链滴

JAVA8 新特性

作者：shixiaoxiang

原文链接：https://ld246.com/article/1509083374640

来源网站：链滴

前言： Java 8 已经发布很久了，很多报道表明Java 8 是一次重大的版本升级。在Java Code Geeks已经有很多介绍Java 8新特性的文章，例如Playing with Java 8 – Lambdas and Concurrency、Java 8 Date Time API Tutorial : LocalDateTime和Abstract Class Versus Interface in the JDK 8 Era。文还参考了一些其他资料，例如：15 Must Read Java 8 Tutorials和The Dark Side of Java 8。本文合了上述资料，整理成一份关于Java 8新特性的参考教材，希望你有所收获。

1. 简介

毫无疑问，Java 8是Java自Java 5（发布于2004年）之后的最重要的版本。这个版本包含语言、编译、库、工具和JVM等方面的十多个新特性。在本文中我们将学习这些新特性，并用实际的例子说明在么场景下适合使用。

这个教程包含Java开发者经常面对的几类问题：

- 语言
- 编译器
- 库
- 工具
- 运行时（JVM）

2. Java语言的新特性

Java 8是Java的一个重大版本，有人认为，虽然这些新特性领Java开发人员十分期待，但同时也需要不少精力去学习。在这一小节中，我们将介绍Java 8的大部分新特性。

2.1 Lambda表达式和函数式接口

Lambda表达式（也称为闭包）是Java 8中最大和最令人期待的语言改变。它允许我们将函数当成参传递给某个方法，或者把代码本身当作数据处理：函数式开发者非常熟悉这些概念。很多JVM平台上语言（Groovy、Scala等）从诞生之日就支持Lambda表达式，但是Java开发者没有选择，只能使用名内部类代替Lambda表达式。

Lambda的设计耗费了很多时间和很大的社区力量，最终找到一种折中的实现方案，可以实现简洁而凑的语言结构。最简单的Lambda表达式可由逗号分隔的参数列表、**->**符号和语句块组成，例如：

```
Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );
```

在上面这个代码中的参数**e**的类型是由编译器推理得出的，你也可以显式指定该参数的类型，例如：

```
Arrays.asList( "a", "b", "d" ).forEach( ( String e ) -> System.out.println( e ) );
```

如果Lambda表达式需要更复杂的语句块，则可以使用花括号将该语句块括起来，类似于Java中的函体，例如：

```
Arrays.asList( "a", "b", "d" ).forEach( e -> {
    System.out.print( e );
    System.out.print( e );
} );
```

Lambda表达式可以引用类成员和局部变量（会将这些变量隐式得转换成**final**的），例如下列两个代

块的效果完全相同：

```
String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
    ( String e ) -> System.out.print( e + separator ) );
```

和

```
final String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
    ( String e ) -> System.out.print( e + separator ) );
```

Lambda表达式有返回值，返回值的类型也由编译器推理得出。如果Lambda表达式中的语句块只有行，则可以不用使用**return**语句，下列两个代码片段效果相同：

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> e1.compareTo( e2 ) );
```

和

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> {
    int result = e1.compareTo( e2 );
    return result;
} );
```

Lambda的设计者们为了让现有的功能与Lambda表达式良好兼容，考虑了很多方法，于是产生了**数接口**这个概念。函数接口指的是只有一个函数的接口，这样的接口可以隐式转换为Lambda表达式。**ava.lang.Runnable和** java.util.concurrent.Callable**是函数式接口的最佳例子**在实践中，函数式接口非常脆弱：只要某个开发者在该接口中添加一个函数，则该接口就不再是函数接口进而导致编译失败。为了克服这种代码层面的脆弱性，并显式说明某个接口是函数式接口，Java 提供了一个特殊的注解**@FunctionalInterface**（Java 库中的所有相关接口都已经带有这个注解了，举个简单的函数式接口的定义：

```
@FunctionalInterface
public interface Functional {
    void method();
}
```

