Two Scoops of Django 3.x

Django 3.x 最佳实践中文译稿

Daniel Roy Greenfeld & Audrey Roy Greenfeld

由袅 残 烟翻 译

June 8, 2026

本 PDF 由 本 地 Markdown 章 节 自 动 生 成

目录

献辞 **xvii**

关于这篇献辞 **xviii**

前言 **xix**

丹尼尔・费尔德罗伊的几句话 **xx**

奥黛丽・费尔德罗伊的几句话 **xxi**

引言 **xxii**

0.1 关于我们的建议 . . . xxii
0.2 为什么叫《两勺 Django》? . . . xxii
0.3 开始之前 . . . xxii
0.4 本书面向 Django 3.x 与 Python 3.8 或 3.9 . . . xxiii
0.5 每一章都可以独立阅读 . . . xxiii
0.6 本书使用的约定 . . . xxiii
0.7 核心理念 . . . xxiv
0.7.1 保持简单，别把事情搞复杂 . . . xxiv
0.7.2 胖模型、工具模块、瘦视图、傻模板 . . . xxv
0.7.3 默认先用 Django 原生方案 . . . xxv
0.7.4 熟悉 Django 的设计哲学 . . . xxv
0.7.5 十二要素应用 . . . xxv
0.8 我们的写作理念 . . . xxv
0.8.1 提供最好的内容 . . . xxv
0.8.2 站在巨人的肩膀上 . . . xxvi
0.8.3 倾听读者和审稿人的声音 . . . xxvi
0.8.4 发布问题与勘误 . . . xxvi

第一章 编码风格 **1**

1.1 让代码易读的重要性 . . . 1
1.2 PEP 8 . . . 1
1.2.1 79 字符限制 . . . 2
1.3 关于 import . . . 2
1.4 理解显式相对导入 . . . 3
1.5 避免使用 `import *` . . . 4
1.5.1 其他 Python 命名冲突 . . . 4
1.6 Django 的编码风格 . . . 6
1.6.1 了解 Django 编码风格指南 . . . 6
1.6.2 URL 模式名里用下划线，不用短横线 . . . 6
1.6.3 模板 block 名里也用下划线，不用短横线 . . . 7
1.7 选择 JS、HTML 和 CSS 风格指南 . . . 7
1.7.1 JavaScript 风格指南 . . . 7
1.7.2 HTML 和 CSS 风格指南 . . . 7
1.8 永远不要为集成开发环境（IDE）或文本编辑器而写代码 . . . 7
1.9 小结 . . . 7

第二章 最优的 Django 环境配置 **8**

2.1 在所有环境里使用同一种数据库引擎 . . . 8

2.2.1 virtualenvwrapper . . . 10
2.3 通过 pip 安装 Django 和其他依赖 . . . 10
2.4 使用 Git 做版本控制 . . . 10
2.5 可选方案：完全一致的环境 . . . 12
2.5.1 Docker . . . 12
2.6 小结 . . . 13

第三章 如何组织 Django 项目结构 14
3.1 Django 3 默认生成的项目结构 . . . 14
3.2 我们偏好的项目结构 . . . 15
3.2.1 顶层：仓库根目录 . . . 15
3.2.2 第二层：Django 项目根目录 . . . 15
3.2.3 第二层：配置根目录 . . . 15
3.3 示例项目结构 . . . 17
3.4 virtualenv 放哪儿？ . . . 18
3.4.1 列出当前依赖 . . . 20
3.5 超越 startproject . . . 20
3.5.1 用 Cookiecutter 生成项目脚手架 . . . 21
3.5.2 用 Cookiecutter Django 生成起始项目 . . . 21
3.6 startproject 的其他替代方案 . . . 21
3.7 小结 . . . 22

第四章 Django App 设计基础 23
4.1 Django App 设计的黄金法则 . . . 23
4.1.1 一个项目里 app 的实际例子 . . . 24
4.2 Django app 应该怎么命名 . . . 24
4.3 拿不准的时候，就让 app 保持小而专 . . . 25
4.4 一个 app 里应该放哪些模块？ . . . 25
4.4.1 常见的 app 模块 . . . 25
4.4.2 不常见的 app 模块 . . . 26
4.5 另一种路线：更偏 Ruby on Rails 风格的做法 . . . 27
4.5.1 服务层 . . . 27
4.5.2 大型单 App 项目 . . . 29
4.6 小结 . . . 30

第五章 Settings 与 Requirements 文件 31
5.1 避免不受版本控制的本地 settings . . . 31
5.2 使用多个 Settings 文件 . . . 32
5.2.1 一个开发 settings 示例 . . . 33
5.2.2 多份开发 settings . . . 34
5.3 把配置与代码分离 . . . 35
5.3.1 在用环境变量存 secret 前，先提醒一句 . . . 35
5.3.2 如何在本地设置环境变量 . . . 35
5.3.3 如何在本地取消环境变量 . . . 36
5.3.4 如何在生产环境设置环境变量 . . . 37
5.3.5 处理缺失 Secret Key 时的异常 . . . 37

5.4 当你无法使用环境变量时 . . . 38
5.4.1 使用 JSON 文件 . . . 38
5.4.2 使用 .env、Config、YAML 和 XML 文件格式 . . . 39
5.5 使用多个 Requirements 文件 . . . 39
5.5.1 从多个 Requirements 文件安装 . . . 40
5.6 在 Settings 中处理文件路径 . . . 41
5.7 小结 . . . 43

第六章 模型最佳实践 44
6.1 基础 . . . 44
6.1.1 一个 app 里模型太多时，就把它拆开 . . . 44
6.1.2 谨慎使用模型继承 . . . 44
6.1.3 实战中的模型继承：TimeStampedModel . . . 45
6.2 数据库迁移 . . . 46
6.2.1 创建迁移时的小建议 . . . 46
6.2.2 在迁移里加入 Python 函数和自定义 SQL . . . 46
6.3 解决 RunPython 的常见障碍 . . . 46
6.3.1 获取自定义模型 manager 的方法 . . . 47
6.3.2 获取自定义模型方法 . . . 47
6.3.3 用 RunPython.noop 什么也不做 . . . 47
6.3.4 迁移的部署与管理 . . . 48
6.4 Django 模型设计 . . . 48
6.4.1 从规范化开始 . . . 50
6.4.2 在反规范化之前，先尝试缓存 . . . 50
6.4.3 只有在绝对必要时才反规范化 . . . 50
6.4.4 什么时候用 null 和 blank . . . 50
6.4.5 什么时候用 BinaryField . . . 52
6.4.6 尽量避免使用泛型关联 . . . 53
6.4.7 把 choices 和 sub-choices 做成模型常量 . . . 53
6.4.8 用枚举类型定义 choices . . . 54
6.4.9 PostgreSQL 专属字段：什么时候用 null 和 blank . . . 55
6.5 模型 _meta API . . . 55
6.6 模型 Managers . . . 56
6.7 理解 Fat Models . . . 57
6.7.1 Model Behaviors，也就是 Mixins . . . 57
6.7.2 无状态辅助函数 . . . 57
6.7.3 Model Behaviors 与 Helper Functions . . . 57
6.8 额外资源 . . . 58
6.9 小结 . . . 58

第七章 查询与数据库层 59
7.1 处理单个对象时，用 get_object_or_404() . . . 59
7.2 小心那些可能抛异常的查询 . . . 59
7.2.1 ObjectDoesNotExist 与 DoesNotExist . . . 59
7.2.2 本来只想取一个对象，结果却回来了三个 . . . 60

7.3 利用惰性求值，让查询更好读 . . . 60
7.3.1 为了可读性而链式排版查询 . . . 61
7.4 多依靠高级查询工具 . . . 62
7.4.1 查询表达式 . . . 62
7.4.2 数据库函数 . . . 63
7.5 不到必要时，不要下沉到 Raw SQL . . . 64
7.6 按需添加索引 . . . 64
7.7 事务 . . . 64
7.7.1 把每个 HTTP 请求都包在事务里 . . . 66
7.7.2 显式声明事务 . . . 67
7.7.3 `django.http.StreamingHttpResponse` 与事务 . . . 68
7.7.4 MySQL 中的事务 . . . 68
7.7.5 Django ORM 事务资源 . . . 68
7.8 小结 . . . 69

第八章 基于函数的视图与基于类的视图 70
8.1 什么时候用 FBV，什么时候用 CBV . . . 70
8.2 不要把 View 逻辑塞进 URLConf . . . 70
8.3 在 URLConf 中坚持松耦合 . . . 72
8.3.1 如果我们不用 CBV 呢？ . . . 73
8.4 使用 URL 命名空间 . . . 74
8.4.1 让 URL 名更短、更直观，也更符合 Don’t Repeat Yourself . . . 75
8.4.2 提升与第三方库的互操作性 . . . 75
8.4.3 更容易搜索、升级和重构 . . . 76
8.4.4 让 app 与模板中的 reverse 技巧更多一些 . . . 76
8.5 尽量让业务逻辑远离 Views . . . 76
8.6 Django 的 Views 本质上都是函数 . . . 76
8.6.1 最简单的 Views . . . 77
8.7 不要在 View Context 里用 `locals()` . . . 77
8.8 小结 . . . 78

第九章 基于函数的视图最佳实践 79
9.1 FBV 的优势 . . . 79
9.2 传递 `HttpRequest` 对象 . . . 79
9.3 Decorator 很甜 . . . 81
9.3.1 使用 Decorator 时要克制 . . . 83
9.3.2 Decorator 的额外资源 . . . 83
9.4 传递 `HttpResponse` 对象 . . . 84
9.5 基于函数的视图更多资源 . . . 84
9.6 小结 . . . 84

第十章 基于类的视图最佳实践 85
10.1 使用 CBV 时的准则 . . . 85
10.2 在 CBV 中使用 Mixins . . . 85
10.3 针对不同任务，该用哪个 Django GCBV？ . . . 86
10.4 Django CBV 的通用技巧 . . . 87

10.4.1 把 Django CBV / GCBV 限制为“仅认证用户可访问” . . . 87
10.4.2 表单有效时，在 View 中执行自定义动作 . . . 88
10.4.3 表单无效时，在 View 中执行自定义动作 . . . 88
10.4.4 使用 View 对象本身 . . . 89
10.5 GCBV 与 Forms 是怎么配合的 . . . 90
10.5.1 Views + ModelForm 示例 . . . 91
10.5.2 Views + Form 示例 . . . 93
10.6 只使用 `django.views.generic.View` . . . 95
10.7 额外资源 . . . 96
10.8 小结 . . . 97

第十一章 异步视图 **98**
11.1 对 Django 3.1a 预发布版异步视图的分析笔记 . . . 98
11.2 资源 . . . 98

第十二章 Forms 的常见模式 **99**
12.1 模式 1：带默认验证器的简单 ModelForm . . . 99
12.2 模式 2：在 ModelForms 中使用自定义表单字段验证器 . . . 99
12.3 模式 3：覆写验证中的 clean 阶段 . . . 103
12.4 模式 4：Hack 表单字段 . . . 105
12.5 模式 5：可复用的搜索 Mixin View . . . 108
12.6 小结 . . . 110

第十三章 Form 基础 **111**
13.1 用 Django Forms 验证所有传入数据 . . . 111
13.2 在 HTML Forms 中使用 POST 方法 . . . 113
13.3 对会修改数据的 HTTP Forms，始终使用 CSRF 保护 . . . 113
13.3.1 通过 AJAX 提交数据 . . . 113
13.4 理解如何给 Django Form 实例添加属性 . . . 114
13.5 理解表单验证是如何工作的 . . . 115
13.5.1 ModelForm 数据先保存到 Form，再保存到 Model 实例 . . . 115
13.6 用 `Form.add_error()` 给表单添加错误 . . . 117
13.6.1 其他值得了解的表单方法 . . . 117
13.7 没有现成 Widget 的字段 . . . 117
13.8 自定义 Widgets . . . 117
13.8.1 覆写内建 Widget 的 HTML . . . 118
13.8.2 创建新的自定义 Widgets . . . 118
13.9 额外资源 . . . 119
13.10 小结 . . . 119

第十四章 Templates：最佳实践 **120**
14.1 尽量把 Templates 放在 `templates/` 目录下 . . . 120
14.2 Template 架构模式 . . . 120
14.2.1 两层 Template 架构示例 . . . 121
14.2.2 三层 Template 架构示例 . . . 121
14.2.3 扁平优于嵌套 . . . 121

14.3 限制在 Templates 中做处理 . . . 122
14.3.1 坑 1：在 Templates 中做聚合 . . . 124
14.3.2 坑 2：在 Templates 中用条件判断做过滤 . . . 125
14.3.3 坑 3：Templates 中隐含的复杂查询 . . . 126
14.3.4 坑 4：Templates 中隐藏的 CPU 负载 . . . 127
14.3.5 坑 5：Templates 中隐藏的 REST API 调用 . . . 128
14.4 别费劲把生成出来的 HTML 弄得很好看 . . . 128
14.5 探索 Template 继承 . . . 129
14.6 `block.super` 赋予你控制力 . . . 131
14.7 一些值得考虑的小事 . . . 132
14.7.1 不要把样式和 Python 代码绑得太死 . . . 132
14.7.2 常见约定 . . . 133
14.7.3 正确使用隐式和具名显式 Context 对象 . . . 133
14.7.4 使用 URL 名称，而不是硬编码路径 . . . 133
14.7.5 调试复杂 Templates . . . 134
14.8 错误页面模板 . . . 134
14.9 坚持最简主义做法 . . . 134
14.10 总结 . . . 135

第十五章 Template Tags 和 Filters **136**
15.1 Filters 本质上就是函数 . . . 136
15.1.1 Filters 很容易测试 . . . 136
15.1.2 Filters 与代码复用 . . . 136
15.1.3 什么时候该写 Filters . . . 137
15.2 自定义 Template Tags . . . 137
15.2.1 Template Tags 更难调试 . . . 137
15.2.2 Template Tags 会让代码复用更难 . . . 137
15.2.3 Template Tags 的性能代价 . . . 137
15.2.4 什么时候该写 Template Tags . . . 137
15.3 给你的 Template Tag Libraries 命名 . . . 138
15.4 加载你的 Template Tag Modules . . . 138
15.4.1 小心这个反模式 . . . 138
15.5 总结 . . . 139

第十六章 Django Templates 与 Jinja2 **140**
16.1 语法上有什么区别？ . . . 140
16.2 我应该切换吗？ . . . 140
16.2.1 DTL 的优势 . . . 140
16.2.2 Jinja2 的优势 . . . 140
16.2.3 到底谁赢？ . . . 141
16.3 在 Django 中使用 Jinja2 时需要考虑的事 . . . 141
16.3.1 CSRF 与 Jinja2 . . . 141
16.3.2 在 Jinja2 Templates 中使用 Template Tags . . . 141
16.3.3 在 Jinja2 Templates 中使用 Django 风格的 Template Filters . . . 141
16.3.4 应把 Jinja2 的 Environment 对象视为静态对象 . . . 143

16.4 资源 . . . 144
16.5 总结 . . . 144

第十七章 使用 Django REST Framework 构建 REST APIs 145
17.1 基础 REST API 设计的基本原理 . . . 145
17.2 用一个简单 API 来说明设计概念 . . . 146
17.3 REST API 架构 . . . 149
17.3.1 对 API Modules 使用一致的命名 . . . 149
17.3.2 项目的代码应该组织得干净利落 . . . 149
17.3.3 App 的代码应尽量留在 App 内 . . . 150
17.3.4 尽量把业务逻辑移出 API Views . . . 150
17.3.5 对 API URLs 做分组 . . . 150
17.3.6 测试你的 API . . . 152
17.3.7 给你的 API 做版本控制 . . . 152
17.3.8 对自定义认证方案保持谨慎 . . . 152
17.4 当 DRF 挡路的时候 . . . 153
17.4.1 远程过程调用 vs REST APIs . . . 153
17.4.2 复杂数据带来的问题 . . . 154
17.4.3 简化！原子化！ . . . 155
17.5 关闭一个对外 API . . . 155
17.5.1 第一步：通知用户 API 即将关停 . . . 155
17.5.2 第二步：用 410 Error View 替换旧 API . . . 155
17.6 给你的 API 做限流 . . . 156
17.6.1 不受约束的 API 访问很危险 . . . 156
17.6.2 REST Frameworks 必须提供限流能力 . . . 157
17.6.3 限流本身也可以成为商业计划 . . . 157
17.7 推广你的 REST API . . . 157
17.7.1 文档 . . . 157
17.7.2 提供客户端 SDKs . . . 158
17.8 延伸阅读 . . . 158
17.9 其他 API 构建方案 . . . 158
17.9.1 CBV 路线：`JsonResponse` 搭配 `View` . . . 158
17.9.2 FBV 路线：`django-jsonview` . . . 159
17.9.3 `django-tastypie` . . . 159
17.10 总结 . . . 159

第十八章 使用 Django 构建 GraphQL APIs 161
18.1 打破性能迷思 . . . 161
18.2 GraphQL API 架构 . . . 162
18.2.1 不要把连续递增的 key 当成公开标识符 . . . 162
18.2.2 对 API Modules 使用一致的命名 . . . 162
18.2.3 尽量把业务逻辑移出 API Views . . . 163
18.2.4 测试你的 API . . . 163
18.2.5 给你的 API 做版本控制 . . . 163
18.2.6 对自定义认证方案保持谨慎 . . . 163

18.3 关闭一个对外 API 163

第十九章 JavaScript 与 Django **164**
19.1 流行的 JavaScript 路线 164
19.1.1 当多页应用已足够时，却硬要构建 Single Page App 165
19.1.2 升级遗留站点 165
19.1.3 不写测试 165
19.1.4 不理解 JavaScript 的内存管理 165
19.1.5 在不是 jQuery 的情况下把数据塞进 DOM 166
19.2 用 JavaScript 消费 Django 提供的 APIs 166
19.2.1 学会如何调试客户端 166
19.2.2 在可能的情况下，使用由 JavaScript 驱动的静态资源预处理器 166
19.3 实时性的烦恼，也就是延迟 167
19.3.1 解决方案：用动画掩盖延迟 167
19.3.2 解决方案：伪造成功事务 167
19.3.3 解决方案：基于地理位置部署服务器 167
19.3.4 解决方案：按地域限制用户 167
19.3.5 AJAX 与 CSRF Token 168
19.4 在 Django 提供的 Templates 中使用 JavaScript 168
19.4.1 JavaScript 又可以放回 Header 里了 168
19.4.2 对供 JavaScript 消费的数据使用 JSON 编码 168
19.5 强化你的 JavaScript 技能 169
19.5.1 学更多 JavaScript! 169
19.6 遵循 JavaScript 编码规范 169
19.7 总结 169

第二十章 替换核心组件的权衡 **170**
20.1 构建 FrankenDjango 的诱惑 170
20.2 非关系型数据库 vs 关系型数据库 170
20.2.1 并不是所有非关系型数据库都符合 ACID 171
20.2.2 不要拿非关系型数据库去做关系型任务 171
20.2.3 别理会炒作，自己做研究 172
20.2.4 我们如何在 Django 中使用非关系型数据库 172
20.3 那替换 Django Template Language 呢? 172
20.4 总结 172

第二十一章 使用 Django Admin **173**
21.1 它不是给终端用户用的 173
21.2 定制 Admin 还是新建 Views 173
21.3 查看对象的字符串表示 174
21.3.1 使用 `__str__()` 174
21.3.2 使用 `list_display` 175
21.4 给 `ModelAdmin` 类添加可调用对象 176
21.5 注意多用户环境下的复杂性 177
21.6 Django 的 Admin 文档生成器 178
21.7 在 Django Admin 中使用自定义皮肤 178

21.7.1 评估点：文档就是一切 . . . 178
21.7.2 对你自己写的任何 Admin 扩展都要写测试 . . . 179
21.8 保护 Django Admin . . . 179
21.8.1 修改默认 Admin URL . . . 179
21.8.2 使用 `django-admin-honeypot` . . . 179
21.8.3 只允许通过 HTTPS 访问 Admin . . . 179
21.8.4 基于 IP 限制 Admin 访问 . . . 179
21.9 保护 Admin Docs . . . 180
21.10 总结 . . . 180

第二十二章 处理 User Model **181**
22.1 使用 Django 自带的工具来定位 User Model . . . 181
22.1.1 在指向 User 的外键中使用 `settings.AUTH_USER_MODEL` . . . 181
22.1.2 不要在指向 User 的外键中使用 `get_user_model()` . . . 181
22.2 为 Django 项目定制 User 字段 . . . 183
22.2.1 方案一：继承 `AbstractUser` . . . 183
22.2.2 方案二：继承 `AbstractBaseUser` . . . 184
22.2.3 方案三：从关联 Model 反向挂回去 . . . 184
22.3 处理多种用户类型 . . . 185
22.3.1 增加一个用户类型字段 . . . 185
22.3.2 增加用户类型字段，再配合 Proxy Models . . . 186
22.3.3 增加额外数据字段 . . . 188
22.3.4 关于多种用户类型的补充资源 . . . 189
22.4 总结 . . . 189

第二十三章 Django 的秘诀：第三方 Packages **190**
23.1 第三方 Packages 的例子 . . . 190
23.2 了解 Python Package Index . . . 190
23.3 了解 DjangoPackages.org . . . 191
23.4 了解你的资源 . . . 191
23.5 安装与管理 Packages 的工具 . . . 192
23.6 包依赖要求 . . . 192
23.7 接入 Django Packages：基础流程 . . . 192
23.7.1 第一步：阅读该 Package 的文档 . . . 192
23.7.2 第二步：把 Package 及其版本号加入 Requirements . . . 192
23.7.3 第三步：把 Requirements 安装进你的 Virtualenv . . . 193
23.7.4 第四步：严格按照 Package 的安装说明来 . . . 193
23.8 排查第三方 Packages 的问题 . . . 193
23.9 发布你自己的 Django Packages . . . 193
23.10 什么样的 Django Package 才算好？ . . . 194
23.10.1 目标 . . . 194
23.10.2 范围 . . . 194
23.10.3 文档 . . . 194
23.10.4 测试 . . . 194
23.10.5 模板 . . . 194

23.10.6 活跃度 . . . 194
23.10.7 社区 . . . 195
23.10.8 模块化 . . . 195
23.10.9 在 PyPI 上可用 . . . 195
23.10.10 尽可能使用最宽的 Requirements Specifiers . . . 195
23.10.11 合理的版本号 . . . 196
23.10.12 名称 . . . 196
23.10.13 许可证 . . . 197
23.10.14 代码的清晰度 . . . 197
23.10.15 使用 URL Namespaces . . . 197
23.11 更轻松地创建你自己的 Packages . . . 197
23.12 维护你的开源 Package . . . 198
23.12.1 为 Pull Requests 署名 . . . 198
23.12.2 如何处理糟糕的 Pull Requests . . . 198
23.12.3 正式发布 PyPI 版本 . . . 199
23.12.4 创建并发布 Wheels 到 PyPI . . . 199
23.12.5 给 Repo 加 Git Tags . . . 200
23.12.6 把 Package 升级到新版本 Django . . . 200
23.12.7 遵循良好的安全实践 . . . 200
23.12.8 提供示例基础模板 . . . 200
23.12.9 把 Package 交出去 . . . 200
23.13 补充阅读 . . . 201
23.14 总结 . . . 201

第二十四章 测试真烂，还浪费钱！ 202
24.1 测试能保住金钱、工作，甚至生命 . . . 202
24.2 如何组织测试 . . . 202
24.3 如何编写单元测试 . . . 203
24.3.1 每个测试方法只测一件事 . . . 203
24.3.2 对 views，能用 Request Factory 时就尽量用 . . . 205
24.3.3 不要写那种还得再被测试的测试 . . . 206
24.3.4 Don’t Repeat Yourself 不适用于测试 . . . 206
24.3.5 不要依赖 Fixtures . . . 207
24.3.6 哪些东西应该被测试 . . . 207
24.3.7 为失败场景写测试 . . . 207
24.3.8 用 Mock 让单元测试不要碰到外部世界 . . . 209
24.3.9 用更花哨一点的断言方法 . . . 210
24.3.10 给每个测试写清楚用途 . . . 211
24.4 集成测试怎么办？ . . . 211
24.5 持续集成 . . . 212
24.6 谁在乎？我们可没时间写测试！ . . . 212
24.7 测试覆盖率游戏 . . . 212
24.8 如何搭起测试覆盖率游戏 . . . 212
24.8.1 第一步：开始写测试 . . . 212
24.8.2 第二步：运行测试并生成覆盖率报告 . . . 213

24.8.3 第三步：生成报告！ . . . 213
24.9 玩测试覆盖率这场游戏 . . . 214
24.10 unittest 的替代方案 . . . 214
24.11 总结 . . . 214

第二十五章 文档：痴迷一点 **215**
25.1 文档请用 GitHub-Flavored Markdown . . . 215

第二十六章 一级标题 **216**
26.1 小节标题 . . . 216
26.2 使用 MkDocs 或带 Myst 的 Sphinx 从 Markdown 生成文档 . . . 216
26.3 Django 项目至少应该包含哪些文档? . . . 217
26.4 更多 Markdown 文档资源 . . . 217
26.5 ReStructuredText 这个替代方案 . . . 217
26.5.1 ReStructuredText 资源 . . . 218
26.6 当文档需要在 Markdown 和 ReStructuredText 之间互转时 . . . 218
26.7 Wiki 以及其他文档方法 . . . 219
26.8 确保代码本身也有文档 . . . 219
26.9 总结 . . . 219

第二十七章 发现并减少瓶颈 **220**
27.1 你真的需要在乎这个吗? . . . 220
27.2 让查询密集型页面更快 . . . 220
27.2.1 用 Django Debug Toolbar 找出过量查询 . . . 220
27.2.2 减少查询数量 . . . 220
27.2.3 提速常见查询 . . . 221
27.2.4 把 ATOMIC_REQUESTS 切到 False . . . 221
27.3 榨干数据库的能力 . . . 221
27.3.1 清楚什么东西不该放进数据库 . . . 221
27.3.2 让 PostgreSQL 发挥到最好 . . . 222
27.3.3 让 MySQL 发挥到最好 . . . 222
27.4 用 Memcached 或 Redis 缓存查询 . . . 222
27.5 找出应该缓存的具体位置 . . . 222
27.6 考虑第三方缓存包 . . . 223
27.7 压缩和压缩整理 HTML、CSS 与 JavaScript . . . 223
27.8 使用上游缓存或 Content Delivery Network . . . 224
27.9 其他资源 . . . 224
27.10 总结 . . . 225

第二十八章 异步任务队列 **226**
28.1 我们真的需要任务队列吗? . . . 226

第二十九章 安全最佳实践 **228**
29.1 参考本书其他章节中的安全小节 . . . 228
29.2 强化你的服务器 . . . 228
29.3 了解 Django 自带的安全特性 . . . 228
29.4 生产环境里关闭 DEBUG 模式 . . . 229

29.5 保守你的 Secret Keys 的秘密 . . . 229
29.6 全站 HTTPS . . . 229
29.6.1 使用 Secure Cookies . . . 230
29.6.2 使用 HTTP Strict Transport Security（HSTS） . . . 230
29.6.3 HTTPS 配置工具 . . . 231
29.7 使用 Allowed Hosts 校验 . . . 231
29.8 对会修改数据的 HTTP 表单始终使用 CSRF 保护 . . . 231
29.9 防御跨站脚本（XSS）攻击 . . . 231
29.9.1 优先使用 format_html，不要用 mark_safe . . . 232
29.9.2 不要允许用户设置单独的 HTML 标签属性 . . . 232
29.9.3 对供 JavaScript 消费的数据使用 JSON 编码 . . . 232
29.9.4 小心那些“诡异 JavaScript” . . . 232
29.9.5 添加 Content Security Policy 头 . . . 232
29.9.6 补充阅读 . . . 232
29.10 防御 Python 代码注入攻击 . . . 232
29.10.1 那些会执行代码的 Python 内建函数 . . . 233
29.10.2 那些会执行代码的 Python 标准库模块 . . . 233
29.10.3 那些会执行代码的第三方库 . . . 233
29.10.4 小心基于 Cookie 的 Sessions . . . 233
29.11 用 Django Forms 校验所有传入数据 . . . 234
29.12 在支付字段上关闭自动补全 . . . 234
29.13 谨慎处理用户上传文件 . . . 234
29.13.1 当 CDN 不是选项时 . . . 235
29.13.2 Django 与用户上传文件 . . . 235
29.14 不要使用 ModelForms.Meta.exclude . . . 235
29.14.1 批量赋值漏洞 . . . 237
29.15 不要使用 ModelForms.Meta.fields = "__all__" . . . 237
29.16 小心 SQL 注入攻击 . . . 237
29.17 不要存储不必要的数据 . . . 238
29.17.1 永远不要存信用卡数据 . . . 238
29.17.2 除非法律要求，否则不要存储 PII 或 PHI . . . 238
29.18 监控你的站点 . . . 238
29.19 让依赖保持最新 . . . 238
29.20 防止 Clickjacking . . . 239
29.21 用 defusedxml 防范 XML Bomb 攻击 . . . 239
29.22 考虑启用双因素认证 . . . 239
29.23 拥抱 SecurityMiddleware . . . 240
29.24 强制使用强密码 . . . 240
29.25 不要阻止用户复制 / 粘贴密码 . . . 240
29.26 给你的站点做一次安全体检 . . . 240
29.27 放出一个漏洞报告页面 . . . 241
29.28 绝不展示顺序型主键 . . . 241
29.28.1 按 Slug 查找 . . . 241
29.28.2 UUIDs . . . 241
29.29 将密码哈希器升级到 Argon2 . . . 242

29.30 从外部来源加载静态资源时使用 SRI . . . 242
29.31 参考我们的安全设置附录 . . . 244
29.32 回头看看安全包清单 . . . 244
29.33 持续跟进一般性安全实践 . . . 244
29.34 总结 . . . 244

第三十章 日志：到底是拿来干什么的? 245
30.1 应用日志与其他日志 . . . 245
30.2 为什么要花心思做日志? . . . 245
30.3 什么时候该用哪个日志级别 . . . 245
30.3.1 用 CRITICAL 记录灾难性故障 . . . 246
30.3.2 用 ERROR 记录生产错误 . . . 246
30.3.3 用 WARNING 记录较低优先级的问题 . . . 246
30.3.4 用 INFO 记录有用的状态信息 . . . 247
30.3.5 把与调试相关的消息记到 DEBUG . . . 247
30.4 在捕获异常时把 traceback 一起记下来 . . . 248
30.5 每个使用日志的模块都各自定义一个 logger . . . 250
30.6 在本地把日志写入轮转文件 . . . 250
30.7 其他日志建议 . . . 250
30.8 必读材料 . . . 251
30.9 有用的第三方工具 . . . 251
30.10 总结 . . . 251

第三十一章 Signals：使用场景与规避技巧 252

第三十二章 那些零散工具该怎么办? 253
32.1 为这些工具代码建一个 Core App . . . 253
32.2 用工具模块优化 App . . . 253
32.2.1 存放被多处使用的代码 . . . 253
32.2.2 给 Models 瘦身 . . . 254
32.2.3 更容易测试 . . . 254
32.3 Django 自带的瑞士军刀 . . . 254
32.3.1 `django.contrib.humanize` . . . 255
32.3.2 `django.utils.decorators.method_decorator(decorator)` . . . 255
32.3.3 `django.utils.decorators.decorator_from_middleware(middleware)` . . . 255
32.3.4 `django.utils.encoding.force_text(value)` . . . 255
32.3.5 `django.utils.functional.cached_property` . . . 255
32.3.6 `django.utils.html.format_html(format_str, args, **kwargs)` . . . 255
32.3.7 `django.utils.html.strip_tags(value)` . . . 256
32.3.8 `django.utils.text.slugify(value)` . . . 256
32.3.9 非英语语言中的 Slug 生成 . . . 256
32.3.10 `django.utils.timezone` . . . 256
32.3.11 `django.utils.translation` . . . 257
32.4 异常 . . . 257
32.4.1 `django.core.exceptions.ImproperlyConfigured` . . . 257
32.4.2 `django.core.exceptions.ObjectDoesNotExist` . . . 257

32.4.3 django.core.exceptions.PermissionDenied . . . 258
32.5 序列化器与反序列化器 . . . 258
32.5.1 django.core.serializers.json.DjangoJSONEncoder . . . 260
32.5.2 django.core.serializers.pyyaml . . . 260
32.5.3 django.core.serializers.xml_serializer . . . 261
32.5.4 rest_framework.serializers . . . 261
32.6 总结 . . . 261

第三十三章 部署：平台即服务 262

第三十四章 部署 Django 项目 263

第三十五章 持续集成 264

第三十六章 调试的艺术 265

第三十七章 去哪里，以及如何提出 Django 问题 266

第三十八章 收尾的话 267

附录 A 本书提到的 Packages 268
A.1 核心 . . . 268
A.2 依赖管理 . . . 268
A.3 异步 . . . 268
A.4 数据库 . . . 269
A.5 部署 . . . 269
A.6 文档 . . . 269
A.7 环境变量 . . . 269
A.8 表单 . . . 269
A.9 前端 . . . 269
A.10 日志与性能监控 . . . 269
A.11 项目模板 . . . 270
A.12 REST API 相关 . . . 270
A.13 安全 . . . 270
A.14 测试 . . . 270
A.15 用户注册 . . . 271
A.16 视图 . . . 271
A.17 时间 . . . 271
A.18 杂项 . . . 271

附录 B 安装问题排查 273
B.1 识别问题 . . . 273
B.2 我们推荐的解决方案 . . . 273
B.2.1 检查你的 Virtualenv 安装 . . . 273
B.2.2 检查你的环境里是否安装了正确版本的 Django . . . 274
B.2.3 检查其他问题 . . . 274

附录 C 补充资源 275

C.1 不过时的 Python 与 Django 材料 . . . 275
C.1.1 书籍 . . . 275
C.1.2 Web 资源 . . . 275
C.2 不过时的 Django 入门材料 . . . 276
C.2.1 Web 资源 . . . 276
C.3 不过时的 Python 入门材料 . . . 276
C.4 不过时、而且实用的 Python 材料 . . . 276
C.5 JavaScript 资源 . . . 276
C.5.1 书籍 . . . 276
C.5.2 Web 资源 . . . 276

附录 D 国际化与本地化 **278**
D.1 尽早开始 . . . 278
D.2 用翻译函数包裹内容字符串 . . . 278
D.3 约定：使用下划线别名减少输入 . . . 278
D.4 不要在句子里插值拼词 . . . 278
D.5 Unicode 小技巧 . . . 280
D.5.1 用 `django.utils.encoding.force_text()`，不要用 `str()` . . . 281
D.6 浏览器页面布局 . . . 281
D.7 时间带来的挑战 . . . 281

附录 E Settings 的替代方案 **282**
E.1 Twelve Factor 风格的 Settings . . . 282

附录 F 安全设置参考 **283**
F.1 跨站请求伪造保护相关设置 . . . 283
F.2 电子邮件 SSL . . . 283
F.3 `SESSION_SERIALIZER` . . . 283
F.4 `SECURE_PROXY_SSL_HEADER` . . . 284

附录 G 处理安全故障 **285**
G.1 在事情出错前，就准备好一套方案 . . . 285
G.2 关闭一切，或者切成只读模式 . . . 285
G.3 挂出一个静态 HTML 页面 . . . 285
G.4 把一切都备份下来 . . . 285
G.5 给 `security@djangoproject.com` 发邮件，哪怕这事本来就是你的错 . . . 286
G.6 开始调查问题 . . . 286

附录 H 使用 Channels 处理 WebSockets **287**
H.1 每个浏览器标签页都有自己的 WebSocket 连接 . . . 287
H.2 要预期 WebSocket 连接会随时掉线 . . . 287
H.3 一定要验证传入数据！ . . . 287
H.4 小心意大利面式代码 . . . 288

致谢 **289**
H.5 Python 与 Django 社区 . . . 289
H.6 3.x 版技术审校者 . . . 289

H.7 3.x 版安全审校者 290
H.8 3.x Alpha 贡献者 290
H.9 版技术审校者 290
H.10 版贡献者 290
H.11 版技术审校者 291
H.12 版贡献者 291
H.13 版技术审校者 291
H.14 版技术审校者 291
H.15 版分章审校者 291
H.16 版贡献者 292
H.17 排版 292

献辞

献给马尔科姆・特雷丁尼克 1971-2013 我们想念你。
feldroy.com/pages/malcolm-tredinnick-memorial

关于这篇献辞

马尔科姆·特雷丁尼克（Malcolm Tredinnick）不只是 Django 的核心开发者，也是《Two Scoops of Django: Best Practices for Django 1.5》的审稿人。对我们来说，他的意义远不止如此。

丹尼尔在 2010 年夏天曾与马尔科姆共事，但我们第一次在线下见到他，是在 2010 年的 DjangoCon。他风趣、迷人、主见鲜明，却始终很有绅士风度；我们一下子就和他成了亲密朋友。

2012 年，我们共同组织第一届 PyCon Philippines。我们刚把这件事告诉他，马尔科姆就立刻宣布他会来。他做了两场令人难忘的演讲，还即兴带了一整天的 Django 教程。他也不断推动并鼓励当地社区为 Django 制作菲律宾语言译本，包括 Tagalog、Tausug、Cebuano 等等。

会议结束后，我们开始着手写一本关于 Django 最佳实践的书。我们找来朋友和同事担任技术审稿人，马尔科姆是其中最活跃的一位。他是我们的导师，逼着我们挖得更深、做得更好。那时他白天还在带一支同时使用 Rails 和 Haskell 的团队；马尔科姆是真正精通多种编程语言的人。

为了这本书，他给了我们太多帮助和指导，以至于我们一度想办法把他列进作者署名。我们把这个难题告诉他时，他只是笑着说："一本叫《Two Scoops》的书，怎么能有三个作者呢?"我们提议下一本书和我们共享署名，他却拒绝了，说他更愿意只是评论我们的作品。他说，人们需要可靠的参考资料，而对他来说，单纯审阅我们的工作，本身就是在为更大的公共利益做贡献。后来，我们两个人还悄悄盘算过，想找机会"逼"他在未来的作品里当一次合著者。

经过数月努力，我们在 2013 年 1 月 17 日发布了第一版。此后马尔科姆从《Two Scoops of Django》中退了出来，但我们一直保持联系。由于他没能参加 2013 年的 PyCon US，我们也不知道下一次什么时候才能再见到他。

两个月后，2013 年 3 月 17 日，马尔科姆去世了。

我们认识马尔科姆还不到三年，可他已经深刻地改变了我们的人生。我们在社区里听过很多类似的故事；对世界各地无数人来说，他都是朋友，也是导师。他留给我们的最后一课，早已超越了代码和写作：永远不要把朋友、家人、导师和老师的存在视作理所当然。

前言

《Two Scoops of Django》第一版出版于 2013 年。那时 Django 还只到 1.5 版本，而我自己的 Django 之路也才刚刚开始。我到现在都还记得，当时我绞尽脑汁想真正弄懂这个“电池齐全”的 Web 框架时，有多么挫败。我学过 Python，官方 Django Polls 教程也跑了不止一遍，能找到的书和教程几乎都读过。可我还是觉得迷失。每当轮到我从零开始搭自己的 Django 应用时，我就会撞上一堵墙。Stack Overflow 能回答零散问题，却给不了更大的理解框架。至于开源 Django 代码，先不说我当时能找到的本来就不多，看起来也复杂得令人绝望，远远超出一个新手开发者所能驾驭的范围。

简而言之，我被困在了许多初学者都会掉进去的经典“教程陷阱”里：能照着 Django 教程一步步做下来，却始终没法真正明白自己到底在做什么。

后来我遇到了这本书。终于，眼前出现了一本立场鲜明的 Django 使用指南。它不只谈框架本身，也从整体上看待 Web 应用的构建过程。起初，我对它的编排方式很意外：它没有沿用传统的、一步一步带你做一个示例 Django 应用的写法。整本书根本不去搭一个完整网站。相反，它更像一部百科全书，覆盖了 Django 的各个主要领域。比如我想知道模型到底该怎么配置，就可以直接翻到相关章节，读完建议，马上用到自己的项目里。

读这本书时，我发现了几个错字，便鼓起勇气联系作者丹尼尔和奥黛丽。没想到他们立刻回了信，感谢了我。几周后，一本带着签名和插画的额外样书就寄到了我家门口。那本书至今还摆在我的书架上。

这种欢迎新人加入的互动方式，在 Django 社区里其实相当常见。它大概是我所知道最友善的技术社群，许多 Django 开发者也都有同感。在熟悉 Django 的人之间，一直流传着一句话：“为框架而来，为社区留下。”这话一点不假。

如果你也想参与进来，路子有很多。Django 代码库是开源的，任何人都可以提交 pull request。美国、欧洲、澳大利亚和非洲每年也都会举办 DjangoCon，此外还有数不清的本地聚会和会议。

技术仍在毫不留情地一路向前，Django 以及更广泛的 Python 社区也一直跟得上节奏。从 2013 年那本最初的《Two Scoops》起，后来又陆续出了 Django 1.6、1.8 和 1.11 版本的新版。这些我都买了，也总能从中学到新东西，重新读到多年前就曾让我受益的睿智建议。Python 2 和 Python 3 孰优孰劣、以及相关第三方包支持如何之类的老争论，如今都已尘埃落定；我猜，新一轮争论大概会围绕 Python 新增的异步能力展开，而这些能力也正在一步步进入 Django。

对新手来说，Django 可能显得很有压迫感；至少对当年的我来说确实如此。但有了这样的社区，也有了《Two Scoops》这样的资源，它比以往任何时候都更容易接近。我鼓励你加入我们。谁知道呢，也许你还会做出属于自己的贡献。

威廉・文森特（William Vincent）Django Software Foundation 董事会成员 LearnDjango.com 创始人

丹尼尔·费尔德罗伊的几句话

2006 年春天，我在 NASA 做一个项目，要实现一个基于 Java 的 RESTful Web 服务，而这个服务交付起来往往要花上好几周。一天晚上，等管理层都下班离开后，我用 Python 在 90 分钟内把这个服务重写了一遍。

那一刻我就知道，我想和 Python 一起工作。

我本想用 Django 来做这个 Web 服务的前端，但管理层坚持使用闭源技术栈，因为“Django 还只到 0.9x 版本，所以还没准备好用于真正的项目。”我并不同意，不过一想到至少核心架构用的是 Python，我也就释然了。那几年正是 Django 还带着几分锋芒的年代，它确实会吓到一些人。

很多年后，Django 已经被视为一个成熟、强大、安全、稳定的框架，世界各地许多极其成功的公司（OctopusEnergy、Instagram、Mozilla、Sentry 等）和政府机构（NASA、美国国会图书馆等）都在使用它。如今，要说服管理层用 Django 已经不难了；就算说服他们仍有难度，能让你用 Django 的工作机会也比过去好找得多。

我写这本书的目的，是把自己学到的东西分享给你。我的知识和经验，来自核心开发者给出的建议、我自己犯过的错误、与他人共享的成功经验，以及海量的笔记整理。我得承认，这本书是有鲜明立场的；但 Django 社区里很多领军人物，也在使用相同或相近的方法。

这本书是写给你们这些开发者的。希望你会喜欢它！

奥黛丽·费尔德罗伊的几句话

我第一次接触 Python，是在 2005 年 MIT 的一门研究生课程上。不到 4 周的作业时间里，每个学生都做出了一个语音控制系统，可以在 MIT 的 Stata Center 各个房间之间导航，运行环境是装着 Debian 的 HP iPaq。我当时被 Python 深深震撼，不明白为什么它不是拿来做一切事情的首选。我也试着用 Zope 做过一个 Web 应用，但过程很不顺利。

几年过去后，我被卷进了硅谷科技创业圈。我曾在一家创业公司里用 C 写图形库，也用 C++ 写桌面应用。后来有一段时间，我离开了那份工作，开始画画和做雕塑。很快，我一边为艺术展疯狂作画，一边联合策划一个 140 人参加的艺术展，还同时管理一连串房地产翻修项目。我意识到自己同时在做的事情太多，必须优化。自然而然地，我又转回到技术工作中，并重新找回了用 Python 工作的乐趣。[原文此处残缺，按上下文保守处理]

硅谷 Google App Engine、SuperHappyDevHouse 以及各种黑客松圈子里的许多朋友，都激励我投身 Django。通过他们，以及通过各种自由职业项目和合作关系，我逐渐认识到 Django 有多么强大。

不知不觉间，我已经在参加 2010 年的 PyCon，也正是在那里，我认识了我的丈夫 Daniel Feldroy。我们是在 James Bennett 的“Django In Depth”教程结束时相识的，而如今，我们人生中的这一章，又随着这本书的出版兜了一个完整的圈。

Django 给我的生活带来的快乐，远远超出我原本以为一个 Web 框架所能带来的程度。我写这本书的目标，是把那些关于 Django 开发常见实践、平时往往没人明说或只被默认存在的体会，尽量周到地交到你手里，好让你跨过常见障碍，也真正体验到使用 Django Web 框架做项目的乐趣。

引言

我们写这本书，是想把这些年来使用 Django 时学到的、那些从未真正写下来的技巧、窍门和常见做法，统统落到纸面上。

写作时，我们把自己当成了抄写员：把人们默认属于“常识”的那些东西，用简单的例子记录下来。

0.1 关于我们的建议

和 Django 官方文档一样，这本书也会讲在 Django 里该怎么做，并用代码示例展示各种情境。

但和 Django 官方文档不同的是，这本书会明确推荐某些编码风格、模式以及库的选择。Django 核心开发者或许会认同其中一些，甚至很多条；不过你要记住，我们的许多建议说到底只是建议而已，是我们多年使用 Django 之后形成的个人判断。

整本书里，我们会主张一些自己认为最好的实践和技巧。对于某些工具和库，我们也会坦率表达自己的个人偏好。

有时，我们会明确反对一些流行做法，认为它们属于反模式。对于大多数我们不认同的东西，我们都会尽量保持礼貌，也尊重作者付出的辛劳。只是极少数情况下，我们可能不会那么客气，因为我们是真心希望你避开那些危险的坑。

我们已经尽了最大努力，力求给出经过深思熟虑的建议，也尽量确保这些做法站得住脚。我们请来了自己极为尊敬的 Django 和 Python 核心开发者，对本书进行严厉得让人紧张的批评；技术审稿人的数量也比一般技术书更多；而我们自己更是在修订上投入了数不清的时间。即便如此，错误或疏漏仍然总有可能存在，也同样有可能出现比书中所写更好的实践。

我们真心实意地想持续迭代、持续改进这本书，这句话不是客套。如果你看到任何自己不认同的做法，或者发现哪些地方还可以做得更好，我们都真诚地希望你把改进建议发给我们。给我们反馈的最好方式，是在下面这个地址提交 issue：

github.com/feldroy/two-scoops-of-django-3.x/issues

如果你觉得哪里还能改进，请千万不要犹豫，直接告诉我们。我们会以建设性的态度接收这些反馈。

0.2 为什么叫《两勺 Django》?

和大多数人一样，写这本书的我们也爱吃冰淇淋。每到周六晚上，我们都会把谨慎抛到脑后，痛痛快快吃一顿冰淇淋。可别告诉别人，有时候不是周六晚上，我们也会来一点！

我们喜欢尝试新口味，也喜欢拿它们和老牌心头好相互比较。一路记录自己试过的这些口味，甚至围绕这件事建一个俱乐部，正好能成为一个很棒的 Django 示例项目。

当我们真的遇到一种特别喜欢的口味时，那种新口味会让我们脸上浮现笑意。读技术书时，碰上特别精彩的代码片段或建议，也是一样的感觉。我们写这本书的一个目标，就是把它写成那种能让读者露出“冰淇淋式微笑”的技术书。

最妙的是，冰淇淋类比还能帮我们想出更鲜活的代码示例。写这本书的过程，我们自己也玩得很开心。所以，如果你偶尔觉得我们在冰淇淋这件事上闹得有点过头，还请多包涵。

0.3 开始之前

这不是一本教程书。如果你是 Django 新手，这本书会对你有帮助，但其中很大一部分内容也会让你觉得有挑战。想要把这本书用到极致，你应该先理解 Python 这门编程语言，至少完整做过官方 Django 教程、Django

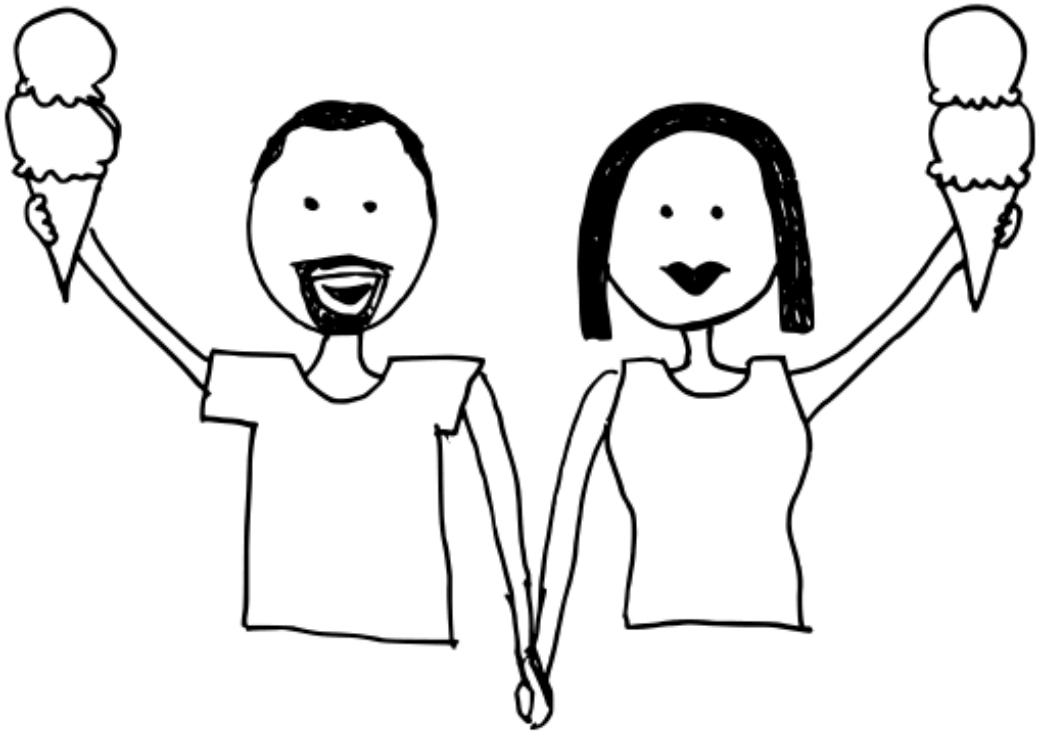

图 1: 图 1：把谨慎抛到脑后。

Crash Course 全套教程（feldroy.com/products/django-crash-course），或者其他类似的入门资源。有面向对象编程经验也会非常有帮助。

0.4 本书面向 Django 3.x 与 Python 3.8 或 3.9

本书和 Django 3.x 系列搭配得最好，和 Django 2.2 的契合度就会差一些，其他版本也是同理。我们并不保证代码层面的功能兼容性，但至少就整体思路而言，书中大多数方法在 Django 1.0 之后的各个版本里都依然站得住。

至于 Python 版本，本书是在 Python 3.8 上测试的。大多数代码示例在 Python 3.7.x 和 3.6.x 上也应该能运行。再往下的版本，就肯定会出问题了。

0.5 每一章都可以独立阅读

和那些每一章都建立在前一章项目之上的教程书、演练书不同，我们特意把这本书写成了每一章都能独立成立的样子。

这么做，是为了让你在做项目时，只要需要查某个特定主题，就能轻松翻到对应章节。

每一章里的示例都是完全独立的。它们并不是要拼成一个完整项目，也不是一套教程。把它们看作彼此隔离、但能帮助你理解各种编码场景的实用片段就好。

0.6 本书使用的约定

全书都会用到类似下面这样的代码示例：

示例 1：代码示例

Code (python)

```
class Scoop:
    def __init__(self):
        self._is_yummy = True
```

为了让这些代码片段尽量紧凑，我们有时会违背 PEP 8 中关于注释和行距的约定。代码样例可以在下面这个地址找到：

github.com/feldroy/two-scoops-of-django-3.x

书里还会出现下面这种特别的“别这样做!”代码块，用来展示你应该避开的糟糕代码：

示例 2：“别这样做!”代码示例

Code (python)

```
class Scoop:
    def __init__(self):
        self._is_yummy = False
```

本书采用下面这些排版约定：

- 等宽字体或带底纹的等宽字体，用于代码片段或命令。
- 斜体，用于文件名。
- 粗体，用于首次引入的新术语或重要词语。

书里还会用到装有注释、警告、提示和小趣闻的方框：

提示：有件事你应该知道

提示框会给出顺手好用的建议。

警告：某个危险陷阱

警告框会帮你避开常见错误和坑。

包提示：某个值得一试的实用包推荐

这类方框会说明与当前章节相关的有用第三方包，也会补充一些关于 Python、Django 和前端包的通用说明。我们还会整理出书中推荐包的完整清单。[原文此处后文残缺]

我们也会用表格把信息压缩成便于查阅、简洁清楚的形式：

0.7 核心理念

在构建 Django 项目时，我们会始终记着下面这些理念。

0.7.1 保持简单，别把事情搞复杂

大约 50 年前，航空史上最著名、产量也最高的飞机设计工程师之一凯利·约翰逊（Kelly Johnson）是这样说的。

项目	Daniel Feldroy	Audrey Feldroy
能吃椰子冰淇淋吗	不能	能
最近最爱的冰淇淋口味	南瓜	薄荷巧克力片

作者的冰淇淋偏好

在构建软件项目时，每多出一份不必要的复杂性，新增功能和维护旧功能就会更难。尽量尝试最简单的方案，但也要小心，不要为了“简单”而做出站不住脚的过度简化假设。这个理念有时会被缩写成“KISS”。

0.7.2 胖模型、工具模块、瘦视图、傻模板

决定一段代码该放在哪里时，我们喜欢遵循“胖模型、工具模块、瘦视图、傻模板”这一思路。

我们建议你把更多逻辑放进除了视图和模板之外的任何地方。这样做的结果很讨人喜欢：代码会更清楚，更像在自我说明，重复更少，可复用性也更强。至于模板标签和过滤器，它们应该只保留完成工作所必需的最少逻辑。

这方面我们会在下面这些地方进一步展开：

- 胖模型：6.7 节《理解胖模型》
- 工具模块：31.2 节《用工具模块优化应用》
- 瘦视图：8.5 节《尽量让业务逻辑远离视图》
- 傻模板 I：14.9 节《遵循极简主义做法》
- 傻模板 II：第 15 章《模板标签与过滤器》

0.7.3 默认先用 Django 原生方案

在考虑把 Django 核心组件替换成别的东西之前，比如替代模板引擎、不同的 ORM，或者非关系型数据库，我们会先试着用 Django 标准组件做一版实现。如果真的碰到了障碍，我们也会先把各种可能性都探索一遍，再决定是否替换核心组件。

参见第 20 章《替换核心组件的取舍》。

0.7.4 熟悉 Django 的设计哲学

隔一段时间读一读 Django 的设计哲学文档，是件很有价值的事，因为它能帮我们理解：为什么 Django 会提供某些约束，也会提供某些工具。和任何框架一样，Django 不只是一个“拿来写视图”的工具，它更是一整套做事方法，目的是让我们能在合理时间内拼出可维护的项目。

参考：docs.djangoproject.com/en/3.2/misc/design-philosophies/

0.7.5 十二要素应用

“十二要素应用”（Twelve-Factor App）是一套完整的 Web 应用设计方法，如今在许多资深 Django 开发者和核心开发者中越来越受欢迎。它是一种关于如何构建可部署、可扩展应用的方法论，值得一读，也值得认真理解。它的某些部分与《两勺 Django》所倡导的实践高度一致，我们也乐于把它推荐给所有做 Web 应用开发的人。

参见：12factor.net

0.8 我们的写作理念

写这本书时，我们想给读者，也想给我们自己，尽可能拿出最好的材料。为此，我们采用了下面这些原则。

0.8.1 提供最好的内容

我们已经尽了最大努力，想把最好的内容交到你手里。书里每一个主题，我们都会去找已知的可靠资料来核对内容；我们也从不害怕提问。接着，我们把专家们的文章、回复和建议提炼成你现在在书里看到的内容。要是这些还不够，我们就自己拿出方案，再找不同领域的专家帮忙审视。整个过程工作量极大，我们也真心希望你会满意最后的结果。

如果你想知道这一版（Django 3.x）和上一版（Django 1.11）之间有哪些不同，可以在下面这个地址看到一份变更短清单：

github.com/feldroy/two-scoops-of-django-3.x/blob/master/changelog

0.8.2 站在巨人的肩膀上

虽然这本书由我们负责，也由我们署名，但书里写到的所有实践，当然并不都是我们自己凭空想出来的。

如果没有 Django、Python 以及更广义开源软件社区里那些才华横溢、富有创造力又慷慨无私的开发者，这本书根本不会存在。我们非常相信：那些当过我们老师和导师的人，那些为我们提供过信息来源的人，都应该被郑重地承认。只要能给到的地方，我们都尽力把 credit 还给该得的人。

0.8.3 倾听读者和审稿人的声音

在本书前几版中，我们收到了海量反馈，真的可以说是来自一整支“军团”般的读者与审稿人。正是这些反馈，让我们得以大幅提升本书的质量。

现在，它已经达到了我们曾经期望、却从未真正敢相信自己能做到的水准。

作为回报，我们在书后分享了署名，也一直在持续寻找“把这份善意继续传下去”的方式，希望借此改善世界各地开发者的生活。

如果你对这一版有任何问题、意见或其他反馈，欢迎通过我们的 issue tracker 告诉我们：

github.com/feldroy/two-scoops-of-django-3.x/issues

另外，在本书结尾还有一个链接，可以让你在 Amazon 上为《Two Scoops of Django》留下评论。这样也能帮助其他人判断，这本书究竟适不适合他们。

0.8.4 发布问题与勘误

没有任何东西是真正完美的，再多轮审阅也不例外。我们会在《Two Scoops of Django 3.x》的 GitHub 仓库中持续发布问题与勘误：

github.com/feldroy/two-scoops-of-django-3.x

第一章 编码风格

稍微花一点心思遵循标准的编码风格规范，长期回报都会非常可观。哪怕你很想跳过这一章，我们也强烈建议你认真读完。

1.1 让代码易读的重要性

> 程序首先是写给人读的，至于机器去执行，那只是附带结果。
>
> 《计算机程序的构造和解释》, Abelson 与 Sussman

代码被阅读的次数，远远多于它被写下的次数。写出一小段代码，可能只需要几分钟；调试它，也许要几小时；而它之后往往会长久地躺在那里，许多年都不再被碰触。真正考验编码风格的时刻，是你或别人回头再看昨天、去年，甚至十年前写下的代码时。只有当代码写得清楚、风格一致，它的价值才会真正显现出来。容易理解的代码，能把人的脑力从“猜这里为什么和别处不一样”这种无谓消耗中解放出来，让各种规模的项目都更容易维护和演进。

这意味着，你应该多走一步，让代码尽可能易读：

- 不要把变量名缩写得过头。
- 函数参数名尽量写完整。
- 为类和方法补上文档。
- 给代码写注释。
- 把重复的代码提炼成可复用的函数或方法。
- 让函数和方法尽量短。一个很好用的经验法则是：阅读完整个函数或方法时，不需要滚动屏幕。

当你隔了一段时间再回来看自己的代码时，这样写会让你更容易接上之前的思路。

拿那些烦人的缩写变量名来说吧。看到 `balance_sheet_decrease` 这样的变量时，你的大脑显然比面对 `bsd` 或 `bal_s_d` 之类的缩写更容易立刻明白它的意思。这类偷懒也许能省下几秒钟打字时间，但换来的却可能是数小时乃至数天的技术债。完全不值得。

1.2 PEP 8

PEP 8 是 Python 的官方风格指南。我们建议你认真细读，并学会遵循 PEP 8 中的编码约定：

python.org/dev/peps/pep-0008/

PEP 8 里会说明诸如下面这样的编码约定：

- “每一级缩进使用 4 个空格。”
- “顶层函数和类定义之间空两行。”
- “类内部的方法定义之间空一行。”

你在 Django 项目里写的所有 Python 文件，都应该遵循 PEP 8。要是你记不住这些规则，就去给自己的代码编辑器装一个插件，让它在你输入时实时检查代码。

当一位有经验的 Python 程序员在 Django 项目里看到严重违背 PEP 8 的代码时，就算嘴上不说，心里多半也已经在默默吐槽了。相信我们。

警告：不要修改一个既有项目原本的约定

PEP 8 的风格主要适用于新的 Django 项目。如果你是中途加入一个已经存在的 Django 项目，而这个项目遵循的约定与 PEP 8 不同，那就应该服从项目既有的约定。

关于这一点，以及其他为什么需要打破规则的原因，请阅读 PEP 8 里的“A Foolish Consistency is the Hobgoblin of Little Minds”一节：

- python.org/dev/peps/pep-0008/#a-foolish-consistency-is-the-hobgoblin-of-little-minds

包提示：用 Black 格式化 Python 代码

Black 是 Łukasz Langa 创建的一款“毫不妥协”的 Python 代码格式化工具。使用它，意味着我们放弃对那些需要手工精雕细琢的格式细节的控制；但作为回报，它给我们带来速度，也给我们带来代码最终长什么样的确定性。我们非常喜欢它，因为它把我们从没完没了地调格式这件事里解放出来，好让我们把精力放在真正要做的工作上。我们会把它接进自己的代码提交流程中。更多信息见：

github.com/psf/black

包提示：用 Flake8 检查代码质量

Flake8 最初由 Tarek Ziadé 创建，如今由 PyCQA 团队维护。它是一款非常有用的命令行工具，可以检查项目中的编码风格、代码质量和逻辑错误。我们建议你在本地开发时使用它，也把它纳入持续集成流程中。

参见：

github.com/PyCQA/flake8

1.2.1 79 字符限制

> 不是开玩笑，我到现在还经常要和只能显示 80 个字符宽度的控制台打交道。
>
> Barry Morrison, Systems Engineer，也是《Two Scoops of Django》的技术审稿人

根据 PEP 8，每行文本长度上限是 79 个字符。这个限制存在的原因是：它是一个足够稳妥的数值，大多数带自动换行的编辑器和开发团队都能轻松适配，而不会损害代码的可理解性。

不过，PEP 8 也允许在“封闭团队项目”里把这一限制放宽到 99 个字符。我们把这里理解为：不对外开源的项目。

我们的偏好如下：

- 对开源项目，应该严格执行 79 字符上限。根据我们的经验，贡献者或访客确实会嘀咕行宽限制这件事；但它并没有把贡献者挡在门外，而且我们认为这条规则依然很有价值。
- 对私有项目，我们会把限制放宽到 99 个字符，充分利用现代显示器的宽度。

请阅读：

python.org/dev/peps/pep-0008/#maximum-line-length

提示：Aymeric Augustin 谈行宽问题

Django 核心开发者 Aymeric Augustin 说过：“把代码硬塞进 79 列，绝不是给变量、函数或类起更差名字的好理由。比起去迎合三十年前硬件的任意限制，可读的变量名重要得多。”

1.3 关于 import

PEP 8 建议把 import 按下面的顺序分组：

1. 标准库 import
2. 相关的第三方 import
3. 本地应用或特定库的 import

我们在 Django 项目里写 import 时，大概会长成下面这样：

示例 1.1：良好的 Python import 写法

Code (python)

```
# 标准库 import
from math import sqrt
from os.path import abspath

# Django 核心 import
```

```
from django.db import models
from django.utils.translation import gettext_lazy as _

# 第三方应用 import
from django_extensions.db.models import TimeStampedModel

# 你自己的应用 import
from splits.models import BananaSplit
```

（注意：你其实不需要像示例里这样给 import 写注释。这里的注释只是为了说明示例结构。）

Django 项目里的 import 顺序如下：

1. 标准库 import。
2. Django 核心 import。
3. 第三方应用 import，包括那些和 Django 无直接关系的库。
4. 你自己在 Django 项目中创建的应用 import。（关于 app，你会在第 4 章《Django 应用设计基础》中读到更多内容。）

包提示：用 isort 排序 import

`isort` 这个 Python 库能替我们自动整理 import。它会按字母顺序排序，并自动把不同类型的 import 分到不同分组里。

1.4 理解显式相对导入

写代码时，一个非常重要的原则是：代码应该尽量容易移动、重命名和做版本演化。在 Python 里，显式相对导入是一个很强大的工具，它可以让单个模块不必和周围架构绑得过紧。Django app 本质上就是 Python 包，所以同样的规则完全适用。

为了说明显式相对导入的好处，我们来看一个例子。假设下面这段代码出自一个你写的 Django 项目，这个项目专门用来追踪你吃过多少冰淇淋，连所有华夫脆筒、砂糖脆筒和蛋糕筒都要一并记录。

示例 1.2：相对 Python import

Code (python)

```
# cones/views.py
from django.views.generic import CreateView

# 来自 'cones' 包的相对导入
from .models import WaffleCone
from .forms import WaffleConeForm

# 来自 'core' 包的绝对导入
from core.views import FoodMixin


class WaffleConeCreateView(FoodMixin, CreateView):
    model = WaffleCone
    form_class = WaffleConeForm
```

只要理解了绝对导入和显式相对导入的区别，我们就能立刻看出哪些 import 属于本地/内部，哪些属于全局/外部，也更能突出“Python 包本身就是一个代码单元”这一点。

总结一下，下面这张表概括了 Django 项目里不同类型 Python import 的用法：

代码	导入类型	适用场景
`from core.views import FoodMixin`	绝对导入	当你导入当前 app 之外的内容时使用
`from .models import WaffleCone`	显式相对导入	当你导入当前 app 内另一个模块的内容时使用

表 1.1：Import：绝对导入 vs. 显式相对导入

养成使用显式相对导入的习惯吧。这件事做起来非常容易，而且对任何 Python 程序员来说，都是值得培养的好习惯。

提示：PEP 328 会不会和 PEP 8 冲突？

看看 Python 的 BDFL Guido van Rossum 是怎么说的：

- python.org/pipermail/python-dev/2010-October/104476.html

延伸阅读：

python.org/dev/peps/pep-0328/

1.5 避免使用 import *

在我们 99% 的工作里，都会显式导入每一个模块：

示例 1.3：显式 Python import

Code (python)

```
from django import forms
from django.db import models
```

永远不要这样写：

示例 1.4：import *

Code (python)

```
# 反模式：别这样做！
from django.forms import *
from django.db.models import *
```

原因在于，这样做会把另一个 Python 模块里的所有局部名称隐式加载进当前模块的命名空间里，甚至覆盖已有名称。这会带来不可预测、而且有时是灾难性的后果。

这个规则有一个特定例外，我们会在第 5 章《设置与 requirements 文件》里讲到。

来看上面的坏代码例子。Django 的 `forms` 库和 `models` 库里都定义了一个叫 `CharField` 的类。由于两个库都被隐式整包导入，`models` 里的那个类就会把 `forms` 版本覆盖掉。类似的问题也会出现在 Python 内建库，或其他第三方库之间，导致关键功能被悄悄覆盖。

使用 `import *`，就像冰淇淋店里那种贪心顾客：向店员要了三十一种口味每一种的一勺免费试吃，最后却只买一两勺。你明明只会用到一两样东西，就别把所有内容都 import 进来。

当然，如果那位顾客最后真的端着一大碗几乎装满所有口味的冰淇淋离开，那又另当别论。

1.5.1 其他 Python 命名冲突

如果你试图导入两个同名对象，也会碰到类似的问题，例如：

示例 1.5：Python 模块名冲突

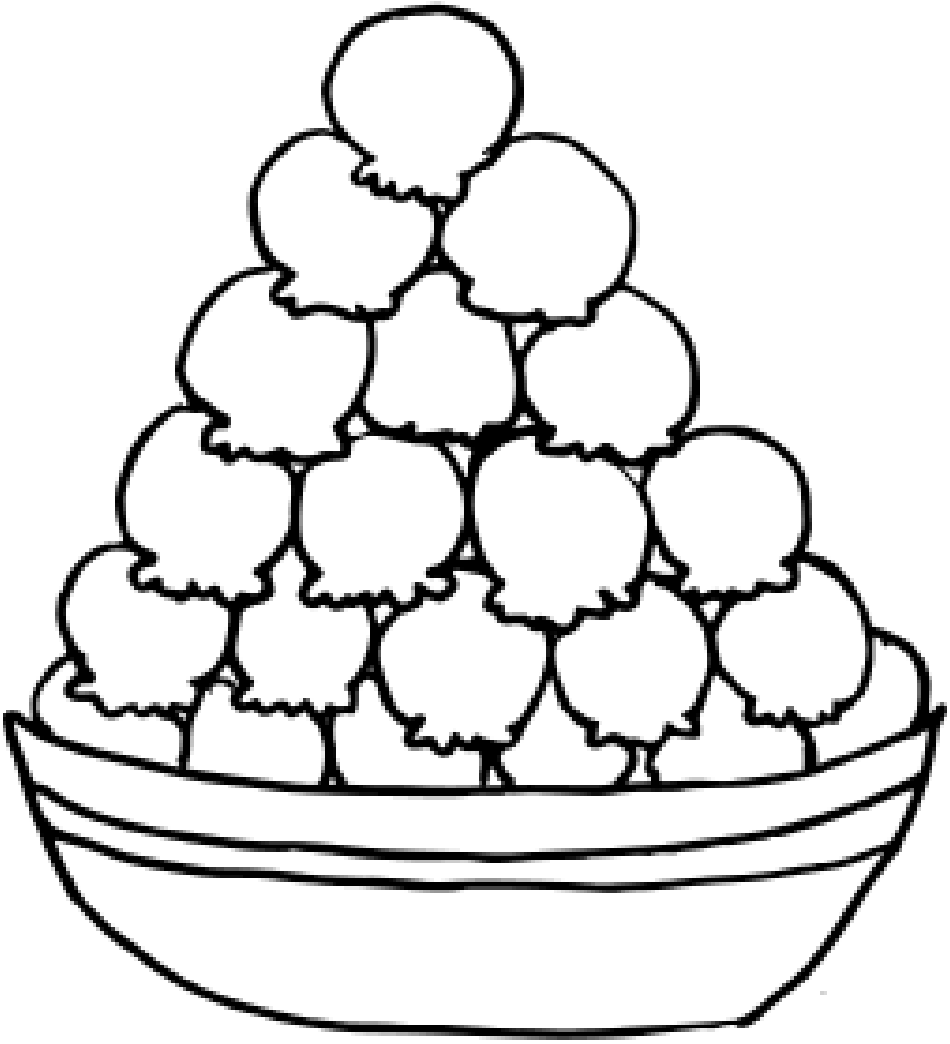

图 1.1: 图 1.1：在冰淇淋店里使用 import *

Code (python)

```
# 反模式：别这样做！
from django.db.models import CharField
from django.forms import CharField
```

如果你确实要处理这种命名冲突，完全可以用别名来解决：

示例 1.6：用别名避免 Python 模块命名冲突

Code (python)

```
from django.db.models import CharField as ModelCharField
from django.forms import CharField as FormCharField
```

1.6 Django 的编码风格

这一节既涵盖官方指南，也包括 Django 社区里虽不成文、但已经广泛接受的惯例。

1.6.1 了解 Django 编码风格指南

不用多说，熟悉常见的 Django 风格约定一定是好事。事实上，Django 内部也有自己的一套风格指南，它是在 PEP 8 基础上的扩展：

- docs.djangoproject.com/en/3.2/internals/contributing/writing-code/coding-style/

另外，下面有些内容虽然没有明确写进官方标准，但在 Django 社区里已经足够普遍，因此你大概率也会想在自己的项目里遵循它们。

提示：去读读 Django internals 文档

Django internals 文档里远不只有编码风格。里面还塞满了很多有用信息，包括 Django 项目的历史、发布流程等等。我们建议你去看看。

docs.djangoproject.com/en/3.2/internals/

1.6.2 URL 模式名里用下划线，不用短横线

我们总是尽量使用下划线（_），而不是短横线。这么做不只是更符合 Python 风格，也对更多 IDE 和文本编辑器更友好。注意，这里说的是 `url()` 的 `name` 参数，而不是浏览器地址栏里真正输入的 URL。

错误示例：在 URL 名称里使用短横线

示例 1.7：不好的 URL 模式名

Code (python)

```
patterns = [
    path(route='add/', view=views.add_topping, name='add-topping'),
]
```

正确示例：在 URL 名称里使用下划线

示例 1.8：好的 URL 模式名

Code (python)

```
patterns = [
    path(route='add/', view=views.add_topping, name='toppings:add_topping'),
]
```

至于真实 URL 里用短横线完全没问题，例如 `route='add-topping/'`。

1.6.3 模板 block 名里也用下划线，不用短横线

原因和 URL 模式名一样：定义模板 block 名称时，我们也推荐使用下划线。这样更符合 Python 风格，也对编辑器更友好。

1.7 选择 JS、HTML 和 CSS 风格指南

1.7.1 JavaScript 风格指南

和 Python 只有一份官方风格指南不同，JavaScript 没有官方统一标准。取而代之的是，不同个人和公司分别提出了不少非官方 JS 风格指南：

- Standard, 同时适用于 JavaScript 和 Node.js 的风格指南：`github.com/standard/standard`
- idiomatic.js: Principles of Writing Consistent, Idiomatic JavaScript：`github.com/rwaldron/idiomatic.js`
- Airbnb JavaScript Style Guide：`github.com/airbnb/javascript`

Django 社区和 JavaScript 社区都没有在这些指南中形成统一共识，所以你挑一个自己喜欢的，坚持下去就行。

不过，如果你使用的是某个自带风格指南的 JavaScript 框架，那就应该遵循它自己的规则。比如 `ember.js` 就有自己的风格指南。

包提示：ESLint，可插拔的 JavaScript 和 JSX lint 工具

ESLint（eslint.org）是一个用来检查 JavaScript 和 JSX 代码风格的工具。它提供了多个风格指南的预设规则，其中就包括上面列出的几种。很多文本编辑器都有 ESLint 插件，Webpack、Gulp、Grunt 等 JavaScript 工具链里也有对应任务可以接入。

1.7.2 HTML 和 CSS 风格指南

- Code Guide by @mdo for HTML and CSS：`codeguide.co`
- idiomatic-css: Principles of Writing Consistent, Idiomatic CSS：`github.com/necolas/idiomatic-css`

包提示：stylelint

Stylelint（stylelint.io）是一个 CSS 编码风格检查工具。它会按照你配置的规则检查一致性，也会检查 CSS 属性的排序方式。和 ESLint 一样，stylelint 也有文本编辑器插件，以及任务/构建工具插件。

1.8 永远不要为集成开发环境（IDE）或文本编辑器而写代码

有些开发者会根据自己所用集成开发环境（IDE）的功能来决定项目布局和实现方式。这会让那些使用不同开发工具的人很难顺利发现、理解项目代码。

你应该永远默认：身边的开发者都喜欢用自己的工具，而你的代码和项目布局必须足够透明，即使某个人只能用 Notepad 或 Nano，也依然能顺利读懂并浏览你的工作成果。

举个例子，如果开发者用的不是那极少数特定 IDE，要追踪模板标签、或者找到它们的来源代码，往往会很困难，也很耗时。因此，我们遵循一个社区里广泛使用的命名模式：`<app_name>_tags.py`。

1.9 小结

这一章介绍了我们偏好的编码风格，也解释了我们为什么偏好这些做法。

即使你不采用我们使用的这一套编码风格，也请至少坚持使用一种前后一致的风格。风格混杂的项目会难维护得多，开发速度会变慢，开发者犯错的概率也会随之上升。

第二章 最优的 Django 环境配置

这一章会介绍我们认为最适合中高级 Django 开发者的本地开发环境配置方式。

2.1 在所有环境里使用同一种数据库引擎

开发者常见的一个坑，是本地开发用 SQLite3，而生产环境用 PostgreSQL 或 MySQL。这一节不仅适用于“SQLite3 + PostgreSQL”这种组合，也适用于任何“开发环境和生产环境用了两套不同数据库，却还指望它们行为完全一致”的场景。

下面这些问题，都是我们在开发环境和生产环境使用不同数据库引擎时亲自踩过的：

2.1.1 你无法在本地检查一份和生产环境完全一致的数据副本

如果你的生产数据库和本地开发数据库不同，你就没法直接抓一份与生产环境完全一致的数据库副本到本地分析数据。

当然，你可以从生产环境导出一份 SQL dump，再导入本地数据库，但这并不意味着导出再导入之后，你得到的依然是一份“完全一模一样”的副本。

2.1.2 不同数据库的字段类型和约束并不相同

你要牢记：不同数据库处理字段数据类型转换的方式并不一样。Django 的 ORM 确实会努力去抹平这些差异，但它能做的也终究有限。

例如，有些人本地开发用 SQLite3，生产环境用 PostgreSQL，还以为 Django ORM 足以让他们不必认真面对两者之间的差别。可迟早，他们还是会撞上问题，因为 SQLite3 使用的是动态、弱类型，而不是强类型。

没错，Django ORM 的确提供了一些能力，让你的代码在和 SQLite3 打交道时也能表现得“更像强类型”；但在开发阶段，哪怕跑测试，也依然可能放过表单和模型校验方面的错误，直到代码真正部署到生产服务器上才暴露出来。比如，你可能在本地毫无阻碍地保存超长字符串，因为 SQLite3 根本不在意；可一到生产环境，你的 PostgreSQL 或 MySQL 就会直接抛出约束错误，而这些错误你在本地从未见过。等到那时，你往往还得先在本地搭出一套完全相同的数据库，才能艰难地复现问题。

大多数这类问题，通常只有在项目运行在强类型数据库（例如 PostgreSQL 或 MySQL）上时才会暴露出来。一旦这种 bug 真撞上来，你最后往往只能一边后悔，一边手忙脚乱地把本地开发机改造成正确的数据库环境。

提示：Django + PostgreSQL 很能打

我们认识的大多数 Django 开发者，都更喜欢在所有环境里统一使用 PostgreSQL：开发、预发布、QA 和生产环境全都如此。

你可以根据自己的操作系统，参考下面这些安装方式：

- Mac：从 `postgresapp.com` 下载一键安装器
- Windows：从 `postgresql.org/download/windows/` 下载一键安装器
- Linux：通过系统包管理器安装，或者参照 `postgresql.org/download/linux/` 上的说明

在某些操作系统上，PostgreSQL 本地跑起来确实要多费些功夫，但我们觉得，这份投入非常值得。

2.1.3 Fixture 不是魔法解法

你可能会问：既然本地和生产环境数据库不同，为什么不能直接用 fixture 来把这些差异抽象掉？

问题在于，fixture 的确很适合创建简单、硬编码的测试数据集。有时候，在项目早期开发阶段，你确实需要提前往数据库里灌一些假的测试数据。

但 fixture 并不是一个可靠的、可以用数据库无关方式把大规模数据集从一种数据库迁移到另一种数据库的工具。它压根不是为这个用途设计的。不要把 fixture 能做基础数据创建（dumpdata/loaddata）这件事，误认为它也有能力在不同数据库工具之间迁移生产数据。

警告：不要在生产环境里用 SQLite3 跑 Django

只要你的 Web 项目不止一个用户，或者对并发要求不只是“轻量级”那么简单，那么 SQLite3 就是在为灾难埋雷。说得最直白一点：SQLite3 在生产环境里看上去一直很好用，直到它突然不好用为止。我们自己就经历过，也听过别人讲过太多惨案。

真正糟糕的是，一旦问题出现，再把数据从 SQLite3 迁到原本就为并发而设计的数据库（例如 PostgreSQL）里，过程本身也会相当困难且复杂。

我们知道，网上确实有不少文章主张在生产环境里使用 SQLite3。但即便有一小撮对 SQLite3 非常精通的高手，在某些极端边缘场景下能把它用得很好，这也不足以成为你在 Django 生产环境里照搬这套做法的理由。

2.2 使用 pip 和（virtualenv 或 venv）

如果你现在还没有这样做，我们强烈建议你尽快熟悉 pip 和 virtualenv。它们已经是 Django 项目的事实标准，大多数使用 Django 的公司也都依赖这两样工具。

pip 是一个从 Python Package Index 及其镜像站抓取 Python 包的工具。它负责管理和安装 Python 包，并且在大多数非 Linux 的 Python 安装中都会默认附带。

virtualenv 是一个用来创建隔离 Python 环境、维护依赖关系的工具。如果你同时维护多个项目，而且这些项目所依赖的库版本彼此冲突，它就会非常有用。

举个例子，假设你正在做两个项目，其中一个需要 Django 3.0，另一个需要 Django 3.2。

- 如果没有 virtualenv（或者其他依赖管理工具），那你每切换一次项目，就得重新安装一次 Django。
- 如果你觉得这听起来已经够折腾了，请记住：大多数真实 Django 项目，通常至少都有十几个依赖要维护。

pip 已经内置在 Python 3.4 及以上版本中。更多阅读材料和安装说明见：

- pip: pip.pypa.io
- virtualenv: virtualenv.pypa.io

包提示：virtualenv 的替代品 Conda

Conda 是一个开源的包管理系统和环境管理系统，支持 Windows、Mac 和 Linux。它起源于数据科学社区。在各种打包环境里，对于想在 Windows 平台上安装编译后二进制包的用户来说，它算是最好上手的一种。Conda 也让多 Python 版本管理变得几乎毫不费力。目前这是 Daniel 最偏爱的开发环境方案，他会把 Conda 和 pip 搭配起来管理依赖。

更多信息：

docs.conda.io/

包提示：pip + virtualenv 的替代品 Poetry 和 Pipenv

Poetry 能帮助你声明、管理并安装 Python 项目的依赖，确保你在所有地方都能拿到一致的依赖栈。我们很欣赏它把这么多功能优雅地封装进一个直观 CLI 的方式。它如今已经足够流行、也足够稳定，可以作为我们愿意推荐的依赖管理平台。

更多信息：

python-poetry.org

Pipenv 则把 pip 和 virtualenv 包装进了一个统一接口里。它会自动处理创建环境之类的流程，并引入 Pipfile.lock 这样的文件，从而支持可重复、确定性的构建。

更多信息：

pipenv.pypa.io/

警告：在很多 Linux 发行版里，pip 不一定默认安装

不少 Linux 发行版的维护者，会把 Python 标准库中的某些模块拆出来，改成独立安装的软件包。比如在 Ubuntu 上，你就必须自行安装 python3-pip、python3-setuptools、python3-wheel 和 python3-distutils。

随着 Windows 10 后续稳定提供 WSL2（Windows 上的 Linux），这个模式也可能以类似方式影响到这部分用户。

2.2.1 virtualenvwrapper

我们还强烈推荐：Mac 和 Linux 上使用 virtualenvwrapper，Windows 上使用 virtualenvwrapper-win。这个项目最初由 Doug Hellman 发起。

说实话，我们觉得单独使用 virtualenv 往往会让人有点痛苦，因为每次想激活一个虚拟环境时，你都得敲一长串命令，比如：

示例 2.1：激活 virtualenv

Code (bash)

```
$ source ~/.virtualenvs/twoscoops/bin/activate
```

而用了 virtualenvwrapper 之后，你只需要输入：

示例 2.2：激活 virtualenv

Code (bash)

```
$ workon twoscoops
```

virtualenvwrapper 是 pip 和 virtualenv 的一个流行搭档，确实能让生活轻松很多，但它并不是绝对必要的。

2.3 通过 pip 安装 Django 和其他依赖

Django 官方文档里描述了好几种安装 Django 的方式。我们推荐的方法，是使用 pip 加 requirements 文件。

简单来说，requirements 文件就像一张 Python 包购物清单：里面列出你想安装的包名，以及可选的版本范围。你再用 pip 按这张清单，把包安装进当前虚拟环境中。

关于 requirements 文件的组织方式，以及如何通过它们完成安装，我们会在第 5 章《设置与 requirements 文件》中详细展开。

提示：设置 PYTHONPATH

如果你对命令行和环境变量已经相当熟悉，你可以给虚拟环境设置 PYTHONPATH，这样就能使用 django-admin 来启动站点和执行其他任务。

你也可以把虚拟环境的 PYTHONPATH 配到包含当前目录的位置，并配合最新版 pip 使用。从项目根目录执行 pip install -e . 就可以做到这一点：它会把当前目录作为一个可原地编辑的包安装进去。

如果你不知道该怎么设置，或者觉得这件事有点复杂，那也完全不用担心，老老实实继续用 manage.py 就好。

延伸阅读：

- hope.simons-rock.edu/~pshields/cs/python/pythonpath.html
- docs.djangoproject.com/en/3.2/ref/django-admin/

2.4 使用 Git 做版本控制

版本控制系统也常被称为 revision control 或 source control。只要你在做任何 Django 项目，就应该使用版本控制系统来追踪代码改动。

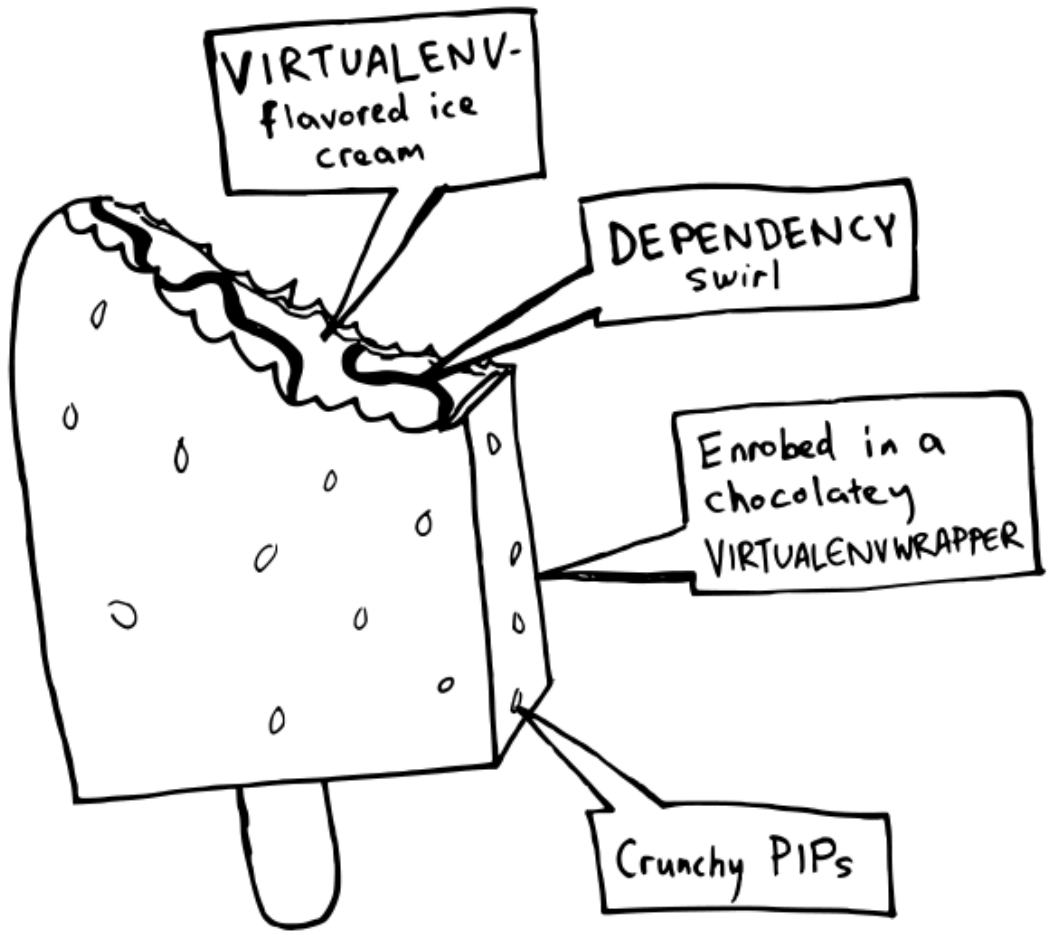

图 2.1: 图 2.1：以冰淇淋吧形式出现的 `pip`、`virtualenv` 和 `virtualenvwrapper`

Git 已经是跨语言、跨工具的行业标准。它让创建分支和合并改动都变得非常容易。使用版本控制系统时，重要的不只是本地保留一份仓库副本，也要用代码托管服务做备份。幸运的是，现在有很多服务和工具都能托管代码仓库。在这些选择里，我们推荐 GitHub（`github.com`）或 GitLab（`gitlab.com`）。

2.5 可选方案：完全一致的环境

“在程序员笔记本上能跑”的东西，未必能在生产环境里跑。但如果你的本地开发环境，和项目的预发布、测试、生产环境几乎一模一样呢？

当然，如果你的生产基础设施由一万台服务器组成，那你不可能真的再配出一万台本地服务器来做开发。所以我们说的“一致”，意思其实是“在现实条件允许下尽量一致”。

下面这些环境差异，是你有机会消除掉的：

- **操作系统差异**。如果你在 Mac 或 Windows 上开发，而站点部署在 Ubuntu Linux 上，那么本地和生产环境之间在系统层面会有非常大的差别。
- **Python 安装差异**。说句实话，很多开发者和运维甚至都搞不清自己本地到底跑的是哪个 Python 版本，只是没人愿意承认。为什么？因为把 Python 环境真正配明白，并完全理解自己的配置，并不是件轻松事。
- **开发者之间的差异**。在大型团队里，大量时间都会浪费在排查“为什么甲的环境能跑、乙的环境却不行”这种问题上。

如今最常见的“搭一套一致开发环境”的方法，就是 Docker。

2.5.1 Docker

在写这本书的时候，Docker 已经是容器化环境的行业标准了。它在各个操作系统上都有很好的支持，包括 Microsoft Windows。和 Docker 打交道，有点像在虚拟机里开发，只不过更轻量。Docker 容器共享宿主操作系统，但拥有自己隔离的进程空间和内存空间。除此之外，由于 Docker 使用的是支持 union 的文件系统，容器构建时通常只需要在一个快照基础上叠加差异，而不必每次都从零开始重新构建，所以速度会快得多。

对于本地开发来说，它最大的好处在于：要搭出一套与开发环境和生产环境高度接近的环境，难度会小很多。

例如，如果我们的开发笔记本跑的是 macOS、Windows、CentOS 等系统，而某个项目的配置却强依赖 Ubuntu，那么我们就可以借助 Docker 和 Docker Compose，很快在本地搭起一套虚拟的 Ubuntu 开发环境，里面把项目需要的包和配置都准备齐全。这样我们就可以：

- 给项目开发团队里的每个人都准备一套一致的本地开发环境。
- 让这些本地环境在配置方式上尽量接近预发布、测试和生产服务器。

它的潜在缺点则包括：

- **额外复杂度**。在很多场景里，这份复杂度其实并不必要。对一些更简单的项目来说，如果我们并不怎么担心操作系统层面的差异，那跳过它会更轻松。
- **性能开销**。在比较老的开发机器上，哪怕只是运行轻量级容器，也可能把性能拖得非常慢。即便是较新的机器，也多少会引入一些虽小但能感觉出来的开销。

关于用 Docker 做开发，可以参考：

- `cookiecutter-django.readthedocs.io/en/latest/developing-locally-docker.html`
- `http://bit.ly/1dWnzVW`，Real Python 关于 Django 和 Docker Compose 的文章
- `dockerbook.com`

