**引用**

**Java并没有没落，人们很快就会发现这一点**

欢迎阅读我编写的[Java 8](https://jdk8.java.net/" \t "_blank)介绍。本教程将带领你一步一步地认识这门语言的新特性。通过简单明了的代码示例，你将会学习到如何使用默认接口方法，Lambda表达式，方法引用和重复注解。看完这篇教程后，你还将对最新推出的[API](http://download.java.net/jdk8/docs/api/" \t "_blank)有一定的了解，例如：流控制，函数式接口，map扩展和新的时间日期API等等。

**目 录**[**[ - ]**](http://www.iteye.com/magazines/129-Java-8-Tutorial)

1. [允许在接口中有默认方法实现](http://www.iteye.com/magazines/129-Java-8-Tutorial#548)
2. [Lambda表达式](http://www.iteye.com/magazines/129-Java-8-Tutorial#549)
3. [函数式接口](http://www.iteye.com/magazines/129-Java-8-Tutorial#550)
4. [方法和构造函数引用](http://www.iteye.com/magazines/129-Java-8-Tutorial#551)
5. [Lambda的范围](http://www.iteye.com/magazines/129-Java-8-Tutorial#552)
6. [内置函数式接口](http://www.iteye.com/magazines/129-Java-8-Tutorial#553)
7. [Streams](http://www.iteye.com/magazines/129-Java-8-Tutorial#554)
8. [Parallel Streams](http://www.iteye.com/magazines/129-Java-8-Tutorial#555)
9. [Map](http://www.iteye.com/magazines/129-Java-8-Tutorial#556)
10. [时间日期API](http://www.iteye.com/magazines/129-Java-8-Tutorial#557)
11. [Annotations](http://www.iteye.com/magazines/129-Java-8-Tutorial#558)
12. [总结](http://www.iteye.com/magazines/129-Java-8-Tutorial#559)

**允许在接口中有默认方法实现****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

Java 8 允许我们使用default关键字，为接口声明添加非抽象的方法实现。这个特性又被称为扩展方法。下面是我们的第一个例子：

**Java代码**

1. **interface** Formula {
2. **double** calculate(**int** a);
4. **default** **double** sqrt(**int** a) {
5. **return** Math.sqrt(a);
6. }
7. }

在接口Formula中，除了抽象方法caculate以外，还定义了一个默认方法sqrt。Formula的实现类只需要实现抽象方法caculate就可以了。默认方法sqrt可以直接使用。

**Java代码**

1. Formula formula = **new** Formula() {
2. @Override
3. **public** **double** calculate(**int** a) {
4. **return** sqrt(a \* 100);
5. }
6. };
8. formula.calculate(100);     // 100.0
9. formula.sqrt(16);           // 4.0

formula对象以匿名对象的形式实现了Formula接口。代码很啰嗦：用了6行代码才实现了一个简单的计算功能：a\*100开平方根。我们在下一节会看到，Java 8 还有一种更加优美的方法，能够实现包含单个函数的对象。 

**Lambda表达式****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

让我们从最简单的例子开始，来学习如何对一个string列表进行排序。我们首先使用Java 8之前的方法来实现：

**Java代码**

1. List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");
3. Collections.sort(names, **new** Comparator<String>() {
4. @Override
5. **public** **int** compare(String a, String b) {
6. **return** b.compareTo(a);
7. }
8. });

静态工具方法Collections.sort接受一个list，和一个Comparator接口作为输入参数，Comparator的实现类可以对输入的list中的元素进行比较。通常情况下，你可以直接用创建匿名Comparator对象，并把它作为参数传递给sort方法。   
  
