# 背景

由于数据库的缓存一般是针对查询的内容，而且粒度也比较小，一般只有表中的数据没有发生变动的时候，数据库的缓存才会产生作用。

但这并不能减少业务逻辑对数据库的增删改操作的 IO 压力，因此缓存技术应运而生，该技术实现了对热点数据的高速缓存，可以大大缓解后端数据库的压力。

# 概述

Redis是一个远程内存数据库（非关系型数据库），性能强劲，具有复制特性以及解决问题而生的独一无二的数据模型。

它可以存储键值对与5种不同类型的值之间的映射，可以将存储在内存的键值对数据持久化到硬盘，可以使用复制特性来扩展读性能。

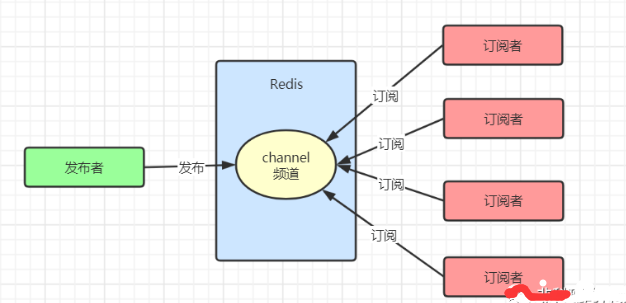
Redis还可以使用客户端分片来扩展写性能，内置了复制（replication），LUA 脚本（Lua scripting），LRU驱动事件（LRU eviction），事务（transactions） 和不同级别的磁盘持久化（persistence）。

并通过Redis哨兵（Sentinel）和自动分区（Cluster）提供高可用性（High Availability）。

## 原理

Redis发布与发布功能（Pub/Sub）是基于事件座位基本的通信机制，是目前应用比较普遍的通信模型，它的目的主要是解除消息的发布者与订阅者之间的耦合关系。

Redis作为消息发布和订阅之间的服务器，起到桥梁的作用，在Redis里面有一个channel的概念，也就是频道，发布者通过指定发布到某个频道，然后只要有订阅者订阅了该频道，该消息就会发送给订阅者，原理图如下所示：



Redis同时也可以使用list类型实现消息队列。

Redis的发布与订阅的功能应用还是比较广泛的，它的应用场景有很多。比如：最常见的就是实现实时聊天的功能，还是有就是博客的粉丝文章的推送，当博主推送原创文章的时候，就会将文章实时推送给博主的粉丝。

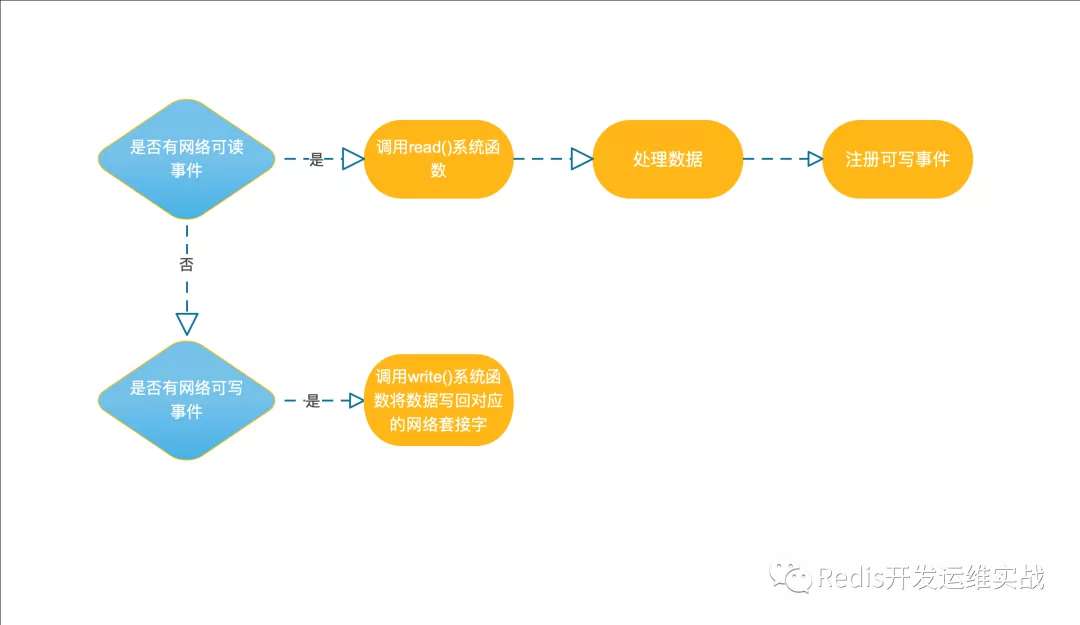
## 单进程模型

首先，我们明确一点，Redis6之前的Redis4，Redis5并不是单线程程序。通常我们说的Redis的单线程，是指Redis接受链接，接收数据并解析协议，发送结果等命令的执行，都是在主线程中执行的。

Redis之前之所以将这些都放在主线程中执行，主要有以下几方面的原因：

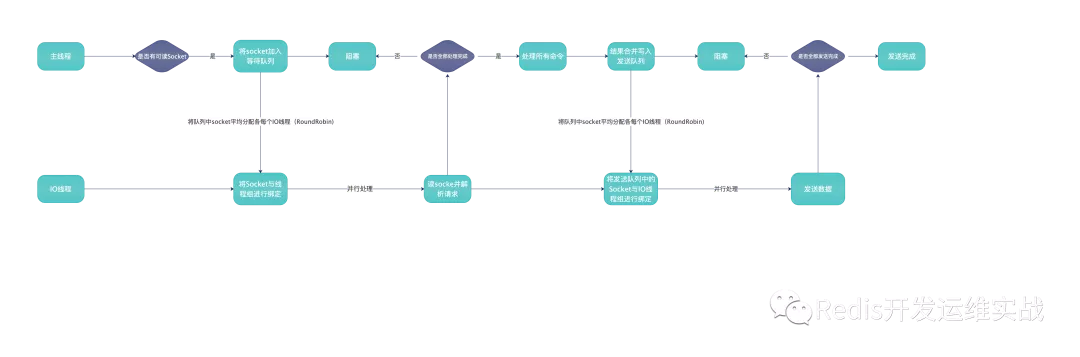
1. Redis的主要瓶颈不在cpu，而在内存和网络IO
2. 使用单线程设计，可以简化数据库结构的设计
3. 可以减少多线程锁带来的性能损耗

既然Redis的主要瓶颈不在CPU，为什么又要引入IO多线程？Redis的整体处理流程如下图：



null结合上图可知，当socket中有数据时，Redis会通过系统调用将数据从内核态拷贝到用户态，供Redis 解析用。这个拷贝过程是阻塞的，术语称作“同步阻塞IO”，数据量越大拷贝的延迟越高，解析协议时间消耗也越大，糟糕的是这些操作都是在主线程中处理的，特别是链接数特别多的情况下，这种情况更加明显。基于以上原因，Redis作者提出了Thread/IO线程，既将接收与发送数据来使用多线程并行处理，从而降低主线程的等待时间。

