索引

> 索引是一个排好序的数据结构

- 二叉树
- 红黑树
  -
- Hash表
- B-tree

索引的优缺点

优点:

1. 大大加快数据检索的速度
2. 将随机I/O变成了顺序I/O(B+树的叶子节点是连接在一起的)
3. 加速表与表之间的连接

缺点:

1. 建立索引需要物理空间
2. 创建和维护索引需要花费时间，增删改查时都需要额外时间去维护索引

索引的种类有哪些

1. 主键索引:不允许重复，不能为空，一个表只能有一个主键索引
2. 组合索引：多个列值组成的索引，最左侧
3. 唯一索引：不允许重复，可空，索引列的值必须唯一，如果是组合索引，组合值必须唯一
4. 全文索引：对文本内容进行搜索
5. 普通索引：基本类型，可空

哪些字段适合加索引

表的主键，外键必须有索引

数据量超过300的表应该有索引

经常与其它表连接的表，连接字段上应该有索引

经常在where子句中的字段

选择性高的字段

java序列化

序列化是一种用来处理对象的机制，所谓对象流也就是降对象的内容进行流化，将数据分解成字节流，以便存储在文中或者网络上传输
 序列化实现java.io.Serializable接口 这是一个空接口 作用只是为了在序列化和反序列化中做一个类型判断

序列化就是把对象改成二进制的过程 可以保存到磁盘或者网络发送

java.io.Externalizable
ObjectInputStream
ObjectOutStream

哈希

底层是哈希表，hashtable是线程安全的, hashmap不是线程安全的

HashMap的get工作原理

1. hashMap底层数据结构是哈希表
2. 哈希表就是一个数组，数组的每一个元素是一个单向链表
3. 先根据键的哈希码，经常hash函数计算得出hash值
4. 根据hash值计算数组的下标 i
5. 访问数组元素 table[i]
    - 如果 table[i]数组元素为null,直接返回null
    - 如果table[i]元素不为null，遍历table[i]单向链表的每个结点，如果有某个结点的key与当前的键 equals相等，就把结点的value值返回， 如果链表中所有结点的key都不匹配 就返回null

初始化容量 hashmap是16 哈市table是11

默认负载因子 0.75，当键值对的数量> 容量*加载因子 就进行扩容

hashmap按2倍大小扩容，hashtable按2倍 +1 扩容

都可以指定初始化容量　hashmap会把初始化容量调整为2的幂次方

把17-31之间的初始化容量调整为32 把33-63之间的初始化容量调整为64 为了快速计算数组的下标 hashtable不调整

hashmap的键与值都可以为null, hashtable的键与值都不能为null

hashMap =new HashMap<>(initialCapacity:23)

hashTable =new HashTable<>(initialCapacity:23)

==和equals

基本数据类型比较的是他们的值是否相等

引用数据类型比较的是他们的内存地址是否同一地址

equals常用来比较对象的内容是否相同

equals使用良好习惯用常量去比较变量 即将不会为null的对象放在前面

如果两个对象都可能为null 可以使用java标准库中的工具类 Objects来进行equals比较，Objects.equals(s1,s2)

equals它不能比较基本数据类型 当比较包装类时 要注意你比较的值是否与包装类一致 包装类中是重写了equals方法的 它首先会判断另一个值是否为同一数据类型

堆和栈

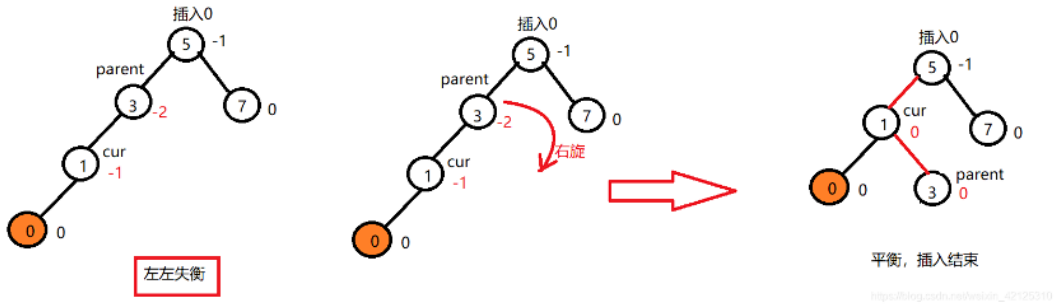
1. 栈内存存储的是局部变量而堆内存存储的是实体
2. 栈内存的更新速度要快于堆内存，因为局部变量的生命周期很短
3. 栈内存存方的变量生命周期一旦结束就会被释放而堆内存存方的实体会被垃圾回收机制不定时的回收

