

Redis

Redis分布式数据库 NoSql 数据库

特点

- 快速
- 分布式数据库
- 支持多种数据类型
- 支持多种编程语言
- 支持多种操作系统

Redis安装

```
## 安装Redis127.0.0.1安装Redis0.0.0.0安装RedisIP  
bind 0.0.0.0  
## 安装Redisyes安装Redis  
daemonize yes  
## 安装RedisRedis安装Redis  
requirepass 123456  
## 安装Redis  
port 6379  
## 安装Redis  
dir .  
## 安装Redis1安装Redis16安装Redis0-15  
databases 1  
## 安装RedisRedis安装Redis  
maxmemory 512mb  
## 安装RedisRedis安装Redis  
logfile "redis.log"
```

Redis命令

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

Redisoptions 选项

- -h 安装Redis安装RedisIP
- -p 安装Redis安装Redis
- -a 安装Redis安装Redis

commands Redis命令

- ping Redis安装Redispong
- help Redis help commands Redis

Redis命令

- KEYS Rediskey
- DEL RediskeyRediskey
- EXISTS Rediskey

- `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

匹配

- `*` 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

- 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` 获取key的String value
- `MSET` 设置多个String
- `MGET` 获取多个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存储为java的HashMap

 image-20250907134423516

命令

- `HSET key field value` 将哈希表key中的field字段的值设置为value
- `HGET key field` 获取哈希表key中field字段的值
- `HMSET key field value` 将哈希表key中的多个字段设置为指定的值
- `HMGET key field` 获取哈希表key中多个字段的值
- `HGETALL key` 获取哈希表key中的所有字段和值
- `HKEYS key` 获取哈希表key中的所有字段
- `HVALS key` 获取哈希表key中的所有值
- `HINCRBY key field increment` 将哈希表key中的field字段的值增加increment
- `HSETNX key field value` 只有当key不存在时，才将哈希表key中的field字段的值设置为value

List

Java中的LinkedList

命令

- `LPUSH key element`
- `LPOP key`
- `RPOP key`
- `LRANGE key start end`

命令

- `LPUSH key element` 将元素添加到哈希表key的列表的头部
- `LPOP key` 从哈希表key的列表的头部移除并返回元素
- `RPOP key` 从哈希表key的列表的尾部移除并返回元素
- `RPOP key` 从哈希表key的列表的尾部移除并返回元素
- `LRANGE key start end` 返回哈希表key的列表在start和end之间的元素
- `BLPOP key BRPOP key` 阻塞式地从哈希表key的列表的头部或尾部移除并返回元素

Set

Java中的HashSet

命令

- `SADD key member`
- `SREM key member`
- `SCARD key`
- `SISMEMBER key member`

命令

- `SADD key member` 将成员添加到哈希表key的集合中
- `SREM key member` 从哈希表key的集合中移除成员
- `SCARD key` 返回哈希表key的集合中的成员数量
- `SISMEMBER key member` 检查成员是否是哈希表key的集合中的成员
- `SMEMBERS key` 返回哈希表key的集合中的所有成员

- `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：

- `ZADD`
- `ZREM`
- `ZSCORE`

SortedSet的常用命令如下：

- `ZADD key score member` 向SortedSet中添加元素，score是元素的分数
- `ZREM key member` 从SortedSet中移除元素
- `ZSCORE key member` 返回SortedSet中元素的分数
- `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中分数在min和max之间的元素数量
- `ZINCRBY key increment member` 将SortedSet中元素的分数增加increment
- `ZRANGE key min max` 返回SortedSet中分数在min和max之间的元素
- `ZRANGEBYSCORE key min max` 返回SortedSet中分数在min和max之间的元素
- `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. 添加依赖

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

2. 配置Jedis

```
private Jedis jedis;

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

```

        // 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;
```

111

 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 RedisTemplate 的 set 方法将 value 序列化后存入 Redis
 - 反序列化时，RedisTemplate 会自动调用 RedisTemplate 的 get 方法将 value 从 Redis 中取出，并反序列化为 Java 对象

```
// 测试
@Autowired
private StringRedisTemplate stringRedisTemplate;
// Json
private String final ObjectMapper mapper = new ObjectMapper();
void testStringTemplate(){
    // 创建 User 对象
    User user = new User("张三",18);
    // 序列化
    String json = mapper.writeValueAsString(user);
    // 存入 Redis
    stringRedisTemplate.opsForValue().set("user:200",json);
    // 取出
    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
 - 用于存储键值对
 - 键值对的值可以是字符串、数字、布尔值、JSON 对象等
 - 键值对的键可以是字符串、数字、布尔值、JSON 对象等
 - 键值对的值可以是字符串、数字、布尔值、JSON 对象等

