目录

[一、接口内允许添加默认实现的方法 1](#_Toc17869)

[二、Lambda 表达式 2](#_Toc29343)

[三、函数式接口 Functional Interface 2](#_Toc9480)

[四、便捷的引用类的构造器及方法 3](#_Toc26019)

[五、Lambda 访问外部变量及接口默认方法 4](#_Toc21760)

[六、Stream 流 9](#_Toc2386)

[七、Parallel-Streams 并行流 12](#_Toc25421)

[八、Map 集合 13](#_Toc13812)

[九、新的日期 API 14](#_Toc3047)

[十、Annotations 注解 17](#_Toc4425)

# 一、接口内允许添加默认实现的方法

Java 8 允许我们通过 default 关键字对接口中定义的抽象方法提供一个默认的实现。

// 定义一个公式接口

interface school{

// 学习

void study();

// 做游戏

default string play(String name) {

return "我和"+ name + "一起做游戏";

}

}

在上面这个接口中，我们除了定义了一个抽象方法 study，还定义了一个带有默认实现的方法 play。 我们在实现这个接口时，可以只需要实现 study方法，默认方法 play可以直接调用即可，也就是说我们可以不必强制实现 sqrt 方法。

注：通过 default 关键字这个新特性，可以非常方便地对之前的接口做拓展，而此接口的实现类不必做任何改动，减少了耦合性。

School school = new School() {

@Override

public void study() {

system.out.println("学习中");

}

};

# 二、Lambda 表达式

在学习 Lambda 表达式之前，我们先来看一段老版本的示例代码，其对一个含有字符串的集合进行排序：

List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {

@Override

public int compare(String a, String b) {

return b.compareTo(a);

}

});

Collections 工具类提供了静态方法 sort 方法，入参是一个 List 集合，一个 Comparator 比较器，创建比较器对象要实现比较方式。以便对给定的 List 集合进行 排序。上面的示例代码创建了一个匿名内部类作为入参，这种类似的操作在我们日常的工作中随处可见。

Java 8 中不再推荐这种写法，而是推荐使用 Lambda 表达：

Collections.sort(names, (String a, String b) -> {

return b.compareTo(a);

});

上面这段代码变得简短很多而且易于阅读。但是我们还可以再精炼一点：

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

对于只包含一行方法的代码块，我们可以省略大括号，直接 return 关键代码即可。追求极致，我们还可以让它再短点：

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

List 集合现在已经添加了 sort 方法。而且 Java 编译器能够根据类型推断机制判断出参数类型。

注：Lambda表达式在下一篇博客就会深入进行讲解，这个很重要。

# 三、函数式接口 Functional Interface

不是每个接口都可以缩写成 Lambda 表达式。只有那些函数式接口（Functional Interface）才能缩写成 Lambda 表示式。

函数式接口（Functional Interface）就是只包含一个抽象方法的声明。针对该接口类型的所有 Lambda 表达式都会与这个抽象方法匹配。

注：Java 8 中允许通过 defualt 关键字来为接口添加默认方法它是不算抽象方法的，因此，你可以毫无顾忌的添加默认方法，它并不违反函数式接口（Functional Interface）的定义。一旦你添加了第二个抽象方法，编译器会立刻抛出错误提示。

总结一下：只要接口中仅仅包含一个抽象方法，我们就可以将其改写为 Lambda 表达式。为了保证一个接口明确的被定义为一个函数式接口（Functional Interface），我们需要为该接口添加注解：@FunctionalInterface。

示例代码：

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);

}

示例代码2：

Converter<String, Integer> converter = (from) -> Integer.valueOf(from);

Integer converted = converter.convert("123");

