Java8 新特性

需要该课程资料同学：关注架构师-余胜军 微信公众号：回复：java8 获取资料

接口中默认方法修饰为普通方法

在 jdk8 之前，interface 之中可以定义变量和方法，变量必须是 public、static、final 的，方法必须是 public、abstract 的，由于这些修饰符都是默认的。

接口定义方法：public 抽象方法 需要子类实现
接口定义变量：public、static、final

在 JDK 1.8 开始 支持使用 static 和 default 修饰 可以写方法体，不需要子类重写。
方法：
普通方法 可以有方法体
抽象方法 没有方法体需要子类实现 重写。

代码案例

```
/**
 * @ClassName JDK8Interface
 * @Author 蚂蚁课堂余胜军 QQ644064779 www.mayikt.com
 * @Version V1.0
 **/
public interface JDK8Interface {
    void addOrder();

    /**
     * 默认方法 可以写方法体
     */
    default void getDefaultOrder() {
        System.out.println("我是默认方法 我可以写方法体");
    }
```

```
    static void getStaticOrder() {
        System.out.println("我是静态的方法 可以写方法体");
    }
}

/**
 * @ClassName JDK8InterfaceImpl
 * @Author 蚂蚁课堂余胜军 QQ644064779 www.mayikt.com
 * @Version V1.0
 **/
public class JDK8InterfaceImpl implements JDK8Interface {
    /**
     * 默认和静态方法不是我们的抽象方法 ， 所以不需要重写
     */
    @Override
    public void addOrder() {
        System.out.println("addOrder");
    }
}
```

Lambda 表达式

什么是 Lambda 表达式

LAMBADA 好处： 简化我们匿名内部类的调用。
Lambda+方法引入 代码变得更加精简。

Lambda 表达式（lambda expression）是一个匿名函数，简化我们调用匿名函数的过程。
百度百科介绍：
https://baike.baidu.com/item/Lambda%E8%A1%A8%E8%BE%BE%E5%BC%8F/4585794?fr=aladdin

为什么要使用 Lambda 表达式

可以非常简洁的形式调用我们的匿名函数接口。

```
public static void main(String[] args) {
    // 1.使用 new 的实现类的形式调用接口
    OrderService orderSerivce1 = new OrderSerivceImpl();
    orderSerivce1.addOrder();
    // 2.使用匿名内部接口调用
    new OrderService() {
        @Override
        public void addOrder() {
            System.out.println("使用匿名内部类的形式调用接口
");
        }
    }.addOrder();
    // 3.使用 Lambda 调用接口
    OrderService orderSerivce2=  ()-> System.out.println("
使用 lambda 调用接口");
    orderSerivce2.addOrder();
}
```

Lambda 表达式的规范

使用 Lambda 表达式 依赖于函数接口

1. 在接口中只能够允许有一个抽象方法
2. 在函数接口中定义 object 类中方法
3. 使用默认或者静态方法
4. @FunctionalInterface 表示该接口为函数接口

Java 中使用 Lambda 表达式的规范，必须是为函数接口
函数接口的定义：在该接口中只能存在一个抽象方法，该接口称作为函数接口

Java 中的 Lambda 表达式的规范，必须是为函数接口。
函数接口的定义：在该接口中只能存在一个抽象方法，该接口称作为函数接口

JDK 中自带的函数接口：
java.lang.Runnable
java.util.concurrent.Callable
java.security.PrivilegedAction
java.util.Comparator
java.io.FileFilter
java.nio.file.PathMatcher
java.lang.reflect.InvocationHandler
java.beans.PropertyChangeListener
java.awt.event.ActionListener
javax.swing.event.ChangeListener

我们也可以使用@FunctionalInterface 修饰为函数接口

函数接口定义

1. 在接口中只能有一个抽象方法
2. @FunctionalInterface 标记为该接口为函数接口
3. 可以通过 default 修饰为普通方法
4. 可以定义 object 类中的方法

```
@FunctionalInterface
public interface MyFunctionalInterface {
    void add();

    default void get() {

    }

    String toString();
}
```

Java 系统内置那些函数接口

消费型接口：
Conusmer<T>
    void accept(T t);
BiConusmer<T,U>
    void accept(T t,U u);//增加一种入参类型

供给型接口
Supplier<T>
    void get();

