阅读本书有两种原因：第一，你是个程序员；第二，你想成为更好的程序员。很好。我们需要更好的程序员。

这是本有关编写好程序的书。它充斥着代码。我们要从各个方向来考察这些代码。从顶向下，从底往上，从里而外。读完后，就能知道许多关于代码的事了。而且，我们还能说出好代码和糟糕的代码之间的差异。我们将了解到如何写出好代码。我们也会知道，如何将糟糕的代码改成好代码。

1.1 要有代码

有人也许会以为，关于代码的书有点儿落后于时代—代码不再是问题；我们应当关注模型和需求。确实，有人说过我们正在临近代码的终结点。很快，代码就会自动产生出来，不需要再人工编写。程序员完全没用了，因为商务人士可以从规约直接生成程序。

扯淡！我们永远抛不掉代码，因为代码呈现了需求的细节。在某些层面上，这些细节无法被忽略或抽象，必须明确之。将需求明确到机器可以执行的细节程度，就是编程要做的事。而这种规约正是代码。

我期望语言的抽象程度继续提升。我也期望领域特定语言的数量继续增加。那会是好事一桩。但那终结不了代码。实际上，在较高层次上用领域特定语言撰写的规约也将是代码！它也得严谨、精确、规范和详细，好让机器理解和执行。

那帮以为代码终将消失的伙计，就像是巴望着发现一种无规范数学的数学家们一般。他们巴望着，总有一天能创造出某种机器，我们只要想想、嘴都不用张就能叫它依计行事。那机器要能透彻理解我们，只有这样，它才能把含糊不清的需求翻译为可完美执行的程序，精确满足需求。

这种事永远不会发生。即便是人类，倾其全部的直觉和创造力，也造不出满足客户模糊感觉的成功系统来。如果说需求规约原则教给了我们什么，那就是归置良好的需求就像代码一样正式，也能作为代码的可执行测试来使用。

记住，代码确然是我们最终用来表达需求的那种语言。我们可以创造各种与需求接近的语言。我们可以创造帮助把需求解析和汇整为正式结构的各种工具。然而，我们永远无法抛弃必要的精确性--所以代码永存。

1.2 糟糕的代码

最近我在读 Kent Beck 著 *Implementation Patterns*（中译版《实现模式》）[1]一书的序言。他这样写道：“……本书基于一种不太牢靠的前提：好代码的确重要……”这前提不牢靠？我反对！我认为这是该领域最强固、最受支持、最被强调的前提了（我想 Kent 也知道）。我们知道好代码重要，是因为其短缺实在困扰了我们太久。

20 世纪 80 年代末，有家公司写了个很流行的杀手应用，许多专业人士都买来用。然后，发布周期开始拉长。缺陷总是不能修复。装载时间越来越久，崩溃的几率也越来越大。至今我还记得自己在某天沮丧地关掉那个程序，从此再不用它。在那之后不久，该公司就关门大吉了。

20 年后，我见到那家公司的一位早期雇员，问他当年发生了什么事。他的回答叫我愈发恐惧起来。原来，当时他们赶着推出产品，代码写得乱七八糟。特性越加越多，代码也越来越烂，最后再也没法管理这些代码了。是糟糕的代码毁了这家公司。

你是否曾为糟糕的代码所深深困扰？如果你是位有点儿经验的程序员，定然多次遇到过这类困境。我们有专用来形容这事的词：沼泽（wading）。我们趟过代码的水域。我们穿过灌木密布、瀑布暗藏的沼泽地。我们拼命想找到出路，期望有点什么线索能启发我们到底发生了什么事；但目光所及，只是越来越多死气沉沉的代码。

你当然曾为糟糕的代码所困扰过。那么—为什么要写糟糕的代码呢？

是想快点完成吗？是要赶时间吗？有可能。或许你觉得自己要干好所需的时间不够；假使花时间清理代码，老板就会大发雷霆。或许你只是不耐烦再搞这套程序，期望早点结束。或许你看了看自己承诺要做的其他事，意识到得赶紧弄完手上的东西，好接着做下一件工作。这种事我们都干过。

我们都曾经瞟一眼自己亲手造成的混乱，决定弃之而不顾，走向新一天。我们都曾经看到自己的烂程序居然能运行，然后断言能运行的烂程序总比什么都没有强。我们都曾经说过有朝一日再回头清理。当然，在那些日子里，我们都没听过勒布朗（LeBlanc）法则：稍后等于永不（Later equals never）。

1.3 混乱的代价

只要你干过两三年编程，就有可能曾被某人的糟糕的代码绊倒过。如果你编程不止两三年，也有可能被这种代码拖过后腿。进度延缓的程度会很严重。有些团队在项目初期进展迅速，但有那么一两年的时间却慢如蜗行。对代码的每次修改都影响到其他两三处代码。修改无小事。每次添加或修改代码，都得对那堆扭纹柴了然于心，这样才能往上扔更多的扭纹柴。这团乱麻越来越大，再也无法理清，最后束手无策。

随着混乱的增加，团队生产力也持续下降，趋向于零。当生产力下降时，管理层就只有一件事可做了：增加更多人手到项目中，期望提升生产力。可是新人并不熟悉系统的设计。他们搞不清楚什么样的修改符合设计意图，什么样的修改违背设计意图。而且，他们以及团队中的其他人都背负着提升生产力的可怕压力。于是，他们制造更多的混乱，驱动生产力向零那端不断下降

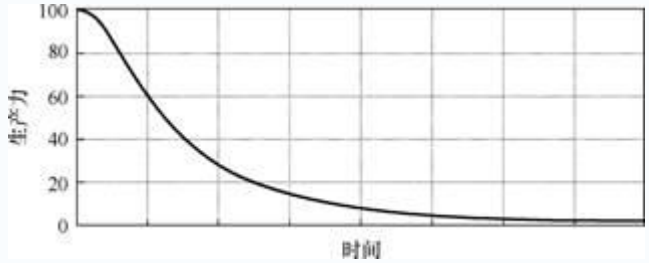

图 1-1　生产力 vs.时间

1.3.1　华丽新设计

最后，开发团队造反了，他们告诉管理层，再也无法在这令人生厌的代码基础上做开发。他们要求做全新的设计。管理层不愿意投入资源完全重启炉灶，但他们也不能否认生产力低得可怕。他们只好同意开发者的要求，授权去做一套看上去很美的华丽新设计。

于是就组建了一支新军。谁都想加入这个团队，因为它是张白纸。他们可以重新来过，搞出点真正漂亮的东西来。但只有最优秀、最聪明的家伙被选中。其余人等则继续维护现有系统。

现在有两支队伍在竞赛了。新团队必须搭建一套新系统，要能实现旧系统的所有功能。另外，还得跟上对旧系统的持续改动。在新系统功能足以抗衡旧系统之前，管理层不会替换掉旧系统。

竞赛可能会持续极长时间。我就见过延续了十年之久的。到了完成的时候，新团队的老成员早已不知去向，而现有成员则要求重新设计一套新系统，因为这套系统太烂了。

假使你经历过哪怕是一小段我谈到的这种事，那么你一定知道，花时间保持代码整洁不但有关效率，还有关生存。

1.3.2　态度

你是否遇到过某种严重到要花数个星期来做本来只需数小时即可完成的事的混乱状况？你是

否见过本来只需做一行修改，结果却涉及上百个模块的情况？这种事太常见了。

怎么会发生这种事？为什么好代码会这么快就变质成糟糕的代码？理由多得很。我们抱怨需求变化背离了初期设计。我们哀叹进度太紧张，没法干好活。我们把问题归咎于那些愚蠢的经理、苛求的用户、没用的营销方式和那些电话消毒剂。不过，亲爱的呆伯特（Dilbert）[2]，我们是自作自受[3]。我们太不专业了。

这话可不太中听。怎么会是自作自受呢？难道不关需求的事？难道不关进度的事？难道不关那些蠢经理和没用的营销手段的事？难道他们就不该负点责吗？

不。经理和营销人员指望从我们这里得到必须的信息，然后才能做出承诺和保证；即便他们没开口问，我们也不该羞于告知自己的想法。用户指望我们验证需求是否都在系统中实现了。项目经理指望我们遵守进度。我们与项目的规划脱不了干系，对失败负有极大的责任；特别是当失败与糟糕的代码有关时尤为如此！

“且慢!”你说。“不听经理的，我就会被炒鱿鱼。”多半不会。多数经理想要知道实情，即便他们看起来不喜欢实情。多数经理想要好代码，即便他们总是痴缠于进度。他们会奋力卫护进度和需求；那是他们该干的。你则当以同等的热情卫护代码。

再说明白些，假使你是位医生，病人请求你在给他做手术前别洗手，因为那会花太多时间，你会照办吗[4]？本该是病人说了算；但医生却绝对应该拒绝遵从。为什么？因为医生比病人更了解疾病和感染的风险。医生如果按病人说的办，就是一种不专业的态度（更别说是犯罪了）。

同理，程序员遵从不了解混乱风险的经理的意愿，也是不专业的做法。

心得：考虑项目进度与需求，从经理的角度看待问题

1.3.3 迷题

程序员面临着一种基础价值谜题。有那么几年经验的开发者都知道，之前的混乱拖了自己的后腿。但开发者们背负期限的压力，只好制造混乱。简言之，他们没花时间让自己做得更快！

真正的专业人士明白，这道谜题的第二部分说错了。制造混乱无助于赶上期限。混乱只会立刻拖慢你，叫你错过期限。赶上期限的唯一方法---做得快的唯一方法---就是始终尽可能保持代码整洁。

1.3.4 整洁代码的艺术

假设你相信混乱的代码是祸首，假设你接受做得快的唯一方法是保持代码整洁的说法，你一定会自问：“我怎么才能写出整洁的代码？”不过，如果你不明白整洁对代码有何意义，尝试去写整洁代码就毫无所益！

