9/4

1、数组和链表的区别？

数组：

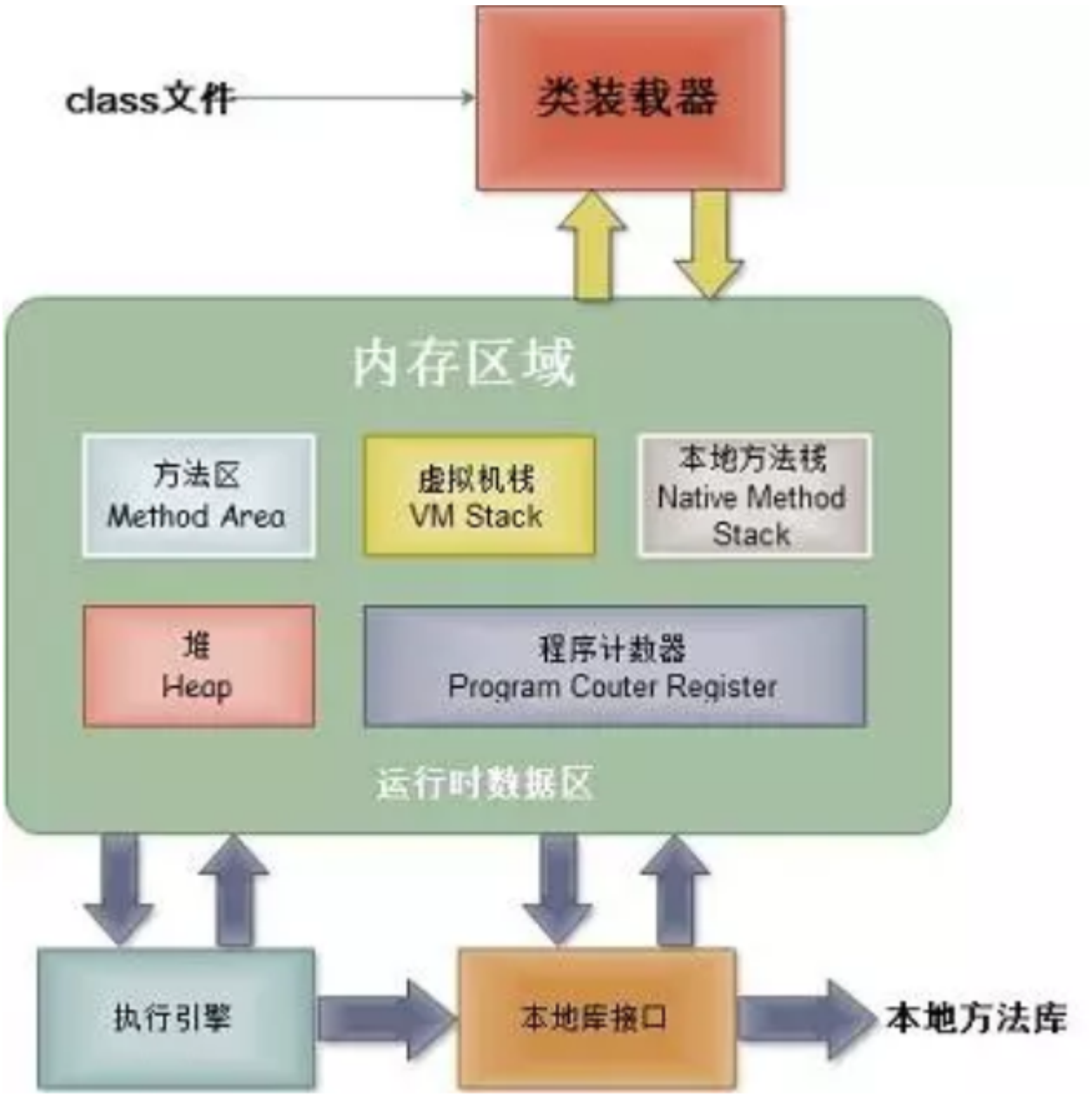
- 在内存中，数组是一块**连续的区域**；
- 数组需要**预留空间，在使用前要先申请占内存的大小，可能会浪费内存空间**；
- **插入数据和删除数据效率低**（插入数据时，这个位置后面的数据在内存中都要向后移。删除数据时，这个数据后面的数据都要往前移动）；
- **随机读取效率很高。**因为数组是连续的，知道每一个数据的内存地址，可以直接找到给地址的数据；
- **不利于扩展，数组定义的空间不够时要重新定义数组。**

链表：

- 在**内存中可以存在任何地方，不要求连续**；
- **每一个数据都保存了下一个数据的内存地址**，通过这个地址找到下一个数据；
- **增加数据和删除数据很容易**；
- **查找数据时效率低，因为不具有随机访问性**，所以访问某个位置的数据都要从第一个数据开始访问，然后根据第一个数据保存的下一个数据的地址找到第二个数据，以此类推；
- **不指定大小，扩展方便。**链表大小不用定义，数据随意增删。

2、Java的内存管理机制？垃圾回收?

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。**

垃圾回收(Garbage Collection)是**Java虚拟机(JVM)垃圾回收器提供的一种用于在空闲时间不定时回收无任何对象引用的对象占据的内存空间的一种机制。**

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。

垃圾：无任何对象引用的对象。

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。**

3、类加载?

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。

4、进程和线程的区别？

- 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。**
- 系统开销： **由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销线程时的开销。**类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而线程切换时只需保存和设置少量寄存器内容，开销很小。
- 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。** 进程间通信（**IPC**，Inter-Process Communication）。

5、什么是死锁？如何避免死锁?

死锁是指**在一组进程中的各个进程均占有不会释放的资源，但因互相申请被其他进程所站用不会释放的资源而处于的一种永久等待状态。**死锁的四个必要条件：

- 互斥条件(Mutual exclusion)：资源不能被共享，只能由一个进程使用。
- 请求与保持条件(Hold and wait)：已经得到资源的进程可以再次申请新的资源。
- 非剥夺条件(No pre-emption)：已经分配的资源不能从相应的进程中被强制地剥夺。
- 循环等待条件(Circular wait)：系统中若干进程组成环路，该环路中每个进程都在等待相邻进程正占用的资源。

java中产生死锁可能性的最根本原因是：1）是多个线程涉及到多个锁，这些锁存在着交叉，所以可能会导致了一个锁依赖的闭环；2）默认的锁申请操作是阻塞的。

如，线程在获得一个锁L1的情况下再去申请另外一个锁L2，也就是锁L1想要包含了锁L2，在获得了锁L1，并且没有释放锁L1的情况下，又去申请获得锁L2，这个是产生死锁的最根本原因。

避免死锁：

- 破坏互斥条件：例如假脱机打印机技术允许若干个进程同时输出，唯一真正请求物理打印机的进程是打印机守护进程。
- 破坏占有和等待条件：一种实现方式是规定所有进程在开始执行前请求所需要的全部资源。
- 破坏不可抢占条件。
- 破坏环路等待：给资源统一编号，进程只能按编号顺序来请求资源。

6、面向对象和多态？

面向对象：是一种程序设计思想。 面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

多态就是指**程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定**，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。因为在程序运行时才确定具体的类，这样，**不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变**，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让**程序可以选择多个运行状态**，这就是多态性。

Java实现多态有三个必要条件：继承、重写、向上转型。

- 继承：在多态中必须存在有继承关系的子类和父类。
- 重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。
- 向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备技能调用父类的方法和子类的方法。

Java中有两种形式可以实现多态，**继承、接口**：

- 基于继承的实现机制主要表现在**父类和继承该父类的一个或多个子类对某些方法的重写，多个子类对同一方法的重写可以表现出不同的行为。**
- 基于接口的多态中，指向接口的引用必须是指定这实现了该接口的一个类的实例程序，**在运行时，根据对象引用的实际类型来执行对应的方法。**