System.out.println(converted); // 123

注：上面的示例代码，即使去掉 @FunctionalInterface 也是好使的，它仅仅是一种约束而已。

这里解释一下，lambda表达式就理解为一个接口的实现方法，然后但凡是调用这个方法，就会执行lambda中写好的实现方式，这样就很好理解了。

# 四、便捷的引用类的构造器及方法

便捷引用方法指的是，Java 8 中允许你通过 :: 关键字来引用类的方法或构造器。  
这里用代码来演示，更进一步简便的调用方法方式，之后会进行深入详解。  
这里先通过lambda进行简化，但是可以更加简化。

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);

}

Converter<String, Integer> converter = (from) -> Integer.valueOf(from);

Integer converted = converter.convert("123");

System.out.println(converted); // 123

上面这段代码，通过 Java 8 的新特性，进一步简化上面的代码：

Converter<String, Integer> converter = Integer::valueOf;

Integer converted = converter.convert("123");

System.out.println(converted); // 123

上面的代码简单的示例了如何引用静态方法，当然，除了静态方法，我们还可以引用普通方法：

class Something {

String startsWith(String s) {

return String.valueOf(s.charAt(0));

}

}

Something something = new Something();

Converter<String, String> converter = something::startsWith;

String converted = converter.convert("Java");

System.out.println(converted); // "J"

接下来，我们再来看看如何通过 :: 关键字来引用类的构造器。

首先，我们先来定义一个示例类，在类中声明两个构造器：

class Person {

String firstName;

String lastName;

Person() {}

Person(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

}

然后，我们再定义一个工厂接口，用来生成 Person 类：

// Person 工厂

interface PersonFactory<P extends Person> {

P create(String firstName, String lastName);

}

我们可以通过 :: 关键字来引用 Person 类的构造器，来代替手动去实现这个工厂接口：

// 直接引用 Person 构造器

PersonFactory<Person> personFactory = Person::new;

Person person = personFactory.create("Peter", "Parker");

Person::new 这段代码，能够直接引用 Person 类的构造器。然后 Java 编译器能够根据上下文选中正确的构造器去实现 PersonFactory.create 方法。

# 五、Lambda 访问外部变量及接口默认方法

这里主要讲如何在 lambda 表达式中访问外部变量（包括：局部变量，成员变量，静态变量，接口的默认方法.），它与匿名内部类访问外部变量很相似。

1.访问局部变量

在 Lambda 表达式中，我们可以访问外部的 final 类型变量，如下面的示例代码：

// 转换器

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);

}

final int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3

与匿名内部类不同的是，我们不必显式声明 num 变量为 final 类型，下面这段代码同样有效：

int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3

但是 num 变量必须为隐式的 final 类型，何为隐式的 final 呢？就是说到编译期为止，num 对象是不能被改变的，如下面这段代码，就不能被编译通过：

//会编译错误

int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

num = 3;

在 lambda 表达式内部改变 num 值同样编译不通过，需要注意, 比如下面的示例代码：

//会编译错误

int num = 1;

Converter<Integer, String> converter = (from) -> {

String value = String.valueOf(from + num);

num = 3;

return value;

};

2.访问成员变量和静态变量

在 Lambda 表达式中对成员变量和静态变量同样拥有读写权限：

代码示例如下：

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);

}

class Lambda4 {

// 静态变量

static int outerStaticNum;

// 成员变量

int outerNum;

void testScopes() {

Converter<Integer, String> stringConverter1 = (from) -> {

// 对成员变量赋值

outerNum = 23;

return String.valueOf(from);

};

Converter<Integer, String> stringConverter2 = (from) -> {

// 对静态变量赋值

outerStaticNum = 72;

return String.valueOf(from);

};

}

}

3.访问接口的默认方法

@FunctionalInterface

interface Formula {

// 计算

double calculate(int a);

// 求平方根

default double sqrt(int a) {

return Math.sqrt(a);

}

}

我们在接口中定义了一个带有默认实现的 sqrt 求平方根方法，在匿名内部类中我们可以很方便的访问此方法：

Formula formula = new Formula() {

@Override

public double calculate(int a) {

return sqrt(a \* 100);

}

};

