9/14 度小满

1、代码最长上升子序列？

```
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];
}
```

2、最长公共子序列?

```
public 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; i++) {
        for (int j = 1; j <= n2; 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];
}
```

3、两个栈实现一个队列?

```
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();
    }
}
```

4、三次握手？为什么是三次？可以去掉一次吗？

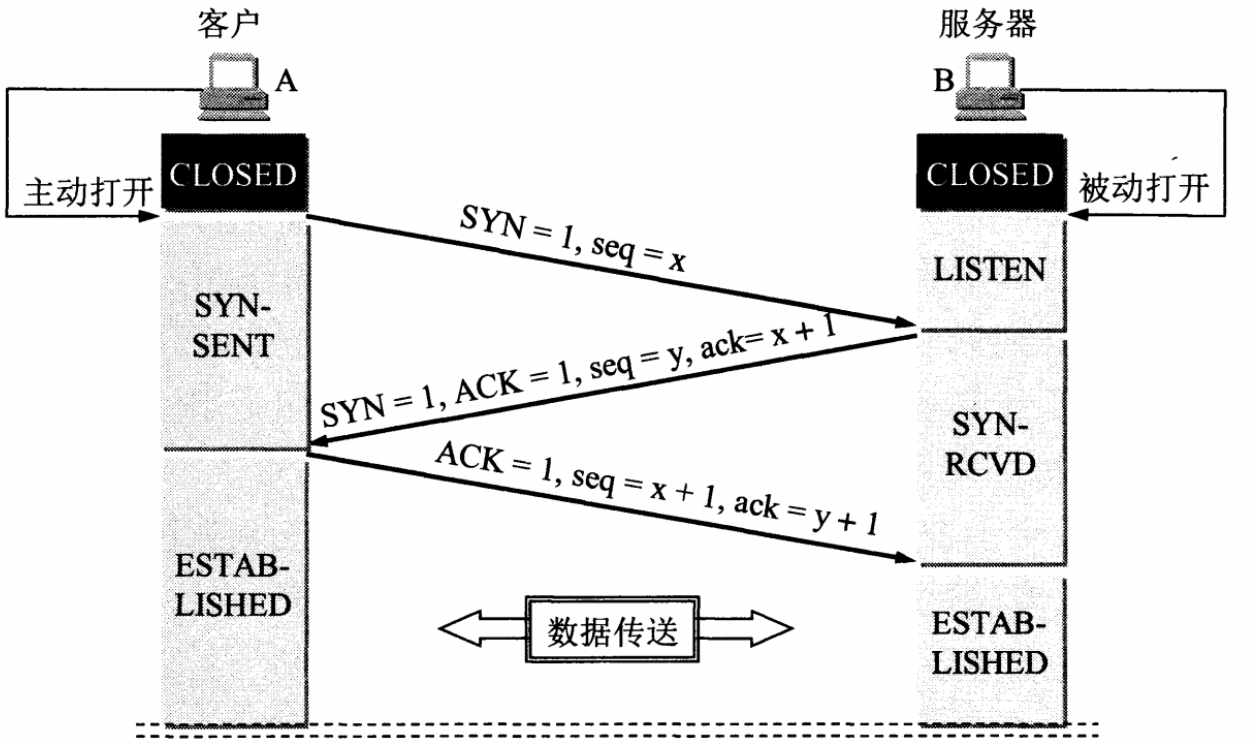


图 5-28 用三报文握手建立 TCP 连接

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。

5、tcp长连接 短链接区别？

- tcp短链接：模拟一下TCP短连接的情况，client向server发起连接请求，server接到请求，然后双方建立连接。client向server发送消息，server回应client，然后一次读写就完成了，这时候双方任何一个都可以发起close操作，不过一般都是client先发起close操作。为什么呢，一般的server不会回复完client后立即关闭连接的，当然不排除有特殊的情况。从上面的描述看，短连接一般只会在client/server间传递一次读写操作

  短连接的优点是：管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段。

- tcp长链接：模拟一下长连接的情况，client向server发起连接，server接受client连接，双方建立连接。Client与server完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

6、怎样保持长连接？

- TCP长连接保持：KeepAlive。TCP协议的实现里有一个KeepAlive机制，自动检测能否和对方连通并保持连接。
- TCP识别不同的请求：每个连接建立时，都会保存一个唯一的套接字，有了这个套接字，你就知道对方的IP地址、端口号等信息。这样，通过这个套接字，就可以向指定方发送信息了。

7、面对对象和面对过程的区别？

- 都可以实现代码重用和模块化编程，但是面向对象的模块化更深，数据更封闭，也更安全！因为面向对象的封装性更强！
- 面向对象的思维方式更加贴近于现实生活，更容易解决大型的复杂的业务逻辑
- 从前期开发角度上来看，面向对象远比面向过程要复杂，但是从维护和扩展功能的角度上来看，面向对象远比面向过程要简单！

8、测试方法有哪些?