2.6 小结

这一章讨论了：在开发环境和生产环境中使用同一种数据库、`pip`、`virtualenv`、`venv`、Conda、Poetry、Pipenv、版本控制以及 Docker。把这些工具收入你的工具箱绝对是值得的，因为它们不仅常见于 Django 开发，也同样广泛存在于绝大多数 Python 软件开发工作中。

第三章 如何组织 Django 项目结构

项目结构这一块，一直是 Django 核心开发者之间都意见不完全一致的话题之一。对于什么才算最佳实践，大家各有判断。在这一章里，我们会介绍我们自己的做法，它也是目前最常见的方案之一。

包提示：Django 项目模板

市面上已经有不少项目模板，可以非常有效地为 Django 项目起步提速，而且它们也遵循了本章所描述的许多模式。下面这些链接，在我们搭项目骨架时往往都很有参考价值：

- `github.com/pydanny/cookiecutter-django`

  本章会重点提到它。
- `github.com/grantmcconnaughey/cookiecutter-django-vue-graphql-aws`

  本章也会提到它。
- `djangopackages.org/grids/g/cookiecutters/`

  一份可选 Cookiecutter 模板清单。

3.1 Django 3 默认生成的项目结构

先来看看，当你运行 `startproject` 和 `startapp` 时，Django 默认会生成怎样的项目结构：

示例 3.1：默认的 `startproject` 和 `startapp`

Code (bash)

```
django-admin startproject mysite
cd mysite
django-admin startapp my_app
```

最终得到的项目结构大致如下：

示例 3.2：默认项目结构

Code (text)

```
mysite/
+-- manage.py
+-- my_app
|   +-- __init__.py
|   +-- admin.py
|   +-- apps.py
|   +-- migrations
|   |   +-- __init__.py
|   +-- models.py
|   +-- tests.py
|   +-- views.py
+-- mysite
    +-- __init__.py
    +-- asgi.py
    +-- settings.py
    +-- urls.py
    +-- wsgi.py
```

Django 默认的项目结构存在不少问题。拿来做教程当然很方便，但一旦你开始真正搭建一个实际项目，它就没那么顺手了。接下来这一章会解释原因。

3.2 我们偏好的项目结构

我们采用的是一种经过调整的“两层式”方案，它建立在 `django-admin startproject` 生成结果的基础上。最高层级的布局大致如下：

示例 3.3：项目根层级

Code (text)

```
<repository_root>/
+-- <configuration_root>/
+-- <django_project_root>/
```

下面我们逐层来看。

3.2.1 顶层：仓库根目录

`<repository_root>` 目录是整个项目的绝对根目录。除了 `<django_project_root>` 和 `<configuration_root>` 之外，我们还会把其他关键组件放在这里，比如 `README.md`、`docs/` 目录、`manage.py`、`.gitignore`、`requirements.txt` 文件，以及部署和运行项目所需的其他高层级文件。

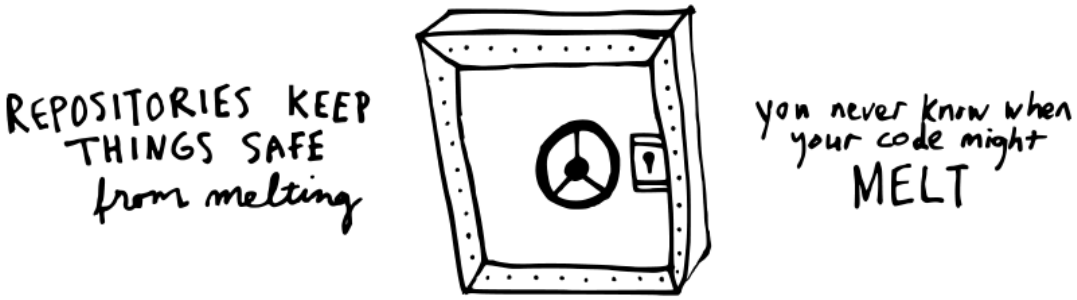

图 3.1: 图 3.1：仓库之所以重要的又一个理由。

提示：这一点上，常见做法并不统一

有些开发者喜欢把 `<django_project_root>` 直接并入项目的 `<repository_root>` 中。

3.2.2 第二层：Django 项目根目录

`<django_project_root>/` 是真正 Django 项目的根目录。所有非配置类的 Python 代码文件，都应该放在这个目录或它的下级目录里。

如果你使用的是 `django-admin startproject`，通常会在仓库根目录中执行这个命令。这样它生成出来的 Django 项目目录，就会成为这个项目根目录。

3.2.3 第二层：配置根目录

`<configuration_root>` 目录用于放置 settings 模块和基础 URLConf（`urls.py`）。它必须是一个合法的 Python 包，因此里面一定要包含一个 `__init__.py` 模块。即使在 Python 3 里，如果没有这个 `__init__.py`，`<configuration_root>` 也不会被识别成 Python 包。

如果你使用的是 `django-admin startproject`，这个配置根目录一开始其实是在 Django 项目根目录内部的。你应该把它移到仓库根目录下。

配置根目录里的这些文件，本来就是 `django-admin startproject` 命令生成出来的那一部分内容。

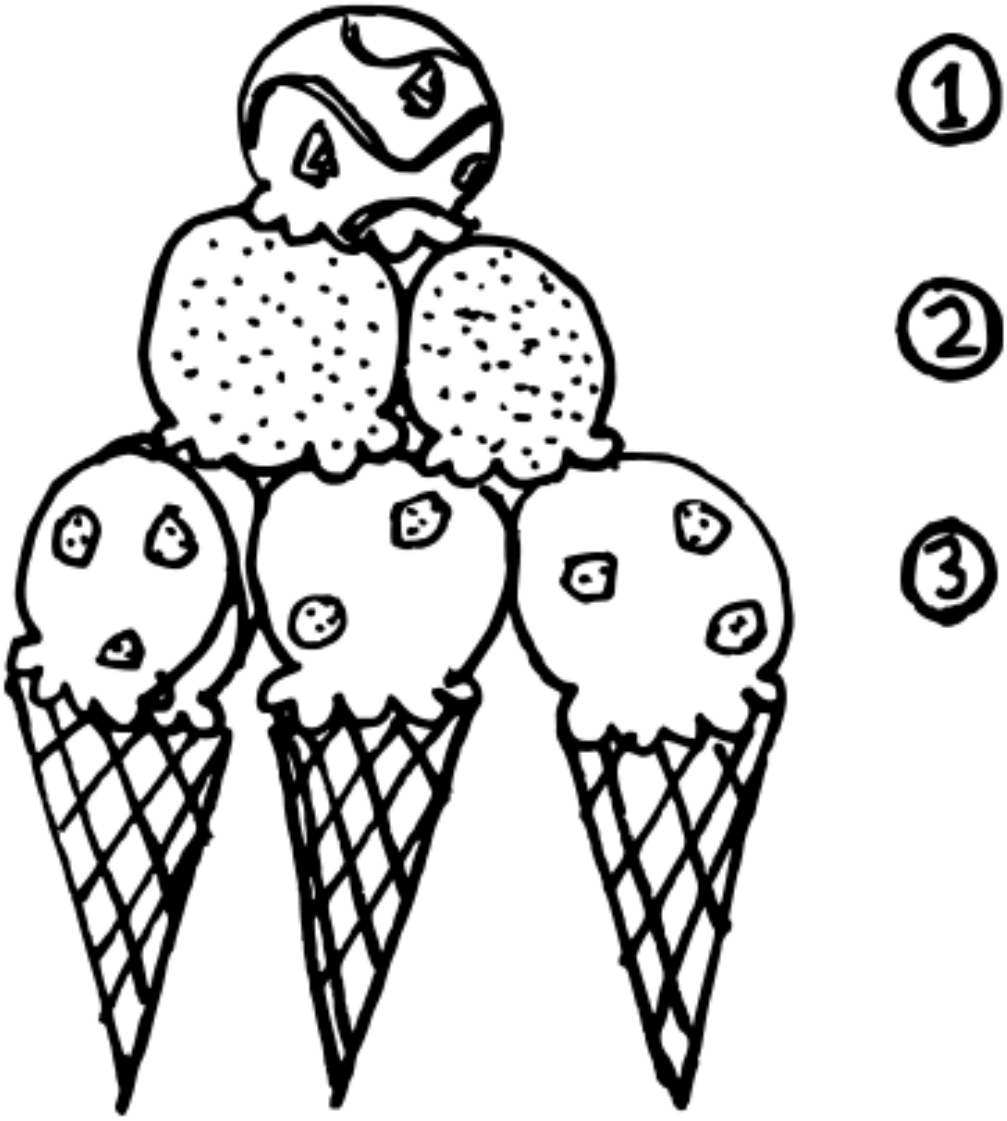

图 3.2: 图 3.2：三层冰淇淋球式布局。

3.3 示例项目结构

来看一个常见例子：一个简单的评分网站。假设我们正在做一个名叫 Ice Cream Ratings 的 Web 应用，用来给不同品牌和口味的冰淇淋打分。

我们会把这样的项目组织成下面这种结构：

示例 3.4：icecreamratings 的布局

Code (text)

```
icecreamratings_project
+-- config/
|   +-- settings/
|   +-- __init__.py
|   +-- asgi.py
|   +-- urls.py
|   +-- wsgi.py
+-- docs/
+-- icecreamratings/
|   +-- media/ # 仅开发环境使用!
|   +-- products/
|   +-- profiles/
|   +-- ratings/
|   +-- static/
|   +-- templates/
+-- .gitignore
+-- Makefile
+-- README.md
+-- manage.py
+-- requirements.txt
```

下面我们来详细过一遍这个布局。正如你看到的，在 icecreamratings_project/ 目录，也就是 <repository_root> 里，我们会放下面这些文件和目录。它们的用途如下：

文件或目录	用途
.gitignore	列出 Git 应该忽略的文件和目录。
config/	项目的 <configuration_root>，用于放置项目级 settings、urls.py 和 wsgi.py 模块。（settings 的组织方式我们会在第 5 章《设置与 requirements 文件》中展开。）
Makefile	包含简单的部署任务和宏。对于更复杂的部署，你可能会改用 Invoke、Paver 或 Fabric 等工具。
manage.py	如果保留它，就不要随意修改内容。更多说明见第 5 章《设置与 requirements 文件》。
README.md 和 docs/	面向开发者的项目文档。我们会在第 25 章《文档：要有执念》中详细讨论。
requirements.txt	列出项目所需的 Python 包，包括 Django 3.x 本身。你会在第 23 章《Django 的秘密武器：第三方包》中看到更多内容。
icecreamratings/	项目的 <django_project_root>。

表 3.1：仓库根目录中的文件和目录

任何人一进入这个项目，就能立刻看到一个高层级的整体视图。我们的经验是，这样的布局能让我们更容易与其他开发者协作，甚至也更方便和非开发人员配合。比如，把偏设计的目录建在根目录下，其实是很常见的做法。

在 icecreamratings_project/icecreamratings 目录，也就是 <django_project_root> 里，我们会放下面这些文件和目录：

文件或目录	用途
media/	只用于开发环境：存放用户生成的静态媒体资源，比如用户上传的照片。对于更大的项目，这些内容通常会托管在独立的静态媒体服务器上。
products/	用于管理和展示冰淇淋品牌的 app。
profiles/	用于管理和展示用户资料的 app。
ratings/	用于管理用户评分的 app。
static/	非用户生成的静态资源，包括 CSS、JavaScript 和图片。对于更大的项目，这些资源通常也会托管在独立的静态服务器上。
templates/	放置全站通用 Django 模板的地方。

表 3.2：Django 项目根目录中的文件和目录

提示：静态资源目录命名的惯例

在上面的例子里，我们遵循 Django 官方文档的惯例，把“非用户生成”的静态资源目录命名为 static/。

如果你觉得这个名字容易让人糊涂，把它叫成 assets/ 或 site_assets/ 也完全没问题。只是别忘了同步更新你的 STATICFILES_DIRS 设置。

3.4 virtualenv 放哪儿？

你会注意到：项目目录及其子目录里，完全没有任何 virtualenv 目录。这是我们有意为之。

这个项目的虚拟环境，更好的做法是放在另一个单独目录里，用来集中保存你所有 Python 项目的虚拟环境。我们喜欢把所有环境放在一个目录里，再把所有项目放到另一个目录里。

例如，在 Mac 或 Linux 上：

示例 3.5：Mac 或 Linux

Code (text)

```
~/projects/icecreamratings_project/
~/.envs/icecreamratings/
```

在 Windows 上：

示例 3.6：Windows

Code (text)

```
c:\projects\icecreamratings_project\
c:\envs\icecreamratings\
```

如果你在用 virtualenvwrapper（Mac 或 Linux）或者 virtualenvwrapper-win（Windows），那个目录默认会是 ~/.virtualenvs/，那么虚拟环境路径通常会变成：

示例 3.7：virtualenvwrapper

Code (text)

```
~/.virtualenvs/icecreamratings/
```

图 3.3: 图 3.3：一个隔离环境，让你的冰淇淋可以自在畅游。

另外也要记住：既然所有依赖都已经记录在 requirements.txt 里，而你又不会直接去修改虚拟环境里的源代码文件，那就完全没有必要把虚拟环境内容纳入版本控制。不过，requirements.txt 本身一定要保留在版本控制中！

警告：不要把环境目录上传到公开仓库

经验还不够足的开发者，在想做 GitHub 作品集时很常见的一个错误，就是把 venv 或 icecreamratings_env 目录也一并塞进公开 GitHub 仓库里。

对任何看项目的人来说，这都是一个非常明显的红旗，几乎等于在最直接的层面暴露出作者经验不足。

.node_modules 目录也是同理。

避免犯这种错误的办法，是所有项目都在根目录放一个 .gitignore 文件。这样一来，那些不该进 Git、也绝不该被提交的文件和目录，就都能被排除掉。

一些有用的参考资料如下：

- help.github.com/en/github/using-git/ignoring-files
  GitHub 官方关于忽略文件的说明。
- github.com/github/gitignore/blob/master/Python.gitignore
  一个不错的 Python 项目基础 .gitignore。

3.4.1 列出当前依赖

如果你不太确定当前虚拟环境里到底用了哪些依赖版本，可以在命令行里输入下面这条命令查看：

示例 3.8：列出当前依赖

Code (bash)

```
$ pip freeze
```

在 Mac 或 Linux 上，你还可以把它直接写进 requirements.txt：

示例 3.9：把当前依赖保存到文件中

Code (bash)

```
$ pip freeze > requirements.txt
```

3.5 超越 startproject

Django 的 startproject 命令允许你创建并使用简单的 Django 项目模板。但随着时间推移，围绕一个项目的控制项（部署、前端工具链等等）会变得越来越复杂。多数人很快就会撞上 startproject 的上限，于是需要更强大的项目模板工具。

这也正是为什么我们会使用 Cookiecutter（cookiecutter.readthedocs.io）。它是一款更高级的项目模板工具，可以用来生成 Django 项目的脚手架。

提示：Audrey 谈 Cookiecutter

我最早在 2013 年创建 Cookiecutter，是为了满足自己生成 Python 包脚手架的需要。它是第一个能同时模板化文件路径和文件内容的项目。这个想法起初我也觉得有点傻，但最后还是决定把它做出来。

现在，针对 Python、Django、FastAPI、C、C++、JavaScript、Ruby、LaTeX/XeTeX、Berkshelf-Vagrant、HTML、Scala、6502 Assembly 等等，已经有成千上万的 Cookiecutter 模板。许多公司也会在内部使用 Cookiecutter，来支撑各种不同工具。

Cookiecutter 不只是一个命令行工具，它本身还是一个被许多组织拿来集成使用的库。你甚至还能在 PyCharm 和 Visual Studio 这样的 IDE 里看到它的集成支持。

在这一节里，我们会展示一个很流行的 Django 项目模板，它就是由 Cookiecutter 渲染出来的。

3.5.1 用 Cookiecutter 生成项目脚手架

Cookiecutter 的工作方式大致如下：

1. 它会先让你填写一系列值（例如 `project_name` 的值）。
2. 然后，它会根据你填入的这些值，生成整套样板项目文件。

第一步，先按照 Cookiecutter 官方文档中的说明安装 Cookiecutter。

3.5.2 用 Cookiecutter Django 生成起始项目

下面是如何用 Cookiecutter 和 Cookiecutter Django 来生成 Django 3 项目脚手架的示例：

示例 3.10：使用 Cookiecutter 和 Cookiecutter Django

Code (text)

```
cookiecutter https://github.com/pydanny/cookiecutter-django
You've downloaded /home/quique/.cookiecutters/cookiecutter-django
→ before. Is it okay to delete and re-download it? [yes]: no
Do you want to re-use the existing version? [yes]: yes
project_name [My Awesome Project]: icecreamratings
project_slug [icecreamratings]:
description [Behold My Awesome Project!]: Support your Ice Cream
→ Flavour!
author_name [Daniel Feldroy]:

<snip for brevity>
```

等你把这些值都填完之后，Cookiecutter 会在你运行命令的那个目录下创建一个项目目录。在上面的例子里，用输入值生成出来的目录名就是 `icecreamratings`。

生成后的项目文件，大体上会和我们前面给出的布局示例相似。它会包含 settings、requirements、起步文档、起始测试套件等等内容。

提示：怎么还有这么多别的文件?

要记住，Cookiecutter Django 远不止覆盖本章前面列出的那些基础项目结构组件。它是我们自己在实际项目中使用的“终极 Django 项目模板”，所以自然也会附带很多额外的功能和细节。

它比 Django 自带的默认 `startproject` 模板要花哨得多。

我们更愿意直接把自己在真实项目里真正使用的模板拿给你看，而不是展示一个我们自己都不用的、删减过的、面向新手的精简模板。

当然，你完全可以 fork Cookiecutter Django，再按自己的 Django 项目需要去做定制。

3.6 startproject 的其他替代方案

人们对“自己的项目结构才是正确做法”这件事，往往会非常有主见。但正如我们前面说过的，并不存在唯一的正确答案。

一个项目和我们的布局不一样，并没有关系。只要它的组织方式具有层级性，或者至少能在根目录的 `README.md` 里把文档、模板、app、settings 等元素分别放在哪里交代清楚，就可以。

我们鼓励你去看看 Cookiecutter Django 的各种 fork，也鼓励你去搜索其他基于 Cookiecutter 的 Django 项目模板。通过研究别人写的项目模板，你会学到各种各样很有意思的技巧。

提示：`cookiecutter-django-vue-graphql-aws`

另一个很不错的开源、基于 Cookiecutter 的模板，是 Grant McConnaughey 的 `cookiecutter-django-vue-graphql-aws`。它以非常简洁的方式，把一些我们自己也很喜欢的技术串在了一起（Django、GraphQL、Vue、AWS Lambda + Zappa 等）。凡是没写文档说明的地方，命名也都足够直白。

更多信息见：

`github.com/grantmcconnaughey/cookiecutter-django-vue-graphql-aws`

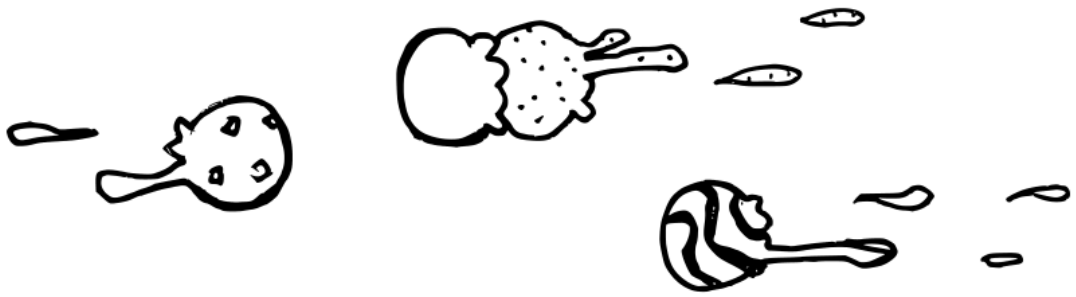

图 3.4: 图 3.4：对项目布局的不同意见，足以引发冰淇淋大战。

3.7 小结

这一章里，我们介绍了自己组织基础 Django 项目结构的做法。我们也给出了一个尽量详细的示例，希望让你尽可能看清我们真实的实践方式。同时，我们还介绍了 Cookiecutter，以及两个相关模板。

项目结构是 Django 里一个差异极大的领域：不同开发者、不同团队之间，实践方式都可能相距很远。适合小团队的结构，未必适合拥有分布式资源的大团队。不管最后选哪一种布局，都应该把它清清楚楚地记录下来。

第四章 Django App 设计基础

不少刚接触 Django 的开发者，都会被 Django 对“app”这个词的用法弄得有些困惑，这完全可以理解。所以，在真正进入 Django app 设计之前，我们必须先把几个定义说清楚。

- **Django project**：由 Django Web 框架驱动的一个 Web 应用。
- **Django app**：用于表达项目中某一个单独方面的小型库。一个 Django project 是由许多个 Django app 组成的。其中一些 app 是项目内部专用的，永远不会复用；另一些则是第三方 Django 包。
- `INSTALLED_APPS`：某个项目通过 `INSTALLED_APPS` 设置启用的 Django app 列表。
- **第三方 Django 包**：本质上就是可插拔、可复用的 Django app，只不过它们已经通过 Python 打包工具打包好了。我们会从第 23 章《Django 的秘密武器：第三方包》开始介绍它们。

图 4.1: 图 4.1：看到下一张图时，这些定义会更容易理解。

图 4.2: 图 4.2：现在明白了吗？如果还没有，就再读一遍。

4.1 Django App 设计的黄金法则

James Bennett 是 Django 核心开发者。关于什么是好的 Django app 设计，我们几乎都是从他那里学来的。他的原话是：

> 创建并维护一个优秀 Django app 的艺术，在于它应当遵循 Douglas McIlroy 所说的那条“截断版 Unix 哲学”：‘Write programs that do one thing and do it well.’

换句话说，每个 app 都应该高度聚焦于自己的任务。如果一个 app 无法用一句长度适中的话解释清楚，或者你在解释时需要说超过一次“and”，那通常就意味着这个 app 太大了，应该继续拆分。

4.1.1 一个项目里 app 的实际例子

假设我们正在为虚构的冰淇淋店“Two Scoops”开发一个 Web 应用。想象一下，我们正准备开张：擦亮台面、做出第一批冰淇淋，同时也在为店铺搭建网站。

我们会把这家店网站对应的 Django project 命名为 `twoscoops_project`。在这个 Django project 里，可能会有如下几个 app：

- 一个 `flavors` app，用来追踪所有冰淇淋口味，并把它们展示在网站上。
- 一个 `blog` app，用于官方 Two Scoops 博客。
- 一个 `events` app，用于在网站上展示门店活动，例如 Strawberry Sundae Sundays 和 Fudgy First Fridays。

这里每个 app 都只做一件特定的事。没错，它们彼此之间是有关联的；你当然也能想象某些活动或博客文章会围绕某些冰淇淋口味展开。但和做一个什么都管的大一统 app 相比，拥有三个各司其职的专门 app 显然要好得多。

将来，我们可能还会继续给网站扩展出这样的 app：

- 一个 `shop` app，让我们可以邮购出售冰淇淋品脱装。
- 一个 `tickets` app，用来处理高级不限量冰淇淋节的门票销售。

注意，`events` 和票务销售是分开的。我们不会把 `events` app 直接扩成一个卖票系统，而是单独建一个 `tickets` app。原因在于：多数活动并不需要门票；而且随着网站扩展，无论是活动日历还是票务销售，本身都可能长出复杂逻辑。

再往后，我们还希望把 `tickets` app 用来给 Icecreamlandia 卖票。那是我们一直想开的冰淇淋主题乐园，里面会塞满各种刺激游乐设施。

我们刚才是不是说这是个虚构例子？咳……总之，这是我们为 Icecreamlandia 想出来的一张早期概念图：

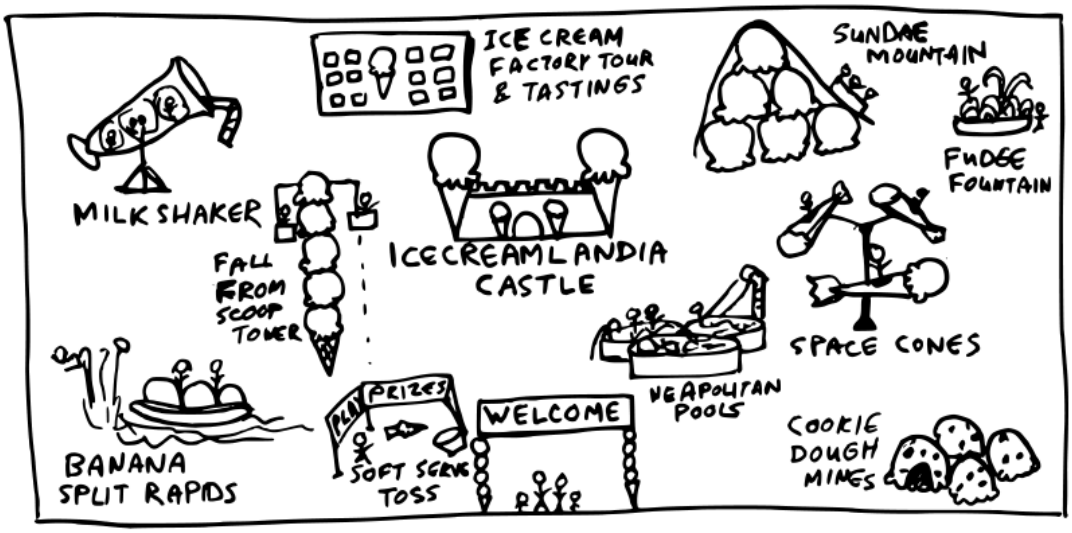

图 4.3: 图 4.3：我们心中的 Icecreamlandia 愿景图。

4.2 Django app 应该怎么命名

每个人都有自己的约定，有些人还很喜欢起非常花哨的名字。我们则偏爱那种平淡、无聊、但一眼就懂的命名体系。事实上，我们主张这样做：

只要可能，就尽量使用单词式名称，比如 `flavors`、`animals`、`blog`、`polls`、`dreams`、`estimates`、`finances`。一个好记、直观的 app 名称，会让整个项目更容易维护。

通常来说，app 名最好是其主要模型的复数形式；不过这个规则也有很多合理例外，`blog` 就是最常见的一个。

不过别只盯着 app 的主要模型看。给 app 起名时，你还应该考虑自己希望 URL 长成什么样。比如，如果你想让网站博客出现在 `http://www.example.com/weblog/` 下，那即便主模型叫 `Post`，也更适合把 app 命名为 `weblog`，而不是 `blog`、`posts` 或 `blogposts`。这样做的好处是，你可以更直观地看出：网站哪一部分对应哪个 app。

还要确保名称本身是合法、符合 PEP 8、并且可 import 的 Python 包名：简短、全小写，不带数字、连字符、点号、空格或特殊字符。为了提升可读性，必要时可以用下划线分隔单词，不过我们总体上仍不鼓励过度使用下划线。

4.3 拿不准的时候，就让 app 保持小而专

别太执着于一次就把 app 设计做到完美。它更像一门艺术，而不是一门科学。有时候你就是得重写，或者拆开它们。这没关系。

尽量让 app 保持小巧。记住：很多个小 app，通常都比少数几个庞然大物式的 app 更好。

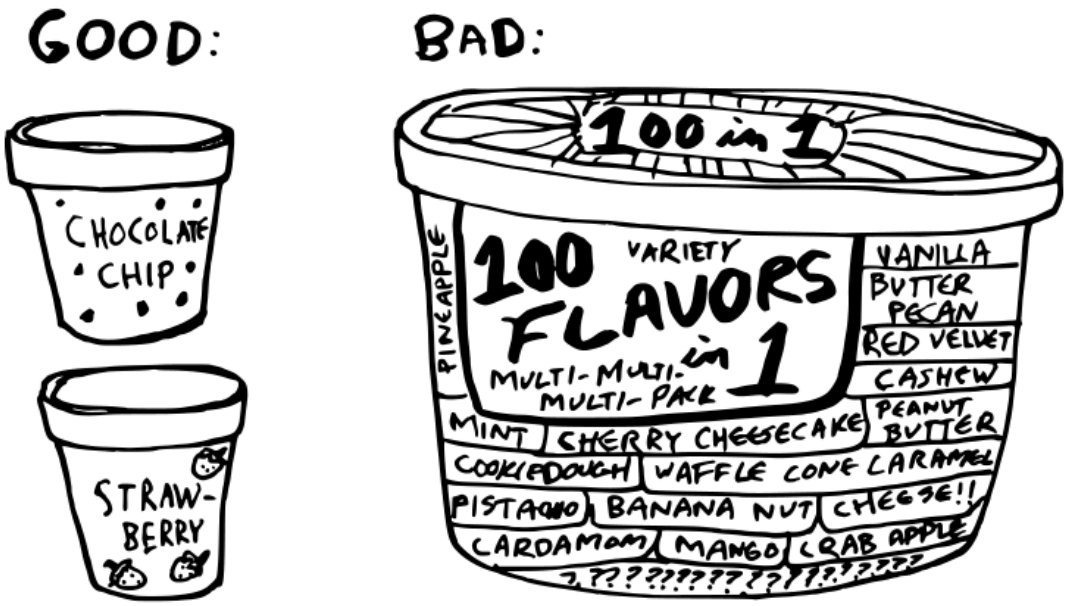

图 4.4: 图 4.4：两个小巧、单口味的品脱装，总比一个混进一百种口味的巨型桶更好。

4.4 一个 app 里应该放哪些模块?

这一节会介绍一个 app 中常见和不常见的 Python 模块。对哪怕稍微用过一阵 Django 的人来说，前面常见模块那部分可能已经很熟，可以直接跳去 4.4.2 节《不常见的 app 模块》。

4.4.1 常见的 app 模块

下面这些模块几乎出现在 99% 的 Django app 里。大多数读者应该都很熟悉，不过我们还是把它们列出来，方便刚进入 Django 世界的人参考。文中凡是以斜杠（`/`）结尾的模块名，都表示它其实是一个 Python package，里面可以包含一个或多个模块。

示例 4.1：常见 app 模块

Code (text)

```
# 常见模块
scoops/
+-- __init__.py
+-- admin.py
+-- forms.py
+-- management/
+-- migrations/
+-- models.py
+-- templatetags/
+-- tests/
+-- urls.py
+-- views.py
```

随着时间推移，为 Django app 构建模块名的约定已经逐渐形成。只要我们在不同 app 中都遵循同一套约定，就等于是在给自己、也给别人建立一套可预期行为模式，让彼此阅读代码时轻松得多。Python 和 Django 确实都足够灵活，所以这些模块名大多并不是“技术上必须”这样命名；但如果你偏不遵循惯例，往后一定会出问题。也许问题不会立刻在技术层面爆炸出来，但将来你或别人回头面对这些非标准模块名时，体验一定会非常折磨人。

4.4.2 不常见的 app 模块

下面这些是不那么常见的模块，许多读者可能熟，也可能不熟：

示例 4.2：不常见的 Django 模块

Code (text)

```
# 不常见模块
scoops/
+-- api/
+-- behaviors.py
+-- constants.py
+-- context_processors.py
+-- decorators.py
+-- db/
+-- exceptions.py
+-- fields.py
+-- factories.py
+-- helpers.py
+-- managers.py
+-- middleware.py
+-- schema.py
+-- signals.py
+-- utils.py
+-- viewmixins.py
```

这些模块各自是做什么的？其中大多数名字从英文上已经足够直白，不过还是有几个值得单独解释一下：

- `api/`：当我们要做 API 时，会专门建这个 package，把相关模块隔离在一起。参见 17.3.1 节《保持 API 模块命名一致》。
- `behaviors.py`：可作为模型 mixin 的存放位置。参见 6.7.1 节《模型行为，也就是 Mixins》。

- `constants.py`：一个适合放 app 级常量配置的文件。如果某个 app 相关常量足够多，把它们单独拆进这个模块，能显著提升清晰度。
- `decorators.py`：我们通常把 decorator 放在这里。更多内容见 9.3 节《Decorator 很甜》。
- `db/`：很多项目会用这个 package 来存放自定义 model field 或相关组件。
- `fields.py`：通常用于 form field，但如果 model field 相关代码还不够多、不值得单独创建一个 `db/` package，有时也会放在这里。
- `factories.py`：我们偏好把测试数据工厂放在这里。24.3.5 节《不要依赖 Fixtures》中会简要提到。
- `helpers.py`：我们放 helper function 的地方。通常是从视图（见 8.5 节《尽量让业务逻辑远离视图》）和模型（见 6.7 节《理解胖模型》）中抽出的代码，用来减轻它们的负担。它基本可以看作 `utils.py` 的同义选择。
- `managers.py`：当 `models.py` 变得过大时，一个常见补救办法就是把自定义 model manager 挪到这里。
- `schema.py`：通常用来放 GraphQL API 背后的代码。
- `signals.py`：虽然我们并不主张随手提供自定义 signal，但需要的话，这里确实是个合适位置。
- `utils.py`：和 `helpers.py` 基本同义。
- `viewmixins.py`：如果把 view mixin 挪到这里，view 模块或 package 会瘦很多。参见 10.2 节《在 CBV 中使用 Mixins》。

本节提到的这些模块，关注点都应该停留在 **app 级别**，而不是全局工具级别。全局级模块的做法，会在 31.1 节《为你的工具代码创建一个 Core App》中介绍。

4.5 另一种路线：更偏 Ruby on Rails 风格的做法

有些人用更接近 Ruby on Rails 风格的方式，也做出了非常成功的项目。Ruby on Rails，通常简称 Rails，是一个和 Django 年纪相仿、同样非常成功的 Ruby Web 应用框架。GitHub、GitLab、Coinbase、Stripe、Square 都是知名 Rails 项目。Django 和 Rails 之间，以及 Python 和 Ruby 之间，确实有足够多相似之处，因此他们的做法也很值得研究。

4.5.1 服务层

在 Django 里写代码时，新人一个非常常见的挣扎点，就是拿不准业务逻辑到底该放哪儿。一个经典例子，就是跨多个模型和 app 去创建用户对象及相关数据的过程。比如在我们的案例里，就是给用户生成一张进入 Icecreamlandia 主题乐园的门票：

1. 系统在一个名为 `create_user()` 的 manager 方法里创建用户记录。
2. `create_user()` 把用户头像上传到某个文件托管系统。
3. `create_user()` 还会通过另一个名为 `create_ticket()` 的 manager 方法创建一张门票。
4. `create_ticket()` 再去调用 Icecreamlandia 入园检票 app 的 API，而这个 app 是第三方服务。

这类逻辑在传统 Django 写法里，通常会分散在分配给 `User` 和 `Ticket` 的 manager 方法上。也有人更喜欢用 class method。于是你可能会看到这样的代码：

示例 4.3：典型的 Django 业务逻辑放置方式

Code (python)

```python
class UserManager(BaseUserManager):
    """In users/managers.py"""

    def create_user(self, email=None, password=None, avatar_url=None):
        user = self.model(
            email=email,
            is_active=True,
```

```
            last_login=timezone.now(),
            registered_at=timezone.now(),
            avatar_url=avatar_url,
        )
        resize_avatar(avatar_url)
        Ticket.objects.create_ticket(user)
        return user


class TicketManager(models.manager):
    """In tasks/managers.py"""

    def create_ticket(self, user: User):
        ticket = self.model(user=user)
        send_ticket_to_guest_checkin(ticket)
        return ticket
```

这种做法当然能工作。但 service layer 流派认为，用户对象和票务对象之间应该有更明确的关注点分离。具体来说，把调用 `Ticket` 逻辑的代码埋进 `User` 代码中，会让这两个领域彼此耦合得过紧。于是他们会新增一层专门用来隔离关注点，这一层就叫 **service layer**。在这种架构里，代码通常会放进 `services.py` 和 `selectors.py`。

示例 4.4：Service Layer

Code (text)

```
# service layer 示例
scoops/
+-- api/
+-- models.py
+-- services.py   # 业务逻辑所在
+-- selectors.py  # 查询逻辑所在
+-- tests/
```

在 service layer 设计下，上面那段逻辑可能会变成这样执行：

示例 4.5：Service Layer 里的业务逻辑放置方式

Code (python)

```
# In users/services.py
from .models import User
from tickets.models import Ticket, send_ticket_to_guest_checkin


def create_user(email: str, password: str, avatar_url: str) -> User:
    user = User(email=email, password=password, avatar_url=avatar_url)
    user.full_clean()
    user.resize_avatar()
    user.save()

    ticket = Ticket(user=user)
    send_ticket_to_guest_checkin(ticket)
    return user
```

一个很流行的说明，可以参考：

`github.com/HackSoftware/Django-Styleguide#services`

这种方案的优点包括：

- 在这个简化示例里，代码行数是 12 行，而不是 17 行。
- 松耦合意味着用户或票务代码将来更容易被替换。
- 关注点分离后，更容易对单个组件做功能测试。
- 相比传统 Django 写法，这种方式给返回对象加类型标注更容易。

它的缺点包括：

- 在现实项目里，几乎不会有“整块替换整个模块而不重写”的场景。
- 对小项目来说，它往往只会额外增加不必要的复杂度。
- 对复杂项目来说，service layer 最后通常会膨胀成几千行代码，迫使你再发明一层新架构来支撑它。
- 到处都要用 `selectors.py`，会让本来一个简单 ORM 查询就能完成的事情，多绕一步。
- Django 的持久化能力，本来就建立在“可 import、放在模型自身上的业务逻辑”之上；service layer 把这条路拿掉了，也就连带牺牲掉了一些更高级的技巧。

更多反驳 service layer 的文章，或者关于如何用传统 Django 工具组织业务逻辑的说明，可以参考长期维护者 James Bennett 的博客，以及 Django REST Framework 创始人 Tom Christie 的文章：

- `b-list.org/weblog/2020/mar/16/no-service/`
- `b-list.org/weblog/2020/mar/23/still-no-service/`
- `dabapps.com/blog/django-models-and-encapsulation/`

对《Two Scoops of Django》的作者来说，service layer 并不是什么新鲜事，我们这些年已经见过太多了。和一切偏离 Django 核心实践的技巧一样，按我们的经验来看，使用 service layer 的项目，失败概率似乎会略高一点。这倒不是说这种做法毫无价值，而是说：额外抽象出来的这一层，未必值得你为它付出成本。

4.5.2 大型单 App 项目

所谓大型单 app 项目，就是把所有代码都塞进同一个 app 里。对于小型 Django 项目来说，这种做法并不少见；但到了更大的项目里，通常就不会继续沿用这种模式了。

它确实有一些好处。最明显的一点是：migration 更简单，数据表命名模式也会更统一。

也必须承认，Rails 之类的框架就很拥抱这种做法，而且做得相当成功。可问题在于：Rails 和其他一些工具本来就是围绕这种模式设计的，而 Django 并不是。要在 Django 里走这条路，你需要非常熟悉那些在 Django 文档里并不常被系统描述的模式和手法。

典型情况下，所有东西都会堆进巨大无比的 `models.py`、`views.py`、`tests.py` 和 `urls.py` 模块里。最后，项目不是被不断膨胀的复杂度拖垮，就是不得不再按照模型领域把这些文件拆成子模块。拿我们的 icecreamlandia 项目来说，凡是和“Users”有关的，就放进 `models/users.py`；和“Tickets”有关的，就放进 `models/tickets.py`。

当然，这种方案如果有足够前瞻性地去做，也不是完全行不通。事实上，Audrey 就曾经用这种方式非常成功地完成过一个项目，只是中途确实比预期多做了不少代码调度工作。我们的意见是：只有当一个 Djangonaut 已经顺利做完好几个 Django 项目之后，才值得认真尝试这种路子。

警告：对过度背离 Django 约定的架构要格外小心

James Beith 是一位我们很尊敬的资深工程师。他对 Django 项目结构有一套颇有意思的看法。在他的设计里，一些 Django 常见模式会被显式改写。虽然看起来对他自己的项目确实有效，但从维护角度看，代价是：任何新加入项目的开发者，都得先学习一套全新范式。

Django 的迷人之处，很大程度上正是建立在“约定优于配置”之上的，而他的设计则是明确地打破了这一点。我们尊重他尝试新方向的意愿，但也觉得，他在“重新架构 Django 项目”这件事上走得太远了。

James Beith 的文章：

`jamesbeith.co.uk/blog/how-to-structure-django-projects/`

4.6 小结

这一章讨论的是 Django app 设计这门艺术。核心观点很明确：每个 Django app 都应该高度聚焦于自己的任务，并且拥有一个简单、好记的名字。如果某个 app 看起来已经复杂得过头，就应该把它继续拆成更小的 app。要把 app 设计做好，需要练习，也需要下功夫，但这份投入绝对值得。

我们也讨论了其他一些组织 app 架构、以及在 app 内部组织逻辑调用方式的替代路线。

第五章 Settings 与 Requirements 文件

Django 3 在 `settings` 模块中有 150 多个可控设置，其中大多数都带有默认值。设置会在服务器启动时载入，因此有经验的 Django 开发者通常不会在生产环境里试图临时改动 settings，因为那意味着必须重启服务器。

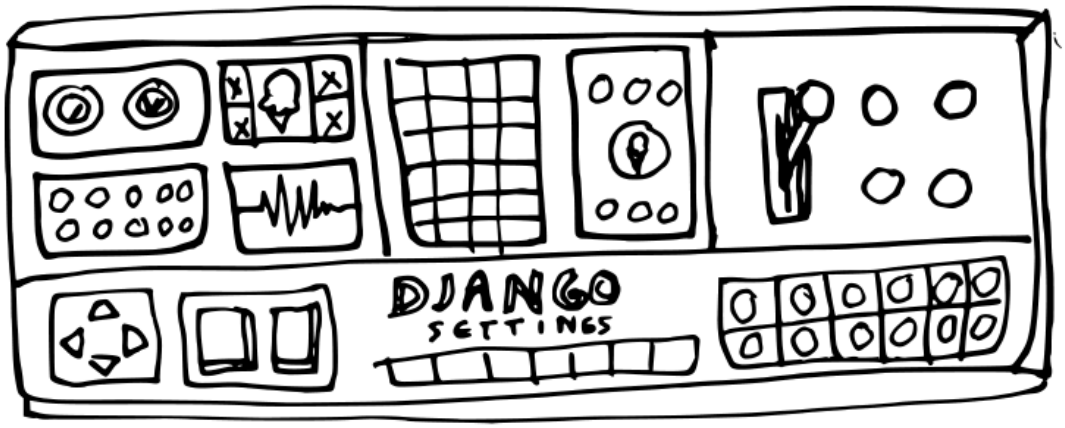

图 5.1: 图 5.1：随着项目不断长大，Django settings 往往会变得相当复杂。

下面是我们偏好遵循的一些最佳实践：

- 所有 settings 文件都必须纳入版本控制。尤其是在生产环境中，settings 变更的时间、原因和说明都必须可追踪。
- 遵循 Don’t Repeat Yourself。应该从基础 settings 文件继承，而不是在文件之间来回复制粘贴。
- 把 secret key 保管好。它们不应该进入版本控制。

5.1 避免不受版本控制的本地 settings

我们以前也曾主张那种不纳入版本控制的 `local_settings` 反模式。现在我们知道那并不好。

作为开发者，我们确实会有一些开发阶段必需的设置，比如调试工具相关设置，而这些设置在 staging 或生产服务器上应该被禁用，很多时候甚至根本不该安装。

此外，也常常有充分理由让某些设置不要出现在公开或私有代码仓库里。最先想到的就是 `SECRET_KEY`，但像 Amazon、Stripe 一类服务的 API key 设置，以及其他密码类变量，也都必须保护起来。

警告：保护好你的 secrets！

`SECRET_KEY` 会用于 Django 的加密签名功能，因此必须设置成唯一且不可预测的值，而且最好不要放进版本控制。使用一个已知的 `SECRET_KEY` 运行 Django，会让 Django 的许多安全保护失效，进而造成严重安全漏洞。更多细节请参阅 `docs.djangoproject.com/en/3.2/topics/signing/`。

对 `SECRET_KEY` 的这条警告，同样适用于生产数据库密码、AWS key、OAuth token，以及项目运行所需的其他任何敏感数据。

本章稍后会在“用环境设置隔离 secret key”一节中说明如何处理 `SECRET_KEY` 问题。

一个常见解法，是为每台服务器或每台开发机器在本地创建 `local_settings.py` 模块，并刻意不把它纳入版本控制。于是，开发者会在其中做开发专用的 settings 调整，甚至把业务逻辑也塞进去，而这些代码都不会被版本控制系统跟踪。staging 和部署服务器也会带着各自位置特定的 settings 与逻辑，但同样不受版本控制追踪。

这能出什么问题呢?!

咳……

- 每台机器上都会有未跟踪代码。

- 当你花了几个小时都无法在本地复现生产 bug，最后才发现问题出在“只存在于生产 settings 中的自定义逻辑”时，你会薅掉多少头发？
- 当你在本地发现一个“bug”、修复它并推到生产，结果后来才发现其实是你自己 `local_settings.py` 里的定制改动导致的问题，而且现在网站已经被你弄崩了，你会以多快的速度从大家面前逃走？
- 每个人都在到处复制粘贴同一个 `local_settings.py`。这不就是把 Don’t Repeat Yourself 的违背扩大了一个量级吗？

我们不如换一种思路。把开发、staging、测试和生产 settings 拆成彼此独立的组件，它们统一继承自一个放在版本控制中的公共基础对象。同时，我们还要确保所采用的方法能让服务器 secrets 继续保持秘密。

继续读下去，看看具体怎么做。

5.2 使用多个 Settings 文件

提示：这种配置模式的来历

这里介绍的做法，源自 Jacob Kaplan-Moss 在 OSCON 2011 的演讲 *The Best (and Worst) of Django* 中提到的所谓“The One True Way”。参见 `slideshare.net/jacobian/the-best-and-worst-of-django`。

在这种结构下，你不会只保留一个 `settings.py`，而是改用一个 `settings/` 目录来容纳各个 settings 文件。这个目录通常会长成下面这样：

示例 5.1：Settings 目录

Code (text)

```
settings/
+-- __init__.py
+-- base.py
+-- local.py
+-- staging.py
+-- test.py
+-- production.py
```

警告：Requirements + Settings

每个 settings 模块都应该对应一份 requirements 文件。我们会在本章后面的 5.5 节《使用多个 Requirements 文件》中说明。

提示：持续集成服务器的多文件配置

你通常还会想准备一个 `ci.py` 模块来存放 CI 服务器的 settings。类似地，如果项目足够大，而且还有其他专用服务器，也可能会为它们分别准备自定义 settings 文件。

Settings 文件	用途
`base.py`	项目所有实例共用的设置。
`local.py`	你在本地开发项目时使用的 settings 文件。本地开发专用设置包括 `DEBUG` 模式、日志级别，以及启用 `django-debug-toolbar` 这类开发者工具。
`staging.py`	用于在生产服务器上运行一个半私有 staging 版本的网站。在你的工作被推进到生产前，管理者和客户应该先看这里。
`test.py`	用于运行测试的设置，包括测试运行器、内存数据库定义和日志设置。
`production.py`	线上生产服务器使用的 settings 文件，也就是承载真实网站的服务器所用配置。这个文件里只应该有生产级设置。有时它也会叫 `prod.py`。

表 5.1：Settings 文件及其用途

下面我们来看看，在这种结构下，如何使用 shell 和 runserver 这两个 management command。你需要借助 --settings 命令行选项，因此命令行里要输入类似下面这样的内容。

如果你想在使用 settings/local.py 的前提下启动 Django 的 Python 交互解释器：

示例 5.2：使用本地 settings 的 shell

Code (bash)

```
python manage.py shell --settings=config.settings.local
```

如果你想在使用 settings/local.py 的前提下运行本地开发服务器：

示例 5.3：使用本地 settings 的 runserver

Code (bash)

```
python manage.py runserver --settings=config.settings.local
```

提示：DJANGO_SETTINGS_MODULE 与 PYTHONPATH

一个很棒的替代方案，是不要处处手打 --settings 命令行参数，而是把 DJANGO_SETTINGS_MODULE 和 PYTHONPATH 这两个环境变量设置成你想使用的 settings 模块路径。要做到这一点，你需要为每个环境都把 DJANGO_SETTINGS_MODULE 指向相应的 settings 模块。

如果你对 virtualenvwrapper 用得比较深入，另一种做法是把 DJANGO_SETTINGS_MODULE 和 PYTHONPATH 写进 postactivate 脚本，并在 postdeactivate 脚本里将它们清掉。这样一来，一旦虚拟环境被激活，你就可以在任何地方直接输入 python，然后把这些值导入到项目里。这也意味着在命令行里输入 django-admin 时，无需额外加 --settings 参数。

对于我们刚才描述的这种 settings 结构，下面这些值可以用于 --settings 命令行参数，或者用于设置 DJANGO_SETTINGS_M 环境变量：

环境	--settings 应使用的值（或 DJANGO_SETTINGS_MODULE 的值）
本地开发服务器	twoscoops.settings.local
staging 服务器	twoscoops.settings.staging
测试服务器	twoscoops.settings.test
生产服务器	twoscoops.settings.production

表 5.2：按运行位置设置 DJANGO_SETTINGS_MODULE

5.2.1 一个开发 settings 示例

如前所述，开发环境需要一组专门的 settings，比如选择 console email backend、把项目设成 DEBUG 模式，以及其他只在开发时使用的配置项。我们会把这类开发设置写进 settings/local.py，例如：

示例 5.4：settings/local.py

Code (python)

```
from .base import *

DEBUG = True

EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'

DATABASES = {
    'default': {
```

```
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'twoscoops',
        'HOST': 'localhost',
    }
}

INSTALLED_APPS += ['debug_toolbar', ]
```

现在可以在命令行里这样试一下：

示例 5.5：使用本地 settings 运行 runserver

Code (bash)

```
python manage.py runserver --settings=config.settings.local
```

打开 http://127.0.0.1:8000，享受这套已经可以放心纳入版本控制的开发 settings 吧。你和其他开发者将共享同一套开发 settings 文件；对于协作项目来说，这非常棒。

而且它还有另一层好处：再也不用在不同项目之间到处复制粘贴 `if DEBUG` 或 `if not DEBUG` 逻辑了。settings 一下子就简单了许多！

此处我们还想特别说明一下：Django settings 文件是我们唯一明确主张可以使用 `import *` 的地方。原因在于，在 Django settings 模块这个特殊场景里，我们的目的就是覆盖整个命名空间。

5.2.2 多份开发 settings

有时我们正在做一个大型项目，不同开发者需要不同的设置，这时和队友共用同一个 `local.py` settings 模块就不够用了。

不过，即便如此，把这些 settings 继续纳入版本控制，也仍然比让每个人都去按自己的喜好定制同一个 `local.py` 模块要好。一个不错的做法，是准备多份开发 settings 文件，例如 `local_audrey.py` 和 `local_pydanny.py`：

示例 5.6：settings/local_pydanny.py

Code (python)

```
# settings/local_pydanny.py
from .local import *

# Set short cache timeout
CACHE_TIMEOUT = 30
```

为什么？因为把你自己的 settings 文件纳入版本控制本来就有好处，而能看到队友的开发 settings 文件同样也有好处。这样一来，你能判断某个人的本地开发环境是不是少了某个关键或有帮助的设置，也能确保大家的本地 settings 文件保持同步。我们的项目里，经常使用下面这样的 settings 布局：

示例 5.7：自定义 Settings

Code (text)

```
settings/
+-- __init__.py
+-- base.py
+-- local_audreyr.py
+-- local_pydanny.py
+-- local.py
+-- staging.py
+-- test.py
+-- production.py
```

5.3 把配置与代码分离

local_settings 反模式之所以会出现，其中一个原因就是：如果把 SECRET_KEY、AWS key、API key 或服务器专属值直接塞进 settings 文件，会带来不少问题：

- 不同部署之间变化很大的，是配置；不是代码。
- secret key 是配置值，不是代码。
- secret 本来就应该是 secret。把它们放进版本控制，意味着所有能访问仓库的人也都能访问这些内容。
- 各类平台即服务通常并不会给你在单台服务器上直接修改代码的能力。就算允许，这也是一种非常危险的做法。

为了解决这个问题，我们的答案是使用环境变量，我们把这种模式称为“环境变量模式”。

Django 和 Python 支持的每一种操作系统，都提供了方便的环境变量能力。

把 secrets 放进环境变量里，有这些好处：

- 把 secrets 挪出 settings 之后，你就可以毫无顾虑地把所有 settings 文件都纳入版本控制。实际上，你的全部 Python 代码，本来就都应该进入版本控制，包括 settings。
- 不再需要每个开发者各自维护一份容易过时、靠复制粘贴得来的 local_settings.py.example 文件，所有人共享同一个版本控制中的 settings/local.py 即可。
- 系统管理员可以快速部署项目，而不必去修改那些包含 Python 代码的文件。
- 大多数平台即服务都推荐用环境变量管理配置，而且通常内建了对应的设置与管理功能。

提示：12 Factor App 提倡把配置放进环境中

如果你读过 12 Factor App 关于配置的文章，就会认出这种模式。可参考 12factor.net/config。有些开发者甚至主张把环境变量和单一 settings 模块组合使用。我们会在第 37 章《附录 E：Settings 的替代方案》中讨论这种做法。

5.3.1 在用环境变量存 secret 前，先提醒一句

在你开始设置环境变量之前，最好先具备以下条件：

- 有一套管理你即将存放进去的 secret 信息的方法。
- 充分理解服务器上 bash 与环境变量的交互方式，或者愿意把项目托管到平台即服务上。

更多信息参见 en.wikipedia.org/wiki/Environment_variable。

警告：环境变量方案在 Apache 下行不通

如果你的目标生产环境使用的是除 Elastic Beanstalk 外的 Apache，你很快就会发现，下面描述的这种操作系统级环境变量设置方式并不能正常工作。更让人困惑的是，Apache 自己也有一套环境变量系统，但它和你真正需要的东西“几乎一样，却又不完全一样”。

如果你在使用 Apache，又想避开 local_settings 反模式，我们建议你阅读本章后面的 5.4 节《当你无法使用环境变量时》。

5.3.2 如何在本地设置环境变量

在 Catalina 之前的 macOS，以及许多默认使用 bash 的 Linux 发行版里，可以把下面这样的内容追加到 .bashrc、.bash_profile 或 .profile 文件末尾。Catalina 及之后的 macOS 使用的是 .zshrc。如果你要处理多个项目，它们用的是同一个 API 但 key 不同，也可以把这些内容写到虚拟环境的 bin/postactivate 脚本末尾：

示例 5.8：在 Linux/OSX 上设置环境变量

Code (bash)

```
export SOME_SECRET_KEY=1c3-cr3am-15-yummy
export AUDREY_FREEZER_KEY=y34h-r1ght-d0nt-t0uch-my-1c3-cr34m
```

在 Windows 系统上会稍微麻烦一点。你可以在命令行（`cmd.exe`）里用 `setx` 命令逐个持久化设置，不过它们只有在你关闭并重新打开命令提示符后才会生效。更好的做法，是把这些命令写进虚拟环境的 `bin/postactivate.bat` 脚本末尾，这样激活时就能自动可用：

示例 5.9：在 Windows 上设置环境变量

Code (bat)

```
> setx SOME_SECRET_KEY 1c3-cr3am-15-yummy
```

PowerShell 比默认的 Windows shell 强大得多，而且从 Windows Vista 开始就自带。使用 PowerShell 设置环境变量时：

仅对当前 Windows 用户生效：

示例 5.10：在 PowerShell 中设置环境变量

Code (powershell)

```
[Environment]::SetEnvironmentVariable('SOME_SECRET_KEY',
    '1c3-cr3am-15-yummy', 'User')

[Environment]::SetEnvironmentVariable('AUDREY_FREEZER_KEY',
    'y34h-r1ght-d0nt-t0uch-my-1c3-cr34m', 'User')
```

全机生效：

示例 5.11：在 PowerShell 中全局设置环境变量

Code (powershell)

```
[Environment]::SetEnvironmentVariable('SOME_SECRET_KEY',
    '1c3-cr3am-15-yummy', 'Machine')

[Environment]::SetEnvironmentVariable('AUDREY_FREEZER_KEY',
    'y34h-r1ght-d0nt-t0uch-my-1c3-cr34m', 'Machine')
```

关于 PowerShell 的更多信息，参见 `en.wikipedia.org/wiki/PowerShell`。

提示：virtualenvwrapper 会让这件事更轻松

本书前面提到过，`virtualenvwrapper` 可以简化“按虚拟环境设置环境变量”这件事，是个很好用的工具。不过，它的配置需要你对 shell 以及 Mac、Linux 或 Windows 有超过入门级别的理解。

5.3.3 如何在本地取消环境变量

通过前面这些命令设置环境变量后，只要当前终端 shell 没有结束、变量没有被取消，它就会一直存在。这意味着，即便你停用了某个虚拟环境，那个环境变量也还在。以我们的经验来看，99% 的时候这都没问题。不过，某些场景下我们确实希望更严密地控制环境变量。这时，就要按不同操作系统或 shell 变体执行相应命令：

示例 5.12：在 Linux/OSX/Windows 中取消环境变量

Code (bash)

```
unset SOME_SECRET_KEY
unset AUDREY_FREEZER_KEY
```

示例 5.13：在 PowerShell 中取消环境变量

Code (powershell)

```
[Environment]::UnsetEnvironmentVariable('SOME_SECRET_KEY', 'User')
[Environment]::UnsetEnvironmentVariable('AUDREY_FREEZER_KEY', 'User')
```

如果你使用了 `virtualenvwrapper`，并且希望每次停用虚拟环境时都自动取消这些环境变量，可以把上述命令放进 `postdeactivate` 脚本。

5.3.4 如何在生产环境设置环境变量

如果你使用的是自有服务器，具体做法会因你采用的工具和部署复杂度而异。对于最简单的单服务器测试项目，你可以手动设置这些环境变量。但如果你使用脚本或工具来做自动化服务器初始化和部署，方案往往会更复杂。更多信息请查看你所用部署工具的文档。

如果你的 Django 项目部署在平台即服务（PaaS）上，例如 Heroku、Python Anywhere、platform.sh 等，也请查看它们各自的文档说明。

要看看 Python 侧如何读取环境变量，可以打开一个新的 Python 提示符并输入：

示例 5.14：在 Python REPL 中访问环境变量

Code (python)

```
>>> import os
>>> os.environ['SOME_SECRET_KEY']
'1c3-cr3am-15-yummy'
```

如果想在某个 settings 文件里读取环境变量，可以这样写：

示例 5.15：在 Python 中访问环境变量

Code (python)

```
# settings/production.py 顶部
import os
SOME_SECRET_KEY = os.environ['SOME_SECRET_KEY']
```

这段代码只是从操作系统中读取名为 SOME_SECRET_KEY 的环境变量值，并把它保存到一个名为 SOME_SECRET_KEY 的 Python 变量中。

遵循这种模式，所有代码都可以继续留在版本控制里，而所有 secret 也都能保持安全。

5.3.5 处理缺失 Secret Key 时的异常

在前面的实现中，如果 SECRET_KEY 不存在，就会抛出 KeyError，从而让项目无法启动。这其实很好，但 KeyError 本身并不会清楚告诉你到底出了什么问题。没有更有帮助的错误信息时，排查起来会很痛苦，尤其是当你正在服务器上紧张部署、用户还在等，而你的冰淇淋也快化了的时候。

下面这段代码能让“缺少环境变量”的问题更容易排查。如果你采用的是我们推荐的“环境变量管理 secrets”方案，那你应该把它加到 settings/base.py 中：

示例 5.16：get_env_variable() 函数

Code (python)

```
# settings/base.py
import os

# Normally you should not import ANYTHING from Django directly
# into your settings, but ImproperlyConfigured is an exception.
from django.core.exceptions import ImproperlyConfigured

def get_env_variable(var_name):
    """Get the environment variable or return exception."""
    try:
        return os.environ[var_name]
    except KeyError:
        error_msg = 'Set the {} environment variable'.format(var_name)
        raise ImproperlyConfigured(error_msg)
```

接着，在任意一个 settings 文件里，你都可以像下面这样从环境变量中加载 secret key：

示例 5.17：使用 get_env_variable()

Code (python)

```
SOME_SECRET_KEY = get_env_variable('SOME_SECRET_KEY')
```

这样一来，如果你没有把 SOME_SECRET_KEY 设成环境变量，就会得到一个最终以如下信息结尾的 traceback：

示例 5.18：get_env_variable() 生成的错误

Code (text)

```
django.core.exceptions.ImproperlyConfigured: Set the
SOME_SECRET_KEY environment variable.
```

警告：不要在 Settings 模块中导入 Django 组件

这么做可能引发许多难以预料的副作用，所以在 settings 里应避免导入任何 Django 组件。ImproperlyConfigured 是个例外，因为它正是 Django 官方用来表示“项目配置不正确”的异常。为了帮你排查问题，我们还把出错设置项的名字一并写进了错误消息。

包提示：用于 Settings 管理的包

不少第三方包都把我们这个 get_env_variable() 函数的思路继续扩展了，加入了默认值、类型处理以及支持 .env 文件等功能。缺点也和所有复杂包一样：边角场景有时会带来麻烦。即便如此，它们大多数仍然相当有用，我们常用的几个包括：

- github.com/joke2k/django-environ（Cookiecutter Django 会使用）
- github.com/jazzband/django-configurations

提示：在多 settings 文件场景里使用 django-admin 而不是 manage.py

Django 官方文档建议，在处理多个 settings 文件时，应该使用 django-admin，而不是 manage.py：docs.djangoproject.com

不过话说回来，如果你在让 django-admin 跑起来这件事上卡住了，那么在开发和启动站点时继续使用 manage.py 也完全没问题。

5.4 当你无法使用环境变量时

把 secrets 存进环境变量的问题在于：它并不总是可行。最常见的场景，是你用 Apache 提供 HTTP 服务；不过即使是基于 Nginx 的环境，如果运维团队有自己特别的处理方式，也可能遇到同样问题。遇到这种情况时，我们主张的不是退回 local_settings 反模式，而是改用一种不受版本控制的、不可执行文件方案，我们把它叫作“secrets file 模式”。

要实现 secrets file 模式，可以按下面三步来：

1. 使用你选择的配置格式创建一个 secrets 文件，可以是 JSON、.env、Config、YAML，甚至 XML。
2. 添加一个 secrets loader，下文会给出基于 JSON 的示例，以一种内聚、显式的方式管理这些 secrets。
3. 把 secrets 文件名加入项目的 .gitignore。

5.4.1 使用 JSON 文件

我们偏好使用扁平的 JSON 文件。JSON 的优势在于，它既是 Python 工具喜欢的格式，也是非 Python 工具喜欢的格式。做法很简单，先创建一个 secrets.json 文件：

示例 5.19：secrets.json

Code (json)

```
{
  "FILENAME": "secrets.json",
  "SECRET_KEY": "I've got a secret!",
```

```
  "DATABASES_HOST": "127.0.0.1",
  "PORT": "5432"
}
```

要使用这个 secrets.json 文件，就把下面的代码加到你的基础 settings 模块里。

示例 5.20：get_settings() 函数

Code (python)

```
# settings/base.py
import json

# Normally you should not import ANYTHING from Django directly
# into your settings, but ImproperlyConfigured is an exception.
from django.core.exceptions import ImproperlyConfigured

# JSON-based secrets module
with open('secrets.json') as f:
    secrets = json.load(f)

def get_secret(setting, secrets=secrets):
    '''Get the secret variable or return explicit exception.'''
    try:
        return secrets[setting]
    except KeyError:
        error_msg = 'Set the {0} environment variable'.format(setting)
        raise ImproperlyConfigured(error_msg)

SECRET_KEY = get_secret('SECRET_KEY')
```

现在，我们加载 secrets 的来源已经从“不受版本控制、且可执行的代码”变成了“不可执行的 JSON 文件”。好耶！

包提示：Zappa 提供了很强的选项

我们偏好用 Zappa 部署到 AWS Lambda，其中一个原因就是它在环境变量管理上提供了很灵活、很强大的选项。这绝对值得看一看，我们也希望其他工具和托管平台能跟上。参考：github.com/Miserlou/Zappa#setting-environment-

5.4.2 使用 .env、Config、YAML 和 XML 文件格式

虽然我们偏爱这种被迫保持简洁的扁平 JSON，但也有人更喜欢其他文件格式。至于如何为这些格式编写额外的 get_secret() 方案，就留给读者自己了。只要记得对像 yaml.safe_load()、XML bomb 之类的问题保持警惕。参见 28.10 节《防御 Python 代码注入攻击》。

5.5 使用多个 Requirements 文件

最后，关于“多个 settings 文件”这种做法，还有一件事你必须知道。最佳实践是：每个 settings 文件都应有一份与之对应的 requirements 文件。这样就能确保每台服务器只安装自己真正需要的依赖。

按照 Jeff Triplett 推荐给我们的做法，第一步是在 <repository_root> 下创建一个 requirements/ 目录。然后创建一组与 settings 目录内容相匹配的 .txt 文件。结果大致如下：

示例 5.21：分段 Requirements

Code (text)

```
requirements/
+-- base.txt
+-- local.txt
+-- staging.txt
+-- production.txt
```

在 base.txt 中，放入所有环境都会用到的依赖。例如，你可能会写成下面这样：

示例 5.22：requirements/base.txt

Code (text)

```
Django==3.2.0
psycopg2-binary==2.8.8
djangorestframework==3.11.0
```

local.txt 中则放本地开发所需依赖，例如：

示例 5.23：requirements/local.txt

Code (text)

```
-r base.txt # includes the base.txt requirements file
coverage==5.1
django-debug-toolbar==2.2
```

持续集成服务器的需求，则可能会让 ci.txt 看起来像这样：

示例 5.24：requirements/ci.txt

Code (text)

```
-r base.txt # includes the base.txt requirements file
coverage==5.1
```

生产环境安装应尽量接近其他环境，因此 production.txt 常常只需要引用 base.txt：

示例 5.25：requirements/production.txt

Code (text)

```
-r base.txt # includes the base.txt requirements file
```

5.5.1 从多个 Requirements 文件安装

用于本地开发时：

示例 5.26：安装本地 Requirements

Code (bash)

```
pip install -r requirements/local.txt
```

用于生产环境时：

示例 5.27：安装生产 Requirements

Code (bash)

```
pip install -r requirements/production.txt
```

提示：精确锁定 Requirements 版本

本章所有 `pip requirements.txt` 示例，都把包版本写得很明确。这样能让项目更稳定。我们会在 23.7.2 节《步骤 2：把包和版本号加入 Requirements》中详细说明。

提示：在 PaaS 中使用多个 Requirements 文件

这个问题我们会在?: ?? 中讨论。[原文此处即为未完成交叉引用]

5.6 在 Settings 中处理文件路径

如果你切换到“多个 settings 文件”的结构后，遇到了 template、media 等文件路径相关的新错误，也别慌。这一节就是帮你解决这些问题的。

我们非常恳切地请求读者：永远不要在 Django settings 文件里硬编码文件路径。这真的很糟糕：

示例 5.28：永远不要硬编码文件路径

Code (python)

```
# settings/base.py

# Configuring MEDIA_ROOT
# DON'T DO THIS! Hardcoded to just one user's preferences
MEDIA_ROOT = '/Users/pydanny/twoscoops_project/media'

# Configuring STATIC_ROOT
# DON'T DO THIS! Hardcoded to just one user's preferences
STATIC_ROOT = '/Users/pydanny/twoscoops_project/collected_static'

# Configuring STATICFILES_DIRS
# DON'T DO THIS! Hardcoded to just one user's preferences
STATICFILES_DIRS = ['/Users/pydanny/twoscoops_project/static']

# Configuring TEMPLATES
# DON'T DO THIS! Hardcoded to just one user's preferences
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': ['/Users/pydanny/twoscoops_project/templates',]
    },
]
```

上面这段代码体现了一种常见陷阱，叫作 hardcoding。它之所以糟糕，是因为据你所知，只有 pydanny（Daniel Feldroy）一个人把自己的电脑目录结构设成了这样。其他任何想照着这个例子做的人，项目都会立刻坏掉，逼得他们不是去改自己的目录结构（通常不会），就是去改 settings 模块以适配自己的偏好，而这又会给其他人，包括 pydanny 自己，带来问题。

不要把路径写死！

要解决路径问题，我们会在基础 settings 模块顶部动态设置一个很直观的项目根变量，名叫 `BASE_DIR`。由于 `BASE_DIR` 是相对于 `base.py` 的位置计算出来的，所以你的项目无论放在哪台开发机器、哪台服务器上的什么位置，都能正常运行。

我们认为，借助 Pathlib 来设置类似 `BASE_DIR` 的值，是最干净的做法。Pathlib 是 Python 3.4 起自带的库，路径计算优雅又清晰：

示例 5.29：使用 Pathlib 发现项目根目录

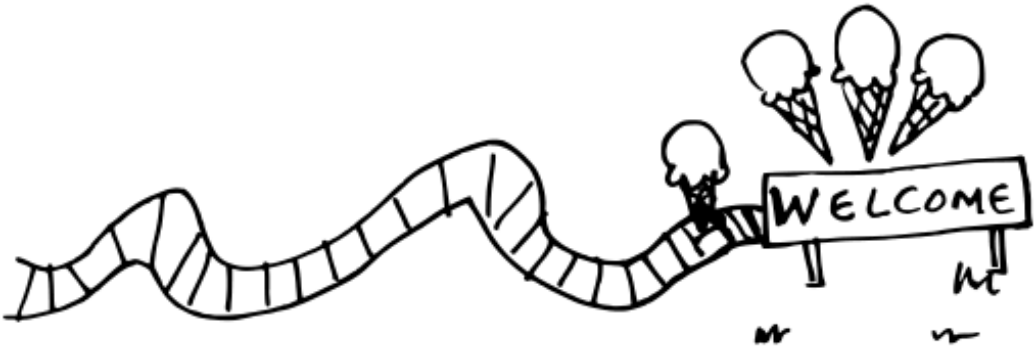

图 5.2: 图 5.2：既然说到了路径，那我们就顺着这条 path 走下去吧。

Code (python)

```
# settings/base.py 顶部
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent.parent.parent
MEDIA_ROOT = BASE_DIR / 'media'
STATIC_ROOT = BASE_DIR / 'static_root'

STATICFILES_DIRS = [BASE_DIR / 'static']
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates']
    },
]
```

如果你真的更想用 Python 标准库里的 os.path 来设置 BASE_DIR，那也可以像下面这样写，同样能正确处理路径：

示例 5.30：使用 os.path 发现项目根目录

Code (python)

```
# settings/base.py 顶部
from os.path import abspath, dirname, join

def root(*dirs):
    base_dir = join(dirname(__file__), '..', '..')
    return abspath(join(base_dir, *dirs))

BASE_DIR = root()
MEDIA_ROOT = root('media')
STATIC_ROOT = root('static_root')
STATICFILES_DIRS = [root('static')]
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [root('templates')],
```

```
    },
]
```

当你的各种路径设置都依赖 `BASE_DIR` 之后，这些文件路径配置通常就能恢复正常，你的 templates 和 media 也就不会再因为路径错误而加载失败了。

提示：你的 Settings 和 Django 默认值到底差了多少？

如果你想知道自己项目里的设置和 Django 默认值到底有哪些不同，可以使用 `diffsettings` management command。

5.7 小结

记住，除了密码和 API key 之外，其他所有东西都应该纳入版本控制。

任何真正要部署到线上生产服务器的项目，最终都离不开多份 settings 文件和多份 requirements 文件。哪怕是 Django 初学者，一旦项目准备离开最初那台开发机器，也迟早会需要这种 `settings/` 与 `requirements/` 布局。

我们给出了自己的方案，也给出了一个对 Apache 友好的方案；无论是初学者还是更资深的开发者，这两套方案都很好用。

另外，如果你偏好使用书中没有列到的其他 shell，也一样可以使用环境变量。你只需要知道对应的定义语法即可。

requirements 文件也是同样的道理。放任依赖差异不受跟踪，带来的风险并不比放任 settings 差异不受跟踪更小。

第六章 模型最佳实践

模型是大多数 Django 项目的基础。写 Django 模型时如果一味求快、不先把事情想清楚，后面往往会出问题。

开发者太常见的一种情况是：还没认真考虑自己到底在做什么，就匆匆加模型、改模型。眼下随手塞进代码库里的那个快速修补，或者那个马虎的“临时”设计决定，几个月甚至几年后都可能反咬我们一口，逼得我们写出疯狂的绕路方案，甚至污染既有数据。

所以，无论是给 Django 新增模型，还是修改已有模型，都请记住这一点：慢一点，先想透，再动手。尽量把你的地基设计得稳固、扎实。

包提示：我们常用的模型相关包

下面快速列一下我们几乎每个项目都会用到的、和模型相关的 Django 包：

- `django-model-utils`：处理 `TimeStampedModel` 这一类常见模式很方便。
- `django-extensions`：提供了一个很强的 management command 叫 `shell_plus`，会自动载入所有已安装 app 的模型类。它的缺点是功能面太宽，不符合我们偏爱“小而专 app”的一贯口味。

6.1 基础

6.1.1 一个 app 里模型太多时，就把它拆开

如果单个 app 里有 20 个以上的模型，就该认真想想有没有办法把它拆成更小的 app，因为这通常意味着这个 app 干了太多事。实际项目里，我们更希望把每个 app 的模型数压到 5 到 10 个以内。

6.1.2 谨慎使用模型继承

Django 里的模型继承是个很微妙的话题。Django 提供了三种模型继承方式：抽象基类、多表继承和代理模型。

警告：Django 的抽象基类不等于 Python 的抽象基类

不要把 Django 的 abstract base class 和 Python 标准库 `abc` 模块里的 abstract base class 混为一谈。它们的目的和行为差别非常大。

下面是三种模型继承风格的优缺点。为了便于完整比较，我们也把“干脆不做模型继承”这个选项一起列进来：

模型继承方式	优点	缺点
不做模型继承：如果两个模型有共同字段，就分别在两个模型里都定义出来	最容易一眼看清 Django 模型如何映射到数据库表	如果多个模型里重复字段很多，维护会比较痛苦
抽象基类：只为派生模型创建数据表	把公共字段放进抽象父类后，不必一遍遍重复敲同样代码	父类本身不能单独使用
多表继承：父类和子类都会建表，父子之间通过隐式 `OneToOneField` 关联	每个模型都有自己的表，因此父类和子类都能单独查询；也可以通过 `parent.child` 从父对象访问子对象	每次查询子表都需要和全部父表做连接，开销很大；我们强烈不建议使用多表继承
代理模型：只为原始模型建表	不会引入多表继承的额外表和连接开销；还可以让同一个模型拥有不同的 Python 层行为别名	不能修改模型字段

表 6.1：模型继承方式的优缺点

警告：避开多表继承

多表继承有时也叫“concrete inheritance”。在作者以及很多开发者看来，它是个坏东西。我们强烈建议不要用它。后面我们还会更详细说明。

下面是一些简单的经验判断，帮你决定该用哪种继承方式、又该在什么时候用：

- 如果模型之间的重叠很小，例如只有一两个模型共享一两个同名字段，那可能根本不需要模型继承，直接在两个模型里都加上那些字段就行。
- 如果模型之间重叠足够多，重复字段已经让维护变得混乱，也容易引入无心之错，那多数情况下都应该重构，把公共字段抽到一个抽象基模型里。
- 代理模型是一个偶尔很实用的便利特性，但它和前两种模型继承风格是非常不同的东西。
- 无论如何，大家都该尽量避开多表继承，因为它既增加理解成本，也带来可观开销。与其用多表继承，不如用显式的 `OneToOneField` 和 `ForeignKey` 在模型之间建立关系，这样你能自己控制什么时候跨表 join。我们两个人合起来做了 20 多年 Django，从没见过多表继承带来过除麻烦之外的任何东西。

6.1.3 实战中的模型继承：TimeStampedModel

在 Django 项目里，几乎总会想给所有模型都带上 `created` 和 `modified` 这两个时间戳字段。你当然可以手动把它们加到每个模型里，但这既费工夫，也会增加人为出错的风险。更好的办法，是写一个 `TimeStampedModel` 帮我们完成这件事：

示例 6.1：`core/models.py`

Code (python)

```
from django.db import models

class TimeStampedModel(models.Model):
    """
    An abstract base class model that provides self-
    updating ``created`` and ``modified`` fields.
    """
    created = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True
```

请特别注意示例最后两行，它们正是把这个例子变成抽象基类的关键：

示例 6.2：定义抽象基类

Code (python)

```
class Meta:
    abstract = True
```

把 `TimeStampedModel` 定义成抽象基类后，当我们定义一个继承自它的新类时，Django 在运行 `migrate` 时就不会创建 `core_timestampedmodel` 这张表。

我们来试试看：

示例 6.3：`flavors/models.py`

Code (python)

```
# flavors/models.py
from django.db import models
```

```
from core.models import TimeStampedModel

class Flavor(TimeStampedModel):
    title = models.CharField(max_length=200)
```

这样只会创建一张表：flavors_flavor。这正是我们想要的行为。

反过来说，如果 TimeStampedModel 不是抽象基类，而是一个具体基类，也就是采用多表继承，那么它还会额外创建一张 core_timestampedmodel 表。不止如此，它的所有子类，包括 Flavor，都会缺少这些字段，并通过隐式外键回指 TimeStampedModel 来处理 created / modified 时间戳。任何对 Flavor 的读写，只要触及 TimeStampedModel，都会连带影响两张表。（幸好它是抽象的！）

记住，具体继承非常容易变成丑陋的性能瓶颈。对同一个具体模型反复多层继承时，这个问题会更加严重。

延伸阅读：

- docs.djangoproject.com/en/3.2/topics/db/models/#model-inheritance

6.2 数据库迁移

Django 自带了一套很强大的数据库变更传播工具，名字也起得很贴切，叫“migrations”；不过在这本书里，我们更愿意把它称作 django.db.migrations。

6.2.1 创建迁移时的小建议

- 新 app 或新模型一创建出来，就多花一分钟把它的初始 django.db.migrations 建好。我们通常只需要敲一句：python manage.py makemigrations。
- 在真正运行迁移前，先看看自动生成的 migration 代码，尤其是在改动复杂时。同时，也用 sqlmigrate 命令审一遍最终会执行的 SQL。
- 对那些自己没有 django.db.migrations 风格迁移文件的第三方 app，可以用 MIGRATION_MODULES 设置来管理它们的迁移编写位置。
- 不必太担心迁移文件数量。如果数量变得难以管理，就用 squashmigrations 来收拢它们。
- 运行迁移前，一定先备份数据。

6.2.2 在迁移里加入 Python 函数和自定义 SQL

django.db.migrations 没法预判你的复杂数据变更，也没法预判那些会和数据交互的外部组件变化。遇到这种情况时，就很适合深入一点，写 Python 或自定义 SQL 来辅助迁移。在任何一个真正上过生产的项目里，你迟早都会遇到需要使用 RunPython 或 RunSQL 的时候：

- docs.djangoproject.com/en/3.2/ref/migration-operations/#runpython
- docs.djangoproject.com/en/3.2/ref/migration-operations/#runsql

如果非要说我们的个人偏好，我们会先选 RunPython，再考虑 RunSQL。不过总的建议还是：优先站在你最擅长的那一边。

6.3 解决 RunPython 的常见障碍

当我们编写供 RunPython 调用的函数时，经常会遇到几个痛点。它们大多都能解决，但不是全部都能。

6.3.1 获取自定义模型 manager 的方法

有时你会想通过自定义模型 manager 的方法去过滤、排除、创建或修改记录。但默认情况下，`django.db.migrations` 会把这些组件排除掉。好在，我们可以给自定义 manager 加一个 `use_in_migrations = True` 标记来覆盖这个行为。

参见：`docs.djangoproject.com/en/3.2/topics/migrations/#model-managers`

6.3.2 获取自定义模型方法

由于 `django.db.migrations` 序列化模型的方式所限，这个限制绕不过去。你就是没法在迁移期间调用任何自定义模型方法。相关说明见下方参考链接：

`docs.djangoproject.com/en/3.2/topics/migrations/#historical-models`

警告：小心自定义的 `save()` 和 `delete()` 方法

如果你覆写了模型的 `save()` 和 `delete()` 方法，那么在 RunPython 调用时，它们并不会被触发。先提醒到这里，别等它坑了你再说，这种陷阱的破坏性可能非常大。

6.3.3 用 RunPython.noop 什么也不做

为了让反向迁移能够工作，RunPython 必须接收一个 `reverse_code` 可调用对象，用来撤销 `code` 可调用对象造成的效果。不过，我们写的某些 `code` 函数本身就是幂等的。比如，它们只是把既有数据合并进一个新加字段。为这种函数专门写 `reverse_code`，要么不可能，要么毫无意义。遇到这种情况时，就把 `RunPython.noop` 用作 `reverse_code`。

例如，假设我们新建了一个叫“Cone”的模型。所有现有的 scoop 都需要对应自己的 cone，于是我们写一个 `add_cones` 函数来把这些 cone 加进数据库。但当迁移回滚时，再写代码去删除这些 cone 就没有意义了；因为 `migrations.CreateModel.database_backwards` 本来就会帮我们删掉 `cone.cone` 表以及其中的所有记录。因此，这里应该把 `RunPython.noop` 用作 `reverse_code`：

示例 6.4：用 `RunPython.noop` 处理回滚

Code (python)

```
from django.db import migrations, models

def add_cones(apps, schema_editor):
    Scoop = apps.get_model('scoop', 'Scoop')
    Cone = apps.get_model('cone', 'Cone')

    for scoop in Scoop.objects.all():
        Cone.objects.create(
            scoop=scoop,
            style='sugar'
        )

class Migration(migrations.Migration):

    initial = True

    dependencies = [
        ('scoop', '0051_auto_20670724'),
    ]
```

```
    operations = [
        migrations.CreateModel(
            name='Cone',
            fields=[
                ('id', models.AutoField(
                    auto_created=True,
                    primary_key=True,
                    serialize=False,
                    verbose_name='ID'
                )),
                ('style', models.CharField(
                    max_length=10,
                    choices=[('sugar', 'Sugar'), ('waffle', 'Waffle')]
                )),
                ('scoop', models.OneToOneField(
                    null=True,
                    to='scoop.Scoop',
                    on_delete=django.db.models.deletion.SET_NULL,
                )),
            ],
        ),
        # RunPython.noop does nothing but allows reverse migrations to occur
        migrations.RunPython(add_cones, migrations.RunPython.noop)
    ]
```

6.3.4 迁移的部署与管理

- 这句话虽然几乎不用说，但我们还是要说：运行迁移前，一定先备份数据。
- 部署前先确认：你能不能回滚迁移！我们不一定总能做到完美往返，但如果完全没法回到更早的状态，排查 bug 会非常痛苦，在大型项目中甚至会直接拖垮部署。
- 如果项目里有些表已经有数百万行数据，那就在生产环境真正运行迁移前，先去 staging 服务器上针对同等数据量做足测试。面对真实数据时，迁移耗时很可能会比你预想的长得多，多很多，真的多很多。
- 如果你在使用 MySQL：
- 做任何 schema 变更前，都必须百分之百先备份数据库。MySQL 在 schema 变更周围缺乏事务支持，因此回滚是不可能的。
- 如果能做到，在执行变更前先把项目切到只读模式。
- 如果不够谨慎，在大表上改 schema 可能会耗费极长时间。不是几秒，也不是几分钟，而是几小时。

提示：数据迁移代码一定要进版本控制

把 migration 代码放进版本控制绝对不是可选项。不把 migration 代码纳入 VCS，和不把 settings 文件纳入 VCS 没什么两样：你也许还能在当前机器上开发，但只要换台机器，或者把其他人拉进项目，一切都会立刻崩掉。

6.4 Django 模型设计

“如何设计出好的 Django 模型”是最难的话题之一，却偏偏又最缺少关注。

怎样才能既为性能做设计，又不至于过早优化？下面我们就来看看一些策略。

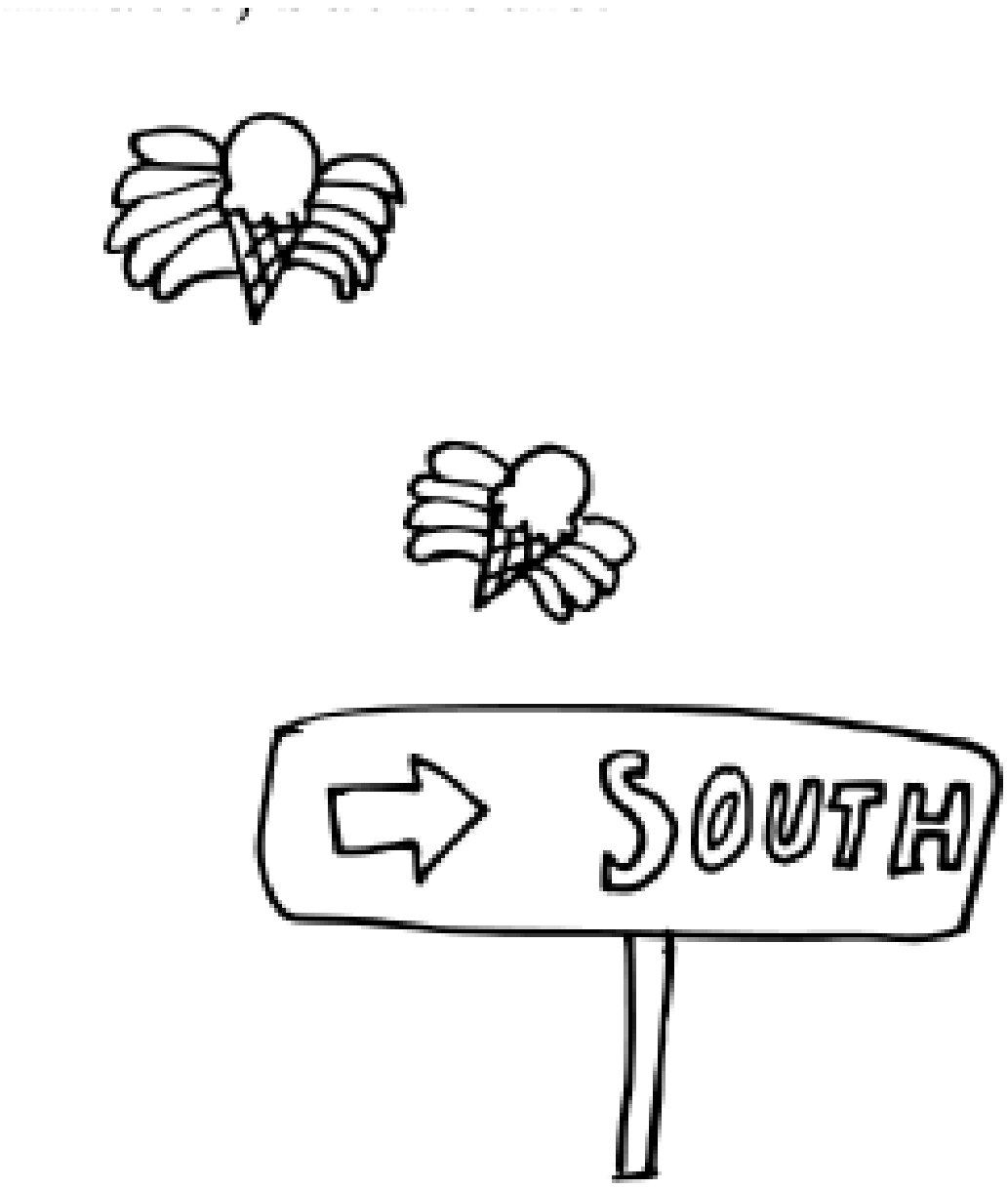

图 6.1: 图 6.1：Cone 们为了过冬向南迁徙。Django 内建迁移系统最早是一个名为 South 的外部项目。

6.4.1 从规范化开始

我们认为，读这本书的人应该熟悉数据库规范化。如果你还不熟，那就请把“补上这块知识”当成自己的责任，因为想要高效地在 Django 里处理模型，必须对这件事有基本理解。这个主题的完整展开超出了本书范围，所以我们建议参考以下资源：

- `en.wikipedia.org/wiki/Database_normalization`
- `en.wikibooks.org/wiki/Relational_Database_Design/Normalization`

设计 Django 模型时，一开始就应该按规范化思路来。花时间确认：某个模型里是否不该再存放另一个模型已经保存过的数据。

在这个阶段，大胆使用关系字段。不要过早反规范化。你首先要对自己的数据形状形成清晰认识。

6.4.2 在反规范化之前，先尝试缓存

很多时候，只要在合适的位置配好缓存，就足以让你免于反规范化模型。我们会在第 26 章《发现并减少瓶颈》中更详细地讲缓存，所以现在先不用太担心细节。

6.4.3 只有在绝对必要时才反规范化

对那些刚接触数据规范化概念的人来说，过早反规范化是很有诱惑力的。别这么做。反规范化看起来好像能一劳永逸地解决项目问题，但它其实是个很棘手的过程，不仅会给项目增加复杂度，还会大幅提高数据丢失风险。

