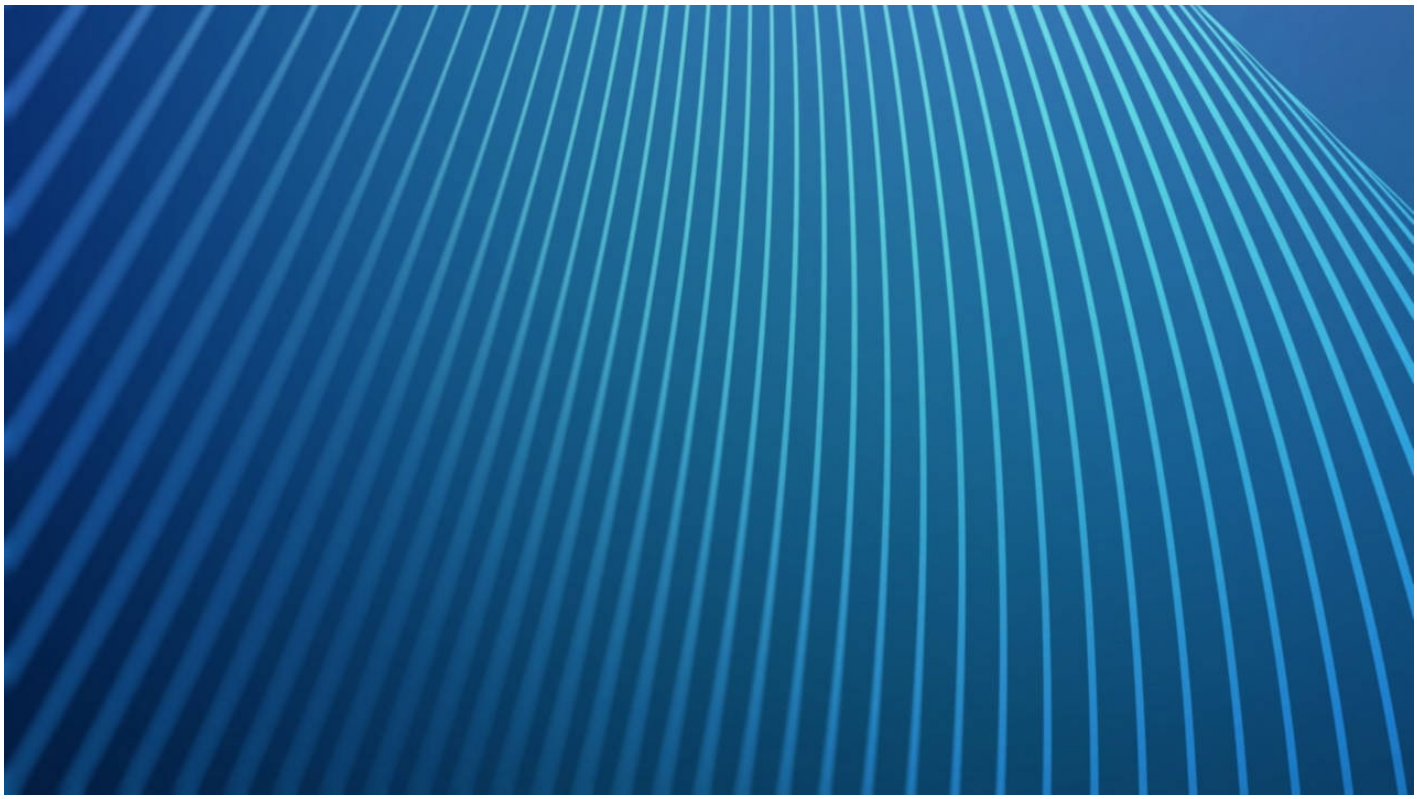
33讲理论五：让你最快速地改善代码质量的20条编程规范（下）



上两节课，我们讲了命名和注释、代码风格，今天我们来讲一些比较实用的编程技巧，帮你切实地提高代码可读性。这部分技巧比较琐碎，也很难罗列全面，我仅仅总结了一些我认为比较关键的，更多的技巧需要你在实践中自己慢慢总结、积累。

