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(logn)

6.说说归并的算法过程(先划分，归并，递归实现，把过程理清)

**排序**

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

术语说明

·稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；

·不稳定：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；

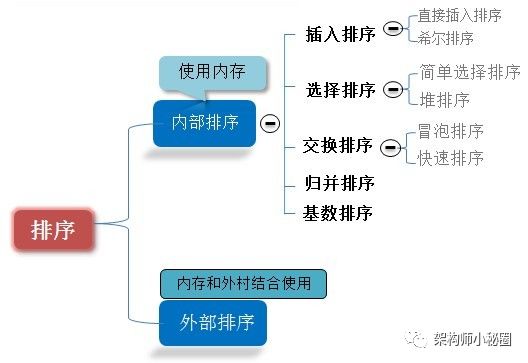
·内排序：所有排序操作都在内存中完成；

·外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；

·时间复杂度： 一个算法执行所耗费的时间。

·空间复杂度：运行完一个程序所需内存的大小。





**比较和非比较的区别**

常见的快速排序、归并排序、堆排序、冒泡排序等属于比较排序。在排序的最终结果里，元素之间的次序依赖于它们之间的比较。每个数都必须和其他数进行比较，才能确定自己的位置。

在冒泡排序之类的排序中，问题规模为n，又因为需要比较n次，所以平均时间复杂度为O(n²)。在归并排序、快速排序之类的排序中，问题规模通过分治法消减为logN次，所以时间复杂度平均O(nlogn)。

比较排序的优势是，适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，比较排序适用于一切需要排序的情况。

计数排序、基数排序、桶排序则属于非比较排序。非比较排序是通过确定每个元素之前，应该有多少个元素来排序。针对数组arr，计算arr[i]之前有多少个元素，则唯一确定了arr[i]在排序后数组中的位置。

非比较排序只要确定每个元素之前的已有的元素个数即可，所有一次遍历即可解决。算法时间复杂度O(n)。

非比较排序时间复杂度底，但由于非比较排序需要占用空间来确定唯一位置。所以对数据规模和数据分布有一定的要求。

1. **冒泡排序（Bubble Sort）**

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

**最佳情况：T(n) = O(n) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)**

·比较相邻的元素。如果第一个比第二个大，就交换它们两个；

·对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；

·针对所有的元素重复以上的步骤，除了最后一个；

·重复步骤1~3，直到排序完成。

for (int i = 0; i < array.length; i++)

for (int j = 0; j < array.length - 1 - i; j++)

if (array[j + 1] < array[j]) {

int temp = array[j + 1];

array[j + 1] = array[j];

array[j] = temp;

}

1. **选择排序（Selection Sort）**

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

**最佳情况：T(n) = O(n2) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)**

表现最稳定的排序算法之一，因为无论什么数据进去都是O(n2)的时间复杂度，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。

**工作原理：**首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

for (int i = 0; i < array.length; i++) {

int minIndex = i;

for (int j = i; j < array.length; j++) {

if (array[j] < array[minIndex]) //找到最小的数

minIndex = j; //将最小数的索引保存

}

int temp = array[minIndex];

array[minIndex] = array[i];

array[i] = temp;

}

1. **插入排序（Insertion Sort）**

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

**最佳情况：T(n) = O(n) 最坏情况：T(n) = O(n2) 平均情况：T(n) = O(n2)**

·从第一个元素开始，该元素可以认为已经被排序；

·取出下一个元素，在已经排序的元素序列中从后向前扫描；

·如果该元素（已排序）大于新元素，将该元素移到下一位置；

·重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；

·将新元素插入到该位置后；

·重复步骤2~5。

for (int i = 0; i < array.length - 1; i++) {

current = array[i + 1];

int preIndex = i;

while (preIndex >= 0 && current < array[preIndex]) {

array[preIndex + 1] = array[preIndex];

preIndex--;

}

array[preIndex + 1] = current;

