第1章 算法和数据结构面试题汇总

1.1 请说一说HashMap，SparseArrary原理，SparseArrary相比HashMap的优点、ConcurrentHashMap如何实现线程安全？

这道题想考察什么？

1、HashMap，SparseArrary基础原理？

2、SparseArrary相比HashMap的优点是什么？

3、ConcurrentHashMap如何实现线程安全？

考察的知识点

HashMap，SparseArrary、ConcurrentHashMap

考生如何回答

HashMap和SparseArray，都是用来存储Key-value类型的数据。

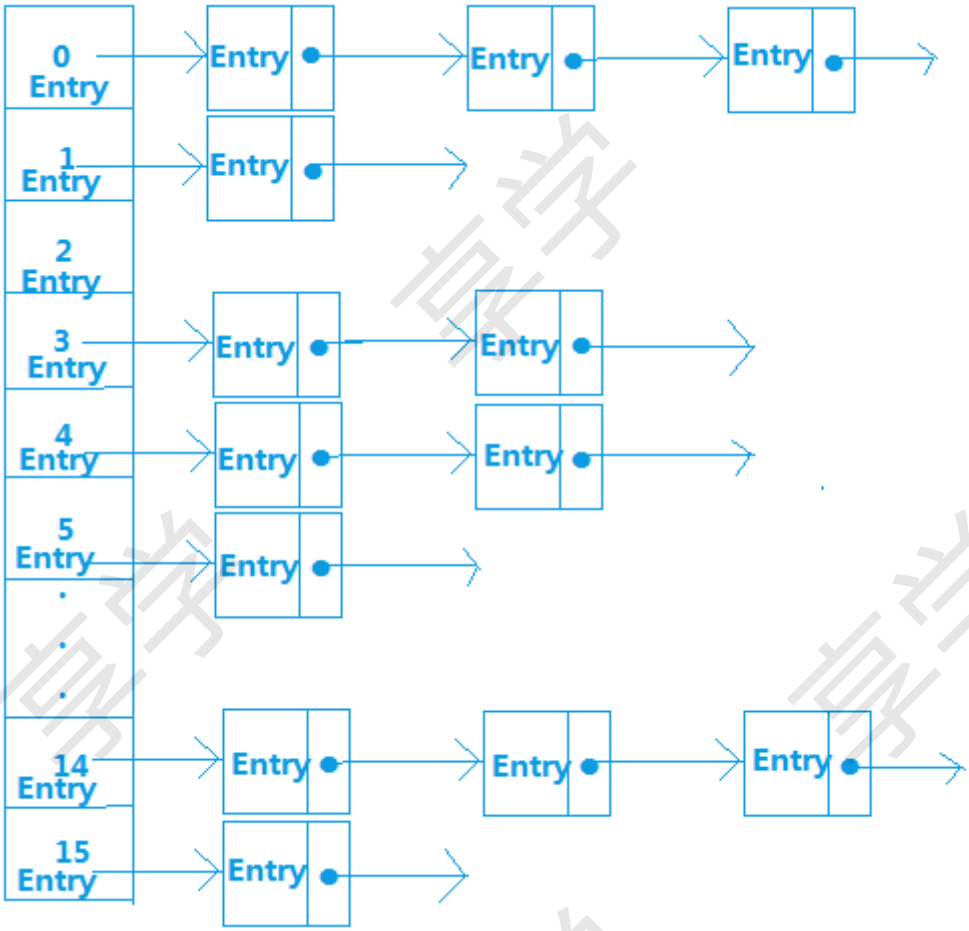
SparseArray和HashMap的区别：

双数组、删除O(1)、二分查找

- 数据结构方面：hashmap用的是链表。sparsearray用的是双数组。
- 性能方面：hashmap是默认16个长度，会自动装箱。如果key是int 的话，hashmap要先封装成Interger。sparseArray的话就就会直接转成int。所以spaseArray用的限制是key是int。数据量小于1k。如果key不是int小于1000的话。可以用Arraymap。

HashMap的基本原理

HashMap内部是使用一个默认容量为16的数组来存储数据的，而数组中每一个元素却又是一个链表的头结点，所以，更准确的来说，HashMap内部存储结构是使用哈希表的拉链结构（数组+链表）。



HashMap中默认的存储大小就是一个容量为16的数组，所以当我们创建出一个HashMap对象时，即使里面没有任何元素，也要分别一块内存空间给它，而且，我们再不断的向HashMap里put数据时，当达到一定的容量限制时，HashMap就会自动扩容。

SparseArray的基本原理

SparseArray比HashMap更省内存，在某些条件下性能更好，主要是因为它避免了对key的自动装箱（int转为Integer类型），它内部则是通过两个数组来进行数据存储的，一个存储key，另外一个存储value，为了优化性能，它内部对数据还采取了压缩的方式来表示稀疏数组的数据，从而节约内存空间，我们从源码中可以看到key和value分别是用数组表示：

```
private int[] mKeys;
private Object[] mValues;
```

我们可以看到，SparseArray只能存储key为int类型的数据，同时，SparseArray在存储和读取数据时候，使用的是二分查找法，

```
public void put(int key, E value) {
        int i = ContainerHelpers.binarySearch(mKeys, mSize, key);
        ...
        }
 public E get(int key, E valueIfKeyNotFound) {
        int i = ContainerHelpers.binarySearch(mKeys, mSize, key);
        ...
        }
```

也就是在put添加数据的时候，会使用二分查找法和之前的key比较当前我们添加的元素的key的大小，然后按照从小到大的顺序排列好，所以，SparseArray存储的元素都是按元素的key值从小到大排列好的。而在获取数据的时候，也是使用二分查找法判断元素的位置，所以，在获取数据的时候非常快，比HashMap快的多，因为HashMap获取数据是通过遍历Entry[]数组来得到对应的元素。

