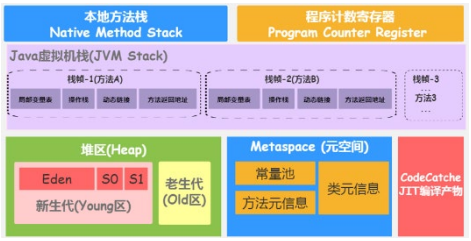
1、JVM 内存结构

方法区、堆、程序计数器、本地方法栈、虚拟机栈：其中方法区、堆是一个系统中线程共享的，随着虚拟机启动而创建，随着虚拟机退出而销毁；而程序计数器、本地方法栈、虚拟机栈是线程私有的，即与线程一一对应，随着线程的开始和结束而创建和销毁。



2、程序计数器是什么？作用？

PC 寄存器用来存储**指向下一条指令的地址**，也即将要执行的指令代码。**执行引擎**的字节码解释器工作时就是通过**改变这个计数器的值**来**选取下一条需要执行的字节码指令**。它是程序控制流的**指示器**，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。（**不存在**溢出和垃圾回收）

3、为什么执行 native 方法时，是 undefined？

任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器**会存储**当前线程正在**执行的 Java 方法**的 **JVM 指令地址**；但是，如果是在**执行 native 方法**，则是**未指定值**（undefined），因为 **native 本地方法**是大多是**通过 C 实现**，并**未编译成**需要执行的**字节码指令**，所以在计数器中当然是空。

4、程序计数器的基本特征

1）它是一块**很小的内存空间**，几乎可以忽略不记。也是**运行速度最快的存储区域**。不会随着程序的运行需要更大的空间。

2）在 JVM 规范中，每个线程都有它自己的程序计数器，是**线程私有的，生命周期与线程的**生命周期**保持一致**。

3）**唯一一个**在 Java 虚拟机规范中**没有**规定任何 **OutOtMemoryError**(OOM)情况的**区域**。

5、使用 PC 寄存器存储字节码指令地址有什么用呢？（为什么使用 PC 寄存器记录当前线程的执行地址呢？）

因为 **CPU** 需要**不停的切换各个线程**，这时候**切换回来**以后，就**得知道接着从哪开始**继续执行。**JVM 的字节码解释器**就需要通过**改变 PC 寄存器的值**来**明确下一条**应该执行什么样的**字节码指令**。

6、PC 寄存器为什么会被设定为线程私有？

多线程在一个**特定的时间段内**只会**执行**其中**某一个线程的方法**，CPU 会不停地做**任务切换**，必然**导致经常中断或恢复**，所以为每一个线程都分配一个 PC 寄存器，能够准确地**记录各个线程**正在**执行**的**当前字节码指令地址**，使得**各个线程之间**便可以进行**独立计算**，从而**不会相互干扰**。

7、Java 中堆和栈有什么区别？

1）GC 和 OOM：

1>栈不存在 GC，方法结束会清除相应栈帧，但存在 OOM，栈溢出通常是深度递归或方法中的局部变量表太大；堆用来存放对象实例的区域，会频繁创建和销毁对象，GC 的主要关注区域，当内存不足会报错 OOM；

2）栈和堆执行效率：

栈的访问速度比堆快，仅次于程序计数器，因为栈后进先出，仅仅移动栈顶指针；

3）内存大小和数据结构：

栈的内存远小于堆，通常在程序启动时确定大小，而堆可以动态调整大小；

4）栈管运行和堆管存储：

栈主管 Java 程序的运行，它保存方法的局部变量(8 种基本数据类型、对象的引用地址)、**部分结果**，并参与**方法的调用和返回**。堆存储具体的实例，包括成员变量、引用类型变量。