7、怎么实现线程同步？

- 同步方法 `synchronized` 关键字修饰的方法（悲观锁）
- 使用特殊域变量(`volatile`)实现线程同步（保持可见性，多线程更新某一个值时，案例线程安全单例双检查锁）
- `ThreadLocal` （每个线程获取的都是该变量的副本）
- 使用重入锁实现线程同步（相对synchronized锁粒度更细了，效率高）

  > 一个java.util.concurrent包来支持同步。
  >
  > `ReentrantLock` 类是可重入、互斥、实现了Lock接口的锁
  >
  > ReentrantLock() : 创建一个ReentrantLock实例
  >
  > lock() : 获得锁
  >
  > unlock() : 释放锁

- java.util.concurrent.atomic包 （乐观锁）

  > 方便程序员在多线程环境下，无锁的进行原子操作
  >
  > AtomicInteger

> CAS，Compare and Swap即比较并交换。java.util.concurrent包**借助CAS实现了区别于synchronized同步锁的一种乐观锁**。乐观锁就是每次去取数据的时候都乐观的认为数据不会被修改，所以不会上锁，但是在更新的时候会判断一下在此期间数据有没有更新。CAS有3个操作数：内存值V，旧的预期值A，要修改的新值B

8、两个栈实现队列？

```
public class Solution {
    Stack<Integer> stack1 = new Stack<Integer>();
    Stack<Integer> stack2 = new Stack<Integer>();
    //栈1用来push
    //栈2用来pop
    public void push(int node) {
        stack1.push(node);
    }

    public int pop() {
        if(stack2.isEmpty()) {
            while(!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }
}
```

9、线程间的通信方式？

- 锁机制：包括互斥锁、条件变量、读写锁
  - 互斥锁提供了以排他方式防止数据结构被并发修改的方法。
  - 读写锁允许多个线程同时读共享数据，而对写操作是互斥的。
  - 条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。
- 信号量机制(Semaphore)：包括无名线程信号量和命名线程信号量。
- 信号机制(Signal)：类似进程间的信号处理。

10、进程间的通信方式?

- 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。
- 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。**

11、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**

12、tcp为什么可靠？

1）**确认和重传机制**

建立连接时三次握手同步双方的"序列号 + 确认号 + 窗口大小信息"，是确认重传、流控的基础

传输过程中，如果Checksum校验失败、丢包或延时，发送端重传。

2）**数据排序**

TCP有专门的序列号SN字段，可提供数据re-order

3）**流量控制**

滑动窗口和计时器的使用。TCP窗口中会指明双方能够发送接收的最大数据量，**发送方通过维持一个发送滑动窗口来确保不会发生由于发送方报文发送太快接收方无法及时处理的问题。**

4） **拥塞控制**

TCP的拥塞控制由4个核心算法组成：

"慢启动"（Slow Start）

"拥塞避免"（Congestion avoidance）

"快速重传 "（Fast Retransmit）

"快速恢复"（Fast Recovery）

如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而**拥塞控制是为了降低整个网络的拥塞程度。**

> 慢开始与拥塞避免：
>
> 发送的最初执行慢开始，令 cwnd = 1，发送方只能发送 1 个报文段；当收到确认后，将 cwnd 加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...
>
> 注意到慢开始每个轮次都将 cwnd 加倍，这样会让 cwnd 增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始门限 ssthresh，当 cwnd >= ssthresh 时，进入拥塞避免，每个轮次只将 cwnd 加 1。
>
> 如果出现了超时，则令 ssthresh = cwnd / 2，然后重新执行慢开始。
>
> 快重传与快恢复：
>
> 在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到 M1 和 M2，此时收到 M4，应当发送对 M2 的确认。
>
> 在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个 M2，则 M3 丢失，立即重传 M3。

> 在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令 ssthresh = cwnd / 2 ， cwnd = ssthresh，注意到此时直接进入拥塞避免。
>
> 慢开始和快恢复的快慢指的是 cwnd 的设定值，而不是 cwnd 的增长速率。慢开始 cwnd 设定为 1，而快恢复 cwnd 设定为 ssthresh。

13、归并排序？快速排序？堆排序？

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

```
//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
        }
    }
}
```

14、协程的区别?

- 协程是属于线程的。**协程程序是在线程里面跑的，因此协程又称微线程和纤程**等。
- 协没有线程的上下文切换消耗。**协程的调度切换是用户(程序员)手动切换的,因此更加灵活**,因此又叫用户空间线程。
- 原子操作性。由于协程是用户调度的，所以不会出现执行一半的代码片段被强制中断了，因此无需原子操作锁。

15、tcp三次握手？

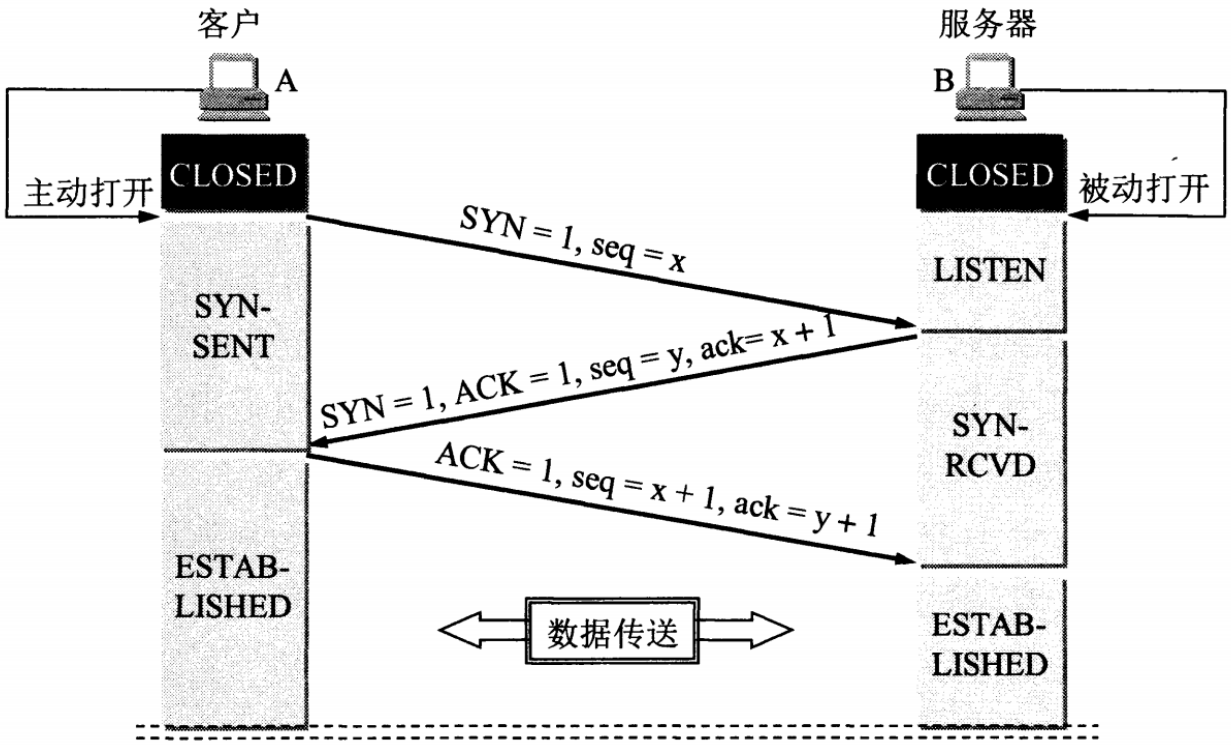


图 5-28 用三报文握手建立 TCP 连接

第一次握手：建立连接时，客户端发送syn包（syn=j）到服务器，并进入SYN_SENT状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态；

