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概念。作为一般规则，Guava集合非常精准地遵循了JDK接口契约。

**Multiset**

统计一个词在文档中出现了多少次，传统的做法是这样的：

|  |  |  |
| --- | --- | --- |
| 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 | } |

这种写法很笨拙，也容易出错，并且不支持同时收集多种统计信息，如总词数。我们可以做的更好。  
  
Guava提供了一个新集合类型 [Multiset](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multiset.html)，它可以多次添加相等的元素。维基百科从数学角度这样定义Multiset：”集合[set]概念的延伸，它的元素可以重复出现…与集合[set]相同而与元组[tuple]相反的是，Multiset元素的顺序是无关紧要的：Multiset {a, a, b}和{a, b, a}是相等的”。——*译者注：这里所说的集合[set]是数学上的概念，Multiset继承自JDK中的Collection接口，而不是Set接口，所以包含重复元素并没有违反原有的接口契约。*

可以用两种方式看待Multiset：

* 没有元素顺序限制的ArrayList<E>
* Map<E, Integer>，键为元素，值为计数

Guava的Multiset API也结合考虑了这两种方式：  
当把Multiset看成普通的Collection时，它表现得就像无序的ArrayList：

* add(E)添加单个给定元素
* iterator()返回一个迭代器，包含Multiset的所有元素（包括重复的元素）
* size()返回所有元素的总个数（包括重复的元素）

当把Multiset看作Map<E, Integer>时，它也提供了符合性能期望的查询操作：

* count(Object)返回给定元素的计数。HashMultiset.count的复杂度为O(1)，TreeMultiset.count的复杂度为O(log n)。
* entrySet()返回Set<Multiset.Entry<E>>，和Map的entrySet类似。
* elementSet()返回所有不重复元素的Set<E>，和Map的keySet()类似。
* 所有Multiset实现的内存消耗随着不重复元素的个数线性增长。

值得注意的是，除了极少数情况，Multiset和JDK中原有的Collection接口契约完全一致——具体来说，TreeMultiset在判断元素是否相等时，与TreeSet一样用compare，而不是Object.equals。另外特别注意，Multiset.addAll(Collection)可以添加Collection中的所有元素并进行计数，这比用for循环往Map添加元素和计数方便多了。

|  |  |
| --- | --- |
| **方法** | **描述** |
| [count(E)](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multiset.html#count(java.lang.Object)) | 给定元素在Multiset中的计数 |
| [elementSet()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multiset.html#elementSet()) | Multiset中不重复元素的集合，类型为Set<E> |
| [entrySet()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multiset.html#entrySet()) | 和Map的entrySet类似，返回Set<Multiset.Entry<E>>，其中包含的Entry支持getElement()和getCount()方法 |
| [add(E, int)](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multiset.html#add(java.lang.Object,int)) | 增加给定元素在Multiset中的计数 |
| [remove(E, int)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Multiset.html#remove(java.lang.Object,%20int)) | 减少给定元素在Multiset中的计数 |
| [setCount(E, int)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Multiset.html#setCount(E,%20int)) | 设置给定元素在Multiset中的计数，不可以为负数 |
| size() | 返回集合元素的总个数（包括重复的元素） |

**Multiset不是Map**

请注意，Multiset<E>不是Map<E, Integer>，虽然Map可能是某些Multiset实现的一部分。准确来说Multiset是一种Collection类型，并履行了Collection接口相关的契约。关于Multiset和Map的显著区别还包括：

* 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)把multiset.size()增加1）
* 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。

**Multiset的各种实现**

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](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/HashMultiset.html) | 是 |
| TreeMap | [TreeMultiset](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/TreeMultiset.html) | 是（如果comparator支持的话） |
| LinkedHashMap | [LinkedHashMultiset](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/LinkedHashMultiset.html) | 是 |
| ConcurrentHashMap | [ConcurrentHashMultiset](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/ConcurrentHashMultiset.html) | 否 |
| ImmutableMap | [ImmutableMultiset](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/ImmutableMultiset.html) | 否 |

**SortedMultiset**

[SortedMultiset](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/SortedMultiset.html)是Multiset 接口的变种，它支持高效地获取指定范围的子集。比方说，你可以用 latencies.subMultiset(0,BoundType.CLOSED, 100, BoundType.OPEN).size()来统计你的站点中延迟在100毫秒以内的访问，然后把这个值和latencies.size()相比，以获取这个延迟水平在总体访问中的比例。

