# 关系数据库 MySQL 和 ORM

## 简介

Web开发中，离不开数据库的支持，在本节实验中我们将学习关系数据库 MySQL 和 ORM SQLAlchemy。[MySQL](https://www.mysql.com/" \t "https://www.shiyanlou.com/courses/996/labs/4186/_blank) 是应用最广泛的数据库，而 [SQLAlchemy](https://www.sqlalchemy.org/" \t "https://www.shiyanlou.com/courses/996/labs/4186/_blank) 是 Python 语言中最流行的 ORM。

### 知识点

* MySQL 基础知识
* 关系数据库基础
* SQLAlchemy 基础知识

## MySQL

MySQL 是应用最广泛的关系数据库。关系数据库使用关系模型作为数据组织存储的的方式，可以分为四级结构：

* 数据库，比如一个应用对应一个数据库；
* 表，一个数据库包含若干张表；
* 记录，一个表包含多条记录；
* 字段，一条记录包含若干字段；

一张表的结构可以想象成一张 Excel 表，有多个字段组成，每一个字段都可以存储特定类型的数据，比如字符串或者数字，在创建表的时可以指定类型。表与表之间通过关系连接（逻辑上的关系），查询数据时可以通过关系跨越多张表进行查询。

关系数据库最重要的特性是是满足 ACID 性质：

* A atomicity 代表原子性；
* C consistency 代表一致性；
* I isolation 代表隔离性；
* D durability 代表持久性；

ACID 能够保证事务的可靠性，什么意思呢？就是说能够保证一系列的数据库操作组成的一个完整逻辑过程，要么全部被执行，要么彻底不执行，不会出现执行一半的情况。例如银行转帐，从原账户扣除金额，以及向目标账户添加金额，这两个数据库操作的总和，构成一个完整的逻辑过程，不可拆分。

为了满足 ACID 性质，MySQL 支持各种约束，比如插入一条数据时，需要检查外键是否存在，这些操作虽然能确保数据的一致性，但是很多时候却降低了并发操作的能力，所以在如今的互联网网站中，如果有高并发的需求往往不再使用关系数据库的 ACID 性质，更有的直接使用非关系数据库。

在接下来的内容中，我们将学习基本的 MySQL 操作。

### 环境准备

实验楼环境中已经安装了 MySQL 数据库软件，每次启动实验后，需要手动启动 MySQL 服务。在实验桌面上启动终端后，通过以下命令启动数据库：

$ sudo service mysql start

还需要安装一个包

$ sudo pip3 install mysqlclient

这个包的作用是让 Python 可以连接 MySQL 数据库，如果不安装使用 Python 及 SQLAlchemy 连接 MySQL 数据库的时候则会出现下面的错误：

ImportError: No module named MySQLdb

或者

Module MySQLdb not found

这类的错误，原因就是在实验环境中没有安装 mysqlclient，或者你在使用 virtualenv（下面Python连接数据库部分会学习到） 的时候没有在虚拟环境中安装这个包（也有可能安装了，但没有重新激活 virtualenv 环境也会造成这个错误）。

如果这个错误出现在用 virtualenv（虚拟环境） 创建的虚拟环境里，需要在 virtualenv 中执行 pip3 install mysqlclient，然后执行 deactivate 退出虚拟环境再进到 virtualenv 就可以解决。

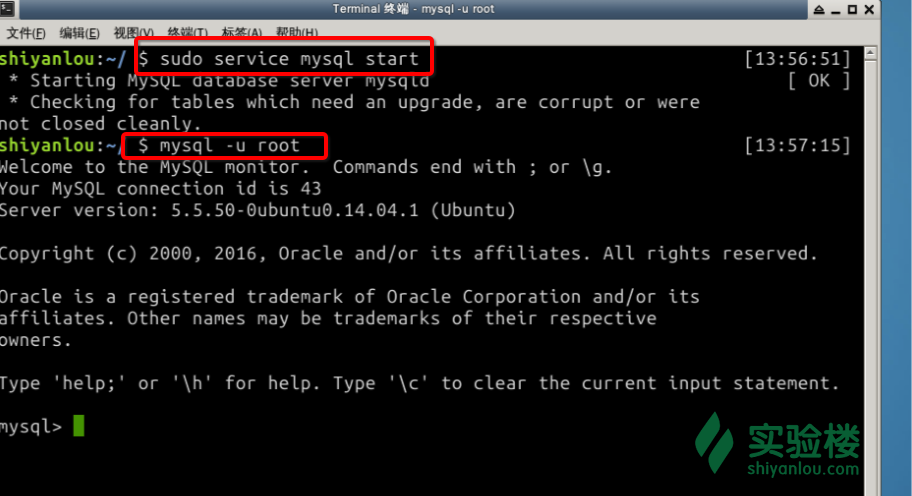
如果没用 virtualenv 创建虚拟环境，报这个错执行 sudo pip3 install mysqlclient 即可。

virtualenv（虚拟环境）的作用是避免 python 版本和软件包版本的使用冲突，例如系统环境里只安装了 python3.5 ，而某程序需要运行在 python2.7 上，或者不同程序用到软件包的不同版本，都可以用 virtualenv 来解决，在 virtualenv 中安装一些 Python 软件包，需要重新进入 virtualenv 环境才能生效。

数据库启动成功以后，通过以下命令链接到数据库：

$ mysql -u root

MySQL 数据库由服务器端和客户端组成，可以通过客户端连接到服务器。以上命令中，使用 root 账户登陆服务器，root 账户是 MySQL 数据库的超级管理员账户，未指定密码信息，这是因为实验楼环境中未设置 root 账户的密码。 登录成功后图如下图所示：



后面的所有 MySQL 命令都通过 MySQL 客户端输入(也就是以 mysql> 标识符开始的命令)。

### 基本操作

创建数据库非常简单，只需要输入 create database <db\_name>; 就可以了，其中 <db\_name> 代表数据库名称，查看所有的数据库可以通过 show databases; 命令，而删除一个数据库则可以通过 drop database <db\_name>; ，下面的例子演示了数据库和删除过程：

mysql> create database shiyanlou;

Query OK, 1 row affected (0.00 sec)

mysql> show databases;

+--------------------+

| Database |

+--------------------+

| information\_schema |

| mysql |

| performance\_schema |

| shiyanlou |

| sys |

+--------------------+

5 rows in set (0.00 sec)

mysql> drop database shiyanlou;

Query OK, 0 rows affected (0.00 sec)

mysql> show databases;

+--------------------+

| Database |

+--------------------+

| information\_schema |

| mysql |

| performance\_schema |

| sys |

+--------------------+

4 rows in set (0.00 sec)

需要注意的是 show databases; 的输出结果显示了多个数据库，这是因为 MySQL 服务器有几个默认的数据库，该命令输出结果在不同的 MySQL 版本下可能不一致。

数据库创建成功以后，就可以创建表了。在创建表之前，需要通过 use shiyanlou; 命令连接到 shiyanlou 数据库。

创建表的基本命令如下：

CREATE TABLE 表的名字

(

列名a 数据类型(数据长度),

列名b 数据类型(数据长度)，

列名c 数据类型(数据长度)

);

