Redis

2024年8月21日 15:18

数据库的数据是存在磁盘中的，如果用户信息数据是经常被访问的话（热点数据），你将这些数据存放再磁盘中，你的磁盘IO是非常大的，会影响你的性能。所以，要将这些数据放入缓存中，提高访问效率

Redis是一个开源的高性能数据库，存储的数据结构是基于键值对的形式，是基于内存存储的，所以它的读写是非常快的

常用于数据库、MySQL的缓存、消息队列、分布式锁。

Redis就是为了解决web服务器高并发的问题才出现的

Redis 支持持久化，可将数据存储的硬盘，其他的数据库不一定支持持久化

支持集群，可以配置多个redis ，如果一个redis断了，不至于导致服务器无法使用，这样就有了redis 数据的同步问题

支持内存淘汰机制、发布订阅、事务（但不支持回滚）

每秒11w读取、8w写入

一、安装Redis

```
sudo apt install redis-server
```

查看版本

```
redis-server --version
```

```
# 修改配置文件
sudo vim /etc/redis/redis.conf
```

```
# 只允许来自 bind 指定网卡的 Redis 请求。
#如果没有指定，则可以接受来自任意一个网课的 Redis 请求
#bind 127.0.0.1

# daemonize 属性改为 yes 这样启动时可以后台启动
daemonize yes

# protected-mode 属性改为 no 关闭保护模式

# 大概第 507 行 修改配置文件里的 requirepass
requirepass 123456
```

```
# 重启redis服务器
sudo service redis restart

# 启动redis客户端
redis-cli
# 认证
auth password

# 启动客户端
redis-cil -h 127.0.0.1 -p 6379 -a 'password'
```

二、Redis常见用途

1、排行榜
排行榜，如果使用传统的关系型数据库来做，非常麻烦，而利用 Redis的 SortSet 数据结构能够非常方便搞定;

2、计算器、限速器
计算器/限速器，利用 Redis 中原子性的自增操作，我们可以统计类似用户点赞数、用户访问数等，这类操作如果用 MySQL，频繁的读写会带来相当大的压力;限速器比较典型的使用场景是限制某个用户访问某个 API的频率，常用的有抢购时，防止用户疯狂点击带来不必要的压力

3、好友关系
好友关系，利用集合的一些命令，比如求交集、并集、差集等，可以方便搞定一些共同好友、共同爱好之类的功能;

4、简单消息队列

5、

三、Redis数据类型

Redis 是一种高级的 KEY:VALUE 存储系统，其中的 value 支持五种数据类型(数据对象)。

string、list、set、zset、hash、　后发布：bitmap（位图）

Redis 的五大数据类型也称五大数据对象;Redis并没有直接使用这些结构来实现键值对数据库，而是使用这些结构构建了一个对象系统redisObject

这个对象，有几个属性：type（类型）、encoding（类型内部的编码格式）、ptr（void*类型的指针）

存字符串使用SDS，简单动态字符串，内部是一个结构体

1、string

由于这种类型是二进制安全的，所以可以把一个图片（音频、视频）文件的内容作为字符串存储

内部实现：

String 类型的底层数据结构实现主要是int 和 SDS（simple dynamic string 简单动态字符串）

SDS数据结构

```
struct sdshdr{
    long len; # 存放内容的长度，这样的话，求长度的复杂度就是 0（1）
    long free; # 字符数组剩余的大小
    char buf[0];
}

# 存储 "hello"

len 5
```

free 0
buf:['h', 'e', 'l', 'l', 'o', '\0']

SDS的获取字符串长度的时间复杂度是O(1)
Redis的SDS API是安全的（在字符串拼接时，如果空间不够，会自动扩容，不会造成缓冲区溢出）
SDS不仅可以保存文本数据，还能保存二进制数据（图片、音频、视频等）

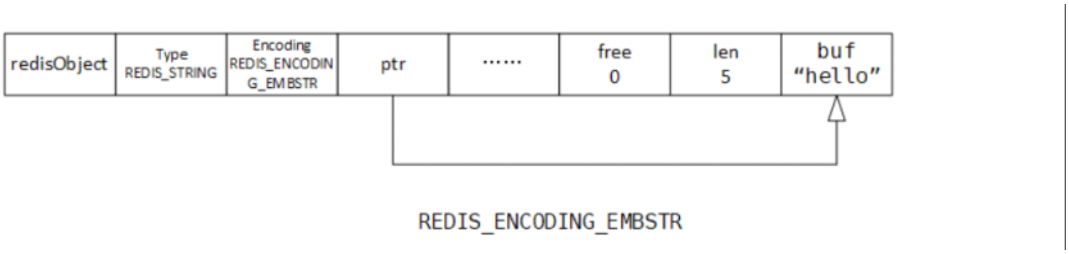
字符串对象内部3种编码格式：

int ：存整数，并且范围不超过long

int:如果一个字符串对象保存的是整数值，并且这个整数值可以用long类型来表示，那么字符串对象会将整数值保存在字符串对象结构的 ptr属性中(void*转换为 long)，并将字符串对象的编码设置为 int 编码。此时可以通过 incr 等命令对它进行运算并且不会改变编码格式。如果对它进行字符串追加操作，那么就会转换为raw编码格式，此时再用incr等命令就会报错

embstr：存的字符串字节数量较小，底层是一片连续的内存空间（这就导致了不能太大），是专门用来存放短字符串的一种优化编码的方式（通过一次性内存分配函数）

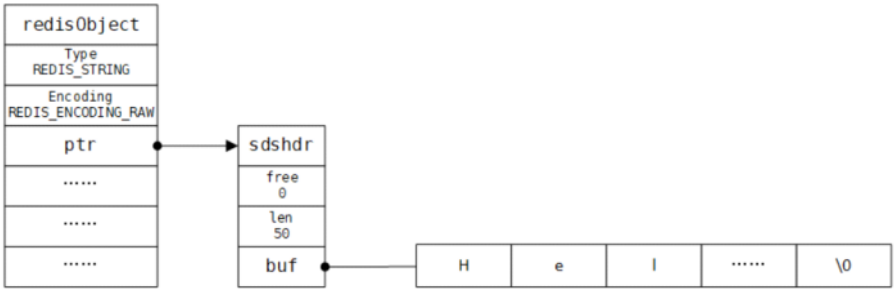
短字符串的长度在不同版本有所区别redis2是32字节、redis3.0-4.0 是39字节、redis5以上是44字节



