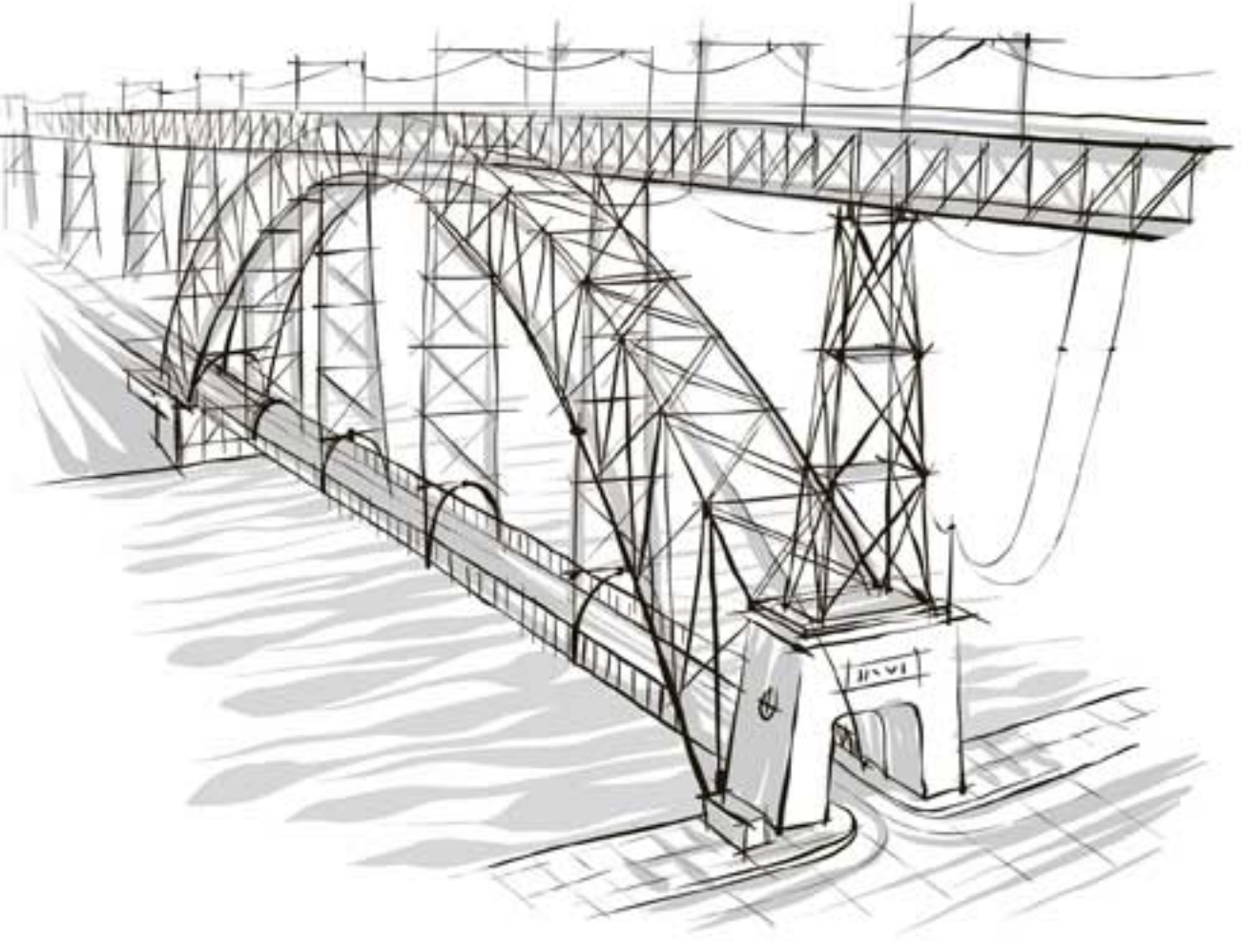
Python 设计模式

[美] 韦塞尔·巴登霍斯特(Wessel Badenhorst) 著
蒲成 译



Apress®

清华大学出版社

Python 设计模式

[美] 韦塞尔·巴登霍斯特(Wessel Badenhorst) 著
蒲 成 译

清华大学出版社
北 京

Wessel Badenhorst
Practical Python Design Patterns：Pythonic Solutions to Common Problems
EISBN：978-1-4842-2679-7

北京市版权局著作权合同登记号　图字：01-2018-4393

图书在版编目(CIP)数据

Python 设计模式 /(美) 韦塞尔·巴登霍斯特(Wessel Badenhorst) 著；蒲成译. —北京：清华大学出版社，2019

书名原文：Practical Python Design Patterns: Pythonic Solutions to Common Problems

ISBN 978-7-302-51645-3

Ⅰ. ①P…　Ⅱ. ①韦… ②蒲…　Ⅲ. ①软件工具－程序设计　Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2018)第 257364 号

责任编辑：王　军　于　平
封面设计：周晓亮
版式设计：思创景点
责任校对：成凤进
责任印制：刘海龙

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市吉祥印务有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张**：17.25　　**字　　数**：348 千字
版　　次：2019 年 1 月第 1 版　　**印　　次**：2019 年 1 月第 1 次印刷
定　　价：98.00 元

产品编号：080061-01

译 者 序

近年来，由于人工智能的兴起，Python 逐渐从科学计算领域进入普通大众的视野，其使用率一直呈线性增长。Python 在设计上坚持了清晰划一的风格，这使得它成为一门易读、易维护，并受到大量用户欢迎、用途广泛的语言。Python 的设计哲学是优雅、明确、简单，追求“用一种方法，最好是只有一种方法来做一件事”。有鉴于此，设计模式与 Python 的结合就变得顺理成章了。