不过有一点需要注意，默认方法和静态方法不会破坏函数式接口的定义，因此如下的代码是合法的。

```
@FunctionalInterface
public interface FunctionalDefaultMethods {
    void method();

    default void defaultMethod() {
    }
}
```

Lambda表达式作为Java 8的最大卖点，它有潜力吸引更多的开发者加入到JVM平台，并在纯Java编中使用函数式编程的概念。如果你需要了解更多Lambda表达式的细节，可以参考官方文档。

2.2 接口的默认方法和静态方法

Java 8使用两个新概念扩展了接口的含义：默认方法和静态方法。默认方法使得接口有点类似traits，过要实现的目标不一样。默认方法使得开发者可以在 不破坏二进制兼容性的前提下，往现存接口中添

新的方法，即不强制那些实现了该接口的类也同时实现这个新加的方法。

默认方法和抽象方法之间的区别在于抽象方法需要实现，而默认方法不需要。接口提供的默认方法会接口的实现类继承或者覆写，例子代码如下：

```
private interface Defaulable {
    // Interfaces now allow default methods, the implementer may or
    // may not implement (override) them.
    default String notRequired() {
        return "Default implementation";
    }
}

private static class DefaultableImpl implements Defaulable {
}

private static class OverridableImpl implements Defaulable {
    @Override
    public String notRequired() {
        return "Overridden implementation";
    }
}
```

Defaulable接口使用关键字**default**定义了一个默认方法**notRequired()**。**DefaultableImpl**类实现这个接口，同时默认继承了这个接口中的默认方法；**OverridableImpl**类也实现了这个接口，但覆写该接口的默认方法，并提供了一个不同的实现。

Java 8带来的另一个有趣的特性是在接口中可以定义静态方法，例子代码如下：

```
private interface DefaulableFactory {
    // Interfaces now allow static methods
    static Defaulable create( Supplier< Defaulable > supplier ) {
        return supplier.get();
    }
}
```

下面的代码片段整合了默认方法和静态方法的使用场景：

```
public static void main( String[] args ) {
    Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new );
    System.out.println( defaulable.notRequired() );

    defaulable = DefaulableFactory.create( OverridableImpl::new );
    System.out.println( defaulable.notRequired() );
}
```

这段代码的输出结果如下：

```
Default implementation
Overridden implementation
```

由于JVM上的默认方法的实现在字节码层面提供了支持，因此效率非常高。默认方法允许在不打破现继承体系的基础上改进接口。该特性在官方库中的应用是：给**java.util.Collection**接口添加新方法，如**tream()**、**parallelStream()**、**forEach()和removeIf()**等等。

尽管默认方法有这么多好处，但在实际开发中应该谨慎使用：在复杂的继承体系中，默认方法可能引歧义和编译错误。如果你想了解更多细节，可以参考官方文档。

2.3 方法引用

方法引用使得开发者可以直接引用现存的方法、Java类的构造方法或者实例对象。方法引用和Lambd表达式配合使用，使得java类的构造方法看起来紧凑而简洁，没有很多复杂的模板代码。

西门的例子中，**Car**类是不同方法引用的例子，可以帮助读者区分四种类型的方法引用。

```
public static class Car {
    public static Car create( final Supplier< Car > supplier ) {
        return supplier.get();
    }

    public static void collide( final Car car ) {
        System.out.println( "Collided " + car.toString() );
    }

    public void follow( final Car another ) {
        System.out.println( "Following the " + another.toString() );
    }

    public void repair() {
        System.out.println( "Repaired " + this.toString() );
    }
}
```

第一种方法引用的类型是构造器引用，语法是**Class::new**，或者更一般的形式：**Class::new**。注意：个构造器没有参数。

```
final Car car = Car.create( Car::new );
final List< Car > cars = Arrays.asList( car );
```

第二种方法引用的类型是静态方法引用，语法是**Class::static_method**。注意：这个方法接受一个Ca类型的参数。

```
cars.forEach( Car::collide );
```

