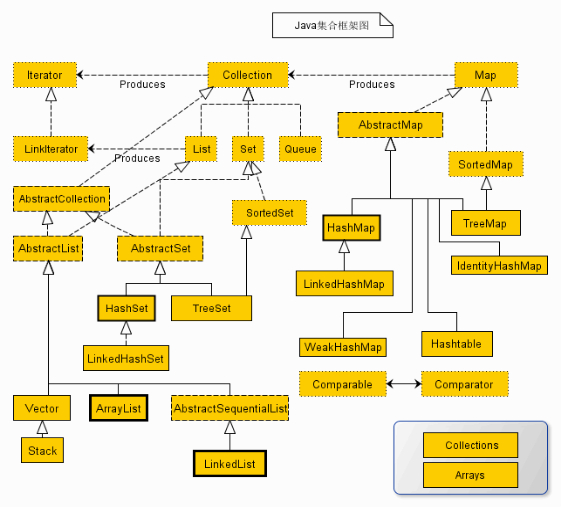
## 1: 集合

以下是Java集合框架图：实线边框的是实现类，折线边框的是抽象类，点线边框的是接口；



Collection是Java.util包中的一个集合类的根接口，Java中没有提供这个接口的直接的实现类。但是却让其被继承产生了两个接口，就是Set和List。Set中不能包含重复的元素。List是一个有序的集合，可以包含重复的元素，提供了按索引访问的方式。

Iterator，所有的集合类，都实现了Iterator接口，这是一个用于遍历集合中元素的接口，主要包含以下三种方法：1.hasNext()是否还有下一个元素。2.next()返回下一个元素。3.remove()删除当前元素。

对于集合，关注的点主要有四点: 1、是否允许空; 2、是否允许重复数据; 3、是否有序，有序的意思是读取数据的顺序和存放数据的顺序是否一致; 4、是否线程安全 ;

Map是Java.util包中的另一个接口，它和Collection接口没有关系，是相互独立的，但是都属于集合类的一部分。Map包含了key-value对。Map不能包含重复的key，但是可以包含相同的value。

### ArrayList

ArrayList就是一个以数组形式实现的集合；允许为空(add null size也加1)、允许重复

数据、有序、非线程安全；

元素: private transient Object[] elementData

作用: ArrayList是基于数组的一个实现，elementData就是底层的数组；

元素: private int size ()

作用: ArrayList里面元素的个数，size是按照add、remove方法的次数进行自增或自减；

优点：

1. ArrayList底层以数组实现，是一种随机访问模式，再加上它实现了RandomAccess接

口，因此查找也就是get的时候非常快；

2、ArrayList在顺序添加一个元素的时候非常方便，只是往数组里面添加了一个元素而已

缺点：

1、删除元素的时候，涉及到一次元素复制，如果要复制的元素很多，那就会比较耗费性能；

2、插入元素的时候，涉及到一次元素复制，如果要复制的元素很多，那就会比较耗费性能；

因此，ArrayList比较适合顺序添加、随机访问的场景。

#### 添加元素与扩容

调用add方法的时候就是给elementData的某个位置添加了一个数据；构造ArrayList的时候，默认的底层数组大小是10；底层数组的大小不够时，底层基于动态数组(底层的数组大小并不是固定的)实现的ArrayList会根据添加的元素大小进行一个判断，不够的话就动态扩容，jdk1.7中扩容newCapacity = oldCapacity + (oldCapacity >> 1) ，即扩容1.5倍；

最后调用到的是Arrays的copyOf方法，将元素组里面的内容复制到新的数组里面去；

(注： << 左移运算符，相当于乘以2； >> 右移运算符，相当于除以2； >>> 无符号右移，忽略符号位，空位都以0补齐)

#### 删除元素

ArrayList支持两种删除方式：1、按照下标删除；2、按照元素删除，这会删除ArrayList

中与指定要删除的元素匹配的第一个元素；其实做的事情就是两件: 1、把指定元素后面位置的所有元素，利用System.arraycopy方法整体向前移动一个位置；2、最后一个位置的元素指定为null，这样让gc可以去回收它；

#### 插入元素

插入的时候，按照指定位置，把从指定位置开始的所有元素利用System,arraycopy方法做一个整体的复制，向后移动一个位置（当然先要用ensureCapacity方法进行判断，加了一个元素之后数组会不会不够大），然后指定位置的元素设置为需要插入的元素，完成了一次插入的操作。

#### 1.1.4 线程非安全

ArrayList是线程非安全的，一个方法是用Collections.synchronizedList方法把ArrayList变成一个线程安全的List，比如：

List<String> synchronizedList = Collections.synchronizedList(list);

synchronizedList.add("aaa");

synchronizedList.add("bbb");

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

{

    System.out.println(synchronizedList.get(i));

}

另一个方法就是Vector，它是ArrayList的线程安全版本，其实现90%和ArrayList都完全一样，区别在于：1、Vector是线程安全的，ArrayList是线程非安全的；2、Vector可以指定增长因子，如果该增长因子指定了，那么扩容的时候会每次新的数组大小会在原数组的大小基础上加上增长因子；如果不指定增长因子，那么就给原数组大小；

#### 1.1.5 transient

ArrayList的elementData是用transient修饰的？

ArrayList的定义：

public class ArrayList<E> extends AbstractList<E>

        implements List<E>, RandomAccess, Cloneable, java.io.Serializable

看到ArrayList实现了Serializable接口，这意味着ArrayList是可以被序列化的，用transient修饰elementData意味着不希望elementData数组被序列化。因为序列化ArrayList的时候，ArrayList里面的elementData未必是满的，比方说elementData有10的大小，但是只用了其中的3个，那么是否有必要序列化整个elementData呢？显然没有这个必要，因此ArrayList中重写了writeObject方法：

private void writeObject(java.io.ObjectOutputStream s)

        throws java.io.IOException{

// Write out element count, and any hidden stuff

int expectedModCount = modCount;

s.defaultWriteObject();

        // Write out array length

       s.writeInt(elementData.length);

    // Write out all elements in the proper order.

for (int i=0; i<size; i++)

           s.writeObject(elementData[i]);

    if (modCount != expectedModCount) {

           throw new ConcurrentModificationException();

    }

}

每次序列化的时候调用这个方法，先调用defaultWriteObject()方法序列化ArrayList中的非transient元素，elementData不去序列化它，然后遍历elementData，只序列化那些有的元素，这样：

1、加快了序列化的速度

2、减小了序列化之后的文件大小

不失为一种聪明的做法，如果以后开发过程中有遇到这种情况，也是值得学习、借鉴的一种思路。

#### 1.1.6 重要

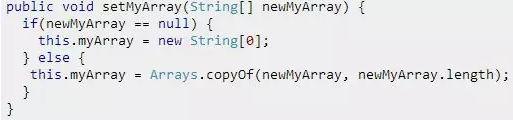
**1、当传递ArrayList到某个方法中，或者某个方法返回ArrayList，什么时候要考虑安全隐患？如何修复安全违规这个问题呢？**

当array被当做参数传递到某个方法中，如果array在没有被复制的情况下直接被分配给了成员变量，那么就可能发生这种情况，即当原始的数组被调用的方法改变的时候，传递到这个方法中的数组也会改变。下面的这段代码展示的就是安全违规以及如何修复这个问题。

ArrayList被直接赋给成员变量——安全隐患：



修复这个安全隐患：



**2、如何复制某个ArrayList到另一个ArrayList中去？写出你的代码？**

下面就是把某个ArrayList复制到另一个ArrayList中去的几种技术：

1：使用clone()方法，比如ArrayList newArray = oldArray.clone();

2：使用ArrayList构造方法，比如：ArrayList myObject = new ArrayList(myTempObject);

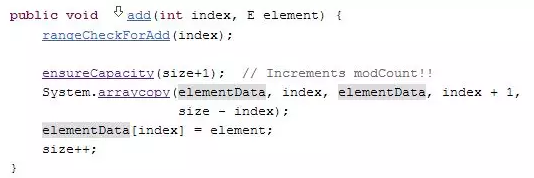
3：使用Collection的copy方法。

注意1和2是浅拷贝(shallow copy)。

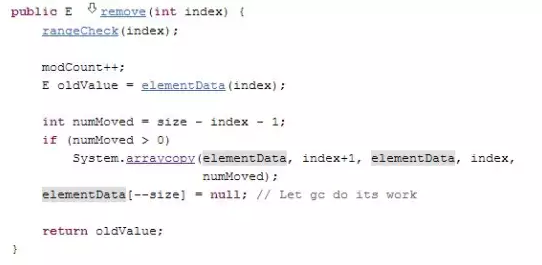
**3、在索引中ArrayList的增加或者删除某个对象的运行过程？效率很低吗？解释为什么？**

在ArrayList中增加或者是删除元素，要调用System.arraycopy这种效率很低的操作，如果遇到了需要频繁插入或者是删除的时候，可以选择其他的Java集合，比如LinkedList。看一下下面的代码：

在ArrayList的某个索引i处添加元素：



删除ArrayList的某个索引i处的元素：



### 1.2 LinkedList

LinkedList是基于链表(链表原先是C/C++的概念，是一种线性的存储结构，意思是将要存储的数据存在一个存储单元里面，这个存储单元里面除了存放有待存储的数据以外，还存储有其下一个存储单元的地址，每次查找数据的时候，通过某个存储单元中的下一个存储单元的地址寻找其后面的那个存储单元)实现的；允许为空(add null size也加1)、允许重复数据、有序、非线程安全；

LinkedList是一种双向链表，1、链表中任意一个存储单元都可以通过向前或者向后寻址的方式获取到其前一个存储单元和其后一个存储单元；2、链表的尾节点的后一个节点是链表的头结点，链表的头结点的前一个节点是链表的尾节点；

LinkedList既然是一种双向链表，必然有一个存储单元，以下是LinkedList的基本存储单元(是LinkedList中的一个内部类)：

private static class Node<E> {

    E item;

    Node <E> next;

    Node <E> prev;

    ...

}

LinkedList中的”E item”，就是它真正存储的数据。”Node<E> next”和” Node <E> prev”表示的就是这个存储单元的前一个存储单元的引用地址和后一个存储单元的引用地址。

#### 1.2.1 添加元素

LinkedList添加一个元素：

void linkLast(E e) {

final Node<E> l = last;

final Node<E> newNode = new Node<>(l, e, null);

last = newNode;

if(l == null)

first = newNode;

else

l.next = newNode;

size++;

modCount++;

}

类属性有一个Node first和Node last，add方法执行时把last赋给本地变量l，然后new一个Node(l,e,null)，其中prev为l的引用地址，e为要新增的元素，next为空；然后把newNode赋给last；判断l是否为空，为空则newNode赋给first，否则更新l的next属性为newNode；

#### 1.2.2 查找元素

LinkedList底层维护的是一个双向链表，所以根据索引查找元素时可以判断索引值与size的关系，index如果小于二分之一size则从头开始查找，否则从最后倒叙查找；

#### **1.2.4 LinkedList和ArrayList的对比**

1、顺序插入速度ArrayList会比较快，因为ArrayList是基于数组实现的，数组是事先new好的，只要往指定位置塞一个数据就好了；LinkedList则不同，每次顺序插入的时候LinkedList将new一个对象出来，如果对象比较大，那么new的时间势必会长一点，再加上一些引用赋值的操作，所以顺序插入LinkedList必然慢于ArrayList；

2、基于上一点，因为LinkedList里面不仅维护了待插入的元素，还维护了Entry的前置Entry和后继Entry，如果一个LinkedList中的Entry非常多，那么LinkedList将比ArrayList更耗费一些内存；

3、数据遍历的速度，使用各自遍历效率最高的方式，ArrayList的遍历效率会比LinkedList的遍历效率高一些；

4、有些说法认为LinkedList做插入和删除更快，这种说法其实是不准确的：

（1）LinkedList做插入、删除的时候，慢在寻址，快在只需要改变前后Entry的引用地址

（2）ArrayList做插入、删除的时候，慢在数组元素的批量copy，快在寻址

所以，如果待插入、删除的元素是在数据结构的前半段尤其是非常靠前的位置的时候，LinkedList的效率将大大快过ArrayList，因为ArrayList将批量copy大量的元素；越往后，对于LinkedList来说，因为它是双向链表，所以在第2个元素后面插入一个数据和在倒数第2个元素后面插入一个元素在效率上基本没有差别，但是ArrayList由于要批量copy的元素越来越少，操作速度必然追上乃至超过LinkedList。

从这个分析看出，如果你十分确定你插入、删除的元素是在前半段，那么就使用LinkedList；如果你十分确定你删除、删除的元素在比较靠后的位置，那么可以考虑使用ArrayList。如果你不能确定你要做的插入、删除是在哪儿呢？那还是建议你使用LinkedList吧，因为一来LinkedList整体插入、删除的执行效率比较稳定，没有ArrayList这种越往后越快的情况；二来插入元素的时候，弄得不好ArrayList就要进行一次扩容，记住，ArrayList底层数组扩容是一个既消耗时间又消耗空间的操作；

最后一点，一切都是纸上谈兵，在选择了List后，有条件的最好可以做一些性能测试，比如在你的代码上下文记录List操作的时间消耗。

#### **1.2.5 对LinkedList以及ArrayList的迭代**

ArrayList使用最普通的for循环遍历，LinkedList使用foreach循环比较快：

public class ArrayList<E> extends AbstractList<E>

        implements List<E>, RandomAccess, Cloneable, java.io.Serializable

public class LinkedList<E>

    extends AbstractSequentialList<E>

    implements List<E>, Deque<E>, Cloneable, java.io.Serializable

注意到ArrayList是实现了RandomAccess接口而LinkedList则没有实现这个接口，关于RandomAccess这个接口的作用，

虽然上面的例子用的Iterator，但是做foreach循环的时候，编译器默认会使用这个集合的Iterator，具体可参见foreach循环原理。测试代码如下：

public class TestMain

{

    private static int SIZE = 111111;

    private static void loopList(List<Integer> list)

    {

        long startTime = System.currentTimeMillis();

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

        {

            list.get(i);

        }

        System.out.println(list.getClass().getSimpleName() + "使用普通for循环遍历时间为" +

                (System.currentTimeMillis() - startTime) + "ms");

        startTime = System.currentTimeMillis();

        for (Integer i : list)

        {

        }

        System.out.println(list.getClass().getSimpleName() + "使用foreach循环遍历时间为" +

                (System.currentTimeMillis() - startTime) + "ms");

    }

    public static void main(String[] args)

    {

        List<Integer> arrayList = new ArrayList<Integer>(SIZE);

        List<Integer> linkedList = new LinkedList<Integer>();

        for (int i = 0; i < SIZE; i++)

        {

            arrayList.add(i);

            linkedList.add(i);

        }

        loopList(arrayList);

        loopList(linkedList);

        System.out.println();

    }

}

我截取三次运行结果：

ArrayList使用普通for循环遍历时间为6ms

ArrayList使用foreach循环遍历时间为12ms

LinkedList使用普通for循环遍历时间为38482ms

LinkedList使用foreach循环遍历时间为11ms

ArrayList使用普通for循环遍历时间为5ms

ArrayList使用foreach循环遍历时间为12ms

LinkedList使用普通for循环遍历时间为43287ms

LinkedList使用foreach循环遍历时间为9ms

ArrayList使用普通for循环遍历时间为4ms

ArrayList使用foreach循环遍历时间为12ms

LinkedList使用普通for循环遍历时间为22370ms

LinkedList使用foreach循环遍历时间为5ms

ArrayList用for循环遍历要快，linkedList用for each循环比较快；

如果使用普通for循环遍历LinkedList，其遍历速度将慢得令人发指。

### 1.3 CopyOnWriteArrayList

CopyOnWriteArrayList位于java.util.concurrent包下，这个类是为并发而设计的；CopyOnWriteArrayList，即Write时总是要Copy，也就是说对于CopyOnWriteArrayList，任何可变的操作（add、set、remove等等）都是伴随复制这个动作的，后面会解读CopyOnWriteArrayList的底层实现机制。

CopyOnWriteArrayList允许为空，允许重复数据，有序，线程安全；

CopyOnWriteArrayList适用于读操作远多于修改操作的并发场景中。

### 1.4 HashMap

HashMap是一种用哈希值来存储和查找键值对（key-value pair，也称作entry）的一种数据结构。每一个键值对也叫做Entry，这些个键值对（Entry）分散存储在一个数组当中，这个数组就是HashMap的主干。

HashMap的key和value都可以为空；key重复会覆盖，value可以重复；遍历HashMap时是无序的；非线程安全；

#### 1.4.1 底层原理

HashMap的一个存储单元Entry：

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

    final K key;

    V value;

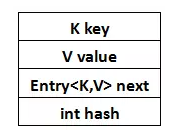
    Entry<K,V> next;

    int hash;

    ...

