8/31 小米

1、强引用，软引用等介绍一下？

引用：如果Reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。

> （1）强引用（Strong Reference）：在 Java中最常见的就是强引用，**把一个对象赋给一个引用变量，这个引用变量就是一个强引用。**当一个对象被强引用变量引用时，它处于可达状态，它是不可能被垃圾回收机制回收的，即使该对象以后永远都不会被用到 JVM 也不会回收。因此强引用是造成 Java 内存泄漏的主要原因之一。只要强引用还存在，垃圾收集器就永远不会回收掉被引用的对象。
>
> （2）软引用（Soft Reference）：软引用需要用 SoftReference 类来实现，对于只有软引用的对象来说，**当系统内存足够时它不会被回收，当系统内存空间不足时它会被回收。**软引用通常用在对内存敏感的程序中。
>
> （3）弱引用（Weak Reference）：弱引用需要用 WeakReference 类来实现，它比软引用的生存期更短，对于只有弱引用的对象 来说，**只要垃圾回收机制一运行，不管 JVM 的内存空间是否足够，总会回收该对象占用的内存。**
>
> （4）虚引用（Phantom Reference）：虚引用需要 PhantomReference 类来实现，它不能单独使用，必须和引用队列联合使用。 虚引用的主要作用是跟踪对象被垃圾回收的状态。

2、ArrayList，LinkedList 区别？vector扩容机制?

List：一个**有序的**Collection（也称序列），**以元素的添加的顺序作为集合的顺序**，**元素可以重复。**列表通常允许满足e1.equals(e2)的元素对e1和e2，并且列表**允许多个null元素。**

- ArrayList：**允许重复，允许放入null元素；以数组实现；非线程安全；默认第一次插入元素时创建数组的大小为10，超出限制会增加50%的容量。**
- LinkedList：**允许重复；以双向链表实现；非线程安全**；同时实现了list接口和Deque接口。
- 对于**随机访问get和set，ArrayList要优于LinkedList**，因为LinkedList要移动指针 。
- 对于**添加和删除操作add和remove，LinkedList要比ArrayList快**，因为ArrayList要移动数据。

Vector，实现与 ArrayList 类似，但是**使用了 synchronized 进行同步。扩容时每次都令 capacity 为原来的两倍。**

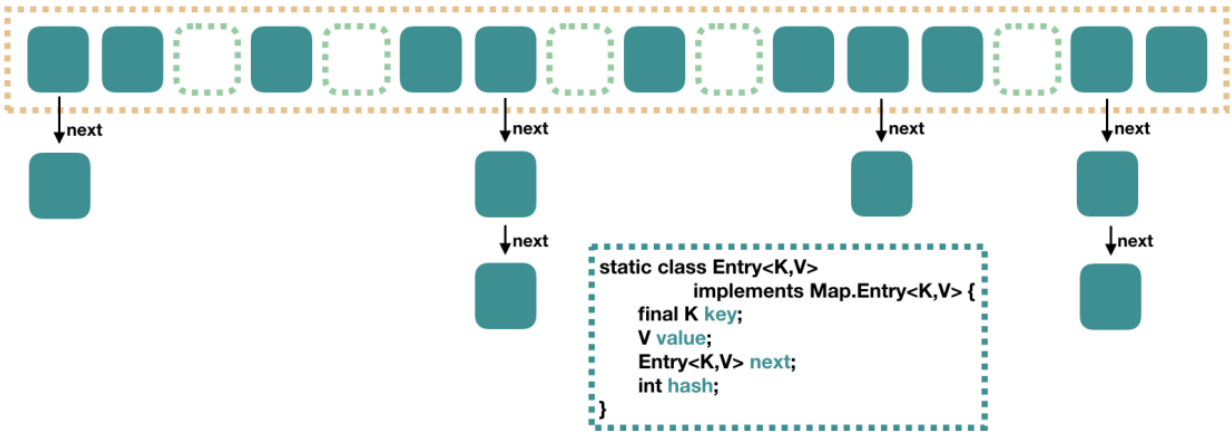
3、Hashmap、HashTable、ConCurrentHashMap区别?

HashMap：**允许键、值为null；基于数组+链表实现；非线程安全。**在HashMap**进行扩容重哈希时导致Entry链形成环。一旦Entry链中有环，会导致在同一个桶中进行插入、查询、删除等操作时陷入死循环。**

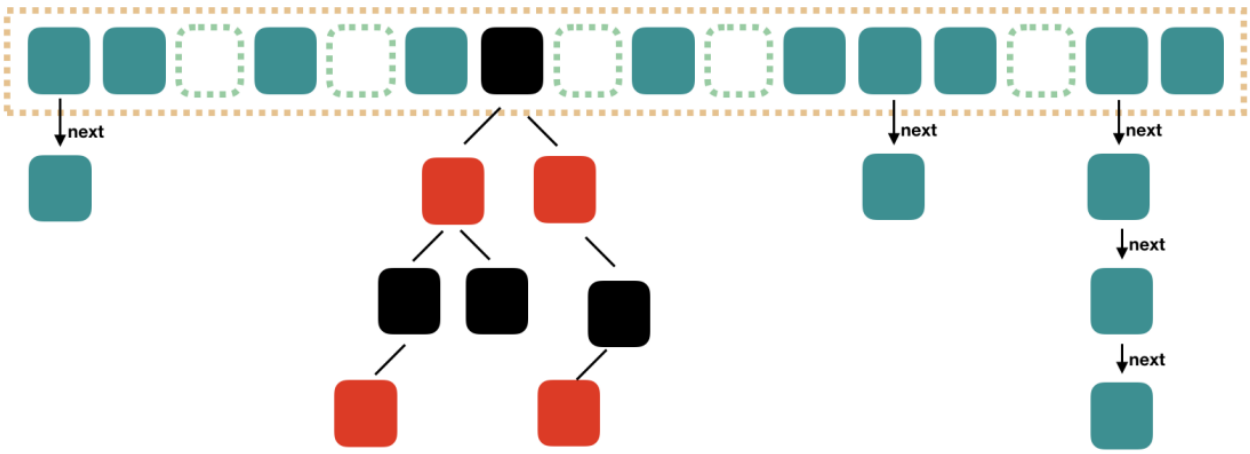
如何实现线程安全：使用 java.util.Collections.synchronizedMap() 方法包装 HashMap object，得到线程安全的 Map，并在此Map上进行操作。

- 内部存储结构：数组+链表+红黑树（JDK8）；
- 默认容量16，默认装载因子0.75；
- key和value对数据类型的要求都是泛型；
- key可以为null，放在table[0]中；
- hashcode：计算键的hashcode作为存储键信息的数组下标用于查找键对象的存储位置。equals：HashMap使用equals()判断当前的键是否与表中存在的键相同。

Java7 HashMap 结构



Java8 HashMap 结构



> - put：将指定的键、值对添加到map里，根据key值计算出hashcode，根据hashcode定位出所在桶。如果桶是一个链表则需要遍历判断里面的hashcode、key值是否和传入key相等，如果相等则进行覆盖，否则插入（头插法）新的Entry。如果桶是空的，说明当前位置没有数据存入，新增一个Entry对象写入当前位置。
> - get：根据指定的key值返回对应的value，根据key计算出hashcode，定位到桶的下标，如果桶是一个链表，依次遍历冲突链表，通过key.equals（k）方法来判断是否是要找的那个entry。如果桶不是链表，根据key是否相等返回值。

注：当hash冲突严重时，**在桶上形成的链表会变的越来越长，这样在查询时的效率就会越来越低。JDK1.8的优化：数组+链表/红黑树。判断当前链表的大小是否大于预设的阈值，大于时就要转换为红黑树。**

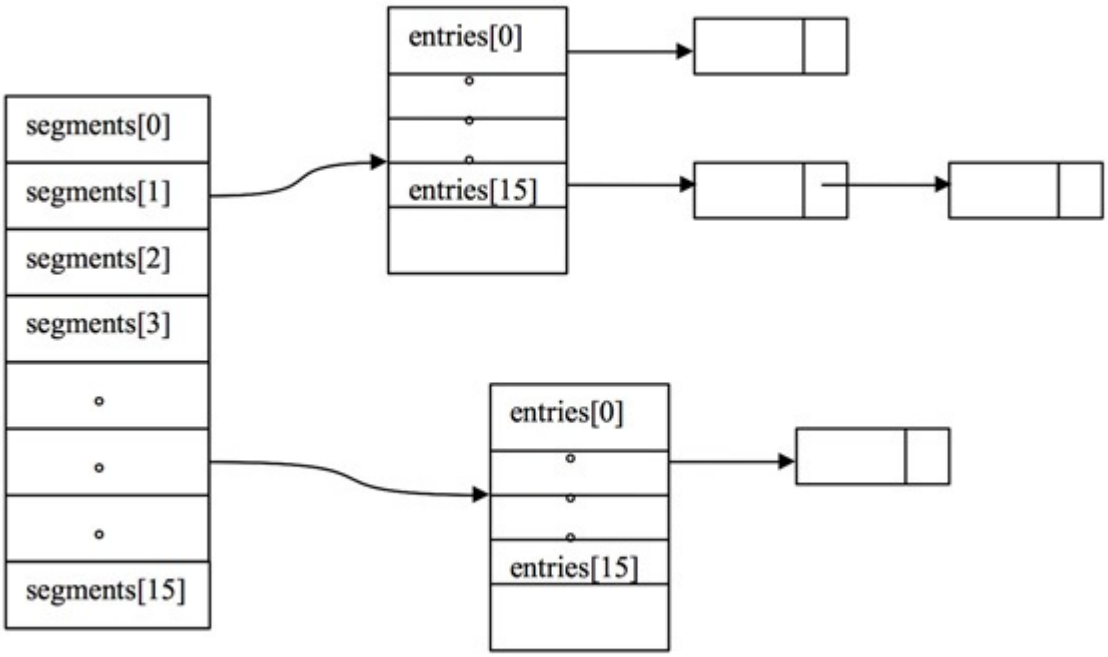
HashTable：**HashTable是线程安全；键、值均不可为null**，其余与HashMap大致相同。

