集合框架

4.集合输出（迭代器输出）-Iterator接口（重点）

集合输出一共有以下形式：

4.1迭代输出Iterator -只能从前向后也可以从后向前（Collection接口提供）

4.2双向迭代接口ListIterator-List接口提供，set不支持

4.3Enumeration枚举输出-Vector类支持

4.4for - each输出（所有子类都满足）

5.fail-fast机制（快速失败机制）

6.fail-safe

5.Map集合（使用版）

5.1Map中的核心方法

5.2 Map接口的使用

5.3 HashMap-类比Hashtable（面试题）

5.4HashMap源码分析

6.1****Map集合使用迭代器（iterator）输出*****

6.2TreeMap子类

Map集合小结：

7栈与队列

7.1stack

7.2Queue接口

8.properties属性文件操作-资源文件

9.Collections工具类

I.将线程不安全集合包装为线程安全集合（不推荐）

II集合排序

III.集合反转

10.Stream 数据流（JDK8新增）：Collection接口

核心方法：取得接口的Stream流

常用方法：

10.1Collection改进

集合框架（数据结构和多线程）

java.util.*；包下（工具包）

考题集中！！（源码理解，整个框架认识）

概念：动态数组 --解决数组长度固定。

-动态数组：当元素个数达到最大值时，动态增加容量。

List 接口：

1.ArrayList 与Vector:区别

（源码理解，多线程同步理解，动态扩容机制，懒加载等）

2.ArrayList 线程不安全的List集合 是否了解JUC包下的线程安全List(CopyOnWriteArrayList)

set：

1.set集合与map集合的关系

2.hashCode.equals方法关系

3.Comparable,Compartor的关系

Map:

1.请对比HashMap,Hashtable关系

2.是否了解ConcurrentHashMap以及实现。

1.类集的产生-JDK1.2

1.1概念

类集是一个动态数组，解决数组定长问题。

数据结构：java版

2.Collection接口-单个对象保存的最顶层父接口

特点：collection接口在每次进行数据处理操作时只能够对单个对象进行处理

里面包含泛型：为了类集服务的（解决向下转型问题的）JDK1.5

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

```
Iterable<E>：迭代器就是为了遍历集合
```

```
Iterator<T> iterator();（取得集合的迭代器）
JDK1.5之前直接写在Collection中
```

2.1Collection中提供的一些核心方法

No.	方法名称	类型	描述
1.	public boolean add(E e);	普通	向集合中添加数据
2.	public boolean addAll(Collection<? extends E> c);	普通	向集合中添加一组数据
3.	public void clear();	普通	清空集合数据
4.	public boolean contains(Object o);	普通	查找数据是否存在，需要使用 equals()方法
5.	public boolean remove(Object o);	普通	删除数据，需要 equals()方法
6.	public int size();	普通	取得集合长度
7.	public Object[] toArray();	普通	将集合变为对象数组返回
8.	public Iterator<E> iterator();	普通	取得 Iterator 接口对象，用于集合输出

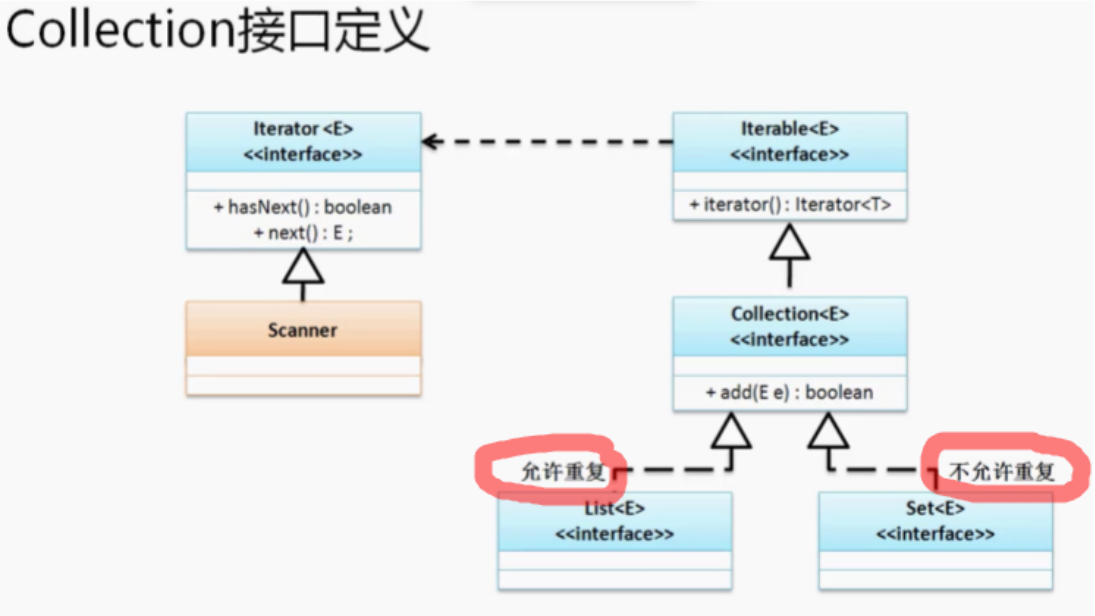
```
1.add(T t):方法向类集中添加元素
```

```
2.Iterator<T> iterator();（取得集合的迭代器）
```

