第 6 章 编写一个应用程序

在前一章中，我们已经了解到一个编写包以及在命名空间中集中代码的可重复方法。可以通过集中一系列的包，并且编写一个将所有这一切联系在一起的包来使它们交互，从而编写出一个 Python 应用程序。

本章将通过一个小的案例来示范如何构建、发行以及分发这样的一个应用程序。

6.1 Atomisator 概述

接下来将实现一个名为 Atomisator 的应用程序。

Atomisator 是一个命令行工具，它能够生成由多个新闻 feed 组合而成的 RSS XML 文件，如下所示。

```
$ atomisator
Reading source http://feeds.feedburner.com/dirtsimple Phillip Eby
10 entries read.
Reading source http://blog.ianbicking.org/feed/ Ian Bicking
10 entries read.
20 total.
Writing feed in atomisator.xml
Feed ready.
```

当调用这个工具时，将根据配置文件中列出的所有数据源从 Web 中读取数据，并将其保存到一个数据库中，然后从数据库生成一个具有最新条目的 XML 文件。除了在数据库中保存所有读取的数据而不进行实时合并之外，这个程序与 Planet（http://www.planetplanet.org）

很相似。

这个工具还可以对条目应用智能过滤器。例如，每读取一个条目时，它可以与现有的条目进行比较以确认没有重复。本章中将会提供这个应用程序的一个轻型版本，以便了解它的构建方法。

本章将提供一个简化的实现，它和实际的 Atomisator 项目并不等价。如果希望获得完整的版本，请查看该项目主页 http://atomisator.ziade.org。

6.2　整 体 描 述

我们要做的第一件事，就是列出组成这个应用程序的所有包。Atomisator 也可以写在单一的包中，但是考虑到可维护性，还是将其分为可以独立演化的不同部件更好一些。

应用前一章中解释过的规则，Atomisator 可以被分割到 4 个包中，如下所示。

- atomisator.parser　一个知道如何读取 feed 并返回一个条目列表的 feed 解析器。
- atomisator.db　负责对存储条目的数据库进行读写访问的包。
- atomisator.feed　一个知道如何使用来自数据库的条目构建兼容 RSS 2.0 的 XML 文件的包。
- atomisator.main　主程序包，它将使用一个配置文件，提供 3 个命令行工具：
  - load_feeds　从各种数据源中读取数据；
  - generate_feed　构建 XML 文件；
  - atomisator　在一个调用中集合前面两个命令。