但是在 lambda 表达式中可不行,带有默认实现的接口方法，是不能在 lambda 表达式中访问的，这段代码将无法被编译通过。

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

4.内置的函数式接口

JDK 1.8 API 包含了很多内置的函数式接口。其中就包括我们在老版本中经常见到的 Comparator 和 Runnable，Java 8 为他们都添加了 @FunctionalInterface 注解，以用来支持 Lambda 表达式。

值得一提的是，除了 Comparator 和 Runnable 外，还有一些新的函数式接口，它们很多都借鉴于知名的 Google Guava 库。

（1）Predicate 断言

注：这个会之后展开详细讲解

Predicate 是一个可以指定入参类型，并返回 boolean 值的函数式接口。它内部提供了一些带有默认实现的方法，可以被用来组合一个复杂的逻辑判断（and, or, negate）：

Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo"); // true

predicate.negate().test("foo"); // false

Predicate<Boolean> nonNull = Objects::nonNull;

Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;

Predicate<String> isNotEmpty = isEmpty.negate();

（2）Function

Function 函数式接口的作用是，我们可以为其提供一个原料，他给生产一个最终的产品。通过它提供的默认方法，组合,链行处理(compose, andThen)：

Function<String, Integer> toInteger = Integer::valueOf;

Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123"); // "123"

（3）Supplier 生产者

Supplier 与 Function 不同，==它不接受入参，直接为我们生产一个指定的结果，==有点像生产者模式：

class Person {

String firstName;

String lastName;

Person() {}

Person(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

}

Supplier<Person> personSupplier = Person::new;

personSupplier.get(); // new Person

（4）Consumer 消费者

对于 Consumer，我们需要提供入参，用来被消费，如下面这段示例代码：

class Person {

String firstName;

String lastName;

Person() {}

Person(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

}

Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);

greeter.accept(new Person("Luke", "Skywalker"));

（5）Comparator

Comparator 在 Java 8 之前是使用比较普遍的。Java 8 中除了将其升级成了函数式接口，还为它拓展了一些默认方法：

Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");

Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2); // > 0

comparator.reversed().compare(p1, p2); // < 0

（6）Optional

注：这个会之后展开详细讲解，Optional有很多常用的方法，十分实用。

首先，Optional 它不是一个函数式接口，设计它的目的是为了防止空指针异常（NullPointerException），要知道在 Java 编程中， 空指针异常可是臭名昭著的。

让我们来快速了解一下 Optional 要如何使用！你可以将 Optional 看做是包装对象（可能是 null, 也有可能非 null）的容器。当你定义了 一个方法，这个方法返回的对象可能是空，也有可能非空的时候，你就可以考虑用 Optional 来包装它，这也是在 Java 8 被推荐使用的做法。

Optional<String> optional = Optional.of("bam");

optional.isPresent(); // true

optional.get(); // "bam"

optional.orElse("fallback"); // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0))); // "b"

# 六、Stream 流

Stream 流简单来说，我们可以使用 java.util.Stream 对一个包含一个或多个元素的集合做各种操作。这些操作可能是 中间操作 亦或是 终端操作。 终端操作会返回一个结果，而中间操作会返回一个 Stream 流。

注：只能对实现了 java.util.Collection 接口的类做流的操作。Map 不支持Stream 流。

Stream 流支持同步执行，也支持并发执行。

## 1.Filter 过滤

首先，我们创建一个 List 集合：

List<String> stringCollection = new ArrayList<>();

stringCollection.add("ddd2");

stringCollection.add("aaa2");

stringCollection.add("bbb1");

stringCollection.add("aaa1");

stringCollection.add("bbb3");

stringCollection.add("ccc");

stringCollection.add("bbb2");

stringCollection.add("ddd1");

Filter 的入参是一个 Predicate, 上面已经说到，Predicate 是一个断言的中间操作，它能够帮我们筛选出我们需要的集合元素。它的返参同样 是一个 Stream 流，我们可以通过 foreach 终端操作，来打印被筛选的元素：

stringCollection

.stream()

