Mysql索引、Redis、Linux Anki

什么是索引?

索引是一种用于快速查询和检索数据的数据结构。常见的索引结构有: B 树， B+树和 Hash。

为什么要用索引?索引的优缺点分析

- **索引的优点**
  - **可以大大加快 数据的检索速度（大大减少的检索的数据量）,将随机IO变为顺序IO， 这也是创建索引的最主要的原因。毕竟大部分系统的读请求总是大于写请求的。** 另外，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
- **索引的缺点**
  - **创建索引和维护索引需要耗费许多时间**：当对表中的数据进行增删改的时候，如果数据有索引，那么索引也需要动态的修改，会降低 SQL 执行效率。
  - **占用物理存储空间** ： 索引需要使用物理文件存储，也会耗费一定空间。

B 树和 B+树区别

- B 树的所有节点既存放 键(key) 也存放 数据(data);而 B+树只有叶子节点存放 key 和 data，其他内节点只存放 key。
- B 树的叶子节点都是独立的;B+树的叶子节点有一条引用链指向与它相邻的叶子节点。
- B 树的检索的过程相当于对范围内的每个节点的关键字做二分查找，可能还没有到达叶子节点，检索就结束了。而 B+树的检索效率就很稳定了，任何查找都是从根节点到叶子节点的过程，叶子节点的顺序检索很明显。

Hash 索引和 B+树索引优劣分析

Hash索引

- hash索引**进行等值查询更快(一般情况下)但是却无法进行范围查询**.因为在hash索引中经过hash函数建立索引之后,索引的顺序与原顺序无法保持一致,不能支持范围查询.
- hash索引**不支持模糊查询以及多列索引的最左前缀匹配**,因为hash函数的不可预测,eg:AAAA和AAAAB的索引没有相关性.
- hash索引任何时候都**避免不了回表查询数据**.
- hash索引虽然在等值上查询叫快,但是不稳定,性能不可预测,**当某个键值存在大量重复的时候,发生hash碰撞,此时查询效率可能极差**.

B+树

- B+树的所有节点皆遵循(左节点小于父节点,右节点大于父节点,多叉树也类似)自然**支持范围查询**.
- **在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询.不需要回表查询**.
- 查询效率比较稳定,对于查询都是从根节点到叶子节点,且树的高度较低.

MySQL索引的类型

- **唯一索引**：唯一索引也是一种约束。**唯一索引的属性列不能出现重复的数据，但是允许数据为 NULL，一张表允许创建多个唯一索引。** 建立唯一索引的目的大部分时候都是为了该属性列的数据的唯一性，而不是为了查询效率
- **普通索引**：普通索引的唯一作用就是为了快速查询数据，一张表允许创建多个普通索引，并允许数据重复和 NULL
- **全文索引(Full Text)**：全文索引主要是为了检索大文本数据中的关键字的信息，是目前搜索引擎数据库使用的一种技术。Mysql5.6 之前只有 MYISAM 引擎支持全文索引，5.6 之后 InnoDB 也支持了全文索引
- **主键索引**：在 mysql 的 InnoDB 的表中，当没有显示的指定表的主键时，InnoDB 会自动先检查表中是否有唯一索引的字段，如果有，则选择该字段为默认的主键，否则 InnoDB 将会自动创建一个 6Byte 的自增主键

聚集索引与非聚集索引

聚集索引

- **聚集索引即索引结构和数据一起存放的索引。主键索引属于聚集索引**
- 聚集索引的查询速度非常的快，因为整个 B+树本身就是一颗多叉平衡树，叶子节点也都是有序的，定位到索引的节点，就相当于定位到了数据
- **依赖于有序的数据** ：因为 B+树是多路平衡树，如果索引的数据不是有序的，那么就需要在插入时排序，如果数据是整型还好，否则类似于字符串或 UUID 这种又长又难比较的数据，插入或查找的速度肯定比较慢
- 如果对索引列的数据被修改时，那么对应的索引也将会被修改， 而且况聚集索引的叶子节点还存放着数据，修改代价肯定是较大的， 所以对于主键索引来说，主键一般都是不可被修改的

