redis_day02回顾

五大数据类型及应用场景

类型	特点	使用场景
string	简单key-value类型，value可为字符串和数字	常规计数（微博数, 粉丝数等功能）
hash	是一个string类型的field和value的映射表，hash特别适合用于存储对象	存储部分可能需要变更的数据（比如用户信息）
list	有序可重复列表	关注列表，粉丝列表，消息队列等
set	无序不可重复列表	存储并计算关系（如微博，关注人或粉丝存放在集合，可通过交集、并集、差集等操作实现如共同关注、共同喜好等功能）
sorted set	每个元素带有分值的集合	各种排行榜

位图操作（bitmap）

```
# 应用场景
1、可以实时的进行数据统计（网站用户的上线次数统计）
# 常用命令
1、setbit key offset value
2、BITCOUNT key
```

哈希（散列）类型

```
# 应用场景
1、很适合存储对象类型，比如说用户ID作为key，用户的所有属性及值作为key对应的value
（用户维度统计-各种数据统计-发帖数、粉丝数等）
# 常用命令
HSET key field value
HSETNX key field value
HMSET key field value field value

HGET key field
```

```
HMGET key field filed
HGETALL key
HKEYS key
HVALS key

HLEN key
HEXISTS key field

HINCRBY key filed increment
HINCRBYFLOAT key field increment

HDEL key field
```

集合类型

```
# 应用场景
1、共同关注、共同好友
# 常用命令

SADD key member1 member2

SMEMBERS key
SCARD key

SREM key member1 member2
SRANDOMMEMBER key [count]

SISMEMBER key member

SDIFF key1 key2
SDIFFSTORE destination key1 key2

SINTER key1 key2
SINTERSTORE destination key1 key2

SUNION key1 key2
SUNIONSTORE destination key1 key2
```

有序集合

```
# 应用场景
1、各种排行榜
    1、游戏：列出前100名高分选手
    2、列出某用户当前的全球排名
    3、各种日排行榜、周排行榜、月排行榜
# 常用命令
zadd key score member

ZRANGE key start stop [withscores]
```

```
ZREVRANGE key start stop [withscores]
ZRANGEBYSCORE key min max [withscores] [limit offset count]
ZSCORE key member
ZCOUNT key min max
ZCARD key

ZRANK key member
ZREVRANK key member

ZINCRBY key increment member

ZREM key member
ZREMRANGEBYSCORE key min max

zunionstore destination numkeys key [weights 权重值] [AGGREGATE SUM|MIN|MAX]
ZINTERSTORE destination numkeys key1 key2 WEIGHTS weight AGGREGATE SUM|MIN|MAX
```

redis_day03笔记

有序集合sortedset

有序集合的交集与并集

```
# 交集（weights代表权重值，aggregate代表聚合方式 - 先计算权重值，然后再聚合）
ZINTERSTORE destination numkeys key1 key2 WEIGHTS weight AGGREGATE SUM|MIN|MAX
# 并集（weights代表权重值，aggregate代表聚合方式 - 先计算权重值，然后再聚合）
ZUNIONSTORE destination numkeys key [weights 权重值] [AGGREGATE SUM|MIN|MAX]
```

案例1：网易音乐排行榜

```
1、每首歌的歌名作为元素
2、每首歌的播放次数作为分值
3、使用ZREVRANGE来获取播放次数最多的歌曲
```

代码实现

```

```

案例2: 京东商品畅销榜

```
# 第1天
ZADD mobile-001 5000 'huawei' 4000 'oppo' 3000 'iphone'
# 第2天
ZADD mobile-002 5200 'huawei' 4300 'oppo' 3230 'iphone'
# 第3天
ZADD mobile-003 5500 'huawei' 4660 'oppo' 3580 'iphone'
问题：如何获取三款手机的销量排名?
ZUNIONSTORE mobile-001:003 mobile-001 mobile-002 mobile-003 # 可否?
# 正确
方法1:
方法2:
```

python代码实现

```

```

数据持久化

持久化定义

```
将数据从掉电易失的内存放到永久存储的设备上
```

为什么需要持久化

```
因为所有的数据都在内存上，所以必须得持久化
```

- **数据持久化分类之 - RDB模式（默认开启）**

默认模式

```
1、保存真实的数据
2、将服务器包含的所有数据库数据以二进制文件的形式保存到硬盘里面
3、默认文件名 ： /var/lib/redis/dump.rdb
```

