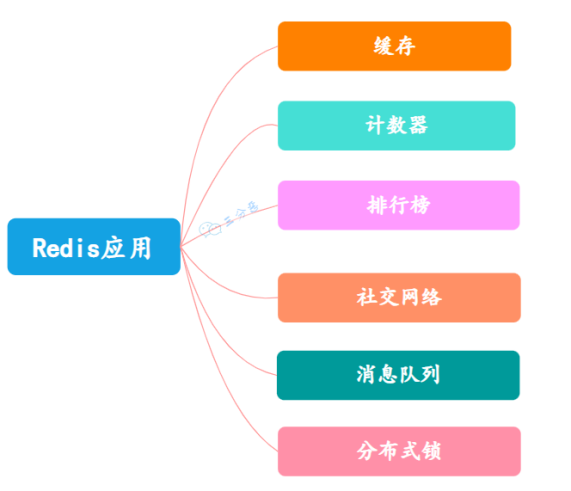
# 一、redis简介

Redis(Remote Dictionary Server)即远程字典服务，是一个开源的高性能键值对存储系统，具有快速、灵活和可扩展的特性。它是一个基于内存的数据结构存储系统，可以用作数据库、缓存和消息代理。Redis数据存储在内存中，因此能够提供极快的读写操作，它采用单线程模型和异步I/O，避免了多线程的竞争和阻塞，从而达到了非常高的性能。

Redis能够解决传统关系型数据库在高并发的情况下读写过慢的问题，是一种以key-value形式存储数据的一款nosql关系型数据库之一。

# 二、redis的应用



**1. 缓存**

这是Redis应用最广泛地方，基本所有的Web应用都会使用Redis作为缓存，来降低数据源压力，提高响应速度。

**2. 计数器**

Redis天然支持计数功能，而且计数性能非常好，可以用来记录浏览量、点赞量等等。

**3. 排行榜**

Redis提供了列表和有序集合数据结构，合理地使用这些数据结构可以很方便地构建各种排行榜系统。

**4. 社交网络**

赞/踩、粉丝、共同好友/喜好、推送、下拉刷新。

**5. 消息队列**

Redis提供了发布订阅功能和阻塞队列的功能，可以满足一般消息队列功能。

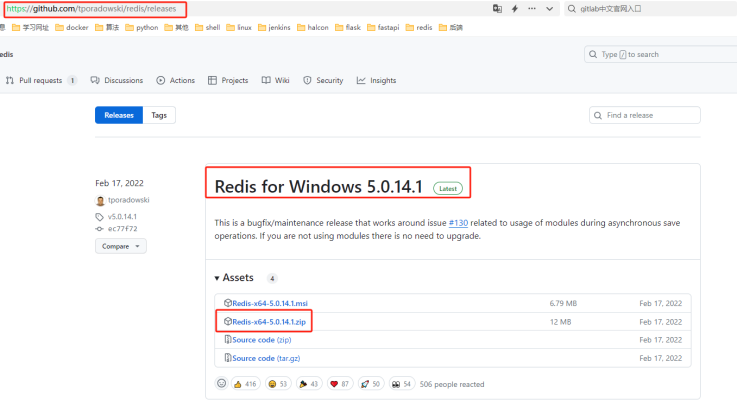
**6. 分布式锁**

分布式环境下，利用Redis实现分布式锁，也是Redis常见的应用。

# 三、redis的安装

## 1.Windows系统

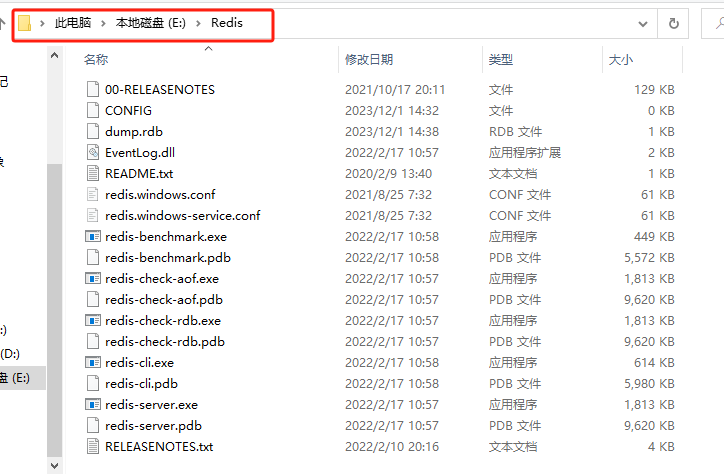
**下载：**https://github.com/tporadowski/redis/releases。



