

## 21 | 理论七：重复的代码就一定违背DRY吗？如何提高代码的复用性？

2019-12-20 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 17:13 大小 15.78M



在上一节课中，我们讲了 KISS 原则和 YAGNI 原则，KISS 原则可以说是人尽皆知。今天，我们再学习一个你肯定听过的原则，那就是 DRY 原则。它的英文描述为：Don't Repeat Yourself。中文直译为：不要重复自己。将它应用在编程中，可以理解为：不要写重复的代码。

你可能会觉得，这条原则非常简单、非常容易应用。只要两段代码长得一样，那就是违反 DRY 原则了。真的是这样吗？答案是否定的。这是很多人对这条原则存在的误解。实际上，重复的代码不一定违反 DRY 原则，而且有些看似不重复的代码也有可能违反 DRY 原则。

听到这里，你可能会有很多疑问。没关系，今天我会结合具体的代码实例，来把这个问题讲清楚，纠正你对这个原则的错误认知。除此之外，DRY 原则与代码的复用性也有一些联系，所以，今天，我还会讲一讲，如何写出可复用性好的代码。

话不多说，让我们正式开始今天的学习吧！

## DRY 原则 (Don't Repeat Yourself)

DRY 原则的定义非常简单，我就不再过度解读。今天，我们主要讲三种典型的代码重复情况，它们分别是：实现逻辑重复、功能语义重复和代码执行重复。这三种代码重复，有的看似违反 DRY，实际上并不违反；有的看似不违反，实际上却违反了。

### 实现逻辑重复

我们先来看下面这样一段代码是否违反了 DRY 原则。如果违反了，你觉得应该如何重构，才能让它满足 DRY 原则？如果没有违反，那又是为什么呢？

 复制代码

```
1 public class UserAuthenticator {
2     public void authenticate(String username, String password) {
3         if (!isValidUsername(username)) {
4             // ...throw InvalidUsernameException...
5         }
6         if (!isValidPassword(password)) {
7             // ...throw InvalidPasswordException...
8         }
9         //... 省略其他代码...
10    }
11
12    private boolean isValidUsername(String username) {
13        // check not null, not empty
14        if (StringUtils.isBlank(username)) {
15            return false;
16        }
17        // check length: 4~64
18        int length = username.length();
19        if (length < 4 || length > 64) {
```

```

20     return false;
21 }
22 // contains only lowercase characters
23 if (!StringUtils.isAllLowerCase(username)) {
24     return false;
25 }
26 // contains only a~z,0~9,dot
27 for (int i = 0; i < length; ++i) {
28     char c = username.charAt(i);
29     if (!(c >= 'a' && c <= 'z') || (c >= '0' && c <= '9') || c == '.')) {
30         return false;
31     }
32 }
33 return true;
34 }
35
36 private boolean isValidPassword(String password) {
37     // check not null, not empty
38     if (StringUtils.isBlank(password)) {
39         return false;
40     }
41     // check length: 4~64
42     int length = password.length();
43     if (length < 4 || length > 64) {
44         return false;
45     }
46     // contains only lowercase characters
47     if (!StringUtils.isAllLowerCase(password)) {
48         return false;
49     }
50     // contains only a~z,0~9,dot
51     for (int i = 0; i < length; ++i) {
52         char c = password.charAt(i);
53         if (!(c >= 'a' && c <= 'z') || (c >= '0' && c <= '9') || c == '.')) {
54             return false;
55         }
56     }
57     return true;
58 }
59 }

```

代码很简单，我就不做过多解释了。在代码中，有两处非常明显的重复的代码片段：`isValidUserName()` 函数和 `isValidPassword()` 函数。重复的代码被敲了两遍，或者简单 copy-paste 了一下，看起来明显违反 DRY 原则。为了移除重复的代码，我们对上面的代码做下重构，将 `isValidUserName()` 函数和 `isValidPassword()` 函数，合并为一个更通用的函数 `isValidUserNameOrPassword()`。重构后的代码如下所示：

```
1 public class UserAuthenticatorV2 {  
2  
3     public void authenticate(String userName, String password) {  
4         if (!isValidUsernameOrPassword(userName)) {  
5             // ...throw InvalidUsernameException...  
6         }  
7  
8         if (!isValidUsernameOrPassword(password)) {  
9             // ...throw InvalidPasswordException...  
10        }  
11    }  
12  
13    private boolean isValidUsernameOrPassword(String usernameOrPassword) {  
14        // 省略实现逻辑  
15        // 跟原来的 isValidUsername() 或 isValidPassword() 的实现逻辑一样...  
16        return true;  
17    }  
18 }
```

