

## 28 | 怎么尽量“不写”代码？

2019-03-08 范学雷 来自北京

《代码精进之路》



最有效率的编码就是少编写代码，甚至不编写代码。前面，我们讨论过避免需求膨胀和设计过度，就是减少编码的办法之一。这一次，我们讨论代码复用的问题。商业的规模依赖于可复制性，代码的质量依赖于可复用性。

比如，Java 提供了很多的类库和工具，就是为了让 Java 程序员不再编写类似的代码，直接拿来使用就可以了。

### 不要重新发明轮子

“不要重新发明轮子”，这是一个流传甚广的关于软件复用的话。如果已经有了一个轮子，可以拿来复用，就不用再重新发明一个新轮子了。**复用**，是这句话的精髓部分。

如果没有现成的轮子，我们需要造一个新的。如果造的轮子可以复用，那就再好不过了。造轮子的过程，就是我们设计和实现复用接口的过程。

我刚参加工作的时候，从事的是银行综合业务系统的研发工作。银行的业务，牵涉到大量的报表。每一个报表的生成和处理，都是一个费力的编码环节。需要大量的代码，反复调试，才能生成一张漂亮的报表。那时候，市面上也没有什么可以使用的解决方案。我有一个同事负责这方面的工作，刚开始的辛苦程度可想而知。

过了几年，我们再聊起报表业务的时候，发现他已经在报表处理方面建立了巨大的优势。这个优势，就是报表处理代码的复用。他把报表的生成和处理，提炼成了一个使用非常简单的产品。用户只要使用图形界面做些简单的配置，就能生成漂亮的报表。编写大量代码、反复调试的时代，已经一去不复返了。传统的方式需要几个月的工作量，使用这个工具几天时间就搞定了。而且，客户还可以自己定义生成什么样的报表。生成花样报表的需求依然存在，但是再也不需要大量的重复劳动了。这个产品的优势，帮助他赢得了很多重要的客户。

什么样的代码可以复用呢？**一般来说，当我们使用类似的代码或者类似的功能超过两次时，就应该考虑这样的代码是不是可以复用了。**比如，当我们拷贝粘贴一段代码时，也许会做一点微小的修改，然后用到新的代码里。这时候，我们就要考虑，这段拷贝的代码是不是可以抽象成一个方法？有了抽象出来的方法，我们就不需要把这段代码拷贝到别的地方了。如果这段代码有错误，我们也只需要修改这个方法的实现就可以了。

## 推动轮子的改进

轮子发明出来了，并不意味着这个轮子就永远没有问题了。它是需要持续改进的，比如，修改错误，修复安全问题，提高计算性能等等。

“不要重新发明轮子”这句话的另外一层意思，就是改进现有的轮子。如果发现轮子有问题，不要首先试图去重新发明一个相同的轮子，而是去改进它。

每一个可以复用的代码，特别是那些经过时间检验的接口，都踩过了很多坑，经过了多年的优化。如果我们试着重新编写一个相同的接口，一般意味着这些坑我们要重新考虑一遍，还不一定能够做得更好。

比如说吧，我们前面提到了 Java 核心类库里 String 类的设计缺陷。为了避免这样的缺陷，我们当然可以发明一个新的 MyString 类。但是，这意味着我们要维护它以保持它长久的生命

力。Java 的 String 类，有 OpenJDK 社区的强大支撑，有几十亿台设备使用，有专业的人员维护、更新和改进。而我们自己发明的 MyString 类，就很难有这样的资源和力量去维护它。

当然，我们也不能坐等轮子的改进。**如果一个可以复用的代码出了问题，我们要第一时间叫喊起来。**这对代码的维护者而言，是一个发现问题、改进代码的机会。一般来说，代码维护者，都喜欢这样的声音，并且能够及时地反馈。我们可以通过发邮件，提交 bug 等我们知道的任何渠道，让代码的维护者知晓问题的存在。这样，我们就加入了改进的过程，间接影响了代码的质量。


使用现有的轮子固然方便，但是如果它满足不了你的需求，或者你不能使用，也不要被“不要重新发明轮子”这句话绊住了脚。需要新轮子的时候，就去发明新轮子。

如果你去观察市场，每一种好东西，都可能有好几个品牌在竞争。手机不仅仅只有一个品牌，豆浆机也不仅仅只有一个型号，云服务也不仅仅由一家提供，互联网支付也有多种选择。如果仔细看，类似的产品也有很多不同的地方。不同的地方，就是不同的产品有意或者无意做的市场区隔。