包之间的交互如图 6.1 所示。

（1）用户通过命令行调用 atomisator.main，要求生成 feed。

（2）atomisator.main 读取配置文件，列出所有要读取的数据源及数据库配置。

（3）atomisator.main 要求 atomisator.parser 从各种数据源读取并返回条目。

（4）atomisator.parser 读取 feed，并以简单的数据结构返回它们。

（5）～（6）atomisator.main 通过 atomisator.db 更新数据库，它将执行一个智能过滤以避免重复添加条目。

（7）atomisator.main 要求 atomisator.feed 基于数据库生成一个 feed。

（8）～（9）atomisator.feed 通过 atomisator.db 读取数据库并创建一个文件。

这个过程中，包之间的相关性定义如图 6.2 所示。

atomisator.parser 和 atomisator.db 是独立的包，应该首先编写。接下来应该编写 atomisator.feed，

紧接着是 main 包，它负责建立所有的交互。

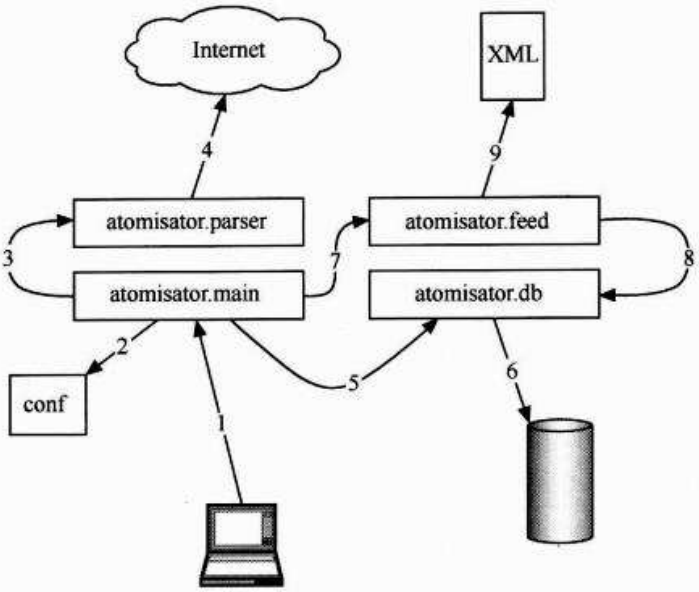

图 6.1

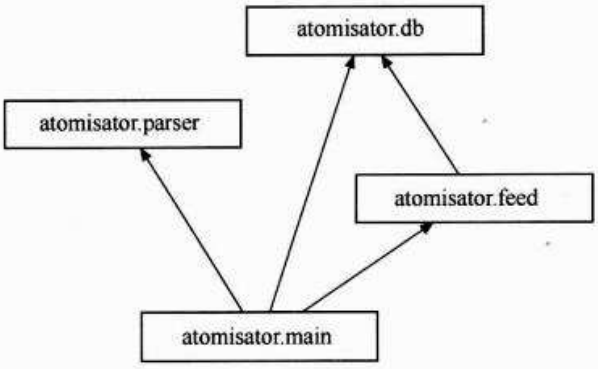

图 6.2

但是第一步，是为 Atomisator 建立一个工作环境。

6.3 工 作 环 境

应用程序中的某些包依赖于其他包，这些相关性可以在 install_requires 元数据中定义。在此，调用 python setup.py develop，读取并安装代码工作所需的所有依赖模型。但是这要求，这些依赖模型在 PyPI 上的可用的 egg，或者已经安装在同一个 Python 安装中。现在不是这种情况，因为将要创建的用于 Atomisator 应用程序的 atomisator.*包都将被一起构建，目前也尚未发布。

当然，通过 develop 命令安装，就能够以正确的依赖顺序完成所有包的安装，但是这将会污染 Python 安装，如果有些依赖性与已经安装的一些包有冲突，还会使得整个系统难以跟踪。

因而，应用程序级别的工作环境应该能够将用于应用程序的所有依赖模型隔离开。

virtualenv （http://pypi.python.org/pypi/virtualenv）项目提供了一个很好的解决方案，它能在已有的 Python 安装之上创建一个新的、独立的 Python 解释程序。

其结果是得到一个本地执行环境，能够自由地安装和开发程序库，以构建出专有的环境，如下所示。

```
mkdir my_env
cd my_env/
$ easy_install -U virtualenv
Searching for virtualenv
Reading http://pypi.python.org/simple/virtualenv/
Best match: virtualenv 1.0
Processing virtualenv-1.0-py2.5.egg
Adding virtualenv 1.0 to easy-install.pth file
...
Finished processing dependencies for virtualenv
$ virtualenv --no-site-packages .
New python executable in ./bin/python
Installing setuptools.............done.
$ ls bin/
activate        easy_install        easy_install-2.5
python      python2.5
```

如果使用的是 Windows 平台，那么这些脚本将生成在 Scripts 目录下，本节中的所有实例都是如此。

virtualenv 命令将生成一个带有新的独立解释程序的文件夹，以及一个 easy_install 脚本和一个 activate 脚本。最后一个脚本只是一个提供方便的脚本，使得能够切换环境变量以使独立的 Python 在系统范围内被调用。--no-site-packages 选项可以用来切断与主 Python 中安装的包之间的依赖性。这个选项在需要一个 Python 裸环境时很有用。

最近刚被接纳的 PEP 370（参见 http://www.python.org/dev/peps/pep-0370）在 Python 中为每个用户添加了一个 site-packages 文件夹，并能构建出与 virtualenv 相同的隔离。Python 2.7 中该特性将可用，它将大大简化定制环境的构建工作。

在新的专用文件夹中为 Atomisator 创建这样一个环境，如下所示。

```
$ mkdir Atomisator
$ cd Atomisator
$ virtualenv --no-site-packages .
New python executable in ./bin/python
Installing setuptools.............done.
```

virtualenv 很了不起的一点是，所有使用本地 Python 解释程序或本地 easy_install 安装的包也将被安装到本地。

6.3.1 添加一个测试运行程序

为了构建我们的应用程序，需要一个测试运行程序。前一章中在包模板中将 Nose 定义为默认的测试运行程序，现在也将它作为全局配置添加到环境中，命令如下。

```
$ bin/easy_install nose
```

