Python 打包与发布

[美] 戴恩·希拉德(Dane Hillard) 著
郭 涛 译

清華大學出版社
北 京

北京市版权局著作权合同登记号　图字：01-2024-0878

Dane Hillard
Publishing Python Packages
EISBN: 978-1-61729-991-9

图书在版编目(CIP)数据

Python 打包与发布 / (美) 戴恩·希拉德(Dane Hillard) 著 ; 郭涛译.
北京 : 清华大学出版社, 2025. 6. -- ISBN 978-7-302-69400-7
Ⅰ. TP312.8
中国国家版本馆 CIP 数据核字第 2025QC9129 号

责任编辑：王　军
封面设计：高娟妮
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：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-8347000　　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：北京瑞禾彩色印刷有限公司
经　　销：全国新华书店
开　　本：148mm×210mm　　**印　　张**：9　　**字　　数**：251 千字
版　　次：2025 年 7 月第 1 版　　**印　　次**：2025 年 7 月第 1 次印刷
定　　价：69.80 元

产品编号：100098-01

译 者 序

“Talk is cheap, show me the code”(空谈无益，代码为证)。程序员始终追求优雅的代码和规范的注释。但现实往往令人失望，通常是好不容易在 GitHub 找到了源码，却发现没有注释、说明文档和接口文档，这让很多人望而却步。在安装他人发布的软件包时，也时常会遇到莫名其妙的报错，费尽周折也找不到解决之道，在项目 GitHub 主页上也找不到合适的解决方案。这是很多从业人员不得不面对的现实问题。

那么，在发布自己的软件包以及对应的技术文档时，是否有工具可用？文档撰写是否有一定的标准呢？严格来说，不同语言的软件包发布通常涉及多个工具，技术文档也缺乏强制标准。然而，行业内已形成共识，程序员应遵守一些基本的规范和约定。

本书是一本关于 Python 打包与发布的工具书，旨在手把手指导读者完成整个软件包的创建过程，包括维护测试套件、自动化代码质量、持续集成、编写与维护文档、构建社区和更新代码。一眼望去，整个流程漫长而又烦琐，你甚至还会在读完全书后感慨发布工具之多。的确，发布自己的软件包并构建技术社区，是一件任重道远但又极具价值的事。

要做好这件事，首先，要对技术抱有极大的热情，具有奉献精神；其次，要在自己发布软件包的相关领域持续跟进和集成；最后，甚至要成为一个兜售者，热情地为技术会议上的每个同行介绍自己的软件包。有些人甚至会牺牲陪伴家人的时间和休息时间来全身心投入这项工作。这也许就是极客(黑客)精神的写照，一群热爱技术的人为了自己的行业孜孜不倦、持续努力。其实，计算机领域不乏

这样的人，如自由软件之父Richard Stallman、Linux之父Linus Torvalds、算法大师Donald E. Knuth，以及C++发明者Bjarne Stroustrup。

发布自己的软件包，就像孕育自己的孩子一样，从无到有，由小到大。从第一行代码到一个庞大的社区，需要付出满腔热忱。当然，有人可能会说，既然这么累，为什么还要自己发布软件包，用他人发布的不好吗？或者参与开源项目不好吗？当然可以，参与开源项目、开源活动是一件很棒的事。如果你能够把自己在某一领域的成果以一个软件包的形式发布出来供大家使用，那更是一件令人骄傲的事。这并不是鼓吹你发布自己的软件包。但是如果你做到了，那一定既酷又炫！

无论是学术界科研人员还是产业界工程师，当技术积累到一定程度时，都会有发布自己软件包的需求，也会有与志同道合的人共同建设社区的需求。从学术角度来看，许多研究往往止步于理论构建、实验设计、编码实现与发表高水平的学术论文。很多人在论文发表后，便会转而去探索新的领域，导致论文永远沉睡于海量文献库里，很多同行无法将其中绝妙的研究复现到实际应用中，只能日复一日地熬夜加班重复攻克同样的问题。如果能够将此类成果梳理成软件包发布出来，并配有完整的技术文档，那么很多看到论文的读者便可以基于技术文档对其进行复现，同时工程师也可以基于论文和软件包对其进行工程化、产品化，最终落地产生商业价值。如果工程师能解决自己遇到的工程难题和各种问题或者以一个性能更佳、效率更高的软件包发布出来，那么对于技术领域来说便是一件值得庆幸的事。

本书内容主要包括 4 部分。第Ⅰ部分(第 1~3 章)介绍常见的打包工具，并对构建的文件进行详细的剖析；第Ⅱ部分(第4~6 章)讲解如何创建并测试可行的软件包；第Ⅲ部分(第 7~9 章)讲述了 GitHub 持续发布，以及技术文档的编写与维护；第Ⅳ部分(第 10~11 章)介绍 cookiecutter 模板的使用和社区的构建。本书适合计算机科学家、企业工程师和对发布自己的 Python 软件包感兴趣的人员阅读。

在翻译本书的过程中，我得到了李静老师的帮助，非常感谢她对本书进行细致审校。此外，我还要感谢清华大学出版社的编辑、校对和排版人员，感谢他们为了保证本书质量所付出的努力。

由于本书涉及内容广泛、深刻，加上译者翻译水平有限，书中难免存有不足之处，恳请各位读者不吝指正。

推 荐 序

每个人的 Python 之旅都始于不同的起点。无论始于何处，我们都注定要逆流而上，体验别样的风景和聆听不同的声音。随着学习的深入，如同河道变窄，树木愈发茂密，你甚至会开始考虑如何向他人交付 Python 应用程序。在河流的源头，你会在一个隐蔽洞穴的入口处发现一些破碎的项目外壳。在它们坠入深渊之前，你会突然听到一个洪钟般的声音质问道：“你最喜欢的打包工具是什么？”。

Python 提供了适合各类开发者的工具，可以在 Python 软件包索引(简称 PyPI，可访问链接 https://pypi.org)上找到。PyPI 在带来便捷的同时也引入了很多新挑战，因为你会开始关心安装、依赖项、环境以及与现实世界中使用软件相关的其他重要问题。这是创建和维护新开源软件包的核心，也是本书重点介绍和讨论的内容。

根据我的经验，大多数 Python 书籍都很少提及软件包的打包。很多书籍都乐于从软件组织的角度讨论模块和软件包，甚至可能会附赠一个基本软件包的简单框架。然而，制作生产级可下载软件包的实际过程往往被“留作练习”。因此，本书应运而生，旨在解决这一特定的问题。

尽管大多数 Python 用户可能并不倾向于发布公开软件包，但仍然经常需要向他人提供 Python 代码，而这样做往往会面临相当大的挑战。这在很大程度上是生态系统复杂性导致的。Python 软件包通常不仅仅涉及 Python 语言本身，还可能混合了 Python、C、C++、Fortran 和 Rust 等多种编程语言。软件包需要在任何类型的操作环境(如 Linux、Mac、Windows 或更新的 Web Assembly)中运行。此外，还需要考虑处理多个 Python 版本的兼容性问题、软件包的依赖性问

题以及软件包管理工具不断变化的问题，这使得问题变得更加复杂。

在动手写这篇推荐序之前，我问了自己一个简单的问题——我能用这本书中的知识来更新自己的一些项目吗？我接触 Python 已 25 年多。在此期间，我只发布并维护了几个小软件包，而且一直都是兼职进行。老实说，我自己通常也在回避打包。这意味着，我最初是带着怀疑和保守的态度翻开此书的，对于软件包的“最佳实践”几乎毫无认知。

