第三季 JAVA基础

这一季主要集中比较基础的知识：java并发、mysql、网络、JDK集合JVM、spring源码、tomcat、linux、系统设计、生产实践。是一些比较常规的问题。

-------------------- 中华石杉：第三季大纲 ------------------
Java集合包
06、为什么在Java面试中一定会深入考察Hashmap？
07、你知道Hashmap底层的数据结构是什么吗？
08、你知道Hashmap是如何解决hash碰撞问题的吗？
09、说说Hashmap是如何进行扩容的可以吗？
10、说说Treemap和LinkedHashMap的实现原理？

Java并发编程
11、BAT面试官为什么都喜欢问并发编程的问题？
12、ConcurrentHashMap实现线程安全的底层原理到底是什么？
13、说说synchronized关键字的底层原理以及跟Lock锁之间的区别？
14、你对JDK中的AQS理解吗？AQS的实现原理是什么？
15、说说线程池的底层工作原理可以吗？
16、那你再说说线程池的核心配置参数都是干什么的？平时我们应该怎么用？
17、你知道如果线程池的队列满了之后，会发生什么事情吗？
18、如果线上机器突然宕机，线程池的阻塞队列中的请求怎么办？
19、如果在线程中使用无界阻塞队列会发生什么问题？
20、能聊聊你对CAS的理解以及其底层实现原理可以吗？
21、谈谈你对Java内存模型的理解可以吗？
22、你知道Java内存模型中的原子性、有序性、可见性是什么吗？
23、你知道指令重排、内存栅栏以及happens-before这些是什么吗？
24、能从Java底层角度聊聊volatile关键字的原理吗？
25、能说说Threadlocal的底层实现原理吗？

Spring
26、为什么Spring框架是互联网公司面试必问的环节？
27、Spring的AOP和IOC机制都是如何实现的？循环依赖应该如何处理呢？
28、了解过cgib动态代理吗？他跟jdk动态代理的区别是什么？
29、Spring的事务实现原理是什么？能聊聊你对事务传播机制的理解吗？
30、从源码实现角度谈谈，Spring中用了哪些设计模式？

Tomcat
31、你知道为什么面试官要考察Tomcat底层知识吗？
32、你能聊聊Tomcat的核心架构原理吗？
33、你知道Tomcat的线程模型是什么样的吗？Tomcat有多少工作线程？
34、平时你们生产环境中是如何配置 Tomcat的JVM的？如何对 Tomcat进行性能优化？
35、说说你对负载均衡算法的理解，以及Nginx的负载均衡原理？

JVM
36、为什么互联网公司的面试官会极为重视JVM的考察？
37、JVM中有哪几块内存区域？Java8之后对内存分代做了什么改进？
38、你知道JMM是如何运行起来的吗？如何创建对象以及何时触发垃圾回收？
39、说说你对JVM的垃圾回收算法以及垃圾回收器的理解？
40、你们生产环境中是如何设置JM的内存参数以及垃圾回收参数的？
41、JyM可能会发生哪几种OOM？如何排查和处理线上系统的OOM问题？
42、你在实际项目中是否做过JVM GC优化，怎么做的？
43、聊聊JVM类加载器体系？为什么要使用双亲委派？如何自定义类加载？

网络
44、为什么BAT工程师的面试中要考察网络的基础知识？
45、说说你对TCP/IP四层模型的理解？
46、说说HTP协议的工作原理，还有HTTP1.0、1.1以及2.0的区别是什么？

47、你现场画一下HTPS协议的原理，如何使用HTPS协议？
48、你知道什么是网络抓包的问题吗？能说说怎么解决这个问题吗？
49、谈谈你对TCP三次握手和四次握手的理解，以及为什么要这么做？
50、当你用浏览器打开一个链接的时候，计算机做了哪些工作步骤。

Linux
51、为什么Java工程师的面试中要考察Linux的基础知识？
52、在Linux中你都关注过哪些关键的内核参数，有没有做过优化？
53、Linux有哪几种IO模型？你知道epoll和poll有什么区别吗？
54、说说你对Linux操作系统中线程切换过程的理解？

MySQL
55、如果Java工程师只会写SQL，能hold住互联网公司的线上系统吗？
56、MySQL的myisam和innodb两种存储引擎的区别是什么？底层文件结构是什么？
57、对MySQL的索引原理了解吗？索引的数据结构是什么？B＋树和B树有什么区别？
58、你知道MySQL中支持哪几种锁吗？说说你对MySQL行锁实现原理的理解？
59、谈谈你对 MYSQL中的事务原理的理解？有哪几种事务隔离级别？
60、谈谈 MYSQL的常见性能优化方法，以及SQL调优的方法？