红黑树

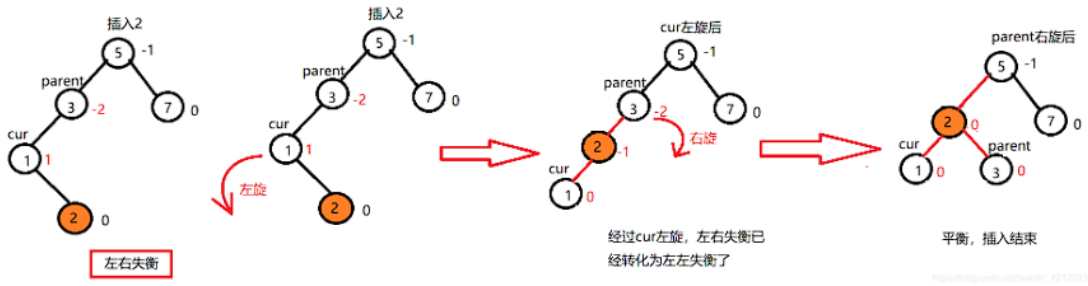
1. 根结点是黑色叶结点是不存储数据的黑色空结点
2. 任何相邻的两个结点不能同时为红色
3. 任意结点到其可到达的叶结点间包含相同数量的黑色节点
4. 频繁插入 频繁删除更有优势

平衡二叉树

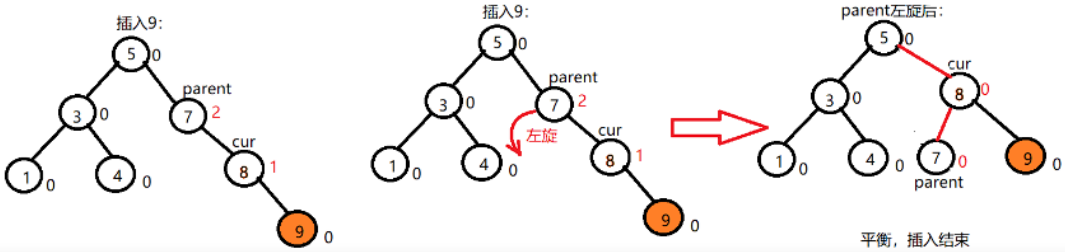
- 根据不同的失衡，口诀中有不同的解决方法，接下来我们一起来验证一下吧：
  - "左左失衡，parent右旋"：



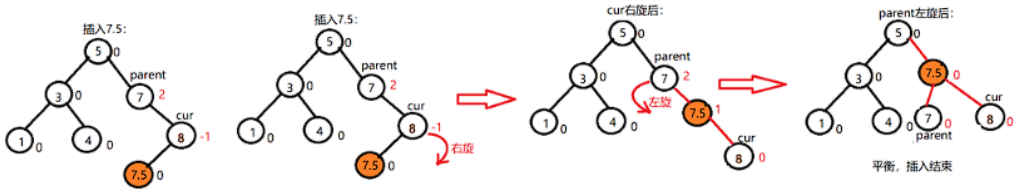
  - "左右失衡，cur左旋，parent右旋"：



  - "右右失衡，parent左旋"：



- 右右失衡
  - “右左失衡，cur右旋，parent左旋”：



红黑树和平衡树的对比和选择

1. 平衡树结构更加直观，读取性能比红黑树要高，增加和删除结点 恢复平衡的性能不如红黑树
2. 红黑树 读取性能不如平衡树 增加和删除结点 恢复平衡性能比平衡树好