但令人欣喜的是，我从这本书中学到了很多。首先，这本书以完全现代的方式介绍了当前的打包工具。其次，它提供了大量与软件包开发有关的背景资料和问题，然后介绍了解决这些问题的实用方法。此外，我还学到了一些与我已经在使用的工具(如 pytest、coverage 等)相关的新技巧。最后，你甚至还能找到关于创建和管理项目社区的建议。

综上所述，这并不一定是一本“简单”的书。即使采用了现代化的处理方式，打包仍然没有放之四海而皆准的方法。你可能会发现自己可以尝试的不同方法，并以自己的方式阅读这本书。我认为，关键是要保持开放的心态，把这本书当作一本实用指南，切实地明白一切皆有可能。这样做，便会发现本书是一个有用的灵感来源。

——*Python Distilled* 作者，David Beazley(https://www.dabeaz.com)

自　　序

2014 年秋天，我开始在 ITHAKA 工作。那时，团队一直在不懈努力，力求摆脱一个专有内容管理系统的束缚，该系统的发布周期长达数月，因此很难进行项目的快速迭代更改。不过，这一努力获得了丰厚的回报，我们成功推出了一个新的交付平台，它与我们所追求的敏捷性能够完美契合。

前端团队选择使用 Django Web 框架，用于 JSTOR 平台(可访问链接 https://www.jstor.org)上面向用户的全新开发，我的加入也与这一选择有关。该团队早期通过开发可独立安装的 Python 软件包支持该项目，事实证明这很有帮助，因为这些包在实现产品多样化的同时还保留了核心共享功能。围绕内容和访问模型的业务领域非常复杂，而这些软件包为它们设定了有用的界限，并赋予了可重用性。尽管这一概念的方向很明确，但仍然存在一些不足之处。

当时，我们没有运行私有软件包仓库，因此所有软件包都必须从我们的版本控制系统中安装。我们没有进行语义化版本管理，仅依赖 Git 的提交哈希值也无法深入了解或管理不同版本之间的变化。我们没有维护更新日志，而是依靠提交消息来管理更改。

在随后的几年里，我们在 Django 和 Python 上的投入逐渐累积到数千行代码，为 JSTOR 的大部分流量提供服务。我们的软件包越来越大，数量也越来越多，打包过程中的摩擦也日益显现。将软件包与应用程序代码混杂在一起，虽然带来了立即反映更改的便利，但导致版本选择能力退化，逐渐形成了“哪里最方便，代码就放在哪里”的惯性模式。

当核心基础架构小组开始为私有打包提供一流的支持时，这一范式发生了转变。看到这一新功能，并对我们的组织结构进行审视

后，我开始思考如何利用持续集成和标准化来保持敏捷性，同时提高质量，并重新获得版本选择的能力。

apiron 项目(可访问链接 https://github.com/ithaka/apiron)是采用新打包方法的首次尝试，现已成为 ITHAKA 第一个积极开发的供第三方使用的开源项目。随着打包工作流程的优势日益明显，我们广泛采用了这一流程。如今，ITHAKA 的前端团队维护着 20 多个 Python 软件包，为同样数量的应用程序提供支持。

ITHAKA 的使命是改善知识的获取方式，我希望本书能为此贡献一份力量，在某些方面帮你拓宽眼界，在另一些方面帮助你实现梦想。虽然这本书注重实用性，但我更希望你能从中获得一些理论和哲学工具，帮助你和你的团队在自动化和可重复流程方面大幅提升工作效率。期待你的不同观点——这是一个向创作者持续迭代的集体知识提供反馈的机会。我希望你能从中受益，并提出不同意见和批评。

如有任何问题、成功故事或争议，请随时通过 pubpypack@danehillard.com 联系我。

作 者 简 介

Dane Hillard 现任 ITHAKA(一家致力于高等教育领域的非营利组织)的技术架构师。他经验丰富，曾为支持数百万用户的 JSTOR 研究平台构建应用架构。目前，他还对安全、松耦合系统和形式化方法感兴趣。

译 者 简 介

郭涛，主要从事人工智能、智能计算、概率与统计学、现代软件工程等前沿交叉领域的研究工作。已出版多部译作，包括《深度强化学习图解》《机器学习图解》和《概率图模型原理与应用(第 2 版)》。

致　　谢

本书的完成过程充满了挑战，如果没有 Python 软件基金会和 Python 打包管理机构工作人员的全力支持，这本书是万万不可能完成的。感谢你们的努力，让 Python 打包领域的发展势头持续向上。我之所以能在这里汇聚知识和流程，全是因为我站在了巨人的肩膀之上。

我写第一本书时，有幸在一家高端家居用品店的后台拥有一间舒适的咖啡店作为写作场所。而本书则完全是在家中完成——大部分时候，我都是坐在搭档Stefanie 的桌子对面奋笔疾书。感谢你的冷静、善良以及各种诙谐打诨。如果我们能在我的拖延症和抱怨声中熬过疫情带来的幽居时光，也许有一天我们真的能一起征服全世界。

感谢 ITHAKA 团队对学习、改进和创新的持续热情。你们精益求精、臻于至善的追求驱使我不断向前。

本书差点就夭折了。感谢我的内容编辑 Toni Arritola 和策划编辑 Mike Stephens，是你们让我有了第二次尝试的机会。你们的鼓励和反馈确保了本书的顺利出版。

我要感谢技术编辑 Al Krinker，感谢你不断追问我的写作初衷。这无疑提高了作品的影响力和清晰度。

感谢 Marjan Bace 和 Manning 团队的其他成员，是你们让这本书焕发生机，并将它送到有需要的人手中。

非常感谢那些在本书出版初期就投入精力并提供反馈意见的勇士们。你们为我指明了方向，让我少走了许多弯路。

致所有为本书付出辛勤努力的审稿人：Aleksei Agarkov、Cage Slagel、Clifford Thurber、Daniel Holth、David Cabrero Souto、David

Cronkite、Delena Malan、Edgar Hassler、Emanuele Piccinelli、Eric Chiang、Ganesh Swaminathan、Håvard Wall、Howard Bandy、Jim Amrhein、Johnny Hopkins、Jose Apablaza、Joshua A. McAdams、Katia Patkin、Kevin Etienne、Kimberly L. Winston-Jackson、Larry Cai、Laxman Singh Tomar、Marc-Anthony Taylor、Mathijs Affourtit、Matthias Busch、Mike Baran、Miki Tebeka、Richard J. Tobias、Richard Meinsen、Robert Vanderwall、Salil Athalye、Sriram Macharla、Vasudevan Surendran、Vidhya Vinay、Vraj Mohan 和 Zoheb Ainapore，你们的建议让本书日臻完善。

最后，我要感谢对本书产生过直接、间接或其他积极影响的所有人。我无法一一列出所有人的名字，如有遗漏，纯属疏忽，还望海涵。感谢 Ee Durbin、Dustin Ingram、Brett Cannon、Paul Ganssle、Filipe Laíns、Bernát Gábor、Łukasz Langa、Sébastien Eustace、Thomas Kluyver、Donald Stufft、Simon Willison、Will McGugan、Dawn Wages、Reuven Lerner、David Beazley、Brett Slatkin、Tzu-Ping Chung、Henry Schreiner、Pradyun Gedam、Paul Moore、Tushar Sadhwani、Sandi Metz、Jason Coombs、Jeff Triplett、Carlton Gibson、Chris Kolosiwsky 和 Peter Ung。

