第 10 章

编写项目文档

文档是常常被开发人员忽略的工作，有时候管理人员也会忽略它。这是因为项目周期即将结束时十分缺乏时间，而且人们觉得自己不擅长写作。虽然有些人确实不擅长此道，但是大部分人都能够制作出精细的文档。

总之，结果就是在忙乱中编写文档造成了无组织的文档。开发人员大部分时候都不喜欢这个工作，当现有的文档需要更新时情况更糟。许多项目只能提供低劣和过期的文档，因为管理人员不知道如何处理它。

但是，在项目一开始就建立文档过程，并且将文档当作代码的模块来处理，会使工作简单一些。遵循一些规则，甚至可以使编写文档变成有趣的工作。

本章将提供一些开始项目文档工作的技巧：

- 最佳实践总结的技术写作的 7 条规则；
- reStructuredText 入门，这是一种在大部分 Python 项目中使用的普通文本标记语言；
- 关于创建良好项目文档的指南。

10.1 技术性写作的 7 条规则

编写良好文档在很多方面比编写代码更容易。大部分开发人员认为它非常难，但是遵循一组简单的规则就可以使它变得比较容易。

这里谈论的不是如何写一本诗集，而是一些用来帮助读者理解设计、API 或建立代码库的任何相关内容的文本。

每个开发人员都能写出这样的素材，本节将提供 7 条适用任何情况的规则，如下所示。

- 分两步编写：先聚焦于思想，然后审查和修整文档。

- 以读者为目标：谁将读这个文档？
- 使用简单的风格：保持简单明了，使用良好的语法。
- 限制信息的范围：每次只介绍一个概念。
- 使用真实的代码示例：应该抛弃 Foos、bars 这些陈腐的用词。
- 保持简单，只要够用即可：写的不是一本书！
- 使用模板：帮助读者养成习惯。

这些规则大部分是 *Agile Documenting* 一书中创造并采用的，这本由 Andreas Rüping 编著的书的主题是为软件项目生成最好的文档。

10.1.1　分两步编写

Peter Elbow 在 *Writing with Power* 中说过，任何人都几乎不可能一次就写出完美的文档。问题是，许多开发人员在文档编写中会尝试直接写出完美的文档。

成功的唯一方法是每写两个句子就停一下并重读一遍，然后做些改正。这意味着需要同时关注内容和文档的风格。这很难，往往效果也不太好。在完全说清楚意思之前，反而花费了许多时间和精力在修饰文本的风格和外形上。

另一个方法是不管文本的样式和组织，而直接关注于其内容，使所有思想都跃然纸上，而不管它是怎么写出来的。开发人员将连续地写作，即使出现语法错误或其他与内容无关的问题时也不停下来。例如，不管句子是否难以理解，只要思想写下来就行，他只是写下自己想要说的和大概的文字组织。

这样做，开发人员可以关注于他想说的，并且可能从他的大脑中获取比一开始想的还多的内容。

进行自由写作的另一个副作用是，与主题不是直接相关的其他思想很容易混入。好的习惯是把这些思想写在另一张纸或另一个屏幕上，这样就不会丢失它们，然后再回到主题上来。

第二步是重新阅读整个文本并对其进行润色，使之对每个人来说都是易于理解的。润色意味着改进文本的风格，改正错误，重新组织，删除冗余信息，等等。

当用于编写文档的时间有限的时候，将这段时间分成两个阶段是个好习惯，一个阶段编写内容，一个阶段清理和组织文本。

先关注于内容，然后才是风格和清晰性。

10.1.2 以读者为目标

在开始编写文档时，作者应该考虑一个简单的问题：谁是本文档的读者？

这个问题的答案并不总是显而易见，因为技术文档用来说明软件的工作方式，常常是写给每个获得和使用这些代码的人看的。读者可能是一个为某个问题寻找合适的解决方案的管理人员，或者一个需要使用它来实现一个功能的开发人员。如果从架构的观点看该软件包符合其需求，设计人员也可能阅读这个文档。

应该应用一个简单的规则：每个文档应该只有一类读者。

这种思想会使文档编写更加容易一些。作者精确地知道他将要面对哪类客户，就可以为所有类型的读者提供简明而精确的文档。

在文档中提供一些简单的介绍性文字，说明文档的相关内容，指导读者找到合适的部分，这是一种良好的习惯，示例如下。

```
Atomisator 可以读取 RSS feed 并将其存储在数据库中，并且带有过滤进程的工具产品。
如果你是一个开发人员，请查看 API 描述（api.txt）;
如果你是一个管理人员，请阅读功能列表和 FAQ（features.txt）;
如果你是一个设计人员，请阅读架构和基础结构说明（arch.txt）。
```

以这种方式来引导读者，能够生成更好的文档。

在开始编写之前，要了解面向的读者。

10.1.3 使用简单的风格

Seth Godin 是销售方面最畅销的作家之一，在 Internet 上可以免费阅读其 *Unleashing the Ideavirus* 一书（http://en.wikipedia.org/wiki/Unleashing_the_Ideavirus）。