HashTable容器使用Synchronized来保证线程安全，但在线程竞争激烈的情况下，HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，**可能会进入阻塞或轮询状态。**

ConCurrentHashMap：**不允许键、值为null；使用锁分段技术**（容器有多把锁，每一把锁用于锁容器其中一部分数据）。由segment数组结构和HashEntry数组结构组成。**segment是一种可重入锁Reentantlock，在concurrentHashMap里扮演锁的角色。HashEntry则用于存储键值对数据。**

为了能通过按位与的哈希算法来定位 segments 数组的索引，必须保证 segments 数组的长度是 2 的 N 次方 。

一个ConcurrentHashMap里包含一个**segment数组，segment的结构和HashMap类似，是一种数组和链表结构**，一个segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素。



> 定位segment：
>
> concurrentHashMap使用分段锁segment来保护不同段的数据，那么**在插入和获取元素的时候，必须先通过哈希算法定位到segment。**
>
> get：
>
> - 先经过一次再哈希，然后使用这个哈希值通过哈希运算定位到segment，再通过哈希算法定位到元素。
> - get操作的高效之处在于**整个get过程不需要加锁，除非读到的值是空的才会加锁重读。**它的get方法里**将要使用的共享变量都定义成Volatile**，如用于统计当前segment大小的count字段和用于存储值的HashEntry的value。**定义成Volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值。**但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），之所以不会读到过期的值，是根据**java内存模型的happen-before原则，对volatile字段的写入操作先于读操作**，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值。（将key通过Hash之后定位到具体的segment，再通过一次Hash定位到具体的元素上）。
>
> put：
>
> 需要对共享变量进行写入操作，所以**为了线程安全，在操作共享变量时必须得加锁。put方法首先定位到segment，然后再segment里进行插入操作。**
>
> - 判断是否需要对segment里的HashEntry数组进行扩容；
> - 定位添加元素的位置，然后放在HashEntry数组里。
>
> segment的扩容判断比HashMap更恰当：**HashMap是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后就没有新元素插入，这时HashMap就进行了一次无效的扩容。**ConcurrentHashMap**扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再hash后插入到新的数组里，不会对整个容器进行扩容，而只对某个segment进行扩容。**

> put流程：将当前segment中的table通过key的hashcode定位到HashEntry，遍历该HashEntry，如果不为空则判断传入的key和当前遍历的key是否相等，相等则覆盖旧的value。为空则需新建一个HashEntry并加入到segment中，同时会先判断是否需要扩容，最后解除当前segment锁。
>
> 查询遍历链表效率太低，**JDK1.8抛弃了原有的segment分段锁，采用了CAS+Synchronized保证并发安全性。**新版的JDK中对Synchronized优化是很到位的。

总结：读操作（几乎）不需要加锁，而**在写操作时通过锁分段技术只对所操作的段加锁而不影响客户端对其它段的访问。**ConcurrentHashMap本质上是一个segment数组，而一个segment实例又包含若干个桶，每个桶中都包含一条由若干个HashEntry对象链接起来的链表，ConcurrentHashMap的高效并发机制是通过以下三方面来保证的；

- **通过锁分段技术保证并发环境下的写操作；**
- **通过 HashEntry的不变性、Volatile变量的内存可见性和加锁重读机制保证高效、安全的读操作；**
- **通过不加锁和加锁两种方案控制跨段操作的的安全性。**

4、多线程？锁关键字Synchronized？为什么用多线程不用单线程？原子性可见性？

多线程：多线程是为了**同步完成多项任务，提高资源使用效率来提高系统的效率。**线程是在同一时间需要完成多项任务的时候实现的。 对那些可共享的资源来说(比如打印机)，它们在使用期间必须进入锁定状态。所以一个线程可将资源锁定，在完成了它的任务后，再解开(释放)这个锁，使其他线程可以接着使用同样的资源。

一个采用了多线程技术的应用程序可以更好地利用系统资源。其主要优势在于**充分利用了CPU的空闲时间片**，可以用尽可能少的时间来对用户的要求做出响应，使得进程的整体运行效率得到较大提高，同时增强了应用程序的灵活性。更为重要的是，由于**同一进程的所有线程是共享同一内存，所以不需要特殊的数据传送机制**，不需要建立共享存储区或共享文件，从而使得不同任务之间的协调操作与运行、数据的交互、资源的分配等问题更加易于解决。

多线程创建方式：

- 继承Thread类创建线程；当调用 start() 方法启动一个线程时，虚拟机会将该线程放入就绪队列中等待被调度，当一个线程被调度时会执行该线程的 run() 方法。
- 实现Runnable接口创建线程；需要实现 run() 方法，通过 Thread 调用 start() 方法来启动线程。
- 实现Callable接口通过FutureTask包装器来创建Thread线程，与 Runnable 相比，Callable 可以有返回值，返回值通过 FutureTask 进行封装。
- 使用ExecutorService、Callable、Future实现有返回结果的线程 。

为了解决并发编程中存在的**原子性、可见性和有序性**问题，提供了一系列和并发处理相关的关键字，比如`synchronized`、`volatile`、`final`、`concurren包`等。

synchronized：

- 被`synchronized`修饰的代码块及方法，**在同一时间，只能被单个线程访问。**
- `synchronized`，是Java中用于**解决并发情况下数据同步访问**的一个很重要的关键字。当我们想要**保证一个共享资源在同一时间只会被一个线程访问到时，我们可以在代码中使用`synchronized`关键字对类或者对象加锁。**

> synchronized与原子性：原子性是指**一个操作是不可中断的，要全部执行完成，要不就都不执行。** 线程是CPU调度的基本单位。CPU有时间片的概念，会根据不同的调度算法进行线程调度。当一个线程获得时间片之后开始执行，在时间片耗尽之后，就会失去CPU使用权。所以在多线程场景下，由于时间片在线程间轮换，就会发生原子性问题。
>
> 被`synchronized`修饰的代码在同一时间只能被一个线程访问，在锁未释放之前，无法被其他线程访问到 。

> synchronized与可见性：可见性是指**当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。**Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中是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。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。所以，就可能出现线程1改了某个变量的值，但是线程2不可见的情况。
>
> 被`synchronized`修饰的代码，在开始执行时会加锁，执行完成后会进行解锁。而为了保证可见性，有一条规则是这样的：**对一个变量解锁之前，必须先把此变量同步回主存中。**这样解锁后，后续线程就可以访问到被修改后的值。
>
> synchronized与有序性：有序性即**程序执行的顺序按照代码的先后顺序执行。** 除了引入了时间片以外，由于处理器优化和指令重排等，CPU还可能对输入代码进行乱序执行，比如load->add->save 有可能被优化成load->save->add 。这就是可能存在有序性问题。
>
> `synchronized`是无法禁止指令重排和处理器优化的。也就是说，`synchronized`无法避免上述提到的问题。Java程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有操作都是天然有序的。如果在一个线程中观察另一个线程，所有操作都是无序的。 由于`synchronized`修饰的代码，同一时间只能被同一线程访问。那么也就是单线程执行的。所以，可以保证其有序性。

volatile：`volatile`通常被比喻成"轻量级的`synchronized`"，也是Java并发编程中比较重要的一个关键字。和`synchronized`不同，`volatile`是一个变量修饰符，**只能用来修饰变量。无法修饰方法及代码块等。** `volatile`的用法比较简单，只需要**在声明一个可能被多线程同时访问的变量时，使用`volatile`修饰**就可以了。

> volatile与可见性：当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。
>
> Java中的`volatile`关键字提供了一个功能，那就是**被其修饰的变量在被修改后可以立即同步到主内存，被其修饰的变量在每次是用之前都从主内存刷新。**因此，可以使用`volatile`来保证多线程操作时变量的可见性。
>
> volatile与有序性：有序性即程序执行的顺序按照代码的先后顺序执行。
>
> 而`volatile`除了可以保证数据的可见性之外，还有一个强大的功能，那就是他可以**禁止指令重排优化**等。通过禁止指令重排优化，就可以保证代码程序会严格按照代码的先后顺序执行。 volatile是通过**内存屏障**来来禁止指令重排的。
>
> **内存屏障（Memory Barrier）**是一类同步屏障指令，是CPU或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作。
>
> volatile与原子性：原子性是指一个操作是不可中断的，要全部执行完成，要不就都不执行。
>
> **volatile是不能保证原子性的** 。