第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1），此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。

在服务端对客户端的请求进行回应(第二次握手)后，就会理所当然的认为连接已建立，而如果客户端并没有收到服务端的回应呢？此时，客户端仍认为连接未建立，服务端会对已建立的连接保存必要的资源，如果大量的这种情况，服务端会崩溃。

16、Linux下面的命令？

- **删除非空目录：rm -rf file目录**
- 查看文件：cat
- **改变文件读、写、执行等属性 chmod**
- 管道：批处理命令连接执行，使用 |
- FIND： 文件查找
- Grep：文本搜索，**-c 统计文件中包含文本的次数**
- 查询进程：ps -ef ， 查询归属于用户xuexue的进程： ps -ef | grep xuexue
- 列出所有端口(包括监听和未监听的)： netstat -a
- 性能监控：
  - 监控CPU： sar -u
  - 查询内存： sar -r 1 2
  - 查询硬盘使用： df -h
- 查看当前目录所占空间大小： du -sh

17、get和post的区别？

	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。

18、http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

19、ACID？

> 原子性（Atomicity）：事务作为一个整体被执行，包含在其中的对数据库的**操作要么全部被执行，要么都不执行。**
>
> 一致性（Consistency）：事务应确保**数据库的状态从一个一致状态转变为另一个一致状态。**在一致性状态下，所有事务对一个数据的读取结果都是相同的。
>
> 隔离性（Isolation）：多个事务并发执行时，**一个事务的执行不应影响其他事务的执行。**一个事务所做的修改在最终提交以前，对其它事务是不可见的。
>
> 持久性（Durability）：已**被提交的事务对数据库的修改应该永久保存在数据库中。**

20、数据库并发策略?

并发控制一般采用三种方法，分别是**乐观锁和悲观锁以及时间戳。**

- 乐观锁：

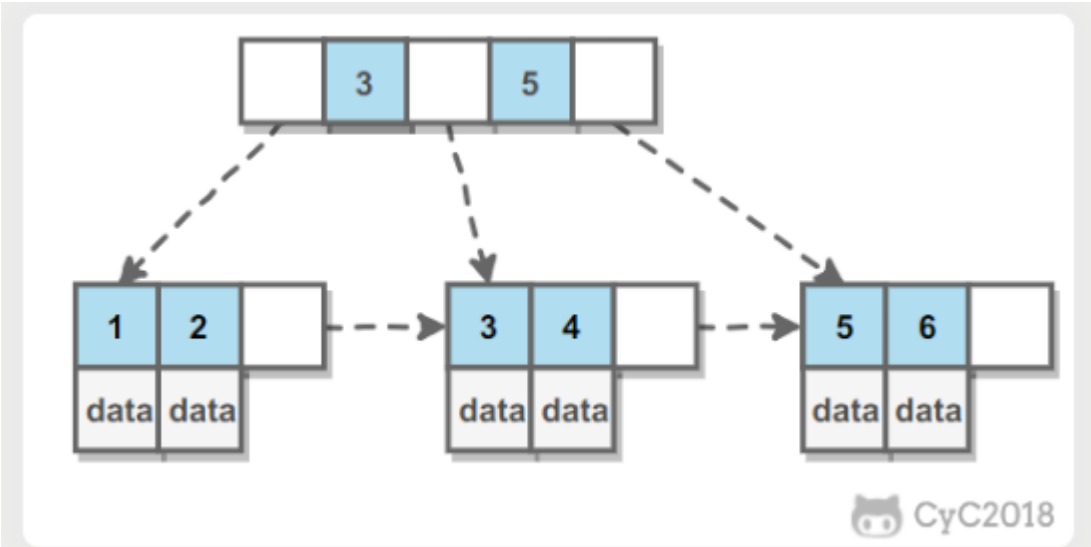
    - 乐观锁认为**一个用户读数据的时候，别人不会去写自己所读的数据**；悲观锁就刚好相反，觉得自己读数据库的时候，别人可能刚好在写自己刚读的数据，其实就是持一种比较保守的态度；时间戳就是不加锁，通过时间戳来控制并发出现的问题。
    - 一般利用表字段的version+1来实现（如SVN、GIT提交代码就是这样的）；性能高、重试失败成本不高建议乐观。
- 悲观锁：
    - 悲观锁就是**在读取数据的时候，为了不让别人修改自己读取的数据，就会先对自己读取的数据加 锁，只有自己把数据读完了，才允许别人修改那部分数据**，或者反过来说，就是自己修改某条数 据的时候，不允许别人读取该数据，只有等自己的整个事务提交了，才释放自己加上的锁，才允 许其他用户访问那部分数据。
    - 一般是 where id=XX for update 来实现 （一般银行转账、工单审批）；性能低，但安全，失败成功高建议悲观，使用不当有死锁风险。
    - 悲观锁所说的加“锁”，其实分为几种锁，分别是： 排它锁（写锁） 和共享锁（读锁） 。
- 时间戳：时间戳就是**在数据库表中单独加一列时间戳**，比如“TimeStamp”， 每次读出来的时候，把该字段也读出来，当写回去的时候，把该字段加 1，提交之前 ，跟数据库的该字段比较一次，如果比数据库的值大的话，就允许保存，否则不允许保存，这种处理方法虽然不使用数据库系统提供的锁机制，但是这种方法可以大大提高数据库处理的并发量。

21、数据库中索引是什么，为什么索引可以实现高效查找，描述一下B-树查找的过程？

数据库索引就是一种**加快海量数据查询**的关键技术。索引是**关系数据库中对某一列或多个列的值进行预排序的数据结构。通过使用索引，可以让数据库系统不必扫描整个表，而是直接定位到符合条件的记录**，这样就大大加快了查询速度。

1）数据结构：

- B Tree 指的是 Balance Tree，也就是平衡树。平衡树是一颗查找树，并且所有叶子节点位于同一层。
- B+ Tree 是**基于 B Tree 和叶子节点顺序访问指针进行实现**，它具有 B Tree 的平衡性，并且**通过顺序访问指针来提高区间查询的性能。**
- 在 B+ Tree 中，一个节点中的 key 从左到右非递减排列 。



2）操作：

进行查找操作时，**首先在根节点进行二分查找，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。**直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的 data。

插入删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

3）与红黑树的比较：

红黑树等平衡树也可以用来实现索引，但是文件系统及数据库系统普遍采用 B+ Tree 作为索引结构，主要有以下两个原因：

> - 更少的查找次数
>
> 平衡树查找操作的时间复杂度和树高 h 相关，O(h)=O(logdN)，其中 d 为每个节点的出度。
>
> 红黑树的出度为 2，而 B+ Tree 的出度一般都非常大，所以红黑树的树高 h 很明显比 B+ Tree 大非常多，查找的次数也就更多。
>
> - 利用磁盘预读特性
>
> 为了减少磁盘 I/O 操作，磁盘往往不是严格按需读取，而是每次都会预读。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的磁盘旋转时间，速度会非常快。
>
> 操作系统一般将内存和磁盘分割成固定大小的块，每一块称为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次 I/O 就能完全载入一个节点。并且可以利用预读特性，相邻的节点也能够被预先载入。

22、二叉树的遍历 递归和非递归？

```
//遍历
public void preOrder() {//前序遍历
    preOrder(root);
}
private void preOrder(Node node) {
    //        if(node == null) {
    //            return;
    //        }
    if(node != null) {
        System.out.println(node.e);
        preOrder(node.left);
        preOrder(node.right);
    }
}

public void preOrderNR() {//前序非递归遍历
    Stack<Node> stack = new Stack<>();
    stack.push(root);
    while (!stack.isEmpty()) {
        Node cur = stack.pop();
        System.out.println(cur.e);
        if(cur.right != null) {
            stack.push(cur.right);
        }
        if(cur.left != null) {
            stack.push(cur.left);
        }
    }
}
```

```
public void inOrder() {//中序排序
    inOrder(root);
}

private void inOrder(Node node) {
    if(node == null) {
        return;
    }
    inOrder(node.left);
    System.out.println(node.e);
    inOrder(node.right);
}

public void postOrder() {//后序遍历
    postOrder(root);
}

private void postOrder(Node node) {
    if(node == null) {
        return;
    }
    postOrder(node.left);
    postOrder(node.right);
    System.out.println(node.e);
}

//层序遍历
public void levelOrder() {
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty()) {
        Node cur = q.remove();
        System.out.println(cur.e);
        if(cur.left != null) {
            q.add(cur.left);
        }
        if(cur.right != null) {
            q.add(cur.right);
        }
    }
}
```

23、String StringBuilder StringBuffer ?

