结构化你的工程

我们对于“结构化”的定义是你关注于怎样使你的项目最好地满足它的对象性，我们 需要去考虑如何更好地利用Python的特性来创造简洁、高效的代码。在实践层面， “结构化”意味着通过编写简洁的代码，并且正如文件系统中文件和目录的组织一样， 代码应该使逻辑和依赖清晰。

哪个函数应该深入到哪个模块？数据在项目中如何流转？什么功能和函数应该组合 或独立？要解决这些问题，你可以开始做个一计划，大体来说，即是你的最终产品 看起来会是怎样的。

在这一章节中，我们更深入地去观察Python的模块和导入系统，因为它们是加强你 的项目结构化的关键因素，接着我们会从不同层面去讨论如何去构建可扩展且测试 可靠的的代码。

仓库的结构

这很重要

在一个健康的开发周期中，代码风格，API设计和自动化是非常关键的。同样的，对于工程的 架构 ,仓库的结构也是关键的一部分。

当一个潜在的用户和贡献者登录到你的仓库页面时，他们会看到这些:

- 工程的名字
- 工程的描述
- 一系列的文件

只有当他们滚动到目录下方时才会看到你工程的README。

如果你的仓库的目录是一团糟，没有清晰的结构，他们可能要到处寻找才能找到你写的漂亮的文档。

> 为你的渴望的事业而奋斗，而不是仅仅只为你现在的工作而工作。

当然，第一印象并不是一切。但是，你和你的同事会和这个仓库并肩战斗很长时间，会熟悉它的每一个角落和细节。拥有良好的布局，事半功倍。

仓库样例

请看这里: 这是 Kenneth Reitz 推荐的。

这个仓库 可以在Github上找到 。

```
README.rst
LICENSE
setup.py
requirements.txt
sample/__init__.py
sample/core.py
sample/helpers.py
docs/conf.py
docs/index.rst
tests/test_basic.py
tests/test_advanced.py
```

让我们看一下细节。

真正的模块

布局	./sample/ or ./sample.py
作用	核心代码

你的模块包是这个仓库的核心，它不应该隐藏起来:

```
./sample/
```

如果你的模块只有一个文件，那么你可以直接将这个文件放在仓库的根目录下:

```
./sample.py
```

这个模块文件不应该属于任何一个模棱两可的src或者python子目录。

License

布局	./LICENSE
作用	许可证.

除了源代码本身以外，这个毫无疑问是你仓库最重要的一部分。在这个文件中要有完整的许可说明和授权。

如果你不太清楚你应该使用哪种许可方式，请查看 choosealicense.com.

当然，你也可以在发布你的代码时不做任何许可说明，但是这显然阻碍潜在的用户使用你的代码。

Setup.py

布局	./setup.py
作用	打包和发布管理

如果你的模块包在你的根目录下，显然这个文件也应该在根目录下。

Requirements File

布局	./requirements.txt
作用	开发依赖.

一个 pip requirements file 应该放在仓库的根目录。它应该指明完整工程的所有依赖包: 测试, 编译和文档生成。

如果你的工程没有任何开发依赖，或者你喜欢通过 setup.py 来设置，那么这个文件不是必须的。

Documentation

布局	./docs/

作用	包的参考文档

没有任何理由把这个放到别的地方。

Test Suite

布局	./test_sample.py or ./tests
作用	包的集合和单元测试

最开始，一组测试例子只是放在一个文件当中:

```
./test_sample.py
```

当测试例子逐步增加时，你会把它放到一个目录里面，像下面这样:

```
tests/test_basic.py
tests/test_advanced.py
```

当然，这些测试例子需要导入你的包来进行测试，有几种方式来处理:

- 将你的包安装到site-packages中。
- 通过简单直接的路径设置来解决导入的问题。

我极力推荐后者。如果使用 setup.py develop 来测试一个持续更新的代码库，需要为每一个版本的代码库设置一个独立的测试环境.太麻烦了。

可以先创建一个包含上下文环境的文件 tests/context.py。 file:

```
import os
import sys
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
```

```
import sample
```