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数据库

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redis数据库null

image-20250920102746920

数据库

- 数据库
 - 数据库操作
 - 数据库操作

- 缓存
 - 缓存数据过期时间TTL
 - 缓存数据过期时间设置
 - 缓存数据过期时间设置

image-20250920102730424

缓存

Redis缓存

- 缓存数据过期时间TTL
- 缓存数据过期时间设置
- 缓存数据过期时间设置 Function
 - Function<ID,R> 缓存数据 ID 缓存数据 R
 - 缓存数据 .apply(id) 缓存数据
 - 缓存数据过期时间Function缓存数据lambda
- 缓存数据过期时间 Class<R> 缓存

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

    // 缓存redis
    String json =
stringRedisTemplate.opsForValue().get(RedisConstants.CACHE_SHOP_KEY + id);
    // 缓存数据过期时间Null
    if (StrUtil.isNotBlank(json)) {
        // 缓存数据
        return JSONUtil.toBean(json, type);
    }
    // 缓存数据
    if(json != null){
        return null;
    }
    // 缓存数据
    R r = dbFallback.apply(id);
    // 缓存数据
    if(r==null){
        stringRedisTemplate.opsForValue().set(RedisConstants.CACHE_SHOP_KEY +
id,"",RedisConstants.CACHE_NULL_TTL,TimeUnit.MINUTES);
        return null;
    }
    // 缓存redis
    this.set(key, r, time, timeUnit);
    return r;
}
```

缓存ID

Redisson是Redis的Java客户端，Java项目集成Redisson的步骤如下：

步骤

1. 添加Maven依赖

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

2. 配置Redisson客户端

- 使用 @Configuration 标注配置类
- 使用 @Bean 标注Redisson客户端

```
@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Redisson客户端
        return Redisson.create(config);
    }
}
```


3. 使用Redisson

- redissonClient.getLock("Lock") 获取锁
 - "Lock" 是锁的key
- redissonClient.tryLock(1,10,TimeUnit.SECONDS) 尝试获取锁
 - 1表示尝试获取锁的超时时间
 - 10表示锁的过期时间
 - 单位是秒
- redissonClient.unlock() 解锁

Redisson分布式锁


Redisson提供了分布式锁的实现

Redisson提供了Hash分布式锁的实现

image-20250921102428234


- Redisson提供了分布式锁的实现
 - Redisson提供了Hash分布式锁的实现
 - Redisson提供了RedissonLock实现
- Redisson提供了RedissonLock实现

- 在 Redis 中，使用 `WATCH` 命令可以实现乐观锁，但只能保证一个操作的原子性。

 image-20250921103216710

Redisson 中的 WatchDog

简介

 image-20250921105223591

简介

- Redisson 中的 **PubSub** 模式可以实现分布式锁。

简介

- Redisson 中的 **MultiLock** 模式可以实现分布式锁。
- Redisson 中的 **MultiLock** 模式可以实现分布式锁。

Redis 中的 MultiLock

Redisson 中的 MultiLock 模式可以实现分布式锁。

- Redisson 中的 **MultiLock** 模式可以实现分布式锁。
- Redisson 中的 **MultiLock** 模式可以实现分布式锁。

 image-20250921105936534

简介

- `@PostConstruct` 注解可以在类初始化时执行。

简介

Redisson 中的 MultiLock 模式可以实现分布式锁。

- Redisson 中的 **MultiLock** 模式可以实现分布式锁。
- Redisson 中的 **MultiLock** 模式可以实现分布式锁。
- Redisson 中的 **MultiLock** 模式可以实现分布式锁。

 image-20250921132004243

Redis 中的 MultiLock 模式可以实现分布式锁。

- Redis 中的 **MultiLock** 模式可以实现分布式锁。
- Redis 中的 **MultiLock** 模式可以实现分布式锁。
- Redis 中的 **MultiLock** 模式可以实现分布式锁。

简介

- Redis 中的 **MultiLock** 模式可以实现分布式锁。

Redis 中的 **MultiLock** 模式可以实现分布式锁。

Redis 中的 **MultiLock** 模式可以实现分布式锁。

####

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

####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 表示创建流组

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 表示不等待确认
 - 消费者 pending-list 中的消息在阻塞时间结束后，如果没有收到确认，则会被自动删除
- STREAMS key 键名
- ID 组ID
 - > 表示从下一个消息开始读取
 - 消费者id pending-list 中的消息在阻塞时间结束后，如果没有收到确认，则会被自动删除

发送消息

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

- key 键名
- group 组名
- ID 消息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

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使用RDB持久化