第三种方法引用的类型是某个类的成员方法的引用，语法是**Class::method**，注意，这个方法没有定入参：

```
cars.forEach( Car::repair );
```

第四种方法引用的类型是某个实例对象的成员方法的引用，语法是**instance::method**。注意：这个法接受一个Car类型的参数：

```
final Car police = Car.create( Car::new );
cars.forEach( police::follow );
```

运行上述例子，可以在控制台看到如下输出（Car实例可能不同）：

```
Collided com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
Repaired com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
```

```
Following the com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197
```

如果想了解和学习更详细的内容，可以参考官方文档

2.4 重复注解

自从Java 5中引入注解以来，这个特性开始变得非常流行，并在各个框架和项目中被广泛使用。不过注解有一个很大的限制是：在同一个地方不能多次使用同一个注解。Java 8打破了这个限制，引入了复注解的概念，允许在同一个地方多次使用同一个注解。

在Java 8中使用**@Repeatable**注解定义重复注解，实际上，这并不是语言层面的改进，而是编译做的一个trick，底层的技术仍然相同。可以利用下面的代码说明：

```
package com.javacodegeeks.java8.repeatable.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

public class RepeatingAnnotations {
    @Target( ElementType.TYPE )
    @Retention( RetentionPolicy.RUNTIME )
    public @interface Filters {
        Filter[] value();
    }

    @Target( ElementType.TYPE )
    @Retention( RetentionPolicy.RUNTIME )
    @Repeatable( Filters.class )
    public @interface Filter {
        String value();
    };

    @Filter( "filter1" )
    @Filter( "filter2" )
    public interface Filterable {
    }

    public static void main(String[] args) {
        for( Filter filter: Filterable.class.getAnnotationsByType( Filter.class ) ) {
            System.out.println( filter.value() );
        }
    }
}
```

正如我们所见，这里的**Filter**类使用@Repeatable(Filters.class)注解修饰，而**Filters**是存放**Filter**注的容器，编译器尽量对开发者屏蔽这些细节。这样，**Filterable**接口可以用两个**Filter**注解注释（这里没有提到任何关于Filters的信息）。

另外，反射API提供了一个新的方法：**getAnnotationsByType()**，可以返回某个类型的重复注解，如Filterable.class.getAnnoation(Filters.class)将返回两个Filter实例，输出到控制台的内容如下所示：

```
filter1
filter2
```

如果你希望了解更多内容，可以参考官方文档。

2.5 更好的类型推断

Java 8编译器在类型推断方面有很大的提升，在很多场景下编译器可以推导出某个参数的数据类型，而使得代码更为简洁。例子代码如下：

```
package com.javacodegeeks.java8.type.inference;

public class Value< T > {
    public static< T > T defaultValue() {
        return null;
    }

    public T getOrDefault( T value, T defaultValue ) {
        return ( value != null ) ? value : defaultValue;
    }
}
```

下列代码是**Value**类型的应用：

```
package com.javacodegeeks.java8.type.inference;

public class TypeInference {
    public static void main(String[] args) {
        final Value< String > value = new Value<>();
        value.getOrDefault( "22", Value.defaultValue() );
    }
}
```

参数**Value.defaultValue()**的类型由编译器推导得出，不需要显式指明。在Java 7中这段代码会有译错误，除非使用Value.defaultValue()。

2.6 拓宽注解的应用场景

Java 8拓宽了注解的应用场景。现在，注解几乎可以使用在任何元素上：局部变量、接口类型、超类接口实现类，甚至可以用在函数的异常定义上。下面是一些例子：

```
package com.javacodegeeks.java8.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.util.ArrayList;
import java.util.Collection;

public class Annotations {
    @Retention( RetentionPolicy.RUNTIME )
    @Target( { ElementType.TYPE_USE, ElementType.TYPE_PARAMETER } )
```

```
    public @interface NonEmpty {
    }

    public static class Holder< @NonEmpty T > extends @NonEmpty Object {
        public void method() throws @NonEmpty Exception {
        }
    }

    @SuppressWarnings( "unused" )
    public static void main(String[] args) {
        final Holder< String > holder = new @NonEmpty Holder< String >();
        @NonEmpty Collection< @NonEmpty String > strings = new ArrayList<>();
    }
}
```