然后，在每一个测试文件中，导入:

```
from .context import sample
```

这样就能够像期待的那样工作，而不用采用安装的方式。

一些人会说应该把你的测试例子放到你的模块里面 – 我不同意。这样会增加你用户使用的复杂度；而且添加测试模块将导致需要额外的依赖和运行环境。

Makefile

布局	./Makefile
作用	常规的管理任务

如果你看看我的项目或者其他开源项目，你都会发现有一个Makefile。为什么？这些项目也不是用C写的啊。。。简而言之，make对于定义常规的管理任务是非常有用的工具。

** 样例 Makefile:**

```
init:
    pip install -r requirements.txt

test:
    py.test tests

PHONY: init test
```

一些其他的常规管理脚本（比如 manage.py 或者 fabfile.py），也放在仓库的根目录下。

关于 Django Applications

从Django 1.4开始，我发现有这样一个现象：很多开发者错误地使用Django自带的应用模板创建项目，导致他们的仓库结构非常糟糕。

这是怎么回事儿? 是的, 他们在进入一个新的仓库后，通常都这样操作：

```
$ django-admin.py startproject samplesite
```

这样的操作生成的仓库结构是这样的:

```
README.rst
samplesite/manage.py
samplesite/samplesite/settings.py
samplesite/samplesite/wsgi.py
samplesite/samplesite/sampleapp/models.py
```

亲，不要这样做。

相对路径会让你的工具和你的开发者都很疑惑。没有必要的嵌套对任何人都没有好处（除非你怀念庞大的SVN仓库）。

让我们这样来做:

```
$ django-admin.py startproject samplesite .
```

注意末尾的 “.“。

生成的结构是这样的:

```
README.rst
manage.py
samplesite/settings.py
```

```
samplesite/wsgi.py
samplesite/sampleapp/models.py
```

结构是一把钥匙

得益于Python提供的导入与管理模块的方式，结构化Python项目变得相对简单。 这里说的简单，指的是结构化过程没有太多约束限制而且模块导入功能容易掌握。 因而你只剩下架构性的工作，包括设计、实现项目各个模块，并整理清他们之间 的交互关系。

容易结构化的项目同样意味着它的结构化容易做得糟糕。糟糕结构的特征包括：

- 多重且混乱的循环依赖关系：假如在 furn.py 内的Table与Chair类需要 导入 workers.py 中的Carpenter类以回答类似 table.isdoneby() 的问题，并且Carpenter类需要引入Table和Chair类以回答 carpenter.whatdo() 这类问题，这就是一种循环依赖的情况。在这种情况下,你得借助一些不怎么靠谱的 小技巧，比如在方法或函数内部使用import语句。
- 隐含耦合：Table类实现代码中每一个改变都会打破20个不相关的测试用例，由于它 影响了Carpenter类的代码，这要求谨慎地操作以适应改变。这样的情况意味着 Carpenter类代码中包含了太多关于Table类的假设关联（或相反）。
- 大量使用全局变量或上下文：如果Table和Carpenter类使用不仅能被修改而且能被 不同引用修改的全局变量，而不是明确地传递 (height, width, type, wood) 变量。你就需要彻底检查全局变量的所有入口，来理解到为什么一个长方形桌子变 成了正方形，最后发现远程的模板代码修改了这份上下文，弄错了桌子尺寸规格的 定义。
- 面条式代码 (Spaghetti code) ：多页嵌套的if语句与for循环，包含大量复制-粘贴 的过程代码，且没有合适的分割——这样的代码被称为面条式代码。Python中有意思 的缩进排版(最具争议的特性之一)使面条式代码很难维持。所以好消息是你也许不 会经常看到这种面条式代码。
- Python中更可能出现混沌代码：这类代码包含上百段相似的逻辑碎片，通常是缺乏 合适结构的类或对象，如果你始终弄不清手头上的任务应该使用FurnitureTable， AssetTable还是Table，甚至TableNew，也许你已经陷入了混沌代码中。