.filter((s) -> s.startsWith("a"))

.forEach(System.out::println);

// "aaa2", "aaa1"

**注：**foreach 是一个终端操作，它的返参是 void, 我们无法对其再次进行流操作。

## 2.Sorted 排序

Sorted 同样是一个中间操作，它的返参是一个 Stream 流。另外，我们可以传入一个 Comparator 用来自定义排序，如果不传，则使用默认的排序规则。

stringCollection

.stream()

.sorted()

.filter((s) -> s.startsWith("a"))

.forEach(System.out::println);

// "aaa1", "aaa2"

需要注意，sorted 不会对 stringCollection 做出任何改变，stringCollection 还是原有的那些个元素，且顺序不变：

System.out.println(stringCollection);

// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1

## 3.Map 转换

中间操作 Map 能够帮助我们将 List 中的每一个元素做功能处理。例如下面的示例，通过 map 我们将每一个 string 转成大写：

stringCollection

.stream()

.map(String::toUpperCase)

.sorted((a, b) -> b.compareTo(a))

.forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"

另外，我们还可以做对象之间的转换，业务中比较常用的是将 DO（数据库对象） 转换成 BO（业务对象） 。

## 4.Match 匹配

顾名思义，match 用来做匹配操作，它的返回值是一个 boolean 类型。通过 match, 我们可以方便的验证一个 list 中是否存在某个类型的元素,主要用来校验操作。

// 验证 list 中 string 是否有以 a 开头的, 匹配到第一个，即返回 true

boolean anyStartsWithA =

stringCollection

.stream()

.anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA); // true

// 验证 list 中 string 是否都是以 a 开头的

boolean allStartsWithA =

stringCollection

.stream()

.allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA); // false

// 验证 list 中 string 是否都不是以 z 开头的,

boolean noneStartsWithZ =

stringCollection

.stream()

.noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ); // true

## 5.Count 计数

count 是一个终端操作，它能够统计 stream 流中的元素总数，返回值是 long 类型。

// 先对 list 中字符串开头为 b 进行过滤，让后统计数量

long startsWithB =

stringCollection

.stream()

.filter((s) -> s.startsWith("b"))

.count();

System.out.println(startsWithB); // 3

## 6.Reduce

Reduce 中文翻译为：减少、缩小。  
**注：**通过入参的 Function，我们能够将 list 归约成一个值。它的返回类型是 Optional 类型

Optional<String> reduced =

stringCollection

.stream()

.sorted()

.reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);

// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"

# 七、Parallel-Streams 并行流

前面章节我们说过，stream 流是支持顺序和并行的。顺序流操作是单线程操作，而并行流是通过多线程来处理的，能够充分利用物理机 多核 CPU 的优势，同时处理速度更快。

首先，我们创建一个包含 1000000 UUID list 集合。

int max = 1000000;

List<String> values = new ArrayList<>(max);

for (int i = 0; i < max; i++) {

UUID uuid = UUID.randomUUID();

values.add(uuid.toString());

}

分别通过顺序流和并行流，对这个 list 进行排序，测算耗时:

顺序流:

顺序流排序

// 纳秒

long t0 = System.nanoTime();

long count = values.stream().sorted().count();

System.out.println(count);

long t1 = System.nanoTime();

// 纳秒转微秒

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);

System.out.println(String.format("顺序流排序耗时: %d ms", millis));

// 顺序流排序耗时: 899 ms

并行流:

// 纳秒

long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();

System.out.println(count);

long t1 = System.nanoTime();

// 纳秒转微秒

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);

System.out.println(String.format("并行流排序耗时: %d ms", millis));

// 并行流排序耗时: 472 ms

同样的逻辑处理，通过并行流，我们的性能提升了近 50%。完成这一切，我们需要做的仅仅是将 stream 改成了 parallelStream。

# 八、Map 集合

前面已经提到过 Map 是不支持 Stream 流的，因为 Map 接口并没有像 Collection 接口那样，定义了 stream() 方法。

注：我们可以对其 key, values, entry 使用 流操作，如 map.keySet().stream(), map.values().stream() 和 map.entrySet().stream().

