技术栈积累

# Java基础

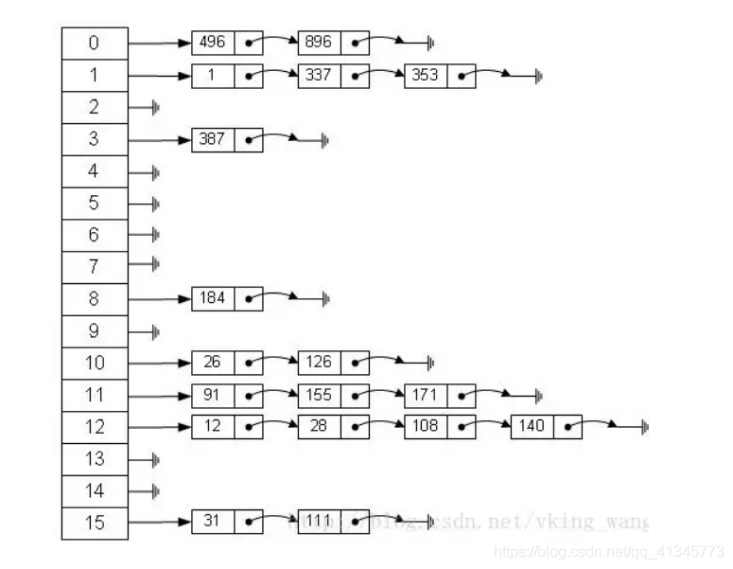
## JVM内存模型

## 1.2数据结构与算法（集合框架）

### 1.2.1 HashMap

**HashMap的工作原理**：通过put和get存储和获取对象。存储对象时，我们将K/V传给put方法时，它调用hashCode计算hash从而得到bucket位置，进一步存储，HashMap会根据当前bucket的占用情况自动调整容量(超过Load Facotr则resize为原来的2倍)。获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。如果发生碰撞的时候，Hashmap通过链表将产生碰撞冲突的元素组织起来，在Java 8中，如果一个bucket中碰撞冲突的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。

关键的信息：基于Map接口实现、允许null键/值、非同步、不保证有序(比如插入的顺序)、也不保证序不随时间变化。



Hashmap数据结构样例

**1. 两个重要的参数**

在HashMap中有两个很重要的参数，容量(Capacity)和负载因子(Load factor)。简单的说，Capacity就是bucket的大小，Load factor就是bucket填满程度的最大比例。如果对迭代性能要求很高的话不要把capacity设置过大，也不要把load factor设置过小。当bucket中的entries的数目大于capacity\*load factor时就需要调整bucket的大小为当前的2倍。

**2. put函数的实现**

put函数大致的思路为：

（1）对key的hashCode()做hash，然后再计算index;

（2）如果没有hash冲突直接放到bucket里；

（3）如果有hash冲突，以链表的形式存在buckets后；

（4）如果hash冲突导致链表过长(大于等于TREEIFY\_THRESHOLD)，就把链表转换成红黑树；

（5）如果节点已经存在就替换old value(保证key的唯一性)

（6）如果bucket满了(超过load factor\*current capacity)，就要resize。

**3. get函数的实现**

在理解了put之后，get就很简单了。大致思路如下：

（1）bucket里的第一个节点，直接命中；

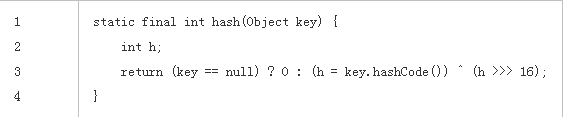
（2）如果有冲突，则通过key.equals(k)去查找对应的entry。若为树，则在树中通过key.equals(k)查找，O(logn)；若为链表，则在链表中通过key.equals(k)查找，O(n)。

**4. hash函数的实现**

在get和put的过程中，计算下标时，先对hashCode进行hash操作，然后再通过hash值进一步计算下标，如下图所示：



在对hashCode()计算hash时具体实现是这样的：



可以看到这个函数大概的作用就是：高16bit不变，低16bit和高16bit做了一个异或。

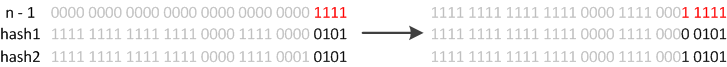
在设计hash函数时，因为目前的table长度n为2的幂，而计算下标的时候，是这样实现的(使用&位操作，而非%求余)：