设计模式是经过总结、优化的，可用于解决我们经常会碰到的一些编程问题的可重用解决方案。一个设计模式并不像一个类或一个库那样能够直接作用于我们的代码。设计模式更高级，它是一种必须在特定情形下实现的方法模板。设计模式不会绑定具体的编程语言。一个好的设计模式应该能够用大部分编程语言实现。最重要的是，设计模式也是一把双刃剑，如果设计模式被用在不恰当的情形下将造成灾难，进而带来无穷的麻烦。然而，如果设计模式在正确的时间被用在正确的地方，那么它将是我们的救星。

本书中的每一章都是以一个引用句子作为开头的，如果读者能明白其背景，那么一定会对本书作者的这一小情趣报以会心一笑，谁说钻研技术的人就毫无情趣了？译者对于这些短句子也是详细了解了其来源和背景，并且对作者在相应章节中放入对应句子的意义深以为然。为了让读者享受到这一乐趣，在翻译本书的过程中，译者并没有给出这些句子的详细出处和历史背景，这些内容就留待读者自行探究吧。毕竟，探索的过程，其本身的意义要远远大于平铺直叙的结果所能带来的影响。

本书从设计模式的角度，结合各个示例深入浅出地讲解了各种设计模式在 Python 中的实现，为读者描述了 Python 式设计模式的应有结构以及应该避免的误区。所以本书的高度较一般的 Python 开发类书籍要高，同时也要求读者具备一定的软件开发和架构经验，否则就可能无法很好地理解本书中讲解的内容。相信在阅读完本书之后，读者对于在 Python 中实现各种设计模式的处理一定会深有感悟。正如作者在书中所言，作为程序员，我们应该追求持续不断的进步。学无止境，大家共勉！

在此要特别感谢清华大学出版社的编辑，在本书翻译过程中他们提供了颇有助益的帮助，没有他们的热情付出，本书将难以顺利付梓。

本书主要章节由蒲成翻译，参与翻译的还有何东武、李鹏、李文强、林超、刘洋洋、茆永锋、潘丽臣和王滨。由于译者水平有限，难免会出现一些错误或翻译不准确的地方，如果有读者能够指出并勘正，译者将不胜感激。

译者

作 者 简 介

Wessel Badenhorst 非常热衷于研究获取专家级能力的过程，尤其是编程领域的专家级能力。他获得了计算机科学的学士学位，并且在真实的应用场景中积累了大量经验。

技术编辑简介

作为个人独立贡献者、团队领导、项目经理以及工程副总裁，Michael Thomas在软件开发领域已有超过 20 年的从业经验，他在移动设备方面的开发经验超过了10 年。他目前专注于医疗领域，使用移动设备加速患者与健康医疗服务提供商之间的信息传递。

致　　谢

非常感谢 Apress 的 Mark Powers 及其团队，没有他们，本书将无法面世。谢谢 Tanya、Lente 和 Philip，是你们让我在稍有闲暇时编著了本书。感谢读者积极追求成为更优秀的程序员，从而让这一职业变得更美好。

目　录

第 1 章

前　　述

设计模式有助于我们借鉴其他人成功的经验，而不是靠自己摸爬滚打。

——Mark Johnson

这个世界正在快速变化。

读者在阅读这段文字时，世界各地的人们正在努力学习编程，但大多数人方法笨拙。市场正面临着如潮水般的“程序员”的涌入，似乎没有什么可以阻止这种情况的发生。和其他事物的发展一样，不久大家就会看到，随着程序员的人数逐渐增多，雇用程序员的成本将逐渐减少。能够简单地编写几行代码或者修改一个软件中的某些东西将变成一种基本技能。

如果读者计划以程序员这一职业作为谋生手段的话，互联网会让这个问题变得更加糟糕。我们不仅要担心来自自身所处领域的程序员的竞争，还要面临来自世界各地的程序员的竞争。

想不想知道谁不用担心由数百个编程新兵训练营所培养的数以百万计的程序员？

1.1　大师

尽管每个人都会写字，但世上仍旧会出现海明威似的人物。尽管每个人都能使用 Excel，但仍旧会有财务建模师。同理，当每个人都会写代码时，世上也仍旧会有 Peter Norvigs[①]这样的专家。

程序是一款工具。曾经有一段时间，只要知道如何使用这款工具就能让我们变得有价值，不过那段时期已结束。现在我们面临着新的现实情况，也就是许多

① Peter Norvigs 是人工智能专家，是 *Paradigms of AI Programming* 和 *Artificial Intelligence：A Modern Approach* 的作者/合著者。在本书写作时任 Google 研发部总监。

行业在过去数年中不得不面对的问题。当一款新工具出现时，没人想要使用它。然后，某些人看到其价值，因而也就使得他们要比那些不会使用该工具的人强。之后，这款工具就会变得流行起来。互联网时代已经悄无声息地来临，每个人都能使用它。突然之间，能够创建一个网站变得不那么有价值了，所有从事咨询行业，依靠咨询服务赚取大笔利润的企业都被边缘化了。不过，无论市场如何变化，花时间掌握工具的人都能够抓住发展的机遇。之所以如此，是因为他们能够在每个层面都表现出众——工作质量、开发速度，以及最终成果的完美程度。

目前，我们可以看到对易于使用工具的普遍利用，接下来就会对容易的事情进行自动化处理。在生活的方方面面，简单的任务之所以简单，是因为它们不需要创造力和深刻的理解，而这些特性所带来的结果就是，这些任务正好就是最早转交给计算机处理的任务。

读者捧起这本书的原因就是希望变成一名更优秀的程序员。大家都不希望止步于数以千计这样那样的说明介绍。我们已准备好迈出下一步，帮助读者掌握其精髓。

在编程领域，我们希望能够解决重大且复杂的问题。为此，需要能够在较高的层次上进行思考。就像国际象棋的国际棋联大师们一样，相较于普通的象棋大师而言，他们能够在更大型的组块联合体(chunk)中领悟，我们也需要开发自己在更大组块联合体中解决问题的能力。当能够查看一个全新的问题并且快速将其解构成高层次的组成部分，并且这些组成部分是我们之前就已经解决过了时，我们就成为经验丰富可以举一反三的程序员了。

