

```

</div>

<div id="content">
    <p class="important">
        我是在子模板中。
    </p>
</div>

</body>
</html>

```

8.6 SQLAlchemy 数据库编程

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 使用 MIT 许可证发行。它采用简单的 Python 语言，为高效和高性能的数据库访问设计，实现了完整的企业级持久模型。 SQLAlchemy 非常关注数据库的量级和性能。

注意：本节介绍的 SQLAlchemy 基于稳定版 1.0.11。

8.6.1 SQLAlchemy 入门

本节通过一套例子分析 SQLAlchemy 的使用方法。

使用 SQLAlchemy 至少需要 3 部分代码，它们分别是定义表、定义数据库连接、进行增、删、改、查等逻辑操作。下面是 SQLAlchemy 定义表的一个例子，保存在 `orm.py` 文件中：

```

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

Base = declarative_base()

class Account(Base):
    __tablename__ = u'account'

    id = Column(Integer, primary_key=True)

```

```

user_name = Column(String(50), nullable=False)
password = Column(String(200), nullable=False)
title = Column(String(50))
salary = Column(Integer)

def is_active(self):
    #假设所有用户都是活跃用户。
    return True

def get_id(self):
    #返回账号 ID，用方法返回属性值提高了表的封装性。
    return self.id

def is_authenticated(self):
    #假设已经通过验证
    return True

def is_anonymous(self):
    #具有登录名和密码的账号不是匿名用户
    return False

```

解析定义表的代码如下。

- SQLAlchemy 表之前必须引入 `sqlalchemy.ext.declarative.declarative_base`，并定义一个它的实例 `Base`。所有表必须继承自 `Base`。本例中定义了一个账户表类 `Account`。
- 通过 `_tablename_` 属性定义了表在数据库中实际的名称 `account`。
- 引入 `sqlalchemy` 包中的 `Column`、`Integer`、`String` 类型，因为需要用它们定义表中的列。本例在 `Account` 表中定义了 5 个列，分别是整型 `id` 和 `salary`，以及字符串类型的 `user_name`、`password`、`title`。
- 在定义列时可以通过给 `Column` 传递参数定义约束。本例中通过 `primary_key` 参数将 `id` 列定义为主键，通过 `nullable` 参数将 `user_name` 和 `password` 定义为非空。
- 在表中还可以自定义其他函数。本例中定义了用户验证时常用的几个函数：`is_active()`、`get_id()`、`is_authenticated()` 和 `is_anonymous()`。

定义数据库连接的示例代码如下：

```

from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker

```

```

db_connect_string='mysql://v_user:v_pass@localhost:3306/test_database?charset=utf8'
ssl_args = {'ssl': {'cert':'/home//ssl/client-cert.pem',
                   'key':'/home/shouse/ssl/client-key.pem',
                   'ca':'/home/shouse/ssl/ca-cert.pem'}}
engine = create_engine(db_connect_string, connect_args=ssl_args)
SessionType = scoped_session(sessionmaker(bind=engine, expire_on_commit=False))
def GetSession():
    return SessionType()

from contextlib import contextmanager
@contextmanager
def session_scope():
    session = GetSession()
    try:
        yield session
        session.commit()
    except:
        session.rollback()
        raise
    finally:
        session.close()

```

解析此连接数据部分的代码如下。

- 引入数据库和会话引擎： sqlalchemy.create_engine、 sqlalchemy.orm.scoped_session、 sqlalchemy.orm.sessionmaker。
- 定义连接数据库需要用到的数据库字符串。本例连接 MySQL 数据库，字符串格式为 [database_type]://[user_name]:[password]@[domain]:[port]/[database]?[parameters]。本例中除了必需的连接信息，还传入了 charset 参数，指定用 utf-8 编码方式解码数据库中的字符串。
- 用 create_engine 建立数据库引擎，如果数据库开启了 SSL 链路，则在此处需要传入 ssl 客户端证书的文件路径。
- 用 scoped_session(sessionmaker(bind=engine)) 建立会话类型 SessionType，并定义函数 GetSession() 用以创建 SessionType 的实例。

至此，已经可以用 GetSession() 函数建立数据库会话并进行数据库操作了。但为了使之后的数据库操作的代码能够自动进行事务处理，本例中定义了上下文函数 session_scope()。在 Python 中定义上下文函数的方法是为其加入 contextlib 包中的 contextmanager 装饰器。在上下文函数中