(21条消息)【数据结构】红黑树与平衡二叉树的区别以及原理详解（附图解）雪花不落的博客-CSDN博客红黑树与平衡二叉树

treemap

元素从小到大的顺序排列　treemap的基本操作的时间复杂度都是log(n)

B树

(21条消息) B树、B+树详解KuoGavin的博客-CSDN博客b树和b+树

多路平衡查找树，B树中所有结点的孩子结点数的最大值称为B树的阶

特性

1. 树中每个结点至多有m颗子树(既至多含有m-1个关键字)
2. 若根结点不是终端结点，则至少有两颗子树
3. 除根结点外的所有非叶结点至少有[m/2]颗子树(即[m/2]-1个关键字)

B+树

B+树比B树更适合数据库索引

1. B+树的磁盘读写代价更低
2. B+树查询效率更加稳定
3. B+树便于范围查询

mysql中的B+Tree

mysql索引数据结构对经典的B+Tree进行了优化。在原B+Tree的基础上，增加了一个指向相邻叶子节点的链表指针，就形成了带有顺序指针的B+Tree 提高区间访问的性能

6. B树和B+树比较

B树	B+树
B树的每个节点，有m个key，m+1个指针，每个指针分别是区间，代表大于前面的key，小于后面的key	B+树的每个节点，有m+1个key，m+1个指针，每个指针与一个key对应，代表子节点中的数全部大于当前key。同时，因此每个节点的key值更多，所以整个树的高度更低。
B树中每个节点的每个key都有数据信息	B+树中只有叶子节点有数据信息，非叶子节点没有。所以B+树的非叶子节点大小更小
B树的所有数据是一个整体：非叶子节点也是数据的一部分，可能还没到叶子节点就已经找到，返回了。B树的所有节点都是数据	B+树的非叶子节点就是单纯的索引，所有实际的数据都存储在叶子节点中，所以每次查询，都必须查询到叶子节点，所以每次查询的速度就十分的稳定
B树不可以进行叶子节点间的顺序查找，同时若是可以也没意义，因为是中序遍历	B+树的叶子节点有指针连着，可以范围查找，即循着范围起点的叶子节点进行顺序遍历

JVM

类加载器

1. 启动类
2. 扩展类
3. 应用程序
4. 自定义

软引用和弱引用

软引用: 基本适用于缓存，当内存空间不足时会被垃圾回收器释放掉

弱引用: 一旦发现了只具有弱引用的对象不管当前内存空间足够与否都会回收它的内存

垃圾回收算法

1. 标记清除算法
    - 执行效率不稳定
    - 内存空间的碎片问题
2. 复制算法: 把内存分为两块等同大小的内存空间。先使用A内存使用 满了的时候把存活的对象放到B内存中。然后把A内存全部删除 然后B满了后再转移到A这样循环
    1. 年轻代使用这个算法
    2. **缺点**: 浪费的一半的内存 降低空间的使用率
3. 标记整理
    1. 原理：标记-整理算法的标记过程与“标记-清理算法”一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向内存空间一端移动，然后直接清理掉边界以外的内存。
    2. 老年代的时候会使用
        1. 避免碎片化
        2. 避免内存浪费

redis

缓存穿透：指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，进而给数据库带来压力。通俗点说，读请求访问时，缓存和数据库都没有某个值，这样就会导致每次对这个值的查询请求都会穿透到数据库，这就是缓存穿透。

1. **业务不合理的设计**
2. **业务/运维/开发失误的操作**
3. **黑客非法请求攻击**

避免缓存穿透

1. .如果是非法请求，我们在API入口，对参数进行校验，过滤非法值。
2. ..如果查询数据库为空，我们可以给缓存设置个空值，或者默认值。但是如有有写请求进来的话，需要更新缓存哈，以保证缓存一致性，同时，最后给缓存设置适当的过期时间。（业务上比较常用，简单有效）
3. 使用布隆过滤器快速判断数据是否存在。即一个查询请求过来时，先通过布隆过滤器判断值是否存在，存在才继续往下查。