最近，他在自己的博客上对他的书籍为什么这么畅销做了一个分析。他制作了一个销售领域所有畅销书的列表，并比较了每本书中每句话的单词数量。

他发现自己的书每句的单词数量最少（13 个单词）。Seth 解释道：这个简单的事实证明，读者喜欢短小精悍的句子，而不是冗长而有格调的。

保持句子短小精悍，那么读者从作品中提取、处理和理解其内容时将消耗较少的脑力。编写技术文档的目标是为读者提供一个软件的指南。它不是一部小说，应该更接近于微波炉说明而不是 Stephen King 的小说。

要记住如下几个技巧。

- 使用简单的句子，句子不应该长于两行。
- 每个段落应该由 3～4 个句子组成，最多表达一个主要的思想，以便文档能够呼吸。
- 不要有太多的重复，避免新闻稿风格——这种风格就是通过不断重复思想以确保读者能够理解。
- 不要使用多种时态，大部分时候用现在时就足够了。
- 如果还不是一个真正的好作家，就不要在文档中开玩笑。技术书籍很难做到有趣，能够掌握该方法的作者也很少。如果你确实希望有点幽默，在代码实例中添加幽默会更好一些。

这不是在写小说，所以应尽量保持简单的风格。

10.1.4　限制信息的范围

不好的软件文档有一个简单的特征：当查找某些已知存在的信息时总是找不到。在花费了一些时间来阅读目录之后，即便开始尝试多种单词的组合搜索，也总是找不到想要找的。

当作者没有按照主题进行组织文档时就会发生这种情况。他们可能提供大量的信息，但是以单一或无逻辑的方式来汇集的。例如，如果读者寻找关于应用程序的总体描述，他就没必要阅读 API 文档——这是一个低水平的问题。

为了避免出现这种效果，段落应该被放在具有意义的标题之下，整个文档的标题应该在一个短句中概述其内容。

目录应该由所有小节的标题组成。

编写标题的简单方法是询问自己：在 Google 中输入什么短语来查找这个小节？

10.1.5　使用真实的代码示例

Foo 和 bar 这样无意义的伪变量是不好的。当读者试图通过用法示例来理解代码的工作方式时，不真实的实例将会使理解变得困难。

为什么不使用一个真实的例子？一个常用的方法是确定每个代码示例可以被剪切并粘贴到实际的程序中。

以下就是一个不好的示例。

有一个解析函数，如下所示。

```
>>> from atomisator.parser import parse
```

其用法如下。

```
>>> stuff = parse('some-feed.xml')
>>> stuff.next()
{'title': 'foo', 'content': 'blabla'}
```

更好的例子应该是，在该解析器知道如何使用解析函数返回一个 feed 的内容时，可作为一个顶级函数使用，如下所示。

```
>>> from atomisator.parser import parse
```

其用法如下。

```
>>> my_feed = parse('http://tarekziade.wordpress.com/feed')
>>> my_feed.next()
{'title': 'eight tips to start with python',
'content': 'The first tip is..., ...'}
```

这个微小的差别可能有点过分夸张，但是实际上它能够使文档更有帮助。读者可以将这些代码行复制到一个 shell 程序中，理解 parse 将使用一个 URL 作为参数，并且返回包含博客条目的一个枚举型变量。

[ 代码示例应该在实际的程序中直接可用。]

10.1.6 保持简单，够用即可

在大部分敏捷方法中，文档并不是首要的。使软件能工作是最重要的事情，其重要性超过了详尽的文档。所以一个好的做法是只编写真正需要的文档，而不是创建一组完备的文档，正如 Scott Ambler 在 *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*（中译版《敏捷建模：极限编程和统一过程的有效实践》）一书中所说的那样。

例如，对于管理员而言，只需用一个文档说明 Atomisator 是如何工作的就足够了。对他们来说，除了需要知道这个工具的配置和运行方法之外没有其他任何需求。这个文档应将范围限制在一个问题上：如何在自己的服务器上运行 Atomisator？

除了读者和范围之外，将每个小节限制在很少的几页之内也是一个好主意。每个小节最多 4 页，这样作者就必须整合他的思想。如果需要更多页，可能就意味着该软件太过复杂而

难以说明和使用。

[ 可以运行的软件胜过面面俱到的文档——敏捷宣言。]

10.1.7　使用模板

Wikipedia 上的每个页面都很相似：左侧的方框用来显示日期或事件摘要；在文档的开始是一个目录，里面有指向本条目中锚点的链接；最后总有一个参考文献小节。

用户习惯于此。例如，他们知道可以快速地看一下目录，如果没有找到所需要的信息，将直接到参考文献小节查看是否可以找到相关主题的另一个网站。这对于 Wikipedia 上的任何页面都是有效的。可以通过学习 Wikipedia way 来获得更高的效率。

