**分布式缓存**

**在高并发环境下，大量的读写请求涌向数据库，此时磁盘IO将成为瓶颈，从而导致过高的响应延迟，缓存应运而生，常见的分布式缓存有memCached和redis。**

# 1. Memcached

传统的web应用将数据保存到RDBMS（关系型数据库）中，应用服务器从RDBMS中读取数据，、处理并在浏览器中显示，但是随着数据量增大，访问的集中，就会出现RDBMS负载过重，数据库响应变慢，导致整个系统响应延迟增加。

Memcached是一款高性能的分布式内存缓存服务器，其目的是为了通过缓存数据库的查询命中减少数据库压力，提高应用响应速度、提高可扩展性。

## 1.1.memcached缓存特点

* 协议简单：

Memcached的协议实现简单，使用的是基于文本行的协议，能通过Telnet直接操作memcached服务存取数据

* 基于libevent的事件处理：

Libevent是一套C开发的程序库，有BSD的kqueue和Linux的epoll等事件处理功能，确保及时服务端连接数增加，也能发挥较好的性能

* 内置内存存储方式

Memcached有自己的一套内存管理方式，所有数据都保存在memcached内置的内存中，当存入的数据占满内存后，将使用LRU算法删除过期缓存，重用过期数据的内存空间

* Memcached不相互通信的分布式特征

各个memcached服务器之间互不通信，都是独立存取数据，不共享任何信息，通过对客户端的设计，让memcached具有分布式功能。

## 1.2.memcached工作原理

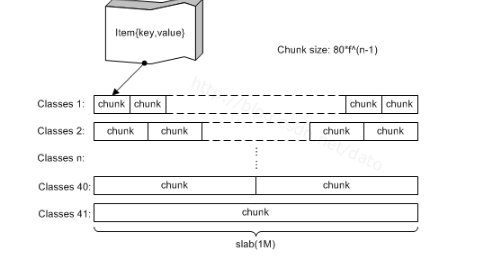
1. memcached是一套C/S模式架构软件，服务器端启动服务守护进程，为memcached服务器指定监听的IP地址、端口号、并发访问的连接数以及分配多少内存
2. memcached软件由C语言实现，全部代码2000多行，采用异步IO，实现方式是基于事件的单线程或单进程，使用libevent作为事件通知机制，每个服务器只管理自己的数据
3. 需要被缓存的数据以<key,value>形式保存在服务器端预分配的内存空间，每个被缓存的数据都有唯一的key，操作数据均是通过key值进行
4. Memcached并没有对缓存的数据进行持久化的存储设计，因此在服务器端的memcached服务进程重启之后，存储在内存中的数据就会丢失。且当内存中缓存的数据容量达到启动时指定的内存值后，就会自动使用LRU算法删除过期的缓存数据。
5. 为满足数据持久性保留需求，sina基于memcached研发了一款NOSQL软件-----MemcachedDB，实现缓存的基础上增加了持久缓存的特性。

## 1.3.内存分配管理

以前的内存分配是通过对记录简单进行malloc和free进行的，会导致内存碎片，memcached的slab allocation解决了该问题。

Memcached在启动时通过-m参数指定最大使用内存，但一开始并不会马上分配这么多，而是逐步分配给各个slab。

Slab allocation是通过按照预先规定的大小，内存分配以slab（or page）为单位，默认为1MB，可通过-l参数在启动时指定，内存的分配是在需要的时候进行，每次申请一定是一个slab，然后将一个slab分割为相同大小的chunk，这些相同大小的chunks称为一个slab class。



每次添加数据item的时候，都会寻找一个大小最接近item且大于item的slab，然后将其放置在其中一个chunk中，当某个slab class中的chunk分配完后，将会申请IMB的空间给该slab class。

即使如此，依然无法完全有效的使用内存空间，memcached在启动时指定growth factor（通过-f选向），可以在某种程度上控制slab之间的差异，默认值为1.25。

当内存分配完后，会对slab进行LRU而不是整个memcached。

预分配方式，每个slab class在初始化阶段公平分配到了约为1M的连续内存，实际业务中，不同大小的chunk使用频率不同，有的slab中chunk很快被用完，有的长期未被用到，造成内存浪费，所以memcached默认采用按需分配的方式。

## 1.4.数据的高效索引

Memcached通过哈希表来对键值对进行管理，具体实现则用链表法解决hash冲突。

HashTable被分成多个桶bucket，每个item通过hash函数确定具体的bucket，如果桶内已有数据，则通过链表法解决冲突。

## 1.5.LRU链表

Memcached的每个slab都维护了一个LRU链表，用来组织该slab中已经被分配的item，每当新的chunk被使用，就会将该chunk的item添加到LRU链表头部，或者原有item被修改，也会将其从链表中移动到头部，这样头部就是最新添加或修改的数据，而item为slab中存储最旧的数据。

Memcached采用惰性删除机制，系统不会主动监视item的数据是否过期，而是在get的时候查看item的时间戳，过期则删除并将该chunk放到空闲链表。

插入数据的时候，memcached会优先查看LRU链表尾部item是否超时，memcached会优先删除超时的item并将其chunk作为新数据的存储空间。

当尾部item未超时，slab中无可用chunk，且系统内存分配完毕，则强行删除尾部item，分配给新数据。

## 1.6.删除操作

* 通过hash表获取该key数据item
* 从哈希表中删除item节点
* 从LRU中移除item节点
* 清空item数据，并将该chunk添加到空闲链表

## 1.7.memcached分布式

客户端采用一致性哈希算法。

### 1.7.1.一致性哈希算法原理

一致性哈希是将整个哈希空间组织成一个虚拟圆环，如假设哈希函数的值空间为[0,232-1]，整个空间按顺时针方向组织，0和232-1重合，利用hash函数确定服务器在哈希环上的位置，可以选择服务器的IP地址或是主机名作为关键字进行哈希，接下来使用相同的哈希函数计算数据key的哈希值，并确定此数据在环上的位置，从此位置沿环顺时针查找，遇到的第一台服务器就是放置数据的位置。

