美团点评技术团队

聊聊clean code

王烨 ·2017-01-19 19:04

clean code，顾名思义就是整洁的代码，或者说清晰、漂亮的代码，相信大多数工程师都希望自己能写出这样的代码。

也许这是个千人千面的话题，每个工程师都有自己的理解。比如我，从一个天天被骂代码写得烂的人，逐渐学习成长，到现在也能写的出“人模人样”的代码来了。这期间算是积累了一点经验心得，想和大家分享，抛砖引玉。

本文主要针对面向对象编程的clean code来阐述，面向过程代码的思路会比较不同，不在本文的讨论范畴。

代码整洁的大前提

代码大部分时候是用来维护的，而不是用来实现功能的

这个原则适用于大部分的工程。我们的代码，一方面是编译好让机器执行，完成功能需求；另一方面，是写给身边的队友和自己看的，需要长期维护，而且大部分项目都不是朝生夕死的短命鬼。

大部分情况下，如果不能写出清晰好看的代码，可能自己一时爽快，后续维护付出的代价和成本将远高于你的想象。

对清晰好看代码的追求精神，比所有的技巧都要重要。

优秀的代码大部分是可以自描述的，好于文档和注释

当你翻看很多开源代码时，会发现注释甚至比我们自己写的项目都少，但是却能看的很舒服。当读完源码时，很多功能设计就都清晰明了了。通过仔细斟酌的方法命名、清晰的流程控制，代码本身就可以拿出来当作文档使用，而且它永远不会过期。

相反，注释不能让写的烂的代码变的更好。如果别人只能依靠注释读懂你的代码的时候，你一定要反思代码出现了什么问题（当然，这里不是说大家不要写注释了）。

说下比较适合写注释的两种场景：

1. public interface，向别人明确发布你功能的语义，输入输出，且不需要关注实现。
2. 功能容易有歧义的点，或者涉及比较深层专业知识的时候。比如，如果你写一个客户端，各种config参数的含义等。

设计模式只是手段，代码清晰才是目的

之前见过一些所谓“高手”的代码都比较抽象，各种工厂、各种继承。想找到一个实现总是要山路十八弯，一个工程里大部分的类是抽象类或者接口，找不到一两句实现的代码，整个读起代码来很不顺畅。我跟他聊起来的时候，他的主要立场是：保留合适的扩展点，克服掉所有的硬编码。

其实在我看来，也许他的代码被“过度设计”了。首先必须要承认的是，在同一个公司工作的同事，水平是参差不齐的。无论你用了如何高大上的设计，如果大多数人都不能理解你的代码或者读起来很费劲的话，其实这是一个失败的设计。

当你的系统内大部分抽象只有一个实现的时候，要好好思考一下，是不是设计有点过度了，清晰永远是第一准则。

代码整洁的常见手段

记住原则后，我们开始进入实践环节，先来看下有哪些促成clean code的常见手段。

code review

很多大公司会用git的pull request机制来做code review。我们重点应该review什么？是代码的格式、业务逻辑还是代码风格？我想说的是，凡是能通过机器检查出来的事情，无需通过人。比如换行、注释、方法长度、代码重复等。除了基本功能需求的逻辑合理没有bug外，我们更应该关注代码的设计与风格。比如，一段功能是不是应该属于一个类、是不是有很多相似的功能可以抽取出来复用、代码太过冗长难懂等等。

我个人非常推崇集体code review，因为很多时候，组里相对高级的工程师能够一眼发现代码存在较大设计缺陷，提出改进意见或者重构方式。我们可以在整个小组内形成一个好的文化传承和风格统一，并且很大程度上培养了大家对clean code的热情。

勤于重构

好的代码，一般都不是一撮而就的。即使一开始设计的代码非常优秀，随着业务的快速迭代，也可能被改的面目全非。

为了避免重构带来的负面影响（delay需求或者带来bug），我们需要做好以下的功课：
① 掌握一些常见的“无痛”重构技巧，这在下文会有具体讲解。
② 小步快跑，不要企图一口吃成个胖子。改一点，测试一点，一方面减少代码merge的痛苦，另一方面减少上线的风险。
③ 建立自动化测试机制，要做到即使代码改坏了，也能保证系统最小核心功能的可用，并且保证自己修改的部分被测试覆盖到。
④ 熟练掌握IDE的自动重构功能。这些会很大程度上减少我们的体力劳动，避免犯错。

静态检查

现在市面上有很多代码静态检查的工具，也是发现bug和风格不好的比较容易的方式。可以与发布系统做集成，强制把主要问题修复掉才可以上线。目前美团点评技术团队内部的研发流程中已经普遍接入了Sonar质量管理平台。

