GoogleGuava 官方教程

极客学院出版

前言

Guava 工程包含了若干被 Google 的 Java 项目广泛依赖 的核心库，我们希望通过此文档为 Guava 中最流行和最强大的功能，提供更具可读性和解释性的说明。

适用人群

本教程是中级教程，适合 Guava 中级开发者的进阶学习。

学习前提

学习本教程前，建议您先学习 java 语言开发。

> 致谢 原文链接 译文链接
> 译者: 沈义扬，罗立树，何一昕，武祖
> 校对：方腾飞

转载自并发编程网 – ifeve.com

目录

1

基本工具

使用和避免 null

Doug Lea 说，“Null 真糟糕。”

当 Sir C. A. R. Hoare 使用了 null 引用后说，”使用它导致了十亿美金的错误。”

轻率地使用 null 可能会导致很多令人惊愕的问题。通过学习 Google 底层代码库，我们发现 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。

此外，Null 的含糊语义让人很不舒服。Null 很少可以明确地表示某种语义，例如，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。使用 Null 以外的特定值，会让你的逻辑描述变得更清晰。

Null 确实也有合适和正确的使用场景，如在性能和速度方面 Null 是廉价的，而且在对象数组中，出现 Null 也是无法避免的。但相对于底层库来说，在应用级别的代码中，Null 往往是导致混乱，疑难问题和模糊语义的元凶，就如同我们举过的 Map.get(key)的例子。最关键的是，Null 本身没有定义它表达的意思。

鉴于这些原因，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。

具体案例

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。

如果你想把 null 作为 map 中某条目的值，更好的办法是 不把这一条目放到 map 中，而是单独维护一个”值为 null 的键集合”（null keys）。Map 中对应某个键的值是 null，和 map 中没有对应某个键的值，是非常容易混淆的两种情况。因此，最好把值为 null 的键分离开，并且仔细想想，null 值的键在你的项目中到底表达了什么语义。

如果你需要在列表中使用 null——并且这个列表的数据是稀疏的，使用 Map<Integer, E>可能会更高效，并且更准确地符合你的潜在需求。

此外，考虑一下使用自然的 null 对象——特殊值。举例来说，为某个 enum 类型增加特殊的枚举值表示 null，比如 java.math.RoundingMode 就定义了一个枚举值 UNNECESSARY，它表示一种不做任何舍入操作的模式，用这种模式做舍入操作会直接抛出异常。

如果你真的需要使用 null 值，但是 null 值不能和 Guava 中的集合实现一起工作，你只能选择其他实现。比如，用 JDK 中的 Collections.unmodifiableList 替代 Guava 的 ImmutableList

Optional

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。例如，Map.get 返回 null 就表示找不到给定键对应的值。

Guava 用 Optional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，而是用存在或缺失来表示。但 Optional 从不会包含 null 值引用。

```
Optional<Integer> possible = Optional.of(5);

possible.isPresent(); // returns true

possible.get(); // returns 5
```

Optional 无意直接模拟其他编程环境中的”可选” or “可能”语义，但它们的确有相似之处。

Optional 最常用的一些操作被罗列如下：

创建 Optional 实例（以下都是静态方法）：

`Optional.of(T)`	创建指定引用的 Optional 实例，若引用为 null 则快速失败
`Optional.absent()`	创建引用缺失的 Optional 实例
`Optional.fromNullable(T)`	创建指定引用的 Optional 实例，若引用为 null 则表示缺失

用 Optional 实例查询引用（以下都是非静态方法）：

`boolean isPresent()`	如果 Optional 包含非 null 的引用（引用存在），返回true
`T get()`	返回 Optional 所包含的引用，若引用缺失，则抛出 java.lang.IllegalStateException
`T or(T)`	返回 Optional 所包含的引用，若引用缺失，返回指定的值
`T orNull()`	返回 Optional 所包含的引用，若引用缺失，返回 null
`Set<T> asSet()`	返回 Optional 所包含引用的单例不可变集，如果引用存在，返回一个只有单一元素的集合，如果引用缺失，返回一个空集合。

使用 Optional 的意义在哪儿?

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。

如同输入参数，方法的返回值也可能是 null。和其他人一样，你绝对很可能会忘记别人写的方法 method(a,b)会返回一个 null，就好像当你实现 method(a,b)时，也很可能忘记输入参数 a 可以为 null。将方法的返回类型指定为 Optional，也可以迫使调用者思考返回的引用缺失的情形。

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：emptyToNull(String)，nullToEmpty(String)，isNullOrEmpty(String)。我们想要强调的是，这些方法主要用来与混淆 null/空的 API 进行交互。当每次你写下混淆 null/空的代码时，Guava 团队都泪流满面。（好的做法是积极地把 null 和空区分开，以表示不同的含义，在代码中把 null 和空同等对待是一种令人不安的坏味道。

前置条件

前置条件：让方法调用的前置条件判断更简单。

Guava 在 Preconditions 类中提供了若干前置条件判断的实用方法，我们强烈建议在 Eclipse 中静态导入这些方法。每个方法都有三个变种：

- 没有额外参数：抛出的异常中没有错误消息；
- 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；
- 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。例如：

```
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
`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

译者注：

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)

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)

相比 Apache Commons 提供的类似方法，我们把 Guava 中的 Preconditions 作为首选。Piotr Jagielski 在他的博客中简要地列举了一些理由：

- 在静态导入后，Guava 方法非常清楚明晰。checkNotNull 清楚地描述做了什么，会抛出什么异常；
- checkNotNull 直接返回检查的参数，让你可以在构造函数中保持字段的单行赋值风格：this.field = checkNotNull(field)
- 简单的、参数可变的 printf 风格异常信息。鉴于这个优点，在 JDK7 已经引入 Objects.requireNonNull 的情况下，我们仍然建议你使用 checkNotNull。

在编码时，如果某个值有多重的前置条件，我们建议你把它们放到不同的行，这样有助于在调试时定位。此外，把每个前置条件放到不同的行，也可以帮助你编写清晰和有用的错误消息。

常见 Object 方法

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 帮助你执行 null 敏感的 equals 判断，从而避免抛出 NullPointerException。例如:

```
Objects.equal("a", "a"); // returns true
Objects.equal(null, "a"); // returns false
Objects.equal("a", null); // returns false
Objects.equal(null, null); // returns true
```

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

hashCode

用对象的所有字段作散列[hash]运算应当更简单。Guava 的 Objects.hashCode(Object...)会对传入的字段序列计算出合理的、顺序敏感的散列值。你可以使用 Objects.hashCode(field1, field2, …, fieldn)来代替手动计算散列值。

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

toString

好的 toString 方法在调试时是无价之宝，但是编写 toString 方法有时候却很痛苦。使用 Objects.toStringHelper 可以轻松编写有用的 toString 方法。例如：

```
// Returns "ClassName{x=1}"
Objects.toStringHelper(this).add("x", 1).toString();
// Returns "MyObject{x=1}"
Objects.toStringHelper("MyObject").add("x", 1).toString();
```

compare/compareTo

实现一个比较器[Comparator]，或者直接实现 Comparable 接口有时也伤不起。考虑一下这种情况：

```
class Person implements Comparable<Person> {
  private String lastName;
  private String firstName;
  private int zipCode;

  public int compareTo(Person other) {
    int cmp = lastName.compareTo(other.lastName);
    if (cmp != 0) {
      return cmp;
    }
    cmp = firstName.compareTo(other.firstName);
    if (cmp != 0) {
      return cmp;
    }
    return Integer.compare(zipCode, other.zipCode);
  }
}
```

这部分代码太琐碎了，因此很容易搞乱，也很难调试。我们应该能把这种代码变得更优雅，为此，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();
}
```

这种 Fluent 接口风格的可读性更高，发生错误编码的几率更小，并且能避免做不必要的工作。更多 Guava 排序器工具可以在下一节里找到。

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

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

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：

```
Ordering<String> byLengthOrdering = new Ordering<String>() {
  public int compare(String left, String right) {
    return Ints.compare(left.length(), right.length());
  }
};
```

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

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。

例如，你需要下面这个类的排序器。

```
class Foo {
```

```
    @Nullable String sortedBy;
    int notSortedBy;
}
```

考虑到排序器应该能处理 sortedBy 为 null 的情况，我们可以使用下面的链式调用来合成排序器：

```
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。

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。

超过一定长度的链式调用，也可能会带来阅读和理解上的难度。我们建议按下面的代码这样，在一个链中最多使用三个方法。此外，你也可以把 Function 分离成中间对象，让链式调用更简洁紧凑。

```
Ordering<Foo> ordering = Ordering.natural().nullsFirst().onResultOf(sortKeyFunction)
```

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
`greatestOf(Iterable iterable, int k)`	获取可迭代对象中最大的k个元素。	`leastOf`
`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。	`isStrictlyOrdered`
`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。	`immutableSortedCopy`
`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。	`max(E, E)`
`min(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。	`max(E, E, E...)`
`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。	`max(Iterable)`, `min(Iterator)`, `max(Iterator)`

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。例如：

```
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。

Guava 中的异常传播方法简要列举如下：

RuntimeException propagate(Throwable)	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`throw Throwables.propagate(t)`，Java 编译器会意识到这行代码保证抛出异常。
void propagateIfInstanceOf(Throwable, Class<X extends Exception>) throws X	Throwable 类型为 X 才抛出
void propagateIfPossible(Throwable)	Throwable 类型为 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

Throwables.propagate 的用法

模仿 Java7 的多重异常捕获和再抛出

通常来说，如果调用者想让异常传播到栈顶，他不需要写任何 catch 代码块。因为他不打算从异常中恢复，他可能就不应该记录异常，或者有其他的动作。他可能是想做一些清理工作，但通常来说，无论操作是否成功，清理工作都要进行，所以清理工作可能会放在 finallly 代码块中。但有时候，捕获异常然后再抛出也是有用的：也许调用者想要在异常传播之前统计失败的次数，或者有条件地传播异常。

当只对一种异常进行捕获和再抛出时，代码可能还是简单明了的。但当多种异常需要处理时，却可能变得一团糟：

```
@Override public void run() {
  try {
    delegate.run();
  } catch (RuntimeException e) {
    failures.increment();
    throw e;
  }catch (Error e) {
    failures.increment();
    throw e;
  }
}
```

Java7 用多重捕获解决了这个问题：

```
} catch (RuntimeException | Error e) {
  failures.increment();
  throw e;
}
```

非 Java7 用户却受困于这个问题。他们想要写如下代码来统计所有异常，但是编译器不允许他们抛出 Throwable（*译者注：这种写法把原本是 Erro r或 RuntimeException 类型的异常修改成了 Throwable，因此调用者不得不修改方法签名*）：

```
} catch (Throwable t) {
  failures.increment();
```

```
    throw t;
}
```

解决办法是用 throw Throwables.propagate(t)替换 throw t。在限定情况下（捕获 Error 和 RuntimeException），Throwables.propagate 和原始代码有相同行为。然而，用 Throwables.propagate 也很容易写出有其他隐藏行为的代码。尤其要注意的是，这个方案只适用于处理 RuntimeException 或 Error。如果 catch 块捕获了受检异常，你需要调用 propagateIfInstanceOf 来保留原始代码的行为，因为 Throwables.propagate 不能直接传播受检异常。

总之，Throwables.propagate 的这种用法也就马马虎虎，在 Java7 中就没必要这样做了。在其他 Java 版本中，它可以减少少量的代码重复，但简单地提取方法进行重构也能做到这一点。此外，使用 propagate 会意外地包装受检异常。

非必要用法：把抛出的 Throwable 转为 Exception

有少数 API，尤其是 Java 反射 API 和（以此为基础的）Junit，把方法声明成抛出 Throwable。和这样的 API 交互太痛苦了，因为即使是最通用的 API 通常也只是声明抛出 Exception。当确定代码会抛出 Throwable，而不是 Exception 或 Error 时，调用者可能会用 Throwables.propagate 转化 Throwable。这里有个用 Callable 执行 Junit 测试的范例：

```
public Void call() throws Exception {
    try {
        FooTest.super.runTest();
    } catch (Throwable t) {
        Throwables.propagateIfPossible(t, Exception.class);
        Throwables.propagate(t);
    }

    return null;
}
```

在这儿没必要调用 propagate()方法，因为 propagateIfPossible 传播了 Throwable 之外的所有异常类型，第二行的 propagate 就变得完全等价于 throw new RuntimeException(t)。（题外话：这个例子也提醒我们，propagateIfPossible 可能也会引起混乱，因为它不但会传播参数中给定的异常类型，还抛出 Error 和 RuntimeException）

这种模式（或类似于 throw new RuntimeException(t)的模式）在 Google 代码库中出现了超过 30 次。（搜索’propagateIfPossible[^;]* Exception.class[]);’）绝大多数情况下都明确用了”throw new RuntimeException(t)”。我们也曾想过有个”throwWrappingWeirdThrowable”方法处理 Throwable 到 Exception 的

转化。但考虑到我们用两行代码实现了这个模式，除非我们也丢弃 propagateIfPossible 方法，不然定义这个 throwWrappingWeirdThrowable 方法也并没有太大必要。

Throwables.propagate 的有争议用法

争议一：把受检异常转化为非受检异常

原则上，非受检异常代表 bug，而受检异常表示不可控的问题。但在实际运用中，即使 JDK 也有所误用——如 Object.clone()、Integer. parseInt(String)、URI(String)——或者至少对某些方法来说，没有让每个人都信服的答案，如 URI.create(String)的异常声明。

因此，调用者有时不得不把受检异常和非受检异常做相互转化：

```
try {
    return Integer.parseInt(userInput);
} catch (NumberFormatException e) {
    throw new InvalidInputException(e);
}
```

```
try {
    return publicInterfaceMethod.invoke();
} catch (IllegalAccessException e) {
    throw new AssertionError(e);
}
```

有时候，调用者会使用 Throwables.propagate 转化异常。这样做有没有什么缺点？最主要的恐怕是代码的含义不太明显。throw Throwables.propagate(ioException)做了什么？throw new RuntimeException(ioException)做了什么？这两者做了同样的事情，但后者的意思更简单直接。前者却引起了疑问：”它做了什么？它并不只是把异常包装进 RuntimeException 吧？如果它真的只做了包装，为什么还非得要写个方法？”。应该承认，这些问题部分是因为”propagate”的语义太模糊了（用来抛出未声明的异常吗？）。也许”wrapIfChecked”更能清楚地表达含义。但即使方法叫做”wrapIfChecked”，用它来包装一个已知类型的受检异常也没什么优点。甚至会有其他缺点：也许比起 RuntimeException，还有更合适的类型——如 IllegalArgumentException。 我们有时也会看到 propagate 被用于传播可能为受检的异常，结果是代码相比以前会稍微简短点，但也稍微有点不清晰：

```
} catch (RuntimeException e) {
    throw e;
}catch (Exception e) {
```

```
    throw new RuntimeException(e);
}
```

```
} catch (Exception e) {
 throw Throwables.propagate(e);
}
```

然而，我们似乎故意忽略了把检查型异常转化为非检查型异常的合理性。在某些场景中，这无疑是正确的做法，但更多时候它被用于避免处理受检异常。这让我们的话题变成了争论受检异常是不是坏主意了，我不想对此多做叙述。但可以这样说，Throwables.propagate 不是为了鼓励开发者忽略 IOException 这样的异常。

争议二：异常穿隧

但是，如果你要实现不允许抛出异常的方法呢？有时候你需要把异常包装在非受检异常内。这种做法挺好，但我们再次强调，没必要用 propagate 方法做这种简单的包装。实际上，手动包装可能更好：如果你手动包装了所有异常（而不仅仅是受检异常），那你就可以在另一端解包所有异常，并处理极少数特殊场景。此外，你可能还想把异常包装成特定的类型，而不是像 propagate 这样统一包装成 RuntimeException。

争议三：重新抛出其他线程产生的异常

```
try {
    return future.get();
} catch (ExecutionException e) {
    throw Throwables.propagate(e.getCause());
}
```

对这样的代码要考虑很多方面：

- ExecutionException 的 cause 可能是受检异常，见上文”争议一：把检查型异常转化为非检查型异常”。但如果我们确定 future 对应的任务不会抛出受检异常呢？（可能 future 表示 runnable 任务的结果——译者注：如 ExecutorService 中的 submit(Runnable task, T result)方法）如上所述，你可以捕获异常并抛出 AssertionError 。尤其对于 Future，请考虑 Futures.get 方法。（TODO：对 future.get()抛出的另一个异常 InterruptedException 作一些说明）
- ExecutionException 的 cause 可能直接是 Throwable 类型，而不是 Exception 或 Error。（实际上这不大可能，但你想直接重新抛出 cause 的话，编译器会强迫你考虑这种可能性）见上文”用法二：把抛出 Throwable 改为抛出 Exception”。

- ExecutionException 的 cause 可能是非受检异常。如果是这样的话，cause 会直接被 Throwables.propagate 抛出。不幸的是，cause 的堆栈信息反映的是异常最初产生的线程，而不是传播异常的线程。通常来说，最好在异常链中同时包含这两个线程的堆栈信息，就像 ExecutionException 所做的那样。（这个问题并不单单和 propagate 方法相关；所有在其他线程中重新抛出异常的代码都需要考虑这点）

异常原因链

Guava 提供了如下三个有用的方法，让研究异常的原因链变得稍微简便了，这三个方法的签名是不言自明的：

Throwable getRootCause(Throwable)
List<Throwable> getCausalChain(Throwable)
String getStackTraceAsString(Throwable)

2
集合

不可变集合

范例

```
public static final ImmutableSet<String> COLOR_NAMES = ImmutableSet.of(
        "red",
        "orange",
        "yellow",
        "green",
        "blue",
        "purple");


class Foo {
    Set<Bar> bars;
    Foo(Set<Bar> bars) {
        this.bars = ImmutableSet.copyOf(bars); // defensive copy!
    }
}
```

为什么要使用不可变集合

不可变对象有很多优点，包括：

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

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。

如果你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。如果你需要在不可变集合中使用 null，请使用 JDK 中的 Collections.unmodifiableXXX 方法。更多细节建议请参考“使用和避免 null”。*

怎么使用不可变集合

不可变集合可以用如下多种方式创建：

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

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

此外，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");
```

会在构造时就把元素排序为 a, b, c, d。

比想象中更智能的 copyOf

请注意，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：

- 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。
- 不会造成内存泄露——例如，你有个很大的不可变集合 ImmutableList 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。
- 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。

在可能的情况下避免线性拷贝，可以最大限度地减少防御性编程风格所带来的性能开销。

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。

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。

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

Guava 引入了很多 JDK 没有的、但我们发现明显有用的新集合类型。这些新类型是为了和 JDK 集合框架共存，而没有往 JDK 集合抽象中硬塞其他概念。作为一般规则，Guava 集合非常精准地遵循了 JDK 接口契约。

Multiset

统计一个词在文档中出现了多少次，传统的做法是这样的：

```
Map<String, Integer> counts = new HashMap<String, Integer>();
for (String word : words) {
   Integer count = counts.get(word);
   if (count == null) {
      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。维基百科从数学角度这样定义 Multiset：”集合[set]概念的延伸，它的元素可以重复出现…与集合[set]相同而与元组[tuple]相反的是，Multiset 元素的顺序是无关紧要的：Multiset {a, a, b}和{a, b, a}是相等的”。——*译者注：这里所说的集合[set]是数学上的概念，Multiset继承自 JDK 中的 Collection 接口，而不是 Set 接口，所以包含重复元素并没有违反原有的接口契约。*

可以用两种方式看待 Multiset：

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

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

- add(E)添加单个给定元素
- iterator()返回一个迭代器，包含 Multiset 的所有元素（包括重复的元素）

- size()返回所有元素的总个数（包括重复的元素）

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

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

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

方法	描述
count(E)	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）

Multiset 不是 Map

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

- Multiset 中的元素计数只能是正数。任何元素的计数都不能为负，也不能是 0。elementSet()和 entrySet()视图中也不会有这样的元素。
- multiset.size()返回集合的大小，等同于所有元素计数的总和。对于不重复元素的个数，应使用 elementSet().size()方法。（因此，add(E)把 multiset.size()增加 1）
- multiset.iterator()会迭代重复元素，因此迭代长度等于 multiset.size()。
- Multiset 支持直接增加、减少或设置元素的计数。setCount(elem, 0)等同于移除所有 elem。
- 对 multiset 中没有的元素，multiset.count(elem)始终返回 0。

Multiset 的各种实现

Guava 提供了多种 Multiset 的实现，大致对应 JDK 中 Map 的各种实现：

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

SortedMultiset

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

Multimap

每个有经验的 Java 程序员都在某处实现过 Map<K, List>或 Map<K, Set>，并且要忍受这个结构的笨拙。例如，Map<K, Set>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>，让你可以按另一种方式看待 Multimap。重要的是，不会有任何键映射到空集合：一个键要么至少到一个值，要么根本就不在 Multimap 中。

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

修改 Multimap

Multimap.get(key)以集合形式返回键所对应的值视图，即使没有任何对应的值，也会返回空集合。ListMultimap.get(key)返回 List，SetMultimap.get(key)返回 Set。

对值视图集合进行的修改最终都会反映到底层的 Multimap。例如：

```
Set<Person> aliceChildren = childrenMultimap.get(alice);
aliceChildren.clear();
aliceChildren.add(bob);
aliceChildren.add(carol);
```

其他（更直接地）修改 Multimap 的方法有：

方法签名	描述	等价于
put(K, V)	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)

Multimap 的视图

Multimap 还支持若干强大的视图：

- asMap为 Multimap<K, V>提供 Map<K,Collection>形式的视图。返回的 Map 支持 remove 操作，并且会反映到底层的 Multimap，但它不支持 put 或 putAll 操作。更重要的是，如果你想为 Multimap 中没有的键返回 null，而不是一个新的、可写的空集合，你就可以使用 asMap().get(key)。（你可以并且应当把 asMap.get(key)返回的结果转化为适当的集合类型——如 SetMultimap.asMap.get(key)的结果转为 Set，ListMultimap.asMap.get(key)的结果转为 List——Java 类型系统不允许 ListMultimap 直接为 asMap.get(key)返回 List——*译者注：也可以用 Multimaps 中的 asMap 静态方法帮你完成类型转换*）

- entries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>，虽然某些 Multimap 实现中可能使用了 map。它们之间的显著区别包括：

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

Multimap 的各种实现

Multimap 提供了多种形式的实现。在大多数要使用 Map<K, Collection>的地方，你都可以使用它们：

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

除了两个不可变形式的实现，其他所有实现都支持 null 键和 null 值

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

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

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

如果你想要更大的定制化，请用 Multimaps.newMultimap(Map, Supplier)或 list 和 set 版本，使用自定义的 Collection、List 或 Set 实现 Multimap。

BiMap

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

```
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

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)。

```
BiMap<String, Integer> userId = HashBiMap.create();
...