比较：

- synchronized是一种锁机制，存在阻塞问题和性能问题，而volatile并不是锁，所以不存在阻塞和性能问题。
- volatile借助了内存屏障来帮助其解决可见性和有序性问题，而内存屏障的使用还为其带来了一个禁止指令重排的附件功能，所以在有些场景中是可以避免发生指令重排的问题的。

像我们一开始学习 JAVA 还没学多线程时，所以代码都是在 main 线程中执行，这没问题。但是，假如我们有多个任务，并且其中有耗时的任务。比方说有三个任务，并且执行顺序依次为任务A，任务B，任务C，假设，任务A是一个耗时的 IO 操作（如读取一个文件中的全部信息），那么，在单线程中，当执行到任务A时，由于任务A进行的是 IO 操作，因此线程会挂起并让出 CPU（注意，IO 操作并不会一直占着 CPU），直到DMA（Direct Memory Access）处理器完成任务（这个过程可能会很久，这是操作系统的知识！！操作系统很重要！），再通过中断告诉

CPU完成了，该线程才会再次运行，任务A才可以继续往下执行。因为是单线程，所以任务B必须等到任务A执行完才能执行，任务C必须等到任务B执行完才能执行。如果任务B又是一个耗时的操作，那么任务C要等待的时间就会更久了。

如果使用了多线程，为两个耗时的任务分别开启线程去执行的话，那么由于三个任务是并发执行的，因此，即使任务A和任务B是耗时操作，那也和任务C没关系，因为他们在不同的线程之间了。

5、计网四层七层模型，为什么TCP要四次挥手？

物理层（比特流）：设备之间比特流的传输。

> 在OSI参考模型中，物理层（Physical Layer）是参考模型的最低层，也是OSI模型的第一层。
>
> 物理层的主要功能是：**利用传输介质为数据链路层提供物理连接，实现比特流的透明传输。**
>
> 物理层的作用是**实现相邻计算机节点之间比特流的透明传送，尽可能屏蔽掉具体传输介质和物理设备的差异。**使其上面的数据链路层不必考虑网络的具体传输介质是什么。“透明传送比特流”表示经实际电路传送后的比特流没有发生变化，对传送的比特流来说，这个电路好像是看不见的。

数据链路层（帧）：将上层数据封装成帧，用 MAC 地址访问媒介，错误检测与修正。

> 数据链路层（Data Link Layer）是OSI模型的第二层，**负责建立和管理节点间的链路。**该层的主要功能是：**通过各种控制协议，将有差错的物理信道变为无差错的、能可靠传输数据帧的数据链路。**

> 在计算机网络中由于各种干扰的存在，物理链路是不可靠的。因此，这一层的主要功能是在物理层提供的比特流的基础上，通过差错控制、流量控制方法，使有差错的物理线路变为无差错的数据链路，即提供可靠的通过物理介质传输数据的方法。
>
> 该层通常又被分为介质访问控制（MAC）和逻辑链路控制（LLC）两个子层。
>
> MAC子层的主要任务是解决共享型网络中多用户对信道竞争的问题，完成网络介质的访问控制；
>
> LLC子层的主要任务是建立和维护网络连接，执行差错校验、流量控制和链路控制。
>
> 数据链路层的具体工作是接收来自物理层的位流形式的数据，并封装成帧，传送到上一层；同样，也将来自上层的数据帧，拆装为位流形式的数据转发到物理层；并且，还负责处理接收端发回的确认帧的信息，以便提供可靠的数据传输。

网络层（包）：提供逻辑地址（IP）、选路，数据从源端到目的端的传输。

> 网络层（Network Layer）是OSI模型的第三层，它是OSI参考模型中最复杂的一层，也是通信子网的最高一层。它**在下两层的基础上向资源子网提供服务。**其主要任务是：**通过路由选择算法，为报文或分组通过通信子网选择最适当的路径。该层控制数据链路层与传输层之间的信息转发，建立、维持和终止网络的连接。**具体地说，数据链路层的数据在这一层被转换为数据包，然后通过路径选择、分段组合、顺序、进/出路由等控制，将信息从一个网络设备传送到另一个网络设备。
>
> 一般地，数据链路层是解决同一网络内节点之间的通信，而网络层主要解决不同子网间的通信。例如在广域网之间通信时，必然会遇到路由（即两节点间可能有多条路径）选择问题。
>
> 在实现网络层功能时，需要解决的主要问题如下：
>
> 寻址：数据链路层中使用的物理地址（如MAC地址）仅解决网络内部的寻址问题。在不同子网之间通信时，为了识别和找到网络中的设备，每一子网中的设备都会被分配一个唯一的地址。由于各子网使用的物理技术可能不同，因此这个地址应当是逻辑地址（如IP地址）。
>
> 交换：规定不同的信息交换方式。常见的交换技术有：线路交换技术和存储转发技术，后者又包括报文交换技术和分组交换技术。
>
> 路由算法：当源节点和目的节点之间存在多条路径时，本层可以根据路由算法，通过网络为数据分组选择最佳路径，并将信息从最合适的路径由发送端传送到接收端。
>
> 连接服务：与数据链路层流量控制不同的是，前者控制的是网络相邻节点间的流量，后者控制的是从源节点到目的节点间的流量。其目的在于防止阻塞，并进行差错检测。

传输层（段）：实现网络不同主机上用户进程之间的数据通信，可靠与不可靠的传输，传输层的错误检测，流量控制等。

> **OSI下3层的主要任务是数据通信，上3层的任务是数据处理。**而传输层（Transport Layer）是OSI模型的第4层。因此该层**是通信子网和资源子网的接口和桥梁**，起到承上启下的作用。
>
> 该层的主要任务是：**向用户提供可靠的端到端的差错和流量控制，保证报文的正确传输。**传输层的作用是向高层屏蔽下层数据通信的细节，即向用户透明地传送报文。该层常见的协议：TCP/IP中的TCP协议、Novell网络中的SPX协议和微软的NetBIOS/NetBEUI协议。
>
> 传输层提供会话层和网络层之间的传输服务，这种服务从会话层获得数据，并在必要时，对数据进行分割。然后，传输层将数据传递到网络层，并确保数据能正确无误地传送到网络层。因此，传输层负责提供两节点之间数据的可靠传送，当两节点的联系确定之后，传输层则负责监督工作。综上，传输层的主要功能如下：
>
> 传输连接管理：提供建立、维护和拆除传输连接的功能。传输层在网络层的基础上为高层提供“面向连接”和“面向无接连”的两种服务。

> 处理传输差错：提供可靠的“面向连接”和不太可靠的“面向无连接”的数据传输服务、差错控制和流量控制。在提供“面向连接”服务时，通过这一层传输的数据将由目标设备确认，如果在指定的时间内未收到确认信息，数据将被重发。
>
> 监控服务质量。

会话层（数据）：提供包括访问验证和会话管理在内的建立和维护应用之间通信的机 制，如服务器验证用户登录便是由会话层完成的。

> 会话层（Session Layer）是OSI模型的第5层，是用户应用程序和网络之间的接口，主要任务是：**向两个实体的表示层提供建立和使用连接的方法。**将不同实体之间的表示层的连接称为会话。因此会话层的任务就是组织和协调两个会话进程之间的通信，并对数据交换进行管理。
>
> 用户可以按照半双工、单工和全双工的方式建立会话。当建立会话时，用户必须提供他们想要连接的远程地址。而这些地址与MAC（介质访问控制子层）地址或网络层的逻辑地址不同，它们是为用户专门设计的，更便于用户记忆。域名（DN）就是一种网络上使用的远程地址例如：www.3721.com就是一个域名。会话层的具体功能如下：
>
> 会话管理：允许用户在两个实体设备之间建立、维持和终止会话，并支持它们之间的数据交换。例如提供单方向会话或双向同时会话，并管理会话中的发送顺序，以及会话所占用时间的长短。
>
> 会话流量控制：提供会话流量控制和交叉会话功能。
>
> 寻址：使用远程地址建立会话连接。
>
> 出错控制：从逻辑上讲会话层主要负责数据交换的建立、保持和终止，但实际的工作却是接收来自传输层的数据，并负责纠正错误。会话控制和远程过程调用均属于这一层的功能。但应注意，此层检查的错误不是通信介质的错误，而是磁盘空间、打印机缺纸等类型的高级错误。