执行如下逻辑：在函数开始时建立数据库会话，此时会自动建立一个数据库事务；当发生异常时回滚（rollback）事务；当退出时关闭（close）连接。在关闭连接时会自动进行事务提交（commit）操作。

【示例 8-15】进行数据库操作的代码如下：

```

import orm
from sqlalchemy import or_
from sqlalchemy import and_

def InsertAccount( user, password, title, salary):          #新增操作
    with session_scope() as session:
        account=orm.Account(user_name=user, password=password , title=title, salary=salary)
        session.add(account)

def GetAccount(id=None, user_name=None):                      #查询操作
    with session_scope() as session:
        return session.query(orm.Account).filter(
            or_(orm.Account.id == id, orm.Account.user_name == user_name)).first()

def DeleteAccount( user_name):                                #删除操作
    with session_scope() as session:
        account = GetAccount(user_name=user_name)
        if account:
            session.delete(account)

def UpdateAccount( id, user_name, password, title, salary): #更新操作
    with session_scope() as session:
        account = session.query(orm.Account).filter(orm.Account.id==id).first()
        if not account: return
        account.user_name=user_name
        account.password=password      account.salary = salary
        account.title = title

InsertAccount("David Li", "123", "System Manager", 3000)      #调用新增操作
InsertAccount("Rebeca Li", "", "Accountant", 3000)

GetAccount(2)                                                 #调用查询操作

DeleteAccount("Howard")
UpdateAccount(1, "admin", "none", "System Admin", 2000)

```

本例演示了数据库中最常用的 4 种基于记录的操作：新增、查找、删除、更新。对此部分代码的解析如下。

- 用 import 语句引入数据表（Account）所在的包 orm。引入多条件查询时的或连接 or_。
- 每个函数中都通过 with 语句启用上下文函数 session_scope()，通过它获取到 session 对象，并自动开启新事务。
- 在 InsertAccount 中，通过新建一个表 Account 实例，并通过 session.add 将其添加到数据库中。由于上下文函数退出时会自动提交事务，所以无须显式地调用 session.commit() 使新增生效。
- 在 GetAccount 中通过 query 语句进行查询，查询条件由 filter 设置，多个查询条件可以用 or_ 或 and_ 连接。
- 在 DeleteAccount 中通过 GetAccount 查询该对象，如果查询到了，则直接调用 session.delete() 将该对象删除。
- 在 InsertAccount 中通过 query 根据 id 查询记录，如果查询到了，则通过设置对象的属性实现对记录的修改。由于上下文函数退出时会自动提交事务，所以无须显式地调用 session.commit() 使修改生效。
- 查询语句的结果是一个对象集合。查询语句后面的 first() 函数用于提取该集合中的第 1 个对象。类似地，如果用 all() 函数替换 first() 函数，查询则会返回该集合。

8.6.2 主流数据库的连接方式

SQLAlchemy 这样的 ORM 数据库操作方式可以对业务开发者屏蔽不同数据库之间的差异，这样当需要进行数据库迁移时（比如从 MySQL 迁移到 SQLite），则只需更换数据库连接字符串。表 8.4 列出了 SQLAlchemy 连接主流数据库时的数据库字符串的编写方法。

表 8.4 SQLAlchemy 对主流数据库的连接字符串

数 据 库	连接字符串
Microsoft SQLServer	'mssql+pymssql://[user]:[pass]@[domain]:[port]/[dbname]'
MySQL	'mysql:// [user]:[pass]@[domain]:[port]/[dbname]'
Oracle	'oracle:// [user]:[pass]@[domain]:[port]/[dbname]'
PostgreSQL	'postgresql:// [user]:[pass]@[domain]:[port]/[dbname]'
SQLite	'sqlite:///[file_pathname]'

8.6.3 查询条件设置

在实际编程时需要根据各种不同的条件查询数据库记录，SQLAlchemy 查询条件被称为过滤器。这里列出了最常用的过滤器的使用方法。

(1) 等值过滤器（==）

等值过滤器用于判断某列是否等于某值，是最常用的过滤器。

session.query(Account).filter(Account.user_name == "Jacky")	#判断字符串类型
session.query(Account).filter (Account.salary == 2000)	#判断数值类型