### 1.7.2.一致性哈希的性质

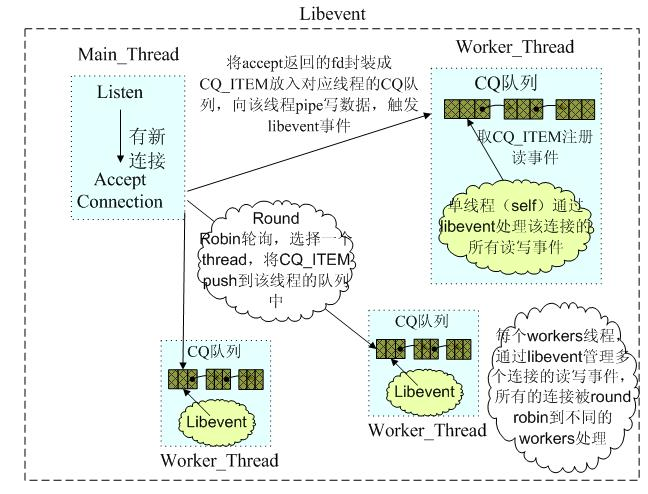
* 平衡性：哈希结果能尽可能分不到所有缓冲区，使得所有缓冲区都能得到有效利用
* 单调性：当有新的缓冲区加入到系统中，保证原有已经分配的内容可以被映射到新的缓冲区，但不能被映射到旧的缓冲区，也就是说数据因为新节点的加入，迁移的方向只能是原有位置到新节点，不能是原有某个节点到原有的另一个节点
* 分散性：由于终端所见的缓冲范围可能不同（有的节点对于client不可见），导致相同的内容被映射到不同的缓冲区，分散性就是定义这种问题的严重程度，好的哈希算法应该能尽量降低分散性

### 1.7.3.虚拟节点

服务器经过哈希函数分布到环上的不同位置，有的服务器由于位置的原因可能会处理大部分的请求，造成一致性哈希倾斜问题，虚拟节点有效解决了这个问题。

我们可以通过在IP地址后面增加编号来实现，为每个节点增加多个虚拟节点，使得环上的服务器能够分布的更均匀。

## 1.8.memcached多线程工作机制



每个线程包括主线程都有各自独立的libevent实例，memcached的listen fd注册到主线程的libevent实例上，由主线程来accept新的连接，接受新的连接后主线程选择工作线程，将新连接的socket fd封装为CQ\_ITEM后push到所选的工作线程CQ队列中，然后主线程通过管道发送字符“c”到工作线程，工作线程收到“c”将从自己的CQ队列中pop出CQ\_ITEM进而取出新连接并将fd注册到工作线程的libevent实例上，从而由工作线程处理该链接的所有后续事件。

需要注意的是，memcached默认开启线程数为4，可以通过参数-t指定。

# 2. redis

Redis和memcached同为分布式缓存，在支持的数据类型、内存管理方式、分布式部署、是否单线程、是否支持持久化以及应用场景等很多方面存在不同。

Struct RedisServer

{

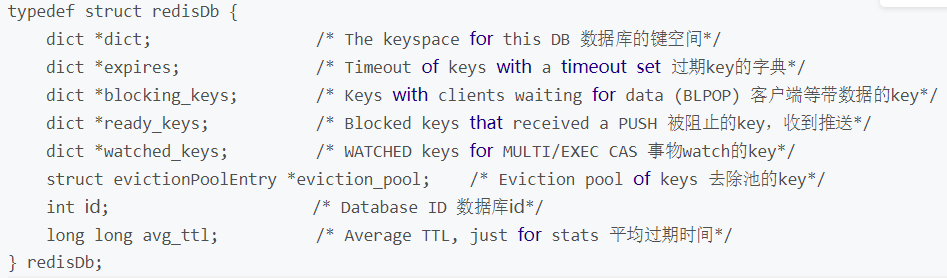
redisDb\* db; //保存服务器中所有数据库

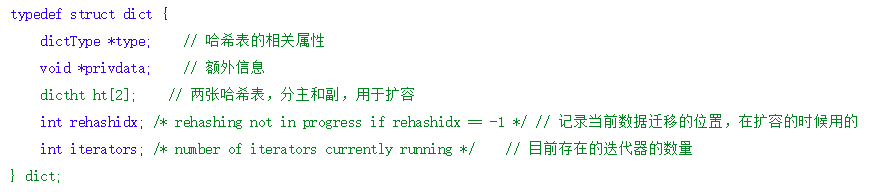
int dbnum； //服务器中数据库数量

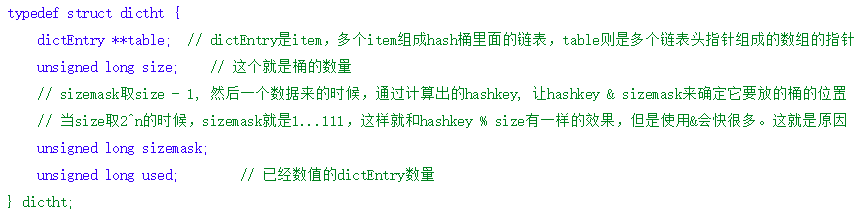
….