函数型接口
Function<T ,R>
    R apply(T t);
UnaryOperator<T>
    T apply(T t);//入参与返回值类型一致
BiFunction <T ,U,R>
    R apply(T t,U u);//增加一个参数类型
BinaryOperator<T>
    T apply(T t1,T t2);//l 两个相同类型入参与同类型返回值
ToIntFunction<T>//限定返回 int
ToLongFunction<T>//限定返回 long
ToDoubleFunction<T>//限定返回 double
IntFunction<R>//限定入参 int,返回泛型 R
LongFunction<R>//限定入参 long,返回泛型 R
DoubleFunction<R>//限定入参 double,返回泛型 R

断言型接口
Predicate<T>
    boolean test(T t);

Lambda 基础语法

（）---参数列表
-> 分隔
{} 方法体
(a,b)->{
}
无参方法调用
带参数方法调用

(函数接口的参数列表 不需要写类型 需要定义参数名称)->{方法体}

():函数方法参数列表
->分隔 {}方法体
(a,b)->{
Sout(a,b)
}

Lambda 语法：
():参数列表
->分隔
{}:方法体
()->{}

无参方法调用

```
public interface AcanthopanaxInterface {
    void get();
}

AcanthopanaxInterface acanthopanaxInterface = () -> {
    System.out.println("使用 lamdba 表达式调用方法");
};
acanthopanaxInterface.get();
```

带参数和返回值

```
@FunctionalInterface
public interface YouShenInterface {
    String get(int i, int j);
}
/**
 * @ClassName Test04
 * @Author 蚂蚁课堂余胜军 QQ644064779 www.mayikt.com
 * @Version V1.0
```

```
 **/
public class Test04 {
    public static void main(String[] args) {
        // 1.使用匿名内部类调用有参数函数方法
//        String result = new YouShenInterface() {
//            @Override
//            public String get(int i, int j) {
//                return i + "-" + j;
//            }
//        }.get(1, 1);
//        System.out.println(result);
        //2.使用 lamdba 调用有参数函数方法
        YouShenInterface youShenInterface = (i, j) -> {
            System.out.println("mayikt:" + i + "," + j);
            return i + "-" + j;
        };
        System.out.println(youShenInterface.get(1, 1));
    }
}
```

精简语法

```
        // 1.精简写法优化
//        AcanthopanaxInterface acanthopanaxInterface = () -> {
//            System.out.println("mayikt");
//        };
//        acanthopanaxInterface.get();
        // 2.精简改为:  如果方法体中只有一条语句的情况下 可以不需要写{}
        AcanthopanaxInterface acanthopanaxInterface2 = () ->
                System.out.println("mayikt");
        acanthopanaxInterface2.get();
        ;
        //3.如果方法体只有一条 return 的情况下不需要些{} 和 return
//        YouShenInterface youShenInterface = (i, j) -> {
//            return i + "-" + j;
//        };
        // 优化后
```

```
        YouShenInterface youShenInterface2 = (i, j) -> i + "-" + j;
        System.out.println(youShenInterface2.get(1, 2));
```

方法引入

什么是方法引入
方法引入：需要结合 lambda 表达式能够让代码变得更加精简。
1. 匿名内部类使用
2. Lambda 调用匿名内部类
3. 方法引入
方法引入
1. 静态方法引入： 类名::（静态）方法名称
2. 对象方法引入 类名:: 实例方法名称
3. 实例方法引入 new 对象 对象实例::方法引入
4. 构造函数引入 类名::new

需要遵循一个规范：
方法引入 方法参数列表、返回类型与函数接口参数列表与返回类型必须
要保持一致。

Lambda： 匿名内部类使用代码简洁问题。

类型	语法	对应 lambda 表达式
构造器引用	Class::new	(args) -> new 类名(args)
静态方法引用	Class::static_method	(args) -> 类名.static_method(args)
对象方法引用	Class::method	(inst,args) -> 类名.method(args)
实例方法引用	instance::method	(args) -> instance.method(args)

方法引用提供了非常有用的语法，可以直接引用已有的 java 类或对象的方法或构造器。方法引用其实也离不开 Lambda 表达式，
与 lambda 联合使用 ，方法引用可以使语言的构造更加紧凑简洁，减少冗余代码。