话不多说，让我们正式开始今天的学习吧！

1.把代码分割成更小的单元块

大部分人阅读代码的习惯都是，先看整体再看细节。所以，我们要有模块化和抽象思维，善于将大块的复杂逻辑提炼成类或者函数，屏蔽掉细节，让阅读代码的人不至于迷失在细节中，这样能极大地提高代码的可读性。不过，只有代码逻辑比较复杂的时候，我们其实才建议提炼类或者函数。毕竟如果提炼出的函数只包含两三行代码，在阅读代码的时候，还得跳过去看一下，这样反倒增加了阅读成本。

这里我举一个例子来进一步解释一下。代码具体如下所示。重构前，在invest()函数中，最开始的那段关于时间处理的代码，是不是很难看懂？重构之后，我们将这部分逻辑抽象成一个函数，并且命名为isLastDayOfMonth，从名字就能清晰地了解它的功能，判断今天是不是当月的最后一天。这里，我们就是通过将复杂的逻辑代码提炼成函数，大大提高了代码的可读性。

```
// 重构前的代码
public void invest(long userId, long financialProductId) {
  Calendar calendar = Calendar.getInstance();
  calendar.setTime(date);
  calendar.set(Calendar.DATE, (calendar.get(Calendar.DATE) + 1));
  if (calendar.get(Calendar.DAY_OF_MONTH) == 1) {
    return;
  }
  //...
}

// 重构后的代码：提炼函数之后逻辑更加清晰
public void invest(long userId, long financialProductId) {
  if (isLastDayOfMonth(new Date())) {
    return;
  }
  //...
}

public boolean isLastDayOfMonth(Date date) {
  Calendar calendar = Calendar.getInstance();
  calendar.setTime(date);
  calendar.set(Calendar.DATE, (calendar.get(Calendar.DATE) + 1));
  if (calendar.get(Calendar.DAY_OF_MONTH) == 1) {
   return true;
  }
  return false;
}
```

2.避免函数参数过多

我个人觉得，函数包含3、4个参数的时候还是能接受的，大于等于5个的时候，我们就觉得参数有点过多了，会影响到代码的可读性，使用起来也不方便。针对参数过多的情况，一般有2种处理方法。

- 考虑函数是否职责单一，是否能通过拆分成多个函数的方式来减少参数。示例代码如下所示：

```
public User getUser(String username, String telephone, String email);

// 拆分成多个函数
public User getUserByUsername(String username);
public User getUserByTelephone(String telephone);
public User getUserByEmail(String email);
```

- 将函数的参数封装成对象。示例代码如下所示：

```
public void postBlog(String title, String summary, String keywords, String content, String category, long autho

// 将参数封装成对象
public class Blog {
  private String title;
  private String summary;
  private String keywords;
  private Strint content;
  private String category;
  private long authorId;
}
public void postBlog(Blog blog);
```

除此之外，如果函数是对外暴露的远程接口，将参数封装成对象，还可以提高接口的兼容性。在往接口中添加新的参数的时候，老的远程接口调用者有可能就不需要修改代码来兼容新的接口了。

3.勿用函数参数来控制逻辑

不要在函数中使用布尔类型的标识参数来控制内部逻辑，true的时候走这块逻辑，false的时候走另一块逻辑。这明显违背了单一职责原则和接口隔离原则。我建议将其拆成两个函数，可读性上也要更好。我举个例子来说明一下。

```
public void buyCourse(long userId, long courseId, boolean isVip);

// 将其拆分成两个函数
public void buyCourse(long userId, long courseId);
public void buyCourseForVip(long userId, long courseId);
```

不过，如果函数是private私有函数，影响范围有限，或者拆分之后的两个函数经常同时被调用，我们可以酌情考虑保留标识参数。示例代码如下所示：

```
// 拆分成两个函数的调用方式
boolean isVip = false;
//...省略其他逻辑...
if (isVip) {
  buyCourseForVip(userId, courseId);
} else {
  buyCourse(userId, courseId);
}

// 保留标识参数的调用方式更加简洁
boolean isVip = false;
//...省略其他逻辑...
buyCourse(userId, courseId, isVip);
```