1）按是否查看程序内部结构分为：

- 黑盒测试（black-box testing）：只关心输入和输出的结果
- 白盒测试（white-box testing）：去研究里面的源代码和程序结构

2）按是否运行程序分为：

- 静态测试（static testing）：是指不实际运行被测软件，而只是静态地检查程序代码、界面或文档可能存在的错误的过程。

  静态测试包括：

  - 对于代码测试，主要是测试代码是否符合相应的标准和规范。
  - 对于界面测试，主要测试软件的实际界面与需求中的说明是否相符。
  - 对于文档测试，主要测试用户手册和需求说明是否真正符合用户的实际需求。

- 动态测试（dynamic testing），是指实际运行被测程序，输入相应的测试数据，检查输出结果和预期结果是否相符的过程

3）按阶段划分：

- 单元测试（unit testing），是指对软件中的最小可测试单元进行检查和验证。

  桩模块（stud）是指模拟被测模块所调用的模块，驱动模块（driver）是指模拟被测模块的上级模块，驱动模块用来接收测试数据，启动被测模块并输出结果。

- 集成测试（integration testing），是单元测试的下一阶段，是指将通过测试的单元模块组装成系统或子系统，再进行测试，重点测试不同模块的接口部门。

  集成测试就是用来检查各个单元模块结合到一起能否协同配合，正常运行。

- 系统测试（system testing），指的是将整个软件系统看做一个整体进行测试，包括对功能、性能，以及软件所运行的软硬件环境进行测试。

  系统测试的主要依据是《系统需求规格说明书》文档。

- 验收测试（acceptance testing），指的是在系统测试的后期，以用户测试为主，或有测试人员等质量保障人员共同参与的测试，它也是软件正式交给用户使用的最后一道工序。

  验收测试又分为a测试和beta测试，其中a测试指的是由用户、 测试人员、开发人员等共同参与的内部测试，而beta测试指的是内测后的公测，即完全交给最终用户测试。

4）黑盒测试分为功能测试和性能测试：

- 功能测试（function testing），是黑盒测试的一方面，它检查实际软件的功能是否符合用户的需求。
  - 包括逻辑功能测试（logic function testing）
  - 界面测试（UI testing）UI=User Interface
  - 易用性测试（usability testing）：是指从软件使用的合理性和方便性等角度对软件系统进行检查，来发现软件中不方便用户使用的地方。
  - 兼容性测试（compatibility testing）：包括硬件兼容性测试和软件兼容性测试
- 性能测试（performance testing）

  软件的性能主要有时间性能和空间性能两种： 时间性能：主要指软件的一个具体事务的响应时间（respond time）。 空间性能：主要指软件运行时所消耗的系统资源。

软件性能测试分为：

- 一般性能测试：指的是让被测系统在正常的软硬件环境下运行，不向其施加任何压力的性能测试。
- 稳定性测试也叫可靠性测试（reliability testing）：是指连续运行被测系统检查系统运行时的稳定程度。
- 负载测试（load testing）：是指让被测系统在其能忍受的压力的极限范围之内连续运行，来测试系统的稳定性。
- 压力测试（stress testing）：是指持续不断的给被测系统增加压力，直到将被测系统压垮为止，用来测试系统所能承受的最大压力。(Validate the system or software can allowed the biggest stress.)

