Redis

- **底层数据结构**
  - SDS简单动态字符串
    - 默认的字符串表示 —结构→
      ```
      struct sdshdr{
      int len;
      int free;
      char buf[];
      }
      ```
    - 为什么不使用原生的C字符串
      - 利用len字段，获取字符串复杂度为O(1)，效率更高
      - 利用free字段，实现空间预分配和惰性空间释放
        - 扩展字符串长度时，不需要每次都一定要重新分配内存
        - 缩短字符串长度时，并不马上回收内存，为将来有可能的增长提供了优化
      - 二进制安全，不利用空字符判断字符串结束，而是使用len，从而支持更多类型的数据格式
  - 链表
    - 双端链表 —结构→
      ```
      struct listNode{
      struct listNode *prev;
      struct listNode *next;
      void * val;
      }

      struct list{
      listNode * head;
      listNode * tail;
      unsigned long len;
      ...
      }
      ```
      - 每一个节点包括prev和next指针，无环
      - 带表头和表尾指针
      - 带长度计数器
      - void *保存不同类型的值
  - 字典
    - 哈希表 —结构→
      ```
      struct dictEntry{
      void * key;
      union {
      void * val;
      ...
      } v;
      struct dictEntry *next;
      }
      ```
      - 由哈希表数组构成，数组的每个元素为dictEntry类型，其中保存着键值对，以及解决冲突的next指针
      - 字典由两个哈希表ht构成，ht[1]将在对ht[0]进行渐近式rehash时使用
  - 跳跃表 —结构→
    ```
    struct zskiplist{
    struct zskiplistNode *header, *
    tail;
    }
    struct zskiplistNode{
       struct zskiplistLevel{
        struct zskiplistNode *forward;
        unsigned int span;
    }level[];
        struct zskiplistNode *backward;
    double score;
    }
    ```
    - 只用于实现有序集合键
    - 实现比平衡树简单
    - 支持平均O(logN)，最坏O(N)的复杂度查找
    - span用于计算排名，score保存节点的分值
  - 整数集合 —结构→
    ```
    struct intset{
      unit32_t encoding;
      unit32_t length;
       int8_t contents[];
    }
    ```
    - 一个集合只包含整数值元素，且数量不多时，采用intset作为底层实现
    - 集合中不会出现重复元素
    - 每一个元素都是contents中的项，按照值的大小从小到大有序排列
    - 元素类型由encoding真正决定，添加更长的新元素会扩展contents，并且将原有元素转换后移动到新位置，改变encoding，称之为升级。一旦升级，不会再降级
  - 压缩列表
    - 少量列表项和少量哈希键值对的情况下，使用压缩列表
    - 节约内存，连续内存块组成的顺序型数据结构
    - 包含任意多个节点，每个节点entry可以保存一个字节数组或者整数值，包括长度，encoding和content内容三个部分
- **对象**
  - 值对象类型
    - 字符串对象 REDIS_STRING
      - int
      - 使用sds
        - raw
        - embstr
    - 列表对象 REDIS_LIST
      - ziplist
      - linkedlist
    - 哈希对象 REDIS_HASH
      - ziplist
      - hashtable
    - 集合对象 REDIS_SET
      - intset
      - hashtable
    - 有序集合对象 REDIS_ZSET
      - ziplist
      - skiplist
  - 键总是字符串对象，值可以是上面任意一种 —结构→
    ```
    struct redisObject{
      unsigned type;   //类型，可以是五种对象的任意一种，可以使用type命令查看具体类型
      unsigned encoding;  //编码，真正决定底层用什么数据结构实现，提升了灵活性和效率，可以使用object encoding查看
      void *ptr;      //指向底层数据结构的指针
      int refcount;