非聚集索引

- **索引结构和数据分开存放的索引****，非聚集索引的叶子节点并不一定存放数据的指针， 因为二级索引的叶子节点就存放的是主键，根据主键再回表查数据
- **更新代价比聚集索引要小** 。非聚集索引的更新代价就没有聚集索引那么大了，非聚集索引的叶子节点是不存放数据的
- 跟聚集索引一样，非聚集索引也依赖于有序的数据
- **可能会二次查询(回表)** :这应该是非聚集索引最大的缺点了。 当查到索引对应的指针或主键后，可能还需要根据指针或主键再到数据文件或表中查询。

什么是覆盖索引

如果一个索引包含（或者说覆盖）所有需要查询的字段的值，我们就称之为“覆盖索引”。我们知道InnoDB存储引擎中，如果不是主键索引，叶子节点存储的是主键+列值。最终还是要“回表”，也就是要通过主键再查找一次。这样就会比较慢覆盖索引就是把要查询出的列和索引是对应的，不做回表操作！

覆盖索引使用实例：

现在我创建了索引(username,age)，我们执行下面的 sql 语句

```
select username , age from user where username = 'Java' and age = 22
```

在查询数据的时候：要查询出的列在叶子节点都存在！所以，就不用回表。

非聚集索引一定回表查询吗

如果一个索引包含（或者说覆盖）所有需要查询的字段的值，我们就称之为“覆盖索引”。我们知道在 InnoDB 存储引擎中，如果不是主键索引，叶子节点存储的是主键+列值。最终还是要“回表”，也就是要通过主键再查找一次。这样就会比较慢覆盖索引就是把要查询出的列和索引是对应的，不做回表操作！

覆盖索引即需要查询的字段正好是索引的字段，那么直接根据该索引，就可以查到数据了， 而无需回表查询。

最左前缀原则

MySQL中的索引可以以一定顺序引用多列，这种索引叫作联合索引。如User表的name和city加联合索引就是(name,city)，而最左前缀原则指的是，如果查询的时候查询条件精确匹配索引的左边连续一列或几列，则此列就可以被用到。如下：

```
select * from user where name=xx and city=xx ; / /可以命中索引
select * from user where name=xx ; // 可以命中索引
select * from user where city=xx ; // 无法命中索引
```

这里需要注意的是，查询的时候如果两个条件都用上了，但是顺序不同，如 `city= xx and name = xx`，那么现在的查询引擎会自动优化为匹配联合索引的顺序，这样是能够命中索引的。

由于最左前缀原则，在创建联合索引时，索引字段的顺序需要考虑字段值去重之后的个数，较多的放前面。ORDER BY子句也遵循此规则。

简单介绍一下 Redis

简单来说 **Redis 就是一个使用 C 语言开发的数据库**，不过与传统数据库不同的是 **Redis 的数据是存在内存中的** ，也就是它是内存数据库，所以读写速度非常快，因此 Redis 被广泛应用于缓存方向。

另外，**Redis 除了做缓存之外，Redis 也经常用来做分布式锁，甚至是消息队列。**

Redis 提供了多种数据类型来支持不同的业务场景。Redis 还支持事务 、持久化、Lua 脚本、多种集群方案。

为什么要用redis而不用Hashmap做缓存

- Redis 可以用几十 G 内存来做缓存，Map 不行，一般 JVM 也就分几个 G 数据就够大了
- Redis 的缓存可以持久化，Map 是内存对象，程序一重启数据就没了
- Redis 可以实现分布式的缓存，Map 只能存在创建它的程序里
- Redis 缓存有过期机制，Map 本身无此功能
- Redis 可以处理每秒百万级的并发，是专业的缓存服务，Map 只是一个普通的对象

Redis 常见数据结构

- **string**：string 数据结构是简单的 key-value 类型

普通字符串的基本操作：

```
127.0.0.1:6379> set key value #设置 key-value 类型的值
OK
127.0.0.1:6379> get key # 根据 key 获得对应的 value
"value"
127.0.0.1:6379> exists key  # 判断某个 key 是否存在
(integer) 1
127.0.0.1:6379> strlen key # 返回 key 所储存的字符串值的长度。
(integer) 5
127.0.0.1:6379> del key # 删除某个 key 对应的值
(integer) 1
127.0.0.1:6379> get key
(nil)
```

过期：

```
127.0.0.1:6379> expire key  60 # 数据在 60s 后过期
(integer) 1
127.0.0.1:6379> setex key 60 value # 数据在 60s 后过期 (setex:[set] + [ex]pire)
OK
127.0.0.1:6379> ttl key # 查看数据还有多久过期
(integer) 56
```