REDIS_ENCODING_EMBSTR

它内部维护一个ptr指针，指向存储的字符串。这样的话，就非常好找

raw：存储比embstr长的字符串使用的编码方式，他需要使用两次内存分配函数来分配两块空间来保存redisObject和SDS结构体



REDIS_ENCODING_RAW

如图所示，它会开辟一个与embstr类似的空间（也有一个ptr指针），但是，它的ptr指针指向的是一个sds的结构体，结构体里存的是len、free还有buf（字符串）

embstr和raw的区别

embstr 编码将创建字符串对象所需要的内存分配次数从两次降低为一次。

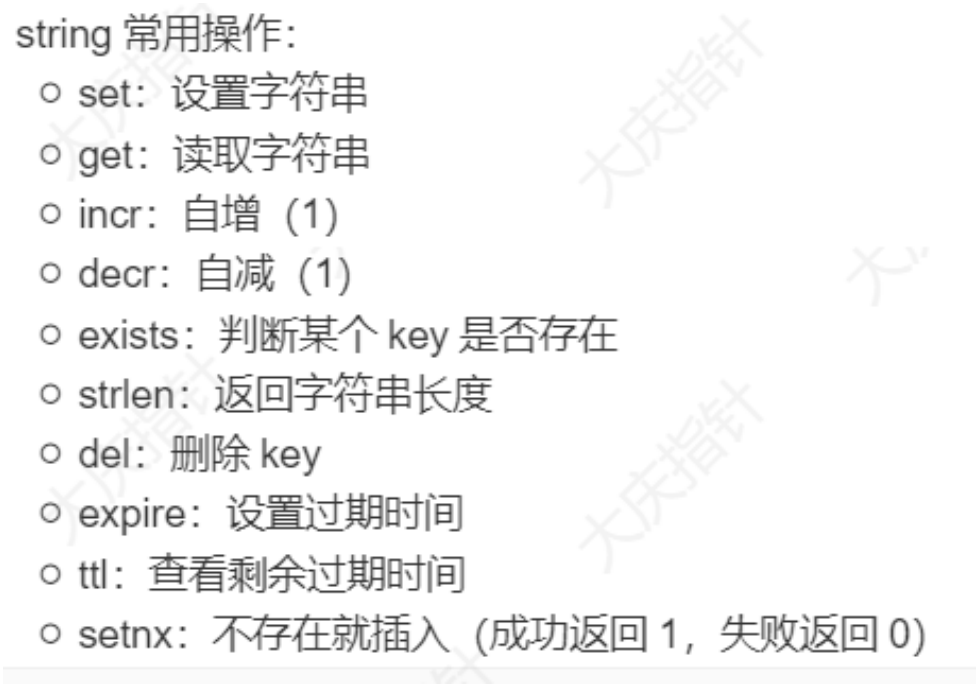
同样的 embstr 释放内存时只用调用一次内存释放函数，而 raw 需要调用两次。

embstr 编码的字符串对象保存在一块连续的内存中，可以更好地利用CPU 缓存提升性能。

在CPU读取磁盘数据时，会将该磁盘附近的内容读到缓存里，这样你第二次再来读的时候，就是从缓存里读数据，而不是磁盘里，就会变快

embstr 编码的字符串是只读的，因为如果字符串的长度增加需要分配内存时，整个redisObject和 SDS 都需要重新分配空间。所以当我们修改它的时候，它会转换为 raw 的编码形式，再对其进行修改，修改后就不会变成 embstr 编码的形式了。

String 常用操作



setnx 可以实现分布式锁

string应用

短链接服务

短链接服务就是把普通网址转换成比较短的网址，当前在一些社交媒体，用户增长、广告投放中有大量短链需求，将长连接转化为短连接使得链接变得清爽，用户点击率更高，同时能规避原始链接中
一些关键词、域名屏蔽等。常见微博、微信等社交软件中，比如微博限制字数为 140，如果需要包含连接，但是这个连接非常长，将会占用非常大的篇幅。短链接除了具有美观清爽的特性外，利用短链每次跳转都需要经过后端的特性，可以在跳转过程中做异步埋点，用于效果数据统计。

比较常见的短网址服务:

微博: http://t.cn

谷歌: https://goo.gl/

百度: http://dwz.cn/

```
a.b.sn.cn/OADFLKSD
1.DNS 服务器解析域名: https://a.b.sn.cn, 获取 IP 地址
2.向该 IP 地址发送 HTTP 请求
3.服务器根据短链接 key 获取对应长链接 (到 redis 中获取)
4.转发到对应长链接
```

Redis字符串数据类型的使用场景：session存储

在 Web 应用程序中，session 是一种跨页面和跨请求的数据存储机制，用于存储用户的会话信息(如登录状态、购物车内容等)。由于 session 需要在多个请求之间共享数据，因此需要一个持久化的储方案，Redis 字符串数据类型正是一种非常适合的选择。

限流控制：

利用 Redis 字符串数据类型的过期时间和自增命令可以实现简单的限流控制，防止系统被恶意攻击或者异常请求所影响

2、list

列表，redis 中的 list 在底层实现上并**不是数组**，而是**压缩列表**或者**双向链表**，也就是说对于一个具有上百万个元素的list 来说，在头部和尾部插入一个新元素，它的时间复杂度是常数级别的，比如用Ipush在 100个元素的头部插入新元素，和在上百万上千万元素 list 头部插入新元素的速度应该是相同的。

但是 lists 有这样的优势也同样有弊端,**弊端就是链表型 lists 的元素定位会比较慢**，而数组型 lists 的元素定位会快得多。

内部实现

双向链表或者是压缩列表，3.2版本之后，是二者的结合，quicklist

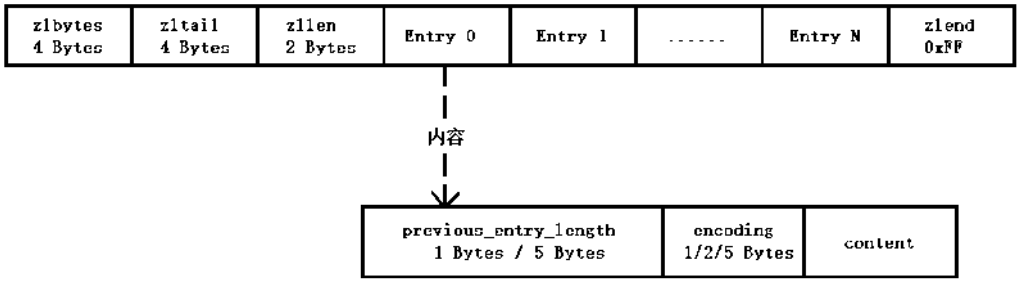
quicklist，当元素较少时，使用压缩列表，当元素变多时，将压缩列表当作双向链表的元素

什么时候用双向链表、什么时候用压缩列表？

如果列表的元素个数小于 512 个(默认值，可由list-max-ziplist-entries 配置)，列表每个元素的值都小于 64 字节(默认值，可由 list-max-ziplist-value 配置)，Redis 会使用压缩列表作为 List 类型的底层数据结构;（源码中，会判断每个元素的数据类型。如果不满足的话就会用双向链表、满足使用压缩列表）

