**1、final、finally、finalize的区别**

- final：修饰符（关键字）有三种用法：如果一个类被声明为final，意味着它不能再派生出新的子类，即不能被继承，将变量声明为final，可以保证它们在使用中不被改变，被声明为final的变量的引用中只能读取不可修改。被声明为final的方法也同样只能使用，不能在子类中被重写。

- finally：通常放在try…catch…的后面构造总是执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。

- finalize：Object类中定义的方法，Java中允许使用finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize()方法可以整理系统资源或者执行其他清理工作。

**2、static关键字**

static是一个修饰符，用于修饰成员。由static修饰的变量，方法称为静态变量，方法。

1、static关键字修饰类

普通类是不允许声明为静态的，只有内部类才可以



2、static关键字修饰方法

可以直接通过类名来进行调用

3、static关键字修饰变量

被static修饰的成员变量叫做静态变量，也叫做类变量，说明这个变量是属于这个类的，而不是属于是对象，没有被static修饰的成员变量叫做实例变量，说明这个变量是属于某个具体的对象的。

4、static关键字修饰代码块

静态代码块在类第一次被载入时执行。类初始化的顺序：父类静态代码块->子类静态代码块->父类构造函数->子类构造函数

**注意事项：**

1、虽然静态成员也可以使用对象.静态成员，但这样不好区分静态成员和非静态成员。

2、静态方法中不可以使用this关键字。

3、静态方法中不可以直接调用非静态方法

4、不能将方法中的局部变量设为静态的

5、可以使用静态区域先执行类的初始化。

**3、String类为何是final修饰的？它的底层数据结构**

1、安全

class Test{

public static void main(String[] args){

HashSet<StringBuilder> hs=new HashSet<StringBuilder>();

StringBuilder sb1=new StringBuilder("aaa");

hs.add(sb1); //这时候HashSet里是{"aaa","aaabbb"}

StringBuilder sb3=sb1;

sb3.append("bbb"); //这时候HashSet里是{"aaabbb","aaabbb"}

System.out.println(hs);

}

}

因为 StringBuilder 没有不可变性的保护， sb3 直接在原先 "aaa" 的地址上改。导致 sb1 的值也变了。这时候， HashSet 上就出现了两个相等的键值 "aaabbb" 。破坏了 HashSet键值的唯一性 。所以千万不要用可变类型做 HashMap和HashSet键值。

2、线程安全

不可变对象不能被写，所以线程安全。

3、效率高

**底层数据结构：**

String类存储字符串的方式是通过char型数组存储的：

public final class String …… {

/\*\* The value is used for character storage. \*/

private final char value[];

…………

}

如果是字符串常量形式的声明首先会查看常量池中是否存在这个常量，如果存在就不会创建新的对象，否则在在常量池中创建该字符串并创建引用，此后不论以**此种方式**创建多少个相同的字符串都是指向这一个地址的引用，而不再开辟新的地址空间放入相同的数据；但是字符串对象每次new都会在堆区形成一个新的内存区域并填充相应的字符串，不论堆区是否已经存在该字符串！

**4、String、StringBuffer与StringBuilder的区别**

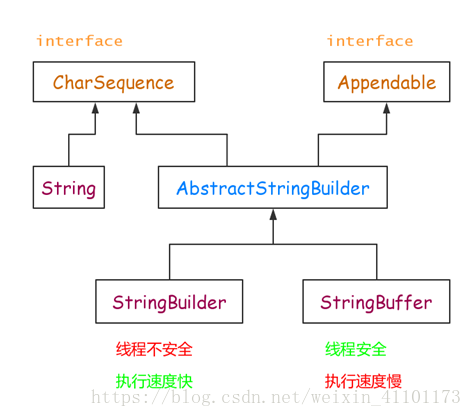
String—字符串常量

String的值是不可变的，这就导致每次对String的操作都会生成**新的String对象**

StringBuffer和 StringBuilder—字符串变量

StringBuffer和StringBuilder类的对象能够被多次的修改，并且不产生新的未使用对象

StringBuilder的方法不是线程安全的（不能同步访问）。由于StringBuilder相较于StringBuffer有速度优势，**所以多数情况下建议使用StringBuilder类**。然而在应用程序要求线程安全的情况下，则必须使用StringBuffer类。StringBuffe很多方法都是synchronized修饰的：length()、append()。

