# ** 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== |

# redis_day03笔记

## 事务

特点

1. 单独的隔离操作：事务中的所有命令会被序列化、按顺序执行，在执行的过程中不会被其他客户端发送来的命令打断
2. 不保证原子性：redis中的一个事务中如果存在命令执行失败，那么其他命令依然会被执行，没有回滚机制

事务命令

```
1、MULTI  # 开启事务          mysql   begin
2、命令1  # 执行命令
3、命令2 ... ...
4、EXEC  # 提交到数据库执行    mysql   commit
4、DISCARD # 取消事务          mysql  'rollback'
```

使用步骤

```
# 开启事务
127.0.0.1:6379> MULTI
OK
# 命令1入队列
127.0.0.1:6379> INCR n1
QUEUED
# 命令2入队列
127.0.0.1:6379> INCR n2
QUEUED
# 提交到数据库执行
127.0.0.1:6379> EXEC
1) (integer) 1
2) (integer) 1
```

事务中命令错误处理

```
# 1、命令语法错误，命令入队失败，直接自动discard退出这个事务
  这个在命令在执行调用之前会发生错误。例如，这个命令可能有语法错误（错误的参数数量，错误的命令名）
  处理方案：语法错误则自动执行discard

案例：
127.0.0.1:6379[7]> MULTI
OK
127.0.0.1:6379[7]> get a
QUEUED
127.0.0.1:6379[7]> getsss a
(error) ERR unknown command 'getsss'
127.0.0.1:6379[7]>
127.0.0.1:6379[7]>
127.0.0.1:6379[7]> EXEC
(error) EXECABORT Transaction discarded because of previous errors.

# 2、命令语法没错，但类型操作有误，则事务执行调用之后失败，无法进行事务回滚
   从我们执行了一个由于错误的value的key操作（例如对着String类型的value施行了List命令操作）
   处理方案：发生在EXEC之后的是没有特殊方式去处理的：即使某些命令在事务中失败，所有的其他命令都将会被执行。

案例
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set num 10
QUEUED
127.0.0.1:6379> LPOP num
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379> get num
"10"
127.0.0.1:6379>
```

思考为什么**redis**不支持回滚？

## pipeline 流水线

定义：批量执行redis命令，减少通信io

注意：此为客户端技术

示例

```
import redis

# 创建连接池并连接到redis
pool = redis.ConnectionPool(host = '127.0.0.1',db=0,port=6379)
r = redis.Redis(connection_pool=pool)

pipe = r.pipeline()
pipe.set('fans',50)
pipe.incr('fans')
pipe.incrby('fans',100)
pipe.execute()
```

性能对比

```
# 创建连接池并连接到redis
pool = redis.ConnectionPool(host = '127.0.0.1',db=0,port=6379)
r = redis.Redis(connection_pool=pool)

def withpipeline(r):
    p = r.pipeline()
    for i in range(1000):
        key = 'test1' + str(i)
        value = i+1
        p.set(key, value)
    p.execute()

def withoutpipeline(r):
    for i in range(1000):
        key = 'test2' + str(i)
        value = i+1
        r.set(key, value)
```

### python 操作 redis事务

```
with r.pipeline(transaction=true) as pipe
    pipe.multi()
    pipe.incr("books")
    pipe.incr("books")
    values = pipe.execute()
```

## watch - 乐观锁

作用： 事务过程中，可对指定key进行监听，命令提交时，若被监听key对应的值未被修改时，事务方可提交成功，否则失败

```
> watch books
OK
> incr books  # 被修改了
(integer) 1
> multi
OK
> incr books
QUEUED
> exec  # 事务执行失败
(nil)
```

**python操作watch**

```
#同时对一个账户进行操作，  当前余额 * 2
```

# ==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，
    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 |
| --- | --- |
| 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 |
| 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 |
| | |

```
# 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。
```

数据恢复（无需手动操作）

```
既有dump.rdb，又有appendonly.aof，恢复时找谁？
先找appendonly.aof
```

配置文件常用配置总结

```
# 设置密码
1、requirepass password
# 开启远程连接
2、bind 127.0.0.1 ::1 注释掉
3、protected-mode no  把默认的 yes 改为 no
# rdb持久化-默认配置
4、dbfilename 'dump.rdb'
5、dir /var/lib/redis
# rdb持久化-自动触发(条件)
6、save 900 1
7、save 300 10
8、save 60  10000
# aof持久化开启
9、appendonly yes
10、appendfilename 'appendonly.aof'
# aof持久化策略
11、appendfsync always
12、appendfsync everysec # 默认
13、appendfsync no
# aof重写触发
14、auto-aof-rewrite-percentage 100
15、auto-aof-rewrite-min-size 64mb
# 设置为从服务器
16、salveof <master-ip> <master-port>
```

**Redis**相关文件存放路径

```
1、配置文件：/etc/redis/redis.conf
2、备份文件：/var/lib/redis/*.rdb|*.aof
3、日志文件：/var/log/redis/redis-server.log
4、启动文件：/etc/init.d/redis-server
# /etc/下存放配置文件
# /etc/init.d/下存放服务启动文件
```

# ==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（Linux命令行实现1）

  redis-server --slaveof  --masterauth

```
# 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
127.0.0.1:6300> set mykey 123
(error) READONLY You can't write against a read only slave.
127.0.0.1:6300>
# 从服务器只能读数据，不能写数据
```

方式一（Redis命令行实现2）

```
# 两条命令
1、>slaveof IP PORT
2、>slaveof no one
```

示例

```
# 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)

```
# 每个redis服务,都有1个和他对应的配置文件
# 两个redis服务
  1、6379 -> /etc/redis/redis.conf
  2、6300 -> /home/tarena/redis_6300.conf

# 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Sentinel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
验证: sudo /etc/init.d/redis-sentinel stop
# 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

#4、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>

#如果master有密码，则需要添加该配置
sentinel auth-pass <master-name> <password>

#master多久失联才认为是不可用了，默认是30秒
sentinel down-after-milliseconds <master-name> <milliseconds>
```

**python获取master**

```
from redis.sentinel import Sentinel

#生成哨兵连接
sentinel = Sentinel([('localhost', 26379)], socket_timeout=0.1)

#初始化master连接
master = sentinel.master_for('tedu', socket_timeout=0.1, db=1)
slave = sentinel.slave_for('tedu',socket_timeout=0.1, db=1)

#使用redis相关命令
master.set('mymaster', 'yes')
print(slave.get('mymaster'))
```