String userForId = userId.inverse().get(id);
```

BiMap 的各种实现

键 - 值实现	值 - 键实现	对应的BiMap实现
HashMap	HashMap	HashBiMap
ImmutableMap	ImmutableMap	ImmutableBiMap
EnumMap	EnumMap	EnumBiMap
EnumMap	HashMap	EnumHashBiMap

注：Maps 类中还有一些诸如 synchronizedBiMap 的 BiMap 工具方法.

Table

```
Table<Vertex, Vertex, Double> weightedGraph = HashBasedTable.create();
weightedGraph.put(v1, v2, 4);
weightedGraph.put(v1, v3, 20);
weightedGraph.put(v2, v3, 5);

weightedGraph.row(v1); // returns a Map mapping v2 to 4, v3 to 20
weightedGraph.column(v3); // returns a Map mapping v1 to 20, v2 to 5
```

通常来说，当你想使用多个键做索引的时候，你可能会用类似 Map<FirstName, Map<LastName, Person>>的实现，这种方式很丑陋，使用上也不友好。Guava 为此提供了新集合类型 Table，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。
- 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。

Table 有如下几种实现：

- 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，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。

为了扩展 Map 接口，ClassToInstanceMap 额外声明了两个方法：T getInstance(Class) 和 T putInstance(Class, T)，从而避免强制类型转换，同时保证了类型安全。

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

```
ClassToInstanceMap<Number> numberDefaults=MutableClassToInstanceMap.create();
numberDefaults.putInstance(Integer.class, Integer.valueOf(0));
```

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

对于 ClassToInstanceMap，Guava 提供了两种有用的实现：MutableClassToInstanceMap 和 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。例如：

```
RangeSet<Integer> rangeSet = TreeRangeSet.create();
rangeSet.add(Range.closed(1, 10)); // {[1,10]}
rangeSet.add(Range.closedOpen(11, 15));//不相连区间:{[1,10], [11,15)}
rangeSet.add(Range.closedOpen(15, 20)); //相连区间; {[1,10], [11,20)}
rangeSet.add(Range.openClosed(0, 0)); //空区间; {[1,10], [11,20)}
rangeSet.remove(Range.open(5, 10)); //分割[1, 10]; {[1,5], [10,10], [11,20)}
```

请注意，要合并 Range.closed(1, 10)和 Range.closedOpen(11, 15)这样的区间，你需要首先用 Range.canonical(DiscreteDomain)对区间进行预处理，例如 DiscreteDomain.integers()。

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

RangeSet 的视图

RangeSet 的实现支持非常广泛的视图：

- complement()：返回 RangeSet 的补集视图。complement 也是 RangeSet 类型,包含了不相连的、非空的区间。
- subRangeSet(Range)：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>表现 RangeSet，这样可以遍历其中的 Range。
- asSet(DiscreteDomain)（仅 ImmutableRangeSet 支持）：用 ImmutableSortedSet表现 RangeSet，以区间中所有元素的形式而不是区间本身的形式查看。（这个操作不支持 DiscreteDomain 和 RangeSet 都没有上边界，或都没有下边界的情况）

RangeSet 的查询方法

为了方便操作，RangeSet 直接提供了若干查询方法，其中最突出的有：

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

RangeMap

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

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

```
rangeMap.put(Range.open(3, 6), "bar"); //{[1,3] => "foo", (3,6) => "bar", [6,10] => "foo"}
rangeMap.put(Range.open(10, 20), "foo"); //{[1,3] => "foo", (3,6) => "bar", [6,10] => "foo", (10,20) => "foo"}
rangeMap.remove(Range.closed(5, 11)); //{[1,3] => "foo", (3,5) => "bar", (11,20) => "foo"}
```

RangeMap 的视图

RangeMap 提供两个视图：

- asMapOfRanges()：用 Map<Range, V>表现 RangeMap。这可以用来遍历 RangeMap。
- subRangeMap(Range)：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：java.util.Collections 中未包含的集合工具

尚未完成: Queues, Tables 工具类

任何对 JDK 集合框架有经验的程序员都熟悉和喜欢 java.util.Collections 包含的工具方法。Guava 沿着这些路线提供了更多的工具方法：适用于所有集合的静态方法。这是 Guava 最流行和成熟的部分之一。

我们用相对直观的方式把工具类与特定集合接口的对应关系归纳如下：

集合接口	属于JDK还是Guava	对应的Guava工具类
Collection	JDK	`Collections2`：不要和 java.util.Collections 混淆
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`

在找类似转化、过滤的方法？请看第四章，函数式风格。

静态工厂方法

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<TypeThatsTooLongForItsOwnGood> list = new ArrayList<TypeThatsTooLongForItsOwnGood>();
```

我想我们都认为这很讨厌。因此 Guava 提供了能够推断范型的静态工厂方法：

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

可以肯定的是，JDK7 版本的钻石操作符(<>)没有这样的麻烦：

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

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");
```

此外，通过为工厂方法命名（Effective Java 第一条），我们可以提高集合初始化大小的可读性：

```
List<Type> exactly100 = Lists.newArrayListWithCapacity(100);
List<Type> approx100 = Lists.newArrayListWithExpectedSize(100);
Set<Type> approx100Set = Sets.newHashSetWithExpectedSize(100);
```

确切的静态工厂方法和相应的工具类一起罗列在下面的章节。

注意：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，例如：

```
Multiset<String> multiset = HashMultiset.create();
```

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。

截至 Guava 1.2 版本，Iterables 使用 FluentIterable 类进行了补充，它包装了一个 Iterable 实例，并对许多操作提供了”fluent”（链式调用）语法。

下面列出了一些最常用的工具方法，但更多 Iterables 的函数式方法将在第四章讨论。

常规方法

`concat(Iterable<Iterable>)`	串联多个 iterables 的懒视图*	`concat(Iterable...)`

`frequency(Iterable, Object)`	返回对象在 iterable 中出现的次数	与 Collections.frequency (Collection, Object)比较；Multiset
`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	`Lists.partition(List, int)`；`paddedPartition(Iterable, int)`
`getFirst(Iterable, T default)`	返回 iterable 的第一个元素，若 iterable 为空则返回默认值	与Iterable.iterator(). next()比较;`FluentIterable.first()`
`getLast(Iterable)`	返回 iterable 的最后一个元素，若 iterable 为空则抛出NoSuchElementException	`getLast(Iterable, T default)`；`FluentIterable.last()`
`elementsEqual(Iterable, Iterable)`	如果两个 iterable 中的所有元素相等且顺序一致，返回 true	与 List.equals(Object)比较
`unmodifiableIterable(Iterable)`	返回 iterable 的不可变视图	与 Collections. unmodifiableCollection(Collection)比较
`limit(Iterable, int)`	限制 iterable 的元素个数限制给定值	`FluentIterable.limit(int)`
`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	`getOnlyElement(Iterable, T default)`

译者注：懒视图意味着如果还没访问到某个 iterable 中的元素，则不会对它进行串联操作。

```
Iterable<Integer> concatenated = Iterables.concat(
        Ints.asList(1, 2, 3),
        Ints.asList(4, 5, 6)); // concatenated包括元素 1, 2, 3, 4, 5, 6
String lastAdded = Iterables.getLast(myLinkedHashSet);
String theElement = Iterables.getOnlyElement(thisSetIsDefinitelyASingleton);
//如果set不是单元素集，就会出错了!
```

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	等价的 FluentIterable 方法
`addAll(Collection addTo, Iterable toAdd)`	Collection.addAll(Collection)	
`contains(Iterable, Object)`	Collection.contains(Object)	`FluentIterable.contains(Object)`

`removeAll(Iterable removeFrom, Collection toRemove)`	Collection.removeAll(Collection)	
`retainAll(Iterable removeFrom, Collection toRetain)`	Collection.retainAll(Collection)	
`size(Iterable)`	Collection.size()	`FluentIterable.size()`
`toArray(Iterable, Class)`	Collection.toArray(T[])	`FluentIterable.toArray(Class)`
`isEmpty(Iterable)`	Collection.isEmpty()	`FluentIterable.isEmpty()`
`get(Iterable, int)`	List.get(int)	`FluentIterable.get(int)`
`toString(Iterable)`	Collection.toString()	`FluentIterable.toString()`

FluentIterable

除了上面和第四章提到的方法，FluentIterable 还有一些便利方法用来把自己拷贝到不可变集合

ImmutableList	
ImmutableSet	`toImmutableSet()`
ImmutableSortedSet	`toImmutableSortedSet(Comparator)`

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。注: 如果 List 是不可变的，考虑改用 `ImmutableList.reverse()`。

```
List countUp = Ints.asList(1, 2, 3, 4, 5);
List countDown = Lists.reverse(theList); // {5, 4, 3, 2, 1}
List<List> parts = Lists.partition(countUp, 2);//{{1,2}, {3,4}, {5}}
```

静态工厂方法

Lists 提供如下静态工厂方法:

具体实现类型	工厂方法

ArrayList	basic, with elements, from `Iterable`, with exact capacity, with expected size, from `Iterator`
LinkedList	basic, from `Iterable`

Sets

Sets 工具类包含了若干好用的方法。

集合理论方法

我们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
`union(Set, Set)`
`intersection(Set, Set)`
`difference(Set, Set)`
`symmetricDifference(Set, Set)`

使用范例：

```
Set<String> wordsWithPrimeLength = ImmutableSet.of("one", "two", "three", "six", "seven", "eight");
Set<String> primes = ImmutableSet.of("two", "three", "five", "seven");
SetView<String> intersection = Sets.intersection(primes,wordsWithPrimeLength);
// intersection包含"two", "three", "seven"
return intersection.immutableCopy();//可以使用交集，但不可变拷贝的读取效率更高
```

其他 Set 工具方法

方法	描述	另请参见
`cartesianProduct(List<Set>)`	返回所有集合的笛卡儿积	`cartesianProduct(Set...)`
`powerSet(Set)`	返回给定集合的所有子集	

```
Set<String> animals = ImmutableSet.of("gerbil", "hamster");
Set<String> fruits = ImmutableSet.of("apple", "orange", "banana");
```

```
Set<List<String>> product = Sets.cartesianProduct(animals, fruits);
// {{"gerbil", "apple"}, {"gerbil", "orange"}, {"gerbil", "banana"},
//  {"hamster", "apple"}, {"hamster", "orange"}, {"hamster", "banana"}}

Set<Set<String>> animalSets = Sets.powerSet(animals);
// {{}, {"gerbil"}, {"hamster"}, {"gerbil", "hamster"}}
```

静态工厂方法

Sets 提供如下静态工厂方法：

具体实现类型	工厂方法
HashSet	basic, with elements, from `Iterable`, with expected size, from `Iterator`
LinkedHashSet	basic, from `Iterable`, with expected size
TreeSet	basic, with `Comparator`, from `Iterable`

Maps

Maps 类有若干值得单独说明的、很酷的方法。

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——*译者注：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。*

比方说，我们有一堆字符串，这些字符串的长度都是独一无二的，而我们希望能够按照特定长度查找字符串：

```
ImmutableMap<Integer, String> stringsByIndex = Maps.uniqueIndex(strings,
new Function<String, Integer> () {
   public Integer apply(String string) {
      return string.length();
   }
});
```

如果索引值不是独一无二的，请参见下面的 Multimaps.index 方法。

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

```
Map<String, Integer> left = ImmutableMap.of("a", 1, "b", 2, "c", 3);
Map<String, Integer> left = ImmutableMap.of("a", 1, "b", 2, "c", 3);
MapDifference<String, Integer> diff = Maps.difference(left, right);

diff.entriesInCommon(); // {"b" => 2}
diff.entriesInCommon(); // {"b" => 2}
diff.entriesOnlyOnLeft(); // {"a" => 1}
diff.entriesOnlyOnRight(); // {"d" => 5}
```

处理 BiMap 的工具方法

Guava 中处理 BiMap 的工具方法在 Maps 类中，因为 BiMap 也是一种 Map 实现。

BiMap工具方法	相应的 Map 工具方法
`synchronizedBiMap(BiMap)`	Collections.synchronizedMap(Map)
`unmodifiableBiMap(BiMap)`	Collections.unmodifiableMap(Map)

静态工厂方法

Maps 提供如下静态工厂方法：

具体实现类型	工厂方法
HashMap	basic, from `Map`, with expected size
LinkedHashMap	basic, from `Map`
TreeMap	basic, from `Comparator`, from `SortedMap`
EnumMap	from `Class`, from `Map`
ConcurrentMap：支持所有操作	basic
IdentityHashMap	basic

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

```
Multiset<String> multiset1 = HashMultiset.create();
multiset1.add("a", 2);

Multiset<String> multiset2 = HashMultiset.create();
multiset2.add("a", 5);

multiset1.containsAll(multiset2); //返回true；因为包含了所有不重复元素，
//虽然multiset1实际上包含2个"a"，而multiset2包含5个"a"
Multisets.containsOccurrences(multiset1, multiset2); // returns false

multiset2.removeOccurrences(multiset1); // multiset2 现在包含3个"a"
multiset2.removeAll(multiset1);//multiset2移除所有"a"，虽然multiset1只有2个"a"
multiset2.isEmpty(); // returns true
```

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

```
Multiset<String> multiset = HashMultiset.create();
multiset.add("a", 3);
```

```
multiset.add("b", 5);
multiset.add("c", 1);

ImmutableMultiset highestCountFirst = Multisets.copyHighestCountFirst(multiset);
//highestCountFirst，包括它的entrySet和elementSet，按{"b", "a", "c"}排列元素
```

Multimaps

Multimaps 提供了若干值得单独说明的通用工具方法

index

作为 Maps.uniqueIndex 的兄弟方法，[Multimaps.index(Iterable, Function)](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Multimaps.html#index(java.lang.Iterable, com.google.common.base.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。

比方说，我们想把字符串按长度分组。

```
ImmutableSet digits = ImmutableSet.of("zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine");
Function<String, Integer> lengthFunction = new Function<String, Integer>() {
public Integer apply(String string) {
return string.length();
}
};

ImmutableListMultimap<Integer, String> digitsByLength= Multimaps.index(digits, lengthFunction);
/*
*  digitsByLength maps:
*  3 => {"one", "two", "six"}
*  4 => {"zero", "four", "five", "nine"}
*  5 => {"three", "seven", "eight"}
*/
```

invertFrom

鉴于 Multimap 可以把多个键映射到同一个值（译者注：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。

注：如果你使用的是 ImmutableMultimap，考虑改用 ImmutableMultimap.inverse()做反转。

```
ArrayListMultimap<String, Integer> multimap = ArrayListMultimap.create();
multimap.putAll("b", Ints.asList(2, 4, 6));
multimap.putAll("a", Ints.asList(4, 2, 1));
multimap.putAll("c", Ints.asList(2, 5, 3));

TreeMultimap<Integer, String> inverse = Multimaps.invertFrom(multimap, TreeMultimap<String, Integer>.create());
//注意我们选择的实现，因为选了TreeMultimap，得到的反转结果是有序的
/*
* inverse maps:
*  1 => {"a"}
*  2 => {"a", "b", "c"}
*  3 => {"c"}
*  4 => {"a", "b"}
*  5 => {"c"}
*  6 => {"b"}
*/
```

forMap

想在 Map 对象上使用 Multimap 的方法吗? forMap(Map)把 Map 包装成 SetMultimap。这个方法特别有用，例如，与 Multimaps.invertFrom 结合使用，可以把多对一的 Map 反转为一对多的 Multimap。

```
Map<String, Integer> map = ImmutableMap.of("a", 1, "b", 1, "c", 2);
SetMultimap<String, Integer> multimap = Multimaps.forMap(map);
// multimap：["a" => {1}, "b" => {1}, "c" => {2}]
Multimap<Integer, String> inverse = Multimaps.invertFrom(multimap, HashMultimap<Integer, String>.create());
// inverse：[1 => {"a","b"}, 2 => {"c"}]
```

包装器

Multimaps 提供了传统的包装方法，以及让你选择 Map 和 Collection 类型以自定义 Multimap 实现的工具方法。

只读包装	`Multimap`	`ListMultimap`	`SetMultimap`	`SortedSetMultimap`
同步包装	`Multimap`	`ListMultimap`	`SetMultimap`	`SortedSetMultimap`
自定义实现	`Multimap`	`ListMultimap`	`SetMultimap`	`SortedSetMultimap`

自定义 Multimap 的方法允许你指定 Multimap 中的特定实现。但要注意的是：

- Multimap 假设对 Map 和 Supplier 产生的集合对象有完全所有权。这些自定义对象应避免手动更新，并且在提供给 Multimap 时应该是空的，此外还不应该使用软引用、弱引用或虚引用。
- 无法保证修改了 Multimap 以后，底层 Map 的内容是什么样的。

- 即使 Map 和 Supplier 产生的集合都是线程安全的，它们组成的 Multimap 也不能保证并发操作的线程安全性。并发读操作是工作正常的，但需要保证并发读写的话，请考虑用同步包装器解决。
- 只有当 Map、Supplier、Supplier 产生的集合对象、以及 Multimap 存放的键值类型都是可序列化的，Multimap 才是可序列化的。
- Multimap.get(key)返回的集合对象和 Supplier 返回的集合对象并不是同一类型。但如果 Supplier 返回的是随机访问集合，那么 Multimap.get(key)返回的集合也是可随机访问的。

请注意，用来自定义 Multimap 的方法需要一个 Supplier 参数，以创建崭新的集合。下面有个实现 ListMultimap 的例子——用 TreeMap 做映射，而每个键对应的多个值用 LinkedList 存储。

