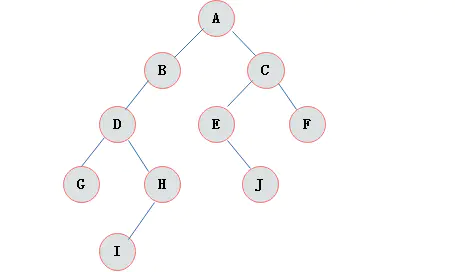
**Java知识点总结**

# Java基础

## 数据结构

### 二叉树

参考地址：https://www.jianshu.com/p/bf73c8d50dc2



#### 特点

1. 每个结点最多有两颗子树，所以二叉树中不存在度大于2的结点。
2. 左子树和右子树是有顺序的，次序不能任意颠倒。
3. 即使树中某结点只有一棵子树，也要区分它是左子树还是右子树

#### 性质

1. 在二叉树的第i层上最多有2i-1 个节点 。（i>=1）
2. 二叉树中如果深度为k,那么最多有2k-1个节点。(k>=1）
3. n0=n2+1 n0表示度数为0的节点数，n2表示度数为2的节点数。
4. 在完全二叉树中，具有n个节点的完全二叉树的深度为[log2n]+1，其中[log2n]是向下取整。
5. 若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点有如下特性：

(1) 若 i=1，则该结点是二叉树的根，无双亲, 否则，编号为 [i/2] 的结点为其双亲结点;

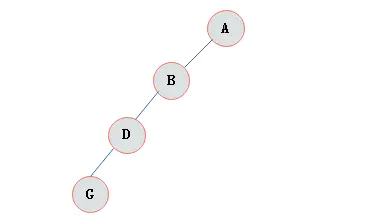
(2) 若 2i>n，则该结点无左孩子， 否则，编号为 2i 的结点为其左孩子结点；

(3) 若 2i+1>n，则该结点无右孩子结点， 否则，编号为2i+1 的结点为其右孩子结点。

#### 特殊二叉树

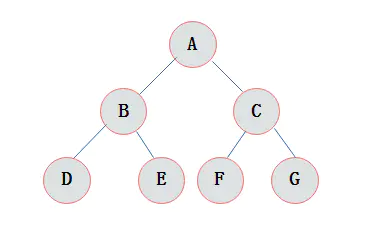
##### 斜树

所有的结点都只有左子树的二叉树叫左斜树。所有结点都是只有右子树的二叉树叫右斜树。这两者统称为斜树。



##### 满二叉树

在一棵二叉树中。如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。  
满二叉树的特点有：  
1）叶子只能出现在最下一层。出现在其它层就不可能达成平衡。  
2）非叶子结点的度一定是2。  
3）在同样深度的二叉树中，满二叉树的结点个数最多，叶子数最多。

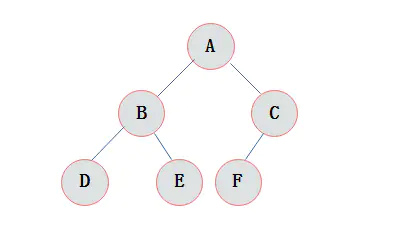


##### 完全二叉树

对一颗具有n个结点的二叉树按层编号，如果编号为i(1<=i<=n)的结点与同样深度的满二叉树中编号为i的结点在二叉树中位置完全相同，则这棵二叉树称为完全二叉树。

图3.5展示一棵完全二叉树。

不知道这个i跟那个i有什么区别？



**特点**：  
1）叶子结点只能出现在最下层和次下层。  
2）最下层的叶子结点集中在树的左部。  
3）倒数第二层若存在叶子结点，一定在右部连续位置。  
4）如果结点度为1，则该结点只有左孩子，即没有右子树。  
5）同样结点数目的二叉树，完全二叉树深度最小。  
**注**：满二叉树一定是完全二叉树，但反过来不一定成立。

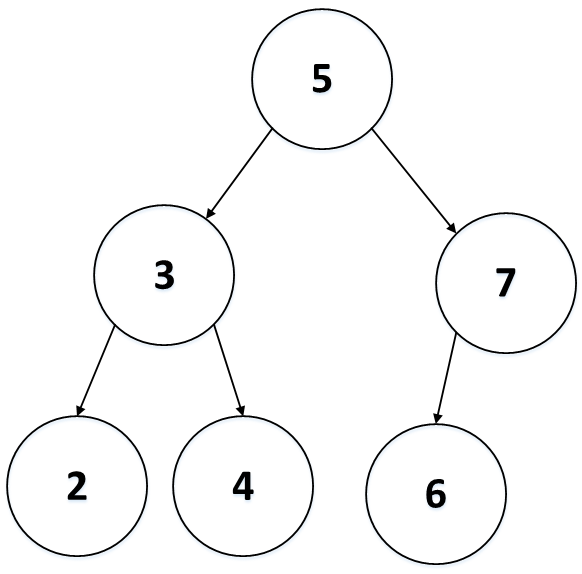
**根据存储数据结构的划分可以划分为二分查找树，平衡二叉树。参考文章**

**https://www.cnblogs.com/sgatbl/p/9426394.html**

##### 二分查找树

性质

二叉查找树实际上是数据域有序的二叉树，即对树上的每个结点，都满足其左子树上所有结点的数据域均小于或等于根结点的数据域，右子树上所有结点的数据域均大于根结点的数据域。

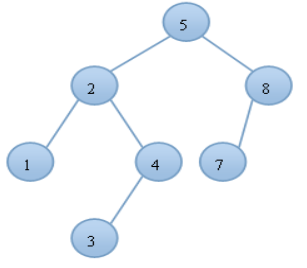


例如：2<3 放入左子树，4>3放入右子树

##### 平衡二叉树

平衡二叉树是由前苏联的两位数学家G.M.Adelse-Velskil和E.M.Landis提出，因此一般也称作AVL树，AVL树本质还是一棵二叉查找树，只是在其基础上增加了“平衡”的要求。所谓平衡是指，对AVL树的任意结点来说，其左子树与右子树的高度之差的绝对值不超过1，其中左子树与右子树的高度因子之差称为平衡因子。

