

redis

Redis(Remote Dictionary Server) 是一个使用 C 语言编写的，开源的（BSD 许可）高性能非关系型（NoSQL）的键值对数据库。

Redis 可以存储键和五种不同类型的值之间的映射。键的类型只能为字符串，值支持五种数据类型：字符串、列表、集合、散列表、有序集合。

数据存储在内存，读写速度快，广泛被用于缓存，分布式锁，除此之外，Redis 支持事务、持久化、LUA 脚本、LRU 驱动事件、多种集群方案。

▼ 优缺点

▼ 优点

- 读写性能快
- 支持数据持久化
- 支持事务
- 数据结构丰富
- 支持主从复制

▼ 缺点

- 容量受限制于物理内存限制，不能用作海量数据的高性能读写
- ▼ 不具备自动容错和恢复功能
 - 读取写入失败
 - 手动重启或者切换前端IP
- 系统可用性低效：主机宕机，部分数据未能及时同步到从机，切换IP后还会引入数据不一致问题
- 难支持在线扩容

▼ 为什么快

- 基于内存操作
- 数据结构简单，操作性能高
- 采用单线程，避免上下文切换带来的CPU消耗和锁的竞争
- 使用多路IO服用模型，非阻塞
- 使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了 VM 机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

▼ 数据类型

— string

▼ string

▼ SDS

- 底层实现
- simple dynamic string
- ▼ 作用
 - 字符KV
 - ▼ 缓冲区

- AOF缓冲区
- 客户端输入缓冲区

▼ 存储

- ▼ 字符串
 - 可操作一部分
 - 整数
 - 浮点数
- 最大容量512M

▼ 场景

- 简单KV缓存
- 验证码
- 配置信息

▼ list

- 列表
- 底层是双向链表
- 固定数据
- ▼ 操作
 - 两端插入
 - 弹出
 - 单个或多个元素的修剪，保留一个范围的元素

▼ 场景

- 字典表
- ▼ 消息队列

- 少重

- 消息会丢失

▼ set

- 无序集合

▼ 操作

- 单元素增, 删, 改, 查
- 随机访问

- 交集

- 并集

- 差集

- 检查是否在集合中

▼ 场景

- 交集/并集/差集场景的应用

▼ zset

- 有序集合

- 跳跃表

▼ 操作

- 增, 改, 查, 删

- 排序

- 根据分值获取元素, 单个或多个

▼ 场景

- 排序

- 排名

▼ hash

- 包含键值对的无序散列表

- key唯一

▼ 操作

- 增, 查, 删

- 所有的KV

- 检查某个Key是否存在

■ 应用场景

- ▼ 应用场景
 - 存储详情
- ▼ 数据结构

▼ SDS

```
struct sdshdr {  
    // 记录 buf 数组中已使用字节的数量  
    // 等于 SDS 所保存字符串的长度  
    int len;  
  
    // 记录 buf 数组中未使用字节的数量  
    int free;  
  
    // 字节数组, 用于保存字符串  
    char buf[];  
};
```

▼ len

- buf长度

▼ free

- buf中未使用的字节数量

▼ buf[]

- 字节数组

▼ 链表

```
typedef struct ListNode {  
    // 前置节点  
    struct ListNode *prev;  
  
    // 后置节点  
    struct ListNode *next;  
  
    // 节点的值  
    void *value;  
}
```

adlist.h/list 来持有链表

```
listNode typedef struct List {  
    // 表头节点  
    ListNode *head;  
  
    // 表尾节点  
    ListNode *tail;  
  
    // 链表所包含的节点数量  
    unsigned long len;  
  
    // 节点值复制函数  
    void (*copy)(void *ptr);  
  
    // 节点值释放函数  
    void (*free)(void *ptr);  
  
    // 节点值对比函数  
    int (*match)(void *ptr, void *key);  
} list;
```

