1. 集合

Collection接口:

List接口:(实现类都是有序、可重复，并且都实现了Cloneable和Serializable)

ArrayList:

底层是基于动态数组实现的，能存放null元素，线程不安全，但是可以使 用Collections.synchrionizedList(new ArrayList<>())达到线程安全。他的初始 大小为10，每次扩充增加原来的0.5倍(只取整数部分)。

随机查询单个元素快，插入、删除元素速度慢。每次操作都要复制、移动。

删除元素时不会减少容量，可以使用调用trimToSize()来减少容量。

LinkedList:

底层是基于双向循环链表实现的，可以存放null元素，线程不安全，但是 可以使用Collections.synchrionizedList(new ArrayList<>())达到线程安全。没 有初始化大小，也没有扩容机制，可以一直插入新的与元素。

随机查询单个元素慢，因为要根据指针位移，插入、删除元素快。

Vector:

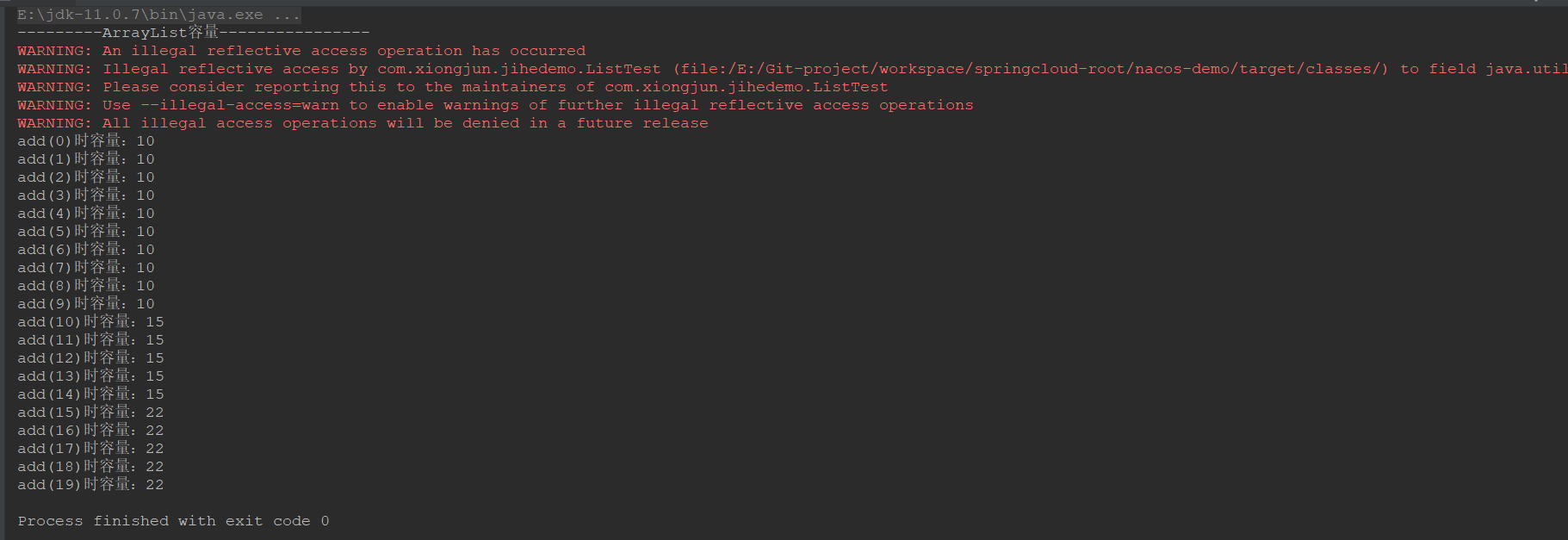
底层也是基于动态数据实现的，能存放null元素，线程安全，初始化容量也是10，每次扩容增加原来的一倍。