如果列表中元素不满足上面的条件，Redis 会使用双向链表作为List类型的底层数据结构

压缩列表（空间连续）

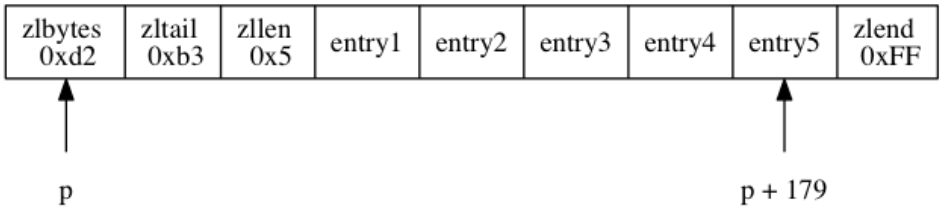


属性名	大小	作用
zlbytes	4 Bytes	记录整个压缩列表所占用的字节数
zltail	4 Bytes	记录压缩列表起始地址到表尾节点的偏移量，通过它可以快速找到表尾节点，无需遍历整个压缩列表
zllen	2 Bytes	记录压缩列表节点数量：当这个属性的值小于 UINT16_MAX （65535）时， 这个属性的值就是压缩列表包含节点的数量； 当这个值等于 UINT16_MAX 时， 节点的真实数量需要遍历整个压缩列表才能计算得出。
entryX	不确定	压缩列表的各个节点
zlend	1 Byte	其值为 0xFF，标记压缩列表的尾端

对于 entry：

属性名	大小	作用
previous_entry_length	1 Byte 或 5 Bytes	记录压缩列表前一个节点的占用的字节，如果前一节点长度小于 254 字节，那么 previous_entry_length 长度为 1 字节，如果大于 254 字节，那么长度为 5 字节，其中前 1 个字节为 0xFE，后 4 个字节才记录长度。
encoding	1 Byte 或 2 Bytes 或 5 Bytes	记录节点数据的类型及其长度对于 encoding 属性，如果其值的最高两位是以 11 开头，那么表示数据类型为整数。如果最高位是 00、01、10，那么数据类型为字节数组，00 表示 encoding 本身占用 1 字节（可以表示小于 63 字节的数据），01 表示占用 2 字节（可以表示小于 16383 字节的数据），10 表示占用 5 字节（可以表示小于 4294967295 字节的数据）
content	不定	记录数据本身

eg：



列表 zlbytes 属性的值为 0xd2(十进制 210)，表示压缩列表的总长为 210 字节。

列表 zltail 属性的值为 0xb3(十进制 179)，这表示如果我们有一个指向压缩列表起始地址的指针p，那么只要用指针p加上偏移量179，就可以计算出表尾节点 entry5 的地址。

列表 zllen 属性的值为 0x5(十进制 5)，表示压缩列表包含五个节点。

优点：

内存紧凑型列表，节省内存空间，提升内存使用率(不需要指针)

压缩就是和双向链表对比的，双向链表还得存上一个结点和下一个结点的指针（8个字节），而压缩列表是动态调整指针的大小的，而且最大也是5个字节。

缺点：

不能保存过多的元素，否则访问性能会降低
不能保存过大的元素，容易引发连锁更新的问题

连锁更新：如果一开始前面的数据所占字节数比较小（不到254字节），该节点就使用1个字节来存储前面结点的长度。
那现在我要在它之前，插入一个500字节的数据，该节点就需要将存储前一节点长度的那1个字节变为5个字节。
如果恰好该节点的长度在253，变为5个字节之后，该节点后面的节点，也需要进行长度改变
这样就造成了连锁更新

常用操作

lpush:在 list 左侧插入一个新元素
rpush:在 list 右侧插入一个新元素
lrange:从list 中指定一个范围来提取元素

list应用

分页查询:由于 Redis List支持按照索引访问元素，因此可以用来实现分页查询。

消息队列:由于 Redis List 支持阻塞式弹出元素，因此可以用来实现消息队列。生产者将消息推入List中，消费者则通过阻塞式地弹出元素来获取消息。

列表: Redis 的 List 是一个双向链表，可以在列表两端进行快速的插入和删除操作。因此，List 可以用来实现队列、等数据结构。

实现轮询队列: Redis的list还可以用来实现轮询队列。比如说，在在线客服系统中，我们可以将客户发送过来的消息 push 到一个 list中，并由多个客服进程并发地从list 中获取消息进行回复。这种方式既能够保证每个消息都会被处理到，又能够提高客服回复效率。

实现缓存淘汰策略: Redis 支持设置 key 过期时间，但是在某些情况下，我们也可以使用list 来实现缓存淘汰策略。比如说，在缓存系统中，我们可以将需要缓存的数据 push 到一个list 中，并设置list 的长度为一定值，当需要插入新数据时就可以将最早插入的数据pop 出去，从而实现缓存淘汰。

排行榜：由于 Redis List 支持按照索引访问元素，并且支持对 List 进行排序，因此可以用来实现排行榜。例如，在一个List中存储用户的积分信息，并按照积分从高到低排序。

地理位置:由于 Redis List 支持按照索引访问元素，并且支持对List 进行排序，因此可以用来实现地理位置。例如，在一个List中存储所有用户的经纬度信息，并按照距离从近到远排序，则可以通过取某个范围内的元素来获取附近的用户。

时间轴: Redis List 还可以用来实现时间轴功能。。比如微博应用中用户发表一条微博就会被添加到自己的时间轴上，而其他用户关注该用户后也会将其微博添加到自己的时间轴上。这个功能可以通过将每个用户的时间轴作为一个 List 来实现。

数据缓存: Redis List还可以用来实现数据缓存功能。比如需要缓存一些数据以加速读取速度，就可以将这些数据作为 List 中的元素，并且使用 Redis 提供的 LRU 淘汰策略控制缓存大小和淘汰过期数据。

实时聊天室: Redis 的 List 可以作为聊天室中消息的缓存区域。每当有用户发送一条新消息时，就将该消息添加到聊天室对应的 List中，并通过订阅/发布机制将该消息广播给所有在线用户。

流水线操作:由于 Redis 支持管道操作，因此我们可以使用Redis List 进行流水线操作。将需要执行的多个命令按顺序存储在 List中，并通过一次性发送给 Redis 服务器来减少网络延迟和通信开销

3、set

redis 的集合，是一种无序集合，集合中的元素没有先后顺序，没有重复的元素（自己去重）

内部实现：

如果集合中的元素都是整数并且元素个数小于 512个(默认值，配置可更改)，Redis 会使用整数集合作为底层数据结构

如果集合中的元素不满足上面的条件，Redis 会使用哈希表作为底层数据结构

常用操作：

sadd:添加新元素
smembers:列出集合中所有元素
sismember:判断元素是否在集合中
scard:返回集合中元素个数
srem:移除集合中一个或多个元素，如果移除的不存在就不处理
sinter:对两个集合求交集
sunion:对两个集合求并集
sdiff:对两个集合求差集
setnx:向 Redis 中添加一个 key，只用当 key 不存在的时候才添加并返回 1，存在则不添加返回 0

set应用：

标签:比如博客网站常常使用到的兴趣标签，关注类似内容的用户可以利用一个标签把他们进行归并

计数相关的功能，利用 set 集合当中元素的唯一性，可以快速实现实时统计的功能

实时在线状态:将在线用户ID 存储在 Redis set 中，并利用其过期时间特性，可以实现实时在线状态的判断

1.当用户登录时，将用户的 ID 添加到 Redis set 中，并设置过期时间为一定的时间
比如 10 分钟;
2.当用户每次访问网站时，更新 Redis set 中对应用户 ID 的过期时间;
3.当需要判断用户是否在线时，只需要查询 Redis set 中是否包含该用户 ID 即可。
4.当用户退出时，将其 ID 从 set 中删除即可