添加数据

```
public void put(int key, E value)
```

删除数据

```
public void remove(int key)
```

获取数据

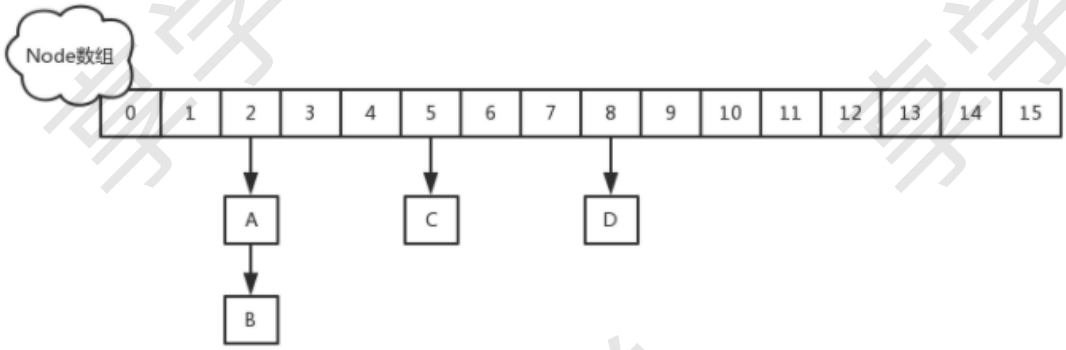
```
public E get(int key)

public E get(int key, E valueIfKeyNotFound)
```

虽说SparseArray性能比较好，但是由于其添加、查找、删除数据都需要先进行一次二分查找，所以在数据量大的情况下性能并不明显，将降低至少50%。满足下面两个条件我们可以使用SparseArray代替HashMap：

- 数据量不大，最好在千级以内
- key必须为int类型，这中情况下的HashMap可以用SparseArray代替：

ConcurrentHashMap基本原理

- JDK1.8的实现降低锁的粒度，JDK1.7版本锁的粒度是基于Segment的，包含多个HashEntry，而JDK1.8锁的粒度就是HashEntry。
- JDK1.8版本的数据结构变得更加简单，使得操作也更加清晰流畅，因为已经使用synchronized来进行同步，所以不需要分段锁的概念，也就不需要Segment这种数据结构了，由于粒度的降低，实现的复杂度也增加了。
- JDK1.8使用红黑树来优化链表，基于长度很长的链表的遍历是一个很漫长的过程，而红黑树的遍历效率是很快的，代替一定阈值的链表。



1.2 请说一说HashMap原理，存取过程，为什么用红黑树，红黑树与完全二叉树对比，HashTab、concurrentHashMap，concurrent包里有啥?

这道题想考察什么？

1、HashMap，HashTab基础原理？

2、ConcurrentHashMap相比HashMap的优点是什么？

3、Concurrent包里面有什么样的的函数？

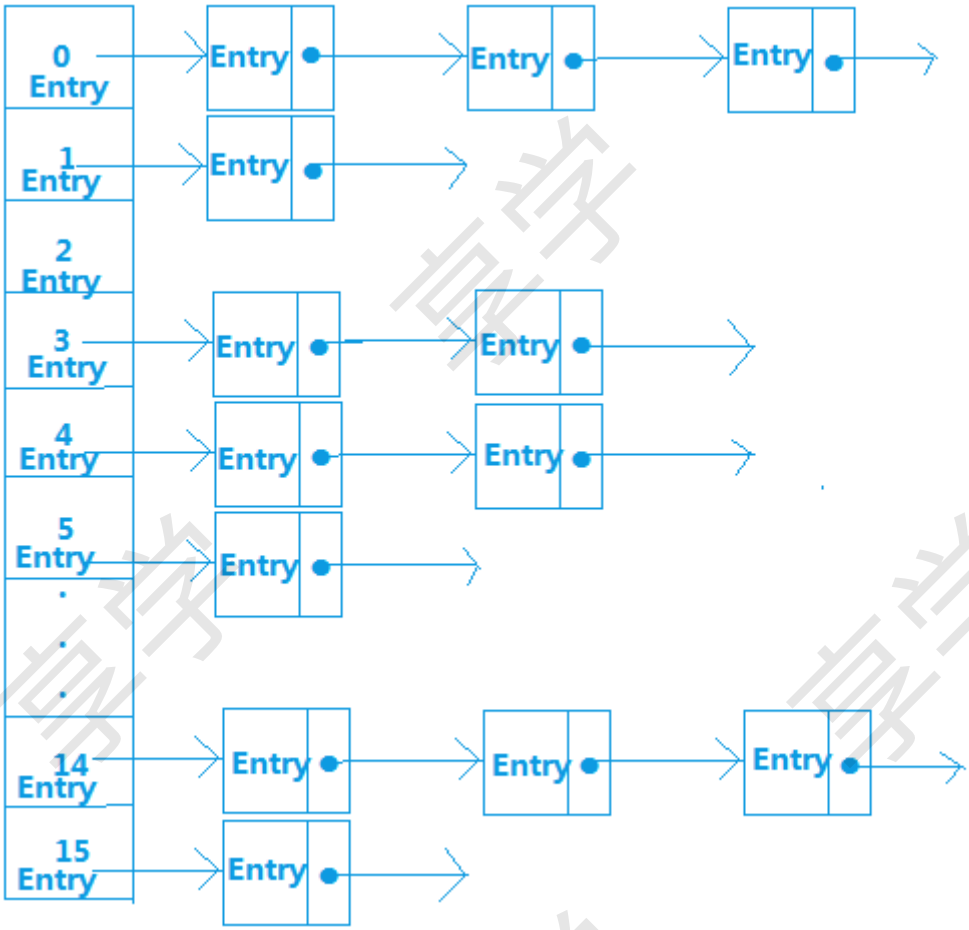
考察的知识点

HashMap，HashTab、ConcurrentHashMap

考生如何回答

HashMap的原理

HashMap内部是使用一个默认容量为16的数组来存储数据的，而数组中每一个元素却又是一个链表的头结点，所以，更准确的来说，HashMap内部存储结构是使用哈希表的拉链结构（数组+链表）。



HashMap中默认的存储大小就是一个容量为16的数组，所以当我们创建出一个HashMap对象时，即使里面没有任何元素，也要分别一块内存空间给它，而且，我们再不断的向HashMap里put数据时，当达到一定的容量限制时，HashMap就会自动扩容。

