Redis

概述

- Redis 是速度非常快的非关系型（NoSQL）内存键值数据库，可以存储键和五种不同类型的值之间的映射。
- 键的类型只能为字符串，值支持五种数据类型：字符串、列表、集合、散列表、有序集合。
- Redis 支持很多特性，例如将内存中的数据持久化到硬盘中，使用复制来扩展读性能，使用分片来扩展写性能。
- 只要作为缓存，存储经常访问的数据或者经常更改的数据。

数据类型

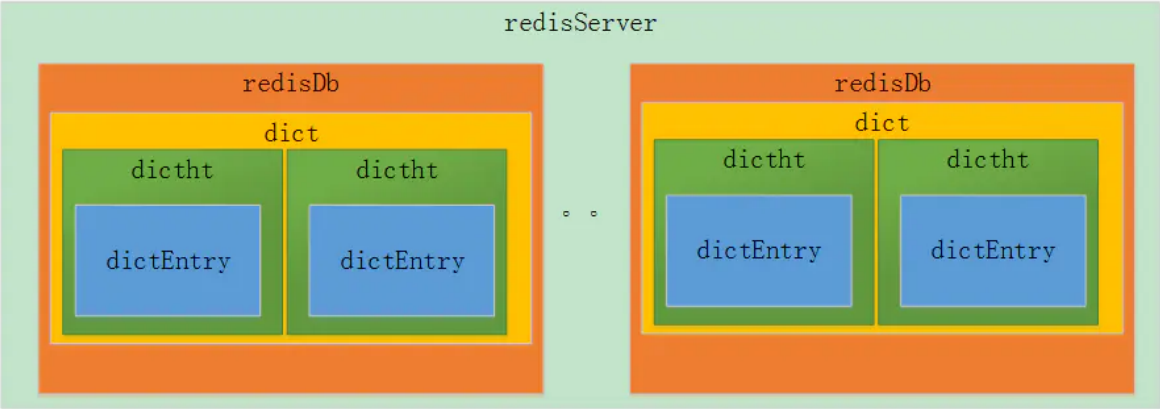
数据类型	存储值	操作	应用
STRING	字符串、整数或者浮点数	字符串或字符串一部分执行操、整数或者浮点数执行自增或者自减操作(set,get,decr,incr,mget)	常规计数、微博数、粉丝数等
LIST	列表	两端压入或者弹出元素、对单个或者多个元素进行修剪、保留以一个范围内的元素(lpush,rpush,lpop,rpop,lrange)	双向链表、关注列表、粉丝列表、消息列；lrange可以实现分页查询（微博下拉分页）

数据类型	存储值	操作	应用
SET	无序集合	添加、获取、移除单个元素、检查一个元素是否存在于集、计算交集、并集、差集、从集合里面随机获取元素。去重（sadd,spop,smembers,sunion）	共同关注、共同粉丝、共同喜好等功能、list排重功能
HASH	键值对的无序散列表	添加、获取、移除单个键值对、获取所有键值对检查某个键是否存在（hget,hset,hgetall）	适合存储对象、用户信息、商品信息（购物车）；后期可以仅仅修改对象中某个字段的值
ZSET	有序集合	添加、获取、删除元素、根据分值范围或者成员来获取、计算一个键的排名（zadd,zrange,zrem,zcard）增加了一个权重参数score，可以依据其进行有序排列	实时排行、礼物排行榜、在线用户列表

Redis存储结构

字典

- redis的存储结构从外层往内层依次是redisDb、dict、dictht、dictEntry。
- redis的Db默认情况下有15个，每个redisDb内部包含一个dict的数据结构。
- redis的dict内部包含dictht的数组，数组个数为2，主要用于hash扩容使用。
- dictht内部包含dictEntry的数组，可以理解就是hash的桶，然后如果冲突**通过挂链法解决**，冲突的时候将欣新节点添加到表头位置。



字典的数据结构

```
typedef struct dict {
    // 特定类型的函数，针对不同类型的键值对创建多态字典而设置的
    // 指向dictType结构的指针，每一个dictType结构保存了一组用于操作特定类型键值对的函数
    dictType *type;

    // 私有数据，保存了需要传给那些类型特定函数的可选参数
    void *privdata;

```

```
    // 哈希表,有两个哈希表，进行扩容的时候使用
    dictht ht[2];

    // rehash 索引，当rehash不在进行时，值为 -1
    long rehashidx; /* rehashing not in progress if rehashidx == -1 */
    unsigned long iterators; /* number of iterators currently running */
} dict;