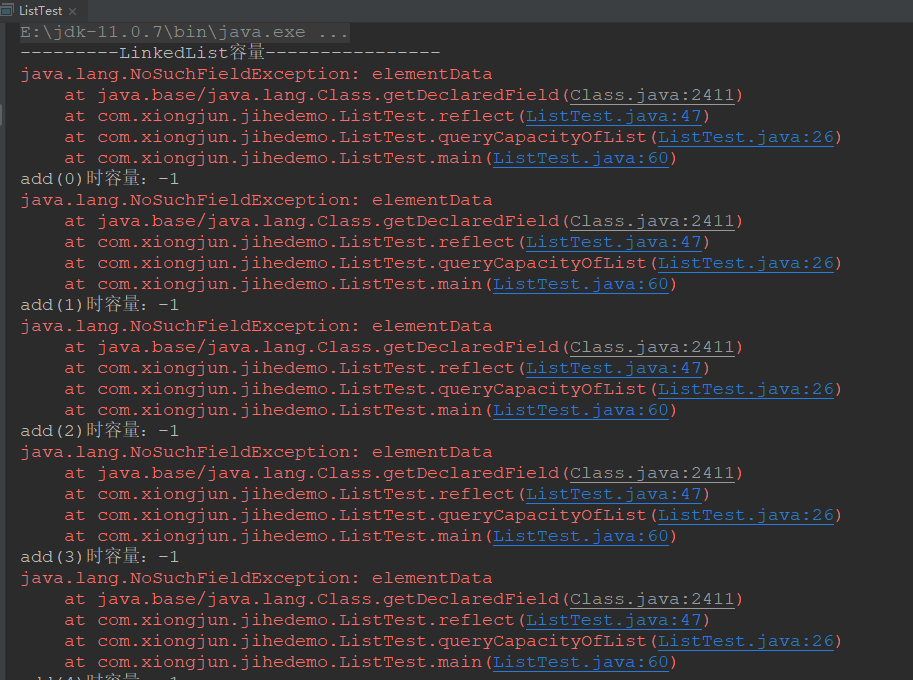
如下图，通过代码来验证扩容的机制：

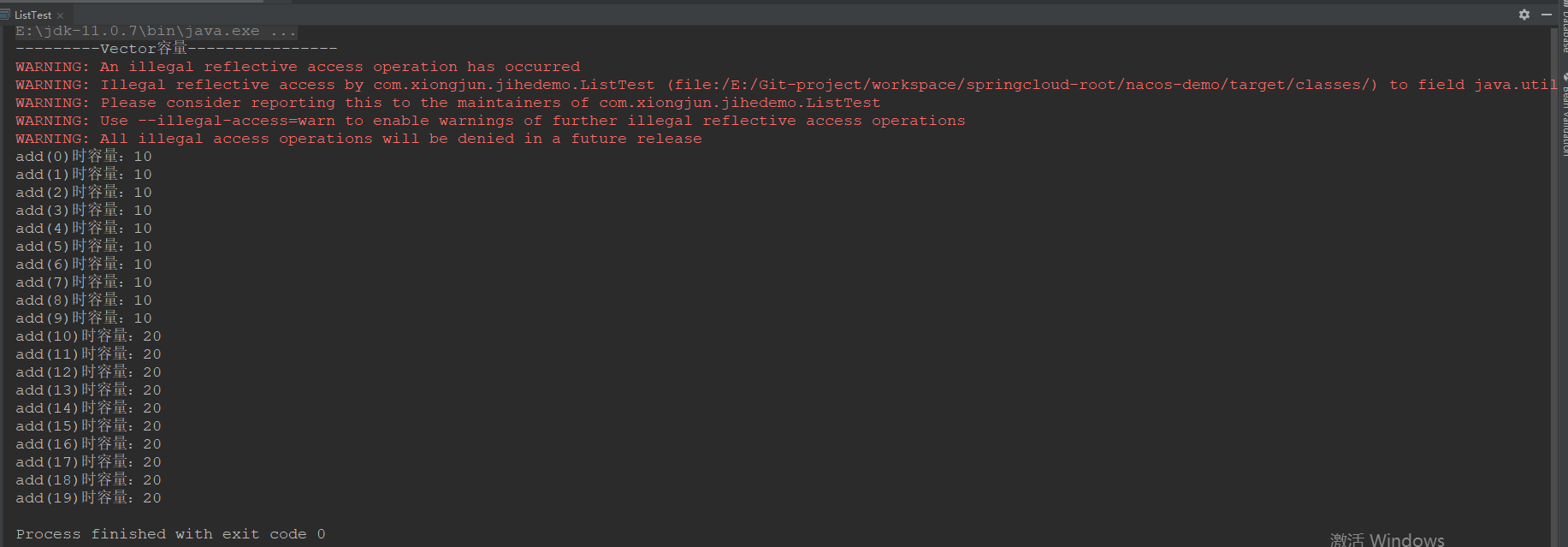
代码：



执行结果：







CopyOnWriteArrayList:

读写分离，线程安全，适合读多写少的场景，相比之下，Collections.synchronizedList写功能效率高，而读功能效率相对低一些。

Set接口(实现类中的元素都是不可重复的，并且都实现了Cloneable和Serializable)):

HashSet:

元素不可重复、无序,但是元素在set中的位置是有该元素的HashCode决定的，其位置其实是固定的。元素可以为null。

底层数据结构是基于HashMap实现的，HashSet的值存放于HashMap的key上。HashSet进行构造时，除了可以使用Collection进行构造外，基本都调用了HashMap的构造函数完成。所以初始化容量为16，加载因子是0.75。线程不安全。

LinkedHashSet:

元素不可重复、有序。元素可以为null。

底层数据结构是链表+hash表，由链表保证元素有序、hash表保证元素唯一。LinkedHashSet使用LinkedHashMap对象来存储它的元素。线程不安全。

当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，因为它是根据链表维护元素的次序的，但是插入时性能稍微逊色于HashSet(百万数据量时才能体现出)