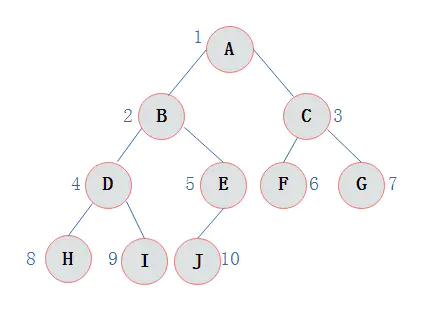
通常来讲就是左子树的高度与右子树的高度差值不能超过1。

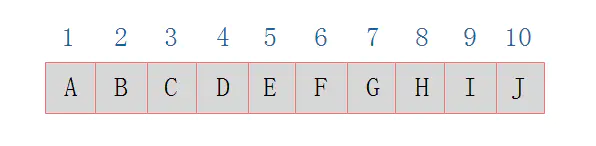


#### 二叉树的存储结构

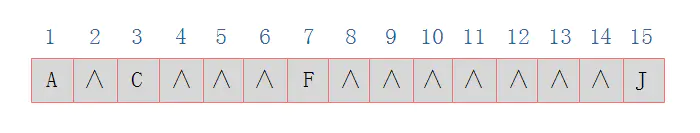
###### 顺序存储

二叉树的顺序存储结构就是使用一维数组存储二叉树中的结点，并且结点的存储位置，就是数组的下标索引





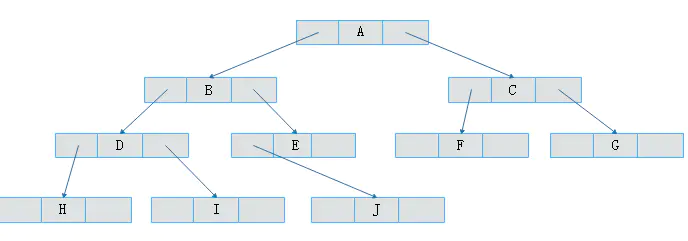
顺序存储适用与完全二叉树，对于其它二叉树结构会存在空间的浪费



###### 二叉链表

二叉树的每个结点最多有两个孩子。因此，可以将结点数据结构定义为一个数据和两个指针域。





### 红黑树

#### 定义

R-B Tree，全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

#### 性质

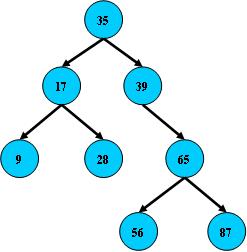
1. **每个节点或者是黑色，或者是红色。  
   （2）根节点是黑色。  
   （3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]  
   （4）如果一个节点是红色的，则它的子节点必须是黑色的。  
   （5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。**



### Btree（二叉搜索树）

#### 性质

* 1. 所有非叶子结点至多拥有两个儿子（Left和Right）；
  2. 所有结点存储一个关键字；
  3. 左边比自己小，右边比自己大；



### B-Tree（平衡多路查找树）

B-Tree是为磁盘等外存储设备设计的一种平衡查找树。

一棵m阶的B-Tree有如下特性：

1. 每个节点最多有m个孩子。

2. 除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子。

3. 若根节点不是叶子节点，则至少有2个孩子

4. 所有叶子节点都在同一层，且不包含其它关键字信息

5. 每个非终端节点包含n个关键字信息（P0,P1,…Pn, k1,…kn）

6. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1

7. ki(i=1,…n)为关键字，且关键字升序排序。

8. Pi(i=1,…n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于ki，但都大于k(i-1)

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个3阶的B-Tree：



索引

每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

模拟查找关键字29的过程：

根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】

比较关键字29在区间（17,35），找到磁盘块1的指针P2。

根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】

比较关键字29在区间（26,30），找到磁盘块3的指针P2。

根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】

在磁盘块8中的关键字列表中找到关键字29。

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。B-Tree相对于AVLTree缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

### B+Tree

B+树索引是B+树在数据库中的一种实现，是最常见也是数据库中使用最为频繁的一种索引。B+树中的B代表平衡（balance），而不是二叉（binary），因为B+树是从最早的平衡二叉树演化而来的。在讲B+树之前必须先了解二叉查找树、平衡二叉树（AVLTree）和平衡多路查找树（B-Tree），B+树即由这些树逐步优化而来。B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

#### B+Tree相对于B-Tree有几点不同：

1. 非叶子节点只存储键值信息。
2. 所有叶子节点之间都有一个链指针。
3. 数据记录都存放在叶子节点中。

将上一节中的B-Tree优化，由于B+Tree的非叶子节点只存储键值信息，假设每个磁盘块能存储4个键值及指针信息，则变成B+Tree后其结构如下图所示：



数据库中的B+Tree索引可以分为聚集索引（clustered index）和辅助索引（secondary index）。上面的B+Tree示例图在数据库中的实现即为聚集索引，聚集索引的B+Tree中的叶子节点存放的是整张表的行记录数据。辅助索引与聚集索引的区别在于辅助索引的叶子节点并不包含行记录的全部数据，而是存储相应行数据的聚集索引键，即主键。当通过辅助索引来查询数据时，InnoDB存储引擎会遍历辅助索引找到主键，然后再通过主键在聚集索引中找到完整的行记录数据。

## 集合

### Map

#### HashMap

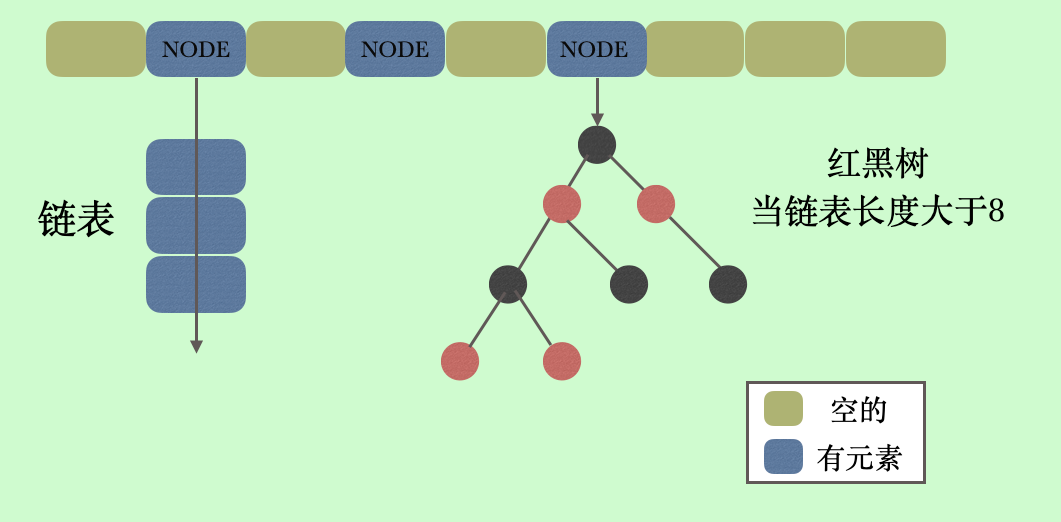
<https://mp.weixin.qq.com/s/keA-xNOHxMUPAFxKnna-Bg>

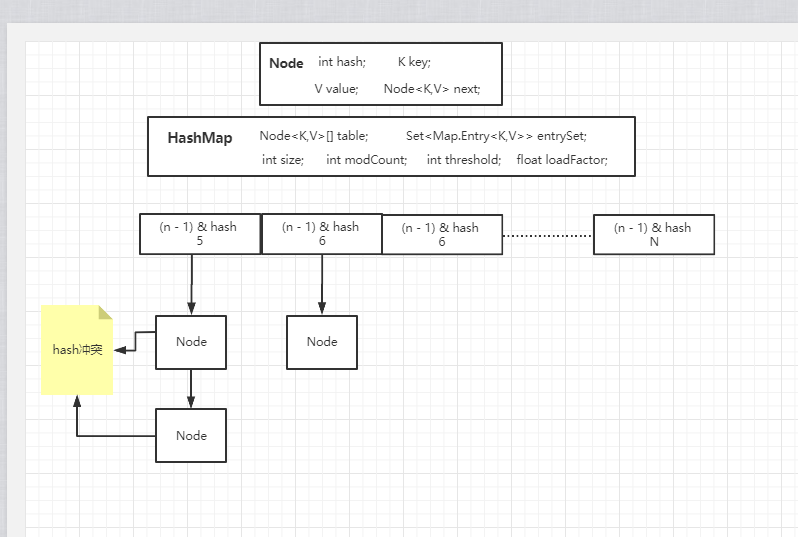
hashMap面试题

##### Jdk1.8

###### 存储结构

使用的是数组+链表，红黑树





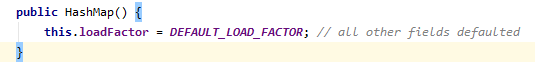
###### 源码分析

HashMap定义属性

**transient** Node<K,V>[] **table**;  
**transient** Set<Map.Entry<K,V>> **entrySet**;  
**transient int size**;  
**transient int modCount**;**int threshold**;  
**final float loadFactor**;

