

# 你见过哪些目瞪口呆的 Java 代码技巧？

Java资料站 2022-10-07 08:46 发表于江苏

大家好，我是锋哥。

## 目录

- 导语
- 开发工具
- 重构
- 技能
- 总结

## 导语

自从毕业后，今年已经是我工作的第 8 个年头了，我甚至都快忘记了到底是哪年毕业的。

从出来，本人一直在做 Java 相关的工作，现在终于有时间坐下来，写一篇关于 Java 写法的一篇文章，来探讨一下如果你真的是一个 Java 程序员，那你真的会写 Java 吗？

笔者是一个务实的程序员，故本文绝非扯淡文章，文中内容都是干货，望读者看后，能有所收获。

本文不是一个吹嘘的文章，不会讲很多高深的架构，相反，会讲解很多基础的问题和写法问题，如果读者自认为基础问题和写法问题都是不是问题，那请忽略这篇文章，节省出时间去做一些有意义的事情。

## 开发工具

不知道有多少“老”程序员还在使用 Eclipse，这些程序员们要不就是因循守旧，要不就是根本就不知道其他好的开发工具的存在，Eclipse 吃内存卡顿的现象以及各种偶然莫名其妙的出现，都告知我们是时候寻找新的开发工具了。

## 更换 IDE

根本就不想多解释要换什么样的 IDE，如果你想成为一个优秀的 Java 程序员，请更换 IntelliJ IDEA。使用 IDEA 的好处，请搜索谷歌。

## 别告诉我快捷键不好用

更换 IDE 不在我本文的重点内容中，所以不想用太多的篇幅去写为什么更换IDE。在这里，我只能告诉你，更换 IDE 只为了更好、更快的写好 Java 代码。原因略。

别告诉我快捷键不好用，请尝试新事物。

## bean

bean 使我们使用最多的模型之一，我将以大篇幅去讲解 bean，希望读者好好体会。

## domain 包名

根据很多 Java 程序员的“经验”来看，一个数据库表则对应着一个 domain 对象，所以很多程序员在写代码时，包名则使用：com.xxx.domain，这样写好像已经成为了行业的一种约束，数据库映射对象就应该是 domain。

但是你错了，domain 是一个领域对象，往往我们再做传统 Java 软件 Web 开发中，这些 domain 都是贫血模型，是没有行为的，或是没有足够的领域模型的行为的。

所以，以这个理论来讲，这些 domain 都应该是一个普通的 entity 对象，并非领域对象，所以请把包名改为：com.xxx.entity。

如果你还不理解我说的话，请看一下 Vaughn Vernon 出的一本叫做《IMPLEMENTING DOMAIN-DRIVEN DESIGN》（实现领域驱动设计）这本书，书中讲解了贫血模型与领域模型的区别，相信你会受益匪浅。

## DTO

数据传输我们应该使用 DTO 对象作为传输对象，这是我们所约定的，因为很长时间我一直都在做移动端 API 设计的工作，有很多人告诉我，他们认为只有给手机端传输数据的时候（input or output），这些对象成为 DTO 对象。

请注意！这种理解是错误的，只要是用于网络传输的对象，我们都认为他们可以当做是 DTO 对象，比如电商平台中，用户进行下单，下单后的数据，订单会发到 OMS 或者 ERP 系统，这些对接

的返回值以及入参也叫 DTO 对象。

我们约定某对象如果是 DTO 对象，就将名称改为 XXDTO，比如订单下发 OMS：OMSOrderInputDTO。

## DTO 转化

正如我们所知，DTO 为系统与外界交互的模型对象，那么肯定会有一个步骤是将 DTO 对象转化为 BO 对象或者是普通的 entity 对象，让 service 层去处理。

## 场景

比如添加会员操作，由于用于演示，我只考虑用户的一些简单数据，当后台管理员点击添加用户时，只需要传过来用户的姓名和年龄就可以了，后端接受到数据后，将添加创建时间和更新时间和默认密码三个字段，然后保存数据库。

```
@RequestMapping("/v1/api/user")
@RestController
public class UserApi {

    @Autowired
    private UserService userService;

    @PostMapping
    public User addUser(UserInputDTO userInputDTO){
        User user = new User();
        user.setUsername(userInputDTO.getUsername());
        user.setAge(userInputDTO.getAge());

        return userService.addUser(user);
    }
}
```