所以，使用模板强制文档按照常见的模式，能够使人们更有效地使用它们。他们习惯于这种结构并且知道如何更快地阅读它。

为每类文档提供一个模板也为作者提供了一个快速启动的方案。

本章中，我们将看到各种软件可能拥有的文档，并使用剪贴本（Paster）来为它们提供框架。首先介绍 Python 文档应该使用的标示语法。

10.2　reStructuredText 入门

reStructuredText 也称为 reST（参见 http://docutils.sourceforge.net/rst.html）。它是一种普通文本标记语言，在 Python 社区中被广泛地应用于包文档中。reST 很好的一点是文本仍然是可读的，因为其标记语法不会像 LaTeX 那样使文本变得混乱。

以下是这种文档的一个样例。

```
=====
Title
=====
Section 1
=========
This *word* has emphasis.
Section 2
=========
Subsection
```

```
::::::::::
Text.
```

reST 从属于 docutils 包。这个包提供了一批脚本，可以将 reST 文件转换为各种格式的文件，如 HTML、LaTeX、XML 甚至 S5（Eric Meyer 的幻灯片系统，参见 http://meyerweb.com/eric/tools/s5）。

作者可以只关注于内容，然后根据需要来决定如何显示它。例如，Python 本身的文档就是 reST 格式，最后以 HTML 显示在http://docs.python.org中，此外还有其他各种格式。

开始编写 reST 之前，至少应该知道以下要素：

- 小节结构；
- 列表；
- 内联标签；
- 文字块；
- 链接。

本节是对该语法的概述，更多信息请参考 http://docutils.sourceforge.net/docs/user/rst/quickref.html，这是着手使用 reST 的一个好起点。

要安装 reStructuredText，需要安装 docutils，命令如下。

```
$ easy_install docutils
```

将得到一组名称以 rst2 开始的脚本，可以将 reST 转换为各种格式显示。

例如，rst2html 脚本将把指定 reST 文件转成 HTML 格式的输出，如下所示。

```
$ more text.txt
Title
=====
content.
$ rst2html.py text.txt > text.html
$ more text.html
<?xml version="1.0" encoding="utf-8" ?>
...
<html ...>
<head>
...
</head>
<body>
<div class="document" id="title">
<h1 class="title">Title</h1>
```

```
<p>content.</p>
</div>
</body>
</html>
```

10.2.1　小节结构

文档的标题及其小节使用以非字母字符作为下划线符号的方法表示。它们可以带有上划线和下划线，常见的做法是对标题同时使用上下划线，而对于小节则只使用下划线。

小节、标题的下划线字符根据常用情况排序为：=、-、_、:、#、+、^。

当在小节中使用某个字符时，它与小节的级别相关并且必须在整个文档中保持一致。示例如下（见图 10.1）。

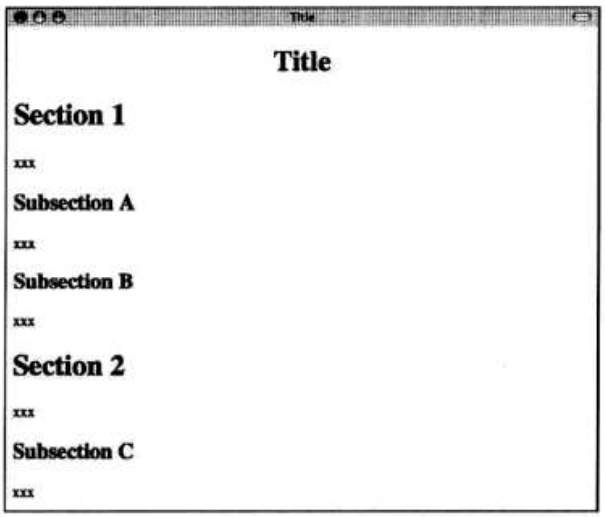

图 10.1

```
=====
Title
=====
Section 1
=========
xxx
Subsection A
------------
xxx
Subsection B
------------
```

```
xxx
Section 2
=========
xxx
Subsection C
------------
xxx
```

这个文件的 HTML 输出如图 10.1 所示。

10.2.2 列表

reST 提供了着重号列表、编号列表和具有自动编号功能的定义列表，如下所示。

```
Bullet list:
- one
- two
- three
Enumerated list:
1. one
2. two
#. 自动编号
Definition list:
one
   one is a number.
two
   two is also a number.
```

10.2.3 内联标签

文本可以使用内联标签来定义样式：

- *emphasis* 斜体；
- **strong emphasis** 黑体；
- ``inline literal`` 内联预格式化文本；
- `a text with a link`_ 只要它出现在文档中，将被一个超链接所替代（参见 10.2.5 小节）。

10.2.4　文字块

当需要介绍一些代码实例时，可以使用文字块。文字块的标识是两个冒号，它是一个带缩进的段落，如下所示。

```
This is a code example
::
  >>> 1 + 1
  2
Let's continue our text
```