关于封面插图

本书封面上的图摘自 Jacques Grasset de Saint-Sauveur 于 1797 年出版的作品集。每幅插图都是手工精细绘制和上色的。

在那个时代，仅从人们的衣着打扮就很容易辨别出他们的居住地、职业或社会地位。Manning 以几个世纪前丰富多彩的地区文化为基础，精选此类藏品中的精美图片作为图书封面，旨在颂扬计算机行业的创造性和主动性。

关 于 本 书

本书介绍了 Python 打包的几个特定方面，以及几乎适用于所有编程语言的若干核心概念，旨在提高团队和个人在软件交付方面的生产力。无论是 DevOps 团队、产品团队，还是站点可靠性团队，都能从中发现新的实践和工具来完善他们的工作。如果你希望尽可能多地实现 Python 项目生命周期的自动化、标准化和编排，本书将是你的不二之选。

本书读者对象

本书适合已经熟悉 Python 并希望与朋友、团队或全世界分享代码的读者阅读。虽然本书是专门为个人的管理(包括实践)而量身定制的，但其内容几乎可以扩展到任何规模的团队。协作是有效软件开发的关键，因此本书所介绍的实践倾向于消除烦琐枯燥的工作，让你可以专注于通过代码和技术文档进行有效的交流。

随着软件在科学界的不断发展，打包软件的价值也与日俱增。成功的开源项目在诸多里程碑式的事件中屡见不鲜，如登陆火星和黑洞成像。无论你是想追求创新，还是仅仅想确保实验室的负责人(PI)能够验证你用来生成结果的代码，可重复的过程都是至关重要的。

如果你以前没有使用过单元测试和 lint 等软件质量保障工具，本书将从基本概念入手，帮助你构建自动化质量体系，并帮助你将质量检查扩展为丰富的自动化套件。你可以把时间花在思考如何提前发现问题上，而不是疲于救火。

本书结构安排

本书共 11 章，分为 4 个部分。第Ⅰ部分介绍打包任何类型软件的内在价值。第Ⅱ部分引导你构建一个可正常运行的软件包，其中包含了软件包可能需要的各项功能。第Ⅲ部分介绍高度协作项目的自动化和维护需求。第Ⅳ部分展示如何重复这一过程，并扩展用户和贡献者群体。

第Ⅰ部分“基本概念”为 Python 打包奠定了基础，并通过涵盖以下方面，让用户在开始构建自己的软件包时拥有正确的心态。

- 第 1 章介绍了打包的起源，以及它在当今共享软件中仍然有价值的原因。通过这一章的学习可能会拓宽人们对软件包定义的理解，并发现软件包的目标受众非常广泛。
- 第 2 章介绍如何开始使用附录中的工具进行产品打包工作。
- 第 3 章展示 Python 软件包的基本含义，包括所涉及的文件和元数据，以及它们是如何在整个过程中流动的。

第Ⅱ部分“创建可行的软件包”将把最小 Python 软件包扩展成一个具有实际行为的包，你可以在完成本书的学习后进一步扩展它。

- 第 4 章讲解如何将第三方依赖、命令行界面和非 Python 扩展整合到软件包中。
- 第 5 章介绍用于编排单元测试活动的单元测试工具，以确保软件行为的质量。
- 第 6 章在质量的基础上更进一步，纳入了对常见错误、类型安全和代码格式一致性的检查。

第Ⅲ部分“让软件包走进公众视野”推荐了一些可以在任何地方采用的实践，这些实践对于与他人协作尤其有用。

第 7 章展示自动化和持续集成原则的力量，帮助你思考如何为贡献者创建严谨的反馈回路。

- 第 8 章介绍文档的重要性，并展示如何集成一个涵盖代码和

文档说明的自动化文档构建系统。

- 第 9 章介绍了如何以最小的代价定期更新 Python 软件包，从而避免积累技术债务。

第Ⅳ部分“路漫漫其修远兮”回答了在掌握新技能后，下一步该怎么走的问题。

- 第 10 章旨在将在前几章中学到的实践转换为可复用的项目模板，以便在未来的项目中使用。
- 第 11 章旨在分享一些实践方法，帮助你在项目中构建一个由用户、贡献者和维护者组成的社区，此类社区将在前几章所述的清晰流程的基础上蓬勃发展。

我建议从头到尾顺序阅读本书。书中的每一章都是在前一章的基础上逐步展开的，每一步都有每一步的收获。

此外，本书附录讲解了如何安装工具，根据我的经验，这些工具会让打包工作变得更有趣。

- 附录 A 中提及的工具，可以让你更轻松地安装多个版本的 Python(和其他语言)以及调用打包过程中创建的各种 Python 解释器和虚拟环境。
- 附录 B 中提及的工具与项目没有太大关系，但可以提高你在大多数 Python 项目中的工作效率。

关于代码

本书包含许多源代码示例，既有带编号的代码清单，也有普通文本。在这两种情况下，源代码的格式都采用等宽字体，以便与普通文本区分开来。有时，代码也以**粗体**显示，以突出改动的代码，例如，在现有代码行中添加新功能时。

在许多情况下，原始源代码都经过了重新格式化；添加了换行符，并重新调整了缩进，以适应图书排版。在极少数情况下，因为

部分代码过长，还在代码行中添加了换行标签(➥)。此外，在正文描述代码时，往往会删除源代码中的注释。

书中的源代码可在 GitHub 上找到，网址为 http://mng.bz/69A5，也可通过扫描本书封底的二维码下载。

每一章配套的代码都严格地与该章结束时软件包的完整状态保持统一。我非常审慎地做出了以上选择——因为打包配置需要在许多文件中使用精准的语法和数值，它甚至比常规编程更难做到正确无误。为了最大限度地减少挫败感，我认为与按代码清单组织代码相比，这是最好的参考方式。

由于打包实践会随时间发生变化，我将陆续提供本代码配套的更新版本。为避免混淆，我会将这些更新版本与本版的代码区分开来，并在配套的代码资源中提供相应的链接。

目　　录

第 I 部分　基本概念

第 II 部分 创建可行的软件包

第III部分 让软件包走进公众视野

第 I 部分

基本概念

软件打包也许是将应用程序及其功能推向消费市场的最重要成就。软件包支持人们在自己的项目中便捷地重用他人的成果，在手机上安装应用程序，等等。如果没有软件包，工作效率就会大幅下降，人们或许仍处在软件开发的黑暗时代。

无论你是 Python 软件包的维护者，还是刚刚开始使用打包的新手，对打包概念的扎实理解都会让你在阅读本书和完成其他项目时保持正确的思维方式。这部分内容涵盖打包的概念、创建自己的 Python 软件包时需要做什么准备，以及什么是最小可行软件包。

第 1 章

Python打包的含义与目的

本章涵盖如下内容：

- 打包代码，使其更容易访问
- 使用打包使项目更易于管理
- 为不同平台构建 Python 软件包

假设你编写了一个用于自动驾驶汽车的开创性 Python 软件，该最新成果将改变世界，你希望更多的人使用它。你已经说服 CarCorp 公司使用该解决方案，他们希望获取代码并开始使用。