```

```
/* This is our hash table structure. Every dictionary has two of this as we
 * implement incremental rehashing, for the old to the new table.
 * 使用渐进的 rehash 操作，将旧的键值对 rehash 到 另一个 dictht 上面
 */
typedef struct dictht {
    dictEntry **table; // 哈希表数组，数组中没用元素有指向一个 dictEntry结构的指针
    unsigned long size; //哈希表大小
    unsigned long sizemask; // 哈希表大小掩码，用于计算索引值 等于 size - 1
    unsigned long used; // 哈希表已经拥有的结点数量
} dictht;

```

```
typedef struct dictEntry {
    // 键
    void *key;
    // 值
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    // 指向下一个哈希表结点，形成链表，用来解决冲突问题
    struct dictEntry *next;
} dictEntry;

```

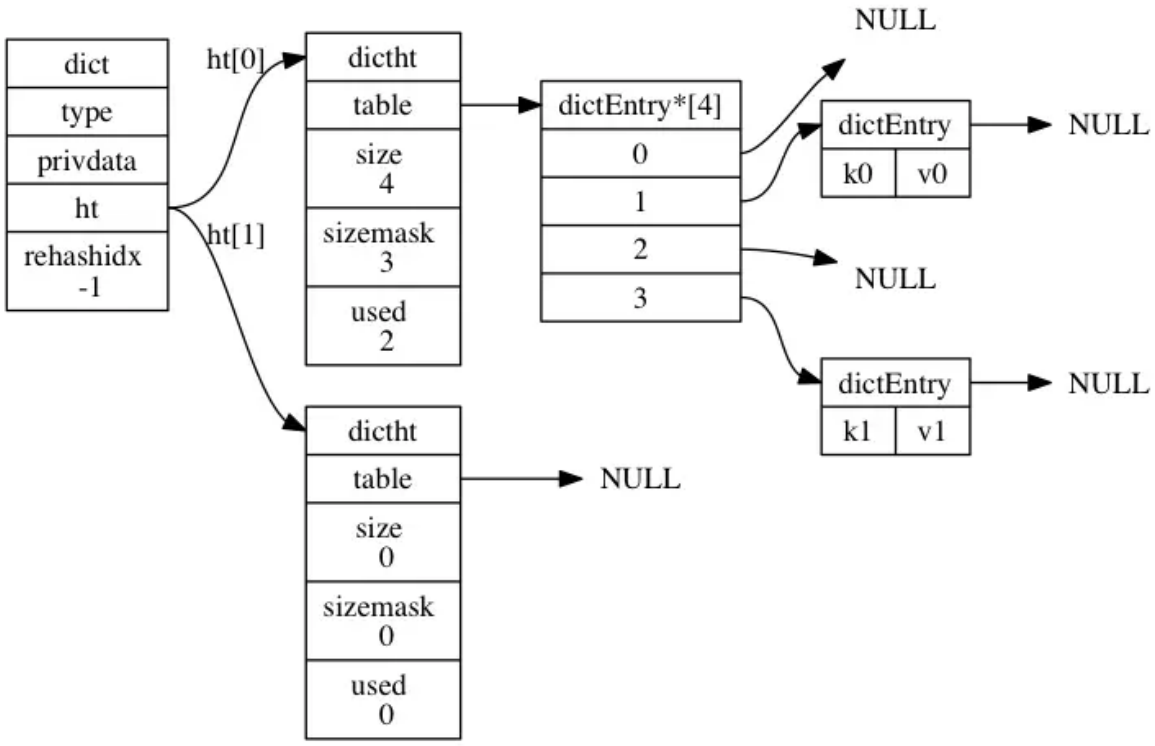


图 4-3　普通状态下的字典

哈希算法

```
// 使用字典设置的哈希函数，计算键 key 的哈希值
hash = dict->type->hashFunction(key);

// 使用哈希表的 sizemask 属性和哈希值，计算出索引值
// 根据情况不同， ht[x] 可以是 ht[0] 或者 ht[1]
index = hash & dict->ht[x].sizemask;
```

rehash

哈希表的键值对会增加或减少，为了让哈希表负载因子维持在一个合理的范围之内，当哈希表保存的键值对太多或者太少时，程序要对哈希表的大小进行相应的扩展或者收缩。

1. 为字典的ht[1]哈希表分配空间，这个空间大小取决于要执行的操作：
   如果执行的是**扩展操作**，则ht[1]的大小为第一个大于等**ht[0].used*2的2^n**；
   如果执行的**收缩操作**，则ht[1]的大小为第一个大于等于**ht[0].used的2^n**；
2. 将保存在ht[0]中的所有键值对rehash到ht[1]上面：rehash指的是重新计算键的哈希值和索引值，然后将键值对放置到ht[1]的指定位置上。
3. 当ht[0]包含的所有键值对都迁移到ht[1]之后，释放ht[0]，将ht[1]设置为ht[0]，并在ht[1]新创建一个空白哈希表，为下一次rehash做准备 （交换角色）