Thread/IO整体实现思路：



1、创建一组大小为io线程个数的等待队列，用来存储客户端的网络套接字。

2、分均分配客户端网络套接字到等待队列中

3、等待线程组接收解协议完毕或者发送数据完毕

4、执行后续操作，然后跳转到第2步继续执行

**选用策略：**

1、因地制宜，优先选择时间复杂度为O(1)的I/O多路复用函数作为底层实现。

2、由于Select要遍历每一个IO，所以其时间复杂度为O(n)，通常被作为保底方案。

3、基于React设计模式监听I/O事件。

## 速度快

Redis 的效率很高，官方给出的数据是 100000+QPS，这是因为：

1、Redis **完全基于内存**，绝大部分请求是纯粹的内存操作，执行效率高。

2、Redis 使用**单进程单线程**模型的（K，V）数据库，将数据存储在内存中，存取均不会受到硬盘IO的限制，因此其执行速度极快。

另外单线程也能处理高并发请求，还可以避免频繁上下文切换和锁的竞争，如果想要多核运行也可以启动多个实例。

3、数据结构简单，对数据操作也简单，Redis不使用表，不会强制用户对各个关系进行关联，不会有复杂的关系限制，其存储结构就是键值对，类似于 HashMap，HashMap最大的优点就是存取的时间复杂度为O(1)。

4、Redis使用多路I/O复用模型，为非阻塞IO。

注：Redis采用的I/O多路复用函数：epoll/kqueue/evport/select。

选用策略：

因地制宜，优先选择时间复杂度为O(1)的I/O多路复用函数作为底层实现。

由于Select要遍历每一个IO，所以其时间复杂度为O(n)，通常被作为保底方案。

基于React设计模式监听I/O事件。

## Redis VS Memcache



Memcache的代码层类似Hash，特点如下：

支持简单数据类型

不支持数据持久化存储

不支持主从

不支持分片

Redis 特点如下：

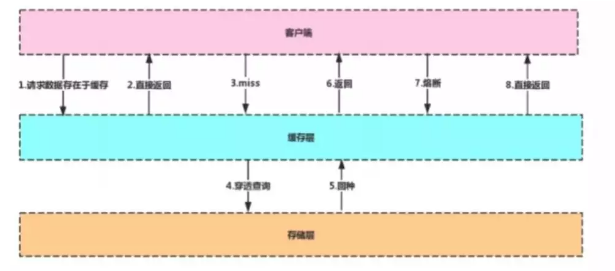
数据类型丰富

支持数据磁盘持久化存储

支持主从

支持分片

# 架构



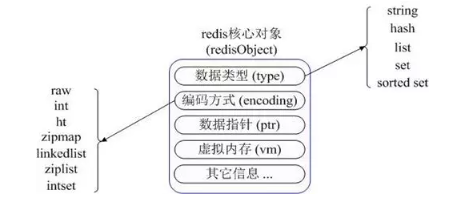
客户端在对数据库发起请求时，先到缓存层查看是否有所需的数据，如果缓存层存有客户端所需的数据，则直接从缓存层返回，否则进行穿透查询，对数据库进行查询。

如果在数据库中查询到该数据，则将该数据回写到缓存层，以便下次客户端再次查询能够直接从缓存层获取数据。

# 数据类型

Redis对应的数据类型，即K-V键值对中Value的数据类型。

Redis的数据模型如下图：



## String

在Redis中String是可以修改的，称为动态字符串(Simple Dynamic String简称SDS)。说是字符串但它的内部结构更像是一个ArrayList，内部维护着一个字节数组，并且在其内部预分配了一定的空间，以减少内存的频繁分配。

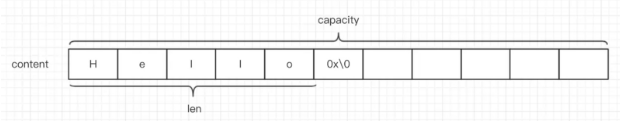
最基本的数据类型，其值最大可存储512M，二进制安全（Redis的String可以包含任何二进制数据，包含jpg对象等）。

Redis的内存分配机制是这样：

1、当字符串的长度小于1MB时，每次扩容都是加倍现有的空间。

2、如果字符串长度超过1MB时，每次扩容时只会扩展1MB的空间。

这样既保证了内存空间够用，还不至于造成内存的浪费，字符串最大长度为 512MB。



上图就是字符串的基本结构，其中content里面保存的是字符串内容，0x\0作为结束字符不会被计算len中。

分析一下字符串的数据结构：

struct SDS{

T capacity; //数组容量

T len; //实际长度

byte flages; //标志位,低三位表示类型

byte[] content; //数组内容

}

capacity和len两个属性都是泛型，为什么不直接用int类型？因为Redis内部有很多优化方案，为更合理的使用内存，不同长度的字符串采用不同的数据类型表示，且在创建字符串的时候len会和capacity一样大，不产生冗余的空间，所以String值可以是字符串、数字（整数、浮点数) 或者二进制。

### 应用场景

存储key-value键值对。

注：如果重复写入key相同的键值对，后写入的会将之前写入的覆盖。

### 常用命令

字符串（String）常用的命令：

set [key] [value] 给指定key设置值（set 可覆盖老的值）

get [key] 获取指定key 的值

del [key] 删除指定key

exists [key] 判断是否存在指定key

mset [key1] [value1] [key2] [value2] ...... 批量存键值对

mget [key1] [key2] ...... 批量取key

expire [key] [time] 给指定key 设置过期时间 单位秒

setex [key] [time] [value] 等价于 set + expire 命令组合

setnx [key] [value] 如果key不存在则set 创建，否则返回0

incr [key] 如果value为整数 可用 incr命令每次自增1

incrby [key] [number] 使用incrby命令对整数值 进行增加 number

## Hash

String元素组成的字典，适用于存储对象。

Redis中的Hash和Java的HashMap更加相似，都是数组+链表的结构，当发生hash碰撞时将会把元素追加到链表上，值得注意的是在Redis的Hash中value只能是字符串.

hset books java "Effective java" (integer) 1

hset books golang "concurrency in go" (integer) 1

hget books java "Effective java"

hset user age 17 (integer) 1

hincrby user age 1

#单个key可以进行计数和incr命令基本一致 (integer) 18

Hash和String都可以用来存储用户信息 ，但不同的是Hash可以对用户信息的每个字段单独存储；String存的是用户全部信息经过序列化后的字符串，如果想要修改某个用户字段必须将用户信息字符串全部查询出来，解析成相应的用户信息对象，修改完后在序列化成字符串存入。而hash可以只对某个字段修改，从而节约网络流量，不过hash内存占用要大于String，这是hash的缺点。