当 CarCorp 公司打电话来询问如何安装和使用代码时，你会详细介绍如何将每个文件复制到正确的目录，如何使某些文件成为可执行文件以便作为命令运行，等等。因为软件是你写的，所以这些对你来说是信手拈来。但令你惊讶的是，电话另一端的开发人员却有点不知所措。这是为何呢？

你已意识到软件制作者和软件使用者之间存在着不可逾越的鸿

沟。如今，人们习惯于在需要新应用时访问 iPhone 上的应用商店。如果想改善软件的用户体验，还有很多工作要做！

本书将讲解如何将 Python 项目作为可安装包发布，从而使他人更容易访问。还将讲解如何创建一个可重复的流程来管理项目，从而减少维护项目所花费的精力，这样软件制作者就可以专注于个人的终极目标：改变世界。在学习过程中，你将使用一些流行的打包工具来构建一个真实的项目，并将该过程的多个方面自动化，从而实现这一切。尽管 Python 社区已经为打包的某些方面制定了标准，但“唯一正确的方法”尚未出现，也可能永远不会出现！

即使你以前创建或发布过 Python 软件包，也能在本书中找到适合自己的内容。你将学到的建议和工具都已经过时间考验，可用于一些标准化程度低的打包实践。Python 打包曾有一段混乱的历史，目前也有许多可供选择的方案，因此除了了解和使用目前可用的工具，你还将学习这些工具背后的原理，以随着环境的变化而不断调整。为此，首先要了解为什么要对软件打包。

1.1 软件打包的准确定义

为了挽回与 CarCorp 公司的关系，你承诺几周后会带着一个经过彻底改进的流程回来，帮助他们快速安装软件。你知道你最喜欢的一些 Python 代码库，如 pandas 和 requests，都可以在网上以软件包的形式获得，你也想为自己的用户提供同样便捷的安装方式。

打包是将软件及其描述文件的元数据一起归档的行为。通常，开发者创建这些归档文件或软件包，目的是共享或发布它们。

重点：Python 生态系统使用 package 一词来表示两个不同的概念。Python 打包管理机构(PyPA)在 *Python Packaging User Guide* (可访问链接 https://packaging.python.org)中将这两个术语区分如下：

- **导入包(import packages)：**将多个 Python 模块组织到一个目录中以便于发现(http://mng.bz/wypg)。

- **发布包(distribution packages)**：将 Python 项目归档发布以供他人安装(http://mng.bz/qoNz)。

导入包并不总是以归档形式发布，尽管发布包通常包含一个或多个导入包。发布包是本书的主题，必要时将与导入包区分开来，以避免混淆。

将软件及其元数据打包在一起的方式可能无限多，那么这些软件的维护者和用户该如何管理预期并减少手动操作呢？这就是软件包管理系统的作用所在。

1.1.1　实现自动化打包的标准化

软件包管理系统或软件包管理器可将特定领域软件包的归档和元数据格式标准化。软件包管理器提供了一系列工具，帮助用户在项目、编程语言、框架或操作系统层面安装依赖项。大多数软件包管理器都附带一套类似的安装、卸载或更新软件包的说明。你可能使用过以下一些软件包管理器：

- pip(https://pip.pypa.io)
- conda(https://docs.conda.io)
- Homebrew(https://brew.sh/)
- npm(https://www.npmjs.com/)
- asdf(https://asdf-vm.com/)

早期的软件包管理

尽管开发人员非正式地进行代码打包已经有一段时间了，但直到 20 世纪 90 年代初软件包管理系统得到广泛应用，这一做法才开始兴起，参见 Jeremy Katz 的 *A Brief History of Package Management*，Tidelift，可访问链接 http://mng.bz/7ZG4。

事实证明，声明式定义项目依赖项的能力显著提升了开发人员的工作效率，因为它抽象出了管理软件项目中的大量基础工作。

软件仓库通过充当发布和托管软件包的集中市场，进一步规范

了软件包的打包流程(见图 1.1)。许多编程语言社区都提供了用于安装软件包的官方或事实上的标准软件仓库。PyPI(https://pypi.org)、RubyGems(https://rubygems.org/)和 Docker Hub(https://hub.docker.com/)就是几个流行的软件仓库。

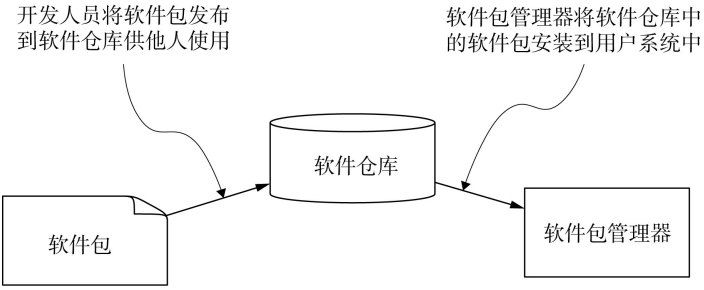


图 1.1 软件包、软件包管理器和软件仓库都是共享软件的关键

如果你有一部智能手机、平板电脑或台式计算机，并从应用商店安装了应用程序，这就是软件包在起作用。软件包是将软件及其元数据捆绑在一起的产物，实质上相当于应用程序。而软件仓库托管着人们可以安装的软件，相当于应用商店。

因此，软件包是将软件和元数据按照约定的格式结合在一起的产物，并在相关的软件包管理系统中进行编码。在更细的层面上，软件包通常还包括在用户系统上构建软件的方法，或者为各种目标系统提供软件的多个预构建版本。

1.1.2 发布软件包的内容

图 1.2 显示了发布软件包中可能包含的一些文件。开发人员通常会在软件包中包含源代码文件，但他们也可以提供编译后的工件、测试数据以及用户或同事可能需要的任何其他内容。通过发布软件包，用户可以一站式获得使用软件所需的所有文件。

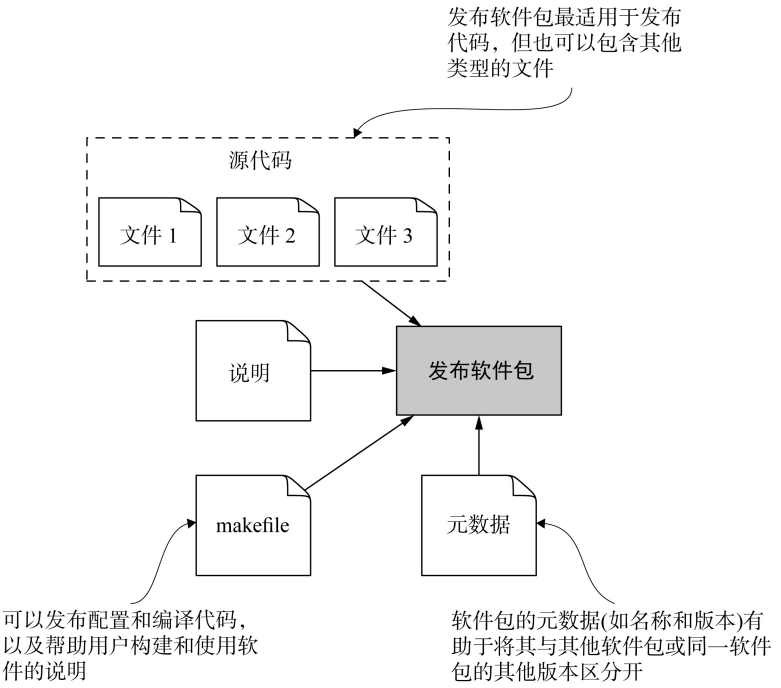


图 1.2　软件包通常包括源代码、用于编译代码的 makefile、关于代码的元数据以及用户说明

软件包中的非代码文件发布是不可或缺的功能。尽管代码通常是发布软件包的核心内容，但许多用户和工具都依赖于代码的元数据来将其与其他代码区分开。开发人员通常会在元数据中指定软件项目的名称、创建者、可重复使用的许可证等。重要的是，元数据通常包括归档的版本，以便与项目以前和未来发布的软件包区分开。