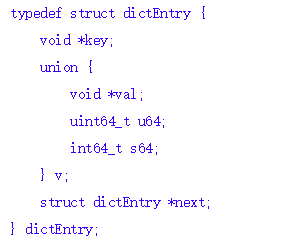
dict \*pubsub\_channels; //订阅频道

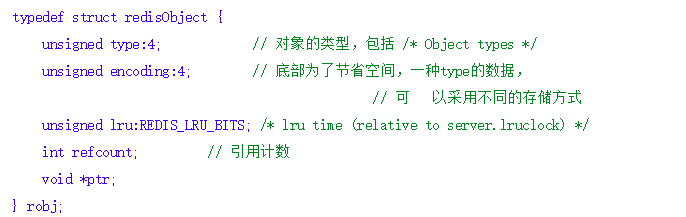
}











## 2.1.数据类型

Redis支持的数据类型有五种，String、list、set、sorted set、hash，其key或value的最大值是512MB（对比memcached的key最大为250字符，value不能超过1MB）。

Redis说到底还是一个key-value数据库，不论它支持多少种数据结构，最终存储还是以key-value方式，只不过value可以是list、set、sorted set、HashTable等。和memcached一样，所有的key都是String，而key与value的关联是怎么实现的，java中用的是HashMap，redis用C语言重写了一张哈希表。

* Dict：哈希表结构
* Dictht：一个dict有两个dictht（一个用于存储数据，一个用于扩容），dictht[0]存储数据，dictht[1]用于扩容（大小是dictht[0]的两倍）
* dictEntry：一个dictht有一个dictEntry数组，每个dictEntry有next指针，dictEntry存储实际的<key,value>，key为字符串，value的实际的对象是RedisObject。实际上和HashMap类似，是数组+链表的方式，解决Hash冲突是链表法。

值得一提的是rehash的方法，称为“单线程渐进式重哈希法”，而concurrentHashMap的rehash也是渐进式的，不过是多线程，redis的rehash是将dictht[0]的数据以桶为单位转移到dictht[1]中，维护rehashidx表示当前迁移的进度。数据的迁移不是一次完成的，因为dictht[0]中如果存储了海量数据，一次迁移必然会带来redis性能下降，真正的迁移是在client的每次请求（插入、删除、更新）过程执行，也就是说将迁移的过程分摊在各个请求中，直至迁移完成。

concurrentHashMap的rehash是如果线程发现当前更新位置正在迁移，在帮助扩容，如果扩容没有进行到当前位置则直接返回结果。

### 2.1.1.String

字符串类型，实际上可以是字符串（XML、json），还有数字（整型、浮点型），二进制等，最大不能超过512MB。

常用命令：set/get/decr/incr/mget

String在redis内部存储默认是一个字符串，被RedisObject引用，遇到incr/decr转化为数值计算。

/\* Redis简单动态字符串的数据结构 \*/

struct sdshdr {

//字符长度，记录buf数组中已使用的字节数量

unsigned int len;

//当前可用空间，记录buf数组中未使用的字节数量

unsigned int free;

//具体存放字符的buf

char buf[];

};

### 2.1.2.Hash

Hash可以使用dict实现，每个dictEntry保存<key,value>，他们都是string。

常用命令：hget/hset/hgetall等

### 2.1.3.List

List的实现由两种，普通双向链表和压缩链表。

常用命令：lpush/rpush/lpop/rpop/lrange等；

### 2.1.4.set

Set有dict和intset（全是整数时使用），set中的dict和java关于HasSet的实现相类似，都是用哈希实现，key可变，value为固定值，这里为null。

### 2.1.5.sorted set（zset）

Sorted set是用skiplist，即跳表实现的，跳表是红黑树的替代品，能够达到红黑树的效率，且实现比红黑树要简单。

使用skiplist实现zset的时候，还使用了一个dict，dict一样存储了键值对，skiplist可以达到logn（有时候n），而dict可以达到O(1)。

跳表的性质：

* 有很多层组成
* 每一层都是一个有序链表
* 如果一个元素出现在level i层，那么level i之下的链表中都有此元素
* 每个节点包含两个指针，一个指向当前level的下一个元素，一个指向下一层level的元素
* 跳表的平衡是通过随机函数维护，当我们往跳表中插入数据时，我们通过随机函数决定这个节点插入哪几层索引中

与红黑树相比，跳表的有点：

* 插入速度快，不需要平衡树的旋转操作
* 算法更容易实现
* 支持无锁操作

### 2.1.6.redisObject

Redis存储所有数据使用redisObject来封装，包括String、hash、list、set、sorted set。RedisObject的字段如下：

* Type字段：保存对象使用的数据类型
* Encoding字段：保存对象使用的内部编码类型
* LRU字段：记录对象最后一次被访问的时间（用于内存回收）
* Recount字段：记录对象的引用计数（用于回收）
* \*ptr字段：存储值对象的数据或指针，如果是整数，直接存储数据，否则指向数据的指针

### 2.1.7.key查询

当需要遍历redis所有key或是指定模式的key时，redis提供了两种模糊查询的指令：KEYS和scan。

指令：KEYS Pattern，当查询的数据过大时，由于redis是单线程，可能会造成线程阻塞。

Redis2.8以上版本提供了一个更好的遍历key的命令：

SCAN cursor [Match pattern] [COUNT count]

