# 第一章:基本类型与集合

1. byte 8位,有符号, 首位是符号位不参与运算(1负,0 正),剩余7位可以有 2^7=128种可能,从0开始的 ,所以数值范围[0:127].加上符号就是 [-127:128] 但是+-0重复,约定 -0是 -128 所以8byte存 [-128:127]
2. 注意short 和 char 是16字节. int 和float 是 32字节, long 和 double是64字节.
3. String:
   1. String 的 equals() 比较的是值.
   2. StringBuffer 的 equals() 没有重写,所以比较的是内存地址.
   3. String 是对象,对象的默认值都是 null
   4. new String() 和 new String(“”) 申明一个对象,空串, 空串不是null.
   5. 对象被存储在堆当中, 所以new String(“123”)对象存储在堆中,而String s1=”123” 是存储在常量池当中. 所有这两个有不同的内存地址.
   6. [常量池]:编译期间被确定,并被保存在已编译的.class文件中的数据,包括 类的 方法,接口中的常量,也包括字符串常量. 可以节省内存空间

String s1=”123”;

String s2=”123”

String s3=”1” +”23” //s3 也是在编译期间被确定的.

String 直接定义常量,JVM会把对应的值方在常量池中.只保留一份.

只要是编译期间不能确定的String 类型,都是存储在堆当中的,否则存储在常量池当中.

1. str.intern() 检查常量池中有没有str相同Unicode的字符串常量, 有返回引用,没有,在常量池中增加一个相同内容的字符串并返回引用. 注意str引用本身还是指向堆空间的str.除非 str=str.intern() 这样str才指向常量池中的字符串.
2. str.split(“分隔符”)方法可以将String按照某种分隔符拆分成数组返回.当String非常大的时候,有性能瓶颈,可能卡死.
3. 基本类型及其转换
4. short 和 char 是同级别的,不能自动互相转换.
5. 每个基本类型都有包装类. 除了Character之外,其他包装类都可以将字符串作为参数来构造实例.
6. 包装类内部都是final类型的,不可变,包装类是引用传递,
7. 所谓的值修改之后会生成新的内存地址,可以理解为值传递.
8. Integer.valuOf( int i) 就是把I 自动装箱成Integer对象.当i属于[-128:127] 返回的是缓存的Integer对象,并不是新建对象,两个对象引用指向同一块地址.
9. 【进阶】long如何自动转化为float? [省略,先不看了]

**集合:**

1. Arrays类,即数组的工具类.包含一组Static函数. Arrays.fuction() 调用.
   1. equals(), arrays.sort() 排序
   2. fill() 将值填入 Arrays.fill(arr3, 10); //将数组全部填充10
   3. binarySearch() 在排好序的array中寻找元素.
   4. System.arraycopy() array 的复制.
2. 集合总的分 Collection( List, Set) Map(HashMap ,Hashtable) 储存在java.util包中.
3. List 以特定次序存储元素,取出来的顺序可能与放入的不同
   1. ArrayList/LinkedList/Vector(ArrayList线程安全版本) ,
   2. Stack是Java自己实现的栈,继承 Vector ,
4. Set不含有重复元素
   1. HashSet/TreeSet
5. Map
   1. Hashtable 不允许null键和null值, 而HashMap允许随意null值和最多一个null键.
   2. Hashtable 是 HashMap的线程安全版本.
   3. TreeMap内部用树来实现的Map,可以方便排序.
6. [问点:]List要删除元素的时候,如何处理呢?
   1. 采用Iterator进行遍历,用 iterator.remove()进行删除.
   2. 如果使用list.remove() 会报ConcurrentModificationException
   3. 可以用for循环遍历list, 或者是增强for. 但是删除元素后,list的数量会发生变化.再次循环便会报错.如果删除完毕之后马上break跳出,就不会触发报错. 只适合删除一个元素. 最好不要用这个方法.

Iterator<String> it = list.iterator();

while(it.hasNext()){

String x = it.next();

if(x.equals("del")){

it.remove();

}

}

1. Collections 是集合的工具类, 包含集合相关的静态方法, 不可实例化. 服务于Java的Collection框架
   1. Collections.sort(list) 对list进行排序. 按照默认规则
   2. 当集合元素(对象)没有排序规则的时候可以自定义规则进行排序, 这里举例一个 实现 **Comparator**接口

该接口用于对象与对象间的比较，返回值意味着这两对象之间的大小关系。  
**-1代表前者小，0代表两者相等，1代表前者大。 最终排序按照从小到大来排**

//对于Comparator接口的重写，这个接口就像一个抽象函数，给出的参数与返回值都是定死的

Collections.sort(studentArr, new Comparator<Student>(){

@Override

public int compare(Student o1, Student o2) {

// 内部的判断逻辑是自定义的，这里根据age进行对象的排序

if (o1.age > o2.age) {

return 1;

} else if (o1.age < o2.age) {

return -1;

}

return 0;

}

});

1. [问点:]List元素要按照对象的属性进行排序，该如何实现呢？答案见7.
2. [高频问点:] 集合的内部源码是如何实现的（比如HashMap或者TreeSet等）？
   1. HashMap内部使用(Entry数组)实现, 每个位置是一个Entry 包含 key value 和hash值.

根据key 算出hashcode值, 这个值对应数组的一个位置.

插入的时候:**当前hashcode对应的位置没有元素的话, 直接放入. 有元素的话,与当前元素进行比较 如果 equals true 一样,就替换.如果不一样(hash冲突),就保存一个链表, 一直存下去(存的都是一个个Entry, Entry是基本单位)**

Map查数据的时候: 根据HashCode找到位置,没有返回null,有 看equals. 相等返回响应的value.