共享软件的早期

在 Unix 操作系统问世后的 10 多年里，团队和个人之间的软件共享仍主要依靠手动操作。下载源代码、编译源代码以及处理编译过程中产生的各种问题，都是由试图使用代码的人自己完成的。这一过程中的每一步都有可能因人为错误和系统之间的架构或环境差

异而导致失败。Make(https://www.gnu.org/software/make/)等工具消除了这一过程中的一些变数，但在软件包版本、依赖项和安装管理方面却止步不前。

现在已经熟悉了软件包中的内容，接下来我们将了解这种共享软件的方法如何解决实际中的具体问题。

1.1.3 共享软件面临的挑战

与 CarCorp 公司的通话越来越紧张，你猛然意识到忘记让他们先安装项目的所有依赖项了。你只好回退几步，先引导他们完成依赖项的安装。遗憾的是，你又忘了检查其中一个主要依赖项的版本，最新版本似乎无法使用。你引导他们安装之前的每个版本，直到最后找到一个能用的版本。最终成功化险为夷。

随着开发的系统越来越复杂，确保正确安装每个依赖项所需版本的工作量也在迅速增加。在最糟糕的情况下，你可能需要同一依赖项的两个不同版本，但它们却无法共存。这种情况被戏称为“依赖项地狱”。在这种情况下，对项目进行脱机处理可能极具挑战性。

即使没有“依赖项地狱”，如果没有标准化的打包方法，也很难以标准方式共享软件，让用户都知道他们需要为项目安装哪些其他依赖项。软件社区创建了管理软件包的惯例和标准，并将这些惯例纳入软件包管理系统中，以便完成工作。

现在，已经讲解了为什么打包有利于共享软件，下面将带你了解打包带来的一些其他优势，即使你并不总是公开自己的软件。

1.2 打包的作用

如果你是打包新手，那么目前看来，打包似乎主要是为了与全球各地的人共享软件。虽然这确实是打包代码的优势，但在开发软件时打包还能带来以下好处：

- 更强的内聚性和封装性。
- 更清晰的所有权定义。
- 代码区域之间的耦合度更低。
- 更多的组合机会。

下面详细介绍这些好处。

1.2.1　通过打包实现内聚和封装

代码的某一特定区域通常只负责一项工作。内聚性衡量的是代码完成某项工作的专注程度。功能越杂乱无章，代码的内聚性就越差。

你可能曾经使用函数、类、模块和导入软件包来组织 Python 代码(参见 Dane Hillard 所著的 *The Hierarchy of Separation in Python*，Manning 出版社，2020 年，第 25-39 页，http://mng.bz/m2N0)。这些结构都在代码的特定工作区域周围设置了一种命名界限。如果命名工作做得好，就能向开发人员传达哪些属于界限内的内容，哪些不属于(后者更重要)。

尽管已经尽了最大努力，但仍然存在命名的不精确性和开发者对代码边界的模糊理解。如果将所有 Python 代码都放在一个应用程序中，那么有些代码最终可能会渗透到不属于它的区域。回想一下你开发过的那些大型项目，你创建了多少个包含各种功能的 utils.py 或 helpers.py 模块？通过函数或模块创建的边界很容易被打破。这些代码中的“工具”区域往往会吸引新的“工具”加入，导致内聚性随时间的推移而降低。

假设你的自动驾驶汽车系统可以使用激光雷达(详见 https://oceanservice.noaa.gov/facts/lidar.html)作为输入。但 CarCorp 公司的车辆并未配备激光雷达传感器。作为一位勤奋的开发人员，你创建了针对激光雷达的特定代码部分，以便将其与其他关注点分离开来。尽管通过评估命名并定期重构代码库可以保持较高的内聚性，但这也会带来维护上的负担。发布软件包提高了在非相关区域添加代码的门槛。因为任何更新都需要经过打包、发布和安装更新的完整流程，这一周期促使开发人员对他们所做的更改进行更深入的思考。

因此，如果没有明确的意图且值得投入更新周期，开发人员就不太可能向软件包中添加代码。

创建具有内聚性的代码区域并将其打包是实现封装的一个方法。通过定义代码的行为是否暴露以及如何暴露，封装可以帮助你与用户建立正确的预期，让他们知道如何与你的代码进行交互。回想一下你创建并分享给他人使用的项目。想想你修改了多少次代码，以及他们因此又不得不修改多少次代码。这让他们有多沮丧？封装可以通过更好地定义 API 合约来减少这种频繁的更改。图 1.3 展示了如何将这些区域打包成多个软件包。

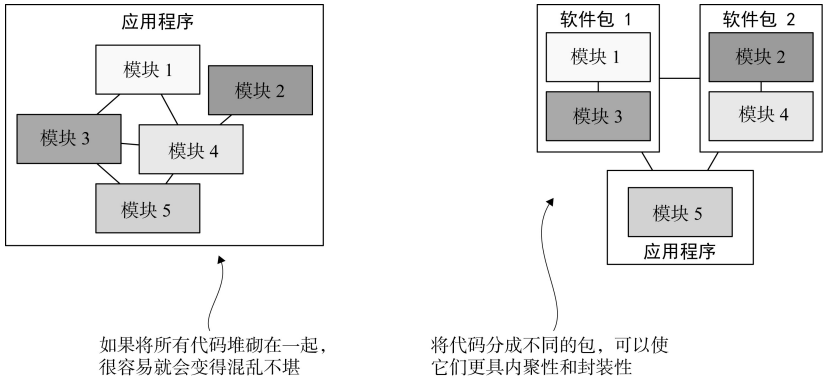


图 1.3 通过引入更强的界限，打包可以减少代码区域之间意外的相互依赖

过去，当你发现一段本应仅用于模块内部的代码被广泛应用于整个代码区域时，可能会备感沮丧。毕竟每次更新“内部”代码时，都需要逐一更新其他地方的用法。在这种高频率变更的环境中，如果未能将更改传播到所有相关地方，就可能会导致错误，从而大大降低你或团队的工作效率。

封装良好、高内聚的代码即使被广泛使用，也很少需要更改。这种代码有时被称为“成熟代码”。成熟的代码非常适合打包发布，因为不需要经常重新发布。可以从代码库中提取一些比较成熟的代码，然后利用所掌握的内聚和封装知识，使不太成熟的代码也能达

