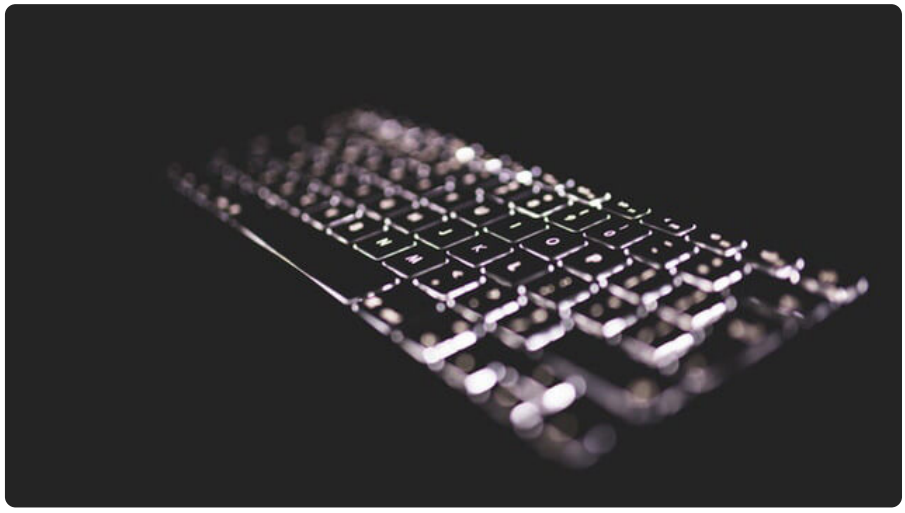
目录

17 如何解决条件语句的多层嵌套问题?

更新时间：2019-12-16 11:30:30



> 富贵必从勤苦得。
>
> ——杜甫

1. 前言

《手册》第 19 页，有关于多 if-else 分支和嵌套的建议和解决方案 1：

> 表达分支时，如果非要使用 if ()…else if ()…else… 方式表达逻辑，避免后续代码维护困难，不允许超过三层。
>
> 如果超过 3 层可以使用卫语句、策略模式、状态模式等来实现。
>
> 其中卫语句代码逻辑优先考虑失败、异常、中断、退出等直接返回的情况。

那么我们要思考以下几个问题：

- 我们该如何将这几种方案落地呢？
- 使用过程中会遇到哪些奇葩的问题呢？

这些都是本节重点研究的问题。

请看下面开发中可能会遇到的典型代码：

```
public double getSalary(Integer position) {
    double result;
    if (position == null) {
```

```
        // 老板
        if (isBoss(position)) {
            result = getBossSalary();
        } else {
            // 领导
            if (isLeader(position)) {
                result = getLeaderSalary();
            } else {
                // 普通员工
                result = getStaffSalary();
            }
        }
        return result;
    }
```

我们如何替代多分支和分支嵌套问题呢？如何让代码变得更容易维护和拓展呢？

请看下面的分析。

2. 卫语句

《重构》 第 9 章 9.5 节 以卫语句取代嵌套条件表达式 中，有如下描述：

> 如果某个条件极其罕见，就应该单独检查该条件，并在条件为真时立即从函数中返回。这样的单独检查常常被称为 “卫语句”。
>
> 卫语句要不就从函数中返回，要不就抛出一个异常。

使用卫语句，我们可以对上面的示例修改为：

```
public double getSalaryGuard(Integer position) {

    // 条件检查
    if (position == null) {
        throw new IllegalArgumentException("职位不能为空");
    }
    // 老板
    if (isBoss(position)) {
        return getBossSalary();
    }
    // 领导
    if (isLeader(position)) {
        return getLeaderSalary();
    }
    // 普通员工
    return getStaffSalary();
}
```