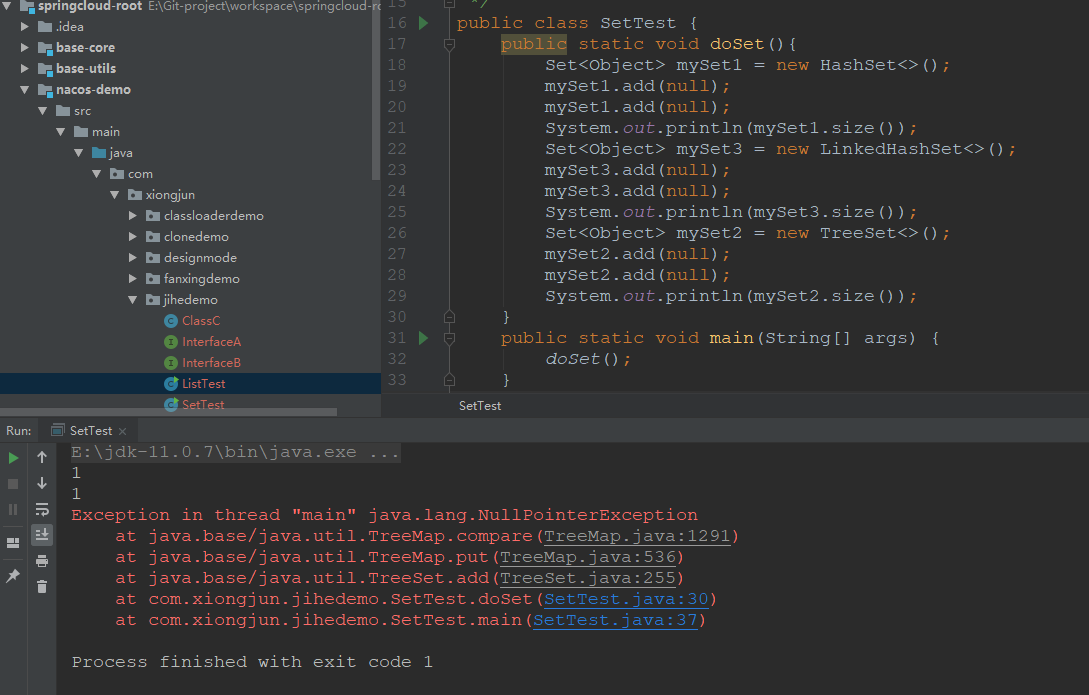
TressSet:

元素不可重复、有序，根据比较的返回值是否是0来决定是否唯一。

底层数据结构是红黑树。

元素不能为null，否则会报空指针异常。它使用了TreeMap来控制数据的有序性。线程不安全。

TreeSet添加元素时，只能添加同类型的元素，否则会报类型转换错误。



TreeSet有以下两种排序方式：

自然排序:实现Comparable接口，重写了compareTo()来进行排序。

自定义排序(比较器排序):实现Comparable接口或者Comparator接口，重写compareTo()方法。两种排序方式对比如下：

TreeSet-自然排序：





执行结果：



TreeSet-自定义排序：







执行结果：



Queue接口:

队列都是先进先出的顺序。阻塞队列与非阻塞队列区别：

当队列是空时，从队列中获取元素的操作会被阻塞，直到其他线程往空的队列插入新的元素；当队列满时，往队列插入新的元素同样会被阻塞，直到队列有位置可以插入。

1. ArrayDeque, （数组双端队列，有界，非阻塞）

内部使用数组进行元素存储，不允许存储null值，可以高效的进行元素查找和尾部插入取出，是用作队列、双端队列、栈的绝佳选择，性能比LinkedList还要好。

1. PriorityQueue, （优先级队列，无界，非阻塞）

一个基于优先级的无界优先级队列。优先级队列的元素按照其自然顺序进行排序，或者根据构造队列时提供的 Comparator 进行排序，具体取决于所使用的构造方法。该队列不允许使用 null 元素也不允许插入不可比较的对象。

1. ConcurrentLinkedQueue, （基于链表的并发队列，无界，线程安全）

是一个适用于高并发场景下的队列，通过无锁的方式，实现了高并发状态下的高性能。ConcurrentLinkedQueue的性能要好于BlockingQueue接口，它是一个基于链接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。该队列不允许null元素。

1. DelayQueue, （延期阻塞队列，无界，实现了BlockingQueue接口）

DelayQueue是一个没有边界BlockingQueue实现，加入其中的元素必须实现Delayed接口。当生产者线程调用put之类的方法加入元素时，会触发Delayed接口中的compareTo方法进行排序，也就是说队列中元素的顺序是按到期时间排序的，而非它们进入队列的顺序。排在队列头部的元素是最早到期的，越往后到期时间越晚。

1. ArrayBlockingQueue, （基于数组的有界，阻塞队列）

ArrayBlockingQueue是一个有边界的阻塞队列，它的内部实现是一个数组。有边界的意思是它的容量是有限的，我们必须在其初始化的时候指定它的容量大小，容量大小一旦指定就不可改变。ArrayBlockingQueue是以先进先出的方式存储数据。

6.LinkedBlockingQueue, （基于链表的FIFO，阻塞队列）

LinkedBlockingQueue阻塞队列大小的配置是可选的，如果我们初始化时指定一个大小，它就是有边界的，如果不指定，它就是无边界的。说是无边界，其实是采用了默认大小为Integer.MAX\_VALUE的容量 。它的内部实现是一个链表。