线程安全性：**线程安全 String 、StringBuffer（**内部使用 synchronized 进行同步**）；非线程安全 StringBuilder。**

执行效率： StringBuilder > StringBuffer > String

存储空间：**String 的值是不可变的**，每次对String的操作都会生成新的String对象，效率低耗费大量内存空间，从而引起GC。**StringBuffer和StringBuilder都是可变。**

使用场景：如果要操作少量的数据用 String ； 单线程操作字符串缓冲区下操作大量数据 = StringBuilder ； 多线程操作字符串缓冲区下操作大量数据 = StringBuffer。

24、直接插入排序？

```
//每次都将当前元素插入到左侧已经排序的数组中，使得插入之后左侧数组依然有序。
public class insert {
    public static void sort(int[] num) {
         //第1个数肯定是有序的，从第2个数开始遍历，依次插入有序序列
         for(int i = 1; i < num.length; i++) {
            for(int j = i; j > 0 && num[j] < num[j - 1]; j--) {
                swap(num, j, j - 1);
            }
        }
    }
    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

25、链表是否有环、怎样找环入口？

```
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;
}
```

26、Java反射机制?

JAVA反射机制是**在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性**；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。Java反射可以**用来获取一个class对象或实例化一个class表示的类的对象，还可以获取构造方法，成员变量，成员方法。**

反射的优点缺点：

- 优点：**可扩展性** ： 应用程序可以利用全限定名创建可扩展对象的实例，来使用来自外部的用户自定义类。**类浏览器和可视化开发环境** ： 一个类浏览器需要可以枚举类的成员。可视化开发环境（如 IDE）可以从利用反

射中可用的类型信息中受益，以帮助程序员编写正确的代码。**调试器和测试工具**： 调试器需要能够检查一个类里的私有成员。测试工具可以利用反射来自动地调用类里定义的可被发现的 API 定义，以确保一组测试中有较高的代码覆盖率。

- 缺点：**性能瓶颈**：反射相当于一系列解释操作，通知 JVM 要做的事情，性能比直接的java代码要慢很多；**安全限制**； **内部暴露。**

反射的应用：

- JDBC 的数据库的连接：通过Class.forName()加载数据库的驱动程序 （通过反射加载，前提是引入相关了Jar包）；
- Spring 框架的使用：Spring 通过 XML 配置模式装载 Bean 的过程；

27、实现线程的方式和线程安全问题？

- 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。

28、 c/s 和 b/s的区别和联系？

1）C / S： Client / server 的简写，这里 Server 指的是 DBServer。

- 特点：**每个客户端必须安装（部署）一份应用程序，一般在局域网使用，只针对特定的客户群**；
- **响应速度快，交互比较好**；
- 缺点：**客户端数目受服务器限制，维护升级比较麻烦**；
- 开发技术： vb，Delphi，winforms，AWT/Swing，SWT。

2）B / S：Brower/Server 的简写，这里的 Server 指的有两个，WebServer 与 DBServer。

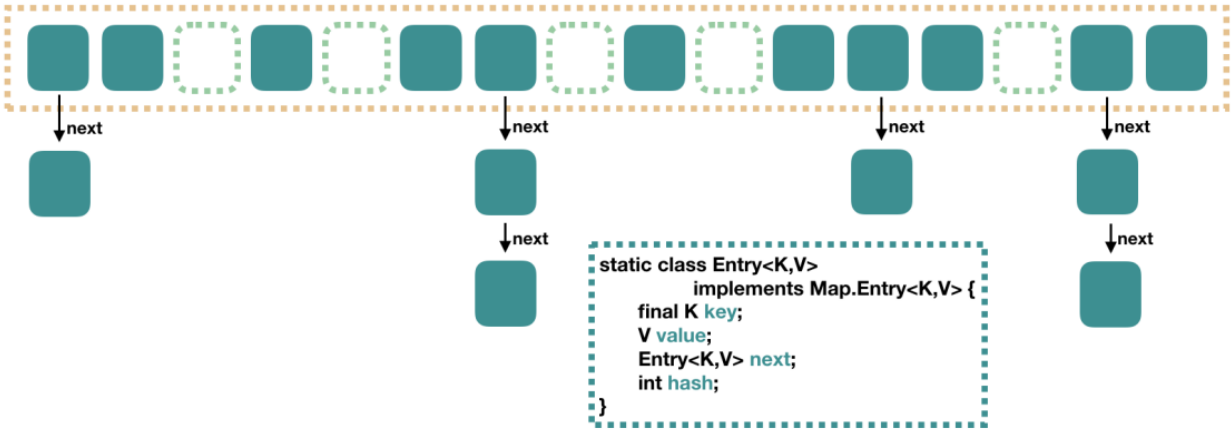
- 特点：**客户端不需要部署应用程序，只要一个浏览器。**一般 web 网使用，但新的趋势是 B/S 项目越来越多，甚至传统使用 C/S 开发的项目也使用 B/S。富客户端技术（ajax，jquery，flex 等）的兴起，使 B/S 更如日中天。
- 优点：**程序升级维护方便，代码只在 WebServer 中有一份。**因为最终程序运行结果在客户浏览器中显示，所以对客户端平台无限制。
- 缺点：**交互式没有 C/S 好。**

29、Hashmap、Hashtable、CurrentHashMap？

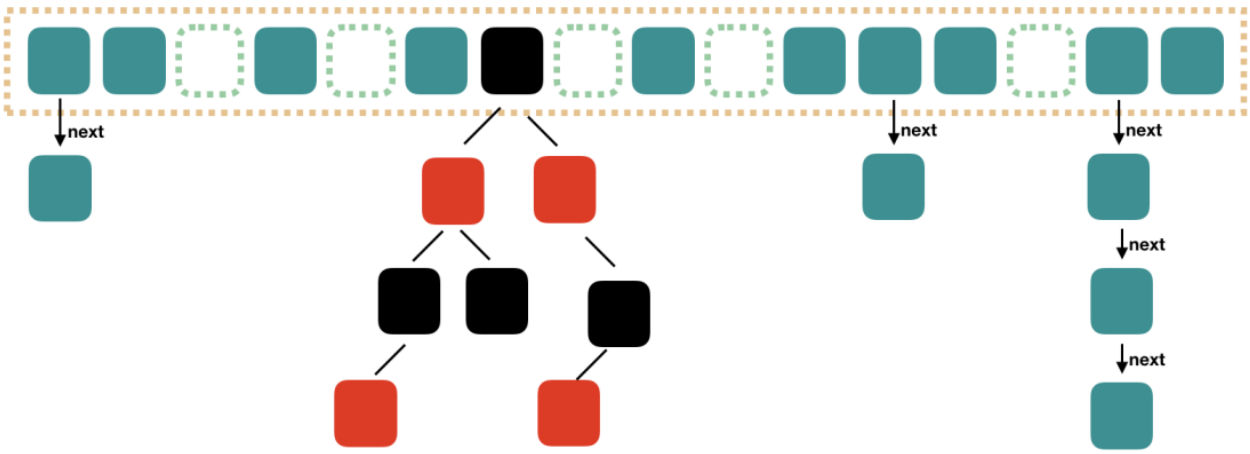
1）HashMap：**允许键、值为null；基于数组+链表实现；非线程安全。**在HashMap**进行扩容重哈希时导致Entry链形成环。一旦Entry链中有环，会导致在同一个桶中进行插入、查询、删除等操作时陷入死循环。**

- 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。**

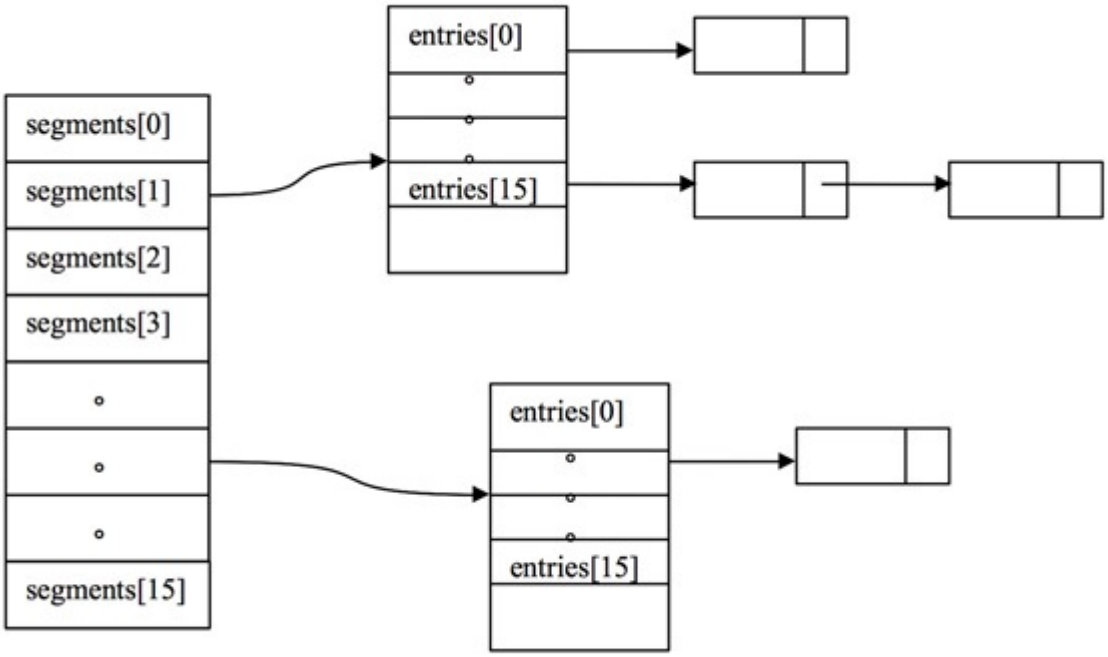
2）HashTable：**HashTable是线程安全；键、值均不可为nul**l，其余与HashMap大致相同。

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。**

3）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。**

附：

自旋锁和阻塞锁区别：要不要放弃处理器的执行时间。**对于阻塞锁和自旋锁来说，都是要等待获得共享资源**，但是阻塞锁是放弃了cpu时间，进入了等待区，等待被唤醒。而自旋锁一直"自旋"在那里，时刻的检查共享资源是否可以被访问。

CAS：是项**乐观锁技术**，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其他线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

30、单例模式是什么，优缺点?

1、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。**

2、单例模式的要点有三个：

- 某个类只能有一个实例；
- 它必须自行创建这个实例；
- 它必须自行向整个系统提供这个实例。

3、优缺点?

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）。

4、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。

5、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;
    }
}
```