在我开始编程时，我还读不懂说明手册。那本手册封面上印有《太空侵略者》的图片，并且该手册许诺教会我如何自行开发一个游戏。它有点忽悠人，因为最后编出来的游戏是一个 for 循环和一个 if 语句，而非《太空侵略者》。不过我已经沉迷其中了。学习热情驱使我学习我能接触到的关于编程的一切知识，不过当时并没有很多内容可供我学习。在掌握其基础之后，我有些停滞不前了，即便在以优异成绩获得计算机科学学士学位期间也是如此。我觉得我所学的不过是新瓶装旧酒的基础知识而已。我切实获悉的一点是，似乎每个人都在等待着什么，虽然这一过程让人感觉缓慢和沮丧。

在现实世界中，情形似乎没太多变化。我意识到，如果我希望成为经验丰富可以举一反三的程序员，就必须做一些不同的事情。一开始，我想过获得一个硕士学位或者博士学位，不过最后我决定靠自己的力量进行更深入的 研究。

我重新阅读了计算机技术类书籍的旧理论，它们被赋予了新的含义和新的生命。我开始定期参与编程的工作挑战并且学习用更地道的方式编写代码。然后，我开始研究对于我本人而言完全陌生的编程语言和范式。在我有所意识之前，我

已经将自己对于编程的思维模式完全转变了。曾经很棘手的问题变得微不足道，而一些看似不可能的任务变得仅仅只是有些挑战性而已。

我仍旧在学习，仍旧在深入研究。

读者们也可以这样做。可以从目前所阅读的这本书开始。在本书中，大家会找到若干工具和思维模式，它们有助于大家更好地思考和编码。掌握这一组模式之后，就要积极主动地寻找和掌握不同的工具与技术。研究不同的框架和语言，并且弄明白其伟大的原因。一旦能够理解某些人为何会钟爱一门不同于我们自己经常所用的语言时，可能就会在其中发现几个可以整合到我们思考方式中的理念。无论何时发现诀窍和模式，都要加以学习，并且将其转化成可以反复使用的抽象方案。有时候，我们可以直接分析已经在用的解决方案：是否可以找到更优雅的方式来实现其概念，或者在某些方面是否缺乏该理念？要持续不断地询问自己，如何才能改进我们的工具、提升我们的技能。

我们将要探究一个真实场景上下文中的一组基础设计模式。与此同时，大家将开始意识到，在未来遇到特定类型问题时，这些设计模式可以如何被视作可供使用的构造块。

我希望大家将本书中的设计模式用作方案蓝图，它将有助于大家尽快开始收集自己的问题解决方案组块。在阅读完本书之后，大家应该可以顺利地继续迈向作为程序员的下一个阶段了。大家还应该会看到，这组抽象工具如何被转换成我们可能遇到的任何编程语言，以及如何才能使用来自其他语言的理念和解决方案，并且将之纳入到我们自己的思维工具箱中。

1.2　成为更优秀的程序员

要成为更优秀的程序员，我们就需要下决心去提高自身在编程方面的精通程度。每一天都是让自己变得比前一天更优秀的新机会。每一行代码都是提升改进的机会。那么该如何做呢？

- 当必须处理一段已有代码时，在处理完成之后应该让它比之前变得更好。
- 尝试每天完成一个快速解决问题的挑战。
- 寻求与编程技术比自己好的程序员共事的机会(开源项目是最好的机会)。
- 专注于刻意训练。
- 无论何时可以找到理由，都要练习系统化思考。
- 收集和分析思维模型。
- 熟练掌握工具并且理解它们适用于何处以及不适用于何处。

1.2.1 刻意训练

Anders Erickson 博士研究过那些达到一定能力水平的我们称之为大师的人。他发现，这些人有一些共同点。首先，正如 Malcolm Gladwell 的著作 *Outliers* 中所普及的概念一样，这些人似乎都在其相关领域的技能上花费了大量的时间。实际的数字可能不太相同，但经过调研，这一时长大致接近 10 000 小时。这是很长的一段时间，但是仅花费 10 000 小时，或者说十年时间进行实践还是不够的。经证明，要达到大师级别的能力水平，需要一种非常特定类型的实践。获得能主导相关领域的技能的关键就是刻意训练。

我们马上就要介绍刻意训练与编程的关系，不过首先来看看刻意训练是什么。

刻意训练是缓慢的、时常停顿的、反流程的。如果读者正在练习小提琴，那么刻意训练就势必涉及非常缓慢地演奏曲子，以确保能够完美地演奏每一个音符。如果读者正在刻意地练习网球，那么这可能意味着与教练一起反复地练习击球，以便进行细微的调整，直到能够在该位置重复完美地击球为止。

知识型工作的问题在于，标准的刻意训练方案看起来都让人觉得很陌生。相对于编程所涉及的过程而言，体育项目或者演奏乐器都很简单。要解构设计和开发软件解决方案的过程是相当困难的。弄明白如何训练一种思维方式是很难的。之所以如此，其中一个原因在于，过去的已经过去了。这句话的意思是，作为开发人员，我们几乎很少会被要求编写一个已经写好的软件。找出可以反复训练的“击球机会”是很难的。

就编程而言，我们希望学习解决问题的不同方式。我们希望找到迫使我们应对具有不同约束条件的同类问题的各种挑战。在彻底理解问题出处以及一组可能的解决方案之前，要持续致力于此。

实质上，刻意训练具有以下这些组成部分：

- 每一次训练环节都具有单一关注点。
- 尝试和反馈之间的间距(时长)要尽可能短。
- 要着手处理自身还不能应对的事情。
- 遵循前人的脚步。

1.2.2 单一关注点

在一个具体的训练环节期间，我们仅仅希望专注于一件事情的原因在于，我们想要将所有的注意力都放在期望提升的元素上。我们希望竭尽所能来完美地处理该事情，即便它是一次性事务。然后，我们希望重复这一过程以便得到另一个完美的回应，然后继续重复进行。每一次训练执行都是对技能的单次强化。我们希望增强那些能得出完美结果的模式，而非那些产生不合意输出的模式。