除了布尔类型作为标识参数来控制逻辑的情况外，还有一种“根据参数是否为null”来控制逻辑的情况。针对这种情况，我们也应该将其拆分成多个函数。拆分之后的函数职责更明确，不容易用错。具体代码示例如下所示：

```
public List<Transaction> selectTransactions(Long userId, Date startDate, Date endDate) {
  if (startDate != null && endDate != null) {
    // 查询两个时间区间的transactions
  }
  if (startDate != null && endDate == null) {
    // 查询startDate之后的所有transactions
  }
  if (startDate == null && endDate != null) {
    // 查询endDate之前的所有transactions
  }
  if (startDate == null && endDate == null) {
    // 查询所有的transactions
  }
}

// 拆分成多个public函数，更加清晰、易用
public List<Transaction> selectTransactionsBetween(Long userId, Date startDate, Date endDate) {
  return selectTransactions(userId, startDate, endDate);
}

public List<Transaction> selectTransactionsStartWith(Long userId, Date startDate) {
  return selectTransactions(userId, startDate, null);
}

public List<Transaction> selectTransactionsEndWith(Long userId, Date endDate) {
  return selectTransactions(userId, null, endDate);
}

public List<Transaction> selectAllTransactions(Long userId) {
  return selectTransactions(userId, null, null);
}

private List<Transaction> selectTransactions(Long userId, Date startDate, Date endDate) {
  // ...
}
```

4.函数设计要职责单一

我们在前面讲到单一职责原则的时候，针对的是类、模块这样的应用对象。实际上，对于函数的设计来说，更要满足单一职责原则。相对于类和模块，函数的粒度比较小，代码行数少，所以在应用单一职责原则的时候，没有像应用到类或者模块那样模棱两可，能多单一就多单一。

具体的代码示例如下所示：

```
public boolean checkUserIfExisting(String telephone, String username, String email)  {
  if (!StringUtils.isBlank(telephone)) {
    User user = userRepo.selectUserByTelephone(telephone);
    return user != null;
  }

  if (!StringUtils.isBlank(username)) {
    User user = userRepo.selectUserByUsername(username);
    return user != null;
  }

  if (!StringUtils.isBlank(email)) {
    User user = userRepo.selectUserByEmail(email);
    return user != null;
  }

  return false;
}

// 拆分成三个函数
public boolean checkUserIfExistingByTelephone(String telephone);
public boolean checkUserIfExistingByUsername(String username);
public boolean checkUserIfExistingByEmail(String email);
```

5.移除过深的嵌套层次

代码嵌套层次过深往往是因为if-else、switch-case、for循环过度嵌套导致的。我个人建议，嵌套最好不超过两层，超过两层之后就要思考一下是否可以减少嵌套。过深的嵌套本身理解起来就比较费劲，除此之外，嵌套过深很容易因为代码多次缩进，导致嵌套内部的语句超过一行的长度而折成两行，影响代码的整洁。

解决嵌套过深的方法也比较成熟，有下面4种常见的思路。

- 去掉多余的if或else语句。代码示例如下所示：

```
// 示例一
public double caculateTotalAmount(List<Order> orders) {
  if (orders == null || orders.isEmpty()) {
    return 0.0;
  } else { // 此处的else可以去掉
    double amount = 0.0;
    for (Order order : orders) {
      if (order != null) {
        amount += (order.getCount() * order.getPrice());
      }
    }
    return amount;
  }
}

// 示例二
public List<String> matchStrings(List<String> strList,String substr) {
  List<String> matchedStrings = new ArrayList<>();
  if (strList != null && substr != null) {
    for (String str : strList) {
      if (str != null) { // 跟下面的if语句可以合并在一起
        if (str.contains(substr)) {
          matchedStrings.add(str);
        }
      }
    }
  }
  return matchedStrings;
}
```

