

## Redis

## Redis NoSql

111

- □□
- □□□□□□□□□□
- □□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□
- □□□□□□□

## Redis

```
## 127.0.0.1 0.0.0.0 IP
bind 0.0.0.0
## yes
daemonize yes
## Redis
requirepass 123456
##
port 6379
##
dir .
## 1 1 16 0-15
databases 1
## Redis
maxmemory 512mb
##
logfile "redis.log"
```

Page 10

```
redis-cli [options] [commands]
```

```
options
```

- -h `redis` IP
- -p `redis`
- -a `redis`

```
commands redis
```

- ping []redis[]pong
- help [] help commands []

1111

- KEYS [pattern] [count] [start 0] [stop -1]
- DEL [pattern] [count] [start 0] [stop -1]
- EXISTS [pattern]

- `EXPIRE` 对key设置过期时间key
- `TTL` 查看key过期时间
  - 返回-1表示永久
  - 返回-2表示key不存在

通配符

- `*` 匹配任意字符串
- `?` 匹配任意一个字符串
  - 匹配任意一个字符串

数据类型

redis支持key-value数据类型 `String` 字符串value数据类型

- String
- Hash
- List
- Set
- SortedSet
- GEO
- BitMap
- HyperLog

## Redis key

Redis key命名规范: `' : ' 数据库:库名:表名:id`

## String

字符串最大512MB

字符串最大32个字节

- string字符串
- int整数
- float浮点数

命令

- `SET` 设置字符串String
- `GET` 查看key的String value
- `MSET` 设置多个String
- `MGET` 查看多个key的String value
- `INCR` 对key自增1
- `INCRBY` 对key自增num `INCRBY num 2`
- `INCRBYFLOAT` 对key自增浮点数
- `SETNX` 设置String key不存在才设置
- `SETEX` 设置String key过期时间

## Hash

字符串value数据类型java的HashMap

 image-20250907134423516

命令

- HSET key field value 将哈希表key中的field的值设为value
- HGET key field 获取哈希表key中field的值
- HMSET 将哈希表key中的field的值设为value
- HMGET 获取哈希表key中field的值
- HGETALL 获取哈希表key中的所有field和value
- HKEYS 获取哈希表key中的所有key
- HVALS 获取哈希表key中的所有value
- HINCRBY 将哈希表key中的field的值增加increment
- HSETNX 将哈希表key中的field的值设为value，如果field不存在则设置，否则不设置

## List

java中的LinkedList

命令

- 添加
- 删除
- 获取
- 遍历

命令

- LPUSH key element 将元素添加到哈希表key的头部
- LPOP key 从哈希表key的头部移除元素并返回
- RPUSH key element 将元素添加到哈希表key的尾部
- RPOP key 从哈希表key的尾部移除元素并返回
- LRange key start end 获取哈希表key中从start到end的元素
- BLPOP 或 BRPOP 从哈希表key的头部或尾部移除元素并返回

## Set

java中的HashSet

命令

- 添加
- 删除
- 获取
- 遍历

命令

- SADD key member 将元素添加到集合key
- SREM key member 从集合key中移除元素
- SCARD key 获取集合key中的元素个数
- SISMEMBER 判断元素是否在集合key中
- SMEMBERS 获取集合key中的所有元素

- `SINTER key1 key2` 返回key1和key2的交集
- `SDIFF key1 key2` 返回key1和key2的差集
- `SUNION key1 key2` 返回key1和key2的并集

## SortedSet

SortedSet的每个元素都有一个score，score可以是任意类型的数字，也可以是字符串。

SortedSet的常用命令如下：

- `SCORE`
- `SCORES`
- `SCORECARD`

SortedSet的常用命令如下：

- `ZADD key score member` 向SortedSet中添加元素，score可以是任意类型的数字，也可以是字符串。
- `ZREM key member` 从SortedSet中移除元素。
- `ZSCORE key member` 返回SortedSet中元素的score。
- `ZRANK key member` 返回SortedSet中元素的排名。
- `ZCARD key` 返回SortedSet中元素的数量。
- `ZCOUNT key min max` 返回SortedSet中元素的个数，min和max可以是任意类型的数字，也可以是字符串。
- `ZINCRBY key increment member` 将SortedSet中元素的score增加increment。
- `ZRANGE key min max` 返回SortedSet中元素的列表，min和max可以是任意类型的数字，也可以是字符串。
- `ZRANGEBYSCORE key min max` 返回SortedSet中元素的列表，min和max可以是任意类型的数字，也可以是字符串。
- `ZDIFF`、`ZINTER`、`ZUNION` 返回SortedSet的差集、交集和并集。

SortedSet的常用命令如下：  
**Z**开头的命令，**REV**开头的命令。

## Jedis

Jedis是Java实现的Redis客户端。

Jedis的常用命令如下：

1. 连接Redis

```
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>3.7.0</version>
</dependency>
```

2. 设置密码

```
private Jedis jedis;

void setup(){
    // 连接Redis
    jedis = new Jedis("ip", 6379);
    // 设置密码
    jedis.auth("密码");
}
```

```

        // 关闭连接
        jedis.select(0);
    }
}

```

3. 测试
4. 部署

```

void tearDown(){
    if(jedis != null){
        // 关闭连接
        jedis.close();
    }
}