在本书的内容背景中，我希望读者一次仅针对单个设计模式进行学习。我们不仅要切实地寻求理解某个模式是如何发挥作用的，还要理解一开始为何要使用它以及它是如何适用于我们手头的问题的。另外，要思考能够使用这一设计模式所解决的其他问题。接着，我们要尝试使用正在研究的模式来解决其中一些问题。

我们希望为问题以及解决该问题的模式创建一个思维工具箱。理想情况下，我们要培养出一种意识，也就是，这些问题何时适合放入我们已经熟练掌握的工具箱之一来解决，然后就是能够快速且容易地解决该问题。

1.2.3　快速反馈

刻意训练经常被忽略的一个方面就是快速反馈的周期。反馈越快，连接性就越强，我们也就会更容易地从中学习到经验。这也就是为何像市场营销和写作这样的事情如此难以被人熟练掌握的原因。将字母码在一页纸上和从市场上得到反馈之间的时间的确太长了，以致我们完全不能看到练习尝试的效果。对于编程而言，情况就不同了，我们可以编写一段代码，然后运行它以便立即得到反馈。这就使得我们可以正确前行并且最终得到一个可以接受的解决方案。如果更进一步并且为我们的代码编写完善的测试，那么我们甚至会得到更多的反馈，并且相对于必须在每次做出变更时手动测试该过程而言，这一方式能够更加快速地得出解决方案。

帮助我们更加快速地从过程中学习经验的另一个技巧是，预测希望编写的代码块的输出结果。记录下与我们为何期望这一特定输出有关的内容。现在编写这段代码并且针对期望结果进行结果检查。如果结果不匹配，则要尝试解释为何是这样的情况，以及如何用另一段代码来验证我们的解释。然后，对这另一段代码进行测试。我们要持续这一过程，直到达到熟练掌握的水平。

我们将发现自己得到了不同层次的反馈，并且每一个反馈都有其价值。

第一个层次会直接表明解决方案是否可行。接着，我们可能会开始思考像“该解决方案的实现会有多难？”“这个解决方案对于该问题而言是否真的合适？”这样的问题。之后，我们可能会寻求额外的反馈，反馈的形式包括：简单的代码复审，着手处理项目，与思维模式相同的人探讨解决方案。

1.2.4　自我拓展

我们回避的事情是什么？这些事情是就编程而言引发我们轻微不适的一些方面。它可能是从磁盘上读取一个文件，或者连接到一个远程 API。无论所回避的事情是一些图形库还是一种机器学习框架的安装配置，这都没什么两样，我们全都面临着会让我们感觉不舒服的编程领域的某些内容。这些事情通常就是我们最

需要着手应对的。这里有一些让我们进行拓展并且迫使我们面对自身缺陷和盲点的领域。克服这些不适感的唯一方式就是深入探究，以各种方式反复多次使用它，以便能开始获得一些经验体会。我们必须能够非常顺手地使用它，以便不再需要翻阅关于堆栈溢出的文件打开协议。实际上，我们已经编写了一个更好的协议。我们可以信手拈来般轻松使用 GUI 并且从数据库中提取出数据。

对于这一层次的精通程度而言，是没有捷径可走的。唯一的方式就是头悬梁锥刺股般地刻苦练习。而这也是为何成为真正大师的人如此之少的原因所在。到达这一层次，意味着要在并不容易处理、让我们深感挫败的事情上花费大量的时间。正是由于在这些自我否定的事情上花费了如此多的时间，因此就任何领域而言，几乎没有大师还会骄傲自大。

从头至尾地练习本书中的设计模式是找出潜在可发展领域的一种好的方式。只要从单例模式开始学习并且坚持下去即可。

1.2.5 站在巨人的肩膀上

有人在编程领域做出了非常惊人的成就。这些人通常会在开发者大会上进行演讲，有时候也会在网上露面。建议看看这些人都在谈论些什么。要尝试理解他们处理一个新颖问题的方式。在他们展示解决方案时，要一边听一边记。要跟上他们的思路并且弄明白他们这样做的动机。尝试以我们所认为的这些人解决问题的方式来解决一个问题，用这种方式得出的解决方案与用我们自己的方式得出的解决方案有何不同？

真正伟大的开发人员对于编程都充满激情，并且应该不用过多的推动力就能让他们讲解其技能的更多细节。寻找这类开发人员常常出现的用户群，并且要借机与他们进行大量探讨，要持开放态度并且进行持续学习。

选取那些迫使我们使用还未精通的设计模式的私人项目，享受其处理过程。最重要的是，要学会对这一过程充满激情，并且不要沉迷于一些认知上的结果，而是要花时间成为一名更优秀的程序员。

1.2.6 如何入手

以达·芬奇决定以绘画作为其职业时开始练习的相同方式入手。

也就是模仿！

这是正确的。首先要识别出一些有意义的问题，一个已经被解决的问题，然后坦然地效仿该解决方案。但千万不要复制/粘贴。要通过亲自动手输入的方式来复制该解决方案。让我们自己的复制版本能够生效。在此之后，则要完全删除它。接下来，尝试一边回想一边解决该问题，仅在记不清楚时参考一下原始的解决方

案。要持续尝试，直到自己能够在不查看原始解决方案的前提下毫无错误地再造该解决方案。如果正在寻求可以复制和从中学习的问题解决方案，Github 则是一个值得挖掘的金矿。

一旦能够熟练运用原始不变的解决方案，则要尝试在其基础上进行改进。我们需要学习思考所找到的解决方案——是什么让它们如此优秀？我们如何才能让它们更加优雅、更加通用、更加简单？要持续寻求以其中一种方式让代码变得更好的机会。

