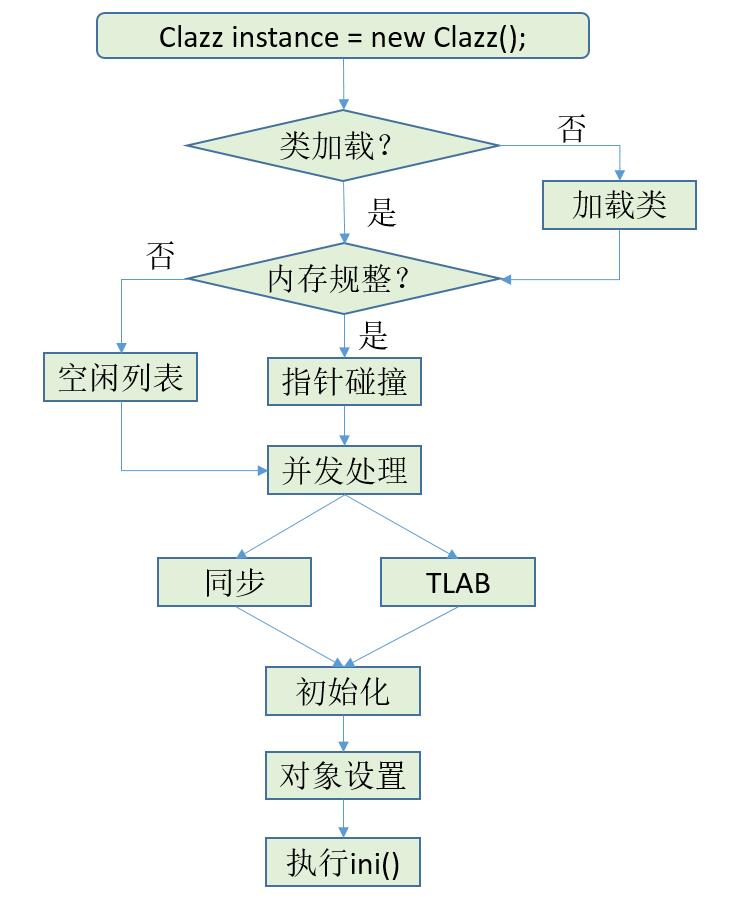
**对象的创建**

一个简单的创建对象语句Clazz instance = new Clazz();包含的主要过程包括了**类加载检查、对象分配内存、并发处理、内存空间初始化、对象设置、执行ini方法等**。

主要流程如下：



**1. 类加载检查**

JVM遇到一条new指令时，首先**检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，那必须先执行相应的类的加载过程。**

**2. 对象分配内存**

对象所需内存的大小在类加载完成后便完全确定（**对象内存布局**），为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。

根据Java堆中是否规整有两种内存的分配方式：（Java堆是否规整由所采用的垃圾收集器是否带有压缩整理功能决定）

* 指针碰撞(Bump the pointer)   
  Java堆中的内存是规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，**分配内存也就是把指针向空闲空间那边移动一段与内存大小相等的距离。**例如：Serial、ParNew等收集器。
* 空闲列表(Free List)   
  Java堆中的内存不是规整的，已使用的内存和空闲的内存相互交错，就没有办法简单的进行指针碰撞了。**虚拟机必须维护一张列表，记录哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，**并更新列表上的记录。例如：CMS这种基于Mark-Sweep算法的收集器。

**3. 并发处理**

对象创建在虚拟机中时非常频繁的行为，即使是仅仅修改一个指针指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况。

* 同步   
  **虚拟机采用CAS配上失败重试的方式保证更新操作的原子性**
* 本地线程分配缓冲（Thread Local Allocation Buffer, TLAB）   
  把内存分配的动作按照线程划分为在不同的空间之中进行，即**每个线程在Java堆中预先分配一小块内存（TLAB）。哪个线程要分配内存，就在哪个线程的TLAB上分配。只有TLAB用完并分配新的TLAB时**，才需要同步锁定。

**4. 内存空间初始化**

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）,如果使用了TLAB，这一工作过程也可以提前至TLAB分配时进行。

内存空间初始化保证了**对象的实例字段在Java代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。**

**注意：类的成员变量可以不显示地初始化（Java虚拟机都会先自动给它初始化为默认值）。方法中的局部变量如果只负责接收一个表达式的值，可以不初始化，但是参与运算和直接输出等其它情况的局部变量需要初始化。**

**5. 对象设置**

虚拟机对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的**元数据信息、对象的哈希码、对象的GC分代年龄等信**息。**这些信息存放在对象的对象头之中。**