```
ListMultimap<String, Integer> myMultimap = Multimaps.newListMultimap(
    Maps.<String, Collection>newTreeMap(),
    new Supplier<LinkedList>() {
        public LinkedList get() {
            return Lists.newLinkedList();
        }
    });
```

Tables

Tables 类提供了若干称手的工具方法。

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。

```
// 使用LinkedHashMaps替代HashMaps
Table<String, Character, Integer> table = Tables.newCustomTable(
Maps.<String, Map<Character, Integer>>newLinkedHashMap(),
new Supplier<Map<Character, Integer>> () {
public Map<Character, Integer> get() {
return Maps.newLinkedHashMap();
}
});
```

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。

包装器

还有很多你熟悉和喜欢的 Table 包装类。然而，在大多数情况下还请使用 ImmutableTable

Unmodifiable	`Table`	`RowSortedTable`

集合扩展工具类

简介

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。（ 毕竟，我们自己也要用这些工具扩展集合框架。）

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——译者注：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。

让我们看看这个例子。假定你想装饰一个 List，让其记录所有添加进来的元素。当然，无论元素是用什么方法——add(int, E), add(E), 或 addAll(Collection)——添加进来的，我们都希望进行记录，因此我们需要覆盖所有这些方法。

```
class AddLoggingList<E> extends ForwardingList<E> {
  final List<E> delegate; // backing list
  @Override protected List<E> delegate() {
    return delegate;
  }
  @Override public void add(int index, E elem) {
    log(index, elem);
    super.add(index, elem);
  }
```

```
    @Override public boolean add(E elem) {
      return standardAdd(elem); // 用add(int, E)实现
    }
    @Override public boolean addAll(Collection<? extends E> c) {
      return standardAddAll(c); // 用add实现
    }
}
```

记住，默认情况下，所有方法都直接转发到被代理对象，因此覆盖 ForwardingMap.put 并不会改变 ForwardingMap.putAll 的行为。小心覆盖所有需要改变行为的方法，并且确保装饰后的集合满足接口契约。

通常来说，类似于 AbstractList 的抽象集合骨架类，其大多数方法在 Forwarding 装饰器中都有对应的”标准方法”实现。

对提供特定视图的接口，Forwarding 装饰器也为这些视图提供了相应的”标准方法”实现。例如，ForwardingMap 提供 StandardKeySet、StandardValues 和 StandardEntrySet 类，它们在可以的情况下都会把自己的方法委托给被装饰的 Map，把不能委托的声明为抽象方法。

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。

举个例子：复制一个 List，并去除连续的重复元素。

```
List<E> result = Lists.newArrayList();
PeekingIterator<E> iter = Iterators.peekingIterator(source.iterator());
while (iter.hasNext()) {
  E current = iter.next();
  while (iter.hasNext() && iter.peek().equals(current)) {
    //跳过重复的元素
    iter.next();
  }
  result.add(current);
}
```

传统的实现方式需要记录上一个元素，并在特定情况下后退，但这很难处理且容易出错。相较而言， PeekingIterator 在理解和使用上就比较直接了。

AbstractIterator

实现你自己的 Iterator? AbstractIterator 让生活更轻松。

用一个例子来解释 AbstractIterator 最简单。比方说，我们要包装一个 iterator 以跳过空值。

```
public static Iterator<String> skipNulls(final Iterator<String> in) {
   return new AbstractIterator<String>() {
      protected String computeNext() {
         while (in.hasNext()) {
            String s = in.next();
            if (s != null) {
               return s;
            }
         }
         return endOfData();
      }
   };
}
```

你实现了 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 endOfDat a()表明已经到达迭代的末尾。

注意：AbstractIterator 继承了 UnmodifiableIterator，所以禁止实现 remove()方法。如果你需要支持 remov e()的迭代器，就不应该继承 AbstractIterator。

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。

```
Iterator<Integer> powersOfTwo = new AbstractSequentialIterator<Integer>(1) { // 注意初始值1!
   protected Integer computeNext(Integer previous) {
   return (previous == 1 << 30) ? null : previous * 2;
}
};
```

我们在这儿实现了 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。

3

缓存

范例

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

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。

通常来说，Guava Cache 适用于：

- 你愿意消耗一些内存空间来提升速度。
- 你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。如果这不符合你的需求，请尝试 Memcached 这类工具）

如果你的场景符合上述的每一条，Guava Cache 就适合你。

如同范例代码展示的一样，Cache 实例通过 CacheBuilder 生成器模式获取，但是自定义你的缓存才是最有趣的部分。

注：如果你不需要 Cache 中的特性，使用 ConcurrentHashMap 有更好的内存效率——但 Cache 的大多数特性都很难基于旧有的 ConcurrentMap 复制，甚至根本不可能做到。

加载

在使用缓存前，首先问自己一个问题：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，你应当使用 CacheLoader。如果没有，或者你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，你应该在调用 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 通常只需要简单地实现 V load(K key) throws Exception 方法。例如，你可以用下面的代码构建 LoadingCache：

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

...
try {
   return graphs.get(key);
} catch (ExecutionException e) {
   throw new OtherException(e.getCause());
}
```

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)。

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
       .expireAfterAccess(10, TimeUnit.MINUTES)
       .build(
          new CacheLoader<Key, Graph>() {
             public Graph load(Key key) { // no checked exception
                return createExpensiveGraph(key);
```

```
            }
          });

...
return graphs.getUnchecked(key);
```

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。

注：CacheLoader.loadAll 的实现可以为没有明确请求的键加载缓存值。例如，为某组中的任意键计算值时，能够获取该组中的所有键值，loadAll 方法就可以实现为在同一时间获取该组的其他键值。校注：getAll(Iterable<? extends K>)方法会调用 loadAll，但会筛选结果，只会返回请求的键值对。

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)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"。

```
Cache<Key, Graph> cache = CacheBuilder.newBuilder()
       .maximumSize(1000)
       .build(); // look Ma, no CacheLoader
...
try {
   // If the key wasn't in the "easy to compute" group, we need to
   // do things the hard way.
   cache.get(key, new Callable<Key, Graph>() {
      @Override
      public Value call() throws AnyException {
         return doThingsTheHardWay(key);
      }
   });
} catch (ExecutionException e) {
   throw new OtherException(e.getCause());
}
```

显式插入

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) 应该总是优先使用。

缓存回收

一个残酷的现实是，我们几乎一定没有足够的内存缓存所有数据。你你必须决定：什么时候某个缓存项就不值得保留了？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（size-based eviction）

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。——警告：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。

另外，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。

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
   .maximumWeight(100000)
   .weigher(new Weigher<Key, Graph>() {
      public int weigh(Key k, Graph g) {
         return g.vertices().size();
      }
   })
   .build(
      new CacheLoader<Key, Graph>() {
         public Graph load(Key key) { // no checked exception
            return createExpensiveGraph(key);
         }
      });
```

定时回收（Timed Eviction）

CacheBuilder 提供两种定时回收的方法：

- 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。

- 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。

如下文所讨论，定时回收周期性地在写操作中执行，偶尔在读操作中执行。

测试定时回收

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（Reference-based Eviction）

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。

显式清除

任何时候，你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()

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。

请注意，RemovalListener 抛出的任何异常都会在记录到日志后被丢弃[swallowed]。

```
CacheLoader<Key, DatabaseConnection> loader = new CacheLoader<Key, DatabaseConnection> () {
public DatabaseConnection load(Key key) throws Exception {
return openConnection(key);
}
};

RemovalListener<Key, DatabaseConnection> removalListener = new RemovalListener<Key, DatabaseConnection>()
public void onRemoval(RemovalNotification<Key, DatabaseConnection> removal) {
DatabaseConnection conn = removal.getValue();
conn.close(); // tear down properly
}
};

return CacheBuilder.newBuilder()
.expireAfterWrite(2, TimeUnit.MINUTES)
.removalListener(removalListener)
.build(loader);
```

警告：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，我们把选择权交到你手里。如果你的缓存是高吞吐的，那就无需担心缓存的维护和清理等工作。如果你的缓存只会偶尔有写操作，而你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 可以帮助你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[swallowed]。

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。

```
//有些键不需要刷新，并且我们希望刷新是异步完成的
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
   .maximumSize(1000)
   .refreshAfterWrite(1, TimeUnit.MINUTES)
   .build(
      new CacheLoader<Key, Graph>() {
         public Graph load(Key key) { // no checked exception
            return getGraphFromDatabase(key);
         }

         public ListenableFuture<Key, Graph> reload(final Key key, Graph prevGraph) {
            if (neverNeedsRefresh(key)) {
               return Futures.immediateFuture(prevGraph);
            }else{
               // asynchronous!
               ListenableFutureTask<Key, Graph> task=ListenableFutureTask.create(new Callable<Key, Graph>() {
                  public Graph call() {
                     return getGraphFromDatabase(key);
                  }
               });
               executor.execute(task);
               return task;
            }
         }
      });
```

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。4

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。

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。

中断

缓存加载方法（如 Cache.get）不会抛出 InterruptedException。我们也可以让这些方法支持 InterruptedException，但这种支持注定是不完备的，并且会增加所有使用者的成本，而只有少数使用者实际获益。详情请继续阅读。

Cache.get 请求到未缓存的值时会遇到两种情况：当前线程加载值；或等待另一个正在加载值的线程。这两种情况下的中断是不一样的。等待另一个正在加载值的线程属于较简单的情况：使用可中断的等待就实现了中断支

持；但当前线程加载值的情况就比较复杂了：因为加载值的 CacheLoader 是由用户提供的，如果它是可中断的，那我们也可以实现支持中断，否则我们也无能为力。

如果用户提供的 CacheLoader 是可中断的，为什么不让 Cache.get 也支持中断？从某种意义上说，其实是支持的：如果 CacheLoader 抛出 InterruptedException，Cache.get 将立刻返回（就和其他异常情况一样）；此外，在加载缓存值的线程中，Cache.get 捕捉到 InterruptedException 后将恢复中断，而其他线程中 InterruptedException 则被包装成了 ExecutionException。

原则上，我们可以拆除包装，把 ExecutionException 变为 InterruptedException，但这会让所有的 LoadingCache 使用者都要处理中断异常，即使他们提供的 CacheLoader 不是可中断的。如果你考虑到所有非加载线程的等待仍可以被中断，这种做法也许是值得的。但许多缓存只在单线程中使用，它们的用户仍然必须捕捉不可能抛出的 InterruptedException 异常。即使是那些跨线程共享缓存的用户，也只是有时候能中断他们的 get 调用，取决于那个线程先发出请求。

对于这个决定，我们的指导原则是让缓存始终表现得好像是在当前线程加载值。这个原则让使用缓存或每次都计算值可以简单地相互切换。如果老代码（加载值的代码）是不可中断的，那么新代码（使用缓存加载值的代码）多半也应该是不可中断的。

如上所述，Guava Cache 在某种意义上支持中断。另一个意义上说，Guava Cache 不支持中断，这使得 LoadingCache 成了一个有漏洞的抽象：当加载过程被中断了，就当作其他异常一样处理，这在大多数情况下是可以的；但如果多个线程在等待加载同一个缓存项，即使加载线程被中断了，它也不应该让其他线程都失败（捕获到包装在 ExecutionException 里的 InterruptedException），正确的行为是让剩余的某个线程重试加载。为此，我们记录了一个 bug。然而，与其冒着风险修复这个 bug，我们可能会花更多的精力去实现另一个建议 AsyncLoadingCache，这个实现会返回一个有正确中断行为的 Future 对象。

4

函数式编程

注意事项

截至 JDK7，Java 中也只能通过笨拙冗长的匿名类来达到近似函数式编程的效果。预计 JDK8 中会有所改变，但 Guava 现在就想给 JDK5 以上用户提供这类支持。

过度使用 Guava 函数式编程会导致冗长、混乱、可读性差而且低效的代码。这是迄今为止最容易（也是最经常）被滥用的部分，如果你想通过函数式风格达成一行代码，致使这行代码长到荒唐，Guava 团队会泪流满面。

比较如下代码：

```
Function<String, Integer> lengthFunction = new Function<String, Integer>() {
   public Integer apply(String string) {
   return string.length();
}
};
Predicate<String> allCaps = new Predicate<String>() {
   public boolean apply(String string) {
   return CharMatcher.JAVA_UPPER_CASE.matchesAllOf(string);
}
};
Multiset<Integer> lengths = HashMultiset.create(
    Iterables.transform(Iterables.filter(strings, allCaps), lengthFunction));
```

或 FluentIterable 的版本

```
Multiset<Integer> lengths = HashMultiset.create(
   FluentIterable.from(strings)
      .filter(new Predicate<String>() {
         public boolean apply(String string) {
            return CharMatcher.JAVA_UPPER_CASE.matchesAllOf(string);
         }
      })
      .transform(new Function<String, Integer>() {
         public Integer apply(String string) {
            return string.length();
         }
      }));
```

还有

```
Multiset<Integer> lengths = HashMultiset.create();
for (String string : strings) {
  if (CharMatcher.JAVA_UPPER_CASE.matchesAllOf(string)) {
    lengths.add(string.length());
  }
}
```

即使用了静态导入，甚至把 Function 和 Predicate 的声明放到别的文件，第一种代码实现仍然不简洁，可读性差并且效率较低。

截至 JDK7，命令式代码仍应是默认和第一选择。不应该随便使用函数式风格，除非你绝对确定以下两点之一：

- 使用函数式风格以后，整个工程的代码行会净减少。在上面的例子中，函数式版本用了 11 行， 命令式代码用了 6 行，把函数的定义放到另一个文件或常量中，并不能帮助减少总代码行。
- 为了提高效率，转换集合的结果需要懒视图，而不是明确计算过的集合。此外，确保你已经阅读和重读了 Effective Java 的第 55 条，并且除了阅读本章后面的说明，你还真正做了性能测试并且有测试数据来证明函数式版本更快。

请务必确保，当使用 Guava 函数式的时候，用传统的命令式做同样的事情不会更具可读性。尝试把代码写下来，看看它是不是真的那么糟糕？会不会比你想尝试的极其笨拙的函数式 更具可读性。

Functions[函数]和 Predicates[断言]

本节只讨论直接与 Function 和 Predicate 打交道的 Guava 功能。一些其他工具类也和”函数式风格”相关，例如 Iterables.concat(Iterable)，和其他用常量时间返回视图的方法。尝试看看 2.3 节的集合工具类。

Guava 提供两个基本的函数式接口：

- Function<A, B>，它声明了单个方法 B apply(A input)。Function 对象通常被预期为引用透明的——没有副作用——并且引用透明性中的”相等”语义与 equals 一致，如 a.equals(b)意味着 function.apply(a).equals(function.apply(b))。
- Predicate，它声明了单个方法 boolean apply(T input)。Predicate 对象通常也被预期为无副作用函数，并且”相等”语义与 equals 一致。

特殊的断言

字符类型有自己特定版本的 Predicate——CharMatcher，它通常更高效，并且在某些需求方面更有用。CharMatcher 实现了 Predicate，可以当作 Predicate 一样使用，要把 Predicate 转成 CharMatcher，可以使用 CharMatcher.forPredicate。更多细节请参考第 6 章–字符串处理。

此外，对可比较类型和基于比较逻辑的 Predicate，Range 类可以满足大多数需求——它表示一个不可变区间。Range 类实现了 Predicate，用以判断值是否在区间内。例如，Range.atMost(2)就是个完全合法的 Predicate。更多使用 Range 的细节请参照第8章。

操作 Functions 和 Predicates

Functions 提供简便的 Function 构造和操作方法，包括：

`forMap(Map<A, B>)`	`compose(Function<B, C>, Function<A, B>)`	`constant(T)`
`identity()`	`toStringFunction()`	

细节请参考 Javadoc。

相应地，Predicates 提供了更多构造和处理 Predicate 的方法，下面是一些例子：

`instanceOf(Class)`	`assignableFrom(Class)`	`contains(Pattern)`
`in(Collection)`	`isNull()`	`alwaysFalse()`
`alwaysTrue()`	`equalTo(Object)`	`compose(Predicate, Function)`

`and(Predicate...)`	`or(Predicate...)`	`not(Predicate)`

细节请参考Javadoc。

使用函数式编程

Guava 提供了很多工具方法，以便用 Function 或 Predicate 操作集合。这些方法通常可以在集合工具类找到，如 Iterables，Lists，Sets，Maps，Multimaps 等。

断言

断言的最基本应用就是过滤集合。所有 Guava 过滤方法都返回”视图”——*译者注：即并非用一个新的集合表示过滤，而只是基于原集合的视图。*

集合类型	过滤方法
Iterable	`Iterables.filter(Iterable, Predicate)FluentIterable.filter(Predicate)`
Iterator	`Iterators.filter(Iterator, Predicate)`
Collection	`Collections2.filter(Collection, Predicate)`
Set	`Sets.filter(Set, Predicate)`
SortedSet	`Sets.filter(SortedSet, Predicate)`
Map	`Maps.filterKeys(Map, Predicate)Maps.filterValues(Map, Predicate)Maps.filterEntries(Map, Predicate)`
SortedMap	`Maps.filterKeys(SortedMap, Predicate)Maps.filterValues(SortedMap, Predicate)Maps.filterEntries(SortedMap, Predicate)`
Multimap	`Multimaps.filterKeys(Multimap, Predicate)Multimaps.filterValues(Multimap, Predicate)Multimaps.filterEntries(Multimap, Predicate)`

*List 的过滤视图被省略了，因为不能有效地支持类似 get(int)的操作。请改用 Lists.newArrayList(Collections2.filter(list, predicate))做拷贝过滤。

除了简单过滤，Guava 另外提供了若干用 Predicate 处理 Iterable 的工具——通常在 Iterables 工具类中，或者是 FluentIterable 的”fluent”（链式调用）方法。

Iterables方法签名	说明	另请参见
`boolean all(Iterable, Predicate)`	是否所有元素满足断言？懒实现：如果发现有元素不满足，不会继续迭代	`Iterators.all(Iterator, Predicate)FluentIterable.allMatch(Predicate)`
`boolean any(Iterable, Predicate)`	是否有任意元素满足元素满足断言？懒实现：只会迭代到发现满足的元素	`Iterators.any(Iterator, Predicate)FluentIterable.anyMatch(Predicate)`

`T find(Iterable, Predicate)`	循环并返回一个满足元素满足断言的元素，如果没有则抛出 NoSuchElementException	`Iterators.find(Iterator, Predicate)` `Iterables.find(Iterable, Predicate, T default)` `Iterators.find(Iterator, Predicate, T default)`
`Optional<T> tryFind(Iterable, Predicate)`	返回一个满足元素满足断言的元素，若没有则返回0ptional.absent()	`Iterators.find(Iterator, Predicate)` `Iterables.find(Iterable, Predicate, T default)` `Iterators.find(Iterator, Predicate, T default)`
`indexOf(Iterable, Predicate)`	返回第一个满足元素满足断言的元素索引值，若没有返回-1	`Iterators.indexOf(Iterator, Predicate)`
`removeIf(Iterable, Predicate)`	移除所有满足元素满足断言的元素，实际调用Iterator.remove()方法	`Iterators.removeIf(Iterator, Predicate)`

函数

到目前为止，函数最常见的用途为转换集合。同样，所有的 Guava 转换方法也返回原集合的视图。

集合类型	转换方法
Iterable	`Iterables.transform(Iterable, Function)FluentIterable.transform(Function)`
Iterator	`Iterators.transform(Iterator, Function)`
Collection	`Collections2.transform(Collection, Function)`
List	`Lists.transform(List, Function)`
Map*	`Maps.transformValues(Map, Function)Maps.transformEntries(Map, EntryTransformer)`
SortedMap*	`Maps.transformValues(SortedMap, Function)Maps.transformEntries(SortedMap, EntryTransformer)`
Multimap*	`Multimaps.transformValues(Multimap, Function)Multimaps.transformEntries(Multimap, EntryTransformer)`
ListMultimap*	`Multimaps.transformValues(ListMultimap, Function)Multimaps.transformEntries(ListMultimap, EntryTransformer)`
Table	`Tables.transformValues(Table, Function)`

