

Guava 学习笔记

Guava 是什么?

Guava 是一种基于开源的 Java 库，其中包含谷歌正在由他们很多项目使用的很多核心库。这个库是为了方便编码，并减少编码错误。这个库提供用于集合，缓存，支持原语，并发性，常见注解，字符串处理，I/O 和验证的实用方法。

Guava 的好处

1. 标准化 - Guava 库是由谷歌托管。
2. 高效，可靠，快速和有效的扩展 JAVA 标准库
3. 优化 -Guava 库经过高度的优化。
4. 函数式编程 -增加 JAVA 功能和处理能力。
5. 实用程序 - 提供了经常需要在应用程序开发的许多实用程序类。
6. 验证 -提供标准的故障安全验证机制。
7. 最佳实践 - 强调最佳的做法。

基本工具

使用和避免 Null

95%的集合类不接受 null 值作为元素。我们认为，相比默默地接受 null，使用快速失败操作拒绝 null 值对开发者更有帮助

Null 的含糊语义：Map.get(key)返回 Null 时，可能表示 map 中的值是 null，亦或 map 中没有 key 对应的值。Null 可以表示失败、成功或几乎任何情况

很多 Guava 工具类对 Null 值都采用快速失败操作，除非工具类本身提供了针对 Null 值的因变措施。此外，Guava 还提供了很多工具类，让你更方便地用特定值替换 Null 值。

不要在 Set 中使用 null，或者把 null 作为 map 的键值。使用特殊值代表 null 会让查找操作的语义更清晰。

大多数情况下，开发人员使用 null 表明的是某种缺失情形：可能是已经有一个默认值，或没有值，或找不到值。

Optional

Optional<T>表示可能为 null 的 T 类型引用。一个 Optional 实例可能包含非 null 的引用（我们称之为引用存在），也可能什么也不包括（称之为引用缺失）。它从不说包含的是 null 值，而是用存

Learn, work and try—Edwin Xu

在或缺失(**present and absent**)来表示。

`java.util.Optional` 和 `com.google.common.base.Optional` 类似，有几个区别：

1. Java `Optional` 不是序列化的，而 Google 是
2. Java 多了一些方法：`ifPresent`, `filter`, `flatMap`, `orElseThrow`
3. Java 有 `primitive-specialized` 版本的 `OptionalInt`、`OptionalLong`、`OptionalDouble`

Java `Optional` 支持函数式表达式：

1. `Consumer`
2. `Predicate`
3. `Function`

```
public class JavaOptionalTest001 {
    public static <T> Optional<T> getOptional(T t){
        return Optional.of(t);
    }
    public static void main(String[] args) {
        Optional<Student> optional = JavaOptionalTest001.getOptional(new Student());
        optional.ifPresent(student -> System.out.println(student.toString()));

        Optional<Student> student1 = optional.filter(student -> student.getAge() > 30);
        System.out.println(student1);
    }
}
```

意义：

使用 `Optional` 除了赋予 `null` 语义，增加了可读性，最大的优点在于它是一种傻瓜式的防护。`Optional` 迫使你积极思考引用缺失的情况，因为你必须显式地从 `Optional` 获取引用。直接使用 `null` 很容易让人忘掉某些情形，尽管 `FindBugs` 可以帮助查找 `null` 相关的问题，但是我们还是认为它并不能准确地定位问题根源。

其他处理 `null` 的便利方法

当你需要用一个默认值来替换可能的 `null`，请使用 `Objects.firstNonNull(T, T)` 方法。如果两个值都是 `null`，该方法会抛出 `NullPointerException`。`Optional` 也是一个比较好的替代方案，例如：`Optional.of(first).or(second)`。

还有其它一些方法专门处理 `null` 或空字符串：`emptyOrNull(String)`, `nullToEmpty(String)`, `isNullOrEmpty(String)`

前置条件

前置条件：让方法调用的前置条件判断更简单。

`Preconditions` 类：每个方法都有三个变种：

1. 没有额外参数：抛出的异常中没有错误消息；
2. 有一个 `Object` 对象作为额外参数：抛出的异常使用 `Object.toString()` 作为错误消息；

