第一章 SQLAlchemy基本认知

`20:30上课！！！`

1.基本认知

2.工具准备

3.连接数据库

4.声明映像

5.创建和映射类

6.数据操作

1.基本认知

ORM全称 `Object Relational Mapping` 对象关系映射。

SQLAlchemy 是一个数据库的 `ORM` 框架， `python` 操作数据库的工具 `ORM` 将数据库中的表与面向对象语言中的类建立了一种对应关系。简单来说，使用 `SQLAlchemy` 可以不用考虑使用的是什么数据库，只要是用 `SQLAlchemy` 提供的方式写好语句， `SQLAlchemy` 会自动根据你连接的数据库把你写的语句转化成对应的数据库 `SQL` 语句。

2.工具准备

1. `mysql` 数据库

2. `pymysql` 用于连接 `MySQL` 服务器的一个库

3. `sqlalchemy`

```
$ pip install pymysql
$ pip install sqlalchemy
```

3.连接数据库

从sqlalchemy中导入create_engin，创建引擎建立与数据库的连接。

```
from sqlalchemy import create_engine
```

准备连接数据库的数据：
HOSTNAME = '127.0.0.1' # ip地址
PORT = '3306' # 端口号
DATABASE = 'mydb' # 数据库名
USERNAME = 'admin' # 用户名
PASSWORD = 'rootqwe123' # 用户登录密码

DB_URI的格式:
数据库类型+数据库驱动名称://用户名:密码@机器地址:端口号/数据库名?字符编码
DB_URI=`mysql+pymysql://<username>:<password>@<host>/<dbname>?charset=utf8`

engine = create_engine(DB_URI)

我们可以尝试着测试一下是否连接上:

`print(dir(engine))`,当有打印出方法时，表示连接成功。

```
#connect.py
from sqlalchemy import create_engine

HOSTNAME = '127.0.0.1'
PORT = '3306'
DATABASE = 'mydb'
USERNAME = 'admin'
PASSWORD = 'Root110qwe'

Db_Uri = 'mysql+pymysql://{}:{}@{}/{}?charset=utf8'.format(USERNAME,PASSWORD,HOSTNAME,DATABASE)

engine = create_engine(Db_Uri)

if __name__=='__main__':
    print(dir(engine))
```

4.声明映像

对象关系型映射，数据库中的表与python中的类相对应，创建的类必须继承自sqlalchemy中的基类。

使用Declarative方法定义的映射类依据一个基类，这个基类是维系类和数据表关系的目录。
应用通常只需要有一个base的实例。我们通过declarative_base()功能创建一个基类。

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base(engine)
```

5.创建和映射类

再次强调，我们用类来表示数据库里面的表！！！

这些表的类都继承于我们的Base基类。在类里面我们定义一些属性，这个属性通过映射，就对应表里面的字段，每个属性定义的时候需要指定属性是属于那
定义好一些属性，与user表中的字段进行映射并且这个属性要属于某个类型

Column用来创建表中的字段的一个方法

```
#sqlalchemy常用的数据类型
Integer #整形，映射到数据库中的int类型。
String  #字符类型，映射到数据库中的varchar类型，使用时，需要提供一个字符长度。
```

```
#---创建数据表对应的类---
from sqlalchemy import Column,Integer,String

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer,primary_key=True,autoincrement=True,doc='id')
    name = Column(String(20),nullable=False,doc='用户姓名')
    number = Column(Integer,nullable=False,doc='用户号')

    def __repr__(self):
        return "<User(id='%s',name='%s',number='%s')"%(self.id,self.name,self.number)

#---将创建好的user类，映射到数据库的user表中---
Base.metadata.create_all()
```

6.数据操作

之前的操作都是准备工作，提供一个类似与交互模式的环境，让我们能够进行增删改查的操作。

在我们对表数据进行增删改查之前，先需要建立会话，建立会话之后才能进行操作，就类似于文件要打开之后才能对文件内容操作。

创建会话

定义个`session`会话对象,使用`sessionmaker`初始化一个类对象

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(engine)
session = Session()
```