31、还知道什么设计模式?

观察者模式：

> 1、定义了对象之间的一对多依赖关系，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并且自动更新。
>
> 2、观察者模式具备：事件的源、事件对象、事件处理对象。以西游记为例，唐僧就是事件源，孙悟空、猪八戒、沙僧都是事件处理对象，事件对象包含有关事件和事件的源信息（就是记录唐僧发生事件的信息）。
>
> 事件源类三要素：
>
> - 私有的监听者列表
> - 公有的向监听者列表添加监听者的方法
> - 公有的发生事情的方法

32、求一个栈的最小元素？

```
public class Solution {

    Stack<Integer> stack = new Stack<>();
    Stack<Integer> stack_min = new Stack<>();

    public void push(int node) {
```

```
        stack.push(node);
        if(stack_min.isEmpty() || stack_min.peek() > node) {
            stack_min.push(node);
        } else {
            stack_min.push(stack_min.peek());
        }
    }

    public void pop() {
        if(stack.isEmpty()) {
            return;
        }
        stack.pop();
        stack_min.pop();
    }

    public int top() {
        if(stack.isEmpty()) {
            return -1;
        }
        return stack.peek();
    }

    public int min() {
        if(stack_min.isEmpty()) {
            return -1;
        }
        return stack_min.peek();
    }
}
```

33、查看80端口的命令?

netstat -anpt | grep 80

34、ARP协议 ?

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。**

35、静态变量和非静态变量在多线程中的区别?

> 静态变量:线程非安全。
>
> 静态变量即类变量,位于方法区,为所有对象共享,共享一份内存,一旦静态变量被修改,其他对象均对修改可见,故线程非安全。
>
> 实例变量:单例模式(只有一个对象实例存在)线程非安全,非单例线程安全。

> 实例变量为对象实例私有,在虚拟机的堆中分配,若在系统中只存在一个此对象的实例,在多线程环境下,"犹如"静态变量那样,被某个线程修改后,其他线程对修改均可见,故线程非安全;如果每个线程执行都是在不同的对象中,那对象与对象之间的实例变量的修改将互不影响,故线程安全。
>
> 局部变量:线程安全。
>
> 每个线程执行时将会把局部变量放在各自栈帧的工作内存中,线程间不共享,故不存在线程安全问题。

36、如何理解计算机网络的分层，为什么分层？

建立七层模型的主要目的是为**解决异种网络互连时所遇到的兼容性问题。**它的最大优点是**将服务、接口和协议这三个概念明确地区分开来**：**服务说明某一层为上一层提供一些什么功能，接口说明上一层如何使用下层的服务，而协议涉及如何实现本层的服务**；这样各层之间具有很强的独立性，互连网络中各实体采用什么样的协议是没有限制的，只要向上提供相同的服务并且不改变相邻层的接口就可以了。网络七层的划分也是为了**使网络的不同功能模块（不同层次）分担起不同的职责**，从而带来如下好处：

- **减轻问题的复杂程度**，一旦网络发生故障，可迅速定位故障所处层次，便于查找和纠错；
- **在各层分别定义标准接口，使具备相同对等层的不同网络设备能实现互操作，各层之间则相对独立，一种高层协议可放在多种低层协议上运行**；
- **能有效刺激网络技术革新**，因为每次更新都可以在小范围内进行，不需对整个网络动大手术；

