Java基础

类

- 无论什么类型的内部类，都会编译成一个独立的 .class 文件。
- 内部类可以访问外部类的所有静态属性和方法。
- 类内方法不加访问权限控制符的时候全包可见。

方法

- 静态代码执行顺序：

  在创建类对象时，会先执行父类和子类的静态代码块 然后再执行父类和子类的构造方法。并不是执行完父类的静态代码块和构造方法后，再去执行子类。静态代码块只运行一次，在第二次对象实例化时，不会运行。

- Override：最终的实现需要在运行期判断，这就是所谓的动态绑定。JVM通过方法表激活实例方法，如果某个子类复写了父类的某个方法，方法表中的方法引用会指向子类的实现处。

- Override需要满足四个条件：

  1. 访问权限不能变小。
  2. 返回类型能够向上转型成父类的返回类型。
  3. 异常也要能向上转型成父类的异常。
  4. 方法名，参数类型及个数必须严格一致。

  一大两小两同：

  - 一大：子类的方法访问权限控制符只能相同或变大。
  - 两小：抛出异常和返回值只能变小，能够转型成父类对象。子类的返回值，抛出异常类型必须与父类的返回值、抛出异常类型存在继承关系。
  - 两同：方法名和参数必须完全相同。

- Overload：选择合适目标方法的顺序如下：

  1. 精确匹配
  2. 如果是基本数据类型，自动转换成更大表示范围的基本类型
  3. 通过自动装箱和拆箱
  4. 通过子类向上转型继承路线依次匹配
  5. 通过可变参数匹配

泛型

1. 尖括号中的每个元素都只代一种未知类型。String 出现在尖括号里，它就不是java.lang.String，而仅仅是一个代号。
2. 尖括号必须在类名之后或方法返回值之前
3. 泛型在定义处只具备执行Object方法的能力（比如只能调用toString() ）
4. 对于编译后的字节码指令，其实没有这些花头花脑的方法签名，充分说明了泛型只是一种编写代码是的语法检查。
   - 这即是“类型擦除”，编译后，设定的泛型参数或返回值是Object，数据返回时也进行了强制类型转化。因此泛型就是在编译期增加了一道检查而已。

字符串

String不可变，对它的任何改动，其实都是创建一个新对象，再把引用指向该对象。String对象赋值操作后，会在常量池中进行缓存，如果下次申请创建对象时，缓存中已经存在，则直接返回相应引用给创建者。

JVM

类加载过程：

- 加载，读取类文件产生二进制流，并转化为特定数据结构，初步校验然后创建对应的java.lang.Class实例
- 验证，准备，解析
    1. 验证：更详细的校验，比如final是否合规，类型是否正确，静态变量是否合理等。
    2. 准备：为静态变量分配内存，设定默认值。
    3. 解析：将常量池中的符号引用转化为直接引用
- 初始化：
    1. 执行被static关键字标识的代码（静态变量，静态代码块）
    2. 类构造器方法