除了创建匿名对象以外，Java 8 还提供了一种更简洁的方式，Lambda表达式。

**Java代码**

1. Collections.sort(names, (String a, String b) -> {
2. **return** b.compareTo(a);
3. });

你可以看到，这段代码就比之前的更加简短和易读。但是，它还可以更加简短：

**Java代码**

1. Collections.sort(names, (String a, String b) -> b.compareTo(a));

只要一行代码，包含了方法体。你甚至可以连大括号对{}和return关键字都省略不要。不过这还不是最短的写法：

**Java代码**

1. Collections.sort(names, (a, b) -> b.compareTo(a));

Java编译器能够自动识别参数的类型，所以你就可以省略掉类型不写。让我们再深入地研究一下lambda表达式的威力吧。 

**函数式接口****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

Lambda表达式如何匹配Java的类型系统？每一个lambda都能够通过一个特定的接口，与一个给定的类型进行匹配。一个所谓的函数式接口必须要有且仅有一个抽象方法声明。每个与之对应的lambda表达式必须要与抽象方法的声明相匹配。由于默认方法不是抽象的，因此你可以在你的函数式接口里任意添加默认方法。   
  
任意只包含一个抽象方法的接口，我们都可以用来做成lambda表达式。为了让你定义的接口满足要求，你应当在接口前加上@FunctionalInterface 标注。编译器会注意到这个标注，如果你的接口中定义了第二个抽象方法的话，编译器会抛出异常。   
  
举例：

**Java代码**

1. @FunctionalInterface
2. **interface** Converter<F, T> {
3. T convert(F from);
4. }
6. Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
7. Integer converted = converter.convert("123");
8. System.out.println(converted);    // 123

注意，如果你不写@FunctionalInterface 标注，程序也是正确的。 

**方法和构造函数引用****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

上面的代码实例可以通过静态方法引用，使之更加简洁：

**Java代码**

1. Converter<String, Integer> converter = Integer::valueOf;
2. Integer converted = converter.convert("123");
3. System.out.println(converted);   // 123

Java 8 允许你通过::关键字获取方法或者构造函数的的引用。上面的例子就演示了如何引用一个静态方法。而且，我们还可以对一个对象的方法进行引用：

**Java代码**

1. **class** Something {
2. String startsWith(String s) {
3. **return** String.valueOf(s.charAt(0));
4. }
5. }
7. Something something = **new** Something();
8. Converter<String, String> converter = something::startsWith;
9. String converted = converter.convert("Java");
10. System.out.println(converted);    // "J"

让我们看看如何使用::关键字引用构造函数。首先我们定义一个示例bean，包含不同的构造方法：

**Java代码**

1. **class** Person {
2. String firstName;
3. String lastName;
5. Person() {}
7. Person(String firstName, String lastName) {
8. **this**.firstName = firstName;
9. **this**.lastName = lastName;
10. }
11. }

接下来，我们定义一个person工厂接口，用来创建新的person对象：

**Java代码**

1. **interface** PersonFactory<P **extends** Person> {
2. P create(String firstName, String lastName);
3. }

然后我们通过构造函数引用来把所有东西拼到一起，而不是像以前一样，通过手动实现一个工厂来这么做。

**Java代码**

1. PersonFactory<Person> personFactory = Person::**new**;
2. Person person = personFactory.create("Peter", "Parker");

我们通过Person::new来创建一个Person类构造函数的引用。Java编译器会自动地选择合适的构造函数来匹配PersonFactory.create函数的签名，并选择正确的构造函数形式。 

**Lambda的范围****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

对于lambdab表达式外部的变量，其访问权限的粒度与匿名对象的方式非常类似。你能够访问局部对应的外部区域的局部final变量，以及成员变量和静态变量。   
  
**访问局部变量**   
  
我们可以访问lambda表达式外部的final局部变量：

**Java代码**

1. **final** **int** num = 1;
2. Converter<Integer, String> stringConverter =
3. (from) -> String.valueOf(from + num);
5. stringConverter.convert(2);     // 3

但是与匿名对象不同的是，变量num并不需要一定是final。下面的代码依然是合法的：

**Java代码**

1. **int** num = 1;
2. Converter<Integer, String> stringConverter =
3. (from) -> String.valueOf(from + num);
5. stringConverter.convert(2);     // 3

然而，num在编译的时候被隐式地当做final变量来处理。下面的代码就不合法：

**Java代码**

1. **int** num = 1;
2. Converter<Integer, String> stringConverter =
3. (from) -> String.valueOf(from + num);
4. num = 3;