我们只关注一下上述代码中的转化代码，其他内容请忽略：

```
User user = new User();
user.setUsername(userInputDTO.getUsername());
user.setAge(userInputDTO.getAge());
```

## 请使用工具

上边的代码，从逻辑上讲，是没有问题的，只是这种写法让我很厌烦，例子中只有两个字段，如果有 20 个字段，我们要如何做呢？一个一个进行 set 数据吗？

当然，如果你这么做了，肯定不会有什么问题，但是，这肯定不是一个最优的做法。网上有很多工具，支持浅拷贝或深拷贝的 Utils。

举个例子，我们可以使用 `org.springframework.beans.BeanUtils#copyProperties` 对代码进行重构和优化：

```
@PostMapping
public User addUser(UserInputDTO userInputDTO){
    User user = new User();
    BeanUtils.copyProperties(userInputDTO,user);

    return userService.addUser(user);
}
```

`BeanUtils.copyProperties` 是一个浅拷贝方法，复制属性时，我们只需要把 DTO 对象和要转化的对象两个的属性值设置为一样的名称，并且保证一样的类型就可以了。

如果你在做 DTO 转化的时候一直使用 `set` 进行属性赋值，那么请尝试这种方式简化代码，让代码更加清晰！

## 转化的语义

上边的转化过程，读者看后肯定觉得优雅很多，但是我们再写 Java 代码时，更多的需要考虑语义的操作，再看上边的代码：

```
User user = new User();
BeanUtils.copyProperties(userInputDTO,user);
```

虽然这段代码很好的简化和优化了代码，但是他的语义是有问题的，我们需要提现一个转化过程才好，所以代码改成如下：

```
@PostMapping
public User addUser(UserInputDTO userInputDTO){
    User user = convertFor(userInputDTO);

    return userService.addUser(user);
}

private User convertFor(UserInputDTO userInputDTO){

    User user = new User();
    BeanUtils.copyProperties(userInputDTO,user);
    return user;
}
```

这是一个更好的语义写法，虽然他麻烦了些，但是可读性大大增加了，在写代码时，我们应该尽量把语义层次差不多的放到一个方法中，比如：

```
User user = convertFor(userInputDTO);  
return userService.addUser(user);
```

这两段代码都没有暴露实现，都是在讲如何在同一个方法中，做一组相同层次的语义操作，而不是暴露具体的实现。

如上所述，是一种重构方式，读者可以参考 Martin Fowler 的《Refactoring Improving the Design of Existing Code》（重构改善既有代码的设计）这本书中的 Extract Method 重构方式。

## 抽象接口定义

当实际工作中，完成了几个 API 的 DTO 转化时，我们会发现，这样的操作有很多很多，那么应该定义好一个接口，让所有这样的操作都有规则的进行。

如果接口被定义以后，那么 convertFor 这个方法的语义将产生变化，它将是一个实现类。

看一下抽象后的接口：

```
public interface DTOConvert<S,T> {  
    T convert(S s);  
}
```

虽然这个接口很简单，但是这里告诉我们一个事情，要去使用泛型，如果你是一个优秀的 Java 程序员，请为你想做的抽象接口，做好泛型吧。

我们再来看接口实现：

```
public class UserInputDTOConvert implements DTOConvert {  
    @Override  
    public User convert(UserInputDTO userInputDTO) {  
        User user = new User();  
        BeanUtils.copyProperties(userInputDTO,user);  
        return user;  
    }  
}
```

我们这样重构后，我们发现现在的代码是如此的简洁，并且那么的规范：

```
@RequestMapping("/v1/api/user")  
@RestController
```

```
public class UserApi {  
  
    @Autowired  
    private UserService userService;  
  
    @PostMapping  
    public User addUser(UserInputDTO userInputDTO){  
        User user = new UserInputDTOConvert().convert(userInputDTO);  
  
        return userService.addUser(user);  
    }  
}
```