8、栈可能抛出的异常？

Java 栈的大小是**动态的**或者是**固定不变**的。

1）**固定大小时**，线程请求分配的**栈容量超过** Java 虚拟机栈**允许的最大容量**，会抛出一个 StackOverflowError 异常；

2）**动态扩展大小时**，无法申请到足够的内存，或者在**创建新的线程时**没有足够的内存去创建对应的虚拟机栈，OutOfMemoryError 异常。

9、Java 中，栈的大小通过什么参数来设置？

-Xss size；一般默认为 512k-1024k，**取决于操作系统**。**栈的大小**直接决定了**函数调用的**最大可达**深度**，但是，设置的**栈空间值过大**，会导致系统可以用于**创建线程的数量减少**。

【注意】jdk5.0 之前，默认栈大小：256k；Jdk5.0 之后，默认栈大小：1024k；

10、方法和栈桢之间存在怎样的关系？

在一个线程上**正在执行的每个方法**都各自**对应一个栈帧**，**栈帧**是一个**内存区块**，**维系着**对应方法执行过程中的**各种数据信息**。

在**一条活动线程中**，**一个时间点上**，**只会**有**一个活动的栈帧**。即只有当前正在执行的方法的栈帧（**栈顶栈帧**）是有效的，这个栈帧被称为**当前栈帧**，与当前栈帧相对应的方法就是**当前方法**，定义这个方法的类就是**当前类**。

如果在该方法中**调用了其他方法**，则**创建**对应的**新栈帧**，放在**栈顶**，成为**新的当前帧**。

【注意】**执行引擎**运行的**所有字节码指令**只针对**当前栈帧**进行操作。

11、栈的 FILO 原理

1）JVM 对 Java 栈的直接操作只有两个：每个方法执行时进栈；执行结束后出栈。遵循“先进后出，后进先出”原则。

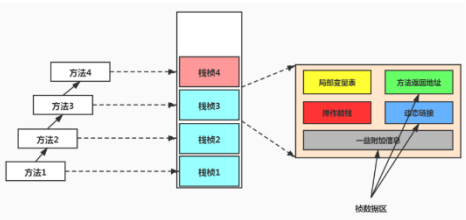
2）**不同线程**中所包含的**栈帧**是**不允许**存在**相互引用的**，即**一个栈帧**中**不能引用另外一个线程的**栈帧。

3）如果当前方法**调用了**其他方法，**方法返回时**，**当前栈帧**会**传回**此方法的**执行结果**给**前一个栈帧**，接着，虚拟机会**丢弃当前栈帧**，使得**前一个栈帧**重新**成为当前栈帧**。

4）Java 方法有**两种返回函数**的方式，一种是使用 **return 指令**的**正常的函数返回**，；另外一种是**抛出异常**。二者都会导致**栈帧被弹出**。

12、栈帧内部结构

每个栈帧中存储着：局部变量表、操作数栈、动态链接（指向运行时常量池的方法引用）、方法返回地址（方法正常退出或者异常退出的定义）、一些附加信息。



13、局部变量表（局部变量数组）

定义为一个数字数组，主要用于**存储方法参数**和**定义在方法体内的局部变量**，这些数据类型包括各类基本数据类型(8 种)、对象引用（reference），以及 returnAddress 类型。

1）局部变量表所需的**容量大小**是在**编译期确定**下来的，并保存在方法的 **Code 属性**的 **maximum local variables 数据项**中。在方法**运行期间**是**不会改变**局部变量表的**大小**的。

2）方法**嵌套调用的次数**由**栈的大小**决定。栈**越大**，方法嵌套**调用次数越多**。对一个函数而言，它的参数和局部变量**越多**，使得**局部变量表膨胀**，它的**栈帧就越大**，函数调用就会**占用更多的栈空间**，导致其**嵌套调用次数**就会**减少**。

3）**局部变量表中的变量**只在**当前方法调用中有效**。在**方法执行时**，虚拟机通过使用**局部变量表**完成**参数值**到**参数变量列表**的**传递**过程。当方法调用**结束后**，随着**方法栈帧的销毁**，局部变量表也会**随之销毁**。

【注意】

1>在 Class 文件的局部变量表中，显示了每个局部变量的**作用域范围**、**所在槽位的索引**(index 列)、**变量名**(name 列)和**数据类型**。