别忘了在::和文本块之后添加一个空行，否则它将不会显示。

注意，冒号可以放在文本行中。这时候，在各种显示格式中都将显示为一个冒号，如下所示。

```
This is a code example::
    >>> 1 + 1
    2
Let's continue our text
```

如果不想保留这个冒号，可以在 example 和::之间插入一个空格。这样，::将被彻底删除。

10.2.5　链接

如果要将文本改成一个外部链接，可以在前面添加两个逗点，如下所示。

```
Try `Plone CMS`_, it is great ! It is based on Zope_.
.. _`Plone CMS`: http://plone.org
.. _Zope: http://zope.org
```

通常的做法是将所有的外部链接集中放在文档的末尾。当链接的文本中包含空格时，必须将其放在两个`字符之间。

也可以在文本中添加一个标签来定义内部链接，如下所示。

```
This is a code example
.. _example:
::
    >>> 1 + 1
    2
Let's continue our text, or maybe go back to
the example_.
```

小节也可以当作目标来用，如下所示。

```
=====
Title
=====
Section 1
=========
xxx
Subsection A
------------
xxx
Subsection B
------------
-> go back to `Subsection A`_
Section 2
=========
xxx
```

10.3 构 建 文 档

指导读者和作者的最简单方法是为每个人提供帮助和指南，就像前面介绍的那样。

从作者的角度，这可以通过拥有一组可复用的模板和描述如何、何时在项目中使用这些模板的指南来完成，它被称为文档工件集（documentation portfolio）。

从读者的角度，建立一个文档景观（document landscape），能够轻松地浏览文档并高效地查找信息。

构建工件集

一个软件项目可以包含许多种文档，从低层级的文档（将直接引用代码）到提供应用程序高级概述的设计描述。

例如，Scott Ambler 在其 *Agile Modeling*（参见 http://www.agilemodeling.com/essays/agileArchitecture.htm）一书中定义了一个详尽的文档类型列表。他构建了一个从早期规格说明书到操作文档的一个工件集，甚至包括了项目管理文档，所以整个文档需求可以使用一个标准模板集来构建。

因为一个完整的工作集和构建软件的方法紧密相关，所以本章将只关注能够完成特定需求的公共子集。构建一个高效的工作集需要很长的时间，因为它将捕获开发人员的工作习惯。

在软件项目中，常见的工作集可以分为 3 类：

- 设计　提供架构信息和详细设计信息，如类图、数据库图等；
- 使用　关于如何使用软件的文档，可以是菜谱和教程的形式或模块级帮助；
- 操作　提供如何部署、升级和操作软件的指南。

1．设计

设计文档的目的是描述软件的工作机制和代码的组织方式。开发人员通过它来理解系统，它也是人们理解应用程序工作方法的一个很好的切入点。

一个软件可能有不同类型的设计文档：

- 架构概述；
- 数据库模型；
- 带有依赖性和层次关系的类图；
- 用户界面线框图；
- 基础设施描述。

这些文档大部分由一些图和少量文字组成。这些图所用的约定和团队及项目有关，只要保持一致就很好。

UML 提供了覆盖软件设计大部分方面的 13 种图。类图可能是最常用的，它可以描述软件的各个方面（请参见 http://en.wikipedia.org/wiki/Unified_Modeling_Language#Diagrams）。

完全遵循某种特定的建模语言（诸如 UML）往往是做不到的，团队只是按照自己的经验来进行工作。他们从 UML 或其他建模语言中选择好的方法，并且创建自己的方法。

例如，对于架构概述图，有些设计人员只在白板上绘制方框和箭头，而不遵循什么特殊的设计规则及图形标准。其他人可能会使用诸如 Dia（http://www.gnome.org/projects/dia）或 Microsoft Visio（不是开源软件，所以不是免费的）之类的简单绘图工具，因为这对于理解设

计已经足够了。例如，第 6 章中介绍的所有架构图都是使用 OmniGraffle 制作的。

数据库模型图则取决于所使用的数据库类型。完整的数据建模软件提供了绘制工具，能够自动生成表及表间关系。但是对于 Python 而言，绝大部分时候都显得有些过载。如果使用诸如 SQLAlchemy 这样的 ORM 工具，一些带有字段列表的简单方框，加上第 6 章中所看到的表间关系就足以在开始编写之前描述映射关系了。

类图通常使用 UML 类图的简化版本，例如，在 Python 中不需要指定类的 protected 类型成员。所以，用于架构概要图的工具也能够满足这种需求。

用户界面图取决于打算编写 Web 还是桌面应用程序。Web 应用程序常常只描述屏幕的中心区域，因为页首、页脚、左右面板都是公用的。许多 Web 开发人员会手工绘制这些屏幕快照，然后再用照相机或扫描仪电子化，还有些人会在 HTML 中创建原型并进行屏幕截图。对于桌面应用程序而言，原型屏幕的截图或者使用诸如 Gimp 或 Photoshop 之类的工具制作的带有注解的模拟图是常见的做法。