### 应用场景

购物车：hset [key] [field] [value]命令，可以实现以用户Id，商品Id为field，商品数量为value，恰好构成了购物车的3个要素。

存储对象：hash类型的(key, field, value)的结构与对象的(对象id, 属性, 值)的结构相似，也可以用来存储对象。

### 常用命令

hash常用的操作命令：

hset [key] [field] [value] 新建字段信息

hget [key] [field] 获取字段信息

hdel [key] [field] 删除字段

hlen [key] 保存的字段个数

hgetall [key] 获取指定key 字典里的所有字段和值（字段信息过多,会导致慢查询慎用：亲身经历 曾经用过这个这个指令导致线上服务故障）

hmset [key] [field1] [value1] [field2] [value2] ...... 批量创建

hincr [key] [field] 对字段值自增

hincrby [key] [field] [number] 对字段值增加number

## List

列表，按照String元素插入顺序排序。其顺序为后进先出。由于其具有栈的特性，所以可以实现如“最新消息排行榜”这类的功能。

Redis中的list和Java中的LinkedList很像，底层都是一种链表结构，list的插入和删除操作非常快，时间复杂度为0(1)，不像数组结构插入、删除操作需要移动数据。

但是redis中的list底层可不是一个双向链表那么简单。

当数据量较少的时候它的底层存储结构为一块连续内存，称之为ziplist(压缩列表)，它将所有的元素紧挨着一起存储，分配的是一块连续的内存；当数据量较多的时候将会变成quicklist(快速链表)结构。

可单纯的链表也是有缺陷的，链表的前后指针 prev和next会占用较多的内存，会比较浪费空间，而且会加重内存的碎片化。在redis 3.2之后就都改用ziplist+链表的混合结构，称之为quicklist(快速链表)。

ziplist(压缩列表)

先看一下ziplist的数据结构，

struct ziplist<T>{

int32 zlbytes; //压缩列表占用字节数

int32 zltail\_offset;

//最后一个元素距离起始位置的偏移量,用于快速定位到最后一个节点

int16 zllength; //元素个数

T[] entries; //元素内容

int8 zlend; //结束位 0xFF

}

int32 zlbytes： 压缩列表占用字节数

int32 zltail\_offset：

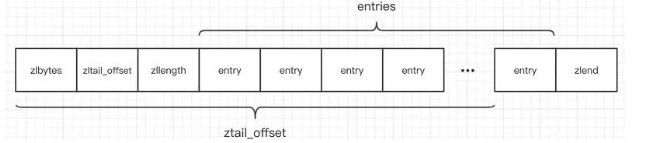
最后一个元素距离起始位置的偏移量,用于快速定位到最后一个节点

`int16 zllength`：元素个数

`T[] entries`：元素内容

`int8 zlend`：结束位 0xFF

压缩列表为了支持双向遍历，所以才会有 ztail\_offset 这个字段，用来快速定位到最后一个元素，然后倒着遍历



entry的数据结构：

struct entry{

int<var> prevlen; //前一个 entry 的长度

int<var> encoding; //元素类型编码

optional byte[] content; //元素内容

}

entry它的 prevlen 字段表示前一个 entry 的字节长度，当压缩列表倒着遍历时，需要通过这个字段来快速定位到下一个元素的位置。

### 应用场景

由于list它是一个按照插入顺序排序的列表，所以应用场景相对还较多的，例如：

消息队列：lpop和rpush（或者反过来，lpush和rpop）能实现队列的功能

朋友圈的点赞列表、评论列表、排行榜：lpush命令和lrange命令能实现最新列表的功能，每次通过lpush命令往列表里插入新的元素，然后通过lrange命令读取最新的元素列表。

### 常用命令

list操作的常用命名：

rpush [key] [value1] [value2] ...... 链表右侧插入

rpop [key] 移除右侧列表头元素，并返回该元素

lpop [key] 移除左侧列表头元素，并返回该元素

llen [key] 返回该列表的元素个数

lrem [key] [count] [value] 删除列表中与value相等的元素，count是删除的个数。count>0表示从左侧开始查找，删除count个元素，count<0表示从右侧开始查找，删除count个相同元素，count=0表示删除全部相同的元素

注：index代表元素下标，index可以为负数，index=表示倒数第一个元素，同理index=-2表示倒数第二个元素。

lindex [key] [index] 获取list指定下标的元素（需要遍历，时间复杂度为O(n)）

lrange [key] [start\_index] [end\_index] 获取list区间内的所有元素 （时间复杂度为O(n)）

ltrim [key] [start\_index] [end\_index] 保留区间内的元素，其他元素删除（时间复杂度为O(n)）

## Set

String元素组成的无序集合，通过哈希表实现（增删改查时间复杂度为 O(1)），不允许重复。

另外，当我们使用Smembers遍历Set中的元素时，其顺序也是不确定的，是通过 Hash 运算过后的结果。

Redis还对集合提供了求交集、并集、差集等操作，可以实现如同共同关注，共同好友等功能。

Redis中的set和Java中的HashSet有些类似，它内部的键值对是无序的、唯一的。它的内部实现相当于一个特殊的字典，字典中所有的value都是一个值 NULL。当集合中最后一个元素被移除之后，数据结构被自动删除，内存被回收。

### 应用场景：

好友、关注、粉丝、感兴趣的人集合：

1、sinter命令可以获得A和B两个用户的共同好友；

2、sismember命令可以判断A是否是B的好友；

3、scard命令可以获取好友数量；

4、关注时，smove命令可以将B从A的粉丝集合转移到A的好友集合

首页展示随机：美团首页有很多推荐商家，但是并不能全部展示，set类型适合存放所有需要展示的内容，而srandmember命令则可以从中随机获取几个。

存储某活动中中奖的用户ID ，因为有去重功能，可以保证同一个用户不会中奖两次。

### 常用命令

sadd [key] [value] 向指定key的set中添加元素

smembers [key] 获取指定key集合中的所有元素

sismember [key] [value] 判断集合中是否存在某个value

scard [key] 获取集合的长度

spop [key] 弹出一个元素

srem [key] [value] 删除指定元素

## Sorted Set

通过分数来为集合中的成员进行从小到大的排序。

zset也叫SortedSet一方面它是个set ，保证了内部value的唯一性，另方面它可以给每个value赋予一个score，代表这个value的排序权重。它的内部实现用的是一种叫作“跳跃列表”的数据结构。

### 应用场景

zset可以用做排行榜，但是和list不同的是zset它能够实现动态的排序，例如：可以用来存储粉丝列表，value值是粉丝的用户ID，score是关注时间，我们可以对粉丝列表按关注时间进行排序。

zset还可以用来存储学生的成绩，value值是学生的ID，score是他的考试成绩。我们对成绩按分数进行排序就可以得到他的名次。

### 常用命令

zadd [key] [score] [value] 向指定key的集合中增加元素

zrange [key] [start\_index] [end\_index] 获取下标范围内的元素列表，按score 排序输出

zrevrange [key] [start\_index] [end\_index] 获取范围内的元素列表 ，按score排序 逆序输出

zcard [key] 获取集合列表的元素个数

zrank [key] [value] 获取元素再集合中的排名

zrangebyscore [key] [score1] [score2] 输出score范围内的元素列表

zrem [key] [value] 删除元素

zscore [key] [value] 获取元素的score

更高级的Redis类型

用于计数的 HyperLogLog、用于支持存储地理位置信息的 Geo。

# 操作

String：set、get、mset、mget

Hash：hset、hget、hmset、hmget

## 数据操作

## 消息操作

### 发布消息

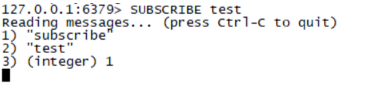
Redis中发布消息的命令是publish，具体使用如下所示：



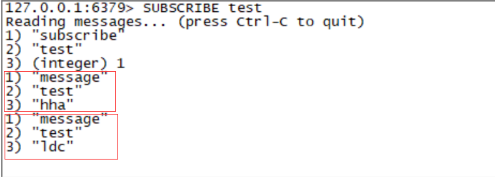
PUBLISH test "haha"：test表示频道的名称，haha表示发布的内容，这样就完成了一个一个消息的发布，后面的返回（integer）0表示0人订阅。