因为hash函数判断数组位置,所以可能有不同的key他的hash值是相同的,就要放在同一个数组位置, 所以在相同位置就用链穿起来来解决哈希冲突.. 这就是哈希冲突.

map根据key来计算哈希值,从数组取值,时间复杂度是O(1)

JDK1.8开始,如果链表长度超过8 就不用链表改成红黑树结构来储存数据, 解决链表较长的时候性能下降的问题.

* 1. HashSet 是一个没有重复元素的集合. 是由HashMap实现的, HashMap保存的是键值对, 我们只能向HashSet添加key,原因在于HashSet的 Value 都是同一个对象.

private static final Object PRESENT = new Object();

public boolean add(E e) {

return map.put(e, PRESENT)==null;

}

* 1. TreeMap 带 tree的 底层都是树. TreeMap直接就是一颗红黑树

树结构的键值对.每个元素都是Entry结构.(entry就是map用来存储元素的类.Map.Entry)

**如何维护数据的唯一性??**在存入数据的时候，他们会调用元素中实现的 Comparable 的 compareTo()方法或者集合本身创建的时候传入的比较器 Comparator.

调用比较方法，返回-1 的时候，添加到左子树，返回1 的时候 添加到 右子树。返回0 有相同数据 不添加该元素！

TreeMap添加元素的时候会调用key.comparatorTo方法进行比较, 很多类都实现了Comparable接口,这个时候key就有compareTo方法. 如果key类 没有实现Comparable接口的话,必须TreeMap(new Comparator) new一个比较器. 即不能使用无参构造.因为如果没有比较器的话,会调用key的compareTo方法.

* 1. TreeSet 是没有重复的排序集合, 底层是由TreeMap实现的.

1. HashMap是线程不安全的,ConcurrentHashMap和Hashtable是HashMap的线程安全版本.他俩就是锁的粒度和如何锁的区别.
2. ArrayList
   1. 通过数组实现, 默认容量大小是 10
   2. 当容量不足的时候会扩容, 新的容量=“(原始容量x3)/2 + 1”。
   3. ArrayList的克隆函数，即是将全部元素克隆到一个数组中。
3. LinkedList
   1. 通过**双向链表**来实现.
   2. 包含重要内部类,Entry Entry是双向链表节点所对应的数据结构，它包括的属性有：当前节点所包含的值，上一个节点，下一个节点。
   3. LinkedList不存在容量不足的问题. 因为链表不是连续空间.
   4. LinkedList的克隆函数，即是将全部元素克隆到一个新的LinkedList对象中
4. Vector 是ArrayList的线程安全版本, vector是线程（Thread）同步（Synchronized）的，此种实现方式（即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性）需要很高的花费，因此，访问它比访问ArrayList慢
5. Stack是java实现的**栈**，Stack继承于Vector，意味着Vector拥有的属性和功能，Stack都拥有, Stack也是线程安全的。
6. Map是否是有序的??
   1. HashMao是无序的
   2. TreeMap是有序的, Tree的都是有序的.
7. equals()相等的两个对象，hashcode是否相同，为什么
   1. Map规定的,他的根据Hashcode来找到位置,然后再根据equals判断是否相等,如果equals相等,hashcode不等,那么HashMap的底层存储就全乱套了.
   2. 规定,如果两对象equals 返回true,hashCode一定相同.
   3. hashcode相同,两对象不一定相同.
   4. 违背这个原则,在使用容器时,相同对象可能出现在set中. 如果不同的hashcode也能equals的话,相同元素可能存储在多个位置,添加新元素的时候效率会大大降低(更多的哈希冲突.)
8. 哈希表hashtable(key，value) 就是把Key通过一个固定的算法函数既所谓的哈希函数转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将value存储在以该数字为下标的数组空间里。
9. Object 类 没有 copy()
10. 哈希冲突: 数据存储的时候,hash值一样,equals不一样,即两个不同的对象存储的数组位置一致,这就是冲突, 解决冲突方式就是在这个数组位置接链表, 使用链表结构会大大降低插入和查找的效率(如果equals一样,那就替换) 即hash冲突时,冲突处会变成链表,所以导致存取性能下降.