方法引用提供非常有用的语法，可以直接引用已有的 java 类或者对象中方法或者构造函数，方法引用需要配合 Lambda 表达式语法一起使用减少代码的冗余性问题。

构造器引入
静态方法引入

对象方法引入
实例方法引入

方法引入规则

方法引入实际上就是 lambda 表达式中直接引入的方法。

必须遵循规范：引入的方法参数列表返回类型必须要和函数接口参数列表、返回类型保持一致。

静态方法引入

```
import com.mayikt.service.MessageInterface;


/**
 * @ClassName MethodReference
 * @Author 蚂蚁课堂余胜军 QQ644064779 www.mayikt.com
 * @Version V1.0
 **/
public class MethodReference {
    public static void main(String[] args) {
        // 1.使用匿名内部类的形式 调用get 方法
//        new MessageInterface() {
//            @Override
//            public void get() {
//                MethodReference.getMethod();
//            }
//        }.get();
        MessageInterface messageInterface2 = () -> {
            MethodReference.getStaticMethod();
        };
        messageInterface2.get();
        // 使用方法引入调用方法 必须满足：方法引入的方法必须和函数接口中的方法参数列表/返回值一定保持一致。
       MessageInterface messageInterface = MethodReference::getStaticMethod;
        messageInterface.get();


    }


    /**
     * 静态方法引入
```

```
    */
    public static void getStaticMethod() {
       System.out.println("我是 getMethod");
    }
}}
@FunctionalInterface
public interface MessageInterface {
    void get();
}
```

对象方法引入

```
public class Test23 {
    public static void main(String[] args) {
        // 1.使用匿名内部类的形式
//        MayiktService mayiktService = new MayiktService()
{
//            @Override
//            public String get(Test23 test23) {
//                return test23.objGet();
//            }
//        };
//        System.out.println(mayiktService.get(new
Test23()));
        // 2.Lambda
//        MayiktService mayiktService = (test23) ->
test23.objGet();
//        System.out.println(mayiktService.get(new
Test23()));
        // 3.方法引入 在这时候我们函数接口 第一个参数传递

test23 返回调用test23.objGet 方法
//        MayiktService mayiktService = Test23::objGet;
//        System.out.println(mayiktService.get(new
Test23()));
        //Test23::objGet;----- (test23) -> test23.objGet();
        //   R apply(T t); T  apply 方法传递的参数类型 :  R

apply 方法返回的类型
```

```
        // 需要将 string 类型字符串获取长度
//          Function<String, Integer> strFunction = (str) -> {
//              return str.length();
//          };
        Function<String, Integer> function2 =
String::length;
        System.out.println(function2.apply("mayikt"));


    }

    public String objGet() {
        return "mayikt";
    }


}
```

实例方法引入

```
/**
 * @ClassName Test009
 * @Author 蚂蚁课堂余胜军 QQ644064779 www.mayikt.com
 * @Version V1.0
 **/
public class Test009 {
    public static void main(String[] args) {
        //1.匿名内部类的写法
        Test009 test009 = new Test009();
//        MessageInterface messageInterface = new MessageInterface() {
//            @Override
//            public void get() {
//                test009.get();
//            }
//        };
//        messageInterface.get();
//        ;
//        MessageInterface messageInterface = () -> {
//            test009.get();
//            ;
```

```
//        };
//        messageInterface.get();
        MessageInterface messageInterface = test009::get;
        messageInterface.get(1);
    }


    public void get(Integer a) {
        System.out.println("方法引入 get 方法:" + a);
    }


@FunctionalInterface
public interface MessageInterface {
    void get(Integer a);
}
```

构造函数引入

```
public class Test011 {
    public static void main(String[] args) {
//        UserInterface userInterface = () -> new
UserEntity();
        UserInterface UserInterface2=  UserEntity::new;;
        UserInterface2.getUser();
    }
}
public class UserEntity {
    private String userName;
    private int age;

    public UserEntity() {

    }
public interface UserInterface {
    UserEntity getUser();
}
```

Lambda 实战案例

Foreach

```
        ArrayList<String> strings = new ArrayList<>();
        strings.add("mayikt");
        strings.add("xiaowei");
        strings.add("xiaomin");
//        strings.forEach(new Consumer() {
//            @Override
//            public void accept(Object o) {
//                System.out.println("o:" + o);
//            }
//        });
        strings.forEach((o) -> {
            System.out.println(o);
        });
```

Lambda 集合排序

```
public class UserEntity {
    private String name;
    private Integer age;

    public UserEntity(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public Integer getAge() {
        return age;
    }

    @Override
```

```
    public String toString() {
        return "UserEntity{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