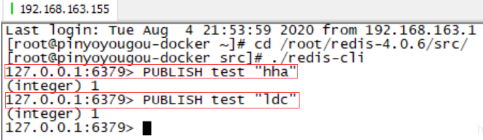
### 订阅频道

与此同时再启动一个窗口，这个窗口作为订阅者，订阅者的命令subscribe，使用SUBSCRIBE test就表示订阅了test这个频道



订阅后返回的结果中由三条信息，第一个表示类型、第二个表示订阅的频道，第三个表示订阅的数量。接着在第一个窗口进行发布消息：





可以看到发布者发布的消息，订阅者都会实时的接收到，并发订阅者收到的信息中也会出现三条信息，分别表示：返回值的类型、频道名称、消息内容。

### 取消订阅

若是想取消之前的订阅可以使用unsubscribe命令，格式为：

unsubscribe 频道名称

// 取消之前订阅的test频道

unsubscribe test

输入命令后，返回以下结果：

[root@pinyoyougou-docker src]# ./redis-cli

127.0.0.1:6379> UNSUBSCRIBE test

1) "unsubscribe"

2) "test"

3) (integer) 0

它分别表示：返回值的类型、频道的名称、该频道订阅的数量。

### 按模式订阅

除了直接以特定的名城进行订阅，还可以按照模式进行订阅，模式的方式进行订阅可以一次订阅多个频道，按照模式进行订阅的命令为psubscribe，具体格式如下：

psubscribe 模式

// 表示订阅名称以ldc开头的频道

psubscribe ldc\*

输入上面的命令后，返回如下结果：

127.0.0.1:6379> PSUBSCRIBE ldc\*

Reading messages... (press Ctrl-C to quit)

1) "psubscribe"

2) "ldc\*"

3) (integer) 1

这个也是非常简单，分别表示：返回的类型（表示按模式订阅类型）、订阅的模式、订阅数。

### 取消按模式订阅

假如你想取消之前的按模式订阅，可以使用punsubscribe来取消，具体格式：

punsubscribe 模式

// 取消频道名称按照ldc开头的频道

punsubscribe ldc\*

他的返回值，如下所示：

127.0.0.1:6379> PUNSUBSCRIBE ldc\*

1) "punsubscribe"

2) "ldc\*"

3) (integer) 0

这个就不多说了，表示的意思和上面的一样，可以看到上面的命令都是有规律的订阅SUBSCRIBE，取消就是UNSUBSCRIBE，前面加前缀UN，按模式订阅也是。

### 查看订阅消息

（1）你想查看某一个模式下订阅数是大于零的频道，可以使用如下格式的命令进行操作：

pubsub channels 模式

// 查看频道名称以ldc模式开头的订阅数大于零的频道

pubsub channels ldc\*

（2）假如你想查看某一个频道的订阅数，可以使用如下命令：

pubsub numsub 频道名称

（3）查看按照模式的订阅数，可以使用如下命令进行操作：

pubsub numpat

# 内存

## 内存大小设置

Redis是基于内存的key-value数据库，因为系统的内存大小有限，所以我们在使用Redis的时候可以配置Redis能使用的最大的内存大小。

### 通过配置文件配置

通过在Redis安装目录下面的redis.conf配置文件中添加以下配置设置内存大小：

//设置Redis最大占用内存大小为100M

maxmemory 100mb

redis的配置文件不一定使用的是安装目录下面的redis.conf文件，启动redis服务的时候是可以传一个参数指定redis的配置文件的。

### 通过命令修改

Redis支持运行时通过命令动态修改内存大小

//设置Redis最大占用内存大小为100M

127.0.0.1:6379> config set maxmemory 100mb

//获取设置的Redis能使用的最大内存大小

127.0.0.1:6379> config get maxmemory

如果不设置最大内存大小或者设置最大内存大小为0，在64位操作系统下不限制内存大小，在32位操作系统下最多使用3GB内存。

## 内存淘汰

既然可以设置Redis最大占用内存大小，那么配置的内存就有用完的时候。那在内存用完的时候，还继续往Redis里面添加数据不就没内存可用了吗？

实际上Redis定义了几种策略用来处理这种情况：

noeviction(默认策略) ：对于写请求不再提供服务，直接返回错误（DEL请求和部分特殊请求除外）

allkeys-lru：从所有key中使用LRU算法进行淘汰

volatile-lru：从设置了过期时间的key中使用LRU算法进行淘汰

allkeys-random：从所有key中随机淘汰数据

volatile-random：从设置了过期时间的key中随机淘汰

volatile-ttl：在设置了过期时间的key中，根据key的过期时间进行淘汰，越早过期的越优先被淘汰

当使用 volatile-lru、volatile-random、volatile-ttl这三种策略时，如果没有key可以被淘汰，则和noeviction一样返回错误。

Redis的数据淘汰策略如下图：



Redis的数据淘汰内部实现如下图：



### 获取内存淘汰策略

获取当前内存淘汰策略：

127.0.0.1:6379> config get maxmemory-policy

通过配置文件设置淘汰策略（修改redis.conf文件）：

maxmemory-policy allkeys-lru

通过命令修改淘汰策略：

127.0.0.1:6379> config set maxmemory-policy allkeys-lru

### LRU算法

LRU(Least Recently Used)，即最近最少使用，是一种**缓存置换算法**。在使用内存作为缓存的时候，缓存的大小一般是固定的。当缓存被占满，这个时候继续往缓存里面添加数据，就需要淘汰一部分老的数据，释放内存空间用来存储新的数据。这个时候就可以使用LRU算法了。

其核心思想是：如果一个数据在最近一段时间没有被用到，那么将来被使用到的可能性也很小，所以就可以被淘汰掉。

**LRU在Redis中的实现**

1、近似LRU算法

Redis使用的是近似LRU算法，它跟常规的LRU算法还不太一样。近似LRU算法通过**随机采样法淘汰数据**，每次随机出5（默认）个key，从里面淘汰掉最近最少使用的key（MySQL是在5/7位置）。

可以通过maxmemory-samples参数修改采样数量：例：maxmemory-samples 10 maxmenory-samples配置的越大，淘汰的结果越接近于严格的LRU算法。

Redis为了实现近似LRU算法，给每个key额外增加了一个24bit的字段，用来存储该key最后一次被访问的时间。

2、Redis3.0对近似LRU的优化

Redis3.0对近似LRU算法进行了一些优化。新算法会维护一个**候选池**（大小为16），池中的数据根据访问时间进行排序，第一次随机选取的key都会放入池中，随后每次随机选取的key只有在访问时间小于池中最小的时间才会放入池中，直到候选池被放满。当放满后，如果有新的key需要放入，则将池中最后访问时间最大（最近被访问）的移除。