7.LinkedBlockingDeque, （基于链表的FIFO双端，有界，阻塞队列）

LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列，即可以从队列的两端插入和移除元素。双向队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比于其他阻塞队列，LinkedBlockingDeque多了addFirst、addLast、peekFirst、peekLast等方法，以first结尾的方法，表示插入、获取获移除双端队列的第一个元素。以last结尾的方法，表示插入、获取获移除双端队列的最后一个元素。LinkedBlockingDeque是可选容量的，在初始化时可以设置容量防止其过度膨胀，如果不设置，默认容量大小为Integer.MAX\_VALUE。

8.PriorityBlockingQueue, （带优先级的无界，阻塞队列）

priorityBlockingQueue是一个无界队列，它没有限制，在内存允许的情况下可以无限添加元素；它又是具有优先级的队列，是通过构造函数传入的对象来判断，传入的对象必须实现comparable接口。

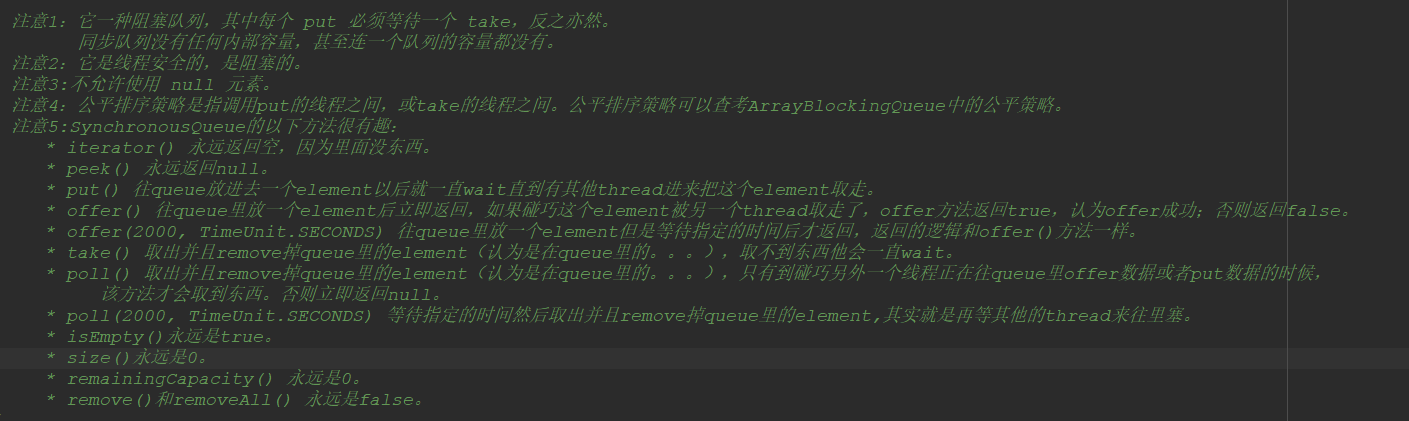
1. SynchronousQueue （无界、并发同步阻塞队列）

该队列内部只能包含一个 元素。

声明一个SynchronousQueue有两种不同的方式(非公平模式和公平模式)：即构造方法有两个new SynchronousQueue();和new SynchronousQueue(boolean fair);其中无参构造函数默认fair=false,即默认为非公平;

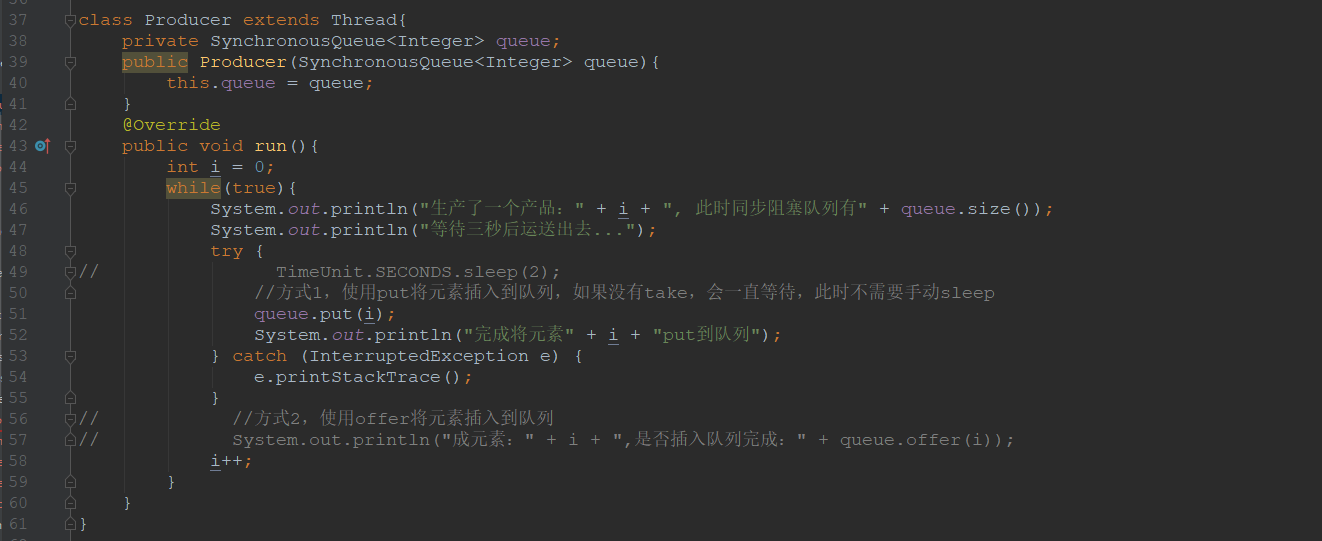
第一种，采用公平模式，SynchronousQueue会采用公平锁，并配合一个LIFO队列来阻塞多余的生产者和消费者，从而体现整体的公平策略。