## review code

如果你是一个优秀的 Java 程序员，我相信你应该和我一样，已经数次重复 review 过自己的代码很多次了。

我们再看这个保存用户的例子，你将发现，API 中返回值是有些问题的，问题就在于不应该直接返回 User 实体，因为如果这样的话，就暴露了太多实体相关的信息，这样的返回值是不安全的。

所以我们更应该返回一个 DTO 对象，我们可称它为 UserOutputDTO：

```
@PostMapping  
public UserOutputDTO addUser(UserInputDTO userInputDTO){  
    User user = new UserInputDTOConvert().convert(userInputDTO);  
    User saveUserResult = userService.addUser(user);  
    UserOutputDTO result = new UserOutputDTOConvert().convertToUser(saveUserResult);  
    return result;  
}
```

这样你的 API 才更健全。

不知道在看完这段代码之后，读者有是否发现还有其他问题的存在，作为一个优秀的 Java 程序员，请看一下这段我们刚刚抽象完的代码：

```
User user = new UserInputDTOConvert().convert(userInputDTO);
```

你会发现，new 这样一个 DTO 转化对象是没有必要的，而且每一个转化对象都是由在遇到 DTO 转化的时候才会出现，那我们应该考虑一下，是否可以将这个类和 DTO 进行聚合呢？

看一下我的聚合结果：

```
public class UserInputDTO {  
    private String username;  
    private int age;
```

```
public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public User convertToUser(){
    UserInputDTOConvert userInputDTOConvert = new UserInputDTOConvert();
    User convert = userInputDTOConvert.convert(this);
    return convert;
}

private static class UserInputDTOConvert implements DTOConvert<UserInputDTO,User> {
    @Override
    public User convert(UserInputDTO userInputDTO) {
        User user = new User();
        BeanUtils.copyProperties(userInputDTO,user);
        return user;
    }
}
}
```

然后 API 中的转化则由：

```
User user = new UserInputDTOConvert().convert(userInputDTO);
User saveUserResult = userService.addUser(user);
```

变成了：

```
User user = userInputDTO.convertToUser();
User saveUserResult = userService.addUser(user);
```

我们再 DTO 对象中添加了转化的行为，我相信这样的操作可以让代码的可读性变得更强，并且是符合语义的。

## 再查工具类

再来看 DTO 内部转化的代码，它实现了我们自己定义的 DTOConvert 接口，但是这样真的就没有问题，不需要再思考了吗？

我觉得并不是，对于 Convert 这种转化语义来讲，很多工具类中都有这样的定义，这中 Convert 并不是业务级别上的接口定义，它只是用于普通 bean 之间转化属性值的普通意义上的接口定义，所以我们应该更多的去读其他含有 Convert 转化语义的代码。

我仔细阅读了一下 GUAVA 的源码，发现了 com.google.common.base.Convert 这样的定义：

```
public abstract class Converter<A, B> implements Function<A, B> {  
    protected abstract B doForward(A a);  
    protected abstract A doBackward(B b);  
    //其他略  
}
```

从源码可以了解到，GUAVA 中的 Convert 可以完成正向转化和逆向转化，继续修改我们 DTO 中转化的这段代码：

```
private static class UserInputDTOConvert implements DTOConvert<UserInputDTO, User> {  
    @Override  
    public User convert(UserInputDTO userInputDTO) {  
        User user = new User();  
        BeanUtils.copyProperties(userInputDTO, user);  
        return user;  
    }  
}
```

修改后：

```
private static class UserInputDTOConvert extends Converter<UserInputDTO, User> {  
    @Override  
    protected User doForward(UserInputDTO userInputDTO) {  
        User user = new User();  
        BeanUtils.copyProperties(userInputDTO, user);  
        return user;  
    }  
  
    @Override  
    protected UserInputDTO doBackward(User user) {  
        UserInputDTO userInputDTO = new UserInputDTO();  
        BeanUtils.copyProperties(user, userInputDTO);  
        return userInputDTO;  
    }  
}
```

看了这部分代码以后，你可能会问，那逆向转化会有什么用呢？其实我们有很多小的业务需求中，入参和出参是一样的，那么我们变可以轻松的进行转化，我将上边所提到的 UserInputDTO 和