HashTab的原理

哈希表（Hash table，也叫散列表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

哈希表hash table(key，value) 就是把Key通过一个固定的算法函数既所谓的哈希函数转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将value存储在以该数字为下标的数组空间里。而当使用哈希表进行查询的时候，就是再次使用哈希函数将key转换为对应的数组下标，并定位到该空间获取value，如此一来，就可以充分利用到数组的定位性能进行数据定位。

在使用的时候，有以下几种方式：

- Hashtable 是一个散列表，它存储的内容是键值对(key-value)映射。
- Hashtable 继承于Dictionary，实现了Map、Cloneable、java.io.Serializable接口。
- Hashtable 的函数都是同步的，这意味着它是线程安全的。它的key、value都不可以为null。

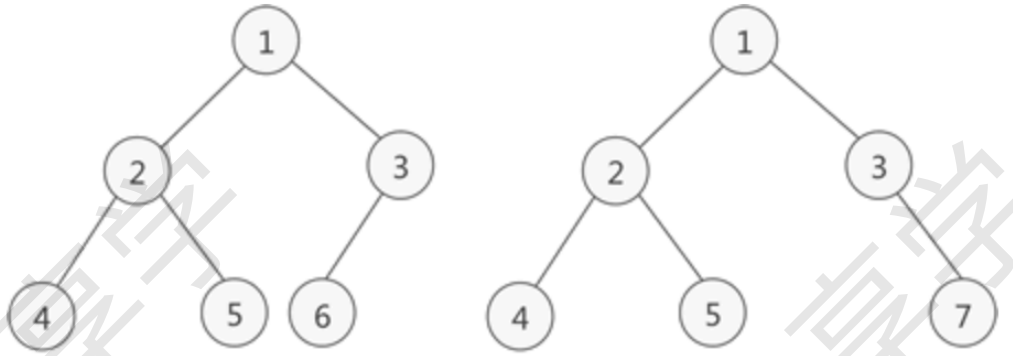
HashMap为什么要用红黑树？红黑树相对完全二叉树有有什么优点？

我们来看看红黑树的主要特性:

- 每个节点都带有颜色属性（颜色为红或黑）的平衡二叉查找树。
- 节点是红色或黑色。
- 根节点是黑色。
- 所有叶子结点都是黑色。
- 每个红色节点必须有两个黑色的子节点（从每个叶子到根的所有路径上不能有两个连续的红色节点）。
- 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

完全二叉树

如果二叉树中除去最后一层节点为满二叉树，且最后一层的结点依次从左到右分布，则此二叉树被称为完全二叉树。



a）完全二叉树　　b）非完全二叉树

红黑树是平衡二叉树的一种，插入时遵循二叉树“左右”定律，

该父节点的左子节点：为**小于父节点**中且子树中最接近父节点值得数。

该父节点的右子节点：为**大于父节点**中且子树中最接近父节点值得数。

Concurrent包里面有什么

ConcorrenctHashMap：将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成，Segment是一种可重入锁ReentrantLock。

CopyOnWriteArrayList：CopyOnWrite并发容器用于读多写少的并发场景，线程安全的写时复制。

ReentrantLock：效果和synchronized一样，都可以同步执行，lock方法获得锁，unlock方法释放锁。ReetrantLockqubie添加了类似锁投票、定时锁等候和可中断锁等候的一些特性争用下的ReentrantLock 实现更具可伸缩性。ReentrantLock支持公平锁，同一个线程可以多次获取同一把锁。ReentrantLock和synchronized都是可重入锁。

CountDownLatch：CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。

Semaphore：它负责协调各个线程, 以保证它们能够正确、合理的使用公共资源。Semaphore可以控制某个资源可被同时访问的个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可实现对资源的保护。

Future：Callable接口可以看作是Runnable接口的补充，call方法带有返回值，并且可以抛出异常。runnable接口实现的没有返回值的并发编程。

CyclicBarrier：这个类是一个同步工具类，它允许一组线程在到达某个栅栏点(common barrier point)互相等待，发生阻塞，直到最后一个线程到达栅栏点，栅栏才会打开，处于阻塞状态的线程恢复继续执行。它非常适用于一组线程之间必需经常互相等待的情况。

1.3 请说一说hashmap put()底层原理,发生冲突时，如何去添加(顺着链表去遍历，挨个比较key值是否一致，如果一致，就覆盖替换，不一致遍历结束后，插入该位置)？

这道题想考察什么？

1、Hashmap的put函数基础原理？

考察的知识点

HashMap底层的源码

考生如何回答

HashMap put函数的底层源码解析

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        if ((tab = table) == null || (n = tab.length) == 0) //首次放入元素时,分配table空间-默认size=16
            n = (tab = resize()).length;

        if ((p = tab[i = (n - 1) & hash]) == null) // 算出新node在table中的位置，若对应位置为null,新建一个node并放入对应位置.
        // 注意:  (n - 1) & hash 求余操作 等价于 hash%n
            tab[i] = newNode(hash, key, value, null);
        else {                                     //在table对应位置有node时
            Node<K,V> e; K k;
            if (p.hash == hash &&                  // key一样 (hash值相同，且key一样，相同实例或者满足Object.equals方法)
                ((k = p.key) == key || (key != null && key.equals(k)))) // 不满足此条件则发生hash碰撞
                e = p;
            else if (p instanceof TreeNode)                // hash碰撞的情况下,用链表解决,链表大于8时，改为红黑树. 当node为treeNode时,putTreeVal->红黑树
```

```
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                //hash
                for (int binCount = 0; ; ++binCount) {
 //用for循环将链表指针后移，将新node在链表加在尾部
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // 当链表长度大于8
时，转成红黑树
                            treeifyBin(tab, hash);
                        break;
                    }
                    if (e.hash == hash &&

                        ((k = e.key) == key || (key != null && key.equals(k))))

