深入理解java中的集合

1. 前言

集合是用来存储多个数据的，除了基本类型之外，集合应该是java中最最常用的类型了。java中的集合类型一般都集中在java.util包和java.util.concurrent包中。

其中util包中的集合类是基础的集合类，而concurrent包中的集合类是为并发特别准备的集合类。

集合类的父类有两个，一个是java.util.Collection, 一个是java.util.Map。

先看下Collection的定义：

```
public interface Collection<E> extends Iterable<E> {
}
```

Collection继承自Iterable接口，表示所有的Collection都是可遍历的。并且Collection中可以保存一种数据类型。

再看下Map的定义：

```
public interface Map<K, V> {
}
```

可以看到Map是一个顶级的接口，里面可以保持两种数据类型，分别是key和value。

其中Collection是List,Set和Queue的父类，这样就组成了集合的四大类型：List，Queue，Set和Map，接下来我们将会一一的进行讲解。

2. List

先看下List的定义：

```
public interface List<E> extends Collection<E> {
}
```

List是一个接口，继承自Collection，表示的是一个有序的链表，常用的list有ArrayList,LinkedList等等。

2.1 fail-safe fail-fast知多少

我们在使用集合类的时候，通常会需要去遍历集合中的元素，并在遍历中对其中的元素进行处理。这时候我们就要用到Iterator,经常写程序的朋友应该都知道，在Iterator遍历的过程中，是不能够修改集合数据的，否则就会抛出ConcurrentModificationException。

因为ConcurrentModificationException的存在，就把Iterator分成了两类，Fail-fast和Fail-safe。

2.1.1 Fail-fast Iterator

Fail-fast看名字就知道它的意思是失败的非常快。就是说如果在遍历的过程中修改了集合的结构，则就会立刻报错。

Fail-fast通常在下面两种情况下抛出ConcurrentModificationException：

1. 单线程的环境中

如果在单线程的环境中，iterator创建之后，如果不是通过iterator自身的remove方法，而是通过调用其他的方法修改了集合的结构，则会报错。

2. 多线程的环境中

如果一个线程中创建了iterator,而在另外一个线程中修改了集合的结构，则会报错。

我们先看一个Fail-fast的例子：

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

        users.put(1, "jack");
```

```
        users.put(2, "alice");
        users.put(3, "jone");

        Iterator iterator1 = users.keySet().iterator();

        //not modify key, so no exception
        while (iterator1.hasNext())
        {
            log.info("{}",users.get(iterator1.next()));
            users.put(2, "mark");
        }
```

上面的例子中，我们构建了一个Map，然后遍历该map的key，在遍历过程中，我们修改了map的value。

运行发现，程序完美执行，并没有报任何异常。

这是因为我们遍历的是map的key，只要map的key没有被手动修改，就没有问题。

再看一个例子：

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

        users.put(1, "jack");
        users.put(2, "alice");
        users.put(3, "jone");

        Iterator iterator1 = users.keySet().iterator();

        Iterator iterator2 = users.keySet().iterator();
        //modify key,get exception
        while (iterator2.hasNext())
        {
            log.info("{}",users.get(iterator2.next()));
            users.put(4, "mark");
        }
```

上面的例子中，我们在遍历map的key的同时，对key进行了修改。这种情况下就会报错。

2.1.2 Fail-fast 的原理

为什么修改了集合的结构就会报异常呢?

我们以ArrayList为例，来讲解下Fail-fast 的原理。

在AbstractList中，定义了一个modCount变量：

```
protected transient int modCount = 0;
```

在遍历的过程中都会去调用checkForComodification()方法来对modCount进行检测：

```
        public E next() {
            checkForComodification();
            try {
                int i = cursor;
                E next = get(i);
                lastRet = i;
                cursor = i + 1;
                return next;
            } catch (IndexOutOfBoundsException e) {
                checkForComodification();
                throw new NoSuchElementException();
            }
        }
```

如果检测的结果不是所预期的，就会报错：

```
        final void checkForComodification() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
```

在创建Iterator的时候会复制当前的modCount进行比较，而这个modCount在每次集合修改的时候都会进行变动，最终导致Iterator中的modCount和现有的modCount是不一致的。

```
        public void set(E e) {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();

            try {
                AbstractList.this.set(lastRet, e);
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }
```

注意，Fail-fast并不保证所有的修改都会报错，我们不能够依赖ConcurrentModificationException来判断遍历中集合是否被修改。

2.1.3 Fail-safe Iterator

我们再来讲一下Fail-safe，Fail-safe的意思是在遍历的过程中，如果对集合进行修改是不会报错的。

Concurrent包下面的类型都是Fail-safe的。看一个ConcurrentHashMap的例子：

```
Map<Integer,String> users = new ConcurrentHashMap<>();
```

```
        users.put(1, "jack");
        users.put(2, "alice");
        users.put(3, "jone");

        Iterator iterator1 = users.keySet().iterator();

        //not modify key, so no exception
        while (iterator1.hasNext())
        {
            log.info("{}",users.get(iterator1.next()));
            users.put(2, "mark");
        }

        Iterator iterator2 = users.keySet().iterator();
        //modify key,get exception
        while (iterator2.hasNext())
        {
            log.info("{}",users.get(iterator2.next()));
            users.put(4, "mark");
        }
```

上面的例子完美执行，不会报错。

2.2 Iterator to list的三种方法

集合的变量少不了使用Iterator，从集合Iterator非常简单，直接调用Iterator方法就可以了。

那么如何从Iterator反过来生成List呢？今天教大家三个方法。

2.2.1 使用while

最简单最基本的逻辑就是使用while来遍历这个Iterator，在遍历的过程中将Iterator中的元素添加到新建的List中去。

如下面的代码所示：

```
    @Test
    public void useWhile(){
        List<String> stringList= new ArrayList<>();
        Iterator<String> stringIterator= Arrays.asList("a","b","c").iterator();
        while(stringIterator.hasNext()){
            stringList.add(stringIterator.next());
        }
        log.info("{}",stringList);
    }
```

2.2.2 使用ForEachRemaining

Iterator接口有个default方法：

```
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
```

实际上这方法的底层就是封装了while循环，那么我们可以直接使用这个ForEachRemaining的方法：

```
    @Test
    public void useForEachRemaining(){
        List<String> stringList= new ArrayList<>();
        Iterator<String> stringIterator= Arrays.asList("a","b","c").iterator();
        stringIterator.forEachRemaining(stringList::add);
        log.info("{}",stringList);
    }
```

2.2.3 使用stream

我们知道构建Stream的时候，可以调用StreamSupport的stream方法：

```
public static <T> Stream<T> stream(Spliterator<T> spliterator, boolean
parallel)
```

该方法传入一个spliterator参数。而Iterable接口正好有一个spliterator()的方法：

```
    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
```

那么我们可以将Iterator转换为Iterable，然后传入stream。

仔细研究Iterable接口可以发现，Iterable是一个FunctionalInterface，只需要实现下面的接口就行了：

```
Iterator<T> iterator();
```

利用lambda表达式，我们可以方便的将Iterator转换为Iterable：

```
Iterator<String> stringIterator= Arrays.asList("a","b","c").iterator();
        Iterable<String> stringIterable = () -> stringIterator;
```

最后将其换行成为List：

```
List<String> stringList=
StreamSupport.stream(stringIterable.spliterator(),false).collect(Collectors.toL
ist());
        log.info("{}",stringList);
```

2.3 asList和ArrayList不得不说的故事

提到集合类，ArrayList应该是用到的非常多的类了。这里的ArrayList是java.util.ArrayList，通常我们怎么创建ArrayList呢?

2.3.1 创建ArrayList

看下下面的例子：

```
List<String> names = new ArrayList<>();
```

上面的方法创建了一个ArrayList，如果我们需要向其中添加元素的话，需要再调用add方法。

通常我们会使用一种更加简洁的办法来创建List：

```
    @Test
    public void testAsList(){
        List<String> names = Arrays.asList("alice", "bob", "jack");
        names.add("mark");

    }
```

看下asList方法的定义：

```
    public static <T> List<T> asList(T... a) {
        return new ArrayList<>(a);
    }
```

很好，使用Arrays.asList，我们可以方便的创建ArrayList。

运行下上面的例子，奇怪的事情发生了，上面的例子居然抛出了UnsupportedOperationException异常。

```
java.lang.UnsupportedOperationException
  at java.util.AbstractList.add(AbstractList.java:148)
  at java.util.AbstractList.add(AbstractList.java:108)
  at com.flydean.AsListUsage.testAsList(AsListUsage.java:18)
```

2.3.2 UnsupportedOperationException

先讲一下这个异常，UnsupportedOperationException是一个运行时异常，通常用在某些类中并没有实现接口的某些方法。

为什么上面的ArrayList调用add方法会抛异常呢?

2.3.3 asList

我们再来详细的看一下Arrays.asList方法中返回的ArrayList：

```
private static class ArrayList<E> extends AbstractList<E>
        implements RandomAccess, java.io.Serializable
```

可以看到，Arrays.asList返回的ArrayList是Arrays类中的一个内部类，并不是java.util.ArrayList。

这个类继承自AbstractList，在AbstractList中add方法是这样定义的：

```
    public void add(int index, E element) {
        throw new UnsupportedOperationException();
    }
```

好了，我们的问题得到了解决。

2.3.4 转换

我们使用Arrays.asList得到ArrayList之后，能不能将其转换成为java.util.ArrayList呢？答案是肯定的。

我们看下下面的例子：

```
    @Test
    public void testList(){
        List<String> names = new ArrayList<>(Arrays.asList("alice", "bob",
"jack"));
        names.add("mark");
    }
```

上面的例子可以正常执行。

在java中有很多同样名字的类，我们需要弄清楚他们到底是什么，不要混淆了。

2.4 Copy ArrayList的四种方式

ArrayList是我们经常会用到的集合类，有时候我们需要拷贝一个ArrayList，今天向大家介绍拷贝ArrayList常用的四种方式。

2.4.1 使用构造函数

ArrayList有个构造函数，可以传入一个集合：

```
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        if ((size = elementData.length) != 0) {
            // c.toArray might (incorrectly) not return Object[] (see 6260652)
            if (elementData.getClass() != Object[].class)
                elementData = Arrays.copyOf(elementData, size, Object[].class);
        } else {
            // replace with empty array.
            this.elementData = EMPTY_ELEMENTDATA;
        }
    }
```

上面的代码我们可以看出，底层实际上调用了Arrays.copyOf方法来对数组进行拷贝。这个拷贝调用了系统的native arraycopy方法，注意这里的拷贝是引用拷贝，而不是值的拷贝。这就意味着这如果拷贝之后对象的值发送了变化，源对象也会发生改变。

