1. 虚拟机HotSpot对象解析
2. 对象的创建
3. 检查类的符号引用

Java 虚拟机遇到一个字节码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，并且检查这个类是否已经加载，解析，初始化，如果没有，则首先需要加载类。

1. 对象分配内存

（1） 两种内存分配方式

类加载检查通过后，虚拟机就要为新的对象分配内存，对象所需的内存的大小在类加载完成后就能够确定。Java虚拟机为对象分配Java堆中的内存空间，若Java堆中的内存是绝对规整的，则通过指针划分已使用的内存和未使用的内存即可，这种分配方式称为“指针碰撞”；若Java堆的内存不规整，虚拟机需要维护一个空闲列表，记录空闲内存块，在分配时在列表中找出合适的内存块，这种分配方式称为“空闲列表”。

Java堆的内存是否规整由所采用的的垃圾收集器是否带有空间压缩能力决定，如Serial,ParNew等带压缩整理过程的垃圾收集器，Java堆使用的分配算法是指针碰撞；如CMS这种基于清除算法的收集器，只能使用“空闲列表”。

（2） 分配内存的线程安全问题

由于对象的创建操作十分频繁，可能会有线程安全问题。这里提供两种解决方法：

一种是对分配内存空间的动作进行同步处理；

另一种是将内存分配的动作按照线程划分在不同的空间中进行，每个线程在Java堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB），线程首先从TLAB中取内存，当TLAB用完时，再用同步锁定的方式申请新的内存。虚拟机是否使用TLAB,通过设置 -XX:+/-UseTLAB参数。

1. 对象初始化

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值，若有TLAB，TLAB分配是即可完成。初始化操作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。

1. 对象设置

之后，Java虚拟机会对对象进行必要的设置，如该对象属于哪个类，如何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中。

1. 构造函数

从虚拟机的角度，此时的对象已创建，但是对于Java程序来说，还没有执行构造函数，即Class文件中的<init>()方法还未执行，所有的字段为默认零值。执行完<init>()方法后，按照构造函数初始化了对象，才算完成对象的创建。

1. 对象的内存布局

HotSpot虚拟机中，对象在堆内存中的存储布局可以划分为三部分：对象头，实例数据，对齐填充。

1. 对象头

对象头包含两类信息，如下所示：

1. 对象运行时数据

如哈希码，GC分代年龄，锁状态标志等，这部分数据在32位和64位的虚拟机 中分别为32bit和64bit，官方称为“Mark Word”。这部分数据的数据结构是动态的，可在极小的空间内存储尽量多的数据。

1. 类型指针

对象指向它的类型元数据的指针，Java虚拟机通过这个指针确定该对象是哪个类的实例，但也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，即查找对象的元数据信息不一定要经过该对象本身。

1. 实例数据

对象存储的有效信息，即我们在代码中定义的各种类型的字段内容，包括父类继承的和子类定义的字段。这些字段的存储顺序会受到虚拟机分配策略参数（-XX:FieldsAllocationStyle参数）和字段定义顺序的影响。顺序一般是按照字段的类型字节的大小从大到小，相同大小的字段一起存放，且父类中定义的变量会在子类之前。

1. 对齐填充

就是一个占位符的作用，由于HotSpot虚拟机的自动内存管理要求对象起始地址必须是8字节的整数倍，即任何对象的大小是8字节的倍数，而对象头已经设计成8字节的倍数了，如果对象实例数据没有对齐的话， 就要通过对齐填充来对齐。

1. 对象的访问定位

Java程序会通过栈上的reference数据来操作堆上的具体对象，对象的访问方式是由虚拟机实现而定的，常用的有两种：句柄和直接指针。

1. 句柄

Java堆中划分出句柄池，reference中存储的是对象的句柄地址，而句柄中包含了Java堆中对象实例数据和方法区中类型数据各自的地址。使用句柄的好处是当对象被移动时，只会改变句柄中的实例数据指针，而不会改变reference中的数据。

1. 直接指针

不用在Java堆中分配句柄池，在Java对象的内存中不仅保存对象实例数据，还保存对象类型数据的指针，reference直接保存对象地址即可。相比于句柄，减少了一次指针定位的时间开销，速度更快，鉴于对象访问操作十分频繁，HotSpot使用直接指针更多一些。

1. Java内存溢出异常
2. Java堆溢出

Java堆用于存储对象实例，如果对象数量过多，且总容量到达了最大堆的容量限制，就会产生内存溢出异常。通过设置虚拟机的参数：

-Xms 表示堆的最小值;

-Xmx表示堆的最大值;

堆的最大值和最小值设置成一样的话表示堆不可扩展；

-XX:+HeapDumpOnOutOfMemoryError 表示在内存溢出异常出现时Dump出当前的内存堆转储快照

当我们获得了堆内存快照后，就可以使用内存映像分析工具分析，首先确认是出现了内存泄露还是内存溢出。内存泄露，是由于垃圾回收器无法回收这些内存导致的；内存溢出，是由于申请内存大于现有的内存导致的。

对于内存泄露，查看泄露对象到GC Roots的引用链，找到泄露对象是通过怎样的引用路径，与哪些GC Roots相关联，可以较精确地定位到这些对象创建的代码位置。

对于内存溢出，检查堆参数与机器的内存对比，向上调整，再从代码上调整，看看有哪些对象生命周期过长，尽量减少程序运行期间的内存消耗。

1. 虚拟机栈和本地方法栈溢出

HotSpot不区分虚拟机栈和本地方法栈，栈容量只能由-Xss参数来设定，且不支持动态扩展，因此除非在创建线程申请内存时就无法获得足够内存抛出OutOfMemeoryError异常，否则在线程运行时不会因为扩展而导致内存溢出，只会因为栈容量无法容纳新的栈帧导致StackOverflowError。

特殊情况，通过多线程的方式，在HotSpot上也可以出现内存溢出异常，但是这种异常与栈空间无直接关系，主要取决于操作系统的内存使用情况。若每个线程所分配的内存过大，所有线程的内存超过了操作系统分配给每个进程的内存限制，则会产生内存溢出异常。这时候，需要通过减少最大堆和减少栈容量的方法来换取更多的线程，这种通过减少内存的手段解决内存溢出比较少见。

1. 方法区和运行时和常量池溢出

JDK6以前，常量池在永久代中，通过-XX:permSize和-XX:MaxPermSize来限制永久代的大小，即限制常量池的容量。JDK7以后，使用元空间，常量池被移到Java堆中，不会再出现内存溢出异常。

String::intern()是一个本地方法，如果字符串常量池中已经包含一个等于此String对象的字符串，则返回池中这个String对象的引用，否则，会将此String对象包含的字符串添加到常量池中。JDK6以前，当首次使用String::intern()方法时，新建的String对象在Java堆中，而调用intern()方法后，该对象的引用也会存储到永久代的常量池中，不在Java堆中，这两个是完全不同的两个值。JDK7以后，常量池在Java堆中，该String对象调用intern()方法得到的引用与该对象是一致的，都在Java堆中。

对于元空间，HotSpot还是提供了一些防御措施：

-XX:MaxMetaspaceSize: 设置元空间最大值，默认是-1表示不限制

-XX:MetaspaceSize: 指定元空间初始大小，以字节为单位，达到该值就会触发垃圾收集进行类型卸载，同时调整空间大小。如果释放了大量空间，则降低该值；如果释放了少量空间，则适当提高该值。

1. 直接内存溢出

直接内存导致的内存溢出，特征是Heap Dump文件不会有明显的异常，且文件大小比较小。如程序直接过着间接地使用了DirectMemory(NIO)，就可以检查直接内存了。