拜托，拜托，拜托，在反规范化之前，先把缓存方案认真走一遍。

当一个项目已经把第 26 章《发现并减少瓶颈》中提到的技术手段都用到极限了，再去研究数据库反规范化的概念和模式，才是更合适的时机。

6.4.4 什么时候用 `null` 和 `blank`

定义模型字段时，你可以设置 `null=True` 和 `blank=True`。默认情况下，它们都是 `False`。

什么时候该用这些选项，是开发者非常常见的困惑来源。为此，我们整理了表 6.2《按字段类型看 `null` 与 `blank` 的使用时机》，作为标准用法参考。

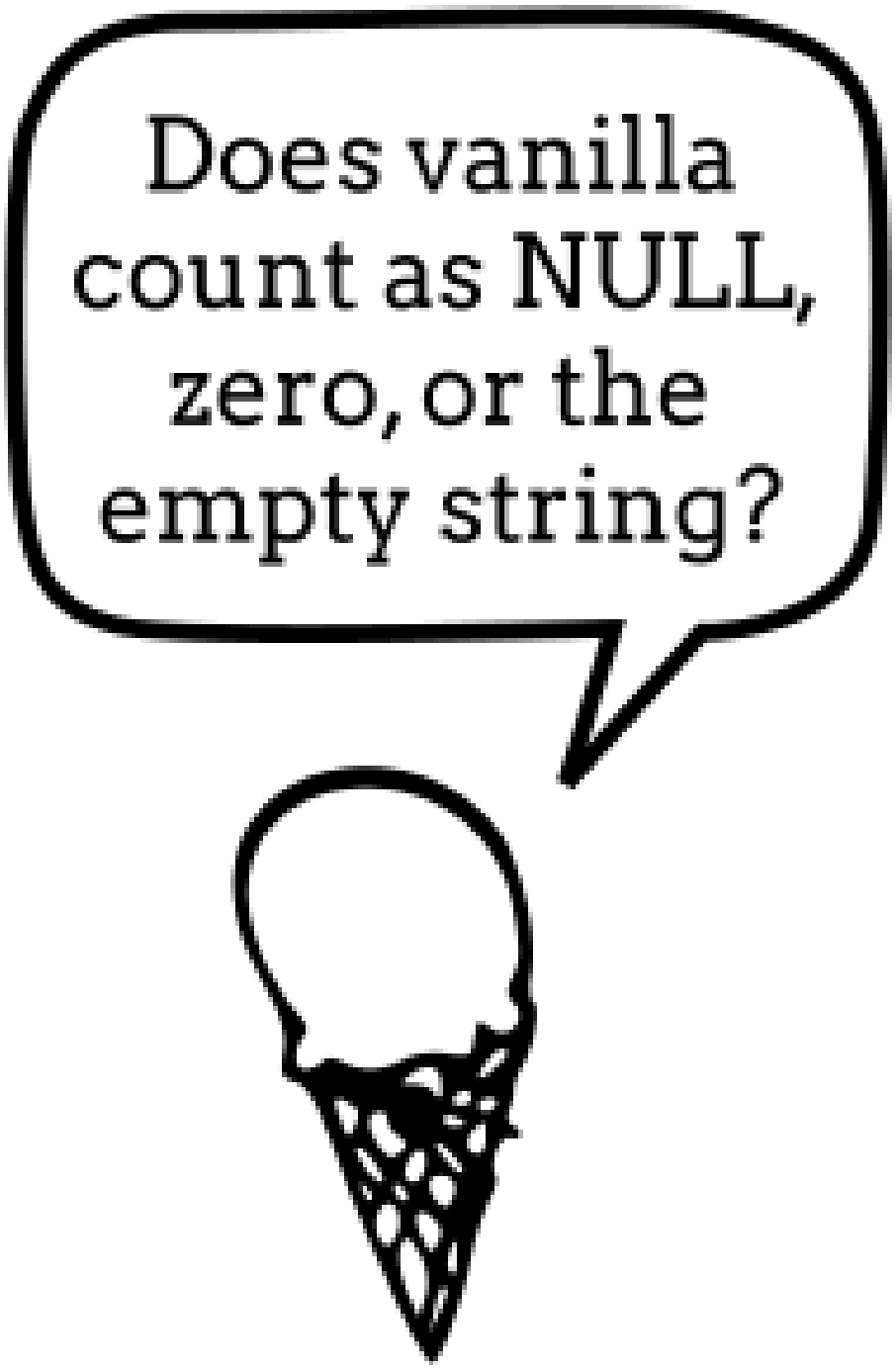

图 6.2: 图 6.2：一个非常常见的困惑来源。

字段类型	null=True 何时使用	blank=True 何时使用
CharField、TextField、SlugField、EmailField、CommaSeparatedIntegerField、UUIDField	如果同时设置了 unique=True 和 blank=True，那么可以用；在这种情况下，为了避免多个空值对象触发唯一约束冲突，往往必须加上 null=True	如果你希望对应的表单控件允许空值，就可以设置；若同时还有 null=True 和 unique=True，空值会以 NULL 存进数据库，否则会以空字符串存储
FileField、ImageField	不要这样做。Django 实际上是把从 MEDIA_ROOT 到文件或图片的路径存进一个 CharField，因此它遵循和 CharField 相同的模式	可以
BooleanField	一般不要这样做	默认就是 blank=True
IntegerField、FloatField、DecimalField、DurationField 等	如果你希望数据库中允许 NULL，可以设置	如果你希望对应表单控件允许空值，也可以设置；这时通常也应该一起设 null=True
DateTimeField、DateField、TimeField 等	如果你希望数据库中允许 NULL，可以设置	如果你希望表单控件允许空值，或者你使用了 auto_now / auto_now_add，就可以设置；前一种情况通常也要同时设 null=True
ForeignKey、OneToOneField	如果你希望数据库中允许 NULL，可以设置	如果你希望对应的表单控件（例如下拉选择框）允许空值，也可以设置；这时通常也要同时设 null=True
ManyToManyField	null 在这里没有效果	如果你希望对应的表单控件（例如多选框）允许空值，就设置
GenericIPAddressField	如果你希望数据库中允许 NULL，可以设置	如果你希望对应字段控件允许空值，通常也要同时设 null=True
JSONField	可以	可以

表 6.2：按字段类型看 null 与 blank 的使用时机

6.4.5 什么时候用 BinaryField

这个字段允许你存放原始二进制数据，也就是 bytes。你不能对它执行过滤、排除或其他 SQL 操作，但它确实有自己的使用场景。例如，你可以拿它来存：

- MessagePack 格式内容。
- 原始传感器数据。
- 压缩数据，例如 Sentry 那类会以 BLOB 形式存储、但又因为历史原因必须做 base64 编码的数据。

可能性很多，但也别忘了：二进制数据块可能非常大，而这会拖慢数据库。如果这件事真的演变成瓶颈，那么解决办法可能是把二进制数据保存成文件，再用 FileField 引用它。

警告：不要用 BinaryField 来提供文件服务！

永远不要把“把文件塞进数据库字段里”当作正常做法。如果它看上去像某个问题的解决方案，那你最好去找一位靠谱的数据库专家，再听一次第二意见。

用 PostgreSQL 专家 Frank Wiles 的话来概括，把数据库当文件存储会有这些问题：

- “对数据库的读写永远比文件系统慢。”
- “数据库备份会变得巨大，而且更耗时。”
- “访问这些文件时，你现在必须穿过应用（Django）和数据库两层。”

参见：revsys.com/blog/2012/may/01/three-things-you-should-never-put-your-database/

当有人觉得“从数据库里提供文件服务”是个好主意，并搬出 npmjs.org 这种成功案例来佐证时，你就该

开始认真做研究了。事实是：npmjs.org 早在很多年前，就已经把那套“数据库即文件存储”的系统迁移回更传统的文件服务方式了。

6.4.6 尽量避免使用泛型关联

总的来说，我们并不主张使用 generic relations，也不主张使用 models.field.GenericForeignKey。它们带来的麻烦通常远多于价值。很多时候，一旦你在用它，就意味着你走了麻烦的捷径，或者说，你正在探索的本来就是错误方案。

泛型关联的核心想法，是通过一个不受约束的外键（GenericForeignKey）把一张表绑定到另一张表上。这种做法很像：你明明做的是非常需要外键约束的项目，却偏偏拿一个缺少外键约束的 NoSQL 存储当基础。结果就是：

- 因为模型之间缺少索引关系，查询速度会下降。
- 一张表可能引用到另一张表中并不存在的记录，于是数据损坏风险上升。

当然，没有约束也并非全无好处。generic relations 确实会让你更容易构建那些需要同时和大量模型类型交互的 app，尤其是 favorites、ratings、voting、messages、tagging 这类场景。的确，也有不少现成 app 是按这种方式构建的。虽然我们自己仍然会犹豫是否采用，但只要那些 app 专注于单一任务（例如 tagging），多少会让人安心一点。

随着时间推移，我们发现 favorites、ratings、voting、messages、tagging 这类 app，其实都能基于 ForeignKey 和 ManyToManyField 做出来。只要多投入一点开发成本，避开 GenericForeignKey 就能换来速度和数据完整性的收益。

GenericForeignKey 真正变得棘手的时候，是当它那种“不受约束”的特性，被拿来定义项目的核心数据时。比如，如果我们做一个冰淇淋主题项目，把 toppings、flavors、containers、orders 和 sales 之间的关系全都用 GenericForeignKey 来跟踪，那就会遇到前面列出的所有问题。简而言之：

- 尽量避免 generic relations 和 GenericForeignKey。
- 如果你觉得自己“必须”用 generic relations，先看看能不能通过更好的模型设计，或者新的 PostgreSQL 字段来解决问题。
- 如果实在避不开，尽量选一个现成的第三方 app。第三方 app 提供的隔离层，会帮助你把数据维持得更干净。

如果你想看一个和我们立场一致的观点，也请读一读：lukeplant.me.uk/blog/posts/avoid-django-genericforeignke

6.4.7 把 choices 和 sub-choices 做成模型常量

一个很好的模式，是把 choices 定义成模型属性，并用 tuple 结构组织。既然这些值是和模型本身、以及它所表达的数据强绑定的常量，那么能在任何地方轻松访问它们，会让开发过程顺手得多。

这个技巧在 docs.djangoproject.com/en/3.2/ref/models/fields/#django.db.models.Field.choices 里有说明。如果把它翻译成冰淇淋示例，就是这样：

示例 6.5：设置 choice 模型属性

Code (python)

```
# orders/models.py
from django.db import models

class IceCreamOrder(models.Model):
    FLAVOR_CHOCOLATE = 'ch'
    FLAVOR_VANILLA = 'vn'
    FLAVOR_STRAWBERRY = 'st'
    FLAVOR_CHUNKY_MUNKY = 'cm'
```

```
    FLAVOR_CHOICES = (
        (FLAVOR_CHOCOLATE, 'Chocolate'),
        (FLAVOR_VANILLA, 'Vanilla'),
        (FLAVOR_STRAWBERRY, 'Strawberry'),
        (FLAVOR_CHUNKY_MUNKY, 'Chunky Munky')
    )

    flavor = models.CharField(
        max_length=2,
        choices=FLAVOR_CHOICES
    )
```

有了这个模型之后，我们就可以这样写：

示例 6.6：访问 choice 模型属性

Code (python)

```
>>> from orders.models import IceCreamOrder
>>> IceCreamOrder.objects.filter(
...     flavor=IceCreamOrder.FLAVOR_CHOCOLATE
... )
[<icecreamorder: 35>, <icecreamorder: 42>, <icecreamorder: 49>]
```

它在 Python 代码和模板里都能工作，而且这些属性既可以通过类访问，也可以通过实例化后的模型对象访问。

6.4.8 用枚举类型定义 choices

Nate Cox 建议在 choices 上使用 Django 的枚举类型。自 Django 3.0 起，这个能力已经内建了，用起来更轻松。

示例 6.7：设置 choice 枚举属性

Code (python)

```
from django.db import models

class IceCreamOrder(models.Model):
    class Flavors(models.TextChoices):
        CHOCOLATE = 'ch', 'Chocolate'
        VANILLA = 'vn', 'Vanilla'
        STRAWBERRY = 'st', 'Strawberry'
        CHUNKY_MUNKY = 'cm', 'Chunky Munky'

    flavor = models.CharField(
        max_length=2,
        choices=Flavors.choices
    )
```

有了这段代码，我们就能这样写：

示例 6.8：访问基于枚举的 choice 模型属性

Code (python)

```
>>> from orders.models import IceCreamOrder
>>> IceCreamOrder.objects.filter(
...     flavor=IceCreamOrder.Flavors.CHOCOLATE
... )
[<icecreamorder: 35>, <icecreamorder: 42>, <icecreamorder: 49>]
```

和上一种做法相比，使用枚举类型也有几个缺点。具体来说：

- 枚举类型不支持命名分组。也就是说，如果你希望在 choices 里再分层分组，就得回退到旧式 tuple 写法。
- 如果你需要的不只是 `str` 和 `int` 这类类型，那就得自己额外定义。

枚举类型为 choice 字段提供了一套很漂亮的 API。我们的经验是：先用它，等真的撞上上面这些限制，再退回老式 tuple 方案。

6.4.9 PostgreSQL 专属字段：什么时候用 `null` 和 `blank`

字段类型	`null=True` 何时使用	`blank=True` 何时使用
`ArrayField`	可以	可以
`HStoreField`	可以	可以
`IntegerRangeField`、`BigIntegerRangeField`、`FloatRangeField`	如果你希望数据库中允许 `NULL`，可以设置	如果你希望对应表单控件允许空值，也应该同时设 `null=True`
`DatetimeRangeField`、`DateRangeField`	如果你希望数据库中允许 `NULL`，可以设置	如果你希望对应表单控件允许空值，或者你使用了 `auto_now` / `auto_now_add`，通常也要一起设 `null=True`

表 6.3：PostgreSQL 字段里 `null` 与 `blank` 的使用时机

6.5 模型 _meta API

这个 `_meta` API 有两个不太寻常的地方：

- 它明明以前缀 `_` 开头，却是一个公开、文档化的 API。
- 和 Django 里其他以下划线开头的组件不同，`_meta` 也遵循框架其余部分一样的弃用规则。

原因在于：在 Django 1.8 之前，模型 `_meta` API 还不是官方 API，也刻意没有文档，因为这类 API 通常可能在没有预告的情况下直接变化。最初，`_meta` 只是 Django 用来为模型存一些额外信息、供自己内部使用的地方。但后来它被证明太有用了，于是就成了正式文档 API。

对大多数项目来说，你其实用不到 `_meta`。它的主要用途一般集中在这些场景：

- 获取某个模型的字段列表。
- 获取模型某个特定字段的类，或者其继承链、及相关派生信息。
- 确保你获取这些信息的方式，在未来 Django 版本中仍然保持稳定。

这类场景的例子包括：

- 构建 Django 模型自省工具。
- 构建你自己的定制化 Django 表单库。
- 创建类似 admin 的工具，用来编辑或操作 Django 模型数据。
- 编写可视化或分析类库，例如分析所有以 `foo` 开头的字段。

延伸阅读：

- 模型 _meta 文档：docs.djangoproject.com/en/3.2/ref/models/meta/

6.6 模型 Managers

每次我们用 Django ORM 查询某个模型时，实际上都是在通过一个叫作 model manager 的接口和数据库交互。人们常说，model manager 是站在“该模型类所有可能实例的全集”之上工作的，也就是先面对整张表，再从中筛出你真正想操作的那一部分。Django 会为每个模型类提供一个默认 manager，不过我们也可以自己定义。

下面是一个自定义 model manager 的简单例子：

示例 6.9：自定义 model manager：published

Code (python)

```
from django.db import models
from django.utils import timezone

class PublishedManager(models.Manager):

    def published(self, **kwargs):
        return self.filter(pub_date__lte=timezone.now(), **kwargs)

class FlavorReview(models.Model):
    review = models.CharField(max_length=255)
    pub_date = models.DateTimeField()

    # add our custom model manager
    objects = PublishedManager()
```

如果我们先想显示全部冰淇淋口味评论的总数，再显示其中已发布评论的数量，就可以这么做：

示例 6.10：自定义 model manager：published

Code (python)

```
>>> from reviews.models import FlavorReview
>>> FlavorReview.objects.count()
>>> FlavorReview.objects.published().count()
```

很简单，对吧？不过你也许会想：如果干脆再加第二个 model manager，不是更合理吗？那样你就能写出类似这样的东西：

示例 6.11：使用两个 model manager 的“表面优势”

Code (python)

```
>>> from reviews.models import FlavorReview
>>> FlavorReview.objects.filter().count()
>>> FlavorReview.published.filter().count()
```

表面上看，替换默认 model manager 好像很聪明。但根据我们在真实项目里的经验，这种做法必须非常谨慎。为什么？

第一，使用模型继承时，抽象基类的子类会继承父类的 model manager，而具体基类的子类不会。

第二，应用到模型类上的第一个 manager，会被 Django 视为默认 manager。它和普通 Python 的默认模式差异非常大，因此很容易让 QuerySet 的结果表现出一种“看起来不可预测”的效果。

知道这一点后，在你的模型类中，objects = models.Manager() 应该手动定义在任何自定义 manager 之前。

警告：搞清楚 model manager 的执行顺序

凡是自定义 manager 采用了新名字，都应该把 `objects = models.Manager()` 放在它上面。这个规则并非绝对不可打破，但那是高级技巧，我们不建议大多数项目这么做。

延伸阅读：`docs.djangoproject.com/en/3.2/topics/db/managers/`

6.7 理解 Fat Models

fat models 的思想是：不要把和数据有关的代码塞进 views 和 templates，而是把逻辑封装到模型方法、`classmethod`、属性，甚至 manager 方法里。这样一来，无论是 view 还是 task，都能复用同一套逻辑。举例来说，如果我们有一个表示冰淇淋评论的模型，可能会给它挂上这样的几个方法：

- `Review.create_review(cls, user, rating, title, description)`：一个创建评论的 `classmethod`。HTML view、REST view，以及接收电子表格导入的工具，都可以直接在模型类上调用它。
- `Review.product_average`：评论实例上的一个属性，返回被评论产品的平均评分。评论详情页可以借此让读者不用离开当前页面，就对整体口碑有个概念。
- `Review.found_useful(self, user, yes)`：一个方法，用来记录读者是否觉得这条评论有帮助。详情页和列表页，无论是 HTML 还是 REST 实现，都可以复用它。

从这个列表就能看出，fat models 确实是提升项目内代码复用性的好办法。事实上，把逻辑从 views 和 templates 挪到 models 里，这种实践这些年已经在很多项目、框架和语言中不断增长。这难道不是件好事吗？

不一定。

把所有逻辑都塞进模型里的问题在于：模型代码体积很容易膨胀，最后长成所谓的“god object”反模式。于是，一个模型类可能会变成几百行、几千行，甚至上万行代码。因为体积巨大、复杂度又高，这种 god object 很难理解，自然也就很难测试、很难维护。

把逻辑往模型里迁移时，我们会尽量记住面向对象编程的一个基本思想：大问题拆成小问题，才更容易解决。如果一个模型开始变得笨重，我们就会着手把那些适合在其他模型中复用的逻辑，或者那些复杂到必须更好管理的逻辑，拆出去。方法、`classmethod` 和属性本身仍然保留，但其中的具体逻辑会被搬到 Model Behaviors 或无状态辅助函数里。下面两小节就讲这两种技术：

6.7.1 Model Behaviors，也就是 Mixins

Model behaviors 体现的是组合与封装思想，具体做法是使用 mixins。模型通过继承抽象模型来获得逻辑。更多内容参见：

- `blog.kevinastone.com/django-model-behaviors.html`：Kevin Stone 关于如何用组合降低重复代码的文章。
- 10.2 节《在 CBV 中使用 Mixins》。

6.7.2 无状态辅助函数

把逻辑从模型中移到工具函数里，会让逻辑本身更独立。而这种隔离，又会让这部分逻辑更容易编写测试。它的缺点是：函数是无状态的，所以所有参数都必须显式传入。

我们会在第 31 章《那些零散工具函数怎么办？》里继续讲这个问题。

6.7.3 Model Behaviors 与 Helper Functions

在我们看来，单独使用这两种技术中的任何一种，都不算完美。不过，只要使用得当，它们组合起来就能让项目非常出彩。什么时候该用哪一种，并不是静态科学，而是一个会不断演化的过程。正因为这种演化本身就很微妙，我们才特别建议你为 fat models 的相关组成部分写好测试。

6.8 额外资源

下面这些文章，都是对本章内容的有益延伸：

- `hakibenita.com/bullet-proofing-django-models`：Haki Benita 关于如何为 Django 模型建立非常严格规则的精彩文章。他整个人的博客都非常值得读。
- Andrew Brooks 对 Django ORM 的深度剖析，也很值得任何关注模型设计性能的人读一读：`spellbookpress.com/books/t`

6.9 小结

模型是大多数 Django 项目的地基，所以一定要花时间认真设计。

先从规范化开始。只有在你已经彻底探索过其他选项之后，才考虑反规范化。你也许可以通过下探到 raw SQL 来简化缓慢而复杂的查询，也可能只要在合适位置加入缓存，就足以解决性能问题。

如果你决定使用模型继承，请继承抽象基类，而不是具体模型。这样能帮你避开那些隐式而多余的 join 带来的混乱。

使用模型字段选项 `null=True` 和 `blank=True` 时，要小心那些“坑点”。拿不准时，就回头查我们那张顺手好用的表。

你可能会发现 `django-model-utils` 和 `django-extensions` 都挺顺手。

最后，fat models 确实能帮助你把逻辑封装进模型，但它们也极其容易失控，长成 god objects。

下一章，我们就会开始谈 queries 和数据库层。

第七章 查询与数据库层

我们写的大多数查询其实都很简单。Django 的对象关系映射，也就是 ORM，提供了一个很棒的效率捷径：它不仅能为常见场景生成足够像样的 SQL 查询，还同时提供了带验证与安全保护的模型访问 / 更新能力。它也让我们几乎可以不费力地写出能在不同数据库引擎上工作的代码。Django 第三方包生态中有很大一部分能力，正是靠 ORM 这一特性支撑起来的。如果一个查询用 ORM 就能轻松表达，那就大胆用它！

Django ORM 和任何 ORM 一样，会先把不同类型的数据转换成对象，让我们能在受支持的数据库之间用相当一致的方式操作它们；然后再提供一整套和这些对象交互的方法。总体来说，Django 在它被设计来处理的事情上做得相当不错。不过它也确实有些怪脾气，而理解这些怪脾气，本来就是学会用 Django 的一部分。那我们就来过一遍吧？

7.1 处理单个对象时，用 get_object_or_404()

在详情页这类 view 中，如果你是要取出一个对象并对它进行处理，请用 `get_object_or_404()`，不要直接用 `get()`。

警告：get_object_or_404() 只应该用于 views

- 只在 views 里使用它。
- 不要在 helper function、form、model method，或任何不是 view、也不直接属于 view 相关层的地方使用它。

很多年前，有一位叫 Daniel 的 Python 程序员兼作者，在部署他的第一个 Django 项目时，被 Django 的 `get_object_or_404()` 迷得神魂颠倒，于是到处都在用它：view 里用、model 里用、form 里也用，反正哪儿都用。开发阶段一切顺利，测试也都通过。不幸的是，这种毫无约束的用法意味着：当后台管理人员删除某些记录时，整个网站直接炸掉了。

把 `get_object_or_404()` 留在你的 views 里！

7.2 小心那些可能抛异常的查询

如果你是通过 `get_object_or_404()` 这个快捷方式来获取单个 Django 模型实例，那就不需要再额外包一层 `try-except`。因为 `get_object_or_404()` 已经替你做了这件事。

但在大多数其他场景里，你还是需要使用 `try-except`。下面是一些建议：

7.2.1 ObjectDoesNotExist 与 DoesNotExist

`ObjectDoesNotExist` 可以用于任何模型对象，而 `DoesNotExist` 则只针对某个具体模型。

示例 7.1：`ObjectDoesNotExist` 的用法示例

Code (python)

```python
from django.core.exceptions import ObjectDoesNotExist

from flavors.models import Flavor
from store.exceptions import OutOfStock

def list_flavor_line_item(sku):
    try:
        return Flavor.objects.get(sku=sku, quantity__gt=0)
    except Flavor.DoesNotExist:
        msg = 'We are out of {0}'.format(sku)
```

```
        raise OutOfStock(msg)

def list_any_line_item(model, sku):
    try:
        return model.objects.get(sku=sku, quantity__gt=0)
    except ObjectDoesNotExist:
        msg = 'We are out of {0}'.format(sku)
        raise OutOfStock(msg)
```

7.2.2 本来只想取一个对象，结果却回来了三个

如果你的查询有可能返回不止一个对象，那就记得处理 `MultipleObjectsReturned` 异常。然后在 `except` 子句里，按你的业务需要处理，比如抛出一个特殊异常，或者把错误记进日志。

示例 7.2：`MultipleObjectsReturned` 的用法示例

Code (python)

```
from flavors.models import Flavor
from store.exceptions import OutOfStock, CorruptedDatabase

def list_flavor_line_item(sku):
    try:
        return Flavor.objects.get(sku=sku, quantity__gt=0)
    except Flavor.DoesNotExist:
        msg = 'We are out of {}'.format(sku)
        raise OutOfStock(msg)
    except Flavor.MultipleObjectsReturned:
        msg = 'Multiple items have SKU {}. Please fix!'.format(sku)
        raise CorruptedDatabase(msg)
```

7.3 利用惰性求值，让查询更好读

Django ORM 很强大。而能力越强，也越该承担起让代码保持可读、也就是可维护的责任。遇到复杂查询时，尽量不要把太多功能链式压在寥寥几行里：

示例 7.3：难以阅读的查询

Code (python)

```
# Don't do this!
from django.db.models import Q

from promos.models import Promo

def fun_function(name=None):
    """Find working ice cream promo"""

    # Too much query chaining makes code go off the screen or page. Not good.
    return Promo.objects.active().filter(...)
```

[原文此处查询链示例在 PDF/OCR 提取中残缺，保守保留为“过长链式查询”的反例含义，不补造缺失代码]

读起来很不舒服，对吧？而如果你再把 Django ORM 的高级工具一起加进去，那它就会从“不舒服”迅速升级成“像是浇了是拉差辣酱的冰淇淋配料”一样糟糕。为了缓解这种痛苦，我们可以借助 Django 查询的惰性求值特性，让 ORM 代码保持干净。

所谓惰性求值，是指：在真正需要数据之前，Django ORM 不会发出 SQL 调用。我们可以随意链式拼接 ORM 方法和函数；只要还没开始遍历结果，Django 就不会碰数据库。这样一来，我们就不用被迫把很多方法和高级数据库特性塞进同一行，而是可以按需要拆成多行来写。可读性提升了，维护就更轻松，腾出来吃冰淇淋的时间也就更多了。

下面我们把糟糕示例 7.3 里的代码改写成更易读的形式：

示例 7.4：可读的查询

Code (python)

```
# Do this!
from django.db.models import Q

from promos.models import Promo

def fun_function(name=None):
    """Find working ice cream promo"""
    results = Promo.objects.active()
    results = results.filter(
        Q(name__startswith=name) |
        Q(description__icontains=name)
    )
    results = results.exclude(status='melted')
    results = results.select_related('flavors')
    return results
```

从改正后的代码里可以很清楚地看出最终结果是什么。更妙的是，把查询语句拆开以后，我们还可以针对特定代码行写注释。

7.3.1 为了可读性而链式排版查询

这个技巧借鉴了 Pandas 和 JavaScript 社区的风格。不使用惰性求值分段变量，也可以把查询写成下面这样：

示例 7.5：链式排版查询

Code (python)

```
# Do this!
from django.db.models import Q

from promos.models import Promo

def fun_function(name=None):
    """Find working ice cream promo"""

    qs = (
        Promo
        .objects
        .active()
        .filter(
            Q(name__startswith=name) |
```

```
            Q(description__icontains=name)
        )
        .exclude(status='melted')
        .select_related('flavors')
    )
    return qs
```

这种写法的缺点是：调试起来不如“惰性求值 + 分段赋值”那种方式方便。你没法在这种定义方式的查询中间，直接插进一个 PDB 或 IPDB 调用点。

为了绕开这个问题，你通常就得先注释掉一部分：

示例 7.6：调试链式查询

Code (python)

```
def fun_function(name=None):
    """Find working ice cream promo"""

    qs = (
        Promo
        .objects
        .active()
        # .filter(
        #     Q(name__startswith=name) |
        #     Q(description__icontains=name)
        # )
        # .exclude(status='melted')
        # .select_related('flavors')
    )

    breakpoint()

    return qs
```

7.4 多依靠高级查询工具

Django ORM 很容易学，直觉也不错，而且覆盖了很多使用场景。但它也确实有不少事情做得没那么好。于是常见的结果就是：拿到 queryset 之后，我们开始在 Python 里做越来越多的数据处理。这其实很可惜，因为任何数据库在管理和转换数据这件事上，都比 Python 更快，更别说 Ruby、JavaScript、Go、Java 之类语言了。

与其让 Python 背这个锅，我们总是尽量先用 Django 的高级查询工具来干重活。这样做不仅性能更好，我们也能享受一种额外好处：我们用到的代码，比我们自己造出来的 Python 绕路方案，要成熟、也更经得起验证得多。毕竟 Django 和大多数数据库一直都在被持续测试。

7.4.1 查询表达式

在对数据库执行读取操作时，query expressions 可以让我们在读取过程中直接创建值或进行计算。这个描述听起来如果有点绕，也不用担心，我们自己也觉得它挺绕的。既然一个代码示例胜过千言万语，那就直接上例子。我们的场景是：要找出所有那些“平均每次到店都点了不止一勺冰淇淋”的顾客。

先看一种没用查询表达式、而且颇为危险的写法：

示例 7.7：不使用查询表达式

Code (python)

```
# Don't do this!
from models.customers import Customer

customers = []
for customer in Customer.objects.iterator():
    if customer.scoops_ordered > customer.store_visits:
        customers.append(customer)
```

这个例子会让我们吓得发抖。为什么？

- 它用 Python 把数据库里所有 Customer 记录一条条循环出来，既慢又吃内存。
- 在任何有一定使用量的环境下，它都会制造竞态条件。也就是说，当脚本运行的同时，顾客还在和这份数据交互时，就有机会出现数据竞争。虽然在这个只有“读”的简单示例里，问题可能还不明显；但在现实代码中，一旦再掺上“更新”操作，就可能造成数据丢失。

好在，借助 query expressions，Django 提供了更高效、也不容易引发竞态的写法：

示例 7.8：使用查询表达式

Code (python)

```
from django.db.models import F
from models.customers import Customer

customers = Customer.objects.filter(
    scoops_ordered__gt=F('store_visits')
)
```

这段代码的意思，是直接让数据库自己完成比较。Django 在底层执行出来的 SQL，大致会像这样：

示例 7.9：查询表达式渲染后的 SQL

Code (sql)

```
SELECT * from customers_customer where scoops_ordered > store_visits
```

query expressions 应该成为你工具箱中的常备武器。它们能同时提升项目性能和稳定性。

- docs.djangoproject.com/en/3.2/ref/models/expressions/

7.4.2 数据库函数

从 Django 1.8 开始，我们已经能很方便地使用常见数据库函数，例如 UPPER()、LOWER()、COALESCE()、CONCAT()、LENGTH() 和 SUBSTR()。在 Django 提供的所有高级查询工具里，这一类是我们的最爱。为什么？

1. 非常容易上手，不管是新项目还是旧项目都好用。
2. 数据库函数能把一部分逻辑从 Python 挪回数据库里执行。这常常会带来性能提升，因为用 Python 处理数据，终究没有数据库原生处理那么快。
3. 各数据库对这些函数的具体实现虽然不同，但 Django ORM 会把这层差异抽象掉。所以你在 PostgreSQL 上写的这类代码，通常也能在 MySQL 或 SQLite3 上工作。
4. 它们本身也是 query expressions，因此遵循另一套 Django ORM 已经建立好的统一模式。

参考：

- docs.djangoproject.com/en/3.2/ref/models/database-functions/

7.5 不到必要时，不要下沉到 Raw SQL

每当我们写 raw SQL，就会失去一部分安全性和复用性。这件事不仅影响项目内部代码，也影响整个 Django 生态。尤其是，如果你以后打算把某个 Django app 发布成第三方包，那么使用 raw SQL 会降低这份工作的可移植性。另外，在极少数需要把数据从一种数据库迁移到另一种数据库的情况下，你在 SQL 查询里使用的那些数据库专属特性，也会让迁移变得更麻烦。

那么，什么时候才真的应该写 raw SQL？如果把查询直接写成 raw SQL，能显著简化 Python 代码，或者能显著简化 ORM 生成出来的 SQL，那就写。比如，如果你正在链式堆叠一连串 `QuerySet` 操作，而且每一步都作用在大数据集上，那么直接写 raw SQL 很可能会更高效。

提示：Malcolm Tredinnick 对 Django 中 SQL 的建议

Django 核心开发者 Malcolm Tredinnick 的意思大致是：

“ORM 能做很多美妙的事，但有时 SQL 才是正确答案。Django ORM 的大致定位是：它是一个存储层，只是刚好用 SQL 来实现功能而已。如果你需要写高级 SQL，那就去写。我会补上一句提醒：谨慎，不要过度使用 `raw()` 和 `extra()`。”

提示：Jacob Kaplan-Moss 对 Django 中 SQL 的建议

Django 项目共同领导者 Jacob Kaplan-Moss 的意思大致是：

“如果一个查询用 SQL 写起来比用 Django 更简单，那就直接写 SQL。`extra()` 很糟糕，应该避免；`raw()` 很不错，适合的时候就该用。”

7.6 按需添加索引

虽然给模型字段加上 `db_index=True` 很容易，但要判断“什么时候该加”，还是需要一点经验。我们的偏好是：一开始先不加索引，等真正有需要时再加。

什么时候可以考虑加索引：

- 这个索引会被高频使用，比如出现在 10% 到 25% 的查询里。
- 已经有真实数据，或者至少有足够接近真实数据的样本，这样我们才能分析加索引后的结果。
- 我们可以跑测试，验证索引是否真的带来了改进。

如果你使用的是 PostgreSQL，`pg_stat_activity` 能告诉我们哪些索引实际上正在被使用。

等项目上线之后，第 26 章《发现并减少瓶颈》里还有关于索引分析的更多内容。

提示：基于类的模型索引

Django 提供了 `django.db.models.indexes` 模块、`Index` 类，以及 `Meta.indexes` 选项。借助它们，可以很容易地创建各种数据库索引：只要继承 `Index`，把它加进 `Meta.indexes`，基本就搞定了。`django.contrib.postgres.indexes` 目前已经包含 `BrinIndex` 和 `GinIndex`，你完全可以想象未来还有 `HashIndex`、`GistIndex`、`SpGistIndex` 等等。

- `docs.djangoproject.com/en/3.2/ref/models/indexes/`
- `docs.djangoproject.com/en/3.2/ref/models/options/#indexes`

7.7 事务

ORM 的默认行为是：每次调用查询时都自动提交。在修改数据的场景里，这意味着每次调用 `.create()` 或 `.update()`，都会立刻修改 SQL 数据库中的数据。它的好处，是让初学者更容易理解 ORM；坏处则是：如果某个视图（或其他操作）需要执行两次及以上数据库修改，而其中一次成功、另一次失败，数据库就有被破坏的风险。

解决数据库损坏风险的方法，就是使用数据库事务。所谓事务，是指把两次或更多次数据库更新打包成一个工作单元。如果其中任意一次更新失败，事务中的全部更新都会回滚。要做到这一点，事务按定义必须具备

图 7.1: 图 7.1：这种口味的冰淇淋里加了 raw SQL。嚼起来还有点费劲。

原子性、一致性、隔离性和持久性。数据库从业者通常会用缩写 ACID 来指代这几个特性。

Django 提供了一套强大且相对容易使用的事务机制。借助装饰器和 context manager，它能让你以一种相当直观的模式，把项目中的数据库完整性保护得更严实。

7.7.1 把每个 HTTP 请求都包在事务里

示例 7.10：把每个 HTTP 请求包进事务

Code (python)

```
# settings/base.py

DATABASES = {
    'default': {
        # ...
        'ATOMIC_REQUESTS': True,
    },
}
```

Django 借助 ATOMIC_REQUESTS 设置，让你很容易把所有 Web 请求都放进事务里处理。像上面这样把它设为 True 后，所有请求都会被事务包裹起来，包括那些只读数据的请求。这样做的好处是安全：view 中所有数据库查询都受到保护；坏处是性能可能会受影响。这个影响到底有多大，我们没法直接告诉你，因为它取决于具体的数据库设计，也取决于不同数据库引擎处理锁的能力。

我们的经验是：对于写操作很多的项目来说，这是一种很好的起步方式，可以先确保数据库完整性不出问题。不过在高流量场景下，我们后来也不得不回头把它改成更聚焦的做法。这个调整工作，视项目体量不同，可能只是小修，也可能是大工程。

使用 ATOMIC_REQUESTS 时，另一个要记住的点是：出错时只有数据库状态会被回滚。你先发出了一封确认邮件，然后包住整个请求的事务却回滚了，这种场面会很尴尬。任何“写入数据库之外东西”的行为都可能踩到这个坑，例如发送邮件或短信、调用第三方 API、写入文件系统，等等。因此，当你在写那些会创建 / 更新 / 删除记录、同时又会和非数据库对象交互的 views 时，你也许会选择给该 view 加上 transaction.non_atomic_requests() 装饰器。

警告：Aymeric Augustin 对 non_atomic_requests() 的看法

Django 核心开发者、也是新事务系统主要实现者 Aymeric Augustin 的原意是：“这个装饰器要求 views 和 models 之间紧密耦合，这会让代码库更难维护。如果不是为了向后兼容，我们本来或许能设计出更好的方案。”

然后，你就可以像下面这个极简 API 风格的函数视图一样，使用更显式的声明方式：

示例 7.11：简单的非原子视图

Code (python)

```
# flavors/views.py

from django.db import transaction
from django.http import HttpResponse
from django.shortcuts import get_object_or_404
from django.utils import timezone

from .models import Flavor

@transaction.non_atomic_requests
def posting_flavor_status(request, pk, status):
```

```
    flavor = get_object_or_404(Flavor, pk=pk)

    # This will execute in autocommit mode (Django's default).
    flavor.latest_status_change_attempt = timezone.now()
    flavor.save()

    with transaction.atomic():
        # This code executes inside a transaction.
        flavor.status = status
        flavor.latest_status_change_success = timezone.now()
        flavor.save()
        return HttpResponse('Hooray')

    # If the transaction fails, return the appropriate status
    return HttpResponse('Sadness', status_code=400)
```

如果你目前使用的是 ATOMIC_REQUESTS=True，并且想切换到下一节会讲到的、更聚焦的做法，我们建议你先理解第 26 章《发现并减少瓶颈》、第 24 章《测试很臭而且浪费钱!》，以及 [原文此处交叉引用后半句残缺，未擅自补写]。

提示：涉及医疗或金融数据的项目

对于这类项目，应优先把系统设计成“最终一致性”，而不是单纯押注“事务完整性”。换句话说，你要做好事务失败、发生回滚的准备。好消息是，正因为有事务机制，即使发生回滚，数据本身依然会保持准确、干净。

7.7.2 显式声明事务

显式声明事务，是提升站点性能的一种办法。也就是说，明确指定：哪些 view 和业务逻辑需要包进事务里，哪些不需要。它的代价是开发时间会增加。

提示：Aymeric Augustin 对 ATOMIC_REQUESTS 与显式事务声明的看法

Aymeric Augustin 的原意是：“只要性能开销还可以承受，就继续使用 ATOMIC_REQUESTS。而对大多数网站来说，这个‘可以承受’基本等于‘一直都行’。”

谈到事务时，下面这些准则很值得长期记住：

- 不会修改数据库的数据库操作，不应该包在事务里。
- 会修改数据库的数据库操作，应该包在事务里。
- 某些特殊情况，例如“既要读数据库、又要改数据库，同时还有性能考量”的操作，可能会影响前面两条准则。

如果这样说还不够直观，下面这张表说明了不同 Django ORM 调用一般是否应放进事务：

目的	ORM 方法	一般应使用事务？
创建数据	.create()、.bulk_create()、.get_or_create()	是
读取数据	.get()、.filter()、.count()、.iterate()、.exists()、.exclude()、.in_bulk() 等	否
修改数据	.update()	是
删除数据	.delete()	是

表 7.1：什么时候使用事务

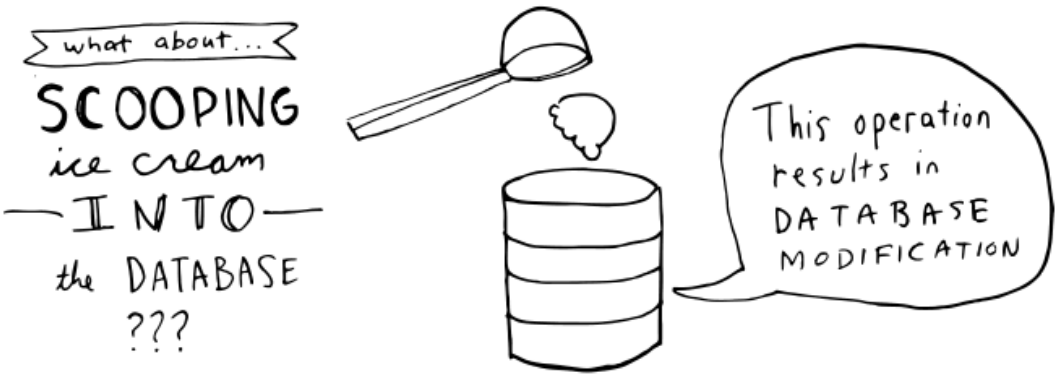

图 7.2: 图 7.2：毕竟，没有谁会像数据库那样热爱冰淇淋。

这件事我们还会在第 26 章《发现并减少瓶颈》中再次讨论，具体见 26.2.4 节《把 ATOMIC_REQUESTS 切回 False》。

提示：永远不要只给单个 ORM 方法调用包事务

Django ORM 自己在内部其实就依赖事务来保证数据一致性。比如，如果一次更新因为具体继承而影响了多张表，Django 本来就会把它包在事务里。

因此，给单个 ORM 方法调用，例如 `.create()`、`.update()`、`.delete()`，单独再包一层事务，通常没有意义。真正该用显式事务的时候，是你在一个 view、函数或方法里连续调用了多个 ORM 方法。

7.7.3 `django.http.StreamingHttpResponse` 与事务

如果某个 view 返回的是 `django.http.StreamingHttpResponse`，那一旦响应已经开始，就不可能再处理事务错误了。如果你的项目用了这种响应方式，那么 ATOMIC_REQUESTS 应该这样处理：

1. 把 ATOMIC_REQUESTS 设回 Django 的默认值，也就是 False。这样你就可以使用 7.7.2 节《显式声明事务》里的那些技术。或者……
2. 用 `django.db.transaction.non_atomic_requests` 装饰器把这个 view 包起来。

要记住：你当然可以在 streaming response 场景里继续使用 ATOMIC_REQUESTS，但事务只会作用在 view 自身。如果在生成响应流的过程中又触发了额外 SQL 查询，那么这些查询都会以自动提交模式执行。只能希望：生成响应时千万别顺手写数据库……

7.7.4 MySQL 中的事务

如果你使用的数据库是 MySQL，那么事务是否受支持，还取决于你选择的表类型，例如 InnoDB 还是 MyISAM。如果事务不受支持，那么无论 ATOMIC_REQUESTS 如何设置、也无论你的代码怎样写来配合事务，Django 都只能始终以自动提交模式运行。更多信息建议阅读：

- `docs.djangoproject.com/en/3.2/topics/db/transactions/#transactions-in-mysql`
- `dev.mysql.com/doc/refman/8.0/en/sql-transactional-statements.html`

7.7.5 Django ORM 事务资源

- `docs.djangoproject.com/en/3.2/topics/db/transactions/`：Django 关于事务的文档。
- Real Python 有一篇很棒的事务教程。虽然它是针对 Django 1.6 写的，但其中很多内容到今天仍然适用：`realpython.com/blog/python/transaction-management-with-django-1-6`

7.8 小结

这一章里，我们过了一遍项目持久化数据的各种查询方式。

我们的建议是：等你对数据在整个项目中的真实工作方式有了更清楚的感觉之后，再去添加索引。

现在我们已经知道了如何存储数据、如何给数据加索引，接下来就该开始把它展示出来了。从下一章起，我们要一头扎进 views！

第八章 基于函数的视图与基于类的视图

基于函数的视图（function-based views，FBVs）和基于类的视图（class-based views，CBVs）都是 Django 的核心组成部分。我们的建议是：这两种视图你都应该会用。

8.1 什么时候用 FBV，什么时候用 CBV

每次实现一个 view 时，都想一想：把它写成 FBV 更合适，还是写成 CBV 更合适。有些 view 天生更适合 CBV，另一些则更适合 FBV。

如果你一时拿不准该选哪一种，下一页那张流程图也许能帮上忙。

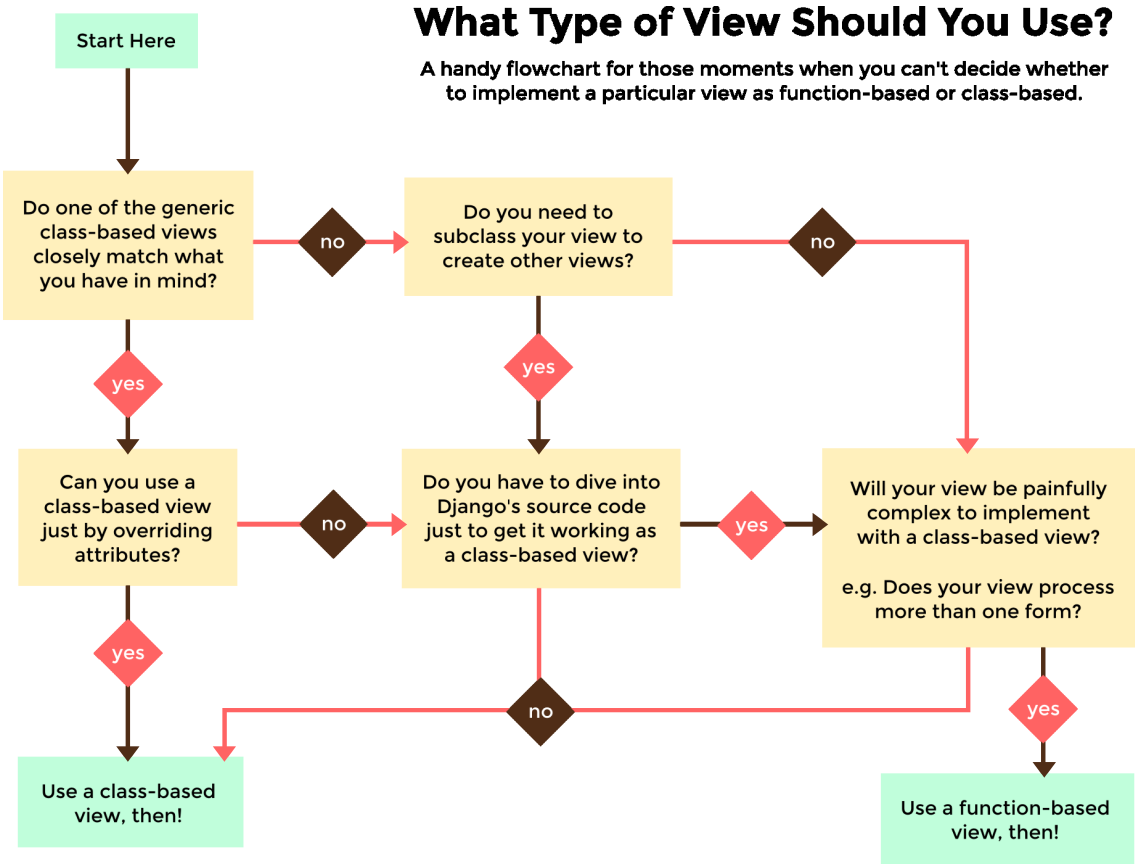

图 8.1: 图 8.1：你该用 FBV 还是 CBV？流程图。

这张流程图体现的是我们偏向 CBV 的习惯。对大多数 view，我们更愿意用 CBV；只有在实现自定义错误页，或者某个 view 用 CBV 写起来会痛苦得要命时，我们才会改用 FBV。

提示：另一种路线，尽量坚持用 FBV

有些开发者更愿意反过来：大多数 view 都用 FBV，只有那些需要被子类化的 view 才使用 CBV。这种策略也完全没问题。

8.2 不要把 View 逻辑塞进 URLConf

请求会通过 URLConf 被路由到 views，而 URLConf 通常放在名为 `urls.py` 的模块中。按照 Django 的 URL 设计哲学（`docs.djangoproject.com/en/3.2/misc/design-philosophies/#url-design`），view 和 URL 的耦合应当保持松散，这样才能带来无限灵活性，也更有利于形成最佳实践。

可即便如此，每次 Daniel 看到复杂的 `urls.py` 文件时，心里还是会忍不住大喊：

“当年我可不是辛辛苦苦写 J2EE XML 和 Zope ZCML 配置文件，好让你们这些小年轻把逻辑塞进 Django URL 模块里的!”

记住，Django 提供了一种极其简单优雅的 URL 路由定义方式。和本书里我们反复强调的其他事情一样，这种简单性是值得珍惜、也值得尊重的。经验法则如下：

1. `views` 模块应该放 view 逻辑。
2. URL 模块应该放 URL 逻辑。

你见过这种代码吗？也许是在 Django 官方 Class-Based Views 文档里见过？

示例 8.1：Django 的 CBV 风格 URLConf 模块

Code (python)

```
# Don't do this!
from django.urls import path
from django.views.generic import DetailView

from tastings.models import Tasting

urlpatterns = [
    path(
        '<int:pk>',
        DetailView.as_view(
            model=Tasting,
            template_name='tastings/detail.html'
        ),
        name='detail'
    ),
    path(
        '<int:pk>/results/',
        DetailView.as_view(
            model=Tasting,
            template_name='tastings/results.html'
        ),
        name='results'
    ),
]
```

乍一看，这段代码似乎还能接受。但我们认为，它违背了 Django 的设计哲学：

- view、url 与 model 之间本该松耦合，现在却变成了紧耦合，这意味着这些 view 定义永远没法复用。
- 同一组或相似参数在多个 CBV 之间反复出现，违背了 Don’t Repeat Yourself。
- URL 的“无限灵活性”被毁掉了。类继承本来是 Class-Based Views 的核心优势，而在这种反模式里根本无从发挥。
- 还有一堆后续问题：将来要加认证怎么办？要加授权怎么办？难道你要在 URLConf 里的每个 view 上再套两层甚至更多 decorator？一旦把 view 代码塞进 URLConf，`URLConf` 很快就会变成一个没法维护的大杂烩。

事实上，我们听过一些开发者说，他们之所以一开始远离 CBV，部分原因就是看到有人把 CBV 这样定义在 URLConf 里。

好了，抱怨到此为止。下一节我们来看看我们自己的偏好写法。

8.3 在 URLConf 中坚持松耦合

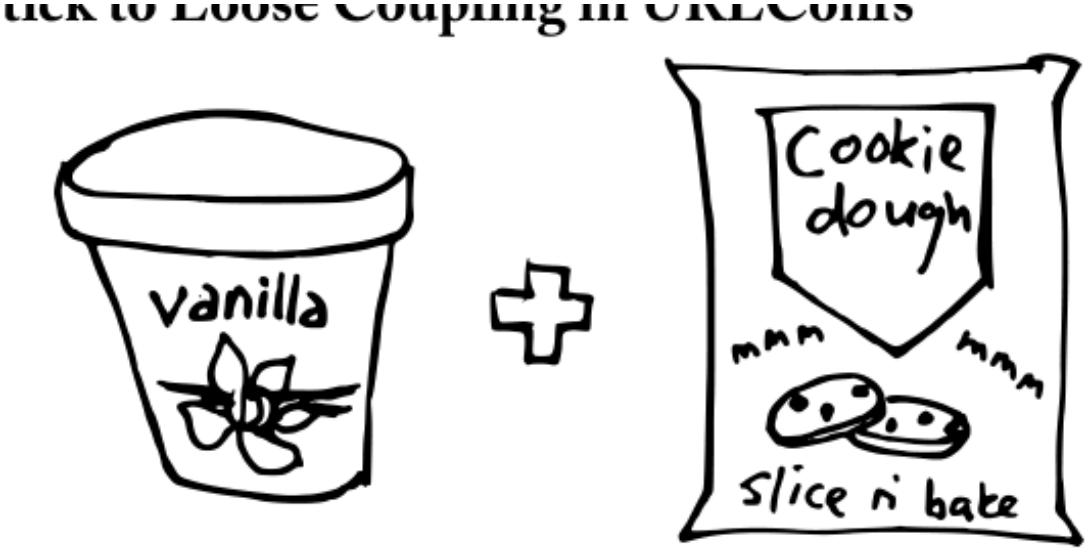

图 8.2: 图 8.2：巧克力豆曲奇面团冰淇淋式的松耦合。

下面这种写法，就能避开前面提到的那些问题。第一步，先写 views：

示例 8.2：tastings/views.py

Code (python)

```
from django.urls import reverse
from django.views.generic import ListView, DetailView, UpdateView

from .models import Tasting

class TasteListView(ListView):
    model = Tasting

class TasteDetailView(DetailView):
    model = Tasting

class TasteResultsView(TasteDetailView):
    template_name = 'tastings/results.html'

class TasteUpdateView(UpdateView):
    model = Tasting

    def get_success_url(self):
        return reverse(
            'tastings:detail',
            kwargs={'pk': self.object.pk}
        )
```

接着再定义 urls：

示例 8.3：tastings/urls.py

Code (python)

```
from django.urls import path

from . import views

urlpatterns = [
    path(
        route='',
        view=views.TasteListView.as_view(),
        name='list'
    ),
    path(
        route='<int:pk>/',
        view=views.TasteDetailView.as_view(),
        name='detail'
    ),
    path(
        route='<int:pk>/results/',
        view=views.TasteResultsView.as_view(),
        name='results'
    ),
    path(
        route='<int:pk>/update/',
        view=views.TasteUpdateView.as_view(),
        name='update'
    )
]
```

你第一眼看到我们的版本时，反应很可能会是："你确定这是好主意？你不仅拆成了两个文件，代码行数还变多了！这到底哪里更好了？"

嗯，我们就是这么做的。下面是我们觉得它非常有用的一些原因：

- Don't Repeat Yourself：views 之间不再重复参数或属性。
- 松耦合：我们把 model 和 template 名从 URLConf 中移走了，因为 view 就该待在 view 里，URLConf 就该待在 URLConf 里。理论上，我们应该能在一个或多个 URLConf 中调用同一个 view，而这种写法正好允许我们这么做。
- URLConf 应该只做一件事，并把它做好：这和上一条相连。现在的 URLConf 只专注一件事，那就是 URL 路由。我们不必在 views 和 URLConfs 两边来回找 view 逻辑，只看 views 就够了。
- views 真正享受到了 class-based 的好处：既然 view 在 `views` 模块中有正式定义，它们就可以继承别的类。这样一来，无论是加认证、加授权、支持新内容格式，还是突然塞来新的业务需求，处理起来都会轻松得多。
- 无限灵活性：既然 view 在 `views` 模块中有正式定义，它们就能实现自己的定制逻辑。

8.3.1 如果我们不用 CBV 呢?

规则还是一样。

我们见过一些项目明明使用的是 FBV，却仍然在 URLConf 里玩大量黑魔法，最终把调试拖进噩梦：比如结合 Python 模块的 `__file__` 属性、目录遍历和正则表达式，自动"变"出 URLConf。听起来就很痛苦？没错，实

际情况也确实很痛苦。

把逻辑从 URLConf 里拿出去！

8.4 使用 URL 命名空间

URL 命名空间的作用，是为 app 级命名空间和实例级命名空间提供标识符。URL 命名空间属于那种“乍看好像没多大用，一旦开始用就会疑惑自己为什么以前不用”的功能。我们可以把它总结成一句话：

不要再写 `tastings_detail` 这种 URL 名了，改写成 `tastings:detail`。

在解释它为什么有用之前，先给一个基于示例 8.2 中 app 级 URLConf 的使用例子。在项目根 URLConf 中，我们会加上：

示例 8.4：项目根 `urls.py` 片段

Code (python)

```python
urlpatterns += [
    path(
        'tastings/',
        include('tastings.urls', namespace='tastings')
    ),
]
```

再来看它在 view 中的实际用法。下面这段代码，来自示例 8.2 的一个片段：

示例 8.5：`tastings/views.py` 片段

Code (python)

```python
# tastings/views.py snippet
class TasteUpdateView(UpdateView):
    model = Tasting

    def get_success_url(self):
        return reverse(
            'tastings:detail',
            kwargs={'pk': self.object.pk}
        )
```

再看看它在 HTML 模板里的表现：

示例 8.6：`taste_list.html`

Code (django)

```django
{% extends 'base.html' %}

{% block title %}Tastings{% endblock title %}

{% block content %}
<ul>
    {% for taste in tastings %}
    <li>
        <a href="{% url 'tastings:detail' taste.pk %}">{{ taste.title }}</a>
        <small>
            (<a href="{% url 'tastings:update' taste.pk %}">update</a>)
        </small>
```

```
    </li>
    {% endfor %}
</ul>
{% endblock content %}
```

现在我们已经知道命名空间怎么实现，下面就来讲它为什么有用。

8.4.1 让 URL 名更短、更直观，也更符合 Don’t Repeat Yourself

在示例 8.3 中，你看不到像 tastings_detail 或 tastings_results 这种重复 app 或 model 名的 URL 名。取而代之的是简单直观的名字，例如 detail 和 results。这会显著提升 app 的可读性，尤其是对刚接触 Django 的开发者来说更友好。

再说了，谁愿意把 tastings 或者其他 app 名多打那么多遍呢?

8.4.2 提升与第三方库的互操作性

把 URL 名写成 <myapp>_detail 的一个问题，是 app 名一旦冲突就会很麻烦。对于我们的 tastings 这种 app，也许问题不大；但作者在 blog 和 contact 这类应用里可没少碰上。好在，URL 命名空间能很轻松地解决这个问题。假设我们已经有一个现成的 contact app，但又需要再接入一个新的，那么利用 URL 命名空间，就可以像下面这样把它们接到根 URLConf 里：

示例 8.7：根 URLConf 的互操作性

Code (python)

```
# urls.py at root of project
urlpatterns += [
    path(
        'contact/',
        include('contactmonger.urls', namespace='contactmonger')
    ),
    path(
        'report-problem/',
        include('contactapp.urls', namespace='contactapp')
    ),
]
```

然后，在模板里这样使用它们：

示例 8.8：contact.html

Code (django)

```
{% extends "base.html" %}
{% block title %}Contact{% endblock title %}
{% block content %}
<p>
    <a href="{% url 'contactmonger:create' %}">Contact Us</a>
</p>
<p>
    <a href="{% url 'contactapp:report' %}">Report a Problem</a>
</p>
{% endblock content %}
```

8.4.3 更容易搜索、升级和重构

在 Django 这种遵循 PEP 8、到处都是下划线的框架里，搜索诸如 `tastings_detail` 这样的名字往往很痛苦。搜到结果之后，你还得分辨：这到底是 view 名、URL 名，还是别的什么？

而如果你搜索的是 `tastings:detail`，结果通常就会直观得多。这一点在 app 或项目升级、重构，以及接入新的第三方库时，已经很多次帮我们省了不少力气。

8.4.4 让 app 与模板中的 reverse 技巧更多一些

我们不打算在这里展开讲这些“技巧”，因为在我们看来，这类操作几乎从来都不值得。实际上，它们通常只会增加项目复杂度，却带不来什么实质收益。不过，还是有两个使用场景值得一提：

- 像 `django-debug-toolbar` 这种需要做调试级自省的开发工具。
- 允许终端用户添加“模块”，以改变或扩展自己账户行为的项目。

开发者当然可以据此为各种创意 URL 命名空间技巧找到理由，但照旧，我们仍然建议你先尝试最简单的方案。

8.5 尽量让业务逻辑远离 Views

过去，我们曾经把大量复杂的业务逻辑直接塞进 views。可惜的是，一旦后来要生成 PDF、添加 REST API，或者支持其他输出格式时，view 里堆满业务逻辑就会让扩展变得困难得多。

这也正是我们偏爱 model method、manager method，或者通用工具辅助函数的原因。当业务逻辑被放进容易复用的组件里，再从 view 中调用时，项目的其他部分想扩展出更多能力，就会轻松很多。

当然，一开始做项目时，并不总能把这些边界划得那么清楚。所以我们后来形成了一条经验法则：只要发现自己在不同 views 之间重复的已经不是 Django 样板代码，而是业务逻辑，那就该把代码从 view 里挪出去了。

8.6 Django 的 Views 本质上都是函数

说到底，每一个 Django view 都是一个函数。这个函数接收一个 HTTP request 对象，再把它变成一个 HTTP response 对象。如果你还记得最基本的数学函数，这种“输入变输出”的过程看起来应该会很眼熟。

示例 8.9：把 Django Views 看成数学函数

Code (python)

```
# Django FBV as a function
HttpResponse = view(HttpRequest)

# Deciphered into basic math (remember functions from algebra?)
y = f(x)

# ... and then translated into a CBV example
HttpResponse = View.as_view()(HttpRequest)
```

这种“把输入变成输出”的概念，其实是你用 Django views 做各种事情的基础，不管这些 views 是基于函数的，还是基于类的。

提示：Class-Based Views 实际上也是以函数形式被调用的

Django 的 CBV 看起来似乎和 FBV 很不一样。但实际上，URLConf 中调用的 `View.as_view()` 这个 class-method，返回的正是一个可调用的 view 实例。换句话说，它本质上就是一个回调函数，以和 function-based view 完全相同的方式处理 request / response 周期！

8.6.1 最简单的 Views

有了这个认识，就很值得记住：在 Django 里，最简单的 view 可以写成什么样：

示例 8.10：simplest_views.py

Code (python)

```python
from django.http import HttpResponse
from django.views.generic import View

# The simplest FBV
def simplest_view(request):
    # Business logic goes here
    return HttpResponse('FBV')

# The simplest CBV
class SimplestView(View):

    def get(self, request, *args, **kwargs):
        # Business logic goes here
        return HttpResponse('CBV')
```

为什么知道这一点有用?

- 有时候我们确实只需要那种做一件极小事情的一次性 view。
- 理解“最简单的 Django view”长什么样，能让我们更明白 view 本质上到底在做什么。
- 它也清楚展示了：Django 的 FBV 对 HTTP 方法是中性的，而 Django 的 CBV 则需要显式声明具体的 HTTP 方法。

8.7 不要在 View Context 里用 locals()

任何可调用对象直接返回 locals()，都是一种反模式。它看上去像个省事捷径，实际上却是一场很耗时间的维护噩梦。我们来看个例子：

示例 8.11：不恰当地使用 locals()

Code (python)

```python
# Don't do this!
def ice_cream_store_display(request, store_id):

    store = get_object_or_404(Store, id=store_id)
    date = timezone.now()
    return render(
        request,
        'melted_ice_cream_report.html',
        locals()
    )
```

表面上看，这似乎没什么问题。

但实际上，我们已经从“显式设计”滑进了“隐式反模式”，让这个原本很简单的 view 变得烦人难维护。最具体的问题是：你根本无法一眼知道这个 view 打算返回哪些上下文变量。于是，只要其中某个变量被改名，这种变化就不会立刻显现出来：

示例 8.12：对前一个坏例子的改动版本

Code (python)

```
# Don't do this!
def ice_cream_store_display(request, store_id):

    store = get_object_or_404(Store, id=store_id)
    now = timezone.now()
    return render(
        request,
        'melted_ice_cream_report.html',
        locals()
    )
```

你花了多久才看出坏例子 8.11 和坏例子 8.12 的区别？这还只是个简单例子。想象一下，如果是更复杂的 view，再配上一个很大的模板，那会有多折磨人。这也就是为什么我们强烈主张在 view 中显式定义 context：

示例 8.13：显式的 View Context

Code (python)

```
def ice_cream_store_display(request, store_id):
    return render(
        request,
        'melted_ice_cream_report.html',
        {
            'store': get_object_or_404(Store, id=store_id),
            'now': timezone.now()
        }
    )
```

- Alex Martelli 的理由：stackoverflow.com/a/1901720

8.8 小结

这一章先讨论了什么时候该用 FBV，什么时候该用 CBV，而我们的个人偏好更偏向后者。实际上，从下一章开始，我们会先深入挖掘 FBV 可以发挥的能力，接着再用一整章来讲 CBV。

我们也讨论了不要把 view 逻辑塞进 URLConf。我们的立场是：view 代码就该放在 app 的 `views.py` 模块里，URLConf 代码就该放在 app 的 `urls.py` 模块里。坚持这种做法，能让 class-based views 真正享受到对象继承带来的好处，也能让代码复用更轻松，设计弹性更大。

第九章 基于函数的视图最佳实践

从 Django 项目诞生之初开始，基于函数的视图就一直被世界各地的开发者频繁使用。虽然基于类的视图这些年越来越常见，但“直接写一个函数”这种简单感，对新手和老手都有吸引力。尽管作者本人更偏爱 CBV，但我们也确实参与过不少使用 FBV 的项目，下面就是一些我们逐渐喜欢上的模式。

9.1 FBV 的优势

FBV 的简单，是以代码复用能力为代价换来的：FBV 不像 CBV 那样，天然具备从父类继承的能力。但它也有自己的长处，那就是更偏函数式，而这会自然导向一些很有意思的策略。

我们在写 FBV 时会遵循这些准则：

- view 代码越少越好。
- 永远不要在 views 里重复代码。
- views 应该处理展示逻辑。业务逻辑尽量放进 models；实在不行，再考虑放进 forms。
- 保持 view 简单。
- 用它们来编写自定义 403、404 和 500 错误处理器。
- 避免写复杂的多层嵌套 `if` 代码块。

9.2 传递 `HttpRequest` 对象

有时候，我们希望在多个 views 中复用代码，但又不想把它绑定到 middleware 或 context processor 这类全局机制上。从本书引言部分开始，我们就一直建议大家编写可以在整个项目中复用的 utility functions。

对很多工具函数来说，我们实际上要做的事情，就是从 `django.http.HttpRequest`（以下简称 `HttpRequest`）对象上取出一个或多个属性，然后据此收集数据或执行操作。

我们的经验是：如果把 `request` 对象本身作为首要参数，那么很多方法的参数表就会简单得多。这意味着你不用在函数 / 方法参数管理上耗费那么多认知负担。直接把 `HttpRequest` 传进去就好！

示例 9.1：`sprinkles/utils.py`

Code (python)

```python
from django.core.exceptions import PermissionDenied
from django.http import HttpRequest

def check_sprinkle_rights(request: HttpRequest) -> HttpRequest:
    if request.user.can_sprinkle or request.user.is_staff:
        return request

    # Return a HTTP 403 back to the user
    raise PermissionDenied
```

`check_sprinkle_rights()` 会快速检查用户是否有撒糖针的权限；如果没有，就抛出一个 `django.core.exceptions.Per` 异常。正如 31.4.3 节《`django.core.exceptions.PermissionDenied`》中会提到的，这会触发一个自定义的 HTTP 403 view。

你会注意到，我们返回的是 `HttpRequest` 对象，而不是某个随意的值，甚至也不是 `None`。这么做的一个原因是：Python 是动态类型语言，我们可以直接往 `HttpRequest` 对象上附加额外属性。例如：

示例 9.2：增强版 `sprinkles/utils.py`

Code (python)

```
from django.core.exceptions import PermissionDenied
from django.http import HttpRequest, HttpResponse

def check_sprinkles(request: HttpRequest) -> HttpRequest:
    if request.user.can_sprinkle or request.user.is_staff:
        # By adding this value here it means our display templates
        # can be more generic. We don't need to have
        # {% if request.user.can_sprinkle or request.user.is_staff %}
        # instead just using
        # {% if request.can_sprinkle %}
        request.can_sprinkle = True
        return request

    # Return a HTTP 403 back to the user
    raise PermissionDenied
```

我们这么做还有另一个原因，后面很快就会讲到。先来看这段代码在实际 view 中如何使用：

示例 9.3：在 FBV 中传递 Request 对象

Code (python)

```
# sprinkles/views.py
from django.shortcuts import get_object_or_404
from django.shortcuts import render
from django.http import HttpRequest, HttpResponse

from .models import Sprinkle
from .utils import check_sprinkles

def sprinkle_list(request: HttpRequest) -> HttpResponse:
    """Standard list view"""

    request = check_sprinkles(request)

    return render(
        request,
        "sprinkles/sprinkle_list.html",
        {"sprinkles": Sprinkle.objects.all()}
    )

def sprinkle_detail(request: HttpRequest, pk: int) -> HttpResponse:
    """Standard detail view"""

    request = check_sprinkles(request)

    sprinkle = get_object_or_404(Sprinkle, pk=pk)

    return render(
        request,
        "sprinkles/sprinkle_detail.html",
```

```
        {"sprinkle": sprinkle}
    )

def sprinkle_preview(request: HttpRequest) -> HttpResponse:
    """Preview of new sprinkle, but without the
    check_sprinkles function being used.
    """
    sprinkle = Sprinkle.objects.all()
    return render(
        request,
        "sprinkles/sprinkle_preview.html",
        {"sprinkle": sprinkle}
    )
```

这种做法还有个好处：接到 class-based views 里也非常容易：

示例 9.4：在 CBV 中传递 Request 对象

Code (python)

```
from django.views.generic import DetailView

from .models import Sprinkle
from .utils import check_sprinkles

class SprinkleDetail(DetailView):
    """Standard detail view"""

    model = Sprinkle

    def dispatch(self, request, *args, **kwargs):
        request = check_sprinkles(request)
        return super().dispatch(request, *args, **kwargs)
```

提示：特定函数参数也有它们的位置

单参数函数的缺点在于：像 `pk`、`flavor`、`text` 这种明确参数名，往往能让人一眼就看出函数用途。换句话说，这种“直接传 request”技巧最好只用于那些尽可能通用的操作。

既然我们都在一层层地“函数里再调函数”，那如果能让这种复用一眼就被认出来，不是很好吗？这正是 decorator 登场的时候。

9.3 Decorator 很甜

这一次，“甜”不是在说冰淇淋，而是在说代码！在计算机科学里，syntactic sugar 指的是那种为了让代码更易读、更易表达，而额外加进语言里的语法。Python 中的 decorator 也是如此：它不是出于“没有就不行”的必要性被加进来的，而是为了让代码对人类读者来说更干净、更甜。所以没错，Decorator 很甜。

当我们把“简单函数的力量”与“decorator 这种语法糖”结合在一起时，就能得到非常顺手、可复用的工具。最经典的例子，就是几乎无处不在的 `django.contrib.auth.decorators.login_required` 装饰器。

下面是一个可以用于 function-based views 的 decorator 模板：

示例 9.5：简单的 Decorator 模板

Code (python)

```
import functools

def decorator(view_func):
    @functools.wraps(view_func)
    def new_view_func(request, *args, **kwargs):
        # You can modify the request (HttpRequest) object here.
        response = view_func(request, *args, **kwargs)
        # You can modify the response (HttpResponse) object here.
        return response
    return new_view_func
```

如果你现在还没完全看懂，也正常。下面我们一步步来，用内联代码注释解释每一部分在做什么。先把前一个例子里的 decorator 模板改造成我们需要的样子：

示例 9.6：Decorator 示例

Code (python)

```
# sprinkles/decorators.py
import functools

from . import utils

# based off the decorator template from the previous example
def check_sprinkles(view_func):
    """Check if a user can add sprinkles"""

    @functools.wraps(view_func)
    def new_view_func(request, *args, **kwargs):

        # Act on the request object with utils.can_sprinkle()
        request = utils.can_sprinkle(request)

        # Call the view function
        response = view_func(request, *args, **kwargs)

        # Return the HttpResponse object
        return response
    return new_view_func
```

然后这样把它挂到函数上：

示例 9.7：Decorator 的使用示例

Code (python)

```
# sprinkles/views.py
from django.shortcuts import get_object_or_404, render

from .decorators import check_sprinkles
from .models import Sprinkle

# Attach the decorator to the view
```

```
@check_sprinkles
def sprinkle_detail(request: HttpRequest, pk: int) -> HttpResponse:
    """Standard detail view"""

    sprinkle = get_object_or_404(Sprinkle, pk=pk)

    return render(
        request,
        "sprinkles/sprinkle_detail.html",
        {"sprinkle": sprinkle}
    )
```

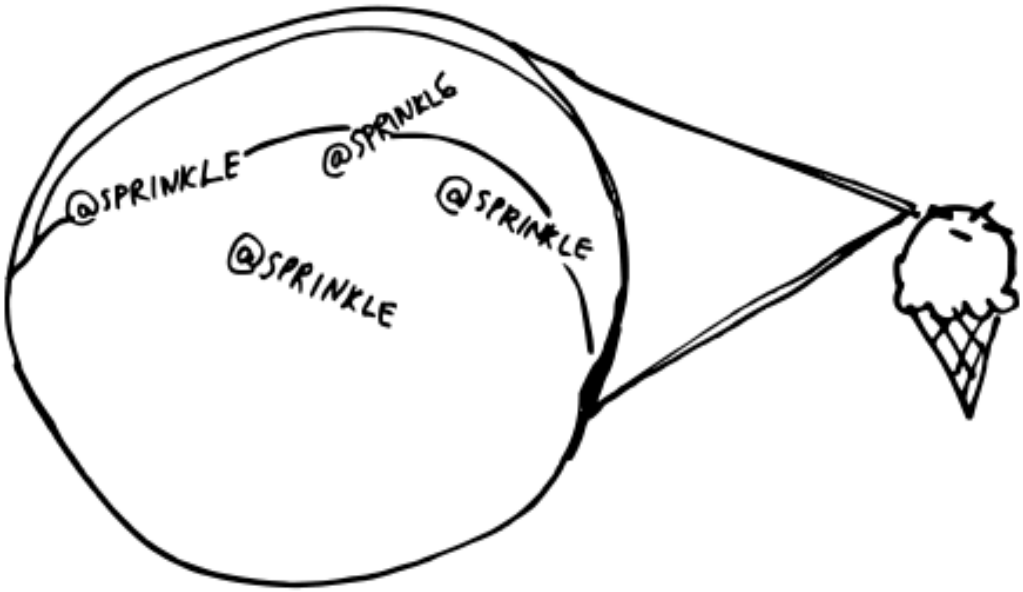

图 9.1: 图 9.1：仔细看 sprinkles，你会发现它们其实就是 Python decorators。

提示：那 `functools.wraps()` 呢?

细心的读者也许已经注意到，我们前面的 decorator 示例用了 Python 标准库里的 `functools.wraps()`。这是一个很方便的小工具，它会把包括 docstring 在内的重要元数据复制到新的装饰后函数上。它不是绝对必需的，但它会让项目维护轻松很多。

9.3.1 使用 Decorator 时要克制

和所有强力工具一样，decorator 也可能被用歪。decorator 一旦叠得太多，就会形成自己的那种“代码迷雾”，甚至让复杂的 class-based view 继承树都显得清爽起来。使用 decorators 时，最好预先给 view 设定一个“最多允许叠几层 decorator”的上限，并坚持执行。相关视频：`pyvideo.org/pycon-us-2011/pycon-2011--how-to-write-obfuscated-pytho`

9.3.2 Decorator 的额外资源

- Decorators Explained：`jeffknupp.com/blog/2013/11/29/improve-your-python-decorators-explained/`
- 作者 Daniel Feldroy 的 Decorator Cheat Sheet：`daniel.feldroy.com/python-decorator-cheatsheet.html`

9.4 传递 HttpResponse 对象

就像 HttpRequest 对象一样，我们也可以把 HttpResponse 对象在函数之间传来传去。你可以把它理解成一种“可选择使用的 Middleware.process_template_response() 方法”。参见：docs.djangoproject.com/en/3.2/topics/htt

没错，这种技巧同样也能和 decorators 结合使用。参见示例 8.5，那里已经暗示了可以怎么做。

9.5 基于函数的视图更多资源

Luke Plant 是 Django 核心开发者，而且对 Function-Based Views 有非常鲜明的偏好。虽然我们并不同意他那篇文章里大多数“反 CBV”的观点，但对任何编写 FBV 的人来说，下面这篇文章仍然有巨大价值：

spookylukey.github.io/django-views-the-right-way/

9.6 小结

Function-based views 在 Django 世界里依然活得很好。只要记住：每个函数都接收一个 HttpRequest 对象，并返回一个 HttpResponse 对象，我们就能把这一点转化为优势。我们可以围绕“修改 HttpRequest / HttpResponse 的通用函数”来构建复用能力，而这些函数又能进一步被拿来构建 decorators。

这一章的结尾，我们也想承认一件事：凡是我们在 function-based views 上学到的这些经验，几乎都可以迁移到下一章即将开始讨论的 class-based views 上去。

第十章 基于类的视图最佳实践

Django 为编写基于类的视图（class-based views，CBVs）提供了一套标准方式。正如我们在前几章反复提到的，Django view 说到底就是一个可调用对象：它接收 request 对象，返回 response。对于基于函数的视图（FBVs）来说，这个可调用对象就是 view 函数本身；对于 CBV 来说，则是 view 类提供的 `as_view()` 类方法返回的那个可调用对象。这套机制定义在 `django.views.generic.View` 中。所有 CBV 都应该直接或间接继承自这个类。

Django 还提供了一系列通用基于类的视图（generic class-based views，GCBVs），它们实现了绝大多数 Web 项目里反复出现的常见模式，也很好地展示了 CBV 的威力。

包提示：补上 Django GCBV 缺失的那一部分

开箱即用的 Django，并没有为 GCBV 提供一些其实很有用的 mixins。`django-braces` 这个库解决了其中大部分问题。它提供了一组实现清晰的 mixins，让 Django GCBV 的开发速度和使用体验都提升不少。这个库实用到什么程度？它里面有三个 mixin 后来直接被抄进了 Django 核心。

10.1 使用 CBV 时的准则

- view 代码越少越好。
- 永远不要在 views 里重复代码。
- views 应该处理展示逻辑。业务逻辑尽量放进 models；实在不行，再考虑放进 forms。
- 保持你的 views 简单。
- 也让你的 mixins 保持更简单。

提示：尽快熟悉 `ccbv.co.uk`

严格来说，这件事甚至应该被列成第六条准则。`ccbv.co.uk` 实在太有用了，所以我们觉得它值得单独拿一个提示框。这个网站会把每个 CBV 定义或继承到的全部属性和方法，摊平成“一页一个 view”的完整视图。多数 Django 开发者一旦过了 CBV 教程阶段，对 `ccbv.co.uk` 的依赖往往会比官方文档还高。

10.2 在 CBV 中使用 Mixins

可以把编程里的 mixin 想成冰淇淋里的 mixin：无论什么口味的冰淇淋，只要拌进去脆糖块、水果切片，甚至培根，都能立刻增强风味。

图 10.1: 图 10.1：冰淇淋里常见和不那么常见的 mixins。

软冰淇淋尤其适合加 mixins：普通香草 soft serve，只要混入 sprinkles、蓝色奶油糖霜和黄色蛋糕块，马上就变成了生日蛋糕口味冰淇淋。

在编程里，mixin 指的是这样一种类：它提供的是“被继承的功能”，但本身并不打算被单独实例化。在支持多重继承的语言里，mixin 可以用来给类增添额外功能和行为。

在 Django app 里，我们完全可以借助 mixin 的力量，拼装出自己的 view 类。

当我们用 mixin 来组合自定义 view 类时，推荐遵循 Kenneth Love 提出的这几条继承规则。这些规则顺着 Python 的方法解析顺序（method resolution order，MRO）来走。最粗略地说，它会从左往右查找：

1. Django 提供的基础 view 类永远放在最右边。
2. mixin 放在基础 view 左边。
3. mixin 不要再继承任何其他类。让你的继承链保持简单！

下面是这些规则的实际例子：

示例 10.1：在 View 中使用 Mixins

Code (python)

```
from django.views.generic import TemplateView

class FreshFruitMixin:

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["has_fresh_fruit"] = True
        return context

class FruityFlavorView(FreshFruitMixin, TemplateView):
    template_name = "fruity_flavor.html"
```

在这个有点傻气的例子里，`FruityFlavorView` 同时继承了 `FreshFruitMixin` 和 `TemplateView`。

由于 `TemplateView` 是 Django 提供的基础 view 类，所以它必须放在最右边（规则 1）；而左边就是 `FreshFruitMixin`（规则 2）。这样一来，我们就能确认方法和属性的执行顺序是正确的。

10.3 针对不同任务，该用哪个 Django GCBV?

generic class-based views 的强大，是拿简洁性换来的：GCBV 的继承链相当复杂，有些 view 在 import 后甚至会带着多达八层父类。要想搞清楚到底该用哪个 view、又该改哪个方法，有时真的很费劲。

为了缓解这种痛苦，下面这张表按名称和用途列出了 Django 常见 CBV。这里默认所有 view 都带有 `django.views.generic` 前缀。

名称	用途	Two Scoops 示例
`View`	基础 view，或者一个几乎什么都能做的趁手 view	参见 10.6 节《使用 `django.views.generic.View`》
`RedirectView`	把用户重定向到另一个 URL	把访问 `/login/` 的用户送到 `/login/`
`TemplateView`	渲染一个 Django HTML 模板	网站的 `/about/` 页面
`ListView`	列出对象	冰淇淋口味列表
`DetailView`	展示一个对象	某种冰淇淋口味的详情页
`FormView`	提交一个表单	网站联系表单或邮件表单
`CreateView`	创建一个对象	新建一种冰淇淋口味
`UpdateView`	更新一个对象	更新已有冰淇淋口味
`DeleteView`	删除一个对象	删除像 Vanilla Steak 这样讨嫌的冰淇淋口味
Generic date views	展示发生在某个时间范围内的对象	博客是典型用途。对 Two Scoops 来说，我们可以做一份公开历史，记录各种口味是什么时候被加入数据库的

表 10.1：Django CBV 用途表

提示：Django CBV / GCBV 使用的三大学派

我们发现，围绕 CBV 和 GCBV 的使用，基本存在三种主要思路：

“能用的 generic views 全都用起来”学派

这派的想法是：既然 Django 已经提供了可以减轻工作量的功能，为什么不用？我们自己大致属于这一派，而且靠这种方式很成功地快速搭出并维护了不少项目。

“只用 `django.views.generic.View`”学派

这派的想法是：基础 CBV 已经刚刚好，才是真正的 CBV，其他都只是 Generic CBV。过去一年里，我们发现这种思路在一些棘手任务上非常有用，尤其是那些“把所有 generic views 都用起来”的资源型思路开始失灵的时候。本章稍后会给出一些使用场景。

“除非你真的要子类化 view，否则就尽量避开它们”学派

Jacob Kaplan-Moss 的建议大意是：“我的总体建议是，先从 function views 开始，因为它们更容易阅读和理解；只有在确实需要时再用 CBV。那什么时候算需要？当你有相当一大块代码要在多个 views 之间复用时。”

我们通常属于第一派，但知道这一点很重要：这里并没有什么真正统一的最佳实践共识。

10.4 Django CBV 的通用技巧

这一节覆盖的是那些适用于全部或大多数 Django CBV / GCBV 实现的技巧。我们的经验是，它们能明显加快 view、template 以及相关测试的编写。这些技巧既适用于普通 Class-Based Views，也适用于 Generic Class-Based Views。照旧，它们都建立在 Django 中的面向对象编程手法之上。

10.4.1 把 Django CBV / GCBV 限制为“仅认证用户可访问”

Django 的 CBV 文档提供了一个如何把 `django.contrib.auth.decorators.login_required` 装饰器和 CBV 结合使用的示例。这个示例能跑，但它违背了“不要把逻辑塞进 `urls.py`”这条规则：

`docs.djangoproject.com/en/3.2/topics/class-based-views/intro/#decorating-class-based-views`

好消息是，Django 已经直接提供了一个现成的 `LoginRequiredMixin`，几分钟就能接上。例如，我们完全可以把它加到前面用过的所有 Django GCBV 上：

示例 10.2：使用 `LoginRequiredMixin`

Code (python)

```python
# flavors/views.py
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import DetailView

from .models import Flavor

class FlavorDetailView(LoginRequiredMixin, DetailView):
    model = Flavor
```

提示：别忘了 GCBV 的 Mixin 顺序！

记住：

- `LoginRequiredMixin` 必须永远放在最左边。
- 基础 view 类必须永远放在最右边。

如果你忘了这件事、把顺序写反，就会得到错误甚至不可预测的结果。

警告：在使用 `LoginRequiredMixin` 时覆写 `dispatch()`

如果你用了 LoginRequiredMixin，又覆写了 dispatch 方法，那一定要保证做的第一件事就是调用 super().dispatch(request, *args, **kwargs)。任何写在 super() 之前的代码，都会在“用户尚未通过认证”的情况下照常执行。

10.4.2 表单有效时，在 View 中执行自定义动作

当你需要在“表单有效”的 view 中执行自定义逻辑时，GCBV 的工作流会把请求送到 form_valid() 方法。

示例 10.3：在有效表单中加入自定义逻辑

Code (python)

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import CreateView

from .models import Flavor

class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor
    fields = ['title', 'slug', 'scoops_remaining']

    def form_valid(self, form):
        # Do custom logic here
        return super().form_valid(form)
```

如果你想对“已经验证通过”的表单数据执行自定义逻辑，只要把这段逻辑加到 form_valid() 中即可。form_valid() 的返回值应当是一个 django.http.HttpResponseRedirect。

10.4.3 表单无效时，在 View 中执行自定义动作

当你需要在“表单无效”的 view 中执行自定义逻辑时，Django GCBV 的工作流会把请求送到 form_invalid() 方法。这个方法应当返回一个 django.http.HttpResponse。

示例 10.4：覆写 form_invalid() 的行为

Code (python)

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import CreateView

from .models import Flavor

class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor

    def form_invalid(self, form):
        # Do custom logic here
        return super().form_invalid(form)
```

就像你能给 form_valid() 添加逻辑一样，也完全可以给 form_invalid() 添加逻辑。

你会在 13.5.1 节《ModelForm 数据先保存到表单，再保存到模型实例》中看到一个同时覆写这两个方法的例子。

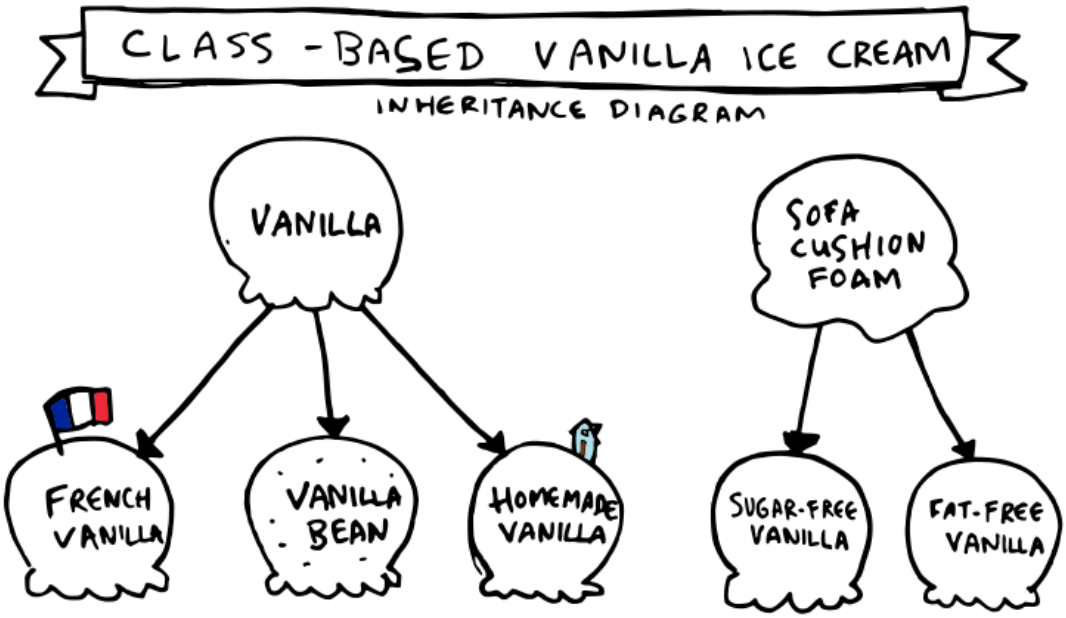

图 10.2: 图 10.2：另一种 CBV：class-based vanilla 冰淇淋。

10.4.4 使用 View 对象本身

如果你使用 class-based views 来渲染内容，可以考虑直接利用 view 对象自身，为其他方法和属性提供可调用的属性或方法。这些东西不仅能被 view 内部的其他方法调用，也能在模板中直接使用。例如：

示例 10.5：使用 View 对象

Code (python)

```python
from django.contrib.auth.mixins import LoginRequiredMixin
from django.utils.functional import cached_property
from django.views.generic import UpdateView, TemplateView

from .models import Flavor
from .tasks import update_user_who_favorited

class FavoriteMixin:

    @cached_property
    def likes_and_favorites(self):
        """Returns a dictionary of likes and favorites"""

        likes = self.object.likes()
        favorites = self.object.favorites()
        return {
            "likes": likes,
            "favorites": favorites,
            "favorites_count": favorites.count(),
        }

class FlavorUpdateView(LoginRequiredMixin, FavoriteMixin, UpdateView):

    model = Flavor
```

```
    fields = ['title', 'slug', 'scoops_remaining']

    def form_valid(self, form):
        update_user_who_favorited(
            instance=self.object,
            favorites=self.likes_and_favorites['favorites']
        )
        return super().form_valid(form)

class FlavorDetailView(LoginRequiredMixin, FavoriteMixin, TemplateView):

    model = Flavor
```

这种写法的好处在于：`flavors/` app 里的不同模板，现在都能直接访问这个属性：

示例 10.6：在 `flavors/base.html` 中使用 View 方法

Code (django)

```
{# flavors/base.html #}
{% extends "base.html" %}

{% block likes_and_favorites %}
<ul>
    <li>Likes: {{ view.likes_and_favorites.likes }}</li>
    <li>Favorites: {{ view.likes_and_favorites.favorites_count }}</li>
</ul>
{% endblock likes_and_favorites %}
```

10.5 GCBV 与 Forms 是怎么配合的

GCBV 一个很常见的困惑来源，就是它们和 Django forms 配合时到底该怎么用。

继续拿我们最爱的“冰淇淋口味跟踪 app”做例子，下面我们用几个场景来梳理：和表单相关的 views 一般是怎么拼在一起的。

先定义一个本节后续示例要用到的 flavor 模型：

示例 10.7：Flavor 模型

Code (python)

```
# flavors/models.py
from django.db import models
from django.urls import reverse

class Flavor(models.Model):
    class Scoops(models.IntegerChoices):
        SCOOPS_0 = 0
        SCOOPS_1 = 1

    title = models.CharField(max_length=255)
    slug = models.SlugField(unique=True)
    scoops_remaining = models.IntegerField(
        choices=Scoops.choices,
```

```
        default=Scoops.SCOOPS_0
    )

    def get_absolute_url(self):
        return reverse(
            "flavors:detail",
            kwargs={"slug": self.slug}
        )
```

接下来，我们来看几个大多数 Django 开发者迟早都会遇到的常见表单场景。

10.5.1 Views + ModelForm 示例

这是最简单、也最常见的 Django 表单场景。通常你一旦创建了一个 model，就希望能够对应地新增记录，也能编辑既有记录。

在这个例子里，我们会展示如何构建一组 views，用来创建、更新和展示 `Flavor` 记录，同时也演示如何在变更后给用户确认反馈。

这里会涉及下面这些 views：

1. `FlavorCreateView`：对应新增 flavor 的表单。
2. `FlavorUpdateView`：对应编辑现有 flavor 的表单。
3. `FlavorDetailView`：对应 flavor 创建和更新后的确认页。

把这些 views 画成流程，就是：

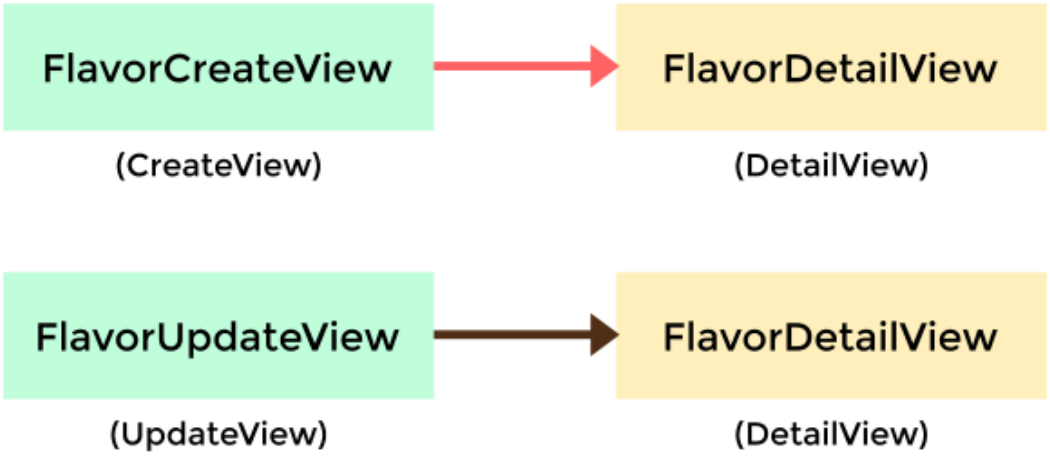

图 10.3: 图 10.3：Views + ModelForm 流程

注意，我们会尽量贴近 Django 的命名约定。`FlavorCreateView` 继承 Django 的 `CreateView`，`FlavorUpdateView` 继承 `UpdateView`，`FlavorDetailView` 继承 `DetailView`。