举个例子：

```
    @Test
    public void withConstructor(){
        List<String> stringList=new ArrayList<>(Arrays.asList("a","b","c"));
        List<String> copyList = new ArrayList<>(stringList);
        copyList.set(0,"e");
        log.info("{}",stringList);
        log.info("{}",copyList);

        List<CustBook> objectList=new ArrayList<>(Arrays.asList(new
CustBook("a"),new CustBook("b"),new CustBook("c")));
        List<CustBook> copyobjectList = new ArrayList<>(objectList);
        copyobjectList.get(0).setName("e");
        log.info("{}",objectList);
        log.info("{}",copyobjectList);
    }
```

运行结果：

```
22:58:39.001 [main] INFO com.flydean.CopyList - [a, b, c]
22:58:39.008 [main] INFO com.flydean.CopyList - [e, b, c]
22:58:39.009 [main] INFO com.flydean.CopyList - [CustBook(name=e),
CustBook(name=b), CustBook(name=c)]
22:58:39.009 [main] INFO com.flydean.CopyList - [CustBook(name=e),
CustBook(name=b), CustBook(name=c)]
```

我们看到对象的改变实际上改变了拷贝的源。而copyList.set(0,"e")实际上创建了一个新的String对象，并把它赋值到copyList的0位置。

2.4.2 使用addAll方法

List有一个addAll方法，我们可以使用这个方法来进行拷贝：

```
    @Test
    public void withAddAll(){

        List<CustBook> objectList=new ArrayList<>(Arrays.asList(new
CustBook("a"),new CustBook("b"),new CustBook("c")));
        List<CustBook> copyobjectList = new ArrayList<>();
        copyobjectList.addAll(objectList);
        copyobjectList.get(0).setName("e");
        log.info("{}",objectList);
        log.info("{}",copyobjectList);
    }
```

同样的拷贝的是对象的引用。

2.4.3 使用Collections.copy

同样的，使用Collections.copy也可以得到相同的效果，看下代码：

```
    @Test
    public void withCopy(){
        List<CustBook> objectList=new ArrayList<>(Arrays.asList(new
CustBook("a"),new CustBook("b"),new CustBook("c")));
        List<CustBook> copyobjectList = new ArrayList<>(Arrays.asList(new
CustBook("d"),new CustBook("e"),new CustBook("f")));
        Collections.copy(copyobjectList, objectList);
        copyobjectList.get(0).setName("e");
        log.info("{}",objectList);
        log.info("{}",copyobjectList);
    }
```

2.4.4 使用stream

我们也可以使用java 8引入的stream来实现：

```
    @Test
    public void withStream(){

        List<CustBook> objectList=new ArrayList<>(Arrays.asList(new
CustBook("a"),new CustBook("b"),new CustBook("c")));
        List<CustBook>
copyobjectList=objectList.stream().collect(Collectors.toList());
        copyobjectList.get(0).setName("e");
        log.info("{}",objectList);
        log.info("{}",copyobjectList);

    }
```

好了，四种方法讲完了，大家要注意四种方法都是引用拷贝，在使用的时候要小心。

3. Map

先看下Map的定义：

```
public interface Map<K, V> {
}
```

Map是一个key-value对的集合，其中key不能够重复，但是value可以重复。常用的Map有TreeMap和hashMap。

3.1 深入理解HashMap和TreeMap的区别

HashMap和TreeMap是Map家族中非常常用的两个类，两个类在使用上和本质上有什么区别呢？本文将从这两个方面进行深入的探讨，希望能揭露其本质。

3.1.1 HashMap和TreeMap本质区别

先看HashMap的定义：

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
```

再看TreeMap的定义：

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
```

从类的定义来看，HashMap和TreeMap都继承自AbstractMap，不同的是HashMap实现的是Map接口，而TreeMap实现的是NavigableMap接口。NavigableMap是SortedMap的一种，实现了对Map中key的排序。

这样两者的第一个区别就出来了，TreeMap是排序的而HashMap不是。

再看看HashMap和TreeMap的构造函数的区别。

```
public HashMap(int initialCapacity, float loadFactor)
```

HashMap除了默认的无参构造函数之外，还可以接受两个参数initialCapacity和loadFactor。

HashMap的底层结构是Node的数组：

```
transient Node<K,V>[] table
```

initialCapacity就是这个table的初始容量。如果大家不传initialCapacity，HashMap提供了一个默认的值：

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

当HashMap中存储的数据过多的时候，table数组就会被装满，这时候就需要扩容，HashMap的扩容是以2的倍数来进行的。而loadFactor就指定了什么时候需要进行扩容操作。默认的loadFactor是0.75。

```
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

再来看几个非常有趣的变量：

```
static final int TREEIFY_THRESHOLD = 8;
static final int UNTREEIFY_THRESHOLD = 6;
static final int MIN_TREEIFY_CAPACITY = 64;
```

上面的三个变量有什么用呢？在java 8之前，HashMap解决hashcode冲突的方法是采用链表的形式，为了提升效率，java 8将其转成了TreeNode。什么时候会发送这个转换呢？

这时候就要看这两个变量TREEIFY_THRESHOLD和UNTREEIFY_THRESHOLD。

有的同学可能发现了，TREEIFY_THRESHOLD为什么比UNTREEIFY_THRESHOLD大2呢？其实这个问题我也不知道，但是你看源代码的话，用到UNTREEIFY_THRESHOLD时候，都用的是<=,而用到TREEIFY_THRESHOLD的时候，都用的是>= TREEIFY_THRESHOLD - 1，所以这两个变量在本质上是一样的。

MIN_TREEIFY_CAPACITY表示的是如果table转换TreeNode的最小容量，只有capacity >= MIN_TREEIFY_CAPACITY的时候才允许TreeNode的转换。

TreeMap和HashMap不同的是，TreeMap的底层是一个Entry：

```
private transient Entry<K,V> root
```

他的实现是一个红黑树，方便用来遍历和搜索。

TreeMap的构造函数可以传入一个Comparator，实现自定义的比较方法。

```
public TreeMap(Comparator<? super K> comparator) {
        this.comparator = comparator;
    }
```

如果不提供自定义的比较方法，则使用的是key的natural order。

3.1.2 排序区别

我们讲完两者的本质之后，现在举例说明，先看下两者对排序的区别：

```
    @Test
    public void withOrder(){
        Map<String, String> books = new HashMap<>();
        books.put("bob", "books");
        books.put("c", "concurrent");
        books.put("a", "a lock");
        log.info("{}",books);
    }
```

```
    @Test
    public void withOrder(){
        Map<String, String> books = new TreeMap<>();
        books.put("bob", "books");
        books.put("c", "concurrent");
        books.put("a", "a lock");
        log.info("{}",books);
    }
```


同样的代码，一个使用了HashMap，一个使用了TreeMap，我们会发现TreeMap输出的结果是排好序的，而HashMap的输出结果是不定的。

3.1.3 Null值的区别

HashMap可以允许一个null key和多个null value。而TreeMap不允许null key，但是可以允许多个null value。

```
    @Test
    public void withNull() {
        Map<String, String> hashmap = new HashMap<>();
        hashmap.put(null, null);
        log.info("{}",hashmap);
    }
```

```
    @Test
    public void withNull() {
        Map<String, String> hashmap = new TreeMap<>();
        hashmap.put(null, null);
        log.info("{}",hashmap);
    }
```

HashMap会报出： NullPointerException。

3.1.4 性能区别

HashMap的底层是Array，所以HashMap在添加，查找，删除等方法上面速度会非常快。而TreeMap的底层是一个Tree结构，所以速度会比较慢。

另外HashMap因为要保存一个Array，所以会造成空间的浪费，而TreeMap只保存要保持的节点，所以占用的空间比较小。

HashMap如果出现hash冲突的话，效率会变差，不过在java 8进行TreeNode转换之后，效率有很大的提升。

TreeMap在添加和删除节点的时候会进行重排序，会对性能有所影响。

3.1.5 共同点

两者都不允许duplicate key,两者都不是线程安全的。

3.2 深入理解HashMap和LinkedHashMap的区别

我们知道HashMap的变量顺序是不可预测的，这意味着便利的输出顺序并不一定和HashMap的插入顺序是一致的。这个特性通常会对我们的工作造成一定的困扰。为了实现这个功能，我们可以使用LinkedHashMap。

3.2.1 LinkedHashMap详解

先看下LinkedHashMap的定义：

```
public class LinkedHashMap<K,V>
    extends HashMap<K,V>
    implements Map<K,V>
```

LinkedHashMap继承自HashMap，所以HashMap的所有功能在LinkedHashMap都可以用。

LinkedHashMap和HashMap的区别就是新创建了一个Entry:

```
    static class Entry<K,V> extends HashMap.Node<K,V> {
        Entry<K,V> before, after;
        Entry(int hash, K key, V value, Node<K,V> next) {
            super(hash, key, value, next);
        }
    }
```

这个Entry继承自HashMap.Node，多了一个before，after来实现Node之间的连接。

通过这个新创建的Entry，就可以保证遍历的顺序和插入的顺序一致。

3.2.2 插入

下面看一个LinkedHashMap插入的例子：

```
    @Test
    public void insertOrder(){
        LinkedHashMap<String, String> map = new LinkedHashMap<>();
        map.put("ddd","desk");
        map.put("aaa","ask");
        map.put("ccc","check");
        map.keySet().forEach(System.out::println);
    }
```

输出结果：

```
ddd
aaa
ccc
```


可以看到输出结果和插入结果是一致的。

3.2.3 访问

除了遍历的顺序，LinkedHashMap还有一个非常有特色的访问顺序。

我们再看一个LinkedHashMap的构造函数：

```
    public LinkedHashMap(int initialCapacity,
                         float loadFactor,
                         boolean accessOrder) {
        super(initialCapacity, loadFactor);
        this.accessOrder = accessOrder;
    }
```

前面的两个参数initialCapacity，loadFactor我们之前已经讲过了，现在看最后一个参数accessOrder。

当accessOrder设置成为true的时候，就开启了 access-order。

access order的意思是，将对象安装最老访问到最新访问的顺序排序。我们看个例子：

```
    @Test
    public void accessOrder(){
        LinkedHashMap<String, String> map = new LinkedHashMap<>(16, .75f,
true);
        map.put("ddd","desk");
        map.put("aaa","ask");
        map.put("ccc","check");
        map.keySet().forEach(System.out::println);
        map.get("aaa");
        map.keySet().forEach(System.out::println);
    }
```

输出结果：