标签系统：将标签作为 set的元素，可以方便进行标签搜索和统计。

标签系统是一种常见的数据管理方式，可以用于将相关的信息进行分类和组织。在这种系统中，每个息都可以被分配一个或多个标签，这些标签可以用来搜索和统计相关信息。

在 Redis中，可以使用 get 数据类型来实现标签系统。每个标签都可以被看作是一个 set元素，而每个信息都可以用一个 get 来表示它所拥有的标签。这样，就可以方便地进行标签搜索和统计。

比如博客中可以用文章名称作为 key名，用标签作为 set 中的元素，这样可以通过get 中求交集，并集等操作计算不同文章中所拥有的共同标签等功能。或者用标签作为key，用文章名称座位 set中的元素，就可以根据标签搜索文章。

分布式锁：通过 Redis 的 SETNX命令实现分布式锁，避免多个客户端同时修改同一个资源造成冲突。

一定要设置过期时间，一旦加锁之后，服务器挂掉了，而你没设置过期时间，这个锁就无法解锁了，其他服务器就无法访问该数据了

```python
import redis

redis_client = redis.Redis(host='localhost', port=6379, db=0)
lock_key = 'my_lock'

def acquire_lock():
    # 使用 SETNX 命令尝试获取锁，如果返回 1 表示获取成功，否则获取失败
    success = redis_client.setnx(lock_key, 1)
    if success:
        # 设置锁的过期时间，防止锁一直占用
        redis_client.expire(lock_key, 60)
        return True
    else:
        return False

def release_lock():
    # 删除锁
    redis_client.delete(lock_key)
```

好友关系：可以将每个用户的好友列表作为 set 中的一个元素，然后通过交集、并集等操作实现好友推荐、共同好友查找等功能。

4、zset

redis不仅提供了无序集合，还提供了有序集合。有序集合中的每个元素都关联一个序号，这就是排序的依据。一般我们将redis 中的有序集合叫做zsets，这是因为在 redis 中，有序集合相关的操作都是以z开头的

内部实现：（压缩列表、跳表）
如果有序集合的元素个数小于 128 个，并且每个元素的值小于 64字节时，Redis 会使用压缩列表作为底层数据结构
如果不满足上面的条件，Redis 会使用跳表作为底层数据结构

zset常用操作

zadd:新增一个有序集合，并加入一个元素
zrange:列出有序集合中的元素
zrem:从有序集合中删除元素
zcard:返回有序集合中的数量

zset 应用

排行榜:最经典的应用场景之一。比如需要统计用户的分数并且实时更新排行榜
浏览量统计:可以使用 zset 存储文章id 及其对应的浏览量，每当有一次浏览时就 +1
最近活跃用户:存储每个用户最近登录的时间戳，并设置分数为时间戳。这样就可以按照时间戳排序，找到最近登录的用户

5、hash

哈希是从 redis-2.0.0 版本之后才有的数据结构。hashes 存的就是字符串和字符串值之间的映射，比如存储用户的信息就很适合用哈希结构。

内部实现

如果哈希类型元素个数少于 512个(默认值，可更改配置)，所有值小于 64 字节(默认值，可更改配置)，Redis 会使用压缩列表作为底层数据结构

如果不满足上面的条件，Redis 会使用哈希表作为底层数据结构o Redis7.0后，废弃压缩列表，改为listpackhash

常用操作:

hset key field value:指定一个 key，设置字段名所对应的值o hget key field:获取指定 key 和字段对应的值(也可以批量添加)，成功返回操作成功的字段数量
hmset key field value [field value...]:批量设置某个 key 的字段和对应值，成功返回 OK
hmget key filed [filed...]:批量返回某个 key 中指定的字段和其对应婚
hgetall key:用于返回一个 hash 中全部的字段和值o hexists key field:是否存在某个字段
hdel key field:删除 key 中的字段，可以删除所有字段，如果删除所有字段，相当于删除了这个 key
hincrby key filed n:增减操作，hash 中没有类似 decrby 的命令如果想要减值就直接用负数
hkeys key:获取所有的字段值
hvals key:获取所有的值

hash 应用:

存储用户信息
存储对象
存储配置信息
数据过滤:可以使用 Redis 哈希数据类型来存储需要过滤的关键字，然后在应用程序中进行匹配和过滤
统计网站访问量:可以使用 redis 哈希数据类型来记录网站每个页面的访问量，以便进行统计分析
分布式锁:可以使用 Redis 哈希数据类型来实现分布式锁，保证多个进程或线程对同一资源的互斥访问
统计用户行为:如果你需要统计用户的行为，如点击次数、购买次数等，那么可以使用哈希数据类型来进行统计。通过使用哈希数据类型，在每个键中记录用户对应行为的数量，并且可以很方便地进行更新和查询

六、缓存

1、缓存雪崩

redis的作用就是存放部分数据到缓存中来，当然存放的都是一些热点数据，不常访问的数据还是在数据库中。
但是这些热点数据也不是一直存在缓存中的，也会设置过期的时间，不然缓存里的数据就会越来越多的，

情况1：大量key过期
但是你设置的这个过期时间就会有问题了，比如说你一个游戏刚开服，一开服就有大量用户涌进来。这个时候你的用户访问数据库的压力时很大的，可能导致数据库宕机。这个时候就要使用缓存进行预热嘛，提前将用户信息放到缓存中来，那你这个时候添加几乎是同时的，如果过期时间是一致的话，过几天这些数据就会同时过期，缓存就大量失效了，那么这个时候如果并发量比较高的话，数据库压力变大，就可能导致你数据库宕机了。这时候，你就算重新启动数据库，还是会有大量的请求，数据库大概率还是宕机

在某一个时刻，出现大量缓存失效的情况下，会导致大量的请求直接访问数据库，导致数据库压力过大。如果在高并发的情况下，可能会直接导致数据库宕机。此时就算运维重启数据库，还会有大量新的请求再次导致数据库宕机。
雪崩，崩的是数据库
产生原因:
1、Redis 宕机
2、大量 key 采用相同的过期时间
解决方案:
1、过期时间增加随机值，不让它同时过期（从让用户访问数据角度考虑）
2、熔断机制，做一个流量检测，当流量达到一定的阈值时，直接返回系统正忙”等提示，防止过多请求打在数据库上。（直接拒绝访问数据库）
3、提高数据库容灾能力，比如使用分布式的数据库（提高数据库的能力）
4、提高 Redis 容灾能力，搭建 Redis 集群（提高redis能力）
5、缓存预热(python 定时脚本)