下面尝试创建一张表名为 user 的表，该表有 3 个字段：

* id 编号，整数类型，使用 int 类型；
* name 用户名，字符串，使用 varchar 可变字符类型；
* email 邮箱，字符串, 使用 varchar 可变字符型；

在 MySQL 客户端输入下面的命令：

mysql> create database shiyanlou;

mysql> use shiyanlou;

Database changed

mysql> create table user

-> (

-> id int(10),

-> name varchar(20),

-> email varchar(64)

-> );

Query OK, 0 rows affected (0.03 sec)

mysql> show tables;

+---------------------+

| Tables\_in\_shiyanlou |

+---------------------+

| user |

+---------------------+

1 row in set (0.00 sec)

通过以上命令就可以创建出 user 表，需要注意的是当输入 create table user 回车以后，客户端会自动识别出这是一个未完成的命令，所以会出现提示符 -> 。表创建成功以后，可以通过 show tables; 查看所有的表, 删除表可以通过 drop table <table\_name>; 。

如果想查看一张表的字段信息，可以通过 show create table <table\_name>; 或者 describe <table\_name>; 指令，如下所示：

mysql> show create table user;

| user | CREATE TABLE `user` (

`id` int(10) DEFAULT NULL,

`name` varchar(20) DEFAULT NULL,

`email` varchar(64) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8 |

1 row in set (0.00 sec)

mysql> describe user;

+-------+-------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+-------+-------------+------+-----+---------+-------+

| id | int(10) | YES | | NULL | |

| name | varchar(20) | YES | | NULL | |

| email | varchar(64) | YES | | NULL | |

+-------+-------------+------+-----+---------+-------+

3 rows in set (0.00 sec)

MySQL 表的字段支持多种类型，如整数，浮点数，字符串，时间戳等，篇幅有限就不一一介绍了。

表创建成功以后，就可以插入数据了。可以使用 insert 指令插入数据，完整的命令格式如下：

INSERT INTO 表的名字(列名a,列名b,列名c) VALUES(值1,值2,值3);

当插入的值和和表定义的字段数量一致且顺序一致时，可以忽略列名信息，通过以下命令往 user 表中插入数据：

mysql> insert into user(id, name, email) values(1, 'aiden', 'luojin@simplecloud.cn');

Query OK, 1 row affected (0.02 sec)

mysql> insert into user values(2, 'lxttx', 'lxttx@gmail.com');

Query OK, 1 row affected (0.00 sec)

成功的插入两条数据。查看某张表的数据可以通过 select \* from <table\_name>; 查看：

mysql> select \* from user;

+------+-------+-----------------------+

| id | name | email |

+------+-------+-----------------------+

| 1 | aiden | luojin@simplecloud.cn |

| 2 | lxttx | lxttx@gmail.com |

+------+-------+-----------------------+

2 rows in set (0.00 sec)

到这里我们已经学会了创建数据库和表，插入记录，查询记录等基本操作，是不是非常简单？

### 约束

MySQL 是一个关系数据库，可以通过约束限制一些数据操作。比如插入数据时，可以检查该数据是否满足关系约束，如果不满足则拒绝操作。下面通过一个简单的例子进行演示。

在前一小节中，我们创建了 user 表，且插入了下面的两条数据：

mysql> select \* from user;

+------+-------+-----------------------+

| id | name | email |

+------+-------+-----------------------+

| 1 | aiden | luojin@simplecloud.cn |

| 2 | lxttx | lxttx@gmail.com |

+------+-------+-----------------------+

现在重新再插入一条数据：

mysql> insert into user values(3, 'lxttx\_1', 'lxttx@gmail.com');

Query OK, 1 row affected (0.01 sec)

mysql> select \* from user;

+------+---------+-----------------------+

| id | name | email |

+------+---------+-----------------------+

| 1 | aiden | luojin@simplecloud.cn |

| 2 | lxttx | lxttx@gmail.com |

| 3 | lxttx\_1 | lxttx@gmail.com |

+------+---------+-----------------------+

3 rows in set (0.00 sec)

数据成功插入，但是现在有点问题，对于一个正式上线的数据库来说，一般不允许在一张用户表存在邮箱相同的用户，也就是说要求 email 字段具有唯一约束。这个时候怎么办呢？对于一个已经存在的表，可以通过 alter 指令修改表的字段，设置唯一约束：

mysql> alter table user modify email varchar(64) unique;

ERROR 1062 (23000): Duplicate entry 'lxttx@gmail.com' for key 'email'

mysql> delete from user where id = 3;

Query OK, 1 row affected (0.00 sec)

mysql> alter table user add constraint unique (email);

Query OK, 0 rows affected (0.03 sec)

Records: 0 Duplicates: 0 Warnings: 0

mysql> insert into user values(3, 'lxttx\_1', 'lxttx@gmail.com');

ERROR 1062 (23000): Duplicate entry 'lxttx@gmail.com' for key 'email'

以上命令中，先尝试修改 email 字段设置唯一约束，但是数据库报错，因为user 表中已存在重复的邮箱记录。接着使用 delete from user where id = 3; 删除了重复记录，该命令的含义是删除 id 为 3 的记录。接着成功了设置了 email 的唯一索引，然后插入重复记录时已被禁止。添加唯一约束还有一种方式为 alter table user modify email varchar(64) unique;，这种方式实际上是通过修改字段添加唯一索引。

还有一种约束，是跨表的。在表中插入数据项时，要求该数据项的某一个字段值必须已经在其他表中存在，比如外键约束。插入数据时，如果不满足外键约束则不允许插入，删除数据如果破坏了外键约束也会禁止删除数据。外键约束创建时，必须要求另一张表中存在主键，主键在表中能唯一的确定某一行的值。下面尝试创建一张具有外键约束的 course 表，在创建 course 表之前，需要在 user 表中设置，主键：

mysql> alter table user add constraint pk\_id primary key (id); Query OK, 0 rows affected (0.14 sec)

Records: 0 Duplicates: 0 Warnings: 0

mysql> describe user;

+-------+-------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+-------+-------------+------+-----+---------+-------+

| id | int(10) | NO | PRI | NULL | |

| name | varchar(20) | YES | UNI | NULL | |

| email | varchar(64) | YES | UNI | NULL | |

+-------+-------------+------+-----+---------+-------+

3 rows in set (0.00 sec)

通过 alter 指令成功在 user 中设置 id 字段为主键。通过 describe user; 命令输出确实发现 id 列变成了主键（因为 Key 列的值为 PRI, 代表 primary）。接着创建 course 表：

mysql> create table course

-> (

-> id int(10) auto\_increment,

-> name varchar(64),

-> teacher\_id int(10),

-> primary key (id),

-> constraint fk\_user foreign key (teacher\_id) references user(id)

-> );

Query OK, 0 rows affected (0.04 sec)

以上命令相对于之前建表命令增加了一些内容，首先通过 auto\_increment 指定了字段 id 为自增模式，这样每次插入数据时不用指定该字段，插入数据后这个字段会自动增加。接着通过 primary key (id) 语句指定了这个表的主键为 id, 最后使用设置了字段 teacher\_id 为外键，且关联到了 user 表的 id 字段。现在尝试往 course 表中插入一条 teacher\_id 在 user 表中不存在的数据：

mysql> insert into course(name, teacher\_id) values('Python基础', 100);

ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`shiyanlou`.`course`, CONSTRAINT `fk\_user` FOREIGN KEY (`teacher\_id`) REFERENCES `user` (`id`))

环境中输入中文并不方便，需要点击右下角的小键盘才可以切换输入法，如果觉得太麻烦可以直接输入英文即可，实验操作都是一致的，并不需要要求每个内容都完全相同，但需要理解每一个步骤和操作参数的作用。

可以看到，数据插入失败。因为 user 表中不存在 id 为 100 的数据，不满足外键约束。删除数据时，如果违反了外键约束也将删除失败：

mysql> select \* from user;

+----+-------+-----------------------+| id | name | email |

+----+-------+-----------------------+| 1 | aiden | luojin@simplecloud.cn || 2 | lxttx | lxttx@gmail.com |

+----+-------+-----------------------+2 rows in set (0.00 sec)

mysql> insert into course(name, teacher\_id) values('Python 基础', 2);

Query OK, 1 row affected (0.00 sec)

mysql> select \* from course;

+----+---------------+------------+| id | name | teacher\_id |

+----+---------------+------------+| 2 | Python 基础 | 2 |

+----+---------------+------------+1 row in set (0.00 sec)

mysql> delete from user where id = 2;

ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`shiyanlou`.`course`, CONSTRAINT `fk\_user` FOREIGN KEY (`teacher\_id`) REFERENCES `user` (`id`))

上面的例子中，为了演示删除失败的情况，先往 course 表中插入了一条数据，接着尝试从 user 表中删除 id 为 2 的数据，发现删除失败，这是因为如果删除该记录就违背了已存在的外键约束。

很多时候需要进行联合跨表查询，比如想知道课程的名称，课程老师的名称，邮箱信息就需要联合查询：

mysql> select \* from course join user on course.teacher\_id = user.id;

+----+---------------+------------+----+-------+-----------------+| id | name | teacher\_id | id | name | email |

+----+---------------+------------+----+-------+-----------------+

| 2 | Python 基础 | 2 | 2 | lxttx | lxttx@gmail.com |

+----+---------------+------------+----+-------+-----------------+1 row in set (0.00 sec)

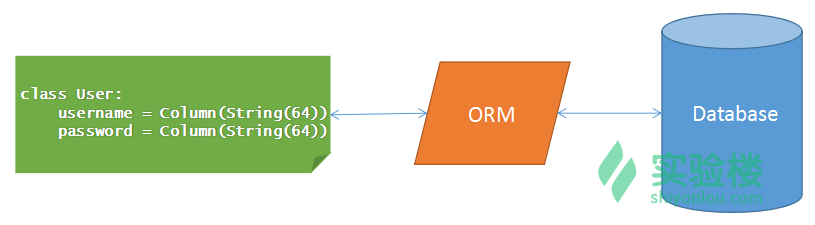
例子中使用 join 指令进行联合查询，on 关键字指定了两张表的关联方式。

### 总结

学习完以上内容，我们已经能够进行 MySQL 的基本操作了。MySQL 博大精深，涉及的知识点非常多，比如上面的内容就没有涉及索引管理和性能优化方面的内容。如果想学习更多的 MySQL 内容，一方面可以在实验楼学习其他课程，另一方面要多查阅 [MySQL 文档](https://dev.mysql.com/doc/" \t "https://www.shiyanlou.com/courses/996/labs/4186/_blank)。

## SQLAlchemy

在实际项目中，一般都不直接写 SQL 语句访问数据库，而是通过 ORM 工具。ORM 全称是对象关系映射（Object Relational Mapping），有了 ORM 以后可以将 Python 对象映射到数据库中，这样就不用再编写各种 SQL 语句了。而在 Python 语言中，SQLAlchemy 是非常强大的 ORM 包，非常值得学习。同时 SQLAlchemy 支持多种关系数据，如果项目后期需要切换到其他类型的数据库，通过 SQLAlchemy 也比较容易。



### 环境准备

本节实验环境主要基于 virtualenv 建立。virtualenv 的作用很重要，可以理解为一个隔离的虚拟环境，安装在 virtualenv 下的包不会影响整个系统的包，避免不同版本包之间的影响，不需要太深入只需要知道如何创建及激活、退出（deactivate命令）就可以。

使用 virtualenv 的时候需要注意，当在 virtualenv 下安装了包之后，需要先用 deactivate 命令退出 virtualenv 后再重新激活 virtualenv 才可以用这个包。

打开桌面上的终端后，依次输入下面的命令建立学习环境($ 是 shell 提示符，之后的内容才是真正需要输入的命令)：

$ cd ~/Code

$ sudo pip install virtualenv

$ virtualenv -p /usr/bin/python3.5 env

$ source env/bin/activate

$ pip install sqlalchemy ipython mysqlclient

$ deactivate

以上命令在 Code 目录创建一个 virtualenv 环境，接着在这个虚拟环境中安装了本节实验需要的软件包，比如 sqlalchemy 和 ipython，然后使用 deactivate 退出 env。后续的交互式命令都通过 IPython 终端输入。 可以通过以下命令启动 IPython 终端：

$ source env/bin/activate

$ ipython

IPython是一个交互式计算系统，类似于我们之前使用的 Python 3 交互式环境，但是一个具备更多便利功能的增加的交互式 “Python shell”。主要优点包括：

1. 可以非常方便获取代码中对象的各种信息
2. 可以使用叹号直接在其中调用 Linux 的命令
3. 支持 TAB 自动补全代码
4. 输入的代码都保存到历史记录数据库里

因此，如果你对 IPython 熟悉的话，会逐步替代先前使用的 Python Shell。

后文出现的代码中， In [1] 类似样式的字符是 IPython 的提示符，不需要输入。

后续的 SQLAlchemy 基本操作都提供了详细的操作代码和预期的结果，不涉及到难以理解的操作内容，操作过程中需要细心理解 sqlalchemy 模块中每个类及函数的作用。

注意：在 ipython 中连续键入两个回车换行就会进入一个新的输入，这个时候如果你的类还没有写完，那么不要连续键入两次回车。

### 连接数据库

使用 SQLAlchemy 连接数据库主要通过 Engine 对象进行，在 IPython 终端中输入以下代码：

In [1]: from sqlalchemy import create\_engine

In [2]: engine = create\_engine('mysql://root:@localhost/shiyanlou')

In [3]: engine.execute('select \* from user').fetchall()

Out[3]: [(1, 'aiden', 'luojin@simplecloud.cn'), (2, 'lxttx', 'lxttx@gmail.com')]

首先导入了 create\_engine, 该方法用于创建 Engine 实例，传递给 create\_engine 的参数定义了 MySQL 服务器的访问地址，其格式为 mysql://<user>:<password>@<host>/<db\_name>，例子中访问的正是上文中创建的 shiyanlou 数据库。