UserOutputDTO 都转成 UserDTO 展示给大家。

DTO:

```
public class UserDTO {
    private String username;
    private int age;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public User convertToUser(){
        UserDTOConvert userDTOConvert = new UserDTOConvert();
        User convert = userDTOConvert.convert(this);
        return convert;
    }

    public UserDTO convertFor(User user){
        UserDTOConvert userDTOConvert = new UserDTOConvert();
        UserDTO convert = userDTOConvert.reverse().convert(user);
        return convert;
    }

    private static class UserDTOConvert extends Converter<UserDTO, User> {
        @Override
        protected User doForward(UserDTO userDTO) {
            User user = new User();
            BeanUtils.copyProperties(userDTO,user);
            return user;
        }

        @Override
        protected UserDTO doBackward(User user) {
            UserDTO userDTO = new UserDTO();
            BeanUtils.copyProperties(user,userDTO);
            return userDTO;
        }
    }
}
```



其实答案是这样的，我从不相信任何调用我 API 或者方法的人，比如前端验证失败了，或者某些人通过一些特殊的渠道(比如 Charles 进行抓包)，直接将数据传入到我的 API，那我仍然进行正常的业务逻辑处理，那么就有可能产生脏数据！

“对于脏数据的产生一定是致命”，这句话希望大家牢记在心，再小的脏数据也有可能让你找几个通宵！

## jsr 303 验证

hibernate 提供的 jsr 303 实现，我觉得目前仍然是很优秀的，具体如何使用，我不想讲，因为谷歌上你可以搜索出很多答案！

再以上班的 API 实例进行说明，我们现在对 DTO 数据进行检查：

```
public class UserDTO {  
    @NotNull  
    private String username;  
    @NotNull  
    private int age;  
    //其他代码略  
}
```

API 验证：

```
@PostMapping  
public UserDTO addUser(@Valid UserDTO userDTO){  
    User user = userDTO.convertToUser();  
    User saveResultUser = userService.addUser(user);  
    UserDTO result = userDTO.convertFor(saveResultUser);  
    return result;  
}
```

我们需要将验证结果传给前端，这种异常应该转化为一个 api 异常（带有错误码的异常）。

```
@PostMapping  
public UserDTO addUser(@Valid UserDTO userDTO, BindingResult bindingResult){  
    checkDTOParams(bindingResult);  
  
    User user = userDTO.convertToUser();  
    User saveResultUser = userService.addUser(user);  
    UserDTO result = userDTO.convertFor(saveResultUser);  
    return result;  
}  
private void checkDTOParams(BindingResult bindingResult){  
    if(bindingResult.hasErrors()){  
        //throw new 带验证码的验证错误异常  
    }  
}
```

BindingResult 是 Spring MVC 验证 DTO 后的一个结果集，可以参考 spring 官方文档：

<https://spring.io/>

检查参数后，可以抛出一个“带验证码的验证错误异常”。

## 拥抱 lombok

上边的 DTO 代码，已经让我看的很累了，我相信读者也是一样，看到那么多的 Getter 和 Setter 方法，太烦躁了，那时候有什么方法可以简化这些呢。

请拥抱 lombok，它会帮助我们解决一些让我们很烦躁的问题。

## 去掉 Setter 和 Getter

其实这个标题，我不太想说，因为网上太多，但是因为很多人告诉我，他们根本就不知道 lombok 的存在，所以为了让读者更好的学习，我愿意写这样一个例子：

```
@Setter
@Getter
public class UserDTO {
    @NotNull
    private String username;
    @NotNull
    private int age;

    public User convertToUser(){
        UserDTOConvert userDTOConvert = new UserDTOConvert();
        User convert = userDTOConvert.convert(this);
        return convert;
    }

    public UserDTO convertFor(User user){
        UserDTOConvert userDTOConvert = new UserDTOConvert();
        UserDTO convert = userDTOConvert.reverse().convert(user);
        return convert;
    }

    private static class UserDTOConvert extends Converter<UserDTO, User> {
        @Override
        protected User doForward(UserDTO userDTO) {
            User user = new User();
            BeanUtils.copyProperties(userDTO,user);
            return user;
        }

        @Override
        protected UserDTO doBackward(User user) {
            throw new AssertionError("不支持逆向转化方法!");
        }
    }
}
```

```
    }  
}  
  
}
```

