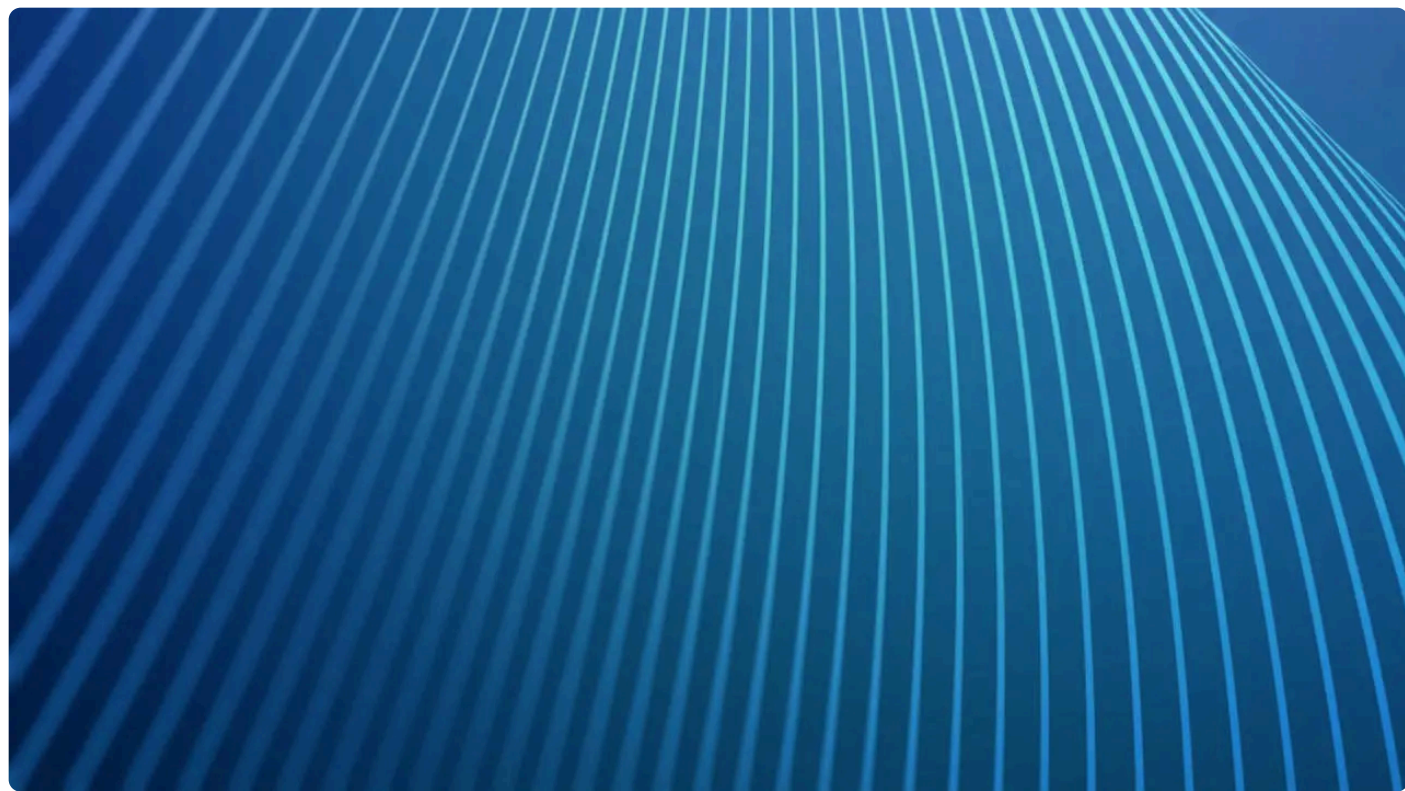


33 | 理论五：让你最快速地改善代码质量的20条编程规范（下）

王争 · 设计模式之美




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。

 复制代码

```
1 // 重构前的代码
2 public void invest(long userId, long financialProductId) {
3     Calendar calendar = Calendar.getInstance();
4     calendar.setTime(date);
5     calendar.set(Calendar.DATE, (calendar.get(Calendar.DATE) + 1));
6     if (calendar.get(Calendar.DAY_OF_MONTH) == 1) {
7         return;
8     }
9     //...
10 }
11
12 // 重构后的代码：提炼函数之后逻辑更加清晰
13 public void invest(long userId, long financialProductId) {
14     if (isLastDayOfMonth(new Date())) {
15         return;
16     }
17     //...
18 }
19
20 public boolean isLastDayOfMonth(Date date) {
21     Calendar calendar = Calendar.getInstance();
22     calendar.setTime(date);
23     calendar.set(Calendar.DATE, (calendar.get(Calendar.DATE) + 1));
24     if (calendar.get(Calendar.DAY_OF_MONTH) == 1) {
25         return true;
26     }
27     return false;
28 }
```

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，是否能够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：

 复制代码

```
1 public User getUser(String username, String telephone, String email);
2
3 // 拆分成多个函数
4 public User getUserByUsername(String username);
5 public User getUserByTelephone(String telephone);
6 public User getUserByEmail(String email);
```

将函数的参数封装成对象。示例代码如下所示：

 复制代码

```
1 public void postBlog(String title, String summary, String keywords, String content);
2
3 // 将参数封装成对象
4 public class Blog {
5     private String title;
6     private String summary;
7     private String keywords;
8     private String content;
9     private String category;
10    private long authorId;
11 }
12 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。

[📄 复制代码](#)

```
1 public void buyCourse(long userId, long courseId, boolean isVip);
2
3 // 将其拆分成两个函数
4 public void buyCourse(long userId, long courseId);
5 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：

[📄 复制代码](#)

```
1 // 拆分成两个函数的调用方式
2 boolean isVip = false;
3 //...省略其他逻辑...
4 if (isVip) {
5     buyCourseForVip(userId, courseId);
6 } else {
7     buyCourse(userId, courseId);
8 }
9
