3.15 电话一面 Java研发：

• 自我介绍

• 项目介绍与遇到的难点

• ArrayList vs LinkedList

• HashMap原理

• JDK 1.8对HashMap的改进

• 多线程间协作与通信方式

一、synchronized+notify+wait+flag

这种方案是通过**对两个线程中分别要执行的方法加锁synchronized**，保证每次执行main时不被sub打断，执行sub循环时，不被main打断。

这里采用了对象object的notify和wait来实现线程之间的通信。当main方法执行完成后，让执行main方法的线程等待，等待sub方法执行完成后，通知(notify)main线程然后继续执行。这种方式有一个缺点，由于notify和wait使用的是Object的方法，所以不能单独的让某个特定的线程收到通知或者让他等待，而**在存在多个线程同时等待时，只能通过notifyAll来通知所有的线程。不够灵活**。

二、lock+condition+flag

这种方式是利用了Java5中提供的lock和condition，利用共享变量flag来实现线程之间的相互通信。同时在这个小例子中，相比上一个例子中增加了一个线程的循环。这是为了体现使用condition的优点。

使用condition可以非常灵活的去控制线程与线程之间的通信。因为在一个类中可以创建多个condition的实例，

我们可以通过condition不同的实例的signal和await方法来标识不同的两个线程之间相互通信的标识，而不是统一使用object的notify和wait方法了。

同时利用lock方法可以利用锁的重入机制实现更加灵活的锁的应用。可以在需要的时候加锁或解锁。

这样我们就可以实现多个线程之间的协调通信了。

三、semaphere+flag

这里semaphere代表一个信号量，它可以指示共享资源的个数，也就是同时访问资源的线程个数。这里主要通过semaphere的acquire和release实现锁的功能从而实现线程之间的通信。

利用semaphere不仅可以实现多个线程协调循环通信，在必要时还可以控制同一时间访问资源的个数。更加的灵活和方便。

以上是实习多个线程之间相互协调通信的几种方案。

• Synchronized vs Lock

• ReentrantLock优点

• CAS

• 自旋锁

• 类加载过程

• 内存泄漏与内存溢出，及其原因

• GC原理及优化策略

• Java的Exception类型

• 数据库四种隔离级别

• Spring框架介绍

• OOP设计原则

操作系统面试题

1、**进程和线程的区别**？

解析：（1）进程是资源的分配和调度的一个独立单元，而线程是CPU调度的基本单元

（2）同一个进程中可以包括多个线程，并且线程共享整个进程的资源（寄存器、堆栈、上下文），一个进行至少包括一个线程。

（3）进程的创建调用fork或者vfork，而线程的创建调用pthread\_create，进程结束后它拥有的所有线程都将销毁，而线程的结束不会影响同个进程中的其他线程的结束

（4）线程是轻两级的进程，它的创建和销毁所需要的时间比进程小很多，所有操作系统中的执行功能都是创建线程去完成的

（5）线程中执行时一般都要进行同步和互斥，因为他们共享同一进程的所有资源

（6）线程有自己的私有属性TCB，线程id，寄存器、硬件上下文，而进程也有自己的私有属性进程控制块PCB，这些私有属性是不被共享的，用来标示一个进程或一个线程的标志

2、**死锁？死锁产生的原因？死锁的必要条件？怎么处理死锁？**

解析：·相互等待资源而产生的一种僵持状态，如果没有外力的干预将一直持续这个状态

·系统资源不足、相互竞争资源、请求资源顺序不当

·互斥、不可抢占、循环等待、请求与保持

·因为互斥是不可改变的，所以只能破坏其他三个条件中的一个来解除死锁，方法：剥夺资源、杀死其中一个线程

**3、Windows内存管理方式：段存储、页存储、段页存储**

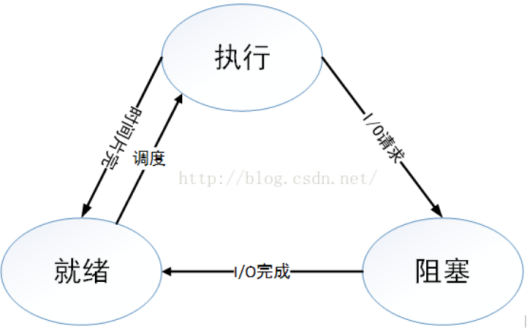
**4、进程的几种状态？**

解析：进程状态：一般来说，进程有三个状态，即就绪状态，运行状态，阻塞状态。

·运行态：进程占用CPU，并在CPU上运行；

·就绪态：进程已经具备运行条件，但是CPU还没有分配过来；

·阻塞态：进程因等待某件事发生而暂时不能运行；



（1）run（运行状态）：正在运行的进程或在等待队列中对待的进程，等待的进程只要以得到cpu就可以运行

（2）Sleep（可中断休眠状态）：相当于阻塞或在等待的状态

（3）D（不可中断休眠状态）：在磁盘上的进程

（4）T（停止状态）：这中状态无法直观的看见，因为是进程停止后就释放了资源，所以不会留在linux中

（5）Z（僵尸状态）：子进程先与父进程结束，但父进程没有调用wait或waitpid来回收子进程的资源，所以子进程就成了僵尸进程，如果父进程结束后任然没有回收子进程的资源，那么1号进程将回收

**5、IPC通信方式？**

解析：（1）管道（匿名管道（pipe亲缘关系的进程通信）、命名管道（mkfifo/mknod））

（2）消息队列：是基于消息的、用无亲缘关系的进程间通信，主要函数：msgget、msgsend、msgrecv、msgctl

（3）信号量：相当于一把互斥锁，通过p、v操作，主要函数：semget、semop、semctl

（4）共享内存：是进程间通信速度最快的，所以用经常是集合信号量或互斥锁来实现同步，shmget、shmat、shmdt、shmctl

**6、什么是虚拟内存？**

解析：是将进程部分装入内存中，从而能实现一个很大的程序能在一个比它小的内存中运行，它的主要实现是靠程序的换进换出来实现的，因为内存中0~3G是用户使用，3~4G才是内存使用，通过映射来实现来进行逻辑地址到物理地址的映射

**7、虚拟地址、逻辑地址、线性地址、物理地址的区别？**

解析： 分段机制把一个逻辑地址转换为线性地址；接着，分页机制把一个线性地址转换为物理地址。

（1）虚拟地址：虚拟内存映射出来的地址

（2）逻辑地址：程序的段加偏移量形成的，C/C++程序中取地址求出来的地址就是逻辑地址

（3）线性地址：是逻辑地址到物理地址的中间层，只有启动分页机制的时候才有线性地址，如果没有分页机制，那么线性地址就是物理地址

（4）物理地址：是内存中实实在在存在的硬件地址，

逻辑地址（启动分段）--》线性地址（启动分页）--》物理地址

**如何选择数据结构**

**--------------------------------------------------------------------------------------**

Array (T[])

•当元素的**数量是固定**的，并且需要使用**下标**时。

Linked list (LinkedList<T>)

•当元素需要**能够在列表的两端添加**时。否则使用 List<T>。

Resizable array list (List<T>)

•当元素的**数量不是固定**的，并且需要使用**下标**时。

Stack (Stack<T>)

•当需要实现 LIFO（Last In First Out）**后进先出**时。

Queue (Queue<T>)

•当需要实现 FIFO（First In First Out）**先进先出**时。

Hash table (Dictionary<K,T>)

•当需要使用**键值对**（Key-Value）**来快速添加和查找**，并且元素**没有特定的顺序**时。

Tree-based dictionary (SortedDictionary<K,T>)

•当需要使用价值对（Key-Value）来快速添加和查找，并且元素根据 **Key 来排序**时。

Hash table based set (HashSet<T>)

•当需要保存一组唯一的值，并且元素没有特定顺序时。

Tree based set (SortedSet<T>)