多读开源代码和身边优秀同学的代码

感谢开源社区，为我们提供了这么好的学习机会。无论是JDK的源码，还是经典的Netty、Spring、Jetty，还是一些小工具如Guava等，都是clean code的典范。多多学习，多多反思和总结，必有收益。

代码整洁的常见技巧

前面的内容都属于热身，让大家有个整体宏观的认识。下面终于进入干货环节了，我会分几个角度讲解编写整洁代码的常见技巧和误区。

通用技巧

单一职责

这是整洁代码的最重要也是最基本的原则了。简单来讲，大到一个module、一个package，小到一个class、一个method乃至一个属性，都应该承载一个明确的职责。要定义的东西，如果不能用一句话描述清楚职责，就把它拆掉。

我们平时写代码时，最容易犯的错误是：一个方法干了好几件事或者一个类承载了许多功能。

先来聊聊方法的问题。个人非常主张把方法拆细，这是复用的基础。如果方法干了两件事情，很有可能其中一个功能的其他业务有差别就不好重用了。另外语义也是不明确的。经常看到一个get()方法里面竟然修改了数据，这让使用你方法的人情何以堪？如果不点进去看看实现，可能就让程序陷入bug，让测试陷入麻烦。

程序和人bug，让测试和人测试。

再来聊聊类的问题。我们经常会看到“又臭又长”的service/biz层的代码，里面有几十个方法，干什么的都有：既有增删改查，又有业务逻辑的聚合。每次找到一个方法都费劲。不属于一个领域或者一个层次的功能，就不要放到一起。

我们team在code review中，最常被批评的问题，就是一个方法应该归属于哪个类。

优先定义整体框架

我写代码的时候，比较喜欢先去定义整体的框架，就是写很多空实现，来把整体的业务流程穿起来。良好的方法签名，用入参和出参来控制流程。这样能够避免陷入业务细节无法自拔。在脑海中先定义清楚流程的几个阶段，并为每个阶段找到合适的方法 / 类归属。

这样做的好处是，阅读你代码的人，无论读到什么深度，都可以清晰地了解每一层的职能，如果不care下一层的实现，完全可以跳过不看，并且方法的粒度也会恰到好处。

简而言之，我比较推崇写代码的时候“广度优先”而不是“深度优先”，这和我读代码的方式是一致的。当然，这件事情跟个人的思维习惯有一定的关系，可能对抽象思维能力要求会更高一些。如果开始写代码的时候这些不够清晰，起码要通过不断地重构，使代码达到这样的成色。

清晰的命名

老生常谈的话题，这里不展开讲了，但是必须要mark一下。有的时候，我思考一个方法命名的时间，比写一段代码的时间还长。原因还是那个逻辑：每当你写出一个类似于"temp"、"a"、"b"这样变量的时候，后面每一个维护代码的人，都需要用几倍的精力才能理顺。

并且这也是代码自描述最重要的基础。

避免过长参数

如果一个方法的参数长度超过4个，就需要警惕了。一方面，没有人能够记得清楚这些函数的语义；另一方面，代码的可读性会很差；最后，如果参数非常多，意味着一定有很多参数，在很多场景下，是没有用的，我们只能构造默认值的方式来传递。

解决这个问题的方法很简单，一般情况下我们会构造paramObject。用一个struct或者一个class来承载数据，一般这种对象是value object，不可变对象。这样，能极大程度提高代码的可复用性和可读性。在必要的时候，提供合适的build方法，来简化上层代码的开发成本。

避免过长方法和类

一个类或者方法过长的时候，读者总是很崩溃的。简单地把方法、类和职责拆细，往往会有立竿见影的成效。以类为例，拆分的维度有很多，常见的是横向 / 纵向。例如，如果一个service，处理的是跟一个库表对象相关的所有逻辑，横向拆分就是根据业务，把建立 / 更新 / 修改 / 通知等逻辑拆到不同的类里去；而纵向拆分，指的是
把数据库操作/MQ操作/Cache操作/对象校验等，拆到不同的对象里去，让主流程尽量简单可控，让同一个类，表达尽量同一个维度的东西。

让相同长度的代码段表示相同粒度的逻辑

这里想表达的是，尽量多地去抽取private方法，让代码具有自描述的能力。举个简单的例子

```
public void doSomeThing(Map params1,Map params2){
Do1 do1 = getDo1(params1);
Do2 do2 = new Do2();
do2.setA(params2.get("a"));
do2.setB(params2.get("b"));
do2.setC(params2.get("c"));
mergeDO(do1,do2);
}
private void getDo1(Map params1);
private void mergeDo(do1,do2){...};
```