                        break;

                    p = e;
                }
            }
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)  //当node中旧的值null或者
onlyIfAbsent==false时,将新的value替换原来的value.
                    e.value = value;
                afterNodeAccess(e); //新node塞入table后做的做的事情，在HashMap中是一
个空方法（LinkedHashMap中有有使用， move node to last）
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)  //新的size大于阈值(默认0.75*table.length)时,扩容.
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

思路如下：

- 对key的hashCode()进行hash后计算数组下标index;
- 如果当前数组table为null，进行resize()初始化；
- 如果没碰撞直接放到对应下标的bucket里；
- 如果碰撞了，且节点已经存在，就替换掉 value；
- 如果碰撞后发现为树结构，挂载到树上。
- 如果碰撞后为链表，添加到链表尾，并判断链表如果过长(大于等于TREEIFY_THRESHOLD，默认8)，就把链表转换成树结构；
- 数据put后,如果数据量超过threshold，就要resize。

通过上面的思路我们可以看到，HashMap的put函数在添加的时候会判断碰撞后是否为链表，如果是链表就添加到链表尾，并判断链表如果过长(大于等于TREEIFY_THRESHOLD，默认8)，就把链表转换成树结构。

1.4 请说一说ArrayList 如何保证线程安全，除了加关键字的方式 ？

这道题想考察什么？

1、ArrayList的底层原理？

考察的知识点

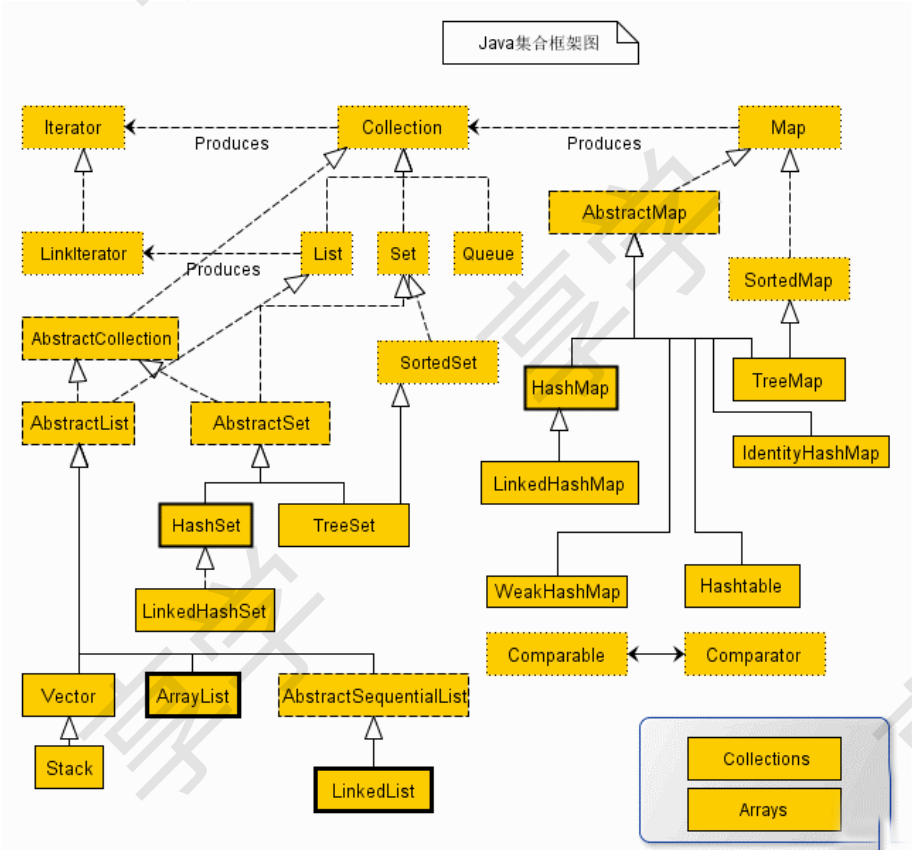
ArrayList底层的源码的理解

考生如何回答

ArrayList 如何保证线程安全

- 继承Arraylist，然后重写或按需求编写自己的方法，这些方法要写成synchronized，在这些synchronized的方法中调用ArrayList的方法。
- 可以使用Collections.synchronizedList()这个函数，使用方法如下

```
List<Map<String,Object>> data=new ArrayList<Map<String,Object>>();
List<Map<String,Object>> data=Collections.synchronizedList(new
ArrayList<Map<String,Object>>());
```



- 使用CopyOnWriteArrayList

```
List<String> k=new CopyOnWriteArrayList<>();
```

1.5 请说一说ArrayList、HashMap、LinkedHashMap ？

这道题想考察什么？

1、ArrayList、HashMap和LinkedHashMap的底层原理？

考察的知识点

ArrayList、HashMap、LinkedHashMap的源码的理解

考生如何回答

ArrayList

ArrayList：底层结构是一个数组，初始长度为10 容量不足时，扩展为原来的1.5倍也就是扩展为15
ArrayList底层是一个双向链表，好处是不用扩容，坏处是当你要寻找第N个元素时，实践复杂度为O(n),就是遍历N个元素去找到他 而ArrayList的时间复杂度是 O(1)

List:元素有序 有序值的是在内存中存放，可重复

HashMap

底层结构是 一个元素为链表的数组 ， 虽然是数组 但是是无序插入数组的。根据哈希值来插入。
当hash相同则需要用到链表结构 ， 把新插入的但 hashcode值相同的 链在之前插入的后面形成链表，当连得太多 就会形成红黑树，新加入的元素形成连头，第一存放在位置上的就成链尾

LinkedHashMap

底层是一个元素为链表的数组 + 元素之间的形成的双向链表 ， 即是单向链表形成的双向链表。 双向链表维护元素的次序，这样使得元素看起来是以插入的顺序保存的。也就是说，当遍历LinkedHashSet集合里元素时，HashSet将会按元素的添加顺序来访问集合里的元素，因此LinkedHashSet可以保证元素按插入顺序输出。

LinkedHashMap底层使用哈希表与双向链表来保存所有元素，它维护着一个运行于所有条目的双向链表（如果学过双向链表的同学会更好的理解它的源代码），此链表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序

