



下载APP



05 | 大类：如何避免写出难以理解的大类？

2021-01-09 郑晔

代码之丑

[进入课程 >](#)**讲述：郑晔**

时长 10:53 大小 9.97M



你好，我是郑晔。

上一讲我们讲了长函数，一个让你感受最直观的坏味道。这一讲，我们再来讲一个你一听名字就知道是怎么回事的坏味道：大类。

一听到大类，估计你的眼前已经浮现出一片无边无际的代码了。类之所以成为了大类，一种表现形式就是我们上节课讲到的长函数，一个类只要有几个长函数，那它就肯定是一眼望不到边了（长函数的话题，我们上一讲已经讨论过了，这里就不再赘述了）。



大类还有一种表现形式，类里面有特别多的字段和函数，也许，每个函数都不大，但架不住数量众多啊，这也足以让这个类在大类中占有一席之地。这一讲，我们就主要来说说这种形式的大类。

分模块的程序

我先来问你一个问题，为什么不把所有的代码都写到一个文件里？

你可能会觉得这个问题很傻，心里想：除了像练习之类的特定场景，谁会在一个正经的项目上把代码写到一个文件里啊？

没错，确实没有人这么做，但你思考过原因吗？把代码都写到一个文件里，问题在哪里呢？

事实是，把代码写到一个文件里，一方面，相同的功能模块没有办法复用；另一方面，也是更关键的，把代码都写到一个文件里，其复杂度会超出一个人能够掌握的认知范围。简言之，**一个人理解的东西是有限的，没有人能同时面对所有细节。**

人类面对复杂事物给出的解决方案是分而治之。所以，我们看到几乎各种程序设计语言都有自己的模块划分方案，从最初的按照文件划分，到后来，使用面向对象方案按照类进行划分，本质上，它们都是一种模块划分的方式。这样，人们面对的就不再是细节，而是模块，模块的数量显然会比细节数量少，人们的理解成本就降低了。

好，你现在已经理解了，对程序进行模块划分，本质上就是在把问题进行分解，而这种做法的背后原因，就是人类的认知能力是有限的。

理解了这一点，我们再回过头来看大类这个坏味道，你就知道问题出在哪了。**如果一个类里面的内容太多，它就会超过一个人的理解范畴，顾此失彼就在所难免了。**


按照这个思路，解决大类的方法也就随之而来了，就是把大类拆成若干个小类。你可能会想，这我也知道，问题是，怎么拆呢？

大类的产生

想要理解怎么拆分一个大类，我们需要知道，这些类是怎么变成这么大的。

职责不单一

最容易产生大类的原因在于**职责的不单一**。我们先来看一段代码：

 复制代码

```
1 public class User {  
2     private long userId;  
3     private String name;  
4     private String nickname;  
5     private String email;  
6     private String phoneNumber;  
7     private AuthorType authorType;  
8     private ReviewStatus authorReviewStatus;  
9     private EditorType editorType;  
10    ...  
11 }
```

这个 User 类拥有着一个大类的典型特征，其中包含着一大堆的字段。面对这样一个类时，我们要问的第一个问题就是，这个类里的字段都是必需的吗？

我们来稍微仔细地看一下这个类，用户 ID (userId)、姓名 (name)、昵称 (nickname) 之类应该是一个用户的基本信息，后面的邮箱 (email)、电话号码 (phoneNumber) 也算是和用户相关联的。今天的很多应用都提供使用邮箱或电话号码登录的方式，所以，这个信息放在这里，也算是可以理解。

再往后看，作者类型 (authorType)，这里表示作者是签约作者还是普通作者，签约作者可以设置作品的付费信息，而普通作者不能。后面的字段是作者审核状态 (authorReviewStatus)，就是说，作者成为签约作者，需要有一个申请审核的过程，这个状态就是审核的状态。

再往后，又出现了一个编辑类型 (editorType)，编辑可以是主编，也可以是小编，他们的权限是不一样的。

这还不是这个 User 类的全部。但是，即便只看这些内容，也足以让我们发现一些问题了。

首先，普通的用户既不是作者，也不是编辑。作者和编辑这些相关的字段，对普通用户来说，都是没有意义的。其次，对于那些成为了作者的用户，编辑的信息意义也不大，因为作者是不能成为编辑的，反之亦然，编辑也不会成为作者，作者信息对成为编辑的用户也是没有意义的。