2、缓存击穿

与缓存雪崩相区别，雪崩是大量热点数据都失效了，而击穿是某个热点数据失效。

缓存中的某个热点数据过期了，在这个时候大量的请求想要获取该热点数据，就会直接访问数据库，导致数据库压力剧增。

产生原因:
某个热点数据过期
解决方案:
1、如果业务允许，热点数据不设置过期时间（一般不行的）
2、热点数据过期前，通知后台线程更新缓存和过期时间
3、使用互斥锁，如果缓存失效，只有拿到锁才可以查询数据库，降低了同一时刻访问数据库的请求量，但是这样会导致系统性能变差
使用互斥锁的话，即使有大量的请求来访问缓存，虽然缓存失效了，但我只让一个请求来数据库那数据，再带回到缓存。后面的请求也就能访问到缓存了。这样就限制了数据库的大量访问

3、缓存穿透

与缓存击穿相比较，击穿是正常的请求，而缓存穿透是不正常的请求，比如大量请求访问某个不存在的数据。

大量请求想要访问一些不存在的数据(非法 key)，这种数据在 Redis中不存在，那么就会有大量的请求落到数据库中，数据库中也不存在这种数据，不能够更新缓存数据，导致数据库压力剧增。跟击穿很像，但根本上的区别在于访问的这些数据在 Redis 中是否存在(是否是合法数据)。如果黑客访问接口，传入大量非法数据去访问，就会把数据库冲宕机，在日常开发中我们需要对参数做好格式校验(如id 不为0等)

产生原因:
请求访问的数据既不在 Redis 中也不在数据库中，发生缓存雪崩和击穿时，数据库中是保存了请求要访问的数据的，一旦缓存恢复对应的数据就可以减轻数据库的压力。
1、误操作，把数据都删了
2、黑客恶意攻击
解决方案:
1、非法请求的限制(参数检测、IP 检测、请求频率：10秒内只能访问5次 限制等）
2、缓存空值或默认值(会导致 Redis 中存在无效 key，redis压力较大)
3、使用**布隆过滤器**判断数据是否存在，不通过查询数据库判断

4、布隆过滤器

主要是为了解决海量数据的存在性问题。

它不能判断某个数据一定存在，但能确定某个数据一定不存在

应用场景:
- 对于海量数据中判定某个数据是否存在且容忍轻微误差
- 解决 Redis 缓存穿透问题
- 解决推荐类业务(刷到的不再刷到)
- 黑名单过滤(垃圾邮件过滤)
- 网络爬虫(判断某个 ur 是否访问过，避免重复爬取相同页面)

概念:
由二进制向量(位数组)和一系列随机映射函数(哈希函数)两部分组成的数据结构。相比于list、map、set等数据结构，占用空间更少并且效率更高。但缺点是返回的结果并不是非常准确的，添加到集合的元素越多，错误的可能性就越大，且删除困难。
布隆过滤器会使用一个较大的 bit 数组来保存所有数据，数据中的每个元素都只占用 1bit，每个元素为0或1(代表false 或 true)，这也是布隆过滤器节省内存的核心。申请 100w个元素的位数组只占用1000000Bit/8=125000Byte=125000/1024Byte~122KB的空间
原理:
当一个元素加入到布隆过滤器中的时候
1、使用布隆过滤器中的哈希函数对元素值进行计算，得到哈希值(有几个哈希函数就可以得到几个哈希值)
2、将得到的哈希值对位数组的长度取模，得到每个哈希值在位数组的对应位置

3根据得到的对应位置，在位数组中把对应下标的值置为1

判断一个元素是否存在的时候:

1、对给定元素再次进行哈希计算，取，得到对应数组中的位置

2、得到位置后判断位数组中对应的每个元素是否都为 1，如果都为1，则说明这个值存在布隆过滤器中，如果有1个或多个不为 1，则说明不存在

注意

1、不同的值可能哈希出来的位置相同，这种情况可以适当增加位数组的长度或调整哈希函数。

2、布隆过滤器返回存在，小概率会误判。布隆过滤器返回不存在那么一定不存在

七、数据一致性的问题

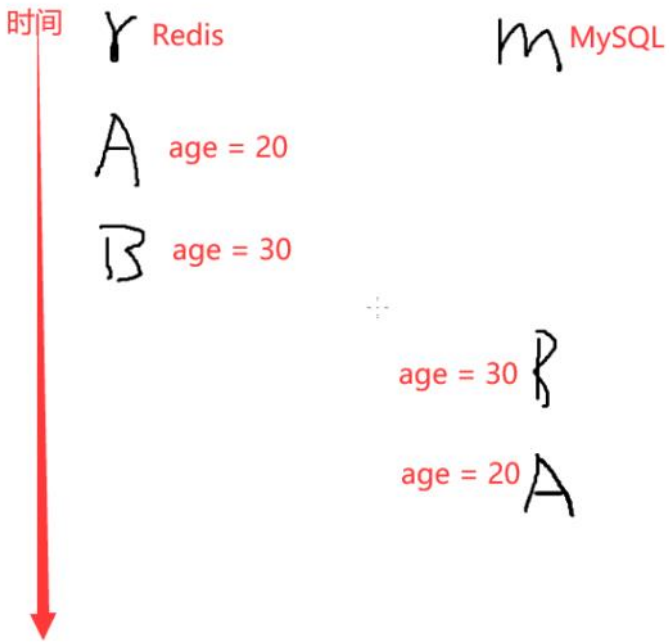
就是 Redis与数据库中的数据不一致的问题

如果有两个请求去修改数据，是先修改数据库？还是先修改Redis？

1、先更新Redis的数据，再去更新数据库中的数据（就以修改age年龄为例子）

比如，服务器A、服务器B，服务器A将Redis中的数据先修改，将18修改为20，然后服务器B将年龄修改为30，这个时候，redis中的age数据是30

但是在写入数据库时，B服务器的数据先写入了30，A服务器的数据后写入了20，就会导致redis与数据库中是数据不一致



2、先更新数据库，再更新Redis，也是一样的道理

而且也会造成性能的浪费，在写多读少的场景下，（比如你写10次读1次，每写一次数据库，就需要更新一次Redis）

每次更新数据，都先去数据库更新，再更新缓存，这样的话，性能也会下降。

数据库中的一个字段更新，涉及到多个表的更新，这样缓存里也是一样的

数据库中的数据类型与缓存中的不一致，数据库中是datetime，Redis是字符串

数据库里面的数据要经过聚合等计算，才能存放到缓存，这样的话就会有大量的计算

这样计算量很大的情况，你每更新一次数据库，据需要更新一次缓存，很浪费性能

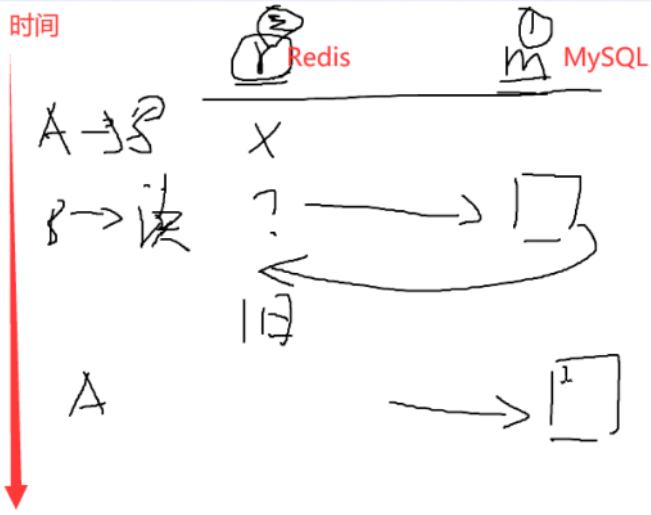
3、在写入（更新）时，只更新数据库，在读取时，再更新Redis

写入时，先删除Redis的数据，再去数据库写入数据

这样你写10次读1次的话，只写入10次数据库，更新一次缓存

但可能出现一致性问题，比如在一个请求为写入，一个请求为查询时，

请求A写入数据时，将缓存删除，此时请求B进行查询，这个时候B只能去数据库中查找数据，但这个时候A还未将更新后的数据写入数据库，B拿到的就是旧的数据。之后的读取拿到的也是旧数据