10 // 保留标识参数的调用方式更加简洁
11 boolean isVip = false;
12 //...省略其他逻辑...
13 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：

[📄 复制代码](#)

```
1 public List<Transaction> selectTransactions(Long userId, Date startDate, Date end
2     if (startDate != null && endDate != null) {
3         // 查询两个时间区间的transactions
4     }
5     if (startDate != null && endDate == null) {
6         // 查询startDate之后的所有transactions
7     }
8     if (startDate == null && endDate != null) {
9         // 查询endDate之前的所有transactions
```

```


10     }
11     if (startDate == null && endDate == null) {
12         // 查询所有的transactions
13     }
14 }
15
16 // 拆分成多个public函数，更加清晰、易用
17 public List<Transaction> selectTransactionsBetween(Long userId, Date startDate, D
18     return selectTransactions(userId, startDate, endDate);
19 }
20
21 public List<Transaction> selectTransactionsStartWith(Long userId, Date startDate)
22     return selectTransactions(userId, startDate, null);
23 }
24
25 public List<Transaction> selectTransactionsEndWith(Long userId, Date endDate) {
26     return selectTransactions(userId, null, endDate);
27 }
28
29 public List<Transaction> selectAllTransactions(Long userId) {
30     return selectTransactions(userId, null, null);
31 }
32
33 private List<Transaction> selectTransactions(Long userId, Date startDate, Date en
34     // ...
35 }

```

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：

 复制代码

```

1 public boolean checkUserIfExisting(String telephone, String username, String emai
2     if (!StringUtils.isBlank(telephone)) {
3         User user = userRepo.selectUserByTelephone(telephone);
4         return user != null;
5     }
6
7     if (!StringUtils.isBlank(username)) {

```


```
8     User user = userRepo.selectUserByUsername(username);
9     return user != null;
10 }
11
12 if (!StringUtils.isBlank(email)) {
13     User user = userRepo.selectUserByEmail(email);
14     return user != null;
15 }
16
17 return false;
18 }
19
20 // 拆分成三个函数
21 public boolean checkUserIfExistingByTelephone(String telephone);
22 public boolean checkUserIfExistingByUsername(String username);
23 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：