```
# 900秒保存一次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
 - 子进程负责将RDB持久化
 - 子进程负责将RDB持久化
 - 子进程负责将RDB持久化
- fork创建子进程，子进程负责将RDB持久化
 - 子进程负责将RDB持久化
 - 子进程负责将RDB持久化

 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

```
## 强制每次写都同步到磁盘AOF
appendfsync always

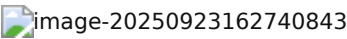
## 强制每秒同步一次AOF
appendfsync everysec

## 默认不强制同步AOF
appendfsync no
```

	RDB	AOF
持久化	持久化	持久化
持久化策略	快照持久化	日志持久化
持久化文件	持久化	持久化
持久化周期	秒	秒
持久化策略	快照持久化AOF	日志持久化
持久化策略	快照持久化CPU	日志持久化IO

Redis

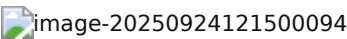
持久化策略



持久化

持久化策略

- 1. 持久化
 - 持久化策略
 - 持久化 Replication Id 持久化策略
 - 持久化策略持久化策略持久化 Replication Id 持久化 offset
- 2. 持久化
 - 持久化 bgsave 持久化RDB持久化RDB
 - 持久化RDB持久化策略 repl_backlog 持久化策略持久化策略
- 3. 持久化
 - 持久化 repl_backlog 持久化策略持久化策略



持久化策略

- Replication Id 持久化 replid 持久化策略id持久化策略
 - 持久化master持久化 replid 持久化slave持久化masetr持久化 replid
- offset 持久化策略持久化 repl_baklog 持久化策略持久化策略

- 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 slave-priority 配置参数 **0** 配置参数
- 配置 slave-priority 配置参数 offset 配置参数
- 配置参数 id 配置参数

配置

配置

- 配置参数 slaveof no one 配置参数 master
- 配置参数 slave ip:port 配置参数 **slave** 配置参数 **master** 配置参数 master 配置参数
- 配置参数 slave 配置参数 master 配置参数 slave

 image-20250924170328553

Redis 配置

配置 Redis 配置参数

配置

- 配置参数 master 配置参数 master 配置参数
- 配置 master 配置参数 slave 配置参数
- master 配置参数 ping 配置参数
- 配置参数 master 配置参数

配置

- redis-cli --cluster 配置参数 redis

配置

Redis 配置参数 master 配置参数 **0-16383** **16384** 配置参数

配置 key 配置参数 redis 配置参数 **key** 配置参数

- key 配置参数 { } 配置参数 1 配置参数 { } 配置参数
- key 配置参数 { } 配置参数 key 配置参数

配置参数 CRC16 配置参数 hash 配置参数 16384 配置参数 slot

配置

redis-cli --cluster add-node 配置参数

- new_host:new_port 配置参数
- existing_host:existing_port 配置参数 ip 配置参数
- --cluster-save 配置参数

redis-cli --cluster reshared 配置参数

- host:port 配置参数 ip 配置参数

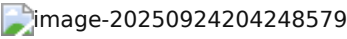
配置

配置参数 master 配置参数 **slave** 配置参数 **master**

集群

集群 failover 主 master 集群 failover 从 slave

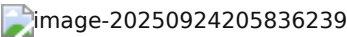
集群



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 时执行的语句 --]
else
```

```
--[ 是否为 false 是否为 --]
end
```

二

Redis的c语言

二

二SDS


二Redis的二进制字符串

Redis的二进制字符串SDS

- SDS的二进制字符串SDS的二进制字符串Redis的二进制字符串16bit32bit64bitSDS

```
struct __attribute__((packed)) sdshdr8{
    uint8_t len; /* buf的二进制字符串 8bit 的二进制字符串255 */
    uint8_t alloc; /* buf的二进制字符串 */
    unsigned char flags; /* SDS的二进制字符串SDS的二进制字符串 */
    char buf[];
}
```

- 二 len 的二进制字符串

image-20250911123033944

- 二SDS的二进制字符串
 - 二SDS的二进制字符串
 - 二SDS的二进制字符串1M的二进制字符串2+1
 - 二SDS的二进制字符串1M的二进制字符串+1M+1
 - +1的二进制字符串 \0 二SDS的二进制字符串

IntSet

Redis的set的二进制字符串

二

