1. 水平拆分

1.1 概念

拆分规则

- 时间
  - 按照时间切分，就是将6个月前，甚至一年前的数据切出去放到另外的一张表，因为随着时间流逝，这些表的数据 被查询的概率变小，所以没必要和“热数据”放在一起，这个也是“冷热数据分离”。
- 业务
  - 按照业务将数据进行分类并拆分，如文章包含金融、科技等多个分类，可以每个分类的数据拆分到一张表中。
- ID范围
  - 从 0 到 100W 一个表，100W+1 到 200W 一个表。
- HASH取模 离散化
  - 取用户id，然后hash取模，分配到不同的数据库上。这样可以同时向多个表中插入数据，提高并发能力，同时由于用户id进行了离散处理，不会出现ID冲突的问题
- 地理区域
  - 比如按照华东，华南，华北这样来区分业务，部分云服务应该就是如此。

可能拆分之后性能没有提升

1.2 代码演示

```
# 水平拆分 User1, User2 字段一样, __bind_key__ 不同库
class User1(db.Model):
    __tablename__ = 'tb_user_1'
    __bind_key__ = 'db1'
    id = Column(Integer, primary_key=True)
    name = Column(String(32))
    age = Column(Integer)


class User2(db.Model):
    __tablename__ = 'tb_user_2'
    __bind_key__ = 'db2'
    id = Column(Integer, primary_key=True)
    name = Column(String(32))
    age = Column(Integer)

```

sqlalchemy 不允许名字重复，本质上应该允许

字段一样

```python
@app.route('/')
def index():

    users_1 = db.session.query(User).all()
    users_2 = db.session.query(User2).all()


    return "index"


if __name__ == '__main__':
    # 重置所有继承自db.Model的表
    db.drop_all()
    db.create_all()

    # 添加测试数据，需要分别往db1和db2中添加一条数据
    user1 = User1(name='zs', age=20)
    db.session.add(user1)
    user2 = User2(name='lisi', age=20)
    db.session.add(user2)
    db.session.commit()

```

db1 独立事务

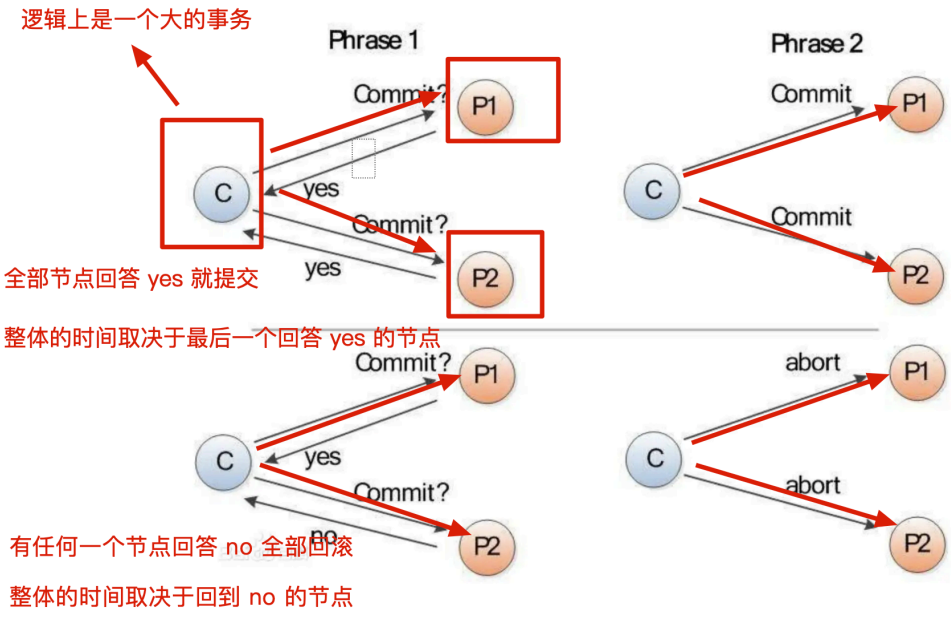
db2 独立事务

在内部分别提交

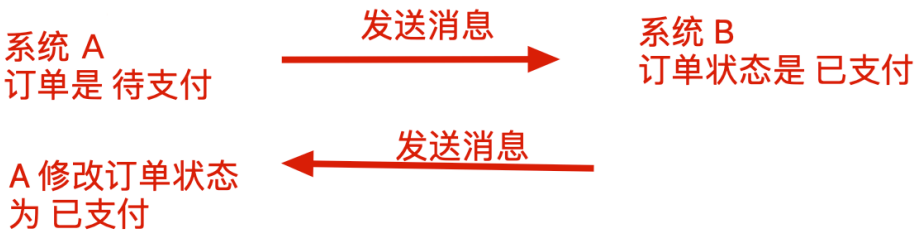
2. 分布式问题

两阶段提交

预提交 成功后，事务管理器才会统一执行提交处理，否则统一进行回滚处




```python
db = SQLAlchemy(app, session_options={'twophase': True})
```