经过重构之后，代码行数减少了，也没有重复的代码了，是不是更好了呢？答案是否定的，这可能跟你预期的不一样，我来解释一下为什么。

单从名字上看，我们就能发现，合并之后的 `isValidUserNameOrPassword()` 函数，负责两件事情：验证用户名和验证密码，违反了“单一职责原则”和“接口隔离原则”。实际上，即便将两个函数合并成 `isValidUserNameOrPassword()`，代码仍然存在问题。

因为 `isValidUserName()` 和 `isValidPassword()` 两个函数，虽然从代码实现逻辑上看起来是重复的，但是从语义上并不重复。所谓“语义不重复”指的是：从功能上来看，这两个函数干的是完全不重复的两件事情，一个是校验用户名，另一个是校验密码。尽管在目前的设计中，两个校验逻辑是完全一样的，但如果按照第二种写法，将两个函数的合并，那就会存在潜在的问题。在未来的某一天，如果我们修改了密码的校验逻辑，比如，允许密码包含大写字母，允许密码的长度为 8 到 64 个字符，那这个时候，`isValidUserName()` 和 `isValidPassword()` 的实现逻辑就会不相同。我们就要把合并后的函数，重新拆成合并前的那两个函数。

尽管代码的实现逻辑是相同的，但语义不同，我们判定它并不违反 DRY 原则。对于包含重复代码的问题，我们可以通过抽象成更细粒度函数的方式来解决。比如将校验只包含 `a~z`、`0~9`、`dot` 的逻辑封装成 `boolean onlyContains(String str, String charlist)` 函数。




## 功能语义重复

现在我们再来看另外一个例子。在同一个项目代码中有下面两个函数：`isValidIp()` 和 `checkIfIpValid()`。尽管两个函数的命名不同，实现逻辑不同，但功能是相同的，都是用来判定 IP 地址是否合法的。

之所以在同一个项目中会有两个功能相同的函数，那是因为这两个函数是由两个不同的同事开发的，其中一个同事在不知道已经有了 `isValidIp()` 的情况下，自己又定义并实现了同样用来校验 IP 地址是否合法的 `checkIfIpValid()` 函数。

那在同一项目代码中，存在如下两个函数，是否违反 DRY 原则呢？

 复制代码

```
1 public boolean isValidIp(String ipAddress) {
2     if (StringUtils.isBlank(ipAddress)) return false;
3     String regex = "(1\\d{2}|2[0-4]\\d|25[0-5]|1-9)\\d|[1-9])\\.|"
4         + "(1\\d{2}|2[0-4]\\d|25[0-5]|1-9)\\d|\\d)\\.|"
5         + "(1\\d{2}|2[0-4]\\d|25[0-5]|1-9)\\d|\\d)\\.|"
6         + "(1\\d{2}|2[0-4]\\d|25[0-5]|1-9)\\d|\\d)$";
7     return ipAddress.matches(regex);
8 }
9
10 public boolean checkIfIpValid(String ipAddress) {
11     if (StringUtils.isBlank(ipAddress)) return false;
12     String[] ipUnits = StringUtils.split(ipAddress, '.');
13     if (ipUnits.length != 4) {
14         return false;
15     }
16     for (int i = 0; i < 4; ++i) {
17         int ipUnitIntValue;
18         try {
19             ipUnitIntValue = Integer.parseInt(ipUnits[i]);
20         } catch (NumberFormatException e) {
21             return false;
22         }
23         if (ipUnitIntValue < 0 || ipUnitIntValue > 255) {
24             return false;
25         }
26         if (i == 0 && ipUnitIntValue == 0) {
27             return false;
28         }
29     }
30     return true;
31 }
```

这个例子跟上个例子正好相反。上一个例子是代码实现逻辑重复，但语义不重复，我们并不认为它违反了 DRY 原则。而在这个例子中，尽管两段代码的实现逻辑不重复，但语义重复，也就是功能重复，我们认为它违反了 DRY 原则。我们应该在项目中，统一一种实现思路，所有用到判断 IP 地址是否合法的地方，都统一调用同一个函数。