坏消息是写整洁代码很像是绘画。多数人都知道一幅画是好还是坏。但能分辨优劣并不表示懂得绘画。能分辨整洁代码和肮脏代码，也不意味着会写整洁代码！

写整洁代码，需要遵循大量的小技巧，贯彻刻苦习得的“整洁感”。这种“代码感”就是关键所在。有些人生而有之。有些人费点劲才能得到。它不仅让我们看到代码的优劣，还予我们以借戒规之力化劣为优的攻略。

缺乏“代码感”的程序员，看混乱是混乱，无处着手。有“代码感”的程序员能从混乱中看出其他的可能与变化。“代码感”帮助程序员选出最好的方案，并指导程序员制订修改行动计划，按图索骥。

简言之，编写整洁代码的程序员就像是艺术家，他能用一系列变换把一块白板变作由优雅代码构成的系统。

1.3.5 什么是整洁代码

有多少程序员，就有多少定义。所以我只询问了一些非常知名且经验丰富的程序员。

Bjarne Stroustrup，C++语言发明者，*C++ Programming Language*（中译版《C++程序设计语言》）一书作者。

我喜欢优雅和高效的代码。代码逻辑应当直截了当，叫缺陷难以隐藏；尽量减少依赖关系，使之便于维护；依据某种分层战略完善错误处理代码；性能调至最优，省得引诱别人做没规矩的优化，搞出一堆混乱来。整洁的代码只做好一件事。

Bjarne 用了“优雅”一词。说得好！我 MacBook 上的词典提供了如下定义：外表或举止上令人愉悦的优美和雅观；令人愉悦的精致和简单。注意对“愉悦”一词的强调。Bjarne 显然认为整洁的代码读起来令人愉悦。读这种代码，就像见到手工精美的音乐盒或者设计精良的汽车一般，让你会心一笑。

Bjarne 也提到效率—而且两次提及。这话出自 C++发明者之口，或许并不出奇；不过我认为并非是在单纯追求速度。被浪费掉的运算周期并不雅观，并不令人愉悦。留意 Bjarne 怎么描述那

种不雅观的结果。他用了“引诱”这个词。诚哉斯言。糟糕的代码引发混乱！别人修改糟糕的代码时，往往会越改越烂。

务实的 Dave Thomas 和 Andy Hunt 从另一角度阐述了这种情况。他们提到破窗理论[5]。窗户破损了的建筑让人觉得似乎无人照管。于是别人也再不关心。他们放任窗户继续破损。最终自己也参加破坏活动，在外墙上涂鸦，任垃圾堆积。一扇破损的窗户开辟了大厦走向倾颓的道路。

Bjarne 也提到完善错误处理代码。往深处说就是在细节上花心思。敷衍了事的错误处理代码只是程序员忽视细节的一种表现。此外还有内存泄漏，还有竞态条件代码。还有前后不一致的命名方式。结果就是凸现出整洁代码对细节的重视。

Bjarne 以“整洁的代码只做好一件事”结束论断。毋庸置疑，软件设计的许多原则最终都会归结为这句警语。有那么多人发表过类似的言论。糟糕的代码想做太多事，它意图混乱、目的含混。整洁的代码力求集中。每个函数、每个类和每个模块都全神贯注于一事，完全不受四周细节的干扰和污染。

Grady Booch，*Object Oriented Analysis and Design with Applications*（中译版《面向对象分析与设计》）一书作者。

整洁的代码简单直接。整洁的代码如同优美的散文。整洁的代码从不隐藏设计者的意图，充满了干净利落的抽象和直截了当的控制语句。

Grady 的观点与 Bjarne 的观点有类似之处，但他从可读性的角度来定义。我特别喜欢“整洁的代码如同优美的散文”这种看法。想想你读过的某本好书。回忆一下，那些文字是如何在脑中形成影像！就像是看了场电影，对吧？还不止！你还看到那些人物，听到那些声音，体验到那些喜怒哀乐。

阅读整洁的代码和阅读 *Lord of the Rings*（中译版《指环王》）自然不同。不过，仍有可类比之处。如同一本好的小说般，整洁的代码应当明确地展现出要解决问题的张力。它应当将这种张力推至高潮，以某种显而易见的方案解决问题和张力，使读者发出“啊哈！本当如此!”的感叹。

窃以为 Grady 所谓“干净利落的抽象”（crisp abstraction），乃是绝妙的矛盾修辞法。毕竟 crisp 几乎就是“具体”（concrete）的同义词。我 MacBook 上的词典这样定义 crisp 一词：果断决绝，就

事论事，没有犹豫或不必要的细节。尽管有两种不同的定义，该词还是承载了有力的信息。代码应当讲述事实，不引人猜测。它只该包含必需之物。读者应当感受到我们的果断决绝。

"老大"Dave Thomas，OTI 公司创始人，Eclipse 战略教父。

整洁的代码应可由作者之外的开发者阅读和增补。它应当有单元测试和验收测试。它使用有意义的命名。它只提供一种而非多种做一件事的途径。它只有尽量少的依赖关系，而且要明确地定义和提供清晰、尽量少的 API。代码应通过其字面表达含义，因为不同的语言导致并非所有必需信息均可通过代码自身清晰表达。

Dave 老大在可读性上和 Grady 持相同观点，但有一个重要的不同之处。Dave 断言，整洁的代码便于其他人加以增补。这看似显而易见，但亦不可过分强调。毕竟易读的代码和易修改的代码之间还是有区别的。

Dave 将整洁系于测试之上！要在十年之前，这会让人大跌眼镜。但测试驱动开发（Test Driven Development）已在行业中造成了深远影响，成为基础规程之一。Dave 说得对。没有测试的代码不干净。不管它有多优雅，不管有多可读、多易理解，微乎测试，其不洁亦可知也。

Dave 两次提及"尽量少"。显然，他推崇小块的代码。实际上，从有软件起人们就在反复强调这一点。越小越好。

Dave 也提到，代码应在字面上表达其含义。这一观点源自 Knuth 的"字面编程"（literate programming）[6]。结论就是应当用人类可读的方式来写代码。

Michael Feathers，*Working Effectively with Legacy Code*（中译版《修改代码的艺术》）一书作者。

我可以列出我留意到的整洁代码的所有特点，但其中有一条是根本性的。整洁的代码总是看起来像是某位特别在意它的人写的。几乎没有改进的余地。代码作者什么都想到了，如果你企图改进它，总会回到原点，赞叹某人留给你的代码—全心投入的某人留下的代码。

一言以蔽之：在意。这就是本书的题旨所在。或许该加个副标题，如何在意代码。

Michael 一针见血。整洁代码就是作者着力照料的代码。有人曾花时间让它保持简单有序。他们适当地关注到了细节。他们在意过。

Ron Jeffries，*Extreme Programming Installed*（中译版《极限编程实施》）以及 *Extreme Programming Adventures in C#*（中译版《C#极限编程探险》）作者。

Ron 初入行就在战略空军司令部（Strategic Air Command）编写 Fortran 程序，此后几乎在每种机器上编写过每种语言的代码。他的言论值得咀嚼。

近年来，我开始研究贝克的简单代码规则，差不多也都琢磨透了。简单代码，依其重要顺序：

— 能通过所有测试；

— 没有重复代码；

— 体现系统中的全部设计理念；

— 包括尽量少的实体，比如类、方法、函数等。

在以上诸项中，我最在意代码重复。如果同一段代码反复出现，就表示某种想法未在代码中得到良好的体现。我尽力去找出到底那是什么，然后再尽力更清晰地表达出来。

在我看来，有意义的命名是体现表达力的一种方式，我往往会修改好几次才会定下名字来。借助 Eclipse 这样的现代编码工具，重命名代价极低，所以我无所顾忌。然而，表达力还不只体现在命名上。我也会检查对象或方法是否想做的事太多。如果对象功能太多，最好是切分为两个或多个对象。如果方法功能太多，我总是使用抽取手段（Extract Method）重构之，从而得到一个能较为清晰地说明自身功能的方法，以及另外数个说明如何实现这些功能的方法。

消除重复和提高表达力让我在整洁代码方面获益良多，只要铭记这两点，改进脏代码时就会大有不同。不过，我时常关注的另一规则就不太好解释了。

这么多年下来，我发现所有程序都由极为相似的元素构成。例如“在集合中查找某物”。不管是雇员记录数据库还是名-值对哈希表，或者某类条目的数组，我们都会发现自己想要从集合中找到某一特定条目。一旦出现这种情况，我通常会把实现手段封装到更抽象的方法或类中。这样做好处多多。

可以先用某种简单的手段，比如哈希表来实现这一功能，由于对搜索功能的引用指向了我那个小小的抽象，就能随需应变，修改实现手段。这样就既能快速前进，又能为未来的修改预留余地。另外，该集合抽象常常提醒我留意“真正”在发生的事，避免随意实现集合行为，因为我真正需要的不过是某种简单的查找手段。

减少重复代码，提高表达力，提早构建简单抽象。这就是我写整洁代码的方法。

Ron 以寥寥数段文字概括了本书的全部内容。不要重复代码，只做一件事，表达力，小规模抽象。该有的都有了。

Ward Cunningham，Wiki 发明者，eXtreme Programming（极限编程）的创始人之一，Smalltalk 语言和面向对象的思想领袖。所有在意代码者的教父。

如果每个例程都让你感到深合己意，那就是整洁代码。如果代码让编程语言看起来像是专为解决那个问题而存在，就可以称之为漂亮的代码。

这种说法很 Ward。它教你听了之后就点头，然后继续听下去。如此在理，如此浅显，绝不故作高深。你大概以为此言深合己意吧。再走近点看看。

“……深合己意”你最近一次看到深合己意的模块是什么时候？模块多半都繁复难解吧？难道没有触犯规则吗？你不是也曾挣扎着想抓住些从整个系统中散落而出的线索，编织进你在读的那个模块吗？你最近一次读到某段代码、并且如同对 Ward 的说法点头一般对这段代码点头，是什么时候的事了？

