Guava引入了很多JDK没有的、但我们发现明显有用的新集合类型。这些新类型是为了和JDK集合框架共存，而没有往JDK集合抽象中硬塞其他
精准地遵循了JDK接口契约。

Guava中定义的新集合有：

- Multiset
- SortedMultiset
- Multimap
- ListMultimap
- SetMultimap
- BiMap
- ClassToInstanceMap
- Table

Multiset

Multiset和Set的区别就是可以保存多个相同的对象。在JDK中，List和Set有一个基本的区别，就是List可以包含多个相同对象，且是有顺序的，而Set不
据了List和Set之间的一个灰色地带：允许重复，但是不保证顺序。

常见使用场景：Multiset有一个有用的功能，就是跟踪每种对象的数量，所以你可以用来进行数字统计。 常见的普通实现方式如下：

```
1. Map<String, Integer> counts = new HashMap<String, Integer>();
2. for (String word : words) {
3.     Integer count = counts.get(word);
4.     if (count == null) {
5.         counts.put(word, 1);
6.     } else {
7.         counts.put(word, count + 1);
8.     }
9. }
```

如果使用实现Multiset接口的具体类就可以很容易实现以上的功能需求：

```
1. public void testMultsetWordCount(){
2.         String strWorld="wer|dfd|dd|dfd|dda|de|dr";
3.         String[] words=strWorld.split("\\|");
4.         List<String> wordList=new ArrayList<String>();
5.         for (String word : words) {
6.             wordList.add(word);
7.         }
8.         Multiset<String> wordsMultiset = HashMultiset.create();
9.         wordsMultiset.addAll(wordList);
10.
11.         for(String key:wordsMultiset.elementSet()){
12.             System.out.println(key+" count: "+wordsMultiset.count(key));
13.         }
14. }
```

Multiset接口定义的接口主要有：

- add(E element) :向其中添加单个元素
- add(E element,int occurrences)：向其中添加指定个数的元素
- count(Object element)：返回给定参数元素的个数
- remove(E element)：移除一个元素，其count值 会响应减少
- remove(E element,int occurrences)：移除相应个数的元素
- elementSet()：将不同的元素放入一个Set中
- entrySet()：类似与Map.entrySet 返回Set<Multiset.Entry>。包含的Entry支持使用getElement()和getCount()
- setCount(E element ,int count)：设定某一个元素的重复次数
- setCount(E element,int oldCount,int newCount)：将符合原有重复个数的元素修改为新的重复次数
- retainAll(Collection c)：保留出现在给定集合参数的所有的元素
- removeAll(Collectionc)：去除出现给给定集合参数的所有的元素

MultiSet接口注意点：

- Multiset中的元素计数只能是正数。任何元素的计数都不能为负，也不能是0。elementSet()和entrySet()视图中也不会有这样的元素。
- multiset.size()返回集合的大小，等同于所有元素计数的总和。对于不重复元素的个数，应使用elementSet().size()方法。（因此，add(E
- multiset.iterator()会迭代重复元素，因此迭代长度等于multiset.size()。
- Multiset支持直接增加、减少或设置元素的计数。setCount(elem, 0)等同于移除所有elem。
- 对multiset 中没有的元素，multiset.count(elem)始终返回0

Guava提供了多种Multiset的实现，大致对应JDK中Map的各种实现：

Map	对应的Multiset	是否支持null元素
HashMap	HashMultiset	是
TreeMap	TreeMultiset	是（如果comparator支
LinkedHashMap	LinkedHashMultiset	是
ConcurrentHashMap	ConcurrentHashMultiset	否
ImmutableMap	ImmutableMultiset	否

Multimap

在日常的开发工作中，我们有的时候需要构造像Map<K, List<V>>或者Map<K, Set<V>>这样比较复杂的集合类型的数据结构，以便做相应的业务逻

Guava的Multimap就提供了一个方便地把一个键对应到多个值的数据结构。让我们可以简单优雅的实现上面复杂的数据结构。

Multimap也支持一系列强大的视图功能：

1.asMap把自身Multimap<K, V>映射成Map<K, Collection<V>>视图。这个Map视图支持remove和修改操作，但是不支持put和putAll，的key，而且你希望返回的是null而不是一个空的可修改的集合的时候就可以调用asMap().get(key)。（你可以强制转型asMap().get(key)的结果类型ListMultimap的结果转成List型－但是直接把ListMultimap转成Map<K, List<V>>是不行的。）

