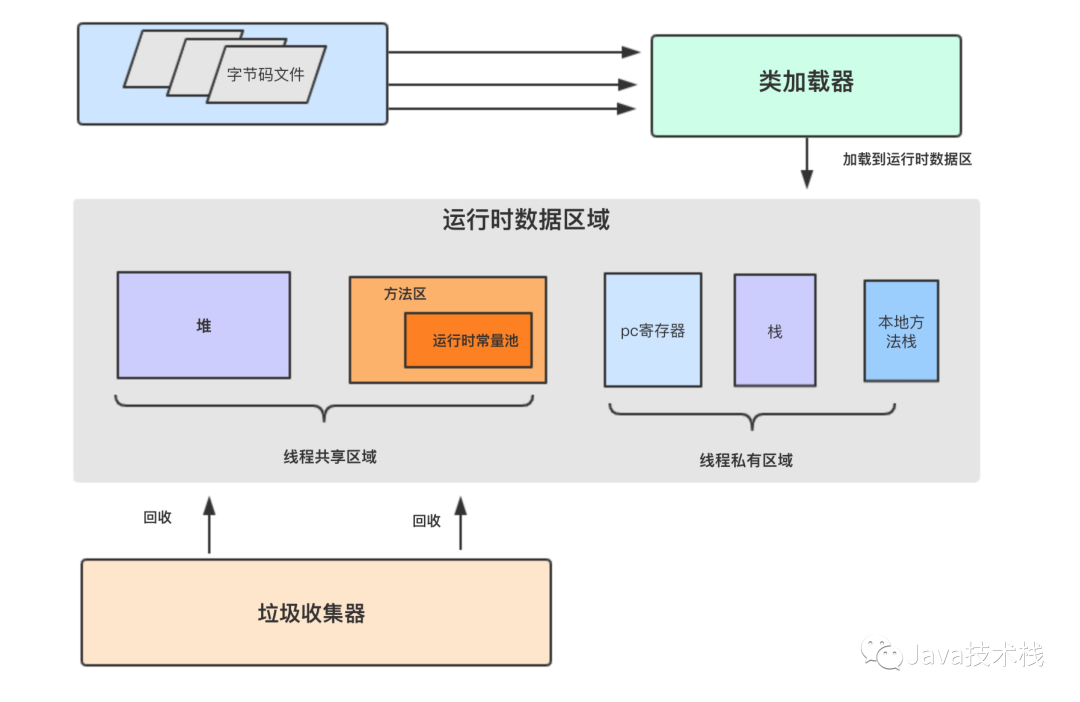
**1、什么是JVM**

Java程序的跨平台特性主要是指字节码文件可以在任何具有Java虚拟机的计算机或者电子设备上运行，Java虚拟机中的Java解释器负责将字节码文件解释成为特定的机器码进行运行。因此在运行时，Java源程序需要通过编译器编译成为.class文件。

JVM是JRE的一部分。它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。JVM有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。Java语言最重要的特点就是跨平台运行。使用JVM就是为了支持与操作系统无关，实现跨平台。所以，JAVA虚拟机JVM是属于JRE的，而现在我们安装JDK时也附带安装了JRE(当然也可以单独安装JRE)。

**2、JVM内存模型**



**栈区：**

栈分为java虚拟机栈和本地方法栈

1)  重点是Java虚拟机栈，它是线程私有的，生命周期与线程相同。

2)  每个方法执行都会创建一个栈帧，用于存放局部变量表，操作栈，动态链接，方法出口等。每个方法从被调用，直到被执行完。对应着一个栈帧在虚拟机中从入栈到出栈的过程。

3)  通常说的栈就是指局部变量表部分，存放编译期间可知的8种基本数据类型，及对象引用和指令地址。局部变量表是在编译期间完成分配，当进入一个方法时，这个栈中的局部变量分配内存大小是确定的。

5)  本地方法栈 为虚拟机使用到本地方法服务（native）JNI

**堆区：**

1)  堆被所有线程共享区域，在虚拟机启动时创建，唯一目的存放对象实例。

2)  堆区是gc的主要区域，通常情况下分为两个区块年轻代和年老代。更细一点年轻代又分为Eden区，放新创建对象，From survivor 和 To survivor 保存gc后幸存下的对象，默认情况下各自占比 8:1:1。

**方法区：**

1)  被所有线程共享区域，用于存放已被虚拟机加载的**类信息，常量，静态变量**等数据。被《Java虚拟机规范》描述为堆的一个逻辑部分(非堆）。习惯是也叫它永久代(元空间)