•当需要**保存一组唯一**的值，并且元素需**要排序**时。

ArrayList、Vector、LinkedList的区别及其优缺点？

**--------------------------------------------------------------------------------------**

ArrayList,LinkedList,Vestor这三个类都实现了java.util.List接口，但它们有各自不同的特性，主要如下：

一、同步性

ArrayList,LinkedList是不同步的，**而Vector是同步的**。所以如果不**要求线程安全**的话，可以使用ArrayList或 LinkedList，可以节省为同步而耗费的开销。但在**多线程的情况下**，有时候就不得不使用Vector了。

二、数据增长

从内部实现机制来讲**ArrayList和Vector都是**使用Objec的**数组形式来存储**的。当你向这两种类型中增加元素的时候，如果元素的数目超出了内部数组目前的长度它们都需要扩展内部数组的长度，**Vector缺省情况下自动增长原来一倍的数组长度**，**ArrayList是原来的50%**,所以最后你获得的这个集合所占的空间总是比你实际需要的要大。所以**如果你要在集合中保存大量的数据那么使用Vector有一些优势**，因为你可以通过设置集合的初始化大小来避免不必要的资源开销。

三、检索、插入、删除对象的效率

**ArrayList和Vector检索数据快，插入和删除慢。LinkedList插入和删除数据快，检索慢。**

ArrayList和Vector中，从指定的位置（用index）**检索**一个对象，或在**集合的末尾插入、删除**一个对象的时间是一样的，可表示为**O(1)**。 但是，如果在集合的其他位置**增加或移除元素**那么花费的时间会呈线形增长：**O(n-i)**，其中n代表集合中元素的个数，i代表元素增加或移除元素的索引位置。以为在进行上述操作的时候集合中第i和第i个元素之后的所有元素都要执行(n-i)个对象的**位移操作**。

LinkedList中，在**插入、删除**集合中任何位置的元素所花费的时间都是一样的—**O(1)**，但它在**索引**一个元素的时候**比较慢，为O(i)**,其中i是索引的位置。

ArrayList和LinkedList的大致区别：

1**.ArrayList**是实现了基于**动态数组**的数据结构，**LinkedList**基于**链表**的数据结构。

2.对于**随机访问**get和set，**ArrayList优于LinkedList**，因为LinkedList要移动指针。

3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

-----------------------------------------------------------------

ArrayList 和Vector他们底层的实现都是一样的，都是使用**数组方式存储数据**，此数组元素数大于实际存储的数据以便增加和插入元素，它们都**允许直接按序号索引元素**，但是插入元素要涉及数组元素移动等内存操作，所以**索引数据快而插入数据慢**。

Vector中的方法由于添加了synchronized修饰，因此Vector是线程安全的容器，但**性能上较ArrayList差**，因此已经是Java中的遗留容器。

LinkedList使用**双向链表实现存储**（将内存中零散的内存单元通过附加的引用关联起来，形成一个可以按序号索引的线性结构，这种链式存储方式与数组的连续存储方式相比，**内存的利用率更高**），按序号**索引数据需要进行前向或后向遍历**，但是**插入数据时只需要记录本项的前后项即可**，所以**插入速度较快**。

**一、**

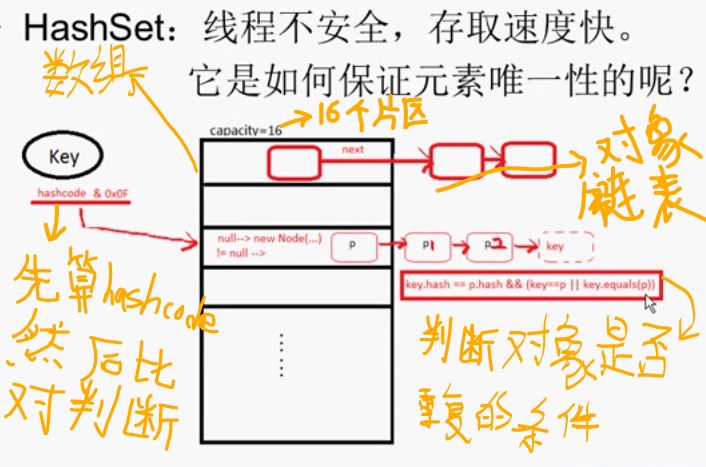
1. hashmap、concurrenthashmap底层实现和区别
2. **描述hashMap内部实现原理；hashSet和hashMap有没有区别。哈希的底层存储原理。**

**hashMap存的是key-value对，key值是唯一的，是不能重复的。**

**hashSet和hashMap本质上是没有区别的，hashSet的集合内部是通过hashMap的key部分实现的。从类的角度来说，HashSet实现的是Set接口，而Set接口继承的是Collection接口，hashMap实现的是Map接口，Collection和Map属于同级。从线程安全角度看他们都是不安全的，所以效率更高。**

**对于其存储机制，它们都用的是哈希，哈希的目的使查询速度更快。哈希就是散列，就是把对象在内存中打散，把对象分到若干个空间，当在搜寻的时候就可以直接定位到某一个空间，这样就缩小了搜寻范围。核心就是哈希码hashCode，底层实现原理是首先对hashCode进行处理和运算，得到一个容量区间的值，也就是在哪个区间上。**

**因为Map在底层存储哈希的时候是通过一个数组来实现的，它默认的容量capacity是16个，也就是16个区间。当它把hashCode拿出来之后会先做一个处理，目的是使它尽量发散，之后再做位运算，把对象定位到数组的一个元素上，也就是某个区间上，而这数组的每个元素存储的是对象链表，然后在这个元素上再进行判断，如果这个元素是空，就直接放进去，如果不为空，然后从链表的头开始与当前对象比对判断，从3个方面比较，先判断hashCode是否相同，如果不同，再判断二者是否同一对象（即内存地址==）和内容（equals()方法）是否相同，如果内存地址相同，则重复，不能添加，如果不是同一对象再比较equals()方法是否相同，相同则不能添加，如果不是重复对象，就继续与链表的下一个节点比对，直到最后一个节点如果还不同的话就可以添加进去，把这个对象添加在链表的末尾。它查找搜索是通过getKey方法，先算要查找对象key的哈希码，然后判断出它在哪个桶里，也就是在数组的哪个元素上。通过key直接判定出对象在哪个区间上。**



别人的hashMap总结：

HashMap采用hash算法来决定Map中key的存储，并通过hash算法来增加集合的大小。hash表里可以存储元素的位置称为桶，如果通过key计算hash值发生冲突时，那么将采用链表的形式，来存储元素。HashMap的扩容操作是一项很耗时的任务，所以如果能估算Map的容量，最好给它一个默认初始值，避免进行多次扩容。HashMap的线程是不安全的，多线程环境中推荐是ConcurrentHashMap。

**HashMap和ConcurrentHashMap的区别：**

**1 HashMap不是线程安全的，而ConcurrentHashMap是线程安全的。**

**2 ConcurrentHashMap采用锁分段技术，将整个Hash桶进行了分段segment，也就是将这个大的数组分成了几个小的片段segment，而且每个小的片段segment上面都有锁存在，那么在插入元素的时候就需要先找到应该插入到哪一个片段segment，然后再在这个片段上面进行插入，而且这里还需要获取segment锁。**

**3 ConcurrentHashMap让锁的粒度更精细一些，并发性能更好。**

**Map将实际数据存储在Entry类的数组中。**

**执行put方法时，根据key的hash值来计算放到table数组的下标，如果hash有相同的下标，则新put进去的元素放到Entry链的头部。**

**用户传入了初始容量和负载因子，这两个值是HashMap性能优化的关键，涉及到了HashMap的扩容问题。**

**HashMap的容量永远是2的倍数，如果传入的不是2的指数则被调整为大于传入值的最近的2的指数，例如如果传入130，则capacity计算后是256。**

**HashMap的遍历方法（两种）**

**for(Iterator it = map.entrySet().iterator();it.hasNext();){**

**Map.Entry e = (Map.Entry)it.next();**

**System.out.println(e.getKey() + "=" + e.getValue());**

**}**

**System.out.println("-----------------------------------------");**

**for(Iterator it = map.keySet().iterator();it.hasNext();){**

**Object key = it.next();**

**Object value = map.get(key);**

**System.out.println(key+"="+value);**

**}**

1. spring框架的原理
2. 3.如何写一个orm框架

1. 4.hibernate一级缓存和二级缓存，hibernate其他缓存
2. hibernate事务传播行为种类
3. springmvc原理
4. restful的好处
5. restful有几种请求，表单如何提交put请求
6. web中安全性问题的考虑，如何防止
7. web系统整体架构
8. hibernate如何实现声明式事务
9. java并发包
10. volatile
11. 平常都看哪些书
12. spring底层数据结构
13. 如何进行反射，如何提高反射的性能
14. 如何实现java的代理，为什么需要实现接口
15. TCP协议三次握手
16. springmvc用过哪些注解
17. springAOP可以使用哪些代理，有什么区别
18. 为什么要分三层