表示层（数据）：主要解决拥护信息的语法表示问题，如加密解密。

> 表示层（Presentation Layer）是OSI模型的第六层，它对来自应用层的命令和数据进行解释，对各种语法赋予相应的含义，并按照一定的格式传送给会话层。其主要功能是“**处理用户信息的表示问题，如编码、数据格式转换和加密解密”等。**表示层的具体功能如下：
>
> 数据格式处理：协商和建立数据交换的格式，解决各应用程序之间在数据格式表示上的差异。
>
> 数据的编码：处理字符集和数字的转换。例如由于用户程序中的数据类型（整型或实型、有符号或无符号等）、用户标识等都可以有不同的表示方式，因此，在设备之间需要具有在不同字符集或格式之间转换的功能。
>
> 压缩和解压缩：为了减少数据的传输量，这一层还负责数据的压缩与恢复。
>
> 数据的加密和解密：可以提高网络的安全性。

应用层（数据）：确定进程之间通信的性质以满足用户需要以及提供网络与用户应用。

> 应用层（Application Layer）是OSI参考模型的最高层，它是计算机用户，以及各种应用程序和网络之间的接口，其功能是**直接向用户提供服务，完成用户希望在网络上完成的各种工作。**它在其他6层工作的基础上，负责完成网络中应用程序与网络操作系统之间的联系，建立与结束使用者之间的联系，并完成网络用户提出的各种网络服务及应用所需的监督、管理和服务等各种协议。此外，该层还负责协调各个应用程序间的工作。
>
> 应用层为用户提供的服务和协议有：文件服务、目录服务、文件传输服务（FTP）、远程登录服务（Telnet）、电子邮件服务（E-mail）、打印服务、安全服务、网络管理服务、数据库服务等。上述的各种网络服务由该层的不同应用协议和程序完成，不同的网络操作系统之间在功能、界面、实现技术、对硬件的支持、安全可靠性以及具有的各种应用程序接口等各个方面的差异是很大的。应用层的主要功能如下：

> 用户接口：应用层是用户与网络，以及应用程序与网络间的直接接口，使得用户能够与网络进行交互式联系。
>
> 实现各种服务：该层具有的各种应用程序可以完成和实现用户请求的各种服务。

四次挥手：

由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。**收到一个 FIN只意味着这一方向上没有数据流动**，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。四次挥手过程：

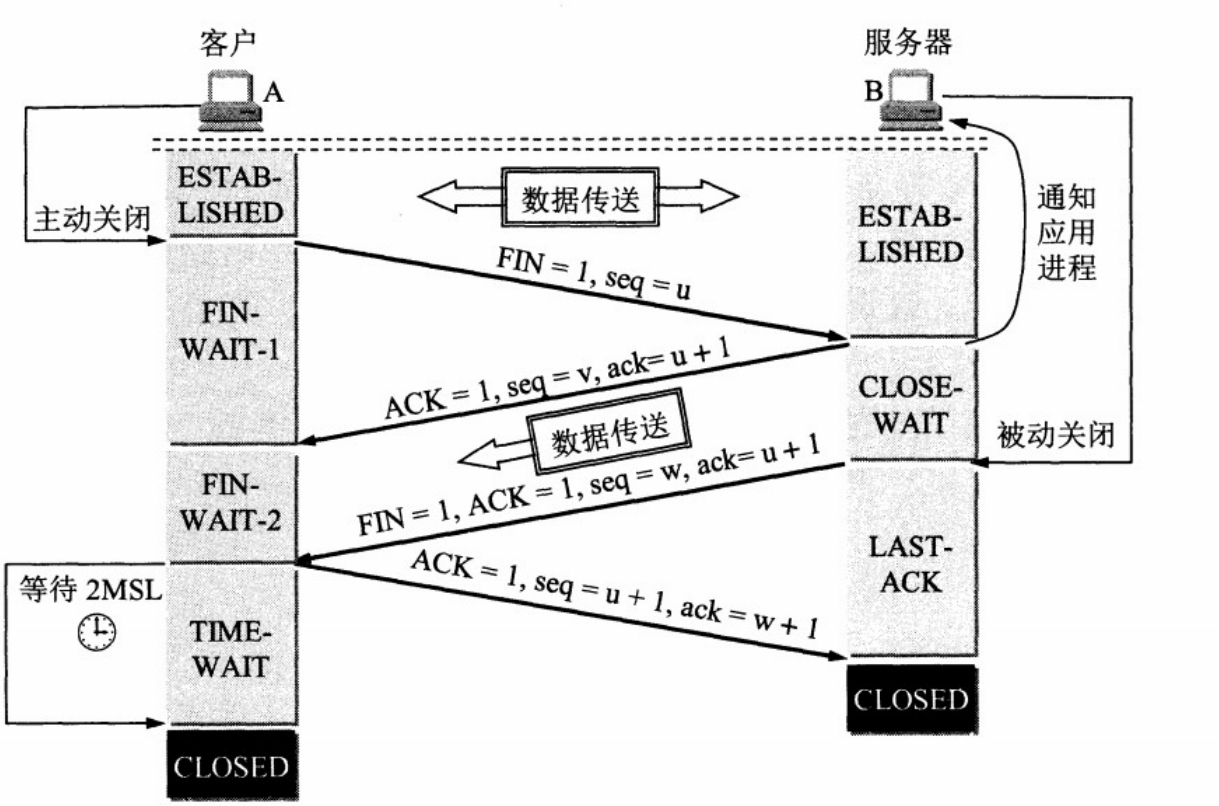


图 5-29　TCP 连接释放的过程

四次挥手原因：这是因为服务端的LISTEN状态下的SOCKET当收到SYN报文的建连请求后，它可以把ACK和SYN（ACK起应答作用，而SYN起同步作用）放在一个报文里来发送。但**关闭连接时，当收到对方的FIN报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭SOCKET**,也即你可能还需要发送一些数据给对方之后，再发送FIN报文给对方来表示你同意现在可以关闭连接了，所以它这里的ACK报文和FIN报文多数情况下都是分开发送的。

6、TCP、UDP区别？

- **TCP提供面向对象的连接，通信前要建立三次握手机制的连接，UDP提供无连接的传输，传输前不用建立连接**
- **TCP提供可靠的，有序的，不丢失的传输，UDP提供不可靠的传输**
- **TCP提供面向字节流的传输，它能将信息分割成组，并在接收端将其充足，UDP提供面向数据报的传输，没有分组开销**
- **TCP提供拥塞控制，流量控制机制，UDP没有**
- **每一条TCP连接只能是点到点的；UDP支持一对一，一对多，多对一和多对多的交互通信**

7、DNS 、ARP？

DNS解析过程：

- **浏览器先检查自身缓存中有没有被解析过的这个域名对应的ip地址**，如果有，解析结束。同时域名被缓存的时间也可通过TTL属性来设置。
- 如果浏览器缓存中没有（专业点叫还没命中），**浏览器会检查操作系统缓存中有没有对应的已解析过的结果。**而操作系统也有一个域名解析的过程。在windows中可通过c盘里一个叫hosts的文件来设置，如果你在这里指定了一个域名对应的ip地址，那浏览器会首先使用这个ip地址。
- 如果至此还没有命中域名，才会**真正的请求本地域名服务器（LDNS）来解析这个域名**，这台服务器一般在你的城市的某个角落，距离你不会很远，并且这台服务器的性能都很好，一般都会缓存域名解析结果，大约80%的域名解析到这里就完成了。
- 如果LDNS仍然没有命中，就**直接跳到Root Server 域名服务器请求解析**
- 根域名服务器返回给LDNS一个所查询域的主域名服务器（gTLD Server，国际顶尖域名服务器，如.com .cn .org等）地址
- 此时LDNS再发送请求给上一步返回的gTLD
- 接受请求的gTLD查找并返回这个域名对应的Name Server的地址，这个Name Server就是网站注册的域名服务器
- Name Server根据映射关系表找到目标ip，返回给LDNS
- LDNS缓存这个域名和对应的ip
- LDNS把解析的结果返回给用户，用户根据TTL值缓存到本地系统缓存中，域名解析过程至此结束

地址解析协议ARP：

ARP 实现由 IP 地址得到 MAC 地址。

每个主机都有一个 ARP 高速缓存，里面有**本局域网上的各主机和路由器的 IP 地址到 MAC 地址的映射表。**

如果主机 A 知道主机 B 的 IP 地址，但是 ARP 高速缓存中没有该 IP 地址到 MAC 地址的映射，此时**主机 A 通过广播的方式发送 ARP 请求分组**，主机 B 收到该请求后会发送 ARP 响应分组给主机 A 告知其 MAC 地址，随后主机 A 向其高速缓存中写入主机 B 的 IP 地址到 MAC 地址的映射。

ARP 高效运行的关键是**每个主机上都有一个 ARP 的高速缓存。**

8、设计模式单例?