ElementType.TYPE_USER和**ElementType.TYPE_PARAMETER**是Java 8新增的两个注解，用于述注解的使用场景。Java 语言也做了对应的改变，以识别这些新增的注解。

3. Java编译器的新特性

3.1 参数名称

为了在运行时获得Java程序中方法的参数名称，老一辈的Java程序员必须使用不同方法，例如Paranaer liberary。Java 8终于将这个特性规范化，在语言层面（使用反射API和**Parameter.getName()方法**和字节码层面（使用新的**javac**编译器以及**-parameters**参数）提供支持。

```
package com.javacodegeeks.java8.parameter.names;

import java.lang.reflect.Method;
import java.lang.reflect.Parameter;

public class ParameterNames {
    public static void main(String[] args) throws Exception {
        Method method = ParameterNames.class.getMethod( "main", String[].class );
        for( final Parameter parameter: method.getParameters() ) {
            System.out.println( "Parameter: " + parameter.getName() );
        }
    }
}
```

在Java 8中这个特性是默认关闭的，因此如果不带**-parameters**参数编译上述代码并运行，则会出如下结果：

```
Parameter: arg0
```

如果带**-parameters**参数，则会输出如下结果（正确的结果）：

```
Parameter: args
```

如果你使用Maven进行项目管理，则可以在**maven-compiler-plugin**编译器的配置项中配置**-parameters**参数：

```
<plugin>
```

```
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <compilerArgument>-parameters</compilerArgument>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
</plugin>
```

4. Java官方库的新特性

Java 8增加了很多新的工具类（date/time类），并扩展了现存的工具类，以支持现代的并发编程、数式编程等。

4.1 Optional

Java应用中最常见的bug就是空值异常。在Java 8之前，Google Guava引入了**Optionals**类来解决**NullPointerException**，从而避免源码被各种**null**检查污染，以便开发者写出更加整洁的代码。Java 8将**Optional**加入了官方库。

Optional仅仅是一个容易：存放T类型的值或者null。它提供了一些有用的接口来避免显式的null检，可以参考Java 8官方文档了解更多细节。

接下来看一点使用**Optional**的例子：可能为空的值或者某个类型的值：

```
Optional< String > fullName = Optional.ofNullable( null );
System.out.println( "Full Name is set? " + fullName.isPresent() );
System.out.println( "Full Name: " + fullName.orElseGet( () -> "[none]" ) );
System.out.println( fullName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );
```

如果**Optional**实例持有一个非空值，则**isPresent()**方法返回true，否则返回false；**orElseGet()*方法，**Optional**实例持有null，则可以接受一个lambda表达式生成的默认值；**map()方法可以将现的Opetional** 实例的值转换成新的值；**orElse()方法与orElseGet()**方法类似，但是在持有null的候返回传入的默认值。

上述代码的输出结果如下：

```
Full Name is set? false
Full Name: [none]
Hey Stranger!
```

再看下另一个简单的例子：

```
Optional< String > firstName = Optional.of( "Tom" );
System.out.println( "First Name is set? " + firstName.isPresent() );
System.out.println( "First Name: " + firstName.orElseGet( () -> "[none]" ) );
System.out.println( firstName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );
System.out.println();
```

这个例子的输出是：

```
First Name is set? true
First Name: Tom
```

Hey Tom!