# 第二章: 运算符与编码

1. 正数的原码反码补码相同. 负数,反码符号位不变 其余取反. 补码是反码+1; 记住这个就记住三个码了
2. 计算机是用补码表示的.
3. 计算机中 2 8 10 进制都有. 8进制开头是[0] 16进制开头[0x]
4. 进制转换要会.
5. 编码与字符集
   1. 字库表: 相当于存储了所有可读可显示的字符的数据库.
   2. 编码字符集:表示字符在字库表中的存储位置[Unicode,ASCII,GB2312等] 就是个二进制串.
   3. 字符编码(编码方式):把**编码字符集**的二进制串转化成字库表中的真实位置.为了节省空间.把高频的字符放在前面.字符编码可以和字符集一致.
6. UTF-8 和 Unicode 的关系:
   1. Unicode 是字符集,UTF-8是一种字符编码,UFT-8 把 Unicode的位置转换成真实的字库中的位置.
   2. Unicode字符集有多种编码方案: utf-8, utf-16,utf-32 等
7. utf-8 占1-4个字节,汉字和中文字符占 3 字节,数字和英文字符占1 字节.
8. ISO8859-1 (欧标) 中文字符和英文字符都占1个字节.GB2312/GBK 一个中文占2字节.
9. Unicode是一个定长的编码标准,每个字符都是2字节.万国码
10. GBK,GB2312以及Unicode都既是字符集，也是编码方式，而UTF-8只是编码方式，并不是字符集, GBK、GB2312等与UTF8之间都必须通过Unicode编码才能相互转换：
11. Java可以提供字符串转码: **String show = new String(oldString.getBytes(“GBK”),“UTF-8”)**
12. Utf-8 char c=’中’ 是否合法?
    1. 是的,因为java是以unicode字符集为编码方式的(JVm内部使用的是UTF-16编码),每个字符2字节,char类型也是2字节.
    2. java在编译时会把utf-8的中文字符转换成对应的Unicode来进行传输运算(UTF-16).
13. 运算符相关
    1. java的运算符运算,底层都是用栈来实现的,a=a\*(b+c) 括号,加号,乘号通过栈,利用优先级来先后运算.
    2. 涉及byte,short,int类型的运算,都会先把值转换成int再来,最终结果是int类型(short+short 结果是 int)
    3. 注意final, final short+ int =int ; final short + final short = short
    4. +=，\*=，>>= 即混合运算符, 混合运算符不会改变类型,short也还是short
14. switch语句在jdk1,7之后,允许参数是 String, 建议在最后加上的default,default没有break;
15. 逻辑与& 全1才1., 逻辑或| 全0才0;
16. && || 短路的与 或. 即前面的是0,后面就不运行了.所以实际都是用短路的与或.
17. 位运算
    1. 位与( & ) 、位或( | )、位非( ~ )、位异或( ^ )
    2. 表面上看似乎有点像逻辑运算符，但逻辑运算符是针对两个数来进行逻辑运算（比如*条件1***&***条件2*），而位运算符主要针对两个二进制数的位进行逻辑运算。
    3. 注意做位运算的时候,都要换成二进制补码来进行.(正数补码是本身,负数补码是反码+1)
    4. & 每个位置一致设1
    5. | 每个位置有一个是1,设1
    6. ~ 取反
    7. ^位置两数相反,设1
18. 其实真正的逻辑与是&&, 一般1个&是按位与，有时见到(str != null & str != ''), 其实只是括号里面是正数1就认为true,本质还是按位与运算的. 一般放在if当中的可以认为是逻辑运算符.
19. 位移运算
    1. << 左移运算符. value << num 丢弃最高位, 0 补最低位. 如果num超过了该类型的最大位数,取模 .int型移动33位,其实是 移动 33-32=1位.
    2. >> 右移运算符. value >> num . 符号位移到了左数第二位,最左边补上原来的符号位.即符号位不变. 无符号的补0
20. Switch中是整数表达式或者枚举常量. 整型(或者精度比他低的 会自动转成int)以及他们的包装类都是可以的,但是double long float 等高精度的不可以作为Switch的参数.

# 第三章:面向对象相关

1. 面向对象三大特性: 封装,继承,多态.
   1. 封装:属性私有化,提供公有方法来访问私有属性
   2. 继承:从已有类中派生出新的类,新类有父类已有的属性和行为,子类是父类功能和属性上的扩充,子类不支持多继承.
   3. 多态:方法的重写,重载与动态连接.
      1. 重写是继承后重新实现父类的方法(重写后访问权限不低于父类,private方法不支持重写.)
      2. 重载是在一个类里一些列参数不同,名字相同的方法.
      3. 动态连接是父类型的引用指向**子类对象**.父类型引用调方法的时候,返回的是子类**重写**的方法.
2. static 不支持重写,不能被覆盖, 重写是基于运行时动态绑定的,而static 是编译时静态绑定的. static是类相关的,不是实例相关的.重写是为了动态连接.
3. 抽象类(被abstract修饰的类.)
   1. 目的是让子类继承,抽象类不能实例化,无法创建对象.
   2. 抽象类的构造函数不能被继承,用于对子类进行初始化. (先执行父类构造,再执行子类构造.)
   3. 抽象类的方法必须是public / protected
   4. 抽象类不一定含有抽象方法.
   5. 子类继承抽象类必须实现抽象方法,不实现的话方法也要设为abstract 让孙子来实现.
4. 接口(interface)
   1. 变量必须是 public static final 的. 可以不写,默认设置.
   2. 方法必须是 public abstract 可以不写,默认设置 . 方法不能有实现.是抽象方法.
5. 抽象类和接口的区别
   1. 抽象类可以提供方法的实现,而接口中只能有抽象方法.
   2. 抽象类的成员变量可以是各种类型的,而接口只能是public static final的
   3. 接口中不能含有静态代码块和静态方法,抽象类中可以.
   4. 一个类只能继承一个抽象类,却可以实现多个接口.
6. static final 常量规范是大写. 类变量是类共享的,实例变量一个对象一份.
7. 局部变量(ex.方法内的) 必须有一个明确的赋值. 类变量和实例变量都有默认值.
8. 为了防止实例变量和局部变量产生冲突, 单独**this**表示当时实例对象,实例变量属于实例对象。