**二、**

1. 简单自我介绍一下

2.项目（聊的比较多主要描述自己负责的某个模块和某个功能的实现细节）

1. JAVA数据结构（HashMap等底层原理）
2. 计算机网络: 输入一个www.alibaba.com到加载整个页面经历的整个过程。
3. 数据结构:各种排序算法的时间复杂度（堆排序，归并排序，快排等），二叉树
4. 你未来的专业规划。
5. 你对新技术的了解，以及了解的途径。
6. 你还有什么要问我的

**三、**

简历上写得东西我们都要能够 ... 的和其他应聘者的聊天中，了解到研发类的会问一些常见的算法，如经典的排序算法。

**四、**

1. Java的HashMap
2. 快排
3. hash表以及hash表的处理冲突问题

1.哈希函数：

  哈希法又称散列法、杂凑法以及关键字地址计算法等，相应的表成为哈希表。

  基本思想：首先在元素的关键字K和元素的位置P之间建立一个对应关系f，使得P=f(K),其中f成为哈希函数。

  创建哈希表时，把关键字K的元素直接存入地址为f(K)的单元；查找关键字K的元素时利用哈希函数计算出该元素的存储位置P=f(K).

  创建哈希表时，把关键字K的元素直接存入地址为f(K)的单元；查找关键字K的元素时利用哈希函数计算出该元素的存储位置P=f(K).

2.哈希冲突：

  当关键字集合很大时，关键字值不同的元素可能会映像到哈希表的同一地址上，即K1!=K2，但f(K1)=f(K2),这种现象称为hash冲突，实际中冲突是 不可避免的，只能通过改进哈希函数的性能来减少冲突。

哈希方法：

平方取中法：

    当无法确定关键字中哪几位分布较均匀时，可以先求出关键字的平方值，然后按需要取平方值的中间几位作为哈希地址。

    这是因为：平方后中间几位和关键字中每一位都相关，故不同关键字会以较高的概率产生不同的哈希地址。

分段叠加法：

        这种方法是按哈希表地址位数将关键字分成位数相等的几部分（最后一部分可以较短），然后将这几部分相加，舍弃最高进位后的结果就是该关键字的哈希地址。

        具体方法有折叠法与移位法。移位法是将分割后的每部分低位对齐相加，折叠法是从一端向另一端沿分割界来回折叠（奇数段为正序，偶数段为倒序），然后将各段相加。

除留余数法：

     假设哈希表长为m，p为小于等于m的最大素数，则哈希函数为

h（k）=k  %  p ，其中%为模p取余运算。

例如，已知待散列元素为（18，75，60，43，54，90，46），表长m=10，p=7，则有

h(18)=18 % 7=4    h(75)=75 % 7=5    h(60)=60 % 7=4

h(43)=43 % 7=1    h(54)=54 % 7=5    h(90)=90 % 7=6

h(46)=46 % 7=4

此时冲突较多。为减少冲突，可取较大的m值和p值，如m=p=13，结果如下：

h(18)=18 % 13=5    h(75)=75 % 13=10    h(60)=60 % 13=8

h(43)=43 % 13=4    h(54)=54 % 13=2    h(90)=90 % 13=12

h(46)=46 % 13=7

处理冲突的方法：

   1.开放定址法(再散列法)：

     基本思想：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，                            直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。

     这种方法有一个通用的再散列函数形式：

               Hi=（H（key）+di）% m   i=1，2，…，n

     其中H（key）为哈希函数，m 为表长，di称为增量序列。增量序列的取值方式不同，相应的再散列方式也不同。

     1.线性探测再散列：

  dii=1，2，3，…，m-1

  冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

     2.二次探测再散列：

  di=12，-12，22，-22，…，k2，-k2    ( k<=m/2 )

  冲突发生时，在表的左右进行跳跃式探测，比较灵活。

     3.伪随机探测再散列：

  di=伪随机数序列。

  具体实现时，应建立一个伪随机数发生器，（如i=(i+p) % m），并给定一个随机数做起点。

1. 数据库（搜面试题）
2. **分布式、**

三步变成：分布式

·将你的**整个软件视为一个系统**（不管它有多复杂）

·将**整个系统分割为一系列的 Process（进程）， 每个 Process 完成一定的功能**

·**将这些 Process 分散到不同的机器上。分散后，选择若干种（没错一种可能不够）通信协议把他们连接起来**

认识本质：**拆分+链接**是分布式系统的本质

所谓分布式，无非就是”**将一个系统拆分成多个子系统并散布到不同设备“的过程**而已。

本质上而言，实现一个分布式系统，最核心的部分无非有两点：

**如何拆分**——可以有很多方式，核心依据一是业务需求，二是成本限制。这是实践中构建分布式系统时最主要的设计依据。

**如何连接**——光把系统拆开成 Process 还不够，关键是拆开后的 Process 之间还要能通信，因此涉及通信协议设计的问题，需要考虑的因素很多，好消息是这部分其实成熟方案很多。

**分布式的作用**

**扩展性能。**系统负载高，单台机器无法承载，希望通过多台机器来提高系统地负载能力，这个时候分布式就可以发挥作用了。

**增强可靠性**。软件，网络，机器随时都可能会出错，为了避免故障，需要将业务分散开。增强可靠性。

实现分布不难，难的在于如何选择正确分布方案

1. 。索引、B+树

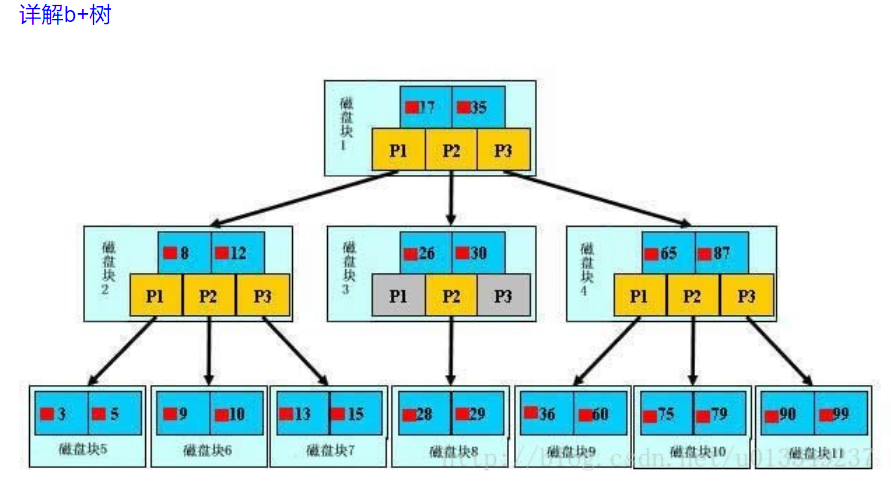
**索引目的**

索引的目的在于提高查询效率，可以类比字典，如果要查“mysql”这个单词，我们肯定需要定位到m字母，然后从上往下找到y字母，再找到剩下的sql。如果没有索引，那么你可能需要把所有的单词看一遍才能找到你想要的，如果我想找到m开头的单词呢？或者ze开头的单词呢？是不是觉得如果没有索引，这个事情根本无法完成？

**索引原理**

除了词典，生活中随处可见索引的例子，如火车站的车次表、图书的目录等。它们的原理都是一样的，**通过不断的缩小想要获得数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是我们总是通过同一种查找方法来锁定数据。**