渐进式的 rehash

rehash是分多次，渐进式的完成的。当服务器包含很多键值对，要一次性的将这些键值对全部rehash到ht[1] 中，庞大的操作量会加重服务器的负担

1. 让字典同时拥有ht[0] 和 ht[1] 两个哈希表
2. 在字典中维持一个索引变量 rehashidx，并将他设置为0，表示rehash工作的开始
3. 每执行一次 rehash rehashidx都会递增。例如在一次 rehash 中，要把 dict[0] rehash 到 dict[1]，这一次会把 dict[0] 上 table[rehashidx] 的键值对 rehash 到 dict[1] 上，dict[0] 的

table[rehashidx] 指向 null，并令 rehashidx++

4. 当ht[0] 的所有键值对都rehash完毕，rehashidx = -1，表示完成

好处就是采取分而治之的方式，将 rehash 的键值对所需要的计算工作均摊到字典的每个添加、删除、查找和更新操作上，从而避免了集中式的rehash。

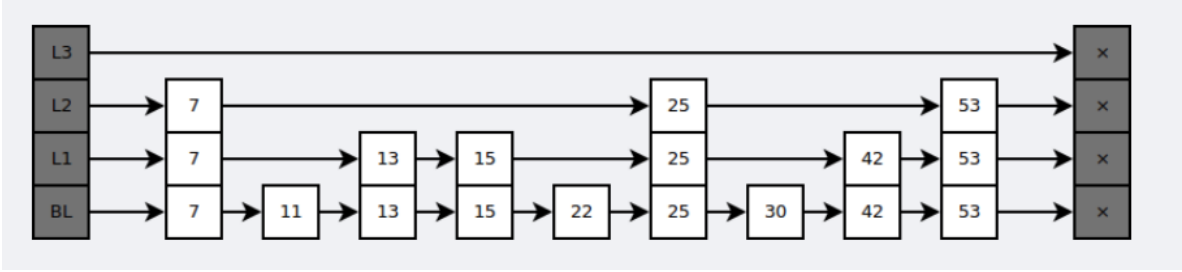
渐进式rehash执行期间的哈希表操作

因为在渐进式rehash的过程中，字典会同时使用ht[0]和ht[1]两个哈希表，所以在渐进式rehash进行期间，字典的删除、查找、更新等操作都是在两个表上进行的。

例如，**查找**操作会先在ht[0]上进行，如果没找到再在ht[1]上进行。**添加**操作的键值对会一律保存到ht[1]中，这一措施保证ht[0]包含的键值对只会减少不会增加。

跳表

跳表就是一个多级索引的链表，将链表每一层抽取结点形成索引。Redis 跳表也是 zset（有序集合）的实现方式。在查找时，从上层指针开始查找，找到对应的区间之后再到下一层去查找。



几种数据集合查询的比较

数据结构	实现原理	key查询方式	查找效率	存储大小	插入、删除效率
Hash	哈希表	支持单key	接近O(1)	小，除了数据没有额外的存储	O(1)
B+树	平衡二叉树扩展而来	单key，范围，分页	O(Log(n))	除了数据，还多了左右指针，以及叶子节点指针	O(Log(n)，需要调整树的结构，算法比较复杂
跳表	有序链表扩展而来	单key，分页	O(Log(n))	除了数据，还多了指针，但是每个节点的指针小于<2,所以比B+树占用空间小	O(Log(n)，只用处理链表，算法比较简单