延时双删，写入的时候，删除两次缓存，第一次是在开始写入时，删除缓存数据，第二次是延时等待一段时间后，再删除一次缓存（B拿到的旧数据），确保更新后的数据已经写入数据库，后面的读操作拿到的是新的数据。

这样虽然B拿到的是旧的数据，但后续的读操作拿到的就是新的数据了。因为我们使用Redis就是要牺牲一部分高准确性，来保证它的快速高效

4、先更新MySQL，再删除Redis

还是两个请求，A是更新数据，B是读取数据

B来读取Redis，缓存里没数据（之前的操作已经删除了），没读到，然后去数据库读取旧的数据，然后A来更新数据库，然后删除Redis，最后B再拿着旧的数据更新Redis，此时就是数据不一致了，之后的读操作，拿到的的都是旧的数据

但是这种情况是很难产生的，我们都知道，数据库的读操作时肯定比写操作要快的，这样的话，在Redis中删除缓存的操作，大概率是在B使用更新旧数据更新Redis之后才进行的。也就是说，大概率是B先更新旧的数据，然后A再删除旧的Redis。这样后续的读操作，拿到的就是新的数据了。

但是这种情况也是有可能发生的。想解决，也可以加一个延时双删。

那如果第二次的延时删除也失败了呢？
1、在业务代码中，添加消息队列，用于验证双删是否成功。这样的话，会增加业务逻辑的复杂度，也会降低效率
2、解决延时双删会导致效率降低的方法，可以通过异步的方式。第二次等待删除的逻辑，交给别的线程去删。主线程只解决
3、数据库里有binlog日志。DDL和DML语言都会记录在日志中。只需要订阅日志。一旦发生更新，就将Redis中的数据再删一遍。（这个操作也是在异步下实现的）

数据库崩溃，可以用日志恢复。用于主从数据库的数据同步

八、高可用

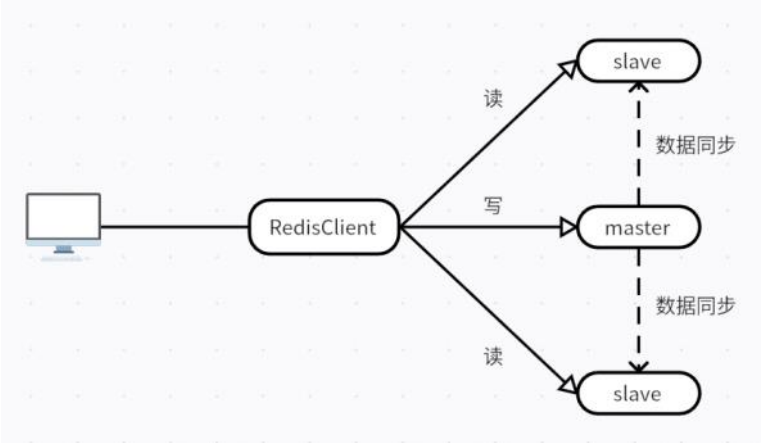
单个redis主要面临两个问题，一是redis挂了怎么办，一个是无法支持高并发

如果 Redis 是单机工作，会面临三个问题:
1、如果 Redis 服务器发生宕机，那么到 Redis 重启完成的这段时间内，无法提供任何服务
2、如果 Redis 服务器硬盘出现问题，数据可能全部丢失
3、单机 Redis 的并发能力是有上限的，如果想要进一步提升 Redis的并发能力，就需要搭建主从集群，实现读写分离

为了避免单点故障所带来的影响，我们可以将这些数据备份至其他的服务器上，这些服务器也可以对外提供服务。即使一台 Redis 服务器因为各种原因无法提供服务，但仍然有其他的 Redis 来填补空缺提供服务。

解决上面的两种问题，就是搭建集群，实现一个主从结构（不能严格保证数据的一致性，保证强一致性只能是上锁，上锁就会降低你的效率）

8.1 主从同步



主从结构就是主节点负责写操作，从节点负责读操作。主节点写入数据后，需要将数据同步到从节点，来保证数据的一致性。redis一般是读多写少，所以需要多个从节点负责读操作，来增加redis的并发能力。

主从结构里，从节点是不能修改数据的（通过修改配置 slave-read-only = yes），如果从节点可以修改数据的话，主节点是不知道数据有修改的，这样就会导致主从节点数据不一致。

删除数据的话，主redis会模拟一个删除的信号，发给从redis。来同步从节点的数据。

主从同步数据有两种模式：全量同步、增量同步

全量同步就是将主节点中的所有数据都同步给从节点。
增量同步就是将主节点增加或修改的地方同步给从节点。

所以在启动服务器的时候，redis的第一次同步是全量同步，后续的是增量同步

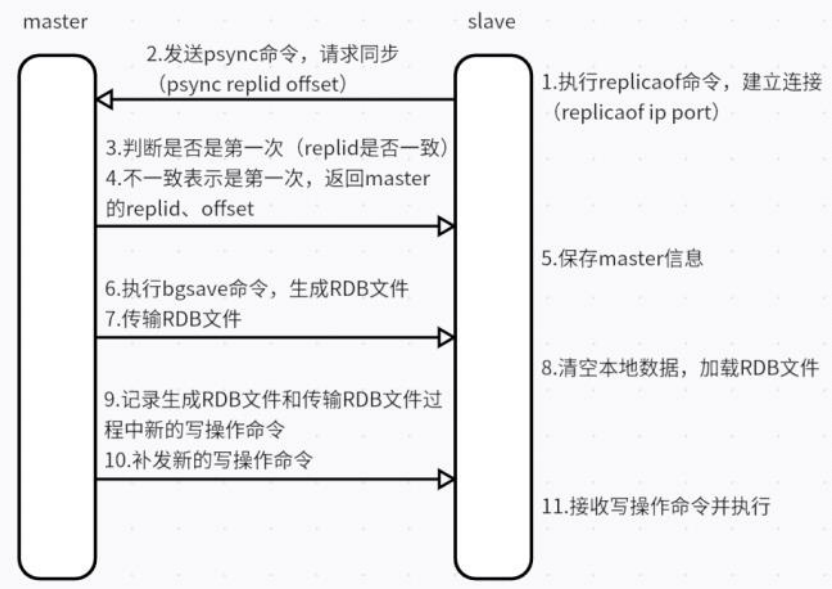
8.1.1 全量同步

- **replication id**
  （1）表示从那个主节点上获取数据
  （2）这个参数由主节点生成，主节点启动或者从节点提升为主节点时，都会生成replication id
  （3）每次生成的replication id都不相同
  （4）当从节点和主节点建立连接后，主节点会发送它的replication id给从节点
- **offset（偏置位）**
  -1表示获取全量数据，正整数表示从当前偏移位获取数据
  （1）主从节点都会维护偏移量
  （2）主节点的偏移量
  ① 主节点会收到很多写操作的命令，每个命令占据几个字节，主节点中的偏移量存储的就是这些写命令字节数的累加和
  （3）从节点偏移量
  ① 表示这个从节点的数据同步到的偏移位
  ② 如果与主节点中的偏移量相同，表示数据一致
- 如果主节点与从节点中的 replicationid 和 offset完全一致，就说明数据同步