接着通过 engine.execute 方法执行了一条 SQL 语句，查询了 user 表中的所有用户，非常容易。

### 对象关系映射

如果想使 Python 类映射到数据库表中，需要基于 SQLAlchemy 的 declarative base class，也就是宣言基类创建类。当基于此基类，创建 Python 类时，就会自动映射到相应的数据库表上。创建宣言基类，可以通过 declarative\_base 方法进行，如下代码：

In [12]: from sqlalchemy.ext.declarative import declarative\_base

In [13]: Base = declarative\_base()

创建基类以后，在 IPython 终端中输入下面代码，创建 User 类，该类将映射到上文中创建的 user 表：

In [19]: from sqlalchemy import Column, Integer, String

In [20]: class User(Base):

...: \_\_tablename\_\_ = 'user'

...: id = Column(Integer, primary\_key=True)

...: name = Column(String)

...: email = Column(String)

...: def \_\_repr\_\_(self):

...: return "<User(name=%s)>" % self.name

...:

In [21]:

以上代码执行以后就成功定义了 User 类，注意 \_\_repr\_\_ 前后各有两个下划线，这种前后有两个下划线的函数表示一个特殊的函数，称为 Python 类的魔法方法，\_\_init\_\_ 也是一个魔法方法，这里 \_\_repr\_\_ 方法会在直接调用实例对象的时候被调用。

此时 User 有一个 \_\_table\_\_ 属性，记录了定义的表信息，该属性如下所示：

In [27]: User.\_\_table\_\_

Out[27]: Table('user', MetaData(bind=None), Column('id', Integer(), table=<user>, primary\_key=True, nullable=False), Column('name', String(), table=<user>), Column('email', String(), table=<user>), schema=None)

如果想通过 User 查询数据库该怎么办呢？需要先引入 Session。Session 是映射类和数据库沟通的桥梁，包含事务管理功能。通过以下代码创建 Session:

In [30]: from sqlalchemy.orm import sessionmaker

In [31]: Session = sessionmaker(bind=engine)

In [32]: session = Session()

这里的代码先从 sqlalchemy.orm 中导入 sessionmaker，然后创建了 sessionmaker 对象 Session，其中 Session 对象中有一个魔法方法（\_\_call\_\_），这个魔法方法让 Session 对象可以像函数那样调用，从而使用 Session() 获得了 session 对象。

Session 创建成功以后，就可以查询用户了，主要通过 session.query 方法：

In [63]: session.query(User).all()

Out[63]: [<User(name=aiden)>, <User(name=lxttx)>]

In [65]: session.query(User).filter(User.name=='aiden').first()

Out[65]: <User(name=aiden)>

可以看到查询成功，而且可以直接使用 User 类的字段进行过滤查询。

如果在上文中创建 Engine 时，通过 echo 参数开启了显示 SQL 语句（engine = create\_engine('mysql://root:@localhost/shiyanlou', echo=True)），则使用 session 查询时，可以看到相应的 SQL 输出：

In [69]: session.query(User).filter(User.name=='aiden').first()

2017-08-30 18:02:37,878 INFO sqlalchemy.engine.base.Engine SHOW VARIABLES LIKE 'sql\_mode'2017-08-30 18:02:37,878 INFO sqlalchemy.engine.base.Engine ()2017-08-30 18:02:37,883 INFO sqlalchemy.engine.base.Engine SELECT DATABASE()2017-08-30 18:02:37,883 INFO sqlalchemy.engine.base.Engine ()2017-08-30 18:02:37,885 INFO sqlalchemy.engine.base.Engine show collation where `Charset` = 'utf8' and `Collation` = 'utf8\_bin'2017-08-30 18:02:37,885 INFO sqlalchemy.engine.base.Engine ()2017-08-30 18:02:37,889 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain returns' AS CHAR(60)) AS anon\_12017-08-30 18:02:37,889 INFO sqlalchemy.engine.base.Engine ()2017-08-30 18:02:37,893 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode returns' AS CHAR(60)) AS anon\_12017-08-30 18:02:37,893 INFO sqlalchemy.engine.base.Engine ()2017-08-30 18:02:37,895 INFO sqlalchemy.engine.base.Engine SELECT CAST('test collated returns' AS CHAR CHARACTER SET utf8) COLLATE utf8\_bin AS anon\_12017-08-30 18:02:37,895 INFO sqlalchemy.engine.base.Engine ()2017-08-30 18:02:37,897 INFO sqlalchemy.engine.base.Engine BEGIN (implicit)2017-08-30 18:02:37,898 INFO sqlalchemy.engine.base.Engine SELECT user.id AS user\_id, user.name AS user\_name, user.email AS user\_emailFROM userWHERE user.name = %s

LIMIT %s2017-08-30 18:02:37,899 INFO sqlalchemy.engine.base.Engine ('aiden', 1)Out[69]: <User(name=aiden)>

### 创建数据库表

可以基于 SQLAlchemy 定义的类中生成数据库表，下面尝试创建一个新的实验表 lab，一个课程将对应于多个实验。所以课程表 course 和 lab 表是 1:M 一对多的关系。在定义 Lab 类之前，需要先将创建 Course 类，使其映射到之前定义的 course 表：

In [80]: from sqlalchemy.orm import relationship

In [81]: from sqlalchemy import ForeignKey

In [84]: class Course(Base):

...: \_\_tablename\_\_ = 'course'

...: id = Column(Integer, primary\_key=True)

...: name = Column(String)

...: teacher\_id = Column(Integer, ForeignKey('user.id'))

...: teacher = relationship('User')

...: def \_\_repr\_\_(self):

...: return '<Course(name=%s)>' % self.name

...:

上面的代码引入了一些新的东西。前文中创建的 course 表有外键 teacher\_id, 在 SQLAlchemy 中可以使用 ForeignKey 设置外键。设置外键后，如果能够直接从 Course 的实例上访问到相应的 user 表中的记录会非常方便，而这可以通过 relationship 实现。上面的代码通过 relationship 定义了 teacher 属性，这样就可以直接通过 course.teacher 获取相应的用户记录。Course 类定义后，接着定义 Lab 类：

In [89]: class Lab(Base):

...: \_\_tablename\_\_ = 'lab'

...: id = Column(Integer, primary\_key=True)

...: name = Column(String(64))

...: course\_id = Column(Integer, ForeignKey('course.id'))

...: course = relationship('Course', backref='labs')

...: def \_\_repr\_\_(self):

...: return '<Lab(name=%s)>' % self.name

...:

以上代码定义了 Lab 类，需要注意的地方是定义 course 属性时，使用了 relationship 的 backref 参数，该参数使得可以在 Course 实例中，通过 course.labs 访问关联的所有实验记录。

定义 Lab 类以后，就可以通过以下命令在 MySQL 中创建相应的 lab 表：

In [90]: Base.metadata.create\_all(engine)

然后通过 MySQL 客户可以看到，相应的表已经创建成功：

mysql> show tables;

+---------------------+

| Tables\_in\_shiyanlou |

+---------------------+

| course |

| lab |

| user |

+---------------------+

3 rows in set (0.00 sec)

mysql> describe lab;

+-----------+-------------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+-----------+-------------+------+-----+---------+----------------+

| id | int(11) | NO | PRI | NULL | auto\_increment |

| name | varchar(64) | YES | | NULL | |

| course\_id | int(11) | YES | MUL | NULL | |

+-----------+-------------+------+-----+---------+----------------+

3 rows in set (0.00 sec)