看到了吧，烦人的 Getter 和 Setter 方法已经去掉了。

但是上边的例子根本不足以体现 lombok 的强大。我希望写一些网上很难查到，或者很少人进行说明的 lombok 的使用以及在使用时程序语义上的说明。

比如：@Data, @AllArgsConstructor, @NoArgsConstructor..这些我就不进行一一说明了，请大家自行查询资料。

## bean 中的链式风格

什么是链式风格？我来举个例子，看下面这个 Student 的 bean：

```
public class Student {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return name;  
    }  
  
    public Student setName(String name) {  
        this.name = name;  
        return this;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public Student setAge(int age) {  
        return this;  
    }  
}
```

仔细看一下 set 方法，这样的设置便是 chain 的 style，调用的时候，可以这样使用：

```
Student student = new Student()  
    .setAge(24)  
    .setName("zs");
```

相信合理使用这样的链式代码，会更多的程序带来很好的可读性，那看一下如果使用 lombok 进行改善呢，请使用 @Accessors(chain = true)，看如下代码：

```
@Accessors(chain = true)
@Setter
@Getter
public class Student {
    private String name;
    private int age;
}
```

这样就完成了一个对于 bean 来讲很友好的链式操作。

## 静态构造方法

静态构造方法的语义和简化程度真的高于直接去 new 一个对象。比如 new 一个 List 对象，过去的使用是这样的：

```
List<String> list = new ArrayList<>();
```

看一下 guava 中的创建方式：

```
List<String> list = Lists.newArrayList();
```

Lists 命名是一种约定（俗话说：约定优于配置），它是指 Lists 是 List 这个类的一个工具类，那么使用 List 的工具类去产生 List，这样的语义是不是要比直接 new 一个子类来的更直接一些呢，答案是肯定的。

再比如如果有一个工具类叫做 Maps，你是否想到了创建 Map 的方法呢：

```
HashMap<String, String> objectObjectHashMap = Maps.newHashMap();
```

好了，如果你理解了我说的语义，那么，你已经向成为 Java 程序员更近了一步了。

再回过头来看刚刚的 Student，很多时候，我们去写 Student 这个 bean 的时候，他会有一些必填字段。

比如 Student 中的 name 字段，一般处理的方式是将 name 字段包装成一个构造方法，只有传入 name 这样的构造方法，才能创建一个 Student 对象。

接上上边的静态构造方法和必填参数的构造方法，使用 lombok 将更改成如下写法（@RequiredArgsConstructor 和 @NonNull）：

```
@Accessors(chain = true)
@Setter
@Getter
@RequiredArgsConstructor(staticName = "ofName")
```

```
public class Student {  
    @NonNull private String name;  
    private int age;  
}
```

测试代码：

```
Student student = Student.ofName("zs");
```

这样构建出的 bean 语义是否要比直接 new 一个含参的构造方法（包含 name 的构造方法）要好很多。

当然，看过很多源码以后，我想相信将静态构造方法 ofName 换成 of 会先的更加简洁：

```
@Accessors(chain = true)  
@Setter  
@Getter  
@RequiredArgsConstructor(staticName = "of")  
public class Student {  
    @NonNull private String name;  
    private int age;  
}
```

测试代码：

```
Student student = Student.of("zs");
```

当然他仍然是支持链式调用的：

```
Student student = Student.of("zs").setAge(24);
```

这样来写代码，真的很简洁，并且可读性很强。

## 使用 builder

Builder 模式我不想再多解释了，读者可以看一下《Head First》(设计模式) 的建造者模式。

今天其实要说的是一种变种的 builder 模式，那就是构建 bean 的 builder 模式，其实主要的思想是带着大家一起看一下 lombok 给我们带来了什么。