模块

Python模块是最主要的抽象层之一，并且很可能是最自然的一个。抽象层允许将代码分为 不同部分，每个部分包含相关的数据与功能。

例如在项目中，一层控制用户操作相关接口，另一层处理底层数据操作。最自然分开这两 层的方式是，在一份文件里重组所有功能接口，并将所有底层操作封装到另一个文件中。 这种情况下，接口文件需要导入封装底层操作的文件，可通过 import 和 from ... import 语句完成。一旦你使用 *import* 语句，就可以使用这个模块。 既可以是内置的模块包括 *os* 和 *sys*，也可以是已经安装的第三方的模块，或者项目 内部的模块。

为遵守风格指南中的规定，模块名称要短、使用小写，并避免使用特殊符号，比如点(.) 和问号(?)。如 my.spam.py 这样的名字是必须不能用的！该方式命名将妨碍 Python的模块查找功能。就 *my.spam.py* 来说，Python 认为需要在 my 文件夹 中找到 spam.py 文件，实际并不是这样。这个例子 example 展示了点表示 法应该如何在Python文件中使用。如果愿意你可以将模块命名为 my_spam.py， 不过并不推荐在模块名中使用下划线。但是，在模块名称中使用其他字符（空格或连字号） 将阻止导入（-是减法运算符），因此请尽量保持模块名称简单，以无需分开单词。 最重要的是，不要使用下划线命名空间，而是使用子模块。

```
# OK
import library.plugin.foo
# not OK
import library.foo_plugin
```

除了以上的命名限制外，Python文件成为模块没有其他特殊的要求，但为了合理地使用这 个观念并避免问题，你需要理解import的原理机制。具体来说，import modu 语句将 寻找合适的文件，即调用目录下的 modu.py 文件（如果该文件存在）。如果没有 找到这份文件，Python解释器递归地在 “PYTHONPATH” 环境变量中查找该文件，如果仍没 有找到，将抛出ImportError异常。

一旦找到 modu.py，Python解释器将在隔离的作用域内执行这个模块。所有顶层 语句都会被执行，包括其他的引用。方法与类的定义将会存储到模块的字典中。然后，这个 模块的变量、方法和类通过命名空间暴露给调用方，这是Python中特别有用和强大的核心概念。

在很多其他语言中，include file 指令被预处理器用来获取文件里的所有代码并‘复制’到调用方的代码中。Python则不一样：include代码被独立放在模块命名空间里，这意味着你 一般不需要担心include的代码可能造成不好的影响，例如重载同名方法。

也可以使用import语句的特殊形式 from modu import * 模拟更标准的行为。但这通常 被认为是不好的做法。**使用** import * **的代码较难阅读而且依赖独立性不足**。 使用 from modu import func 能精确定位你想导入的方法并将其放到全局命名空间中。 比 import * 要好些，因为它明确地指明往全局命名空间中导入了什么方法，它和 import modu 相比唯一的优点是可以少打点儿字。

差

```
[...]
from modu import *
[...]
x = sqrt(4)  # sqrt是模块modu的一部分么？或是内建函数么？上文定义了么？
```

稍好

```
from modu import sqrt
[...]
x = sqrt(4)  # 如果在import语句与这条语句之间，sqrt没有被重复定义，它也许是模块modu的一部分。
```

最好的做法

```
import modu
[...]
x = modu.sqrt(4)  # sqrt显然是属于模块modu的。
```

在 代码风格 章节中提到，可读性是Python最主要的特性之一。可读性意味着避免 无用且重复的文本和混乱的结构，因而需要花费一些努力以实现一定程度的简洁。但不能 过份简洁而导致简短晦涩。除了简单的单文件项目外，其他项目需要能够明确指出类和方法 的出处，例如使用 modu.func 语句，这将显著提升代码的可读性和易理解性。

