21讲理论七：重复的代码就一定违背DRY吗如何提高代码的复用性

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?

```
public class UserAuthenticator {
  public void authenticate(String username, String password) {
    if (!isValidUsername(username)) {
      // ...throw InvalidUsernameException...
    }
    if (!isValidPassword(password)) {
      // ...throw InvalidPasswordException...
    }
    //...省略其他代码...
  }

  private boolean isValidUsername(String username) {
    // check not null, not empty
    if (StringUtils.isBlank(username)) {
      return false;
    }
    // check length: 4~64
    int length = username.length();
    if (length < 4 || length > 64) {
      return false;
    }
    // contains only lowcase characters
    if (!StringUtils.isAllLowerCase(username)) {
      return false;
    }
    // contains only a~z,0~9,dot
    for (int i = 0; i < length; ++i) {
      char c = username.charAt(i);
      if (!(c >= 'a' && c <= 'z') || (c >= '0' && c <= '9') || c == '.') {
        return false;
      }
    }
    return true;
  }

  private boolean isValidPassword(String password) {
    // check not null, not empty
    if (StringUtils.isBlank(password)) {
      return false;
    }
    // check length: 4~64
    int length = password.length();
```

```
    if (length < 4 || length > 64) {
      return false;
    }
    // contains only lowcase characters
    if (!StringUtils.isAllLowerCase(password)) {
      return false;
    }
    // contains only a~z,0~9,dot
    for (int i = 0; i < length; ++i) {
      char c = password.charAt(i);
      if (!(c >= 'a' && c <= 'z') || (c >= '0' && c <= '9') || c == '.') {
        return false;
      }
    }
    return true;
  }
}
```

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：

```
public class UserAuthenticatorV2 {

  public void authenticate(String userName, String password) {
    if (!isValidUsernameOrPassword(userName)) {
      // ...throw InvalidUsernameException...
    }

    if (!isValidUsernameOrPassword(password)) {
      // ...throw InvalidPasswordException...
    }
  }

  private boolean isValidUsernameOrPassword(String usernameOrPassword) {
    //省略实现逻辑
    //跟原来的isValidUsername()或isValidPassword()的实现逻辑一样...
    return true;
  }
}
```

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符，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?

```
public boolean isValidIp(String ipAddress) {
  if (StringUtils.isBlank(ipAddress)) return false;
  String regex = "^(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|[1-9])\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)\\."
          + "(1\\d{2}|2[0-4]\\d|25[0-5]|[1-9]\\d|\\d)$";
  return ipAddress.matches(regex);
}

public boolean checkIfIpValid(String ipAddress) {
  if (StringUtils.isBlank(ipAddress)) return false;
  String[] ipUnits = StringUtils.split(ipAddress, '.');
  if (ipUnits.length != 4) {
    return false;
  }
  for (int i = 0; i < 4; ++i) {
    int ipUnitIntValue;
    try {
      ipUnitIntValue = Integer.parseInt(ipUnits[i]);
    } catch (NumberFormatException e) {
      return false;
    }
    if (ipUnitIntValue < 0 || ipUnitIntValue > 255) {
      return false;
    }
    if (i == 0 && ipUnitIntValue == 0) {
      return false;
    }
  }
  return true;
}
```

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：

```
public class UserService {
  private UserRepo userRepo;//通过依赖注入或者IOC框架注入

  public User login(String email, String password) {
    boolean existed = userRepo.checkIfUserExisted(email, password);
    if (!existed) {
      // ... throw AuthenticationFailureException...
    }
    User user = userRepo.getUserByEmail(email);
    return user;
  }
}

public class UserRepo {
  public boolean checkIfUserExisted(String email, String password) {
    if (!EmailValidation.validate(email)) {
      // ... throw InvalidEmailException...
    }

    if (!PasswordValidation.validate(password)) {
      // ... throw InvalidPasswordException...
    }

    //...query db to check if email&password exists...
  }

  public User getUserByEmail(String email) {
    if (!EmailValidation.validate(email)) {
      // ... throw InvalidEmailException...
    }
    //...query db to get user by email...
  }
}
```