**错误与异常**

1. Throwable 两个重要子类: Exception(异常),Error(错误)
2. Error 错误,我无需处理. 描述了Java运行时系统的内部错误和资源耗尽错误。是程序无法处理的错误，例如JVM需要的内存不足，出现内存爆了或者类找不到.
   1. VirtulMachiineError
      1. OutOfMemoryError内存爆了.
      2. StackOverflowError栈爆了
   2. AWTError 图形界面错误.
3. Exception 是程序本身可以处理的异常
   1. RuntimeException 运行时异常 编译器不检查,我也不用try catch 由程序错误导致的异常。
      1. ★错误的类型转换
      2. ★数组访问越界
      3. ★访问空指针
      4. ★等等
   2. 编译器可检查的异常,如IOException：要么try-catch捕获,要么throw抛出.否则编译不通过.
      1. EOFException
      2. FileNotFoundException
      3. ☆试图在文件尾部后面读取数据
      4. ☆试图打开一个错误格式的URL
      5. ☆试图根据一个给定的字符串查找CLASS对象，而这个字符串表示的类并不存在
4. 可查异常与不可查异常
   1. 可查异常: 除了RuntimeException及其子类以外，其他的Exception类及其子类异常.遇到了必须try-catch捕获,要么throw抛出.否则编译不通过.
   2. 不可查异常(编译器不要求强制处置的异常):包括运行时异常（RuntimeException与其子类）和错误（Error）
5. 异常处理机制:
   1. 捕获异常: try、catch 和 finally
   2. 抛出异常: throws Exception
      1. 在方法体{ throw new RuntimeException("1234");} 创建异常类的实例对象,用throw抛出,
      2. 可以在方法名后public void method() throws Exception{…}.抛了异常给其他人.当别人调用这个方法的时候,就要把调用方法的语句包在try-catch 中捕获 不然不行.
6. Throwable类中的常用方法
   1. *getCause() 返回抛出异常的原因。如果 cause 不存在或未知，则返回 null。*
   2. getMessage()：返回异常的消息信息。
   3. printStackTrace()：对象的堆栈跟踪输出至错误输出流，作为字段 System.err 的值。控制台输出.
7. 可以自定义异常。用户自定义异常类，只需继承Exception类即可。
8. 异常流开发模式: 在自己提供的API底层代码throws, 然后外围代码捕获,这样调用方就知道出了什么问题.
9. final/finally/finalize
   1. final 用于声明属性，对象，方法和类，分别表示属性不可变，方法不可覆盖，类不可继承。

final声明对象表示的是对应的引用不可变，但是对象的值可以变，参照如下范例：

* + 1. final StringBuffer buffer = new StringBuffer();
    2. buffer.append("123");
  1. finally 是异常处理语句结构的一部分，表示总是执行。
     1. finally外的return语句并不是函数的最终出口，如果有finally语句，在这个return之后还会执行finally
     2. return的值会暂存在栈里面，等待finally执行后再返回,
        1. 所以说如果return num在try中,或者finally之前,finally去改变num的值,return的值是不会变的.
        2. 如果return 引用(对象), 那么在finally中改变引用的属性,是会改变return内容的.
     3. finally中有return,直接执行这个return 之前保存的值不用.
  2. finalize:

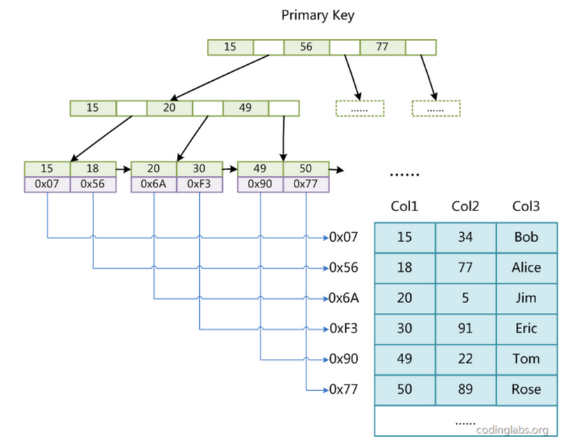
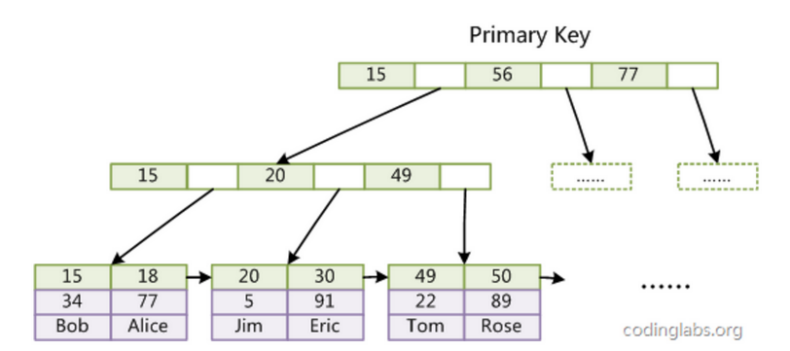
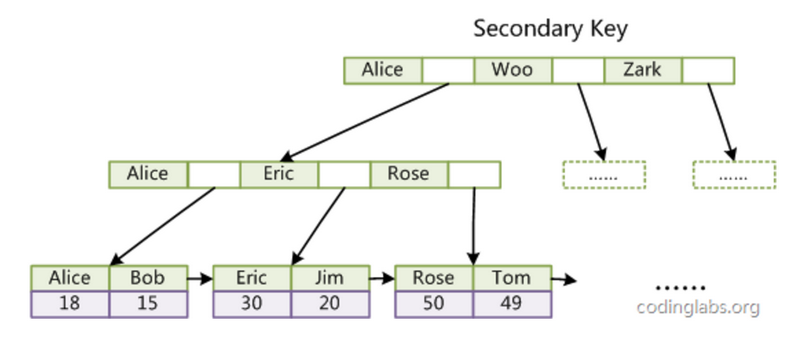
finalize 是Object类的一个方法，在*垃圾收集器执行*的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等。如果必须保证采用特定的顺序，则必须提供自己的特有清理方法。实际项目不会使用. finalize起什么作用可以作为面试题

1. finally与return总结
   * + - 1. 如果finally中有return,返回这个return,之前保存的值启用.
         2. 如果finally中没有return语句，也没有改变要返回值，则执行完finally中的语句后，会接着执行try中的return语句，返回之前保留的值。

