**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上已经有很多介绍Java 8新特性的文章，例如[Playing with Java 8 – Lambdas and Concurrency](http://www.javacodegeeks.com/2014/04/playing-with-java-8-lambdas-and-concurrency.html)、[Java 8 Date Time API Tutorial : LocalDateTime](http://www.javacodegeeks.com/2014/04/java-8-date-time-api-tutorial-localdatetime.html)和[Abstract Class Versus Interface in the JDK 8 Era](http://www.javacodegeeks.com/2014/04/abstract-class-versus-interface-in-the-jdk-8-era.html)。本文还参考了一些其他资料，例如：[15 Must Read Java 8 Tutorials](http://www.javacodegeeks.com/2014/04/15-must-read-java-8-tutorials.html)和[The Dark Side of Java 8](http://www.javacodegeeks.com/2014/04/java-8-friday-the-dark-side-of-java-8.html)。本文综合了上述资料，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](http://www.oracle.com/technetwork/java/javase/8train-relnotes-latest-2153846.html)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、编译器、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什么场景下适合使用。

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，但同时也需要花不少精力去学习。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数传递给某个方法，或者把代码本身当作数据处理：[函数式开发者](http://www.javacodegeeks.com/2014/03/functional-programming-with-java-8-lambda-expressions-monads.html)非常熟悉这些概念。很多JVM平台上的语言（Groovy、[Scala](http://www.javacodegeeks.com/tutorials/scala-tutorials/)等）从诞生之日就支持Lambda表达式，但是Java开发者没有选择，只能使用匿名内部类代替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，可以实现简洁而紧凑的语言结构。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中的函数体，例如：

1. Arrays.asList( "a", "b", "d" ).forEach( e -> {
2. System.out.print( e );
3. System.out.print( e );
4. } );

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），例如下列两个代码块的效果完全相同：

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

和

1. final String separator = ",";
2. Arrays.asList( "a", "b", "d" ).forEach(
3. ( 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一行，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

1. Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> {
2. int result = e1.compareTo( e2 );
3. return result;
4. } );

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[**函数接口**](http://www.javacodegeeks.com/2013/03/introduction-to-functional-interfaces-a-concept-recreated-in-java-8.html)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。**java.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式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 8 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：

1. @FunctionalInterface
2. public interface Functional {
3. void method();
4. }

不过有一点需要注意，[默认方法和静态方法](https://www.javacodegeeks.com/2014/05/java-8-features-tutorial.html#Interface_Default)不会破坏函数式接口的定义，因此如下的代码是合法的。

1. @FunctionalInterface
2. public interface FunctionalDefaultMethods {
3. void method();
5. default void defaultMethod() {
6. }
7. }

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编程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](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)。

**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](http://blog.csdn.net/yczz/article/details/50896975)使得接口有点类似traits，不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，往现存接口中添加新的方法，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被接口的实现类继承或者覆写，例子代码如下：

1. private interface Defaulable {
2. *// Interfaces now allow default methods, the implementer may or*
3. *// may not implement (override) them.*
4. default String notRequired() {
5. return "Default implementation";
6. }
7. }
9. private static class DefaultableImpl implements Defaulable {
10. }
12. private static class OverridableImpl implements Defaulable {
13. @Override
14. public String notRequired() {
15. return "Overridden implementation";
16. }
17. }

**Defaulable**接口使用关键字**default**定义了一个默认方法**notRequired()**。**DefaultableImpl**类实现了这个接口，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，但覆写了该接口的默认方法，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：

1. private interface DefaulableFactory {
2. *// Interfaces now allow static methods*
3. static Defaulable create( Supplier< Defaulable > supplier ) {
4. return supplier.get();
5. }
6. }

下面的代码片段整合了默认方法和静态方法的使用场景：

1. public static void main( String[] args ) {
2. Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new );
3. System.out.println( defaulable.notRequired() );
5. defaulable = DefaulableFactory.create( OverridableImpl::new );
6. System.out.println( defaulable.notRequired() );
7. }

这段代码的输出结果如下：

1. Default implementation
2. 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有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**stream()**、**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，默认方法可能引起歧义和编译错误。如果你想了解更多细节，可以参考[官方文档](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html)。