增

我们通过sqlalchemy往表里面插入数据，需要先有这表的映射，也就是上面我们创建的类，然后我们通过会话来进行需要的操作。

创建`user`对象添加到会话对象中

添加单个对象：
`session.add(user)`
添加多个对象：
`session.add_all([user1,user2,....])`
将会话对象进行提交:
`session.commit()`
如果你不想将修改提交则使用回滚:`session.rollback()`

```
def add_user():
    #添加单个对象
    #person = User(name='budong',number=11)
    #session.add(person)

    #添加多个对象
    session.add_all([User(name='tuple',number=2),\
            User(name='which',number=3)])
    #提交才会生效，和命令行有区别
    session.commit()

add_user()
```

查

`session`会话中的`query`对象可以查找数据。

查找`User`这张表中的所有数据:`session.query(User).all()`
查找`User`这张表中的第一条数据:`session.query(User).first()`
通过`username=?`来进行过滤查找:`session.query(User).filter_by(username='budong')`
通过`get`方法，用主键查找对象:`session.query(User).get(primary_key)`

注意：
`filter`引用列名时，使用`类名.属性名`的方式，比较使用两个等号`==`
`filter_by`引用列名时，使用`属性名`，比较使用一个等号`=`赋值这种形式

这两个只是用法不一样而已，常用`filter`

```
def search_user():
    #查询所有
    rows = session.query(User).all()
    print(rows)

    #查询第一条
    rows = session.query(User).first()
    print(rows)

    #查询number=11的数据
    rows = session.query(User).filter_by(number=11).all()
    print(rows)

    #查询name='budong'的数据
    rows = session.query(User).filter(User.name=='budong').all()
    print(rows)
```

改

修改数据是在`filter`里面过滤需要修改的数据，再使用`update`即可。

```
def update_user():
    rows = session.query(User).filter(User.name=='budong').update({User.number:1})
    session.commit()
```

删

我们可以先查出数据，然后再删除，使用`session.delete`方法进行删除.

```
def delete_user():
    rows = session.query(User).filter(User.name=='which')[0]
    print(rows)
    session.delete(rows)
    session.commit()
```

总结

SQLAlchemy是一个ORM，便于我们的代码从一个数据库迁移到另一个数据库，提高代码的复用率。

用SQLAlchemy连接数据库需要大家掌握怎样连接数据库，然后怎样创建会话，创建会话之后怎样才可以进行增删改查的操作，这就是这节课要掌握的内容。

补充vim自动缩进

```
#~/下执行
$ sudo yum -y install git
$ git clone https://github.com/gmarik/Vundle.vim.git ~/.vim/bundle/Vundle.vim
$ touch ~/.vimrc
####写到 ~/.vimrc
set nocompatible              " required
filetype off                  " required

" set the runtime path to include Vundle and initialize
set rtp+=~/.vim/bundle/Vundle.vim
call vundle#begin()
"
" " alternatively, pass a path where Vundle should install plugins
" "call vundle#begin('~/some/path/here')
"
" " let Vundle manage Vundle, required
Plugin 'gmarik/Vundle.vim'
"
" " Add all your plugins here (note older versions of Vundle used Bundle
" instead of Plugin)
"
"
" " All of your Plugins must be added before the following line
call vundle#end()            " required
filetype plugin indent on    " required

"代码折叠
" Enable folding
set foldmethod=indent
set foldlevel=99
" Enable folding with the spacebar
nnoremap <space> za
Plugin 'tmhedberg/SimpylFold'
let g:SimpylFold_docstring_preview=1

"au BufNewFile,BufRead *.py
set tabstop=4
set softtabstop=4
set shiftwidth=4
set textwidth=79
set expandtab
set autoindent
set fileformat=unix


####最后在vim里面执行    :PluginInstall
```