创建rdb文件的两种方式

方式一：服务器执行客户端发送的SAVE或者BGSAVE命令

```
127.0.0.1:6379> SAVE
OK
# 特点
1、执行SAVE命令过程中，redis服务器将被阻塞，无法处理客户端发送的命令请求，在SAVE命令执行完毕后，服务器才会重新开始处理客户端发送的命令请求
2、如果RDB文件已经存在，那么服务器将自动使用新的RDB文件代替旧的RDB文件
# 工作中定时持久化保存一个文件

127.0.0.1:6379> BGSAVE
Background saving started
# 执行过程如下
1、客户端 发送 BGSAVE 给服务器
```

```
2、服务器马上返回 Background saving started 给客户端
3、服务器 fork() 子进程做这件事情
4、服务器继续提供服务
5、子进程创建完RDB文件后再告知Redis服务器

# 配置文件相关操作
/etc/redis/redis.conf
263行: dir /var/lib/redis # 表示rdb文件存放路径
253行: dbfilename dump.rdb  # 文件名

# 两个命令比较
SAVE比BGSAVE快，因为需要创建子进程，消耗额外的内存

# 补充：可以通过查看日志文件来查看redis都做了哪些操作
# 日志文件：配置文件中搜索 logfile
logfile /var/log/redis/redis-server.log
```

方式二：设置配置文件条件满足时自动保存**（使用最多）**

```
# 命令行示例
redis>save 300 10
  表示如果距离上一次创建RDB文件已经过去了300秒，并且服务器的所有数据库总共已经发生了不少于10次修改，那么自动执行BGSAVE命令
redis>save 60 10000
  表示如果距离上一次创建rdb文件已经过去60秒，并且服务器所有数据库总共已经发生了不少于10000次修改，那么执行bgsave命令

# redis配置文件默认
218行: save 900 1
219行: save 300 10
220行: save 60 10000
  1、只要三个条件中的任意一个被满足时，服务器就会自动执行BGSAVE
  2、每次创建RDB文件之后，服务器为实现自动持久化而设置的时间计数器和次数计数器就会被清零，并重新开始计数，所以多个保存条件的效果不会叠加
```

- **数据持久化分类之 - AOF（AppendOnlyFile，默认未开启）**

特点

```
1、存储的是命令，而不是真实数据
2、默认不开启
# 开启方式（修改配置文件）
1、/etc/redis/redis.conf
  672行: appendonly yes # 把 no 改为 yes
  676行: appendfilename "appendonly.aof"
2、重启服务
  sudo /etc/init.d/redis-server restart
```

RDB缺点

```
1、创建RDB文件需要将服务器所有的数据库的数据都保存起来，这是一个非常消耗资源和时间的操作，所以服务器需要隔一段时间才创建一个新的RDB文件，也就是说，创建RDB文件不能执行的过于频繁，否则会严重影响服务器的性能
2、可能丢失数据
```

AOF持久化原理及优点

```
# 原理
    1、每当有修改数据库的命令被执行时，服务器就会将执行的命令写入到AOF文件的末尾
    2、因为AOF文件里面存储了服务器执行过的所有数据库修改的命令，所以给定一个AOF文件，服务器只要重新执行一遍AOF文件里面包含的所有命令，就可以达到还原数据库的目的

# 优点
   用户可以根据自己的需要对AOF持久化进行调整，让Redis在遭遇意外停机时不丢失任何数据，或者只丢失一秒钟的数据，这比RDB持久化丢失的数据要少的多
```