(2) 不等过滤器（!=、<、>、<=、>=）

与等值过滤器相对的是不等过滤器，不等过滤器可以延伸为几种形式：不等于、小于、大于、小于等于、大于等于。

session.query(Account).filter (Account.user_name != "Jacky")	#判断字符串类型
session.query(Account).filter (Account.salary != 2000)	#判断数值类型
session.query(Account).filter (Account.salary>3000)	#大于过滤器
session.query(Account).filter (Account.salary < 3000)	#小于过滤器
session.query(Account).filter (Account.salary <= 3000)	#小于等于
session.query(Account).filter (Account.salary >= 3000)	#大于等于

(3) 模糊查询（like）

模糊查询适用于只知道被查字符串的一部分内容时，通过设置通配符的位置，可以查询出不同的结果。通配符用百分号%表示。

假设数据库中 Account 表的内容如表 8.6 所示，则模糊查询的方法如下：

#查询所有名字中包含字母 i 的用户，结果包括 id 为 1、2、3、4 的 4 条记录
session.query(Account).filter (Account.user_name.like('%i%'))
#查询所有 title 中以 Manager 结尾的用户，结果包括 id 为 1、5 的两条记录
session.query(Account).filter (Account.title.like('%Manager'))
#查询所有名字中以 Da 开头的用户，结果包括 id 为 1、3 的两条记录
session.query(Account).filter (Account.user_name.like('Da%'))

表 8.6 查询示例表

id	user_name	title	salary
1	David Li	System Manager	3000
2	Rebeca Li	Accountant	3000
3	David Backer	Engineer	3000
4	Siemon Bond	Engineer	4000
5	Van Berg	General Manager	NULL

注意：模糊查询只适用于查询字符串类型，不适用于数值类型。

(4) 包括过滤器 (in_)

当确切地知道要查询记录的字段内容，但是一个字段有多个内容要查询时，可以用包含过滤器。以下例子演示了用包含关系查询表 8.6 的内容的结果：

```
#查询 id 不为 1、3、5 的记录，结果包含 id 为 2、4 的两条记录
session.query(Account).filter (~Account.id.in_([1,3,5])) 

#查询工资不为 2000、3000、4000 的记录，结果包含 id 为 5 的 1 条记录
session.query(Account).filter (~Account.id.in_([2000, 3000,4000])) 

#查询所有 title 不为 Engineer 和 Accountant 的记录，结果包括 id 为 1、5 的两条记录
session.query(Account).filter (~Account.title.in(['Accountant', 'Engineer']))
```

(5) 判断是否为空 (is NULL、is not NULL)

空值 NULL 是数据库字段中比较特殊的值。在 SQLAlchemy 中支持对字段是否为空进行判断。判断时可以用等值、不等值过滤器筛选，也可以用 is、isnot 进行筛选。以下例子演示了对表 8.6 的内容进行基于空值查询的结果：

```
#查询 salary 为空值的记录，结果包含 id 为 5 的记录
#此下两种方式的效果相同
session.query(Account).filter (Account.salary == None)
session.query(Account).filter (Account.salary.is_(None)) 

#查询 salary 不为空值的记录，结果包含 id 为 1、2、3、4 的记录
#此下两种方式的效果相同
session.query(Account).filter (Account.salary != None)
session.query(Account).filter (Account.salary.isnot( None))
```

(6) 非逻辑 (~)

当需要查询不满足某条件的记录时可以使用非逻辑。以下例子演示了用非逻辑查询表 8.6 的内容的结果：

```
#查询 id 不为 1、3、5 的记录，结果包含 id 为 2、4 的两条记录
session.query(Account).filter (~Account.id.in_([1,3,5]))  
  
#查询工资不为 2000、3000、4000 的记录，结果包含 id 为 5 的 1 条记录
session.query(Account).filter (~Account.id.in_([2000, 3000,4000]))  
  
#查询所有 title 不为 Engineer 和 Accountant 的记录，结果包括 id 为 1、5 的 2 条记录
session.query(Account).filter (~Account.title.in(['Accountant', 'Engineer']))
```

(7) 与逻辑 (and_)