单例模式(Singleton Pattern)：单例模式**确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例**，这个类称为单例类，它**提供全局访问的方法。**

优点：**节约资源，节省时间。**

- 由于频繁使用的对象，可以省略创建对象所花费的时间，这对于那些重量级的对象而言，是很重要的。
- 因为不需要频繁创建对象，我们的GC压力也减轻了。

缺点：简单的单例模式设计开发都比较简单，但是**复杂的单例模式需要考虑线程安全等并发问题**，引入了部分复杂度。**职责过重**（既充当工厂角色，提供工厂方法，同时又充当了产品角色。包含一些业务方法）。

三要素：

- 一个静态类变量；
- 一个私有构造方法；
- 一个全局静态的类方法。

类型：懒汉式（线程不安全）、饿汉式（天生线程安全）。

饿汉式：初始化类时，直接就创建唯一实例；

（1）

```
public class Singleton {
    private static Singleton instance = new Singleton();////私有静态变量
    private Singleton() {}//私有构造函数
    public static Singleton getInstance() {//公有静态函数
        return instance;
    }
}
```

（2）枚举方式：（线程安全，**可防止反射构建**），在枚举类对象被反序列化的时候，保证反序列的返回结果是同一对象。

```
public enum Singleton {
    INSTANCE;
}
```

懒汉式：

（1）双重校验锁：

```
//第一次判断 instance是否为空是为了确保返回的实例不为空
//第二次判断 instance是否为空是为了防止创建多余的实例
public class Singleton {
    private volatile static Singleton instance = null;
    private Singleton() {}
    public static Singleton getInstance() {
        if(instance == null) {
            synchronized(Singleton.class){
                if(instance == null) {//线程B
                    instance = new Singleton();//并非是一个原子操作     线程A
                    /**
                    * 会被编译器编译成如下JVM指令：
                    * memory = allocate(); - 1、分配对象的内存空间
                    * ctorInstance(memory); - 2、初始化对象
                    * instance = memory; - 3、设置instance指向刚分配的内存地址
                    */
                }
            }
        }
        return instance;
    }
}
```

synchronized同步块里面能够保证只创建一个对象。但是**通过在synchronized的外面增加一层判断，就可以在对象一经创建以后，不再进入synchronized同步块。这种方案不仅减小了锁的粒度，保证了线程安全，性能方面也得到了大幅提升。** 进入Synchronized 临界区以后，还要再做一次判空。因为当两个线程同时访问的时候，线程A构建完对象，线程B也已经通过了最初的判空验证，不做第二次判空的话，线程B还是会再次构建instance对象。

Volatile：

- 可见性；

- 防止指令重排序：**防止new Singleton时指令重排序导致其他线程获取到未初始化完的对象。**指令顺序并非一成不变，有可能会经过JVM和CPU的优化，指令重排成下面的顺序：1，3，2。在3执行完毕，2未执行之前，被线程2抢占了，这时instance已经是非null了（但却没有初始化），所以线程2会直接返回instance，然后使用，报错。

（2）静态内部类：线程安全，懒加载

```
//INSTANCE对象初始化的时机并不是在单例类Singleton被加载的时候，而是在调用getInstance方法，使得静态内部类SingletonHolder被加载的时候。因此这种实现方式是利用classloader的加载机制来实现懒加载，并保证构建单例的线程安全。
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton() {}
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

9、项目 Mybatis和Mysql有什么关系？MVC?

Mybatis是ORM的一种实现框架，对JDBC的一种封装。ORM模型就是数据库的表和简单Java对象（Plain Ordinary Java Object，简称 POJO）的映射关系模型，它主要解决的就**是数据库数据和POJO对象的相互映射。**我们通过这层映射关系就可以简单迅速地把数据库表的数据转化为POJO，以便程序员更加容易理解和应用Java程序。 MyBatis灵活，可以**动态生成映射关系的框架。**

MVC是三个单词的首字母缩写，它们是Model（模型）、View（视图）和Controller（控制）。

> 1）最上面的一层，是直接面向最终用户的"视图层"（View）。它是提供给用户的操作界面，是程序的外壳。
>
> 2）最底下的一层，是核心的"数据层"（Model），也就是程序需要操作的数据或信息。
>
> 3）中间的一层，就是"控制层"（Controller），它负责根据用户从"视图层"输入的指令，选取"数据层"中的数据，然后对其进行相应的操作，产生最终结果。

10、单链表反转?

```
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}
//递归
public ListNode ReverseList(ListNode head) {
    if(head == null || head.next == null) {
        return head;
    }
    ListNode newHead = ReverseList(head.next);

    head.next.next = head;
```

```
        head.next = null;
        return newHead;
    }
    //非递归
    public ListNode ReverseList(ListNode head) {
        ListNode newHead = null;//已反转的链表
        while(head != null) {
            ListNode next = head.next;
            head.next = newHead;
            newHead = head;
            head = next;
        }
        return newHead;
    }
```

11、股票?