类加载器：

1. Bootstrap

   负责加载 JDK 核心类，并且在加载完核心类后构造 Extension ClassLoader 和 App ClassLoader

   用C++实现

2. Extension

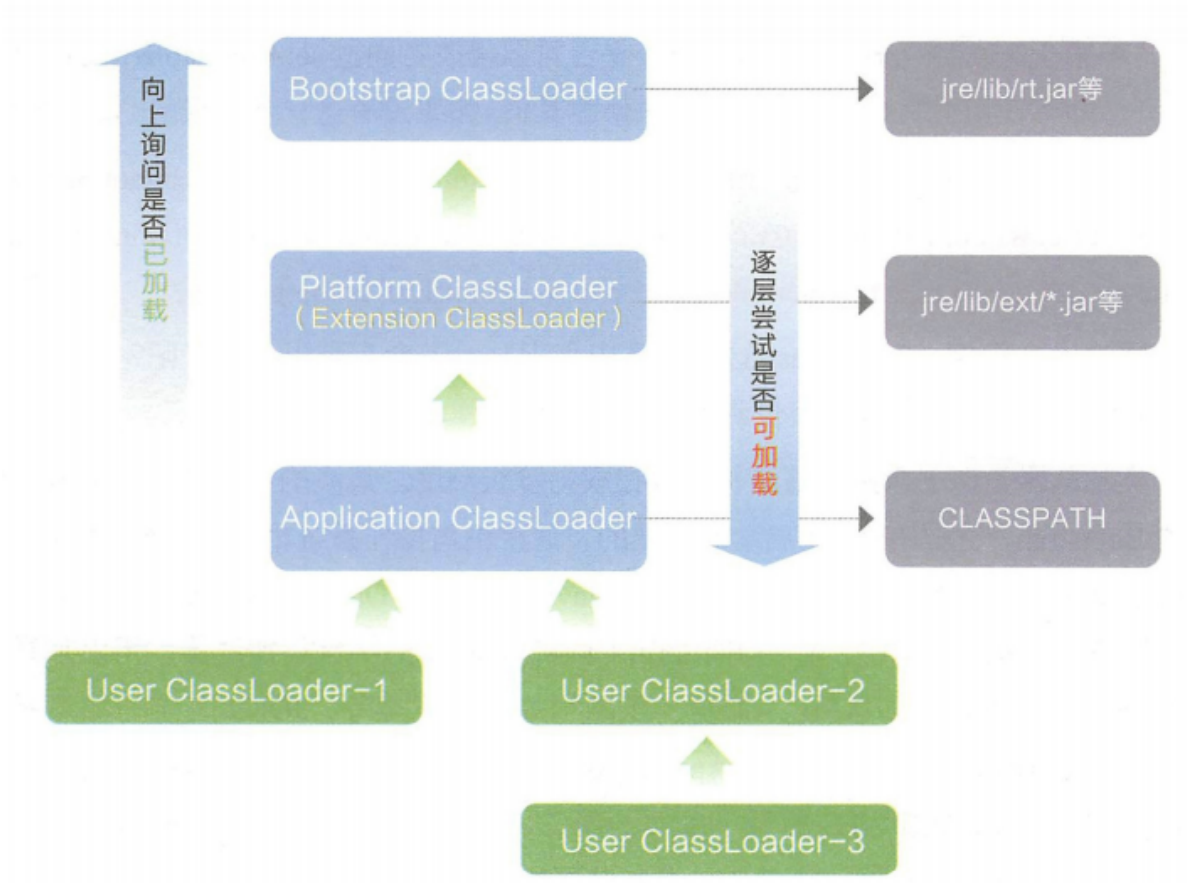
   加载扩展的系统类

3. Application

   加载用户定义的CLASSPATH路径下的类

类加载器具有等级制度，但是并非继承关系。

双亲委派模型：



什么情况下需要自定义类加载器?

1. 隔离加载类。在某些框架内进行中间件与应用的模块的隔离，把类加载到不同的环境。比如容器框架内通过自定义类加载器确保应用中依赖的jar包不会影响到中间件运行时使用的jar包。
2. 修改类加载方式。根据实际情况在某个时间点按需进行动态加载。
3. 扩展加载源。从数据库、网络等等进行加载。
4. 防止源码泄露。Java代码可以进行编译加密，那么类加载器也需要自定义，还原加密的字节码。

Tomcat为什么打破双亲委派模型?

- tomcat 为了实现隔离性，没有遵守这个约定，每个webappClassLoader加载自己的目录下的class文件，不会传递给父类加载器
- 对于各个 `webapp` 中的 `class` 和 `lib`，需要相互隔离，不能出现一个应用中加载的类库会影响另一个应用的情况，而对于许多应用，需要有共享的lib以便不浪费资源。

实现自定义类加载器的步骤继承 ClassLoader ，重写 findClass（）方法，调用 defineClass（） 方法

线程上下文加载器:

为了解决类加载器命名空间的限制。在双亲委托机制下，类加载是由下至上的，即下层的类加载器会委托上层进行加载。在SPI（Service Provider Interface）场景下，**有些接口是Java核心库提供的，并且是由启动类加载器来加载的，而这些接口的实现却来自不同的jar包（实现厂商提供），Java的启动类加载器是不会加载其它来源的Jar包**，这样传统的双亲委托机制无法满足SPI要求。通过给当前线程设置上下文类加载器，就可以由其负责加载接口实现的类。

双亲委派会影响 **被加载类** 之间的可见性，若类加载器 A 的父类加载器为 B，则：

- B 加载的类对 A 加载的类可见；
- A 加载的类对 B 加载的类不可见；

因此应用本身的代码可以使用 JDK 核心类库，但反之不行，典型例子即 SPI 机制。

借助上下文类加载器，可以打破双亲委派的依赖关系，原本被 **系统类加载器** 加载的 *用户 SPI 实现* 无法被 JDK 核心类库调用，此时 SPI 实现可以由 **线程上下文类加载器** 加载，进而可被核心类库调用。

内存布局：

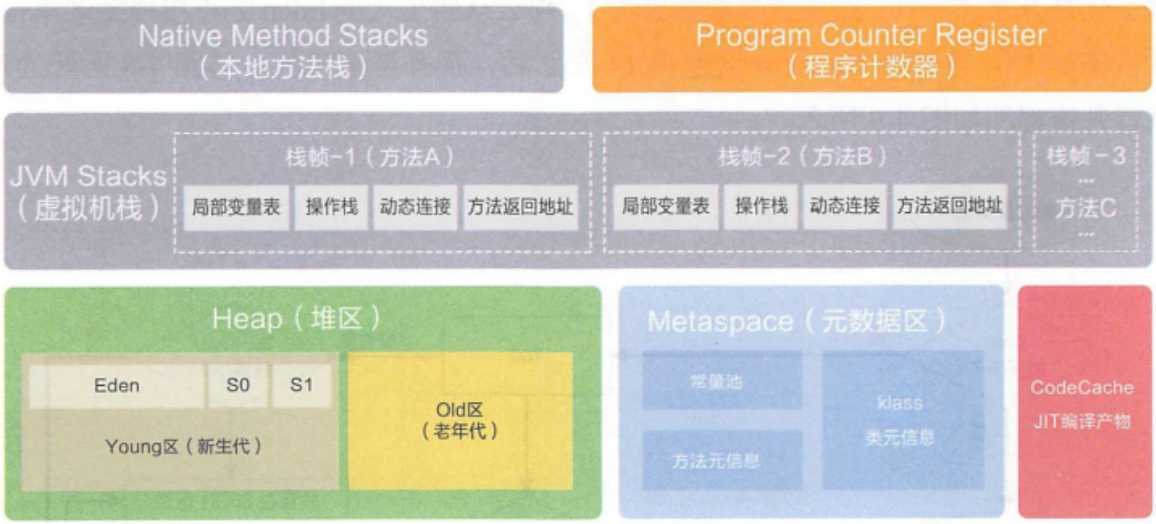


图 4-8　经典的 JVM 内存布局

1. Heap（堆区）

   堆区由各子线程共享使用。分为新生代和老年代。

   - 新生代：1个Eden区 + 2个Survivor区。

     绝大部分对象在Eden区生成，当Eden填满的时候出发Young Garbage Collection。Eden区没有被引用的对象直接回收，依然存活的对象会被移送到Survivor区。

   - Survivor分为S0和S1，每次GC时将存活的对象复制到未使用的那块空间，然后将当前正在使用的空间完全清除，交换两块空间的使用状态。如果YGC需要移送的对象大于Survivor区容量上限，直接移交给老年代。默认在Survivor区交换14次后，晋升至老年代。