基础结构概要图和架构类似，但是它们关注于软件与第三方元素（如邮件服务器、数据库以及各种数据流）之间的交互。

2．公用的模板

创建这样的文档时，重要的是确定完全了解目标读者，以及内容的限定范围。这样用于设计文档的通用模板可以为作者提供一个轻巧的结构和一些写作建议。

这样的结构可以包含：

- 标题；
- 作者；
- 标签（关键字）；
- 描述（抽象）；
- 目标（谁应该阅读）；
- 内容（带图形）；
- 对其他文档的引用。

内容应该最多占用 3～4 个页面（1024×768），要确保将其限制在这个范围内。如果范围过大，应该分成几个文档，或者对其进行概括。

模板还提供了作者姓名和一个标签列表，用来管理其发展并且简化分类。这将在本章稍后部分介绍。

剪贴板（Paster）是用来为文档提供模板的正确工具。pbp.skels 实现前面描述的设计模板，而且可以像代码生成一样的使用。提供一个目标文件夹并回答几个问题，如下所示。

```
$ paster create -t pbp_design_doc design
Selected and implied templates:
```

```
pbp.skels#pbp_design_doc A Design document
Variables:
  egg:    design
  package: design
  project:    design
Enter title ['Title']: Database specifications for atomisator.db
Enter short_name ['recipe']: mappers
Enter author (Author name) ['John Doe']: Tarek
Enter keywords ['tag1 tag2']: database mapping sql
Creating template pbp_design_doc
Creating directory ./design
   Copying +short_name+.txt_tmpl to ./design/mappers.txt
```

然后，就将生成以下的结果。

```
=======================================
Database specifications for atomisator.db
=======================================
:Author: Tarek
:Tags: database mapping sql
:abstract:
   Write here a small abstract about your design document.
.. contents ::
Who should read this ?
::::::::::::::::::::::
Explain here who is the target readership.
Content
:::::::
Write your document here. Do not hesitate to split it in several
sections.
References
::::::::::
Put here references, and links to other documents.
```

3．使用

使用文档用来描述软件特定部分的工作模式。这个文档可以描述低层级的部分，比如一个函数是如何工作的，也可以是较高层级的部分，比如调用程序的命令行参数。这是框架应用程序文档中的最重要部分，因为目标读者主要是那些打算复用这些代码的开发人员。

主要的文档有以下 3 种。

- Recipe　解释如何完成某事的一个简短文档。这种文档针对一个读者群体，只关注一个特定的主题。
- 教程　一个渐进地解释如何使用软件功能的文档。这个文档可以引用 recipe，每个实例针对一个读者群体。
- 模块级帮助　说明模块所包含的内容的低层级文档。这个文档可以在调用模块内建的 help 命令时显示。

（1）recipe

一个 recipe 用来回答一个很具体的问题，并且提供相应的解决方案。

例如，ActiveState 提供了一个联机的 Python 说明书（即一个 recipe 集合），开发人员可以描述在 Python 中如何做某事（参见http://aspn.activestate.com/ASPN/Python/Cookbook）。

这些 recipe 必须简单，其结构类似于：

- 标题；
- 提交者；
- 最后更新时间；
- 版本；
- 类别；
- 描述；
- 源代码；
- 讨论（代码的文字解释）；
- 注释（来自 Web）。

recipe 往往只有一个屏幕长，不会特别详细。这个结构完全符合软件的需求，而且可以在一种通用结构中使用，在通用结构中添加目标读者群，用标签替换类别：

- 标题（短语）；
- 作者；
- 标签（关键字）；
- 谁应该阅读这个文档；
- 先决条件（例如其他需要阅读的文档）；
- 问题（简单的描述）；
- 解决方案（主文本，1～2 个页面）；
- 引用（指向其他文档的链接）。

日期和版本在这里都没有用，因为稍后将看到，文档的管理与项目中的源代码管理十分类似。

和 design 模板类似，pbp.skels 提供了 pbp_recipe_doc 模板，它可以用来生成如下这种结构。

```
$ paster create -t pbp_recipe_doc recipes
Selected and implied templates:
   pbp.skels#pbp_recipe_doc A recipe
Variables:
   egg:         recipes
   package:     recipes
   project:     recipes
Enter title (use a short question): How to use atomisator.db
Enter short_name ['recipe'] : atomisator-db
Enter author (Author name) ['John Doe']: Tarek
Enter keywords ['tag1 tag2']: atomisator db
Creating template pbp_recipe_doc
Creating directory ./recipes
   Copying +short_name+.txt_tmpl to ./recipes/atomisator-db.txt
```