37、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。**

38、如何往GitHub上提交数据的？

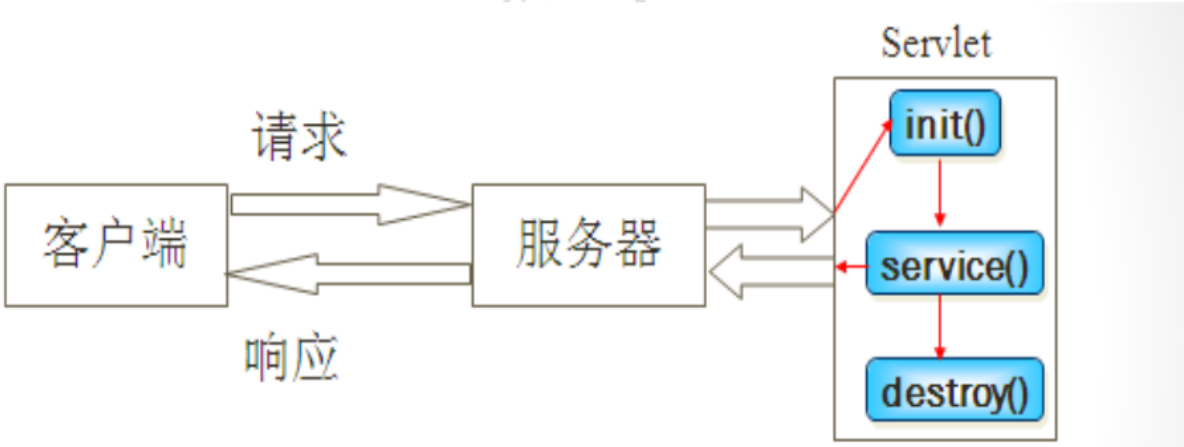
- 执行指令进行**初始化**，会在原始文件夹中生成一个隐藏的文件夹.git（ git init）
- 执行指令**将文件添加到本地仓库**：（ git add . //添加当前文件夹下的所有文件）
- **git commit** -m "layout" //引号中的内容为对该文件的描述
- **关联git仓库**：新建一个repository时会出现下面的代码，直接复制即可：

  git remote add origin <https://github.com/xuexuehan/first_spring_mybatis.git
- **git push -u origin master -f**

39、说一说**简单用户界面登录过程**都需要做哪些分析?

40、如何对淘宝搜索框进行测试?

41、Servlet 的生命周期?



在一个Servlet的生命期中

- init()调用一次，一般第一次访问Servlet时调用
  - <load-on-startup>1</load-on-startup> 0 ,1,2,3
- service()调用多次，每次访问时调用
- destroy()调用一次，应用程序关闭时调用

- 构造方法（实例化）与 init(初始化)方法是一起执行的，要么第一次访问时执行，要么是 web 程序启动时执行。当加上1时，两个方法在 web 应用程序启动时调用，多个调用顺序 0,1,2,3。
- init()调用一次，一般第一次访问 Servlet 时调用。 service()调用多次，每次访问时调用。也有可能调用 0 次。
- destroy()调用一次， 应用程序关闭时调用（服务器关了，应用程序一定关；程序关闭，服务器不一定关）

- 实例化，或称创建， new，创建对象，分配内存空间。由容器完成。
- 初始化，指的是调用 init 方法，在对象实例化后执行。完成初始 servlet 要使用的资源，给变量赋初值等工作。
- 服务，调用 service 方法。
- 销毁， destroy,释放初始化占用的资源。
- 不可用，垃圾回收车收集内存。

42、Session VS Cookie ?

比较内容	Session	Cookie
保存方式	数据内容保存在服务器端	数据内容保存在客户端
安全性	数据比较安全	数据相对不安全
生命周期	使用内存存放数据，当用户长时间未请求服务器或服务器重启，内容可能丢失	保存在客户端的内存或文件中，可以指定Cookie的生存周期
资源占用	占用服务器的内存	每次请求时发送Cookie内容，占用带宽
存放内容	可以存放各种数据类型的数据	只能存放字符串类型的数据

43、说一下Spring AOP？Spring AOP的原理？

AOP（Aspect Oriented Programming），即**面向切面编程**，可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过**OOP允许开发者定义纵向的关系，但并不适合定义横向的关系**，例如日志功能。**日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。**

AOP技术恰恰相反，它利用一种称为"横切"的技术，**剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。**

使用"横切"技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

原理：

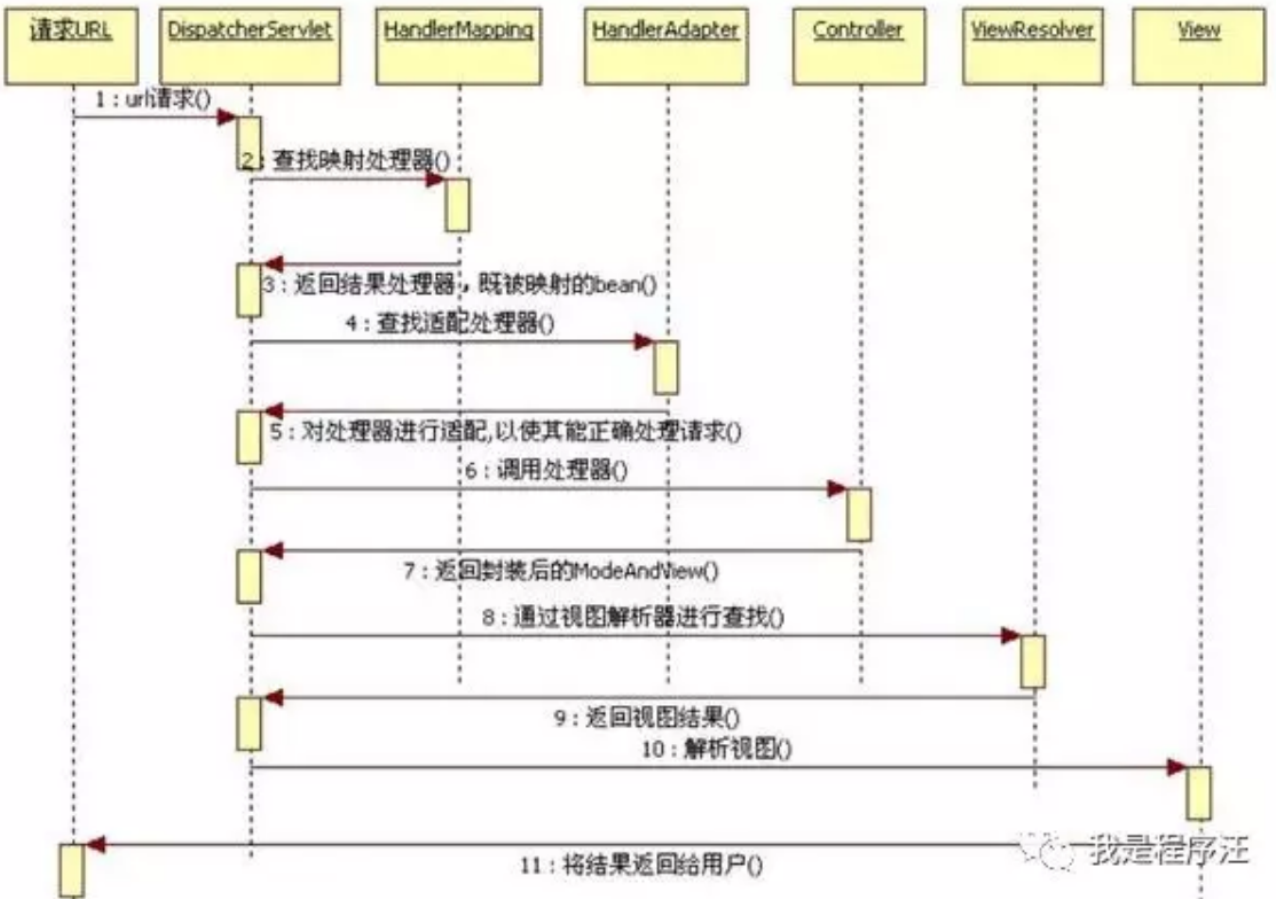
- 通过**预编译方式和运行期动态代理方式实现程序功能的统一维护**的一种技术；
- 主要功能：**日志记录、性能统计、安全控制、事务处理、异常处理**等等；
- AOP实现方式：
  - 预编译：AspectJ
  - 运行期动态代理（JDK动态代理、CGLib动态代理）：SpringAOP、JbossAOP
- 面向切面的核心思想就是，**让核心的业务逻辑代码，不需要去管理一些通用的逻辑，比如说事务，安全等这方面的共同逻辑，解耦业务逻辑和通用逻辑 。**