2.entries视图是把Multimap里所有的键值对以Collection<Map.Entry<K, V>>的形式展现。

3.keySet视图是把Multimap的键集合作为视图

4.keys视图返回的是个Multiset，这个Multiset是以不重复的键对应的个数作为视图。这个Multiset可以通过支持移除操作而不是添加操作来修改M

5.values()视图能把Multimap里的所有值“平展”成一个Collection<V>。这个操作和Iterables.concat(multimap.asMap().values())很相似，只

尽管Multimap的实现用到了Map，但Multimap<K, V>不是Map<K, Collection<V>>。因为两者有明显区别：

1.Multimap.get(key)一定返回一个非null的集合。但这不表示Multimap使用了内存来关联这些键，相反，返回的集合只是个允许添加元素的视图，

2.如果你喜欢像Map那样当不存在键的时候要返回null，而不是Multimap那样返回空集合的话，可以用asMap()返回的视图来得到Map<K, Collec的Collection<V>强转型为List或Set）。

3.Multimap.containsKey(key)只有在这个键存在的时候才返回true。

4.Multimap.entries()返回的是Multimap所有的键值对。但是如果需要key-collection的键值对，那就得用asMap().entries()。

5.Multimap.size()返回的是entries的数量，而不是不重复键的数量。如果要得到不重复键的数目就得用Multimap.keySet().size()。

Multimap提供了多种形式的实现。在大多数要使用Map<K, Collection<V>>的地方，你都可以使用它们：

实现	键行为类似	值行为类似
ArrayListMultimap	HashMap	ArrayList
HashMultimap	HashMap	HashSet
LinkedListMultimap*	LinkedHashMap*	LinkedList*
LinkedHashMultimap**	LinkedHashMap	LinkedHashMap
TreeMultimap	TreeMap	TreeSet
`ImmutableListMultimap`	ImmutableMap	ImmutableList
ImmutableSetMultimap	ImmutableMap	ImmutableSet

BiMap

传统上，实现键值对的双向映射需要维护两个单独的map，并保持它们间的同步。但这种方式很容易出错，而且对于值已经在map中的情况，

```
Map<String, Integer> nameToId = Maps.newHashMap();
Map<Integer, String> idToName = Maps.newHashMap();

nameToId.put("Bob", 42);
idToName.put(42, "Bob");
//如果"Bob"和42已经在map中了，会发生什么?
//如果我们忘了同步两个map，会有诡异的bug发生...
```

BiMap提供了一种新的集合类型，它提供了key和value的双向关联的数据结构。

- 可以用 inverse()反转BiMap<K, V>的键值映射
- 保证值是唯一的，因此 values()返回Set而不是普通的Collection

在BiMap中，如果你想把键映射到已经存在的值，会抛出IllegalArgumentException异常。如果对特定值，你想要强制替换它的键，请使用 BiM

创建方式

```
BiMap<String, Integer> userId = HashBiMap.create();
...

String userForId = userId.inverse().get(id);
```

BiMap的各种实现

键-值实现	值-键实现	对应的BiMap实现
HashMap	HashMap	HashBiMap
ImmutableMap	ImmutableMap	ImmutableBiMap
EnumMap	EnumMap	EnumBiMap
EnumMap	HashMap	EnumHashBiMap

Table

使用Table可以实现二维矩阵的数据结构，当你想使用多个键做索引的时候，你可能会用类似Map(FirstName, Map(LastNa
丑陋，使用上也不友好。Guava为此提供了新集合类型Table，Table是Guava提供的一个接口 Interface Table(R,C,V)，由ro
组成

它有两个支持所有类型的键："行"和"列"。Table提供多种视图，以便你从各种角度使用它：

- rowMap()：用Map<R, Map<C, V>>表现Table<R, C, V>。同样的， rowKeySet()返回"行"的集合Set<R>。
- row(r)：用Map<C, V>返回给定"行"的所有列，对这个map进行的写操作也将写入Table中。
- 类似的列访问方法：columnMap()、columnKeySet()、column(c)。（基于列的访问会比基于的行访问稍微低效点）
- cellSet()：用元素类型为Table.Cell<R, C, V>的Set表现Table<R, C, V>。Cell类似于Map.Entry，但它是用行和列两个键区分的。