}

1. **希尔排序（Shell Sort）**

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

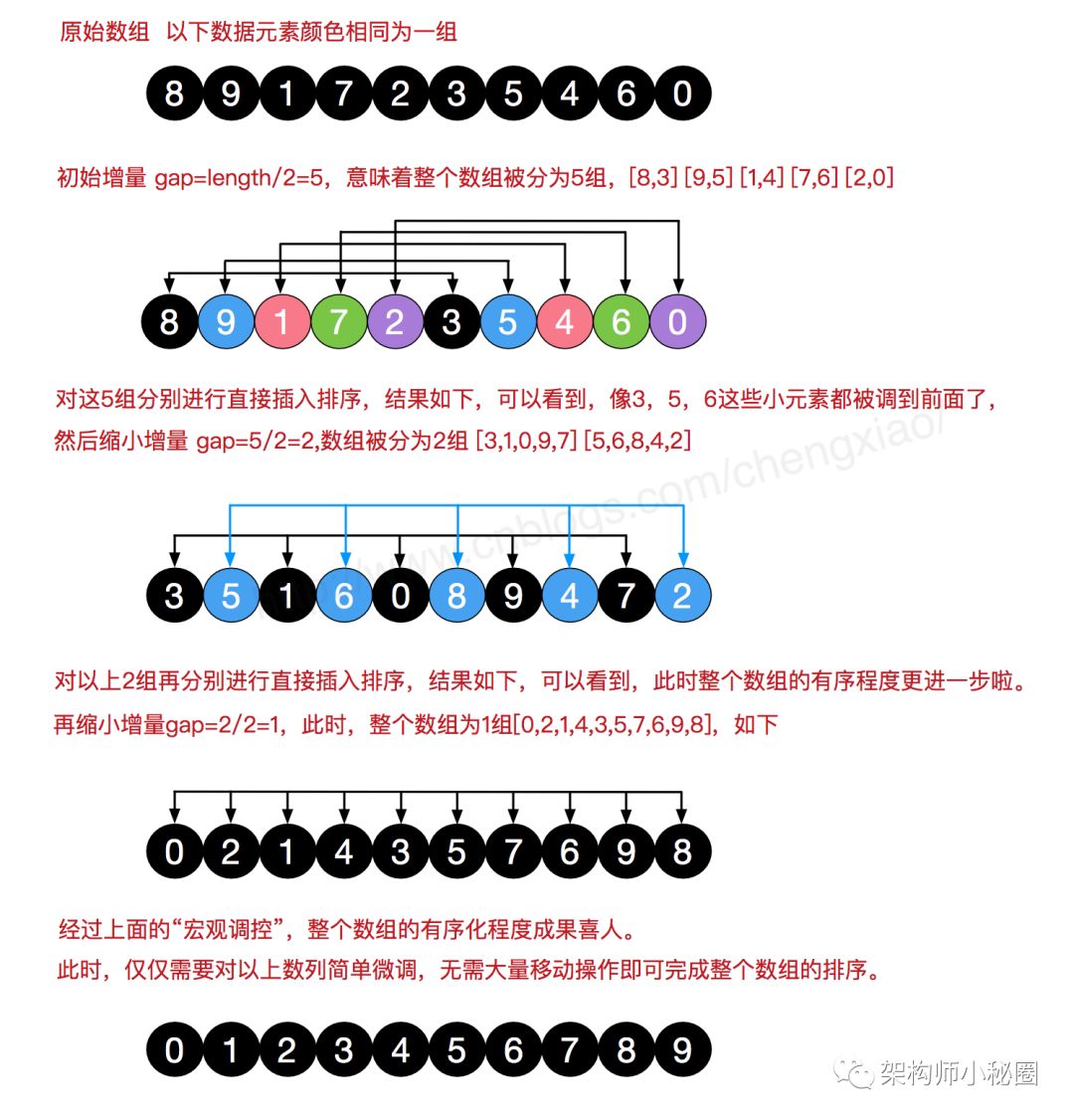
**最佳情况：T(n) = O(nlog2 n) 最坏情况：T(n) = O(nlog2 n) 平均情况：T(n) =O(nlog2n)**

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

·选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

·按增量序列个数k，对序列进行k 趟排序；

·每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。



while (gap > 0) {

for (int i = gap; i < len; i++) {

temp = array[i];

int preIndex = i - gap;

while (preIndex >= 0 && array[preIndex] > temp) {

array[preIndex + gap] = array[preIndex];

preIndex -= gap;

}

array[preIndex + gap] = temp;

}

gap /= 2;

}

return array;