```
typedef struct intset{
    uint32_t encoding; /* 二SDS的二进制字符串163264 */
    uint32_t length; /* 二SDS的二进制字符串 */
    int8_t contents[]; /* 二SDS的二进制字符串encoding */
}intset;
```

二 encoding 二SDS的二进制字符串


```

#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` 0bytes
- `startPtr` 0bytes
- `sizeof()` 0bytes `encoding` 0bytes

inset

- Redis 的 `inset` 数据结构
 1. 0bytes
 2. 0bytes
 3. 0bytes
 4. `intset` 4 `encoding` 4 `length` 0bytes

Dict

Redis 的

- Redis 的
- Redis
- Redis

Redis

```

typedef struct dictht{
    // entry
    // table 指向 dictEntry*
    // 指向 dictEntry
    dictEntry **table;
    // 指向 2^n
    unsigned long size;
    // 指向 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dictEntry
    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;
```

Dict 字典

字典的 rehash 过程

Dict 字典的 rehash 过程 (LoadFactor = used/size) 字典的 rehash 过程

- 字典 LoadFactor >= 1 字典的 rehash 过程
 - 字典的 rehash 过程 CPU 消耗较大
 - BGSAVE 字典 Redis 字典的 rehash 过程 RDB 字典
 - RDB 字典的 rehash 过程
 - BGREWRITEAOF 字典 AOF 字典的 rehash 过程
 - AOF 字典的 rehash 过程

- 当 `LoadFactor > 5`

当 `LoadFactor < 0.1` 时，进行 rehash

Dict Rehash

当 `size < sizemask` 时，进行 rehash

当 `key` 不存在时，进行 rehash

1. 计算 `realSize`
 - `dictht[0].used + 1` 个 `2^n`
 - `dictht[0].used + 1` 个 `2^n` 的 `4` 倍
2. 计算 `realSize` 并更新 `dictht` 为 `dict.ht[1]`
3. 将 `dict.rehashidx = 0` 进行 rehash
4. 将 `dict.ht[0]` 中的 `dictEntry` 移动到 `dict.ht[1]`
5. 将 `dict.ht[1]` 中的 `dict.ht[0]` 移动到 `dict.ht[1]` 中

Dict 重哈希

当 `rehash` 时，进行 rehash

当 `Dict` 重哈希时，进行 rehash


- 当 `dict.rehashidx` 为 `-1`
- 当 `dict.ht[0].table[rehashindex]` 为 `entry` 时，将 `dict.ht[1]` 中的 `rehashidx++`
- 当 `dict.ht[0]` 中的 `rehash` 移动到 `dict.ht[1]`

当


- 当 `rehash` 时，将 `dict.ht[1]` 中的 `dict.ht[0]` 移动到 `dict.ht[1]` 中

ZipList

当 `ZipList` 时，进行 rehash

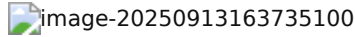
 image-20250912171110588

- `zlend` 为 `0xff`
- `zlbytes` 为 `0`
- `zltail` 为 `0`
- `zllen` 为 `entry` 的个数
- `entry` 的个数
 - `entry` 的个数不超过 `16` 个

 image-20250913162441318

- `previous_entry_length` 为 `1` 到 `5` 之间的数
 - 当 `previous_entry_length` 为 `254` 时，表示 `1` 个 `entry`

- [illegible]



- 01100000 content 000000 encoding 0000001000
 - 0000000000000000 encoding 000**4**00000000000



- contents []

111

- ZipList

ZipList

- . []N[]250-253[]entry[]previous_entry_length []1[]
- . []254[]entry[]previous_entry_length []5[]
- . []

[illegible]

QuickList

ZipList

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

QuickList

```
typedef struct quicklist{  
    // 00000  
    quickListNode *head;  
    // 00000  
    quickListNode *tail;  
    // 00ziplistEntry000  
    unsigned long count;  
    // ziplists000  
    unsigned long len;  
    // ziplistEntry000000-2  
    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头  
    unsigned char *zl;  
    // 指向ZipList尾  
    unsigned int sz;  
    // 指向ZipList entry  
    unsigned int count : 16;  
    // 指向ZipList 2个lzf头  
    unsigned int encoding : 2 ;  
    // 容器大小  
    unsigned int container : 2;  
    // 是否压缩  
    unsigned int recompress : 1;  
    unsigned int attempted_compress : 1;  
    unsigned int extra : 10;  
} quickListNode;
```