这将在使用本地 Python 解释程序的本地 bin 文件夹中添加一个 nosetests 命令。这样，这个测试运行程序能看到所有添加到这个环境中的包。注意，已经对包模板做过了设计，以便使用 test 命令时自动地调用 Nose 测试运行程序。

可以在系统中添加一个符号链接，或者在 PATH 环境变量中添加上 bin 文件夹，以让用户可易于使用。

如果有多个环境，最好是指定特定的名称，并且使它们全局可用。在 Linux 或 Mac OS X 下，可以采用以下方法。

```
$ sudo ln -s bin/nosetests /usr/bin/atomisator-nosetests
$ sudo ln -s bin/python /usr/bin/atomisator-python
```

[ 第 11 章中将说明测试运行程序的定义，并且对比一些产品。]

6.3.2 添加一个包结构

目前，我们的 Atomisator 文件夹中有一个带有 Python 解释程序和一个测试运行程序的 bin 文件夹。我们打算构建的包应该集中在 packages 子目录下，以便简化版本系统的跟踪，并推进其部署：packages 中的所有文件夹将是用于 Atomisator 应用程序的定制 Python 包。

这个初始结构足以让我们着手开始编写包代码。

6.4　编写各个包

按照前面的图示，各个包将按照依赖顺序进行构建，也就是：

（1）atomisator.parser；

（2）atomisator.db；

（3）atomisator.feed；

（4）atomisator.main。

6.4.1　atomisator.parser

Python 中读取 RSS 2.0 的标准工具是 Universal Feed Parser（http://www.feedparser.org）。许多需要从 feed 中提取条目的程序都在使用它，而不管这些 feed 是以 RSS 1.0、RSS 2.0 或 Atom 格式提供的。对于我们的需求来说，这是个完美的工具。

不需要特别的封装就可以直接使用这个工具，但是控制外部程序在一个程序中的使用方式是一个好习惯。确认外部包中的所有调用都是从相同的定制包或模块中发起的，这样能使系统不断演化时更容易重构。这个工作只需要在代码库的一个地方就能完成，因此与应用程序的其余部分的依赖性更容易控制。

包封装器有如下两种类型。

- 漏封装器（Leaky wrappers）：它们在外部包之上提供一个只用于发布外部包的包。它可以是一个简单的导入程序，也可以是外部程序库 API 周边的几个辅助类（参见第 14 章中介绍的外观设计模式）。
- 全封装器（Full wrappers）：它们像程序库之外的黑盒子，提供全功能的 API。

后者可能是确保外部程序库和程序其余部分没有依赖关系的最佳方法。但是，这常常意味着必须编写很多额外的代码来遮蔽它。当项目开始时要正确地使用它也很困难。一个智能的 Leaky 封装器 API 常常更简单，而且能更好地避免重新发明轮子。

对于我们的 feed 解析器而言，选择很简单：创建一个具有单一外观函数的封装器，因为 Universal Feed Parser 将返回基本类型。

编写这样一个包的过程如下：

（1）使用合适的模板创建初始包；

（2）创建初始的 doctest，描述该示例如何使用；

（3）构建测试环境；

（4）编写代码并调整初始的 doctest。

1．创建初始包

该包将在 packages 文件夹中使用 pbp_package 模板创建，命令如下。

```
$ cd Atomisator/packages
$ paster create -t pbp_package atomisator.parser
```

这个命令将生成包结构。现在，该包可以调用 develop 命令链接到解释程序中去，如下所示。

```
$ cd atomisator.parser
$ atomisator-python setup.py develop
running develop
...
Finished processing dependencies for atomisator.parser==0.1.0
```

因为我们的模板中有一个位于 atomisator/parser/README.txt 的默认 doctest，在空包上运行带--doctest-extension=.txt 选项的 nosetest 就将运行这个文件，如下所示。

```
$ atomisator-nosetests --doctest-extension=.txt
----------------------------------------------------------------------
Ran 1 test in 0.162s
OK
```

这些测试可能已经通过 python setup.py test 启动了，但是使用全局的 nosetests 脚本更容易添加一些选项，如果需要可以在多个包上运行测试。