**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Lambda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。

1. public static class Car {
2. public static Car create( final Supplier< Car > supplier ) {
3. return supplier.get();
4. }
6. public static void collide( final Car car ) {
7. System.out.println( "Collided " + car.toString() );
8. }
10. public void follow( final Car another ) {
11. System.out.println( "Following the " + another.toString() );
12. }
14. public void repair() {
15. System.out.println( "Repaired " + this.toString() );
16. }
17. }

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<T>::new**。注意：这个构造器没有参数。

1. final Car car = Car.create( Car::new );
2. 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Car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，这个方法没有定义入参：

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：这个方法接受一个Car类型的参数：

1. final Car police = Car.create( Car::new );
2. 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）：

1. Collided com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
2. Repaired com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
3. Following the com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d

如果想了解和学习更详细的内容，可以参考[官方文档](http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html)

**2.4 重复注解**

自从Java 5中引入[注解](http://www.javacodegeeks.com/2012/08/java-annotations-explored-explained.html)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，引入了重复注解的概念，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，而是编译器做的一个trick，底层的技术仍然相同。可以利用下面的代码说明：

1. package com.javacodegeeks.java8.repeatable.annotations;
3. import java.lang.annotation.ElementType;
4. import java.lang.annotation.Repeatable;
5. import java.lang.annotation.Retention;
6. import java.lang.annotation.RetentionPolicy;
7. import java.lang.annotation.Target;
9. public class RepeatingAnnotations {
10. @Target( ElementType.TYPE )
11. @Retention( RetentionPolicy.RUNTIME )
12. public @interface Filters {
13. Filter[] value();
14. }
16. @Target( ElementType.TYPE )
17. @Retention( RetentionPolicy.RUNTIME )
18. @Repeatable( Filters.class )
19. public @interface Filter {
20. String value();
21. };
23. @Filter( "filter1" )
24. @Filter( "filter2" )
25. public interface Filterable {
26. }
28. public static void main(String[] args) {
29. for( Filter filter: Filterable.class.getAnnotationsByType( Filter.class ) ) {
30. System.out.println( filter.value() );
31. }
32. }
33. }

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**注解的容器，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（这里并没有提到任何关于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，例如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：

1. filter1
2. filter2

如果你希望了解更多内容，可以参考[官方文档](http://docs.oracle.com/javase/tutorial/java/annotations/repeating.html)。

**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，从而使得代码更为简洁。例子代码如下：

1. package com.javacodegeeks.java8.type.inference;
3. public class Value< T > {
4. public static< T > T defaultValue() {
5. return null;
6. }
8. public T getOrDefault( T value, T defaultValue ) {
9. return ( value != null ) ? value : defaultValue;
10. }
11. }

下列代码是**Value<String>**类型的应用：

1. package com.javacodegeeks.java8.type.inference;
3. public class TypeInference {
4. public static void main(String[] args) {
5. final Value< String > value = new Value<>();
6. value.getOrDefault( "22", Value.defaultValue() );
7. }
8. }

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中这段代码会有编译错误，除非使用Value.<String>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、超类和接口实现类，甚至可以用在函数的异常定义上。下面是一些例子：

1. package com.javacodegeeks.java8.annotations;
3. import java.lang.annotation.ElementType;
4. import java.lang.annotation.Retention;
5. import java.lang.annotation.RetentionPolicy;
6. import java.lang.annotation.Target;
7. import java.util.ArrayList;
8. import java.util.Collection;
10. public class Annotations {
11. @Retention( RetentionPolicy.RUNTIME )
12. @Target( { ElementType.TYPE\_USE, ElementType.TYPE\_PARAMETER } )
13. public @interface NonEmpty {
14. }
16. public static class Holder< @NonEmpty T > extends @NonEmpty Object {
17. public void method() throws @NonEmpty Exception {
18. }
19. }
21. @SuppressWarnings( "unused" )
22. public static void main(String[] args) {
23. final Holder< String > holder = new @NonEmpty Holder< String >();
24. @NonEmpty Collection< @NonEmpty String > strings = new ArrayList<>();
25. }
26. }