安全性问题考虑

```
# 因为
   虽然服务器执行一个修改数据库的命令，就会把执行的命令写入到AOF文件，但这并不意味着AOF文件持久化不会丢失任何数据，在目前常见的操作系统中，执行系统调用write函数，将一些内容写入到某个文件里面时，为了提高效率，系统通常不会直接将内容写入硬盘里面，而是将内容放入一个内存缓存区（buffer）里面，等到缓冲区被填满时才将存储在缓冲区里面的内容真正写入到硬盘里

# 所以
   1、AOF持久化：当一条命令真正的被写入到硬盘里面时，这条命令才不会因为停机而意外丢失
   2、AOF持久化在遭遇停机时丢失命令的数量，取决于命令被写入到硬盘的时间
   3、越早将命令写入到硬盘，发生意外停机时丢失的数据就越少，反之亦然
```

策略 - 配置文件

```
# 打开配置文件:/etc/redis/redis.conf，找到相关策略如下
1、701行：alwarys
    服务器每写入一条命令，就将缓冲区里面的命令写入到硬盘里面，服务器就算意外停机，也不会丢失任何已经成功执行的命令数据
2、702行：everysec（# 默认）
    服务器每一秒将缓冲区里面的命令写入到硬盘里面，这种模式下，服务器即使遭遇意外停机，最多只丢失1秒的数据
3、703行：no
    服务器不主动将命令写入硬盘，由操作系统决定何时将缓冲区里面的命令写入到硬盘里面，丢失命令数量不确定

# 运行速度比较
always：速度慢
everysec和no都很快，默认值为everysec
```

AOF文件中是否会产生很多的冗余命令?

```
为了让AOF文件的大小控制在合理范围，避免胡乱增长，redis提供了AOF重写功能，通过这个功能，服务器可以产生一个新的AOF文件
  -- 新的AOF文件记录的数据库数据和原由的AOF文件记录的数据库数据完全一样
  -- 新的AOF文件会使用尽可能少的命令来记录数据库数据，因此新的AOF文件的提及通常会小很多
  -- AOF重写期间，服务器不会被阻塞，可以正常处理客户端发送的命令请求
```

示例

原有AOF文件	重写后的AOF文件

原有AOF文件	重写后的AOF文件
select 0	SELECT 0
sadd myset peiqi	SADD myset peiqi qiaozhi danni lingyang
sadd myset qiaozhi	SET msg 'hello tarena'
sadd myset danni	RPUSH mylist 2 3 5
sadd myset lingyang	
INCR number	
INCR number	
DEL number	
SET message 'hello world'	
SET message 'hello tarena'	
RPUSH mylist 1 2 3	
RPUSH mylist 5	
LPOP mylist	

AOF文件重写方法触发

```
1、客户端向服务器发送BGREWRITEAOF命令
   127.0.0.1:6379> BGREWRITEAOF
   Background append only file rewriting started

2、修改配置文件让服务器自动执行BGREWRITEAOF命令
  auto-aof-rewrite-percentage 100
  auto-aof-rewrite-min-size 64mb
  # 解释
    1、只有当AOF文件的增量大于100%时才进行重写，也就是大一倍的时候才触发
        # 第一次重写新增：64M
        # 第二次重写新增：128M
        # 第三次重写新增：256M（新增128M）
```

RDB和AOF持久化对比

RDB持久化	AOF持久化
全量备份，一次保存整个数据库	增量备份，一次保存一个修改数据库的命令
保存的间隔较长	保存的间隔默认为一秒钟
数据还原速度快	数据还原速度一般，冗余命令多，还原速度慢

RDB持久化	AOF持久化
执行SAVE命令时会阻塞服务器，但手动或者自动触发的BGSAVE不会阻塞服务器	无论是平时还是进行AOF重写时，都不会阻塞服务器
更适合数据备份	更适合用来保存数据，通常意义上的数据持久化，在appendfsync always模式下运行

```
# 用redis用来存储真正数据，每一条都不能丢失，都要用always，有的做缓存，有的保存真数据，我可以开多个redis服务，不同业务使用不同的持久化，新浪每个服务器上有4个redis服务，整个业务中有上千个redis服务，分不同的业务，每个持久化的级别都是不一样的。
```

配置文件常用配置总结

```
# 设置密码
1、
# 开启远程连接
2、
3、
# rdb持久化-默认配置
4、
5、
# rdb持久化-自动触发
6、
7、
8、
# aof持久化开启
9、
10、
# aof持久化策略
11、
12、
13、
# aof重写触发
14、
15、
```