第二章 SQLAlchemy查询

`20:30上课！！！`

1.查询结果

2.过滤条件

3.Column常用参数

4.sqlalchemy常用数据类型

1.查询结果

我们使用`session`中的`query`可以查出数据，但是我们对返回的结果还不太熟悉，我们有必要了解一下返回的结果，这样才能方便我们取数。

```
#返回一个对象，就好像函数没有加括号一样
    rs = session.query(User).filter(User.name=='budong')
    print('***',rs,type(rs))

#这里加上0和加上first()是一样的，返回一个User对象，属性值就是数据
    rs = session.query(User).filter(User.name=='budong')[0]
    print('aaa',rs,type(rs))
    rs = session.query(User).filter(User.name=='budong').first()
    print('aaabbb',rs,type(rs))

#和上面的返回的对象一样，但是值放在列表里面，对于对象的值，我们可以使用getattr方法来获取
    rs = session.query(User).filter(User.name=='budong').all()
    print('bbb',rs,type(rs))
    print('++++++',getattr(rs[0],'name'))

#和最开始类似
    rs = session.query(User.number).filter(User.name=='budong')
    print('ccc',rs,type(rs))

#这里的[0]也就是first()，但是结果不是一个对象，是一个元组，元组的值就是查询出的数据
    rs = session.query(User.number).filter(User.name=='budong')[0]
    print('ddd',rs,type(rs))

#把上面的结果放到列表里面，同时是查询所有符合条件的结果
    rs = session.query(User.number).filter(User.name=='budong').all()
    print('eee',rs,type(rs))
```

2.过滤条件

`filter`和`filter_by`都可以过滤，结果都一样，只是用法有点不一样：

`filter`引用列名时，使用“类名.属性名”的方式，比较使用两个等号`==`

`filter_by`引用列名时，使用“属性名”，比较使用一个等号`=`

`filter`就相当与我们`MySQL`里面的`where`，很多`where`里面的判断条件也是可以用的，具体写法参考于下：

```
#等于
rows = session.query(User.number).filter(User.name=='budong').all()

#不等于
rows = session.query(User.number).filter(User.name!='budong').all()
#filter_by里面不能用!= 还有> < 等等，所有filter用得更多,filter_by只能用=

#模糊匹配like
rows = session.query(User.number).filter(User.name.like('budong%')).all()

#成员属于 in_
rows = session.query(User.number).filter(User.name.in_(['budong','tuple'])).all()

#成员不属于 notin_
rows = session.query(User.number).filter(User.name.notin_(['budong'])).all()

#为空
rows = session.query(User.number).filter(User.name==None).all()
#或者
rows = session.query(User.number).filter(User.name.is_(None)).all()

#不为空
rows = session.query(User.number).filter(User.name!=None).all()
#或者
rows = session.query(User.number).filter(User.name.isnot(None)).all()

#多个条件
rows = session.query(User.number).filter(User.name.isnot(None),User.number!=0).all()

#选择条件or_
from sqlalchemy import or_
rows = session.query(User.number).filter(or_(User.name=='budong',User.number==2)).all()
```

3.Column常用参数

`Column`声明类属性时使用，也即数据库表里面的字段，常用的参数如下：

1. primary_key：主键，True和False。
2. autoincrement：是否自动增长，True和False。
3. unique：是否唯一。
4. nullable：是否可空，默认是True。
5. default：默认值。

4.SQLAlchemy常用数据类型

sqlalchemy中也有很多的数据类型，这些数据类型和我们数据库里面的各种数据类型相对应。常用的如下：

1. Integer：整型，映射到数据库中的int类型。
2. String：字符类型，映射到数据库中的varchar类型，使用时，需要提供一个字符长度。
3. Text：文本类型，映射到数据库中的text类型。
4. Boolean：布尔类型，映射到数据库中的tinyint类型，在使用的时候，传递True/False进去。
5. Date：日期类型，没有时间。映射到数据库中是date类型，在使用的时候，传递datetime.date()进去。