在lambda表达式内部企图改变num的值也是不允许的。   
  
**访问成员变量和静态变量**   
  
与局部变量不同，我们在lambda表达式的内部能获取到对成员变量或静态变量的读写权。这种访问行为在匿名对象里是非常典型的。

**Java代码**

1. **class** Lambda4 {
2. **static** **int** outerStaticNum;
3. **int** outerNum;
5. **void** testScopes() {
6. Converter<Integer, String> stringConverter1 = (from) -> {
7. outerNum = 23;
8. **return** String.valueOf(from);
9. };
11. Converter<Integer, String> stringConverter2 = (from) -> {
12. outerStaticNum = 72;
13. **return** String.valueOf(from);
14. };
15. }
16. }

**访问默认接口方法**   
  
还记得第一节里面formula的那个例子么？ 接口Formula定义了一个默认的方法sqrt，该方法能够访问formula所有的对象实例，包括匿名对象。这个对lambda表达式来讲则无效。   
  
默认方法无法在lambda表达式内部被访问。因此下面的代码是无法通过编译的：

**Java代码**

1. Formula formula = (a) -> sqrt( a \* 100);

**内置函数式接口****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

JDK 1.8 API中包含了很多内置的函数式接口。有些是在以前版本的Java中大家耳熟能详的，例如Comparator接口，或者Runnable接口。对这些现成的接口进行实现，可以通过@FunctionalInterface 标注来启用Lambda功能支持。   
  
此外，Java 8 API 还提供了很多新的函数式接口，来降低程序员的工作负担。有些新的接口已经在[Google Guava](https://code.google.com/p/guava-libraries/" \t "_blank)库中很有名了。如果你对这些库很熟的话，你甚至闭上眼睛都能够想到，这些接口在类库的实现过程中起了多么大的作用。   
  
**Predicates**   
  
Predicate是一个布尔类型的函数，该函数只有一个输入参数。Predicate接口包含了多种默认方法，用于处理复杂的逻辑动词（and, or，negate）：

**Java代码**

1. Predicate<String> predicate = (s) -> s.length() > 0;
3. predicate.test("foo");              // true
4. predicate.negate().test("foo");     // false
6. Predicate<Boolean> nonNull = Objects::nonNull;
7. Predicate<Boolean> isNull = Objects::isNull;
9. Predicate<String> isEmpty = String::isEmpty;
10. Predicate<String> isNotEmpty = isEmpty.negate();

**Functions**   
  
Function接口接收一个参数，并返回单一的结果。默认方法可以将多个函数串在一起（compse, andThen）：

**Java代码**

1. Function<String, Integer> toInteger = Integer::valueOf;
2. Function<String, String> backToString = toInteger.andThen(String::valueOf);
4. backToString.apply("123");     // "123"

**Suppliers**   
  
Supplier接口产生一个给定类型的结果。与Function不同的是，Supplier没有输入参数。

**Java代码**

1. Supplier<Person> personSupplier = Person::**new**;
2. personSupplier.get();   // new Person

**Consumers**   
  
Consumer代表了在一个输入参数上需要进行的操作。

**Java代码**

1. Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
2. greeter.accept(**new** Person("Luke", "Skywalker"));

**Comparators**   
  
Comparator接口在早期的Java版本中非常著名。Java 8 为这个接口添加了不同的默认方法。

**Java代码**

1. Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);
3. Person p1 = **new** Person("John", "Doe");
4. Person p2 = **new** Person("Alice", "Wonderland");
6. comparator.compare(p1, p2);             // > 0
7. comparator.reversed().compare(p1, p2);  // < 0

**Optionals**   
  
Optional不是一个函数式接口，而是一个精巧的工具接口，用来防止NullPointerEception产生。这个概念在下一节会显得很重要，所以我们在这里快速地浏览一下Optional的工作原理。   
  
Optional是一个简单的值容器，这个值可以是null，也可以是non-null。考虑到一个方法可能会返回一个non-null的值，也可能返回一个空值。为了不直接返回null，我们在Java 8中就返回一个Optional。