### 简单的 CRUD 操作

基于前面创建的 User, Course 和 Lab 类，我们进一步学习创建，查询，更新，删除操作。

有了 ORM 映射以后了，创建数据库记录就非常简单，比如想创建一个实验，该实验关联到前文插入的课程上，只需要创建 Lab 的实例就可以了：

In [37]: course = session.query(Course).first()

In [38]: lab1 = Lab(name='ORM 基础', course\_id=course.id)

In [39]: lab2 = Lab(name='关系数据库', course=course)

In [41]: session.add(lab1)

In [42]: session.add(lab2)

In [43]: session.commit()

In [44]: course.labs

Out[44]: [<Lab(name=关系数据库)>, <Lab(name=ORM 基础)>]

上面的代码中，首先查询出需要管理的课程对象，接着创建了两个实验，并通过 session.commit() 操作提交到了数据库中，在将数据变更提交到数据库中前，需要将数据通过 session.add 方法添加到 session 中。还可以看到，创建实验时，关联到课程有两种办法，一种是直接赋值给 course\_id, 另外一种办法是赋值给定义的关系属性 course。当数据成功插入数据库后，就可以通过 course.labs 获取这两个实验。

MySQL 客户端中可以查询出 lab 表中成功的插入了两条数据：

mysql> select \* from lab;

+----+-----------------+-----------+

| id | name | course\_id |

+----+-----------------+-----------+

| 2 | 关系数据库 | 2 |

| 3 | ORM 基础 | 2 |

+----+-----------------+-----------+

2 rows in set (0.00 sec)

更新操作也非常简单，只需要更新对象的属性，然后通过 session.commit（） 提交到数据库即可，下面的代码演示了如何更新课程名称：

In [58]: course.name

Out[58]: 'Python 基础'

In [59]: lab1.course

Out[59]: <Course(name=Python 基础)>

In [60]: course.name = 'Python 数据分析'

In [61]: session.add(course)

In [62]: session.commit()

In [63]: lab1.course

Out[63]: <Course(name=Python 数据分析)>

In [64]: session.query(Course).all()

Out[64]: [<Course(name=Python 数据分析)>]

删除数据时通过 session.delete 方法删除相应的对象即可：

In [65]: session.delete(lab1)

In [66]: session.commit()

In [67]: course.labsOut[67]: [<Lab(name=关系数据库)>]

删除记录以后，通过课程实例获取到的实验数量同样减少。

可以看到有了 SQLAlchemy 以后，对于数据库的 CRUD 操作全部变成了相应的 Python 对象操作，非常方便。

### 1:1 和 M:M 的关系建立

在关系数据库中，关系类型有三种：

* 1:1，两张表中的数据项是一对一的关系，假设用户只有一项附属信息，这个时候用户表和附属信息表就是一对一的关系；
* 1:M，比如上文中课程和实验的关系，一个课程对应多个实验；
* M:M，多对多的关系，比如课程和标签的关系，一个课程有多个标签，而一个标签可以关联到多个课程；

我们已经知道 1:M 的关系创建方法，主要通过外键进行。其实 1:1 的关系也可以通过外键进行创建。通过以下代码创建 UserInfo 用户附属信息表：

In [86]: class UserInfo(Base):

...: \_\_tablename\_\_ = 'userinfo'

...: user\_id = Column(Integer, ForeignKey('user.id'), primary\_key=True)

...: addr = Column(String(512))

...:

In [87]: Base.metadata.create\_all(engine)

以上代码中，成功创建了 UserInfo 表，它和 User 表的关系就是 1:1 的关系，因为 UserInfo 表的主键和外键都是 user\_id, 且依赖于 User 表的主键。

对于 M:M 多对多的关系，可以由两个 1:M 的关系构造出来。如果两张表对于同一张表（假如表名是 T）都是 1:M 的关系，那么就可以把 T 当做中间表，创建出两张表的多对多关系。比如课程表和标签表的关系，可以由以下代码创建：

In [90]: from sqlalchemy import Table, Text

In [91]: course\_tag = Table('course\_tag', Base.metadata,

...: Column('course\_id', ForeignKey('course.id'), primary\_key=True),

...: Column('tag\_id', ForeignKey('tag.id'), primary\_key=True)

...: )

In [92]: class Tag(Base):

...: \_\_tablename\_\_ = 'tag'

...: id = Column(Integer, primary\_key=True)

...: name = Column(String(64))

...: courses = relationship('Course',

...: secondary=course\_tag,

...: backref='tags')

...: def \_\_repr\_\_(self):

...: return '<Tag(name=%s)>' % self.name

In [93]: Base.metadata.create\_all(engine)

1. Base.metadata 是 sqlalchemy.schema.MetaData 对象,表示所有 Table 对象集合, create\_all() 会触发 CREATE TABLE 语句创建所有的表
2. course\_tag 是双主键，双主键的目的就是为了约束避免出现重复的一对主键记录，大部分情况都是应用在这种多对多的中间表中
3. secondary 指的是中间表，backref 指向自己的这个表，我也不清楚为什么这个参数叫secondary
4. session 内部的实现都是调用 engine 的各种接口，相当于 session 是 engine 的一个封装，比如 session.commit 的时候会先调用 engine.connect() 去连接数据库，再调用执行 sql 相关的接口。

上面的代码中，首先通过 Table 类创建了一张中间表 course\_tag，因为这张表不需要映射到任何 Python 对象上，所以直接创建了表。 可以看到 Course， Tag 表和 course\_tag 表都是 1:M 的关系。通过 course\_tag 这张中间表成功的建立了 M:M 的关系。还需要注意的是，Tag 表中定义了 courses 属性，该属性通过 relationship 函数的 secondary 参数告诉 SQLAlchemy 通过表 course\_tag 关联到 Course 表的对象上。下面演示下多对多关系的 CRUD 操作：

In [126]: session.close()

In [130]: course = session.query(Course).first()

In [131]: course.tags

Out[131]: []

In [132]: tag1 = Tag(name='tag\_1')

In [133]: tag2 = Tag(name='tag\_2')

In [136]: course.tags.append(tag1)

In [137]: course.tags.append(tag2)

In [138]: session.add(course)

In [139]: session.commit()

In [140]: course.tags

Out[140]: [<Tag(name=tag\_1)>, <Tag(name=tag\_2)>]

In [141]: engine.execute('select \* from tag').fetchall()

Out[141]: [(1, 'tag\_1'), (2, 'tag\_2')]

In [142]: engine.execute('select \* from course\_tag').fetchall()

Out[142]: [(2, 1), (2, 2)]

上面的代码中，通过 course.tags.append(tag1) 向课程中添加标签，就像操作 Python 列表一样方便，最后还通过 engine.execute 执行 SQL 查询验证了标签确实插入了数据库中，而且 course\_tag 表中也生成了相应的关系记录。下面尝试向 Tag对象中添加课程：

In [153]: teacher = session.query(User).filter(User.name=='aiden').first()

In [154]: course1 = Course(name='Linux 基础', teacher=teacher)