SCAN命令每次都只返回少量元素，所以不会出现KEYS命令可能带来服务器阻塞问题。

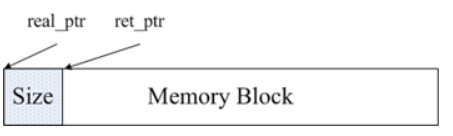
* Cursor：Scan命令是一个基于游标的迭代器，每次都需要使用上一次这个调用返回的游标作为该次调用的游标参数
* Match pattern：模糊查询，模式匹配
* COUNT count：可以使用count控制每个迭代返回元素数量

## 2.2.内存管理

### 2.2.1.内存分配

Memcached使用slab allocation机制实现管理，虽然不会造成内存碎片，但会造成空间浪费。

Redis内存管理主要通过源码找那个的zmalloc.h和zmalloc.c两个文件实现的，redis为了方便内存管理，分配一块内存后，会将这块内存的大小存入内存块的头部，



Redis通过定义一个数组来记录所有内存的分配情况，数组的每个元素代表当前程序分配的内存块的个数，内存块的大小就是下标，源码中这个数组是zmalloc\_allocations，zmalloc\_allocations[16]代表已经分配的长度为16字节的内存块的个数，还有一个静态变量used\_memory用来当前分配的内存总大小。

总的来看，redis采用的是包装的malloc/free，相比较于memcached的内存管理，简单很多。

### 2.2.2.过期键删除策略

Redis提供了两种方案删除过期键：

* 惰性删除，即当我们访问某个key时检查它是否过期
* 定期删除，是由redis eventloop触发

AOF和RDB键过期处理策略：

* RDB持久化时，发现key已经过期，直接忽略，只持久化那些没有过期的key
* AOF持久化时，每次遇到过期的key，redis都会追加一条del命令到AOF文件中

### 2.2.3.内存淘汰策略

当client要求分配内存存放key发现内存不够用时，Redis支持6种淘汰策略：

* No eviction：不删除策略，达到最大内存限制，如果需要更多内存，直接返回错误
* Allkeys-lru：所有key通用，优先删除最近最少使用的key
* Volatile-lru：只限于设置了expire部分，优先删除最近最少使用的key
* Allkeys-random：所有key通用，随机删除一部分key
* Volatile-random：只限于设置了expire部分，随机删除一部分key
* Volatile-ttl：只限于设置了expire部分，删除过期时间剩余时间最短的key

### 2.2.4.内存优化

* 缩减键值对象的长度：简化键名，使用高效的序列化工具序列化对象，还可以使用压缩工作压缩序列化后的数据
* 共享对象池：redis内部维护[0,9999]的整数对象池，对于0-9999内部整数类型元素整数值对象都会直接引用整数对象池中对象（注意：启用LRU相关溢出策略，无法使用共享对象池）
* Redis对于字符串有一些优化

1. redis的string采用SDS，特点是获取长度和未用长度块，内部实现了预分配机制，惰性删除机制，字符串删除后不释放，而是作为预分配空间保留
2. 减少使用append，改用set一次性创建字符串，减少预分配带来的内存浪费和内存碎片
3. 字符串重构

## 2.3.持久化支持

Redis与memcached的最大不同，就是redis支持数据持久化，redis的持久化，分为两种策略，用户可以配置使用不同的策略。

### 2.3.1.RDB持久化

RDB就是snapshot存储，是默认的持久化方式，按照一定策略周期性将数据写到磁盘，产生dump.rdb文件，可以通过配置文件中的save参数定义快照周期。

Redis借助fork命令的copy on write机制，在生成快照时，将当前进程fork出一个子进程，然后在子进程中循环所有数据，将数据写为RDB文件。

RDB文件是二进制文件，方便在网络中传输与保存，所以redis的主从复制可由RDB实现。

#### 2.3.1.1.实现方式

Redis提供了两种命令生成RDB文件

* Save：会阻塞redis线程，知道RDB文件创建完成，期间无法提供服务
* Bgsave：fork命令产生子进程，然后由子进程创建RDB文件，不会阻塞redis线程服务

Client也可以通过save或者bgsave命令通知redis做一次快照持久化，save操作时在主线程中保存快照，由于redis用一个主线程处理所有client请求，这种方式会阻塞client的请求，不推荐使用，每次持久化都是将内存数据完整写入磁盘，并不是增量同步脏数据。

#### 2.3.1.2.存在问题

由于RDB快照存储按照一定策略周期将数据写到磁盘，所以RDB方式容易造成数据丢失， redis恢复的时候将缺失快照之后到出问题时的数据。

### 2.3.2.AOF持久化

快照并不可靠，如果节点突然crash，最新数据将丢失，AOF（append only file）文件提供了一种更为可靠的持久化方式，每当redis接收到修改数据集的命令时，就会把命令追加到AOF文件中，当redis重启后，AOF的命令将被重新执行。

#### 2.3.2.1.AOF配置参数

* appendonly：是否打开AOF持久化功能
* appendFilename：AOF文件名称
* appendfsync：同步频率，如下表

|  |  |
| --- | --- |
| 选项 | 同步频率 |
| Always | 每个redis命令都要同步写入磁盘，严重降低性能 |
| Everysec | 每秒执行一次同步 |
| No | 让操作系统决定何时同步 |

#### 2.3.2.2.实现过程

* redis打开AOF持久化功能
* redis执行完写命令，将写命令追加到redis内部缓冲区
* 接下来缓冲区写命令被追加到磁盘AOF文件，当设置选项为always，每次EventLoop都会将缓冲写入磁盘，当选项为everysec，每秒钟都会写入磁盘，当选项为no，redis不管缓冲区的写命令而交由操作系统处理何时写入磁盘

