# Guava的基本使用

说明：以下例子依赖于JDK1.8 和 Guava 27.1-jre (2683KB)

1. **<dependency>**
2. **<groupId>**com.google.guava**</groupId>**
3. **<artifactId>**guava**</artifactId>**
4. **<version>**27.1-jre**</version>**
5. **</dependency>**

## 基本工具（Basic utilities）

### 1.1 使用和避免null

#### 1.1.1 说明

null是模棱两可的，会引起令人困惑的错误。很多Guava工具类用快速失败拒绝null值，而不是盲目的接受。

Guava用[Optional<T>](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/base/Optional.html)表示可能为null的T类型引用，它从不说包含的是null值，而是用存在或缺失来表示。Optional不会包含null值引用。

#### 1.1.2 Optional使用

1. **public** **static** **void** main(String[] args) {
2. //Optional.of(T) 创建制定引用的Optional实例，如果引用为null则快速失败
3. Optional<Integer> possible = Optional.of(2);
4. // 创建引用缺失的Optional实例
5. Optional<Object> absent = Optional.absent();
6. // 创建制定引用的Optional实例，若引用为null则表示缺失
7. Optional<Object> nullable = Optional.fromNullable(**null**);
9. //如果Optional包含引用实例，返回true
10. **boolean** present = possible.isPresent();
11. //返回Optional所包含的引用，若引用缺失，则抛出IllegalStateException异常
12. Integer get = possible.get();
13. //返回Optional所包含的引用，若引用缺失，则返回指定的值
14. Integer or = possible.or(10);
15. //返回Optional所包含的引用，若引用缺失，则返回null
16. Object o = nullable.orNull();
18. //true
19. System.out.println(possible.isPresent());
20. //false
21. System.out.println(absent.isPresent());
22. //false
23. System.out.println(nullable.isPresent());
24. //2
25. System.out.println(get);
26. //2
27. System.out.println(or);
28. //null
29. System.out.println(o);
30. }

### 1.2 前置条件

#### 1.2.1 说明

让方法调用的前置条件判断更简单。Preconditions类中提供了若干前置条件判断的实用方法。

|  |  |  |
| --- | --- | --- |
| 方法声明 | 描述 | 检查失败抛出异常 |
| checkArgument(boolean) | 检查boolean是否为true，用来检查传递给方法的参数 | IllegalArgumentException |
| checkNotNull(T) | 检查value是否为null，该方法直接返回value，因此可以内嵌使用checkNotNull。 | NullPointerException |
| checkState(boolean) | 用来检查对象的某些状态。 | IllegalStateException |
| checkElementIndex(int index, int size) | 检查index作为索引值对某个列表、字符串或数组是否有效。index>=0 && index<size \* | IndexOutOfBoundsException |
| checkPositionIndex(int index, int size) | 检查index作为位置值对某个列表、字符串或数组是否有效。index>=0 && index<=size \* | IndexOutOfBoundsException |
| checkPositionIndexes(int start, int end, int size) | 检查[start, end]表示的位置范围对某个列表、字符串或数组是否有效\* | IndexOutOfBoundsException |

以上每个方法都有三种变种：

1. 没有额外参数：抛出的异常中没有错误信息
2. 有一个Object对象参数：抛出的异常实用Object.toString()作为错误消息
3. 有一个String对象，并且有一组任意数量的附件Object对象：处理错误消息类似printf，但是只支持%s指示符。

#### 1.2.2 例子

1. **public** **static** **void** main(String[] args) {
2. //java.lang.IllegalArgumentException: 抛出异常
3. Preconditions.checkArgument(**false**,"抛出异常");
4. //java.lang.NullPointerException: object(null) is null
5. Preconditions.checkNotNull(**null**,"%s is null","object(null)");
6. }

#### 1.2.3 注

1. 索引值常用来查找列表、字符串或数组中的元素，如List.get(int), String.charAt(int)

2. 位置值和位置范围常用来截取列表、字符串或数组，如List.subList(int，int), String.substring(int)

### 1.3 常见Object方法

#### 1.3.1 equals

当一个对象中的字段可以为null时，实现Object.equals方法会很痛苦，因为不得不分别对它们进行null检查，使用Objects.equal可以帮助执行null敏感的equals判断，从而避免NPE

1. **public** **static** **void** main(String[] args) {
2. //true
3. **boolean** e = Objects.equal("a", "a");
4. //false
5. **boolean** e1 = Objects.equal(**null**, "a");
6. //false
7. **boolean** e2 = Objects.equal("a", **null**);
8. //true
9. **boolean** e3 = Objects.equal(**null**, **null**);
10. }

注：JDK7引入的Objects类提供了一样的方法Objects.equals

#### 1.3.2 hashCode

用对象的所有字段作散列运算应当更简单，Objects.hashCode会对传入的字段序列计算出合理的、顺序敏感的散列值。

方法为：Objects.hashCode(Object…) 返回int类型的散列值

注：JDK7引入的Objects类提供了一样的方法Objects.hash(Object…)

#### 1.3.3 toString

好的toString方法在调试的时候会有很大帮助，但是编写toString方法有时候确实痛苦，

1. **public** **class** MoreObjectsDemo {
2. **private** **int** a;
4. @Override
5. **public** String toString() {
6. **return** MoreObjects.toStringHelper(**this**)
7. .add("a",**this**.a)
8. .add("x", 1).toString();
9. }
11. **public** **static** **void** main(String[] args) {
12. //MoreObjectsDemo{a=0, x=1}
13. System.out.println(**new** MoreObjectsDemo());
14. }
15. }

### 1.4 排序

排序器(Ordering)是Guava流畅风格比较器的实现，它可以用来构建复杂的比较器，以完成集合排序的功能。从实现上说，Ordering实例就是一个复杂的Comparator实例，把很多基于Comparator的静态方法包装为自己的实例方法，并提供链式调用方法。

#### 1.4.1 创建排序器

|  |  |
| --- | --- |
| 方法 | 描述 |
| natural() | 对可排序类型做自然排序，如数字按大小，日期按先后排序 |
| usingToString() | 按对象的字符串形式做字典排序[lexicographical ordering] |
| from(Comparator) | 把给定的Comparator转化为排序器 |

举个例子：

1. **public** **static** **void** main(String[] args) {
2. List<String> list = Arrays.asList("a", "f", "e", "dfadf");
3. //[a, f, e, dfadf]
4. System.out.println(list);
5. list.sort(Ordering.<String>natural());
6. //[a, dfadf, e, f]
7. System.out.println(list);
8. }

实现西定义的排序器时，除了用上面的from方法，也可以跳过实现Comparator，直接继承Ordering，这是因为：public abstract class Ordering<T> implements Comparator<T>

举个例子：

1. List<String> aList = Arrays.asList("aaa", "bbbb", "ccccc", "d");
2. Ordering<String> ordering = **new** Ordering<String>() {
3. @Override
4. **public** **int** compare(@Nullable String s, @Nullable String t1) {
5. **return** Ints.compare(s.length(), t1.length());
6. }
7. };
8. aList.sort(ordering);
9. // [d, aaa, bbbb, ccccc]
10. System.out.println(aList);

#### 1.4.2 链式调用方法

通过链式调用，可以由给定的排序器衍生出其他排序器。方法如下：

|  |  |
| --- | --- |
| 方法 | 描述 |
| reverse() | 获取语义相反的排序器 |
| nullsFirst() | 使用当前排序器，但额外把null值排到最前面 |
| nullsLast() | 使用当前排序器，但额外把null值排到最后面 |
| compound(Comparator) | 合成另一个比较器，以处理当前排序器中的相等情况 |
| lexicographical() | 基于处理类型T的排序器，返回该类型的可迭代对象Iterable<T>的排序器 |
| onResultOf(Function) | 对集合中元素调用Function，再按返回值用当前排序器排序 |

举个栗子

1. **public** **class** Foo {
2. @Nullable
3. String sortedBy;
4. **int** notSortedBy;
6. **public** Foo(@Nullable String sortedBy, **int** notSortedBy) {
7. **this**.sortedBy = sortedBy;
8. **this**.notSortedBy = notSortedBy;
9. }
11. @Override
12. **public** String toString() {
13. **return** "Foo{" +
14. "sortedBy='" + sortedBy + '\'' +
15. ", notSortedBy=" + notSortedBy +
16. '}';
17. }
18. }

需要Foo类的排序器，考虑到能处理sortedBy为null的情况，可以使用下面链式调用来合成排序器

1. //链式调用
2. Ordering<Foo> fooOrdering = Ordering.natural().nullsFirst().onResultOf(**new** Function<Foo, String>() {
3. @Nullable
4. @Override
5. **public** String apply(@Nullable Foo foo) {
6. **return** foo.sortedBy;
7. }
8. });
9. List<Foo> foos = Arrays.asList(**new** Foo(**null**, 2), **new** Foo("ad", 3), **new** Foo("hsg", 1));
10. foos.sort(fooOrdering);
11. //[Foo{sortedBy='null', notSortedBy=2}, Foo{sortedBy='ad', notSortedBy=3}, Foo{sortedBy='hsg', notSortedBy=1}]
12. System.out.println(foos);

当阅读链式调用产生的排序器时，应该从后往前读。上面的例子中，排序器首先调用apply方法获取sortedBy值，并把sortedBy为null的元素都放到最前面，然后把剩下的元素按sortedBy进行自然排序。之所以要从后往前读，是因为每次链式调用都是用后面的方法包装了前面的排序器。

**注：**用compound方法包装排序器时，就不应遵循从后往前读的原则。为了避免理解上的混乱，请不要把compound写在一长串链式调用的中间，你可以另起一行，在链中最先或最后调用compound。

#### 1.4.3 运用排序器

Guava的排序器实现有若干操纵集合或元素值的方法

|  |  |
| --- | --- |
| 方法 | 描述 |
| greatestOf(Iterable iterable, int k) | 获取可迭代对象中最大的k个元素 |
| isOrdered(Iterable) | 判断可迭代对象是否已按排序器排序：允许有排序值相等的元素。 |
| sortedCopy(Iterable) | 判断可迭代对象是否已严格按排序器排序：不允许排序值相等的元素。 |
| min(E, E) | 返回两个参数中最小的那个。如果相等，则返回第一个参数。 |
| min(E, E, E, E...) | 返回多个参数中最小的那个。如果有超过一个参数都最小，则返回第一个最小的参数。 |
| min(Iterable) | 返回迭代器中最小的元素。如果可迭代对象中没有元素，则抛出NoSuchElementException |

举个例子：

1. List<Integer> bList = Arrays.asList(10, 3, 4, 23, 4, 9, 11, 0);
2. Ordering<Integer> ordering1 = **new** Ordering<Integer>() {
3. @Override
4. **public** **int** compare(@Nullable Integer integer, @Nullable Integer t1) {
5. **return** integer.compareTo(t1);
6. }
7. };
8. Integer max = ordering1.max(bList);
9. List<Integer> maxs = ordering1.greatestOf(bList, 2);
10. //23
11. System.out.println(max);
12. //[23, 11]
13. System.out.println(maxs);

### 1.5 Throwables

有时候，想把捕获的异常再次抛出。这种情况通常发生在Error和RuntimeException被捕获的时候，本意并不是捕获他们，但是声明捕获Throwable和Exception的时候，也包括了Error和RuntimeException。可以使用Throwables的若干方法来判断异常类型，并且重新传播异常。

#### 1.5.1 异常传播

Guava中的异常传播方法简要列举如下：

|  |  |
| --- | --- |
| void throwIfInstanceOf(Throwable throwable, Class<X> declaredType) throws X | Throwable类型为X才抛出 |
| void throwIfUnchecked(Throwable throwable) | 类型为未检异常（Error或RuntimeException）才抛出 |
| void propagateIfPossible( Throwable, Class<X extends Throwable>) throws X | Throwable类型为X, Error或RuntimeException才抛出 |

所有这些方法都会自己决定是否要抛出异常

**举个栗子：**

1. **try**{
2. **throw** **new** OutOfMemoryError();
3. }**catch** (Throwable throwable){
4. //异常的传播
5. //抛出异常，当throwabel 为declaredType类型的实例
6. Throwables.throwIfInstanceOf(throwable,NullPointerException.**class**);
8. //here will throw java.lang.OutOfMemoryError
9. Throwables.throwIfUnchecked(throwable);
11. // Throwables.propagateIfPossible(throwable,ArrayIndexOutOfBoundsException.class);
12. }

#### 1.5.2 异常原因链

1. **try** {
2. **throw** **new** NullPointerException();
3. }**catch** (Throwable t){
4. //异常原因链
6. Throwable rootCause = Throwables.getRootCause(t);
7. //null
8. System.out.println(rootCause.getCause());
10. List<Throwable> causalChain = Throwables.getCausalChain(t);
11. //[java.lang.NullPointerException]
12. System.out.println(causalChain);
14. String stackTraceAsString = Throwables.getStackTraceAsString(t);
15. //java.lang.NullPointerException
16. System.out.println(stackTraceAsString);
17. }

## 集合（Collections）

Guava对JDK集合的扩展，这是Guava最成熟和为人所知的部分。

### 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；
* 不可变对象因为有固定不变，可以作为常量来安全使用。

创建对象的不可变拷贝是一项很好的防御性编程技巧。Guava为所有JDK标准集合类型和Guava新集合类型都提供了简单易用的不可变版本。  
 JDK也提供了Collections.unmodifiableXXX方法把集合包装为不可变形式，但我们认为不够好：

* 笨重而且累赘：不能舒适地用在所有想做防御性拷贝的场景；
* 不安全：要保证没人通过原集合的引用进行修改，返回的集合才是事实上不可变的；
* 低效：包装过的集合仍然保有可变集合的开销，比如并发修改的检查、散列表的额外空间，等等。

如果没有修改某个集合的需求，或者希望某个集合保持不变时，把它防御性地拷贝到不可变集合是个很好的实践。

重要提示：所有Guava不可变集合的实现都不接受null值。我们对Google内部的代码库做过详细研究，发现只有5%的情况需要在集合中允许null元素，剩下的95%场景都是遇到null值就快速失败。如果需要在不可变集合中使用null，请使用JDK中的Collections.unmodifiableXXX方法。

#### 如何创建不可变集合

1. //方式一：copyOf方法。如ImmutableSet.copyOf(set)
2. Set<String> set = **new** HashSet<>();
3. set.add("red");
4. set.add("orange");
5. set.add("yellow");
6. set.add("green");
7. ImmutableSet<String> strings = ImmutableSet.copyOf(set);
8. //方式二：of方法
9. ImmutableSet<String> of = ImmutableSet.of("red", "orange", "yellow", "green", "blue", "purple");
10. //方式三：Builder工具
11. ImmutableSet<String> build = ImmutableSet.<String>builder().add("red").add("orange").add("yellow").add("green").build();

对有序不可变集合来说，排序是在构造集合的时候完成的

1. ImmutableSortedSet<String> sortedSet = ImmutableSortedSet.of("a", "b", "c", "a", "d", "b");
2. //[a, b, c, d]
3. System.out.println(sortedSet);

#### asList视图

所有不可变集合都有一个asList()方法提供ImmutableList视图，来帮助你用列表形式方便地读取集合元素。asList()返回的ImmutableList通常是（并不总是）开销稳定的视图实现，而不是简单地把元素拷贝进List。也就是说，asList返回的列表视图通常比一般的列表平均性能更好，比如，在底层集合支持的情况下，它总是使用高效的contains方法。

1. ImmutableSortedSet<String> sortedSet = ImmutableSortedSet.of("a", "b", "c", "a", "d", "b");
3. ImmutableList<String> list = sortedSet.asList();
4. //[a, b, c, d]
5. System.out.println(list);

#### 关联可变集合和不可变集合

|  |  |  |
| --- | --- | --- |
| 可变集合接口 | 属于JDK还是Guava | 不可变版本 |
| 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 |
| ListMultimap | Guava | ImmutableListMultimap |
| SetMultimap | Guava | ImmutableSetMultimap |
| BiMap | Guava | ImmutableBiMap |
| ClassToInstanceMap | Guava | ImmutableClassToInstanceMap |
| Table | Guava | ImmutableTable |