TreeMultiset实现SortedMultiset接口。在撰写本文档时，ImmutableSortedMultiset还在测试和GWT的兼容性。

**Multimap**

每个有经验的Java程序员都在某处实现过Map<K, List<V>>或Map<K, Set<V>>，并且要忍受这个结构的笨拙。例如，Map<K, Set<V>>通常用来表示非标定有向图。Guava的 [Multimap](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimap.html)可以很容易地把一个键映射到多个值。换句话说，Multimap是把键映射到任意多个值的一般方式。

可以用两种方式思考Multimap的概念：”键-单个值映射”的集合：

a -> 1 a -> 2 a ->4 b -> 3 c -> 5

或者”键-值集合映射”的映射：

a -> [1, 2, 4] b -> 3 c -> 5

一般来说，Multimap接口应该用第一种方式看待，但asMap()视图返回Map<K, Collection<V>>，让你可以按另一种方式看待Multimap。重要的是，不会有任何键映射到空集合：一个键要么至少到一个值，要么根本就不在Multimap中。

很少会直接使用Multimap接口，更多时候你会用ListMultimap或SetMultimap接口，它们分别把键映射到List或Set。

**修改Multimap**

[Multimap.get(key)](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimap.html#get(K))以集合形式返回键所对应的值视图，即使没有任何对应的值，也会返回空集合。ListMultimap.get(key)返回List，SetMultimap.get(key)返回Set。

对值视图集合进行的修改最终都会反映到底层的Multimap。例如：

|  |  |  |
| --- | --- | --- |
| 1 | Set<Person> aliceChildren = childrenMultimap.get(alice); | |
| 2 | aliceChildren.clear(); |

|  |  |
| --- | --- |
| 3 | aliceChildren.add(bob); |
| 4 | aliceChildren.add(carol); | |

其他（更直接地）修改Multimap的方法有：

|  |  |  |
| --- | --- | --- |
| **方法签名** | **描述** | **等价于** |
| [put(K, V)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Multimap.html#put(K,%20V)) | 添加键到单个值的映射 | multimap.get(key).add(value) |
| [putAll(K, Iterable<V>)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Multimap.html#putAll(K,%20java.lang.Iterable)) | 依次添加键到多个值的映射 | Iterables.addAll(multimap.get(key), values) |
| [remove(K, V)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Multimap.html#remove(java.lang.Object,%20java.lang.Object)) | 移除键到值的映射；如果有这样的键值并成功移除，返回true。 | multimap.get(key).remove(value) |
| [removeAll(K)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Multimap.html#removeAll(java.lang.Object)) | 清除键对应的所有值，返回的集合包含所有之前映射到K的值，但修改这个集合就不会影响Multimap了。 | multimap.get(key).clear() |
| [replaceValues(K, Iterable<V>)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Multimap.html#replaceValues(K,%20java.lang.Iterable)) | 清除键对应的所有值，并重新把key关联到Iterable中的每个元素。返回的集合包含所有之前映射到K的值。 | multimap.get(key).clear(); Iterables.addAll(multimap.get(key), values) |

**Multimap的视图**

Multimap还支持若干强大的视图：

* [asMap](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimap.html#asMap())为Multimap<K, V>提供Map<K,Collection<V>>形式的视图。返回的Map支持remove操作，并且会反映到底层的Multimap，但它不支持put或putAll操作。更重要的是，如果你想为Multimap中没有的键返回null，而不是一个新的、可写的空集合，你就可以使用asMap().get(key)。（你可以并且应当把asMap.get(key)返回的结果转化为适当的集合类型——如SetMultimap.asMap.get(key)的结果转为Set，ListMultimap.asMap.get(key)的结果转为List——Java类型系统不允许ListMultimap直接为asMap.get(key)返回List——*译者注：也可以用Multimaps中的asMap静态方法帮你完成类型转换*）
* [entries](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimap.html#entries())用Collection<Map.Entry<K, V>>返回Multimap中所有”键-单个值映射”——包括重复键。（对SetMultimap，返回的是Set）
* [keySet](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimap.html#keySet())用Set表示Multimap中所有不同的键。
* [keys](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimap.html#keys())用Multiset表示Multimap中的所有键，每个键重复出现的次数等于它映射的值的个数。可以从这个Multiset中移除元素，但不能做添加操作；移除操作会反映到底层的Multimap。
* [values()](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Multimap.html#values())用一个”扁平”的Collection<V>包含Multimap中的所有值。这有一点类似于Iterables.concat(multimap.asMap().values())，但它直接返回了单个Collection，而不像multimap.asMap().values()那样是按键区分开的Collection。