然后，writer 将完成如下所示的结果。

```
========================
How to use atomisator.db
========================
:Author: Tarek
:Tags: atomisator db
.. contents ::
Who should read this ?
::::::::::::::::::::::
Explain here who is the target readership.
Prerequisites
:::::::::::::
Put here the prerequisites for people to follow this recipe.
Problem
:::::::
Explain here the problem resolved in a few sentences.
Solution
::::::::
Put here the solution.
References
::::::::::
Put here references, and links to other recipes.
```

（2）教程

教程的用途和 recipe 不同。它不是用来解决一个独立问题的，而是一步一步地描述应用程序各个功能的使用方法。它可能比 recipe 长，并且可以关注应用程序的许多部分。例如，Django 在其网站上提供一系列教程。*Writing your first Django App, part 1*（参见 http://www.djangoproject.com/documentation/tutorial01）就用了 10 个页面来说明使用 Django 构建一个应用程序的方法。

这样一个文档的结构可以是：

- 标题（短句）；
- 作者；
- 标签（单词）；
- 描述（抽象）；
- 谁应该阅读本教程；
- 先决条件（例如其他需要阅读的文档）；
- 教程（主文本）；
- 引用（指向其他文档的链接）。

在 pbp.skels 中提供的 pbp_tutorial_doc 模板也使用了这个结构，这与设计模板相似。

（3）模块帮助

在我们的集合中，最后一个可以加入的模板就是模块帮助模板。模块帮助将引用单个模块，并提供其内容的一个描述，以及用法示例。

一些工具可以通过提取 docstrings 并使用 pydoc 计算模块级帮助来构建这样的文档，如 Epydoc （参见 http://epydoc.sourceforge.net）。所以，根据 API 内省来生成一个大规模的文档也是有可能的。这种文档往往在 Python 框架中提供，例如，Plone 提供一个http://api.plone.org 服务器，用来保存模块帮助的最新集合。

这种方法的主要问题是：

- 没有在真正对文档感兴趣的模块上进行智能选择；
- 代码可能被文档搞乱。

而且，模块文档中提供的示例有时候可能会引用模块的多个部分，很难分清函数和类的 docstring。可在模块的最开始写一段文字，模块的 docstring 可以用来完成这个任务。但是这样做会造成一个由一个文本和之后的代码块组成的混合文件。如果代码的总长度只占到全部的 50%以下，就会显得很混乱。如果是作者，这完全没问题；但是当人们试图阅读代码（而不是文档）时，他们就必须跳过 docstring 部分。

另一种方法是将文本单独放在一个文件中，通过一个手工的选择操作来决定哪个 Python 模块将拥有模块级帮助文件。之后这些文档从代码库中分离出来，并且存在于自己的文件中，就像稍后将要看到的那样。这是 Python 的文档制作方法。

许多开发人员承认，文档和代码分离比使用 docstring 更好一些。这种方法意味着文档处理完全集成到开发周期中了，否则，它将很快变得十分陈旧。docstring 方法通过使代码和用法示例之间保持相近来解决这个问题，但是不要将它带到更高的级别上——一个可以作为普通文档一部分的文档。

用于模块级帮助的模板实际上很简单，因为它在编写的内容之前只包含一点元数据。因为是开发人员想要使用模块，所以目标不需要定义，而只需要：

- 标题（模块名称）；
- 作者；
- 标签（单词）；
- 内容。

下一章将介绍如何使用 doctest 和模级块帮助进行测试驱动开发。

4．操作

操作文档用来描述软件操作的方法。例如：

- 安装和部署文档；
- 管理文档；
- 在出现故障时为用户提供帮助的“常见问题解答”文档；
- 说明如何获得帮助或者提供反馈的文档。

这些文档很具体，但可以使用前面小节中定义的教程模板。

10.4　建立自己的工件集

前面讨论的模板只是用于制作软件文档的一个基础。由此，就像介绍剪贴板的那一章中所说明的那样，可以调整它并添加入其他模板，以建立自己的文档工作集。

记住，项目文档应保持简单，只要够用即可。每个加入的文档应该有一个清晰的目标读者群，应该符合一个实际的需求，而不应该写没有增加真正价值的文档。

构建景观

前一节中构建的文档工件集提供了文档级别的一个结构，但是没有提供一种组织它并为读者创建文档的方法。这就是 Andreas Rüping 称之为文档景观的东西，指的是读者在浏览文档时使用的意境地图。他得出的结论是，组织文档的最好方式就是建立一棵逻