9、白盒和黑盒有哪些测试方法？

- 常用的黑盒测试方法有：等价类划分法；边界值分析法；因果图法；场景法；正交实验设计法；判定表驱动分析法；错误推测法；功能图分析法。
- 常用白盒测试方法：
  - 静态测试：不用运行程序的测试，包括代码检查、静态结构分析、代码质量度量、文档测试等等，它可以由人工进行，充分发挥人的逻辑思维优势，也可以借助软件工具（Fxcop）自动进行。
  - 动态测试：需要执行代码，通过运行程序找到问题，包括功能确认与接口测试、覆盖率分析、性能分析、内存分析等。

白盒测试中的逻辑覆盖包括语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖和路径覆盖。

10、sql怎么添加一列， 添加索引？

- mysql添加列： alter table tableName add columnName varchar(30)；
- 添加索引：ALTER TABLE `table_name` ADD INDEX index_name (`column`)

11、快排？

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

```java
//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;
    }
}
```

12、访问一个URL流程？

- 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。

13、四次挥手及原因？

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：

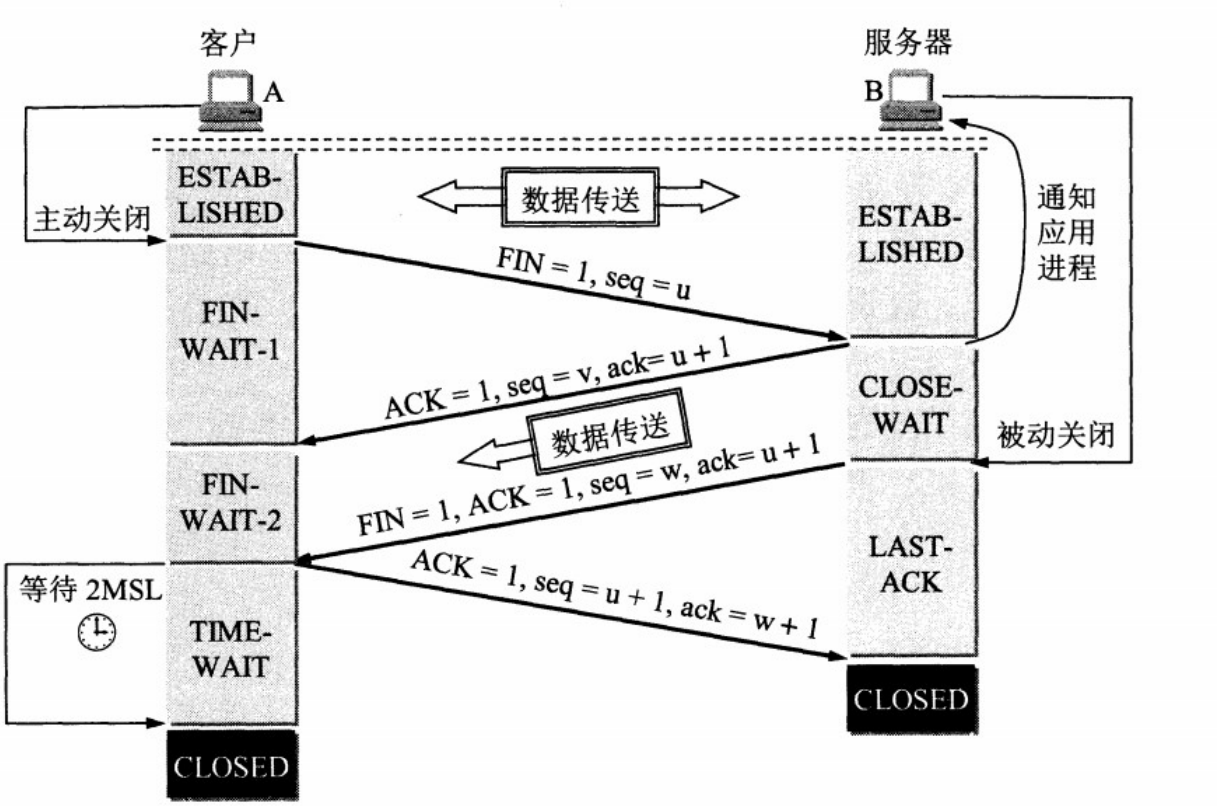


图 5-29　TCP 连接释放的过程

（1）客户端A发送一个FIN，用来关闭客户A到服务器B的数据传送。

（2）服务器B收到这个FIN，它发回一个ACK，确认序号为收到的序号加1。和SYN一样，一个FIN将占用一个序号。

（3）服务器B关闭与客户端A的连接，发送一个FIN给客户端A。

（4）客户端A发回ACK报文确认，并将确认序号设置为收到序号加1。

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。

14、int integer区别？

int 是基本类型，直接存数值，而integer是对象，用一个引用指向这个对象

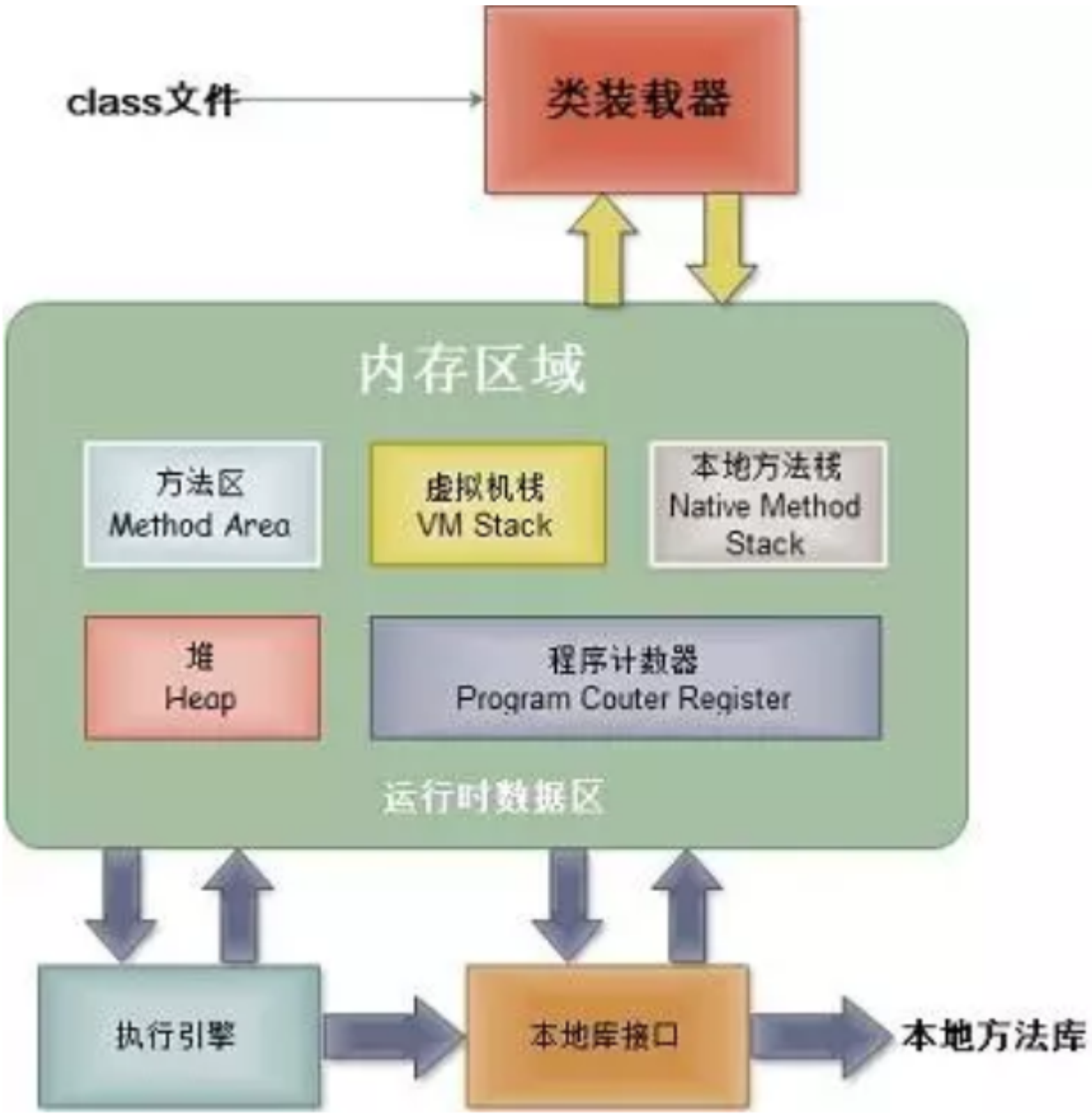
- int是基本的数据类型；
- Integer是int的封装类；
- int和Integer都可以表示某一个数值；
- int和Integer不能够互用，因为他们两种不同的数据类型；

15、java八大基础类型？

byte short int long floate double char boolean

16、jvm内存分配？

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。**

内存泄漏与内存溢出?

