链滴

redis【数据类型 & 通用指令 & jedis】

作者：haxLook

原文链接：https://ld246.com/article/1617261442495

来源网站：链滴

五大数据类型

redis中的key都为String类型，主要进行研究的就是Value的数据类型

String

基本操作

```
//设置String
set key value
mset key1 value1 key2 value2...
//设置生命周期
setex key seconds value

//得到String
get key
mget key1 key2...

//删除String
del key

//向字符串的后面追加字符，如果有就补在后面，如果没有就新建
append key value
```

string 类型数据的扩展操作

String作为数值的操作

```
//增长指令，只有当value为数字时才能增长
incr key
incrby key increment
incrbyfloat key increment

//减少指令，有当value为数字时才能减少
decr key
decrby key increment
```

- redis用于控制数据库表主键id，为数据库表主键 **提供生成策略**，保障数据库表的主键**唯一性**
- 此方案适用于所有数据库，且支持数据库集群
- 实际开发中应用在流水号的自增操作，可以更具redis的自增原理生成单据的流水编码。

说明

- string在redis内部存储默认就是一个 **字符串**，当遇到增减类操作incr，decr时会**转成数值型**进行算。
- redis所有的操作都是 **原子性**的，采用**单线程**处理所有业务，命令是一个一个执行的，因此无需考并发带来的数据影响。
- 注意：按数值进行操作的数据，如果原始数据不能转成数值，或超越了redis 数值上限范围，将报。 9223372036854775807（java中long型数据最大值，Long.MAX_VALUE）

指定生命周期

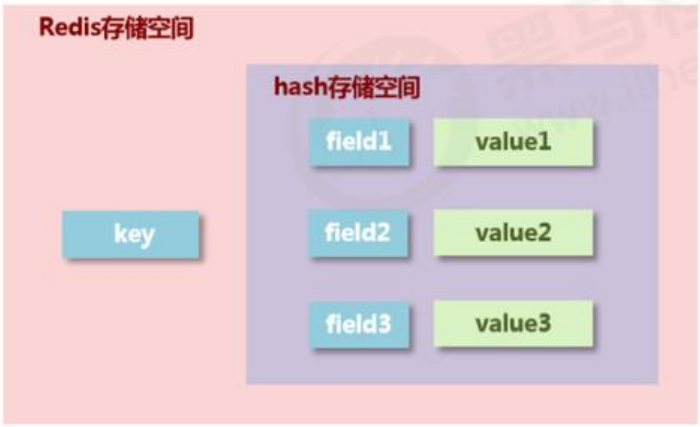
```
//设置数据的生命周期，单位 秒
setex key seconds value
//设置数据的生命周期，单位 毫秒
psetex key milliseconds value
```

- 数据库中的热点数据key命名惯例

	表名	主键名	主键值	字段名
eg1：	order	id	29437595	name
eg2：	equip	id	390472345	type
eg3：	news	id	202004150	title

Hash

- 新的存储需求：对一系列存储的数据进行编组，方便管理，典型应用存储对象信息
- 需要的存储结构：一个存储空间保存多个键值对数据
- hash类型：底层使用哈希表结构实现数据存储



hash存储结构优化

- 如果field数量较少，存储结构优化为类数组结构
- 如果field数量较多，存储结构使用HashMap结构

```
//插入（如果已存在同名的field，会被覆盖）
hset key field value
hmset key field1 value1 field2 value2...

//插入（如果已存在同名的field，不会被覆盖）
hsetnx key field value

//取出
hget key field
hgetall key

//删除
hdel key field1 field2...

//获取field数量
hlen key
```

```
//查看是否存在
hexists key field

//获取哈希表中所有的字段名或字段值
hkeys key
hvals key

//设置指定字段的数值数据增加指定范围的值
hincrby key field increment
hdecrby key field increment
```

hash 类型数据操作的注意事项

- hash类型下的value **只能存储字符串**，不允许存储其他数据类型，**不存在嵌套现象。**如果数据未取到， 对应的值为（nil）
- 每个 hash 可以存储 2^32 - 1 个键值
- hash类型十分贴近对象的数据存储形式，并且可以灵活添加删除对象属性。但hash设计初衷不是了存储大量对象而设计的，**切记不可滥用**，更**不可以将hash作为对象列表使用**
- hgetall 操作可以获取全部属性，如果内部field过多，遍历整体 **数据效率就很会低**，有可能成为据访问瓶颈