### 新集合类型

#### Multiset

Multiset可以多次添加相等的元素。可以用两种方式看待Multiset：

1. 没有顺序限制的Arraylist<E>
2. Map<E，Integer>，键为元素，数为计数。

Multiset API也结合考虑着两种方式：

1. 当把Multiset 看成普通的Collection时，它表现得就像无序的ArrayList：2.3 强大的集合工具类

* 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）

1. 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。

Multiset的一些常见方法

|  |  |
| --- | --- |
| 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支持getElement()和getCount()方法 |
| 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 |
| size() | 返回集合元素的总个数（包括重复的元素） |

举个例子： 单词统计出现次数

1. List<String> list = Arrays.asList("a", "b", "c", "a", "b", "a", "c", "d");
2. //传统方式
3. Map<String, Integer> counts = **new** HashMap<>(16);
4. **for** (String word: list) {
5. Integer count = counts.get(word);
6. **if** (count == **null**) {
7. counts.put(word,1);
8. }**else** {
9. counts.put(word,count +1);
10. }
11. // jdk1.8 可以写成这样  counts.merge(word, 1, (a, b) -> a + b);
12. }
13. // 3
14. System.out.println(counts.get("a"));
16. //multiset
17. HashMultiset<String> strings1 = HashMultiset.create(list);
18. // 3
19. System.out.println(strings1.count("a"));

但是，Multiset不是Map。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。还有一些其他的区别：

* 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。

Guava提供了多种Multiset的实现，如下表格：

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

#### Multimap

写程序一般都遇到过Map<K，List<V>> 或者 Map<K，Set<V>>这样的机构，Multimap可以很容易的把一个键映射到多个值。也就是说，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：

1.“键-单个值映射”的集合：a -> 1 a -> 2 a ->4 b -> 3 c -> 5

2. “键-值集合映射”的集合：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。但是asMap()视图返回Map<K, Collection<V>>，可以按另一种方式看待Multimap。不会有任何键映射到空集合：一个键至少有一个值，要么根本不在Multimap中。

Multimap的一些方法：对值视图集合进行的修改最终都会反映到底层的Multimap

|  |  |  |
| --- | --- | --- |
| 方法签名 | 描述 | 等价于 |
| get(key) | 以集合形式返回键所对应的值视图，没有对应的值，返回空集合 |  |
| put(K, V) | 添加键到单个值的映射 | multimap.get(key).add(value) |
| putAll(K, Iterable<V>) | 依次添加键到多个值的映射 | Iterables.addAll(multimap.get(key), values) |
| 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true。 | multimap.get(key).remove(value) |
| removeAll(K) | 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>) | 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) |

举个栗子：

1. ArrayListMultimap<String, Integer> multimap = ArrayListMultimap.create();
2. multimap.put("a",1);
3. multimap.put("a",2);
4. multimap.put("a",4);
5. multimap.put("a",4);
6. multimap.put("b",3);
7. multimap.put("c",5);
8. List<Integer> a = multimap.get("a");
9. //[1, 2, 4, 4]
10. System.out.println(a);
11. a.remove(1);
12. //{a=[1, 4, 4], b=[3], c=[5]}
13. System.out.println(multimap);
14. //[3]
15. System.out.println(multimap.get("b"));
16. //[]
17. System.out.println(multimap.get("d"));
18. List<Integer> c = multimap.replaceValues("c", Arrays.asList(7, 8, 9));
19. //[5]
20. System.out.println(c);
21. //[7, 8, 9]
22. System.out.println(multimap.get("c"));

**Multimap的视图：**

* 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——(也可以用Multimaps中的asMap静态方法帮你完成类型转换）
* 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用Set表示Multimap中所有不同的键。
* 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()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。它们之间还有显著的区别如下：

* 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——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()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：**

|  |  |  |
| --- | --- | --- |
| **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 |

**注：**

* 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。

#### BiMap

传统上，实现键值对的双向映射需要维护两个单独的Map，并且保持它们的同步。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。

BiMap<K,V>是一个特殊的Map:

* 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 BiMap.forcePut(key, value)。

举个栗子：

1. HashMap<String, Integer> hashMap = **new** HashMap<>();
2. hashMap.put("Joy",1);
3. hashMap.put("Bob",12);
4. hashMap.put("Tony",2);
5. hashMap.put("Jack",21);
6. HashBiMap<String, Integer> user = HashBiMap.create();
7. user.putAll(hashMap);
8. //java.lang.IllegalArgumentException: value already present: 1
9. //Integer bob = user.put("Bob", 1);
10. //forcePut 强制更换已经存在的值，会把原来存在的键-值删除，
11. Integer bob = user.forcePut("Bob", 1);
12. //12
13. System.out.println(bob);
14. String s = user.inverse().get(1);
15. //Bob
16. System.out.println(s);
17. //{Tony=2, Bob=1, Jack=21}
18. System.out.println(user);

**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

通常来说，如果想用多个值做索引的时候，可能会用类似Map<K，Map<T，V>>的实现，这种方式在使用上并不友好，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。

**举个例子：**比如要实现存储一下的表格的数据。

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

1. Table<Integer, Integer, String> table = HashBasedTable.create();
2. table.put(0,0,"1");
3. table.put(0,1,"2");
4. table.put(0,2,"3");
5. table.put(1,0,"4");
6. table.put(1,1,"5");
7. table.put(1,2,"6");
8. table.put(2,0,"7");
9. table.put(2,1,"8");
10. table.put(2,2,"9");
12. Map<Integer, Map<Integer, String>> integerMapMap = table.rowMap();
13. //{0={0=1, 1=2, 2=3}, 1={0=4, 1=5, 2=6}, 2={0=7, 1=8, 2=9}}
14. System.out.println(integerMapMap);
15. Map<Integer, String> row = table.row(1);
16. //{0=4, 1=5, 2=6}
17. System.out.println(row);
18. Map<Integer, String> column = table.column(1);
19. //{0=2, 1=5, 2=8}
20. System.out.println(column);
21. Set<Table.Cell<Integer, Integer, String>> cells = table.cellSet();
22. //[(0,0)=1, (0,1)=2, (0,2)=3, (1,0)=4, (1,1)=5, (1,2)=6, (2,0)=7, (2,1)=8, (2,2)=9]
23. System.out.println(cells);

**关于Table的几种实现：**

* 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与其他Table的工作原理有点不同，

#### 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换，同时保证了类型安全。

从技术上来说，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。

**举个栗子：**

1. MutableClassToInstanceMap<Number> number = MutableClassToInstanceMap.create();
2. number.putInstance(Integer.**class**,0);
3. number.putInstance(Integer.**class**,3);
4. number.putInstance(Long.**class**,2L);
5. Integer instance = number.getInstance(Integer.**class**);
6. //3
7. System.out.println(instance);

**ClassToInstanceMap的两种实现：**

* MutableClassToInstanceMap
* ImmutableClassToInstanceMap。

#### 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。

**RangeSet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。

1. TreeRangeSet<Integer> rangeSet = TreeRangeSet.create();
2. //添加区间 [1,10]
3. rangeSet.add(Range.closed(1,10));
4. //[[1..10]]
5. System.out.println(rangeSet);
6. //添加区间 [11,15)
7. rangeSet.add(Range.closedOpen(11,15));
8. //[[1..10], [11..15)]
9. System.out.println(rangeSet);
10. //添加区间 [15,20)
11. rangeSet.add(Range.closedOpen(15,20));
12. //[[1..10], [11..20)]
13. System.out.println(rangeSet);
14. //添加空区间 （0,0]
15. rangeSet.add(Range.openClosed(0,0));
16. //[[1..10], [11..20)]
17. System.out.println(rangeSet);
18. //移除区间 (5,10)
19. rangeSet.remove(Range.open(5,10));
20. //[[1..5], [10..10], [11..20)]
21. System.out.println(rangeSet);
23. //返回RangeSet的补集视图
24. RangeSet<Integer> complement = rangeSet.complement();
25. //[(-∞..1), (5..10), (10..11), [20..+∞)]
26. System.out.println(complement);
27. Set<Range<Integer>> ranges = rangeSet.asRanges();
28. System.out.println(ranges);
29. //是否有任何区间包含给定元素
30. **boolean** contains = rangeSet.contains(6);
31. //false
32. System.out.println(contains);
33. //返回包括RangeSet中所有区间的最小区间
34. Range<Integer> span = rangeSet.span();
35. //[1..20)
36. System.out.println(span);

#### 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。

**举个栗子吧：**

1. TreeRangeMap<Integer, String> rangeMap = TreeRangeMap.create();
2. rangeMap.put(Range.closed(1,10),"foo");
3. // [[1..10]=foo]
4. System.out.println(rangeMap);
5. rangeMap.put(Range.open(3,6),"bar");
6. // [[1..3]=foo, (3..6)=bar, [6..10]=foo]
7. System.out.println(rangeMap);
9. rangeMap.put(Range.open(10,20),"foo");
10. // [[1..3]=foo, (3..6)=bar, [6..10]=foo, (10..20)=foo]
11. System.out.println(rangeMap);
13. rangeMap.remove(Range.closed(5,11));
14. // [[1..3]=foo, (3..5)=bar, (11..20)=foo]
15. System.out.println(rangeMap);

**RangeMap的视图：**

* 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。

### 强大的集合工具类

把工具类和特定集合接口的对应关系归纳一下：

|  |  |  |
| --- | --- | --- |
| 集合接口 | JDK 还是 Guava | 对应的Guava工具类 |
| Collection | JDk | Collections2 |
| List | JDk | Lists |
| Set | JDk | Sets |
| SortedSet | JDk | Sets |
| Map | JDk | Maps |
| SortedMap | JDk | Maps |
| Queue | JDk | Queues |
| Multiset | Guava | Multisets |
| Multimap | Guava | Multimaps |
| BiMap | Guava | Maps |
| Table | Guava | Tables |

#### 静态工厂方法

Guava提供了能够推断泛型的静态工厂方法，可以方便地在初始化时就指定起始元素，以提高集合初始化大小的可读性。Guava引入的新集合类型没有暴露原始构造器，也没有在工具类中提供初始化方法。而是直接在集合类中提供了静态工厂方法。

**举个栗子：**

1. ArrayList<String> list = Lists.newArrayList();
2. LinkedHashMap<String, Integer> maps = Maps.newLinkedHashMap();
3. //初始化制定元素
4. ArrayList<String> list1 = Lists.newArrayList("a","a","f","e");
5. //初始化大小
6. ArrayList<String> list2 = Lists.newArrayListWithCapacity(100);
7. //新集合类型没有暴露原始构造器，
8. HashMultiset<Object> multiset = HashMultiset.create();

#### Iterables

很多你期望的支持所有集合的操作都在Iterables类中。大多数Iterables方法有一个在Iterators类中的对应版本，用来处理Iterator。

**常见方法：**

|  |  |
| --- | --- |
| concat(Iterable<Iterable>) | 串联多个iterables的懒视图 |
| frequency(Iterable, Object) | 返回对象在iterable中出现的次数 |
| partition(Iterable, int) | 把iterable按指定大小分割，得到的子集都不能进行修改操作 |
| getFirst(Iterable, T default) | 返回iterable的第一个元素，若iterable为空则返回默认值 |
| getLast(Iterable) | 返回iterable的最后一个元素，若iterable为空则抛出NoSuchElementException |
| elementsEqual(Iterable, Iterable) | 如果两个iterable中的所有元素相等且顺序一致，返回true |
| unmodifiableIterable(Iterable) | 返回iterable的不可变视图 |
| limit(Iterable, int) | 限制iterable的元素个数限制给定值 |
| getOnlyElement(Iterable) | 获取iterable中唯一的元素，如果iterable为空或有多个元素，则快速失败 |

**举个栗子：**

1. Iterable<Integer> concat = Iterables.concat(Ints.asList(1, 2, 3), Ints.asList(4, 6, 5));
2. // [1, 2, 3, 4, 6, 5]
3. System.out.println(concat);
4. **int** frequency = Iterables.frequency(concat, 1);
5. // 1
6. System.out.println(frequency);
7. Iterable<List<Integer>> partition = Iterables.partition(concat, 3);
8. // [[1, 2, 3], [4, 6, 5]]
9. System.out.println(partition);
10. Integer first = Iterables.getFirst(concat, 0);
11. // 1
12. System.out.println(first);
13. Integer last = Iterables.getLast(concat, 0);
14. // 5
15. System.out.println(last);
16. **boolean** b = Iterables.elementsEqual(concat, partition);
17. // false
18. System.out.println(b);
19. Iterable<Integer> integers = Iterables.unmodifiableIterable(concat);
20. // [1, 2, 3, 4, 6, 5]
21. System.out.println(integers);
22. Iterable<Integer> limit = Iterables.limit(concat, 2);
23. // [1, 2]
24. System.out.println(limit);
25. Integer onlyElement = Iterables.getOnlyElement(concat);
26. // java.lang.IllegalArgumentException: expected one element but was: <1, 2, 3, 4, 6, ...>
27. System.out.println(onlyElement);

**与Collection方法相似的工具方法:**

通常来说，Collection的实现天然支持操作其他Collection，但却不能操作Iterable。

下面的方法中，如果传入的Iterable是一个Collection实例，则实际操作将会委托给相应的Collection接口方法。例如，往Iterables.size方法传入是一个Collection实例，它不会真的遍历iterator获取大小，而是直接调用Collection.size。

|  |  |
| --- | --- |
| 方法 | 类似的Collection方法 |
| addAll(Collection addTo,Iterable toAdd) | Collection.addAll(Collection) |
| contains(Iterable, Object) | Collection.contains(Object) |
| removeAll(Iterable removeFrom, Collection toRemove) | Collection.removeAll(Collection) |
| retainAll(Iterable removeFrom, Collection toRetain) | Collection.retainAll(Collection) |
| size(Iterable) | Collection.size() |
| toArray(Iterable, Class) | Collection.toArray(T[]) |
| isEmpty(Iterable) | Collection.isEmpty() |
| get(Iterable, int) | List.get(int) |
| toString(Iterable) | Collection.toString() |

**举个栗子：**

1. ArrayList<Integer> list = Lists.newArrayList(9, 8, 7, 6, 5, 4);
2. **boolean** contains = Iterables.contains(list, 4);
3. // true
4. System.out.println(contains);
5. Integer[] values = Iterables.toArray(list, Integer.**class**);
6. // 6
7. System.out.println(values.length);

#### Lists

除了静态工厂方法和函数式编程方法，Lists为List类型的对象提供了若干工具方法

|  |  |
| --- | --- |
| 方法 | 描述 |
| partition(List, int) | 把List按指定大小分割 |
| reverse(List) | 返回给定List的反转视图 |

**举个栗子：**

1. List<Integer> values = Ints.asList(1, 2, 3, 4, 5);
2. List<Integer> reverse = Lists.reverse(values);
3. // [5, 4, 3, 2, 1]
4. System.out.println(reverse);
5. List<List<Integer>> partition = Lists.partition(values, 2);
6. // [[1, 2], [3, 4], [5]]
7. System.out.println(partition);

#### Sets

提供了很多标准的集合运算（Set-Theoretic）方法，这些方法接受Set参数并返回SetView，可用于：

* 直接当作Set使用，因为SetView也实现了Set接口；
* 用copyInto(Set)拷贝进另一个可变集合；
* 用immutableCopy()对自己做不可变拷贝。

|  |  |
| --- | --- |
| 方法 | 描述 |
| union(Set, Set) | 返回两个集合的并集 |
| intersection(Set, Set) | 返回两个集合的交集 |
| difference(Set, Set) | 返回第一个集合对第二个集合的差集 |
| symmetricDifference(Set,Set) | 返回两个集合的补集 |
| cartesianProduct(List<Set>) | 返回所有集合的笛卡儿积 |
| powerSet(Set) | 返回给定集合的所有子集 |