      unsigned lru; // 最后一次被访问时间
    }
    ```
  - 类型检查，检查命令与键是否匹配 — 通过检查type实现
  - 命令多态，相同的命令对不同的数据结构都生效 — 根据encoding实现，调用指定数据结构API
  - 内存回收 — 通过引用计数，refcount字段
  - 共享对象 — 初始化服务器时创建0-9999的所有整数值作为共享对象
- **单机数据库**
  - 数据库
    - struct redisServer{ redisDb *db; ... }
    - struct redisClient{ redisDb * db; ... } —结构→
      ```
      struct redisDb{
        dict * dict;
        dict * expires;
      }
      ```
      - redisDb中的dict存储了所有键值对，键(字符串对象) -> 值(五种对象的一种)
        - flushdb是通过删除所有键值对实现的
      - expires字典包括了所有键的过期时间，键(某个键对象)->值(毫秒精度的时间戳)
    - redisServer中db保存所有数据库的数组，默认大小为16
    - redisClient中的db的指向当前redisServer的数据库，可以使用select修改
  - 键
    - 过期设置 — 三个命令expire，pexpire和expireat，最终都是通过pexpireat实现
    - 过期删除策略
      - 定时删除 — 每个键创建定时器，删除快速，但是性能差
      - 惰性删除 — 使用时检查是否过期，对CPU友好，内存不友好
      - 定期删除 — 每隔一段时间检查进行删除，定时删除和惰性删除的折中方案
      - 实际采取的方案：惰性删除+定期删除
    - 过期键处理策略
      - 持久化
        - RDB
          - 使用save和bgsave创建新的RDB时，不会保存过期键
          - 载入RDB时，主服务器会忽略过期键，但从服务器不会
        - AOF — 重写AOF不会写入过期键
      - 复制 — 主服务器删除过期键会通知从服务器删除 — 从服务器只有收到主服务器发送的del命令才会删除过期键
    - 键的变化 — 客户端通过订阅频道或者模式，服务器调用notify-KeyspaceEvent判断是否需要发送通知
  - 持久化
    - RDB
      - 将内存中的键值对保存到磁盘，生成经过压缩的二进制文件
      - save和bgsave都可以生成，但是save会阻塞服务器
      - 启动时检测到RDB的存在，会自动载入，载入时是阻塞状态
    - AOF
      - 保存服务器执行的写命令，包括命令追加，文件写入和文件同步 —AOF过大如何处理？→ AOF重写
        - 不需要分析现有的AOF文件，而是通过读取服务器当前数据库状态实现 — 使用一条命令代替原先多条命令
        - 后台重写，在子进程中执行重写，不阻塞服务器
          - 避免数据库状态不一致，新增了AOF缓冲区；执行完写命令，会将命令同时追加到AOF缓冲区和AOF重写缓冲区
          - 重写完成后，将AOF重写缓冲区命令写入到新AOF中，并进行改名覆盖
      - 载入AOF时，会创建不带网络连接的伪客户端，执行所有命令，还原数据库 — 如果开启了AOF持久化，优先使用AOF还原数据库
  - 事件
    - redis服务器是事件驱动程序
    - 文件事件
      - 套接字 —监听→
      - I/O多路复用 —分发产生的事件→
      - 文件事件分派器 —关联→
      - 事件处理器
    - 时间事件
      - 定时事件
      - 周期性事件serverCron — 将所有时间事件放在无序链表中，遍历整个链表，查找已到达的时间事件，调用对应的处理器，复杂度O(N)
    - 事件调度
      - 总是优先处理文件事件，再执行时间事件，因此时间事件实际处理时间可能会比设定时间稍晚
      - 服务器最大阻塞时间由到达时间最接近当前时间的时间事件决定，避免频繁轮询以及阻塞过久 →
        ```
        def aeProcessEvents():
             #时间计算并根据时间进行阻塞
             processFileEvents()
             processTimeEvents()
        ```
      - 处理完文件事件，仍未有时间事件到达，会回到等待文件事件的状态
  - 客户端
    - 每一个与服务器连接的客户端都在服务器侧有相应的redisClient，使用链表保存
    - 服务器初始化时会创建一个Lua脚本的伪客户端，运行到服务器结束
    - 执行AOF载入的伪客户端，fd的值为-1，一旦执行完毕会被关闭
  - 服务器
    - 处理客户端请求
    - 使用serverCron负责管理服务器资源，处理持久化，默认每100ms执行一次
    - 启动过程
      - 创建redisServer结构体
      - 载入配置
      - 还原数据库
      - 启动事件循环
- **集群**
  - 复制