Collection接口之**定义了一个存储数据的标准，但无法区分存储类型。**

因此在实际中我们往往使用两个子接口 **List（允许保存重复数据）**、**Set(不允许数据重复)**。一般不直接使用Collection接口。

2.2Collection的接口定义：



2.3List接口（80%）-允许数据重复

在进行单个集合处理，优先考虑List接口
在List接口中拓展了**两个重要的，独有的方法(List接口独有)**

```
1 public E get(int index):根据索引下标取得数据
```

```
1 public E set（int index)：根据索引下标更新数据，返回修改之前的数据
```

List接口有**三个重要的子类 ArrayList（90%） ,Vector,LinkedList**

List接口**要想保存自定义类的对象**，该类**必须覆写equals方法**来使用我们 **contain()、remove()** 等方法。

2.4面试题一

***********ArrayList,Vector,LinkedList的区别*************

2.4.1ArrayList,Vector区别： （5点法）

- 版本
- 初始化策略
- 扩容机制
- 线程安全，效率
- 在遍历上的差异

1.出现版本：
ArrayList:JDk1.2
Vector：JDk1.0(出现在List,Collection接口之前)

2.调用无参构造的区别(初始化策略)
Vector在无参构造执行后**将对象数组大小初始化为10**
ArrayList采用**懒加载策略**，在构造方法阶段并不初始化对象数组。
在**第一次添加元素时才初始化对象数组大小为10（初始化策略）**

- [x] ~~源码刨析这句话~~

源码刨析

Vector（初始化策略）：

```
public Vector(int initialCapacity, int capacityIncrement) {
 super();
 if (initialCapacity < 0)
 throw new IllegalArgumentException("Illegal Capacity: "+
 initialCapacity);
 this.elementData = new Object[initialCapacity]; //初始化数组
 this.capacityIncrement = capacityIncrement; //增量
}

/**
 * Constructs an empty vector with the specified initial capacity and
 * with its capacity increment equal to zero.
 *
 * @param initialCapacity the initial capacity of the vector
 * @throws IllegalArgumentException if the specified initial capacity
 * is negative
 */
public Vector(int initialCapacity) {
 this(initialCapacity, 0); //initialCapacity=10 Vector容量
}
```

```
21
22 /**
23  * Constructs an empty vector so that its internal data array
24  * has size {@code 10} and its standard capacity increment is
25  * zero.
26  */
27 public Vector() {
28  this(10); //调用有参构造
29 }
```

ArrayList(初始化策略)：

```
1 private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
```

```
1 public ArrayList() { //默认无参构造时懒加载（用时增加）
2  this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
3 }
```

3.扩容策略

ArrayList 扩容时，新数组大小变为原来数组的1.5倍。

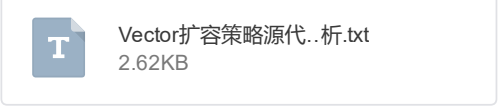
> **假如要设置的值超出1.5倍旧值时 将容量设置为需要的值**

ArrayList 源码详细分析：

Vector:扩容时，新数组大小变为原来数组的2倍。（无参构造）-可以自定义增容策略

> **假如要设置的值超出1.5倍旧值时 将容量设置为需要的值**
> **capacityIncrement:增容策略**
> **如不设置增容策略则是增加两倍**

Vector 源码详细分析：



对比

ArrayList:

```
1 int newCapacity = oldCapacity + (oldCapacity >> 1);1.5倍
```

```
假如要设置的值超出1.5倍旧值时 将容量设置为需要的值
```

Vector

```
int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
 capacityIncrement : oldCapacity);
 capacityIncrement:增容策略
 如不设置增容策略则是增加两倍
```

4.线程安全问题:

ArrayList采用异步处理，线程不安全，效率较高;

```
public boolean add(E e) {
 ensureCapacityInternal(size + 1); // Increments modCount!!
 elementData[size++] = e;
 return true;
}
```

Vector采用在方法上加锁，线程安全，效率较低;

```
public synchronized boolean add(E e) {
 modCount++;
 ensureCapacityHelper(elementCount + 1);
 elementData[elementCount++] = e;
 return true;
}
```

(即便要使用线程安全的List,也不用Vector)?

5.在遍历上的区别:

Vector可以使用较老的迭代器的Enumration,ArrayList不支持

输出形式：ArrayList支持Iterator、ListIterator、foreach；Vector支持Iterator、ListIterator、foreach、Enumeration

- [x] ~~了解Enumeration遍历方式~~

Vector -Enumeration遍历源码刨析:

```
public Enumeration<E> elements() {
 return new Enumeration<E>() { //接口Enumeration
 int count = 0; //计数器
```

```
4
5  public boolean hasMoreElements() {//查看是否还有容量
6  return count < elementCount; //只要count小于elementCount
7  }
8
9  public E nextElement() {//返回下一位元素
10  synchronized (Vector.this) {
11  if (count < elementCount) { //如果count小于内置数组下标则返回下一位元素
12  return elementData(count++);
13  }
14  }
15  throw new NoSuchElementException("Vector Enumeration");
16  }
17  }; //匿名内部类
18 }
```

- [ ] 并没有用到Enum枚举，返回Enumeration相当于简易版iterator?