**举个栗子：**

1. ImmutableSet<Integer> first = ImmutableSet.of(1, 2, 3, 5);
2. ImmutableSet<Integer> second = ImmutableSet.of(4, 5, 6, 7);
3. Sets.SetView<Integer> union = Sets.union(first, second);
4. // [1, 2, 3, 5, 4, 6, 7]
5. System.out.println(union);
6. Sets.SetView<Integer> intersection = Sets.intersection(first, second);
7. // [5]
8. System.out.println(intersection);
9. Sets.SetView<Integer> difference = Sets.difference(first, second);
10. // [1, 2, 3]
11. System.out.println(difference);
12. Sets.SetView<Integer> integers = Sets.symmetricDifference(first, second);
13. // [1, 2, 3, 4, 6, 7]
14. System.out.println(integers);
15. Set<List<Integer>> lists = Sets.cartesianProduct(first, second);
16. // [[1, 4], [1, 5], [1, 6], [1, 7],...,[5, 7]]
17. System.out.println(lists);
19. Set<Set<Integer>> sets = Sets.powerSet(first);
20. // powerSet({1=0, 2=1, 3=2, 5=3})
21. System.out.println(sets);

#### Maps

**uniqueIndex:**

Maps.uniqueIndex(Iterable,Function)通常针对的场景是：有一组对象，它们在某个属性上分别有独一无二的值，而我们希望能够按照这个属性值查找对象——这个方法返回一个Map，键为Function返回的属性值，值为Iterable中相应的元素，因此我们可以反复用这个Map进行查找操作。

**举个栗子**：如果有一堆长度独一无二的字符串，希望按照特定长度查找字符串。

1. ArrayList<String> list = Lists.newArrayList("aa", "fdasf", "ccc", "dddd");
2. ImmutableMap<Integer, String> strings = Maps.uniqueIndex(list, **new** Function<String, Integer>() {
3. @Nullable
4. @Override
5. **public** Integer apply(@Nullable String s) {
6. **return** s.length();
7. }
8. });
9. //{2=aa, 5=fdasf, 3=ccc, 4=dddd}
10. System.out.println(strings);

**difference:**

Maps.difference(Map, Map)用来比较两个Map以获取所有不同点。该方法返回MapDifference对象，把不同点的维恩图分解为：

|  |  |
| --- | --- |
| entriesInCommon() | 两个Map中都有的映射项，包括匹配的键与值 |
| entriesDiffering() | 键相同但是值不同值映射项。返回的Map的值类型为MapDifference.ValueDifference，以表示左右两个不同的值 |
| entriesOnlyOnLeft() | 键只存在于左边Map的映射项 |
| entriesOnlyOnRight() | 键只存在于右边Map的映射项 |

**举个栗子：**

1. ImmutableMap<String, Integer> left = ImmutableMap.of("a", 1, "b", 2, "c", 3);
2. ImmutableMap<String, Integer> right = ImmutableMap.of("b", 2, "c", 4, "d", 5);
3. MapDifference<String, Integer> difference = Maps.difference(left, right);
5. // {b=2}
6. System.out.println(difference.entriesInCommon());
7. // {c=(3, 4)}
8. System.out.println(difference.entriesDiffering());
9. // {a=1}
10. System.out.println(difference.entriesOnlyOnLeft());
11. // {d=5}
12. System.out.println(difference.entriesOnlyOnRight());

#### Multisets

标准的Collection操作会忽略Multiset重复元素的个数，而只关心元素是否存在于Multiset中，如containsAll方法。为此，Multisets提供了若干方法，以顾及Multiset元素的重复性：

|  |  |  |
| --- | --- | --- |
| 方法 | 说明 | 和Collection方法的区别 |
| containsOccurrences(Multiset sup, Multiset sub) | 对任意o，如果sub.count(o)<=  super.count(o)，返回true | Collection.containsAll忽略个数，而只关心sub的元素是否都在super中 |
| removeOccurrences(Multiset removeFrom, Multiset toRemove) | 对toRemove中的重复元素，仅在removeFrom中删除相同个数。 | Collection.removeAll移除所有出现在toRemove的元素 |
| retainOccurrences(Multiset removeFrom,Multiset toRetain) | 修改removeFrom，以保证任意o都符合removeFrom.count(o)<=  toRetain.count(o) | Collection.retainAll保留所有出现在toRetain的元素 |
| intersection(Multiset,Multiset) | 返回两个multiset的交集; | 没有类似方法 |

**举个栗子：**

1. HashMultiset<String> multiset = HashMultiset.create();
2. multiset.add("a",2);
3. HashMultiset<Object> multiset1 = HashMultiset.create();
4. multiset1.add("a",5);
6. //返回 true,都包含 a 值
7. System.out.println(multiset.containsAll(multiset1));
8. //返回false multiset 的值（2） 不小于 multiset1的值（5）
9. System.out.println(Multisets.containsOccurrences(multiset, multiset1));
10. Multisets.removeOccurrences(multiset1,multiset);
11. //[a x 3] 移除了两个 a 还剩下 3个
12. System.out.println(multiset1);
13. //全部移除
14. System.out.println(multiset1.removeAll(multiset));
15. // multiset1为空了
16. System.out.println(multiset1.isEmpty());

**Multisets中的其他工具方法还包括：**

|  |  |
| --- | --- |
| copyHighestCountFirst(Multiset) | 返回Multiset的不可变拷贝，并将元素按重复出现的次数做降序排列 |
| unmodifiableMultiset(Multiset) | 返回Multiset的只读视图 |
| unmodifiableSortedMultiset(SortedMultiset) | 返回SortedMultiset的只读视图 |

**举个栗子：**

1. HashMultiset<String> set = HashMultiset.create();
2. set.add("a",3);
3. set.add("b",5);
4. set.add("c",1);
5. ImmutableMultiset<String> strings = Multisets.copyHighestCountFirst(set);
6. // [b x 5, a x 3, c]
7. System.out.println(strings);

#### Multimaps

**Index:** 作为Maps.uniqueIndex的兄弟方法，Multimaps.index(Iterable, Function)通常针对的场景是：有一组对象，它们有共同的特定属性，我们希望按照这个属性的值查询对象，但属性值不一定是独一无二的。

**举个栗子**：比如想按字符串长度进行分组。

1. ImmutableSet<String> digits = ImmutableSet.of("zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine");
2. Function<String, Integer> function = **new** Function<String, Integer>() {
3. @Override
4. **public** Integer apply(String s) {
5. **return** s.length();
6. }
7. };
8. ImmutableListMultimap<Integer, String> index = Multimaps.index(digits, function);
9. //{4=[zero, four, five, nine], 3=[one, two, six], 5=[three, seven, eight]}
10. System.out.println(index);

**invertFrom：**

Multimap可以把多个键映射到同一个值（实际上这是任何map都有的特性），也可以把一个键映射到多个值，反转Multimap也会很有用。Guava 提供了invertFrom(Multimap toInvert,Multimap dest)做这个操作，并且你可以自由选择反转后的Multimap实现。

**举个栗子：**

1. ArrayListMultimap<String, Integer> multimap = ArrayListMultimap.create();
2. multimap.putAll("b", Ints.asList(2,4,6));
3. multimap.putAll("a", Ints.asList(4,2,1));
4. multimap.putAll("c", Ints.asList(2,5,3));
5. //注意我们选择的实现，因为选了TreeMultimap，得到的结果是有序的
6. TreeMultimap<Integer, String> invert = Multimaps.invertFrom(multimap, TreeMultimap.create());
7. // {1=[a], 2=[a, b, c], 3=[c], 4=[a, b], 5=[c], 6=[b]}
8. System.out.println(invert);

#### Tables

**自定义Table:**

堪比Multimaps.newXXXMultimap(Map, Supplier)工具方法，Tables.newCustomTable(Map, Supplier<Map>)允许你指定Table用什么样的map实现行和列。

举个栗子：

1. Table<String, Character, Integer> table = Tables.newCustomTable(
2. Maps.<String, Map<Character, Integer>>newLinkedHashMap(),
3. **new** Supplier<Map<Character, Integer>>() {
4. @Override
5. **public** Map<Character, Integer> get() {
6. **return** Maps.newLinkedHashMap();
7. }
8. });

**transpose:**

transpose(Table<R, C, V>)方法允许你把Table<C, R, V>转置成Table<R, C, V>。例如，如果你在用Table构建加权有向图，这个方法就可以把有向图反转。

### 扩展工具类

有时候你需要实现自己的集合扩展。也许你想要在元素被添加到列表时增加特定的行为，或者你想实现一个Iterable，其底层实际上是遍历数据库查询的结果集。Guava为你，也为我们自己提供了若干工具方法，以便让类似的工作变得更简单。

#### Forwarding装饰器

针对所有类型的集合接口，Guava都提供了Forwarding抽象类以简化装饰者模式的使用。Forwarding抽象类定义了一个抽象方法：delegate()，你可以覆盖这个方法来返回被装饰对象。所有其他方法都会直接委托给delegate()。例如说：ForwardingList.get(int)实际上执行了delegate().get(int)。

通过创建ForwardingXXX的子类并实现delegate()方法，可以选择性地覆盖子类的方法来增加装饰功能，而不需要自己委托每个方法——因为所有方法都默认委托给delegate()返回的对象，你可以只覆盖需要装饰的方法。

此外，很多集合方法都对应一个”标准方法[standardxxx]”实现，可以用来恢复被装饰对象的默认行为，以提供相同的优点。比如在扩展AbstractList或JDK中的其他骨架类时，可以使用类似standardAddAll这样的方法。

举个栗子：假设需要装饰一个List，让其记录所有添加进来的元素。无论元素用什么方法，add(int,E) add(E) 或addAll(Collection)都希望进行记录，因此需要覆盖所有这些方法。

1. **public** **class** AddLoggingList<E> **extends** ForwardingList<E> {
2. **final** List<E> delegate;
4. **public** AddLoggingList(List<E> delegate) {
5. **this**.delegate = delegate;
6. }
8. @Override
9. **protected** List<E> delegate() {
10. **return** delegate;
11. }
12. @Override
13. **public** **void** add(**int** index, E elem){
14. System.out.println("记录日志");
15. **super**.add(index,elem);
16. }
18. @Override
19. **public** **boolean** add(E element) {
20. // standardAdd 实际上调用 this.add(this.size(), element); 返回true
21. **return** standardAdd(element);
22. }
24. @Override
25. **public** **boolean** addAll(Collection<? **extends** E> collection) {
26. // standardAddAll 实际上也是调用 this.add(E)
27. **return** standardAddAll(collection);
28. }
30. **public** **static** **void** main(String[] args) {
32. ArrayList<Integer> objects = **new** ArrayList<>();
33. AddLoggingList<Integer> integers = **new** AddLoggingList<>(objects);
34. //会输出  记录日志
35. integers.add(1);
36. }
37. }

#### PeekingIterator

有时候，普通的Iterator接口还不够。

Iterators提供一个Iterators.peekingIterator(Iterator)方法，来把Iterator包装为PeekingIterator，这是Iterator的子类，它能让你事先窥视[peek()]到下一次调用next()返回的元素。

注意：Iterators.peekingIterator返回的PeekingIterator不支持在peek()操作之后调用remove()方法。

**举个栗子：** 复制一个List，并去除连续的重复元素。

1. List<Integer> integers = Ints.asList(1, 2, 2, 3, 4, 5, 5, 5, 5);
2. List<Integer> result = Lists.newArrayList();
3. PeekingIterator<Integer> iter = Iterators.peekingIterator(integers.iterator());
4. **while** (iter.hasNext()){
5. Integer current = iter.next();
6. // 跳过重复的元素
7. **while** (iter.hasNext() && iter.peek().equals(current)) {
8. iter.next();
9. }
10. result.add(current);
11. }
12. // [1, 2, 3, 4, 5]
13. System.out.println(result);

#### AbstractIterator

实现自己的Iterator？AbstractIterator让生活更轻松。

**举个栗子：**比如要包装一个Iterator 跳过null 值 和 空值

1. **public** **static** Iterator<String> skipNulls(**final** Iterator<String> in){
2. **return** **new** AbstractIterator<String>() {
3. @Override
4. **protected** String computeNext() {
5. **while** (in.hasNext()){
6. String next = in.next();
7. **if** (next != **null** && !"".equals(next)) {
8. **return** next;
9. }
10. }
11. **return** endOfData();
12. }
13. };
14. }
16. **public** **static** **void** main(String[] args) {
17. List<String> list = Arrays.asList("fadsf", **null**, "", **null**, "dsafa");
18. Iterator<String> stringIterator = skipNulls(list.iterator());
19. **while** (stringIterator.hasNext()){
20. //fadsf
21. //dsafa
22. System.out.println(stringIterator.next());
23. }
24. }

实现了computeNext()方法，来计算下一个值。如果循环结束了也没有找到下一个值，请返回endOfData()表明已经到达迭代的末尾。

#### AbstractSequentialIterator

有一些迭代器用其他方式表示会更简单。AbstractSequentialIterator 就提供了表示迭代的另一种方式。

**举个栗子：**

1. Iterator<Integer> powersOfTwo = **new** AbstractSequentialIterator<Integer>(1) {
2. @Override
3. **protected** Integer computeNext(Integer previous) {
4. **return** (previous == 1 << 30) ? **null** : previous \* 2;
5. }
6. };

在这儿实现了computeNext(T)方法，它能接受前一个值作为参数。

注意，你必须额外传入一个初始值，或者传入null让迭代立即结束。因为computeNext(T)假定null值意味着迭代的末尾——AbstractSequentialIterator不能用来实现可能返回null的迭代器。

## 图（Graph）

### 3.1 概要

Graph 模块是一个描述实体以及实体之间的关系的图数据结构模型库。例如：网页与超链接、科学家与他们写的论文、机场与其航线、人与其家族等。Graph模块的目的是提供一种通用以及可扩展的语言来描述类似上述的举例。

### 3.2 定义

图中包含一组节点（node）和 一组连接节点的边（edge）;边缘的节点称为端点（endpoint）。如果一条边定义了开始（source）和结束（target），这条边称为有向边（directed）,否则称为无向边（undirected）。有向边适用于非对称的关系模型。而无向边适用于对称关系模型。图中每一条边都是有向边的，被称为有向图；每一条边都是无向边的，被称为无向图。（graph模块不支持图中既有有向边又有无向边的情形）。

如：graph.addEdge(nodeU,nodeV,edgeUV);

nodeU和nodeV是两个邻接点(adjacent)。

edgeUV是顶点nodeU到顶点nodeV的事件(incident)（反之亦然）

**在有向图中，有如下定义：**

nodeU是nodeV的一个前趋(predecessor)

nodeV是nodeU的一个后继(successor)

edgeUV是nodeU的一条出度(outgoing)边

edgeUV是nodeV的一条入度(incoming)边

nodeU是边edgeUV的起点(source)

nodeV是边edgeUV的终点(target)

在无向图中，有如下定义：

nodeU既是nodeV的前趋也是nodeV的后继

nodeV既是nodeU的前趋也是nodeU的后继

edgeUV既是nodeU的入度也是nodeU的出度

edgeUV既是nodeV的入度也是nodeV的出度

一条连接节点本身的边被称为自环（self-loop）.。也就是说，一条边连接了两个相同的节点。如果这个自环是有向的，那么这条边既是节点的入度边也是节点的出度边，这个节点既是边的起点(source)也是边的终点(target)。

如果两条边以相同的顺序连接相同的节点，则称这两条边为平行边(parallel)；

如果以相反的顺序连接相同的节点则称这两条边为逆平行边(antiparallel)。（无向边不能被称为逆平行边）

### 3.3 功能

graph模块的核心是提供图相关操作的接口和类。另外，它没有提供类似I/O或者可视化的功能。如果选用这个模块将会有非常多的限制，具体详细信息可以查看下面FAQ的相关主题。总体来讲，它提供了如下几种类型的图：

* 有向图
* 无向图
* 节点和（或）边带权图
* 允许（不允许）自环图
* 允许（不允许）平行边图（允许平行边图有时也称为多重图(multigraphs)
* 节点或边被有序插入、顺序、无序图

graph中的各种类型的图都是通过与其相关的Builder具体实现类型来构建的，不过这些Builder实现类型不一定支持上面提到的所有图类型，但也可能支持其他类型的图。

库中图的数据结构是通过矩阵、邻接list或邻接map等方式来存储的，选择何种存储方式取决于适用的实现场景。

对于以下这些变形图在common.graph中没有确切的支持，尽管它们可以通过已有的图类型进行建模：

* 树(trees)、森林(forests)
* 由不同类型的同类元素（节点或边）构成的图。（二分图/k分图、multimodal graphs）
* 超图

### 3.4 图的类型

有三种通过边作为区分依据的接口：Graph、ValueGraph、Network;