# 第四章:数据库基础

1. 实际项目中不会在数据库中设置外键.而是在逻辑中实现相关功能.
2. 索引是对数据库表的一列或者多列(可以用在多个属性上,即复合键)进行排序的一种结构, 类似一本书的目录.通过建立索引文件,可以更快的找到信息
3. 复合键(组合键): 将多个列作为一个索引键.
4. 事务: 被绑定在一起作为一个逻辑工作单元的 SQL 语句组. 任何一个语句失败,整个事务都失败,事务会回滚到操作之前的状态.
5. 锁是实现事务的关键，锁可以保证事务的**完整性**和**并发性.**
6. 数据库事务的四个特性ACID
   1. 原子性**A**tomicity:操作要么全完成,要么全不完成.
   2. 一致性**C**onsistency:事务开始和事务结束之后,数据库的完整性约束没有被破坏
   3. 隔离性**I**solation;(串行化) 在一个时间内,只有一个事务在使用系统. 隔离级别是问点
   4. 持久性**D**urability:事务完成之后所做的更改持久的保存在数据库,不会被回滚.
7. 触发器:不怎么考察了. 触发器是一个特殊的存储过程,即实现某种功能的一系列的SQL语句. 触发器简单，但是不利于后期维护，互联网项目都是使用自写业务逻辑(原因是高并发和大数据量情况下数据库一般都是瓶颈，最难被扩充).
8. 索引
   1. 索引是一种数据结构,可以快读查询和更新数据表中的数据. 底层是B树 B+树 (面试3连问.)
   2. 为表设置索引要付出代价的：一是增加了数据库的存储空间(典型的空间换时间.)，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。
   3. 一般在经常用到的where 子句后面创建索引,加快条件的判断速度.某些列 如 text,等数据量很大,或者很少数据值的列,不要加索引.没有意义.
   4. 如果修改需求远远大于查找需求的话,不应该添加索引.
   5. 索引分 唯一索引,主键索引,普通索引.
   6. *创建索引应指定索引的列为Not Null,否则很难进行查询优化.以空串 0 等代替.*
9. drop,delete与truncate的区别
   1. drop直接删掉表 。
   2. truncate删除表中数据，再插入时自增长id又从1开始 。
   3. delete删除表中数据，可以加where字句。
10. SQL语句,多表关联等等再看Mysql的手册.
11. 数据库优化
    1. SQL语句优化: 避免在where子句中使用!=或者<>操作,对字段进行null值判断,这将放弃索引使用全盘扫描.
    2. 互联网项目中，尽量避免使用存储过程，尽量避免多表关联查询.走SQL的话,必须得是单表.再逻辑里面做多表关联.
    3. 索引优化:看看有没有走索引,没加索引加上他
    4. 数据库结构优化.

**Mysql索引**

1. 索引的本质是 一种数据结构. (在学习任何知识点时,除了要知道是什么,更要知道为什么这么设计,我来设计会如何??)
2. Mysql的存储引擎
   1. MyISAM
      1. 不支持事务,不支持外键.但是访问速度快. *对事务完整性没有要求或者以SELECT、INSERT为主的应用基本都可以使用这个引擎来创建表。*
   2. InnoDB
      1. 支持事务,可以提交回滚,比较安全.效率较MyISAM差,占用更多空间来保留数据和索引.
   3. 区别:
      1. MyISAM是非事务安全型的.,InnoDB是事务安全型的
      2. MyISAM锁的粒度是表级,InnoDB支持行锁.
      3. MyISAM支持全文索引, InnoDB不支持全文索引
      4. MyISAM简单,效率高,小型引用可以考虑使用
      5. MyISAM表保存成了文件的形式,跨平台数据转移时更方便.
      6. InnoDB比MyISAM更安全.可以安全将表从非事务表转成事务表. alter table tablename type=innodb
3. 索引的数据结构: B+Tree
   1. B-Tree
      1. 类似平衡二叉树,但是每个节点可有多个子节点. 每个节点存储一个二元组[Key和data]
      2. 叶子节点也有key 还有两个null指针. 所有的叶子的深度一致,等于树高.
   2. B+Tree
      1. 和B-Tree相比,非叶子节点不包含数据.
   3. MySql的B+Tree 即 带有顺序访问指针的B+Tree
      1. 每个叶子都增加一个指向相邻叶子的指针,提高了区间查询的效率.
4. 为什么B+Tree作为了Mysql的物理存储方式?
   1. 内部节点不包含数据,在同样大的盘块中能够容纳更多的单位索引结构,一次IO读入内存的关键字信息索引更多.所以相对IO读写次数也更少.
   2. 这是B+TREE更适合做文件索引.
5. 两种引擎的索引存储机制 无论什么引擎,MYSQL就是用的B+Tree
   1. MyISAM在B+Tree的叶子节点上存储这条记录的内存地址.索引文件和数据文件是分开存放的[非聚集索引].主索引和辅助索引(普通索引)在结构上没有区别,只是主索引的key要唯一,辅助索引key可重复.
   2. InnoDB的数据文件本身就是索引文件. 叶子中直接存储数据文件.[聚集索引]
   3. InnoDB中,主键索引的key即主键值,data即该行所有的属性值.辅助索引的key就是索引列的列值,叶子data是主键值.
   4. MyISAM 的key也是列值(索引所在的列的值),但data是地址,指向真实值的内存空间.
      1. 主键索引和辅助索引的结构是一样的.
      2. 主键key不可重复 (主键值唯一 学号id),辅助的key可重复.(不是主键所以值可以不唯一 有多个人叫Bob))
6. 聚集索引/非聚集索引(聚簇索引,非聚簇索引)

聚簇索引不会回表. 即不会去访问基表直接从叶子节中拿数据.InnoDB的主键索引是聚簇索引.

