上节课我们学习了如何用 pyquery 提取 HTML 中的信息，但是当我们成功提取了数据之后，该往哪里存放呢？

用文本文件当然是可以的，但文本存储不方便检索。有没有既方便存，又方便检索的存储方式呢？

当然有，本课时我将为你介绍一个文档型数据库 —— MongoDB。

MongoDB 是由 C++ 语言编写的非关系型数据库，是一个基于分布式文件存储的开源数据库系统，其内容存储形式类似 JSON 对象，它的字段值可以包含其他文档、数组及文档数组，非常灵活。

在这个课时中，我们就来看看 Python 3 下 MongoDB 的存储操作。

### 准备工作

在开始之前，请确保你已经安装好了 MongoDB 并启动了其服务，同时安装好了 Python 的 PyMongo 库。

MongoDB 的安装方式可以参考：https://cuiqingcai.com/5205.html，安装好之后，我们需要把 MongoDB 服务启动起来。

注意：这里我们为了学习，仅使用 MongoDB 最基本的单机版，MongoDB 还有主从复制、副本集、分片集群等集群架构，可用性可靠性更好，如有需要可以自行搭建相应的集群进行使用。

启动完成之后，它会默认在本地 localhost 的 27017 端口上运行。

接下来我们需要安装 PyMongo 这个库，它是 Python 用来操作 MongoDB 的第三方库，直接用 pip3 安装即可：`pip3 install pymongo`。

更详细的安装方式可以参考：https://cuiqingcai.com/5230.html。

安装完成之后，我们就可以使用 PyMongo 来将数据存储到 MongoDB 了。

### 连接 MongoDB

连接 MongoDB 时，我们需要使用 PyMongo 库里面的 MongoClient。一般来说，我们只需要向其传入 MongoDB 的 IP 及端口即可，其中第一个参数为地址 host，第二个参数为端口 port（如果不给它传递参数，则默认是 27017）：

```
import pymongo
client = pymongo.MongoClient(host='localhost', port=27017)
```

这样我们就可以创建 MongoDB 的连接对象了。

另外，MongoClient 的第一个参数 host 还可以直接传入 MongoDB 的连接字符串，它以 mongodb 开头，例如：

```
client = MongoClient('mongodb://localhost:27017/')
```

这样也可以达到同样的连接效果。

### 指定数据库

MongoDB 中可以建立多个数据库，接下来我们需要指定操作其中一个数据库。这里我们以 test 数据库作为下一步需要在程序中指定使用的例子：

```
db = client.test
```

这里调用 client 的 test 属性即可返回 test 数据库。当然，我们也可以这样指定：

```
db = client['test']
```

这两种方式是等价的。

### 指定集合

MongoDB 的每个数据库又包含许多集合（collection），它们类似于关系型数据库中的表。

下一步需要指定要操作的集合，这里我们指定一个名称为 students 的集合。与指定数据库类似，指定集合也有两种方式：

```
collection = db.students
```

或是

```
collection = db['students']
```

这样我们便声明了一个 Collection 对象。

### 插入数据

接下来，便可以插入数据了。我们对 students 这个集合新建一条学生数据，这条数据以字典形式表示：

```
student = {
    'id': '20170101',
    'name': 'Jordan',
    'age': 20,
    'gender': 'male'
}
```

新建的这条数据里指定了学生的学号、姓名、年龄和性别。接下来，我们直接调用 collection 的 insert 方法即可插入数据，代码如下：

```
result = collection.insert(student)
print(result)
```

在 MongoDB 中，每条数据其实都有一个 _id 属性来唯一标识。如果没有显式指明该属性，MongoDB 会自动产生一个 ObjectId 类型的 _id 属性。insert() 方法会在执行后返回 _id 值。

运行结果如下：

```
5932a68615c2606814c91f3d
```

当然，我们也可以同时插入多条数据，只需要以列表形式传递即可，示例如下：

```
student1 = {
    'id': '20170101',
    'name': 'Jordan',
    'age': 20,
    'gender': 'male'
}

student2 = {
    'id': '20170202',
    'name': 'Mike',
    'age': 21,
    'gender': 'male'
}

result = collection.insert([student1, student2])
print(result)
```