**Multimap不是Map**

Multimap<K, V>不是Map<K,Collection<V>>，虽然某些Multimap实现中可能使用了map。它们之间的显著区别包括：

* Multimap.get(key)总是返回非null、但是可能空的集合。这并不意味着Multimap为相应的键花费内存创建了集合，而只是提供一个集合视图方便你为键增加映射值——*译者注：如果有这样的键，返回的集合只是包装了Multimap中已有的集合；如果没有这样的键，返回的空集合也只是持有Multimap引用的栈对象，让你可以用来操作底层的Multimap。因此，返回的集合不会占据太多内存，数据实际上还是存放在Multimap中。*
* 如果你更喜欢像Map那样，为Multimap中没有的键返回null，请使用asMap()视图获取一个Map<K, Collection<V>>。（或者用静态方法[Multimaps.asMap()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimaps.html#asMap%28com.google.common.collect.ListMultimap%29)为ListMultimap返回一个Map<K, List<V>>。对于SetMultimap和SortedSetMultimap，也有类似的静态方法存在）
* 当且仅当有值映射到键时，Multimap.containsKey(key)才会返回true。尤其需要注意的是，如果键k之前映射过一个或多个值，但它们都被移除后，Multimap.containsKey(key)会返回false。
* Multimap.entries()返回Multimap中所有”键-单个值映射”——包括重复键。如果你想要得到所有”键-值集合映射”，请使用asMap().entrySet()。
* Multimap.size()返回所有”键-单个值映射”的个数，而非不同键的个数。要得到不同键的个数，请改用Multimap.keySet().size()。

**Multimap的各种实现**

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](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/ArrayListMultimap.html) | HashMap | ArrayList |
| [HashMultimap](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/HashMultimap.html) | HashMap | HashSet |
| [LinkedListMultimap](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/LinkedListMultimap.html)\* | LinkedHashMap\* | LinkedList\* |
| [LinkedHashMultimap](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/LinkedHashMultimap.html)\*\* | LinkedHashMap | LinkedHashMap |
| [TreeMultimap](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/TreeMultimap.html) | TreeMap | TreeSet |
| [ImmutableListMultimap](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/ImmutableListMultimap.html) | ImmutableMap | ImmutableList |
| [ImmutableSetMultimap](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/ImmutableSetMultimap.html) | ImmutableMap | ImmutableSet |

除了两个不可变形式的实现，其他所有实现都支持null键和null值

\*LinkedListMultimap.entries()保留了所有键和值的迭代顺序。详情见doc链接。

\*\*LinkedHashMultimap保留了映射项的插入顺序，包括键插入的顺序，以及键映射的所有值的插入顺序。

请注意，并非所有的Multimap都和上面列出的一样，使用Map<K, Collection<V>>来实现（特别是，一些Multimap实现用了自定义的hashTable，以最小化开销）

如果你想要更大的定制化，请用[Multimaps.newMultimap(Map, Supplier<Collection>)](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimaps.html" \l "newMultimap(java.util.Map,%20com.google.common.base.Supplier))或[list](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimaps.html" \l "newListMultimap(java.util.Map,%20com.google.common.base.Supplier))和 [set](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimaps.html" \l "newSetMultimap(java.util.Map,%20com.google.common.base.Supplier))版本，使用自定义的Collection、List或Set实现Multimap。

**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，会变得非常混乱。例如：

|  |  |
| --- | --- |
| 1 | Map<String, Integer> nameToId = Maps.newHashMap(); |
| 2 | Map<Integer, String> idToName = Maps.newHashMap(); |

|  |  |
| --- | --- |
| 3 |  |
| 4 | nameToId.put("Bob", 42); | |

|  |  |
| --- | --- |
| 5 | idToName.put(42, "Bob"); |
| 6 | //如果"Bob"和42已经在map中了，会发生什么? | |

|  |  |
| --- | --- |
| 7 | //如果我们忘了同步两个map，会有诡异的bug发生... |

[BiMap<K, V>](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/BiMap.html)是特殊的Map：

