TURING 图灵原创

代码里的世界观

通往架构师之路

余叶◎著

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵原创

代码里的世界观

通往架构师之路

余叶◎著

人民邮电出版社
北 京

图书在版编目（CIP）数据

代码里的世界观 : 通往架构师之路 / 余叶著. -- 北京 : 人民邮电出版社, 2018.11（2019.2重印）
（图灵原创）
ISBN 978-7-115-49523-5

Ⅰ. ①代… Ⅱ. ①余… Ⅲ. ①程序设计 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2018)第224651号

内 容 提 要

本书分为两大部分，第一部分讲述程序员在编写程序和组织代码时遇到的很多通用概念和共同问题，比如程序里的基本元素，如何面向对象，如何面向抽象编程，什么是耦合，如何进行单元测试等。第二部分讲述程序员在编写代码时的思考和选择，比如程序员的两种工作模式，如何坚持技术成长，程序员的组织生产方法，程序员的职业生涯规划等。

本书适合工作2~5年，有一定基础的程序员阅读。

◆ 著　　余　叶
责任编辑　王军花
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
涿州市京南印刷厂印刷

◆ 开本：800×1000　1/16
印张：15
字数：354千字　　2018年11月第1版
印数：3001-4000册　　2019年2月河北第2次印刷

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字20170147号

站在巨人的肩上

Standing on Shoulders of Giants

iTuring.cn

站在巨人的肩上

Standing on Shoulders of Giants

iTuring.cn

前　言

安安静静地写代码不知不觉已经十几年，这期间偶尔会有同龄的程序员朋友对我倾诉：他变得不喜欢编程啦，又或者因升为领导而沾沾自喜——总算不用编程了。我听了挺难过的，一个你不喜欢的高强度脑力工作，还吭哧吭哧干了十多年，这该多么痛苦啊。我遇到更普遍的老程序员的态度是：不喜欢也不讨厌，就一份工作而已。

我发现自己对于编程的兴趣曲线似乎与众不同，是从一开始的既不喜欢也不讨厌，到现在的越来越喜欢，算是后知后觉型。为什么会有这样的变化？除了天赋有限，我觉得和自己内心深处的兴趣点密切相关。

我也在不断追逐技术潮流，但更感兴趣的始终不是新框架带来了哪些新概念，而是背后那些最朴实、最基本的代码结构的本质，是那些最通用的编程技巧。跟随并学习潮流技术只是我的手段，能不断补充属于我的编程技巧才是最终目的。渐渐地，对软件编程的很多困惑，在我坚持了十多年后纷纷得以解答。这正是我的兴趣曲线不降反升的原因。

所以呢，我的程序员生涯并不完美，始终慢别人一拍地去学习。也有沮丧也有迷茫，还好我一直清楚自己最想要的是什么：在能满足温饱的同时，我要解答自己最想知道的那部分疑惑。坚持到现在，也足以让我写本书与大众分享了。

我总认为代码所描述的世界，也有它的哲学内在，这个世界有自己的运行规律，有属于自己的世界观。每次我理解完一个新东西，总是试图往规律或内在方面去挂钩。哪怕是一些简单技术，也要琢磨一下它的本质到底是什么。

比如，我认为编程中组织代码的能力，说白了就是将各种 API 串起来的能力，无论是针对新鲜的、高大上的 API，还是已经淘汰的 API。最近层出不穷的新技术，比如搜索、云计算、区块链等，要使用它们，不可能自己去从头开发。那些基础设施，牛人们一般已经给你开发好了。作为一名普通的程序员，你面对的终究还是一堆一堆的 API，然后以业务需求为基础，去组织这些 API。**如何能让凌乱的代码组织得更好，正是本书要讨论的重点内容。**

因此，本书所讲的基本都是编程的通用知识点，它们是 10 年甚至 20 年都不会淘汰的编程技术，也是每个人都会遇到且思考过的问题。而决意写这种题材的编程书确实需要勇气。

网上当然有很多相关资料，但太分散。其中很多都是初学者自己学习时的体验心得，但初学者总结的规律往往缺乏厚度。可当你有足够多的实战经验，可以总结更有价值的内容时，却在忙