**启动：**

第一步：对下载的Redis-x64-xxx.zip进行解压，将文件夹重新命名为redis。

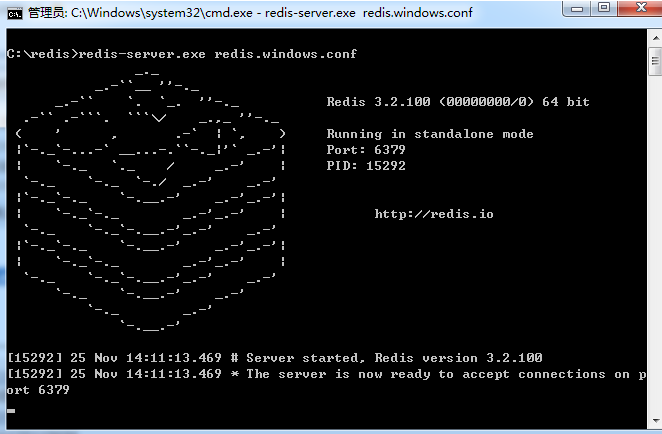
第二步：打开redis文件夹，文件内容如下：



第三步：打开一个命令行窗口，并切换到redis目录，运行如下命令：

redis-server.exe redis.windows.conf

**注意：**如果想方便的话，可以把 redis 的路径加到系统的环境变量里，这样就省得再输路径了，后面的那个 redis.windows.conf 可以省略，如果省略，会启用默认的。输入之后，会显示如下界面：

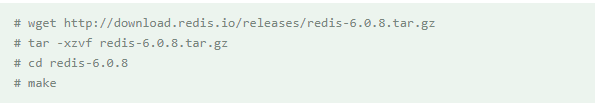


第四步：重新打开一个命令行窗口，原来的命令行窗口不要关闭，不然就无法访问服务端了，切换到 redis 目录，运行如下命令:

redis-cli.exe -h 127.0.0.1 -p 6379

## 2.Linux系统

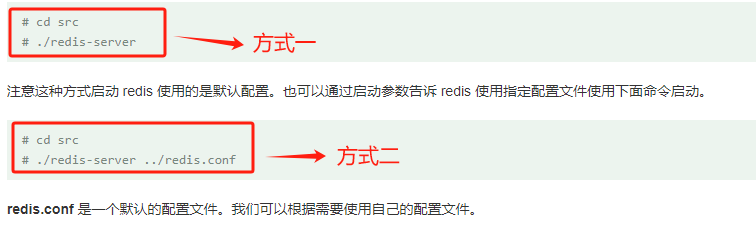
**下载：**



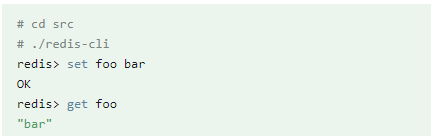
执行完 make 命令后，redis-6.0.8 的 src 目录下会出现编译后的 redis 服务程序 redis-server，还有用于测试的客户端程序 redis-cli。

**启动：**

第一步：启动 redis 服务



第二步：启动redis服务进程后，测试客户端redis-cli和 redis 服务交互。



# 四、redis的常用数据结构和命令

## 1.字符串-String

### 1.1 String的简介

String是Redis最基本的类型，一个key对应一个value。String是二进制安全的，意味着String可以包含任何数据，比如序列化对象或者一张图片。String最多可以放512M的数据，value 除了是字符串以外还可以是数字。

### 1.2 String的使用场景

计数器；

统计多单位的数量；

粉丝数；

对象缓存存储；

分布式锁。

### 1.3 String的常用命令

**SET key value**

设置指定 key 的值。

**GET key**

获取指定 key 的值。

**GETRANGE key start end**

返回 key 中字符串值的子字符。

**GETSET key value**

将给定key的值设为value，并返回 key 的旧值(old value)。

**MGET key1 [key2..]**

获取所有(一个或多个)给定 key 的值。

**SETEX key seconds value**

将值value关联到key，并将key的过期时间设为seconds(以秒为单位)。

**SETNX key value**

只有在key不存在时设置key的值。

**SETRANGE key offset value**

用 value 参数覆写给定key所储存的字符串值，从偏移量offset开始。

**STRLEN key**

返回key所储存的字符串值的长度。

**MSET key value [key value ...]**

同时设置一个或多个 key-value 对。

**MSETNX key value [key value ...]**

同时设置一个或多个key-value对，当且仅当所有给定key都不存在。

**PSETEX key milliseconds value**

这个命令和SETEX命令相似，但它以毫秒为单位设置key的生存时间，而不是像 SETEX命令那样，以秒为单位。

**INCR key**

将key中储存的数字值加1。

**INCRBY key increment**

将key所储存的值加上给定的增量值(increment) 。

**INCRBYFLOAT key increment**

将key所储存的值加上给定的浮点增量值(increment) 。

**DECR key**

将key中储存的数字值减一。

**DECRBY key decrement**

key所储存的值减去给定的减量值(decrement) 。

**APPEND key value**

如果key已经存在并且是一个字符串，APPEND命令将指定的value追加到该key原来值(value)的末尾。

## 2.列表-List

### 2.1 List的简介

List是简单的字符串列表，按照插入顺序排序，添加一个元素到列表的头部（左边）或者尾部（右边）。底层是一个双向链表，对两端的操作性能极高，通过索引操作中间的节点性能较差。一个List最多可以包含2的32次方-1个元素（每个列表超过40亿个元素）

