

第 3 章

一种编程理论

就算是再巨细靡遗的模式列表，也不可能涵盖编程中所遇到的每一种情况。你免不了（甚至常常）会遭遇到这种情景：上穷碧落，也找不到对应的现成解决方案。于是便需要有针对性问题的通用解决方案。这也正是学习编程理论的原因之一。原因之二则是那种知晓如何去做、为何去做之后所带来的胸有成竹。当然，如果把编程的理论和实践结合起来讨论，内容就会更加精彩了。

每个模式都承载着一点点理论。但实际编程中存在一些更加深广的影响力，远不是孤立的模式所能概括的。本章将会讲述这些贯穿于编程中的横切概念，它们被分为两类：价值观与原则。价值观是编程过程的统一支配性主题。珍视与其他人沟通的重要性，把代码中多余的复杂性去掉，并保持开放的心态，这才是我工作状态最佳的表现。这些价值观——沟通、简单和灵活——影响了我在编程时所做的每个决策。

此处描述的原则不像上面的价值观那样意义深远，不过每一项原则都在许多模式中得以体现。价值观有普遍的意义，但往往难以直接应用；模式虽可以直接应用，却是针对于特定情景；原则在价值观和模式之间搭建了桥梁。我早已发现，在那种没有模式可以应用，或是两个相互排斥的模式可以同等应用的场合，如果把编程原则弄清楚，对解决疑难会是一件好事。在面对不确定性的时候，对原则的理解让我可以“无中生有”创造出一些东西，同时能和其他的实践保持一致，而且结果一般都不错。

价值观、原则和模式，这 3 种元素互为补充，组成了一种稳定的开发方式。模式描述了要做什么，价值观提供了动机，原则把动机转化成了实际行动。

这里的价值观、原则和模式，是通过我的亲身实践、反思以及与其他人的讨论总结出来的。我们都曾经从前人那里吸收经验，最终会形成一种开发方式，但不是唯一的开发方式。不同的价值观和不同的原则会产生不同的方式。把编程方式用价值观、原则和模式的形式展现出来，其优点之一就是可以更加有效地展现编程方法的差异。如果你喜欢用某种方式来做事，而我喜欢另一种，那么就可以识别出我们在哪种层次上存在分歧，从而避免浪费时间。如果我们各自认可不同的原则，那么争论哪里该用大括号根本无助于解决问题。

3.1 价值观

有 3 个价值观与卓越的编程血脉相连，它们分别是：沟通、简单和灵活。虽然它们有时候会有所冲突，但更多的时候则是相得益彰。最优秀的程序会为未来的扩展留下充分的选择余地，不包含不相关的元素，容易阅读，容易理解。

3.1.1 沟通

如果阅读者可以理解某段代码，并且进一步修改或使用它，那么这段代码的沟通效果就很好。在编程时，我们很容易从计算机的角度进行思考。但只有一面编程一面考虑其他人的感受，我才能编写出好的代码。在这种前提下编写出的代码更加干净易读，更有效率，更清晰地展现出我的想法，给了我全新的视角，减轻了我的压力。我的一些社会性需要得到了自我满足。最开始编程吸引我的部分原因在于我可以通过编程与外界交流，然而，我不想与那些难缠又无法理喻的烦人家伙打交道。过了 20 年，把别人当作空气一样

的编程方式才在我眼中褪尽了颜色。耗尽心神去精心搭建一座糖果城堡，于我而言已毫无意义。

Knuth 所提出的文学编程理论促使我把注意力放到沟通上来：程序应该读起来像一本书一样。它需要有情节和韵律，句子间应该有优雅的小小跌宕起伏。

我和 Ward Cunningham 第一次接触到文学性程序这个概念以后，我们决定来试一试。我们找出 Smalltalk 中最干净的代码之一——ScrollController，坐到一起，然后试着把它写成一个故事。几个小时以后，我们以自己的方式完全重写了这段代码，把它变成了一篇合情合理文章。每次遇到难以解释清楚的逻辑，重新把它写一遍都要比解释这段代码为何难以理解容易得多。沟通的需要改变了我们对于编码的看法。

在编程时注重沟通还有一个很明显的经济学基础。软件的绝大部分成本都是在第一次部署以后才产生的。从我自己修改代码的经验出发，我花在阅读既有代码上的时间要比编写全新的代码长得多。如果我想减少代码所带来的开销，我就应该让它容易读懂。