返回结果是对应的 _id 的集合：

```
[ObjectId('5932a80115c2606a59e8a048'), ObjectId('5932a80115c2606a59e8a049')]
```

实际上，在 PyMongo 中，官方已经不推荐使用 insert 方法了。但是如果你要继续使用也没有什么问题。目前，官方推荐使用 insert_one 和 insert_many 方法来分别插入单条记录和多条记录，示例如下：

```
student = {
    'id': '20170101',
    'name': 'Jordan',
    'age': 20,
    'gender': 'male'
}

result = collection.insert_one(student)
print(result)
print(result.inserted_id)
```

运行结果如下：

```
<pymongo.results.InsertOneResult object at 0x10d68b558>
5932ab0f15c2606f0c1cf6c5
```

与 insert 方法不同，这次返回的是 InsertOneResult 对象，我们可以调用其 inserted_id 属性获取 _id。

对于 insert_many 方法，我们可以将数据以列表形式传递，示例如下：

```
student1 = {
    'id': '20170101',
    'name': 'Jordan',
    'age': 20,
    'gender': 'male'
}

student2 = {
    'id': '20170202',
    'name': 'Mike',
    'age': 21,
    'gender': 'male'
}

result = collection.insert_many([student1, student2])
print(result)
print(result.inserted_ids)
```

运行结果如下：

```
<pymongo.results.InsertManyResult object at 0x101dea558>
[ObjectId('5932abf415c2607083d3b2ac'), ObjectId('5932abf415c2607083d3b2ad')]
```

该方法返回的类型是 InsertManyResult，调用 inserted_ids 属性可以获取插入数据的 _id 列表。

### 查询

插入数据后，我们可以利用 find_one 或 find 方法进行查询，其中 find_one 查询得到的是单个结果，find 则返回一个生成器对象。示例如下：

```
result = collection.find_one({'name': 'Mike'})
print(type(result))
print(result)
```

这里我们查询 name 为 Mike 的数据，它的返回结果是字典类型，运行结果如下：

```
<class 'dict'>
{'_id': ObjectId('5932a80115c2606a59e8a049'), 'id': '20170202', 'name': 'Mike', 'age': 21, 'gender': 'male'}
```

可以发现，它多了 _id 属性，这就是 MongoDB 在插入过程中自动添加的。

此外，我们也可以根据 ObjectId 来查询，此时需要调用 bson 库里面的 objectid：

```
from bson.objectid import ObjectId

result = collection.find_one({'_id': ObjectId('593278c115c2602667ec6bae')})
print(result)
```

其查询结果依然是字典类型，具体如下：

```
{'_id': ObjectId('593278c115c2602667ec6bae'), 'id': '20170101', 'name': 'Jordan', 'age': 20, 'gender': 'male'}
```

如果查询结果不存在，则会返回 None。

对于多条数据的查询，我们可以使用 find 方法。例如，这里查找年龄为 20 的数据，示例如下：

```
results = collection.find({'age': 20})
print(results)
for result in results:
    print(result)
```

运行结果如下：

```
<pymongo.cursor.Cursor object at 0x1032d5128>
{'_id': ObjectId('593278c115c2602667ec6bae'), 'id': '20170101', 'name': 'Jordan', 'age': 20, 'gender': 'male'}
{'_id': ObjectId('593278c815c2602678bb2b8d'), 'id': '20170102', 'name': 'Kevin', 'age': 20, 'gender': 'male'}
{'_id': ObjectId('593278d815c260269d7645a8'), 'id': '20170103', 'name': 'Harden', 'age': 20, 'gender': 'male'}
```

