1、查询

- 排序order_by，desc()倒序/asc()升序

```
# 用户按照id倒序排序
SQL:
select user_id,mobile from user_basic order by user_id desc;
# ORM
User.query.order_by(User.id.desc()) # 查询对象
User.query.order_by(User.id.desc()).all()
```

- 复合查询

```
# 用户按照id倒序排序，手机号13开头
select user_id,mobile from user_basic where mobile like '13%' order by user_id
desc;
# ORM
User.query.filter(User.mobile.startswith('13')).order_by(User.id.desc()).all()

# 用户按照id倒序排序，手机号13开头，跳过第一条数据，展示13条
# SQL:limit 跳过几条，展示几条；
select user_id,mobile from user_basic where mobile like '13%' order by user_id
desc limit 1,13;
# ORM
User.query.filter(User.mobile.startswith('13')).order_by(User.id.desc()).limit
(1).offset(13).all()
```

- ORM优化查询，ORM默认是全字段扫描数据，效率较低，需要优化

```
# 需要导入函数load_only
User.query.options(load_only(User.id,User.mobile)).filter(User.mobile=='135522
85417').all()
```

- 聚合查询、分组

```
# 查询所有用户的粉丝数?
SQL:
select user_id,count(target_user_id) from user_relation;
+---------+-----------------------+
| user_id | count(target_user_id) |
+---------+-----------------------+
|       1 |                    10 |
+---------+-----------------------+
```

```
select user_basic.user_id,count(user_relation.target_user_id) from
user_relation join user_basic on user_relation.user_id=user_basic.user_id
group by user_id;
+---------+--------------------------------------+
| user_id | count(user_relation.target_user_id) |
+---------+--------------------------------------+
|       1 |                                    3 |
|       2 |                                    1 |
|       5 |                                    3 |
|      33 |                                    1 |
|      52 |                                    1 |
|      61 |                                    1 |
+---------+--------------------------------------+

# ORM,Relation表中没有字段叫做count(target_user_id)，所以，不能基于模型类查询。
# count(target_user_id)结果是通过sql函数计算得出的数据，ORM也需要使用函数实现。
# User.query.group_by(User.id,Relation)
from sqlalchemy import func
db.session.query(Relation.user_id,func.count(Relation.target_user_id)).group_b
y(Relation.user_id).all()
# [(1, 3), (2, 1), (5, 3), (33, 1), (52, 1), (61, 1)]
db.session.query(Relation.user_id,func.count(Relation.target_user_id)).filter(
Relation.relation==Relation.RELATION.FOLLOW).group_by(Relation.user_id).all()
# [(1, 3), (2, 1), (33, 1), (61, 1)]
```

- 关联查询
  - 1、**Foreignkey**，User表和Relation表，一方定义关系，多方定义外键

```
class User:
    # 参数1表示另外一方的类名，
    # 参数2backref可选，表示反向引用，用来从Relation查询User
    # relationship仅仅是连表查询时触发，在数据库中没有实体字段
   follows = db.relationship('Relation',backref='user')

class Relation:
    # ForeignKey表示外键，在数据库中没有实体，仅仅是连表查询时的条件。
    # 如果，当前模型类通过数据库迁移创建表，会产生真正的外键。
  user_id = db.Column(db.Integer,db.ForeignKey('user_basic.user_id')
,doc='用户ID')

u = User.query.get(1)
u.follows # [<Relation 11>, <Relation 19>, <Relation 10>]
r = Relation.query.get(11)
r.user
```

  - 2、**primaryjoin**，仅仅在一个模型类中定义即可

```
class User:
  # uselist表示使用列表，默认列表返回，如果False返回1个
  f =
db.relationship('Relation',primaryjoin='User.id==foreign(Relation.user_id)
',uselist=False)

u = User.query.get(1)
u.follows # [<Relation 11>, <Relation 19>, <Relation 10>]
```

- 指定字段连表查询优化：

```
# 根据手机号18516952650查询关注的人。
# SQL:
select
user_relation.user_id,user_relation.target_user_id,user_relation.relation_
id from user_relation join user_basic on
user_relation.user_id=user_basic.user_id where
user_basic.mobile='18516952650';
+---------+----------------+
| user_id | target_user_id |
+---------+----------------+
|       1 |              2 |
|       1 |              3 |
|       1 |              5 |
+---------+----------------+
+---------+----------------+-------------+
| user_id | target_user_id | relation_id |
+---------+----------------+-------------+
|       1 |              2 |          11 |
|       1 |              3 |          19 |
|       1 |              5 |          10 |
+---------+----------------+-------------+
# ORM优化查询，导入contains_eager
from sqlalchemy.orm import load_only,contains_eager
# join表示连接
User.query.join(User.follows).options(load_only(User.id),contains_eager(Us
er.follows).load_only(Relation.target_user_id)).filter(User.mobile=='18516
952650').all()

# orm对比sql
# 连表
User.query.join(User.follows)<==>user_relation join user_basic
# 指定字段
```

```
options(load_only(User.id),contains_eager(User.follows).load_only(Relation
.target_user_id))<==>select
user_relation.user_id,user_relation.target_user_id,user_relation.relation_
id
# 过滤条件
filter(User.mobile=='18516952650')<==>where
user_basic.mobile='18516952650';
```

- 添加、修改、删除

```
# 需要使用数据库会话对象session
# 添加
# session封装了数据库的基本操作，
add/add_all/commit/rollback/delete/create_all/drop_all
user1 = User(mobile='13012345678',name='python28')
db.session.add(user1) # add表示添加一个对象、add_all([user1,user2])
db.session.commit() # commit表示提交数据到数据库

# 修改，两种方式
# 1、直接update
User.query.filter(User.mobile=='13012345678').update({'name':'itcast_python28'
})
db.session.commit()
# 2、根据查询结果，对象点属性修改
user = User.query.filter(User.mobile=='13012345678').first()
user.name = 'python28'
db.session.add(user)
db.session.commit()

# 删除，两种方式
# 1
User.query.filter(User.mobile=='13012345678').delete()
db.session.commit()

# 2
user = User.query.filter(User.mobile=='13012345678').first()
db.session.delete(user)
db.session.commit()
```