*Map 和 Multimap 有特殊的方法，其中有个 EntryTransformer<K, V1, V2>参数，它可以使用旧的键值来计算，并且用计算结果替换旧值。

对 Set 的转换操作被省略了，因为不能有效支持 contains(Object)操作——译者注：懒视图实际上不会全部计算转换后的 Set 元素，因此不能高效地支持 contains(Object)。*请改用 Sets.newHashSet(Collections2.transform(set, function))进行拷贝转换。

```
List<String> names;
Map<String, Person> personWithName;
List<Person> people = Lists.transform(names, Functions.forMap(personWithName));

ListMultimap<String, String> firstNameToLastNames;
// maps first names to all last names of people with that first name

ListMultimap<String, String> firstNameToName = Multimaps.transformEntries(firstNameToLastNames,
   new EntryTransformer<String, String, String> () {
      public String transformEntry(String firstName, String lastName) {
         return firstName + " " + lastName;
      }
   });
```

可以组合 Function 使用的类包括：

Ordering	`Ordering.onResultOf(Function)`
Predicate	`Predicates.compose(Predicate, Function)`
Equivalence	`Equivalence.onResultOf(Function)`
Supplier	`Suppliers.compose(Function, Supplier)`
Function	`Functions.compose(Function, Function)`

此外，ListenableFuture API 支持转换 ListenableFuture。Futures 也提供了接受 AsyncFunction 参数的方法。

AsyncFunction 是 Function 的变种，它允许异步计算值。

`Futures.transform(ListenableFuture, Function)`
`Futures.transform(ListenableFuture, Function, Executor)`
`Futures.transform(ListenableFuture, AsyncFunction)`
`Futures.transform(ListenableFuture, AsyncFunction, Executor)`

5

并发

google Guava 包的 ListenableFuture 解析

并发编程是一个难题，但是一个强大而简单的抽象可以显著的简化并发的编写。出于这样的考虑，Guava 定义了 ListenableFuture 接口并继承了 JDK concurrent 包下的 Future 接口。

我们强烈地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（Callbacks）

多数用户喜欢使用 `Futures.addCallback(ListenableFuture<V>, FutureCallback<V>, Executor)`???, 或者 另外一个版本version（译者注：addCallback(ListenableFuture<V> future,FutureCallback<? super V> callback)），默认是采用 MoreExecutors.sameThreadExecutor()线程池, 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。

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。

```
ListeningExecutorService service = MoreExecutors.listeningDecorator(Executors.newFixedThreadPool(10));
ListenableFuture explosion = service.submit(new Callable() {
  public Explosion call() {
return pushBigRedButton();
  }
});
Futures.addCallback(explosion, new FutureCallback() {
  // we want this handler to run immediately after we push the big red button!
  public void onSuccess(Explosion explosion) {
walkAwayFrom(explosion);
  }
  public void onFailure(Throwable thrown) {
battleArchNemesis(); // escaped the explosion!
  }
});
```

另外, 假如你是从 FutureTask 转换而来的, Guava 提供 ListenableFutureTask.create(Callable) 和 `ListenableFutureTask.create(Runnable, V)`. 和 JDK 不同的是, ListenableFutureTask 不能随意被继承（译者注：ListenableFutureTask 中的 done 方法实现了调用 listener 的操作）。

假如你喜欢抽象的方式来设置 future 的值，而不是想实现接口中的方法，可以考虑继承抽象类 AbstractFuture 或者直接使用 SettableFuture 。

假如你必须将其他 API 提供的 Future 转换成 ListenableFuture，你没有别的方法只能采用硬编码的方式 JdkFutureAdapters.listenInPoolThread(Future) 来将 Future 转换成 ListenableFuture。尽可能地采用修改原生的代码返回 ListenableFuture 会更好一些。

Application

使用 ListenableFuture 最重要的理由是它可以进行一系列的复杂链式的异步操作。

```
ListenableFuture rowKeyFuture = indexService.lookUp(query);
AsyncFunction<RowKey, QueryResult> queryFunction =
new AsyncFunction<RowKey, QueryResult>() {
public ListenableFuture apply(RowKey rowKey) {
return dataService.read(rowKey);
}
};
ListenableFuture queryFuture = Futures.transform(rowKeyFuture, queryFunction, queryExecutor);
```

其他更多的操作可以更加有效的支持而 JDK 中的 Future 是没法支持的.

不同的操作可以在不同的 Executors 中执行，单独的 ListenableFuture 可以有多个操作等待。

当一个操作开始的时候其他的一些操作也会尽快开始执行 - “fan-out” - ListenableFuture 能够满足这样的场景：促发所有的回调（callbacks）。反之更简单的工作是，同样可以满足“fan-in”场景，促发 ListenableFuture 获取（get）计算结果，同时其它的 Futures 也会尽快执行：可以参考 the implementation of Futures.allAsList 。（译者注：fan-in 和 fan-out 是软件设计的一个术语，可以参考这里：http://baike.baidu.com/view/388892.htm#1 或者看这里的解析 Design Principles: Fan-In vs Fan-Out，这里 fan-out 的实现就是封装的 ListenableFuture 通过回调，调用其它代码片段。fan-in 的意义是可以调用其它 Future）

方法	描述	参考
`transform(ListenableFuture<A>, AsyncFunction<A, B>, Executor)`*	`??????` `ListenableFuture`，该 `ListenableFuture` 返回的 result 是由传入的 `AsyncFunction` 参数指派到传入的 `ListenableFuture` ?.	`transform(ListenableFuture<A>, AsyncFunction<A, B>)`
`transform(ListenableFuture<A>, Function<A, B>, Executor)`	`??????` `ListenableFuture`，该 `ListenableFuture` 返回的 result 是由传入的`Function` 参数指派到传入的 `ListenableFuture` ?.	`transform(ListenableFuture<A>, Function<A, B>)`
`allAsList(Iterable<ListenableFuture<V>>)`	`????` `ListenableFuture`，该`ListenableFuture` 返回的 result 是一个 List，List 中的值是每个 ListenableFuture 的返回值，假如传入的其中之一 fails 或者 cancel，这 个Future fails 或者 canceled	`allAsList(ListenableFuture<V>...)`
`successfulAsList(Iterable<ListenableFuture<V>>)`	`????` `ListenableFuture`，该 Future 的结果包含所有成功的 Future，按照原来的顺序，当其中之一 Failed 或者 cancel，则用 null 替代	`successfulAsList(ListenableFuture<V>...)`

AsyncFunction<A, B> 中提供一个方法 `ListenableFuture<B> apply(A input)`????????????

```
List<ListenableFuture> queries;
// The queries go to all different data centers, but we want to wait until they're all done or failed.

ListenableFuture<List> successfulQueries = Futures.successfulAsList(queries);
```

```
Futures.addCallback(successfulQueries, callbackOnSuccessfulQueries);
```

CheckedFuture

Guava ???? `CheckedFuture<V, X extends Exception>` ???CheckedFuture ??? ListenableFuture ?????????? get ????????????.???????????????????? Future ???? ?? ListenableFuture ??? CheckedFuture????? `Futures.makeChecked(ListenableFuture<V>, Function<Exception, X>)`?

Google-Guava Concurrent 包里的 Service 框架浅析

概述

Guava ??? Service ????????????????? start ? stop ?????? web ????RPC ??????????
????????????????????????????/????????????????????????Guava ????????????????????
???????

使用一个服务

???????????

- Service.State.NEW
- Service.State.STARTING
- Service.State.RUNNING
- Service.State.STOPPING
- Service.State.TERMINATED

????????????????????? starting?running?stopping ?????????? Service.State.FAILED.?
???? startAsync()???????????,???? this ????????????????????? NEW ????? startAs
ync()??? stopAsync() ????? startAsyn
c(),???????????????????????????????

Service 也提供了一些方法用于等待服务状态转换的完成:

?? addListener()????????????????????? Service.Listener ????????????????????????
????? Listener??????? NEW????????????????????? Listener????????

???? awaitRunning()??? IllegalStateExcept
ion ??????? awaitTerminated() ?????????????TERMINATED ?? FAILED????????????????
???

Service ??? Guava ????????????????
?????????????

基础实现类

AbstractIdleService

`AbstractIdleService` ?????? Service ????? running ???????????–??? running ????????–???????/????????????????????? AbstractIdleService ???????? `startUp()` ? `shutDown()`???????

```
protected void startUp() {
servlets.add(new GcStatsServlet());
}
protected void shutDown() {}
```

????????????? GcStatsServlet ????????????????????????????????

AbstractExecutionThreadService

`AbstractExecutionThreadService` ?????????????????????? run()?????????????????????????????

```
public void run() {
  while (isRunning()) {
    // perform a unit of work
  }
}
```

????????? `triggerShutdown()`??? run()???????

?? startUp()? shutDown()???????????????????

```
protected void startUp() {
dispatcher.listenForConnections(port, queue);
 }
 protected void run() {
   Connection connection;
   while ((connection = queue.take() != POISON)) {
 process(connection);
   }
 }
```

```
protected void triggerShutdown() {
 dispatcher.stopListeningForConnections(queue);
 queue.put(POISON);
}
```

start()????? startUp()???????????????????? run()???stop()??? triggerShutdown()?????? ?????

AbstractScheduledService

AbstractScheduledService ????????????????????????? runOneIteration()????????????????? startUp()? shutDown()????????????????????? scheduler()????????????? AbstractScheduledService.Scheduler ?????????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；具体用法见 javadoc。

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。

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，你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。

我们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。

附：TestCase、也可以作为练习 Demo

ServiceTest

```
</pre>
/*
 * Copyright (C) 2013 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

```
package com.google.common.util.concurrent;

import static com.google.common.util.concurrent.Service.State.FAILED;
import static com.google.common.util.concurrent.Service.State.NEW;
import static com.google.common.util.concurrent.Service.State.RUNNING;
import static com.google.common.util.concurrent.Service.State.STARTING;
import static com.google.common.util.concurrent.Service.State.STOPPING;
import static com.google.common.util.concurrent.Service.State.TERMINATED;

import junit.framework.TestCase;

/**
 * Unit tests for {@link Service}
 */
public class ServiceTest extends TestCase {

/** Assert on the comparison ordering of the State enum since we guarantee it. */
 public void testStateOrdering() {
 // List every valid (direct) state transition.
 assertLessThan(NEW, STARTING);
 assertLessThan(NEW, TERMINATED);

 assertLessThan(STARTING, RUNNING);
 assertLessThan(STARTING, STOPPING);
 assertLessThan(STARTING, FAILED);

 assertLessThan(RUNNING, STOPPING);
 assertLessThan(RUNNING, FAILED);

 assertLessThan(STOPPING, FAILED);
 assertLessThan(STOPPING, TERMINATED);
 }

 private static <T extends Comparable<? super T>> void assertLessThan(T a, T b) {
 if (a.compareTo(b) >= 0) {
 fail(String.format("Expected %s to be less than %s", a, b));
 }
 }
}
<pre>
```

AbstractIdleServiceTest

```
/*
```

```
 * Copyright (C) 2009 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.google.common.util.concurrent;

import static org.truth0.Truth.ASSERT;

import com.google.common.collect.Lists;

import junit.framework.TestCase;

import java.util.List;
import java.util.concurrent.Executor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

/**
 * Tests for {@link AbstractIdleService}.
 *
 * @author Chris Nokleberg
 * @author Ben Yu
 */
public class AbstractIdleServiceTest extends TestCase {

// Functional tests using real thread. We only verify publicly visible state.
 // Interaction assertions are done by the single-threaded unit tests.

public static class FunctionalTest extends TestCase {

private static class DefaultService extends AbstractIdleService {
 @Override protected void startUp() throws Exception {}
 @Override protected void shutDown() throws Exception {}
 }
```

```
public void testServiceStartStop() throws Exception {
 AbstractIdleService service = new DefaultService();
 service.startAsync().awaitRunning();
 assertEquals(Service.State.RUNNING, service.state());
 service.stopAsync().awaitTerminated();
 assertEquals(Service.State.TERMINATED, service.state());
 }

public void testStart_failed() throws Exception {
 final Exception exception = new Exception("deliberate");
 AbstractIdleService service = new DefaultService() {
 @Override protected void startUp() throws Exception {
 throw exception;
 }
 };
 try {
 service.startAsync().awaitRunning();
 fail();
 } catch (RuntimeException e) {
 assertSame(exception, e.getCause());
 }
 assertEquals(Service.State.FAILED, service.state());
 }

public void testStop_failed() throws Exception {
 final Exception exception = new Exception("deliberate");
 AbstractIdleService service = new DefaultService() {
 @Override protected void shutDown() throws Exception {
 throw exception;
 }
 };
 service.startAsync().awaitRunning();
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (RuntimeException e) {
 assertSame(exception, e.getCause());
 }
 assertEquals(Service.State.FAILED, service.state());
 }
 }

public void testStart() {
 TestService service = new TestService();
```

```
 assertEquals(0, service.startUpCalled);
 service.startAsync().awaitRunning();
 assertEquals(1, service.startUpCalled);
 assertEquals(Service.State.RUNNING, service.state());
 ASSERT.that(service.transitionStates).has().exactly(Service.State.STARTING).inOrder();
 }

public void testStart_failed() {
 final Exception exception = new Exception("deliberate");
 TestService service = new TestService() {
 @Override protected void startUp() throws Exception {
 super.startUp();
 throw exception;
 }
 };
 assertEquals(0, service.startUpCalled);
 try {
 service.startAsync().awaitRunning();
 fail();
 } catch (RuntimeException e) {
 assertSame(exception, e.getCause());
 }
 assertEquals(1, service.startUpCalled);
 assertEquals(Service.State.FAILED, service.state());
 ASSERT.that(service.transitionStates).has().exactly(Service.State.STARTING).inOrder();
 }

public void testStop_withoutStart() {
 TestService service = new TestService();
 service.stopAsync().awaitTerminated();
 assertEquals(0, service.startUpCalled);
 assertEquals(0, service.shutDownCalled);
 assertEquals(Service.State.TERMINATED, service.state());
 ASSERT.that(service.transitionStates).isEmpty();
 }

public void testStop_afterStart() {
 TestService service = new TestService();
 service.startAsync().awaitRunning();
 assertEquals(1, service.startUpCalled);
 assertEquals(0, service.shutDownCalled);
 service.stopAsync().awaitTerminated();
 assertEquals(1, service.startUpCalled);
 assertEquals(1, service.shutDownCalled);
 assertEquals(Service.State.TERMINATED, service.state());
```

```
 ASSERT.that(service.transitionStates)
 .has().exactly(Service.State.STARTING, Service.State.STOPPING).inOrder();
 }

public void testStop_failed() {
 final Exception exception = new Exception("deliberate");
 TestService service = new TestService() {
 @Override protected void shutDown() throws Exception {
 super.shutDown();
 throw exception;
 }
 };
 service.startAsync().awaitRunning();
 assertEquals(1, service.startUpCalled);
 assertEquals(0, service.shutDownCalled);
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (RuntimeException e) {
 assertSame(exception, e.getCause());
 }
 assertEquals(1, service.startUpCalled);
 assertEquals(1, service.shutDownCalled);
 assertEquals(Service.State.FAILED, service.state());
 ASSERT.that(service.transitionStates)
 .has().exactly(Service.State.STARTING, Service.State.STOPPING).inOrder();
 }

public void testServiceToString() {
 AbstractIdleService service = new TestService();
 assertEquals("TestService [NEW]", service.toString());
 service.startAsync().awaitRunning();
 assertEquals("TestService [RUNNING]", service.toString());
 service.stopAsync().awaitTerminated();
 assertEquals("TestService [TERMINATED]", service.toString());
 }

public void testTimeout() throws Exception {
 // Create a service whose executor will never run its commands
 Service service = new TestService() {
 @Override protected Executor executor() {
 return new Executor() {
 @Override public void execute(Runnable command) {}
 };
 }
```

```
 };
 try {
 service.startAsync().awaitRunning(1, TimeUnit.MILLISECONDS);
 fail("Expected timeout");
 } catch (TimeoutException e) {
 ASSERT.that(e.getMessage()).contains(Service.State.STARTING.toString());
 }
 }

private static class TestService extends AbstractIdleService {
 int startUpCalled = 0;
 int shutDownCalled = 0;
 final List<State> transitionStates = Lists.newArrayList();

@Override protected void startUp() throws Exception {
 assertEquals(0, startUpCalled);
 assertEquals(0, shutDownCalled);
 startUpCalled++;
 assertEquals(State.STARTING, state());
 }

@Override protected void shutDown() throws Exception {
 assertEquals(1, startUpCalled);
 assertEquals(0, shutDownCalled);
 shutDownCalled++;
 assertEquals(State.STOPPING, state());
 }

@Override protected Executor executor() {
 transitionStates.add(state());
 return MoreExecutors.sameThreadExecutor();
 }
 }
}

<pre>
```

AbstractScheduledServiceTest

```
</pre>
/*
 * Copyright (C) 2011 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
```

```
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.google.common.util.concurrent;

import com.google.common.util.concurrent.AbstractScheduledService.Scheduler;
import com.google.common.util.concurrent.Service.State;

import junit.framework.TestCase;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Unit test for {@link AbstractScheduledService}.
 *
 * @author Luke Sandberg
 */

public class AbstractScheduledServiceTest extends TestCase {

volatile Scheduler configuration = Scheduler.newFixedDelaySchedule(0, 10, TimeUnit.MILLISECONDS);
 volatile ScheduledFuture<?> future = null;

volatile boolean atFixedRateCalled = false;
 volatile boolean withFixedDelayCalled = false;
 volatile boolean scheduleCalled = false;
```

```
final ScheduledExecutorService executor = new ScheduledThreadPoolExecutor(10) {
 @Override
 public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay,
 long delay, TimeUnit unit) {
 return future = super.scheduleWithFixedDelay(command, initialDelay, delay, unit);
 }
 };

public void testServiceStartStop() throws Exception {
 NullService service = new NullService();
 service.startAsync().awaitRunning();
 assertFalse(future.isDone());
 service.stopAsync().awaitTerminated();
 assertTrue(future.isCancelled());
 }

private class NullService extends AbstractScheduledService {
 @Override protected void runOneIteration() throws Exception {}
 @Override protected Scheduler scheduler() { return configuration; }
 @Override protected ScheduledExecutorService executor() { return executor; }
 }

public void testFailOnExceptionFromRun() throws Exception {
 TestService service = new TestService();
 service.runException = new Exception();
 service.startAsync().awaitRunning();
 service.runFirstBarrier.await();
 service.runSecondBarrier.await();
 try {
 future.get();
 fail();
 } catch (ExecutionException e) {
 // An execution exception holds a runtime exception (from throwables.propogate) that holds our
 // original exception.
 assertEquals(service.runException, e.getCause().getCause());
 }
 assertEquals(service.state(), Service.State.FAILED);
 }

public void testFailOnExceptionFromStartUp() {
 TestService service = new TestService();
 service.startUpException = new Exception();
 try {
 service.startAsync().awaitRunning();
 fail();
```