第二种，非公平模式(默认)，SynchronousQueue采用非公平锁，同时配合一个FIFO队列管理多余的生产者和消费者，如果生产者和消费者的处理速度有差距，则很容易出现有些生产者和消费者的数据得不到处理。

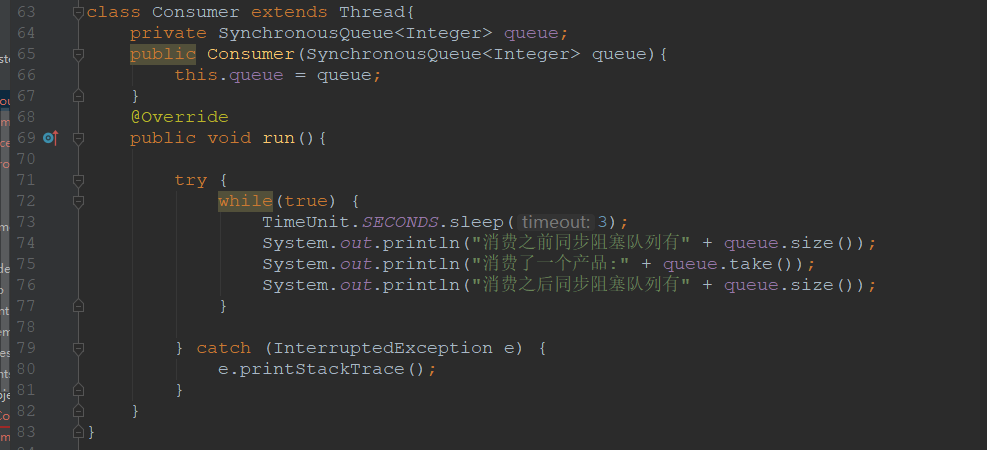


使用示例如下：

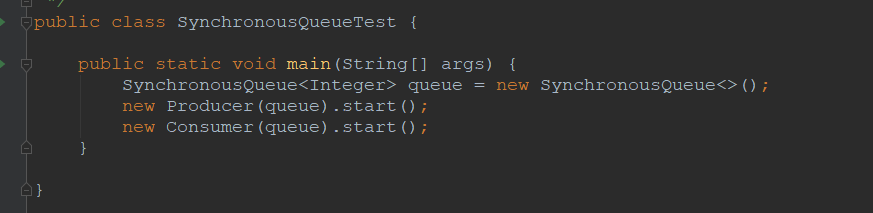
生产者代码：



消费者：



Main方法：



队列中常用方法：



Map接口：

HashMap:

无序，但是是按照一定的顺序(key的hash值顺序)排列;key唯一且可以为null，value可以为null;数据结构是数组+链表,当多个key的hash值相等时，会形成链表存储，所以数组里的每个元素都是一个单向链表;默认的容量大小为16，加载因子为0.75,扩容后容量增加一倍; 非线程安全;jdk1.8之后，数据结构改为数组+链表+红黑树，当数组中的链表元素长度>8且数组容量达到64个时，会由链表转换为红黑树，而当红黑树的节点<6时，会由红黑树转换为链表；非线程安全;

为什么是8？

如果 hashCode 分布良好，也就是 hash 计算的结果离散好的话，那么红黑树这种形式是很少会被用到的，因为各个值都均匀分布，很少出现链表很长的情况。在理想情况下，链表长度符合泊松分布，各个长度的命中概率依次递减，当长度为 8 的时候，概率仅为 0.00000006。这是一个小于千万分之一的概率。

HashMap的读取过程与存储过程类似，先获取key的hash值在table中对应的位置，然后遍历该链获取相同的key的值即可。

TreeMap:

有序，分为默认排序和自定义排序;key唯一，且不能为null必须要实现Comparable接口，从写compareTo()方法，或者实现Comparator接口，重写compare()方法和equals()方法，value可以为null;数据结构是红黑树;非线程安全;没有初始化大小，也没有扩容机制，可以一直插入新的与元素;非线程安全;

LinkedHashMap:

有序，按插入的先后顺序排列，key唯一且可以为null，value可 以为null;因为它继承了HashMap,所以它的默认容量也是16，扩容因子也是0.75，扩容后容量增加一倍；非线程安全；

LinkedHashMap底层就是数组+双向链表，使用双向链表保证了节点的有序性。

Hashtable:

无序;key唯一，key和value都不能为空; 线程安全；数据结构与 HashMap相似，数组+单向链表。默认容量是11(因为除质数求余的分散效果好)，扩容因子是0.75，扩容后的元素的容量是扩容前容量的2倍 + 1；