2>由于**局部变量表**是建立在线程的栈上，是**线程的私有数据**，因此**不存在**数据**线程安全问题**。

3>**实体方法**中需要存储一个 **this 参数**，long 和 double 占用两个槽位 slot。

4>**局部变量表中的变量**也是重要的**垃圾回收根节点**，只要被局部变量表中**直接或间接**引用的**对象**都不会被回收。

14、关于 Slot 的理解

1）**参数值的存放**总是在局部变量数组的 **index 为 0 开始**，到**数组长度-1** 的索引**结束**；

2）局部变量表，**最基本的存储单元**是 **Slot**；

3）在局部变量表里，**32 位以内的类型**只占用**一个 slot**（包括 returnAddress 类型），**64 位的类型**（long 和 double)占用**两个 slot**。

【注意】byte 、short 、char 在**存储前被转换为** int，boolean 也**被转换为** int，**0** 表示 false，**非 0** 表示 true。

4）**JVM** 会为局部变量表中的**每一个 Slot** 都**分配一个访问索引**，通过这个**索引**即可成功**访问到**局部变量表中**指定的局部变量值**；

5）当一个实例方法被调用的时候，它的方法**参数**和方法**体内部定义的局部变量**将**会按照顺序**被**复制到**局部变量表中的**每一个 Slot 上**；

6）如果需要**访问**局部变量表中一个 **64bit 的**局部变量值时，只需要**使用前一个索引**即可。

7）如果**当前帧**是由**构造方法**或**实例方法**创建的，那么**该对象引用 this** 将会存放在 **index 为 0 的 slot 处**，其余的参数按照参数表顺序继续排列。

8）栈帧中**局部变量表的槽位**是**可以重用**的，如果一个局部变量**过了其作用域**，那么在其作用域**之后申明的新的局部变量**就很有可能**会复用**过期局部变量的**槽位**，从而达到节省资源的目的。

```
public class SlotTest {
    public void localVar1() {
        int a = 0;
        System.out.println(a);
        int b = 0;
    }

    public void localVar2() {
        {
            int a = 0;
            System.out.println(a);
        }
        //此时的b就会复用a的槽位
        int b = 0;
    }
}
```

15、操作数栈

操作数栈，用于**保存**计算过程的**中间结果**，同时作为计算过程中**变量临时的存储空间**。

1）操作数栈是 **JVM 执行引擎**的一个**工作区**，当一个方法**刚开始执行**的时候，一个**新的栈帧**也会随之被创建出来，这个方法的**操作数栈是空的**；

2）每一个操作数栈都会拥有一个**明确的栈深度**用于存储数值，其所需的**最大深度在编译期**就**定义好**；

3）栈中的任何一个元素可以是任意 Java 数据类型。（32bit→1 栈单位深度，64bit→2 栈~）

4）操作数栈，在**方法执行过程**中，**根据字节码指令，并非**采用访问索引的方式来进行数据访问的，**而是**只能通过标准的**入栈**和**出栈**操作，往栈中**写入数据**或**提取数据**来完成一次数据访问。

5）如果被调用的方法**带有返回值**的话，**其返回值**将会被**压入当前栈帧的操作数栈**中，并**更新 PC 寄存器**中下一条需要执行的字节码指令。

16、什么是栈顶缓存技术？

由于**操作数**是**存储在内存中**的，因此**频繁地**执行**内存读/写操作**必然会**影响执行速度**。栈顶缓存技术是将**栈顶元素全部缓存**在**物理 CPU 的寄存器**中，以此**降低对内存的读/写次数**，提升执行引擎的执行效率。

17、动态链接（指向运行时常量池的方法引用）

每一个栈帧内部都包含一个**指向运行时常量池**中**该栈帧所属方法的引用**，然后通过**常量池中**找到**指向方法的符号引用**并***转换***为**直接引用**，以**动态链接**该方法对其他方法的调用。

18、方法中定义的局部变量是否线程安全?