1. **归并排序（Merge Sort）**

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

**最佳情况：T(n) = O(n) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)**

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是O(n log n）的时间复杂度。代价是需要额外的内存空间。

·把长度为n的输入序列分成两个长度为n/2的子序列；

·对这两个子序列分别采用归并排序；

·将两个排序好的子序列合并成一个最终的排序序列

for (int index = 0, i = 0, j = 0; index < result.length; index++) {

if (i >= left.length)

result[index] = right[j++];

else if (j >= right.length)

result[index] = left[i++];

else if (left[i] > right[j])

result[index] = right[j++];

else

result[index] = left[i++];

}

return result;

1. **快速排序（Quick Sort）**

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

**最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(nlogn)**

·从数列中挑出一个元素，称为 “基准”（pivot）；

·重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

·递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

public static int partition(int[] array, int start, int end) {

int pivot = (int) (start + Math.random() \* (end - start + 1));

int smallIndex = start - 1;

swap(array, pivot, end);

for (int i = start; i <= end; i++)

if (array[i] <= array[end]) {

smallIndex++;

if (i > smallIndex)

swap(array, i, smallIndex);

}

return smallIndex;

}

public static void swap(int[] array, int i, int j) {

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

1. **堆排序（Heap Sort）**

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

**最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)**

·将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；

·将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1]<=R[n]；

·由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

public static int[] HeapSort(int[] array) {

len = array.length;

if (len < 1) return array;

//1.构建一个最大堆

buildMaxHeap(array);

//2.循环将堆首位（最大值）与末位交换，然后在重新调整最大堆

while (len > 0) {

swap(array, 0, len - 1);

len--;

adjustHeap(array, 0);

}

return array;

}

// 建立最大堆

public static void buildMaxHeap(int[] array) {

//从最后一个非叶子节点开始向上构造最大堆

for (int i = (len - 1) / 2; i >= 0; i--) {

adjustHeap(array, i);

}

}

//调整使之成为最大堆

public static void adjustHeap(int[] array, int i) {

int maxIndex = i;

//如果有左子树，且左子树大于父节点，则将最大指针指向左子树

if (i \* 2 < len && array[i \* 2] > array[maxIndex])

maxIndex = i \* 2;

//如果有右子树，且右子树大于父节点，则将最大指针指向右子树

if (i \* 2 + 1 < len && array[i \* 2 + 1] > array[maxIndex])

maxIndex = i \* 2 + 1;

//如果父节点不是最大值，则将父节点与最大值交换，并且递归调整与父节点交换的位置。

if (maxIndex != i) {

swap(array, maxIndex, i);

adjustHeap(array, maxIndex);

}

}

1. **计数排序（Counting Sort）**

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

**最佳情况：T(n) = O(n+k) 最差情况：T(n) = O(n+k) 平均情况：T(n) = O(n+k)**

计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。然后根据数组C来将A中的元素排到正确的位置。它只能对整数进行排序。

·找出待排序的数组中最大和最小的元素；

·统计数组中每个值为i的元素出现的次数，存入数组C的第i项；

·对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；

·反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

1. **桶排序（Bucket Sort）**

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

**最佳情况：T(n) = O(n+k) 最差情况：T(n) = O(n+k) 平均情况：T(n) = O(n2)**

桶排序 (Bucket sort)的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排

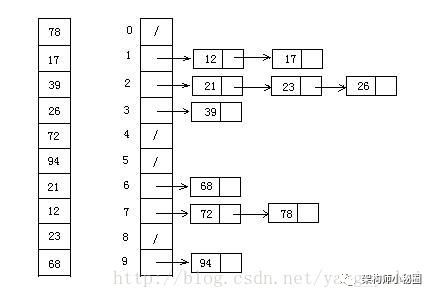
·人为设置一个BucketSize，作为每个桶所能放置多少个不同数值（例如当BucketSize==5时，该桶可以存放｛1,2,3,4,5｝这几种数字，但是容量不限，即可以存放100个3）；

·遍历输入数据，并且把数据一个一个放到对应的桶里去；

·对每个不是空的桶进行排序，可以使用其它排序方法，也可以递归使用桶排序；

·从不是空的桶里把排好序的数据拼接起来。

注意，如果递归使用桶排序为各个桶排序，则当桶数量为1时要手动减小BucketSize增加下一循环桶的数量，否则会陷入死循环，导致内存溢出。



public static ArrayList<Integer> BucketSort(ArrayList<Integer> array, int bucketSize) {

if (array == null || array.size() < 2)

return array;

int max = array.get(0), min = array.get(0);