- adlist.h/listNode

▼ 字典

```
typedef struct dict {  
    // 类型特定函数  
    dictType *type;  
  
    // 私有数据  
    void *privdata;  
  
    // 哈希表  
    dictHashTable *ht;  
  
    // rehash 索引  
    // 当 rehash 不在进行时，值为 -1  
    int rehashidx; /* rehashing not in progress if rehashidx == -1 */  
}; dict;  
  
typedef struct dictht {  
    // 哈希表数组  
    dictEntry **table;  
  
    // 哈希表大小  
    unsigned long size;  
  
    // 哈希表大小掩码，用于计算索引值  
    // 总是等于 size - 1  
    unsigned long sizemask;  
  
    // 该哈希表已有节点的数量  
    unsigned long used;  
}; dictht;  
// 节点  
  
typedef struct dictEntry {  
    // 键  
    void *key;  
  
    // 值  
    union {  
        // 基本类型  
        char c;  
        short s;  
        int i;  
        long l;  
        float f;  
        double d;  
  
        // 指向字典的指针  
        dict *dict;  
  
        // 指向哈希表的指针  
        dictht *ht;  
  
        // 指向链表的指针  
        dictList *list;  
    } value;  
}; dictEntry;
```

```
    void *val;
    int64_t u64;
    int64_t s64;
}

// 指向下个哈希表节点，形成链表
struct dictentry *next;
}

dictentry;
```

▼ 底层哈希表实现

- dict.h/dictht

▼ 字段

▼ **table

▼ 数组

- 元素

- 存储KV地方

▼ size

- table大小

▼ sizemask

- size-1

- 用于计算索引值

▼ used

- 已有节点数量

▼ 节点

▼ 字段

▼ *key

- 键

▼ union v 值

- *val

- u64

- s64

▪ *next

▼ 操作结构dict

▼ 字段

```
▼ *type
typedef struct dictType {

    // 计算哈希值的函数
    unsigned (*hashFunction)(const void *key);

    // 复制键的函数
    void (*keyDup)(void *privdata, const void *key);

    // 复制值的函数
    void (*valDup)(void *privdata, const void *obj);

    // 对比键的函数
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);

    // 销毁键的函数
    void (*keyDestructor)(void *privdata, void *key);

    // 销毁值的函数
    void (*valDestructor)(void *privdata, void *obj);

} dictType;
```

- 保持则privdata操作函数
- ▼ privdata
 - 保存传给操作函数的可选参数
- ▼ ht[2]
 - 元素指向哈希表
 - ▼ 2个元素
 - ▼ 0
 - 一般使用
 - ▼ 1
 - rehash使用
- ▼ rehashindex

- 记录rehash进度
 - 没有rehash :-1
- ▼ 哈希算法
 - 计算哈希值hash
 - ▼ 计算索引值
 - hash&ht[x].sizemask
- ▼ rehash 扩缩容
 - ▼ 负载因子
 - load_factor = ht[0].used / ht[0].size
 - ▼ 触发时机
 - 服务器目前没有在执行 BGSAVE 命令或者 BGREWRITEAOF 命令，并且哈希表的负载因子大于等于 1
 - 服务器目前正在执行 BGSAVE 命令或者 BGREWRITEAOF 命令，并且哈希表的负载因子大于等于 5；
 - ▼ 流程
 - ▼ 分配ht[1]空间T
 - ▼ T>=ht[0].used*2(2^n)
 - 扩
 - ▼ T==ht[0].used
 - 缩
 - ▼ 对ht[0]上的键rehash,存入ht[1]
 - 渐进
 - 初始化rehashindex=0
 - 每完成一个rehashindex++
 - 迁移完后，释放ht[0],ht[1]->ht[0],ht[1]创建空白哈希，为下次准备
- ▼ 跳跃表
 - ▼ zskiplist


```
typedef struct zskiplist {
    // 表头节点和表尾节点
    struct zskiplistnode *header, *tail;
    // 表中节点的数量
    unsigned long length;
    // 表中层数最大的节点的层数
    int max_level;
}
```

```
    int level;
} zskiplist;
```

▼ length

- j节点数量

▼ level

- 表中层数最大的节点的层数

▪ *headper

▪ *tail

▼ zskiplistNode

```
typedef struct zskiplistNode {
    // 后退指针
    struct zskiplistNode *backward;

    // 分值
    double score;

    // 成员对象
    robj *obj;

    // 层
    struct zskiplistLevel {
        // 前进指针
        struct zskiplistNode *forward;

        // 跨度
        unsigned int span;
    } level[];

} zskiplistNode;
```