不一定安全，1）如果局部变量在内部产生并在内部消亡的，那就是线程安全的。2）如果局部变量表里的局部变量是一个引用变量，它指向堆空间的一个对象，这时考虑的就不是这个引用变量本身安不安全，而是考虑堆空间中被引用的这个对象安不安全，所以需要看这个对象有没有被其他方法线程引用，若存在则不安全，反之则安全。

1> s1 的声明方式是线程安全，因为线程私有，在线程内创建的 s1，不会被其它线程调用。

```
public static void method1() {
    //StringBuilder:线程不安全
    StringBuilder s1 = new StringBuilder();
    s1.append("a");
    s1.append("b");
    //...
}
```

2> stringBuilder 的操作过程：是线程不安全的，因为 stringBuilder 是外面传进来的，有可能被多个线程调用。

```
public static void method2(StringBuilder stringBuilder) {
    stringBuilder.append("a");
    stringBuilder.append("b");
    //...
}
```

3> stringBuilder 的操作：是线程不安全的；因为返回了一个 stringBuilder，stringBuilder 有可能被其他线程共享。

```
public static StringBuilder method3() {
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("a");
    stringBuilder.append("b");
    return stringBuilder;
}
```

4> stringBuilder 的操作：是线程安全的；因为返回了一个 stringBuilder.toString()相当于 new 了一个 String，所以 stringBuilder 没有被其他线程共享的可能。

```
public static String method4() {
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("a");
    stringBuilder.append("b");
    return stringBuilder.toString();
}
```

19、什么是本地方法?

本地方法就是 Java 调用**非 Java 代码的接口**，该方法的实现由**非 Java 语言实现**，比如 C。【注意】**标识符 native** 可以与所有其它的 java 标识符连用，但是 **abstract 除外**。

20、为什么使用本地方法?

1）与 Java 环境外交互：Java 需要与一些**底层系统**，如**操作系统**或某些**硬件交换信息**时的情况。本地方法正为我们**提供**了一个非常**简洁的接口**，而且我们**无需**去**了解** Java 应用之外的**繁琐的细节**。

2）与操作系统交互：通过使用**本地方法**，我们得以**用 Java 实现**了 **jre** 的与**底层系统**的交互，甚至 JVM 的一些部分就是用 C 写的。

21、本地方法栈

Java **虚拟机栈**用于管理 **Java 方法的调用**，而**本地方法栈**用于管理**本地方法的调用**。本地方法栈中**登记本地方法**，在执行引擎执行时**加载本地方法库**，，也是**线程私有的**。

22、堆

1）一个 **JVM 实例**只存在**一个堆内存**，Java 堆区在 JVM **启动时即被创建**，并确定其空间大小（堆内存大小可以调节），是 JVM 管理的**最大一块内存空间**。

2）堆可以处于**物理上不连续**的内存空间中，但在**逻辑上**被视为**连续的**。

3）堆，是 GC 执行**垃圾回收**的重点区域，在**方法结束后**，堆中的对象**不会马上被移除**，仅仅**在垃圾收集时**才会被移除。

23、对象都分配在堆上?

所有的对象实例以及**数组**都应当**在运行时分配在堆上**。数组和对象可能永远不会存储在栈上，因为**栈帧中保存引用**，这个引用**指向堆中对象或数组**的位置。

【特殊情况】**栈上分配**：对于一些对象**只在当前方法中使用**，那么可以直**接分配到栈帧**中，运行速度快并随着方法出栈而消亡，但仅仅只是做了**标量替换**，并非整个对象都放栈中。

24、所有的线程都共享堆?

所有的线程共享 Java 堆，但是除了线程私有的缓冲区(TLAB)。因为在 JVM 中**创建对象实例很频繁**，在**并发环境下**从对空间中**划分内存空间**是线程不安全的，使用**加锁等机制**避免多线程操作同一地址会**影响分配速度**。所以多线程同时分配内存时，使用 **TLAB** 可以**避免**一系列的**线程安全问题**，同时还能够**提升**内存分配的**吞吐量**，被称为**快速分配策略**。

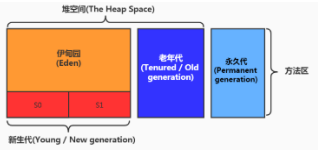
25、什么是 TLAB?