类似这种代码，在业务代码中随处可见。获取do1是一个方法，merge是一个方法，但获取do2的代码却在主流程里写了。这种代码，流程越长，读起来越累。很多人读代码的逻辑，是“广度优先”的。先读懂主流程，再去看细节。类似这种代码，如果能够把构造do2的代码，提取一个private 方法，就会舒服很多。

面向对象设计技巧

贫血与领域驱动

不得不承认，Spring已经成为企业级Java开发的事实标准。而大部分公司采用的三层/四层贫血模型，已经让我们的编码习惯，变成了面向DAO而不是面向对象。

缺少了必要的模型抽象和设计环节，使得代码冗长，复用程度比较差。每次撸代码的时候，从mapper撸起，好像已经成为不成文的规范。

好处是上手简单，学习成本低。但是每次都不能重用，然后面对两三千行的类看着眼花的时候，我的心是很痛的。关于领域驱动的设计模式，本文不会展开去讲。回归面向对象，还是跟大家share一些比较好的code技巧，能够在一个通用的框架下，尽量好的写出漂亮可重用的code。

个人认为，一个好的系统，一定离不开一套好的模型定义。梳理清楚系统中的核心模型，清楚的定义每个方法的类归属，无论对于代码的可读性、可交流性，还是和产品的沟通，都是有莫大好处的。

为每个方法找到合适的类归属，数据和行为尽量要在一起

如果一个类的所有方法，都是在操作另一个类的对象。这时候就要仔细想一想类的设计是否合理了。理论上讲，面向对象的设计，主张数据和行为在一起。这样，对象之间的结构才是清晰的，也能减少很多不必要的参数传递。

不过这里面有一个要讨论的方法：service对象。如果操作一个对象数据的所有方法都建立在对象内部，可能使对象承载了很多并不属于它本身职能的方法。

例如，我定义一个类，叫做person，。这个类有很多行为，比如：吃饭、睡觉、上厕所、生孩子；也有很多字段，比如：姓名、年龄、性格。

很明显，字段从更大程度上来讲，是定义和描述我这个人的，但很多行为和我的字段并不相关。上厕所的时候是不会关心我是几岁的。如果把所有关于人的行为全部在person内部承载，这个类一定会膨胀的不行。

这时候就体现了service方法的价值，如果一个行为，无法明确属于哪个领域对象，牵强地融入领域对象里，会显得很不自然。这时候，无状态的service可以发挥出它的作用。但一定要把握好这个度，回归本质，我们要把属于每个模型的行为合理的去划定归属。

警惕static

static方法，本质上来讲是面向过程的，无法清晰地反馈对象之间的关系。虽然有一些代码实例（自己实现单例或者Spring托管等）的无状态方法可以用static来表示，但这种抽象是浅层次的。说白了，如果我们所有调用static的地方，都写上import static，那么所有的功能就由类自己在承载了。

让我画一个类图？尴尬了……画不出来。

而单例的膨胀，很大程度上也是贫血模型带来的副作用。如果对象本身有血有肉，就不需要这么多无状态方法。

static真正适用的场景：工具方法，而不是业务方法。

巧用method object

method object是大型重构的常用技巧。当一段逻辑特别复杂的代码，充斥着各种参数传递和是非因果判断的时候，我首先想到的重构手段是提取method object。所谓method object，是一个有数据有行为的对象。依赖的数据会成为这个对象的变量，所有的行为会成为这个对象的内部方法。利用成员变量代替参数传递，会让代码简洁清爽很多。并且，把一段过程式的代码转换成对象代码，为很多面向对象编程才可以使用的继承 / 封装 / 多态等提供了基础。

举个例子，上文引用的代码如果用method object表示大概会变成这样

```
class DoMerger{
   map params1;
   map params2;
   Do1 do1;
   Do2 do2;
   public DoMerger(Map params1,Map params2){
      this.params1 = params1;
      this.params2 = parmas2;
   }
   public void invoke(){
       do1 = getDo1();
       do2 = getDo2();
      mergeDO(do1,do2);
   }
   private Do1 getDo1();
    private Do2 getDo2();
    private void mergeDo(){
       print(do1+do2);
    }
}
```

面向接口编程

面向接口编程是很多年来大家形成的共识和最佳实践。最早的理论是便于实现的替换，但现在更显而易见的好处是避免public方法的膨胀。一个对外publish的接口，一定有明确的职责。要判断每一个public方法是否应该属于同一个interface，是很容易的。