缓存雪奔： 指缓存中数据大批量到过期时间，而查询数据量巨大，请求都直接访问数据库，引起数据库压力过大甚至down机。

1. 可通过均匀设置过期时间解决，即让过期时间相对离散一点。

缓存击穿： 指热点key在某个时间点过期的时候，而恰好在这个时间点对这个Key有大量的并发请求过来，从而大量的请求打到db。

1. **2. “永不过期”**，是指没有设置过期时间，但是热点数据快要过期时，异步线程去更新和设置过期时间。
2. **1.使用互斥锁方案。**缓存失效时，不是立即去加载db数据，而是先使用某些带成功返回的原子操作命令，如(Redis的setnx）去操作，成功的时候，再去加载db数据库数据和设置缓存。否则就去重试获取缓存。

说说Redis的常用应用场景

- 缓存
- 排行榜
- 计数器应用
- 共享Session
- 分布式锁
- 社交网络
- 消息队列
- 位操作

Redis提供了RDB和AOF两种持久化机制

- **RDB**，就是把内存数据以快照的形式保存到磁盘上。
    - 适合大规模的数据恢复场景，如备份，全量复制等
    - 没办法做到实时持久化/秒级持久化。
    - 新老版本存在RDB格式兼容问题
- **AOF（append only file）** 持久化，采用日志的形式来记录每个写操作，追加到文件中，重启时再重新执行AOF文件中的命令来恢复数据。它主要解决数据持久化的实时性问题。默认是不开启的。
    - AOF记录的内容越多，文件越大，数据恢复变慢。
    - 数据的一致性和完整性更高

MySQL与Redis 如何保证双写一致性

- 缓存延时双删
- 删除缓存重试机制
- 读取biglog异步删除缓存

spring

@Autowired ： 自动按类型注入，如果有多个匹配则按照指定的Bean的id查找，查找不到会报错。
@Qualifier ： 在自动按照类型注入的基础上再按照Bean的id注入，给变量注入时必须搭配
@Autowired，给方法注入时可单独使用。
@Resource：直接按照Bean的id注入，只能注入Bean类型。
@Value：用于注入基本数据类型和String类型
@scope: @Scope注解是 Spring IOC 容器中的一个作用域

1. singleton(单例)
2. prototype(多例)
3. reqeust 同一次请求
4. session 同一个级别会话

Ioc:

即控制反转，把原来的代码里需要实现的对象创建，依赖反转给容器来帮忙实现，需要创建一个容器并且需要一种描述让容器知道要创建的对象间的关系，在Spring中管理对象及其依赖关系是通过Spring IOC容器实现的。Ioc的实现方式有依赖注入和依赖查找，由于依赖查找使用的很少，因此Ioc也叫做依赖注入。依赖注入指对象被动地接收依赖类而不用自己主动去找，对象不是从容器中查找它依赖的类，而是在容器实例化对象时主动将它依赖的类注入给它。假设一个Car类需要一个Engine的对象，那么一般需要手动new 一个Engine，利用Ioc就只需要定义一个私有的Engine类型的成员变量，容器会在运行时自动创建一个Engine的实例对象并将引入自动注入给成员变量。

AOP:

即面向切面编程，通过预编译和运行期动态代理实现程序功能的统一维护。常用场景：权限认证、自动缓存、错误处理、日志、调试和事务等。

@Aspect：声明被注解的类是一个切面的Bean；
@Before：前置通知，指在某个连接点之前执行的通知；
@After：后置通知，指某个连接点退出时执行的通知（不论程序是正常返回还是异常退出）
@AfterReturning：返回后通知，指某连接点正常完成之后执行的通知，返回值使用returning属性接收。
@AfterThrowing：异常通知，指方法抛出异常导致退出时执行的通知，和@AfterReturning只会有一个执行，异常使用throwing属性接收。