## 一、java线程安全

线程安全类：当多个线程访问某个类时,这个类始终保持正确行为那么这个类是线程安全的

无状态对象:既不包含任何域,也不包含对任何其他域的引用。使用的局部变量存于栈中的局部变量里,只能由正在执行的线程访问

Java提供了一种内置的锁机制来支持原子性：Synchronized关键字。同步代码块包括两部分：一个作为锁的对象引用,一个由该锁保护的代码块。以synchronized修饰的方法的锁就是方法调用所在的对象,静态的synchronized方法以Class对象作为锁。

线程进入同步代码块之前会自动获得锁,并在退出同步代码块时自动释放锁。

重入：如果某个线程视图获得一个已经由它自己持有的锁,那么这个请求就会成功。实现方法：为每个锁关联一个获取计数值和一个所有者线程；当计数值为0时这个锁被认为是没有被任何线程持有。当线程请求一个未被持有的锁时,jvm将记下锁的持有者并将获取计数值置位1.如果同一个线程再次获取这个锁,计数值将+1,当线程退出同步代码块时计数器会相应递减,计数值为0时这个锁会被释放。

当执行较长时间的计算或者可能无法快速完成的操作时（网络I/O或者控制台I/O）,一定不要持有锁。

Volatile变量：确保将变量的更新操作通知其他线程.当变量声明为volatile类型后,编译器与运行时都会注意到这个变量是共享的,因此不会讲该变量上的操作与其他内存操作一起重排序。Volatile变量不会缓存在寄存器或者其他处理器不可见的地方,因此读取volatile类型的变量时总会返回最新写入的值。访问volatile变量时不会执行加锁操作,也不会使执行线程阻塞,因此volatile是一种比sychronized关键字更轻量级的同步机制

当且仅当满足以下所有条件时,才应该使用volatile变量：

对变量的写入操作不依赖变量当前的值,或者只有单个线程更新变量的值

该变量不会与其他状态变量一起纳入不变性条件中

访问变量时不需要加锁

ThreadLocal类:维持线程封闭性的方法。这个类能使线程中的某个值与保存值的对象关联起来,ThreadLocal类提供了get与set等访问接口或方法,这些方法为每个使用该变量的线程都存有一份独立的副本,因为get总返回由当前执行线程在调用set时设置的最新值。

ThreadLocal对象通常用于放置对可变的单实例变量或全局变量进行共享。比如将JDBC链接保存到ThreadLocal对象中,每个线程都会拥有属于自己的连接。

## 二、java集合类

Java集合类有两大分支:Collection和Map

### Collection

|  
├List 列表  
 │├LinkedList   
 │├ArrayList 通过ensureCapacity(int n)方法可提高ArrayList的初始化速度  
 │└Vector  
 │　└Stack  
└Set 集合 不允许有重复元素

├LinkedHashSet外部按成员的插入顺序遍历成员

├treeset外部有序地遍历成员;附加实现了SortedSet,支持子集等要求顺序的操作

├hashset

**LinkedList和ArrayList**

LinkedList底层采用双向循环列表实现，对列表中任何位置的成员的增加和删除支持较好，但对基于索引的成员访问支持性能较差；可以使用LinkedList来实现队列和栈。

ArrayList是实现了基于动态数组的数据结构，对于随机访问(get()方法) 它的速度要比LinkedList快。LinkedList中的get方法是按照顺序从列表的一端开始检查，直到另外一端。

1．对ArrayList和LinkedList而言，在列表末尾增加一个元素所花的开销都是固定的。对ArrayList而言，主要是在内部数组中增加一项，指向所添加的元素，偶尔可能会导致对数组重新进行分配；而对LinkedList而言，这个开销是统一的，分配一个内部Entry对象。

2．在ArrayList的中间插入或删除一个元素意味着这个列表中剩余的元素都会被移动；而在LinkedList的中间插入或删除一个元素的开销是固定的。

3．LinkedList不支持高效的随机元素访问。

4．ArrayList的空间浪费主要体现在在list列表的结尾预留一定的容量空间,而LinkedList的空间花费则体现在它的每一个元素都需要消耗相当的空间.(保存下一个元素的位置)