详解重新写一篇博客：目前参考： **http://www.cnblogs.com/zhaoyan001/p/6077492.html**

https://blog.csdn.net/u012461986/article/details/62419207

iterator和Enumeration比较.note

2.4.2ArrayList,Vector共同点：

1.ArrayList,Vector的都实现了List接口

2..底层都使用数组实现

2.4.3LinkedList子类

跟ArrayList没有什么区别只是在底层实现上有所不同，查找多用ArrayList，改动多用LinkedList

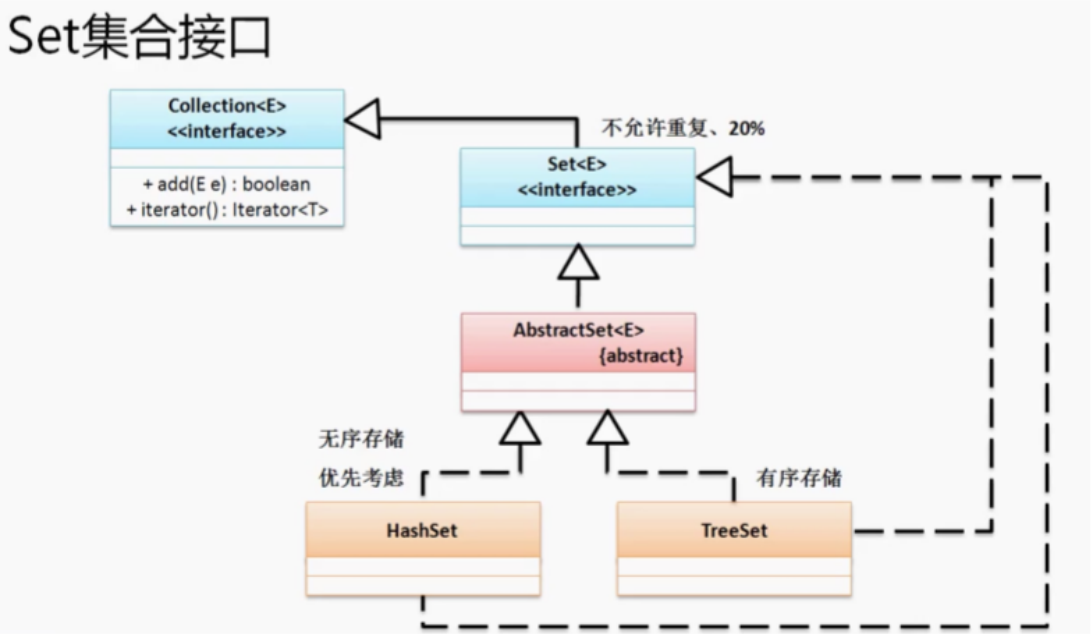
> **1. 观察ArrayList源码，可以发现ArrayList里面存放的是一个数组，如果实例化此类对象时传入了数组大 小，则里面保存的数组就会开辟一个定长的数组，但是后面再进行数据保存的时候发现数组个数不够了 会进行数组动态扩充。 所以在实际开发之中，使用ArrayList最好的做法就是设置初始化大小。**
> **2. LinkedList：是一个纯粹的链表实现，与之前编写的链表程序的实现基本一样（人家性能高）**

2.4.4ArrayList、LinkedList区别：

LinkedList底层采用双向链表实现，ArrayList底层是采用数组实现。

3.Set接口

set接口：不允许数据重复（set接口就是value值相同得Map集合，key值不同）-由于key值是唯一的
set接口：没有在Collection基础上扩充其他方法



3.1set接口常用子类

3.1.1HashSet:（无序存储）-本质上HashMap

- [x] ~~无序在哪里（HashMap）~~

> **按照桶的顺序进行打印（哈希策略）**
> *本质上是一个**HashMap***
> *不允许有重复的元素（底层实际上是使用**HashMap**的**key**值存储数据，*
> ***value**值存了一个无用的值**PRESENT** ）*
> ***Object** 实例对象**PRESENT***
> *且得出的值是按照**hash**算法的出的**hashcode**按照**hashcode**存储的数据（无序性）*

> **1.底层使用哈希表+红黑树**
> **2.允许存放null,无序存储**

3.1.2TreeSet: （有序存储）Comparable Compartor - 本质上TreeMap

***TreeSet**不重复性跟**HashSet**类似，底层是由**Map**实现的（**TreeMap**）*

***TreeSet**:只使用了**K**值，**value**存放了一个**value**值存了一个无用的值**PRESENT** ）*

Object** 实例对象**PRESENT

*但**TreeSet**是有序的集合，使用的是升序排列的模式完成的。*

判断元素是否重复，所以Treeset 或者Treemap要使用（add()）则必须先要调用比较方法。

因为它不允许有重复的数据。（所以添加时需要，进行比较操作，所以**自定义类要想保存到TreeSet中要么实现comparable（内部比较器）并且 覆写compareTo()方法 ,.**
要么使用一个**其他的类当最自定义类的外部比较器实现Comparator(外部排序接口) 并且覆写compare()方法**。