包

Python提供非常简单的包管理系统，即简单地将模块管理机制扩展到一个目录上(目录扩 展为包)。

任意包含 __init__.py 文件的目录都被认为是一个Python包。导入一个包里不同 模块的方式和普通的导入模块方式相似，特别的地方是 __init__.py 文件将集合 所有包范围内的定义。

pack/ 目录下的 modu.py 文件通过 import pack.modu 语句导入。 该语句会在 pack 目录下寻找 __init__.py 文件，并执行其中所有顶层 语句。以上操作之后，modu.py 内定义的所有变量、方法和类在pack.modu命名空 间中均可看到。

一个常见的问题是往 __init__.py 中加了过多代码，随着项目的复杂度增长， 目录结构越来越深，子包和更深嵌套的子包可能会出现。在这种情况下，导入多层嵌套 的子包中的某个部件需要执行所有通过路径里碰到的 __init__.py 文件。如果包内的模块和子包没有代码共享的需求，使用空白的 __init__.py 文件是正常 甚至好的做法。

最后，导入深层嵌套的包可用这个方便的语法：import very.deep.module as mod。 该语法允许使用 *mod* 替代冗长的 very.deep.module。

面向对象编程

Python有时被描述为面向对象编程的语言，这多少是个需要澄清的误导。在Python中 一切都是对象，并且能按对象的方式处理。这么说的意思是，例如函数是一等对象。 函数、类、字符串乃至类型都是Python对象：与其他对象一样，他们有类型，能作为 函数参数传递，并且还可能有自己的方法和属性。这样理解的话，Python是一种面向 对象语言。

然而，与Java不同的是，Python并没有将面向对象编程作为最主要的编程范式。非面向 对象的Python项目(比如，使用较少甚至不使用类定义，类继承，或其它面向对象编程的 机制)也是完全可行的。

此外在 模块 章节里曾提到，Python管理模块与命名空间的方式提供给开发者一个自然 的方式以实现抽象层的封装和分离，这是使用面向对象最常见的原因。因而，如果业务逻辑 没有要求，Python开发者有更多自由去选择不使用面向对象。

在一些情况下，需要避免不必要的面向对象。当我们想要将状态与功能结合起来，使用 标准类定义是有效的。但正如函数式编程所讨论的那个问题，函数式的“变量”状态与类的 状态并不相同。

在某些架构中，典型代表是web应用，大量Python进程实例被产生以响应可能同时到达的 外部请求。在这种情况下，在实例化对象内保持某些状态，即保持某些环境静态信息， 容易出现并发问题或竞态条件。有时候在对象状态的初始化(通常通过 __init__() 方法实现)和在其方法中使用该状态之间，环境发生了变化，保留的状态可能已经过时。 举个例子，

某个请求将对象加载到内存中并标记它为已读。如果同时另一个请求要删除 这个对象，删除操作可能刚好发生在第一个请求加载完该对象之后，结果就是第一个请 求标记了一个已经被删除的对象为已读。

这些问题使我们产生一个想法：使用无状态的函数是一种更好的编程范式。另一种建议 是尽量使用隐式上下文和副作用较小的函数与程序。函数的隐式上下文由函数内部访问 到的所有全局变量与持久层对象组成。副作用即函数可能使其隐式上下文发生改变。如 果函数保存或删除全局变量或持久层中数据，这种行为称为副作用。

把有隐式上下文和副作用的函数与仅包含逻辑的函数(纯函数)谨慎地区分开来，会带来 以下好处：

- 纯函数的结果是确定的：给定一个输入，输出总是固定相同。
- 当需要重构或优化时，纯函数更易于更改或替换。
- 纯函数更容易做单元测试：很少需要复杂的上下文配置和之后的数据清除工作。
- 纯函数更容易操作、修饰和分发。

总之，对于某些架构而言，纯函数比类和对象在构建模块时更有效率，因为他们没有任何 上下文和副作用。但显然在很多情况下，面向对象编程是有用甚至必要的。例如图形桌面 应用或游戏的开发过程中，操作的元素(窗口、按钮、角色、车辆)在计算机内存里拥有相 对较长的生命周期。