**每次查找数据时把磁盘IO次数控制在一个很小的数量级**，最好是常数数量级。那么我们就想到如果一个高度可控的多路搜索树是否能满足需求呢？就这样，**b+树**应运而生。



如上图，是一颗b+树，关于b+树的定义可以参见b+树，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3,P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。**非叶子节点不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实的存在于数据表中**。

**b+树的查找过程**

如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生**一次IO**，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生**第二次IO**，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生**第三次IO**，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，**3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。**

**b+树性质**

IO次数取决于**b+树的高度h**，b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。

当数据项等于1时将会退化成线性表。

**索引的数据结构可以用B+树来实现。**

**索引的类型有聚族索引和非聚族索引。聚族索引与非聚族索引的区别是聚族索引的叶子节点就是数据节点，而非聚族索引的叶子节点还是索引节点，即存储的是指向数据的物理地址。**

**不同的引擎在创建主键索引和二级索引所使用的索引类型是不一样的。**

索引的类型

普通索引

唯一索引

全文索引

单列索引、多列索引

组合索引（最左前缀）

普通索引

这也是最基本的索引，它没有任何的限制，MyISAM中默认的B树索引，也是我们大多数情况下用到的索引。

唯一索引

与普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值（注意和主键不同）。如果是组合索引，则列值的组合必须唯一，创建方法和普通索引类似。

组合索引（最左前缀）

平时用的SQL查询语句一般都有比较多的限制条件，所以为了进一步榨取MySQL的效率，就要考虑建立组合索引。建立这样的组合索引，其实是相当于分别建立了下面两组组合索引（最左前缀原则）：

**索引的不足之处**

上面都在说使用索引的好处，但过多的使用索引将会造成滥用。索引也会有它的缺点：

1 虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT, UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。

2 建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在一个大表上创建了多种组合索引，索引文件的会膨胀很快。

3 索引只是提高效率的一个因素，如果你的MySQL有大数据量的表，就需要花时间研究建立最优秀的索引，或优化查询语句

-------------------------------------------------------------------------------

**一、为什么要创建索引呢（优点）？**

这是因为，创建索引可以大大提高系统的性能。

第一，   通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，   可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，   可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，   在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，   通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

**二、建立方向索引的不利因素（缺点）**

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？这种想法固然有其合理性，然而也有其片面性。虽然，索引有许多优点，但是，为表中的每一个列都增加索引，是非常不明智的。这是因为，增加索引也有许多不利的一个方面。

第一，   创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，   索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，   当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

1. 找前100大 的元素（维护一个堆）

**五、**

-- 介绍一下SVM和KNN（我的论文用到了这两个）

-- 网络七层协议

------------------------------------------------------------------------------------

OSI，OSI是一个开放性的通信系统互连参考模型，他是一个定义得非常好的协议规范。OSI模型有7层结构，每层都可以有几个子层。

·应用层

示例：TELNET，HTTP，FTP，NFS，SMTP等。

·表示层

示例：加密，ASCII等。

·会话层

示例：RPC，SQL等。

·传输层

示例：TCP，UDP，SPX。

·网络层

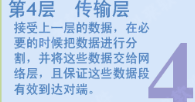
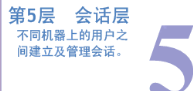
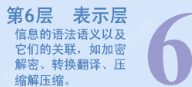
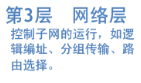
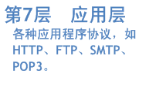
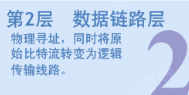
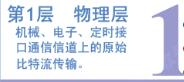
示例：IP，IPX等。

·数据链路层

示例：ATM，FDDI等。

·物理层

示例：Rj45，802.3等。



-- http、TCP、UDP的区别和联系

------------------------------------------------------------------------------------

TCP/IP

建立起一个TCP连接需要经过“三次握手”：

·第一次握手：客户端发送syn包(syn=j)到服务器，并进入SYN\_SEND状态，等待服务器确认；

·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状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开TCP连接的请求，**断开过程需要经过“四次握手”**（过程就不细写 了，就是服务器和客户端交互，最终确定断开）

**UDP与TCP的区别**

|  |  |  |
| --- | --- | --- |
|  | TCP | UDP |
| 是否连接 | 面向连接 | 面向非连接 |
| 传输可靠性 | 可靠 | 不可靠 |
| 应用场合 | 传输大量数据 | 少量数据 |
| 速度 | 慢 | 快 |

1.http是要基于TCP连接基础上的，简单的说，TCP就是单纯建立连接，不涉及任何我们需要请求的实际数据，简单的传输。http是用来收发数据，即实际应用上来的。

2. **TCP**是底层通讯协议，定义的是**数据传输和连接方式的规范**

**HTTP**是应用层协议，定义的是**传输数据的内容的规范**

HTTP协议中的数据是利用TCP协议传输的，所以支持HTTP也就一定支持TCP

3. HTTP支持的是www服务

而TCP/IP是协议

它是Internet国际互联网络的基础。TCP/IP是网络中使用的基本的通信协议。

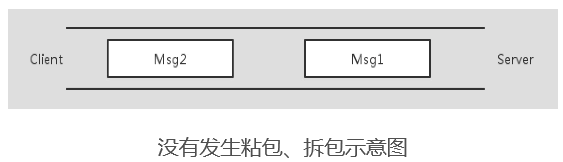
**TCP/IP**实际上是**一组协议**，它**包括上百个各种功能的协议**，如：远程登录、文件传输和电子邮件等，而TCP协议和IP协议是**保证数据完整传输的两个基本的重要协议**。通常说TCP/IP是Internet协议族，而不单单是TCP和IP。

-- TCP和UDP各自的优势，知道哪些使用UDP协议的成功案例

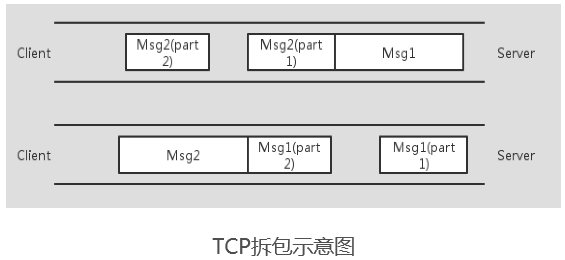
-- TCP和UDP各用了底层什么协议

-- 单个UDP报文最大容量

-- TCP粘包怎么处理







**如何解决拆包粘包**

既然知道了tcp是无界的数据流，且协议本身无法避免粘包，拆包的发生，那我们只能**在应用层数据协议上，加以控制**。通常在制定传输数据时，可以使用如下方法：

1.使用**带消息头的协议**、消息头存储消息**开始标识**及消息**长度信息**，服务端获取消息头的时候解析出消息长度，然后向后读取该长度的内容。

2.设置**定长消息**，服务端**每次读取既定长度**的内容作为一条完整消息。

3.设置**消息边界**，服务端从网络流中按消息编辑分离出消息内容。

-- DDOS攻击了解吗

-- 用过那些数据结构

-- 进程和线程的区别和练习

-- Innodb和Myisam的区别

-- C 的内存分配

-- STL标准库中的数据结构占用的哪一部分的内存

即标准模板库，是一个具有工业强度的，高效的C++程序库。

-- map的实现用的什么数据结构，插入的时间复杂度是多少

----------------------------------------------------------------------------------------------

如果插入单个元素且无暗示，时间复杂度为 **O(logn)**，其中 n 为容器的大小。

如果插入多个元素，时间复杂度为 **O(nlogn)**，“第一个” n 为插入元素数，“第二个” n 为插入元素数加容器大小。如果插入的范围中的元素已经按同样的排序规则排序，执行过程将被优化，时间复杂度甚至会降到 **O(n)**。

