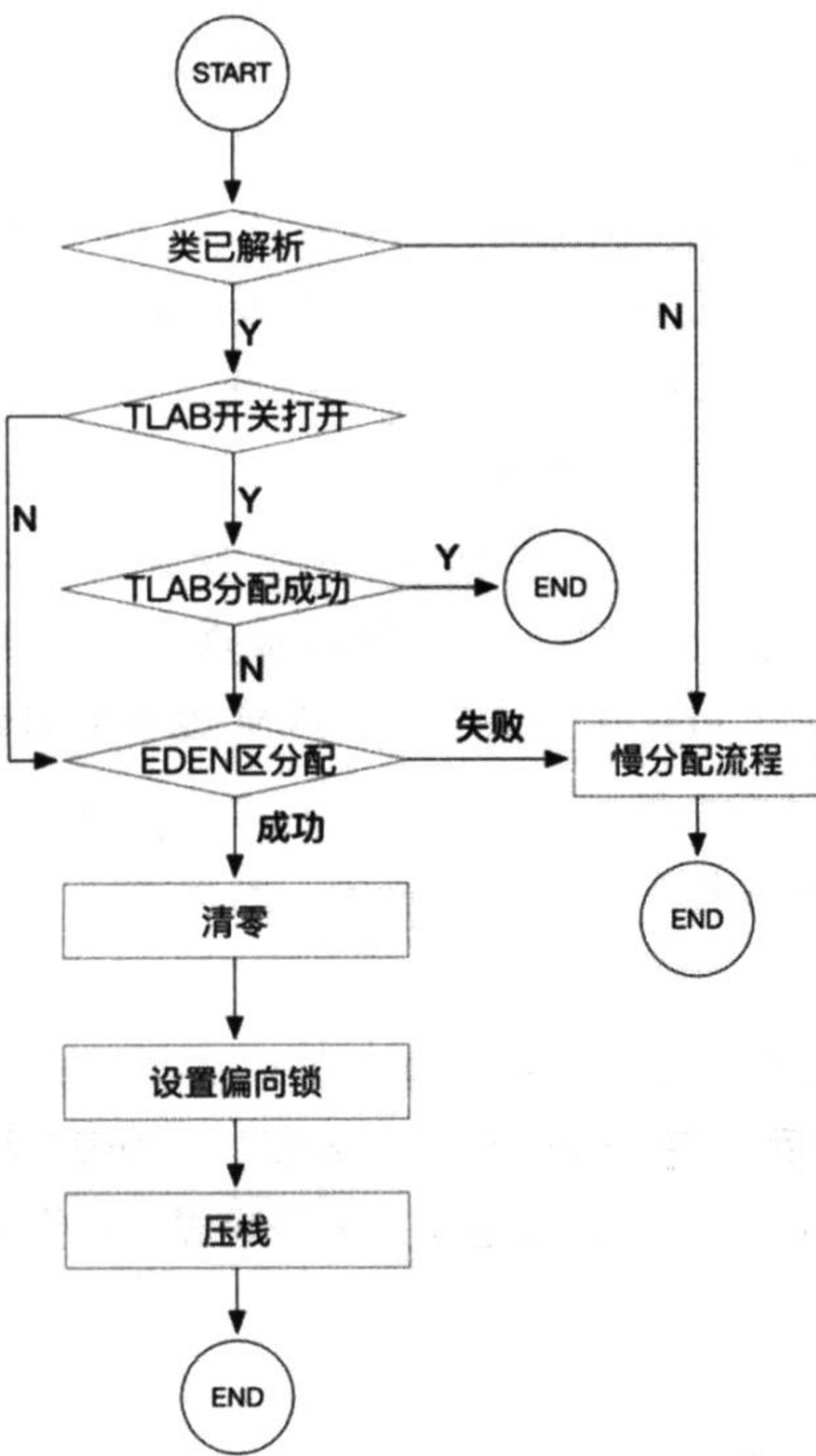


图 10.6 Java 类实例对象内存快速分配流程

如果快速分配失败，则最终会进入慢分配流程，慢分配流程也会首先尝试在 TLAB 中分配，如果分配失败，则继续尝试使用指针碰撞技术在新生代分配，这种分配是无锁的，效率仍然很高。如果仍然分配失败，则最终才会使用互斥锁，在堆区进行分配。在这个过程中如果碰到 GC 正在回收垃圾，则会等待 GC 回收完成。慢分配流程中所使用的优化手段与快速分配类似，都是优先使用无锁分配方案（TLAB 或指针碰撞）。慢分配与 GC 在理论和源码上是紧密耦合在一起的，而 GC 本身是一个非常复杂的管理体系，本书限于篇幅不展开细讲，如果有很多道友都对此感兴趣，我们可以在后续版本中继续交流。

10.6 本章总结

类的加载机制与生命周期等概念，在各种书籍与各种网络博客里随处可见，然而对于一个想要真正了解其内部实现的人而言，那些都涉入过浅。本章“拨云见日”，从 JVM 源码的角度，

还原出 Java 类加载的真实机制，以及类生命周期的实现方式。

从程序的角度看，其实 Java 技术体系仅仅定义了两种类加载器——一种是 boot class loader，即引导类加载器，其是使用 C++ 编写而成的，打包在 JVM 程序内部。另一种则是 JDK 类库中的 `java.lang.ClassLoader` 加载器，这种加载器专门提供给 Java 程序使用，所有在 Java 程序中定义的类加载器最终都通过 `java.lang.ClassLoader` 得以实现。

当 JVM 启动时，会加载核心的几个类库，例如 `Object`、`Long` 及 `Integer` 等。剩下的 Java 类，无论是 JDK 类库中的，还是 Java 应用程序中的，或者是 Java 应用程序所依赖的第三方 jar 包，都采用“延迟加载”机制。在这种机制下，只有当 Java 真正需要使用某个类时，JVM 才会真正加载，否则，即便在程序中显式 `import` 了某个类，JVM 也不会加载。

为了确保 JDK 核心类库的安全性，Java 技术生态体系实现了“双亲委派”机制，无论是内部的 boot class loader，还是 Java 程序层面的加载器，都通过双亲委派机制，保护核心类不被开发者破坏和伪造。

类的实例分配的技术实现相当精彩，在这本书里你随处都能看到那些伟大工程师们的思想闪光点。为了实现类实例内存的快速分配，各种“阴谋”、“阳谋”纷纷上场，TLAB 无锁技术、栈上分配、标量替换、指针碰撞，等等，真是让人大开眼界，让你不禁感慨，这才是把技术“玩到了家”。