- 使用编程语言提供的continue、break、return关键字，提前退出嵌套。代码示例如下所示：

```
// 重构前的代码
public List<String> matchStrings(List<String> strList,String substr) {
  List<String> matchedStrings = new ArrayList<>();
  if (strList != null && substr != null){
    for (String str : strList) {
      if (str != null && str.contains(substr)) {
        matchedStrings.add(str);
        // 此处还有10行代码...
      }
    }
  }
  return matchedStrings;
}

// 重构后的代码：使用continue提前退出
public List<String> matchStrings(List<String> strList,String substr) {
  List<String> matchedStrings = new ArrayList<>();
  if (strList != null && substr != null){
    for (String str : strList) {
      if (str == null || !str.contains(substr)) {
        continue;
      }
      matchedStrings.add(str);
      // 此处还有10行代码...
    }
  }
  return matchedStrings;
}
```

- 调整执行顺序来减少嵌套。具体的代码示例如下所示：

```
// 重构前的代码
public List<String> matchStrings(List<String> strList,String substr) {
  List<String> matchedStrings = new ArrayList<>();
  if (strList != null && substr != null) {
    for (String str : strList) {
      if (str != null) {
        if (str.contains(substr)) {
          matchedStrings.add(str);
        }
      }
    }
  }
  return matchedStrings;
}

// 重构后的代码：先执行判空逻辑，再执行正常逻辑
public List<String> matchStrings(List<String> strList,String substr) {
  if (strList == null || substr == null) { //先判空
    return Collections.emptyList();
  }

  List<String> matchedStrings = new ArrayList<>();
  for (String str : strList) {
    if (str != null) {
      if (str.contains(substr)) {
        matchedStrings.add(str);
      }
    }
  }
  return matchedStrings;
}
```

- 将部分嵌套逻辑封装成函数调用，以此来减少嵌套。具体的代码示例如下所示：

```
// 重构前的代码
public List<String> appendSalts(List<String> passwords) {
  if (passwords == null || passwords.isEmpty()) {
    return Collections.emptyList();
  }

  List<String> passwordsWithSalt = new ArrayList<>();
  for (String password : passwords) {
    if (password == null) {
```

```
      continue;
    }
    if (password.length() < 8) {
      // ...
    } else {
      // ...
    }
  }
  return passwordsWithSalt;
}

// 重构后的代码：将部分逻辑抽成函数
public List<String> appendSalts(List<String> passwords) {
  if (passwords == null || passwords.isEmpty()) {
    return Collections.emptyList();
  }

  List<String> passwordsWithSalt = new ArrayList<>();
  for (String password : passwords) {
    if (password == null) {
      continue;
    }
    passwordsWithSalt.add(appendSalt(password));
  }
  return passwordsWithSalt;
}

private String appendSalt(String password) {
  String passwordWithSalt = password;
  if (password.length() < 8) {
    // ...
  } else {
    // ...
  }
  return passwordWithSalt;
}
```

除此之外，常用的还有通过使用多态来替代if-else、switch-case条件判断的方法。这个思路涉及代码结构的改动，我们会在后面的章节中讲到，这里就暂时不展开说明了。

6.学会使用解释性变量

常用的用解释性变量来提高代码的可读性的情况有下面2种。

- 常量取代魔法数字。示例代码如下所示：

```
public double CalculateCircularArea(double radius) {
  return (3.1415) * radius * radius;
}

// 常量替代魔法数字
public static final Double PI = 3.1415;
public double CalculateCircularArea(double radius) {
  return PI * radius * radius;
}
```

- 使用解释性变量来解释复杂表达式。示例代码如下所示：

```
if (date.after(SUMMER_START) && date.before(SUMMER_END)) {
  // ...
} else {
  // ...
}

// 引入解释性变量后逻辑更加清晰
boolean isSummer = date.after(SUMMER_START)&&date.before(SUMMER_END);
if (isSummer) {
  // ...
} else {
  // ...
}
```