```
 } catch (IllegalStateException e) {
 assertEquals(service.startUpException, e.getCause());
 }
 assertEquals(0, service.numberOfTimesRunCalled.get());
 assertEquals(Service.State.FAILED, service.state());
 }

public void testFailOnExceptionFromShutDown() throws Exception {
 TestService service = new TestService();
 service.shutDownException = new Exception();
 service.startAsync().awaitRunning();
 service.runFirstBarrier.await();
 service.stopAsync();
 service.runSecondBarrier.await();
 try {
 service.awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(service.shutDownException, e.getCause());
 }
 assertEquals(Service.State.FAILED, service.state());
 }

public void testRunOneIterationCalledMultipleTimes() throws Exception {
 TestService service = new TestService();
 service.startAsync().awaitRunning();
 for (int i = 1; i < 10; i++) {
 service.runFirstBarrier.await();
 assertEquals(i, service.numberOfTimesRunCalled.get());
 service.runSecondBarrier.await();
 }
 service.runFirstBarrier.await();
 service.stopAsync();
 service.runSecondBarrier.await();
 service.stopAsync().awaitTerminated();
 }

public void testExecutorOnlyCalledOnce() throws Exception {
 TestService service = new TestService();
 service.startAsync().awaitRunning();
 // It should be called once during startup.
 assertEquals(1, service.numberOfTimesExecutorCalled.get());
 for (int i = 1; i < 10; i++) {
 service.runFirstBarrier.await();
 assertEquals(i, service.numberOfTimesRunCalled.get());
```

```
 service.runSecondBarrier.await();
 }
 service.runFirstBarrier.await();
 service.stopAsync();
 service.runSecondBarrier.await();
 service.stopAsync().awaitTerminated();
 // Only called once overall.
 assertEquals(1, service.numberOfTimesExecutorCalled.get());
 }

public void testDefaultExecutorIsShutdownWhenServiceIsStopped() throws Exception {
 final CountDownLatch terminationLatch = new CountDownLatch(1);
 AbstractScheduledService service = new AbstractScheduledService() {
 volatile ScheduledExecutorService executorService;
 @Override protected void runOneIteration() throws Exception {}

@Override protected ScheduledExecutorService executor() {
 if (executorService == null) {
 executorService = super.executor();
 // Add a listener that will be executed after the listener that shuts down the executor.
 addListener(new Listener() {
 @Override public void terminated(State from) {
 terminationLatch.countDown();
 }
 }, MoreExecutors.sameThreadExecutor());
 }
 return executorService;
 }

@Override protected Scheduler scheduler() {
 return Scheduler.newFixedDelaySchedule(0, 1, TimeUnit.MILLISECONDS);
 }};

service.startAsync();
 assertFalse(service.executor().isShutdown());
 service.awaitRunning();
 service.stopAsync();
 terminationLatch.await();
 assertTrue(service.executor().isShutdown());
 assertTrue(service.executor().awaitTermination(100, TimeUnit.MILLISECONDS));
 }

public void testDefaultExecutorIsShutdownWhenServiceFails() throws Exception {
 final CountDownLatch failureLatch = new CountDownLatch(1);
 AbstractScheduledService service = new AbstractScheduledService() {
```

```
 volatile ScheduledExecutorService executorService;
 @Override protected void runOneIteration() throws Exception {}

@Override protected void startUp() throws Exception {
 throw new Exception("Failed");
 }

@Override protected ScheduledExecutorService executor() {
 if (executorService == null) {
 executorService = super.executor();
 // Add a listener that will be executed after the listener that shuts down the executor.
 addListener(new Listener() {
 @Override public void failed(State from, Throwable failure) {
 failureLatch.countDown();
 }
 }, MoreExecutors.sameThreadExecutor());
 }
 return executorService;
 }

@Override protected Scheduler scheduler() {
 return Scheduler.newFixedDelaySchedule(0, 1, TimeUnit.MILLISECONDS);
 }};

try {
 service.startAsync().awaitRunning();
 fail("Expected service to fail during startup");
 } catch (IllegalStateException expected) {}
 failureLatch.await();
 assertTrue(service.executor().isShutdown());
 assertTrue(service.executor().awaitTermination(100, TimeUnit.MILLISECONDS));
 }

public void testSchedulerOnlyCalledOnce() throws Exception {
 TestService service = new TestService();
 service.startAsync().awaitRunning();
 // It should be called once during startup.
 assertEquals(1, service.numberOfTimesSchedulerCalled.get());
 for (int i = 1; i < 10; i++) {
 service.runFirstBarrier.await();
 assertEquals(i, service.numberOfTimesRunCalled.get());
 service.runSecondBarrier.await();
 }
 service.runFirstBarrier.await();
 service.stopAsync();
```

```
 service.runSecondBarrier.await();
 service.awaitTerminated();
 // Only called once overall.
 assertEquals(1, service.numberOfTimesSchedulerCalled.get());
 }

private class TestService extends AbstractScheduledService {
 CyclicBarrier runFirstBarrier = new CyclicBarrier(2);
 CyclicBarrier runSecondBarrier = new CyclicBarrier(2);

volatile boolean startUpCalled = false;
 volatile boolean shutDownCalled = false;
 AtomicInteger numberOfTimesRunCalled = new AtomicInteger(0);
 AtomicInteger numberOfTimesExecutorCalled = new AtomicInteger(0);
 AtomicInteger numberOfTimesSchedulerCalled = new AtomicInteger(0);
 volatile Exception runException = null;
 volatile Exception startUpException = null;
 volatile Exception shutDownException = null;

@Override
 protected void runOneIteration() throws Exception {
 assertTrue(startUpCalled);
 assertFalse(shutDownCalled);
 numberOfTimesRunCalled.incrementAndGet();
 assertEquals(State.RUNNING, state());
 runFirstBarrier.await();
 runSecondBarrier.await();
 if (runException != null) {
 throw runException;
 }
 }

@Override
 protected void startUp() throws Exception {
 assertFalse(startUpCalled);
 assertFalse(shutDownCalled);
 startUpCalled = true;
 assertEquals(State.STARTING, state());
 if (startUpException != null) {
 throw startUpException;
 }
 }

@Override
 protected void shutDown() throws Exception {
```

```
 assertTrue(startUpCalled);
 assertFalse(shutDownCalled);
 shutDownCalled = true;
 if (shutDownException != null) {
 throw shutDownException;
 }
 }

@Override
 protected ScheduledExecutorService executor() {
 numberOfTimesExecutorCalled.incrementAndGet();
 return executor;
 }

@Override
 protected Scheduler scheduler() {
 numberOfTimesSchedulerCalled.incrementAndGet();
 return configuration;
 }
 }

public static class SchedulerTest extends TestCase {
 // These constants are arbitrary and just used to make sure that the correct method is called
 // with the correct parameters.
 private static final int initialDelay = 10;
 private static final int delay = 20;
 private static final TimeUnit unit = TimeUnit.MILLISECONDS;

// Unique runnable object used for comparison.
 final Runnable testRunnable = new Runnable() {@Override public void run() {}};
 boolean called = false;

private void assertSingleCallWithCorrectParameters(Runnable command, long initialDelay,
 long delay, TimeUnit unit) {
 assertFalse(called); // only called once.
 called = true;
 assertEquals(SchedulerTest.initialDelay, initialDelay);
 assertEquals(SchedulerTest.delay, delay);
 assertEquals(SchedulerTest.unit, unit);
 assertEquals(testRunnable, command);
 }

public void testFixedRateSchedule() {
 Scheduler schedule = Scheduler.newFixedRateSchedule(initialDelay, delay, unit);
 schedule.schedule(null, new ScheduledThreadPoolExecutor(1) {
```

```
 @Override
 public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay,
 long period, TimeUnit unit) {
 assertSingleCallWithCorrectParameters(command, initialDelay, delay, unit);
 return null;
 }
 }, testRunnable);
 assertTrue(called);
 }

public void testFixedDelaySchedule() {
 Scheduler schedule = Scheduler.newFixedDelaySchedule(initialDelay, delay, unit);
 schedule.schedule(null, new ScheduledThreadPoolExecutor(10) {
 @Override
 public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay,
 long delay, TimeUnit unit) {
 assertSingleCallWithCorrectParameters(command, initialDelay, delay, unit);
 return null;
 }
 }, testRunnable);
 assertTrue(called);
 }

private class TestCustomScheduler extends AbstractScheduledService.CustomScheduler {
 public AtomicInteger scheduleCounter = new AtomicInteger(0);
 @Override
 protected Schedule getNextSchedule() throws Exception {
 scheduleCounter.incrementAndGet();
 return new Schedule(0, TimeUnit.SECONDS);
 }
 }

public void testCustomSchedule_startStop() throws Exception {
 final CyclicBarrier firstBarrier = new CyclicBarrier(2);
 final CyclicBarrier secondBarrier = new CyclicBarrier(2);
 final AtomicBoolean shouldWait = new AtomicBoolean(true);
 Runnable task = new Runnable() {
 @Override public void run() {
 try {
 if (shouldWait.get()) {
 firstBarrier.await();
 secondBarrier.await();
 }
 } catch (Exception e) {
 throw new RuntimeException(e);
```

```
 }
 }
 };
 TestCustomScheduler scheduler = new TestCustomScheduler();
 Future<?> future = scheduler.schedule(null, Executors.newScheduledThreadPool(10), task);
 firstBarrier.await();
 assertEquals(1, scheduler.scheduleCounter.get());
 secondBarrier.await();
 firstBarrier.await();
 assertEquals(2, scheduler.scheduleCounter.get());
 shouldWait.set(false);
 secondBarrier.await();
 future.cancel(false);
 }

public void testCustomSchedulerServiceStop() throws Exception {
 TestAbstractScheduledCustomService service = new TestAbstractScheduledCustomService();
 service.startAsync().awaitRunning();
 service.firstBarrier.await();
 assertEquals(1, service.numIterations.get());
 service.stopAsync();
 service.secondBarrier.await();
 service.awaitTerminated();
 // Sleep for a while just to ensure that our task wasn't called again.
 Thread.sleep(unit.toMillis(3 * delay));
 assertEquals(1, service.numIterations.get());
 }

public void testBig() throws Exception {
 TestAbstractScheduledCustomService service = new TestAbstractScheduledCustomService() {
 @Override protected Scheduler scheduler() {
 return new AbstractScheduledService.CustomScheduler() {
 @Override
 protected Schedule getNextSchedule() throws Exception {
 // Explicitly yield to increase the probability of a pathological scheduling.
 Thread.yield();
 return new Schedule(0, TimeUnit.SECONDS);
 }
 };
 }
 };
 service.useBarriers = false;
 service.startAsync().awaitRunning();
 Thread.sleep(50);
 service.useBarriers = true;
```

```
 service.firstBarrier.await();
 int numIterations = service.numIterations.get();
 service.stopAsync();
 service.secondBarrier.await();
 service.awaitTerminated();
 assertEquals(numIterations, service.numIterations.get());
 }

private static class TestAbstractScheduledCustomService extends AbstractScheduledService {
 final AtomicInteger numIterations = new AtomicInteger(0);
 volatile boolean useBarriers = true;
 final CyclicBarrier firstBarrier = new CyclicBarrier(2);
 final CyclicBarrier secondBarrier = new CyclicBarrier(2);

@Override protected void runOneIteration() throws Exception {
 numIterations.incrementAndGet();
 if (useBarriers) {
 firstBarrier.await();
 secondBarrier.await();
 }
 }

@Override protected ScheduledExecutorService executor() {
 // use a bunch of threads so that weird overlapping schedules are more likely to happen.
 return Executors.newScheduledThreadPool(10);
 }

@Override protected void startUp() throws Exception {}

@Override protected void shutDown() throws Exception {}

@Override protected Scheduler scheduler() {
 return new CustomScheduler() {
 @Override
 protected Schedule getNextSchedule() throws Exception {
 return new Schedule(delay, unit);
 }};
 }
 }

public void testCustomSchedulerFailure() throws Exception {
 TestFailingCustomScheduledService service = new TestFailingCustomScheduledService();
 service.startAsync().awaitRunning();
 for (int i = 1; i < 4; i++) {
 service.firstBarrier.await();
```

```
 assertEquals(i, service.numIterations.get());
 service.secondBarrier.await();
 }
 Thread.sleep(1000);
 try {
 service.stopAsync().awaitTerminated(100, TimeUnit.SECONDS);
 fail();
 } catch (IllegalStateException e) {
 assertEquals(State.FAILED, service.state());
 }
 }

private static class TestFailingCustomScheduledService extends AbstractScheduledService {
 final AtomicInteger numIterations = new AtomicInteger(0);
 final CyclicBarrier firstBarrier = new CyclicBarrier(2);
 final CyclicBarrier secondBarrier = new CyclicBarrier(2);

@Override protected void runOneIteration() throws Exception {
 numIterations.incrementAndGet();
 firstBarrier.await();
 secondBarrier.await();
 }

@Override protected ScheduledExecutorService executor() {
 // use a bunch of threads so that weird overlapping schedules are more likely to happen.
 return Executors.newScheduledThreadPool(10);
 }

@Override protected Scheduler scheduler() {
 return new CustomScheduler() {
 @Override
 protected Schedule getNextSchedule() throws Exception {
 if (numIterations.get() > 2) {
 throw new IllegalStateException("Failed");
 }
 return new Schedule(delay, unit);
 }};
 }
 }
 }
}
<pre>
```

AbstractServiceTest

```
</pre>
/*
 * Copyright (C) 2009 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.google.common.util.concurrent;

import static java.lang.Thread.currentThread;
import static java.util.concurrent.TimeUnit.SECONDS;

import com.google.common.collect.ImmutableList;
import com.google.common.collect.Iterables;
import com.google.common.collect.Lists;
import com.google.common.util.concurrent.Service.Listener;
import com.google.common.util.concurrent.Service.State;

import junit.framework.TestCase;

import java.lang.Thread.UncaughtExceptionHandler;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicReference;

import javax.annotation.concurrent.GuardedBy;

/**
 * Unit test for {@link AbstractService}.
 *
 * @author Jesse Wilson
 */
```

```
public class AbstractServiceTest extends TestCase {

private Thread executionThread;
 private Throwable thrownByExecutionThread;

public void testNoOpServiceStartStop() throws Exception {
 NoOpService service = new NoOpService();
 RecordingListener listener = RecordingListener.record(service);

assertEquals(State.NEW, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.running);

service.startAsync();
 assertEquals(State.RUNNING, service.state());
 assertTrue(service.isRunning());
 assertTrue(service.running);

service.stopAsync();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.running);
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());
 }

public void testNoOpServiceStartAndWaitStopAndWait() throws Exception {
 NoOpService service = new NoOpService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());
 }

public void testNoOpServiceStartAsyncAndAwaitStopAsyncAndAwait() throws Exception {
 NoOpService service = new NoOpService();

service.startAsync().awaitRunning();
```

```
 assertEquals(State.RUNNING, service.state());

service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());
 }

public void testNoOpServiceStopIdempotence() throws Exception {
 NoOpService service = new NoOpService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.stopAsync();
 service.stopAsync();
 assertEquals(State.TERMINATED, service.state());
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());
 }

public void testNoOpServiceStopIdempotenceAfterWait() throws Exception {
 NoOpService service = new NoOpService();

service.startAsync().awaitRunning();

service.stopAsync().awaitTerminated();
 service.stopAsync();
 assertEquals(State.TERMINATED, service.state());
 }

public void testNoOpServiceStopIdempotenceDoubleWait() throws Exception {
 NoOpService service = new NoOpService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.stopAsync().awaitTerminated();
 service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());
 }
```

```
public void testNoOpServiceStartStopAndWaitUninterruptible()
 throws Exception {
 NoOpService service = new NoOpService();

currentThread().interrupt();
 try {
 service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());

assertTrue(currentThread().isInterrupted());
 } finally {
 Thread.interrupted(); // clear interrupt for future tests
 }
 }

private static class NoOpService extends AbstractService {
 boolean running = false;

@Override protected void doStart() {
 assertFalse(running);
 running = true;
 notifyStarted();
 }

@Override protected void doStop() {
 assertTrue(running);
 running = false;
 notifyStopped();
 }
 }

public void testManualServiceStartStop() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync();
 assertEquals(State.STARTING, service.state());
 assertFalse(service.isRunning());
 assertTrue(service.doStartCalled);

service.notifyStarted(); // usually this would be invoked by another thread
 assertEquals(State.RUNNING, service.state());
```

```
 assertTrue(service.isRunning());

service.stopAsync();
 assertEquals(State.STOPPING, service.state());
 assertFalse(service.isRunning());
 assertTrue(service.doStopCalled);

service.notifyStopped(); // usually this would be invoked by another thread
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());

}

public void testManualServiceNotifyStoppedWhileRunning() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync();
 service.notifyStarted();
 service.notifyStopped();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.doStopCalled);

assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.TERMINATED),
 listener.getStateHistory());
 }

public void testManualServiceStopWhileStarting() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync();
 assertEquals(State.STARTING, service.state());
```

```
 assertFalse(service.isRunning());
 assertTrue(service.doStartCalled);

service.stopAsync();
 assertEquals(State.STOPPING, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.doStopCalled);

service.notifyStarted();
 assertEquals(State.STOPPING, service.state());
 assertFalse(service.isRunning());
 assertTrue(service.doStopCalled);

service.notifyStopped();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());
 }

/**
 * This tests for a bug where if {@link Service#stopAsync()} was called while the service was
 * {@link State#STARTING} more than once, the {@link Listener#stopping(State)} callback would get
 * called multiple times.
 */
 public void testManualServiceStopMultipleTimesWhileStarting() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 final AtomicInteger stopppingCount = new AtomicInteger();
 service.addListener(new Listener() {
 @Override public void stopping(State from) {
 stopppingCount.incrementAndGet();
 }
 }, MoreExecutors.sameThreadExecutor());

service.startAsync();
 service.stopAsync();
 assertEquals(1, stopppingCount.get());
 service.stopAsync();
 assertEquals(1, stopppingCount.get());
 }
```

```
public void testManualServiceStopWhileNew() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);

service.stopAsync();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.doStartCalled);
 assertFalse(service.doStopCalled);
 assertEquals(ImmutableList.of(State.TERMINATED), listener.getStateHistory());
 }

public void testManualServiceFailWhileStarting() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync();
 service.notifyFailed(EXCEPTION);
 assertEquals(ImmutableList.of(State.STARTING, State.FAILED), listener.getStateHistory());
 }

public void testManualServiceFailWhileRunning() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync();
 service.notifyStarted();
 service.notifyFailed(EXCEPTION);
 assertEquals(ImmutableList.of(State.STARTING, State.RUNNING, State.FAILED),
 listener.getStateHistory());
 }

public void testManualServiceFailWhileStopping() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync();
 service.notifyStarted();
 service.stopAsync();
 service.notifyFailed(EXCEPTION);
 assertEquals(ImmutableList.of(State.STARTING, State.RUNNING, State.STOPPING, State.FAILED),
 listener.getStateHistory());
 }

public void testManualServiceUnrequestedStop() {
 ManualSwitchedService service = new ManualSwitchedService();

service.startAsync();
```

```
service.notifyStarted();
 assertEquals(State.RUNNING, service.state());
 assertTrue(service.isRunning());
 assertFalse(service.doStopCalled);

service.notifyStopped();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.doStopCalled);
 }

/**
 * The user of this service should call {@link #notifyStarted} and {@link
 * #notifyStopped} after calling {@link #startAsync} and {@link #stopAsync}.
 */
 private static class ManualSwitchedService extends AbstractService {
 boolean doStartCalled = false;
 boolean doStopCalled = false;

@Override protected void doStart() {
 assertFalse(doStartCalled);
 doStartCalled = true;
 }

@Override protected void doStop() {
 assertFalse(doStopCalled);
 doStopCalled = true;
 }
 }

public void testAwaitTerminated() throws Exception {
 final NoOpService service = new NoOpService();
 Thread waiter = new Thread() {
 @Override public void run() {
 service.awaitTerminated();
 }
 };
 waiter.start();
 service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());
 service.stopAsync();
 waiter.join(100); // ensure that the await in the other thread is triggered
 assertFalse(waiter.isAlive());
 }
```