6. DateTime：日期时间类型。映射到数据库中的是datetime类型，在使用的时候，传递datetime.datetime()进去。
7. Float：浮点类型。

```
from user import session,Base
from datetime import datetime,date
from sqlalchemy import Column,Integer,String,Boolean,Date,DateTime

class Column_test(Base):
    __tablename__ = 'column_test'
    id = Column(Integer,primary_key=True,autoincrement=True)
    name = Column(String(20),nullable=False,unique=True)
    gender = Column(Boolean)
    start_time = Column(DateTime,default=datetime.now())
    birth_time = Column(Date)

Base.metadata.create_all()

haha = Column_test(name='haha',gender=True,birth_time=date(2017,8,22))
session.add(haha)
session.commit()
```

总结

在`SQLAlchemy`中的查询和我们写原生的`SQL`是类似的，我们理解了在`MySQL`中如何书写，在`SQLAlchemy`中思维是一样的，只不过表达方式有些区别，大家学会使用就行。

第三章 sqlalchemy查询使用

1.带条件查询

2.表关系查询

3.多表查询

4.原生**SQL**的查询以及其他使用

1.带条件的查询

查询是最常用的，对于各种查询我们必须要十分清楚，首先是带条件的查询

```
#带条件查询
rows = session.query(User).filter_by(username='budong').all()
print(rows)
rows1 = session.query(User).filter(User.username=='budong').all()
print(rows1)
rows2 = session.query(User.username).filter(User.username=='budong').all()
print(rows2)
rows3 = session.query(User.username).filter(User.username=='budong')
print(rows3)
```

`filter_by`和`filter`都是过滤条件，只是用法有区别`filter_by`里面不能用`!=`还有`> <` 等等，所有`filter`用得更多,`filter_by`只能用`=`。

前两个查询的是`User`,所以返回结果也是一个对象，但是`rows2`查询的是属性值，所以返回的是属性值。

`rows3`可以看到`SQLAlchemy`转成的`SQL`语句，`SQLAlchemy`最后都是会转成`SQL`语句，通过这个方法可以查看原生`SQL`,甚至有些时候我们需要把`SQLAlchemy`转成的`SQL`交给DBA审查，合适的才能使用。

查询要知道查询结果的返回怎样的数据

```
#基本查询
print( session.query(User).filter(User.username=='budong').all() )
print( session.query(User).filter(User.username=='budong').first())
print( session.query(User).filter(User.username=='budong').one())
print( session.query(User).get(2))
```

上面三条记录，第一个查出所有符合条件的记录，第二个查出所有符合记录的第一条记录，第三个返回一个对象，如果结果有多条就会报错，第四个通过主键获取记录

除此之外，我们偶尔也会需要限制返回的结果数量

```
#限制查询返回结果
print( session.query(User).filter(User.username!='budong').limit(2).all())
print( session.query(User).filter(User.username!='budong').offset(2).all())
print( session.query(User).filter(User.username!='budong').slice(2,3).all())

#可以排序之后再进行限制
from sqlalchemy import desc
print( session.query(User).filter(User.username!='budong').order_by(User.username).all())
print(
session.query(User).filter(User.username!='budong').order_by(desc(User.username)).slice(1,3).all
())
```

第一个是限制返回条数，从第一条开始；第二个是从第三条开始返回查询结果；第三个是切片返回记录。

`order_by`默认是顺序，`desc`是降序。

还有其他的带条件查询

```
#不等于
print( session.query(User).filter(User.username!='budong').all() )
#模糊匹配 like
print( session.query(User).filter(User.username.like('budong')).all() )
print( session.query(User).filter(User.username.notlike('budong')).all() )
#成员属于  in_
print( session.query(User).filter(User.username.in_(['budong','tuple'])).all() )
#成员不属于  notin_
print( session.query(User).filter(User.username.notin_(['budong','tuple'])).all() )
#空判断
print( session.query(User).filter(User.username==None).all() )
print( session.query(User).filter(User.username.is_(None)).all() )
print( session.query(User).filter(User.username.isnot(None)).all() )
#多条件
print( session.query(User).filter(User.username.isnot(None),User.password=='qwe123').all() )
#选择条件
from sqlalchemy import or_,and_,all_,any_
print( session.query(User).filter(or_(User.username=='budong',User.password=='qwe123')).all() )
print( session.query(User).filter(and_(User.username=='budong',User.password=='111')).all() )
```