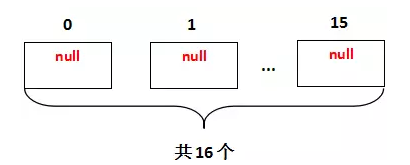
}

LinkedList是一个双向链表，HashMap的Entry组成的是一个单向链表，因为里面只有Entry的后继Entry，而没有Entry的前驱Entry。用图表示应该是以下一个数据结构：



#### 1.4.2 添加元素

DEFAULT\_INITIAL\_CAPACITY为16(最大容量2的30次方，即1073741824)，也就是说，HashMap在new的时候构造出了一个大小为16的Entry数组，Entry内所有数据都取默认值，如下图示：



看到new出了一个大小为16的Entry数组来，以下代码是put方法源码：

public V put(K key, V value) {

    if (key == null)

        return putForNullKey(value);

    int hash = hash(key.hashCode());

    int i = indexFor(hash, table.length);

    for (Entry<K,V> e = table[i]; e != null; e = e.next) {

       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++;

    addEntry(hash, key, value, i);

    return null;

}

static int hash(int h) {

    // This function ensures that hashCodes that differ only by

    // constant multiples at each bit position have a bounded

    // number of collisions (approximately 8 at default load factor).

    h ^= (h >>> 20) ^ (h >>> 12);

    return h ^ (h >>> 7) ^ (h >>> 4);

}

static int indexFor(int h, int length) {

     return h & (length-1);

 }

看一下put方法的几个步骤：

1、第2，3行是HashMap允许Key值为空的原因，空的Key会默认放在数组的第0位；

2、第4行拿到Key值的HashCode，由于HashCode是Object的方法，因此每个对象都

有一个HashCode，对这个HashCode做一次hash计算。按JDK源码注释的说法，

这次hash的作用是根据给定的HashCode对它做一次打乱的操作，防止一些糟糕的

Hash算法产生的糟糕的Hash值；

3、第5行根据重新计算的HashCode，对Entry数组的大小取模得到一个Entry数组的位

置。看到这里使用了&，移位加快一点代码运行效率。另外，这个取模操作的正确性依

赖于length必须是2的N次幂，这个熟悉二进制的朋友一定理解，因此注意HashMap

构造函数中，如果指定HashMap初始数组的大小initialCapacity，如果initialCapacity

不是2的N次幂，HashMap会算出大于initialCapacity的最小2的N次幂的值，作为

Entry数组的初始化大小。这里为讲解方便，我们假定字符串111和字符串222算出来

的i都是1

4、第6行~第14行会先判断一下原数据结构中是否存在相同的Key值，存在则覆盖并返

回，不执行后面的代码。注意一下recordAccess这个方法，它也是HashMap的子类

比如LinkedHashMap用的，HashMap中这个方法为空。另外，注意一点，对比Key

是否相同，是先比HashCode是否相同，HashCode相同再判断equals是否为true，

这样大大增加了HashMap的效率；

5、第16行的modeCount++是用于fail-fast机制的，每次修改HashMap数据结构的时候

都会自增一次这个值

然后就到了关键的addEntry方法了：

void addEntry(int hash, K key, V value, int bucketIndex) {

    Entry<K,V> e = table[bucketIndex];

    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

    if (size++ >= threshold)

        resize(2 \* table.length);

}

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

    value = v;

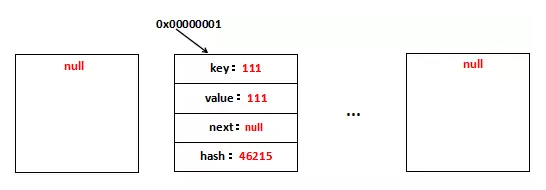
    next = n;

    key = k;

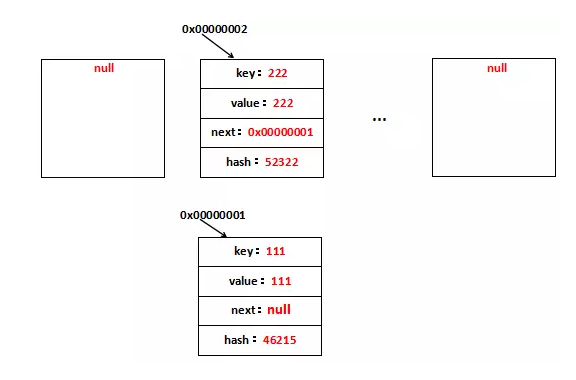
    hash = h;

}

假设new出来的Entry地址为0×00000001，那么，put(“111″, “111″)用图表示应该是这样的：



每一个新增的Entry都位于table[1]上，另外，里面的hash是rehash之后的hash而不是Key最原始的hash。看到table[1]上存放了111—->111这个键值对，它持有原table[1]的引用地址，因此可以寻址到原table[1]，这就是单向链表。 再看一下put(“222″, “222″)做了什么，一张图就可以理解了：



新的Entry再次占据table[1]的位置，并且持有原table[1]，也就是111—->111这个键值对。至此，HashMap进行put数据的过程就呈现清楚了。不过还有一个问题，就是HashMap如何进行扩容，再看一下addEntry方法：

void addEntry(int hash, K key, V value, int bucketIndex) {

      Entry<K,V> e = table[bucketIndex];

      table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

      if (size++ >= threshold)

          resize(2 \* table.length);

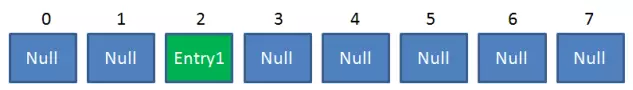
 }

看到第4行~第5行，也就是说在每次放置完Entry之后都会判断是否需要扩容。这里不讲扩容是因为HashMap扩容在不正确的使用场景下将会导致死循环，这是一个值得探讨的话题，也是我工作中实际遇到过的一个问题，因此下一篇文章将会详细说明为什么不正确地使用HashMap会导致死循环。

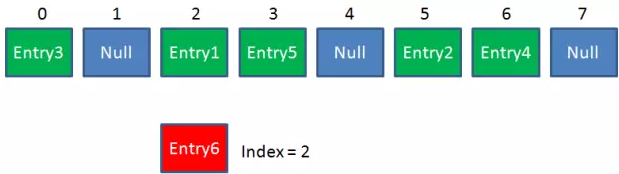
**Put方法的原理：**

HashMap数组每一个元素的初始值都是Null。调用 hashMap.put("apple", 0) ，插入一个Key为“apple"的元素。这时需要利用一个哈希函数来确定Entry的插入位置index：

index =  Hash（“apple”）；假定最后计算出的index是2，那么结果如下：



但是，因为HashMap的长度是有限的，当插入的Entry越来越多时，再完美的Hash函数也难免会出现index冲突的情况。比如下面这样：



HashMap数组的每一个元素不止是一个Entry对象，也是一个链表的头节点。每一个Entry对象通过Next指针指向它的下一个Entry节点。当新来的Entry映射到冲突的数组位置时，只需要插入（“头插法”）到对应的链表即可：



HashMap的默认初始长度是16(JDK1.8 1<<4)，每次自动扩展或手动初始化时，长度必须是2的N次幂；其中16是为了服务于从key映射到index的Hash算法；

index =  HashCode(Key) & (Length – 1)

其中2的N次幂减1得到的二进制数每个位上都是1，可以说，Hash算法最终得到的index结果，完全取决于Key的Hashcode值的最后几位，只要输入的HashCode本身分布均匀，Hash算法的结果就是均匀的；效果等同于取模但是效率更高；如果初始长度不是2的N次幂，HashMap会算出大于initialCapacity的最小2的N次幂的值，作为Entry数组的初始化大小。

#### 1.4.3 删除数据

HashMap删除指定键值对的源代码是：

public V remove(Object key) {

      Entry<K,V> e = removeEntryForKey(key);

      return (e == null ? null : e.value);

 }

final Entry<K,V> removeEntryForKey(Object key) {

    int hash = (key == null) ? 0 : hash(key.hashCode());

    int i = indexFor(hash, table.length);

    Entry<K,V> prev = table[i];

    Entry<K,V> e = prev;

    while (e != null) {

        Entry<K,V> next = e.next;

        Object k;

        if (e.hash == hash &&

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

            modCount++;

            size--;

            if (prev == e)

                table[i] = next;

            else

                prev.next = next;

            e.recordRemoval(this);

            return e;

        }

        prev = e;

        e = next;

    }

    return e;

}

分析一下remove元素的时候做了几步：

1、根据key的hash找到待删除的键值对位于table的哪个位置上

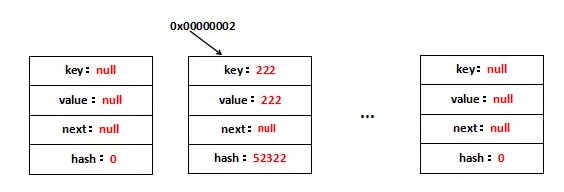
2、记录一个prev表示待删除的Entry的前一个位置Entry，e可以认为是当前位置

3、从table[i]开始遍历链表，假如找到了匹配的Entry，要判断这个Entry是不是table[i]：

（1）是的话，也就是第14行~第15行，table[i]就直接是table[i]的下一个节点，后面的都不需要动

（2）不是的话，也就是第16行~第17行，e的前一个Entry也就是prev，prev的next指向e的后一个节点，也就是next，这样，e所代表的Entry就被踢出了，e的前后Entry就连起来了

remove(“111″)用图表示就是：



整个过程只需要修改一个节点的next的值即可，非常方便。

#### 1.4.4 修改数据

修改元素也是put，看一下源代码：

public V put(K key, V value) {

    if (key == null)

        return putForNullKey(value);

    int hash = hash(key.hashCode());

    int i = indexFor(hash, table.length);

    for (Entry<K,V> e = table[i]; e != null; e = e.next) {

        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++;

    addEntry(hash, key, value, i);

    return null;

}

这个其实前面已经提到过了，第6行~第14行就是修改元素的逻辑，如果某个Key已经在数据结构中存在的话，那么就会覆盖原value，也就是第10行的代码。

**插入数据**

所谓”插入元素”，一定是基于数据结构是有序的前提下的。像ArrayList、LinkedList，再远点说就是数据库，一条一条都是有序的。而HashMap，它的顺序是基于HashCode，HashCode是一个随机性很强的数字，所以HashMap中的Entry完全是随机存放的。HashMap又不像LinkedHashMap这样维护了插入元素的顺序，所以对HashMap这个数据结构谈插入元素是没有意义的。所以，HashMap并没有插入的概念。

#### 1.4.5 重要问题

##### ****1.4.5.1 再谈HashCode的重要性****

HashMap中对Key的HashCode要做一次rehash，防止一些糟糕的Hash算法生成的糟糕的HashCode，那么为什么要防止糟糕的HashCode？

糟糕的HashCode意味着的是Hash冲突，即多个不同的Key可能得到的是同一个HashCode，糟糕的Hash算法意味着的就是Hash冲突的概率增大，这意味着HashMap的性能将下降，表现在两方面：

1. 有10个Key，可能6个Key的HashCode都相同，另外四个Key所在的Entry均匀分布在table的位置上，而某一个位置上却连接了6个Entry。这就失去了HashMap的意义，HashMap这种数据结构性高性能的前提是，Entry均匀地分布在table位置上，但现在确是1 1 1 1 6的分布。所以，我们要求HashCode有很强的随机性，这样就尽可能地可以保证了Entry分布的随机性，提升了HashMap的效率。
2. HashMap在一个某个table位置上遍历链表的时候的代码：

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

由于采用了”&&”运算符，因此先比较HashCode，HashCode都不相同就直接pass了，不会再进行equals比较了。HashCode因为是int值，比较速度非常快，而equals方法往往会对比一系列的内容，速度会慢一些。Hash冲突的概率大，意味着equals比较的次数势必增多，必然降低了HashMap的效率了。

##### ****1.4.5.2 HashMap的table为什么是transient的****

transient Entry[] table;

table用了transient修饰，也就是说table里面的内容全都不会被序列化，因为HashMap是基于HashCode的，HashCode作为Object的方法，是native的：

public native int hashCode();

这意味着的是：HashCode和底层实现相关，不同的虚拟机可能有不同的HashCode算法。如果table不被transient修饰，在虚拟机A上可以用的程序到虚拟机B上可以用的程序就不能用了，失去了跨平台性，因为：

1、Key在虚拟机A上的HashCode=100，连在table[4]上

2、Key在虚拟机B上的HashCode=101，这样，就去table[5]上找Key，明显找不到

整个代码就出问题了。因此，为了避免这一点，Java采取了重写自己序列化table的方法，在writeObject选择将key和value追加到序列化的文件最后面：

private void writeObject(java.io.ObjectOutputStream s)

        throws IOException

{

Iterator<Map.Entry<K,V>> i =

    (size > 0) ? entrySet0().iterator() : null;

// Write out the threshold, loadfactor, and any hidden stuff

s.defaultWriteObject();

// Write out number of buckets

s.writeInt(table.length);

// Write out size (number of Mappings)

s.writeInt(size);

    // Write out keys and values (alternating)

if (i != null) {

 while (i.hasNext()) {

    Map.Entry<K,V> e = i.next();

    s.writeObject(e.getKey());

    s.writeObject(e.getValue());

    }

    }

}

而在readObject的时候重构HashMap数据结构：

private void readObject(java.io.ObjectInputStream s)

         throws IOException, ClassNotFoundException

{

// Read in the threshold, loadfactor, and any hidden stuff

s.defaultReadObject();

// Read in number of buckets and allocate the bucket array;

int numBuckets = s.readInt();

table = new Entry[numBuckets];

    init();  // Give subclass a chance to do its thing.

// Read in size (number of Mappings)

int size = s.readInt();

// Read the keys and values, and put the mappings in the HashMap

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

    K key = (K) s.readObject();

    V value = (V) s.readObject();

    putForCreate(key, value);

}

}

一种麻烦的方式，但却保证了跨平台性。

这个例子也告诉了我们：尽管使用的虚拟机大多数情况下都是HotSpot，但是也不能对其它虚拟机不管不顾，有跨平台的思想是一件好事。

##### **1.4.5.3 HashMap和Hashtable的区别**

HashMap和Hashtable是一组相似的键值对集合，它们的区别也是面试常被问的问题之一，这里简单总结一下HashMap和Hashtable的区别：

1、Hashtable是线程安全的，Hashtable所有对外提供的方法都使用了synchronized，也就是同步，而HashMap则是线程非安全的

2、Hashtable不允许空的value，空的value将导致空指针异常，而HashMap则无所谓，没有这方面的限制

3、两个的rehash算法不同，Hashtable的是：

private int hash(Object k) {

    // hashSeed will be zero if alternative hashing is disabled.

    return hashSeed ^ k.hashCode();

}

这个hashSeed是使用sun.misc.Hashing类的randomHashSeed方法产生的。HashMap的rehash算法是：

static int hash(int h) {

    // This function ensures that hashCodes that differ only by

    // constant multiples at each bit position have a bounded

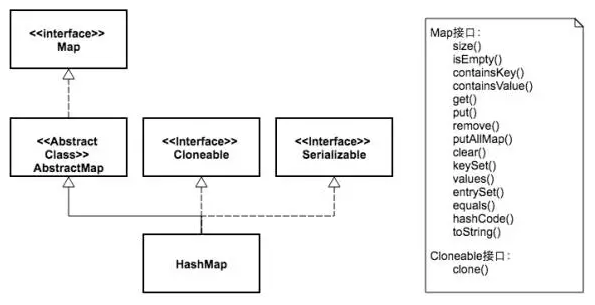
    // number of collisions (approximately 8 at default load factor).