Ward 期望你不会为整洁代码所震惊。你无需花太多力气。那代码就是深合你意。它明确、简单、有力。每个模块都为下一个模块做好准备。每个模块都告诉你下一个模块会是怎样的。整洁的程序好到你根本不会注意到它。设计者把它做得像一切其他设计般简单。

那 Ward 有关“美”的说法又如何呢？我们都曾面临语言不是为要解决的问题所设计的困境。但

Ward 的说法又把球踢回我们这边。他说，漂亮的代码让编程语言像是专为解决那个问题而存在！所以，让语言变得简单的责任就在我们身上了！当心，语言是冥顽不化的！是程序员让语言显得简单。

1.4 思想流派

我（鲍勃大叔）又是怎么想的呢？在我眼中整洁代码是什么样的？本书将以详细到吓死人的程度告诉你，我和我的同道对整洁代码的看法。我们会告诉你关于整洁变量名的想法，关于整洁函数的想法，关于整洁类的想法，如此等等。我们视这些观点为当然，且不为其逆耳而致歉。对我们而言，在职业生涯的这个阶段，这些观点确属当然，也是我们整洁代码派的圭旨。

武术家从不认同所谓最好的武术，也不认同所谓绝招。武术大师们常常创建自己的流派，聚徒而授。因此我们才看到格雷西家族在巴西开创并传授的格雷西柔术（Gracie Jiu Jistu），看到奥山龙峰（Okuyama Ryuho）在东京开创并传授的八光流柔术（Hakkoryu Jiu Jistu），看到李小龙（Bruce Lee）在美国开创并传授的截拳道（Jeet Kune Do）。

弟子们沉浸于创始人的授业。他们全心师从某位师傅，排斥其他师傅。弟子有所成就后，可以转投另一位师傅，扩展自己的知识与技能。有些弟子最终百炼成钢，创出新招数，开宗立派。

任何门派都并非绝对正确。不过，身处某一门派时，我们总以其所传之技为善。归根结底，练习八光流柔术或截拳道，自有其善法，但这并不能否定其他门派所授之法。

可以把本书看作是对象导师（Object Mentor）[7]整洁代码派的说明。里面要传授的就是我们勤操己艺的方法。如果你遵从这些教诲，你就会如我们一般乐受其益，你将学会如何编写整洁而专业的代码。但无论如何也别错以为我们是“正确的”。其他门派和师傅和我们一样专业。你有必要也向他们学习。

实际上，书中很多建议都存在争议。或许你并不完全同意这些建议。你可能会强烈反对其中一些建议。这样挺好的。我们不能要求做最终权威。另外一方面，书中列出的建议，乃是我们长久苦思、从数十年的从业经验和无数尝试与错误中得来。无论你同意与否，如果你没看到或是不尊敬我们的观点，就真该自己害臊。

1.5 我们是作者

Javadoc 中的@author 字段告诉我们自己是什么人。我们是作者。作者都有读者。实际上，作者有责任与读者做良好沟通。下次你写代码的时候，记得自己是作者，要为评判你工作的读者写代码。

你或许会问：代码真正“读”的成分有多少呢？难道力量主要不是用在“写”上吗？

你是否玩过“编辑器回放”？20 世纪 80、90 年代，Emac 之类编辑器记录每次击键动作。你可以在一小时工作之后，回放击键过程，就像是看一部高速电影。我这么做过，结果很有趣。

回放过程显示，多数时间都是在滚动屏幕、浏览其他模块！

鲍勃进入模块。

他向下滚动到要修改的函数。

他停下来考虑可以做什么。

哦，他滚动到模块顶端，检查变量初始化。

现在他回到修改处，开始键入。

喔，他删掉了键入的内容。

他重新键入。

他又删除了！

他键入了一半什么东西，又删除掉。

他滚动到调用要修改函数的另一函数，看看是怎么调用的。

他回到修改处，重新键入刚才删掉的代码。

他停下来。

他再一次删掉代码！

他打开另一个窗口，查看别的子类。那是个复载函数吗？

……

你该明白了。读与写花费时间的比例超过 10:1 。写新代码时，我们一直在读旧代码。既然比例如此之高，我们就想让读的过程变得轻松，即便那会使得编写过程更难。没可能光写不读，所以使之易读实际也使之易写。

这事概无例外。不读周边代码的话就没法写代码。编写代码的难度，取决于读周边代码的难度。要想干得快，要想早点做完，要想轻松写代码，先让代码易读吧。

1.6 童子军军规

光把代码写好可不够。必须时时保持代码整洁。我们都见过代码随时间流逝而腐坏。我们应当更积极地阻止腐坏的发生。

借用美国童子军一条简单的军规，应用到我们的专业领域：

让营地比你来时更干净。[8]

如果每次签入时，代码都比签出时干净，那么代码就不会腐坏。清理并不一定要花多少功夫，也许只是改好一个变量名，拆分一个有点过长的函数，消除一点点重复代码，清理一个嵌套 if 语句。

你想要为一个代码随时间流逝而越变越好的项目工作吗？你还能相信有其他更专业的做法吗？难道持续改进不是专业性的内在组成部分吗？

1.7 前传与原则

从许多角度看，本书都是我 2002 年写那本 *Agile Software Development：Principles，Patterns，and Practices*（中译版《敏捷软件开发：原则、模式与实践》，简称 PPP）的“前传”。PPP 关注面向对象设计的原则，以及专业开发者采用的许多实践方法。假如你没读过 PPP，你会发现它像这本书的延续。如果你读过，会发现那本书的主张在代码层面于本书中回响。

在本书中，你会发现对不同设计原则的引用，包括单一权责原则（Single Responsibility Principle，SRP）、开放闭合原则（Open Closed Principle，OCP）和依赖倒置原则（Dependency Inversion Principle，DIP）等。

1.8 小结

艺术书并不保证你读过之后能成为艺术家，只能告诉你其他艺术家用过的工具、技术和思维过程。本书同样也不担保让你成为好程序员。它不担保能给你“代码感”。它所能做的，只是展示好程序员的思维过程，还有他们使用的技巧、技术和工具。

和艺术书一样，本书也充满了细节。代码会很多。你会看到好代码，也会看到糟糕的代码。你会看到糟糕的代码如何转化为好代码。你会看到启发、规条和技巧的列表。你会看到一个又一个例子。但最终结果取决于你自己。

还记得那个关于小提琴家在去表演的路上迷路的老笑话吗？他在街角拦住一位长者，问他怎么才能去卡耐基音乐厅（Carnegie Hall）。长者看了看小提琴家，又看了看他手中的琴，说道：“你还得练，孩子，还得练！”

1.9 文献

[Beck07]：*Implementation Patterns*, Kent Beck, Addison-Wesley, 2007.

[Knuth92]：*Literate Programming*, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.Tim Ottinger

2.1 介绍

软件中随处可见命名。我们给变量、函数、参数、类和封包命名。我们给源代码及源代码所在目录命名。我们给 jar 文件、war 文件和 ear 文件命名。我们命名、命名，不断命名。既然有这么多命名要做，不妨做好它。下文列出了取个好名字的几条简单规则。

2.2 名副其实、见名知意

名副其实说起来简单。我们想要强调，这事很严肃。选个好名字要花时间，但省下来的时间比花掉的多。注意命名，而且一旦发现有更好的名称，就换掉旧的。这么做，读你代码的人（包括你自己）都会更开心。通过看代码的名称，就能知道代码具体要干什么

变量、函数或类的名称应该已经答复了所有的大问题。它该告诉你，它为什么会存在，它做什么事，应该怎么用。如果名称需要注释来补充，那就不算是名副其实。

```
int d;  // 消逝的时间，以日计
```

名称 d 什么也没说明。它没有引起对时间消逝的感觉，更别说以日计了。我们应该选择指明了计量对象和计量单位的名称：

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

选择体现本意的名称能让人更容易理解和修改代码。下列代码的目的何在？

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

为什么难以说明上列代码要做什么事？里面并没有复杂的表达式。空格和缩进中规中矩。只用到三个变量和两个常量。甚至没有涉及任何其他类或多态方法，只是（或者看起来是）一个数组的列表而已。

问题不在于代码的简洁度，而是在于代码的模糊度：即上下文在代码中未被明确体现的程度。上列代码要求我们了解类似以下问题的答案：

（1）theList 中是什么类型的东西？

（2）theList 零下标条目的意义是什么？

（3）值 4 的意义是什么？

（4）我怎么使用返回的列表？

问题的答案没体现在代码段中，可那就是它们该在的地方。比方说，我们在开发一种扫雷游戏，我们发现，盘面是名为 theList 的单元格列表，那就将其名称改为 gameBoard。

盘面上每个单元格都用一个简单数组表示。我们还发现，零下标条目是一种状态值，而该种状态值为 4 表示“已标记”。只要改为有意义的名称，代码就会得到相当程度的改进：

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED) 状态量大写
      flaggedCells.add(cell);
  return flaggedCells;
}
```

注意，代码的简洁性并未被触及。运算符和常量的数量全然保持不变，嵌套数量也全然保持不变。但代码变得明确多了。

还可以更进一步，不用 int 数组表示单元格，而是另写一个类。该类包括一个名副其实的函数（称为 isFlagged），从而掩盖住那个魔术数[9]。于是得到函数的新版本：

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

只要简单改一下名称，就能轻易知道发生了什么。这就是选用好名称的力量。

2.3 避免误导

程序员必须避免留下掩藏代码本意的错误线索。应当避免使用与本意相悖的词。例如，hp、aix 和 sco 都不该用做变量名，因为它们都是 UNIX 平台或类 UNIX 平台的专有名称。即便你是在编写三角计算程序，hp 看起来是个不错的缩写[10]，但那也可能会提供错误信息。

别用 accountList 来指称一组账号，除非它真的是 List 类型。List 一词对程序员有特殊意义。如果包纳账号的容器并非真是个 List，就会引起错误的判断[11]。所以，用 accountGroup 或 bunchOfAccounts，甚至直接用 accounts 都会好一些。

避免直接使用容器类型

提防使用不同之处较小的名称。想区分模块中某处的 XYZControllerFor EfficientHandlingOfStrings 和另一处的 XYZControllerForEfficientStorageOfStrings，会花多长时间呢？这两个词外形实在太相似了。

以同样的方式拼写出同样的概念才是信息。拼写前后不一致就是误导。我们很享受现代 Java 编程环境的自动代码完成特性。键入某个名称的前几个字母，按一下某个热键组合（如果有的话），就能得到一列该名称的可能形式。假如相似的名称依字母顺序放在一起，且差异很明显，那就会相当有助益，因为程序员多半会压根不看你的详细注释，甚至不看该类的方法列表就直接看名字挑一个对象。

避免使用有歧义的字母与数字

误导性名称真正可怕的例子，是用小写字母 l 和大写字母 O 作为变量名，尤其是在组合使用的时候。当然，问题在于它们看起来完全像是常量“壹”和“零”。