着学习其他新知识，并不会特意花时间分享你早已掌握的知识。这就是现实情况：很多真正有价值的东西，仅在高手的脑子里，而得不到传承。

另一方面，这些 10 年甚至 20 年都不会被淘汰的编程知识，市面上也极少有将它们综合起来并讲得有意思的书。我按照自己的理解和领悟，把许多知识点汇入到这本书里。它们都不是潮流的知识点，而是厚重的基础知识，因此，本书值得保存在你的书架里很多年。

我不敢推测大家是否喜欢这本书，但是如果是我自己，刚工作 3~5 年时看到这本书，一定会非常喜欢。

我力图让“本书处处充满干货”这句话落到实处。如果你工作了 3~5 年，我相信这本书会对你的技术提升有立竿见影的效果。如果你才开始工作，它能帮你建立良好的代码世界观，更容易理解代码的世界。

我深知，充分理解这些技术的过程是枯燥的。为了保持易读性和趣味性，我尽量让书里的每段代码都足够短，代码难度尽量低。书里还有大量的比喻，每一句都是仔细斟酌过的，自认为类比蛮贴切的。我一直认为类比推论如果相似度不高，其实会起反效果，那还不如不用。所以每当我想出了一个绝好的比喻时，可能比想通一个技术难点还开心。

为了保证原创性，我尽量避免看同类的文章和书，生怕思路被带走。虽然这样做，有些结论可能失之偏颇，但我认为值得。所以请每一位读者带着怀疑的态度来阅读本书，这样能让你们受益最多。

最后，总结一下本书的几个要点。

- 编程最基础的是语法，但语法仅仅是编程最底层的强制性约束。在这之上还有很多需要自我约束的规则，而这些规则正是本书要讲的重点。比如：
  - 如何面向抽象编程和面向接口编程；
  - 耦合的本质，解耦的原则；
  - 用面向对象的方式看世界，以及对象之间的关系；
  - 把变化抽象成数据；
  - 把容易变化的逻辑，放在容易改变的地方；
  - 隐式约定和显式约定。
- 本书的代码并没有局限于某种特定语言，基本上是针对某种场合，哪种语言描述最合理、最简洁，就采取哪种。有的知识点即使不是所有语言都支持的，但肯定也是绝大多数语言都支持的功能。而且，我也相信一个程序员一生不可能只用一种编程语言。
- 很多知识点阐述的角度可能与一般人不同。我喜欢见微知著，从小往大讲。所以书中会从简单的 `if...else` 语句深入到开闭设计原则，从 `static` 关键字深入到类扩展，从 `bool` 变量深入到描述业务的规约，从散列表深入到控制反转原则，从数据化过渡到反射。我相信一个技术哪怕你懂了，当换个角度去理解它时，还是能收获很多新东西。

- 每一种技术都会对应着各自的应用场景，不熟悉它们的应用场景，就存在滥用的风险。而这些只能靠多年的经验去积累。所以本书极其强调需求背景，也就是应用场景，这也是它区别于其他书最重要的特征。套用一句话："一切不以应用场景为背景去探讨设计优劣的，都是耍流氓。"本书有大量我经历的真实案例，里面都描述了详细的需求背景。你看书的时候，就像身边有一位老程序员，隔着时空，把他多年的实战经历和思考的结晶对你娓娓道来。
- 本书包含了很多编程之外的章节，即"武戏"之余还添有"文戏"，例如：
  - 编程就是用数学来写作；
  - 语言到底哪种好；
  - 程序员的精神分裂；
  - 程序员的组织生产；
  - 程序员的技术成长。

  这些章节和写代码并没有直接关系，但我认为这些都是代码世界观的延伸，都是为了更好地去理解代码世界。我相信，很多程序员对这些话题也会很感兴趣，这些内容值得一看。

目　录

第 1 章　程序世界的两个基本元素 …… 1

1.1　数据和代码的相互伪装 …… 1
1.2　数据和代码的关系 …… 2
1.3　总结 …… 3

第 2 章　用面向对象的方式去理解世界 …… 4

2.1　好的程序员是安徒生 …… 4
2.2　封装——招兵买马，等级森严 …… 5
2.2.1　从单细胞到高等生物 …… 5
2.2.2　public——对象的外观 …… 6
2.2.3　private——水下的冰川 …… 10
2.2.4　protected——内外兼修 …… 11
2.2.5　封装总结 …… 12
2.3　继承——快速进化 …… 13
2.4　多态——抽象的基石 …… 15
2.5　总结 …… 16