-- C 的类中那些成员会被算入sizeof中

-- 为什么要内存对齐

-- 为什么会有大端小端，htol这一类函数的作用

-- RAII

-- new和malloc的区别

-- linux 文件查找指令

-- top显示出来的系统信息都是什么含义

-- 手写一个智能指针类，怎么处理并发的情况，加锁的话怎么提高效率

-- linux tcp的建立过程

-- 用过vim的那些插件

-- linux下抓过指令

-- linux下查看IO占用情况

-- 讲一下线程池

-- 多线程模型和epoll模型原理，各自优点

-- 进程之间的通信方式

-- 线程之间的通信方式

-- gdb用过哪些指令，在其中怎么查看线程的运行情况

暂时就这么多吧，应该还遗漏了一些，想起来再补充

有些问题比较开放，我就把我知道的都给回答了，比如说TCP的时候，我把握手挥手拥塞控制什么的都说了

还有一些其他问题：

-- 介绍一下做过的比较自豪的项目

-- 未来3-5年的规划

-- 自己的优点

**六、**

首先让自我介绍

问项目，给了两个具体的情景让优化

问项目在安全方面有什么考虑吗（并没有，被面试官鄙视了）

----------------------------------------------------------------------------------------

**·密码保存**

保存密码的第一准则是不能明文保存密码，之前CSDN密码泄露一事还记忆犹新。通常的做法是对密码进行不可逆加密，加密时不要使用MD5或者SHA系列的算法加密，现在对这两个算法的破解研究工作已经有了相当的进展。推荐使用bcrypt。

**·CI服务器的安全性**

CI服务器和Build服务器经常是攻击者最完美的跳板，这些服务器几乎拥有你系统的所有权限（代码库，各个生产环境的访问权限等等）。请确保其安全。

**·校验客户端安全证书的“指纹”**

当使用SSL时，我们需要验证安全证书的“指纹”来确认该证书是有目标网站颁发的证书，这可以有效防止“中间人攻击”漏洞。

**·随机生成密码**

不要对所有账户设置同一个密码，一锅端就不好了，使用密码管理软件为每一个账户生成一个独立的密码。当前比较推荐的是1Password，期待有更好的密码管理软件诞生。

**·使用安全系数更高的随机数生成类库**

要想得到真正的随机数是很困难的，尤其是要满足安全领域的随机数标准，语言自带的一般随机数生成算法很难满足需要，所以JDK还提供了SecuredRandom这样的类，Ruby中也有SecureRandom这样的模块。

**·总是使用HTTPS和HSTS**

如无值得信服的理由，所有的请求都应该使用加密协议

**·不要使用“安全问题”找回密码**

一个桶能装多少水是有最短的那块木板决定的，一个安全系统的安全性也是有最薄弱的环节决定的。不要让安全问题成为你的那块短木板。

**·不要限制密码长度**

通过密码长度来保证安全性的时代已经结束了，让用户使用"Pass Phrases"是一个不错的主意。

**·允许对密码或者密码短语使用更直观的复杂度控制**

使用更有效，直观的方法帮助用户增加密码或者密码短语的难度。而不是简单的必须有大小写，必须超过8位等这些生硬的条件。

**·避免缓冲溢出**

在弱类型语言中，最常见的安全性问题就是缓冲溢出，这也是最常见的一种远程攻击方法，一般有四种防范方法： \* 通过操作系统使得缓冲区不可执行，从而阻止攻击者植入攻击代码。 \* 写正确的代码 \* 利用编译器的边界检查来实现缓冲区的保护。 \* 在程序指针失效前进行完整性检查

**·避免注入攻击**

所有的用户输入必须经过验证和清理才能传递给下游，最常见的就是SQL注入攻击

**·避免跨站脚本攻击**

系统需要保证用户不能在页面中嵌入未经验证的信息

**·避免泄露信息给第三方**

目前很多系统倾向于泄露信息给第三方，比方说Google，Facebook，新浪微博等等。这些第三方会窃取很多用户信息。

数据库的行级锁和表级锁悲观锁和乐观锁思想

---------------------------------------------

我们经常在开发中遇到数据库并发处理时，处理不一致的问题。需要对程序做并发控制。

典型的并发时出现的冲突有两种：

1、丢失更新：一个事务的更新覆盖了其它事务的更新结果，就是所谓的更新丢失。

例如X=0 A1把X=X+1 A2也设置X=X+1 如果A1和A2同时执行，可能出现最终X=1的情况。而我们需要得到的是X=2 。 这就需要update处理为串行化的。

2、脏读：读到的数据可能是旧的。 初使X=0 A1把X设置为1，A2读取时X是1，但A1后面又修改成了2，或回滚到了0，那么A2读取的数据就是脏数据。

A2读取的是未提交的数据。一般数据库不会设为读未提交，所以一般不会出现脏读。

**悲观锁：**需要使用数据库的锁机制，如数据库有表级排它锁，有行级排它锁。

**假定一切操作都可能发现并发冲突**，所以采取悲观态度。通过加锁，**屏蔽一切可能违反数据完整性的操作**

**乐观锁：**其实不是真实的去锁住记录不让访问，或者不让更新。

**假定操作很少发生冲突**，一般对于**读多写少**的情况。**只在提交操作时检查是否违反数据完整性**。

·乐观锁不能解决脏读的问题。

·可以通过版本号是否比上个版本号或者时间戳来实现。

·对于冲突检测后的处理，需要业务逻辑去处理。

------------------------------------------------

**表级，**直接锁定整张表，在你锁定期间，其它进程无法对该表进行写操作。如果你是写锁，则其它进程则读也不允许

**行级,**仅对指定的记录进行加锁，这样其它进程还是可以对同一个表中的其它记录进行操作。

**页级，**表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，**一次锁定相邻的一组记录。**

上述三种锁的特性可大致归纳如下：

1） 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。

2） 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

3） 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

InnoDB有两种模式的**行锁**：

1）共享锁：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。

( Select \* from table\_name where ......lock in share mode)

2）排他锁：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和 排他写锁。(select \* from table\_name where.....for update)

为了允许行锁和表锁共存，实现多粒度锁机制；同时还有两种内部使用的意向锁（都是表锁），分别为意向共享锁和意向排他锁。

后台缓存为啥选择redis，问了redis的数据结构，持久化

项目做压测了吗，用什么工具压测的，大概并发量有多少，你认为自己项目的性能瓶颈在什么地方

怎么优化瓶颈

大量请求同时过来怎么削峰

说一下rabbitmq的几个交换模式

了解其他的消息队列吗？（并不了解）

JVM内存分区，内存泄露与内存溢出的区别

---------------------------------------------------------------------------

**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**，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

一般我们所说的内存泄漏是指**堆内存的泄漏**，堆内存是指程序从堆中分配的，大小任意的（内存块的大小可以在程序运行期决定），使用完成之后必须显示释放内存。应用程序一般使用malloc、realoc、new等函数从堆中分配到一块内存块，使用完成后，程序必须负责相应的释放。在C中使用free(),C++中delete、delete[]、free()。而Java中由于垃圾回收机制不用手动释放。

如果内存不能释放，这块内存就不能再次使用，我们就说这块内存泄漏了。

**内存溢出**（out of memory）：**程序要求的内存，超出了系统所能分配的范围**。如：我们用一个int型4字节的数据来装一个float型8字节的数据，就会产生内存溢出。不过我们在编程是可以强制类型转换int XX = (int)float;只取float 4字节数据给int.