关于跳表的原理可以参考这篇文章：跳表原理解析

为什么 Redis 使用跳表？不使用B+索引

B+树叶子节点存储数据，非叶子节点存储索引，查询数据时首先要根据索引查询到叶子节点，再到叶子节点所指向的地址去磁盘读取数据；这涉及到 I /O 操作。

B+树索引的原理

因为B+树的原理是 叶子节点存储数据，非叶子节点存储索引，B+树的每个节点可以存储多个关键字，它将节点大小设置为磁盘页的大小，充分利用了磁盘预读的功能。每次读取磁盘页时就会读取一整个节点,每个叶子节点还有指向前后节点的指针，为的是最大限度的降低磁盘的IO;因为数据在内存中读取耗费的时间是从磁盘的IO读取的百万分之一

跳表的优点

而Redis是基于内存的不需要I/O操作，并且跳表与B+树相比有很少的内存占用，B+树有2个以上的指针，而跳表的指针数平均为1/(1-p)(p为结点具有的指针概率，如Redis为 1/4)；在删除和修改操作上跳表只需修改相邻结点的指针，而B树要分裂或合并结点来调整树；且跳表具有较简单的实现（基于链表）

为什么使用Redis作为缓存系统

- **高性能**：当做用户和数据库之间的缓存，可以有效提高访问速度。用户从数据库中读取数据是从硬盘上读取数据，而从缓存中读取数据就是从内存中读取数据，这样速度更快。
- **高并发**：当多个用户并发的修改或者访问数据库时，尤其是访问数据库数据库为了保持数据的一致性会加锁，这会严重影响数据。

Redis过期时间设置

Redis中有个设置时间过期的功能，即对存储在 redis 数据库中的值可以设置一个过期时间。作为一个缓存数据库，这是非常实用的。如我们一般项目中的 token 或者一些登录信息，尤其是短信验证码都是有时间限制的，按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性能。

我们 set key 的时候，都可以给一个 expire time，就是过期时间，通过过期时间我们可以指定这个 key 可以存活的时间。

如何删除过期的Key?

- **定期删除**：每隔100ms **随机抽取** 一些过期时间的key，检查是否删除，若过期则删除。（如果数据量过大遍历过期的key 会带来性能损耗）。这种方法能够及时释放内存，但是大规模的删除会消耗CPU资源
- **惰性删除**：当程序读写一个 key 时，来判断这个key是否过期，如果过期再将其删除。这样能够避免CPU在固定时间去排查过期的key，但是内存存在大量过期的key时，会造成内存空间的浪费。
- **按规则删除**：结合两种策略结合起来。Redis 中懒删除是内置策略，可以对定时删除设置执行时间和频率。（hz 选项：设置定期频率，每秒执行多少次，越大CPU消耗越大；最大内存：超过最大内存触发一个清除策略。

Redis内存淘汰策略

当客户端会发起需要更多内存的申请的时候，Redis检查内存使用情况，如果实际使用内存已经超出maxmemory，Redis就会根据用户配置的淘汰策略选出无用的key;

- **LRU淘汰**

LRU(Least recently used，最近最少使用)算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”

  - **dictGetRandomKeys**随机获取**指定数目**的dictEntry，默认选择5个键。
  - 将获取的的dictEntry进行下**sort按照最近时间**进行排序。
  - 选择最近使用时间最久远的数据进行过期
  - 每次过期的数据其实是**采样的结果数据**中的最近未被访问数据而非全局的。
- **TTL淘汰**
  - Redis 数据结构中保存了键值对过期时间的表，即 redisDb.expires。和 LRU 数据淘汰机制类似，TTL 数据淘汰机制是这样的：从过期时间的表中随机挑选几个键值对，取出其中 TTL (剩余过期时间）最大的键值对（将要过期的）淘汰。同样你会发现，**redis 并不是保证取得所有过期时间的表中最快过期的键值对，而只是随机挑选的几个键值对**
- **随机淘汰**
  - 在随机淘汰的场景下获取待删除的键值对，随机找hash桶再次hash指定位置的dictEntry

配置文件中的淘汰策略

1. **volatile-lru**：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
2. **volatile-ttl**：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. **volatile-random**：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
4. **allkeys-lru**：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key（这个是最常用的）
5. **allkeys-random**：从数据集（server.db[i].dict）中任意选择数据淘汰
6. **no-eviction**：禁止驱逐数据，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。可以保证数据不被丢失

4.0版本后增加以下两种：

1. **volatile-lfu**：从已设置过期时间的数据集(server.db[i].expires)中挑选最不经常使用的数据淘汰
2. **allkeys-lfu**：当内存不足以容纳新写入数据时，在键空间中，移除最不经常使用的key

如何选择?