注重沟通还可以帮助我们改进思想，让它更加现实。一方面是由于投入更多的思考，考虑“如果别人看到这段代码会怎么想”所需要调动的脑细胞，和只关注自己是不一样的。这时我会退后一步，从全新的视角来审视面对的问题和解决方案。另一方面则是由于压力的减轻，因为我知道自己所做的事情是在务正业，我做的是对的。最后，作为社会性的产物，明确地考虑社会因素要比在假设它们不存在的情况下工作更为现实。

3.1.2 简单

在 *Visual Display of Quantitative Information* 一书中，Edward Tufte 做过一个实验，他拿过一张图，把上面没有增加任何信息的标记全都擦掉，最终得到了一张很新颖的图，比原先那张更容易理解。

去掉多余的复杂性可以让那些阅读、使用和修改代码的人更容易理解。

有些复杂性是内在的，它们准确地反映出所要解决的问题的复杂性。但有些复杂性的产生完全是因为我们忙着让程序运行起来，在摆弄过程中留下来的“指甲印”没擦干净。这种多余的复杂性降低了软件的价值，因为一方面软件正确运行的可能性降低了，另一方面将来也很难进行正确的改动。回顾自己做过的事情，把麦子和糠分开，是编程中不可或缺的一部分。

简单存在于旁观者的眼中。一个可以将专业工具使用得得心应手的高级程序员，他所认为的简单事情对一个初学者来说可能会比登天还难。只有把读者放在心里，你才可以写出动人的散文。同样，只有把读者放在心里，你才可以编写出优美的程序。给读者一点挑战没有关系，但过多的复杂性会让你失去他们。

在复杂与简单的波动中，计算机技术不断向前推进。直到微型计算机出现之前，大型机架构的发展倾向仍然是越来越复杂。微型计算机并没有解决大型机的所有问题，只不过在很多应用中，那些问题已经变得不再重要。编程语言也在复杂和简单的起伏中前行。C++在C的基础上产生，而后在C++的基础上又出现了Java，现在Java本身也变得越来越复杂了。

追求简单推动了进化。JUnit比它所大规模取代的上一代测试工具简单得多。JUnit催生了各种模仿者、扩展软件和新的编程/测试技术。它最近一个版本JUnit 4已经失去了那种“一目了然”的效果，虽然每一个导致其复杂化的决定都有我参与其中，但亦未能阻止这种趋势。总有一天，会有人发明一种比JUnit简单许多的方式，以方便编程人员编写测试。这种新的想法又会推动另一轮进化。

在各个层次上都应当要求简单。对代码进行调整，删除所有不提供信息的代码。设计中不出现无关元素。对需求提出质疑，找出最本质的概念。去掉多余的复杂性后，就好像有一束光照亮了余下的代码，你就有机会用全新的视角来处理它们。

沟通和简单通常都是不可分割的。多余的复杂性越少，系统就越容易理解；在沟通方面投入越多，就越容易发现应该被抛弃的复杂性。不过有时

候我也会发现某种简化会使程序难以理解，这种情况下我会优先考虑沟通。这样的情形很少，但常常都表示这里应该有一些我尚未察觉的更大规模的简化。

3.1.3 灵活

在三种价值观中，灵活是衡量那些低效编码与设计实践的一把标尺。以获取一个常量为例，我曾经见到有人会用环境变量保存一个目录名，而那个目录下放着一个文件，文件中写着那个常量的值。为什么弄这么复杂？为了灵活。程序是应该灵活，但只有在发生变化的时候才需如此。如果这个常量永远不会变化，那么付出的代价就都白费了。

因为程序的绝大部分开销都是在它第一次部署以后才产生，所以程序必须要容易改动。想象中明天或许会用得上的灵活性，可能与真正修改代码时所需要的灵活性不是一回事。这就是简单性和大规模测试所带来的灵活性比专门设计出来的灵活性更为有效的原因。

要选择那些提倡灵活性并能够带来及时收益的模式。对于会立刻增加成本但收效却缓慢的模式，最好让自己多一点耐心，先把它们放回口袋里，需要的时候再拿出来。这样就可以用最恰当的方式使用它们。

灵活性的提高可能以复杂性的提高为代价。比如说，给用户提供一个可自定义配置的选择提高了灵活性，但是因为多了一个配置文件，编程时也需要考虑这一点，所以也就更复杂了。反过来简单也可以促进灵活。在前面的例子中，如果可以找到取消配置选项但又不丧失价值的方式，那么这个程序以后就更容易改动。