当需要淘汰的时候，则直接从池中选取最近访问时间最小（最久没被访问）的key淘汰掉就行。

### LFU算法

LFU算法是Redis4.0里面新加的一种淘汰策略。它的全称是Least Frequently Used，它的核心思想是根据key的最近被访问的**频率**进行淘汰，很少被访问的优先被淘汰，被访问的多的则被留下来。

**LFU算法能更好的表示一个key被访问的热度**。假如你使用的是LRU算法，一个key很久没有被访问到，只刚刚是偶尔被访问了一次，那么它就被认为是热点数据，不会被淘汰，而有些key将来是很有可能被访问到的则被淘汰了。如果使用LFU算法则不会出现这种情况，因为使用一次并不会使一个key成为热点数据。

LFU一共有两种策略：

volatile-lfu：在设置了过期时间的key中使用LFU算法淘汰key

allkeys-lfu：在所有的key中使用LFU算法淘汰数据

注：要注意的一点是这两种策略只能在Redis4.0及以上设置，如果在Redis4.0以下设置会报错。

# 持久化

持久化，即将数据持久存储，而不因断电或其他各种复杂外部环境影响数据的完整性。

由于Redis将数据存储在内存而不是磁盘中，所以内存一旦断电，Redis中存储的数据也随即消失，这往往是用户不期望的，所以Redis有持久化机制来保证数据的安全性。

**Redis 如何做持久化？**

Redis目前有两种持久化方式，即**RDB和AOF**，RDB是通过保存某个时间点的**全量数据快照**实现数据的持久化，当恢复数据时，直接通过RDB文件中的快照，将数据恢复。

## 持久化方案

一般我们在生产上采用的持久化策略为

1、master关闭持久化

2、slave开RDB即可，必要的时候AOF和RDB都开启

该策略能够适应绝大部分场景，绝大部分集群架构。

**为什么是绝大部分场景？**

因为这套策略存在部分的数据丢失可能性。redis的主从复制是异步的，master执行完客户端请求的命令后会立即返回结果给客户端，然后异步的方式把命令同步给slave。因此master可能还未来得及将命令传输给slave，就宕机了，此时slave变为master，数据就丢了。

幸运的是，绝大部分业务场景，都能容忍数据的部分丢失。假设，真的遇到缓存雪崩的情况，代码中也有熔断器来进行资源保护，不至于所有的请求都转发到数据库上，导致我们的服务崩溃！

注：这里的缓存雪崩是指同一时间来了一堆请求，请求的key在redis中不存在，导致请求全部转发到数据库上。

**为什么是绝大部分集群架构？**

因为在集群中存在redis读写分离的情况，就不适合这套方案了。

幸运的是，由于采用redis读写分离架构，就必须要考虑主从同步的延迟性问题，徒增系统复杂度。

目前业内采用redis读写分离架构的项目，真的太少了。

**master关闭持久化**

原因很简单，因为无论哪种持久化方式都会影响redis的性能，哪一种持久化都会造成CPU卡顿，影响对客户端请求的处理。为了保证读写最佳性能，将master的持久化关闭！

**slave开RDB即可，必要的时候AOF和RDB都开启**

首先，我先说明一下，我不推荐单开AOF的原因是，基于AOF的数据恢复太慢。

你要想，我们已经做了主从复制，数据已经实现备份，为什么slave还需要开持久化?

因为某一天可能因为某某工程，把机房的电线挖断了，就会导致master和slave机器同时宕机。

那么这个时候，我们需要迅速恢复集群，而RDB文件文件小、恢复快，因此灾难恢复常用RDB文件。

其次，官网也不推荐单开AOF，地址如下:

https://redis.io/topics/persistence

所以，如果实在对数据安全有一定要求，将AOF和RDB持久化都开启。

另外，做好灾难备份。利用linux的scp命令，定期将rdb文件拷贝到云服务器上。

注：scp是secure copy的简写，用于在Linux下进行远程拷贝文件的命令，和它类似的命令有cp，不过cp只是在本机进行拷贝不能跨服务器，而且scp传输是加密的。

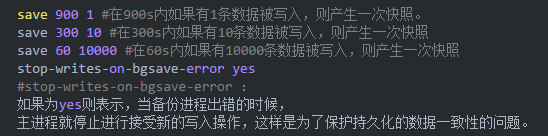
## RDB（快照）持久化

RDB持久化是将当前进程中的数据生成快照保存到硬盘(因此也称作快照持久化)，保存的文件后缀是rdb；当Redis重新启动时，可以读取快照文件恢复数据。

RDB持久化会在某个特定的间隔保存那个时间点的**全量数据的快照**。

### 原理

RDB 配置文件，redis.conf：



1、RDB 的创建与载入

**SAVE：**阻塞Redis的服务器进程，直到RDB文件被创建完毕。SAVE命令很少被使用，因为其会阻塞主线程来保证快照的写入，由于Redis是使用一个主线程来接收所有客户端请求，这样会阻塞所有客户端请求。

**BGSAVE：**该指令会fork出一个子进程来创建RDB文件，不阻塞服务器进程，子进程接收请求并创建RDB快照，父进程继续接收客户端的请求。

子进程在完成文件的创建时会向父进程发送信号，父进程在接收客户端请求的过程中，在一定的时间间隔通过轮询来接收子进程的信号。

我们也可以通过使用lastsave指令来查看BGSAVE是否执行成功，lastsav可以返回最后一次执行成功BGSAVE的时间。

2、自动化触发RDB持久化的方式

自动化触发RDB持久化的方式如下：

根据 redis.conf配置里的SAVE m n定时触发（实际上使用的是BGSAVE）。

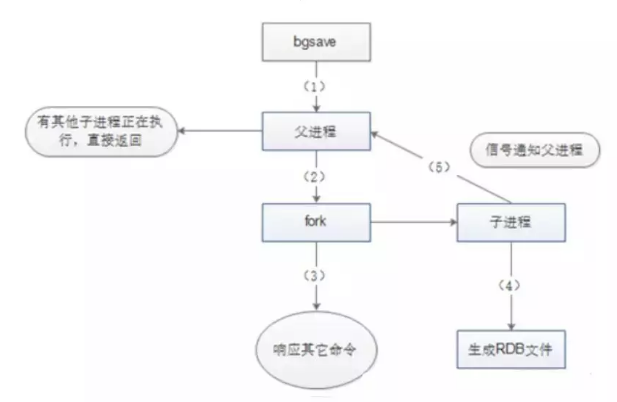
主从复制时，主节点自动触发。

执行Debug Reload。

执行Shutdown且没有开启AOF持久化。

1. BGSAVE的原理

那么RDB持久化的过程，相当于在执行bgsave命令。该命令执行过程如下图所示：



**启动：**

检查是否存在子进程正在执行AOF或者RDB的持久化任务。如果有则返回 false。主线程需要调用系统函数fork()，构建出一个子进程进行持久化！很不幸的是，在构建子进程的过程中，父进程就会阻塞，无法响应客户端的请求！