```
int a = l;
if (O == l)
  a = O1;
else
  l = 01;
```

读者可能会认为这纯属虚构，但我们确曾见过充斥这类玩意的代码。有一次，代码作者建议用不同字体写变量名，好显得更清楚些，不过这种方案得要通过口头和书面传递给未来所有的开发者才行。后来，只是做了简单的重命名操作，就解决了问题，而且也没搞出别的事。

2.4 做有意义的区分

如果程序员只是为满足编译器或解释器的需要而写代码，就会制造麻烦。例如，因为同一作用范围内两样不同的东西不能重名，你可能会随手改掉其中一个的名称。有时干脆以错误的拼写充数，结果就是出现在更正拼写错误后导致编译器出错的情况。[12]

光是添加数字系列或是废话远远不够，即便这足以让编译器满意。如果名称必须相异，那其意思也应该不同才对。

以数字系列命名（a1、a2，……aN）是依义命名的对立面。这样的名称纯属误导—完全没有提供正确信息；没有提供导向作者意图的线索。试看：

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

如果参数名改为 source 和 destination，这个函数就会像样许多。

废话是另一种没意义的区分。假设你有一个 Product 类。如果还有一个 ProductInfo 或 ProductData 类，那它们的名称虽然不同，意思却无区别。Info 和 Data 就像 a、an 和 the 一样，是意义含混的废话。

命名要有实质意义上的区分

注意，只要体现出有意义的区分，使用 a 和 the 这样的前缀就没错。例如，你可能把 a 用在域内变量，而把 the 用于函数参数[13]。但如果你已经有一个名为 zork 的变量，又想调用一个名为 theZork 的变量，麻烦就来了。

废话都是冗余。Variable 一词永远不应当出现在变量名中。Table 一词永远不应当出现在表名中。NameString 会比 Name 好吗？难道 Name 会是一个浮点数不成？如果是这样，就触犯了关于误导的规则。设想有个名为 Customer 的类，还有一个名为 CustomerObject 的类。区别何在呢？哪一个是表示客户历史支付情况的最佳途径？

有个应用反映了这种状况。为当事者讳，我们改了一下，不过犯错的代码的确就是这个样子：

```
getActiveAccount();
```

```
getActiveAccounts();
getActiveAccountInfo();
```

程序员怎么能知道该调用哪个函数呢？

如果缺少明确约定，变量 moneyAmount 就与 money 没区别，customerInfo 与 customer 没区别，accountData 与 account 没区别，theMessage 也与 message 没区别。要区分名称，就要以读者能鉴别不同之处的方式来区分。

2.5 使用读得出来的名称

人类长于记忆和使用单词。大脑的相当一部分就是用来容纳和处理单词的。单词能读得出来。人类进化到大脑中有那么大的一块地方用来处理言语，若不善加利用，实在是种耻辱。

如果名称读不出来，讨论的时候就会像个傻鸟。“哎，这儿，鼻涕阿三喜摁踢（bee cee arr three cee enn tee）[14]上头，有个皮挨死极翘（pee ess zee kyew）[15]整数，看见没？”这不是小事，因为编程本就是一种社会活动。

有家公司，程序里面写了个 genymdhms（生成日期，年、月、日、时、分、秒），他们一般读作“gen why emm dee aich emm ess”[16]。我有个见字照读的恶习，于是开口就念“gen-yah-mudda-hims”。后来好些设计师和分析师都有样学样，听起来傻乎乎的。我们知道典故，所以会觉得很搞笑。搞笑归搞笑，实际是在强忍糟糕的命名。在给新开发者解释变量的意义时，他们总是读出傻乎乎的自造词，而非恰当的英语词。比较

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /*  ...  */
};
```

和

```
class Customer {
  private Date generationTimestamp;
```

```
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /*  ...  */
};
```

现在读起来就像人话了：“喂，Mikey，看看这条记录！生成时间戳（generation timestamp）[17] 被设置为明天了！不能这样吧？”

2.6 使用可搜索的名称

单字母名称和数字常量有个问题，就是很难在一大篇文字中找出来。

找 MAX_CLASSES_PER_STUDENT 很容易，但想找数字 7 就麻烦了，它可能是某些文件名或其他常量定义的一部分，出现在因不同意图而采用的各种表达式中。如果该常量是个长数字，又被人错改过，就会逃过搜索，从而造成错误。

同样，e 也不是个便于搜索的好变量名。它是英文中最常用的字母，在每个程序、每段代码中都有可能出现。由此而见，长名称胜于短名称，搜得到的名称胜于用自造编码代写就的名称。

窃以为单字母名称仅用于短方法中的本地变量。名称长短应与其作用域大小相对应 [N5]。若变量或常量可能在代码中多处使用，则应赋其以便于搜索的名称。再比较

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

和

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
```

```
}
```

注意，上面代码中的 sum 并非特别有用的名称，不过它至少搜得到。采用能表达意图的名称，貌似拉长了函数代码，但要想想看，WORK_DAYS_PER_WEEK 要比数字 5 好找得多，而列表中也只剩下了体现作者意图的名称。

2.7 避免使用编码

编码已经太多，无谓再自找麻烦。把类型或作用域编进名称里面，徒然增加了解码的负担。没理由要求每位新人都在弄清要应付的代码之外（那算是正常的），还要再搞懂另一种编码"语言"。这对于解决问题而言，纯属多余的负担。带编码的名称通常也不便发音，容易打错。

2.7.1 匈牙利语标记法

在往昔名称长短很要命的时代，我们毫无必要地破坏了不编码的规矩，如今后悔不迭。Fortran 语言要求首字母体现出类型，导致了编码的产生。BASIC 早期版本只允许使用一个字母再加上一位数字。匈牙利语标记法（Hungarian Notation，HN）将这种态势愈演愈烈。

在 Windows 的 C 语言 API 的时代，HN 相当重要，那时所有名称要么是个整数句柄，要么是个长指针或者 void 指针，要不然就是 string 的几种实现（有不同的用途和属性）之一。那时候编译器并不做类型检查，程序员需要匈牙利语标记法来帮助自己记住类型。

现代编程语言具有更丰富的类型系统，编译器也记得并强制使用类型。而且，人们趋向于使用更小的类、更短的方法，好让每个变量的定义都在视野范围之内。

Java 程序员不需要类型编码。对象是强类型的，代码编辑环境已经先进到在编译开始前就侦测到类型错误的程度！所以，如今 HN 和其他类型编码形式都纯属多余。它们增加了修改变量、函数或类的名称或类型的难度。它们增加了阅读代码的难度。它们制造了让编码系统误导读者的可能性。

```
PhoneNumber  phoneString;
//  类型变化时，名称并不变化！
```

2.7.2 成员前缀

也不必用 m_前缀来标明成员变量。应当把类和函数做得足够小，消除对成员前缀的需要。你应当使用某种可以高亮或用颜色标出成员的编辑环境。

```
public class Part {
  private String m_dsc; // The textual description
  void setName(String name) {
    m_dsc = name;
  }
}
```

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

此外，人们会很快学会无视前缀（或后缀），只看到名称中有意义的部分。代码读得越多，眼中就越没有前缀。最终，前缀变作了不入法眼的废料，变作了旧代码的标志物。

2.7.3 接口和实现

有时也会出现采用编码的特殊情形。比如，你在做一个创建形状用的抽象工厂（Abstract Factory）。该工厂是个接口，要用具体类来实现。你怎么来命名工厂和具体类呢？IShapeFactory 和 ShapeFactory 吗？我喜欢不加修饰的接口。前导字母 I 被滥用到了说好听点是干扰，说难听点根本就是废话的程度。我不想让用户知道我给他们的是接口。我就想让他们知道那是个 ShapeFactory。如果接口和实现必须选一个来编码的话，我宁肯选择实现。ShapeFactoryImp，甚至是丑陋的 CShapeFactory，都比对接口名称编码来得好。

2.8 避免思维映射

不应当让读者在脑中把你的名称翻译为他们熟知的名称。这种问题经常出现在选择是使用问题领域术语还是解决方案领域术语时。

单字母变量名就是个问题。在作用域较小、也没有名称冲突时，循环计数器自然有可能被命名为 i 或 j 或 k。（但千万别用字母 l！）这是因为传统上惯用单字母名称做循环计数器。然而，在多数其他情况下，单字母名称不是个好选择；读者必须在脑中将它映射为真实概念。仅仅是因为有了 a 和 b，就要取名为 c，实在并非像样的理由。

程序员通常都是聪明人。聪明人有时会借脑筋急转弯炫耀其聪明。总而言之，假使你记得 r 代表不包含主机名和图式（scheme）的小写字母版 url 的话，那你真是太聪明了。

聪明程序员和专业程序员之间的区别在于，专业程序员了解，明确是王道。专业程序员善用其能，编写其他人能理解的代码。

2.9 类名

类名和对象名应该是名词或名词短语，如 Customer、WikiPage、Account 和 AddressParser。避免使用 Manager、Processor、Data 或 Info 这样的类名。类名不应当是动词。

2.10 方法名

方法名应当是动词或动词短语，如 postPayment、deletePage 或 save。属性访问器、修改器和断言应该根据其值命名，并依 Javabean 标准[18]加上 get、set 和 is 前缀。