接下来，我们希望实际检验新的解决方案。找到使用该解决方案的地方。在现实环境中进行实践会迫使我们应对不同的环境约束，这些约束我们绝不会靠自己凭空想象出来。它会迫使我们以从未考虑过的方式来扭曲我们良好干净的解决方案。有时候，我们的代码将被破坏，并且我们将了解到问题最初的解决方式的价值所在。当然也有这样的情况，我们可能会发现，解决方案要比最初的版本更加有效。要问自己，为何一个解决方案会比另一个更加出色，还有就是我们从中学到了哪些与问题和解决方案有关的内容。

要试图用具有不同范式的语言来解决类似的问题，从中汲取经验并且形成自己的解决方案。如果像实际解决问题那样花费心思关注其过程，我们就能应对任何项目。

研究开源项目的好处在于，通常都会有一群合适的人一路提供帮助，并且还有另外一些人会告诉我们代码到底有什么问题。要评估其反馈，取其精华，去其糟粕。

1.2.7　保持路线正确的能力

在研究一个问题时，我们有两个选项：继续尝试现有方式，或者抛弃所有东西并且利用所学知识从头开始。Daniel Kahnamann 在其著作 *Thinking，Fast and Slow* 中阐释了沉没成本谬论。也就是要持续对一项糟糕的投资进行投入，因为已经对其投入太多了。这对于开发人员而言是糟糕的陷阱。让一个两天工作量的项目变成需要几个月才能完成的最快速方法就是试图在糟糕的解决方案上越走越远。如果打算废弃一天、一周甚至一个月的工作并且重新开始，那么通常从情感上讲就像是巨大的损失。

实际上，我们绝不会完全从头开始，有时要成为能够使用极其简洁优雅的解决方案来解决问题的程序员，走一些弯路是必然的。

我们需要彰显思想上的坚韧性去说出“适可而止”这句话，然后从头再来，用所学知识构建一个新的解决方案。

如果解决方案本身就是错误的，那么对其花费的所有精力都是毫无意义的。

越快意识到这一点越好。

这一自我意识会让我们能够知道何时进行更多的尝试以及何时掉头转向。

1.3 系统化思考

每样事物都是一个系统。通过理解该系统的组成要素、这些要素的连接方式，以及它们彼此如何交互，我们就能理解这个系统。软件展露在外的每一块都是这三个核心组成部分的表现：构成该解决方案的要素、一组连接及其之间的交互。

从很基础的层面上讲，我们首先可以问一问自己：“我希望构建的系统，其要素是什么？”将这些答案写下来。然后，记下它们是如何彼此连接的。最后，列出这些要素之间的交互以及哪些连接参与到这些交互中。这样一来，我们就已经在着手设计该系统了。

所有的设计模式都是在应对这三个基础问题：①要素是什么以及它们是如何创建的？②要素之间的连接是什么，或者说其结构看起来是什么样子的？③这些要素如何交互，或者说其行为看起来是什么样子的？

存在其他的方式对设计模式进行归类，不过目前要使用经典的三组划分方式来帮助我们开发系统化思考技能。

1.4 思维模型

思维模型是外部领域的内在表现形式。领域模型越精确，我们思考的效率就越高。纸上得来终觉浅，所以，拥有更精确的思维模型会让我们对于相关领域的看法更为准确，我们的思维模型集越灵活通用，我们所能够解决的问题集就越多样。通读本书，读者将学习到一组思维工具，它们有助于读者解决程序员这一职业生涯中常常会面临的具体编程问题。

看待思维模型的另一种方式是，将它们视为单个思维单元中的概念分组。对于设计模式的学习将帮助我们开发新的思维模型。问题的结构将意味着我们为了解决该问题而需要实现的解决方案类型。因而，完全清楚我们正尝试解决的问题到底是什么这一点是很重要的。

问题定义或者描述越好——也就是越完整，就越能得到与可行的解决方案有关的更多线索。因此，针对该问题要多提出一些无关紧要的疑问，直到得出一个比较清晰且简单的问题描述。

设计模式有助于我们从 A(问题描述)直接跳到 C(解决方案)，而不必经历中间过程 B，因而也就能够避免许多其他的错误开端。

1.5　这项任务的适用工具

要砸掉一堵砖墙，就需要一把榔头，不过并非一把普通的榔头——我们需要一把手柄很长的大而重的榔头。当然，也可以使用将钉子钉入音乐盒的榔头来砸墙，但工欲善其事，必先利其器，否则将事倍功半。

如果使用不趁手的工具，那么任何工作都会变得一团糟并且需要花费比预计更长的时间才能完成。为手头的任务选择合适的工具是一个经验问题。如果完全不知道除了过去所使用的小榔头之外还有其他的榔头存在的话，那么我们就会艰难地寄希望于有人会站出来帮忙并且在几个小时内帮忙砸掉一堵墙。我们甚至可能会寻求专业高效砸墙熟练工的帮助。我正试图阐述的观点就是，如果使用合适的工具，那么相较于那些尝试使用自己手头上的工具来处理工作的人而言，我们就能够完成数倍于他们的工作量。就扩充工具箱、掌握不同的工具这一点而言，是值得花费时间和精力的，因为这样一来，当遇到一个新的问题时，我们会知道选择哪个工具。

要成为大师级程序员，需要持续地将新工具添加到我们的武器库中，这并不仅仅是要求我们熟悉它们，而是要精通并掌握它们。我们已经介绍了如何精通并掌握所选定工具的过程，不过就 Python 这一领域而言，我可以提出一些具体的建议。