先进行条件检查，然后将 if-else 逻辑转成对应的卫语句格式。

另外我们还可以参考 org.apache.commons.lang3.ObjectUtils#isEmpty 的源码：

```
        if (object == null) {
            return true;
        }
        // 第 2 处
        if (object instanceof CharSequence) {
            return ((CharSequence) object).length() == 0;
        }
        // 第 3 处
        if (object.getClass().isArray()) {
            return Array.getLength(object) == 0;
        }

        // 第 4 处
        if (object instanceof Collection<?>) {
            return ((Collection<?>) object).isEmpty();
        }
        // 第 5 处
        if (object instanceof Map<?, ?>) {
            return ((Map<?, ?>) object).isEmpty();
        }
        return false;
    }
```

第 1 处代码满足：某个条件极其罕见，就应该单独检查该条件，并在条件为真时立即从函数中返回。

第 2 到第 5 处代码将某个分支条件转化成卫语句。

在这里特别提醒的是：**对于复杂的判断逻辑，选择使用卫语句时，建议加上注释，并且要仔细核实逻辑是否正确**。

请看下面的伪代码：

```
// 第 1 处
// 同时满足 a 和 b 两个条件
if(condition_a && condition_b){
   if(conditon_c){
     // 业务代码
     return;
   }
}

// 第 2 处
// 条件a 和 b至少有一个不满足
if(!conditon_c){
  // 业务代码
  return;
}
```

上面代码看似正确，其实有很大的问题。

如果同时满足条件 a 和 条件 b 且不满足条件 c，代码依然会执行到 第 2 处，此时 “条件 a 和 b 同时满足” 和 第 2 处的的注释 “条件 a 和 b 至少有一个不满足” 不一致。

我们需要对代码做出如下修改：

```
if(condition_a && condition_b){
  if(conditon_c){
    // 业务代码
  }
// 第 3 处
  // 保证整个if执行后返回
  return;
}

// 第 2 处
// 条件a 和 b至少有一个不满足
if(!conditon_c){
  // 业务代码
  return;
}
```

因此使用卫语句是要特别注重卫语句的先后顺序，当条件非常复杂时，要特别注意卫语句的中断是否符合希望的逻辑。

3. 策略枚举

正如前面的枚举小节讲到的，《Effective Java 中文版》 第 34 条 ：用 enum 代替 int 常量 2 小节描述了使用策略枚举，来替代分支语句，虽然失去了简洁性，但是更加安全和灵活。

通过在枚举内部定义抽象函数，每个枚举常量重写该函数，这样根据枚举值获取枚举常量后调用该函数即可获得期待的计算结果。

示例代码如下：

```
public enum SalaryStrategyEnum {

    BOSS(0) {
        @Override
        double getSalary() {
            return 100000;
        }
    },
    LEADER(1) {
        @Override
        double getSalary() {
            return 50000;
        }
    },
    STAFF(2) {
        @Override
        double getSalary() {
            return 10000;
        }
    };

    private final int position;

    SalaryStrategyEnum(int position) {
        this.position = position;
    }

    abstract double getSalary();
```

```
            return salaryStrategyEnum;
        }
    }
    return null;
  }
}
```

使用时根据枚举值获取枚举对象，直接调用该枚举常量对应的策略:

```
@Test
public void getSalary() {
    SalaryStrategyEnum salaryStrategyEnum = SalaryStrategyEnum.valueOf(0);
    if(salaryStrategyEnum != null){
        log.info("角色： {}-->{} 元",salaryStrategyEnum.name(),salaryStrategyEnum.getSalary());
    }
}
```

当然，大家也可以用非枚举的策略模式来替代多个条件语句。

> 看到这里，可能有些人会认为这种写法工作中并不会用到。
>
> 实则不然，很多知识是你真正理解之后就会想到使用它，恰恰是自认为没用和没有真正理解才导致工作不能灵活运用。
>
> 在工作中，看到多个项目涉及到根据不同枚举计算不同的值时，都用到过类似的写法。