初始化一个map





1. 加载因子赋值为0.75
2. 创建一个空的实例，并不会初始化数组长度

放入数据



**public** V put(K key, V value) {  
 **return** putVal(*hash*(key), key, value, **false**, **true**);  
}

/\*\*

\* onlyIfAbsent如果为true，则不更改现有值

\* evict如果为false，则表处于创建模式

\*/

**final** V putVal(**int** hash, K key, V value, **boolean** onlyIfAbsent,  
 **boolean** evict) {  
 Node<K,V>[] tab; Node<K,V> p; **int** n, i;  
 **if** ((tab = **table**) == **null** || (n = tab.**length**) == 0)  
 n = (tab = resize()).**length**;  
 **if** ((p = tab[i = (n - 1) & hash]) == **null**)  
 tab[i] = newNode(hash, key, value, **null**);  
 **else** {  
 Node<K,V> e; K k;  
 **if** (p.**hash** == hash &&  
 ((k = p.**key**) == key || (key != **null** && key.equals(k))))  
 e = p;  
 **else if** (p **instanceof** TreeNode)  
 e = ((TreeNode<K,V>)p).putTreeVal(**this**, tab, hash, key, value);  
 **else** {  
 **for** (**int** binCount = 0; ; ++binCount) {  
 **if** ((e = p.**next**) == **null**) {  
 p.**next** = newNode(hash, key, value, **null**);  
 **if** (binCount >= ***TREEIFY\_THRESHOLD*** - 1) *// -1 for 1st* treeifyBin(tab, hash);  
 **break**;  
 }  
 **if** (e.**hash** == hash &&  
 ((k = e.**key**) == key || (key != **null** && key.equals(k))))  
 **break**;  
 p = e;  
 }  
 }  
 **if** (e != **null**) { *// existing mapping for key* V oldValue = e.**value**;  
 **if** (!onlyIfAbsent || oldValue == **null**)  
 e.**value** = value;  
 afterNodeAccess(e);  
 **return** oldValue;  
 }  
 }

}

定义了一些变量

Node<K,V>[] tab; Node<K,V> p; **int** n, i;

给table一个初始值[]

n = (tab = resize()).**length**;

tab = resize();

**final** Node<K,V>[] resize(){

}

Resize()方法第一次放入数据代码提取

int newCap = ***DEFAULT\_INITIAL\_CAPACITY***; 初始容量16  
int newThr = (**int**)(***DEFAULT\_LOAD\_FACTOR*** \* ***DEFAULT\_INITIAL\_CAPACITY***); 0.75\*16=12 扩容数

**threshold** = newThr; 12

Node<K,V>[] newTab = (Node<K,V>[])**new** Node[newCap];初始化一个长度为16的Node数组  
**table** = newTab;

**return** newTab; 返回数组，resize方法结束

将数据放入数组

**if** ((p = tab[i = (n - 1) & hash]) == **null**)  
 tab[i] = newNode(hash, key, value, **null**);

++**modCount**;   
**if** (++**size** > **threshold**) 大于12就扩容  
 resize();

afterNodeInsertion(evict); true 一个空方法

put方法总结

判断table是否为空

table 为空

执行resize()方法,给table[]一个初始长度16,负载因子0.75,扩容长度为12

判断tab[i]的是否为空

通过hash算法，算出i,判断该位置是否空数据

tab[i] 等于空

tab[i] = newNode(hash, key, value, **null**);

tab[i] 不等于空

1.判断key是否重复

* 1. key重复

直接将value替换掉之前的

* 1. key不重复并且key属于TreeNode

* 1. key不重复

**for** (**int** binCount = 0; ; ++binCount) {

**if** ((e = p.**next**) == **null**) {  
 p.**next** = newNode(hash, key, value, **null**);  
 **if** (binCount >= ***TREEIFY\_THRESHOLD*** - 1) *// -1 for 1st* treeifyBin(tab, hash);  
 **break**;  
}

* + 1. 向下一个节点插入节点
    2. 当下一个节点的长度大于8的时候执行treeifyBin();
       1. 判断是否扩容 当tab长度<64,并且tab[i]的链表长度大于8的时候会进行扩容;
          1. tab==null 或者 tab长度小于64 进行扩容
       2. 不扩容
          1. Node数据结构转为TreeNode

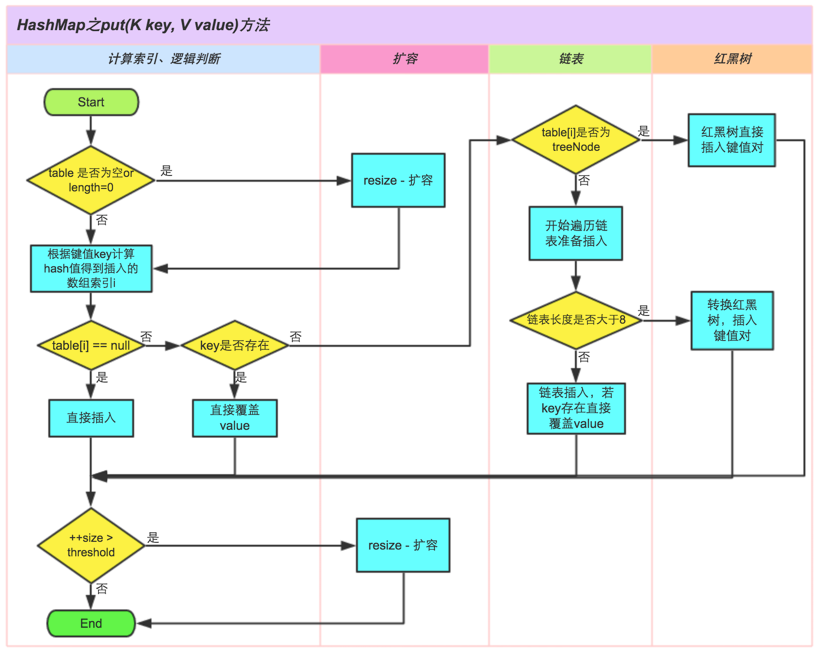
++modCount;