#### 2.3.2.3.可能存在问题

* 每次eventloop都将写命令追加到磁盘，性能过低？
* 写操作不断执行，AOF文件越来越大？

解决方案如下：

* 主线程fork一个子线程，根据现有内存数据库反向生成保存数据的命令（与RDB方式相同）
* AOF文件过大，我们只需要保证能够复现内存数据库的完整状态命令，对于多余的历史写命令可以删除，redis支持不断提供服务的情况下，后台重建AOF日志，新的AOF日志会保留重建当前数据集最短命令序列

Redis server中有一个sds aof\_buf，如果aof持久化打开，每个数据修改命令都会存入这个aof\_buf中，然后event loop每次循环，都会调用flushaofbuff，把aof\_buf命令写入aof文件中（其实是写入redis内部缓冲区），写入磁盘的机制可以通过redis配置，可以设置每秒一次fsync、每次event loop进行fsync。

如果不想保存每次的修改命令，也可以使用aof持久化，redis提供aof\_rewrite，即根据现有的数据库生成命令，主线程fork一个子线程，子线程遍历数据库，根据<key,value>生成不同的保存命令，即根据数据反向生成保存数据的命令，然后将这些命令存储aof文件。注意，子进程写aof文件时，主线程可能还在接收client的写请求，父进程会把更新写在一个文件中，子进程完成通知父进程后，父进程将新的更新命令append到aof文件中。

随着写操作的不断执行，AOF变得越来越大，而这些AOF日志中针对某一特定数据的命令，我们只需要保证能够复现其完整状态的命令，redis支持在不断提供服务的情况下后台重建AOF日志，在重建过程中，新的写请求依然写在旧文件尾部，新的AOF日志会保留重建当前数据集的最短命令序列。

### 2.3.3.推荐使用

RDB和AOF都是顺序IO操作，性能很高，恢复时也是顺序读取数据加载到内存，都不会造成随机IO。

官方建议如下：如果想要提供很高的数据库保障，建议使用两种持久化方式，如果可以接受几分钟的数据丢失，可以使用RDB。

RDB在数据启动时间更短，原因如下：

* RDB中数据每条数据只有一条记录，不会像AOF那样可能一条数据对应多条操作记录，所以每条数据只写一次
* RDB的文件存储格式和redis在内存中的编码格式是一致的，所以不需要进行数据编码，CPU消耗远小于AOF日志加载

综上所述，我们推荐RDB，因为它可以提供数据的完整快照以及提供更快的重启。

## 2.4.redis的分布式

Memcached本身并不支持分布式，是在client通过一致性哈希算法实现memcached的分布式存储。

### 2.4.1.无中心

Redis则是在服务器端构建分布式存储。Redis cluster是一个实现了分布式且允许单点故障的redis高级版本，他没有中心节点，具有线性可伸缩性功能。Redis集群中节点与节点之间通过二进制协议进行通信，节点与客户端之间通过文本协议进行通信。

每个节点都与其他所有节点连接，连接保持活跃，使用gossip协议传播信息以及发现新节点。

### 2.4.2.预分桶

和memcached一样，redis也采用一定算法进行key🡪slot之间的映射，memcached采用一致性哈希进行映射，而redis集群使用集群公式计算key属于哪个slot，每个redis节点负责一部分slot的管理

Hash\_Slot(key) = CRC16(key)%16384

* 当需要增加一个节点时，只需要把其他节点的某些slot挪到新节点
* 当需要移除一个节点时，只需要把移除节点上的slot挪到其他节点

### 2.4.3.可用性

为了保证服务的可用性，采取master-slave，当master宕机后，选举一个slave成为新的master。

### 2.4.4.Hash Tag

通过分片的手段，可以将数据合理划分到不同节点上，但有时会将一些关联性很强的数据分布到不同的节点上，如果对这些关联性强的数据做原子操作，那么会产生分布式事务。

Redis中引入Hash Tag概念，使得分布算法可以根据key的某一部分进行计算，然后让相关的key落到同一个数据分片。

### 2.4.5.分片迁移

在一个稳定的redis集群中，每一个slot对应的节点是确定的，但是在某些情况下，节点和分片的对应关系会发生变更。比如新节点的加入和删除某个节点。

* 新增节点：redis集群的做法是从每个节点中拿出一部分前面的slot到新节点中
* 删除节点：将待删除节点的slot数据分均分布到各个节点中

### 2.4.6.slot迁移

槽的迁移过程存在一个不稳定的状态，这个不稳定状态会有一些规则，通过这些规则定义client的行为，从而使得redis集群不必要暂停服务进行槽迁移。

* 向节点B发送状态变更命令，设置其状态为importing
* 向节点A发送状态变更命令，设置其状态为migrating
* 如果节点A处于migrating状态，则如果key还在节点A直接返回，若已经迁移到节点B，则返回给client ASK信息让他跳转到节点B执行
* 如果节点B处于importing状态，如果key不在节点B上，则告知其去节点A取数据

### 2.4.7.redis集群

* 高可用：Redis哨兵机制着眼于高可用，在master宕机后自动将slave提升为master，继续提供服务
* 扩展性：redis集群着眼于扩展性，在单个redis内存不足，使用cluster进行分片存储

## 2.5.redis的主从复制

Redis中，用户可以通过执行slaveof命令或者设置slaveof选项，让一个服务器去复制另一个服务器，client是向从节点发送命令 slaveof 主机ip，告诉从节点自己的主节点是谁。