> **1.底层使用红黑树**
> **2.不允许出现空箱，有序存储**
> **3.自定义类要想保存到TreeSet中要么实现comparable,**
> **要么向TreeSet传入比较器（Compartor接口）**

包装类或者引用类型本身就已经继承了Comparable接口，无须再继承。

```
public final class String
 implements java.io.Serializable, Comparable<String>, CharSequence {
```

Comparable 接口和 Compartor接口区别：
在java中，要想实现**自定义类的比较**，提供以下两个接口

3.1.3 java.lang.Comparable接口（内部比较器）--排序接口：

1.若一个类实现comparable接口就意味着**该类支持排序。**
并且**存放该类的(Collection)**或者**对象数组**，可以直接通过**Collection.sort()或Arrays.sort进行排序。**

2.实现了Comparable接口的类可以直接存放在 TreeSet或 TreeMap中。
覆写compareto()方法

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

3种返回值：
返回正数：当前对象大于目标对象

返回0：当前对象等于目标对象

返回负数：当前对象小于目标对象

3.1.4Comparator(外部排序接口)
若要控制某个自定义类的顺序，而该类本身不支持排序（类本身没有实现Comparable接口）。我们可以建立一个该类的“比较器”来进行排序。比较器实现Comparator接口即可。
“比较器”:实现了Comparator接口的类作为比较器，通过该比较器来进行类的排序。

覆写compare（)方法

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

返回值与compareTo返回值完全一样

返回正数：o1>o2
返回0： o1=o2
返回负数：o1<o2

总结：

> 实现了Comparable接口进行第三方排序-策略模式，此方法更加灵活。可以轻松改变策略进行第三方的排序算法。

3.1.4 Comparable接口与Comparator接口的关系：

1.Comparable是一个排序接口，若一个类实现了Comparator接口，意味着该类支持排序，是一个内部类比较器（自己去和别人比）

2.Comparator接口是比较器接口，自定义的类的本身不支持排序，需要有若干个第三方比较器（实现了Compartor接口的类）进行类的排序，是一个外部比较器（**策略模式**）。

3.2重复元的比较

3.2.1重复元素的比较（TreeSet）

TreeSet与TreeMap依靠Compartor或Comparable接口来区分重复元素。

自定义类要想保存在TreeSet或TreeMap中：
I.要么该类直接实现Comparable接口，覆写Compareto方法
II.要么实现一个比较器Comparator，覆写Compare方法，并将其传入TreeSet或TreeMap来进行外部比较。

3.2.2重复元素比较（HashSet）-覆写equals方法原则

而 HashSet与HashMap**并不依赖比较接口**，此时要想**区分自定义元素是否重复**，**需要同时覆写equals与hashCode方法。（Object两部走策略）**

要覆写equals（）方法来判定两个元素内容是否相等。

覆写equals方法原则：
1.自反性：对于任何非空对象引用值x,x.equals(x)都返回true
2.对称性：对于任何非空的x,y,当且仅当x.equals(y)返回true,y.equals(x)也返回true;
3.传递性：对于任何非空的x,y,z，如果x.equals(y)返回true,y.equals(z)返回true,一定有x.equals（z）返回true

4.一致性：对于任何非空的x,y,若x与y中属性没有改变，则多次调用x.equals（y）始终返回true或false。
5.非空性：对于任何非空引用x,x.equals(null)一定返回false

> 先**调用hashCode**计算对象hash码决定存放的数据桶（**保证两个相同数据放在同一个桶里）**
> 而后**使用equals来比较元素是否相等。**
> 若相等，则不再放置元素；
> 若equals返回false,则再相同桶之后，使用链表将若干元素链起来**（拉链法）**。
>
> Object类提供的**hashCode方法**默认使用**对象的地址进行hash。**

结合源码总结

> 若两个对象**equals方法返回true他们得到hashcode必然要保证相同**，但是两**个对象的hashCode相等，equals不一定相等。**
> 当且**仅当equals与hashCode方法均返回true**，才认为这两个**对象真正相等。**

哈希表：
为何要分桶来存放元素。哈希的优势便于查找，（相比链表数组结合他俩的优先）

3.3使用原则

个人建议：
1. 保存**自定义对象**的时候使用List接口；
2. 保存**系统类信息**的时候使用Set接口**(避免重复)。**

4.集合输出（迭代器输出）-Iterator接口（重点）

在之前进行集合输出的时候利用了toString（）,或者利用了List接口种的get（）方法。这些都不是几个的标准输出。从标准输出来讲，集合输出一共有四种手段：
Iterator,ListIterator,Enumeration,foreach循环。