Redis主从复制

- **定义**

```
1、一个Redis服务可以有多个该服务的复制品，这个Redis服务成为master，其他复制品成为slaves
2、master会一直将自己的数据更新同步给slaves，保持主从同步
3、只有master可以执行写命令，slave只能执行读命令
```

- **作用**

```
分担了读的压力（高并发）
```

- **原理**

```
从服务器执行客户端发送的读命令，比如GET、 LRANGE、 SMEMMBERS、HGET、ZRANGE等等，客户端可以连接slaves执行读请求，来降低master的读压力
```

- **两种实现方式**

方式一（命令行实现1）

redis-server --slaveof

```
# 从服务端
redis-server --port 6300 --slaveof 127.0.0.1 6379
# 从客户端
redis-cli -p 6300
127.0.0.1:6300> keys *
# 发现是复制了原6379端口的redis中数据
127.0.0.1:6380> set mykey 123
(error) READONLY You can't write against a read only slave.
127.0.0.1:6380>
# 从服务器只能读数据，不能写数据
```

方式一（命令行实现2）

```
# 服务端启动
redis-server --port 6301
# 客户端连接
tarena@tedu:~$ redis-cli -p 6301
127.0.0.1:6301> keys *
1) "myset"
2) "mylist"
127.0.0.1:6301> set mykey 123
OK
# 切换为从
127.0.0.1:6301> slaveof 127.0.0.1 6379
OK
127.0.0.1:6301> set newkey 456
(error) READONLY You can't write against a read only slave.
127.0.0.1:6301> keys *
1) "myset"
2) "mylist"
# 再切换为主
127.0.0.1:6301> slaveof no one
OK
127.0.0.1:6301> set name hello
OK
```

方式二(修改配置文件)

```
# 修改配置文件
vi redis_6300.conf
slaveof 127.0.0.1 6379
port 6300
# 启动redis服务
redis-server redis_6300.conf
# 客户端连接测试
redis-cli -p 6300
127.0.0.1:6300> hset user:1 username guods
(error) READONLY You can't write against a read only slave.
```

问题总结：master挂了怎么办?

```
1、一个Master可以有多个Slaves
2、Slave下线，只是读请求的处理性能下降
3、Master下线，写请求无法执行
4、其中一台Slave使用SLAVEOF no one命令成为Master，其他Slaves执行SLAVEOF命令指向这个新的Master，从它这里同步数据
# 以上过程是手动的，能够实现自动，这就需要Sentine哨兵，实现故障转移Failover操作
```

演示

```
1、启动端口6400redis，设置为6379的slave
   redis-server --port 6400
   redis-cli -p 6400
   redis>slaveof 127.0.0.1 6379
2、启动端口6401redis，设置为6379的slave
   redis-server --port 6401
   redis-cli -p 6401
   redis>slaveof 127.0.0.1 6379
3、关闭6379redis
   sudo /etc/init.d/redis-server stop
4、把6400redis设置为master
   redis-cli -p 6401
   redis>slaveof no one
5、把6401的redis设置为6400redis的salve
   redis-cli -p 6401
   redis>slaveof 127.0.0.1 6400
# 这是手动操作，效率低，而且需要时间，有没有自动的？？？
```

官方高可用方案Sentinel

Redis之哨兵 - sentinel

```
1、Sentinel会不断检查Master和Slaves是否正常
2、每一个Sentinel可以监控任意多个Master和该Master下的Slaves
```

案例演示

1、环境搭建

```
# 共3台redis的服务器，如果是不同机器端口号可以是一样的
1、启动6379的redis服务器
    sudo /etc/init.d/redis-server start
2、启动6380的redis服务器，设置为6379的从
    redis-server --port 6380
    tarena@tedu:~$ redis-cli -p 6380
    127.0.0.1:6380> slaveof 127.0.0.1 6379
    OK
3、启动6381的redis服务器，设置为6379的从
    redis-server --port 6381
    tarena@tedu:~$ redis-cli -p 6381
    127.0.0.1:6381> slaveof 127.0.0.1 6379
```

