Redis

- 介绍
  - 简介
    - Redis：Remote Dictionary Server(远程字典服务器)
    - 是完全开源免费的，用C语言编写的，遵守BSD协议，是一个高性能的(key/value)分布式内存数据库，基于内存运行
      并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之一,也被人们称为数据结构服务器
  - Redis 与其他 key - value 缓存产品有以下三个特点
    - Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用
    - Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储
    - Redis支持数据的备份，即master-slave模式的数据备份
  - 优势
    - 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。
    - 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
    - 原子 – Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。
    - 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。
  - 管道技术
    - Redis是一种基于客户端-服务端模型以及请求/响应协议的TCP服务。这意味着通常情况下一个请求会遵循以下步骤：
    - 1、客户端向服务端发送一个查询请求，并监听Socket返回，通常是以阻塞模式，等待服务端响应。
    - 2、服务端处理命令，并将结果返回给客户端。
- 数据结构
  - 五种数据结构类型
    - String（字符串）
      - 介绍
      - 实例
      - 特性
      - 应用场景
    - Hash（哈希，类似java里的Map）
      - 介绍
      - 实例
      - 特性
      - 应用场景

- List（列表）
  - 介绍
  - 实例
  - 特性
  - 应用场景
- Set（集合）
  - 介绍
  - 实例
  - 特性
  - 应用场景
- Zset(sorted set：有序集合)
  - 介绍
  - 实例
  - 特性
  - 应用场景

- HyperLogLog
  - 介绍
  - 什么是基数?
    - 不重复元素
- redis命令大全

- 发布订阅
  - Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。
  - Redis 客户端可以订阅任意数量的频道。
  - 命令
  - 案例
- 事物
  - 什么是redis事物
    - Redis 事务可以一次执行多个命令
    - 带有以下三个重要的保证：
      - 批量操作在发送 EXEC 命令前被放入队列缓存。
      - 收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行。
      - 在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中。
  - 三个阶段
    - 开始事务。
    - 命令入队。
    - 执行事务。
  - 案例
  - redis事物不具备原子性
  - 命令
    - DISCARD：取消事务，放弃执行事务块内的所有命令。

- EXEC： 执行所有事务块内的命令。
- MULTI： 标记一个事务块的开始。
- WATCH
  - Redis Watch 命令用于监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断
  - 案例
- UNWATCH： 取消 WATCH 命令对所有 key 的监视。

- 哨兵模式
  - 介绍
    - 由于所有的写操作都是先在Master上操作，然后同步更新到Slave上，所以从Master同步到Slave机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，Slave机器数量的增加也会使这个问题更加严重。
    - sentinel可以让redis实现主从复制，当一个集群中的master失效之后，sentinel可以选举出一个新的master用于自动接替master的工作，集群中的其他redis服务器自动指向新的master同步数据。一般建议sentinel采取奇数台，防止某一台sentinel无法连接到master导致误切换。
  - Sentinel工作方式（每个Sentinel实例都执行的定时任务）
    - 每个Sentinel以每秒钟一次的频率向它所知的Master，Slave以及其他 Sentinel 实例发送一个PING命令。
    - 如果一个实例（instance）距离最后一次有效回复PING命令的时间超过 own-after-milliseconds 选项所指定的值，则这个实例会被Sentinel标记为主观下线。
    - 如果一个Master被标记为主观下线，则正在监视这个Master的所有 Sentinel 要以每秒一次的频率确认Master的确进入了主观下线状态。
    - 当有足够数量的Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认Master的确进入了主观下线状态，则Master会被标记为客观下线。
    - 在一般情况下，每个Sentinel 会以每10秒一次的频率向它已知的所有Master，Slave发送 INFO 命令。
    - 当Master被Sentinel标记为客观下线时，Sentinel 向下线的 Master 的所有Slave发送 INFO命令的频率会从10秒一次改为每秒一次。
    - 若没有足够数量的Sentinel同意Master已经下线，Master的客观下线状态就会被移除。 若 Master重新向Sentinel 的PING命令返回有效回复，Master的主观下线状态就会被移除。
  - 三个定时任务
    - 1、每10秒每个sentinel会对master和slave执行info命令，这个任务达到两个目的：
      - 发现slave节点
      - 确认主从关系
    - 2、每2秒每个sentinel通过master节点的channel交换信息(pub/sub)
      - 通过_sentinel_:hello频道交互
      - 交互对节点的“看法”和自身信息
    - 3、每1秒每个sentinel对其他sentinel和redis节点执行ping操作（相互监控），这个其实是一个心跳检测，是失败判定的依据。
  - 主观下线