迭代器为了遍历集合而生
Iterator接口两个核心方法：

```
boolean hasNext(); /判断是否还有下一个元素
```

```
E next(); 取得下一个元素
```

```
public default void remove(); 删除元素此方法从JDK1.8开始变为default完整方法。
```

集合输出一共有以下形式：

4.1迭代输出Iterator -只能从前向后也可以从后向前（Collection接口提供）

调用Collection集合子类的Iterator方法取得内置的迭代器，使用以下输出格式

```
Iterator<String> iterator = list.iterator();//实例化Iterator对象

while(iterator.hasNext()){
 System.out.println(iterator.next());
}
```

4.2双向迭代接口ListIterator-List接口提供，set不支持

除了hasNext与next方法外还有如下方法

```
public boolean hashPrevious():判断是否有上一个元素
```

```
public E pervious:取得上一个元素
```

取得ListIterator对象 List接口提供支持

```
public ListIterator listIterator();
```

```


while(listIterator.hasPreviour()){
 System.out.println(listIterator.pervious);
}
```

特点：

要想使用从后向前遍历，**首先至少要从前向后遍历一遍才能使用从后向前遍历。**

- [ ] **为什么必须要从前向后遍历后才能从后向前遍历原因：**

```
public ListIterator<E> listIterator() {
 return new ListItr(0); //生成双向迭代接口的实例化，构造函数传参为0
}

private class ListItr extends Itr implements ListIterator<E> {
 ListItr(int index) {//继承Itr ，实现ListIterator接口
 super();
 cursor = index; //从后向前时遍历数组的下标传参为0
 }
  public boolean hasPrevious() {
  return cursor != 0; //不为0才会进行从后向前遍历
  }
```

```
13  public int nextIndex() {
14  return cursor;
15  }
16  public int previousIndex() {
17  return cursor - 1;
18  }
19  @SuppressWarnings("unchecked") //注解：压制警告
20  public E previous() {
21  checkForComodification(); //快速失败机制
22  int i = cursor - 1;
23  if (i < 0)
24  throw new NoSuchElementException();
25  Object[] elementData = ArrayList.this.elementData;
26  if (i >= elementData.length)
27  throw new ConcurrentModificationException();
28  cursor = i;
29  return (E) elementData[lastRet = i];
30  }
31
32  //还实现了ListIterator的set(),add()方法? 不知道要干啥?
33  set();
34  add();
35
```

由源码可以看出必须要进行一个从前向后遍历使遍历数组的下标为集合的最后一个元素，才能实行从后向前遍历（且必须是同一个"迭代器"）。（这样做岂麻烦了很多，想从后向前遍历数组必须经过从前向后的过程。）

4.3Enumeration枚举输出-Vector类支持

只有Vector的elements（）方法取得Enumeration对象才能进行遍历

速度是最快的迭代器

```
1 public boolean hasMoreElements(): 判断是否有下一个元素
```

```
1 :public E nextElements():取得下一个元素
```

```
1 :public Enumeration elements() : 取得Enumeration接口对象
```

```
1 Enumeration<String> enumeration = vector.elements(); //取得Enumeration接口对象
```

```
while (enumeration.hasMoreElements()) {//判断
 System.out.println(enumeration.nextElement());//遍历
}
```

一些旧的操作类库上依然只支持Enumeration，而不支持Iterator。

4.4for - each输出（所有子类都满足）

能使用foreach输出的本质在于各个集合都内置了迭代器。

总结：

> **1. 看见集合输出就使用Iterator**
> **2. Iterator和Enumeration中的方法掌握好**

5.fail-fast机制（快速失败机制）

ConcureentModificationException发生在Collection集合使用迭代器遍历时，**使用了集合类提供的修改内容的方法报错**，而如果使用**iterator的迭代器的remove()不会出现此错误。**

注意：以后在进行集合输出的时候不要修改集合中元素！！！

- [x] ~~(同步问题，如何产生的？课后思考)~~

解答：list支持多线程同时访问，当一个线程对list集合进行更改时，再有另个线程对list集合进行访问时，访问的就不是最新的数据，造成（"脏读"）。

- [x] ~~修改iterator,会反应在原本的集合当中吗？~~

解答：会

- [x] ~~产生错误的原因，以及为何要有这种机制。~~

解答：产生错误的原因

防止脏读！！

结合源码：

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

Collection集合中的modCount表示当前集合修改的次数（集合的一个属性）

expectedModCount 是迭代器中记录当前集合的修改次数(迭代器的一个属性)

当取得集合迭代器时 list.iterator()，int expectedModCount = modCount;（才赋值）（调用构造方法时才会赋值）。换言之，迭代器就是当前集合的一个副本。

```
int expectedModCount = modCount;//迭代器中代码
```

当使用list本身的方法取修改list集合时就会改变modCount的数据，在进行遍历时，**每次获取数据时都会执行checkForComodification方法，检查list集合中modCount和迭代器中的expectedModCount是否相同，（保证读到的数据是最新的数据）。**所以只要两个变量数组不相等时就会抛出异常**ConcurrentModificationException。**