**ElementType.TYPE\_USER**和**ElementType.TYPE\_PARAMETER**是Java 8新增的两个注解，用于描述注解的使用场景。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[Paranamer liberary](https://github.com/paul-hammant/paranamer)。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。

1. package com.javacodegeeks.java8.parameter.names;
3. import java.lang.reflect.Method;
4. import java.lang.reflect.Parameter;
6. public class ParameterNames {
7. public static void main(String[] args) throws Exception {
8. Method method = ParameterNames.class.getMethod( "main", String[].class );
9. for( final Parameter parameter: method.getParameters() ) {
10. System.out.println( "Parameter: " + parameter.getName() );
11. }
12. }
13. }

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，则会输出如下结果：

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：

1. <plugin>
2. <groupId>org.apache.maven.plugins</groupId>
3. <artifactId>maven-compiler-plugin</artifactId>
4. <version>3.1</version>
5. <configuration>
6. <compilerArgument>-parameters</compilerArgument>
7. <source>1.8</source>
8. <target>1.8</target>
9. </configuration>
10. </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、函数式编程等。

**4.1 Optional**

Java应用中最常见的bug就是[空值异常](http://examples.javacodegeeks.com/java-basics/exceptions/java-lang-nullpointerexception-how-to-handle-null-pointer-exception/)。在Java 8之前，[Google Guava](http://code.google.com/p/guava-libraries/)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也将**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检查，可以参考[Java 8官方文档](http://docs.oracle.com/javase/8/docs/api/)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：

1. Optional< String > fullName = Optional.ofNullable( null );
2. System.out.println( "Full Name is set? " + fullName.isPresent() );
3. System.out.println( "Full Name: " + fullName.orElseGet( () -> "[none]" ) );
4. 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()**方法可以将现有的**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的时候返回传入的默认值。

上述代码的输出结果如下：

1. Full Name is set? false
2. Full Name: [none]
3. Hey Stranger!

再看下另一个简单的例子：

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

这个例子的输出是：

1. First Name is set? true
2. First Name: Tom
3. Hey Tom!

如果想了解更多的细节，请参考[官方文档](http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)。

**4.2 Streams**

新增的[Stream API](http://www.javacodegeeks.com/2014/05/the-effects-of-programming-with-java-8-streams-on-algorithm-performance.html)（java.util.stream）将生成环境的函数式编程引入了Java库中。这是目前为止最大的一次对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。

Steam 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：

1. public class Streams {
2. private enum Status {
3. OPEN, CLOSED
4. };
6. private static final class Task {
7. private final Status status;
8. private final Integer points;
10. Task( final Status status, final Integer points ) {
11. this.status = status;
12. this.points = points;
13. }
15. public Integer getPoints() {
16. return points;
17. }
19. public Status getStatus() {
20. return status;
21. }
23. @Override
24. public String toString() {
25. return String.format( "[%s, %d]", status, points );
26. }
27. }
28. }

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task集合：

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

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表，并且支持顺序和并行处理。

1. *// Calculate total points of all active tasks using sum()*
2. final long totalPointsOfOpenTasks = tasks
3. .stream()
4. .filter( task -> task.getStatus() == Status.OPEN )
5. .mapToInt( Task::getPoints )
6. .sum();
8. 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**操作会过滤掉所有CLOSED的task；第三，**mapToInt**操作基于每个task实例的**Task::getPoints**方法将task流转换成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](http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamOps)）的知识点。Steam之上的操作可分为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，而是创建一个新的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，steam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，我们可以用下面的代码计算所有任务的点数之和：

1. *// Calculate total points of all tasks*
2. final double totalPoints = tasks
3. .stream()
4. .parallel()
5. .map( task -> task.getPoints() ) *// or map( Task::getPoints )*
6. .reduce( 0, Integer::sum );
8. 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。控制台输出如下：

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可以很快完成这类任务，代码如下：

1. *// Group tasks by their status*
2. final Map< Status, List< Task > > map = tasks
3. .stream()
4. .collect( Collectors.groupingBy( Task::getStatus ) );
5. 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，具体处理的代码如下：