### 2.2 List的使用场景

消息队列；

排行榜；

最新列表。

### 2.3 List的常用命令

**BLPOP key1 [key2 ] timeout**

移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

**BRPOP key1 [key2 ] timeout**

移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

**BRPOPLPUSH source destination timeout**

从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

**LINDEX key index**

通过索引获取列表中的元素

**LINSERT key BEFORE|AFTER pivot value**

在列表的元素前或者后插入元素

**LLEN key**

获取列表长度

**LPOP key**

移出并获取列表的第一个元素

**LPUSH key value1 [value2]**

将一个或多个值插入到列表头部

**LPUSHX key value**

将一个值插入到已存在的列表头部

**LRANGE key start stop**

获取列表指定范围内的元素

**LREM key count value**

移除列表元素

**LSET key index value**

通过索引设置列表元素的值

**LTRIM key start stop**

对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

**RPOP key**

移除列表的最后一个元素，返回值为移除的元素。

**RPOPLPUSH source destination**

移除列表的最后一个元素，并将该元素添加到另一个列表并返回

**RPUSH key value1 [value2]**

在列表中添加一个或多个值到列表尾部

**RPUSHX key value**

为已存在的列表添加值

## 3.哈希-Hash

### 3.1 Hash的简介

Hash是一个键值对的集合，Hash是一个String类型的Field（字段）和Value（值）的集合，Hash特别适用于存储对象。

### 3.2 Hash的使用场景

购物车；

存储对象。

### 3.3 Hash的常用命令

**HDEL key field1 [field2]**

删除一个或多个哈希表字段

**HEXISTS key field**

查看哈希表 key 中，指定的字段是否存在。

**HGET key field**

获取存储在哈希表中指定字段的值。

**HGETALL key**

获取在哈希表中指定 key 的所有字段和值

**HINCRBY key field increment**

为哈希表 key 中的指定字段的整数值加上增量 increment 。

**HINCRBYFLOAT key field increment**

为哈希表 key 中的指定字段的浮点数值加上增量 increment 。

**HKEYS key**

获取哈希表中的所有字段

**HLEN key**

获取哈希表中字段的数量

**HMGET key field1 [field2]**

获取所有给定字段的值

**HMSET key field1 value1 [field2 value2 ]**

同时将多个 field-value (域-值)对设置到哈希表 key 中。

**HSET key field value**

将哈希表 key 中的字段 field 的值设为 value 。

**HSETNX key field value**

只有在字段 field 不存在时，设置哈希表字段的值。

**HVALS key**

获取哈希表中所有值。

**HSCAN key cursor [MATCH pattern] [COUNT count]**

迭代哈希表中的键值对。

## 4.集合-Set

### 4.1 Set的简介

和List功能类似，但是Set是自动重排的，如果是存储一个列表数据，不希望出现重复数据时，Set是一个很好的选择。Set是String类型的无序集合，它底层是一个value为null的hash表，所以添加、删除、查找的时间复杂度都是O(1)。

### 4.2 Set的使用场景

黑白名单；

随机展示；

好友；

关注人；

粉丝；

感兴趣的人集合。

### 4.3 Set的常用命令

**SADD key member1 [member2]**

向集合添加一个或多个成员

**SCARD key**

获取集合的成员数

**SDIFF key1 [key2]**

返回第一个集合与其他集合之间的差异。

**SDIFFSTORE destination key1 [key2]**

返回给定所有集合的差集并存储在 destination 中

**SINTER key1 [key2]**

返回给定所有集合的交集

**SINTERSTORE destination key1 [key2]**

返回给定所有集合的交集并存储在 destination 中

**SISMEMBER key member**

判断 member 元素是否是集合 key 的成员

**SMEMBERS key**

返回集合中的所有成员

**SMOVE source destination member**

将 member 元素从 source 集合移动到 destination 集合

**SPOP key**

移除并返回集合中的一个随机元素

**SRANDMEMBER key [count]**

返回集合中一个或多个随机数

**SREM key member1 [member2]**

移除集合中一个或多个成员

**SUNION key1 [key2]**

返回所有给定集合的并集

**SUNIONSTORE destination key1 [key2]**

所有给定集合的并集存储在 destination 集合中

**SSCAN key cursor [MATCH pattern] [COUNT count]**

迭代集合中的元素

## 5.有序集合-ZSet

### 5.1 ZSet的简介

ZSet和Set很相似，是一个没有重复元素的String集合，不同之处是ZSet的每个元素都关联了一个分数(Score)，这个分数被用来按照从低分到高分的方式排序集合中的元素，集合的元素是唯一的，但是分数可以重复。

### 5.2 ZSet的使用场景

活跃天数；

打卡天数；

登录天数；

用户签到；

统计活跃用户；

统计用户是否在线；

实现布隆过滤器。

### 5.3 ZSet的常用命令

**ZADD key score1 member1 [score2 member2]**

向有序集合添加一个或多个成员，或者更新已存在成员的分数

**ZCARD key**

获取有序集合的成员数

**ZCOUNT key min max**

计算在有序集合中指定区间分数的成员数

**ZINCRBY key increment member**

有序集合中对指定成员的分数加上增量 increment

**ZINTERSTORE destination numkeys key [key ...]**

计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 destination 中

**ZLEXCOUNT key min max**

在有序集合中计算指定字典区间内成员数量

**ZRANGE key start stop [WITHSCORES]**

通过索引区间返回有序集合指定区间内的成员

**ZRANGEBYLEX key min max [LIMIT offset count]**

通过字典区间返回有序集合的成员

**ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT]**

通过分数返回有序集合指定区间内的成员

**ZRANK key member**

返回有序集合中指定成员的索引

**ZREM key member [member ...]**

移除有序集合中的一个或多个成员

**ZREMRANGEBYLEX key min max**

移除有序集合中给定的字典区间的所有成员

**ZREMRANGEBYRANK key start stop**

移除有序集合中给定的排名区间的所有成员

**ZREMRANGEBYSCORE key min max**

移除有序集合中给定的分数区间的所有成员

**ZREVRANGE key start stop [WITHSCORES]**

返回有序集中指定区间内的成员，通过索引，分数从高到低

**ZREVRANGEBYSCORE key max min [WITHSCORES]**

返回有序集中指定分数区间内的成员，分数从高到低排序

**ZREVRANK key member**

返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序

**ZSCORE key member**

返回有序集中，成员的分数值

**ZUNIONSTORE destination numkeys key [key ...]**

计算给定的一个或多个有序集的并集，并存储在新的 key 中

**ZSCAN key cursor [MATCH pattern] [COUNT count]**

迭代有序集合中的元素（包括元素成员和元素分值）

## 6.其他数据类型



# 五、NoSQL的四大分类

**NoSQL(NoSQL = Not Only SQL)：**

即“不仅仅是SQL”，泛指非关系型的数据库。随着互联网web2.0 网站的兴起，传统的关系数据库在应付特别是超大规模和高并发类型纯动态网站已经显得力不从心，暴露了很多难以克服的问题。