```
//一次股票交易包含买入和卖出，只进行一次交易，求最大收益。
public int maxProfit(int[] prices) {
    //只要记录前面的最小价格，将这个最小价格作为买入价格，然后将当前的价格作为售出价格，查看当前收益是不是最大收益。
    if(prices == null || prices.length == 0) {
        return 0;
    }
    int max = 0, min = prices[0];
    for(int i = 1; i < prices.length; i++) {
        if(prices[i] < min) {
            min = prices[i];
        } else {
            max = Integer.max(max, prices[i] - min);
        }
    }
    return max;
}
//可以进行多次交易，多次交易之间不能交叉进行。
public int maxProfit(int[] prices) {
    //对于 [a, b, c, d]，如果有 a <= b <= c <= d ，那么最大收益为 d - a。而 d - a = (d - c) + (c - b) + (b - a) ，因此当访问到一个 prices[i] 且 prices[i] - prices[i-1] > 0，那么就把 prices[i] - prices[i-1] 添加到收益中。
    int profit = 0;
    for(int i = 1; i < prices.length; i++) {
        if(prices[i] > prices[i - 1]) {
            profit += prices[i] - prices[i - 1];
        }
    }
    return profit;
}
```

12、synchronized和reentrantlock的区别，当都被interrupt时，有什么区别？

Java 提供了两种锁机制来控制多个线程对共享资源的互斥访问， synchronized和ReentrantLock。

共同点：

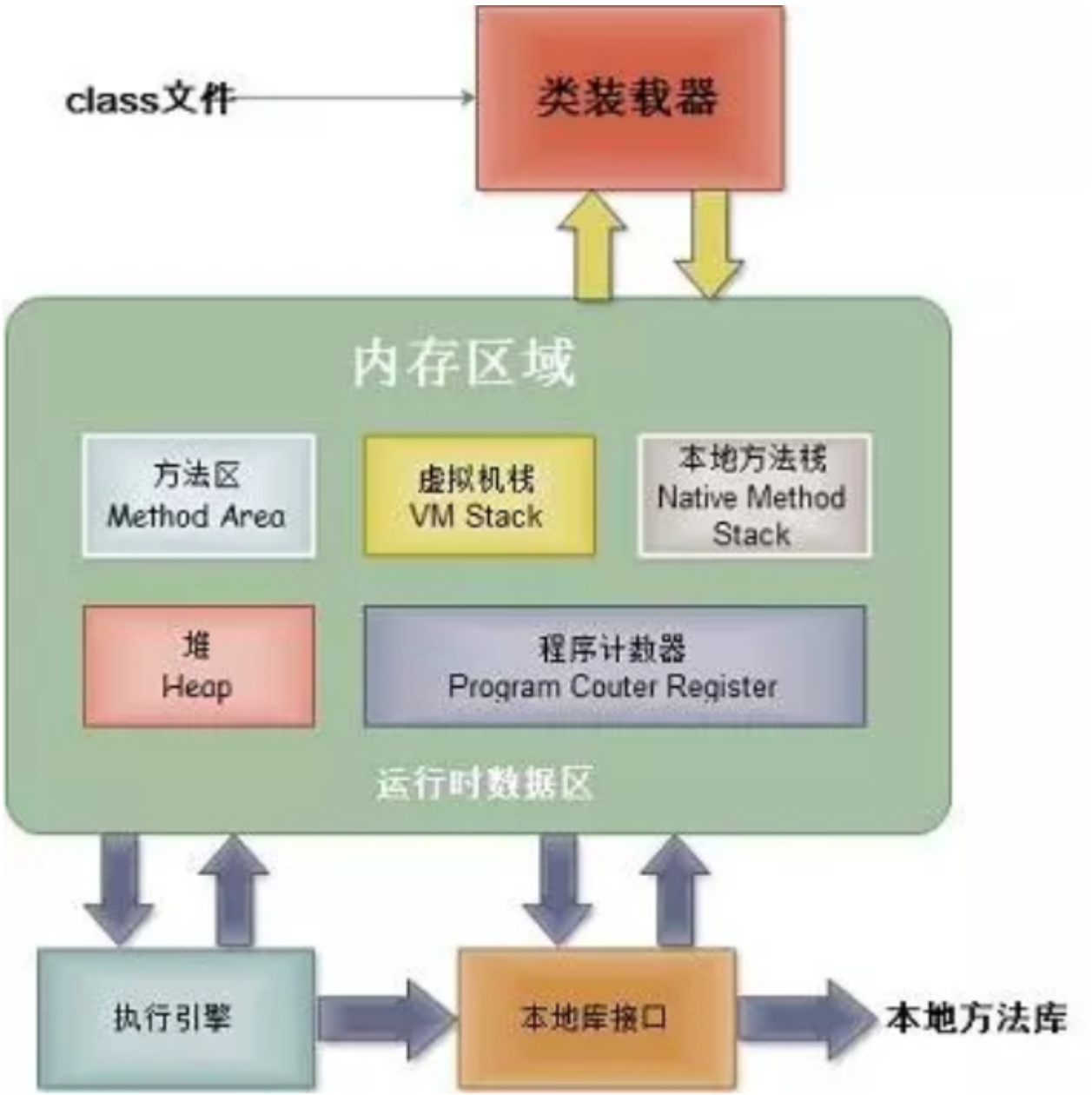
> 都是用来协调多线程对共享对象、变量的访问
>
> 都是可重入锁，同一线程可以多次获得同一个锁
>
> 都保证了可见性和互斥性

不同点：

> 锁的实现：**synchronized 是 JVM 实现的，而 ReentrantLock 是 JDK 实现的。**
>
> 性能：新版本 Java 对 synchronized 进行了很多优化，例如自旋锁等，synchronized 与 ReentrantLock 大致相同。
>
> 等待可中断：当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情。**ReentrantLock 可中断，而 synchronized 不行。**
>
> 公平锁：公平锁是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁。
>
> synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但是也可以是公平的。
>
> 锁绑定多个条件：一个 ReentrantLock 可以同时绑定多个 Condition 对象。

13、jvm内存模型，类的加载过程？

Java内存区域划分：



- 程序计数器：可以看做是**当前线程所执行的字节码的行号指示器。**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。（线程私有的内存区域）
- Java虚拟机栈：描述**Java方法执行的内存模型，每个方法执行的同时会创建一个栈帧**，栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。（线程私有的）
- 本地方法栈：本地方法栈与虚拟机栈的区别：虚拟机栈为虚拟机执行Java方法服务（也就是字节码），而**本地方法栈为虚拟机使用到的Native方法服务。**
- Java堆：Java**堆是被所有的线程共享的一块内存区域**，在虚拟机启动时创建。Java堆的唯一目的就是**存放对象实例**，几乎所有的对象实例都在这里分配内存。
- 方法区：**被所有的线程共享的一块内存区域。**它**用于存储已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数据。**

类加载：类的加载指的是**将类的.class文件中的二进制数据读入到内存中**，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。

类加载分为五个部分：加载，验证，准备，解析，初始化。

> - 加载：会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入口。
> - 验证：为了确保 Class 文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。
> - 准备：正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使用的内存空间。
> - 解析：指虚拟机将常量池中的符号引用替换为直接引用的过程。
> - 初始化：是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载 器以外，其它操作都由 JVM 主导。到了初始阶段，才开始真正执行类中定义的 Java 程序代码。

双亲委派模型：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是**把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中**，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。 好处：**使得 Java 类随着它的类加载器一起具有一种带有优先级的层次关系**，从而使得基础类得到统一。

14、synchronized，说一下它的锁的不同粒度？synchronized修饰普通方法和静态方法有什么区别？

重量级锁：

> Synchronized是通过对象内部的一个叫做监视器锁（monitor）来实现的。但是**监视器锁本质又是依赖于底层的操作系统的Mutex Lock来实现**的。而**操作系统实现线程之间的切换这就需要从用户态转换到核心态**，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么Synchronized效率低的原因。因此，这种**依赖于操作系统Mutex Lock所实现的锁我们称之为“重量级锁”**。JDK中对Synchronized做的种种优化，其核心都是为了减少这种重量级锁的使用。JDK1.6以后，为了减少获得锁和释放锁所带来的性能消耗，提高性能，引入了“轻量级锁”和“偏向锁”。

轻量级锁：

> “轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是**在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。**在解释轻量级锁的执行过程之前，先明白一点，**轻量级锁所适应的场景是线程交替执行同步块的情况**，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

偏向锁：

> 引入偏向锁是为了**在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径**，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）。上面说过，轻量级锁是为了在线程交替执行同步块时提高性能，而**偏向锁则是在只有一个线程执行同步块时进一步提高性能。**

其他优化：

> - **适应性自旋（Adaptive Spinning）：**从轻量级锁获取的流程中我们知道，当线程在获取轻量级锁的过程中执行CAS操作失败时，是要通过自旋来获取重量级锁的。问题在于，自旋是需要消耗CPU的，如果一直获取不到锁的话，那该线程就一直处在自旋状态，白白浪费CPU资源。解决这个问题最简单的办法就是指定自旋的次数，例如让其循环10次，如果还没获取到锁就进入阻塞状态。但是JDK采用了更聪明的方式——适应性自旋，简单来说就是线程如果自旋成功了，则下次自旋的次数会更多，如果自旋失败了，则自旋的次数就会减少。
> - **锁粗化（Lock Coarsening）：**锁粗化的概念应该比较好理解，就是将多次连接在一起的加锁、解锁操作合并为一次，将多个连续的锁扩展成一个范围更大的锁。
> - **锁消除（Lock Elimination）：**锁消除即删除不必要的加锁操作。根据代码逃逸技术，如果判断到一段代码中，堆上的数据不会逃逸出当前线程，那么可以认为这段代码是线程安全的，不必要加锁。

总结：

锁	优点	缺点	适应场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。	如果线程间存在锁竞争，会带来额外的锁撤销的消耗。	适用于只有一个线程访问同步块场景。
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度。	如果始终得不到锁竞争的线程使用自旋会消耗CPU。	追求响应时间。同步块执行速度非常快。
重量级锁	线程竞争不使用自旋，不会消耗CPU。	线程阻塞，响应时间缓慢。	追求吞吐量 同步块执行速度较长。 。

在Java中，synchronized是用来表示同步的，我们可以synchronized来修饰一个方法。也可以synchronized来修饰方法里面的一个语句块。

在static方法前加synchronizedstatic：**静态方法属于类方法，它属于这个类，获取到的锁，是属于类的锁。**

在普通方法前加synchronizedstatic：**非static方法获取到的锁，是属于当前对象的锁。**

类锁和对象锁不同，他们之间不会产生互斥。

15、 不同排序算法的稳定性和空间复杂度？

排序类别	排序方法	时间复杂度			空间复杂度	稳定性	复杂性
		平均情况	最坏情况	最好情况			
交换排序	冒泡排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
	快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定	较复杂
插入排序	直接插入排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
	希尔排序	$O(nlog_2n)$	$O(n^2)$		O(1)	不稳定	较复杂
选择排序	简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定	简单
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	O(1)	不稳定	较复杂
归并排序	归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	O(n)	稳定	较复杂
基数排序	基数排序	O(d * n)	O(d * n)	O(d * n)	O(n)	稳定	较复杂

16、手写归并排序？快速排序？堆排序？

```
//归并排序：将数组分成两部分，分别进行排序，然后归并起来。
public static void sort(int[] num, int low, int high) {
    int mid = low + (high - low) / 2;
    if(low < high) {
        sort(num, low, mid);
        sort(num, mid + 1, high);
        merge(num, low, mid, high);
    }
}
```

```
public static void merge(int[] num, int low, int mid, int high) {
    int[] temp = new int[high - low + 1];
    int i = low;//左指针
    int j = mid + 1;//右指针
    int k = 0;
    while(i <= mid && j <= high) {//将较小的数先移入新数组中
        if(num[i] <= num[j]) {
            temp[k++] = num[i++];
        } else {
            temp[k++] = num[j++];
        }
    }
    //把左边剩余的数移入数组
    while(i <= mid) {
        temp[k++] = num[i++];
    }
    //把右边剩余的数移入数组
    while(j <= high) {
        temp[k++] = num[j++];
    }
    //将新数组中的值拷入原数组
    k = 0;
    while(low <= high) {
        num[low++] = temp[k++];
    }
}
//快速排序：通过一个切分元素将数组分为两个子数组。左子数组小于等于切分元素，右子数组大于等于切分元素，将这两个子数组排序也就将整个数组排序了。
public class quick {
    public static void sort(int[] num, int left, int right) {
        if(left > right) {
            return;
        } else {
            int base  = divide(num, left, right);
            sort(num, left, base - 1);
            sort(num, base + 1, right);
        }
    }
    public static int divide(int[] num, int left, int right) {
        int base = num[left];
        while(left < right) {
            //从数组右端开始，向左遍历，直到找到小于base的数
            while(left < right && num[right] > base) {
                right--;
            }
            //找到了比base小的元素，将这个元素放到最左边的位置
            num[left] = num[right];
            //从数组左端开始，向右遍历，直到找到大于base的数
            while(left < right && num[left] < base) {
                left++;
            }
            //找到了比base大的元素，将这个元素放到最右边的位置

            num[right] = num[left];
```

```
        }
        //最后将base放到left位置，此时，left左边的值都比它小，右边的值都比它大
        num[left] = base;
        return left;
    }
}
//堆排序
public class HeadSort {
    public static void buildHeap(int[] arr, int parent, int length) {
        for(int child = 2 * parent + 1; child < length; child = 2 * child + 1) {
            if(child + 1 < length && arr[child] < arr[child + 1]) {// 让child先指向子节点中最大的
节点
                child++;
            }
            if(arr[parent] < arr[child]) {// 如果发现子节点更大，则进行值的交换
                swap(arr, parent, child);
                // 下面就是非常关键的一步了
                // 如果子节点更换了，那么，以子节点为根的子树会不会受到影响呢?
                // 所以，循环对子节点所在的树继续进行判断
                parent = child;
            } else {
                break;
            }
        }
    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void sort(int[] arr){
        for(int parent = arr.length / 2 - 1; parent >= 0; parent--) {//构建最大堆
            buildHeap(arr, parent, arr.length);
        }
        for(int i = arr.length - 1; i > 0; i--) {//把最大的元素扔在最后面
            swap(arr, 0, i);
            buildHeap(arr, 0, i);//将arr中前i - 1个记录重新调整为最大堆
        }
    }
}
```

17、删除链表中重复的结点?