当需要查询同时满足多个条件的记录时，需要用到与逻辑。在 SQLAlchemy 中与逻辑可以有 3 种表达方式。以下 3 条语句对表 8.6 的查询结果相同，都是 id 为 3 的记录：

```
#直接在 filter 中添加多个条件即表示与逻辑
session.query(Account).filter (Account.title=='Engineer', Account.salary=3000)  
  
#用关键字 and_ 进行与逻辑查询
from sqlalchemy import and_
session.query(Account).filter (and_(Account.title=='Engineer', Account.salary=3000))  
  
#通过多个 filter 的链接表示与逻辑
session.query(Account).filter (Account.title=='Engineer').filter( Account.salary=3000)
```

(8) 或逻辑 (or_)

当需要查询多个条件但只需其中一个条件满足时，需要用到或逻辑。以下例子演示用非逻辑查询表 8.6 的内容的结果：

```
#引入或逻辑关键字 or_
from sqlalchemy import or_
  
#查询 tilte 是 Engineer 或者 salary 为 3000 的记录，返回结果为 id 为 1、2、3、4 的记录
session.query(Account).filter (or_(Account.title=='Engineer', Account.salary=3000))  
  
#查询 tilte 是 Accountant 或者 salary 为 4000 的记录，返回结果为 id 为 2、4 的记录
session.query(Account).filter (or_(Account.title=='Accountant', Account.salary=4000))
```

8.6.4 关系操作

关系数据库是建立在关系模型基础上的数据库，所以表之间的关系在数据库编程中尤为重要。本节围绕在 SQLAlchemy 中如何定义关系及如何使用关系进行查询进行讲解，使读者能够快速掌握 SQLAlchemy 的关系操作。

1. E-R 图设计

通过 E-R 图可以完成数据库的逻辑和物理设计。作为演示实例，设计学校信息管理系统的老师、班级、学生子系统如图 8.7 所示。

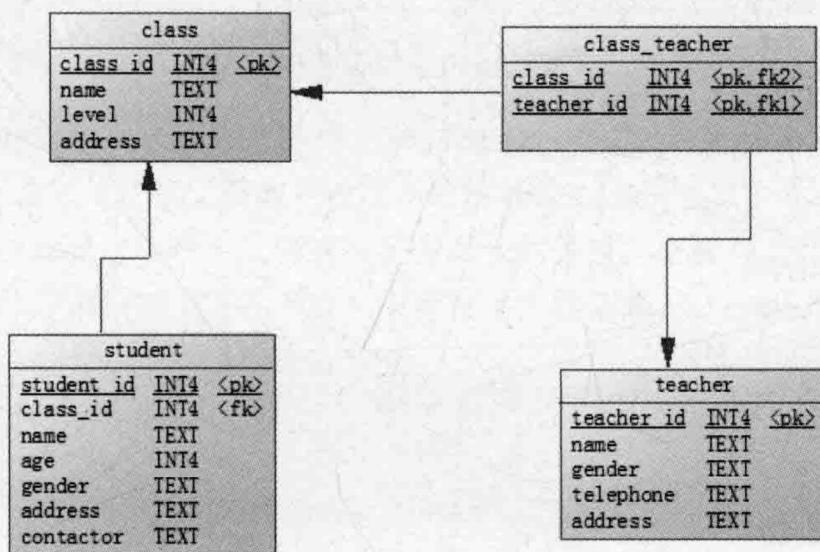


图 8.7 学校老师、班级、学生子系统的 E-R 图

图中设计了 3 个实体表（班级表 class、学生表 student、老师表 teacher）和 1 个关系表（class_teacher）。图中每个表中的第 1 列为属性名（name、class_id、address 等），第 2 列为属性类型（INT4、TEXT 等），第 3 列为约束（<pk> 表示主键约束，<fk> 表示外键约束）。图中班级与学生为一对多关系，班级与老师之间为多对多关系。

2. 将 E-R 图转为 SQLAlchemy 表达方式

图 8.5 包含了表、主键、外键等关系操作定义的核心内容。

【示例 8-16】如下是将其转换为 SQLAlchemy 模型定义的代码文件：

```

from sqlalchemy import Table, Column, Integer, ForeignKey, String
from sqlalchemy.orm import relationship, backref