Python 生态系统的其中一些优势在于，有各种各样可以随时获取的包。这些包的数量很多，不过情况往往是，对于我们可能会遇到的每一种问题类型而言，都会有一个或两个明确的领路人。这些包都是有价值的工具，并且我们应该每周都花费数小时来研究它们。一旦我们已经掌握了本书中的模式，则要利用 Numpy 或 Scipy 并且掌握它们。然后，顺着我们自己的认知方向前行即可。下载我们感兴趣的包，学习其基础，然后开始使用已经涉及的框架来尝试应用它。它们的亮点在何处，以及有什么遗漏的没有？它们尤其擅长解决哪类问题？未来可以如何使用它们？哪个业余项目可以让我们能够尝试在真实场景中使用相关的包？

1.6　设计模式的概念

关于设计模式的大名鼎鼎的四人组(Gang of Four)的著作《设计模式：可复用面向对象软件的基础》似乎是一切的源头。他们提出了一种用于描述设计模式(专门针对 C++)的框架，不过描述专注于一般通用的解决方案，因而其中许多模式都被转译成若干语言。该著作中提出的 23 种设计模式的目标就是，将最佳实践的解决方案整理成面向对象编程中时常会遇到的问题。正因如此，这些解决方案都专注于类和类的方法与属性。

这些设计模式中的每一个都代表了一个完整的解决方案理念，并且它们能够将会变更的内容与不会变更的内容分离。

对于最初的设计模式来说，有许多至关重要的人物。其中一个关键人物，Peter Norvig，展示了如何通过 Lisp 中的语言结构来替换其中 16 种模式。这些替换内容中的许多也都可以在 Python 中实现。

在本书中，我们打算介绍几种最初的设计模式以及它们如何适用于真实场景的项目。我们还会考量 Python 语言环境中关于它们的争论，有时候要丢弃用于具有该语言老一套标准的解决方案的模式，有时候要修改 Gof(四人组)的解决方案以便有效利用 Python 的功能和丰富表达性，而其他时候则要直接以 Python 化方式来实现最初的解决方案。

1.6.1 是什么造就了设计模式

设计模式可以从很多方面来阐释，不过它们全都包含以下要素(在此对 Peter Norvig 表示赞扬和感谢，以下内容引用自 http://norvig.com/design-patterns/ppframe. htm)：

- 模式名称
- 意图/目的
- 别名
- 动机/背景
- 问题
- 解决方案
- 结构
- 参与方
- 协作
- 影响/约束
- 实现
- 示例代码
- 已知用途
- 相关模式

在附录中，大家可以找到本书中所探讨的所有设计模式，其编排结构正是根据这些要素来组织的。出于可阅读性和学习过程的考量，与设计模式有关的章节内容，其本身将不会全部遵循这一结构。

1.6.2 分类

设计模式被分类为不同的组，以便帮助我们像程序员那样彼此探讨解决方案的类别，并且在讨论这些解决方案时为我们提供一种共同语言。这使得我们可以

清楚地沟通并且在围绕主题进行探讨时能够准确地进行表述。

我们将根据创建性、结构化以及行为性模式这一最初分组来对设计模式进行分类。这样做不仅仅是为了坚持完成任务的通用方式，也是为了帮助读者在发现这些模式的系统上下文中研读这些模式。

1. 创建性

第一个类别处理的是系统中的要素——具体而言，也就是它们是如何被创建的。我们在进行面向对象的编程时，对象的创建是通过类的实例化来实现的。稍后大家将会看到，在解决具体问题时，有一些不同的特征是值得拥有的，并且，创建一个对象的方式对这些特征具有重大影响。

2. 结构化

结构化模式处理的是类和对象的构成方式。可以使用继承构成对象以便获得新的功能。

3. 行为性

这些设计模式专注于对象之间的交互。

1.7　将用到的工具

目前我们正在转换到 Python 3。这一转换是缓慢且经过深思熟虑的，并且这一过程就像是移动的冰川一样，是不可阻拦的。出于本书的目的，我们将放开 Python 2 并且拥抱未来。不过尽管如此，大家还是应该可以用 Python 2 运行大部分代码而不会碰到太多的麻烦(这并非是由于本书中的代码具有太多优良的特性，更多的原因在于 Python 核心开发人员所完成的出色工作)。

使用 Python，具体而言就是 CPython(默认的 Python 解释器)，我们就会用到 Pip，也就是 Python 的包安装器。Pip 集成了 PyPI，也就是 Python Package Index，并且让我们可以从包索引处下载并安装一个包，而不必手动下载这个包，解压它，然后再运行 python setup.py 进行安装。Pip 使得为环境安装库变成一件很顺畅的事情。Pip 也会为我们处理所有的依赖，因此我们不需要四处寻找还没有安装好的所需的包。

在后面开始处理第三个项目时，我们将需要大量的包。所有这些包并非都需要用于我们所有的项目。我们希望维持每一个项目的包随时可用的状态。进入 VirtualEnv(这是一个虚拟的 Python 解释器)，它将为该解释器所安装的包与系统上的其他环境隔离开。我们需要将每一个项目保持在其自己的最简化空间中，仅仅安装它所需要的包以便让其可以运行，而不会干扰我们可能正在着手处理的其他项目。

1.8 本书的阅读方式

阅读一本编程书籍的方式有许多种。大部分人一开始翻开一本编程书都希望逐页阅读，而最后却变成直接从一个代码示例跳到另一个代码示例。需要牢记的一点是，阅读本书的方式各异，但是在花时间阅读之后，一定要达到最好的效果。

第一类读者仅仅希望获得 GoF 设计模式的 Python 式版本的快速、可靠参考。如果是这样的话，则可以直接翻到附录查看每一种设计模式的正式定义。在时间允许的情况下，大家可以翻回相关的章节，并且阅读一下感兴趣的设计模式的探究。

第二类读者希望能够找出特定问题的特定解决方案。对于这些读者而言，每一个设计模式的章节，其开篇都具有该模式所处理的相关问题。这应该能帮助读者判断该模式是否有助于解决所面临的问题。