注意，如果在主目录中添加了一个.noserc 文件，可以省略 doctest-extension 选项（将在第 11 章中解释）。

接下来，与 feedparser（PyPI 上的全局 Feed 解析器名称）之间的依赖关系将被添加到 atomisator.parser/setup.py 文件中，如下所示。

```
...
setup(name='atomisator.parser',
      version=version,
      description=("A thin layer on the top of "
                   "the Universal Feed Parser"),
      long_description=long_description,
      classifiers=classifiers,
      keywords='python best practices',
```

```
        author='Tarek Ziade',
        author_email='tarek@ziade.org',
        url='http://atomisator.ziade.org',
        license='GPL',
        packages=find_packages(exclude=['ez_setup']),
        namespace_packages=['atomisator'],
        include_package_data=True,
        zip_safe=False,
        install_requires=[
            'setuptools',
            'feedparser'
            # -*- Extra requirements: -*-
    ],
    entry_points="""
    # -*- Entry points: -*-
    """,
    )
...
```

再次运行 atomisator-python setup.py develop 命令，将从 PyPI 获得 feedparser 并将其链接到环境中。然后，就可以开始编写初始包了。

2．创建初始的 doctest

位于 atomisator.parser/atomisator/parser/的 README.txt 文件是人们在使用包时将参考的文档，其内容是一小段说明包的用途和使用范例的文字。

因为它是一个 doctest，所以可以帮助构建使用 reSTructuredText 正确的测试驱动开发的实际代码。

第 11 章中将详细说明如何编写测试，而 reSTructuredText 的格式将在第 10 章中介绍。

这个文件的第一稿可以如下所示。

```
================
atomisator.parser
================
```

```
The parser knows how to return a feed content, with
the 'parse' function, available as a top-level function::
   >>> from atomisator.parser import parse
This function takes the feed url and returns an iterator
over its content. A second parameter can specify a maximum
number of entries to return. If not given, it is fixed to 10::
   >>> res = parse('http://example.com/feed.xml')
   >>> res
   <generator ...>
Each item is a dictionary that contain the entry::
   >>> res.next()
```

这段文本指定的元素足够开始构建这个包。确保如果调用 bin/test 脚本，那么将执行它，并且会抛出一个错误，因为现在还没有任何代码，如下所示。

```
$ atomisator-nosetests --doctest-extension=.txt
...
File "atomisator.parser/atomisator/parser/docs/README.txt", line 8, in
README.txt
Failed example:
   from atomisator.parser import parse
Exception raised:
  Traceback (most recent call last):
    ...
    File "<doctest README.txt[0]>", line 1, in ?
      from atomisator.parser import parse
  ImportError: cannot import name parse
----------------------------------------------------------------------
Ran 1 test in 0.170s
FAILED (failures=1)
```

现在，这种通过修改代码直到测试通过的测试驱动开发是很容易的。

3. 构建测试环境

当创建了一个包时，确保其包含的所有测试可以在没有任何外部依赖模型的情况下启动是金科玉律。在测试装置中需创建许多虚拟函数来模拟所有指向外部元素的调用，有时候这很难做到。例如，一个依赖于 LDAP 服务器的包应该得到真实的数据才能被正确地构建和测试。这种情况下，从一个真实的服务器开始并记录其输出是一个好的习惯。然后，可以用一

个虚拟的服务器来处理这些收集到的数据。

当虚拟部件的创建很复杂时，也可以使用模拟对象。有关这种方法的详细信息请参见第 11 章。

对于 atomisator.parser，避免调用 URL 的最简单方法是使用一个普通的 XML 文件，因为 feedparser 也支持它，从而使包依赖于一个 Web 连接。获取一个 feed 并将它保存在测试文件夹中，文件名为 sample.xml，如下所示。

```
cd atomisator/parser/tests
wget http://ziade.org/atomisator/sample.xml
```

这是为本书所制作的一个简单 feed。所以，稍后的实例在测试中看起来会很类似，但是任何其他的 feed 也能满足需要。

可以对 README.txt 做相应修改以便使用它，如下所示。

```
...
>>> res = parse(os.path.join(test_dir, 'sample.xml'))
...
```