遇到过项目频繁出现FULL GC的情况吗？怎么排查（项目规模小，并没有遇到过）

---------------------------------------------------------------------------

**GC机制**

要准确理解Java的垃圾回收机制，就要从：“什么时候”，“对什么东西”，“做了什么”三个方面来具体分析。

第一：“什么时候”即就是**GC触发的条件**。GC触发的条件有两种。（1）程序调用System.gc时可以触发；（2）系统自身来决定GC触发的时机。

系统判断GC触发的依据：根据Eden区和From Space区的内存大小来决定。当内存大小不足时，则会启动GC线程 并停止应用线程。

第二：“对什么东西”笼统的认为是**Java对象**并没有错。但是准确来讲，GC操作的对象分为：**通过可达性分析法无法搜索到的对象**和**可以搜索到的对象**。对于搜索不到的方法进行标记。

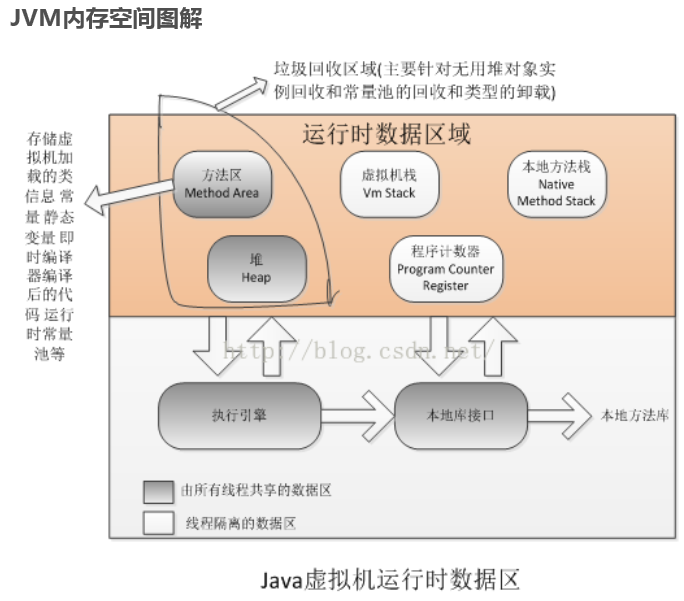
第三：“做了什么”最浅显的理解为**释放对象**。但是从GC的底层机制可以看出，对于可以搜索到的对象进行**复制操作**，对于搜索不到的对象，调用**finalize()方法进行释放**。

**具体过程：**当GC线程启动时，会通过可达性分析法把Eden区和From Space区的存活对象复制到To Space区，然后把Eden Space和From Space区的对象释放掉。当GC轮训扫描To Space区一定次数后，把依然存活的对象复制到老年代，然后释放To Space区的对象。

对于用可达性分析法搜索不到的对象，GC并不一定会回收该对象。要完全回收一个对象，至少需要经过两次标记的过程。

GC过程中用到的回收算法：

通过上面的GC过程不难看出，Java堆中的年轻代和老年代采用了不同的回收算法。**年轻代采用了复制法**；而**老年代采用了标记-整理法**



**程序计数器：**线程私有。是一块较小的内存，是当前线程所执行的字节码的行号指示器。是Java虚拟机规范中唯一没有规定OOM（OutOfMemoryError）的区域。

**Java栈：**线程私有。生命周期和线程相同。是Java方法执行的内存模型。执行每个方法都会创建一个栈帧，用于存储局部变量和操作数（对象引用）。局部变量所需要的内存空间大小在编译期间完成分配。所以栈帧的大小不会改变。存在两种异常情况：若线程请求深度大于栈的深度，抛StackOverflowError。若栈在动态扩展时无法请求足够内存，抛OOM。

**Java堆**：所有线程共享。虚拟机启动时创建。存放对象实力和数组。所占内存最大。分为新生代（Young区），老年代（Old区）。新生代分Eden区，Servior区。Servior区又分为From space区和To Space区。Eden区和Servior区的内存比为8:1。 当扩展内存大于可用内存，抛OOM。

**方法区：**所有线程共享。用于存储已被虚拟机加载的类信息、常量、静态变量等数据。又称为非堆（Non – Heap）。方法区又称“永久代”。GC很少在这个区域进行，但不代表不会回收。这个区域回收目标主要是针对常量池的回收和对类型的卸载。当内存申请大于实际可用内存，抛OOM。

**本地方法栈**：线程私有。与Java栈类似，但是不是为Java方法（字节码）服务，而是为本地非Java方法服务。也会抛StackOverflowError和OOM。

**Minor GC ，Full GC 触发条件**

Minor GC触发条件：当Eden区满时，触发Minor GC。

**Full GC触发条件：**

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法去空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

平时怎么学习的，有什么要问我的吗

**七、**

之前过了个简单的简历面，过了几天后没打来以为凉了，然后昨晚又接到了电话，括号内容是回答说的，理解有限，不一定都对，欢迎纠正～加油每一个牛友们！

阿里一面:

1.学过哪些技术知识呢？

2.说说接口和抽象类的区别(语法，设计，跨域三方面)

1、抽象类

包含一个抽象方法的类就是抽象类

2、抽象方法

声明而未被实现的方法，抽象方法必须使用abstract关键词字声明

3、抽象类被子类继承，子类（如果不是抽象类）必须重写抽象类中的所有抽象方法

4、抽象类不能被直接实例化，要通过其子类进行实例化

5、只要包含一个抽象方法的抽象类，该方法必须要定义成抽象类，不管是否还包含有其他方法。

6、子类中的抽象方法不能与父类的抽象方法同名。

7、abstract不能与final并列修饰同一个类。

8、abstract 不能与private、static、final或native并列修饰同一个方法。

为什么需要用到抽象类？

当子类HPPrinter和CannonPrinter要继承printer类的时候，必须重写print函数。这样写的好处是，每个打印机都有自己打印方式，此举可以防止子类没有写明打印方式。

---------------------------------------------

1、接口是java中最重要的概念，接口可以理解为一种特殊的类，里面全部是由**全局常量**和**公共的抽象方法**组成

接口中所有方法都是抽象方法；

接口当中所有的方法都是public类型

1. 接口的实现也必须通过子类，使用关键字implements，而且接口是可以多实现的
2. 一个类可以同时继承抽象类和接口
3. 一个接口不能继承抽象类，但可以通过extends关键字继承多个接口，实现接口的多继承

----------------------------------------------------

**抽象类和接口区别**

（一）语法

·抽象类方式中，**抽象类可以拥有任意范围的成员数据，同时也可以拥有自己的非抽象方法**，

·但是接口方式中，它**仅能够有静态、不能修改的成员数据**（但是我们一般是不会在接口中使用成员数据），同时它**所有的方法都必须是抽象的**。

·在某种程度上来说，**接口是抽象类的特殊化**。

·对子类而言，它**只能继承一个抽象类（**这是java为了数据安全而考虑的），但是却**可以实现多个接口**。

（二）设计层次

1、 抽象层次不同

**抽象类是对类抽象，而接口是对行为的抽象。**抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。

2、 跨域不同

**抽象类所跨域的是具有相似特点的类，而接口却可以跨域不同的类。**我们知道抽象类是从子类中发现公共部分，然后泛化成抽象类，子类继承该父类即可，但是接口不同。实现它的子类可以不存在任何关系，共同之处。例如猫、狗可以抽象成一个动物类抽象类，具备叫的方法。鸟、飞机可以实现飞Fly接口，具备飞的行为，这里我们总不能将鸟、飞机共用一个父类吧！所以说抽象类所体现的是一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"is-a" 关系，即**父类和派生类在概念本质上应该是相同的**。对于接口则不然，**并不要求接口的实现者和接口定义在概念本质上是一致的**， 仅仅是实现了接口定义的契约而已。

3、 设计层次不同

对于**抽象类而言，它是自下而上来设计**的，我们要先知道子类才能抽象出父类，而接口则不同，它根本就不需要知道子类的存在，只需要定义一个规则即可，至于什么子类、什么时候怎么实现它一概不知。比如我们只有一个猫类在这里，如果你这是就抽象成一个动物类，是不是设计有点儿过度？我们起码要有两个动物类，猫、狗在这里，我们在抽象他们的共同点形成动物抽象类吧！所以说**抽象类往往都是通过重构而来的**！但是接口就不同，比如说飞，我们根本就不知道会有什么东西来实现这个飞接口，怎么实现也不得而知，我们要做的就是事前定义好飞的行为接口。所以说抽象类是自底向上抽象而来的，**接口是自顶向下设计**出来的。

3.怎么判断两个对象是否相等？(名称和类加载器)

if (**getClass() != obj.getClass()**)

return false;

 if (!**name.equals(other.name**))

return false;

4.==和equals的区别(比较方式)

equals被用来判断两个对象是否相等。

**equals通常用来比较两个对象的内容是否相等，==用来比较两个对象的地址是否相等**。

equals方法默认等同于“==”

Object类中的equals方法定义为判断**两个对象的地址是否相等**（可以理解成是否是同一个对象），地址相等则认为是对象相等。这也就意味着，我们新建的所有类如果没有复写equals方法，那么判断两个对象是否相等时就等同于“==”，也就是两个对象的地址是否相等。

equals返回true有两种可能，一种是两个对象地址相同，一种是两个对象内容相同。

5.排序算法有哪些呢(分类说，插入，选择，交换，归并，基排)