重点回顾

好了，今天的内容到此就讲完了。除了今天讲的编程技巧，前两节课我们还分别讲解了命名与注释、代码风格。现在，我们一块来回顾复习一下这三节课的重点内容。

1.关于命名

- 命名的关键是能准确达意。对于不同作用域的命名，我们可以适当地选择不同的长度。
- 我们可以借助类的信息来简化属性、函数的命名，利用函数的信息来简化函数参数的命名。
- 命名要可读、可搜索。不要使用生僻的、不好读的英文单词来命名。命名要符合项目的统一规范，也不要用些反直觉的命名。
- 接口有两种命名方式：一种是在接口中带前缀“I”；另一种是在接口的实现类中带后缀“Impl”。对于抽象类的命名，也有两种方式，一种是带上前缀“Abstract”，一种是不带前缀。这两种命名方式都可以，关键是要在项目中统一。

2.关于注释

- 注释的内容主要包含这样三个方面：做什么、为什么、怎么做。对于一些复杂的类和接口，我们可能还需要写明“如何用”。
- 类和函数一定要写注释，而且要写得尽可能全面详细。函数内部的注释要相对少一些，一般都是靠好的命名、提炼函数、

解释性变量、总结性注释来提高代码可读性。

3.关于代码风格

- 函数、类多大才合适？函数的代码行数不要超过一屏幕的大小，比如50行。类的大小限制比较难确定。
- 一行代码多长最合适？最好不要超过IDE的显示宽度。当然，也不能太小，否则会导致很多稍微长点的语句被折成两行，也会影响到代码的整洁，不利于阅读。
- 善用空行分割单元块。对于比较长的函数，为了让逻辑更加清晰，可以使用空行来分割各个代码块。
- 四格缩进还是两格缩进？我个人比较推荐使用两格缩进，这样可以节省空间，尤其是在代码嵌套层次比较深的情况下。不管是用两格缩进还是四格缩进，一定不要用tab键缩进。
- 大括号是否要另起一行？将大括号放到跟上一条语句同一行，可以节省代码行数。但是将大括号另起新的一行的方式，左右括号可以垂直对齐，哪些代码属于哪一个代码块，更加一目了然。
- 类中成员怎么排列？在Google Java编程规范中，依赖类按照字母序从小到大排列。类中先写成员变量后写函数。成员变量之间或函数之间，先写静态成员变量或函数，后写普通变量或函数，并且按照作用域大小依次排列。

4.关于编码技巧

- 将复杂的逻辑提炼拆分成函数和类。
- 通过拆分成多个函数或将参数封装为对象的方式，来处理参数过多的情况。
- 函数中不要使用参数来做代码执行逻辑的控制。
- 函数设计要职责单一。
- 移除过深的嵌套层次，方法包括：去掉多余的if或else语句，使用continue、break、return关键字提前退出嵌套，调整执行顺序来减少嵌套，将部分嵌套逻辑抽象成函数。
- 用字面常量取代魔法数。
- 用解释性变量来解释复杂表达式，以此提高代码可读性。

5.统一编码规范

除了这三节讲到的比较细节的知识点之外，最后，还有一条非常重要的，那就是，项目、团队，甚至公司，一定要制定统一的编码规范，并且通过Code Review督促执行，这对提高代码质量有立竿见影的效果。

课堂讨论

到此为止，我们整个20条编码规范就讲完了。不知道你掌握了多少呢？除了今天我提到的这些，还有哪些其他的编程技巧，可以明显改善代码的可读性？

试着在留言区总结罗列一下，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

黄林晴

打卡

明天最后一天上班

就放假了

2020-01-17 00:04

再见孙悟空

不要在函数中使用布尔类型的标识参数来控制内部逻辑，true 的时候走这块逻辑，false 的时候走另一块逻辑。这明显违背了单一职责原则和接口隔离原则。我建议将其拆成两个函数，可读性上也要更好。这个深有感触

2020-01-17 08:23

老师晚上好、关于代码规范这块，是不是有好的Java开发脚手架推荐呢？我发现公司的代码没有统一的脚手架，各小组重复造

轮子，想规范化这块，但又不知道有哪些通用的脚手架。

2020-01-17 19:05

作者回复

可以看下这篇文章：

https://mp.weixin.qq.com/s/0eOm3dBOlFUy8Si1_k7OAw

代码中的很多低级质量问题不需要人工去审查，java开发有很多现成的工具可以使用，比如：checkstyle，findbugs, pmd, jacaco, sonar等。