4. 状态模式

《设计模式之禅》 第 26 章 状态模式 (第 343 页) 中讲到：

> 状态模式的使用场景有两类：一种是行为随着状态改变而改变的场景；另外一种是条件、分支判断语句的替代者。

状态模式的其中一个优点就是 “结构清晰”。状态模式体现了开闭原则和单一职责原则，易于拓展和维护。

所谓的结构清晰就是避免了过多的 switch-case 或者 if-else 语句的使用，避免了程序的复杂性，提高了程序的可维护性 3。

接下来我们采用状态模式通过另外一个例子来演示。

原始的 if-else 语句和文章首部给出的非常类似，根据当前状态来执行不同的行为：

学生类：

```
public class Student {

    private Long id;

    private String name;

    private Long age;
}
```

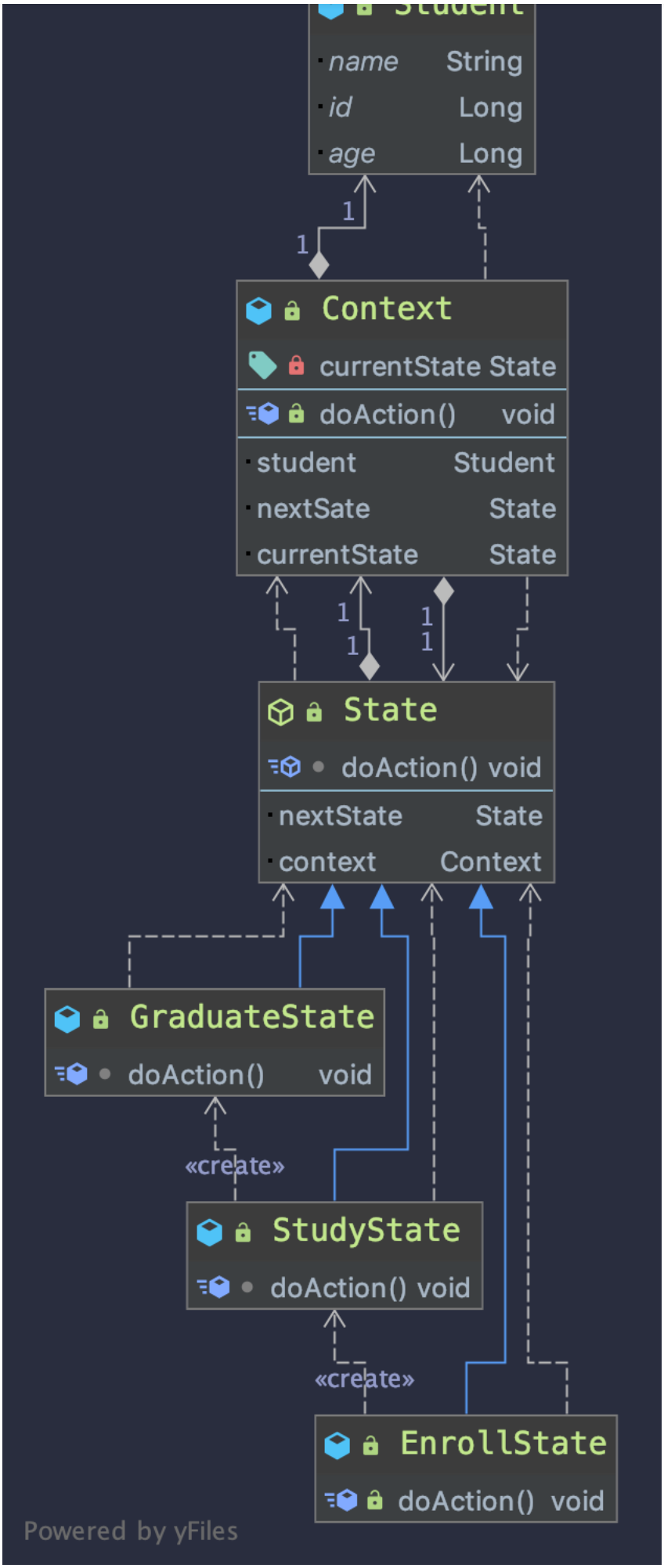
对应的根据状态执行不同的处理函数代码：

```
private void doAction(Integer state, Student student) {

    if (state == null) {
        throw new IllegalArgumentException("状态不能为空");
    }

    switch (state) {
        case 0:
            enroll(student);
            break;
        case 1:
            study(student);
            break;

        case 2:
            graduate(student);
            break;
        default:
    }
}

/**
 * 入学
 */
private void enroll(Student student) {
    System.out.println(String.format("学生%s报名中....", student.getName()));
}

/**
 * 学习
 */
private void study(Student student) {
    System.out.println(String.format("学生%s正在学习....", student.getName()));
}

/**
 * 毕业
 */
private void graduate(Student student) {
    System.out.println(String.format("学生%s毕业了....", student.getName()));
}
```

