JVM_05 运行时数据区 4- 对象的实例化内存布局与访问定位 + 直接内存

作者：wlgzs-sjl
原文链接：https://ld246.com/article/1602683686993
来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20190307.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="1-对象的实例化">1.对象的实例化</h2>
<img src="https://ld246.com/images/img-loading.svg" alt="1728e4cbb14c0bd3.png" da
a-src="https://b3logfile.com/file/2020/10/1728e4cbb14c0bd3-0568e768.png?imageView2/2/
nterlace/1/format/jpg">
<h3 id="1-1-创建对象的方式">1.1 创建对象的方式</h3>

new

最常见的方式
变形 1： Xxx 的静态方法
变形 2： XxBuilder/XxoxFactory 的静态方法

Class 的 newInstance（）：反射的方式，只能调用空参的构造器，权限必须是 public
Constructor 的 newInstance（Xxx）：反射的方式，可以调用空参、带参的构造器，权限没有
求
使用 clone（）：不调用任何构造器，当前类需要实现 Cloneable 接口，实现 clone（）
使用反序列化：从文件中、从网络中获取一个对象的二进制流
第三方库 Objenesis

<h3 id="1-2-创建对象的步骤">1.2 创建对象的步骤</h3>

判断对象对应的类是否加载、链接、初始化
为对象分配内存

如果内存规整：指针碰撞
如果内存不规整：

虚拟机需要维护一个列表
空闲列表分配

处理并发安全问题

采用 CAS 配上失败重试保证更新的原子性
每个线程预先分配一块 TLAB

初始化分配到的空间一所有属性设置默认值，保证对象实例字段在不赋值时可以直接使用
设置对象的对象头
执行 init 方法进行初始化。

<h4 id="1--判断对象对应的类是否加载-链接-初始化">1) 判断对象对应的类是否加载、链接、初始
</h4>
虚拟机遇到一条 new 指令，首先去检查这个指令的参数能否在 Metaspace 的常量池中定位到
个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化。（即判断类元信
是否存在）。如果没有，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
ey 进行查找对应的.class 文件。如果没有找到文件，则抛出 ClassNotFoundException 异常，如果
到，则进行类加载，并生成对应的 Class 类对象
<h4 id="2--为对象分配内存">2) 为对象分配内存</h4>

首先计算对象占用空间大小，接着在堆中划分一块内存给新对象。
如果实例成员变量是引用变量，仅分配引用变量空间即可，即 4 个字节大小。

如果内存规整，使用指针碰撞
如果内存是规整的，那么虚拟机将采用的是指针碰撞法（BumpThePointer）来为对象分配内存
意思是所有用过的内存在一边，空闲的内存在另外一边，中间放着一个指针作为分界点的指示器，分
内存就仅仅是把指针向空闲那边挪动一段与对象大小相等的距离罢了。如果垃圾收集器选择的是 Seria
、ParNew 这种基于压缩算法的，虚拟机采用这种分配方式。一般使用带有 compact （整理）过程
收集器时，使用指针碰撞。

如果内存不规整，虚拟机需要维护一个列表，使用空闲列表分配
如果内存不是规整的，已使用的内存和未使用的内存相互交错，那么虚拟机将采用的是空闲列表
来为对象分配内存。意思是虚拟机维护了一个列表，记录上哪些内存块是可用的，再分配的时候从列
中找到一块足够大的空间划分给对象实例，并更新列表上的内容。这种分配方式成为“空闲列表（Free
List）”。

说明：选择哪种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器
否带有压缩整理功能决定。
<blockquote>
给对象的属性赋值的操作：
① 属性的默认初始化
② 显式初始化
③ 代码块中初始化
④ 构造器中初始化
</blockquote>
<h4 id="3--处理并发安全问题">3) 处理并发安全问题</h4>
在分配内存空间时，另外一个问题是及时保证 new 对象时候的线程安全性：创建对象是非常频
的操作，虚拟机需要解决并发问题。虚拟机采用 了两种方式解决并发问题：