SkipList

跳表结构

- 跳表头
- 跳表节点

 image-20250913171827926

跳表

```
typedef struct zskiplist{  
    // 跳表头  
    struct zskiplistNode *head,*tail;  
    // 长度  
    unsigned long length;  
    // 跳表头1  
    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; // 最近最少使用
    int refcount; // 引用计数
    void *str; // 指向字符串
} 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 RedisDict 10
- ZSET 11 ZipList 12 HT SkipList 13
- HASH 14 ZipList 15 HT 16

String

- RAW是RedisObject的SDS类型，最大512MB
- SDS是RedisObject的SDS类型，最大44字节，RedisObject head指向SDS的起始位置

image-20250914145411630

- RedisObject的type字段为LONG_MAX时，表示RedisObject指向RedisDict的ptr，指向RedisDict的SDS

List

RedisObject的type字段

- 3.2版本Redis的ZipList和LinkedList，最大512字节，RedisObject的type字段为64时，表示RedisObject指向LinkedList
- 3.2版本Redis的QuickList和List

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 Set 转为 **IntSet**



image-20250914153838878

ZSet

有序字典 SortedSet 元素为 **score** 成员 **member**

- **score** 分数
- **member** 成员
- 字典 member 为 score

实现

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

```
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
- **ZipList** 字典 **score** **element** 字典 **entry**
- **score** 字典 **score** 字典 **score**

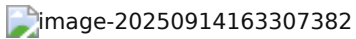


image-20250914162700600

Hash

字典字典 Zset 字典

- Hash `entry` ZipList `entry` field `value`



- `HashDict`
 - `ZipList` `hash-max-ziplist-entries`
 - `ZipList` `entry` `hash-max-ziplist-value`



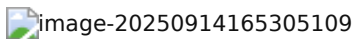
Redis

□□□□□□□□

32□□□□□□□□\$2^{\{32\}}\$□□□□4GB

[illegible]

- [illegible]

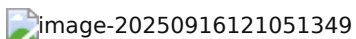


- [illegible]



IO Linux IO

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



IO

- □□□□□□□□□□
- □□□□□□□□□□□□□□□□

IO□□□□□□□□□□

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



IO Blocking IO

IO

- [illegible]



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多路复用 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 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 0/1
    __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
    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

```

```

    nfdst_t nfdst, //nfdst
    int timeout //timeout
);

```

pollfd

1. pollfd 的 fd 成员变量
2. poll 的 pollfd 成员变量
3. poll fd 成员变量
4. pollfd 的 revents 成员变量 0 表示 pollfd 成员变量 fd 为 n
5. pollfd n 成员变量 0
6. pollfd 的 pollfd 成员变量 fd

epoll

select/poll 的 epoll 成员变量

```

struct eventpoll{
    struct rb_root rbr; // rb_root 成员变量 FD(epitem)
    struct list_head rdlist; // list_head 成员变量 FD(epitem)
}

// 1. eventpoll 成员变量 epfd
int epoll_create(int size);

// 2. FD 成员变量 epoll 成员变量 ep_poll_callback 成员变量
// callback 成员变量 fd 成员变量 rdlist 成员变量
int epoll_ctl(
    int epfd, // epoll 成员变量
    int op, // 成员变量 ADD,MOD,DEL
    int fd, // fd 成员变量
    struct epoll_event *event // 成员变量
);