```

## Jedis连接池

Jedis连接池是Redis连接池的实现，它提供了Redis连接池的接口，Jedis连接池Jedis连接池

```

public class JedisConnectFactory{
    private static final JedisPool jedisPool;

    static{
        // 创建连接池配置
        JedisPoolConfig poolConfig = new JedisPoolConfig();
        // 设置最大连接数
        poolConfig.setMaxTotal(8);
        // 设置最大空闲连接数
        poolConfig.setMaxIdle(8);
        // 设置最小空闲连接数
        poolConfig.setMinIdle(0);
        // 设置最大等待时间
        poolConfig.setMaxWaitMillis(1000);
        // 创建连接池
        jedisPool = new JedisPool(poolConfig, "127.0.0.1", 6379, 1000);
    }

    public static Jedis getJedis(){
        return jedisPool.getResource();
    }
}

```

## Spring Data Redis

Spring Data Redis是Redis的Spring集成，它提供了Redis的Spring集成，Spring Data Redis

- Spring Data Redis 1.0.0 版本
- Spring Data Redis 1.0.0 版本
- Spring Data Redis 1.0.0 版本

## API

- redisTemplate.opsForValue() 返回 String 类型

- 

```
<!-- 依赖 -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
```

```
spring:
  redis:
    host:
    port:
    password:
    lettuce:
      pool:
        max-active:
        max-idle:
        min-idle:
        max-wait:
```

```
@Autowired
private RedisTemplate redisTemplate;
```

000

 image-20250908160754439

111

- 序列化
- 反序列化

RedisTemplate 的用法

- key 为 StringRedisSerializer 序列化 String 类型
- value 为 GenericJackson2JsonRedisSerializer 序列化 Json 类型
  - 序列化时，RedisTemplate 会自动调用 ObjectMapper 的 writeValueAsString 方法
  - 反序列化时，RedisTemplate 会自动调用 ObjectMapper 的 readValue 方法，将 String 类型的 key 和 value 转换为 Java 对象

```
// 测试
@Autowired
private StringRedisTemplate stringRedisTemplate;
// Json
private String final ObjectMapper mapper = new ObjectMapper();
void testStringTemplate(){
    // 序列化
    User user = new User("18");
    // 反序列化
    String json = mapper.writeValueAsString(user);
    // 序列化redis
    stringRedisTemplate.opsForValue().set("user:200", json);
    // 反序列化
    String val = stringRedisTemplate.opsForValue().get("user:200");
    // 反序列化
    User user1 = mapper.readValue(val, User.class);
}
```

- Spring 提供了 StringRedisTemplate 用于 key 和 value 都是 String 类型的 Redis 操作

Hash 类型

- .put() 设置
- .entries() 获取 value

Redis

Redis 的 Hash 类型

Redis 的 Hash 类型


- Redis 的 Hash 类型
  - Redis 的 Hash 类型
  - Redis 的 Hash 类型
  - Redis 的 Hash 类型
  - Redis 的 Hash 类型

image-20250919122313670

- 数据库操作
- 数据库操作
  - 数据库CRUD操作

数据库

数据库操作


image-20250919125659906

数据库

- 数据库
  - Redis数据库Null值TTL操作


image-20250919125921921

- 数据库
  - Redis数据库操作
  - Redis数据库操作

image-20250919130012267

数据库

数据库操作keyRedis数据库操作

image-20250920100653966


数据库

- 数据库keyTTL操作
- 数据库操作
- 数据库操作
- Redis数据库Redis操作

数据库

数据库key数据库操作key数据库操作

- 数据库操作
- 数据库key数据库操作redis数据库操作null


image-20250920102746920

数据库

- 数据库
  - 数据库操作
  - 数据库操作



- 数据库
  - 数据库查询返回结果设置**TTL**过期时间
  - 数据库查询返回结果设置过期时间，过期时间过期后，数据自动删除
  - 数据库查询返回结果设置过期时间，过期时间过期后，数据自动删除

 image-20250920102730424

Redis

Redis数据库操作

- 数据库操作
- 数据库id操作
- 数据库操作函数 Function
  - Function<ID,R> 数据库 ID 操作 R 操作
  - 数据库 .apply(id) 操作
  - 数据库操作函数 Function 操作 lambda 操作
- 数据库操作 Class<R> 操作

```
public <R,ID> R querywithPassThrough(String keyPrefix, ID id, Class<R> type,
Function<ID,R> dbFallback,Long time, TimeUnit timeUnit) {
    String key = keyPrefix + id;

    // Redis数据库
    String json =
stringRedisTemplate.opsForValue().get(RedisConstants.CACHE_SHOP_KEY + id);
    // 数据库返回结果设置Null操作
    if (StrUtil.isNotBlank(json)) {
        // 数据库
        return JSONUtil.toBean(json, type);
    }
    // 数据库
    if(json != null){
        return null;
    }
    // 数据库
    R r = dbFallback.apply(id);
    // 数据库
    if(r==null){
        stringRedisTemplate.opsForValue().set(RedisConstants.CACHE_SHOP_KEY +
id,"",RedisConstants.CACHE_NULL_TTL,TimeUnit.MINUTES);
        return null;
    }
    // Redis
    this.set(key, r, time, timeUnit);
    return r;
}
```

Redis ID操作

Redis ID 生成

- 时间
- 主机
- 进程
- 线程
- 随机数

Redis ID 生成 Redis 生成 ID 的格式

- 时间 1bit
- 主机 31bit
- 进程 32bit
- 线程 32bit
- 随机数  $2^{32}$  个 ID

image-20250920123640942

Redis

Redis **synchronized** 命令

Redis 命令

image-20250920155346096

Redis

image-20250920160104737

Redis 命令

Redis 命令

- 命令
  - SETNX lock thread1
  - EXPIRE lock 10
  - SET lock thread1 EX 10 NX
- 命令
  - DEL key
- 命令

