20 基于SQL的数据导出机制

更新时间：2019-06-17 10:56:38

> 勤能补拙是良训，一分辛劳一分才。
>
> ——华罗庚

豆瓣读书的数据是结构化的，而且数据量并不像网易那样庞大所以更适宜采用SQL数据库作为爬取数据的后端存储。本节将深入 scrapy 的数据**后处理**机制，了解 scrapy 是怎样将`Item`数据导出成不同格式的数据文件。并巧妙地运用此机制为 scrapy 加装一个将`Item`数据直接导出至 sqlite,mysql,postgresql 这些流行的 SQL 数据库之中。

采用 SQL 存储的考虑与理由

在第4章我们就基于Scrapy`Pipline`(管道)技术实现将数据项存储到MongoDB的定义**数据管道**，采用MongoDB的最大好处是它的使用极为之简单，在python内与MongoDB的交互就是普通的JSON数据读写，这样可以让没有任何数据库基础的读者都能上手。而NoSQL数据库其实只在**高频读**的场景最合适并不适宜于**高频写**的应用；其次，对爬取后的数据要进行批量的统计与查询NoSQL并没有SQL方便。

再者，数据库是后端开发人员永远绕不开的一个重要技术主题，我也想借此能为一些刚开始接触Python的读者打开一扇通向学习数据库技术的大门，在学习爬虫技术后也能将其中的内容应用到其它领域的开发中。

什么是导出器(Feed Exportor)

前文我之所以用管道来实现一个MongoDB的数据存储是因为管道是Scrapy的一种标准插件，可以由Scrapy去调用它，只要实现一次以后就可以通过配置将它"插入"到其它的项目里面重用。

其实Scrapy还有另一种机制是专门负责数据的输出的，还记得我们从第一章中就介绍过通过指令将数据输出到JSON文件：

```
(venv) $ scrapy crawl <蜘蛛名> -o 数据文件.json
```

当我们需要将爬取后的数据项进行持久化存储或导出数据时，就可以采用导出器来完成这一步骤。

导出器(Feed Exportor) 并不是一个具体的类，而是Scrapy中的一种数据导出的处理机制，这种机制由两个具体部分组成：

1. 数据项导出器（Item Exporter）
2. 存储端 （Storage backends）

数据项导出器(Item Exporter)

数据项导出器负责对数据项`Iitem`进行**序列化**处理，也就是将对象实例转化并保存到某一指定的可储格式的文件中。

> **序列化**——可以说是面向对象语言中必备的功能，一般意义上的序列化只是将对象实例与类型信息直接转化为二进制流的形式，这样就能将对象的“状态”存储起来，以后用的时候再通过反序列化将其从文件中恢复成对象实例，这个操作有点像科幻片中人类被冰冻与解冻的过程。

Scrapy默认提供了可以输出至 XML、CSV、JSON及Python二进制对象文件的几种数据导出器。

我们都可以通过 `scrapy crawl` 指令的`-o`参数来使用它们而无须编写任何的代码，例如我们可以把第2章中的新闻供稿爬虫的结果输出到xml中：

```
$ scrapy crawl news -a url='http://www.chinanews.com/rss/scroll-news.xml' -o chinanews.xml
```

又或者输出成为 CSV:

```
$ scrapy crawl news -a url='http://www.chinanews.com/rss/scroll-news.xml' -o chinanews.csv
```

上述的这个过程就Scrapy识别了输出文件的后缀名自动匹配了一种文件格式的数据项导出器将`Item`的数据实例转化为对应的文本格式，然后使用默的存储端(文件)将文件进行输出。

我们可以将`-o`参数的内容写入到`settings.py`配置文件中，每次运行就不需要打输出文件名了:

```
FEED_URI='file:///export.csv'
```

> Windows 下不能使用相对路径:
>
> ```
> FEED_URI='file:///c:\export.csv'
> ```

存储端(Storage backends)

存储端就是将数据导出器输出结果保存到不同协议的输出上。Scrapy提供以下4种标准的存储端：

协议	说明
`file://`	本地文件系统
`s3://`	Amazon S3存储（URI中可带有用户名与密码参数）
`ftp://`	标准FTP的地址格式（URI中可指定FTP用户名与密码）
`stdout:`	标准输出

Scrapy是通过`FEED_URI`中指定的存储协议来匹配使用哪一种存储端的，例如我们可以将文件保存到FTP服务器的话就可以在`FEED_URI`上这样来指定:

```
FEED_URI='ftp://username@password:/home/export.csv'
```