看一下 Student 这个类的原始 builder 状态：

```
public class Student {
```

```
private String name;
private int age;

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public static Builder builder(){
    return new Builder();
}
public static class Builder{
    private String name;
    private int age;
    public Builder name(String name){
        this.name = name;
        return this;
    }

    public Builder age(int age){
        this.age = age;
        return this;
    }

    public Student build(){
        Student student = new Student();
        student.setAge(age);
        student.setName(name);
        return student;
    }
}
}
```

调用方式：

```
Student student = Student.builder().name("zs").age(24).build();
```

这样的 builder 代码，让我是在恶心难受，于是我打算用 lombok 重构这段代码：

```
@Builder
public class Student {
    private String name;
```



```

        private int age;
    }

```

调用方式：

```
Student student = Student.builder().name("zs").age(24).build();
```

## 代理模式

正如我们所知的，在程序中调用 rest 接口是一个常见的行为动作，如果你和我一样使用过 spring 的 RestTemplate，我相信你会和我一样，对他抛出的非 http 状态码异常深恶痛绝。

所以我们考虑将 RestTemplate 最为底层包装器进行包装器模式的设计：

```

public abstract class FilterRestTemplate implements RestOperations {
    protected volatile RestTemplate restTemplate;

    protected FilterRestTemplate(RestTemplate restTemplate){
        this.restTemplate = restTemplate;
    }

    //实现RestOperations所有的接口
}

```

然后再由扩展类对 FilterRestTemplate 进行包装扩展：

```

public class ExtractRestTemplate extends FilterRestTemplate {
    private RestTemplate restTemplate;
    public ExtractRestTemplate(RestTemplate restTemplate) {
        super(restTemplate);
        this.restTemplate = restTemplate;
    }

    public <T> RestResponseDTO<T> postForEntityWithNoException(String url, Object request,
        throws RestClientException {
        RestResponseDTO<T> restResponseDTO = new RestResponseDTO<T>();
        ResponseEntity<T> tResponseEntity;
        try {
            tResponseEntity = restTemplate.postForEntity(url, request, response
            restResponseDTO.setData(tResponseEntity.getBody());
            restResponseDTO.setMessage(tResponseEntity.getStatusCode().name());
            restResponseDTO.setStatusCode(tResponseEntity.getStatusCodeValue())
        } catch (Exception e){
            restResponseDTO.setStatusCode(RestResponseDTO.UNKNOWN_ERROR);
            restResponseDTO.setMessage(e.getMessage());
            restResponseDTO.setData(null);
        }
        return restResponseDTO;
    }
}

```

包装器 `ExtractRestTemplate` 很完美的更改了异常抛出的行为，让程序更具有容错性。

在这里我们不考虑 `ExtractRestTemplate` 完成的功能，让我们把焦点放在 `FilterRestTemplate` 上，“实现 `RestOperations` 所有的接口”。

这个操作绝对不是一时半会可以写完的，当时在重构之前我几乎写了半个小时，如下：

```
public abstract class FilterRestTemplate implements RestOperations {

    protected volatile RestTemplate restTemplate;

    protected FilterRestTemplate(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @Override
    public <T> T getForObject(String url, Class<T> responseType, Object... uriVariables)
        return restTemplate.getForObject(url,responseType,uriVariables);
    }

    @Override
    public <T> T getForObject(String url, Class<T> responseType, Map<String, ?> uriVariables)
        return restTemplate.getForObject(url,responseType,uriVariables);
    }

    @Override
    public <T> T getForObject(URI url, Class<T> responseType) throws RestClientException
        return restTemplate.getForObject(url,responseType);
    }

    @Override
    public <T> ResponseEntity<T> getForEntity(String url, Class<T> responseType, Object... uriVariables)
        return restTemplate.getForEntity(url,responseType,uriVariables);
    }
    //其他实现代码略。。。
}
```

我相信你看了以上代码，你会和我一样觉得恶心反胃，后来我用 `lombok` 提供的代理注解优化了我的代码（`@Delegate`）：

```
@AllArgsConstructor
public abstract class FilterRestTemplate implements RestOperations {
    @Delegate
    protected volatile RestTemplate restTemplate;
}
```

