**知识点**

1. **JVM**
   1. **内存区域的划分**

JVM的内存空间分为五个部分，分别是：   
1. 程序计数器

程序计数器是一块较小的内存空间，记录的是当前线程正在执行的那一条字节码指令的地址，如果当前线程正在执行的是一个本地方法，那么此时程序计数器为空。字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，在多线程的情况下，程序计数器用于记录当前线程执行的位置。线程私有，每条线程都有一个程序计数器，是唯一一个不会出现OutOfMemoryError的内存区域，生命周期随着线程的创建而创建，随着线程的结束而死亡。  
2. Java虚拟机栈

Java虚拟机栈是描述Java方法运行过程的内存模型，会为每一个即将运行的Java方法创建一块叫做“栈帧”的区域，这块区域用于存储该方法在运行过程中所需要的局部变量表、操作数栈、动态链接、方法出口信息等。Java虚拟机栈也是线程私有的，每个线程都有各自的Java虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。  
3. 本地方法栈

本地方法栈和Java虚拟机栈实现的功能类似，只不过本地方法区是本地方法运行的内存模型。  
4. 堆

堆是用来存放对象的内存空间， **几乎所有**的对象都存储在堆中，线程共享 。整个Java虚拟机只有一个堆，所有的线程都访问同一个堆，在虚拟机启动时创建，垃圾回收的主要场所。可以进一步细分为：新生代、老年代。 新生代又可被分为：Eden、From Survior、To Survior。 不同的区域存放具有不同生命周期的对象。  
5. 方法区

是堆的一个逻辑部分，方法区中存放已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等，其中常量存储在运行时常量池中。线程共享，方法区称为老年代，对方法区的内存回收的主要目标是对常量池的回收和对类型的卸载。

6.直接内存

直接内存是除Java虚拟机之外的内存，但也有可能被Java 使用。

总结：两栈(Java虚拟机栈和本地方法栈)两堆(原本的堆和方法区)一计数(计数器)，堆共享其余随线程。

* 1. **对象的创建，内存布局以及访问**

对象的创建包括以下几个步骤，分别是：

1.类加载检查

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.为对象分配内存

根据Java堆中是否规整有两种内存的分配方式，若内存规整则选用指针碰撞，若不规整则采用空闲列表。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。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，称为本地线程分配缓冲(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。

3.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，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。

4.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。这些信息放在对象的对象头之中

5.init

执行new指令之后会接着执行方法，把对象按照程序员的意愿进行初始化。

在虚拟机中对象在内存中存储的布局包括

1.对象头

2.实例数据

3.对齐补充(对象的大小必须是8字节的整数倍)

Java程序需要通过栈上的引用数据来操作堆上的具体对象一般有两种方式

1. 使用句柄

Java堆中将会划分出一块内存来作为句柄池，引用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与类型数据各自的具体地址信息，在对象被移动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。

1. 直接指针

引用中存储的直接就是对象地址，速度更快，节省了一次指针定位的时间开销。

* 1. **垃圾收集器与垃圾收集算法**

内存回收与分配重点关注的是堆内存和方法区内存。

判断对象是否存活

1.引用计数算法

2.可达性分析算法

对象到GC Roots没有引用链，则回收。

方法区的垃圾收集主要回收两部分内容：废弃常量和无用的类，如何判定类无用

该类的实例全部被回收

该类的 ClassLoader 已经被回收

该类的 Class 对象没有任何地方引用

垃圾回收算法

1.标记清除

2.复制算法(将内存分为两个半区，将区A中的存活对象全部复制到B区的连续空间，然后清理A中所有空间)

3.标记整理(将标记后的存活对象进行移动，清除剩余对象)

4.分代收集(新生代复制，老年代标记整理)

* 1. **内存分配与回收策略**

回收策略

JVM采用分代的垃圾回收策略：不同对象的生命周期是不一样的。目前JVM分代主要是分三个年代**：**

1. 新生代

新创建的对象都在新生代进行内存分配，新生代又分为Eden，from survivor，to survivor，大部分对象都被分配在Eden区，Eden满存活的对象复制到fs，fs满复制到ts，ts满复制到老年代。