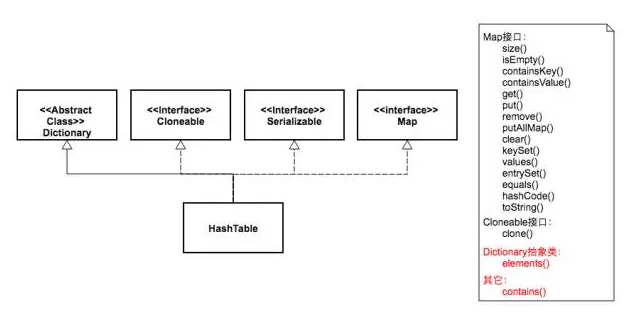
    h ^= (h >>> 20) ^ (h >>> 12);

    return h ^ (h >>> 7) ^ (h >>> 4);

}

1. HashTable产生于JDK 1.1，而HashMap产生于JDK 1.2。HashMap的作者多了大神Doug Lea；
2. HashMap和HashTable都是基于哈希表来实现键值映射的工具类。





HashMap继承自抽象类AbstractMap，而HashTable继承自抽象类Dictionary。其中Dictionary类是一个已经被废弃的类；从公开的方法上来看，这两个类提供的，是一样的功能。都提供键值映射的服务，可以增、删、查、改键值对，可以对建、值、键值对提供遍历视图。支持浅拷贝，支持序列化。

HashMap是支持null键和null值的，而HashTable在遇到null时，会抛出NullPointerException异常。这并不是因为HashTable有什么特殊的实现层面的原因导致不能支持null键和null值，这仅仅是因为HashMap在实现时对null做了特殊处理，将null的hashCode值定为了0，从而将其存放在哈希表的第0个bucket中。

##### **1.4.5.4 在HashMap中使用可变对象作为Key带来的问题**

如果HashMap Key的哈希值在存储键值对后发生改变，Map可能再也查找不到这个Entry了。如果Key对象是可变的，那么Key的哈希值就可能改变。在HashMap中可变对象作为Key会造成数据丢失。

在HashMap中使用不可变对象。在HashMap中，使用String、Integer等不可变类型用作Key是非常明智的。如果可变对象在HashMap中被用作键，那就要小心在改变对象状态的时候，不要改变它的哈希值了。

##### 1.4.5.5 HashMap是线程不安全

**resize死循环，**HashMap初始容量大小为16,一般来说，当有数据要插入时，都会检查容量有没有超过设定的thredhold，如果超过，需要增大Hash表的尺寸，但是这样一来，整个Hash表里的元素都需要被重算一遍。这叫rehash，这个成本相当的大。

void resize(int newCapacity) {

        Entry[] oldTable = table;

        int oldCapacity = oldTable.length;

        if (oldCapacity == MAXIMUM\_CAPACITY) {

            threshold = Integer.MAX\_VALUE;

            return;

        }

        Entry[] newTable = new Entry[newCapacity];

        transfer(newTable, initHashSeedAsNeeded(newCapacity));

        table = newTable;

        threshold = (int)Math.min(newCapacity \* loadFactor, MAXIMUM\_CAPACITY + 1);

}

void transfer(Entry[] newTable, boolean rehash) {

        int newCapacity = newTable.length;

        for (Entry<K,V> e : table) {

            while(null != e) {

                Entry<K,V> next = e.next;

                if (rehash) {

                    e.hash = null == e.key ? 0 : hash(e.key);

                }

                int i = indexFor(e.hash, newCapacity);

                e.next = newTable[i];

                newTable[i] = e;

                e = next;

            }

        }

}

transfer：

对索引数组中的元素遍历

1. 对链表上的每一个节点遍历：用 next 取得要转移那个元素的下一个，将 e 转移到新

Hash 表的头部，使用头插法插入节点。

2、循环2，直到链表节点全部转移

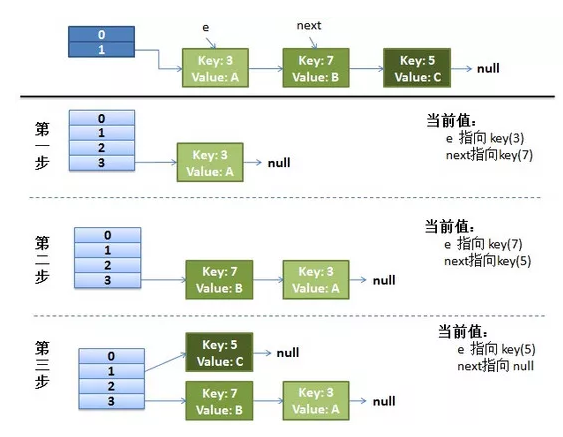
3、循环1，直到所有索引数组全部转移

其中转移的时候是逆序的。假如转移前链表顺序是1->2->3，那么转移后就会变成3->2->1。死锁问题不就是因为1->2的同时2->1造成的吗？所以，HashMap 的死锁问题就出在这个transfer()函数上。

单线程情况下，rehash 不会出现任何问题：

* 假设hash算法就是最简单的 key mod table.length（也就是数组的长度）。
* 最上面的是old hash 表，其中的Hash表的 size = 2, 所以 key = 3, 7, 5，在 mod 2以后碰撞发生在 table[1]
* 接下来的三个步骤是 Hash表 resize 到4，并将所有的 <key,value> 重新rehash到新 Hash 表的过程

如图所示：



**多线程 rehash 详细演示**

为了思路更清晰，我们只将关键代码展示出来

while(null != e) {

    Entry<K,V> next = e.next;

    e.next = newTable[i];

    newTable[i] = e;

    e = next;

}

1. Entry<K,V> next = e.next;——因为是单链表，如果要转移头指针，一定要保存下一个

结点，不然转移后链表就丢了；

2、e.next = newTable[i];——e 要插入到链表的头部，所以要先用 e.next 指向新的 Hash

表第一个元素（为什么不加到新链表最后？因为复杂度是 O（N））

3、newTable[i] = e;——现在新 Hash 表的头指针仍然指向 e 没转移前的第一个元素，

所以需要将新 Hash 表的头指针指向 e；

4、e = next——转移 e 的下一个结点

假设这里有两个线程同时执行了put()操作，并进入了transfer()环节

while(null != e) {

    Entry<K,V> next = e.next; //线程1执行到这里被调度挂起了

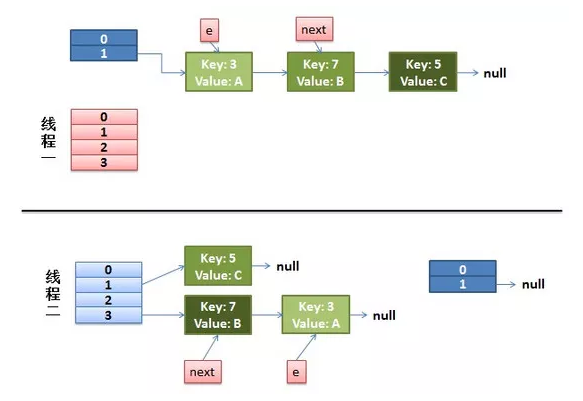
    e.next = newTable[i];

    newTable[i] = e;

    e = next;

}

那么现在的状态为：



从上面的图我们可以看到，因为线程1的 e 指向了 key(3)，而 next 指向了 key(7)，在线程2 rehash 后，就指向了线程2 rehash 后的链表。

然后线程1被唤醒了：

执行e.next = newTable[i]，于是 key(3)的 next 指向了线程1的新 Hash 表，因为新 Hash 表为空，所以e.next = null，

执行newTable[i] = e，所以线程1的新 Hash 表第一个元素指向了线程2新 Hash 表的 key(3)。好了，e 处理完毕。

执行e = next，将 e 指向 next，所以新的 e 是 key(7)

然后该执行 key(3)的 next 节点 key(7)了:

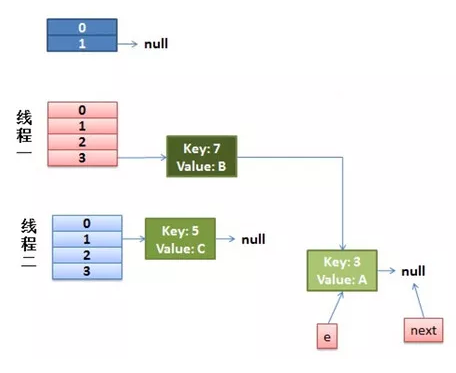
现在的 e 节点是 key(7)，首先执行Entry<K,V> next = e.next,那么 next 就是 key(3)了

执行e.next = newTable[i]，于是key(7) 的 next 就成了 key(3)

执行newTable[i] = e，那么线程1的新 Hash 表第一个元素变成了 key(7)

执行e = next，将 e 指向 next，所以新的 e 是 key(3)

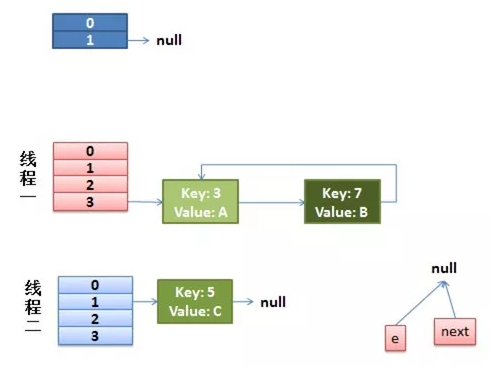
这时候的状态图为：



然后又该执行 key(7)的 next 节点 key(3)了：

1. 现在的 e 节点是 key(3)，首先执行Entry<K,V> next = e.next,那么 next 就是 null
2. 执行e.next = newTable[i]，于是key(3) 的 next 就成了 key(7)
3. 执行newTable[i] = e，那么线程1的新 Hash 表第一个元素变成了 key(3)
4. 执行e = next，将 e 指向 next，所以新的 e 是 key(7)

这时候的状态如图所示：



很明显，环形链表出现了！！当然，现在还没有事情，因为下一个节点是 null，所以transfer()就完成了，等put()的其余过程搞定后，HashMap 的底层实现就是线程1的新 Hash 表了。如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

##### 1.4.5.6 不正确地使用 HashMap 引发死循环及元素丢失

HashMap在多线程环境下会引发死循环的这个问题，下面先用一段代码简单模拟出HashMap的死循环：

public class HashMapThread extends Thread

{

    private static AtomicInteger ai = new AtomicInteger(0);

    private static Map<Integer, Integer> map = new HashMap<Integer, Integer>(1);

    public void run()

    {

        while (ai.get() < 100000)

        {

            map.put(ai.get(), ai.get());

            ai.incrementAndGet();

        }

    }

}

这个线程的作用很简单，给AtomicInteger不断自增并写入HashMap中，其中AtomicInteger和HashMap都是全局共享的，也就是说所有线程操作的都是同一个AtomicInteger和HashMap。开5个线程操作一下run方法中的代码：

public static void main(String[] args)

{

    HashMapThread hmt0 = new HashMapThread();

    HashMapThread hmt1 = new HashMapThread();

    HashMapThread hmt2 = new HashMapThread();

    HashMapThread hmt3 = new HashMapThread();

    HashMapThread hmt4 = new HashMapThread();

    hmt0.start();

    hmt1.start();

    hmt2.start();

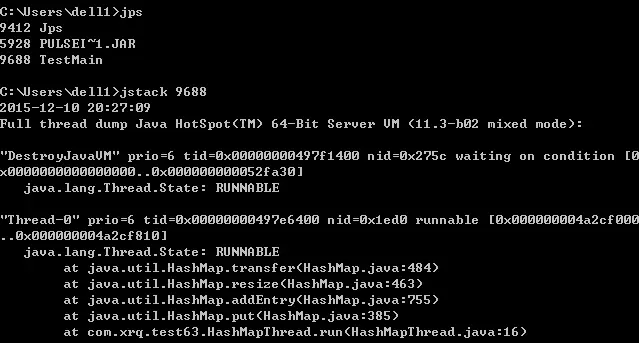
    hmt3.start();

    hmt4.start();

}

多运行几次之后死循环就出来了，我大概运行了7次、8次的样子，其中有几次是数组下标越界异常ArrayIndexOutOfBoundsException。这里面要提一点，多线程环境下代码会出现问题并不意味着多线程环境下一定会出现问题，但是只要出现了问题，或者是死锁、或者是死循环，那么你的项目除了重启就没有什么别的办法了，所以代码的线程安全性在开发、评审的时候必须要重点考虑到。

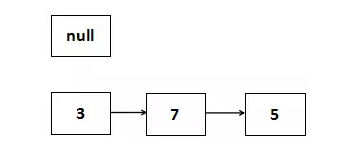
死循环问题的定位一般都是通过jps+jstack查看堆栈信息来定位的：



看到Thread-0处于RUNNABLE，而从堆栈信息上应该可以看出，这次的死循环是由于Thread-0对HashMap进行扩容而引起的。

**正常的扩容过程**

HashMap一次正常的扩容过程。简单一点看吧，假设有三个经过了最终rehash得到的数字，分别是5 7 3，HashMap的table也只有2，那么HashMap把这三个数字put进数据结构了之后应该是这么一个样子的：



然后看一下resize的代码：

void addEntry(int hash, K key, V value, int bucketIndex) {

Entry<K,V> e = table[bucketIndex];

    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

    if (size++ >= threshold)

        resize(2 \* table.length);

}

void resize(int newCapacity) {

    Entry[] oldTable = table;

    int oldCapacity = oldTable.length;

    if (oldCapacity == MAXIMUM\_CAPACITY) {

        threshold = Integer.MAX\_VALUE;

        return;

    }

    Entry[] newTable = new Entry[newCapacity];

    transfer(newTable);

    table = newTable;

    threshold = (int)(newCapacity \* loadFactor);

}

void transfer(Entry[] newTable) {

    Entry[] src = table;

    int newCapacity = newTable.length;

    for (int j = 0; j < src.length; j++) {

        Entry<K,V> e = src[j];

        if (e != null) {

            src[j] = null;

            do {

                Entry<K,V> next = e.next;

                int i = indexFor(e.hash, newCapacity);

                e.next = newTable[i];

                newTable[i] = e;

                e = next;

            } while (e != null);

        }

    }

}

总结一下这三段代码，HashMap一次扩容的过程应该是：

1、取当前table的2倍作为新table的大小；

2、根据算出的新table的大小new出一个新的Entry数组来，名为newTable；

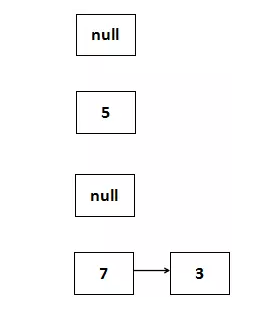
3、轮询原table的每一个位置，将每个位置上连接的Entry，算出在新table上的位置，

并以链表形式连接；

4、原table上的所有Entry全部轮询完毕之后，意味着原table上面的所有Entry已经移

到了新的table上，HashMap中的table指向newTable；

这样就完成了一次扩容，用图表示是这样的：



**扩容导致的死循环**

既然是扩容导致的死循环，那么继续看扩容的代码：

void transfer(Entry[] newTable) {

    Entry[] src = table;

    int newCapacity = newTable.length;

    for (int j = 0; j < src.length; j++) {

        Entry<K,V> e = src[j];

        if (e != null) {

            src[j] = null;

            do {

                Entry<K,V> next = e.next;

                int i = indexFor(e.hash, newCapacity);

                e.next = newTable[i];

                newTable[i] = e;

                e = next;