// 3. rdlist 成员变量 FD 成员变量
int epoll_wait(
    int epfd, // eventpoll 成员变量
    struct epoll_event *events, // event 成员变量 FD
    int maxevents, // events 成员变量
    int timeout // 成员变量 -1 表示 0 成员变量 0 表示 0
);


```

poll

1. poll_create 成员变量 eventpoll 成员变量
2. poll_ctl 成员变量 eventpoll 成员变量 ep_poll_callback 成员变量
3. poll_wait 成员变量 FD(epitem) 成员变量 eventpoll 成员变量 FD(epitem) 成员变量
4. poll_events 成员变量 FD(epitem) 成员变量

- Redis 数据库的持久化策略
- Redis 数据库的备份与恢复
- Redis 数据库的监控与运维

Redis 简介

 image-20250927102155796

- Redis 是什么
- Redis 的特点
 - Redis 是开源的
 - Redis 是跨平台的
 - Redis 是高性能的
 - Redis 是支持多种数据类型的
 - Redis 是支持集群的

RESP 协议

Redis 使用 CS 模型进行通信


1. 客户端发送请求
2. 服务器返回响应

Redis 使用 RESP 协议进行通信


Redis 使用 **RESP** 协议

RESP 协议是 Redis 的通信协议

- 请求以 '+' 开头，以 CRLF ("\r\n") 结尾
 - 请求的格式为 "+command\r\n"
- 响应以 '-' 开头，以 CRLF 结尾
- 响应的格式为 "-value\r\n"
- 响应的大小限制为 512MB
 - 响应的大小不能超过 512MB
 - 响应的大小不能超过 0MB
 - 响应的大小不能超过 1MB

 image-20250918105136261

- Redis 使用 '*' 表示批量操作

 image-20250918105158262

Redis

Redis

Redis 使用 expire 命令设置 key 的 TTL

DB

Redis数据库是 **key-value** 数据库，key和value都是字符串 Dict 结构

数据库database是Dict 结构，key-value 结构，key-TTL

```
typedef struct redisDb{
    dict *dict; // key-value
    dict *expires; // 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 dict
    list *defrag_later; // key
} redisDb;
```



image-20250918105635565

Redis

数据库TTL是key-value结构，key是字符串，value是字典

Redis

数据库是key-value结构

- Redis数据库 serverCron() 是key-value结构 **SLOW**
 - server.hz 是1000000/1000=1000，1000ms
 - 是25%
 - db 是 db 桶 bucket 是20key
- Redis数据库 beforeSleep() 是key-value结构 **FAST**
 - beforeSleep() 是FAST，2ms
 - 是1ms
 - db 是 db 桶 bucket 是20key

Redis

数据库Redis是key-value结构，Redis是key-value结构

Redis是8key-value结构

- noeviction 是key-value结构
- volatile-ttl 是TTLkey-value结构
- allkeys-random 是key-value结构
- volatile-random 是TTLkey-value结构
- allkeys-lru 是key-value结构
- volatile-lru 是TTLkey-value结构
- allkeys-lfu 是key-value结构
- volatile-lfu 是TTLkey-value结构

Redis 数据结构

- **LRU** Least Recently Used 最近最少使用
- **LFU** Least Frequently Used 最不经常使用 **key** 的访问次数 **255-LFU** 的访问次数

Redis 的 RedisObject 结构体

```
typedef struct redisObject{
    unsigned type:4; // 数据类型 string, hash, list, set, zset
    unsigned encoding:4; // 编码方式 11
    unsigned lru:LRU_BITS; // LRU 最近最少使用 24bit
                          // LFU 最不经常使用 8bit
    int refcount; // 引用计数
    void *str; // 指向字符串的指针
} robj;
```

LFU 的访问次数

1. 从 0~1 中随机一个 R
2. 计算 $1 / (\text{访问次数} * \text{lfu_log_factor} + 1)$ 得到一个 P
3. 如果 $R < P$ 则访问次数 +1
4. 如果访问次数达到 **lfu_decay_time** 则访问次数 * (0.1)

Redis

- 访问次数 P 的计算
- $R < P$ 的判断
- **key** 的访问次数

Redis

image-20250918113723379

Redis

Redis

Redis 的 key 结构体

- 结构体成员: [id]
- 44 字节的 String 类型 **Raw**
- 指向字符串的指针

Redis 的 BigKey

- key 的 value 大小 **10kb**
- key 的访问次数 **1000**

Redis

- 使用 `redis-cli --bigkeys`
 - 显示 **Top 1 big key**
- 使用 `scan` 命令遍历 key
 - 使用 **Redis** 命令
 - 使用 `keys *` 命令遍历 **Redis** 中的 key

- `unlink` 删除big key

Redis

Redis

Redis

- `mset` 和 `hmset` 批量设置
- `Pipeline` 批量操作
-

Redis

- `Spring` 使用 `slot`

 image-20250926120009590

Redis

Redis

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

Redis

- `slowlog-log-slower-than` 配置

Redis

- `slowlog-max-len` 配置

Redis

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

Redis

- `Redis` 集群
- `Redis` 集群 `redis` 持久化 `RDB` 持久化
- `ssh` 连接

Redis

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