快速失败机制作用：

快速失败策略保证了所有用户在进行迭代遍历时，拿到的数据一定是最新的数据（避免”脏读“产生）。

6.fail-safe

：不产生ConcurrentModeifiicationException异常

juc包下所有线程安全集合（CopyOnWriteArrayList）

总结：以后在迭代器遍历时，不要修改集合内容。

5.Map集合（使用版）

Map接口是java中保存**二元偶对象（键值对）**的最顶层接口

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

特点：key值唯一，通过一个key值一定能唯一找到一个value值。

5.1Map中的核心方法

```
public V put (K key,V value):向Map集合中添加数据
public V get(K key):根据指定的key值取得相应的value值，若没有此key值，返回null

public Set <Map.Entry<K,V>> entrySet():将Map集合转为Set集合.

public Set<K> keySet :返回所有key值得 Set 集合，key不重复。
public collection<V> values():  反回所有value值，value可以重复。
```

No.	方法名称	类型	描述
1.	public V put(K key, V value);	普通	向 Map 中追加数据
2.	public V get(Object key);	普通	根据 key 取得对应的 value，如果没有返回 null
3.	public Set<K> keySet();	普通	取得所有 key 信息、key 不能重复
4.	public Collection<V> values();	普通	取得所有 value 信息，可以重复
5.	public Set<Map.Entry<K, V>> entrySet();	普通	将 Map 集合变为 Set 集合

Map接口中有如下常用四个子类

HashMap（使用频率最高的-必考），TreeMap,Hashtable,ConcurrentHashMap

5.2 Map接口的使用

//当key值重复时，再次put变为相应得value更新得操作。（key值重复则会实行更新（覆盖旧值））
//key值不存在则反回null

5.3 HashMap-类比Hashtable（面试题）

HashMap

1.**允许key和value为null,且key值有且只有一个为null，value允许有任意多个为null。**

2.版本号 JDK1.2

3.异步处理，效率高，线程不安全

4.底层哈希表+红黑树（JDK1.8）

Hashtable(古老类)

1.key与value均不能为null

2.版本号：JDK1.0

3.使用方法加锁（同步处理），效率低，线程安全

4.底层哈希表

No.	区别	HashMap	Hashtable
1	推出版本	JDK1.2	JDK1.0
2	性能	异步处理，性能高	同步处理、性能较低
3	安全性	非线程安全	线程安全
4	null 操作	允许存放 null(有且只有一个)	key 与 value 都不为空，否则出现 NullPointerException

5.4HashMap源码分析

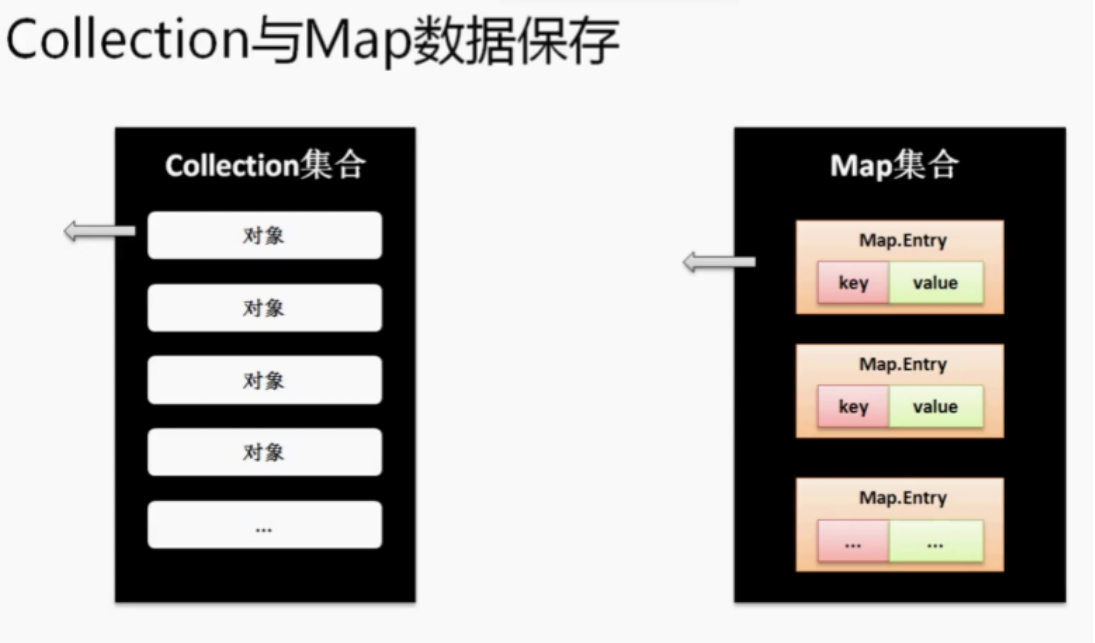
HashMap源码分析: HashMap 设计与实现是个非常高频的面试题，所以我会在这进行相对详细的源码解读，主要围绕：