```
        ArrayList<UserEntity> userlists = new ArrayList<>();
        userlists.add(new UserEntity("mayikt", 22));
        userlists.add(new UserEntity("xiaomin", 18));
        userlists.add(new UserEntity("xiaoha", 36));
//        userlists.sort(new Comparator<UserEntity>() {
//            @Override
//            public int compare(UserEntity o1, UserEntity o2) {
//                return o1.getAge() - o2.getAge();
//            }
//        });
        userlists.sort((o1, o2) ->
                o1.getAge() - o2.getAge()
        );
        userlists.forEach((Consumer) o -> System.out.println("o:" + o.toString()));
```

线程调用

```
new Thread(()-> System.out.println("我是子线程")).start();
```

java 8 stream 流

什么是 stream 流

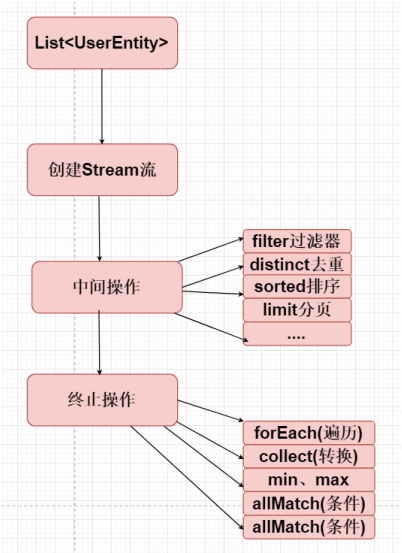
Stream 是 JDK1.8 中处理集合的关键抽象概念，Lambda 和 Stream 是 JDK1.8 新增的函数式编程最有亮点的特性了，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用 Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。Stream 使用一种类似用 SQL 语句从数据库查询数据的直观方式来提供一种对 Java 集合运算和表达的高阶抽象。Stream API 可以极大提高 Java 程序员的生产力，让程

序员写出高效率、干净、简洁的代码。

这种风格将要处理的元素集合看作一种流， 流在管道中传输， 并且可以在管道的节点上进行处理， 比如筛选， 排序，聚合等。

元素流在管道中经过中间操作（intermediate operation）的处理，最后由最终操作(terminal operation)得到前面处理的结果。

Stream ：非常方便精简的形式遍历集合实现 过滤、排序等。
Mysql：select userName from mayikt where userName =‘mayikt’
Order by age limt(0,2)



Stream 创建方式

parallelStream 为并行流采用多线程执行
Stream 采用单线程执行
parallelStream 效率比 Stream 要高。

```
ArrayList<UserEntity> userEntities = new ArrayList<>();
userEntities.add(new UserEntity("mayikt", 20));
userEntities.add(new UserEntity("meite", 28));
userEntities.add(new UserEntity("zhangsan", 35));
userEntities.add(new UserEntity("xiaowei", 16));
```

```
userEntities.add(new UserEntity("xiaowei", 16));

userEntities.stream();
userEntities.parallelStream();
```

Stream 将 list 转换为 Set

```
Stream<UserEntity> stream = userEntities.stream();
//将我们的集合转为Set
Set<UserEntity> collectSet =
stream.collect(Collectors.toSet());
System.out.println(collectSet);
```

Stream 将 list 转换为 Map

```
import com.mayikt.entity.UserEntity;

import java.util.ArrayList;
import java.util.Map;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**
 * @ClassName Test001
 * @Author 蚂蚁课堂余胜军 QQ644064779 www.mayikt.com
 * @Version V1.0
 **/
public class Test001 {
    public static void main(String[] args) {
        ArrayList<UserEntity> userEntities = new ArrayList<UserEntity>();
        userEntities.add(new UserEntity("mayikt", 20));
        userEntities.add(new UserEntity("meite", 28));
        userEntities.add(new UserEntity("zhangsan", 35));
        userEntities.add(new UserEntity("xiaowei", 16));
//        userEntities.add(new UserEntity("xiaowei", 16));
```

```
//        strings.add("xiaowei");
//        strings.add("xiaomin");
//        strings.add("xiaowei");
        /**
         * 创建一个串行的stream 流
         */
        Stream<UserEntity> stream = userEntities.stream();
        // key 为string 类型 value UserEntity  集合中的数据：UserEntity , string 类型
       Map<String, UserEntity> collect = stream.collect(Collectors.toMap(new
Function<UserEntity, String>() {
            @Override
            public String apply(UserEntity userEntity) {
                return userEntity.getUserName();
            }
        }, new Function<UserEntity, UserEntity>() {
            @Override
            public UserEntity apply(UserEntity userEntity) {
                return userEntity;
            }
        }));
        collect.forEach(new BiConsumer<String, UserEntity>() {
            @Override
            public void accept(String s, UserEntity userEntity) {
                System.out.println("s:" + s + ",:" + userEntity.toString());
            }
        });
    }
}
```

Stream 将 Reduce 求和