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：

```
public class UserService {
  private UserRepo userRepo;//通过依赖注入或者IOC框架注入

  public User login(String email, String password) {
    if (!EmailValidation.validate(email)) {
      // ... throw InvalidEmailException...
    }
    if (!PasswordValidation.validate(password)) {
      // ... throw InvalidPasswordException...
    }
    User user = userRepo.getUserByEmail(email);
    if (user == null || !password.equals(user.getPassword()) {
      // ... throw AuthenticationFailureException...
    }
    return user;
  }
}

public class UserRepo {
  public boolean checkIfUserExisted(String email, String password) {
    //...query db to check if email&password exists
  }

  public User getUserByEmail(String email) {
    //...query db to get user by email...
  }
}
```

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。

精选留言

辣么大

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，将方法写的见名知意，尽量容易阅读。注释不必过多。

对于2、例如类
class User
String id
Date registerDate
int age
int registedDays
其中 age可以由身份证号码算出来，而且每年都会递增。注册会员多少天了，也可以算出来。所以是不是可以考虑，数据只存储id和注册时间。其余两个字段可以算出来。

补充：
DRY不是只代码重复，而是“知识”的重复，意思是指业务逻辑。例如由于沟通不足，两个程序员用两种不同的方法实现同样功能的校验。
DRY is about the duplication of knowledge, of intent. It’s about expressing the same thing in two different places, possibly in two totally different ways.

当代码的某些地方必须更改时，你是否发现自己在多个位置以多种不同格式进行了更改？ 你是否需要更改代码和文档，或更改包含其的数据库架构和结构，或者…？ 如果是这样，则您的代码不是DRY。

when some single facet of the code has to change, do you find yourself making that change in multiple places, and in multiple different formats? Do you have to change code and documentation, or a database schema and a structure that holds it, or…? If so, your code isn’t DRY.

参考：

The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition (2nd Edition)
2019-12-20 09:04


岁月
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.
2019-12-20 19:30


啦啦啦
产品经理有时候设计产品功能的时候也会重复
2019-12-20 08:46


magict4
> 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。

这样处理会有一个问题：如果别的 xxxService 也需要用到 UserRepo，而且没有对 email 跟 password 进行校验，直接调用了 UserRepo.checkIfUserExisted() ，会产生异常。

一种方法是约定，所有关于 User 的操作都只能通过 UserService 进行，不能直接调用 UserRepo。

另一种方法是“强制” xxxService 进行校验。我们可以把 UserRepo.checkIfUserExisted 的方法签名改成

UserRepo.checkIfUserExisted(Email email, Password password)

并且把 validation 的逻辑封装在 Email 跟 Password 类的构造函数中。这样 xxxService 必须先把 email 跟 password 从 String 类型转成对应的 Email/Password 类，才能调用 UserRepo，validation 的逻辑会在转换中被强制执行。
2019-12-20 07:36


李小四
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：
- 看的人就会懵，(描述相同时)写了两个地方，看一下是不是还有别的地方有描述；(描述不同时)，应该以哪个为准。
- 改的人也会懵，很容易忘记修改更新，更何况文档不更新程序又不会报错。。。

#感想：
回到“少干活 和 少犯错”的宗旨，重复的代码不仅写的时候会多些一遍，改的时候也要多看很多地方，多想很多差异性，多改很多地方，这样就违背了“少干活”；改的时候，容易忘记一些地方，维护多种逻辑实现的同一个逻辑，也容易疏忽而出错，这样就违背了“少出错”。

说句题外话，文中提到“Rule of Three”时，原来外国人也用“三”表示多个，而且表示的还是2个。。。
2019-12-20 15:28


岁月
看完最后这个Rlue 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?
2019-12-23 11:39


blacknhole
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 >= 'a' && c <= 'z') || (c >= '0' && c <= '9') || c ==

'.')) {}。

2，说个小体会：

“可复用性、可扩展性、可维护性……”与“复用性、扩展性、维护性……”，加不加“可”，其实没有本质差别。我以为，在通常的语境中（或者几乎任何情况下），两者都是可以通用的。