```
public void testAwaitTerminated_FailedService() throws Exception {
 final ManualSwitchedService service = new ManualSwitchedService();
 final AtomicReference<Throwable> exception = Atomics.newReference();
 Thread waiter = new Thread() {
 @Override public void run() {
 try {
 service.awaitTerminated();
 fail("Expected an IllegalStateException");
 } catch (Throwable t) {
 exception.set(t);
 }
 }
 };
 waiter.start();
 service.startAsync();
 service.notifyStarted();
 assertEquals(State.RUNNING, service.state());
 service.notifyFailed(EXCEPTION);
 assertEquals(State.FAILED, service.state());
 waiter.join(100);
 assertFalse(waiter.isAlive());
 assertTrue(exception.get() instanceof IllegalStateException);
 assertEquals(EXCEPTION, exception.get().getCause());
 }

public void testThreadedServiceStartAndWaitStopAndWait() throws Throwable {
 ThreadedService service = new ThreadedService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.awaitRunChecks();

service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());

throwIfSet(thrownByExecutionThread);
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());
```

```
 }

public void testThreadedServiceStopIdempotence() throws Throwable {
 ThreadedService service = new ThreadedService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.awaitRunChecks();

service.stopAsync();
 service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());

throwIfSet(thrownByExecutionThread);
 }

public void testThreadedServiceStopIdempotenceAfterWait()
 throws Throwable {
 ThreadedService service = new ThreadedService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.awaitRunChecks();

service.stopAsync().awaitTerminated();
 service.stopAsync();
 assertEquals(State.TERMINATED, service.state());

executionThread.join();

throwIfSet(thrownByExecutionThread);
 }

public void testThreadedServiceStopIdempotenceDoubleWait()
 throws Throwable {
 ThreadedService service = new ThreadedService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.awaitRunChecks();

service.stopAsync().awaitTerminated();
```

```
 service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());

throwIfSet(thrownByExecutionThread);
 }

public void testManualServiceFailureIdempotence() {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener.record(service);
 service.startAsync();
 service.notifyFailed(new Exception("1"));
 service.notifyFailed(new Exception("2"));
 assertEquals("1", service.failureCause().getMessage());
 try {
 service.awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals("1", e.getCause().getMessage());
 }
 }

private class ThreadedService extends AbstractService {
 final CountDownLatch hasConfirmedIsRunning = new CountDownLatch(1);

/*
 * The main test thread tries to stop() the service shortly after
 * confirming that it is running. Meanwhile, the service itself is trying
 * to confirm that it is running. If the main thread's stop() call happens
 * before it has the chance, the test will fail. To avoid this, the main
 * thread calls this method, which waits until the service has performed
 * its own "running" check.
 */
 void awaitRunChecks() throws InterruptedException {
 assertTrue("Service thread hasn't finished its checks. "
 + "Exception status (possibly stale): " + thrownByExecutionThread,
 hasConfirmedIsRunning.await(10, SECONDS));
 }

@Override protected void doStart() {
 assertEquals(State.STARTING, state());
 invokeOnExecutionThreadForTest(new Runnable() {
 @Override public void run() {
 assertEquals(State.STARTING, state());
 notifyStarted();
 assertEquals(State.RUNNING, state());
```

```
 hasConfirmedIsRunning.countDown();
 }
 });
 }

@Override protected void doStop() {
 assertEquals(State.STOPPING, state());
 invokeOnExecutionThreadForTest(new Runnable() {
 @Override public void run() {
 assertEquals(State.STOPPING, state());
 notifyStopped();
 assertEquals(State.TERMINATED, state());
 }
 });
 }
 }

private void invokeOnExecutionThreadForTest(Runnable runnable) {
 executionThread = new Thread(runnable);
 executionThread.setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
 @Override
 public void uncaughtException(Thread thread, Throwable e) {
 thrownByExecutionThread = e;
 }
 });
 executionThread.start();
 }

private static void throwIfSet(Throwable t) throws Throwable {
 if (t != null) {
 throw t;
 }
 }

public void testStopUnstartedService() throws Exception {
 NoOpService service = new NoOpService();
 RecordingListener listener = RecordingListener.record(service);

service.stopAsync();
 assertEquals(State.TERMINATED, service.state());

try {
 service.startAsync();
 fail();
 } catch (IllegalStateException expected) {}
```

```
 assertEquals(State.TERMINATED, Iterables.getOnlyElement(listener.getStateHistory()));
 }

public void testFailingServiceStartAndWait() throws Exception {
 StartFailingService service = new StartFailingService();
 RecordingListener listener = RecordingListener.record(service);

try {
 service.startAsync().awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(EXCEPTION, service.failureCause());
 assertEquals(EXCEPTION, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testFailingServiceStopAndWait_stopFailing() throws Exception {
 StopFailingService service = new StopFailingService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync().awaitRunning();
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(EXCEPTION, service.failureCause());
 assertEquals(EXCEPTION, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testFailingServiceStopAndWait_runFailing() throws Exception {
 RunFailingService service = new RunFailingService();
 RecordingListener listener = RecordingListener.record(service);
```

```
service.startAsync();
 try {
 service.awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(EXCEPTION, service.failureCause());
 assertEquals(EXCEPTION, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testThrowingServiceStartAndWait() throws Exception {
 StartThrowingService service = new StartThrowingService();
 RecordingListener listener = RecordingListener.record(service);

try {
 service.startAsync().awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(service.exception, service.failureCause());
 assertEquals(service.exception, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testThrowingServiceStopAndWait_stopThrowing() throws Exception {
 StopThrowingService service = new StopThrowingService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync().awaitRunning();
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(service.exception, service.failureCause());
```

```
 assertEquals(service.exception, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testThrowingServiceStopAndWait_runThrowing() throws Exception {
 RunThrowingService service = new RunThrowingService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync();
 try {
 service.awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(service.exception, service.failureCause());
 assertEquals(service.exception, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testFailureCause_throwsIfNotFailed() {
 StopFailingService service = new StopFailingService();
 try {
 service.failureCause();
 fail();
 } catch (IllegalStateException e) {
 // expected
 }
 service.startAsync().awaitRunning();
 try {
 service.failureCause();
 fail();
 } catch (IllegalStateException e) {
 // expected
```

```
 }
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(EXCEPTION, service.failureCause());
 assertEquals(EXCEPTION, e.getCause());
 }
 }

public void testAddListenerAfterFailureDoesntCauseDeadlock() throws InterruptedException {
 final StartFailingService service = new StartFailingService();
 service.startAsync();
 assertEquals(State.FAILED, service.state());
 service.addListener(new RecordingListener(service), MoreExecutors.sameThreadExecutor());
 Thread thread = new Thread() {
 @Override public void run() {
 // Internally stopAsync() grabs a lock, this could be any such method on AbstractService.
 service.stopAsync();
 }
 };
 thread.start();
 thread.join(100);
 assertFalse(thread + " is deadlocked", thread.isAlive());
 }

public void testListenerDoesntDeadlockOnStartAndWaitFromRunning() throws Exception {
 final NoOpThreadedService service = new NoOpThreadedService();
 service.addListener(new Listener() {
 @Override public void running() {
 service.awaitRunning();
 }
 }, MoreExecutors.sameThreadExecutor());
 service.startAsync().awaitRunning(10, TimeUnit.MILLISECONDS);
 service.stopAsync();
 }

public void testListenerDoesntDeadlockOnStopAndWaitFromTerminated() throws Exception {
 final NoOpThreadedService service = new NoOpThreadedService();
 service.addListener(new Listener() {
 @Override public void terminated(State from) {
 service.stopAsync().awaitTerminated();
 }
 }, MoreExecutors.sameThreadExecutor());
 service.startAsync().awaitRunning();
```

```
Thread thread = new Thread() {
 @Override public void run() {
 service.stopAsync().awaitTerminated();
 }
 };
 thread.start();
 thread.join(100);
 assertFalse(thread + " is deadlocked", thread.isAlive());
 }

private static class NoOpThreadedService extends AbstractExecutionThreadService {
 final CountDownLatch latch = new CountDownLatch(1);
 @Override protected void run() throws Exception {
 latch.await();
 }
 @Override protected void triggerShutdown() {
 latch.countDown();
 }
 }

private static class StartFailingService extends AbstractService {
 @Override protected void doStart() {
 notifyFailed(EXCEPTION);
 }

@Override protected void doStop() {
 fail();
 }
 }

private static class RunFailingService extends AbstractService {
 @Override protected void doStart() {
 notifyStarted();
 notifyFailed(EXCEPTION);
 }

@Override protected void doStop() {
 fail();
 }
 }

private static class StopFailingService extends AbstractService {
 @Override protected void doStart() {
 notifyStarted();
```

```
 }

@Override protected void doStop() {
 notifyFailed(EXCEPTION);
 }
 }

private static class StartThrowingService extends AbstractService {

final RuntimeException exception = new RuntimeException("deliberate");

@Override protected void doStart() {
 throw exception;
 }

@Override protected void doStop() {
 fail();
 }
 }

private static class RunThrowingService extends AbstractService {

final RuntimeException exception = new RuntimeException("deliberate");

@Override protected void doStart() {
 notifyStarted();
 throw exception;
 }

@Override protected void doStop() {
 fail();
 }
 }

private static class StopThrowingService extends AbstractService {

final RuntimeException exception = new RuntimeException("deliberate");

@Override protected void doStart() {
 notifyStarted();
 }

@Override protected void doStop() {
 throw exception;
 }
```

```
 }

private static class RecordingListener extends Listener {
 static RecordingListener record(Service service) {
 RecordingListener listener = new RecordingListener(service);
 service.addListener(listener, MoreExecutors.sameThreadExecutor());
 return listener;
 }

final Service service;

RecordingListener(Service service) {
 this.service = service;
 }

@GuardedBy("this")
 final List<State> stateHistory = Lists.newArrayList();
 final CountDownLatch completionLatch = new CountDownLatch(1);

ImmutableList<State> getStateHistory() throws Exception {
 completionLatch.await();
 synchronized (this) {
 return ImmutableList.copyOf(stateHistory);
 }
 }

@Override public synchronized void starting() {
 assertTrue(stateHistory.isEmpty());
 assertNotSame(State.NEW, service.state());
 stateHistory.add(State.STARTING);
 }

@Override public synchronized void running() {
 assertEquals(State.STARTING, Iterables.getOnlyElement(stateHistory));
 stateHistory.add(State.RUNNING);
 service.awaitRunning();
 assertNotSame(State.STARTING, service.state());
 }

@Override public synchronized void stopping(State from) {
 assertEquals(from, Iterables.getLast(stateHistory));
 stateHistory.add(State.STOPPING);
 if (from == State.STARTING) {
 try {
 service.awaitRunning();
```

```
 fail();
 } catch (IllegalStateException expected) {
 assertNull(expected.getCause());
 assertTrue(expected.getMessage().equals(
 "Expected the service to be RUNNING, but was STOPPING"));
 }
 }
 assertNotSame(from, service.state());
 }

@Override public synchronized void terminated(State from) {
 assertEquals(from, Iterables.getLast(stateHistory, State.NEW));
 stateHistory.add(State.TERMINATED);
 assertEquals(State.TERMINATED, service.state());
 if (from == State.NEW) {
 try {
 service.awaitRunning();
 fail();
 } catch (IllegalStateException expected) {
 assertNull(expected.getCause());
 assertTrue(expected.getMessage().equals(
 "Expected the service to be RUNNING, but was TERMINATED"));
 }
 }
 completionLatch.countDown();
 }

@Override public synchronized void failed(State from, Throwable failure) {
 assertEquals(from, Iterables.getLast(stateHistory));
 stateHistory.add(State.FAILED);
 assertEquals(State.FAILED, service.state());
 assertEquals(failure, service.failureCause());
 if (from == State.STARTING) {
 try {
 service.awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(failure, e.getCause());
 }
 }
 try {
 service.awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(failure, e.getCause());
```

```
 }
 completionLatch.countDown();
 }
 }

public void testNotifyStartedWhenNotStarting() {
 AbstractService service = new DefaultService();
 try {
 service.notifyStarted();
 fail();
 } catch (IllegalStateException expected) {}
 }

public void testNotifyStoppedWhenNotRunning() {
 AbstractService service = new DefaultService();
 try {
 service.notifyStopped();
 fail();
 } catch (IllegalStateException expected) {}
 }

public void testNotifyFailedWhenNotStarted() {
 AbstractService service = new DefaultService();
 try {
 service.notifyFailed(new Exception());
 fail();
 } catch (IllegalStateException expected) {}
 }

public void testNotifyFailedWhenTerminated() {
 NoOpService service = new NoOpService();
 service.startAsync().awaitRunning();
 service.stopAsync().awaitTerminated();
 try {
 service.notifyFailed(new Exception());
 fail();
 } catch (IllegalStateException expected) {}
 }

private static class DefaultService extends AbstractService {
 @Override protected void doStart() {}
 @Override protected void doStop() {}
 }

private static final Exception EXCEPTION = new Exception();
```

```
}
<pre>
```

ServiceManagerTest

```
</pre>
/*
 * Copyright (C) 2012 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package com.google.common.util.concurrent;

import static java.util.Arrays.asList;

import com.google.common.collect.ImmutableMap;
import com.google.common.collect.ImmutableSet;
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
import com.google.common.testing.NullPointerTester;
import com.google.common.testing.TestLogHandler;
import com.google.common.util.concurrent.ServiceManager.Listener;

import junit.framework.TestCase;

import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.Set;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.Executor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import java.util.logging.Formatter;
import java.util.logging.Level;
```

```
import java.util.logging.LogRecord;
import java.util.logging.Logger;

/**
 * Tests for {@link ServiceManager}.
 *
 * @author Luke Sandberg
 * @author Chris Nokleberg
 */
public class ServiceManagerTest extends TestCase {

private static class NoOpService extends AbstractService {
 @Override protected void doStart() {
 notifyStarted();
 }

@Override protected void doStop() {
 notifyStopped();
 }
 }

/*
 * A NoOp service that will delay the startup and shutdown notification for a configurable amount
 * of time.
 */
 private static class NoOpDelayedSerivce extends NoOpService {
 private long delay;

public NoOpDelayedSerivce(long delay) {
 this.delay = delay;
 }

@Override protected void doStart() {
 new Thread() {
 @Override public void run() {
 Uninterruptibles.sleepUninterruptibly(delay, TimeUnit.MILLISECONDS);
 notifyStarted();
 }
 }.start();
 }

@Override protected void doStop() {
 new Thread() {
 @Override public void run() {
 Uninterruptibles.sleepUninterruptibly(delay, TimeUnit.MILLISECONDS);
```

```
 notifyStopped();
 }
 }.start();
 }
 }

private static class FailStartService extends NoOpService {
 @Override protected void doStart() {
 notifyFailed(new IllegalStateException("failed"));
 }
 }

private static class FailRunService extends NoOpService {
 @Override protected void doStart() {
 super.doStart();
 notifyFailed(new IllegalStateException("failed"));
 }
 }

private static class FailStopService extends NoOpService {
 @Override protected void doStop() {
 notifyFailed(new IllegalStateException("failed"));
 }
 }

public void testServiceStartupTimes() {
 Service a = new NoOpDelayedSerivce(150);
 Service b = new NoOpDelayedSerivce(353);
 ServiceManager serviceManager = new ServiceManager(asList(a, b));
 serviceManager.startAsync().awaitHealthy();
 ImmutableMap<Service, Long> startupTimes = serviceManager.startupTimes();
 assertEquals(2, startupTimes.size());
 assertTrue(startupTimes.get(a) >= 150);
 assertTrue(startupTimes.get(b) >= 353);
 }

public void testServiceStartStop() {
 Service a = new NoOpService();
 Service b = new NoOpService();
 ServiceManager manager = new ServiceManager(asList(a, b));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
 assertState(manager, Service.State.NEW, a, b);
 assertFalse(manager.isHealthy());
 manager.startAsync().awaitHealthy();
```

```
 assertState(manager, Service.State.RUNNING, a, b);
 assertTrue(manager.isHealthy());
 assertTrue(listener.healthyCalled);
 assertFalse(listener.stoppedCalled);
 assertTrue(listener.failedServices.isEmpty());
 manager.stopAsync().awaitStopped();
 assertState(manager, Service.State.TERMINATED, a, b);
 assertFalse(manager.isHealthy());
 assertTrue(listener.stoppedCalled);
 assertTrue(listener.failedServices.isEmpty());
 }

public void testFailStart() throws Exception {
 Service a = new NoOpService();
 Service b = new FailStartService();
 Service c = new NoOpService();
 Service d = new FailStartService();
 Service e = new NoOpService();
 ServiceManager manager = new ServiceManager(asList(a, b, c, d, e));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
 assertState(manager, Service.State.NEW, a, b, c, d, e);
 try {
 manager.startAsync().awaitHealthy();
 fail();
 } catch (IllegalStateException expected) {
 }
 assertFalse(listener.healthyCalled);
 assertState(manager, Service.State.RUNNING, a, c, e);
 assertEquals(ImmutableSet.of(b, d), listener.failedServices);
 assertState(manager, Service.State.FAILED, b, d);
 assertFalse(manager.isHealthy());

manager.stopAsync().awaitStopped();
 assertFalse(manager.isHealthy());
 assertFalse(listener.healthyCalled);
 assertTrue(listener.stoppedCalled);
 }

public void testFailRun() throws Exception {
 Service a = new NoOpService();
 Service b = new FailRunService();
 ServiceManager manager = new ServiceManager(asList(a, b));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
```

```
 assertState(manager, Service.State.NEW, a, b);
 try {
 manager.startAsync().awaitHealthy();
 fail();
 } catch (IllegalStateException expected) {
 }
 assertTrue(listener.healthyCalled);
 assertEquals(ImmutableSet.of(b), listener.failedServices);

manager.stopAsync().awaitStopped();
 assertState(manager, Service.State.FAILED, b);
 assertState(manager, Service.State.TERMINATED, a);

assertTrue(listener.stoppedCalled);
 }

public void testFailStop() throws Exception {
 Service a = new NoOpService();
 Service b = new FailStopService();
 Service c = new NoOpService();
 ServiceManager manager = new ServiceManager(asList(a, b, c));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);

manager.startAsync().awaitHealthy();
 assertTrue(listener.healthyCalled);
 assertFalse(listener.stoppedCalled);
 manager.stopAsync().awaitStopped();

assertTrue(listener.stoppedCalled);
 assertEquals(ImmutableSet.of(b), listener.failedServices);
 assertState(manager, Service.State.FAILED, b);
 assertState(manager, Service.State.TERMINATED, a, c);
 }

public void testToString() throws Exception {
 Service a = new NoOpService();
 Service b = new FailStartService();
 ServiceManager manager = new ServiceManager(asList(a, b));
 String toString = manager.toString();
 assertTrue(toString.contains("NoOpService"));
 assertTrue(toString.contains("FailStartService"));
 }

public void testTimeouts() throws Exception {
```

```
 Service a = new NoOpDelayedSerivce(50);
 ServiceManager manager = new ServiceManager(asList(a));
 manager.startAsync();
 try {
 manager.awaitHealthy(1, TimeUnit.MILLISECONDS);
 fail();
 } catch (TimeoutException expected) {
 }
 manager.awaitHealthy(100, TimeUnit.MILLISECONDS); // no exception thrown

manager.stopAsync();
 try {
 manager.awaitStopped(1, TimeUnit.MILLISECONDS);
 fail();
 } catch (TimeoutException expected) {
 }
 manager.awaitStopped(100, TimeUnit.MILLISECONDS); // no exception thrown
 }

/**
 * This covers a case where if the last service to stop failed then the stopped callback would
 * never be called.
 */
 public void testSingleFailedServiceCallsStopped() {
 Service a = new FailStartService();
 ServiceManager manager = new ServiceManager(asList(a));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
 try {
 manager.startAsync().awaitHealthy();
 fail();
 } catch (IllegalStateException expected) {
 }
 assertTrue(listener.stoppedCalled);
 }

/**
 * This covers a bug where listener.healthy would get called when a single service failed during
 * startup (it occurred in more complicated cases also).
 */
 public void testFailStart_singleServiceCallsHealthy() {
 Service a = new FailStartService();
 ServiceManager manager = new ServiceManager(asList(a));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
```

```
try {
manager.startAsync().awaitHealthy();
fail();
} catch (IllegalStateException expected) {
}
assertFalse(listener.healthyCalled);
}