**HashSet和TreeSet**

HashSet是由一个hash表来实现的，因此，它的元素是无序的。add()，remove()，contains()方法的时间复杂度是O(1)。添加到 HashSet 的对象需要采用恰当分配散列码的方式来实现hashCode() 方法。虽然大多数系统类覆盖了 Object 中缺省的hashCode()实现，但创建要添加到 HashSet 的类时，需要覆盖 hashCode()。

另一方面，TreeSet是由一个树形的结构来实现的，它里面的元素是有序的。因此，add()，remove()，contains()方法的时间复杂度是O(logn)。添加到TreeSet中的元素必须是可排序的。

### Map

├Hashtable key和value都不允许出现null值，Hashtable 中的方法是同步的

├TreeMap支持对键有序地遍历

├HashMap中null可以作为键，这样的键只有一个；

└LinkedHashMap 保留了键的插入顺序

HashMap输出是无序的，这个无序不是每次遍历的结果顺序不一样，而是与插入顺序不一样。

对hashmap的按值排序的实现：

public static void sortHashMap(Map<Integer , String> hashmap){

//第一步，用HashMap构造一个LinkedList

Set<Entry<Integer , String>> sets = hashmap.entrySet();

LinkedList<Entry<Integer , String>> linkedList = new LinkedList<Entry<Integer , String>>(sets);

//用Collections的sort方法排序

Collections.sort(linkedList , new Comparator<Entry<Integer , String>>(){

public int compare(Entry<Integer , String> o1, Entry<Integer , String> o2) {

// TODO Auto-generated method stub

/\*String object1 = (String) o1.getValue();

String object2 = (String) o2.getValue();

return object1.compareTo(object2);\*/

return o1.getValue().compareTo(o2.getValue());

}

});

//第三步，将排序后的list赋值给LinkedHashMap

Map<Integer , String> map = new LinkedHashMap();

for(Entry<Integer , String> entry : linkedList){

map.put(entry.getKey(), entry.getValue());

}

}

Hashmap按键排序的实现：TreeMap默认是按键排序的，默认升序，所以先用comparator对象构造一个TreeMap，再将要排序的HashMap添加到TreeMap中。

//第一步：先创建一个TreeMap实例，构造函数传入一个Comparator对象。

TreeMap<Integer , String> treemap = new TreeMap<Integer , String>(new Comparator<Integer>(){

@Override

public int compare(Integer o1,Integer o2) {

// TODO Auto-generated method stub

return Integer.compare(o1, o2);

}

});

//第二步：将要排序的HashMap添加到我们构造的TreeMap中。

treemap.putAll(hashmap);

### 支持并发的容器

Map:hashtable,该类的方法都是通过synchronized来进行方法层次同步的,以达到线程安全的作用。性能更好的Map为ConcurrenthashMap,内部使用lock而不是synchronize，相比Hashtable效率上提高很大同时兼顾线程安全。ConcurrentHashMap允许多个修改并发操作,因为ConcurrentHaspMap采用了内部使用段机制,将ConcurrentHashMap分成了很多段,只要不是在同一个小段上写就可以并发。

Collection：CopyOnWriteArraySet、CopyOnWriteArrayList/Vector.使用CopyOnWrite机制,即当往容器里添加元素的时候,先对这个容器进行一次复制,对副本进行写操作。写操作结束后,将原容器的引用指向新副本容器,就完成了写的刷新。缺点是如果容器较大会占用较多内存,还有在写操作时只能在最终结束后使数据同步,不能实时同步。适用于读操作多写操作少的场景。

ConcurrentSkipListMap 对应的非并发容器：TreeMap

目标：代替synchronizedSortedMap(TreeMap)