写这些 views 非常容易，因为大部分只是把 Django 已有的能力直接拿来用：

示例 10.8：快速用 CBV 搭建 Views

Code (python)

```
# flavors/views.py
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import CreateView, DetailView, UpdateView

from .models import Flavor
```

```
class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor
    fields = ['title', 'slug', 'scoops_remaining']

class FlavorUpdateView(LoginRequiredMixin, UpdateView):
    model = Flavor
    fields = ['title', 'slug', 'scoops_remaining']

class FlavorDetailView(DetailView):
    model = Flavor
```

第一眼看上去很简单，对吧？只用一点点代码，就做成了这么多事。

但等等，这里有个坑。如果我们把这些 views 接进 `urls.py`，再配好对应模板，就会发现一个问题：`FlavorDetailView` 还不能算真正的“确认页”。

至少目前确实如此。好在，这个问题修起来很快，只需要对已有 view 和模板做一些小改动。

修复的第一步，是使用 `django.contrib.messages`，让访问 `FlavorDetailView` 的用户知道：他们刚刚创建或更新了一个 flavor。

为此，我们需要覆写 `FlavorCreateView.form_valid()` 和 `FlavorUpdateView.form_valid()`。更方便的做法，是写一个 `FlavorActionMixin`，让两个 view 共同继承。

为了把确认页修好，我们把 `flavors/views.py` 改成下面这样：

示例 10.9：成功消息示例

Code (python)

```
# flavors/views.py
from django.contrib import messages
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import CreateView, DetailView, UpdateView

from .models import Flavor

class FlavorActionMixin:

    fields = ['title', 'slug', 'scoops_remaining']

    @property
    def success_msg(self):
        return NotImplemented

    def form_valid(self, form):
        messages.info(self.request, self.success_msg)
        return super().form_valid(form)

class FlavorCreateView(LoginRequiredMixin, FlavorActionMixin, CreateView):

    model = Flavor
    success_msg = "Flavor created!"

class FlavorUpdateView(LoginRequiredMixin, FlavorActionMixin, UpdateView):
```

```
    model = Flavor
    success_msg = "Flavor updated!"

class FlavorDetailView(DetailView):
    model = Flavor
```

本章前面我们已经看过一个更简单的覆写 `form_valid()` 的例子。这里我们把类似的 `form_valid()` 覆写逻辑抽进一个 mixin，让多个 views 都能复用。

现在，我们利用 Django 的 messages framework，在每次成功新增或编辑后都给用户显示确认消息。`FlavorActionMixin` 的职责，就是根据当前 view 触发的动作，把相应确认消息排进队列。

提示：这其实复刻了 `SuccessMessageMixin`

自从我们在 2013 年写下这一节以来，Django 已经内建了 `django.contrib.messages.views.SuccessMessageMixin`，提供了类似功能。

参考：`https://docs.djangoproject.com/en/3.2/ref/contrib/messages/#django.contrib.messages.views.Succe`

提示：Mixin 不应该再继承别的对象

请特别注意，`FlavorActionMixin` 在我们的代码里是一个基础类，它没有再继承其他类，也没有继承已有的 mixin 或 view。mixin 的继承链越浅越好。简单，就是美德！

当 flavor 被创建或更新后，一组 message 会被送进 `FlavorDetailView` 的 context。只要把下面这段代码加进对应模板，然后再去创建或更新一个 flavor，你就能看到这些消息：

示例 10.10：`flavor_detail.html`

Code (django)

```
{% if messages %}
<ul class="messages">
{% for message in messages %}
<li id="message_{{ forloop.counter }}"
{% if message.tags %} class="{{ message.tags }}"
{% endif %}>
{{ message }}
</li>
{% endfor %}
</ul>

{% endif %}
```

提示：把消息模板代码复用起来！

一种常见做法，是把上面这段代码直接放进项目的基础 HTML 模板里。这样一来，你的整个项目中的模板就都自动支持 message 机制了。

回顾一下，这个例子再次演示了：如何覆写 `form_valid()`、如何把这段逻辑抽进 mixin、如何在一个 view 中组合多个 mixin，以及如何快速上手非常有用的 `django.contrib.messages` 框架。

10.5.2 Views + Form 示例

有时你想用的是 Django `Form`，而不是 `ModelForm`。搜索表单就是一个特别典型的场景，当然你还会遇到其他类似情况。

在这个例子里，我们会实现一个简单的 flavor 搜索表单。这个表单渲染成 HTML 后，并不会修改任何 flavor 数据。它的 action 会去查询 ORM，然后把找到的记录列在搜索结果页上。

我们的预期是：当用户在 flavor 搜索页里搜索“Dough”时，应该被带到一个结果页，列出像“Chocolate Chip Cookie Dough”、“Fudge Brownie Dough”、“Peanut Butter Cookie Dough”之类标题里包含“Dough”的冰淇淋口味。嗯，我们真的很想在自己的网站里要这个功能。

实现这个需求当然还有更复杂的方式，但对这个简单场景来说，我们只需要一个单独的 view。`FlavorListView` 同时负责搜索页和搜索结果页。

实现结构大致如下：

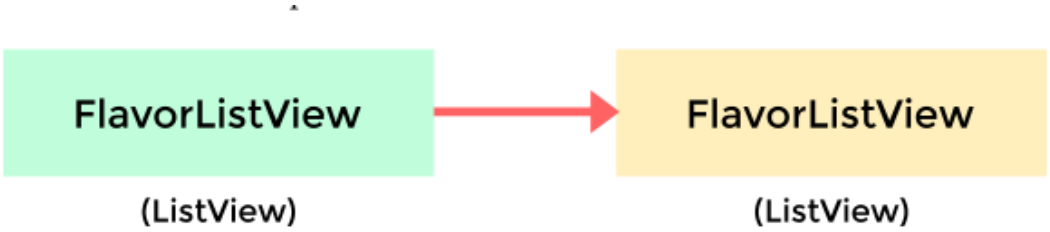

图 10.4: 图 10.4：Views + Form 流程

在这个场景里，我们希望遵循互联网搜索页的通用惯例，用 `q` 作为查询参数名。我们也希望接受的是 GET 请求，而不是 POST。对表单来说这有点少见，但在这里完全合理。别忘了：这个表单不会新增、编辑或删除对象，所以这里根本不需要 POST。

为了根据搜索词返回匹配结果，我们需要修改 `ListView` 默认给出的 queryset。具体做法，就是覆写 `ListView` 的 `get_queryset()` 方法。我们在 `flavors/views.py` 中加入下面这段代码：

示例 10.11：结合 Q 搜索的 List View

Code (python)

```python
from django.views.generic import ListView

from .models import Flavor

class FlavorListView(ListView):
    model = Flavor

    def get_queryset(self):
        # Fetch the queryset from the parent get_queryset
        queryset = super().get_queryset()

        # Get the q GET parameter
        q = self.request.GET.get("q")
        if q:
            # Return a filtered queryset
            return queryset.filter(title__icontains=q)

        # Return the base queryset
        return queryset
```

这样一来，我们列出的就不再是全部 flavors，而只会是标题里包含搜索字符串的那些 flavor。

正如前面提到的，搜索表单有一个很特别的地方：和绝大多数 HTML 表单不同，它会在 HTML 中显式指定 GET 请求。这是因为搜索表单不是在改数据，而是在向服务器取信息。搜索表单大致应该长成这样：

示例 10.12：搜索表单 HTML 片段

Code (django)

```
{# templates/flavors/_flavor_search.html #}
{% comment %}
```

```
Usage: {% include "flavors/_flavor_search.html" %}
{% endcomment %}
<form action="{% url "flavor_list" %}" method="GET">
    <input type="text" name="q" />
    <button type="submit">search</button>
</form>
```

提示：在搜索表单里显式指定表单目标

我们也特别注意在 `form action` 中显式写出 URL，因为搜索表单经常会被嵌进很多不同页面里。这也是为什么我们给它起了 `_` 前缀，并把它写成“可以被别的模板 include 进来”的形式。

一旦越过“覆写 `ListView.get_queryset()`”这一步，整个例子剩下的部分其实就只是一个很简单的 HTML 表单。我们很喜欢这种程度的简单。

10.6 只使用 django.views.generic.View

完全可以只用 `django.views.generic.View` 来搭完整个项目的所有 views。这并不像乍看上去那么极端。比如，如果你去看 Django 官方文档里关于 class-based views 的介绍（`docs.djangoproject.com/en/3.2/topics/class-based-v`会发现它和 function-based views 的写法其实已经非常接近。实际上，我们在两章前的 8.6.1 节《最简单的 Views》里专门强调过这一点，因为它真的很重要。

想象一下：与其在 function-based views 里用多层 `if` 来分别处理不同 HTTP 方法，或者在 class-based views 里把 HTTP 方法藏进 `get_context_data()`、`form_valid()` 这类方法背后，不如直接把不同 HTTP 方法明晃晃地摆在开发者面前。比如这样：

示例 10.13：使用基础 View 类

Code (python)

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.shortcuts import get_object_or_404
from django.shortcuts import render, redirect
from django.views.generic import View

from .forms import FlavorForm
from .models import Flavor

class FlavorView(LoginRequiredMixin, View):

    def get(self, request, *args, **kwargs):
        # Handles display of the Flavor object
        flavor = get_object_or_404(Flavor, slug=kwargs['slug'])
        return render(
            request,
            "flavors/flavor_detail.html",
            {"flavor": flavor}
        )

    def post(self, request, *args, **kwargs):
        # Handles updates of the Flavor object
        flavor = get_object_or_404(Flavor, slug=kwargs['slug'])
        form = FlavorForm(request.POST, instance=flavor)
```

```
        if form.is_valid():
            form.save()
            return redirect("flavors:detail", flavor.slug)
```

[原文示例在此处结束，未展示无效表单分支；译文未擅自补写后续代码]

这类写法当然也能在 function-based view 中实现，但可以说，把 GET / POST 方法直接声明在 `FlavorView` 里，读起来要比传统的 `if request.method == ...` 条件分支更直观。此外，由于继承链足够浅，即便再叠加 mixins，也不太容易把人拖进认知过载。

我们真正觉得它特别好用的场景是：哪怕项目大量使用 generic class-based views，遇到要输出 JSON、PDF 或其他非 HTML 内容时，我们仍然很愿意回到 `django.views.generic.View`，然后只实现一个 `GET` 方法。你在 function-based views 里用来渲染 CSV、Excel、PDF 的所有技巧，在这里也都同样适用。例如：

示例 10.14：用 `View` 类生成 PDF

Code (python)

```python
from django.contrib.auth.mixins import LoginRequiredMixin
from django.http import HttpResponse
from django.shortcuts import get_object_or_404
from django.views.generic import View

from .models import Flavor
from .reports import make_flavor_pdf

class FlavorPDFView(LoginRequiredMixin, View):

    def get(self, request, *args, **kwargs):
        # Get the flavor
        flavor = get_object_or_404(Flavor, slug=kwargs['slug'])

        # create the response
        response = HttpResponse(content_type='application/pdf')

        # generate the PDF stream and attach to the response
        response = make_flavor_pdf(response, flavor)

        return response
```

这是一个相当直白的例子，但如果我们需要叠更多 mixins、或者处理更多定制逻辑，那么 `django.views.generic.View` 这种轻量级写法，往往会比更厚重的 view 类轻松得多。某种意义上，这让我们同时拿到了 function-based views 的轻便，以及 CBV 提供的面向对象力量。

10.7 额外资源

- docs.djangoproject.com/en/3.2/topics/class-based-views/
- docs.djangoproject.com/en/3.2/topics/class-based-views/generic-display/
- docs.djangoproject.com/en/3.2/topics/class-based-views/generic-editing/
- docs.djangoproject.com/en/3.2/topics/class-based-views/mixins/
- docs.djangoproject.com/en/3.2/ref/class-based-views/

- GCBV inspector：ccbv.co.uk
- python.org/download/releases/2.3/mro/：虽然讲的是 Python 2.3，但对理解 Python 如何处理 MRO 仍然非常出色
- daniel.feldroy.com/tag/class-based-views.html
- spapas.github.io/2018/03/19/comprehensive-django-cbv-guide/：Serafeim Papastefanos 对 Django CBV 的一篇很好的深度文章
- djangodeconstructed.com/2020/04/27/roll-your-own-class-based-views-in-django:另一篇对 CBV 的深挖，这篇展示了如何用 DRF 创建 RESTful API

包提示：其他有用的 CBV 库

- django-extra-views：另一个很不错的 CBV 库，覆盖了 django-braces 没涉及到的那些场景。
- django-vanilla-views：一个很有意思的库，用大幅简化、也更易用的方式，提供了经典 Django GCBV 的全部力量。和 django-braces 搭配起来尤其好用。

10.8 小结

这一章我们讲了这些内容：

- 如何在 CBV 中使用 mixins
- 不同任务该选哪种 Django CBV
- 使用 CBV 的通用技巧
- 如何把 CBV 和 forms 接起来
- 如何使用基础类 django.views.generic.View

下一章会讲异步 views。第 12 章会讲常见的 CBV / form 模式。这两方面的知识，都很值得你放进自己的开发者工具箱里。

第十一章 异步视图

警告：本章仍在编写中

我们目前还在持续撰写这一章，接下来几天会继续扩展内容。也欢迎大家对想覆盖的话题和条目提出建议，请提交到：`github.com/feldroy/two-scoops-of-django-3.x/issues`

11.1 对 Django 3.1a 预发布版异步视图的分析笔记

- 不要为了那种“只是把数据读出来展示给用户”的简单场景而使用 Django async。因为通常在真正渲染 HTML 或 JSON 之前，数据本身就已经先被取回来了。
- 使用 Class-Based Views 时，尽量让它们保持最简单。可以考虑直接继承 `django.views.generic.View`。更复杂 view 底层那套 machinery 已被证明并不稳定。随着 Django 的 async 故事继续推进，这条建议未来也许会变化。

示例 11.1：一个简单更新表单的示例，它是一个可工作的 async view

Code (python)

```
class AsyncViewMixin:
    async def __call__(self):
        return super().__call__(self)

class SimpleBookUpdateView(LoginRequiredMixin, AsyncViewMixin, View):

    def get(self, request, *args, **kwargs):
        flavor = get_object_or_404(Flavor, slug=slug)
        return render(request, "flavor_form.html", {"flavor": Flavor})

    def post(self, request, *args, **kwargs):
        form = FlavorForm(request.POST)
        if form.is_valid():
            sync_to_async(form.save())
        else:
            return render({'form': form}, "flavor_form.html")
        return redirect("flavor:detail")
```

[原文示例本身存在明显未完成与可疑之处，译文未擅自修补实现，只保留其“预发布阶段实验性示例”的状态]

11.2 资源

- `docs.djangoproject.com/en/dev/topics/async/`

第十二章 Forms 的常见模式

Django forms 强大、灵活、可扩展，而且结实耐用。正因如此，Django admin 和 CBVs 都大量依赖它们。实际上，所有主流 Django API 框架也都会把 `ModelForm` 或类似实现用作验证链路的一部分。

把 forms、models 和 views 组合起来，我们往往能用很小的力气完成很多工作。它们的学习曲线绝对值得投入：一旦你能流畅地驾驭这几个组件，就会发现 Django 能让你以惊人的速度，构建出大量真正有用、而且稳定的功能。

包提示：好用的表单相关包

- `django-floppyforms`：用来把 Django 输入控件渲染成 HTML5 形式。
- `django-crispy-forms`：提供更高级的表单布局控制。默认情况下，它会把表单渲染成 Bootstrap 风格的表单元素和样式。这个包和 `django-floppyforms` 配合得很好，所以二者经常一起使用。

这一章会明确深入 Django 最棒的部分之一：forms、models 和 CBVs 如何协同工作。我们会覆盖 5 种常见表单模式，它们都应该出现在每位 Django 开发者的工具箱里。

12.1 模式 1：带默认验证器的简单 ModelForm

我们能写出的最简单“会改数据”的表单，就是直接使用 `ModelForm`，并原样依赖若干默认验证器，不做任何改动。事实上，在 10.5.1 节《Views + ModelForm 示例》中，我们就已经用过一次这种默认验证器。

如果你还记得，用 `ModelForm` 配合 CBV 来实现新增 / 编辑表单，其实只需要很少几行代码：

示例 12.1：`flavors/views.py`

Code (python)

```python
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import CreateView, UpdateView

from .models import Flavor

class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor
    fields = ['title', 'slug', 'scoops_remaining']

class FlavorUpdateView(LoginRequiredMixin, UpdateView):
    model = Flavor
    fields = ['title', 'slug', 'scoops_remaining']
```

这里“原样使用默认验证”的意思，可以总结成这样：

- `FlavorCreateView` 和 `FlavorUpdateView` 都把 `Flavor` 指定为自己的 model。
- 这两个 view 都会基于 `Flavor` 模型自动生成一个 `ModelForm`。
- 这些 `ModelForm` 依赖的，就是 `Flavor` 模型字段自带的默认验证规则。

没错，Django 在数据验证方面给了我们很多很好的默认值。但在真实项目里，这些默认值几乎从来都不够。所以，下一种模式就会先迈出第一步，演示如何创建一个自定义字段验证器。

12.2 模式 2：在 ModelForms 中使用自定义表单字段验证器

假设我们想强制要求：项目里所有甜点 app 中，凡是使用到 `title` 字段的地方，其值都必须以单词 `Tasty` 开头，该怎么办？

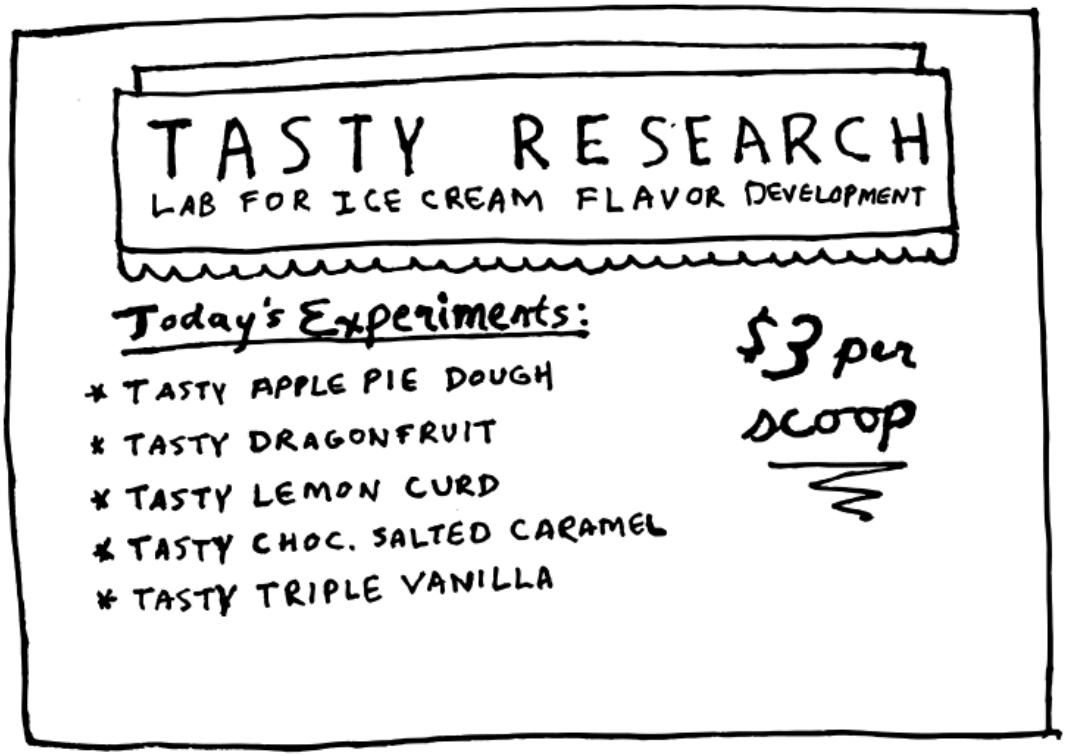

图 12.1: 图 12.1：在 Tasty Research，每个口味都必须以“Tasty”开头。

这是一个字符串验证问题，可以通过一个简单的自定义字段验证器解决。

在这个模式里，我们会讲：如何创建自定义单字段验证器，以及如何把它同时加到抽象模型和表单上。

为了举这个例子，假设我们的项目里有两个不同的甜点相关模型：一个 `Flavor` 模型表示冰淇淋口味，一个 `Milkshake` 模型表示不同种类的奶昔。并假设这两个模型都有 `title` 字段。

为了验证所有可编辑的模型标题，我们先创建一个 `validators.py` 模块：

示例 12.2：`validators.py`

Code (python)

```
# core/validators.py
from django.core.exceptions import ValidationError

def validate_tasty(value):
    """Raise a ValidationError if the value doesn't start with the
    word 'Tasty'.
    """

    if not value.startswith('Tasty'):
        msg = 'Must start with Tasty'
        raise ValidationError(msg)
```

在 Django 里，自定义字段验证器本质上就是一个可调用对象，通常是函数。只要传入的参数没有通过它的测试，它就抛出错误。

为了举例，我们这里的 `validate_tasty()` 只是做了个很简单的字符串检查。但别忘了：实际项目里的表单字段验证器，往往会复杂得多。

提示：认真测试你的验证器

验证器肩负着防止脏数据污染 Django 项目数据库的重任，所以一定要为它们编写足够细致的测试。

这些测试应该包含充分思考过的边界情况，覆盖所有和验证器自定义逻辑有关的条件。

如果我们想让 validate_tasty() 能在不同甜点模型之间复用，第一步就是把它加到一个名为 TastyTitleAbstractModel 的抽象模型上，并计划在整个项目中复用这个抽象模型。

假设 Flavor 和 Milkshake 分别位于不同 app 中，那就没必要把这个验证器只塞进某一个 app。更合理的做法，是新建 core/models.py，并把 TastyTitleAbstractModel 放进去。

示例 12.3：把自定义验证器加到模型上

Code (python)

```
# core/models.py
from django.db import models

from .validators import validate_tasty

class TastyTitleAbstractModel(models.Model):

    title = models.CharField(
        max_length=255,
        validators=[validate_tasty]
    )

    class Meta:
        abstract = True
```

上面 core/models.py 示例里的最后两行，让 TastyTitleAbstractModel 变成了抽象模型，这正是我们想要的。参见 6.1.2 节《谨慎使用模型继承》。

接下来，我们把原本 flavors/models.py 中的 Flavor 改成继承 TastyTitleAbstractModel：

示例 12.4：继承验证器

Code (python)

```
# flavors/models.py
from django.db import models
from django.urls import reverse

from core.models import TastyTitleAbstractModel

class Flavor(TastyTitleAbstractModel):
    slug = models.SlugField()
    scoops_remaining = models.IntegerField(default=0)

    def get_absolute_url(self):
        return reverse(
            'flavors:detail',
            kwargs={'slug': self.slug}
        )
```

这对 Flavor 模型有效，对其他以甜点为主题的模型也同样有效，比如 WaffleCone 或 Cake。任何继承了 TastyTitleAbstractModel 的模型，只要有人试图保存一个 title 不以 Tasty 开头的实例，就会触发验证错误。

看到这里，你脑子里可能已经冒出两个问题：

- 如果我们只想在 forms 中使用 validate_tasty() 呢？

- 如果我们想把它挂到 `title` 之外的其他字段上呢？

为了支持这些用法，我们需要创建一个自定义 `FlavorForm`，在里面显式使用我们的自定义字段验证器：

示例 12.5：把自定义验证器加到模型表单上

Code (python)

```
# flavors/forms.py
from django import forms

from .models import Flavor
from core.validators import validate_tasty

class FlavorForm(forms.ModelForm):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields['title'].validators.append(validate_tasty)
        self.fields['slug'].validators.append(validate_tasty)

    class Meta:
        model = Flavor
```

这个模式里两种验证器用法有一个很棒的地方：`validate_tasty()` 自己的代码完全没改。我们只是把它 import 到新的地方继续使用。

下一步，就是把这个自定义表单接到 views 上。Django 基于 model 的编辑 views，默认会根据 view 的 `model` 属性自动生成 `ModelForm`。而现在，我们要覆写这个默认行为，把自定义的 `FlavorForm` 显式传进去。这一步发生在 `flavors/views.py` 中，我们会像下面这样改写 create 和 update 表单：

示例 12.6：覆写 CBV 的 `form_class` 属性

Code (python)

```
# flavors/views.py
from django.contrib import messages
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import CreateView, DetailView, UpdateView

from .models import Flavor
from .forms import FlavorForm

class FlavorActionMixin:

    model = Flavor
    fields = ['title', 'slug', 'scoops_remaining']

    @property
    def success_msg(self):
        return NotImplemented

    def form_valid(self, form):
        messages.info(self.request, self.success_msg)
        return super().form_valid(form)

class FlavorCreateView(LoginRequiredMixin, FlavorActionMixin, CreateView):
```

```
    success_msg = 'created'
    # Explicitly attach the FlavorForm class
    form_class = FlavorForm

class FlavorUpdateView(LoginRequiredMixin, FlavorActionMixin, UpdateView):

    success_msg = 'updated'
    # Explicitly attach the FlavorForm class
    form_class = FlavorForm

class FlavorDetailView(DetailView):
    model = Flavor
```

现在，`FlavorCreateView` 和 `FlavorUpdateView` 都会使用新的 `FlavorForm` 来验证传入数据。

注意，做完这些修改之后，`Flavor` 模型既可以还是本章开头那个版本，也可以是后来改成继承 `TastyTitleAbstractModel` 的版本。

12.3 模式 3：覆写验证中的 clean 阶段

下面这些验证场景都很有意思：

- 多字段联合验证
- 需要依赖数据库中“已经通过基础验证的既有数据”的验证

这两类场景，都很适合通过覆写 `clean()` 和 `clean_<field_name>()` 方法来加入自定义验证逻辑。

在默认字段验证器和自定义字段验证器都跑完之后，Django 还提供了第二阶段的验证流程，也就是 `clean()` 和 `clean_<field_name>()`。你可能会想：为什么 Django 要再提供一组验证钩子？下面是我们最喜欢的两个理由：

1. `clean()` 是专门用来做“两个或更多字段之间互相校验”的地方，因为它不绑定任何单一字段。
2. clean 阶段更适合挂那些“要对照持久化数据做验证”的逻辑。因为此时数据已经先过了一轮基础验证，你不会把数据库查询浪费在太多本可提前拦下的无效输入上。

我们继续用另一个验证例子来看看。假设我们要做一个冰淇淋下单表单，用户可以选口味、加 topping，然后到店自取。

既然我们想阻止用户订购已经缺货的口味，那就可以写一个 `clean_slug()` 方法。带上这个 flavor 验证之后，表单可能长成这样：

示例 12.7：自定义 `clean_slug()` 方法

Code (python)

```
# flavors/forms.py
from django import forms

from flavors.models import Flavor

class IceCreamOrderForm(forms.Form):
    """Normally done with forms.ModelForm. But we use forms.Form here
    to demonstrate that these sorts of techniques work on every
    type of form.
    """
    slug = forms.ChoiceField(label='Flavor')
```

```
    toppings = forms.CharField()

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # We dynamically set the choices here rather than
        # in the flavor field definition. Setting them in
        # the field definition means status updates won't
        # be reflected in the form without server restarts.
        self.fields['slug'].choices = [
            (x.slug, x.title) for x in Flavor.objects.all()
        ]
        # NOTE: We could filter by whether or not a flavor
        # has any scoops, but this is an example of
        # how to use clean_slug, not filter().

    def clean_slug(self):
        slug = self.cleaned_data['slug']
        if Flavor.objects.get(slug=slug).scoops_remaining <= 0:
            msg = 'Sorry, we are out of that flavor.'
            raise forms.ValidationError(msg)
        return slug
```

在 HTML 驱动的 views 里，这个 `clean_slug()` 一旦抛出错误，就会把一条“Sorry, we are out of that flavor.”消息挂到 flavor 对应的 HTML 输入字段上。对于写 HTML 表单来说，这是个特别省事的捷径。

再进一步想想。假设我们收到了大量用户投诉，说订单里的巧克力实在太多了。是的，这个设定既荒唐又近乎不可能，但我们只是借“巧克力太多”这个神话般的例子来说明问题。

总之，这里就可以用 `clean()` 方法，把 `flavor` 和 `toppings` 两个字段放到一起验证。

示例 12.8：自定义表单 `clean()` 方法

Code (python)

```
# attach this code to the previous example (12.7)
def clean(self):

    cleaned_data = super().clean()
    slug = cleaned_data.get('slug', '')
    toppings = cleaned_data.get('toppings', '')

    # Silly "too much chocolate" validation example
    in_slug = 'chocolate' in slug.lower()
    in_toppings = 'chocolate' in toppings.lower()
    if in_slug and in_toppings:

        msg = 'Your order has too much chocolate.'
        raise forms.ValidationError(msg)
    return cleaned_data
```

好了，这就是对那个“不可能发生的巧克力过量条件”的一个实现。

提示：多字段验证里常见的字段组合

用户账户相关表单里，经常会要求用户把 email 或 password 之类的数据输入两次，以便确认一致。除此之外，这些字段常见的额外检查还包括：

- 提交密码的强度。
- 如果 email 对应的模型字段没有设置 `unique=True`，那就要额外检查这个 email 是否唯一。

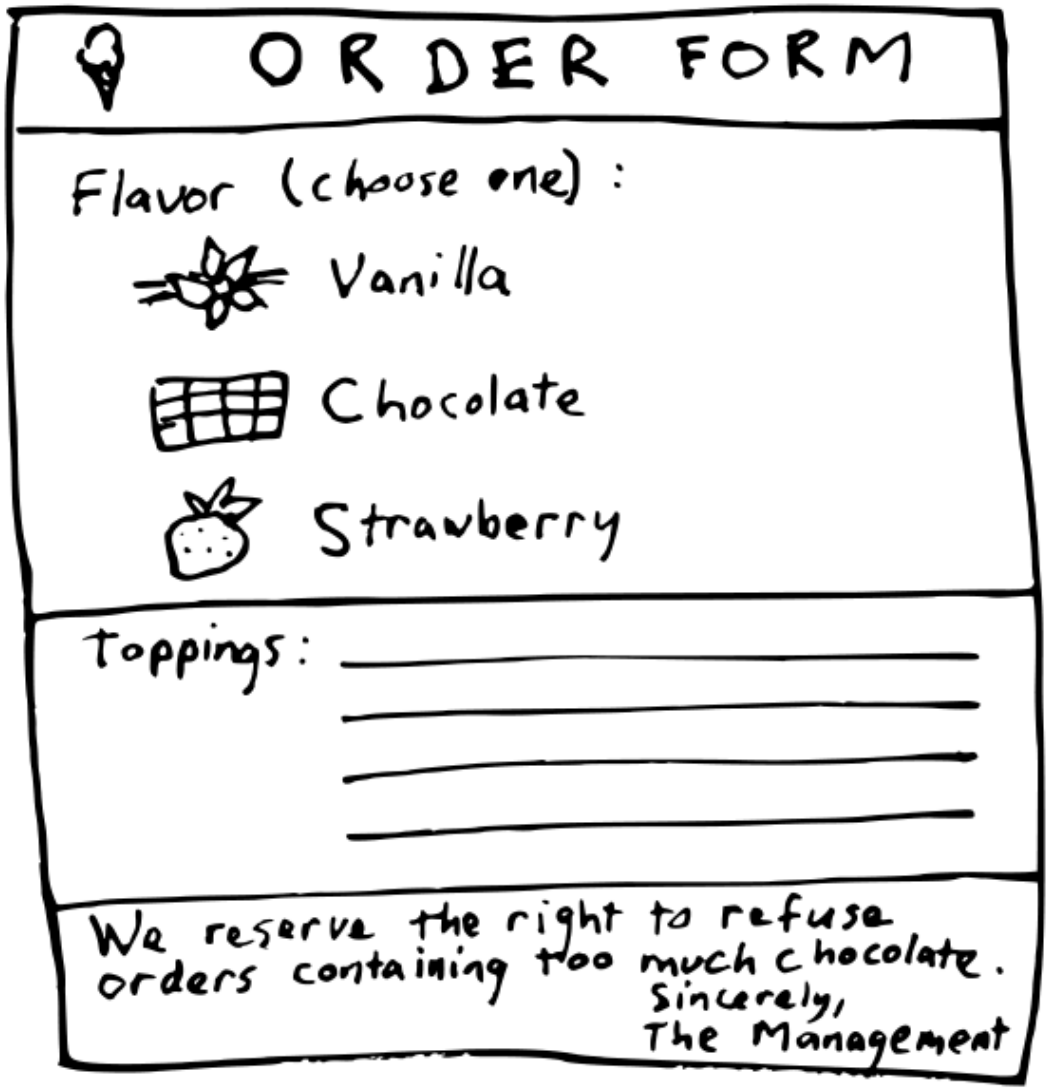

图 12.2: 图 12.2：他们为什么要这样对我们?

12.4 模式 4：Hack 表单字段

（2 个 CBV，2 个 Form，1 个 Model）

从这里开始，我们要稍微玩得花一点。这个场景里，是两个 view / 两个 form 对应一个 model。我们会“hack” Django forms，让它们表现出自定义行为。

用户先创建一条记录，里面有几个字段先空着，后面再补数据，这种情况一点也不少见。比如一个门店列表：我们希望先尽快把每家门店录进系统，但电话号码、描述等信息可以之后再补。下面是我们的 `IceCreamStore` 模型：

示例 12.9：`IceCreamStore` 模型

Code (python)

```python
# stores/models.py
from django.db import models
from django.urls import reverse

class IceCreamStore(models.Model):
    title = models.CharField(max_length=100)
    block_address = models.TextField()
    phone = models.CharField(max_length=20, blank=True)
    description = models.TextField(blank=True)

    def get_absolute_url(self):
        return reverse(
            'stores:store_detail',
            kwargs={'pk': self.pk}
        )
```

这个模型默认生成的 ModelForm 会强制要求用户填写 title 和 block_address，但允许跳过 phone 和 description。这对“初次录入”来说很好，但正如前面说的，我们希望到了后续编辑阶段，phone 和 description 必须填写。

在我们还没真正理解 Django forms 的构造方式之前，过去通常会直接在编辑表单里覆写 phone 和 description 字段。结果写出来的代码大概是这样，重复得非常夸张：

示例 12.10：大量重复的代码

Code (python)

```python
# stores/forms.py
from django import forms

from .models import IceCreamStore

class IceCreamStoreUpdateForm(forms.ModelForm):
    # Don't do this! Duplication of the model field!
    phone = forms.CharField(required=True)
    # Don't do this! Duplication of the model field!
    description = forms.TextField(required=True)

    class Meta:
        model = IceCreamStore
```

这个表单是不是看起来特别眼熟？

因为我们几乎是在复制 IceCreamStore 模型本身！

这还只是个简单例子。字段一多，这种重复就会迅速变得难以管理。现实里更常见的情况是：直接把 models 里的字段定义复制粘贴进 forms，这简直是对 Don’t Repeat Yourself 的严重践踏。

有多严重？举个很简单的例子：如果你后来在模型里的 description 字段上加了一个 help_text 属性，那么它不会自动出现在模板里，除非你还去同步修改 form 里的 description 字段定义。听起来已经很乱了？因为它本来就很乱。

更好的方式，是利用 Django forms 一个很值得记住的小细节：实例化后的表单对象，会把字段放在一个类似字典的属性 fields 里。

所以，与其从 model 把字段定义整段复制进 form，不如在 ModelForm 的 __init__() 方法里，直接修改指定字段已有的属性：

示例 12.11：通过覆写 __init__() 修改既有字段属性

Code (python)

```
# stores/forms.py
# Call phone and description from the self.fields dict-like object
from django import forms

from .models import IceCreamStore

class IceCreamStoreUpdateForm(forms.ModelForm):

    class Meta:
        model = IceCreamStore

    def __init__(self, *args, **kwargs):
        # Call the original __init__ method before assigning
        # field overloads
        super().__init__(*args, **kwargs)
        self.fields['phone'].required = True
        self.fields['description'].required = True
```

这种改进后的方式，让我们终于不用继续复制粘贴字段定义，而只需要专注于“这个字段有哪些属性需要改”。

还要记住一点：Django forms 说到底也只是 Python 类。它们会被实例化成对象、可以继承别的类、也可以成为别人的父类。

因此，我们完全可以依靠继承，进一步压缩冰淇淋门店表单的代码量：

示例 12.12：利用继承让表单更干净

Code (python)

```
# stores/forms.py
from django import forms

from .models import IceCreamStore

class IceCreamStoreCreateForm(forms.ModelForm):

    class Meta:
        model = IceCreamStore
        fields = ['title', 'block_address']

class IceCreamStoreUpdateForm(IceCreamStoreCreateForm):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields['phone'].required = True
        self.fields['description'].required = True

    class Meta(IceCreamStoreCreateForm.Meta):
```

```
        # show all the fields!
        fields = ['title', 'block_address', 'phone', 'description']
```

警告：使用 Meta.fields，永远不要用 Meta.exclude

我们使用 Meta.fields，而不是 Meta.exclude，因为我们必须清清楚楚地知道自己到底暴露了哪些字段。参见 28.14 节《不要使用 ModelForms.Meta.exclude》。

最后，既然表单类已经有了，对应的 CBV 也就水到渠成。我们已经准备好了 forms，那就直接把它们接进 IceCreamStore 的 create / update views：

示例 12.13：修订后的 Create / Update Views

Code (python)

```
# stores/views
from django.views.generic import CreateView, UpdateView

from .forms import IceCreamStoreCreateForm, IceCreamStoreUpdateForm
from .models import IceCreamStore

class IceCreamCreateView(CreateView):
    model = IceCreamStore
    form_class = IceCreamStoreCreateForm

class IceCreamUpdateView(UpdateView):
    model = IceCreamStore
    form_class = IceCreamStoreUpdateForm
```

现在，我们就得到了：两个 view、两个 form、共同服务于一个 model。

12.5 模式 5：可复用的搜索 Mixin View

在这个例子里，我们要讲：如何复用一个搜索表单，让它同时服务于两个 view，而这两个 view 又分别对应两个不同模型。

假设这两个模型都有一个叫 title 的字段。（这也再次说明：在项目里坚持命名规范真的是件好事。）这个例子会展示：如何用一个单独的 CBV，为 Flavor 和 IceCreamStore 两个模型都加上简单搜索功能。

我们先从一个简单的搜索 mixin 开始：

示例 12.14：TitleSearchMixin，一个简单搜索类

Code (python)

```
# core/views.py
class TitleSearchMixin:

    def get_queryset(self):
        # Fetch the queryset from the parent's get_queryset
        queryset = super().get_queryset()

        # Get the q GET parameter
        q = self.request.GET.get('q')
        if q:
            # return a filtered queryset
            return queryset.filter(title__icontains=q)
```

```
        # No q is specified so we return queryset
        return queryset
```

这段代码应该看起来很眼熟，因为我们在前面的“Forms + View”例子里几乎原样用过。接下来，把它挂到 `Flavor` 和 `IceCreamStore` 两边去。

先看 flavor view：

示例 12.15：把 `TitleSearchMixin` 加到 `FlavorListView`

Code (python)

```
# add to flavors/views.py
from django.views.generic import ListView

from .models import Flavor
from core.views import TitleSearchMixin

class FlavorListView(TitleSearchMixin, ListView):
    model = Flavor
```

再把它加到冰淇淋门店 views 上：

示例 12.16：把 `TitleSearchMixin` 加到 `IceCreamStoreListView`

Code (python)

```
# add to stores/views.py
from django.views.generic import ListView

from .models import Store
from core.views import TitleSearchMixin

class IceCreamStoreListView(TitleSearchMixin, ListView):
    model = Store
```

至于表单？我们直接在每个 `ListView` 对应的 HTML 里手写就好：

示例 12.17：`stores/store_list.html` 片段

Code (django)

```
{# form to go into stores/store_list.html template #}
<form action="" method="GET">
    <input type="text" name="q" />
    <button type="submit">search</button>
</form>
```

以及：

示例 12.18：`flavors/flavor_list.html` 片段

Code (django)

```
{# form to go into flavors/flavor_list.html template #}
<form action="" method="GET">
    <input type="text" name="q" />
    <button type="submit">search</button>
</form>
```

现在，同一个 mixin 已经被接到了两个不同的 view 里。mixin 的确是代码复用的好方法，但如果一个类里堆了太多 mixin，代码就会变得极难维护。照旧，尽量让代码保持简单。

12.6 小结

这一章我们从最简单的表单模式开始：使用 `ModelForm`、CBV 和默认验证器。然后在此基础上，进一步演示了自定义验证器的例子。

接着，我们讨论了更复杂的验证方式，尤其是如何覆写 `clean` 系列方法。我们也仔细拆解了一个场景：两个 view 与它们对应的 forms，一起服务同一个 model。

最后，我们讲了一个可复用搜索 mixin 的例子，把同一个表单能力加到了两个不同 app 上。

第十三章 Form 基础

100% 的 Django 项目都应该使用 Forms。95% 的 Django 项目都应该使用 ModelForms。91% 的 Django 项目都会使用 ModelForms。80% 的 ModelForms 只需要非常简单的逻辑。20% 的 ModelForms 需要复杂得多的逻辑。

Daniel 自创统计学 ™

Django 的 forms 非常强大，而只要数据是从应用外部流入的，知道该如何用 forms 去处理它，就是保持数据整洁的重要一环。

当然，也会有一些边缘场景让人头疼。但只要你理解 forms 的组成结构，以及它们是如何被调用的，大多数边缘情况其实都能比较顺利地拿下。

关于 Django forms，最重要的一点是：所有传入数据都应该通过它们来验证。

13.1 用 Django Forms 验证所有传入数据

Django forms 是一套用来验证 Python 字典的优秀框架。虽然大多数时候我们用它来验证带有 POST 数据的 HTTP 请求，但它绝不只局限于这种用法。

比如说，假设我们有一个 Django app，会通过从另一个项目抓来的 CSV 文件来更新自己的 model。这种场景里，常见但糟糕的代码大概会像这样：

示例 13.1：不要这样导入 CSV

Code (python)

```
import csv
from django.utils.six import StringIO

from .models import Purchase

def add_csv_purchases(rows):

    rows = StringIO.StringIO(rows)
    records_added = 0

    # Generate a dict per row, with the first CSV row being the keys
    for row in csv.DictReader(rows, delimiter=','):

        # DON'T DO THIS: Tossing unvalidated data into your model.
        Purchase.objects.create(**row)
        records_added += 1
    return records_added
```

更糟的是，这段代码背后其实还藏着一个问题：我们根本没检查 `Purchase` 模型里以字符串形式存储的 seller，到底是不是真的合法 seller。

你当然可以把验证代码硬塞进 `add_csv_purchases()`，但老实说，随着需求与数据变化，想让一大坨自制验证逻辑一直保持可读，可太难了。

更好的方式，是像下面这样，直接用 Django Form 来验证传入数据：

示例 13.2：如何安全地导入 CSV

Code (python)

```
import csv
```

```
from django.utils.six import StringIO

from django import forms

from .models import Purchase, Seller

class PurchaseForm(forms.ModelForm):

    class Meta:
        model = Purchase

    def clean_seller(self):
        seller = self.cleaned_data['seller']
        try:
            Seller.objects.get(name=seller)
        except Seller.DoesNotExist:
            msg = '{0} does not exist in purchase #{1}.'.format(
                seller,
                self.cleaned_data['purchase_number']
            )
            raise forms.ValidationError(msg)
        return seller

def add_csv_purchases(rows):

    rows = StringIO.StringIO(rows)

    records_added = 0
    errors = []
    # Generate a dict per row, with the first CSV row being the keys.
    for row in csv.DictReader(rows, delimiter=','):

        # Bind the row data to the PurchaseForm.
        form = PurchaseForm(row)
        # Check to see if the row data is valid.
        if form.is_valid():

            # Row data is valid so save the record.
            form.save()
            records_added += 1
        else:
            errors.append(form.errors)

    return records_added, errors
```

这种做法最棒的一点在于：我们不是重新发明一套自己的输入验证系统，而是直接复用 Django 内建、经过长期验证的数据校验框架。

提示：那 code 参数怎么办?

Arnaud Limbourg 提醒过，Django 官方文档建议在 `ValidationError` 中传入 `code` 参数，例如：

```
forms.ValidationError(_('Invalid value'), code='invalid')
```

在我们的例子里没有这样写，但如果你愿意，完全可以在自己的代码里使用它。

Django 核心开发者 Marc Tamlyn 的大意是："就我个人而言，我觉得 Django 文档可能有点过度用力地把 `code` 推荐成了处处都该遵守的最佳实践，虽然在第三方应用里确实应该鼓励这么做。不过，只要你的场景需要判断错误类型，那它绝对就是最佳实践，因为这总比去检查错误消息文本强得多，毕竟后者很容易随着文案改动而失效。"

参考：

- `docs.djangoproject.com/en/3.2/ref/forms/validation/#raising-validationerror`

13.2 在 HTML Forms 中使用 POST 方法

所有会修改数据的 HTML 表单，都必须通过 POST 方法提交数据：

示例 13.3：如何在 HTML 中使用 POST

Code (html)

```
<form action="{% url 'flavor_add' %}" method="POST">
```

你唯一会看到"不用 POST"的表单，是搜索表单。因为搜索通常不会引起数据修改，所以这类幂等表单应当使用 GET 方法。

13.3 对会修改数据的 HTTP Forms，始终使用 CSRF 保护

Django 内建了跨站请求伪造（CSRF）保护，而且 Django 入门教程第 4 部分就已经介绍了它。它非常容易使用，而一旦你忘记启用，Django 在开发阶段甚至还会友好地给你一个提醒。这是一个关键安全问题，所以无论在这里还是在后面的安全章节里，我们都强烈建议始终使用 Django 的 CSRF 保护能力。

按我们的经验，唯一比较常见"不使用 CSRF"的场景，是在创建机器可访问的 API，并且这个 API 已经通过像 `github.com/jazzband/dj-rest-auth` 这样成熟的库完成认证时。此类工具和 Django REST Framework 搭配后，通常会替你处理这件事。因为 API 请求本来就应该逐请求签名 / 认证，所以再依赖 HTTP cookie 来做认证通常并不现实。也正因为如此，在这些框架下，CSRF 并不总会成为问题。

如果你是在从零开始写一个允许数据变更的 API，那就很有必要先熟悉 Django 的 CSRF 文档：`docs.djangoproject.com/e`

提示：HTML 搜索表单

因为 HTML 搜索表单不会改数据，所以它们使用 HTTP GET 方法，也不会触发 Django 的 CSRF 保护。

你应该把 Django 的 `CsrfViewMiddleware` 当作全站兜底保护来使用，而不是靠手动给每个 view 加 `csrf_protect` 装饰器。至于怎么让 CSRF 在 Jinja2 模板里正常工作，参见 16.3 节《在 Django 中使用 Jinja2 时的注意事项》。

13.3.1 通过 AJAX 提交数据

即便是通过 AJAX 提交数据，也应该继续使用 Django 的 CSRF 保护。不要把你的 AJAX views 标成 CSRF 豁免。

正确做法是：当你通过 AJAX POST 数据时，需要设置一个名为 `X-CSRFToken` 的 HTTP header。

Django 官方文档里给了一段示例代码，展示如何在只对 POST 请求设置该 header 的同时，结合 jQuery 1.5.1 及以上版本的跨域检查一起工作：`docs.djangoproject.com/en/3.2/ref/csrf/#ajax`

更多内容见 19.3.5 节《AJAX 与 CSRF Token》。

13.4 理解如何给 Django Form 实例添加属性

有时候，在 Django form 的 `clean()`、`clean_FOO()` 或 `save()` 方法里，我们需要额外的 form 实例属性可用。一个典型例子，就是让 `request.user` 对象在 form 中可访问。下面我们用一个和试吃员有关的简单例子说明。

先看 form：

示例 13.4：Taster 表单

Code (python)

```python
from django import forms

from .models import Taster

class TasterForm(forms.ModelForm):

    class Meta:
        model = Taster

    def __init__(self, *args, **kwargs):
        # set the user as an attribute of the form
        self.user = kwargs.pop('user')
        super().__init__(*args, **kwargs)
```

注意我们是在调用 `super()` 之前，先把 `self.user` 设好，并且从 `kwargs` 中把它取出来。Christopher Lambacher 曾向我们指出，这样写会让 form 更健壮，尤其是在使用多重继承时。

再来看 view：

示例 13.5：Taster 更新 View

Code (python)

```python
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import UpdateView

from .forms import TasterForm
from .models import Taster

class TasterUpdateView(LoginRequiredMixin, UpdateView):
    model = Taster
    form_class = TasterForm
    success_url = '/someplace/'

    def get_form_kwargs(self):
        """This method is what injects forms with keyword arguments."""
        # grab the current set of form #kwargs
        kwargs = super().get_form_kwargs()
        # Update the kwargs with the user_id
        kwargs['user'] = self.request.user
        return kwargs
```

包提示：`django-braces` 的 ModelForm Mixins

把 `request.user` 注入 form 这件事常见到什么程度？常见到 `django-braces` 已经能直接替我们做了。尽管如此，理解它底层是怎么工作的仍然非常有价值，尤其是当你要塞进去的东西不只是 `request.user` 时。

- django-braces.readthedocs.io/en/latest/form.html#userformkwargsmixin
- django-braces.readthedocs.io/en/latest/form.html#userkwargmodelformmixin

13.5 理解表单验证是如何工作的

表单验证是 Django 中那种“只要你搞清楚内部机制，代码质量就会大幅提升”的地方。我们来稍微往里挖一层，把几个关键点过一遍。

当你调用 `form.is_valid()` 时，后台其实会发生很多事情。它们的流程大致如下：

1. 如果表单绑定了数据，`form.is_valid()` 会调用 `form.full_clean()`。
2. `form.full_clean()` 会遍历表单字段，并让每个字段自行验证：

a. 进入字段的数据会通过 `to_python()` 被强制转换为 Python 值；如果失败，就抛出 `ValidationError`。b. 数据会再根据字段自己的规则验证，包括自定义验证器；失败同样抛出 `ValidationError`。c. 如果表单里定义了任何自定义 `clean_<field>()` 方法，也会在这个阶段被调用。

1. `form.full_clean()` 会执行表单的 `form.clean()` 方法。
2. 如果这是一个 `ModelForm` 实例，`form._post_clean()` 还会继续做下面这些事：

a. 无论 `form.is_valid()` 最终是 `True` 还是 `False`，`ModelForm` 的数据都会先被设置到 model 实例上。b. 然后调用 model 的 `clean()` 方法。顺便提醒一下：通过 ORM 直接 `save()` 一个 model 实例时，并不会自动调用 model 的 `clean()` 方法。

如果这看起来有点复杂，也别慌。真正写起来的时候，其实会简单很多。而且这整套机制，恰恰能让我们更清楚地理解：传入数据到底经历了什么。下一节的例子会帮助你进一步消化这一点。

图 13.1: 图 13.1：当冰淇淋验证失败时。

13.5.1 ModelForm 数据先保存到 Form，再保存到 Model 实例

我们喜欢把这件事称作表单验证里的“WHAT?!?”时刻。乍一看，“表单数据先被设置到 form 实例上”好像像个 bug，但它并不是 bug，而是刻意设计出来的行为。

在 `ModelForm` 中，表单数据会分两步保存：

1. 先保存到 form 实例上。
2. 之后才保存到 model 实例上。

既然 `ModelForm` 要等到 `form.save()` 被调用时，才真正把数据写进 model 实例，我们就可以把这层分离当作一个很有用的特性来利用。

比如，你也许想捕获表单提交失败时的详细信息，把“用户提交的表单数据”和“打算写入 model 的最终实例变化”都记录下来。

一个简单到近乎简陋的做法，大概如下。首先，我们在 core/models.py 中定义一个表单失败历史模型：

示例 13.6：表单失败历史模型

Code (python)

```
# core/models.py
from django.db import models

class ModelFormFailureHistory(models.Model):
    form_data = models.TextField()
    model_data = models.TextField()
```

接着，在 flavors/views.py 中，把下面这段加到 FlavorActionMixin 里：

示例 13.7：FlavorActionMixin

Code (python)

```
# flavors/views.py
import json

from django.contrib import messages
from django.core import serializers

from core.models import ModelFormFailureHistory

class FlavorActionMixin:

    @property
    def success_msg(self):
        return NotImplemented

    def form_valid(self, form):
        messages.info(self.request, self.success_msg)
        return super().form_valid(form)

    def form_invalid(self, form):
        """Save invalid form and model data for later reference."""
        form_data = json.dumps(form.cleaned_data)
        # Serialize the form.instance
        model_data = serializers.serialize('json', [form.instance])
        # Strip away leading and ending bracket leaving only a dict
        model_data = model_data[1:-1]
        ModelFormFailureHistory.objects.create(
            form_data=form_data,
            model_data=model_data
        )
        return super().form_invalid(form)
```

如你所知，form_invalid() 会在“表单因坏数据验证失败”之后被调用。在这个例子里，一旦走到这里，cleaned_data 中已清洗过的表单数据，以及最终写到 model 实例上的数据，都会一起被存成一条 ModelFormFailureHistory 记录。

13.6 用 Form.add_error() 给表单添加错误

Michael Barr 分享给我们的一个小技巧是：你可以用 Form.add_error() 来让 Form.clean() 写得更利落。

示例 13.8：使用 Form.add_error()

Code (python)

```
from django import forms

class IceCreamReviewForm(forms.Form):
    # Rest of tester form goes here
    ...

    def clean(self):
        cleaned_data = super().clean()
        flavor = cleaned_data.get('flavor')
        age = cleaned_data.get('age')

        if flavor == 'coffee' and age < 3:
            # Record errors that will be displayed later.
            msg = 'Coffee Ice Cream is not for Babies.'
            self.add_error('flavor', msg)
            self.add_error('age', msg)

        # Always return the full collection of cleaned data.
        return cleaned_data
```

13.6.1 其他值得了解的表单方法

下面这些表单验证相关方法，也很值得进一步看看：

- docs.djangoproject.com/en/3.2/ref/forms/api/#django.forms.Form.errors.as_data
- docs.djangoproject.com/en/3.2/ref/forms/api/#django.forms.Form.errors.as_json
- docs.djangoproject.com/en/3.2/ref/forms/api/#django.forms.Form.non_field_errors

13.7 没有现成 Widget 的字段

django.contrib.postgres 里有两个较新的字段，ArrayField 和 HStoreField，它们和现有 Django HTML 字段的配合并不好，因为 Django 根本没有为它们提供对应 widget。

即便如此，你仍然应该继续对这些字段使用 forms。

至于怎么做，思路其实和前面一样，参见 13.1 节《用 Django Forms 验证所有传入数据》。

13.8 自定义 Widgets

我们特别喜欢现代 Django 的一个点是：无论是覆写内建 widget 的 HTML，还是创建全新的自定义 widget，都已经变得非常轻松。我们的总体建议是：

- 照旧，保持简单。只专注展示层，不要越界。
- widget 永远不应该修改数据。它们的职责纯粹是展示。
- 遵循 Django 约定，把所有自定义 widget 都放进 widgets.py 模块。

13.8.1 覆写内建 Widget 的 HTML

这项技巧在集成 Bootstrap、Zurb 以及其他响应式前端框架时非常有用。缺点则是：一旦你用这种方式覆写了默认模板，所有表单元素都会统一应用这些改动。

要想覆写这些模板，你需要先在 `settings.py` 里做如下修改：

示例 13.9：覆写 Django 表单 Widget 的 HTML

Code (python)

```python
# settings.py
FORM_RENDERER = 'django.forms.renderers.TemplatesSetting'

INSTALLED_APPS = [
    ...
    'django.forms',
    ...
]
```

完成这一步后，再在你的 templates 目录里创建对应目录，并开始覆写模板。如果你想知道哪些模板可以被覆写，可以直接去 Django 源码里看：`github.com/django/django/tree/master/django/forms/templates/django/forms/wid`

更多信息：

- `docs.djangoproject.com/en/3.2/ref/forms/renderers/#overriding-built-in-widget-templates`
- `docs.djangoproject.com/en/3.2/ref/forms/renderers/#templatessetting`

13.8.2 创建新的自定义 Widgets

如果你想要对 widget 进行更细粒度控制，比如只对特定数据类型生效，那就该自己动手做自定义 widget 了。一般步骤如下：

1. 去 `https://github.com/django/django/blob/master/django/forms/widgets.py`，找到一个最接近你需求的 widget。
2. 基于它扩展出你想要的行为。改动尽可能少！

下面是一个例子：

示例 13.10：创建一个冰淇淋 Widget

Code (python)

```python
# flavors/widgets.py
from django.forms.widgets import TextInput

class IceCreamFlavorInput(TextInput):
    """Ice cream flavors must always end with 'Ice Cream'"""

    def get_context(self, name, value, attrs):
        context = super().get_context(name, value, attrs)
        value = context['widget']['value']
        if not value.strip().lower().endswith('ice cream'):
            context['widget']['value'] = '{} Ice Cream'.format(value)
        return context
```

这个例子虽然有点搞笑，但它很好地说明了：如何在现有 widget 之上做最小量扩展，来满足自己的需求。请特别注意以下几点：

- widget 所做的，只是修改“值如何被展示”。
- widget 不会去验证，也不会改写浏览器回传的数据。那分别是 forms 和 models 的职责。
- 我们只是对 `django.forms.widgets.TextInput` 做了最小限度的扩展，刚好让它工作起来。

13.9 额外资源

- Classy Django Forms 是一个 API inspector，会详细列出 Django 表单系统里各种元素的说明：`df.9vo.lt/`
- Matt Layman 关于 forms 的深度文章很值得读：`mattlayman.com/understand-django/user-interaction-forms`
- 作者 Daniel Feldroy 关于 forms 的博客系列，对本书中的一些概念做了更进一步展开：`daniel.feldroy.com/tag/forms.h`
- Brad Montgomery 关于如何为 `ArrayField` 创建 widget 的文章：`bradmontgomery.net/blog/2015/04/25/nice-arrayfie`
- 渲染自定义 Django widgets：`docs.djangoproject.com/en/3.2/ref/forms/renderers/`

13.10 小结

一旦你真的深入 forms，就要始终把注意力放在“代码清晰”和“便于测试”上。Forms 是 Django 项目里最核心的验证工具之一，也是抵御攻击与意外数据损坏的重要防线。

下一章，我们会开始深入 templates。

第十四章 Templates：最佳实践

Django 在早期做出的一个设计决定，就是刻意限制模板语言的功能。这会大幅约束 Django templates 能做的事情。我们通常认为这反而是件很好的事，因为它会逼着我们把业务逻辑留在 Python 那一侧。

想想看：Django templates 的这些限制，会迫使我们把项目里最关键、最复杂、最细致的部分放进 .py 文件，而不是模板文件。Python 本来就是地球上最清晰、最简洁、最优雅的编程语言之一，那我们为什么还要反着来呢？

提示：在 Django 中使用 Jinja2

自 1.8 发布以来，Django 就已经原生支持 Jinja2，也提供了接入其他模板语言的接口。这个主题我们会在第 16 章《Django Templates 与 Jinja2》中展开。

14.1 尽量把 Templates 放在 templates/ 目录下

在我们的项目里，大多数 templates 都放在主 templates/ 目录下。我们会在 templates/ 里按 app 建子目录，大致如下：

示例 14.1：我们的 Templates 目录结构

Code (text)

```
templates/
+-- base.html
+-- ... (other sitewide templates in here)
+-- freezers/
|   +-- ... ("freezers" app templates in here)
```

不过，也有一些教程提倡把 templates 放进每个 app 自己的子目录里。我们觉得那种额外嵌套很折磨人，大概像这样：

示例 14.2：过于复杂的 Template 目录结构

Code (text)

```
freezers/
+-- templates/
|   +-- freezers/
|   |   +-- ... ("freezers" app templates in here)
templates/
+-- base.html
+-- ... (other sitewide templates in here)
```

当然，也有人就喜欢第二种做法，那也完全没问题。

这里的例外，是当我们使用那些以 pluggable package 形式安装的 Django app 时。Django package 通常会自带一个 app 内部的 templates/ 目录。然后我们往往还是会在项目主 templates/ 目录里覆盖那些模板，以便加上自己的设计和样式。我们会在 23.9 节《发布你自己的 Django Packages》中继续讨论这件事。

14.2 Template 架构模式

就我们的用途而言，简单的两层或三层 template 架构最理想。所谓层数，区别就在于：app 中的内容真正显示出来之前，需要发生多少层 template 继承。看下面这些例子就很清楚了。

14.2.1 两层 Template 架构示例

在两层 template 架构里，所有模板都继承自一个根 `base.html` 文件。

示例 14.3：两层 Template 架构

Code (text)

```
templates/
+-- base.html
+-- dashboard.html # extends base.html
+-- profiles/
|   +-- profile_detail.html # extends base.html
|   +-- profile_form.html # extends base.html
```

这种方式最适合整体布局在各 app 之间都很一致的网站。

14.2.2 三层 Template 架构示例

在三层 template 架构里：

- 每个 app 都有一个 `base_<app_name>.html` 模板；这些 app 级基础模板共享同一个父模板 `base.html`。
- app 内部的模板共享同一个父模板 `base_<app_name>.html`。
- 与 `base.html` 处在同一层级的模板，直接继承 `base.html`。

示例 14.4：三层 Template 架构

Code (text)

```
templates/
+-- base.html
+-- dashboard.html # extends base.html
+-- profiles/
|   +-- base_profiles.html # extends base.html
|   +-- profile_detail.html # extends base_profiles.html
|   +-- profile_form.html # extends base_profiles.html
```

三层架构最适合那种每个板块都需要鲜明布局差异的网站。比如新闻网站，可能会有本地新闻、分类广告和活动信息这几个板块。每个板块都需要自己定制的布局。

当我们希望站点中某个功能分组对应的 HTML 在外观或行为上与其他部分明显不同时，这种做法会特别有用。

14.2.3 扁平优于嵌套

复杂的 template 层级，会让 HTML 页面以及相关 CSS 样式变得极难调试、修改和扩展。一旦 template block 的布局变成不必要的多层嵌套，你就会为了改一个盒子的宽度之类的小事，不得不翻来覆去地钻一个又一个文件。

尽可能让你的 template blocks 保持浅层继承结构，templates 会更容易维护，也更容易协作。如果你正在和设计师一起工作，设计师会感谢你的。

当然，过度复杂的 template block 层级，和“为了复用代码而合理使用 blocks 的模板”之间，是有区别的。如果多个模板里有大块、多行、完全相同或非常相似的代码，把它们重构成可复用的 blocks，会让代码更易维护。

《The Zen of Python》里那句“Flat is better than nested”说得很有道理。每增加一层嵌套，脑力负担就会上升一点。设计 Django templates 架构时，要始终记着这一点。

提示：The Zen of Python

在命令行里执行下面这条命令：

图 14.1: 图 14.1：冰淇淋之禅摘录。

Code (bash)

```
python -c 'import this'
```

你会看到 The Zen of Python。那是一组表达得非常优雅的指导原则，用来说明 Python 这门语言应当如何设计。

14.3 限制在 Templates 中做处理

你试图在 templates 里做的处理越少越好。尤其是查询和迭代，只要发生在模板层，往往就容易出问题。

每当你准备在 template 中遍历一个 queryset 时，都先问自己下面这几个问题：

1. 这个 queryset 有多大？在 templates 里遍历超大的 queryset，几乎总是个坏主意。
2. 取回来的对象有多大？这个 template 真的需要这些字段全部出现吗？
3. 每次循环迭代时，会发生多少处理？

如果你脑子里已经开始响警报，那八成说明这段 template 代码有更好的改写方式。

警告：为什么不直接做缓存？

有时候，你的确可以直接用缓存把 template 低效的问题“糊过去”。这么做不是不行，但在上缓存之前，你应该先试着从根源上解决问题。

只要在脑子里顺着 template 代码走一遍，做一点快速运行时分析，再做些重构，你往往就能省下很多工作。

现在我们来看几个可以重写得更高效的 template 代码示例。先暂时放下怀疑，假装《Two Scoops》背后那对不太正常的搭档，在超级碗中场投了一个 30 秒广告：“前一百万个提出申请的开发者，都能免费领一品脱冰淇淋！你只需要填个表单，就能拿到可在门店兑换的代金券！”

很自然地，我们有一个 `vouchers` app，用来追踪所有申请免费一品脱代金券的人的姓名和邮箱地址。这个 app 的 model 大致如下：

示例 14.5：`Voucher` Model 示例

Code (python)

```
# vouchers/models.py
from django.db import models

from .managers import VoucherManager
```

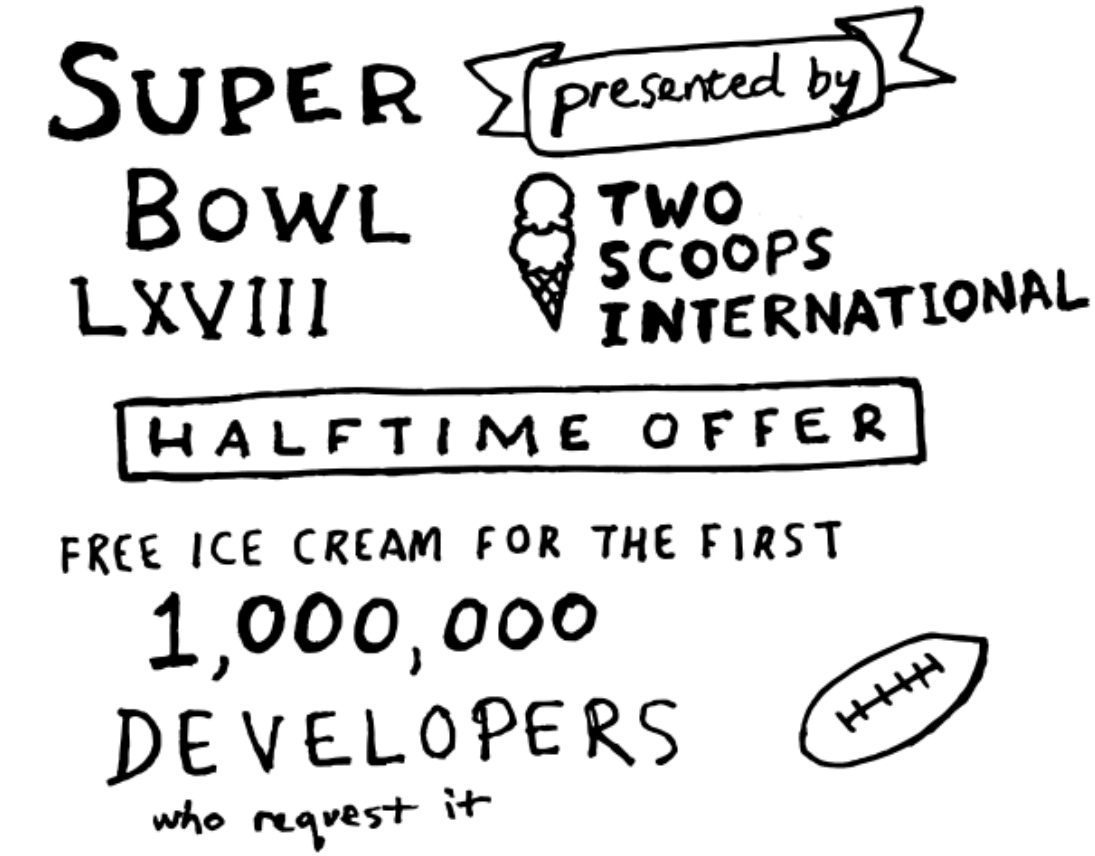

图 14.2: 图 14.2：Two Scoops 是超级碗官方中场赞助商。

```
class Voucher(models.Model):
    """Vouchers for free pints of ice cream."""

    name = models.CharField(max_length=100)
    email = models.EmailField()
    address = models.TextField()
    birth_date = models.DateField(blank=True)
    sent = models.DateTimeField(null=True, default=None)
    redeemed = models.DateTimeField(null=True, default=None)

    objects = VoucherManager()
```

后面的例子会用这个 model 来说明几种你应该避开的“gotchas”。

14.3.1 坑 1：在 Templates 中做聚合

既然我们掌握了出生日期信息，那么把代金券申请和兑换情况按年龄段做一个粗略统计展示出来，会很有意思。

一种非常糟糕的实现方式，是把所有处理都放在 template 层。更具体地说，在这个例子里：

- 不要在 template 的 JavaScript 片段里遍历整个 voucher 列表，再用 JavaScript 变量去累计各年龄段人数。
- 不要用 `add` 这个 template filter 去把 voucher 数量加总起来。

这些做法本质上是在绕开 Django templates 的逻辑限制，但它们会显著拖慢页面。

更好的方式，是把这些处理从 template 中拿出去，放进 Python 代码里。坚持我们那种“templates 只负责显示已处理数据”的最简主义做法，模板会像这样：

示例 14.6：用 Templates 展示预处理后的数据

Code (html)

```
{# templates/vouchers/ages.html #}
{% extends "base.html" %}

{% block content %}
<table>
    <thead>
        <tr>
            <th>Age Bracket</th>
            <th>Number of Vouchers Issued</th>
        </tr>
    </thead>
    <tbody>
    {% for age_bracket in age_brackets %}
        <tr>
            <td>{{ age_bracket.title }}</td>
            <td>{{ age_bracket.count }}</td>
        </tr>
    {% endfor %}
    </tbody>
</table>
{% endblock content %}
```

在这个例子里，我们可以把处理放进 model manager，使用 Django ORM 的聚合方法，以及第 37 章《附录 A：本书提到的 Packages》中介绍过的好用 `dateutil` 库：

示例 14.7：在 Template 展示前预处理数据

Code (python)

```
# vouchers/managers.py
from django.db import models
from django.utils import timezone

from dateutil.relativedelta import relativedelta


class VoucherManager(models.Manager):
    def age_breakdown(self):
        """Returns a dict of age brackets/counts."""
        age_brackets = []
        delta = timezone.now() - relativedelta(years=18)
        count = self.model.objects.filter(birth_date__gt=delta).count()
        age_brackets.append({'title': '0-17', 'count': count})
        count = self.model.objects.filter(birth_date__lte=delta).count()
        age_brackets.append({'title': '18+', 'count': count})
        return age_brackets
```

这个方法会在 view 中被调用，然后把结果作为 context 变量传给 template。

14.3.2 坑 2：在 Templates 中用条件判断做过滤

假设我们想列出所有申请过免费一品脱代金券的 Greenfeld 和 Roy 家族成员，好邀请他们参加家族聚会。我们要按 `name` 字段来过滤记录。一个非常糟糕的实现方式，就是在 template 层写巨大的循环和 `if` 判断。

示例 14.8：灾难级的数据过滤方式

Code (html)

```
<h2>Greenfelds Who Want Ice Cream</h2>
<ul>
{% for voucher in voucher_list %}

    {# Don't do this: conditional filtering in templates #}
    {% if 'greenfeld' in voucher.name.lower %}
        <li>{{ voucher.name }}</li>
    {% endif %}
{% endfor %}
</ul>

<h2>Roys Who Want Ice Cream</h2>
<ul>
{% for voucher in voucher_list %}

    {# Don't do this: conditional filtering in templates #}
    {% if 'roy' in voucher.name.lower %}
        <li>{{ voucher.name }}</li>
    {% endif %}
```

```
{% endfor %}
</ul>
```

在这段糟糕的代码里，我们一边循环，一边检查各种 `if` 条件。这等于在 template 里过滤一个可能巨大无比的记录列表，而 template 根本不是为这种工作设计的，性能瓶颈自然也就跟着来了。相反，PostgreSQL、MySQL 这类数据库非常擅长过滤记录，所以这件事应该在数据库层完成。Django ORM 也能像下面这样帮上忙。

示例 14.9：使用 ORM / 数据库过滤数据

Code (python)

```python
# vouchers/views.py
from django.views.generic import TemplateView

from .models import Voucher


class GreenfeldRoyView(TemplateView):
    template_name = 'vouchers/views_conditional.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['greenfelds'] = Voucher.objects.filter(
            name__icontains='greenfeld'
        )
        context['roys'] = Voucher.objects.filter(name__icontains='roy')
        return context
```

然后我们只需要用下面这个更简单的 template 去显示结果：

示例 14.10：更简洁、更快的 Template 展示

Code (html)

```
<h2>Greenfelds Who Want Ice Cream</h2>
<ul>
{% for voucher in greenfelds %}
    <li>{{ voucher.name }}</li>
{% endfor %}
</ul>

<h2>Roys Who Want Ice Cream</h2>
<ul>
{% for voucher in roys %}
    <li>{{ voucher.name }}</li>
{% endfor %}
</ul>
```

只要把过滤逻辑挪到 view，这个 template 就很容易提速。改完后，我们又回到了自己偏爱的那种最小职责模板写法：template 只负责显示已经过滤好的数据。

14.3.3 坑 3：Templates 中隐含的复杂查询

尽管 Django templates 在逻辑表达上有限制，但我们还是很容易在 view 中不小心制造出反复执行的无谓查询。比如，如果我们用下面这种方式列出站点用户及其喜欢的口味：

示例 14.11：会额外产生查询的 Template 代码

Code (html)

```
{# list generated via User.objects.all() #}
<h1>Ice Cream Fans and their favorite flavors.</h1>
<ul>
{% for user in user_list %}

    <li>
        {{ user.name }}:
        {# DON'T DO THIS: Generated implicit query per user #}
        {{ user.flavor.title }}
        {# DON'T DO THIS: Second implicit query per user!!! #}
        {{ user.flavor.scoops_remaining }}
    </li>
{% endfor %}
</ul>
```

那么每次访问一个 user，都会引发额外查询。单看一次似乎没什么，但我们敢肯定，只要用户够多，而且这种错误犯得够频繁，站点很快就会吃不消。

一个快速修正办法，是使用 Django ORM 的 `select_related()` 方法：

示例 14.12：用 `select_related` 查询数据

Code (html)

```
{% comment %}
List generated via User.objects.all().select_related('flavor')
{% endcomment %}
<h1>Ice Cream Fans and their favorite flavors.</h1>
<ul>
{% for user in user_list %}

    <li>
        {{ user.name }}:
        {{ user.flavor.title }}
        {{ user.flavor.scoops_remaining }}
    </li>
{% endfor %}
</ul>
```

还有一点：如果你已经在拥抱 model methods，这条原则同样适用。凡是从 template 中调用的 model methods，只要里面塞了太多查询逻辑，就要格外小心。

14.3.4 坑 4：Templates 中隐藏的 CPU 负载

要小心 template 里那些看起来很无害、实际却会带来重 CPU 处理的调用。一个 template 可能表面上很简单、代码量也很少，但其中某一行完全可能在调用一个要做大量处理的对象方法。

常见例子，是会处理图片的 template tags，比如 `sorl-thumbnail` 这类库提供的模板标签。在很多场景里，这些工具表现很好，但我们也确实踩过坑。尤其是，当图片处理和图片数据写入文件系统的动作发生在 template 内部时，那里就会形成一个阻塞点，而这个文件系统还常常跨网络。

这就是为什么：一旦项目里有大量图片处理或数据处理需求，把这些工作从 templates 挪到 views、models、helper methods，或者 Celery、Django Channels 这样的异步消息队列里，站点性能通常就会明显提升。

图 14.3: 图 14.3：泡泡糖冰淇淋看起来很好入口，但处理起来却很费劲。

14.3.5 坑 5：Templates 中隐藏的 REST API 调用

在上一节的坑里你已经看到，只要访问对象方法，就很容易把 template 加载拖慢。这个问题不仅会出现在高负载方法上，也会出现在那些内部发起 REST API 调用的方法上。一个很典型的例子，是去请求某个不幸地很慢、但你的项目又绝对离不开的第三方地图 API。不要在 template 代码里通过调用 context 中对象的方法来做这种事。

真正去消费 REST API，应该放在哪里？我们建议放在：

- JavaScript 代码里。这样项目把内容先发出去之后，就由客户端浏览器自己去处理后续工作。用户在等数据加载时，你也更容易给他们一些反馈或转移注意力。
- view 的 Python 代码里。慢流程可以在那里通过很多方式处理，比如消息队列、额外线程、多进程，或者别的方法。

14.4 别费劲把生成出来的 HTML 弄得很好看

说得直白一点，没人会在乎你的 Django 项目最终生成出来的 HTML 漂不漂亮。真有人要看渲染后的 HTML，也多半是通过浏览器检查器去看，而检查器本身就会重新整理 HTML 缩进。因此，如果你为了生成“漂亮 HTML”而把 Django templates 写得七扭八歪，那你其实是在为了取悦自己而浪费时间，同时还把代码弄得更难读。

但我们确实见过下面这种代码。它虽然能生成排版整齐的 HTML，本身却是不可读、不可维护的 template 噩梦：

示例 14.13：为了漂亮 HTML 而把 Template 代码写得面目全非

Code (html)

```
{% comment %}Don't do this! This code bunches everything
together to generate pretty HTML.
{% endcomment %}
{% if list_type=='unordered' %}<ul>{% else %}<ol>{% endif %}{% for
syrup in syrup_list %}<li class="{{ syrup.temperature_type|roomtemp
}}"><a href="{% url 'syrup_detail' syrup.slug %}">{{ syrup.title }}
</a></li>{% endfor %}{% if list_type=='unordered' %}</ul>{% else %}
</ol>{% endif %}
```

更好的写法，是用缩进并坚持“一行一个操作”，让 template 保持可读、可维护：

示例 14.14：可理解的 Template 代码

Code (html)

```
{# Use indentation/comments to ensure code quality #}
{# start of list elements #}
{% if list_type=='unordered' %}
```

```
    <ul>
{% else %}
    <ol>
{% endif %}

{% for syrup in syrup_list %}
    <li class="{{ syrup.temperature_type|roomtemp }}">
        <a href="{% url 'syrup_detail' syrup.slug %}">
            {{ syrup.title }}
        </a>
    </li>
{% endfor %}
{# end of list elements #}

{% if list_type=='unordered' %}
    </ul>
{% else %}
    </ol>
{% endif %}
```

你担心这样会产生太多空白字符吗？不用。首先，经验丰富的开发者更看重代码可读性，而不是为了所谓优化去故意把代码写乱。其次，压缩与最小化工具能帮你的忙，远比你在这里手工抠格式有用。更多细节请看第 26 章《发现并减少瓶颈》。

14.5 探索 Template 继承

我们先从一个简单的 base.html 文件开始，后面会让其他 template 来继承它：

示例 14.15：一个基础 HTML 文件

Code (html)

```
{# simple base.html #}
{% load staticfiles %}
<html>
<head>
    <title>
        {% block title %}Two Scoops of Django{% endblock title %}
    </title>
    {% block stylesheets %}
    <link rel="stylesheet" type="text/css"
          href="{% static 'css/project.css' %}">
    {% endblock stylesheets %}
</head>
<body>
    <div class="content">
        {% block content %}
        <h1>Two Scoops</h1>
        {% endblock content %}
    </div>
```

```
</body>
</html>
```

这个 base.html 文件包含以下特性：

- 一个 title block，内容是“Two Scoops of Django”。
- 一个 stylesheets block，里面链接到整个站点共用的 project.css 文件。
- 一个 content block，里面有 <h1>Two Scoops</h1>。

上面这个例子只依赖三个 template tags，概括如下：

Template Tag	用途
{% load %}	加载 staticfiles 这个内建 template tag 库。
{% block %}	因为 base.html 是父模板，所以这些 tags 用来定义哪些子 block 可以被填充。我们会把链接和脚本放在这些 block 里，以便在必要时覆盖。
{% static %}	把具名的静态资源参数解析到静态资源服务器路径。

表 14.1：base.html 中的 Template Tags

为了演示 base.html 的用法，我们让一个简单的 about.html 从它那里继承以下内容：

- 一个自定义标题。
- 原有 stylesheet 再加一个新的 stylesheet。
- 原有标题、一段子标题和一段正文内容。
- 子 blocks 的使用方式。
- {{ block.super }} 这个 template 变量的用法。

示例 14.16：从 base.html 继承

Code (html)

```
{% extends "base.html" %}
{% load staticfiles %}
{% block title %}About Audrey and Daniel{% endblock title %}
{% block stylesheets %}
    {{ block.super }}
    <link rel="stylesheet" type="text/css"
          href="{% static 'css/about.css' %}">
{% endblock stylesheets %}
{% block content %}
    {{ block.super }}
    <h2>About Audrey and Daniel</h2>
    <p>They enjoy eating ice cream</p>
{% endblock content %}
```

当我们在 view 中渲染这个 template 时，会得到下面的 HTML：

示例 14.17：渲染后的 HTML

Code (html)

```
<html>
<head>
    <title>
        About Audrey and Daniel
    </title>
```

```
    <link rel="stylesheet" type="text/css"
          href="/static/css/project.css">
    <link rel="stylesheet" type="text/css"
          href="/static/css/about.css">
</head>
<body>
    <div class="content">
        <h1>Two Scoops</h1>
        <h2>About Audrey and Daniel</h2>
        <p>They enjoy eating ice cream</p>
    </div>
</body>
</html>
```

注意到了吗？渲染出来的 HTML 已经带上了我们自定义的标题、额外的 stylesheet 链接，以及更多的正文内容。下面这张表可以帮助我们回顾 about.html 中出现的 template tags 和变量。

请注意，{% block %} tag 在 about.html 里的用法和在 base.html 里并不一样：在这里它承担的是“覆盖内容”的职责。而对于那些我们希望保留 base.html 原有内容的 blocks，我们会用 {{ block.super }} 变量把父 block 的内容也一起显示出来。这就引出了下一个主题：{{ block.super }}。

Template Object	用途
{% extends %}	告诉 Django：about.html 是从 base.html 继承或扩展而来的。
{% block %}	因为 about.html 是子模板，所以 block 会覆盖 base.html 提供的内容。这意味着我们的标题最终会渲染成 <title>Audrey and Daniel</title>。
{{ block.super }}	放在子模板的 block 里时，它可以确保父模板中的内容也被包含进当前 block。在 about.html 的 content block 中，这会渲染出 Two Scoops 。

表 14.2：about.html 中的 Template Objects

14.6 block.super 赋予你控制力

想象一下：我们有个 template，它继承 base.html 的所有内容，但要把项目里链接到 project.css 的地方换成链接 dashboard.css。这种需求很常见，比如一个项目面向普通用户是一套设计，而给员工用的 dashboard 则是另一套设计。

如果不使用 {{ block.super }}，这通常意味着你要再写一个全新的基础文件，名字往往像 base_dashboard.html。不管你喜不喜欢，结果都是：你现在得维护两套 template 架构了。

如果用了 {{ block.super }}，就不需要第二个、第三个、第四个 base template 了。只要大家都从 base.html 继承，我们就可以用 {{ block.super }} 来掌控模板行为。下面有三个例子：

同时使用 project.css 和自定义链接的 template：

示例 14.18：使用基础 CSS 和自定义 CSS 链接

Code (html)

```
{% extends "base.html" %}
{% block stylesheets %}
    {{ block.super }} {# this brings in project.css #}
```

```
    <link rel="stylesheet" type="text/css"
          href="{% static 'css/custom.css' %}" />
{% endblock stylesheets %}
```

不包含 `project.css` 链接的 dashboard template：

示例 14.19：排除基础 CSS

Code (html)

```
{% extends "base.html" %}
{% block stylesheets %}
    <link rel="stylesheet" type="text/css"
          href="{% static 'css/dashboard.css' %}" />

    {% comment %}
    By not using {{ block.super }}, this block overrides the
    stylesheet block of base.html
    {% endcomment %}
{% endblock stylesheets %}
```

只链接 `project.css` 文件的 template：

示例 14.20：使用基础 CSS 文件

Code (html)

```
{% extends "base.html" %}
{% comment %}
By not using {% block stylesheets %}, this template inherits the
stylesheets block from the base.html parent, in this case the
default project.css link.
{% endcomment %}
```

这三个例子展示了 `{{ block.super }}` 所带来的控制力度。这个变量确实是降低 template 复杂度的好工具，但要真正吃透它，可能还是需要花一点时间。

提示：block.super 和 super() 相似，但并不相同

如果你有面向对象编程背景，可以把 `{{ block.super }}` 的行为想象成 Python 内建函数 `super()` 的一个极度受限版本。它们在本质上都提供了访问“父级”的能力。

但要记住，它们并不是同一个东西。这只是一个帮助记忆的小比喻，有些开发者会觉得挺有用。

14.7 一些值得考虑的小事

下面是一组较小但我们在 template 开发时始终会记着的点。

14.7.1 不要把样式和 Python 代码绑得太死

应当尽量通过 CSS 和 JS，完全控制所有渲染出来的 templates 的样式。

能用 CSS 处理样式，就尽量用 CSS。像菜单栏宽度、颜色选择这种东西，绝不要硬编码进 Python 代码里。甚至连放进 Django templates 里，我们都建议尽量避免。

这里有几个提示：

- 如果你的 Python 代码里出现了一些完全只和视觉布局相关的 magic constants，那它们大概率应该被挪进 CSS 文件。
- JavaScript 也是同样的道理。

14.7.2 常见约定

下面是一些我们推荐的命名和风格约定：

- 在 template 名、block 名以及 template 中的其他命名里，我们偏爱下划线而不是连字符。大多数 Django 用户似乎也都是这么做的。为什么？因为 Python 对象名允许下划线，但不允许连字符。
- 我们依赖清楚、直观的 block 名。像 `{% block javascript %}` 就很好。
- 我们会在 `endblock` 里写出 block tag 的名字。不要只写 `{% endblock %}`，而是把完整的 `{% endblock javascript %}` 写出来。
- 被其他 templates 调用的模板，会用 _ 作前缀。这适用于通过 `{% include %}` 或自定义 template tags 调用的模板；但不适用于 `{% extends %}`、`{% block %}` 这类模板继承控制结构。

14.7.3 正确使用隐式和具名显式 Context 对象

当你使用通用展示类的 CBVs 时，可以在 template 里使用通用的 `{{ object_list }}` 和 `{{ object }}`。另一种选择，是使用按 model 命名的那些对象名。

例如，如果你有一个 `Topping` model，那么在 template 中就可以使用 `{{ topping_list }}` 和 `{{ topping }}`，而不是 `{{ object_list }}` 与 `{{ object }}`。这意味着，下面两个 template 示例都能工作：

示例 14.21：隐式与显式 Context 对象

Code (html)

```
{# templates/toppings/topping_list.html #}
{# Using implicit names, good for code reuse #}

<ol>
{% for object in object_list %}
    <li>{{ object }}</li>
{% endfor %}
</ol>

{# Using explicit names, good for object specific code #}
<ol>
{% for topping in topping_list %}
    <li>{{ topping }}</li>
{% endfor %}
</ol>
```

14.7.4 使用 URL 名称，而不是硬编码路径

开发者常犯的一个错误，是在 template 里这样硬编码 URL：

示例 14.22：Template 中硬编码 URL。悲伤！

Code (html)

```
<a href="/flavors/">
```

问题在于：只要站点的 URL patterns 有变，整个站点上的 URL 都得跟着逐一修改。这会影响 HTML、JavaScript，甚至 RESTful APIs。

相反，我们会使用 `{% url %}` tag，并引用 URLConf 文件里定义的名称：

示例 14.23：使用 `url` Tag

Code (html)

```
<a href="{% url 'flavors:list' %}">
```

14.7.5 调试复杂 Templates

Lennart Regebro 推荐过一个技巧：当 templates 足够复杂，以至于很难判断某个变量到底在哪里失效时，你可以在 TEMPLATES 设置的 OPTIONS 中使用 string_if_invalid 选项，从而强制输出更详细的错误信息：

示例 14.24：使用 string_if_invalid 选项

Code (python)