最后一类读者希望使用本书提升技能。对于这些读者，建议从本书开头开始阅读并且按照本书内容循序渐进地动手编码，要手动敲写每一个示例，修补每一个解决方案，完成所有的练习。看看修改完代码之后会发生些什么。有哪些缺陷，以及为什么会有这些缺陷？尽力让解决方案变得更好。一次仅应对一种模式并且要完全掌握它。然后，找到可以应用新知识的其他真实场景。

1.9 配置 Python 环境

首先使计算机上可以运行一个正常的 Python 3 环境。在这一节中，我们将介绍在 Linux、Mac 和 Windows 上安装 Python 3。

1.9.1 Linux 环境

使用 apt 包管理器的命令在 Ubuntu 上配置环境。对于不适用 apt 的其他 Linux 发行版，可以查看其安装 Python 3 和 pip 的过程，使用类似 yum 的另一种包管理器，或者从源上安装相关的包。

整个安装过程将利用终端，因此现在可以打开它了。

首先检查系统上是否已经安装了 Python 的一个版本。

只是要注意：在阅读本书期间，将使用前缀$表明终端命令，在终端中输入下面这样的注解时不要输入$或者其后的空格。

```
$ python --version
```

如果已经安装了 Python 3(例如 Ubuntu 16.04 及其以上版本)，则可以跳过这段内容，直接阅读安装 pip 的章节。

如果系统上安装了 Python 2 或者没有安装 Python，则可以继续阅读并且使用以下命令安装 Python 3：

```
$ sudo apt-get install python3-dev
```

这个命令将会安装 Python 3。可以检查所安装的 Python 版本，以便验证安装的是正确的版本：

```
$ python3 --version
```

现在，系统上的 Python 3 应该可用了。

接下来，安装构建必备项以及 Python pip：

```
$ sudo apt-get install python-pip build-essential
```

现在，检查 pip 是否可有效运行：

```
$ pip --version
```

大家应该看到系统上安装好了 pip 的一个版本，现在准备好安装 virtualenv 包。请跳过 Mac 和 Windows 环境的安装介绍。

1.9.2　Mac 环境

默认情况下，macOS 已经安装了 Python 2 的一个版本，不过我们不需要它。

要安装 Python 3，就要使用 Homebrew，它是 macOS 的一个命令行包管理器。为此，需要 Xcode，可以从 Mac 的 AppStore 上免费获得。

安装好 Xcode 之后，打开 Terminal 应用并且安装 Xcode 的命令行工具：

```
$ xcode-select --install
```

直接根据所打开窗口中的命令提示符安装 Xcode 的命令行工具即可。完成之后，就可以安装 Homebrew 了：

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
master/install)"
```

如果还没有安装 XQuartz，那么 macOS 可能会遇到一些错误。如果遇到这样的情况，则可以在网站 https://www.xquartz.org/上下载 XQuartz.dmg。然后，检查是否已成功安装 Homebrew 以及它是否能正常运行：

```
$ brew doctor
```

要在 Terminal 中从任意位置运行 brew 命令，需要将 Homebrew 的路径添加到 PATH 环境变量。打开或者创建~/.bash_profile 并且在文件结尾处添加以下内容：

```
export PATH=/usr/local/bin:$PATH
```

关闭并且重新打开 Terminal。在重新打开的过程中，新的 PATH 变量将会被添加到环境中，现在就可以从任意位置调用 brew 了。使用 brew 找到 Python 的可用包：

```
brew search python
```

现在会看到所有与 Python 相关的包，python3 是其中之一。接着，使用以下 brew 命令安装 Python 3：

```
$ brew install python3
```

最后，可以检查 Python 3 是否已安装并且正常运行：

```
python3 --version
```

在使用 Homebrew 安装 Python 时，也会安装对应的包管理器(pip)、Setuptools 以及 pyvenv(这是 virtualenv 在 Mac 上的替代项，不过本书仅需要使用 pip)。

检查 pip 是否正常运行：

```
$ pip --version
```

如果查看版本信息，则会发现 pip 在系统上已经成功安装了，大家可以跳过 Windows 环境安装的内容，直接阅读使用 pip 安装 VirtualEnv 的章节。

1.9.3 Windows 环境

首先下载 Python 3 Windows 安装程序。可以在这里下载该安装程序：https://www.python.org/。

下载完成后，运行该安装程序并且选择 Customize 选项。确保选中安装 pip。另外，要选中将 Python 添加到环境变量的选项。

在此安装过程中，还会安装 IDLE，它是一个交互式的 Python shell 程序。这会让我们可以在实时解释器中运行 Python 命令，它对于验证想法或者练习使用新的包是很好的工具。

完成此安装之后，可以打开命令行界面：

```
windowsbutton-r
cmd
```

这样就会打开一个类似于 Mac 和 Linux 上使用的终端。

现在，检查 Python 是否正常运行：

```
$ python --version
```

如果查看版本信息，则会显示 Python 已安装并且链接到了它应该在的路径。

在继续处理之前，检查一下 pip 是否正常运行：

```
$ pip --version
```

1.9.4　VirtualEnv

在开始安装 VirtualEnv 之前，先要确保已经安装了最新版本的 pip：

```
$ pip install --upgrade pip
```

Windows 用户将看到如下所示的命令提示符：

```
c:\some\directory>
```

在 Mac 上是这样：

```
#
```

在 Linux 上是这样：

```
$
```

在本书内容中将使用$表示终端命令。

使用 pip 安装 virtualenv 包：

```
$ pip install virtualenv
```

现在已经安装了 virtualenv 包，我们要使用它创建一个本书内容使用的环境。

在我们的终端中，打开将要在其中展开工作的目录。接下来，创建一个虚拟环境，我们要在其中安装本书项目所需的包。我们的虚拟环境将被称为 ppdpenv；env 这个后缀只是为了让我们知道它是一个虚拟环境。无论何时启动这个环境并且使用 pip 命令安装一个新的包，这个新的包都将仅安装在这个环境中，并且将仅在这个环境启动时才可用于我们的程序。