**结构化数据和非结构化数据：**

结构化数据指的是由二维表结构来逻辑表达和实现的数据，严格遵循数据格式与长度规范，也称作为行数据。

非结构化数据，指的是数据结构不规则或不完整，没有任何预定义的数据模型，不方便用二维逻辑表来表现的数据，例如办公文档（Word）、文本、图片、HTML、各类报表、视频音频等。

## 1.KV型NoSql（redis）

KV型NoSql，顾名思义就是以键值对形式存储的非关系型数据库，KV型NoSql最大的优点就是高性能，利用Redis自带的BenchMark做基准测试，TPS可达到10万的级别，性能非常强劲。

数据基于内存，读写效率高

KV型数据，时间复杂度为O(1)，查询速度快

## 2.列式NoSql（HBase）

列式NoSql，大数据时代最具代表性的技术之一，以HBase为代表。查询时只有指定的列会被读取，不会读取所有列；列数据被组织到一起，一次磁盘IO可以将一列数据一次性读取到内存中。

## 3.文档型NoSql（MongoDB）

文档型NoSql指的是将半结构化数据存储为文档的一种NoSql，文档型NoSql通常以JSON或者XML格式存储数据。关系型数据库是按部就班地每个字段一列存，在MongDB里面就是一个JSON字符串存储。

## 4.搜索型NoSql（ElasticSearch）

传统关系型数据库主要通过索引来达到快速查询的目的，但是在全文搜索的场景下，索引是无能为力的，like查询，一来无法满足所有模糊匹配需求，二来使用限制太大且使用不当容易造成慢查询，搜索型NoSql的诞生正是为了解决关系型数据库全文搜索能力较弱的问题，ElasticSearch是搜索型NoSql的代表产品。

# 六、redis的持久化机制

Redis的持久化机制有两种，分别是RDB和AOF。

由于redis的数据都放在内存中，如果没有配置持久化，Redis重启后数据就全部丢失了，于是重启Redis需要开启持久化功能。将数据保存到磁盘上，当Redis重启后，可以从磁盘中恢复数据。对于Redis而言，持久化机制是指把内存中的数据存为硬盘文件， 这样当Redis重启或服务器故障时能根据持久化后的硬盘文件恢复数 据。

## ****1.RDB持久化机制****

**含义：**

RDB，全称为Redis DataBase；RDB持久化是把当前进程数据生成快照保存到硬盘的过程，恢复时直接将快照读取到内存里。RDB⽂件是⼀个压缩的⼆进制⽂件，通过它可以还原某个时刻数据库的状态。由于RDB⽂件是保存在硬盘上的，所以即使Redis崩溃或者退出，只要RDB⽂件存在，就可以⽤它来恢复还原数据库的状态。

**文件位置：**

RDB保存的文件，可以在redis.conf中配置文件名称，默认为dump.rdb。(rdb文件的保存位置，也可以修改。默认在Redis启动时命令行所在的目录下。)

**RDB快照默认配置：**

1) save 3600 1：表示3600秒内（一小时）如果至少有1个key的值变化，则保存。

2) save 300 100：表示300秒内（五分钟）如果至少有100个 key 的值变化，则保存。

3) save 60 10000：表示60秒内如果至少有 10000个key的值变化，则保存。

**触发RDB持久化过程**：

分为手动触发和自动触发。手动触发有2种命令:

save：该命令会阻塞当前Redis服务器，执行save命令期间，Redis不能处理其他命令，直到RDB过程完成为止，不建议使用。

bgsave：执行该命令时，Redis会在后台异步进行快照操作，快照同时还可以响应客户端请求。

以下场景会自动触发RDB持久化：

1) 使用save相关配置，如“save m n”。表示m秒内数据集存在n次修改时，自动触发bgsave；

2) 如果从节点执行全量复制操作，主节点自动执行bgsave生成RDB文件并发送给从节点；

3) 执行debug reload命令重新加载Redis时，也会自动触发save操作；

4) 默认情况下执行shutdown命令时，如果没有开启AOF持久化功能则自动执行bgsave。

**恢复数据：**

只需要将rdb文件放在Redis的启动目录，Redis启动时会自动加载dump.rdb并恢复数据。

**优点：**

1) 适合大规模的数据恢复；

2) 对数据完整性和一致性要求不高更适合使用；

3) 节省磁盘空间；

4) 恢复速度快；

**缺点：**

1) 实时性低，RDB 是间隔一段时间进行持久化，没法做到实时持久化/秒级持久化。如果在这一间隔事件发生故障，数据会丢失。

2) 存在兼容问题，Redis演进过程存在多个格式的RDB版本，存在老版本Redis无法兼容新版本RDB的问题。

**注意：**

RDB机制无法保证数据的绝对安全是因为这个机制不是实时将数据序列化到磁盘上的，如果在还没有持久化的时候服务器宕机的话就有可能会丢失数据。

## ****2.AOF持久化机制****

**含义：**

AOF（append only file）,以日志的形式来记录每个写操作，将Redis执行过的所有写指令记录下来。AOF默认不开启可以在redis.conf中配置文件名称，默认为appendonly.aof。

