https://chen310.gitbooks.io/guava/content/collections.html
http://jackyrong.iteye.com/blog/2150912

Guava对JDK的集合做了扩充，主要表现在：

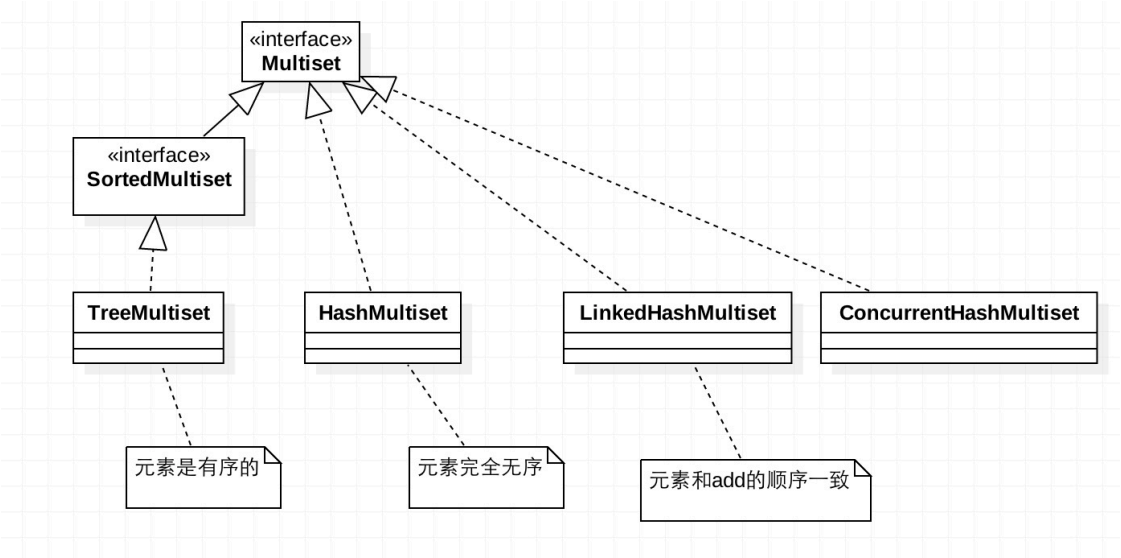
- 增加了一些新的集合类
- 更好的不可变集合
- 增加了更多实用的集合处理方法