每一个系统 都有自己独立的状态信息

系统之间通过发送消息，触发另外一个系统的状态更新，从而实现最终状态是一致的

3. 跨节点查询的问题

文章: db1，db2 水平拆分

Db1 : 1，2，3，4，5，6，7，8，9，10

Db2: 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

11，12，13

1，2，3

分页查询: 按照时间排序，取最新的数据，每页 3 条数据，第一页

> 如果是一个库: ctime.desc() limit(3) offset 0

1. 先取 db1 ctime.desc() limit(3) offset 0 --> 获取 db1 最新 3 条数据
2. 再取 db2 ctime.desc() limit(3) offset 0 --> 获取 db2 最新 3 条数据
3. 合并 db1 和 db2 最新 3 条得到--> 整体的最新 6 条数据
4. 对最新 6 条数据排序 取前面 3 条

> 分别从不同库区前面所有页的数据，手动再应用中进行排序，获取排序之后的分页数据

用户+用户地址: 用户 db1，用户地址 db2

user: name,age; address: detail,user_id

name,age,detail

> 查询用户 id=1
>
> 先查询db1 1，zhangsan,100
>
> 再查询 db2 1，sh，1; 2 ，bg, 1; 3, ng，1

```
user = ( 1, zhangsan,100)
addresses = [(1, sh, 1) (2 , bg, 1) (3, ng,  1)]
for adr in addresses:
  pritn(user[1],user[2],adr[1]) # name,age,detail
```

> 数据库中的 join 实现: 嵌套 for 循环

4. redis 语法回顾

5. redis 事务

5.1 事务命令

```
$ redis-cli
127.0.0.1:6379> set user1 zs  # 设置string类型的键 user1
OK
127.0.0.1:6379> type user1  # 查看user1类型
string
127.0.0.1:6379> multi  # 开启事务
OK
127.0.0.1:6379> set age 20  # 设置string类型的键 age, 事务中的操作不会立即执行, 只是入列
QUEUED
127.0.0.1:6379> hset user1 name zs  # 设置hash类型的键 user1, 由于user1已存在, 且为string类
QUEUED
127.0.0.1:6379> set height 1.8  # 设置string类型的键 height
OK
127.0.0.1:6379> exec  # 提交事务, 即使部分操作失败, 不回滚且继续执行
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
3) OK
```

指令是独立的，失败了不会影响到其他指令

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set age 1000
QUEUED
127.0.0.1:6379> set height 10000
QUEUED
127.0.0.1:6379> DISCARD
OK
127.0.0.1:6379> exec
(error) ERR EXEC without MULTI
127.0.0.1:6379>
```

1. 不仅取消指令队列还会取消当前事务
2. 事务已经被取消

5.2 事务 acid

- 原子性
  - 不支持
  - 不会回滚并且继续执行
- 隔离性
  - 支持
  - 事务中命令顺序执行, 并且不会被其他客户端打断 (先EXEC的先执行)
  - 单机redis读写操作使用 **单进程单线程**
- 持久性
  - 支持, 但相比Mysql, redis数据易丢失
- 一致性
  - 不支持

老版本单线程，新版本多线程

aof 和 rdb 持久化方案

5.3 python 操作

```
In [4]: pipe = client.pipeline?
Signature: client.pipeline(transaction=True, shard_hint=None)
Docstring:
Return a new pipeline object that can queue multiple commands for
later execution. ``transaction`` indicates whether all commands
should be executed atomically. Apart from making a group of operations
atomic, pipelines are useful for reducing the back-and-forth overhead
between the client and server.
File:      ~/.envs/py3/lib/python3.8/site-packages/redis/client.py
Type:      method

In [5]: pipe = client.pipeline()

In [6]: a = pipe.set('name','zhangsan')

In [7]: print(a)
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>

In [8]: b = pipe.get('name')

In [9]: print(b)
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>

In [10]: c = pipe.execute() # EXEC

In [11]: print(c)
[True, b'zhangsan']

In [12]:
```

不是立即执行

返回所有指令的执行结果

6. 乐观锁

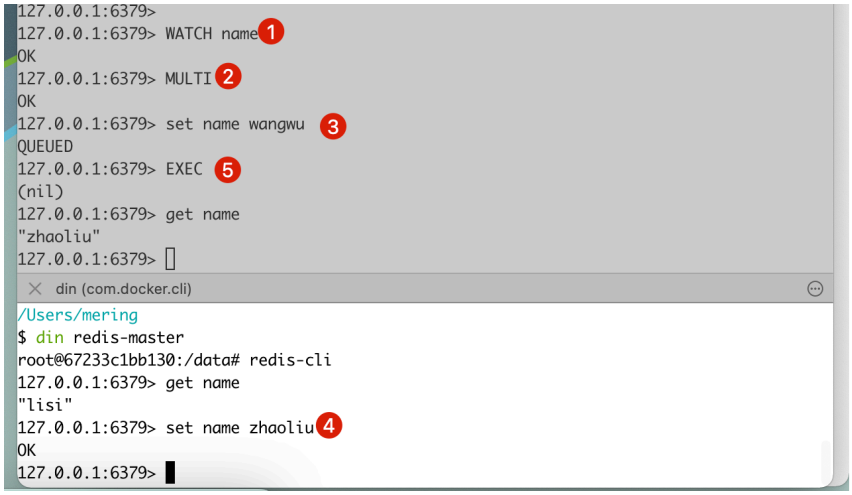
```
"zhangsan"
127.0.0.1:6379> WATCH name
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set name lisi
QUEUED
127.0.0.1:6379> EXEC
1) OK
```

1

2

3

ok 表示成功 nil 表示不成功



5 Redis 发现当前客户端监听的 key 被其他客户端修改了，所以这里会自动取消事务

6.1 乐观锁实现

6.2 乐观锁 python 操作