**文件位置：**

AOF文件的保存路径，同RDB的路径一致，如果AOF和RDB同时启动，Redis默认读取AOF的数据。

**AOF启动/修复/恢复：**

设置Yes：修改默认的appendonly no，改为yes

**AOF同步频率设置：**

1) appendfsync always：始终同步，每次Redis的写入都会立刻记入日志，性能较差但数据完整性比较好。

2) appendfsync everysec：每秒同步，每秒记入日志一次，如果宕机，本秒的数据可能丢失。

3) appendfsync no redis：不主动进行同步，把同步时机交给操作系统。

**优点：**

1) 备份机制更稳健，丢失数据概率更低。

2) 可读的日志文本，通过操作AOF稳健，可以处理误操作。

**缺点：**

1) 比起RDB占用更多的磁盘空间。

2) AOF 文件比 RDB 文件大，恢复备份速度要慢。

3) 每次读写都同步的话，有一定的性能压力。

## ****3.注意事项****

如果同时幵启了AOF和RDB持久化，那么在Redis宕机重启之后，需要加载一个持久化文件，优先选择AOF文件。

如果先开启了RDB,再次开启AOF,如果RDB执行了持久化，那么RDB文件中的内容会被AOF覆盖掉。

# 七、redis事务

**含义：**

Redis 事务的本质是一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，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。

redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。

**Redis事务没有隔离级别的概念：**

批量操作在发送 EXEC 命令前被放入队列缓存，并不会被实际执行，也就不存在事务内的查询要看到事务里的更新，事务外查询不能看到。

**Redis不保证原子性：**

Redis中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。

**使用事务的步骤：**

1) 开启事务：multi

2) 输入要执行的命令：被放入到一个队列中

3) 执行事务：exec

4) 取消事务：discard

**ACID的特性：**

这四个特性分别是原子性(atomicity)、一致性(consistency)、隔离性(isolation)和持久性(durability)。

**原子性：**

一个事务(transaction)中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被恢复(Rollback)到事务开始前的状态，就像这个事务从来没有执行过一样。

**一致性：**

在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。

**隔离性：**

数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据 的不一致。事务隔离分为不同级别，包括读未提交(Readuncommitted)、读提交(read committed)、可重复读(repeatable read)和串行化(Serializable)。

**持久性：**

事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

# 八、redis的三种集群模式

## 1.主从模式

**含义：**

指的是将一台redis服务器的数据，复制到其它的redis服务器，前者称为主节点(master），后者称为从节点(slave)，数据的复制是单向的，只能由主节点到从节点。

**主从复制的作用：**

数据冗余：实现数据的热备份，是持久化之外的一种数据冗余方式。

故障恢复：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复；(实际是一种服务的冗余)

负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务。(即写redis数据时应用连接主节点，读redis数据时应用连接从节点)，分担服务器负载；

高可用：主从复制还可以是哨兵和集群能够实施的基础。

**主从复制的三个阶段**

连接建设阶段

数据同步阶段

命令传播阶段

## 2.哨兵模式

**含义：**

哨兵可以帮助我们解决主从架构中的单点故障问题，哨兵模式是一种特殊的模式，redis提供了哨兵的命令，哨兵是一个独立的进程，作为进程，可以独立运行。原理是哨兵通过发送命令，等待Redis服务器响应，从而监控运行的多个Redis实例。

**哨兵作用：**

集群监控：负责监控redis master和slave进程是否正常工作

消息通知：如果某个redis实例有故障，那么哨兵负责发送消息作为报警通知给管理员

故障转移：如果master node挂掉了，会自动转移到slave node上

配置中心：如果故障转移发生了，通知client客户端新的master地址

**工作原理：**

1) 监控阶段

sentinel(哨兵1)----->向master(主)和slave(从)发起info，拿到全信息。

sentinel(哨兵2)----->向master(主)发起info，就知道已经存在的sentinel(哨兵1)的信息，并且连接slave(从)。

sentinel(哨兵2)----->向sentinel(哨兵1)发起subscribe(订阅)。

2) 通知阶段

sentinel不断的向master和slave发起通知，收集信息。

3) 故障转移阶段

通知阶段sentinel发送的通知没得到master的回应，就会把master标记为SRI\_S\_DOWN,并且把master的状态发给各个sentinel，其他sentinel听到master挂了，说我不信，我也去看看，并把结果共享给各个sentinel，当有一半的sentinel都认为master挂了的时候，就会把master标记为SRI\_0\_DOWN。

**投票方式：**

最先接到哪个sentinel的竞选通知就会把票投给它

**故障转移：**

哨兵系统的主从节点，与普通的主从节点并没有什么区别，故障发现和转移是由哨兵来控制和完成的。

哨兵节点本质上是redis节点。

每个哨兵节点，只需要配置监控主节点，便可以自动发现其它的哨兵节点和从节点。

在哨兵节点启动和故障转移阶段，各个节点的配置文件会被重写。

**哨兵模式缺点：**

1) 当master挂掉时，sentinel会选举一个新的master，选举时无法访问Redis，会存在瞬断的情况。

2) 哨兵模式，对外只有master节点可以写，slave节点只能用于读，尽管redis单节点最多支持10W的QPS，但是在节日大促时，写数据的压力都在master上。