5.堆排序的时间复杂度？怎么计算的(nlogn)

6.说说归并的算法过程(先划分，归并，递归实现，把过程理清)

10种排序

7.知道先序，中序，后序遍历吗？(把算法过程说一遍)

深度遍历有前序、中序以及后序三种遍历方法，广度遍历即我们平常所说的层次遍历。

四种主要的遍历思想为：

前序遍历：根结点 ---> 左子树 ---> 右子树

中序遍历：左子树---> 根结点 ---> 右子树

后序遍历：左子树 ---> 右子树 ---> 根结点

层次遍历：只需按层次遍历即可

一、前序遍历

1）根据上文提到的遍历思路：根结点 ---> 左子树 ---> 右子树，很容易写出递归版本：

public void preOrderTraverse1(TreeNode root) {

        if (root != null) {

            System.out.print(root.val+"  ");

            preOrderTraverse1(root.left);

            preOrderTraverse1(root.right);

        }

    }

2）现在讨论非递归的版本：

根据前序遍历的顺序，优先访问根结点，然后在访问左子树和右子树。所以，对于任意结点node，第一部分即直接访问之，之后在判断左子树是否为空，不为空时即重复上面的步骤，直到其为空。若为空，则需要访问右子树。注意，在访问过左孩子之后，需要反过来访问其右孩子，所以，需要栈这种数据结构的支持。对于任意一个结点node，具体步骤如下：

a)访问之，并把结点node入栈，当前结点置为左孩子；

b)判断结点node是否为空，若为空，则取出栈顶结点并出栈，将右孩子置为当前结点；否则重复a)步直到当前结点为空或者栈为空（可以发现栈中的结点就是为了访问右孩子才存储的）

代码如下：

public void preOrderTraverse2(TreeNode root) {

        LinkedList<TreeNode> stack = new LinkedList<>();

        TreeNode pNode = root;

        while (pNode != null || !stack.isEmpty()) {

            if (pNode != null) {

                System.out.print(pNode.val+"  ");

                stack.push(pNode);

                pNode = pNode.left;

            } else { //pNode == null && !stack.isEmpty()

                TreeNode node = stack.pop();

                pNode = node.right;

            }

        }

    }

-----------------------------------

层次遍历

层次遍历的代码比较简单，只需要一个队列即可，先在队列中加入根结点。之后对于任意一个结点来说，在其出队列的时候，访问之。同时如果左孩子和右孩子有不为空的，入队列。代码如下：

public void levelTraverse(TreeNode root) {

        if (root == null) {

            return;

        }

        LinkedList<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        while (!queue.isEmpty()) {

            TreeNode node = queue.poll();

            System.out.print(node.val+"  ");

            if (node.left != null) {

                queue.offer(node.left);

            }

            if (node.right != null) {

                queue.offer(node.right);

            }

        }

    }

8.知道Java高并发相关知识吗？(并发包，把里面的内容介绍一边，介绍了闭锁，同步屏障，信号量的应用和底层实现:流控，线程执行顺序等)

9.谈谈JVM(从javac编译4阶段:词法，语法语义，生成，到类加载过程:双亲委派:加载，验证，准备，解析，实例化，实例化先谈堆分区，再说JVM内存结构，然后到GC,GC算法，触发条件，晋升，YGC,CMS过程实现，可达性分析等等，把整个体系说完后面试官肯定了一句说理解得不错…)

一.Javac

1.Javac编译器的作用

将符合Java语言规范的源代码转化成符合Java虚拟机规范的Java字节码。

2.编译器主要的几个处理阶段

词法分析、语法分析、语义分析和代码生成,基于访问者模式来遍历语法树的过程。

二.ClassLoader

将Class加载到JVM中,审查每个类应该由谁加载,将Class字节码重新解析成JVM统一要求的对象格式。

1.ClassLoader等级加载机制

(1)Bootstrap ClassLoader,主要加载JVM自身工作需要的类,这个ClassLoader完全是JVM自己控制的,没有更高一级的父加载器,也没有子加载器。

(2)ExtClassLoader,它服务的特定目标在System.getProperty("java.ext.dirs")目录下。

(3)AppClassLoader,它的父类是ExtClassLoader。所有在System.getProperty("java.class.path")目录下的类都可以被这个类加载器加载。

扩展类加载器,通常是继承URLClassLoader类或是其他子类,它的父加载器都是AppClassLoader,因为不管调用哪个父类构造器,创建的对象都必须最终调用getSystemClassLoader()作为父加载器获得AppClassLoader。

2.加载class文件

(1)找到.class文件并把这个文件包含的字节码加载到内存中,通过ClassLoader类findClass()方法来实现。

(2)又可分为三个步骤,分别是字节码验证、Class类数据结构分析及相应的内存分配和最后的符号表的链接。

(3)类中静态属性和初始化赋值,以及静态块的执行等。

3.显示加载一个类的方式

(1)通过类Class中的forName()方法。

(2)通过类ClassLoader中的loadClass()方法。

(3)通过类ClassLoader中的findSystemClass()。

4.什么情况下实现自己的ClassLoader

(1)在自定义路径下查找自定义的class类文件。

(2)保证通过网络传输的类的安全性,可以将类经过加密后再传输,在加载到JVM之前需要对类的字节码再解密。

(3)如果我们可以检查已经加载的class文件是否被修改,如果修改了,可以重新加载这个类,从而实现类的热部署。

三.JVM

1.体系结构

(1)类加载器,在JVM启动时或者在类运行时将需要的class加载到JVM中。

(2)执行引擎,任务是负责执行class文件中包含的字节码指令,相当于实际机器上的CPU。

(3)内存区,将内存划分成若干个区以模拟实际机器上的存储、记录和调度功能模块,如实际机器上的各种功能的寄存器或者PC指针的记录器等。

(4)本地方法调用,调用C或C++实现的本地方法代码返回结果。

2.类加载器

每一个被JVM装载的类型都有一个对应的java.lang.Class类的实例来表示该类型。

3.执行引擎

执行引擎是JVM的核心部分,执行引擎的作用就是解析JVM字节码指令,得到执行结果。执行引擎具体采取什么方式由JVM的实现厂家自己去实现。

(1)直接解释执行。

(2)采用JIT技术,将字节码转成本地代码去执行。如JVM在执行程序时会记录某个方法的执行次数,如果次数到一个阀值就会编译这个方法为本地代码。

(3)基于栈的架构,实现JVM跨平台,更好地优化代码和指令的紧凑性。如SUN的hotspot是基于栈的执行引擎。

(4)基于寄存器的架构,性能较好,但无法跨平台,因为在没有或者很少的寄存器的机器上也要同样能正确地执行Java代码。如Google的Android平台上的Dalvik VM就是基于特定芯片(ARM)设计的基于寄存器的架构。

4.内存管理

(1)PC寄存器,用于保存当前正常执行的程序的内存地址。

(2)本地方法栈,为JVM运行Native方法准备的空间,很多Native方法都是用C语言实现的,所有它通常又叫C栈。

(3)Java栈,当创建一个线程时,JVM就会为这个线程创建一个对应的Java栈,这个Java栈中又会含有多个栈帧,一个栈帧对应一个方法和方法内的变量,返回值等信息。

(4)Java堆,存储Java对象的地方,它是JVM管理Java对象的核心存储区域,被所有Java线程所共享。

(5)方法区,用于存储类结构信息,包括:常量池、域、方法数据、方法体、构造函数、实例初始化、接口初始化都存储在这个区域。属于Java堆中的永久区。

(6)常量池,包括:编译期的数字常量、方法或者域的引用。属于方法区的一部分。

5.内存分配策略

(1)静态内存分配

(2)栈内存分配

(3)堆内存分配

6.内存回收策略

(1)静态内存分配和回收,Java被编译时就已经能够确定需要的内存空间,当程序被加载时系统把内存一次性分配给它。如局部变量包括数据原生类型(long等占用8个字节)和对象的引用(Object对象类型引用会占用4个字节空间)。