▼ *backward

- 节点中用BW字样标记节点的后退指针

- 表尾向表头遍历使用

- 指向当前节点的前一个节点

▼ score

- 用于排序

- 生序排列

▼ *obj

- 对象
- ▼ level[]
 - 层
 - 跳跃的实现
 - 随机1~32直接的值作为数组大小
- ▼ struct
 - ▼ *forward
 - 指向表尾方向的下一个指针
 - ▼ span
 - 层的跨度
 - ▼ 两个节点之间的距离
 - 越大则越远
 - 指向NULL的forward时，为0

▼ 内存映射

▼ 整数集合intset

```
typedef struct intset {  
    // 编码方式  
    uint32_t encoding;  
  
    // 集合包含的元素数量  
    int32_t length;  
  
    // 保存元素的数组  
    int32_t contents[];  
} intset;
```

▼ 保持整数值的集合抽象数据结构

- int16_t
- int32_t
- int64_t
- 不重复
- ▼ 属性
 - ▼ encoding
 - 类型编码
 - ▼ length

- 元素个数
- ▼ contents[]
 - 声明为int8_t
 - 实际存储类型由encoding决定
 - 生序存储
- ▼ 升级
 - 新增元素类型大于原来的，encoding进行变更，contents存储位图进行变动
 - ▼ 操作
 - 根据新元素类型，进行数组空间扩容，并为新元素分配空间
 - 老数据类型变换，调整存储位置，顺序不变
 - 新元素入列
 - 不支持降级
- ▼ 压缩列表
 - 节约内存，特殊编码的连续内存块组成的顺序型数据结构
 - 一个压缩列表保护任意个节点
 - 节点可以是字节数组或者整数值
- ▼ 结构
 - ▼ zlbytes
 - uint32_t
 - 4字节
 - 列表占用字节数
 - ▼ 作用
 - 内存重新分配
 - 计算zlen位置使用
 - ▼ zltail
 - uint32_t
 - 4字节
 - 记录尾节点到起始地址多少字节
 - ▼ zllen
 - uint16_t 2字节
 - 记录节点数

- ▼ 记录节点数
 - <65535 是准确的
 - ==65535时，需要遍历才确定
- ▼ entryX
 - 节点数
 - 长度自己保存
- ▼ zlend
 - uint8_t 1字节
 - 0xFF 标记

▼ 对象

- 基于redis抽象数据结构构建系统
- ▼ encoding
 - 编码属性
 - 记录底层数据结构
- ▼ 编码类型
 - ▼ REDIS_ENCODING_INT
 - long 类型的整数
 - ▼ REDIS_ENCODING_EMBSTR
 - embstr 编码的简单动态字符串
 - ▼ REDIS_ENCODING_RAW
 - 简单动态字符串
 - REDIS_ENCODING_HT 字典
 - REDIS_ENCODING_LINKEDLIST 双端链表
 - REDIS_ENCODING_ZIPLIST 压缩列表
 - REDIS_ENCODING_INTSET 整数集合
 - REDIS_ENCODING_SKIPLIST 跳跃表和字典

▼ type类型

- ▼ 字符串对象
 - REDIS_STRING
- ▼ 列表对象

- REDIS_LIST
- ▼ 哈希对象
 - REDIS_HASH
- ▼ 集合对象
 - REDIS_SET
- ▼ 有序集合对象
 - REDIS_ZSET
- ▼ *ptr
 - 指向底层数据实现结构
 - 类型检测&命令多态
- ▼ 实现垃圾回收
 - ▼ 引用计数 refcount属性
 - 初始值1
 - 引用加一
 - 不引用减一
 - 0释放
 - ▼ 对象共享机制
 - 多个数据库键共享同一对象
 - ▼ 针对值非字符串类型
 - 内容检测成本低
 - 相同Key 值相同， 使用同一个对象
 - ▼ 过期淘汰机制 | 对象空转时长
 - ▼ Iru 属性
 - 访问时间
 - 访问频次
- ▼ 线程模型
 - 基于Reactor模式开发了网络事件处理器
 - ▼ 文件事件处理器
 - ▼ 多个套接字
 - 应答accept
 - 读取

- 读取 read
- 写入 write
- 关闭 close
- IO多路复用程序
- ▼ 文件事件分派器
 - ▼ 队列
 - 单线程消费，故也称为单线程模型