(n - 1) & hash

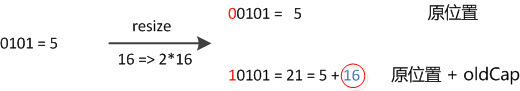
**5. resize的实现**

当put时，如果发现目前的bucket占用程度已经超过了Load Factor所希望的比例，那么就会发生resize。在resize的过程，简单的说就是把bucket扩充为2倍，之后重新计算index，把节点再放到新的bucket中。当超过限制的时候会resize，然而又因为我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。

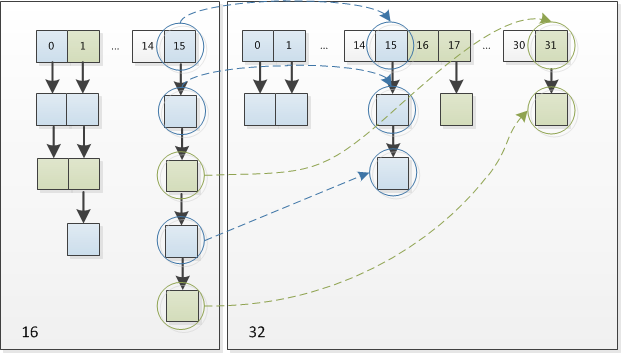
例如我们从16扩展为32时，具体的变化如下所示：



因此元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”。可以看看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。

**6.红黑树**

红黑树是一种**自平衡二叉查找树**， 它的统计性能要好于平衡二叉树（AVL树）。它通过**红黑两种颜色域保证树的高度近似平衡(左右子节点的高度差不大于1)。**它的每个节点是一个**五元组**：**color（颜色），key（数据），left（左孩子），right（右孩子）和p（父节点）。**

红黑树的定义也是它的性质，有以下五条：

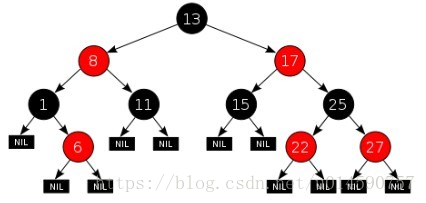
性质1.节点是红色或黑色

性质2.根是黑色

性质3.所有叶子都是黑色（叶子是NIL节点）

性质4.如果一个节点是红的，则它的两个儿子都是黑的

性质5.从任一节点到其叶子的所有简单路径都包含相同数目的黑色节点。



这五个性质强制了红黑树的关键性质: **从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。**为什么呢？性质4暗示着任何**一个简单路径上不能有两个毗连的红色节点**，这样，**最短的可能路径全是黑色节点，最长的可能路径有交替的红色和黑色节点。**同时根据性质5知道：**所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。**

**7.常见问题**

1. 你知道HashMap的工作原理吗？

通过hash的方法，通过put和get存储和获取对象。存储对象时，我们将K/V传给put方法时，它调用hashCode计算hash从而得到bucket位置，进一步存储，HashMap会根据当前bucket的占用情况自动调整容量(超过Load Facotr则resize为原来的2倍)。获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。如果发生碰撞的时候，Hashmap通过链表将产生碰撞冲突的元素组织起来，在Java 8中，如果一个bucket中碰撞冲突的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。

2. 你知道get和put的原理吗？equals()和hashCode()的都有什么作用？

通过对key的hashCode()进行hashing，并计算下标( n-1 & hash)，从而获得buckets的位置。如果产生碰撞，则利用key.equals()方法去链表或树中去查找对应的节点。

3. 你知道hash的实现吗？为什么要这样实现？

在Java 1.8的实现中，是通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在bucket的n比较小的时候，也能保证考虑到高低bit都参与到hash的计算中，同时不会有太大的开销。

4. 如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？

如果超过了负载因子(默认0.75)，则会重新resize一个原来长度两倍的HashMap，并且重新调用hash方法。

5. 为什么Hashmap要用红黑树，他比AVL树的优势在哪？

红黑树的查询性能略微逊色于AVL树，因为他比avl树会稍微不平衡最多一层，也就是说红黑树的查询性能只比相同内容的avl树最多多一次比较，但是，红黑树在插入和删除上完爆avl树，avl树每次插入删除会进行大量的平衡度计算，而红黑树为了维持红黑性质所做的红黑变换和旋转的开销，相较于avl树为了维持平衡的开销要小得多。