1. *// Calculate the weight of each tasks (as percent of total points)*
2. final Collection< String > result = tasks
3. .stream() *// Stream< String >*
4. .mapToInt( Task::getPoints ) *// IntStream*
5. .asLongStream() *// LongStream*
6. .mapToDouble( points -> points / totalPoints ) *// DoubleStream*
7. .boxed() *// Stream< Double >*
8. .mapToLong( weigth -> ( long )( weigth \* 100 ) ) *// LongStream*
9. .mapToObj( percentage -> percentage + "%" ) *// Stream< String>*
10. .collect( Collectors.toList() ); *// List< String >*
12. 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行得读取数据）可以受益于steam处理，这里有一个小例子：

1. final Path path = new File( filename ).toPath();
2. try( Stream< String > lines = Files.lines( path, StandardCharsets.UTF\_8 ) ) {
3. lines.onClose( () -> System.out.println("Done!") ).forEach( System.out::println );
4. }

Stream的方法**onClose** 返回一个等价的有额外句柄的Stream，当Stream的close（）方法被调用的时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，是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)](https://jcp.org/en/jsr/detail?id=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（甚至令开发者更加迷茫）。

因为上面这些原因，诞生了第三方库[Joda-Time](http://www.joda.org/joda-time/)，可以替代Java的时间管理API。Java 8中新的时间和日期管理API深受Joda-Time影响，并吸收了很多Joda-Time的精华。新的java.time包包含了所有关于日期、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改，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秒时间和日期。**Clock**可以替代**System.currentTimeMillis()**和**TimeZone.getDefault()**。

1. *// Get the system clock as UTC offset*
2. final Clock clock = Clock.systemUTC();
3. System.out.println( clock.instant() );
4. System.out.println( clock.millis() );

这个例子的输出结果是：

1. 2014-04-12T15:19:29.282Z
2. 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。

1. // Get the local date and local time
2. final LocalDate date = LocalDate.now();
3. final LocalDate dateFromClock = LocalDate.now( clock );
5. System.out.println( date );
6. System.out.println( dateFromClock );
8. // Get the local date and local time
9. final LocalTime time = LocalTime.now();
10. final LocalTime timeFromClock = LocalTime.now( clock );
12. System.out.println( time );
13. System.out.println( timeFromClock );

上述例子的输出结果如下：

1. 2014-04-12
2. 2014-04-12
3. 11:25:54.568
4. 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](https://www.javacodegeeks.com/2014/04/java-8-date-time-api-tutorial-localdatetime.html)：

1. *// Get the local date/time*
2. final LocalDateTime datetime = LocalDateTime.now();
3. final LocalDateTime datetimeFromClock = LocalDateTime.now( clock );
5. System.out.println( datetime );
6. System.out.println( datetimeFromClock );

上述这个例子的输出结果如下：

1. 2014-04-12T11:37:52.309
2. 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日期系统的日期和时间，而且有时区信息。下面是一些使用不同时区的例子：

1. *// Get the zoned date/time*
2. final ZonedDateTime zonedDatetime = ZonedDateTime.now();
3. final ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now( clock );
4. final ZonedDateTime zonedDatetimeFromZone = ZonedDateTime.now( ZoneId.of( "America/Los\_Angeles" ) );
6. System.out.println( zonedDatetime );
7. System.out.println( zonedDatetimeFromClock );
8. System.out.println( zonedDatetimeFromZone );

这个例子的输出结果是：

1. 2014-04-12T11:47:01.017-04:00[America/New\_York]
2. 2014-04-12T15:47:01.017Z
3. 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的不同，例子代码如下：

1. *// Get duration between two dates*
2. final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL, 16, 0, 0, 0 );
3. final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16, 23, 59, 59 );
5. final Duration duration = Duration.between( from, to );
6. System.out.println( "Duration in days: " + duration.toDays() );
7. 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：

1. Duration in days: 365
2. 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，另一部分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](http://docs.oracle.com/javase/tutorial/datetime/index.html)。