#### 3.4.1 Graph

Graph是最简单也是最基本的图类型。为了处理节点与节点之间的关系它定义了一些基本的操作。这些节点在图中都是唯一的对象，在其内部数据结构中，你可以认为它们是Map的键值(Key)。

Graph中的边是完全匿名的，他们只能根据端点来定义。举例：Graph<Airport>中，其边连接任意两个可以直航的机场。

**举个栗子：**

1. //构建一个无向图
2. MutableGraph<String> graph = GraphBuilder.undirected().build();
3. //增加a,b,c,d 四个节点
4. graph.addNode("a");
5. graph.addNode("b");
6. graph.addNode("c");
7. graph.addNode("d");
8. //增加 a-b  a-b c-d 三条边
9. graph.putEdge("a","c");
10. graph.putEdge("a","b");
11. graph.putEdge("c","d");
13. //查看a节点所在的边
14. Set<EndpointPair<String>> a = graph.incidentEdges("a");
15. // [[b, a], [c, a]]
16. System.out.println(a);
17. // 查看b节点的后继节点
18. Set<String> b = graph.successors("b");
19. // [a]
20. System.out.println(b);
21. //查看所有边
22. Set<EndpointPair<String>> edges = graph.edges();
23. // [[b, a], [c, a], [d, c]]
24. System.out.println(edges);
26. //c节点出度数量  2
27. System.out.println(graph.outDegree("c"));
28. //获取c 的邻接点
29. Set<String> c = graph.adjacentNodes("c");
30. //[d, a]
31. System.out.println(c);

#### 3.4.2 ValueGraph

接口ValueGraph包含了Graph中的所有与节点相关的方法，并增加了一些检索指定边权值的方法。

ValueGraph中的每一条边都有一个与之相关的特定权值，但是这些权值不能保证唯一性。ValueGraph与Graph的关系类似与Map与Set的关系（Graph中的边是以顶点对的形式保存在Set中，而ValueGraph的边是以顶点对与其权值的映射关系保存在Map中）。

ValueGraph提供了一个asGraph()的函数，它可以从ValueGraph中返回一个Graph视图，这样作用于Graph实例上的方法也能作用于ValueGraph的实例上。

举例：ValueGraph<Airport, Integer>，其边表示在能直航的两个机场之间航班必须花费的时间。

**举个例子：**

1. MutableValueGraph<String, Integer> graph = ValueGraphBuilder.directed().build();
2. //添加三个节点
3. graph.addNode("a");
4. graph.addNode("b");
5. graph.addNode("c");
6. //添加两条有向边
7. graph.putEdgeValue("a","b",2);
8. graph.putEdgeValue("a","c",3);
10. //返回一个Graph视图
11. Graph<String> stringGraph = graph.asGraph();
12. //isDirected: true, allowsSelfLoops: false, nodes: [a, b, c], edges: [<a -> b>, <a -> c>]
13. System.out.println(stringGraph);
14. //取权值
15. Optional<Integer> integer = graph.edgeValue("a", "b");
16. **if** (integer.isPresent()){
17. //2
18. System.out.println(integer.get());
19. }
20. Integer integer1 = graph.edgeValueOrDefault("b", "c", -1);
21. //-1
22. System.out.println(integer1);

#### 3.4.3 Network

Network中包含了Graph中的所有与节点相关的方法，还增加了操作边以及操作顶点与边的关系的方法。

Network中每一条边都是唯一的，就像节点在所有的Graph类型中是唯一的一样。边的唯一性限制使得Network能够天然的支持并行边，以及与边和节点与边相关的方法。

Network类提供了一个asGraph()的方法，它可以从Network中返回一个Graph视图，这样作用于Graph实例上的方法也能操作Network的实例上。

举例：Network<Airport, Flight>，它的每一条边代表了从一个机场到另一个机场可以乘坐的特定航班（两个机场之间可以同时有多趟航班）。

**举个栗子：**

1. MutableNetwork<String, String> network = NetworkBuilder.undirected().build();
3. network.addNode("a");
4. network.addNode("b");
5. network.addNode("c");
6. network.addNode("d");
8. network.addEdge("a","c","d");
9. network.addEdge("b","d","c");
11. Graph<String> stringGraph = network.asGraph();
12. System.out.println(stringGraph);
14. Set<String> a = network.adjacentNodes("a");
15. System.out.println(a);
16. Optional<String> s = network.edgeConnecting("a", "c");
17. **if** (s.isPresent()){
18. System.out.println(s.get());
19. }

#### 3.4.4 三者的选择

* Graph中的边是节点之间的匿名连接，没有自己的标识或属性。如果每一对节点之间都是通过最多一条边连接的，而且这些边没有任何与之相关的信息时，则选用它。
* ValueGraph中的边带有一个值（例如权值或标签），且边在整个图中不能保证其唯一性。如果每一对节点之间都是通过最多一条边连接的，并且每一条边都有与之相关的权值时，则选用它。
* Network中边是全局唯一的，就像节点在图中是唯一的一样。如果边对象需要唯一，并且希望能查询它们的引用时，则选用它。(请注意，这种唯一性使得Network支持平行边。)

## 缓存（Caches）

### 适用性

缓存在很多场景下都是相当有用的。例如，计算或检索一个值的代价很高，并且对同样的输入需要不止一次获取值的时候，就应当考虑使用缓存。

Guava Cache与ConcurrentMap很相似，但也不完全一样。最基本的区别是ConcurrentMap会一直保存所有添加的元素，直到显式地移除。相对地，Guava Cache为了限制内存占用，通常都设定为自动回收元素。在某些场景下，尽管LoadingCache 不回收元素，它也是很有用的，因为它会自动加载缓存。

Guava Cache 适用于：

* 愿意消耗一些内存空间来提升速度。
* 预料到某些键会被查询一次以上。
* 缓存中存放的数据总量不会超出内存容量。

Guava Cache是单个应用运行时的本地缓存。它不把数据存放到文件或外部服务器

### 加载

在使用缓存前，提出一个问题：有没有合理的默认方法来加载或计算与键关联的值？如果有的话，应当使用CacheLoader。如果没有，或者想要覆盖默认的加载运算，同时保留"获取缓存-如果没有-则计算"[get-if-absent-compute]的原子语义，应该在调用get时传入一个Callable实例。缓存元素也可以通过Cache.put方法直接插入，但自动加载是首选的，因为它可以更容易地推断所有缓存内容的一致性。

#### CacheLoader

LoadingCache是附带CacheLoader构建而成的缓存实现。创建自己的CacheLoader只需要简单地实现V load(K key) throws Exception方法。

**举个栗子：**

1. LoadingCache<String, String> cache = CacheBuilder.newBuilder()
2. .build(**new** CacheLoader<String, String>() {
3. @Override
4. **public** String load(String s) **throws** Exception {
5. **return** "hello " + s;
6. }
7. });
9. **try** {
10. String test = cache.get("test");
11. // hello test
12. System.out.println(test);
13. } **catch** (ExecutionException e) {
14. e.printStackTrace();
15. }

从LoadingCache查询的正规方式是使用get(K)方法。这个方法要么返回已经缓存的值，要么使用CacheLoader向缓存原子地加载新值。由于CacheLoader可能抛出异常，LoadingCache.get(K)也声明为抛出ExecutionException异常。如果你定义的CacheLoader没有声明任何检查型异常，则可以通过getUnchecked(K)查找缓存；但必须注意，一旦CacheLoader声明了检查型异常，就不可以调用getUnchecked(K)。

**举个栗子：**

1. LoadingCache<String, String> build = CacheBuilder.newBuilder()
2. .build(**new** CacheLoader<String, String>() {
3. @Override
4. **public** String load(String s){
5. **return** "hello " + s;
6. }
7. });
8. String test = build.getUnchecked("test");
9. // hello test
10. System.out.println(test);

getAll(Iterable<? extends K>)方法用来执行批量查询。默认情况下，对每个不在缓存中的键，getAll方法会单独调用CacheLoader.load来加载缓存项。如果批量的加载比多个单独加载更高效，你可以重载CacheLoader.loadAll来利用这一点。getAll(Iterable)的性能也会相应提升。

**举个栗子：**

1. LoadingCache<String, String> cache1 = CacheBuilder.newBuilder()
2. .build(**new** CacheLoader<String, String>() {
3. @Override
4. **public** String load(String s) **throws** Exception {
5. **return** "hi " + s;
6. }
7. @Override
8. **public** Map<String, String> loadAll(Iterable<? **extends** String> keys) **throws** Exception {
9. Map<String, String> map = **new** HashMap<>(64);
10. Iterator<? **extends** String> iterator = keys.iterator();
11. **while** (iterator.hasNext()) {
12. String key = iterator.next();
13. map.put(key, "hi all " + key);
14. }
15. **return** map;
16. }
17. });
19. List<String> list = Arrays.asList("demo", "test");
20. **try** {
21. ImmutableMap<String, String> all = cache1.getAll(list);
22. //{demo=hi all demo, test=hi all test}
23. System.out.println(all);
24. } **catch** (ExecutionException e) {
25. e.printStackTrace();
26. }

#### Callable

所有类型的Guava Cache，不管有没有自动加载功能，都支持get(K, Callable<V>)方法。这个方法返回缓存中相应的值，或者用给定的Callable运算并把结果加入到缓存中。在整个加载方法完成前，缓存项相关的可观察状态都不会更改。这个方法简便地实现了模式"如果有缓存则返回；否则运算、缓存、然后返回"。

举个栗子：

1. Cache<String, String> cache = CacheBuilder.newBuilder()
2. .build();
4. **try** {
5. String demo = cache.get("demo", **new** Callable<String>() {
6. @Override
7. **public** String call() **throws** Exception {
8. **return** "default";
9. }
10. });
11. //default
12. System.out.println(demo);
13. } **catch** (ExecutionException e) {
14. e.printStackTrace();
15. }

需要注意：LoadingCache 继承 Cache。而Cache 只有V get(K var1, Callable<? extends V> var2) throws ExecutionException; 接口。LoadingCache 多一个 V get(K var1) throws ExecutionException;接口。

#### 显示插入

使用cache.put(key, value)方法可以直接向缓存中插入值，这会直接覆盖掉给定键之前映射的值。使用Cache.asMap()视图提供的任何方法也能修改缓存。但请注意，asMap视图的任何方法都不能保证缓存项被原子地加载到缓存中。进一步说，asMap视图的原子运算在Guava Cache的原子加载范畴之外，所以相比于Cache.asMap().putIfAbsent(K,V)，Cache.get(K, Callable<V>) 应该总是优先使用。

### 缓存回收

Guava Cache提供了三种基本的缓存回收方式：基于容量回收、定时回收和基于引用回收。

#### 基于容量的回收

如果要规定缓存项的数目不超过固定值，只需使用CacheBuilder.maximumSize(long)。缓存将尝试回收最近没有使用或总体上很少使用的缓存项。——注意：在缓存项的数目达到限定值之前，缓存就可能进行回收操作——通常来说，这种情况发生在缓存项的数目逼近限定值时。

**举个栗子：**

1. LoadingCache<String, String> cache = CacheBuilder.newBuilder()
2. .maximumSize(3)
3. .build(**new** CacheLoader<String, String>() {
4. @Override
5. **public** String load(String s) **throws** Exception {
6. **return** "hi " + s;
7. }
8. });
9. cache.put("test1","test1");
10. //test1  异常在方法声明出抛出了
11. System.out.println(cache.get("test1"));
13. cache.put("test2","test2");
14. cache.put("test3","test3");
15. cache.put("test4","test4");
16. // hi test1 可见 test1 被清除了
17. System.out.println(cache.get("test1"));
18. //test4
19. System.out.println(cache.get("test4"));

不同的缓存项有不同的“权重”（weights）——例如，如果你的缓存值，占据完全不同的内存空间，你可以使用CacheBuilder.weigher(Weigher)指定一个权重函数，并且用CacheBuilder.maximumWeight(long)指定最大总重。在权重限定场景中，除了要注意回收也是在重量逼近限定值时就进行了，还要知道重量是在缓存创建时计算的，因此要考虑重量计算的复杂度。

**举个栗子：**

1. LoadingCache<String, String> build = CacheBuilder.newBuilder()
2. .maximumWeight(3)
3. .weigher(**new** Weigher<String, String>() {
4. @Override
5. **public** **int** weigh(String s, String s2) {
6. **return** s2.length();
7. }
8. }).build(**new** CacheLoader<String, String>() {
9. @Override
10. **public** String load(String s) **throws** Exception {
11. **return** "hi " + s;
12. }
13. });
15. build.put("a","a");
16. //a
17. System.out.println(build.get("a"));
18. build.put("b","bb");
19. build.put("c","ccc");
20. //ccc
21. System.out.println(build.get("c"));
22. build.put("d","dd");
23. //hi c  c被清除了
24. System.out.println(build.get("c"));
25. // dd
26. System.out.println(build.get("d"));

#### 定时回收

CacheBuilder提供两种定时回收的方法：

1. expireAfterAccess(long, TimeUnit)：缓存项在给定时间内没有被读/写访问，则回收。请注意这种缓存的回收顺序和基于大小回收一样。
2. expireAfterWrite(long, TimeUnit)：缓存项在给定时间内没有被写访问（创建或覆盖），则回收。如果认为缓存数据总是在固定时候后变得陈旧不可用，这种回收方式是可取的。

注：定时回收周期性地在写操作中执行，偶尔在读操作中执行。

**举个栗子：**

1. LoadingCache<String, String> cache = CacheBuilder.newBuilder()
2. .maximumSize(1000)
3. .expireAfterWrite(1, TimeUnit.SECONDS)
4. .build(**new** CacheLoader<String, String>() {
5. @Override
6. **public** String load(String s) **throws** Exception {
7. **return** "hi " + s;
8. }
9. });
10. cache.put("test","test");
11. //test
12. System.out.println(cache.get("test"));
13. TimeUnit.SECONDS.sleep(1);
14. // hi test
15. System.out.println(cache.get("test"));
16. cache.put("demo","demo");
17. // hi test
18. System.out.println(cache.get("test"));

注意： 对定时回收进行测试时，不一定非得花费两秒钟去测试两秒的过期。你可以使用Ticker接口和CacheBuilder.ticker(Ticker)方法在缓存中自定义一个时间源，而不是非得用系统时钟。

#### 基于引用的回收

通过使用弱引用的键、或弱引用的值、或软引用的值，Guava Cache可以把缓存设置为允许垃圾回收：

* CacheBuilder.weakKeys()：使用弱引用存储键。当键没有其它（强或软）引用时，缓存项可以被垃圾回收。因为垃圾回收仅依赖恒等式（==），使用弱引用键的缓存用==而不是equals比较键。
* CacheBuilder.weakValues()：使用弱引用存储值。当值没有其它（强或软）引用时，缓存项可以被垃圾回收。因为垃圾回收仅依赖恒等式（==），使用弱引用值的缓存用==而不是equals比较值。
* CacheBuilder.softValues()：使用软引用存储值。软引用只有在响应内存需要时，才按照全局最近最少使用的顺序回收。考虑到使用软引用的性能影响，我们通常建议使用更有性能预测性的缓存大小限定（见上文，基于容量回收）。使用软引用值的缓存同样用==而不是equals比较值。

#### 显示清除

任何时候，都可以显式地清除缓存项，而不是等到它被回收：

* 个别清除：Cache.invalidate(key)
* 批量清除：Cache.invalidateAll(keys)
* 清除所有缓存项：Cache.invalidateAll()

**举个栗子：**

1. CacheLoader<String, String> cacheLoader1 = **new** CacheLoader<String, String>() {
2. @Override
3. **public** String load(String s) **throws** Exception {
4. **return** "hi " + s;
5. }
6. };
8. LoadingCache<String, String> build = CacheBuilder.newBuilder().build(cacheLoader1);
9. build.put("demo","demo");
10. // demo
11. System.out.println(build.get("demo"));
12. build.invalidate("demo");
13. // hi demo
14. System.out.println(build.get("demo"));

#### 4.3.5 移除监听器

通过CacheBuilder.removalListener(RemovalListener)，你可以声明一个监听器，以便缓存项被移除时做一些额外操作。缓存项被移除时，RemovalListener会获取移除通知[RemovalNotification]，其中包含移除原因[RemovalCause]、键和值。