- **list**：Redis 的 list 的实现为一个 **双向链表**，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销
  - **应用场景:** 发布与订阅或者说消息队列、慢查询
    下面我们简单看看它的使用！

通过 `rpush/lpop` 实现队列：

```
127.0.0.1:6379> rpush myList value1 # 向 list 的头部（右边）添加元素
(integer) 1
127.0.0.1:6379> rpush myList value2 value3 # 向list的头部（最右边）添加多个元素
(integer) 3
127.0.0.1:6379> lpop myList # 将 list的尾部(最左边)元素取出
"value1"
127.0.0.1:6379> lrange myList 0 1 # 查看对应下标的list列表， 0 为 start,1为 end
1) "value2"
2) "value3"
127.0.0.1:6379> lrange myList 0 -1 # 查看列表中的所有元素，-1表示倒数第一
1) "value2"
2) "value3"
```

通过 `rpush/rpop` 实现栈：

```
127.0.0.1:6379> rpush myList2 value1 value2 value3
(integer) 3
127.0.0.1:6379> rpop myList2 # 将 list的头部(最右边)元素取出
"value3"
```

- **hash**：**hash 类似于 JDK1.8 前的 HashMap，内部实现也差不多(数组 + 链表)。不过，Redis 的 hash 做了更多优化。另外，hash 是一个 string 类型的 field 和 value 的映射表**，特别适合用于存储对象，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。 比如我们可以 hash 数据结构来存储用户信息，商品信息等等。

- **set**：set 类似于 Java 中的 `HashSet` 。**Redis 中的 set 类型是一种无序集合，集合中的元素没有先后顺序。当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择**，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。比如：你可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。R**edis 可以非常方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程。**
- **sorted set**：和 set 相比，**sorted set 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列**，还可以通过 score 的范围来获取元素的列表。有点像是 Java 中 HashMap 和 TreeSet 的结合体，适合需要对数据根据某个权重进行排序的场景。比如在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息
- **bitmap**：bitmap 存储的是连续的二进制数字（0 和 1），**通过 bitmap, 只需要一个 bit 位来表示某个元素对应的值或者状态**，key 就是对应元素本身 。我们知道 8 个 bit 可以组成一个 byte，所以 bitmap 本身会极大的节省储存空间， **适合需要保存状态信息（比如是否签到、是否登录...）并需要进一步对这些信息进行分析的场景。**比如用户签到情况、活跃用户情况、用户行为统计（比如是否点赞过某个视频）

    ```
    # SETBIT 会返回之前位的值（默认是 0）这里会生成 7 个位
    127.0.0.1:6379> setbit mykey 7 1
    (integer) 0
    127.0.0.1:6379> setbit mykey 7 0
    (integer) 1
    127.0.0.1:6379> getbit mykey 7
    (integer) 0
    127.0.0.1:6379> setbit mykey 6 1
    (integer) 0
    127.0.0.1:6379> setbit mykey 8 1
    (integer) 0
    # 通过 bitcount 统计被被设置为 1 的位的数量。
    127.0.0.1:6379> bitcount mykey
    (integer) 2
    ```

过期的数据的删除策略了解么

如果假设你设置了一批 key 只能存活 1 分钟，那么 1 分钟后，Redis 是怎么对这批 key 进行删除的呢?

常用的过期数据的删除策略就两个（重要！自己造缓存轮子的时候需要格外考虑的东西）：

1. **惰性删除**： 只会在取出 key 的时候才对数据进行过期检查。这样对 CPU 最友好，但是可能会造成太多过期 key 没有被删除。
2. **定期删除**： 每隔一段时间抽取一批 key 执行删除过期 key 操作。并且，Redis 底层会通过限制删除操作执行的时长和频率来减少删除操作对 CPU 时间的影响。

定期删除对内存更加友好，惰性删除对 CPU 更加友好。两者各有千秋，所以 Redis 采用的是 **定期删除+惰性/懒汉式删除** 。

但是，仅仅通过给 key 设置过期时间还是有问题的。因为还是可能存在定期删除和惰性删除漏掉了很多过期 key 的情况。这样就导致大量过期 key 堆积在内存里，然后就 Out of memory 了。

怎么解决这个问题呢？答案就是： **Redis 内存淘汰机制。**

Redis 内存淘汰机制了解么

Redis 提供 6 种数据淘汰策略：

1. **volatile-lru（least recently used）**：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（least recently used）**：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。这个应该没人使用吧！