## 不要重复多个轮子

市场上存在多个轮子是合理的。但是在一个软件产品中，一个单一功能，只应该有一个轮子。如果有多个相同的轮子，不仅难以维护，而且难以使用，会造成很多编码的困扰。

比如说，在 JDK 11 中，我们引入了一个通过标准名称命名已知参数的类。

 复制代码


```
1 package java.security.spec;
2
3 /**
4  * This class is used to specify any algorithm parameters that are determined
5  * by a standard name.
6  * <snipped>
7  */
8 public class NamedParameterSpec implements AlgorithmParameterSpec {
9     public NamedParameterSpec(String standardName) {
10         // snipped
11     }
12 }
```

```
13     public String getName() {
14         // snipped
15     }
16 }
```

这个类单独看，并没有什么不妥当的地方。但是，如果放在更大范围里来看，这个新添加的类就引起了不小的麻烦。这是因为还存在另外一个相似的扩展类。


而且，由于这个扩展类和它继承的类，功能几乎完全重合，带来的困扰就是，本来我们只需要一个轮子就能解决的问题，现在不得不考虑两个轮子的问题。而且，由于 `ECGenParameterSpec` 的存在，我们还可能忘记了要考虑使用更基础的 `NamedParameterSpec` 类。

问题代码：

 复制代码

```
1 @Override
2 public void initialize(AlgorithmParameterSpec params)
3     throws InvalidAlgorithmParameterException {
4     // snipped
5     if (params instanceof ECGenParameterSpec) {
6         String name = ((ECGenParameterSpec)params).getName();
7     } else {
8         throw new InvalidAlgorithmParameterException(
9             "ECParameterSpec or ECGenParameterSpec required for EC");
10    }
11    // snipped
12 }
```

正确代码：

 复制代码

```
1 @Override
2 public void initialize(AlgorithmParameterSpec params)
3     throws InvalidAlgorithmParameterException {
4     // snipped
5     if (params instanceof NamedParameterSpec) {
6         String name = ((NamedParameterSpec)params).getName();
```

```
7     } else {
8         throw new InvalidAlgorithmParameterException(
9             "ECParameterSpec or ECGenParameterSpec required for EC");
10    }
11    // snipped
12 }
```

上面的问题，是 JDK 11 引入的一个编码困扰。这个困扰，导致了很多使用的问题。由于是公开接口，它的影响，要经过好多年才能慢慢消除。也许很快，在 JDK 的某一个版本中，这个扩展的 ECGenParameterSpec 类就会被废弃掉。

## 该放手时就放手

你有没有这样的体验，一个看起来很微不足道的修改，或者没有任何问题的修改，会带来一连串的连锁反应，导致意想不到的问题出现？

前不久，OpenJDK 调整了两个方法的调用顺序。大致的修改就像下面的例子。

修改前：

 复制代码

```
1 Signature getSignature(PrivateKey privateKey,
2     AlgorithmParameterSpec signAlgParameter) throws NoSuchAlgorithmException,
3     InvalidAlgorithmParameterException, InvalidKeyException {
4
5     Signature signer = Signature.getInstance("RSASSA-PSS");
6     if (signAlgParameter != null) {
7         signer.setParameter(signAlgParameter);
8     }
9     signer.initSign(privateKey);
10
11     return signer;
12 }
```

修改后：

 复制代码

```
1 Signature getSignature(PrivateKey privateKey,
2     AlgorithmParameterSpec signAlgParameter) throws NoSuchAlgorithmException,
3     InvalidAlgorithmParameterException, InvalidKeyException {
4
5     Signature signer = Signature.getInstance("RSASSA-PSS");
6     signer.initSign(privateKey);
7     if (signAlgParameter != null) {
8         signer.setParameter(signAlgParameter);
9     }
10
11     return signer;
12 }
```

这个修改仅仅调换了一下两个方法的调用顺序。根据这两个方法的接口规范，调用顺序的修改不应该出现任何问题。然而，让人意向不到的是，这个接口的实现者，大都依赖于严格的调用顺序。修改前的调用顺序，已经使用了十多年了，大家都习以为常，认为严格的调用顺序依赖并没有问题。一旦改变了这个调用顺序，很多应用程序就不能正常工作了，就会出现严重的兼容性问题。