* 可以用 [inverse()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/BiMap.html" \l "inverse())反转BiMap<K, V>的键值映射
* 保证值是唯一的，因此 [values()](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/BiMap.html" \l "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 [BiMap.forcePut(key, value)](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/BiMap.html" \l "forcePut(java.lang.Object,java.lang.Object))。

|  |  |  |
| --- | --- | --- |
| 1 | BiMap<String, Integer> userId = HashBiMap.create(); | |
| 2 | ... |

|  |  |
| --- | --- |
| 3 |  |
| 4 | 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](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/HashBiMap.html) |
| ImmutableMap | ImmutableMap | [ImmutableBiMap](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/ImmutableBiMap.html) |
| EnumMap | EnumMap | [EnumBiMap](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/EnumBiMap.html) |
| EnumMap | HashMap | [EnumHashBiMap](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/EnumHashBiMap.html) |

注：[Maps](https://code.google.com/p/guava-libraries/wiki/CollectionUtilitiesExplained" \l "Maps)类中还有一些诸如synchronizedBiMap的BiMap工具方法.

**Table**

|  |  |  |
| --- | --- | --- |
| 1 | Table<Vertex, Vertex, Double> weightedGraph = HashBasedTable.create(); | |
| 2 | weightedGraph.put(v1, v2, 4); |

|  |  |  |
| --- | --- | --- |
| 3 | weightedGraph.put(v1, v3, 20); | |
| 4 | weightedGraph.put(v2, v3, 5); |

|  |  |
| --- | --- |
| 5 |  |
| 6 | weightedGraph.row(v1); // returns a Map mapping v2 to 4, v3 to 20 | |

|  |  |
| --- | --- |
| 7 | weightedGraph.column(v3); // returns a Map mapping v1 to 20, v2 to 5 |

通常来说，当你想使用多个键做索引的时候，你可能会用类似Map<FirstName, Map<LastName, Person>>的实现，这种方式很丑陋，使用上也不友好。Guava为此提供了新集合类型[Table](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Table.html)，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()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Table.html" \l "rowMap())：用Map<R, Map<C, V>>表现Table<R, C, V>。同样的， [rowKeySet()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Table.html" \l "rowKeySet())返回”行”的集合Set<R>。
* [row(r)](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Table.html" \l "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()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Table.html" \l "columnMap())、[columnKeySet()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Table.html" \l "columnKeySet())、[column(c)](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Table.html" \l "column(C))。（基于列的访问会比基于的行访问稍微低效点）
* [cellSet()](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Table.html" \l "cellSet())：用元素类型为[Table.Cell<R, C, V>](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Table.Cell.html)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](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/HashBasedTable.html)：本质上用HashMap<R, HashMap<C, V>>实现；
* [TreeBasedTable](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/TreeBasedTable.html)：本质上用TreeMap<R, TreeMap<C,V>>实现；
* [ImmutableTable](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/ImmutableTable.html)：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](http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/ArrayTable.html)：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与其他Table的工作原理有点不同，请参见Javadoc了解详情。

**ClassToInstanceMap**

[ClassToInstanceMap](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/ClassToInstanceMap.html)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>)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/ClassToInstanceMap.html" \l "getInstance(java.lang.Class)) 和[T putInstance(Class<T>, T)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/ClassToInstanceMap.html" \l "putInstance(java.lang.Class,java.lang.Object))，从而避免强制类型转换，同时保证了类型安全。

ClassToInstanceMap有唯一的泛型参数，通常称为B，代表Map支持的所有类型的上界。例如：

|  |  |  |
| --- | --- | --- |
| 1 | ClassToInstanceMap<Number> numberDefaults=MutableClassToInstanceMap.create(); | |
| 2 | numberDefaults.putInstance(Integer.class, Integer.valueOf(0)); |

从技术上讲，ClassToInstanceMap<B>实现了Map<Class<? extends B>, B>——或者换句话说，是一个映射B的子类型到对应实例的Map。这让ClassToInstanceMap包含的泛型声明有点令人困惑，但请记住B始终是Map所支持类型的上界——通常B就是Object。

对于ClassToInstanceMap，Guava提供了两种有用的实现：[MutableClassToInstanceMap](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/MutableClassToInstanceMap.html)和 [ImmutableClassToInstanceMap](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/ImmutableClassToInstanceMap.html)。