```

```

from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Class(Base):
    __tablename__ = 'class'
    class_id = Column(Integer, primary_key=True)
    name= Column(String(50))
    level = Column(Integer)
    address = Column(String(50))

    class_teachers = relationship("ClassTeacher", backref="class")
    students = relationship("Student", backref="class")

class Student(Base):
    __tablename__ = 'student'
    student_id = Column(Integer, primary_key=True)
    name= Column(String(50))
    age = Column(Integer)
    gender= Column(String(10))
    address= Column(String(50))
    class_id = Column(Integer, ForeignKey('class.id'))

class Teacher(Base):
    __tablename__ = 'teacher'
    teacher_id = Column(Integer, primary_key=True)
    name= Column(String(50))
    gender= Column(String(10))
    telephone= Column(String(50))
    address= Column(String(50))
    class_teachers = relationship("ClassTeacher", backref="teacher")

class ClassTeacher(Base):
    __tablename__ = 'class_teacher'
    teacher_id = Column (Integer, ForeignKey('teacher.teacher_id'), primary_key=True)
    class_id = Column(Integer, ForeignKey('class.id'),primary_key=True)

```

代码中用 4 个 SQLAlchemy 模型对 4 个表进行了定义，其中与关系定义相关的部分如下。

- 外键设置：在列的定义中，为 Column 传入 ForeignKey 进行外键设置。

```
class_id = Column(Integer, ForeignKey('class.id'))
```

- 关系设置：通过 relationship 关键字在父模型中建立对子表的引用，例如 Class 模型中的关系设置如下。

```
students = relationship("Student", backref="class")
```

其中的 backref 参数为可选参数，如果设置 backref，则此条语句同时设置了从父表对子表的引用。

- 一对多关系的使用：以后可以直接通过该 students 属性获得相关班级中所有学生的信息。如下代码可以打印班级“三年二班”的所有学生信息。

```
class_=session.query(Class).filter(Class.name=="三年二班").first()
for student in class_.students:
    print "姓名: %s, 年龄: %d"%(student.name, student.age)
```

- 多对多关系的使用：通过关联模型 ClassTeacher 实现，在其中分别设置模型 Class 和 Teacher 的外键，并且在父模型中设置相应的 relationship 实现。多对多关系也可以想象成一个关联表，分别对两个父表实现了多对一的关系。班级与老师之间为多对多关系，如下代码可以打印班级“三年二班”中所有老师的信息。

```
class_=session.query(Class).filter(Class.name=="三年二班").first()
for class_teacher in class_.class_teachers:
    teacher = class_teacher.teacher
    print "姓名: %s, 电话: %s"%(teacher.name, teacher.telephone)
```

注意：上述代码中 class_teacher.teacher 是在模型 Teacher 中针对 ClassTeacher 定义的反向引用。

3. 连接查询

在实际开发中，有了关系就必不可少地会有多表连接查询的需求。下面通过实际例子演示如何进行多表连接查询。

【示例 8-17】在查询语句中可以使用 join 关键字进行连接查询，打印出所有三年级学生的姓名：

```
students = session.query(Student).join(Class).filter(Class.level==3).all()
for student in students:
    print student.name
```

上述查询函数会自动把外键关系作为连接条件，该查询被 SQLAlchemy 自动翻译为如下

SQL 语句并执行：

```
SELECT student.student_id AS student_student_id,
       student.name AS student_name,
       student.age AS student_age,
       student.gender AS student_gender,
       student.address AS student_address,
       student.class_id AS student_class_id
  FROM student JOIN class ON student.class_id = class.class_id
 WHERE class.level= ?
(3,)
```

【示例 8-18】如果需要将被连接表的内容同样打印出来，则可以在 query 中指定多个表对象。下面的语句在打印出所有三年级学生的姓名的同时，打印出其所在班级的名字。

```
for student, class_ in session.query(Student,
Class).join(Class).filter(Class.level==3).all():
    print student.name, class_.name
```

上述查询函数会自动把外键关系作为连接条件，该查询被 SQLAlchemy 自动翻译为如下 SQL 语句并执行：

```
SELECT student.student_id AS student_student_id,
       student.name AS student_name,
       student.age AS student_age,
       student.gender AS student_gender,
       student.address AS student_address,
       student.class_id AS student_class_id,
       class.class_id AS class_class_id,
       class.name AS class_name,
       class.level AS class_level,
       class.address AS class_location
  FROM student JOIN class ON student.class_id = class.class_id
 WHERE class.level= ?
(3,)
```