```
        Stream<Integer> integerStream = Stream.of(10, 30,
80, 60, 10, 70);
//        Optional<Integer> reduce =
integerStream.reduce(new BinaryOperator<Integer>() {
//            @Override
//            public Integer apply(Integer a1, Integer a2) {
//                return a1 + a2;
//            }
//        });
        Optional<Integer> reduce =
```

```
integerStream.reduce((a1, a2) -> a1 + a2);
        System.out.println(reduce.get());
```

```
//        Optional<UserEntity> reduce = stream.reduce(new
BinaryOperator<UserEntity>() {
//            @Override
//            public UserEntity apply(UserEntity
userEntity, UserEntity userEntity2) {
//                userEntity.setAge(userEntity.getAge() +
userEntity2.getAge());
//                return userEntity;
//            }
//        });
        Optional<UserEntity> reduce = stream.reduce((user1,
user2) -> {
            user1.setAge(user1.getAge() + user2.getAge());
            return user1;
        });
```

StreamMax 和 Min

```
//        Optional<UserEntity> max = stream.max(new
Comparator<UserEntity>() {
//            @Override
//            public int compare(UserEntity o1, UserEntity
o2) {
//                return o1.getAge() - o2.getAge();
//            }
//        });
        Optional<UserEntity> max = stream.max((o1, o2) ->
o1.getAge() - o2.getAge());
        System.out.println(max.get());
Optional<UserEntity> min = stream.min(((o1, o2) ->
o1.getAge() - o2.getAge()));
System.out.println(min.get());
```

StreamMatch 匹配

anyMatch 表示，判断的条件里，任意一个元素成功，返回 true
allMatch 表示，判断条件里的元素，所有的都是，返回 true
noneMatch 跟 allMatch 相反，判断条件里的元素，所有的都不是，返回 true

```
//        boolean result = stream.noneMatch(new
Predicate<UserEntity>() {
//            @Override
//            public boolean test(UserEntity userEntity) {
//                return userEntity.getAge() >35;
//            }
//        });
        boolean result = stream.noneMatch((user) ->
user.getAge() > 35);
        System.out.println(result);
```

StreamFor 循环

```
//        stream.forEach(new Consumer<UserEntity>() {
//            @Override
//            public void accept(UserEntity userEntity) {
//
System.out.println(userEntity.toString());
//            }
//        });
        stream.forEach((userEntity ->
System.out.println(userEntity.toString())));
```

Stream 过滤器

```
//        stream.filter(new Predicate<UserEntity>() {
//            @Override
//            public boolean test(UserEntity userEntity) {
//                return userEntity.getAge() >= 35;
//            }
//        }).filter(new Predicate<UserEntity>() {
//            @Override
```

```
//            public boolean test(UserEntity userEntity) {
//                return
userEntity.getUserName().equals("zhangsan");
//            }
//        }).forEach(new Consumer<UserEntity>() {
//            @Override
//            public void accept(UserEntity userEntity) {
//
System.out.println(userEntity.toString());
//            }
//        });
        stream.filter((userEntity -> userEntity.getAge() >=
35)).filter(userEntity -> userEntity.equals("zhangsan"))
                .forEach((userEntity ->
System.out.println(userEntity.toString())));
```

Stream 排序 sorted

```
//        stream.sorted(new Comparator<UserEntity>() {
//            @Override
//            public int compare(UserEntity o1, UserEntity
o2) {
//                return o1.getAge() - o2.getAge();
//            }
//        }).forEach(new Consumer<UserEntity>() {
//            @Override
//            public void accept(UserEntity userEntity) {
//
System.out.println(userEntity.toString());
//            }
//        });
        stream.sorted(((o1, o2) -> o1.getAge() -
o2.getAge())).forEach(userEntity ->
System.out.println(userEntity.toString()));
```

Stream limit 和 skip

Limit 从头开始获取

Skip 就是跳过

```
stream.skip(2).limit(1).forEach(userEntity ->
```

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

Stream 综合案例