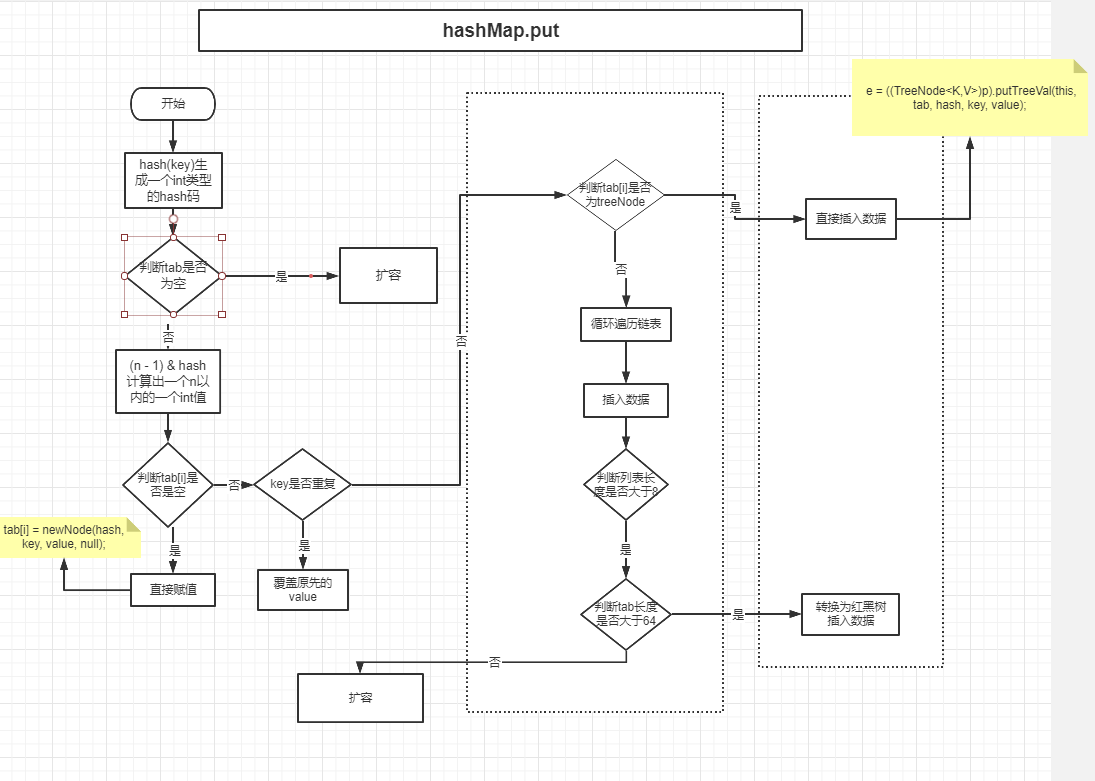
++size

判断是否需要扩容

* 1. 是，扩容 resize();

return null;

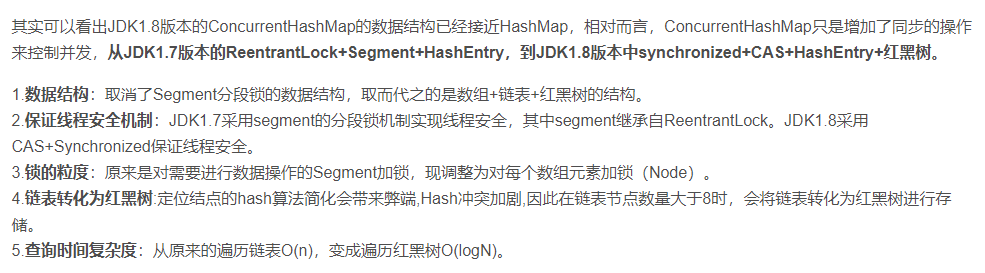




扩容

* 1. 数组长度扩大两倍
  2. Node.next==null重新计算node在数组中的位置

#### ConcurrentHashMap



## 多线程

### 线程安全问题产生的原因

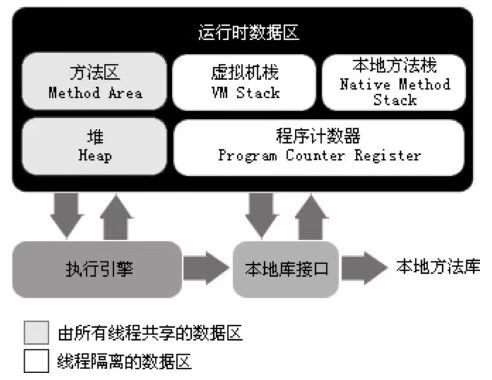
java的主内存与线程私有内存是线程安全问题产生的根本原因

## 设计模式

# Java虚拟机

## java内存模型

### 运行时内存



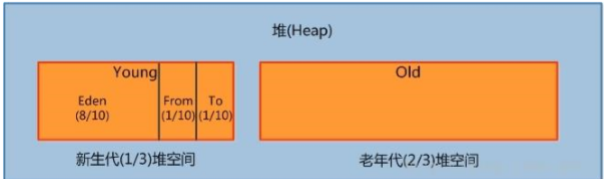
#### 程序计数器

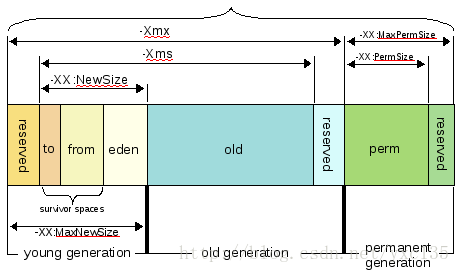
是当前线程所执行的字节码的行号指示器。

#### 堆

是java虚拟机管理内存最大的一块，存放的是对象实例。几乎所有的对象实例都在这里分配内存。

堆从垃圾回收的角度可以分为新生代，老年代。



#### java虚拟机栈

描述的是java方法执行的内存模型。每个方法在执行的时候会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出栈等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈出栈的过程。



##### 栈帧

每个方法执行，都会创建一个栈帧，伴随着方法从创建到执行完成。用于存储局部变量表，操作数栈，动态链接，方法出口等。

###### 局部变量表

主要用于存储方法参数和定义在方法体内的局部变量这些数据类型包括各类基本数据类型、对象引用（reference），以及returnAddressleixing。大小在编译时确定，运行期间不会更改局部变量表大小。

局部变量表中的变量只在当前方法调用中有效

#### 本地方法栈

本地方法栈与java虚拟机栈非常相似，java虚拟机栈为本地方法提供服务，本地方法栈为native方法提供服务。

#### 方法区

它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

永久代是jdk1.7之前方法区的具体实现，元空间是jdk1.8之后方法区的具体实现



###### 永久代

使用的是jvm内存空间。

"java.lang.OutOfMemoryError: PermGen space

###### 元空间

使用的是本地内存。

-XX:MetaspaceSize，初始空间大小

-XX:MaxMetaspaceSize，最大空间，默认是没有限制的。

元空间的特点

1. 充分利用了Java语言规范中的好处：类及相关的元数据的生命周期与类加载器的一致。
2. 每个加载器有专门的存储空间
3. 只进行线性分配
4. 不会单独回收某个类
5. 省掉了GC扫描及压缩的时间
6. 元空间里的对象的位置是固定的
7. 如果GC发现某个类加载器不再存活了，会把相关的空间整个回收掉