第 3 章　面向抽象编程——玩玩虚的更健康 …… 17

3.1　抽象最讨厌的敌人：new …… 17
3.2　消灭 new 的两件武器 …… 19
3.2.1　控制反转——脏活让别人去干 …… 19
3.2.2　工厂模式——抽象的基础设施 …… 20
3.2.3　new 去哪里了呢 …… 21
3.3　抽象到什么程度 …… 22
3.4　总结 …… 23

第 4 章　耦合其实无处不在 …… 24

4.1　耦合的种类 …… 24
4.1.1　数据之间的耦合 …… 24
4.1.2　函数之间的耦合 …… 24
4.1.3　数据和函数之间的耦合 …… 25
4.1.4　耦合种类的总结 …… 26
4.2　耦合中既有敌人也有朋友 …… 26
4.3　坏耦合的原因 …… 28
4.3.1　刻舟求剑 …… 28
4.3.2　“谈恋爱”是个危险的行为 …… 29
4.3.3　侵占公共资源 …… 29
4.3.4　需求变化——防不胜防 …… 30
4.4　解耦的原则 …… 30
4.4.1　让模块逻辑独立而完整 …… 30
4.4.2　让连接桥梁坚固而兼容 …… 34
4.5　总结 …… 35

第 5 章　数据的种类——生命如此多娇 …… 36

5.1　常用数据类型 …… 36
5.1.1　string 类型：人机沟通的桥梁 …… 36
5.1.2　int 类型：多变的万花筒 …… 37
5.1.3　bool 类型：能量巨大的原子 …… 37
5.2　按生命周期划分数据 …… 39
5.3　两个重要的数据容器 …… 40
5.3.1　数组——容器之王 …… 40
5.3.2　散列表——银行的保险柜 …… 41
5.3.3　容器总结 …… 46
5.4　对象的种类 …… 46
5.4.1　实体对象——光鲜的主角 …… 46
5.4.2　值对象——配角或道具 …… 46
5.5　描述数据的数据 …… 47
5.6　总结 …… 48

第 6 章 数据驱动——把变化抽象成数据 ……49
6.1 三个案例 ……49
6.2 数据驱动的好帮手：反射 ……53
6.2.1 反射是将代码数据化 ……54
6.2.2 反射也是一把双刃剑 ……54
6.2.3 各种语言对反射的支持 ……55
6.3 总结 ……55
第 7 章 对象之间的关系——父子、朋友或情人 ……57
7.1 继承——父子关系 ……57
7.1.1 里氏替换原则——儿子顶替父亲 ……57
7.1.2 鸵鸟非鸟 ……58
7.1.3 不听老人言 ……60
7.2 组合——朋友关系 ……61
7.2.1 组合与继承的区别 ……62
7.2.2 组合和继承的联系 ……63
7.2.3 策略模式——组装车间 ……64
7.2.4 组合的总结 ……66
7.3 依赖——情人关系 ……67
7.3.1 依赖和组合的差别 ……67
7.3.2 迷人的双向依赖 ……68
7.3.3 扑朔迷离的访问者模式 ……69
7.3.4 依赖的总结 ……74
7.4 总结 ……74
第 8 章 函数的种类——迷宫的结构 ……75
8.1 面向对象的函数叫方法 ……75
8.2 参数是函数的原材料 ……75
8.2.1 参数在函数中的地位 ……75
8.2.2 参数存在的形式 ……76
8.3 返回值对函数的意义 ……76
8.3.1 有返回值函数 ……77
8.3.2 void 函数 ……77
8.4 值传递、引用传递和指针传递 ……78
8.5 有状态函数和无状态函数 ……80
8.6 静态函数和普通函数 ……82
8.7 能驾驭其他函数的函数 ……84
8.8 编译器做过手脚的函数 ……84
8.8.1 函数重载 ……84
8.8.2 泛型函数 ……85
8.9 总结 ……86
第 9 章 面向接口编程——遵循契约办事 ……87
9.1 接口和抽象类——分工其实挺明确 ……87
9.2 接口的应用场景 ……89
9.2.1 先签约，后对接 ……90
9.2.2 专注抽象，脱离具体 ……92
9.2.3 解开耦合，破除缠绕 ……93
9.2.4 3 个场景的总结 ……95
9.3 接口和函数指针 ……95
9.3.1 原来是亲兄弟 ……95
9.3.2 接口的优势 ……96
9.3.3 函数指针的优势 ……97
9.3.4 两兄弟的总结 ……99
9.4 函数指针的应用场景 ……99
9.4.1 简化版的 Command 模式 ……99
9.4.2 行为外包 ……101
9.4.3 结尾回调——异步搭档 ……102
9.5 总结 ……104
第 10 章 if...else 的多面性 ……105
10.1 两条兄弟语句 ……105
10.2 if...else 的黑暗面 ……106
10.2.1 永无止境的长长铁链 ……106
10.2.2 牵一发而动全身 ……107
10.2.3 其实黑化不是我的错 ……108
10.3 开闭原则——if...else 的天敌 ……108
10.3.1 扩展和修改的区别 ……109
10.3.2 为什么扩展比修改好 ……110
10.4 化解 if...else 黑暗面 ……110
10.4.1 抽出共性 ……110
10.4.2 利用多态 ……112
10.4.3 数据驱动 ……114
10.4.4 动态类型 ……114
10.5 总结 ……116