Hashtable的sychronized是针对整个表的。

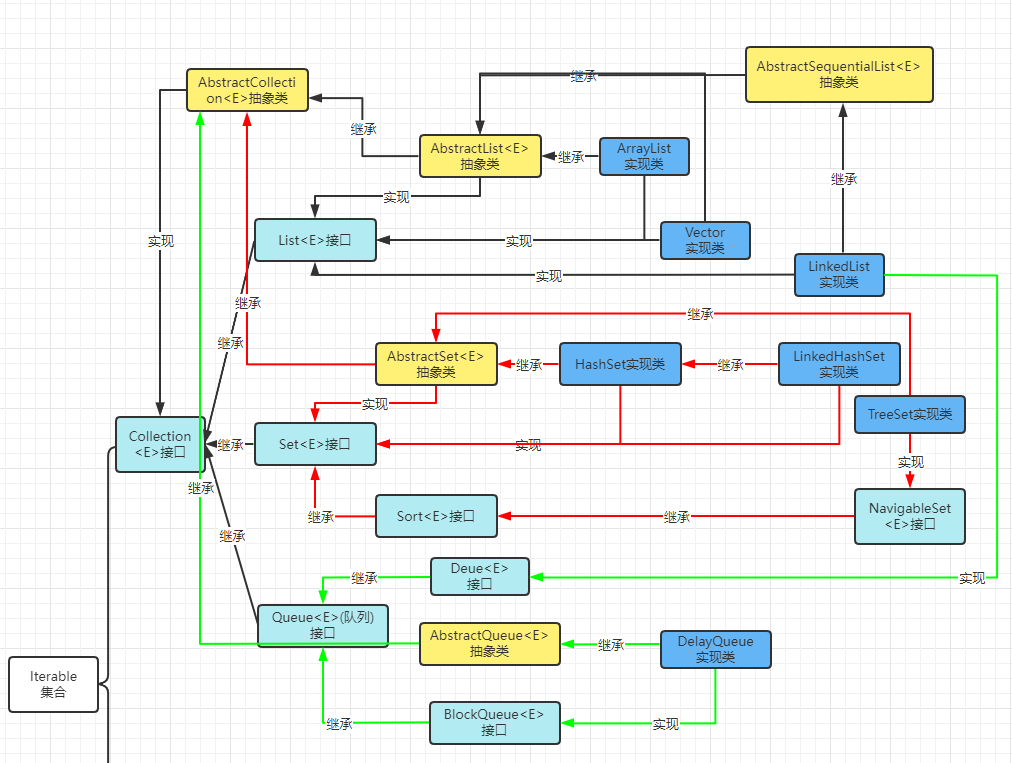
ConcurrentHashMap:

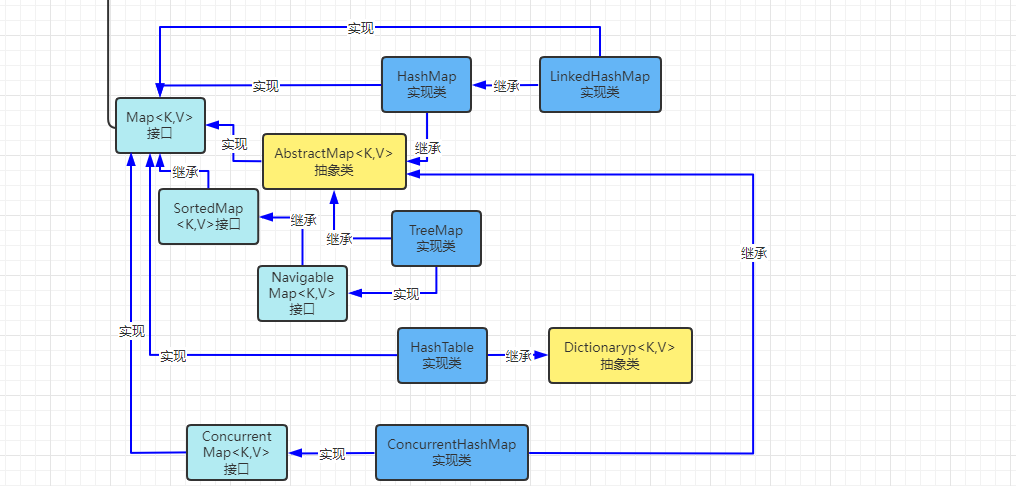
无序;key唯一，key和value都不能为空; 线程安全；底层采 用分段(Segment)数组+链表,Segment本身就相当于一个 HashMap对象，相当于是二级数组。因为是分段的，所以，ConcurrentHashMap允许多个修改操作并发进行，读操作不加锁;因为HashEntry的value变量是volatilede ,也能保证读取到最新的值；默认将hash表分为16个桶，get、put、remove等常用操作只所著对应的桶，所以效率相对于Hashtable高16倍;

Jdk1.8中，ConcurrentHashMap没有用“锁分段”来实现线程安全，而是使用CAS算法和synchronized来确保线程安全，但是底层segment并没有被删除的；底层数据结构是分段数组+链表+红黑树；类似HashMap

