目录

26 代码重构的正确姿势

更新时间：2019-12-16 09:45:15

> 天才就是这样，终身努力，便成天才。
>
> ——门捷列夫

1. 前言

在软件迭代过程中常常会因为原来的功能有 BUG、无法满足新的需求、性能遇到瓶颈等原因为需要对代码进行重构。

那么：

- 为什么要重构？
- 如何保证重构代码的正确性？
- 有哪些重构技巧？

这三个关键问题都是本节的重点探讨的内容。

2. 什么是重构？何时重构？

2.1 什么是重构?

想了解为何要重构以及如何重构，就要先搞清楚什么是重构。

> 重构（ refactoring ）是这样的一个过程：在不改变代码外在行为的前提下，对代码进行修改，以改进程序的内部结构。本质上讲，重构就是在代码写好之后，改进它的设计。 –《重构》1

2.2.1 添加新功能的时候

当因为新的需求要为系统添加新功能时，可能会发现很多问题。

比如发现不同的类需要使用同一段代码，而这段代码在之前的一个类中；发现分支条件越来越多，难以维护；发现随着功能的增强，函数的参数列表越来越长，代码长度太长难以理解等。

可以借助开发新功能的时机去对代码进行重构。

2.2.2 修复错误时重构

当我们收到一份来自测试或者技术支持提过来的 “ 编码缺陷 ” 的 jira 时几乎就意味着我们要重构代码了。

可能是接口的结果不符合预期，也可能是接口的性能达不到要求。

2.2.3 代码审查时重构

很多公司都会有代码审查机制，复杂、重要的项目都要通过代码审查（ Code Review ） 后才能上线。

在代码审查阶段，代码审查人员可能对我们代码的可读性、可维护性、代码的性能等进行评价并给出建议。

如果代码审车人员给出了比较合理的建议，此时就要对有问题的代码进行重构。

3. 如何保证重构代码的正确性?

重构技巧千万种，保证正确性是关键。 那么如何保证重构代码的正确性呢？

正如单元测试的章节所讲的一样，**单元测试是保证代码正确性的强有力保证**。

《重构》第二版 2 “ 重构第一步 ” 小节有这样一段描述：

> 进行重构时，我们需要依赖测试。我们将测试视为 bug 检查器，它们能够保护我们不被自己犯的错所困扰。
>
> 把我们想表达的目标写两遍 -- 代码里写一遍，测试里再写一遍 -- 我们的错误才能骗过检测器。这降低了我们犯错的概率。尽管编写测试需要花费时间，但却为我们节省下可观的调试时间。

因此要保证重构的代码都可以通过测试，如果前人并没有编写对应的单元测试，可以在重构时补上对应的单元测试。

4. 一些重构技巧

Java 代码的重构主要包括以下几个方面：代码的 “ 坏味道 ”，对象之间的重构，数据的重构，函数调用的重构和表达式简化的重构。

代码的坏味道有很多种，常见的包括：重复代码，过长的函数，过大的类，过长的参数列表，过多的注释等。

重复代码通常有 3 种情况，

1、**同一个类的多个函数包含重复代码**， 此时可以将公共代码提取为该类的私有函数，在上述函数中调用；

2、**互为兄弟的子类之间包含相同的代码**，此时应该将重复代码上移到父类中；

3、**两个毫不相关的出现重复代码**，此时应该将公共代码抽取到一个新类中。

比如在实际开发中，经常需要根据将某个字段和枚举的值进行比较，可能频繁出现如下代码：

```
Integer someType = xxxDTO.getType();
// 第一种形式
if (CoinEnum.PENNY.getValue() == someType) {
    // 代码省略
}

// 第二种形式
if (CoinEnum.PENNY == CoinEnum.getByValue(someType)) {
    // 代码省略
}
```

那么如何变得更优雅呢？

项目中多处需要执行批量逻辑，可能需要对接口数量做限制，会在项目中多处出现这种代码：

```
public <T> void aRun(List<T> dataList) {

  if (CollectionUtils.isEmpty(dataList)) {
        return;
  }
  int size = 10;

  // 每 10 个元素为一组执行一次
   Lists.partition(dataList, size).forEach((data) -> someRun(dataList));
 }

 private <T> void someRun(List<T> dataList) {
      // 省略
 }
```

此时可以将其封装到工具类中：

```
public static <T> void partitionRun(List<T> dataList, int size, Consumer<List<T>> consumer) {
    if (CollectionUtils.isEmpty(dataList)) {
        return;
    }
    Preconditions.checkArgument(size > 0, "size must > 0");
    Lists.partition(dataList, size).forEach(consumer);
}
```

```
// 每批 10 个
ExecuteUtil.partitionRun(mockDataList, 10, (eachList) -> a.someRun(eachList));

// 每批 20 个
ExecuteUtil.partitionRun(mockDataList, 20, (eachList) -> b.otherRun(eachList));
```