```
ddd
aaa
ccc
ddd
ccc
aaa
```

我们看到，因为访问了一次“aaa“，从而导致遍历的时候排到了最后。

3.2.4 removeEldestEntry

最后我们看一下LinkedHashMap的一个特别的功能removeEldestEntry。这个方法是干什么的呢?

通过重新removeEldestEntry方法，可以让LinkedHashMap保存特定数目的Entry,通常用在LinkedHashMap用作缓存的情况。

removeEldestEntry将会删除最老的Entry，保留最新的。

```
ublic class CustLinkedHashMap<K, V> extends LinkedHashMap<K, V> {

    private static final int MAX_ENTRIES = 10;

    public CustLinkedHashMap(
            int initialCapacity, float loadFactor, boolean accessOrder) {
        super(initialCapacity, loadFactor, accessOrder);
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry eldest) {
        return size() > MAX_ENTRIES;
    }
```

```
}
```

看上面的一个自定义的例子，上面的例子我们创建了一个保留10个Entry节点的LinkedHashMap。

3.2.5 总结

LinkedHashMap继承自HashMap，同时提供了两个非常有用的功能。

3.3 EnumMap和EnumSet

一般来说我们会选择使用HashMap来存储key-value格式的数据，考虑这样的特殊情况，一个HashMap的key都来自于一个Enum类，这样的情况则可以考虑使用本文要讲的EnumMap。

3.3.1 EnumMap

先看一下EnumMap的定义和HashMap定义的比较：

```
public class EnumMap<K extends Enum<K>, V> extends AbstractMap<K, V>
    implements java.io.Serializable, Cloneable
```

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
```

我们可以看到EnumMap几乎和HashMap是一样的，区别在于EnumMap的key是一个Enum。

下面看一个简单的使用的例子：

先定义一个Enum：

```
public enum Types {
    RED, GREEN, BLACK, YELLO
}
```

再看下怎么使用EnumMap：

```
    @Test
    public void useEnumMap(){
        EnumMap<Types, String> activityMap = new EnumMap<>(Types.class);
        activityMap.put(Types.BLACK,"black");
        activityMap.put(Types.GREEN,"green");
        activityMap.put(Types.RED,"red");
    }
```

其他的操作其实和hashMap是类似的，我们这里就不多讲了。

3.3.2 什么时候使用EnumMap

因为在EnumMap中，所有的key的可能值在创建的时候已经知道了，所以使用EnumMap和hashMap相比，可以提升效率。

同时，因为key比较简单，所以EnumMap在实现中，也不需要像HashMap那样考虑一些复杂的情况。

3.3.3 EnumSet

跟EnumMap很类似，EnumSet是一个set，然后set中的元素都是某个Enum类型。

EnumSet是一个抽象类，要创建EnumSet类可以使用EnumSet提供的两个静态方法，noneOf和allOf。

先看一个noneOf：

```
    public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) {
        Enum<?>[] universe = getUniverse(elementType);
        if (universe == null)
            throw new ClassCastException(elementType + " not an enum");

        if (universe.length <= 64)
            return new RegularEnumSet<>(elementType, universe);
        else
            return new JumboEnumSet<>(elementType, universe);
    }
```

noneOf传入一个Enum类，返回一个空的Enum类型的EnumSet。

从上面的代码我们可以看到EnumSet有两个实现，长度大于64的时候使用JumboEnumSet，小有64的时候使用RegularEnumSet。

注意，JumboEnumSet和RegularEnumSet不建议直接使用，他是内部使用的类。

再看一下allOf：

```
public static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType) {
        EnumSet<E> result = noneOf(elementType);
        result.addAll();
        return result;
    }
```

allOf很简单，先调用noneOf创建空的set，然后调用addAll方法将所有的元素添加进去。

3.3.4 总结

EnumMap和EnumSet对特定的Enum对象做了优化，可以在合适的情况下使用。

3.4 SkipList和ConcurrentSkipListMap的实现

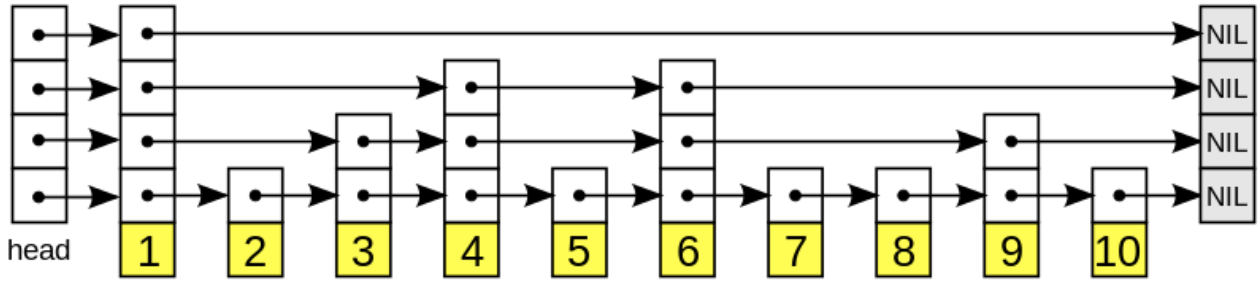
一开始听说SkipList我是一脸懵逼的，啥？还有SkipList？这个是什么玩意。

后面经过我的不断搜索和学习，终于明白了SkipList原来是一种数据结构，而java中的ConcurrentSkipListMap和ConcurrentSkipListSet就是这种结构的实现。

接下来就让我们一步一步的揭开SkipList和ConcurrentSkipListMap的面纱吧。

3.4.1 SkipList

先看下维基百科中SkipList的定义：



SkipList是一种层级结构。最底层的是排序过的最原始的linked list。

往上是一层一层的层级结构，每个底层节点按照一定的概率出现在上一层list中。这个概率叫做p，通常p取1/2或者1/4。

先设定一个函数f，可以随机产生0和1这两个数，并且这两个数出现的几率是一样的，那么这时候的p就是1/2。

对每个节点，我们这样操作：

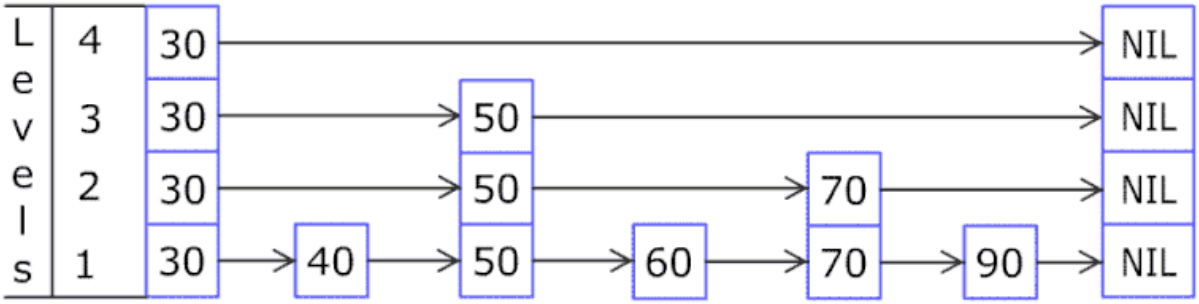
我们运行一次f，当f=1时，我们将该节点插入到上层layer的list中去。当f=0时，不插入。

举个例子，上图中的list中有10个排序过的节点，第一个节点默认每层都有。对于第二个节点，运行f=0，不插入。对于第三个节点，运行f=1,将第三个节点插入layer 1，以此类推，最后得到的layer 1 list中的节点有：1，3，4，6，9。

然后我们再继续往上构建layer。最终得到上图的SkipList。

通过使用SkipList，我们构建了多个List，包含不同的排序过的节点，从而提升List的查找效率。

我们通过下图能有一个更清晰的认识：



每次的查找都是从最顶层开始，因为最顶层的节点数最少，如果要查找的节点在list中的两个节点中间，则向下移一层继续查找，最终找到最底层要插入的位置，插入节点，然后再次调用概率函数f，决定是否向上复制节点。

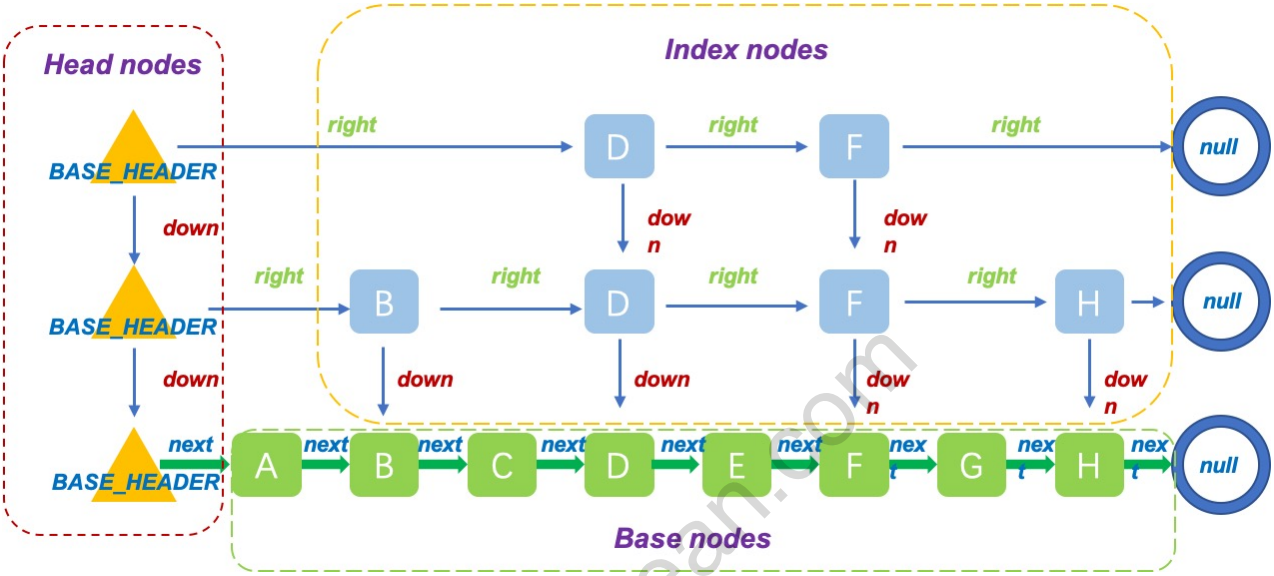
其本质上相当于二分法查找，其查找的时间复杂度是O(logn)。

3.4.2 ConcurrentSkipListMap

ConcurrentSkipListMap是一个并发的SkipList，那么它具有两个特点，SkipList和concurrent。我们分别来讲解。

- SkipList的实现

上面讲解了SkipList的数据结构，接下来看下ConcurrentSkipListMap是怎么实现这个skipList的：



ConcurrentSkipListMap中有三种结构，base nodes,Head nodes和index nodes。

base nodes组成了有序的链表结构，是ConcurrentSkipListMap的最底层实现。

```
static final class Node<K,V> {
    final K key;
    volatile Object value;
    volatile Node<K,V> next;

    /**
     * Creates a new regular node.
     */
    Node(K key, Object value, Node<K,V> next) {
        this.key = key;
        this.value = value;
        this.next = next;
    }
}
```