```python
pipe = client.pipeline()  # 默认开启式事务
# 3. while True 循环
while True:
    #    try: 捕获执行中抛出的 WatchError，如果程序抛出这个异常，说明监听的 key 被修改了，乐观更新失败
    try:
        #        1. 监听key watch(key), watch 之后再通过管道读取 redis 数据，会立即执行
        pipe.watch('goods_count')
        #        2. 读取库存
        goods_count_bytes = pipe.get('goods_count')  # 字节数据,直接读取 redis 的数据
        goods_count_str = goods_count_bytes.decode()
        goods_count = int(goods_count_str)

        #        3. 如果有库存
        if goods_count > 0:
            #            1. 开启事务 pipe.multi()
            pipe.multi()
            #            2. 通过管道添加要执行指令
            pipe.decr('goods_count')
            #            3. pipe.execute() 提交事务
            pipe.execute()# 如果乐观更新失败，会抛出 WatchError 异常
            print("购买成功")
        #        4. 没有库存
        else:
            #            重置管道移除掉监听 pipe.reset()
            pipe.reset()
            print('没有库存了')
```

7. 悲观锁

7.1 悲观锁介绍

7.2 使用悲观锁

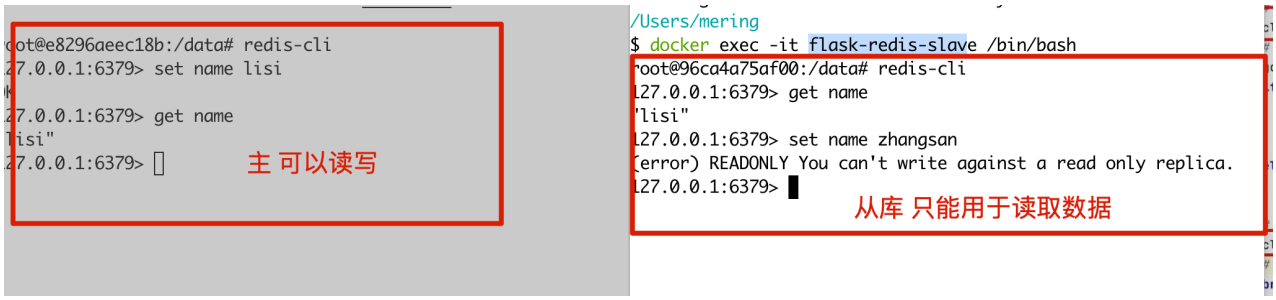
```python
client = StrictRedis(decode_responses=True)
# 2. while True 循环
while True:
    #     1. 设置悲观锁 client.setnx(锁的 key, 值)
    lock_success = client.setnx('count:lock', 1)
    #     2. 如果设置锁成功
    if lock_success:
        #         1. 设置锁的过期时间 client.expire(锁的 key, 过期时间)
        client.expire('count:lock', 5)
        #         2. 读取库存
        goods_count = int(client.get('goods_count'))
        if goods_count > 0:
            #         3. 如果有库存，就减少库存
            client.decr('goods_count')
            print('下单成功')
        else:
            #         4. 没有库存，打印消息
            print('没有库存了')
        #         5. 删除锁 client.delete(锁的 key)
        client.delete('count:lock')
        #         6. break 循环
        break
    else:
        #     3. 如果设置失败
        #         睡眠一会继续
        time.sleep(2)
```

指令的执行可能更早就结束了，主动释放锁

8. 非事务型管道介绍

9. redis 主从

搭建环境以及演示



- 只能一主多从 (mysql可以多主多从)
- 从数据库不能写入 (mysql可以写)

10. redis 哨兵

介绍

- 作用
  - 监控redis服务器的运行状态, 可以进行 **自动故障转移**(failover), 实现高可用
  - 与 **数据库主从** 配合使用的机制
- 特点
  - 独立的进程, 每台redis服务器应该至少配置一个哨兵程序
  - 监控redis主服务器的运行状态
  - 出现故障后可以向管理员/其他程序发出通知
  - 针对故障,可以进行自动转移, 并向客户端提供新的访问地址

环境搭建

python 操作