 复制代码


```
1 // 示例一
2 public double caculateTotalAmount(List<Order> orders) {
3     if (orders == null || orders.isEmpty()) {
4         return 0.0;
5     } else { // 此处的else可以去掉
6         double amount = 0.0;
7         for (Order order : orders) {
8             if (order != null) {
9                 amount += (order.getCount() * order.getPrice());
10            }
11        }
12        return amount;
13    }
```

```

13     }
14 }
15
16 // 示例二
17 public List<String> matchStrings(List<String> strList,String substr) {
18     List<String> matchedStrings = new ArrayList<>();
19     if (strList != null && substr != null) {
20         for (String str : strList) {
21             if (str != null) { // 跟下面的if语句可以合并在一起
22                 if (str.contains(substr)) {
23                     matchedStrings.add(str);
24                 }
25             }
26         }
27     }
28     return matchedStrings;
29 }

```

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：

 复制代码


```

1 // 重构前的代码
2 public List<String> matchStrings(List<String> strList,String substr) {
3     List<String> matchedStrings = new ArrayList<>();
4     if (strList != null && substr != null){
5         for (String str : strList) {
6             if (str != null && str.contains(substr)) {
7                 matchedStrings.add(str);
8                 // 此处还有10行代码...
9             }
10        }
11    }
12    return matchedStrings;
13 }
14
15 // 重构后的代码：使用continue提前退出
16 public List<String> matchStrings(List<String> strList,String substr) {
17     List<String> matchedStrings = new ArrayList<>();
18     if (strList != null && substr != null){
19         for (String str : strList) {
20             if (str == null || !str.contains(substr)) {
21                 continue;
22             }
23             matchedStrings.add(str);

```

```
24     // 此处还有10行代码...
25 }
26 }
27 return matchedStrings;
28 }
```

调整执行顺序来减少嵌套。具体的代码示例如下所示：

 复制代码

```
1 // 重构前的代码
2 public List<String> matchStrings(List<String> strList,String substr) {
3     List<String> matchedStrings = new ArrayList<>();
4     if (strList != null && substr != null) {
5         for (String str : strList) {
6             if (str != null) {
7                 if (str.contains(substr)) {
8                     matchedStrings.add(str);
9                 }
10            }
11        }
12    }
13    return matchedStrings;
14 }
15
16 // 重构后的代码：先执行判空逻辑，再执行正常逻辑
17 public List<String> matchStrings(List<String> strList,String substr) {
18     if (strList == null || substr == null) { //先判空
19         return Collections.emptyList();
20     }
21
22     List<String> matchedStrings= new ArrayList<>();
23     for (String str : strList) {
24         if (str != null) {
25             if (str.contains(substr)) {
26                 matchedStrings.add(str);
27             }
28         }
29     }
30     return matchedStrings;
31 }
```

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：


```
1 // 重构前的代码
2 public List<String> appendSalts(List<String> passwords) {
3     if (passwords == null || passwords.isEmpty()) {
4         return Collections.emptyList();
5     }
6
7     List<String> passwordsWithSalt = new ArrayList<>();
8     for (String password : passwords) {
9         if (password == null) {
10             continue;
11         }
12         if (password.length() < 8) {
13             // ...
14         } else {
15             // ...
16         }
17     }
18     return passwordsWithSalt;
19 }
20
21 // 重构后的代码: 将部分逻辑抽成函数
22 public List<String> appendSalts(List<String> passwords) {
23     if (passwords == null || passwords.isEmpty()) {
24         return Collections.emptyList();
25     }
26
27     List<String> passwordsWithSalt = new ArrayList<>();
28     for (String password : passwords) {
29         if (password == null) {
30             continue;
31         }
32         passwordsWithSalt.add(appendSalt(password));
33     }
34     return passwordsWithSalt;
35 }
36
37 private String appendSalt(String password) {
38     String passwordWithSalt = password;
39     if (password.length() < 8) {
40         // ...
41     } else {
42         // ...
43     }
44     return passwordWithSalt;
45 }
```

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：

 复制代码

```
1 public double CalculateCircularArea(double radius) {
2     return (3.1415) * radius * radius;
3 }
4
5 // 常量替代魔法数字
6 public static final Double PI = 3.1415;
7 public double CalculateCircularArea(double radius) {
8     return PI * radius * radius;
9 }
```