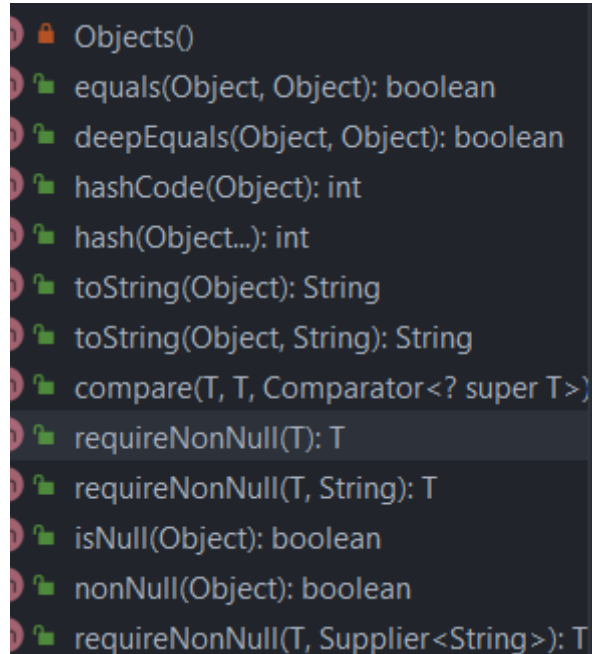
3. 有一个 `String` 对象作为额外参数，并且有一组任意数量的附加 `Object` 对象：这个变种处理异常消息的方式有点类似 `printf`，但考虑 GWT 的兼容性和效率，只支持 `%s` 指示符。

```
checkArgument(i >= 0, "Argument was %s but expected nonnegative", i);  
checkArgument(i < j, "Expected i < j, but %s > %s", i, j);
```

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

常见 Object 方法

Java 和 Google 都有 `Objects` 类：
Java `Objects` 更加完善



常用的: `Objects.requireNonNull`

compare/compareTo:

Guava 提供了 `ComparisonChain`。`ComparisonChain` 执行一种懒比较: 它执行比较操作直至发现非零的结果, 在那之后的比较输入将被忽略。

```
public int compareTo(Foo that) {
    return ComparisonChain.start()
        .compare(this.aString, that.aString)
        .compare(this.anInt, that.anInt)
        .compare(this.anEnum, that.anEnum, Ordering.natural().nullsLast())
        .result();
}
```

排序: Guava 强大的“流畅风格比较器”

排序器 [`Ordering`] 是 Guava 流畅风格比较器 [`Comparator`] 的实现, 它可以用来为构建复杂的比较器, 以完成集合排序的功能。

从实现上说, `Ordering` 实例就是一个特殊的 `Comparator` 实例。`Ordering` 把很多基于 `Comparator` 的静态方法 (如 `Collections.max`) 包装为自己的实例方法 (非静态方法), 并且提供了链式调用方法, 来定制和增强现有的比较器。

创建排序器: 常见的排序器可以由下面的静态方法创建

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

实现自定义的排序器时，除了用上面的 `from` 方法，也可以跳过实现 `Comparator`，而直接继承 `Ordering`

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它排序器

<code>reverse()</code>	获取语义相反的排序器
<code>nullsFirst()</code>	使用当前排序器，但额外把null值排到最前面。
<code>nullsLast()</code>	使用当前排序器，但额外把null值排到最后面。
<code>compound(Comparator)</code>	合成另一个比较器，以处理当前排序器中的相等情况。
<code>lexicographical()</code>	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。
<code>onResultOf(Function)</code>	对集合中元素调用Function，再按返回值用当前排序器排序。

```
class Foo {
    @Nullable String sortedBy;
    int notSortedBy;
}

Ordering<Foo> ordering = Ordering.natural().nullsFirst().onResultOf(new Function<Foo, String>() {
    public String apply(Foo foo) {
        return foo.sortedBy;
    }
});
```

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。

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	另请参见
<code>greatestOf(Iterable iterable, int k)</code>	获取可迭代对象中最大的k个元素。	<code>leastOf</code>
<code>isOrdered(Iterable)</code>	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。	<code>isStrictlyOrdered</code>
<code>sortedCopy(Iterable)</code>	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。	<code>immutableSortedCopy</code>
<code>min(E, E)</code>	返回两个参数中最小的那个。如果相等，则返回第一个参数。	<code>max(E, E)</code>
<code>min(E, E, E, E...)</code>	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。	<code>max(E, E, E, E...)</code>
<code>min(Iterable)</code>	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 <code>NoSuchElementException</code> 。	<code>max(Iterable)</code> , <code>min(Iterator)</code> , <code>max(Iterator)</code>