增进软件的沟通效果同样会提高灵活性。能够快速阅读、理解和修改你的代码的人越多，它将来发生变化的选择就越多。

本书中介绍的模式会通过帮助编程人员创建简单、可以理解、可以修改的应用程序来提高程序的灵活性。

3.2 原则

实现模式并不是无缘无故产生的。每一种模式都或多或少体现了沟通、简单和灵活这些价值观。原则是另一个层次上的通用思想，比价值观更贴近于编程实际，同时又是模式的基础。

我们有很多理由来检查一下这些原则。正如元素周期表帮助人们发现了新的元素，清晰的原则也可以引出新的模式。原则可以解释模式背后的动机，它是有普遍意义的。在对立模式间进行选择时，最好的方式就是用原则来说话，而不是让模式争来争去。最后，如果遇到从未碰到过的情况，对原则的理解可以充当我们的向导。

例如，假如要使用新的编程语言，我可以根据自己对原则的理解发展出有效的编程方式，不必盲目模仿现有的编程方式，更不用拘泥于在其他语言中形成的习惯（虽然可以用任何语言编写 FORTRAN 风格的代码，但不该那么做）。对原则的充分理解使我能够快速学习，即使在新鲜局面下仍然能够一以贯之地符合原则。接下来的部分，我将为你讲述隐藏在模式背后的原则。

3.2.1 局部化影响

组织代码结构时，要保证变化只会产生局部化影响。如果这里的一个变化会引出那里的问题，那么变化的代价就会急剧上升了。把影响范围缩到最小，代码就会有极佳的沟通效果。它可以被逐步深入理解，不必一开始就要鸟瞰全景。因为实现模式背后一条最主要的动机就是减少变化所引起的代价，所以局部化影响这条原则也是很多模式的形成缘由之一。

3.2.2 最小化重复

最小化重复这条原则有助于保证局部化影响。如果相同的代码出现在很多地方，那么改动其中一处副本时，就不得不考虑是否需要修改其他副本；

变动不再只发生在局部。代码的复制越多，变化的代价就越大。

复制代码只是重复的一种形式。并行的类层次结构也是其一，同样破坏了局部化影响原则。如果修改一处概念需要修改两个或更多的类层次结构，就表示变化的影响已经扩散了。此时应重新组织代码，让变化只对局部产生影响。这种做法可以有效改进代码质量。

重复不容易被预见到，有时在出现以后一段时间才会被觉察。重复不是罪过，它只是增加了变化的开销。

我们可以把程序拆分成许多更小的部分——小段语句、小段方法、小型对象和小型包，从而消除重复。大段逻辑很容易与其他大段逻辑出现重复的代码片断，于是就有了模式诞生的可能，虽然不同的代码段落中存在差异，但也有很多相似之处。如果能够清晰地表述出哪些部分程序是等同的，哪些部分相似性很少，而哪些部分则截然不同，程序就会更容易阅读，修改的代价也会更小。

3.2.3 将逻辑与数据捆绑

局部化影响的必然结果就是将逻辑与数据捆绑。把逻辑与逻辑所处理的数据放在一起，如果有可能尽量放到一个方法中，或者退一步，放到一个对象里面，最起码也要放到一个包下面。在发生变化时，逻辑和数据很可能会同时被改动。如果它们被放在一起，那么修改它们所造成的影响就会只停留在局部。

在编码开始的那一刻，我们往往不太清楚该把逻辑和数据放到哪里。我可能在A中编写代码的时候才意识到需要B中的数据。在代码正常工作之后，我才意识到它与数据离得太远。这时候我需要做出选择：是该把代码挪到数据那边去，还是把代码挪到逻辑这边来，或者把代码和数据都放到一个辅助对象中？也许还可能意识到，这时我还没法找出如何组合它们以便增进沟通的最好方式。

3.2.4 对称性

对称性也是我随时随地运用的一项原则。程序中处处充满了对称性。比如 `add()` 方法总会伴随着 `remove()` 方法，一组方法会接受同样的参数，一个对象中所有的字段都具有相同的生命周期。识别出对称性，把它清晰地表述出来，代码将更容易阅读。一旦阅读者理解了对称性所涵盖的某一半，他们就会很快地理解另外一半。

对称性往往用空间词汇进行表述：左右对称的、旋转的，等等。程序中的对称性指的是概念上的对称，而不是图形上的对称。代码中对称性的表现，是无论在什么地方，同样的概念都以同样的形式呈现。

这是一个缺少对称性的例子：

```
void process() {
    input();
    count++;
    output();
}
```