另外, JDK 8 中对 map 提供了一些其他新特性:

putIfAbsent():

Map<Integer, String> map = new HashMap<>();

for (int i = 0; i < 10; i++) {

// 与老版不同的是，putIfAbent() 方法在 put 之前，

// 会判断 key 是否已经存在，存在则直接返回 value, 否则 put, 再返回 value

map.putIfAbsent(i, "val" + i);

}

// forEach 可以很方便地对 map 进行遍历操作

map.forEach((key, value) -> System.out.println(value));

除了上面的 putIfAbsent() 和 forEach() 外，我们还可以很方便地对某个 key 的值做相关操作：

// computeIfPresent(), 当 key 存在时，才会做相关处理

// 如下：对 key 为 3 的值，内部会先判断值是否存在，存在，则做 value + key 的拼接操作

map.computeIfPresent(3, (num, val) -> val + num);

map.get(3); // val33

// 先判断 key 为 9 的元素是否存在，存在，则做删除操作

map.computeIfPresent(9, (num, val) -> null);

map.containsKey(9); // false

// computeIfAbsent(), 当 key 不存在时，才会做相关处理

// 如下：先判断 key 为 23 的元素是否存在，不存在，则添加

map.computeIfAbsent(23, num -> "val" + num);

map.containsKey(23); // true

// 先判断 key 为 3 的元素是否存在，存在，则不做任何处理

map.computeIfAbsent(3, num -> "bam");

map.get(3); // val33

关于删除操作，JDK 8 中提供了能够新的 remove() API:

map.remove(3, "val3");

map.get(3); // val33

map.remove(3, "val33");

map.get(3); // null

如上代码，只有当给定的 key 和 value 完全匹配时，才会执行删除操作。

关于添加方法，JDK 8 中提供了带有默认值的 getOrDefault() 方法：

// 若 key 42 不存在，则返回 not found

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

对于 value 的合并操作也变得更加简单：

// merge 方法，会先判断进行合并的 key 是否存在，不存在，则会添加元素

map.merge(9, "val9", (value, newValue) -> value.concat(newValue));

map.get(9); // val9

// 若 key 的元素存在，则对 value 执行拼接操作

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));

map.get(9); // val9concat

# 九、新的日期 API

Java 8 中在包 java.time 下添加了新的日期 API. 它和 Joda-Time 库相似，但又不完全相同。接下来，我会通过一些示例代码介绍一下新 API 中 最关键的特性：

1.Clock

Clock 提供对当前日期和时间的访问。我们可以利用它来替代 System.currentTimeMillis() 方法。另外，通过 clock.instant() 能够获取一个 instant 实例， 此实例能够方便地转换成老版本中的 java.util.Date 对象。

Clock clock = Clock.systemDefaultZone();

long millis = clock.millis();

Instant instant = clock.instant();

Date legacyDate = Date.from(instant); // 老版本 java.util.Date

2.Timezones 时区

ZoneId 代表时区类。通过静态工厂方法方便地获取它，入参我们可以传入某个时区编码。另外，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。

System.out.println(ZoneId.getAvailableZoneIds());

// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");

ZoneId zone2 = ZoneId.of("Brazil/East");

System.out.println(zone1.getRules());

System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]

// ZoneRules[currentStandardOffset=-03:00]

3.LocalTime

LocalTime 表示一个没有指定时区的时间类，例如，10 p.m.或者 17：30:15，下面示例代码中，将会使用上面创建的 时区对象创建两个 LocalTime。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。

LocalTime now1 = LocalTime.now(zone1);

LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2)); // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);

long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween); // -3

System.out.println(minutesBetween); // -239

LocalTime 提供多个静态工厂方法，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等。

LocalTime late = LocalTime.of(23, 59, 59);

System.out.println(late); // 23:59:59

DateTimeFormatter germanFormatter =

DateTimeFormatter

.ofLocalizedTime(FormatStyle.SHORT)

.withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);

System.out.println(leetTime); // 13:37

4.LocalDate

LocalDate 是一个日期对象，例如：2014-03-11。它和 LocalTime 一样是个 final 类型对象。下面的例子演示了如何通过加减日，月，年等来计算一个新的日期。

LocalDate, LocalTime, 因为是 final 类型的对象，每一次操作都会返回一个新的时间对象。

LocalDate today = LocalDate.now();

// 今天加一天

LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);

// 明天减两天

LocalDate yesterday = tomorrow.minusDays(2);

// 2014 年七月的第四天

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);

DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();

System.out.println(dayOfWeek); // 星期五

也可以直接解析日期字符串，生成 LocalDate 实例。（和 LocalTime 操作一样简单）

DateTimeFormatter germanFormatter =

DateTimeFormatter

.ofLocalizedDate(FormatStyle.MEDIUM)

.withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);

System.out.println(xmas); // 2014-12-24

5.LocalDateTime

LocalDateTime 是一个日期-时间对象。你也可以将其看成是 LocalDate 和 LocalTime 的结合体。操作上，也大致相同。

LocalDateTime 同样是一个 final 类型对象。

LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();

System.out.println(dayOfWeek); // 星期三

Month month = sylvester.getMonth();

System.out.println(month); // 十二月

// 获取改时间是该天中的第几分钟

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE\_OF\_DAY);

System.out.println(minuteOfDay); // 1439

如果再加上的时区信息，LocalDateTime 还能够被转换成 Instance 实例。Instance 能够被转换成老版本中 java.util.Date 对象。

Instant instant = sylvester

.atZone(ZoneId.systemDefault())

.toInstant();

Date legacyDate = Date.from(instant);

System.out.println(legacyDate); // Wed Dec 31 23:59:59 CET 2014

格式化 LocalDateTime 对象就和格式化 LocalDate 或者 LocalTime 一样。除了使用预定义的格式以外，也可以自定义格式化输出。

DateTimeFormatter formatter =

DateTimeFormatter

.ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);

String string = formatter.format(parsed);

System.out.println(string); // Nov 03, 2014 - 07:13

注意：和 java.text.NumberFormat 不同，新的 DateTimeFormatter 类是 final 类型的，同时也是线程安全的。更多细节请查看这里

# 十、Annotations 注解

在 Java 8 中，注解是可以重复的。Java 8 中，通过 @Repeatable，允许我们对同一个类使用多重注解。让我通过下面的示例代码，来看看到底是咋回事。

首先，我们定义一个包装注解，里面包含了一个有着实际注解的数组：

@interface Hints {

Hint[] value();

}

@Repeatable(Hints.class)

@interface Hint {

String value();

}

第一种形态：使用注解容器（老方法）

@Hints({@Hint("hint1"), @Hint("hint2")})

class Person {}

第二种形态：使用可重复注解（新方法）

@Hint("hint1")

@Hint("hint2")

class Person {}

使用第二种形态，Java 编译器能够在内部自动对 @Hint 进行设置。这对于需要通过反射来读取注解信息时，是非常重要的。

Hint hint = Person.class.getAnnotation(Hint.class);

System.out.println(hint); // null

Hints hints1 = Person.class.getAnnotation(Hints.class);

System.out.println(hints1.value().length); // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);

System.out.println(hints2.length); // 2

尽管我们绝对不会在 Person 类上声明 @Hints 注解，但是它的信息仍然是可以通过 getAnnotation(Hints.class) 来读取的。 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。

@Target({ElementType.TYPE\_PARAMETER, ElementType.TYPE\_USE})

@interface MyAnnotation {}