调用Redis源码中的rdbSaveBackground方法，方法中执行fork()产生子进程执行 RDB 操作。

注：在测试中发现，fork函数在虚拟机上较慢，真机上较快。考虑到现在都是部署在docker容器中，很少部署在真机上，为了性能，master不建议打开RDB持久化！

关于fork()中的Copy-On-Write。

fork()在Linux中创建子进程采用Copy-On-Write（写时拷贝技术），即如果有多个调用者同时要求相同资源（如内存或磁盘上的数据存储）。

他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本给调用者，而其他调用者所见到的最初的资源仍然保持不变。

### 特点

RDB优点：**全量**数据快照，**文件小，恢复快**。

RDB缺点：**无法保存最近一次快照之后的数据**。

RDB持久化方式的缺点如下：

内存数据**全量同步**，**数据量大的状况下，会由于I/O而严重影响性能**。

可能会因为Redis宕机而丢失从当前至最近一次快照期间的数据。

## AOF持久化

AOF持久化（保存写状态）是通过**保存Redis的写状态**来记录数据库的。

相对RDB来说，RDB持久化是通过备份数据库的状态来记录数据库，而AOF持久化是备份数据库接收到的指令：

AOF记录除了查询以外的所有变更数据库状态的**指令**。

以**增量**的形式追加保存到 AOF 文件中。

### 开启AOF持久化

1、打开 redis.conf 配置文件，将appendonly属性改为yes。

2、修改 appendfsync属性，该属性可以接收三种参数，分别是always，everysec，no。

always表示总是即时将缓冲区内容写入AOF文件当中，everysec表示每隔一秒将缓冲区内容写入AOF文件，no表示将写入文件操作交由操作系统决定。

一般来说，操作系统考虑效率问题，会等待缓冲区被填满再将缓冲区数据写入 AOF 文件中。

appendonly yes

#appendsync always

appendfsync everysec

# appendfsync no

### 日志重写解决AOF文件不断增大

随着写操作的不断增加，AOF文件会越来越大。于是redis有一套rewrite机制，来缩小AOF文件的体积。然而，在rewrite的过程中也是需要父进程来fork出一个子进程进行rewrite操作。至于fork函数的影响，前面已提及。

还有一个就是刷盘策略fsync，这个值推荐是配everysec,也就是Redis会默认每隔一秒进行一次fsync调用，将缓冲区中的数据写到磁盘。

然而，如果磁盘性能不稳定，fsync的调用时间超过1秒钟。此时主线程进行AOF的时候会对比上次fsync成功的时间；如果距上次不到2s，主线程直接返回；如果超过2s，则主线程阻塞直到fsync同步完成。

因此AOF也是会影响redis的性能的。

注：linux函数中，wrtie函数将数据写入文件的时候，是将数据写入操作系统的缓冲区，还并未刷入磁盘。而fsync函数，可以强制让操作系统将缓冲区数据刷入磁盘。

综上所述，我们为了保证读写性能最大化，将master的持久化关闭。

假设递增一个计数器100次，如果使用RDB持久化方式，我们只要保存最终结果100即可。

而AOF持久化方式需要记录下这100次递增操作的指令，而事实上要恢复这条记录，只需要执行一条命令就行，所以那一百条命令实际可以精简为一条。

Redis支持这样的功能，在不中断前台服务的情况下，可以重写AOF文件，同样使用到了COW（写时拷贝）。

**重写过程如下：**

调用 fork()，创建一个子进程。

子进程把新的 AOF 写到一个临时文件里，不依赖原来的 AOF 文件。

主进程持续将新的变动同时写到内存和原来的 AOF 里。

主进程获取子进程重写 AOF 的完成信号，往新 AOF 同步增量变动。

使用新的 AOF 文件替换掉旧的 AOF 文件。

### 特点

AOF优点：可读性高，适合保存**增量**数据，**数据不易丢失**。

AOF缺点：**文件体积大，恢复时间长**。

## RDB-AOF混合持久化方式

Redis 4.0之后推出了此种持久化方式，**RDB作为全量备份，AOF作为增量备份，并且将此种方式作为默认方式使用**。

在上述两种方式中，RDB方式是将全量数据写入RDB文件，这样写入的特点是文件小，恢复快，但无法保存最近一次快照之后的数据，AOF则将Redis指令存入文件中，这样又会造成文件体积大，恢复时间长等弱点。

在RDB-AOF方式下，持久化策略首先将缓存中数据以RDB方式全量写入文件，再将写入后新增的数据以AOF的方式追加在RDB数据的后面，在下一次做RDB持久化的时候将 AOF 的数据重新以RDB的形式写入文件。

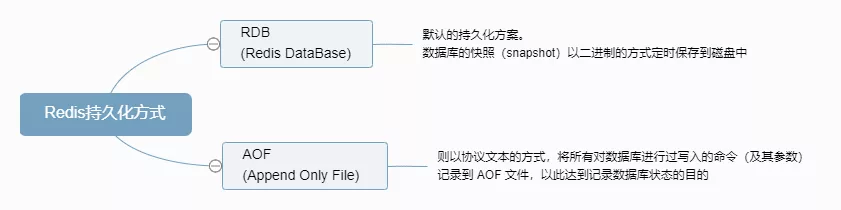
这种方式既可以提高读写和恢复效率，也可以减少文件大小，同时可以保证数据的完整性。

在此种策略的持久化过程中，子进程会通过管道从父进程读取增量数据，在以 RDB 格式保存全量数据时，也会通过管道读取数据，同时不会造成管道阻塞。

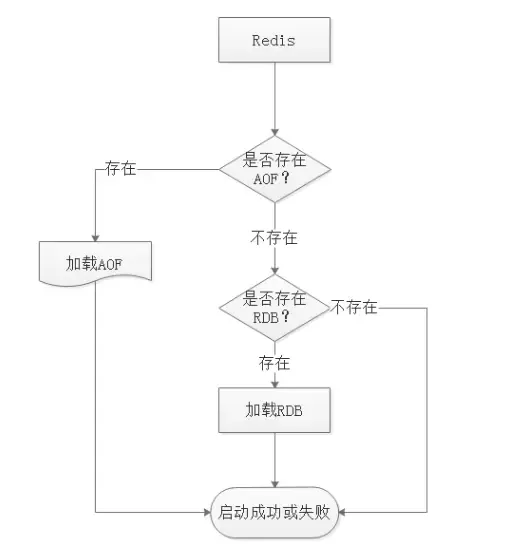
可以说，在此种方式下的持久化文件，前半段是RDB格式的全量数据，后半段是AOF格式的增量数据。此种方式是目前较为推荐的一种持久化方式。

# 恢复

Redis的持久化方式如下图：



RDB和AOF文件共存情况下的恢复流程如下图：



从图可知，Redis启动时会先检查AOF是否存在，如果AOF存在则直接加载AOF，如果不存在AOF，则直接加载RDB文件。

# Pipeline

Pipeline和Linux的管道类似，它可以让Redis批量执行指令。

Redis基于请求/响应模型，单个请求处理需要一一应答。如果需要同时执行大量命令，则每条命令都需要等待上一条命令执行完毕后才能继续执行，这中间不仅仅多了RTT，还多次使用了系统IO。