第 11 章　挖掘一件神秘武器——static……117
11.1　static 神秘在哪里……117
11.2　static 的特性……118
11.2.1　对代码的直接访问……118
11.2.2　隔离性和游离性……119
11.2.3　将函数参数反客为主……119
11.3　static 的应用场景……120
11.3.1　实现工具包函数……120
11.3.2　实现单例也有门道……120
11.3.3　实现类扩展……123
11.3.4　让数据互动起来……128
11.3.5　构建上层建筑……128
11.4　总结……131
第 12 章　把容易变化的逻辑，放在容易修改的地方……132
12.1　一个和用户的故事……132
12.2　一个和销售的故事……134
12.3　一个和产品经理的故事……136
12.4　一个和运维的故事……136
12.5　总结……137
第 13 章　隐式约定——犹抱琵琶半遮面……139
13.1　拨开隐式约定的神秘面纱……139
13.1.1　隐式约定就在你身边……139
13.1.2　隐式约定的重要特征……141
13.1.3　隐式约定的其他形式……142
13.1.4　隐式约定的风险与缺陷……142
13.2　调料包数据……143
13.3　越简单的功夫越厉害……145
13.4　总结……148
第 14 章　异常，天使还是魔鬼……150
14.1　三个江湖派别……150
14.2　异常的种类……151
14.3　异常的 throw：手榴弹什么时候扔……153
14.4　异常的 catch——能收炸弹的垃圾筐……154
14.5　异常的使用技巧……156
14.5.1　看病要趁早……156
14.5.2　不要加大 catch 的负担……156
14.5.3　避免 try 花了眼……157
14.5.4　保持克制，不要滥用……157
14.6　总结……158
第 15 章　多线程编程——在混沌中永生……159
15.1　几个基础概念……159
15.1.1　每个线程都有独立的安全港——栈区……159
15.1.2　超乎想象的细微步骤——线程安全……160
15.2　互斥——相互竞争……161
15.3　同步——相互协作……163
15.3.1　同步的本质……163
15.3.2　共享变量——一块公用的黑板……164
15.3.3　条件变量——用交通灯来指挥……165
15.3.4　同步和互斥——本是同根生……165
15.4　异步——各忙各的……166
15.4.1　异步的本质……166
15.4.2　等待烧水，顺便洗碗……167
15.4.3　明修栈道，暗度陈仓……167
15.4.4　异步和函数回调……169
15.4.5　有关异步的问与答……170
15.4.6　异步总结……171
15.5　阻塞与非阻塞……172
15.6　总结……173
第 16 章　单元测试——对代码庖丁解牛……174
16.1　单元测试的诞生……174
16.2　单元测试的进化……175
16.2.1　大量繁殖……175
16.2.2　寻找盟友……176
16.2.3　划分地盘……176

16.2.4 反客为主 ····· 177
16.3 编写单元测试的基本原则 ····· 178
16.4 如何让代码面向单元测试 ····· 181
16.4.1 买一个西瓜，无须先买菜市场 ····· 181
16.4.2 只是演习，不玩真的 ····· 183
16.4.3 人机交互代码，怎么攻克 ····· 183
16.5 最后的忠告：无招胜有招 ····· 185
16.6 总结 ····· 186