3) Redis单节点内存不能设置过大，如果数据过大在主从同步时会变得很慢，在节点启动时，时间会很长。

## 3.Cluster模式

**含义：**

Redis集群是一个由多个主从节点群组成的分布式服务集群，它具有复制，高可用和分片特性。Redis Cluster将所有数据划分为16384个slots(槽位)，每个节点负责其中一部分槽位。槽位的信息存于每个节点中，只有master节点会被分配槽位，slave节点不会分配槽位。

**优点：**

有多个master，可以减小访问瞬断问题的影响；

有多个master，可以提供更高的并发量；

可以分片存储，可以存储更多的数据；

**注意：**Redis集群最少需要三个节点；

# 九、redis缓存常见问题

## 1.Redis脑裂

**定义：**

Redis的集群脑裂是指因为网络问题，导致Redis Master节点跟redis slave节点和Sentinel 集群处于不同的网络分区，此时因为sentinel集群无法感知到master的存在，所以将slave节点提升为master节点。

此时存在两个不同的master节点，就像一个大脑分裂成两个，集群脑裂问题中，如果客户端还基于原来的master节点继续写入数据，那么新的Master节点将无法同步这些数据，当网络问题解决之后，sentinel集群将原先的Master节点降为slave节点，此时再从新的master中同步数据。

## 2.Redis缓存预热

**定义：**

新启动的系统中没有缓存数据，在缓存重建数据的过程中，系统性能和数据库负载都不太好，所以最好就是在系统上线之前就要把缓存的热点数据加载在缓存中，这种缓存预加载就是缓存预热。缓存预热解决数据库裸奔后的宕机问题。

**缓存冷启动**

缓存中没有数据，由于缓存冷启动没有数据，如果直接对外提供服务，当并发量上来时，MySQL就会挂掉了。

**解决方法：**

1) 提前给redis中灌入部分数据，再提供服务；

2) 如果数据量非常大，不能将所有数据都写入redis，因为数据量太大了会耗费大量时间，redis也容纳不下这么多数据；

3) 需要根据当天的具体访问情况，实时统计出访问频率较高的热数据，然后将这些热数据写入redis中；

4) 在热数据比较多的情况下可以用多个服务并行读取数据去写，并行的分布式的缓存预热。

## 3.缓存穿透

**定义：**

缓存穿透指的是缓存和数据库中都没有的数据，而用户不断发起请求，如发起id为“-1”的数据或者id特别大的数据。这个时候的用户请求很可能是攻击者，攻击会导致数据库压力过大。

**操作过程：**

某用户查询的数据在数据库中没有，自然缓存也不会有，这样就导致用户查询的时候，在缓存中查询不到数据，每次都要去数据库中再查询一遍，然后返回null（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查询数据库（缓存命中率问题）。

**解决方法：**

1) 对空值缓存：如果请求查询返回的数据为空（不管数据是否存在）仍然把这个空结果缓存，设置这个空结果的过期时间较短，最长不超过5分钟。

2) 布隆过滤器：如果想判断一个元素是不是在一个集合里，一般想到的是将集合中所有的元素保存起来，然后通过比较确定。

**布隆过滤器：**

布隆过滤器是一种数据结构，比较巧妙的概率型数据结构（probability data structure），特点是高效的插入和查询。(布隆过滤器是一种数据库结构，底层是bit数组)

## 4.缓存击穿

**定义：**

缓存击穿是指一个热点数据一直在不停的被大量请求访问，某一个瞬间这个key失效了，于是大量的请求就穿破缓存，直接请求数据库，造成数据库瞬间崩溃。

**解决方案：**

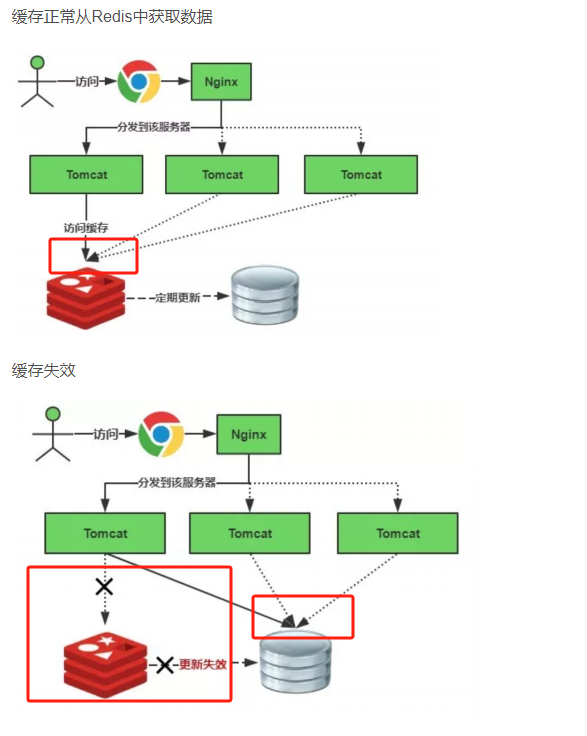
1) 互斥锁：在并发的多个请求中，只有第一个请求线程能拿到锁并执行数据库查询操作，其他线程拿不到锁就阻塞等待，等第一个线程将数据写入缓存后，其他线程直接查询缓存。