原理：Skip list（跳表）是一种可以代替平衡树的数据结构，默认是按照Key值升序的。Skip list让已排序的数据分布在多层链表中，以0-1随机数决定一个数据的向上攀升与否，通过空间来换取时间的一个算法。ConcurrentSkipListMap提供了一种线程安全的并发访问的排序映射表。内部是SkipList（跳表）结构实现，在理论上能够在O(log(n))时间内完成查找、插入、删除操作。

ConcurrentSkipListSet对应的非并发容器：TreeSet

目标：代替synchronizedSortedSet

原理：内部基于ConcurrentSkipListMap实现

ConcurrentLinkedQueue不会阻塞的队列；对应的非并发容器：Queue

原理：基于链表实现的FIFO队列（LinkedList的并发版本）

LinkedBlockingQueue、ArrayBlockingQueue、PriorityBlockingQueue

对应的非并发容器：BlockingQueue

特点：拓展了Queue，增加了可阻塞的插入和获取等操作

原理：通过ReentrantLock实现线程安全，通过Condition实现阻塞和唤醒

实现类：

LinkedBlockingQueue：基于链表实现的可阻塞的FIFO队列

ArrayBlockingQueue：基于数组实现的可阻塞的FIFO队列

PriorityBlockingQueue：按优先级排序的队列

haspMap的遍历:

for(Entry entry:map.entrySet()){

K key = entry.getKey();

V value = entry.getValue();

}

entrySet()方法返回了set<Map.entry<K,V>>,即Map内部接口Map.entry的集合，Map.entry提供了getKey()和getValue()方法

## 三、java克隆

浅克隆 对象中其他对象的成员变量会克隆引用；

深克隆 会对对象成员变量也克隆(实现深克隆的方法：序列化与反序列化(需要实现Serializable接口))

public Object deepClone()

{

//将对象写到流里

ByteArrayOutoutStream bo=new ByteArrayOutputStream();

ObjectOutputStream oo=new ObjectOutputStream(bo);

oo.writeObject(this);

//从流里读出来

ByteArrayInputStream bi=new ByteArrayInputStream(bo.toByteArray());

ObjectInputStream oi=new ObjectInputStream(bi);

return(oi.readObject());

}

这样做的前提是对象以及对象内部所有引用到的对象都是可串行化的

## 四、String、StringBuffer与StringBuilder

String为字符串常量,StringBuffer与StringBuilder是字符串变量。

当声明一个String变量时:String s = “a” s是指向堆内存中的引用，存储了“a”在堆中的地址；s本身存储在栈内存中。当对s重新赋值时:s=“b” 此时s将指向“b”在堆中的地址。此时堆中将同时存在“a”和“b”对象，不过“a”如果不被其他变量引用则会被虚拟机垃圾回收掉。JVM为了提高性能和减少内存开销，内部维护了一个字符串常量池，每当创建字符串常量时，JVM首先检查字符串常量池，如果常量池中已经存在，则返回池中的字符串对象引用，否则创建该字符串对象并放入池中。

1.String中使用 + 字符串连接符进行字符串连接时，连接操作最开始时如果都是字符串常量，编译后将尽可能多的直接将字符串常量连接起来，形成新的字符串常量参与后续连接；

2.接下来的字符串连接是从左向右依次进行，对于不同的字符串，首先以最左边的字符串为参数创建StringBuilder对象，然后依次对右边进行append操作，最后将StringBuilder对象通过toString()方法转换成String对象。

也就是说String c = "xx" + "yy " + a + "zz" + "mm" + b; 实质上的实现过程是： String c = new StringBuilder("xxyy").append(a).append("zz").

append("mm").append(b).toString();

由于得出结论：当使用+进行多个字符串连接时，实际上是产生了一个StringBuilder对象和一个String对象。

StringBuffer是线程安全的，StringBuilder是非线程安全的。执行速度方面StringBuilder快于StringBuffer。(都实现了append()方法)

如果操作少量的数据使用String，单线程操作字符缓冲区下大量数据用StringBuilder，多线程字符串缓冲区下操作大量数据用StringBuffer.