第 17 章 代码评审——给身体排排毒 ····· 187
17.1 排毒要养成习惯 ····· 187
17.2 磨刀不误砍柴工 ····· 188
17.3 经验点滴——关键是流程化 ····· 188
17.4 11 个案例 ····· 189
17.5 总结 ····· 196

第 18 章 编程就是用代码来写作 ····· 197
18.1 程序员与作家的区别 ····· 197
18.2 如何提高写作水平 ····· 198
18.2.1 英语还是躲不了的 ····· 198
18.2.2 重视的态度 ····· 198
18.2.3 需要长期的积累 ····· 199
18.3 案例解析——咬文嚼字很重要 ····· 201
18.4 谨慎对待注释 ····· 202
18.4.1 必须存在的注释 ····· 203
18.4.2 做做样子的注释 ····· 203
18.5 总结 ····· 204

第 19 章 程序员的精神分裂——扮演上帝与木匠 ····· 205
19.1 一个脑袋，两种身份 ····· 205
19.2 上帝模式：开天辟地，指点江山 ····· 205
19.2.1 “上帝”在干什么 ····· 206
19.2.2 和产品设计的争夺 ····· 206
19.3 木匠模式：致富只有勤劳一条路 ····· 208
19.4 总结 ····· 209

第 20 章 程序员的技术成长——打怪升级之路 ····· 210
20.1 技术成长三部曲 ····· 210
20.2 码农都是好老师 ····· 211
20.3 重视编程效率 ····· 212
20.4 尽量通过工作去锻炼 ····· 212
20.5 三分之一的工匠精神 ····· 214
20.6 明白架构师的含义 ····· 214
20.7 总结 ····· 214

第 21 章 语言到底哪种好——究竟谁是屠龙刀 ····· 216
21.1 军队的背后是国家实力的较量 ····· 216
21.2 专一和多情哪个好 ····· 216
21.2.1 切换语言的成本到底有多大 ····· 217
21.2.2 海、陆、空齐备最好 ····· 217
21.3 如何快速学习一门新语言 ····· 218
21.3.1 边学边练 ····· 219
21.3.2 抓住该语言的主要特性去学 ····· 219
21.4 总结 ····· 219

第 22 章 程序员的组织生产——让大家更高效和亲密 ····· 220
22.1 敏捷开发：及时反馈，小步快跑 ····· 220
22.2 双人编程：双人搭配，干活超累 ····· 222
22.3 封闭开发：并不是蹲大狱 ····· 222
22.4 总结 ····· 223

第 23 章 程序员的职业生涯——选择比努力更重要 ····· 224
23.1 程序员到底能干多久 ····· 224
23.2 程序员的中年危机 ····· 225
23.3 自问一：你适不适合当程序员 ····· 226
23.4 自问二：程序员是否最适合你 ····· 227
23.5 自问三：问问自己有没有双门槛 ····· 228
23.6 自问四：程序员最适合转什么行 ····· 229
23.7 总结 ····· 230

第 1 章 程序世界的两个基本元素

每个程序的运行过程，都可以比喻成弹珠穿越迷宫的游戏。

有一个竖直方向的复杂迷宫，上面有若干入口，底下有若干出口，里面的路径连接很复杂。我们让众多大小不一、形状各异的弹珠从迷宫上面的入口顺着迷宫管道往下落，直到出口。弹珠从入口跑到出口的过程，就相当于程序运行的过程。

实际上，真实模型会更复杂一些。入口并不是弹珠的唯一来源，有的管道自己会生产弹珠往下落。此外，在运行过程中，有的弹珠会消失在管道里，永远不再出来。如果要对应多线程，迷宫模型也要相应扩展：在前后叠加多个迷宫，由平面变成立体。迷宫相互之间还有桥梁连接，路是通的。

迷宫入口的弹珠，就是程序的原始数据，这些弹珠在下落的过程中会被加工，它们可能会变大或变小，还可能分裂或组合。最终走出迷宫的弹珠，则是呈现给用户需要的最终数据。这里的迷宫管道，就是程序的代码结构。