辑树。

换句话说，组成工作集的不同类型文档必须在一棵目录树中找到适合存在的位置。这个位置在作者创建文档和读者查找文档时都应该是显而易见的。

浏览文档时，每个级别上的索引页面是对作者和读者的有效助手。

建立一个文档景观分两步完成：

- 为作者建立一棵树；
- 在作者的树的顶端为读者建立一棵树。

作者和读者之间的区别很重要，因为他们将在不同位置以不同的格式访问文档。

1. 作者的布局

从作者的角度看，每个文档和 Python 的模块一样处理。它应该保存在版本控制系统中，像代码一样运作。

作者不关心文章的最后外观和出现的地方，他们只希望确保编写了一个文档，所以唯一的真理是覆盖所需的主题。

保存在一个文件夹树中的 reStructuredText 文件，在版本控制系统中可以与软件代码一起利用，这是建立作者文档景观的一种方便的解决方案。

如果回头看第 6 章中 Atomisator 的文件夹结构，可以将 docs 文件夹作为这棵树的根。

组织树的最简单方式是根据特性为文档分组，如下所示。

```
$ cd atomisator
$ find docs
docs
docs/source
docs/source/design
docs/source/operations
docs/source/usage
docs/source/usage/cookbook
docs/source/usage/modules
docs/source/usage/tutorial
```

注意，这棵树位于 source 文件夹中，因为 docs 文件夹将在下一小节被用作根文件夹来创建一个特殊的工具。

由此，可以在每个级别添加一个 index.txt 文件（除了根以外），说明文件夹中包含了哪类文档，或者总结每个子文件夹中包含的内容。这些索引文件可以定义其包含文档的列表。例如，operation 文件夹可以包含一个可用操作文档的列表，如下所示。

```
==========
Operations
==========
This section contains operations documents:
- How to install and run Atomisator
- How to install and manage a PostgreSQL database
for Atomisator
```

为了让人们不会忘记更新这些文档，可以自动生成这些列表。

2．读者的布局

从读者的角度，制作索引文件并将整个文档以易于阅读的漂亮格式提交是很重要的。Web 页面是最好的选择，并且它很容易从 reStructuredText 文件中生成。

Sphinx （http://sphinx.pocoo.org）是一组可以用来从文本树生成一个 HTML 结构的脚本和 docutils 扩展。这个工具用来（例如）创建 Python 文档，现在很多项目都使用它来制作文档。使用其内建的功能，可以生成一个真正精细的浏览系统，以及一个简单但足够用的客户端 JavaScript 搜索引擎。它还使用 pygments 来显示代码示例，生成真正漂亮的语法强调显示。

Sphinx 能够简单地配置，以绑定在前面小节中定义的文档景观。

安装时只需要调用 easy_install，如下所示。

```
$ sudo easy_install-2.5 Sphinx
Searching for Sphinx
Reading http://cheeseshop.python.org/pypi/Sphinx/
...
Finished processing dependencies for Sphinx
```

这将安装几个脚本，如 sphinx-quickstart。这个脚本将生成一个脚本和一个 Makefile，可以用于在每次需要时生成 Web 文档。在 docs 文件夹中运行这个脚本，并回答相应的问题，如下所示。

```
$ sphinx-quickstart
Welcome to the Sphinx quickstart utility.
Enter the root path for documentation.
> Root path for the documentation [.]:
> Separate source and build directories (y/n) [n]: y
> Name prefix for templates and static dir [.]:
> Project name: Atomisator
> Author name(s): Tarek Ziadé
```

```
> Project version: 0.1.0
> Project release [0.1.0]:
> Source file suffix [.rst]: .txt
> Name of your master document (without suffix) [index]:
> Create Makefile? (y/n) [y]: y
Finished: An initial directory structure has been created.
You should now populate your master file ./source/index.txt and create
other documentation
source files. Use the sphinx-build.py script to build the docs, like so:
   make <builder>
```

这将在源文件夹添加一个 conf.py 文件，其中包含通过回答问题定义的配置信息，还有根目录下的 index.txt 和 docs 文件夹中的 Makefile。

运行 make html 将在 build 中生成一棵树，如下所示。

```
$ make html
mkdir -p build/html build/doctrees
sphinx-build.py -b html -d build/doctrees -D latex_paper_size= source
build/html
Sphinx v0.1.61611, building html
trying to load pickled env... done
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
creating index...
writing output... index
finishing...
writing additional files...
copying static files...
dumping search index...
build succeeded.
Build finished. The HTML pages are in build/html.
```

现在，生成的文档将被放在 build/html 中，起点是 index.html（见图 10.2 所示）。

除了文档的 HTML 版本之外，该工具还将创建一些自动生成的页面，如模块类别和索引。Sphinx 提供了少量的 docutils 扩展，它是这些功能的驱动力。这些扩展主要包括：