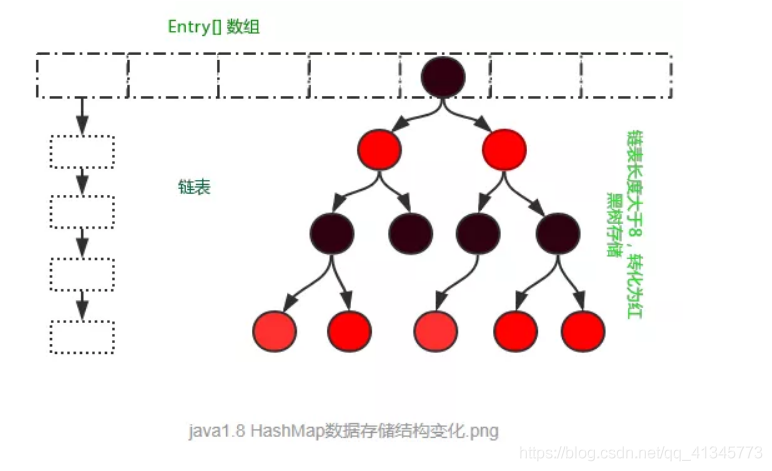


**5、transient关键字**

将不需要序列化的属性前添加关键字transient，序列化对象的时候，这个属性就不会被序列化。

**6、红黑树的数据结构，左旋和右旋是怎样实现的**

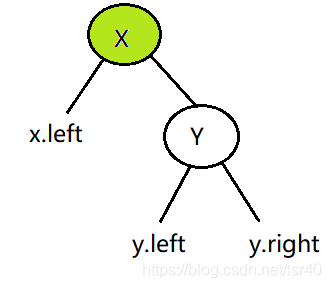
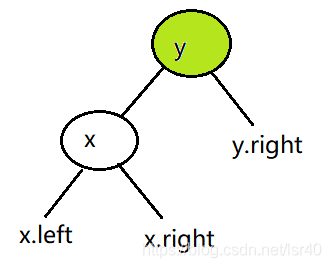
当冲突大于等于8时，就会将冲突的Entry转换为红黑树进行存储。

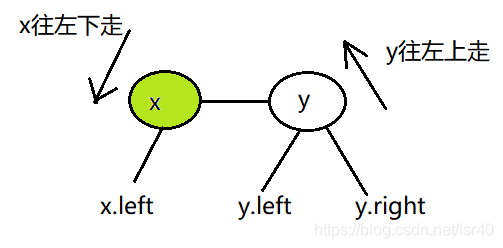
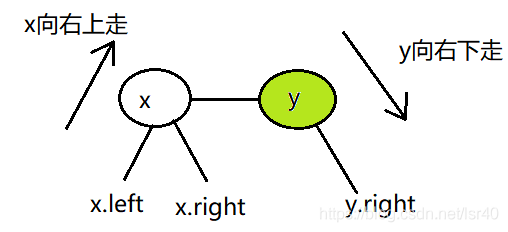


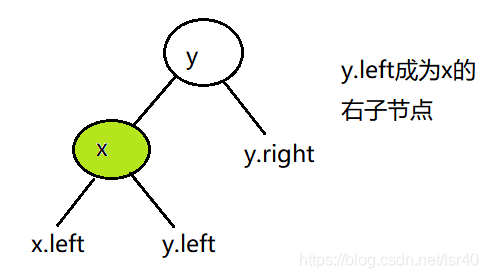
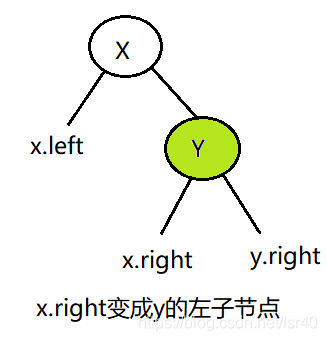
红黑树是一种自平衡二叉查找树，是一种特化的AVL树（[平衡二叉树](https://baike.baidu.com/item/%E5%B9%B3%E8%A1%A1%E4%BA%8C%E5%8F%89%E6%A0%91/10421057)）每一颗红黑树都是一颗二叉排序树。

1）每个结点要么是红的，要么是黑的。   
2）根结点是黑的。   
3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。   
4）如果一个结点是红的，那么它的俩个儿子都是黑的。   
5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

-1.左旋 -2.右旋







**7、HashMap底层数据结构**

HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对。Entry是HashMap中的一个静态内部类。代码如下

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;//存储指向下一个Entry的引用，单链表结构

int hash;

Entry(int h, K k, V v, Entry<K,V> n) {

value = v;

next = n;

key = k;

hash = h;