- 内存泄漏（memory leak）：是指**程序在申请内存后，无法释放已申请的内存空间**，一次内存泄漏似乎不会有大的影响，但内存泄漏堆积后的后果就是内存溢出。

  > 常见的内存泄漏：
  >
  > - 单例造成内存泄漏：由于单例的静态特性使得其生命周期和应用的生命周期一样长，如果一个对象已经不再需要使用了，而单例对象还持有该对象的引用，就会使得该对象不能被正常回收，从而导致了内存泄漏。
  > - 资源未关闭造成的内存泄漏：对于使用了File，Stream等资源，应该在Activity销毁时及时关闭或者注销，否则这些资源将不会被回收，从而造成内存泄漏。

- 内存溢出 （out of memory）：指**程序申请内存时，没有足够的内存供申请者使用**，或者说，给了你一块存储int类型数据的存储空间，但是你却存储long类型的数据，那么结果就是内存不够用，此时就会报错OOM，即所谓的内存溢出。

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。**

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。**

17、单例模式？

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;
    }
}
```

18、深度 广度优先遍历？

- 深度优先遍历：对于一颗二叉树，深度优先搜索(Depth First Search)是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。 先序、中序、后序。
- 广度优先遍历（层次遍历）：从树的root开始，从上到下从从左到右遍历整个树的节点

19、测试微信红包用例？

> 单个红包的情况
>
> 单个金额输入：红包上限、小数点位数限制、数值为0情况、多位小数点
>
> 留言
>
> 群发红包的情况
>
> 设置个数
>
> 抢红包记录
>
> 安卓用户和苹果用户互相之间的转换
>
> 是否可以多次抢一个红包
>
> 发红包者是否可以抢红包
>
> 未领取的红包24小时退回

20、TCP和UDP的区别？

- **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**

21、http和https的区别？

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。**

22、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。**

23、对称加密和非对称加密？

- **对称加密：指的就是加、解密使用的同是一串密钥**，所以被称做对称加密。对称加密只有一个密钥作为私钥。
- **非对称加密：指的是加、解密使用不同的密钥**，一把作为公开的公钥，另一把作为私钥。公钥加密的信息，只有私钥才能解密。反之，私钥加密的信息，只有公钥才能解密。

24、判断单链表是否有环？