以上是各种带条件的查询，大家知道怎么使用，但是需要注意的是，所以的模糊匹配是十分耗费时间的，能不用就尽量不要用。

当然还有聚合函数的使用

```
#聚合函数的使用
from sqlalchemy import func,extract
print( session.query(User.password,func.count(User.id)).group_by(User.password).all() )
print(
session.query(User.password,func.count(User.id)).group_by(User.password).having(func.count(User.
id)>1).all() )
print( session.query(User.password,func.sum(User.id)).group_by(User.password).all() )
print( session.query(User.password,func.max(User.id)).group_by(User.password).all() )
print( session.query(User.password,func.min(User.id)).group_by(User.password).all() )
#使用extract提取时间中的分钟或者天来分组
print( session.query(extract('minute',
User.creatime).label('minute'),func.count('*').label('count')).group_by('minute').all() )
print( session.query(extract('day',
User.creatime).label('day'),func.count('*').label('count')).group_by('day').all() )
```

这里只是告诉大家的用法，其中`group_by`是分组，如果要使用聚合函数，就必须导入`func`，`label`是取别名的意思 。

2.表关系查询

对于有表关系的，也有些不同的查询，首先我们来建立一个有外键关系的表

```
from sqlalchemy.orm import relationship
from sqlalchemy import ForeignKey

class UserDetails(Base):
    __tablename__ = 'user_details'
    id = Column(Integer,primary_key=True,autoincrement=True)
    id_card = Column(Integer,nullable=False,unique=True)
    lost_login = Column(DateTime)
    login_num = Column(Integer,default=0)
    user_id = Column(Integer,ForeignKey('user.id'))

    userdetail_for_foreignkey =
relationship('User',backref='details',uselist=False,cascade='all')

    def __repr__(self):
        return '<UserDetails(id=%s,id_card=%s,lost_login=%s,login_num=%s,user_id=%s)>'%(
            self.id,
            self.id_card,
            self.login_login,
            self.login_num,
            self.user_id
        )
```

这里要注意`relationship`默认是一对多的关系，使用`uselist=False`则表示一对一的关系，`cascade`是自动关系处理，就和MySQL中的`ON DELETE`类似，但是有区别，参数选项如下：

`cascade`所有的可选字符串项是:

- *all* , 所有操作都会自动处理到关联对象上.
- *save-update* , 关联对象自动添加到会话.
- *delete* , 关联对象自动从会话中删除.
- *delete-orphan* , 属性中去掉关联对象, 则会话中会自动删除关联对象.
- *merge* , `session.merge()` 时会处理关联对象.
- *refresh-expire* , `session.expire()` 时会处理关联对象.
- *expunge* , `session.expunge()` 时会处理关联对象.

有如上的表关系之后，查询可以十分方便

```
#表关系查询
row = session.query(UserDetails).all()
print(row,dir(row[0]))
row = session.query(User).filter(User.id==1).first()
print(row,dir(row))
print(row.details)
print(row.details[0].lost_login)
```

`relationship`会在`User`表里面添加一个属性，通过这个属性就可以查询对应的`user_details`表中的所有字段。省去了很多的代码。

3.多表查询

多表查询也是必须要掌握的知识点。以下是常见的几种表关联方式，需要熟练掌握。

```
#多表查询
print( session.query(UserDetails,User).all() )  #这个是 cross join
print( session.query(UserDetails,User).filter(User.id==UserDetails.id).all() )  #这是也是cross
join 但是加上了where条件

print(
session.query(User.username,UserDetails.lost_login).join(UserDetails,UserDetails.id==User.id).al
l() )  #这个是inner join

print(
session.query(User.username,UserDetails.lost_login).outerjoin(UserDetails,UserDetails.id==User.i
d).all() )  #这个才是左连接，sqlalchemy没有右连接

q1 = session.query(User.id)
q2 = session.query(UserDetails.id)
print(q1.union(q2).all())  #这个是union关联
```