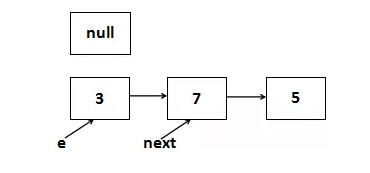
            } while (e != null);

        }

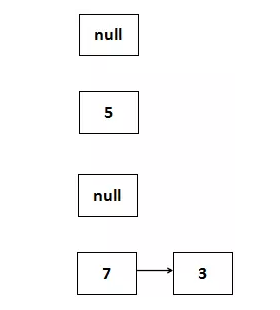
    }

}

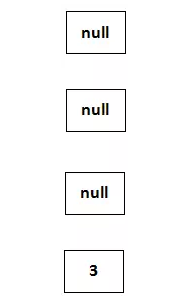
两个线程，线程A和线程B。假设第9行执行完毕，线程A切换，那么对于线程A而言，是这样的：



CPU切换到线程B运行，线程B将整个扩容过程全部执行完毕，于是就形成了：



此时CPU切换到线程A上，执行第8行~第14行的do…while…循环，首先放置3这个Entry：



我们必须要知道，由于线程B已经执行完毕，因此根据Java内存模型（JMM），现在table里面所有的Entry都是最新的，也就是7的next是3，3的next是null。3放置到table[3]的位置上了，下面的步骤是：

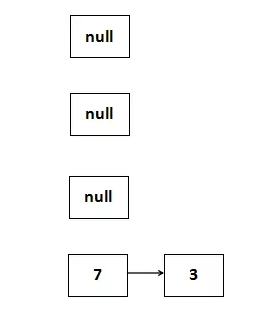
1、e=next，即e=7

2、判断e不等于null，循环继续

3、next=e.next，即next=7的next，也就是3

4、放置7这个Entry

所以，用图表示就是：



放置完7之后，继续运行代码：

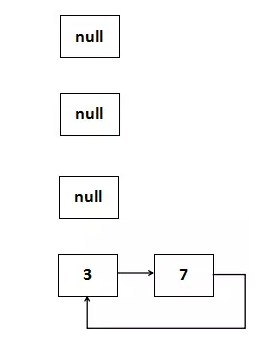
1、e=next，也就是说e=3

2、判断e不等于null，循环继续

3、next=e.next，即3的next，也就是null

4、放置3这个Entry

把3移到table[3]上去，死循环就出来了：



3移到table[3]上去了，3的next指向7，由于原先7的next指向3，这样就成了一个死循环。

此时执行13行的e=next，那么e=null，循环终止。尽管此次循环确实结束了，但是后面的操作，只要涉及轮询HashMap数据结构的，无论是迭代还是扩容，都将在table[3]这个链表处出现死循环。这也就是前面的死循环堆栈出现的原因，transfer的484行，因为这是一次扩容操作，需要遍历HashMap数据结构，transfer方法是扩容的最后一个方法。

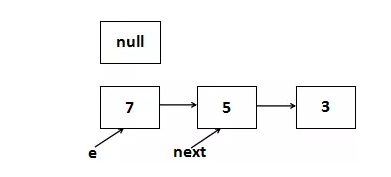
**3 5 7又会有怎样的结果**

可能有人觉得上面的数字5 7 3太巧了，像是专门为了产生HashMap的死循环而故意选择的数字。

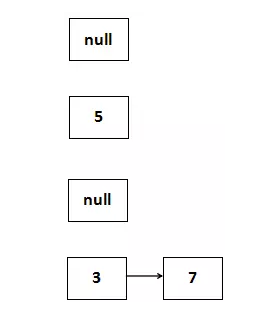
这个问题，我这么回答：我记得在《从Paxos到Zookeeper分布式一致性原理与实践》有一段话大概是这么描述的，有一个被反复实践得出的结论是，任何在多线程下可能发生的错误场景最终一定会发生。

5 7 3这个数字可不巧，扩容前相邻两个Entry被分配到扩容后同样的table位置是很正常的。关键的是，即使这种异常场景发生的可能性再低，只要发生一次，那么你的系统就部分甚至全部不可用了—-除了重启系统没有任何办法。所以，这种可能会发生的异常场景必须提前扼杀。

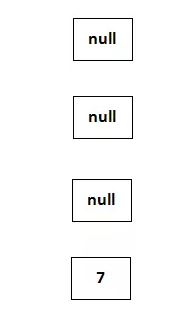
OK，不扯了，前面讲了5 7 3导致了死循环，现在看一下正常的顺序3 5 7，会发生什么问题。简单看一下，就不像上面讲得这么详细了：



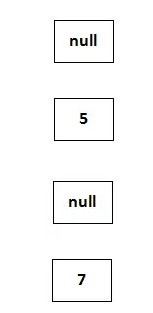
这是扩容前数据结构中的内容，扩容之后正常的应该是：



现在在多线程下遇到问题了，某个线程先放7：



再接着放5：



由于5的next此时为null，因此扩容操作结束，3 5 7造成的结果就是元素丢失。

**如何解决**

把一个线程非安全的集合作为全局共享的，本身就是一种错误的做法，并发下一定会产生错误。

所以，解决这个问题的办法很简单，有两种：

1、使用Collections.synchronizedMap(Mao<K,V> m)方法把HashMap变成一个线程安全的Map

2、使用Hashtable、ConcurrentHashMap这两个线程安全的Map

不过，既然选择了线程安全的办法，那么必然要在性能上付出一定的代价—-毕竟这个世界上没有十全十美的事情，既要运行效率高、又要线程安全。

##### 1.4.5.7 HashMap 遍历方式

JDK8之前，可以使用keySet或者entrySet来遍历HashMap，JDK8中引入了map.foreach来进行遍历。

原因：keySet其实是遍历了2次，一次是转为Iterator对象，另一次是从hashMap中取出key所对应的value。而entrySet只是遍历了一次就把key和value都放到了entry中，效率更高。如果是JDK8，使用Map.foreach方法*。*

**1. keySet和entrySet**

**1.1 基本用法**

keySet:

Map map=new HashMap();

Iterator it=map.keySet().iterator();

Object key;

Object value;

while(it.hasNext()){

key=it.next();

value=map.get(key);

System.out.println(key+":"+value);

}

entrySet:

Map map=new HashMap();

Iterator it=map.entrySet().iterator();

Object key;

Object value;

while(it.hasNext()){

Map.Entry entry = (Map.Entry)it.next();

key=entry.getKey();

value=entry.getValue();

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

}

源码上看：

keySet:

final class KeyIterator extends HashIterator

        implements Iterator<K> {

        public final K next() { return nextNode().key; }

    }

entrySet:

final class EntryIterator extends HashIterator

        implements Iterator<Map.Entry<K,V>> {

        public final Map.Entry<K,V> next() { return nextNode(); }

    }

其实这里已经很明显了，当要得到某个value时，keySet还需要从HashMap中get，entrySet相比keySet少了遍历table的过程，这也是两者性能上的主要差别。

**2. Map.foreach**

在JDK8以后，引入了Map.foreach。Map.foreach本质仍然是entrySet

default void forEach(BiConsumer<? super K, ? super V> action) {

        Objects.requireNonNull(action);

        for (Map.Entry<K, V> entry : entrySet()) {

            K k;

            V v;

            try {

                k = entry.getKey();

                v = entry.getValue();

            } catch(IllegalStateException ise) {

                // this usually means the entry is no longer in the map.

                throw new ConcurrentModificationException(ise);

            }

            action.accept(k, v);

        }

    }

配合lambda表达式一起使用，操作起来更加方便。

**2.1 使用Java8的foreach+lambda表达式遍历Map**

Map<String, Integer> items = new HashMap<>();

items.put("A", 10);

items.put("B", 20);

items.put("C", 30);

items.put("D", 40);

items.put("E", 50);

items.put("F", 60);

items.forEach((k,v)->System.out.println("Item : " + k + " Count : " + v));

items.forEach((k,v)->{

    System.out.println("Item : " + k + " Count : " + v);

    if("E".equals(k)){

        System.out.println("Hello E");

    }

});

### 1.5 LinkedHashMap

HashMap是一种非常常见、非常有用的集合，并且在多线程情况下使用不当会有线程安全问题。大多数情况下，只要不涉及线程安全问题，Map基本都可以使用HashMap，不过HashMap有一个问题，就是迭代HashMap的顺序并不是HashMap放置的顺序，也就是无序。LinkedHashMap虽然增加了时间和空间上的开销，但是通过维护一个运行于所有条目的双向链表，LinkedHashMap保证了元素迭代的顺序。

LinkedHashMap允许键值对为空(即Key和Value都允许为空)，允许重复数据(Key重复会覆盖、Value允许重复)，有序，非线程安全；

#### 1.5.1 基本数据结构

关于LinkedHashMap，需要注意两点：

1、LinkedHashMap可以认为是HashMap+LinkedList，即它既使用HashMap操作数据结构，又使用LinkedList维护插入元素的先后顺序；

2、LinkedHashMap的基本实现思想就是—-多态。可以说，理解多态，再去理解LinkedHashMap原理会事半功倍；反之也是，对于LinkedHashMap原理的学习，也可以促进和加深对于多态的理解。

为什么可以这么说，首先看一下，LinkedHashMap的定义：

public class LinkedHashMap<K,V>

    extends HashMap<K,V>

    implements Map<K,V>

{

    ...

}

看到，LinkedHashMap是HashMap的子类，自然LinkedHashMap也就继承了HashMap中所有非private的方法。再看一下LinkedHashMap中本身的方法：

看到LinkedHashMap中并没有什么操作数据结构的方法，也就是说LinkedHashMap操作数据结构（比如put一个数据），和HashMap操作数据的方法完全一样，无非就是细节上有一些的不同罢了。

LinkedHashMap和HashMap的区别在于它们的基本数据结构上，看一下LinkedHashMap的基本数据结构，也就是Entry：

private static class Entry<K,V> extends HashMap.Entry<K,V> {

    // These fields comprise the doubly linked list used for iteration.

    Entry<K,V> before, after;

Entry(int hash, K key, V value, HashMap.Entry<K,V> next) {

        super(hash, key, value, next);

    }

    ...

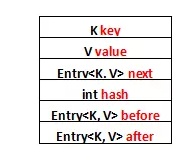
}

列一下Entry里面有的一些属性吧：

* K key
* V value
* Entry<K, V> next
* int hash
* Entry<K, V> before
* Entry<K, V> after

其中前面四个，也就是红色部分是从HashMap.Entry中继承过来的；后面两个，也就是蓝色部分是LinkedHashMap独有的。不要搞错了next和before、After，next是用于维护HashMap指定table位置上连接的Entry的顺序的，before、After是用于维护Entry插入的先后顺序的。

还是用图表示一下，列一下属性而已：



**初始化LinkedHashMap**

假如有这么一段代码：

public static void main(String[] args)

{

    LinkedHashMap<String, String> linkedHashMap =

            new LinkedHashMap<String, String>();

    linkedHashMap.put("111", "111");

    linkedHashMap.put("222", "222");

}

首先是第3行~第4行，new一个LinkedHashMap出来，看一下做了什么：

public LinkedHashMap() {

      super();

      accessOrder = false;

 }

public HashMap() {

     this.loadFactor = DEFAULT\_LOAD\_FACTOR;

     threshold = (int)(DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);

     table = new Entry[DEFAULT\_INITIAL\_CAPACITY];

     init();

}

void init() {

      header = new Entry<K,V>(-1, null, null, null);

      header.before = header.after = header;

 }

/\*\*

 \* The head of the doubly linked list.

 \*/

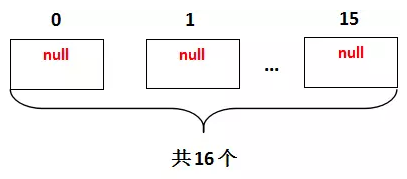
private transient Entry<K,V> header;

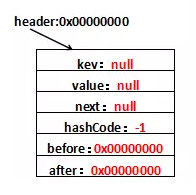
这里出现了第一个多态：init()方法。尽管init()方法定义在HashMap中，但是由于：

1、LinkedHashMap重写了init方法

2、实例化出来的是LinkedHashMap

因此实际调用的init方法是LinkedHashMap重写的init方法。假设header的地址是0×00000000，那么初始化完毕，实际上是这样的：





#### 1.5.2 添加元素

继续看LinkedHashMap添加元素，也就是put(“111″,”111″)做了什么，首先当然是调用HashMap的put方法：

public V put(K key, V value) {

    if (key == null)

        return putForNullKey(value);

    int hash = hash(key.hashCode());

    int i = indexFor(hash, table.length);

    for (Entry<K,V> e = table[i]; e != null; e = e.next) {

        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++;

    addEntry(hash, key, value, i);

    return null;

}

第17行又是一个多态，因为LinkedHashMap重写了addEntry方法，因此addEntry调用的是LinkedHashMap重写了的方法：

void addEntry(int hash, K key, V value, int bucketIndex) {

    createEntry(hash, key, value, bucketIndex);

    // Remove eldest entry if instructed, else grow capacity if appropriate

    Entry<K,V> eldest = header.after;

    if (removeEldestEntry(eldest)) {

        removeEntryForKey(eldest.key);

    } else {

        if (size >= threshold)

            resize(2 \* table.length);

    }

}

因为LinkedHashMap由于其本身维护了插入的先后顺序，因此LinkedHashMap可以用来做缓存，第5行~第7行是用来支持FIFO算法的，这里暂时不用去关心它。看一下createEntry方法：

void createEntry(int hash, K key, V value, int bucketIndex) {

    HashMap.Entry<K,V> old = table[bucketIndex];

    Entry<K,V> e = new Entry<K,V>(hash, key, value, old);

    table[bucketIndex] = e;

    e.addBefore(header);

    size++;

}

private void addBefore(Entry<K,V> existingEntry) {

    after  = existingEntry;

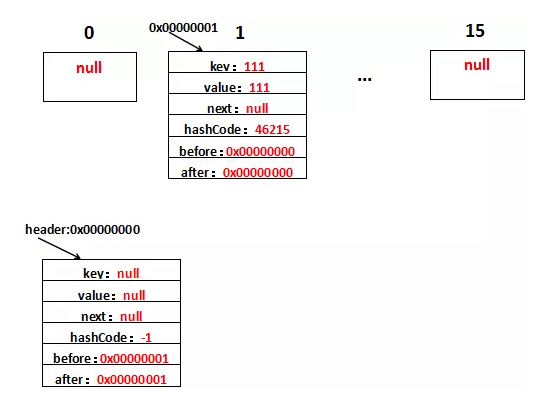
    before = existingEntry.before;

    before.after = this;

    after.before = this;

}

第2行~第4行的代码和HashMap没有什么不同，新添加的元素放在table[i]上，差别在于LinkedHashMap还做了addBefore操作，这四行代码的意思就是让新的Entry和原链表生成一个双向链表。假设字符串111放在位置table[1]上，生成的Entry地址为0×00000001，那么用图表示是这样的：



如果熟悉LinkedList的源码应该不难理解，还是解释一下，注意下existingEntry表示的是header：

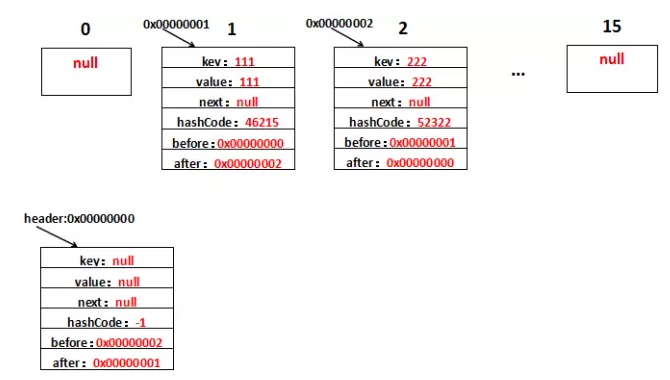
1、after=existingEntry，即新增的Entry的after=header地址，即after=0×00000000