- 按插入顺序的链表：在LinkedHashMap调用get方法后，输出的顺序和输入时的相同，这就是按插入顺序的链表，默认是按插入顺序排序。
- 按访问顺序的链表：在LinkedHashMap调用get方法后，会将这次访问的元素移至链表尾部，不断访问可以形成按访问顺序排序的链表。简单的说，按最近最少访问的元素进行排序（类似LRU算法）。

```
public static void main(String[] args) {
    Map<String, String> map = new LinkedHashMap<String, String>();
    map.put("apple", "苹果");
    map.put("watermelon", "西瓜");
    map.put("banana", "香蕉");
    map.put("peach", "桃子");

    Iterator iter = map.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry entry = (Map.Entry) iter.next();
        System.out.println(entry.getKey() + "=" + entry.getValue());
    }
}

// print
apple=苹果
watermelon=西瓜
banana=香蕉
```

```
peach=桃子
```

LinkedList与ArrayList的区别

- LinkedList底层是双向链表
  ArrayList底层是可变数组
- LinkedList不允许随机访问，即查询效率低
  ArrayList允许随机访问，即查询效率高
- LinkedList插入和删除效率快
  ArrayList插入和删除效率低

对于随机访问的两个方法，get和set函数，ArrayList优于LinkedList，因为LinkedList要移动指针；对于新增和删除两个方法，add和remove函数，LinedList比较占优势，因为ArrayList要移动数据。

1.6 请说一说HashMap实现原理，扩容的条件，链表转红黑树的条件是什么 ？

这道题想考察什么？

1、HashMap的底层原理？

2、HashMap的扩容条件以及链表转换红黑树的条件

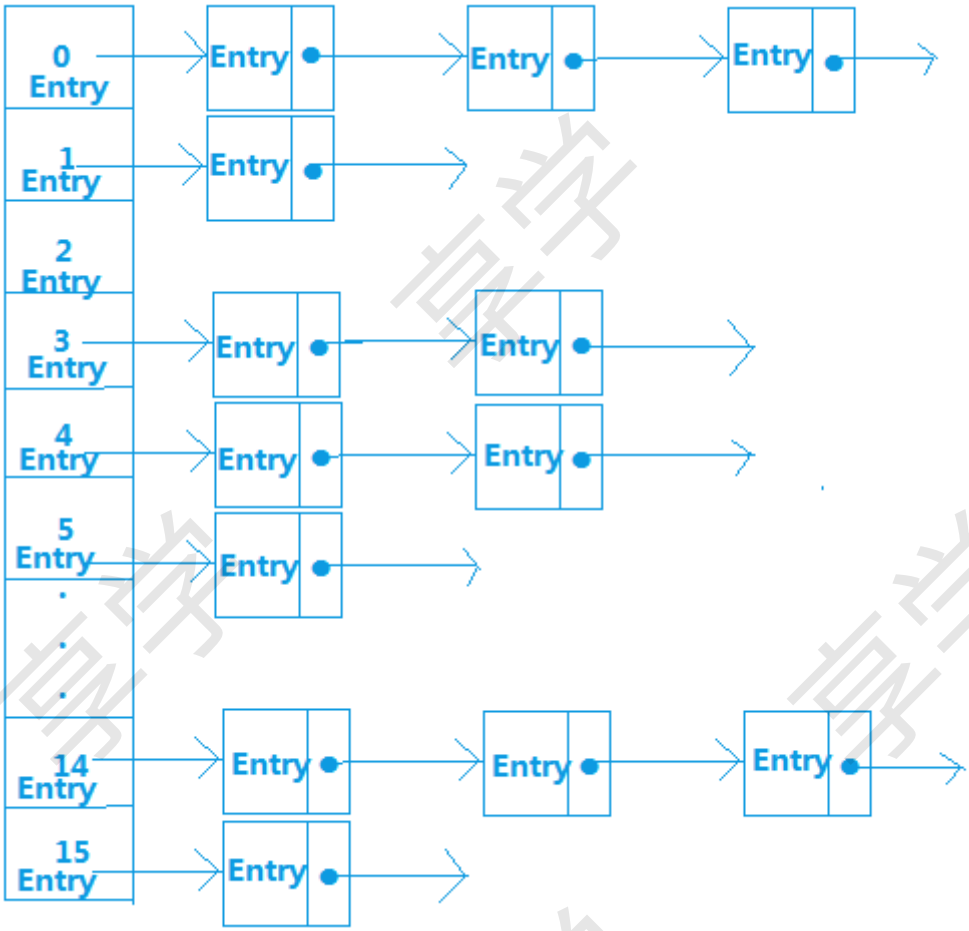
考察的知识点

HashMap原理、HashMap扩容条件的理解

考生如何回答

HashMap实现原理

HashMap内部是使用一个默认容量为16的数组来存储数据的，而数组中每一个元素却又是一个链表的头结点，所以，更准确的来说，HashMap内部存储结构是使用哈希表的拉链结构（数组+链表）。



HashMap中默认的存储大小就是一个容量为16的数组，所以当我们创建出一个HashMap对象时，即使里面没有任何元素，也要分别一块内存空间给它，而且，我们再不断的向HashMap里put数据时，当达到一定的容量限制时，HashMap就会自动扩容。

HashMap扩容条件

HashMap 的实例有两个参数影响其性能：“初始容量” 和 “加载因子”。容量 是哈希表中桶的数量，初始容量 只是哈希表在创建时的容量。加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，并且要存放的位置已经有元素了（hash碰撞），必须满足这两个条件，才要对该哈希表进行 rehash 操作，会将容量扩大为原来两倍。通常，默认加载因子是 0.75, 这是在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本。

我们看下HashMap的put函数方法:

```
public V put(K key, V value) {
    if (table == EMPTY_TABLE) {//如果散列表是空的
        inflateTable(threshold);//会去建一个表
    }
    if (key == null) //hashmap会把key为空的放在数组头部
        return putForNullKey(value);
    int hash = sun.misc.Hashing.singleWordWangJenkinsHash(key);//根据key生成hash值
    int i = indexFor(hash, table.length); //生成散列表中的索引，也就是数组下标
    for (HashMapEntry<K,V> e = table[i]; e != null; e = e.next) {//遍历链表，看是否存在key值一样的对象，如果有的话就替换value值
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
```

```
            return oldValue;
        }
    }

    modCount++;
    //如果没找到key值一样的，就添加
    addEntry(hash, key, value, i);
    return null;
}
```