2)  垃圾回收很少光顾这个区域，不过也是需要回收的，主要针对常量池回收，类型卸载。

3)  常量池具有一定的动态性，里面可以存放编译期生成的常量；运行期间的常量也可以添加进入常量池中，比如string的intern()方法。

**程序计数器：**

1)  当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。

2)  Java虚拟机多线程是通过线程轮流切换并分配处理器执行时间的方式实现的。为了线程切换能恢复到正确的位置，每条线程都需要一个独立的程序计数器，所以它是线程私有的。

**3、JVM垃圾回收算法**

**1. 标记–清除算法**

标记：遍历内存区域，对需要回收的对象打上标记。

清除：再次遍历内存，对已经标记过的内存进行回收。

**2. 标记–复制算法（年轻代）**

将内存划分为等大的两块，每次只使用其中的一块。当一块用完了，触发GC时，将该块中存活的对象复制到另一块区域，然后一次性清理掉这块没有用的内存。下次触发GC时将那块中存活的的又复制到这块，然后抹掉那块，循环往复。

**3. 标记–整理算法（老年代）**

标记：对需要回收的进行标记

整理：让存活的对象，向内存的一端移动，然后直接清理掉没有用的内存。

**4、GC的两种判定方法：引用计数与引用链。**

**1. 引用计数算法**

在对象头中分配一个空间来保存该对象被引用的次数。如果该对象被其它对象引用，则它的引用计数加一，如果删除对该对象的引用，那么它的引用计数就减一，当该对象的引用计数为0时Minor GC，那么该对象就会被回收。

**2. 引用链**

确定一系列的GC Root的对象作为起点，从这些节点向下搜索，搜索的路径称为引用链（Reference Chain），当一个对象无法通过引用链连向GC Root的话证明此对象是不可用的

**5、GC收集器有哪些？CMS收集器与G1收集器的特点。**

**1. CMS收集器——**基于“标记-清除”算法

**过程：**初始标记->并发标记->重新标记->并发清理

初始标记、重新标记这两个步骤仍然需要“stop the world”，初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing，而重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生表动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长点，但远比并发标记的时间短。



**优点：**

并发收集、低停顿

**缺点：**

1）CMS收集器对CPU资源非常敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程而导致应用程序变慢，总吞吐量会降低。

2）CMS收集器无法处理浮动垃圾

浮动垃圾：由于CMS并发清理阶段用户线程还在运行着，伴随着程序运行自然就会有新的垃圾不断产生，这部分垃圾出现的标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC中再清理。这些垃圾就是“浮动垃圾”。

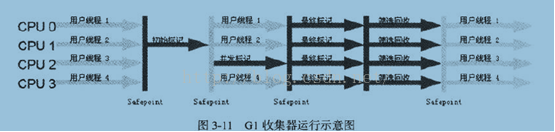
3）CMS是一款“标记--清除”算法实现的收集器，容易出现大量空间碎片。

**2. G1收集器——**基于“标记-整理”算法

G1是一款面向服务端应用的垃圾收集器

**过程：**初始标记->并发标记->最终标记->筛选回收

在筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。



**特点：**

1、并行于并发：G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短stop-The-World停顿时间。

2、分代收集：G1将新生代，老年代的物理空间划分取消了。取而代之的是，G1算法将堆划分为若干个区域（Region），它仍然属于分代收集器。不过，这些区域的一部分包含新生代，新生代的垃圾收集依然将存活对象拷贝到老年代或者Survivor空间。老年代也分成很多区域，G1收集器通过将对象从一个区域复制到另外一个区域，完成了清理工作。

3、空间整合：与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。

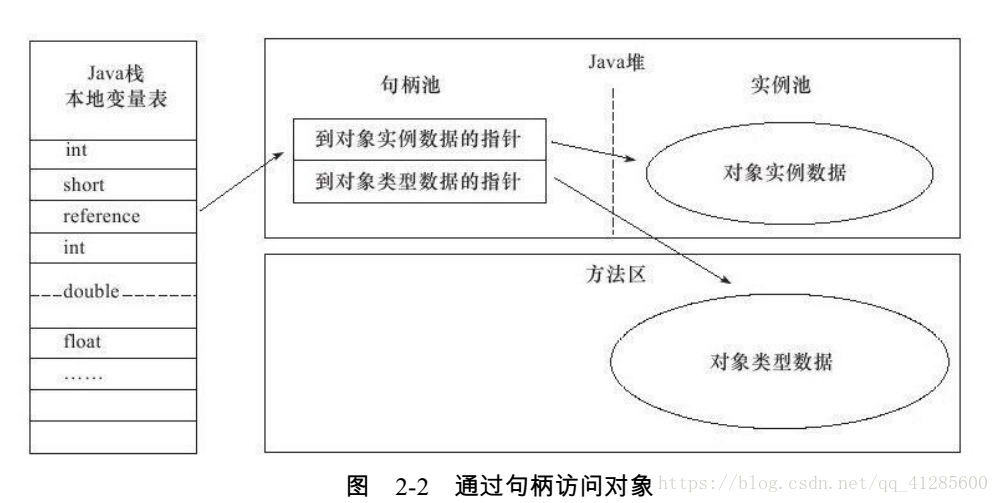
4、可预测的停顿：这是G1相对于CMS的另一个大优势，降低停顿时间是G1和CMS共同的关注点，但Ｇ１除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，