**举个栗子：**

1. RemovalListener<String, String> removalListener = **new** RemovalListener<String, String>() {
2. @Override
3. **public** **void** onRemoval(RemovalNotification<String, String> removalNotification) {
4. String key = removalNotification.getKey();
5. String value = removalNotification.getValue();
6. RemovalCause cause = removalNotification.getCause();
7. System.out.println("remove cause is : " + cause + " key is:" + key + " value is:" + value);
8. }
9. };
10. CacheLoader<String, String> cacheLoader = **new** CacheLoader<String, String>() {
11. @Override
12. **public** String load(String s) **throws** Exception {
13. **return** "hello " + s;
14. }
15. };
16. LoadingCache<String, String> cache = CacheBuilder.newBuilder()
17. .recordStats()
18. .maximumSize(1000)
19. .expireAfterWrite(1, TimeUnit.SECONDS)
20. .removalListener(removalListener)
21. .build(cacheLoader);
23. cache.put("test", "world");
24. //同时会输出 remove cause is : EXPLICIT key is:test value is:world
25. cache.invalidate("test");
26. //hello test 说明已经被清除了
27. System.out.println(cache.get("test"));

**注意：**

RemovalListener抛出的任何异常都会在记录到日志后被丢弃

默认情况下，监听器方法是在移除缓存时同步调用的。因为缓存的维护和请求响应通常是同时进行的，代价高昂的监听器方法在同步模式下会拖慢正常的缓存请求。在这种情况下，你可以使用RemovalListeners.asynchronous(RemovalListener, Executor)把监听器装饰为异步操作。

**清理什么时候发生：**

使用CacheBuilder构建的缓存不会"自动"执行清理和回收工作，也不会在某个缓存项过期后马上清理，也没有诸如此类的清理机制。相反，它会在写操作时顺带做少量的维护工作，或者偶尔在读操作时做——如果写操作实在太少的话。

这样做的原因在于：如果要自动地持续清理缓存，就必须有一个线程，这个线程会和用户操作竞争共享锁。此外，某些环境下线程创建可能受限制，这样CacheBuilder就不可用了。

相反，把选择权交到你手里。如果缓存是高吞吐的，那就无需担心缓存的维护和清理等工作。如果缓存只会偶尔有写操作，而又不想清理工作阻碍了读操作，那么可以创建自己的维护线程，以固定的时间间隔调用Cache.cleanUp()。ScheduledExecutorService可以很好地实现这样的定时调度。

#### 刷新

刷新和回收不太一样。正如LoadingCache.refresh(K)所声明，刷新表示为键加载新值，这个过程可以是异步的。在刷新操作进行时，缓存仍然可以向其他线程返回旧值，而不像回收操作，读缓存的线程必须等待新值加载完成。

如果刷新过程抛出异常，缓存将保留旧值，而异常会在记录到日志后被丢弃。

重载CacheLoader.reload(K, V)可以扩展刷新时的行为，这个方法允许开发者在计算新值时使用旧的值。

**举个栗子：**

1. **public** **static** **void** main(String[] args) **throws** ExecutionException, InterruptedException {
2. CacheLoader<String, String> cacheLoader = **new** CacheLoader<String, String>() {
3. @Override
4. **public** String load(String s) **throws** Exception {
5. **return** s;
6. }
8. @Override
9. **public** ListenableFuture<String> reload(String key, String oldValue) **throws** Exception {
10. ListenableFutureTask<String> task = ListenableFutureTask.create(**new** Callable<String>() {
11. @Override
12. **public** String call() **throws** Exception {
13. **return** "new " + oldValue;
14. }
15. });
16. task.run();
17. **return** task;
18. }
19. };
20. LoadingCache<String, String> cache = CacheBuilder.newBuilder()
21. .maximumSize(100)
22. .refreshAfterWrite(1, TimeUnit.SECONDS)
23. .build(cacheLoader);
24. cache.put("t","tttt");
25. //tttt
26. System.out.println(cache.get("t"));
27. TimeUnit.SECONDS.sleep(1);
28. // new tttt
29. System.out.println(cache.get("t"));
30. }

CacheBuilder.refreshAfterWrite(long, TimeUnit)可以为缓存增加自动定时刷新功能。和expireAfterWrite相反，refreshAfterWrite通过定时刷新可以让缓存项保持可用，但请注意：缓存项只有在被检索时才会真正刷新（如果CacheLoader.refresh实现为异步，那么检索不会被刷新拖慢）。因此，如果你在缓存上同时声明expireAfterWrite和refreshAfterWrite，缓存并不会因为刷新盲目地定时重置，如果缓存项没有被检索，那刷新就不会真的发生，缓存项在过期时间后也变得可以回收。

### 其他特性

#### 统计

CacheBuilder.recordStats()用来开启Guava Cache的统计功能。统计打开后，Cache.stats()方法会返回CacheStats对象以提供如下统计信息：

* hitRate()：缓存命中率；
* averageLoadPenalty()：加载新值的平均时间，单位为纳秒；
* evictionCount()：缓存项被回收的总数，不包括显式清除。

此外，还有其他很多统计信息。这些统计信息对于调整缓存设置是至关重要的，在性能要求高的应用中我们建议密切关注这些数据。

**举个栗子：**

1. LoadingCache<String, String> cache = CacheBuilder.newBuilder()
2. .recordStats()
3. .maximumSize(1000)
4. .build(**new** CacheLoader<String, String>() {
5. @Override
6. **public** String load(String s) **throws** Exception {
7. **return** s;
8. }
9. });
10. cache.put("t", "world");
11. cache.put("a", "apple");
12. cache.put("b", "banana");
13. //tt
14. System.out.println(cache.get("tt"));
15. //world
16. System.out.println(cache.get("t"));
17. cache.invalidate("a");
18. CacheStats stats = cache.stats();
19. //0.5
20. System.out.println(stats.hitRate());
21. //2793672.0
22. System.out.println(stats.averageLoadPenalty());
23. //0
24. System.out.println(stats.evictionCount());
25. //CacheStats{hitCount=1, missCount=1, loadSuccessCount=1, loadExceptionCount=0, totalLoadTime=2793672, evictionCount=0}
26. System.out.println(stats);

#### asMap视图

asMap视图提供了缓存的ConcurrentMap形式，但asMap视图与缓存的交互需要注意：

* cache.asMap()包含当前所有加载到缓存的项。因此相应地，cache.asMap().keySet()包含当前所有已加载键;
* asMap().get(key)实质上等同于cache.getIfPresent(key)，而且不会引起缓存项的加载。这和Map的语义约定一致。
* 所有读写操作都会重置相关缓存项的访问时间，包括Cache.asMap().get(Object)方法和Cache.asMap().put(K, V)方法，但不包括Cache.asMap().containsKey(Object)方法，也不包括在Cache.asMap()的集合视图上的操作。比如，遍历Cache.asMap().entrySet()不会重置缓存项的读取时间。

## 函数式风格(Functional idioms)

略

## 并发（Concurrency）

### ListenableFuture

Guava 定义了 ListenableFuture接口并继承了JDK concurrent包下的Future 接口。建议你在代码中多使用ListenableFuture来代替JDK的 Future, 因为：

* 大多数Futures 方法中需要它。
* 转到ListenableFuture 编程比较容易。
* Guava提供的通用公共类封装了公共的操作方方法，不需要提供Future和ListenableFuture的扩展方法。

#### 接口

传统JDK中的Future通过异步的方式计算返回结果:在多线程运算中可能或者可能在没有结束返回结果，Future是运行中的多线程的一个引用句柄，确保在服务执行返回一个Result。

ListenableFuture可以允许你注册回调方法(callbacks)，在运算（多线程执行）完成的时候进行调用, 或者在运算（多线程执行）完成后立即执行。这样简单的改进，使得可以明显的支持更多的操作，这样的功能在JDK concurrent中的Future是不支持的。

ListenableFuture 中的基础方法是addListener(Runnable, Executor), 该方法会在多线程运算完的时候，指定的Runnable参数传入的对象会被指定的Executor执行。

#### 添加回调

使用 Futures.addCallback(ListenableFuture<V>, FutureCallback<V>, Executor)的方式。为了简化使用，Callback采用轻量级的设计. FutureCallback<V> 中实现了两个方法:

onSuccess(V),在Future成功的时候执行，根据Future结果来判断。

onFailure(Throwable), 在Future失败的时候执行，根据Future结果来判断。

#### 6.1.3 ListenableFuture的创建

对应JDK中的 ExecutorService.submit(Callable) 提交多线程异步运算的方式，Guava 提供了ListeningExecutorService 接口, 该接口返回 ListenableFuture 而相应的 ExecutorService 返回普通的 Future。将 ExecutorService 转为 ListeningExecutorService，可以使用MoreExecutors.listeningDecorator(ExecutorService)进行装饰。

**举个栗子：**

1. //JDk
2. ExecutorService executorService = Executors.newFixedThreadPool(10);
3. Future<Object> future = executorService.submit(**new** Callable<Object>() {
4. @Override
5. **public** Object call() **throws** Exception {
6. System.out.println("JDK running");
7. **return** **new** Object();
8. }
9. });
10. executorService.shutdown();
12. //Guava
13. ListeningExecutorService service = MoreExecutors.listeningDecorator(Executors.newFixedThreadPool(10));
14. ListenableFuture<Object> explosion = service.submit(**new** Callable<Object>() {
15. @Override
16. **public** Object call() **throws** Exception {
17. System.out.println("Guava Running");
18. **return** **new** Object();
19. }
20. });
21. //ListenableFuture 继承Future 并且多了一个addListener方法
22. //该方法在多线程运算完的时候，指定的Runnable参数传入的对象会被指定的Executor执行
23. explosion.addListener(**new** Runnable() {
24. @Override
25. **public** **void** run() {
26. System.out.println("Guava listener running");
27. }
28. },service);
29. // 添加回调
30. Futures.addCallback(explosion, **new** FutureCallback<Object>() {
31. @Override
32. **public** **void** onSuccess(@Nullable Object o) {
33. System.out.println("Guava callback success");
34. }
36. @Override
37. **public** **void** onFailure(Throwable throwable) {
38. System.out.println("Guava callback fail");
39. }
40. },service);
41. service.shutdown();
43. //output
44. //JDK running
45. //Guava Running
46. //Guava listener running
47. //Guava callback success

### Service

Guava包里的Service接口用于封装一个服务对象的运行状态、包括start和stop等方法。例如web服务器，RPC服务器、计时器等可以实现这个接口。对此类服务的状态管理并不轻松、需要对服务的开启/关闭进行妥善管理、特别是在多线程环境下尤为复杂。Guava包提供了一些基础类帮助你管理复杂的状态转换逻辑和同步细节。

#### 使用Service

一个服务的周期有：Service.State.NEW、Service.State.STARTING、Service.State.RUNNING、Service.State.STOPPING、Service.State.TERMINATED

服务一旦被停止就无法再重新启动了。如果服务在starting、running、stopping状态出现问题、会进入Service.State.FAILED.状态。调用 startAsync()方法可以异步开启一个服务,同时返回this对象形成方法调用链**。注意：**只有在当前服务的状态是NEW时才能调用startAsync()方法，因此最好在应用中有一个统一的地方初始化相关服务。停止一个服务也是类似的、使用异步方法stopAsync() 。但是不像startAsync(),多次调用这个方法是安全的。这是为了方便处理关闭服务时候的锁竞争问题。

**Service提供了一个方法用于等待服务状态转换的完成：**

通过 addListener()方法异步添加监听器。此方法允许你添加一个 Service.Listener 、它会在每次服务状态转换的时候被调用。注意：最好在服务启动之前添加Listener（这时的状态是NEW）、否则之前已发生的状态转换事件是无法在新添加的Listener上被重新触发的。

同步使用awaitRunning()。这个方法不能被打断、不强制捕获异常、一旦服务启动就会返回。如果服务没有成功启动，会抛出IllegalStateException异常。同样的， awaitTerminated() 方法会等待服务达到终止状态（TERMINATED 或者 FAILED）。两个方法都有重载方法允许传入超时时间。

Service 接口本身实现起来会比较复杂、且容易碰到一些捉摸不透的问题。因此我们不推荐直接实现这个接口。而是请继承Guava包里已经封装好的基础抽象类。每个基础类支持一种特定的线程模型。

#### 6.2.2 基础实现类

##### 6.2.2.1 AbstractIdleService

AbstractIdleService 类简单实现了Service接口、其在running状态时不会执行任何动作–因此在running时也不需要启动线程–但需要处理开启/关闭动作。要实现一个此类的服务，只需继承AbstractIdleService类，然后自己实现startUp() 和shutDown()方法就可以了。

##### 6.2.2.2 AbstractExecutionThreadService

AbstractExecutionThreadService 通过单线程处理启动、运行、和关闭等操作。你必须重载run()方法，同时需要能响应停止服务的请求。具体的实现可以在一个循环内做处理：

1. **protected** **void** run() throws Exception {
2. **while** (isRunning()){
4. }
5. }

还可以重载triggerShutdown()方法让run()方法结束返回

重载startUp()和shutDown()方法是可选的，不影响服务本身状态的管理。

doStart()内部会调用startUp()方法，创建一个线程、然后在线程内调用run()方法。stop()会调用 triggerShutdown()方法并且等待线程终止

##### 6.2.2.3 AbstractScheduledService

AbstractScheduledService类用于在运行时处理一些周期性的任务。子类可以实现 runOneIteration()方法定义一个周期执行的任务，以及相应的startUp()和shutDown()方法。为了能够描述执行周期，你需要实现scheduler()方法。

通常情况下，你可以使用AbstractScheduledService.Scheduler类提供的两种调度器：newFixedRateSchedule(initialDelay, delay, TimeUnit) 和newFixedDelaySchedule(initialDelay, delay, TimeUnit)，类似于JDK并发包中ScheduledExecutorService类提供的两种调度方式。如要自定义schedules则可以使用 CustomScheduler类来辅助实现；

##### 6.2.2.4 AbstractService

如需要自定义的线程管理、可以通过扩展 AbstractService类来实现。一般情况下、使用上面的几个实现类就已经满足需求了，但如果在服务执行过程中有一些特定的线程处理需求、则建议继承AbstractService类。

继承AbstractService方法必须实现两个方法.

doStart(): 首次调用startAsync()时会同时调用doStart(),doStart()内部需要处理所有的初始化工作、如果启动成功则调用notifyStarted()方法；启动失败则调用notifyFailed()

doStop(): 首次调用stopAsync()会同时调用doStop(),doStop()要做的事情就是停止服务，如果停止成功则调用 notifyStopped()方法；停止失败则调用 notifyFailed()方法。

doStart和doStop方法的实现需要考虑下性能，尽可能的低延迟。如果初始化的开销较大，如读文件，打开网络连接，或者其他任何可能引起阻塞的操作，建议移到另外一个单独的线程去处理。

##### 6.2.2.5 使用ServiceManager

除了对Service接口提供基础的实现类，Guava还提供了 ServiceManager类使得涉及到多个Service集合的操作更加容易。通过实例化ServiceManager类来创建一个Service集合，可以通过以下方法来管理它们：

* startAsync() ： 将启动所有被管理的服务。如果当前服务的状态都是NEW的话、那么你只能调用该方法一次、这跟 Service#startAsync()是一样的。
* stopAsync() ：将停止所有被管理的服务。
* addListener ：会添加一个ServiceManager.Listener，在服务状态转换中会调用该Listener
* awaitHealthy() ：会等待所有的服务达到Running状态
* awaitStopped()：会等待所有服务达到终止状态

检测类的方法有：

* isHealthy() ：如果所有的服务处于Running状态、会返回True
* servicesByState()：以状态为索引返回当前所有服务的快照
* startupTimes() ：返回一个Map对象，记录被管理的服务启动的耗时、以毫秒为单位，同时Map默认按启动时间排序。

建议整个服务的生命周期都能通过ServiceManager来管理，不过即使状态转换是通过其他机制触发的、也不影响ServiceManager方法的正确执行。例如：当一个服务不是通过startAsync()、而是其他机制启动时，listeners 仍然可以被正常调用、awaitHealthy()也能够正常工作。ServiceManager 唯一强制的要求是当其被创建时所有的服务必须处于New状态。