假设我们不统一实现思路，那有些地方调用了 `isValidIp()` 函数，有些地方又调用了 `checkIfIpValid()` 函数，这就会导致代码看起来很奇怪，相当于给代码“埋坑”，给不熟悉这部分代码的同事增加了阅读的难度。同事有可能研究了半天，觉得功能是一样的，但又有点疑惑，觉得是不是有更高深的考量，才定义了两个功能类似的函数，最终发现居然是代码设计的问题。

除此之外，如果哪天项目中 IP 地址是否合法的判定规则改变了，比如：255.255.255.255 不再被判定为合法的了，相应地，我们对 `isValidIp()` 的实现逻辑做了相应的修改，但却忘记了修改 `checkIfIpValid()` 函数。又或者，我们压根就不知道还存在一个功能相同的 `checkIfIpValid()` 函数，这样就会导致有些代码仍然使用老的 IP 地址判断逻辑，导致出现一些莫名其妙的 bug。

## 代码执行重复

前两个例子一个是实现逻辑重复，一个是语义重复，我们再来看第三个例子。其中，`UserService` 中 `login()` 函数用来校验用户登录是否成功。如果失败，就返回异常；如果成功，就返回用户信息。具体代码如下所示：

 复制代码

```
1 public class UserService {
2     private UserRepo userRepo; // 通过依赖注入或者 IOC 框架注入
3
4     public User login(String email, String password) {
5         boolean existed = userRepo.checkIfUserExisted(email, password);
6         if (!existed) {
7             // ... throw AuthenticationFailureException...
8         }
9         User user = userRepo.getUserByEmail(email);
10        return user;
11    }
12 }
13
14 public class UserRepo {
15     public boolean checkIfUserExisted(String email, String password) {
16         if (!EmailValidation.validate(email)) {
17
```

```

18     // ... throw InvalidEmailException...
19 }
20
21 if (!PasswordValidation.validate(password)) {
22     // ... throw InvalidPasswordException...
23 }
24
25 //...query db to check if email&password exists...
26 }
27
28 public User getUserByEmail(String email) {
29     if (!EmailValidation.validate(email)) {
30         // ... throw InvalidEmailException...
31     }
32     //...query db to get user by email...
33 }
}

```

上面这段代码，既没有逻辑重复，也没有语义重复，但仍然违反了 DRY 原则。这是因为代码中存在“执行重复”。我们一块儿来看下，到底哪些代码被重复执行了？

重复执行最明显的一个地方，就是在 login() 函数中，email 的校验逻辑被执行了两次。一次是在调用 checkIfUserExisted() 函数的时候，另一次是调用 getUserByEmail() 函数的时候。这个问题解决起来比较简单，我们只需要将校验逻辑从 UserRepo 中移除，统一放到 UserService 中就可以了。

除此之外，代码中还有一处比较隐蔽的执行重复，不知道你发现了没有？实际上，login() 函数并不需要调用 checkIfUserExisted() 函数，只需要调用一次 getUserByEmail() 函数，从数据库中获取到用户的 email、password 等信息，然后跟用户输入的 email、password 信息做对比，依次判断是否登录成功。

实际上，这样的优化是很有必要的。因为 checkIfUserExisted() 函数和 getUserByEmail() 函数都需要查询数据库，而数据库这类的 I/O 操作是比较耗时的。我们在写代码的时候，应当尽量减少这类 I/O 操作。

按照刚刚的修改思路，我们把代码重构一下，移除“重复执行”的代码，只校验一次 email 和 password，并且只查询一次数据库。重构之后的代码如下所示：

```
1 public class UserService {
2     private UserRepo userRepo; // 通过依赖注入或者 IOC 框架注入
3
4     public User login(String email, String password) {
5         if (!EmailValidation.validate(email)) {
6             // ... throw InvalidEmailException...
7         }
8         if (!PasswordValidation.validate(password)) {
9             // ... throw InvalidPasswordException...
10        }
11        User user = userRepo.getUserByEmail(email);
12        if (user == null || !password.equals(user.getPassword())) {
13            // ... throw AuthenticationFailureException...
14        }
15        return user;
16    }
17 }
18
19 public class UserRepo {
20     public boolean checkIfUserExisted(String email, String password) {
21         //...query db to check if email&password exists
22     }
23
24     public User getUserByEmail(String email) {
25         //...query db to get user by email...
26     }
27 }
```