- 客观下线
- 怎么玩（使用步骤）
  - 新建sentinel.conf文件，名字绝不能错
  - 配置哨兵，填写内容：sentinel monitor <masterName> <ip> <port> <quorum>
    - 参数1：masterName这个是对某个master+slave组合的一个区分标识（一套sentinel是可以监听多套master+slave这样的组合的）。
    - 参数2：ip 是master节点的 ip 。
    - 参数3：port 是master节点的端口号。
    - 参数4：quorum这个参数是进行客观下线的一个依据，意思是至少有 quorum 个sentinel主观的认为这个master有故障，才会对这个master进行下线以及故障转移。因为有的时候，某个sentinel节点可能因为自身网络原因，导致无法连接master，而此时master并没有出现故障，所以这就需要多个sentinel都一致认为该master有问题，才可以进行下一步操作，这就保证了公平性和高可用。
  - 启动哨兵
    - Redis-sentinel /myredis/sentinel.conf
    - 上述目录依照各自的实际情况配置，可能目录不同
  - 问题：如果之前的master重启回来，会不会双master冲突?
    - 之前的master会变成slave
- Sentinel主要负责三个方面的任务
  - 监控（Monitoring）
  - 提醒（Notification）
  - 自动故障迁移（Automatic failover）
- sentinel.conf配置文件
  - sentinel monitor mymaster 192.168.10.202 6379 2
  - sentinel down-after-milliseconds mymaster 30000
  - sentinel parallel-syncs mymaster 2
  - sentinel can-failover mymaster yes
  - sentinel auth-pass mymaster 20180408
  - sentinel failover-timeout mymaster 180000
  - sentinel config-epoch mymaster 0
  - sentinel notification-script mymaster /var/redis/notify.sh
  - sentinel leader-epoch mymaster 0
- 故障转移过程
  - 1、多个sentinel发现并确认master有问题。
  - 2、选举出一个sentinel作为领导。
    - 原因：只有一个sentinel节点完成故障转移
    - 选举：通过sentinel is-master-down-by-addr命令都希望成为领导者
      - 每个做主观下线的Sentinel节点向其他Sentinel节点发送命令，要求将它设置为领导者。
      - 收到命令的Sentinel节点如果没有同意通过其他Sentinel节点发送的命令，那么将同意该请求，否则拒绝

- 如果该Sentinel节点发现自己的票数已经超过Sentinel集合半数且超过quorum，那么它将成为领导者。
- 如果此过程有多个Sentinel节点成为了领导者，那么将等待一段时间重新进行选举。

- 3、选出一个slave作为master。
  - 故障转移（sentinel领导者节点完成）
    - 1.从slave节点中选出一个"合适的"节点作为新的master节点
    - 2.对上面的slave节点执行slaveof no one命令让其成为master节点。
    - 3.向剩余的slave节点发送命令，让它们成为新master节点的slave节点，复制规则和parallel-syncs参数有关。
    - 4.更新对原来master节点配置为slave，并保持着对其"关注"，当其恢复后命令它去复制新的master节点。
- 4、通知其余slave成为新的master的slave。
- 5、通知客户端主从变化
- 6、等待老的master复活成为新master的slave。

- 缓存设计与优化
  - 缓存收益与成本
    - 收益
    - 成本
  - 缓存更新策略
    - 更新策略
      - LRU/LFU/FIFO算法剔除
      - 超时剔除
      - 主动更新
    - 两点建议
      - 低一致性
      - 高一致性
  - 缓存穿透
    - 描述
    - 解决方案
      - 缓存空对象
      - 布隆过滤器
  - 缓存雪崩
    - 描述
    - 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。
      - 如果缓存数据库是分布式部署，将热点数据均匀分布在不同搞得缓存数据库中。
      - 设置热点数据永远不过期。
  - 缓存击穿
    - 描述
    - 解决方案