```
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;
}
```

25、讲下Spring中用到的设计模式？

- **Spring IOC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的** 。Spring使用工厂模式可以通

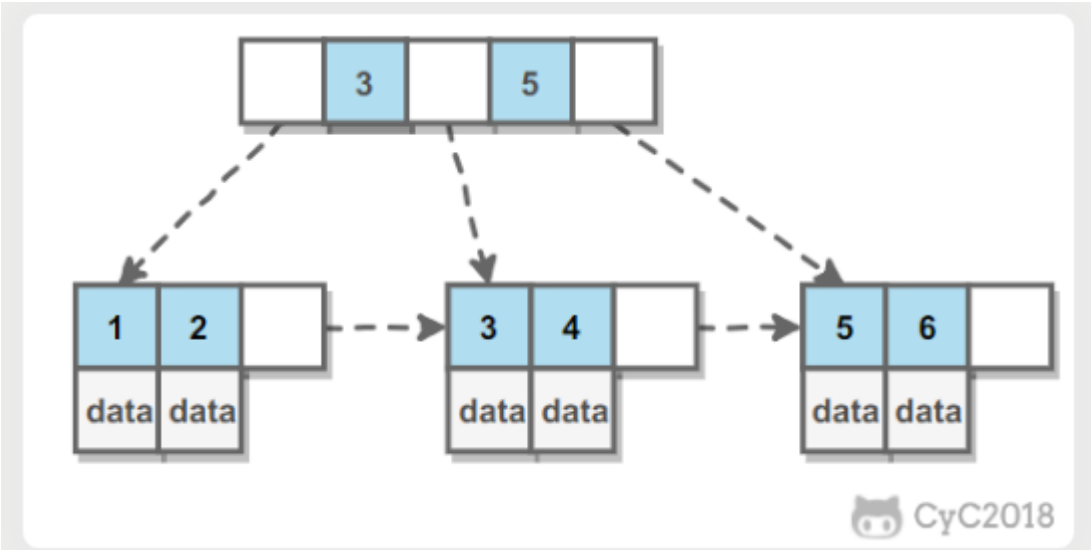
过 `BeanFactory` 或 `ApplicationContext` 创建 bean 对象。

- 单例模式：在我们的系统中，有一些对象其实我们只需要一个，比如说：线程池、缓存、对话框、注册表、日志对象、充当打印机、显卡等设备驱动程序的对象。事实上，这一类对象只能有一个实例，如果制造出多个实例就可能会导致一些问题的产生，比如：程序的行为异常、资源使用过量、或者不一致性的结果。
- 模板模式：Spring 中 `jdbcTemplate`、`hibernateTemplate` 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。 定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。 模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤的实现方式。

26、讲下索引结构？

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。

27、反射了解吗？

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；

28、说一下多态?

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。**

29、 线程和进程的区别？

- 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。

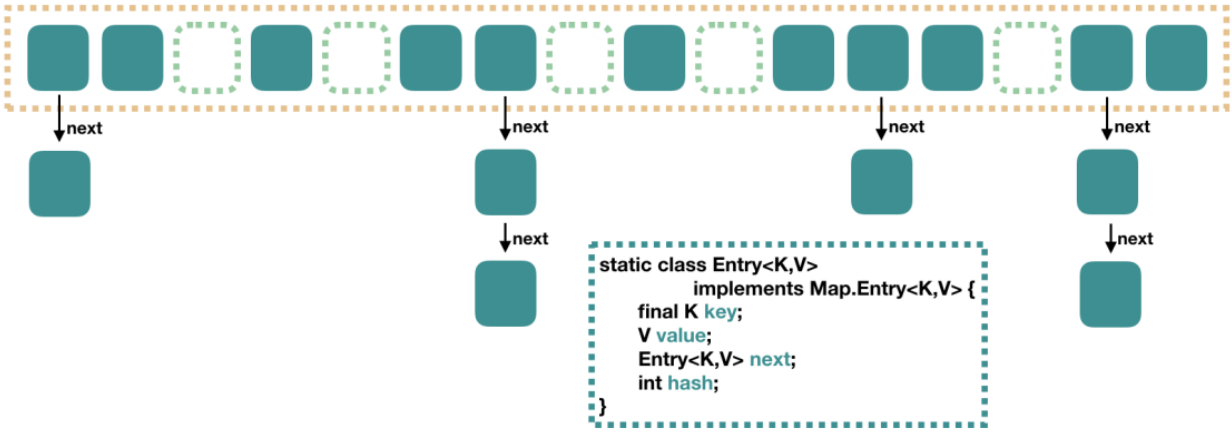