生产实践
61、系统启动10分钟后机器就会CPU负载100%，重启之后依然如此，怎么排查？
62、如果线上系统的内存使用率一直不停上涨，重启之后依然如此，怎么排查？
63、如果线上系统突然假死，无法访问了，此时如何排查？
64、如果线上系统出现线程死锁或者 MYSQL死锁问题，应该如何排查？
65、平时你们对线上系统是如何进行监控的？cpu、内存、磁盘、io和jvm都监控什么？

场景设计
66、为什么BAT大厂要考察经典场景的系统设计问题？
67、如果让你来设计12306售票系统，应该如何设计？
68、如果让你来设计淘宝双十一场景下的峰值系统，应该如何设计？
69、如果让你来设计一个电商场景下的秒杀系统，应该如何设计？
70、如果让你来设计一个百万TPS的高并发支付系统，应该如何设计？

总结
71、第三季的结束语:技术学习永无止境，面试也远不止是这些东西心

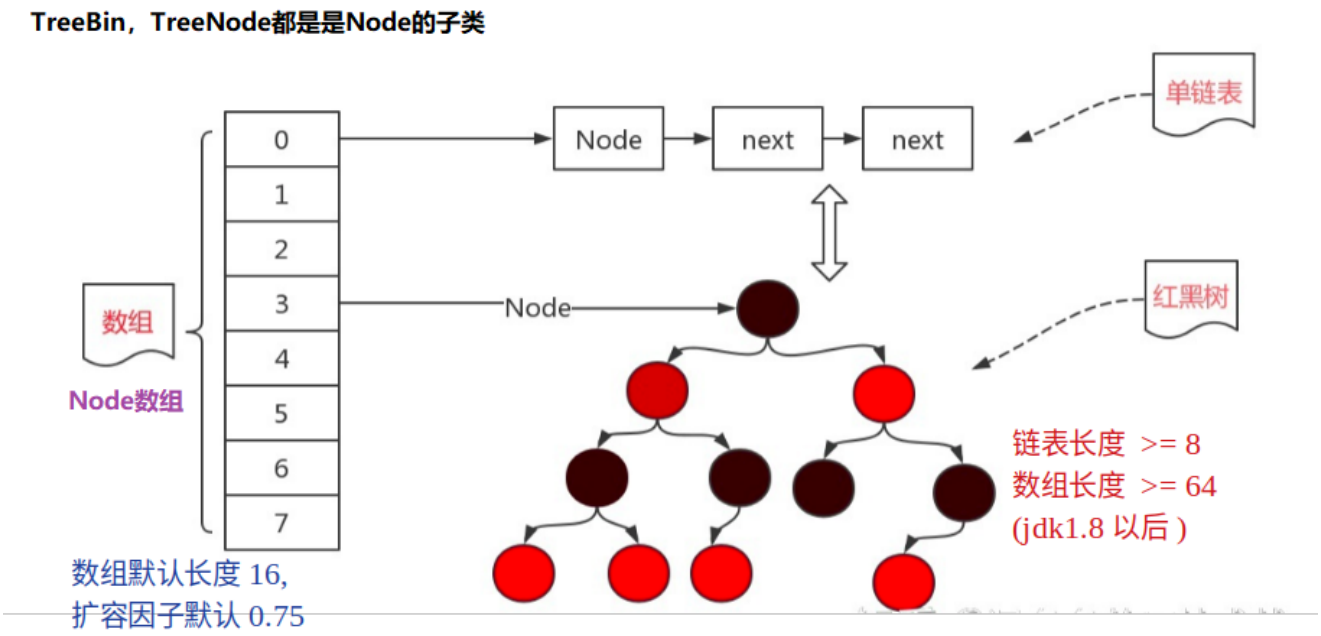
一、java集合包、并发包

1、HashMap

1.hashMap的数据结构

```
1.数组 + 单链表: 单链表的查询复杂度O(n)，需要遍历
2.数组 + 红黑树: 红黑树的查询复杂度O(logn)
```