**\*\*6、对象的访问定位**

Java是通过虚拟机栈中的局部变量表中的reference数据来操作Java堆上的具体对象。但reference只是虚拟机规范中规定指向一个对象的引用，它并没有定义这个引用通过何种方式去定位、访问堆中的对象的具体位置，所以对象访问方法也取决于虚拟机的实现而定的。目前主流的访问方式有使用**句柄**和**直接指针**两种。

**一、句柄访问**

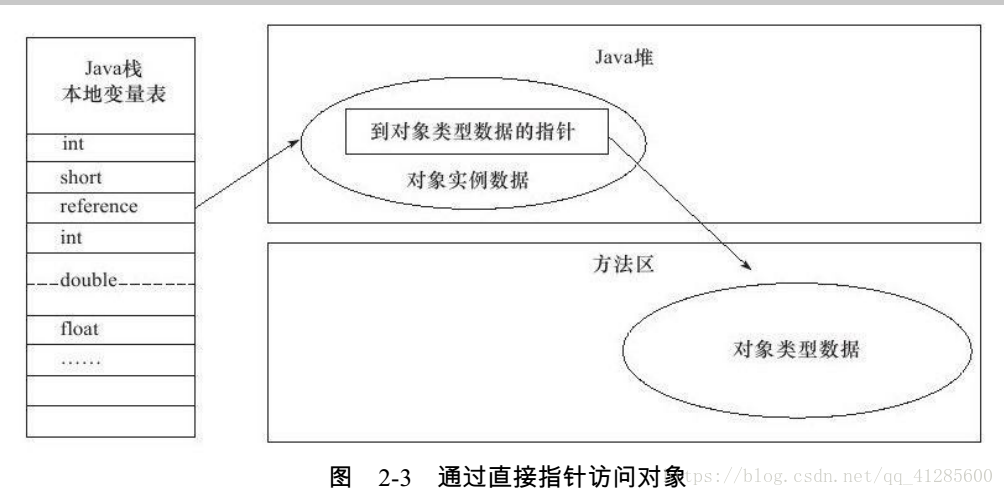
     如果使用句柄访问，Java堆中将会划分出一块儿内存作为**句柄池**，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和对象类型数据的具体地址信息。实际上是采用了句柄池这样一个中间介质进行了两次指针定位，有效的避免了对象的移动或改变直接导致reference本身发生改变。句柄访问方式如下图所示：



     使用句柄访问最大的好处就是reference中存储的是稳定的句柄地址，在对象回收过程中或者其它对象需要移动的时，只会改变句柄中的实例数据的指针，而reference本身不需要做任何修改。

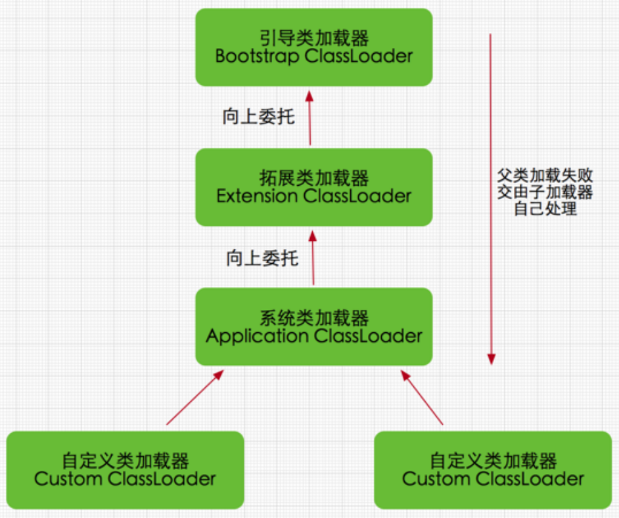
**二、直接指针访问**

     如果使用直接指针访问，那么Java堆对象的布局必须考虑如何放置访问类型的数据的相关信息，而reference中存储的直接就是对象地址，而不再是句柄地址信息，相当于在reference与对象地址信息直接少了句柄池这样一个中间地址，reference中直接存储的就是对象地址。