- HashMap 内部实现基本点分析。
- 容量（capcity）和负载系数（load factor）、树化

6.1****Map集合使用迭代器（iterator）输出*****

collection几个

```
1 public Set <Map.Entry<K,V>> entrySet():将Map集合转为Set集合.
```

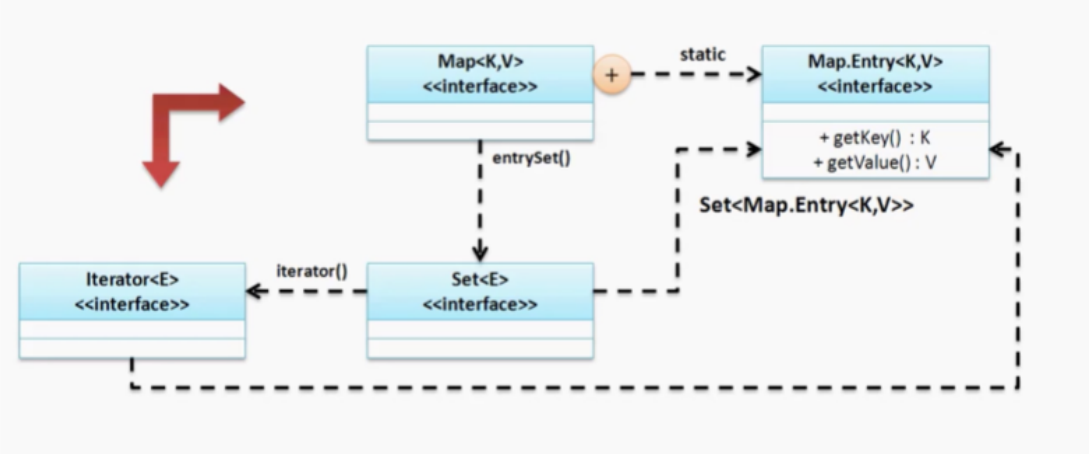


Collection和map一个一个实体对象的保存。

在Map接口里面有一个重要的方法，将Map集合转变为Set集合：

```
1 public Set <Map.Entry<K,V>> entrySet():将Map集合转为Set集合.
```

map并没有继承iterator接口，但是它如何使用迭代器输出呢?



利用**核心方法entrySet（）返回一个set集合**，然后利用set集合进行迭代器输出。**（标准输出）**

6.2TreeMap子类

TreeMap是一个可以排序的map子类，它是按照key的内容排序得到。

使用TreeMap由于是有序的所以是需要实现Comparable compareto()或者有的单独一个类实现Comparator

compare()方法。

结论：有Comparable出现的地方，判断数据就依靠compareTo()方法完成，不再需要equals()与hashCode()

Map集合小结：

> 1.Collection 保存数据的目的一般用于输出iterator()，Map保存数据的目的是为了根据key查找，找不到返回null.
> 2.Map使用Iterator输出（Map Entry的作用）
> 3.HashMap数据结构一定要理解（链表与红黑树）、HashMap与Hashtable区别

7栈与队列

栈：FILO 先入后出

函数栈帧，浏览器的标签页的后退，安卓Activity 的后退，编辑器撤销

7.1stack

java库中 Stack类

入栈：push();

```
1 public E push(E item) {
```

出栈：pop();

```
1 public synchronized E pop() {
```

返回栈定元素，但不出栈：peek();

```
1 public synchronized E peek() {
```

括号匹配，自己实现一个html识别器

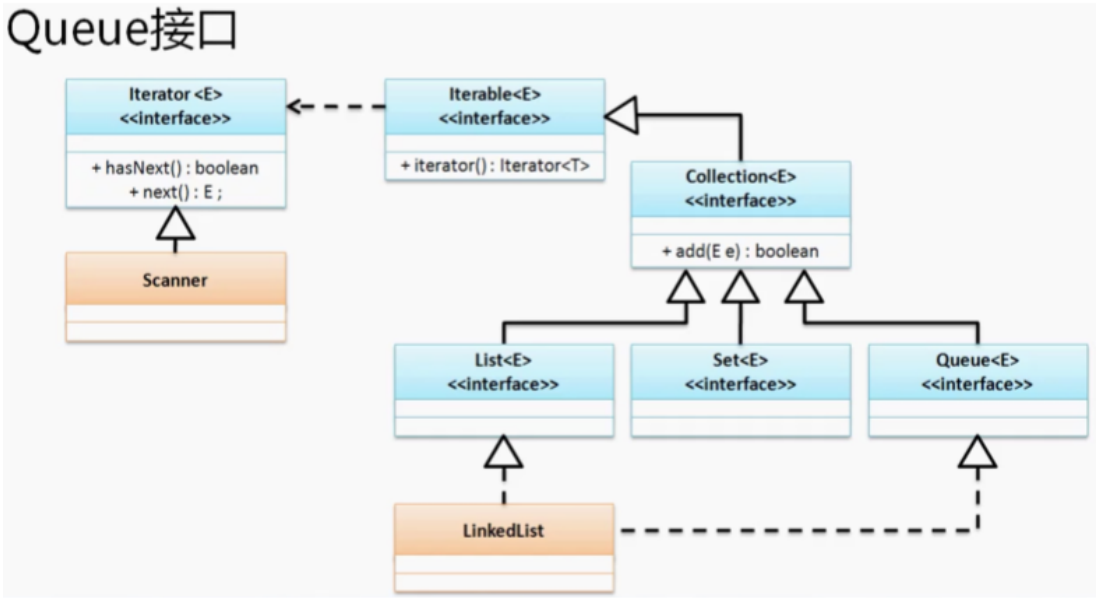
<html>
    <head></head>
    <body></body>
</html>