**RangeSet**

RangeSet描述了一组不相连的、非空的区间。当把一个区间添加到可变的RangeSet时，所有相连的区间会被合并，空区间会被忽略。例如：

|  |  |  |
| --- | --- | --- |
| 1 | RangeSet<Integer> rangeSet = TreeRangeSet.create(); | |
| 2 | rangeSet.add(Range.closed(1, 10)); // {[1,10]} |

|  |  |
| --- | --- |
| 3 | rangeSet.add(Range.closedOpen(11, 15));//不相连区间:{[1,10], [11,15)} |
| 4 | rangeSet.add(Range.closedOpen(15, 20)); //相连区间; {[1,10], [11,20)} |

|  |  |
| --- | --- |
| 5 | rangeSet.add(Range.openClosed(0, 0)); //空区间; {[1,10], [11,20)} |
| 6 | rangeSet.remove(Range.open(5, 10)); //分割[1, 10]; {[1,5], [10,10], [11,20)} | |

请注意，要合并Range.closed(1, 10)和Range.closedOpen(11, 15)这样的区间，你需要首先用[Range.canonical(DiscreteDomain)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Range.html" \l "canonical(com.google.common.collect.DiscreteDomain))对区间进行预处理，例如DiscreteDomain.integers()。

注：RangeSet不支持GWT，也不支持JDK5和更早版本；因为，RangeSet需要充分利用JDK6中NavigableMap的特性。

|  |  |  |
| --- | --- | --- |
| **Guava Range 概念，范围和方法** | | |
| **概念** | **表示范围** | **guava对应功能方法** |
| (a..b) | {x | a < x < b} | open(C, C) |
| [a..b] | {x | a <= x <= b} | closed(C, C) |
| [a..b) | {x | a <= x < b} | closedOpen(C, C) |
| (a..b] | {x | a < x <= b} | openClosed(C, C) |
| (a..+∞) | {x | x > a} | greaterThan(C) |
| [a..+∞) | {x | x >= a} | atLeast(C) |
| (-∞..b) | {x | x < b} | lessThan(C) |
| (-∞..b] | {x | x <= b} | atMost(C) |
| (-∞..+∞) | all values | all() |

**RangeSet的视图**

RangeSet的实现支持非常广泛的视图：

* complement()：返回RangeSet的补集视图。complement也是RangeSet类型,包含了不相连的、非空的区间。
* subRangeSet(Range<C>)：返回RangeSet与给定Range的交集视图。这扩展了传统排序集合中的headSet、subSet和tailSet操作。
* asRanges()：用Set<Range<C>>表现RangeSet，这样可以遍历其中的Range。
* asSet(DiscreteDomain<C>)（仅ImmutableRangeSet支持）：用ImmutableSortedSet<C>表现RangeSet，以区间中所有元素的形式而不是区间本身的形式查看。（这个操作不支持DiscreteDomain 和RangeSet都没有上边界，或都没有下边界的情况）

**RangeSet的查询方法**

为了方便操作，RangeSet直接提供了若干查询方法，其中最突出的有:

* contains(C)：RangeSet最基本的操作，判断RangeSet中是否有任何区间包含给定元素。
* rangeContaining(C)：返回包含给定元素的区间；若没有这样的区间，则返回null。
* encloses(Range<C>)：简单明了，判断RangeSet中是否有任何区间包括给定区间。
* span()：返回包括RangeSet中所有区间的最小区间。

**RangeMap**

RangeMap描述了”不相交的、非空的区间”到特定值的映射。和RangeSet不同，RangeMap不会合并相邻的映射，即便相邻的区间映射到相同的值。例如：

|  |  |
| --- | --- |
| 1 | RangeMap<Integer, String> rangeMap = TreeRangeMap.create(); |
| 2 | rangeMap.put(Range.closed(1, 10), "foo"); //{[1,10] => "foo"} | |

|  |  |
| --- | --- |
| 3 | rangeMap.put(Range.open(3, 6), "bar"); //{[1,3] => "foo", (3,6) => "bar", [6,10] => "foo"} |
| 4 | rangeMap.put(Range.open(10, 20), "foo"); //{[1,3] => "foo", (3,6) => "bar", [6,10] => "foo", (10,20) => "foo"} |

|  |  |
| --- | --- |
| 5 | rangeMap.remove(Range.closed(5, 11)); //{[1,3] => "foo", (3,5) => "bar", (11,20) => "foo"} |

**RangeMap的视图**

RangeMap提供两个视图：

* asMapOfRanges()：用Map<Range<K>, V>表现RangeMap。这可以用来遍历RangeMap。
* subRangeMap(Range<K>)：用RangeMap类型返回RangeMap与给定Range的交集视图。这扩展了传统的headMap、subMap和tailMap操作。