Java代码测试元空间存储

直接上代码

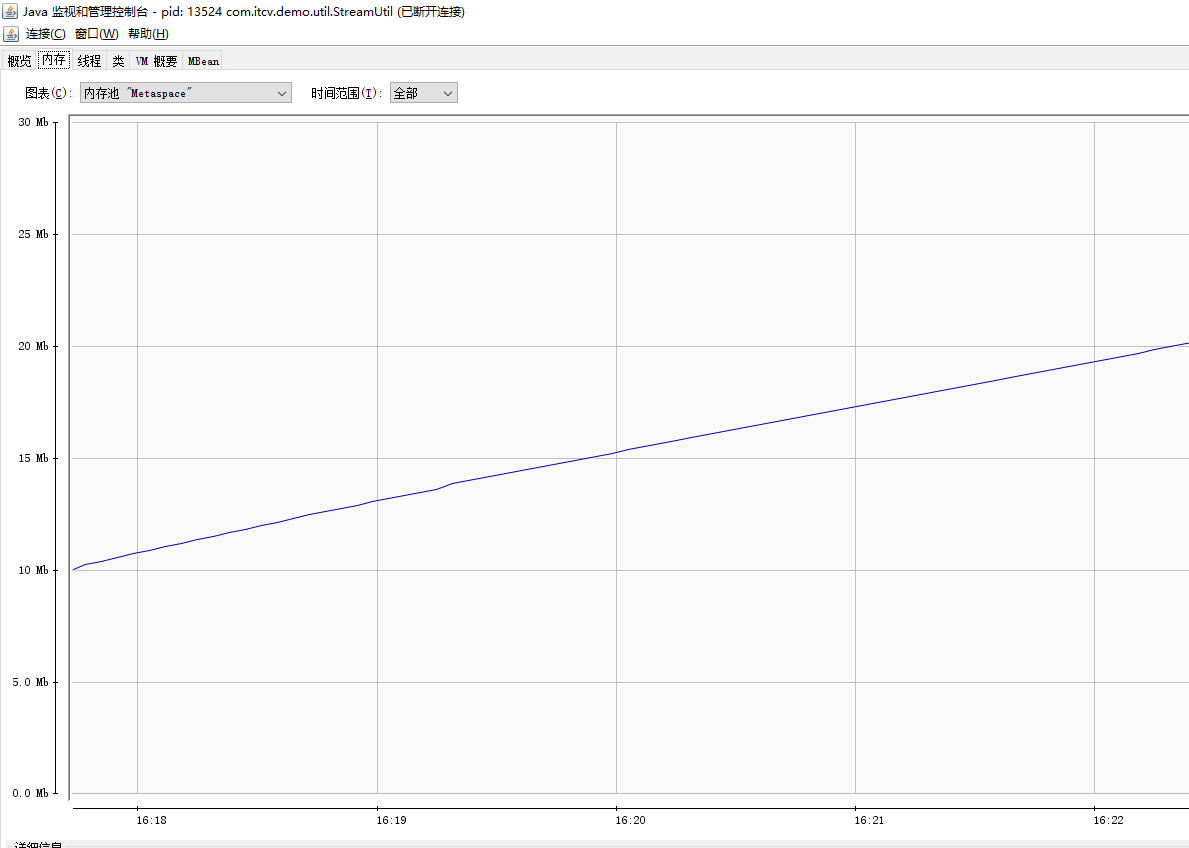
**public class** StreamUtil {  
 **public static void** main(String[] args) {  
 URL url = **null**;  
 List<ClassLoader> classLoaderList = **new** ArrayList<ClassLoader>();  
 **try** {  
 *//获取有关类型加载的JMX接口* ClassLoadingMXBean loadingBean = ManagementFactory.*getClassLoadingMXBean*();  
 url = **new** File(**"E:/data/logs"**).toURI().toURL();  
 URL[] urls = {url};  
 **while** (**true**){  
 ClassLoader loader = **new** URLClassLoader(urls);  
 Class<?> clazz = loader.loadClass(**"Persion"**);  
 clazz.getName();  
 classLoaderList.add(loader);  
 *//显示数量信息（共加载过的类型数目，当前还有效的类型数目，已经被卸载的类型数目）* System.***out***.println(**"total: "**+ loadingBean.getTotalLoadedClassCount());  
 System.***out***.println(**"active: "**+ loadingBean.getLoadedClassCount());  
 System.***out***.println(**"unloaded: "**+ loadingBean.getUnloadedClassCount());  
 Thread.*sleep*(100);  
 *// System.out.println(classLoaderList.size());* }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

实现原理

从外部不停的加载class文件到内存中，查看元空间什么时候奔溃。

元空间最小值为20M,最大值为本机内存。

查看结果



通过jconsole查看内存分配情况

###### 运行时常量池

运行时常量池是方法区的一部分。用于存放编译期生成的各种字面量和符号引用。相对与Class文件常量池的另外一个特征是具备动态性，指：运行期间也可以将新的常量放入池中。

#### 直接内存

避免了在Java堆和Native堆中来回复制数据。

### Java内存分配机制

参考文章

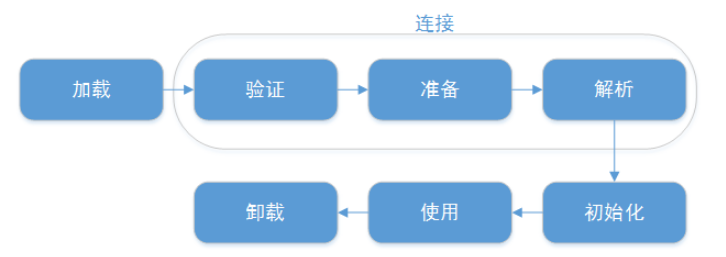
https://blog.csdn.net/freekiteyu/article/details/77992277

## Jvm类加载机制

参考文章

https://www.cnblogs.com/Cubemen/p/10913633.html

### 类加载的过程



类从加载虚拟机内存中开始到卸载出内存为止，生命周期包括：加载、验证、准备、解析、初始化、使用、卸载。

#### 1.加载

把class文件加载到内存中（**方法区**），不一定是文件可以是二进制流

#### 2.验证

##### **文件格式验证**

即验证类文件结构

##### 元数据验证

这个是否有父类，父类是否继承了不允许被继承的类等等

##### 字节码验证

对类的方法体进行校验，JDK1.6后只需检查StackMapTable属性中的记录是否合法，JDK1.7后对于主版本号大于50的Class文件，使用类型检查来完成数据流分析

##### 符号引用验证

全限定名是否能找到对应的类，在指定类中是否存在符合方法的字段描述以及简单名称描述的方法，字段。访问性是否正确。验证不成功会抛出java.lang.incompatibleClassChangeError异常的子类。

#### 3．准备

正式为**类变量**分配内存并设置类变量**初始值**的阶段；

。这个时候进行内存分配的只包括类变量（被static修饰的变量），并不包括实例变量，实例变量是在对象实例化时随对象一起分配在java堆中。

Private static int a =123;初始值为0

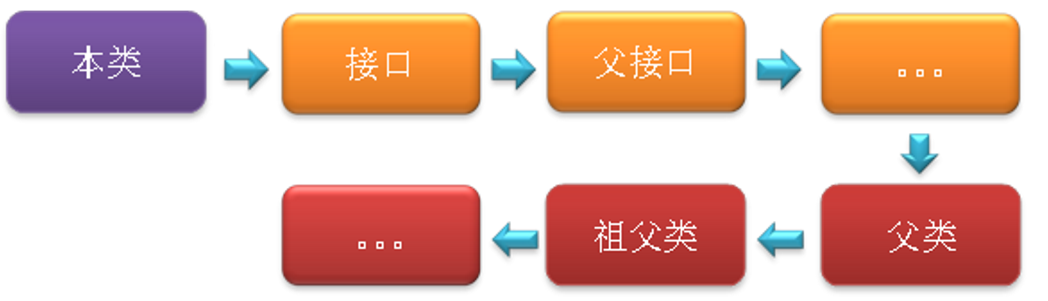
Private finale static int a =123; 初始值为123；

#### 4.解析

将符号引用转换为直接引用的过程。

解析动作主要针对：

1. 类或接口、字段（类成员变量）、类方法、接口方法等引用进行。
2. 类或接口的解析：判断所要转化成的直接引用是对数组类型，还是对普通的对象类型的引用，从而进行不同的解析。
3. 字段解析：对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束，查找流程如下图所示：