▼ 事件处理器

- 命令请求处理器
- 命令回复处理器
- 链接应答处理器

▼ 事务

```
... MULTI
OK
> INCR Foo
QUEUED
> INCR bar
QUEUED
> EXEC
```

▼ ACID

▼ 原子性

- 命令原子性
- 事务不支持，不支持回滚

▼ 一致性

▼ 入队错误

- 标记事务状态为REDIS_DIRTY_EXEC

▼ 执行错误

- 错误结果返回在事务返回信息

▼ redis进程被终结

▼ 内存模式

- 重启还是空白

▼ RDB模式

- 事物完成后才能被保存到RDB

- ▼ AOF模式
 - 事务语句未写入到AOF,或AOF未被SYNC刷新到硬盘时, 只能恢复上一次成功的数据
 - ▼ 事务「部分语句」被写入AOF文件, 并成功保存
 - 重启检测不完整, 需要调用redis-check-aof工具修复, 才能恢复之前数据
- ▼ 隔离性
 - 单线程消费操作保证
- ▼ 持久性
 - 内存模式下, 重启就丢失
 - RDB模式下, 未保持到RDB就丢失
 - AOF模式下, 刷新到硬盘期间异常, 则丢失
- ▼ 指令
 - ▼ MULTI
 - ▼ 开启事务
 - 开启后, 后续进来的命令, 不会立即执行, 进入队列, 知道EXEC后才被执行
 - 总是返回ok
 - ▼ EXEC
 - 执行事务块内的命令
 - 返回事务块所有命令的返回值, 按命令执行顺序给出
 - 被中断时, 返回nil
 - 拒绝事物状态为REDIS_DIRTY_EXEC的事物
 - ▼ DISCARD
 - 客户端清空事物队列, 并放弃执行, 客户端退出事务状态
 - ▼ WATCH
 - 乐观锁
 - 提供check-and-set (CAS) 行为
 - ▼ 可以监控一个或多个键
 - 一旦有键被修改, 之后的事务都不会执行
 - 监控直到EXEC命令
 - ▼ 监视带过期时间的键

- 即使过期了，事务也能正常执行
- 实现ZPOP操作

```
以下代码实现了原创的 ZPOP 命令。它可以原子地弹出有序集合中分值 (score) 最小的元素。  
WATCH zset  
element = ZRANGE zset 0 0  
MULTI  
    ZREM zset element  
EXEC
```

- ▼ 实现
 - ▼ redis.h/redisDb.watched_keys 字典
 - key是监听的键
 - val 是客户端链表
- ▼ UNWATCH
 - 取消对所有KEY的监控
- ▼ 作用
 - ▼ 一次性
 - 一次性执行多个命令
 - ▼ 顺序性
 - 一个事务中所有命令都会被序列化，按照顺序串行化执行
 - ▼ 排他性
 - 执行过程中，其他事务被阻塞
- ▼ 三个阶段
 - 开始MULTI
 - ▼ 命令入队
 - 2.6.5以前事务中入队失败命令会被忽略
 - 事物执行EXEC
- ▼ 不支持回滚
 - 事务失败，仍然继续执行余下指令
- ▼ 命令失败
 - 所有命令都不会执行
- ▼ 失败场景
 - ▼ 提交前，命名语法错误，

- 各广播返回错误
 - 清空队列命令
 - 取消事物
- ▼ 提交后，有命令错误
- 不取消，继续执行正确的
 - 乐观锁失败，监听Key改变，事物放弃所有指令
- ▼ 其他实现方式
- Lua
- ▼ 中间标记变量
- 通过标记的变量识别事务是否执行完成
 - 读时先标记该变量判断是否事务执行完成
 - 需要额外写代码
- ▼ 持久化
- ▼ RDB
- 默认
 - Redis DataBase
- ▼ 操作
- 按一定时间将内存数据以快照的形式保存到硬盘中
 - 对应产生的数据文件为dump.rdb
 - 可配置文件中的save参数来定义快照周期
- ▼ 优点
- 只有一个dump.rdb文件，方便持久化
 - 容灾性好，一个文件可以保存到安全的磁盘
- ▼ 性能最大化
- fork子进程来完成写操作，主进程继续处理命令，不会进行任何IO，来保证高性能
 - 相对于数据集中式，比AOF的启动效率更高
- ▼ 缺点
- ▼ 安全性能低
- 周期快照模式，若期间发送故障，容易数据丢失
- 子主题 2