Throwables：简化异常和错误的传播与检查

异常传播

有时候，你会想把捕获到的异常再次抛出。这种情况通常发生在 `Error` 或 `RuntimeException` 被捕获的时候，你没想捕获它们，但是声明捕获 `Throwable` 和 `Exception` 的时候，也包括了 `Error` 或 `RuntimeException`。Guava 提供了若干方法，来判断异常类型并且重新传播异常。

Throwables

```
try {
    someMethodThatCouldThrowAnything();
} catch (IKnowWhatToDoWithThisException e) {
    handle(e);
} catch (Throwable t) {
    Throwables.propagateIfInstanceOf(t, IOException.class);
    Throwables.propagateIfInstanceOf(t, SQLException.class);
    throw Throwables.propagate(t);
}
```

所有这些方法都会自己决定是否要抛出异常，但也能直接抛出方法返回的结果——例如，`throw Throwables.propagate(t);`—— 这样可以向编译器声明这里一定会抛出异常。

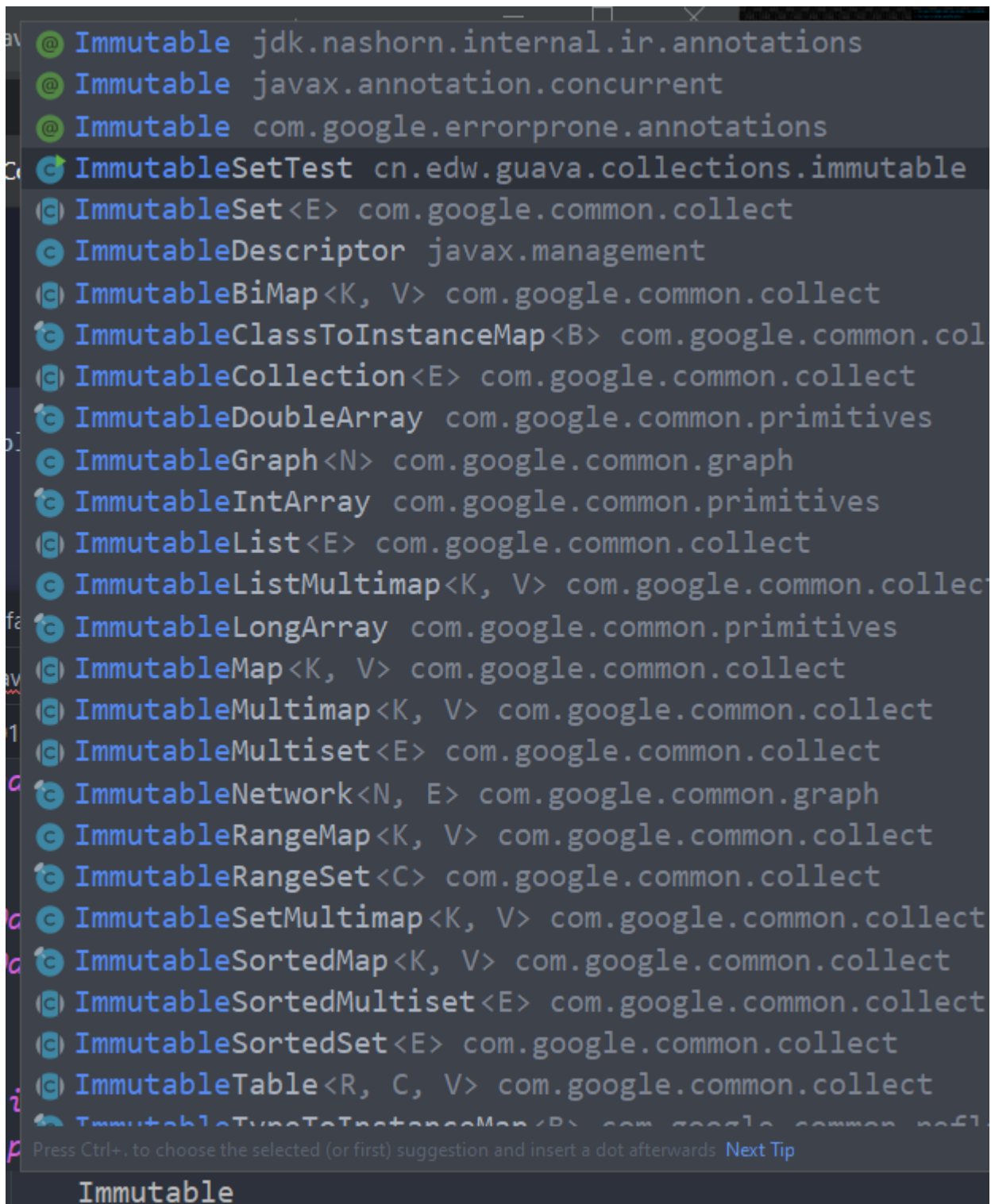
<code>RuntimeException propagate(Throwable)</code>	如果Throwable是Error或RuntimeException，直接抛出；否则把Throwable包装成RuntimeException抛出。返回类型是RuntimeException，所以你可以像上面说的那样写成 <code>throw Throwables.propagate(t)</code> ，Java编译器会意识到这行代码保证抛出异常。
<code><u>void</u> <u>propagateIfInstanceOf</u>(<u>Throwable</u>, <u>Class<X</u> <u>extends Exception></u>), <u>throws X</u></code>	Throwable类型为X才抛出
<code>void propagateIfPossible(Throwable)</code>	Throwable类型为Error或RuntimeException才抛出
<code>void propagateIfPossible(Throwable, Class<X extends Throwable>) throws X</code>	Throwable类型为X, Error或RuntimeException才抛出