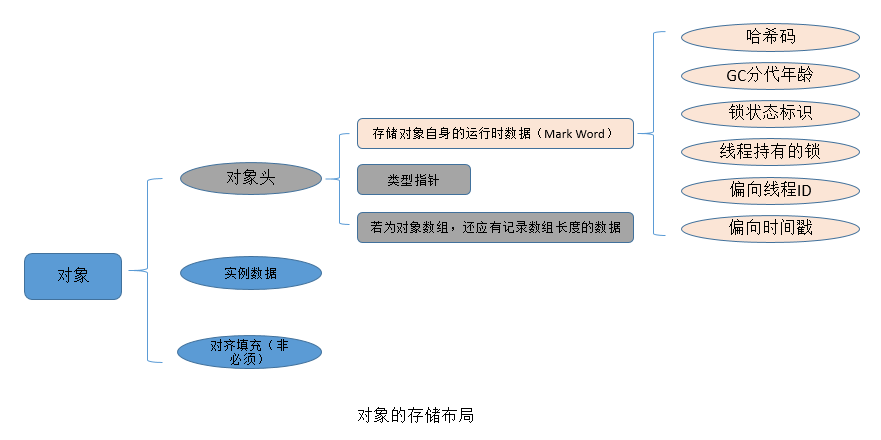
**6. 执行init()**

在上面的工作都完成之后，从虚拟机的角度看，一个新的对象已经产生了。但是从Java程序的角度看，对象的创建才刚刚开始init()方法还没有执行，所有的字段都还是零。

所以，一般来说（由字节码中是否跟随invokespecial指令所决定），**执行new指令之后会接着执行init()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产生出来。**

**对象内存布局**

在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头(Header)、实例数据(Instance Data)和对齐填充(Padding)。

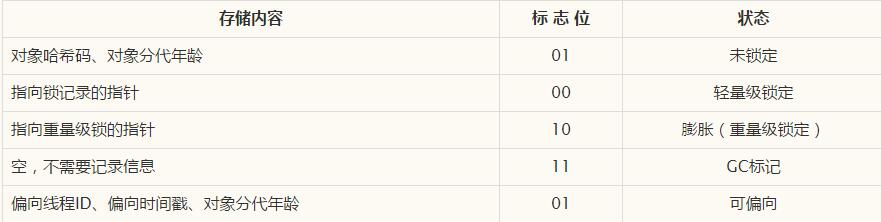


**对象头**

HotSpot虚拟机的对象头包括两部分信息：运行时数据和类型指针。

**运行时数据**

用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。



**类型指针**

即对象指向它的**类元数据的指针**，虚拟机**通过这个指针来确定这个对象是哪个类的实例。**

如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通Java对象的元数据信息确定Java对象的大小，但是从数组的元数据中无法确定数组的大小。   
(并不是所有的虚拟机实现都必须在对象数据上保留类型指针，换句话说，查找对象的元数据并不一定要经过对象本身，可参考对象的访问定位)

**实例数据**

实例数据部分**是对象真正存储的有效信息**，也是在程序代码中**所定义的各种类型的字段内容**。无论是从父类中继承下来的，还是在子类中定义的，都需要记录下来。HotSpot虚拟机默认的分配策略为longs/doubles、ints、shorts/chars、bytes/boolean、oops(ordinary object pointers)，从分配策略中可以看出，相同宽度的字段总是分配到一起。

**(我认为，实例数据是对象赋值的字段的数据，只是数据，然后你还需要去方法区寻找类里面定义的这些数据是什么，也就是这些数据的数据类型)**

**对齐填充**

* 对齐填充部分仅仅起到占位符的作用，并非必须。

**对象访问定位**

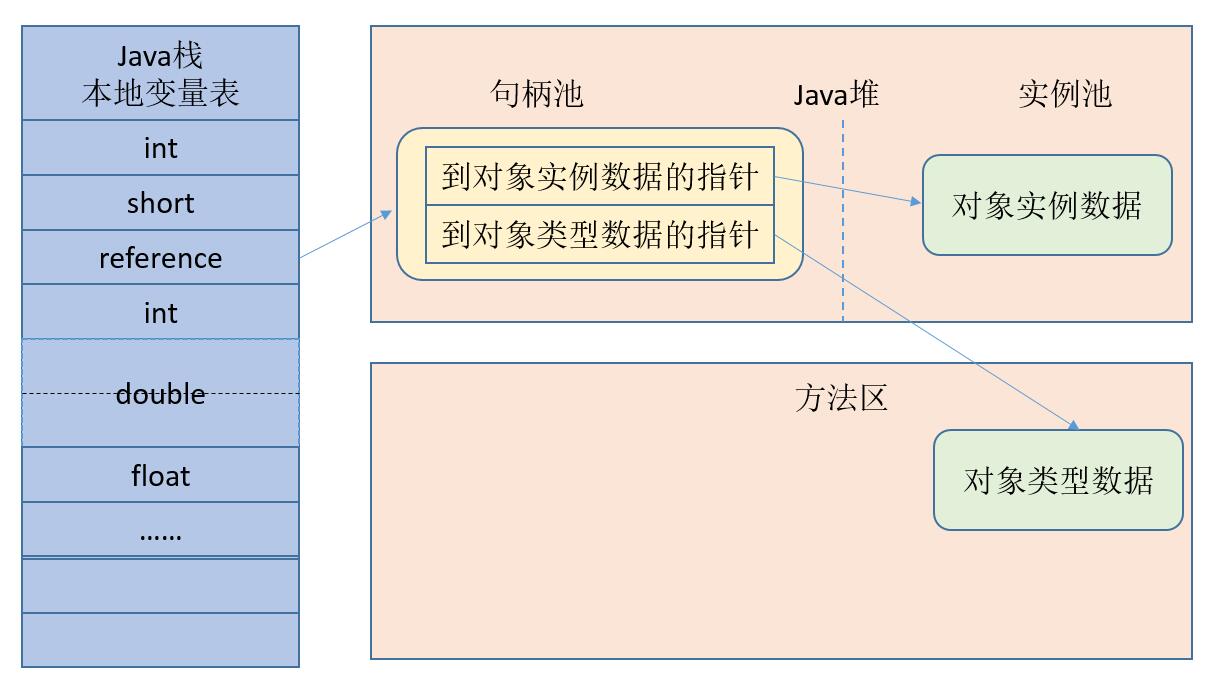
Java程序需要通过**栈上的引用数据来操作堆上的具体对象**。对象的访问方式取决于虚拟机实现，目前主流的访问方式有使用**句柄和直接指针**两种。