▼ AOF

▼ Append-only file

- 是指所有的命令行记录以 redis 命令请求协议的格式完全持久化存储)保存为 aof 文件
- 重启Redis会重新将持久化的日志中文件恢复数据。

▼ 恢复优先级别高于RDB

- 同时设置两种模式时，优先AOF模式恢复

▼ 保存模式

▼ 不保存

- 每次调用 flushAppendOnlyFile 函数， WRITE 都会被执行，但 SAVE 会被略过。
- 阻塞

▼ save 被执行情况

- redis 被关闭
- AOF被关闭
- 系统写缓存

▼ 每秒保存一次

- 后台子线程调用
- 不阻塞主线程
- 并非每秒一次，与redis状态有关
- ▼ flushAppendOnlyFile 函数被调用出现情况

▼ 子线程正在执行save

- ▼ 超过2秒
 - 直接返回
- ▼ 未超过2秒
 - 只执行write

▼ 子线程没有执行save

- ▼ 上次成功执行距今不超过1秒
 - 执行write
- ▼ 上次成功执行已经超过一秒
 - 执行write和save

- ▼ 每次执行一个命令保存一次
 - 主进程操作
 - 会阻塞
- ▼ 数据还原
 - ▼ AOF文件存储有协议
 - 重新执行一遍即可
 - ▼ 流程
 - 创建一个不带网络链接的伪客户端
 - 读取AOF文本，根据协议还原指令
 - 客户端执行还原的指令
- ▼ 重写
 - 解决命令语句冗余问题
 - 子进程操作
- ▼ 重写缓存
 - 记录重写过程中出现的命令
- ▼ 重写触发条件
 - ▼ serverCron函数被执行时，检测
 - 没有 BGSAVE 命令在进行
 - 没有 BGREWRITEAOF 在进行。
 - 当前 AOF 文件大小大于 server.aof_rewrite_min_size （默认值为 1 MB）
 - 当前 AOF 文件大小和最后一次 AOF 重写后的大小之间的比率大于等于指定的增长百分比。
- ▼ 优点
 - ▼ 数据安全
 - ▼ 可配置appendsync属性来控制记录aof操作
 - always:每次命令都记录
 - ▼ 数据一致性
 - 即使遇到宕机，也可以用redis-check-aof工具解决数据一致性问题
 - ▼ 写模式：合并操作

- 文件过大时对重复命令，进行合并
 - 删除其中某些命令，例如误操作flushall
- ▼ 缺点
- 一般文件比RDB大，恢复慢
 - 数据集大时，启动效率低
- ▼ 抢择持久化模式
- ▼ 数据安全性重要时,能承受数分钟内的丢失
- RDB
 - 可以都启动两种
- ▼ 扩容
- ▼ 缓存场景
- 使用一致性哈希实现动态扩缩所容
- ▼ 持久化存储场景
- 一般使用固定key-nodes映射关系，节点固定
 - 若需要运行时扩缩容，用redis集群
- ▼ 过期键策略
- ▼ 使用内存设置
- 配置文件maxmemory <bytes>
 - 命令config set maxmemory XGB
- ▼ 大小
- 32位 最大3GB
 - 64位不限制
- ▼ 过期键
- ▼ 秒
- expire key ttl
 - expireat key timestamp
- ▼ 毫秒
- pexpire key ttl
 - ▼ pexpireat key timestamp
 - 最底层操作