接下来我们使用状态模式对上面的示例进行修改。

对应的类图如下：

目录



Context 定义客户端所需的接口，并且负责状态的切换。

状态抽象类：

```
@Data
public abstract class State {
    protected Context context;

    protected State nextState;

    public void setContext(Context context) {
        this.context = context;
    }

    abstract void doAction();
}
```

报名状态：

```
/**
 * 报名状态
 */
public class EnrollState extends State {

    public EnrollState() {
        super();
        nextState = new StudyState();
    }

    @Override
    public void doAction() {
        System.out.println(String.format("学生%s报名中....", context.getStudent().getName()));
    }
}
```

学习状态：

```
/**
 * 学习状态
 */
public class StudyState extends State {

    public StudyState() {
        nextState = new GraduateState();
    }

    @Override
    public void doAction() {
        System.out.println(String.format("学生%s正在学习....", context.getStudent().getName()));
    }
}
```

毕业状态：

```
 */
public class GraduateState extends State {

    public GraduateState() {
        nextState = null;
    }

    @Override
    public void doAction() {
        System.out.println(String.format("学生%s毕业了....", context.getStudent().getName()));
    }
}
```

上下文类：

```
public class Context {
    private Student student;
    private State currentState;

    public void doAction() {
        currentState.doAction();
    }

    public State getCurrentState() {
        return currentState;
    }

    public void setCurrentState(State currentState) {
        this.currentState = currentState;
        this.currentState.setContext(this);
    }

    public State getNextSate() {
        return currentState.nextState;
    }

    public Student getStudent() {
        return student;
    }

    public void setStudent(Student student) {
        this.student = student;
    }
}
```

具体使用：

```
public class StateClinet {
    public static void main(String[] args) {
        Student student = new Student();
        student.setName("tomcat");

        Context context = new Context();
        context.setStudent(student);

        // 报名状态
        context.setCurrentState(new EnrollState());
        context.doAction();

        // 学习状态
```

```
            nextSate.doAction();
            nextSate = nextSate.nextState;
        }
    }
}
```

输出：

> 学生 tomcat 报名中…
> 学生 tomcat 正在学习…
> 学生 tomcat 毕业了…

上述示例通过状态模式解决了条件嵌套问题。

5. 拦截器过滤器模式

如果是 Spring Web 项目中还可以通过实现 org.springframework.context.ApplicationContextAware 接口，构造待处理的类型到对应处理器的映射，这也是简化 if-else if-else 的一个重要手段，在实际开发中这种方式也很常见。

定义校验基类：

```
@Data
public abstract class Validator<P> {
    /**
     * 校验分组，枚举
     */
    private Set<Enum> groups;

    /**
     * 验证参数
     */
    abstract void validate(P param);
}
```

自定义校验器：

```
@Component
public class UserSexValidator extends Validator<UserParam> {

    @Override
    void validate(UserParam param) {
        System.out.println("验证性别");
        if (param == null) {
            throw new BusinessException("");
        }
        // 模拟服务，根据userId查询性别
        boolean isFemale = RandomUtils.nextBoolean();
        if (!isFemale) {
            throw new BusinessException("仅限女性玩家哦！");
        }
    }
}
```