上面可以看到每个Node都是一个k，v的entry，并且其有一个next指向下一个节点。

index nodes是构建SkipList上层结构的基本节点：

```
static class Index<K,V> {
    final Node<K,V> node;
```

```
        final Index<K,V> down;
        volatile Index<K,V> right;

        /**
         * Creates index node with given values.
         */
        Index(Node<K,V> node, Index<K,V> down, Index<K,V> right) {
            this.node = node;
            this.down = down;
            this.right = right;
        }
    }
```

从上面的构造我们可以看到，Index节点包含了Node节点，除此之外，Index还有两个指针，一个指向同一个layer的下一个节点，一个指向下一层layer的节点。

这样的结构可以方便遍历的实现。

最后看一下HeadIndex，HeadIndex代表的是Head节点：

```
    static final class HeadIndex<K,V> extends Index<K,V> {
        final int level;
        HeadIndex(Node<K,V> node, Index<K,V> down, Index<K,V> right, int level)
{
            super(node, down, right);
            this.level = level;
        }
    }
```

HeadIndex和Index很类似，只不过多了一个level字段，表示所在的层级。

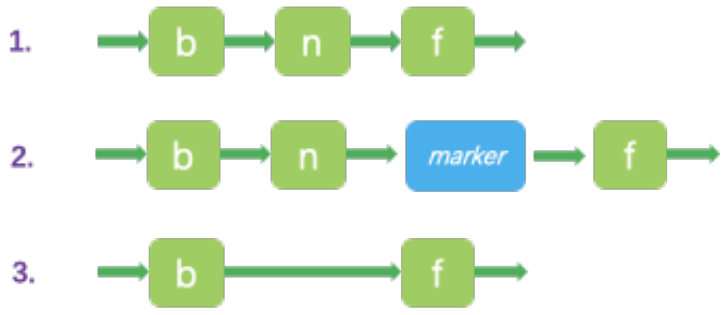
在ConcurrentSkipListMap初始化的时候，会初始化HeadIndex：

```
head = new HeadIndex<K,V>(new Node<K,V>(null, BASE_HEADER, null),null, null,
1);
```

我们可以看到HeadIndex中的Node是key=null，value=BASE_HEADER的虚拟节点。初始的level=1。

- concurrent的实现

接下来，我们再看一下并发是怎么实现的：



基本上并发类都是通过UNSAFE.compareAndSwapObject来实现的，ConcurrentSkipListMap也不例外。

假如我们有三个节点，b-n-f。现在需要删除节点n。

第一步，使用CAS将n的valu的值从non-null设置为null。这个时候，任何外部的操作都会认为这个节点是不存在的。但是那些内部的插入或者删除操作还是会继续修改n的next指针。

第二步，使用CAS将n的next指针指向一个新的marker节点，从这个时候开始，n的next指针将不会指向任何其他的节点。

我们看下marker节点的定义：

```
        Node(Node<K,V> next) {
            this.key = null;
            this.value = this;
            this.next = next;
        }
```

我们可以看到marker节点实际上是一个key为null，value是自己的节点。

第三步，使用CAS将b的next指针指向f。从这一步起，n节点不会再被其他的程序访问，这意味着n可以被垃圾回收了。

我们思考一下为什么要插入一个marker节点，这是因为我们在删除的时候，需要告诉所有的线程，节点n准备被删除了，因为n本来就指向f节点，这个时候需要一个中间节点来表示这个准备删除的状态。

4. Queue

先看下Queue的定义：

```
public interface Queue<E> extends Collection<E> {
}
```

Queue表示的是队列，其特点就是先进先出。常用的Queue有DelayQueue，BlockingQueue等等。

4.1 java中的Queue家族

java中Collection集合有三大家族List，Set和Queue。当然Map也算是一种集合类，但Map并不继承Collection接口。

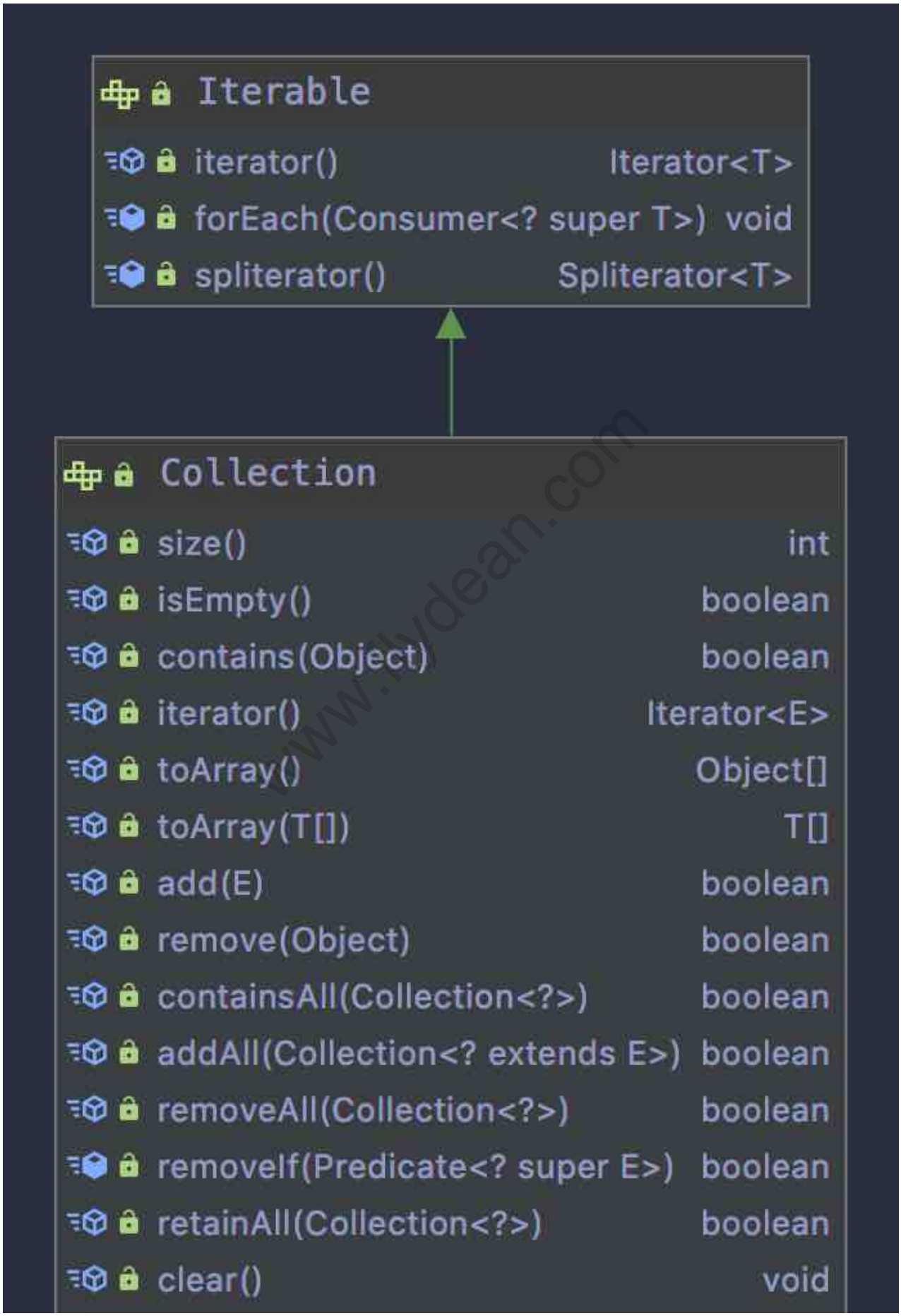
List，Set在我们的工作中会经常使用，通常用来存储结果数据，而Queue由于它的特殊性，通常用在生产者消费者模式中。

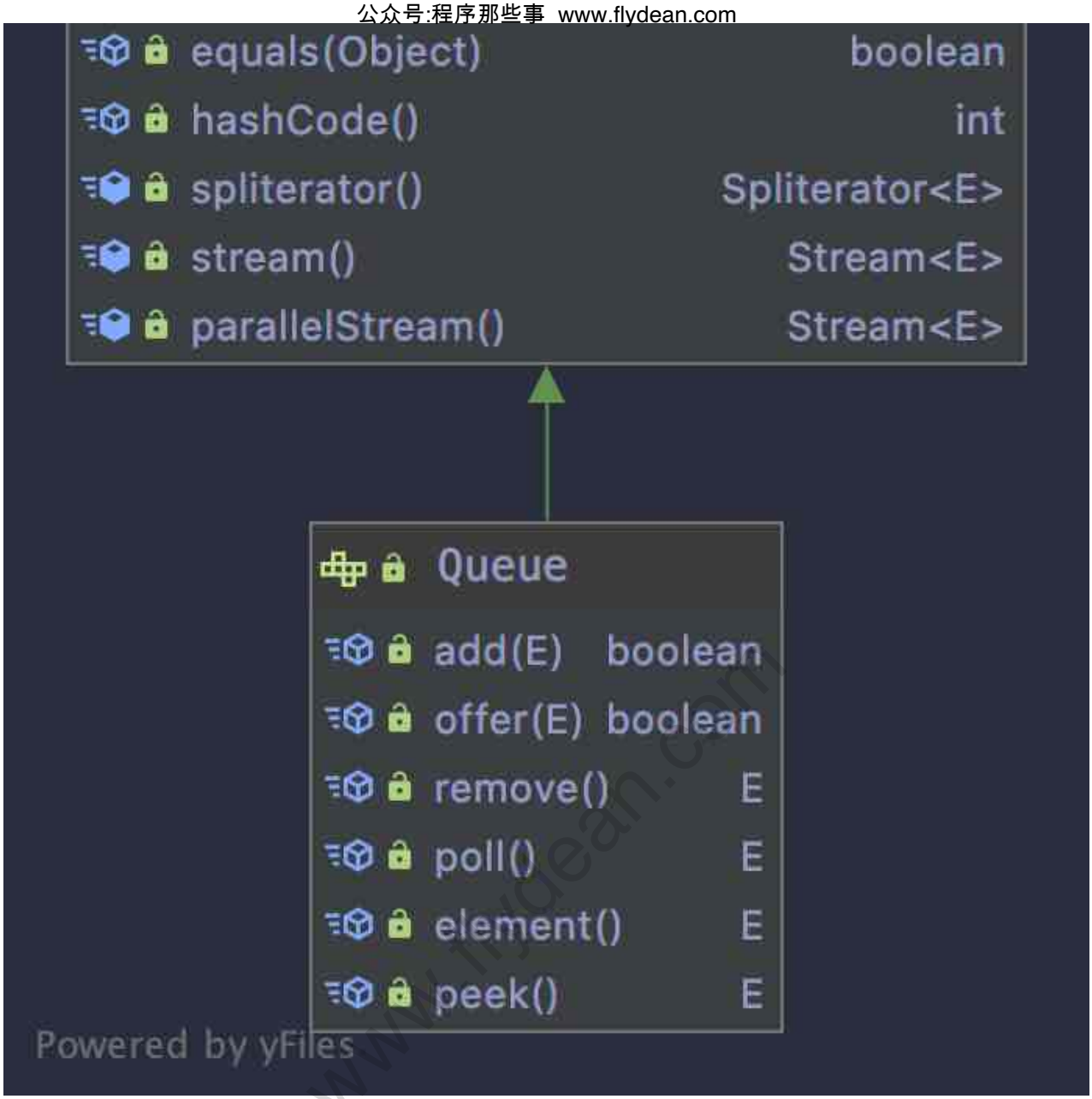
现在很火的消息中间件比如：Rabbit MQ等都是Queue这种数据结构的展开。