在这个类的设计里面，总有一些信息对一部分人是没有意义，但这些信息对于另一部分人来说又是必需的。之所以会出现这样的状况，关键点就在于，这里只有“一个”用户类。

普通用户、作者、编辑，这是三种不同角色，来自不同诉求的业务方关心的是不同的内容。只是因为它们都是这个系统的用户，就把它都放到用户类里，造成的结果就是，任何业务方的需求变动，都会让这个类反复修改。这种做法实际上是违反了单一职责原则。

在《软件设计之美》中，我曾经专门用了一讲的篇幅讲 [🔗 单一职责原则](#)，它让我们把模块的变化纳入考量。单一职责原则是衡量软件设计好坏的一把简单而有效的尺子，通常来说，很多类之所以巨大，大部分原因都是违反了单一职责原则。**而想要破解“大类”的谜题，关键就是能够把不同的职责拆分开来。**

回到我们这个类上，其实，我们前面已经分析了，虽然这是一个类，但其实，它把不同角色关心的东西都放在了一起，所以，它变得如此庞大。我们只要把不同的信息拆分开来，问题也就迎刃而解了。下面就是把不同角色拆分出来的结果：

[📄 复制代码](#)

```
1 public class User {
2     private long userId;
3     private String name;
4     private String nickname;
5     private String email;
6     private String phoneNumber;
7     ...
8 }
```

[📄 复制代码](#)

```
1 public class Author {
2     private long userId;
3     private AuthorType authorType;
4     private ReviewStatus authorReviewStatus;
5     ...
6 }
```

[📄 复制代码](#)

```
1 public class Editor {
2     private long userId;
3     private EditorType editorType;
```

```
4    ...  
5    }
```

这里，我们拆分出了 Author 和 Editor 两个类，把与作者和编辑相关的字段分别移到了这两个类里面。在这两个类里面分别有一个 userId 字段，用以识别这个角色是和哪个用户相关。这个大 User 类就这样被分解了。

字段未分组

大类的产生往往还有一个常见的原因，就是**字段未分组**。

有时候，我们会觉得有一些字段确实都是属于某个类，结果就是，这个类还是很大。比如，我们看一下上面拆分的结果，那个新的 User 类：

```
1 public class User {  
2     private long userId;  
3     private String name;  
4     private String nickname;  
5     private String email;  
6     private String phoneNumber;  
7     ...  
8 }
```

[复制代码](#)

前面我们分析过，这些字段应该都算用户信息的一部分。但是，即便相比于原来的 User 类小了许多，这个类依然也不算是一个小类，原因就是，这个类里面的字段并不属于同一种类型的信息。比如，userId、name、nickname 几项，算是用户的基本信息，而 email、phoneNumber 这些则属于用户的联系方式。

从需求上看，基本信息是那种一旦确定就不怎么会改变的内容，而联系方式则会根据实际情况调整，比如，绑定各种社交媒体的账号。所以，如果我们把这些信息都放到一个类里面，这个类的稳定程度就要差一些。所以，我们可以根据这个理解，把 User 类的字段分个组，把不同的信息放到不同的类里面。

```
1 public class User {  
2     private long userId;  
3     private String name;
```

[复制代码](#)


```
4     private String nickname;  
5     private Contact contact;  
6     ...  
7 }
```

[复制代码](#)

```
1 public class Contact {  
2     private String email;  
3     private String phoneNumber;  
4     ...  
5 }
```

这里我们引入了一个 Contact 类（也就是联系方式），把 email 和 phoneNumber 放了进去，后面再有任何关于联系方式的调整就都可以放在这个类里面。经过这次调整，我们把不同的信息重新组合了一下，但每个类都比原来要小。

对比一下，如果说前后两次拆分有什么不同，那就是：前面是根据职责，拆分出了不同的实体，后面是将字段做了分组，用类把不同的信息分别做了封装。

或许你已经发现了，**所谓的将大类拆解成小类，本质上在做的工作是一个设计工作**。我们分解的依据其实是单一职责这个重要的设计原则。没错，很多人写代码写不好，其实是缺乏软件设计的功底，不能有效地把各种模型识别出来。所以，想要写好代码，还是要好好学学软件设计的。

学了这一讲，如果你还想有些极致的追求，我给你推荐《[ThoughtWorks 文集](#)》这本书里“对象健身操”这一篇，这里提到一个要求：**每个类不超过 2 个字段**。

《[ThoughtWorks 文集](#)》是我当年参与翻译的一本书，今天看来，里面的内容大部分都过时了，但“对象健身操”这一篇还是值得一读的。

关于大类的讨论差不多就接近尾声了，但我估计结合这一讲最初的讨论，有些人心中会升起一些疑问：如果我们把大类都拆成小类，类的数量就会增多，那人们理解的成本是不是也会增加呢？