现在，这个包可以独立于 Web 连接进行测试了。

4．编写代码

由此，可以添加一个 Parse 函数到包中，并且一直构建到测试通过。最后的形式如下。

```
from feedparser import parse as feedparse
from itertools import islice
from itertools import imap

def _filter_entry(entry):
    """Filters entry fields."""
    entry['links'] = [link['href'] for link in entry['links']]
    return entry
def parse(url, size=10):
    """Returns entries of the feed."""
    result = feedparse(url)
```

```
    return islice(imap(_filter_entry,
                       result['entries']), size)
```

调整后的 doctest 是：

```
...
Each item is a dictionnary that contain the entry::
    >>> entry = res.next()
    >>> entry['title']
    u'CSSEdit 2.0 Released'
The keys available are:
    >>> keys = sorted(entry.keys())
    >>> list(keys)
    ['id', 'link', 'links', 'summary', 'summary_detail',
    'tags', 'title', 'title_detail']
```

6.4.2 atomisator.db

按照相同的原则来构建 atomisator.db 包。添加一个新的名为 atomisator.db.的包，并且连接到本地解释程序上。

注意，在使用 Nose 测试运行程序时，如果在相同的命名空间中有多个包，将会运行所有的测试。所以，将在进行 atomisator.db 测试时也运行 atomisator.parser 测试。虽然这往往不是一个问题，但是关注于一个特定的包时可能会想使用过滤选项。

本小节剩余的内容中将着重介绍使用数据库的常见方式，即 SQL 相关的元素。

1. SQLAlchemy

Python 中使用关系数据库最方便的方法是使用 SQLAlchemy（http://www.sqlalchemy.org），这是一个对象—关系映射程序（参见http://en.wikipedia.org/wiki/Object-relational_mapping）。这个工具提供一个能够实现 Python 对象和 SQL 数据库表中的行同步的映射系统，而且不需要编写任何代码。

SQLAlchemy 有多个可用的数据库后端，也就是可以使用多种数据库，如 PostgreSQL、SQLite、MySQL 甚至 Oracle。这种方法的一种优秀特性是，不管切换成哪种后端系统，都只需要使用相同的代码。这意味着在测试环境中可以使用一个简单的 SQLite 文件来构建连接器，而在生产环境中则使用 PostgreSQL，如下所示。

```
>>> if big_daddy_server:
...     sqluri = 'postgres://tarek@localhost/database'
```

```
... else:
...     sqluri = 'sqlite://relative/path/to/database.db'
```

当然，这种特性只限于对所有数据库系统共用的并且为 SQLAlchemy 所包含的操作。但是，只要所提供的 API 被用于与数据库交互，这种切换就是可能的。这在必须使用诸如为了优化而定义的存储过程之类的特殊调用时是很难做到的。但是对于大部分数据库中的东西，Python 包都可以使用 SQLite 作为测试数据库来构建。

因此 SQLAlchemy 可以被看作 Python 的一个通用数据库使用程序。社区中的许多 Python 项目依赖于这个工具，它现在已经被认为是处理数据库的最佳方法之一。另一个相似的项目也可以考虑，这就是来自 Ubuntu 制造商 Canonical 的 Storm（https://storm.canonical.com）。这个工具使用一个隐含的映射系统，而 SQLAlchemy 依赖一个必须在 Python 端定义的显式映射系统，用来描述 Python 对象与 SQL 数据库表的链接方式。

2．创建映射

atomisator.db 数据库模型相当简单，因为它只包含一个条目表、一个链接表和一个标记表。图 6.3 提供了一个简化的数据库视图。主表 atomisator_entry 中包含 atomisator.parser 所提供的 feed 条目以及当前日期，atomisator_link 和 atomisator_tag 是保存一系列唯一值以及使条目指向它们（links 和 tags 字段）的两个辅助表。SQLAlchemy 自动地提供了这些一对多关系，这样一个结构可以被描述为图 6.3 所示的模型。