整个代码基于接口去组织，会很自然地变得非常清晰易读。关注实现的人才去看实现，不是嘛？

正确使用继承和组合

这也是个在业界被讨论过很久的问题，也有很多论调。最新的观点是组合的使用一般情况下比继承更为灵活，尤其是单继承的体系里，所以倾向于使用组合
，否则会让子类承载很多不属于自己的职能。

个人对此观点持保留意见，在我经历过的代码中，有一个小规律，我分析一下。

protected abstract 这种是最值得使用继承的，父类保留扩展点，子类扩展，没什么好说的。

protected final 这种方法，子类是只能使用不能修改实现的。一般有两种情况：
① 抽象出主流程不能被修改的，然而一般情况下，public final更适合这个职能。如果只是流程的一部分，需要思考这个流程的类归属，大部分变成public组合到其他类里是更合适的。
② 父类是抽象类无法直接对外提供服务，又不希望子类修改它的行为，这种大多数情况下属于工具方法，比较适合用另一个领域对象来承载并用组合的方式来使用。

protected 这种是有争议的，是父类有默认实现但子类可以扩展的。凡是有扩展可能的，使用继承更理想一些。否则，定义成final并考虑成组合。

综上所述，个人认为继承更多的是为扩展提供便利，为复用而存在的方法最好使用组合的方式。当然，更为大的原则是明确每个方法的领域划分。

代码复用技巧

模板方法

这是我用得最多的设计模式了。每当有两个行为类似但又不完全相同的代码段时，我总是会想到模板方法。提取公共流程和可复用的方法到父类，保留不同的地方作为abstract方法，由不同的子类去实现。

并在合适的时机，pull method up（复用）或者 pull method down（特殊逻辑）。

最后，把不属于流程的、但可复用的方法，判断是不是属于基类的领域职责，再使用继承或者组合的方法，为这些方法找到合适的安家之处。

extract method

很多复用的级别没有这么大，也许只是几行相同的逻辑被copy了好几次，何不尝试提取方法（private）。又能明确方法行为，又能做到代码复用，何乐不为？

责任链

经常看到这样的代码，一连串类似的行为，只是数据或者行为不一样。如一堆校验器，如果成功怎么样、失败怎么样；或者一堆对象构建器，各去构造一部分数据。碰到这种场景，我总是喜欢定义一个通用接口，入参是完整的要校验 / 构造的参数，
出参是成功/失败的标示或者是void。然后有很多实现器分别实现这个接口，再用一个集合把这堆行为串起来。最后，遍历这个集合，串行或者并行的执行每一部分的逻辑。

这样做的好处是：
① 很多通用的代码可以在责任链原子对象的基类里实现；
② 代码清晰，开闭原则，每当有新的行为产生的时候，只需要定义行的实现类并添加到集合里即可；
③ 为并行提供了基础。

为集合显式定义它的行为

集合是个有意思的东西，本质上它是个容器，但由于泛型的存在，它变成了可以承载所有对象的容器。很多非集合的类，我们可以定义清楚他们的边界和行为划分，但是装进集合里，它们却都变成了一个样子。不停地有代码，各种循环集合，做一些相似的操作。

其实很多时候，可以把对集合的操作显示地封装起来，让它变得更有血有肉。

例如一个Map，它可能表示一个配制、一个缓存等等。如果所有的操作都是直接操作Map，那么它的行为就没有任何语义。第一，读起来就必须要深入细节；第二，如果想从获取配置读取缓存的地方加个通用的逻辑，例如打个log什么的，你可以想象是多么的崩溃。

个人提倡的做法是，对于有明确语义的集合的一些操作，尤其是全局的集合或者被经常使用的集合，做一些封装和抽象，如把Map封装成一个Cache类或者一个config类，再提供GetFromCache这样的方法。

总结

本文从clean code的几个大前提出发，然后提出了实践clean code的一些手段，重点放在促成clean code的一些常用编码和重构技巧。
当然，这些只代表笔者本人的一点点感悟。好的代码，最最需要的，还是大家不断追求卓越的精神。欢迎大家一起探索交流这个领域，为clean code提供更多好的思路与方法。

作者简介

王烨，现在是美团点评旅游后台研发组的工程师，之前曾经在百度、去哪儿和优酷工作过，专注Java后台开发。对于网络编程和并发编程具有浓厚的兴趣，曾经做过一些基础组件，也翻过一些源码，属于比较典型的宅男技术控。期待能够与更多知己，在coding的路上并肩前行~
联系邮箱：wangye03@meituan.com