### 1.2.2 CurrentHashMap

### 1.2.3 List

### 1.2.4 Set

## 1.3 Java中的锁

### Synchronized锁的原理

### 1.3.2 CAS原理

### 1.3.3 CPU指令重排

Cpu为了提高效率会对指令进行重排序，以适合cpu的顺序运行。但是指令重排会遵守As-if-serial的规则，就是所有的动作(Action)都可以为了优化而被重排序，但是必须保证它们重排序后的结果和程序代码本身的应有结果是一致的。所以这种情况在单线程中不会出现什么问题。而对于多线程，这个规则就失效了，所以可能会导致结果出现问题。

解决办法就是内存屏障，也叫内存栅栏。是一种屏障指令，cpu指令。Java中的实现方式就是使用volatile关键字，既可以解决可见性，又可以禁止指令重排。

### 1.3.4 可重入锁不可重入锁

### 1.3.5 同步器AQS

## 1.4线程及线程池

## IO（BIO、NIO、AIO、Netty框架）

## 1.6设计模式

# Spring生态及Springboot

## SpringIOC

## SpringAOP

## Spring事务

# Mybatis

# Mysql

## 4.1 MySQL的索引原理

### 4.1.1 索引的理解

MySQL数据库支持多种索引类型，如B+树索引，哈希索引，全文索引等等.本文只关注B+树索引，因为这是平常使用 MySQL 时接触最多的索引. 高效的查找算法往往需要特定的数据结构支持, 在数据之外,数据库系统就维护着满足特定查找算法的数据结构.这些数据结构以某种方式指向数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构, 就是索引. 索引作为一种数据结构,一般来说本身也很大,不可能全部存储在内存中, 因此索引往往以索引文件的形式存储在磁盘上.

### 4.2.2 局部性原理与磁盘预读

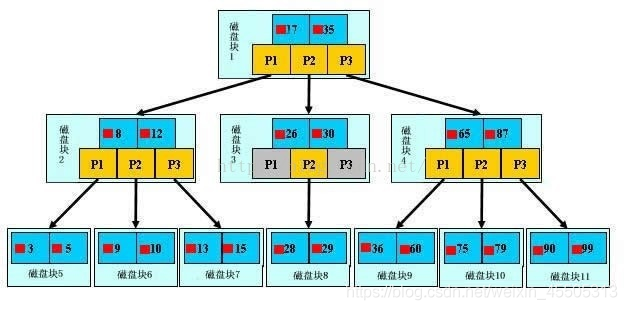
（1）由于存储介质的特性，磁盘本身存取比主存慢很多，再加上磁盘I/O存在机械运动耗费，因此磁盘I/O的时间消耗是巨大的。为了提高磁盘与主存的交互效率, 首要的手段就是减少磁盘 I/O 次数。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存. 这样做的理论依据是局部性原理：当一个数据被用到时，其附近的数据通常也会马上被使用.

（2）预读的长度一般为页(page)的整倍数. 页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块, 每个存储块称为一页(页大小通常为4k)，主存和磁盘以页为单位交换数据. 当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序拿到数据继续运行.

### 4.1.3 索引的数据结构

索引一般以文件形式存储在磁盘上, 为了提高性能, 每次又要把部分数据读入内存来处理,这就意味着索引检索需要磁盘I/O操作, 而我们又一般以使用磁盘I/O次数来评价索引结构的优劣. 这样, B+ 树就理所当然地被选做理想的索引数据结构了. B+ 树是 B 树的变种, 要了解B+ 树的性质还是要建立在B 树的基础之上.

假设一个m 阶 B-Tree:m 为一个节点最大子节点个数



**B 树的性质:**

* 一个节点最多有m个分支
* 一个节点有n个关键字,则有n+1个分支
* 根节点最少有两个分支,非根非叶节点至少有ceil [m/2] 个分支
* 节点的关键字递增排序
* 节点中的关键字都大于其左边分支上的关键字,小于其右边分支上的关键字
* 所有叶节点属于同一层,也就是具有同样的深度 h