第二条语句比其他的语句更加具体。我会根据对称性的原则重写它，结果是：

```
void process() {
    input();
    incrementCount();
    output();
}
```

这个方法依然违反了对称性。这里的 `input()` 和 `output()` 操作都是通过方法意图来命名的，但是 `incrementCount()` 这个方法却以实现方式来命名。在追求对称性的时候，我会考虑为什么我会增加这个数值，于是就有了下面的结果：

```
void process() {
    input();
    tally();
    output();
}
```

在准备消灭重复之前，常常需要寻找并表示出代码中的对称性。如果在很多代码中都存在类似的想法，那么可以先把它们用对称的方式表示出来，让接下来的重构有一个良好开端。

3.2.5 声明式表达

实现模式背后的另一条原则是尽可能声明式地表达出意图。命令式的编程语言功能强大灵活，但是在阅读时需要跟随着代码的执行流程。我必须在脑海中建起一个程序状态、控制流和数据流的模型。对于那些只是陈述简单事实，不需要一系列条件语句的程序片断，如果用简单的声明方式写出来，读着就容易多了。

比如在 JUnit 的早期版本中，测试类里可能会有一个静态的 `suite()` 方法，该方法会返回需要运行的测试集合。

```
public static junit.framework.Test suite() {
    Test result= new TestSuite();
    ...complicated stuff...
    return result;
}
```

现在就有了一个很简单很常见的问题：哪些测试会被执行？在大多数情况下，`suite()` 方法只是将多个类中的测试汇总起来。但是因为它是一个通用方法，所以我必须要读过、理解该方法以后，才能够百分之百确定它的功能。

JUnit 4 用了声明式表达原则来解决这个问题。它不是用一个方法来返回测试集，而是用了一个特殊的 `test runner` 来执行多个类中的所有测试（这是最常见的情况）：

```
@RunWith(Suite.class)
@TestClasses({
    SimpleTest.class,
    ComplicatedTest.class
})
class AllTests {
}
```

如果测试是用这种方式汇总的，那么我只需要读一下 `TestClasses` 注解就可以知道哪些测试会被执行。面对这种声明式的表达方式，我不需要臆测会出现什么奇怪的例外情况。这个解决方案放弃了原始的 `suite()` 方法所具备的能力和通用性，但是它声明式的风格使得代码更加容易阅读。（在运行测试方面，`RunWith` 注解比 `suite()` 方法更为灵活，但这应该是另外一本书里的故事了。）

3.2.6 变化率

最后一个原则就是把具有相同变化率的逻辑、数据放在一起，把具有不同变化率的逻辑、数据分离。变化率具有时间上的对称性。有时候可以将变化率原则应用于人为的变化。例如，如果开发一套税务软件，我会把计算通用税金的代码和计算某年特定税金的代码分离开。两类代码的变化率是不同的。在下一年中做调整的时候，我会希望能够确保上一年中的代码依然奏效。分离两类代码可以让我更确信每年的修改只会产生局部化影响。

变化率原则也适用于数据。一个对象中所有成员变量的变化率应该差不多是相同的。只会在一个方法的生命周期内修改的成员变量应该是局部变量。两个同时变化但又和其他成员的变化步调不一致的变量可能应该属于某个辅助对象。比如金融票据的数值与币种会同时变化，那么这两个字段最好放到一个辅助对象 `Money` 中：

```
setAmount(int value, String currency) {
    this.value= value;
```

```

        this.currency= currency;
    }

```

上面这段代码就变成了：

```

    setAmount(int value, String currency) {
        this.value= new Money(value, currency);
    }

```

然后进一步调整：

```

    setAmount(Money value) {
        this.value= value;
    }

```

变化率原则也是对称性的一个应用，不过是时间上的对称。在上面的例子中，`value` 和 `currency` 这两个初始字段是对称的，它们会同时变化。但它们与对象中其他的字段是不对称的。把它们放到自己应该从属的对象中，让新的对象向阅读者传达出它们的对称关系，这样就更有可能在将来消除重复，进一步达到影响的局部化。

3.3 小结

本章介绍了实现模式的理论基础。沟通、简单和灵活这三条价值观为模式提供了广泛的动机。局部化影响、最小化重复、将逻辑与数据捆绑、对称性、声明式表达和变化率这 6 条原则帮助我们将价值观转化为实际行动。接下来我们将会进入模式的世界，看一看针对编程实战中频繁出现的问题，会有哪些特定的解决方案。

注重通过代码与人沟通是一件有价值的事情，我们将在下一章“动机”中探寻其背后的经济因素。