- **LRU 和 Random**：知道某些数据的访问频率较高或者较低，以及无法预测数据的访问频率时，可使用 lru，访问概率相等时 可以使用random
- **TTL**：如果研发者需要通过设置不同的ttl来判断数据过期的先后顺序，可采用ttl
- **allkeys 和 volatie**：知道或确定一些数据能够淘汰掉时或者希望某些确定数据能长期被保存，可以设值过期时间，淘汰机制就可以采用 volatie。设置expire会消耗额外的内存，如果计划避免Redis内存在此项上的浪费，或者不知到哪些确切的数据时要保存或者淘汰。可以选用allkeys

Redis 持久化

持久化就是将内存写入到硬盘上，为了设备故障或者重启机器而恢复数据和数据备份

快照持久化（RDB）

- 可以通过设置快照来保存某个时间点上的数据副本，快照创立后可以使用快照进行备份，可以复制到其他服务器上从而创建其他服务器的副本（对于主从结构的Redis集群来说）

- 也可以将快照留在原地以重启服务器的时候使用
- 默认持久化方式

```
save 900 1           #在900秒(15分钟)之后，如果至少有1个key发生变化，Redis就会自动触发BGSAVE命令创建快照。
save 300 10          #在300秒(5分钟)之后，如果至少有10个key发生变化，Redis就会自动触发BGSAVE命令创建快照。
save 60 10000        #在60秒(1分钟)之后，如果至少有10000个key发生变化，Redis就会自动触发BGSAVE命令创建快照。
```

AOF（append-only file）持久化

- 实时性更好，每执行一定条数的数据命令，Redis就会将该命令写入到硬盘中的AOF文件中
- 存储在 `appendonly.aof` 文件中和rdb文件一样的位置
- AOF工作流程操作：命令写入 （append）、文件同步（sync）、文件重写（rewrite）、重启加载 （load）
  - 将所有写入命令追加到 `aof_buf` **缓冲区**当中
  - 缓冲区根据对应的策略向硬盘做同步操作
- AOF文件越来越大则需要对AOF进行重写，达到压缩的目的

  为什么重写？旧的AOF文件包含很多无效命令，重写使用进程内数据直接生成，新的AOF只确保最终数据的写入命令。多条写命令可以合并为1个，降低占用空间；（手动触发 调用bgrewriteaof命令、自动触发）
  - 服务器重启可以加载进行数据恢复
- 三种配置方式：
  - `appendfsync always`：每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度
  - `appendfsync everysec`:每秒钟同步一次，显示地将多个写命令同步到硬盘
  - `appendfsync no`:让操作系统决定何时进行同步

4.0 持久机制的优化

混合模式，AOF重写的时候直接把RDB的内容直接写到AOF文件头。可以结合 RDB 和 AOF 的优点, 快速加载同时避免丢失过多的数据。缺点是 AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式，可读性较差。

Redis 事务

可以一次执行多个命令，本质是一组命令的集合。一个事务中的所有命令都会序列化，按顺序串行化的执行而不会被其他命令插入

一个队列中，一次性、顺序性、排他性的执行一系列命令

Redis 事务的三个阶段

- 开启：以MULTI 开启一个事务
- 入队：将多个命令入队到事务中，接到这些命令不会立即执行，而是放到等待执行的事务队列里面
- 执行：由EXEC命令触发事务

Redis 事务的三个特征

- 单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- 没有隔离级别的概念：队列中的命令没有提交之前都不会实际的被执行，因为事务提交前任何指令都不会被实际执行，也就不存在”事务内的查询要看到事务里的更新，在事务外查询不能看到”这个让人万分头痛的问题
- 不保证原子性：redis同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

Watch 指令

watch指令类似于乐观锁，在事务提交时，如果watch监控的多个KEY中任何KEY的值已经被其他客户端更改，则使用EXEC执行事务时，事务队列将不会被执行，同时返回Nullmulti-bulk应答以通知调用者事务执行失败。

缓存雪崩和缓存穿透

一般的数据处理流程：

用户发出请求，如请求数据经过缓存处理的话，一般先从缓存中取数据，若存在直接返回结果；若不存在，则从数据库获取并更新缓存，返回结果。若数据库和缓存都没有则返回空值。

缓存雪崩：

缓存在同一时间大面积实效（如数据大批量达到过期时间），后面的请求都会落在数据库上，造成数据库在短时间内承受不了大量的请求而崩掉。

解决方案：

- 缓存数据的过期时间设置为随机，防止同一时间大量数据过期的现象发生
- 缓存数据库是分布式存储，将热点数据均匀分布在不同服务器上
- 设置热点数据永远不过期

缓存穿透：

缓存穿透说简单点就是大量请求的 key 根本不存在于缓存中，导致请求直接到了数据库上，根本没有经过缓存这一层。如黑客存中不存在的 key 发起大量请求，导致大量请求落到数据库。