1. 辅助索引,主键索引.
   1. 对表的主键加索引,就是主键索引.对表的非主键加索引,就是普通索引或者说辅助索引.
   2. InnoDB的辅助索引B+Tree的key是列值,叶子的data记录的是该属性的主键值. 比如name索引的叶子data保存的是主键索引里面对应的key(指向叶子data是张三的key),key保存的是张三.
2. InnoDB不建议主键字段过长,因为辅助索引最后都是存储的主键值,过长的主键导致辅助索引文件过大.
3. 主键最好是单调的,因为B+Tree在插入新记录时,若记录不单调,为了保持特性,会频繁的分裂调整.十分低效.
4. InnoDB如果涉及到事务并发,要按照统一的加锁顺序,避免在不同的事务给主键索引和辅助索引分别加锁.会引起死锁.
5. InnoDB的行锁 只能作用在加了索引的字段上面.

**索引相关的补充**

1. 索引类型
   1. 哈希索引:
      1. 哈希表: 键值存储,输入key,得value.思路很简单: 把值放在数组里,用一个哈希函数把key换算成一个确定的位置,然后把value放在数组的这个位置,如果出现哈希冲突,就采用拉链法解决(接链表)
      2. 因为哈希表中的数据不是有序的,因此不适合做区间查询,而等值查询(key取value)的速度很快.
   2. 有序数组
      1. 将数据排好序后放进数组里.等值查询和范围查询性能都很好.
      2. 可以通过二分查找O(logN) 快速找到数据.
      3. 插入数据的成本较高,只适用于静态存储引擎.一旦建立不再修改那种的.
   3. B+树
      1. 最终的叶子带有顺序访问指针,可以横着(区间)查找. 效率高.
2. 在 InnoDB 中，表都是根据主键顺序以索引的形式存放的，这种存储方式的表称为索引组织表,每一个索引在InnoDB中对应一棵B+树.
3. 在这个表里面,会自动有一个主键索引,一个普通索引,所以对应两个B+tree..

CREATE TABLE T(

id int primary key,

k int not null,

index (k) //这一句就是加了个普通索引.

)engine=InnoDB;

1. 如果不主动创建主键，那么Innodb会选着一个不包含null值的唯一索引作为主键索引。如果连唯一索引都没有，Innodb会为该表默认生成一个6字节的rowid作为主键索引。一般情况下，建议创建一个**自增主键**，这样非主键索引占用的空间最小。
2. 基于主键索引和非主键索引的区别
   1. 非主键索引有一个回表操作.所以要多扫描一棵索引树.
3. 联合索引: 联合索引是指对表上的多个列进行索引。

// 表

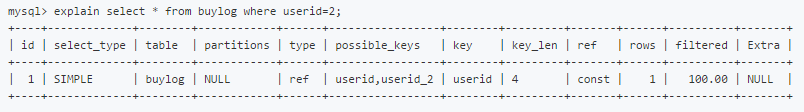
CREATE TABLE buylog(userid int not null, buy\_date DATE)ENGINE=InnoDB;

// 添加索引

alter table buylog add index(userid);//主键索引

alter table buylog add index(userid, buy\_date);//联合索引

1. 通过 explain来查看Mysql 的sql执行计划.



possible\_keys: 可能用到的索引

key是最终用到的索引. 是由优化器的最终选择出来的. 根据实际情况选择最有效率的方式.

1. 当数据量很大,在男女字段上加了索引,男女的区分度很低.查找男的结果,作为辅助索引,要回表,再去在主键索引里查一遍,效率不如全文查找效率高.所以**用索引不一定会比全文查找更有效.**
2. 联合索引的key 是包含联合的两个列的内容的,所以比较占用空间. 每一页能取的记录更少.
3. 联合索引(A,B)的key 存储是有序的,先以A字段排序,A字段一样时,再以B字段排序.

select \* from buylog where userid=1 order by buy\_date desc limit 2;

所以,当使用order by排序的时候,使用联合索引是最快的.

1. 最左前缀原则:

联合索引(A,B) 只根据A查找的话,也是能用到这个联合索引的.

但是先查B,不能用到联合索引. 因为确定B之后,想确定A,要遍历很多的元素,最差都要全表遍历了.而先确定A再确定B,非常的轻松.

联合索引是包含单个索引的记录的.

因为联合索引存储的时候是有一个排序的,所以才会很快.不满足最左前缀原则的话,就利用不到这个排序了.

**只要满足最左前缀，就可以利用索引来加速检索。这个最左前缀可以是联合索引的最左 N 个字段，也可以是字符串索引的最左 M 个字符。where name like ‘张%’**

**所以 当有了 (a,b) 联合索引的话,就没必要建立 a 索引了.**

1. **建立联合索引把使用频率高的放在左边, 保证高复用性. 但是建的多了之后 意义就不大的.**
2. **对于联合索引(a, b, c)，下列语句同样可以直接通过联合索引得到结果：**

**select ... from xxx where a=xxx order by b;**

**select ... from xxx where a=xxx and b=xxx order by c;//使用了联合索引(a,b,c)**

1. **而下列语句则不行，需要执行一次filesort排序操作。**

**select ... from xxx where a=xxx order by c;**

1. **覆盖索引优化: 不用回表的操作,就是覆盖索引. 当select 的值是主键值或者辅助索引列的值,数据存储在辅助索引树上时,不用回表 效率会高. 查询主键的话,因为普通索引的叶子只存储主键,所以一页查询的数量就多.效率会比主键索引高. count(\*)会用到.**
2. **索引下推优化 或称 ICP 优化 ( mysql5.6以及之后 ):**

select \* from tuser where name like '张 %' and age=10 and ismale=1;

这里用到了模糊查询张% 这样在联合索引中,利用最左前缀原则,只会找到名字第一个字是”张”的记录,利用不到后面的age和ismale字段,找到张之后立刻就回表然后再在主键索引里面再进行筛选. 效率较低.