}

简单来说，**HashMap由数组+链表组成的**，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前entry的next指向null）,那么查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。

**9、HashMap扩容机制**

变量size，它记录HashMap的底层数组中已用槽的数量；

变量threshold，它是HashMap的阈值，（threshold = 容量\*负载因子）

变量loadFactor，负载因子，默认为0.75

当发生哈希冲突并且size大于阈值的时候，需要进行数组扩容，扩容时，需要新建一个长度为之前数组2倍的新的数组，然后将当前的Entry数组中的元素全部传输过去，扩容后的新数组长度为之前的2倍。

**10、为什么HashMap长度为2的次幂**

HashMap的数组长度一定保持2的次幂，比如16的二进制表示为 10000，那么length-1就是15，二进制为01111，同理扩容后的数组长度为32，二进制表示为100000，length-1为31，二进制表示为011111。从下图可以我们也能看到这样会保证低位全为1，而扩容后只有一位差异，也就是多出了最左位的1，会使得获得的数组索引index更加均匀。如果低位全部为1，那么对于hash低位部分来说，任何一位的变化都会对结果产生影响，也就是说，要得到index=21这个存储位置，hash的低位只有这一种组合。减小哈希冲突出现的可能性。

**11、HashMap put元素的过程**

当准备添加一个key-value对时，首先通过hash(key)方法计算hash值，然后通过indexFor(hash,length)求该key-value对的存储位置，计算方法是hash& (length-1)，这就保证每一个key-value对都能存入HashMap中，当计算出的位置相同时，由于存入位置是一个链表，则把这个key-value对插入链表头。

**12、HashMap的负载因子为何默认为0.75**

负载因子是0.75的时候，空间利用率比较高，而且避免了相当多的Hash冲突，使得底层的链表或者是红黑树的高度比较低，提升了空间效率。

**13、HashMap和HashTable的区别**

HashMap只实现Map接口，HashTable继承Dictionary类，实现Map接口。

HashMap是非线程安全的，HashTable是线程安全的，用Synchronize修饰方法。

HashMap计算hash：key的hash值高16位不变，低16位与高16位异或作为key最终的hash值。

Hashtable计算hash：直接使用对象自身的hash。

HashMap计算index = hash & (length-1)

Hashtable计算index = (hash & 0x7FFFFFFF) % tab.length

Hashtable默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。

**14、Java的四种引用**

**1、强引用**

是指创建一个对象并把这个对象赋给一个引用变量。

Object object =new Object();

String str ="hello";

强引用有引用变量指向时永远不会被垃圾回收，JVM宁愿抛出OutOfMemory错误也不会回收这种对象。

**2、软引用**

如果一个对象具有软引用，内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。软引用可用来实现内存敏感的高速缓存,比如网页缓存、图片缓存等。使用软引用能防止内存泄露，增强程序的健壮性。

**3、弱引用**

弱引用也是用来描述非必需对象的，当JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。在java中，用java.lang.ref.WeakReference类来表示。

**4、虚引用**

虚引用和前面的软引用、弱引用不同，它并不影响对象的生命周期。在java中用java.lang.ref.PhantomReference类表示。如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收。

**15、ArrayList、LinkedList、Vector**

**ArrayList**

作为List的主要实现类；主要特点有线程不安全，但是执行效率高效，底层实现是数组结构（Collections中定义了synchronizedList(List list)将此ArrayList转化为线程安全的）默认构造方法创建一个空数组，第一次添加元素时候，扩展容量为10，之后的扩充算法：原来数组大小+原来数组的一半（也就是1.5倍）。

**LinkedList**

对于频繁的插入、删除操作，我们建议使用此类，因为它的执行效率高；但是内存消耗比ArrayList大；底层实现的双向链表实现。

**Vector**

List的古老实现类；线程安全的，效率低；底层使用数组实现的（如今逐渐被淘汰）默认构造方法创建一个大小为10的对象数组，如果增量为0的情况下每次扩容容量是翻倍，即为原来的2倍，而当增量>0的时候，扩充为原来的大小加增量。

**16、equals与==的区别**

基本数据类型byte,short,char,int,long,float,double,boolean。他们之间的比较，应用双等号（==）,比较的是他们的值。

equals()是Object里的方法。在Object的equals中，就是使用==来进行比较，比较的是内存地址。与==不同的是，在某些Object的子类中，覆盖了equals()方法，比如String中的equals()方法比较两个字符串对象的内容是否相同。