Redis

- 命令 1
- 命令 2



RedissonRedisJavaJava

0000

## 1. Maven

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.13.6</version>
</dependency>
```

## 2. Redisson

- `@Configuration` `ApplicationContext`
- `@Bean` `ApplicationContext`Bean

```
@Configuration
public class RedissonConfig {

    @Bean
    public RedissonClient redissonClient() {
        // 配置
        Config config = new Config();

        config.useSingleServer().setAddress("redis://127.0.0.1:6379").setPassword("123456");
        // 创建RedissonClient
        return Redisson.create(config);
    }
}
```

## 3. □□□□


- o redissonClient.getLock("Lock") 返回
  - "Lock" 返回null
- o redissonClient.tryLock(1,10,TimeUnit.SECONDS) 返回
  - 1返回true表示成功
  - 10返回false表示失败
  - 返回false表示失败
- o redissonClient.unlock() 返回

## **Redisson**

□ □


## Redisson **Hash**



 image-20250921102428234

- redisson
  - redisson
    - redisson
      - redisson

- 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

 image-20250921103216710

## Redisson 和 WatchDog

在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

 image-20250921105223591

在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

- 在 Redis 中，我们使用 `PubSub` 来操作 Redis，而 `PubSub` 又依赖于 `Redis` 的 `PubSub` 功能。


在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

- 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。
- 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

## Redis 的 MultiLock

Redisson 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

- 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。
- 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

 image-20250921105936534

在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

- `@PostConstruct` 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

- 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。
- 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。
- 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

 image-20250921132004243

Redis 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

- `list` 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。
- `PubSub` 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。
- `Stream` 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

- 在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

在 Redis 中，我们使用 `Redisson` 来操作 Redis，而 `Redisson` 又依赖于 `Redis` 的 `PubSub` 功能。

####

- #####
- #####

## ####PubSub#####

#####channel#####channel#####

- SUBSCRIBE channel[channel] #####
- PUBLISH channel msg #####
- PSUBSCRIBE pattern[pattern] #####pattern#####

image-20250921133111828

####

- #####
- #####
- #####

## ####stream#####

#####

##### [] #####


- XADD #####
  - key #####
  - [NOMKSTREAM] #####
  - [MAXLEN] #####
  - [\*|ID] #####id##### '###-####'
    - #####redis#####
  - field value ##### Entry ▯ #####key-value #####
  - ##### XADD users \* name jack age 21
- XREAD #####
  - [COUNT] #####
  - [BLOCK milliseconds] #####
  - STREAMS key [key ...] #####key#####
  - ID [ID ...] #####id#####id#####
    - 0 #####
    - \$ #####
- XLEN #####

####

#####

#####

- #####
- #####
- #####

 image-20250921135538561

创建组

```
XGROUP CREATE key groupName ID [MKSTREAM]
```

- key 键名
- groupName 组名
- ID 组ID
  - 0 表示创建新组
  - \$ 表示继承父组ID
- MKSTREAM 是否创建流

删除组

```
XGROUP DESTORY key groupName
```

创建消费者

```
XGROUP CREATECONSUMER key groupName consumername
```

删除消费者

```
XGROUP DELCONSUMER key groupName consumername
```

读取消息

```
XREADGROUP GROUP group consumer [COUNT count] [BLOCK milliseconds] [NOACK] STREAMS key [key ...] ID [ID ...]
```

- group 组名
- consumer 消费者名
- count 读取消息数量
- BLOCK milliseconds 阻塞时间
- NOACK 是否不ack
  - 表示不ack，消息会一直留在pending-list中
- STREAMS key 键名
- ID 组ID
  - > 表示从下一个消息开始读取
  - 表示从id开始读取，id为0表示从pending-list中读取

ack消息

```
XACK key group ID[ID ...]
```

- key 键名
- group 组名
- ID 组ID

查看pending-list

```
XPENDING key group [[IDLE min-dile-time]] start end count [[consumer]]
```

- key 键名
- group 组名
- IDLE min-dile-time 空闲时间

- start end ID ID
  - - ID
  - + ID
- count
- consumer pending-list

## Feed


- Timeline
- 

Timeline


- 

 image-20250923094109650

- 

 image-20250923094049259

- - - 
    -
  - -

 image-20250923094355770

Feed Redis ZSet

- 
- 0

## GEO

GEO Geolocation Redis

- **GEOADD** longitude latitude member
- **GEODIST**
- **GEOHASH** member hash
- **GEOPOS** member
- **GEORADIUS** member
- **GEOSEARCH** member
- **GEOSEARCHSTORE** GEOSEARCH key

GEO ZSet

- score



- value

 image-20250923105504319

## BitMap

Redis String 的 BitMap

- 0 1
- bit 0 1

- **SETBIT**: (offset) 0 1
- **GETBIT**: (offset) bit
- **BITCOUNT**: BitMap 1 bit
- **BITFIELD**: (BitMap bit (offset))
  - type bit
  - offset bit
  - u i
- **BITFIELD\_RO**: BitMap bit
- **BITOP**: BitMap ()
- **BITPOS**: bit 0 1

## HyperLogLog

- UV 1
- PV PV

Redis HyperLogLog String HLL 16kb

- **PFADD** HLL
- **PFCOUNT** HLL
- **PFMERGE** HLL

- Redis
- Redis
- Redis
- Redis

## Redis

### RDB

RDB Redis

- save RDB
  - Redis RDB
- bgsave RDB

- Redis使用RDB持久化
- Redis使用RDB持久化
- Redis持久化