```
$ virtualenv -p python3 ppdpenv
```

在 Windows 上，需要一个稍微不同的命令：

```
$ virtualenv -p python ppdpenv
```

如果 Python 3 并非 PATH 的一部分，则使用希望在 virtualenv 内部使用的 Python 可执行程序的完整路径。

安装过程执行完成后，运行 virtualenv 命令时所在的目录中将出现一个名称为 ppdpenv 的新目录。

要启动该虚拟环境，在 Mac 和 Linux 上需要运行以下命令：

```
$ source ./ppdpenv/bin/activate
```

在 Windows 上则是运行这个命令：

```
$ source ppdpenv\Scripts\activate
```

最后，检查 Python 的版本以确保安装的是正确版本。

好了——到这里环境就全部安装好了，可以开始用 Python 编程了！

要退出该虚拟环境，可以运行以下命令：

```
$ deactivate
```

1.10 编辑器

任何可以保存纯文本文件的文本编辑器都可用于编写 Python 程序。我们可以直接使用记事本来编写下一个独角兽级别的应用，不过这一过程估计会非常痛苦。无论何时编写一个任意大小的程序，我们都想要一个好的编辑器，它需要能够高亮显示我们的代码，使用不同的颜色标记出不同的关键字和表达式，这样我们才能轻易地区分它们。

下面的内容是我偏爱的一些文本编辑器的简要清单，并附有简要的介绍。

1.10.1 Atom

Github 开发了一个编辑器，将之称为 Atom，它是一个简单、优美且功能齐全的可开箱即用的编辑器。它提供了简单的 git 集成以及一个好用的包管理系统(其中包括一个实时的 markdown 编译器)。实际上它是基于 Electron 的，这意味着可以跨平台使用该编辑器。如果平台可以运行 Chromium，那么它就可以运行 Atom。

可以通过网站 https://atom.io/获取它。

1.10.2 LightTable

这个编辑器基于 ClojureScript(一个编译成 JavaScript 的 Lisp 方言)，它同时具有 Vim 和 Emacs 模式，这样我们就能不再依赖鼠标了。就像大多数现代代码编辑器，LightTable 也提供了直接的 git 集成。它是开源的并且易于自定义。

可以在此获取它：http://lighttable.com/。

1.10.3 PyCharm

在谈论 Python 编辑器时，公认的是 Jetbrains 制定了行业标准。它具有很棒的代码补全、Python linter、未使用的导入警告，此外还有更多特性，不过对于我来

说，最关键的特性就是代码拼写检查，该特性同时将 Camel 和 Snake 拼写方式纳入了考量。

PyCharm 的企业版并不便宜，不过他们提供了免费的学生版和社区许可。

可以在此获取它：https://www.jetbrains.com/pycharm/download。

1.10.4　Vim

大多数基于 UNIX 的操作系统都已经安装了 Vim。Vim 有时候被戏称为无法逃脱的编辑器(小提示：使用“:q!”可以退出该编辑器)。Vim 最强大之处就在于，其快捷键涵盖了从选择区段内容到跳转到代码中指定行的一切功能——而这一切都可以使用键盘来实现。多年来，Vim 累积了大量的扩展、颜色主题，以及一个成熟 IDE 所期望拥有的一切特性。它具有自己的包管理器，并且让我们可以完全掌控代码编辑的方方面面。它是免费的，并且在我印象中是最快从所有编辑器中脱颖而出的。就其所有优点而言，Vim 唯一的麻烦之处在于其学习难度很大——非常大。我会在希望编写一个快速脚本或者进行小变更的任何时候都使用 Vim，并且不希望等待其他一些编辑器/IDE 的启动过程。

可以访问这里来设置用于 Python 的 Vim：https://realpython.com/blog/python/vim-and-python-a-match-made-in-heaven/。

1.10.5　Emacs

Emacs 是基于 Emacs Lisp 来构建的，并且可以随心所欲地对其进行自定义。我们可以用它做任何事情，从发送电子邮件到让我们的咖啡机启动，而这只要使用一个简单的快捷键组合即可。它就像 Vim 一样，可以完全不用鼠标，不过它具有更为陡峭的学习曲线。它具有所有的代码自动补全和分屏模式选项，这是一个现代 IDE 所提供的能力，以便在我们认为需要时调整该系统。我们不必等待其他供应商来创建一个包以处理一些新的语言特性，我们可以轻易地自行完成。轻易的意思就是可以非常随意地使用。使用 Emacs，就可以让编辑器以我们认为所应有的方式来处理任务，而大部分的编程工作目前的完成方式都不太令人满意，反而需要我们自己承担重担以便以不同的方式来完成这些工作。

可以访问这里来为 Python 开发设置 Emacs：https://realpython.com/blog/python/emacs-the-best-python-editor/。

1.10.6　Sublime Text

这是值得提及而非真实建议的一个可选编辑器。Sublime 曾经是可用的最佳免费选项(每隔几次保存就会跳出让人感觉唠叨的购买屏幕)。它拥有好看的默认颜

色体系，启动非常快，并且是可扩展的。它默认支持 Python，不过属于它的时代已经过去了。

如果对其感兴趣，则可以在这里获取它：https://www.sublimetext.com/3。

1.11 本章小结

最后要说明的是，无论选择使用何种编辑器都没有关系。它必须是我们自己用着顺手的。无论我们做何选择，都要花时间掌握这个工具。如果希望工作高效，则必须精通我们每天都得使用的工具或过程，无论它看上去可能会是多么平淡。我所能给出的建议就是，选取其中一款工具并且完全掌握它，如果所选择的是 Vim 或 Emacs，那么要学会足够的 Vim 和 Emacs 知识以便在其中编辑文本。

现在，我们已经设置好了环境并且让其与系统的其他部分隔离开来了，我们也安装好了所选择的编辑器并且准备好继续实践了。让我们开始研究一些实际的 Python 设计模式吧。