事实上，这些存储端都没有什么具体实用性。我们可以按需要来改造成为我们需要的存储端，例如将PostgreSQL作为存储端。

我们能不能使用Scrapy的数据导出器机制在配置文件上这样定义就能将数据导入到SQL的数据库呢？

例如，用`FEED_URI`来指定连接数据库的连接串(Connection String)：

```
FEED_URI = 'postgres://postgres:@localhost:10086/postgres'
```

设计基于SQL的数据导出机制

在进行设计之前我们需要先学习一下在python中如何来访问数据库的基础知识。在Linux世界中比较流行的数据库有SQLite、MySQL、PostgreSQL等，那么是不是对每一种SQL数据库都要学习一种工具库呢？当然不是了！Python有一个非常著名的数据工具SQLAlchemy，掌握了它就能操控各种各样的SQL数据库了。

> 如果你是一位完全没有数据库基础的读者，那么我建议你可以到菜鸟学院的SQLite栏目或MySQL教程栏目先学习一些数据库的基础知识然后再继续下面内容的阅读，这会对你对下面示例内容的理解有很大帮助。

SQLAlchemy 的使用简介

由于本专栏的篇幅所限，而且数据库与SQLAlchemy的相关话题所涉及的范围极广，难以用短短的字数将其囊括。考虑到一些没有接触过SQLAlchemy的读者，在此会简单地以代码为例介绍SQLAlchemy的最基本用法，更具体的内容可以到SQLArchemy的官网中仔细学习。对于已经熟悉SQLAlchemy读者可以直接跳过本段进入下一部分的内容。

SQLAlchemy是Python编程语言下的一款开源软件。提供了SQL工具包及对象关系映射（ORM）工具。

SQLAlchemy的设计理念是：SQL数据库的量级和性能重要于对象集合；而对象集合的抽象又重要于表和行。

因此，SQLAlchmey采用了类似于Java里Hibernate的数据映射模型。

安装SQLAlchmey

```
(venv) $ pip install sqlalchmey
```

由于ORM是一种模式，也就是说，只要是ORM，其使用方法都是大同小异的，归纳起来有以下4步：

1. 建模 - 建立与数据库对等的数据对象模型。
2. 建立数据连接并产生数据上下文——通过连接字符串与指定数据库建立连接并取得可操作当前数据库的上下文对象。
3. 操作数据 - 通过数据库上下文对象对数据进行增加、删除、修改、查询等常规操作。
4. 提交更改 - 将变更后的数据内容提交并永久性写入数据库。

第一步：建模

SQLAlchemy 定义对象模型后可以调用`db.create_all()`函数就可以自动在数据库中建立与模型对应的数据表。

首先用`declarative_base()`建立数据表的基类，然后就实体就可以从此基类继承。这个类必须定义`__tablename__`用于声明该类与数据库中进行映射的表名。每个性属性将对应数据表的一个列(字段)。

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer

# 创建对象的基类:
db = declarative_base()

class Book(db):
    __tablename__ = 'books'
    id = Column(Integer, primary_key=True)
```

用`Column`类定义列，第一个参数为数据类型，上述代码中的`primary_key=True`表示`id`字段是一个主键，每个数据表都应该有一个唯一值的主键，当定义`Integer`类型的主键后该值性我们并不需要赋值，数据库会自动将其设置为**自增值**每次新增数据时这个字段将会被自动填充。

以下是`Book`类的全部代码：

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, Float, Text, DateTime

# 创建对象的基类:
db = declarative_base()


class Book(db):
    __tablename__ = 'books'

    id = Column(Integer, primary_key=True)
    # 书名
    name = Column(String(255))
    # 作者
    authors = Column(String(50))
    # 出版社
    publishing_house = Column(String(255))
    # 出品方
    publisher = Column(String(255))
    # 原名
    origin_name = Column(String(255))
    # 译者
    translators = Column(String(255))
    # 出版时间
    pub_date = Column(DateTime)
    # 页数
    pages = Column(Integer)
    # 定价
    price = Column(Float)
    # ISBN
    isbn = Column(String(255))
    # 豆瓣评分
    rates = Column(Float)
    # 评价数
    rating_count = Column(Integer)
    summary = Column(Text)
    # 作者简介
    about_authors = Column(Text)
```

第二步：建立连接

接下来我们可以先创建一个测试，在测试中来实践演练数据库的连接与操控。在测试环境我们可以使用SQLite这个小型的文件数据库。

`test_db.py`的代码如下所示：

```
# coding:utf-8
from unittest import TestCase
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

class DataBaseTestCase(TestCase):

    def test_add_data(self):
        # 初始化数据库连接：
        engine = create_engine('sqlite:///test.db')
        # 创建DBSession类型：
        DBSession = sessionmaker(bind=engine)
        session = DBSession()
```