2、安装并搭建sentinel哨兵

```
# 1、安装redis-sentinel
sudo apt install redis-sentinel

# 2、新建配置文件sentinel.conf
port 26379
Sentinel monitor tedu 127.0.0.1 6379 1

# 3、启动sentinel
方式一: redis-sentinel sentinel.conf
方式二: redis-server sentinel.conf --sentinel

# 将master的redis服务终止，查看从是否会提升为主
sudo /etc/init.d/redis-server stop
# 发现提升6381为master，其他两个为从
# 在6381上设置新值，6380查看
127.0.0.1:6381> set name tedu
OK

# 启动6379，观察日志，发现变为了6381的从
主从+哨兵基本就够用了
```

sentinel.conf解释

```
# sentinel监听端口，默认是26379，可以修改
port 26379
# 告诉sentinel去监听地址为ip:port的一个master，这里的master-name可以自定义，quorum是一个数字，指明当有多少个sentinel认为一个master失效时，master才算真正失效
sentinel monitor <master-name> <ip> <redis-port> <quorum>
```

分布式锁

高并发产生的问题?

```
1、购票：多个用户抢到同一张票?
2、购物：库存只剩1个,被多个用户成功买到?
... ...
```

怎么办?

```
在不同进程需要互斥地访问共享资源时，分布式锁是一种非常有用的技术手段
```

原理

```
1、多个客户端先到redis数据库中获取一把锁,得到锁的用户才可以操作数据库
2、此用户操作完成后释放锁，下一个成功获取锁的用户再继续操作数据库
```

实现

```

```

博客项目解决高并发问题

1、在数据库中创建库 blog，指定字符编码utf8

```
mysql -uroot -p123456
mysql>create database blog charset utf8;
```

2、同步数据库，并在user_profile中插入表记录

```
1、python3 manage.py makemigrations
2、python3 manage.py migrate
3、insert into user_profile values
('guoxiaonao','guoxiaonao','guoxiaonao@tedu.cn','123456','aaaaaaaa','bbbbbbbb','cccccccc');
```

3、启动django项目，并找到django路由测试 test_api函数

```
1、python3 manage.py runserver
2、查看项目的 urls.py 路由，打开firefox浏览器输入地址：http://127.0.0.1:8000/test
# 返回结果：  HI HI HI
```

4、在数据库表中创建测试字段score

```
1、user/models.py添加:
   score = models.IntegerField(verbose_name=u'分数',null=True,default=0)
2、同步到数据库
   python3 manage.py makemigrations user
   python3 manage.py migrate user
3、到数据库中确认查看
```

3、在blog/views.py中补充 test函数，对数据库中score字段进行 +1 操作

```
from user.models import UserProfile
def test(request):


    u = UserProfile.objects.get(username='guoxiaonao')
    u.score += 1
    u.save()

    return JsonResponse('HI HI HI')
```

4、启多个服务端，模拟30个并发请求

(1)多台服务器启动项目

```
python3 manage.py runserver 127.0.0.1:8000
python3 manage.py runserver 127.0.0.1:8001
```

(2)在tools中新建py文件 test_api.py，模拟30个并发请求

```
import threading
import requests
import random


def getRequest():
    url='http://127.0.0.1:8000/test'
    url2='http://127.0.0.1:8001/test'
    get_url = random.choice([url, url2])
    requests.get(get_url)

ts = []
for i in range(30):

    t=threading.Thread(target=getRequest,args=())
    ts.append(t)

if __name__ == '__main__':

    for t in ts:
        t.start()

    for t in ts:
        t.join()
```

(3) python3 test_api.py

(4) 在数据库中查看 score 字段的值

```
并没有+30，而且没有规律，每次加的次数都不同，如何解决？？？
```

解决方案：redis分布式锁

```
def test(request):
```

```python
    # 解决方法二:redis分布式锁
    import redis
    pool = redis.ConnectionPool(host='localhost', port=6379, db=0)
    r = redis.Redis(connection_pool=pool)
    while True:
        try:
            with r.lock('guoxiaonao', blocking_timeout=3) as lock:
                u = UserProfile.objects.get(username='guoxiaonao')
                u.score += 1
                u.save()
            break
        except Exception as e:
            print('lock is failed')

    return HttpResponse('HI HI HI')
```