返回结果是 Cursor 类型，它相当于一个生成器，我们需要遍历获取的所有结果，其中每个结果都是字典类型。

如果要查询年龄大于 20 的数据，则写法如下：

```
results = collection.find({'age': {'$gt': 20}})
```

这里查询的条件键值已经不是单纯的数字了，而是一个字典，其键名为比较符号 $gt，意思是大于，键值为 20。

我将比较符号归纳为下表：

| 符　　号 | 含　　义 | 示　　例 |
| --- | --- | --- |
| $lt | 小于 | {'age': {'$lt': 20}} |
| $gt | 大于 | {'age': {'$gt': 20}} |
| $lte | 小于或等于 | {'age': {'$lte': 20}} |
| $gte | 大于或等于 | {'age': {'$gte': 20}} |
| $ne | 不等于 | {'age': {'$ne': 20}} |
| $in | 在范围内 | {'age': {'$in': [20, 23]}} |
| $nin | 不在范围内 | {'age': {'$nin': [20, 23]}} |

另外，还可以进行正则匹配查询。例如，查询名字以 M 开头的学生数据，示例如下：

```
results = collection.find({'name': {'$regex': '^M.*'}})
```

这里使用 $regex 来指定正则匹配，^M.* 代表以 M 开头的正则表达式。

我将一些功能符号归类为下表：

| 符号 | 含义 | 示例 | |
| --- | --- | --- | --- |
| $regex | 匹配正则表达式 | {'name': {'$regex': '^M.*'}} | na |
| $exists | 属性是否存在 | {'name': {'$exists': True}} | na |
| $type | 类型判断 | {'age': {'$type': 'int'}} | ag |
| $mod | 数字模操作 | {'age': {'$mod': [5, 0]}} | 年 |
| $text | 文本查询 | {'$text': {'$search': 'Mike'}} | text 类 |
| $where | 高级条件查询 | {'$where': 'obj.fans_count == obj.follows_count'} | 自身粉 |

关于这些操作的更详细用法，可以在 MongoDB 官方文档找到： https://docs.mongodb.com/manual/reference/operator/query/。

**计数**

要统计查询结果有多少条数据，可以调用 count 方法。我们以统计所有数据条数为例：

```
count = collection.find().count()
print(count)
```

我们还可以统计符合某个条件的数据：

```
count = collection.find({'age': 20}).count()
print(count)
```

运行结果是一个数值，即符合条件的数据条数。

**排序**

排序时，我们可以直接调用 sort 方法，并在其中传入排序的字段及升降序标志。示例如下：

```
results = collection.find().sort('name', pymongo.ASCENDING)
print([result['name'] for result in results])
```

运行结果如下：

```
['Harden', 'Jordan', 'Kevin', 'Mark', 'Mike']
```

这里我们调用 pymongo.ASCENDING 指定升序。如果要降序排列，可以传入 pymongo.DESCENDING。

**偏移**

在某些情况下，我们可能只需要取某几个元素，这时可以利用 skip 方法偏移几个位置，比如偏移 2，就代表忽略前两个元素，得到第 3 个及以后的元素：

```
results = collection.find().sort('name', pymongo.ASCENDING).skip(2)
print([result['name'] for result in results])
```

运行结果如下：

```
['Kevin', 'Mark', 'Mike']
```

另外，我们还可以用 limit 方法指定要取的结果个数，示例如下：

```
results = collection.find().sort('name', pymongo.ASCENDING).skip(2).limit(2)
print([result['name'] for result in results])
```

运行结果如下：

```
['Kevin', 'Mark']
```

如果不使用 limit 方法，原本会返回 3 个结果，加了限制后，就会截取两个结果返回。

值得注意的是，在数据量非常庞大的时候，比如在查询千万、亿级别的数据库时，最好不要使用大的偏移量，因为这样很可能导致内存溢出。此时可以使用类似如下操作来查询：

```
from bson.objectid import ObjectId
collection.find({'_id': {'$gt': ObjectId('593278c815c2602678bb2b8d')}})
```

这时需要记录好上次查询的 _id。

### 更新

对于数据更新，我们可以使用 update 方法，指定更新的条件和更新后的数据即可。例如：

```
condition = {'name': 'Kevin'}
student = collection.find_one(condition)
student['age'] = 25
result = collection.update(condition, student)
print(result)
```

这里我们要更新 name 为 Kevin 的数据的年龄：首先指定查询条件，然后将数据查询出来，修改年龄后调用 update 方法将原条件和修改后的数据传入。

运行结果如下：

```
{'ok': 1, 'nModified': 1, 'n': 1, 'updatedExisting': True}
```

返回结果是字典形式，ok 代表执行成功，nModified 代表影响的数据条数。

另外，我们也可以使用 $set 操作符对数据进行更新，代码如下：

```
result = collection.update(condition, {'$set': student})
```