如果想了解更多的细节，请参考官方文档。

4.2 Streams

新增的Stream API（java.util.stream）将生成环境的函数式编程引入了Java库中。这是目前为止最的一次对Java库的完善，以便开发者能够写出更加有效、更加简洁和紧凑的代码。

Stream API极大得简化了集合操作（后面我们会看到不止是集合），首先看下这个叫Task的类：

```
public class Streams  {
    private enum Status {
        OPEN, CLOSED
    };

    private static final class Task {
        private final Status status;
        private final Integer points;

        Task( final Status status, final Integer points ) {
            this.status = status;
            this.points = points;
        }

        public Integer getPoints() {
            return points;
        }

        public Status getStatus() {
            return status;
        }

        @Override
        public String toString() {
            return String.format( "[%s, %d]", status, points );
        }
    }
}
```

Task类有一个分数（或伪复杂度）的概念，另外还有两种状态：OPEN或者CLOSED。现在假设有一个ask集合：

```
final Collection< Task > tasks = Arrays.asList(
    new Task( Status.OPEN, 5 ),
    new Task( Status.OPEN, 13 ),
    new Task( Status.CLOSED, 8 )
);
```

首先看一个问题：在这个task集合中一共有多少个OPEN状态的点？在Java 8之前，要解决这个问题则需要使用**foreach**循环遍历task集合；但是在Java 8中可以利用steams解决：包括一系列元素的列，并且支持顺序和并行处理。

```
// Calculate total points of all active tasks using sum()
```

```
final long totalPointsOfOpenTasks = tasks
    .stream()
    .filter( task -> task.getStatus() == Status.OPEN )
    .mapToInt( Task::getPoints )
    .sum();

System.out.println( "Total points: " + totalPointsOfOpenTasks );
```

运行这个方法的控制台输出是：

```
Total points: 18
```

这里有很多知识点值得说。首先，tasks集合被转换成steam表示；其次，在steam上的**filter**操作会滤掉所有CLOSED的task；第三，**mapToInt**操作基于每个task实例的**Task::getPoints**方法将task流换成Integer集合；最后，通过**sum**方法计算总和，得出最后的结果。

在学习下一个例子之前，还需要记住一些steams（点此更多细节）的知识点。Steam之上的操作可分中间操作和晚期操作。

中间操作会返回一个新的steam——执行一个中间操作（例如**filter**）并不会执行实际的过滤操作，是创建一个新的steam，并将原steam中符合条件的元素放入新创建的steam。

晚期操作（例如**forEach**或者**sum**），会遍历steam并得出结果或者附带结果；在执行晚期操作之后，team处理线已经处理完毕，就不能使用了。在几乎所有情况下，晚期操作都是立刻对steam进行遍历。

steam的另一个价值是创造性地支持并行处理（parallel processing）。对于上述的tasks集合，我们以用下面的代码计算所有任务的点数之和：

```
// Calculate total points of all tasks
final double totalPoints = tasks
  .stream()
  .parallel()
  .map( task -> task.getPoints() ) // or map( Task::getPoints )
  .reduce( 0, Integer::sum );

System.out.println( "Total points (all tasks): " + totalPoints );
```

这里我们使用**parallel**方法并行处理所有的task，并使用**reduce**方法计算最终的结果。控制台输出如：

```
Total points（all tasks）: 26.0
```

对于一个集合，经常需要根据某些条件对其中的元素分组。利用steam提供的API可以很快完成这类务，代码如下：

```
// Group tasks by their status
final Map< Status, List< Task > > map = tasks
    .stream()
    .collect( Collectors.groupingBy( Task::getStatus ) );
System.out.println( map );
```

控制台的输出如下：

```
{CLOSED=[[CLOSED, 8]], OPEN=[[OPEN, 5], [OPEN, 13]]}
```

最后一个关于tasks集合的例子问题是：如何计算集合中每个任务的点数在集合中所占的比重，具体理的代码如下：

```
// Calculate the weight of each tasks (as percent of total points)
final Collection< String > result = tasks
    .stream()                                   // Stream< String >
    .mapToInt( Task::getPoints )                    // IntStream
    .asLongStream()                             // LongStream
    .mapToDouble( points -> points / totalPoints )   // DoubleStream
    .boxed()                                    // Stream< Double >
    .mapToLong( weigth -> ( long )( weigth * 100 ) ) // LongStream
    .mapToObj( percentage -> percentage + "%" )      // Stream< String>
    .collect( Collectors.toList() );                // List< String >

System.out.println( result );
```