- 事务：flask-sqlalchemy对数据的操作，隐式的开启事务

```
try:
  # 查询数据
  # 修改数据
  # flush表示把数据给刷到数据库中，但是，事务没有完成。
  # 删除数据
  # 提交事务
  db.session.commit()
except Exception as e:
  # 需要进行回滚
  db.session.rollback()
```

2、数据库理论

- 复制集，主从复制集，多台机器，集群
  - 1、一份数据存储在多台机器上，每台机器存储的数据是一样的。
  - 2、相当于数据备份；
  - 3、数据高可用；
- 分布式，多台机器，分布式集群(redis分布式集群)
  - 1、多台机器存储一份数据，每台机器存储的数据是不一样的。
  - 2、相当于数据切割，每台机器存储一部分，合在一起是一个整体。
- redis分布式集群包含了复制集，7000~7005共6台redis，三主三从；redis分布式有槽位slots，默认有16384个。
- 思考题：读写分离和事务的影响？结论：对重复读和串行化有影响。
  - 事务的隔离级别：
    - 1、读已提交，read-committed
    - 2、读未提交，read-uncommitted
    - 3、重复读，repeatable
    - 4、串行化，serializable
- 分库分表：先垂直，再水平。
  - 1、垂直拆分，项目初期就可以考虑，
  - 2、水平拆分，项目数据积累到一定程度，分库分表
  - 头条项目中只有垂直分表，单表字段太多。

3、分布式id

- 项目中多台机器分布式存储数据，多台机器之间是一份完整的数据；
- 问题：主键id？**唯一、自增**。
- 解决问题的思路：10个mysql数据库。
  - 1、uuid，全局唯一标识符，比较长，16进制，不自增；

- 2、每台机器设立**初始值+步长**，1000 + 1；2000+1...，扩展性不强，主键id容易被破解；
- 3、利用1台mysql生产主键id，然后，给其他mysql使用，容易单台机器故障；
- 4、利用1台redis生产主键id，单台机器故障；
- 雪花算法 = 时间戳(从1970年到当前时间毫秒数) + 机器码 + 序号
- 特点：强依赖于时间戳，如果时间不准，时钟回拨;
  - 时钟回拨：没生产一个时间戳，判断是否大于上一个时间戳，如果小于，发生了时间回拨，只能抛出异常。

4、数据库优化

- 索引：🎄树

5、sql语句优化

- 1、避免全表扫描，禁止select *
- 2、SQL语句尽量大写
- 3、尽量避免在 where 子句中使用!=或<>操作符，会让索引失效。
- 4、遵循最左原则，在where子句中写查询条件时把索引字段放在前面
- 5、能使用关联查询解决的尽量不要使用子查询；能不使用关联查询的尽量不要使用关联查询；
- 6、不需要获取全表数据的时候，不要查询全表数据，使用LIMIT来限制数据。

6、数据库优化

- 在进行表设计时，可适度增加冗余字段(反范式设计)，减少JOIN操作；
- 多字段表可以进行垂直分表优化，多数据表可以进行水平分表优化；
- 选择恰当的数据类型，如整型的选择；
- 对于强调快速读取的操作，可以考虑使用MyISAM数据库引擎，全文索引；
- 对较频繁的作为查询条件的字段创建索引；唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件；更新非常频繁的字段不适合创建索引；
- 编写SQL时使用上面的方式对SQL语句进行优化；
- 使用慢查询工具找出效率低下的SQL语句进行优化；
- 构建缓存，减少数据库磁盘操作；
- 可以考虑结合使用内存型数据库，如Redis，进行混合存储。

7、redis事务

- 概念：保证一组操作执行，没有回滚操作；
- 实现：multi、exec，在一次连接过程中，执行多条指令；

```
# python代码中应用
r = Redis('127.0.0.1:6379')
# 减少数据库的连接操作
p = r.pipeline()
p.multi()
p.set()
p.get()
p.setex()
p.execute()
```

- watch看管、监视，配合multi使用，
  - 1、如果看管的数据发生变化，事务不会执行
  - 2、相当于Django阶段学习的乐观锁；

8、redis持久化

- 概念：redis能够自动把数据备份到磁盘中；
- 1、RDB表示快照：**redis默认开启的持久化方案**；
  - 手动快照，save/bgsave，推荐使用bgsave，save会影响redis的性能，线上不能使用。
  - 手动快照，shutdown

```
# save <seconds> <changes>
# 表示关闭快照
#   save ""

save 900 1 # 900秒有1次写操作，快照1次
save 300 10 # 300秒有10次写操作
save 60 10000 # 60秒有10000次写操作
save 10 100000
```

- 2、AOF表示追加文件：**redis默认未开启AOF机制**，会记录对redis操作的指令。

```
appendonly yes  # 是否开启AOF
appendfilename "appendonly.aof"  # AOF文件

# appendfsync always  # 每个操作都写到磁盘中，不会使用
appendfsync everysec  # 每秒写一次磁盘，默认
# appendfsync no  # 由操作系统决定写入磁盘的时机
```

- 项目中使用：RDB和AOF如何选择？两种都开启，最坏情况，只丢失1秒内的数据。

9、redis高可用

- redis分布式集群 = 主从同步+Sentinel哨兵机制

- 哨兵：是用来看护redis实例进程的，可以自动进行故障转移。