```
# 900秒保存key到RDB
save 900 1

# 使用RDB
save ""

# 压缩
rdbcompression yes

# RDB文件名
dbfilename dump.rdb

# 持久化目录
dir ./
```

## bgsave

- bgsave 使用fork创建子进程，子进程负责将RDB持久化
  - 子进程创建后，父进程继续运行
  - 子进程将内存中的key-value对写入RDB文件
  - 子进程完成后，父进程继续运行
- fork创建子进程
  - 子进程创建后，父进程继续运行
  - 子进程将内存中的key-value对写入RDB文件

 image-20250923160732193

## AOF持久化

Redis使用AOF持久化

AOF持久化redis.conf配置AOF

```
## 使用AOF
appendonly yes

## AOF文件名
appendfilename "appendonly.aof"
```

AOF持久化redis.conf配置AOF

- always 持久化
- everysec 持久化
- no 持久化



- slave 的 `offset` 等于 slave 的 `offset` 减去 master 的 `offset` 等于 **slave** 的 `offset` 减去 master 的 `offset`

那么

slave 的 `offset`

1. 那么
  - 那么 `replid` 那么
  - 那么 `replid` 那么
  - 那么 `continue`
2. 那么
  - 那么 `offset` 那么

 image-20250924122410199

**repl\_baklog** 那么


- 那么 `repl_baklog` 那么
- 那么 `repl_baklog` 那么

 image-20250924122831274

- 那么 `offset` 那么
- 那么 `slave` 那么

 image-20250924122705493

那么 `repl_baklog` 那么 `slave` 那么 `log` 那么


 image-20250924123049307

**Redis** 那么

Redis 那么

那么

- 那么 Sentinel 那么 master 那么 slave 那么
- 那么 master 那么 **Sentinel** 那么 **slave** 那么 **master** 那么 master 那么
- 那么 Sentinel 那么 **Redis** 那么 **Redis** 那么

 image-20250924161457495

Sentinel 那么 1 那么 `ping` 那么

- 那么 `quorum` 那么
- 那么 `quorum` 那么
  - `quorum` 那么

那么 **master**

配置参数

- 配置参数 master-require-passwd 配置密码
- 配置参数 slave-priority 配置主节点的优先级 0 表示最低
- 配置参数 slave-priority 配置主节点的 offset 配置主节点的偏移量
- 配置参数 id 配置主节点的 ID

配置

配置

- 配置参数 slaveof no one 配置主节点 master
- 配置参数 slave ip:port 配置主节点 master 的地址
- 配置参数 slaveof no one 配置主节点 master 的地址

image-20250924170328553

## Redis 配置

Redis 配置

配置

- 配置参数 master 配置主节点
- 配置参数 master 配置主节点
- 配置参数 master ping 配置主节点
- 配置参数 master 配置主节点

配置

- redis-cli --cluster 配置主节点

配置

Redis 配置 master 配置 0-16383 16384 配置

配置 key 配置主节点 redis 配置 key 配置主节点

- key 配置 { } 配置 1 配置 { } 配置
- key 配置 { } 配置 key 配置

配置 CRC16 配置 hash 配置 16384 配置 slot

配置

redis-cli --cluster add-node 配置

- new\_host:new\_port 配置
- existing\_host:existing\_port 配置 ip 配置
- --cluster-save 配置

redis-cli --cluster reshared 配置

- host:port 配置 ip 配置

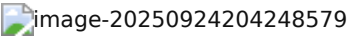
配置

配置 master 配置 slave 配置 master

集群

集群 failover 主 master 集群 failover 从 slave

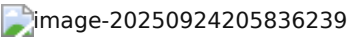
集群



集群

集群

- 集群
- **Nginx**
- **Redis**
- **JVM**
- 集群



**JVM**

**Nginx**

**Lua**

- 集群
  - nil 集群false
  - boolean 集群true
  - number 集群
  - string 集群
  - function 集群lua
  - table 集群lua
    - 集群 {} 集群
- 集群
  - 集群 local 集群
  - 集群 type 集群

```
-- 集群
local str = 'hello'

-- 集群
local num = 21

-- 集群
local flag = true

-- 集群
local arr = {'java','python','lua'}
```

```
-- 表table
local map = {name = 'jack',age = 21}
```

- 表table

```
-- 访问
print(arr[1])
-- 表map
print(map['name'])
print(map.name)
```

- 遍历

- 遍历表

```
-- 遍历 key-value table
local arr = {'java','python','lua'}
-- 遍历
for index,value in ipairs(arr) do
    print(index,value)
end
```

- 遍历table

```
-- 遍历table
local map = {name = 'jack',age = 21}
-- 遍历table
for key,value in pairs(map) do
    print(key,value)
end
```

- 函数

- 函数定义

```
function 函数名(argument 1,argument2 ... ,argumentn)
    return 返回值
end
```

- 逻辑运算符
  - and 与
  - or 或
  - not 非

```
if(条件)
then
    --[ 条件 true 执行的语句 --]
else
```

```
--[ 000000 false 00000000 --]
end
```

00

Redis000c000000

00000

00000SDS

000Redis0000000000000000

Redis0000000000000000000000SDS

- SDS00000000000000000000SDS0008bit00Redis000016bit032bit064bit0SDS