除了上面的几种关联方式，子表查询也是用得很多的，也是要掌握的

```
from sqlalchemy import all_,any_
sql_0 = session.query(UserDetails.lost_login).subquery()  #这是声明一个子表
print( session.query(User).filter((User.creatime > all_(sql_0)) ).all()  )
print( session.query(User).filter((User.creatime > any_(sql_0)) ).all()  )
```

注意`any_`和`all_`的区别，`all_`要求的是所有都满足，`any_`只需要有满足的就行。

4.原生SQL的查询以及其他使用

再次强调，使用`ORM`或者原生`SQL`没有绝对的那个好一点，怎么方便怎么使用。

```
#第一步写好原生的sql，如果需要传递参数，可以使用字符串拼接的方式
sql_1 = """
    select * from `user`
"""
#第二步执行，得到返回的结果
row = session.execute(sql_1)
print(row,dir(row))
#第三步，自己控制得到数据的方式
print( row.fetchone() )
print( row.fetchmany() )
print( row.fetchall() )
#也可以循环获得
for i in row:
    print('===',i)
```

第四章 SQLAlchemy多表查询

`20:30上课！！！`

1.多表查询

2.原生SQL的查询

3.序列化

1.多表查询

在MySQL中我们讲了多表查询，在`SQLAlchemy`中也有多表查询的概念

```
#不一定两张表有外键关系才可以一起关联查询，只要给出关联条件就可以
rows =
session.query(User.id,User.name,Column_test.id,Column_test.name).filter(User.id==Column_test.id)
.all()

#也可以使用join
rows = session.query(Student.name,User.number).join(User,Student.name==User.name).all()
print(rows)

#outerjoin代表left join ，在SQLAlchemy中没有右连接
rows = session.query(Student.name,User.number).outerjoin(User,Student.name==User.name).all()

#对于union也是可以使用的
q1 = session.query(Student.name)
q2 = session.query(User.name)
rows = q1.union(q2).all()

#子表查询
from sqlalchemy import func
q3 =
session.query(Student.dep_id,func.count('*').label('dep_count')).group_by(Student.dep_id).subque
ry()
print(q3)
rows = session.query(Department,q3.c.dep_count).outerjoin(q3,q3.c.dep_id==Department.id).all()
print(rows)
#这里注意，使用查询子表查询的时候使用  q3.c.dep_count  这种方式去取得对应的属性
#subquery 就是子查询的意思
#group_by  having  order_by 等这些都是可用的
#label 是标签的意思，这里的作用类似与MySQL中的as
#如果要使用聚合函数，需要导入func模块，导入之后就可以是使用各种函数，只要连接的数据库支持
#这个查询，几乎把我们常会用到的各种情况给演示出来，加上之前我们讲过的一些查询，我们在工作中可能会遇到的查
询，基本上都已经讲解
```

多表查询使用`SQL Alchemy`同样也可以达到我们写`SQL`才能做到的事情，要是要想熟练掌握，需要自己多练习，多尝试。

2.原生SQL的查询

在实际的使用过程中，有些时候可能会遇到用SQLAlchemy不能够很好利用数据库的特性，或者需要写很多关联的时候，我们也可以写原生的SQL，然后使用SQLAlchemy去执行。

```
sql_0 = """
SELECT
    *
FROM
    `user`
"""
#print(dir(session))
rows = session.execute(sql_0)
#print(rows,type(rows),dir(rows))
r = rows.fetchone()
print('***',r)
for i in rows:
    print('===',i)
#这里的 execute 便是执行原生SQL的方法
#fetchone 是每次取一条数据的意思
#可以通过循环把这张表的数据依次全部取完
```

在同样可以传递参数

```
sql_1 = """
SELECT
    *
FROM
    student
WHERE
    dep_id = :id
"""
rows = session.execute(sql_1,{'id':1})
r = rows.fetchall()
print(r)
for i in r:
    print(i)

#这里的 fetchall 是全部取出的意思，也可以通过for循环给依次打印出来
#这里的 :id  是其语法规则，但是这里的 sql_1 是个字符串，我们可以使用字符串拼接或者格式化输出的方式来传递变量
sql_1 = """
SELECT
    *
FROM
    student
WHERE
    dep_id = %s
"""%(1)
#或者
sql_1 = """
SELECT
    *
FROM
    student
WHERE
    dep_id = {}
""".format(1)
#如果使用字符串的方法，那么 execute 需要改成如下的形式
rows = session.execute(sql_1)
```