CAS （ Compare And Swap ）失败重试、区域加锁：保证指针更新操作的原子性；
TLAB 把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配
小块内存，称为本地线程分配缓冲区，（TLAB ，Thread LocalAllocation Buffer） 虚拟机是否使用
LAB，可以通过一 XX：+/-UseTLAB 参数来 设定。

<h4 id="4--初始化分配到的空间">4) 初始化分配到的空间</h4>
内存分配结束，虚拟机将分配到的内存空间都初始化为零值（不包括对象头）。这一步保证了对
的实例字段在 Java 代码中可以不用赋初始值就可以直接使用，程序能访问到这些字段的数据类型所
应的零值。
<h4 id="5--设置对象的对象头">5) 设置对象的对象头</h4>
将对象的所属类（即类的元数据信息）、对象的 HashCode 和对象的 GC 信息、锁信息等数据
储在对象的对象头中。这个过程的具体设置方式取决于 JVM 实现。
<h4 id="6--执行init方法进行初始化">6) 执行 init 方法进行初始化</h4>
在 Java 程序的视角看来，初始化才正式开始。初始化成员变量，执行实例化代码块，调用类的
造方法，并把堆内对象的首地址赋值给引用变量。
因此一般来说（由字节码中是否跟随有 invokespecial 指令所决定），new 指令之 后会接着就是执
方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全创建出来。
代码示例
<pre><code class="highlight-chroma"><span class="highlight
cl">/**
* 测试对象实例化
过程

```
</span></span><span class="highlight-line"><span class="highlight-cl"> * ① 加载类元信息 - ② 为对象分配内存 - ③ 处理并发问题 - ④ 属性的默认初始化（零值初始化）
</span></span><span class="highlight-line"><span class="highlight-cl"> * - ⑤ 设置对象的信息 - ⑥ 属性的显式初始化、代码块中初始化、构造器中初始化
</span></span><span class="highlight-line"><span class="highlight-cl"> *
</span></span><span class="highlight-line"><span class="highlight-cl"> * 给对象的属性值的操作：
</span></span><span class="highlight-line"><span class="highlight-cl"> * ① 属性的默认始化 - ② 显式初始化 / ③ 代码块中初始化 - ④ 构造器中初始化
</span></span><span class="highlight-line"><span class="highlight-cl"> *
</span></span><span class="highlight-line"><span class="highlight-cl"> */
</span></span><span class="highlight-line"><span class="highlight-cl">public class Customer{
</span></span><span class="highlight-line"><span class="highlight-cl">    int id = 1001;
</span></span><span class="highlight-line"><span class="highlight-cl">    String name;
</span></span><span class="highlight-line"><span class="highlight-cl">    Account acct;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    {
</span></span><span class="highlight-line"><span class="highlight-cl">        name = "匿客户";
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    public Customer(){
</span></span><span class="highlight-line"><span class="highlight-cl">        acct = new Account();
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">class Account{
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h2 id="2--对象的内存布局">2. 对象的内存布局</h2>
<h3 id="对象头-Header-">对象头（Header）</h3>
<p>包含两部分</p>
<ul>
<li>运行时元数据
<ul>
<li>哈希值（ HashCode ）</li>
<li>GC 分代年龄</li>
<li>锁状态标志</li>
<li>线程持有的锁</li>
<li>偏向线程 ID</li>
<li>偏向时间戳</li>
</ul>
</li>
<li>类型指针：指向类元数据的 InstanceKlass，确定该对象所属的类型</li>
<li>说明：如果是数组，还需记录数组的长度</li>
</ul>
<h3 id="实例数据-Instance-Data-">实例数据（Instance Data）</h3>
<p>说明：它是对象真正存储的有效信息，包括程序代码中定义的各种类型的字段（包括从父类继承来的和本身拥有的字段）<br>
规则：</p>
```