Throwables.propagate 的用法：模仿 Java7 的多重异常捕获和再抛出

.....

集合

不可变集合



不可变集合的好处：

当对象被不可信的库调用时，不可变形式是安全的；

不可变对象被多个线程调用时，不存在竞态条件问题

不可变集合不需要考虑变化，因此可以节省时间和空间。所有不可变的集合都比它们的可变形式有更好的内存利用率（分析和测试细节）；

不可变对象因为有固定不变，可以作为常量来安全使用。

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：

1. 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；
2. 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；
3. 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。

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。如果你需要在不可变集合中使用 `null`，请使用 JDK 中的 `Collections.unmodifiableXXX` 方法

不可变集合可以用如下多种方式创建：

1. `copyOf` 方法，如 `ImmutableSet.copyOf(set)`；
2. `of` 方法，如 `ImmutableSet.of("a", "b", "c")` 或 `ImmutableMap.of("a", 1, "b", 2)`；
3. Builder 工具，如

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

```
ImmutableSortedSet.of("a", "b", "c", "a", "d", "b");
```

`ImmutableXXX.copyOf` 方法会尝试在安全的时候避免做拷贝——实际的实现细节不详，但通常来说是很智能的

比如：

```
ImmutableSet<String> foobar = ImmutableSet.of("foo", "bar", "baz");
thingamajig(foobar);

void thingamajig(Collection<String> collection) {
    ImmutableList<String> defensiveCopy = ImmutableList.copyOf(collection);
    ...
}
```

在这段代码中，`ImmutableList.copyOf(foobar)` 会智能地直接返回 `foobar.asList()`，它是一个 `ImmutableSet` 的常量时间复杂度的 **List 视图**。作为一种探索，`ImmutableXXX.copyOf(ImmutableCollection)` 会试图对如下情况避免线性时间拷贝：

1. 在常量时间内使用底层数据结构是可能的——例如，`ImmutableSet.copyOf(ImmutableList)` 就不能在常量时间内完成。
2. 不会造成内存泄露——例如，你有个很大的不可变集合 `ImmutableList<String> hugeList`，`ImmutableList.copyOf(hugeList.subList(0, 10))` 就会显式地拷贝，以免不必要地持有 `hugeList` 的引用。
3. 不改变语义——所以 `ImmutableSet.copyOf(myImmutableSortedSet)` 会显式地拷贝，因为和基于比较器的 `ImmutableSortedSet` 相比，`ImmutableSet` 对 `hashCode()` 和 `equals` 有不同语