```
# settings/local.py
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'APP_DIRS': True,
        'OPTIONS': {
            'string_if_invalid': 'INVALID EXPRESSION: %s'
        }
    },
]
```

14.8 错误页面模板

哪怕是经过充分测试和分析的网站，偶尔也还是会出点问题，这很正常。真正的问题，在于你如何处理这些错误。你最不想看到的，就是把一张丑陋的响应页，甚至一张空白的 Web 服务器页面，直接甩回给终端用户。

至少创建 404.html 和 500.html 模板，已经算是行业标准做法了。至于你是否还要准备其他类型的错误页，可以去看看本节末尾提到的 GitHub HTML Styleguide。

我们建议通过静态文件服务器（例如 Nginx 或 Apache）来提供这些错误页，并且把它们做成完全自包含的静态 HTML 文件。这样一来，就算整个 Django 站点都挂了，只要静态文件服务器还活着，错误页也照样能发出去。

如果你部署在 PaaS 上，也要去查对应平台关于错误页的文档。比如 Heroku 就允许用户上传自定义静态 HTML 页面，用来显示 500 错误。

警告：克制住把错误页面做复杂的冲动

有趣或者好玩的错误页，的确可能成为站点的一种吸引点，但别走火入魔。要是你的 404 页面布局坏了，或者 500 页面连 CSS 和 JavaScript 都加载不出来，那就很尴尬。更糟的是，那种依赖动态渲染的 500 错误页，一旦数据库故障，自己也会跟着一起炸掉。

GitHub 的 500 错误页，就是一个“花哨但完全静态、完全自包含”的好例子。去看看 github.com/500 的页面源码，你会发现：

- 所有 CSS 样式都以内联方式写在同一个 HTML 页面的 head 里，不需要单独的 stylesheet。
- 所有图片都直接以内嵌数据形式包含在 HTML 页面中，没有任何指向外部 URL 的 <img> 链接。
- 页面需要的全部 JavaScript，也都包含在 HTML 页面内部，没有外链的 JavaScript 资源。

更多信息可以参考 GitHub HTML Styleguide：

- github.com/styleguide

14.9 坚持最简主义做法

我们建议你在 template 代码上坚持最简主义路线。把 Django templates 所谓的“限制”当成一种伪装成坏事的祝福。让这些限制反过来启发你，用更简单、更优雅的方式，把更多业务逻辑放进 Python 代码，而不是塞进 templates。

对 templates 采取最简主义策略，还有一个额外好处：当 Django apps 需要适配不断变化的格式类型时，会轻松得多。如果你的 templates 又臃肿、又充满多层循环、复杂条件和数据处理，那么业务逻辑代码就很难在不同模板间复用；更别提那些根本不使用 template 的 views，比如 API views，就更不可能复用同一套业务逻辑了。随着 API 开发日渐普及，这种代码复用能力会越来越重要，因为 API 和网页往往需要暴露完全相同的数据，只是格式不同而已。

直到今天，HTML 依然是内容表达的标准形式之一，而本章讲到的实践与模式，也正是建立在这一点之上的。

14.10 总结

本章我们讨论了下面这些内容：

- Template 继承，包括 `{{ block.super }}` 的使用。
- 如何编写清晰、可维护的 templates。
- 优化 template 性能的简单方法。
- Template 处理能力受限时会带来的问题。
- 错误页面模板。
- 以及许多其他关于 templates 的实用细节。

下一章我们会来看 template tags 和 filters。

第十五章 Template Tags 和 Filters

Django 提供了几十个默认的 template tags 和 filters，而它们都有下面这些共同特征：

- 默认项的名字都清楚、直观。
- 默认项都只做一件事。
- 默认项都不会修改任何持久化数据。

当你不得不自己写 template tags 时，这几条几乎就是一套非常好的最佳实践。下面我们就更深入一点，看看在编写自定义 filters 和 template tags 时，有哪些实践与建议值得遵守。

15.1 Filters 本质上就是函数

Filters 是函数。它们只接受一个或两个参数，而且不会给开发者在 Django templates 中加入行为控制能力。

我们觉得，正是这种简洁性，让 filters 不太容易被滥用。因为它们本质上就是带了装饰器的函数，好让 Python 能在 Django templates 里使用。这也意味着，它们本来就可以像普通函数一样被调用。不过我们的偏好，还是让 filters 去调用从 utility modules 导入的函数。

事实上，快速扫一眼 Django 默认 filters 的源码 `/git.io/vyzxO` 就会发现，`slugify()` 这个 template filter 实际上只是调用了 `django.utils.text.slugify` 函数。

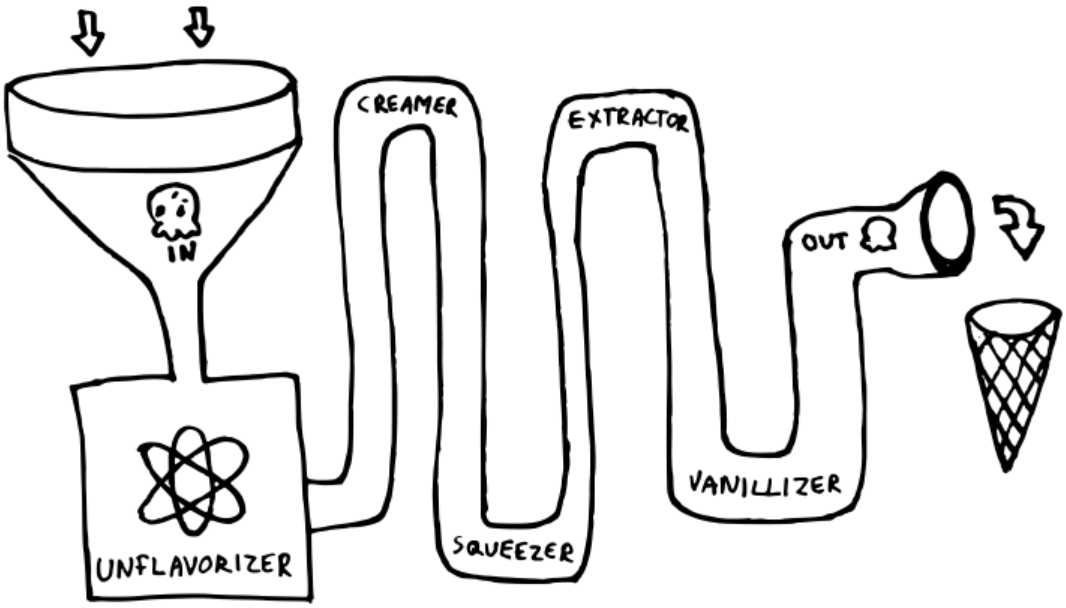

图 15.1: 图 15.1：这个 filter 会把 1 到 2 种冰淇淋口味变成香草味，然后输出到甜筒里。

15.1.1 Filters 很容易测试

测试一个 filter，本质上就是测试一个函数。而关于函数测试，我们会在第 24 章《Testing Stinks and Is a Waste of Money!》里详细讨论。

15.1.2 Filters 与代码复用

从 Django 源码里的 `defaultfilter.py` `github.com/django/django/blob/stable/3.0.x/django/template/defaultf` 可以看出，大部分 filter 逻辑其实都是从其他库导入的。比如，并没有必要去导入 `django.template.defaultfilters.slugify`。

我们完全可以直接使用 `django.utils.text.slugify`。

看起来，直接导入 filter 本身似乎也没什么问题，但它会额外增加一层代码抽象，让问题排查变得稍微更麻烦一点。

既然 filters 本来就是函数，我们主张：除了最简单的逻辑之外，其余逻辑都应该挪到更可复用的 utility functions 里，也许就放在 `utils.py` 这样的模块中。这样做更方便理解代码库、编写测试，还往往能显著减少 imports。随着时间推移，Django core 自己也越来越多地采用这种模式。

15.1.3 什么时候该写 Filters

Filters 很适合用来修改数据的呈现方式，而且它们通常也能直接复用到 REST APIs 和其他输出格式中。由于被限制在两个参数以内，它们的功能边界天然会收窄，所以更难被写成难以忍受的复杂怪物。当然，这也不是说绝对不可能。

15.2 自定义 Template Tags

“别再写那么多 template tags 了。它们调试起来太痛苦了。”

作者 Audrey Feldroy 在调试 Daniel Feldroy 写的代码时如是说。

Template tags 确实是很棒的工具，前提是开发者有足够的自制力，不让它们失控。但在实际项目里，它们很容易被滥用。本节要讨论的，就是当你把太多逻辑塞进 template tags 和 filters 之后，会遇到哪些问题。

15.2.1 Template Tags 更难调试

只要稍微复杂一点，template tags 调试起来就会很有挑战。要是它们还带有成对的开始 / 结束元素，处理起来就更难了。我们的经验是：一旦它们变得难以检查和修正，大量使用日志语句和测试会非常有帮助。

15.2.2 Template Tags 会让代码复用更难

如果你还需要在 REST APIs、RSS feeds，或者 PDF / CSV 生成中，对不同输出格式稳定地应用和 template tag 相同的效果，这件事往往会变得很困难。假如你确实需要生成这些替代格式，那就值得考虑把 template tag 的全部逻辑都放进 `utils.py`，这样别的 views 也能方便复用。

15.2.3 Template Tags 的性能代价

Template tags 可能带来显著的性能成本，尤其是在它们还会加载其他 templates 的时候。虽然 Django templates 在 1.8 之后比早期版本快了很多，但如果你对 Django 如何加载 templates 缺乏足够深入的理解，这些性能优势依然很容易被你亲手耗掉。

如果你的自定义 template tags 会加载大量 templates，那么你可能需要考虑缓存这些已加载的模板。更多细节可参考：`docs.djangoproject.com/en/3.2/ref/templates/api/#django.template.loaders.cached.Loader`

15.2.4 什么时候该写 Template Tags

如今，我们对新增 template tags 这件事非常谨慎。写之前，我们会先考虑两件事：

- 任何会触发数据读写的事情，也许更适合放进 model 或对象方法中。
- 因为我们会在项目中统一实现命名标准，所以可以在 `core.models` 模块里加一个抽象基类 model。项目里的抽象基类 model 上的某个方法或属性，能不能完成和自定义 template tag 一样的工作？

那究竟什么时候应该写新的 template tags 呢？我们的建议是：当它们只负责渲染 HTML 的时候，可以考虑写。比如某些项目的 HTML 布局极其复杂，涉及很多不同的 models 或数据类型，这时可以借助 template tags 去构建一种更灵活、更容易理解的 template 架构。

包提示：我们确实会使用自定义 Template Tags

听起来好像我们总是在躲着自定义 template tags 走，但其实并不是。我们只是很谨慎。挺有意思的是，Daniel 至少参与过三个大量使用 template tags 的知名库：

- `django-crispy-forms`
- `django-uni-form`（已废弃，请改用 `django-crispy-forms`）
- `django-wysiwyg`（已废弃）

15.3 给你的 Template Tag Libraries 命名

我们遵循的约定是 `<app_name>_tags.py`。如果用 twoscoops 这个例子，那文件名会像这样：

- `flavors_tags.py`
- `blog_tags.py`
- `events_tags.py`
- `tickets_tags.py`

这样一来，某个 template tag library 来自哪里，就会变得一目了然。

提示：不要把 Template Tag Library 和 App 取成同名

例如，把 `events` app 的 templatetag library 命名为 `events.py` 就会有问题。过去，这种做法会因为 Django 加载 template tags 的方式而引发各种麻烦。虽然这些问题后来已经修掉了，但给 template tag libraries 加上 `_tags.py` 后缀的约定还是保留了下来。这样每个人都能很容易地找到 template tag libraries。

警告：不要拿 IDE 的能力当作把代码写糊的借口

不要依赖文本编辑器或 IDE 的自动洞察能力，来判断你的 template tag library 叫什么名字。

15.4 加载你的 Template Tag Modules

在 template 中，紧跟在 `{% extends "base.html" %}`（或者任何其他父模板，而不一定非得是 `base.html`）之后，就是加载 template tags 的位置：

示例 15.1：加载 Template Tag Library 的正确方式

Code (html)

```
{% extends "base.html" %}

{% load flavors_tags %}
```

就这么简单。显式加载功能！太好了！

15.4.1 小心这个反模式

很不幸，有一种反模式，每次遇到都会让你抓狂：

示例 15.2：隐式加载 Template Tag Libraries

Code (python)

```
# settings/base.py
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
```

```
        'OPTIONS': {
            # Don't do this!
            # It's an evil anti-pattern!
            'builtins': ['flavors.templatetags.flavors_tags'],
        },
    },
]
```

这种反模式，用隐式行为取代了上面提到的显式 `load` 方式，表面上像是在修复某种“Don’t Repeat Yourself”（DRY）问题。但它带来的所谓 DRY 改进，很快就会被下面这些问题抵消掉：

- 它会增加一些额外开销，因为这等于把这个 template tag library 真的加载进了每一个由 `django.template.Template` 载入的模板中。这意味着所有继承模板、template `{% include %}`、`inclusion_tag` 等等，都会受到影响。我们虽然反对过早优化，但也并不赞成为了这种事情，给代码平白加入这么多没有必要的额外计算工作，尤其是在有更好替代方案的情况下。
- 因为 template tag library 是被隐式加载的，所以代码的可理解性和调试难度都会大幅恶化。正如《The Zen of Python》所说：“Explicit is better than Implicit.”

好在，这种写法并不常见。因为刚入门的 Django 开发者一般还不知道能犯这种错，而有经验的 Django 开发者一旦碰到它，也都会烦得不行。

15.5 总结

我们的核心观点是：template tags 和 filters 应该只关心“如何处理可呈现的数据”。只要在编写和使用它们时始终记住这一点，项目运行速度会更快，维护起来也会轻松得多。

下一章我们会继续讨论 Django templates 与 Jinja2 的使用。

第十六章 Django Templates 与 Jinja2

从 Django 1.8 开始，Django 支持多个 template engines。到目前为止，Django 模板系统内建可用的后端只有两个：Django template language（DTL）和 Jinja2。

16.1 语法上有什么区别?

在语法层面，DTL 和 Jinja2 非常相似。事实上，Jinja2 本来就是受 DTL 启发而来的。它们最显著的语法差异如下：

主题	DTL	Jinja2
方法调用	`{{ user.get_favorites }}`	`{{ user.get_favorites() }}`
filter 参数	`{{ toppings join:', ' }}`	`{{ toppings join(', ') }}`
循环空分支	`{% empty %}`	`{% else %}`
循环变量	`{{ forloop }}`	`{{ loop }}`
cycle	`{% cycle 'odd' 'even' %}`	`{{ loop.cycle('odd', 'even') }}`

表 16.1：DTL 与 Jinja2 的语法差异

16.2 我应该切换吗?

首先，用 Django 时，我们其实不必在 DTL 和 Jinja2 之间二选一。我们完全可以把 `settings.TEMPLATES` 配置成：某些 template 目录用 DTL，另一些用 Jinja2。如果你的代码库里已经有很多 templates，就可以保留现有模板，同时在确实需要的地方利用 Jinja2 的优势。这样就能两头都占到：既能接上庞大的 Django 第三方包生态，也能享受到 DTL 替代方案带来的能力。

简而言之，我们可以让多种模板语言在一个项目里和平共处。

例如，站点的大部分内容都可以继续用 DTL 渲染，而那些体量更大、渲染更重的页面则改由 Jinja2 处理。一个很好的例子是 `djangopackages.org` 的 grids。因为这些页面规模大、复杂度高，未来它们也许会被重构为由 Jinja2 驱动，而不是 DTL。当然，前提是项目不会直接演变成单页应用（SPA）。

16.2.1 DTL 的优势

下面是使用 Django template language 的一些理由：

- 它开箱即用，所有功能在 Django 文档里都写得很清楚。Django 官方关于 DTL 的文档非常完整，也很容易跟着走。官方文档中的模板代码示例也都使用 DTL。
- 大多数第三方 Django packages 使用的都是 DTL。把它们改成 Jinja2 会增加额外工作量。
- 把一个大型代码库从 DTL 转到 Jinja2，是一件工作量很大的事。

16.2.2 Jinja2 的优势

下面是使用 Jinja2 的一些理由：

- 它可以独立于 Django 使用。

- Jinja2 的语法更接近 Python，所以很多人会觉得它更直观。
- Jinja2 通常更显式，例如模板中的函数调用会带括号。
- Jinja2 对逻辑施加的人为限制更少。比如在 Jinja2 里，你可以给一个 filter 传无限个参数；而 DTL 只能传 1 个参数。
- 根据网上的 benchmarks 以及我们自己的实验，Jinja2 更快。不过话说回来，templates 往往远不是性能瓶颈中最大的一块，数据库优化通常重要得多。详见第 26 章《发现并减少瓶颈》。

16.2.3 到底谁赢？

这取决于你的具体情况：

- 新手用户应该始终坚持使用 DTL。
- 已有大型代码库的项目，基本应继续使用 DTL；只有那些确实需要性能提升的少数页面，才考虑换成 Jinja2。
- 有经验的 Djangonauts 应该两种都试一试，权衡 DTL 和 Jinja2 的优缺点，再自己做决定。

提示：选择一种主要模板语言

虽然我们可以在一个项目里混用多种模板语言，但这很容易显著增加项目的认知负担。为了降低这种风险，最好选定一种单一的主要模板语言。

16.3 在 Django 中使用 Jinja2 时需要考虑的事

下面这些点，是在 Django 项目中使用 Jinja2 templates 时要记住的。

16.3.1 CSRF 与 Jinja2

Jinja2 访问 Django 的 CSRF 机制的方式和 DTL 不一样。要在 Jinja2 templates 中接入 CSRF，在渲染 forms 时务必包含必要的 HTML：

示例 16.1：在 Jinja2 Templates 中使用 Django 的 CSRF Token

Code (html)

```
<div style="display:none">
    <input type="hidden" name="csrfmiddlewaretoken" value="{{ csrf_token }}">
</div>
```

16.3.2 在 Jinja2 Templates 中使用 Template Tags

截至目前，Django 风格的 Template Tags 还不能直接在 Jinja2 中使用。如果我们确实需要某个 template tag 的功能，那么根据想做的事情不同，可以用下面这些方式转换：

- 把这部分功能改写成函数。
- 创建一个 Jinja2 Extension。参见：

  `jinja.pocoo.org/docs/dev/extensions/#module-jinja2.ext`

16.3.3 在 Jinja2 Templates 中使用 Django 风格的 Template Filters

我们已经习惯了 DTL 里随手就能用到 Django 默认 template filters。好在，Django filters 本来就是函数（见 15.1 节《Filters 本质上就是函数》），所以我们完全可以定义一个自定义 Jinja2 environment，把这些 template filters 一起带进去：

示例 16.2：把 Django Filters 注入 Jinja2 Templates

Code (python)

```python
# core/jinja2.py
from django.contrib.staticfiles.storage import staticfiles_storage
from django.template import defaultfilters
from django.urls import reverse

from jinja2 import Environment


def environment(**options):
    env = Environment(**options)
    env.globals.update({
        'static': staticfiles_storage.url,
        'url': reverse,
        'dj': defaultfilters,
    })
    return env
```

下面是一个在 Jinja2 template 中把 Django template filters 当函数使用的例子：

示例 16.3：在 Jinja2 Template 中使用 Django Filters

Code (html)

```html
<table><tbody>
{% for purchase in purchase_list %}
    <tr>
        <a href="{{ url('purchase:detail', pk=purchase.pk) }}">
            {{ purchase.title }}
        </a>
    </tr>
    <tr>{{ dj.date(purchase.created, 'SHORT_DATE_FORMAT') }}</tr>
    <tr>{{ dj.floatformat(purchase.amount, 2) }}</tr>
{% endfor %}
</tbody></table>
```

如果你不想采用这么“全局”的做法，也可以使用 10.4.3 节《对无效 Forms 的 Views 执行自定义操作》中讨论过的一种技巧。这里我们创建一个 mixin，把 Django template filters 挂到 views 的一个属性上：

示例 16.4：供 Jinja2 使用的 Django Filter View Mixin

Code (python)

```python
# core/mixins.py
from django.template import defaultfilters


class DjFilterMixin:
    dj = defaultfilters
```

如果某个 view 继承了我们的 `core.mixins.DjFilterMixin` 类，那么在它对应的 Jinja2 template 里，我们就可以这样写：

示例 16.5：在 Jinja2 中使用由 View 注入的 Django Filters

Code (html)

```
<table><tbody>
{% for purchase in purchase_list %}
    <tr>
        <a href="{{ url('purchase:detail', pk=purchase.pk) }}">
            {{ purchase.title }}
        </a>
    </tr>
    <!-- Call the django.template.defaultfilters functions from the view -->
    <tr>{{ view.dj.date(purchase.created, 'SHORT_DATE_FORMAT') }}</tr>
    <tr>{{ view.dj.floatformat(purchase.amount, 2) }}</tr>
{% endfor %}
</tbody></table>
```

提示：在 Jinja2 中避免使用 Context Processors

Django 文档建议，不要把 context processors 用在 Jinja2 上。参见下面这个 warning box：docs.djangoproject.com/en/3.2/t

文档更推荐的做法，是向 template 传入一个函数，让模板按需调用。你可以按单个 view 来做，也可以像本小节介绍的那样，把可调用对象注入进去。

16.3.4 应把 Jinja2 的 Environment 对象视为静态对象

在示例 15.1 中，我们演示了 Jinja2 的核心组件 `jinja2.Environment` 类的用法。这个对象是 Jinja2 共享配置、filters、tests、globals 等内容的地方。当项目里的第一个 template 被加载时，Jinja2 会实例化这个类，而它在本质上几乎就是一个静态对象。

例如：

示例 16.6：Jinja2 Environment 的静态特性

Code (python)

```
# core/jinja2.py
from jinja2 import Environment

import random


def environment(**options):
    env = Environment(**options)
    env.globals.update({
        # Runs only on the first template load! The three displays below
        # will all present the same number.
        # {{ random_once }} {{ random_once }} {{ random_once }}
        'random_once': random.randint(1, 5),
        # Can be called repeated as a function in templates. Each call
        # returns a random number:
        # {{ random() }} {{ random() }} {{ random() }}
        'random': lambda: random.randint(1, 5),
    })
    return env
```

警告：不要在实例化之后修改 `jinja.Environment`

虽然技术上做得到，但修改 `jinja.Environment` 对象是危险的。Jinja2 API 文档里写得很明确："在第一个 template 被加载之后再修改 environments，会导致令人意外的效果与未定义行为。"

参考：`jinja.pocoo.org/docs/dev/api/#jinja2.Environment`

16.4 资源

- Django 关于使用 Jinja2 的文档：
  `docs.djangoproject.com/en/3.2/topics/templates/#django.template.backends.jinja2.Jinja2`
- `jinja.pocoo.org`

16.5 总结

本章我们讨论了 DTL 与 Jinja2 之间的相同点和不同点，也看了在项目中使用 Jinja2 时会带来的影响，以及一些应对办法。

从下一章开始，我们会暂时离开 templates，转而从服务端和客户端两个角度，一起进入 REST 的世界。

第十七章 使用 Django REST Framework 构建 REST APIs

如今的互联网早就不只是由 HTML 驱动的网站了。开发者需要同时支持 Web 客户端、原生移动应用，以及企业对企业的应用。能够轻松生成 JSON、YAML、XML 等格式的工具，因此就变得非常重要。按照设计，表现层状态转移（Representational State Transfer，REST）应用程序编程接口（API）会把应用数据暴露给其他使用方。

在 Django 世界里，用来构建这类 REST APIs 的事实标准包，就是 Django REST Framework（DRF）。事实上，DRF 已经普及到了这样一种程度：你经常会听到别人问，“Django 和 Django REST Framework 到底有什么区别?”我们估计，从 2013 年前后开始，大约 90%-95% 带 API 的新 Django 项目都会使用 DRF。

为什么这么流行？我们觉得，Django REST Framework 之所以成功，是因为：

- DRF 大量依赖面向对象设计，而且天生就是为了易于扩展而设计的。
- DRF 直接建立在 Django 的 CBVs 之上。如果你理解 CBVs，那么 DRF 的设计会像是 Django 的一种自然延伸，很容易理解。
- 它自带一整套用于生成 API 的 views，既有类似 `django.views.generic.View` 的 `APIView`，也有更深层的抽象，比如 generic API views 和 viewsets。
- serializer 系统非常强大，但如果你不想用，也能很轻松地绕开或替换它。
- Authentication 和 Authorization 都有强大而可扩展的支持。
- 如果你真的想在 API 中使用 FBVs，DRF 也照顾到了这一点。

也正因为这些原因，DRF 的社区非常庞大。这很重要，因为这意味着：用它构建 REST APIs 时遇到的很多问题，别人多半已经解决过了。也许不是在 DRF 核心里解决的，但往往已经有第三方包补上了。另外，想找到懂它、能回答问题的人，也并不难。

我们会在第 18 章《使用 Django 构建 GraphQL APIs》中讨论 API 这种新范式的另一面。

如果你还不会用 DRF，我们推荐先看官方教程：

- `django-rest-framework.org/tutorial/quickstart/`
- `django-rest-framework.org/tutorial/1-serialization/`

提示：Django REST Framework 需要你的支持！

DRF 是一个协作资助项目。如果你在商业环境中使用它，我们强烈建议你通过付费计划为它的持续开发提供支持。也正是因为有这些资金，我们才亲眼看到这个项目在功能层面不断向前跃进。

资助金额从 15 美元起，往上还有更高档位。在更高的资助等级下，DRF 项目甚至会提供优先支持：`fund.django-rest-fra`

17.1 基础 REST API 设计的基本原理

我们先退一步，从 HTTP 以及它如何与 Django REST Framework 协作说起。

超文本传输协议（HTTP）是一种分发内容的协议，它提供了一组用来声明动作的方法。按照惯例，REST APIs 正是依赖这些方法来工作的，因此针对不同类型的动作，就应该使用合适的 HTTP method：

请求目的	HTTP Method	粗略对应的 SQL
创建新资源	POST	INSERT
读取现有资源	GET	SELECT
更新现有资源	PUT	UPDATE
更新现有资源的一部分	PATCH	UPDATE
删除现有资源	DELETE	DELETE
返回与 GET 相同的 HTTP headers，但不返回 body 内容	HEAD	
返回给定 URL 支持的 HTTP methods	OPTIONS	
回显请求本身	TRACE	

表 17.1：HTTP Methods

关于上表，再补几句说明：

- 如果你实现的是只读 API，那么你可能只需要实现 GET。
- 如果你实现的是可读可写 API，那么应该使用 GET、POST、PUT 和 DELETE。
- 所有动作都只依赖 GET 和 POST，会让 API 使用者非常痛苦。
- 按定义，GET、PUT 和 DELETE 是幂等的；POST 和 PATCH 则不是。
- PATCH 经常没人实现，但如果你的 API 支持 PUT，那么把 PATCH 一起支持掉通常是个好主意。
- Django REST Framework 就是围绕这些 methods 设计的；理解它们，理解 DRF 也就会容易得多。

下面是实现 REST API 时常见的一些 HTTP status codes。DRF 的 generic views 和 viewsets 会根据被调用的方法，在合适的时候返回这些值。注意，这里只是一份节选；更完整的状态码列表可以参考：en.wikipedia.org/wiki/List_of_HT

17.2 用一个简单 API 来说明设计概念

为了说明 DRF 是如何把 HTTP methods、HTTP status codes、serialization 与 views 串起来的，我们来构建一个简单的 JSON API。我们继续使用前面章节里的 `flavors` app 作为基础，通过 AJAX、`python-requests` 或其他库发起 HTTP 请求，实现 flavor 的创建、读取、更新与删除。

我们先从“确保 API 默认足够安全”开始。在 settings 文件中，把默认 permission classes 设成只允许管理员访问：

示例 17.1：我们默认使用的 DRF Permission Classes

Code (python)

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAdminUser',
    ),
}
```

提示：把 `IsAdminUser` 作为默认 Permission Class

我们喜欢把项目锁得很紧，尤其是 REST APIs。最好的默认方式之一，就是把 `rest_framework.permissions.IsAdminUser` 设成默认 permission class。之后如果某个 view 需要放宽权限，我们再按 view 单独覆盖。这样一来，API views 在默认情况下会非常安全，而这点价值，绝对比多写几行代码要高得多。

说完这些，再来看一次我们的 `Flavor` model，不过这次多加一个供 API 查找记录用的 UUID：

示例 17.2：REST API 使用的 `Flavor` Model

Code (python)

```
# flavors/models.py
import uuid as uuid_lib

from django.db import models
from django.urls import reverse


class Flavor(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField(unique=True)  # Used to find the web URL
    uuid = models.UUIDField(  # Used by the API to look up the record
        db_index=True,
        default=uuid_lib.uuid4,
        editable=False,
```

```
    )
    scoops_remaining = models.IntegerField(default=0)

    def get_absolute_url(self):
        return reverse('flavors:detail', kwargs={'slug': self.slug})
```

警告：不要把连续递增的 key 当成公开标识符

像 Django 默认给 model 主键提供的那种顺序递增 key，如果拿来公开使用，会带来安全隐患。我们会在 28.28 节《永远不要显示顺序递增主键》中详细解释。

在这个例子里，我们会用 model 的 UUID，而不是 model 主键，来查找记录。我们一直都尽量避免在查找时使用这种顺序数字。

先定义 serializer 类：

示例 17.3：Flavor Model 的 Serializer

Code (python)

```
# flavors/api/serializers.py
from rest_framework import serializers

from ..models import Flavor


class FlavorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Flavor
        fields = ['title', 'slug', 'uuid', 'scoops_remaining']
```

接着加入 API views：

示例 17.4：Flavor API Views

Code (python)

```
# flavors/api/views.py
from rest_framework.generics import (
    ListCreateAPIView,
    RetrieveUpdateDestroyAPIView,
)
from rest_framework.permissions import IsAuthenticated

from ..models import Flavor
from .serializers import FlavorSerializer


class FlavorListCreateAPIView(ListCreateAPIView):
    queryset = Flavor.objects.all()
    permission_classes = (IsAuthenticated,)
    serializer_class = FlavorSerializer
    lookup_field = 'uuid'  # Don't use Flavor.id!


class FlavorRetrieveUpdateDestroyAPIView(RetrieveUpdateDestroyAPIView):
    queryset = Flavor.objects.all()
```

```
    permission_classes = (IsAuthenticated,)
    serializer_class = FlavorSerializer
    lookup_field = 'uuid'  # Don't use Flavor.id!
```

这就做完了！没错，就是这么快。

提示：Classy Django REST Framework 是很好用的参考

在使用 Django REST Framework 时，我们觉得 http://cdrf.co 是一份非常好用的速查表。它借鉴了著名的 ccbv.co.uk 参考站的风格，但内容是为 Django REST Framework 量身定制的。

现在把它们接进 flavors/urls.py：

示例 17.5：接入 API Views

Code (python)

```
# flavors/urls.py
from django.urls import path

from flavors.api import views

urlpatterns = [
    # /flavors/api/
    path(
        route='api/',
        view=views.FlavorListCreateAPIView.as_view(),
        name='flavor_rest_api',
    ),
    # /flavors/api/:uuid/
    path(
        route='api/<uuid:uuid>/',
        view=views.FlavorRetrieveUpdateDestroyAPIView.as_view(),
        name='flavor_rest_api',
    ),
]
```

我们这里做的事，是复用 URLConf name。这样当你需要做一个 heavily JavaScript-driven 的前端时，会更容易管理。你只需要通过 {% url %} template tag 去访问 Flavor 资源即可。

如果你还不太确定这段 URLConf 到底做了什么，我们可以用一张表再过一遍：

URL	View	URL Name
/flavors/api/	FlavorListCreateAPIView	flavor_rest_api
/flavors/api/:uuid/	FlavorRetrieveUpdateDestroyAPIView	flavor_rest_api

表 17.3：Flavor REST APIs 的 URLConf

警告：我们这个简单 API 没有使用更细粒度的 Permissions

我们把默认的 IsAdminUser permission 覆盖成了 IsAuthenticated。如果你照着这个例子实现 API，别忘了按照实际需要，认真分配用户权限！

- django-rest-framework.org/api-guide/authentication
- django-rest-framework.org/api-guide/permissions

最后得到的，就是一种传统的 REST 风格 API 定义：

示例 17.6：传统 REST 风格 API 定义

Code (text)

```
flavors/api/
flavors/api/:uuid/
```

提示：描述 REST APIs 时常见的语法

你经常会看到像“接入 API Views”这个示例里那样的写法。比如这里的 `/flavors/api/:uuid/` 包含一个 `:uuid` 值。它代表一个变量，但这种写法更适合跨框架、跨语言的文档描述，因此你会在很多第三方 REST API 文档里见到它。

如果你之前还不知道的话，现在你已经看到了：在 Django 中构建 REST APIs 其实非常容易。接下来我们继续说一些关于如何维护和扩展它们的建议。

17.3 REST API 架构

用 Django REST Framework 快速做出简单 API 并不难，但要把它扩展、维护到适合你项目的程度，就需要多一些思考了。人们通常就是卡在 API 设计这一层。下面是一些改进设计的建议。

17.3.1 对 API Modules 使用一致的命名

和其他任何东西一样，命名在整个项目里都应该保持一致。我们对 API 相关 modules 的命名偏好如下：

示例 17.7：我们偏好的 API 相关 Modules 命名方式

Code (text)

```
flavors/
+-- api/
|   +-- __init__.py
|   +-- authentication.py
|   +-- parsers.py
|   +-- permissions.py
|   +-- renderers.py
|   +-- serializers.py
|   +-- validators.py
|   +-- views.py
|   +-- viewsets.py
```

请注意下面几点：

- 我们喜欢把所有 API 组件都放进 app 内一个叫 `api/` 的 package 中。这样 API 组件就能在统一位置隔离出来。如果把它们都丢在 app 根目录，那么 app 的公共区域就会堆满 API 专用 modules。
- Viewsets 应该放在它们自己的 module 里。
- Routers 我们永远放在 `urls.py`。不管是在 app 级还是 project 级，routers 都属于 `urls.py`。

17.3.2 项目的代码应该组织得干净利落

如果项目里有很多彼此联动的小 app，那么想追踪某个 API view 到底定义在哪儿，可能会变得很麻烦。相较于把所有 API 代码都放在各自相关的 app 里，有时候更合理的方式，是专门建一个 app 来承载 API。所有 serializers、renderers 和 views 都放在那里。

因此，这个 app 的名字应该能够反映它的 API 版本（见 17.3.7 节《给你的 API 做版本控制》）。

例如，我们可能会把所有 views、serializers 和其他 API 组件都放进一个叫 `apiv4` 的 app 里。

这种做法的缺点，是 API app 可能会膨胀得过大，而且和真正为它提供业务能力的那些 app 脱节。因此，下一小节我们会看另一种思路。

17.3.3 App 的代码应尽量留在 App 内

归根结底，REST APIs 也只是 views。对于更简单、更小型的项目，REST API views 应该直接放进 `views.py` 或 `viewsets.py`，并遵守我们对其他 views 一样的组织准则。app 专属或 model 专属的 serializers、renderers 也是同样的道理。

如果某个 app 里的 REST API view classes 多到让单个 `api/views.py` 或 `api/viewsets.py` 难以浏览，我们就把它们拆开。具体做法是：把视图（或 viewset）类放进 `api/views/`（或 `api/viewsets/`）package，再按 model 命名 Python modules。于是你可能会看到：

Code (text)

```
flavors/
+-- api/
|   +-- __init__.py
|   +-- ... other modules here
|   +-- views/
|   |   +-- __init__.py
|   |   +-- flavor.py
|   |   +-- ingredient.py
```

这种方式的缺点在于：如果项目里有太多彼此交织的小 app，那 API 组件散落的位置也会变得很难追踪。因此，上一小节里那种专门建 API app 的方案，才会成为另一种备选。

17.3.4 尽量把业务逻辑移出 API Views

不管你采用哪种架构方式，尽量把逻辑从 API views 中拿出去，始终是个好主意。如果这听起来很耳熟，那是因为确实如此。我们在 8.5 节《尽量把业务逻辑移出 Views》中已经讲过这一点。别忘了，说到底，API views 也只是另一种 view。

17.3.5 对 API URLs 做分组

如果你的 REST API views 分散在多个 Django apps 中，那么怎样才能构建出一个项目范围内、长这样子的 API：

示例 17.9：项目级 API 设计

Code (text)

```
api/flavors/ # GET, POST
api/flavors/:uuid/ # GET, PUT, DELETE
api/users/ # GET, POST
api/users/:uuid/ # GET, PUT, DELETE
```

过去，我们会把所有 API view 代码放进一个专门的 Django app，比如 `api` 或 `apiv1`，并把某些 REST views、serializers 等里的自定义逻辑也一起放进去。理论上这是个不错的办法，但在实践中，它意味着某个具体 app 的逻辑会分散在不止一个地方。

我们现在更倾向于依赖 URL 配置。构建项目级 API 时，我们会把 REST views 写在各自 app 的 `api/views.py` 或 `api/viewsets.py` 中，然后把它们接进一个类似 `core/api_urls.py` 或 `core/apiv1_urls.py` 的 URLConf，再由项目根的 `urls.py` 去 include 它。于是代码大致会像这样：

示例 17.10：把多个 App 的 API Views 合并到一起

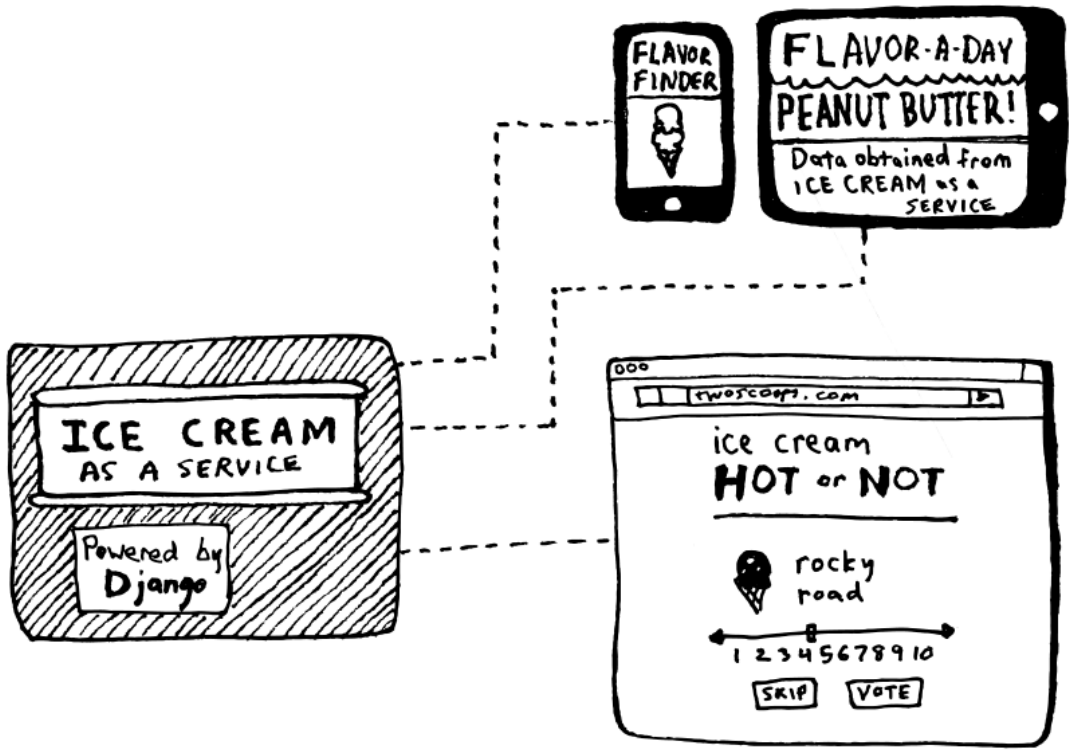

图 17.1: 图 17.1：一个“冰淇淋即服务”（Ice Cream as a Service）API。

Code (python)

```
# core/api_urls.py
"""Called from the project root's urls.py URLConf thus:

path('api/', include('core.api_urls', namespace='api')),
"""
from django.urls import path

from flavors.api import views as flavor_views
from users.api import views as user_views

urlpatterns = [
    # {% url 'api:flavors' %}
    path(
        route='flavors/',
        view=flavor_views.FlavorCreateReadView.as_view(),
        name='flavors',
    ),
    # {% url 'api:flavors' flavor.uuid %}
    path(
        route='flavors/<uuid:uuid>/',
        view=flavor_views.FlavorReadUpdateDeleteView.as_view(),
        name='flavors',
    ),
    # {% url 'api:users' %}
    path(
```

```
        route='users/',
        view=user_views.UserCreateReadView.as_view(),
        name='users',
    ),
    # {% url 'api:users' user.uuid %}
    path(
        route='users/<uuid:uuid>/',
        view=user_views.UserReadUpdateDeleteView.as_view(),
        name='users',
    ),
]
```

17.3.6 测试你的 API

我们发现，Django 的测试套件让 API 实现的测试变得非常容易。肯定比盯着 `curl` 的返回结果发呆轻松多了！关于测试，我们会在第 24 章《Testing Stinks and Is a Waste of Money!》里详细展开。在那一章里，我们甚至还放进了这个简单 JSON API 的测试代码（见 24.3.1 节《每个测试方法只测试一件事》）。

17.3.7 给你的 API 做版本控制

一个很好的实践，是在 API 的 URLs 中显式带上版本号，例如 `/api/v1/flavors` 或 `/api/v1/users`；当 API 发生变化时，就改成 `/api/v2/flavors` 或 `/api/v2/users`。我们偏爱这种方式，或者主机名版本控制风格，例如 `v1.icecreamlandia.com/api/users`。当版本号变化时，现有客户仍然可以继续使用旧版，而不会在不知情的情况下把自己的 API 调用搞崩。

另外，为了不惹怒 API 消费者，在升级期间和升级之后，同时维护现有 API 与它的前一代 API 这件事，非常关键。被废弃的 API 持续使用几个月，这一点都不罕见。

当我们实现新版 API 时，会在真正发生 breaking changes 很久之前，就提前向客户 / 用户发出 deprecation warning，好让他们有时间做升级，不至于把自己的应用弄坏。就我们的亲身经验而言，即便是免费开源 API 服务，向用户索要邮箱地址也很有价值，因为这样你才有能力向终端用户发送这类废弃通知。

Django REST Framework 内建了对我们偏好的这类版本控制方案以及其他方案的支持，文档在这里：`django-rest-framew`

17.3.8 对自定义认证方案保持谨慎

如果你正在构建 API，而且确实需要一套自定义认证方案，那一定要格外小心。安全问题很难做对，而且永远会有难以预料的边界情况，这正是站点被攻破的原因之一。我们虽然只做过几次自定义认证方案，但我们始终记着下面几点：

- 如果我们要创建新的认证方案，就让它尽量简单，而且测试要充分。
- 在代码之外，写清楚为什么现有标准认证方案不够用。见下面的提示框。
- 也在代码之外，详细记录这套认证方案预期如何工作。见下面的提示框。
- 除非我们实现的是非 cookie 的认证方案，否则不要关闭 CSRF。

提示：自定义认证的文档至关重要

把一套自定义认证方案的“为什么”和“怎么做”写出来，是整个流程中非常关键的一部分。不要跳过！原因如下：

- 它能帮助我们验证：自己提出新方案的理由到底站不站得住。如果连问题都写不清，那就说明我们其实并没有真正理解它。
- 文档会逼着我们在写代码之前先把解决方案架构清楚。

- 等系统落地之后，这些文档还能帮助我们自己，或者后来接手的人，理解当初为什么做出这些设计决定。

17.4 当 DRF 挡路的时候

Django REST Framework 是个功能强大的工具，但它也带来了很多抽象层。想硬穿过这些抽象，有时会让人极其沮丧。下面我们来看看，如何跨过去。

17.4.1 远程过程调用 vs REST APIs

REST frameworks 用来暴露数据的 resource model 非常强大，但它并不能覆盖所有情况。更具体地说，resource 并不总能完美映射真实的应用设计。

例如，把 syrup 和 sundae 表示成两个资源，这很容易；但“往圣代上倒糖浆”这个动作该怎么表示呢？用这个比喻来说，我们会改变 sundae 的状态，同时让 syrup 库存减少 1。虽然也可以让 API 用户分别去修改它们，但那样很可能会带来数据库完整性问题。因此，在某些场景下，把 `sundae.pour_syrup(syrup)` 这种方法，作为 RESTful API 的一部分提供给客户端，反而是个好主意。

用计算机科学术语来说，`sundae.pour_syrup(syrup)` 可以被归类为远程过程调用（Remote Procedure Call，RPC）。

参考：

- `en.wikipedia.org/wiki/Remote_Procedure_Call`
- `en.wikipedia.org/wiki/Resource-oriented_architecture`

好在，用 Django REST Framework 实现 RPC 调用很容易。我们要做的，只是忽略 DRF 的那些高级抽象工具，转而依赖它最基础的 `APIView`：

示例 17.11：用 DRF 实现 `pour_syrup()` RPC

Code (python)

```
# sundaes/api/views.py
from django.shortcuts import get_object_or_404

from rest_framework.response import Response
from rest_framework.views import APIView

from ..models import Sundae, Syrup
from .serializers import SundaeSerializer, SyrupSerializer


class PourSyrupOnSundaeView(APIView):
    """View dedicated to adding syrup to sundaes"""

    def post(self, request, *args, **kwargs):
        # Process pouring of syrup here,
        # Limit each type of syrup to just one pour
        # Max pours is 3 per sundae
        sundae = get_object_or_404(Sundae, uuid=request.data['uuid'])
        try:
            sundae.add_syrup(request.data['syrup'])
        except Sundae.TooManySyrups:
            msg = "Sundae already maxed out for syrups"
            return Response({'message': msg}, status_code=400)
```

```
        except Syrup.DoesNotExist:
            msg = "{} does not exist".format(request.data['syrup'])
            return Response({'message': msg}, status_code=404)
        return Response(SundaeSerializer(sundae).data)

    def get(self, request, *args, **kwargs):
        # Get list of syrups already poured onto the sundae
        sundae = get_object_or_404(Sundae, uuid=request.data['uuid'])
        syrups = [SyrupSerializer(x).data for x in sundae.syrup_set.all()]
        return Response(syrups)
```

于是我们的 API 设计现在会长成这样：

示例 17.12：Sundae 与 Syrup 的 API 设计

Code (text)

```
/sundae/ # GET, POST
/sundae/:uuid/ # PUT, DELETE
/sundae/:uuid/syrup/ # GET, POST
/syrup/ # GET, POST
/syrup/:uuid/ # PUT, DELETE
```

17.4.2 复杂数据带来的问题

好吧，我们承认，这种错误我们在 DRF 里差不多每个月都要犯一次。下面用一个很简单的 API 设计来概括这个问题：

示例 17.13：Cone 与 Scoop API

Code (text)

```
/api/cones/ # GET, POST
/api/cones/:uuid/ # PUT, DELETE
/api/scoops/ # GET, POST
/api/scoops/:uuid/ # PUT, DELETE
```

1. 我们有一个 model（Scoop），想把它嵌套表示在另一个 model（Cone）里面。
2. 写一个能返回 Cone 及其 Scoops 列表的 GET，非常容易。
3. 但如果要写一个针对 Cone 的 POST 或 PUT，并且还要同时新增或更新它的 Scoops，事情就会开始变难，尤其是只要这里面还夹杂着任何验证或后处理逻辑。
4. 然后挫败感袭来，我们离开电脑，去吃一份现实世界里的冰淇淋。

虽然也确实存在一些漂亮但复杂的嵌套数据解决方案，不过我们发现还有个更好的办法：稍微简化一下。例如：

- 保留 Cone 的 GET 表示，其中仍然包含它的 Scoops。
- 去掉 Cone 的 POST 或 PUT 直接修改其 Scoops 的能力。
- 为属于某个 Cone 的 Scoops 单独创建 GET / POST / PUT API views。

于是 API 最终会变成这样：

示例 17.14：一种更简化的 Cone 与 Scoop API

Code (text)

```
/api/cones/ # GET, POST
/api/cones/:uuid/ # PUT, DELETE
```

```
/api/cones/:uuid/scoops/ # GET, POST
/api/cones/:uuid/scoops/:uuid/ # PUT, DELETE
/api/scoops/ # GET, POST
/api/scoops/:uuid/ # PUT, DELETE
```

没错，这种方式确实会增加额外的 views，也会多出更多 API 调用。但另一方面，这种数据建模方式反而可能让你的 API 更简单。而这种简化，最终会带来更容易的测试，也就是更稳健的 API。

顺带一提，如果你仔细看看 Stripe 的 API 参考文档（`stripe.com/docs/api`），会发现他们走的也是这条路线。复杂数据你可以查看，但要创建它，就得一步一步来。

17.4.3 简化！原子化！

在前面两小节（RPC 调用与复杂数据问题）里，我们已经建立起一种“简化”的模式。本质上，当我们在 DRF 里遇到问题时，会这样问自己：

- 我们能不能把 views 简化？切换成 `APIView` 能不能解决问题？
- 我们能不能把 views 所描述的 REST 数据模型简化？多加几个性质直接、职责单一的 views，能不能解决问题？
- 如果某个 serializer 太折磨人，而且复杂得离谱，那为什么不针对同一个 model，拆成两个不同的 serializers？

正如你看到的，要解决 DRF 带来的问题，我们通常会把 API 分解成更小、更原子的组件。我们的经验是：与其只有少量 views 但每个都带着一大堆选项，不如做更多个尽量原子化的 views。任何有经验的程序员都知道：选项越多，边界情况就越多。

原子化组件在这些方面都有帮助：

- 文档更容易写，也更快，因为每个组件做的事更少。
- 测试更容易，因为代码分支更少。
- 性能瓶颈更容易定位，因为阻塞点被隔离得更清楚。
- 安全性更好，因为我们可以更轻松地按 view 调整访问控制，而不是把各种判断塞进 view 代码内部。

17.5 关闭一个对外 API

当你决定关停一个旧版对外 API，并用新版替代它时，下面这些步骤会很有帮助：

17.5.1 第一步：通知用户 API 即将关停

提前通知，越早越好。最好能提前 6 个月，最短也尽量提前 1 个月。通过邮件、博客和社交媒体告诉 API 用户。我们自己会反复强调这件事，强调到甚至担心别人已经听烦了为止。

17.5.2 第二步：用 410 Error View 替换旧 API

当 API 终于真正下线时，我们会提供一个简单的 410 Error View。里面放一段非常简短的消息，并包含以下信息：

- 指向新版 API endpoint 的链接。
- 指向新版 API 文档的链接。
- 指向一篇解释此次关停细节的文章的链接。

下面是一个适用于任意 HTTP method 的下线 view 示例：

示例 17.15：关停 API 的代码

Code (python)

```
# core/apiv1_shutdown.py
from django.http import HttpResponseGone

apiv1_gone_msg = """APIv1 was removed on April 2, 2017. Please switch to APIv2:
<ul>
    <li>
        <a href="https://www.example.com/api/v3/">APIv3 Endpoint</a>
    </li>
    <li>
        <a href="https://example.com/apiv3_docs/">APIv3 Documentation</a>
    </li>
    <li>
        <a href="http://example.com/apiv1_shutdown/">APIv1 shut down notice</a>
    </li>
</ul>
"""


def apiv1_gone(request):
    return HttpResponseGone(apiv1_gone_msg)
```

17.6 给你的 API 做限流

Rate limiting，指的是：API 限制某个用户在一段时间内最多能发出多少请求。这样做有很多原因，下面我们会逐一解释。

17.6.1 不受约束的 API 访问很危险

在远古时代的 2010 年，我们上线了 djangopackages.org。这个项目最初是在 Django Dash 比赛期间启动的，一上线就在 Django 社区里爆红。我们疯狂冲刺开发，功能集也增长得非常快。不幸的是，我们很快就撞上了 GitHub 第一代 API 的限流。也就是说，每小时请求达到一定量之后，我们在新的一小时到来之前，就再也不能继续请求了。

好在，2010 年的 DjangoCon 上，我们有机会问 GitHub 的一位创始人：能不能给我们无限制访问 API 的权限？他非常爽快地说了“可以”，然后不到一天时间，我们就可以随心所欲地从 GitHub 拿数据了。

我们很高兴。用户也很高兴。站点使用量持续上涨，大家都很想知道哪些项目最活跃。我们自己也对数据极度渴望，于是每个小时都会去请求 GitHub 最新的数据。然后，这就给 GitHub 带来了问题。

要知道，那可是 2010 年，GitHub 还远不是今天这个庞大而强悍的公司。每到整点过后 17 分钟，Django Packages 就会在极短时间里向 GitHub API 发出成千上万次请求。由于我们拥有不受限制的访问权，实际上已经在给他们添麻烦了。

最终，GitHub 联系了我们，请求我们降低 API 使用强度。无限制访问权限仍然保留，只是希望我们给他们一点喘息空间。我们照做了，把数据抓取频率从每小时一次改成每天一次，而且请求节奏也更温和。直到今天，我们仍然保持着这种做法。

当然，现代 GitHub 能承受的 API 访问量，远远大于 2010 年底的它自己。但我们还是愿意认为，当年我们一起学到了同一个教训：对 API 的不受约束访问，必须谨慎授予。

17.6.2 REST Frameworks 必须提供限流能力

能否控制 REST API 的访问量，往往就是一个项目最终走向欢欣胜利还是彻底灾难的分水岭。

提示：HTTP Server 层的限流

你也可以用 nginx 或 apache httpd 来做限流。好处是性能更高；坏处是，这会把限流功能从 Python 代码层剥离出去。对一个 Python 系统来说，DRF 自己已经提供了一些可以继续构建的基础工具。

- `nginx.com/blog/rate-limiting-nginx/`
- `httpd.apache.org/docs/2.4/mod/mod_ratelimit.html`
- `django-rest-framework.org/api-guide/throttling/#setting-the-throttling-policy`

17.6.3 限流本身也可以成为商业计划

想象一下，我们做了一个基于 API 的创业项目，允许用户把各种配料图片叠加到冰淇淋图片上。我们知道每个人都会想用这个 API，于是围绕访问额度设计出几个收费档位：

- 开发者档免费，但每小时只能发 10 次 API 请求。
- One Scoop 每月 24 美元，每分钟 25 次请求。
- Two Scoops 每月 79 美元，每分钟 50 次请求。
- Corporate 每月 5000 美元，每分钟 200 次请求。

这种方式叫作 Bucket-Based Pricing。剩下要做的，就是让更多人来用我们的 API。

提示：考虑按使用量计费，而不是按限流桶计费

Randall Degges 强烈主张：本节建议的这种 bucket-based pricing，其实并不是一个好商业模式。他提倡从一开始就提供不限流的使用方式，然后按访问量收费，而且随着使用量增加，单价递减。他给出的版本大概会像这样：

- 每月前 1000 次请求免费。
- 每月第 1001 到 100,000 次请求，每次收费 0.4 美分。
- 每月第 100,001 到 200,000 次请求，每次收费 0.3 美分。
- 每月第 200,001 次及以上，每次收费 0.2 美分。

按使用量计费的好处，是它鼓励用户提升使用量，而不是去努力卡在某个“桶”或某个档位里。坏处是：你需要在项目里做更多工作，才能把这种计费方案搭起来。像 Stripe、Paddle 这样的第三方服务，能够对订阅增加额外费用或附加金额，所以在常见支付系统里，这并不是做不到。

参考：`rdegges.com/2020/the-only-type-of-api-services-ill-use`

17.7 推广你的 REST API

假设我们已经做完 REST API，而且希望外部开发者和公司来使用它。那该怎么推广？

17.7.1 文档

最重要的事情，就是提供全面的文档。而且越容易读、越容易理解越好。提供上手即用的代码示例更是必不可少。

你既可以从零手写文档，也可以使用 `django-rest-framework` 自带的自动文档工具，或者各种第三方包（`django-rest-framework.org/topics/documenting-your-api/#third-party-packages`）。你甚至还可以直接拥抱像 `readthedocs.com`、`swagger.io` 这样的商业化文档生成服务。

第 25 章《Documentation: Be Obsessed》中的一些内容，对于面向外部的 REST API 文档也会很有帮助。

17.7.2 提供客户端 SDKs

另一件可能有助于推广 API 的事，是为不同编程语言提供软件开发工具包（SDK）。覆盖的语言越多越好。以我们的经验来看，至少应该覆盖这些语言：Python、JavaScript、Ruby、PHP、Go 和 Java。

我们的经验还告诉我们：最好至少亲手写其中一个库，并顺便做一个 demo project。原因是，这不仅能帮你宣传 API，还能逼着你站在 API 使用者的角度，亲自体验自己的 API。

幸运的是，借助 OpenAPI（`openapis.org/`）底层的 OpenAPI document object model，DRF 提供了与 JSON Hyperschema 兼容的功能。只要客户端和服务端都遵循这种格式，OpenAPI 就允许我们编写动态驱动的 client libraries，与任何暴露出受支持 schema 或 hypermedia format 的 API 交互。

这些工具对大多数由 DRF 驱动的 API 都应该能直接工作：

- 命令行客户端

`https://www.django-rest-framework.org/api-guide/schemas/#generating-an-openapi-schema`

- 多语言 Client SDK 生成器

`github.com/OpenAPITools/openapi-generator`

- 供 JavaScript 使用的 `swagger-cli`

`npmjs.com/package/swagger-cli`

如果你要构建 Python 驱动的 client SDKs，读一读 23.9 节《发布你自己的 Django Packages》也会有帮助。

17.8 延伸阅读

我们强烈推荐阅读下面这些内容：

- `en.wikipedia.org/wiki/REST`
- `en.wikipedia.org/wiki/List_of_HTTP_status_codes`
- `github.com/OAI/OpenAPI-Specification`
- `jacobian.org/writing/rest-worst-practices/`

17.9 其他 API 构建方案

基于本章开头提到的那些理由，我们推荐 Django REST Framework。不过，如果你选择不使用 DRF，也可以考虑下面这些方式：

17.9.1 CBV 路线：JsonResponse 搭配 View

你完全可以使用 Django 内建的 `django.http.JsonResponse` 类。它是 `django.http.HttpResponse` 的一个子类，可以配合 `django.views.generic.View` 使用。这样一来就能支持完整范围的 HTTP methods，但它不提供 OpenAPI 支持。这个方案已经被证明可以和 async views 一起工作。

示例 17.16：简单的 `JsonResponse` View

Code (python)

```
class FlavorApiView(LoginRequiredMixin, View):
    def post(self, request, *args, **kwargs):
        # logic goes here
        return JsonResponse({})

    def get(self, request, *args, **kwargs):
        # logic goes here
        return JsonResponse({})
```

```
    def put(self, request, *args, **kwargs):
        # logic goes here
        return JsonResponse({})

    def delete(self, request, *args, **kwargs):
        # logic goes here
        return JsonResponse({})
```

17.9.2 FBV 路线：django-jsonview

虽然你也可以在 FBVs 中使用 DRF，但那种方式并不具备基于 CBV 的 DRF 全部特性。一个更简单的办法是使用 `django-jsonview`。缺点在于：一旦你要处理完整范围的 HTTP methods，以及更复杂的 API 设计，我们发现 Django 的 FBVs 会因为缺乏对 API 构建和 OpenAPI 的天然支持，而逐渐成为阻碍。

17.9.3 django-tastypie

`django-tastypie` 是一个成熟的 API framework，它实现了自己的一套 class-based view 系统。它比 Django REST Framework 早出现 3 年，是一套功能丰富、成熟、强大、稳定的工具，适合基于 Django models 创建 APIs。

`django-tastypie.readthedocs.io/`

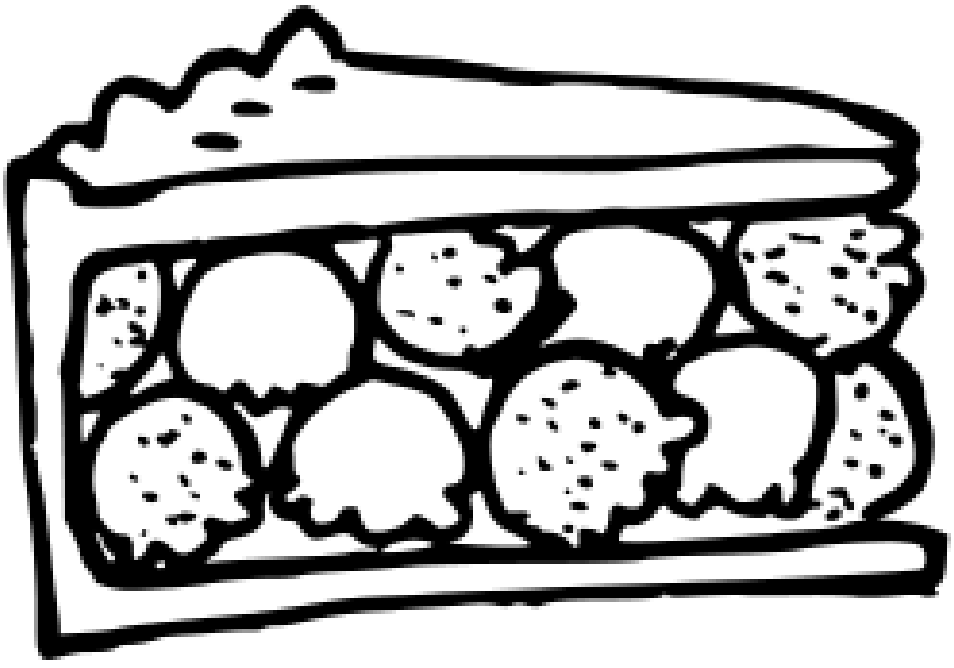

图 17.2: 图 17.2：好吃的派，当然应该装满冰淇淋。

17.10 总结

本章我们讨论了：

- 为什么你应该使用 Django REST Framework。
- 基础 REST API 概念，以及它们与 Django REST Framework 的关系。
- 安全方面的注意事项。
- API 分组策略。
- API 简化策略。
- 基础 REST API 设计原理。
- Django REST Framework 的替代方案。

下一章我们会继续讨论 REST APIs 的另一面，进入第 19 章《JavaScript 与 Django》。

第十八章 使用 Django 构建 GraphQL APIs

警告：本章仍在编写中

由于 Async Views 仍未进入 Django core，Django 与 GraphQL 的整体格局目前还不完整。我们也没有足够时间把自己的想法彻底整理完。

因此，在接下来的日子里，我们会继续扩展这里的内容。也欢迎大家对希望覆盖的话题和条目提出建议，请提交到：`github.com/feldroy/two-scoops-of-django-3.x/issues`

GraphQL 从 2012 年起步，在 API 格式的世界里算是后来者。和 REST 不同，GraphQL 自带 schemas、types，而它最让我们喜欢的一点之一，是内建了处理实时更新的方法（Subscriptions）。API 的客户端可以明确指定自己想要的数据，而由于 schemas 和 types 的存在，查询方式也很容易摸清，因为这让构建 introspection tools 变得非常容易。GraphQL 的响应也很容易序列化成 JSON 或 YAML。正因为这些优点，以及更多别的原因，GraphQL 在过去几年里增长得非常快。

如果你想学习 GraphQL，我们推荐去看 `graphql.org` 上的资料：`graphql.org/learn`

由于 GraphQL 自带查询语言，而且 types 的命名通常都是清晰的英文术语，我们发现：和 REST 相比，把 GraphQL 教给初学者反而更容易。

18.1 打破性能迷思

很多人第一次了解 GraphQL 客户端是如何定义自己想接收的数据时，都会立刻担心性能问题。最常见的质疑是：如果每一次访问请求都可能不一样，那针对查询返回的数据该怎么做缓存？

GraphQL Method	REST Method	动作
`query GetRecords()`	`GET /records`	只读查询
`subscription GetRecords()`	无对应项	打开 websocket，并在发生变化时更新
`query GetRecord(id: X)`	`GET /record/:id`	只读查询
`subscription GetRecord(id: X)`	`GET /record/:id`	打开 websocket，并在发生变化时更新
`mutation CreateRecord()`	`POST /records/`	创建记录
`mutation UpdateRecord(id: X)`	`PUT /records/:id`	更新记录
`mutation DeleteRecord(id: X)`	`DELETE /records/:id`	删除记录

有几件事足以击破这个迷思：

1. 客户端只请求自己真正需要的数据，而不是被迫吞下一整个 REST 风格的大响应，因此它们实际消耗的数据量会明显更少。这会同时显著降低服务器和数据库的负担。
2. 就像 REST API 一样，常见的访问模式仍然可以被识别出来，并通过缓存、索引或代码优化做更好的处理。

当然，这并不是说 GraphQL APIs 对瓶颈免疫。我们的意思只是：它们至少和 REST APIs 一样可控，甚至由于带宽更低，很多时候还会更容易处理。

我们在 Django 中构建 GraphQL APIs 时偏爱的库是 Ariadne。原因如下：

- 当与 ASGI 和 Channels 一起使用时，它是完全异步的，因此可以让 Django 项目通过 GraphQL Subscriptions 提供实时更新。
- 如果你还没准备好进入 async 世界，也依然可以跑在 WSGI 下。
- 它遵循 schema-first，也就是说，你在开始写代码之前就被迫先把设计规格写出来。考虑到 GraphQL schema 的分量极重，这是个非常强的特性。
- 创建 queries 和 mutations 时，只需要遵循一个简单而轻量的 API。
- 它支持 Apollo Federation，也就是一种把多个 GraphQL services 组合成一个 GraphQL service 的规范。

包提示：那 Graphene 呢?

Graphene 是 Python 世界里最早支持 GraphQL specification 的主流库，而且它甚至还带了 Django 集成。能够在 nodes 里使用 Django forms 或 serializers，确实是个加分项。不过，虽然我们尊重它背后的投入，但对我们自己的项目来说，它的一些短板还是让人难以接受。

具体来说：

- Graphene-Django 不支持 GraphQL Subscriptions。这意味着，如果你想拿到实时更新，就只能在客户端轮询 API，或者额外跑一个独立的 websocket 进程。无论哪种，对我们来说都是不可接受的负担。我们也希望，随着 Django 新的 Async Views 逐步成熟，库维护者未来能把这个问题补上。
- 在 Apollo server 之类的平台上，实现 GraphQL APIs 所需的代码量通常很少。相比之下，Graphene 给我们的感觉偏重。

参考：`docs.graphene-python.org/projects/django/`

18.2 GraphQL API 架构

用 Ariadne 快速搭出简单的 GraphQL APIs 并不难，但如果要把它扩展、维护到匹配项目真实需求的程度，就需要多一点思考了。人们通常就是卡在 API 设计这一层。本节会说明一些需要记住的点。

18.2.1 不要把连续递增的 key 当成公开标识符

像 Django 默认提供的 model 主键这种顺序递增 key，如果拿来公开使用，会带来安全隐患。我们会在 28.28 节《永远不要显示顺序递增主键》中详细讨论这个问题。

在这个例子里，我们打算使用 model 的 UUID，而不是 model 主键，来查找记录。我们一直都尽量避免在查找时使用顺序数字。

18.2.2 对 API Modules 使用一致的命名

和其他任何部分一样，项目里的命名必须保持一致。这里有两种我们实践过的模式。

第一种是简单的 GraphQL 架构，适合大多数中小型应用。由于 Ariadne 的 API 很轻量，我们发现把所有 queries 和 mutations 都写进一个 `schema.py` 模块，通常也完全可行。

示例 18.1：简单的 GraphQL 架构

Code (text)

```
config/
+-- schema.py # imports forms & models from flavors app
+-- settings/
+-- urls.py
flavors/
+-- __init__.py
+-- app.py
+-- forms.py
+-- models.py
```

有时候，单个 `schema.py` 模块会大到让人不舒服。这种情况下，把它按 apps 拆开，效果也很好。queries 和 mutations 被移到各自的 app 中，然后再导入到核心 `schema.py` 模块里，在那里调用 `make_executable_schema()`。

示例 18.2：复杂一点的 GraphQL 架构

Code (text)

```
config/
+-- schema.py # imports queries/mutations from flavors app
```

```
+-- settings/
+-- urls.py
flavors/
+-- __init__.py
+-- app.py
+-- api/
|   +-- queries.py # imports models
|   +-- mutations.py # imports forms and models
+-- forms.py
+-- models.py
```

关于这两种架构，请注意下面几点：

- 我们依然在依赖 Django forms 和 models。
- 我们坚持使用直观且一致的命名模式。

18.2.3 尽量把业务逻辑移出 API Views

不管你的 API 是大是小，把逻辑放回它应该待的地方，始终是个好主意。验证应该放在 forms 里（如果你更喜欢 DRF serializers，也可以放在那里），数据库处理则应该放在 models 中。

在 Ariadne 中，很容易开始把业务逻辑一股脑塞进函数或类定义里。要避开这种反模式，因为等到将来你发布 APIv2 时，它会让测试、升级和逻辑复用都变得更困难。

18.2.4 测试你的 API

我们的经验是，测试基于 Ariadne 的 GraphQL，最好的方式是使用 Django 内建的 `RequestFactory` 系统。Ariadne 自己测试 Django 集成的方式可以参考这里：`github.com/mirumee/ariadne/blob/master/tests/django/test_query_`

18.2.5 给你的 API 做版本控制

[原文此节仍为 in progress，未提供正文内容]

18.2.6 对自定义认证方案保持谨慎

[原文此节仍为 in progress，未提供正文内容]

18.3 关闭一个对外 API

[原文此节仍为 in progress，未提供正文内容]

第十九章 JavaScript 与 Django

这一章讨论的是 Django 语境下的 JavaScript。JavaScript 和 Django 最常见的配合方式有下面几种：

- 用 JavaScript 提供整个前端，并消费由 Django 提供的 REST / GraphQL APIs。这就是 Single Page App 路线，通常会配合 React、Vue 等框架实现。
- 用 JavaScript 去增强由 Django 提供、使用 DTL 或 Jinja2 编写的 templates。
- 让 Django admin 变得更高效、更好用。要记住这一点很重要，因为哪怕 Django 创建出来的 APIs 主要是给原生应用提供服务，JavaScript 也往往仍然是项目的一部分。

本章会讨论以下主题：

- 19.1 节：流行的 JavaScript 路线
- 19.2 节：用 JavaScript 消费由 Django 提供的 APIs
- 19.3 节：实时性的烦恼，也就是延迟
- 19.4 节：在 Django 提供的 Templates 中使用 JavaScript
- [原文此处为?? 占位]

19.1 流行的 JavaScript 路线

随着 JavaScript 引擎越来越快，以及相关社区逐渐成熟，一批新的 JavaScript 框架开始崛起，它们就是为了和 REST 或 GraphQL APIs 集成而设计的。到 2020 年，最流行的三个大概是：

React.js `facebook.github.io/react/`

这是一个由 Facebook 创建并维护的 JavaScript 框架及生态系统。它既可以用来创建 HTML、iOS 和 Android 应用，也经常被拿来构建 Single Page Apps，不过同样可以用来增强已有页面。

Vue.js `vuejs.org`

Vue 的设计目标之一，就是让它可以被渐进式采用。虽然它的生态不像 React.js 那么庞大，但更简单的数据模型通常被认为更容易上手。在 Django 世界里，Vue 经常被用于增强现有页面，或者构建 Single Page Apps 前端。

Angular `angular.io`

Angular 是 Google 开源的、只使用 TypeScript 的框架。在 Django 社区里，它没有 React 或 Vue 那么流行。

“React 属于 2019 年。HTML + 最少量的 JS 才是 2020 年的方案。”

作者 David Heinemeier Hansson，Rails、Basecamp 与 Hey 的创始人

除了这些完整的 Single Page Application 框架之外，使用服务端渲染 HTML templates 的方式依然很受欢迎。虽然这看起来像是在“退回旧路”，但这种较老架构下的一些新做法已经证明，它仍然完全可行。像 `hey.com`、`basecamp.com`，以及即将推出的 Writernaut 项目，都展示了如何用传统 HTML templates 做出接近 SPA 的速度和能力。

Vanilla JavaScript `developer.mozilla.org/en-US/docs/Web/JavaScript`

Vanilla JavaScript 指的是不依赖 React、Vue、jQuery 之类额外库的原生 JavaScript。曾经确实有过一个时代，不借助这些库几乎寸步难行，但如今浏览器里的 JavaScript 已经变成了一套功能强大、特性丰富的工具，可以独立支撑项目开发。

小而专注的库 + Vanilla JavaScript

虽然近些年 Vanilla JavaScript 在 DOM 交互方面已经成长了很多，但仍有一些很好用的库可以增强开发者能力、降低代码复杂度。例如：

- **Alpine.js** `github.com/alpinejs/alpine`

Alpine 是一个强韧而极简的框架，专门用来在 markup 中组织 JavaScript 行为。它的语法几乎刻意做得和 Vue 一模一样，因此你能得到 Vue 的那种能力，却不需要背上相应的重量级架构。

- **Turbolinks** `github.com/turbolinks/turbolinks`

它通过 HTML 在服务端渲染 views，并照常用链接跳转页面。当用户点击链接时，Turbolinks 会自动抓取页面、替换其中的 `<body>`，并合并 `<head>`，整个过程都无需承担一次完整页面加载的成本。在 Django 里把它跑顺并不算轻松，但它确实能显著提速那种传统服务端渲染 template 的站点。

jQuery `jquery.com`

jQuery 曾经是 JavaScript 世界的中流砥柱，而它提供的大量功能如今都已经被吸收进了 Vanilla JavaScript，因此在新项目中已经很少再用到它。不过，大量现存 Django 项目依然深度依赖它，这些项目未来多年里仍然需要维护，或者逐步迁移到其他工具。

上面这五种选择，都能大幅改善我们喜欢称之为“即时用户体验”的东西。不过，任何好东西都会伴随着需要思考和需要处理的问题。下面就是我们在“前端项目通过 API 获取内容”这件事上总结出来的一些反模式。

19.1.1 当多页应用已足够时，却硬要构建 Single Page App

用 React、Vue、Angular 这类框架去构建单页应用，当然很有趣。但一个传统 CMS 站点真的有必要做成 SPA 吗？内容页面当然可以带上由 API 驱动的编辑控件，但在做这种网站时，传统 HTML 页面仍然有它的价值。

例如，我们在 2017 年使用过的医疗服务提供商网站，就是一个 SPA 风格站点。它做得很漂亮，所有东西一起滑动、联动的方式也非常惊艳。但一旦我们需要做任何比较型研究，它就几乎毫无用处。

最糟糕的例子，是那个网站的医生搜索系统。它返回了一串医生列表，但我们根本没法方便地比较他们。点开某个医生查看更多信息后，内容会出现在一个滑动 modal 里。我们没法右键把好几个医生同时开到不同标签页中，因为那样做只会把我们带回搜索根页面。虽然可以打印或者给自己发邮件保存单个医生的信息，但和在多个标签页之间来回对比相比，PDF 和邮件根本不是好用的比较工具。

这个网站本来应该给每位医生都提供一个独立的领域引用，也就是独立 URL。无论是在后端由服务器来解析，还是在前端靠 JavaScript 管理 URL，这件事都并不难做。但可惜的是，这种问题在我们换到另一家医疗服务提供商之前，一直都很常见，而且令人痛苦。

19.1.2 升级遗留站点

除非整个站点都要被彻底废弃并重做，否则不要一次性把整个前端全部升级。

在处理 legacy 项目时，通常更容易把新特性作为单页应用局部加入进去。这样项目维护者既能为新功能提供更好的体验，又能保住现有代码库的稳定性。一个很好的例子，就是给现有项目增加一个日历应用。这件事用 Vue 很容易做到，用 React 也可以，只是没那么顺手。

19.1.3 不写测试

当我们刚开始接触一门新语言或新框架时，包括客户端 JavaScript，很容易会想先把测试跳过去。用一句话说：别这样。

客户端开发每年都在变得更复杂、更精细。伴随着客户端标准不断演进，很多事情在客户端这边就是没有服务端那么容易读懂。

我们会在第 24 章《Testing Stinks and Is a Waste of Money!》里讲 Django / Python 测试。一个关于 JavaScript 测试的不错参考是：`stackoverflow.com/questions/300855/javascript-unit-test-tools-for-tdd`

19.1.4 不理解 JavaScript 的内存管理

Single Page Apps 很棒，但那些用户会长时间一直开着不关的复杂实现，会让对象在浏览器中驻留很久。最终，如果管理不好，就会导致浏览器变慢甚至崩溃。每个 JavaScript 框架都会提供一些工具或建议来处理这个潜在问题，所以最好提前了解你所用框架推荐的做法。

19.1.5 在不是 jQuery 的情况下把数据塞进 DOM

在多年使用 jQuery 之后，我们中的一些人已经习惯了把数据直接存进 DOM 元素里，尤其是 Daniel。不过在其他 JavaScript 框架下，这并不是理想做法。它们都有自己处理客户端数据的机制，如果我们不遵守这些机制，就很可能白白错过这些框架承诺的某些能力。

我们建议你认真查清楚自己所选 JavaScript 框架的数据管理方法，并尽可能深地拥抱它们。

19.2 用 JavaScript 消费 Django 提供的 APIs

既然前面我们已经讨论过 REST APIs 的构建，也讲过 template 最佳实践，那现在就把两者合起来。换句话说，这里讲的是：如何利用 Django 提供的工具，把由 REST / GraphQL APIs 管理的内容，通过现代 JavaScript 框架展示给浏览器中的终端用户。

19.2.1 学会如何调试客户端

调试客户端 JavaScript，绝不只是到处写 `console.log()` 和 `console.dir()`。调试和查错有很多工具，其中有些还是特定框架专属的。一旦选定了工具，最好专门花一天时间，认真学会如何编写客户端测试。

参考资料：

- `developers.google.com/web/tools/chrome-devtools`
- `developer.mozilla.org/en-US/docs/Mozilla/Debugging/Debugging_JavaScript`

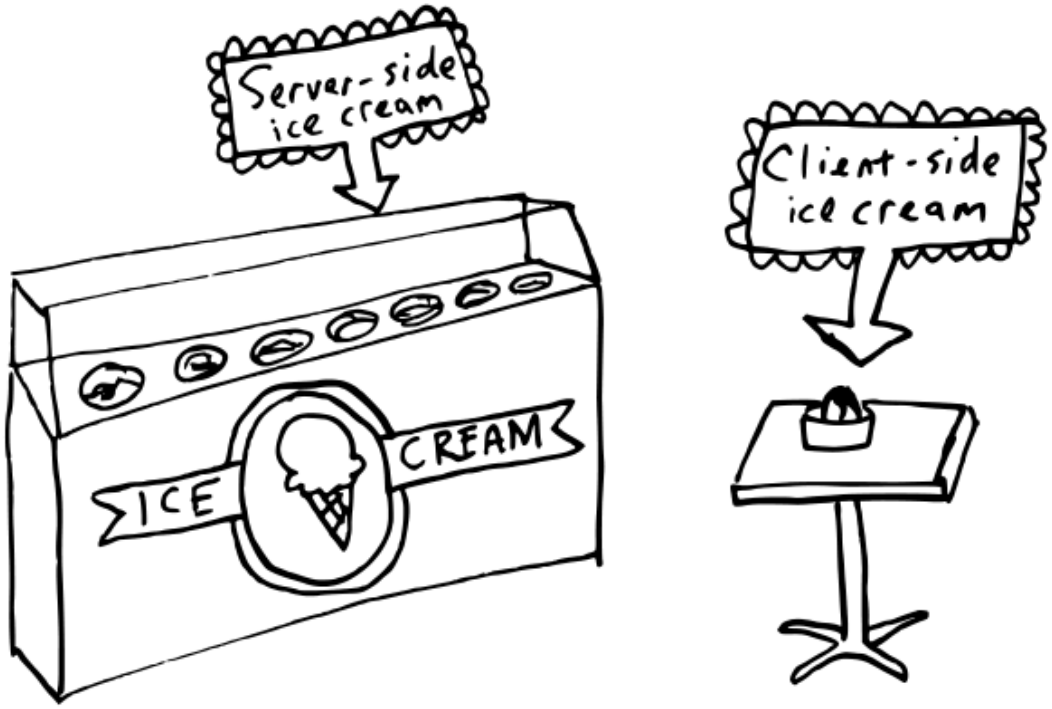

图 19.1: 图 19.1：服务端冰淇淋 vs 客户端冰淇淋。

19.2.2 在可能的情况下，使用由 JavaScript 驱动的静态资源预处理器

大约到 2017 年之前，我们几乎什么都用 Python 做，连 JavaScript 和 CSS 的压缩也不例外。不过，JavaScript 社区在维护这类工具上，比 Python 社区做得更好。这完全没问题，因为既然他们已经把这部分工具链工作做好了，我们就可以把注意力放到别的事情上。

在写下这些文字的时候，这类工作的最常用工具是 webpack。React、Vue 以及几乎所有主流框架都在使用它。webpack 会把浏览器端脚本打包成可以作为静态资源提供的资产。虽然我们更喜欢直接用 Vue 或 React CLI 工具自带的原生 webpack 配置，但真正理解 webpack，本身也是一项非常强大的技能。

参考：

- `webpack.js.org`
- `github.com/owais/django-webpack-loader`

  Owais Lone 的 Django 包，可让 webpack 与 Django 透明协作

19.3 实时性的烦恼，也就是延迟

假设我们已经构建了一个设计良好、索引完善、缓存充分的实时项目，拥有极宽的带宽，把内容输送到全世界。任何负载都扛得住，测试用户也都在为系统的速度和功能鼓掌。一切看起来都很好，我们甚至已经开始期待奖金和加薪。

然后，地球另一端的用户开始抱怨应用太慢。对一大块潜在用户来说，我们费尽心思做出来的“实时”根本不够实时，而客户 / 老板对此非常不满。

恭喜，我们刚刚撞上了光速。

这不是玩笑，而是一个非常现实的问题。这里 Django 不是罪魁祸首，真正的问题是物理定律。HTTP 请求绕地球半圈来回传输所花的时间，是人类可以明显感知到的。再叠加服务端和客户端的处理时间，就很可能让潜在用户或现有用户感到挫败。

另外也别忘了，即便是最快的本地连接，也一样会有抖动和降速。所以所谓“实时”应用，往往都会准备一些机制，来处理这类情况。

19.3.1 解决方案：用动画掩盖延迟

最常见的一种补救方式，就是用由 JavaScript 驱动的动画，把用户的注意力从延迟问题上移开。每次我们使用界面漂亮的单页应用时，几乎都会遇到这种手法，现代 Web 邮箱客户端尤其如此。

19.3.2 解决方案：伪造成功事务

另一种方案，是在客户端先把这次请求当成已经成功到达服务器并完成处理。我们仍然需要补上客户端逻辑来处理失败情况，但因为 JavaScript 框架处理 HTTP 请求本来就是异步的，所以这种做法是可行的，只是可能会比较复杂。

如果你曾经突然发现某个云端表格并没有保存你过去 30 秒输入的数据，那你就见识过这种由 JavaScript 驱动的“小把戏”了。因为这类体验可能非常挫败，所以有些在线工具一旦检测到连接失败，就会直接禁止继续输入数据。

19.3.3 解决方案：基于地理位置部署服务器

在全球七大洲都部署基于地理位置的服务器，当然是一种方案。不过对 Django 来说，这件事实现起来并不简单，无论是编程层面还是数据库层面都一样。它需要大量超出本书范围的技能和经验。

如果你有时间也有预算，这会是一个很值得探索的方向，我们也鼓励你去尝试。不过除非你以前做过这件事，否则你大概率会低估其中投入的工作量。

19.3.4 解决方案：按地域限制用户

有时候我们就是别无选择。也许应用对“实时”性能依赖太强，而多地部署服务器又超出预算。这样做可能会让一些人不高兴，但你多少可以靠一些说法来缓和，比如：“你所在国家 / 地区的支持即将上线！”

19.3.5 AJAX 与 CSRF Token

当你在浏览器里向服务器发 AJAX 调用时，Django 的 CSRF 保护看起来很像是一种“麻烦”。如果你用 AJAX 与 Django 通信，尤其是在 REST APIs 场景下，可能很快就会发现：CSRF token 校验一触发，你就没法顺利向 API 发 POST、PATCH 或 DELETE 请求了。

但它正是 Django 安全性的一部分，所以不要把它关掉！

我们给出的答案是：去认真阅读 Django CSRF 文档里关于 AJAX 的那部分内容，相关资源如下。文档里自带可直接拷进项目使用的 JavaScript 示例代码。

参考：

- `docs.djangoproject.com/en/3.2/ref/csrf/`
- `github.com/jazzband/dj-rest-auth`

  这是一个经过验证的 DRF 认证库，在某些场景下可以在不依赖 CSRF 的情况下使用。
- [原文此处为?? 占位]

警告：不要拿 AJAX 当借口去关闭 CSRF

Django core 开发者 Aymeric Augustin 曾说过，大意是：“……CSRF 保护几乎总是被关闭，因为开发者没能完全搞清楚该怎么把它接对……尤其是在它接受 cookie 认证的时候。”

除非你本来就打算构建的是机器到机器的 API，否则不要做一个关闭了 CSRF 的站点。如果你实在搞不定，就去求助。没人会嘲笑一个想把自己站点做得更安全的人。

19.4 在 Django 提供的 Templates 中使用 JavaScript

19.4.1 JavaScript 又可以放回 Header 里了

很多年来，把脚本链接和 `<script>` tags 放在页面 `body` 的底部都被认为更有优势。现代浏览器在页面加载解析方面已经聪明了很多，因此这种技巧不再是必须的了。

19.4.2 对供 JavaScript 消费的数据使用 JSON 编码

应该依赖 JSON 编码，而不是想办法把 Python 结构直接倾倒进 templates。这样做不仅更容易和客户端 JavaScript 集成，而且更安全。要做到这一点，始终使用 `json_script` filter。

示例 19.1：为 JavaScript 正确地进行 JSON 编码

Code (html)

```
{{ page_data|json_script:"page-data" }}
<script>
var data = JSON.parse(document.getElementById('page').textContent);
injectNameIntoDiv('scoopName', data.scoop.name);
</script>
```

不要像下面这样做。我们必须承认，我们自己多年来也这么干过。

示例 19.2：为 JavaScript 进行不正确的 JSON 编码

Code (javascript)

```
// Rendering a value from the Django template
// Hoping no one injected anything malignant
injectNameIntoDiv('scoopName', {{ scoop.name }});
```

参考：`docs.djangoproject.com/en/3.2/ref/templates/builtins/#json-script`

19.5 强化你的 JavaScript 技能

当我们在客户端实现 REST API 消费时，最重要的事情之一，就是确保自己的 JavaScript 技能已经跟得上。Python 开发者有时喜欢抱怨 JavaScript，但它本身其实是一门能力很强的语言。任何负责任的 Web 开发者，都应该愿意投入时间，把自己的技能提升到足以享受现代 JavaScript 红利的程度。

TODO

19.5.1 学更多 JavaScript!

有很多资源可以帮助你提升基础 JavaScript 技能。我们会在附录 C《更多资源》末尾列出我们最喜欢的一些。

19.6 遵循 JavaScript 编码规范

对于 JavaScript，我们主张前后端工作都参考下面这些指南：

- Felix 的 Node.js 风格指南

  nodeguide.com/style.html
- idiomatic.js

  github.com/rwaldron/idiomatic.js

19.7 总结

TODO

第二十章 替换核心组件的权衡

有些人主张把 Django 技术栈里的核心部件替换成别的东西。到底该不该这么做？

简短回答：不要。

就连 Instagram 的联合创始人之一 Kevin Systrom 都曾在接受 Forbes.com 采访时说过，这完全没必要（bit.ly/2pZxOBO）。

长一点的回答是：从技术上讲，这当然做得到，因为 Django 的各个模块本质上也只是 Python 模块。但它值不值得？只有在下面这些条件同时成立时，它才可能值得：

- 你可以接受牺牲一部分，甚至全部，使用第三方 Django packages 的能力。
- 你可以接受放弃功能强大的 Django admin。
- 你已经非常认真地尝试过用 Django 的核心组件来构建项目，但现在确实撞上了重大阻塞。
- 你已经分析过自己的代码，并且修掉了问题的根因。比如，你已经把 template 中产生的查询数量尽可能降到了最低。
- 你已经探索过所有其他选项，包括缓存、反规范化等手段。
- 你的项目已经是真实在线、拥有海量用户的生产站点。换句话说，你非常确定自己不是在过早优化。
- 你已经认真评估过 Service Oriented Approach（SOA）是否能解决 Django 难以处理的那些问题，并最终否决了这条路。
- 你愿意接受一个事实：从今往后，升级 Django 会变得极其痛苦，甚至根本不可能。

这样听起来是不是就没那么美妙了？

20.1 构建 FrankenDjango 的诱惑

每隔几年，总会有一股新潮流，推动一拨又一拨开发者去替换 Django 的某个核心组件。下面是我们这些年看过的一些流行风向的缩影。

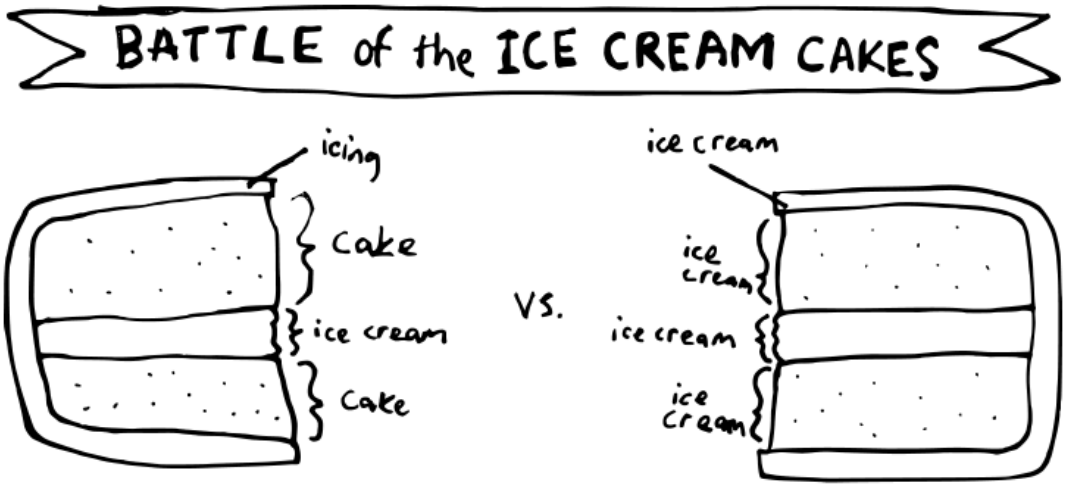

图 20.1: 图 20.1：把蛋糕的更多核心部分替换成冰淇淋，听起来像个好主意。哪块蛋糕会赢？右边那块！

20.2 非关系型数据库 vs 关系型数据库

即便一个 Django 项目用的是关系型数据库来做持久化存储，它通常也仍然会依赖非关系型数据库。比如，如果项目使用了 Memcached 做缓存、Redis 做队列，那它实际上已经在使用非关系型数据库了。

问题出在：有人在没有深入考虑长期影响的前提下，用 NoSQL 方案彻底替代 Django 在关系型数据库这部分的能力。

20.2.1 并不是所有非关系型数据库都符合 ACID

ACID 是下面四个词的缩写：

原子性（Atomicity）这意味着事务中的所有部分要么全部成功，要么全部失败。没有它，你就会面临数据损坏的风险。

一致性（Consistency）这意味着任何事务都会让数据保持在合法状态。字符串还是字符串，整数还是整数。没有它，你就会面临数据损坏的风险。

隔离性（Isolation）这意味着事务中的并发执行不会彼此冲突，也不会泄漏到其他事务。没有它，你就会面临数据损坏的风险。

持久性（Durability）这意味着一旦事务提交成功，即便数据库服务器被关闭，它也依然会保持生效。没有它，你就会面临数据损坏的风险。

表 20.1：基于潮流而替换 Django 组件的理由

潮流	理由
出于性能原因，用 NoSQL 数据库及相应 ORM 替换数据库 / ORM	不可接受：“我有个做给讨厌冰淇淋的人用的社交网络点子。上个月才刚开始做，但它肯定得扩展到几十亿用户！”
出于数据处理原因，用 NoSQL 数据库及相应 ORM 替换数据库 / ORM	可接受：“我们的网站有 5000 万用户，而我已经把索引、查询优化、缓存等能做的都做到头了。我们的 Postgres 集群也快到极限了。我为此做了大量研究，准备试着把一个简单的反规范化数据视图存进 Cassandra，看看是否有帮助。我也清楚 CAP 定理（`en.wikipedia.org/wiki/CAP_theorem`），而对这个视图来说，最终一致性是可以接受的。”
出于数据处理原因，用 NoSQL 数据库及相应 ORM 替换数据库 / ORM	不可接受：“SQL 烂透了！我们要上 MongoDB 这种面向文档的数据库！”
出于数据处理原因，用 NoSQL 数据库及相应 ORM 替换数据库 / ORM	可接受：“PostgreSQL 和 MySQL 的 JSON 数据类型已经能复制 MongoDB 数据存储系统几乎所有核心能力。没错，MongoDB 确实内建了 MapReduce，但把这类事交给任务队列去做反而更容易。”
用 Jinja2、Mako 或别的东西替换 Django 模板引擎	不可接受：“我听说 Jinja2 更快。我完全不懂缓存或优化，但我就是需要 Jinja2！”
用 Jinja2、Mako 或别的东西替换 Django 模板引擎	不可接受：“我讨厌把逻辑放在 Python 模块里。我想把逻辑写进 templates！”
用 Jinja2、Mako 或别的东西替换 Django 模板引擎	可接受：“我有少量几个 views，会生成 1MB 以上、专门给 Google 抓取的 HTML 页面。我会利用 Django 原生支持的多模板语言能力，用 Jinja2 渲染这些超大页面，其余部分继续用 Django Template Language。”

你注意到了吗？上面关于 ACID 的每一条描述最后都落在了同一句话上：“没有它，你就会面临数据损坏的风险。”这是因为对很多 NoSQL 引擎来说，ACID 合规机制要么很弱，要么几乎没有。对缓存这种用途来说，数据偶尔坏掉往往问题不大；但如果你的项目处理的是持久化医疗数据或电商数据，那就是完全另一回事了。

20.2.2 不要拿非关系型数据库去做关系型任务

想象一下，如果我们要用一个非关系型数据库来追踪房产交易、房产所有者，以及美国 50 个州中适用于他们的房产法律。这里面有大量不可预知的细节，所以无 schema 的数据存储，看起来是不是很适合这项任务？

也许吧……

但我们还得追踪房产、房产所有者与 50 个州法律之间的关系。我们的 Python 代码必须自己维护所有这些组件之间的引用完整性。我们还得确保正确的数据进到正确的位置。

像这种任务，老老实实用关系型数据库。

20.2.3 别理会炒作，自己做研究

人们常说，非关系型数据库比关系型数据库更快，也更容易扩展。不管这话到底有几分真，都不要不加思考地吞下任何某种替代数据库方案背后公司的营销话术。

更好的做法，是像我们一样：去找 benchmark，看那些讲“什么情况下成功、什么情况下失败”的案例研究，然后尽可能独立地形成自己的判断。

另外，在你对主项目基础设施做重大变更之前，先拿那些不熟悉的 NoSQL 数据库去做些小型 hobby side projects 试手。你的主代码库不是游乐场。

来自公司和个人的经验教训：

- Pinterest：
  `medium.com/@Pinterest_Engineering/stop-using-shiny-3e1613c2ce14`
- Dan McKinley 在 Etsy 工作时的总结：
  `mcfunley.com/why-mongodb-never-worked-out-at-etsy`
- 《什么时候该在 Django 里使用 MongoDB》：
  `daniel.feldroy.com/when-to-use-mongodb-with-django.html`

20.2.4 我们如何在 Django 中使用非关系型数据库

我们的偏好做法如下：

- 如果要使用非关系型数据存储，就把它限制在缓存、队列，以及有时用于反规范化数据这种短期用途上。但如果能避免，还是尽量避免，以减少系统中的活动部件数量。
- 对于长期的关系型数据，以及有时也包括反规范化数据，我们使用关系型数据存储（PostgreSQL 的数组字段和 JSON 字段就很适合这类任务）。

对我们来说，这就是能让 Django 项目真正发光的甜蜜点。

20.3 那替换 Django Template Language 呢?