**Java代码**

1. Optional<String> optional = Optional.of("bam");
3. optional.isPresent();           // true
4. optional.get();                 // "bam"
5. optional.orElse("fallback");    // "bam"
7. optional.ifPresent((s) -> System.out.println(s.charAt(0)));     // "b"

**Streams****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

java.util.Stream表示了某一种元素的序列，在这些元素上可以进行各种操作。Stream操作可以是中间操作，也可以是完结操作。完结操作会返回一个某种类型的值，而中间操作会返回流对象本身，并且你可以通过多次调用同一个流操作方法来将操作结果串起来（就像StringBuffer的append方法一样————译者注）。Stream是在一个源的基础上创建出来的，例如java.util.Collection中的list或者set（map不能作为Stream的源）。Stream操作往往可以通过顺序或者并行两种方式来执行。   
  
我们先了解一下序列流。首先，我们通过string类型的list的形式创建示例数据：

**Java代码**

1. List<String> stringCollection = **new** ArrayList<>();
2. stringCollection.add("ddd2");
3. stringCollection.add("aaa2");
4. stringCollection.add("bbb1");
5. stringCollection.add("aaa1");
6. stringCollection.add("bbb3");
7. stringCollection.add("ccc");
8. stringCollection.add("bbb2");
9. stringCollection.add("ddd1");

Java 8中的Collections类的功能已经有所增强，你可以之直接通过调用Collections.stream()或者Collection.parallelStream()方法来创建一个流对象。下面的章节会解释这个最常用的操作。   
  
**Filter**   
  
Filter接受一个predicate接口类型的变量，并将所有流对象中的元素进行过滤。该操作是一个中间操作，因此它允许我们在返回结果的基础上再进行其他的流操作（forEach）。ForEach接受一个function接口类型的变量，用来执行对每一个元素的操作。ForEach是一个中止操作。它不返回流，所以我们不能再调用其他的流操作。

**Java代码**

1. stringCollection
2. .stream()
3. .filter((s) -> s.startsWith("a"))
4. .forEach(System.out::println);
6. // "aaa2", "aaa1"

**Sorted**   
  
Sorted是一个中间操作，能够返回一个排过序的流对象的视图。流对象中的元素会默认按照自然顺序进行排序，除非你自己指定一个Comparator接口来改变排序规则。

**Java代码**

1. stringCollection
2. .stream()
3. .sorted()
4. .filter((s) -> s.startsWith("a"))
5. .forEach(System.out::println);
7. // "aaa1", "aaa2"

一定要记住，sorted只是创建一个流对象排序的视图，而不会改变原来集合中元素的顺序。原来string集合中的元素顺序是没有改变的。

**Java代码**

1. System.out.println(stringCollection);
2. // ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1

**Map**   
  
map是一个对于流对象的中间操作，通过给定的方法，它能够把流对象中的每一个元素对应到另外一个对象上。下面的例子就演示了如何把每个string都转换成大写的string. 不但如此，你还可以把每一种对象映射成为其他类型。对于带泛型结果的流对象，具体的类型还要由传递给map的泛型方法来决定。

**Java代码**

1. stringCollection
2. .stream()
3. .map(String::toUpperCase)
4. .sorted((a, b) -> b.compareTo(a))
5. .forEach(System.out::println);
7. // "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"

**Match**   
  
匹配操作有多种不同的类型，都是用来判断某一种规则是否与流对象相互吻合的。所有的匹配操作都是终结操作，只返回一个boolean类型的结果。

**Java代码**

1. **boolean** anyStartsWithA =
2. stringCollection
3. .stream()
4. .anyMatch((s) -> s.startsWith("a"));
6. System.out.println(anyStartsWithA);      // true
8. **boolean** allStartsWithA =
9. stringCollection
10. .stream()
11. .allMatch((s) -> s.startsWith("a"));
13. System.out.println(allStartsWithA);      // false
15. **boolean** noneStartsWithZ =
16. stringCollection
17. .stream()
18. .noneMatch((s) -> s.startsWith("z"));
20. System.out.println(noneStartsWithZ);      // true