2) 热点数据不过期：直接将缓存设置为不过期，然后由定时任务去异步加载数据，更新缓存。

## 5.缓存雪崩

**定义：**

缓存雪崩是指Redis里同一时间大量缓存数据同时失效，这个时候依然有大量请求来访，结果请求都怼到数据库上，从而导致数据库雪崩。



**解决方法：**

1) 过期时间打散：给缓存过期时间加上一个随机时间，使得每个key的过期时间分布开来，不会集中在同一时刻失效。

2) 热点数据不过期：该方式和缓存击穿一样，也是要着重考虑刷新的时间和数据库异常如何处理的情况

3) 加互斥锁：该方式和缓存击穿一样，按key维度加锁，对于同一个key，只允许一个线程去计算其他线程原地阻塞等待第一个线程的计算结果，然后直接缓存即可。

# 十、redis常见面试问题

**1. Redis 使用的到底是多线程还是单线程？**

因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了。

**2. key的生存时间到了，Redis会立即删除吗？**

在Redis中，当key的生存时间到达时，它并不会立即被删除。这是因为Redis采用了两种主要的过期策略：惰性删除和定期删除。

惰性删除：这种策略的优点是对CPU消耗最小，只有在访问key的时候才会检查该key是否已过期，如果过期则删除。这种方式基本不会消耗CPU性能。

定期删除：这种策略默认每隔100ms就随机抽取一些key，检查其中是否有过期的key。需要明确的是，redis并不是每隔100ms就遍历所有的key，而是随机选择一部分进行检查。除此之外，Redis也会默认每秒进行10次过期扫描，采用一种贪心算法来选择并删除过期的key。这种方式通过限制删除操作执行的时长和频率来减少对CPU时间的影响，同时，定期删除过期键，也有效地减少了因为过期键而带来的内存浪费。

**3. 什么是Redis的淘汰机制？**

Redis的淘汰机制是当缓存数据增多，Redis被写满时，为了保证有足够的空间存储新的数据，需要选择某种策略将“不重要”的数据从Redis中清除。这个过程就是所谓的内存淘汰机制。

具体来说，Redis主要有四种内存淘汰策略：

volatile-lru：从已设置过期时间的key中挑选最近最少使用（LRU）的key进行删除。

allkeys-lru：从所有的key中挑选最近最少使用的key进行删除。

volatile-random：从已设置过期时间的key中任意选取一个key进行删除。

allkeys-random：从所有的key中任意选取一个key进行删除。

此外，当Redis运行在持久化模式下，它还会使用一种名为noeviction的策略，当写操作导致内存不足以容纳新写入数据时，会返回错误。这种策略可以作为内存淘汰的兜底方案。

**4. 怎么实现多个redis实例？**

一种常见的方法是在不同的文件夹中创建不同的Redis配置文件，然后为每个配置文件启动一个不同的Redis实例。在Linux上，你可以使用systemd服务来管理不同的Redis实例。

例如，以下是创建和启动两个Redis实例的步骤：

在文件系统上创建两个不同的文件夹，例如/redis1和/redis2。

在每个文件夹中创建一个新的Redis配置文件，例如redis1.conf和redis2.conf。

编辑每个配置文件，设置不同的端口号和pid文件路径等。

在每个文件夹中启动一个不同的Redis实例。例如，对于redis1.conf，你可以使用以下命令：redis-server /redis1/redis1.conf。

对于第二个实例，你可以使用类似的命令：redis-server /redis2/redis2.conf。

以上步骤是使用Linux系统进行操作，如果你使用的是其他操作系统，可以参考对应的官方文档或者教程进行操作。

请注意，创建多个Redis实例时，需要确保每个实例都有唯一的端口号和pid文件路径等配置。同时，如果你需要使用Redis集群模式，还需要为每个实例配置不同的集群节点信息。

# 十一、redis优化

## **1.数据存储优化**

### 1) 控制元素的大小

在保证key在简单、清晰的前提下，尽可能把key定义得短一些来控制key的长度，如 uerOreder:0001 缩写为uo:0001。

建议：做好key规范管理的文档，定义好每个key缩写的含义。

如按以上规范来做当redis中有大量的key时也会节约redis大量的内存空间，使其性能更高。

### 2) 控制key的长度

在决定使用redis建议强制规定元素的大小，推荐规范如下：

1. String类型数据的值控制在10K以下；

2. List/Hash/Set/ZSet数据类型的元素要控制在1W以内。

以上的是杜绝bigkey有效措施，在bigkey对redis的性能影响是最为致命的。

### **3)** 存储合适数据类型

除非业务的强要求，建议选择合适的数据类型来优化内存，具体如下：

1. String、Set：尽可能存储 int 类型数据；

2. Hash、ZSet：存储的元素数量控制在转换阈值之下，以压缩列表存储，节约内存。

### **4)** 设置过期key

强制要求所有的key必须设置过期时间，以优化redis内存。

### 5) 冷热分离

热key需要单独存放并分配合理的资源，防止大流量下直接冲垮整个缓存系统。

### 6) 数据压缩

可以采用snappy、gzip 等压缩算法来先将数据压缩后再存入缓存中，来节约redis的内存空间，但这种方法会使客户端在读取时还需要解压缩，在这期间会消耗更多CPU资源，你需要根据实际情况进行权衡。

## **2.内存淘汰优化**

杜绝使用默认的内存淘汰策略，避免在业务扩展下Redis的内存持续膨胀，需要根据你的业务设置对应内存淘汰策略。

## **3.过期策略优化**

由于redis采用的是定期删除+懒加载删除策略，且这个过程在redis 6.0之前是在主线程上执行的，建议所有key的过期时间用随机数打散，杜绝大批量的数据同时过期，拉胯redis的性能和造成缓存雪崩。