使用`execute`也可以执行更新，删除和插入操作

```
sql_2 = """
UPDATE `user`
SET age = %s
WHERE
    id = %s ;
"""%(18,1)
rows = session.execute(sql_2)
print(rows)
```

3.序列化

如果希望透明地存储 Python 对象，而不丢失其身份和类型等信息，则需要某种形式的对象序列化：它是一个将任意复杂的对象转成对象的文本或二进制表示的过程。同样，必须能够将对象经过序列化后的形式恢复到原有的对象。

`Json`

用于字符串和`python`数据类型之间的转换，`JSON(JavaScript Object Notation)`是一种轻量级的数据交换格式,在前后端传输数据中经常使用。

假如，我要传送一个字典出去，需要解决两个主要问题：

怎么表达这个字典成一个字符串？怎么把一个字符串，读回一个字典？

因此，我们需要一个标准化的，字符串表达方式，例如`JSON`，有点类似于，编码与解码。

基本接口：

```
import json
di = {'a':1,1:2}
a = json.dumps(di)
#1.dumps(python对象)
#通常是，字典，元祖，列表，字符串，数字
#返回的是一个，JSON字符串

b = json.loads(a)
#2. loads(string)
#读取一个JSON字符串，如果满足格式标准，那么就会返回出如下四种之一
#字典，列表，字符串，数字

#3. dump(obj, file)
#dumps + open + write

#例：
import json

the_dict = {
    'a':123,
    'b':456
}
with open('json.txt', 'w') as f:
    json.dump(the_dict, f)

#4. load(file)
#loads + open + read

with open('json.txt','r') as f:
    print json.load(f)
```

Json主要应用于前后台的数据传输，在前台也有json格式的文件，前台可以把数据转成json格式的文件，传输到后台，后台再把json解析出来，再做相应处理。

`Pickle`

把变量从内存中变成可存储或传输的过程称之为序列化,也称之为对象的持久化保存

`pickle`实现`python`的`bytes`类型与`python`其他数据类型之间的转换。

```
import pickle
di = {'a':1,'b':2}
#1.dumps(python对象)
#通常是，字典，元祖，列表，字符串，数字
#返回的是一个，bytes字符串

#2. loads(string)
#读取一个bytes字符串，如果满足格式标准，那么就会返回出如下四种之一
#字典，列表，字符串，数字

#3. dump(obj, file)
#dumps + open + write

#例：
import pickle

the_dict = {
    'a':123,
    'b':456
}
with open('pickle.txt', 'wb') as f:
    pickle.dump(the_dict, f)

#4. load(file)
#loads + open + read

with open('pickle.txt','rb') as f:
    xx = pickle.load(f)
print(xxx)
```

在`python`内部，`json`和`pickle`是一样的，但是`pickle`是`python`才有的模块，所有一般情况下会比`json`要快一些，但是使用方法是一样的。

从空间和时间上说，`Pickle`是可移植的。换句话说，`pickle`文件格式独立于机器的体系结构，这意味着，例如，可以在`Linux`下创建一个`pickle`，然后将它发送到在`Windows`或`Mac OS`下运行的`Python`程序。并且，当升级到更新版本的`Python`时，不必担心可能要废弃已有的`pickle`。`Python`开发人员已经保证`pickle`格式将可以向后兼容`Python`各个版本。

总结

`SQLALchemy`的学习就这么多，在工作中常用的也就是这些，大家要熟练两个地方，一个是写表的对应的类，即`module`；第二个是`query`。这两个是一定要会的。今天讲的多表查询和原生`SQL`的查询，也是常用的，需要多加练习。

`Json`和`Pickle`模块的使用很简单，使用过一次就知道，具体的应用在后面的学习中会逐步地体会到。