- 通信方面：**线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助 IPC。** 进程间通信（**IPC**，Inter-Process Communication）。

30、hashmap和concurrenthashmap的区别？

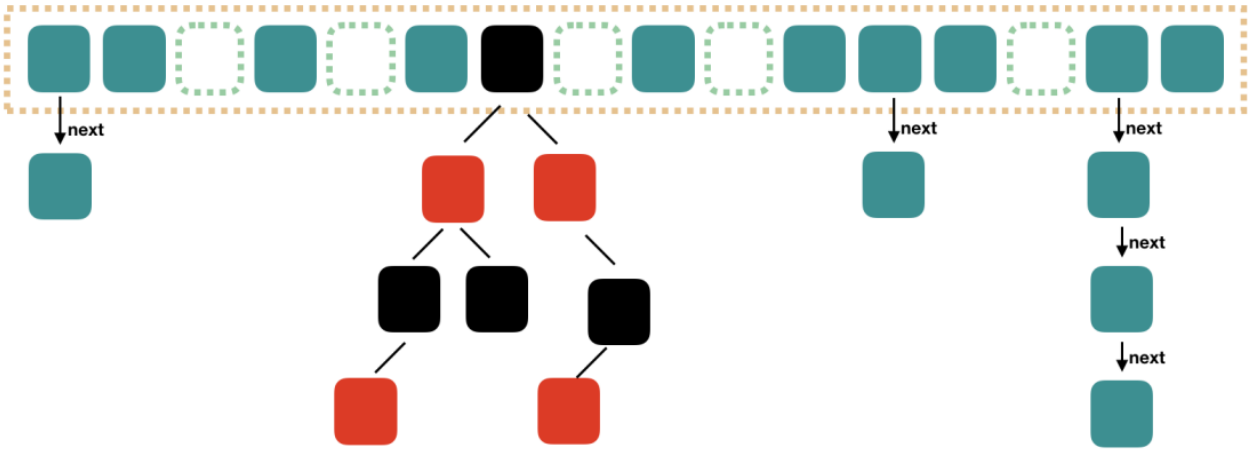
1）HashMap：**允许键、值为null；基于数组+链表实现；非线程安全。**在HashMap**进行扩容重哈希时导致Entry链形成环。一旦Entry链中有环，会导致在同一个桶中进行插入、查询、删除等操作时陷入死循环。**

- 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。

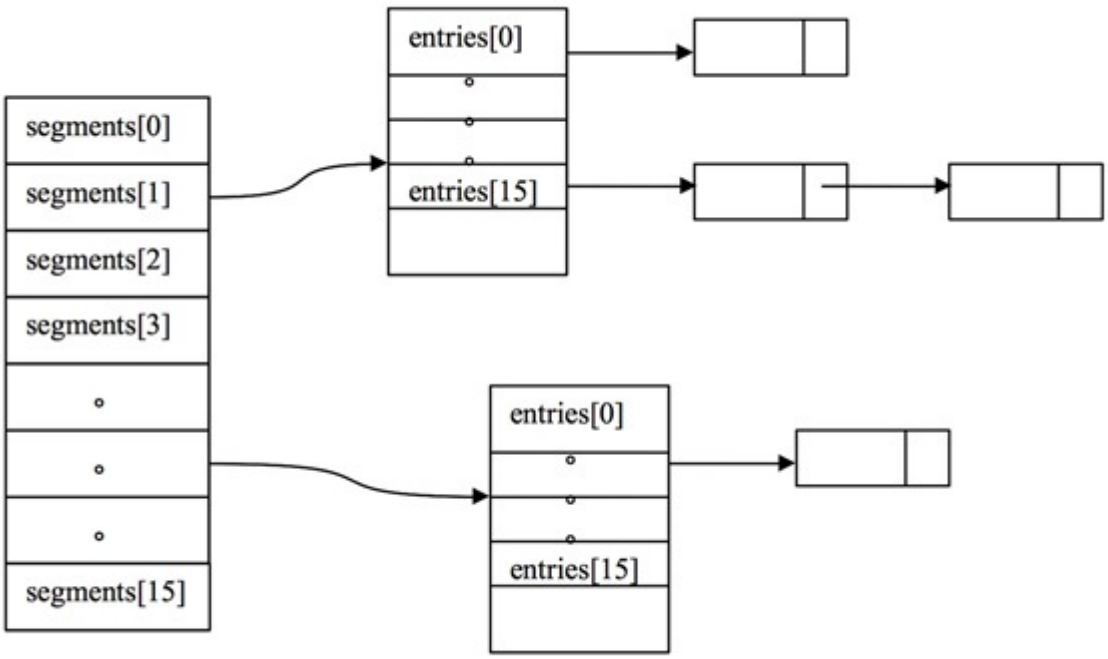
- get：根据指定的key值返回对应的value，根据key计算出hashcode，定位到桶的下标，如果桶是一个链表，依次遍历冲突链表，通过key.equals（k）方法来判断是否是要找的那个entry。如果桶不是链表，根据key是否相等返回值。

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）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：

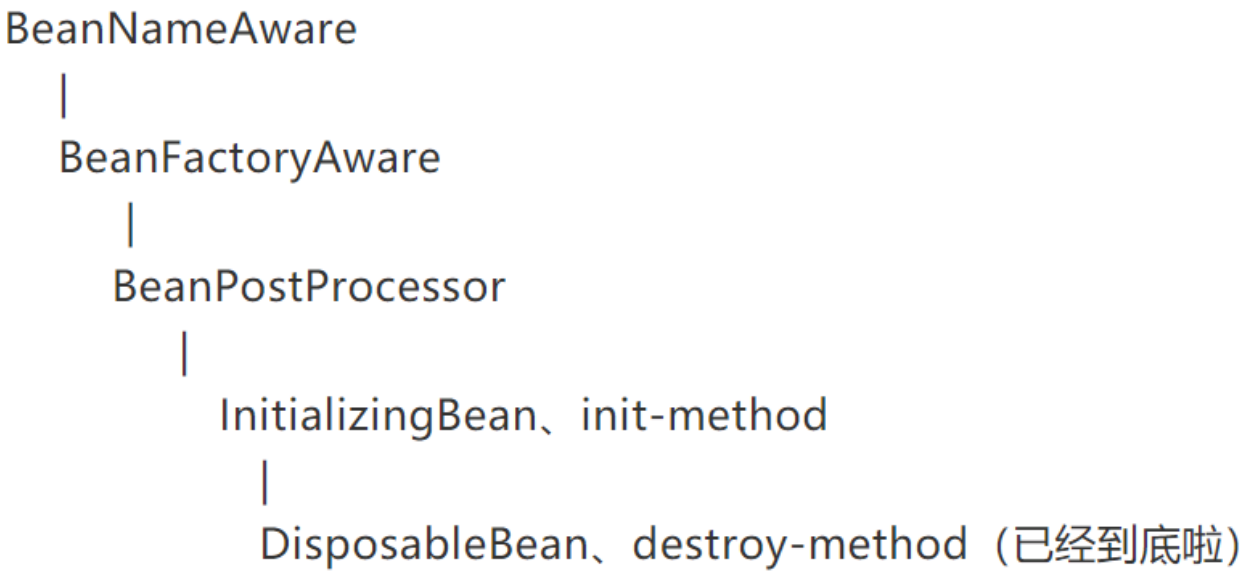
自旋锁和阻塞锁区别：要不要放弃处理器的执行时间。**对于阻塞锁和自旋锁来说，都是要等待获得共享资源**，但是阻塞锁是放弃了cpu时间，进入了等待区，等待被唤醒。而自旋锁一直“自旋”在那里，时刻的检查共享资源是否可以被访问。

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。

31、synchronized和lock的区别？

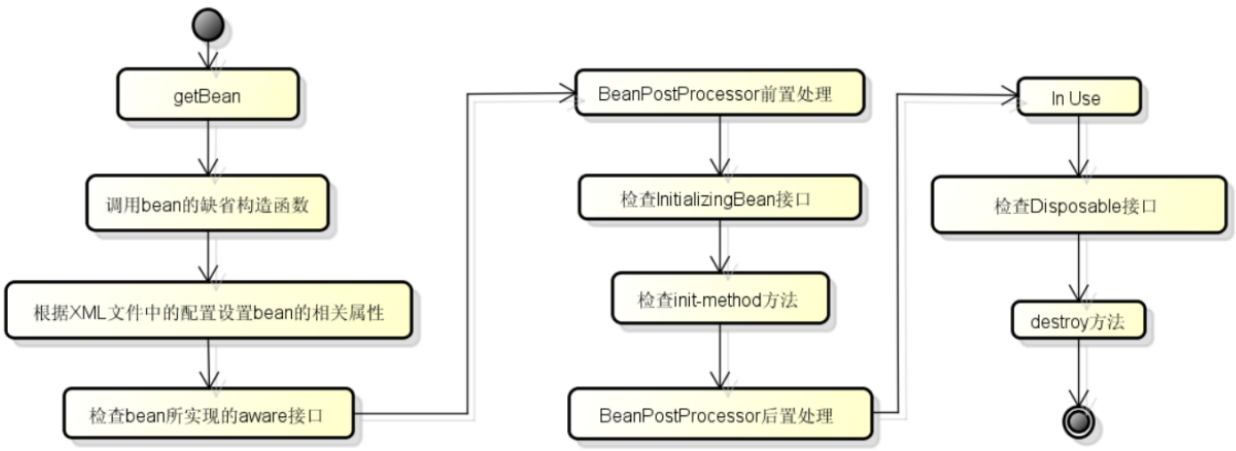
类别	synchronized	Lock
存在层次	Java的关键字，在jvm层面上	是一个类
锁的释放	1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm会让线程释放锁	在finally中必须释放锁，不然容易造成线程死锁
锁的获取	假设A线程获得锁，B线程等待。如果A线程阻塞，B线程会一直等待	分情况而定，Lock有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待
锁状态	无法判断	可以判断
锁类型	可重入 不可中断 非公平	可重入 可判断 可公平（两者皆可）
性能	少量同步	大量同步

32、说一下spring bean的生命周期？



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



33、Linux 查看端口占用?

可以使用 **lsof** 和 **netstat** 命令 。

```
#查看服务器 8000 端口的占用情况
lsof -i:8000
```

34、求连续子序列最大和？

```
public int maxSubArray(int[] nums) {
    if(nums == null || nums.length == 0) {
        return 0;
    }
    int preSum = nums[0];
    int maxSum = preSum;
    for(int i = 1; i < nums.length; i++) {
        if(preSum < 0) {
            preSum = nums[i];
        } else {
            preSum = preSum + nums[i];
        }
        maxSum = Math.max(maxSum, preSum);
    }
    return maxSum;
}
```

35、死锁四个必要条件？

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。

36、get和post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。