举个栗子：

AbstractScheduledServiceDemo 类

1. **public** **class** AbstractScheduledServiceDemo **extends** AbstractScheduledService {
2. @Override
3. **protected** **void** runOneIteration() **throws** Exception {
4. System.out.println("schedule Task");
5. }
7. @Override
8. **protected** Scheduler scheduler() {
9. **return** AbstractScheduledService.Scheduler.newFixedRateSchedule(1,2, TimeUnit.SECONDS);
10. }
11. }
12. AbstractScheduledServiceDemo de = **new** AbstractScheduledServiceDemo();
13. ServiceManager serviceManager = **new** ServiceManager(Arrays.asList(de));
14. serviceManager.startAsync();

能够实现间隔2秒输出一个schedule task

## 字符串（Strings）

### 连接器

用分隔符把字符串序列连接起来也可能会遇上不必要的麻烦。如果字符串序列中含有null，那连接操作会更难。

1. Joiner joiner = Joiner.on(";").skipNulls();
2. String str = joiner.join("Harry", **null**, "ron", "jack");
3. //Harry;ron;jack
4. System.out.println(str);

useForNull(String)方法可以给定某个字符串来替换null，而不像skipNulls()方法是直接忽略null。

1. Joiner bob = Joiner.on(";").useForNull("Bob");
2. String str2 = bob.join("Harry", **null**, "ron", "jack");
3. //Harry;Bob;ron;jack
4. System.out.println(str2);

Joiner也可以用来连接对象类型，在这种情况下，它会把对象的toString()值连接起来。也可以是Iterable类型的对象。

1. String join = Joiner.on(",").join(Arrays.asList(1, 2, 6));
2. //1,2,6
3. System.out.println(join);

### 7.2 拆分器

  Splitter使用令人放心的、直白的流畅API模式对这些混乱的特性作了完全的掌控

1. // JDK内建的字符串拆分工具
2. String[] split = "foo,bar,,qux,".split(",");
3. //输出的是 foo bar "" qux
4. Arrays.asList(split).forEach(System.out::println);
6. Iterable<String> strings = Splitter.on(",")
7. .trimResults()
8. .omitEmptyStrings()
9. .split(",foo,bar,,qux,");
10. //[foo, bar, qux]
11. System.out.println(strings);

**拆分器工厂**

|  |  |  |
| --- | --- | --- |
| 方法 | 描述 | 范例 |
| Splitter.on(char) | 按单个字符拆分 | Splitter.on(‘;’) |
| Splitter.on(CharMatcher) | 按字符匹配器拆分 | Splitter.on(  CharMatcher.BREAKING\_WHITESPACE) |
| Splitter.on(String) | 按字符串拆分 | Splitter.on(“, “) |
| Splitter.on(Pattern) Splitter.onPattern(String) | 按正则表达式拆分 | Splitter.onPattern(“\r?\n”) |
| Splitter.fixedLength(int) | 按固定长度拆分；最后一段可能比给定长度短，但不会为空。 | Splitter.fixedLength(3) |

**拆分器修饰符**

|  |  |
| --- | --- |
| 方法 | 描述 |
| omitEmptyStrings() | 从结果中自动忽略空字符串 |
| trimResults() | 移除结果字符串的前导空白和尾部空白 |
| trimResults(CharMatcher) | 给定匹配器，移除结果字符串的前导匹配字符和尾部匹配字符 |
| limit(int) | 限制拆分出的字符串数量 |

如果你想要拆分器返回List，只要使用Lists.newArrayList(splitter.split(string))或类似方法。

注意：splitter实例总是不可变的。用来定义splitter目标语义的配置方法总会返回一个新的splitter实例。这使得splitter实例都是线程安全的，你可以将其定义为static final常量。

### 字符匹配器

CharMatcher实例代表着某一类字符，如数字或空白字符。事实上来说，CharMatcher实例就是对字符的布尔判断

CharMatcher的好处更在于它提供了一系列方法，可以对字符作特定类型的操作：修剪[trim]、折叠[collapse]、移除[remove]、保留[retain]等等

**举个栗子：**

1. //只保留数字字符
2. String digit = CharMatcher.digit().retainFrom("dasfaf1245adfdas4234");
3. // 12454234
4. System.out.println(digit);
5. //去除两端的空格，并把中间的连续空格替换成单个空格
6. String tt = CharMatcher.whitespace().trimAndCollapseFrom(" te stt ", ';');
7. // te;stt
8. System.out.println(tt);

CharMatcher中常见的工厂方法能够满足需求：

any()、none()、whitespace()、breakingWhitespace()、ascii()、javaIsoControl()

其他获取字符匹配器的常见方法包括：

|  |  |
| --- | --- |
| 方法 | 描述 |
| anyOf(CharSequence) | 枚举匹配字符。如CharMatcher.anyOf(“aeiou”)匹配小写英语元音 |
| is(char) | 给定单一字符匹配。 |
| inRange(char, char) | 给定字符范围匹配，如CharMatcher.inRange(‘a’, ‘z’) |

此外：CharMatcher还有negate()、and(CharMatcher)和or(CharMatcher)方法。

**使用字符匹配器**

CharMatcher提供了多种多样的方法操作CharSequence中的特定字符。其中最常用的罗列如下：

|  |  |
| --- | --- |
| 方法 | 描述 |
| collapseFrom(CharSequence, char) | 把每组连续的匹配字符替换为特定字符。如WHITESPACE.collapseFrom(string, ‘ ‘)把字符串中的连续空白字符替换为单个空格。 |
| matchesAllOf(CharSequence) | 测试是否字符序列中的所有字符都匹配。 |
| removeFrom(CharSequence) | 从字符序列中移除所有匹配字符。 |
| retainFrom(CharSequence) | 在字符序列中保留匹配字符，移除其他字符。 |
| trimFrom(CharSequence) | 移除字符序列的前导匹配字符和尾部匹配字符。 |
| replaceFrom(CharSequence, CharSequence) | 用特定字符序列替代匹配字符。 |

所有这些方法返回String，除了matchesAllOf返回的是boolean。

### 字符集

Charsets针对所有Java平台都要保证支持的六种字符集（US-ASCII、ISO-8859-1、UTF-8、UTF-16BE、UTF-16LE、UTF-16）提供了常量引用。尝试使用这些常量，而不是通过名称获取字符集实例。

1. String str = "test";
2. **try** {
3. //不要这样使用
4. **byte**[] bytes = str.getBytes("UTF-8");
5. } **catch** (UnsupportedEncodingException e) {
6. e.printStackTrace();
7. }
9. **byte**[] bytes = str.getBytes(Charsets.UTF\_8);

### 大小写格式

CaseFormat被用来方便地在各种ASCII大小写规范间转换字符串——比如，编程语言的命名规范。CaseFormat支持的格式如下：

|  |  |
| --- | --- |
| 格式 | 范例 |
| LOWER\_CAMEL | lowerCamel |
| LOWER\_HYPHEN | lower-hyphen |
| LOWER\_UNDERSCORE | lower\_underscore |
| UPPER\_CAMEL | UpperCamel |
| UPPER\_UNDERSCORE | UPPER\_UNDERSCORE |

1. String str = CaseFormat.UPPER\_UNDERSCORE.to(CaseFormat.LOWER\_CAMEL, "CONSTANT\_NAME");
2. //constantName
3. System.out.println(str);

CaseFormat在某些时候尤其有用，比如编写代码生成器的时候

## 原生类型（Primitives）

Java的原生类型就是指基本类型：byte、short、int、long、float、double、char和boolean。

原生类型不能当作对象或泛型的类型参数使用，这意味着许多通用方法都不能应用于它们。Guava提供了若干通用工具，包括原生类型数组与集合API的交互，原生类型和字节数组的相互转换，以及对某些原生类型的无符号形式的支持。

|  |  |
| --- | --- |
| 原生类型 | Guava工具类（都在com.google.common.primitives包） |
| byte | Bytes, SignedBytes, UnsignedBytes |
| short | Shorts |
| int | Ints, UnsignedInteger, UnsignedInts |
| long | Longs, UnsignedLong, UnsignedLongs |
| float | Floats |
| double | Doubles |
| char | Chars |
| boolean | Booleans |

### 原生类型方法

原生类型提供了很多有用的方法。

举个栗子：

1. //常用工具
2. //把数组转为相应包装类的List
3. List<Integer> integers = Ints.asList(1, 2, 3, 4);
4. List<Float> floats = Floats.asList(1, 2, 3, 4);
6. **int**[] ints = {1, 2, 3};
7. //串联多个原生类型数组
8. **int**[] concat = Ints.concat(ints, **new** **int**[]{5, 6, 7});
9. //判断原生类型数组是否包含给定值
10. **boolean** contains = Ints.contains(**new** **int**[]{4, 5, 6}, 9);
11. //false
12. System.out.println(contains);
13. // 把数组用给定分隔符连接为字符串
14. String join = Ints.join(",", **new** **int**[]{1, 2, 3});
15. //1,2,3
16. System.out.println(join);
17. //数组中最小的值
18. **int** min = Ints.min(**new** **int**[]{5, 6, 7, 8, 9});
19. //5
20. System.out.println(min);
21. //数组中最大的值
22. **int** max = Ints.max(**new** **int**[]{4, 5, 6, 7, 8});
23. //8
24. System.out.println(max);
25. //传统的Comparator.compare方法，但针对原生类型。JDK7的原生类型包装类也提供这样的方法
26. **int** compare = Ints.compare(1, 2);
27. //-1 （-1小于  等于 0  大于 1）
28. System.out.println(compare);

### 8.2 字节转换方法

**举个栗子：**

1. //字节转换方法
2. //常量：表示该原生类型需要的字节数
3. **int** bytes = Ints.BYTES;
4. //接受Prims.BYTES个字节参数，按大字节序返回原生类型值
5. **int** i = Ints.fromBytes((**byte**) 1, (**byte**)2,(**byte**) 3,(**byte**) 4);
6. //00000001 00000010 00000011 00000100(2) =  16909060
7. System.out.println(i);
8. **byte**[] bytes1 = Ints.toByteArray(32);

### 8.3 无符号支持

**举个栗子：**

1. //无符号支持
3. //按无符号十进制解析字符串
4. **int** i1 = UnsignedInts.parseUnsignedInt("12");
5. //12
6. System.out.println(i1);
7. //数字按无符号十进制转为字符串
8. String s = UnsignedInts.toString(-56);
9. //4294967240 (二进制数为 11111111111111111111111111001000 )
10. System.out.println(s);

## 区间（Ranges）

区间，有时也称为范围，是特定域中的凸性（非正式说法为连续的或不中断的）部分。

### 区间的构建

**举个栗子：**

1. //构造区间
2. //(1..2)
3. Range<Integer> open = Range.open(1, 2);
4. //[1..2]
5. Range<Integer> closed = Range.closed(1, 2);
6. //[1,2)
7. Range<Integer> integerRange = Range.closedOpen(1, 2);
8. //(1,2]
9. Range<Integer> integerRange1 = Range.openClosed(1, 2);
10. // (1..+∞)
11. Range<Integer> integerRange2 = Range.greaterThan(1);
12. //[1..+∞)
13. Range<Integer> integerRange3 = Range.atLeast(1);
14. //(-∞..1)
15. Range<Integer> integerRange4 = Range.lessThan(1);
16. //(-∞..1]
17. Range<Integer> integerRange5 = Range.atMost(1);
18. //(-∞..+∞)
19. Range<Comparable<?>> all = Range.all();
20. // 有界区间 [1,2]
21. Range<Integer> range = Range.range(1, BoundType.CLOSED, 2, BoundType.CLOSED);
22. // 无上界区间 [1..+∞)
23. Range<Integer> integerRange6 = Range.downTo(1, BoundType.CLOSED);
24. // 无下界区间 (-∞..1]
25. Range<Integer> integerRange7 = Range.upTo(1, BoundType.CLOSED);

### 9.2 区间的运算

#### 9.2.1 查询运算

Range的基本运算是它的contains(C) 方法，和你期望的一样，它用来区间判断是否包含某个值。任何Range实例也都支持containsAll(Iterable<? extends C>)方法。

**举个栗子：**

1. **boolean** contains = Range.closed(1, 3).contains(4);
2. //false
3. System.out.println(contains);
4. **boolean** contains1 = Range.closed(1, 4).containsAll(Ints.asList(2, 3, 4));
5. //true
6. System.out.println(contains1);

Range类提供了以下方法来 查看区间的端点：

* hasLowerBound()和hasUpperBound()：判断区间是否有特定边界，或是无限的；
* lowerBoundType()和upperBoundType()：返回区间边界类型，CLOSED或OPEN；如果区间没有对应的边界，抛出IllegalStateException；
* lowerEndpoint()和upperEndpoint()：返回区间的端点值；如果区间没有对应的边界，抛出IllegalStateException；
* isEmpty()：判断是否为空区间。

**举个栗子：**

1. **boolean** empty = Range.closed(4, 4 ).isEmpty();
2. //false
3. System.out.println(empty);
4. Integer integer = Range.closed(3, 10).lowerEndpoint();
5. // 3 返回区间的端点
6. System.out.println(integer);
7. BoundType boundType = Range.open(3, 10).lowerBoundType();
8. //OPEN
9. System.out.println(boundType);

#### 9.2.1 关系运算

**包含：** 区间之间的最基本关系就是包含[encloses(Range)]：如果内区间的边界没有超出外区间的边界，则外区间包含内区间。包含判断的结果完全取决于区间端点的比较！如：[3..6] 包含[4..5] ；(3..6) 包含(3..6) ；

**举个栗子：**

1. //包含运算
2. **boolean** encloses = Range.closed(3, 6).encloses(Range.closed(4, 6));
3. //true
4. System.out.println(encloses);
5. **boolean** encloses1 = Range.closed(3, 5).encloses(Range.closed(4, 6));
6. //false
7. System.out.println(encloses1);

**相连：**Range.isConnected(Range)判断区间是否是相连的。具体来说，isConnected测试是否有区间同时包含于这两个区间，这等同于数学上的定义”两个区间的并集是连续集合的形式”（空区间的特殊情况除外）。

**举个栗子：**

1. //相连
2. **boolean** connected = Range.closed(3, 5).isConnected(Range.open(5, 10));
3. //true
4. System.out.println(connected);
5. **boolean** connected1 = Range.closed(1, 5).isConnected(Range.closed(6, 10));
6. //false
7. System.out.println(connected1);

**交集：**Range.intersection(Range)返回两个区间的交集：既包含于第一个区间，又包含于另一个区间的最大区间。当且仅当两个区间是相连的，它们才有交集。如果两个区间没有交集，该方法将抛出IllegalArgumentException。

**举个栗子：**

1. //交集
2. Range<Integer> intersection = Range.closed(3, 5).intersection(Range.open(5, 10));
3. //(5..5]
4. System.out.println(intersection);
5. //java.lang.IllegalArgumentException: Invalid range: [6..5]
6. Range<Integer> intersection1 = Range.closed(1, 5).intersection(Range.closed(6, 10));
7. // System.out.println(intersection1);

**跨区域：**Range.span(Range)返回”同时包括两个区间的最小区间”，如果两个区间相连，那就是它们的并集。

**举个栗子：**

1. //跨区域
2. Range<Integer> span = Range.closed(3, 5).span(Range.open(5, 10));
3. //[3..10)
4. System.out.println(span);
5. Range<Integer> span1 = Range.closed(1, 5).span(Range.closed(3, 9));
6. //[1..9]
7. System.out.println(span1);

## I/O

### 字节流和字符流

Guava使用术语”流”来表示可关闭的，并且在底层资源中有位置状态的I/O数据流。术语”字节流”指的是InputStream或OutputStream，”字符流”指的是Reader 或Writer（虽然他们的接口Readable 和Appendable被更多地用于方法参数）。相应的工具方法分别在ByteStreams 和CharStreams中。

大多数Guava流工具一次处理一个完整的流，并且/或者为了效率自己处理缓冲。还要注意到，接受流为参数的Guava方法不会关闭这个流：关闭流的职责通常属于打开流的代码块。