```
string name = employee.getName();
customer.setName("mike");

if (paycheck.isPosted())...
```

重载构造器时，使用描述了参数的静态工厂方法名。例如，

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

通常好于

```
Complex fulcrumPoint = new Complex(23.0);
```

可以考虑将相应的构造器设置为 private，强制使用这种命名手段。

2.11 别扮可爱

如果名称太耍宝，那就只有同作者一般有幽默感的人才能记得住，而且还是在他们记得那个笑话的时候才行。谁会知道名为 HolyHandGrenade[19]的函数是用来做什么的呢？没错，这名字挺伶俐，不过 DeleteItems[20]或许是更好的名称。宁可明确，毋为好玩。

扮可爱的做法在代码中经常体现为使用俗话或俚语。例如，别用 whack( )[21]来表示 kill( )。别用 eatMyShorts( )[22]这类与文化紧密相关的笑话来表示 abort( )。

言到意到。意到言到。

2.12 每个概念对应一个词

给每个抽象概念选一个词，并且一以贯之。例如，使用 fetch、retrieve 和 get 来给在多个类中的同种方法命名。你怎么记得住哪个类中是哪个方法呢？很悲哀，你总得记住编写库或类的公司、机构或个人，才能想得起来用的是哪个术语。否则，就得耗费大把时间浏览各个文件头及前面的代码。

Eclipse 和 IntelliJ 之类现代编程环境提供了与环境相关的线索，比如某个对象能调用的方法列表。不过要注意，列表中通常不会给出你为函数名和参数列表编写的注释。如果参数名称来自函数声明，你就太幸运了。函数名称应当独一无二，而且要保持一致，这样你才能不借助多余的浏览就找到正确的方法。

同样，在同一堆代码中有 controller，又有 manager，还有 driver，就会令人困惑。DeviceManager 和 Protocol-Controller 之间有何根本区别？为什么不全用 controllers 或 managers？他们都是 Drivers 吗？这种名称，让人觉得这两个对象是不同类型的，也分属不同的类。

对于那些会用到你代码的程序员，一以贯之的命名法简直就是天降福音。

2.13 别用双关语

避免将同一单词用于不同目的。同一术语用于不同概念，基本上就是双关语了。如果遵循“一词一义”规则，可能在好多个类里面都会有 add 方法。只要这些 add 方法的参数列表和返回值在语义上等价，就一切顺利。

但是，可能会有人决定为“保持一致”而使用 add 这个词来命名，即便并非真的想表示这种意思。比如，在多个类中都有 add 方法，该方法通过增加或连接两个现存值来获得新值。假设要

写个新类，该类中有一个方法，把单个参数放到群集（collection）中。该把这个方法叫做 add 吗？这样做貌似和其他 add 方法保持了一致，但实际上语义却不同，应该用 insert 或 append 之类词来命名才对。把该方法命名为 add，就是双关语了。

代码作者应尽力写出易于理解的代码。我们想把代码写得让别人能一目尽览，而不必殚精竭虑地研究。我们想要那种大众化的作者尽责写清楚的平装书模式；我们不想要那种学者挖地三尺才能明白个中意义的学院派模式。

2.14 使用解决方案领域名称

记住，只有程序员才会读你的代码。所以，尽管用那些计算机科学（Computer Science，CS）术语、算法名、模式名、数学术语吧。依据问题所涉领域来命名可不算是聪明的做法，因为不该让协作者老是跑去问客户每个名称的含义，其实他们早该通过另一名称了解这个概念了。

对于熟悉访问者（VISITOR）模式的程序来说，名称 AccountVisitor 富有意义。哪个程序员会不知道 JobQueue 的意思呢？程序员要做太多技术性工作。给这些事取个技术性的名称，通常是最靠谱的做法。

2.15 使用源自所涉问题领域的名称

如果不能用程序员熟悉的术语来给手头的工作命名，就采用从所涉问题领域而来的名称吧。至少，负责维护代码的程序员就能去请教领域专家了。

优秀的程序员和设计师，其工作之一就是分离解决方案领域和问题领域的概念。与所涉问题领域更为贴近的代码，应当采用源自问题领域的名称。

2.16 添加有意义的语境

很少有名称是能自我说明的—多数都不能。反之，你需要用有良好命名的类、函数或名称空间来放置名称，给读者提供语境。如果没这么做，给名称添加前缀就是最后一招了。

设想你有名为 firstName、lastName、street、houseNumber、city、state 和 zipcode 的变量。当它们搁一块儿的时候，很明确是构成了一个地址。不过，假使只是在某个方法中看见孤零零一个 state 变量呢？你会理所当然推断那是某个地址的一部分吗？

可以添加前缀 addrFirstName、addrLastName、addrState 等，以此提供语境。至少，读者会明白这些变量是某个更大结构的一部分。当然，更好的方案是创建名为 Address 的类。这样，即便是编译器也会知道这些变量隶属某个更大的概念了。

看看代码清单 2-1 中的方法。以下变量是否需要更有意义的语境呢？函数名仅给出了部分语境；算法提供了剩下的部分。遍览函数后，你会知道 number、verb 和 pluralModifier 这三个变量是“测估”信息的一部分。不幸的是这语境得靠读者推断出来。第一眼看到这个方法时，这些变量的含义完全不清楚。

代码清单 2-1　语境不明确的变量

```
private void printGuessStatistics(char candidate, int count) {
  String number;
  String verb;
  String pluralModifier;
  if (count == 0) {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  } else if (count == 1) {
    number = "1";
    verb = "is";
    pluralModifier = "";
  } else {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }
  String guessMessage = String.format(
```

```
    "There %s %s %s%s", verb, number, candidate, pluralModifier
  );
  print(guessMessage);
}
```

上列函数有点儿过长，变量的使用贯穿始终。要分解这个函数，需要创建一个名为 GuessStatisticsMessage 的类，把三个变量做成该类的成员字段。这样它们就在定义上变作了 GuessStatisticsMessage 的一部分。语境的增强也让算法能够通过分解为更小的函数而变得更为干净利落。（如代码清单 2-2 所示。）

代码清单 2-2　有语境的变量

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;
  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
            "There %s %s %s%s",
        verb, number, candidate, pluralModifier );
  }
  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
```

```
    }
  }
  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }
  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }
  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

2.17 不要添加没用的语境

设若有一个名为“加油站豪华版”（Gas Station Deluxe）的应用，在其中给每个类添加 GSD 前缀就不是什么好点子。说白了，你是在和自己在用的工具过不去。输入 G，按下自动完成键，结果会得到系统中全部类的列表，列表恨不得有一英里那么长。这样做聪明吗？为什么要搞得 IDE 没法帮助你？

再比如，你在 GSD 应用程序中的记账模块创建了一个表示邮件地址的类，然后给该类命名为 GSDAccountAddress。稍后，你的客户联络应用中需要用到邮件地址，你会用 GSDAccountAddress 吗？这名字听起来没问题吗？在这 17 个字母里面，有 10 个字母纯属多余和与当前语境毫无关联。

只要短名称足够清楚，就要比长名称好。别给名称添加不必要的语境。

对于 Address 类的实体来说，accountAddress 和 customerAddress 都是不错的名称，不过用在类名上就不太好了。Address 是个好类名。如果需要与 MAC 地址、端口地址和 Web 地址相区别，我会考虑使用 PostalAddress、MAC 和 URI。这样的名称更为精确，而精确正是命名的要点。

2.18 最后的话

取好名字最难的地方在于需要良好的描述技巧和共有文化背景。与其说这是一种技术、商业或管理问题，还不如说是一种教学问题。其结果是，这个领域内的许多人都没能学会做得很好。

我们有时会怕其他开发者反对重命名。如果讨论一下就知道，如果名称改得更好，那大家真的会感激你。多数时候我们并不记忆类名和方法名。我们使用现代工具对付这些细节，好让自己集中精力于把代码写得就像词句篇章、至少像是表和数据结构（词句并非总是呈现数据的最佳手段）。改名可能会让某人吃惊，就像你做到其他代码改善工作一样。别让这种事阻碍你的前进步伐。

不妨试试上面这些规则，看你的代码可读性是否有所提升。如果你是在维护别人写的代码，使用重构工具来解决问题。效果立竿见影，而且会持续下去。

在编程的早年岁月，系统由程序和子程序组成。后来，在 Fortran 和 PL/1 的年代，系统由程序、子程序和函数组成。如今，只有函数存活下来。函数是所有程序中的第一组代码。本章将讨论如何写好函数。

请看代码清单 3-1。在 FitNesse[23]中，很难找到长函数，不过我还是搜寻到一个。它不光长，

而且代码也很复杂，有大量字符串、怪异而不显见的数据类型和 API。花 3 分钟时间，看能读懂多少？

代码清单 3-1　HtmlUtil.java（FitNesse 20070619）

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
            SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
```

```
  if (setup != null) {
    WikiPagePath setupPath =
      wikiPage.getPageCrawler().getFullPath(setup);
    String setupPathName = PathParser.render(setupPath);
    buffer.append("!include -setup .")
          .append(setupPathName)
          .append("\n");
  }
}
buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
  WikiPage teardown =
    PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
  if (teardown != null) {
    WikiPagePath tearDownPath =
      wikiPage.getPageCrawler().getFullPath(teardown);
    String tearDownPathName = PathParser.render(tearDownPath);
    buffer.append("\n")
          .append("!include -teardown .")
          .append(tearDownPathName)
          .append("\n");
  }
  if (includeSuiteSetup) {
    WikiPage suiteTeardown =
      PageCrawlerImpl.getInheritedPage(
          SuiteResponder.SUITE_TEARDOWN_NAME,
```

```
          wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

搞懂这个函数了吗？大概没有。有太多事发生，有太多不同层级的抽象。奇怪的字符串和函数调用，混以双重嵌套、用标识来控制的 if 语句等，不一而足。

不过，只要做几个简单的方法抽离和重命名操作，加上一点点重构，就能在 9 行代码之内搞掂（如代码清单 3-2 所示）。用 3 分钟阅读以下代码，看你能理解吗？

代码清单 3-2　HtmlUtil.java（重构之后）

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
```

```
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }
  return pageData.getHtml();
}
```