Redis 持久化机制(怎么保证 Redis 挂掉之后再重启数据可以进行恢复)

很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据（比如重启机器、机器故障之后恢复数据），或者是为了防止系统故障而将数据备份到一个远程位置。

Redis 不同于 Memcached 的很重要一点就是，Redis 支持持久化，而且支持两种不同的持久化操作。**Redis 的一种持久化方式叫快照（snapshotting，RDB），另一种方式是只追加文件（append-only file, AOF）**。这两种方法各有千秋，下面我会详细这两种持久化方法是什么，怎么用，如何选择适合自己的持久化方法。

快照（snapshotting）持久化（RDB）

Redis 可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis 创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis 主从结构，主要用来提高 Redis 性能），还可以将快照留在原地以便重启服务器的时候使用。

快照持久化是 Redis 默认采用的持久化方式，在 Redis.conf 配置文件中默认有此下配置：

```
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

与快照持久化相比，AOF 持久化 的实时性更好，因此已成为主流的持久化方案。默认情况下 Redis 没有开启 AOF（append only file）方式的持久化，可以通过 appendonly 参数开启：

```
appendonly yes
```

开启 AOF 持久化后每执行一条会更改 Redis 中的数据的命令，Redis 就会将该命令写入硬盘中的 AOF 文件。AOF 文件的保存位置和 RDB 文件的位置相同，都是通过 dir 参数设置的，默认的文件名是 appendonly.aof。

在 Redis 的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