【示例 8-19】如果需要用除外键外的其他字段作为连接条件，则需要开发者在 join 中自行设置。下面打印出所有班级的 address 与学生的 address 相同的学生的姓名：

```
for student_name, in session.query(Student.name) . \
join(Class, Class.address == Student.address).filter(Class.level==3).all():
    print student_name
```

上述查询函数根据开发者指定的语句作为连接条件，并且因为直接指定了被查询的字段，所以减少了实际 SQL 中的被查询字段，提高了性能。该查询被 SQLAlchemy 自动翻译为如下 SQL 语句并执行：

```
SELECT student.name AS student_name,
FROM student JOIN class ON student.address = class.address
```

8.6.5 级联

级联是在一对多关系中父表与子表进行联动操作的数据库术语。因为父表与子表通过外键关联，所以对父表或子表的增、删、改操作会对另一张表也产生相应的影响。适当地利用级联可以开发出更优雅、健壮的数据库程序。本节学习 SQLAlchemy 中级联的操作方法。

注意： SQLAlchemy 级联独立于 SQL 本身针对外键的级联定义。即使在数据库表定义中没有定义 ON DELETE 等属性，也不影响开发者在 SQLAlchemy 中使用级联。

1. 级联定义

SQLAlchemy 中的级联通过对父表中的 relationship 属性定义 cascade 参数来实现，代码如下：

```
from sqlalchemy import Table, Column, Integer, ForeignKey, String
from sqlalchemy.orm import relationship, backref
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Class(Base):
    __tablename__ = 'class'
    class_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    level = Column(Integer)
    address = Column(String(50))

    students = relationship("Student", backref="class_", cascade="all")

class Student(Base):
    __tablename__ = 'student'
    student_id = Column(Integer, primary_key=True)
```

```

name= Column(String(50))
age = Column(Integer)
gender= Column(String(10))
address= Column(String(50))
class_id = Column(Integer, ForeignKey('class.class_id'))

```

上述代码定义了班级表 Class（父表）和学生表 Student（子表）。一对多的关系由父表中的 relationship 属性 students 进行定义。relationship 中的 cascade 参数定义了要在该关系上实现的级联方法“all”。

SQLAlchemy 中另外一种设置级联的方式是在子表的 relationship 的 backref 中进行设置。比如上述代码可以写为如下形式，意义保持不变。

```

from sqlalchemy import Table, Column, Integer, ForeignKey, String
from sqlalchemy.orm import relationship, backref
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Class(Base):
    __tablename__ = 'class'
    class_id = Column(Integer, primary_key=True)
    name= Column(String(50))
    level = Column(Integer)
    address = Column(String(50))

class Student(Base):
    __tablename__ = 'student'
    student_id = Column(Integer, primary_key=True)
    name= Column(String(50))
    age = Column(Integer)
    gender= Column(String(10))
    address= Column(String(50))
    class_id = Column(Integer, ForeignKey('class.class_id'))
    class_ = relationship("Class", backref=backref("students", cascade="all"))

```

上述代码没有在父表 Class 中设置 relationship 和 cascade，而是在子表中设置 relationship，并在其 backref 中设置 cascade。

SQLAlchemy 中可选的 cascade 取值范围如表 8.7 所示。

表 8.7 cascade 属性取值范围

可选值	意 义
save-update	当一个父对象被新增到session中时，该对象当时关联的子对象也自动被新增到session中
merge	Session.merge是一个对数据库对象进行新增或更新的方法。cascade取值为merge时的意义为：当父对象进行merge操作时，该对象当时关联的子对象也会被merge
expunge	Session.expunge是一种将对象从session中移除的方法。cascade取值为expunge时的意义为：当父对象进行了expunge操作时，该对象当时关联的子对象也会被从session中删除
delete	当父对象被delete时，子对象也同时被delete
delete-orphan	当子对象不再与任何父对象关联时，会自动将该子对象删除
refresh-expire	Session.expire是一种设置对象已过期、下次引用时需要从数据库即时读取的方法。cascade取值为refresh-expire时的意义为：当父对象进行了expire操作时，该对象当时关联的子对象也进行expire操作
all	是一个集合值，表示save-update、merge、refresh-expire、expunge、delete同时被设置