装饰器

Python语言提供一个简单而强大的语法: ‘装饰器’。装饰器是一个函数或类，它可以 包装(或装饰)一个函数或方法。被‘装饰’的函数或方法会替换原来的函数或方法。 由于在Python中函数是一等对象，它也可以被‘手动操作’，但是使用 @decorators 语法更清晰，因此首选这种方式。

```
def foo():
    # 实现语句

def decorator(func):
    # 操作func语句
    return func

foo = decorator(foo)  # 手动装饰
```

```
@decorator
def bar():
    # 实现语句
# bar()被装饰了
```

这个机制对于分离概念和避免外部不相关逻辑“污染”主要逻辑很有用处。 *记忆化* <*https://en.wikipedia.org/wiki/Memoization#Overview*> 或缓存就是一个很 好的使用装饰器的例子：你需要在table中储存一个耗时函数的结果，并且下次能直接 使用该结果，而不是再计算一次。这显然不属于函数的逻辑部分。

上下文管理器

上下文管理器是一个Python对象，为操作提供了额外的上下文信息。 这种额外的信息， 在使用 with 语句初始化上下文，以及完成 with 块中的所有代码时，采用可调用的形式。 这里展示了使用上下文管理器的为人熟知的示例，打开文件：

```
with open('file.txt') as f:
    contents = f.read()
```

任何熟悉这种模式的人都知道以这种形式调用 open 能确保 f` 的 ``close 方法会在某个时候被调用。 这样可以减少开发人员的认知负担，并使代码更容易阅读。

实现这个功能有两种简单的方法：使用类或使用生成器。 让我们自己实现上面的功能，以使用类方式开始：

```
class CustomOpen(object):
    def __init__(self, filename):
        self.file = open(filename)

    def __enter__(self):
        return self.file

    def __exit__(self, ctx_type, ctx_value, ctx_traceback):
        self.file.close()
```

```
with CustomOpen('file') as f:
    contents = f.read()
```

这只是一个常规的Python对象，它有两个由 with 语句使用的额外方法。 CustomOpen 首先被实例化，然后调用它的 `__enter__``方法，而且 __enter__ 的返回值在 as f 语句中被赋给 f 。 当 with 块中的内容执行完后，会调用 *__exit__*` 方法。

而生成器方式使用了Python自带的 contextlib:

```
from contextlib import contextmanager

@contextmanager
def custom_open(filename):
    f = open(filename)
    try:
        yield f
    finally:
        f.close()

with custom_open('file') as f:
    contents = f.read()
```

这与上面的类示例道理相通，尽管它更简洁。custom_open 函数一直运行到 yield 语句。 然后它将控制权返回给 with 语句，然后在 as f 部分将yield的 *f* 赋值给f。 finally 确保不论 with 中是否发生异常， close() 都会被调用。

由于这两种方法都是一样的，所以我们应该遵循Python之禅来决定何时使用哪种。 如果封装的逻辑量很大，则类的方法可能会更好。 而对于处理简单操作的情况，函数方法可能会更好。

动态类型

Python是动态类型语言，这意味着变量并没有固定的类型。实际上，Python 中的变量和其他 语言有很大的不同，特别是静态类型语言。变量并不是计算机内存中被写入的某个值，它们 只是指向内存的 ‘标签’ 或 ‘名称’ 。因此可能存在这

样的情况，变量 ‘a’ 先代表值1，然后变成 字符串 ‘a string’，然后又变为指向一个函数。

Python 的动态类型常被认为是它的缺点，的确这个特性会导致复杂度提升和难以调试的代码。 命名为 ‘a’ 的变量可能是各种类型，开发人员或维护人员需要在代码中追踪命名，以保证它 没有被设置到毫不相关的对象上。