新增集合

MultiSet

记录集合中元素的重复次数。注意，这个类并不继承Set接口，而是直接继承Collection

MultiSet继承关系：



BiMap

BiMap继承于Map接口

功能：双向Map，既可通过key取value，也可通过value取key。

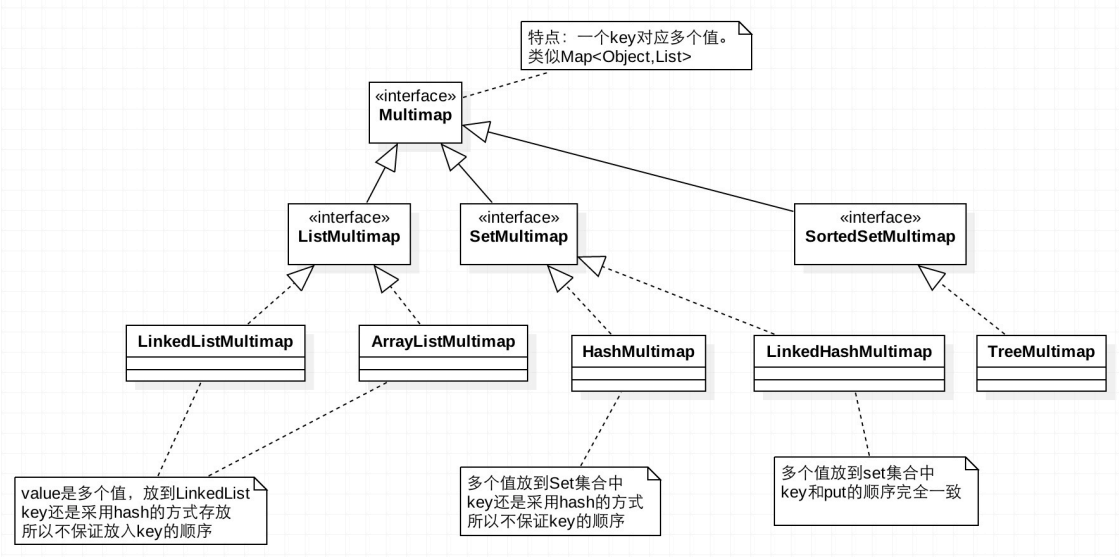
要求：key唯一，value唯一

实现类：HashBiMap

MultiMap

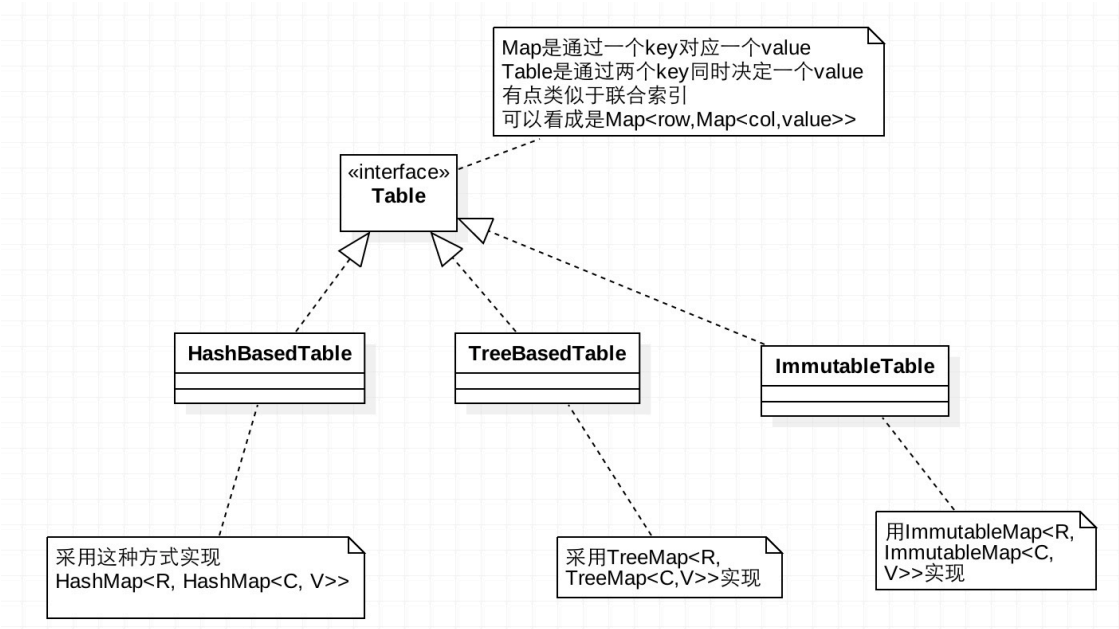
MultiMap不继承于JDK的任何接口。

功能：一个key对应多个value的map，类似`Map<Object,List>`。
添加元素时自动将key相同的元素放到List中
类图如下：

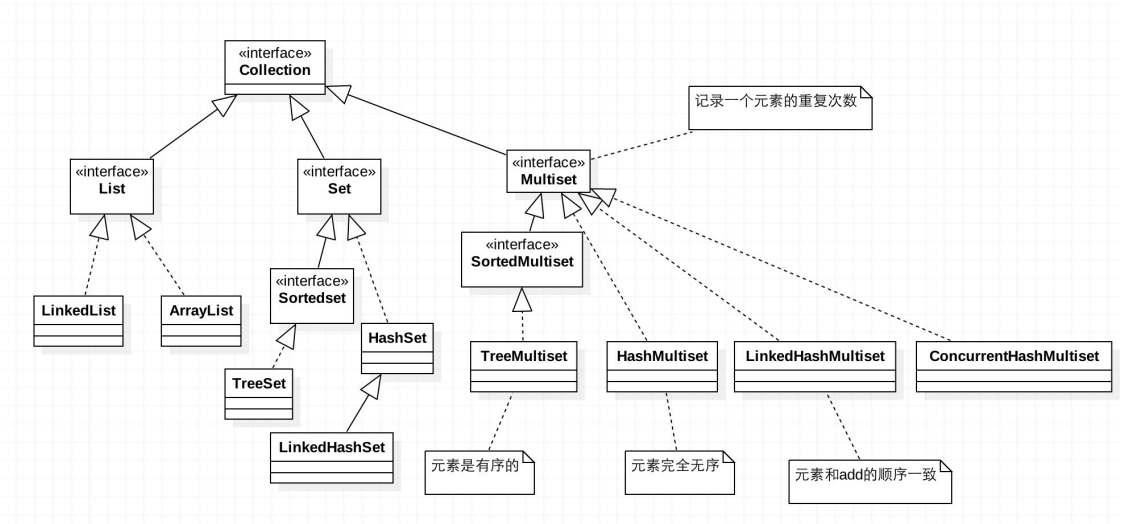


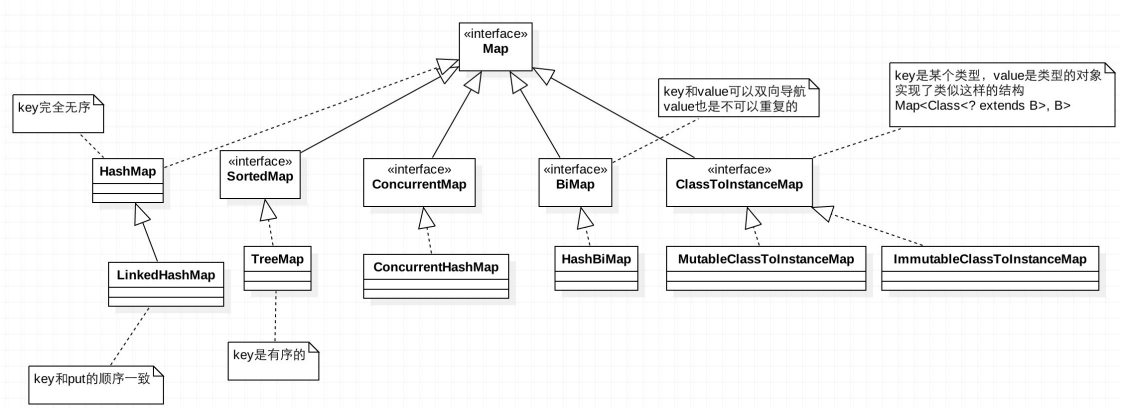
Table

Table不继承于JDK的任何接口。
功能：Map是通过一个key对应一个value，Table是通过两个key同时决定一个value。有点类似于联合索引，
可以看成是 `Map<row,Map<col,value>>`
类图如下：



Guava集合类和JDK集合类的关系类图





更多的集合处理方法

对集合的处理，目前我经常用到的有：

- 索引
- 去重
- 集合运算
- 过滤器(filter)

索引

经常有这样的需求，比如有一个技师信息的List，现有一个TechnicianId，要在List中找出这个TechnicianId对应的技师信息。 原来我们可能会写类似这样的代码：

```
public TechnicianDTO findByTechnicianId(int technicianId) {
```

```
        for (TechnicianDTO technicianDTO : technicianDTOs) {
            if (technicianDTO.getTechnicianId() == technicianId) {
                return technicianDTO;
            }
        }
        return null;
    }
```