当没有索引下推优化 ICP优化:首先根据索引在辅助索引找主键id,再回表查找位于主键索引的全部适配记录，然后再根据 where条件来过滤记录；

而有了索引下推优化之后, 在取出索引的同时，判断是否可以进行where条件过滤，也就是说提前执行where的部分过滤操作，

17. 补充: 普通索引也叫辅助索引,

唯一索引本质上是辅助索引,不过列加了唯一性约束.

联合索引即普通索引包含了多个列.

# 第五章: 事务,乐观锁与悲观锁

1. 直接会问并发事务引起的问题, 数据库的隔离级别, Mysql的隔离级是什么. 80%会考.
2. 数据库事务:
   1. 单个完整逻辑的一系列操作. 要么完全执行,要么完全不执行.保证了事务的原子性.
3. ACID 四原则
   1. 原子性(Atomicity) 操作要么执行,要么不执行
   2. 一致性(Consistency) 数据一致性,操作的时候内部数据必须是正确的.
   3. 隔离性(Isolation) 不同事务之间执行是隔离的,互不影响.
   4. 持久性(Durability) 事务执行之后,数据要能存储在数据库中.
4. 隔离级别 (两个事务之间互相影响的程度) 实习问题超高频
   1. **读未提交** Read Uncommitted
      1. 事务可以读到其他事务的数据, 别的事务正在修改 但没提交,我能读到
      2. 产生问题: **脏读**
         1. 一个事务处理过程里处理了另一个未提交的事务中的数据.
   2. **读已提交** Read Committed
      1. 解决了脏读问题.
      2. 事务只能读到别人提交了的数据更改. 如果别人没提交,读到的是原值.
      3. 产生问题: **不可重复读**
         1. 在同一个事务里,前后查询的结果不一致(我查一次,别人改了提交了,我再查,不一样了.)
         2. 可以理解成对单条记录查询结果不一致.
         3. (对应行锁)
   3. **可重复读** Repeatable Read
      1. 解决了脏读和不可重复读的问题.
      2. 在我事务之内,读到的一条数据应该是一致的,不管别人是否更改提交.(数据改了,我读的还是原值 吊)
      3. 产生问题: **幻读**
         1. 一个事务读到了别的事务插入的数据.
         2. 对应区间锁(在插入的时候要等待别的事务释放)
   4. **串行化** Serializable
      1. 解决了所有并发问题. 但一般不用这个隔离级别.
      2. 所有事务都串起来了,一个事务执行完才能接着执行下一个事务.
5. 隔离级别的具体实现
   1. 锁: 不会对整个数据库都加锁,而是只锁住需要访问的数据项. .锁分为两种: 共享锁(读锁) 互斥锁(写锁)
   2. MVCC(多版本并发控制): 维护多个版本的数据,允许事务在数据库被其他事物更新时对旧版本数据的读取.
6. Mysql默认隔离界别: 可重复读 Oracle默认级别是: 读已提交.

**乐观锁和悲观锁 锁相关**

1. 悲观锁: 上物理锁,真的上锁. 每次拿数据都要加锁.
   1. 认知中的行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁，这些锁统称为悲观锁
2. 乐观锁: 逻辑上加锁,不是真的加锁. 每次拿数据的时候不认为别人会更改. 适用于多读应用,提高吞吐量.
   1. 比如: 更新时判断一下有没有其他人来动这个数据. 没有就更新成功.
   2. 版本号机制来实现. 每次更新都有一个号,力度最大.
3. 经常发生冲突,建议使用悲观锁, 使用乐观锁会一直导致更新失败,反倒系统性能降低.
4. ABA问题, 即乐观锁使用时,判断之前A被事务改成B,又改回来了.
5. 乐观锁的力度很重要, 高并发秒杀时, 可以设置库存数>0 而不是版本号. 可以避免提示操作失败,提高吞吐量.
6. 受到 fail-fast思路影响,悲观锁已经非常少见了,悲观锁有锁等待,效率不高.
   1. fail-fast是一种架构思路 快速失败, 比如在秒杀项目里,如果能实现高效率,那么高失败率也是可以接受的.
   2. fail-fast 是一种乐观锁的思路.
7. 注册判断用户名重复可以用悲观锁了.
8. 判断使用悲观锁还是乐观锁
   1. 响应速度: 非常高的响应速度,采用乐观锁, 成功执行,不成功失败,无需进行锁等待.
   2. 冲突频率: 如果冲突频率高,采用悲观锁,保证成功率. 冲突过多,失败太多,乐观锁多次重启 代价较大.
   3. 重试代价: 如果重试的代价很大,用悲观锁.
9. 悲观锁实现

set autocommit=0; 开启事务, 不写这个的话,mysql默认是自动提交的.

begin; 开始事务 后面开始写sql,这些sql是被锁住的.

select status from items where id=10000 **for update**;

当for update的字段为索引或者主键的时候，只会锁住索引或者主键对应的**行**。

而当for update的字段为普通字段的时候，Innodb会锁住整张表。

insert into orders (id,item\_id) values (null,10000);

update items set status=2 where id=10000;

commit; 提交事务 之后这些更新才会生效.