其实hashMap就是Node[16]的数组，只不过有的Node是TreeBin实例。TreeBin里面是由TreeNode组成。



为何数组长度大于等于64以后，才能从链表转成红黑树？

因为如果数组长度太小，采用红黑树的时候，TreeBin里面元素会太多。

```
// 单向链表的 Node
Node(int hash, K key, V value, Node<K,V> next) {
    this.hash = hash;  // 经过hash算法计算得到的hash值
    this.key = key;
    this.value = value;
    this.next = next;
}

// 红黑树的TreeNode
static final class TreeNode<K, V> extends LinkedHashMap.Entry<K, V> {
    TreeNode<K, V> parent;  // red-black tree links
    TreeNode<K, V> left;
    TreeNode<K, V> right;
    TreeNode<K, V> prev;    // needed to unlink next upon deletion
    boolean red;
}
```

2. hash算法和寻址算法

都用与运算替代了取模运算！

2.1 hash算法

优化：如果直接采用key.hashCode()时，在后面寻址算法中，高16位没有任何意义。很容易产生碰撞。而且大部分hashcode的区别都是体现在高位。

将hashcode值右移16位再做异或运算后，返回的hash值同时包含了高16位和低16的信息。尽可能使得数组分散。

```
// hash算法
static final int hash(Object key) {
    int h;
    // 将hashcode值(32位)，右移16位再做异或运算
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

```
# 示例
1111 1111 1111 1111 1111 1010 0111 1100    (h)
0000 0000 0000 0000 1111 1111 1111 1111    (h>>16)
1111 1111 1111 1111 0000 0101 1000 0011    (^运算)   -> 转成int值返回
```

2.2 寻址算法

```
// n表示数组长度(默认16)
Node p = tab[i = (n - 1) & hash]

// hash算法优化的hash值，包含了高16位和低16位的信息
1111 1111 1111 1111 0000 0101 1000 0011  // hash算法返回的hash
0000 0000 0000 0000 0000 0000 0000 1111  // (n-1 = 15)

// 采用原始hash值
1111 1111 1111 1111 1111 1010 0111 1100  // key.hashcode()
0000 0000 0000 0000 0000 0000 0000 1111  // (n-1 = 15)
总结：如果不经过优化，高16位的与运算是没有任何意义。可以直接被忽略掉。
```

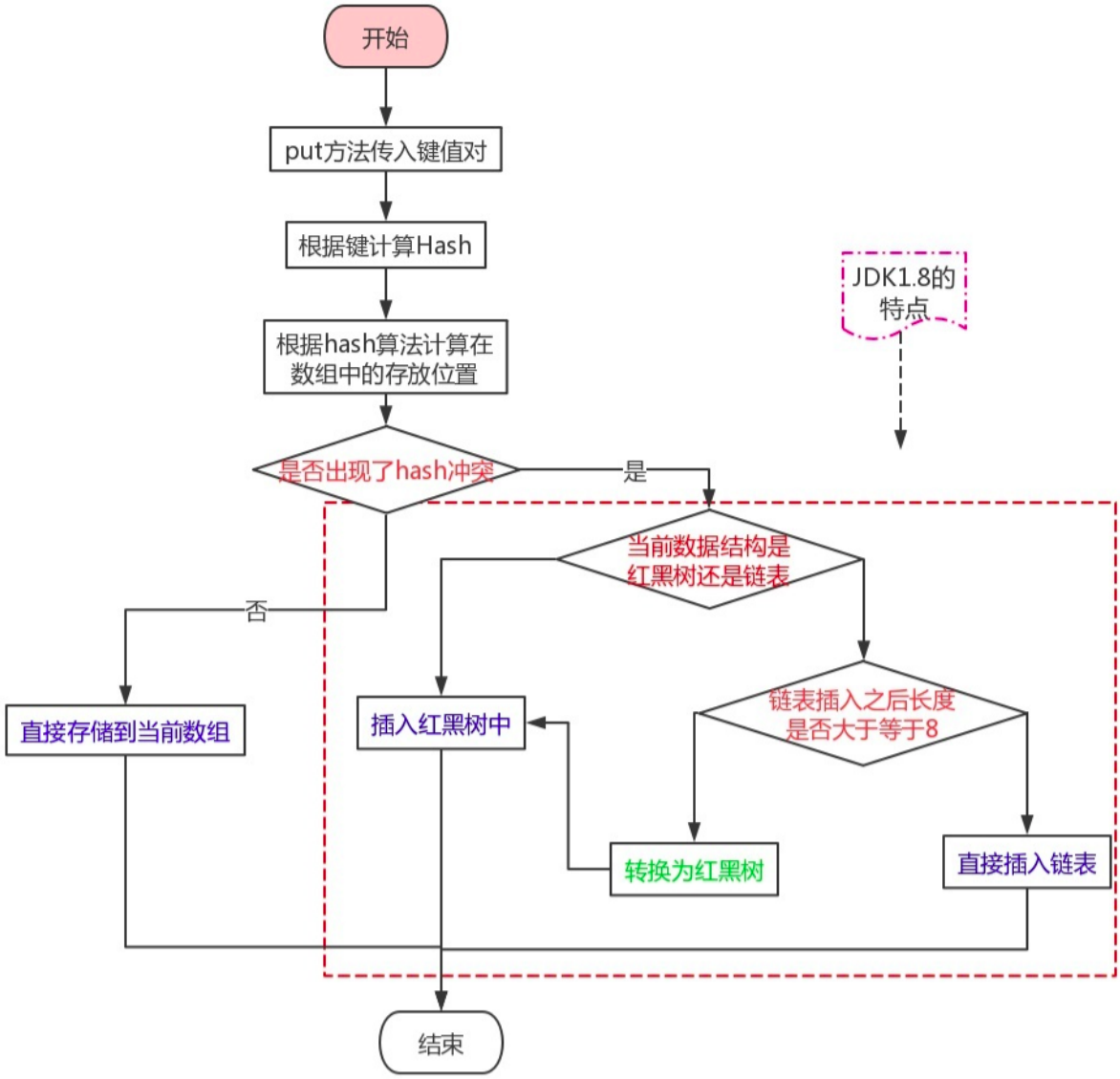
2.3 tableSizeFor

tableSizeFor方法主要是在初始化HashMap(int capacity)的重新就是真实的capacity。因为传入的capacity可能不是2的整数倍。

tableSizeFor(1000) -> 1024