这样的写法，每次查询technicianId都需要遍历一遍technicianDTOs，更聪明的做法，将technicianDTOs转换为一个`Map<Integer,TechnicianDTO>`，key为TechnicianId，value为对应的TechnicianDTO。

在Guava看来，完成这样的转换就相当于给一组TechnicianDTO数据的TechnicianId字段上加了索引，事实上也是的　　确如此，所以Guava提供了方便的加索引方法，索引本身有两种：**唯一索引**和**非唯一索引**。上面这个列子中，根据业务含义，可以加唯一索引，如果加非唯一索引，返回的结果类似于`Map<Integer,List<TechnicianDTO>>`，相当于根据某个字段聚合了。分别看如下代码：

```
/**
 * 对集合添加唯一索引
 * @param technicianId
 * @return
 */
public TechnicianDTO findByTechnicianId2(int technicianId) {
    ImmutableMap<Integer, TechnicianDTO> uniqueIndex =
FluentIterable.from(technicianDTOs)
        .uniqueIndex(new Function<TechnicianDTO, Integer>() {
        @Override
        public Integer apply(TechnicianDTO technicianDTO) {
            return technicianDTO.getTechnicianId();
        }
    });
    return uniqueIndex.get(technicianId);
}

/**
 * 对集合添加非唯一索引
 * @param technicianId
 * @return
 */
public List<TechnicianDTO> findByTechnicianId(int technicianId) {
```

```
    ImmutableListMultimap<Integer, TechnicianDTO> index =
FluentIterable.from(technicianDTOs)
        .index(new Function<TechnicianDTO, Integer>() {
        @Override
        public Integer apply(TechnicianDTO technicianDTO) {
            return technicianDTO.getTechnicianId();
        }
    });
    return index.get(technicianId);
}
```

去重

展示了两种去重情况：1. 保证去重后和原始序一致 2. 不保证顺序

```
@Test
public void test50() {
    List<Integer> list = Lists.newArrayList(1, 2, 2, 1, 4, 5, 4, 3);
    ImmutableList<Integer> distinct
        = ImmutableSet.copyOf(list).asList();//保证序和原始序一致
    HashSet<Integer> distinct1 = Sets.newHashSet(list);//不保证序和原始序
一致

    System.out.println(distinct);
    System.out.println(distinct1);
}
```

集合运算

集合运算主要有：并集、差集、交集。

代码：

```
@Test
public void test51() {
    List<Integer> list1 = Lists.newArrayList(1, 2, 2, 1, 4, 5, 4, 3);
    List<Integer> list2 = Lists.newArrayList(1, 2, 3, 7, 8, 9);
    //并集
    Sets.SetView<Integer> union
        = Sets.union(Sets.newHashSet(list1), Sets.newHashSet(list2));

    //差集(在list1中,不在list2中)
    Sets.SetView<Integer> difference
        = Sets.difference(Sets.newHashSet(list1), Sets.newHashSet(list2));
```

```
        //差集(在list2中,不在list1中)
        Sets.SetView<Integer> difference1
                = Sets.difference(Sets.newHashSet(list2), Sets.newHashSet(list1));

        //交集
        Sets.SetView<Integer> intersection
                = Sets.intersection(Sets.newHashSet(list1), Sets.newHashSet(list2));

        System.out.println(union);     //~out:[1, 2, 3, 4, 5, 7, 8, 9]
        System.out.println(difference);     //~out:[4, 5]
        System.out.println(difference1);     //~out:[7, 8, 9]
        System.out.println(intersection);     //~out:[1, 2, 3]
}
```

过滤器(filter)

1) . 给出一个list，过滤出含有字母a的元素

```
@Test
public void whenFilterWithIterables_thenFiltered() {
    List<String> names = Lists.newArrayList("John", "Jane", "Adam", "Tom");
    //过滤出list中含有含有a的子集
    Iterable<String> result = Iterables.filter(names,
Predicates.containsPattern("a"));
    assertThat(result, containsInAnyOrder("Jane", "Adam"));   //true
}
```