我们每个人都会写很多烂代码，过去写过，未来可能还会再写。这些烂代码，如果运行得很好，没有出现明显的问题，我们就放手吧。

但不是说烂代码我们就永远不管不问。那么，什么时候修改烂代码呢？代码投入使用之前，以及代码出问题的时候，就是我们修改烂代码的时候。

那么代码的修改都有哪些需要注意的地方呢？

代码规范方面的修改，可以大胆些。比如命名规范、代码整理，这些都动不了代码的逻辑，是安全的修改。

代码结构方面的修改，则要谨慎些，不要伤及代码的逻辑。比如把嵌套太多的循环拆分成多个方法，把几百行的代码，拆分成不同的方法，把相似的代码抽象成复用的方法，这些也是相对安全的修改。

代码逻辑方面的修改，要特别小心，除了有明显的问题，我们都尽量避免修改代码的逻辑。即使像上面例子中那样的微小的调用顺序的改变，都可能有意想不到的问题。

## 小结

今天，我们聊了代码复用的一些基本概念。关键的有三点：


1. 要提高代码的复用比例，减少编码的绝对数量；
2. 要复用外部的优质接口，并且推动它们的改进；
3. 烂代码该放手时就放手，以免引起不必要的兼容问题。

## 一起来动手

今天的练手题，我们来分析下 OpenJDK 的一个接口设计问题。

不可更改的集合，是 OpenJDK 的核心类库提供的一个重要功能。这个功能，有助于我们设计实现“一成不变”的接口，降低编码的复杂度。

从 JDK 1.2 开始，这个功能是通过 Collections 类的方法实现的。比如 Collections.unmodifiableList() 方法。

 复制代码

```
1 public static <T> List<T> unmodifiableList(List<? extends T> list)
2
3 Returns an unmodifiable view of the specified list. Query operations on the retur
4
5 The returned list will be serializable if the specified list is serializable. Sim
6
7 Type Parameters:
8     T - the class of the objects in the list
9 Parameters:
10     list - the list for which an unmodifiable view is to be returned.
11 Returns:
12     an unmodifiable view of the specified list.
```

在 JDK 10 里，又添加了新的生成不可更改的集合的方法。比如 List.copyOf() 方法。

 复制代码

```
1 static <E> List<E> copyOf(Collection<? extends E> coll)
```



```
2 Returns an unmodifiable List containing the elements of the given Collection, in
3
4 Implementation Note:
5     If the given Collection is an unmodifiable List, calling copyOf will generally
6
7 Type Parameters:
8     E - the List's element type
9
10 Parameters:
11     coll - a Collection from which elements are drawn, must be non-null
12
13 Returns:
14     a List containing the elements of the given Collection
15
16 Throws:
17     NullPointerException - if coll is null, or if it contains any nulls
18
19 Since:
20     10
```

比较两个接口，你能够理解新接口的改进吗？为什么新加了一个接口，而不是改进原来的接口？为什么使用了一个新的类（List），而不是在原来的类（Collections）里加一个新方法？

欢迎你在留言区讨论上面的问题，我们一起来了解很多接口设计背后的妥协，以及接口演进的办法。也欢迎点击“请朋友读”，把这篇文章分享给你的朋友或者同事，一起交流一下。

© 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。

## 精选留言 (8)



**丁丁历险记**

2019-10-14

感觉老师在复述google 整理术，代码大全，代码简洁之道



👍 6



**王子瑞Aliloke有事电...**

2019-03-08

原来传的是List的子集，JDK 10 里传的是Collection的子集，并且明确要求非空。

扩展了适用范围，增加了非空要求。

基于向下兼容的功能，所以没有在原有方法上修改，而是增加接口。



👍 3





**ifelse**

2022-07-27

不直接改原来方法是开闭原则，没记错List也是继承collection，用collection更通用。



**ifelse**

2022-07-27

我们每个人都会写很多烂代码，过去写过，未来可能还会再写。这些烂代码，如果运行得很好，没有出现明显的问题，我们就放手吧。--记下来



**进化菌**

2021-12-08

说白了，就是要尽可能的做到复用，并且避免继续使用烂代码~



**KEEP**

2019-12-08

我想说，如果烂代码影响了新功能开发时，就是不得不改的时候了，要果断改掉，长痛不如短痛，否则只会越来越烂，后期维护成本会很大。

没有问题的烂代码就不要轻易修改。



**轻歌赋**

2019-03-08

看接口的描述，似乎是形参传入子集的问题，不知道对不对



**往事随风，顺其自然**

2019-03-08

扩展功能更好，不改变原来方法有可能向下转型错误，为了更好兼容以前版本