/**
* This covers a bug where if a listener was installed that would stop the manager if any service
* fails and something failed during startup before service.start was called on all the services,
* then awaitStopped would deadlock due to an IllegalStateException that was thrown when trying to
* stop the timer(!).
*/
public void testFailStart_stopOthers() throws TimeoutException {
Service a = new FailStartService();
Service b = new NoOpService();
final ServiceManager manager = new ServiceManager(asList(a, b));
manager.addListener(new Listener() {
@Override public void failure(Service service) {
manager.stopAsync();
}});
manager.startAsync();
manager.awaitStopped(10, TimeUnit.MILLISECONDS);
}

private static void assertState(
ServiceManager manager, Service.State state, Service... services) {
Collection<Service> managerServices = manager.servicesByState().get(state);
for (Service service : services) {
assertEquals(service.toString(), state, service.state());
assertEquals(service.toString(), service.isRunning(), state == Service.State.RUNNING);
assertTrue(managerServices + " should contain " + service, managerServices.contains(service));
}
}

/**
* This is for covering a case where the ServiceManager would behave strangely if constructed
* with no service under management. Listeners would never fire because the ServiceManager was
* healthy and stopped at the same time. This test ensures that listeners fire and isHealthy
* makes sense.
*/
public void testEmptyServiceManager() {
Logger logger = Logger.getLogger(ServiceManager.class.getName());
logger.setLevel(Level.FINEST);
```

```
 TestLogHandler logHandler = new TestLogHandler();
 logger.addHandler(logHandler);
 ServiceManager manager = new ServiceManager(Arrays.<Service>asList());
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener, MoreExecutors.sameThreadExecutor());
 manager.startAsync().awaitHealthy();
 assertTrue(manager.isHealthy());
 assertTrue(listener.healthyCalled);
 assertFalse(listener.stoppedCalled);
 assertTrue(listener.failedServices.isEmpty());
 manager.stopAsync().awaitStopped();
 assertFalse(manager.isHealthy());
 assertTrue(listener.stoppedCalled);
 assertTrue(listener.failedServices.isEmpty());
 // check that our NoOpService is not directly observable via any of the inspection methods or
 // via logging.
 assertEquals("ServiceManager{services=[]}", manager.toString());
 assertTrue(manager.servicesByState().isEmpty());
 assertTrue(manager.startupTimes().isEmpty());
 Formatter logFormatter = new Formatter() {
 @Override public String format(LogRecord record) {
 return formatMessage(record);
 }
 };
 for (LogRecord record : logHandler.getStoredLogRecords()) {
 assertFalse(logFormatter.format(record).contains("NoOpService"));
 }
 }

/**
 * This is for a case where a long running Listener using the sameThreadListener could deadlock
 * another thread calling stopAsync().
 */

public void testListenerDeadlock() throws InterruptedException {
 final CountDownLatch failEnter = new CountDownLatch(1);
 Service failRunService = new AbstractService() {
 @Override protected void doStart() {
 new Thread() {
 @Override public void run() {
 notifyStarted();
 notifyFailed(new Exception("boom"));
 }
 }.start();
 }
```

```
@Override protected void doStop() {
notifyStopped();
}
};
final ServiceManager manager = new ServiceManager(
Arrays.asList(failRunService, new NoOpService()));
manager.addListener(new ServiceManager.Listener() {
@Override public void failure(Service service) {
failEnter.countDown();
// block forever!
Uninterruptibles.awaitUninterruptibly(new CountDownLatch(1));
}
}, MoreExecutors.sameThreadExecutor());
// We do not call awaitHealthy because, due to races, that method may throw an exception. But
// we really just want to wait for the thread to be in the failure callback so we wait for that
// explicitly instead.
manager.startAsync();
failEnter.await();
assertFalse("State should be updated before calling listeners", manager.isHealthy());
// now we want to stop the services.
Thread stoppingThread = new Thread() {
@Override public void run() {
manager.stopAsync().awaitStopped();
}
};
stoppingThread.start();
// this should be super fast since the only non stopped service is a NoOpService
stoppingThread.join(1000);
assertFalse("stopAsync has deadlocked!.", stoppingThread.isAlive());
}

/**
* Catches a bug where when constructing a service manager failed, later interactions with the
* service could cause IllegalStateExceptions inside the partially constructed ServiceManager.
* This ISE wouldn't actually bubble up but would get logged by ExecutionQueue. This obfuscated
* the original error (which was not constructing ServiceManager correctly).
*/
public void testPartiallyConstructedManager() {
Logger logger = Logger.getLogger("global");
logger.setLevel(Level.FINEST);
TestLogHandler logHandler = new TestLogHandler();
logger.addHandler(logHandler);
NoOpService service = new NoOpService();
service.startAsync();
try {
```

```
 new ServiceManager(Arrays.asList(service));
 fail();
 } catch (IllegalArgumentException expected) {}
 service.stopAsync();
 // Nothing was logged!
 assertEquals(0, logHandler.getStoredLogRecords().size());
 }

public void testPartiallyConstructedManager_transitionAfterAddListenerBeforeStateIsReady() {
 // The implementation of this test is pretty sensitive to the implementation <img src="http://ifeve.com/wp-includes/imag
 // ensure that if weird things happen during construction then we get exceptions.
 final NoOpService service1 = new NoOpService();
 // This service will start service1 when addListener is called. This simulates service1 being
 // started asynchronously.
 Service service2 = new Service() {
 final NoOpService delegate = new NoOpService();
 @Override public final void addListener(Listener listener, Executor executor) {
 service1.startAsync();
 delegate.addListener(listener, executor);
 }
 // Delegates from here on down
 @Override public final Service startAsync() {
 return delegate.startAsync();
 }

@Override public final Service stopAsync() {
 return delegate.stopAsync();
 }

@Override public final ListenableFuture<State> start() {
 return delegate.start();
 }

@Override public final ListenableFuture<State> stop() {
 return delegate.stop();
 }

@Override public State startAndWait() {
 return delegate.startAndWait();
 }

@Override public State stopAndWait() {
 return delegate.stopAndWait();
 }
```

```
@Override public final void awaitRunning() {
 delegate.awaitRunning();
 }

@Override public final void awaitRunning(long timeout, TimeUnit unit)
 throws TimeoutException {
 delegate.awaitRunning(timeout, unit);
 }

@Override public final void awaitTerminated() {
 delegate.awaitTerminated();
 }

@Override public final void awaitTerminated(long timeout, TimeUnit unit)
 throws TimeoutException {
 delegate.awaitTerminated(timeout, unit);
 }

@Override public final boolean isRunning() {
 return delegate.isRunning();
 }

@Override public final State state() {
 return delegate.state();
 }

@Override public final Throwable failureCause() {
 return delegate.failureCause();
 }
 };
 try {
 new ServiceManager(Arrays.asList(service1, service2));
 fail();
 } catch (IllegalArgumentException expected) {
 assertTrue(expected.getMessage().contains("started transitioning asynchronously"));
 }
 }

/**
 * This test is for a case where two Service.Listener callbacks for the same service would call
 * transitionService in the wrong order due to a race. Due to the fact that it is a race this
 * test isn't guaranteed to expose the issue, but it is at least likely to become flaky if the
 * race sneaks back in, and in this case flaky means something is definitely wrong.
 *
 * <p>Before the bug was fixed this test would fail at least 30% of the time.
```

```
 */

public void testTransitionRace() throws TimeoutException {
 for (int k = 0; k < 1000; k++) {
 List<Service> services = Lists.newArrayList();
 for (int i = 0; i < 5; i++) {
 services.add(new SnappyShutdownService(i));
 }
 ServiceManager manager = new ServiceManager(services);
 manager.startAsync().awaitHealthy();
 manager.stopAsync().awaitStopped(1, TimeUnit.SECONDS);
 }
 }

/**
 * This service will shutdown very quickly after stopAsync is called and uses a background thread
 * so that we know that the stopping() listeners will execute on a different thread than the
 * terminated() listeners.
 */
 private static class SnappyShutdownService extends AbstractExecutionThreadService {
 final int index;
 final CountDownLatch latch = new CountDownLatch(1);

SnappyShutdownService(int index) {
 this.index = index;
 }

@Override protected void run() throws Exception {
 latch.await();
 }

@Override protected void triggerShutdown() {
 latch.countDown();
 }

@Override protected String serviceName() {
 return this.getClass().getSimpleName() + "[" + index + "]";
 }
 }

public void testNulls() {
 ServiceManager manager = new ServiceManager(Arrays.<Service>asList());
 new NullPointerTester()
 .setDefault(ServiceManager.Listener.class, new RecordingListener())
 .testAllPublicInstanceMethods(manager);
```

```
 }

private static final class RecordingListener extends ServiceManager.Listener {
 volatile boolean healthyCalled;
 volatile boolean stoppedCalled;
 final Set<Service> failedServices = Sets.newConcurrentHashSet();

@Override public void healthy() {
 healthyCalled = true;
 }

@Override public void stopped() {
 stoppedCalled = true;
 }

@Override public void failure(Service service) {
 failedServices.add(service);
 }
 }
}
<pre>
```

6

字符串处理：分割，连接，填充

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");
```

上述代码返回”Harry; Ron; Hermione”。另外，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。 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。

```
Joiner.on(",").join(Arrays.asList(1, 5, 7)); // returns "1,5,7"
```

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

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”
5. 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");
```

上述代码返回 Iterable，其中包含”foo”、”bar”和”qux”。Splitter 可以被设置为按照任何模式、字符、字符串或字符匹配器拆分。

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。 *警告：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]

在以前的 Guava 版本中，StringUtil 类疯狂地膨胀，其拥有很多处理字符串的方法：allAscii、collapse、collapseControlChars、collapseWhitespace、indexOfChars、lastIndexNotOf、numSharedChars、removeChars、removeCrLf、replaceChars、retainAllChars、strip、stripAndCollapse、stripNonDigits。所有这些方法指向两个概念上的问题：

1. 怎么才算匹配字符?
2. 如何处理这些匹配字符?

为了收拾这个泥潭，我们开发了 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——但类似”所有空白字符”或”所有小写字母”的需求太普遍了，Guava 因此创建了这一 API。

然而使用 CharMatcher 的好处更在于它提供了一系列方法，让你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。CharMatcher 实例首先代表概念 1：怎么才算匹配字符? 然后它还提供了很多操作概念 2：如何处理这些匹配字符? 这样的设计使得 API 复杂度的线性增加可以带来灵活性和功能两方面的增长。

```
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
```

注：CharMatcher 只处理 char 类型代表的字符；它不能理解 0x10000 到 0x10FFFF 的 Unicode 增补字符。这些逻辑字符以代理对[surrogate pairs]的形式编码进字符串，而 CharMatcher 只能将这种逻辑字符看待成两个独立的字符。

获取字符匹配器

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		

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]

不要这样做字符集处理：

```
try {
    bytes = string.getBytes("UTF-8");
 } catch (UnsupportedEncodingException e) {
    // how can this possibly happen?
    throw new AssertionError(e);
}
```

试试这样写：

```
bytes = string.getBytes(Charsets.UTF_8);
```

Charsets 针对所有 Java 平台都要保证支持的六种字符集提供了常量引用。尝试使用这些常量，而不是通过名称获取字符集实例。

大小写格式[CaseFormat]

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

CaseFormat的用法很直接：

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

我们 CaseFormat 在某些时候尤其有用，比如编写代码生成器的时候。

7

原生类型

概述

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

在从 Guava 查找原生类型方法之前，可以先查查 Arrays 类，或者对应的基础类型包装类，如 Integer。

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`

Bytes 工具类没有定义任何区分有符号和无符号字节的方法，而是把它们都放到了 SignedBytes 和 UnsignedBytes 工具类中，因为字节类型的符号性比起其它类型要略微含糊一些。

int 和 long 的无符号形式方法在 UnsignedInts 和 UnsignedLongs 类中，但由于这两个类型的大多数用法都是有符号的，Ints 和 Longs 类按照有符号形式处理方法的输入参数。

此外，Guava 为 int 和 long 的无符号形式提供了包装类，即 UnsignedInteger 和 UnsignedLong，以帮助你使用类型系统，以极小的性能消耗对有符号和无符号值进行强制转换。

在本章下面描述的方法签名中，我们用 Wrapper 表示 JDK 包装类，prim 表示原生类型。（Prims 表示相应的 Guava 工具类。）

原生类型数组工具

原生类型数组是处理原生类型集合的最有效方式（从内存和性能双方面考虑）。Guava 为此提供了许多工具方法。

方法签名	描述	类似方法	可用性
List<Wrapper> asList(prim… backingArray)	把数组转为相应包装类的 List	Arrays.asList	符号无关*
prim[] toArray(Collection<Wrapper> collection)	把集合拷贝为数组，和 collection.toArray()一样线程安全	Collection.toArray()	符号无关
prim[] concat(prim[]… arrays)	串联多个原生类型数组	Iterables.concat	符号无关
boolean contains(prim[] array, prim target)	判断原生类型数组是否包含给定值	Collection.contains	符号无关
int indexOf(prim[] array, prim target)	给定值在数组中首次出现处的索引，若不包含此值返回-1	List.indexOf	符号无关
int lastIndexOf(prim[] array, prim target)	给定值在数组最后出现的索引，若不包含此值返回-1	List.lastIndexOf	符号无关
prim min(prim… array)	数组中最小的值	Collections.min	符号相关*
prim max(prim… array)	数组中最大的值	Collections.max	符号相关
String join(String separator, prim… array)	把数组用给定分隔符连接为字符串	Joiner.on(separator).join	符号相关
Comparator<prim[]> lexicographicalComparator()	按字典序比较原生类型数组的 Comparator	Ordering.natural().lexicographical()	符号相关

*符号无关方法存在于 Bytes, Shorts, Ints, Longs, Floats, Doubles, Chars, Booleans。而UnsignedInts, UnsignedLongs, SignedBytes, 或 UnsignedBytes 不存在。

*符号相关方法存在于 SignedBytes, UnsignedBytes, Shorts, Ints, Longs, Floats, Doubles, Chars, Booleans, UnsignedInts, UnsignedLongs。而 Bytes 不存在。

通用工具方法

Guava 为原生类型提供了若干 JDK6 没有的工具方法。但请注意，其中某些方法已经存在于 JDK7 中。

方法签名	描述	可用性
int compare(prim a, prim b)	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	符号相关
prim checkedCast(long value)	把给定 long 值转为某一原生类型，若给定值不符合该原生类型，则抛出 IllegalArgumentException	仅适用于符号相关的整型*
prim saturatedCast(long value)	把给定 long 值转为某一原生类型，若给定值不符合则使用最接近的原生类型值	仅适用于符号相关的整型

*这里的整型包括 byte, short, int, long。不包括 char, boolean, float, 或 double。

**_译者注：不符合主要是指 long 值超出 prim 类型的范围，比如过大的 long 超出 int 范围。_

注：com.google.common.math.DoubleMath 提供了舍入 double 的方法，支持多种舍入模式。相见第 12 章的”浮点数运算”。

字节转换方法

Guava 提供了若干方法，用来把原生类型按**大字节序**与字节数组相互转换。所有这些方法都是符号无关的，此外 Booleans 没有提供任何下面的方法。

方法或字段签名	描述
int BYTES	常量：表示该原生类型需要的字节数
prim fromByteArray(byte[] bytes)	使用字节数组的前 Prims.BYTES 个字节，按**大字节序**返回原生类型值；如果 bytes.length <= Prims.BYTES，抛出 IAE
prim fromBytes(byte b1, ⋯, byte bk)	接受 Prims.BYTES 个字节参数，按**大字节序**返回原生类型值
byte[] toByteArray(prim value)	按**大字节序**返回 value 的字节数组

无符号支持

JDK 原生类型包装类提供了针对有符号类型的方法，而 UnsignedInts 和 UnsignedLongs 工具类提供了相应的无符号通用方法。UnsignedInts 和 UnsignedLongs 直接处理原生类型：使用时，由你自己保证只传入了无符号类型的值。

此外，对 int 和 long，Guava 提供了无符号包装类（UnsignedInteger 和 UnsignedLong），来帮助你以极小的性能消耗，对有符号和无符号类型进行强制转换。

无符号通用工具方法

JDK 的原生类型包装类提供了有符号形式的类似方法。

方法签名	说明
`int UnsignedInts.parseUnsignedInt(String)long UnsignedLongs.parseUnsignedLong(String)`	按无符号十进制解析字符串
`int UnsignedInts.parseUnsignedInt(String string, int radix)long UnsignedLongs.parseUnsignedLong(String string, int radix)`	按无符号的特定进制解析字符串
`String UnsignedInts.toString(int)String UnsignedLongs.toString(long)`	数字按无符号十进制转为字符串
`String UnsignedInts.toString(int value, int radix)String UnsignedLongs.toString(long value, int radix)`	数字按无符号特定进制转为字符串

无符号包装类

无符号包装类包含了若干方法，让使用和转换更容易。

方法签名	说明
UnsignedPrim add(UnsignedPrim), subtract, multiply, divide, remainder	简单算术运算
UnsignedPrim valueOf(BigInteger)	按给定 BigInteger 返回无符号对象，若 BigInteger 为负或不匹配，抛出 IAE
UnsignedPrim valueOf(long)	按给定 long 返回无符号对象，若 long 为负或不匹配，抛出 IAE
UnsignedPrim asUnsigned(prim value)	把给定的值当作无符号类型。例如，UnsignedInteger.asUnsigned(1<<31)的值为 2^{31},尽管 1<<31 当作 int 时是负的
BigInteger bigIntegerValue()	用 BigInteger 返回该无符号对象的值

toString(), toString(int radix)	返回无符号值的字符串表示

译者注：UnsignedPrim 指各种无符号包装类，如 UnsignedInteger、UnsignedLong。

8

区间

范例

```
List scores;
Iterable belowMedian =Iterables.filter(scores,Range.lessThan(median));
...
Range validGrades = Range.closed(1, 12);
for(int grade : ContiguousSet.create(validGrades, DiscreteDomain.integers())) {
...
}
```

简介

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。在形式上，凸性表示对a<=b<=c, range.contains(a)且 range.contains(c)意味着 range.contains(b)。

区间可以延伸至无限——例如，范围”x>3″包括任意大于3的值——也可以被限制为有限，如”2<=x<5″。Guava 用更紧凑的方法表示范围，有数学背景的程序员对此是耳熟能详的：

- (a..b) = {x | a < x < b}
- [a..b] = {x | a <= x <= b}
- [a..b) = {x | a <= x < b}
- (a..b] = {x | a < x <= b}
- (a..+∞) = {x | x > a}
- [a..+∞) = {x | x >= a}
- (−∞..b) = {x | x < b}
- (−∞..b] = {x | x <= b}
- (−∞..+∞) = 所有值

上面的 a、b 称为端点。为了提高一致性，Guava 中的 Range 要求上端点不能小于下端点。上下端点有可能是相等的，但要求区间是闭区间或半开半闭区间（至少有一个端点是包含在区间中的）：

- [a..a]：单元素区间
- [a..a); (a..a]：空区间，但它们是有效的
- (a..a)：无效区间

Guava 用类型 Range 表示区间。所有区间实现都是不可变类型。

构建区间

区间实例可以由 Range 类的静态方法获取：

(a..b)	open(C, C)
[a..b]	closed(C, C)
[a..b)	closedOpen(C, C)
(a..b]	openClosed(C, C)
(a..+∞)	greaterThan(C)
[a..+∞)	atLeast(C)
(−∞..b)	lessThan(C)
(−∞..b]	atMost(C)
(−∞..+∞)	all()

```
Range.closed("left", "right"); //字典序在"left"和"right"之间的字符串，闭区间
Range.lessThan(4.0); //严格小于4.0的double值
```

此外，也可以明确地指定边界类型来构造区间：

有界区间	range(C, BoundType, C, BoundType)
无上界区间：((a..+∞) 或[a..+∞))	downTo(C, BoundType)
无下界区间：((−∞..b) 或(−∞..b])	upTo(C, BoundType)

这里的 BoundType 是一个枚举类型，包含 CLOSED 和 OPEN 两个值。