我们的主张是：除了那种需要渲染超大体量内容的场景外，尽量完整坚持使用 Django Template Language（DTL）。不过，既然 Django 现在已经原生支持替代模板系统，这个用例相关的讨论我们就移到第 16 章《Django Templates 与 Jinja2》里了。

20.4 总结

永远为合适的工作选择合适的工具。我们偏好使用原生 Django 组件，就像我们偏好用冰淇淋勺来挖冰淇淋一样。不过，确实也会有某些时候，别的工具更合适。

只是别追随那种“把蔬菜拌进冰淇淋”的潮流。你不可能真的用号称“高性能”的西兰花、玉米和菠菜口味，去替代经典的草莓、巧克力和香草。这就太过头了。

第二十一章 使用 Django Admin

当人们问起："和其他 Web 框架相比，Django 的优势是什么？"时，最先浮现在大家脑海里的，通常就是 admin。

想象一下，如果每一加仑冰淇淋都自带一个 admin 界面。你不仅能看到配料列表，还能新增 / 编辑 / 删除配料。要是有人以你不喜欢的方式动了你的冰淇淋，你还可以限制甚至撤销他们的访问权限。

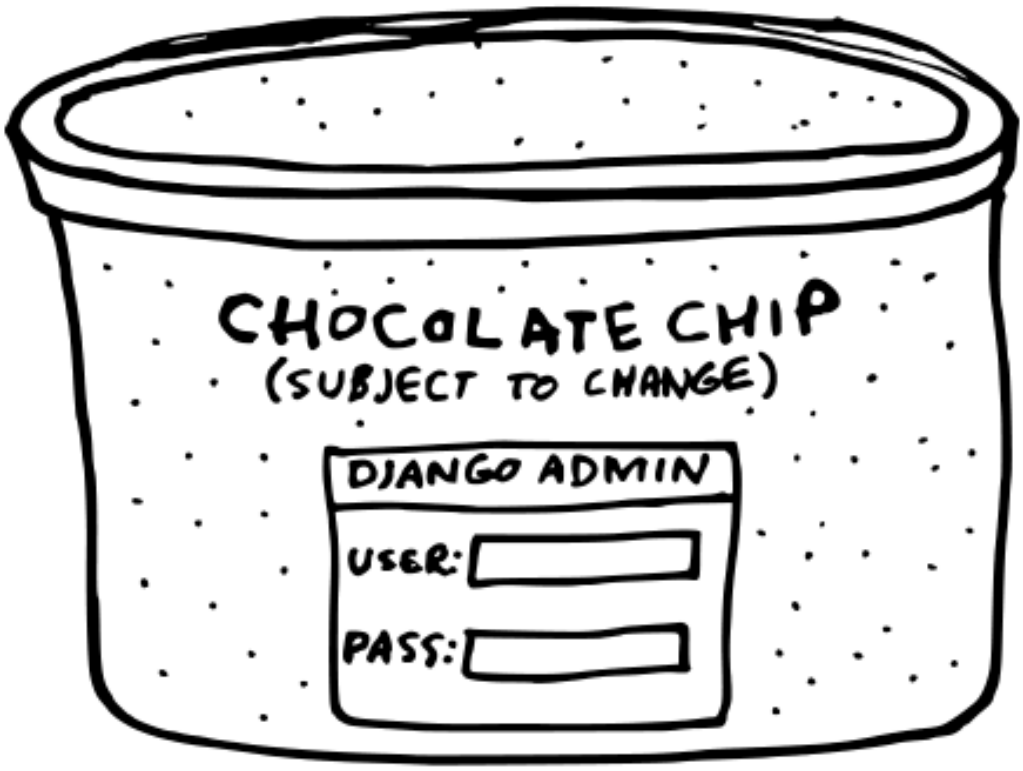

图 21.1: 图 21.1：带有 admin 界面的巧克力碎冰淇淋。

很超现实，对吧？但这正是那些从其他背景转过来的 Web 开发者，第一次使用 Django admin 界面时的感受。它几乎不需要你做什么额外工作，就能自动为 Web 应用赋予巨大的管理能力。

21.1 它不是给终端用户用的

Django admin 界面是为站点管理员设计的，不是给终端用户用的。它是一个让站点管理员新增 / 编辑 / 删除数据，以及执行站点管理任务的地方。

虽然你确实可以硬把它拉伸成终端用户也能使用的东西，但你真的不该这么做。它本来就不是为每个站点访客设计的。

21.2 定制 Admin 还是新建 Views

通常来说，对 Django admin 做重度定制并不值得。很多时候，自己从头写一个简单的 view 或 form，就能用更少的工作量得到同样的功能。

对客户项目来说，我们一贯发现：与其拼命把 admin 改造成客户想要的样子，不如直接给客户做一套定制管理 dashboard，效果反而更好。

21.3 查看对象的字符串表示

一个 Django app 的默认 admin 页面，通常会显示一串看起来非常泛泛的对象列表，大概像这样：

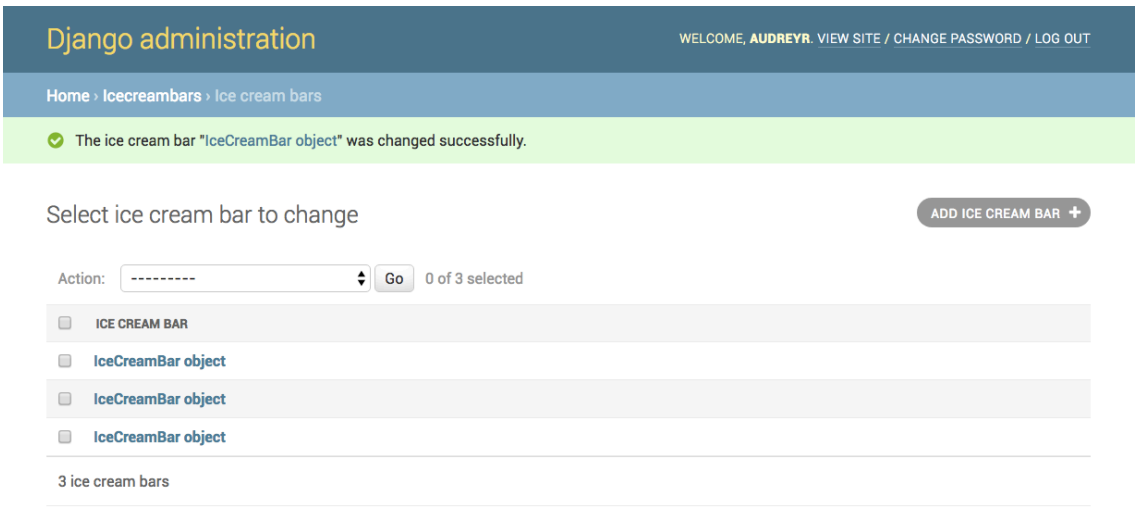

图 21.2: 图 21.2：某个冰淇淋棒 app 的 admin 列表页。

之所以会这样，是因为 `IceCreamBar` 对象默认的字符串表示只是“IceCreamBar object”。如果能显示得更有意义一些，不是更好吗？

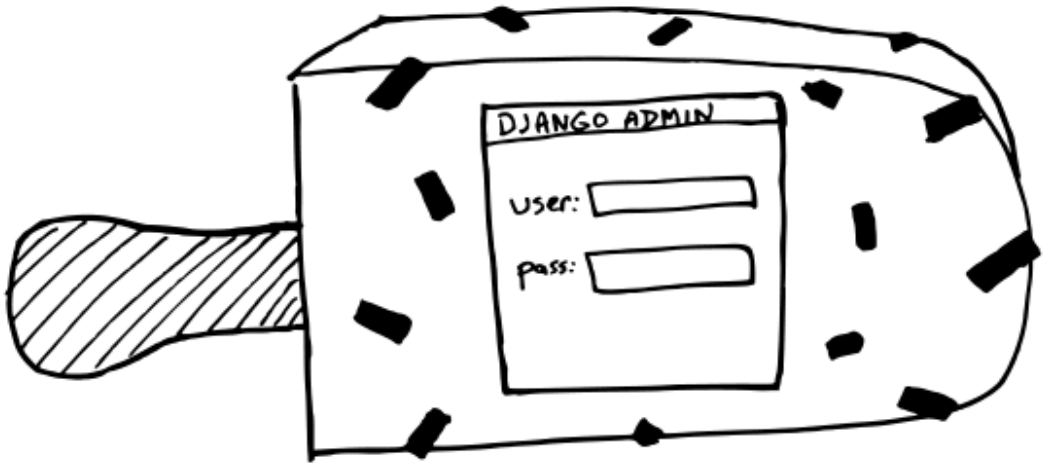

图 21.3: 图 21.3：什么？冰淇淋棒居然也有 admin 界面？

21.3.1 使用 `__str__()`

实现 `__str__()` 非常直接：

示例 21.1：对象的字符串表示

Code (python)

```
from django.db import models


class IceCreamBar(models.Model):
    name = models.CharField(max_length=100)
    shell = models.CharField(max_length=100)
    filling = models.CharField(max_length=100)
    has_stick = models.BooleanField(default=True)

    def __str__(self):
        return self.name
```

结果如下：

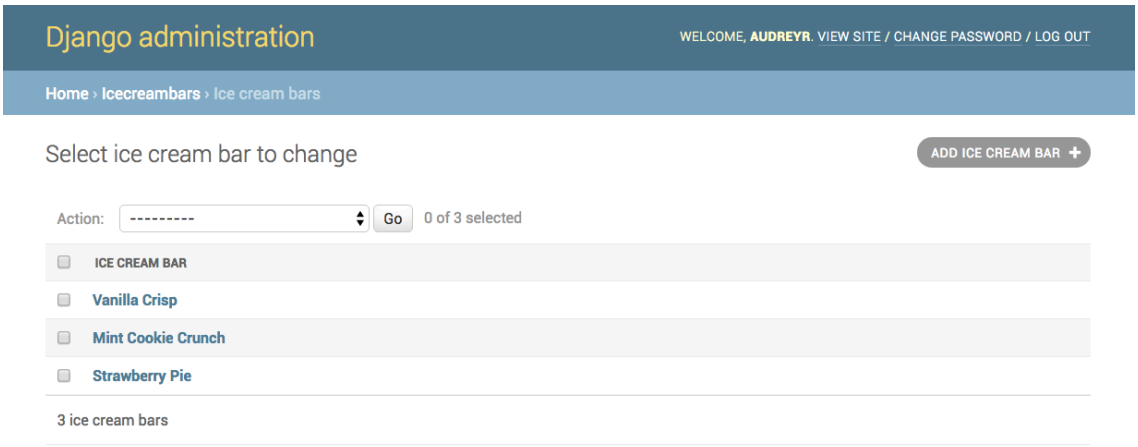

图 21.4: 图 21.4：改进后的 admin 列表页，对象字符串表示更友好了。

而且这不只是 admin 页面里更好看而已。当你在 shell 里操作时，也能看到更友好的字符串表示：

示例 21.2：冰淇淋棒类型列表

Code (python)

```
>>> IceCreamBar.objects.all()
[<IceCreamBar: Vanilla Crisp>, <IceCreamBar: Mint Cookie Crunch>,
<IceCreamBar: Strawberry Pie>]
```

每当你对一个对象调用 `str()` 时，`__str__()` 方法都会被触发。这会发生在 Django shell、templates，以及进一步延伸到 Django admin 中。

因此，尽量让 `__str__()` 的结果成为一种友好、可读的 Django model 实例表示。

21.3.2 使用 `list_display`

如果你想以一种“并不完全等同于对象字符串表示”的方式来改变 admin 列表展示，那么就使用 `list_display`。

示例 21.3：Admin 列表展示

Code (python)

```
from django.contrib import admin
```

```
from .models import IceCreamBar


@admin.register(IceCreamBar)
class IceCreamBarModelAdmin(admin.ModelAdmin):
    list_display = ('name', 'shell', 'filling')
```

指定这些字段后的结果如下：

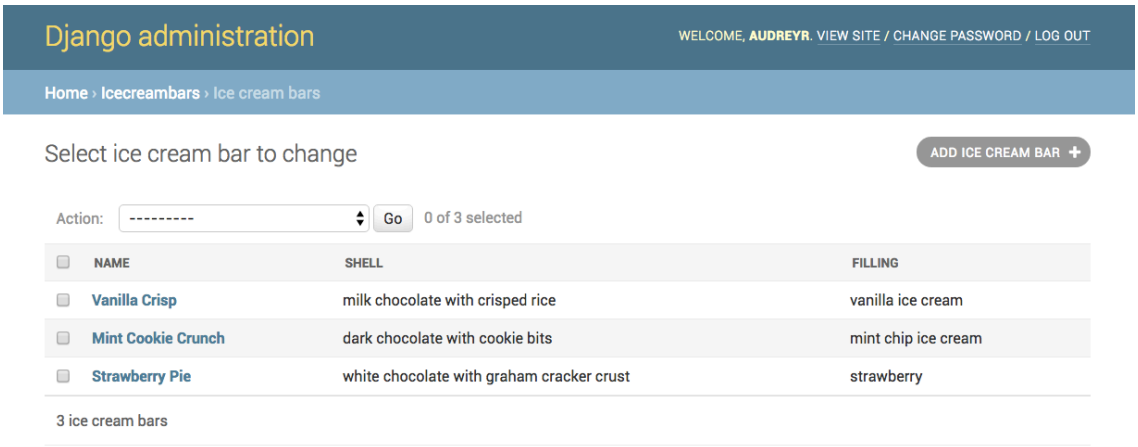

图 21.5: 图 21.5：对 admin 列表页的进一步改进。

21.4 给 ModelAdmin 类添加可调用对象

你可以使用 methods 和 functions 这类 callable，给 Django 的 `django.contrib.admin.ModelAdmin` 类添加功能。这样你就能真正去调整列表页和详情页，让它们更符合你的冰淇淋项目需求。

例如，在 Django admin 中查看某个 model 实例的精确 URL，是一种很常见的需求。即使你给 model 定义了 `get_absolute_url()` 方法，Django admin 默认提供的也只是一个指向 redirect view 的链接，而这个 URL 和对象真实 URL 往往并不一样。还有些情况下，`get_absolute_url()` 根本没有意义，比如 REST APIs。

在下面的例子里，我们演示了如何使用一个简单 callable，来给目标 URL 提供一个直接链接：

示例 21.4：给 `ModelAdmin` 类添加可调用对象

Code (python)

```
# icecreambars/admin.py
from django.contrib import admin
from django.urls import reverse, NoReverseMatch
from django.utils.html import format_html

from .models import IceCreamBar


@admin.register(IceCreamBar)
class IceCreamBarModelAdmin(admin.ModelAdmin):
    list_display = ('name', 'shell', 'filling')
```

```
    readonly_fields = ('show_url',)

    def show_url(self, instance):
        url = reverse(
            'icecreambars:ice_cream_bar_detail',
            kwargs={'pk': instance.pk},
        )
        response = format_html("""<a href="{0}">{0}</a>""", url)
        return response

    show_url.short_description = 'Ice Cream Bar URL'
```

一图胜千言，下面就是这个 callable 带来的效果：

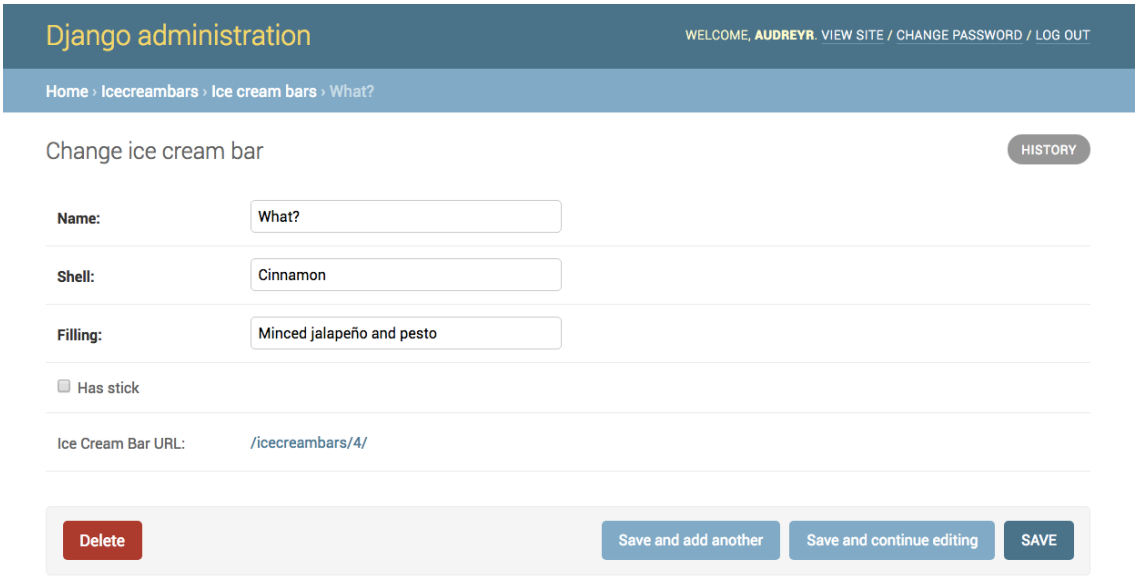

图 21.6: 图 21.6：在 Django Admin 中显示 URL。

警告：展示用户数据中的 HTML 时，始终使用 `format_html`

任何时候，只要你在 admin 中展示的数据包含 HTML tags，都应该使用 `format_html`，以确保参数在渲染过程中会被正确转义。

关于这一点，我们会在 28.9.1 节《优先使用 `format_html` 而不是 `mark_safe`》中讲得更详细。

21.5 注意多用户环境下的复杂性

Django admin 并不会把某条记录锁定给某个特定 staff 或 admin 用户。对于只有一个 admin 的项目来说，这通常没问题；但在多用户项目里，这可能会变成一个很严重的问题。情况大概会是这样：

1. Daniel 通过 Django admin 编辑“Peppermint Sundae”冰淇淋棒这条记录。他刚开始修改，就接到了 Icecream-landia 市场总监打来的电话，于是把页面开着离开了。
2. 与此同时，Audrey 也决定修改“Peppermint Sundae”这条记录。她花了十分钟做完自己的修改，然后保存。
3. Daniel 挂掉电话，回来终于保存了自己的修改。于是，他把 Audrey 的修改覆盖掉了。

如果你的 Django admin 有多个用户能访问，就必须意识到这种情况确实会发生。

21.6 Django 的 Admin 文档生成器

Django 提供的更有意思的开发者工具之一，是 `django.contrib.admindocs` 包。它诞生于一个比我们在第 25 章《Documentation: Be Obsessed》里提到的那些文档工具更早的年代，但直到今天，它仍然是个很有用的工具。

它之所以有用，是因为它会对 Django 框架做 introspection，并显示项目组件的 docstrings，比如 models、views、自定义 template tags 和自定义 filters。即便项目组件里根本没有写任何 docstrings，光是能看到一份那些平时不太容易摸清楚的项目元素列表，比如命名奇怪的自定义 template tags 和 filters，在探索一个复杂的现有应用架构时，也已经很有帮助了。

使用 `django.contrib.admindocs` 很简单，不过我们喜欢把官方文档里的步骤顺序调整一下：

1. 先在项目 virtualenv 中执行 `pip install docutils`。
2. 把 `django.contrib.admindocs` 加入 `INSTALLED_APPS`。
3. 在根 URLConf 中加入：

`path('admin/doc/', include('django.contrib.admindocs.urls'))` 并确保它出现在 `admin/` 入口之前，这样对 `/admin/doc/` 的请求才不会先被后者接走。

1. 可选：如果你想使用 admindocs bookmarklets，需要安装 `XViewMiddleware`。

都配置好之后，访问 `/admin/doc/` 去看看。你很可能会发现项目里大量代码根本没有任何文档。这件事既可以参考 `django.contrib.admindocs` 的官方文档：`docs.djangoproject.com/en/3.2/ref/contrib/admin/admindocs/` 也可以去看我们自己的第 25 章《Documentation: Be Obsessed》。

21.7 在 Django Admin 中使用自定义皮肤

这些年来，社区里一直有人尝试给 Django Admin 换皮或做主题。范围从历史悠久、稳定而且非常流行的 `django-grappelli`，到较新的后来者都有。它们允许你做从简单到复杂不同层次的定制。

包提示：自定义 `django.contrib.admin` 皮肤

下面是一些更流行的自定义皮肤：

- `django-grappelli` 可以说是所有 Django 自定义皮肤的祖师爷。稳定、强悍，而且风格独特但友好。
- `django-suit` 基于大家熟悉的 Bootstrap 前端框架构建。
- `jet-admin` 是一个基于 API 的 Admin Panel Framework。

更完整的列表可以在这里找到：`djangopackages.org/grids/g/admin-styling/`

Django 社区这么大，为什么皮肤却没多到泛滥？

原因是，除了最基础的那类纯 CSS 级别修改之外，制作自定义 Django 主题其实非常困难。只要认真读过这些项目源码，就会很清楚：要写一个自定义 admin skin，你往往得动用一些相当“玄学”的代码，去兼容 `django.contrib.admin` 里的各种古怪细节。

`django-grappelli` 的维护者 Patrick Kranzlmueller 在他那篇《A Frontend Framework for the Django Admin Interface》的文章中，对此做了非常细致的说明。链接如下：

- `sehmaschine.net/blog/django-admin-frontend-framework`

下面是在使用自定义 `django.contrib.admin` 皮肤时的一些建议。

21.7.1 评估点：文档就是一切

前面已经提到过，给 `django.contrib.admin` 写一套自定义皮肤并不轻松。那些成功的皮肤项目，虽然接进项目通常不算难，但真正容易伤人的，是各种边角情况，而这些情况几乎总是和扩展 `ModelAdmin` 对象有关。

因此，在评估某个项目是否值得引入时，要看它的文档是否远远超出了“安装说明”这个层次。

21.7.2 对你自己写的任何 Admin 扩展都要写测试

按我们的经验，客户通常会喜欢那些更现代的主题，但你必须非常小心自己把这些 admin skins 扩展到什么程度。那些在原生 `django.contrib.admin` 下工作得很好的东西，在自定义皮肤里完全可能坏掉。因为这些皮肤往往得用一些奇怪的方式去包裹 `django.contrib.admin` 的抽象层，所以一旦出问题，调试起来就可能成为令人麻木的噩梦。

因此，如果你使用了自定义皮肤，最佳实践就是给 admin 写测试，尤其是任何定制过的部分。没错，前期确实会多花一点工夫，但这意味着你能早得多地把 bug 抓出来。

关于测试的更多内容，请看第 24 章《Testing Stinks and Is a Waste of Money!》。

21.8 保护 Django Admin

既然 Django admin 赋予了站点管理员普通用户不具备的特殊权力，那么把它做得更安全，就是一项很好的实践。

21.8.1 修改默认 Admin URL

默认情况下，admin URL 是 `yoursite.com/admin/`。把它改成某个更长、更难猜的路径。

提示：Jacob Kaplan-Moss 谈修改 Admin URL

Django 项目共同领导者 Jacob Kaplan-Moss 曾表示，大意是：给 admin 想一个不同的名字，甚至不同的域名，都是一层额外但非常容易实现的安全防护。

这样做还能防止攻击者轻松地给你的网站做指纹识别。比如，他们可以通过观察 `admin/` 登录页的 HTML 内容，判断你使用的是哪个 Django 版本，有时甚至能精确到小版本号。

21.8.2 使用 `django-admin-honeypot`

```
TODO - Ask Derek to update this package
```

如果你特别担心有人试图攻破 Django 站点，那么 `django-admin-honeypot` 是个会在 `admin/` 放一个伪 Django admin 登录页，并记录所有尝试登录者信息的包。

更多信息见：`github.com/dmpayton/django-admin-honeypot`

21.8.3 只允许通过 HTTPS 访问 Admin

这一点其实已经隐含在 28.6 节《HTTPS Everywhere》里了，但我们还是想在这里特别强调：你的 admin 必须放在 TLS 保护之下。

如果你的站点仍然允许纯 HTTP 访问，那么你就得把 admin 部署到一个正确加固过的域名上，这会让部署复杂度进一步上升。你不仅要多维护一套部署流程，还得在 URLConf 中加入逻辑，把 admin 从 HTTP 访问里剔除出去。以作者们的经验来看，最简单的做法还是把整个站点都放到 TLS / HTTPS 上。

没有 TLS 的话，如果你在开放 WiFi 网络中登录 Django admin，那么别人要嗅探到你的 admin 用户名和密码，几乎轻而易举。

21.8.4 基于 IP 限制 Admin 访问

把 Web 服务器配置成只允许某些 IP 地址访问 Django admin。去查你所使用的具体 Web 服务器的配置方式。

- Nginx 相关说明：

`tech.marksblogg.com/django-admin-logins.html`

一个勉强可以接受的替代方案，是把这套逻辑写进 middleware。之所以更推荐在 Web 服务器层处理，是因为每一个 middleware 组件，都会在 views 外面再包上一层额外逻辑；不过有些情况下，这可能是你唯一的选择。比如，你所使用的平台即服务并没有给你足够细粒度的 Web 服务器配置权限。

21.9 保护 Admin Docs

既然 Django admin docs 会向站点管理员展示项目是如何构建出来的，那么和 Django admin 一样，把它们额外保护起来，也是很好的实践。沿用前一节关于 Django admin 的思路，我们建议：

- 把 admin docs 的 URL 改成不是 `yoursite.com/admin/doc/` 的其他路径。
- 只允许通过 HTTPS 访问 admin docs。
- 基于 IP 限制 admin docs 的访问。

21.10 总结

本章我们讨论了以下内容：

- 谁应该使用 Django admin。
- 什么时候该使用 Django admin，什么时候该自己做新的 dashboard。
- 对象的字符串表示。
- 如何给 Django admin 类添加 callables。
- Django 自带的 admin docs。
- 为什么我们鼓励你认真保护 Django admin。
- 以及在使用自定义 Django skins 时的一些建议。

第二十二章 处理 User Model

Django 自带了对用户记录的内建支持。这是个非常有用的特性，而一旦你学会如何在它的基础上扩展和增强，它的价值就会翻倍。下面我们来过一遍相关的最佳实践。

22.1 使用 Django 自带的工具来定位 User Model

官方推荐的方式，是像下面这样获取 user 类：

示例 22.1：使用 `get_user_model()` 获取 User 记录

Code (python)

```
# 默认 user model 定义
>>> from django.contrib.auth import get_user_model
>>> get_user_model()
<class django.contrib.auth.models.User>

# 当项目定义了自定义 user model 时
>>> from django.contrib.auth import get_user_model
>>> get_user_model()
<class profiles.models.UserProfile>
```

根据项目配置不同，这里有可能拿到两种不同的 User model 定义。这并不意味着一个项目里可以同时存在两个不同的 User model；它真正的意思是：每个项目都可以定制自己的 User model。

22.1.1 在指向 User 的外键中使用 `settings.AUTH_USER_MODEL`

在 Django 里，把 `ForeignKey`、`OneToOneField` 或 `ManyToManyField` 关联到 User 的官方推荐方式如下：

示例 22.2：使用 `settings.AUTH_USER_MODEL` 定义 Model 关系

Code (python)

```
from django.conf import settings
from django.db import models


class IceCreamStore(models.Model):
    owner = models.OneToOneField(settings.AUTH_USER_MODEL)
    title = models.CharField(max_length=255)
```

没错，看起来是有点怪，但这确实是 Django 官方文档建议的做法。

警告：不要修改 `settings.AUTH_USER_MODEL`！

一旦在项目里设定了 `settings.AUTH_USER_MODEL`，再去修改它，就意味着你还要相应地修改数据库 schema。给 User model 增加或调整字段是一回事，重新创建一整个新的 User 对象体系则是另一回事。

22.1.2 不要在指向 User 的外键中使用 `get_user_model()`

这样做不好，因为它很容易制造 import loops。

示例 22.3：错误地使用 `get_user_model()`

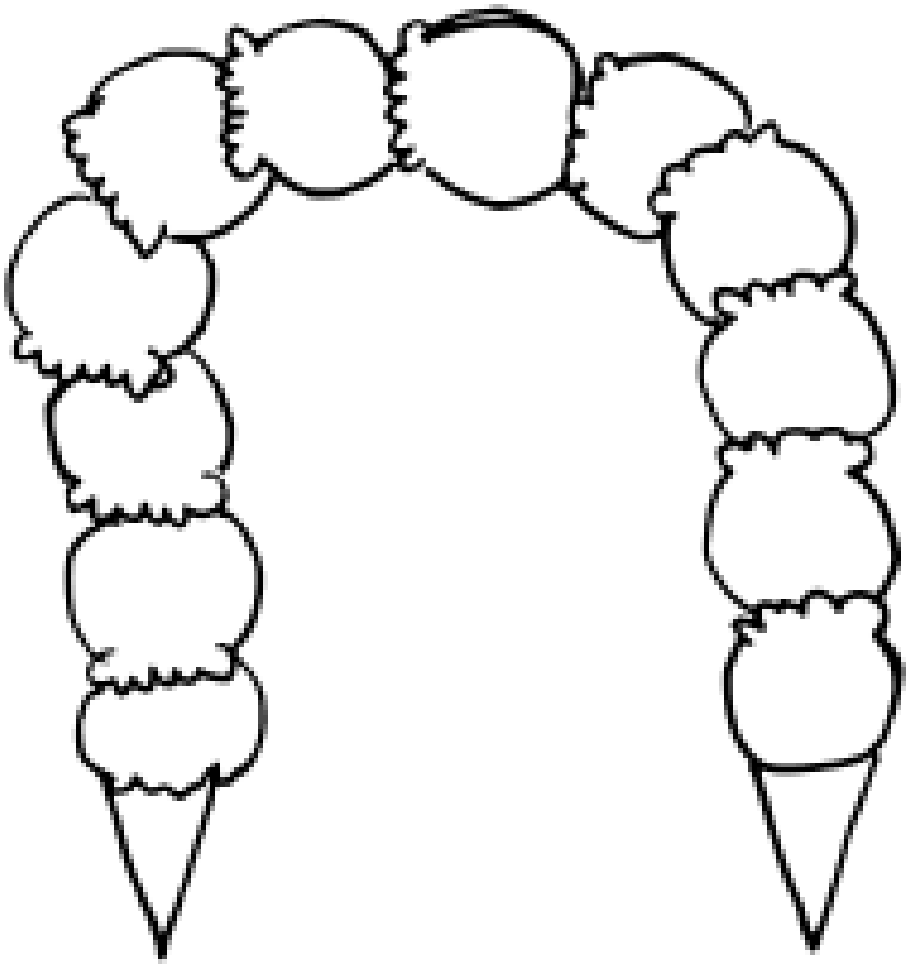

图 22.1: 图 22.1：这看起来也很怪。

Code (python)

```
# DON'T DO THIS!
from django.contrib.auth import get_user_model
from django.db import models


class IceCreamStore(models.Model):
    # 下面这一行很容易制造 import loops。
    owner = models.OneToOneField(get_user_model())
    title = models.CharField(max_length=255)
```

22.2 为 Django 项目定制 User 字段

在 Django 中，只要我们补齐所需的方法和属性，就可以创建带有自己字段的 user model。

包提示：定义自定义 User Models 的库

`django-authtools` 是一个能让定义自定义 user models 更轻松的库。其中尤其有用的是 `AbstractEmailUser` 和 `AbstractNamedUser` 这两个 models。

即便你最后并没有使用 `django-authtools`，它的源码也很值得读一读。

在本书写作时，`django-authtools` 还不支持 Django 3，尽管已经有人提了对应的 pull request。尽管如此，正如前面说的，它的代码依然很值得参考。

22.2.1 方案一：继承 AbstractUser

如果你总体上喜欢 Django 默认 `User` model 自带的那些字段，只是还需要额外字段，那就选这个方案。顺带一提，每次我们开始一个新项目时，首先都会看这个方向。尤其当配合 `django-authtools` 的基础 models、forms 和 admin objects 使用时，我们觉得这通常是实现自定义 user models 最快、最轻松的办法。

下面是一个继承 `AbstractUser` 的例子：

示例 22.4：继承 `AbstractUser`

Code (python)

```
# profiles/models.py
from django.contrib.auth.models import AbstractUser
from django.db import models


class KarmaUser(AbstractUser):
    karma = models.PositiveIntegerField(
        verbose_name='karma',
        default=0,
        blank=True,
    )
```

你还需要在 settings 里加上这个配置：

示例 22.5：设置 `AUTH_USER_MODEL`

Code (python)

```
AUTH_USER_MODEL = 'profiles.KarmaUser'
```

22.2.2 方案二：继承 AbstractBaseUser

AbstractBaseUser 是最精简的方案，只自带 3 个字段：password、last_login 和 is_active。

如果下面这些情况成立，就可以考虑这个方案：

- 你不满意默认 User model 提供的字段，比如 first_name 和 last_name。
- 你更希望从一块极其干净的白板开始继承，但又想保留 AbstractBaseUser 在密码存储上的合理默认做法。

如果你准备沿着这条路走，我们建议阅读下面这些内容：

官方 Django 文档示例：docs.djangoproject.com/en/3.2/topics/auth/customizing/#a-full-example

django-authtools 的源码，尤其是 admin.py、forms.py 和 models.py：github.com/fusionbox/django-authtools

22.2.3 方案三：从关联 Model 反向挂回去

这套做法和 Django 1.5 之前项目里常见的“Profile”models 很像。不过先别急着把它当作纯 legacy 技术扔掉，先看看下面这些使用场景。

使用场景：构建第三方包

- 我们正在开发一个准备发布到 PyPI 的第三方包。
- 这个包需要为每个用户存储额外信息，比如 Stripe ID 或其他支付网关标识符。
- 我们希望尽量少侵入现有项目代码。Loose coupling!

使用场景：内部项目需求

- 我们正在开发自己的 Django 项目。
- 我们希望不同类型的用户拥有不同字段。
- 某些用户也许会同时具备多种用户类型的组合。
- 我们希望在 model 层处理这件事，而不是把逻辑推到其他层去。
- 我们希望它能和前面方案一或方案二里的自定义 user model 一起使用。

以上任一使用场景，都足以成为继续使用这项技术的理由。

为了让这个方案成立，我们继续使用 django.contrib.auth.models.User（最好通过 django.contrib.auth.get_user_ 来拿），并把相关字段放进独立 models 中，例如 Profiles。下面是个例子：

示例 22.6：自定义 User Profile 示例

Code (python)

```python
# profiles/models.py
from django.conf import settings
from django.db import models

from flavors.models import Flavor


class EaterProfile(models.Model):
    # 默认用户 profile

    # 如果你这样做，那么你需要要么挂一个 post_save signal，
    # 要么在首次登录时重定向到 profile_edit view。
    user = models.OneToOneField(settings.AUTH_USER_MODEL)

    favorite_ice_cream = models.ForeignKey(
        Flavor,
        null=True,
        blank=True,
    )
```

```
class ScooperProfile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL)
    scoops_scooped = models.IntegerField(default=0)


class InventorProfile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL)
    flavors_invented = models.ManyToManyField(
        Flavor,
        null=True,
        blank=True,
    )
```

用这种方式,我们就可以很轻松地通过 ORM 查询任意用户最喜欢的冰淇淋:`user.eaterprofile.favorite_ice_cream`。

另外，`ScooperProfile` 和 `InventorProfile` 也分别保存了只适用于那类用户的数据。因为这些数据被隔离进了各自专用的 models，不同用户类型之间发生意外干扰的概率就会小很多。

这种方案唯一的缺点，是 profile 的复杂度或者配套代码很容易越走越远。和往常一样，让代码尽可能简单、尽可能干净。

警告：第三方库不应该去定义 User Model

除非一个库存在的明确目的，就是替项目定义自定义 user models，像 `django-authtools` 那样，否则第三方库不应该使用方案一或方案二去给 user models 增加字段。它们应该依赖方案三。

22.3 处理多种用户类型

Django 初学者经常会问的一个问题是：如何处理多种用户类型？例如在冰淇淋店场景里，要怎么处理顾客、员工和店主这几类用户。很多人第一反应是定义两个不同的 user models，但这条路非常危险，Django 根本不是按这种方式设计的。

正确做法是：只使用一个 model，然后用合适方式给它打标记。这里主要有两种做法。

22.3.1 增加一个用户类型字段

这种方法假设不同角色拥有相同的数据和方法。在 user model 中增加一个 `choices` 字段，用来区分用户类型。这样整个 Django 项目里就都可以基于角色做检查了。

示例 22.7：基于 Choices 的用户类型字段

Code (python)

```
class User(AbstractUser):
    class Types(models.TextChoices):
        EATER = "EATER", "Eater"
        SCOOPER = "SCOOPER", "Scooper"
        INVENTOR = "INVENTOR", "Inventor"

    # 这个用户属于什么类型?
    type = models.CharField(
        _("Type"),
        max_length=50,
```

```
        choices=Types.choices,
        default=Types.EATER,
    )
```

请注意，我们并不依赖多个 BooleanField 的组合来控制角色。那种方式只在非常有限的场景里能工作，而随着站点规模和复杂度增长，久而久之就会变得越来越混乱。更好的做法，是用一个单独字段来控制 User 的角色。

再进一步说，我们在实际项目里更常用的，往往不是这里展示的 CharField，而是指向 Role model 的 ForeignKey 关系。如果用户需要拥有多个角色，那我们要么会把它做成 ManyToManyField，要么就直接使用 Django 内建的 Group 系统。使用 Django 自带 Group 系统的缺点，是相关文档比较少。

22.3.2 增加用户类型字段，再配合 Proxy Models

不同类型用户通常会有不同的方法和属性。比如，一个 SCOOPER 可能会有 scoop_icecream() 方法，而一个 EATER 可能会有 consume() 方法。

我们还希望：针对某种用户类型发起查询时，代码能显得尽可能明显且明确。因为一旦忘记加上对用户类型的过滤，就很容易不小心把某些本不该给他们的能力暴露出去。

Proxy models 让这件事变得很容易。首先，在 model 中增加一个 base_type 属性，并扩展内建的 save() 方法：

示例 22.8：base_type 属性

Code (python)

```
class User(AbstractUser):
    class Types(models.TextChoices):
        EATER = "EATER", "Eater"
        SCOOPER = "SCOOPER", "Scooper"
        INVENTOR = "INVENTOR", "Inventor"

    # 确保通过 proxy models 创建新用户时也能正常工作
    base_type = Types.EATER

    # 这个用户属于什么类型?
    type = models.CharField(
        _("Type"),
        max_length=50,
        choices=Types.choices,
        default=Types.EATER,
    )

    # ...

    def save(self, *args, **kwargs):
        # 如果是新用户，就根据 base_type 属性设置用户类型
        if not self.pk:
            self.type = self.base_type
        return super().save(*args, **kwargs)
```

然后，在 users/models.py 模块里、User model 下方，再加入下面这些内容：

示例 22.9：加入 Inventor Proxy Model

Code (python)

```python
class InventorManager(BaseUserManager):
    def get_queryset(self, *args, **kwargs):
        results = super().get_queryset(*args, **kwargs)
        return results.filter(type=User.Types.INVENTOR)


class Inventor(User):
    # 这会在创建记录时把用户类型设为 INVENTOR
    base_type = User.Types.INVENTOR

    # 确保对 Inventor model 的查询只返回 Inventors
    objects = InventorManager()

    # 把 proxy 设为 True 表示不会为这条记录创建新表
    class Meta:
        proxy = True

    # 只有发明家才能发明新口味!
    def invent(self):
        # 神奇的自定义逻辑写在这里
        return "Delicious!"
```

正如注释里提到的，proxy models 不会增加字段。它们真正带来的，是对同一个 model object 的另一层引用，在这层引用上我们可以挂自定义 managers、methods 和 properties。下面这些查询就能很好地体现这种方式的威力：

示例 22.10：展示 Proxy Models 的威力

Code (python)

```python
>>> from users.models import User, Inventor
>>> User.objects.count()  # 超过 3 亿用户!
>>> Inventor.objects.count()  # 但只有 3 个发明家
>>> # 同一个人，既当作 User 调，也当作 Inventor 调
>>> user = User.objects.get(username='umafeldroy')
>>> user
<User: uma>
>>> inventor = Inventor.objects.get(username='umafeldroy')
>>> inventor
<Inventor: uma>
>>> # 调用一个只有 inventors 才有的方法
>>> user.invent()
AttributeError
>>> inventor.invent()
'Delicious'
```

这种 proxy 做法，能让我们在不创建新 User 表、也不把 `django.contrib.auth` 改造到第三方库都没法用的程度下，支持多种用户类型。

而且，由于 proxy models 可以有自己的 model managers，我们就可以写出更明确的查询。这有助于减少授权时犯错的机会。比较下面两种写法的清晰度：

示例 22.11：显式过滤出 Inventor 类型

Code (python)

```
>>> User.objects.filter(type=User.Types.INVENTOR)
>>> Inventor.objects.filter()  # 我们更偏爱这一种
```

22.3.3 增加额外数据字段

对于不同用户类型的额外数据字段，我们发现主要有两种处理方式：

1. 使用 `OneToOneField` 关联到 profile models，就像 22.2.3 节《方案三：从关联 Model 反向挂回去》里那样。
2. 把所有字段都直接塞进基础 `User` model。这个方案很简单，但一旦用户很多、而且各类用户专属数据也很多，`User` 表就可能开始变慢。哪怕先不谈性能，也会有很大的风险：把不该属于某类用户的数据，挂到了错误的用户类型上。

我们的偏好是第一种，也就是从关联 model 反向挂回去，并和 proxy models 结合使用。下面是我们如何把 22.2.3 节中所有“Profile”models 和 proxy model 结合起来使用：

Code (python)

```
class Inventor(User):
    # ...
    objects = InventorManager()

    class Meta:
        proxy = True

    @property
    def extra(self):
        return self.inventorprofile


class Scooper(User):
    # ...
    objects = ScooperManager()

    class Meta:
        proxy = True

    @property
    def extra(self):
        return self.scooperprofile


class Eater(User):
    # ...
    objects = EaterManager()

    class Meta:
        proxy = True

    @property
```

```
    def extra(self):
        return self.eaterprofile
```

这样一来，不管是哪种用户类型，来自 One-to-One 关系的 Profile 都可以统一通过 `extra` 属性访问：

- `eater.extra.favorite_ice_cream`
- `scooper.extra.scoops_scooped`
- `inventor.extra.flavors_invented`

我们很喜欢这种方式，因为它非常容易记：通往 profile 表的关系统一挂在 `extra` 属性上。哪怕系统允许用户拥有多个角色，这种方式也一样能工作。

Proxy models 通常应该谨慎使用，因为一旦实现复杂起来，很容易让人困惑。不过在像“多种用户类型”这样的合适场景下，它们又确实非常有用。

22.3.4 关于多种用户类型的补充资源

- Daniel Feldroy 关于这个主题的 YouTube 教程：
  `feld.to/multiple-user-types`
- Proxy Models 官方文档：
  `docs.djangoproject.com/en/3.2/topics/db/models/#proxy-models`
- Vitor Freitas 在“Simple is better than Complex”上写的多用户类型文章。即便他没有使用 proxy models，这篇文章也非常值得一读：
  `feld.to/2VSi3My`

22.4 总结

本章我们讨论了寻找 User model 的新方法，以及如何定义自己的自定义 user models。根据项目需求不同，你既可以继续沿用默认方式，也可以定制真正的 user model。我们还讨论了如何通过 proxy models 处理多种用户类型这一用例。

下一章我们将深入第三方 packages 的世界。

第二十三章 Django 的秘诀：第三方 Packages

TODO - Add mention of https://www.python.org/dev/peps/pep-0517/

Django 真正的威力，并不只来自框架本身，以及 djangoproject.com 上的官方文档。它更来自开源社区提供的海量第三方 Django 与 Python packages。可用于 Django 项目的第三方包多得惊人，而且它们能替你完成非常大量的工作。这些 packages 出自各行各业的人之手，而今天这个世界的很多系统，正是靠它们在驱动。

图 23.1: 图 23.1：一罐象征 Django 成功秘诀的神秘酱料。大多数人根本不知道这是什么。

专业级的 Django 与 Python 开发，很大一部分其实就是把第三方 packages 融入 Django 项目。如果你试图把自己需要的每一个工具都从零手写出来，那你会很难真正把事情做完。

对我们这种做咨询项目的人来说，这一点尤其真实，因为客户项目往往都由许多相同或相似的基础构件组成。

23.1 第三方 Packages 的例子

本书在很多地方都提到过各种第三方 packages。这份清单本身，就是一个很好的起点，适合那些正在寻找“值得考虑纳入项目”的高价值 packages 的人。

要注意，并不是这些包里每一个都专属于 Django。这意味着，其中有一些同样也可以在其他 Python 项目里使用。（通常来说，Django 专属包的名字会以 `django-` 或 `dj-` 开头，但也有不少例外。）

23.2 了解 Python Package Index

Python Package Index（PyPI）位于 `pypi.org/`，是 Python 编程语言的软件仓库。在写下这句话的时候，它已经收录了超过十万个 packages，其中也包括 Django 自己。

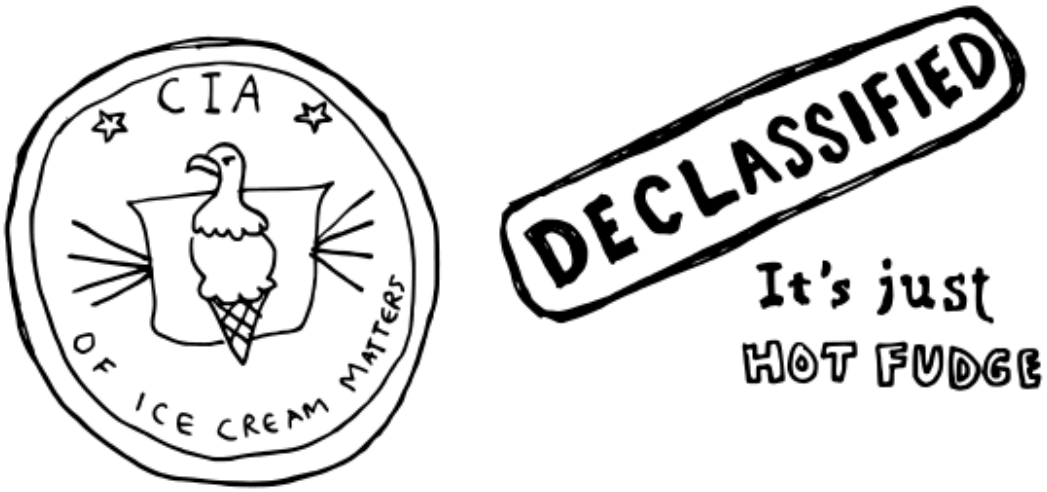

图 23.2: 图 23.2：秘密揭晓了。其实就是热巧克力酱。

对绝大多数 Python 社区成员来说，一个开源项目如果还没发布到 Python Package Index，就还不能算真正的“正式发布”。

Python Package Index 远不只是一个目录而已。把它看成世界上最大的 Python package 信息与文件中心会更贴切。每当你用 `pip` 安装某个特定版本的 Django 时，`pip` 实际上就是从 Python Package Index 下载那些文件。绝大多数 Python 和 Django packages，除了能通过 `pip` 安装外，也都可以直接从 Python Package Index 下载。

23.3 了解 DjangoPackages.org

Django Packages（`djangopackages.org`）是一个面向 Django 项目的可复用 apps、sites、tools 等资源目录。和 PyPI 不同，它并不存放 package 本身，而是提供一套混合信息：既包括从 Python Package Index、GitHub、ReadTheDocs 收集来的硬指标，也包括用户手动录入的“软数据”。

Django Packages 最著名的用途，是作为一个比较站点来评估 packages 的特性。在 Django Packages 上，packages 会被组织成非常方便的 grids，从而可以彼此对照比较。

顺带一提，Django Packages 恰好也是本书作者创建的项目之一，当然它也得到了 Python 社区中许许多多人的贡献。正因为有众多志愿者持续维护和改进，它才能一直作为对 Django 用户有帮助的资源存在下去。

23.4 了解你的资源

如果一个 Django 开发者不了解 Django Packages 和 Python Package Index 这两项关键资源，那他其实是在主动放弃使用 Django 与 Python 最重要的一项优势。

如果你还不了解这些工具，那么花时间去熟悉它们，绝对值得。

作为 Django 和 Python 开发者，你应该把“尽可能优先使用第三方库，而不是重复造轮子”当成自己的使命。最优秀的库，是由世界各地能力惊人的开发者编写、文档化并测试出来的。站在这些巨人的肩膀上，往往就是“惊人成果”和“悲剧式翻车”之间的差别。

在使用各种 packages 的过程中，去研究并学习它们的代码。你会从中学到许多模式和技巧，它们会让你成为更好的开发者。

但另一方面，识别“好包”和“坏包”的能力也非常重要。花时间像评估自己作品一样去评估别人的 package，是完全值得的。关于这一点，我们会在本章后面 23.10 节《什么样的 Django Package 才算好》里继续讨论。

23.5 安装与管理 Packages 的工具

如果你想真正用好这些可用于项目的 packages，那么安装好 `virtualenv` 与 `pip`（或者 Conda、Poetry）绝不是可以跳过的小事。这是硬性要求。

更多细节见第 2 章《最佳 Django 环境搭建》。

23.6 包依赖要求

正如我们在第 5 章《Settings 与 Requirements Files》中提到过的，我们使用 requirements files 来管理 Django / Python 依赖。这些文件位于项目根目录下的 `requirements/` 目录中。

23.7 接入 Django Packages：基础流程

当你找到一个想使用的第三方 package 时，按下面的步骤来。

23.7.1 第一步：阅读该 Package 的文档

你确定真的要用它吗？在安装任何 package 之前，先确保自己知道将要面对什么。

23.7.2 第二步：把 Package 及其版本号加入 Requirements

如果你还记得第 5 章《Settings 与 Requirements Files》，一个 `requirements/base.txt` 文件通常会长这样（当然，你自己的大概率会更长）：

示例 23.1：把带版本号的 Packages 加入 Requirements

Code (text)

```
Django==1.11
coverage==4.3.4
django-extensions==1.7.6
django-braces==1.11
```

注意，每个 package 都被钉死到了一个具体版本号。永远要把 package 依赖固定到明确版本。

如果你不固定依赖版本，会发生什么？几乎可以肯定，在未来某个时刻，当你尝试重新安装项目，或者修改 Django 项目时，你一定会撞上问题。因为一旦 package 发布了新版本，你根本不能指望它永远向后兼容。

说个让人难受的真实案例：有一次我们照着某个 SaaS 平台的说明去使用他们家的库。由于他们自己并没有官方 Python 客户端，而只是有一个早期采用者在 GitHub 上放了个可用实现，所以那份说明让我们把下面这行写进 `requirements/base.txt`：

示例 23.2：错误的 Requirements 写法

Code (text)

```
-e git+https://github.com/erly-adptr/py-junk.git#egg=py-jnk
```

这是我们的错。我们当时就该更谨慎一点，把它固定到某个具体的 git revision number。

这完全不是那个早期采用者的错，但他往自己的 repo 推了一个坏掉的 commit。后来我们需要非常快地修一个线上 bug，于是先在本地开发环境里写了修复并跑了测试，测试通过了。接着我们部署到生产环境，而部署流程会抓取所有依赖更新；不幸的是，那个坏掉的 commit 被当成了一次合法变更。

结果就是：我们在修一个 bug 的同时，把整个站点打崩了。

那可真不是美好的一天。

之所以要使用固定版本发布，本质上是在给我们自己发布出去的工作增加一点形式化约束和流程。尤其在 Python 世界里，GitHub 以及其他代码仓库，是开发者发布进行中代码的地方，而不是我们构建生产级项目时应该依赖的最终稳定版本来源。

还有一点：在固定依赖时，尽量连“依赖的依赖”也一起固定。这样部署和测试的可预测性会强得多。

23.7.3 第三步：把 Requirements 安装进你的 Virtualenv

假设你已经进入一个可用的 virtualenv，并且当前位于项目的 `<repo_root>`，那么接下来就用 `pip` 安装与你环境相匹配的 requirements file，比如 `requirements/dev.txt`。

如果这是你第一次在某个 virtualenv 里这么做，那么它需要一些时间去抓取所有依赖并完成安装。

23.7.4 第四步：严格按照 Package 的安装说明来

除非你对这个 package 已经非常熟悉，否则要克制住“跳步骤”的冲动。开源 Django package 作者通常都很重视文档，而且很乐意让别人来使用他们的包，所以大多数时候，他们写出来的安装说明都足够清楚，能让你顺利把东西跑起来。

23.8 排查第三方 Packages 的问题

有时候你在配置某个 package 时就是会遇到问题。那怎么办？

首先，认真努力地自己定位并解决问题。反复读文档，确认自己没有漏掉任何步骤。上网搜搜看，别人是不是也遇到过一样的问题。愿意卷起袖子去读 package 的源码，因为你也许真的发现了一个 bug。

如果看起来像是 bug，就去 package 仓库的 issue tracker 里看看有没有人已经报告过。有时候你能在那里找到 workaround 或修复办法。如果这是个还没人提过的 bug，那就自己去提。

如果你还是卡住了，就去那些常规渠道求助：StackOverflow、IRC 的 `#django`、项目自己的 IRC 频道（如果它有的话），以及你本地的 Python 用户组。尽可能清楚描述问题，并提供尽可能多的上下文信息。

23.9 发布你自己的 Django Packages

每当你写出了一个特别有用的 Django app，都应该认真考虑把它打包出来，以便在其他项目中复用。

最好的入门方式，是先跟着 Django 的高级教程《How to Write Reusable Apps》把基础打好：`docs.djangoproject.com/en/3`

除此之外，我们还建议你：

- 创建一个公开 repo 来放代码。如今大多数 Django packages 都托管在 GitHub 上，因此在那里最容易吸引贡献者，当然也还有其他选择，比如 GitLab、Bitbucket、Assembla 等。
- 把 package 发布到 Python Package Index（`pypi.org`）。上传说明见：
  `packaging.python.org/distributing/#uploading-your-project-to-pypi`
- 把 package 加入 Django Packages（`djangopackages.org`）。
- 使用 Read the Docs（`readthedocs.io`）或类似服务来托管你的 Sphinx 文档。

提示：我应该把公开 Repo 建在哪里？

有很多网站为开源项目提供免费的源码托管和版本控制服务。正如第 2 章《最佳 Django 环境搭建》中提过的，GitHub 和 GitLab 是两个常见选择。

如果你考虑别的托管版本控制服务，要记住：`pip` 只支持 Git、Mercurial、Bazaar 和 Subversion。

23.10 什么样的 Django Package 才算好?

下面是一份清单，你可以在创建新的开源 Django package 时用来自检。本节很多内容同样适用于那些并不专属于 Django 的 Python packages。这份清单也同样适合在你评估某个 Django package 是否值得引入项目时使用。

23.10.1 目标

你的 package 应该做一件真正有用的事，而且把它做好。名字也应该清晰可描述。package repo 的根目录最好带上 `django-` 或 `dj-` 前缀，这样更容易被找到。

如果 package 的部分功能，其实可以由一个不依赖 Django 的相关 Python package 来完成，那么就把它拆成一个单独的 Python package，并让 Django package 依赖它。

23.10.2 范围

你的 package 的职责范围应该高度聚焦在一个小任务上。这样一来，它的应用逻辑会更紧凑，而用户也更容易修补或替换它。

23.10.3 文档

没有文档的 package，只能算 pre-alpha package。仅仅写 docstrings，并不能算真正的文档。

正如第 25 章《Documentation: Be Obsessed》里讲的，文档应该用 Markdown 编写。你还应该用 MkDocs、Sphinx 或其他工具生成一份排版良好的文档版本，并公开托管。我们鼓励你使用 `readthedocs.io` 并配置 webhooks，这样每次你修改文档后，在线文档都能自动更新。

如果 package 有依赖，它们应当被写进文档。package 的安装说明也应当被文档化，而且安装步骤必须足够可靠、足够防呆。

23.10.4 测试

你的 package 应该有测试。测试会提升可靠性，降低升级 Python / Django 版本时的成本，也能让其他人更容易有效地贡献代码。

同时，写清楚如何运行 package 的测试套件。如果你自己，或者任何贡献者，都能在提交 pull request 之前轻松跑测试，那么你最终就更有可能收到质量更高的贡献。

23.10.5 模板

过去，有些 Django packages 会在文档中告诉用户“自己创建 templates”，而不是实际附带模板文件。但如今，Django packages 通常都会自带一组极简模板，用来演示最基础的功能，这已经变得相当标准。

通常这些模板只包含最少量的 HTML、必要的 JavaScript，而不带 CSS。唯一的例外，是那些 widget 本身就需要 CSS 才能工作的 packages。

23.10.6 活跃度

如果有需要，你的 package 应该能从你或其他贡献者那里得到定期更新。当你更新 repo 中的代码时，就应该考虑把一个 minor 或 major release 上传到 Python Package Index。

23.10.7 社区

优秀的开源 packages，包括 Django 生态中的那些，往往最后都会收到其他开发者的贡献。所有贡献者都应该在 `CONTRIBUTORS.rst` 或 `AUTHORS.rst` 这样的文件中获得署名。

如果你的 package 已经有贡献者或分叉，那么你就要积极扮演社区领导者的角色。如果别的开发者 fork 了你的 package，就关注他们在做什么。想一想，他们的部分工作，甚至全部工作，是否值得合并回你的版本。

如果这个 package 的功能已经和你原本想做的事情偏离很多，那就谦逊一点，认真考虑建议对方给自己的 fork 起个新名字。

23.10.8 模块化

你的 package 应该尽可能容易插入到任何一个“没有把核心组件（templates、ORM 等）替换掉”的 Django 项目中。安装过程应该尽量少侵入。

当然，也别把 modularity 和 over-engineering 混为一谈。

23.10.9 在 PyPI 上可用

你的 package 的所有 major 和 minor releases，都应该能从 Python Package Index 下载。想用你 package 的开发者，不应该被迫去你的 repo 里翻一个“勉强能用”的版本。版本号也应该像下一节说的那样，保持规范。

23.10.10 尽可能使用最宽的 Requirements Specifiers

你的第三方 package 应该在 `setup.py` 的 `install_requires` 参数里，用尽可能宽松的范围来声明依赖的其他库。

下面这种 requirements 定义方式非常糟糕：

示例 23.3：过窄的 Package Requirements

Code (python)

```
# DON'T DO THIS!
# requirements for django-blarg

Django==3.0
requests==2.11.0
```

问题出在依赖图。每隔一阵子，你死死钉在某个特定 Django 版本、或者其他库版本上的依赖，就会在别人的项目里制造麻烦。

例如，如果 `icecreamratings.com` 是我们的网站，而它的部署 requirements.txt 文件长这样，同时我们又安装了 `django-blarg`，会发生什么?

示例 23.4：`icecreamratings.com` 的 `requirements.txt`

Code (python)

```
# requirements.txt for the mythical web site 'icecreamratings.com'
Django==3.1
requests==2.13.0
django-blarg==1.0

# Note that unlike the django-blarg library, we explicitly pin
# the requirements so we have total control over the environment
```

如果把前面那个坏示例安装到一个像示例 23.4 这样要求 Django 3.1 的项目里，那么 `django-blarg` 在安装时就会把 Django 3.1 覆盖回 Django 3.0。考虑到 Django 3.0 和 3.1 之间本来就有一些向后不兼容点，`django-blarg` 很可能会直接让 `icecreamratings.com` 整站开始返回 HTTP 500。

因此，你的第三方 package 应该用尽可能宽的范围来声明依赖：

示例 23.5：宽松定义的 Package Requirements

Code (python)

```
# requirements for django-blarg

Django>=3.1,<3.0
requests>=2.13.0,<=3.0.0
```

更多阅读：

- `pip.pypa.io/en/stable/reference/pip_install/#requirement-specifiers`
- `nvie.com/posts/pin-your-packages/`

23.10.11 合理的版本号

和 Django、Python 一样，我们更倾向于遵循严格版的 PEP 386 命名规范。实际上我们用的是 `A.B.C` 模式。下面分别说明。

`A` 表示 major version number。只有当发生大型变更、并且会打破上一 major version 向后兼容性时，才应该增加这一位。不同大版本之间出现大规模 API 变化，并不罕见。

`B` 表示 minor version number。它的增长通常对应较小的破坏性变更，或者是在为即将到来的变更提供 deprecation notices。

`C` 表示 bug fix release，纯粹派会把这一位叫作 micro release。很多开发者在尝试某个已有项目的最新 major 或 minor 版本前，都会等它先出现至少一个这种级别的修复版本。

对于 alpha、beta 或 release candidate，一般会把这些信息作为后缀附在即将发布的版本号后面。例如：

- `Django 3.2-alpha`
- `django-crispy-forms 1.9.1-beta`

警告：不要把未完成代码上传到 PyPI

PyPI，也就是 Python Package Index，应该是一个能让人拿到可靠、稳定 packages 来构建 Python 项目的地方。PyPI 不是放 Alpha、Beta 或 Release Candidate 代码的地方，尤其是在 `pip` 以及其他工具默认会抓取“最新发布版”的情况下更是如此。

善待其他开发者，只把真正正式的 releases 放到 PyPI 上。

补充说明：虽然现代版本的 `pip` 默认已经不会安装 pre-releases，但你不能指望所有使用你代码的人都已经安装了最新版 `pip`。

更多阅读：

- `python.org/dev/peps/pep-0386`
- `semver.org`

23.10.12 名称

项目名称至关重要。一个命名良好的项目，容易被发现，也容易被记住；而一个糟糕的名字，会把自己藏起来，吓退某些开发团队，甚至还可能直接阻止它被列进 PyPI、Django Packages 以及其他资源站。

我们在 4.2 节《给 Django Apps 起什么名字》里已经讲过一些基础，不过这里再补几条专门适用于开源 Django packages 的建议：

- 先检查这个名字有没有已经在 PyPI 上注册。否则它就没法直接用 `pip` 轻松安装。

- 再检查这个名字是不是已经在 Django Packages 上存在。这一点只针对 Django packages。
- 不要使用带有脏话或冒犯意味的名字。你也许觉得有趣，但对其他人并不友好。曾经就有位知名开发者写过一个库，结果 NASA 在他改名之前根本不能使用它。

23.10.13 许可证

你的 package 必须带许可证。对个人开发者来说，最好使用 MIT 许可证，因为它通常足够宽松，能覆盖大多数商业与非商业用途。如果你对专利问题更敏感，那么就用 Apache license。

在 repo 根目录创建一个 `LICENSE.rst` 文件，把许可证名写在顶部，然后把来自 OSI 批准清单的对应文本贴进去。可以从 `choosealicense.com` 获取。

提示：许可证保护的是你，也保护整个世界

在这个随意诉讼和专利流氓横行的时代，给软件加许可证，绝不只是“保护你对代码的所有权”这么简单，它远不止于此。

如果你不给代码加许可证，或者使用一个没有被专业律师审核过的非标准许可证，那么你的作品就有可能被专利流氓拿去当武器；又或者在某些金融或医疗事故中，你自己也有可能被追责。

OSI 批准的许可证通常都会包含几段非常关键的声明：关于版权、再分发、免责声明，以及责任限制。

23.10.14 代码的清晰度

你的 Django package 代码应该尽可能清晰、尽可能简单。不要使用那些奇怪、反常的 Python / Django hack，除非你清楚解释自己在做什么。

23.10.15 使用 URL Namespaces

这一点在 8.4 节《使用 URL Namespaces》中已经讲过。URL namespaces 能带来更好的互操作性。使用它们，就更容易处理项目之间的冲突，甚至还能提前为潜在冲突做好准备。

如果担心未来发生冲突，也可以实现一套基于 settings 的 URL namespace 系统。也就是让项目用 settings 来定义 URL namespace，再额外提供一个 Django context processor 和详细使用说明。实现这件事本身并不难，但它确实会引入一层额外抽象，从而让项目稍微更难维护一点。

23.11 更轻松地创建你自己的 Packages

发布自己写的一小段代码，会是一种非常有成就感的体验。每个人都应该试试看！

不过，把所有零件拼装好，从而做出一个真正可复用的 Django package，工作量其实很大，而且非常容易哪里弄错。幸运的是，Cookiecutter 可以大大简化这件事。

包提示：Cookiecutter，让项目模板变得简单

2013 年，Audrey 创建了很受欢迎的 Cookiecutter 工具，用来生成项目模板。它非常好用，也非常强大。Python 和 Django packages 都有许多现成模板。更棒的是，如今很多 IDE，例如 PyCharm 和 Visual Studio Code，也已经支持基于 Cookiecutter 的模板。

- `github.com/cookiecutter/cookiecutter`
- `cookiecutter.readthedocs.io`

对于下面提到的 Cookiecutter 模板，我们已经非常积极地请 Django 和 Python 社区的领军人物帮忙做过评审。你只需要在命令行里运行下面这个 bash 示例：

示例 23.6：使用 Cookiecutter 快速起步 Packages

Code (bash)

```
# 只有当你还没安装 cookiecutter 时才需要
$ pip install cookiecutter
```

```
# 从零创建一个 Django Package
$ cookiecutter https://github.com/pydanny/cookiecutter-djangopackage.git

# 从零创建一个 Python Package
$ cookiecutter https://github.com//ionelmc/cookiecutter-pylibrary.git
```

接着它会提示你填写一些信息。最终生成的结果，会是一套基础 Django / Python / 等等 package 模板实现，其中包括代码、文档、测试、许可证，以及更多其他东西。

23.12 维护你的开源 Package

警告：开源倦怠，以及付出过度

除非你是拿着工资、以专业方式去做开源工作，否则请记住，这本质上是为了乐趣而做的志愿劳动。量力而行，按自己的节奏来，尽力就好。

你创建出来的开源 packages，会慢慢拥有自己的生命。它们会随着时间成熟，也会随着需求和开发标准的演进而变化。下面是一些在维护开源项目时我们应该做的事。

23.12.1 为 Pull Requests 署名

当有人提交的 pull request 被接受后，要善待他们。务必把贡献者名字加入项目作者文档，比如 `CONTRIBUTORS.md` 或 `AUTHORS.md`。

23.12.2 如何处理糟糕的 Pull Requests

有时候你就是不得不拒绝某些 pull requests。拒绝时也请尽量友善、积极，因为一个处理得好的“被拒绝 PR”，也可能换来一个终生朋友。

下面这些类型的 pull requests，应该认真考虑直接拒绝：

- 任何测试没过的 pull request。要求对方先修好。详见第 24 章《Testing Stinks and Is a Waste of Money!》。
- 任何新增代码却让测试覆盖率下降的 pull request。同样，参见第 24 章。
- Pull request 应该只改最少、最聚焦的内容。那种大而全、横扫一片的变更，应该拒绝，并在评论中要求对方拆成更小、更原子的 pull requests。
- 过于复杂的代码提交要谨慎对待。要求对方简化、补注释，甚至直接拒绝过度复杂的 pull request，本身都没有问题。
- 不符合 PEP 8 的代码，需要让对方重新提交。Django 世界非常看重 PEP 8，你的项目也应该如此。违反 PEP 8 的提交完全可以要求改进。
- 把代码修改和大规模空白字符清理混在一起的提交。如果有人只改了两行逻辑代码，却顺手修了 200 行空格，那这个 pull request 的 diff 在功能上就几乎不可读了，应当拒绝。空白字符清理应该单独开一个 pull request。

警告：不要把代码修改和大规模空白清理混在一起

我们之所以专门再强调一次，是因为这几乎可以被视为一种由第三方引入的代码混淆。甚至可以说，这潜在上就是一种安全风险。毕竟，想偷偷塞入恶意代码，还有什么比借着一个 pull request 顺手混进去更合适的办法呢？

23.12.3 正式发布 PyPI 版本

在 Python 社区里，如果一个关键开源项目的 PyPI 版本长期不更新，却逼着开发者去依赖一个“稳定”的 master 或 trunk 分支，这会被视为不负责任。

这会带来很多问题，因为开源代码仓库本来就不应被视作生产质量代码的可靠来源。比如，到底应该用哪个具体 commit？哪个 tag 才算对？相比之下，PyPI 是一个公认的、专门用来安全提供可安装 package 的地方。

在 Python 世界里，被接受的最佳实践是：只要 trunk 或 master 上出现了重要变更，甚至哪怕只是小型 bug fixes，都应该发布版本。事实上，小型 bug fix releases 本来就是任何持续软件项目的一部分，除了某些美国政府 IT 合同环境之外，基本不会有人为此责怪你。

如果你还不太确定这件事应该怎么运作，建议去看看 `python-requests` 的变更历史。它是 Python 世界最流行的项目之一：`github.com/psf/requests/blob/master/HISTORY.md`

要创建并上传你的发行包，可以按下面步骤做：

示例 23.7：使用 Twine 上传 Package Distributions

Code (bash)

```
pip install twine
python setup.py sdist
twine upload dist/*
```

包提示：Twine 是什么？

Twine 是上传 package 到 PyPI 的首选库。`python setup.py` 的问题在于：它会通过不安全的连接发送文件，让你的库暴露在中间人攻击面前。相比之下，Twine 只通过经过验证的 TLS 来上传 package。

而且还不止如此。Twine 在上传 Wheels 时也更好用（下一小节会说），它不需要执行 `setup.py`，甚至还能预签名你的 releases。如果你真的很看重安全，那它就是首选工具。

23.12.4 创建并发布 Wheels 到 PyPI

根据 PEP 427，Wheel 是 Python 发行格式的新标准。它的目标是替代 eggs，并提供一系列优势，包括更快的安装速度，以及支持安全数字签名。`pip >= 1.4` 和 `setuptools >= 0.8` 都已经支持它。

示例 23.8：安装 Wheel

Code (bash)

```
pip install wheel
```

然后，在你把 package 发布到 PyPI 之后，运行下面这些命令：

示例 23.9：创建 Wheel Distributions 并上传

Code (bash)

```
python setup.py bdist_wheel
twine upload dist/*
```

当一个可选的 `setup.cfg` 文件和 `setup.py` 处于同一级目录，并包含下面这段配置时，Twine 可以生成 universal wheels：

示例 23.10：配置 Universal Wheels

Code (ini)

```
# setup.cfg
[wheel]
universal = 1
```

Wheel 相关资源：

- 规范：PEP 427 python.org/dev/peps/pep-0427
- PyPI 上的 Wheel Package：https://feld.to/3eTOdyS/project/wheel
- 文档：wheel.readthedocs.io
- 倡导站点：pythonwheels.com

23.12.5 给 Repo 加 Git Tags

我们还喜欢给 releases 打上 git tag。这相当于记录了一份“提交到 PyPI 时的代码快照”（见上一小节）。它除了能作为历史记录，还可以直接用来安装 package。例如，如果某个组织不想自建私有 package server，那么它就可以依赖 git tags，或者依赖像 GitHub 这种 repo host 提供的 releases 功能。

示例 23.11：创建并推送 Tag 到 Repo

Code (bash)

```
git tag -a v1.4 -m "my version 1.4"
git push origin v1.4
```

参考：git-scm.com/book/en/v2/Git-Basics-Tagging

23.12.6 把 Package 升级到新版本 Django

时不时地，Django 会发布一个 minor release。大约每年还会有一次 major release。每当这种时候发生，非常重要的一件事就是：在包含最新 Django 版本的 virtualenv 里，把你的 package 测试套件完整跑一遍。

哪怕没有别的理由，这本身也足以说明：为什么项目里必须带测试。

23.12.7 遵循良好的安全实践

我们会在第 28 章《Security Best Practices》中深入讨论安全问题。不过，Django、Python 和 PyPy 的核心开发者 Alex Gaynor 还写过一篇非常有价值的文章，适用于任何开源项目的维护者：alexgaynor.net/2013/oct/19/security-proces

提示：Alex Gaynor 谈开源项目安全

“安全漏洞会让你的用户陷入风险，而进一步地，也常常会让他们的用户陷入风险。作为软件的作者和分发者，你有责任以最有可能帮助用户避免被利用的方式来处理安全发布。”

23.12.8 提供示例基础模板

对于使用你项目 views 的场景，始终都应该附带一些基本模板。我们自己偏好写得极其简单的 HTML，或者使用 Bootstrap、Tailwind 这类常见前端框架。这样一来，那些正在考虑是否用你的项目来解决问题的开发者，就能更容易“试驾”这个项目。

当然，他们最终大概率还是会在自己项目的 templates/ 目录里改写这些模板，但起步阶段这样会省心太多。

另外，也请记得附带一个 templates/myapp/base.html，以增强互操作性。关于这一点，你可以在 cookiecutter-djangop 中看到说明与示例：bit.ly/2onSzCV

23.12.9 把 Package 交出去

有时候，人生会把你从一个 package 的维护工作里带走。可能是家庭，也可能是一份新工作，又或者只是因为你自己已经不再需要那个开源项目了。时间上的现实，也可能意味着你再也没法及时 review pull requests，或继续探索新功能点。对于项目创建者来说，把一个项目放手，常常是极其困难的。

但如果你把项目交给活跃的维护者，它就可能被重新激活，变得比以前更有用。这件事也会为你赢得更广泛开发者社区的尊重。

Django 和 Python 社区里一些比较知名的“交棒”案例包括：

- Ian Bicking 与 `pip` / `virtualenv`
- Daniel 和 Audrey Feldroy 与 `djangopackages.org`
- Daniel Feldroy 与 `django-uni-form`、`dj-stripe` 和 `django-mongonaut`
- Audrey Feldroy 与 Cookiecutter
- Rob Hudson 与 `django-debug-toolbar`

23.13 补充阅读

下面这些文章，对任何正在贡献、创建或维护开源库的人都很有帮助：

- `djangoappschecklist.com`

  这是我们非常喜欢的一个站点，Django Apps Checklist 几乎涵盖了本章的一切要点。
- `alexgaynor.net/2013/sep/26/effective-code-review`
- `hynek.me/articles/sharing-your-labor-of-love-pypi-quick-and-dirty`

23.14 总结

Django 真正的威力，就在于有海量第三方 packages 可供你在 Django 项目里直接使用。

确保你已经安装并掌握了 `pip` 和 `virtualenv`（或者可选的 Poetry），因为它们是你以可管理方式在系统中安装 packages 的最佳工具。

主动去了解现有 packages。Python Package Index 和 Django Packages，都是查找 package 信息的绝佳起点。

在评估一个 Django package 时，成熟度、文档、测试和代码质量，都是很好的起步标准。

安装稳定 package，是大项目和小项目共同的基础。真正会用 packages，就意味着要坚持使用特定 release，而不是直接依赖某个项目的 trunk 或 master。假如真的没有正式 release，那至少也该依赖一个明确 commit。修 package 与你项目之间的兼容问题，确实需要耐心和时间；但如果你卡住了，别忘了去求助。

我们还讨论了如何创建自己的第三方 package，并给出了如何借助 Cookiecutter 快速起步、最终把东西发布到 Python Package Index 的基础说明。我们也顺带讲到了新的 Wheel 格式。

最后，我们也给出了一些关于如何维护 package 的建议。

第二十四章 测试真烂，还浪费钱！

好了，总算把你骗进这一章了。

现在你得把它读完。

我们会尽量让这一章有意思一点。

24.1 测试能保住金钱、工作，甚至生命

Daniel 的故事：你听过 `smoke test` 这个词吗？

NASA 的管理与项目分析师 Gretchen Davidian 曾告诉我，在她还是工程师的时候，作为测试人员，她的工作就是把那些准备上天的设备放进极其严苛的环境里，直到它们开始冒烟，最后甚至真的起火。

听起来很刺激吧！工作、金钱和生命都押在上面。而考虑到 Gretchen 对细节的关注程度，我敢肯定她确实烧掉过不少硬件。

要记住，对我们很多软件工程师来说，押上的风险和 NASA 并没有本质区别。我还记得 2004 年我在一家私营公司工作时，仅仅一次 miles 和 kilometers 搞错的失误，就在几个小时内让公司损失了几十万美元。Quality Assurance（QA）团队成员因此失去了工作，也就失去了收入和医疗福利。换句话说，如果没有足够的测试，丢掉的可能是工作、金钱，甚至生命。虽然 QA 团队非常敬业，但当时一切都只能靠人工逐个点击项目来完成，人的失误自然会混进测试流程。

今天，随着 Django 被用到越来越广的应用场景里，自动化测试的重要性，丝毫不亚于当年 Gretchen 在 NASA 的工作，也不亚于 2004 年那支可怜 QA 团队所面对的处境。如今 Django 被用在下列这些同样有严格质量要求的场景里：

- 处理医疗信息的应用。
- 为有需要的人提供关乎生命的关键资源的应用。
- 处理用户金融信息的应用。

包提示：测试 Django 项目的实用库

我们很喜欢用 `coverage.py`。

这个工具能清楚地告诉你，代码库里哪些部分已经被测试覆盖，哪些行还从来没被碰到。它还会顺手给你一个很方便的百分比，显示代码总体覆盖率。即便 100% 覆盖率也不能保证应用绝对没有 bug，但它确实很有帮助。

我们想感谢 Ned Batchelder 在维护 `coverage.py` 上做出的惊人工作。它是个极好的项目，对任何和 Python 有关的工程都很有用。

24.2 如何组织测试

假设我们刚创建了一个新的 Django app。我们做的第一件事，就是把 `python manage.py startapp` 默认生成、但基本没什么用的 `tests.py` 模块删掉。取而代之，我们会创建一个名为 `tests` 的目录，并在里面放一个空的 `__init__.py`。

在这个新目录里，由于大多数 app 都会用到，我们会创建 `test_forms.py`、`test_models.py`、`test_views.py` 这几个模块。属于 forms 的测试放进 `test_forms.py`，model 测试放进 `test_models.py`，依此类推。

结构看起来是这样的：

示例 24.1：如何组织测试

Code (text)

```
popsicles/
    __init__.py
```

```
admin.py
forms.py
models.py
tests/
    __init__.py
    test_forms.py
    test_models.py
    test_views.py
views.py
```

另外，如果除了 forms.py、models.py 和 views.py 之外，还有别的文件也需要测试，我们就给它们创建对应的测试文件。

我们喜欢这种做法，因为虽然它确实多了一层目录嵌套，但另一种选择通常是让 app 里堆满一大堆很痛苦的模块，导航起来特别难受。

提示：测试模块一律用 test_ 前缀

始终给测试模块加上 test_ 前缀，这一点非常重要。否则 Django 的 test runner 就没法轻松发现你的测试文件。除此之外，这也是个很好的约定，在 IDE 和文本编辑器里浏览文件名时都会更灵活。和往常一样，不要把如何命名这种事交给你偏好的 IDE 来替你决定。

24.3 如何编写单元测试

程序员在写代码的那一刻，常常会觉得自己状态正好、手感正热。可等到几个月、几周、几天，甚至几小时后再回来看同一段代码时，他们又常常会觉得这代码质量不怎么样。

写单元测试这件事也是一样。

这些年里，我们逐渐形成了一套自己喜欢遵循的测试实践，其中当然也包括单元测试。我们的目标始终是：用最短的时间，写出最有意义的测试。因此有了下面这些原则。

24.3.1 每个测试方法只测一件事

测试方法测试的范围必须极窄。一个单独的单元测试，不应该同时断言多个 views、models、forms 的行为，甚至也不该同时覆盖一个类里的多个方法。相反，一个测试只应该断言一个 view、一个 model、一个 form、一个 method 或一个 function 的行为。

这里会有个两难问题。一个 view 往往依赖 models、forms、methods 和 functions，那要怎么只测试一个 view 呢？

诀窍在于：为某个特定测试构造环境时，务必要极简。如下例所示：

示例 24.2：只测试一件事

Code (python)

```
# flavors/tests/test_api.py
import json

from django.test import TestCase
from django.urls import reverse

from flavors.models import Flavor


class FlavorAPITests(TestCase):
```

```
    def setUp(self):
        Flavor.objects.get_or_create(
            title='A Title',
            slug='a-slug',
        )

    def test_list(self):
        url = reverse('flavors:flavor_object_api')
        response = self.client.get(url)
        self.assertEquals(response.status_code, 200)
        data = json.loads(response.content)
        self.assertEquals(len(data), 1)
```

这个测试来自 17.2 节《用一个简单 API 来说明设计概念》中的 API 示例代码测试。我们通过 `setUp()` 方法，只创建运行测试所需的最少记录数。

下面再给一个大得多的例子，它基于我们之前给出的 REST API 示例：

示例 24.3：测试 API 代码

Code (python)

```
# flavors/tests/test_api.py
import json

from django.test import TestCase
from django.urls import reverse

from flavors.models import Flavor


class DjangoRestFrameworkTests(TestCase):

    def setUp(self):
        Flavor.objects.get_or_create(title='title1', slug='slug1')
        Flavor.objects.get_or_create(title='title2', slug='slug2')
        self.create_read_url = reverse('flavors:flavor_rest_api')
        self.read_update_delete_url = reverse(
            'flavors:flavor_rest_api',
            kwargs={'slug': 'slug1'},
        )

    def test_list(self):
        response = self.client.get(self.create_read_url)

        # 内容里是否同时包含这两个标题?
        self.assertContains(response, 'title1')
        self.assertContains(response, 'title2')

    def test_detail(self):
        response = self.client.get(self.read_update_delete_url)
        data = json.loads(response.content)
```

```
        content = {
            'id': 1,
            'title': 'title1',
            'slug': 'slug1',
            'scoops_remaining': 0,
        }
        self.assertEquals(data, content)

    def test_create(self):
        post = {'title': 'title3', 'slug': 'slug3'}
        response = self.client.post(self.create_read_url, post)
        data = json.loads(response.content)
        self.assertEquals(response.status_code, 201)
        content = {
            'id': 3,
            'title': 'title3',
            'slug': 'slug3',
            'scoops_remaining': 0,
        }
        self.assertEquals(data, content)
        self.assertEquals(Flavor.objects.count(), 3)

    def test_delete(self):
        response = self.client.delete(self.read_update_delete_url)
        self.assertEquals(response.status_code, 204)
        self.assertEquals(Flavor.objects.count(), 1)
```

24.3.2 对 views，能用 Request Factory 时就尽量用

`django.test.client.RequestFactory` 提供了一种方式，可以生成一个 request 实例，并把它作为第一个参数传给任意 view。这比 Django 标准测试客户端提供了更高程度的隔离，但确实也要求测试编写者多做一点额外工作。

原因在于，请求工厂并不支持 middleware，包括 session 和 authentication。

参见 `docs.djangoproject.com/en/3.2/topics/testing/advanced/`

遗憾的是，官方文档没有覆盖这样一种情况：你想测试一个被单独某个 middleware 类包裹起来的 view。例如，如果一个 view 依赖 sessions，我们会这样写：

示例 24.4：如何把 Middleware 加到 Requests 和 Responses 上

Code (python)

```
from django.contrib.auth.models import AnonymousUser
from django.contrib.sessions.middleware import SessionMiddleware
from django.test import TestCase, RequestFactory

from .views import cheese_flavors


def add_middleware_to_request(request, middleware_class):
    middleware = middleware_class()
    middleware.process_request(request)
```

```
    return request


def add_middleware_to_response(request, middleware_class):
    middleware = middleware_class()
    middleware.process_response(request)
    return request


class SavoryIceCreamTest(TestCase):
    def setUp(self):
        # 每个测试都需要访问 request factory。
        self.factory = RequestFactory()

    def test_cheese_flavors(self):
        request = self.factory.get('/cheesy/broccoli/')
        request.user = AnonymousUser()

        # 给 request 对象补一个 session
        request = add_middleware_to_request(request, SessionMiddleware)
        request.session.save()

        # 处理并测试这个 request
        response = cheese_flavors(request)
        self.assertContains(response, 'bleah!')
```

24.3.3 不要写那种还得再被测试的测试

测试代码应该尽可能简单。如果测试本身的代码，或者为运行测试而调用的辅助代码，看起来很复杂、抽象层很多，那我们就已经有问题了。事实上，我们自己过去也犯过错，写过过于复杂的测试工具函数，复杂到那些工具函数还得再给自己写测试。可以想象，这会让真正测试逻辑的调试过程变成一场噩梦。

24.3.4 Don’t Repeat Yourself 不适用于测试

`setUp()` 方法对于在同一个测试类的所有测试方法之间生成可复用数据，确实很有用。不过有时不同测试方法需要的数据只是相似、却并不相同，这时我们就很容易掉进陷阱，开始写一些花哨的测试工具函数。更糟的是，我们可能会觉得，与其写 20 个相似的测试，不如写一个大而全的方法，传不同参数进去，由它替我们把所有事情都做完。

我们最喜欢的处理方式，其实就是埋头去写，把同样或类似的代码写很多遍。说实话，我们甚至会悄悄在测试之间复制粘贴代码，以便更快完成工作。

提示：我们为什么不喜欢 Django Testing Tutorial #5

Django 官方的测试入门教程里演示了一个 `create_question()` 工具方法，目的是减少创建问题对象时的重复代码。

我们认为，把它放进教程里是个错误。官方文档里如果用简单示例去演示糟糕实践，往往会起到鼓励作用，而且这种鼓励常常会在真实项目里发展成很可怕的结果。我们见过一些项目受到这个例子的启发，在测试代码里叠出很深的抽象层，而这种抽象会让修正和增强现有测试变得异常困难。

再说一遍，Don’t Repeat Yourself 不适用于测试。

24.3.5 不要依赖 Fixtures

我们慢慢学到，使用 fixtures 是很成问题的。问题在于，随着项目数据不断演变，fixtures 很难维护。你得去修改 JSON 格式的文件，让它们和最新 migration 匹配，这件事本身就很难；而且在 JSON 加载过程中，想定位到底是 JSON 文件损坏了，还是它只是对数据库做了一个带细微偏差的不准确表达，也常常不容易。

与其和 fixtures 死磕，我们发现，直接写依赖 ORM 的代码反而更轻松。当然，也有人更喜欢用第三方包。

包提示：生成测试数据的工具

下面这些都是很流行的测试数据生成工具：

- `factory boy`：一个用来生成 model 测试数据的包。
- `faker`：这个包也能生成测试数据，但它给的不是一堆随机乱码，而是本地化的人名、地址和文本。它甚至还自带了如何和 `factory boy` 集成的说明：`faker.readthedocs.io/en/master/#how-to-use-with-factory-boy`
- `model bakery`：另一个用来生成 model 测试数据的包。
- `mock`：它并不是专门给 Django 用的，但它允许我们把系统中的一部分替换成 mock objects。这个项目在 Python 3.3 起被并入了标准库。

24.3.6 哪些东西应该被测试

所有东西！真的，能测的就都去测，包括：

- `Views`：数据展示、数据变更，以及自定义 class-based view methods。
- `Models`：model 的创建、更新、删除，model methods，以及 model manager methods。
- `Forms`：form methods、`clean()` methods，以及自定义字段。
- `Validators`：对你写的每个自定义 validator，真的要深入下去，写出多个测试方法。把自己想象成一个恶意入侵者，正试图破坏站点里的数据。
- `Signals`：它们在远处生效，所以如果没有测试，signals 很容易带来痛苦。
- `Filters`：filters 本质上不过就是接收一个或两个参数的函数，因此给它们写测试应该很容易。
- `Template Tags`：template tags 什么都能做，甚至还能接收模板上下文，所以给它们写测试通常会困难得多。也正因如此，你更得测试它们，否则很容易踩到边界情况。
- `Miscellany`：context processors、middleware、email，以及这份清单里没列到的其他任何东西。

`Failure`：上面这些东西一旦失败会发生什么？测试系统报错时的行为，和测试系统成功时的行为一样重要。

唯一不该测试的，是那些已经被 Django 核心和第三方包测试覆盖过的项目部分。比如，如果你原封不动使用 Django 自带的标准字段，那么 model 的字段本身就不需要再测了。但是，如果你创建了某种新的字段类型，例如通过继承 `FileField`，那就应该把新字段类型里一切可能出错的地方都写上细致测试。

24.3.7 为失败场景写测试

假设我们有一个 view，允许用户编辑自己对冰淇淋店的点评。典型测试通常是：登录、尝试修改点评、然后检查它是否真的被改掉了。测试成功场景，覆盖率 100%。对吧？

但这其实只测到了场景的一部分。如果用户没登录怎么办？如果用户试图编辑别人的点评怎么办？这个 view 会不会抛错？更重要的是，对象最终是否保持不变？有人甚至认为，这类测试比成功场景更重要：成功场景失败了，用户只是会遇到不便，而且通常会报告问题；而失败场景失败了，则可能制造一个悄无声息的安全漏洞，直到为时已晚才被发现。

这还只是我们没有测试系统崩坏时行为，所可能引发问题的冰山一角。我们必须学会去测试：当代码抛出各种异常时，它的表现是否符合预期。

- `docs.python.org/2/library/unittest.html#unittest.TestCase.assertRaises`
- `bit.ly/2pAxLtm` PyTest assertion 文档

图 24.1: 图 24.1：尽你所能把项目尽量多的部分都测掉，就像它是免费的冰淇淋一样。

24.3.8 用 Mock 让单元测试不要碰到外部世界

单元测试不应该去测试函数或方法之外的外部事物。这意味着，在测试期间，我们不应该访问外部 API，不应该真的去发邮件或收 webhooks，也不该碰任何不属于当前被测动作本身的东西。可问题也正出在这里：如果你要给一个会和外部 API 交互的函数写单元测试，该怎么办？

这时你有两个选择：

- 选择 1：把这个单元测试改成集成测试。
- 选择 2：使用 Mock 库，伪造来自外部 API 的响应。

Mock 库由 Michael Foord 创建，它的一个重要能力，就是可以临时 monkey-patch 某些库，让它们返回我们想要的精确值。这样一来，我们测试的就不是外部 API 是否可用，而只是我们自己代码的逻辑。

在下面这个例子里，我们对一个虚构的冰淇淋 API 库中的函数做了 monkey-patch，这样测试代码就不会访问应用之外的任何东西。

示例 24.5：用 Mock 让单元测试不要碰到外部世界

Code (python)

```
from unittest import mock, TestCase

import icecreamapi

from flavors.exceptions import CantListFlavors
from flavors.utils import list_flavors_sorted


class TestIceCreamSorting(TestCase):

    # 设置对 icecreamapi.get_flavors() 的 monkeypatch
    @mock.patch.object(icecreamapi, 'get_flavors')
    def test_flavor_sort(self, get_flavors):

        # 让 icecreamapi.get_flavors() 返回一个未排序列表。
        get_flavors.return_value = [
            'chocolate',
            'vanilla',
            'strawberry',
        ]

        # list_flavors_sorted() 会调用 icecreamapi.get_flavors()。
        # 由于这个函数已经被 monkeypatch，它始终只会返回上面的列表，
        # 然后 list_flavors_sorted() 会按字母顺序把它排好。
        flavors = list_flavors_sorted()

        self.assertEqual(
            flavors,
            ['chocolate', 'strawberry', 'vanilla'],
        )
```

下面再演示一下：当 Ice Cream API 不可访问时，如何测试 `list_flavors_sorted()` 的行为。

示例 24.6：测试 API 不可用时的情况

Code (python)

```
@mock.patch.object(icecreamapi, 'get_flavors')
def test_flavor_sort_failure(self, get_flavors):

    # 让 icecreamapi.get_flavors() 抛出 FlavorError。
    get_flavors.side_effect = icecreamapi.FlavorError()

    # list_flavors_sorted() 会捕获 icecreamapi.FlavorError()
    # 并进一步抛出 CantListFlavors 异常。
    with self.assertRaises(CantListFlavors):
        list_flavors_sorted()
```

顺带送给 API 作者一个额外 bonus，下面是我们如何测试代码处理两种不同 python-requests 连接问题的方式：

示例 24.7：测试 python-requests 连接失败

Code (python)

```
@mock.patch.object(requests, 'get')
def test_request_failure(self, get):

    """ 测试目标站点不可访问时的情况。"""

    get.side_effect = requests.exceptions.ConnectionError()

    with self.assertRaises(CantListFlavors):
        list_flavors_sorted()


@mock.patch.object(requests, 'get')
def test_request_failure_ssl(self, get):

    """ 测试我们能否优雅处理 SSL 问题。"""

    get.side_effect = requests.exceptions.SSLError()

    with self.assertRaises(CantListFlavors):
        list_flavors_sorted()
```

24.3.9 用更花哨一点的断言方法

比较两个列表（或元组）是非常常见的场景。不过，如果这两个列表允许有不同的排序顺序，我们是不是就得先把它们排成同样顺序，再执行 self.assertEqual(control_list, candidate_list)？

如果你知道 unittest 里的 assertItemsEqual() 断言方法，那就不用了！事实上，Python 和 Django 的 unittest 文档里都列出了很多非常好用、而且我们本来就免费得到的断言类型：

- docs.python.org/3/library/unittest.html#assert-methods
- docs.djangoproject.com/en/3.2/topics/testing/tools/#assertions

我们发现下面这些断言方法特别有用：

- assertRaises

- `assertRaisesMessage()`
- `assertCountEqual()`
- `assertDictEqual()`
- `assertFormError()`
- `assertContains()`：检查状态码 200，并在 `response.content` 中查找内容。
- `assertHTMLEqual()`：它有很多优点，其中之一是会忽略空白差异。
- `assertInHTML()`：特别适合在一整个巨大的 HTML 页面里确认一小段 HTML。
- `assertJSONEqual()`
- `assertURLEqual()`

24.3.10 给每个测试写清楚用途

正如我们会用 docstrings 去说明 class、method 或 function 的用途一样，也同样值得给这些东西的测试对应物写清楚用途说明。

没有文档的代码，会让项目变得稍微难维护一些；而没有文档的测试代码，则可能让项目根本没法继续测试。要解决这个问题，一点点 docstring 往往就很有帮助。

如果你觉得这很无聊，那么我们发现，处理那种几乎不可能 debug 的问题，一个很好的办法就是把相关测试写上文档。等这些测试文档写完时，我们要么已经找到了问题，要么至少收获了写清楚的测试。无论哪种情况，都是赢。

资源：

- `hynek.me/articles/document-your-tests/`
- `interrogate.readthedocs.io/`：一个用来检查代码库里是否缺少 docstrings 的库

24.4 集成测试怎么办?