- 使用互斥锁(mutex key)
- "提前"使用互斥锁(mutex key)
  - "永远不过期"
    - 从redis上看，确实没有设置过期时间，这就保证了，不会出现热点key过期问题，也就是“物理”不过期。
    - 从功能上看，如果不过期，那不就成静态的了吗？所以我们把过期时间存在key对应的value里，如果发现要过期了，通过一个后台的异步线程进行缓存的构建，也就是“逻辑”过期
  - 缓存屏障
    - 流程图
    - 代码
    - 该方法类似于方法一：使用countDownLatch和atomicInteger.compareAndSet()方法实现轻量级锁

- Redis实现分布式锁
- nosql数据库
  - CAP理论
    - 传统关系型数据库ACID
    - nosql数据库CAP
      - C:Consistency（强一致性）
      - A:Availability（可用性）
      - P:Partition tolerance（分区容错性）
    - CAP的3进2
    - 在分布式系统下nosql数据库必须满足P
  - BASE理论
    - 是什么
  - 分布式+集群
    - 介绍
- redis.conf
  - Units单位
  - INCLUDES包含
  - GENERAL通用
  - SNAPSHOTTING快照
    - Save
      - save 秒钟 写操作次数
      - 禁用
    - Stop-writes-on-bgsave-error
    - rdbcompression
    - rdbchecksum
    - dbfilename
    - dir

- REPLICATION复制
- SECURITY安全
  - requirepass foobared
- LIMITS限制
- APPEND ONLY MODE追加
  - appendonly
  - appendfilename
  - Appendfsync
    - Always：同步持久化 每次发生数据变更会被立即记录到磁盘 性能较差但数据完整性比较好
    - Everysec：出厂默认推荐，异步操作，每秒记录 如果一秒内宕机，有数据丢失
    - No
  - No-appendfsync-on-rewrite：重写时是否可以运用Appendfsync，用默认no即可，保证数据安全性。
  - Auto-aof-rewrite-min-size：设置重写的基准值
  - Auto-aof-rewrite-percentage：设置重写的基准值
- 常见配置redis.conf介绍

- 持久化
  - RDB（Redis DataBase ）
    - 什么是RDB
    - 触发条件
      - 手动触发
        - save命令
        - bgsave命令
      - 自动触发
        - 配置文件配置（save m n ）
          - 默认配置
          - 实现原理
        - 主从复制场景
        - 执行shutdown命令之后执行
    - Fork
      - 说明
      - 为什么Redis进行RDB持久化数据时，新起一个进程而不是在原进程中起一个线程
    - 执行流程
      - 流程图
      - 步骤
        - 1、Redis父进程首先判断：当前是否在执行save，或bgsave的子进程，如果在执行则bgsave命令直接返回。
        - 2、父进程执行fork操作创建子进程，这个过程中父进程是阻塞的，Redis不能执行来自客户端的任何命令

- 3、父进程fork后，bgsave命令返回"Background saving started"信息并不再阻塞父进程，并可以响应其他命令
- 4、子进程创建RDB文件，根据父进程内存快照生成临时快照文件，完成后对原有文件进行原子替换
- 5、子进程发送信号给父进程表示完成，父进程更新统计信息

- 数据恢复
  - 直接将dump.rdb文件移动到redis安装目录,并启动redis即可
  - 服务器启动自动执行
- 常用配置总结
  - save m n
  - stop-writes-on-bgsave-error yes
  - rdbcompression yes
  - rdbchecksum yes
  - dbfilename dump.rdb
  - dir ./
- 优缺点
  - 优点
  - 缺点
- redis默认开启RDB，设置redis-cli config set save ""，可以停止RDB