到要求，从而开始打包工作。

1.2.2　促进代码所有权的明确化

明确代码区域的所有权对团队大有裨益。所有权不仅涉及维护代码本身的行为，团队还会通过构建自动化来简化单元测试、部署、集成测试、性能测试等工作。这需要同时处理很多事情。保持较小的代码界限区域范围，以便团队能够掌控所有这些方面，这将有助于确保代码的持久性。打包是管理范围的一种有效工具。

通过打包代码形成的封装，能够开发出独立于其他代码的自动化功能。例如，对于结构简单的代码库而言，自动化可能需要编写条件逻辑，以根据更改的文件决定运行哪些测试。或者，可以在每次更改时运行所有测试，尽管这可能会很慢。创建可以独立于其他代码进行测试和发布的软件包，将使源代码、测试代码和发布代码之间的映射更加清晰(见图 1.4)。

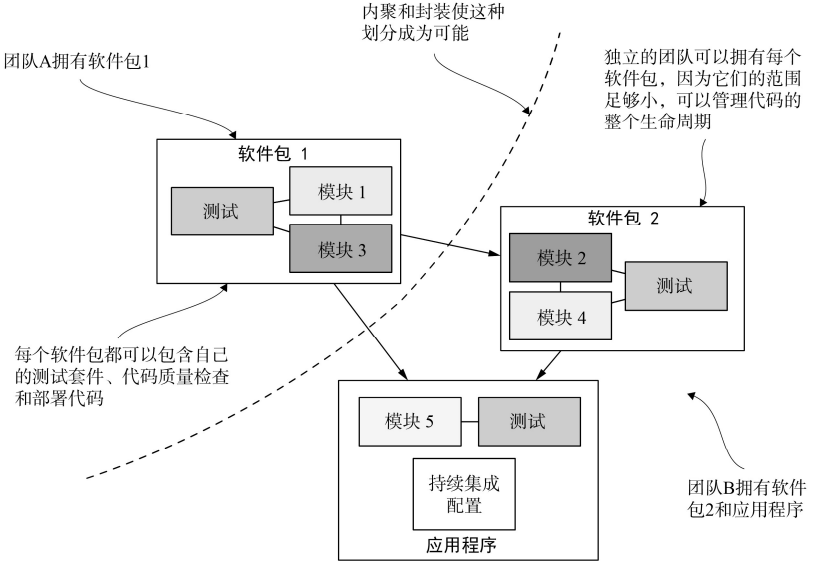


图 1.4　团队可以全权管理单个软件包，自行决定如何管理其开发、测试和发布生命周期

对软件包的目的进行明确划分，就更有可能对所有权进行明确划分。如果一个团队不清楚他们对某些代码的所有权是什么，就会心存顾忌。试着提供一个具有明确范围、背景故事和操作手册的软件包，看看团队的情绪会发生怎样的变化。

1.2.3 实现与使用解耦

你可能听说过“松耦合”这个术语，该术语用来描述代码区域之间的相互依赖程度。

定义：耦合是衡量代码区域之间相互依赖程度的指标。松耦合代码提供了多种灵活的途径，因此可以实施并选择各种执行策略，而不是被迫遵循一条特定的路径。低耦合的两段代码间几乎不存在相互依赖关系，而且可以以不同的速度进行更改。

本章前面提及的内聚和封装实践是一种减少因代码组织不当而导致紧密耦合的可能性的方法。内聚性高的代码内部耦合较紧，与界限外的耦合较松。封装通过暴露一个有意设计的 API 来限制与该 API 的耦合度。因此，合适的打包和封装的策略有助于你将用户与代码中的实现细节解耦。封装还可以通过版本控制、命名空间，甚至编写软件的编程语言，与用户实现解耦。

在一个大的代码“泥团”中，你只能运行每个模块中的代码。如果你或团队中的某个人更新了某个模块，那么所有使用该模块的代码都必须立即适应这一更改。如果更新改变了函数的调用签名或返回值，它的影响范围可能会很大。而打包可以大大减少这种限制(见图 1.5)。

试想一下，如果请求包的每次更新都要求你立即做出反应，迫使你不断更新自己的代码。那将是怎样的噩梦！由于软件包会对其包含的代码进行版本控制，而且用户可以指定安装版本，因此软件包可以在不影响用户代码的情况下进行多次更新。开发人员可以精确地选择何时更新代码，以适应软件包最新版本的更改。

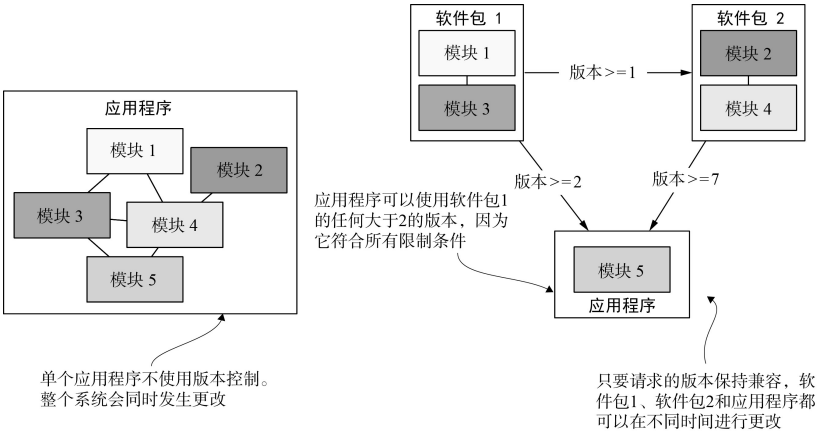


图 1.5　打包提供了灵活性，使两个区域的代码可以以不同的速度进行改进和优化

另一个可以实现代码解耦的地方是命名空间。命名空间将值和行为附加到人类可读的名称上。当安装一个软件包时，就可以在它指定的命名空间中使用它。例如，requests 软件包在 requests 命名空间中可用。

不同的软件包可以拥有相同的命名空间。这意味着如果安装了多个软件包，那么它们就可能会发生冲突，但命名空间的这种灵活性也意味着软件包可以相互替代。如果开发者创建了一个更快、更安全或更易维护的流行软件包的替代品，那么只要 API 相同，就可以安装它来替代原来的软件包。例如，以下软件包都提供了与 MySQL(https://www.mysql.com)大致相同的客户端功能。具体来说，它们在一定程度上实现了与 PEP 249(https://www.python.org/dev/peps/pep-0249/)的兼容：

- mysqlclient (https://github.com/PyMySQL/mysqlclient)
- PyMySQL (https://github.com/PyMySQL/PyMySQL)
- mysql-Python(https://github.com/arnaudsj/mysql-python)
- oursql (https://github.com/python-oursql/oursql)

最后，Python 软件包甚至可以实现 Python 调用层与底层实现语言(编写软件包的语言)的解耦！许多 Python 软件包都是用 C 甚至 Fortran 编写的，以提高性能或方便与遗留系统集成。如果需要，软

件包作者可以提供这些软件包的预编译版本，用户也可以从源代码构建版本。这也使得软件包更具可移植性，使开发人员在一定程度上与他们正在使用的计算机或服务器的细节相解耦。更多关于打包构建目标的信息详见第 3 章。