- 一条创建目录的指令；
- 一个可以将文档注册为模块帮助的标记；
- 一个在索引中添加元素的标记。

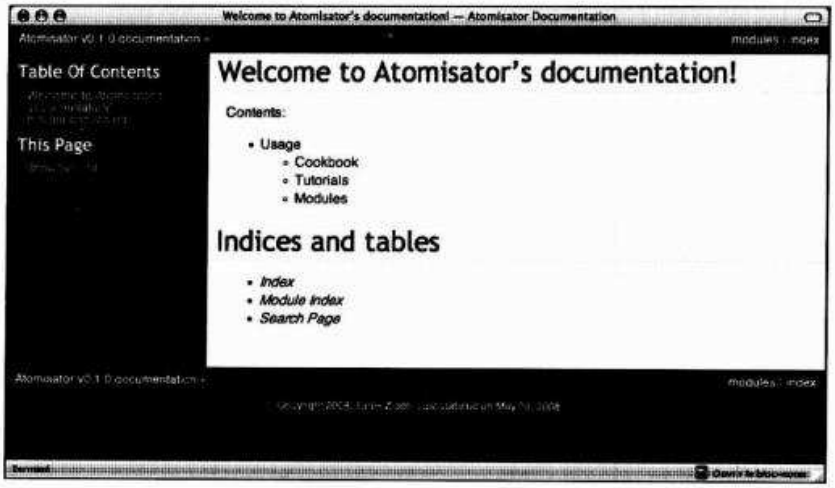

图 10.2

（1）建立索引页面

Sphinx 提供了一条 toctree 指令，可以用来在文档中注入一个目录，以及指向其他文档的链接。每一行必须是一个文件以及针对当前文档的相对路径。也可以使用通配符来添加符合条件的多个文件。

例如，之前已经在作者景观中定义的 cookbook 文件夹所对应的索引文件可以类似于：

```
========
Cookbook
========
Welcome to the CookBook.
Available recipes:
.. toctree::
   :glob:
   *
```

使用这种语法，HTML 页面将显示 cookbook 文件夹中所有可用的 reStructuredText 文档。这条指令可用来为所有索引文件创建可浏览的文档。

（2）注册模块帮助

对于模块帮助，可以添加一个标志，这样它就可以自动在模块索引页面中列出并可用，如下所示。

```
=======
session
=======
.. module:: db.session
The module session...
```

注意，db 前缀可以用来避免模块冲突。Sphinx 将它作为模块的类别，并且将所有以 db. 开头的模块分到这个类别中。

对于 Atomisator，条目分组将使用 db、 feed、main 和 parser，如图 10.3 所示。

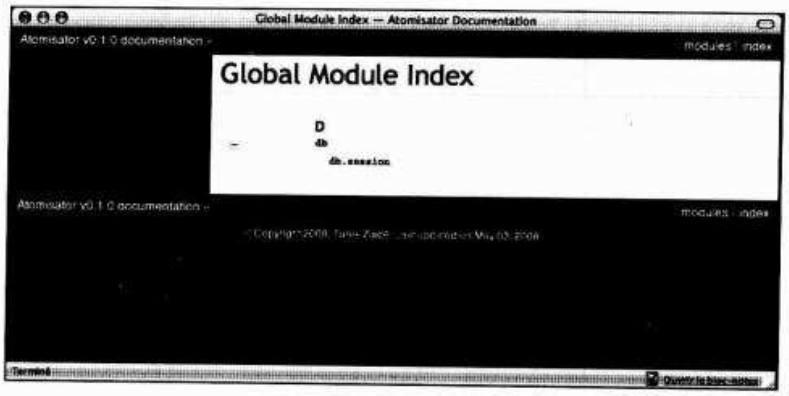

图 10.3

在文档中，可以在有很多模块的时候使用这个功能。

注意，前面创建的模块帮助模板（pbp_module_doc）可以被修改成默认添加 module 指令。

（3）添加 Index 标志

另一个可用的选项是通过将文档链接到一个条目上，填充索引页面，如下所示。

```
=======
session
=======
.. module:: db.session
.. index::
   Database Access
   Session
The module session...
```

这时，将在索引页面中添加两个新的条目：Database Access 和 Session。

（4）交叉引用

最后，Sphinx 提供了一个内联标签来设置交叉引用。例如，一个指向模块的链接可以如下这样完成。

```
:mod:`db.session`
```

:mod:是模块标签的前缀，`db.session`是所链接到的模块名称（前面所注册的）。记住，:mod:和前面的元素都是 Sphinx 在 reSTructuredText 中引入的特殊指令。

Sphinx 提供了许多功能，可以在它的网站上看到。例如，autodoc 功能是个很了不起的选项，能够自动地从 doctest 中提取并建立文档。

具体信息参见http://sphinx.pocoo.org。

10.5 小　　结

本章详细地说明了如何：

- 使用几条规则来高效地编写文档；
- 使用 Python 风格的 LaTeX，即 reStructuredText；
- 构建一个文档工作集和文档景观；
- 使用 Sphinx 生成精巧的 Web 文档。

制作项目文档中最困难的事情是保持文档准确和及时更新，将文档作为代码库的一部分可以大大简化这件事。由此，每当开发人员修改一个模块时，他（她）应该同时修改对应的文档。

这在大型的项目中可能会相当困难，在模块的标题中添加一个相关文档的列表，对此会有帮助。

确保文档始终准确的辅助方法是通过 doctest 将文档和测试合并。

这方面的内容将在下一章中介绍：将介绍测试驱动开发的原则，然后是文档驱动开发。