这里有些避免发生类似问题的参考方法：

- 避免对不同类型的对象使用同一个变量名

差

```
a = 1
a = 'a string'
def a():
    pass  # 实现代码
```

好

```
count = 1
msg = 'a string'
def func():
    pass  # 实现代码
```

使用简短的函数或方法能降低对不相关对象使用同一个名称的风险。即使是相关的不同 类型的对象，也更建议使用不同命名：

差

```
items = 'a b c d'  # 首先指向字符串...
items = items.split(' ')  # ...变为列表
items = set(items)  # ...再变为集合
```

重复使用命名对效率并没有提升：赋值时无论如何都要创建新的对象。然而随着复杂度的 提升，赋值语句被其他代码包括 ‘if’ 分支和循环分开，使得更难查明指定变量的类型。 在某些代码的做法中，例如函数编程，推荐的是从不重复

对同一个变量命名赋值。Java 内的实现方式是使用 ‘final’ 关键字。Python并没有 ‘final’ 关键字而且这与它的哲学 相悖。尽管如此，避免给同一个变量命名重复赋值仍是是个好的做法，并且有助于掌握 可变与不可变类型的概念。

可变和不可变类型

Python提供两种内置或用户定义的类型。可变类型允许内容的内部修改。典型的动态类型 包括列表与字典：列表都有可变方法，如 **list.append()** 和 **list.pop()**， 并且能就地修改。字典也是一样。不可变类型没有修改自身内容的方法。比如，赋值为整数 6的变量 x 并没有 “自增” 方法，如果需要计算 x + 1，必须创建另一个整数变量并给其命名。

```
my_list = [1, 2, 3]
my_list[0] = 4
print my_list  # [4, 2, 3] <- 原列表改变了

x = 6
x = x + 1  # x 变量是一个新的变量
```

这种差异导致的一个后果就是，可变类型是不 ‘稳定 ‘的，因而不能作为字典的键使用。合理地 使用可变类型与不可变类型有助于阐明代码的意图。例如与列表相似的不可变类型是元组， 创建方式为 (1, 2)。元组是不可修改的，并能作为字典的键使用。

Python 中一个可能会让初学者惊讶的特性是：字符串是不可变类型。这意味着当需要组合一个 字符串时，将每一部分放到一个可变列表里，使用字符串时再组合 (‘join’) 起来的做法更高效。 值得注意的是，使用列表推导的构造方式比在循环中调用 append() 来构造列表更好也更快。

差

```
# 创建将0到19连接起来的字符串 (例 "012..1819")
nums = ""
for n in range(20):
  nums += str(n)   # 慢且低效
print nums
```

好

```
# 创建将0到19连接起来的字符串 (例 "012..1819")
nums = []
for n in range(20):
  nums.append(str(n))
print "".join(nums)  # 更高效
```

更好好

```
# 创建将0到19连接起来的字符串 (例 "012..1819")
nums = [str(n) for n in range(20)]
print "".join(nums)
```

最好Best

```
# 创建将0到19连接起来的字符串 (例 "012..1819")
nums = map(str, range(20))
print "".join(nums)
```

最后关于字符串的说明的一点是，使用 join() 并不总是最好的选择。比如当用预先 确定数量的字符串创建一个新的字符串时，使用加法操作符确实更快，但在上文提到的情况 下或添加到已存在字符串的情况下，使用 join() 是更好的选择。

```
foo = 'foo'
bar = 'bar'

foobar = foo + bar  # 好的做法
foo += 'ooo'  # 不好的做法, 应该这么做:
foo = ''.join([foo, 'ooo'])
```

注解:

除了 **str.join()** 和 **+**，你也可以使用 **%** 格式运算符来连接确定数量的字符串，但 **PEP 3101** 建议使用 **str.format()** 替代 **%** 操作符。

```
foo = 'foo'
bar = 'bar'

foobar = '%s%s' % (foo, bar) # 可行
foobar = '{0}{1}'.format(foo, bar) # 更好
foobar = '{foo}{bar}'.format(foo=foo, bar=bar) # 最好
```

提供依赖关系

Runners

更多阅读

- http://docs.python.org/2/library/
- http://www.diveintopython.net/toc/index.html