## 代码复用性 (Code Reusability)

在专栏的最开始，我们有提到，代码的复用性是评判代码质量的一个非常重要的标准。当时只是点到为止，没有展开讲解，今天，我再带你深入地学习一下这个知识点。

### 什么是代码的复用性？

我们首先来区分三个概念：代码复用性 (Code Reusability)、代码复用 (Code Resue) 和 DRY 原则。

代码复用表示一种行为：我们在开发新功能的时候，尽量复用已经存在的代码。代码的可复用性表示一段代码可被复用的特性或能力：我们在编写代码的时候，让代码尽量可复用。DRY 原则是一条原则：不要写重复的代码。从定义描述上，它们好像有点类似，但深究起来，三者的区别还是蛮大的。



**首先，“不重复”并不代表“可复用”。**在一个项目代码中，可能不存在任何重复的代码，但也并不表示里面有可复用的代码，不重复和可复用完全是两个概念。所以，从这个角度来说，DRY 原则跟代码的可复用性讲的是两回事。

**其次，“复用”和“可复用性”关注角度不同。**代码“可复用性”是从代码开发者的角度来讲的，“复用”是从代码使用者的角度来讲的。比如，A 同事编写了一个 `UrlUtils` 类，代码的“可复用性”很好。B 同事在开发新功能的时候，直接“复用”A 同事编写的 `UrlUtils` 类。

尽管复用、可复用性、DRY 原则这三者从理解上有所区别，但实际上要达到的目的都是类似的，都是为了减少代码量，提高代码的可读性、可维护性。除此之外，复用已经经过测试的老代码，bug 会比从零重新开发要少。

“复用”这个概念不仅可以指导细粒度的模块、类、函数的设计开发，实际上，一些框架、类库、组件等的产生也都是为了达到复用的目的。比如，Spring 框架、Google Guava 类库、UI 组件等等。

## 怎么提高代码复用性？

实际上，我们前面已经讲到过很多提高代码可复用性的手段，今天算是集中总结一下，我总结了 7 条，具体如下。

### 减少代码耦合

对于高度耦合的代码，当我们希望复用其中的一个功能，想把这个功能的代码抽取出来成为一个独立的模块、类或者函数的时候，往往会发现牵一发而动全身。移动一点代码，就要牵连到很多其他相关的代码。所以，高度耦合的代码会影响到代码的复用性，我们要尽量减少代码耦合。

### 满足单一职责原则

我们前面讲过，如果职责不够单一，模块、类设计得大而全，那依赖它的代码或者它依赖的代码就会比较多，进而增加了代码的耦合。根据上一点，也就会影响到代码的复用性。相反，越细粒度的代码，代码的通用性会越好，越容易被复用。

## 模块化

这里的“模块”，不单单指一组类构成的模块，还可以理解为单个类、函数。我们要善于将功能独立的代码，封装成模块。独立的模块就像一块一块的积木，更加容易复用，可以直接拿来搭建更加复杂的系统。

## 业务与非业务逻辑分离

越是跟业务无关的代码越是容易复用，越是针对特定业务的代码越难复用。所以，为了复用跟业务无关的代码，我们将业务和非业务逻辑代码分离，抽取成一些通用的框架、类库、组件等。

## 通用代码下沉

从分层的角度来看，越底层的代码越通用、会被越多的模块调用，越应该设计得足够可复用。一般情况下，在代码分层之后，为了避免交叉调用导致调用关系混乱，我们只允许上层代码调用下层代码及同层代码之间的调用，杜绝下层代码调用上层代码。所以，通用的代码我们尽量下沉到更下层。

## 继承、多态、抽象、封装

在讲面向对象特性的时候，我们讲到，利用继承，可以将公共的代码抽取到父类，子类复用父类的属性和方法。利用多态，我们可以动态地替换一段代码的部分逻辑，让这段代码可复用。除此之外，抽象和封装，从更加广义的层面、而非狭义的面向对象特性的层面来理解的话，越抽象、越不依赖具体的实现，越容易复用。代码封装成模块，隐藏可变的细节、暴露不变的接口，就越容易复用。

## 应用模板等设计模式

一些设计模式，也能提高代码的复用性。比如，模板模式利用了多态来实现，可以灵活地替换其中的部分代码，整个流程模板代码可复用。关于应用设计模式提高代码复用性这一部分，我们留在后面慢慢来讲解。