**17、Java如何实现序列化,有什么意义?**

**1、实现序列化:**

1)让类实现Serializable接口,该接口是一个标志性接口,标注该类对象是可被序列

2)然后使用一个输出流来构造一个对象输出流并通过writeObject(Obejct)方法就可以将实现对象写出

3)如果需要反序列化,则可以用一个输入流建立对象输入流,然后通过readObeject方法从流中读取对象

**2、作用:**

1)序列化就是一种用来处理对象流的机制,所谓对象流也就是将对象的内容进行流化,可以对流化后的对象进行读写操作,也可以将流化后的对象传输与网络之间;

2)为了解决对象流读写操作时可能引发的问题(如果不进行序列化,可能会存在数据乱序的问题)

3)序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆

**18、JDBC是什么**

Java数据库连接，（ JDBC）是[Java语言](https://baike.baidu.com/item/Java%E8%AF%AD%E8%A8%80)中用来规范客户端程序如何来访问数据库的[应用程序接口](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F%E6%8E%A5%E5%8F%A3/10418844)，提供了诸如查询和更新数据库中数据的方法。

**JDBC驱动程序：**

**JDBC-ODBC桥**

这种类型的驱动把所有JDBC的调用传递给ODBC，再让后者调用数据库本地驱动代码

**本地API驱动**

这种类型的驱动通过客户端加载数据库厂商提供的本地代码库（C/[C++](https://baike.baidu.com/item/C%2B%2B)等）来访问数据库，而在[驱动程序](https://baike.baidu.com/item/%E9%A9%B1%E5%8A%A8%E7%A8%8B%E5%BA%8F)中则包含了Java代码。

**[网络协议](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE" \t "_blank)驱动**

这种类型的驱动给客户端提供了一个网络API，客户端上的JDBC[驱动程序](https://baike.baidu.com/item/%E9%A9%B1%E5%8A%A8%E7%A8%8B%E5%BA%8F)使用[套接字](https://baike.baidu.com/item/%E5%A5%97%E6%8E%A5%E5%AD%97)（Socket）来调用服务器上的[中间件](https://baike.baidu.com/item/%E4%B8%AD%E9%97%B4%E4%BB%B6)程序，后者在将其请求转化为所需的具体API调用。