今天这篇文章将带大家进入Queue家族。

4.1.1 Queue接口

先看下Queue的继承关系和其中定义的方法：





Queue继承自Collection，Collection继承自Iterable。

Queue有三类主要的方法，我们用个表格来看一下他们的区别：

方法类型	方法名称	方法名称	区别
Insert	add	offer	两个方法都表示向Queue中添加某个元素，不同之处在于添加失败的情况，add只会返回true，如果添加失败，会抛出异常。offer在添加失败的时候会返回false。所以对那些有固定长度的Queue，优先使用offer方法。
Remove	remove	poll	如果Queue是空的情况下，remove会抛出异常，而poll会返回null。
Examine	element	peek	获取Queue头部的元素，但不从Queue中删除。两者的区别还是在于Queue为空的情况下，element会抛出异常，而peek返回null。

> 注意，因为对poll和peek来说null是有特殊含义的，所以一般来说Queue中禁止插入null，但是在实现中还是有一些类允许插入null比如LinkedList。

> 尽管如此，我们在使用中还是要避免插入null元素。

4.1.2 Queue的分类

一般来说Queue可以分为BlockingQueue，Deque和TransferQueue三种。

- BlockingQueue

BlockingQueue是Queue的一种实现，它提供了两种额外的功能：

1. 当当前Queue是空的时候，从BlockingQueue中获取元素的操作会被阻塞。
2. 当当前Queue达到最大容量的时候，插入BlockingQueue的操作会被阻塞。

BlockingQueue的操作可以分为下面四类：

操作类型	Throws exception	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable

第一类是会抛出异常的操作，当遇到插入失败，队列为空的时候抛出异常。

第二类是不会抛出异常的操作。

第三类是会Block的操作。当Queue为空或者达到最大容量的时候。

第四类是time out的操作，在给定的时间里会Block，超时会直接返回。

BlockingQueue是线程安全的Queue,可以在生产者消费者模式的多线程中使用，如下所示：

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while (true) { queue.put(produce()); }
    } catch (InterruptedException ex) { ... handle ...}
  }
  Object produce() { ... }
}

class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while (true) { consume(queue.take()); }
```

```
    } catch (InterruptedException ex) { ... handle ...}
  }
  void consume(Object x) { ... }
}

class Setup {
  void main() {
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

最后，在一个线程中向BlockQueue中插入元素之前的操作happens-before另外一个线程中从BlockQueue中删除或者获取的操作。

- Deque

Deque是Queue的子类，它代表double ended queue，也就是说可以从Queue的头部或者尾部插入和删除元素。

同样的，我们也可以将Deque的方法用下面的表格来表示，Deque的方法可以分为对头部的操作和对尾部的操作：

方法类型	Throws exception	Special value	Throws exception	Special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

和Queue的方法描述基本一致，这里就不多讲了。

当Deque以 FIFO (First-In-First-Out)的方法处理元素的时候，Deque就相当于一个Queue。

当Deque以LIFO (Last-In-First-Out)的方式处理元素的时候，Deque就相当于一个Stack。

- TransferQueue

TransferQueue继承自BlockingQueue，为什么叫Transfer呢？因为TransferQueue提供了一个transfer的方法，生产者可以调用这个transfer方法，从而等待消费者调用take或者poll方法从Queue中拿取数据。

还提供了非阻塞和timeout版本的tryTransfer方法以供使用。

我们举个TransferQueue实现的生产者消费者的问题。

先定义一个生产者：

```
@Slf4j
@Data
@AllArgsConstructor
class Producer implements Runnable {
    private TransferQueue<String> transferQueue;

    private String name;

    private Integer messageCount;

    public static final AtomicInteger messageProduced = new AtomicInteger();

    @Override
    public void run() {
        for (int i = 0; i < messageCount; i++) {
            try {
                boolean added = transferQueue.tryTransfer( "第"+i+"个", 2000,
TimeUnit.MILLISECONDS);
                log.info("transfered {} 是否成功: {}","第"+i+"个",added);
                if(added){
                    messageProduced.incrementAndGet();
                }
            } catch (InterruptedException e) {
                log.error(e.getMessage(),e);
            }
        }
        log.info("total transfered {}",messageProduced.get());
    }
}
```

在生产者的run方法中，我们调用了tryTransfer方法，等待2秒钟，如果没成功则直接返回。

再定义一个消费者：

```
@Slf4j
@Data
@AllArgsConstructor
public class Consumer implements Runnable {

    private TransferQueue<String> transferQueue;

    private String name;

    private int messageCount;

    public static final AtomicInteger messageConsumed = new AtomicInteger();

    @Override
```

```
    public void run() {
        for (int i = 0; i < messageCount; i++) {
            try {
                String element = transferQueue.take();
                log.info("take {}",element );
                messageConsumed.incrementAndGet();
                Thread.sleep(500);
            } catch (InterruptedException e) {
                log.error(e.getMessage(),e);
            }
        }
        log.info("total consumed {}",messageConsumed.get());
    }

}
```

在run方法中，调用了transferQueue.take方法来取消息。

下面先看一下一个生产者，零个消费者的情况：

```
    @Test
    public void testOneProduceZeroConsumer() throws InterruptedException {

        TransferQueue<String> transferQueue = new LinkedTransferQueue<>();
        ExecutorService exService = Executors.newFixedThreadPool(10);
        Producer producer = new Producer(transferQueue, "ProducerOne", 5);

        exService.execute(producer);

        exService.awaitTermination(50000, TimeUnit.MILLISECONDS);
        exService.shutdown();
    }
```

输出结果：

```
[pool-1-thread-1] INFO com.flydean.Producer - transfered 第0个 是否成功: false
[pool-1-thread-1] INFO com.flydean.Producer - transfered 第1个 是否成功: false
[pool-1-thread-1] INFO com.flydean.Producer - transfered 第2个 是否成功: false
[pool-1-thread-1] INFO com.flydean.Producer - transfered 第3个 是否成功: false
[pool-1-thread-1] INFO com.flydean.Producer - transfered 第4个 是否成功: false
[pool-1-thread-1] INFO com.flydean.Producer - total transfered 0
```

可以看到，因为没有消费者，所以消息并没有发送成功。

再看下一个有消费者的情况：

```
    @Test
    public void testOneProduceOneConsumer() throws InterruptedException {
```

```
        TransferQueue<String> transferQueue = new LinkedTransferQueue<>();
        ExecutorService exService = Executors.newFixedThreadPool(10);
        Producer producer = new Producer(transferQueue, "ProducerOne", 2);
        Consumer consumer = new Consumer(transferQueue, "ConsumerOne", 2);

        exService.execute(producer);
        exService.execute(consumer);

        exService.awaitTermination(50000, TimeUnit.MILLISECONDS);
        exService.shutdown();
    }
```

输出结果：

```
[pool-1-thread-2] INFO com.flydean.Consumer - take 第0个
[pool-1-thread-1] INFO com.flydean.Producer - transfered 第0个 是否成功：true
[pool-1-thread-2] INFO com.flydean.Consumer - take 第1个
[pool-1-thread-1] INFO com.flydean.Producer - transfered 第1个 是否成功：true
[pool-1-thread-1] INFO com.flydean.Producer - total transfered 2
[pool-1-thread-2] INFO com.flydean.Consumer - total consumed 2
```

可以看到Producer和Consumer是一个一个来生产和消费的。

4.2 PriorityQueue和PriorityBlockingQueue

Queue一般来说都是FIFO的，当然之前我们也介绍过Deque可以做为栈来使用。今天我们介绍一种PriorityQueue，可以安装对象的自然顺序或者自定义顺序在Queue中进行排序。

4.2.1 PriorityQueue

先看PriorityQueue，这个Queue继承自AbstractQueue，是非线程安全的。

PriorityQueue的容量是unbounded的，也就是说它没有容量大小的限制，所以你可以无限添加元素，如果添加的太多，最后会报OutOfMemoryError异常。

这里教大家一个识别的技能，只要集合类中带有CAPACITY的，其底层实现大部分都是数组，因为只有数组才有capacity，当然也有例外，比如LinkedBlockingDeque。

只要集合类中带有comparator的，那么这个集合一定是个有序集合。

我们看下PriorityQueue：

```
private static final int DEFAULT_INITIAL_CAPACITY = 11;
 private final Comparator<? super E> comparator;
```

定义了初始Capacity和comparator，那么PriorityQueue的底层实现就是Array，并且它是一个有序集合。

有序集合默认情况下是按照natural ordering来排序的，如果你传入了 Comparator,则会按照你指定的方式进行排序，我们看两个排序的例子：

```
@Slf4j
public class PriorityQueueUsage {

    @Test
    public void usePriorityQueue(){
        PriorityQueue<Integer> integerQueue = new PriorityQueue<>();

        integerQueue.add(1);
        integerQueue.add(3);
        integerQueue.add(2);

        int first = integerQueue.poll();
        int second = integerQueue.poll();
        int third = integerQueue.poll();

        log.info("{},{},{}",first,second,third);
    }

    @Test
    public void usePriorityQueueWithComparator(){
        PriorityQueue<Integer> integerQueue = new PriorityQueue<>((a,b)-> b-a);
        integerQueue.add(1);
        integerQueue.add(3);
        integerQueue.add(2);

        int first = integerQueue.poll();
        int second = integerQueue.poll();
        int third = integerQueue.poll();

        log.info("{},{},{}",first,second,third);
    }
}
```

默认情况下会按照升序排列，第二个例子中我们传入了一个逆序的Comparator，则会按照逆序排列。

4.2.2 PriorityBlockingQueue

PriorityBlockingQueue是一个BlockingQueue，所以它是线程安全的。

我们考虑这样一个问题，如果两个对象的natural ordering或者Comparator的顺序是一样的话，两个对象的顺序还是固定的吗?