正如水由氢元素和氧元素构成，程序世界则由数据和代码构成。

1.1 数据和代码的相互伪装

继续顺着这个思路去理解：我们写的代码里，到底哪些属于数据，哪些属于代码？

先举一个简单的例子：

```
bool flag = true;
```

这里，`true` 就是一个形状最小的弹珠，即占内存最小的数据。变量 `flag` 属于代码，也就是说 `flag` 属于迷宫管道结构的一部分。这意味着编译器遇到 `bool flag = true;`，只会判断 `true` 这个数据是不是符合 `bool` 这个类型，并不会立刻把 `true` 保存到 `flag` 中去；而是到了运行的时候，才把 `true` 这个弹珠塞到 `flag` 这个通道里通过！

但有时候双方不容易区分，甚至相互伪装。我们再看一个例子：

```
if(number == 123)
```

这里的 number 是代码，虽然 123 是数字，但也属于代码！它是已经内化到迷宫管道的数字，是属于迷宫结构的一部分。它和弹珠这种动态数据有本质的区别。搞懂这类伪装很重要，这也是我们理解第 6 章的基础。

继续看例子：

```
Person findPerson(string name, int age);
```

其中 string name 和 int age 看起来像数据，却属于代码，它们都属于函数定义里的形参。

但当这个函数被调用，实参被传入时：

```
Person person = findPerson("Jessie", 18);
```

其中的 "Jessie" 和 18 则是数据，属于弹珠。

上面这 3 个例子中，代码都伪装成了数据。

接下来再看一个例子：

```
Dictionary dic = LoadFromFile(file);
```

这里等号左边的变量 dic 是代码，等号右边的 LoadFromFile(file) 也是代码，但 LoadFromFile(file) 的返回值是数据。这意味着这两节迷宫管道结合得比较紧密，嵌套在一起没有缝隙，外面看不出来弹珠的流动。

这个例子就是数据伪装成了代码，或者说数据隐藏在了代码里面。

最后一个问题：函数指针算不算数据？如果一个函数本身作为参数，传递给另一个函数时，那它算不算数据呢？例如：

```
int addFunc(int a, int b) {
    return a + b;
}
int num = Calculate(1, 2, addFunc);
```

这时的 addFunc 算数据吗？如果是，又是一种什么样的数据？

我认为是算的。至少局部肯定算。这就好比迷宫的众多管道中，有些管道是可以移动的，自己化作数据移动，然后嵌入到另一个管道里。**就是说代码本身在特定时候，也可以充当数据。**

但是，这种数据永远只在迷宫内部转，不会出迷宫，也就是说对最终用户是不可见的。最终用户想要的，始终是弹珠这种数据。而迷宫里面是什么形状，又怎么运动，他们并不关心。

所以我们称函数指针是一种特殊的数据，具有封闭性，它的作用只是为了更灵活地处理弹珠数据。

1.2　数据和代码的关系

关系一：数据是根本目的，代码是手段，代码永远是为数据服务的。

数据分为输入数据和输出数据，代码是将输入数据转化为输出数据的工具。用户最关心的永远是最终数据是不是他想要的，并能否在规定的时间内得到；代码如何实现的，并不是用户的关心点。可能有人反驳：对于最近的围棋软件 AlphaGo，大家似乎更关心它的代码算法呀！确实，但这只是暂时好奇，不是永恒的。试想一下，100 年以后，人们在单机版的情况下和 AlphaGo 下围棋，大家早就不关心它的算法了，而是关心具体的棋局。

关系二：有什么样的数据，决定了会有什么样的代码。

有的系统处理的数据量小，有的系统处理的数据量大，两者代码的复杂度肯定不一样。

有的系统虽然数据量大，但主要躺在数据库里，像一潭死水难得动弹，有的系统则不停地实时处理大并发数据，它们的代码复杂度肯定也会不一样。

有的系统输入数据大，输出数据小，比如人工智能，系统可能要分析近百万张猫的图片，才能识别出一只新的猫。这种系统最大的难点主要是分析和加工这些输入数据是否合理和准确。

有的系统输入数据小，输出数据大，比如游戏，玩家的输入肯定是有限的，而系统要向玩家展示一个虚幻世界。这种系统最大的难点是输出数据是否贴合用户的心意。

有的系统，并非所有的输出数据都很重要，所以产生一些 bug，用户是可以容忍的；但有的系统，所有的数据处理必须万无一失，可能业务逻辑并不复杂，但要求无比复杂且精确的代码。