也可以打包一些代码，尝试版本解耦带来的自由度，看看自己的版本化软件包如何随时间演变。那些更改较频繁的软件包可能表明其内聚性较低，毕竟代码频繁更改通常有其原因。另一方面，这可能只是表明代码仍在不断成熟。至少，这些有关版本变更的关键数据点是可以观察到的！更多关于版本管理的信息详见第 9 章。

1.2.4 通过组合小软件包填补角色空缺

将代码提取为多个软件包的行为有点类似于分解。成功的分解需要妥善处理松耦合。分解代码是一门艺术，可以分离代码段，使它们能够以新的方式重新组合(有关分解和耦合的精彩简述，可参阅 Josh Justice 撰写的 *Breaking Up Is Hard to Do：How to Decompose Your Code，Big Nerd Ranch*，http://mng.bz/5mpq)。

通过对代码中较小的部分进行打包，就能识别出那些能实现特定目标的代码，而这些目标可以进一步概括或扩展，以实现某种作用。例如，可以使用内置的 Python 工具(如 urllib.request.urlopen)创建一次性 HTTP 请求。一旦你多次进行这样的实践，就会发现用例之间的共性，进而将这一概念推广到更高级的实用程序中。因此，requests 软件包并不是为了执行某个特定的 HTTP 请求而构建的；它扮演着 HTTP 客户端的一般角色。某些代码目前可能非常具体，但一旦发现需要类似行为的新领域，就可能发现一个确定所需角色的契机，将其稍加归纳整理，便可创建一个填补该角色空缺的软件包。

为 CarCorp 公司改造的软件代码中，有很大一部分是关于汽车导航系统的。只要稍加调整，导航代码也能用于 Acme Auto 公司的车辆。这段代码可以充当与汽车导航系统通信的角色。因为我们已经了解到软件包可以依赖于其他软件包，而且导航系统代码本身具有高内聚性，所以我们有信心在下一次 CarCorp 会议之前创建两个

软件包，而不是一个。

组合的成功故事

可以通过Python框架(如Django，可访问链接https://www.djangoproject.com)，观察到组合在打包中发挥作用的绝佳实例。Django 本身就是一个软件包，并且由于它是基于插件的架构，可以通过安装和配置附加软件包来扩展其功能。浏览 Django 软件包网站(https://djangopackages.org)上列出的数百个软件包，即可了解打包方法的广泛应用。

有关组合和分解的思考表明一个事实：发布的软件包可以具有任意的大小，就像函数、类、模块和导入软件包一样。在寻找合适的平衡点时，内聚和解耦应成为我们的指导原则。如果 100 个发布软件包各提供一个函数，那么维护就成为负担；如果一个发布的软件包提供 100 个导入软件包，则等同于没有打包。如果其他办法都不奏效，不妨自问：“我想让这段代码扮演什么角色？”

至此，我们已经了解到，打包可以帮助编写内聚且松耦合的代码，这些代码具有明确的所有权，可以以一种易于访问的方式提供给用户。接下来，让我们继续深入探讨细节。

1.3　小结

- 软件包可归档软件文件和有关软件的元数据(如名称、创建者、许可证和版本)。
- 软件包管理器可自动安装软件包并管理它们之间的依赖项。
- 打包过程虽然存在许多缺陷，但可以通过工具和可重复的过程来克服。
- 软件仓库托管已发布的软件包，供他人安装使用。
- 打包是分离和封装具有高内聚性代码的好方法。
- 打包可用作解耦工具，以获得开发和维护代码的灵活性。
- 对软件包进行版本控制是减少每次更新时代码库流失的好方法。

第 2 章

为打包开发做准备

本章涵盖如下内容：

- 使用 venv 管理虚拟环境
- 使用虚拟环境隔离项目依赖项
- 使用 venv 管理虚拟环境的创建和激活
- 使用 pip 列出已安装的依赖项

在项目开始时，你很可能会急于开展工作，完成一些实实在在的事情。这可以理解，立即着手解决问题确实会带来回报。但是，稳扎稳打、逐步推进更是一种明智的策略，随着项目的逐渐成熟，这种策略可以让进展更迅速，并且维持得更长久。在探索一种新技术或流程时，若先采用此策略进行练习，便能达到熟能生巧之效。事先做一些规划可以大大提升工作效率和团队士气。本章将使用 asdf 和 venv，为本书后续将要使用的软件包创建一个开发环境。

重点：在继续阅读之前，请先阅读附录 A 安装本章所需的工具。

2.1 管理 Python 虚拟环境

当我们更深入考虑 CarCorp 项目的潜在成功时，就会意识到，如果这个即将发布的软件包变得流行起来，那么使用各种 Python 版本的用户可能都想要安装和使用它。由于他们不可能总是在生产系统上运行最新版本的 Python，因此明确说明软件包支持的 Python 版本范围，并在所有这些版本中测试软件包，是一种很好的实践。因为采用了附录 A 中的 asdf 和 python-launcher 的功能，我们已经获得了实现这一目标所需的大部分功能。最后一步是创建一个虚拟环境，用于软件包的本地开发。

安装 Python 时，会附带一系列软件包，这些软件包在 Python 标准库中可用。

定义：标准库定义了哪些功能被视为编程语言的核心部分。一种语言的标准库内置于该语言或其安装过程中，并且在系统上安装该语言的软件后默认可用。

与某些语言相比，Python 的标准库非常丰富，但即便如此，它也不能提供项目可能需要的所有功能。Python 软件包、Python 软件包索引(PyPI)和 pip 软件包管理器的存在就是为了共享 Python 标准库之外的软件或提供 Python 标准库的替代品。

假设在 CarCorp 项目初次启动时，便使用 pip 安装了一些 request 之类的软件包，还从 Vehicle Ventures 的早期项目中安装了一些其他软件包。你是否注意到，无论在哪个项目中使用这些软件包，它们最终都会集中安装到某一个位置？

默认情况下，pip 会将软件包安装到与安装时所使用的 Python 版本相关的位置，即站点软件包目录。也就是说，当安装 Python 3.7 并使用它自带的 pip 副本时，安装的软件包将存储在 Python 3.7 的站点软件包目录中。将所有软件包都安装到这个站点软件包目录中可能尚能应付，但当需要为不同的项目安装不同版本的软件包时，该

怎么办呢？如果需要列出单个项目所需的最小依赖项列表，又该怎么办？如果站点软件包目录中充满了来自各个项目的软件包，这些问题就很难解决，甚至无法解决。

解决这些问题的方法之一是隔离每个项目中的软件包。在隔离的情况下，可以保存一份每个项目所需的最小依赖项列表。此外，即使一个项目使用 requests==2.24.0，另一个项目也可以自由使用 requests==2.1.0。第 1 章已经讲解了解耦的价值。软件包依赖项的隔离能让项目相互解耦。可以在 Python 中使用虚拟环境来实现这种隔离。

定义：Python 虚拟环境是一个隔离的 Python 副本，具有隔离的站点软件包目录。虚拟环境 Python 中的 pip 副本会将软件包安装到其隔离的站点软件包目录中，使之与其他环境分开。

虚拟环境在概念上与普通的 Python 安装并没有什么不同。与其安装 Python 3.7 并将所有项目的依赖项都安装到其中，不如多次安装 Python 3.7，并为每个安装赋予一个与每个项目对应的唯一名称。随后，便可以将每个唯一命名的 Python 安装程序用于其对应的项目(见图 2.1)。虚拟环境的实际工作方式与此相差无几。