## **4.持久化优化**

除非业务的强要求，否则不要开启AOF，避免写磁盘拖垮redis的性能。如果有业务要求开启AOF，建议配置为 appendfsync everysec 把数据持久化的刷盘操作，放到后台线程中去执行，尽量降低 Redis 写磁盘对性能的影响。

## **5.部署优化**

### 1) 物理机部署

Redis在做数据持久化时，采用创建子进程的方式进行(会调用操作系统的 fork 系统)，而 虚拟机环境执行 fork 的耗时，要比物理机慢得多。

### 2) 内网部署

Redis集群有做副本数据同步，为了解决网络问题建议整个集群都部署在同一个局域网中。

## **6.集群优化**

### 1) 只使用db0

具体原因如下：

1.在一个连接上操作多个db数据时，每次都会先执行查询，有额外开销；

2.建议不同业务数据尽量存储到不同的实例中而不是存放在不同db中；

3.redis cluster只支持db0。

### 2) 架构优化

读写分离能最大限度提高redis的性能，其中主库负责数据写入，从库负责数据读取；分片集群是解决超大量数据导致性能瓶颈方案，如rediscluster。以上是在大流量下提高redis性能在架构上的优化。

### 3) 实例内存优化

Redis集群有做副本数据同步，如果redis的内存过大，那么在做持久化或数据同步时，磁盘IO和网络IO会拉垮redis性能，导致卡顿。建议redis每一个实例都不要超过8G，要从业务上解耦，不同的业务用不同的redis实例，杜绝所有业务都使用同一个redis实例。

## **7.其他优化**

### 1) lazy-free机制

在redis4.0+中支持，开启lazy-free机制后，由主线程删除bigkey，而较耗时的内存释放会在后台线程中执行，不会影响到主线程。

### 2) 批量命令

尽量减少客户端和服务端之间来回的io次数，建议string/hash使用mget/mset代替get/set，hmget/hmset代替hget/hset;其他数据类型使用pipeline。

### 3) 避免复杂度过高命令

redis是单线程模型处理请求，在执行复杂度过高的命令(消耗更多cpu资源)时后面的请求会排队导致延迟，如 SORT、SINTER、SINTERSTORE、ZUNIONSTORE、ZINTERSTORE 等聚合类命令。

### 4) 注意容器类型数据

容器类型数据查询：list、hash、set、zset类型数据在查询时要先确认元素的数量，元素非常多时需要分批查询(LRANGE/HASCAN/SSCAN/ZSCAN)，否则大量数据会导致延迟（如网络传输问题等）。

容器类型数据删除：建议分批删除，如list执行多次lpop/rpop；hash/set/zset先执行hscan/sscan/scan查询元素，再执行hdel/srem/zrem,不关注元素的数量直接del会有大量内存释放，拖垮主线程性能。

# 十二、python与redis交互

## 1.安装redis模块

**pip install redis**

## 2. Redis连接

Redis提供两个类 Redis和 StrictRedis用于实现 Redis的命令。

1) StrictRedis用于实现大部分官方的命令，并使用官方的语法和命令，

2) Redis是 StrictRedis的子类，用于向后兼容旧版本的 redis-py库

**方式1：单机连接**

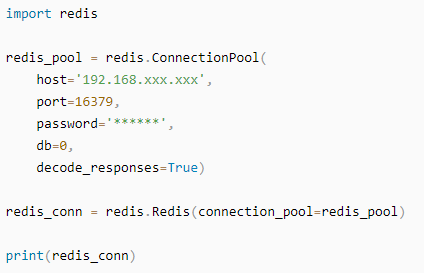


**注意：**

redis 取出的结果默认都是字节（bytes）类型，我们可以设定 decode\_responses=True 改成字符串。

**方式2：连接池**

redis-py 使用 connection pool 来管理对一个 redis server 的所有连接，避免每次建立、释放连接的开销。默认，每个Redis实例都会维护一个自己的连接池。可以直接建立一个连接池，然后作为参数 Redis，这样就可以实现多个 Redis 实例共享一个连接池。



# 十三、参考网站

第一个：

[https://blog.csdn.net/qq\_31432773/article/details/129251561?utm\_medium=distribute.pc\_relevant.none-task-blog-2~default~baidujs\_baidulandingword~default-0-129251561-blog-131904046.235^v39^pc\_relevant\_3m\_sort\_dl\_base3&spm=1001.2101.3001.4242.1&utm\_relevant\_index=1](https://blog.csdn.net/qq_31432773/article/details/129251561?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-129251561-blog-131904046.235%5ev39%5epc_relevant_3m_sort_dl_base3&spm=1001.2101.3001.4242.1&utm_relevant_index=1)

第二个：

[https://blog.csdn.net/weixin\_45680962/article/details/108218969?utm\_medium=distribute.pc\_relevant.none-task-blog-2~default~baidujs\_baidulandingword~default-5-108218969-blog-122215202.235^v39^pc\_relevant\_3m\_sort\_dl\_base3&spm=1001.2101.3001.4242.4&utm\_relevant\_index=8](https://blog.csdn.net/weixin_45680962/article/details/108218969?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-5-108218969-blog-122215202.235%5ev39%5epc_relevant_3m_sort_dl_base3&spm=1001.2101.3001.4242.4&utm_relevant_index=8)

第三个：

<https://blog.csdn.net/qq_43545600/article/details/126877695?spm=1001.2101.3001.6650.14&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-14-126877695-blog-108218969.235%5Ev39%5Epc_relevant_3m_sort_dl_base3&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-14-126877695-blog-108218969.235%5Ev39%5Epc_relevant_3m_sort_dl_base3&utm_relevant_index=24>