这几行代码完全替代上述那些冗长的代码。是不是很简洁，做一个拥抱 lombok 的程序员吧。

## 重构

### 需求案例

### 项目需求

项目开发阶段，有一个关于下单发货的需求：如果今天下午 3 点前进行下单，那么发货时间是明天，如果今天下午 3 点后进行下单，那么发货时间是后天，如果被确定的时间是周日，那么在此时间上再加 1 天为发货时间。

### 思考与重构

我相信这个需求看似很简单，无论怎么写都可以完成。

很多人可能看到这个需求，就动手开始写 Calendar 或 Date 进行计算，从而完成需求。

而我给的建议是，仔细考虑如何写代码，然后再去写，不是说所有的时间操作都用 Calendar 或 Date 去解决，一定要看场景。

对于时间的计算我们要考虑 joda-time 这种类似的成熟时间计算框架来写代码，它会让代码更加简洁和易读。

请读者先考虑这个需求如何用 Java 代码完成，或先写一个你觉得完成这个代码的思路，再来看我下边的代码，这样，你的收获会更多一些：

```
final DateTime DISTRIBUTION_TIME_SPLIT_TIME = new DateTime().withTime(15,0,0,0);
private Date calculateDistributionTimeByOrderCreateTime(Date orderCreateTime){
    DateTime orderCreateDateTime = new DateTime(orderCreateTime);
    Date tomorrow = orderCreateDateTime.plusDays(1).toDate();
    Date theDayAfterTomorrow = orderCreateDateTime.plusDays(2).toDate();
    return orderCreateDateTime.isAfter(DISTRIBUTION_TIME_SPLIT_TIME) ? wrapDistributionTime(tomorrow) : theDayAfterTomorrow;
}
private Date wrapDistributionTime(Date distributionTime){
    DateTime currentDistributionDateTime = new DateTime(distributionTime);
    DateTime plusOneDay = currentDistributionDateTime.plusDays(1);
    boolean isSunday = (DateTimeConstants.SUNDAY == currentDistributionDateTime.getDayOfYear());
    return isSunday ? plusOneDay.toDate() : currentDistributionDateTime.toDate();
}
```

读这段代码的时候，你会发现，我将判断和有可能出现的不同结果都当做一个变量，最终做一个三目运算符的方式进行返回。

这样的优雅和可读性显而易见，当然这样的代码不是一蹴而就的，我优化了 3 遍产生的以上代码。读者可根据自己的代码和我写的代码进行对比。

## 提高方法

如果你做了 3 年+的程序员，我相信像如上这样的需求，你很容易就能完成，但是如果你想做一个会写 Java 的程序员，就好好思考和重构代码吧。

写代码就如同写字一样，同样的字，大家都会写，但是写出来是否好看就不一定了。如果想把程序写好，就要不断的思考和重构，敢于尝试，敢于创新，不要因循守旧，一定要做一个优秀的 Java 程序员。

提高代码水平最好的方法就是有条理的重构！（注意：是有条理的重构）

## 设计模式

设计模式就是工具，而不是提现你是否是高水平程序员的一个指标。

我经常看到某一个程序员兴奋的大喊，哪个程序哪个点我用到了设计模式，写的多么多么优秀，多么多么好。我仔细去翻阅的时候，却发现有很多是过度设计的。

## 业务驱动技术 or 技术驱动业务

业务驱动技术 or 技术驱动业务？其实这是一个一直在争论的话题，但是很多人不这么认为，我觉得就是大家不愿意承认罢了。我来和大家大概分析一下作为一个 Java 程序员，我们应该如何判断自己所处于的位置。

**业务驱动技术：**如果你所在的项目是一个收益很小或者甚至没有收益的项目，请不要搞其他创新的东西，不要驱动业务要如何如何做，而是要熟知业务现在的痛点是什么？如何才能帮助业务盈利或者让项目更好，更顺利的进行。

**技术驱动业务：**如果你所在的项目是一个很牛的项目，比如淘宝这类的项目，我可以在满足业务需求的情况下，和业务沟通，使用什么样的技术能更好的帮助业务创造收益。

比如说下单的时候要进队列，可能几分钟之后订单状态才能处理完成，但是会让用户有更流畅的体验，赚取更多的访问流量，那么我相信业务愿意被技术驱动，会同意订单的延迟问题，这样便是技术驱动业务。