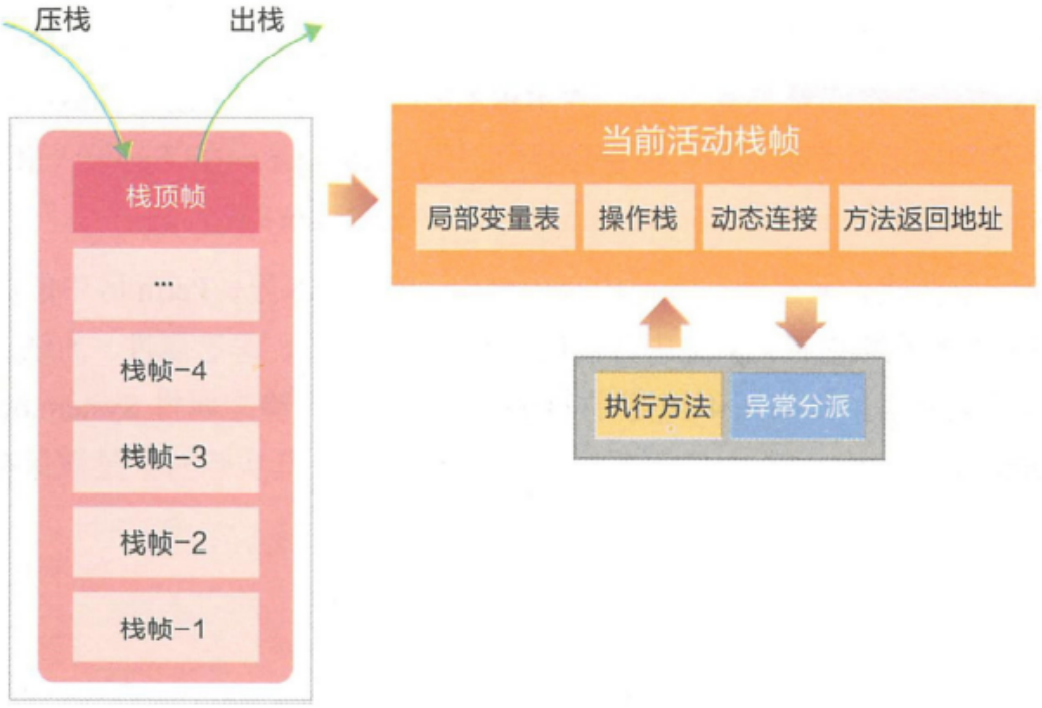
   - 如果Survivor区无法放下，或者超大对象的阈值超过上限，尝试在老年代中分配内存。如果依然无法放下，抛出OOM。

2. Metaspace（元空间）

   包括类元信息、字段、静态属性、方法、常量等。字符串常量在堆内存。

3. JVM Stack（虚拟机栈）

   线程私有，栈中的元素用于支持虚拟机进行方法调用，每个方法从开始调用到执行完成的过程，就是栈帧从入栈到出栈的过程。



- 局部变量表

  存放方法参数和局部变量的区域。

- 操作栈

  是一个初始状态为空的桶式结构栈。方法执行过程中，会有各种指令往栈中写入和提取信息。

- 动态连接

  每个栈帧中包含一个在常量池中对当前方法的引用，目的是支持方法调用过程的动态连接

- 方法返回地址

  方法退出的过程相当于弹出当前栈帧。

4. Native Method Stacks（本地方法栈）

   线程对象私有。本地方法栈为native方法服务。本地方法可以使用Java Native Interface（JNI）来访问虚拟机运行时的数据区，甚至可以调用寄存器，具有和JVM相同的能力和权限。例子：System.currentTimeMillis()。

5. Program Counter Register（程序计数寄存器）

   众多线程在并发执行过程中，任何一个确定的时刻，一个处理器只会执行某个线程中的一条指令，这样必然导致经常中断和恢复。程序计数器用来存放执行指令的偏移量和行号指示器等，线程执行或者恢复都要依赖程序计数器。