比如，可复用性高，说明能够复用，与当前是否已经复用无关。复用性高，是指当前已经大量复用，说明在这之前可复用性高。已经大量复用时，依然可以更多地复用，也即：复用性高，意味着可复用性依然高。

通常的语境中，也即通常提到“复用性”时，人们几乎只关注能不能复用，而不是已经复用了多少。所以，可以认为，可复用性高等同于复用性高。

2019-12-20 17:14

作者回复

嗯嗯 确实有点问题 已经安排在改了

2020-01-27 18:51

黄林晴

“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

2019-12-20 08:20

作者回复

2019-12-20 08:28

花颜

老师，我有个问题，在大型多人协作项目当中，类、功能都是分散给不同的人开发的，不同的开发者质量良莠不齐，而实现逻辑重复有代码重复率校验工具可以做检测，而功能语义重复和代码执行重复其实不是那么容易能够发现，即使通过有效的code Review，有没有什么工具可以辅助我们查找功能语义重复和代码执行重复这两类重复，以及在大型团队项目下，如何应用这些原则呢？毕竟靠自觉总是很难的

2019-12-30 22:40

作者回复

好像只能靠人本身 工具很难去断定是否符合某一设计原则

2020-01-03 08:14

AaronChun

数据库转换对象beanDb和数据展现beanVo，从属性定义上来看可能存在大量重复，但从业务或系统分层来看，却是职责明确，功能单一的对象，所以这并不违反DRY原则。相反如果将两者共性部分抽离提取，后期倘若业务变更，修改就会牵扯到前台和后台，不符合单一职责和接口隔离原则。

2019-12-24 18:57

哈喽沃德

啥时能出设计模式的教程，我的大刀早已饥渴难耐了

2019-12-20 16:59

DullBird

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。

2019-12-20 12:28

plain

设计每个模块、类、函数，都要像设计外部api一样去思考，隐藏可变的细节、暴露不变的接口。

2019-12-20 11:18

墨雨

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

2019-12-20 08:44

小晏子

老师概括的很全面了，提一个看看，

数据定义重复，比如数据库里定义了两个schema几乎相同的数据表，然后数据表映射到代码里的结构体或xml也几乎相同，没有把公共部分剥离。

2019-12-20 08:33

张文浩@有赞

在开发新项目时，需要建新表，然后每次都需要在工程里，对新加的Repository增加add、select、update方法，逻辑大都差不多，这个感觉也是一个重复，有什么好的方法可以解决这类重复吗?

2020-02-09 16:33

杨成龙

老师，咨询一个场景问题。

问题描述：我经常在写代码的时候碰到这样的场景，在service类有多个方法，做同样的状态校验，然后我就把状态校验抽取成了一个公共方法放在本service类里，但是后来发现其他的service类也有这样的校验，我就想把这个校验方法抽出去，但是又觉得这个代码无处安放。。。

困惑：1、如果在这些service类之外，独立一个类出来的话，又不知道如何命名这个类，也不知道这个类应该放在哪个包里?

2、还是说把这个校验方法放到对应的业务bo或者domain里呢?

这种场景老师建议如何处理呢?

2020-01-29 10:11

helloworld

2. 给我印象深刻的一点：『实现逻辑重复，但功能语义不重复的代码，并不违反 DRY 原则』。主要考虑的是如果将这些代码合并后，在将来如果各自的业务逻辑修改时，代码不够灵活

3. 在写代码的过程中不要刻意地去做代码的复用性设计，当遇到代码复用的问题时再进行代码复用性的设计

2020-01-15 10:28

evalcony

DRY，我的理解是，让代码在哪个角度（维度）上进行解耦/独立/正交。

使代码“积木化”。

2020-01-08 01:37

柴柴777

我 之前就有个问题就说 我们如果用了组件化 每个模块算是单独的 尽管可能会写一个单独的util模块但是 还是存在着 重复代码,但是这些重复代码不在一个module里,那这样的到底算不算重复呢,,这些简单的部分的少量的重复不值得去单独加一个module

2020-01-06 14:46

作者回复

有点重复是问题不大的，开发软件本身就没有绝对的对与错，也不是非黑即白，怎么合适怎么来，怎么舒服怎么来，不行就再重构。

2020-01-07 14:46