构造校验类和校验处理器的映射：

```
@Component
public class ValidatorChain implements ApplicationContextAware {

    private Map<Class, List<Validator>> validatorMap = new HashMap<>();

    /**
     * 根据自定义的校验器进行参数校验
     */
    public <P> void checkParam(P param) {
        checkParam(param, validator -> true);
    }

    /**
     * 符合某种条件才参数校验
     */
    public <P> void checkParam(P param, Predicate<Validator> predicate) {
        List<Validator> validators = getValidators(param.getClass());
        if (CollectionUtils.isNotEmpty(validators)) {
            validators.stream()
                    .filter(predicate)
                    .forEach(validator -> validator.validate(param));
        }
    }


    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansExc
        Map<String, Validator> beansOfType = applicationContext.getBeansOfType(Validator.cla
        this.validatorMap = beansOfType.values().stream().collect(Collectors.groupingBy(validat
    }

    /**
     * 查找相关的所有校验器
     */
    private List<Validator> getValidators(Class clazz) {
        return validatorMap.get(clazz);
    }

    /**
     * 解析泛型待校验参数类型
     */
    private Class getParamType(Class clazz) {
        ResolvableType resolvableType = ResolvableType.forClass(clazz);
        return resolvableType.getSuperType().getGeneric(0).resolve();
    }
}
```

使用校验链：

```
@Service
public class UserServiceImpl implements UserService {

    @Resource
    private ValidatorChain validatorChain;

    @Override
    public UserDTO checkUser(UserParam userParam) {
```

```
        return new UserDTO("测试");
    }

    @Override
    public UserDTO checkUserSome(UserParam userParam) {
        // 参数校验(只校验类型为Some的)
        validatorChain.checkParam(userParam, param -> param.getGroups().contains(UserValic

        // 业务逻辑
        return new UserDTO("测试");
    }
}
```

通过这种方式，对于不同类型对象的属性校验不需要通过 if -else 判断，新增某种类型的校验只需要添加一个自定义校验器即可。还可以支持通过 lambda 表达式传入过滤条件，让符合条件的自定义校验器生效。

6. 嵌套条件语句

嵌套条件语句是多条件语句的变种，相当于增加了内层的一个或者多个嵌套层次。

实际开发中可以将多次使用同一个设计模式，也可以将各种设计模式综合在一起使用。

下面以一个简单的具体例子为例，为大家讲解如何解决嵌套的条件的情况：

用户类：

```
import lombok.Data;

@Data
public class User {

    private Short age;

    private Boolean male;

    private Long id;

    private String name;
}
```

示例代码：

```
public class Demo {

    public static void main(String[] args) {
        Demo demo = new Demo();
        User user = new User();
        user.setAge((short) 17);
        user.setMale(true);
        demo.some(user);
    }

    private void some(User user) {
        Short age = user.getAge();
        if (age < 18) {
```

```
            } else {
                // 其他代码
                System.out.println("18岁 以下女性");
            }
        } else if (age <= 60) {
            if ("张三".equals(user.getName())) {
                // 其他代码
                System.out.println("18到60岁 张三");
            } else if ("李四".equals(user.getName())) {
                System.out.println("18到60岁 李四");
            } else {
                // 其他代码
                System.out.println("18到60岁 其他");
            }
        } else {
            System.out.println("60岁以上");
        }
    }
}
```

接下来，我们使用职责链模式和 Map （也可以用标准的工厂模式）对该条件嵌套示例进行重构。

抽象类：

```
import java.util.function.Predicate;

public abstract class AbstractAgeHandler {

    /**
     * 下一个处理器
     */
    protected AbstractAgeHandler nextAgeHandler;

    /**
     * 设置下一个处理器
     */
    public void setNextAgeHandler(AbstractAgeHandler nextAgeHandler) {
        this.nextAgeHandler = nextAgeHandler;
    }

    public void handle(User user) {
        if (getCondition().test(user.getAge())) {
            doHandle(user);
        }
        if (nextAgeHandler != null) {
            nextAgeHandler.handle(user);
        }
    }

    /**
     * 实际处理函数
     */
    protected abstract void doHandle(User user);

    /**
     * 获取查询条件
     */
    public abstract Predicate<Short> getCondition();
}
```