从内存模型而不是垃圾收集的角度，对 **Eden 区域**继续进行**划分**，JVM **为每个线程分配**了一个**私有缓存区域**。默认情况下，**TLAB 空间**的**内存非常小**，仅占有整个 Eden 空间的 **1%**。【注意】**不是所有的对象实例**都能够在 TLAB 中**成功分配内存**，**但** JVM 确实是将 **TLAB** 作为**内存分配的*首选***。一旦对象在 TLAB 空间分配内存**失败时**，JVM 就会尝试着通过**使用加锁机制**确保数据操作的原子性，从而直接在 Eden 空间中**分配内存**。

26、堆的内部结构

1）Java 7 及之前，堆内存逻辑上分为三部分：新生代+老年代+永久代；

2）Java 8 及之后，堆内存逻辑上分为三部分：新生代+老年代+元空间。



27、JVM 内存分区，为什么要有新生代和老年代?

JVM 中的 Java 对象可以被划分为两类：

1）一类是**生命周期较短**的**瞬时对象**，这类对象的创建和消亡都非常迅速；2）另外一类对象的**生命周期非常长**，在某些极端的情况下甚至**与 JVM 的生命周期保持一致**。不同对象的生命周期不同，优化 GC 性能。

【注意】**几乎所有的 Java 对象**都是在 Eden 区被 new 出来的。特例：**大对象**直接分配到老年代，尽量避免程序中出现过多的大对象。

28、如何设置堆内存大小?

“**-Xms**”用于表示堆区的**起始内存**，等价于-XX:InitialHeapSize；

“**-Xmx**”则用于表示堆区的**最大内存**，等价于-XX:MaxHeapSize；

一旦**堆区中内存大小***超过*“-Xmx”**所指定的最大内存**时抛出 OutOfMemoryError:heap 异常。

【注意】1）通常会将**-Xms 和-Xmx** 两个参数**配置相同的值**，其**目的**是为了能够在 **java 垃圾回收机制清理完堆区后***不需要***重新分隔计算堆区的大小**，从而**提高性能**。

2）heap 默认**最大值**计算方式：如果物理内存**少于 192M**,那么 heap 最大值为物理内存的**一半**。如果物理内存**大于等于 1G**，那么 **heap 的最大值**为物理内存的 **1/4**。

3）heap 默认**最小值**计算方式：最少**不得少于 8M**，如果物理内存**大于等于 1G**，那么默认值为物理内存的 **1/64**，即 1024/64=16M。最小堆内存在 jvm 启动的时候就会被初始化。

29、如何设置新生代与老年代比例?

默认-XX:**NewRatio**=2，表示新生代占 1，**老年代**占 2，新生代占整个堆的 1/3；

可以修改-XX:NewRatio=4，表示新生代占 1，老年代占 4，新生代占整个堆的 1/5。

30、如何设置 Eden、幸存者区比例?

在 HotSpot 中，**Eden 空间**和**另外两个 Survivor** 空间缺省**所占的比例**是 8:1:1；开发人员可以通过选项“**-XX:SurvivorRatio**”调整这个空间比例。比如-XX:SurvivorRatio=8。

31、对象分配策略

如果对象在 Eden 出生并经过第一次 MinorGC **后仍然存活**，并且**能被 Survivor 容纳**的话，将被**移动到 Survivor 空间中**，并将**对象年龄设为 1**。对象在 **Survivor 区中每熬过一次 MinorGC**，年龄就**增加 1 岁**，当它的年龄增加到**一定程度**（默认为 **15 岁**，其实每个 JVM、每个 GC 都有所不同）时，就会**被晋升到老年代中**。

32、对象分配具体过程

1）new 的对象**先放 Eden 区**,此区有大小限制。

2）当 **Eden 区**空间**填满时**，程序**又需要创建对象**，JVM 的**垃圾回收器**将对 **Eden 区**进行垃圾回收(Minor GC/YGC)，将 **Eden 区中的不再被**其他对象**所引用**的对象进行**销毁**，再加载**新的对象**放到 **Eden 区**。