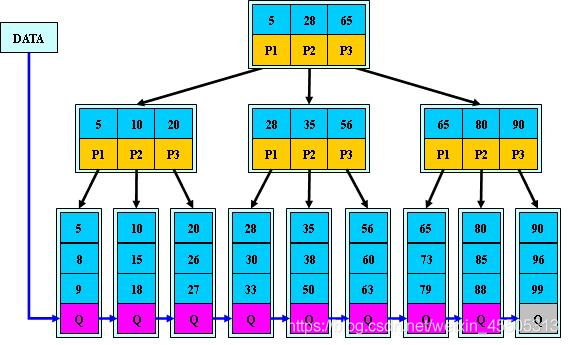
由于B-Tree的特性, 在B-Tree中按key检索数据的算法很直观：首先从根节点进行二分查找，如果找到则返回对应key的data，找不到则对相应区间的指针指向的节点向下递归进行查找, 直到找到key成功返回数据或找到null指针失败.以上图举例来说, 要检索 key=9 的数据, 则需要经过磁盘块1--> 磁盘块2-->磁盘块6过程,**可知要访问的磁盘块数量和树高直接相关,也就是I/O次数取决于树的高度h**.

数据库系统的设计者利用磁盘预读原理,将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入.为了达到这个目的，在实际实现B-Tree时有如下技巧：

1. 每次新建节点时直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加上计算机存储分配都是按页对齐的，这就实现了一个节点只需一次I/O.
2. B-Tree中一次检索最多需要h-1次 I/O(根节点常驻内存)，渐进复杂度为O(h)=O(logd N).一般实际应用中出度d是非常大的数字，通常超过100，因此h非常小(通常不超过3).

综上所述，如果采用B-Tree存储结构，搜索时I/O次数一般不会超过3次,所以用B-Tree作为索引结构效率是非常高的.

我们已经知道外存储器的信息读取最大的时间消耗在于磁盘I/O, 那么一个基本的思路就是在一个磁盘页上多存储一些索引信息, 以进一步减少I/O次数, 换句话说就是在B-tree 的基础上实现同样多的数据以更小的树高存储.这样 B 树的变种 B+ 树就产生了.



**B+ 树与 B 树的差异:**

* 一个节点有n个关键字,则有n个分支
* B+ 树只有叶节点会存储 data, 内节点只有关键字key, 而B树任意节点中 key-data 都是成对出现
* 所有叶节点保存了所有父节点关键字记录的指针,所有数据地址都必须到叶节点才能获取,故每次查询数据I/O次数相同(因为都需要到叶节点,而叶节点都在同一层),查询效率稳定
* B+ 树所有叶节点通过指针连接到一起, 范围查询效率更高
* B+ 树内节点不存储 data, 则同一个页的空间可以容纳更多的关键字(这就是索引字段要尽量小的目的), 一次性读入内存可供检索的关键字也更多. 这样在数据量足够大的情况下, B+ 树拥有比 B 树更小的树高,也就可以减少 I/O 次数,提高性能.

### 4.1.4 MySQL 支持的存储引擎

在MySQL中，可以使用show engines语句来显示可用的数据库引擎和默认引擎。MySQL提供了多个不同的存储引擎，包括处理事务安全表的引擎和处理非事务安全表的引擎。在MySQL中，不需要再整个服务器中使用同一种存储引擎，针对具体的要求，可以对每一个表使用不同的存储引擎。

MySQL5.7支持的存储引擎有InnoDB、MyISAM、Memory、Merge、Archive、Federated、CSV、BLACKHOLE等。MySQL默认的存储引擎是InnoDB。

MyISAM和InnoDB的区别：

1 MyIsAM 引擎

MyISAM 索引文件和数据文件是分离的，索引文件仅保存数据记录的地址, 类似于使用偏旁部首查字典.使用索引搜索时, 如果指定的Key存在,则取出其data域的值,然后以data域的值为地址,读取相应数据记录. 这种索引方式也叫**非聚集索引**，它的数据在索引结构中逻辑上连续，但是实际存储的物理位置并不连续.

2 InnoDB 引擎

InnoDB 数据文件本身就是索引文件, 类似于使用拼音查字典,正文内容本就是按照一定规则排列的目录. 在 InnoDB 索引中 B+Tree的叶节点 key 为数据表主键, data域保存完整的数据记录, 这种索引方式也可以叫**聚集索引**，它的数据记录存储在物理上连续的位置