1. 可重入锁, 我在锁住你的时候,我是可以对你进行任何操作的, 只有其他事务才会等待..
2. 乐观锁实现为：

InnoDB中,商品表items表中有一个字段status，status=1:该商品未被下单，status=2表示该商品已经被下单

下单操作：

//查询出商品信息,version即版本控制

select (status,version) from items where id=#{id}

//得到version之后,修改商品status为2,即标注下单. 如果version不同了,那么就无法update.

update items set status=2,version=version+1 where id=#{id} and version=#{version};

若不用版本,挑选以库存数作为乐观锁，则失败率会小很多，如下（用乐观锁方式扣减库存，确保有库存即可）：

update item

set

quantity=quantity-#sub\_quantity#

where

item\_id = #id#

and quantity-#sub\_quantity# > 0

1. MyISAM引擎采用表锁, InnoDB表锁行锁都可以,默认行锁.
   1. 表锁,开销小,加锁快,不会死锁.发生锁冲突概率高,并发度低
   2. 行锁,开销大,加锁慢,会死锁,冲突概率最低,并发度最高
   3. 页面锁(BDB存储引擎),开销和时间冲突,并发都在表行之间,会死锁 不常用
2. 表锁适合查询,只应用在按照索引条件更新数据的项目, 如web.
3. 行锁适合并发更新多,少量查询的引用,如在线事务处理(OLTP)系统
4. InnoDB的行锁
   1. 共享锁(S): 我读的时候,你可以读,但不能写.
   2. 排它锁(X): 是我写的时候,你不能读也不能写.
      1. select \* from table where ? for update;
      2. insert into table values (…);
      3. update table set ? where ?;
      4. delete from table where ?;
   3. 其实就是MyISAM的读锁和写锁.只是应用对象不同.
5. InnoDB的表锁
   1. 意向共享锁(IS),即准备给数据行加共享锁之前,必须先取得表的IS锁.
   2. 意向排他锁(IX),即…………………….加排他锁…………………………………..IX锁.
   3. 防止遍历时效率低下. 当别人要遍历表, 看到意向锁,就知道某一行在被操作,,他就先不要马上遍历这个表
6. select … lock in share mode：加 S 锁 . 容易死锁,实际项目基本不使用共享锁.
7. select … for update：加 X 锁 最好只用这个.
8. SELECT quantity FROM products WHERE id=3; //快照读, 这一句没有加锁,读到的只是快照,可能已经变了.

UPDATE products SET quantity = 1 WHERE id=3;// 这时再更新数据,如果高并发秒杀,库存一切都乱套了.

1. 快照读：读取的是快照版本，也就是历史版本。不用加锁.普通的SELECT就是快照读，
2. 当前读：读取的是最新版本。UPDATE、DELETE、INSERT、SELECT … LOCK IN SHARE MODE、SELECT … FOR UPDATE是当前读。读取之后，还需要保证其他并发事务不能修改当前记录, 对记录加锁.
3. 可以多刷一些SQL的题目.多看一些相关的知识. 复杂语句 堆叠等.

# 第六章: Mysql锁进阶

1. 正因为MyISAM只支持表锁,InnoDB支持表锁和行锁,所以InnoDB才能支持事务.
2. MyISAM表锁
   1. 表共享读锁 **Table Read Lock**
   2. 表独占写作 **Table Write Lock**
3. InnoDB的锁
   1. 事务隔离机制实现方式有两种
      1. 读数据前,对其加锁.
      2. 不加任何锁,通过一个机制生成一致性数据快照.并用快照提供一定级别的一致读取,从用户角度,好像是数据库可以提供同一数据的多个版本. 这就是多版本并发控制MVCC.
      3. 一般我们读的都是MVCC提供的快照,无需加锁.
4. 只有通过索引条件检索数据(where跟着索引)，InnoDB才使用行级锁，否则，InnoDB将使用表锁！
5. MySQL InnoDB行锁是通过给索引上的索引项加锁来实现的
6. Oracle的行锁是通过在数据块中对相应数据行加锁来实现的。
7. 行锁+间隙锁 Next-Key
   1. 在可重复读隔离级别下的一种锁，目的是防止幻读(插入数据)
   2. 除了行锁本身,还要把数据附近的一个区间都要锁住. 这样别人就插不进来了
   3. 高并发时,间隙锁会造成严重的锁等待
   4. 并发插入比较多的应用,尽量使用相等条件访问更新数据，避免使用范围条件。
   5. 多数时候为了避免出现Next-Key锁，会直接放弃RR可重复读这个MySql默认隔离级别而采用RC读已提交。
8. 一般间隙锁伴随幻读来问, 一定注意 在可重复读隔离级别下才有间隙锁. 在读未提交和读已提交是没有的.要严谨.
9. InnoDB的行锁是基于索引实现的，如果不通过索引访问数据，InnoDB会使用表锁。
10. 更新辅助索引值时,不仅要主键索引加锁,辅助索引也要加锁.
11. gap锁(间隙锁) 只在可重复读隔离级别存在.
    1. where num=10 而数据列里面 有8 和12 所以就会把8与12这两行锁住.
12. 串行,隔离级别太高, 效率太低了. 读未提交,隔离级别太低了.也不怎么用.
13. 事务A在for update锁定id=1的记录时，实际上只是去主键索引上查找id=1的记录并锁定
14. 事务B的update操作，针对的是二级索引，那么MySQL会先查找二级索引的age=6的所有记录，锁定，然后再去通过主键值找到对应主键索引的记录，并锁定！
15. 但是步骤1里的id=1的这条记录实际上已经被锁定了，所以事务B只能等待事务A释放锁
16. 事务A进行的delete操作，会先去主键索引查找记录，锁记录（在第一步实际上已锁定），然后去二级索引查找记录，并锁定！
17. 但是事务B已在步骤2中锁定了二级索引中的记录，所以事务A只能等待事务B释放锁
18. 由此，造成了事务A和事务B互相等待，死锁形成。

一些避免此类死锁的方法：

* 一定要注意加锁的顺序性！
* 针对同一张表，尽量只对其主键索引进行条件查询的操作
* 慎用for update！ 而用乐观锁, redis缓存,消息队列等形式.

1. 1
2. 1
3. 1
4. 1
5. 11
6. 1
7. 1
8. 1
9. 1
10. 11
11. 1
12. 1
13. 1
14. 1
15. 1
16. 11