Pipeline由于可以批量执行指令，所以可以节省多次IO和请求响应往返的时间。但是如果指令之间存在依赖关系，则建议分批发送指令。

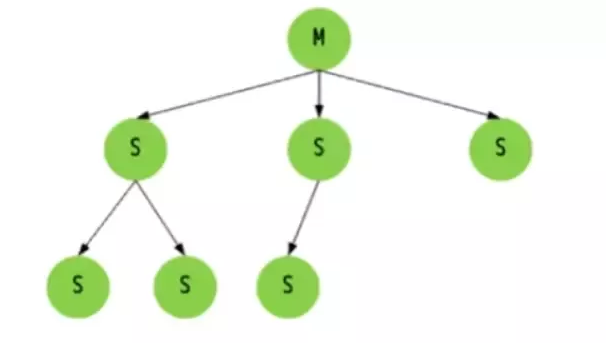
# 同步复制

## 主从同步

Redis一般是使用一个Master节点来进行写操作，而若干个Slave节点进行读操作，Master和Slave分别代表了一个个不同的Redis Server实例。

另外定期的数据备份操作也是单独选择一个Slave去完成，这样可以最大程度发挥Redis的性能，为的是保证数据的**弱一致性和最终一致性**。

另外，Master和Slave的数据不是一定要即时同步的，但是在一段时间后Master和Slave的数据是趋于同步的，这就是最终一致性。



**全同步过程如下：**

Slave 发送 Sync 命令到 Master。

Master 启动一个后台进程，将 Redis 中的数据快照保存到文件中。

Master 将保存数据快照期间接收到的写命令缓存起来。

Master 完成写文件操作后，将该文件发送给 Slave。

使用新的 AOF 文件替换掉旧的 AOF 文件。

Master 将这期间收集的增量写命令发送给 Slave 端。

**增量同步过程如下：**

Master 接收到用户的操作指令，判断是否需要传播到 Slave。

将操作记录追加到 AOF 文件。

将操作传播到其他 Slave：对齐主从库；往响应缓存写入指令。

将缓存中的数据发送给 Slave。

## Redis Sentinel（哨兵）

主从模式弊端：当Master宕机后，Redis集群将不能对外提供写入操作。Redis Sentinel可解决这一问题。

解决主从同步Master宕机后的主从切换问题：

监控：检查主从服务器是否运行正常。

提醒：通过API向管理员或者其它应用程序发送故障通知。

自动故障迁移：主从切换（在Master宕机后，将其中一个Slave转为Master，其他的Slave从该节点同步数据）。

# 集群

**如何从海量数据里快速找到所需？**

## 分片

1、分片

按照某种规则去划分数据，分散存储在多个节点上。通过将数据分到多个Redis服务器上，来减轻单个Redis服务器的压力。

## 一致性hash算法

2、一致性Hash算法

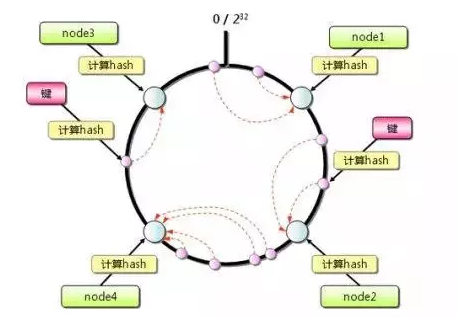
既然要将数据进行分片，那么通常的做法就是获取节点的Hash值，然后根据节点数求模。

但这样的方法有明显的弊端，当Redis节点数需要动态增加或减少的时候，会造成大量的Key无法被命中。所以Redis中引入了一致性Hash算法。

该算法对2^32取模，将Hash值空间组成虚拟的圆环，整个圆环按顺时针方向组织，每个节点依次为0、1、2…2^32-1。

之后将每个服务器进行Hash运算，确定服务器在这个Hash环上的地址，确定了服务器地址后，对数据使用同样的Hash算法，将数据定位到特定的Redis服务器上。

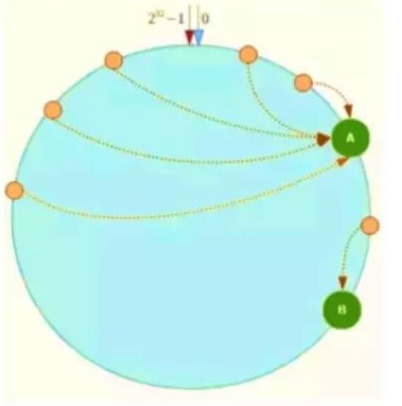
如果定位到的地方没有Redis服务器实例，则继续顺时针寻找，找到的第一台服务器即该数据最终的服务器位置。



## Hash环数据倾斜

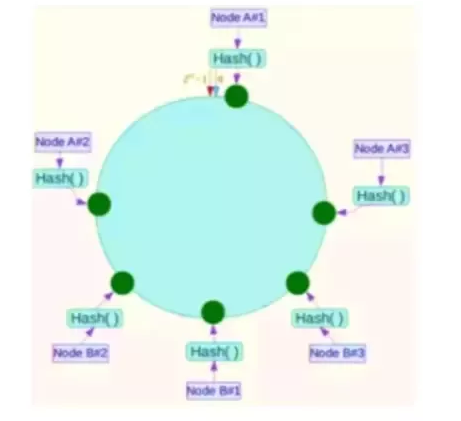
3、Hash环的数据倾斜问题

Hash环在服务器节点很少的时候，容易遇到服务器节点不均匀的问题，这会造成数据倾斜，数据倾斜指的是被缓存的对象大部分集中在Redis集群的其中一台或几台服务器上。



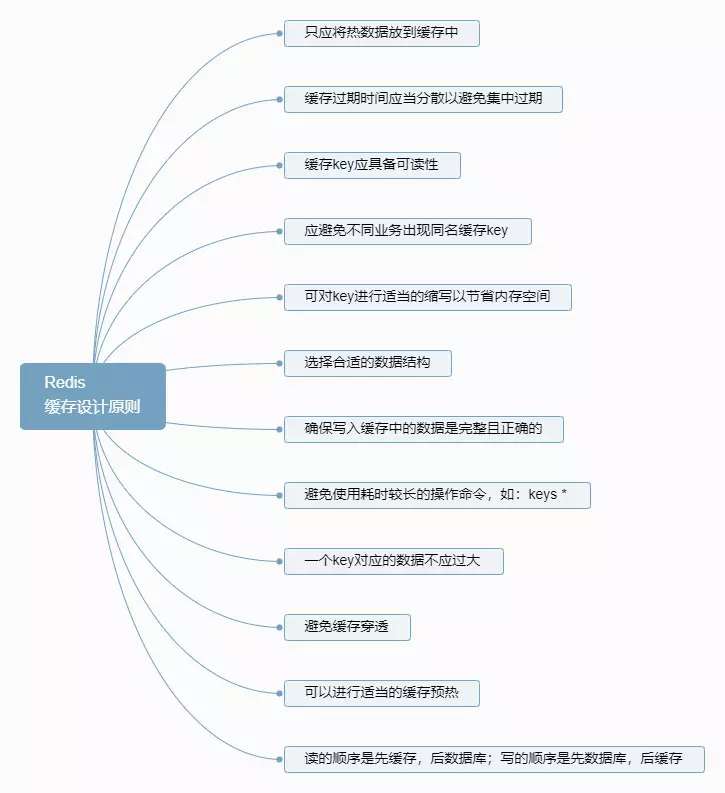
如上图，一致性Hash算法运算后的数据大部分被存放在A节点上，而B节点只存放了少量的数据，久而久之A节点将被撑爆。