其实，这也是很多人不拆分大类的借口。

在这个问题上，程序设计语言早就已经有了很好的解决方案，所以，我们会看到在各种程序设计语言中，有诸如包、命名空间之类的机制，将各种类组合在一起。在你不需要展开细节时，面对的是一个类的集合。再进一步，还有各种程序库把这些打包出来的东西再进一步打包，让我们只要面对简单的接口，而不必关心各种细节。

如此层层封装，软件不就是这样构建出来的吗？

总结时刻

我们今天讲了大类这个坏味道，这是程序员日常感知最为深刻的坏味道之一。

应对大类的解决方案，主要是将大类拆分成小类。我们需要认识到，模块拆分，本质上是帮助人们降低理解成本的一种方式。

我们还介绍了两种产生大类的原因：

职责不单一；

字段未分组。

无论是哪种原因，想要有效地对类进行拆分，我们需要对不同内容的变动原因进行分析，而支撑我们来做这种分析的就是单一职责原则。将大类拆分成小类，本质上在做的是设计工作，所以，想要写好代码，程序员需要学好软件设计。

有人觉得拆分出来的小类过多，不易管理，但其实程序设计语言早就为我们提供了各种构造类集合的方式，比如包、命名空间等，再进一步，还可以封装出各种程序库。

如果今天的内容你只能记住一件事，那请记住：**把类写小，越小越好。**

划重点 05

坏味道：大类

产生大类的原因

1. 职责不单一。
2. 字段未分组。

软件设计的原则

单一职责原则。

极致的追求

每个类不超过 2 个字段。

记住一句话

把类写小，越小越好。



思考题

你在实际工作中遇到过多大的类，你分析过它是怎样产生的吗？又是如何拆分的呢？欢迎在留言区分享你的经历。如果你身边有同事总是写出大类，你不妨把这节课分享给他，帮他解决大类的烦恼。

感谢阅读，我们下一讲再见！

参考资料：

🔗 单一职责原则：你的模块到底为谁负责？

🔗 你的代码是怎么变混乱的？

提建议

© 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。

上一篇 04 | 长函数：为什么你总是不可避免地写出长函数？

下一篇 06 | 长参数列表：如何处理不同类型的长参数？

精选留言 (14)

🗨 写留言



qinsi

2021-01-09

正如类太大会超出人的理解范畴，类太多也会。举个java的例子，做业务开发时大部分需求都可以通过spring boot完成，但如果要对spring进行定制，需要理解的内容就多了一个数量级。框架本身在满足更多需求的同时不断重构，类的职责越来越单一，数量也越来越多。虽然每个改动都有正当的理由，但如果不知道这些改动的历史，一下子就被那么多类糊一脸，任谁也是吃不消的。个人理解这可能是在业务优先的场景下，两害相权取其轻...

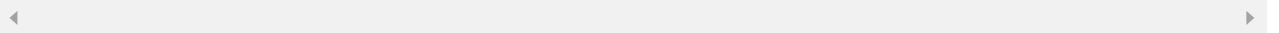
展开 ∨

作者回复: 认为类太多是个问题，我在文章中已经说了一个角度，这里再补充一个角度。这里有一个假设是，一上来就要把所有的类理解掉，

然而，这种假设是不成立的，作为 Java 程序员，你不会去看所有 JDK 里的类，也不会看 Spring 所有的类。

一般的做法是理解主线，然后，根据需要了解相应的类，这是做事方法的问题。不能因为我们可能要面对所有代码，就一下子去吃透所有的代码，这是普通人做不到的。

所以，类的数量多少不是问题，通过怎样的方式，降低代码理解的难度才是我们要考虑的问题。



💬 1

👍 9



wang_acmilan

2021-01-09

目前在做嵌入式的项目，里面的C代码如郑老师所言，大部分都是以“效率”为名，写的巨长无比。

我有一个问题想咨询郑老师，我们的项目不算大，4万行左右，我在进行一代往二代的重构，我个人感觉应该把功能都拆开，以不同文件夹的层级方式进行代码的整理；而组内的同事以及外包的员工觉得代码不算多，一个文件夹，几个文件一把就搞定了，没必要搞...
展开 ▼

作者回复: 马斯洛需求层次理论告诉我们，人有不同层次的追求。生存是底线，总有人会告诉你，温饱解决就行。

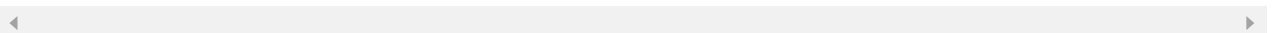
写代码也一样，功能实现就可以了，代码规模不大，可以理解。问题在于，等规模大了，你真的改得动吗？