- AOF（Append Only File）
  - 什么是AOF
    - 说明
  - 默认redis关闭AOF
    - appendonly yes：开启AOF
  - 执行流程
    - 命令追加(append)
    - 文件写入(write)和文件同步(sync)
      - 说明
      - AOF缓存区的同步文件策略由参数appendfsync控制
        - always
        - everysec
        - no
    - 文件重写(rewrite)
      - 说明
      - 触发条件
        - 手动触发
        - 自动触发
          - auto-aof-rewrite-min-size：执行AOF重写时，文件的最小体积，默认值为64MB。

- auto-aof-rewrite-percentage：执行AOF重写时，当前AOF大小(即aof_current_size)和上一次重写时AOF大小(aof_base_size)的比值。
- 流程
  - 流程图
  - 注意
  - 步骤
- 数据加载
  - 说明
  - 步骤
    - 文件校验
    - 伪客户端
- 常用配置及默认值
  - appendonly no
  - appendfilename “appendonly.aof”
  - dir ./
  - appendfsync everysec
  - no-appendfsync-on-rewrite no
  - auto-aof-rewrite-percentage 100
  - auto-aof-rewrite-min-size 64mb
  - aof-load-truncated yes
- 优缺点
  - 优点
    - 1、使用AOF Redis会更具有可持久性(durable)：你可以有很多不同的fsync策略：没有fsync，每秒fsync，每次请求时fsync。使用默认的每秒fsync策略，写性能也仍然很不错(fsync是由后台线程完成的，主线程继续努力地执行写请求)，即便你也就仅仅只损失一秒钟的写数据。
    - 2、AOF日志是一个追加文件，所以不需要定位，在断电时也没有损坏问题。即使由于某种原因文件末尾是一个写到一半的命令(磁盘满或者其他原因),redis-check-aof工具也可以很轻易的修复。
    - 3、当AOF文件变得很大时，Redis会自动在后台进行重写。重写是绝对安全的，因为Redis继续往旧的文件中追加，使用创建当前数据集所需的最小操作集合来创建一个全新的文件，一旦第二个文件创建完毕，Redis就会切换这两个文件，并开始往新文件追加。
    - 4、AOF文件里面包含一个接一个的操作，以易于理解和解析的格式存储。你也可以轻易的导出一个AOF文件。例如，即使你不小心错误地使用FLUSHALL命令清空一切，如果此时并没有执行重写，你仍然可以保存你的数据集，你只要停止服务器，删除最后一条命令，然后重启Redis就可以。
  - 缺点
    - 1、对同样的数据集，AOF文件通常要大于等价的RDB文件。
    - 2、AOF可能比RDB慢，这取决于准确的fsync策略。通常fsync设置为每秒一次的话性能仍然很高，如果关闭fsync，即使在很高的负载下也和RDB一样的快。不过，即使在很大

的写负载情况下，RDB还是能提供能好的最大延迟保证。

- 3、在过去，我们经历了一些针对特殊命令(例如，像BRPOPLPUSH这样的阻塞命令)的罕见bug，导致在数据加载时无法恢复到保存时的样子。这些bug很罕见，我们也在测试套件中进行了测试，自动随机创造复杂的数据集，然后加载它们以检查一切是否正常，但是，这类bug几乎不可能出现在RDB持久化中。为了说得更清楚一点：Redis AOF是通过递增地更新一个已经存在的状态，像MySQL或者MongoDB一样，而RDB快照是一次又一次地从头开始创造一切，概念上更健壮。但是，1)要注意Redis每次重写AOF时都是以当前数据集中的真实数据从头开始，相对于一直追加的AOF文件(或者一次重写读取老的AOF文件而不是读内存中的数据)对bug的免疫力更强。2)我们还没有收到一份用户在真实世界中检测到崩溃的报告。