从线程共享的角度看，堆和元空间时线程共享的，而虚拟机栈、本地方法栈、程序计数器是线程内部私有的。

对象实例化：

- 确认类元信息是否存在。JVM接收到new指令时，首先在metaspace里检查需要创建的类元信息是否存在。

  若不存在，那么在双亲委派模式下，使用当前类加载器以ClassLoader + 包名 + 类名为key进行查找对应的.class文件。没有找到就抛出notFound异常，找到就进行类加载，并生成对应的Class对象。

- 分配对象内存。计算对象占用空间大小，如果实例成员变量是引用变量，仅分配引用变量空间即可（4个字节），接着在堆中划分一块内存给新对象。分配内存空间时需要进行同步操作，比如CAS、区域加锁。
- 设定默认值
- 设置对象头。设置新对象的哈希码、GC信息、锁信息、对象所属类元信息等
- 执行init方法。初始化成员变量，执行实例化代码块，调用类的构造方法，并把堆内对象的首地址赋值给引用变量。

垃圾回收：

- 引用计数算法：

  早期的策略。每个对象有一个引用计数器，创建时为1，有引用时引用加1，引用超过生命周期或者被设置为新地址，计数器减一。计数器为0时，成为回收对象。

  优点：简单，高效，执行速度快，不会长时间打断程序

  缺点：无法检测出循环引用

- 根搜索算法：

  如果一个对象与GC Roots之间没有直接或间接的引用关系，比如某个失去任何引用的对象，或者两个互相环岛状循环引用的对象等，这些对象是可以回收的。可以作为GC Roots的对象：类静态属性中引用的对象、常量引用的对象、虚拟机栈中引用的对象、本地方法栈中引用的对象等。

垃圾回收的相关算法：

- 标记-清除 算法

  从每个GC Roots出发，依次标记有引用关系的对象，最后将没有被标记的对象清除。但这种算法会带来大量的空间碎片，导致需要分配一个较大连续空间时容易触发FGC。

- 标记-整理 算法

  首先从GC Roots出发标记存活的对象，然后将存活对象整理到内存空间的一端，形成连续的已使用空间，最后把已使用空间之外的部分全部清除掉。

- Mark-Copy 算法

  为了能够**并行**地标记和整理将空间分成两块，每次只激活其中一块，垃圾回收时只需要把存活对象复制到另一块未激活空间上，将未激活空间标记为已激活，将已激活空间标记为未激活，然后清除原空间中的原对象。

  优点：减少了内存空间的浪费，现作为主流的YGC算法进行新生代的垃圾回收。

垃圾回收器：

- Serial：

  采用串行单线程方式，在垃圾回收的某个阶段会暂停整个应用程序的执行（Stop The World）。FGC时间相对较长，频繁FGC会严重影响程序性能。

- CMS（Concurrent Mark Sweep Collector）

  回收停顿时间比较短。通过初始标记（Initial mark）、并发标记（Concurrent mark）、重新标记（Remark）、并发清除（Concurrent sweep）四个步骤完成。

  初始标记和重新标记仍然会触发STW，而2，4步的并发标记和并发清除两个阶段可以和应用程序并发执行，也是比较耗时的操作。

  CMS采用标记-清除算法，会产生大量空间碎片。可通过参数强制JVM在FGC完成后对老年代进行压缩，执行一次空间碎片整理。

  初始标记：只找到root，STW时间非常短

并发标记：工作线程和垃圾回收线程并发

重新标记： 第二阶段因为多线程，会出现漏标和错标，第三步进行修正，同样STW，时间很短

并发清除：工作线程和垃圾回收线程并发清除

存在的问题：

当CMS碎片化特别严重时，会使用serial old（单线程）清除，卡顿时间极长

- G1

  将Java堆空间分割成了若干相同大小的区域（Region），包括Eden、Survivor、Old、Humongous四种类型。Humongous是特殊的Old类型，专门放置大型对象。这样的划分方式意味着不需要一个连续的内存空间管理对象。G1将空间分为多个区域，优先回收垃圾最多的区域。

  G1采用的是mark-copy，一大优势是在于可预测的停顿时间，能尽快在指定时间内完成垃圾回收任务。

  逻辑分代，物理不分代