你的代码为什么要改动？还不是一代的代码改起来很吃力了。为什么吃力了？是因为功能实现不了吗？还不是代码可维护性差。为什么可维护性差？还不是根本不知道可维护性好的代码是什么样。

你不妨推演一下，按照他们的建议写代码，距离走回老路上，还有多远的距离。

既然有机会建立新的标准，既然有机会知道可维护的代码长什么样，为什么要按照老路走呢？

如果知道什么是“积重难返”，就会懂得“勿以善小而不为”的价值所在了。

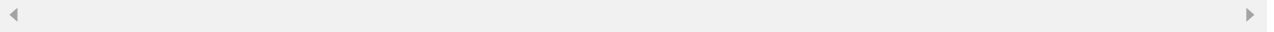


**邓志国**

2021-01-11

看了郑大的例子，正合我当前做法，给自己吃颗定心丸。我现在java一个4万行代码的项目，最大的类不到300行代码。普遍100行以下。很多是得益于对象健身操。操练了对象健身操，对面向对象会有更深刻理解

作者回复: 你做得已经很好了!

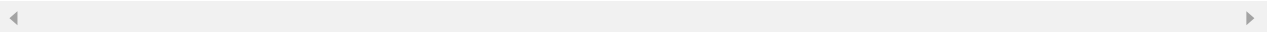
**Sinvi**

2021-01-09

今天有了新的追求

展开 ∨

作者回复: 加油!

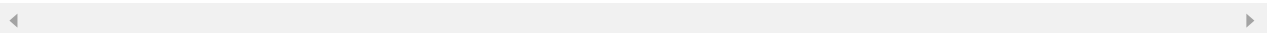
**246小言**

2021-01-09

每个类两个字段? ? ? ?

展开 ∨

作者回复: 我知道大多数人做不到，但我们应该知道更高的追求是什么。

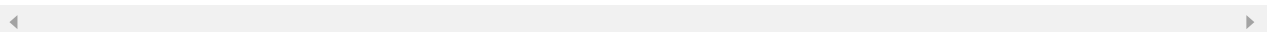
**Hobo**

2021-01-11

将user里的联系信息单独拆出一个类来以后，我要对外暴露一个修改联系信息的方法是应该放user里还是放contact里?

展开 ∨

作者回复: 如果 user 类是整个的根，修改联系方式就可以在 user 类里提供一个入口，然后，再调用 contact 类。



**lifeYY**

2021-01-11

有的时候，拆分和聚合是需要做取舍的。比如，微服务的聚合层，是为了适配和性能考虑会做裁剪或聚合，不可避免的会出现些大类，此时可以进行字段分组，但是聚合后的大类是无法进行拆分。

作者回复：我的理解，你问的问题是，在微服务中，有专门把多个服务的结果聚合起来，返回给客户端，这是你所说的聚合层。你的问题是，在这个聚合层里，表示请求和应答的类可能会比较大，改怎么办。

根据这个理解，我的回答是，这里的聚合层扮演的角色其实一个防腐层，它本身的职责就是和请求应答去做一一应对。一般来说，这样类行为很单一，主要的职责就是数据转换。对于这种类，大一点是可以的，因为它不会对业务造成什么影响。重点在于，这个类里没有业务。

如果你想把类里面的字段做一个分组，可以研究一下不同的转换库，比如，在 Java 中把 JSON 和对象进行相互转换的 Jackson。看看它们怎么可以把不同类的字段和协议中的平铺字段直接映射，肯定有方案的。

不过，最后要提醒你的一个重点是，简单的聚合并不是一个好办法，而是需要谨慎地设计通信协议，这才是保证类比较小的根本办法。

**熊斌**

2021-01-09

我们的项目中，或者是用到的类似于JPUSH这种开源软件中，都存在“大类”的问题。最头疼的是遇见没有注释说明的大类，里面罗列了很多字段、函数，但是你都没法一眼就看出来各自都是干嘛的，只能先猜一下，完了之后根据调用关系去梳理。

最近用PHP进行二次开发，用到一个扩展包，里面的入口class写得巨长无比.....