```
struct __attribute__((__packed__)) sdshdr8{
    uint8_t len; /* buf00000000000000000000 008bit 00000000255000 */
    uint8_t alloc; /* buf00000000000000000000 */
    unsigned char flags; /* 00SDS0000000000SDS00000 */
    char buf[];
}
```

- 0000 len 000000000000

image-20250911123033944

- 00000000000000000000
  - 0000SDS00000000000000000000
    - 000000000000000000001M00000000000000000000002+1
    - 000000000000000000001M0000000000000000000000+1M+1
    - +100000 \0 00SDS000000

IntSet

0Redis0set000

00000

```
typedef struct intset{
    uint32_t encoding; /* 00000000001600320064000 */
    uint32_t length; /* 0000 */
    int8_t contents[]; /* 000000000000000000000000encoding00 */
}intset;
```

00 encoding 00000000000000000000

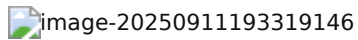


```

#define INTSET_ENC_INT16 (sizeof(int16_t)) /* 2bytes, 对应java short */
#define INTSET_ENC_INT32 (sizeof(int32_t)) /* 4bytes, 对应java int */
#define INTSET_ENC_INT64 (sizeof(int64_t)) /* 8bytes, 对应java long */

```

Redis 的 `intset` 数据结构存储 `contents` 数据



Redis 的 `int16_t` 数据结构存储 `INTSET_ENC_INT16` 数据

- `encoding` 4bytes
- `length` 4bytes
- `contents` 2\*3=6bytes

Redis 的 `startPtr + (sizeof(int16) * index)`

- `index` 0-5
- `startPtr` 指向 `contents` 的起始位置
- `sizeof()` 返回 `encoding` 的大小

**inset**

- Redis 的 `inset` 数据结构
  1. 存储 `contents` 数据
  2. 存储 `encoding` 和 `length` 信息
  3. 存储 `startPtr`
  4. `intset` 的 `encoding` 和 `length` 信息

**Dict**

Redis 的 `Dict`

- 存储 `contents` 数据
- 存储 `encoding` 和 `length` 信息
- 存储 `startPtr`

Redis

```

typedef struct dictht{
    // entry
    // table 指向 dictEntry*
    // 指向 dictEntry
    dictEntry **table;
    // 指向 dictEntry 2^n
    unsigned long size;
    // 指向 dictEntry size-1
    unsigned long sizemask;
    // entry
    unsigned long used;
}dictht;

```

字典

```
typedef struct dictEntry{
    void *key; // 键
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    }v; // 值
    // 指向下一个Entry
    struct dictEntry *next;
} dictEntry;
```

字典 Dict 是 Redis 字典 key 的哈希表 h 和 sizemask 的集合

- 字典的 size 是 2^n，sizemask 是 size-1
- 字典的 size 是 2 的幂次方

```
字典4
字典0000 0000 0100

字典7
字典0000 0000 0111

字典15
字典0000 0000 0011

字典3
```

字典

```
typedef struct dict{
    dictType *type; // dict 类型
    void *privdata; // 字典的私有数据
    dict ht[2]; // 字典的哈希表，用于 rehash
    long rehashidx; // rehash 的索引
    int16_t pauserehash; // rehash 是否暂停
} dict;
```

字典 Dict

字典的 rehash 操作

字典的 rehash 操作 (LoadFactor = used/size) 是字典的 rehash 操作

- 字典的 LoadFactor >= 1 时，字典会进行 BGSAVE 或 BGREWRITEAOF 操作
  - 字典的 CPU 使用率会很高
  - BGSAVE 是 Redis 的持久化操作，会将字典的数据写入 RDB 文件
    - RDB 文件是 Redis 的持久化文件
  - BGREWRITEAOF 是 Redis 的持久化操作，会将字典的数据写入 AOF 文件
    - AOF 文件是 Redis 的持久化文件

- 如果 `LoadFactor > 5`

那么就需要重新哈希 `LoadFactor < 0.1` 那么就不需要

## Dict 重哈希

那么就需要重新哈希 `size` 和 `sizemask` 那么就需要 `key` 和 `sizemask` 那么

那么就需要重新哈希 `key` 和 `sizemask` 那么就需要 `rehash`

1. 那么就需要 `realeSize` 那么就需要 `dictht`
  - 那么就需要 `size` 那么就需要 `dict.ht[0].used + 1` 那么就需要 `$2^n$`
  - 那么就需要 `size` 那么就需要 `dict.ht[0].used + 1` 那么就需要 `$2^n$` 那么就需要 `4`
2. 那么就需要 `realeSize` 那么就需要 `dictht` 那么就需要 `dict.ht[1]`
3. 那么就需要 `dict.rehashidx = 0` 那么就需要 `rehash`
4. 那么就需要 `dict.ht[0]` 那么就需要 `dictEntry` 那么就需要 `rehash` 那么就需要 `dict.ht[1]`
5. 那么就需要 `dict.ht[1]` 那么就需要 `dict.ht[0]` 那么就需要 `dict.ht[1]` 那么就需要 `dict.ht[0]` 那么就需要

## Dict 重哈希

那么就需要重新哈希 `rehash` 那么就需要

那么就需要 `Dict` 那么就需要 `rehash` 那么就需要


- 那么就需要 `dict.rehashidx` 那么就需要 `-1`
- 那么就需要 `dict.ht[0].table[rehashindex]` 那么就需要 `entry` 那么就需要 `rehash` 那么就需要 `dict.ht[1]` 那么就需要 `rehashidx++`
- 那么就需要 `dict.ht[0]` 那么就需要 `rehash` 那么就需要 `dict.ht[1]`

那么就需要


- 那么就需要 `rehash` 那么就需要 `dict.ht[1]` 那么就需要 `dict.ht[0]` 那么就需要 `dict.ht[1]` 那么就需要

## ZipList

那么就需要 `ZipList` 那么就需要

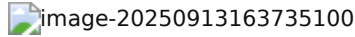
 image-20250912171110588

- `zlend` 那么就需要 `0xff`
- `zlbytes` 那么就需要
- `zltail` 那么就需要
- `zllen` 那么就需要 `entry` 那么就需要
- `entry` 那么就需要
  - `entry` 那么就需要 `16` 那么就需要

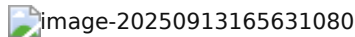
 image-20250913162441318

- `previous_entry_length` 那么就需要 `1` 那么就需要 `5`
  - 那么就需要 `254` 那么就需要 `1` 那么就需要

- [illegible]



- 01100000 content 000000 encoding 0000001000
  - 0000000000000000 encoding 0004000000000000



- contents [ ]

111

- ZipList 

## ZipList

- . entry[0] = 250; entry[1] = previous\_entry\_length + 1;
- . entry[0] = 254; entry[1] = previous\_entry\_length + 5;
- .

□ □

## QuickList

ZipList

- `list-max-ziplist-size` `QuickList``ZipList``entry`
- `list-compress-depth` `ZipList`

## QuickList