目录


```
mport java.util.HashMap;
import java.util.Map;
import java.util.function.Predicate;

public class LessThan18Handler extends AbstractAgeHandler {

    // 存储策略
    private static final Map<Boolean, Runnable> SEX_STRATEGY_MAP = new HashMap<>();

    static {
        SEX_STRATEGY_MAP.put(Boolean.TRUE, () -> {
            // 一种处理策略
            System.out.println("小于18岁  男性");
        });

        SEX_STRATEGY_MAP.put(Boolean.FALSE, () -> {
            // 另外的处理策略
            System.out.println("小于18岁  女性");
        });

    }

    /**
     * 该条件的处理函数
     */
    @Override
    protected void doHandle(User user) {
        // 处理小于18岁的代码逻辑

        // 处理性别部分
        SEX_STRATEGY_MAP.get(user.getMale()).run();
    }

    /**
     * 条件18岁
     */
    @Override
    public Predicate<Short> getCondition() {
        return (age) -> age < 18;
    }
}
```

18 到 60 岁之间的处理：

```
import java.util.HashMap;
import java.util.Map;
import java.util.function.Predicate;

public class Between18And60Handler extends AbstractAgeHandler {
    // 存储策略
    private static final Map<String, Runnable> NAME_STRATEGY_MAP = new HashMap<>();

    static {
        NAME_STRATEGY_MAP.put("张三", () -> {
            // 一种处理策略
            System.out.println("18到60岁 张三");
        });

        NAME_STRATEGY_MAP.put("李四", () -> {
```

目录


```
    }

    /**
     * 该条件的处理函数
     */
    @Override
    protected void doHandle(User user) {
        // 处理18 到60岁的代码逻辑

        // 处理性别部分
        Runnable runnable = NAME_STRATEGY_MAP.get(user.getName());
        if (runnable != null) {
            runnable.run();
        } else {
            System.out.println("18到60岁 其他");
        }
    }

    /**
     * 条件18岁
     */
    @Override
    public Predicate<Short> getCondition() {
        return (age) -> age >= 18 && age <= 60;
    }
}
```

大于 60 岁的处理方式：

```
import java.util.function.Predicate;

public class MoreThan60Handler extends AbstractAgeHandler {


    @Override
    protected void doHandle(User user) {
        System.out.println("没有分支逻辑，支持处理");
    }

    @Override
    public Predicate<Short> getCondition() {
        return (age) -> age > 60;
    }
}
```

示例代码：

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

        // 构造年龄处理器
        AbstractAgeHandler first = new LessThan18Handler();
        AbstractAgeHandler second = new Between18And60Handler();
        AbstractAgeHandler third = new MoreThan60Handler();

        // 编排
        first.setNextAgeHandler(second);
```

```
        User user = new User();
        user.setAge((short) 19);
        user.setMale(true);

        first.handle(user);
    }
}
```

究竟选择哪种设计模式要结合具体的场景。

大家可以通过《设计模式之禅》、《Head Ffirst 设计模式》和菜鸟教程等学习常见的设计模式，了解其适合的场景，优缺点等，根据具体场景灵活使用。

7. 总结

本节主要讲了如何解决 if-else 语句拓展性和多层嵌套问题。可以通过卫语句、策略模式、状态模式和过滤拦截器模式等方式解决。

希望大家能够在实际开发中尝试使用这些方法来编写更加优雅的代码。

下一节我们将学习异常处理的相关知识，给出异常处理不当的坑，还会给出一些异常处理的建议。

参考资料

1. 阿里巴巴与 Java 社区开发者.《 Java 开发手册 1.5.0》华山版. 2019 ↩

2. [美] Joshua Bloch.《Effective Java 中文版 (原书第 3 版)》. [译] 俞黎敏。机械工业出版社. 2019 ↩

3. 秦小波.《 设计模式之禅》[M]. 机械工业出版社. 2010. 329-345 ↩

← 16 虚拟机退出时机问题研究 加餐1：工欲善其事必先利其器 →

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论

目录