**4.4 Nashorn JavaScript引擎**

Java 8提供了新的[Nashorn JavaScript引擎](http://www.javacodegeeks.com/2014/02/java-8-compiling-lambda-expressions-in-the-new-nashorn-js-engine.html)，使得我们可以在JVM上开发和运行JS应用。Nashorn JavaScript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Java和JavaScript交互使用，例子代码如下：

1. ScriptEngineManager manager = new ScriptEngineManager();
2. ScriptEngine engine = manager.getEngineByName( "JavaScript" );
4. System.out.println( engine.getClass().getName() );
5. System.out.println( "Result:" + engine.eval( "function f() { return 1; }; f() + 1;" ) );

这个代码的输出结果如下：

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

**4.5 Base64**

[对Base64编码的支持](http://www.javacodegeeks.com/2014/04/base64-in-java-8-its-not-too-late-to-join-in-the-fun.html)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编码，例子代码如下：

1. package com.javacodegeeks.java8.base64;
3. import java.nio.charset.StandardCharsets;
4. import java.util.Base64;
6. public class Base64s {
7. public static void main(String[] args) {
8. final String text = "Base64 finally in Java 8!";
10. final String encoded = Base64
11. .getEncoder()
12. .encodeToString( text.getBytes( StandardCharsets.UTF\_8 ) );
13. System.out.println( encoded );
15. final String decoded = new String(
16. Base64.getDecoder().decode( encoded ),
17. StandardCharsets.UTF\_8 );
18. System.out.println( decoded );
19. }
20. }

这个例子的输出结果如下：

1. QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==
2. 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.*getMimeDecoder*()**)。

**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()**，可以显著加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：

1. package com.javacodegeeks.java8.parallel.arrays;
3. import java.util.Arrays;
4. import java.util.concurrent.ThreadLocalRandom;
6. public class ParallelArrays {
7. public static void main( String[] args ) {
8. long[] arrayOfLong = new long [ 20000 ];
10. Arrays.parallelSetAll( arrayOfLong,
11. index -> ThreadLocalRandom.current().nextInt( 1000000 ) );
12. Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
13. i -> System.out.print( i + " " ) );
14. System.out.println();
16. Arrays.parallelSort( arrayOfLong );
17. Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
18. i -> System.out.print( i + " " ) );
19. System.out.println();
20. }
21. }

上述这些代码使用**parallelSetAll()**方法生成20000个随机数，然后使用**parallelSort()**方法进行排序。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：

1. Unsorted: 591217 891976 443951 424479 766825 351964 242997 642839 119108 552378
2. 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**类添加了新的方法来支持聚焦操作；另外，也为**java.util.concurrentForkJoinPool**类添加了新的方法来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](http://academy.javacodegeeks.com/course/java-concurrency-essentials/)）。

Java 8还添加了新的**java.util.concurrent.locks.StampedLock**类，用于支持基于容量的锁——该锁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**的替代者）。

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**文件，内容如下：

1. function f() {
2. return 1;
3. };
5. 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](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jjs.html)。

**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**文件、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](http://blog.csdn.net/yczz/article/details/50896975)，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上找不到依赖，则显示"not found".

1. org.springframework.core-3.0.5.RELEASE.jar -> C:\Program Files\Java\jdk1.8.0\jre\lib\rt.jar
2. org.springframework.core (org.springframework.core-3.0.5.RELEASE.jar)
3. -> java.io
4. -> java.lang
5. -> java.lang.annotation
6. -> java.lang.ref
7. -> java.lang.reflect
8. -> java.util
9. -> java.util.concurrent
10. -> org.apache.commons.logging not found
11. -> org.springframework.asm not found
12. -> org.springframework.asm.commons not found
13. org.springframework.core.annotation (org.springframework.core-3.0.5.RELEASE.jar)
14. -> java.lang
15. -> java.lang.annotation
16. -> java.lang.reflect
17. -> java.util

更多的细节可以参考[官方文档](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html)。

**6. JVM的新特性**

使用[**Metaspace**](http://www.javacodegeeks.com/2013/02/java-8-from-permgen-to-metaspace.html)（[JEP 122](http://openjdk.java.net/jeps/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Java 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2014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，虽然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——[Spring 4支持的Java 8新特性一览](http://www.infoq.com/cn/articles/spring-4-java-8)