In [155]: session.add(course1)

In [156]: session.commit()

In [157]: tag1.courses

Out[157]: [<Course(name=Python 数据分析)>]

In [159]: tag1.courses.append(course1)

In [160]: session.add(tag1)

In [161]: session.commit()

In [163]: tag1.courses

Out[163]: [<Course(name=Python 数据分析)>, <Course(name=Linux 基础)>]

In [164]: engine.execute('select \* from course\_tag').fetchall()

Out[164]: [(2, 1), (3, 1), (2, 2)]

首先创建了另外一门课程 course1, 然后成功的将 course1 添加到了标签 tag1 中。

相信通过以上例子，你已经基本学会了 SQLAlchemy 中创建各种关系的方法。

# NoSQL 数据库

## 简介

如今的网站对数据存储要求越来越灵活，在这种需求下 NoSQL 也就是非关系数据库越来越流行。所谓非关系数据库，是指不使用 SQL 语言进行数据操作的数据库的统称。这类数据库存储数据时没有固定的模式，不支持数据表 join 的操作，可以很方便的进行横向扩展。非关系数据库种类很多，其中 MongoDB 和 Redis 应用广泛。在本节实验中，我们将学习 MongoDB 和 Redis 的基本操作，以及怎么样使用 Python 代码访问这些数据库。

### 知识点

* MongoDB 的基础操作；
* Redis 基础操作；
* 使用 Python 访问 MongoDB 和 Redis；

## MongoDB

MongoDB 是非常流行的 NoSQL 数据库，支持自动化的水平扩展，同时也被称为文档数据库，因为数据按文档的形式进行存储（BSON 对象，类似于 JSON）。在 MongoDB 中数据存储的组织方式组要分为四级：

* 数据库实例，比如一个 app 使用一个数据库；
* collection 文档集合 ，一个数据库包含多个文档集合，类似于 MySQL 中的表；
* document 文档，一个文档代表一项数据，类似于 JSON 对象，对应于MySQL 表中的一条记录；
* 字段：一个文档包含多个字段；

MongoDB 存储的数据可以是无模式的，比如在一个集合中的所有文档不需要有一致的结构。也就是说往同一个表中插入不同的数据时，这些数据之间不必有同样的字段。这和 MySQL 彻底不同，在 MySQL 中创建表时就已经确定了数据项的字段，往其中插入数据时，必须是相同的结构。

本节实验基于 MongoDB 3.4 版本讲解。

### 环境准备

实验楼环境中已经安装了 MongoDB，每次启动实验后，需要启动 MongoDB 服务。在实验桌面上启动终端后，通过以下命令启动数据库：

$ sudo service mongod start

数据库启动成功以后，通过以下命令链接到数据库：

$ mongo

mongo 是 MongoDB 的客户端 Shell，执行该命令时可以指定连接的 MongoDB 地址等信息，未指定时将连接默认地址，默认情况下 MongoDB 服务会监听在 127.0.0.1:27017 地址。

后文出现的所有 MongoDB 操作命令都将基于 mongo shell 输入。

对于使用 Python 访问 MongoDB，需要先安装 PyMongo 软件包，该包实现了 Python 的 MongoDB 驱动。通过以下命令建立工作环境，安装软件包：

$ cd ~/Code

$ sudo pip install virtualenv

$ virtualenv -p /usr/bin/python3.5 env

$ source env/bin/activate

$ pip install pymongo ipython

注意这里使用 virtualenv 环境，并且指定了使用 Python 3.5 的版本，然后在其中安装所需的软件包可以使用 pip，pip 默认就是安装的 Python 3.5 对应的包。

以上命令在 Code 目录创建一个 virtualenv 环境，接着在这个虚拟环境中安装了本节实验需要的软件包。后续的交互式命令都通过 IPython 终端输入。 可以通过以下命令启动 IPython 终端：

$ source env/bin/activate

$ ipython

后文出现的代码中， In [1] 类似样式的字符是 IPython 的提示符，不需要输入。

### CRUD 操作

MongoDB 存储的文档记录是一个 BSON 对象，类似于 JSON 对象，由键值对组成。比如一条用户记录：

{

name: "Aiden",

age: 30,

email: "luojin@simplecloud.cn"

}

每一个文档都有一个 \_id 字段，该字段是主键，用于唯一的确定一条记录。如果往 MongoDB 中插入数据时没有指定 \_id 字段，那么会自动产生一个 \_id 字段，该字段的类型是 [ObjectId](https://docs.mongodb.com/manual/reference/bson-types/" \l "objectid" \t "https://www.shiyanlou.com/courses/996/labs/4188/_blank)，长度是 12 个字节。在 MongoDB 文档的字段支持字符串，数字，时间戳等类型。一个文档最大可以达到 16M, 可以存储相当多的数据。

先尝试往 MongoDB 中插入一条数据：

$ mongo

> use shiyanlou

> db.user.insertOne({name: "Aiden", age: 30, email: "luojin@simplecloud.cn", addr: ["CD", "SH"]})

{

"acknowledged" : true,

"insertedId" : ObjectId("59a8034064e0acb13483d512")

}

> show databases;

admin 0.000GBlocal 0.000GB

shiyanlou 0.000GB

> show collections;

user

可以看到，在插入数据前使用 use 指令，切换到了 shiyanlou 数据库，尽管该数据库暂时不存在，但当我们插入数据后，该数据库就被自动创建了。show databases 和 show collection 分别显示了当前存在的数据库和当前数据库的所有文档集合。而且数据插入后，自动添加了 \_id 字段。插入多条数据，可以使用 db.collection.insertMany 方法：

> db.user.insertMany([... {name: "lxttx", age: 28, email: "lxttx@simplecloud.cn", addr: ["BJ", "CD"]},... {name: "jin", age: 31, email: "jin@simplecloud.cn", addr: ["GZ", "SZ"]},... {name: "nan", age: 26, email: "nan@simplecloud.cn", addr: ["NJ", "AH"]}... ])

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("59a8034564e0acb13483d513"),

ObjectId("59a8034564e0acb13483d514"),

ObjectId("59a8034564e0acb13483d515")

]

}

查询数据可以使用 db.collection.find 方法，可以指定查询过滤条件：

> db.user.find()

{ "\_id" : ObjectId("59a8034064e0acb13483d512"), "name" : "Aiden", "age" : 30, "email" : "luojin@simplecloud.cn", "addr" : [ "CD", "SH" ] }

{ "\_id" : ObjectId("59a8034564e0acb13483d513"), "name" : "lxttx", "age" : 28, "email" : "lxttx@simplecloud.cn", "addr" : [ "BJ", "CD" ] }

{ "\_id" : ObjectId("59a8034564e0acb13483d514"), "name" : "jin", "age" : 31, "email" : "jin@simplecloud.cn", "addr" : [ "GZ", "SZ" ] }

{ "\_id" : ObjectId("59a8034564e0acb13483d515"), "name" : "nan", "age" : 26, "email" : "nan@simplecloud.cn", "addr" : [ "NJ", "AH" ] }

> db.user.find({name: "jin"})