针对这一问题，可以引入**虚拟节点**解决。简单地说，就是为每一个服务器节点计算多个Hash，每个计算结果位置都放置一个此服务器节点，称为虚拟节点，可以在服务器IP或者主机名后放置一个编号实现。



例如上图：将NodeA和NodeB两个节点分为Node A#1-A#3，NodeB#1-B#3。

# 缓存设计原则



# 应用

## 分布式锁

分布式锁是控制分布式系统之间共同访问共享资源的一种锁的实现。如果一个系统，或者不同系统的不同主机之间共享某个资源时，往往需要互斥，来排除干扰，满足数据一致性。

**分布式锁需要解决的问题如下：**

**互斥性**：任意时刻只有一个客户端获取到锁，不能有两个客户端同时获取到锁。

**安全性**：锁只能被持有该锁的客户端删除，不能由其他客户端删除。

**死锁**：获取锁的客户端因为某些原因而宕机继而无法释放锁，其他客户端再也无法获取锁而导致死锁，此时需要有特殊机制来避免死锁。

**容错**：当各个节点，如某个 Redis 节点宕机的时候，客户端仍然能够获取锁或释放锁。

**如何使用Redis实现分布式锁**

使用SETNX实现，SETNX key value：如果Key不存在，则创建并赋值。

该命令时间复杂度为O(1)，如果设置成功，则返回1，否则返回0。



由于SETNX指令操作简单，且是原子性的，所以初期的时候经常被人们作为分布式锁，我们在应用的时候，可以在某个共享资源区之前先使用SETNX指令，查看是否设置成功。

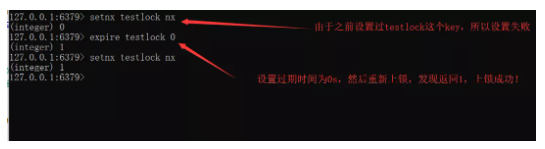
如果设置成功则说明前方没有客户端正在访问该资源，如果设置失败则说明有客户端正在访问该资源，那么当前客户端就需要等待。

但是如果真的这么做，就会存在一个问题，因为SETNX是长久存在的，所以假设一个客户端正在访问资源，并且上锁，那么当这个客户端结束访问时，该锁依旧存在，后来者也无法成功获取锁，这个该如何解决呢？

由于SETNX并不支持传入EXPIRE参数，所以我们可以直接使用EXPIRE指令来对特定的Key来设置过期时间。

用法：

EXPIRE key seconds



程序：

RedisService redisService = SpringUtils.getBean(RedisService.class);

long status = redisService.setnx(key,"1");

if(status == 1){

redisService.expire(key,expire);

doOcuppiedWork();

}

这段程序存在的问题：假设程序运行到第二行出现异常，那么程序来不及设置过期时间就结束了，则Key会一直存在，等同于锁一直被持有无法释放。

出现此问题的根本原因为：原子性得不到满足。

解决：从Redis 2.6.12版本开始，我们就可以使用Set操作，将SETNX和EXPIRE融合在一起执行，具体做法如下：

EX second：设置键的过期时间为 Second 秒。

PX millisecond：设置键的过期时间为 MilliSecond 毫秒。

NX：只在键不存在时，才对键进行设置操作。

XX：只在键已经存在时，才对键进行设置操作。

SET KEY value [EX seconds] [PX milliseconds] [NX|XX]

注：SET操作成功完成时才会返回OK，否则返回 nil。

有了SET我们就可以在程序中使用类似下面的代码实现分布式锁了：

RedisService redisService = SpringUtils.getBean(RedisService.class);

String result = redisService.set(lockKey,requestId,SET\_IF\_NOT\_EXIST,SET\_WITH\_EXPIRE\_TIME,expireTime);

if("OK.equals(result)"){

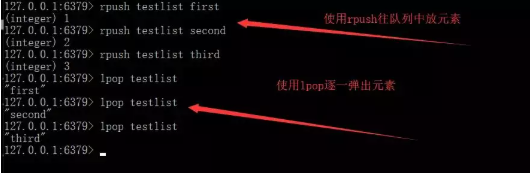
doOcuppiredWork();

}

## 异步队列

1、使用Redis中的List作为队列

使用上文所说的Redis的数据结构中的List作为队列Rpush生产消息，LPOP 消费消息。



此时我们可以看到，该队列是使用Rpush生产队列，使用LPOP消费队列。

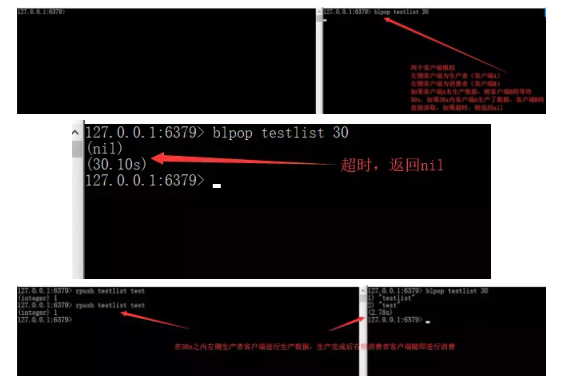
在这个生产者-消费者队列里，当LPOP没有消息时，证明该队列中没有元素，并且生产者还没有来得及生产新的数据。

缺点：LPOP不会等待队列中有值之后再消费，而是直接进行消费。

弥补：可以通过在应用层引入Sleep机制去调用LPOP重试。

2、使用BLPOP key [key…] timeout

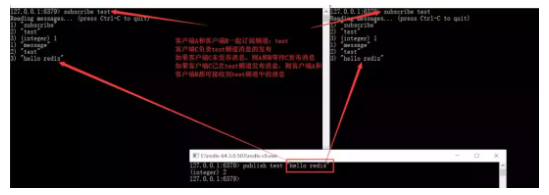
BLPOP key [key …] timeout：阻塞直到队列有消息或者超时。



缺点：按照此种方法，我们生产后的数据只能提供给各个单一消费者消费。能否实现生产一次就能让多个消费者消费呢？

3、Pub/Sub：主题订阅者模式

发送者（Pub）发送消息，订阅者（Sub）接收消息。订阅者可以订阅任意数量的频道。



Pub/Sub模式的缺点：消息的发布是无状态的，无法保证可达。对于发布者来说，消息是“即发即失”的。

此时如果某个消费者在生产者发布消息时下线，重新上线之后，是无法接收该消息的，要解决该问题需要使用专业的消息队列，如 Kafka…此处不再赘述。