```
public ListNode deleteDuplication(ListNode pHead) {
    ListNode dummyHead = new ListNode(-1);
    dummyHead.next = pHead;
    ListNode pre = dummyHead;
    ListNode cur = dummyHead.next;
    while(cur != null) {

        if(cur.next != null && cur.val == cur.next.val) {
```

```
            while(cur.next != null && cur.val == cur.next.val) {//找出最后一个相同结点
                cur = cur.next;
            }
            pre.next = cur.next;
            cur = cur.next;
        } else {
            pre = pre.next;
            cur = cur.next;
        }
    }
    return dummyHead.next;
}
```

18、合并两个有序的链表?

```
public ListNode Merge(ListNode list1,ListNode list2) {
    ListNode dummyHead = new ListNode(-1);
    ListNode cur = dummyHead;
    while(list1 != null && list2 != null) {
        if(list1.val <= list2.val) {
            cur.next = list1;
            list1 = list1.next;
            cur = cur.next;
        } else if(list1.val > list2.val) {
            cur.next = list2;
            list2 = list2.next;
            cur = cur.next;
        }
    }
    if(list1 != null) {
        cur.next = list1;
    }
    if(list2 != null) {
        cur.next = list2;
    }
    return dummyHead.next;
}
```

19、最长公共子序列：对于两个子序列 S1 和 S2，找出它们最长的公共子序列。

定义一个二维数组 **dp 用来存储最长公共子序列的长度，其中 `dp[ i ][ j ]` 表示 S1 的前 i 个字符与 S2 的前 j 个字符最长公共子序列的长度。**考虑 S1i 与 S2j 值是否相等，分为两种情况：

- 当 S1i==S2j 时，那么就能在 S1 的前 i-1 个字符与 S2 的前 j-1 个字符最长公共子序列的基础上再加上 S1i 这个值，最长公共子序列长度加 1，即 `dp [i][j] = dp [i-1][j-1] + 1`。

  当 S1i != S2j 时，此时最长公共子序列为 S1 的前 i-1 个字符和 S2 的前 j 个字符最长公共子序列，或者 S1 的前 i 个字符和 S2 的前 j-1 个字符最长公共子序列，取它们的最大者，即 `dp [i][j] = max{ dp [i-1][j], dp [i][j-1] }`。

```
public static int lengthOfLCS(int[] nums1, int[] nums2) {

    int n1 = nums1.length, n2 = nums2.length;
```

```
    int[][] dp = new int[n1 + 1][n2 + 1];
    for(int i = 1; i < n1 + 1; i++) {
        for(int j = 1; j < n2 + 1; j++) {
            if(nums1[i - 1] == nums2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[n1][n2];
}
```

20、最长递增子序列：给定一个无序的整数数组，找到其中最长上升子序列的长度。

```
public int lengthOfLIS(int[] nums) {
    if(nums == null || nums.length == 0) {
        return 0;
    }
    int[] dp = new int[nums.length];
    for(int i = 0; i < nums.length; i++) {
        int max = 1;
        for(int j = 0; j < i; j++) {
            if(nums[i] > nums[j]) {
                max = Math.max(max, dp[j] + 1);
            }
        }
        dp[i] = max;
    }
    Arrays.sort(dp);
    return dp[dp.length - 1];
}
```

21、常见垃圾回收算法?

判断对象是否是垃圾的算法：

> - 引用计数算法（Reference Counting Collector）：**给对象添加一个引用计数器，每当有一个地方引用他时，计数器就加一，引用失败计数器减一**，计数器为零的对象就不可能再被使用。
> - 根搜索算法（Tracing Collector）：通过一系列的称为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索走过的路径称为引用链。**当一个对象到GC Roots没有任何引用链相连的时候，则证明此对象是不可用的**。否则即是可达的。

GC Root可以是：虚拟机栈（栈帧中的本地变量表）中引用的对象；本地方法栈中JNI（即一般说的native方法）引用的对象；方法区中的静态变量和常量引用的对象。

回收：清理"垃圾"占用的内存空间而非对象本身。

标记—清除算法：分为"标记"和"清除"两个阶段：首先**标记出所需回收的对象，在标记完成后统一回收掉所有被标记的对象**，算法最大的问题是**内存碎片化严重** 。

标记—整理算法：标记的过程与标记—清除算法中的标记过程一样，但**对标记后出的垃圾对象的处理情况有所不同，它不是直接对可回收对象进行清理，而是让所有的对象都向一端移动，然后直接清理掉端边界以外的内存。**

复制算法（Copying Collector）：**将内存按容量分为大小相等的两块，每次只使用其中的一块（对象面），当这一块的内存用完了，就将还存活着的对象复制到另外一块内存上面（空闲面），然后再把已使用过的内存空间一次清理掉。**可用内存被压缩到了原本的一半。且存活对象增多的话， Copying 算法的效率会大大降低。

发生地点：**一般发生在堆内存中，因为大部分的对象都储存在堆内存中。**

分代收集算法：

Java的堆内存基于Generation算法（Generational Collector）划分为**新生代、年老代和持久代。**新生代又被进一步划分为Eden和Survivor区，最后Survivor由FromSpace（Survivor0）和ToSpace（Survivor1）组成。所有通过new创建的对象的内存都在堆中分配。分代收集基于这样一个事实：**不同的对象的生命周期是不一样的。因此，可以将不同生命周期的对象分代，不同的代采取不同的回收算法进行垃圾回收（GC）**，以便提高回收效率。

- 新生代与复制算法：目前大部分 JVM 的 GC **对于新生代都采取 Copying 算法**，因为**新生代中每次垃圾回收都要回收大部分对象，即要复制的操作比较少**，但通常并不是按照 1： 1 来划分新生代。一般将新生代 划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用 Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另 一块 Survivor 空间中。
- 老年代与标记整理算法：**老年代因为每次只回收少量对象，因而采用标记-整理算法。**

22、从输入URL到显示页面，整个过程？

- 首先对输入的url**进行DNS解析**（找出对端服务器ip地址）。优先访问浏览器缓存，如果未命中则访问OS缓存，最后再访问DNS服务器，然后DNS服务器会递归式的查找域名记录。
- 通过ip地址，**建立TCP请求**（三次握手协议）。查询到域名对应的ip地址之后，浏览器设置Remote Address，以及端口号port等信息，确保传输层的正常通信（即TCP正常连接）。
- **发送HTTP请求。**（传输层连接已经建立完成，准备发送HTTP请求）。
- **服务器处理请求并返回HTTP报文。**（Http服务器读取HTTP Get报文，生成一个HTTPP响应报文，将web页面内容放入报文主体中，发回给主机）。
- **解析渲染服务器的响应数据。**（浏览器收到HTTP响应报文后，抽取出web页面内容，之后进行渲染，显示web页面）。
- **断开TCP连接**：TCP四次挥手。

23、find查找大于10m的文件？

```
find . -type f -size +1000000k
```

24、final、finally和finalize()之间的区别？

final 用于声明属性,方法和类, 分别表示属性不可变, 方法不可覆盖, 类不可继承. **finally**是异常处理语句结构的一部分，表示总是执行. **finalize** 是Object类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等 。

25、使用Spring的好处有哪些？缺陷有哪些?

Spring优点：

- 使用Spring的IOC容器，**将对象之间的依赖关系交给Spring，降低组件之间的耦合性**，让我们更专注于应用逻辑
- 提供了一种管理对象的方法，可以把中间层对象有效地组织起来。一个完美的框架“黏合剂”。
- 采用了分层结构，可以增量引入到项目中。
- 有利于面向接口编程习惯的养成。
- 目的之一是为了写出易于测试的代码。
- 非侵入性，应用程序对Spring API的依赖可以减至最小限度。

- 一致的数据访问介面。
- 一个轻量级的架构解决方案。

Spring缺点：

- jsp中要写很多代码、控制器过于灵活，缺少一个公用控制器
- Spring不支持分布式，这也是EJB仍然在用的原因之一。

26、操作系统中，进程间通信的方式有哪些？

- 管道：只支持半双工通信（单向交替传输）；**只能在父子进程或者兄弟进程中使用。**
- FIFO：也称为命名管道，**去除了管道只能在父子进程中使用的限制。**
- 消息队列：**消息队列可以独立于读写进程存在**，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；**读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。**
- 信号量：它是一个计数器，用于**为多个进程提供对共享数据对象的访问。**
- 共享存储：允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以这是最快的一种 IPC。 需要**使用信号量用来同步对共享存储的访问。**多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。另外 XSI 共享内存不是使用文件，而是使用内存的匿名段。
- 套接字：与其它通信机制不同的是，它**可用于不同机器间的进程通信。**

27、代码题：将一个整型数字，翻转以后得到的整型数字，需要考虑溢出的情况，

例如：123 -> 321