- **replication buffer 复制缓冲区**
  在执行 bgsave 命令生成 RDB 文件时，可能会有新的写操作此时的新数据会被存放在replication buffer 中

  如果主节点传输 RDB 文件并且从节点加载 RDB 文件耗时过长，主节点接收的写操作较多，就可能会导致replication buffer溢出。溢出后，主节点会关闭和从节点的连接，重新开始全量同步，最坏的情况下可能会陷入死循环。

  全量同步时，发生网络中断，也会向这个buffer中写入数据，缓冲区溢出的话，重新执行全量同步



全量同步步骤：

1、从节点（slave）执行一个replicaof命令，从redis与主redis建立连接。 （命令后面跟一个ip和端口）
replicaof ip port。 建立连接之后，主节点会给从节点发送一个replication id

2、发送psync命令，请求同步。该命令携带两个参数 replicationid 和 offset （psync replicationid offset）

3、主节点根据从节点传过来的id和偏移量offset进行判断本次请求是否能够进行全量同步（选择全量还是增量）
判断id是否一致
（1）如果从节点想进行增量同步，但是发过来的id和主节点不一致，主节点就不会进行增量同步，而是进行全量同步
（2）主节点会维持一个复制积压缓存区（**repl-backlog-buffer**），在进行增量同步时，由于网络问题，断开再次连接了。这个时候就回去buffer中进行同步数据，如果offset没在这个缓存区内，他就无法进行增量。只能进行全量同步

4、如果判断进行全量同步，主节点会返回自己的 replication id 和 offset给从节点

5、从节点保存master的信息

6、主节点执行bgsave命令，异步生成一个RDB文件，文件中存的是主redis中的数据
- AOF文件是持久化文件，存的是操作命令。你使用这个文件恢复数据的话，其实就是将它的命令执行了一遍。
- 所以**RDB文件**的恢复速度是快的。相当于是复制一份快照
- bgsave是新版本的命令，旧版本是save命令，save命令会阻塞主进程
- bgsave是异步生成RDB文件，异步的话，就会有一个问题，就是你在生成这个文件的是，依然可以改变你的数据。就会涉及到写时复制的问题。新的写操作会存到buffer中

7、主redis传输RDB文件给从redis文件

8、从节点清空本地数据，加载RDB文件

9、主节点记录生成RDB文件和传输RDB文件过程中，产生的 新的写操作命令，存放在bufer中
- 所以要提前预估缓冲区的大小，如果buffer超了，他会直接放弃本次全量同步，重新进行全量同步，所以说，如果你没有预估好缓冲区的大小的话，就有可能一直进行全量同步。

- 得提前预估生成RDB和传输RDB文件这个过程中，可能产生多少数据

10、主节点 补发新的写操作命令给 从节点

11、从节点接收写操作命令并执行

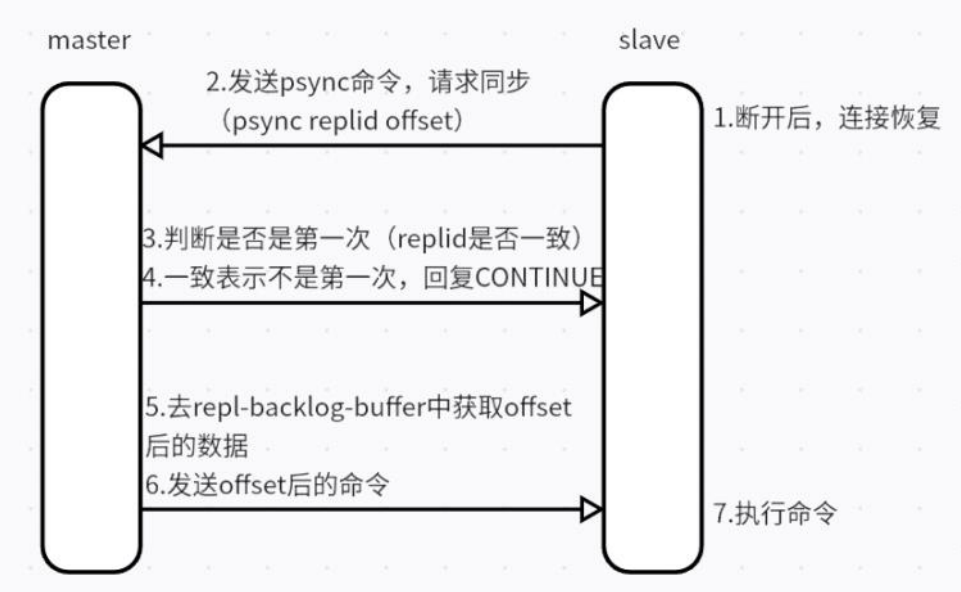
主节点决定全量同步还是增量同步
（1）如果主节点版本低于 2.8，返回-ERR，从节点重新发送 sync 命令执行全量同步
（2）如果主节点为新版本，replicationid 与从节点发送相同，并且从节点发送的 offset 之后的数据均在复制积压缓冲区中，返回+CONTINUE，进行增量同步
（3）如果主节点为新版本，但 replicationid 与从节点发送不同，或者从节点发送的 offset 之后的数据不在复制积压缓冲区了，回复+FULLRESYNC，进行全量同步

8.1.2 增量同步

- **命令传播**
（1）主从服务器完成第一次同步后，双方会维护一个TCP 连接
（2）后续主服务器会通过这个连接保证主从服务器数据的一致性
（3）长连接是为了避免频繁 TCP 连接和断开降低性能

- **repl-backlog-buffer复制积压缓冲区**

（1）内存中的一个队列，大小由repl-backlog-size 指定，默认为 1MB
（2）所有从节点共享一个缓冲区
（3）在主从命令传播阶段，主节点除了将写命令发送给从节点，还会发送一份到缓冲区中，作为写命令的备份。
（4）缓冲区中不仅存储最近的写命令，还会存储每个字节对应的偏移置，
（5）主要是用于全量同步中断后的增量同步
（6）在网络恢复后，如果从节点想要读取的数据不存在于主节点的复制积压缓冲区了，那么就会采用全量同步的方式，为了避免这种情况，我们需要根据网络情况(断线到恢复平均所需时长)和主节点写操作产生的频率(每秒产生写命令数据量的大小)来估算复制积压缓冲区的大小