**Count**   
  
Count是一个终结操作，它的作用是返回一个数值，用来标识当前流对象中包含的元素数量。

**Java代码**

1. **long** startsWithB =
2. stringCollection
3. .stream()
4. .filter((s) -> s.startsWith("b"))
5. .count();
7. System.out.println(startsWithB);    // 3

**Reduce**   
  
该操作是一个终结操作，它能够通过某一个方法，对元素进行削减操作。该操作的结果会放在一个Optional变量里返回。

**Java代码**

1. Optional<String> reduced =
2. stringCollection
3. .stream()
4. .sorted()
5. .reduce((s1, s2) -> s1 + "#" + s2);
7. reduced.ifPresent(System.out::println);
8. // "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"

**Parallel Streams****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

像上面所说的，流操作可以是顺序的，也可以是并行的。顺序操作通过单线程执行，而并行操作则通过多线程执行。   
  
下面的例子就演示了如何使用并行流进行操作来提高运行效率，代码非常简单。   
  
首先我们创建一个大的list，里面的元素都是唯一的：

**Java代码**

1. **int** max = 1000000;
2. List<String> values = **new** ArrayList<>(max);
3. **for** (**int** i = 0; i < max; i++) {
4. UUID uuid = UUID.randomUUID();
5. values.add(uuid.toString());
6. }

现在，我们测量一下对这个集合进行排序所使用的时间。   
  
**顺序排序**

**Java代码**

1. **long** t0 = System.nanoTime();
3. **long** count = values.stream().sorted().count();
4. System.out.println(count);
6. **long** t1 = System.nanoTime();
8. **long** millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
9. System.out.println(String.format("sequential sort took: %d ms", millis));
11. // sequential sort took: 899 ms

**并行排序**

**Java代码**

1. **long** t0 = System.nanoTime();
3. **long** count = values.parallelStream().sorted().count();
4. System.out.println(count);
6. **long** t1 = System.nanoTime();
8. **long** millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
9. System.out.println(String.format("parallel sort took: %d ms", millis));
11. // parallel sort took: 472 ms

如你所见，所有的代码段几乎都相同，唯一的不同就是把stream()改成了parallelStream(), 结果并行排序快了50%。 

**Map****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

正如前面已经提到的那样，map是不支持流操作的。而更新后的map现在则支持多种实用的新方法，来完成常规的任务。

**Java代码**

1. Map<Integer, String> map = **new** HashMap<>();
3. **for** (**int** i = 0; i < 10; i++) {
4. map.putIfAbsent(i, "val" + i);
5. }
7. map.forEach((id, val) -> System.out.println(val));

上面的代码风格是完全自解释的：putIfAbsent避免我们将null写入；forEach接受一个消费者对象，从而将操作实施到每一个map中的值上。   
  
下面的这个例子展示了如何使用函数来计算map的编码：

**Java代码**

1. map.computeIfPresent(3, (num, val) -> val + num);
2. map.get(3);             // val33
4. map.computeIfPresent(9, (num, val) -> **null**);
5. map.containsKey(9);     // false
7. map.computeIfAbsent(23, num -> "val" + num);
8. map.containsKey(23);    // true
10. map.computeIfAbsent(3, num -> "bam");
11. map.get(3);             // val33

接下来，我们将学习，当给定一个key值时，如何把一个实例从对应的key中移除：

**Java代码**

1. map.remove(3, "val3");
2. map.get(3);             // val33
4. map.remove(3, "val33");
5. map.get(3);             // null

另一个有用的方法：

**Java代码**

1. map.getOrDefault(42, "not found");  // not found

将map中的实例合并也是非常容易的：

**Java代码**

1. map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
2. map.get(9);             // val9
4. map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
5. map.get(9);             // val9concat

合并操作先看map中是否没有特定的key/value存在，如果是，则把key/value存入map，否则merging函数就会被调用，对现有的数值进行修改。 

**时间日期API****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

Java 8 包含了全新的时间日期API，这些功能都放在了java.time包下。新的时间日期API是基于Joda-Time库开发的，但是也不尽相同。下面的例子就涵盖了大多数新的API的重要部分。   
  