2、before=existingEntry.before，即新增的Entry的before是header的before的地址，header的before此时是0×00000000，因此新增的Entry的before=0×00000000

3、before.after=this，新增的Entry的before此时为0×00000000即header，header的after=this，即header的after=0×00000001

4、after.before=this，新增的Entry的after此时为0×00000000即header，header的before=this，即header的before=0×00000001

这样，header与新增的Entry的一个双向链表就形成了。再看，新增了字符串222之后是什么样的，假设新增的Entry的地址为0×00000002，生成到table[2]上，用图表示是这样的：



就不细解释了，只要before、after清除地知道代表的是哪个Entry的就不会有什么问题。

总得来看，再说明一遍，LinkedHashMap的实现就是HashMap+LinkedList的实现方式，以HashMap维护数据结构，以LinkList的方式维护数据插入顺序。

**利用LinkedHashMap实现LRU算法缓存**

前面讲了LinkedHashMap添加元素，删除、修改元素就不说了，比较简单，和HashMap+LinkedList的删除、修改元素大同小异，下面讲一个新的内容。

LinkedHashMap可以用来作缓存，比方说LRUCache，看一下这个类的代码，很简单，就十几行而已：

public class LRUCache extends LinkedHashMap

{

    public LRUCache(int maxSize)

    {

        super(maxSize, 0.75F, true);

        maxElements = maxSize;

    }

    protected boolean removeEldestEntry(java.util.Map.Entry eldest)

    {

        return size() > maxElements;

    }

    private static final long serialVersionUID = 1L;

    protected int maxElements;

}

顾名思义，LRUCache就是基于LRU算法的Cache（缓存），这个类继承自LinkedHashMap，而类中看到没有什么特别的方法，这说明LRUCache实现缓存LRU功能都是源自LinkedHashMap的。LinkedHashMap可以实现LRU算法的缓存基于两点：

1、LinkedList首先它是一个Map，Map是基于K-V的，和缓存一致

2、LinkedList提供了一个boolean值可以让用户指定是否实现LRU

那么，首先我们了解一下什么是LRU：LRU即Least Recently Used，最近最少使用，也就是说，当缓存满了，会优先淘汰那些最近最不常访问的数据。比方说数据a，1天前访问了；数据b，2天前访问了，缓存满了，优先会淘汰数据b。

我们看一下LinkedList带boolean型参数的构造方法：

public LinkedHashMap(int initialCapacity,

         float loadFactor,

                     boolean accessOrder) {

    super(initialCapacity, loadFactor);

    this.accessOrder = accessOrder;

}

就是这个accessOrder，它表示：

（1）false，所有的Entry按照插入的顺序排列

（2）true，所有的Entry按照访问的顺序排列

第二点的意思就是，如果有1 2 3这3个Entry，那么访问了1，就把1移到尾部去，即2 3 1。每次访问都把访问的那个数据移到双向队列的尾部去，那么每次要淘汰数据的时候，双向队列最头的那个数据不就是最不常访问的那个数据了吗？换句话说，双向链表最头的那个数据就是要淘汰的数据。

“访问”，这个词有两层意思：

1、根据Key拿到Value，也就是get方法

2、修改Key对应的Value，也就是put方法

首先看一下get方法，它在LinkedHashMap中被重写：

public V get(Object key) {

    Entry<K,V> e = (Entry<K,V>)getEntry(key);

    if (e == null)

        return null;

    e.recordAccess(this);

    return e.value;

}

然后是put方法，沿用父类HashMap的：

public V put(K key, V value) {

    if (key == null)

        return putForNullKey(value);

    int hash = hash(key.hashCode());

    int i = indexFor(hash, table.length);

    for (Entry<K,V> e = table[i]; e != null; e = e.next) {

        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++;

    addEntry(hash, key, value, i);

    return null;

}

修改数据也就是第6行~第14行的代码。看到两端代码都有一个共同点：都调用了recordAccess方法，且这个方法是Entry中的方法，也就是说每次的recordAccess操作的都是某一个固定的Entry。

recordAccess，顾名思义，记录访问，也就是说你这次访问了双向链表，我就把你记录下来，怎么记录？把你访问的Entry移到尾部去。这个方法在HashMap中是一个空方法，就是用来给子类记录访问用的，看一下LinkedHashMap中的实现：

void recordAccess(HashMap<K,V> m) {

    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;

    if (lm.accessOrder) {

        lm.modCount++;

        remove();

        addBefore(lm.header);

    }

}

private void remove() {

    before.after = after;

    after.before = before;

}

private void addBefore(Entry<K,V> existingEntry) {

    after  = existingEntry;

    before = existingEntry.before;

    before.after = this;

    after.before = this;

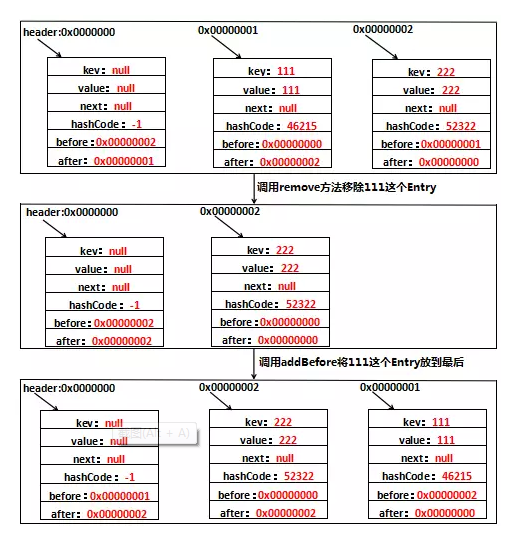
}

看到每次recordAccess的时候做了两件事情：

1、把待移动的Entry的前后Entry相连

2、把待移动的Entry移动到尾部

当然，这一切都是基于accessOrder=true的情况下。最后用一张图表示一下整个recordAccess的过程吧：



**代码演示LinkedHashMap按照访问顺序排序的效果**

最后代码演示一下LinkedList按照访问顺序排序的效果，验证一下上一部分LinkedHashMap的LRU功能：

public static void main(String[] args)

{

    LinkedHashMap<String, String> linkedHashMap =

            new LinkedHashMap<String, String>(16, 0.75f, true);

    linkedHashMap.put("111", "111");

    linkedHashMap.put("222", "222");

    linkedHashMap.put("333", "333");

    linkedHashMap.put("444", "444");

    loopLinkedHashMap(linkedHashMap);

    linkedHashMap.get("111");

    loopLinkedHashMap(linkedHashMap);

    linkedHashMap.put("222", "2222");

    loopLinkedHashMap(linkedHashMap);

}

public static void loopLinkedHashMap(LinkedHashMap<String, String> linkedHashMap)

{

    Set<Map.Entry<String, String>> set = inkedHashMap.entrySet();

    Iterator<Map.Entry<String, String>> iterator = set.iterator();

    while (iterator.hasNext())

    {

        System.out.print(iterator.next() + "\t");

    }

    System.out.println();

}

注意这里的构造方法要用三个参数那个且最后的要传入true，这样才表示按照访问顺序排序。看一下代码运行结果：

111=111    222=222    333=333    444=444

222=222    333=333    444=444    111=111

333=333    444=444    111=111    222=2222

代码运行结果证明了两点：

1、LinkedList是有序的；

1. 每次访问一个元素（get或put），被访问的元素都被提到最后面去了。

### 1.6 红黑树(TreeMap)

红黑树（Red Black Tree） 是一种自平衡二叉查找树，是在[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA)科学中用到的一种[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84)，典型的用途是实现[关联数组](https://baike.baidu.com/item/%E5%85%B3%E8%81%94%E6%95%B0%E7%BB%84)。它是在1972年由Rudolf Bayer发明的，当时被称为平衡二叉B树（symmetric binary B-trees）。后来，在1978年被 Leo J. Guibas 和 Robert Sedgewick 修改为如今的“红黑树”。红黑树和AVL树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。它虽然是复杂的，但它的最坏情况运行时间也是非常良好的，并且在实践中是高效的： 它可以在O(log n)时间内做查找，插入和删除，这里的n 是树中元素的数目。

HashMap与LinkedHashMap，它们保证了以O(1)的时间复杂度进行增、删、改、查，从存储角度考虑，这两种数据结构是非常优秀的。另外，LinkedHashMap还额外地保证了Map的遍历顺序可以与put顺序一致，解决了HashMap本身无序的问题。尽管如此，HashMap与LinkedHashMap还是有自己的局限性----它们不具备统计性能，或者说它们的统计性能时间复杂度并不是很好才更准确，所有的统计必须遍历所有Entry，因此时间复杂度为O(N)。HashMap和LinkedHashMap在统计上做得比较差，此时可以使用TreeMap。TreeMap的Key按照自然顺序进行排序或根据创建映射时提供的Comparator接口进行排序。TreeMap为增、删、改、查这些操作提供了log(N)的时间开销，从存储角度而言，这比HashMap与LinkedHashMap的O(1)时间复杂度要差些；但是在统计性能上，TreeMap同样可以保证log(N)的时间开销，这又比HashMap与LinkedHashMap的O(N)时间复杂度好不少。

因此总结而言：如果只需要存储功能，使用HashMap与LinkedHashMap是一种更好的选择；如果还需要保证统计性能或者需要对Key按照一定规则进行排序，那么使用TreeMap是一种更好的选择。

#### 1.6.1 红黑树特性

红黑树的特性：

1、根节点与叶节点都是黑色节点，其中叶节点为Null节点；

2、每个红色节点的两个子节点都是黑色节点，换句话说就是不能有连续两个红色节点；

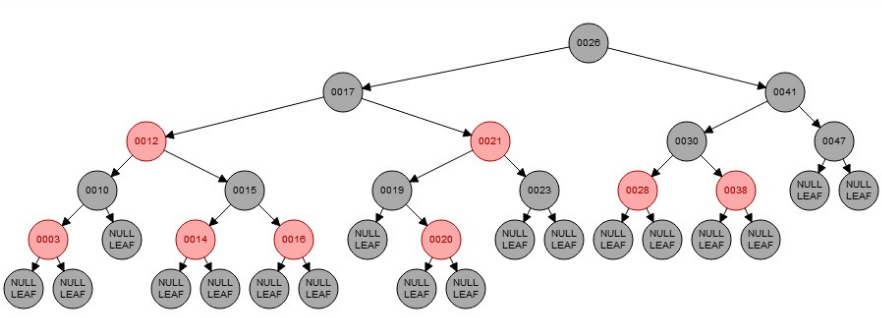
3、从根节点到所有叶子节点上的黑色节点数量是相同的；

上述的性质约束了红黑树的关键：从根到叶子的最长可能路径不多于最短可能路径的两倍长。得到这个结论的理由是：

1. 红黑树中最短的可能路径是全部为黑色节点的路径；
2. 红黑树中最长的可能路径是红黑相间的路径；

此时（2）正好是（1）的两倍长。结果就是这个树大致上是平衡的，因为比如插入、删除和查找某个值这样的操作最坏情况都要求与树的高度成比例，这个高度的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树，最终保证了红黑树能够以O(log2 n) 的时间复杂度进行搜索、插入、删除。

下面展示一张红黑树的实例图：



红黑树并不是高度的平衡树。所谓平衡树指的是一棵空树或它的左右两个子树的高度差的绝对值不超过1，它放弃了高度平衡的特性而只追求部分平衡，这种特性降低了插入、删除时对树旋转的要求，从而提升了树的整体性能。而其他平衡树比如AVL树虽然查找性能为性能是O(logn)，但是为了维护其平衡特性，可能要在插入、删除操作时进行多次的旋转，产生比较大的消耗。

#### 1.6.2 TreeMap基本数据结构

TreeMap的Key不允许为空，Value允许为空；其中Key重复会覆盖，Value允许重复；按照Key的自然顺序排序或者Comparator接口指定的排序算法进行排序；非线程安全；

TreeMap的节点Java代码定义为：

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

        K key;

        V value;

        Entry<K,V> left = null; // 左子节点引用

        Entry<K,V> right = null; // 右子节点引用

        Entry<K,V> parent; // 父节点引用

        boolean color = BLACK; // 节点颜色，true黑，false红

        ...

}

#### 1.6.3 添加数据

首先看一下TreeMap如何添加数据，测试代码为：

public class MapTest {

    @Test

    public void testTreeMap() {

        TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();

        treeMap.put(10, "10");

        treeMap.put(85, "85");

        treeMap.put(15, "15");

        treeMap.put(70, "70");

        treeMap.put(20, "20");

        treeMap.put(60, "60");

        treeMap.put(30, "30");

        treeMap.put(50, "50");

        for (Map.Entry<Integer, String> entry : treeMap.entrySet()) {

            System.out.println(entry.getKey() + ":" + entry.getValue());

        }

    }

}

本文接下来的内容会给出插入每条数据之后红黑树的数据结构是什么样子的。首先看一下treeMap的put方法的代码实现：

public V put(K key, V value) {

    Entry<K,V> t = root;

    if (t == null) {

        compare(key, key); // type (and possibly null) check

        root = new Entry<>(key, value, null);

        size = 1;

        modCount++;

        return null;

    }

    int cmp;

    Entry<K,V> parent;

    // split comparator and comparable paths

    Comparator<? super K> cpr = comparator;

    if (cpr != null) {

        do {

            parent = t;

            cmp = cpr.compare(key, t.key);

            if (cmp < 0)

                t = t.left;

            else if (cmp > 0)

                t = t.right;

            else

                return t.setValue(value);

        } while (t != null);

    }

    else {

        if (key == null)

            throw new NullPointerException();

        Comparable<? super K> k = (Comparable<? super K>) key;

        do {

            parent = t;

            cmp = k.compareTo(t.key);

            if (cmp < 0)

                t = t.left;

            else if (cmp > 0)

                t = t.right;

            else

                return t.setValue(value);

        } while (t != null);

    }

    Entry<K,V> e = new Entry<>(key, value, parent);

    if (cmp < 0)

        parent.left = e;

    else

        parent.right = e;

    fixAfterInsertion(e);

    size++;

    modCount++;

    return null;

}

从这段代码，先总结一下TreeMap添加数据的几个步骤：

1. 获取根节点，根节点为空，产生一个根节点，将其着色为黑色，退出余下流程；
2. 获取比较器，如果传入的Comparator接口不为空，使用传入的Comparator接口实现类进行比较；如果传入的Comparator接口为空，将Key强转为Comparable接口进行比较；
3. 从根节点开始逐一依照规定的排序算法进行比较，取比较值cmp，如果cmp=0，表示插入的Key已存在；如果cmp>0，取当前节点的右子节点；如果cmp<0，取当前节点的左子节点；
4. 排除插入的Key已存在的情况，第（3）步的比较一直比较到当前节点t的左子节点或右子节点为null，此时t就是我们寻找到的节点，cmp>0则准备往t的右子节点插入新节点，cmp<0则准备往t的左子节点插入新节点；
5. new出一个新节点，默认为黑色，根据cmp的值向t的左边或者右边进行插入；

6、插入之后进行修复，包括左旋、右旋、重新着色这些操作，让树保持平衡性；

第1~第5步都没有什么问题，红黑树最核心的应当是第6步插入数据之后进行的修复工作，对应的Java代码是TreeMap中的fixAfterInsertion方法，下面看一下put每个数据之后TreeMap都做了什么操作，借此来理清TreeMap的实现原理。