我相信大部分人还都处于业务驱动技术的方向吧。所以你既然不能驱动业务，那就请拥抱业务变化吧。

## 代码设计

一直在做 Java 后端的项目，经常会有一些变动，我相信大家也都遇到过。

比如当我们写一段代码的时候，我们考虑将需求映射成代码的状态模式，突然有一天，状态模式里边又添加了很多行为变化的东西，这时候你就挠头了，你硬生生的将状态模式中增加过多行为和变化。

慢慢的你会发现这些状态模式，其实更像是一簇算法，应该使用策略模式，这时你应该已经晕头转向了。

说了这么多，我的意思是，只要你觉得合理，就请将状态模式改为策略模式吧，所有的模式并不是凭空想象出来的，都是基于重构。

Java 编程中没有银弹，请拥抱业务变化，一直思考重构，你就有一个更好的代码设计！

## 你真的优秀吗？

真不好意思，我取了一个这么无聊的标题。

国外流行一种编程方式，叫做结对编程，我相信国内很多公司都没有这么做，我就不在讲述结对编程带来的好处了，其实就是一边 code review，一边互相提高的一个过程。既然做不到这个，那如何让自己活在自己的世界中不断提高呢？

“平时开发的时候，做出的代码总认为是正确的，而且写法是完美的。”，我相信这是大部分人的心声，还回到刚刚的问题，如何在自己的世界中不断提高呢？

答案就是：

- 多看成熟框架的源码
- 多回头看自己的代码
- 勤于重构

你真的优秀吗？如果你每周都完成了学习源码，回头看自己代码，然后勤于重构，我认为你就真的很优秀了。

即使也许你只是刚刚入门，但是一直坚持，你就是一个真的会写 Java 代码的程序员了。

## 技能

### UML

不想多讨论 UML 相关的知识，但是我觉得你如果真的会写 Java，请先学会表达自己，UML 就是你说话的语言。

做一名优秀的 Java 程序员，请至少学会这两种 UML 图：

- 类图
- 时序图

### clean code

我认为保持代码的简洁和可读性是代码的最基本保证，如果有一天为了程序的效率而降低了这两点，我认为是可以谅解的，除此之外，没有任何理由可以让你任意挥霍你的代码。

无论如何，请保持你的代码的整洁。

### Linux 基础命令

这点其实和会写 Java 没有关系，但是 Linux 很多时候确实承载运行 Java 的容器，请学好 Linux 的基础命令。

## 总结

Java 是一个大体系，今天讨论并未涉及框架和架构相关知识，只是讨论如何写好代码。

本文从写 Java 程序的小方面一直写到大方面，来阐述了如何才能写好 Java 程序，并告诉读者们如何才能提高自身的编码水平。

我希望看到这篇文章的各位都能做一个优秀的 Java 程序员

来自：掘金，作者：java小瓜哥

链接：<https://juejin.cn/post/6844903954308939784>

End

○○○○○○

锋哥的 SpringSecurity+Vue权限系统 震撼发布！ ...

安排一个福利，Java全栈就业实战课程 免费哦...

66套Java实战项目课程领取...

2022年粉丝福利

<http://download.java1234.com/>

每月送 666 套Java海量资源网站 VIP会员，供大伙一起学Java

如果没加过锋哥微信的

加一下锋哥微信备注 VIP 即可开通



👉 长按上方二维码2秒，备注vip



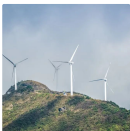
锋哥，10年Java老司机，小锋网络科技 光杠司令员，司令部：[www.java1234.vip](http://www.java1234.vip) 每天坚持锻炼身体，坚持早睡早起，崇尚自由，平时喜欢带带Java学员（已经成功指导1000+学员高薪就业），喜欢搞搞Java技术自媒体，搞搞小产品，后期继续研究主流技术，以及进军短视频+直播领域，每天进步一点，奥利给。

阅读原文

喜欢此内容的人还喜欢

Mybatis中SQL注入攻击的3种方式，真是防不胜防！

江南一点雨



探索组件在线预览和调试

政采云前端



两万字长文让你彻底掌握 celery

古明地觉的编程教室