44、SpringMVC主流程？

核心类与接口：

- DispatcherServlet 前置控制器
- HandlerMapping 请求映射（到Controller）
- HandlerAdapter 请求映射（到Controller类的方法上）
- Controller 控制器

- HandlerIntercepter 拦截器
- ViewResolver 视图映射
- View 视图处理



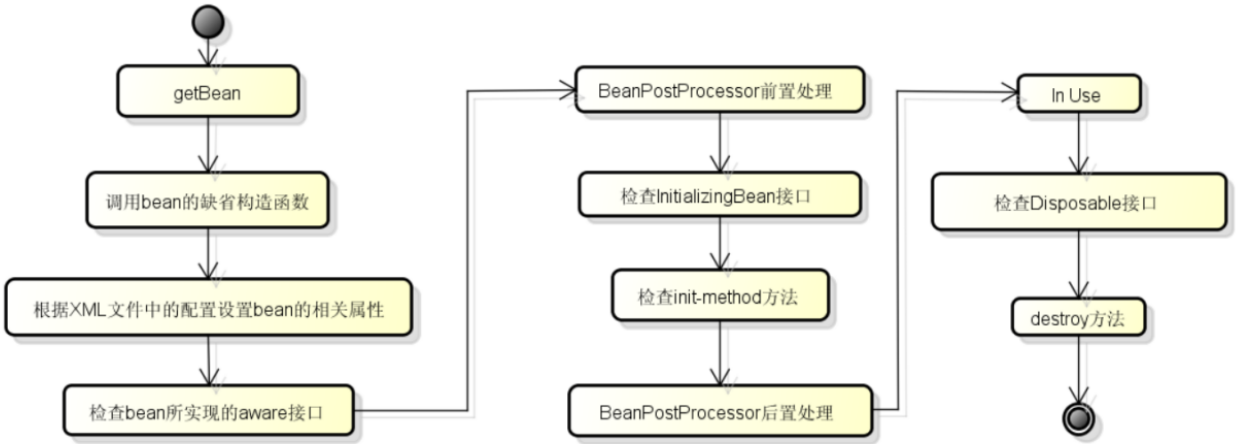
> - 客户端的所有请求都交给前端控制器DispatcherServlet来处理，它会负责调用系统的其他模块来真正处理用户的请求。
> - DispatcherServlet收到请求后，将根据请求的信息（包括URL、HTTP协议方法、请求头、请求参数、Cookie等）以及HandlerMapping的配置找到处理该请求的Handler（任何一个对象都可以作为请求的Handler）。
> - 在这个地方Spring会通过HandlerAdapter对该处理器进行封装。
> - HandlerAdapter是一个适配器，它用统一的接口对各种Handler中的方法进行调用。
> - Handler完成对用户请求的处理后，会返回一个ModelAndView对象给DispatcherServlet，ModelAndView顾名思义，包含了数据模型以及相应的视图的信息。
> - ModelAndView的视图是逻辑视图，DispatcherServlet还要借助ViewResolver完成从逻辑视图到真实视图对象的解析工作。
> - 当得到真正的视图对象后，DispatcherServlet会利用视图对象对模型数据进行渲染。
> - 客户端得到响应，可能是一个普通的HTML页面，也可以是XML或JSON字符串，还可以是一张图片或者一个PDF文件。

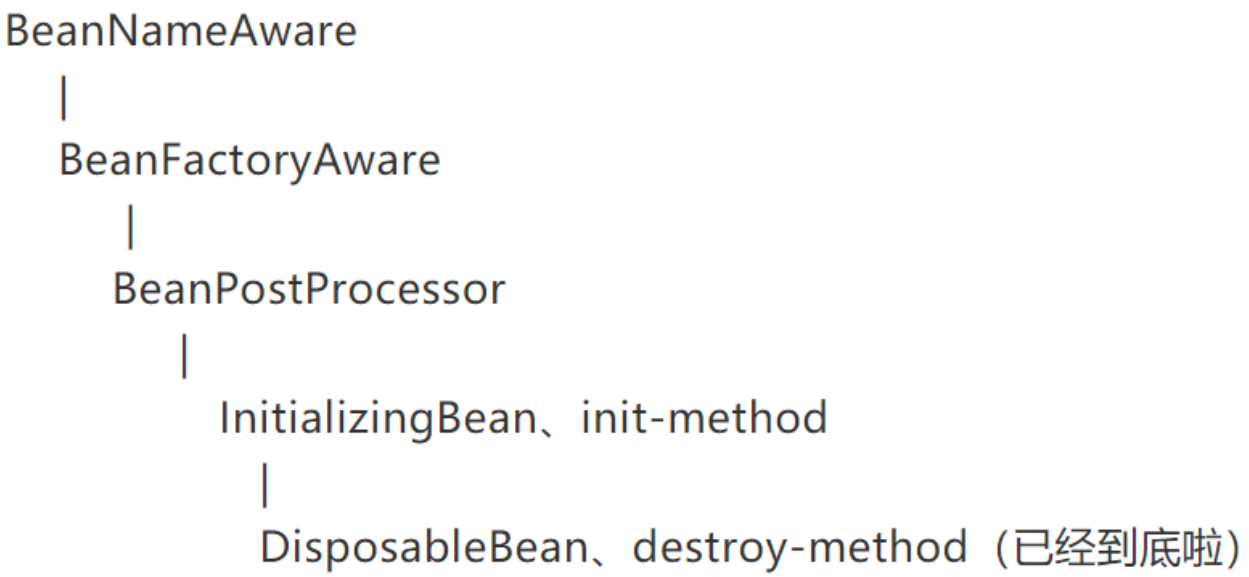
- 用户发送请求时会先从DispathcherServler的doService方法开始，在该方法中会将ApplicationContext、localeResolver、themeResolver等对象添加到request中，紧接着就是调用doDispatch方法。
- 进入该方法后首先会检查该请求是否是文件上传的请求(校验的规则是是否是post并且contenttType是否为multipart/为前缀)即调用的是checkMultipart方法；如果是的将request包装成MultipartHttpServletRequest。
- 然后调用getHandler方法来匹配每个HandlerMapping对象，如果匹配成功会返回这个Handle的处理链HandlerExecutionChain对象，在获取该对象的内部其实也获取我们自定定义的拦截器，并执行了其中的方

法。

- 执行拦截器的preHandle方法，如果返回false执行afterCompletion方法并理解返回
- 通过上述获取到了HandlerExecutionChain对象，通过该对象的getHandler()方法获得一个object通过HandlerAdapter进行封装得到HandlerAdapter对象。
- 该对象调用handle方法来执行Controller中的方法，该对象如果返回一个ModelAndView给DispatcherServlet。
- DispatcherServlet借助ViewResolver完成逻辑试图名到真实视图对象的解析，得到View后DispatcherServlet使用这个View对ModelAndView中的模型数据进行视图渲染。