展开 ∨

作者回复：不意外，大多数代码都是有问题的，Clean Code是稀缺的。



2021-01-13

项目中好像都是大类，每个人加点东西，就更庞大了。

展开 ∨

作者回复：同情你。



大碗

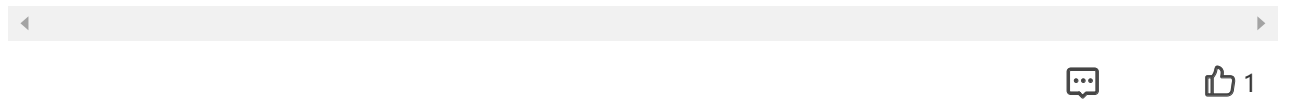
2021-01-12

MVC里常出现的大类是UserService,提供登陆，注册，修改密码，查询用户等方法里面会有UserDao,LoginRecordDao, CacheDao等很多个注入的“字段”。这种应该怎么拆分呢，如果按职责拆分成UserLoginService,UserRegisterService,UserPasswordService,UserInfoService，保持每个service里面又少量注入“Service、Dao”，这么拆对不对呢

展开 ∨

作者回复：我现在通常的做法是，引入应用服务层，在应用服务层去协调各个领域服务层。而在领域服务层，一个服务通常只对应一个数据库访问层的代码。

你这里的纠结其实就是缺少了应用服务层。如果不引入应用服务层，可以考虑的做法是，服务层内部可以引用，但服务只对应一个数据库访问层代码。



Jxin

2021-01-12

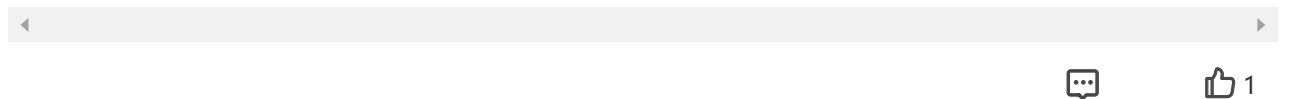
1.总结：小类大对象+面相意图的抽象。

2.好的抽象只需关心使用不用关心实现，所以能降低复杂性。但不好的抽象。。。

3.个人觉得还是讲究平衡，没必要一上来就拆分得很细致。当复杂度让你皱眉头时再拆...

展开 ∨

作者回复：看一下我在开篇词中提供的自查表，那里面的括号就是你要根据自己团队的实际情况填写的内容。



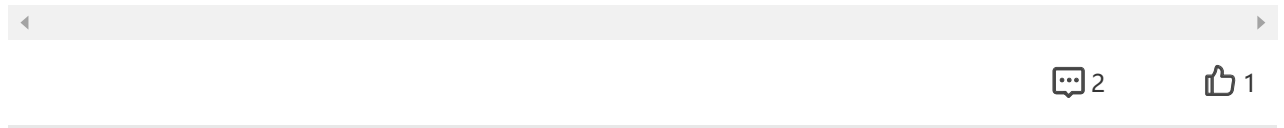
Hobo

2021-01-11

我们项目实体类和数据库表是直接映射的，如果拆分成上面这样的话，数据库是也需要拆多表？(用户、作者、编辑)

展开 ▾

作者回复: 这是设计怎么与实现映射的事，通常来说，应该把实现拆分成多个表。



明

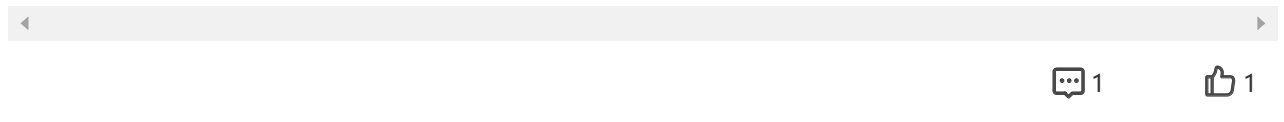
2021-01-11

老师，我也有个一直找不到平衡点的问题，就是类爆炸问题，如果拆分的类太多 会不会出现累爆炸的问题呢，从而影响系统性能，这个有没有个比较数字化的标准呢。还是我看这个问题的角度就完全不对。（ps 拍个马屁：我真的越来越崇拜老师了😁😁）

展开 ▾

作者回复: 不要把性能放在这里说事，因为这属于发力找错了对象。性能优化肯定不是在这个层面上的，而是要从一个系统的层面上进行考量。

类爆炸，首先是，啥叫类爆炸？类多就是类爆炸吗？那岂不是所有代码都在一个类里完成就是最好的。类多不是问题，问题是过度设计造成的难以理解的结构才是问题。



刘大明

2021-01-09

现在项目中很多大类，把很多东西都返回前端。美其名曰这样前端想怎么用怎么用。还有就是拆分成多个小类之后，很多人没有管理好小类的能力。分散的越来越复杂。

作者回复: 唉，用加班代替思考。