除非你正在研究 FitNesse，否则就理解不了所有细节。不过，你大概能明白，该函数包含把一些设置和拆解页放入一个测试页面，再渲染为 HTML 的操作。如果你熟悉 JUnit[24]，或许会想到，该函数归属于某个基于 Web 的测试框架。而且，这当然没错。从代码清单 3-2 中获得信息很容易，而代码清单 3-1 则晦涩难明。

是什么让代码清单 3-2 易于阅读和理解？怎么才能让函数表达其意图？该给函数赋予哪些属性，好让读者一看就明白函数是属于怎样的程序？

3.1 短小

函数的第一规则是要短小。第二条规则是还要更短小。我无法证明这个断言。我给不出任何证实了小函数更好的研究结果。我能说的是，近 40 年来，我写过各种不同大小的函数。我写过令人憎恶的长达 3000 行的厌物，也写过许多 100 行到 300 行的函数，我还写过 20 行到 30 行的。经过漫长的试错，经验告诉我，函数就该小。

在 20 世纪 80 年代，我们常说函数不该长于一屏。当然，说这话的时候，VT100 屏幕只有 24 行、80 列，而编辑器就得先占去 4 行空间放菜单。如今，用上了精致的字体和宽大的显示器，一屏里面可以显示 100 行，每行能容纳 150 个字符。每行都不应该有 150 个字符那么长。函数也不该有 100 行那么长，20 行封顶最佳。

函数到底该有多长？1991 年，我去 Kent Beck 位于奥勒冈州（Oregon）的家中拜访。我们坐到一起写了些代码。他给我看一个叫做 Sparkle（火花闪耀）的有趣的 Java/Swing 小程序。程序在屏幕上描画电影 Cinderella（《灰姑娘》）中仙女用魔棒造出的那种视觉效果。只要移动鼠标，光标

所在处就会爆发出一团令人欣喜的火花，沿着模拟重力场划落到窗口底部。肯特给我看代码的时候，我惊讶于其中那些函数尺寸之小。我看惯了 Swing 程序中长度数以里计的函数。但这个程序中每个函数都只有两行、三行或四行长。每个函数都一目了然。每个函数都只说一件事。而且，每个函数都依序把你带到下一个函数。这就是函数应该达到的短小程度！[25]

函数应该有多短小？通常来说，应该短于代码清单 3-2 中的函数！代码清单 3-2 实在应该缩短成代码清单 3-3 这个样子。

代码清单 3-3　HtmlUtil.java（再次重构之后）

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

代码块和缩进

if 语句、else 语句、while 语句等，其中的代码块应该只有一行。该行大抵应该是一个函数调用语句。这样不但能保持函数短小，而且，因为块内调用的函数拥有较具说明性的名称，从而增加了文档上的价值。

这也意味着函数不应该大到足以容纳嵌套结构。所以，函数的缩进层级不该多于一层或两层。当然，这样的函数易于阅读和理解。

3.2　只做一件事

代码清单 3-1 显然想做好几件事。它创建缓冲区、获取页面、搜索继承下来的页面、渲染路径、添加神秘的字符串、生成 HTML，如此等等。代码清单 3-1 手忙脚乱。而代码清单 3-3 则只做一件简单的事。它将设置和拆解包纳到测试页面中。

过去 30 年以来，以下建议以不同形式一再出现：

函数应该做一件事。做好这件事。只做这一件事。

问题在于很难知道那件该做的事是什么。代码清单 3-3 只做了一件事，对吧？其实也很容易看作是三件事：

（1）判断是否为测试页面；

（2）如果是，则容纳进设置和分拆步骤；

（3）渲染成 HTML。

那件事是什么？函数是做了一件事呢，还是做了三件事？注意，这三个步骤均在该函数名下的同一抽象层上。可以用简洁的 TO[26]起头段落来描述这个函数：

TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML。

（要 RenderPageWithSetupsAndTeardowns，检查页面是否为测试页，如果是测试页，就容纳进设置和分拆步骤。无论是否测试页，都渲染成 HTML）

如果函数只是做了该函数名下同一抽象层上的步骤，则函数还是只做了一件事。编写函数毕竟是为了把大一些的概念（换言之，函数的名称）拆分为另一抽象层上的一系列步骤。

代码清单 3-1 明显包括了处于多个不同抽象层级的步骤。显然，它所做的不止一件事。即便是代码清单 3-2 也有两个抽象层，这已被我们将其缩短的能力所证明。然而，很难再将代码清单 3-3 做有意义的缩短。可以将 if 语句拆出来做一个名为 includeSetupAndTeardonws IfTestpage 的函数，但那只是重新诠释代码，并未改变抽象层级。

所以，要判断函数是否不止做了一件事，还有一个方法，就是看是否能再拆出一个函数，该函数不仅只是单纯地重新诠释其实现[G34]。

函数中的区段

请看代码清单 4-7。注意，generatePrimes 函数被切分为 declarations、initializations 和 sieve 等区段。这就是函数做事太多的明显征兆。只做一件事的函数无法被合理地切分为多个区段。

3.3 每个函数一个抽象层级

要确保函数只做一件事，函数中的语句都要在同一抽象层级上。一眼就能看出，代码清单 3-

1 违反了这条规矩。那里面有 getHtml( )等位于较高抽象层的概念，也有 String pagePathName = PathParser.render(pagePath)等位于中间抽象层的概念，还有.append("\n")等位于相当低的抽象层的概念。

函数中混杂不同抽象层级，往往让人迷惑。读者可能无法判断某个表达式是基础概念还是细节。更恶劣的是，就像破损的窗户，一旦细节与基础概念混杂，更多的细节就会在函数中纠结起来。

自顶向下读代码：向下规则

我们想要让代码拥有自顶向下的阅读顺序。[27]我们想要让每个函数后面都跟着位于下一抽象层级的函数，这样一来，在查看函数列表时，就能循抽象层级向下阅读了。我把这叫做向下规则。

换一种说法。我们想要这样读程序：程序就像是一系列 TO 起头的段落，每一段都描述当前抽象层级，并引用位于下一抽象层级的后续 TO 起头段落。

To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.（要容纳设置和分拆步骤，就先容纳设置步骤，然后纳入测试页面内容，再纳入分拆步骤。）

To include the setups, we include the suite setup if this is a suite, then we include the regular setup.（要容纳设置步骤，如果是套件，就纳入套件设置步骤，然后再纳入普通设置步骤。）

To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.（要容纳套件设置步骤，先搜索"SuiteSetUp"页面的上级继承关系，再添加一个包括该页面路径的语句。）

To search the parent. . .（要搜索……）

程序员往往很难学会遵循这条规则，写出只停留于一个抽象层级上的函数。尽管如此，学习这个技巧还是很重要。这是保持函数短小、确保只做一件事的要诀。让代码读起来像是一系列自顶向下的 TO 起头段落是保持抽象层级协调一致的有效技巧。

看看本章末尾的代码清单 3-7。它展示了遵循这条原则重构的完整 testableHtml 函数。留意每

个函数是如何引出下一个函数，如何保持在同一抽象层上的。

3.4 switch 语句

写出短小的 switch 语句很难[28]。即便是只有两种条件的 switch 语句也要比我想要的单个代码块或函数大得多。写出只做一件事的 switch 语句也很难。Switch 天生要做 N 件事。不幸我们总无法避开 switch 语句，不过还是能够确保每个 switch 都埋藏在较低的抽象层级，而且永远不重复。当然，我们利用多态来实现这一点。

请看代码清单 3-4。它呈现了可能依赖于雇员类型的仅仅一种操作。

代码清单 3-4 Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
  switch (e.type) {
    case COMMISSIONED:
      return calculateCommissionedPay(e);
    case HOURLY:
      return calculateHourlyPay(e);
    case SALARIED:
      return calculateSalariedPay(e);
    default:
      throw new InvalidEmployeeType(e.type);
  }
}
```

该函数有好几个问题。首先，它太长，当出现新的雇员类型时，还会变得更长。其次，它明显做了不止一件事。第三，它违反了单一权责原则（Single Responsibility Principle[29]，SRP），因为有好几个修改它的理由。第四，它违反了开放闭合原则（Open Closed Principle[30]，OCP），因为每当添加新类型时，就必须修改之。不过，该函数最麻烦的可能是到处皆有类似结构的函数。例如，可能会有

isPayday(Employee e, Date date),

或

deliverPay(Employee e, Money pay),

如此等等。它们的结构都有同样的问题。

该问题的解决方案（如代码清单 3-5 所示）是将 switch 语句埋到抽象工厂[31]底下，不让任何人看到。该工厂使用 switch 语句为 Employee 的派生物创建适当的实体，而不同的函数，如 calculatePay、isPayday 和 deliverPay 等，则藉由 Employee 接口多态地接受派遣。

对于 switch 语句，我的规矩是如果只出现一次，用于创建多态对象，而且隐藏在某个继承关系中，在系统其他部分看不到，就还能容忍[G23]。当然也要就事论事，有时我也会部分或全部违反这条规矩。

代码清单 3-5　Employee 与工厂

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
```

```
    case HOURLY:
      return new HourlyEmployee(r);
    case SALARIED:
      return new SalariedEmploye(r);
    default:
      throw new InvalidEmployeeType(r.type);
   }
  }
}
}
```