Table有如下几种实现：

- HashBasedTable：本质上用HashMap<R, HashMap<C, V>>实现；
- TreeBasedTable：本质上用TreeMap<R, TreeMap<C,V>>实现；
- ImmutableTable：本质上用ImmutableMap<R, ImmutableMap<C, V>>实现；注：ImmutableTable对稀疏或密集的数据集都有优化。
- ArrayTable：要求在构造时就指定行和列的大小，本质上由一个二维数组实现，以提升访问速度和密集Table的内存利用率。ArrayTable
  见Javadoc了解详情。

具体代码示例：

```
import java.util.Collection;
import java.util.Map;
```

```java
import java.util.Set;

import com.google.common.collect.HashBasedTable;
import com.google.common.collect.Table;
import com.google.common.collect.Table.Cell;
import com.google.common.collect.Tables;

/**
 * 测试Table ： Table就是有了双键的Map
 * 学生(rowkey)--课程(columkey)--成绩(value)
 *      lf      --      a       -- 80
 *      dn      --      b       -- 90
 *      cf      --      a       -- 88
 *
 */
public class Demo06 {
    public static void main(String[] args) {
        Table<String,String,Integer> table = HashBasedTable.create();
        table.put("a", "javase", 80);
        table.put("b", "javase", 90);
        table.put("a", "javame", 100);
        table.put("d", "guava", 70);

        //得到所有的行数据
        Set<Cell<String,String,Integer>> cellset = table.cellSet();

        for(Cell<String,String,Integer> temp:cellset){
            System.out.println(temp.getRowKey()+"--"+temp.getColumnKey()+"--"+temp.getValue());
        }

        System.out.println("-------rowKey和columnKey转换---------");
        Table<String,String,Integer> table1 = Tables.transpose(table);
        Set<Cell<String,String,Integer>> cellset1 = table1.cellSet();

        for(Cell<String,String,Integer> temp:cellset1){
            System.out.println(temp.getRowKey()+"--"+temp.getColumnKey()+"--"+temp.getValue());
        }

        System.out.println("-------按学生查看成绩---------");

        System.out.print("学生\t");
        Set<String> cours = table.columnKeySet();
        for(String temp:cours){
            System.out.print(temp+"\t");
        }

        System.out.println();

        Set<String> stu = table.rowKeySet();
        for(String temp:stu){
            System.out.print(temp);
            Map<String,Integer> map = table.row(temp);
            for(String temp1:cours){
                System.out.print("\t"+map.get(temp1));
            }
            System.out.println();
        }

        System.out.println("-------按课程查看成绩---------");

        System.out.print("课程\t");
        Set<String> stu1 = table.rowKeySet();
        for(String temp:stu1){
            System.out.print(temp+"\t");
        }

        System.out.println();

        Set<String> cours1 = table.columnKeySet();
        for(String temp:cours1){
            System.out.print(temp);
            Map<String,Integer> map1 = table.column(temp);
            for(String temp1:stu1){
                System.out.print("\t"+map1.get(temp1));
            }
            System.out.println();
        }
    }

}
```

输出结果如下：

```
d--guava--70
b--javase--90
a--javase--80
a--javame--100
-------转换---------
guava--d--70
javase--b--90
javase--a--80
javame--a--100
-------按学生查看成绩---------
```

```
学生  guava   javase  javame
d   70  null    null
b   null    90  null
a   null    80  100
--------按课程查看成绩---------
课程  d   b   a
guava   70  null    null
javase  null    90  80
javame  null    null    100
```

ClassToInstanceMap

ClassToInstanceMap是一种特殊的Map：它的键是类型，而值是符合键所指类型的对象。

为了扩展Map接口，ClassToInstanceMap额外声明了两个方法：T getInstance(Class<T>) 和T putInstance(Class<T>, T)，从而避免强制类型转

对于ClassToInstanceMap，Guava提供了两种有用的实现：MutableClassToInstanceMap和 ImmutableClassToInstanceMap。