参考：<https://www.cnblogs.com/yaoyaoo/p/14381493.html>

<https://baijiahao.baidu.com/s?id=1666189505160307677&wfr=spider&for=pc>





1. java集合的快速失败机制fail-fast与fail-safe

fail-fast是当线程在遍历集合时，改变了集合的结构（新增/删除元素），那么就会抛出

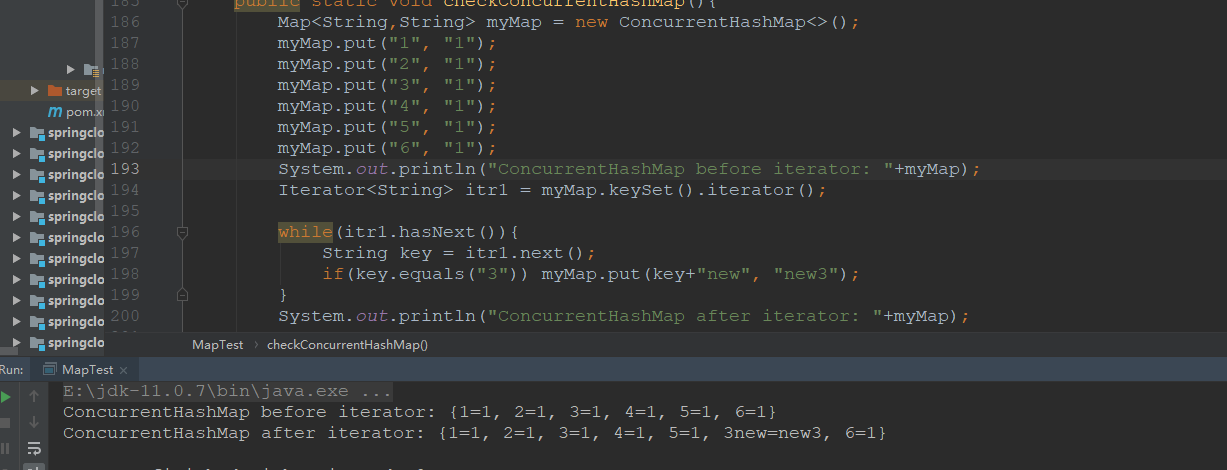
ConCurrentModificationException异常产生的机制。比如下面代码就会抛出该异常：



为什么使用Iterator.remove()就不会抛异常呢？

因为集合类中有一个成员变量modCount,代表集合修改的次数，集合类的add、put、remove方法调用都会使得modCount自增，而迭代器对象有一个属性expectedModCount,他被赋值为迭代遍历开始前modCount的值，在遍历迭代器时，我们使用了集合的add、put、remove方法，会使modCount发生变化，而expectedModCount得值还是之前得值，所以导致expectedModCount与modCount不相等，在调用next()时，第一步就是检查modeCount与expectedModCount是否相对，不相等就抛出ConcurrentModificationException。而iterator.remove()方法不会抛出异常，是因为该方法执行时，会修改modCount的值，同时会把expectedModCount值修改成modCount的值。

fail-safe在遍历集合时不是在集合内容上访问的，而是先复制原有集合内容，然后再拷贝的集合上进行遍历，所以对原集合的修改并不会被迭代器检测到，即原集合的modCount不会被修改，expectedModCount也不会被修改，所以不会抛ConCurrentModificationException异常。java.util.concurrent包下面的所有的类都是安全失败的，比如：ConcurrentHashMap, CopyOnWriteArrayList等。看下面代码：



1. Iterator与ListIterator的区别？

Iterator可以遍历Set与List集合，ListIterator只能遍历List;

Iterator只能单向遍历，而ListIterator能双向遍历;

ListIterator继承了Iterator接口;

1. 为什么 ArrayList 的 elementData 加上 transient 修饰？

Transient的作用是让被修饰的属性不被序列化，ArrayList中重写了writeObject()方法，在每次被序列化时，先调用defaultWriteIbeject()方法序列化ArrayList中的非transient属性，然后遍历elementData,之序列化已存入的元素，这样既加快了序列化的速度，又减小了序列化之后的文件。