3.5 使用描述性的名称

在代码清单 3-7 中，我把示例函数的名称从 testableHtml 改为 SetupTeardownIncluder.render。这个名称好得多，因为它较好地描述了函数做的事。我也给每个私有方法取个同样具有描述性的名称，如 isTestable 或 includeSetupAndTeardownPages。好名称的价值怎么好评都不为过。记住沃德原则：“如果每个例程都让你感到深合己意，那就是整洁代码。”要遵循这一原则，泰半工作都在于为只做一件事的小函数取个好名字。函数越短小、功能越集中，就越便于取个好名字。

别害怕长名称。长而具有描述性的名称，要比短而令人费解的名称好。长而具有描述性的名称，要比描述性的长注释好。使用某种命名约定，让函数名称中的多个单词容易阅读，然后使用这些单词给函数取个能说清其功用的名称。

别害怕花时间取名字。你当尝试不同的名称，实测其阅读效果。在 Eclipse 或 IntelliJ 等现代 IDE 中改名称易如反掌。使用这些 IDE 测试不同名称，直至找到最具有描述性的那一个为止。

选择描述性的名称能理清你关于模块的设计思路，并帮你改进之。追索好名称，往往导致对代码的改善重构。

命名方式要保持一致。使用与模块名一脉相承的短语、名词和动词给函数命名。例如，includeSetupAndTeardownPages、includeSetupPages、includeSuiteSetupPage 和 includeSetupPag

e 等。这些名称使用了类似的措辞，依序讲出一个故事。实际上，假使我只给你看上述函数序列，你就会自问：“includeTeardownPages、includeSuiteTeardownPages 和 includeTeardownPage 又会如何？”这就是所谓“深合己意”了。

3.6 函数参数

最理想的参数数量是零（零参数函数），其次是一（单参数函数），再次是二（双参数函数），应尽量避免三（三参数函数）。有足够特殊的理由才能用三个以上参数（多参数函数）—所以无论如何也不要这么做。

参数不易对付。它们带有太多概念性。所以我在代码范例中几乎不加参数。比如，以 StringBuffer 为例，我们可能不把它作为实体变量，而是当作参数来传递，那样的话，读者每次看到它都得要翻译一遍。阅读模块所讲述的故事时，includeSetupPage( )要比 includeSetupPageInto（newPage-Content）易于理解。参数与函数名处在不同的抽象层级，它要求你了解目前并不特别重要的细节（即那个 StringBuffer）。

从测试的角度看，参数甚至更叫人为难。想想看，要编写能确保参数的各种组合运行正常的测试用例，是多么困难的事。如果没有参数，就是小菜一碟。如果只有一个参数，也不太困难。有两个参数，问题就麻烦多了。如果参数多于两个，测试覆盖所有可能值的组合简直让人生畏。

输出参数比输入参数还要难以理解。读函数时，我们惯于认为信息通过参数输入函数，通过返回值从函数中输出。我们不太期望信息通过参数输出。所以，输出参数往往让人苦思之后才恍然大悟。

相较于没有参数，只有一个输入参数算是第二好的做法。SetupTeardownInclude.render（pageData）也相当易于理解。很明显，我们将渲染 pageData 对象中的数据。

3.6.1 一元函数的普遍形式

向函数传入单个参数有两种极普遍的理由。你也许会问关于那个参数的问题，就像在 boolean fileExists("MyFile")中那样。也可能是操作该参数，将其转换为其他什么东西，再输出之。例如，InputStream fileOpen("MyFile")把 String 类型的文件名转换为 InputStream 类型的返回

值。这就是读者看到函数时所期待的东西。你应当选用较能区别这两种理由的名称，而且总在一致的上下文中使用这两种形式。

还有一种虽不那么普遍但仍极有用的单参数函数形式，那就是事件（event）。在这种形式中，有输入参数而无输出参数。程序将函数看作是一个事件，使用该参数修改系统状态，例如 void passwordAttemptFailedNtimes(int attempts)。小心使用这种形式。应该让读者很清楚地了解它是个事件。谨慎地选用名称和上下文语境。

尽量避免编写不遵循这些形式的一元函数，例如，void includeSetupPageInto(StringBuffer pageText)。对于转换，使用输出参数而非返回值令人迷惑。如果函数要对输入参数进行转换操作，转换结果就该体现为返回值。实际上，StringBuffer transform(StringBuffer in)要比 void transform(StringBuffer out)强，即便第一种形式只简单地返回输参数也是这样。至少，它遵循了转换的形式。

3.6.2 标识参数

标识参数丑陋不堪。向函数传入布尔值简直就是骇人听闻的做法。这样做，方法签名立刻变得复杂起来，大声宣布本函数不止做一件事。如果标识为 true 将会这样做，标识为 false 则会那样做！

在代码清单 3-7 中，我们别无选择，因为调用者已经传入了那个标识，而我想把重构范围限制在该函数及该函数以下范围之内。方法调用 render(true)对于可怜的读者来说仍然摸不着头脑。卷动屏幕，看到 render(Boolean isSuite)，稍许有点帮助，不过仍然不够。应该把该函数一分为二：reanderForSuite( )和 renderForSingleTest( )。

3.6.3 二元函数

有两个参数的函数要比一元函数难懂。例如，writeField(name)比 writeField(outputStream, name)[32]好懂。

尽管两种情况下意义都很清楚，但第一个只要扫一眼就明白，更好地表达了其意义。第二个就得暂停一下才能明白，除非我们学会忽略第一个参数。而且最终那也会导致问题，因为我们根本就不该忽略任何代码。忽略掉的部分就是缺陷藏身之地。

当然，有些时候两个参数正好。例如，Point p = new Point(0,0);就相当合理。笛卡儿点天生拥有两个参数。如果看到 new Point(0)，我们会倍感惊讶。然而，本例中的两个参数却只是单个值的有序组成部分！而 output-Stream 和 name 则既非自然的组合，也不是自然的排序。

即便是如 assertEquals(expected, actual)这样的二元函数也有其问题。你有多少次会搞错 actual 和 expected 的位置呢？这两个参数没有自然的顺序。expected 在前，actual 在后，只是一种需要学习的约定罢了。

二元函数不算恶劣，而且你当然也会编写二元函数。不过，你得小心，使用二元函数要付出代价。你应该尽量利用一些机制将其转换成一元函数。例如，可以把 writeField 方法写成 outputStream 的成员之一，从而能这样用：outputStream.writeField(name)。或者，也可以把 outputStream 写成当前类的成员变量，从而无需再传递它。还可以分离出类似 FieldWriter 的新类，在其构造器中采用 outputStream，并且包含一个 write 方法。

3.6.4 三元函数

有三个参数的函数要比二元函数难懂得多。排序、琢磨、忽略的问题都会加倍体现。建议你在写三元函数前一定要想清楚。

例如，设想 assertEquals 有三个参数：assertEquals(message, expected, actual)。有多少次，你读到 message，错以为它是 expected 呢？我就常栽在这个三元函数上。实际上，每次我看到这里，总会绕半天圈子，最后学会了忽略 message 参数。

另一方面，这里有个并不那么险恶的三元函数：assertEquals(1.0, amount, .001)。虽然也要费点神，还是值得的。得到“浮点值的等值是相对而言”的提示总是好的。

3.6.5 参数对象

如果函数看来需要两个、三个或三个以上参数，就说明其中一些参数应该封装为类了。例如，下面两个声明的差别：

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

从参数创建对象，从而减少参数数量，看起来像是在作弊，但实则并非如此。当一组参数被共同传递，就像上例中的 x 和 y 那样，往往就是该有自己名称的某个概念的一部分。

3.6.6 参数列表

有时，我们想要向函数传入数量可变的参数。例如，String.format 方法：

```
String.format("%s worked %.2f hours.", name, hours);
```

如果可变参数像上例中那样被同等对待，就和类型为 List 的单个参数没什么两样。这样一来，String.formate 实则是二元函数。下列 String.format 的声明也很明显是二元的：

```
public String format(String format, Object... args)
```

同理，有可变参数的函数可能是一元、二元甚至三元。超过这个数量就可能要犯错了。

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

3.6.7 动词与关键字

给函数取个好名字，能较好地解释函数的意图，以及参数的顺序和意图。对于一元函数，函数和参数应当形成一种非常良好的动词/名词对形式。例如，write(name)就相当令人认同。不管这个“name”是什么，都要被“write”。更好的名称大概是 writeField(name)，它告诉我们，“name”是一个“field”。

最后那个例子展示了函数名称的关键字（keyword）形式。使用这种形式，我们把参数的名称编码成了函数名。例如，assertEqual 改成 assertExpectedEqualsActual(expected, actual)可能会好些。这大大减轻了记忆参数顺序的负担。

3.7 无副作用

副作用是一种谎言。函数承诺只做一件事，但还是会做其他被藏起来的事。有时，它会对自己类中的变量做出未能预期的改动。有时，它会把变量搞成向函数传递的参数或是系统全局变量。无论哪种情况，都是具有破坏性的，会导致古怪的时序性耦合及顺序依赖。

以代码清单 3-6 中看似无伤大雅的函数为例。该函数使用标准算法来匹配 userName 和 password。如果匹配成功，返回 true，如果失败则返回 false。但它会有副作用。你知道问题所在吗？

代码清单 3-6 UserValidator.java

```
public class UserValidator {
```

```
  private Cryptographer cryptographer;
  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

当然了，副作用就在于对 Session.initialize( )的调用。checkPassword 函数，顾名思义，就是用来检查密码的。该名称并未暗示它会初始化该次会话。所以，当某个误信了函数名的调用者想要检查用户有效性时，就得冒抹除现有会话数据的风险。

这一副作用造出了一次时序性耦合。也就是说，checkPassword 只能在特定时刻调用（换言之，在初始化会话是安全的时候调用）。如果在不合适的时候调用，会话数据就有可能沉默地丢失。时序性耦合令人迷惑，特别是当它躲在副作用后面时。如果一定要时序性耦合，就应该在函数名称中说明。在本例中，可以重命名函数为 checkPasswordAndInitializeSession，虽然那还是违反了“只做一件事”的规则。

输出参数

参数多数会被自然而然地看作是函数的输入。如果你编过好些年程序，我担保你一定被用作输出而非输入的参数迷惑过。例如：