义。

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。例如，你可以使用 `sortedSet.asList().get(k)` 从 `ImmutableSortedSet` 中读取第 `k` 个最小元素。

细节：关联可变集合和不可变集合		
可变集合接口	属于**JDK还是Guava**	不可变版本
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

统计一个词在文档中出现了多少次，传统的做法是这样的：

```
Map<String, Integer> counts = new HashMap<String, Integer>();
for (String word : words) {
    Integer count = counts.get(word);
    if (count == null) {
```

Learn, work and try—Edwin Xu

```
        counts.put(word, 1);
    } else {
        counts.put(word, count + 1);
    }
}
```

//这种写法很笨拙，也容易出错，并且不支持同时收集多种统计信息，如总词数。我们可以做的更好。

Guava 提供了一个新集合类型 **Multiset**，它可以多次添加相等的元素

Multiset 继承自 JDK 中的 **Collection** 接口，而不是 **Set** 接口，所以包含重复元素并没有违反原有的接口契约

可以用两种方式看待 **Multiset**:

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

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

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

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

1. **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 $O(\log n)$ 。
2. **entrySet()** 返回 **Set<Multiset.Entry<E>>**，和 **Map** 的 **entrySet** 类似。
3. **elementSet()** 返回所有不重复元素的 **Set<E>**，和 **Map** 的 **keySet()** 类似。
4. 所有 **Multiset** 实现的内存消耗随着不重复元素的个数线性增长。

TreeMultiset 在判断元素是否相等时，与 **TreeSet** 一样用 **compare**，而不是 **Object.equals**。另外特别注意，**Multiset.addAll(Collection)** 可以添加 **Collection** 中的所有元素并进行计数，这比用 **for** 循环往 **Map** 添加元素和计数方便多了。

<code>count(E)</code>	给定元素在Multiset中的计数
<code>elementSet()</code>	Multiset中不重复元素的集合，类型为Set<E>
<code>entrySet()</code>	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
<code>add(E, int)</code>	增加给定元素在Multiset中的计数
<code>remove(E, int)</code>	减少给定元素在Multiset中的计数
<code>setCount(E, int)</code>	设置给定元素在Multiset中的计数，不可以为负数
<code>size()</code>	返回集合元素的总个数（包括重复的元素）

Multiset 不是 Map

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

1. 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。
2. 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）
3. multiset.iterator()会迭代重复元素，因此迭代长度等于multiset.size()。
4. 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。
5. 对multiset中没有的元素，multiset.count(elem)始终返回0。

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

各种实现

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

TreeMultiset 实现 SortedMultiset 接口

Multimap

每个有经验的 Java 程序员都在某处实现过 `Map<K, List<V>>` 或 `Map<K, Set<V>>`，并且要忍受这个结构的笨拙。例如，`Map<K, Set<V>>` 通常用来表示非标定有向图。Guava 的 `Multimap` 可以很容易地把一个键映射到多个值。换句话说，`Multimap` 是把键映射到任意多个值的一般方式。

可以用两种方式思考 `Multimap` 的概念：“键-单个值映射”的集合：

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

或者“键-值集合映射”的映射：

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

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

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

`Multimap` 还支持若干强大的视图

`Multimap<K, V>` 不是 `Map<K, Collection<V>>`

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

BiMap

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

Learn, work and try—Edwin Xu

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

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

BiMap<K, V>是特殊的 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**

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, 当你想使用多个键做索引的时候, 你可能会用类似 **Map<FirstName, Map<LastName, Person>>**的实现, 这种方式很丑陋, 使用上也不友好。Guava 为此提供了新集合类型 **Table**, 它有两个支持所有类型的键:”行”和”列”。**Table** 提供多种视图

Table 有如下几种实现:

1. **HashBasedTable**: 本质上用 **HashMap<R, HashMap<C, V>>**实现;
2. **TreeBasedTable**: 本质上用 **TreeMap<R, TreeMap<C,V>>**实现;
3. **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。
4. **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, 请参见 Javadoc 了解详情。