出现这种情况，默认顺序是不能确定的，但是我们可以这样封装对象，让对象可以在排序顺序一致的情况下，再按照创建顺序先进先出FIFO的二次排序：

```
public class FIFOEntry<E extends Comparable<? super E>>
```

```
        implements Comparable<FIFOEntry<E>> {
    static final AtomicLong seq = new AtomicLong(0);
    final long seqNum;
    final E entry;
    public FIFOEntry(E entry) {
        seqNum = seq.getAndIncrement();
        this.entry = entry;
    }
    public E getEntry() { return entry; }
    public int compareTo(FIFOEntry<E> other) {
        int res = entry.compareTo(other.entry);
        if (res == 0 && other.entry != this.entry)
            res = (seqNum < other.seqNum ? -1 : 1);
        return res;
    }
}
```

上面的例子中，先比较两个Entry的natural ordering，如果一致的话，再按照seqNum进行排序。

4.3 SynchronousQueue详解

SynchronousQueue是BlockingQueue的一种，所以SynchronousQueue是线程安全的。SynchronousQueue和其他的BlockingQueue不同的是SynchronousQueue的capacity是0。即SynchronousQueue不存储任何元素。

也就是说SynchronousQueue的每一次insert操作，必须等待其他线性的remove操作。而每一个remove操作也必须等待其他线程的insert操作。

这种特性可以让我们想起了Exchanger。和Exchanger不同的是，使用SynchronousQueue可以在两个线程中传递同一个对象。一个线程放对象，另外一个线程取对象。

4.3.1 举例说明

我们举一个多线程中传递对象的例子。还是举生产者消费者的例子，在生产者中我们创建一个对象，在消费者中我们取出这个对象。先看一下用CountDownLatch该怎么做：

```
    @Test
    public void useCountdownLatch() throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        AtomicReference<Object> atomicReference= new AtomicReference<>();
        CountDownLatch countDownLatch = new CountDownLatch(1);

        Runnable producer = () -> {
            Object object=new Object();
            atomicReference.set(object);
            log.info("produced {}",object);
            countDownLatch.countDown();
        };
```

```
        Runnable consumer = () -> {
            try {
                countDownLatch.await();
                Object object = atomicReference.get();
                log.info("consumed {}",object);
            } catch (InterruptedException ex) {
                log.error(ex.getMessage(),ex);
            }
        };

        executor.submit(producer);
        executor.submit(consumer);

        executor.awaitTermination(50000, TimeUnit.MILLISECONDS);
        executor.shutdown();
    }
```

上例中，我们使用AtomicReference来存储要传递的对象，并且定义了一个型号量为1的CountDownLatch。

在producer中，我们存储对象，并且countDown。

在consumer中，我们await,然后取出对象。

输出结果：

```
[pool-1-thread-1] INFO com.flydean.SynchronousQueueUsage - produced
java.lang.Object@683d1b4b
[pool-1-thread-2] INFO com.flydean.SynchronousQueueUsage - consumed
java.lang.Object@683d1b4b
```

可以看到传入和输出了同一个对象。

上面的例子我们也可以用SynchronousQueue来改写：

```
    @Test
    public void useSynchronousQueue() throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        SynchronousQueue<Object> synchronousQueue=new SynchronousQueue<>();

        Runnable producer = () -> {
            Object object=new Object();
            try {
                synchronousQueue.put(object);
            } catch (InterruptedException ex) {
                log.error(ex.getMessage(),ex);
            }
            log.info("produced {}",object);
        };
```

```
        Runnable consumer = () -> {
            try {
                Object object = synchronousQueue.take();
                log.info("consumed {}",object);
            } catch (InterruptedException ex) {
                log.error(ex.getMessage(),ex);
            }
        };

        executor.submit(producer);
        executor.submit(consumer);

        executor.awaitTermination(50000, TimeUnit.MILLISECONDS);
        executor.shutdown();
    }
```

上面的例子中，如果我们使用synchronousQueue，则可以不用手动同步，也不需要额外的存储。

如果我们需要在代码中用到这种线程中传递对象的情况，那么使用synchronousQueue吧。

4.4 DelayQueue的使用

今天给大家介绍一下DelayQueue,DelayQueue是BlockingQueue的一种，所以它是线程安全的，DelayQueue的特点就是插入Queue中的数据可以按照自定义的delay时间进行排序。只有delay时间小于0的元素才能够被取出。

4.4.1 DelayQueue

先看一下DelayQueue的定义：

```
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
    implements BlockingQueue<E>
```

从定义可以看到，DelayQueue中存入的对象都必须是Delayed的子类。

Delayed继承自Comparable，并且需要实现一个getDelay的方法。

为什么这样设计呢?

因为DelayQueue的底层存储是一个PriorityQueue，在之前的文章中我们讲过了，PriorityQueue是一个可排序的Queue，其中的元素必须实现Comparable方法。而getDelay方法则用来判断排序后的元素是否可以从Queue中取出。

4.4.2 DelayQueue的应用

DelayQueue一般用于生产者消费者模式，我们下面举一个具体的例子。

首先要使用DelayQueue，必须自定义一个Delayed对象：

```
@Data
```

```
public class DelayedUser implements Delayed {
    private String name;
    private long avaibleTime;

    public DelayedUser(String name, long delayTime){
        this.name=name;
        //avaibleTime = 当前时间+ delayTime
        this.avaibleTime=delayTime + System.currentTimeMillis();

    }

    @Override
    public long getDelay(TimeUnit unit) {
        //判断avaibleTime是否大于当前系统时间，并将结果转换成MILLISECONDS
        long diffTime= avaibleTime- System.currentTimeMillis();
        return unit.convert(diffTime,TimeUnit.MILLISECONDS);
    }

    @Override
    public int compareTo(Delayed o) {
        //compareTo用在DelayedUser的排序
        return (int)(this.avaibleTime - ((DelayedUser) o).getAvaibleTime());
    }
}
```

上面的对象中，我们需要实现getDelay和compareTo方法。

接下来我们创建一个生产者：

```
@Slf4j
@Data
@AllArgsConstructor
class DelayedQueueProducer implements Runnable {
    private DelayQueue<DelayedUser> delayQueue;

    private Integer messageCount;

    private long delayedTime;

    @Override
    public void run() {
        for (int i = 0; i < messageCount; i++) {
            try {
                DelayedUser delayedUser = new DelayedUser(
                        new Random().nextInt(1000)+"", delayedTime);
                log.info("put delayedUser {}",delayedUser);
                delayQueue.put(delayedUser);
                Thread.sleep(500);
            } catch (InterruptedException e) {
```

```
                log.error(e.getMessage(),e);
            }
        }
    }
}
```

在生产者中，我们每隔0.5秒创建一个新的DelayedUser对象，并入Queue。

再创建一个消费者：

```
@Slf4j
@Data
@AllArgsConstructor
public class DelayedQueueConsumer implements Runnable {

    private DelayQueue<DelayedUser> delayQueue;

    private int messageCount;

    @Override
    public void run() {
        for (int i = 0; i < messageCount; i++) {
            try {
                DelayedUser element = delayQueue.take();
                log.info("take {}",element );
            } catch (InterruptedException e) {
                log.error(e.getMessage(),e);
            }
        }
    }
}
```

在消费者中，我们循环从queue中获取对象。

最后看一个调用的例子：

```
    @Test
    public void useDelayedQueue() throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        DelayQueue<DelayedUser> queue = new DelayQueue<>();
        int messageCount = 2;
        long delayTime = 500;
        DelayedQueueConsumer consumer = new DelayedQueueConsumer(
                queue, messageCount);
        DelayedQueueProducer producer = new DelayedQueueProducer(
                queue, messageCount, delayTime);

        // when
```

```
        executor.submit(producer);
        executor.submit(consumer);

        // then
        executor.awaitTermination(5, TimeUnit.SECONDS);
        executor.shutdown();

    }
```

上面的测试例子中，我们定义了两个线程的线程池，生产者产生两条消息，delayTime设置为0.5秒，也就是说0.5秒之后，插入的对象能够被获取到。

线程池在5秒之后会被关闭。

运行看下结果：

```
[pool-1-thread-1] INFO com.flydean.DelayedQueueProducer - put delayedUser
DelayedUser(name=917, avaibleTime=1587623188389)
[pool-1-thread-2] INFO com.flydean.DelayedQueueConsumer - take
DelayedUser(name=917, avaibleTime=1587623188389)
[pool-1-thread-1] INFO com.flydean.DelayedQueueProducer - put delayedUser
DelayedUser(name=487, avaibleTime=1587623188899)
[pool-1-thread-2] INFO com.flydean.DelayedQueueConsumer - take
DelayedUser(name=487, avaibleTime=1587623188899)
```

我们看到消息的put和take是交替进行的，符合我们的预期。

如果我们做下修改，将delayTime修改为50000，那么在线程池关闭之前插入的元素是不会过期的，也就是说消费者是无法获取到结果的。

DelayQueue是一种有奇怪特性的BlockingQueue，可以在需要的时候使用。

5. 其他的要点

5.1 Comparable和Comparator的区别

java.lang.Comparable和java.util.Comparator是两个容易混淆的接口,两者都带有比较的意思，那么两个接口到底有什么区别，分别在什么情况下使用呢?