`create_engine()`用来初始化数据库连接。SQLAlchemy用一个字符串表示连接信息,这个就是连接字符串（Connection String）它的标准格式如下所示：

```
'数据库类型+数据库驱动名称://用户名:口令@机器地址:端口号/数据库名'
```

第三步：操作数据对象

下面，我们看看如何向数据库表中添加一行记录。

由于有了ORM，向数据库表中添加一行记录可以视为添加一个`Book`对象：

```
# 创建session对象：
DBSession = sessionmaker(bind=engine)
session = DBSession()

# 创建新Book对象：
book = Book(name="解忧杂货店",authors="东野圭吾")

# 添加到session中：
session.add(book)
```

第四步：提交更改

SQLAlchamy 的操作依赖于`session`对象，当实例化`session`后就可以调用`add`方法将对象加入到到`session`中，此时SQLAlchamy并没有直正将数据保存到数据库中，直至我们调用`session.commit()`后SQLAlchamy才会一次性将`session`内的数据变更提交到数据库。最后使用完数据库后就要调用`session.close()`方法关闭数据库连接释放连接资源。

```
# 提交即保存到数据库中：
session.commit()

# 关闭session：
session.close()
```

查询数据

如何从数据库表中查询数据呢？SQLAlchemy提供的查询接口如下：

```
# 创建Session:
session = DBSession()

# 创建Query查询，filter是where条件，最后调用one()返回唯一行，如果调用all()则返回所有行：
book = session.query(Book).filter(Book.id==1).one()

# 打印类型和对象的name属性：
print('type:', type(book))
print('name:', book.name)

# 关闭Session:
session.close()
```

由于关系数据库的多个表还可以用外键实现一对多、多对多等关联，相应地，ORM框架也可以提供两个对象之间的一对多、多对多等功能。由于本示例中并没有使用到关系表，所以在此不作过多的表述。但作为扩展学习，你可以到SQLAlchamy的官网上仔细阅读相关的内容。

安装数据库

安装 SQLite

Windows

先下载以下的安装文件:

- SQLite for Windows 32位运行库
- SQLite for Windows 64位运行库
- SQLite 工具

创建文件夹 `C:\\sqlite`，并在此文件夹下解压上面两个压缩文件，将得到 `sqlite3.def`、`sqlite3.dll` 和 `sqlite3.exe` 文件。

添加 `C:\\sqlite` 到 `PATH` 环境变量，最后在命令提示符下，使用 sqlite3 命令，将显示如下结果。

```
C:\>sqlite3
SQLite version 3.28.0 2019-03-02 15:25:24
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Linux

目前，几乎所有版本的 Linux 操作系统都附带 SQLite。所以，只要使用下面的命令来检查您的机器上是否已经安装了 SQLite。

```
$ sqlite3
SQLite version 3.28.0 2019-03-02 15:25:24
Enter ".help" for usage hints.
sqlite>
```

如果没有就直接下载 sqlite-tools-linux-x86-3280000.zip 到本地执行安装。

macOS

macOS上可以通过Home brew来安装，具体如下所示:

```
$ brew install sqlite3
```

安装 PostgreSQL

我们要以借助Docker避免安装数据库这样麻烦的过程，PostgreSQL的docker:

```
$ docker run -p 5432:5432 -e POSTGRES_PASSWORD=<你想设置的数据库密码> -d postgres
```

安装 MySQL

```
$ docker run -p 3306:3306 -e MYSQL_ROOT_PASSWORD=<你想设置的数据库密码> -d mysql
```

SQL数据项导出器与SQL存储端的实现

Scrapy的数据导出机制是由数据项导出器(`ItemExporter`)和存储端所组成（Storage backends)。`ItemExporter`负责数据格式的转换，存储端则负责对数据项进行存储。我们要实现基于SQL的数据导出机制就要同时提供以下两个类:

- `SQLItemExporter` - 将数据项转换为实体类，例如将`BookItem`转换为`Book`
- `SQLStorage` - 将数据实体写入到SQL数据库

`SQLFeedStorage`

Scrapy的存储端技术虽然有很强的可扩充性，但官方只给出了文件形式的存储。如果仔细阅读上一节的内容，会发现存储后端仅限于文件类型使用，不能用作数据库，这是由于存储端需要实现的`IFeedStorage`接口所限定的，我们先来看看`IFeedStorage`的定义：