Redis的主从复制功能分为同步（sync）和命令传播（command propagate）两个操作：

* 同步操作用于将主节点的数据库状态更新至从节点上
* 命令传播用于主节点数据库状态改变，导致主从数据库不一致性，让主从重新恢复一致

### 2.5.1.同步

从节点对主节点的同步操作需要通过向主节点发送sync命令完成：

1. 从节点发送sync命令
2. 收到sync命令后主节点执行bgsave，在后台生成一个RDB文件，并使用一个缓冲区记录从现在开始的所有写命令
3. 当bgsave执行完毕，主节点将RDB发送给从节点，从节点接受并载入这个RDB文件
4. 主节点将记录在缓冲区的所有写命令发送给从节点，从节点执行这些命令，将自己的数据库状态更新至主节点一致状态

### 2.5.2.命令传播

执行完同步之后，主从数据一致，当主节点再次执行写操作，其数据库状态就会修改，导致 不再一致，主节点对从节点执行命令传播操作，将自己的执行的写命令发送给从节点。

### 2.5.3.旧版缺陷

Redis中，主从复制可分为两种情况：

* 初次复制：从节点第一次连接到主节点
* 断线后重复制：重新连接主节点后会复制所有内容

问题在于断线重连后，如果复制主节点所有内容，效率很低。Sync同步命令是一个非常耗费资源的操作

* 主节点需要执行bgsave命令生成RDB文件，会耗费大量CPU、内存和磁盘IO
* 如果RDB文件过大，占用的网络带宽过大
* 从节点接收RDB文件载入过程中，从节点因为阻塞无法提供对外服务

### 2.5.4.新版改进

为了解决旧版在断线重连时的低效，redis2.8版后使用psync命令复制同步操作。

Psync有两种复制模式：

* 完整重同步：用于处理初次复制，基本和sync相同，
* 部分重同步：主要解决断线重连，主节点将断开期间的写命令发送给从节点，从节点接收并执行

### 2.5.5.部分重同步

部分重同步功能由以下三部分组成：

* 主节点复制偏移量、从节点复制偏移量：主节点每次发送N字节数据，复制偏移量加N，从节点接收N字节，复制偏移量加N，如果主从复制偏移量相同，则处于一致状态
* 主节点复制积压缓冲区：是一个主节点维护的固定长度的先进先出队列，默认大小为1MB，当队列满后，最先入队的命令出队，主节点的复制积压缓冲区放置最近传播的写命令，当从节点重连主节点，通过psync发送自己的offset：

1. 如果从节点offset之后的数据仍在复制积压缓冲区，则主从执行部分重同步
2. 如果offset之后复制有一部分或是完全不在复制积压缓冲区，主从执行完全重同步

* 服务器的运行ID：除了offset和复制积压缓冲区，实现部分重同步还需要用到服务器的ID，如果重连接的ID不是之前的主节点，则执行完全重同步

### 2.5.6.pysnc

Psync命令是实现完全重同步和部分重同步，其调用方法有两种：

* 未进行任何复制，则发送psync ？-1命令，请求完全重同步
* 已经复制过，发送psync <runid> <offset>：runid是上次的主节点ID，offset是复制偏移量，主节点通过这两个参数决定执行完全还是部分重同步

## 2.6.复制的实现

* 步骤1：设置主节点地址和端口，client向从节点发送slaveOf命令，从节点向client返回OK，表示主从同步指令收到
* 步骤2：建立套接字，slaveOF命令后，从节点根据命令与主节点建立连接
* 步骤3：发送ping命令，从节点成为主节点的客户端后，首先发送ping命令，检查连接状态
* 步骤4：身份验证，从节点在发送ping后会收到主节点的pong回复，将进行身份验证（密码）
* 步骤5：从节点向主节点发送服务器的监听端口后
* 步骤6：同步，从节点发送psync命令
* 步骤7：命令传播

## 2.7.哨兵模式

哨兵机制主要是为了解决主从复制的缺点，当主节点故障后，哨兵自动完成故障发现和转移，并通知应用方，实现高可用。

* 监控master和slave是否正常运行
* Master出现故障时自动将slave数据库升级为master
* 哨兵不仅监听master和slave，还会相互监督，这种方式称为哨兵集群

### 2.7.1.哨兵集群相互感知

哨兵节点之间因为共同监督同一个master而产生关联，一个新加入的哨兵节点需要和其他监视相同master的节点相互感知和通信。

* 需要相互感知的哨兵都向他们共同监督的master节点订阅channel：sentinel：hello
* 新加入的哨兵节点向这个channel发布一条消息，包含自身的信息，这样订阅这个channel的哨兵就可以发现这个新的哨兵
* 新加入的哨兵和其他哨兵建立长连接

### 2.7.1.哨兵定时任务

* 任务1：每隔10s每个哨兵节点会向主节点和从节点发送“info replication”命令来获取最新的拓扑结构
* 任务2：每隔2s每个哨兵节点会向redis节点的某个频道发送自己对主节点是否故障的判断以及自身的节点信息，并且其他的哨兵节点也会订阅这个频道来了解其他哨兵节点的信息以及对主节点的判断
* 任务3：每1s每个哨兵会向主从节点以及哨兵节点发送一个ping命令来做心跳检测