    - 同步：将从服务器数据库状态更新至主服务器所处状态
    - 命令传播：主服务器被修改时，主从服务器恢复到一致状态。命令传播期间通过发送心跳传送复制偏移量。
    - 旧版复制
      - 同步
        - 从服务器发送SYNC命令
        - 主服务器收到命令，使用BGSAVE生成RDB，发送给从服务器，从服务器载入
        - 主服务器将执行BGSAVE时产生的缓冲区里的写命令发送给从服务器
      - 命令传播
        - 同步完毕后，通过命令传播保持一致
      - 缺陷：命令传播时如果断线，从服务器仍然会发送SYNC命令，从而重新生成RDB文件，包含了已经同步的重复数据
    - 新版复制 — 使用PSYNC命令取代SYNC命令，包括完整重同步和部分重同步模式(用于断线重复制情况)
  - Sentinel
    - 本质是一个运行在特殊模式下的redis服务器，但不载入RDB和AOF文件，使用指定的port，也只支持特殊的命令 —结构→
      ```
      struct sentinelState{
        unit64_t current_epoch;
        dict * masters; // 代表被sentinel监视的服务器实例，可以是主服务器，从服务器，或者是sentinel
      }
      ```
    - 会创建与服务器之间的异步连接
      - 命令连接
      - 订阅连接
    - 监视同一个服务器的sentinel之间是可以互相感知的，它们之间会创建命令连接
    - 检测监视的实例是否下线
      - 主观下线 — 实例在超过一定时间没有回复，sentinel认为该实例主观下线
      - 客观下线 — 收到quorum数量的sentinel认为主观下线，标记为客观下线 — 当主服务器进入客观下线状态，监视sentinel进行协商选举领头sentinel，由其对失效主服务器进行故障迁移
  - redis集群
    - 通过分片进行数据共享，提供复制和故障转移
    - 数据库分为16384个槽(slot)
      - 每个节点可以处理0-16384个槽
      - 如果有任一个槽没有得到处理，集群处于fail状态
      - 数据库中的每个键都属于槽中的一个，利用算法计算键所在的槽值
        - 不同键可能位于同一个槽，键与槽的关系使用跳跃表保存
        - 跳跃表中的节点，分值代表槽，成员变量代表key，相同槽的节点按照字典序排列
        - 如果客户端请求的数据库键不属于当前节点槽，则会触发MOVED错误(不可见)，指引客户端转向正确节点
      - 每个节点会记录自己处理的槽，并且保存每个槽由哪个节点处理
      - 重新分片，原子性将某个节点的槽改派给另一个节点
      - 主节点用于处理槽，从节点用于复制主节点，在主节点下线时，通过选举新的主节点来处理命令请求
    - 消息
      - MEET
      - PING
      - PONG
      - FAIL
      - PUBLISH
      - —Gossip→ 消息正文一样，根据type来判断是哪种消息。每次发送时，都从已知节点中选取两个节点
- **发布/订阅**
  - PUBLISH
    - 在pubsub_channels中寻找channel，遍历订阅者名单，将message发送给频道订阅者
    - 在pubsub_patterns中查询相匹配的模式，将消息发给订阅者
  - SUBSCRIBE — 频道订阅关系保存在字典pubsub_channels中，key：订阅的频道->value:链表，代表所有订阅的客户端
  - PSUBSCRIBE — 模式订阅，将订阅模式以及订阅模式的客户端作为一个节点，放置于链表pubsub_patterns中
- **事务**
  - 将多个命令请求打包，一次性、按顺序地执行多个命令
  - 使用multi标记进入事务状态，除了exec/discard/watch/multi四个命令，其他命令服务器不会立即执行 → watch命令监视键，当键被修改时，会将所有监视该键的客户端的REDIS_DIRTY_CAS打开，打开了该标志位的客户端，服务器将拒绝执行其提交的事务
  - ACID
    - 原子性：事务队列中的命令要么全部执行，要么全部不执行，但不支持回滚
    - 一致性：不会执行有语法错误的命令，出错的命令不会对数据库做任何修改
    - 隔离性：使用单线程的方式执行事务
    - 持久性：由持久化模式决定，无持久化模式不具有持久性
- **分布式锁**
  - 通常使用setnx key value实现，返回1表明获锁成功，使用完后使用del释放 —使用redis做分布式锁真的可靠吗→ 问题
    - 1.没有设置超时时间，可能导致锁永远没有被释放。而追加expire语句，会导致所有操作不是原子性的
    - redis2.8后可以使用ex和nx来实现原子操作
    - 2.如果设置了超时时间，A执行太久导致超时释放，B可能获取锁，最后A执行完毕将B的锁释放
    - 可以通过守护进程，发现即将超时了就延长时间
    - 3.使用主从机制，保证redis锁高可用，但是可能导致从master获得的锁还没有同步到slave上，就有新的客户端从slave上获取到了锁
    - 针对多redis节点，只能使用redlock解决。从大部分节点中获取锁成功，才能视为获取成功