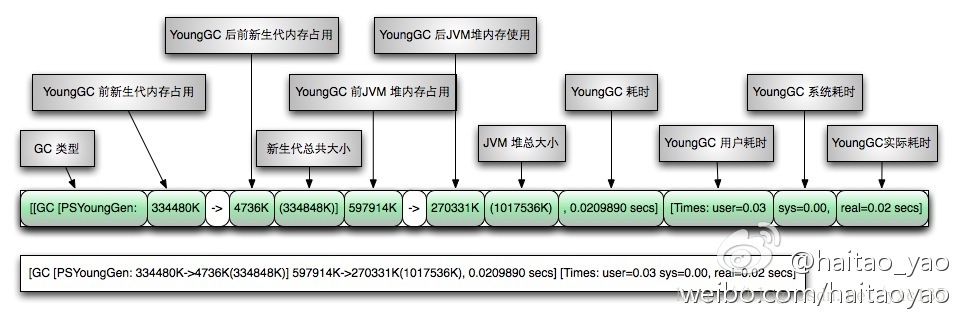
1. 类方法解析：对类方法的解析与对字段解析的搜索步骤差不多，只是多了判断该方法所处的是类还是接口的步骤，而且对类方法的匹配搜索，是先搜索父类，再搜索接口。
2. 接口方法解析：与类方法解析步骤类似，由于接口不会有父类，因此，只递归向上搜索父接口就行了。

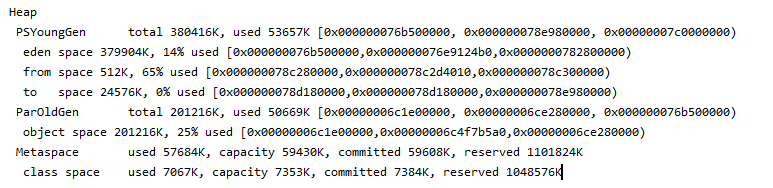
#### 5.初始化

是类加载过程的最后一步，到了此阶段，才真正开始执行类中定义的Java程序代码。

### Java中的类加载器与双亲委派机制

## JvmGc日志





# Spring

# SpringMVC

# SpringBoot

# Dubbo

## DubboSpi

### javaSpi

#### javaSpi的机制

是一种将服务接口与服务实现分离以达到解耦、大大提升了程序可扩展性的机制。

#### 实现javaSpi

1. 定义一个接口
2. 在META-INF/services目录下，创建该接口的同名文件
3. 该文件的内容就是接口的具体实现类的全类名（可以是多个）
4. 通过ServiceLoader.load()加载实现类

#### JavaSpi的缺点

1. 不能按需加载,全部加载,浪费资源；
2. 获取某个实现类的方式不够灵活，只能通过遍历；
3. 多个并发多线程使用 ServiceLoader 类的实例是不安全的；
4. 加载不到实现类时抛出并不是真正原因的异常，错误很难定位；

### DubboSpi的增强

1. 对 Dubbo 进行扩展，不需要改动 Dubbo 的源码；
2. 延迟加载，可以一次只加载自己想要加载的扩展实现。
3. 增加了对扩展点 IOC 和 AOP 的支持，一个扩展点可以直接 setter 注入其它扩展点
4. Dubbo 的扩展机制能很好的支持第三方 IoC 容器，默认支持 Spring Bean。

### 实现dubboSpi

1. Spi文件存储路径在META-INF\dubbo\internal 目录下 文件名为接口的全路径名 接口的包名+接口名；
2. 每个spi文件里面的格式定义为： 扩展名=具体的类名，例如 dubbo=com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtoco

## Dubbo 原理



1. 服务端启动时会把所有接口注册到注册中心
2. 客户端启动时，订阅所需要的服务
3. 订阅内容变更时，会推送订阅的消息
4. 客户端启动时，会与服务端建立长连接，然后进行数据通信
5. 服务端，客户端启动后，后台会启动定时器，发送统计数据给monitor
6. 服务启动的时候，provider和consumer根据配置信息，连接到注册中心register，分别向注册中心注册和订阅服务
7. register根据服务订阅关系，返回provider信息到consumer，同时consumer会把provider信息缓存到本地。如果信息有变更，consumer会收到来自register的推送
8. consumer生成代理对象，同时根据负载均衡策略，选择一台provider，同时定时向monitor记录接口的调用次数和时间信息
9. 拿到代理对象之后，consumer通过代理对象发起接口调用
10. provider收到请求后对数据进行反序列化，然后通过代理调用具体的接口实现

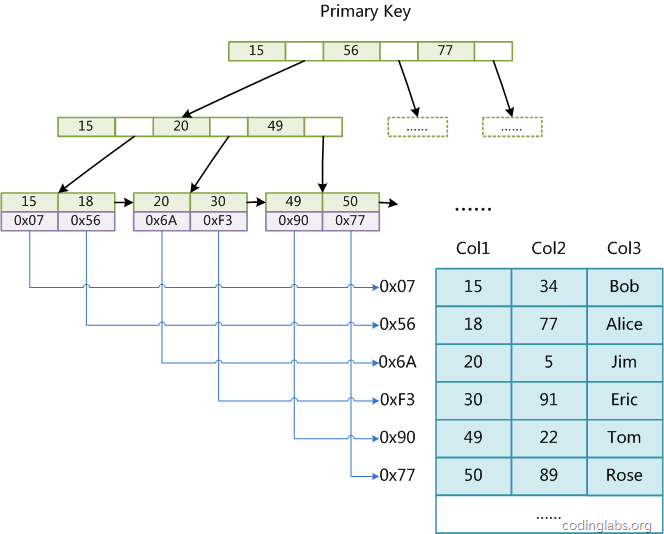
# Mysql

## Mysql索引实现

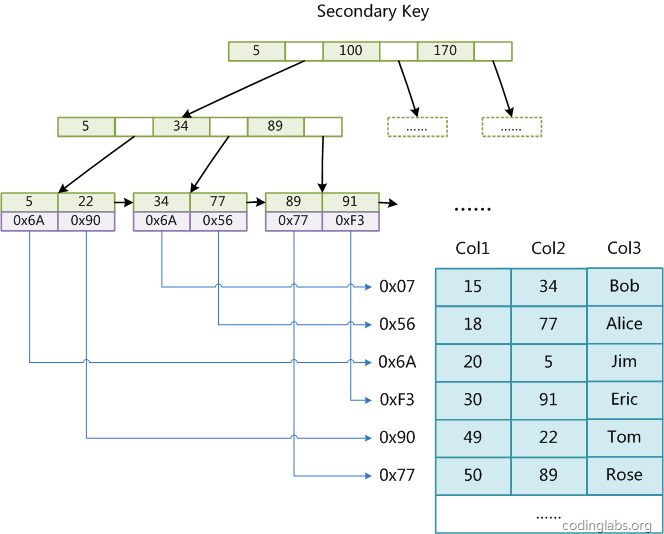
在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，本文主要讨论MyISAM和InnoDB两个存储引擎的索引实现方式。

### MyISAM索引实现

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM索引的原理图：



这里设表一共有三列，假设我们以Col1为主键，则图8是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，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。

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分

### InnoDB索引实现

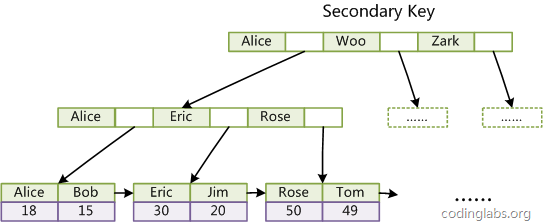
虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，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。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