List

- 数据存储需求：存储多个数据，并对数据进入存储空间的顺序进行区分
- 需要的存储结构：一个存储空间保存多个数据，且通过数据可以体现进入顺序
- list类型：保存多个数据，底层使用双向链表存储结构实现
- **元素有序，且可重**

```
//添加修改数据,lpush为从左边添加，rpush为从右边添加
lpush key value1 value2 value3...
rpush key value1 value2 value3...

//查看数据, 从左边开始向右查看. 如果不知道list有多少个元素，end的值可以为-1,代表倒数第一个素
//lpush先进的元素放在最后,rpush先进的元素放在最前面
lrange key start end
//得到长度
llen key
//取出对应索引的元素
lindex key index

//获取并移除元素（从list左边或者右边移除）
lpop key
rpop key
```

拓展操作

```
//规定时间内获取并移除数据,b=block,给定一个时间，如果在指定时间内放入了元素，就移除
```

```
blpop key1 key2... timeout
brpop key1 key2... timeout//移除指定元素 count:移除的个数 value:移除的值。 移除多个相同元素，从左边开始移除
lrem key count value
```

注意事项

- list中保存的数据都是string类型的，数据总容量是有限的，最多2^32 - 1 个元素 (4294967295)。
- list具有索引的概念，但是操作数据时通常以 **队列**的形式进行入队出队(rpush, rpop)操作，或以**栈**形式进行入栈出栈(lpush, lpop)操作
- 获取全部数据操作结束索引设置为-1 (倒数第一个元素)
- list可以对数据进行分页操作，通常第一页的信息来自于list，第2页及更多的信息通过数据库的形式载

Set

- **不重复且无序**

基本操作

```
//添加元素
sadd key member1 member2...

//查看元素
smembers key

//移除元素
srem key member

//查看元素个数
scard key

//查看某个元素是否存在
sismember key member
```

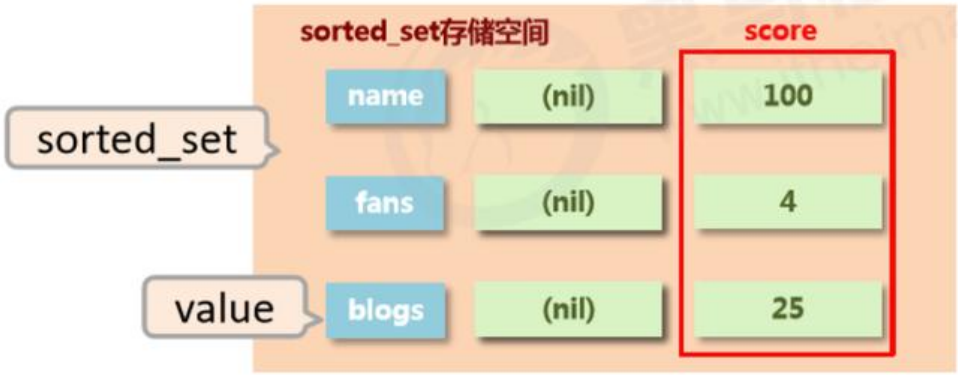
扩展操作

```
//从set中任意选出count个元素
srandmember key count

//从set中任意选出count个元素并移除
spop key count

//求两个集合的交集、并集、差集
sinter key1 key2...
sunion key1 key2...
sdiff key1 key2...

//求两个set的交集、并集、差集，并放入另一个set中
sinterstore destination key1 key2...
```

```
sunionstore destination key1 key2...
sdiffstore destination key1 key2...

//求指定元素从原集合放入目标集合中
smove source destination key
```

sorted_set

- **不重但有序（score）**
- 新的存储需求：数据排序有利于数据的有效展示，需要提供一种可以根据自身特征进行 **排序**的方式
- 需要的存储结构：新的存储模型，可以保存 **可排序**的数据
- sorted_set类型：在set的存储结构基础上添加可排序字段



基本操作

```
//插入元素, 需要指定score(用于排序)
zadd key score1 member1 score2 member2

//查看元素(score升序), 当末尾添加withscore时，会将元素的score一起打印出来
zrange key start end (withscore)
//查看元素(score降序), 当末尾添加withscore时，会将元素的score一起打印出来
zrevrange key start end (withscore)

//移除元素
zrem key member1 member2...

//按条件获取数据, 其中offset为索引开始位置，count为获取的数目
zrangebyscore key min max [withscore] [limit offset count]
zrevrangebyscore key max min [withscore] [limit offset count]

//按条件移除元素
zremrangebyrank key start end
zremrangebysocre key min max
//按照从大到小的顺序移除count个值
zpopmax key [count]
//按照从小到大的顺序移除count个值
zpopmin key [count]

//获得元素个数
```

```
zcard key

//获得元素在范围内的个数
zcount min max

//求交集、并集并放入destination中, 其中numkey1为要去交集或并集集合的数目
zinterstore destination numkeys key1 key2...
zunionstore destination numkeys key1 key2...
```

