**Java8新特性学习**

1. Lambda表达式和函数式接口
   1. lambda表达式的使用示例

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

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

c. String separator = ",";

Arrays.asList( "a", "b", "d" ).forEach(

    ( String e ) -> System.out.print( e + separator ));

* 1. 官方文档说明
     1. 准备一个Person类，后续的代码会在这个代码上进行展示

public class Person {

public enum Sex {

MALE, FEMALE

}

String name;

LocalDate birthday;

Sex gender;

String emailAddress;

public int getAge() {

// ...

}

public void printPerson() {

// ...

}

}

* + 1. 使用for each 迭代器遍历集合，对满足要求的对象进行打印

Code 1：

public static void printPersonsOlderThan(List<Person> roster, int age) {

for (Person p : roster) {

if (p.getAge() >= age) {

p.printPerson();

}

}

}

上述代码在遇到Person类的变量修改时，会使得需要对Person类中的方法进行修改。

* + 1. 更常见的代码：

Code 2:

public static void printPersonsWithinAgeRange(

List<Person> roster, int low, int high) {

for (Person p : roster) {

if (low <= p.getAge() && p.getAge() < high) {

p.printPerson();

}

}

}

当我们需要其他的比较方式就需要写更多的比较方法。

* + 1. 我们现在对代码进行拆分创建一个接口用于对Person进行比较

interface CheckPerson {

boolean test(Person p);

}

class CheckPersonEligibleForSelectiveService implements CheckPerson {

public boolean test(Person p) {

return p.gender == Person.Sex.MALE &&

p.getAge() >= 18 &&

p.getAge() <= 25;

}

}

public static void printPersons(

List<Person> roster, CheckPerson tester) {

for (Person p : roster) {

if (tester.test(p)) {

p.printPerson();

}

}

}

调用方法

printPersons(

roster, new CheckPersonEligibleForSelectiveService());

使用这样的方法可以在修改Person类的之后在Service中去修改方法就行了。但是如果有不同的比较方法，就需要增加接口方法或者增加实现类，对代码质量没有太大的提升。

* + 1. 使用匿名内部类

在调用方法的时候，实现自己需要的接口。

printPersons(

roster,

new CheckPerson() {

public boolean test(Person p) {

return p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25;

}

}

);

这在Java8之前是非常常用的方法，特别是在写GUI编程或者网络编程的时候，使得代码量特别大，不易阅读

* + 1. 使用Lambda 表达式调用接口

printPersons(

roster,

**(Person p) -> p.getGender() == Person.Sex.MALE**

**&& p.getAge() >= 18**

**&& p.getAge() <= 25**

);

是不是很简洁!

* + 1. 使用标准的函数式接口，使用泛型

interface Predicate<T> {

boolean test(T t);

}

public static void printPersonsWithPredicate(

List<Person> roster, Predicate<Person> tester) {

for (Person p : roster) {

if (tester.test(p)) {

p.printPerson();

}

}

}

printPersonsWithPredicate(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25

);

更多的Lambda知识推荐查看官方文档 <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach1>

推荐书籍 《Scala 函数式编程》

1. 接口的默认方法与静态方法
   1. 默认方法

默认方法与抽象方法不同之处在于抽象方法必须要求实现，但是默认方法则没有这个要求。相反，每个接口都必须提供一个所谓的默认实现，这样所有的接口实现者将会默认继承它（如果有必要的话，可以覆盖这个默认实现）

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";

    }

}

* 1. 静态方法

private interface DefaulableFactory {

    // Interfaces now allow static methods

    static Defaulable create( Supplier< Defaulable > supplier ) {

        return supplier.get();

    }

}

* 1. 测试

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() );

}

结果：

Default implementation

Overridden implementation

1. 方法引用

public 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());  
 }  
 public static void main(String[] args) {  
 //引用构造函数  
 Car car = Car.create(Car::new);  
 List<Car> cars = Arrays.asList(car);  
 //对静态方法的引用  
 cars.forEach(Car::collide);  
 //对特定类的方法进行引用  
 cars.forEach(Car::repair);  
 Car other = Car.create(Car::new);  
 //特定对象方法的引用  
 cars.forEach(other::follow);  
 }  
}

1. 重复注解

重复注解机制本身必须用@Repeatable注解

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() );

        }

    }

}

1. 更好的类型推测机制

public class Value< T > {

    public static< T > T defaultValue() {

        return null;

    }

    public T getOrDefault( T value, T defaultValue ) {

        return ( value != null ) ? value : defaultValue;

    }

}

public class TypeInference {

    public static void main(String[] args) {

        final Value< String > value = new Value<>();

        value.getOrDefault( "22", Value.defaultValue() );

    }

}

Value.defaultValue()的参数类型可以被推测出，所以就不必明确给出。在Java 7中，相同的例子将不会通过编译，正确的书写方式是 Value.<String >defaultValue()

1. 扩展注解的支持

Java 8扩展了注解的上下文。现在几乎可以为任何东西添加注解：局部变量、泛型类、父类与接口的实现，就连方法的异常也能添加注解

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<>();

    }

}

1. 编译器新特性
2. 库类新特性
   1. 映入Optional 类来优化代码。
   2. 引入Stream 使代码更加简洁明了，（不过有社区验证，stream的效率不如foreach的使用效率（代码在最后贴）

优点：代码简洁明了，函数式编程，不会影响到原有数据，原生支持并行处理

1. Date/Time API （不是很喜欢）
2. Base64 编码成为标准的库类

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 );

    }

}

1. 并行（parallel）数组

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();

    }

}

1. 并发（Concurrency）

在新增Stream机制与lambda的基础之上，在java.util.concurrent.ConcurrentHashMap中加入了一些新方法来支持聚集操作。同时也在java.util.concurrent.ForkJoinPool类中加入了一些新方法来支持共有资源池（common pool）

1. Java虚拟机（JVM）的新特性

PermGen空间[被移除了，取而代之的是Metaspace](http://www.javacodegeeks.com/2013/02/java-8-from-permgen-to-metaspace.html)（[JEP 122](http://openjdk.java.net/jeps/122)）。JVM选项**-XX:PermSize**与**-XX:MaxPermSize**分别被**-XX:MetaSpaceSize**与**-XX:MaxMetaspaceSize**所代替。