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。

RangeSet

Learn, work and try—Edwin Xu

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。

集合工具类

java.util.Collections 包含很多工具类

集合接口	属于**JDK还是Guava**	对应的**Guava**工具类
Collection	JDK	<div>Collections2</div> : 不要和java.util.Collections混淆
List	JDK	<div>Lists</div>
Set	JDK	<div>Sets</div>
SortedSet	JDK	<div>Sets</div>
Map	JDK	<div>Maps</div>
SortedMap	JDK	<div>Maps</div>
Queue	JDK	<div>Queues</div>
Multiset	Guava	<div>Multisets</div>
Multimap	Guava	<div>Multimaps</div>
BiMap	Guava	<div>Maps</div>
Table	Guava	<div>Tables</div>

在 JDK 7 之前，构造新的范型集合时要讨厌地重复声明范型：

```
List<TypeThatIsTooLongForItsOwnGood> list = new ArrayList<TypeThatIsTooLongForItsOwnGood>();
```

因此 Guava 提供了能够推断范型的静态工厂方法：

```
List<TypeThatIsTooLongForItsOwnGood> list = Lists.newArrayList();
Map<KeyType, LongishValueType> map = Maps.newLinkedHashMap();
```

JDK8 也不需要重复声明了

但 Guava 的静态工厂方法远不止这么简单。用工厂方法模式，我们可以方便地在初始化时就指定起始元素。

```
Set<Type> copySet = Sets.newHashSet(elements);
List<String> theseElements = Lists.newArrayList("alpha", "beta", "gamma");
```


Iterables

在可能的情况下，Guava 提供的工具方法更偏向于接受 `Iterable` 而不是 `Collection` 类型。在 Google，对于不存放在主存的集合——比如从数据库或其他数据中心收集的结果集，因为实际上还没有攫取全部数据，这类结果集都不能支持类似 `size()` 的操作——通常都不会用 `Collection` 类型来表示。

因此，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`。

<code>concat(Iterable<T>...)</code>	串联多个iterables的懒视图*	<code>concat(Iterable...)</code>
<code>frequency(Iterable, Object)</code>	返回对象在iterable中出现的次数	与 <code>Collections.frequency(Collection, Object)</code> 比较； <code>Multiset</code>
<code>partition(Iterable, int)</code>	把iterable按指定大小分割，得到的子集都不能进行修改操作	<code>Lists.partition(List, int)</code> ； <code>paddedPartition(Iterable, int)</code>
<code>getFirst(Iterable, T default)</code>	返回iterable的第一个元素，若iterable为空则返回默认值	与 <code>Iterable.iterator().next()</code> 比较； <code>FluentIterable.first()</code>
<code>getLast(Iterable)</code>	返回iterable的最后一个元素，若iterable为空则抛出 <code>NoSuchElementException</code>	<code>getLast(Iterable, T default)</code> ；

集合扩展工具类

缓存

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
    .maximumSize(1000)
    .expireAfterWrite(10, TimeUnit.MINUTES)
    .removeListener(MY_LISTENER)
    .build(
        new CacheLoader<Key, Graph>() {
            public Graph load(Key key) throws AnyException {
                return createExpensiveGraph(key);
            }
        }
    );
```

函数式编程

JDK8 已经支持了

函数式接口

自带的有：

1. Predicate
2. Function
3. Consumer
4. Supplier

并发

字符串处理

连接器[Joiner]

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。Fluent 风格的 `Joiner` 让连接字符串更简单。

```
Joiner joiner = Joiner.on("; ").skipNulls();  
return joiner.join("Harry", null, "Ron", "Hermione");
```

拆分器[Splitter]

JDK 内建的字符串拆分工具有一些古怪的特性。比如，`String.split` 悄悄丢弃了尾部的分隔符。问题：`"a,,b,".split(",")` 返回？

1. `"", "a", "", "b", ""`
2. `null, "a", null, "b", null`
3. `"a", null, "b"`
4. `"a", "b"`

以上都不对
正确答案是 5:””，“a”，“”，“b”。只有尾部的空字符串被忽略了

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。

```
Splitter.on(',')
    .trimResults()
    .omitEmptyStrings()
    .split("foo,bar,, qux");
```

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)

方法	描述	拆分器修饰符
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	

字符匹配器 [CharMatcher]

直观上，你可以认为一个 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 确实也实现了 Predicate<Character>——但类似”所有空白字符”或”所有小写字母”的需求太普遍了，Guava Learn, work and try—Edwin Xu

因此创建了这一 API。