1. 使用句柄访问对象。即reference中存储的是对象句柄的地址，而句柄中包含了对象实例数据与类型数据的具体地址信息，相当于二级指针。
2. 直接指针访问对象。即reference中存储的就是对象地址，相当于一级指针**。**

　　两种方式有各自的优缺点。当垃圾回收移动对象时，对于方式一而言，reference中存储的地址是稳定的地址，不需要修改，仅需要修改对象句柄的地址；而对于方式二，则需要修改reference中存储的地址。从访问效率上看，方式二优于方式一，因为方式二只进行了一次指针定位，节省了时间开销，而这也是HotSpot采用的实现方式。下图是句柄访问与指针访问的示意图。

句柄，可以理解为指向指针的指针，维护指向对象的指针变化，而对象的句柄本身不发生变化；指针，指向对象，代表对象的内存地址。

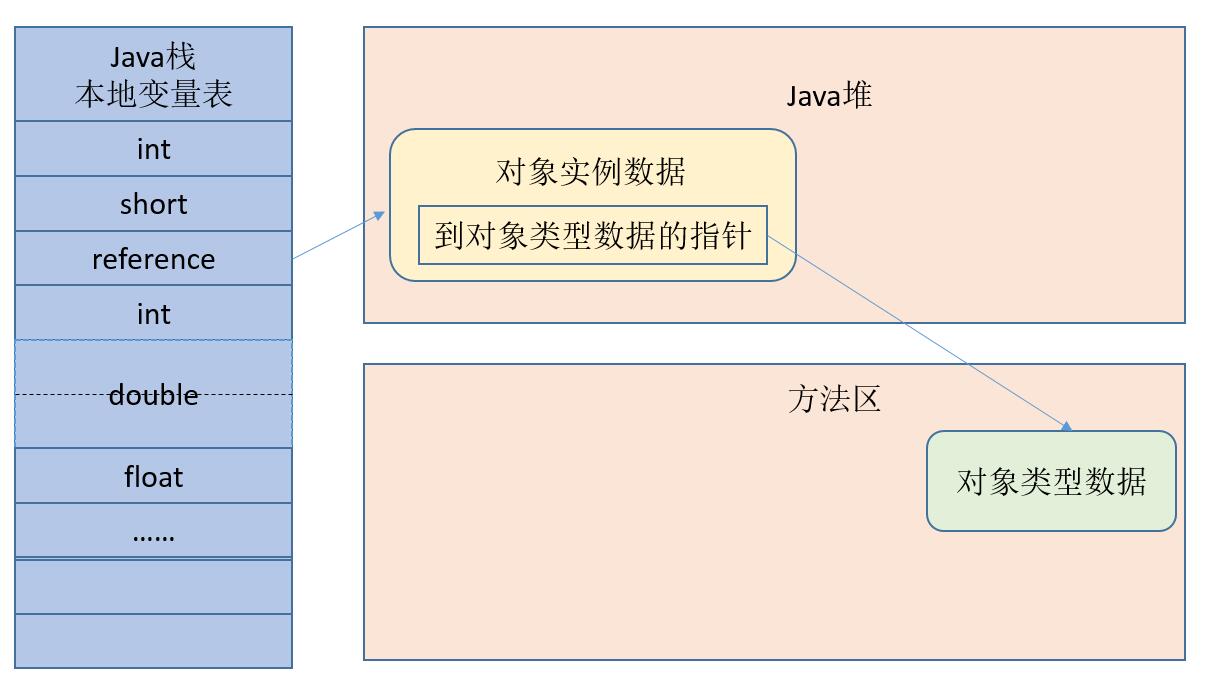
**句柄**   
Java堆中划分出一块内存来作为句柄池，引用中存储对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。



优势：**引用中存储的是稳定的句柄地址**，在对象被移动(垃圾收集时移动对象是非常普遍的行为)时**只会改变句柄中的实例数据指针**，而引用本身不需要修改。

**直接指针**

如果使用直接指针访问，那么Java堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而引用中存储的直接就是对象地址。



优势：速度更快，节省了一次指针定位的时间开销。由于对象的访问在Java中非常频繁，因此这类开销积少成多后也是非常可观的执行成本。（例如HotSpot）