5.1.1 Comparable

Comparable是java.lang包下面的接口，lang包下面可以看做是java的基础语言接口。

实际上Comparable接口只定义了一个方法：

```
public int compareTo(T o);
```

实现这个接口的类都需要实现compareTo方法，表示两个类之间的比较。

这个比较排序之后的order，按照java的说法叫做natural ordering。这个order用在一些可排序的集合比如：SortedSet，SortedMap等等。

当使用这些可排序的集合添加相应的对象时，就会调用compareTo方法来进行natural ordering的排序。

几乎所有的数字类型对象：Integer, Long，Double等都实现了这个Comparable接口。

5.1.2 Comparator

Comparator是一个FunctionalInterface，需要实现compare方法：

```
int compare(T o1, T o2);
```

Comparator在java.util包中，代表其是一个工具类，用来辅助排序的。

在讲Comparable的时候，我们提到Comparable指定了对象的natural ordering，如果我们在添加到可排序集合类的时候想按照我们自定义的方式进行排序，这个时候就需要使用到Comparator了。

Collections.sort(List,Comparator),Arrays.sort(Object[],Comparator) 等这些辅助的方法类都可以通过传入一个Comparator来自定义排序规则。

在排序过程中，首先会去检查Comparator是否存在，如果不存在则会使用默认的natural ordering。

还有一个区别就是Comparator允许对null参数的比较，而Comparable是不允许的，否则会爬出NullPointerException。

5.1.3 举个例子

最后，我们举一个natural ordering和Comparator的例子：

```
    @Test
    public void useCompare(){
        List<Integer> list1 = Arrays.asList(5, 3, 2, 4, 1);
        Collections.sort(list1);
        log.info("{}",list1);

        List<Integer> list2 = Arrays.asList(5, 3, 2, 4, 1);
        Collections.sort(list2, (a, b) -> b - a);
        log.info("{}",list2);
    }
```

输出结果：

```
[main] INFO com.flydean.CompareUsage - [1, 2, 3, 4, 5]
[main] INFO com.flydean.CompareUsage - [5, 4, 3, 2, 1]
```

默认情况下Integer是按照升序来排的，但是我们可以通过传入一个Comparator来改变这个过程。

5.2 Reference和引用类型

java中有值类型也有引用类型，引用类型一般是针对于java中对象来说的，今天介绍一下java中的引用类型。java为引用类型专门定义了一个类叫做Reference。Reference是跟java垃圾回收机制息息相关的类，通过探讨Reference的实现可以更加深入的理解java的垃圾回收是怎么工作的。

本文先从java中的四种引用类型开始，一步一步揭开Reference的面纱。

java中的四种引用类型分别是：强引用，软引用，弱引用和虚引用。

5.2.1 强引用Strong Reference

java中的引用默认就是强引用，任何一个对象的赋值操作就产生了对这个对象的强引用。

我们看一个例子：

```
public class StrongReferenceUsage {

    @Test
    public void stringReference(){
        Object obj = new Object();
    }
}
```

上面我们new了一个Object对象，并将其赋值给obj，这个obj就是new Object()的强引用。

强引用的特性是只要有强引用存在，被引用的对象就不会被垃圾回收。

5.2.2 软引用Soft Reference

软引用在java中有个专门的SoftReference类型，软引用的意思是只有在内存不足的情况下，被引用的对象才会被回收。

先看下SoftReference的定义：

```
public class SoftReference<T> extends Reference<T>
```

SoftReference继承自Reference。它有两种构造函数：

```
    public SoftReference(T referent)
```

和：

```
    public SoftReference(T referent, ReferenceQueue<? super T> q)
```

第一个参数很好理解，就是软引用的对象，第二个参数叫做ReferenceQueue，是用来存储封装的待回收Reference对象的，ReferenceQueue中的对象是由Reference类中的ReferenceHandler内部类进行处理的。

我们举个SoftReference的例子：

```
    @Test
    public void softReference(){
        Object obj = new Object();
        SoftReference<Object> soft = new SoftReference<>(obj);
        obj = null;
        log.info("{}",soft.get());
        System.gc();
        log.info("{}",soft.get());
    }
```

输出结果：

```
22:50:43.733 [main] INFO com.flydean.SoftReferenceUsage -
java.lang.Object@71bc1ae4
22:50:43.749 [main] INFO com.flydean.SoftReferenceUsage -
java.lang.Object@71bc1ae4
```

可以看到在内存充足的情况下，SoftReference引用的对象是不会被回收的。

5.2.3 弱引用weak Reference

weakReference和softReference很类似，不同的是weekReference引用的对象只要垃圾回收执行，就会被回收，而不管是否内存不足。

同样的WeakReference也有两个构造函数：

```
public WeakReference(T referent);

 public WeakReference(T referent, ReferenceQueue<? super T> q);
```

含义和SoftReference一致，这里就不再重复表述了。

我们看下弱引用的例子：

```
    @Test
    public void weakReference() throws InterruptedException {
        Object obj = new Object();
        WeakReference<Object> weak = new WeakReference<>(obj);
        obj = null;
        log.info("{}",weak.get());
        System.gc();
        log.info("{}",weak.get());
    }
```

输出结果：

```
22:58:02.019 [main] INFO com.flydean.WeakReferenceUsage -
java.lang.Object@71bc1ae4
22:58:02.047 [main] INFO com.flydean.WeakReferenceUsage - null
```

我们看到gc过后，弱引用的对象被回收掉了。

5.2.4 虚引用PhantomReference

PhantomReference的作用是跟踪垃圾回收器收集对象的活动，在GC的过程中，如果发现有PhantomReference，GC则会将引用放到ReferenceQueue中，由程序员自己处理，当程序员调用ReferenceQueue.pull()方法，将引用出ReferenceQueue移除之后，Reference对象会变成Inactive状态，意味着被引用的对象可以被回收了。

和SoftReference和WeakReference不同的是，PhantomReference只有一个构造函数，必须传入ReferenceQueue：

```
public PhantomReference(T referent, ReferenceQueue<? super T> q)
```

看一个PhantomReference的例子：

```
@Slf4j
public class PhantomReferenceUsage {

    @Test
    public void usePhantomReference(){
        ReferenceQueue<Object> rq = new ReferenceQueue<>();
        Object obj = new Object();
        PhantomReference<Object> phantomReference = new PhantomReference<>
(obj,rq);
        obj = null;
        log.info("{}",phantomReference.get());
        System.gc();
        Reference<Object> r = (Reference<Object>)rq.poll();
        log.info("{}",r);
    }
}
```

运行结果：

```
07:06:46.336 [main] INFO com.flydean.PhantomReferenceUsage - null
07:06:46.353 [main] INFO com.flydean.PhantomReferenceUsage -
java.lang.ref.PhantomReference@136432db
```

我们看到get的值是null，而GC过后，poll是有值的。

因为PhantomReference引用的是需要被垃圾回收的对象，所以在类的定义中，get一直都是返回null：

```
    public T get() {
        return null;
    }
```

5.2.5 Reference和ReferenceQueue

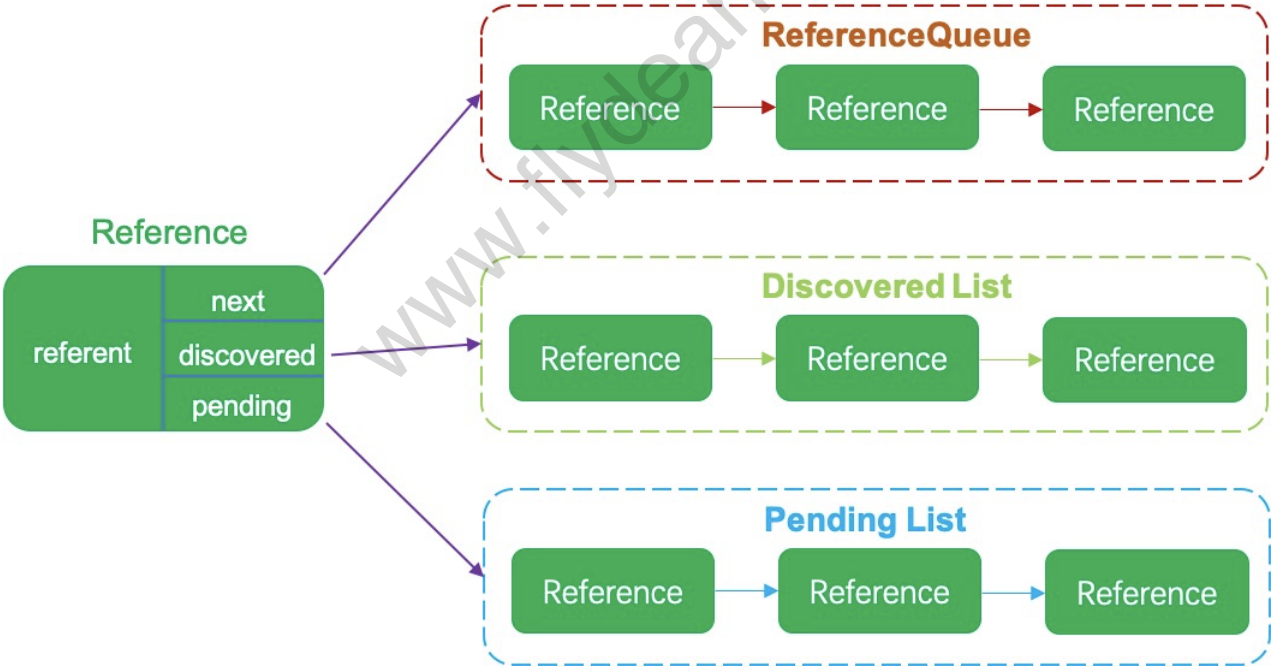
讲完上面的四种引用,接下来我们谈一下他们的父类Reference和ReferenceQueue的作用。

Reference是一个抽象类，每个Reference都有一个指向的对象，在Reference中有5个非常重要的属性：referent，next，discovered，pending，queue。

```
private T referent;         /* Treated specially by GC */
volatile ReferenceQueue<? super T> queue;
Reference next;
transient private Reference<T> discovered;  /* used by VM */
private static Reference<Object> pending = null;
```

每个Reference都可以看成是一个节点，多个Reference通过next，discovered和pending这三个属性进行关联。

先用一张图来对Reference有个整体的概念：



referent就是Reference实际引用的对象。

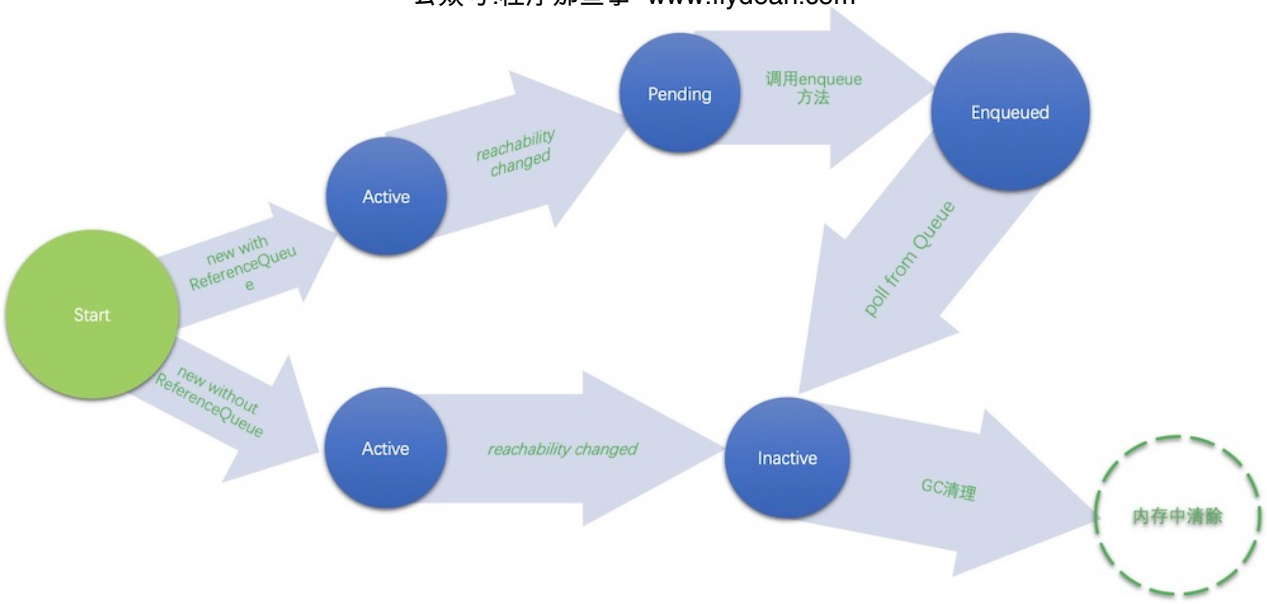
通过next属性，可以构建ReferenceQueue。

通过discovered属性，可以构建Discovered List。

通过pending属性，可以构建Pending List。

- 四大状态

在讲这三个Queue/List之前，我们先讲一下Reference的四个状态：



从上面的图中，我们可以看到一个Reference可以有四个状态。

因为Reference有两个构造函数，一个带ReferenceQueue,一个不带。

```
Reference(T referent) {
    this(referent, null);
}

Reference(T referent, ReferenceQueue<? super T> queue) {
    this.referent = referent;
    this.queue = (queue == null) ? ReferenceQueue.NULL : queue;
}
```