注意

- min与max用于限定搜索查询的 **条件**
- start与stop用于限定 **查询范围**，作用于索引，表示开始和结束索引
- offset与count用于限定查询范围，作用于查询结果，表示 **开始位置**和**数据总量**

```
//查看某个元素的索引(排名)
zrank key member
zrevrank key member

//查看某个元素索引的值
zscore key member
//增加某个元素索引的值
zincrby key increment membe
```

注意事项

- score保存的数据存储空间是64位，如果是整数范围是-9007199254740992~9007199254740992
- score保存的数据也可以是一个双精度的double值，基于双精度浮点数的特征， **可能会丢失精度** 使用时候要**慎重**
- sorted_set 底层存储还是 **基于set**结构的，因此数据**不能重复**，如果重复添加相同的数据，score将被反复覆盖，**保留最后一次**修改的结果

通用指令

1、Key的特征

- key是一个 **字符串**，通过key获取redis中保存的数据

2、Key的操作

基本操作

```
//查看key是否存在
exists key

//删除key
del key

//查看key的类型
```

type key

拓展操作（时效性操作）

//设置生命周期
expire key seconds
pexpire key milliseconds

//查看有效时间, 如果有有效时间则返回剩余有效时间, 如果为永久有效，则返回-1, 如果Key不存在则回-2
ttl key
pttl key

//将有时限的数据设置为永久有效
persist key

拓展操作（查询操作）

//根据key查询符合条件的数据
keys pattern

查询规则

查询模式规则

*　匹配任意数量的任意符号　　?　配合一个任意符号　　[]　匹配一个指定符号

keys *	查询所有
keys it*	查询所有以it开头
keys *heima	查询所有以heima结尾
keys ??heima	查询所有前面两个字符任意，后面以heima结尾
keys user:?	查询所有以user:开头，最后一个字符任意
keys u[st]er:1	查询所有以u开头，以er:1结尾，中间包含一个字母，s或t

拓展操作（其他操作）

//重命名key，为了避免覆盖已有数据，尽量少去修改已有key的名字，如果要使用最好使用renamenx
rename key newKey
renamenx key newKey

//查看所有关于key的操作, 可以使用Tab快速切换
help @generic

3、数据库通用操作

数据库

- Redis为每个服务提供有16个数据库，编号从0到15
- 每个数据库之间的数据相互独立

基本操作

```
//切换数据库 0~15
select index

//其他操作
quit
ping
echo massage
```

拓展操作

```
//移动数据, 必须保证目的数据库中没有该数据
mov key db

//查看该库中数据总量
dbsize
```

Jedis

1、Java操作redis的步骤

- 连接Redis

```
//参数为Redis所在的ip地址和端口号
Jedis jedis = new Jedis(String host, int port)
```

- 操作Redis

```
//操作redis的指令和redis本身的指令几乎一致
jedis.set(String key, String value);

//断开连接
jedis.close();
```

配置工具【springBoot整合】

```
redis.host=192.168.128.144
redis.port=6379
redis.maxTotal=30
redis.maxIdle=10
```

- 工具类

```
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;
import java.util.ResourceBundle;

/**
 * @author Chen Panwen
```

```
 * @data 2020/4/6 16:24
 */
public class JedisUtil {
    private static Jedis jedis = null;
    private static String host = null;
    private static int port;
    private static int maxTotal;
    private static int maxIdle;

    //使用静态代码块，只加载一次
    static {
        //读取配置文件
        ResourceBundle resourceBundle = ResourceBundle.getBundle("redis");
        //获取配置文件中的数据
        host = resourceBundle.getString("redis.host");
        port = Integer.parseInt(resourceBundle.getString("redis.port"));
        //读取最大连接数
        maxTotal = Integer.parseInt(resourceBundle.getString("redis.maxTotal"));
        //读取最大活跃数
        maxIdle = Integer.parseInt(resourceBundle.getString("redis.maxIdle"));
        JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
        jedisPoolConfig.setMaxTotal(maxTotal);
        jedisPoolConfig.setMaxIdle(maxIdle);
        //获取连接池
        JedisPool jedisPool = new JedisPool(jedisPoolConfig, host, port);
        jedis = jedisPool.getResource();
    }

    public Jedis getJedis() {
        return jedis;
    }
}
```