```
/**
 * Returns a power of two size for the given target capacity.
 */
static final int tableSizeFor(int cap) {
    int n = -1 >>> Integer.numberOfLeadingZeros(cap - 1);
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

3. HashMap的put流程



4.常见基本问题汇总

```
1.可以用链表代替底层的数组结构吗？
hash可以。但是数组查询效率更高

2.HashMap什么时候resize()？
hash当元素超过0.75 × Capacity时。扩容为两倍。
ArrayList扩容是1.5倍。

3.为什么扩容是两倍？
保证扩容是，每个元素要么在当前位置，要么在当前位置 + OldCapacity位置。

4.当链表转为红黑树后，什么时候退化为链表
长度为6的时候退化为链表。中间有个差值7可以防止链表和树之间频繁的转换。
```

5.HashMap在并发环境下有哪些问题？

- 情况1：多线程扩容，引起链表死循环（通过ConcurrentHashMap解决了）
- 情况2：多线程put时，导致元素丢失。或者put非null元素，但get出来却是null流程。

通常解决这些问题的办法就是采用线程安全的集合类：ConcurrentHashMap，HashTable

情况1：
HashMap在put的时候，插入的元素超过了容量（initialCapacity * loadFactor）的范围就会触发扩容操作，就是rehash，
这个会重新将原数组的内容重新hash到新的扩容数组中，在多线程的环境下，存在同时其他的元素也在进行put操作，如果hash值相同，可能出现同时在同一数组下用链表表示，造成闭环，导致在get时会出现死循环，所以HashMap是线程不安全的。
线程1：rehash后，第一个链表顺序为： 0 -> 4 挂起
线程2：rehash后，第一个链表顺序为： 4 -> 0

再次执行线程1：存储下来的链表就是 0 -> 4 -> 0。即node.next指向了之前的元素，形成死循环。CPU飙升！！

情况2：
将一下代码循环执行多次，可以发现并不是每次都是返回0。
原因就是多线程put导致HashMap发生扩容，get(0L)返回了NULL

```
static void testHashMap() {
    HashMap<Long, String> ss = new HashMap<>();
    final AtomicInteger in = new AtomicInteger(0);
    ss.put(0L, "ssssssss");
    for (int i = 0; i < 200; i++) {
        new Thread(() -> {
            ss.put(System.nanoTime() + new Random().nextLong(), "sssssssssss");
            String s = ss.get(0L);
            if (s == null) {
                in.incrementAndGet();
            }
        }).start();
    }
    System.out.println(in);
}
```

6.HashMap的key一般什么格式？

可以用null吗？可以用可变类吗？

通常采用Interger，String这种不可变类型作为key。并且可以为null，null的hash计算后为0.