```
String noControl = CharMatcher.JAVA_ISO_CONTROL.removeFrom(string); //移除 control 字符
String theDigits = CharMatcher.DIGIT.retainFrom(string); //只保留数字字符
String spaced = CharMatcher.WHITESPACE.trimAndCollapseFrom(string, ' ');
//去除两端的空格，并把中间的连续空格替换成单个空格
String noDigits = CharMatcher.JAVA_DIGIT.replaceFrom(string, "*"); //用*号替换所有数字
String lowerAndDigit = CharMatcher.JAVA_DIGIT.or(CharMatcher.JAVA_LOWER_CASE).retainFrom(string);
// 只保留数字和小写字母
```

CharMatcher中的常量可以满足大多数字符匹配需求：

ANY	NONE	WHITESPACE	BREAKING_WHITESPACE
INVISIBLE	DIGIT	JAVA_LETTER	JAVA_DIGIT
JAVA_LETTER_OR_DIGIT	JAVA_ISO_CONTROL	JAVA_LOWER_CASE	JAVA_UPPER_CASE
ASCII	SINGLE_WIDTH		

Charsets

大小写格式[CaseFormat]

LOWER_CAMEL	lowerCamel
LOWER_HYPHEN	lower-hyphen
LOWER_UNDERSCORE	lower_underscore
UPPER_CAMEL	UpperCamel
UPPER_UNDERSCORE	UPPER_UNDERSCORE

```
CaseFormat.UPPER_UNDERSCORE.to(CaseFormat.LOWER_CAMEL, "CONSTANT_NAME")); // returns "constantName"
```

原生类型

IO

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）

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，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。

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

关于 `InputSupplier` 和 `OutputSupplier` 要注意：

在 `ByteStreams`、`CharStreams` 以及 `com.google.common.io` 包中的一些其他类中，某些方法仍然在使用 `InputSupplier` 和 `OutputSupplier` 接口。这两个借口和相关的方法是不推荐使用的：它们已经被下面描述的 `source` 和 `sink` 类型取代了，并且最终会被移除。

源与汇

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。例如，Guava 有 `Files.toByteArray(File)` 和 `Files.write(File, byte[])`。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

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。

Guava提供了若干源与汇的实现：

字节	字符
Files.asByteSource(File)	Files.asCharSource(File, Charset)
Files.asByteSink(File, FileMode...)	Files.asCharSink(File, Charset, FileMode...)
Resources.asByteSource(URL)	Resources.asCharSource(URL, Charset)
ByteSource.wrap(byte[])	CharSource.wrap(CharSequence)
<u>ByteSource.concat(ByteSource...)</u>	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。

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：

1. openStream(): 根据源与汇的类型, 返回 InputStream、OutputStream、Reader 或者 Writer。
2. 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
<code>byte[] read()</code>	<code>String read()</code>
N/A	<code>ImmutableList<String> readLines()</code>
N/A	<code>String readFirstLine()</code>
<code>long copyTo(ByteSink)</code>	<code>long copyTo(CharSink)</code>
<code>long copyTo(OutputStream)</code>	<code>long copyTo(Appendable)</code>
<code>long size()</code> (in bytes)	N/A
<code>boolean isEmpty()</code>	<code>boolean isEmpty()</code>
<code>boolean contentEquals(ByteSource)</code>	N/A
<code>HashCode hash(HashFunction)</code>	N/A

汇操作

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

```
//Read the lines of a UTF-8 text file
ImmutableList<String> lines = Files.asCharSource(file, Charsets.UTF_8).readLines();
//Count distinct word occurrences in a file
Multiset<String> wordOccurrences = HashMultiset.create(
    Splitter.on(CharMatcher.WHITESPACE)
```

Learn, work and try—Edwin Xu


```
        .trimResults()
        .omitEmptyStrings()
        .split(Files.asCharSource(file, Charsets.UTF_8).read()));

//SHA-1 a file
HashCode hash = Files.asByteSource(file).hash(Hashing.sha1());

//Copy the data from a URL to a file
Resources.asByteSource(url).copyTo(Files.asByteSink(file));
```

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。

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

散列

事件总线

数学运算

反射

Learn, work and try—Edwin Xu