控制台输出结果如下：

```
[19%, 50%, 30%]
```

最后，正如之前所说，Steam API不仅可以作用于Java集合，传统的IO操作（从文件或者网络一行一得读取数据）可以受益于steam处理，这里有一个小例子：

```
final Path path = new File( filename ).toPath();
try( Stream< String > lines = Files.lines( path, StandardCharsets.UTF_8 ) ) {
    lines.onClose( () -> System.out.println("Done!") ).forEach( System.out::println );
}
```

Stream的方法**onClose** 返回一个等价的有额外句柄的Stream，当Stream的close（）方法被调用的候这个句柄会被执行。Stream API、Lambda表达式还有接口默认方法和静态方法支持的方法引用，Java 8对软件开发的现代范式的响应。

4.3 Date/Time API(JSR 310)

Java 8引入了新的Date-Time API(JSR 310)来改进时间、日期的处理。时间和日期的管理一直是最令Java开发者痛苦的问题。**java.util.Date**和后来的**java.util.Calendar**一直没有解决这个问题（甚至令发者更加迷茫）。

因为上面这些原因，诞生了第三方库Joda-Time，可以替代Java的时间管理API。Java 8中新的时间日期管理API深受Joda-Time影响，并吸收了很多Joda-Time的精华。新的java.time包包含了所有关日期、时间、时区、Instant（跟日期类似但是精确到纳秒）、duration（持续时间）和时钟操作的类新设计的API认真考虑了这些类的不变性（从java.util.Calendar吸取的教训），如果某个实例需要修，则返回一个新的对象。

我们接下来看看java.time包中的关键类和各自的使用例子。首先，**Clock**类使用时区来返回当前的纳时间和日期。**Clock**可以替代**System.currentTimeMillis()和TimeZone.getDefault()**。

```
// Get the system clock as UTC offset
final Clock clock = Clock.systemUTC();
System.out.println( clock.instant() );
System.out.println( clock.millis() );
```

这个例子的输出结果是：

```
2014-04-12T15:19:29.282Z
1397315969360
```

第二，关注下**LocalDate**和**LocalTime**类。**LocalDate**仅仅包含ISO-8601日历系统中的日期部分；**LocalTime**则仅仅包含该日历系统中的时间部分。这两个类的对象都可以使用Clock对象构建得到。

```
// Get the local date and local time
final LocalDate date = LocalDate.now();
final LocalDate dateFromClock = LocalDate.now( clock );

System.out.println( date );
System.out.println( dateFromClock );

// Get the local date and local time
final LocalTime time = LocalTime.now();
final LocalTime timeFromClock = LocalTime.now( clock );

System.out.println( time );
System.out.println( timeFromClock );
```

上述例子的输出结果如下：

```
2014-04-12
2014-04-12
11:25:54.568
15:25:54.568
```

LocalDateTime类包含了LocalDate和LocalTime的信息，但是不包含ISO-8601日历系统中的时区信息。这里有一些关于LocalDate和LocalTime的例子：

```
// Get the local date/time
final LocalDateTime datetime = LocalDateTime.now();
final LocalDateTime datetimeFromClock = LocalDateTime.now( clock );

System.out.println( datetime );
System.out.println( datetimeFromClock );
```

上述这个例子的输出结果如下：

```
2014-04-12T11:37:52.309
2014-04-12T15:37:52.309
```

如果你需要特定时区的data/time信息，则可以使用**ZoneDateTime**，它保存有ISO-8601日期系统日期和时间，而且有时区信息。下面是一些使用不同时区的例子：

```
// Get the zoned date/time
final ZonedDateTime zonedDatetime = ZonedDateTime.now();
final ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now( clock );
final ZonedDateTime zonedDatetimeFromZone = ZonedDateTime.now( ZoneId.of( "America/
os_Angeles" ) );

System.out.println( zonedDatetime );
System.out.println( zonedDatetimeFromClock );
System.out.println( zonedDatetimeFromZone );
```