除了刚刚我们讲到的几点，还有一些跟编程语言相关的特性，也能提高代码的复用性，比如泛型编程等。实际上，除了上面讲到的这些方法之外，复用意识也非常重要。在写代码的时候，我们要多去思考一下，这个部分代码是否可以抽取出来，作为一个独立的模块、类或者函数供多处使用。在设计每个模块、类、函数的时候，要像设计一个外部 API 那样，去思考它的复用性。

## 辩证思考和灵活应用

实际上，编写可复用的代码并不简单。如果我们在编写代码的时候，已经有复用的需求场景，那根据复用的需求去开发可复用的代码，可能还不算难。但是，如果当下并没有复用的需求，我们只是希望现在编写的代码具有可复用的特点，能在未来某个同事开发某个新功能的时候复用得上。在这种没有具体复用需求的情况下，我们就需要去预测将来代码会如何复用，这就比较有挑战了。

实际上，除非有非常明确的复用需求，否则，为了暂时用不到的复用需求，花费太多的时间、精力，投入太多的开发成本，并不是一个值得推荐的做法。这也违反我们之前讲到的 YAGNI 原则。

除此之外，有一个著名的原则，叫作“Rule of Three”。这条原则可以用在很多行业和场景中，你可以自己去研究一下。如果把这个原则用在这里，那就是说，我们在第一次写代码的时候，如果当下没有复用的需求，而未来的复用需求也不是特别明确，并且开发可复用代码的成本比较高，那我们就不需要考虑代码的复用性。在之后我们开发新的功能的时候，发现可以复用之前写的这段代码，那我们就重构这段代码，让其变得更加可复用。

也就是说，第一次编写代码的时候，我们不考虑复用性；第二次遇到复用场景的时候，再进行重构使其复用。需要注意的是，“Rule of Three”中的“Three”并不是真的就指确切的“三”，这里就是指“二”。

## 重点回顾

今天的内容到此就讲完了。我们一块来回顾一下，你需要重点掌握的内容。

### 1.DRY 原则

我们今天讲了三种代码重复的情况：实现逻辑重复、功能语义重复、代码执行重复。实现逻辑重复，但功能语义不重复的代码，并不违反 DRY 原则。实现逻辑不重复，但功能语义重复的代码，也算是违反 DRY 原则。除此之外，代码执行重复也算是违反 DRY 原则。

## 2. 代码复用性

今天，我们讲到提高代码可复用性的一些方法，有以下 7 点。

减少代码耦合

满足单一职责原则

模块化

业务与非业务逻辑分离

通用代码下沉

继承、多态、抽象、封装

应用模板等设计模式

实际上，除了上面讲到的这些方法之外，复用意识也非常重要。在设计每个模块、类、函数的时候，要像设计一个外部 API 一样去思考它的复用性。

我们在第一次写代码的时候，如果当下没有复用的需求，而未来的复用需求也不是特别明确，并且开发可复用代码的成本比较高，那我们就不需要考虑代码的复用性。在之后开发新的功能的时候，发现可以复用之前写的这段代码，那我们就重构这段代码，让其变得更加可复用。

相比于代码的可复用性，DRY 原则适用性更强一些。我们可以不写可复用的代码，但一定不能写重复的代码。

## 课堂讨论

除了实现逻辑重复、功能语义重复、代码执行重复，你还知道有哪些其他类型的代码重复？这些代码重复是否违反 DRY 原则？

欢迎在留言区写下你的想法，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

点击参加小程序学习打卡 

## 8个月，攻克设计模式



扫一扫参与小程序打卡



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | 理论六：我为何说KISS、YAGNI原则看似简单，却经常被用错？

下一篇 22 | 理论八：如何用迪米特法则（LOD）实现“高内聚、松耦合”？

### 精选留言 (46)

 写留言



辣么大

2019-12-20

- 1、注释或者文档违反DRY
- 2、数据对象违反DRY

对于1，例如一个方法。写了好多的注释解释代码的执行逻辑，后续修改的这个方法的时候可能，忘记修改注释，造成对代码理解的困难。实际应用应该使用KISS原则，将方法写...

展开 ▾

 3

 13



岁月





2019-12-20

加油啊感觉更新太慢了一个下午就看完了..,一个星期至少更新10课吧.