集成测试，就是把单独的软件模块组合起来，作为一个整体进行测试。它最适合在单元测试完成之后再做。集成测试的例子包括：

- Selenium 测试，用来确认应用在浏览器里确实可用。
- 真正对第三方 API 发起测试，而不是 mock 掉响应。例如，Django Packages 会定期对 GitHub 和 PyPI API 做测试，以确保它和这些系统之间的交互仍然有效。
- 与 `httpbin.org` 交互，用来确认发出的外部请求是有效的。
- 使用 `runscope.com` 或 `postman.com` 来验证我们的 API 是否按预期工作。

集成测试是确认“所有零件都能一起工作”的好办法。我们可以借此确认，用户确实会看到他们应该看到的内容，我们的 API 也确实在正常工作。

集成测试的缺点则包括：

- 搭建集成测试可能要花很多时间。
- 和单元测试相比，集成测试非常慢。因为它们测试的不是最小组件，而是按定义在测试整个系统。
- 当集成测试抛错时，定位问题通常比单元测试更难。例如，一个只影响某类浏览器的问题，根因可能是数据库层发生了一次 unicode 转换。
- 和单元测试相比，集成测试更脆弱。一个组件或一项设置上的微小变动，就可能把它们弄坏。到目前为止，我们还没做过哪个稍微重要点的项目，能做到至少没有一个人长期卡在“就是跑不通集成测试”这件事上。

即便有这些问题，集成测试依然有用，而且值得考虑纳入你的测试栈。

24.5 持续集成

对于任何规模的项目，我们都建议搭建一个持续集成（CI）服务器，在代码被提交并推送到项目 repo 时自动运行测试套件。参见

24.6 谁在乎？我们可没时间写测试！

“Tests are the Programmer’s stone, transmuting fear into boredom.”

“测试是程序员的贤者之石，能把恐惧转化成无聊。”

- Kent Beck

假设你对自己的编码能力很有信心，于是决定跳过测试，以提升开发速度。或者也许你只是懒。很容易有人会争辩说，即便有各种测试生成器，即便用测试来替代 shell，它们还是会让做事变慢。

哦，真的吗？

那等到该升级的时候呢？

这时候，你在前面额外做的那一点点测试工作，就会替你省下大量劳动。

举个例子。2010 年夏天，我们启动 Django Packages（djangopackages.org）项目时，Django 1.2 还是标准版本。从那以后，这个项目一直能跟上 Django 新版本，而这件事非常有价值。因为它拥有还算不错的测试覆盖率，所以无论把 Django 升一个版本，还是把各种依赖往上升一个版本，都很轻松。我们的升级路径是：

- 在本地的 Django Packages 实例里先升级版本。
- 运行测试。
- 修复测试抛出的所有错误。
- 做一些手工检查。

如果 Django Packages 没有测试，那么每次我们升级任何东西时，就都得手工点过几十上百种场景，而这件事本身非常容易出错。有测试意味着，我们可以带着信心去做改动和依赖升级，因为我们知道用户，也就是 Django 社区，不至于被一个 bug 满满的体验坑到。

这就是测试带来的好处。

24.7 测试覆盖率游戏

一个很好玩、也很有趣的游戏，就是努力把测试覆盖率尽可能提高。每天只要测试覆盖率上升一点，就是一次胜利；而只要覆盖率往下降，那就是一次失败。

24.8 如何搭起测试覆盖率游戏

没错，我们把测试覆盖率叫作一个游戏。它是推动开发者不断前进的好工具。同时，它也是一个很不错的指标，开发者以及他们的客户、雇主、投资人都可以借此帮助评估项目状态。

我们之所以主张下面这些步骤，是因为大多数时候，我们想测的只是项目自己的 apps，而不是 Django 本身，以及构成项目积木的海量第三方库。去测试那些“积木”要花大量时间，这很浪费，因为它们大多本来就已经被测试过，或者还需要额外配置各种资源。

24.8.1 第一步：开始写测试

这个我们已经做过了，对吧？

24.8.2 第二步：运行测试并生成覆盖率报告

来试试看！在命令行里，切到 <project_root>，输入：

示例 24.8：使用 coverage.py 运行 Django 测试

Code (bash)

```
$ coverage run manage.py test --settings=twoscoops.settings.test
```

如果除了两个 app 默认自带的测试之外什么都没有，我们应该会看到类似下面这样的响应：

示例 24.9：正向测试结果

Code (text)

```
Creating test database for alias "default"...
..
----------------------------------------------
Ran 2 tests in 0.008s

OK

Destroying test database for alias "default"...
```

看起来没什么大不了，但它真正说明的是：我们已经把应用限制成只运行自己想运行的测试了。接下来，就是去看看并分析那份低得让人脸红的测试覆盖率数字。

24.8.3 第三步：生成报告！

coverage.py 提供了一种非常有用的方式来生成 HTML 报告。它不只是给出一个“哪些东西被测了”的百分比数字，还会直接告诉我们哪些地方根本没有测试。在命令行里，切到 <project_root>：

示例 24.10：忽略 admin.py 时生成测试报告

Code (bash)

```
$ coverage html --omit="admin.py"
```

咳，别忘了把 <project_root> 换成你自己开发机器上的真实目录结构！例如，具体路径可能会像这样：

- /Users/audreyr/code/twoscoops/twoscoops/
- /Users/pydanny/projects/twoscoops/twoscoops/
- c:\\twoscoops

命令运行后，<project_root> 目录下会新出现一个 htmlcov/ 目录。进入这个 htmlcov/ 目录，用任意浏览器打开其中的 index.html 文件。

你在浏览器里看到的，就是这次测试运行的结果。除非你本来就已经写了不少测试，否则首页上的总覆盖率通常只会是个位数，甚至可能是 0%。点进列出的各个模块后，你会看到大片大片的红色代码。红色可不是什么好事。

我们不妨大大方方承认：自己的项目测试覆盖率很低。如果你的项目覆盖率也很低，那你也得承认它。只要你下定决心把覆盖率慢慢提高，这没什么丢人的。

事实上，公开说自己正在努力提升某个项目的测试覆盖率，一点问题都没有。其他开发者，当然也包括我们自己，反而会为你加油。

24.9 玩测试覆盖率这场游戏

这场游戏只有一条规则：

任何一次 commit，都不允许让测试覆盖率下降。

如果我们新增了一个功能，或者修了一个 bug，而开始时的覆盖率是 65%，那在覆盖率至少回到 65% 之前，这段代码就不能被合并。每天结束时，只要测试覆盖率比前一天高了，不管高多少，我们就算赢。

请记住，覆盖率渐进式上升，往往比那种一次性大跳跃更可贵。渐进式提升通常意味着：我们并没有为了冲数字而塞进一堆凑数测试，而是真的在改善项目质量。

24.10 unittest 的替代方案

到目前为止，本章里的所有例子都使用了 `unittest` 库。虽然所有我们知道的测试权威都认同 `unittest` 是个强大而有用的工具，但并不是每个人都喜欢它。原因也很具体，而且我们完全理解：它需要太多样板代码。

幸运的是，还有一种需要更少样板代码的 `pytest` 替代方案：`pypi.org/project/pytest-django/`

这个库是对 `pytest` 库的一层包装。只需要多做一点点额外设置，它不仅能运行基于 `unittest` 的测试，还能运行任何带有 `test_` 前缀的 function，以及带有这一前缀的 class、directory、module。例如，你可以写出一个像下面这样的简单测试：

示例 24.11：`py.test` 示例

Code (python)

```
# test_models.py
from pytest import raises

from cones.models import Cone


def test_good_choice():
    assert Cone.objects.filter(type='sugar').count() == 1


def test_bad_cone_choice():
    with raises(Cone.DoesNotExist):
        Cone.objects.get(type='spaghetti')
```

虽然这个例子基于 `pytest`，但 `nose` 及其 `nose.tools.raises` decorator 也能实现类似能力。

这种基于函数的测试之所以看起来很简洁，一个潜在代价就是缺乏继承能力。如果项目里有很多不同测试都需要相似行为，那么用这种方式写测试也许就不太合适了。

24.11 总结

这一切听起来也许有点傻，但测试确实可能是一门非常严肃的事情。在很多开发团队里，这个主题虽然被游戏化了，但大家仍然会非常认真地对待它。一个项目缺乏稳定性，可能意味着失去客户、失去合同，甚至失去工作。

下一章我们会讲 Python 开发者的另一种常见执念：文档。

第二十五章 文档：痴迷一点

如果要在吃冰淇淋和写出色文档之间二选一，大多数 Python 开发者大概会选写文档。当然，一边吃冰淇淋一边写文档就更好了。

当你手里有 Markdown、MkDocs、Sphinx、Sphinx 的 Myst-Parser 这类优秀文档工具时，你几乎会忍不住想给项目补上文档。

25.1 文档请用 GitHub-Flavored Markdown

你需要学习并遵循文档编写的标准最佳实践。如今，GitHub-Flavored Markdown（GFM）是为 Python 项目写文档时最常见的标记语言。其他工具和公司也采用了 GFM，例如 GitLab 就在它们的 README 和 issue tracker 里使用 GFM。

下面是 GFM 的正式规范，以及一个能很好体现它价值的示例项目：

- `github.github.com/gfm/`
- `django-rest-framework.org/`

当然，你可以去认真研读 GFM 的正式文档，再从中掌握基础知识。不过这里先给一个非常快的入门示例，列出一些你应该学会的实用写法。

示例 25.1：Markdown 入门速览

~~~md
~~~

第二十六章 一级标题

emphasis (bold/strong)

italics

underline

link: Two Scoops Press Note: Markdown 里链接的诀窍，是把它想成一次函数调用。括号里的链接，就是被调用的值。

26.1 小节标题

1. An enumerated list item
2. Second item

- First bullet
- Second bullet
- Indented Bullet
- Note carriage return and indents

Literal code block:

Code (python)

```
def like():
    print("I like Ice Cream")


for i in range(10):
    like()
```

JavaScript colored code block:

Code (js)

```
console.log("Don't use alert()");
```

~~~

26.2 使用 MkDocs 或带 Myst 的 Sphinx 从 Markdown 生成文档

MkDocs 和带 Myst 的 Sphinx 都是静态站点生成器，可以把你的 `.md` 文件渲染成好看的文档站点。输出格式包括 HTML、LaTeX、manual pages 和纯文本。

MkDocs 基本上按照 `mkdocs.org/#getting-started` 上的使用说明去做就能跑起来。

如果你用的是 Sphinx，则需要按照 Myst 的说明来生成 Markdown 文档：`myst-parser.readthedocs.io/en/latest/using`

提示：至少每周构建一次文档

你永远不知道什么时候，错误的交叉引用或无效格式就会把文档构建搞挂。与其在尴尬时刻才发现文档根本 build 不出来，不如养成定期构建文档的习惯。我们的偏好是把这件事纳入 Continuous Integration 流程，正如第 34 章《Continuous Integration》里会讲到的那样。

包提示：其他文档生成器

上面我们列的是两个主要的 Python 系文档站点生成器，但如果你愿意探索非 Python 方案，那么开源和商业文档工具其实还有很多。下面是一个简短清单，列出我们过去几年亲自用过、而且觉得很不错的：
~~~

- docusaurus.io
- docsify.js.org
- bookdown.org

26.3 Django 项目至少应该包含哪些文档?

面向开发者的文档，指的是开发者为了搭建和维护项目所需要的说明和指南。这包括安装、部署、架构、如何运行测试、如何提交 pull requests 等等。我们的经验是，不管项目是私有还是公开，把这类文档放进项目里都非常有帮助。

下面这张表，列出了我们认为最低限度也该具备的文档：

文件名或目录	理由	备注
README.md	无论你开始的是哪种框架、哪种语言的编码项目，都应该在仓库根目录放一个 README.md 文件。	至少写一小段话，说明这个项目是干什么的。另外，再链接到 docs/ 目录里的安装说明。
docs/	项目文档应该放在一个统一且稳定的位置。这是 Python 社区的标准做法。	一个简单的目录即可。
docs/deployment.md	这份文件能让你放心给自己放一天假。	按步骤写清楚如何把项目安装到生产环境、以及如何更新它。即便项目用了各种理论上让事情“变简单”的 devops 工具，这里也还是应该把过程写清楚。
docs/installation.md	当新成员加入项目，或者你换了台新电脑需要重新配置项目时，这份文档会特别有用。	按步骤写清楚如何为自己或其他开发者完成项目的软件环境搭建。
docs/architecture.md	当项目随着时间推移不断长大、范围不断扩展时，这份文档能帮助你理解系统最初是怎么一步步演化到现在的。	这里可以用简单文本写下你心中期望项目长成的样子，长短不限。在一项工作刚开始时，它尤其有助于团队保持聚焦。

表 25.1：Django 项目应该包含的文档

26.4 更多 Markdown 文档资源

- python.org/dev/peps/pep-0257：docstrings 的正式规范。
- readthedocs.io：Read the Docs 是一项免费服务，可以托管你的 Sphinx 或 MkDocs 文档。
- pythonhosted.org：Python Hosted 是另一项免费的文档托管服务。
- en.wikipedia.org/wiki/Markdown
- documentup.com：可以托管用 Markdown 格式写成的 README 文档。

26.5 ReStructuredText 这个替代方案

ReStructuredText 是一种纯文本格式语法，和 Markdown 不算差得太远。它内建的特性比 Markdown 多得多，但也更难学、写起来更慢。Django、Python 以及很多较老的第三方库等核心工具，都在使用它。

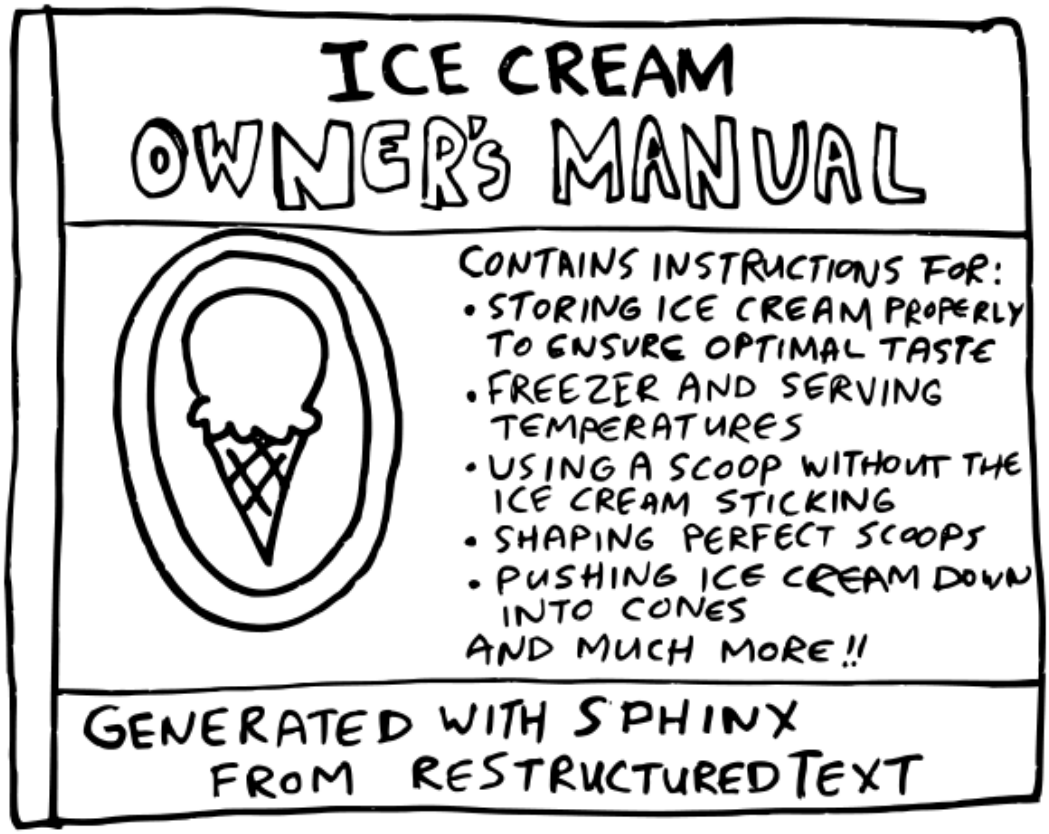

图 26.1: 图 25.1：连冰淇淋都能从文档中受益。

26.5.1 ReStructuredText 资源

- `docutils.sourceforge.net/docs/ref/rst/restructuredtext.html`：ReStructuredText 的正式规范。
- `docs.readthedocs.io/en/stable/intro/getting-started-with-sphinx.html`:Read the Docs 为 Sphinx 提供的入门说明。
- `sphinx-doc.org/`：Sphinx 项目主页。虽然它也能配合 Markdown 使用，但 Sphinx 真正最发光的时候，还是和 ReStructuredText 搭配在一起。

26.6 当文档需要在 Markdown 和 ReStructuredText 之间互转时

Pandoc 是一个命令行工具，可以把文件从一种标记格式转换成另一种。我们完全可以先用一种格式写，再迅速转成另一种。下面是使用这个工具进行转换的方法：

示例 25.2：使用 Pandoc 在不同格式之间转换

Code (bash)

```
$ # 把一个 ReStructuredText 文档转成 GitHub-Flavored Markdown
$ pandoc -t gfm README.rst -o README.md
$ # 把一个 Markdown 文档转成 ReStructuredText
$ pandoc -f gfm README.md -o README.rst
```

Pandoc 的文档在 `pandoc.org`。

26.7 Wiki 以及其他文档方法

如果出于某种原因，你没法把面向开发者的文档直接放进项目本身，那至少也该有别的选择。wiki、在线文档存储系统、文字处理文档，虽然都没有“能被纳入版本控制”这个优点，但总比没有文档强。

请考虑在这些替代方式里，也沿用我们在上一页表格中建议的那些文档名称。

26.8 确保代码本身也有文档

即便你已经有了清晰的命名模式和 type hints，要判断某个 class、method 或 function 究竟是干什么的，仍然可能不容易。这正是 docstrings 多年来一次又一次救场的原因。为了更容易强制执行这种级别的文档覆盖，请使用 Interrogate 库。

`interrogate.readthedocs.io/`

26.9 总结

这一章我们讲了下面这些内容：

- 使用 Markdown 以纯文本格式编写文档。
- 使用静态站点生成器把文档渲染成 HTML。有些工具还能把文档渲染为 EPUB、MOBI 或 PDF。
- 任何 Django 项目都应该满足的文档要求。
- 使用 ReStructuredText 作为一种替代性文档格式。
- 使用 Pandoc 作为不同格式之间的转换工具。
- 用 Interrogate 来强制检查文档覆盖。

接下来，我们会看看 Django 项目里常见的瓶颈，以及处理这些瓶颈的方法。

第二十七章 发现并减少瓶颈

警告：本章仍在编写中

我们正在撰写这一章，接下来几天还会继续扩充内容。欢迎就应该覆盖的话题和条目向我们提建议，请提交到：`github.com/feldroy/two-scoops-of-django-3.x/issues`

本章会介绍几种基础策略，用来识别瓶颈，并让你的 Django 项目跑得更快。

27.1 你真的需要在乎这个吗?

记住，过早优化不是好事。如果你的网站规模还小或中等，而且页面加载一切正常，那跳过这一章也没关系。

但如果你的网站用户群正在稳定增长，或者你马上就要和某个热门品牌达成战略合作，那就继续读下去。

27.2 让查询密集型页面更快

这一节会讲，如何减少“查询数量太多”带来的瓶颈，以及如何减少“单条查询本来还能更快”带来的瓶颈。

我们也强烈建议你去读官方 Django 文档里关于数据库访问优化的内容:`docs.djangoproject.com/en/3.2/topics/db/o`

27.2.1 用 Django Debug Toolbar 找出过量查询

你可以使用 `django-debug-toolbar` 来帮助判断：大多数查询究竟是从哪里冒出来的。它能帮你发现这样的瓶颈：

- 页面里存在重复查询。
- 某些 ORM 调用最终展开成了远超你预期的查询数量。
- 慢查询。

你大概已经对一些值得优先查看的 URL 有个模糊判断了。比如，哪些页面加载时明显不够利索?

如果你还没在本地安装 `django-debug-toolbar`，现在就装上它。用浏览器打开项目，并展开 SQL 面板。它会告诉你当前页面包含多少条查询。

包提示：用于 Profiling 和性能分析的工具

`django-debug-toolbar` 是关键的开发工具,对逐页分析来说非常有价值。我们还建议你把 `django-cache-panel` 加入项目里，但只在调用 `settings/local.py` 模块时才启用它。这样能让你更清楚地看到缓存到底在干什么。

`django-extensions` 自带一个名为 `RunProfileServer` 的工具，它会在启用 `hotshot` / profiling 工具的情况下启动 Django 的 `runserver` 命令。

`silk`（`github.com/mtford90/silk`）是一个实时 profiling Django app。它会拦截并存储 HTTP requests 和数据库查询，然后通过一套用户界面把它们展示出来，方便进一步检查。

27.2.2 减少查询数量

一旦你知道了哪些页面包含不理想的查询数量，就该想办法把它降下来。下面是一些可以尝试的手段：

- 试着在 ORM 调用中使用 `select_related()` 来合并查询。它会沿着 `ForeignKey` 关系追下去，把更多数据并进一条更大的查询里。如果你在用 CBVs，那么 `django-braces` 提供的 `SelectRelatedMixin` 会让这件事变得非常省事。注意，显式传入你真正关心的关联字段名，以免查询变得大得离谱。只有被明确指定的关系才会被跟随。再配合仔细测试!
- 对于无法用 `select_related()` 优化的 many-to-many 和 many-to-one 关系,可以改为研究 `prefetch_related()`。

- 如果同一条查询在每个 template 中被生成了不止一次，那就把查询挪到 Python view 里，作为变量塞进上下文，然后让模板里的 ORM 调用改用这个新的上下文变量。
- 使用类似 Memcached 或 Redis 这样的 key/value 存储来实现缓存。然后写测试，断言某个 view 实际跑了多少条查询。相关说明见：docs.djangoproject.com/en/3.2/topics/testing/tools/#django.test.TransactionTestCa
- 使用 django.utils.functional.cached_property decorator，在对象实例的生命周期内，把某次方法调用结果缓存在内存中。这招非常好用，所以请去看第 31 章 31.3.5 节《django.utils.functional.cached_property》。

27.2.3 提速常见查询

单条查询本身花费的时间，也可能成为瓶颈。下面这些建议可以当作起步点，但请把它们仅仅视为起步点：

- 确保你的 indexes 确实在帮最常见的慢查询提速。去看这些查询生成的原始 SQL，并在你最常用来过滤 / 排序的字段上建索引。重点观察生成出来的 WHERE 和 ORDER_BY 子句。
- 弄清楚你的 indexes 在生产环境里到底发挥了什么作用。开发机永远无法完美复现生产现场，所以你必须学会分析和理解数据库里真实发生了什么。
- 看看常见查询生成出来的 query plans。
- 打开数据库的 slow query logging 功能，看看是否有某些慢查询频繁出现。
- 在开发环境里用 django-debug-toolbar 主动识别潜在的慢查询，尽量在它们打到生产前就发现。

一旦你已经有了合理的索引，也做了足够的分析，知道哪些查询值得重写，那么可以从下面这些方向入手：

1. 尽可能重写业务逻辑，让它返回更小的结果集。
2. 重新设计数据模型，让 indexes 能更高效地工作。
3. 在某些位置下沉到 raw SQL，因为它可能比自动生成的 query 更高效。

提示：使用 EXPLAIN ANALYZE / EXPLAIN

如果你使用 PostgreSQL，那么可以用 EXPLAIN ANALYZE 来拿到任意原始 SQL 查询非常详细的 query plan 与分析结果。更多信息见：

- revsys.com/writings/postgresql-performance.html
- craigkerstiens.com/2013/01/10/more-on-postgres-performance/

MySQL 中对应的是 EXPLAIN 命令。它没有那么详细，但依然很有帮助。更多信息见：

- dev.mysql.com/doc/refman/5.7/en/explain.html

django-debug-toolbar 还有个很不错的点：SQL 面板里自带 EXPLAIN 功能。

27.2.4 把 ATOMIC_REQUESTS 切到 False

绝大多数 Django 项目把 ATOMIC_REQUESTS 设成 True 都能跑得很好。一般来说，让所有数据库查询都运行在事务中的代价并不明显。不过，如果你的瓶颈分析显示：事务本身已经造成了过多延迟，那就该把项目调整为在 ATOMIC_REQUESTS = False 下运行了。关于这个设置的指导，请参见 7.7.2 节《显式事务声明》。

27.3 榨干数据库的能力

你还可以比“优化数据库访问”再往下挖一层，直接优化数据库本身！这里面很多内容都和具体数据库有关，而且其他书已经讲得很多了，所以我们不会在这里展开太细。

27.3.1 清楚什么东西不该放进数据库

Revolution Systems 的 Frank Wiles 教会我们：有两种东西，永远不该塞进任何大型站点的关系型数据库里：

`Logs`。不要把 logs 放进数据库。它们表面上看起来没什么，尤其在开发环境里更是如此。但在生产数据库里额外加入这么多写操作，会拖慢数据库性能。如果你确实需要对 logs 做复杂查询，我们建议使用第三方服务，比如 Splunk 或 Loggly，或者改用基于文档的 NoSQL 数据库。

`Ephemeral data`。不要把短命数据存在数据库里。所谓短命数据，就是那些需要被不断重写的数据，这类数据并不适合关系型数据库。例子包括 `django.contrib.sessions`、`django.contrib.messages` 以及各种 metrics。更好的做法，是把这类数据迁移到 Memcached、Redis 或其他非关系型存储中。

提示：Frank Wiles 谈数据库里的二进制数据

其实，Frank 说的是“三种东西”永远不该进数据库，第三种就是二进制数据。数据库里二进制数据的存储问题，可以通过 `django.db.models.FileField` 来处理，它会替你把文件存到像 AWS CloudFront 或 S3 这样的文件服务器上。这里的例外情况，会在 6.4.5 节《何时使用 BinaryField》中详细说明。

27.3.2 让 PostgreSQL 发挥到最好

如果你使用 PostgreSQL，请务必确认它在生产环境里被正确配置。由于这超出了本书范围，我们推荐阅读下面这些文章：

- `wiki.postgresql.org/wiki/Detailed_installation_guides`
- `wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server`
- `revsys.com/writings/postgresql-performance.html`
- `craigkerstiens.com/2012/10/01/understanding-postgres-performance`
- `craigkerstiens.com/2013/01/10/more-on-postgres-performance`

27.3.3 让 MySQL 发挥到最好

让 MySQL 跑起来很容易，但要把生产环境安装优化好，就需要经验和理解了。由于这同样超出本书范围，我们建议你参考下面这些链接来入门：

- `TODO - list good resources`

27.4 用 Memcached 或 Redis 缓存查询

只要把 Django 自带的缓存系统配上 Memcached 或 Redis，你通常就能获得不小的收益。你需要安装其中一个工具，安装为它提供 Python bindings 的包，并配置你的项目。

你可以很轻松地设置整站缓存，也可以缓存单个 view 的输出，或者缓存模板片段。你还可以使用 Django 的底层 cache API 来缓存 Python objects。

参考资料：

- `docs.djangoproject.com/en/3.2/topics/cache/`
- `github.com/niwinz/django-redis`

27.5 找出应该缓存的具体位置

决定“该缓存哪里”，有点像在 Ben and Jerry’s 免费挖球日那天，排在一条漫长、而且所有人都不耐烦的队伍最前面。你得在巨大压力下迅速做决定，却根本来不及看清每种口味到底长什么样。

下面这些问题值得认真想一想：

- 哪些 views / templates 包含的查询最多？
- 哪些 URL 被请求得最频繁？
- 某个页面的缓存应该在什么时候失效？

下面我们来看看，哪些工具能帮助你应对这些场景。

27.6 考虑第三方缓存包

第三方包会给你带来一些额外能力，例如：

- `QuerySets` 的缓存。
- 缓存失效设置 / 机制。
- 不同的缓存 backend。
- 替代性或实验性的缓存思路。

一些流行的 Django 缓存包包括：

- `django-cacheops`
- `django-cachalot`

更多选项见：`djangopackages.org/grids/g/caching/`

警告：第三方缓存库并不总是答案

我们试过很多第三方 Django 缓存库，最后不得不提醒读者：一定要非常仔细地测试它们，并且做好随时放弃它们的准备。它们通常是廉价、快速的短期胜利，但也可能在最糟糕的时候，把你拖进令人头皮发麻的调试现场。

缓存失效本来就很难，而按我们的经验，那些“很神奇”的缓存库更适合内容更静态的项目。手工缓存工作量更大，但从长期看性能往往更好，也不容易引爆那些可怕时刻。

27.7 压缩和压缩整理 HTML、CSS 与 JavaScript

当浏览器渲染一个网页时，通常必须加载 HTML、CSS、JavaScript 和图片文件。这些文件都会消耗用户带宽，从而拖慢页面加载。减少带宽消耗的一种方式，就是 compression 和 minification。Django 甚至直接为你提供了工具：`GZipMiddleware` 和 `{% spaceless %}` 模板标签。更广义的 Python 社区里，我们还能用执行同样任务的 WSGI middleware。

问题在于，如果让 Django 和 Python 自己来做这些活，compression 和 minification 会占用系统资源，而这本身也可能制造新的瓶颈。更好的方案，是配置 Nginx 或 Apache 这类 web server 来压缩输出内容。如果你维护的是自己的 web server，这绝对是应该走的路。

一个常见做法，是使用第三方压缩模块或 Django 库，提前把 HTML、CSS 和 JavaScript 压缩 / 压缩整理好。我们的偏好是 `django-pipeline`，这是 Django 核心开发者 Jannis Leidel 推荐的方案。

对于 CSS 和 JavaScript，大多数人会使用由 JavaScript 驱动的压缩工具。像 `django-webpack-loader` 这类工具，会在 Django 的上下文里管理 JavaScript libraries。这种方式的优势，在于这个领域的工具心智占有率更高，已有解法也更多。

可以参考的工具和库：

- Apache 和 Nginx 的压缩模块
- `django-webpack-loader`
- `django-pipeline`
- `django-compressor`
- `django-htmlmin`
- Django 内建的 `spaceless` 标签：`docs.djangoproject.com/en/3.2/ref/templates/builtins/spaceless`
- `djangopackages.org/grids/g/asset-managers/`

27.8 使用上游缓存或 Content Delivery Network

像 Varnish 这样的 upstream caches 很有用。它们运行在你的 web server 前面，能显著加快网页或内容的分发速度。参见 `varnish-cache.org`。

像 Fastly、Akamai 和 Amazon CloudFront 这样的 Content Delivery Networks（CDNs），会提供图片、视频、CSS 和 JavaScript 等静态媒体。它们通常在全世界各地都有服务器，会从离用户最近的位置把静态内容发出去。与其从应用服务器直接提供静态资源，使用 CDN 往往能让项目更快。

27.9 其他资源

关于扩容、性能、调优和优化的高级技巧，已经超出了本书范围，不过这里还是给几个起点。

关于 Web 性能的一般最佳实践：

- `TODO - add useful links`

关于大型 Django 站点的扩展：

- 《The Temple of Django Database Performance》是一本深入讲 Django 项目速度与可扩展性优化的书。它写得很讨喜，充满奇幻和 RPG 梗，而且值回票价：`spellbookpress.com/books/temple-of-django-database-performance/`
- 那本《High Performance Django》也是围绕 Django 扩展性写成的，里面有很多不错的实践。它充满技巧、提示，以及每节都会逼你认真思考的提问。虽然有些地方已经显得过时，但仍然包含大量有用信息：`highperformancedjang`
- 去看往届 DjangoCon 和 PyCon 的相关演讲视频，了解不同开发者的实战经验。扩展策略会随着年份和公司背景而不同：`https://www.youtube.com/results?search_query=django+scaling`

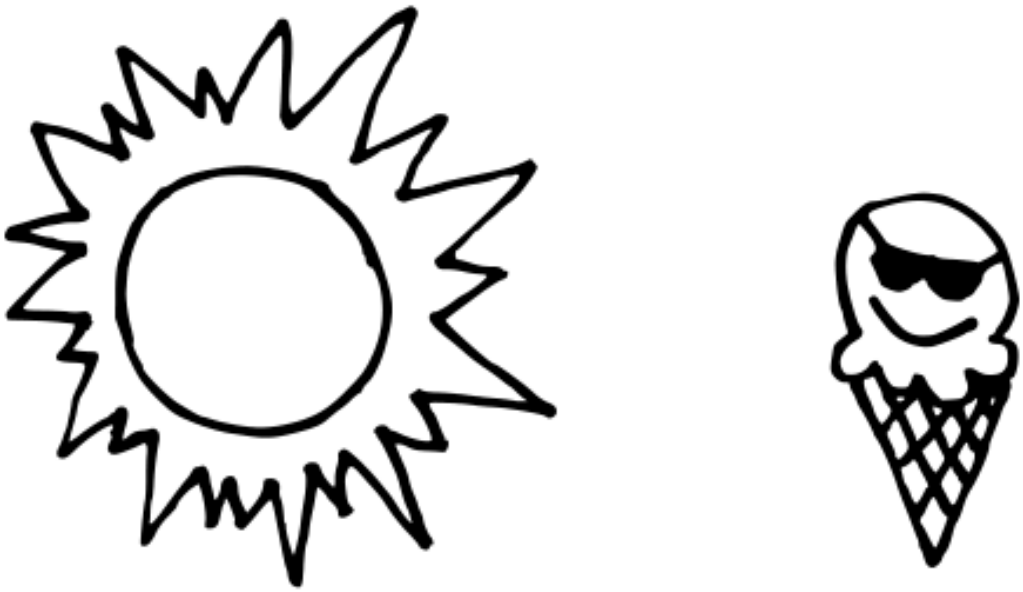

图 27.1: 图 26.1：当你的网站顺畅运转起来时，你会酷得像个甜筒。

提示：高流量站点请看《High Performance Django》

我们想再次强调：如果你的网站流量已经大到会引发问题，那《High Performance Django》真的值得买。虽然它已经有些年头了，但 Peter Baumgartner 和 Yann Malet 写这本书时更注重概念层面，所以它仍然是一本值得考虑购买的书。

- `highperformancedjango.com`
- `amazon.com/High-Performance-Django/dp/1508748128`

27.10 总结

这一章里，我们看了不少减少瓶颈的策略，包括：

- 先判断自己到底要不要在这个阶段关心瓶颈。
- 对页面和查询做 profiling。
- 优化查询。
- 更明智地使用数据库。
- 缓存查询。
- 识别哪些地方该缓存。
- 压缩 HTML、CSS 和 JavaScript。
- 查看其他资源。

下一章我们会讲各种与异步任务队列有关的实践，而它们也许正能帮我们解决这些瓶颈问题。

第二十八章 异步任务队列

警告：本章仍在编写中

我们正在撰写这一章，接下来几天还会继续扩充内容。欢迎就应该覆盖的话题和条目向我们提建议，请提交到：`github.com/feldroy/two-scoops-of-django-3.x/issues`

所谓异步任务队列，是指：任务会在“被创建之后的另一个时间点”才执行，而且执行顺序也未必和创建顺序相同。下面先看一个“由人力驱动”的异步任务队列示例：

1. Audrey 和 Daniel 业余时间会给亲友做冰淇淋蛋糕。他们用一个 issue tracker 来跟踪每个蛋糕的挖球、抹面和装饰任务。
2. 他们隔一阵子、等自己有空时，就会回头看一眼任务列表，然后挑一项来做。Audrey 更喜欢挖球和装饰，所以总是优先做这些事。Daniel 更喜欢挖球和抹面，通常也是先把这两类任务做完再说装饰。结果就是：做蛋糕的任务会以异步方式被完成。
3. 当某个蛋糕任务完成并交付后，他们就把对应 issue 标记为 closed。

提示：Task Queue 与 Asynchronous Task Queue

在 Django 世界里，这两个词常常都用来指代异步任务队列。当有人在 Django 语境下写 `task queue` 时，他们通常说的其实就是 `asynchronous task queue`。

在进入最佳实践之前，我们先过一遍几个定义：

`Broker` 任务本身的存储层。它可以由任何一种持久化工具实现，不过在 Django 世界里，最常见的是 RabbitMQ 和 Redis。在前面那个“人力驱动”的例子里，存储层就是在线 issue tracker。

`Producer` 负责把任务加进队列、以便稍后执行的代码。这就是应用代码，也就是构成 Django 项目的那些东西。在前面的例子里，这对应 Audrey 和 Daniel，以及任何被他们拉来帮忙的人。

`Worker` 负责从 broker 里取出任务并执行的代码。通常不会只有一个 worker。大多数情况下，每个 worker 都会作为一个受监管的 daemon 进程运行。在前面的例子里，这就是 Audrey 和 Daniel。

`Serverless` 通常由 AWS Lambda 这类服务提供。借用 Martin Fowler 的说法，它指的是：“某些服务端逻辑由我们编写，但不同于传统架构的是，它运行在无状态、事件触发、短暂存在（每次调用只活一次）且完全由第三方托管的计算容器里。”Serverless 同时接管了 Broker 和 Worker 的角色。在前面的“人力驱动”例子里，这就像 Audrey 和 Daniel 改用一个第三方服务来接单，然后完全按照他们的明确指令去完成工作。

28.1 我们真的需要任务队列吗?

这要看情况。任务队列会增加复杂度，但也能改善用户体验。归根到底，问题在于：某段代码是否已经形成瓶颈，以及它是否可以被推迟到稍后、等有更多空闲 CPU 周期时再处理。

下面有一个经验法则，可以帮助判断某件事是否该上任务队列：

结果处理起来要花时间：大概率应该使用任务队列。用户能够也应该立刻看到结果：大概率不应该使用任务队列。

下面来看几个可能的使用场景：

场景	使用任务队列？
发送批量邮件	是
修改文件（包括图片）	是
从第三方冰淇淋 API 拉取大量数据	是
向某张表插入或更新大量记录	是
更新用户资料	否
添加博客或 CMS 条目	否
执行耗时计算	是
发送或接收 webhooks	是

表 27.1：项目是否应该使用任务队列？

不过也请记住，所有这些场景都存在“由站点流量规模驱动的例外”：

- 对于流量较小或中等的网站，这些动作也许一个都不需要任务队列。
- 对于流量更大的网站，几乎每个用户动作都可能最终需要借助任务队列。

判断一个站点、或某个动作，到底需不需要任务队列，本身就有点像门艺术。我们没法给出一个简单的统一答案。不过，学会如何使用它们，绝对会成为开发者工具箱里非常强大的一件武器。

第二十九章 安全最佳实践

说到安全，Django 的纪录一直相当不错。这得益于 Django 自带的安全工具、围绕安全主题的扎实文档，以及一支对安全问题反应极快、做事也很审慎的核心开发团队。不过，最终还是要靠像我们这样的 Django 开发者，真正理解应该怎样把基于 Django 的应用保护好。

这一章列出了一些有助于保护 Django 应用的事情。当然，这份清单绝不是完整的。把它当成一个起点更合适。

提示：如果你已经发生了安全事故该怎么办

如果你正身处安全危机中，请直接跳去第 37 章附录 G：《处理安全故障》。

29.1 参考本书其他章节中的安全小节

本书其他若干章节里也有专门的安全小节，或者会触及安全问题。位置如下：

- 5.3 节《将配置与代码分离》
- 13.3 节《对会修改数据的 HTTP 表单始终使用 CSRF 保护》
- ??：??
- 28.28 节《绝不展示顺序型主键》
- 21.8 节《保护 Django Admin》
- 第 37 章附录 G：《处理安全故障》

29.2 强化你的服务器

去网上搜索与你平台对应的 server hardening 说明和检查清单。服务器强化措施包括但不限于：设置防火墙（`help.ubuntu.com/community/UFW`）、修改 SSH 端口、禁用或移除不必要的服务等。

29.3 了解 Django 自带的安全特性

Django 的安全特性包括：

- 跨站脚本（XSS）防护。
- 跨站请求伪造（CSRF）防护。
- SQL 注入防护。
- Clickjacking 防护。
- 对 TLS / HTTPS / HSTS 的支持，包括 secure cookies。
- 自动 HTML escaping。
- 针对 XML bomb 攻击加固过的 expat parser。
- 加固过的 JSON、YAML 与 XML 序列化 / 反序列化工具。
- 安全的密码存储。默认使用带 SHA256 哈希的 PBKDF2 算法。不过，请按照 28.29 节《将密码哈希器升级到 Argon2》继续升级。

Django 的大多数安全特性开箱即用，不需要额外配置；但也有些地方仍然必须由你自己配置。本章会特别强调其中一些细节，不过也请务必阅读 Django 官方安全文档：`docs.djangoproject.com/en/3.2/topics/security/`

29.4 生产环境里关闭 DEBUG 模式

你的生产站点绝不应该在 DEBUG 模式下运行。攻击者可以从那个“贴心”的 DEBUG 堆栈跟踪页面里知道太多关于生产环境的信息。更多内容见：docs.djangoproject.com/en/3.2/ref/settings/#debug

还要记住，一旦关闭 DEBUG，你就必须设置 ALLOWED_HOSTS，否则很容易抛出 SuspiciousOperation，进而返回一个很难排查的 400 BAD REQUEST。关于如何设置 / 调试 ALLOWED_HOSTS，参见：

- 28.7 节《使用 Allowed Hosts 校验》
- ??: ??

29.5 保守你的 Secret Keys 的秘密

如果 SECRET_KEY 不再是秘密，那么根据项目配置不同，攻击者可能会接管别人的 sessions、重置密码，甚至做更多事。我们的 API keys 和其他 secrets 也一样，都必须被小心保管。这些 keys 甚至都不该放进版本控制里。

关于如何把 SECRET_KEY 排除在版本控制之外，我们在第 5 章《Settings and Requirements Files》里的 5.3 节《将配置与代码分离》和 5.4 节《当你不能使用环境变量时》都讲过具体做法。

29.6 全站 HTTPS

站点必须始终部署在 HTTPS 后面。没有 HTTPS，就意味着恶意网络用户可以在你的网站和终端用户之间嗅探认证凭证。事实上，网站与用户之间传输的所有数据都可能被截走。

而且，你也无法保证用户看到的内容就是你原本想让他们看到的：攻击者完全可以篡改请求或响应里的任何东西。所以，就算你的信息全是公开的，只要你在乎信息完整性，HTTPS 也依然有意义。

你的整个站点都必须走 HTTPS。站点的静态资源也一样必须通过 HTTPS 提供，否则用户会看到“不安全资源”的警告，而这种警告本来就足以把人吓跑。之所以会有这些警告，是因为它们本身就是潜在的中间人攻击入口。

提示：Jacob Kaplan-Moss 谈 HTTPS 与 HTTP

Django 联合负责人 Jacob Kaplan-Moss 说：“你的整个站点都应该只通过 HTTPS 提供，而不该同时开放 HTTP。这样可以防止被‘firesheeped’（也就是在 HTTP 连接上把 session cookie 偷走）。代价通常非常小。”

如果访问者试图通过 HTTP 打开你的网站，他们必须被重定向到 HTTPS。这件事可以通过 web server 配置完成，也可以通过 Django middleware 完成。就性能而言，在 web server 层做更好；不过如果你无法控制 web server 设置，那么通过 Django middleware 重定向也完全可以接受。

在现代环境下，获取 SSL 证书已经又便宜又容易。视平台而定，甚至还是免费的。具体搭建时，请遵循你所用 web server 或平台即服务的说明。我们偏好的服务是声誉很好的 letsencrypt.org 和 cloudflare.com。它们能让你很轻松地为站点配好 SSL certificates。

AWS、Google 和 Microsoft 这类大型云服务商也都提供相应方案。

警告：最安全的选择是自己配置 SSL

虽然 CloudFlare、AWS、Google Cloud 和 Azure 之类的服务，让配置 SSL 变得很容易，但它们并不像 letsencrypt.org 那样安全。最容易解释这一点的方法，是最小权限原则（POLP，Principle of Least Privilege）。这个原则的含义是：把用户的访问权限压到“完成工作所必需的最低限度”。

这些系统被外部攻击者、甚至内部员工攻破的概率确实不高，但也绝不是完全不可能。

对很多系统来说，借助别人的 SSL 能省下大量时间，这个收益是值得的。不过，如果网站承载的是高度敏感的隐私信息，例如 HIPAA 相关数据，那么就应该自己配置 SSL。

警告：没有 HTTPS 是不可原谅的

本书出版时已经是 2020 年，把 HTTPS / SSL 配起来已经相对容易。更重要的是，浏览器会对没有 SSL 的站点给出醒目而且几乎无法忽视的警告。各年龄段、各种技术背景的用户，都越来越知道这些警告意味着什么；搜索引擎也会严厉惩罚没有它的站点。

从现在起，生产站点不上 HTTPS 已经没有任何借口。别这么做，哪怕只是 demo 或 MVP 也不行。

提示：使用 `django.middleware.security.SecurityMiddleware`

在 Django 项目里，如果你想通过 middleware 在整个站点范围内强制启用 HTTPS / SSL，最佳工具其实就是内建的 `django.middleware.security.SecurityMiddleware`。启用它只需要：

1. 把 `django.middleware.security.SecurityMiddleware` 加进 `settings.MIDDLEWARE`。
2. 把 `settings.SECURE_SSL_REDIRECT` 设成 `True`。

警告：`django.middleware.security.SecurityMiddleware` 不会覆盖 static/media

即便所有 Django 请求都走了 HTTPS，如果像 JavaScript 这类资源仍然没走 HTTPS，攻击者依然可以借此攻陷你的网站。

由于 JavaScript、CSS、图片以及其他静态资源，通常都是由 Web 服务器（如 nginx、Apache）直接提供的，所以一定要确认这些内容也是通过 HTTPS 提供的。像 Amazon S3 这种静态资源提供者，现在默认就会这么做。

29.6.1 使用 Secure Cookies

你的网站应该告诉目标浏览器：除非通过 HTTPS，否则永远不要发送 cookies。你需要在 settings 里配置下面这些项：

示例 28.1：保护 Cookies

Code (python)

```python
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
```

更多细节见：`docs.djangoproject.com/en/3.2/topics/security/#ssl-https`

29.6.2 使用 HTTP Strict Transport Security（HSTS）

HSTS 可以在 web server 层配置。请遵循你的 web server、平台即服务，以及 Django 自身（通过 `settings.SECURE_HSTS_SEC` 的配置说明。

如果你自己搭建 web server，Wikipedia 上提供了一些 HSTS 配置片段可作为起点：`en.wikipedia.org/wiki/HTTP_Strict_`

当你启用 HSTS 后，站点的网页会带上一个 HTTP header，通知支持 HSTS 的浏览器以后只能通过安全连接访问该站点：

- 支持 HSTS 的浏览器会把 HTTP 链接自动重定向到 HTTPS。
- 如果安全连接不可用，例如证书是自签名的、或者已经过期，浏览器就会显示错误信息，并阻止访问。

为了让你更直观地理解这一点，下面是一个 HTTP Strict Transport Security 响应头的示例：

示例 28.2：HSTS 响应头

Code (text)

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

关于 HSTS，有几条配置建议：

1. 在一个刚启用 HTTPS 的站点上，初次部署时先把 `max-age` 设成较小值，比如 300（5 分钟），确保你没有把什么地方配错，也没有忘了让站点某部分支持 HTTPS。我们之所以建议先用这么小的值，是因为一旦浏览器记住了 `max-age`，我们自己是没法替用户取消它的；过期时间由用户浏览器控制，不由我们控制。
2. 等确认站点确实已经正确加固后，再逐步把 `max-age` 调大：先到 1 小时（3600），再到 1 周（604800）、1 个月（2592000），最后到 1 年（31536000）。这样一来，即使初期检查没发现问题，也还能留出修正空间。

3. 当项目的 max-age 已经达到 1 年或更长（31536000）后，就把项目提交到 HSTS preload 列表：hstspreload.org/。这有助于阻止主动攻击者拦截用户对站点发起的第一次请求。

警告：请谨慎选择 HSTS 策略时长

记住，HSTS 是个单向开关。它相当于向浏览器宣告：未来 N 秒内，你的网站只允许 HTTPS。不要把 max-age 设得超过你自己能长期维护的时长。浏览器并没有简便的方法可以把它撤销掉。

另外，HSTS 应该和前面说的“把所有页面重定向到 HTTPS”一起启用，而不是二选一。

警告：includeSubDomains 的额外风险

我们建议大家都用长时长的 HSTS，也建议使用 includeSubDomains。但请注意，尤其是在有大量遗留组件的项目里，这个组合必须配置得极其小心。

举个例子：假设我们新建了一个 Django 网站 example.com。站点本身已经启用了 HTTPS 和 HSTS，我们也测试过配置，一切正常，于是把时长拉到了一年。结果一个月后，才有人意识到 legacy.example.com 仍然是个在线生产服务，而且它根本不支持 HTTPS。我们这时把 includeSubDomains 从响应头里删掉，但已经晚了：公司里所有客户端早就记住了旧的 HSTS 头。

简而言之，在你认真考虑 includeSubDomains 之前，必须先完全搞清楚：这个域名之下到底还托管着什么。

29.6.3 HTTPS 配置工具

Mozilla 提供了一个 SSL 配置生成器：mozilla.github.io/server-side-tls/ssl-config-generator/，它可以作为你自己配置的起点。虽然它并不完美，但确实能加快 HTTPS 的搭建。正如我们的安全评审所说：“一般来说，任何 HTTPS 都比裸奔 HTTP 强。”

当你把服务器配好后，最好先用一台测试服务器，再去跑 Qualys SSL Labs 的服务器测试：ssllabs.com/ssltest/index.ht 看看你做得怎么样。一个挺有意思的安全小游戏，就是努力刷到 A+。尤其是按 Two Scoops of Django 官方规则，拿到这种分数的奖励，是可以去本地最喜欢的冰淇淋店庆祝一趟。

29.7 使用 Allowed Hosts 校验

在生产环境中，你必须在 settings 里把 ALLOWED_HOSTS 设为允许的 host / domain name 列表，否则就会抛出 SuspiciousOperation 异常。这是一项安全措施，用来防止攻击者利用伪造的 HTTP host header 发请求。

我们建议你避免在这里使用通配符。更多信息请阅读 Django 文档里关于 ALLOWED_HOSTS 和 get_host() 的部分：

- docs.djangoproject.com/en/3.2/ref/settings/#allowed-hosts
- docs.djangoproject.com/en/3.2/ref/request-response/#django.http.HttpRequest.get_host

29.8 对会修改数据的 HTTP 表单始终使用 CSRF 保护

Django 内建了易于使用的跨站请求伪造（CSRF）保护，而且默认会通过 middleware 在全站启用。务必确保那些会修改数据的表单使用 POST 方法，这样服务器端默认就会启用 CSRF 防护。唯一合理的 GET 例外，是搜索表单，因为让用户在 URL 中直接看到搜索参数通常是有帮助的。关于这个问题，我们在 13.3 节《对会修改数据的 HTTP 表单始终使用 CSRF 保护》里还有更强的建议。

29.9 防御跨站脚本（XSS）攻击

XSS 攻击通常发生在用户输入了恶意 JavaScript，而这些内容随后又被直接渲染进模板里。当然，这不是唯一方式，但却是最常见的一种。幸运的是，Django 默认会转义 <、>、'、" 和 &，而这正是正确 HTML escaping 所需的关键内容。

下面这些建议来自 Django 安全团队：

29.9.1 优先使用 format_html，不要用 mark_safe

Django 确实允许开发者把某些字符串显式标记为安全内容，但这等于把 Django 自己的保护栏杆给拆掉了。更好的替代方案，是 `django.utils.html.format_html`。它很像 Python 的 `str.format()`，但专门用于构造 HTML fragments。所有 `args` 和 `kwargs` 都会先被转义，再传给 `str.format()` 做拼接。

参考：`docs.djangoproject.com/en/3.2/ref/utils/#django.utils.html.format_html`

29.9.2 不要允许用户设置单独的 HTML 标签属性

如果你允许用户单独设置 HTML 标签的属性，就等于给了他们一个入口，让他们往里注入恶意 JavaScript。

29.9.3 对供 JavaScript 消费的数据使用 JSON 编码

与其想办法把 Python 结构直接 dump 到模板里，不如老老实实依赖 JSON 编码。这不仅更容易和客户端 JavaScript 集成，也更安全。要做到这一点，请始终使用 `json_script` filter。

参考：`docs.djangoproject.com/en/3.2/ref/templates/builtins/#json-script`

29.9.4 小心那些“诡异 JavaScript”

由于 JavaScript 的语义相当古怪，理论上可以只靠极少数字符就构造出语法合法、还能执行的程序。根据 `feld.to/unusual-javascript`，普通外观的 JavaScript 甚至可以被转换成只由六个字符组成的一套“字母表”（再加上加号、感叹号、左右中括号和左右小括号）。

警告：NSFW：`feld.to/unusual-javascript` 不适合在工作环境打开

`feld.to/unusual-javascript` 指向的那个 URL，本身可能违反教育机构或公司环境的浏览政策。

29.9.5 添加 Content Security Policy 头

Content Security Policy，也就是 CSP，提供了一种标准机制，让你声明：浏览器在某个网站上被允许加载哪些来源的内容。被覆盖的类型包括 JavaScript、CSS、HTML frames、web workers、字体、图片、可嵌入对象（如 Java applets、ActiveX）、音视频文件，以及其他 HTML5 特性。可选地，你还可以为 CSP 配置一个 violation-report URL，由项目自己或第三方服务来收集违规报告。

- `en.wikipedia.org/wiki/Content_Security_Policy`
- `github.com/mozilla/django-csp`

29.9.6 补充阅读

XSS 的攻击路径并不只有这些，所以持续学习非常重要。

- `docs.djangoproject.com/en/3.2/ref/templates/builtins/#escape`
- `en.wikipedia.org/wiki/Cross-site_scripting`

29.10 防御 Python 代码注入攻击

我们曾被请去协助一个存在安全问题的项目。那个站点收到的请求，会先把 `django.http.HttpRequest` 对象通过某种“创造性”的 `str()` 用法直接转成字符串，再存进数据库表里。之后，系统会定期把这些归档请求从数据库里取出来，再通过 `eval()` 函数把它们还原成 Python dict。这样一来，站点在任何时候都可能执行任意 Python 代码。

不用说，这个关键安全漏洞一被发现，就立刻被清掉了。这件事再次说明：无论 Python 和 Django 自身多么安全，我们都必须意识到，有些做法就是危险得离谱。

29.10.1 那些会执行代码的 Python 内建函数

警惕 `eval()`、`exec()` 和 `execfile()` 这些内建函数。如果你的项目允许任意字符串或文件被传进这些函数中的任何一个，你就是把系统敞开着等人攻击。

更多内容请阅读 Ned Batchelder 的《Eval Really Is Dangerous》:`nedbatchelder.com/blog/201206/eval_really_is_dange`

29.10.2 那些会执行代码的 Python 标准库模块

“永远不要 unpickle 任何可能来自不可信来源、或者可能已经被篡改过的数据。”

– `docs.python.org/3/library/pickle.html`

如果某些数据可能被用户修改过，就绝不应该用 Python 标准库里的 `pickle` 模块去反序列化它。一般规则就是：无论任何原因，都尽量避免接受来自用户的 pickled 值。关于 `pickle` 和安全的具体警告，见下：

- `lincolnloop.com/blog/playing-pickle-security/`
- `blog.nelhage.com/2011/03/exploiting-pickle/`

29.10.3 那些会执行代码的第三方库

如果你在用 PyYAML，请只使用 `safe_load()`。虽然在 Python 和 Django 社区里，除了 continuous integration 之外，YAML 并不算特别常见，但从其他服务收到这种格式并不罕见。所以，只要你要接收 YAML 文档，就只用 `yaml.safe_load()` 来加载它。

之所以要这样，是因为 `yaml.load()` 会允许你直接创建 Python objects，这非常糟糕。正如 Ned Batchelder 所说，`yaml.load()` 真该被改名成 `yaml.dangerous_load()`:`nedbatchelder.com/blog/201302/war_is_peace.html`

29.10.4 小心基于 Cookie 的 Sessions

大多数 Django 站点通常都使用数据库型或缓存型 sessions。它们的工作方式是：在 cookie 里只存一个经过哈希处理的随机值，这个值作为 key 去指向真正的 session 数据，而真实数据则保存在数据库或缓存里。这样做的好处是，发给客户端的只是 session 数据的 key，因此恶意开发者很难直接突破 Django 的 session 机制。

不过，Django 也可以构建成使用基于 cookie 的 sessions，即把 session 数据完整存放在客户端机器上。虽然这样会稍微减少服务器的存储需求，但它也会带来一些足以让人保持高度警惕的安全问题。具体来说：

1. 用户有可能直接读到基于 cookie 的 session 内容。
2. 如果攻击者拿到了项目的 `SECRET_KEY`，而你的 session serializer 又是基于 JSON 的，那么他们就获得了伪造 session 数据的能力；一旦站点使用了认证，这意味着他们可以冒充任意用户。
3. 如果攻击者拿到了项目的 `SECRET_KEY`，而你的 session serializer 又是基于 `pickle` 的，那么他们不仅能伪造 session 数据，还能执行任意代码。换句话说，他们不仅可以获得新的权限，还可以直接上传并运行 Python 代码。如果你正在使用、或考虑使用基于 `pickle` 的 sessions，请务必读下面那条提示。
4. 这种配置的另一个缺点，是 sessions 无法被可靠地失效化，除非等它自然过期。你可以试着在浏览器里用新值覆盖旧 cookie，但没法强制攻击者采用它；如果他们继续带着旧 cookie 发请求，session backend 根本分辨不出来。

提示：基于 Cookie 的 Sessions 请用 JSON

Django 默认的 cookie serializer 是基于 JSON 的，这意味着即便攻击者拿到了项目的 `SECRET_KEY`，他们也没法借此执行任意代码。如果你决定自己写 cookie serializer，那么格式也一定要坚持使用 JSON。永远、永远不要使用那个可选的 `pickle` serializer。

关于这个主题的资源：

- docs.djangoproject.com/en/3.2/topics/http/sessions/#session-serialization
- docs.djangoproject.com/en/3.2/ref/settings/#session-serializer
- http://bit.ly/2plfHqU：Threatpost.com 上关于 cookies 的文章
- yuiblog.com/blog/2007/03/01/performance-research-part-3/

另一件值得考虑的事是：基于 cookie 的 sessions 还可能成为一种客户端性能瓶颈。把 session 数据从服务端发到客户端，通常问题不大；但从客户端再传回服务端就慢得多。这几乎就是所有网民都熟悉的下载速度和上传速度差异。

总的来说，我们会尽量避免使用基于 cookie 的 sessions。

29.11 用 Django Forms 校验所有传入数据

Django forms 应该被用来校验所有进入项目的数据，包括那些来自非 Web 来源的数据。这样做既能保护数据完整性，也是应用安全的一部分。关于这点，我们在 13.1 节《使用 Django Forms 校验所有传入数据》中已经讲过。

提示：用 DRF Serializers 替代 Django Forms

Django REST Framework 的校验机制和 Django forms 一样构造良好、也一样安全。如果你对 DRF 更熟，那么用 serializers 来校验所有输入数据完全没有问题。

29.12 在支付字段上关闭自动补全

你应该在那些通往支付流程的 HTML 字段上，关闭浏览器的 autocomplete 功能。这包括信用卡号、CVV、PIN、信用卡到期日等等。原因很简单：很多人会在公共电脑上，或在公共场所使用自己的私人电脑。

作为参考，Django forms 可以很轻松地做到这一点：

示例 28.3：在表单字段中关闭自动补全

Code (python)

```
from django import forms


class SpecialForm(forms.Form):
    my_secret = forms.CharField(
        widget=forms.TextInput(attrs={'autocomplete': 'off'})
    )
```

对于任何可能在公共场景中使用的站点，例如机场里的自助设备，甚至可以考虑把字段直接换成 PasswordInput：

示例 28.4：将公开环境下的 Widget 改为 PasswordInput

Code (python)

```
from django import forms


class SecretInPublicForm(forms.Form):
    my_secret = forms.CharField(widget=forms.PasswordInput())
```

29.13 谨慎处理用户上传文件

如果你想完全安全地提供用户上传内容，唯一真正彻底的方式，是把它们放到一个完全独立的域名上。无论好坏，绕过文件类型校验器的方法几乎是无限多的。这也是为什么安全专家会推荐使用内容分发网络（CDNs）：

它们可以作为潜在危险文件的存储位置。

如果你必须允许任意文件类型被上传和下载，那至少要确保服务器使用 `Content-Disposition: attachment` 头，这样浏览器就不会把这些内容直接内联显示出来。

29.13.1 当 CDN 不是选项时

如果做不到走 CDN，那么上传文件必须被保存到一个“不允许执行”的目录里。另外，至少还要确保 HTTP 服务器被配置成：用正确的 image content type headers 提供图片，并且把允许上传的扩展名限制在一个白名单子集之内。

这里对 web server 配置一定要格外小心，因为恶意用户可能会通过上传可执行文件，例如 CGI 或 PHP 脚本，再直接访问对应 URL 来攻击你的网站。这并不能解决所有问题，但总比默认配置强。

具体怎么配，请查阅你所用 web server 的文档；如果你用的是平台即服务，也请查它们关于静态资源和用户上传文件应如何存储的说明。

29.13.2 Django 与用户上传文件

Django 里有两个允许用户上传内容的 model field：`FileField` 和 `ImageField`。它们自带了一些内建校验，但 Django 文档也明确强烈建议你“密切关注这些文件被上传到哪里，以及它们到底是什么类型，以免留下安全漏洞。”

如果你只接受某几种文件类型，那就尽你所能，确保用户真的只上传这些类型的文件。例如，你可以：

- 使用 `python-magic` 库检查上传文件的 headers：`github.com/ahupp/python-magic`
- 使用专门针对该文件类型的 Python 库去验证文件本身。虽然这点文档里没明说，但你如果去翻 `ImageField` 的源码，就会看到 Django 是如何借助 PIL 验证“上传的图片文件确实是图片”的。
- 使用 `defusedxml`，而不是 Python 原生 XML 库或 `lxml`。见 28.21 节《使用 defusedxml 防范 XML Bomb 攻击》。

警告：这里不能指望自定义 validators 解决问题

不要只是写一个自定义 validator，然后就以为它能在危险发生前把上传文件校验干净。自定义 validators 是在字段内容已经先通过字段自己的 `to_python()` 方法被转换成 Python 对象之后才运行的。

如果上传文件本身就带有恶意内容，那么任何发生在 `to_python()` 之后的校验，都可能已经太迟了。

延伸阅读：

- `docs.djangoproject.com/en/3.2/ref/models/fields/#filefield`
- `youtube.com/watch?v=HS8KQbswZkU`：Tom Eastman 在 PyCon AU 上关于《安全处理用户上传文件这门危险而精妙的艺术》的演讲，如果你在做涉及文件上传的网站，这几乎是必看内容。

29.14 不要使用 ModelForms.Meta.exclude

在使用 `ModelForm` 时，请始终使用 `Meta.fields`。绝不要使用 `Meta.exclude`。使用 `Meta.exclude` 被认为是严重的安全风险，具体来说，它会造成一种 Mass Assignment Vulnerability。我们再怎么强调都不为过：不要这样做。

之所以要避开 `Meta.exclude`，一个常见原因是：它的行为隐式地允许“除你列出的字段之外，其他所有 model fields 都可以被修改”。一旦 model 在表单写好之后又发生变化，你就必须记得同步修改表单。如果忘了让表单跟上 model 的变化，后果可能非常灾难。

下面用一个例子来展示这种错误是怎么发生的。先从一个简单的冰淇淋店 model 开始：

示例 28.5：示例店铺 Model

Code (python)

```
# stores/models.py
from django.conf import settings
from django.db import models


class Store(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    owner = models.ForeignKey(settings.AUTH_USER_MODEL)
    # 假设还有 10 个字段，负责地址和联系方式。
```

下面是为这个 model 定义 ModelForm 字段的错误方式：

示例 28.6：隐式定义表单字段

Code (python)

```
# DON'T DO THIS!
from django import forms

from .models import Store


class StoreForm(forms.ModelForm):

    class Meta:
        model = Store

        # DON'T DO THIS: 隐式定义字段。
        #               太容易出错了！
        excludes = ("pk", "slug", "modified", "created", "owner")
```

而下面才是为同一个 ModelForm 明确定义字段的正确方式：

示例 28.7：显式定义表单字段

Code (python)

```
from django import forms

from .models import Store


class StoreForm(forms.ModelForm):

    class Meta:
        model = Store
        # 显式指定我们想暴露的字段
        fields = (
            "title", "address_1", "address_2", "email",
            "usstate", "postal_code", "city",
        )
```

第一个例子因为打字少，看起来似乎更省事，但其实并不是。因为只要你给 model 新增字段，就必须在多个位置跟踪这个字段（一个 model，加一个或多个 forms）。

我们把这件事继续演示下去。假设项目上线后，我们决定需要支持“店铺共同所有者”。这些 co-owners 拥有和 owner 完全一样的权限：能访问账号信息、改密码、下订单、填写银行信息。于是 Store model 新增了一个字段：

示例 28.8：新增共同所有者字段

Code (python)

```
# stores/models.py
from django.conf import settings
from django.db import models


class Store(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    owner = models.ForeignKey(settings.AUTH_USER_MODEL)
    co_owners = models.ManyToManyField(settings.AUTH_USER_MODEL)
    # 假设还有 10 个字段，负责地址和联系方式。
```

前面那个被我们明确警告不要使用的表单写法，会要求我们记得把新加的 co_owners 字段同步排除掉。只要忘了，任何能访问该店铺 HTML 表单的人，就都能添加或移除 co-owners。一个表单你也许还记得住，但如果同一个 model 对应不止一个 ModelForm 呢？在复杂应用里，这一点都不罕见。

反过来，在第二种示例里，我们用的是 Meta.fields，因此我们清清楚楚知道：每个表单到底打算处理哪些字段。修改 model 本身，并不会改变表单实际暴露出去的字段范围；于是我们也就能睡得更安稳，因为冰淇淋店的数据更安全了。

29.14.1 批量赋值漏洞

这一节里描述的问题，就叫作 Mass Assignment Vulnerability。

这类问题会在某些模式中出现，比如 Active Record 这类本来是为了赋能开发者的设计，却反过来给 Web 应用制造了安全风险。解决办法，就是我们在本节一直主张的那种做法：显式定义哪些字段可以被修改。

更多内容见：n.wikipedia.org/wiki/Mass_assignment_vulnerability

29.15 不要使用 ModelForms.Meta.fields = "__all__"

这会把 model 里的每一个字段都包括进表单里。它是个捷径，而且是危险的捷径。它和 28.14 节《不要使用 ModelForms.Meta.exclude》里描述的问题非常相似；即便你额外写了自定义校验代码，它依旧会让项目暴露在基于表单的 Mass Assignment Vulnerabilities 下。我们强烈主张尽量避开这种写法，因为我们认为：你根本不可能把所有输入变体都堵干净。

29.16 小心 SQL 注入攻击

Django ORM 会生成经过正确转义的 SQL，因此能保护你的网站，避免用户试图执行恶意、任意的 SQL 代码。

不过，Django 也允许你绕过 ORM，直接更底层地访问数据库。当你使用这项能力时，一定要格外小心，确保 SQL 被正确转义。特别要注意 Django 中下面这些组件：

- ORM 的 `.raw()` 方法。
- ORM 的 `.extra()` 方法。
- 直接访问数据库 cursor。

参考：

- `docs.djangoproject.com/en/3.2/topics/security/#sql-injection-protection`

29.17 不要存储不必要的数据

有些数据，我们应该出于财务和法律原因避免存储。

29.17.1 永远不要存信用卡数据

除非你对 PCI-DSS 安全标准（`pcisecuritystandards.org`）有足够深入的理解，并且也有足够的时间、资源和资金去完成 PCI 合规验证，否则存储信用卡数据就是过大的责任风险，应该避免。

更好的做法，是使用 Stripe、Braintree、Adyen、PayPal 等第三方服务，由它们替你存储这些信息，再通过特殊 token 来引用数据。大多数这类服务都有很好的教程，对 Python 和 Django 也都很友好，非常值得投入时间去集成。

提示：主动学习 PCI 合规

Ken Cochrane 写过一篇非常好的 PCI 合规文章，请务必阅读：`kencochrane.net/blog/2012/01/developers-guide-to-p`

提示：去读开源电商方案的源码

如果你打算使用现有某个开源 Django 电商方案，请仔细检查它是如何处理支付的。如果它把信用卡数据存进了数据库里，哪怕是加密存储，也请换一个方案。

29.17.2 除非法律要求，否则不要存储 PII 或 PHI

PII 是 Personally Identifying Information 的缩写，PHI 是 Protected Health Information 的缩写。这类数据构成了我们线上身份的重要部分，而且一旦被利用，就可能被拿去访问其他账号，甚至造成更糟后果。以美国为例，Social Security Number（SSN）和州身份证号码这类数据，都应当尽量避免存储，除非法律明确要求。

即便法律确实要求你存，只要有可能把它存成单向哈希，也请务必这么做。

如果你在一个本来没有法律义务存储 PII 的项目里发现了这类数据，请立刻把它作为问题提给项目负责人。无论通过技术手段还是社会工程，这类数据一旦泄露，都会在法律和财务层面给组织带来灾难性后果。信用卡泄露后通常还能补发和注销，但 PII 并不能这样轻易“换掉”。

至于 PHI，也就是医疗数据，我们的警告会更严厉。具体来说，相关人员都可能在 HIPAA 第二篇章的规定下同时面临民事和刑事追责。参见：`en.wikipedia.org/wiki/HIPAA#Security_Rule`

虽然这条规则对应的是美国项目，但许多国家也有类似法规。举个例子，作者就认识有人因为 PHI 泄露而在肯尼亚被传唤出庭。

29.18 监控你的站点

定期检查 web server 的访问日志和错误日志。安装监控工具，并经常查看。对任何可疑活动都保持警觉。

29.19 让依赖保持最新

你应该始终把项目更新到与 Django 最新稳定版本、以及各个第三方依赖的最新稳定版本兼容。尤其当某次发布带有安全修复时，这一点更是如此。对此，我们推荐 `pyup.io`，它会自动把你的 requirements 文件与 PyPI 提

供的最新版本做比对。

“我把（这类服务）配置成针对不同动作分别工作：每周它会给我每个项目发一封邮件，告诉我哪些依赖已经过时；如果它发现某个版本有安全问题，就会自动在 GitHub 上创建一个 pull request，这样测试会自动运行，而我也就能更快部署。”

- Sasha Romijn，Django core dev，也是 Two Scoops of Django 1.8 的安全评审

关于 Django 自身更新的有用链接：

- Django 官方博客：djangoproject.com/weblog/
- 官方 django-announce 邮件列表：groups.google.com/forum/#!forum/django-announce

29.20 防止 Clickjacking

Clickjacking 指的是：恶意网站诱骗用户点击另一个站点中被隐藏框架或 iframe 遮住的元素。一个例子是：某个站点上看起来像“社交媒体登录按钮”的东西，实际上是另一个站点上的购买按钮。

Django 已经提供了防止这类事情发生的说明和组件：

- docs.djangoproject.com/en/3.2/ref/clickjacking/

29.21 用 defusedxml 防范 XML Bomb 攻击

针对 XML 库的攻击并不新鲜。比如那个名字听起来很滑稽、但破坏性极强的“Billion Laughs”攻击（http://en.wikipedia
早在 2003 年就被发现了。

不幸的是，Python 和许多其他编程语言一样，并不会默认防住这类 XML 攻击。更进一步说，lxml 这类第三方 Python 库，至少也已知存在 4 种基于 XML 的攻击面。关于 Python 以及其相关库的漏洞列表，请见：https://feld.to/2KKwuMq

好在 Christian Heimes 创建了 defusedxml。这是一个专门用来给 Python 核心 XML 库以及某些第三方库（包括 lxml）打补丁的 Python 库。

更多内容请阅读：

- https://pypi.org/project/defusedxml

29.22 考虑启用双因素认证

双因素认证（2FA）要求用户通过两个独立的识别手段组合在一起完成认证。

对现代 Web 应用来说，这通常意味着：除了输入密码之外，还要再输入一段发送到移动设备上的值。另一种 One-Time Passwords（OTP）的做法，是把它们登记进 password manager 里。

2FA 的好处是：它在认证流程中额外加入了一个经常变化的因素，因此对任何涉及个人身份、金融或医疗要求的站点都非常有帮助。

缺点则在于：用户必须持有一台有电、而且能联网的移动设备，才能登录你的网站。对于那些未必总有电、也未必总能轻松联网的用户来说，这就不那么理想了。

- en.wikipedia.org/wiki/Two_factor_authentication
- https://pypi.org/project/django-two-factor-auth

包提示：在 2FA 产品和包里优先寻找 TOTP

TOTP 是 Time-based One-time Password Algorithm 的缩写，也就是：en.wikipedia.org/wiki/Time-based_One-tim
它是 Google Authenticator 以及许多其他服务所使用的开放标准。TOTP 不需要网络访问，这对某些类型的 Django 项目尤其有用。相比之下，SMS 方案则需要蜂窝网络，或依赖像 twilio.com 这样的第三方服务。

警告：2FA 恢复流程的问题

一个非常重要的问题，是当用户丢失 2FA token 或手机号时，应该怎么恢复。密码通常可以通过发送带密钥链接的邮件来重置；但如果 2FA token 也能通过电子邮件重置，那么“能进用户邮箱”就几乎重新退化成了唯一认证因素。常见做法包括：提供 TOTP 认证，同时用 SMS 作为回退；或者提供若干组需要用户自己妥善保管的 recovery codes。有些组织甚至只会在收到账号持有人的身份证扫描件之后，才重置这些 token。无论采用哪种方式，恢复流程都是必需的，所以请提前把这件事想清楚。

29.23 拥抱 SecurityMiddleware

这一章里我们已经多次提到 Django 内建的 `django.middleware.security.SecurityMiddleware`。无论是为了我们自己，还是为了用户，我们都应该认真拥抱并使用 Django 这个功能。

29.24 强制使用强密码

强密码绝不只是“一串字符”这么简单。它应该足够长，而且最好也足够复杂，包括标点、数字，以及大小写混合。让我们承诺：通过强制使用这类密码，来更好地保护用户。

那么，什么才算“最好的密码”？

我们的看法是：在这个时代，长度比复杂度更重要。一个只有 8 个字符、但混合了大小写、数字和特殊字符的密码，比起一整句 50 个字符、全部是小写字母的长句子，其实要容易被破解好几个数量级。更好的情况，是你有一段 30 到 50 个字符长的句子，并且其中还包含数字、大小写和特殊字符。

参考：`xkcd.com/936/`

质量	密码规格
差	只有 6-10 个字母字符
还行	至少 8 个字符，且包含大小写、数字和特殊字符
更好	至少 30 个字符，只有字母
最好	至少 30 个字符，同时包含大小写、数字和特殊字符

表 28.1：密码强度：长度 vs 复杂度

29.25 不要阻止用户复制 / 粘贴密码

有些站点强迫用户手动输入密码，而不允许复制 / 粘贴。这是一种非常糟糕的反模式。它会鼓励用户依赖那些容易记住、或者在多个站点反复使用的密码，而不是每个站点都用由 `1Password` 或 `LastPass` 等安全 password manager 生成的强而独特的密码。

29.26 给你的站点做一次安全体检

现在有不少服务可以对站点做自动化安全检查。它们并不是正式的安全审计，但作为免费手段，用来确认你的生产部署里没有明显巨大的安全窟窿，已经非常有价值。

`pyup.io` 的 Safety 库（`github.com/pyupio/safety`）会检查你已安装的依赖里是否存在已知安全漏洞。默认情况下，它使用开放的 Python 漏洞数据库 Safety DB；也可以通过 `--key` 选项升级为使用 `pyup.io` 的 Safety API。

```
TODO - consider adding snikt or other commercial tools
```

Mozilla 也提供了一个类似服务，虽然并不专门面向 Django，名叫 Observatory：`observatory.mozilla.org`

29.27 放出一个漏洞报告页面

在站点上公开说明：用户如果发现安全漏洞，应该如何向你报告，这会是个好主意。

GitHub 的《Responsible Disclosure of Security Vulnerabilities》页面就是一个很好的例子。它甚至还会通过公开名字来感谢问题报告者:help.github.com/articles/responsible-disclosure-of-security-vulnerabilities/

29.28 绝不展示顺序型主键

应该避免展示顺序型主键，原因包括：

- 它会把你的业务体量暴露给潜在竞争者或攻击者。
- 一旦展示这些值，就等于让不安全的直接对象引用（Insecure Direct Object Reference）变得极其容易被利用。
- 你还额外为 XSS 攻击提供了目标。

下面是几种在不暴露顺序型标识符的前提下查找记录的模式：

29.28.1 按 Slug 查找

在 Django 世界里，这是非常常见的做法。相关示例几乎成百上千。这对很多 Django 项目来说都是首选方式。不过，一旦你遇到重复 slug，它就会稍微变得棘手一些，比如出现 `vanilla`、`vanilla-2`、`vanilla-3`。这时你就该考虑改用别的方法了。

29.28.2 UUIDs

Django 自带非常有用的 `models.UUIDField`。虽然在大型分布式系统里，用它做主键确实有其合理场景，但它同样也很适合做公开查找字段。下面是一个示例 model：

示例 28.9：使用 UUID 做公开查找

Code (python)

```python
import uuid
from django.db import models


class IceCreamPayment(models.Model):
    uuid = models.UUIDField(
        unique=True,
        default=uuid.uuid4,
        editable=False,
    )

    def __str__(self):
        return str(self.pk)
```

下面是我们如何使用这个 model：

示例 28.10：按 UUID 查找支付记录

Code (python)

```python
>>> from payments import IceCreamPayment
>>> payment = IceCreamPayment()
>>> IceCreamPayment.objects.get(id=payment.id)
```

```
<IceCreamPayment: 1>
>>> payment.uuid
UUID('0b0fb68e-5b06-44af-845a-01b6df5e0967')
>>> IceCreamPayment.objects.get(uuid=payment.uuid)
<IceCreamPayment: 1>
```

提示：把 UUID 作为主键

有些人喜欢直接用 UUID 作为主键。事实上，这些年我们自己也在多个项目里这么做过。下面是我们和 Django 安全团队成员对这件事的一些观察：

UUID 作为主键，确实能简化数据层面的安全设计。原本你要处理两个字段（id 和 uuid），现在只剩一个 id。

但另一方面，顺序型 ID 是人类还能勉强记住的，UUID 则不是。如果某个 model 只允许通过 UUID 访问，那么和数据打交道就会变得稍微更麻烦一些。想抓某条记录时，往往需要完整的 UUID 字符串。

此外，使用 UUID 做查找也存在性能上的权衡。你是在用速度换安全。参考：percona.com/blog/2019/11/22/uuids-are-p

这一切综合起来说明：把 UUID 用作主键（以及查找字段）是一件应该被认真权衡的事。如果一个项目未来预期要承载数十亿条记录，那么就该考虑其他方式，包括继续使用顺序型 ID。

警告：对顺序 ID 做混淆的危险

Slug 和 UUID 各自都有短板。基于 slug 的做法很容易发生冲突，于是就会出现类似 vanilla、vanilla-2、vanilla-3 这种情况。UUID 则简单说就是：太长，而且大多数人根本记不住。那还能怎么办？

你当然可以去混淆顺序型 ID。但我们不推荐。为什么？

短答案：混淆不是隐藏顺序 ID 的有效方式。

长答案：混淆数字的方法非常多，从 base64 编码，到使用 hashids 库，都属于这一类。它们会把数字编码成字母数字串，再在需要时反向解码。这样做不但能把数字藏起来，还能把它缩短。听起来很棒，对吧？

问题在于：所有混淆顺序型 ID 的方法，本质上都不安全。base64 编码几乎是随手可逆的；像 hashids 这种库，则可以被暴力破解，或者被任何具备一定密码学知识的人直接攻破。参见：carnage.github.io/2015/08/cryptanalysis-of-

总结一下：如果你真的想隐藏顺序型标识符，就别指望混淆。

29.29 将密码哈希器升级到 Argon2

“……Django 默认的密码哈希器是 PBKDF2。虽然它作为最低公分母选项仍然算可接受，但其实多年以来一直都有更好的选择存在。”

- James Bennett，Django Core Developer，也是 Two Scoops of Django 的安全评审

PBKDF2 之所以是 Django 默认值，是因为它由 Python 标准库直接支持，所以不需要第三方包。对 Django 原生支持的方案来说，Argon2 是更好的选项。安装说明和参考资料如下：

- https://docs.djangoproject.com/en/3.2/topics/auth/passwords/#using-argon2-with-django
  说明如何在 Django 中安装并使用 Argon2。
- https://docs.djangoproject.com/en/3.2/topics/auth/passwords/#argon2
  说明如何自定义 Argon2 参数。这个属于高级内容，大多数项目并不需要。
- latacora.micro.blog/2018/04/03/cryptographic-right-answers.html
  这是一份很好懂的答案清单，用来回答那些“不专做安全的开发者”经常会被问到的密码学问题。

29.30 从外部来源加载静态资源时使用 SRI

项目从外部来源加载库并不罕见。例如，jQuery、Google Fonts、Bootstrap、Tailwindcss，甚至 React 和 Vue，都常常通过 CDNJS、unpkg、Google Hosted Libraries 等 CDN 来加载。不幸的是，如果没有 SRI（Subresource Integrity）

去确认资源来源，攻击者就能替换或篡改用户所使用的资源。换句话说，下面这样不要做：

示例 28.11：天真地加载 Bootstrap 资源

Code (html)

```
<!-- DON'T DO THIS - loading static assets without SRI -->
<link rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">

<script
    src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>
<script
    src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"></script>
<script
    src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></script>
```

这也是为什么我们建议：只要从外部项目引入 CSS、JavaScript 或 web fonts，就尽量带上 subresource integrity hash。CDNJS、jQuery 项目、Bootstrap 项目，以及数量少得令人吃惊的一小部分其他项目，都已经提供了这项能力。

下面是官方 Bootstrap 站点给出的使用 SRI 的样子：

示例 28.12：加载 Bootstrap 资源时使用 SRI

Code (html)

```
<!-- Loading Static Assets with SRI -->
<link rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
      integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
      crossorigin="anonymous">

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
        integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
        crossorigin="anonymous"></script>

<script
    src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
    integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
    crossorigin="anonymous"></script>

<script
    src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
    integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
    crossorigin="anonymous"></script>
```

如果某个项目在安装说明里没有附带 SRI hashes，我们建议你改用 CDNJS 加载它们，因为 CDNJS 会在“copy”下拉框里直接提供这些值。只要选择“copy link tag”即可。

推荐工具和参考资料：

- `cdnjs.com`：它为所托管的所有库都提供 SRI hashes。
- `developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity`
  Mozilla 关于 SRI 的文档。
- `w3.org/TR/SRI/`：官方规范。

29.31 参考我们的安全设置附录

和 Django 相关的安全事项太多，追踪起来并不容易。光这一章就将近 30 页，而且我们一开始就明确说过，它并不是绝对完整的参考手册。

为了让事情更清楚，我们专门写了附录 G：《Security Settings Reference》。在那里面，我们汇总了大量关于如何更好配置 Django 项目安全设置的重要信息。

29.32 回头看看安全包清单

在附录 A《Packages》的安全小节里，我们列了十多个和安全有关的 package，它们都可能对你的站点产生实质帮助。其中有些在本章已经提到，另一些则只出现在那个附录章节里。

29.33 持续跟进一般性安全实践

本章最后，我们留下一些很朴素、但很重要的常识性建议。

第一，要记住：无论在 Django 社区内部，还是整个技术世界里，安全实践都在不断演化。请订阅 `groups.google.com/forum` 并定期查看 Twitter、Hacker News 以及各种安全博客。

第二，要记住：安全最佳实践绝不只局限于 Django 自己。你应该研究整个 Web 应用技术栈中每一个组成部分的安全问题，并跟进对应的信息源，让自己保持最新认知。

提示：关于安全的好书与好文章

Paul McMillan，Django 核心开发者、安全专家，也是 Two Scoops 的评审，推荐下面这些书：

- 《The Tangled Web: A Guide to Securing Modern Web Applications》：
  `amzn.to/1hXAAyx`
- 《The Web Application Hacker’s Handbook》：
  `amzn.to/1dZ7xEY`

另外，我们也推荐下面这个参考站点：

- `wiki.mozilla.org/WebAppSec/Secure_Coding_Guidelines`

29.34 总结

请把这一章当作 Django 安全实践的出发点，而不要把它当作最终权威手册。更多安全主题，请继续查看 Django 官方文档里那份附加安全主题列表：`docs.djangoproject.com/en/3.2/topics/security/#additional-security-t`

Django 之所以拥有不错的安全纪录，靠的是社区的勤勉和对细节的重视。安全恰恰是那种特别值得主动求助的领域之一。如果你对任何事情感到困惑，请开口提问，也请转向 Django 社区里的其他人寻求帮助。

第三十章 日志：到底是拿来干什么的?

日志就像 rocky road 口味冰淇淋。你要么离不开它，要么平时把它忘得一干二净，偶尔才会纳闷这东西为什么存在。

凡是做过大型生产项目、也经历过高压需求的人，都懂得这些事有多重要：恰当地使用不同日志级别、为不同模块建立专属 logger、认真记录关键事件的信息，并在记录这些事件时带上应用当时状态的更多细节。

日志看起来也许不算光鲜，但请记住，它正是构建极其稳定、健壮、可扩展，并且能从容应对异常负载的 Web 应用的秘诀之一。日志不仅可以用来调试应用错误，也可以用来跟踪有意思的性能指标。

把异常活动记进日志、并定期检查日志，也是在保障服务器安全时不可缺少的一环。上一章我们讲过，要定期查看服务器访问日志和错误日志。别忘了，应用日志也可以以类似方式使用，无论是追踪登录失败尝试，还是发现应用层面的异常行为。

30.1 应用日志与其他日志

这一章重点讲应用日志。凡是来自你的 Python Web 应用、并被写入日志文件的数据，都算应用日志。

除了应用日志之外，你也应该知道还有其他类型的日志，而且所有服务器日志都必须使用并定期查看。你的服务器日志、数据库日志、网络日志等等，都会为生产系统提供关键洞察，所以它们同样重要。

30.2 为什么要花心思做日志?