- 如何选择持久化方案
  - 前提
  - 几种方案
    - 不需要持久化
    - 二选一
    - 主从备份环境下
      - master：完全关闭持久化（包括RDB和AOF），这样可以让master的性能达到最好
      - slave：关闭RDB，开启AOF（如果对数据安全要求不高，开启RDB关闭AOF也可以），并定时对持久化文件进行备份（如备份到其他文件夹，并标记好备份的时间）；然后关闭AOF的自动重写，然后添加定时任务，在每天Redis闲时（如凌晨12点）调用bgrewriteaof。
      - 总结
    - 异地灾备
- fork阻塞：CPU的阻塞
  - fork子进程过程详解（写时复制技术）
  - redis有两处需要fork子进程
    - RDB持久化的bgsave
    - AOF重写的bgrewriteaof
  - Redis内存过大对fork子进程影响
  - 改善fork的几点建议
    - 1、优先使用物理机或者高效支持fork操作的虚拟化技术
    - 2、控制Redis实例最大可用内存：maxmemory
    - 3、合理配置linux内存分配策略：vm.overcommit_memory=1
    - 4、降低fork触发频率，例如放宽aof自动重写时机
- AOF追加阻塞：硬盘的阻塞
  - 原因
  - Redis的处理策略
  - AOF追加阻塞问题定位的方法及解决方案
    - 1、监控info Persistence中的aof_delayed_fsync：当AOF追加阻塞发生时（即主线程等待fsync而阻塞），该指标累加。

- 2、AOF阻塞时的Redis日志：
  Asynchronous AOF fsync is taking too long (disk is busy?). Writing the AOF buffer without waiting for fsync to complete, this may slow down Redis.
- 3、如果AOF追加阻塞频繁发生，说明系统的硬盘负载太大；可以考虑更换IO速度更快的硬盘，或者通过IO监控分析工具对系统的IO负载进行分析，如iostat（系统级io）、iotop（io版的top）、pidstat等。

- 主从复制
  - 实现主从复制的两种方式
    - slaveof命令
      - 建立主从命令：slaveof ip port
      - 取消主从命令：slaveof no one
    - redis.conf配置文件配置
      - 格式：slaveof ip port
      - 从节点只读：slave-read-only yes #配置只读
  - 复制过程
    - 1、从节点执行 slaveof 命令。
    - 2、从节点只是保存了 slaveof 命令中主节点的信息，并没有立即发起复制。
    - 3、从节点内部的定时任务发现有主节点的信息，开始使用 socket 连接主节点。
    - 4、连接建立成功后，发送 ping 命令，希望得到 pong 命令响应，否则会进行重连。
    - 5、如果主节点设置了权限，那么就需要进行权限验证，如果验证失败，复制终止。
    - 6、权限验证通过后，进行数据同步，这是耗时最长的操作，主节点将把所有的数据全部发送给从节点。
    - 7、当主节点把当前的数据同步给从节点后，便完成了复制的建立流程。接下来，主节点就会持续的把写命令发送给从节点，保证主从数据一致性。
  - 数据同步
    - 数据同步命令
      - sync：redis 2.8之前的同步命令
      - psync：redis 2.8之后的同步命令
        - 格式：psync{runId}{offset}
        - runId：从节点所复制主节点的运行 id
        - offset：当前从节点已复制的数据偏移量
        - 主节点复制积压缓冲区：
        - 执行流程
    - 全量复制
      - 流程图
      - 步骤
        - 1、slave发送psync，由于是第一次复制，不知道master的runid，自然也不知道offset，所以发送psync？ -1
        - 2、master收到请求，发送master的runid和offset给从节点。
        - 3、从节点slave保存master的信息

- 4、主节点bgsave保存rdb文件
- 5、RDB文件生成完毕之后，主节点会将RDB发送给slave。
- 并且在④和⑤的这个过程中产生的数据，会写到复制缓冲区repl_back_buffer之中去。
- 6、从节点收到 RDB 文件并加载到内存中
- 7、主节点在从节点接受数据的期间，将新数据保存到“复制客户端缓冲区”，当从节点加载 RDB 完毕，再发送过去。 （如果从节点花费时间过长，将导致缓冲区溢出，最后全量同步失败）
- 8、从节点清空数据后加载 RDB 文件，如果 RDB 文件很大，这一步操作仍然耗时，如果此时客户端访问，将导致数据不一致，可以使用配置slave-server-stale-data 关闭.
- 9、从节点成功加载完 RBD 后，如果开启了 AOF，会立刻做 bgrewriteaof。