▼ 过期策略

▼ 惰性删除

- 访问时才判断是否过期

▼ 定期扫描

- 定时

- 清理过期的KEY

▼ 淘汰策略

针对没有设置过期键，内存满时的处理

▼ volatile-lru

- 针对设置过期键

▼ allkeys-lru

- 所有键

▼ volatile-lfu

- 针对设置过期键

▼ allkeys-lfu

- 所有键

▼ volatile-random

- 针对设置过期键

- 随机删除

▼ allkeys-random

- 针对设置过期键

- 随机删除

▼ volatile-ttl

- 针对设置过期键，根据键值ttl属性

▼ noevasion

- 默认不处理

▼ 淘汰策略设置

- 配置文件 maxmemory-policy配置

- 指令 config set maxmemory-policy <策略>

▼ 算法

▼ redisObject.lru

- unsigned lru:LRU_BITS
 - 24位
 - 记录对象最后一次被访问的时间
- ▼ 改进的LRU
- ▼ 针对使用时间
 - ▼ 维护lru_clock
 - 全局函数serverCron 每100ms更新
 - 记录当前unix时间戳
 - 最大194天
 - 避免系统调用
 - ▼ 通过抽样删除
 - 配置 maxmemory_samples 设置样本大小
 - key 越大， 删除几率越高
- ▼ 如何删除
- ▼ lru_clock - redisObject.lru
 - ▼ 大于0
 - 得到空闲时间
 - ▼ 小于0时
 - lruclock_max (即 194 天) - lru + lruclock得到空闲时间
- ▼ LFU
- ▼ 针对使用频率
 - ▼ 存储在redisObject.lru
 - ▼ 高16位
 - 记录访问时间 ldt
 - 单位分钟
 - ▼ 低8位
 - 记录访问频率 logc
 - counter
 - 最大255
 - ▼ lfu_log_factor

- 对数因子

- 默认10

- 可配置

- ▼ 访问频次递增算法

给定一个旧的访问频次，当键访问递增counter

- 取(0~1)随机数 R

- ▼ counter - 初始值5

- <=0

- >0 取实数

- ▼ $1 / (\text{baseval} * \text{lru_log_factocr} + 1)$

- 概率P

- ▼ $R < P$

- counter++

- ▼ 访问频次递减算法

- 针对一段时间不访问不访问的操作 减少counter

- ▼ lru-decay-time

- 控制减少counter速度

- 默认为1

- ▼ 判断流出

- 当前时间截高取16位，转分钟 标记为now

- 取对象lru高16位，标记为ldt

- ▼ $ldt > now$

- 默认过了一周

- 取 $65535 - ldt + now$

- ▼ $lru \leq now$

- 取 $now - ldt$

- ▼ 计算num_periods

- $\text{idle_time} / \text{lru_decay_time}$

- ▼ 减少counter

- $counter - num_periods$

- ▼ 集群方案

- ▼ 哨兵模式
 - ▼ sentinel 哨兵组件
 - ▼ 功能
 - ▼ 监控集群
 - 监控redis master slave进程是否正常工作
 - ▼ 消息通知
 - 故障消息告警
 - ▼ 故障转移
 - 主节点宕机，自动转移到slave
 - ▼ 配置中心
 - 故障转移后，通知客户端新的master地址
 - ▼ 本身是分布式，作为集群工作
 - 故障转移，判断是否master宕机需要所有哨兵同意
 - 至少需要3个实例
 - 哨兵+redis主从，不保证数据零丢失
- ▼ 官方Redis Cluster
 - 服务端路由查询
 - 服务端Sharding技术
 - 3.0开始
- ▼ 采用slot(槽)概念
 - 一共可以有16384个槽（节点）
 - 请求发送到任意节点，节点会转发到正确的节点上执行
- ▼ 设计方案
 - 通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值)区间的数据，默认分配了16384个槽位
 - 每份数据分片会存储在「多个互为主从」的多节点上
 - 数据写入先写主节点，再同步到从节点(支持配置为阻塞同步)
 - 同一分片多个节点间的数据不保持一致性
 - 读取数据时，当客户端操作的key没有分配在该节点上时，redis会返回转向指令，指向正确的节点
 - 扩容时时需要把旧节点的数据迁移一部分到新节点

- ▼ 开放端口
 - 比如一个是6379
 - ▼ 另一个加1W,16379
 - 用来节点通信cluster bus
- ▼ cluster bus通信
 - ▼ gossip协议
 - 二进制
 - 高效
 - 占用带宽少
 - 处理时间短
 - ▼ 功能
 - 故障检测
 - 配置更新
 - 故障转移授权
- ▼ key如何寻址
 - 流程
 - ▼ 算法
 - ▼ hash算法
 - 大量缓存重建
 - 一致性 hash 算法（自动缓存迁移）+ 虚拟节点（自动负载均衡）
 - redis cluster 的 hash slot 算法
- ▼ 优点
 - 去中心化
 - 支持动态扩容
 - 业务透明
 - 具备监控和故障转移能力
 - 客户端不需要连接集群所有节点，可直连，免去代理消耗
- ▼ 缺点
 - 运维复杂，数据迁移人工干预
 - 只能使用0号数据库