3

9



啦啦啦

2019-12-20

产品经理有时候设计产品功能的时候也会重复

展开

5

5



blackhole

2019-12-20

1, 提个小问题:

“实现逻辑重复”一节的代码是不是有点问题啊?

if (!(c >= 'a' && c <= 'z') || (c >= '0' && c <= '9') || c == '.') {}似乎应该改为if (!(c >...

展开

2

2



magict4

2019-12-20

> 重复执行最明显的一个地方, 就是在 login() 函数中, email 的校验逻辑被执行了两次。一次是在调用 checkIfUserExisted() 函数的时候, 另一次是调用 getUserByEmail() 函数的时候。这个问题解决起来比较简单, 我们只需要将校验逻辑从 UserRepo 中移除, 统一放到 UserService 中就可以了。

...

展开

2

2



哈喽沃德

2019-12-20

啥时能出设计模式的教程, 我的大刀早已饥渴难耐了

展开

1

1



李小四

2019-12-20

设计模式\_20

# 作业:

想到的只有文档和注释的重复了, 比如两个不同功能的文档, 同时描写一个细节时, 可能“负责”的产品经理会各自清清楚楚地写一遍。然后: ...

展开 ∨



1



DullBird

2019-12-20

课堂讨论没有想到其他的了。

理解一下DRY, 总结就是抽取统一“逻辑”, 还有相似逻辑的简化统一, 为的就是同一“逻辑”, 维护一块地方就行了。



1



墨雨

2019-12-20

整体来说我们要做的是不写“重复”代码, 同时考虑代码的复用性, 但要避免过度设计。这几点说起来简单其实做起来还是有些难度的, 在平常写代码的时候需要多思考, 写完之后要反复审视自己的代码看看有没有可以优化的地方。说起来我感觉我还算是对代码有些追求的.....但是真的需求来了为了赶需求基本就一遍过了.....😂, 对于一些脚本代码更是过程编程, 惭愧啊

展开 ∨

2

1



黄林晴

2019-12-20

“Rule of Three”中的“Three”并不是真的就指确切的“三”, 这里就是指“二”。😂这句话看了好几遍

展开 ∨

作者回复: 😂



4

1



岁月

2019-12-23

看完最后这个Rule of three, 我感觉把可扩展填进去也是有道理的, 一开始不一定写得出扩展性很好的代码, 所以可以先简单来, 后面需求明确了再慢慢重构把代码变得更加可以扩展?





enjoylearning

2019-12-22

关键点在于语义不重复，即使里面的执行逻辑重复，也并不违反DRY原则，而是SRP的体现。单一职责不仅体现在模块级，还体现在类级别，甚至函数级别，而很多人就错误的认为可复用就是没有重复的代码执行逻辑。细粒度的DRY指函数功能不重复，宏观的DRY指层与层之间职责不重复。

展开 ∨



落叶飞逝的恋

2019-12-22

字段冗余设计

展开 ∨



守拙

2019-12-21

课堂讨论：

除了实现逻辑重复、功能语义重复、代码执行重复，你还知道有哪些其他类型的代码重复？这些代码重复是否违反 DRY 原则？

Answer: ...

展开 ∨



L 🚲 🐱

2019-12-21

注释和model 违反了 DRY 原则 注释写重复了, 或者 逻辑改了, 注释没改, model 则是 属性命名多余等等



花儿少年

2019-12-21

之前只是觉得DRY就仅仅是代码上的重复，现在终于厘清了。还有语义，功能和执行上的重复。

我们团队约定在写代码的时候，每层都需要检验参数，为了防止NPE和别人调用时出错，就会造成很多重复的校验，但是由于每层的职责不一样，很多校验也算不上完全重复，不知道这个算不算的上是代码执行重复。

展开 ∨





小建

2019-12-21

dry 、 rule of three. get

展开 ▾



小刀

2019-12-21

复用

可复用性

易复用性

DRY不要去重复



\_xcc

2019-12-21

校验IP合法性的代码我收藏了，感谢

展开 ▾



小猴子吹泡泡

2019-12-21

老师，我有一个关于DAO层代码复用的疑问，例如，在User类里有name、email、code等字段，现在我有两个方法分别会通过code去查询name和email，我是在DAO层写两个方法（getNameByCode()、getEmailByCode()）好呢还是写一个getUserByCode()，然后通过getter方法去获取name和email好？我个人一直觉得是写两个方法好，因为可以减少数据库没必要的查询，节省时间，但是这样可能导致在DAO层写的接口和SQL偏多， ...

展开 ▾