列举一些常见的工具方法：

|  |  |
| --- | --- |
| ByteStreams | CharStreams |
| byte[] toByteArray(InputStream) | String toString(Readable) |
| N/A | List<String> readLines(Readable) |
| long copy(InputStream, OutputStream) | long copy(Readable, Appendable) |
| void readFully(InputStream, byte[]) | N/A |
| void skipFully(InputStream, long) | void skipFully(Reader, long) |
| OutputStream nullOutputStream() | Writer nullWriter() |

**举个栗子：**

1. //实现文件的复制
2. FileInputStream inputStream = **new** FileInputStream(**new** File("file.txt"));
3. ByteStreams.copy(inputStream,**new** FileOutputStream(**new** File("fileCopy.txt")));

### 源与汇（Sources and sinks）

通常我们都会创建I/O工具方法，这样可以避免在做基础运算时总是直接和流打交道。然而，流工具方法的创建经常最终导致散落各处的相似方法，每个方法读取不同类型的源

[sources]或写入不同类型的汇[sink]。例如，Guava中的Resources.toByteArray(URL)和Files.toByteArray(File)做了同样的事情，只不过数据源一个是URL，一个是文件

为了解决这个问题，Guava有一系列关于源与汇的抽象。源或汇指某个你知道如何从中打开流的资源，比如File或URL。源是可读的，汇是可写的。此外，源与汇按照字节和字符划分类型

|  |  |  |
| --- | --- | --- |
|  | 字节 | 字符 |
| 读 | ByteSource | CharSource |
| 写 | ByteSink | CharSink |

源与汇API的好处是它们提供了通用的一组操作。比如，一旦你把数据源包装成了ByteSource，无论它原先的类型是什么，你都得到了一组按字节操作的方法。

**创建源与汇：**

|  |  |
| --- | --- |
| 字节 | 字符 |
| Files.asByteSource(File) | Files.asCharSource(File, Charset) |
| Files.asByteSink(File, FileWriteMode...) | Files.asCharSink(File, Charset, FileWriteMode...) |
| Resources.asByteSource(URL) | Resources.asCharSource(URL, Charset) |
| ByteSource.wrap(byte[]) | CharSource.wrap(CharSequence) |
| ByteSource.concat(ByteSource...) | CharSource.concat(CharSource...) |
| ByteSource.slice(long, long) | N/A |
| N/A | ByteSource.asCharSource(Charset) |
| N/A | ByteSink.asCharSink(Charset) |

**注：**把已经打开的流（比如InputStream）包装为源或汇听起来是很有诱惑力的，但是应该避免这样做。源与汇的实现应该在每次openStream()方法被调用时都创建一个新的流。始终创建新的流可以让源或汇管理流的整个生命周期，并且让多次调用openStream()返回的流都是可用的。此外，如果你在创建源或汇之前创建了流，你不得不在异常的时候自己保证关闭流，这压根就违背了发挥源与汇API优点的初衷。

**使用源与汇**

所有源与汇都有一些方法用于打开新的流用于读或写。默认情况下，其他源与汇操作都是先用这些方法打开流，然后做一些读或写，最后保证流被正确地关闭了。这些方法列举如下：

* openStream()：根据源与汇的类型，返回InputStream、OutputStream、Reader或者Writer。
* openBufferedStream()：根据源与汇的类型，返回InputStream、OutputStream、BufferedReader或者BufferedWriter。返回的流保证在必要情况下做了缓冲。例如，从字节数组读数据的源就没有必要再在内存中作缓冲，这就是为什么该方法针对字节源不返回BufferedInputStream。字符源属于例外情况，它一定返回BufferedReader，因为BufferedReader中才有readLine()方法。

**源操作**

|  |  |
| --- | --- |
| 字节源 | 字符源 |
| byte[] read() | String read() |
| N/A | ImmutableList<String> readLines() |
| N/A | String readFirstLine() |
| long copyTo(ByteSink) | long copyTo(CharSink) |
| long copyTo(OutputStream) | long copyTo(Appendable) |
| long size() (in bytes) | N/A |
| boolean isEmpty() | boolean isEmpty() |
| boolean contentEquals(ByteSource) | N/A |
| HashCode hash(HashFunction) | N/A |

**汇操作**

|  |  |
| --- | --- |
| 字节汇 | 字符汇 |
| void write(byte[]) | void write(CharSequence) |
| long writeFrom(InputStream) | long writeFrom(Readable) |
| N/A | void writeLines(Iterable<? extends CharSequence>) |
| N/A | void writeLines(Iterable<? extends CharSequence>, String) |

**举个栗子：**

1. File file = **new** File("file.txt");
2. //读取文件内容第一行
3. String s = Files.asCharSource(file, Charsets.UTF\_8).readFirstLine();
4. System.out.println(s);
5. //分割文件中的数据
6. HashMultiset<String> strings = HashMultiset.create(
7. Splitter.on(CharMatcher.whitespace())
8. .trimResults()
9. .omitEmptyStrings()
10. .split(Files.asCharSource(file, Charsets.UTF\_8).read()));
12. System.out.println(strings);
13. //对文件内容 SHA-1运算
14. HashCode hash = Files.asByteSource(file).hash(Hashing.sha1());
15. System.out.println(hash);
17. //从网络上获取数据保存到文件中
18. Resources.asByteSource(**new** URL("http://www.planetb.ca/projects/syntaxHighlighter/popup.php")).copyTo(Files.asByteSink(file));

**文件复制**

1. //JDK IO 一般复制文件方式
2. FileInputStream fileInputStream = **new** FileInputStream(**new** File("file.txt"));
3. FileOutputStream fileOutputStream = **new** FileOutputStream(**new** File("fileCopy.txt"));
4. byte[] buff = **new** byte[1024];
5. **int** len = 0;
6. **while** ((len = fileInputStream.read(buff)) != -1) {
7. fileOutputStream.write(buff,0,len);
8. }
9. fileOutputStream.flush();
10. fileInputStream.close();
11. fileOutputStream.close();
13. //实现文件复制 方式1
14. ByteSource byteSource = Files.asByteSource(**new** File("file.txt"));
15. **long** l = byteSource.copyTo(Files.asByteSink(**new** File("fileCopy.txt"), FileWriteMode.APPEND));
17. //实现文件复制 方式2
18. FileOutputStream outputStream = **new** FileOutputStream(**new** File("fileCopy.txt"));
19. Files.copy(**new** File("file.txt"),outputStream);

### 文件操作

除了创建文件源和文件的方法，Files类还包含了若干你可能感兴趣的便利方法。

|  |  |
| --- | --- |
| createParentDirs(File) | 必要时为文件创建父目录 |
| getFileExtension(String) | 返回给定路径所表示文件的扩展名 |
| getNameWithoutExtension(String) | 返回去除了扩展名的文件名 |
| simplifyPath(String) | 规范文件路径，并不总是与文件系统一致，请仔细测试 |
| fileTreeTraverser() | 返回TreeTraverser用于遍历文件树 |

**举个栗子：**

1. String fileExtension = Files.getFileExtension("file.txt.txt");
2. //txt
3. System.out.println(fileExtension);
4. String s1 = Files.simplifyPath("file.txt");
5. //file.txt
6. System.out.println(s1);

## 散列（Hashing）

### 概述

Java内建的散列码[hash code]概念被限制为32位，并且没有分离散列算法和它们所作用的数据，因此很难用备选算法进行替换。此外，使用Java内建方法实现的散列码通常是劣质的，部分是因为它们最终都依赖于JDK类中已有的劣质散列码。

Object.hashCode往往很快，但是在预防碰撞上却很弱，也没有对分散性的预期。这使得它们很适合在散列表中运用，因为额外碰撞只会带来轻微的性能损失，同时差劲的分散性也可以容易地通过再散列来纠正（Java中所有合理的散列表都用了再散列方法）。然而，在简单散列表以外的散列运用中，Object.hashCode几乎总是达不到要求——因此，有了com.google.common.hash包。

### 散列包的组成

**举个栗子：**

1. HashFunction hashFunction = Hashing.md5();
2. HashCode hash = hashFunction.newHasher()
3. .putLong(124512312L)
4. .putString("fdjsafkljdas", Charsets.UTF\_8)
5. .hash();
6. //facb9a2aa4ea0ae4a95326bcd19fe92e
7. System.out.println(hash.toString());
8. //714787834
9. System.out.println(hash.hashCode());

**HashFunction:**

HashFunction是一个单纯的（引用透明的）、无状态的方法，它把任意的数据块映射到固定数目的位指，并且保证相同的输入一定产生相同的输出，不同的输入尽可能产生不同的输出。

**Hasher:**

HashFunction的实例可以提供有状态的Hasher，Hasher提供了流畅的语法把数据添加到散列运算，然后获取散列值。Hasher可以接受所有原生类型、字节数组、字节数组的片段、字符序列、特定字符集的字符序列等等，或者任何给定了Funnel实现的对象。

Hasher实现了PrimitiveSink接口，这个接口为接受原生类型流的对象定义了fluent风格的API

**Funnel:**

Funnel描述了如何把一个具体的对象类型分解为原生字段值，从而写入PrimitiveSink

举个栗子：

1. **public** **class** Person {
2. **private** **int** id;
3. **private** String firstName;
4. **private** String lastName;
5. **private** **int** age;
6. //省略构造器 ，set get 方法
7. }
8. Funnel<Person> funnel = **new** Funnel<Person>() {
9. @Override
10. **public** **void** funnel(Person person, PrimitiveSink into) {
11. into.putInt(person.getId())
12. .putString(person.getFirstName(), Charsets.UTF\_8)
13. .putString(person.getLastName(), Charsets.UTF\_8)
14. .putInt(person.getAge());
15. }
16. };
17. HashFunction hashFunction = Hashing.md5();
18. HashCode hash = hashFunction.newHasher()
19. .putObject(**new** Person(1,"jack","lam",20), funnel)
20. .hash();
21. //9bd9a7155e98d6dbd3ade93d3d35f79b
22. System.out.println(hash.toString());
23. //363321755
24. System.out.println(hash.hashCode());

**注：**putString(“abc”, Charsets.UTF\_8).putString(“def”, Charsets.UTF\_8)完全等同于putString(“ab”, Charsets.UTF\_8).putString(“cdef”, Charsets.UTF\_8)，因为它们提供了相同的字节序列。这可能带来预料之外的散列冲突。增加某种形式的分隔符有助于消除散列冲突。

**HashCode:**

一旦Hasher被赋予了所有输入，就可以通过hash()方法获取HashCode实例（多次调用hash()方法的结果是不确定的）。HashCode可以通过asInt()、asLong()、asBytes()方法来做相等性检测，此外，writeBytesTo(array, offset, maxLength)把散列值的前maxLength字节写入字节数组。

### 布鲁姆过滤器[BloomFilter]

布鲁姆过滤器是哈希运算的一项优雅运用，它可以简单地基于Object.hashCode()实现。简而言之，布鲁姆过滤器是一种概率数据结构，它允许你检测某个对象是一定不在过滤器中，还是可能已经添加到过滤器了。布鲁姆过滤器的介绍详见[附录1](#_1._布鲁姆过滤器)

Guava散列包有一个内建的布鲁姆过滤器实现，你只要提供Funnel就可以使用它。你可以使用create(Funnel funnel, int expectedInsertions, double falsePositiveProbability)方法获取BloomFilter<T>，缺省误检率[falsePositiveProbability]为3%。BloomFilter<T>提供了boolean mightContain(T) 和void put(T)，它们的含义都不言自明了。

举个栗子：

1. ArrayList<Person> objects = **new** ArrayList<>();
3. objects.add(**new** Person(1,"jack","cc",20));
4. objects.add(**new** Person(2,"jack","cc",20));
5. objects.add(**new** Person(3,"jack","cc",20));
6. objects.add(**new** Person(4,"jack","cc",20));
7. objects.add(**new** Person(5,"jack","cc",20));
9. Funnel<Person> funnel = **new** Funnel<Person>() {
10. @Override
11. **public** **void** funnel(Person person, PrimitiveSink into) {
12. into.putInt(person.getId())
13. .putString(person.getFirstName(), Charsets.UTF\_8)
14. .putString(person.getLastName(), Charsets.UTF\_8)
15. .putInt(person.getAge());
16. }
17. };
18. BloomFilter<Person> friends = BloomFilter.create(funnel, 500, 0.01);
19. **for** (Person p : objects) {
20. friends.put(p);
21. }
22. Person person = **null**;
23. **for** (**int** i = 0; i < 1000000; i++) {
24. person = **new** Person(i, "jack", "cc", 20);
25. **if** (friends.mightContain(person)){
26. System.out.println("id 为" +i+ "  存在");
27. }**else** {
28. //
29. }
30. }
31. //output 除了1-5 还有两条输出
32. //id 为579534  存在
33. //id 为652363  存在

### Hashing类

Hashing类提供了若干散列函数，以及运算HashCode对象的工具方法。

**已提供的散列函数**

md5()、murmur3\_128()、murmur3\_32()、sha1()、sha256()、sha512()、 goodFastHash(int bits)

**HashCode运算**

|  |  |
| --- | --- |
| 方法 | 描述 |
| HashCode combineOrdered( Iterable<HashCode>) | 以有序方式联接散列码，如果两个散列集合用该方法联接出的散列码相同，那么散列集合的元素可能是顺序相等的 |
| HashCode combineUnordered( Iterable<HashCode>) | 以无序方式联接散列码，如果两个散列集合用该方法联接出的散列码相同，那么散列集合的元素可能在某种排序下是相等的 |
| int consistentHash( HashCode, int buckets) | 为给定的”桶”大小返回一致性哈希值。当”桶”增长时，该方法保证最小程度的一致性哈希值变化。 |

## 事件总线 (EventBus)

略

## 数学运算（Math）

### 为什么使用Guava Math

* Guava Math针对各种不常见的溢出情况都有充分的测试；对溢出语义，Guava文档也有相应的说明；如果运算的溢出检查不能通过，将导致快速失败；
* Guava Math的性能经过了精心的设计和调优；虽然性能不可避免地依据具体硬件细节而有所差异，但Guava Math的速度通常可以与Apache Commons的MathUtils相比，在某些场景下甚至还有显著提升；
* Guava Math在设计上考虑了可读性和正确的编程习惯；IntMath.log2(x, CEILING) 所表达的含义，即使在快速阅读时也是清晰明确的。而32-Integer.numberOfLeadingZeros(x – 1)对于阅读者来说则不够清晰。

### 13.2 整数运算

Guava Math主要处理三种整数类型：int、long和BigInteger。这三种类型的运算工具类分别叫做IntMath、LongMath和BigIntegerMath。

Guava Math提供了若干有溢出检查的运算方法：结果溢出时，这些方法将快速失败而不是忽略溢出

|  |  |
| --- | --- |
| IntMath.checkedAdd | LongMath.checkedAdd |
| IntMath.checkedSubtract | LongMath.checkedSubtract |
| IntMath.checkedMultiply | LongMath.checkedMultiply |
| IntMath.checkedPow | LongMath.checkedPow |

**举个栗子：**

1. //java.lang.ArithmeticException: overflow: checkedAdd(2147483647, 2147483647)
2. **int** i = IntMath.checkedAdd(Integer.MAX\_VALUE, Integer.MAX\_VALUE);

### 13.3 实数运算

I ntMath、LongMath和BigIntegerMath提供了很多实数运算的方法，并把最终运算结果舍入成整数。这些方法接受一个java.math.RoundingMode枚举值作为舍入的模式：

|  |  |
| --- | --- |
| DOWN | 向零方向舍入（去尾法） |
| UP | 远离零方向舍入 |
| FLOOR | 向负无限大方向舍入 |
| CEILING | 向正无限大方向舍入 |
| UNNECESSARY | 不需要舍入，如果用此模式进行舍入，应直接抛出ArithmeticException |
| HALF\_UP | 向最近的整数舍入，其中x.5远离零方向舍入 |
| HALF\_DOWN | 向最近的整数舍入，其中x.5向零方向舍入 |
| HALF\_EVEN | 向最近的整数舍入，其中x.5向相邻的偶数舍入 |