Checkstyle,findbugs,pmd是静态代码分析工具，通过分析源代码或者字节码，找出代码的缺陷，比如参数不匹配，有歧义的嵌套语句，错误的递归，非法计算，可能出现的空指针引用等等。三者都可以集成到gradle等构建工具中。

Jacoco是一种单元测试覆盖率统计工具，也可以集成到gradle等构建工具中，可以生成漂亮的测试覆盖率统计报表，同时Eclipse提供了插件可以EclEmma可以直观的在IDE中查看单元测试的覆盖情况。

Sonar Sonar 是一个用于代码质量管理的平台。可以在一个统一的平台上显示管理静态分析，单元测试覆盖率等质量报告。

2020-01-27 17:46

青青子衿

个人以为还有善用和合理运用各个编程语言提供的语法糖和语言特性。比如Java开发，工作中有的老程序员不喜欢不适应lambda表达式，实际上合理恰当的使用lambda表达式可以让代码简洁明了

2020-01-17 09:41

linlong

一般大公司都有自己的编程规范，但执行的效果取决于commiter，而最终还是项目交付进度决定的。

2020-01-17 00:44

守拙

课堂讨论:

简单说一个本人常用的改善项目可读性的方法:
在每一个module/package下编写一个description.md,简要说明是做什么的,有哪些需要注意的地方.

不会花很多时间,但可以提高项目整体的可维护性.

2020-01-17 11:57

程斌

作为一名phper，这里有很多话想说，但是最后汇成一句话，没有什么参数不是一个数组不能解决的。解决函数嵌套那块，挺实用的。

2020-01-17 08:20

Jxin

1.先提问题：
第一块代码里面，存在一点瑕疵:if (calendar.get(Calendar.DAY_OF_MONTH) == 1) { return true; } return false;
直接 return calendar.get(Calendar.DAY_OF_MONTH) == 1 ; 即可。
2.请老师谈谈你的看法
A.boolean isSummer = date.after(SUMMER_START)&&date.before(SUMMER_END);if (isSummer){};
这个场景是定义“isSummer”这个临时变量，还是if(date.after(SUMMER_START)&&date.before(SUMMER_END)){};好点。看过<重构>第三版，里面其实偏向于用函数代替临时变量（变量是魔鬼）。但这可能就会有这种if里面包含比较长的函数调用的场景，可读性其实不好，有点做了两件事的味道。但在代码重构上是比较好的，毕竟没有变量滥用带来的不确定性。拿捏不准，我最后是跟着<重构>的思路走。但这里特请栏主谈谈自己的看法。

B.boolean isSummer = date.after(SUMMER_START)&&date.before(SUMMER_END); 是否需要写成final boolean isSummer。我的习惯对不可变临时变量都会加final，事实上我基本没有可变临时变量，对可变临时变量很敏感。final会导致语句行变长，但能规范代码，具有明确语义，方便其他人阅读和扩展（约束了行为）。这个也拿捏不准，栏主怎么看?

C.类中成员属性按字母大小顺序排列。这个感觉不是很合理。拿订单类为例，我会让金额相关字段，地址相关字段，和状态相关字段分隔开各自聚合在一块。这时候就没办法按字母大小排，但语义更强。当然，对金额和地址字段，其实属于值对象，可以单独成类（存对象序列化）。但老项目难有这种设计，往往是一张表平级包含一切。所以这个按大小排序的规范，感觉太“任性”了。

3.其他编程规范，篇幅有限，而且是死的东西，不罗列了。感兴趣的同学有时间看看<Effective java>（一礼拜），没时间就看看<阿里开发手册>（2小时）。平时工作重视Sonar的每个告警，慢慢积累就好了。