(2)动态内存分配和回收,Java中对象的内存空间是动态分配的,所谓的动态分配就是在程序执行时才知道要分配的存储空间大小。

(3)基于分代的垃圾收集方式,把对象按照寿命长短来分组,分为年轻代和年老代,新创建的对象被分在年轻代,如果对象经过几次回收后仍然存活,那么再把这个对象划分到年老代。年老代的收集频度不像年轻代那么频繁。

10.做过哪些项目？用过哪些框架和技术呢？(SSH,M,Redis,Netty)

11.看过spring 源码吗？(把容器启动过程说了一遍，xml解析，bean装载，bean缓存等)

容器启动流程中涉及的主要类：

ContextLoaderListener：注册在web.xml中，web应用启动时，会创建它，并回调它的initWebApplicationContext()方法，从而创建并启动spring容器。必须继承ServletContextListener。

WebApplicationContext：用于web应用的spring容器上下文，它代表了spring容器，继承自ApplicationContext。是一个接口，在ContextLoader.properties配置文件中可以声明它的实现类。默认实现类为XmlWebApplicationContext。ApplicationContext继承自BeanFactory，并扩展了它的很多功能。

ServletContext：web容器（如tomcat）的上下文，不要和ApplicationContext搞混了。

**web容器的初始化过程为**

·web容器（如tomcat）读取web.xml, 读取文件中两个节点和

·容器创建ServletContext，它是web的上下文，整个web项目都会用到它

·读取context-param节点，它以 键值对的形式出现。将节点值转化为键值对，传给ServletContext

·容器创建中的实例，创建监听器。监听器必须继承ServletContextListener

·调用ServletContextListener的contextInitialized()方法，spring容器的创建和初始化就是在这个方法中

在web容器初始化过程中，会创建节点的监听器，并调用它的contextInitialized()方法。这个方法中会完成spring容器的创建，初始化，以及beans的创建。默认listener为ContextLoaderListener

initWebApplicationContext()主要做三件事

·创建WebApplicationContext，通过createWebApplicationContext()方法

·加载spring配置文件，并创建beans。通过configureAndRefreshWebApplicationContext()方法

·将spring容器context挂载到ServletContext 这个web容器上下文中。通过servletContext.setAttribute()方法。

创建spring容器的流程比较简单，具体为两步

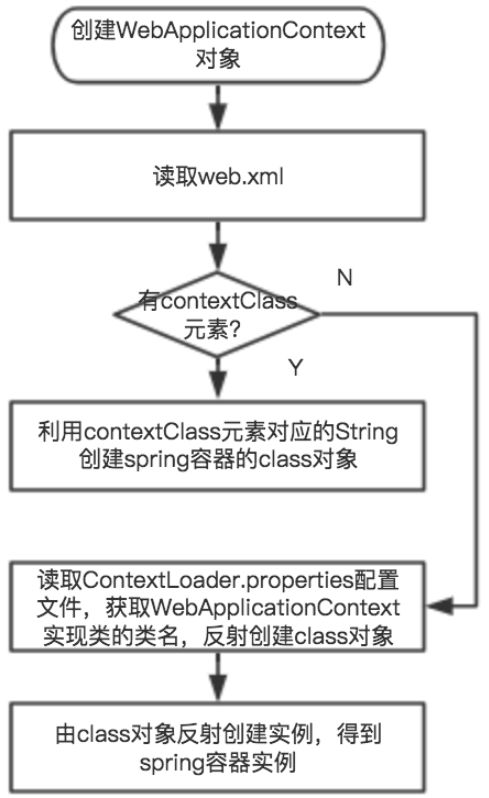
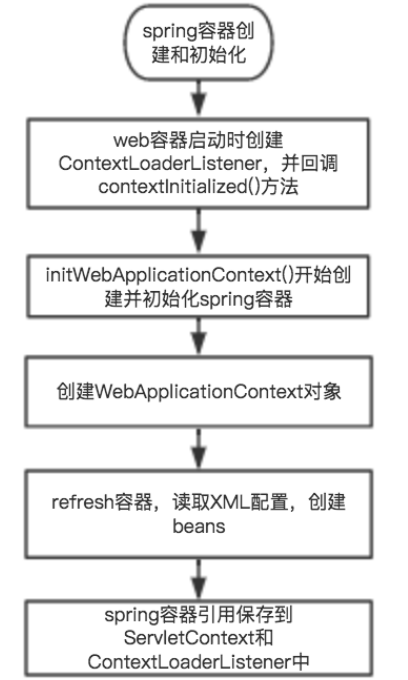
·获取WebApplicationContext实现类的class对象

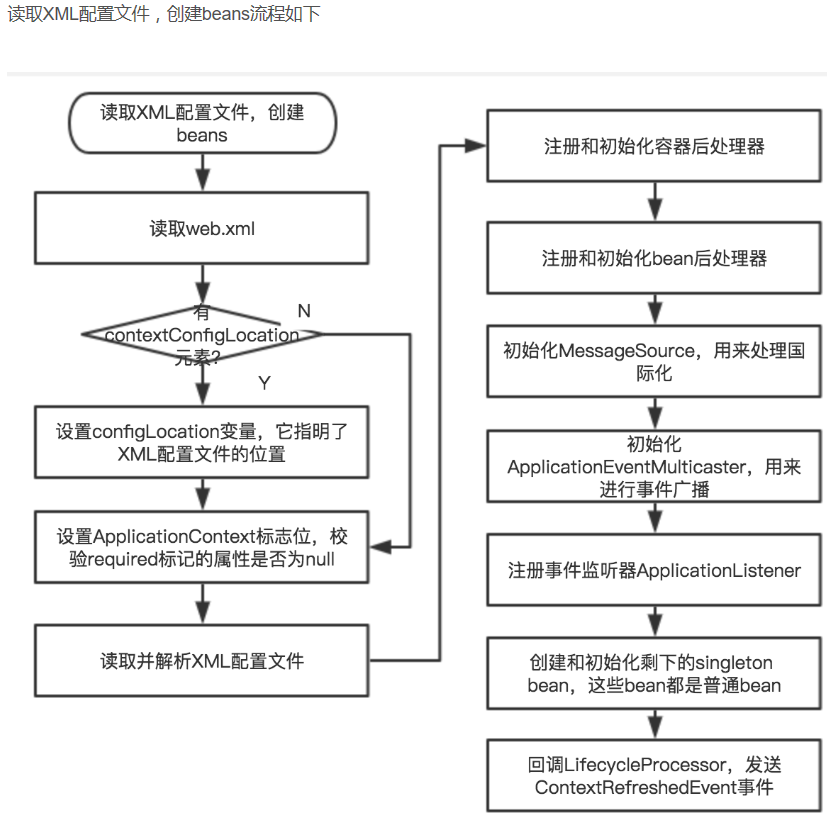
·根据class对象创建实例对象

创建完Spring容器后，就会加载spring配置文件，并创建beans。这个过程在configureAndRefreshWebApplicationContext()方法

configureAndRefreshWebApplicationContext()这个方法会先读取web.xml中声明的contextConfigLocation元素，通过它找到spring配置文件。然后在refresh()方法中读取配置文件，并创建和初始化beans。所以重中之重还是refresh()方法。

spring容器初始化的整个流程图如下： 创建WebApplicationContext对象流程如下：





12.说说AOP实现原理(动态代理:jdk,cglib，分别用了哪些类或接口)

13.事务的实现原理？(源码)

14.说说四层网络模型(每层的协议及应用)

15.谈谈tcp,udp的区别，分别有哪些应用(可靠，面向连接，首部)

16.说说tcp三次握手过程？

17.为什么是3次，2次可以吗？4次呢？(从失效的数据报谈2次为什么不行，从响应时间说为什么不用4次)

18.你有什么想问我的？(问了分布式，负载均衡，缓存相关的，面试官回答说这些对校招是加分项，不懂也没关系，懂了的就更好，说校招关键还是基础，多看源码分析，多关注底层原理)

19.面了50分钟左右