// 找到最大值最小值

for (int i = 0; i < array.size(); i++) {

if (array.get(i) > max)

max = array.get(i);

if (array.get(i) < min)

min = array.get(i);

}

int bucketCount = (max - min) / bucketSize + 1;

ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketCount);

ArrayList<Integer> resultArr = new ArrayList<>();

for (int i = 0; i < bucketCount; i++) {

bucketArr.add(new ArrayList<Integer>());

}

for (int i = 0; i < array.size(); i++) {

bucketArr.get((array.get(i) - min) / bucketSize).add(array.get(i));

}

for (int i = 0; i < bucketCount; i++) {

if (bucketCount == 1)

bucketSize--;

ArrayList<Integer> temp = BucketSort(bucketArr.get(i), bucketSize);

for (int j = 0; j < temp.size(); j++)

resultArr.add(temp.get(j));

}

return resultArr;

}

1. **基数排序（Radix Sort）**

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

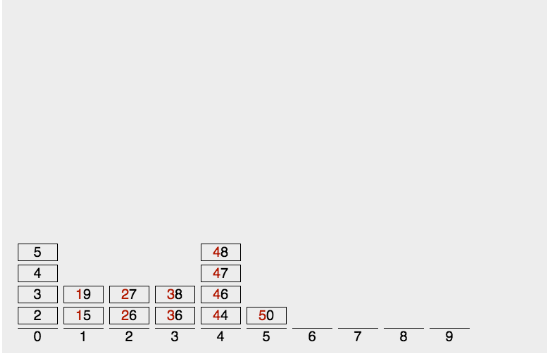
**最佳情况：T(n) = O(n \* k) 最差情况：T(n) = O(n \* k) 平均情况：T(n) = O(n \* k)**

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

·取得数组中的最大数，并取得位数；

·arr为原始数组，从最低位开始取每个位组成radix数组；

·对radix进行计数排序（利用计数排序适用于小范围数的特点）；



基数排序有两种方法：

MSD 从高位开始进行排序 LSD 从低位开始进行排序

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

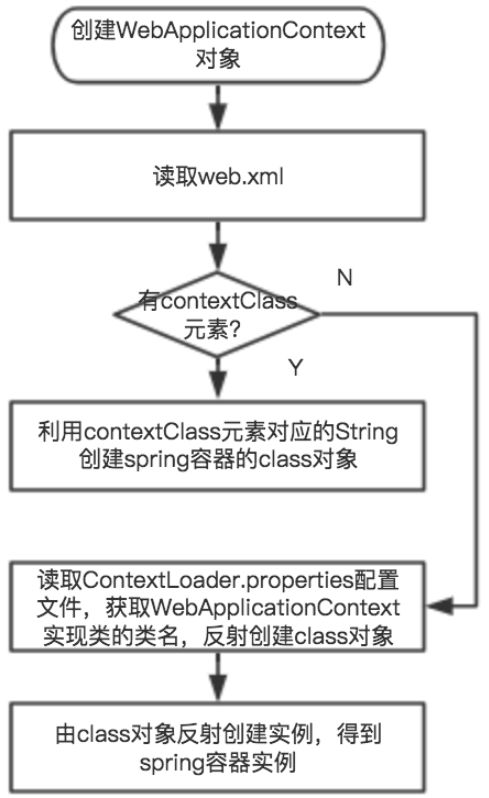
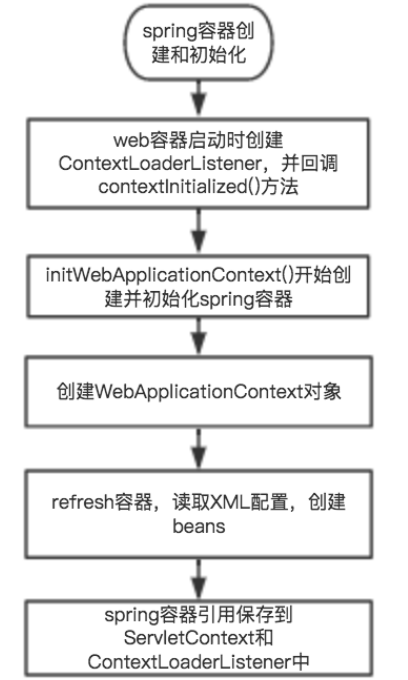
·获取WebApplicationContext实现类的class对象