如何根据hash值生成数组下标，看indexFor()函数：

```
static int indexFor(int h, int length) {
    // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of
2";
    return h & (length-1);
   //为什么长度要是偶数，因为hash值与length的与值不能超过length -1,要不然数组/就越界了，例
如hash值是11000111B，length = 1000B,h & (length-1) = 111B,这样得到数组索引肯定不会越界
了
}



void resize(int newCapacity) {
    HashMapEntry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }
    HashMapEntry[] newTable = new HashMapEntry[newCapacity];//新建一个数组
    transfer(newTable);//完成新旧数组拷贝
    table = newTable;
    threshold = (int)Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1);
}
```

我们在看看最后的扩容步骤：

```
void transfer(HashMapEntry[] newTable) {
    int newCapacity = newTable.length;
    for (HashMapEntry<K,V> e : table) {//遍历整个数组
        while(null != e) {//将同一个位置的元素按链表顺序取出
            HashMapEntry<K,V> next = e.next;//先将当前元素指向的下一个元素存起来，一个
一个存放到新表的位置中，记住不一定是同一位置，因为长度变了
            int i = indexFor(e.hash, newCapacity);//根据新数组长度，重新生成数组索引
            e.next = newTable[i];//将当前位置的元素链表头指向即将新加入的元素，
            newTable[i] = e;//然后放入数组中，完成同一位置元素链表的拼接，最先添加的元素
总在链表末尾
            e = next;//然后继续循环，拿出下一个元素
        }
    }
}
```

链表转红黑树的条件

首先通过源码来分析下问题：

```
 //用来衡量是否要转红黑树的重要参数
 static final int TREEIFY_THRESHOLD = 8;
 //转红黑树需要的最小数组长度
static final int MIN_TREEIFY_CAPACITY = 64;
 for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        // TREEIFY_THRESHOLD - 1=7，也就是说一旦binCount=7时就会执行
下面的转红黑树代码
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }

 final void treeifyBin(Node<K,V>[] tab, int hash) {
        int n, index; Node<K,V> e;
        //这里的tab指的是本HashMap中的数组，n为数字长度，如果数组为null或者数组长度小于64
        if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
            //则调用resize()方法直接扩容，不转红黑树
            resize();
        //否则走转红黑树逻辑
        else if ((e = tab[index = (n - 1) & hash]) != null) {
            TreeNode<K,V> hd = null, tl = null;
            do {
                TreeNode<K,V> p = replacementTreeNode(e, null);
                if (tl == null)
                    hd = p;
                else {
                    p.prev = tl;
                    tl.next = p;
                }
                tl = p;
            } while ((e = e.next) != null);
            if ((tab[index] = hd) != null)
                hd.treeify(tab);
        }
    }
```

可以知道【TREEIFY_THRESHOLD - 1】=7，所以binCount=7时才会转红黑树，而binCount初始赋值是0，++count是先加再用，所以其实binCount是从1开始的，1，2，3，4，5，6，7，每一个数值对应的都会创建一个newNode，所以binCount到数值7时，创建了7个新的Node节点，但是情不要忘记，我们创建的节点都是p.next，也就是p的后继节点，所以加上原来的p节点，也可以理解成是链表首节点，7+1=8，就是8个节点，所以链表里元素数目到8个时，会开始转红黑树。

> 总结：
>
> 当链表元素数目到8个，同时HashMap的数组长度要大于64，链表才会转红黑树，否则都是做扩容。

1.7 请说一说二叉树遍历步骤？

这道题想考察什么？

1、二叉树的基本原理和遍历的方法？

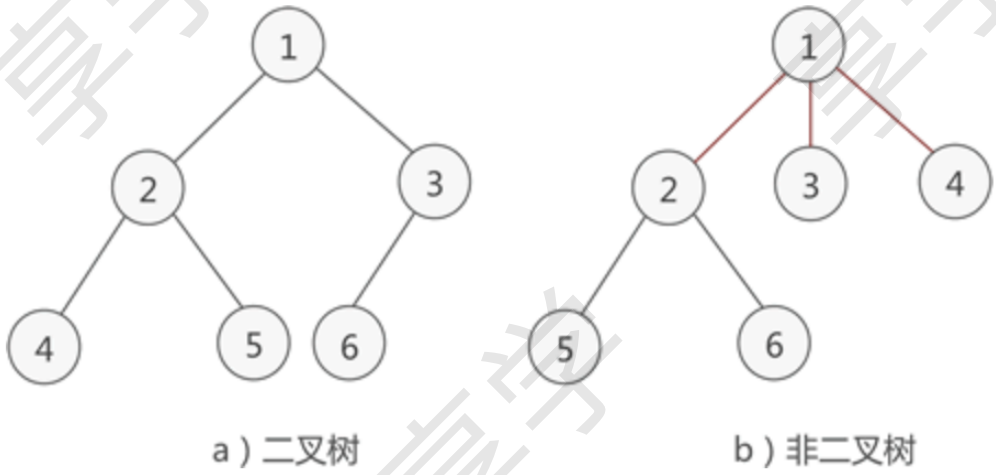
考察的知识点

二叉树遍历的基本流量、二叉树的基本原理

考生如何回答

二叉树的基本概念

简单地理解，满足以下两个条件的树就是二叉树：

1. 本身是有序树；
2. 树中包含的各个节点的度不能超过 2，即只能是 0、1 或者 2；
3. 