1、一个是断开后，恢复连接，一个是全量同步之后直接请求同步数据

2、发送psync命令，请求同步。该命令携带两个参数 replicationid 和 offset （psync replicationid offset）

3、主节点判断id是否一致，判断是否是第一次连接

4、id一致，说明不是第一次连接，要进行增量同步，回复从节点一个 continue 表示继续。
不用回复id和offset了，从节点已经有这俩个数据了

5、去repl-backlog-buffer中获取offset中的数据
新的写操作都会写到这个缓冲区中，

- **sync:**
  在redis2.6 及以前的版本，同步采用 sync 命令，当从库断开连接后，不能够继续同步，而是重新进行全量同步
- **psync(第一版):**
  redis2.8版本中引入了 psync 命令进行主从数据同步，此时 psync依赖runid(服务器唯一标识 id，每一个runid标记一个redis)replication backlog buffer(复制积压缓冲区)、offset(偏移量)
- **psync(第二版):**
  redis4.0版本优化了psync，实现了即使redis 重启也能够实现增量同步。主要是增加了两个复制id(replicationid:master_replid、master_replid2)，一般情况下用不到 master_replid2

  - **特殊情况**
    - 主节点 A和从节点 B
    - 主节点A生成 replid，从节点 B获取到主节点的 replid
    - 主节点A 与从节点 B 通信时网络异常
    - 从节点 B 认为主节点 A 挂掉了，从节点 B 自己成为主节点，并生成了一个自己的 replid
    - 节点 B 通过 replid2 记录了之前主节点A的 replid

- 后续网络恢复时，节点 B 可以根据replid2 来重新变成主节点 A 的从节点
- 这个过程需要手动干预，但哨兵模式可以自动完成这个过程

8.2 拓扑结构

1、一主一从

写请求太多时，主节点压力增加(可通过关闭主节点 AOF，在从节点开启 AOF 来分担主节点的持久化压力)

2、一主多从

主节点数据发生改变时，需要将数据同步给所有从节点
从节点个数越多，同步数据时需要传输的次数越多，需要更大的网络带宽，成本就越高

3、树形主从

主节点不需要像一主多从模式下更大的网络带宽
同步延时比一主多从模式更长

8.3 配置主从结构

1、步骤

找到 Redis 配置文件，复制出两份配置文件作为从节点配置文原来的配置文件作为主节点的配置文件。

修改两份从节点配置文件(端口号、主从结构配置)

```
port 6380
slaveof 127.0.0.1 6379

port 6381
slaveof 127.0.0.1 6379
```

2、新建两个工作目录，分别放入两个配置文件

```
/etc/redis/slave1
/etc/redis/slave2

修改配置文件中的 dir
dir /etc/redis/slave1
dir /etc/redis/slave2
```

3、根据配置文件，分别启动两个Redis服务器

4、查看

- **使用 netstat 命令查看**

  netstat -anp | grep redis

  （1）三个运行的 Redis 服务器

  （2）两个 Redis 从节点和主节点之间的 cp连接

- **使用 info replication 查看主从结构信息(在 redis 命令行中执行)**

- **测试**

  （1）主节点中写操作，发现从节点中的 key-value 也会变化

  （2）从节点中写操作，报错，从节点为只读

- **断开主从关系**

  （1）使用 slaveof no one(在 redis 命令行中执行)

  （2）断开后，从节点之前存储的数据不会丢弃

- **切换主从关系**

  （1）先断开现有主从关系

  （2）从节点再使用 slaveofip port 命令重新绑定主从关系

  （3）这种修改方式是临时的，如果重启 Redis 服务器，会按照配置文件中的内容来建立主从关系

8.4 主从数据不一致

1、现象

客户端在主节点中读取的值与从节点中读取的值不相同

2、原因

主节点与从节点的数据同步是异步进行的，传输过程中，主节点是可以执行写操作的。这个就是造成输出数据不一致的原因。

主节点与从节点的数据同步是异步进行的，一定会有某一时刻出现数据的不一致，无法保证数据的强一致性。因为主节点收到写操作命令后，会发送给从节点，但并不会等待从节点也完成写操作再将结果返回给客户端，而是主节点执行完命令后直接返回给客户端。如果要保证强一致性，主节点就需要等待所有从节点也完成写操作再返回，这样就违当了使用缓存的初衷(提升性能)

3、解决方案

（1）尽量保证主从节点间网络状态良好，避免主从节点在不同的机房，减少传播时间。

（2）有些情况下我们需要更强的一致性，比如用 Redis 作为分布式锁的情况下。可以考虑 红锁RedLock(在某段时间内成功获取一半以上的锁就是有效锁)

红锁：比如一个主节点下面有5个从节点，他会保证，数据同步一半以上的从节点之后，才会给客户端返回数据。具体实现，还是和业务挂扣的。根据业务要求来决定如何配置

（3）开发程序进行监控主从节点之间的复制进度，如果复制进度大于某个预设的阈值，就不让客户端在这个从节点读取数据。

我们可以设一个阈值是100，通过检测offset，比如主节点的offset是10000，而从节点的offset是9800，主从之间的差值大于100，客户端就不能在这个从节点上获取数据。

8.5 哨兵机制

主要是为了解决主从结构的问题中，主节点挂掉的问题

因为在主从结构里，只有主节点可以写数据，从节点只等读数据。一旦主节点挂掉了，你的写操作就无法进行了

这个时候，只能通过人工干预，来将某个从节点提升为主节点。那你半夜3点挂掉了，还需要去手动进行干预

所以就引入了哨兵机制

用来监测你各个节点的网络状况

但是一个哨兵监测的话，万一哨兵的网络不好，也会认定是节点挂掉了。所以要多个哨兵同时监测

哨兵有三个阶段：监控、选主、通知

如何监控的？

（1）哨兵，每隔一段时间，给主从节点发送命令 ping一下，看看有没有响应

（2）如果没有响应，哨兵会标记为**主观下线**（这个哨兵认为节点挂掉了），就会给其他的哨兵发送请求，询问该节点是否网络异常

（3）其他哨兵去给节点发送命令，如果在多个哨兵那里也是无响应的话，就会标记为**客观下线**
不需要所有的哨兵都标记为主观下线，大部分标记为主观下线之后，就会被标记为客观下线

如何选主？

（1）当主节点被标记为客观下线时，就会进行选主
（2）选主之前，会在多个哨兵中决定，由哪个哨兵完成故障转移（确定谁去给节点发送命令）
① 一般都是第一个发现主观下线的哨兵作为候选者。
② 候选者哨兵，给其他哨兵发送命令，通知其他哨兵，它想称为故障转移的哨兵
③ 拿到其他哨兵一半以上的赞成票（它自己也算一票），还有某个值大于阈值（值是几，就需要拿到几票），就会称为故障转移的哨兵

（3）给从节点发送命令，将某个从节点提升为主节点（根据从节点的状态，筛选最合适的主节点）
如果有多个合适的从节点，就选application id最小的从节点

通知
选主之后，通知其他节点新的主节点