这些方法旨在提高代码的可读性，例如，divide(x, 3, CEILING) 即使在快速阅读时也是清晰。此外，这些方法内部采用构建整数近似值再计算的实现，除了在构建sqrt（平方根）运算的初始近似值时有浮点运算，其他方法的运算全过程都是整数或位运算，因此性能上更好。

|  |  |  |  |
| --- | --- | --- | --- |
| 运算 | IntMath | LongMath | BigIntegerMath |
| 除法 | divide(int, int, RoundingMode) | divide(long, long, RoundingMode) | divide(BigInteger, BigInteger, RoundingMode) |
| 2为底的对数 | log2(int, RoundingMode) | log2(long, RoundingMode) | log2(BigInteger, RoundingMode) |
| 10为底的对数 | log10(int, RoundingMode) | log10(long, RoundingMode) | log10(BigInteger, RoundingMode) |
| 平方根 | sqrt(int, RoundingMode) | sqrt(long, RoundingMode) | sqrt(BigInteger, RoundingMode) |

**举个栗子：**

1. **int** sqrt = IntMath.sqrt(5, RoundingMode.DOWN);
2. **int** sqrt1 = IntMath.sqrt(5, RoundingMode.UP);
3. **int** divide = IntMath.divide(7, 3, RoundingMode.CEILING);
4. //2
5. System.out.println(sqrt);
6. //3
7. System.out.println(sqrt1);
8. //3
9. System.out.println(divide);

Guava还另外提供了一些有用的运算函数

|  |  |  |  |
| --- | --- | --- | --- |
| 运算 | IntMath | LongMath | BigIntegerMath |
| 最大公约数 | gcd(int, int) | gcd(long, long) | BigInteger.gcd(BigInteger) |
| 取模 | mod(int, int) | mod(long, long) | BigInteger.mod(BigInteger) |
| 取幂 | pow(int, int) | pow(long, int) | BigInteger.pow(int) |
| 是否2的幂 | isPowerOfTwo(int) | isPowerOfTwo(long) | isPowerOfTwo(BigInteger) |
| 阶乘 | factorial(int) | factorial(int) | factorial(int) |
| 二项式系数\* | binomial(int, int) | binomial(int, int) | binomial(int, int) |

**注：**

BigInteger的最大公约数和取模运算由JDK提供

阶乘和二项式系数的运算结果如果溢出，则返回MAX\_VALUE

**举个栗子：**

1. // 6
2. System.out.println(IntMath.gcd(18, 24));
3. // 13 % 3 = 1
4. System.out.println(IntMath.mod(13, 3));
5. // 2 ^3 = 8
6. System.out.println(IntMath.pow(2, 3));
7. //true
8. System.out.println(IntMath.isPowerOfTwo(1024));
9. //24 = 1 \* 2 \* 3 \* 4
10. System.out.println(IntMath.factorial(4));

### 13.4 浮点数运算

JDK比较彻底地涵盖了浮点数运算，但Guava在DoubleMath类中也提供了一些有用的方法。

|  |  |
| --- | --- |
| 方法 | 描述 |
| isMathematicalInteger(double) | 判断该浮点数是不是一个整数 |
| roundToInt(double, RoundingMode) | 舍入为int；对无限小数、溢出抛出异常 |
| roundToLong(double, RoundingMode) | 舍入为long；对无限小数、溢出抛出异常 |
| roundToBigInteger(double, RoundingMode) | 舍入为BigInteger；对无限小数抛出异常 |
| log2(double, RoundingMode) | 2的浮点对数，并且舍入为int，比JDK的Math.log(double) 更快 |

**举个栗子：**

1. // true
2. System.out.println(DoubleMath.isMathematicalInteger(13.0));
3. // 13
4. System.out.println(DoubleMath.roundToInt(13.2, RoundingMode.FLOOR));

## 反射（Reflection）

### TypeToken

由于类型擦拭，不能够在运行时传递泛型类对象。即使强制转换它们，并假装这些对象是有泛型的，但实际上他们并没有。

**举个栗子：**

1. ArrayList<String> string = Lists.newArrayList();
2. ArrayList<Integer> integers = Lists.newArrayList();
4. //true  即使ArrayList<String> 不同于 ArrayList<Integer>
5. System.out.println(string.getClass().isAssignableFrom(integers.getClass()));

Guava提供了TypeToken，它使用了基于反射的技巧，甚至运行时都能够巧妙的操作和查询泛型类型。

Java不能在运行时保留对象的泛型类型信息。如果你在运行时有一个ArrayList<String>对象，你不能够判定这个对象是有泛型类型ArrayList<String>的 —— 并且通过不安全的原始类型，你可以将这个对象强制转换成ArrayList<Object>。

但是，反射允许你去检测方法和类的泛型类型。如果你实现了一个返回List的方法，并且你用反射获得了这个方法的返回类型，你会获得代表List<String>的ParameterizedType。

TypeToken类使用这种变通的方法以最小的语法开销去支持泛型类型的操作。

获取一个基本的、原始类的TypeToken非常简单：

1. TypeToken<String> str = TypeToken.of(String.**class**);
2. TypeToken<Integer> in = TypeToken.of(Integer.**class**);

获取一个含有泛型的类型TypeToken，当你知道在编译时的泛型参数类型，使用一个空的匿名内部类。

1. TypeToken<List<String>> typeToken = **new** TypeToken<List<String>>() {};
2. //java.util.List<java.lang.String>
3. System.out.println(typeToken.getType());
4. //interface java.util.List
5. System.out.println(typeToken.getRawType());

TypeToken提供了一种方法来动态的解决泛型类型参数：

举个栗子：

1. **static** <K,V> TypeToken<Map<K,V>> mapToken(TypeToken<K> keyToken,TypeToken<V> valueToken){
2. **return** **new** TypeToken<Map<K, V>>() {}
3. .where(**new** TypeParameter<K>() {}, keyToken)
4. .where(**new** TypeParameter<V>() {}, valueToken);
5. }
6. …
7. TypeToken<Map<String, Integer>> mapTypeToken = mapToken(TypeToken.of(String.**class**), TypeToken.of(Integer.**class**));
8. // java.util.Map<java.lang.String, java.lang.Integer>
9. System.out.println(mapTypeToken.getType());

TypeToken支持很多种类能支持的查询：

|  |  |
| --- | --- |
| 方法 | 描述 |
| getType() | 获得包装的java.lang.reflect.Type. |
| getRawType() | 返回大家熟知的运行时类 |
| getSubtype(Class<?>) | 返回那些有特定原始类的子类型。举个例子，如果这有一个Iterable并且参数是List.class，那么返回将是List。 |
| getSupertype(Class<?>) | 产生这个类型的超类，这个超类是指定的原始类型。举个例子，如果这是一个Set并且参数是Iterable.class，结果将会是Iterable。 |
| isAssignableFrom(type) | 如果这个类型是 assignable from 指定的类型，并且考虑泛型参数，返回true。List<? extends Number>是assignable from List，但List没有. |
| getTypes() | 返回一个Set，包含了这个所有接口，子类和类是这个类型的类。返回的Set同样提供了classes()和interfaces()方法允许你只浏览超类和接口类。 |
| isArray() | 检查某个类型是不是数组，甚至是<? extends A[]>。 |
| getComponentType() | 返回组件类型数组。 |

再举个栗子：

1. **public** **class** MyTypeTokenDemo<T> {
3. **private** TypeToken<T> recordType;
5. **public** MyTypeTokenDemo(){
6. **this**.recordType = **new** TypeToken<T>(getClass()) {};
7. }
9. **protected** Class<T> getRecordType(){
11. **return** (Class<T>) recordType.getRawType();
12. }

15. **public** **static** **void** main(String[] args) {
16. MyTypeTokenDemo<Integer> token = **new** MyTypeTokenDemo<Integer>() {};
17. //class java.lang.Integer
18. System.out.println(token.getRecordType());
20. }
21. }

注：如果是new MyTypeTokenDemo<Integer>()；的方式返回实例，getRecordType（）返回的会是java.lang.Object。具体原因可以查看：<https://google.github.io/guava/releases/snapshot/api/docs/com/google/common/reflect/TypeToken.html> 对TypeToken实例的三种方式。

resolveType是一个可以用来“替代”context token 中的类型参数的一个强大而复杂的查询操作。

1. TypeToken<Function<Integer, String>> funcToken = **new** TypeToken<Function<Integer, String>>() {};
2. TypeToken<?> typeToken1 = funcToken.resolveType(Function.**class**.getTypeParameters()[1]);
3. //java.lang.String
4. System.out.println(typeToken1);

### Invokable

Guava的Invokable是对java.lang.reflect.Method和java.lang.reflect.Constructor的流式包装。它简化了常见的反射代码的使用。

**举个栗子：**

1. Class<Object> objectClass = Object.**class**;
2. Method[] methods = objectClass.getMethods();
3. **for** (Method m : methods) {
4. String name = m.getName();
5. Invokable<?, Object> invokable = Invokable.from(m);
7. //JDK判断方法是不是public
8. System.out.println("JDK 判断方法是不是publie " + name + ": " + Modifier.isPublic(m.getModifiers()));
9. //Invokable
10. System.out.println("Invokable 判断方法是不是publie " + name + ": " + invokable.isPublic());
11. //JDK判断方法是不是package private
12. System.out.println("JDK 判断方法是不是package private " + name + ": " + ( !( Modifier.isPrivate(m.getModifiers()) || Modifier.isPublic(m.getModifiers()) ) ));
13. //Invokable
14. System.out.println("Invokable 判断方法是不是package private " + name + ": " + invokable.isPackagePrivate());
15. //JDK 判断 方法第一个参数是否被定义了注解@Nullable
16. **boolean** flag = **false**;
17. **int** length = m.getParameterAnnotations().length;
18. **if**(length <= 0) {
19. **continue**;
20. }
21. Annotation[] parameterAnnotation = m.getParameterAnnotations()[0];
22. **for** (Annotation annotation : parameterAnnotation) {
23. **if** (annotation **instanceof** Nullable){
24. flag = **true**;
25. }
26. }
27. System.out.println("JDK judge " + name + ": " + "第一个参数是否被定义了注解@Nullable" + flag);
28. //invokable
29. **boolean** annotationPresent = invokable.getParameters().get(0).isAnnotationPresent(Nullable.**class**);
30. System.out.println("Invokable judge " + name + ": " + "第一个参数是否被定义了注解@Nullable" + annotationPresent );
31. }

### Dynamic Proxies

实用方法Reflection.newProxy(Class, InvocationHandler)是一种更安全，更方便的API，它只有一个单一的接口类型需要被代理来创建Java动态代理时。

**举个栗子：**

先定义一个接口

1. **public** **interface** Animal {
3. /\*\*
4. \*
5. \*/
6. **void** run();
7. }

Dog实现类

1. **public** **class** Dog **implements** Animal {
2. @Override
3. **public** **void** run() {
4. System.out.println("dog run is fast");
5. }
6. }

实现一个InvocationHandler

1. **public** **class** AnimalInvocationHandler **implements** InvocationHandler {
3. **private** Animal animal;
4. **public** AnimalInvocationHandler(Animal animal){
5. **this**.animal = animal;
6. }
7. @Override
8. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
9. System.out.println("hhhh");
10. method.invoke(animal,args);
11. **return** proxy;
12. }
13. }

不同实现动态代理

1. //JDK 实现动态代理
2. Animal dog = (Animal) Proxy.newProxyInstance(Dog.**class**.getClassLoader(), **new** Class[]{Animal.**class**}, **new** AnimalInvocationHandler(**new** Dog()) );
3. dog.run();

6. // Guava 实现动态代理
7. Animal animal = Reflection.newProxy(Animal.**class**, **new** AnimalInvocationHandler(**new** Dog()));
8. animal.run();

**AbstractInvocationHandler**

有时候你可能想动态代理能够更直观的支持equals()，hashCode()和toString()，那就是：

* 一个代理实例equal另外一个代理实例，只要他们有同样的接口类型和equal的invocation handlers。
* 一个代理实例的toString()会被代理到invocation handler的toString()，这样更容易自定义。

AbstractInvocationHandler实现了以上逻辑。

除此之外，AbstractInvocationHandler确保传递给handleInvocation(Object, Method, Object[]))的参数数组永远不会空，从而减少了空指针异常的机会。

### 14.4 ClassPath

严格来讲，Java没有平台无关的方式来浏览类和类资源。不过一定的包或者工程下，还是能够实现的，比方说，去检查某个特定的工程的惯例或者某种一直遵从的约束。

ClassPath是一种实用工具，它提供尽最大努力的类路径扫描。

**举个栗子：**

1. Dog dog = **new** Dog();
2. ClassPath from = ClassPath.from(dog.getClass().getClassLoader());
3. ImmutableSet<ClassPath.ClassInfo> classInfos = from.getTopLevelClasses("xin.liujiajun.guava.reflection");
4. **for** (ClassPath.ClassInfo c: classInfos) {
5. System.out.println(c.getName());
6. //包名
7. c.getPackageName();
8. //简单类名
9. c.getSimpleName();
10. }
11. //output
12. //xin.liujiajun.guava.reflection.Animal
13. //xin.liujiajun.guava.reflection.AnimalInvocationHandler
14. //xin.liujiajun.guava.reflection.ClassPathDemo
15. //xin.liujiajun.guava.reflection.Dog
16. //xin.liujiajun.guava.reflection.DynamicProxiesDemo
17. //xin.liujiajun.guava.reflection.InvokableDemo
18. //xin.liujiajun.guava.reflection.TypeTokenDemo

在上面的例子中，ClassInfo是被加载类的句柄。它允许程序员去检查类的名字和包的名字，让类直到需要的时候才被加载。

值得注意的是，ClassPath是一个尽力而为的工具。它只扫描jar文件中或者某个文件目录下的class文件。也不能扫描非URLClassLoader的自定义class loader管理的class，所以不要将它用于关键任务生产任务。

### 14.5 Class Loading

工具方法Reflection.initialize(Class…)能够确保特定的类被初始化——执行任何静态初始化。

**举个栗子：**

**定义一个类**

1. **public** **class** Constant {
3. **public** **static** **final** String OK = "ok";
5. **static** {
6. System.out.println("static init");
7. }
9. }
10. **public** **static** **void** main(String[] args) {
11. //OK是编译期静态常量，不会触发类初始化，所以只会输出ok
12. System.out.println(Constant.OK);
13. //这里会输出 static init
14. Reflection.initialize(Constant.**class**);
15. }

使用这种方法的是一个代码异味，因为静态伤害系统的可维护性和可测试性。在有些情况下，你别无选择，而与传统的框架，操作间，这一方法有助于保持代码不那么丑。

**补充：**关于类加载的初始化阶段，在虚拟机规范严格规定了有且只有5种场景必须对类进行初始化：

1. 使用new关键字实例化对象时、读取或者设置一个类的静态字段(不包含编译期常量)以及调用静态方法的时候，必须触发类加载的初始化过程(类加载过程最终阶段)。
2. 使用反射包(java.lang.reflect)的方法对类进行反射调用时，如果类还没有被初始化，则需先进行初始化，这点对反射很重要。
3. 当初始化一个类的时候，如果其父类还没进行初始化则需先触发其父类的初始化。
4. 当Java虚拟机启动时，用户需要指定一个要执行的主类(包含main方法的类)，虚拟机会先初始化这个主类
5. 当使用JDK 1.7 的动态语言支持时，如果一个java.lang.invoke.MethodHandle 实例最后解析结果为REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄对应类没有初始化时，必须触发其初始化

# 附录

## 1. 布鲁姆过滤器

布鲁姆过滤器的维基页面：

<https://zh.wikipedia.org/wiki/%E5%B8%83%E9%9A%86%E8%BF%87%E6%BB%A4%E5%99%A8>

布鲁姆过滤器github教程：

<https://llimllib.github.io/bloomfilter-tutorial/zh_CN>

## 2. Github地址

git@github.com:liujiajun2/guava-pattern-demo.git

## 3. 图论地址

<https://zhuanlan.zhihu.com/p/25498681>

## 4. Lambda表达式

### 4.1 初识Lambda

可以把Lambda表达式理解为简洁地表示可传递的匿名函数的一种方式：它没有名称，但是有参数列表，函数主题，返回类型，可能还有一个可以抛出异常的列表。

匿名—不像普通的方法那样有一个明确的名称，写得少而想的多

函数—Lambda函数不像方法那样属于某个特定的类。但是和方法一样，Lambda有参数列表，函数主题，返回类型，可能还会有抛出的异常列表。

传递—Lambda表达式可以作为参数传递给方法或存储在变量中。

简洁—无需像匿名类那样写很多模板代码。