首先是put(10, "10")，由于此时TreeMap中没有任何节点，因此10为根且根节点为黑色节点，put(10, "10")之后的数据结构为：



接着是put(85, "85")， 85比10大，因此在10的右节点上，但由于85不是根节点，因此会执行fixAfterInsertion方法进行数据修正，看一下fixAfterInsertion方法代码实现：

private void fixAfterInsertion(Entry<K,V> x) {

    x.color = RED;

    while (x != null && x != root && x.parent.color == RED) {

        if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {

            Entry<K,V> y = rightOf(parentOf(parentOf(x)));

            if (colorOf(y) == RED) {

                setColor(parentOf(x), BLACK);

                setColor(y, BLACK);

                setColor(parentOf(parentOf(x)), RED);

                x = parentOf(parentOf(x));

            } else {

                if (x == rightOf(parentOf(x))) {

                    x = parentOf(x);

                    rotateLeft(x);

                }

                setColor(parentOf(x), BLACK);

                setColor(parentOf(parentOf(x)), RED);

                rotateRight(parentOf(parentOf(x)));

            }

        } else {

            Entry<K,V> y = leftOf(parentOf(parentOf(x)));

            if (colorOf(y) == RED) {

                setColor(parentOf(x), BLACK);

                setColor(y, BLACK);

                setColor(parentOf(parentOf(x)), RED);

                x = parentOf(parentOf(x));

            } else {

                if (x == leftOf(parentOf(x))) {

                    x = parentOf(x);

                    rotateRight(x);

                }

                setColor(parentOf(x), BLACK);

                setColor(parentOf(parentOf(x)), RED);

                rotateLeft(parentOf(parentOf(x)));

            }

        }

    }

    root.color = BLACK;

}

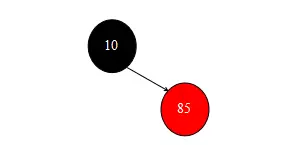
第2行的代码，它将默认的插入的那个节点着色成为红色：

根据红黑树的性质（3），红黑树要求从根节点到叶子所有叶子节点上经过的黑色节点个数是相同的，因此如果插入的节点着色为黑色，那必然有可能导致某条路径上的黑色节点数量大于其他路径上的黑色节点数量，因此默认插入的节点必须是红色的，以此来维持红黑树的性质（3）

当然插入节点着色为红色节点后，有可能导致的问题是违反性质（2），即出现连续两个红色节点，这就需要通过旋转操作去改变树的结构，解决这个问题。

接着看第4行的判断，前两个条件都满足，但是因为85这个节点的父节点是根节点的，根节点是黑色节点，因此这个条件不满足，while循环不进去，直接执行一次30行的代码给根节点着色为黑色（因为在旋转过程中有可能导致根节点为红色，而红黑树的根节点必须是黑色，因此最后不管根节点是不是黑色，都要重新着色确保根节点是黑色的）。

那么put(85, "85")之后，整个树的结构变为：



在看put(15, "15")之前，必须要先过一下fixAfterInsertion方法。第5行～第21行的代码和第21行~第38行的代码是一样的，无非一个是操作左子树另一个是操作右子树而已，因此就看前一半：

while (x != null && x != root && x.parent.color == RED) {

    if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {

        Entry<K,V> y = rightOf(parentOf(parentOf(x)));

        if (colorOf(y) == RED) {

            setColor(parentOf(x), BLACK);

            setColor(y, BLACK);

            setColor(parentOf(parentOf(x)), RED);

            x = parentOf(parentOf(x));

        } else {

            if (x == rightOf(parentOf(x))) {

                x = parentOf(x);

                rotateLeft(x);

            }

            setColor(parentOf(x), BLACK);

            setColor(parentOf(parentOf(x)), RED);

            rotateRight(parentOf(parentOf(x)));

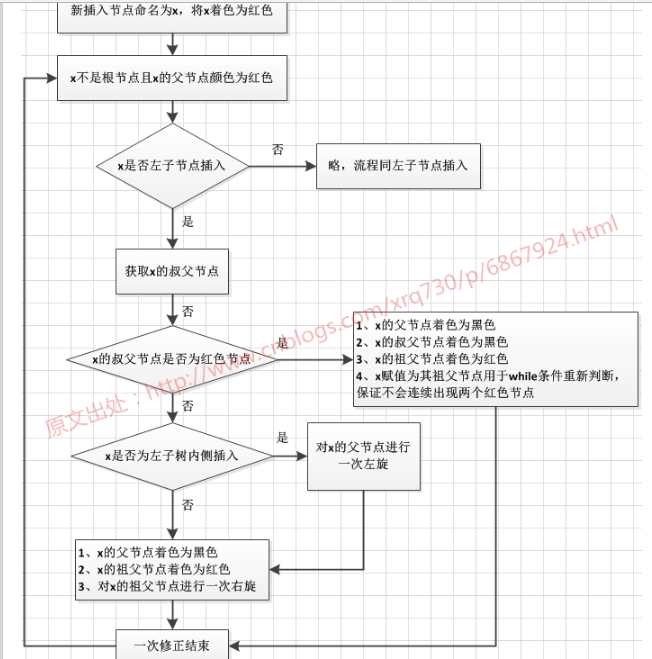
        }

    }

    ....

}

第2行的判断用语言描述出来就是：判断当前节点的父节点与当前节点的父节点的父节点的左子节点是否同一个节点。翻译一下就是：当前节点是否左子节点插入，这整段代码我用流程图描述一下：



## 2: 多线程

### 2.1 进程与线程

进程： 执行中的程序，一个进程至少包含一个线程；

线程： 进程中负责程序执行的执行单元，是进程中独立运行的子任务；

单线程：程序中只存在一个线程；

多线程：在一个程序中运行多个任务（目的是更好地使用CPU资源）；

多进程与多线程本质的区别就是每个进程拥有自己的一整套变量，而线程则共享数据。

### 2.2 线程对象

在Java中，每个线程都是[Thread](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html)类的实例。并发应用中有两种不同的线程创建策略：

1：控制线程的创建和管理，每当应用程序需要执行一个异步任务时就为其创建一个线程；

2：将线程的管理从应用程序中抽象出来作为执行器，应用程序将任务传递给执行器，由

执行器负责执行。

#### 2.2.1 定义并启动一个线程

1：提供一个Runnable对象。[Runnable](http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html)对象仅包含一个run()方法，在这个方法中定义的代码将在会线程中执行，即实现Runnable接口并重写(Override) run()方法，然后将Runnable对象传递给Thread类的构造函数即可。

1. **public** **class** HelloRunnable **implements** Runnable {
2. **public** **void** run() {
3. System.out.println("Hello from a thread!");
4. }
5. **public** **static** **void** main(String args[]) {
6. (**new** Thread(**new** HelloRunnable())).start();
7. }
8. }

2：继承Thread类。Thread类自身已实现了Runnable接口，但它的run()方法中并没有定义任何代码。应用程序可以继承与Thread类，并重写run()方法。

1. **public** **class** HelloThread **extends** Thread {
2. **public** **void** run() {
3. System.out.println("Hello from a thread!");
4. }
5. **public** **static** **void** main(String args[]) {
6. (**new** HelloThread()).start();
7. }
8. }

第一种更加通用，因为Runnable对象可以继承于其他类（Java只支持单继承，当一个类继承与Thread类后，就无法继承与其他类）。

不要调用Thread类或Runnable对象的run方法(run方法只是定义需要执行的任务)，直接调用run方法，只会执行同一个线程中的任务，而不会启动新线程；应该调用Thread.start()方法，该方法将创建一个执行run方法的新线程。新线程创建的过程不会阻塞主线程的后续执行。

使用ExecutorService、Callable、Future实现有返回结果的多线程；可返回值的任务必须实现Callable接口，类似的，无返回值的任务必须实现Runnable接口。执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了，再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。

#### 2.2.2 暂停

Thread.sleep()方法可以暂停当前线程一段时间，但不会释放“锁标志”。sleep()方法使当前线程进入阻塞状态，在指定时间内不会执行。这是一种使处理器时间可以被其他线程或者运用程序使用的有效方式。sleep()方法还可以用于调整线程执行节奏和等待其他有执行时间需求的线程。

由于操作系统的限制休眠时间并不能保证十分精确。休眠周期可以被interrups所终止。不管在任何情况下，都不应该假定调用了sleep()方法就可以将一个线程暂停一个十分精确的时间周期。

Object.wait()方法导致当前线程等待，且线程会释放掉它所占有的“锁标志”，从而使别的线程有机会抢占该锁。wait()和notify()必须在synchronized函数或synchronized block中调用，否则虽能编译通过但运行时报IllegalMonitorStateException的异常。

Thread.yield()方法的作用是放弃当前的CPU资源，将它让给其他的任务去占用CPU执行时间。但放弃时间不确定，有可能刚刚放弃，马上又获得CPU时间片。这里需要注意的是yield()方法和sleep方法一样，线程并不会让出锁，和wait不同。

#### 2.2.3 中断

当线程的run方法执行方法体中最后一句后，并经由执行return语句返回时，或者出现了在方法中没有捕获的异常时，线程将终止。

中断是给线程的一个指示，告诉它应该停止正在做的事并去做其他事情。一个线程响应中断请求很普遍的做法是让其终止。一个线程通过调用对被中断线程的Thread对象的interrupt()方法，发送中断信号(当对一个线程调用interrupt方法时，线程的中断状态将被置位，这是每个线程都具有的Boolean标志)。为了让中断机制正常工作，被中断的线程必须支持它自己的中断（即要自己处理中断）。

线程如何支持自身的中断？这取决于它当前正在做什么。如果线程正在频繁调用会抛InterruptedException异常的方法，在捕获异常之后，它只是从run()方法中返回。许多会抛InterruptedException异常的方法(如sleep()），被设计成接收到中断后取消它们当前的操作，并在立即返回。

中断机制通过使用称为中断状态的内部标记来实现。调用Thread.interrupt()向线程发送中断请求，线程的中断状态设置为true，即设置这个标记。当线程通过调用静态方法Thread.interrupted()检测中断时，中断状态会被清除。非静态的isInterrupted()方法被线程用来检测其他线程的中断状态，不改变中断状态标记。

当在一个被阻塞的线程(调用sleep或wait)上调用interrupt方法时，阻塞调用将会被InterruptedException异常中断。

#### 2.2.4 join

Thread.join()方法可以让一个线程等待另一个线程执行完成。若t是一个正在执行的Thread对象， t.join()将会使当前线程暂停执行并等待t执行完成。重载的join()方法可以让开发者自定义等待周期。然而，和sleep()方法一样join()方法依赖于操作系统的时间处理机制，你不能假定join()方法将会精确的等待你所定义的时长。 如同sleep()方法，join()方法响应中断并在中断时抛出InterruptedException。

#### 2.2.5 线程状态

线程有如下6种状态：

New (新创建)：

Runnable (可运行)：一旦调用start方法，线程处于Runnable状态；

Blocked (被阻塞)：当一个线程试图获取一个内部的对象锁（非java.util.concurrent库中的

锁）而该锁被其他线程持有，则该线程进入阻塞状态；当所有其他线程释放该锁，并且

线程调度器允许本线程持有它的时候，该线程变成非阻塞状态；

Waiting (等待)：当线程等待另一个线程通知调度器一个条件时，它自己进入等到状态；线程

暂时不活动，直到线程调度器重新激活它；

Timed Waiting (计时等待)：

Terminated (被终止)：run方法正常退出，或因为一个没有捕获的异常终止了run方法；

现在所有桌面以及服务器操作系统都使用抢占式调度，手机等小型设备使用协作式调度。

线程调度的细节依赖于操作系统提供的服务。抢占式调度系统给每一个可运行线程一个时间片来执行任务。当时间片用完，操作系统剥夺该线程的运行权，并给另一个线程运行机会。当选择下一个线程时，操作系统会考虑线程的优先级（1,5,20）；在给定的任何时刻，一个可运行的线程可能正在运行也可能没有运行；

#### 2.2.6 守护线程

Java中有两种线程，一种是用户线程，另一种是守护线程。当进程中不存在非守护线程了，则守护线程自动销毁。通过setDaemon(true)设置线程为后台线程。注意thread.setDaemon(true)要在thread.start()之前设置，否则报IllegalThreadStateException异常；在Daemon线程中产生的新线程也是Daemon的；在使用ExecutorSerice等多线程框架时，会把守护线程转换为用户线程，并且把优先级设置为Thread.NORM\_PRIORITY。在构建Daemon线程时，不能依靠finally块中的内容来确保执行关闭或清理资源的逻辑。

### 2.3 同步

线程间的通信主要是通过共享域和引用相同的对象。这种通信方式非常高效，不过可能会引发两种错误：线程干扰和内存一致性错误。防止这些错误发生的方法是同步。 不过，同步会引起线程竞争，当两个或多个线程试图同时访问相同的资源，随之就导致Java运行时环境执行其中一个或多个线程比原先慢很多，甚至执行被挂起，这就出现了线程竞争。线程饥饿和活锁都属于线程竞争的范畴。

#### 2.3.1 线程干扰

当两个运行在不同的线程中却作用在相同的数据上的操作交替执行时，就发生了线程干扰。

假设线程A调用increment()的同时线程B调用decrement().如果c的初始值为0，线程A和B之间的交替执行顺序可能是下面这样：

* 线程A：获取c；
* 线程B：获取c；
* 线程A：对获取的值加1，结果为1；
* 线程B：对获取的值减1，结果为-1；
* 线程A：结果写回到c,c现在是1；
* 线程B：结果写回到c,c现在是-1；

线程A的结果因为被线程B覆盖而丢失了。这个交替执行的结果只是其中一种可能性。在不同的环境下，可能是线程B的结果丢失了，也可能是不会出任何问题。由于结果是不可预知的，所以线程干扰的bug很难检测和修复。

#### 2.3.2 内存一致性错误

当不同的线程对相同的数据产生不一致的视图时会发生内存一致性错误。避免内存一致性错误的关键是理解happens-before关系。这种关系只是确保一个特定语句的写内存操作对另外一个特定的语句可见。

假设定义和初始化了一个简单int字段：

1. int counter =0 ;

这个counter字段被A，B两个线程共享。假设线程A对counter执行递增:

1. counter++;

然后，很快的，线程B输出counter:

1. System.out.println(counter);

如果这两个语句已经在同一个线程中被执行过，那么输出的值应该是“1”。不过如果这两个语句在不同的线程中分开执行，那输出的值很可能是“0”，因为无法保证线程A对counter的改动对线程B是可见的——除非我们在这两个语句之间已经建立了happens-before关系。

当一条语句调用Thread.start方法时，和该语句有happens-before关系的每一条语句，跟新线程执行的每一条语句同样有happens-before关系。创建新线程之前的代码的执行结果对线新线程是可见的。

当一个线程终止并且当导致另一个线程中Thread.join返回时，被终止的线程执行的所有语句和在join返回成功之后的所有语句间有happens-before关系。线程中代码的执行结果对执行join操作的线程是可见的。

#### 2.3.3 同步方法

Java编程语言提供两种同步方式：同步方法和同步语句。要让一个方法成为同步方法，只需要在方法声明中加上synchronized关键字：首先， 当一个线程正在执行对象的同步方法时，所有其他调用该对象同步方法的线程会被阻塞（挂起执行）,直到第一个线程处理完该对象。 其次，当一个同步方法退出时，它会自动跟该对象同步方法的任意后续调用建立起一种happens-before关系。这确保对象状态的改变对所有线程是可见的。

构造方法不能是同步的——构造方法加synchronized关键字会报语法错误。同步的构造方法没有意义，因为当这个对象被创建的时候，只有创建对象的线程能访问它。

同步(synchronized)方法使用一种简单的策略来防止线程干扰和内存一致性错误：如果一个对象对多个线程可见，对象域上的所有读写操作都是通过synchronized方法来完成的。（一个重要的例外：final域，在对象被创建后不可修改,能被非synchronized方法安全的读取）。synchronized同步策略很有效，不过会引起活跃度问题。

#### 2.3.4 内部锁

同步机制的建立是基于其内部一个叫内部锁或者监视锁的实体。（在Java API规范中通常被称为监视器。）内部锁在同步机制中起到两方面的作用：对一个对象的排他性访问；建立一种happens-before关系，而这种关系正是可见性问题的关键所在。