```
class IFeedStorage(Interface):
    """所有的存储后端口都必须实现此接口"""

    def __init__(uri):
        """通过指定的URI参数初始化存储"""

    def open(spider):
        """为指定的蜘蛛打开存储。此方法必须返回一个类似文件的对象，用作数据导出"""

    def store(file):
        """存储文件流"""
```

所有的存储后端口都必须实现此接口，如果想扩充成数据库方式，则要对`store(file)`方法进行一点**活用**，我们先将整个数据导出机制的运作过程撸一撸：

1. 调用`IFeedStorage.open()`打开数据文件，`open`必须返回一个`file`对象
2. Scrapy 会将`file`对象在构造ItemExporter作为构造参传递至ItemExporter
3. ItemExporter将数据项一条一条进行导出转换，然后后写入`file`对象。
4. 完成所有数据项的导出后，Scrapy会调用`IFeedStorage`的`store(file)`保存数据。

在`open`方法中返回的`file`与`store`中的`file`参数并没有规定具体的类型，所以我们可以设计一个类模拟`file`的写入行为，而实际上却是用于将数据实体写入到`session`

```
class EntityFile():
    def __init__(self, session, entity_cls):
      """
      构造函数
      :param session: 数据库Session实例
      :param entity_cls: 数据实体类
      """
        self.session = session
        if entity_cls is None:
            raise NotConfigured
        self.entity_cls = entity_cls

    def write(self, keys, values):
        """
        将值写入到
        :param key: 实体的成员变量
        :param value: 实体字段值
        """
        # 实例化数据实体
        entity = self.entity_cls()
        # 根据keys中声明的实体成员变量名称向实体实例赋值
        for key in keys:
            val = values.get(key)
            if val is not None:
                entity.__setattr__(key, val)
        self.session.add(entity)

    def close(self):
        self.session.commit()
        self.session.close()
```

如果频繁地向PostgreSQL提交变更（`commit()`），数据库的写入速度就会变得越来越慢，这是由于PostgreSQL在提交时会写入数据日志（很多数据库都有此功能）。又因为蜘蛛可能一次性爬取的数据量会非常庞大，解决这一问题的最佳方法是在蜘蛛开始爬网时先连接数据并产生数据库的会话上下文，每爬取一项数据就保存当前会话上下文的数据，当然这种保存是写入到会话缓存中的。当蜘蛛完成爬网后，利用PostgreSQL的批量写入功能一次性地将大量的数据变更存入数据库（批量写入是一种效率极高、数据库I/O消耗又很低的操作）并断开与数据库的连接以释放资源。所以我们在`close`方法中才对数据库执行一次全面提交更改的动作，以实现批量更新。

这个`EntityFile`是非常通用的，因为它并没有与数据模型产生任何的依赖，只有被实例化时才会具体指定的需要处理的实体类的类型。

接下来就是设计`SQLFeedStorage`了，为了让这个存储端可以被重用，我们就需要将所有的依赖信息外置，例如：

- 数据模型的类型信息
- 连接字符串

信息外置最好的办法当然就是放在`settings.py`文件内，然后定义一个`from_crawler`类方法从配置文件中将它们读入。具体代码如下所示：

```
from zope.interface import Interface, implementer
from scrapy.extensions.feedexport import IFeedStorage

@implementer(IFeedStorage)
class SQLFeedStorage():
    """
    SQL存储端
    @uri - SQL的连接字符串
    """
    @classmethod
    def from_crawler(cls, crawler, uri):
        return cls(uri,
                   crawler.settings.get('ORM_MODULE'),
                  crawler.settings.get('ORM_METABASE'),
                  crawler.settings.get('ORM_ENTITY'))

    def __init__(self, uri, mod_name=None, metabase_name=None, entity_name=None):
        """
        初始化SQL的存储后端
        FEED_URI 作为连接字符串使用
        """
        self.connection_str = uri
        self.mod_name = mod_name
        self.metabase = metabase_name
        self.entity_name = entity_name
```

接下来就是实现`open`方法,在这个方法中我们要实现连接到数据库、数据库的初始化(建表)、实例化`session` 这一系列的动作:

```
def open(self, spider):
  """
  通过连接字符串打开SQL数据库并返回生成的数据库上下文
  """
  engine = create_engine(self.connection_str)

  # 动态载入MetaData
  mod = import_module(self.mod_name)
  metabase = getattr(mod, self.metabase)
  entity_cls = getattr(mod, self.entity_name)

  # 初始化数据库
  metabase.metadata.bind = engine
  metabase.metadata.create_all()

  # 产生Session实例并返回EntityFile对象
  DBSession = sessionmaker(bind=engine)
  return EntityFile(session=DBSession(), entity_cls=entity_cls)
```

上述代码中应用了python编程的小技巧：动态加载块。