{ "\_id" : ObjectId("59a8034564e0acb13483d514"), "name" : "jin", "age" : 31, "email" : "jin@simplecloud.cn", "addr" : [ "GZ", "SZ" ] }

> db.user.find({age: {$gt: 30}})

{ "\_id" : ObjectId("59a8034564e0acb13483d514"), "name" : "jin", "age" : 31, "email" : "jin@simplecloud.cn", "addr" : [ "GZ", "SZ" ] }

> db.user.find({addr: "CD"})

{ "\_id" : ObjectId("59a8034064e0acb13483d512"), "name" : "Aiden", "age" : 30, "email" : "luojin@simplecloud.cn", "addr" : [ "CD", "SH" ] }

{ "\_id" : ObjectId("59a8034564e0acb13483d513"), "name" : "lxttx", "age" : 28, "email" : "lxttx@simplecloud.cn", "addr" : [ "BJ", "CD" ] }

上面例子中，我们先通过 db.user.find() 获取到了之前插入的全部数据。接着使用不同的过滤条件进行了查询，其中有一些查询如 {age: {$gt: 30}} 表示查询年龄大于 30 的用户。还可以发现查询数组中是否存在某一元素也非常方便，上面的例子中查询出了所有地址含有 CD 用户。

MongoDB 的查询功能非常强大，可以组合各种查询条件，更多的使用方法可以学习实验楼的其他课程。更新数据主要通过 db.user.updateOne 或者 db.user.updateMany 方法，前者更新一条记录，后者更新多条记录：

> db.user.updateOne(... {name: "Aiden"},... {$set: {age: 29, addr: ["CD", "SH", "BJ"]}}... )

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.user.find({name: "Aiden"})

{ "\_id" : ObjectId("59a8034064e0acb13483d512"), "name" : "Aiden", "age" : 29, "email" : "luojin@simplecloud.cn", "addr" : [ "CD", "SH", "BJ" ] }

可以看到成功的更新了一条记录。删除数据也非常简单，可以通过 db.user.deleteMany 或db.user.deleteOne 方法：

> db.user.deleteMany({addr: "CD"})

{ "acknowledged" : true, "deletedCount" : 2 }

> db.user.find()

{ "\_id" : ObjectId("59a8034564e0acb13483d514"), "user" : "jin", "age" : 31, "email" : "jin@simplecloud.cn", "addr" : [ "GZ", "SZ" ] }

{ "\_id" : ObjectId("59a8034564e0acb13483d515"), "user" : "nan", "age" : 26, "email" : "nan@simplecloud.cn", "addr" : [ "NJ", "AH" ] }

上面的命令成功的删除所有地址包含 "CD" 的用户，共删除了两条记录。

### Python 操作 MongoDB

在 Python 中访问 MongoDB 数据库，主要通过 [PyMongo](https://api.mongodb.com/python/current/tutorial.html" \t "https://www.shiyanlou.com/courses/996/labs/4188/_blank) 软件包。该软件包含一个 MongoClient 对象，可以用于建立 MongoDB 客户端。在 IPython 中输入下面的示例代码，创建客户端：

In [2]: from pymongo import MongoClient

In [3]: client = MongoClient('127.0.0.1', 27017)

In [4]: db = client.shiyanlou

前文中，已经知道 MongoDB 默认监听在 127.0.0.1:27017 地址上，所以在创建 client 时，使用了该地址。客户端创建成功后，我们通过 client.shiyanlou 方式选择了 shiyanlou 数据库。接着就可以查询所有的文档了：

In [13]: for user in db.user.find():

...: print(user)

...:

{'\_id': ObjectId('59a8034564e0acb13483d514'), 'user': 'jin', 'age': 31.0, 'email': 'jin@simplecloud.cn', 'addr': ['GZ', 'SZ']}

{'\_id': ObjectId('59a8034564e0acb13483d515'), 'user': 'nan', 'age': 26.0, 'email': 'nan@simplecloud.cn', 'addr': ['NJ', 'AH']}

通过 PyMongo 插入数据也非常简单，直接通过 insert\_one 方法：

In [14]: user = {'name': 'Aiden', 'age': 30, 'addr': ['CD', 'SH', 'BJ']}

In [15]: db.user.insert\_one(user)

Out[15]: <pymongo.results.InsertOneResult at 0x10730aa08>

In [17]: db.user.find\_one({'name': 'Aiden'})

Out[17]:

{'\_id': ObjectId('59a80988a75acb3615913dc6'),

'addr': ['CD', 'SH', 'BJ'],

'age': 30,

'name': 'Aiden'}

数据插入以后，我们使用 find\_one 方法，查询了该记录，查询方法几乎和 mongo shell 查询方法相同。查询发现，没有设置 email 字段，可以通过 update\_one 方法更新记录：

In [19]: db.user.update\_one({'name': 'Aiden'}, {'$set': {'email': 'aiden@simplecloud.cn'}})

Out[19]: <pymongo.results.UpdateResult at 0x1070dce08>

In [20]: db.user.find\_one({'name': 'Aiden'})

Out[20]:

{'\_id': ObjectId('59a80988a75acb3615913dc6'),

'addr': ['CD', 'SH', 'BJ'],

'age': 30,

'email': 'aiden@simplecloud.cn',

'name': 'Aiden'}

可以发现 PyMongo 的很多操作类似于 mongo shell 的操作，比较简单。

## Redis

Redis 一个内存数据库，通过 Key-Value 键值对的的方式存储数据。由于 Redis 的数据都存储在内存中，所以访问速度非常快，因此 Redis 大量用于缓存系统，存储热点数据，可以极大的提高网站的响应速度。相对于其他内存数据库，Redis 具有以下几个优点：

* 支持数据的持久化，通过配置可以将内存中的数据保存在磁盘中，Redis 重启以后再将数据加载到内存中；
* 支持列表，哈希，有序集合等数据结构，极大的扩展了 Redis 用途；
* 原子操作，Redis 的所有操作都是原子性的，这使得基于 Redis 实现分布式锁非常简单；
* 支持发布/订阅功能，数据过期功能；

### 环境准备

实验楼环境中已经安装了Redis，每次启动实验后，需要手动启动 Redis 。在实验桌面上启动终端后，通过以下命令启动数据库：

$ sudo service redis-server start

数据库启动成功以后，通过以下命令链接到数据库：

$ redis-cli127.0.0.1:6379>

redis-cli 是 Redis 的客户端 Shell，执行该命令时可以指定连接的 Redis 服务地址等信息，未指定时将连接默认地址，Redis 服务默认监听在 127.0.0.1:6379 地址。后文出现的所有 Redis 操作命令都将基于 redis-cli 输入。

对于使用 Python 访问 Redis，我们需要先安装 redis-py 软件包，该包实现了 Python 的 Redis 驱动。通过以下命令建立工作环境，安装软件包：

$ cd ~/Code

$ sudo pip install virtualenv

$ virtualenv -p /usr/bin/python3.5 env

$ source env/bin/activate

$ pip install redis ipython

以上命令在 Code 目录创建一个 virtualenv 环境，接着在这个虚拟环境中安装了本节实验需要的软件包。后续的交互式命令都通过 IPython 终端输入。 可以通过以下命令启动 IPython 终端：