队列FIFO:先入先出

美味不用等

消息队列：kafakm,RobbitMQ

7.2Queue接口



入队列：add()

```
boolean add(E e);
```

出队列：poll()

```
E poll();
```

返回队列头元素，但不出队：peek()

```
E peek();
```

8.properties属性文件操作-资源文件

在java中有一种属性文件(资源文件)的定义：*.properties文件，在这种文件里面其内容的保存形式为"key = value"，通过ResourceBundle类读取的时候只能读取内容，要想编辑其内容则需要通过Properties类来完成，这 个类是专门做属性处理的。

资源文件内容都是k-v格式，并且无论key,values 都是String类型设置属性。

设置属性

```
setProperty(Strinhg key,String value)
```

取得属性

```
getProperty(String key):String
```

若没有指定key值，返回null

```
getProperty(String key,String defaultValue);
```

若没有指定key值，返回默认值。

将资源内容输出输入到目标终端

```
Store(OuptStream cut,String,comments)
```

从目标终端中读取数据

```
load(InputSteam in);
```

9.Collections工具类

Array-Arrays

Executor-Executors

Collection-Collections

集合反转、排序

I.将线程不安全集合包装为线程安全集合（不推荐）

```
SynchronizedList(List<E> list) { //Collections 中的内部类
```

在add,remove等修改方法上使用同步代码块保证线程安全，效率较低。

```
eg:public void add(int index, E element) {
 synchronized (mutex) {list.add(index, element);}
}
public E remove(int index) {
 synchronized (mutex) {return list.remove(index);}
}
```

要使用线程安全集合，推荐使用JUC包下的并发集合类
(ConCurrentHashMap,CopyOnWriteArrayList)

II集合排序

Collections.sort():

```
public static <T extends Comparable<? super T>> void sort(List<T> list) {
```

自定义类必须自身实现Comparetable(内部比较器)， 或者有一个外部比较器（Comparator）

III.集合反转

Collection.reverse(集合名称)

```
public static void reverse(List<?> list) {
```

10.Stream 数据流（JDK8新增）：Collection接口

JDK1.8发行的时候实际上是世界上大数据兴起的时候，在整个大数据的开发里面有一个最经典的模型:MapReduce。 实际上这属于数据的两个操作阶段：

> **1. Map: 处理数据**
> **2. Reduce: 分析数据**

而在Java类集中，由于其本身的作用就可以进行大量数据的存储，所以就顺其自然的产生了MapReduce操作，而这 些操作可以通过Stream数据流来完成。

核心方法：取得接口的Stream流

Stream<E> Stream()

```
default Stream<E> stream() {
```

常用方法：

1.forEach （）方法：集合输出

```
void forEach(Consumer<? super T> action);
```

2.数据过滤

filter():

```
Stream<T> filter(Predicate<? super T> predicate);
```

3.取得最大最小值

max()/min()方法

```
Optional<T> max(Comparator<? super T> comparator);
```

```
Optional<T> min(Comparator<? super T> comparator);
```

4.数据前期处理

map()：

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

5.数据处理后的收集擦操作：

reduce():

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

> **MapReduce是整个Stream的核心所在。MapReduce的操作也是由两个阶段所组成：**
> **1. map():指的是针对于数据进行先期的操作处理。例如：简单的数学运算等**
> **2. reduce():进行数据的统计分析。**

为了进一步观察更加丰富的处理操作，可以再做一些数据的统计分析。对于当前的操作如果要进行一些数量的统 计，其最终的结果应为double型数据。在Stream接口中就提供有一个map结果变为Double型的操作:

```
统计分析: public DoubleStream mapToDouble(ToDoubleFunction<? super T> mappe
r);
```

此时的方法返回的是一个DoubleStream接口对象，这里面就可以完成统计操作，这个统计使用的方法如下:

```
统计方法:DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);
```

在Stream接口中重点有两个操作方法:

```
1. 设置取出最大内容: public Stream limit(long maxSize);
```

```
2. 跳过的数据量: public Stream skip(long n);
```

以上代码就使用了skip()、limit()方法对数据做了分页处理，还结合了map()方法做了简单的数据处理。

10.1Collection改进

从JDK1.8开始，Collection口里面除了定义一些抽象方法外，也提供了一些普通方法，下面来观察如下几个方法:

> **1. forEach()输出支持: default void forEach(Consumer<? super T> action)**
> **2. 取得Stream数据流对象: default Stream stream()**