多个 cascade 属性可以通过逗号分隔并同时赋值给 cascade。比如如下代码同时设置了 save-update、merge 和 expunge 的属性：

```
students = relationship("Student", backref="class_", cascade="save-update, merge, expunge")
```

在默认情况下，任何 relationship 的级联属性都被设置为 cascade="save-update, merge"。本节在后续部分将对表 8.7 中最常用的参数 save-update、delete、delete-orphan 的功能进行举例说明。

2. save-update 级联

save-update 级联是指当一个父对象被新增到 session 中时，该对象当时关联的子对象也自动被新增到 session 中。

【示例 8-20】通过如下代码建立一个父对象 class 和两个子对象 student1 与 student2：

```
class_=Class()
student1, student2=Student(), Student()
class_.students.append(student1)
class_.students.append(student2)
```

如果父子级联关系包含 save-update，则只需将父对象保存到 session 中，子对象会自动被保存。续写上述代码如下：

```
session.add(class_)
if student1 in session:
    print "The student1 has been added too!"
```

这段代码将打印 “The student1 has been added too!”。

技巧：“in”语句可以判断某对象是否被关联到了 session 中。已被关联的对象在 session 被 commit 时会被写入到数据库中。

【示例 8-21】即使父对象已经被新增到 session 中，新关联的子对象仍然可以被添加：

```
class_=Class()
session.add(class_)
student3= Student()
if student3 in session:
    print "The student3 is added before append to the class_!"
class_.students.append(student3)
if student1 in session:
    print "The student3 is added after append to the class_!"
```

这段代码将打印“`The student3 is added after append to the class_!`”。

3. delete 级联

顾名思义，`delete` 级联是指当父对象被从 `session` 中 `delete` 时，其关联的子对象也自动被从 `session` 中 `delete`。通过一个例子演示 `delete` 的作用，假设数据库中 `class` 表和 `student` 表的内容如表 8.8 和表 8.9 所示。

表 8.8 例表—class

class_id	name	level	address
1	三年二班	3	李冰路410号1楼
2	五年一班	5	李冰路410号3楼
3	五年二班	5	李冰路410号3楼

表 8.9 例表—student

student_id	class_id	name	age	gender	address	contactor
1	1	李晓	10	男	虹口区...	NULL
2	1	单梦童	10	女	虹口区...	NULL
3	1	林一雷	9	女	闸北区...	林廷玉
4	2	丁辉	10	男	宝山区...	NULL
5	2	王文文	12	女	虹口区...	王飞
6	2	李超凡	11	男	闸北区...	NULL
7	1	魏伟	10	男	虹口区...	NULL
8	3	李天一	12	男	闸北区...	NULL
9	3	赵蕊	12	女	宝山区...	NULL

从例表中可知，系统中有3个班级，它们分别有4、3、2个学生。如果 SQLAlchemy 中没有把它们的 relationship 的 cascade 设置为 delete，则删除父表内容不会删除相应的子表内容，而是把子表的相应外键置为空。比如：

```
class_ = session.query(Class).filter(name="三年二班").first()      #三年二班的 class_id 为 1
session.delete(class_)                                              #删除 class_id 为 1 的班级
```

当 cascade 不包含 delete 时，上述代码中的 delete 语句相当于执行了如下 SQL 语句：

```
UPDATE student SET class_id=None WHERE class_id=1;
DELETE FROM class WHERE class_id=1;
COMMIT;
```

执行后数据库表 class 和 student 的内容变化如表 8.10 和 8.11 所示。

表 8.10 例表—class（没有 delete 级联时删除 class_id=1 班级后）

class_id	name	level	address
2	五年一班	5	李冰路410号3楼
3	五年二班	5	李冰路410号3楼

表 8.11 例表—student（没有 delete 级联时删除 class_id=1 班级后）

student_id	class_id	name	age	gender	address	contactor
1	NULL	李晓	10	男	虹口区...	NULL
2	NULL	单梦童	10	女	虹口区...	NULL
3	NULL	林一雷	9	女	闸北区...	林廷玉
4	2	丁辉	10	男	宝山区...	NULL
5	2	王文文	12	女	虹口区...	王飞
6	2	李超凡	11	男	闸北区...	NULL
7	NULL	魏伟	10	男	虹口区...	NULL
8	3	李天一	12	男	闸北区...	NULL
9	3	赵蕊	12	女	宝山区...	NULL