这个例子的输出结果是：

```
2014-04-12T11:47:01.017-04:00[America/New_York]
2014-04-12T15:47:01.017Z
2014-04-12T08:47:01.017-07:00[America/Los_Angeles]
```

最后看下**Duration**类，它持有的时间精确到秒和纳秒。这使得我们可以很容易得计算两个日期之间不同，例子代码如下：

```
// Get duration between two dates
final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL, 16, 0, 0, 0 );
final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16, 23, 59, 59 );

final Duration duration = Duration.between( from, to );
System.out.println( "Duration in days: " + duration.toDays() );
System.out.println( "Duration in hours: " + duration.toHours() );
```

这个例子用于计算2014年4月16日和2015年4月16日之间的天数和小时数，输出结果如下：

```
Duration in days: 365
Duration in hours: 8783
```

对于Java 8的新日期时间的总体印象还是比较积极的，一部分是因为Joda-Time的积极影响，另一部是因为官方终于听取了开发人员的需求。如果希望了解更多细节，可以参考官方文档。

4.4 Nashorn JavaScript引擎

Java 8提供了新的Nashorn JavaScript引擎，使得我们可以在JVM上开发和运行JS应用。Nashorn JaScript引擎是javax.script.ScriptEngine的另一个实现版本，这类Script引擎遵循相同的规则，允许Ja和JavaScript交互使用，例子代码如下：

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName( "JavaScript" );

System.out.println( engine.getClass().getName() );
System.out.println( "Result:" + engine.eval( "function f() { return 1; }; f() + 1;" ) );
```

这个代码的输出结果如下：

```
jdk.nashorn.api.scripting.NashornScriptEngine
Result: 2
```

4.5 Base64

对Base64编码的支持已经被加入到Java 8官方库中，这样不需要使用第三方库就可以进行Base64编，例子代码如下：

```
package com.javacodegeeks.java8.base64;

import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class Base64s {
    public static void main(String[] args) {
```

```
        final String text = "Base64 finally in Java 8!";

        final String encoded = Base64
            .getEncoder()
            .encodeToString( text.getBytes( StandardCharsets.UTF_8 ) );
        System.out.println( encoded );

        final String decoded = new String(
            Base64.getDecoder().decode( encoded ),
            StandardCharsets.UTF_8 );
        System.out.println( decoded );
    }
}
```

这个例子的输出结果如下：

```
QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==
Base64 finally in Java 8!
```

新的Base64API也支持URL和MINE的编码解码。

(**Base64.**getUrlEncoder**() / Base64.**getUrlDecoder**()**, **Base64.**getMimeEncoder**() / Base64.**getMieDecoder**())**。

4.6 并行数组

Java8版本新增了很多新的方法，用于支持并行数组处理。最重要的方法是**parallelSort()**，可以显著快多核机器上的数组排序。下面的例子论证了**parallexXxx**系列的方法：

```
package com.javacodegeeks.java8.parallel.arrays;

import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

public class ParallelArrays {
    public static void main( String[] args ) {
        long[] arrayOfLong = new long [ 20000 ];

        Arrays.parallelSetAll( arrayOfLong,
            index -> ThreadLocalRandom.current().nextInt( 1000000 ) );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();

        Arrays.parallelSort( arrayOfLong );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();
    }
}
```

上述这些代码使用**parallelSetAll()**方法生成20000个随机数，然后使用**parallelSort()**方法进行排。这个程序会输出乱序数组和排序数组的前10个元素。上述例子的代码输出的结果是：