 每个对象都有一个与之关联的内部锁。通常当一个线程需要排他性的访问一个对象的域时，首先需要请求该对象的内部锁，当访问结束时释放内部锁。在线程获得内部锁到释放内部锁的这段时间里，我们说线程拥有这个内部锁。那么当一个线程拥有一个内部锁时，其他线程将无法获得该内部锁。其他线程如果去尝试获得该内部锁，则会被阻塞。

当线程释放一个内部锁时，该操作和对该锁的后续请求间将建立happens-before关系。

#### 2.3.5 同步方法中的锁

当线程调用一个同步方法时，它会自动请求该方法所在对象的内部锁。当方法返回结束时则自动释放该内部锁，即使退出是由于发生了未捕获的异常，内部锁也会被释放。 由于静态方法是和类（而不是对象）相关的，所以线程会请求类对象(Class Object)的内部锁。因此用来控制类的静态域访问的锁不同于控制对象访问的锁。

#### 2.3.6 同步块

另外一种同步的方法是使用同步块，同步块必须指定所请求的是哪个对象的内部锁：

1. **public** **void** addName(String name) {
2. **synchronized**(**this**) {
3. lastName = name;
4. nameCount++;
5. }
6. nameList.add(name);
7. }

在上面的例子中，addName方法需要使lastName和nameCount的更改保持同步，而且要避免同步调用该对象的其他方法。如果不使用同步块，那么必须要定义一个额外的非同步方法，而这个方法仅仅是用来调用nameList.add。

使用同步块对于更细粒度的同步很有帮助。例如类MsLunch有两个实例域c1和c2，他们并不会同时使用（即c1和c2是彼此无关的两个域），所有对这两个域的更新都需要同步，但是完全不需要防止c1的修改和c2的修改相互之间干扰（这样做只会产生不必要的阻塞而降低了并发性）。这种情况下不必使用同步方法，可以使用和this对象相关的锁。这里我们创建了两个“锁”对象（译者注：起到加锁效果的普通对象lock1和lock2）。

1. **public** **class** MsLunch {
2. **private** **long** c1 = 0;
3. **private** **long** c2 = 0;
4. **private** Object lock1 = **new** Object();
5. **private** Object lock2 = **new** Object();
6. **public** **void** inc1() {
7. **synchronized**(lock1) {
8. c1++;
9. }
10. }
11. **public** **void** inc2() {
12. **synchronized**(lock2) {
13. c2++;
14. }
15. }
16. }

使用这种方法时要特别小心，需要十分确定c1和c2是彼此无关的域。

#### 2.3.7 可重入同步

一个线程不能获得其他线程所拥有的锁。但是它可以获得自己已经拥有的锁。允许一个线程多次获得同一个锁实现了可重入同步。这里描述了一种同步代码的场景，直接的或间接地，调用了一个也拥有同步代码的方法，且两边的代码使用的是同一把锁。如果没有这种可重入的同步机制，同步代码则需要采取许多额外的预防措施以防止线程阻塞自己。

#### 2.3.8 原子访问

在编程过程中，原子操作是指所有操作都同时发生。原子操作不能被中途打断：要么全做，要么不做。原子操作在完成前不会有看得见的副作用。   
  
我们发现像c++这样的增量表达式，并没有描述原子操作。即使是非常简单的表达式也能够定义成能被分解为其他操作的复杂操作。然而，有些操作你可以定义为原子的： 

* 对引用变量和大部分基本类型变量（除long和double之外）的读写是原子的。
* 对所有声明为volatile的变量（包括long和double变量）的读写是原子的。

原子操作不会交错，于是可以放心使用，不必担心线程干扰。然而，这并不能完全消除原子操作上的同步，因为内存一致性错误仍可能发生。使用volatile变量可以降低内存一致性错误的风险，因为对volatile变量的任意写操作，对于后续在该变量上的读操作建立了happens-before关系。这意味着volatile变量的修改对于其他线程总是可见的。更重要的是，这同时也意味着当一个线程读取一个volatile变量时，它不仅能看到该变量最新的修改，而且也能看到致使该改变发生的代码的副效应。   
  
使用简单的原子变量访问比通过同步代码来访问更高效，但是需要程序员更加谨慎以避免内存一致性错误。至于这额外的付出是否值得，得看应用的大小和复杂度。

### 2.4 活跃度

一个并发应用程序能及时执行的能力称为活跃性。本节将介绍最常见的活跃性问题：死锁（deadlock）、饥饿（starvation）和活锁（livelock）。

#### 2.4.1 死锁

死锁描述了这样一种情景，两个或多个线程永久阻塞，互相等待对方释放资源。

下面是一个例子。 Alphone和Gaston是朋友，都很讲究礼节。礼节有一个严格的规矩，当你向一个朋友鞠躬时，你必须保持鞠躬的姿势，直到你的朋友有机会回鞠给你。不幸的是，这个规矩没有算上两个朋友相互同时鞠躬的可能。

#### 2.4.2 饥饿和活锁

饥饿是指当一个线程不能正常的访问共享资源并且不能正常执行的情况。这通常在共享资源被其他“贪心”的线程长期时发生。举个例子，假设一个对象提供了一个同步方法，这个方法通常需要执行很长一段时间才返回。如果一个线程经常调用这个方法，那么其他需要同步的访问这个对象的线程就经常会被阻塞。

一个线程通常会有会响应其他线程的活动。如果其他线程也会响应另一个线程的活动，那么就有可能发生活锁。同死锁一样，发生活锁的线程无法继续执行。然而线程并没有阻塞——他们在忙于响应对方无法恢复工作。这就相当于两个在走廊相遇的人：Alphonse向他自己的左边靠想让Gaston过去，而Gaston向他的右边靠想让Alphonse过去。可见他们阻塞了对方。Alphonse向他的右边靠，而Gaston向他的左边靠，他们还是阻塞了对方。

### 2.5 保护块

多线程之间经常需要协同工作，最常见的方式是使用Guarded Blocks，它循环检查一个条件（通常初始值为true），直到条件发生变化才跳出循环继续执行。在使用Guarded Blocks时有以下几个步骤需要注意：

1： 假设guardedJoy()方法必须要等待另一线程为共享变量joy设值才能继续执行。那么理论上可以用一个简单的条件循环来实现，但在等待过程中guardedJoy方法不停的检查循环条件实际上是一种资源浪费。

1. **public** **void** guardedJoy() {
2. // Simple loop guard. Wastes
3. // processor time. Don't do this!
4. **while**(!joy) {}
5. System.out.println("Joy has been achieved!");
6. }

更加高效的方法是调用Object.wait将当前线程挂起，直到有另一线程发起事件通知（尽管通知的事件不一定是当前线程等待的事件）。

1. **public** **synchronized** **void** guardedJoy() {
2. // This guard only loops once for each special event, which may not
3. // be the event we're waiting for.
4. **while**(!joy) {
5. **try** {
6. wait();
7. } **catch** (InterruptedException e) {}
8. }
9. System.out.println("Joy and efficiency have been achieved!");
10. }

注意：一定要在循环里面调用wait方法，不要想当然的认为线程唤醒后循环条件一定发生了改变。 和其他可以暂停线程执行的方法一样，wait方法会抛出InterruptedException，在上面的例子中，因为我们关心的是joy的值，所以忽略了InterruptedException。 为什么guardedJoy是synchronized方法？假设d是用来调用wait的对象，当一个线程调用d.wait，它必须要拥有d的内部锁（否则会抛出异常），获得d的内部锁的最简单方法是在一个synchronized方法里面调用wait。

当一个线程调用wait方法时，它释放锁并挂起。然后另一个线程请求并获得这个锁并调用[Object.notifyAll](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)通知所有等待该锁的线程。

1. **public** **synchronized** notifyJoy() {
2. joy = **true**;
3. notifyAll();
4. }

当第二个线程释放这个该锁后，第一个线程再次请求该锁，从wait方法返回并继续执行。   
注意：还有另外一个通知方法，notify()，它只会唤醒一个线程。但由于它并不允许指定哪一个线程被唤醒，所以一般只在大规模并发应用（即系统有大量相似任务的线程）中使用。因为对于大规模并发应用，我们其实并不关心哪一个线程被唤醒。   
  
现在我们使用Guarded blocks创建一个生产者/消费者应用。这类应用需要在两个线程之间共享数据：生产者生产数据，消费者使用数据。两个线程通过共享对象通信。在这里，线程协同工作的关键是：生产者发布数据之前，消费者不能够去读取数据；消费者没有读取旧数据前，生产者不能发布新数据。   
  
在下面的例子中，数据通过[Drop](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Drop.java)对象共享的一系列文本消息： 

**Java代码**

1. **public** **class** Drop {
2. // Message sent from producer
3. // to consumer.
4. **private** String message;
5. // True if consumer should wait
6. // for producer to send message,
7. // false if producer should wait for
8. // consumer to retrieve message.
9. **private** **boolean** empty = **true**;
11. **public** **synchronized** String take() {
12. // Wait until message is
13. // available.
14. **while** (empty) {
15. **try** {
16. wait();
17. } **catch** (InterruptedException e) {}
18. }
19. // Toggle status.
20. empty = **true**;
21. // Notify producer that
22. // status has changed.
23. notifyAll();
24. **return** message;
25. }
27. **public** **synchronized** **void** put(String message) {
28. // Wait until message has
29. // been retrieved.
30. **while** (!empty) {
31. **try** {
32. wait();
33. } **catch** (InterruptedException e) {}
34. }
35. // Toggle status.
36. empty = **false**;
37. // Store message.
38. **this**.message = message;
39. // Notify consumer that status
40. // has changed.
41. notifyAll();
42. }
43. }

[Producer](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Producer.java)是生产者线程，发送一组消息，字符串DONE表示所有消息都已经发送完成。为了模拟现实情况，生产者线程还会在消息发送时随机的暂停。 

**Java代码**

1. **import** java.util.Random;
3. **public** **class** Producer **implements** Runnable {
4. **private** Drop drop;
6. **public** Producer(Drop drop) {
7. **this**.drop = drop;
8. }
10. **public** **void** run() {
11. String importantInfo[] = {
12. "Mares eat oats",
13. "Does eat oats",
14. "Little lambs eat ivy",
15. "A kid will eat ivy too"
16. };
17. Random random = **new** Random();
19. **for** (**int** i = 0;
20. i &lt; importantInfo.length;
21. i++) {
22. drop.put(importantInfo[i]);
23. **try** {
24. Thread.sleep(random.nextInt(5000));
25. } **catch** (InterruptedException e) {}
26. }
27. drop.put("DONE");
28. }
29. }

[Consumer](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Consumer.java)是消费者线程，读取消息并打印出来，直到读取到字符串DONE为止。消费者线程在消息读取时也会随机的暂停。 

**Java代码**

1. **import** java.util.Random;
3. **public** **class** Consumer **implements** Runnable {
4. **private** Drop drop;
6. **public** Consumer(Drop drop) {
7. **this**.drop = drop;
8. }
10. **public** **void** run() {
11. Random random = **new** Random();
12. **for** (String message = drop.take();
13. ! message.equals("DONE");
14. message = drop.take()) {
15. System.out.format("MESSAGE RECEIVED: %s%n", message);
16. **try** {
17. Thread.sleep(random.nextInt(5000));
18. } **catch** (InterruptedException e) {}
19. }
20. }
21. }

[ProducerConsumerExample](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ProducerConsumerExample.java)是主线程，它启动生产者线程和消费者线程。 

**Java代码**

1. **public** **class** ProducerConsumerExample {
2. **public** **static** **void** main(String[] args) {
3. Drop drop = **new** Drop();
4. (**new** Thread(**new** Producer(drop))).start();
5. (**new** Thread(**new** Consumer(drop))).start();
6. }
7. }

注意：Drop类是用来演示Guarded Blocks如何工作的。为了避免重新发明轮子，当你尝试创建自己的共享数据对象时，请查看[Java Collections Framework](http://docs.oracle.com/javase/tutorial/collections/index.html)中已有的数据结构。如需更多信息，请参考[Questions and Exercises](http://docs.oracle.com/javase/tutorial/essential/concurrency/QandE/questions.html)。

### 2.6 不可变对象

一个对象如果在创建后不能被修改，那么就称为不可变对象。在并发编程中，一种被普遍认可的原则就是：尽可能的使用不可变对象来创建简单、可靠的代码。 由于创建后不能被修改，所以不会出现由于线程干扰产生的错误或是内存一致性错误。

### 2.7 高级并发对象

Java 5.0新增的一些高级并发特征。大多数特征已经在新的java.util.concurrent包中实现。Java集合框架中也定义了新的并发数据结构。

* 锁对象提供了可以简化许多并发应用的锁的惯用法。
* Executors为加载和管理线程定义了高级API。Executors的实现由java.util.concurrent包提供，提供了适合大规模应用的线程池管理。
* 并发集合简化了大型数据集合管理，且极大的减少了同步的需求。
* 原子变量有减小同步粒度和避免内存一致性错误的特征。
* 并发随机数（JDK7）提供了高效的多线程生成伪随机数的方法。

#### 2.7.1 锁对象

同步代码依赖于一种简单的可重入锁。这种锁使用简单，但也有诸多限制。[java.util.concurrent.locks](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/package-summary.html)包提供了更复杂的锁。我们不会详细考察这个包，但会重点关注其最基本的接口，锁。  锁对象作用非常类似同步代码使用的隐式锁。如同隐式锁，每次只有一个线程可以获得锁对象。通过关联[Condition](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html)对象，锁对象也支持wait/notify机制。 锁对象之于隐式锁最大的优势在于，它们有能力收回获得锁的尝试。如果当前锁对象不可用，或者锁请求超时（如果超时时间已指定），tryLock方法会收回获取锁的请求。如果在锁获取前，另一个线程发送了一个中断，lockInterruptibly方法也会收回获取锁的请求。 让我们使用锁对象来解决我们在[活跃度](http://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html)中见到的死锁问题。

#### 2.7.2 执行器

Thread对象表示的线程和Runnable对象表示的线程所执行的任务之间是紧耦合的。这对于小型应用程序来说没问题，但对于大规模并发应用来说，合理的做法是将线程的创建与管理和程序的其他部分分离开。封装这些功能的对象就是执行器，接下来的部分将讲详细描述执行器。

* [执行器接口](http://ifeve.com/oracle-executor-interfaces/)定义了三种类型的执行器对象。
* [线程池](http://ifeve.com/thread-pool/)是最常见的一种执行器的实现。
* [Fork/Join](http://ifeve.com/forkjoin/)是JDK 7中引入的并发框架。

**执行器接口** ：

java.util.concurrent中包括三个Executor接口：

* Executor，一个运行新任务的简单接口。
* ExecutorService，扩展了Executor接口。添加了一些用来管理执行器生命周期和任务生命周期的方法。
* ScheduledExecutorService，扩展了ExecutorService。支持Future和定时任务。

通常来说，指向Executor对象的变量应被声明为以上三种接口之一，而不是具体的实现类。

**Executor接口**   
  
[Executor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html)接口只有一个execute方法，用来替代通常创建（启动）线程的方法。例如：r是一个Runnable对象，e是一个Executor对象。可以使用

**Java代码**

1. e.execute(r);

来代替

**Java代码**

1. (**new** Thread(r)).start();

但execute方法没有定义具体的实现方式。对于不同的Executor实现，execute方法可能是创建一个新线程并立即启动，但更有可能是使用已有的工作线程运行r，或者将r放入到队列中等待可用的工作线程。（我们将在线程池一节中描述工作线程。）