**本地协议驱动**

这种类型的驱动使用Socket，直接在客户端和数据库间通信。

**19、int 和 Integer的关系**

int 是java的基本数据类型。

Integer 继承了Object类，是对象类型，是 int 的包装类。

**区别：**

1、值的存储

int 存储在栈中

Integer 对象的引用存储在栈空间中，对象的数据存储在堆空间中。

2、初始化

int 初始化值为0。

Integer 初始化值为null。

3、传参

int 是值传递，栈中的数据不可变。

Integer 对象是引用传递，引用不可变，但是引用指向的堆空间地址中的值是可以改变的。

4、泛型支持

泛型不支持int，但是支持Integer。

5、运算

int 可以直接做运算，是类的特性。

Integer 的对象可以调用该类的方法，但是在拆箱之前不能进行运算，需要转化为基本类型int。

**20、反射**

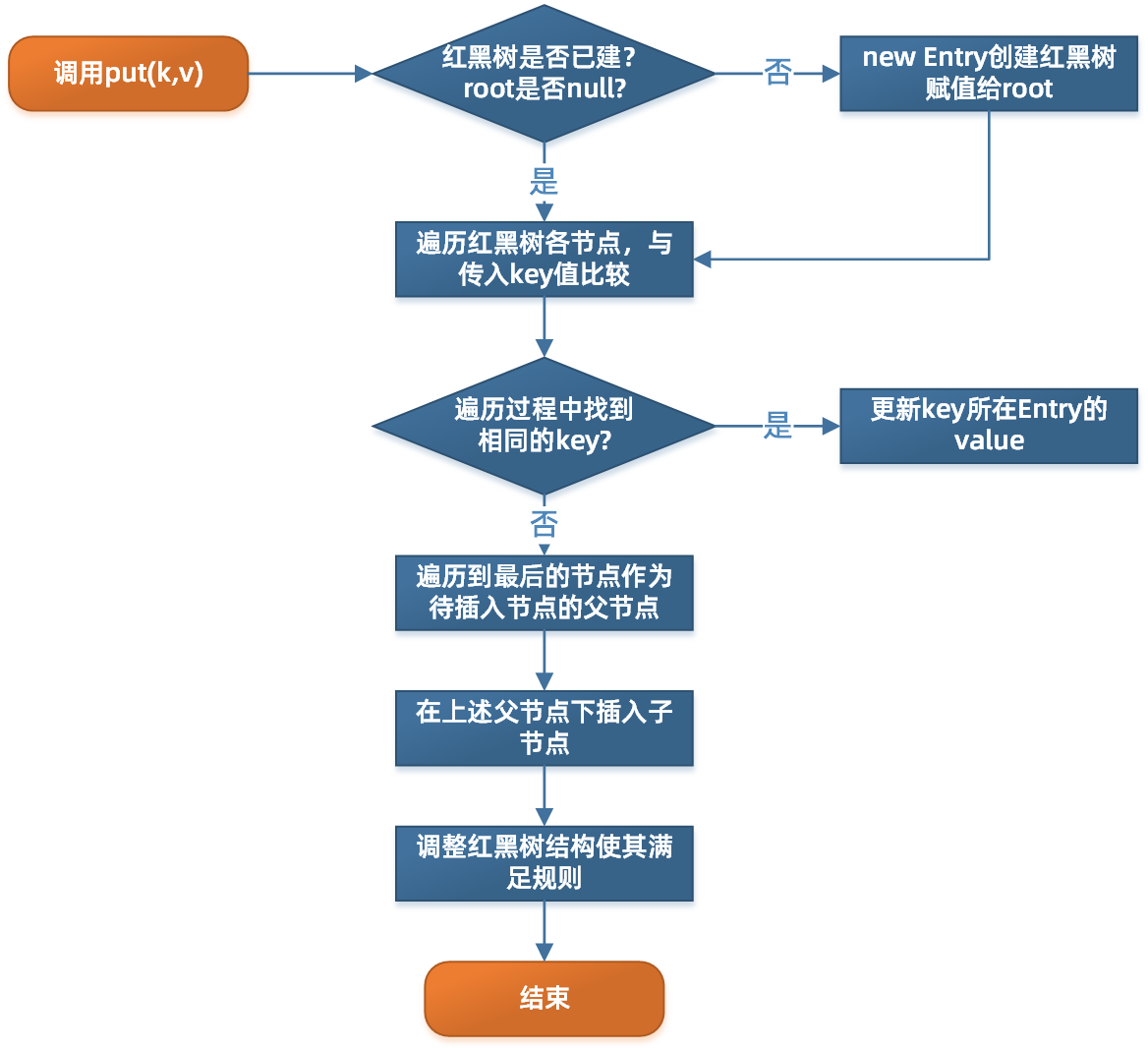
[JAVA反射机制](https://baike.baidu.com/item/JAVA%E5%8F%8D%E5%B0%84%E6%9C%BA%E5%88%B6/6015990)是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

在日常的第三方应用开发过程中，经常会遇到某个类的某个成员变量、方法或是属性是私有的或是只对系统应用开放，这时候就可以利用Java的反射机制通过反射来获取所需的私有成员或是方法。

**21、TreeMap**

TreeMap存储K-V键值对，通过红黑树（R-B tree）实现；天然支持排序，默认情况下通过Key值的自然顺序进行排序；

**put：**



**remove：**

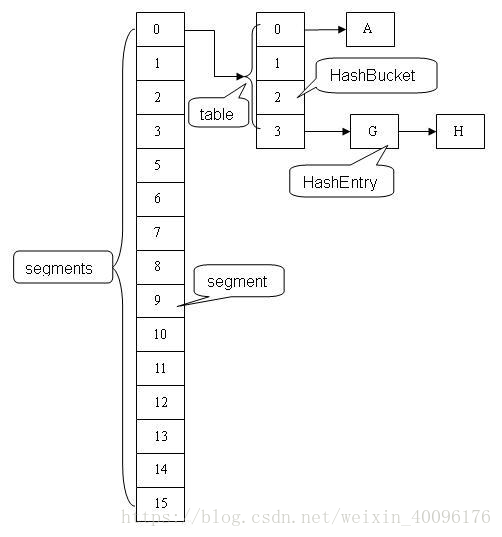
1、删除的是根节点，则直接将根节点置为null;

2、待删除节点的左右子节点都为null，删除时将该节点置为null;

3、待删除节点的左右子节点有一个有值，则用有值的节点替换该节点即可；

4、待删除节点的左右子节点都不为null，则找前驱或者后继，将前驱或者后继的值复制到该节点中，然后删除前驱或者后继（前驱：左子树中值最大的节点，后继：右子树中值最小的节点）；