```
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
```

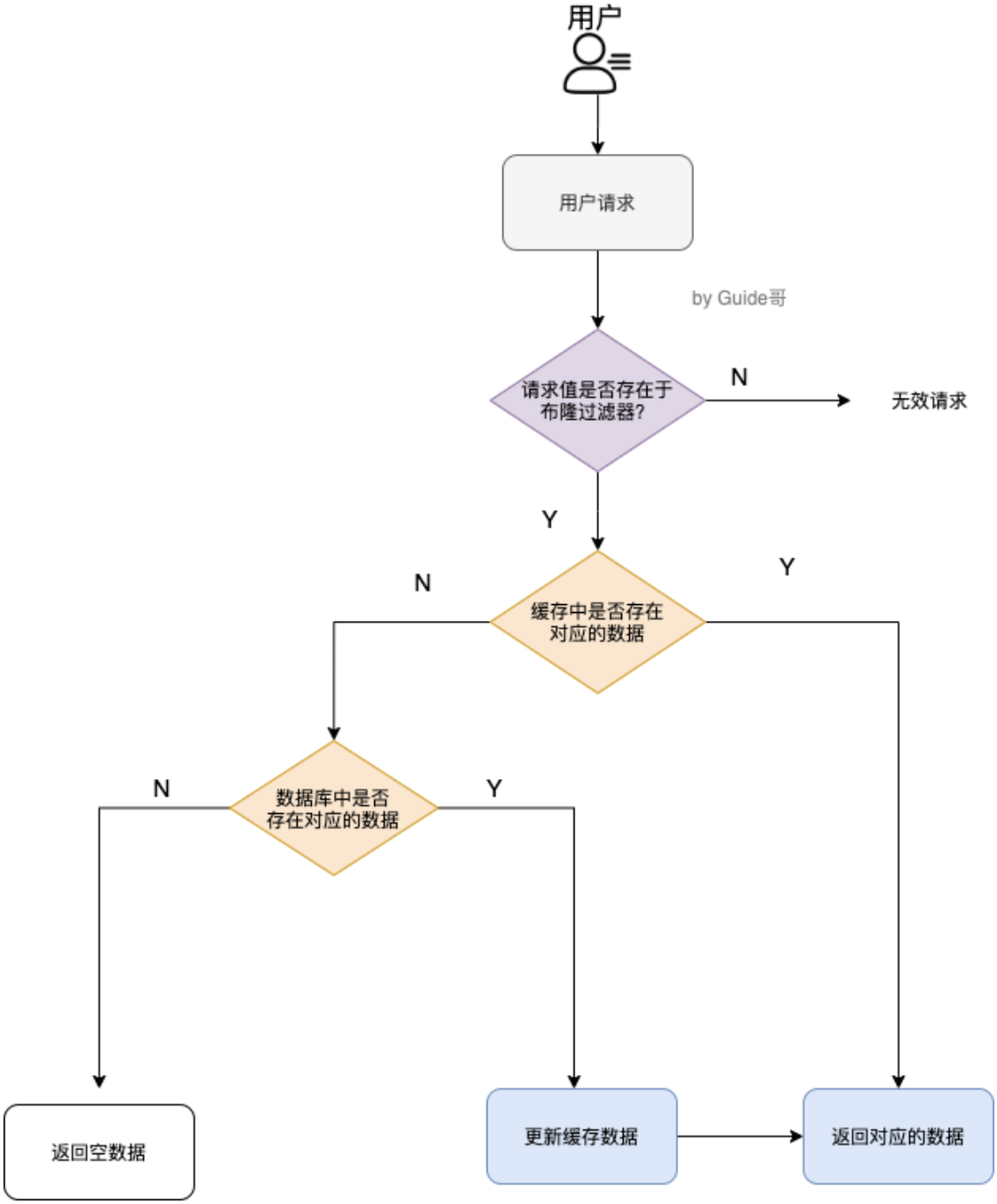
为了兼顾数据和写入性能，用户可以考虑 appendfsync everysec 选项 ，让 Redis 每秒同步一次 AOF 文件，Redis 性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis 还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

介绍下Redis的缓存穿透以及如何解决

- 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。**举个例子：某个黑客故意制造我们缓存中不存在的 key 发起大量请求，导致大量请求落到数据库。
- 解决办法：
  - **缓存无效 key**
  - **布隆过滤器**：布隆过滤器是一个非常神奇的数据结构，通过它我们可以非常方便地判断一个给定数据是否存在于海量数据中，把所有可能存在的请求的值都存放在布隆过滤器中，当用户请求过来，先判断用户发来的请求的值是否存在于布隆过滤器中。不存在的话，直接返回请求参数错误信息给客户端，存在的话才会走下面的流程

琐碎知识点

- Redis 给缓存数据设置过期时间有啥用？- 因为内存是有限的，如果缓存中的所有数据都是一直保存的话，很容易直接 Out of memory。
- **Redis 是如何判断数据是否过期的呢：**Redis 通过一个叫做过期字典（可以看作是 hash 表）来保存数据过期的时间。过期字典的键指向 Redis 数据库中的某个 key(键)，过期字典的值是一个 long long 类型的整数，这个整数保存了 key 所指向的数据库键的过期时间（毫秒精度的 UNIX 时间戳）。

- 

- 需要注意的是布隆过滤器可能会存在误判的情况。总结来说就是： **布隆过滤器说某个元素存在，小概率会误判。布隆过滤器说某个元素不在，那么这个元素一定不在。**

介绍下什么是缓存雪崩

- **缓存雪崩**：描述的就是这样一个简单的场景：**缓存在同一时间大面积的失效，后面的请求都直接落到了数据库上，造成数据库短时间内承受大量请求。** 这就好比雪崩一样，摧枯拉朽之势，数据库的压力可想而知，可能直接就被这么多请求弄宕机了。
- **解决办法**
  - **针对 Redis 服务不可用的情况：**
    - 采用 Redis 集群，避免单机出现问题整个缓存服务都没办法使用
    - 限流，避免同时处理大量的请求
  - **针对热点缓存失效的情况**：

- 设置不同的失效时间比如随机设置缓存的失效时间
- 缓存永不失效

如何保证缓存和数据库数据的一致性

聊聊**Cache Aside Pattern（旁路缓存模式）**

Cache Aside Pattern 中遇到写请求是这样的：更新 DB，然后直接删除 cache 。

如果更新数据库成功，而删除缓存这一步失败的情况的话，简单说两个解决方案：

1. **缓存失效时间变短（不推荐，治标不治本）** ：我们让缓存数据的过期时间变短，这样的话缓存就会从数据库中加载数据。另外，这种解决办法对于先操作缓存后操作数据库的场景不适用。
2. **增加 cache 更新重试机制（常用）**： 如果 cache 服务当前不可用导致缓存删除失败的话，我们就隔一段时间进行重试，重试次数可以自己定。如果多次重试还是失败的话，我们可以把当前更新失败的 key 存入队列中，等缓存服务可用之后，再将 缓存中对应的 key 删除即可。

(还有两个有时间再看)

Linux常用命令

- 进入目录：cd 目录名
- 显示当前路径：pwd
- 显示路径下的文件：ls
  - **-a** （显示隐藏文件，隐藏文件以 . 开头命名 ）
  - **-l** （以列表的形式显示文件详情）
- 查看创建文本：touch abc.txt （查看abc.txt 如果不存在则自动创建）
- 创建文件夹：
  - mkdir 文件名 当前目录创建一个文件夹
  - mkdir -p name1/name2 当期目录递归创建name1/name2文件
  - 如 提示 mkdir: xxx: Permission denied，则需要admin账号 sudo -i 输入密码 即可
- 删除操作：
  - rm file 删除file文件(存在子文件时不可删除)
  - rm -r /file 删除file文件下的所有目录文件
  - rm -rf ./* 删库跑路专用命令
- 复制：
  - cp file /home 复制file命令至home目录下
  - cp -r test /home/wechat 复制test文件夹和其所有子文件 至 /home/wechat目录下
    1
    2
    3
    9、
- 压缩、解压：
  - 解压tar
    tar xvf test.tar
  - 压缩tar
    tar cvf test1.tar name 将name文件夹压缩为test1.tar
  - 解压tar.gz
    tar zxvf test.tar.gz

- 压缩
  tar zxvf test.tar.gz name

- cat test.log　　查看test.log 的文件内容
  cat -n test.log　　查看test.log的文件内容并显示行号

- find命令