```
appendFooter(s);
```

这个函数是把 s 添加到什么东西后面吗？或者它把什么东西添加到了 s 后面？s 是输入参数还是输出参数？稍许花点时间看看函数签名：

```
public void appendFooter(StringBuffer report)
```

事情清楚了，但付出了检查函数声明的代价。你被迫检查函数签名，就得花上一点时间。应该避免这种中断思路的事。

在面向对象编程之前的岁月里，有时的确需要输出参数。然而，面向对象语言中对输出参数的大部分需求已经消失了，因为 this 也有输出函数的意味在内。换言之，最好是这样调用 appendFooter：

```
report.appendFooter();
```

普遍而言，应避免使用输出参数。如果函数必须要修改某种状态，就修改所属对象的状态吧。

3.8 分隔指令与询问

函数要么做什么事，要么回答什么事，但二者不可得兼。函数应该修改某对象的状态，或是返回该对象的有关信息。两样都干常会导致混乱。看看下面的例子：

```
public boolean set(String attribute, String value);
```

该函数设置某个指定属性，如果成功就返回 true，如果不存在那个属性则返回 false。这样就导致了以下语句：

```
if (set("username", "unclebob"))...
```

从读者的角度考虑一下吧。这是什么意思呢？它是在问 username 属性值是否之前已设置为 unclebob 吗？或者它是在问 username 属性值是否成功设置为 unclebob 呢？从这行调用很难判断其含义，因为 set 是动词还是形容词并不清楚。

作者本意，set 是个动词，但在 if 语句的上下文中，感觉它像是个形容词。该语句读起来像是说“如果 username 属性值之前已被设置为 uncleob”，而不是“设置 username 属性值为 unclebob，看看是否可行，然后……”。要解决这个问题，可以将 set 函数重命名为 setAndCheckIfExists，但这对提高 if 语句的可读性帮助不大。真正的解决方案是把指令与询问分隔开来，防止混淆的发生：

```
if (attributeExists("username")) {
        setAttribute("username", "unclebob");
        ...
}
```

3.9 使用异常替代返回错误码

从指令式函数返回错误码轻微违反了指令与询问分隔的规则。它鼓励了在 if 语句判断中把指令当作表达式使用。

```
if (deletePage(page) == E_OK)
```

这不会引起动词/形容词混淆，但却导致更深层次的嵌套结构。当返回错误码时，就是在要求调用者立刻处理错误。

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else {
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

另一方面，如果使用异常替代返回错误码，错误处理代码就能从主路径代码中分离出来，得到简化：

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

3.9.1 抽离 Try/Catch 代码块

Try/catch 代码块丑陋不堪。它们搞乱了代码结构，把错误处理与正常流程混为一谈。最好把 try 和 catch 代码块的主体部分抽离出来，另外形成函数。

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}
private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
private void logError(Exception e) {
  logger.log(e.getMessage());
```

```
}
```

在上例中，delete 函数只与错误处理有关。很容易理解然后就忽略掉。deletePageAndAllReference 函数只与完全删除一个 page 有关。错误处理可以忽略掉。有了这样美妙的区隔，代码就更易于理解和修改了。

3.9.2　错误处理就是一件事

函数应该只做一件事。错误处理就是一件事。因此，处理错误的函数不该做其他事。这意味着（如上例所示）如果关键字 try 在某个函数中存在，它就该是这个函数的第一个单词，而且在 catch/finally 代码块后面也不该有其他内容。

3.9.3 Error.java 依赖磁铁

返回错误码通常暗示某处有个类或是枚举，定义了所有错误码。

```
public enum Error {
        OK,
   INVALID,
   NO_SUCH,
   LOCKED,
   OUT_OF_RESOURCES,
   WAITING_FOR_EVENT;
}
```

这样的类就是一块依赖磁铁（dependency magnet）；其他许多类都得导入和使用它。当 Error 枚举修改时，所有这些其他的类都需要重新编译和部署。[33]这对 Error 类造成了负面压力。程序员不愿增加新的错误代码，因为这样他们就得重新构建和部署所有东西。于是他们就复用旧的错误码，而不添加新的。

使用异常替代错误码，新异常就可以从异常类派生出来，无需重新编译或重新部署[34]。

3.10 别重复自己[35]

回头仔细看看代码清单 3-1，你会注意到，有个算法在 SetUp、SuiteSetUp、TearDown 和 SuiteTearDown 中总共被重复了 4 次。识别重复不太容易，因为这 4 次重复与其他代码混在一起，而且也不完全一样。这样的重复还是会导致问题，因为代码因此而臃肿，且当算法改变时需要修改 4 处地方。而且也会增加 4 次放过错误的可能性。

使用代码清单 3-7 中的 include 方法修正了这些重复。再读一遍那段代码，你会注意到，整个模块的可读性因为重复的消除而得到了提升。

重复可能是软件中一切邪恶的根源。许多原则与实践规则都是为控制与消除重复而创建。例如，全部考德（Codd）[36]数据库范式都是为消灭数据重复而服务。再想想看，面向对象编程是如何将代码集中到基类，从而避免了冗余。面向方面编程（Aspect Oriented Programming）、面向组件编程（Component Oriented Programming）多少也都是消除重复的一种策略。看来，自子程序发明以来，软件开发领域的所有创新都是在不断尝试从源代码中消灭重复。

3.11 结构化编程

有些程序员遵循 Edsger Dijkstra 的结构化编程规则[37]。Dijkstra 认为，每个函数、函数中的每个代码块都应该有一个入口、一个出口。遵循这些规则，意味着在每个函数中只该有一个 return 语句，循环中不能有 break 或 continue 语句，而且永永远远不能有任何 goto 语句。

我们赞成结构化编程的目标和规范，但对于小函数，这些规则助益不大。只有在大函数中，这些规则才会有明显的好处。

所以，只要函数保持短小，偶尔出现的 return、break 或 continue 语句没有坏处，甚至还比单入单出原则更具有表达力。另外一方面，goto 只在大函数中才有道理，所以应该尽量避免使用。

3.12 如何写出这样的函数

写代码和写别的东西很像。在写论文或文章时，你先想什么就写什么，然后再打磨它。初稿也许粗陋无序，你就斟酌推敲，直至达到你心目中的样子。

我写函数时，一开始都冗长而复杂。有太多缩进和嵌套循环。有过长的参数列表。名称是随意取的，也会有重复的代码。不过我会配上一套单元测试，覆盖每行丑陋的代码。

然后我打磨这些代码，分解函数、修改名称、消除重复。我缩短和重新安置方法。有时我还拆散类。同时保持测试通过。

最后，遵循本章列出的规则，我组装好这些函数。

我并不从一开始就按照规则写函数。我想没人做得到。

3.13 小结

每个系统都是使用某种领域特定语言搭建，而这种语言是程序员设计来描述那个系统的。函数是语言的动词，类是名词。这并非是退回到那种认为需求文档中的名词和动词就是系统中类和函数的最初设想的可怕的旧观念。其实这是个历史更久的真理。编程艺术是且一直就是语言设计的艺术。

大师级程序员把系统当作故事来讲，而不是当作程序来写。他们使用选定编程语言提供的工具构建一种更为丰富且更具表达力的语言，用来讲那个故事。那种领域特定语言的一个部分，就是描述在系统中发生的各种行为的函数层级。在一种狡猾的递归操作中，这些行为使用它们定义的与领域紧密相关的语言讲述自己那个小故事。

本章所讲述的是有关编写良好函数的机制。如果你遵循这些规则，函数就会短小，有个好名字，而且被很好地归置。不过永远别忘记，真正的目标在于讲述系统的故事，而你编写的函数必须干净利落地拼装到一起，形成一种精确而清晰的语言，帮助你讲故事。

3.14 SetupTeardownIncluder 程序

代码清单 3-7 SetupTeardownIncluder.java

```
package fitnesse.html;
import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;
public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;
  public static String render(PageData pageData) throws Exception {
```

```
    return render(pageData, false);
  }
  public static String render(PageData pageData, boolean isSuite)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }
  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }
  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }
  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }
  private void includeSetupAndTeardownPages() throws Exception {
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updatePageContent();
```

```
}
private void includeSetupPages() throws Exception {
  if (isSuite)
    includeSuiteSetupPage();
  includeSetupPage();
}
private void includeSuiteSetupPage() throws Exception {
  include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
}
private void includeSetupPage() throws Exception {
  include("SetUp", "-setup");
}
private void includePageContent() throws Exception {
  newPageContent.append(pageData.getContent());
}
private void includeTeardownPages() throws Exception {
  includeTeardownPage();
  if (isSuite)
    includeSuiteTeardownPage();
}
private void includeTeardownPage() throws Exception {
  include("TearDown", "-teardown");
}
private void includeSuiteTeardownPage() throws Exception {
  include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
}
```

```
  private void updatePageContent() throws Exception {
    pageData.setContent(newPageContent.toString());
  }
  private void include(String pageName, String arg) throws Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
      String pagePathName = getPathNameForPage(inheritedPage);
      buildIncludeDirective(pagePathName, arg);
    }
  }
  private WikiPage findInheritedPage(String pageName) throws Exception {
    return PageCrawlerImpl.getInheritedPage(pageName, testPage);
  }
  private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
  }
  private void buildIncludeDirective(String pagePathName, String arg) {
    newPageContent
    .append("\n!include ")
    .append(arg)
    .append(" .")
    .append(pagePathName)
    .append("\n");
  }
}
```

3.15 文献

[KP78]：Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw- Hill, 1978.

[PPP02]：Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

[GOF]：*Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[PRAG]：*The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[SP72]：*Structured Programming*, O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.