此外，可以使用Collections2.filter() 去进行过滤

```
@Test
public void whenFilterWithCollections2_thenFiltered() {
    List<String> names = Lists.newArrayList("John", "Jane", "Adam", "Tom");
    Collection<String> result = Collections2.filter(names,
Predicates.containsPattern("a"));

    assertEquals(2, result.size());
    assertThat(result, containsInAnyOrder("Jane", "Adam"));

    result.add("anna");
    assertEquals(5, names.size());
}
```

这里注意的是，Collections2.filter中，当在上面的result中增加了元素后，会直接影响

原来的names这个list的，就是names中的集合元素是5了。

再来看下predicates判断语言，

com.google.common.base. Predicate : 根据输入值得到 true 或者 false

2) 拿Collections2中有2个函数式编程的接口： filter , transform ,例如： 在Collection<Integer>中过滤大于某数的内容：

```
Collection<Integer> filterList = Collections2.filter(collections
    , new Predicate<Integer>(){
                @Override
                public boolean apply(Integer input) {
                    if(input > 4)
                        return false;
                    else
                        return true;
                }
});
```

3) 把Lis<Integer>中的Integer类型转换为String , 并添加test作为后缀字符：

```
List<String> c2 = Lists.transform(list, new Function<Integer , String>(){
                @Override
                public String apply(Integer input) {
                    return String.valueOf(input) + "test";
                }
});
```

4) 找出包含J字母或包含a的元素

```
@Test
public void whenFilterCollectionWithCustomPredicate_thenFiltered() {
   Predicate<String> predicate = new Predicate<String>() {
      @Override
      public boolean apply(String input) {
         return input.startsWith("A") || input.startsWith("J");
      }
   };

   List<String> names = Lists.newArrayList("John", "Jane", "Adam", "Tom");
   Collection<String> result = Collections2.filter(names, predicate);
```

```
    assertEquals(3, result.size());
    assertThat(result, containsInAnyOrder("John", "Jane", "Adam"));
}
```

5) 将多个prdicate进行组合,找出包含J字母或不包含a的元素

```
@Test
public void whenFilterUsingMultiplePredicates_thenFiltered() {
    List<String> names = Lists.newArrayList("John", "Jane", "Adam", "Tom");
    Collection<String> result = Collections2.filter(names,
      Predicates.or(Predicates.containsPattern("J"),
      Predicates.not(Predicates.containsPattern("a"))));

    assertEquals(3, result.size());
    assertThat(result, containsInAnyOrder("John", "Jane", "Tom"));
}
```

6) 将集合中的空元素删除:

```
@Test
public void whenRemoveNullFromCollection_thenRemoved() {
    List<String> names = Lists.newArrayList("John", null, "Jane", null, "Adam",
"Tom");
    Collection<String> result = Collections2.filter(names, Predicates.notNull());

    assertEquals(4, result.size());
    assertThat(result, containsInAnyOrder("John", "Jane", "Adam", "Tom"));
}
```

7) 检查一个collection中的所有元素是否符合某个条件:

```
 @Test
public void whenCheckingIfAllElementsMatchACondition_thenCorrect() {
    List<String> names = Lists.newArrayList("John", "Jane", "Adam", "Tom");

    boolean result = Iterables.all(names, Predicates.containsPattern("n|m"));
    assertTrue(result);

    result = Iterables.all(names, Predicates.containsPattern("a"));
```

```
    assertFalse(result);
}
```

8) 下面看如何把一个list进行转换

```
@Test
public void whenTransformWithIterables_thenTransformed() {
    Function<String, Integer> function = new Function<String, Integer>() {
        @Override
        public Integer apply(String input) {
            return input.length();
        }
    };

    List<String> names = Lists.newArrayList("John", "Jane", "Adam", "Tom");
    Iterable<Integer> result = Iterables.transform(names, function);

    assertThat(result, contains(4, 4, 4, 3));
}
```