1. HashSet如何检查重复？HashSet是如何保证数据不可重复的？

向HashSet中add元素时，不仅要判断hash值，如果hash值相等，再调用equals()方法比较，如果返回true，表示元素重复；

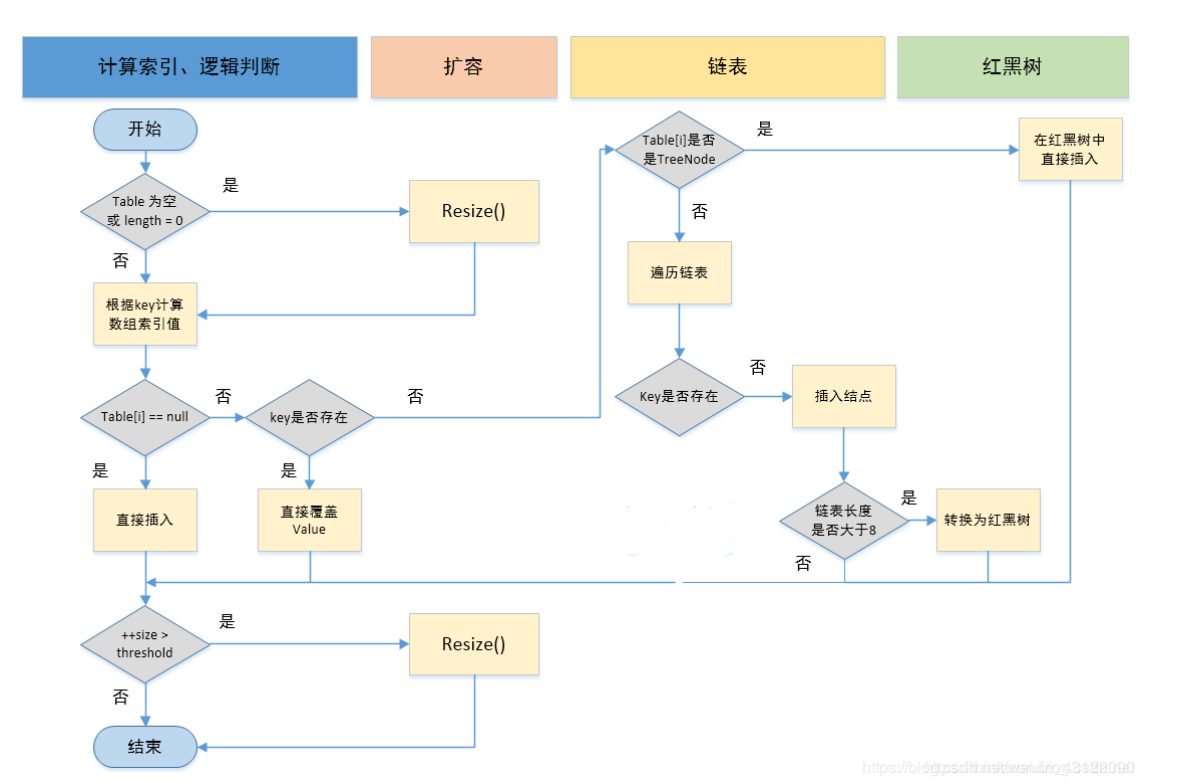
因为HashSet底层是基于HashMap实现的，所以它的key一定唯一，HashSet添加进去的值就是作为HashMap的key，并且在HashMap中如果K相同时，会用新的V覆盖旧的V,所以HashSet中不会重复。

1. HashMap的底层原理

Jdk1.8以前是数组+链表;1.8是数组+链表+红黑树

HashMap中的key会以数组的形式存放key的hash值，但是如果hash值相同，key值不同，会将两个key的元素都放在该hash值下面，以链表的形式存放。Jdk1.8及以后，如果一个hash值下面的key值超过8个，就会把链表自动转为红黑树，如果红黑树的节点少于了6个，就会自动变成链表。

1. HashMap的put具体流程？
2. 先判断hashMap的键值对数组table是否为空或null，如果是就执行resize()进行扩容;
3. 调用hashCode()方法计算key的hash值,并确定hash值所在数组中的位置i，如果该位置tab[i]为null,则直接将元素插入table[i]处;如果tab[i]不为null，则判断新增的key值是否存在，如果存在直接覆盖，如果不存在，(此黄色部分不确定有没有)判断table[i]是否是红黑树TreeNode，如果是，则在红黑树中直接插入，如果不是红黑树TreeNode，遍历链表，插入节点，判断链表长度是否大于8，大于就转为红黑树。
4. 判断数组的长度是否达到了容量的0.75，达到了就扩容，没达到就结束。



1. 为什么重写equals()方法时，必须要重写hashCode()方法？

hashCode()方法是用来获取对象的哈希码值，这个值是用来确定该对象再哈希表中索引所在的位置。

equals()是用来判断两个对象是否相等，如果对象没有重写equals方法，即比较两个对象的地址是否相同，等价于”==”。

先看看代码的例子,有两个Person类，Person2只重写了equals()，Person1重写了equals()和hashCode():





Main():



运行结果：



根据上图的结果想要说明的是:

①如果只重写equals方法，虽然equals的结果是true，但是如果把对象放到HashSet或者其他自动去重的集合中，因为没有重写hashCode，所以此时两个对象的hashcode值其实是不一样的，HashSet会把他们当作两个不同的对象，不会去重。所以，重写equals时必须重写hashCode,是指在需要用到集合(HashSet)去重对象时，必须重写hashCode()。

②如果两个对象equals相等，那么他们的hashCode值一定相等。

③两个对象如果hashCode值一样，他们也不一定相等，需要调用equals来比较。

④hashCode()默认是在堆栈上的对象产生独特的哈希值。如果没有重写hashCode()，

那么两个对象无论如何都不会相等。

1. 为什么HashMap一定要以String、Integer这样的包装类作为key？

因为String、Integer类中都重写了equal()和hashCode()方法，这样减少了hash碰撞（即避免了hash一样，key不一样的情况）。

1. HashMap为什么不直接使用hashCode()处理后的hash值直接作为数组的下标？数组的长度为什么是2的幂次方？

hashCode()方法返回的是int整数类型，其范围是-2^31 ~ (2^31 -1),约有40亿个存储空间，而HashMap的容量范围是在16 ~ 2^30,HashMap是取不到最大值的。

那怎么解决这个问题呢？

HashMap实现了自己的hash()方法，通过两次扰动将自己的哈希值高低位进行异或运算，降低哈希配装概率，也使得数据分布更平均。