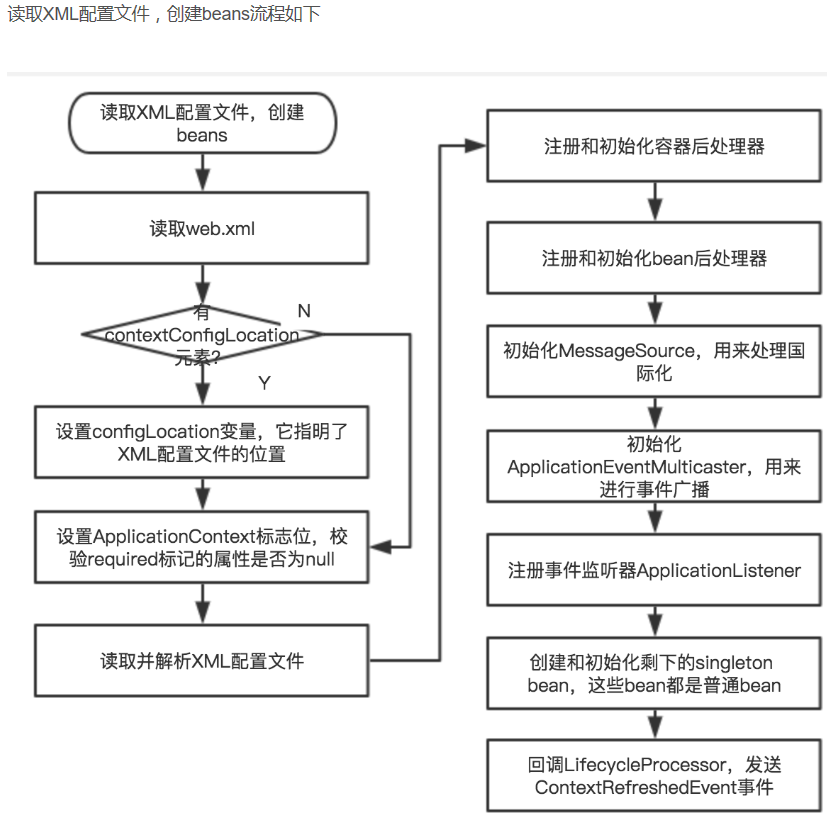
·根据class对象创建实例对象

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

1. Java的HashMap
2. hashmap、concurrenthashmap底层实现和区别
3. **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。**

**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. 快排
2. 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

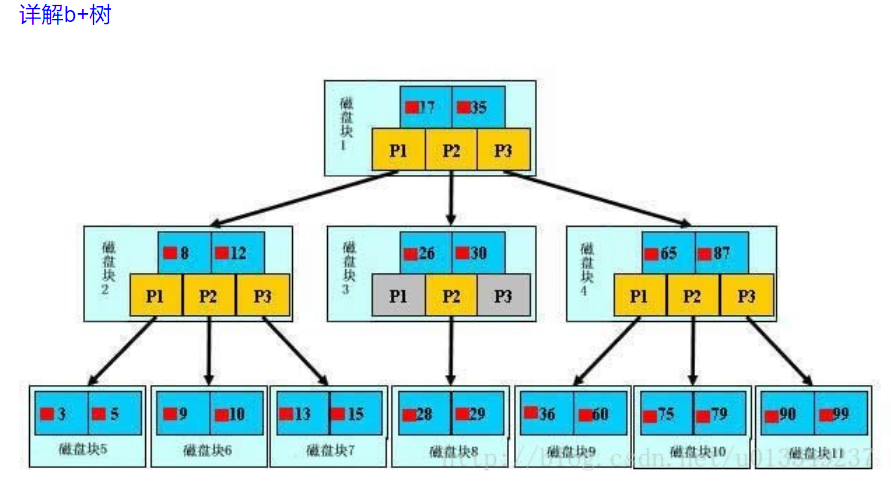
**索引目的**

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。