```
Unsorted: 591217 891976 443951 424479 766825 351964 242997 642839 119108 552378
Sorted: 39 220 263 268 325 607 655 678 723 793
```

4.7 并发性

基于新增的lambda表达式和steam特性，为Java 8中为**java.util.concurrent.ConcurrentHashMap**添加了新的方法来支持聚焦操作；另外，也为**java.util.concurrentForkJoinPool**类添加了新的方法支持通用线程池操作（更多内容可以参考我们的并发编程课程）。

Java 8还添加了新的**java.util.concurrent.locks.StampedLock**类，用于支持基于容量的锁——该有三个模型用于支持读写操作（可以把这个锁当做是**java.util.concurrent.locks.ReadWriteLock**替代者）。

在**java.util.concurrent.atomic**包中也新增了不少工具类，列举如下：

- DoubleAccumulator
- DoubleAdder
- LongAccumulator
- LongAdder

5. 新的Java工具

Java 8提供了一些新的命令行工具，这部分会讲解一些对开发者最有用的工具。

5.1 Nashorn引擎：jjs

jjs是一个基于标准Nashorn引擎的命令行工具，可以接受js源码并执行。例如，我们写一个**func.js**文，内容如下：

```
function f() {
    return 1;
};

print( f() + 1 );
```

可以在命令行中执行这个命令：jjs func.js，控制台输出结果是：

```
2
```

如果需要了解细节，可以参考官方文档。

5.2 类依赖分析器：jdeps

jdeps是一个相当棒的命令行工具，它可以展示包层级和类层级的Java类依赖关系，它以**.class**文、目录或者Jar文件为输入，然后会把依赖关系输出到控制台。

我们可以利用jedps分析下Spring Framework库，为了让结果少一点，仅仅分析一个JAR文件：**org.springframework.core-3.0.5.RELEASE.jar**。

```
jdeps org.springframework.core-3.0.5.RELEASE.jar
```

这个命令会输出很多结果，我们仅看下其中的一部分：依赖关系按照包分组，如果在classpath上找到依赖，则显示"not found".

```
org.springframework.core-3.0.5.RELEASE.jar -> C:\Program Files\Java\jdk1.8.0\jre\lib\rt.jar
   org.springframework.core (org.springframework.core-3.0.5.RELEASE.jar)
      -> java.io
      -> java.lang
      -> java.lang.annotation
      -> java.lang.ref
      -> java.lang.reflect
      -> java.util
      -> java.util.concurrent
      -> org.apache.commons.logging                 not found
      -> org.springframework.asm                    not found
      -> org.springframework.asm.commons            not found
   org.springframework.core.annotation (org.springframework.core-3.0.5.RELEASE.jar)
      -> java.lang
      -> java.lang.annotation
      -> java.lang.reflect
      -> java.util
```

更多的细节可以参考官方文档。

6. JVM的新特性

使用**Metaspace（**JEP 122**）**代替持久代（PermGen space）。在JVM参数方面使用**-XX:MetaSpaceSize**和**-XX:MaxMetaspaceSize**代替原来的**-XX:PermSize**和**-XX:MaxPermSize*。

7. 结论

通过为开发者提供很多能够提高生产力的特性，Java 8使得Java平台前进了一大步。现在还不太适合将ava 8应用在生产系统中，但是在之后的几个月中Java 8的应用率一定会逐步提高（PS:原文时间是201年5月9日，现在在很多公司Java 8已经成为主流，我司由于体量太大，现在也在一点点上Java 8，虽慢但是好歹在升级了）。作为开发者，现在应该学习一些Java 8的知识，为升级做好准备。

关于Spring：对于企业级开发，我们也应该关注Spring社区对Java 8的支持，可以参考这篇文章——pring 4支持的Java 8新特性一览

8. 参考资料

- What’ s New in JDK 8
- The Java Tutorials
- WildFly 8, JDK 8, NetBeans 8, Java EE
- Java 8 Tutorial
- JDK 8 Command-line Static Dependency Checker
- The Illuminating Javadoc of JDK
- The Dark Side of Java 8
- Installing Java™tm 8 Support in Eclipse Kepler SR2

- Java 8
- Oracle Nashorn. A Next-Generation JavaScript Engine for the JVM

作者：杜琪
链接：http://www.jianshu.com/p/5b800057f2d8
來源：简书