- 不可变集合

为什么要使用不可变集合?

不可变对象有很多优点，包括：

- 当对象被不可信的库调用时，不可变形式是安全的；
- 不可变对象被多个线程调用时，不存在竞态条件问题
- 不可变集合不需要考虑变化，因此可以节省时间和空间。所有不可变的集合都比它们的可变形式有更好的内存利用率（分析和测试细节）；
- 不可变对象因为有固定不变，可以作为常量来安全使用。
- 所有的guava不可变集合不接受null值, 如果你需要在不可变集合中使用null，请使用JDK中的Collections.unmodifiableXXX方法.

怎么使用不可变集合?

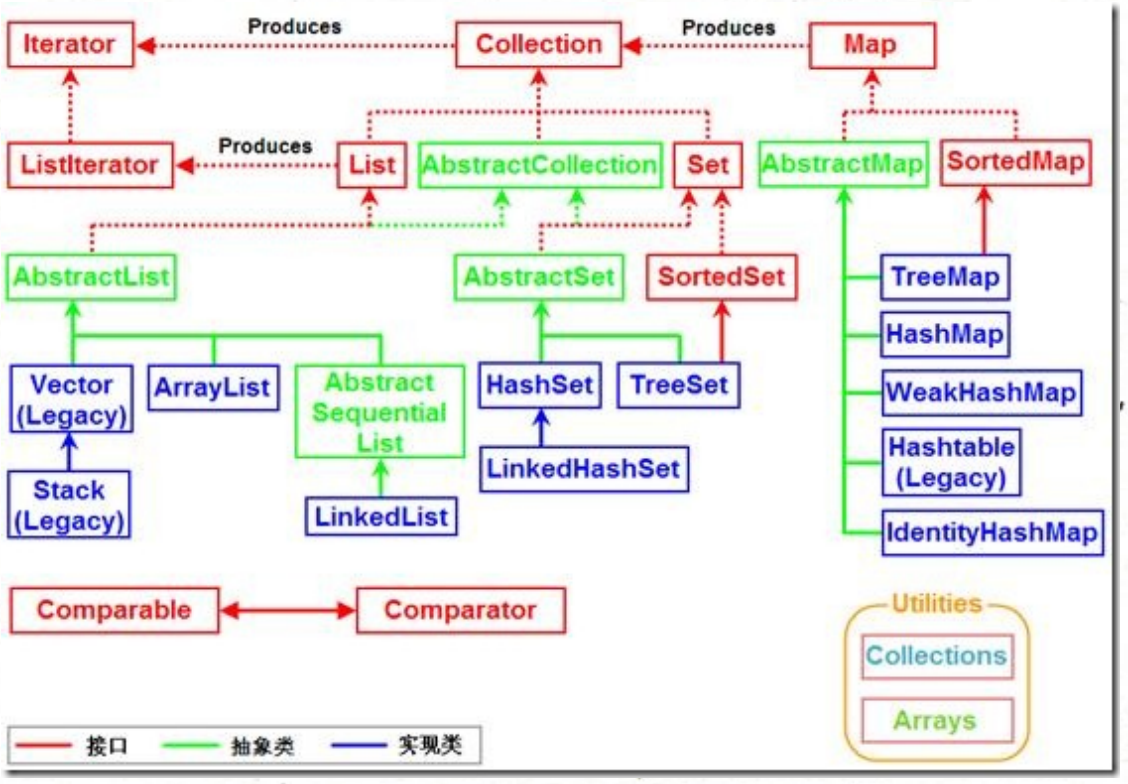
- copyOf方法，如ImmutableSet.copyOf(set);
- of方法，如ImmutableSet.of(“a”, “b”, “c”)或 ImmutableMap.of(“a”, 1, “b”, 2);
- Builder工具，如

```
public static final ImmutableSet<Color>
GOOGLE_COLORS =
        ImmutableSet.<Color>builder()
        .addAll(WEBSAFE_COLORS)
        .add(new Color(0, 191, 255))
        .build();
```

细节：关联可变集合和不可变集合

可变集合接口	属于JDK还是Guava	不可变版
Collection	JDK	ImmutableCollection
List	JDK	ImmutableList
Set	JDK	ImmutableSet
SortedSet/NavigableSet	JDK	ImmutableSortedSet
Map	JDK	ImmutableMap
SortedMap	JDK	ImmutableSortedMap
Multiset	Guava	ImmutableMultiset
SortedMultiset	Guava	ImmutableSortedMultiset
Multimap	Guava	ImmutableMultimap

java中集合类的关系图



Multiset(元素可重复)

Multiset：它可以多次添加相等的元素，Multiset元素的顺序是无关紧要的： Multiset {a,

a, b}和{a, b, a}是相等的
因此可以把Multiset看作 无序的ArrayList<E> 或者 Map<E, Integer>，键为元素，值为计数

方法	描述
count(E)	给定元素在Multiset中的计数
elementSet()	Multiset中不重复元素的集合，类型为Set<E>
entrySet()	和Map的entrySet类似，返回Set<Multiset.Entry<E>>，其中包含的Entry
add(E, int)	增加给定元素在Multiset中的计数
remove(E, int)	减少给定元素在Multiset中的计数
setCount(E, int)	设置给定元素在Multiset中的计数，不可以为负数；注:setCount(elem, 0)等
size()	返回集合元素的总个数(包括重复的元素)； 注:elementSet().size()返回不重
iterator()	会迭代重复元素，因此迭代长度等于multiset.size()。

Multimap(一键多值)

Multimap是把键映射到任意多个值的一般方式, 相当于Map<K, List<V>>
例如 : a -> [1, 2, 4]; b -> 3; c -> [4, 5]

方法签名	描述
put(K, V)	添加键到单个值的映射
putAll(K, Iterable<V>)	依次添加键到多个值的映射
remove(K, V)	移除键到值的映射；如果有这样的键值并成功移除，返回tru

BiMap(键值呼唤)

可以实现map的键值的互换

例如:

```
BiMap<Integer, String> biMap = HashBiMap.create();
biMap.put(1, "A");
biMap.put(2, "B");
System.out.println(biMap.toString());
//~out: {1=A, 2=B}
BiMap<String, Integer> inverseBiMap = biMap.inverse();
System.out.println(inverseBiMap);
//~out: {A=1, B=2}
```

Table(二维map)

二维表结构, 行key和列key一起决定value的值

```
Table<keyR, keyC, Value>  == Map<keyR Map<keyC, Value>> || Map<keyC
Map<keyR, Value>>
keyR - the type of the table row keys
keyC - the type of the table column keys
Value - the type of the mapped values

Table<String, Integer, String> aTable = HashBasedTable.create();
for (char a = 'A'; a <= 'C'; ++a) {
   for (Integer b = 1; b <= 3; ++b) {
      aTable.put(Character.toString(a), b, String.format("%c%d", a, b));
   }
}
System.out.println(aTable.row("A"));  //row为A的map
//~out : {1=A1, 2=A2, 3=A3}
System.out.println(aTable.column(2));  //column为2的map
//~out : {A=A2, B=B2, C=C2}
```

System.out.println(aTable.rowMap()); //Map<keyR Map<keyC, Value>>
//~out : {A={1=A1, 2=A2, 3=A3}, B={1=B1, 2=B2, 3=B3}, C={1=C1, 2=C2, 3=C3}}
System.out.println(aTable.columnMap()); //Map<keyC Map<keyR, Value>>
//~out : {1={A=A1, B=B1, C=C1}, 2={A=A2, B=B2, C=C2}, 3={A=A3, B=B3, C=C3}}