```
.代表当前目录
find . -name '*.txt'          查找当前目录及其子目录下扩展名为txt的文件
find . -mtime -2              列出两天内修改过的文件
find . -atime -3              列出三天内被存取的文件
find . -mmin +30              半个小时内被修改过的文件
find . -amin +40              四十分钟内被存取过的文件
find . -size +1M              查找当前目录超过1M的文件
find .  -size -1M         查找当前目录小于1M的文件
find .  -size   +512k     超过512k的文件
find . -empty             查找当前目录为空的文件或者文件夹
```

- whereis命令： whereis name/ **搜索name文件的路径**
  4、grep命令

- ps -ef|grep nginx　　查看nginx的进程
  ps -ef|grep nginx -c　　查看nginx的进程个数
  cat test.log | grep ^a　　查找test.log 中以o开头的内容
  cat test.log | grep $k　　查找test.log中以K结尾的内容
  cat test.log | grep 'bd4f63cc918611e8a14f7c04d0d7fdcc' --color　　在test.log中搜索bd4f63cc918611e8a14f7c04d0d7fdcc并高亮
  等同于 grep 'bd4f63cc918611e8a14f7c04d0d7fdcc' test.log --color

- tail命令
  - tail -f xxx.log　查看xxx.log 默认显示最后10行
  - tail -f 100 xx.log /tail -100f xx.log 查看100行
    1
    2

    6、vim

- vim命令

```
vim file 查看文本
vim file1 file2 ... 查看多个文本
正常模式/vim模式 通过ESC进行切换
vim模式下
i：在当前位置插入
dd： 删除光标所在行
D:删除光标所在行
2dd: 删除光标之后的2行
G：切换光标至末尾
w! 强制写入
wq 保存并退出
q! 强制退出 不保存
/abc  在文本中查找abc
set nu 显示文本行数
移动光标 k(上)、j(下)、h(左)、l(右)
yy 复制光标所在行
p粘贴复制的
o:另起一行
```

- **其他常用操作**

  1、查看用户信息

  ```
  w
  who
  ```

  2、修改文件权限

  ```
  chmod 777 file1          每个人都可以对file文件进行读写和执行的权限
  chmod 666 file1          每个人都可以对file文件进行读写操作
  ```

  - 3、系统级别

    ```
    top 实时显示系统资源使用情况
    dh -h 查看当前那磁盘使用情况
    du -sh /usr 计算usr文件大小

    netstat -a 列出 tcp, udp 和 unix 协议下所有套接字的所有连接

    kill 端口号 终止该端口
    kill -9 端口 立即强制终止端口
    rz lz 上传 和下载文件
    ```

参考

https://github.com/Snailclimb/JavaGuide/blob/master/docs/database/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B4%A2%E5%BC%95.md

https://blog.csdn.net/qq_44590469/article/details/97877397

https://blog.csdn.net/qq_32534441/article/details/97100545

https://blog.csdn.net/weixin_43499626/article/details/84864694