### 2.7.2.主观下线和客观下线

任务3中，每1s向主从节点以及哨兵节点发送ping命令做心跳检测，如果这些被监控的节点在配置的时间内没有pong回复，则哨兵会部主管认为这个节点故障不可导。

哨兵判断主观下线的节点为主节点的时候，就会通过命令向其他哨兵节点询问它们对于主节点的判断，当主观下线票数超过多数派，这个这节点可能真的故障，这时就由主观下线变为客观下线。

### 2.7.3.故障转移哨兵leader选举

如果主节点被判定为客观下线后，则要选举一个leader哨兵完成后面的故障转移，选举算法为raft。

### 2.7.4.故障转移

首先要选择一个新的从节点作为主节点，选择的流程如下：

1. 选择slave-priority最高的节点
2. 选择复制偏移量最大的节点，复制偏移量越大越完整
3. 选择run\_id最小的节点

故障转移的流程如下（选出slave1作为主节点）：

* 将slave1脱离从节点身份，升级为主节点
* 向其他从节点发送slaveof命令，告知新的主节点
* 通知客户端主节点已换
* 将原来的主节点降为从节点，指向新的主节点

## 2.8.redis常见的性能问题和解决办法

* Master写内存快照，save命令会阻塞主线程的工作，master最好不要写内存快照
* Master AOF持久化，如果不重写AOF文件，但是AOF文件会不断增大，会影响master重启的恢复速度
* 主从复制的性能问题，为了速度和连接的稳定性，slave和master最好在同一个局域网内

## 2.9.redis应用场景

### 2.9.1.redis实现消息队列

Redis提供轻量级的消息队列功能，与专业的消息队列产品相比，存在如下问题：

* 可消费性：没有相应的机制保证消息的消费，当消费失败后，消息丢失，需手动处理
* 可靠发布：redis不提供，需自行实现
* 高可用：主从复制，读写分离
* 持久化：redis支持将实例持久化到磁盘
* 消费者负载均衡：redis不提供，需要自行实现
* 队列监控：redis不提供
* Redis适合于轻量级、高并发、延迟敏感、即时数据分析、秒杀计数器、缓存
* 使用list结构作为队列，rpush生产消息，lpop消费消息，当无消息的时候，可以sleep一会重试
* 当没有消息的时候，可以通过blpop指令，实现阻塞等待消息的到来
* Rpush（reverse）从尾部插入，lpush从头部插入
* 一次生产多次消费：使用pub/sub主题订阅者模式，实现1：N的消息队列
* Pub/sub模式在消费者下线后，生产的消息会丢失，可以使用专业的消息队列kafka
* 如何实现延时队列：使用sortedset，拿时间戳作为score，消息内容作为key调用zadd来生产消息，消费者用zrangebyscore指令获取N秒之前的数据轮询进行处理

### 2.9.2.redis实现发布订阅模式

Redis还提供了一组命令支持发布订阅模式：

* Publish channel message：发布消息，返回值为接收到该消息的订阅者数量
* Subscribe channel：订阅，返回值包括client订阅的频道，目前已订阅频道数量，以及接收到的消息
* Unsubscribe：取消订阅
* 模式匹配：允许client订阅符合某个模式的频道
* 如果发送的消息订阅者没有接受，消息就会丢失

实现原理如下：

每个redis服务器都维护一个表示服务器状态的redis.h/redisServer结构，

Struct RedisServer

{ dict \*pubsub\_channels；

List \*pubsub\_patterns；

}

Struct pubsubPattern

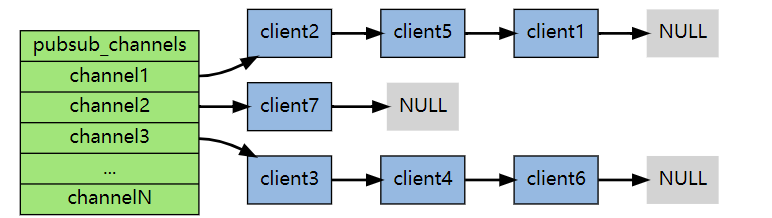
{

RedisClient \*client；

Robj \*pattern；

}

* Dict内部有两张哈希表，一张用作存储数据，一张用作扩容使用
* 哈希表为数组加链表形式
* <key,value>中key为client订阅的频道channel，value为链表，即订阅该频道的client
* Redis支持订阅模式，pubsub\_pattern为一个链表，链表中保存着所有和模式相关的信息，每个节点保存着client和其订阅的模式



* 据上图逻辑，subscribe其实是将client添加到对应订阅频道的链表中，或是将订阅的模式加入到订阅模式链表中
* Publish其实是将发布的消息遍历发送给该channel的所有client，同时遍历所有订阅模式，如果匹配则发送消息

### 2.9.3.redis实现延时队列

延时队列就是一种带有延迟功能的消息队列，

* Java.util.concurrent.DelayQueue：不支持分布式运行和消息持久化
* Rocketmq：消息持久化、分布式，不支持任务时间精度
* Rabbitmq：消息持久化、分布式，延时相同消息在同一队列

#### 2.9.3.1.功能特性

* 消息可靠性、持久化，至少被消费一次
* 实时性：存在一定时间误差
* 支持消息的remove
* 高可用

#### 2.9.3.2.redis延时队列

Redis利用sortedset（zset）实现延时队列，zset除了添加数据的value之外，还提供另一属性score，zset按照score自动排序，score为执行时间的时间戳，在某个时间将其插入zset中，zset按照时间戳大小排序，while循环不断取zset的第一个key值，比较其执行时间戳与当前时间的大小决定是否进行消费。