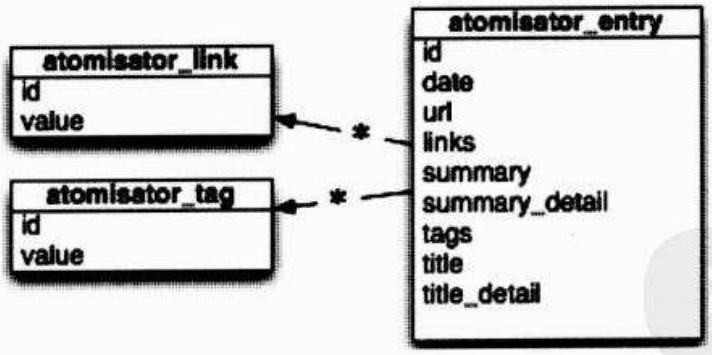

图 6.3

```
from sqlalchemy import *
from sqlalchemy.orm import *
from sqlalchemy.orm import mapper
metadata = MetaData()
link = Table('atomisator_link', metadata,
             Column('id', Integer, primary_key=True),
             Column('url', String(300)),
```

```
        Column('atomisator_entry_id', Integer,
               ForeignKey('atomisator_entry.id')))
class Link(object):
    def __init__(self, url):
        self.url = url
    def __repr__(self):
        return "<Link('%s')>" % self.url
mapper(Link, link)
tag = Table('atomisator_tag', metadata,
            Column('id', Integer, primary_key=True),
            Column('value', String(100)),
            Column('atomisator_entry_id', Integer,
                   ForeignKey('atomisator_entry.id')))
class Tag(object):
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return "<Tag('%s')>" % self.value
mapper(Tag, tag)
entry = Table('atomisator_entry', metadata,
              Column('id', Integer, primary_key=True),
              Column('url', String(300)),
              Column('date', DateTime()),
              Column('summary', Text()),
              Column('summary_detail', Text()),
              Column('title', Text()),
              Column('title_detail', Text()))
class Entry(object):
    def __init__(self, title, url, summary, summary_detail='',
                 title_detail=''):
        self.title = title
        self.url = url
        self.summary = summary
        self.summary_detail = summary_detail
        self.title_detail = title_detail
    def add_links(self, links):
        for link in links:
```

```
            self.links.append(Link(link))
    def add_tags(self, tags):
        for tag in tags:
            self.tags.append(Tag(tag))
    def __repr__(self):
        return "<Entry(%r)>" % self.title
mapper(Entry, entry, properties={
        'links':relation(Link, backref='atomisator_entry'),
        'tags':relation(Tag, backref='atomisator_entry'),
        })
```

本书中不打算详细地研究这些代码，可以通过阅读官方教程（http://www.sqlalchemy.org/docs/04/ormtutorial.html）来了解更多信息。而且，SQLAlchemy 是非常活跃的框架，所以到本书付印时，在这里列出的代码可能就不是最好的做法了。

理解数据库上的每个数据库表将与 Python 端的一个类相连是很重要的一点。每个类的实例将表示数据库表中的一行。

atomisator.db 和本书创建的所有包类似，可以在 PyPI 找到。最近的版本已经完成，但是还有与此不同的演化版本。

3．提供 API

在这些映射之上，API 必须提供添加和查询条目的方法。最后，和代码一起构建的主 doctest 看上去如下所示。

```
=============
atomisator.db
=============
This package provides a few mappers to store feed entries
in a SQL database.
The SQL uri is provided in the config module::
    >>> from atomisator.db import config
    >>> config.SQLURI = 'sqlite://:memory:'
Let's create an entry::
    >>> from atomisator.db import create_entry
    >>> entry = {'url': 'http://www.python.org/news',
    ... 'summary': 'Summary goes here',
```

```
    ... 'title': 'Python 2.6alpha1 and 3.0alpha3 released',
    ... 'links': ['http://www.python.org'],
    ... 'tags': ['cool', 'fun']}
    >>> id_ = create_entry(entry)
    >>> type(id_)
    <type 'int'>
We get the database id back. Now let's look for entries::
    >>> from atomisator.db import get_entries
    >>> entries = get_entries() # returns a generator object
    >>> entries.next()
    <Entry('Python 2.6alpha1 and 3.0alpha3 released')>
Some filtering can be done ::
    >>> entries = \
    ...     get_entries(url='http://www.python.org/news')
    >>> entries.next()
    <Entry('Python 2.6alpha1 and 3.0alpha3 released')>
When no entry is found, the generator is empty::
    >>> entries = get_entries(url='xxxx')
    >>> entries.next()
    Traceback (most recent call last):
    ...
    StopIteration
```