2020-01-18 00:11

失火的夏天

for里面有时候会出现下标0的特殊判断，这个时候就把0下标单独拉出去玩，for从下标1开始。

我发现我的代码风格居然和争哥有点像，我仿佛在膨胀

2020-01-17 08:48

Monday

为没有代码规范的我们项目，默哀三分钟。

2020-02-11 21:46

batman

老师公司制定的统一编码规范长什么样子，能不能供大家学习学习

2020-01-30 22:35

桂城老托尼

感谢争哥分享

文中提交的技巧都是实际工作中code的原则，可以作为CR时代码规范项的参考标准。

以前经常踩 问题3 的，主要理论依据就是对外隐藏更多的细节，但违反了单一职责。

还有更多的代码规约方面的， Google Java代码规约 和 Alibaba Java 代码开发规范 其实都可以作为案头必备手册了，安利一下。

2020-01-18 10:13

李小四

设计模式_33

使用参数作为控制逻辑，这一点深有感触，除了故意设计成这样，还有一些是改成这样的(不想改程序结构，或者不能改)，在原来的基础上扩展功能，这样加一个用于控制逻辑的参数，程序就分成了两部分；如果后面再加，代码分成2^n个部分，而是会有大量的重复代码，同一个逻辑要该好几个地方，很容易忘。

对于代码质量，我有些个人的心得就是：写完代码之后，再看看，如果发现“不舒服”的地方，多想一想。

2020-01-17 17:34

刘大明

1.命名长度问题
2.利用上下文简化命名
3.命名要可读，可搜索
4.如何命名接口和抽象类
5.注释应该怎么写
6.注释是不是越多越好
7.类和函数多大才合适
8.一行代码多长才合适
9.善于用空行分隔符
10.缩进是两格还是四格
11.大括号是否需要另起一行
12.类中成员的排列顺序
13.代码应该分割成更小的单元块
14.函数参数不要过多
15.不要用函数参数来控制逻辑

16.函数设计要职责单一
17.移除过深的嵌套
18.学会使用解释性变量
19.
20.统一编码规范
这些都是开发过程中，让代码更好的一些编码规范
我自己在项目开发过程中也会时刻注意是否符合规范。
自己在项目中还遇到很多人提交代码不写注释，
因为重构很多注释掉的代码不删除
重复代码不提取公共类
临时代码随意修改，随意提交线上等等很多代码混乱问题。
从自身做起，让代码更加整洁，提交的代码尽量减少代码的坏味道。
2020-01-17 08:21

Chen
函数中不要使用参数来做代码执行逻辑的控制。我之前写代码从来没关注到这点，学习了
2020-01-17 07:40

Joker
看到函数多参数的解决方法，就想到以前项目，项目负责人就喜欢吧多个参数封装到一个map里面，然后各种put，get，转换。。
2020-02-17 19:59

helloworld
印象深刻的有：函数不要通过参数的值的判断来执行不同的逻辑；可以将一些逻辑提取为函数减少if-else嵌套；可以通过函数重载的形式来减少函数参数的个数
2020-02-16 00:22

疯
文章中：3. 勿用函数参数来控制逻辑，文中意思尽量要拆分接口，但实际业务判断某个参数的时候会加段逻辑处理包含才其中，但是其他逻辑是一致的，拆分的话是否与DRY原则有矛盾，若接口本身是public的呢，是否需要拆分，这个度要怎么把握?
2020-02-12 12:30

wind
那个continue的例子我觉得改之前和改之后没啥区别啊?
10行代码在if里又不会走到.

还有我数来数去加上统一规范也才19条, 没有20条啊?
2020-02-12 10:04

DullBird
对于不要把控制逻辑参数暴露出来，而是拆分成多个方案。这个点。我有点疑惑。
1. 比方说有个组织查询组织下人的抽象，暴露了listUserByNode()的接口
2. 组织包括(部门-员工),(班级-学生),由于某种原因，有4张表，部门表，员工，班级，学生。
3. 如果我只想查询学生的时候 调用 listUserByNode()，传入type=student，但是由于底层表是分离的，实际上执行了另一段代码。只不过调用方认为这只是个筛选条件。

还有是最近在看Dubbo，dubbo里面用的 spi机制，我感觉也是把控制留给了参数，实现对外代码通用。只不过它是用多态实现，我们有时候方法逻辑简单。就用if...else解决了。这点不是特别能理解。
2020-02-08 17:40