**ExecutorService接口**   
  
[ExecutorService](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html)接口在提供了execute方法的同时，新加了更加通用的submit方法。submit方法除了和execute方法一样可以接受Runnable对象作为参数，还可以接受Callable对象作为参数。使用Callable对象可以能使任务返还执行的结果。通过submit方法返回的Future对象可以读取Callable任务的执行结果，或是管理Callable任务和Runnable任务的状态。 ExecutorService也提供了批量运行Callable任务的方法。最后，ExecutorService还提供了一些关闭执行器的方法。如果需要支持即时关闭，执行器所执行的任务需要正确处理中断。   
  
**ScheduledExecutorService接口**   
  
[ScheduledExecutorService](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html)扩展ExecutorService接口并添加了schedule方法。调用schedule方法可以在指定的延时后执行一个Runnable或者Callable任务。ScheduledExecutorService接口还定义了按照指定时间间隔定期执行任务的scheduleAtFixedRate方法和scheduleWithFixedDelay方法。

**4.  线程池**   
  
在java.util.concurrent包中多数的执行器实现都使用了由工作线程组成的线程池，工作线程独立于所它所执行的Runnable任务和Callable任务，并且常用来执行多个任务。 使用工作线程可以使创建线程的开销最小化。   
  
在大规模并发应用中，创建大量的Thread对象会占用占用大量系统内存，分配和回收这些对象会产生很大的开销。一种最常见的线程池是固定大小的线程池。这种线程池始终有一定数量的线程在运行，如果一个线程由于某种原因终止运行了，线程池会自动创建一个新的线程来代替它。需要执行的任务通过一个内部队列提交给线程，当没有更多的工作线程可以用来执行任务时，队列保存额外的任务。 使用固定大小的线程池一个很重要的好处是可以实现优雅退化。例如一个Web服务器，每一个HTTP请求都是由一个单独的线程来处理的，如果为每一个HTTP都创建一个新线程，那么当系统的开销超出其能力时，会突然地对所有请求都停止响应。如果限制Web服务器可以创建的线程数量，那么它就不必立即处理所有收到的请求，而是在有能力处理请求时才处理。 创建一个使用线程池的执行器最简单的方法是调用[java.util.concurrent.Executors](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html)的[newFixedThreadPool](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html)方法。Executors类还提供了下列一下方法： 

* [newCachedThreadPool](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html)方法创建了一个可扩展的线程池。适合用来启动很多短任务的应用程序。
* [newSingleThreadExecutor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html)方法创建了每次执行一个任务的执行器。
* 还有一些创建ScheduledExecutorService执行器的方法。

如果上面的方法都不满足需要，可以尝试[java.util.concurrent.ThreadPoolExecutor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html)或者[java.util.concurrent.ScheduledThreadPoolExecutor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html)。   
  
**5.  Fork/Joint**   
  
fork/join框架是ExecutorService接口的一种具体实现，目的是为了帮助你更好地利用多处理器带来的好处。它是为那些能够被递归地拆解成子任务的工作类型量身设计的。其目的在于能够使用所有可用的运算能力来提升你的应用的性能。   类似于ExecutorService接口的其他实现，fork/join框架会将任务分发给线程池中的工作线程。fork/join框架的独特之处在与它使用工作窃取(work-stealing)算法。完成自己的工作而处于空闲的工作线程能够从其他仍然处于忙碌(busy)状态的工作线程处窃取等待执行的任务。 fork/join框架的核心是[ForkJoinPool](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html)类，它是对AbstractExecutorService类的扩展。ForkJoinPool实现了工作偷取算法，并可以执行[ForkJoinTask](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html)任务。   
  
**基本使用方法**   
  
使用fork/join框架的第一步是编写执行一部分工作的代码。你的代码结构看起来应该与下面所示的伪代码类似： 

**Java代码**

1. **if** (当前这个任务工作量足够小)
2. 直接完成这个任务
3. **else**
4. 将这个任务或这部分工作分解成两个部分
5. 分别触发(invoke)这两个子任务的执行，并等待结果

你需要将这段代码包裹在一个ForkJoinTask的子类中。不过，通常情况下会使用一种更为具体的的类型，或者是[RecursiveTask](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RecursiveTask.html)(会返回一个结果)，或者是[RecursiveAction](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RecursiveAction.html)。 当你的ForkJoinTask子类准备好了，创建一个代表所有需要完成工作的对象，然后将其作为参数传递给一个ForkJoinPool实例的invoke()方法即可。   
  
**要清晰，先模糊**   
  
想要了解fork/join框架的基本工作原理，接下来的这个例子会有所帮助。假设你想要模糊一张图片。原始的source图片由一个整数的数组表示，每个整数表示一个像素点的颜色数值。与source图片相同，模糊之后的destination图片也由一个整数数组表示。 对图片的模糊操作是通过对source数组中的每一个像素点进行处理完成的。处理的过程是这样的：将每个像素点的色值取出，与周围像素的色值（红、黄、蓝三个组成部分）放在一起取平均值，得到的结果被放入destination数组。因为一张图片会由一个很大的数组来表示，这个流程会花费一段较长的时间。如果使用fork/join框架来实现这个模糊算法，你就能够借助多处理器系统的并行处理能力。下面是上述算法结合fork/join框架的一种简单实现： 

**Java代码**

1. **public** **class** ForkBlur **extends** RecursiveAction {
2. **private** **int**[] mSource;
3. **private** **int** mStart;
4. **private** **int** mLength;
5. **private** **int**[] mDestination;
7. // Processing window size; should be odd.
8. **private** **int** mBlurWidth = 15;
10. **public** ForkBlur(**int**[] src, **int** start, **int** length, **int**[] dst) {
11. mSource = src;
12. mStart = start;
13. mLength = length;
14. mDestination = dst;
15. }
17. **protected** **void** computeDirectly() {
18. **int** sidePixels = (mBlurWidth - 1) / 2;
19. **for** (**int** index = mStart; index &lt; mStart + mLength; index++) {
20. // Calculate average.
21. **float** rt = 0, gt = 0, bt = 0;
22. **for** (**int** mi = -sidePixels; mi &lt;= sidePixels; mi++) {
23. **int** mindex = Math.min(Math.max(mi + index, 0),
24. mSource.length - 1);
25. **int** pixel = mSource[mindex];
26. rt += (**float**)((pixel &amp; 0x00ff0000) &gt;&gt; 16)
27. / mBlurWidth;
28. gt += (**float**)((pixel &amp; 0x0000ff00) &gt;&gt;  8)
29. / mBlurWidth;
30. bt += (**float**)((pixel &amp; 0x000000ff) &gt;&gt;  0)
31. / mBlurWidth;
32. }
34. // Reassemble destination pixel.
35. **int** dpixel = (0xff000000     ) |
36. (((**int**)rt) &lt;&lt; 16) |
37. (((**int**)gt) &lt;&lt;  8) |
38. (((**int**)bt) &lt;&lt;  0);
39. mDestination[index] = dpixel;
40. }
41. }

接下来你需要实现父类中的compute()方法，它会直接执行模糊处理，或者将当前的工作拆分成两个更小的任务。数组的长度可以作为一个简单的阀值来判断任务是应该直接完成还是应该被拆分。 

**Java代码**

1. **protected** **static** **int** sThreshold = 100000;
3. **protected** **void** compute() {
4. **if** (mLength &lt; sThreshold) {
5. computeDirectly();
6. **return**;
7. }
9. **int** split = mLength / 2;
11. invokeAll(**new** ForkBlur(mSource, mStart, split, mDestination),
12. **new** ForkBlur(mSource, mStart + split, mLength - split,
13. mDestination));
14. }

如果前面这个方法是在一个RecursiveAction的子类中，那么设置任务在ForkJoinPool中执行就再直观不过了。通常会包含以下一些步骤：   
  
(1) 创建一个表示所有需要完成工作的任务。 

**Java代码**

1. // source image pixels are in src
2. // destination image pixels are in dst
3. ForkBlur fb = **new** ForkBlur(src, 0, src.length, dst);

(2) 创建将要用来执行任务的ForkJoinPool。 

**Java代码**

1. ForkJoinPool pool = **new** ForkJoinPool();

(3) 执行任务。 

**Java代码**

1. pool.invoke(fb);

想要浏览完成的源代码，请查看[ForkBlur](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ForkBlur.java)，其中还包含一些创建destination图片文件的额外代码。   
  
**标准实现**   
  
除了能够使用fork/join框架来实现能够在多处理系统中被并行执行的定制化算法（如前文中的ForkBlur.java例子），在Java SE中一些比较常用的功能点也已经使用fork/join框架来实现了。在Java SE 8中，java.util.Arrays类的一系列parallelSort()方法就使用了fork/join来实现。这些方法与sort()系列方法很类似，但是通过使用fork/join框架，借助了并发来完成相关工作。在多处理器系统中，对大数组的并行排序会比串行排序更快。这些方法究竟是如何运用fork/join框架并不在本教程的讨论范围内。想要了解更多的信息，请参见Java API文档。 其他采用了fork/join框架的方法还包括java.util.streams包中的一些方法，此包是作为Java SE 8发行版中[Project Lambda](http://openjdk.java.net/projects/lambda/)的一部分。想要了解更多信息，请参见[Lambda Expressions](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)一节。   
  
**6.  并发集合**   
  
java.util.concurrent包囊括了Java集合框架的一些附加类。它们也最容易按照集合类所提供的接口来进行分类： 

* [BlockingQueue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html)定义了一个先进先出的数据结构，当你尝试往满队列中添加元素，或者从空队列中获取元素时，将会阻塞或者超时。
* [ConcurrentMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentMap.html)是[java.util.Map](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)的子接口，定义了一些有用的原子操作。移除或者替换键值对的操作只有当key存在时才能进行，而新增操作只有当key不存在时。使这些操作原子化，可以避免同步。ConcurrentMap的标准实现是[ConcurrentHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html)，它是[HashMap](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html)的并发模式。
* [ConcurrentNavigableMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentNavigableMap.html)是ConcurrentMap的子接口，支持近似匹配。ConcurrentNavigableMap的标准实现是[ConcurrentSkipListMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html)，它是[TreeMap](http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html)的并发模式。
* 所有这些集合，通过 在集合里新增对象和访问或移除对象的操作之间，定义一个happens-before的关系，来帮助程序员避免[内存一致性错误](http://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html)。

**7.  原子变量**   
  
[java.util.concurrent.atomic](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html)包定义了对单一变量进行原子操作的类。所有的类都提供了get和set方法，可以使用它们像读写volatile变量一样读写原子类。就是说，同一变量上的一个set操作对于任意后续的get操作存在happens-before关系。原子的compareAndSet方法也有内存一致性特点，就像应用到整型原子变量中的简单原子算法。   为了看看这个包如何使用，让我们返回到最初用于演示线程干扰的[Counter](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Counter.java)类： 

**Java代码**

1. **class** Counter {
2. **private** **int** c = 0;
3. **public** **void** increment() {
4. c++;
5. }
7. **public** **void** decrement() {
8. c--;
9. }
11. **public** **int** value() {
12. **return** c;
13. }
14. }

使用同步是一种使Counter类变得线程安全的方法，如[SynchronizedCounter](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/SynchronizedCounter.java)： 

**Java代码**

1. **class** SynchronizedCounter {
2. **private** **int** c = 0;
3. **public** **synchronized** **void** increment() {
4. c++;
5. }
6. **public** **synchronized** **void** decrement() {
7. c--;
8. }
9. **public** **synchronized** **int** value() {
10. **return** c;
11. }
12. }

对于这个简单的类，同步是一种可接受的解决方案。但是对于更复杂的类，我们可能想要避免不必要同步所带来的活跃度影响。将int替换为AtomicInteger允许我们在不进行同步的情况下阻止线程干扰，如[AtomicCounter](http://docs.oracle.com/javase/tutorial/essential/concurrency/examples/AtomicCounter.java)： 

**Java代码**

1. **import** java.util.concurrent.atomic.AtomicInteger;
2. **class** AtomicCounter {
3. **private** AtomicInteger c = **new** AtomicInteger(0);
4. **public** **void** increment() {
5. c.incrementAndGet();
6. }
8. **public** **void** decrement() {
9. c.decrementAndGet();
10. }
12. **public** **int** value() {
13. **return** c.get();
14. }

**8.  并发随机数**   
  
在JDK7中，java.util.concurrent包含了一个相当便利的类，ThreadLocalRandom，当应用程序期望在多个线程或ForkJoinTasks中使用随机数时。   
  
对于并发访问，使用TheadLocalRandom代替Math.random()可以减少竞争，从而获得更好的性能。   
  
你只需调用ThreadLocalRandom.current()， 然后调用它的其中一个方法去获取一个随机数即可。下面是一个例子： 

**Java代码**

1. **int** r = ThreadLocalRandom.current().nextInt(4,77);

### 2.12 多线程面试题

1： SimpleDateFormat 是线程安全类吗？

SimpleDateFormat不是线程安全类，其父类DateFormat中有一个Calendar成员变量，在多线程场景下，会发生资源共享造成前后不一致的问题；

最简单的方法就是每次要使用 SimpleDateFormat 时都创建一个局部的 SimpleDateFormat 对象。局部变量，自然就不存在线程安全的问题了。但如果需要频繁进行调用的话，每次都要创建新的对象，开销太大。

第二种方式，就是对 SimpleDateFormat 进行加锁，这样可以确保同一时间只有一个线程可以持有锁，进而解决线程安全的问题。但是这种方法在多线程竞争激烈的时候会带来效率问题。

第三种方式，就是使用 ThreadLocal。 ThreadLocal 可以确保每个线程都可以得到单独的一个 SimpleDateFormat 的对象，那么自然也就不存在竞争问题了。

2：ThreadLocal 用途是什么，原理是什么，用的时候要注意什么

用途：

Java中的ThreadLocal可以将一个非线程安全的对象转换成支持多线程访问的对象。最常见的使用是包裹线程不安全的工具类对象，比如Random SimpleDateFormat之类。

对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而 ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队 访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

ThreadLocal占用内存较大，但是速度快，而线程同步相对内存占用小，但是速度慢。如果在内存比较充足的情况，对并发部分的执行效率要求很高的话，那么就用ThreadLocal。一般情况下用同步机制还是居多的。

原理：

ThreadLocal提供了set和get访问器用来访问与当前线程相关联的线程局部变量。从ThreadLocal的get函数中可知，先获取当前线程t，然后getmap函数是用t作为参数，即 get函数就是从当前线程的threadlocalmap中取出当前线程对应的变量的副本【注意，变量是保存在线程中的，而不是保存在ThreadLocal变量中】。当前线程中，有一个变量引用名字是threadLocals，这个引用是在ThreadLocal类中createmap函数内初始化的。每个线程都有一个这样的threadLocals引用的ThreadLocalMap，以ThreadLocal和ThreadLocal对象声明的变量类型作为参数。这样，我们所使用的ThreadLocal变量的实际数据，通过get函数取值的时候，就是通过取出Thread中threadLocals引用的map，然后从这个map中根据当前threadLocal作为参数，取出数据。

需要注意：

内存泄漏；

2：什么叫线程安全？

线程安全就是每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的。线程安全就是说多线程访问同一代码，不会产生不确定的结果。线程安全问题多是由全局变量和静态变量引起的，当多个线程对共享数据只执行读操作，不执行写操作时，一般是线程安全的；当多个线程都执行写操作时，需要考虑线程同步来解决线程安全问题。

3：什么叫线程同步？

多个线程操作一个资源的情况下，导致资源数据前后不一致。这样就需要协调线程的调度，即线程同步。 解决多个线程使用共通资源的方法是：线程操作资源时独占资源，其他线程不能访问资源。使用锁可以保证在某一代码段上只有一条线程访问共用资源。有两种方式实现线程同步：synchronized和同步锁。

4：什么叫线程间通信？

有时候线程之间需要协作和通信。有两种方式实现线程通信：

1、synchronized 实现内存可见性，满足线程共享变量；

2、wait/notify\notifyAll（synchronized同步方法或同步块中使用） 实现内存可见性，及生产消费模式的相互唤醒机制；

3、同步锁（Lock）的Condition（await\signal\signalAll）；

4、管道，实现数据的共享，满足读写模式