当堆栈跟踪和现有调试工具都不够用时，日志就是你的首选工具。只要系统里有多个活动部件彼此交互，或者存在不可预测的情况，日志就能帮你看清到底发生了什么。

你手上的日志级别有 DEBUG、INFO、WARNING、ERROR 和 CRITICAL。下面我们就来看看，分别在什么情况下适合使用这些级别。

30.3 什么时候该用哪个日志级别

在生产环境之外，你大可以把所有日志级别都用起来。日志级别由项目的 settings 模块控制，因此我们也可以针对负载测试和大规模用户测试之类的场景按需微调这个建议。

在生产环境里，我们建议除了 DEBUG 之外，其他级别都用起来。

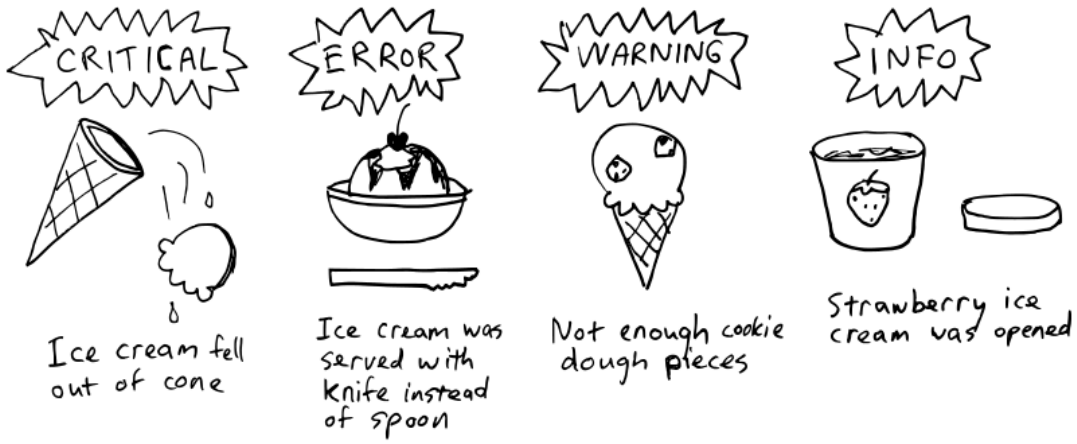

图 30.1: 图 29.1：冰淇淋语境下 CRITICAL / ERROR / WARNING / INFO 日志的合适用法。

由于无论在生产还是开发环境中，CRITICAL、ERROR、WARNING 和 INFO 这些日志都会被记录下来，所以排查有缺陷的代码时，需要为了“看清楚发生了什么”而去改动代码的情况会少很多。这一点很重要，因为开发者为了修一个问题临时加进去的调试代码，本身就可能制造出新问题。

本节余下部分会逐一说明各个日志级别的用法。

30.3.1 用 CRITICAL 记录灾难性故障

只有在发生灾难性问题、并且必须立刻处理时，才使用 CRITICAL 日志级别。

举个例子，如果你的代码依赖某个内部 Web 服务可用，而这个 Web 服务又是站点核心功能的一部分，那么只要这个服务不可访问，你就可以按 CRITICAL 级别记录下来。

Django 核心代码里从来不用这个级别，但在你自己的代码中，只要某个极其严重的问题确实可能发生，就完全应该使用它。

30.3.2 用 ERROR 记录生产错误

我们来看 Django core 里什么时候适合用 ERROR 级日志。在 Django core 里，ERROR 日志级别用得非常克制。它有一个极其重要的使用点：只要有异常被抛出且没有被捕获，这个事件就会由 Django 用下面这段代码记录下来：

示例 29.1：记录生产错误

Code (python)

```python
# Taken directly from core Django code.
# Used here to illustrate an example only, so don't
# copy this into your project.
logger.error('Internal Server Error: %s', request.path,
    exc_info=exc_info,
    extra={
        'status_code': 500,
        'request': request
    }
)
```

Django 是怎么把这件事用起来的？当你的 settings 里 `DEBUG=False` 时，所有列在 `ADMINS` 里的收件人都会立刻收到一封邮件，内容包括：

- 错误的简要说明。
- 出错位置的完整 Python traceback。
- 导致这次错误的 HTTP 请求信息。

如果你收过这种邮件通知，就知道在最需要的时候，ERROR 日志到底有多有用。

同样地，我们也建议你在那些“值得发邮件通知给你或站点管理员”的错误上使用 ERROR 级日志。当你的代码自己捕获了异常时，也要尽可能多地把信息记下来，方便后续定位问题。

比如，当某个视图访问所需的第三方 API 失败时，就可能抛出异常。你在捕获这个异常后，可以把一条有帮助的消息以及 API 的失败响应（如果有的话）一起记下来。

30.3.3 用 WARNING 记录较低优先级的问题

这个级别很适合记录那些“不太正常，而且可能不妙，但还没有糟到 ERROR 级别”的事件。

比如，如果你用 `django-admin-honeypot` 搭了一个假的 `admin/` 登录表单，你也许会想把入侵者的登录尝试记到这个级别。

Django 在 CsrfViewMiddleware 的几个地方就使用了这个级别，去记录那些最终会导致 403 Forbidden 错误的事件。举例来说，当一个传入的 POST 请求缺少它自己的 csrf_token 时，这个事件会被这样记录：

示例 29.2：记录缺失的 CSRF

Code (python)

```
# Taken directly from core Django code.
# Used here to illustrate an example only, so don't
# copy this into your project.
logger.warning('Forbidden (%s): %s',
    REASON_NO_CSRF_COOKIE, request.path,
    extra={
        'status_code': 403,
        'request': request,
    }
)
```

30.3.4 用 INFO 记录有用的状态信息

我们建议把任何在分析问题时可能特别重要的细节都记在这个级别里，包括：

- 未在其他地方记录的重要组件的启动与停止。
- 对重要事件作出响应时发生的状态变化。
- 权限变更，例如用户被授予 admin 访问权。

除此之外，INFO 级别也很适合记录那些可能帮助你做性能分析的一般性信息。当你在追查应用里的瓶颈、做 profiling 时，这会是一个很好用的级别。

30.3.5 把与调试相关的消息记到 DEBUG

在开发环境里，凡是你会想为了调试而往代码里塞一个 print() 的地方，我们都建议改用 DEBUG 级日志，偶尔也可以用 INFO。

与其这样：

示例 29.3：用 print() 输出数据

Code (python)

```
from django.views.generic import TemplateView

from .helpers import pint_counter

class PintView(TemplateView):

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(**kwargs)
        pints_remaining = pint_counter()
        print(f'Only {pints_remaining} pints of ice cream left.')
        return context
```

我们会这样做：

示例 29.4：用日志输出数据

Code (python)

```
import logging
```

```
from django.views.generic import TemplateView

from .helpers import pint_counter

logger = logging.getLogger(__name__)

class PintView(TemplateView):

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(**kwargs)
        pints_remaining = pint_counter()
        logger.debug('Only %d pints of ice cream left.' %
            pints_remaining)
        return context
```

提示：不要在日志语句里使用 f-strings

截至 Python 3.9，f-strings 不能直接用于 logger 语句中。

在整个项目里到处撒 `print` 语句，会留下问题和技术债：

- 视所用 Web 服务器而定，一个被遗忘的 `print` 语句就可能把站点拖垮。
- `print` 语句不会被记录下来。要是你没看到它们，就等于错过了它们试图告诉你的信息。

与 `print` 语句不同，日志允许你设置不同的报告级别，也允许你为不同级别配置不同的响应方式。这意味着：

- 我们可以写下 DEBUG 级语句，把它们留在代码里，等代码部署到生产环境后也不用担心它们会造成实际影响。
- 响应方式既可以是邮件、日志文件、控制台和 stdout，也可以进一步做成像 Sentry 这样的应用所接收的 HTTP 推送请求。

不过也别把 debug 级日志写得太夸张。调试时临时加一些 `logging.debug()` 很好，但没必要把每一行代码都记一遍，把代码本身塞得乱七八糟。

30.4 在捕获异常时把 traceback 一起记下来

每当你记录异常时，通常都值得把这个异常的堆栈跟踪也一起记下来。Python 的 logging 模块原生支持这件事：

1. `Logger.exception()` 会自动带上 traceback，并按 ERROR 级别记录。
2. 如果你想用其他日志级别，就传可选关键字参数 `exc_info`。

下面是一个把 traceback 加进 WARNING 级日志消息里的例子：

示例 29.5：用 `exc_info` 捕获 traceback

Code (python)

```
import logging
import requests

logger = logging.getLogger(__name__)

def get_additional_data():
    try:
        r = requests.get('http://example.com/something-optional/')
    except requests.HTTPError as e:
```

图 30.2: 图 29.2：冰淇淋语境下 DEBUG 日志的合适用法。

```
        logger.exception(e)
        logger.debug('Could not get additional data',
            exc_info=True)
        return None
    return r
```

30.5 每个使用日志的模块都各自定义一个 logger

凡是在别的模块里要用日志，都不要从别处 import 一个现成的 logger 来复用。应该像下面这样，为当前模块单独定义一个 logger：

示例 29.6：每个模块一个 logger

Code (python)

```
# You can place this snippet at the top
# of models.py, views.py, or any other
# file where you need to log.
import logging

logger = logging.getLogger(__name__)
```

这样做的好处，是你可以只打开或关闭此刻真正需要的那一个 logger。如果生产环境里出现了一个本地无法复现的怪问题，你可以临时只为相关模块打开 DEBUG 日志。等问题定位出来后，再把生产里的那个 logger 关回去。

30.6 在本地把日志写入轮转文件

当你用 `startproject` 创建一个新的 Django 项目时，默认 settings 文件已经配置好：把 ERROR 及以上级别的日志邮件发送给 `ADMINS` 中列出的收件人。这是通过 Django 自带的一个 handler `AdminEmailHandler` 完成的。

除此之外，我们还建议把 INFO 及以上级别的日志也写入磁盘上的轮转日志文件。这样即便网络挂了，或者由于某些原因邮件无法发到管理员手里，磁盘日志依然有用。日志轮转还能防止日志无限膨胀、最终把磁盘空间吃满。

一个常见做法，是配合 `logging.handlers.WatchedFileHandler` 使用 UNIX 的 `logrotate` 工具来实现日志轮转。

请注意，如果你使用的是平台即服务，可能没法自己设置轮转日志文件。这种情况下，你也许得使用像 Loggly 这样的外部日志服务：`loggly.com`。

30.7 其他日志建议

- 按照 Django 关于 logging 的文档，在 settings 文件中控制日志行为：`docs.djangoproject.com/en/3.2/topics/logging/`
- 调试时，使用 Python logger 的 DEBUG 级别。
- 在 DEBUG 级别跑完测试后，再试着用 INFO 和 WARNING 级别跑一遍。你看到的信息变少了，反而可能更容易发现第三方库即将到来的弃用提示。
- 别等到太晚才开始补日志。等网站真的出故障时，你会庆幸自己早就把日志留好了。
- 对于 ERROR 及以上级别事件触发的邮件，你完全可以做更多有用的事情。比如配置一个 PagerDuty（`pagerduty.com`）账户，让它持续提醒你和你的团队，直到有人采取行动为止。

包提示：Logutils 提供了一些很有用的 handlers

Vinay Sajip 编写的 logutils 包里自带了不少很有意思的 logging handlers，包括：

- 在 Windows、Linux 和 Mac 下为控制台输出着色。
- 支持把日志写入队列。这在你希望先把日志消息排队，再交给像 `SMTPHandler` 这样较慢的 handler 处理时特别有用。
- 提供一些类，让你可以为日志消息编写单元测试。
- 一个增强版 `HTTPHandler`，支持通过 HTTPS 建立安全连接。

logutils 里有些最基础、但也最实用的特性，后来甚至被吸收进了 Python 标准库。

30.8 必读材料

- `docs.djangoproject.com/en/3.2/topics/logging/`
- `docs.python.org/3/library/logging.html`
- `docs.python.org/3/library/logging.config.html`
- `docs.python.org/3/library/logging.handlers.html`
- `docs.python.org/3/howto/logging-cookbook.html`

30.9 有用的第三方工具

- Sentry（`sentry.io`）会帮你聚合错误信息，作者本人、Dropbox、AirBnB 以及许多其他公司都信任它。他们的产品是开源的，也长期积极支持各种开发者的开源工作。我们怎么推荐都不嫌多。
- `loggly.com`（`loggly.com`）简化日志管理，并提供多种查询工具。

30.10 总结

Django 项目完全可以轻松用上 Python 自带的丰富日志能力。把日志和 handlers、分析工具结合起来，你立刻就会获得真正的力量。你可以借助日志来提升项目的稳定性与性能。

下一章我们会谈 signals。借助日志，signals 会更容易追踪、调试和理解。

第三十一章 Signals：使用场景与规避技巧

警告：本章仍在编写中

我们正在撰写这一章，接下来几天还会继续扩充内容。欢迎就应该覆盖的话题和条目向我们提建议，请提交到：`github.com/feldroy/two-scoops-of-django-3.x/issues`

第三十二章 那些零散工具该怎么办?

32.1 为这些工具代码建一个 Core App

有时候，我们最终会写出一些共享类，或者一些到处都能派上用场的小型通用工具。这些零零碎碎的东西并不真正属于某个特定 app。我们不会因为它“多少沾点边”就把它硬塞进某个随机 app 里，因为等真正要找它时，那样会特别难找。我们也不喜欢把它们当成“随机模块”扔在项目根目录下。

我们处理这类工具代码的方式，是把它们放进一个叫 `core` 的 Django app 中。这个 app 里的模块，专门放那些能在整个项目范围内复用的函数和对象。（其他开发者也有类似做法，只是可能会把这种 app 叫做 `common`、`generic`、`util` 或 `utils`。）

举个例子，假设我们的项目里既有一个自定义 model manager，也有一个会被多个 app 共用的自定义 view mixin。那么，`core` app 大致会长成这样：

示例 31.1：Core App 布局示例

Code (text)

```
core/
+-- __init__.py
+-- managers.py # contains the custom model manager(s)
+-- models.py
+-- views.py    # contains the custom view mixin(s)
```

提示：始终让 Core App 成为一个真正的 Django App

我们总是把 `core` 目录做成一个 Django app。因为走着走着，你几乎一定会遇到下面至少一种情况：

- 在 `core` 里放入非抽象模型。
- 让 admin auto-discovery 在 `core` 里正常工作。
- 在 `core` 里放 template tags 和 filters。

这样一来，如果我们要导入自己的 model manager 和 / 或 view mixin，就可以和导入其他东西一样，遵循同一种导入模式：

示例 31.2：从 Core App 导入

Code (python)

```
from core.managers import PublishedManager
from core.views import IceCreamMixin
```

32.2 用工具模块优化 App

这类模块和 helpers 基本是同义的，通常叫 `utils.py`，有时也叫 `helpers.py`。它们是我们放置函数和类的地方，用来把常见模式写得更短、更顺手。下面来看看为什么这会有帮助。

32.2.1 存放被多处使用的代码

有时我们会有一些函数或类，会在多个地方用到，但它们又不太适合放进 `models.py`、`forms.py`，或者其他那些名字非常明确的模块里。这种时候，我们就把这部分逻辑放进 `utils.py` 模块。

32.2.2 给 Models 瘦身

这件事最适合通过例子来说明。

我们经常使用 `Flavor` 模型。然后不停地往上加字段、加方法、加 property、加 classmethod。某一天，我们突然发现这个胖模型已经膨胀成了一个庞然巨物，代码超过一千行。调试和维护都变得困难起来。这时该怎么办？

我们会开始寻找那些方法（或者 properties、classmethods），看看它们的逻辑是否可以很容易地封装成函数，并放进 `flavors/utils.py`。原有的方法（或者 properties、classmethods）则退化成简单包装器，负责调用 `flavors/utils.py` 里的函数。这样得到的代码库结构会更分散、更利于复用，同时也更容易测试。

32.2.3 更容易测试

把逻辑从更复杂的构造中挪到独立模块里的函数中，还有一个副作用：测试会更容易。所谓“独立”，是指它通常在 app 内部被导入，而不是做 app 内 / 项目内层层交叉导入。这样业务逻辑负担会更轻，因此更容易为模块中真正存在的逻辑编写测试。

提示：尽可能让工具代码保持聚焦

无论是函数还是类，都不要让它承担多种行为或多种条件分支。每个工具函数都应该只把一件事做好，而且只做好这一件事。遵循 Don’t Repeat Yourself。不要造出那些只是把模型行为重复一遍的工具函数。

32.3 Django 自带的瑞士军刀

瑞士军刀是一种紧凑而实用的多功能工具。Django 里也有不少很有用的辅助函数，它们最合适的归宿就是 `django.utils` 包。你会很想直接钻进 `django.utils` 的源码里，看到什么就拿来用，但别这么干。那里面的大多数模块都是给 Django 内部使用的，它们的行为，甚至是否继续存在，都可能随着 Django 版本变化而变化。

更好的做法，是去看 `docs.djangoproject.com/en/3.2/ref/utils/`，确认其中哪些模块是稳定可用的。

提示：Malcolm Tredinnick 如何看待 Django 的 Utils 包

Django core 开发者 Malcolm Tredinnick 很喜欢把 `django.utils` 想成和蝙蝠侠工具腰带同一主题的东西：里面装满了不可或缺、并且在内部到处都会用到的工具。

if there are stable.

TIP: Malcolm Tredinnick on Django’s Utils Package.

Django core developer Malcolm Tredinnick liked to think of `django.utils` as being in the same theme as Batman’s utility belt: indispensable tools that are used everywhere internally.

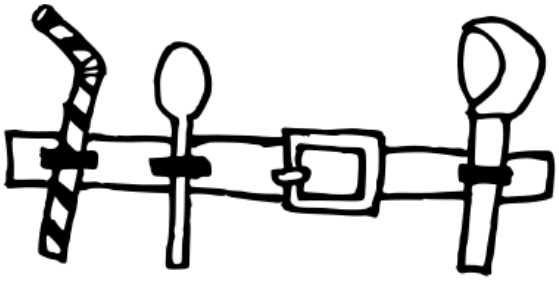

图 32.1: 图 31.1：严肃冰淇淋爱好者的工具腰带。

这里面有一些宝贝，已经成长为最佳实践：

32.3.1 django.contrib.humanize

这是一组带本地化能力的 template filters，专门用来让呈现给用户的数据更“像人话”一点。比如它里面有个叫 intcomma 的 filter，会把整数转换成带逗号的字符串；如果 locale 不同，也可能是每三位用句点分隔。

虽然 django.contrib.humanize 的 filters 很适合让模板输出更好看，但我们也可以把每个 filter 单独作为函数导入。这在处理各种文本时会非常方便，尤其是和 REST API 配合使用时。

32.3.2 django.utils.decorators.method_decorator(decorator)

Django 有一些非常棒的函数装饰器。很多人都给 Django 项目写过装饰器，特别是在使用基于函数的视图时。不过，总会有某个时刻你发现，自己最喜欢的那个函数装饰器，拿来做方法装饰器也很合适。好在 Django 提供了 method_decorator。

32.3.3 django.utils.decorators.decorator_from_middleware(middleware)

中间件是个很棒的工具，但它天然是全局性的。这可能带来额外查询，或者其他复杂问题。好在我们可以借助这个 view decorator，把某个中间件的使用范围限制到单个 view 上。

另外，也可以看看相关的 decorator_from_middleware_with_args decorator。

32.3.4 django.utils.encoding.force_text(value)

这个函数会强制 Django 把任何东西都转成 Python 中普通的 str 表示，从而避免把一个 django.utils.functional.__pro 对象原样显示出来。更多细节请参阅附录 D：《国际化与本地化》。

32.3.5 django.utils.functional.cached_property

Reinout van Rees 让我们认识到了这个极其有用的方法装饰器。它是在 Django 1.5 中引入的。它所做的事情，是把一个只接受单个 self 参数的方法结果，以 property 的形式缓存在内存中。

这对项目的性能优化有非常美妙的意义。我们在每个项目里都会用它，因为它能让那些开销昂贵的计算结果被轻松缓存起来。

至于怎么使用 cached_property decorator，Django 官方文档已经写得非常好：docs.djangoproject.com/en/3.2/ref/ut

除了潜在的性能收益，我们也用这个 decorator 来确保：某个方法取回的值，在其对象的整个生命周期内保持不变。处理第三方 API 或数据库事务时，这一点尤其有用。

警告：在 Django 之外使用 cached_property

你很容易想把 cached_property 的源码直接 copy/paste 出来，在 Django 外部使用。不过我们发现，一旦它离开 Web 框架环境，在多线程环境下就会出现问题。因此，如果你是在 Django 之外写代码，请改用下面这些方案：

- docs.python.org/dev/library/functools.html?highlight=#functools.cached_property：适用于 Python 3.8 及以上。
- github.com/pydanny/cached-property：适用于 Python 3.7 及以下。
- daniel.feldroy.com/cached-property.html

32.3.6 django.utils.html.format_html(format_str, args, **kwargs)

它很像 Python 的 str.format() 方法，但专门用于拼装 HTML fragments。所有 args 和 kwargs 都会先被转义，然后再传给 str.format() 去组合内容。

参考：docs.djangoproject.com/en/3.2/ref/utils/#django.utils.html.format_html

32.3.7 django.utils.html.strip_tags(value)

当我们需要接收用户提供的内容，并剥掉其中一切可能成为 HTML 的东西时，这个函数会移除相应标签，同时保留标签之间原有的文本。

警告：关于 strip_tags 安全性的建议

使用 strip_tags 函数，或者 striptags template tag 时，一定要确保输出内容没有被标记成 safe。尤其是在你关闭了模板自动转义的情况下，更要小心。参考：djangoproject.com/weblog/2014/mar/22/strip-tags-advisory/

32.3.8 django.utils.text.slugify(value)

无论如何，我们都建议你不要自己重写一个 slugify() 函数。只要它和 Django 的这份实现哪怕有一点不一致，就会在数据里制造出那些细微但恶心的问题。正确做法是：直接使用 Django 自己用的那个函数，并始终一致地使用 slugify()。

当然，也可以在 Python 代码里使用 django.template.defaultfilters.slugify()，因为它最终调用的也是这里说的这个函数。不过，我们更喜欢直接从 Django 的 utils 目录导入它，因为这条 import path 更合适。

无论你最后怎么导入它，我们都会尽量在项目中保持一致，因为在某些场景下它确实可能需要被替换，而下一小节就会提到这种情况。

32.3.9 非英语语言中的 Slug 生成

Tomek Paczkowski 指出，Django 内建的 slugify() 在本地化场景里可能会出问题：

示例 31.3：会被转成英文字母的 Slug 生成

Code (pycon)

```
>>> from django.utils.text import slugify
>>> slugify('straße') # German
'strae'
```

好在你可以用 allow_unicode 标志来绕过这个问题：

示例 31.4：保留原始字符的 Slug 生成

Code (pycon)

```
>>> slugify('straße', allow_unicode=True) # Again with German
'straße'
```

包提示：python-slugify

如果你想对 slug 生成过程有更强的控制力，那么 python-slugify 这个包准没错。它能提供更细粒度的 slug 化控制，例如：

- 自定义分隔符。
- 细致控制输出字符串的大小写。
- 翻译映射。
- 不依赖 Django，这对微服务很有用。
- 以及更多能力。

参考：github.com/un33k/python-slugify

32.3.10 django.utils.timezone

启用时区支持，是一种很好的实践。你的用户很可能生活在不止一个时区里。

当我们使用 Django 的时区支持时，日期和时间信息会统一以 UTC 格式存进数据库，并在需要时转换为本地时区。

32.3.11 django.utils.translation

世界上大量非英语用户都会感激这个工具，因为它提供了 Django 的 i18n 支持。更深入的参考，请见附录 D：《国际化与本地化》。

32.4 异常

Django 自带了很多异常。大部分异常都主要给内部使用，但其中有少数特别醒目，因为它们和 Django 的交互方式，能被我们以一些聪明又有创意的方式拿来利用。这些异常，以及其他 Django 内建异常，都记录在 docs.djangoproject.com/en/dev/ref/exceptions。

32.4.1 django.core.exceptions.ImproperlyConfigured

这个异常的用途，是告诉任何试图运行 Django 的人：当前存在配置问题。它几乎是唯一一个可以接受在 Django settings 模块里导入的 Django 代码组件。我们在第 5 章《Settings 与 Requirements 文件》以及附录 E：《Settings 的替代方案》中都讨论过它。

32.4.2 django.core.exceptions.ObjectDoesNotExist

这是所有 DoesNotExist 异常的基类。我们发现，它非常适合用于那些“取回某个通用模型实例并对其做点什么”的工具函数。下面是一个简单例子：

示例 31.5：通用加载函数示例

Code (python)

```
# core/utils.py
from django.core.exceptions import ObjectDoesNotExist

class BorkedObject:
    loaded = False

def generic_load_tool(model, pk):
    try:
        instance = model.objects.get(pk=pk)
    except ObjectDoesNotExist:
        return BorkedObject()

    instance.loaded = True
    return instance
```

借助这个异常，我们还可以做出一个自定义变体，模仿 Django 的 django.shortcuts.get_object_or_404，只是它抛出的可能是 HTTP 403，而不是 404：

示例 31.6：get_object_or_403

Code (python)

```
# core/utils.py
from django.core.exceptions import MultipleObjectsReturned
from django.core.exceptions import ObjectDoesNotExist
from django.core.exceptions import PermissionDenied

def get_object_or_403(model, **kwargs):
```

```
    try:
        return model.objects.get(**kwargs)
    except ObjectDoesNotExist:
        raise PermissionDenied
    except MultipleObjectsReturned:
        raise PermissionDenied
```

32.4.3 django.core.exceptions.PermissionDenied

当用户无论是否已经认证，只要试图从自己不该访问的地方获取响应，就会用到这个异常。在 view 中抛出它，会触发该 view 返回一个 django.http.HttpResponseForbidden。

这个异常非常适合用在那些会触碰高安全项目中的敏感数据或敏感组件的函数里。它的意义在于：一旦出了问题，我们不必只是返回一个 500 异常，吓到用户；我们可以直接提供一个“Permission Denied”页面。

示例 31.7：PermissionDenied 的实际用法

Code (python)

```
# stores/calc.py

def finance_data_adjudication(store, sales, issues):
    if store.something_not_right:
        msg = 'Something is not right. Please contact the support team.'
        raise PermissionDenied(msg)

    # Continue on to perform other logic.
```

在这个例子里，如果这个函数是由某个 view 调用的，而事情“出了点不对劲”，那么 PermissionDenied 异常就会强制该 view 显示项目自己的 403 错误页。说到 403 错误页，我们完全可以把它指定成任何自己想要的 view。只要在项目的根 URLConf 里加上：

示例 31.8：指定自定义 Permission Denied View

Code (python)

```
# urls.py

# This demonstrates the use of a custom permission denied view. The default
# view is django.views.defaults.permission_denied
handler403 = 'core.views.permission_denied_view'
```

和所有异常处理 view 一样，因为它们会以同样方式处理所有 HTTP 方法，所以我们依然更偏好使用基于函数的视图。

32.5 序列化器与反序列化器

无论你是想创建数据文件，还是生成一次性的简单 REST API，Django 都提供了一些很不错的工具来处理 JSON、Python、YAML 和 XML 数据的序列化与反序列化。它们能够把模型实例变成序列化数据，也能再把这些数据还原成模型实例。

下面是如何进行序列化：

示例 31.9：serializer_example.py

Code (python)

```
# serializer_example.py
from django.core.serializers import get_serializer

from favorites.models import Favorite

# Get and instantiate the serializer class
# The 'json' can be replaced with 'python' or 'xml'.
# If you have pyyaml installed, you can replace it with
# 'pyyaml'
JSONSerializer = get_serializer('json')
serializer = JSONSerializer()

favs = Favorite.objects.filter()[:5]

# Serialize model data
serialized_data = serializer.serialize(favs)

# save the serialized data for use in the next example
with open('data.json', 'w') as f:
    f.write(serialized_data)
```

下面是如何进行反序列化：

示例 31.10：deserializer_example.py

Code (python)

```
# deserializer_example.py
from django.core.serializers import get_serializer

from favorites.models import Favorite

# Get and instantiate the serializer class
# The 'json' can be replaced with 'python' or 'xml'.
# If you have pyyaml installed, you can replace it with
# 'pyyaml'
JSONSerializer = get_serializer('json')
serializer = JSONSerializer()

# open the serialized data file
with open('data.txt') as f:
    serialized_data = f.read()

# deserialize model data into a generator object
# we'll call 'python data'
python_data = serializer.deserialize(serialized_data)

# iterate through the python_data
for element in python_data:
    # Prints 'django.core.serializers.base.DeserializedObject'
    print(type(element))
```

```
    # Elements have an 'object' that are literally instantiated
    # model instances (in this case, favorites.models.Favorite)
    print(
        element.object.pk,
        element.object.created
    )
```

Django 其实已经提供了命令行工具来使用这些序列化器和反序列化器，也就是 `dumpdata` 和 `loaddata` 这两个 management command。它们当然能用，但没有“在代码里直接访问序列化器”那样高的控制力。

这也引出了我们在使用 Django 内建序列化器和反序列化器时，始终要记住的一点：它们会出问题。根据我们的痛苦经验，它们对复杂数据结构的处理并不理想。

下面这些是我们在项目中遵循的准则：

- 只在最简单的层级上序列化数据。
- 任何数据库 schema 变更，都可能让原有序列化数据失效。
- 不要只是把序列化数据直接导入。更稳妥的做法，是在保存到数据库之前，用 Django 的 form 库，或者 Django REST Framework 的 serializers，对传入数据做验证。

下面我们依次看看 Django 在处理具体格式时提供的一些特性：

32.5.1 django.core.serializers.json.DjangoJSONEncoder

开箱即用状态下，Python 内建的 JSON 模块无法处理日期 / 时间或 decimal 类型的编码。只要做 Django 做得稍微久一点，几乎都踩过这个坑。幸好，Django 为我们提供了一个非常有用的 `JSONEncoder` 类。看下面的例子：

示例 31.11：`DjangoJSONEncoder`

Code (python)

```
# json_encoding_example.py
import json

from django.core.serializers.json import DjangoJSONEncoder
from django.utils import timezone

data = {'date': timezone.now()}

# If you don't add the DjangoJSONEncoder class then
# the json library will throw a TypeError.
json_data = json.dumps(data, cls=DjangoJSONEncoder)

print(json_data)
```

32.5.2 django.core.serializers.pyyaml

虽然底层依赖第三方库 pyyaml，但 Django 这套 YAML 序列化工具处理了 pyyaml 本身不会处理的 Python 到 YAML 时间转换问题。

在反序列化时，它底层还会使用 `yaml.safe_load()`，这意味着我们不必担心代码注入。更多细节请见 28.10.3 节《那些会执行代码的第三方库》。

32.5.3 django.core.serializers.xml_serializer

默认情况下，Django 的 XML 序列化器使用的是 Python 内建 XML 处理器。同时，它也吸收了 Christian Heimes 的 defusedxml 库中的一部分机制，用来防止 XML bomb 攻击。更多信息请参阅 28.21 节《用 defusedxml 防止 XML Bomb 攻击》。

32.5.4 rest_framework.serializers

有些时候，Django 自带的序列化器就是不够用。下面是它们一些常见的局限：

- 它们只能序列化字段里存的数据，没法包含 methods 或 properties 里的数据。
- 你没法限制“到底序列化哪些字段”。这可能会变成安全问题，也可能变成性能问题。

当我们遇到这些障碍时，就应该认真考虑切换到 Django REST Framework 的 Serializers 工具集。它们在序列化和反序列化流程两端，都提供了高得多的定制能力。虽然这份能力也伴随着复杂度，但我们发现，它依然比从头手搓一套流程更值得。

参考：

- django-rest-framework.org/api-guide/serializers/
- django-rest-framework.org/api-guide/serializers/#serializing-objects
- django-rest-framework.org/api-guide/serializers/#deserializing-objects

32.6 总结

我们遵循的做法，是把那些经常复用的文件放进工具包里。我们很享受这种“总能记住自己把常用代码放在哪里”的感觉。相反，如果一个项目里同时混着 core、common、util 和 utils 这几种目录，导航起来只会更痛苦。

Django 自己的“工具腰带”里也装着一大堆有用的工具，包括函数、异常以及序列化器。善用它们，是有经验的 Django 开发者加快开发速度、并避开那些恰恰由 Django 某些特性本身带来的纠缠的方式之一。

既然前面我们已经讲了让事情“运转起来”的工具，下一章就要开始谈如何把一个项目分享给全世界了。

第三十三章 部署：平台即服务

警告：本章仍在编写中

我们正在撰写这一章，接下来几天还会继续扩充内容。欢迎就应该覆盖的话题和条目向我们提建议，请提交到：github.com/feldroy/two-scoops-of-django-3.x/issues

第三十四章 部署 Django 项目

警告：本章仍在编写中

我们正在撰写这一章，接下来几天还会继续扩充内容。欢迎就应该覆盖的话题和条目向我们提建议，请提交到：github.com/feldroy/two-scoops-of-django-3.x/issues

第三十五章 持续集成

警告：本章仍在编写中

我们正在撰写这一章，接下来几天还会继续扩充内容。欢迎就应该覆盖的话题和条目向我们提建议，请提交到：github.com/feldroy/two-scoops-of-django-3.x/issues

第三十六章 调试的艺术

警告：本章仍在编写中

我们正在撰写这一章，接下来几天还会继续扩充内容。欢迎就应该覆盖的话题和条目向我们提建议，请提交到：github.com/feldroy/two-scoops-of-django-3.x/issues

第三十七章 去哪里，以及如何提出 Django 问题

警告：本章仍在编写中

我们正在撰写这一章，接下来几天还会继续扩充内容。欢迎就应该覆盖的话题和条目向我们提建议，请提交到：`github.com/feldroy/two-scoops-of-django-3.x/issues`

第三十八章 收尾的话

虽然在本书第五版里，我们已经走过了很多内容，但这也不过只是冰淇淋甜筒的尖尖一角。我们还在继续写更多技术书，也希望你会去读！

至于《Two Scoops of Django》，如果还会有下一版，那也得等到 Django 5 快要到来时才会推出。原因是 Django 3.2 属于 Django 的长期支持版本，这意味着本书里的内容至少到 2024 年 4 月之前都还会保持不过时。

我们真心很想听到你的反馈，Django 社区里的其他人也是一样。对于和本书具体内容相关的反馈，我们正在使用 GitHub issues 来跟踪读者提交的问题与评论。下面这些内容，都欢迎到 `github.com/feldroy/two-scoops-of-django-3.` 提交：

- 你是否觉得某些主题写得不够清楚，或者让人困惑？
- 有没有我们应该知道的错误或遗漏？
- 你觉得这一版还应该补充哪些额外主题？

我们希望这本书对你来说是一次有用、也值得的阅读体验。如果你读得开心，也请通过写一条正面评价，把它推荐给其他人。我们需要，也真心感谢你的支持。

祝你的 Django 项目一路顺利！

Daniel Feldroy 和 Audrey Feldroy

- `twitch.tv/danielfeldroy` / `dev.to/audreyfeldroy` / `feldroy.com`
- `patreon.com/feldroy`：欢迎加入我们的 Patreon，支持我们持续进行创作。
- GitHub：`@pydanny`、`@audreyr` 和 `@feldroy`
- Twitter：`@pydanny`、`@audreyr` 和 `@feldroyteam`
- Facebook：`facebook.com/feldroy`

附录 A 本书提到的 Packages

这里列出了本书中介绍过或提到过的第三方 Python、Django 以及前端包。我们还偷偷加进来几个虽然书里没有专门提到、但我们觉得非常好用的包。

至于我们目前在自己项目里实际使用的包：其中有一部分和这份列表重合，但它本身一直在变化。请不要把这份清单当成“你应该用什么、不该用什么”的最终权威名单。

A.1 核心

`Django` `djangoproject.com` 为有截止日期压力的完美主义者准备的 Web 框架。

`django-debug-toolbar` `django-debug-toolbar.readthedocs.org` 显示用于调试 Django HTML views 的面板。

`django-model-utils` `https://pypi.org/project/django-model-utils` 有用的 model 工具集，其中包括带时间戳的 model。

`ipdb` `https://pypi.org/project/ipdb` 启用了 IPython 的 `pdb`。

`Pillow` `https://pypi.org/project/Pillow` Python Imaging Library 的友好安装包。

A.2 依赖管理

`conda` `http://docs.conda.io/` 科学计算社区默认使用的包管理器。

`pip` `pip-installer.org` Python 的包安装器。Python 自带它。

`pipenv` `docs.pipenv.org` 把 Pipfile、Pip 和 Virtualenv 组合在一起。基本替代了 `virtualenvwrapper`，同时还额外带来一些功能。

`poetry` `python-poetry.org` 让 Python 打包与依赖管理变得更简单。

`virtualenv` `virtualenv.org` Python 的虚拟环境。

`virtualenvwrapper` `doughellmann.com/projects/virtualenvwrapper` 让 `virtualenv` 在 Mac 和 Linux 上更好用！

`virtualenvwrapper-win` `https://pypi.org/project/virtualenvwrapper-win` 让 `virtualenv` 在 Windows 上更好用！

A.3 异步

`celery` `celeryproject.org` 分布式任务队列。

`flower` `https://pypi.org/project/flower` 用于监控和管理 Celery 任务的工具。

`django-channels` `https://pypi.org/project/channels` Django 官方的 websockets 接口，也可以作为任务队列使用。

`rq` `https://pypi.org/project/rq` RQ 是一个简单、轻量的库，用来创建后台任务并处理它们。

`django-rq` `https://pypi.org/project/django-rq` 一个简单 app，为 RQ（Redis Queue）提供 Django 集成。

`django-background-tasks` `github.com/arteria/django-background-tasks` 由数据库支撑的异步任务队列。

A.4 数据库

psycopg2-binary https://pypi.org/project/psycopg2-binary PostgreSQL 数据库适配器的二进制包。

aldjemyy github.com/aldjemy/aldjemy 在 Django 项目中集成 SQLAlchemy，以便更有效地构建查询。

django-maintenancemode-2 github.com/alsoicode/django-maintenancemode-2 一个基于数据库驱动的方案，非常适合把站点的某些部分切换成只读模式，或从只读模式切回来。不要把它用于安全相关的紧急关站。

A.5 部署

Invoke https://pypi.org/project/invoke 像 Fabric 一样，也支持 Python 3。

Supervisor supervisord.org Supervisord 是一个 client/server 系统，让用户能够在类 UNIX 操作系统上监控并控制多个进程。

A.6 文档

mkdocs mkdocs.org Markdown 文档工具。

Sphinx sphinx-doc.org 基于 ReStructuredText 或 Markdown 的文档工具。

Myst Markdown Renderer for Sphinx myst-parser.readthedocs.io Sphinx 的 Markdown 渲染器。

A.7 环境变量

dynaconf github.com/rochacbruno/dynaconf 用于 Python 和 Django 的配置管理工具，支持 toml、env、ini 等等。

python-decouple pypi.org/project/python-decouple/ 也适用于 .ini 和 .env 文件。

A.8 表单

django-crispy-forms django-crispy-forms.readthedocs.io Django forms 的渲染控制工具。默认使用 Bootstrap widgets，但可以换皮。

crispy-tailwind github.com/django-crispy-forms/crispy-tailwind 为 django-crispy-forms 提供的 Tailwind 模板包。

django-floppyforms django-floppyforms.readthedocs.io 可以和 django-crispy-forms 配合使用的 form field、widget 和 layout 工具。

A.9 前端

eslint eslint.org/ 查找并修复 JavaScript 代码中的问题。

django-webpack-loader TODO

A.10 日志与性能监控

Sentry sentry.io 卓越的错误聚合与性能监控工具，代码库是开源的。

A.11 项目模板

这里列出的这些工具，都在第 3 章《How to Lay Out Django Projects》里介绍过。

`Cookiecutter Django` `github.com/pydanny/cookiecutter-django` 本书作者之一制作的一个很受欢迎的 Cookiecutter。

`django-crash-starter` `github.com/pydanny/cookiecutter-django` `Cookiecutter Django` 的简化版，同样出自本书作者之一之手。

`cookiecutter-django-vue-graphql-aws` `github.com/grantmcconnaughey/cookiecutter-django-vue-graphql-aws` 本书第 3 章里提到的另一个示例项目布局。

`Cookiecutter` `cookiecutter.readthedocs.io` 虽然不是专门给 Django 用的，但这是一个命令行工具，可以用来创建项目和 app 模板。它聚焦明确、测试充分、文档完善，而且同样出自本书作者之一之手。

A.12 REST API 相关

`django-rest-framework` `django-rest-framework.org` 事实上的 Django REST 包标准。它可以把 model 和非 model 资源都暴露为 RESTful API。

`django-jsonview` `github.com/jsocol/django-jsonview` 提供一个简单 decorator，用来把 Python 对象转成 JSON，并确保被装饰的 views 始终返回 JSON。

`django-tastypie` `django-tastypie.readthedocs.io` 把 model 和非 model 资源暴露为 RESTful API。

A.13 安全

`bleach` `https://pypi.org/project/bleach` 一个简单、基于白名单的 HTML 清洗工具。

`defusedxml` `https://pypi.org/project/defusedxml` 只要你要接收来自外部来源的 XML，这就是必备的 Python 库。

`django-admin-honeypot` `https://pypi.org/project/django-admin-honeypot` 一个假的 Django admin 登录界面，用来在有人尝试未授权访问时通知管理员。

`django-axes` `github.com/django-pci/django-axes` 跟踪 Django 驱动站点中的登录失败尝试。

`django-csp` `github.com/mozilla/django-csp` 为 Django 添加 Content Security Policy。

`django-restricted-sessions` `github.com/mxsasha/django-restricted-sessions` 这个第三方包让你可以把 sessions 限制到某个 IP、IP 范围和 / 或 user agent。

`django-stronghold` `github.com/mgrouchy/django-stronghold` 走进你的 stronghold，让所有 Django views 默认都带上 `login_required`。

`django-two-factor-auth` `https://pypi.org/project/django-two-factor-auth` 为 Django 提供完整的双因素认证。

`django-user-sessions` `https://pypi.org/project/django-user-sessions` 带有指向 user 的外键的 Django sessions。

`Twine` `https://pypi.org/project/twine` 只使用经过验证的 TLS 把内容上传到 PyPI，从而保护你的凭证不被窃取。它还有其他一些值得一看的有用特性。

A.14 测试

`coverage` `coverage.readthedocs.io` 检查你的代码到底有多少被测试覆盖了。

`django-test-plus` `github.com/revsys/django-test-plus` 对 Django 默认 `TestCase` 的实用增强。Two Scoops 的作者们是从这个包的作者那里学来的，并一直沿用至今。很高兴现在它已经被打包好，使用起来更方便了。

`factory boy` `https://pypi.org/project/factory_boy` 一个用来生成 model 测试数据的包。

`model bakery` `https://pypi.org/project/model_bakery` 另一个用来生成 model 测试数据的包。

`mock` `https://pypi.org/project/mock` 虽然不是专门给 Django 用的，但它允许你用 mock objects 替换系统中的某些部分。这个项目后来在 Python 3.4 中进入了标准库。

`pytest` `pytest.org` 成熟而全功能的 Python 测试工具，对 Python 和 Django 项目都非常有用。

`pytest-django` `pytest-django.readthedocs.io` `pytest-django` 是 `py.test` 的一个插件，为测试 Django 应用和项目提供了一组很有用的工具。

`tox` `tox.readthedocs.io` 一个通用的 `virtualenv` 管理与测试命令行工具，允许你在 shell 中用一条命令测试项目在多个 Python 版本下的表现。

`nox` `nox.readthedocs.io` `nox` 是一个命令行工具，用来自动化在多个 Python 环境中的测试，类似于 `tox`。和 `tox` 不同，Nox 使用标准 Python 文件来配置。

A.15 用户注册

`django-allauth` `django-allauth.readthedocs.io` 通用型的注册与认证方案。内置 Email、Twitter、Facebook、GitHub、Google 等很多渠道。

`python-social-auth` `github.com/python-social-auth/social-core` 面向 Twitter、Facebook、GitHub、Google 等平台的简易社交认证与注册方案。

`django-registration` `github.com/ubernostrum/django-registration` 一个简单、可扩展的 Django 用户注册 app。

A.16 视图

`django-braces` `django-braces.readthedocs.io` 开箱即用的 mixins，能真正强化 Django 的基于类的视图。

`django-vanilla-views` `django-vanilla-views.org` 通过简化继承链，来简化 Django 的通用基于类的视图。

A.17 时间

`pendulum` `https://pypi.org/project/pendulum` 让 Python 的 datetimes 变得更简单，Audrey Feldroy 强烈推荐。

`python-dateutil` `https://pypi.org/project/python-dateutil` 为 Python 的 `datetime` 模块提供强大扩展。

`pytz` `https://pypi.org/project/pytz` 把 Olson tz 数据库带进 Python。这个库可以进行精确、跨平台的时区计算，也解决了夏令时结束时的模糊时间问题。Library Reference

A.18 杂项

`dj-stripe` `pypi.org/project/dj-stripe` 让 Django + Stripe 变得更简单。

`django-compressor` `django-compressor.readthedocs.io` 借助 Node，把链接式和内联的 JavaScript 或 CSS 压缩成单个缓存文件。

`django-extensions` `django-extensions.readthedocs.io` 提供 `shell_plus` management command，以及很多其他实用工具。

`django-haystack` `github.com/django-haystack/django-haystack` 支持 SOLR、Elasticsearch 等的全文搜索。

`django-js-reverse` `github.com/ierror/django-js-reverse` 不会让人头疼的 Django JavaScript URL 处理工具。

`django-pipeline` `github.com/jazzband/django-pipeline` 用于压缩 CSS 和 JS。可配合 `cssmin` 与 `jsmin` 包一起使用。

`django-htmlmin` `github.com/cobrateam/django-htmlmin` 用于 Django 的 HTML 压缩器。

`django-reversion` `github.com/etianen/django-reversion` 一个 Django Web 框架扩展，提供全面的版本控制能力。

`django-watson` `github.com/etianen/django-watson` 一个基于 SQL 数据库特性的 Django 全文多表搜索应用。

`envdir` `github.com/jezdez/envdir` `daemontools` 的 `envdir` 的一个 Python 移植版。

`flake8` `pypi.org/project/flake8` 借助 PyFlakes、pep8 和其他工具检查代码质量。

`isort` `pypi.org/project/flake8`

`pip-tools` `github.com/nvie/pip-tools` 一组用来保持固定版 Python 依赖始终新鲜的工具。

`python-slugify` `github.com/un33k/python-slugify` 一个灵活的 `slugify` 函数。

`pyyaml` `pypi.org/project/PyYAML` Python 的 YAML 解析器与发射器。

`requests` `docs.python-requests.org` 一个好用的 HTTP 库，用来替代 Python 的 `urllib2`。

`silk` `github.com/mtford90/silk` Silk 是 Django 框架的实时 profiling 与检查工具。它会拦截并保存 HTTP 请求与数据库查询，然后通过一个用户界面把它们展示出来，供进一步排查。

附录 B 安装问题排查

这个附录汇总了一些用来排查常见 Django 安装问题的建议。

B.1 识别问题

很多时候，问题无非是下面两类之一：

- Django 不在你的系统路径里，或者
- 你运行的是错误版本的 Django

在命令行里运行下面这条命令：

示例 1：检查你的 Django 版本

Code (bash)

```
python -c "import django; print django.get_version()"
```

如果你运行的是 Django 3.x，那么你应该会看到下面这样的输出：

示例 2：Django 版本

Code (text)

```
3.x，其中 "x" 是你当前使用的 Django 版本号
```

如果你看到的不是同样的输出，那至少现在你已经知道问题出在哪了。继续往下读，看看怎么解决。

B.2 我们推荐的解决方案

解决 Django 安装问题的方式有很多（比如手动编辑你的 PATH 环境变量），但下面这些建议能帮助你用一种和《The Optimal Django Environment Setup》那一章一致的方式，把环境修好。

B.2.1 检查你的 Virtualenv 安装

你的电脑上是否正确安装了 virtualenv？请在命令行里试着创建一个测试虚拟环境并激活它。

如果你用的是 Mac 或 Linux，请确认下面这组命令可以正常工作：

示例 3：在 Mac 或 Linux 上检查 Virtualenv

Code (bash)

```
$ virtualenv testenv
$ source testenv/bin/activate
```

如果你用的是 Windows，请确认下面这组命令可以正常工作：

示例 4：在 Windows 上检查 Virtualenv

Code (text)

```
C:\code\> virtualenv testenv
C:\code\> testenv\Scripts\activate
```

你的 virtualenv 现在应该已经被激活了，而命令行提示符前面也应该会多出虚拟环境名称。

在 Mac 或 Linux 上，它大概会长这样：

示例 5：Mac 与 Linux 上的 Virtualenv 提示符

Code (text)

```
(testenv) $
```

在 Windows 上，它大概会长这样：

示例 6：Windows 上的 Virtualenv 提示符

Code (text)

```
(testenv) >
```

过程中有碰到什么问题吗？如果有，请去认真读一遍 Virtualenv 文档（`virtualenv.org`），并修好你的 Virtualenv 安装。

如果没有，那就继续。

B.2.2 检查你的环境里是否安装了正确版本的 Django

在激活 `virtualenv` 的情况下，再检查一次你的 Django 版本：

示例 7：再次检查 Django 版本

Code (bash)

```
python -c "import django; print django.get_version()"
```

如果你看到的仍然不是你预期定义的那个版本，那么试着使用 `pip` 把 Django 安装进 `testenv`：

示例 8：用 Pip 安装 Django 3.x

Code (bash)

```
(testenv) $ pip install Django==3.2
```

成功了吗？如果成功，再检查一次 Django 版本。如果还是不行，请按照官方文档（`pip-installer.org`）检查你是否正确安装了 `pip`。

B.2.3 检查其他问题

如果你遇到的是运行 `django-admin` 相关的问题，请按照 Django 官方文档里的排查说明继续检查：`docs.djangoproject.com/en/3.2/faq/troubleshooting/`

附录 C 补充资源

这个附录列出了一些我们认为不过时的补充资源。它们之中有些针对的是 Python 或 Django 的旧版本，但其中表达的概念本身超越了具体版本。

C.1 不过时的 Python 与 Django 材料

C.1.1 书籍

Best Django Books (2020) `wsvincent.com/best-django-books/` William Vincent 维护了一份他心目中最佳 Django 书单。那份清单里的书优秀到让我们觉得，自己的书能和它们并列其中两个位置，都是一种荣幸。

Speed Up Your Django Tests `gumroad.com/l/suydt` 只推荐给中高级 Djangonauts。Django 技术委员会成员 Adam Johnson 写了一本非常出色的书，专讲如何加速测试。在更大或更复杂的项目里，这会是一个极其关键的问题。

High Performance Django `amazon.com/High-Performance-Django/dp/1508748128/` `highperformancedjango.com` 这本书聚焦于 Django 的扩展与伸缩，倡导了很多优秀实践。里面塞满了有用的信息和技巧，而且每一节后面都附带问题，逼着你认真思考自己正在做什么。

Two Scoops of Django: Best Practices for Django 1.11 (print) `twoscoopspress.com/products/two-scoops-of-django-1-` 本书的第四个纸质版。

C.1.2 Web 资源

Django Packages `djangopackages.org` 一个为 Django 项目收集可复用 apps、站点、工具等资源的目录，由《Two Scoops of Django》的作者维护。

Classy Class-Based Views `ccbv.co.uk` 一个网站，为 Django 的每一个通用基于类的视图提供详细说明，包括完整的方法与属性列表。

Classy Django Forms `cdf.9vo.lt` 为 Django 的每个 form class 和 method 提供详细说明，包括完整的方法与属性列表。

Classy Django REST Framework `cdrf.co` 为 Django REST Framework 的每个基于类的 view 和 serializer 提供详细说明，包括完整的方法与属性列表。

Daniel's blog `daniel.feldroy.com/tag/django.html` 这个博客里有相当一部分内容都在讨论现代 Django。由于博客作者也是本书作者之一，所以博客风格和本书内容在松散意义上也有些相似。他目前正在重写自己那些已经过时的 Django 文章。

Django Model Behaviors `blog.kevinastone.com/django-model-behaviors.html` Kevin Stone 在这里探讨了：在大型 Django 项目中，应该如何组织 models 以及与之相关的代码。

Awesome-Django `awesome-django.com` 一个精挑细选的 Django apps、项目与资源列表。

Real Python Blog `realpython.com/blog/categories/django/` 除了他们出色的教程书之外，Real Python 的博客还包含了大量很有用的 Django 内容，覆盖面也非常广。

Simple is Better than Complex `simpleisbetterthancomplex.com/` Vitor Freitas 是一位很棒的 Python 与 Django 写作者，能把复杂话题拆解成真正可用的部件。

Django Tricks `djangotricks.com/` Aidas Bendoraitis 多年来一直在整理关于 Django 和 Python 的技巧、窍门与代码片段。他也是《Django 3 Web Development Cookbook》的作者，这本书可以在 Amazon 上找到：`https://amzn.to/3dYaXfQ`

testdriven.io `testdriven.io/blog/topics/django/` Michael Herman 多年来一直在教授和写作 Django、Python、JavaScript 等主题。我们一直觉得他的作品很有启发性，也非常实用。

C.2 不过时的 Django 入门材料

C.2.1 Web 资源

Official Django 3.0 Documentation `docs.djangoproject.com/en/3.2/` Django 官方文档极其有用。如果你之前用过旧版本 Django，请务必确认自己看的是对应版本的文档。

Django Girls Tutorial `tutorial.djangogirls.org` 由国际性组织 Django Girls 创建并维护。不论你的性别是什么，这都是一份非常优秀的资源。

Simple is Better than Complex `simpleisbetterthancomplex.com/` Vitor Freitas 的入门级内容写得非常好。

C.3 不过时的 Python 入门材料

Learn Python the Hard Way Online Edition `learnpythonthehardway.org` 直接去原始出处是最好的。这套资源提供免费的 HTML 内容，以及付费的视频内容，是最适合起步的地方之一。尤其是视频资源，格外有帮助。

Automate the Boring Stuff with Python `amazon.com/gp/product/1593275994` 这是一本非常迷人的书，它通过教你如何把枯燥的计算机任务自动化，来教你 Python。既然 Python 能替你更新电子表格里的 150 列数据，那你何必自己手动更新？

C.4 不过时、而且实用的 Python 材料

Fluent Python `amzn.to/2oHTORa` 这是我们最喜欢的 Python 书之一。作者 Luciano Ramalho 会带你走遍 Python 语言核心特性和标准库，并展示怎样让代码在同一时间变得更短、更快、更易读。

Effective Python `amzn.to/1NsiqVr` 讲解了很多在写 Python 时非常有用的实践与技巧。

Python Cookbook, 3rd Edition `amzn.to/I3Sv6q` 这本了不起的书出自 Python 名家 David Beazley 和 Brian Jones 之手，里面装满了美味的冰淇淋配方……呃，不对，是为任何使用 Python 3.3 及以上版本的开发者准备的大量实用 Python 配方。

Treading on Python Volume 2 `amzn.to/1kVWi2a` 覆盖了更高级的 Python 结构。

Writing Idiomatic Python 3.3 `amzn.to/1aS5df4` 里面有很多优化代码、提升可读性的好建议。确实有少数地方和我们的做法不完全一致（最大的分歧点是 imports），但总体上我们是认同的。作者也提供了 2.7 版：`amzn.to/1fj9j7z`

C.5 JavaScript 资源

警告：注意：这里需要帮忙！

C.5.1 书籍

Secrets of a JavaScript Ninja (Print and Kindle) `amzn.to/2AknxYp`

`TODO - Add more good, modern JS books!`

C.5.2 Web 资源

Mozilla Developer Network `developer.mozilla.org/en-US/docs/Web/JavaScript`

Stack Overflow `stackoverflow.com/questions/tagged/javascript`

`TODO - Add at least one more good, modern JS web resource!`

警告：离 W3Schools 远一点

在网上查 JavaScript 和 CSS 资料时，有个问题是：W3Schools 往往会排在搜索引擎结果的最前面。这很糟，因为那里的很多资料已经老旧到足以误导人。聪明一点，避开它。

我们通常会在搜索结果页里找到 Mozilla Developer Network（MDN）的链接，它往往排在第三位左右，然后点那个。

附录 D 国际化与本地化

Django 和 Python 提供了许多非常有用的工具，用来处理国际化、本地化以及 Unicode。

这个附录列出了一些在为非英语读者、以及非美国用户准备 Django 应用时会很有帮助的事项。当然，这份清单绝不完整，我们也欢迎读者继续补充反馈。

D.1 尽早开始

从一开始就构建一个支持国际化和本地化、并随着项目一起成长的系统，总是比后面再把一个现有项目改造成支持国际化更容易。

D.2 用翻译函数包裹内容字符串

所有会展示给终端用户的字符串，都应该包在翻译函数里。这个主题在 Django 官方文档的 `django.utils.translation` 一节中有深入说明：`docs.djangoproject.com/en/3.2/topics/i18n/translation/`。不过那一大坨文字确实不太好吞，所以我们把下一页那张表当作一个参考索引，帮助你快速判断：在什么场景下，该用哪一个翻译函数来完成什么任务。

函数	用途	链接
`gettext()`	用于运行时才执行的内容，例如 form 验证错误。	`docs.djangoproject.com/en/3.2/topics/i18n/tra`
`gettext_lazy()`	用于导入 / 编译阶段就会执行的内容，例如 models 中的 `verbose_name`。	`docs.djangoproject.com/en/3.2/topics/i18n/tra`
`django.utils.text.format_lazy()`	用于延迟格式化字符串；在需要把懒加载字符串组合起来时，可以替代标准字符串格式化与拼接方式。	`docs.djangoproject.com/en/3.2/topics/i18n/tra`

表 1：翻译函数参考

D.3 约定：使用下划线别名减少输入

你也知道，通常我们并不喜欢缩写或快捷写法。不过，在 Python 代码做国际化这件事上，现成的约定就是使用 _，也就是下划线，来省点按键。

示例 9：下划线别名的实际用法

Code (python)

```
from django.utils.translation import gettext as _

print(_('We like gelato.'))
```

D.4 不要在句子里插值拼词

黄金法则是：尽可能把语法本身放进字符串里，不要让代码去把语法拼装出来；而一般来说，最麻烦的通常是动词。

- Patrick McLoughlan

我们以前经常构造翻译字符串，甚至在本书 1.5 版里都这么写过。也就是说，我们会用一些自以为有点聪明的代码，把各种 Python 对象拼成一句完整的话。下面这个例子，就是当年 8.7 示例的一部分：

示例 10：我们在 1.5 版里写过的糟糕代码

Code (python)

```
# DON'T DO THIS!

# Skipping the rest of imports for the sake of brevity
class FlavorActionMixin:

    @property
    def action(self):
        msg = '{0} is missing action.'.format(self.__class__)
        raise NotImplementedError(msg)

    def form_valid(self, form):
        msg = 'Flavor {0}!'.format(self.action)
        messages.info(self.request, msg)
        return super().form_valid(form)

# Snipping the rest of this module for the sake of brevity
```

它看起来似乎挺方便，因为这样就能做出一个“自维护”的 mixin，但这其实聪明过头了，因为我们没法对 `self.__class__` 的调用结果做国际化。换句话说，你不能只是再加上 `django.utils.translation`，就指望它能产生出任何对译者真正有意义的东西：

示例 11：把事情搞得让译者无法处理

Code (python)

```
# DON'T DO THIS!
from django.utils.translations import gettext as _

# Skipping the rest of this module for the sake of brevity

def form_valid(self, form):

    # This generates a useless translation object.
    msg = _('Flavor {0}!'.format(self.action))
    messages.info(self.request, msg)
    return super().form_valid(form)

# Skipping the rest of this module for the sake of brevity
```

现在，我们不会再写这种“把句子从各种 Python 构件里拼出来”的代码，而是会写出更完整、更有意义的对话文本，让它们能被直接翻译。这确实会稍微多一点工作量，但结果是：项目更容易翻译。因此，我们现在遵循下面这种模式：

示例 12：使用完整字符串

Code (python)

```
# Skipping the rest of imports for the sake of brevity
from django.utils.translation import gettext as _
```

```
class FlavorActionMixin:

    @property
    def success_msg(self):
        return NotImplemented

class FlavorCreateView(LoginRequiredMixin, FlavorActionMixin,
                       CreateView):

    model = Flavor

    # Slightly longer but more meaningful dialogue
    success_msg = _('Flavor created!')

    # Skipping the rest of this module for the sake of brevity
```

顺带一提，你当然可以把若干个本身就有完整意义的字符串和对话，再组合成一个更大的值。不过，你不应该靠拼接零散片段来造句，因为其他语言所需的语序可能完全不同。出于同样原因，翻译字符串里也应该始终带上标点。看下面的例子：

示例 13：在翻译字符串里使用标点

Code (python)

```
from django.utils.translation import gettext as _

class FlavorActionMixin:

    @property
    def success_msg(self):
        return NotImplemented

class FlavorCreateView(LoginRequiredMixin, FlavorActionMixin,
                       CreateView):

    model = Flavor

    # Example combining strings
    part_one = _('Flavor created! ')
    part_two = _("Let's go try it!")
    success_msg = part_one + part_two

    # Skipping the rest of this module for the sake of brevity
```

D.5 Unicode 小技巧

下面这些，是我们在处理 Unicode 相关问题时学到的一些经验。

D.5.1 用 django.utils.encoding.force_text()，不要用 str()

当我们想确保返回的是某个有用的字符串值时，不要使用内建的 str()。因为在某些情况下，Django 返回的不是一个真正有意义的字符串对象，而是一个几乎没法直接理解的 django.utils.functional.__proxy__ 对象。它其实是所请求数据的一个 lazy instance。

更好的做法，是像我们的朋友 Douglas Miranda 建议的那样，使用 django.utils.encoding.force_text。如果你手上拿到的是 proxy object 或 lazy instance，它会把它们解析成字符串。

提示：Django 是 Lazy 的

Django 做优化的一种方式，就是使用 lazy loading。这是一种设计模式：只有在对象真正需要时，才进行初始化。这个机制用得最广的地方，就是 Django ORM，详见：docs.djangoproject.com/en/3.2/topics/db/queries/#querysets 但也正因为 Django 大量使用了这种 lazy objects，展示内容时有时就会出问题，于是才需要 django.utils.encoding.force_te

D.6 浏览器页面布局

假设你已经把内容和 Django templates 都做了国际化、本地化，然后你会发现：布局可能还是会坏掉。

Mozilla 以及它围绕 Firefox 等工具构建的各种站点，就是一个很好的 Django 相关例子。那些站点支持 80 多种语言。不幸的是，一个在英语里刚刚好能放得下的标题，换到像德语这样更啰嗦的语言后，就可能把页面撑坏。

Mozilla 的应对方式，是先测量标题容器的宽度，然后用 JavaScript 逐步把标题文字的字号调小，直到文字能在容器内换行并完整放下为止。

更简单的处理方式，是预先假设其他语言可能需要占据英语两倍的空间。英语本身相当简洁，而且因为词比较短，所以文本换行时往往表现得很好。

D.7 时间带来的挑战

即便完全不考虑时区和不同历法系统，时间本身也是个很棘手的话题。下面这些资源能帮助你在处理时间相关编码时少踩坑：

- yourcalendricalfallacyis.com

附录 E Settings 的替代方案

这里介绍几种我们认为值得推荐的 settings 管理替代模式。它们避开了 `local_settings` 这种反模式，同时也允许你用既兼容“环境变量模式”、又兼容“Secrets 文件模式”的方式来管理配置。

警告：把已有 Settings 方案迁移掉很难

如果你手上已经有一个使用多 settings 模块的现有项目，而你又想把它改造成单一 settings 风格，那你最好先重新考虑一下。迁移 settings 方案始终都是一件棘手的事，而且需要既深又广的测试覆盖。即便测试覆盖已经做得很好，也仍然有可能最后发现这件事根本不值得。

正因为如此，我们建议：对切换新的 settings 管理方式这件事保持保守。只有在当前这套 settings 管理方法已经成了明确痛点时再动手，而不是因为某个新方法突然流行起来就跟着切。

E.1 Twelve Factor 风格的 Settings

既然我们已经依赖环境变量了,那为什么不干脆使用一个尽可能简单的 `settings.py` 系统?`django-floppyforms` 和 FeedHQ（`feedhq.org`）的作者 Bruno Renié 倡导另一种 Django settings 文件方案：所有环境都共用同一个单一 settings 文件。

这种做法的核心观点是：一旦你采用多 settings 文件方案，就 inevitably 会写出与环境绑定的代码。比如在本地开发时，你跑的就不是“带生产 settings 的那套代码”。这会提高一种风险：你改了某些代码，却忘了同步更新生产 settings，结果就只在生产环境里撞上 bug。

这种风格的做法，是尽量使用合理默认值，并且把环境专属的值压缩到最少。当它和 Vagrant、Docker 这类工具配合使用时，镜像式复现生产环境会变得非常轻松。

最后得到的 settings 文件会简单得多；而对于 Twelve Factor App 的拥趸来说，它和那套思路也完全一致。

如果你想看看这种方法的实际示例,可以去看 FeedHQ 的 settings 模块:`github.com/feedhq/feedhq/blob/master/feedh`

我们在新的、小一些的项目里很喜欢这种方式。只要做得对，它会把事情简化得非常优雅。

不过，它也不是解决所有问题的完美方案：

- 当开发环境与生产环境差异极大时，这种方法在简化层面带来的收益并不大。
- 当项目要部署到不止一种操作系统上时，它的效果也没那么好。
- 对大型项目来说，那些复杂 settings 并不会因为这种方案就真的变短、变简单。对于大型或复杂项目，它仍然可能很难用。

如果你想更深入了解这种方法，我们推荐下面这些文章：

- `bruno.im/2013/may/18/django-stop-writing-settings-files/`
- `12factor.net/config`

附录 F 安全设置参考

在 Django 中，要知道某个 setting 在开发环境和生产环境里分别应该设成什么值，不幸的是需要相当多的领域知识。这个附录就是一个参考，用来帮助你更好地理解：怎样同时为开发与生产环境配置 Django 项目。

Setting	开发环境	生产环境
ALLOWED_HOSTS	任意列表	见下一页
Cross Site Request Forgery protection	见下一页	见下一页
DEBUG	True	False
DEBUG_PROPAGATE_EXCEPTIONS	False	True
Email SSL	见下一页	见下一页
MIDDLEWARE_CLASSES	Standard	加上 SecurityMiddleware
SECRET_KEY	使用加密强度足够的密钥	见 5.3 节
SECURE_CONTENT_TYPE_NOSNIFF	False	True
SECURE_PROXY_SSL_HEADER	None	见下文
SECURE_SSL_HOST	False	见下一页
SESSION_COOKIE_SECURE	False	True
SESSION_SERIALIZER	见下文	见下文

表 2：安全设置参考

F.1 跨站请求伪造保护相关设置

对绝大多数情况来说，Django 在这些设置上的默认值已经足够用了。下面这份清单列出的是一些边界场景，以及相应的 CSRF setting 文档，必要时可以拿来缓解问题：

- Internet Explorer 与 CSRF 失败：docs.djangoproject.com/en/3.2/ref/settings/#csrf-cookie-age
- 跨子域请求排除（例如从 vanilla.twoscoopspress.com 向 chocolate.twoscoopspress.com 发起 POST）：docs.djangoproject.com/en/3.2/ref/settings/#csrf-cookie-domain
- 修改默认的 CSRF 失败视图：docs.djangoproject.com/en/3.2/ref/settings/#csrf-failure-view

F.2 电子邮件 SSL

Django 支持与 SMTP 服务器建立安全连接。我们强烈建议启用这项能力。下面这些 settings 的说明文档起点在：docs.djangoproject.com/en/3.2/ref/settings/#email-use-tls

- EMAIL_USE_TLS
- EMAIL_USE_SSL
- EMAIL_SSL_CERTFILE
- EMAIL_SSL_KEYFILE

F.3 SESSION_SERIALIZER

正如 28.10.4 小节所说：

Code (python)

```
SESSION_SERIALIZER = django.contrib.sessions.serializers.JSONSerializer
```

F.4 SECURE_PROXY_SSL_HEADER

在某些部署环境里，尤其是 Heroku，它应该写成：

Code (python)

```
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')
```

附录 G 处理安全故障

G.1 在事情出错前，就准备好一套方案

处理安全故障会让人压力爆表。那种紧迫感与恐慌，很容易压过我们更冷静的判断，导致拍脑袋做决定，比如上线不靠谱的“bug 修复”，或者发表会让问题进一步恶化的公开声明。

因此，必须事先写好一套逐点执行的方案，并让维护者，甚至项目中的非技术参与者，都能拿到它。下面是一份示例方案：

1. 立刻全部停机，或者把系统切成只读模式。
2. 挂出一个静态 HTML 页面。
3. 把一切都备份下来。
4. 先阅读 `docs.djangoproject.com/en/dev/internals/security/`，然后把你的安全相关问题发邮件到 `security@djang`
   哪怕这件事本来就是你的锅。
5. 开始调查问题。

下面我们逐条来看：

G.2 关闭一切，或者切成只读模式

第一件要做的事，就是把这个安全问题继续造成伤害的能力拿掉。这样才有希望阻止进一步损失。

在 Heroku 上：

示例 14：为 Heroku 项目开启维护模式

Code (bash)

```
$ heroku maintenance:on
Enabling maintenance mode for myapp... done
```

如果你的项目是自行部署，或者借助自动化工具部署的，那么你就得自己把这项能力准备出来。好在，别人之前已经遇到过这种事，所以我们手里已经有了一些参考资料：

- `cyberciti.biz/faq/custom-nginx-maintenance-page-with-http503`：讲的是如何挂出维护中的 503 页面。
- 其他工具可以在 `djangopackages.org/grids/g/emergency-management` 找到。

G.3 挂出一个静态 HTML 页面

你在项目上线时，就应该把维护页面提前准备好。这样一来，一旦出了事而你又已经停掉了系统，就可以把这张页面展示给终端用户。只要做得妥当，用户往往是能理解的，也会给你一点喘息空间来处理问题。

G.4 把一切都备份下来

先把代码拿一份，再把数据从服务器上拷下来，放到本地硬盘或 SSD 上。你也可以考虑用一家带担保、专业的存储公司。

为什么要这么做？第一，在这个阶段先备份，你是在保护自己的审计轨迹。这也许能让你之后追查出问题究竟是在什么时候、什么地方发生的。

第二，这话可能不太好听，但恶意员工能制造出来的问题，并不比 bug 或成功入侵少。也就是说，再好的软件安全措施，在一个故意留后门的开发者，或者一个决定搞事情的非技术岗位员工面前，都可能毫无作用。

G.5 给 security@djangoproject.com 发邮件，哪怕这事本来就是你的错

只要你的问题和安全有关，就先读一遍 docs.djangoproject.com/en/dev/internals/security/，然后迅速发一封邮件概述问题。顺便也可以直接求助。

这么做很重要，有几个原因：

- 先写一份简短摘要，本身就会帮你聚焦并整理思路。你此时会承受极大压力，而这种压力和紧迫感会让人很容易去做蠢事，从而把问题搞得更糟。
- 你永远不知道，Django 安全团队会不会正好给你一个好建议，甚至直接给出答案。
- 说不定这压根就是 Django 本身的问题！如果真是这样，Django 安全团队必须尽快知道，好在问题公开之前先替所有人做好缓解。

提示：Jacob Kaplan-Moss 谈向 Django 项目报告安全问题

前 Django BDFL、Heroku 前安全主管 Jacob Kaplan-Moss 说过："比起人们因为不懂情况而把安全问题公开泄露出去，我宁可他们把那些其实并不是 Django 真漏洞的东西发到 security@djangoproject.com。"

G.6 开始调查问题

现在，你已经停掉系统、完成备份、挂出了静态 HTML 页面、给 security@djangoproject.com 发了邮件，也已经开始盯问题本身了。只要按前面的步骤做，你就已经给自己（也可能包括你的团队）争取到了一点喘息空间，可以真正去弄清楚到底发生了什么。

这会是一段高压时期，人会站在恐慌边缘。开始做调查吧，也许翻翻这本书，或者按第 36 章《去哪里，以及如何提出 Django 问题》里的方式去提问，并找出解决办法。

在你真正实施修复之前，通常更好的做法是：先确保自己手里拿到的是一个真实、正确的修复方案，而不是匆忙打一块应急补丁，结果把整个系统一并炸掉。没错，这正是测试和持续集成真正发光的时候。

保持积极：现在正是大家一起把问题修好的时刻。开始记笔记，向你认识的最强的人求助，提醒自己（或提醒团队）你们既有意志也有能力把事情修好，并把一切拉回正轨。

警告：零日攻击的噩梦

零日攻击，指的是攻击那些公众或小范围安全名单里可能已经知道、但仍然没有补丁的漏洞。例如：

- 在软件更新发布当天就发起攻击，而多数人甚至还没来得及安装更新。
- 某个用户刚写完一篇博客，公开了自己发现的漏洞，而包维护者还没来得及写出并发布补丁，就已经有人开始攻击。
- 针对那些正在封闭式安全邮件列表中讨论的漏洞发起攻击。

面对零日攻击时，往往根本没有足够时间去处理并打补丁，因此一旦系统失守，局面就尤其难以管理。如果说有什么场景最能说明为什么必须准备一份应对安全事件的作战方案，那就是这个。

参见：en.wikipedia.org/wiki/0day

附录 H 使用 Channels 处理 WebSockets

正如第 27 章《异步任务队列》里已经提到过的，Channels 让 Django 具备了处理 WebSockets 的能力。用 Channels 做这件事的好处在于：和 FastAPI、Node.js 之类替代方案相比，Channels 让我们可以用一种非常接近 views 的方式去使用 WebSockets。更棒的是，借助 Channels，我们可以直接访问项目自己的代码库，也就是可以直接用到 models 和其他自定义代码。我们发现，只要继续遵守本书其他章节里一直倡导的那些实践，Channels 会是一件非常强大的工具。

要记住，WebSockets 的优势远不只是“实时界面”。更重要的是，这个协议本身比 HTTP 更轻量，所以客户端和服务器之间来回传输数据时，速度也会更快。

下面是一些基于踩坑经验总结出来的想法：

H.1 每个浏览器标签页都有自己的 WebSocket 连接

如果某个用户给你的项目同时开着一百个标签页，那它实际上就是通过一百条 WebSocket 连接连到你这里。可以想象，这会让浏览器崩掉；如果很多用户都这么干，服务器也会被拖垮。我们当然不太可能强迫用户关标签页，但我们可以在服务端做优化。这一点我们在第 26 章《发现并减少瓶颈》里讲过。

另一种做法，是跟踪某个用户当前连到系统上的 WebSocket 数量。如果他同时开着超过二十条（或者任何你自己设定的阈值）连接，那就关闭那些看起来什么都没在做的连接。我们发现，这是一种相当有效的办法，可以保护服务器免受不必要的负载冲击。遗憾的是，目前还没有现成的 stock solution 能直接这样解决问题。

H.2 要预期 WebSocket 连接会随时掉线

Channels 里的 WebSockets，和典型的 Django request-response cycle 工作方式完全不同。后者是：从用户那里收一个 HTTP request，再回给他一个 HTML 或 JSON 形式的 HTTP response。而 WebSockets 则是在服务器与浏览器之间打开一条持续存在的连接，也就是 socket。这也正是“WebSocket”这个名字的由来。

问题在于，这条连接的生存概率并不乐观。来看看都有什么东西在威胁它：

- 浏览器和服务器之间哪怕很小的延迟
- 服务器抛出某种等价于 500 错误的故障
- 服务器崩溃
- 浏览器标签页崩溃
- 浏览器本身崩溃
- 用户把电脑切到睡眠状态

与其像 `socket.io` 或 SockJS 那样用某种长轮询回退方案来处理这个问题，Django Channels 选择的是：让连接直接死掉。等它死掉以后，客户端再自己触发一条新连接。好在 Django Channels 提供了一个小型 JavaScript 库，在旧连接挂掉后会自动帮你建立新连接。

这里最关键的一点，是要记住：只要你在用 Channels，就必须把“长轮询不是我们能用的选项”这个事实考虑进去。

H.3 一定要验证传入数据！

WebSockets 只是用户往项目里发数据的另一种方式而已。和往常一样，在你对任何传入数据做其他处理之前，先验证它。我们既用过 Django Forms，也用过 Django REST Framework Serializers 来做这件事。前一种做法，我们已经在 13.1 节《用 Django Forms 验证所有传入数据》中详细讲过。

H.4 小心意大利面式代码

好了，坦白时间到了：我们第一次认真用 Channels 做项目时，最后搞出来的是一大盘难看的意大利面式代码。拼起来的时候我们玩得很开心，可等做完后就意识到，后端代码已经完全不可维护了。

后来我们放慢速度，重新依靠 fat models 和 helper files 这类思路，一边做一边写文档，补上更好的测试，并把 Generic Consumers 当成基于类的视图那样去拥抱。换句话说，我们后来真的开始践行自己在书里一直说的那些话。

我们和其他开发者交流后发现，原来并不是只有我们第一次做 Channels 项目时会搞得一团糟。从中得到的教训是：一旦兴奋地开始玩 Django Channels，就很容易犯下那些最基础的错误。所以，第一次写 Channels 代码时，帮自己一个忙：放慢一点，用标准最佳实践。尤其是在写测试这件事上：`channels.readthedocs.io/en/stable/topics/testing`

致谢

这本书并不是在真空里写出来的。我们想向所有参与过这本书的人表达感谢。

H.5 Python 与 Django 社区

Python 和 Django 社区就像一个由朋友和导师组成的了不起大家庭。正是因为这个共同体的力量，我们才得以相遇、相爱，并受到鼓舞写下这本书。

H.6 3.x 版技术审校者

这些审校者贡献了他们宝贵的时间，帮我们抓出错误，也多次在关键时刻把我们带回正确方向。下面按字母顺序列出：

Adeyinka Adegbenro Adeyinka Adegbenro 是一位软件工程师，对计算机科学和问题求解充满热情。工作之外，她喜欢写作（随笔和技术教程）、锻炼，以及看纪录片。她生活在尼日利亚拉各斯。

Carol Willing Carol Willing 是 Python Steering Council 成员，也是 CPython 核心开发者。她是 Python Software Foundation Fellow，也曾担任董事。2019 年，她因在 Python 技术与社区上的贡献，获得了 Frank Willison Award：`python.org/community/awards/frank-willison/#carol-willing-2019`。Carol 长期投入社区外展工作，也是 PyLadies San Diego 和 San Diego Python User Group 的共同组织者。

Enrique Matías Sánchez Enrique Matías Sánchez 是一位软件工程师，同时也特别偏爱系统管理。除了编程之外，他喜欢户外运动（骑行、徒步、跑步、攀岩……）以及学习如何用外语“胡言乱语”。他住在西班牙萨拉戈萨，和自己迷人的女友 Elena 以及一只黑猫一起生活。

Haris Ibrahim K V Haris Ibrahim K V 是一位人类，自 2014 年起担任软件开发者，主要使用 Python 与 Django 构建网站，以此养家糊口。他喜欢教学、写作、唱歌、宗教、学习，以及用酥油烤香蕉（是的，不是冰淇淋，就这样！）。

Iacopo Spalletti Iacopo Spalletti 花在电脑上的时间比他自己愿意承认的还要多。Iacopo 在 21 世纪第一个十年的后半段遇见 Python 和 Django，立刻就觉得“回家了”。他把时间分在为客户构建东西、做开源、做讲者、做组织者与导师（meetups、PyCon Italia、DjangoCon Europe、DjangoGirls）之间。最近则把重点放在：一边养育一个小人类，一边尽量少制造 bug。

Madelene Campos Madelene Campos 正在从职业音乐人的身份“恢复中”，曾拿到两个音乐学位。虽然她大学并没有读计算机科学，但 Web 开发对她来说像是音乐学习的自然延伸。编程和音乐，都是强大而重要的沟通工具。她 2015 年开始走上这条路，2016 年起成为职业开发者。Maddie 愉快地使用 Python 和 Django 已经略多于两年，并希望继续沿着这条光辉道路前行！

Modasser Billah Modasser Billah 是一位 djangonaut，而他最喜欢的宠物是 Python。他热爱在云端构建企业软件系统，目前担任 BriteCore 的 Lead Software Engineer。Modasser 是 AWS certified developer associate，也是“借助软件工程师群体共同智慧，也就是 best practices，来做事”的坚定支持者。他热爱远程工作，也热爱没有边界的团队。Modasser 拥有 Bangladesh University of Engineering & Technology 的 CS 学士学位，现居孟加拉国库米拉。不写代码的时候，他多半是在看当天的 Dilbert，或者和自己年幼的女儿 Naseeha、Naaera 一起玩。

Renato Oliveira Renato Oliveira 是一位巴西软件工程师，也是 Labcodes Software Studio 的联合创始人。他使用 Python 和 Django 工作已经接近 10 年。他喜欢构建可持续演进的 Web 应用：既不会每加一个功能都拖得天荒地老，也能帮助别人更顺利地完成工作。

H.7 3.x 版安全审校者

下面这些 Django 安全团队成员，审阅了我们与隐私、认证、密码学及其他安全主题相关的内容。

Florian Apolloner Florian Apolloner 是一位软件工程师，热爱调试那些看起来已经无药可救的案例，并且乐于挖出奇怪 bug 的真正根因。他大约在 2007 年开始为 Django 做贡献，2012 年成为 Django Team 成员，如今则是 Security Team 的一员。他喜欢骑车和各种冬季运动，住在没有袋鼠的奥地利（不是澳大利亚啦）。

James Bennett James Bennett 从 Django 几乎刚发布那天起，就一直在使用并热爱它。他曾在 Lawrence Journal-World 工作五年，也是在那里第一次直接参与这个项目。此后，他曾担任 committer、release manager、框架安全团队与技术委员会成员，并曾在 Django Software Foundation 董事会任职。如今他住在加州。Daniel 和 Audrey 是在他 2010 年 PyCon 的 Django Deep Dive 教程上认识的，所以从某种意义上说，《Two Scoops of Django》也该算他的一份“功劳”。

Markus Holtermann（Security）Markus Holtermann 已经在多个开源社区和项目里活跃了十多年。他通过在这些社区中承担各种任务，积累了工程、安全、应用与基础设施安全以及社区管理经验。Markus 是 Django Web 框架安全团队的一员，也做过多场关于 Django 及其周边安全主题的会议演讲。空闲时，他是一名业余摄影师。

Michael Manfre Michael Manfre 是一位软件工程师，参与开源项目已经超过二十年。他是 Django Web 框架安全团队的一员。Michael 从 2008 年开始为 Django 做贡献，最初主要专注于维护 Microsoft SQL Server 数据库后端，以及让 Django 更好地运行在 Windows 上。

H.8 3.x Alpha 贡献者

下面这些人通过提交修正和内容建议，帮助我们改进了这一版：Luca Bezerra, Jane Upshur, Denis Morozov, Jack Pote, Sam Partington, Sim Kimsia, Nathan Hinchey, Kamil Kijak, Eman Calso, Jeff Maher, Veli Eroglu, Davide Brunato, Ryan Harrington, Yong Shean, Laurent Steffan, Alexander Viklund, Ramast Magdy, Steve Ratcliffe, Jeremiah Cooper, Arnav Choudhury, Keith Richards, Victor David, Hugh Macdonald, Anthony Shaw, Andy Grabow, Lance Goyke, Edwin Lileo, Tomi Okretič, Tim McCurrach, Ambrose Andrews, Remy Charlot, José Francisco Martínez Salgado, Holger Rother, Mark Dobossy, Vlad Dmitrievich, Marek Turnovec, Marcos Escalona, Bill Beal, Patrick Bollinger, Foad Mohammadi, Dominique Bischof, Noah Lipsyc, Rakibul Islam, Michael Scharf, Mauro Allegrini

H.9 版技术审校者

下面这些人在支持 1.11 版时起到了关键作用。

Matt Braymer-Hayes Nathan Cox Ola Sendecka Jannis Gebauer Haris Ibrahim K V Tom Christie Michael Herman Humphrey Butau Sasha Romijn - Security Reviewer James Bennett - Security Reviewer Florian Apolloner - Security Reviewer Aymeric Augustin - Security Reviewer

H.10 版贡献者

下面这些人帮助我们改进了这一版：Andrés Pérez-Albela H., Leila Loezer, Martin Koistinen, Miguel Pachas, José Augusto Costa Martins Jr., Daniel Bond, John Carter, Bernard‘BJ’Jauregui, Peter Inglesby, Michael Scharf, Jason Wolosonovich, Dipanjan Sarkar, Anish Menon, Ramon Maria Gallart Escolà, You Zhou, Khaled Alqenaei, Xus Zoyo, Joris Derese, Laurent Steffan, Louie Pascual, Charlie Tian, Thijs van Dien, Paulina Rybicka, Qi Ying Koo, Bassem Ali, Jonathan Mitchell, Anton Backer, Martijn Mhkuu, Photong, Michael John Barr, Kevin Marsh, Greg Smith, Muraoka Yusuke, Michael Helmick, Zachery Tapp, Jesús Gómez, Klemen Strušnik, Peter Brooks, Bernat Bonet, Danilo Cabello,

Alenajk, Piotr Szpetkowski, Nick Wright, Michael Sanders, Nate Guerin, David Adam Hernandez, Brendan M. Sleight, Maksim Iakovlev, and David Dahan

H.11 版技术审校者

下面这些人在支持 1.8 版时起到了关键作用。

Bartek Ogryczak Barry Morrison Kevin Stone Paul Hallett Saurabh Kumar Sasha Romijn - Security Reviewer

H.12 版贡献者

Kenneth Love, Patrick McLoughlan, Sebastián J. Seba, Kevin Campbell, Doug Folland, Kevin London, Ramon Maria Gallart Escolà, Eli Bendersky, Dan O’Donovan, Ryan Currah, Shafique Jamal, Russ Ferriday, Charles L. Johnson, Josh Wiegand, William Vincent, Tom Atkins, Martey Dodoo, Krace Kumar Ramaraju, Felipe Arruda Pontes, Ed Patrick Tan, Sven Aßmann, Christopher Lambacher, Colin O’Brien, Sebastien de Menten, Evangelos Mantadakis, Silas Wegg, Michal Hoftich, Markus Holterman, Pat Curry, Gaston Keller, Mihail Russu, Jean-Baptiste Lab, Kaleb Elwert, Tim Bell, Zuhair Parvez, Ger Schinkel, Athena Yao, Norberto Bensa, Abhaya Agarwal, Steve Sarjeant, Karlo Tamayo, Cary Kempston, José Padilla, Konstantinos Faliagkas, Kelsey Gilmore-Innis, Adam Bogdał, Tyler Davis, Javier Liendo, Kevin Xu, Michael Barr, Caroline Simpson, John Might, Tom Christie, Nicolas Pannetier, Marc Tamlyn, Loïc Bistuer, Arnaud Limbourg, Alasdair Nicol, and Ludvig Wadenstein.

H.13 版技术审校者

下面这些人在支持 1.6 版时起到了关键作用。

Aymeric Augustin Barry Morrison Ken Cochrane Paul McMillan - Security Reviewer

H.14 版技术审校者

下面这些人给予了我们无比宝贵的帮助、支持和鼓励，促成了本书的初次发布。我们尤其想在这里特别致敬 Malcolm，感谢他对本书以及整个世界做出的贡献。

Malcolm Tredinnick Malcolm Tredinnick 生前住在澳大利亚悉尼，也把大量时间花在国际旅行上。他使用 Python 超过 15 年，在 Django 于 2005 年中公开发布后不久就开始使用 Django，并在 2006 年成为 Django 核心开发者。作为一位使用过多种编程语言的人，他觉得 Django 是自己职业生涯中用过的更优秀的 Web libraries / frameworks 之一，也很高兴看到它这些年来被极其广泛地采用。2012 年，当他得知我们正在联合举办第一届 PyCon Philippines 时，他立刻主动提出飞过来，做两场演讲，并共同主持 sprint。遗憾的是，他在 2013 年 3 月离世，那距离本书首次发行仅仅两个月。他在 Python 和 Django 社区中的领导力与慷慨，将永远被铭记。

下面这些人同样在支持 1.5 版时起到了关键作用。

Kenneth Love Lynn Root Barry Morrison Jacob Kaplan-Moss Jeff Triplett Lennart Regebro Randall Degges Sean Bradley

H.15 版分章审校者

下面这些人在 1.5 版写作过程中，为特定章节提供了令人惊叹的大量帮助与支持。我们想特别感谢 Preston Holmes 对 User model 章节的贡献，Tom Christie 对 REST API 章节的睿智观察，以及 Donald Stufft 对 Security 章节的支持。

H.16 版贡献者

下面这些人向我们发送了修正、清理建议、bug 修复和改进意见，包括：Álex González, Alex Gaynor, Amar Šahinović, Andrew Halloran, Andrew Jordan, Anthony Burke, Aymeric Augustin, Baptiste Mispelon, Bernardo Brik, Branko Vukelic, Brian Shumate, Carlos Cardoso, Charl Botha, Charles Denton, Chris Foresman, Chris Jones, Dan Loewenherz, Dan Poirier, Darren Ma, Daryl Yu, Dave Castillo, Dave Murphy, David Beazley, David Sauve, Davide Rizzo, Deric Crago, Dolugen Buuraldaa, Dominik Aumayr, Douglas Miranda, Eric Woudenberg, Sasha Romijn, Esteban Gaviota, Fabio Natali, Farhan Syed, Felipe Coelho, Felix Ingram, Florian Apolloner, Francisco Barros, Gabe Jackson, Gabriel Duman, Garry Cairns, Graham Dumpleton, Hamid Hoorzad, Hamish Downer, Harold Ekstrom, Hrayr Artunyan, Jacinda Shelly, Jamie Norrish, Jason Best, Jason Bittel, Jason Novinger, Jannis Leidel, Jax, Jim Kalafut, Jim Munro, João Oliveira, Joe Golton, John Goodleaf, John Jensen, Jonas Obrist, Jonathan Hartley, Jonathan Miller, Josh Schreuder, Kal Sze, Karol Breguła, Kelly Nicholes, Kelly Nichols, Kevin Londo, Khee Chin, Lachlan Musicman, Larry Prince, Lee Hinde, Maik Hoepfel, Marc Tamlyn, Marcin Pietranik, Martin Báchtold, Matt Harrison, Matt Johnson, Michael Reczek, Mickey Cheong, Mike Dewhirst, Myles Braithwaite, Nick August, Nick Smith, Nicola Marangon, Olav Andreas Lindekleiv, Patrick Jacobs, Patti Chen, Peter Heise, Peter Valdez, Phil Davis, Prahlad Nrsimha Das, R. Michael Herberge, Richard Cochrane, Richard Corden, Richard Donkin, Robbie Totten, Robert Węglarek, Rohit Aggarwa, Russ Ferriday, Saul Shanabrook, Simon Charettes, Stefane Fermigier, Steve Klass, Tayfun Sen, Tiberiu Ana, Tim Baxter, Timothy Goshinski, Tobias G. Waaler, Tyler Perkins, Vinay Sajip, Vinod Kurup, Vraj Mohan, Wee Liat, William Adams, Xianyi Lin, Yan Kalchevskiy, Zed Shaw, and Zoltán Árokszállási.

H.17 排版

我们感谢 Laura Gelsomino，感谢她帮我们解决所有 LaTeX 问题，也感谢她进一步改进了整本书的版式设计。

Laura Gelsomino Laura Gelsomino 是一位热爱艺术与写作、又对计算机怀有偏爱的经济学家。她在发现 LaTeX 的那一天，找到了这些兴趣的交汇点。从那以后，只要手边有任何文本，她几乎都会忍不住把自己的审美感“发泄”在上面，经济模型自然也不例外。

我们最初使用 iWork Pages 为 1.5 版 alpha 版本排版。之后的版本则改为使用 LaTeX 写作。到 1.11 为止的所有版本，都是在 2011 款 Macbook Air 上写出来的。3.x 版则是在 Macbook Pros 与 Macbook Airs 混合使用的环境中完成的。