45、spring bean生命周期？（这个问题其实是考察你对spring接口的一个熟悉程度，回答不全不要慌，能说几个是几个）





下面以BeanFactory为例，说明一个Bean的生命周期活动：

> - Bean的建立， 由BeanFactory读取Bean定义文件，并生成各个实例
> - Setter注入，执行Bean的属性依赖注入
> - BeanNameAware的setBeanName(), 如果实现该接口，则执行其setBeanName方法
> - BeanFactoryAware的setBeanFactory()，如果实现该接口，则执行其setBeanFactory方法
> - BeanPostProcessor的processBeforeInitialization()，如果有关联的processor，则在Bean初始化之前都会执行这个实例的processBeforeInitialization()方法
> - InitializingBean的afterPropertiesSet()，如果实现了该接口，则执行其afterPropertiesSet()方法

> - Bean定义文件中定义init-method
> - BeanPostProcessors的processAfterInitialization()，如果有关联的processor，则在Bean初始化之前都会执行这个实例的processAfterInitialization()方法
> - DisposableBean的destroy()，在容器关闭时，如果Bean类实现了该接口，则执行它的destroy()方法
> - Bean定义文件中定义destroy-method，在容器关闭时，可以在Bean定义文件中使用“destory-method”定义的方法

> - Spring容器 从XML 文件中读取bean的定义，并实例化bean。
> - Spring根据bean的定义填充所有的属性。
> - 如果bean实现了BeanNameAware 接口，Spring 传递bean 的ID 到 setBeanName方法。
> - 如果Bean 实现了 BeanFactoryAware 接口， Spring传递beanfactory 给setBeanFactory 方法。
> - 如果有任何与bean相关联的BeanPostProcessors，Spring会在postProcesserBeforeInitialization()方法内调用它们。
> - 如果bean实现IntializingBean了，调用它的afterPropertySet方法，如果bean声明了初始化方法，调用此初始化方法。
> - 如果有BeanPostProcessors 和bean 关联，这些bean的postProcessAfterInitialization() 方法将被调用。
> - 如果bean实现了 DisposableBean，它将调用destroy()方法。

46、说明一下springmvc和spring-boot区别是什么?

总的来说，Spring 就像一个大家族，有众多衍生产品例如 Boot，Security，JPA等等。但他们的基础都是Spring 的 IOC 和 AOP，IOC提供了依赖注入的容器，而AOP解决了面向切面的编程，然后在此两者的基础上实现了其他衍生产品的高级功能；因为 Spring 的配置非常复杂，各种xml，properties处理起来比较繁琐。于是为了简化开发者的使用，Spring社区创造性地推出了Spring Boot，它遵循约定优于配置，极大降低了Spring使用门槛，但又不失Spring原本灵活强大的功能。

47、什么是IoC和DI？并且简要说明一下DI是如何实现的?

IoC叫控制反转，是Inversion of Control的缩写，**DI（Dependency Injection）叫依赖注入，是对IoC更简单的诠释。**控制反转是把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的**"控制反转"就是对组件对象控制权的转移，从程序代码本身转移到了外部容器，由容器来创建对象并管理对象之间的依赖关系。**IoC体现了好莱坞原则 - "Don’t call me, we will call you"。依赖注入的基本原则是应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由容器负责，查找资源的逻辑应该从应用组件的代码中抽取出来，交给容器来完成。DI是对IoC更准确的描述，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

一个类A需要用到接口B中的方法，那么就需要为类A和接口B建立关联或依赖关系，最原始的方法是在类A中创建一个接口B的实现类C的实例，但这种方法需要开发人员自行维护二者的依赖关系，也就是说当依赖关系发生变动的时候需要修改代码并重新构建整个系统。如果通过一个容器来管理这些对象以及对象的依赖关系，则只需要在类A中定义好用于关联接口B的方法（构造器或setter方法），将类A和接口B的实现类C放入容器中，通过对容器的配置来实现二者的关联。 依赖注入可以通过setter方法注入（设值注入）、构造器注入和接口注入三种方式来实现，Spring支持setter注入和构造器注入，通常使用构造器注入来注入必须的依赖关系，对于可选的依赖关系，则setter注入是更好的选择，setter注入需要类提供无参构造器或者无参的静态工厂方法来创建对象。

48、简要说明一下IOC和AOP是什么?

依赖注入的三种方式：（1）**接口注入**（2）**Construct注入**（3）**Setter注入**

控制反转（IoC）与依赖注入（DI）是同一个概念，引入IOC的目的：（1）脱开、降低类之间的耦合；（2）倡导面向接口编程、实施依赖倒换原则；（3）提高系统可插入、可测试、可修改等特性。

具体做法：（1）将bean之间的依赖关系尽可能地抓换为关联关系；

（2）将对具体类的关联尽可能地转换为对Java interface的关联，而不是与具体的服务对象相关联；

（3）Bean实例具体关联相关Java interface的哪个实现类的实例，在配置信息的元数据中描述；

（4）由IoC组件（或称容器）根据配置信息，实例化具体bean类、将bean之间的依赖关系注入进来。

AOP（Aspect Oriented Programming），即面向切面编程，可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过**OOP允许开发者定义纵向的关系，但并不适合定义横向的关系，例如日志功能。**日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP技术恰恰相反，它利用一种称为"横切"的技术，**剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块**，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

使用"横切"技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

49、aop的应用场景有哪些?

Authentication 权限， Caching 缓存， Context passing 内容传递， Error handling 错误处理， Lazy loading 懒加载， Debugging 调试， logging, tracing, profiling and monitoring 记录跟踪　优化　校准，Performance optimization 性能优化， Persistence 持久化， Resource pooling 资源池， Synchronization同步，Transactions 事务。

50、Spring框架为企业级开发带来的好处有哪些?

> - 非侵入式：支持基于POJO的编程模式，不强制性的要求实现Spring框架中的接口或继承Spring框架中的类。
> - IoC容器：IoC容器帮助应用程序管理对象以及对象之间的依赖关系，对象之间的依赖关系如果发生了改变只需要修改配置文件而不是修改代码，因为代码的修改可能意味着项目的重新构建和完整的回归测试。有了IoC容器，程序员再也不需要自己编写工厂、单例，这一点特别符合Spring的精神"不要重复的发明轮子"。
> - AOP（面向切面编程）：将所有的横切关注功能封装到切面（aspect）中，通过配置的方式将横切关注功能动态添加到目标代码上，进一步实现了业务逻辑和系统服务之间的分离。另一方面，有了AOP程序员可以省去很多自己写代理类的工作。
> - MVC：Spring的MVC框架为Web表示层提供了更好的解决方案。
> - 事务管理：Spring以宽广的胸怀接纳多种持久层技术，并且为其提供了声明式的事务管理，在不需要任何一行代码的情况下就能够完成事务管理。
> - 其他：选择Spring框架的原因还远不止于此，Spring为Java企业级开发提供了一站式选择，你可以在需要的时候使用它的部分和全部，更重要的是，甚至可以在感觉不到Spring存在的情况下，在你的项目中使用Spring提供的各种优秀的功能。

51、Spring 框架中都用到了哪些设计模式?

Spring 框架中使用到了大量的设计模式，下面列举了比较有代表性的：1、代理模式—在 AOP 和 remoting 中被用的比较多。2、单例模式：在 spring 配置文件中定义的 bean 默认为单例模式。3、模板模式：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。4、委派模式：Spring 提供了 DispatcherServlet 来对请求进行分发。5、工厂模式：BeanFactory 用来创建对象的实例，贯穿于 BeanFactory / ApplicationContext接口的核心理念。6、代理模式：AOP 思想的底层实现技术，Spring 中采用 JDK Proxy 和 CgLib 类库。