相同宽度的字段总被分配在一起
父类中定义的变量会出现在子类之前
如果 CompactFields 参数为 true（默认为 true），子类的窄变量可能插入到父类变量的空隙</l
>

<h3 id="对齐填充-Padding-">对齐填充（Padding）</h3>
不是必须的，也没特别含义，仅仅起到占位符作用
<h3 id="小结">小结</h3>
<pre><code class="highlight-chroma"><span class="highlight
cl">public class CustomerTest {
    public static vo
d main(String[] args) {
        Customer cus
 = new Customer();
    }
}
</code></pre>
<img src="https://ld246.com/images/img-loading.svg" alt="1728e4d06baca2bf.png" dat
-src="https://b3logfile.com/file/2020/10/1728e4d06baca2bf-3cc2d47c.png?imageView2/2/in
erlace/1/format/jpg">
<h2 id="3-对象的访问定位">3.对象的访问定位</h2>
JVM 是如何通过栈帧中的对象引用访问到其内部的对象实例的呢？-> 定位,通过栈上 referenc
 访问
<img src="https://ld246.com/images/img-loading.svg" alt="1728e4d419936de2.png" da
a-src="https://b3logfile.com/file/2020/10/1728e4d419936de2-8f6beeac.png?imageView2/2/i
terlace/1/format/jpg">
对象访问的主要方式有两种

句柄访问

<img src="https://ld246.com/images/img-loading.svg" alt="1728e4d9d0cb81f4.png" dat
-src="https://b3logfile.com/file/2020/10/1728e4d9d0cb81f4-03e6e2ce.png?imageView2/2/in
erlace/1/format/jpg">

直接指针(HotSpot 采用)

<img src="https://ld246.com/images/img-loading.svg" alt="1728e4de9c0c7bc9.png" dat
-src="https://b3logfile.com/file/2020/10/1728e4de9c0c7bc9-6b2ed479.png?imageView2/2/in
erlace/1/format/jpg">
<h2 id="直接内存-Direct-Memory-">直接内存(Direct Memory)</h2>

不是虚拟机运行时数据区的一部分，也不是《Java 虚拟机规范》中定义的内存区域
直接内存是 Java 堆外的、直接向系统申请的内存区间
来源于 NIO，通过存在堆中的 DirectByteBuffer 操作 Native 内存

<img src="https://ld246.com/images/img-loading.svg" alt="1728f3d6c0cb33c5.png" dat
-src="https://b3logfile.com/file/2020/10/1728f3d6c0cb33c5-fe90185a.png?imageView2/2/int
rlace/1/format/jpg">
<img src="https://ld246.com/images/img-loading.svg" alt="1728f3daca9a3f28.png" data
src="https://b3logfile.com/file/2020/10/1728f3daca9a3f28-f4a3ff10.png?imageView2/2/interl
ce/1/format/jpg">

通常，访问直接内存的速度会优于 Java 堆。即读写性能高

因此出于性能考虑，读写频繁的场合可能会考虑使用直接内存
Java 的 NIO 库允许 Java 程序使用直接内存，用于数据缓冲区

也可能导致 OutOfMemoryError 异常:OutOfMemoryError: Direct buffer memory
由于直接内存在 Java 堆外，因此它的大小不会直接受限于一 Xmx 指定的最大
堆大小，但是系统内存是有限的，Java 堆和直接内存的总和依然受限于操作系统能给出的最大内存。
/li>
缺点

分配回收成本较高
不受 JVM 内存回收管理

直接内存大小可以通过 MaxDirectMemorySize 设置
如果不指定，默认与堆的最大值一 Xmx 参数值一致

简单理解：
java process memory = java heap + native memory
<img src="https://ld246.com/images/img-loading.svg" alt="1728f3dfc381eb0e.png" dat
-src="https://b3logfile.com/file/2020/10/1728f3dfc381eb0e-fe32ef0e.png?imageView2/2/inte
lace/1/format/jpg">