**Clock**   
  
Clock提供了对当前时间和日期的访问功能。Clock是对当前时区敏感的，并可用于替代System.currentTimeMillis()方法来获取当前的毫秒时间。当前时间线上的时刻可以用Instance类来表示。Instance也能够用于创建原先的java.util.Date对象。

**Java代码**

1. Clock clock = Clock.systemDefaultZone();
2. **long** millis = clock.millis();
4. Instant instant = clock.instant();
5. Date legacyDate = Date.from(instant);   // legacy java.util.Date

**Timezones**   
  
时区类可以用一个ZoneId来表示。时区类的对象可以通过静态工厂方法方便地获取。时区类还定义了一个偏移量，用来在当前时刻或某时间与目标时区时间之间进行转换。

**Java代码**

1. System.out.println(ZoneId.getAvailableZoneIds());
2. // prints all available timezone ids
4. ZoneId zone1 = ZoneId.of("Europe/Berlin");
5. ZoneId zone2 = ZoneId.of("Brazil/East");
6. System.out.println(zone1.getRules());
7. System.out.println(zone2.getRules());
9. // ZoneRules[currentStandardOffset=+01:00]
10. // ZoneRules[currentStandardOffset=-03:00]

**LocalTime**   
  
本地时间类表示一个没有指定时区的时间，例如，10 p.m.或者17：30:15，下面的例子会用上面的例子定义的时区创建两个本地时间对象。然后我们会比较两个时间，并计算它们之间的小时和分钟的不同。

**Java代码**

1. LocalTime now1 = LocalTime.now(zone1);
2. LocalTime now2 = LocalTime.now(zone2);
4. System.out.println(now1.isBefore(now2));  // false
6. **long** hoursBetween = ChronoUnit.HOURS.between(now1, now2);
7. **long** minutesBetween = ChronoUnit.MINUTES.between(now1, now2);
9. System.out.println(hoursBetween);       // -3
10. System.out.println(minutesBetween);     // -239

LocalTime是由多个工厂方法组成，其目的是为了简化对时间对象实例的创建和操作，包括对时间字符串进行解析的操作。

**Java代码**

1. LocalTime late = LocalTime.of(23, 59, 59);
2. System.out.println(late);       // 23:59:59
4. DateTimeFormatter germanFormatter =
5. DateTimeFormatter
6. .ofLocalizedTime(FormatStyle.SHORT)
7. .withLocale(Locale.GERMAN);
9. LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
10. System.out.println(leetTime);   // 13:37

**LocalDate**   
  
本地时间表示了一个独一无二的时间，例如：2014-03-11。这个时间是不可变的，与LocalTime是同源的。下面的例子演示了如何通过加减日，月，年等指标来计算新的日期。记住，每一次操作都会返回一个新的时间对象。

**Java代码**

1. LocalDate today = LocalDate.now();
2. LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
3. LocalDate yesterday = tomorrow.minusDays(2);
5. LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
6. DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
7. System.out.println(dayOfWeek);    // FRIDAY<span style="font-family: Georgia, 'Times New Roman', 'Bitstream Charter', Times, serif; font-size: 13px; line-height: 19px;">Parsing a LocalDate from a string is just as simple as parsing a LocalTime:</span>

解析字符串并形成LocalDate对象，这个操作和解析LocalTime一样简单。

**Java代码**

1. DateTimeFormatter germanFormatter =
2. DateTimeFormatter
3. .ofLocalizedDate(FormatStyle.MEDIUM)
4. .withLocale(Locale.GERMAN);
6. LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);
7. System.out.println(xmas);   // 2014-12-24

**LocalDateTime**   
  
LocalDateTime表示的是日期-时间。它将刚才介绍的日期对象和时间对象结合起来，形成了一个对象实例。LocalDateTime是不可变的，与LocalTime和LocalDate的工作原理相同。我们可以通过调用方法来获取日期时间对象中特定的数据域。