问题：

**1.如何选择MySQL的存储引擎？**

**1. 是否要支持事务，如果要请选择innodb，如果不需要可以考虑MyISAM；**

**2. 如果表中绝大多数都只是读查询，可以考虑MyISAM，如果既有读也有写，请使用InnoDB。**

**3. 系统奔溃后，MyISAM恢复起来更困难，能否接受；**

**4. MySQL5.5版本开始Innodb已经成为Mysql的默认引擎(之前是MyISAM)，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。**

2. **InnoDB为什么推荐使用自增ID作为主键？**

**答：自增ID可以保证每次插入时B+索引是从右边扩展的，可以避免B+树和频繁合并和分裂（对比使用UUID）。如果使用字符串主键和随机主键，会使得数据随机插入，效率比较差。**

**3.innodb引擎的4大特性**

**插入缓冲（insert buffer),二次写(double write),自适应哈希索引(ahi),预读(read ahead)**

## **索引及SQL优化**

### 4.2.1 索引最左前缀原理

在（a,b,c）字段上建立联合索引, 真正查询时是否会用到索引其实和查询语句也有关系, mysql会一直从左向右匹配直到遇到范围查询(>、<、between、like)就停止匹配.

以下的查询方式都可以用到索引

select \* from table where a=1；

select \* from table where a=1 and b=2；

select \* from table where a=1 and b=2 and c=3；

三个查询按照 （a ）, （a，b ）,（a，b，c ）顺序都可以利用到索引，这就是最左前缀匹配.

如果查询语句是：

select \* from table where a=1 and c=3；

只会用到索引a.

如果查询语句是：

select \* from table where b=2 and c=3；

因为没有用到最左前缀a，所以这个查询用不到索引

用到了最左前缀但是顺序颠倒,比如

select \* from table where b=2 and a=1；

select \* from table where b=2 and a=1 and c=3；

如果用到了最左前缀而只是颠倒了顺序,也可以用到索引

### 4.2.2 索引优化建议

（1）查询优化

1. 最左前缀匹配原则
2. 对 where,on,group by,order by 中出现的列使用索引
3. 对于like查询，”%”不要放在前面  
   SELECT \* FROM COOK WHERE name LIKE ‘后盾%’ – 走索引  
   SELECT \* FROM COOK WHERE name LIKE ‘%后盾%’ – 不走索引
4. 查询where条件数据类型不匹配无法使用索引
5. 字符串与数字比较不使用索引
6. 正则表达式不使用索引

（2） 索引结构优化

1. 主键外键一定要建索引
2. 尽量选择区分度高的列作为索引,区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例. 比例越大我们扫描的记录数越少,唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0.
3. 对较小的数据列使用索引,这样会使索引文件更小,同时内存中也可以装载更多的索引键
4. 为较长的字符串使用前缀索引
5. 尽量扩展索引,不要新建索引.
6. 不要过多创建索引. 建立索引是为了提高查询效率，但建立的索引过多，会影响插入、删除数据的速度，因为我们修改表数据之后，索引也需要进行调整重建

### 4.2.3 MySQL查询语句的优化

* 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及 的列上建立索引。
* 应尽量避免在 where 子句中使用!=或<>操作符，否则引擎将放弃使用索引而进行 全表扫描。
* 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用 索引而进行全表扫描，如： select id from t where num is null 可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询:select id from t where num=0
* 应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引 而进行全表扫描，如： select id from t where num=10 or num=20 ，可以使用可以这样查询： select id from t where num=10 union all select id from t where num=20
* 以%开头的模糊查询也会导致全表扫描： select id from t where name like '%abc%' ，如果要提高效率的话，可以考虑全文检索来解决。
* in 和 not in 也要慎用，否则会导致全表扫描，如： select id from t where num in(1,2,3) 对于连续的数值，能用 between 就不要用 in 了： select id from t where num between 1 and 3
* 应尽量避免在 where 子句中对字段进行表达式操作，这将导致放弃使用索引 而进行全表扫描。如：select id from t where num/2=100 应改为: select id from t where num=100\*2
* 应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用索引而 进行全表扫描。

比如说查询name以abc开头的数据： select id from t where substring(name,1,3)='abc' ，可以改为select id from t where name like 'abc%'

* 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系 统将可能无法正确使用索引。
* 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中 的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可 能的让字段顺序与索引顺序相一致。
* 很多时候用 exists 代替 in 是一个好的选择： select num from a where num in(select num from b) 用下面的语句替换： select num from a where exists(select 1 from b where num=a.num)
* 并不是所有索引对查询都有效，SQL 是根据表中数据来进行查询优化的，当索引 列有大量数据重复时，SQL 查询可能不会去利用索引，如一表中有字段 sex，男、女的值 几乎各一半，那么即使在 sex 上建了索引也对查询效率起不了作用。
* 索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低 了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索 引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过 6 个

## 4.2分库分表

# Redis

# Zookeeper

# Kafka（消息队列中间件）