1. 老年代

老年代是存放一些生命周期较长的对象

1. 持久代

用于存放静态文件，如Java类等

内存分配

1. 优先在Eden中分配，Eden无空间时发起一次minor GC
2. 大对象直接进入老年代，长期存活的对象进入老年代
3. minor GC前若老年代最大可用连续空间小于新生代所有对象总空间且不允许冒险，则发起一次full GC
4. 新生代GC：minorGC，老年代GC：majorGC，堆GC：fullGC
   1. **虚拟机类加载机制**

JVM类加载机制分为五个部分：加载，验证，准备，解析，初始化。

1. 加载

通过一个类的全限定名（包名与类名）来获取定义此类的二进制字节流;可以通过jar、war、网络获取、JSP文件生成等方式。将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构，在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。用户可以通过自定义的类加载器参与此阶段。

1. 连接

连接阶段负责将类的二进制数据合并入JRE中。类的连接大致分三个阶段。验证：验证被加载后的类是否有正确的结构；准备：为类的静态变量在方法区分配内存并赋默认初值；解析：将类的二进制数据中的符号引用换为直接引用。

1. 初始化

类的初始化的主要工作是为静态变量赋程序设定的初值。有且只有五种情况必须对类进行初始化：new、调用静态方法，反射，父类未初始化，main，REF\_getStatic、REF\_putStatic、RE\_invokeStatic句柄。

* 1. **内存模型与线程**

所有的变量都存储在主内存中，每条线程有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。

* 1. **类加载器和双亲委派**

jvm提供三种类加载器

启动类加载器bootstrap classloader

主要加载的是JVM自身需要的类，虚拟机自身的一部分，将 <JAVA\_HOME>/lib路径下的核心类库或-Xbootclasspath参数指定的路径下的jar包加载到内存中。

扩展类加载器extension classloader

负责加载<JAVA\_HOME>/lib/ext目录下或者由系统变量-Djava.ext.dir指定位路径中的类库，开发者可以直接使用标准扩展类加载器。

应用程序类加载器

负责加载系统类路径java -classpath或-D java.class.path 指定路径下的类库，也就是我们经常用到的classpath路径，开发者可以直接使用系统类加载器

双亲委派模型

除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器，而这种父子关系一般通过组合关系来实现。

类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。Java类随着它的类加载器一起具备了一种带有优先级的层次关系。

破坏双亲委派模型

重写loadClass

* 1. **JVM调优**

1. 堆大小设置

-Xmx3550m 最大可用内存

-Xms3550m 初始内存

-Xmn2g  新生代

-Xss128k 每个线程堆栈

1. 回收器选择
2. 辅助信息
3. **Java基础语法特性**

**Volatile**

1. 可见性

对volatile变量的写会立即刷新到主存

对volatile变量的读会强制刷新缓存

1. 没有原子性

AtomicInteger、AtomicLong、AtomicReference

1. 有序性

禁止指令重排序

**Sychronized**

可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性

普通同步方法，锁是当前实例对象

静态同步方法，锁是当前类的class对象

同步方法块，锁是括号里面的对象

Java对象头和monitor是实现synchronized的基础，synchronized用的锁是存在对象头里的，同步代码块是使用monitorenter和monitorexit指令使对象的锁计数+/-1。

**Lock**

Lock vs Sychronized

1. Lock不是Java内置的，synchronized是Java关键字，是内置特性。Lock是一个接口，可以实现同步访问
2. synchronized不需要手动释放锁而lock需要
3. Lock 等待可中断，reentrantlock可以实现公平锁，可绑定多个condition
4. ReentrantLock 和synchronized 都是可重入锁
5. ReentrantLock 添加了类似锁投票、定时锁等候和可中断锁等候的一些特性

**Threadlocal**

当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本

场景：数据库连接、Session管理等

**List**

**Arraylist<T>:**AbstractList<E>,List<E>,RandomAccess,Cloneable, Serializable

动态数组，不是同步，底层为数组，默认初始容量为10，扩容newCapacity = oldCapacity + (oldCapacity >> 1)。

**Linkedlist<T>:**数据结构为链表，增删快查询慢

**Vector<T>:**Arraylist类似，是同步类，开销大。

**Map**

**Hashmap<k,v>:**基于hash表的map接口非同步实现，不保证映射顺序，允许null键值，本质上是Entry数组，数组的每一项是一个链表，线程不安全，通过容量总是2的N次方优化速度。

Put：先计算key 的 hashCode取模决定 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖原有 Entry 的 value，但key不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部。