如果**函数过长**，读懂函数的逻辑将变得非常困难，接手代码的人需要花费较多时间才能读懂这些代码。

在工作中，如果接手的代码某一行报错，但是代码行数很多，一般需要读懂整个函数逻辑才敢动手修改，是一件非常痛苦的事情。根据《手册》"【推荐】单个函数总行数不超过 80 行" 3 的建议，需要将大函数拆分成多个子步骤（函数）。最好的办法是搞清楚该函数分为几个步骤，分别将每个子步骤提取为一个子函数即可。

如果**类过大**，通常是函数太多，成员变量过多。如果是函数太多，通常可以根据将函数归类，拆分到不同的类中，一个常见的做法是将 OrderService`` 拆成 OrderSearchService 和 OrderOperateService 分别承担订单的搜索和非搜索业务。如果是成员变量过多，则需要考虑是否应该多个成员变量抽取到某个类中，后者一部分成员变量是否应该属于某个类，通过将新类当做成员变量来消减成员变量的数量。

如果**函数的参数较长**，传参时需仔细核实参数列表以避免误传。如果对外暴露的接口，需要新增一个属性时，为了避免修改签名让二方被迫跟着修改调用的代码，就需要新增一个接口，这种不优雅的方案。根据《手册》的分层领域模型规约部分的建议，**应该将请求的参数封装成查询对象**。这也是一个宝贵的开发经验，尤其是暴露给二方 RPC 接口时，如果未来可能修改参数，尽量使用对象来接收参数，避免因函数签名不同而导致错误。

如果**代码中的注释过多**，应该简化注释，尽量只在关键步骤，特殊逻辑上添加注释，应该使用变量和函数名来表意。

4.2 重新组织函数

当函数中条件表达式较为复杂时，应该将复杂表达式或者其中一部分放到临时变量中，并通过变量名来表达其用途，也可以将部分表达式在一起组成一个含义，还可以将其封装到函数中。

可以参见 spring TypeConverterDelegate#convertToTypedCollection 源码：

```
// 提取为变量
boolean approximable = CollectionFactory.isApproximableMapType(requiredType);
if (!approximable && !canCreateCopy(requiredType)) {
   if (logger.isDebugEnabled()) {
      logger.debug("Custom Map type [" + original.getClass().getName() +
            "] does not allow for creating a copy - injecting original Map as-is");
   }
   return original;
}
// 提取为函数
      private boolean canCreateCopy(Class<?> requiredType) {
            return (!requiredType.isInterface() && !Modifier.isAbstract(requiredType.getModifie
                  Modifier.isPublic(requiredType.getModifiers()) && ClassUtils.hasConstru
      }
```

如下列代码使用一个临时变量表达多种含义：

```
int temp = array.length;
// 省略中间代码
temp = user.getAge();
```

应该修改为：

```
int length = array.length;
// 省略中间代码
int age = user.getAge();
```

如果调用二方批量接口响应很慢容易超时，除了可以像 4.1 重复代码所给出的示例一样，将其改为小批次调用，并将小批次调用的结果进行聚合。

通过封装成工具函数实现复用，可以通过控制 size 来避免接口超时：

```
public static <T, V> List<V> partitionCall2List(List<T> dataList, int size, Function<List<T>, List<

    if (CollectionUtils.isEmpty(dataList)) {
        return new ArrayList<>(0);
    }
    Preconditions.checkArgument(size > 0, "size must > 0");

    return Lists.partition(dataList, size)
            .stream()
            .map(function)
            .filter(Objects::nonNull)
            .reduce(new ArrayList<>(),
                    (resultList1, resultList2) -> {
                        resultList1.addAll(resultList2);
                        return resultList1;
                    });


}
```


为了获取更快的响应速度，可以使用并发或并行特性:

```
public static <T, V> List<V> partitionCall2ListWithCompletable(List<T> dataList,
                                        int size,
                                        ExecutorService executorService,
                                        Function<List<T>, List<V>> function) {

    if (CollectionUtils.isEmpty(dataList)) {
        return new ArrayList<>(0);
    }
    Preconditions.checkArgument(size > 0, "size must >0");

  // 异步调用并获取CompletableFuture对象列表
    List<CompletableFuture<List<V>>> completableFutures = Lists.partition(dataList, size)
            .stream()
            .map(eachList -> {
                if (executorService == null) {
                    return CompletableFuture.supplyAsync(() -> function.apply(eachList));
                } else {
```

```
            })
            .collect(Collectors.toList());

    // 等待全部完成
    CompletableFuture<Void> allFinished = CompletableFuture.allOf(completableFutures.toArr
    try {
        allFinished.get();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }

    // 组合结果
    return completableFutures.stream()
            .map(CompletableFuture::join)
            .filter(CollectionUtils::isNotEmpty)
            .reduce(new ArrayList<V>(), ((list1, list2) -> {
                list1.addAll(list2);
                return list1;
            }));
}
```

还有无数种可以重构的情况，更多重构的场景和范例请参考《重构》这本经典著作，在编码过程中认真体会和运用。

在平时开发时，在满足功能需求的基础上要注重代码的性能。

比如下面这段代码：


```
public List<String> getImages(String type) {

    List<String> result = new ArrayList<>();

    if ("10*20".equals(type)) {

        result.add("http://xxxxxximg1.png");
        result.add("http://xxxxxximgx.png");
        // 省略其他
    } else if ("10*30".equals(type)) {
        result.add("http://yyyyimg1.png");
        result.add("http://yyyyimgy.png");
        // 省略其他
    }
    return result;
}
```