```
Range.downTo(4, boundType);// (a..+∞)或[a..+∞)，取决于boundType
Range.range(1, CLOSED, 4, OPEN);// [1..4)，等同于Range.closedOpen(1, 4)
```

区间运算

Range 的基本运算是它的 contains(C)http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Range.html#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。此外，Range 实例也可以当作 Predicate，并且在函数式编程中使用（译者注：见第 4 章）。任何 Range 实例也都支持 containsAll(Iterable<? extends C>)方法：

```
Range.closed(1, 3).contains(2);//return true
Range.closed(1, 3).contains(4);//return false
Range.lessThan(5).contains(5); //return false
Range.closed(1, 4).containsAll(Ints.asList(1, 2, 3)); //return true
```

查询运算

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。

```
Range.closedOpen(4, 4).isEmpty(); // returns true
Range.openClosed(4, 4).isEmpty(); // returns true
Range.closed(4, 4).isEmpty(); // returns false
Range.open(4, 4).isEmpty(); // Range.open throws IllegalArgumentException
Range.closed(3, 10).lowerEndpoint(); // returns 3
Range.open(3, 10).lowerEndpoint(); // returns 3
Range.closed(3, 10).lowerBoundType(); // returns CLOSED
Range.open(3, 10).upperBoundType(); // returns OPEN
```

关系运算

包含[enclose]

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！

- [3..6] 包含[4..5]；
- (3..6) 包含(3..6)；
- [3..6] 包含[4..4)，虽然后者是空区间；
- (3..6]不 包含[3..6]；
- [4..5]不 包含(3..6)，虽然前者包含了后者的所有值，离散域[discrete domains]可以解决这个问题（见 8.5 节）；
- [3..6]不 包含(1..1]，虽然前者包含了后者的所有值。

包含是一种偏序关系[partial ordering]。基于包含关系的概念，Range 还提供了以下运算方法。

相连[isConnected]

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）。

相连是一种自反的[reflexive]、对称的[symmetric]关系。

```
Range.closed(3, 5).isConnected(Range.open(5, 10)); // returns true
Range.closed(0, 9).isConnected(Range.closed(3, 4)); // returns true
Range.closed(0, 5).isConnected(Range.closed(3, 9)); // returns true
Range.open(3, 5).isConnected(Range.open(5, 10)); // returns false
Range.closed(1, 5).isConnected(Range.closed(6, 10)); // returns false
```

交集[intersection]

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。

交集是可互换的[commutative] 、关联的[associative] 运算[operation]。

```
Range.closed(3, 5).intersection(Range.open(5, 10)); // returns (5, 5]
Range.closed(0, 9).intersection(Range.closed(3, 4)); // returns [3, 4]
Range.closed(0, 5).intersection(Range.closed(3, 9)); // returns [3, 5]
Range.open(3, 5).intersection(Range.open(5, 10)); // throws IAE
Range.closed(1, 5).intersection(Range.closed(6, 10)); // throws IAE
```

跨区间[span]

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。

span 是可互换的[commutativhttp://code.google.com/p/guava-libraries/wiki/GuavaTermsExplained#commutativee] 、关联的[associative] 、闭合的[closed]运算[operation]。

```
Range.closed(3, 5).span(Range.open(5, 10)); // returns [3, 10)
Range.closed(0, 9).span(Range.closed(3, 4)); // returns [0, 9]
Range.closed(0, 5).span(Range.closed(3, 9)); // returns [0, 9]
Range.open(3, 5).span(Range.open(5, 10)); // returns (3, 10)
Range.closed(1, 5).span(Range.closed(6, 10)); // returns [1, 10]
```

离散域

部分（但不是全部）可比较类型是离散的，即区间的上下边界都是可枚举的。

在 Guava 中，用 DiscreteDomain实现类型 C 的离散形式操作。一个离散域总是代表某种类型值的全集；它不能代表类似”素数”、”长度为 5 的字符串”或”午夜的时间戳”这样的局部域。

DiscreteDomain 提供的离散域实例包括：

类型	离散域
Integer	integers()
Long	longs()

一旦获取了 DiscreteDomain 实例，你就可以使用下面的 Range 运算方法：

- ContiguousSet.create(range, domain)：用 ImmutableSortedSet 形式表示 Range 中符合离散域定义的元素，并增加一些额外操作——译者注：实际返回 ImmutableSortedSet 的子类 ContiguousSet。（对无限区间不起作用，除非类型 C 本身是有限的，比如 int 就是可枚举的）
- canonical(domain)：把离散域转为区间的”规范形式”。如果 ContiguousSet.create(a, domain).equals(ContiguousSet.create(b, domain))并且!a.isEmpty()，则有 a.canonical(domain).equals(b.canonical(domain))。（这并不意味着 a.equals(b)）

```
ImmutableSortedSet set = ContigousSet.create(Range.open(1, 5), iscreteDomain.integers());
//set包含[2, 3, 4]
ContiguousSet.create(Range.greaterThan(0), DiscreteDomain.integers());
//set包含[1, 2, ..., Integer.MAX_VALUE]
```

注意，ContiguousSet.create 并没有真的构造了整个集合，而是返回了 set 形式的区间视图。

你自己的离散域

你可以创建自己的离散域，但必须记住 DiscreteDomain 契约的几个重要方面。

- 一个离散域总是代表某种类型值的全集；它不能代表类似”素数”或”长度为 5 的字符串”这样的局部域。所以举例来说，你无法构造一个 DiscreteDomain 以表示精确到秒的 JODA DateTime 日期集合：因为那将无法包含 JODA DateTime 的所有值。

- DiscreteDomain 可能是无限的——比如 BigInteger DiscreteDomain。这种情况下，你应当用 minValue()和 maxValue()的默认实现，它们会抛出 NoSuchElementException。但 Guava 禁止把无限区间传入 ContiguousSet.create——*译者注：那明显得不到一个可枚举的集合。*

如果我需要一个Comparator呢?

我们想要在 Range 的可用性与 API 复杂性之间找到特定的平衡，这部分导致了我们没有提供基于 Comparator 的接口：我们不需要操心区间是怎样基于不同 Comparator 互动的；所有 API 签名都是简单明确的；这样更好。

另一方面，如果你需要任意 Comparator，可以按下列其中一项来做：

- 使用通用的 Predicate 接口，而不是 Range 类。（ Range 现了 Predicate 接口，因此可以用Predicates.compose(range, function)获取 Predicate 实例 ）
- 使用包装类以定义期望的排序。

译者注：实际上 Range 规定元素类型必须是 Comparable，这已经满足了大多数需求。如果需要自定义特殊的比较逻辑，可以用 Predicates.compose(range, function)组合比较的 function。

9

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。

其中的一些工具方法列举如下：

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()`

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。

创建源与汇

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, 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)`

此外，你也可以继承这些类，以创建新的实现。

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

一旦有了源与汇的实例，就可以进行若干读写操作。

通用操作

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)`

范例

```
//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)
.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。

`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

10

散列

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

在这个包的 Java doc 中，我们可以看到很多不同的类，但是文档中没有明显地表明它们是怎样 一起配合工作的。在介绍散列包中的类之前，让我们先来看下面这段代码范例：

```
HashFunction hf = Hashing.md5();
HashCode hc = hf.newHasher()
    .putLong(id)
    .putString(name, Charsets.UTF_8)
    .putObject(person, personFunnel)
    .hash();
```

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。比如，如果我们有这样一个类：

```
class Person {
    final int id;
    final String firstName;
    final String lastName;
    final int birthYear;
}
```

它对应的 Funnel 实现可能是：

```
Funnel<Person> personFunnel = new Funnel<Person>() {
   @Override
   public void funnel(Person person, PrimitiveSink into) {
      into
         .putInt(person.id)
         .putString(person.firstName, Charsets.UTF_8)
         .putString(person.lastName, Charsets.UTF_8)
         .putInt(birthYear);
   }
}
```

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。布鲁姆过滤器的维基页面对此作了全面的介绍，同时我们推荐 github 中的一个教程。

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，缺省误检率[falsePositiveProbability]为 3%。BloomFilter 提供了 boolean mightContain(T) 和 void put(T)，它们的含义都不言自明了。

```
BloomFilter<Person> friends = BloomFilter.create(personFunnel, 500, 0.01);
for(Person friend : friendsList) {
friends.put(friend);
}

// 很久以后
if (friends.mightContain(dude)) {
//dude不是朋友还运行到这里的概率为1%
//在这儿，我们可以在做进一步精确检查的同时触发一些异步加载
}
```

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。详见一致性哈希。

11

事件总线

传统上，Java 的**进程内事件分发**都是通过发布者和订阅者之间的显式注册实现的。设计 EventBus 就是为了取代这种显示注册方式，使组件间有了更好的解耦。EventBus 不是通用型的发布–订阅实现，不适用于进程间通信。

范例

```
// Class is typically registered by the container.
class EventBusChangeRecorder {
@Subscribe public void recordCustomerChange(ChangeEvent e) {
recordChange(e.getChange());
}
}
// somewhere during initialization
eventBus.register(new EventBusChangeRecorder());
// much later
public void changeCustomer() {
ChangeEvent event = getChangeEvent();
eventBus.post(event);
}
```

一分钟指南

把已有的进程内事件分发系统迁移到 EventBus 非常简单。

事件监听者[Listeners]

监听特定事件（如，CustomerChangeEvent）：

- 传统实现：定义相应的事件监听者类，如 CustomerChangeEventListener；
- EventBus 实现：以 CustomerChangeEvent 为唯一参数创建方法，并用 Subscribe 注解标记。

把事件监听者注册到事件生产者：

- 传统实现：调用事件生产者的 registerCustomerChangeEventListener 方法；这些方法很少定义在公共接口中，因此开发者必须知道所有事件生产者的类型，才能正确地注册监听者；
- EventBus 实现：在 EventBus **实例**上调用 EventBus.register(Object)方法；请保证事件生产者和监听者共享相同的 EventBus **实例**。

按事件超类监听（如，EventObject 甚至 Object）：

- 传统实现：很困难，需要开发者自己去实现匹配逻辑；
- EventBus 实现：EventBus 自动把事件分发给事件超类的监听者，并且允许监听者声明监听接口类型和泛型的通配符类型（wildcard，如？super XXX）。

检测没有监听者的事件：

- 传统实现：在每个事件分发方法中添加逻辑代码（也可能适用 AOP）；
- EventBus 实现：监听 DeadEvent；EventBus 会把所有发布后没有监听者处理的事件包装为 DeadEvent（对调试很便利）。

事件生产者[Producers]

管理和追踪监听者：

传统实现：用列表管理监听者，还要考虑线程同步；或者使用工具类，如 EventListenerList； EventBus实现：EventBus 内部已经实现了监听者管理。

向监听者分发事件：

- 传统实现：开发者自己写代码，包括事件类型匹配、异常处理、异步分发；
- EventBus 实现：把事件传递给 EventBus.post(Object)方法。异步分发可以直接用 EventBus 的子类 AsyncEventBus。

术语表

事件总线系统使用以下术语描述事件分发：

事件	可以向事件总线发布的对象
订阅	向事件总线注册*监听者*以接受事件的行为
监听者	提供一个*处理方法*，希望接受和处理事件的对象
处理方法	监听者提供的公共方法，事件总线使用该方法向监听者发送事件；该方法应该用 Subscribe 注解
发布消息	通过事件总线向所有匹配的监听者提供事件

常见问题解答[FAQ]

为什么一定要创建 EventBus 实例，而不是使用单例模式?

EventBus 不想给定开发者怎么使用；你可以在应用程序中按照不同的组件、上下文或业务主题分别使用不同的事件总线。这样的话，在测试过程中开启和关闭某个部分的事件总线，也会变得更简单，影响范围更小。

当然，如果你想在进程范围内使用唯一的事件总线，你也可以自己这么做。比如在容器中声明 EventBus 为全局单例，或者用一个静态字段存放 EventBus，如果你喜欢的话。

简而言之，EventBus 不是单例模式，是因为我们不想为你做这个决定。你喜欢怎么用就怎么用吧。

**可以从事件总线中注销监听者吗？ **

当然可以，使用 EventBus.unregister(Object)方法，但我们发现这种需求很少：

- 大多数监听者都是在启动或者模块懒加载时注册的，并且在应用程序的整个生命周期都存在；
- 可以使用特定作用域的事件总线来处理临时事件，而不是注册/注销监听者；比如在请求作用域[request-scoped]的对象间分发消息，就可以同样适用请求作用域的事件总线；
- 销毁和重建事件总线的成本很低，有时候可以通过销毁和重建事件总线来更改分发规则。

为什么使用注解标记处理方法，而不是要求监听者实现接口?

我们觉得注解和实现接口一样传达了明确的语义，甚至可能更好。同时，使用注解也允许你把处理方法放到任何地方，和使用业务意图清晰的方法命名。

传统的 Java 实现中，监听者使用方法很少的接口——通常只有一个方法。这样做有一些缺点:

- 监听者类对给定事件类型，只能有单一处理逻辑；
- 监听者接口方法可能冲突；
- 方法命名只和事件相关（handleChangeEvent），不能表达意图（recordChangeInJournal）；
- 事件通常有自己的接口，而没有按类型定义的公共父接口（如所有的UI事件接口）。

接口实现监听者的方式很难做到简洁，这甚至引出了一个模式，尤其是在 Swing 应用中，那就是用匿名类实现事件监听者的接口。比较以下两种实现：

```
class ChangeRecorder {
```

```
    void setCustomer(Customer cust) {
        cust.addChangeListener(new ChangeListener() {
            public void customerChanged(ChangeEvent e) {
                recordChange(e.getChange());
            }
        };
    }
}
```

```
//这个监听者类通常由容器注册给事件总线
class EventBusChangeRecorder {
    @Subscribe public void recordCustomerChange(ChangeEvent e) {
        recordChange(e.getChange());
    }
}
```

第二种实现的业务意图明显更加清晰：没有多余的代码，并且处理方法的名字是清晰和有意义的。

通用的监听者接口 Handler 怎么样?

有些人已经建议过用泛型定义一个通用的监听者接口 Handler。这有点牵扯到 Java 类型擦除的问题，假设我们有如下这个接口：

```
interface Handler<T> {
    void handleEvent(T event);
}
```

因为类型擦除，Java 禁止一个类使用不同的类型参数多次实现同一个泛型接口（即不可能出现 MultiHandler implements Handler, Handler）。这比起传统的 Java 事件机制也是巨大的退步，至少传统的 Java Swing 监听者接口使用了不同的方法把不同的事件区分开。

EventBus 不是破坏了静态类型，排斥了自动重构支持吗?

有些人被 EventBus 的 register(Object) 和 post(Object) 方法直接使用 Object 做参数吓坏了。

这里使用 Object 参数有一个很好的理由：EventBus 对事件监听者类型和事件本身的类型都不作任何限制。

另一方面，处理方法必须要明确地声明参数类型——期望的事件类型（或事件的父类型）。因此，搜索一个事件的类型引用，可以马上找到针对该事件的处理方法，对事件类型的重命名也会在 IDE 中自动更新所有的处理方法。

在 EventBus 的架构下，你可以任意重命名@Subscribe 注解的处理方法，并且这类重命名不会被传播（即不会引起其他类的修改），因为对 EventBus 来说，处理方法的名字是无关紧要的。如果测试代码中直接调用了处理

方法，那么当然，重命名处理方法会引起测试代码的变动，但使用 EventBus 触发处理方法的代码就不会发生变更。我们认为这是 EventBus 的特性，而不是漏洞：能够任意重命名处理方法，可以让你的处理方法命名更清晰。

如果我注册了一个没有任何处理方法的监听者，会发生什么?

什么也不会发生。

EventBus 旨在与容器和模块系统整合，Guice 就是个典型的例子。在这种情况下，可以方便地让容器/工厂/运行环境传递任意创建好的对象给 EventBus 的 register(Object)方法。

这样，任何容器/工厂/运行环境创建的对象都可以简便地通过暴露处理方法挂载到系统的事件模块。

编译时能检测到 EventBus 的哪些问题?

Java 类型系统可以明白地检测到的任何问题。比如，为一个不存在的事件类型定义处理方法。

运行时往 EventBus 注册监听者，可以立即检测到哪些问题?

一旦调用了 register(Object) 方法，EventBus 就会检查监听者中的处理方法是否结构正确的[well-formedness]。具体来说，就是每个用@Subscribe 注解的方法都只能有一个参数。

违反这条规则将引起 IllegalArgumentException（这条规则检测也可以用 APT 在编译时完成，不过我们还在研究中）。

哪些问题只能在之后事件传播的运行时才会被检测到?

如果组件传播了一个事件，但找不到相应的处理方法，EventBus 可能会指出一个错误（通常是指出@Subscribe 注解的缺失，或没有加载监听者组件）。

请注意这个指示并不一定表示应用有问题。一个应用中可能有好多场景会故意忽略某个事件，尤其当事件来源于不可控代码时

你可以注册一个处理方法专门处理 DeadEvent 类型的事件。每当 EventBus 收到没有对应处理方法的事件，它都会将其转化为 DeadEvent，并且传递给你注册的 DeadEvent 处理方法——你可以选择记录或修复该事件。

怎么测试监听者和它们的处理方法?

因为监听者的处理方法都是普通方法，你可以简便地在测试代码中模拟 EventBus 调用这些方法。

为什么我不能在 EventBus 上使用<泛型魔法>?

EventBus 旨在很好地处理一大类用例。我们更喜欢针对大多数用例直击要害，而不是在所有用例上都保持体面。

此外，泛型也让 EventBus 的可扩展性——让它有益、高效地扩展，同时我们对 EventBus 的增补不会和你们的扩展相冲突——成为一个非常棘手的问题。

如果你真的很想用泛型，EventBus 目前还不能提供，你可以提交一个问题并且设计自己的替代方案。

12

数学运算

范例

```
int logFloor = LongMath.log2(n, FLOOR);
int mustNotOverflow = IntMath.checkedMultiply(x, y);
long quotient = LongMath.divide(knownMultipleOfThree, 3, RoundingMode.UNNECESSARY); // fail fast on non-multipl
BigInteger nearestInteger = DoubleMath.roundToBigInteger(d, RoundingMode.HALF_EVEN);
BigInteger sideLength = BigIntegerMath.sqrt(area, CEILING);
```

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。

注意：Guava Math 和 GWT 格外不兼容，这是因为 Java 和 Java Script 语言的运算溢出逻辑不一样。

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。

有溢出检查的运算

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`

```
IntMath.checkedAdd(Integer.MAX_VALUE, Integer.MAX_VALUE); // throws ArithmeticException
```

实数运算

IntMath、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)`

```
// returns 3162277660168379331998893544432718533719555139325 2
BigIntegerMath.sqrt(BigInteger.TEN.pow(99), RoundingMode.HALF_EVEN);
```

附加功能

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)`

*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

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。

`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

13

google Guava 包的 reflection 解析

译者：万天慧(武祖)

由于类型擦除，你不能够在运行时传递泛型类对象——你可能想强制转换它们，并假装这些对象是有泛型的，但实际上它们没有。

举个例子：

```
ArrayList<String> stringList = Lists.newArrayList();
ArrayList<Integer> intList = Lists.newArrayList();
System.out.println(stringList.getClass().isAssignableFrom(intList.getClass()));
returns true, even though ArrayList<String> is not assignable from ArrayList<Integer>
```

Guava 提供了 TypeToken, 它使用了基于反射的技巧甚至让你在运行时都能够巧妙的操作和查询泛型类型。想象一下 TypeToken 是创建，操作，查询泛型类型（以及，隐含的类）对象的方法。

Guice 用户特别注意：TypeToken 与类 Guice 的 TypeLiteral 很相似，但是有一个点特别不同：它能够支持非具体化的类型，例如 T，List，甚至是 List<? extends Number>；TypeLiteral 则不能支持。TypeToken 也能支持序列化并且提供了很多额外的工具方法。

背景：类型擦除与反射

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 对象，你不能够判定这个对象是有泛型类型 ArrayList 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
。

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，会获得代表 List 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。