使用解释性变量来解释复杂表达式。示例代码如下所示：

 复制代码

```
1 if (date.after(SUMMER_START) && date.before(SUMMER_END)) {
2     // ...
3 } else {
4     // ...
5 }
6
7 // 引入解释性变量后逻辑更加清晰
8 boolean isSummer = date.after(SUMMER_START)&&date.before(SUMMER_END);
9 if (isSummer) {
10    // ...
11 } else {
12    // ...
13 }
```

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，也不要有些反直觉的命名。

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。

AI智能总结

本文总结了20条编程规范，旨在帮助程序员快速改善代码质量。其中包括将代码分割成更小的单元块，避免函数参数过多，勿用函数参数来控制逻辑，以及函数设计要职责单一等实用的编程技巧。作者强调了模块化和抽象思维的重要性，提倡将复杂逻辑提炼成类或者函数，以提高代码的可读性。此外，文章还提到了处理函数参数过多的方法，包括拆分成多个函数或将参数封装成对象。同时，作者也警示不要在函数中使用布尔类型的标识参数来控制内部逻辑，而是建议将其拆分成多个函数，以提高可读性和代码质量。最后，文章强调了函数设计要职责单一的重要性，通过示例代码展示了如何将一个函数拆分成多个职责单一的函数。这些编程规范和技巧能够帮助程序员在实践中提高代码的可读性和质量，是非常实用的编程指南。文章还提到了移除过深的嵌套层次和学会使用解释性变量等编码技巧，以及统一编码规范的重要性。通过这些技巧和规范，读者可以明显改善代码的可读性，提高代码质量。

© 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。

全部留言 (90)

最新 精选



2020-01-17

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。

作者回复: 可以看下这篇文章：

https://mp.weixin.qq.com/s/0eOm3dBOIFUy8Si1_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, jacoco, 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。

共 11 条评论 >

👍 65



黄林晴

2020-06-19

老师好 现在我是开始二刷 意识到一个问题就是识不要用标记位来控制代码的执行逻辑 但拆分多个函数不也要先判断标记位再执行对应的方法吗

作者回复: 你说的没错，关键看这个标记位的判断是放到上层逻辑还是下层逻辑，理论上讲，这种“ugly”的代码尽量放到上层。



👍 5



淤白

2020-11-25

Tap键、CodeReview没有做到，别的都在无意识之间做到了，打个卡。。。

作者回复: 加油



👍 1



小情绪

2020-08-28

王争老师，我瞅了一眼Android的java层源代码，类里面的import包没有按字母从小到大，作用域也没有按大小依次排列，是不是Google并不是严格执行这套标准，还是别的原因？

作者回复: Android是收购的吧💎💎💎💎💎

共 2 条评论 >

👍 1



feifei

2020-05-20

这个if else拆的函数太多了，类就大了，类大了，同样阅读性很差，找一个方法，翻来翻去的，所以我觉得只要方法不是太长，就不必要拆开多个小的方法，老师觉得呢

作者回复: 你说的没错! 拆太细、太小的函数, 也没意思, 读代码的时候跳来跳去, 容易打断思路



Michael

2020-01-18

`public void getUser`

这个返回值和函数命名好像不是很搭哦

作者回复: 是的, 我改下, 多谢指出



记事本

2020-08-13

为什么不能用tab缩进?

作者回复: 文章中有解释



编程界的小学生

2020-01-17

// 拆分成三个函数

```
public boolean checkUserIfExistingByTelephone(String telephone);
```

```
public boolean checkUserIfExistingByUsername(String username);
```

```
public boolean checkUserIfExistingByEmail(String email);
```

这种的, 不也得判断是不是空吗? 不是空的话调用。是不是可以把判断 放到每个小方法里面, 是空就false

作者回复: 没太看懂你说的😂

共 2 条评论 >



再见孙悟空

2020-01-17

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

共 10 条评论 >

 94



黄林晴

2020-01-17

打卡

明天最后一天上班

就放假了

共 23 条评论 >

 39