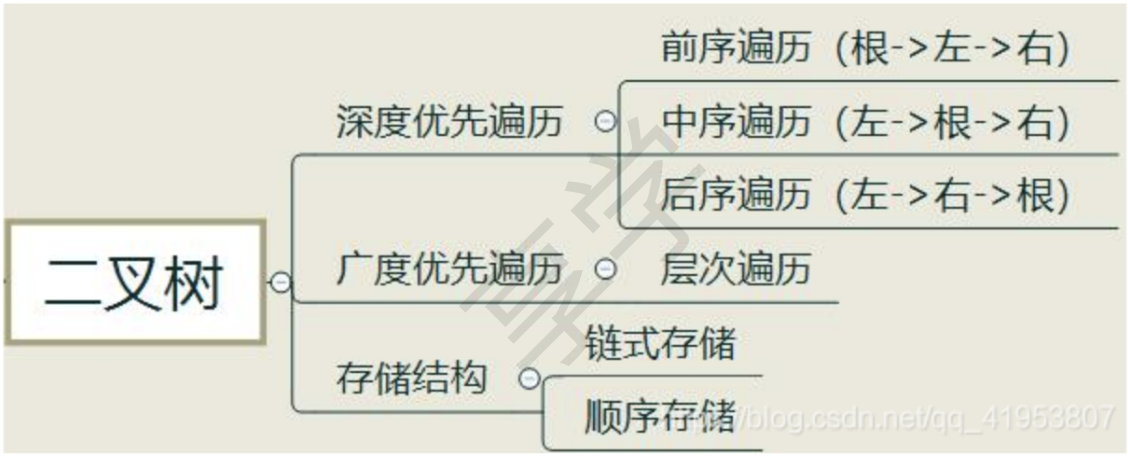
a）二叉树　　b）非二叉树

二叉树的性质

二叉树具有以下几个性质：

1. 二叉树中，第 i 层最多有 2i-1 个结点。
2. 如果二叉树的深度为 K，那么此二叉树最多有 2K-1 个结点。
3. 二叉树中，终端结点数（叶子结点数）为 n0，度为 2 的结点数为 n2，则 n0=n2+1。

二叉树的遍历



二叉树的遍历方式主要有：先序遍历、中序遍历、后序遍历、层次遍历。先序、中序、后序其实指的是父节点被访问的次序。若在遍历过程中，父节点先于它的子节点被访问，就是先序遍历；父节点被访问的次序位于左右孩子节点之间，就是中序遍历；访问完左右孩子节点之后再访问父节点，就是后序遍历。不论是先序遍历、中序遍历还是后序遍历，左右孩子节点的相对访问次序是不变的，总是先访问左孩子节点，再访问右孩子节点。而层次遍历，就是按照从上到下、从左向右的顺序访问二叉树的每个节点。

先序遍历

代码如下：

```
//filename: BinTreeNode.h
template <typename T>
void travPre_R(BinTreeNode<T> * root) {//二叉树先序遍历算法（递归版）
    if (!root) return;
    cout << root->data;
    travPre_R(root->LeftChild);
    travPre_R(root->RightChild);
}
```

中序遍历

代码如下：

```
template <typename T>
void travIn_R(BinTreeNode<T> * root) {//二叉树先序遍历算法（递归版）
    if (!root)
        return;
    travPre_R(root->LeftChild);
    cout << root->data;
    travPre_R(root->RightChild);
}
```

1.8采用递归和非递归对二叉树进行遍历？

这道题想考察什么？

1、二叉树的基本原理和遍历的方法？

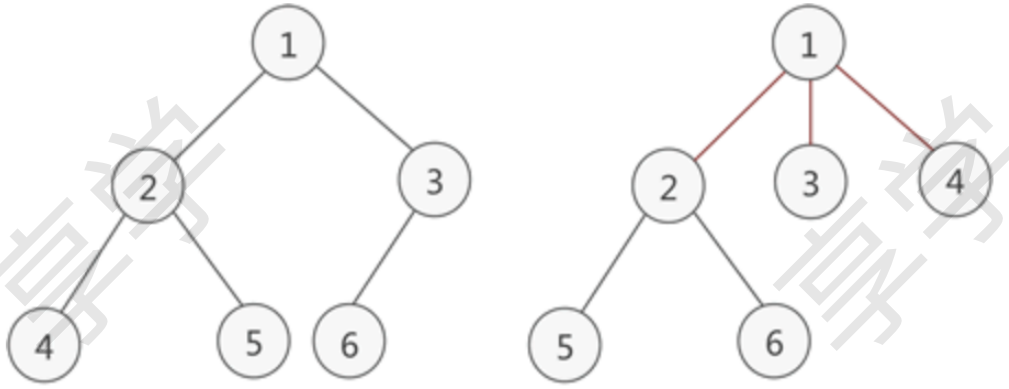
考察的知识点

二叉树遍历的基本概念、二叉树的基本原理

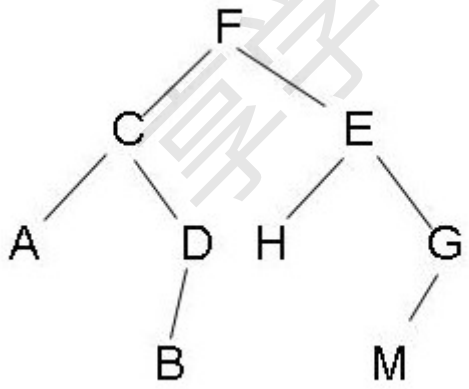
考生如何回答

二叉树的基本概念

简单地理解，满足以下两个条件的树就是二叉树：

1. 本身是有序树；
2. 树中包含的各个节点的度不能超过 2，即只能是 0、1 或者 2；
3. 

a）二叉树　　b）非二叉树



二叉树的遍历

- 前序遍历：每个树的遍历顺序为：根节点→左节点→右节点。上图的前序遍历输出为：FCADBEHGM
- 中序遍历：每个树的遍历顺序为：左节点→根节点→右节点。上图的前序遍历输出为：ACBDFHEMG
- 后序遍历：每个树的遍历顺序为：左节点→右节点→根节点。上图的前序遍历输出为：ABDCHMGEF