3）然后将 **Eden 区**中的**剩余对象**移动到**幸存者 0 区**。

4）如果**再次触发垃圾回收**，此时**上次幸存**下来的**放到幸存者 0 区**的，如果**没有回收**，就会**放到幸存者 1 区**。

5）**如果再次经历垃圾回收**，此时会**重新放回幸存者 0 区**，接着**再去幸存者 1 区**。

6）可以设置次数。默认是 15 次。

-XX:MaxTenuringThreshold=<N>**设置对象**晋升老年代的**年龄阈值**。

7）在老年代，相对悠闲。当**老年代内存不足**时，**再次触发 GC**：**Major GC**，进行老年代的内存清理。

8）若老年代执行了 Major GC 之后发现依然无法进行对象的保存，就会产生 **OOM 异常**。

【注意】1）关于垃圾回收：

频繁在新生代收集；**很少**在老年代收集；

几乎不在永久区/元空间收集。

2）s0 和 s1 复制之后有交换，谁空谁是 to。

33、对不同年龄段的对象分配原则如下所示：

1）优先分配到 Eden；

2）大对象直接分配到老年代：尽量避免程序中出现过多的大对象；

3）长期存活的对象分配到老年代：动态对象年龄判断；

4）如果 Survivor 区中**相同年龄的所有对象大小的总和***大于* Survivor 空间的**一半，年龄大于或等于该年龄**的对象**可以直接进入老年代**，无须等到 MaxTenuringThreshold 要求的年龄。

5)**空间分配担保**：-XX:HandlePromotionFailure。

34、空间分配担保

在发生 Minor GC 之前，虚拟机会检查**老年代最大可用的**连续空间*是否大于***新生代所有对象的总空间**，

>如果**大于**，则此次 **Minor GC** 是**安全的**；

> 如果**小于**，则虚拟机会**查看**-XX:HandlePromotionFailure 设置值**是否允许担保失败**。

如果 HandlePromotionFailure=true，那么会继续检查**老年代最大可用**连续空间***是否大于历次晋升***到老年代的对象的**平均大小**，如果大于，则**尝试进行一次 Minor GC**，但这次 Minor GC 依然是**有风险**的；如果**小于**或者 HandlePromotionFailure=**false**，则改为进行一次 **Full GC**。

35、解释 MinorGC、MajorGC、FullGC

JVM 在进行 **GC 时，并非每次**都对上面三个内存(新生代、老年代；方法区)区域**一起回收**的，**大部分时候**回收的**都是指新生代**。

GC 按照回收区域又分为两大种类型：

1）部分收集

1>新生代收集(Minor GC)：只是新生代（Eden\S0,S1）的垃圾收集；

2>老年代收集(Major GC)：只是老年代的垃圾收集；（只有 **CMS GC** 会有**单独收集老年代**的行为）

3>混合收集(Mixed GC)：收集**整个新生代**以及**部分老年代**的垃圾收集。（只有 **G1 GC** 有）

2）整体收集(Full GC)

收集**整个 java 堆和方法区**的垃圾收集。

36、OOM 如何解决?

1）要解决 OOM 异常或 heap space 的异常，首先通过**内存映像分析工具**（如 Eclipse Memory Analyzer）对 dump 出来的堆转储快照进行分析，**确认内存中的对象***是否是必要的*，也就是要**先分清楚**到底是出现了**内存泄漏**还是**内存溢出**。

2）如果**内存泄漏**，可进一步通过工具**查看泄漏对象到 GC Roots 的引用链**。掌握了泄漏对象的类型信息，以及 GC Roots 引用链的信息，可以比较**准确地定位出泄漏代码的位置**。

3）如果**不存在内存泄漏**，即内存中的对象确实都还必须存活着，那应当**检查虚拟机的堆参数**（-Xmx 与-Xms），与机器物理内存对比看**是否还可以调大**，从代码上检查**是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗**。