- 支持批量操作 (pipeline)
 - 分布式逻辑和存储模块耦合等
- ▼ 基于客户端分区
- ▼ 思想
 - 采用哈希算法将redis数据的key进行散列
 - 通过hash函数，特定的key会映射到特定的Redis节点上
 - ▼ 特点
 - 各个实例独立，无关联，容易线性扩展
 - 由于sharding处理放到客户端，规模进一步扩大时给运维带来挑战。
 - 客户端sharding不支持动态增删节点。redis实例群数量变动，需重新调整客户端
 - 连接不能共享
- ▼ 基于服务器分片
- ▼ 特点
 - 透明接入，切换成本低，业务程序不关心后端实例
 - Proxy的逻辑和存储的逻辑隔开
 - 代理有消耗
 - ▼ 业界开源方案
 - Twitter 开源的Twemproxy
 - 豌豆荚开源的Codis
- 主从
- ▼ 分区
- 解决单机内存有限问题
 - 增加网络带宽
- ▼ 方案
- 客户端分区
 - 代理端分区
 - 官方Redis Cluster
- ▼ 应用场景
- 计数器
 - 缓存

- 会话缓存
- 全页缓存
- 查找表
- 消息队列（发布/订阅）
- ▼ 延时队列
 - 使用sortedset，使用时间戳做score，消息内容作为key，调用zadd来生产消息，消费者使用zrangebytime获取n秒之前的数据做轮询处理。
- ▼ 分布式锁实现
 - setnx
 - RedLock
- ▼ 主从架构
 - ▼ replication复制
 - 异步复制数据到slave
 - 2.8开始 slave node会周期性确认自己复制的数据量
 - ▼ slave 节点
 - 横向扩容
 - 读写分离
 - 一主可配置多从
 - slave 节点可以连接其他slave node
 - slave节点做复制的时候，不会阻塞正常工作，包括主节点的，但是复制完成时，删除旧数据，加载新的时候，它会停止对外提供服务
 - ▼ 主从复制核心原理
 - slave 发送PSYNC给master
 - ▼ 全量复制
 - 第一次链接时
 - master 会启动一个后台线程，开始生成一份 RDB 快照文件，
 - 期间收到命令写入缓存中，RDB生成后，发给slave
 - slave 会先写入本地磁盘，然后再从本地磁盘加载到内存中，
 - master 会将内存中缓存的写命令发送到 slave，slave 也会同步这些数据
 - ▼ 网络故障处理
 - 自动重连，连接之后 master node 仅会复制给 slave 部分缺少的数据

- 开启时，建议开启主节点持久化
- ▼ 最好是链模式
 - master->slave->slave
 - 避免压力过大
- ▼ 分布式锁
 - ▼ Redlock
 - 互斥访问，即永远只有一个 client 能拿到锁
 - ▼ 避免死锁
 - 最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分
 - ▼ 容错性
 - 只要大部分 Redis 节点存活就可以正常提供服务
 - ▼ setnx
 - SET if Not eXists
 - 成功1，失败0
 - 过期时间 避免一直占用
- ▼ 问题
 - ▼ 缓存异常
 - ▼ 雪崩
 - 缓存同一时间大面积的失效，请求都在数据库上，导致的崩掉
 - ▼ 解决方案
 - 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
 - 一般并发量不是特别多的时候，使用最多的解决方案是加锁排队
 - 给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。
 - ▼ 穿透
 - 缓存和数据库中都没有的数据，导致所有的请求都落到数据库上而崩掉
 - ▼ 解决方案
 - 接口层增加校验，如用户鉴权校验，id做基础校验， $id \leq 0$ 的直接拦截
 - 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null。缓存有效时间可以设置短点，如20秒（设置太长会导致

对与 /key-null，缓存失效时可以直报，如果失败（直入云导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击

▼ 布隆过滤器

- 所有可能存在的数据哈希到一个足够大的bitmap上，0, 1表示是否存在，来判断避免无效key到数据库上
- K($k > 1$)多个哈希函数，减少冲突概率
- 完成去重

▼ 击穿

- 缓存没有，数据库有，并发时同时读取「相同数据」，压力到数据库上

▼ 解决方案

- 设置热点数据永远不过期
- 加互斥锁，互斥锁

▼ 预热

▼ 解决方案

- 数据量不大时，项目启动时刷上去
- 定时刷新缓存

▼ 降级

▪ 解决方案

▼ 热点key

▼ 解决方案

- 对缓存查询加锁，不存在时，加锁阻塞其他请求进入DB,

▪ 内存相关