前序遍历

递归法：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
```

```
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private List<Integer> res = new ArrayList<>();
    public List<Integer> preorderTraversal(TreeNode root) {
        //中 --> 左 --> 右
        preorder(root);
        return res;
    }

    private void preorder(TreeNode node){
        if(node == null) return;
        res.add(node.val);
        preorder(node.left);
        preorder(node.right);
    }
}
```

非递归法：
===基本的算法思想===
创建一个栈，用来储存遍历的轨迹：
1.如果栈不为空则储存当前栈顶元素的值，并弹栈；
2.如果栈顶元素存在右儿子，将右儿子压入；
3.如果栈顶元素有左儿子，将左儿子压入
4.重复1直至栈为空

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode cur = root;
        while(cur != null || !stack.isEmpty()){
            if(cur != null){
                res.add(cur.val);
                stack.push(cur);
                cur = cur.left;
            }else{
                cur = stack.pop();
                cur = cur.right;
            }
        }
        return res;
    }
}
```

中序遍历

递归法：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private List<Integer> res = new ArrayList<>();
    public List<Integer> inorderTraversal(TreeNode root) {
        inorder(root);
        return res;
    }

    public void inorder(TreeNode node){
        if(node == null) return;
        inorder(node.left);
        res.add(node.val);
        inorder(node.right);
    }
}
```

非递归法：

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode cur = root;
        while(cur != null || !stack.isEmpty()){
            //压栈
            if(cur != null){
                stack.push(cur);
                cur = cur.left;
            }else{
                //左边已经存完，弹栈
                cur = stack.pop();
                res.add(cur.val);
                cur = cur.right;
            }
        }
        return res;
    }
}
```

后序遍历

递归法：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
```

```
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private List<Integer> res = new ArrayList<>();
    public List<Integer> postorderTraversal(TreeNode root) {
        //左右中
        postorder(root);
        return res;
    }

    private void postorder(TreeNode node){
        if(node == null) return;
        postorder(node.left);
        postorder(node.right);
        res.add(node.val);
    }
}
```

非递归法：
基本的算法思想：
使用栈来记录遍历轨迹，并使用一个变量来储存上一次方法的元素，当当前元素左右儿子为空或当前元素已经在上一轮访问过（即上一次方法访问的元素为当前访问元素的节点），则栈顶元素出栈。

```
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root == null) return res;

        Stack<TreeNode> stack = new Stack<>();
        TreeNode pre = null;
        stack.push(root);

        while(!stack.isEmpty()){
            TreeNode cur = stack.peek();
            if((cur.left == null && cur.right == null)
              || (pre != null && (cur.right == pre || cur.left == pre))){
                res.add(cur.val);
                pre = cur;
                stack.pop();
            }else{
                if(cur.right != null) stack.add(cur.right);
                if(cur.left != null) stack.add(cur.left);
            }
        }

        return res;
    }
}
```

1.9对称和非对称加密，MD5的原理？

这道题想考察什么？

1、对称和非对称加密算法的原理？

2、MD5的基本的概念和原理？

考察的知识点

MD5算法原理、对称和非对称加密算法

考生如何回答

对称和非对称加密算法的基本概念

对称加密和非对称加密的基本概念

对称加密指的就是加密和解密使用同一个秘钥，所以叫做对称加密。对称加密只有一个秘钥，作为私钥。常见的对称加密算法：DES，AES，3DES等等。非对称加密指的是：加密和解密使用不同的秘钥，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。私钥加密的信息，只有公钥才能解密。常见的非对称加密算法：RSA，ECC。

对称加密和分对称加密算法的区别

- 对称加密：加密解密用同一个密钥，被黑客拦截不安全 。
- 非对称加密：公钥加密，私钥解密；公钥可以公开给别人进行加密，私钥永远在自己手里，非常安全，黑客拦截也没用，因为私钥未公开。

指加密和解密使用不同密钥的加密算法，也称为公私钥加密。假设两个用户要加密交换数据，双方交换公钥，使用时一方用对方的公钥加密，另一方即可用自己的私钥解密。常见的非对称加密算法：RSA、DSA（数字签名用）、ECC（移动设备用）、Diffie-Hellman、El Gamal。

- **RSA：**由 RSA 公司发明，是一个支持变长密钥的公共密钥算法，需要加密的文件块的长度也是可变的。
- **DSA（Digital Signature Algorithm）**：数字签名算法，是一种标准的 DSS（数字签名标准
- **ECC（Elliptic Curves Cryptography）**：椭圆曲线密码编码学。

  ECC和RSA相比，在许多方面都有对绝对的优势，主要体现在以下方面：
  （1）抗攻击性强。相同的密钥长度，其抗攻击性要强很多倍。
  （2）计算量小，处理速度快。ECC总的速度比RSA、DSA要快得多。
  （3）存储空间占用小。ECC的密钥尺寸和系统参数与RSA、DSA相比要小得多，意味着它所占的存贮空间要小得多。这对于加密算法在IC卡上的应用具有特别重要的意义。
  （4）带宽要求低。当对长消息进行加解密时，三类密码系统有相同的带宽要求，但应用于短消息时ECC带宽要求却低得多。带宽要求低使ECC在无线网络领域具有广泛的应用前景。

对称加密和分对称加密算法的区别

MD5的基本概念

MD5加密

MD5本身是一个128位的0/1比特。一般被表示为16进制的字符串。4个比特位组成一个16进制字符，因此常常能见到的是（128/4=）32个16进制字符组成的字符串 4951 dd1c bff8 cbbe 4cd4 475c a939 fc8b，当然它实质是一种**消息摘要算法**。

MD5加密的特点：

1. 不可逆运算
2. 对不同的数据加密的结果是定长的32位字符（不管文件多大都一样）

3. 对相同的数据加密，得到的结果是一样的（也就是复制）。
4. 抗修改性 : 信息“指纹”，对原数据进行任何改动,哪怕只修改一个字节,所得到的 MD5 值都有很大区别.
5. 弱抗碰撞 : 已知原数据和其 MD5 值,想找到一个具有相同 MD5 值的数据(即伪造数据)是非常困难的.
6. 强抗碰撞: 想找到两个不同数据,使他们具有相同的 MD5 值,是非常困难的