这样可以只更新 student 字典内存在的字段。如果原先还有其他字段，则不会更新，也不会删除。而如果不用 $set 的话，则会把之前的数据全部用 student 字典替换；如果原本存在其他字段，则会被删除。

另外，update 方法其实也是官方不推荐使用的方法。这里也分为 update_one 方法和 update_many 方法，用法更加严格，它们的第 2 个参数需要使用 $ 类型操作符作为字典的键名，示例如下：

```
condition = {'name': 'Kevin'}
student = collection.find_one(condition)
student['age'] = 26
result = collection.update_one(condition, {'$set': student})
print(result)
print(result.matched_count, result.modified_count)
```

上面的例子中调用了 update_one 方法，使得第 2 个参数不能再直接传入修改后的字典，而是需要使用 {'$set': student} 这样的形式，其返回结果是 UpdateResult 类型。然后分别调用 matched_count 和 modified_count 属性，可以获得匹配的数据条数和影响的数据条数。

运行结果如下：

```
<pymongo.results.UpdateResult object at 0x10d17b678>
1 0
```

我们再看一个例子：

```
condition = {'age': {'$gt': 20}}
result = collection.update_one(condition, {'$inc': {'age': 1}})
print(result)
print(result.matched_count, result.modified_count)
```

这里指定查询条件为年龄大于 20，然后更新条件为 {'$inc': {'age': 1}}，表示年龄加 1，执行之后会将第一条符合条件的数据年龄加 1。

运行结果如下：

```
<pymongo.results.UpdateResult object at 0x10b8874c8>
1 1
```

可以看到匹配条数为 1 条，影响条数也为 1 条。

如果调用 update_many 方法，则会将所有符合条件的数据都更新，示例如下：

```
condition = {'age': {'$gt': 20}}
result = collection.update_many(condition, {'$inc': {'age': 1}})
print(result)
print(result.matched_count, result.modified_count)
```

这时匹配条数就不再为 1 条了，运行结果如下：

```
<pymongo.results.UpdateResult object at 0x10c6384c8>
3 3
```

可以看到，这时所有匹配到的数据都会被更新。

### 删除

删除操作比较简单，直接调用 remove 方法指定删除的条件即可，此时符合条件的所有数据均会被删除。

示例如下：

```
result = collection.remove({'name': 'Kevin'})
print(result)
```

运行结果如下：

```
{'ok': 1, 'n': 1}
```

另外，这里依然存在两个新的推荐方法 —— delete_one 和 delete_many，示例如下：

```
result = collection.delete_one({'name': 'Kevin'})
```

```
print(result)
print(result.deleted_count)
result = collection.delete_many({'age': {'$lt': 25}})
print(result.deleted_count)
```

运行结果如下：

```
<pymongo.results.DeleteResult object at 0x10e6ba4c8>
1
4
```

delete_one 即删除第一条符合条件的数据，delete_many 即删除所有符合条件的数据。它们的返回结果都是 DeleteResult 类型，可以调用 deleted_count 属性获取删除的数据条数。

### 其他操作

另外，PyMongo 还提供了一些组合方法，如 find_one_and_delete、find_one_and_replace 和 find_one_and_update，它们分别用于查找后删除、替换和更新操作，其使用方法与上述方法基本一致。

另外，我们还可以对索引进行操作，相关方法有 create_index、create_indexes 和 drop_index 等。

关于 PyMongo 的详细用法，可以参见官方文档：http://api.mongodb.com/python/current/api/pymongo/collection.html。

另外，还有对数据库和集合本身等的一些操作，这里不再一一讲解，可以参见官方文档：http://api.mongodb.com/python/current/api/pymongo/。

本课时的内容我们就讲到这里了，你是不是对如何使用 PyMongo 操作 MongoDB 进行数据增删改查更加熟悉了呢？下一课时我将会带你在实战案例中应用这些操作进行数据存储。