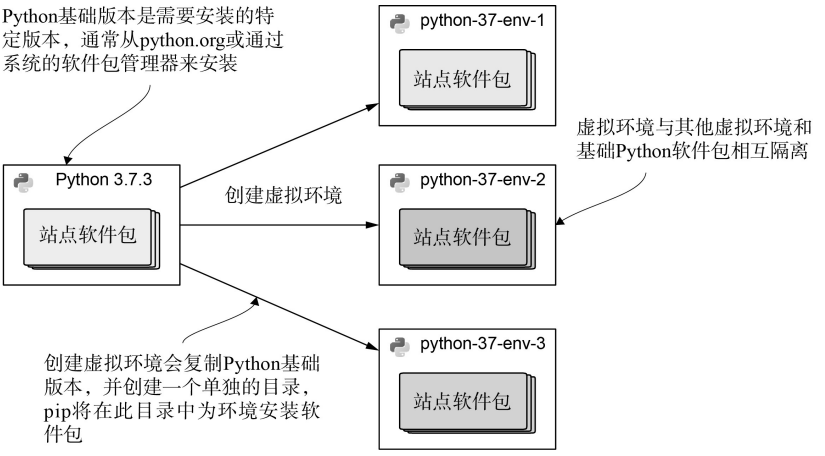


图 2.1　虚拟环境创建了 Python 和 pip 的隔离副本，它们有自己的软件包安装目录

在虚拟环境中使用 Python 时，将用到环境创建时所用的 Python 基础版本的副本。

若要测试软件包，不仅需要安装与其他项目隔离的软件包，还需要在多个 Python 基础版本中安装软件包。随着项目支持的 Python 版本数量增加，手动管理所有虚拟环境及其 Python 安装会变得非常烦琐(见图 2.2)。

你可能已经开始意识到工具能让这些事情变得井井有条。asdf 可以帮助安装和管理 Python 基础版本，而 venv 则可以帮助从这些基础 Python 版本创建虚拟环境。

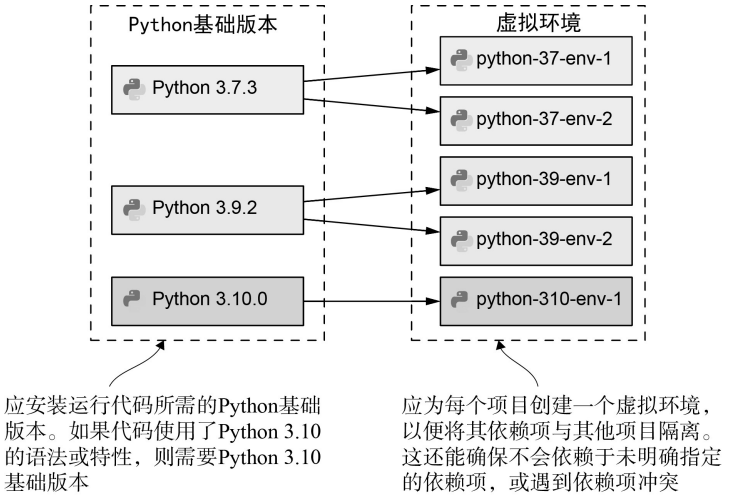


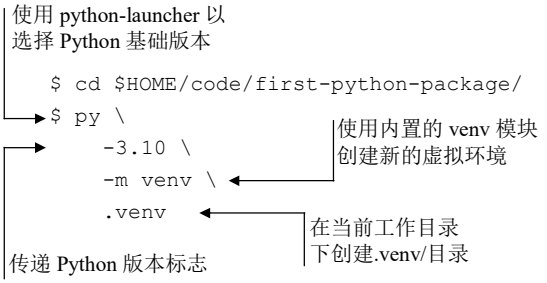
图 2.2 一个系统中可能存在许多 Python 基础版本，每个版本都有许多虚拟环境

使用 venv 创建虚拟环境

使用 asdf 直接从互联网上下载源代码并进行安装，便可在你的系统上部署一个 Python 基础版本。使用一个唯一的名称复制一个已安装的 Python 基础版本，即可创建一个虚拟环境。

使用 Python 基础版本中的 venv 模块，并向它传递虚拟环境目录的名称，即可创建虚拟环境。通常，习惯将该目录称为.venv/。现

在，运行以下命令在项目中创建虚拟环境：



如果命令成功执行，通常不会有任何输出，但应该会创建一个.venv/目录。Unix 系统上的 python-launcher 会检测到这个新虚拟环境的存在，并在你进入该目录或其子目录时默认使用它。如果虚拟环境当前处于活动状态，Windows 的 Python 启动器就会选择该虚拟环境。可以通过运行不带参数的 py 命令来验证这一点。启动的解释器应当与创建虚拟环境时使用的 Python 基础版本相匹配，可以使用下面的代码来确定它就是虚拟环境的 Python 副本：

```
>>> import sys
>>> sys.executable
'/Users/<you>/code/first-python-package/.venv/bin/python'
```

如果向 python-launcher 传递版本标志，仍会得到基础版本。例如，在使用 py -3.9 时，将显示类似下面的内容：

```
>>> import sys
>>> sys.executable
'/Users/<you>/.asdf/installs/python/3.9.3/bin/python3.9'
```

要证明虚拟环境与创建时的 Python 基础版本是隔离的，首先要在项目目录下运行以下命令，在虚拟环境中安装 request 软件包，并检查安装包的列表：

```
$ py -m pip install requests
$ py -m pip list
```

```
Package            Version
------------------ ---------
certifi            2022.6.15
charset-normalizer 2.0.12
idna               3.3
pip                21.2.4
requests           2.28.0
setuptools         58.1.0
urllib3            1.26.9
```

现在通过显式传递-3.10 版本标志来确认这些软件包没有安装在 Python 基础版本中：

```
$ py -3.10 -m pip list
Package    Version
---------- -------
pip        21.1.2
setuptools 57.0.0
```

可以看到，创建虚拟环境后，默认情况下只安装了 pip 和 setuptools 软件包。这些默认软件包及其版本是由 Python 基础安装决定的。最好养成将 pip 和 Setuptools 更新到最新可用版本的习惯，并安装 wheel 软件包，这样就可以安装为系统预构建的软件包，而不用自己编译。现在就安装这些软件包：

```
$ py -m pip install --upgrade pip setuptools wheel
```

今后，你将可以在项目中使用 py 命令，并确保始终能从项目的虚拟环境中获得 Python 副本，除非明确要求使用不同的 Python(基础)。这样可以减轻认知负担，因为每次开始或停止项目工作时，都不再需要手动激活或停用虚拟环境。

提示：如果习惯在 PyCharm (https://www.jetbrains.com/pycharm/)或 Visual Studio Code(https://code.visualstudio.com/)等集成开发环境中自动使用虚拟环境，那么即使在命令行中使用的是 python-launcher，也可以这么做；.venv/目录仍然是一个标准的虚拟环境。

到此已经讲解了使用 asdf 和 venv 管理 Python 版本和虚拟环境的来龙去脉。现在，创建第一个 Python 软件包的所有准备工作已就绪。

2.2　小结

- 虚拟环境能够解耦并隔离不同 Python 项目的依赖项。
- 使用 python-launcher 可确保获得正确的 Python 版本。