**22、ConcurrentHashMap**



在ConcurrentHashMap中有个重要的概念就是Segment。我们知道HashMap的结构是数组+链表形式，从图中我们可以看出其实每个segment就类似于一个HashMap。Segment包含一个HashEntry数组，数组中的每一个HashEntry既是一个键值对，也是一个链表的头节点。在ConcurrentHashMap中有2的N次方个Segment，共同保存在一个名为segments的数组当中。可以说，ConcurrentHashMap是一个二级哈希表。在一个总的哈希表下面，有若干个子哈希表。  
为什么说ConcurrentHashMap的性能要比HashTable好，HashTables是用全局同步锁，而CconurrentHashMap采用的是锁分段，每一个Segment就好比一个自治区，读写操作高度自治，Segment之间互不干扰。

**Case1:不同Segment的并发写入**  
不同Segment的写入是可以并发执行的。  
**Case2:同一Segment的一写一读**  
同一Segment的写和读是可以并发执行的。  
**Case3:同一Segment的并发写入**  
Segment的写入是需要上锁的，因此对同一Segment的并发写入会被阻塞。  
由此可见，ConcurrentHashMap当中每个Segment各自持有一把锁。在保证线程安全的同时降低了锁的粒度，让并发操作效率更高。  
**Get方法：**

1.为输入的Key做Hash运算，得到hash值。

2.通过hash值，定位到对应的Segment对象

3.再次通过hash值，定位到Segment当中数组的具体位置。

**Put方法：**

1.为输入的Key做Hash运算，得到hash值。

2.通过hash值，定位到对应的Segment对象

3.获取可重入锁

4.再次通过hash值，定位到Segment当中数组的具体位置。

5.插入或覆盖HashEntry对象。

6.释放锁。

**size方法：**

1.遍历所有的Segment。

2.把Segment的元素数量累加起来。

3.把Segment的修改次数累加起来。

4.判断所有Segment的总修改次数是否大于上一次的总修改次数。如果大于，说明统计过程中有修改，重新统计，尝试次数+1；如果不是。说明没有修改，统计结束。

\*\*在JDK1.8中ConcurrentHashMap的实现方式有了很大的改变，在JDK1.7中采用的是Segment + HashEntry，而Sement继承了ReentrantLock，所以自带锁功能，而在JDK1.8中则取消了Segment，作者认为Segment太过臃肿，采用CAS + Synchronized保证线程安全

**23、Collection和Collections**

java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。

java.util.Collections则是集合类的一个工具类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

**24、Exception和Error**

定义一个基类java.lang.Throwable作为所有异常的超类，有两个子类Error和Exception，分别表示错误和异常

1、Error是程序无法处理的错误，比如OutOfMemoryError

2、Exception是程序本身可以处理的异常，这种异常分两大类运行时异常和非运行时异常。程序中应当尽可能去处理这些异常。

**运行时异常**都是RuntimeException类及其子类异常，如NullPointerException、IndexOutOfBoundsException等。程序中可以选择捕获处理，也可以不处理。

**非运行时异常**是RuntimeException以外的异常，类型上都属于Exception类及其子类。如IOException、SQLException等以及用户自定义的Exception异常。对于这种异常，JAVA编译器强制要求我们必需对出现的这些异常进行catch并处理

**25、interface和abstract**

1、抽象类可以有构造方法，接口中不能有构造方法。

2、抽象类中可以有普通成员变量，接口中没有普通成员变量

3、抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。

4、抽象类中的抽象方法的访问类型可以是public，protected和（默认类型,虽然eclipse下不报错，但应该也不行），但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。

5、抽象类中可以包含静态方法，接口中不能包含静态方法

6、抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。

7、一个类可以实现多个接口，但只能继承一个抽象类。

**25、Java多态性**

多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为**编译**时的多态性和**运行**时的多态性。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：1). 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2). 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

**26、Java IO与NIO**

| **IO** | **NIO** |
| --- | --- |
| 面向字节流 | 面向缓冲区 |
| 阻塞 | 非阻塞 |

**1、面向流与面向缓冲**

     Java IO和NIO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

**2、阻塞与非阻塞IO**

     Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

**28、泛型实现方式**

泛型是通过**类型擦除**来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息List<String>不能转为List<Object>，因为泛型的类型不一样，这种转换只能在子类与父类之间转换，虽然Object是String的父类，但是List<Object>和List<String>在编译器看来，是两种完全不同的东西

**29、Hash冲突解决方法**

1、开放定址法：

线性探测

平方探测

2、再哈希法

3、链地址法

**30、各种排序比较**