```
# 从指定的模块字符串中导入类相当于import的作用
mod = import_module(self.mod_name)
# 从模块中加载具体的metabase类与数据实体类
metabase = getattr(mod, self.metabase)
entity_cls = getattr(mod, self.entity_name)
```

最后要在`close`方法中关闭数据连接

```
def store(self, file):
  """向数据提提交更改并关闭数据库"""
  file.close()
```

以下是`SQLFeedStorage`的完整代码:

```
@implementer(IFeedStorage)
class SQLFeedStorage():
    """
    SQL的存储后端
    @uri - SQL的连接字符串
    """

    @classmethod
    def from_crawler(cls, crawler, uri):
        return cls(uri,
                   crawler.settings.get('ORM_MODULE'),
                   crawler.settings.get('ORM_METABASE'),
                   crawler.settings.get('ORM_ENTITY'))

    def __init__(self, uri, mod_name=None, metabase_name=None, entity_name=None):
        """
        初始化SQL的存储后端
        FEED_URI 作为连接字符串使用
        """
        self.connection_str = uri
        self.mod_name = mod_name
        self.metabase = metabase_name
        self.entity_name = entity_name

    def open(self, spider):
        """
        通过连接字符串打开SQL数据库并返回生成的数据库上下文
        """
        engine = create_engine(self.connection_str)

        # 动态载入MetaData
        mod = import_module(self.mod_name)
        metabase = getattr(mod, self.metabase)
        entity_cls = getattr(mod, self.entity_name)
        metabase.metadata.bind = engine
        metabase.metadata.create_all()

        DBSession = sessionmaker(bind=engine)
        return EntityFile(session=DBSession(), entity_cls=entity_cls)

    def store(self, file):
        """
        向数据提提交更改并关闭数据库
        """
        file.close()
```

SQLItemExporter

Scrapy 的数据项导出器都继承自`BaseItemExporter`，继承这个类必须重写`export_item`方法，代码如下：

```
class SQLItemExporter(BaseItemExporter):
    """
    将Item中的数据写入转换成为实体
    """
    def __init__(self, file, **kwargs):
        self.file = file
        self._configure(kwargs, dont_fail=True)

    def export_item(self, item):
        """
        将Item插入到数据库
        可以通过FEED_EXPORT_FIELDS设置要从Item中序列化至数据库的字段
        """
        self.file.write(self.fields_to_export if self.fields_to_export is not None and self.fields_to_export.__len__() else
 item.fields.keys(),
                        item)
```

`fields_to_export`是继承自`BaseItemExporter`的，是在构造函中被`_configure`产生的，我们可以在配置文件中用`FEED_EXPORT_FIELDS`配置项来控制要将数据项中的哪些字段映射到实体对象上。

配置项

最后我们要将自定义的`SQLItemExporter`和`SQLFeedStorage`加入到Scrapy的数据导出机制内。

首先是新增的配置项：

```
ORM_MODULE = 'douban.modals' # 数据模型所在的模块
ORM_METABASE = 'db'       # 数据元定义类
ORM_ENTITY = 'Book'       # 数据实体类
```

然后是对数据导出机制的相关配置项的修改：

```
FEED_FORMAT = 'entity'      # 数据导出的格式
FEED_EXPORTERS = {          # 关联导出格式与数据项导出器
    'entity': 'douban.extensions.SQLItemExporter'
}

FEED_URI = 'sqlite:///test.db'  # 数据库的连接字符串

# 增加对sqlite，postgresql和mysql内种协议的存储端支持
FEED_STORAGES = {
    'sqlite': 'douban.extensions.SQLFeedStorage',
    'postgresql': 'douban.extensions.SQLFeedStorage',
    'mysql': 'douban.extensions.SQLFeedStorage'
}
```

> 注：如果要连接PostgreSQL的话需要在python中安装psycopg2。如果psycopg2不能安装要以安装psycopg2-binary，指令如下：
>
> ```
> pip install psycopg2-binary
> ```

小结

本节所介绍的数据导出机制是Scrapy中的一个比较大的技术主题，基于这个机制来实现SQL存储不知道你是不是会觉得非常复杂？确实，你还可以用另一个办法来实现SQL存储：管道(Pipline)，接下来试试自动手写一个SQLPipline来实现相同的功能如何？这将会是一个化繁为简的过程，而且到了本小节你应该可以具有一些基本的代码设计能力，对Scrapy也有一定的认知了，是时候脱离教程自己设计一下代码了。

19 测试BookSpider

21 反爬之客户端仿真