```
typedef struct quicklist{  
    // 0000  
    quickListNode *head;  
    // 0000  
    quickListNode *tail;  
    // 00ziplistEntry00  
    unsigned long count;  
    // ziplists00  
    unsigned long len;  
    // ziplistEntry00000-2  
    int fill : QL_FILL_BITS;  
    // 000000000000000000000000  
    unsigned int compress : QL_COMP_BITS;  
    // 00000000000000000000  
    unsigned int bookmark_count : QL_BM BITS;
```

```
    quicklistBookmark bookmarks[];  
}quicklist;
```


## QuickListNode

```
typedef struct quickListNode{  
    // 前一个节点  
    struct quickListNode *prev;  
    // 下一个节点  
    struct quickListNode *next;  
    // 指向ZipList的指针  
    unsigned char *zl;  
    // 指向ZipList的指针  
    unsigned int sz;  
    // 指向ZipList的entry  
    unsigned int count : 16;  
    // 指向ZipList的lzf的指针  
    unsigned int encoding : 2 ;  
    // 容器  
    unsigned int container : 2;  
    // 是否压缩  
    unsigned int recompress : 1;  
    unsigned int attempted_compress : 1;  
    unsigned int extra : 10;  
} quickListNode;
```

## SkipList

跳表

- 跳表
- 跳表的实现

 image-20250913171827926

跳表

```
typedef struct zskiplist{  
    // 头指针  
    struct zskiplistNode *head,*tail;  
    // 长度  
    unsigned long length;  
    // 跳表的层数  
    int level;  
}zskiplist;
```

跳表

```
typedef struct zskiplistNode{  
    sds ele; // 元素
```

```

double score; // 分数
struct zskiplistNode *backward; // 前驱
struct zskiplistLevel{
    struct zskiplistNode *forward; // 后继
    unsigned long span; // 跨度
}level []; // 层级
}zskiplistNode;

```

注意

- 分数score是double类型
- 分数score是double类型
- 分数score是double类型
- 分数score是double类型

## RedisObject

RedisObject是Redis的通用对象结构体

```

typedef struct redisObject{
    unsigned type:4; // 类型: string, hash, list, set, zset
    unsigned encoding:4; // 编码: 11
    unsigned lru:LRU_BITS; // 最近使用次数
    int refcount; // 引用计数
    void *str; // 指向字符串的指针
} robj;

```

RedisObject的type字段

- STRING 0 INT 1 RAW 2 EMBSTR 3
- LIST 4 ZipList 5 QuickList 6
- SET 7 intset 8 HT Hash Table 9 Redis Dict 10
- ZSET 11 ZipList 12 HT SkipList 13
- HASH 14 ZipList 15 HT 16

## String

- RAW是RedisObject的SDS结构体，最大512MB
- SDS是RedisObject的SDS结构体，最大44字节，RedisObject head指向SDS结构体

image-20250914145411630

- RedisObject的type字段为LONG\_MAX，表示是INT类型，RedisObject的ptr指向SDS结构体

## List

注意

- 3.2版本Redis的ZipList和LinkedList结构体，最大512字节，RedisObject的ptr指向ZipList结构体
- 3.2版本Redis的QuickList结构体，最大64字节，RedisObject的ptr指向QuickList结构体

image-20250914152319695

## Set

字典

- 无序集合
- 元素不可重复
- 元素为任意类型

实现

- Set 字典 **HT** (Dict) 字典 **key** 为元素 **value** 为 **null**
- 元素个数超过 `set-max-insert-entries` 字典 **IntSet** 实现



image-20250914153838878

## ZSet

字典 SortedSet 字典 **score** 为 **member**

- **score** 为浮点
- **member** 为字符串
- 字典 **member** 为 **score**

实现

- **SkipList** 字典 **score** 为 **ele**
- **HT** (Dict) 字典 **key** 为 **value**
- **RedisObject** **head** 字典 **SkipList** , 字典 **ZSet** 字典 **score**
  - 字典 **score**