**Java代码**

1. LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);
3. DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
4. System.out.println(dayOfWeek);      // WEDNESDAY
6. Month month = sylvester.getMonth();
7. System.out.println(month);          // DECEMBER
9. **long** minuteOfDay = sylvester.getLong(ChronoField.MINUTE\_OF\_DAY);
10. System.out.println(minuteOfDay);    // 1439

如果再加上的时区信息，LocalDateTime能够被转换成Instance实例。Instance能够被转换成以前的java.util.Date对象。

**Java代码**

1. Instant instant = sylvester
2. .atZone(ZoneId.systemDefault())
3. .toInstant();
5. Date legacyDate = Date.from(instant);
6. System.out.println(legacyDate);     // Wed Dec 31 23:59:59 CET 2014

格式化日期-时间对象就和格式化日期对象或者时间对象一样。除了使用预定义的格式以外，我们还可以创建自定义的格式化对象，然后匹配我们自定义的格式。

**Java代码**

1. DateTimeFormatter formatter =
2. DateTimeFormatter
3. .ofPattern("MMM dd, yyyy - HH:mm");
5. LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);
6. String string = formatter.format(parsed);
7. System.out.println(string);     // Nov 03, 2014 - 07:13

不同于java.text.NumberFormat，新的DateTimeFormatter类是不可变的，也是线程安全的。   
  
更多的细节，请看[这里](http://download.java.net/jdk8/docs/api/java/time/format/DateTimeFormatter.html" \t "_blank) 

**Annotations****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

Java 8中的注解是可重复的。让我们直接深入看看例子，弄明白它是什么意思。   
  
首先，我们定义一个包装注解，它包括了一个实际注解的数组

**Java代码**

1. **@interface** Hints {
2. Hint[] value();
3. }
5. @Repeatable(Hints.**class**)
6. **@interface** Hint {
7. String value();
8. }

只要在前面加上注解名：@Repeatable，Java 8 允许我们对同一类型使用多重注解：   
  
变体1：使用注解容器（老方法）：

**Java代码**

1. @Hints({@Hint("hint1"), @Hint("hint2")})
2. **class** Person {}

变体2：使用可重复注解（新方法）：

**Java代码**

1. @Hint("hint1")
2. @Hint("hint2")
3. **class** Person {}

使用变体2，Java编译器能够在内部自动对@Hint进行设置。这对于通过反射来读取注解信息来说，是非常重要的。

**Java代码**

1. Hint hint = Person.**class**.getAnnotation(Hint.**class**);
2. System.out.println(hint);                   // null
4. Hints hints1 = Person.**class**.getAnnotation(Hints.**class**);
5. System.out.println(hints1.value().length);  // 2
7. Hint[] hints2 = Person.**class**.getAnnotationsByType(Hint.**class**);
8. System.out.println(hints2.length);          // 2

尽管我们绝对不会在Person类上声明@Hints注解，但是它的信息仍然可以通过getAnnotation(Hints.class)来读取。并且，getAnnotationsByType方法会更方便，因为它赋予了所有@Hints注解标注的方法直接的访问权限。

**Java代码**

1. @Target({ElementType.TYPE\_PARAMETER, ElementType.TYPE\_USE})
2. **@interface** MyAnnotation {}

**总结****[op](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)**

Java 8编程指南就到此告一段落。当然，还有很多内容需要进一步研究和说明。这就需要靠读者您来对JDK 8进行探究了，例如：Arrays.parallelSort, StampedLock和CompletableFuture等等 ———— 我这里只是举几个例子而已。   
  
我希望这个博文能够对您有所帮助，也希望您阅读愉快。完整的教程源代码放在了[GitHub](https://github.com/winterbe/java8-tutorial)上。您可以尽情地[fork](https://github.com/winterbe/java8-tutorial/fork)，并请通过[Twitter](https://twitter.com/benontherun)告诉我您的反馈。   
  
原文链接： [winterbe](http://winterbe.com/posts/2014/03/16/java-8-tutorial/) 翻译： [ImportNew.com](http://www.importnew.com/)- [黄小非](http://www.importnew.com/author/huangxiaofei)   
译文链接： <http://www.importnew.com/10360.html>