我们可以看到图片的内容是固定的，不需要每次都要查询，上面的写法每个请求都要创建一个 List 将对应的图片塞进去再返回，完全没有必要。

可以参考下面的代码进行重构：

```
private static Map<String, List<String>> images;

static {
    images = new HashMap<>();

    // 根据元素的个数设置初始长度
    List<String> first = new ArrayList<>(16);
```

```
    List<String> second = new ArrayList<>();
    second.add("http://yyyyimg1.png");
    second.add("http://yyyyimg2.png");
    // 省略其他

    images.put("10*20", first);
    images.put("10*30", second);
    images = Collections.unmodifiableMap(images);

}

public List<String> getImages(String type) {
    if (StringUtils.isBlank(type)) {
        return new ArrayList<>();
    }
    return images.getOrDefault(type, new ArrayList<>());
}
```

这样每次请求都会从“缓存”中获取，而且为了防止 map 被修改，将其设置为 unmodifiableMap，而且使用 Map 接口的 getOrDefault 功能，大大简化了代码。

4.3 线程安全问题

当多线程共享变量是，要特别注意线程安全问题。

请看下面的例子：


```
@Service
public class DemoServiceImpl implements DemoService{

    private static List<String> data;

    private List<String> doGetData() {
      // 第一处代码
        if (data == null) {
            data = new ArrayList<>();
            data.add("a");
            data.add("b");
            data.add("c");
        }
        return data;
    }

    @Override
    public List<String> getData(String param) {
      // 省略其他
        List<String> data = doGetData();
      // 省略其他
    }
}
```

假设有多个线程并发调用 doGetData 函数，最初的前两个线程极短时间内依次走到第一处代码的判断处，都会进入 if 代码块。

如果第一个线程在调用 data.addd（“c”）后，如果第二个线程执行 data = new ArrayList<>(); 第一个线程返回 data 时，该集合内没有元素（元素丢失）。

参考修改 1：

```
private static List<String> data;

static {
    data = new ArrayList<>();
    data.add("a");
    data.add("b");
    data.add("c");
    data = Collections.unmodifiableList(data);
}
```

参考修改 2：

```
 private static final Object LOCK = new Object();
// 第 1 处代码
 private static volatile List<String> data;

 private List<String> doGetData() {
     if (data == null) {
         synchronized (LOCK) {
             if (data == null) {
               // 第 2 处代码
                 ArrayList<String> inner = new ArrayList<>();
                 inner.add("a");
                 inner.add("b");
                 inner.add("c");
                 // 第 3 处代码
                 data = Collections.unmodifiableList(inner);
             }
         }
     }
     return data;
 }
```


我们在第 1 处代码加上 volatile 关键字来保证可见性，我们在第 2 处代码创建一个局部变量，避免使用 data = new ArrayList<>();，因为这样会导致还没添加数据就已经创建了对象，另外一个线程并发访问时直接进行最外层判断时就满足 data != null 返回没有元素的 data 集合。

4.4 使用权威的工具类

我们尽量使用 JDK 封装好的类，使用大公司开源的工具类，避免重复劳动。

平时可以多去 commons-lang3 、commons-collections4 、 guava 等知名工具类框架中了解其提供的简单实用的工具类。

比如让当前线程 sleep 一段时间，不要用数字自行计算：

```
Thread.sleep(3 * 1000*60);
```

而应该使用时间相关的类：

```
TimeUnit.MINUTES.sleep(1);
```

如开发中计算耗时，通常获取开始和结束的时间，然后结束时间减去开始时间：

```
    long start = System.currentTimeMillis();
    // 省略一些代码
    long end = System.currentTimeMillis();
    System.out.println(end - start);
}
```

应该使用 StopWatch 类，不仅简单方便，而且该类还提供了更多强大功能：

```
@Test
public void useStopWatch() {
    StopWatch stopWatch = StopWatch.createStarted();
    // 省略一些代码
    System.out.println(stopWatch.getTime());
}
```

非常建议大家在开发中使用第三方工具类时能够主动进入其源码，打开函数列表，去查看里面提供的核心工具类，有时候会有意外发现。

5. 总结

本文主要讲述什么是重构，何时重构，并选取几个典型场景为例说明如何重构。更多重构的场景和范例请参考《重构》这本经典著作。另外推荐《编写可读代码的艺术》、《代码简洁之道》这两本书，它们都是提高代码可读性和重构代码的不错参考资料。

参考资料 9

1. [美] Martin Fowler.《重构：改善既有代码的设计》[M]. [译] 熊节。人民邮电出版社. 2010 ↩

2. [美] Martin Fowler.《重构：改善既有代码的设计（第 2 版）》[M]. [译] 熊节。人民邮电出版社. 2019 ↩

3. 阿里巴巴与 Java 社区开发者.《 Java 开发手册 1.5.0》华山版. 2018 ↩

← 25 阅读源码的正确姿势 27 Code Review的正确姿势 →

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论

目录

千学不如一看，千看不如一练