这个包将提供两个全局函数，用来完成数据库的处理：

- get_entries　返回可过滤的条目；
- create_entry　添加一个条目。

6.4.3 atomisator.feed

atomisator.feed 使用 atomisator.db 读取最新的条目，并且生成一个在 RSS 中表示它们的 XML 文件。这可以通过前一章中用于创建代码骨架的 Cheetah 模板引擎来完成。RSS 模板文件实现这个 RSS 2.0 结构，如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntaxns#">
<channel>
<title><![CDATA[${channel.title}]]></title>
<description><![CDATA[${channel.description}]]></description>
```

```
<link>${channel.link}</link>
<language>en</language>
<copyright>Copyright 2008, Atomisator</copyright>
<pubDate>${publication_date}</pubDate>
<lastBuildDate>${build_date}</lastBuildDate>
#for $entry in $entries
  <item>
    <title><![CDATA[${entry.title}]]></title>
    <description><![CDATA[${entry.summary}]]></description>
    <link><![CDATA[${entry.url}]]></link>
    <pubDate>${entry.date}</pubDate>
  </item>
#end for
</channel>
</rss>
```

条目内容由数据库提供，诸如频道标题这样的额外信息由配置给出。这个包的 doctest 看上去如下所示。

```
===============
atomisator.feed
===============
Generates a feed using a template::
    >>> from atomisator.feed import generate
    >>> print generate('feed', 'the feed', 'http://link')
    <?xml version="1.0" encoding="utf-8"?>
    <rss version="2.0" xmlns:rdf="...">
    <channel>
    <title><![CDATA[feed]]></title>
    <description><![CDATA[the feed]]></description>
    <link>http://link</link>
    <language>en</language>
    ...
    <item>
       <title><![CDATA[Python 2.6alpha1 and
                       3.0alpha3 released]]></title>
       <description><![CDATA[Summary goes here]]></description>
       <link><![CDATA[http://www.python.org/news]]></link>
```

```
        <pubDate>...</pubDate>
    </item>
    ...
    </channel>
    </rss>
```

这个包可以供 main 包使用，用来刷新 feed 的条目。

6.4.4 atomisator.main

main 包通过读取一个名为 atomisator.cfg 的配置文件来将所有东西组合在一起，这个配置文件提供了一个 feed 列表和几个全局变量，如下所示。

```
[atomisator]
# 要读取的 feed
sites =
    sample1.xml
    sample2.xml
# 数据库位置
database = sqlite:///atomisator.db
# 用于频道的字段
title = My Feed
description = The feed
link = the link
# 生成文件的名字
file = atomisator.xml
```

这个文件被专门的模块 config 所读取。由此，主模块将提供 3 个方法来组合拼图的每个小块，如下所示。

```
from atomisator.main.config import parser
from atomisator.parser import parse
from atomisator.db import config
from atomisator.db import create_entry
from atomisator.feed import generate
config.SQLURI = parser.database
def _log(msg):
    print msg
def load_feeds():
```

```
    """Fetches feeds."""
    for count, feed in enumerate(parser.feeds):
      _log('Parsing feed %s' % feed)
       for entry in parse(feed):
           count += 1
           create_entry(entry)
    _log('%d entries read.' % count+1)
def generate_feed():
    """Creates the meta-feed."""
    _log('Writing feed in %s' % parser.file)
    feed = generate(parser.title,
                    parser.description, parser.link)
    f = open(parser.file, 'w')
    try:
       f.write(feed)
    finally:
       f.close()
    _log('Feed ready.')
def atomisator():
    """Calling both."""
    load_feeds()
    generate_feed()
```

然后，它们在 setup.py 中连接成控制台脚本，如下所示。

```
...
entry_points = {
    "console_scripts": [
       "load_feeds = atomisator.main:load_feeds",
       "generate_feed = atomisator.main:generate_feed",
       "atomisator = atomisator.main:atomisator"
   ]
}
...
```

这将在系统中添加 3 个新的指向这 3 个函数的可执行脚本。

这个包还在 setup.py 中将其他 atomisator 定义为依赖模块，以便在安装 atomisator.main 的同时安装它们，如下所示。