```
typedef struct zset{
    // Dict
    dict *dict;
    // SkipList
    zskiplist *zsl;
} zset;
```



image-20250914161559970

- 字典 **zset\_max\_ziplist\_entries** 字典 **zset\_max\_ziplist\_value** 字典 **ZipList** 字典 **entry**
  - **ZipList** 字典 **score** 为 **element** 字典 **entry**
  - **score** 字典 **score** 字典 **score**



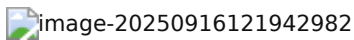
image-20250914162700600

## Hash

字典 **Zset** 字典







## 阻塞IO Nonblocking IO

阻塞IO recvfrom 阻塞等待数据到达

- 阻塞等待数据到达
- 阻塞等待数据到达
  - 阻塞等待数据到达CPU



## IO 复用 IO Multiplexing

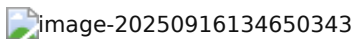
阻塞IO 阻塞IO 阻塞IO recvfrom 阻塞等待数据到达

- 阻塞 recvfrom 阻塞等待数据到达IO CPU 阻塞等待数据到达CPU
- 阻塞 recvfrom 阻塞等待数据到达

阻塞等待数据到达Socket 阻塞等待数据到达Socket 阻塞等待数据到达Socket 阻塞等待数据到达Socket 阻塞等待数据到达

阻塞

- 阻塞等待数据到达Socket 阻塞等待数据到达Socket
- 阻塞等待FD 阻塞等待Linux 阻塞等待Linux Socket
- 阻塞等待FD 阻塞等待FD
  - 阻塞等待 select 阻塞等待FD
  - FD 阻塞等待 readable \*\* 阻塞等待
  - 阻塞等待 recvfrom 阻塞等待



阻塞FD 阻塞等待数据到达

- select
- poll
- epoll

阻塞

- select poll 阻塞等待FD 阻塞等待FD 阻塞等待FD
- epoll 阻塞等待FD 阻塞等待FD

## select

Linux 阻塞IO 阻塞等待数据到达

```
// 阻塞等待 __fd_mask, 阻塞等待 long int (4 字节 32 bit)
typedef long int __fd_mask;

// fd_set 阻塞等待 fd 阻塞等待
typedef struct{
```

```


        // fds_bits long int 1024/32=32
        // 1024bit bit fd 0 10000000/100010000/00
        __fd_mask fds_bits[__FD_SETSIZE / __NFDBITS];
    } fd_set;

// select FD
int select(
    int nfd, // fd_set fd+1
    fd_set *readfds, // fd
    fd_set *writefds, // fd
    fd_set *exceptfds, // fd
    // null-0-0-0-0
    struct timeval *timeout
)

```

说明

1. fd\_set fd 1 nfd fd 1
2. select fd\_set
3. fd\_set fd fd\_set fd
4. fd fd
5. select fd fd\_set fd\_set
6. fd\_set fd

 image-20250916143656363

说明

- select
- select fd fd\_set
- fd\_set fd

**poll**

说明

```

// pollfd
#define POLLIN //
#define POLLOUT //
#define POLLERR //
#define POLLNVAL //fd

// pollfd
struct pollfd {
    int fd; //fd
    short int events; /* */
    short int revents; /* */
};

// poll
int poll(
    struct pollfd *fds, //pollfd

```

```

nfds_t nfds, // 文件描述符数量
int timeout // 超时时间
);

```

☐ ☐ ☐ ☐ ☐

1. `pollfd` 構造体を `fd` 構造体で初期化する
2. `poll` 関数で `pollfd` 構造体で初期化された `fd` 構造体で `poll` 関数を呼び出す
3. `fd` 構造体で初期化された `fd` 構造体で `poll` 関数を呼び出す
4. `pollfd` 構造体の `revents` 変数で `pollfd` 構造体の `fd` 変数で `poll` 関数を呼び出す
5. `pollfd` 構造体の `fd` 変数で `poll` 関数を呼び出す
6. `pollfd` 構造体の `fd` 変数で `poll` 関数を呼び出す

## epoll

```
select poll
```

```

struct eventpoll{
    struct rb_root rbr;           // 红黑树根节点FD(epitem)
    struct list_head rdlist; // 红黑树根节点FD(epitem)
}

// 1.创建eventpoll对象epfd
int epoll_create(int size);


// 2.将FD加入epoll对象ep_poll_callback函数
// callback函数fd加入rdlist
int epoll_ctl(
    int epfd, // epoll对象
    int op,    // 操作ADD,MOD,DEL
    int fd,    // 文件描述符
    struct epoll_event *event // 事件结构体
);