总之，和什么样的数据打交道，会最终决定存在什么样的代码。而代码不断地升级修改，永远是为了匹配数据的要求或者追逐数据的变化。

1.3 总结

程序的运行过程，就像弹珠穿越迷宫的过程。

数据和代码是组成程序的两个基本元素。数据是目的，代码是手段。一定要明白代码是为数据服务的，数据才是整个系统的中心。要时刻提醒自己：归根结底，面向用户的是数据。

如果重构一个系统，抓不住头绪，不妨从数据的角度进行重新梳理和思考。这样抓住源头往往能拨开迷雾，站在更高的角度去理解这个系统，从而生成最佳的想法。

特别标注：本书是基于面向对象讨论的，所以在其他章节里，“数据”一般是指类成员数据，“代码”一般是指“类函数”。

第2章

用面向对象的方式去理解世界

2

毫无疑问，面向对象是各大语言采用的主流技术。本书大部分章节也是建立在面向对象的基础之上的，而本章就是串联其他章节的基础。

从面向过程到面向对象，给程序员带来的是一种思维方式的转变。这个思维转变的影响力是巨大的，把它称为思维革命也不为过。而这种转变，是通过多个精妙技术的组合实现的。

下面就好好剖析一下什么是面向对象。

2.1 好的程序员是安徒生

通过面向对象的方式，程序员把各种事物对应成一个一个对象。所以，在现实世界中原本不会动的东西（比如书、房子等），在程序员的脑子里，把它们转变成了有生命的个体，让程序世界更像一个童话世界。进一步讲，在对象拟人化的基础上，我们把对象与对象之间的互动关系给故事化了。大家知道，人类大脑最喜欢听故事和讲故事了，因此用这种思维理解世界最省劲，容易编又记得住。所以程序员写代码的过程很精彩，好比安徒生创作童话的过程。

讲得太抽象？举个例子意会一下。

如果是面向过程，描述动物吃饭：

```
吃(动物，食物);
```

而面向对象描述同样的事情能自然一些：

```
动物.吃(食物);
```

这还不算完，随着动物具体演化，可以演变成：

```
狗.吃(骨头);
猫.吃(鱼);
```

无须定义更多的函数。你看，三两下，一个童话世界似乎要跃然纸上。

面向对象的代码会带来代码层级增加，如果站在编译器的角度，软件的复杂度有时候说不定

还上升了。由于采用最被人类理解的组织结构，因此在人类大脑里反而变简单了。面向对象的方式本质上是借用人类理解世界的方式去构架这些数据和方法之间的关系。说白了，它是人类理解这个世界最省劲的方式。面向对象的方式还能减少人类之间的沟通成本，就是说，它不但有利于开发人员去设计架构，还有利于其他程序员阅读和记忆你的架构。

那么，面向对象究竟通过哪些手段实现代码的拟人化和故事化呢？

主要是我们耳熟能详的三招：封装、继承和多态。这些是每种面向对象语言都会遵循的基本技术。而面向对象的其他技术，不同语言的侧重点会不一样。

2.2 封装——招兵买马，等级森严

“封装”，从字面上去延伸想象，可以知道它所做的是“将一些相互关联的因素装在一起”。它是面向对象的起始步骤，想要更好地面向对象，必须学会封装的相关技巧。

2.2.1 从单细胞到高等生物

一些简单的数据，好比低等生物，例如：

```
int i = 0;
bool ret = true;
```

它们有生命，但没有灵性。而一些数据已经进化到有灵性的高等生物，这就是对象。对象要成为有灵性的生命，首先依靠的是封装，例如：

```
class Person {
    public string name;
    public int age;
}
```

此时对象封装好了肉体，如果再添加一些方法，具备了行为能力，就彻底活了。例如：

```
class Person {
    public string name;
    public int age;
    public void Swim();
    public void Drive();
}
```

封装将若干数据和方法组合在一个叫“类”的身体里。数据是肉体，而方法是让身体动起来的各种行为。方法存在的意义就是操作它对应的数据。

这个身体并不是简单地把它们包裹在一起，普通的结构体 `struct` 也能处理这些事（C# 和 Swift 的 `struct` 都能包含数据和方法，C 语言的 `struct` 虽然不能直接包含方法，但是能定义函数指针这种特殊数据来代替方法）。所以，仅仅将各种元素归档在一起并不是面向对象的标志，面向对象的封装还将这些数据和方法定义进行了等级森严的区分：`public`、`private` 和 `protected`。这三种划分是各大语言最主流的分类。它们同时修饰着数据和方法，划分出这个类