```
...
```

```
        install_requires=[
            'atomisator.db',
            'atomisator.feed',
            'atomisator.parser'
        ],
...
```

6.5 分发 Atomisator

这个程序现在可以通过将 egg 放入 PyPI 中的方法来分发。每个包可以使用 sdist、bdist 或 bdist_egg 命令来作为 egg 发行。对于我们的应用程序，因为没有需要编译的代码，因此一个源代码分发版本就足够满足所有平台的需求。

对每个 egg，register 和 upload 命令可以和 sdist 一起调用，但是之前必需要按照前一章中的方法创建了一个用户账户。register sdist upload 命令序列将在 PyPI 注册这个包，构建一个源分发版本，并且上传，如下所示。

```
$ cd atomisator.main
$ python setup.py register sdist upload
Using PyPI login from /Users/tarek/.pypirc
Registering atomisator.parser to http://pypi.python.org/pypi
Server response (200): OK
running sdist
...
running upload
Using PyPI login from /Users/tarek/.pypirc
Submitting dist/atomisator.parser-0.1.0.tar.gz to http://pypi.python.org/
pypi
Server response (200): OK
```

之后，该包就在 PyPI 上可用。

调用 alias 命令为每个包创建一个 release 别名也是一个好习惯，如下所示。

```
$ python setup.py alias release register sdist upload
running alias
Writing setup.cfg
$ more setup.cfg
[aliases]
```

```
release = register sdist upload
```

release 命令可以用来推送这个包。在 atomisator.db 中完成这一任务，并且为其他包也完成这项工作，如下所示。

```
$ cd atomisator.db
$ python setup.py alias release register sdist upload
running alias
Writing setup.cfg
$ python setup.py release
running register
Using PyPI login from /Users/tarek/.pypirc
Registering atomisator.db to http://pypi.python.org/pypi
Server response (200): OK
...
running upload
Using PyPI login from /Users/tarek/.pypirc
Submitting dist/atomisator.db-0.1.0.tar.gz to http://pypi.python.org/pypi
Server response (200): OK
$ cd ../atomisator.feed/
$ python setup.py alias release register sdist upload
...
$ python setup.py release
...
$ cd ../atomisator.parser
...
```

这样，这 4 个包就在 PyPI 上可用了。如果在 PyPI 中尝试搜索“atomisator”（http://pypi. python.org/pypi?%3Aaction=search&term=atomisator&submit=search），就应该能够找到所有 4 个包。

现在从任何安装了 setuptools 的计算机上，都可以使用以下命令安装 Atomisator。

```
$ easy_install atomisator.main
```

这个命令将安装所有的包及其依赖模块，并且使 atomisator.main 中的 3 个命令立即可用。

6.6　包之间的依赖性

以多个包的形式分发应用程序，在发行的时候会产生一些开销。

例如，如果在 atomisator.db 中做了一些影响 atomisator.main 的修改，就必须：

- 升级每个包的版本；
- 确保新的 atomisator.main 获得正确的 atomisator.db 版本；
- 重新发行两个包。

版本依赖性可以在 setup.py 中的 install_requires 元数据中直接配置。例如，如果 atomisator.main 的 1.4.6 版本需要至少 1.4.4 版本的 atomisator.db 才能运行，那么它的 setup.py 的内容将如下所示。

```
version = '1.4.6'
...
      install_requires=[
          'atomisator.db>=1.4.4',
          'atomisator.feed',
          'atomisator.parser'
      ],
...
```

要减少这种开销，可以通过编写几个脚本自动化应用程序来实现。在任何情况下，将应用程序分割为几个包要比依赖于单个包更好一些。如果每个包表现应用程序的一个逻辑部分，那么每一个都将按照自己的计划来进行演化。同时，如果两个包总是同时被修改，那么可能意味着它们应该被合并到一个包里。

在开始编码时不要太担心应用程序如何分割。如果分割不正确，那么在开发期间出现问题，将总是能够通过重构代码来修复问题。

6.7 小　结

本章介绍了一个示例应用程序的实现，这个应用以多个 PyPI 上相同命名空间下的包来分发。

我们使用 virtualenv 创建了一个工作环境，并讨论了与所开发的包一起工作的测试运行程序的设置方法。

要创建更大的程序时，必须完成一些 Python 安装之外的包安装器的工作。

下一章将进一步介绍这一主题，并且提出 zc.buildout，这是一个用于构建应用程序环境的常用工具。