此时将表定义中的 relationship 的 cascade 属性设置为 delete：

```
students = relationship("Student", backref="class", cascade="delete")
```

现在通过如下语句删除“五年一班”：

```
class_ = session.query(Class).filter(name="五年一班").first()      #五年一班的 class_id 为 2
session.delete(class_)                                              #删除 class_id 为 2 的班级
```

当 cascade 包含“delete”时，上述代码中的 delete 语句相当于执行了如下 SQL 语句：

```
DELETE FROM student WHERE class=2;
DELETE FROM class WHERE class=2;
COMMIT;
```

执行后数据库表 class 和 student 的内容变化如表 8.12 和 8.13 所示。

表 8.12 例表—class（有 delete 级联时删除 class_id=2 班级后）

class_id	name	level	address
3	五年二班	5	李冰路410号3楼

表 8.13 例表—student（有 delete 级联时删除 class_id=2 班级后）

student_id	class_id	name	age	gender	address	contactor
1	NULL	李晓	10	男	虹口区...	NULL
2	NULL	单梦童	10	女	虹口区...	NULL
3	NULL	林一雷	9	女	闸北区...	林廷玉
7	NULL	魏伟	10	男	虹口区...	NULL
8	3	李天一	12	男	闸北区...	NULL
9	3	赵蕊	12	女	宝山区...	NULL

4. delete-orphan 级联

delete-orphan 级联是指当子对象不再与任何父对象关联时，会自动将该子对象删除。继续以表 8.12 和表 8.13 为例，设置父表与子表的 relationship 中的 cascade 包含“delete-orphan”：

```
students = relationship("Student", backref="class", cascade="delete-orphan")
```

通过如下代码将与班级“五年二班”关联的学生全部脱离：

```
class_ = session.query(Class).filter(name="五年二班").first()
unattachedStudent=[]
while len(class_.students)>0:
    unattachedStudent.append(class_.students.pop())          #与父对象脱离
session.commit                                         #显式地提交事务
```

上述代码中没有显式地删除任何学生，但由于使用了 delete-orphan 级联，所以被脱离出班级对象的学生会在 session 事务提交时被自动从数据库中删除。代码执行后数据库表中的内容变化如表 8.14 和表 8.15 所示。

表 8.14 例表—class (delete-orphan 级联)

class_id	name	level	address
3	五年二班	5	李冰路410号3楼

表 8.15 例表—student (delete-orphan 级联)

student_id	class_id	name	age	gender	address	contactor
1	NULL	李晓	10	男	虹口区...	NULL
2	NULL	单梦童	10	女	虹口区...	NULL
3	NULL	林一雷	9	女	闸北区...	林廷玉
7	NULL	魏伟	10	男	虹口区...	NULL

8.7 WTForm 表单编程

表单（Form）是用 HTML 互动式网站进行客户端与服务器交互的核心。随着网页内容的丰富，直接通过请求上下文获取客户端数据并进行解析，会使代码变得越来越杂乱无章。通过使用 WTForm 表单库，可以大大简化表单的处理流程，并使代码的结构和可读性更好。本节介绍如何在 Flask 中使用 WTForm 进行表单处理。WTForm 由如下几个概念组成。

- **Form 类：**所有开发者自定义的表单需要继承自 Form 类或其子类，Form 类的最主要功能是通过其所包含的 Field 类提供对表单内数据的快捷访问方式。
- 一系列 **Field**：即字段。WTForm 定义了若干个 Field 类型，每个 Field 类型对应一种 HTML input 标签控件。比如 BooleanField 用于显示和获取<input type="checkbox">标签数据；SubmitField 用于显示和获取<input type="submit">标签数据。
- **Validator：**验证器。用于验证用户在客户端输入的数据，当不符合要求时提醒用户重新输入。例如 Length 验证器用于指定文本输入必须满足的长度要求、FileAllowed 验证器用于指定可以上传的文件类型。

8.7.1 定义表单

可以通过定义一个继承自 Form 类的 Python 类来实现对表单类的定义，并且在其中定义一系列 Field 作为表单属性。比如下面是一个公告表单（BulletinForm）的定义：