```
public int reverse(int x) {
    long temp = 0;
    while(x != 0) {
        temp *= 10;
        temp += x % 10;
        x /= 10;
    }
    if(temp < Integer.MIN_VALUE || temp > Integer.MAX_VALUE) {
        return 0;
    }
    return (int)temp;
}
```

28、说一说http和https区别?

HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了**SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。**简单来说，**HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。**

HTTPS和HTTP的区别主要如下：

- **https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。**
- **http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。**
- **http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。**
- **http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。**

29、代码题：一个数组，除了一个数出现次数为1，其他数的出现次数都为2，找到出现一次的那个数。

```
//异或
public int singleNumber(int[] nums) {
    int ans = nums[0];
    if(nums.length > 0) {
        for(int i = 1; i < nums.length; i++) {
            ans = ans ^ nums[i];
        }
    }
    return ans;
}
```

30、String a = new String("abc")几个对象？

使用这种方式一共会**创建两个字符串对象**（前提是 String Pool 中还没有 "abc" 字符串对象）。

- "abc" 属于字符串字面量，因此编译时期会在 String Pool 中创建一个字符串对象，指向这个 "abc" 字符串字面量；
- 而使用 new 的方式会在堆中创建一个字符串对象。

31、spring中autowired resource区别？

共同点：两者都可以写在字段和setter方法上。两者如果都写在字段上，那么就不需要再写setter方法。

不同点：

- @Autowired为Spring提供的注解，需要导入包org.springframework.beans.factory.annotation.Autowired;只按照byType注入。 @Autowired注解是按照类型（byType）装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它的required属性为false。
- @Resource默认按照ByName自动注入，由J2EE提供，需要导入包javax.annotation.Resource。@Resource有两个重要的属性：name和type，而Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以，如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不制定name也不制定type属性，这时将通过反射机制使用byName自动注入策略。

32、不借助jdk实现加法？

```
public int Add(int num1,int num2) {
    while(num2 != 0) {
        int sum = num1 ^ num2;
        int carry = (num1 & num2) << 1;
        num1 = sum;
        num2 = carry;
    }
    return num1;
}
```

33、进程与线程区别？

- 拥有资源：**进程是资源分配的基本单位，但是线程不拥有资源**，线程可以访问隶属进程的资源。

- 调度：**线程是独立调度的基本单位**，在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。
- 通信方面：**线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助 IPC。**

34、链表中环的入口结点?

```
public ListNode EntryNodeOfLoop(ListNode pHead) {
    ListNode first = pHead;
    ListNode second = pHead;
    while(first != null && first.next != null) {
        first = first.next;
        second = second.next.next;
        if(first == second) {
            first = pHead;
            while(first != second) {
                first = first.next;
                second = second.next;
            }
            return first;
        }
    }
    return null;
}
```

35、run()方法和start()方法的区别？

- start（） 方法来启动线程，真正实现了多线程运行。这时无需等待 run 方法体代码执行完毕， 可以直接继续执行下面的代码；
- 通过调用 Thread 类的 start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行；
- 方法 run()称为线程体，它包含了要执行的这个线程的内容，线程就进入了运行状态，开始运行 run 函数当中的代码。 Run 方法运行结束， 此线程终止。然后 CPU 再调度其它线程。

36、wait() 和 sleep() 的区别?

- wait() 是 Object 的方法，而 sleep() 是 Thread 的静态方法；
- sleep()方法导致了程序暂停执行指定的时间，让出 cpu 该其他线程，但是他的监控状态依然 保持者，当指定的时间到了又会自动恢复运行状态；
- wait() 会释放锁，sleep() 不会；
- 当调用 wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用 notify() 方法后本线程才进入对象锁定池准备获取对象锁进入运行状态。

37、接口和抽象类的区别？

- 抽象类是用来**捕捉子类的通用特性**的。它不能被实例化，只能被用作子类的超类。抽象类是被用来创建继承层级里子类的模板。
- **接口是抽象方法的集合。**如果一个类实现了某个接口，那么它就继承了这个接口的抽象方法。

区别：

> 从语法层面来说：
>
> - 抽象类可以提供成员方法的实现细节，而接口中只能存在抽象方法
> - 抽象类中成员变量可以是多种类型，接口中成员变量必须用public，static，final修饰

> - 一个类只能继承一个抽象类，但可以实现多个接口
> - 抽象类中允许含有静态代码块和静态方法，接口不能
>
> 从设计层面而言：
>
> - 抽象类是对整一个类的属性，行为等方面进行抽象，而接口则是对行为抽象。就好比飞机和鸟，抽象类抽象出的是飞行物类。而接口则是抽闲出飞行方法。
> - 抽象类是一个模板式的设计，当在开发过程中出现需求更改的情况，只需要更改抽象类而不需要更改它的子类。接口是一种辐射性设计，当接口的内容发生改变时，需要同时对实现它的子类进行相应的修改。
> - 抽象类可以类比为模板，而接口可以类比为协议。

什么时候使用抽象类和接口：

- 如果拥有一些方法并且想让它们中的一些有默认实现，使用抽象类。
- 如果想实现多重继承，必须使用接口。由于Java不支持多继承，子类不能够继承多个类，但可以实现多个接口。
- 如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口，那么就需要改变所有实现了该接口的类。

38、http中get与post的区别？

GET：**从指定的资源请求数据。**

POST：**向指定的资源提交要被处理的数据。**

由于HTTP的规定和浏览器/服务器的限制，导致它们在应用过程中体现出一些不同。

	GET	POST
后退按钮/刷新	无害	数据会被重新提交（浏览器应该告知用户数据会被重新提交）。
缓存	能被缓存	不能缓存
编码方式	只能进行url编码	支持多种编码方式
是否保留在浏览历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	发送数据，GET 方法向 URL 添加数据，但URL的长度是受限制的。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	安全性较差，因为参数直接暴露在url中	因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。
传参方式	get参数通过url传递	post放在request body中。

39、http状态码有哪几种?

状态码，100~199表示请求已收到继续处理，200~299表示成功，300~399表示资源重定向，400~499表示客户端请求出错，500~599表示服务器端出错

> 200：响应成功

> 302：跳转，重定向
>
> 400：客户端有语法错误
>
> 403：服务器拒绝提供服务
>
> 404：请求资源不存在
>
> 500：服务器内部错误

40、写代码：输出一个数组中某元素出现的次数超过数组长度一半的元素，没有则输出null。

```
public int MoreThanHalfNum_Solution(int [] array) {
    Map<Integer, Integer> map = new HashMap<>();
    for(int num : array) {
        if(map.containsKey(num)) {
            map.put(num, map.get(num) + 1);
        } else {
            map.put(num, 1);
        }
    }
    for(int num : array) {
        if(map.get(num) > array.length / 2) {
            return num;
        }
    }
    return 0;
}
```

41、反转单词序列?

```
public String ReverseSentence(String str) {
    StringBuffer res = new StringBuffer();
    if("".equals(str.trim())) {
        return str;
    }
    String[] strs = str.split(" ");
    for(int i = strs.length - 1; i >= 0; i--) {
        res.append(strs[i]).append(" ");
    }
    return res.toString().trim();
}
```

42、为什么要有端口?

所谓端口号就是**具有网络功能的应用软件的标识号**。注意，端口号是不固定的，即可以由用户手工可以分配（当然，一般在软件编写时就已经定义）。当然，有很多应用软件有公认的默认的端口，比如FTP：20和21，HTTP：80，TELNET：23等。一个软件可以拥有多个端口号，这证明这个软件拥有不止一个网络功能。

43、信号量?

信号量(Semaphore)，是在多线程环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被并发调用。在进入一个关键代码段之前，线程必须获取一个信号量；一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。为了完成这个过程，需要创建一个信号量VI，然后将Acquire Semaphore VI以及Release Semaphore VI分别放置在每个关键代码段的首末端。确认这些信号量VI引用的是初始创建的信号量。

44、静态内部类?

静态内部类和非静态内部类区别：

- **内部静态类不需要有指向外部类的引用。**但非静态内部类需要持有对外部类的引用。
- 非静态内部类能够访问外部类的静态和非静态成员。**静态类不能访问外部类的非静态成员。**他只能访问外部类的静态成员。
- 一个非静态内部类不能脱离外部类实体被创建，一个非静态内部类可以访问外部类的数据和方法，因为他就在外部类里面。