Rehash：元素大于数组大小乘以负载因子时扩容，扩大一倍并重新计算元素在数组中的位置即rehash。

Fail-fast：使用迭代器的过程中如果有其他线程修改了map则抛出异常。通过modCount(volatile)实现，modCount就是修改次数，对HashMap内容的修改都将增加这个值，在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount，判断modCount跟expectedModCount是否相等。

Jdk1.8：使用node数组，node可能是链表可能是红黑树，同一个格子的key超过8个则链表转为红黑树，查找效率O(logn)

**Linkedhashmap<k,v>:** 通过维护一个额外的双向链表保证了迭代顺序,Entry增加了两个指针 before 和 after，它们分别用于维护双向链接列表

**Treemap<k,v>:**有序的，红黑树结构

Put：以根节点为初始节点进行检索，与当前节点进行比对，若新增节点值较大，则以当前节点的右子节点作为新的当前节点否则以左子节点作为新的当前节点，直到找出合适的叶子节点和新增节点比对，若新增节点大则加为右子节点否则加为左子节点，此时树达到排序状态，再进行调整使其达到平衡。

**Hashtable<k,v>:**线程安全(函数是同步的)，不允许null，直接用hash值不取模，锁定整个map性能较差。

**Concurrenthashmap<k,v>:**把Map分成了N个segment，每一个segment都包含了一个HashEntry数组的hashtable，put和get的时候，都是现根据key.hashCode()算出放到哪个segment中。

segment 通过继承 ReentrantLock 来进行加锁,默认16个segment。

**Set**

**Hashset<T>:**内部基于Hashmap<k,v>,没有重复的元素,k为T对象，v为固定的object对象，通过hashcode()计算hash值，contains通过hash值比较

**Linkedhashset<T>:**内部基于Linkedhashmap<k,v>，继承Hashset，元素顺序可以保证即遍历序和插入序是一致的。

**Treeset<T>:**TreeSet可以确保集合元素处于排序状态,通过equals比较。

**String,Stringbuffer,Stringbuilder**

**String**

String 是不可变的对象, 每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象

**Stringbuffer**

每次结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再改变对象引用，线程安全

**Stringbuilder**

线程不安全

**动态代理与cglib**

**代理模式**

代理类持有被代理类的实例，二者都实现同一个接口

**静态代理**

由程序员创建或特定工具自动生成源代码，也就是在编译时就已经将接口，被代理类，代理类等确定下来。在程序运行之前，代理类的.class文件就已经生成

**动态代理**

代理类在程序运行时创建的代理方式被成为动态代理，本质上不用建很多的代理类，只需要代理接口和真实对象就可以创建动态代理类(静态代理需要代码创建从而类型爆炸)。

**线程池**

corePoolSize：核心池的大小;

maximumPoolSize：线程池最大线程数

Executor是一个顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable类型，用来执行传进去的任务的；

然后ExecutorService接口继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等；

抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法；

然后ThreadPoolExecutor继承了类AbstractExecutorService；

一般不直接使用ThreadPoolExecutor创建线程池而使用

Executors 提供了一系列工厂方法用于创建线程池，返回的线程池都实现了 ExecutorService 接口。

newFixedThreadPool(int nThreads) 固定数目线程线程池

多数针对一些很稳定很固定的正规并发线程

newCachedThreadPool() 缓存型线程池

通常用于执行一些生存期很短的异步型任务

newSingleThreadExecutor() 单线程线程池

可保证顺序地执行各个任务

newScheduledThreadPool(int corePoolSize) 支持定时及周期性的任务执行线程池

Executor 执行 Runnable 任务和Callable任务

1. **框架**

**Spring**

**SpringMVC**

**Mybatis**

**RPC**

**Restful**

**Netty**

**MQ**

**Cache**

**Monitor**

**Zookeeper**

1. **数据结构和算法**

**红黑树：**自平衡的排序二叉树

平衡二叉树：一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

红黑树特点：

1. 每个节点都只能是红色或者黑色
2. 根节点是黑色
3. 每个叶节点（NIL节点，空节点）是黑色的
4. 如果一个结点是红的，则它两个子节点都是黑的
5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点

增加节点：插入新节点总是红色节点，若父节点为黑色则维持性质，若父节点为红色则需要重新着色或旋转。

1. **数据库**

**索引**

**分库分表**

**隔离级别**

1. **设计模式**

**迭代器(游标)**

**观察者(发布-订阅)**

**单例**

**工厂**

1. **linux**