图10是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，图11为定义在Col3上的一个辅助索引：



这里以英文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

## 索引使用策略及优化

MySQL的优化主要分为结构优化（Scheme optimization）和查询优化（Query optimization）。本章讨论的高性能索引策略主要属于结构优化范畴。本章的内容完全基于上文的理论基础，实际上一旦理解了索引背后的机制，那么选择高性能的策略就变成了纯粹的推理，并且可以理解这些策略背后的逻辑。

### 最左前缀原理与相关优化

# 面试

## Mysql

### 数据库三范式

**1NF:字段不可分;**  
**2NF:有主键，非主键字段依赖主键;**  
**3NF:非主键字段不能相互依赖; （**每列都与主键有直接关系，不存在传递依赖; **）**

### 分别说一下范式和反范式的优缺点

#### 优点

1.范式化的更新操作通常比反范式化要快。

2.当数据较好地范式化时，就只有很少或者没有重复数据，所以只需要修改更少的数据。

3.范式化的表通常更小，可以更好的放在内存里，所以执行操作会更快。

4.很少有多余的数据意味着检索列表数据时更少需要DINSTINCT或者GROUP BY语句

#### 缺点

表关联

### Mysql 数据库索引

#### B+ 树和 B 树的区别

1. 非叶子节点只存储键值信息。
2. 所有叶子节点之间都有一个链指针。
3. 数据记录都存放在叶子节点中。

#### 为什么 B+ 树比 B 树更适合应用于数据库索引，除了数据库索引，还有什么地方用到了（操作系统的文件索引）

1、 B+树的磁盘读写代价更低：B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。

2、B+树的查询效率更加稳定：由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

3、由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

1、由于非叶子节点只存储了键值信息，B+树会减少读取磁盘的次数

2、方便扫库

#### 聚簇索引和非聚簇索引

聚簇索引并不是一种单独的索引类型，而是一种数据存储方式。MySQL数据库中innodb存储引擎，B+树索引可以分为聚簇索引（也称聚集索引，clustered index）和辅助索引（有时也称非聚簇索引或二级索引，secondary index，non-clustered index）。这两种索引内部都是B+树，聚集索引的叶子节点存放着一整行的数据。Innobd中的主键索引是一种聚簇索引，非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引。

Innodb使用的是聚簇索引，MyISam使用的是非聚簇索引。

##### 聚簇索引(聚集索引)

聚簇索引就是按照每张表的主键构造一颗B+树，同时叶子节点中存放的就是整张表的行记录数据，也将聚集索引的叶子节点称为数据页。这个特性决定了索引组织表中数据也是索引的一部分，每张表只能拥有一个聚簇索引。

Innodb通过主键聚集数据，如果没有定义主键，innodb会选择非空的唯一索引代替。如果没有这样的索引，innodb会隐式的定义一个主键来作为聚簇索引。

###### 聚簇索引的优缺点

优点

1.数据访问更快，因为聚簇索引将索引和数据保存在同一个B+树中，因此从聚簇索引中获取数据比非聚簇索引更快

2.聚簇索引对于主键的排序查找和范围查找速度非常快

缺点

1.插入速度严重依赖于插入顺序，按照主键的顺序插入是最快的方式，否则将会出现页分裂，严重影响性能。因此，对于InnoDB表，我们一般都会定义一个自增的ID列为主键

2.更新主键的代价很高，因为将会导致被更新的行移动。因此，对于InnoDB表，我们一般定义主键为不可更新。

3.二级索引访问需要两次索引查找，第一次找到主键值，第二次根据主键值找到行数据。

#### 前缀索引和覆盖索引

##### 前缀索引

前缀索引，一种优化索引大小的解决方案；前缀索引说白了就是对文本的前几个字符（具体是几个字符在建立索引时指定）建立索引，这样建立起来的索引更小，所以查询更快。

建立前缀索引的语法为：

ALTER TABLE table\_name ADD KEY(column\_name(prefix\_length));

下面总结一下什么情况下使用前缀索引：

字符串列(varchar,char,text等)，需要进行全字段匹配或者前匹配。也就是=‘xxx’ 或者 like ‘xxx%'

字符串本身可能比较长，而且前几个字符就开始不相同。比如我们对中国人的姓名使用前缀索引就没啥意义，因为中国人名字都很短，另外对收件地址使用前缀索引也不是很实用，因为一方面收件地址一般都是以XX省开头，也就是说前几个字符都是差不多的，而且收件地址进行检索一般都是like ’%xxx%’，不会用到前匹配。相反对外国人的姓名可以使用前缀索引，因为其字符较长，而且前几个字符的选择性比较高。同样电子邮件也是一个可以使用前缀索引的字段。

前一半字符的索引选择性就已经接近于全字段的索引选择性。如果整个字段的长度为20，索引选择性为0.9，而我们对前10个字符建立前缀索引其选择性也只有0.5，那么我们需要继续加大前缀字符的长度，但是这个时候前缀索引的优势已经不明显，没有太大的建前缀索引的必要了。

##### 覆盖索引

就是select的数据列只用从索引中就能够取得，不必从数据表中读取，换句话说查询列要被所使用的索引覆盖

#### 介绍一下数据库的事务

##### 事务简介

1. 在 MySQL 中只有使用了 Innodb 数据库引擎的数据库或表才支持事务。
2. 事务处理可以用来维护数据库的完整性，保证成批的 SQL 语句要么全部执行，要么全部不执行。
3. 事务用来管理 insert,update,delete 语句。

##### 事务的四大特性

###### 原子性

一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，

会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

###### 一致性

在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，

这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。(比如：A向B转账，不可能A扣了钱，B却没有收到)

###### 隔离性

数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。

事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。

（比如：A正在从一张银行卡里面取钱，在A取钱的过程中，B不能向这张银行卡打钱）

###### 持久性

事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

#### Mysql 有哪些隔离级别

https://www.cnblogs.com/jian-gao/p/10795407.html

##### ****Read Uncommitted（读取未提交内容）****

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

##### Read Committed（读取提交内容）

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。

##### Repeatable Read（可重复读）

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。

指的是：事务A在事务未提交之前，获得的结果都是一样的。

A：启动事务，此时数据为初始状态

B：启动事务，更新数据，但不提交

A：再次读取数据，发现数据未被修改

B：提交事务

A：再次读取数据，发现数据依然未发生变化，这说明这次可以重复读了

B：插入一条新的数据，并提交

A：再次读取数据，发现数据依然未发生变化，虽然可以重复读了，但是却发现读的不是最新数据，这就是所谓的“幻读”

A：提交本次事务，再次读取数据，发现读取正常了。

select @@tx\_isolation;

show variables like 'autocommit';

set autocommit = off;

select \* from test1;

set session TRANSACTION ISOLATION level repeatable read;

start transaction;

select \* from test1;

COMMIT;

select @@tx\_isolation;

show variables like 'autocommit';

set autocommit = off;

start transaction;

update test1 set num =1 where id =1;

COMMIT;

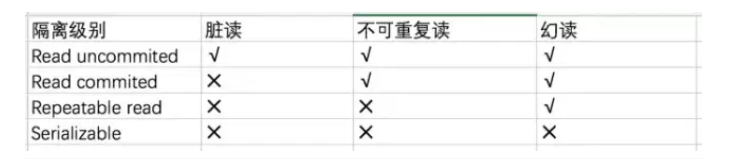
insert into test1 value (9,9,'2020-09-16 22:32:31')

COMMIT;

由以上的实验可以得出结论，可重复读隔离级别只允许读取已提交记录，而且在一个事务两次读取一个记录期间，其他事务部的更新该记录。但该事务不要求与其他事务可串行化。例如，当一个事务可以找到由一个已提交事务更新的记录，但是可能产生幻读问题(注意是可能，因为数据库对隔离级别的实现有所差别)。像以上的实验，就没有出现数据幻读的问题

##### Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。



select version() from dual;

select @@tx\_isolation; -- 查询当前会话隔离级别

set session TRANSACTION ISOLATION level Read Uncommitted; -- 设置当前会话隔离级别

set global transaction isolation level Read Committed; -- 设置系统当前隔离级别

-- 开启一个事务

start TRANSACTION;

select \* from test1 where id =1;

update test1 set num =2 where id=1;

COMMIT;

ROLLBACK;

#### Mysql 在可重复度的隔离级别下会不会有幻读的情况，为什么？

会，因为刚插入的数据，不会被当前事务读取。

#### Mysql 事务是如何实现的

原子性：通过undo log实现，要么提交，要么回滚。

#### Binlog 和 Redo log 的区别是什么，分别是什么用？

##### Binlog

binlog只会在日志提交后，一次性记录执行过的事务中的sql语句以及其反向sql(作为回滚用)，保存的是逻辑日志->**执行的sql语句**

###### binlog作用

binlog可以作为恢复数据使用，主从复制搭建

##### redo log

redo log叫做重做日志，是用来实现事务的持久性。该日志文件由两部分组成：重做日志缓冲（redo log buffer）以及重做日志文件（redo log）,前者是在内存中，后者在磁盘中。当事务提交之后会把所有修改信息都会存到该日志中。

###### redo log 有什么作用？

mysql 为了提升性能不会把每次的修改都实时同步到磁盘，而是会先存到Boffer Pool(缓冲池)里头，把这个当作缓存来用。然后使用后台线程去做缓冲池和磁盘之间的同步。

那么问题来了，如果还没来的同步的时候宕机或断电了怎么办？还没来得及执行上面图中红色的操作。这样会导致丢部分已提交事务的修改信息！

所以引入了redo log来记录已成功提交事务的修改信息，并且会把redo log持久化到磁盘，系统重启之后在读取redo log恢复最新数据。

##### undo log

#### 谈一谈 MVCC 多版本并发控制

#### Innodb 和 MyISAM 的区别是什么

#### Innodb 的默认加锁方式是什么，是怎么实现的

#### 如何高效处理大库 DDL

#### Mysql 索引重建

#### 对于多列索引，哪些情况下能用到索引，哪些情况用不到索引

#### 为什么使用数据库索引可以提高效率，在什么情况下会用不到数据库索引？

共享锁和排他锁的使用场景，

#### 关系型数据库和非关系数据库的优缺点

#### Mysql 什么情况会造成慢查，如何查看慢查询

#### 如何处理慢查询，你一般是怎么处理慢查询的

#### Mysql 中 varchar 和 char 的区别

数据库外键的优缺点

有没有使用过数据库的视图

Mysql 中插入数据使用自增 id 好还是使用 uuid，为什么？

Mysql 有哪些数据类型，使用的时候有没有什么注意点

Mysql 集群有哪几种方式，分别适用于什么场景

Mysql 主从模式如何保证主从强一致性

Mysql 集群如何保证主从可用性

Mysql 读写分离有哪些解决办法

非关系型数据库

redis 的底层数据结构有哪些

redis 中的 SDS 和 C 语言中的字符串有什么区别，优点是什么

redis 中的字典是如何实现的，如何解决冲突和扩容

redis 的跳表的使用场景是什么，可以实现一下吗

redis 缓存穿透，缓存击穿，缓存雪崩，热点数据集中失效 （常问）

redis 的淘汰策略，来写一下 LRU 吧

redis 的持久化方式，RDB 和 AOF 分别的使用场景

redis 如何处理事务

redis 为什么那么快？

redis 是单线程为什么还那么快？

redis 的操作为什么是原子性的，如何保证原子性

redis 集群用过哪些方案，分别怎么做。讲一下一致性哈希

redis 什么情况下会出现性能问题，有什么处理办法？

有没有使用过 redis 的分布式锁，有什么优缺点

说一下 redis 的内存模型

说一下 redis 和 memcache 的区别

你用 redis 做过什么？（这里尽量不要讲只做过缓存，可以说一下队列，排行榜/计数器，发布/订阅）

你用过哪些非关系型数据库，都有什么特点，使用场景分别是什么（体现你技术广度的时刻到了，尽可能多说，但是不会的不要说，防止被问死）

Mongodb 相对于 Mysql 有哪些优势，底层索引使用的数据结构是什么，为什么要使用这个

Mongodb 中的分片是什么意思

概述

什么是 Redis？

Redis 有哪些优缺点？

为什么要用 Redis / 为什么要用缓存？

为什么要用 Redis 而不用 map/guava 做缓存?

Redis 为什么这么快？

数据类型

Redis 有哪些数据类型？

Redis 的应用场景？

持久化

什么是 Redis 持久化？

Redis 的持久化机制是什么？各自的优缺点？

如何选择合适的持久化方式

Redis 持久化数据和缓存怎么做扩容？

过期键的删除策略

Redis 的过期键的删除策略

Redis key 的过期时间和永久有效分别怎么设置？

我们知道通过 expire 来设置 key 的过期时间，那么对过期的数据怎么处理呢?

内存相关

MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据

Redis 的内存淘汰策略有哪些

Redis 主要消耗什么物理资源？

Redis 的内存用完了会发生什么？

Redis 如何做内存优化？

线程模型

Redis 线程模型

事务

什么是事务？

Redis 事务的概念

Redis 事务的三个阶段

Redis 事务相关命令

事务管理（ACID）概述

Redis 事务支持隔离性吗

Redis 事务保证原子性吗，支持回滚吗

Redis 事务其他实现

集群方案

哨兵模式

官方 Redis Cluster 方案(服务端路由查询)

基于客户端分配

基于代理服务器分片

Redis 主从架构

Redis 集群的主从复制模型是怎样的？

生产环境中的 redis 是怎么部署的？

说说 Redis 哈希槽的概念？

Redis 集群会有写操作丢失吗？为什么？

Redis 集群之间是如何复制的？

Redis 集群最大节点个数是多少？

Redis 集群如何选择数据库？

分区

Redis 是单线程的，如何提高多核CPU的利用率？

为什么要做 Redis 分区？

你知道有哪些 Redis 分区实现方案？

Redis 分区有什么缺点？

分布式问题

Redis 实现分布式锁

如何解决 Redis 的并发竞争 Key 问题

分布式 Redis 是前期做还是后期规模上来了再做好？为什么？

什么是 RedLock

缓存异常

缓存雪崩

缓存穿透

缓存击穿

缓存预热

缓存降级

热点数据和冷数据

缓存热点 key

常用工具

Redis 支持的 Java 客户端都有哪些？官方推荐用哪个？

Redis 和 Redisson 有什么关系？

Jedis 与 Redisson 对比有什么优缺点？

其他问题

Redis 与 Memcached 的区别

如何保证缓存与数据库双写时的数据一致性？

Redis 常见性能问题和解决方案？

Redis 官方为什么不提供 Windows 版本？

一个字符串类型的值能存储最大容量是多少？

Redis 如何做大量数据插入？

假如 Redis 里面有 1 亿个 key，其中有 10w 个 key 是以某个固定的已知的前缀开头的，如果将它们全部找出来？

使用 Redis 做过异步队列吗，是如何实现的

Redis 如何实现延时队列

Redis 回收进程如何工作的？

Redis 回收使用的是什么算法？