// 3.从rdlist中取出FD
int epoll_wait(
    int epfd, // eventpoll对象
    struct epoll_event *events, // 事件结构体FD
    int maxevents, // events数组大小
    int timeout // 超时时间-1表示无限等待
);

```

☐ ☐ ☐ ☐

1. 通过 `epoll_create` 函数创建 `eventpoll` 对象，并返回一个 `int` 类型的句柄。
2. 通过 `epoll_ctl` 函数向 `eventpoll` 对象中添加、删除或修改 `ep_poll_callback` 函数所监视的 `FD`。
3. 通过 `epoll_wait` 函数等待 `FD(epitem)` 上的 `eventpoll` 事件发生，并返回 `FD(epitem)` 上的事件。
4. 通过 `epoll_events` 函数返回 `FD(epitem)` 上的事件。

 image-20250916152045771

阻塞IO

FD阻塞IO epoll\_wait 阻塞IO

- **LevelTriggered** 阻塞LT FD阻塞IO
- **EdgeTriggered** 阻塞ET FD阻塞IO

阻塞IO

- ET阻塞IO LT阻塞IO
  - 阻塞IO FD LT阻塞IO
- ET阻塞IO IO FD阻塞IO LT阻塞IO


IO阻塞WEB阻塞IO

- 阻塞 serverSocket 阻塞 fd 阻塞 ssfd 阻塞IO
- 阻塞 serverSocket 阻塞 ssfd 阻塞IO
- 阻塞 socket 阻塞 fd 阻塞IO

 image-20250917182625014

阻塞IO **Signal Driven IO**

阻塞IO **SIGIO** FD阻塞IO **SIGIO**

 image-20250917183101977

阻塞IO

- 阻塞IO SIGIO
- 阻塞IO

阻塞IO **ASynchronous IO**

阻塞IO **API**

 image-20250917183543485

阻塞IO

- 阻塞IO

阻塞IO

IO阻塞IO

阻塞IO

Redis阻塞IO

- 阻塞Redis
- 阻塞Redis

阻塞Redis

- Redis 的持久化策略
- Redis 的复制策略
- Redis 的集群策略

## Redis 持久化

 image-20250927102155796

- Redis 的持久化策略
- Redis 的复制策略
  - Redis 的复制策略
  - Redis 的复制策略
  - Redis 的复制策略
  - Redis 的复制策略
  - Redis 的复制策略

## RESP 协议

Redis 的 CS 模型


1. 客户端发送请求
2. 服务器返回响应

Redis 的 RESP 协议


Redis 的 RESP 协议

Redis 的 RESP 协议

- Redis 的 RESP 协议
  - Redis 的 RESP 协议
- Redis 的 RESP 协议
  - Redis 的 RESP 协议
- Redis 的 RESP 协议
  - Redis 的 RESP 协议
- Redis 的 RESP 协议
  - Redis 的 RESP 协议

 image-20250918105136261

- Redis 的 RESP 协议

 image-20250918105158262

Redis

Redis

Redis expire 命令 Redis key 的 TTL

Redis **key-value** Dict

```
database Dict key-value key-TTL
```

```
typedef struct redisDb{
    dict *dict; // 字典key-value
    dict *expires; // 字典key-TTL
    TTL key
    dict *blocking_keys; //
    dict *ready_keys; //
    dict *watched_keys; //
    int id; // ID
    long long avg_ttl; // TTL
    unsigned long expires_cursor; // expire字典游标
    list *defrag_later; // 字典key
} redisDb;
```



1111

```
0000TTL000000000000key000000key0000000000000000
```

□□□□

```
00000000000000000000key00000000
```

- Redis `serverCron()` `key` **SLOW**
  - `server.hz` `10` `10` **100ms**
  - 25%**
  - `db` `db` `bucket` **20key**
- Redis `beforeSleep()` `key` **FAST**
  - `beforeSleep()` **FAST** **2ms**
  - 1ms**
  - `db` `db` `bucket` **20key**

□□□□□□

Redis Redis key

Redis8key

- noeviction 不删除任何key
- volatile-ttl 删除带TTL的key
- allkeys-random 随机删除key
- volatile-random 随机删除带TTL的key
- allkeys-lru 删除key中的LRU
- volatile-lru 删除带TTL的key中的LRU
- allkeys-lfu 删除key中的LFU
- volatile-lfu 删除带TTL的key中的LFU

Redis

- **LRU** **Least Recently Used** 最近最少使用
- **LFU** **Least Frequently Used** 最不经常使用 **key** 的 **255-LFU** 值

Redis 的 `RedisObject` 结构体

```
typedef struct redisObject{
    unsigned type:4; // string,hash,list,set,zset
    unsigned encoding:4; // 11
    unsigned lru:LRU_BITS; // LRU 24bit
                        // LFU 16bit
    int refcount; // 0
    void *str; //
} robj;
```

LFU 的 `key` 的 `lfu_decay_time`

1. `0~1` `R`
2. `1/(log * lfu_log_factor + 1)` `P` `lfu_log_factor` `10`
3. `R < P` `255`
4. `lfu_decay_time` `(1)`

Redis

- `P` `R < P`
- `key` `255`

Redis

image-20250918113723379

Redis

Redis

Redis `key`

- `[id]`
- `44` `String` `Raw`
- `key`

Redis `BigKey`

- `key` `value` `10kb`
- `key` `1000`

Redis

- `redis-cli --bigkeys`
  - `Key` `Top1` `big key`
- `scan` `key`
  - `Redis`
  - `keys * key` `Redis`

- `unlink` `big key`

Redis


Redis

Redis

- `mset` `hmset`
- `Pipeline`
- 

Redis

- `Spring` `slot`

 image-20250926120009590

Redis

Redis

- `Redis`
- `AOF`
- `slave` `RDB`
- `no-appendfsync-on-rewrite=yes` `AOF` `rewrite` `RDB` `fork` `AOF`

Redis

- `slowlog-log-slower-than`

Redis

- `slowlog-max-len`

Redis

- `slowlog len`
- `slowlog get[n]`
- `slowlog reset`

Redis

- `Redis`
- `Redis` `redis` `RDB`
- `ssh`

Redis

- `cluster-require-full-coverage no`
- `ping`
- `Redis`