对于带ReferenceQueue的Reference，GC会把要回收对象的Reference放到ReferenceQueue中，后续该Reference需要程序员自己处理（调用poll方法）。

不带ReferenceQueue的Reference，由GC自己处理，待回收的对象其Reference状态会变成Inactive。

创建好了Reference，就进入active状态。

active状态下，如果引用对象的可到达状态发送变化就会转变成Inactive或Pending状态。

Inactive状态很好理解，到达Inactive状态的Reference状态不能被改变，会等待GC回收。

Pending状态代表等待入Queue，Reference内部有个ReferenceHandler，会调用enqueue方法，将Pending对象入到Queue中。

入Queue的对象，其状态就变成了Enqueued。

Enqueued状态的对象，如果调用poll方法从ReferenceQueue拿出，则该Reference的状态就变成了Inactive，等待GC的回收。

这就是Reference的一个完整的生命周期。

- 三个Queue/List

有了上面四个状态的概念，我们接下来讲三个Queue/List：ReferenceQueue，discovered List和pending List。

ReferenceQueue在讲状态的时候已经讲过了，它本质是由Reference中的next连接而成的。用来存储GC待回收的对象。

pending List就是待入ReferenceQueue的list。

discovered List这个有点特别，在Pending状态时候，discovered List就等于pending List。

在Active状态的时候，discovered List实际上维持的是一个引用链。通过这个引用链，我们可以获得引用的链式结构，当某个Reference状态不再是Active状态时，需要将这个Reference从discovered List中删除。

5.2.6 WeakHashMap

最后讲一下WeakHashMap，WeakHashMap跟WeakReference有点类似，在WeakHashMap如果key不再被使用，被赋值为null的时候，该key对应的Entry会自动从WeakHashMap中删除。

我们举个例子：

```
    @Test
    public void useWeakHashMap(){
        WeakHashMap<Object, Object> map = new WeakHashMap<>();
        Object key1= new Object();
        Object value1= new Object();
        Object key2= new Object();
        Object value2= new Object();

        map.put(key1, value1);
        map.put(key2, value2);
        log.info("{}",map);

        key1 = null;
        System.gc();
        log.info("{}",map);

    }
```

输出结果：

```
[main] INFO com.flydean.WeakHashMapUsage -
{java.lang.Object@14899482=java.lang.Object@2437c6dc,
java.lang.Object@11028347=java.lang.Object@1f89ab83}
[main] INFO com.flydean.WeakHashMapUsage -
{java.lang.Object@14899482=java.lang.Object@2437c6dc}
```

可以看到gc过后，WeakHashMap只有一个Entry了。

5.3 类型擦除type erasure

泛型是java从JDK 5开始引入的新特性，泛型的引入可以让我们在代码编译的时候就强制检查传入的类型，从而提升了程序的健壮度。

泛型可以用在类和接口上，在集合类中非常常见。本文将会讲解泛型导致的类型擦除。

5.3.1 举个例子

我们先举一个最简单的例子：

```
@Slf4j
public class TypeErase {

    public static void main(String[] args) {
        ArrayList<String> stringArrayList = new ArrayList<String>();
        stringArrayList.add("a");
        stringArrayList.add("b");
        action(stringArrayList);
    }

    public static void action(ArrayList<Object> al){
        for(Object o: al)
            log.info("{}",o);
    }
}
```

上面的例子中，我们定义了一个ArrayList，其中指定的类型是String。

然后调用了action方法，action方法需要传入一个ArrayList，但是这个list的类型是Object。

乍看之下好像没有问题，因为String是Object的子类，是可以进行转换的。

但是实际上代码编译出错：

```
Error:(18, 16) java: 不兼容的类型: java.util.ArrayList<java.lang.String>无法转换为
java.util.ArrayList<java.lang.Object>
```

5.3.2 原因

上面例子的原因就是类型擦除（type erasure)。java中的泛型是在编译时做检测的。而编译后生成的二进制文件中并不保存类型相关的信息。

上面的例子中，编译之后不管是ArrayList<String> 还是ArrayList<Object> 都会变成ArrayList。其中的类型Object/String对JVM是不可见的。

但是在编译的过程中，编译器发现了两者的类型不同，然后抛出了错误。

5.3.3 解决办法

要解决上面的问题，我们可以使用下面的办法：

```
    public static void actionTwo(ArrayList<?> al){
        for(Object o: al)
            log.info("{}",o);
    }
```

通过使用通配符？，可以匹配任何类型，从而通过编译。

但是要注意这里actionTwo方法中，因为我们不知道传入的类型到底是什么，所以我们不能在actionTwo中添加任何元素。

5.3.4 总结

从上面的例子我们可以看出，ArrayList<String>并不是ArrayList<Object>的子类。如果一定要找出父子关系，那么ArrayList<String>是Collection<String>的子类。

但是Object[] objArray是String[] strArr的父类。因为对Array来说，其具体的类型是已知的。

5.4 深入理解java的泛型

泛型是JDK 5引入的概念，泛型的引入主要是为了保证java中类型的安全性，有点像C++中的模板。

但是Java为了保证向下兼容性，它的泛型全部都是在编译期间实现的。编译器执行类型检查和类型推断，然后生成普通的非泛型的字节码。这种就叫做类型擦除。编译器在编译的过程中执行类型检查来保证类型安全，但是在随后的字节码生成之前将其擦除。

这样就会带来让人困惑的结果。本文将会详细讲解泛型在java中的使用，以避免进入误区。

5.4.1 泛型和协变

有关协变和逆变的详细说明可以参考:

深入理解协变和逆变

这里我再总结一下，协变和逆变只有在类型声明中的类型参数里才有意义，对参数化的方法没有意义，因为该标记影响的是子类继承行为，而方法没有子类。

当然java中没有显示的表示参数类型是协变还是逆变。

协变意思是如果有两个类 A<T> 和 A<C>, 其中C是T的子类，那么我们可以用A来替代A。

逆变就是相反的关系。

Java中数组就是协变的，比如Integer是Number的子类，那么Integer[]也是 Number[]的子类，我们可以在需要 Number[] 的时候传入 Integer[]。

接下来我们考虑泛型的情况，List<Number> 是不是 List<Integer>的父类呢？很遗憾，并不是。

我们得出这样一个结论：泛型不是协变的。

为什么呢？我们举个例子：

```
List<Integer> integerList = new ArrayList<>();
        List<Number> numberList = integerList; // compile error
        numberList.add(new Float(1.111));
```

假如integerList可以赋值给numberList，那么numberList可以添加任意Number类型，比如Float，这样就违背了泛型的初衷，向Integer list中添加了Float。所以上面的操作是不被允许的。

刚刚我们讲到Array是协变的，如果在Array中带入泛型，则会发生编译错误。比如new List<String>[10]是不合法的，但是 new List<?>[10]是可以的。因为在泛型中?表示的是未知类型。

```
List<?>[] list1 = new List<?>[10];

List<String>[] list2 = new List<String>[10]; //compile error
```

5.4.2 泛型在使用中会遇到的问题

因为类型擦除的原因，List<String>和List<Integer>在运行是都会被当做成为List。所以我们在使用泛型时候的一些操作会遇到问题。

假如我们有一个泛型的类，类中有一个方法，方法的参数是泛型，我们想在这个方法中对泛型参数进行一个拷贝操作。

```
public class CustUser<T> {

    public void useT(T param){
        T copy = new T(param);  // compile error
    }
}
```

上面操作会编译失败，因为我们并不知道T是什么，也不知道T到底有没有相应的构造函数。

直接clone T是没有办法了，如果我们想copy一个Set，set中的类型是未定义的该怎么做呢?

```
    public void useTSet(Set<?> set){
        Set<?> copy1 = new HashSet<?>(set);  // compile error
        Set<?> copy2 = new HashSet<>(set);
        Set<?> copy3 = new HashSet<Object>(set);
    }
```

可以看到?是不能直接用于实例化的。但是我们可以用下面的两种方式代替。

再看看Array的使用：

```
    public void useArray(){
         T[] typeArray1= new T[20];  //compile error
        T[] typeArray2=(T[]) new Object[20];
        T[] typeArray3 = (T[]) Array.newInstance(String.class, 20);
    }
```

同样的，T是不能直接用于实例化的，但是我们可以用下面两种方式代替。

5.4.3 类型擦除要注意的事项

因为类型擦除的原因，我们在接口实现中，实现同一个接口的两个不同类型是无意义的：

```
public class someClass implements Comparable<Number>, Comparable<String> { ...
} // no
```

因为在编译过后的字节码看来，两个Comparable是一样的。

同样的，我们使用T来做类型强制转换也是没有意义的：

```
public <T> T cast(T t, Object o) { return (T) o; }
```

因为编译器并不知道这个强制转换是对还是错。

总结

集合是java中一个非常重要的工具类型，希望大家能够熟练掌握。

本文的例子https://github.com/ddean2009/learn-java-collections

> 本文已收录于 www.flydean.com
>
> 最通俗的解读，最深刻的干货，最简洁的教程，众多你不知道的小技巧等你来发现！
>
> 欢迎关注我的公众号:「程序那些事」,懂技术，更懂你！