- 复制缓冲区（repl_back_buffer）
- 注意
  - 1、如过 RDB 文件大于 6GB，并且是千兆网卡，Redis 的默认超时机制（60 秒），会导致全量复制失败。可以通过调大 repl-timeout 参数来解决此问题。
  - 2、Redis 虽然支持无盘复制，即直接通过网络发送给从节点，但功能不是很完善，生产环境慎用。

- 部分复制
  - 流程图
  - 步骤
    - 1、当从节点出现网络中断，超过了 repl-timeout 时间，主节点就会中断复制连接。
    - 2、主节点会将请求的数据写入到“复制积压缓冲区”，默认 1MB。
    - 3、当从节点恢复，重新连接上主节点，从节点会将 offset 和主节点 id 发送到主节点。
    - 4、主节点校验后，如果偏移量的数后的数据在缓冲区中，就发送 cuntinue 响应 —— 表示可以进行部分复制。
    - 5、主节点将缓冲区的数据发送到从节点，保证主从复制进行正常状态。
  - 部分重同步功能主要由以下三个部分构成
    - 1、复制偏移量
    - 2、复制积压缓冲区
    - 3、服务器运行ID

- 主从节点心跳链接
  - 说明：主从节点在建立复制后，他们之间维护着长连接并彼此发送心跳命令。
  - 1、主从都有心跳检测机制，各自模拟成对方的客户端进行通信，通过 client list 命令查看复制相关客户端信息， 主节点的连接状态为 flags = M，从节点的连接状态是 flags = S。
  - 2、主节点默认每隔 10 秒对从节点发送 ping 命令，可修改配置 repl-ping-slave-period 控制发送频率。
  - 3、从节点在主线程每隔 1秒发送 replconf ack{offset} 命令，给主节点上报自身当前的复制偏移量。
  - 4、主节点收到 replconf 信息后，判断从节点超时时间，如果超过 repl-timeout 60 秒，则判断节点下线。
- 其他

- 1、过期key处理 slave不会过期key，只会等待master过期key。 如果master过期了一个key，或者通过LRU淘汰了一个key，那么会模拟一条del命令发送给slave。
- 2、无磁盘化复制 master在内存中直接创建rdb，然后发送给slave，不会在自己本地落地磁盘了
  repl-diskless-sync
  repl-diskless-sync-delay，等待一定时长再开始复制，因为要等更多slave重新连接过来

缺点

- 复制延时

Redis集群(Redis cluster)

为什么使用集群

- redis单机最高可以达到10万/s,如果业务需要100万/s呢?
- 单机器内存太小，无法满足需求

数据分布

- 顺序分布
  - 详情图
  - 特点
- 哈希分布
  - 详情图
  - 特点
- 一致性哈希分布
  - 详细图
  - 特点
- 虚拟槽分布
  - 详细图
  - 特点

Redis Cluster基本概念

- 结构设计
  - 结构图
  - 高性能
    - 采用了异步复制机制，向某个节点写入数据时，无需等待其它节点的写数据响应。
    - 无中心代理节点，而是将客户端直接重定向到拥有数据的节点。
    - 对于N个 Master 节点的 Cluster ，整体性能理论上相当于单个 Redis 的性能的N倍。
  - 高可用
    - 采用了主从复制的机制，Master 节点失效时 Slave 节点自动提升为 Master 节点。如果 Cluster 中有N个 Master 节点，每个 Master 拥有1个 Slave 节点，那么这个 Cluster 的失效概率为 1/(2*N-1)，可用概率为 1-1/(2*N-1)。
  - 高可扩展
    - 可支持多达1000个服务节点。随时可以向 Cluster 中添加新节点，或者删除现有节点。Cluster 中每个节点都与其它节点建立了相互连接。
- 结构特点
- 主要组件

- 重定向
- 故障转移
  - 心跳和 gossip 消息
  - 故障发现
  - 故障恢复

- Redis的过期策略和内存淘汰策略
  - 过期策略
    - 定时过期
    - 惰性过期
    - 定期过期
    - Redis中同时使用了惰性过期和定期过期两种过期策略。
  - 内存淘汰策略
    - noeviction
    - allkeys-lru
    - allkeys-random
    - volatile-lru
    - volatile-random
    - volatile-ttl