解决方案

- 接口层增加参数校验
- 设置无效缓存key：数据库没有数据时，写一个空的key的到缓存中： key - null。针对经常变化的请求key，可以设置无效的key的缓存时间短一点避免存内存中存在大量的无效key
- 布隆过滤器增加在用户和缓存之间，存放所有请求的合法值。用户请求首先会判断请求的值是否存在布隆过滤器中，不存在直接返回给客户端。

缓存击穿：

缓存中没有数据但是数据库中有数据，这个时候出现大量并发用户读缓存读不到，去数据库中读数据造成数据库压力瞬间增大。

解决方案：

- 互斥锁，若缓存中无数据，对数据库的读取数据加锁
- 设置数据不过期

Redis 如何保证与数就可读写一致

可以偶尔不一致。如果必须要保持一致：读写串行化，读请求和写请求串行化，保存到一个内存队列中

Cache Aside Pattern

- 读的时候先读缓存，如果缓存没有读数据库，读出数据放入缓存，同时返回相应
- 更新的时候，先更新数据库，再删除缓存

读写问题分析

为什么是删除缓存，而不是更新缓存?

（1）**线程安全角度：**如果有请求A和请求B进行更新操作：线程A更新数据库，线程B更新数据库，线程B更新缓存，线程A更新缓存。按道理来说应该A先更新缓存，但是B却更新了缓存，这就导致了脏数据。

（2）**业务场景角度：**

- 写场景多，读场景少的业务需求。这种方法导致频繁更新数据
- 写入数据库的值，不是直接写入缓存的，而是经过一系列负责的计算再写入缓存。

比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据并进行运算，才能计算出缓存最新的值的。

频繁更新的缓存需要查询其他表才能做计算，更新的代价大，访问频率不高，存在大量冷数据。若删除缓存的话只需要用缓存时才会算缓存，并不需要依赖于对应的数据库频繁更新而增加开销

先删除缓存再更新数据库?

缓存不一致分析：先删掉缓存的话，同时一个请求更新操作，另一个请求查询操作

问题：数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，**查到了修改前的旧数据，放到了缓存中。**随后数据变更的程序完成了数据库的修改。数据库和缓存中的数据不一样了。

解决思路：**延时双删策略**：先删除缓存、再修改数据库，休眠1s再淘汰缓存。可以将 1s内造成的脏数据再次删除。（1s如何确定：确保读请求结束后，写请求可以删除读请求造成的缓存脏数据）但是读请求读的还是脏数据。

如果你用了mysql的读写分离架构怎么办?

两个请求，一个请求A进行更新操作，另一个请求B进行查询操作。

（1）请求A进行写操作，删除缓存
（2）请求A将数据写入数据库了，
（3）请求B查询缓存发现，缓存没有值
（4）请求B去从库查询，这时，还没有完成主从同步，因此查询到的是旧值
（5）请求B将旧值写入缓存
（6）数据库完成主从同步，从库变为新值

上述情形，就是数据不一致的原因。**还是使用双删延时策略，只是，睡眠时间修改为在主从同步的延时时间基础上，加几百ms。**

采用这种同步淘汰策略，吞吐量降低怎么办?

将第二次删除作为异步的。自己起一个线程，异步删除。这样，写的请求就不用沉睡一段时间后了，再返回。这么做，加大吞吐量。

先更新数据库，再更新缓存

缓存不一致分析： 后删除缓存后删除缓存失败

问题：先更新数据库，再删除缓存，**删除缓存失败**？如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。

解决思路：重试删除方案

1. 更新缓存中的数据
2. 缓存删除失败
3. 将要删除的key发至消息队列
4. 消费自己的信息，获取到要删除的key
5. 继续重试删除的操作，直到成功

肯能造成业务代码的大量侵入，可以启动一个订阅程序去订阅数据库的binlog，获取需要操作的程序。

参考

- https://www.jianshu.com/p/bfecf4ccf28b
- https://blog.csdn.net/diweikang/article/details/94406186
- https://www.cnblogs.com/DeepInThought/p/10720132.html
- https://cyc2018.github.io/CS-Notes/#/notes/Redis