```
ArrayList<UserEntity> userEntities = new ArrayList<>();
userEntities.add(new UserEntity("mayikt", 20));
userEntities.add(new UserEntity("meite", 28));
userEntities.add(new UserEntity("zhangsan", 35));
userEntities.add(new UserEntity("xiaowei", 16));
userEntities.add(new UserEntity("mayikt_list", 109));
userEntities.add(new UserEntity("mayikt_zhangsan", 110));
userEntities.add(new UserEntity("lisi", 109));
//要求：对数据流的数据实现降序排列、且名称包含 mayikt 获取前两位
userEntities.stream().filter(userEntity ->
userEntity.getUserName().contains("mayikt_")).limit(2)
        .sorted((o1, o2) -> o1.getAge() - o2.getAge())
        .forEach(userEntity ->
System.out.println(userEntity.toString()));
```

并行流与串行流区别

串行流：单线程的方式操作； 数据量比较少的时候。
并行流：多线程方式操作；数据量比较大的时候，原理：
Fork join 将一个大的任务拆分 n 多个小的子任务并行执行，
最后在统计结果，有可能会非常消耗 cpu 的资源，确实可以
提高效率。

注意：数据量比较少的情况下，不要使用并行流。

JDK8Optional

Optional 类是一个可以为 null 的容器对象。如果值存在则 isPresent()方法会返回 true，调用 get()方法会返回该对象。
Optional 是个容器：它可以保存类型 T 的值，或者仅仅保存 null。Optional 提供很多有用的方法，这样我们就不用显式进行空值检测。

Optional 类的引入很好的解决空指针异常。

判断参数是否为空

ofNullable(可以传递一个空对象)

Of(不可以传递空对象)

```
Integer a1 = 1;
Optional<Integer> a = Optional.ofNullable(a1);
System.out.println(a.isPresent());
```

isPresent true 不为空 isPresent 返回 false 为空。

参数为空可以设定默认值

```
        Integer a1 = 5;
//        Optional<Integer> a = Optional.ofNullable(a1);
//        System.out.println(a.get());
//        System.out.println(a.isPresent());
        Integer a = Optional.ofNullable(a1).orElse(10);
        System.out.println(a);
```

参数实现过滤

```
Integer a1 = 16;
Optional<Integer> a = Optional.ofNullable(a1);
boolean isPresent = a.filter(a2 -> a2 > 17).isPresent();
System.out.println(isPresent);
```

与 Lambda 表达式结合使用，优化代码

优化方案 1

```
        // 优化前
        String mayiktName = "meite";
```

```
        if (mayiktName != null) {
            System.out.println(mayiktName);
        }
        //优化后
        Optional<String> mayiktName2 =
Optional.ofNullable(mayiktName);
//        // 当value 不为空时，则不会调用
//        mayiktName2.ifPresent(s ->
System.out.println(s));
        mayiktName2.ifPresent(System.out::print);
```

优化方案 2

```
    private static OrderEntity order = null;

    public static void main(String[] args) {
        OrderEntity order = Test06.getOrder();
        System.out.println(order);

    }

    public static OrderEntity getOrder() {
//        // 优化前
//        if (order == null) {
//            return createOrder();
//        }
//        return order;
        // 优化后
//        return Optional.ofNullable(order).orElseGet(new
Supplier<OrderEntity>() {
//            @Override
//            public OrderEntity get() {
//                return createOrder();
//            }
//        });
        return Optional.ofNullable(order).orElseGet(() ->
createOrder());
    }
```

```
    private static OrderEntity createOrder() {
        return new OrderEntity("123456", "mayikt");
    }
```

优化方案 3

map 中获取的返回值自动被 Optional 包装,即返回值 -> Optional<返回值>

flatMap 中返回值保持不变,但必须是 Optional 类型,即 Optional<返回值> -> Optional<返回值>

eg:

```
public class Test07 {
    public static void main(String[] args) {
        String orderName = Test07.getOrderName();
        System.out.println(orderName);
    }

    public static String getOrderName() {
        // 优化前写法:
        OrderEntity order = new OrderEntity("123456",
"MAyikt");
        if (order != null) {
            String orderName = order.getOrderName();
            if (orderName != null) {
                return orderName.toLowerCase();
            }
        }
//        return null;
        //优化后写法:
        return Optional.ofNullable(order).map(orderEntity
-> {
            return orderEntity.getOrderName();
        }).map(name -> {
            return name.toLowerCase();
        }).orElse(null);
    }
}
```