     这种定位方式也就导致了在对象被移动时，reference本身必须发生改变。但是我们都知道，使用句柄访问方式时，相当于进行了两次指针定位，而直接指针访问方式恰好节省了这一次指针定位的时间开销，由于对象的访问在Java中非常的频繁，时间开销的减少也是一种可观的执行成本。例如，常见的HotSpot虚拟机就使用的是直接指针访问方式。

**7、类加载机制——双亲委派**



如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式  
1. 带有优先级的层次关系，通过这种层级关可以避免类的**重复加载**

2. 安全因素，java**核心api**中定义类型不会被随意替换，

**\*\*8、内存分配机制**

**栈上分配：**

JVM通过逃逸分析确保对象不会被方法外部所引用，JVM不会创建该对象，而是将该对象成员变量分解若干个被这个方法使用的成员变量所代替，这些代替的成员变量在栈帧或寄存器上分配空间。

**堆分配：**

1、大对象直接进入老年代：需要连续空间内存的对象(字符串、数组)，为了避免对象在S0-S1之间复制效率低下的问题

2、长期存活的对象进入老年代：内存回收的时候必须要识别哪些对象应该在新生代，那些对象在老年代。这里判断的重要依据就是GC对象的年龄，Young GC过后，年轻代依然存活的对象如果年龄达到了JVM所规定的GC年龄，则会被移动到老年代中。

3、对象动态年龄判断：当前放对象的Survivor区域里(其中一块区域，放对象的那块s区)，一批对象的总大小大于这块Survivor区域内存大小的50%，那么此时大于等于这批对象年龄最大值的对象，就可以直接进入老年代了。动态年龄判断机制一般是在Young GC之后触发的

4、老年代空间分配担保机制：年轻代每次minor gc之前JVM都会计算下老年代剩余可用空间，如果这个可用空间小于年轻代里现有的所有对象大小之和(包括垃圾对象)，就会触发一次Full GC，回收完还是没有足够空间存放新的对象就会发生"OOM"

**9、类加载的过程**

1)  加载：根据查找路径找到相应的class文件，然后导入。类的加载方式分为隐式加载和显示加载两种。隐式加载指的是程序在使用new关键词创建对象时，会隐式的调用类的加载器把对应的类加载到jvm中。显示加载指的是通过直接调class.forName方法来把所需的类加载到jvm中。

2)  验证：检查加载的class文件的正确性。

3)  准备；给类中的静态变量分配内存空间，设置变量初始值。

4)  解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址。

5)  初始化：对静态变量和静态代码块执行初始化工作。

**10、Minor GC和Full GC触发条件**

一、Minor GC触发条件

1、eden区满时，触发MinorGC。即申请一个对象时，发现eden区不够用，则触发一次MinorGC。

二、Full GC触发条件

1、老生代空间不够分配新的内存

2、Minor GC前，eden space和from space区大小大于to space且大于老年代内存，触发Full GC。

**\*\*11、JDBC破坏双亲委派机制**

JDBC的Driver接口定义在JDK中，其实现由各个数据库的服务商来提供， DriverManager 类中要加载各个实现了Driver接口的类，然后进行管理，但是DriverManager位于 $JAVA\_HOME中jre/lib/rt.jar 包，由BootStrap类加载器加载，而其Driver接口的实现类是位于服务商提供的 Jar 包，**根据类加载机制，当被装载的类引用了另外一个类的时候，虚拟机就会使用装载第一个类的类装载器装载被引用的类。**也就是说BootStrap类加载器还要去加载jar包中的Driver接口的实现类。我们知道，BootStrap类加载器默认只负责加载 $JAVA\_HOME中jre/lib/rt.jar 里所有的class，所以需要由子类加载器去加载Driver实现，这就破坏了双亲委派模型。在启动类加载器中有方法获取应用程序类加载器，然后通过它去加载就可以了。这就是所谓的线程上下文加载器。

**12、对象创建方法、内存分配**

**对象创建方法：**

　　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、连接和初始化过。

如果没有，那必须先执行相应的类的加载过程。

**对象的内存分配：**

　　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。

**指针碰撞**：（标记整理）所有用过的内存在一边，空闲内存在另一边，中间放着一个指针作为分界点的指示器，分配内存就是把指针往空闲内存那边挪一段与对象大小相等的距离。

**空闲列表**：（标记清除）虚拟机维护一个列表，记录哪些内存是可用的，分配的时候从列表中找到一块足够大的空间划分给对象，并更新列表。

**13、静态分派、动态分派**

**静态分派**：方法重载 所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派 编译期就能确定

**动态分派**：方法重写 运行时才能确定