## 2.10.redis线程模型

### 2.10.1.redis单线程模型

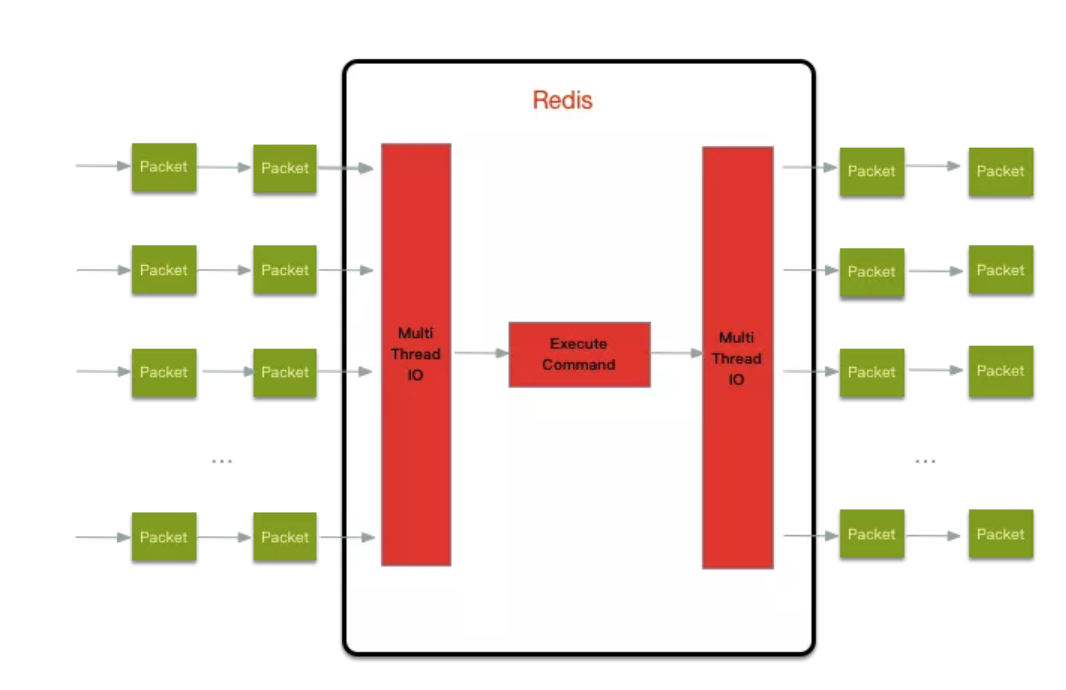
单线程是指redis的网络请求模块使用了一个线程，即一个线程处理所有网络请求，其他模块（RDB模块多线程或多进程）。

Redis单线程模型快速的原因：

* 绝大部分请求是纯内存操作
* 单线程避免了不必要的上下文切换和竞争
* 采用IO多路复用
* Redis的数据结构简单，操作简单
* 单线程不需要考虑各种锁问题，不存在加锁释放锁的操作，没有死锁

### 2.10.2.redis多线程模型

Redis的多线程模型与memcached从IO处理到数据访问的多线程实现模式有差异，redis的多线程部分只是用来处理网络数据的读写和协议解析，执行命令仍然是单线程，之所以这么设计是不想因为多线程而变得复杂，需要去控制key、lua、事务等并发问题



## 2.11.redis事务实现

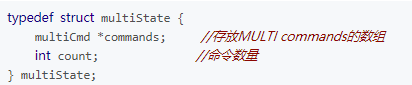
Redis事务通常会使用multi、exec、watch等命令完成，redis实现事务的机制与常见的关系型数据库由很大的区别：

* Redis事务不支持回滚
* Redis事务执行会阻塞其他事务（单线程）

事务的过程如下：

* 事务开始---multi
* 命令入队---Queue
* 事务执行---exec

Redis客户端有自己的事务状态：



Redis事务也具有ACID特性：

* 原子性：redis事务中的操作一方面要么都执行，要么都不执行。另一方面事务中的操作执行失败后面的命令还是会执行，错误之前的命令不会回滚。Redis不支持回滚操作
* 一致性：数据库在执行事务前后是一致的
* 隔离性：多个事务并发执行，各个事务不会相互影响，因为redis采用单线程方式执行事务，所以事务是串行执行的，所以满足隔离性
* 持久性：RDB的bgsave操作是异步执行，数据不会被第一时间保存到磁盘，所以不具备持久化特性，AOF模式下appendfsync值为always，程序总会在命令执行之后调用同步函数sync将数据保存到磁盘，所以具备持久性

# 3.缓存问题

## 3.1.缓存穿透

当查询缓存时没有此key对应的值，然后去数据库查询，数据有key对应的值会返回此值，没有就返回没有此值，如果大量访问缓存和数据库都没有的值会造成较大的压力。

解决方案如下：

* 缓冲空值的方式，如果数据库查询的对象为空，也放入缓存，并设置缓存过期的时间。
* 布隆过滤，不存在的数据一定会被过滤

## 3.2.缓存雪崩

大批量的缓存同时过期或是缓存服务器重启，导致读取压力落到数据库上。

解决方案如下：

* 将缓存的key的到期时间设置为不同时间，避免同一个时间段大规模的缓存失效
* 将缓存备份
* 设置热点数据永不过期

## 3.3.缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据，在某一时刻，缓存中某一key值缓存失效，对于此key值的大量请求会像击穿缓存一样落到数据库上，造成数据库的瞬时压力过大。

解决方案

* 设置热点数据永不过期
* 对数据加锁，第一次请求缓存中不存在，先加锁，然后去数据库查询并加载到缓存中