$ source env/bin/activate

$ ipython

后文出现的代码中， In [1] 类似样式的字符是 IPython 的提示符，不需要输入。

### 基本操作

Redis 是 Key-Value 内存数据库，操作是通过各种指令进行的，比如 SET 指令可以设置键值对，而 GET 指令则获取某一个键的值。不同的数据结构，Redis 有不同的指令，这样指令一共有几十个，下面主要介绍一些常用的指令。

Redis 对 Key 也就是键有各种各样的指令，主要有下面的指令（下面的指令中小写字符串都是参数，可以自定义）：

* SET key value 设置键值；
* EXISTS key 判断键是否存在；
* EXPIRE key seconds 设置 Key 的过期时间，过期以后Key 将被自动删除；
* TTL key 获取 Key 的剩余生存时间；
* DEL key 删除 Key；
* TYPE key 获取 Key 对应的 Value 的类型；

通过 redis-cli 演示以上的指令如下：

127.0.0.1:6379> exists user

(integer) 0

127.0.0.1:6379> set user aidenOK

127.0.0.1:6379> get user

"aiden"

127.0.0.1:6379> type userstring

127.0.0.1:6379> expire user 5

(integer) 1

127.0.0.1:6379> ttl user

(integer) 4

127.0.0.1:6379> ttl user

(integer) 2

127.0.0.1:6379> ttl user

(integer) -2

127.0.0.1:6379> exists user

(integer) 0

上面例子中，首先判断 user 键是否存在，接着通过 SET 设置了值，接着还使用 EXPIRE 指令设置了过期时间为 5 秒。可以看到 5 秒后，user 键就被自动删除了。

有的时候会看到输出的字符串前有 b 这样的前缀，表示字节编码的字符串，对应的还有 u'xxxxx' 这类 unicode 编码的字符串，都比较常见。

上文中已提到，Redis 还支持其他的数据结构，不仅仅是简单的字符串键值对，比如支持哈希类型的键值，这种数据结构中 Key 对应于一个哈希，而哈希又包含多个字段和相应的值。对于这种类型主要有下面的操作指令:

* HSET key field value 设置名称为 key 的哈希的字段 field 为值 value；
* HGET key field 获取名为 key 的哈希的字段 field;
* HGETALL key 获取名为 Key 的哈希所有字段和 Value;
* HKEYS key 获取名为 Key 的哈希的所有字段；
* HLEN key 获取名为 Key 的哈希的字段数量；

通过 redis-cli 演示如下：

127.0.0.1:6379> exists user

(integer) 0

127.0.0.1:6379> hset user name aiden

(integer) 1

127.0.0.1:6379> hset user age 30

(integer) 1

127.0.0.1:6379> hmset user email luojin@simplecloud.cn addr chengdu

OK127.0.0.1:6379> hgetall user1) "name"2) "aiden"3) "age"4) "30"5) "email"6) "luojin@simplecloud.cn"7) "addr"8) "chengdu"127.0.0.1:6379> hkeys user1) "name"2) "age"3) "email"4) "addr"127.0.0.1:6379> hget user addr"chengdu"127.0.0.1:6379> hlen user

(integer) 4

上面的例子中，设置了一个名为 user 的哈希。先使用 HSET 为单个字段赋值，接着使用 HMSET 为多个字段赋值。使用 HGETALL 能一次获取全部的字段和值。

Redis 还支持有序集合，有序集合可以用于快速实现排名功能，只要的操作指令如下：

* ZADD key score member 将成员和对应的评分添加到有序集合中；
* ZREVRANK key member 获取 member 在有序集合 key 中的排名；

通过 redis-cli 演示如下：

127.0.0.1:6379> zadd rank 100 aiden

(integer) 1

127.0.0.1:6379> zadd rank 120 lxtttx

(integer) 1

127.0.0.1:6379> zadd rank 80 jin

(integer) 1

127.0.0.1:6379> zrevrank rank aiden

(integer) 1

127.0.0.1:6379> zrevrank rank lxtttx

(integer) 0

127.0.0.1:6379> zrevrank rank jin

(integer) 2

127.0.0.1:6379> zrevrank rank not\_exist

(nil)

上面例子中，我们通过 ZADD 往 rank 中添加了三个成员，最后通过 ZREVRANK 依次获取了成员的排名，可以发现排名是从 0 开始计算的，排第 0 的成员得分最高。Redis 还有其他各种操作指令，篇幅有限就不一一介绍了。

### Python 操作 Redis

Python 中访问 Redis 可以通过 [redis-py](https://github.com/andymccurdy/redis-py" \t "https://www.shiyanlou.com/courses/996/labs/4188/_blank) 软件包进行。类似于 PyMongo, 也是需要先创建一个 Redis 客户端，如下代码:

In [1]: import redis

In [2]: r = redis.StrictRedis(host='127.0.0.1', db=0)

In [3]: r.ping()

Out[3]: True

上面的代码中，通过 redis.StrictRedis 创建了一个 Redis 客户端，其中 db 参数指定了链接的逻辑数据库编号为 0。不同编号的数据库，可以有同名的 Key。客户端创建成功后，就可以进行各种指令操作了。 先尝试了 ping 方法，该方法返回 True 表示数据库工作正常。redis-py 客户端有各种和 Redis 指令同名的方法，调用这些方法就可以完成各种操作，比如获取上文中创建的哈希键 user：

In [4]: r.hgetall('user')

Out[4]:

{b'addr': b'chengdu',

b'age': b'30',

b'email': b'luojin@simplecloud.cn',

b'name': b'aiden'}

可以看到，redis-py 将返回的结果自动转换成了字典。

看到输出的字符串前有 b 这样的前缀，表示字节编码的字符串，对应的还有 u'xxxxx' 这类 unicode 编码的字符串，都比较常见。

Redis 也支持发布订阅消息模式。该功能使发布者和订阅者解耦，不需要对方的存在，只需要简单的往某一频道上发送数据就行了，订阅了该频道的订阅者会自动收到消息。下面我们进行演示，首先在 redis-py 客户端中订阅 labreport-channel 频道，并监听消息：

In [5]: p = r.pubsub()

In [6]: p.subscribe('labreport-channel')

In [7]: for msg in p.listen():

...: print(msg)

...:

{'type': 'subscribe', 'pattern': None, 'channel': b'labreport-channel', 'data': 1}

然后在 redis-cli 客户端中，通过 PUBLISH channel message 指令往频道中发布消息 message：

127.0.0.1:6379> publish labreport-channel "1 msg from redis-cli"

(integer) 1

127.0.0.1:6379> publish labreport-channel "2 msg from redis-cli"

(integer) 1

127.0.0.1:6379>

消息发布以后，就可以看到 IPython 终端中马上收到了消息：

In [7]: for msg in p.listen():

...: print(msg)

...:

{'type': 'subscribe', 'pattern': None, 'channel': b'labreport-channel', 'data': 1}

{'type': 'message', 'pattern': None, 'channel': b'labreport-channel', 'data': b'1 msg from redis-cli'}

{'type': 'message', 'pattern': None, 'channel': b'labreport-channel', 'data': b'2 msg from redis-cli'}