- String不可变，而且hashcode在创建时已经被缓存。使得String很适合作为key，处理速度很快
- get时需要用到key的equals()和hashcode()方法。正确重写这两个方法是很重要的。

不能采用可变类作为key，会导致put进去的值无法被查找到，返回null。

8.如何自定义class作为HashMap的key？

要点：

- 重写equals和hashcode方法要注意什么
- 如何设计一个不可变类

```
1.final修饰类，保证类不被继承。防止子类通过方法修改父类的成员变量。
2.private final修饰成员变量。保证成员变量不可变。(但还不够，如果成员变量是对象，还是可能在外部被改变)
3.通过构造方法初始化所有成员变量，进行深copy。
  比如this.array = array.clone();
4.不提供修改成员变量的方法。比如set方法
5.get方法不返回对象本身，而是返回对象的copy。
```

9.HashMap中的hash()和tableSizeFor()

```
static final int hash(Object key) {
    int h;
    // h >>> 16 高位移到低位。想当于将hashcode值的高位与低位直接异或(^)了一下
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

而
n = table.length;      // 默认16
index = (n-1) & hash   // 此时只有最低的4为参与了运算
```

在计算index时，只有低位才会参与计算。如果直接采用hashcode值，很容易产生碰撞。而且大部分hashcode的区别都是体现在高位。所以设计者先对hashcode的高位和低位进行异或运行，再与table.length进行&运算。

tableSizeFor方法主要是在初始化HashMap(int capacity)的重新就是真实的capacity。因为传入的capacity可能不是2的整数倍。

tableSizeFor(1000) -> 1024

```
/**
 * Returns a power of two size for the given target capacity.
 */
static final int tableSizeFor(int cap) {
    int n = -1 >>> Integer.numberOfLeadingZeros(cap - 1);
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

10. string为什么不可变

```
public final class String implements java.io.Serializable, Comparable<String>, CharSequence {

    private final char value[];

    private int hash; // Default to 0
}
```

1. 采用final修饰，类不可变继承，值不可被修改。否则才用string作为key时，一点被修改就找不到value

1. 字符串常量池。A、B都指向同一个常量池，如果string可变，A改变了常量池中的值，B就会指向错误的值。

1. 缓存hash值。不需要每次都反复计算，提高效率，以为string不可变，不用担心hash值被改变。
2. 安全性。String被作为参数广泛应用java类、网络连接、文件操作中，如果string可变，会引发安全问题。
3. 不可变对象天生就是线程安全的。get的时候，不是返回对象本身，而是返回的copy后的值。

11. hashMap如何解决hash冲突

一般解决hash冲突的方法有两种：

- 开放地址法
- 链地址法

而HashMap就是采用链地址法。

1.开放地址法：容易产生堆积问题；不适于大规模的数据存储；散列函数的设计对冲突会有很大的影响；插入时可能会出现多次冲突的现象，删除的元素是多个冲突元素中的一个，需要对后面的元素作处理，实现较复杂；结点规模很大时会浪费很多空间；

2.链地址法：处理冲突简单，且无堆积现象，平均查找长度短；链表中的结点是动态申请的，适合构造表不能确定长度的情况；相对而言，拉链法的指针域可以忽略不计，因此较开放地址法更加节省空间。插入结点应该在链首，删除结点比较方便，只需调整指针而不需要对其他冲突元素作调整。

12. HashMap的扩容

```
size      ：表示HashMap里面put进去元素的个数
threshold ：元素个数的阈值,capacity * factor
factor    ：扩容因子
capacity  ：数组容量，初始默认为16

在put元素的时候：
// 超过最大容量，就扩容
if (++size > threshold) {
   resize();
}
```

注意：resize方法的作用有两个：

- 初始化HashMap
- 扩容HashMap

JDK1.8之后，在HashMap扩容的时候，不会出现链表反转的情况了。

```java
final HashMap.Node<K,V>[] resize() {
    // ......省略前面的内容,前面主要就是根据当前数组的大小，确定newCap、threshold等
    HashMap.Node<K,V>[] newTab = (HashMap.Node<K,V>[]) new HashMap.Node[newCap];
    table = newTab;
    if (oldTab != null) {
        for (int j = 0; j < oldCap; ++j) {
            HashMap.Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                // 如果只有一个Node
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                // 如果是红黑树
                else if (e instanceof HashMap.TreeNode)
                    ((HashMap.TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                // 如果是链表
                else {
                    HashMap.Node<K,V> loHead = null, loTail = null;
                    HashMap.Node<K,V> hiHead = null, hiTail = null;
                    HashMap.Node<K,V> next;
                    do {
                        next = e.next;
                        // 注意确定index的时候，是hash & (oldCap-1)
                        // 这里是确定e.hash值是在高位high还是低位low
                        // 其实道理很简单：比如oldCap = 16 (对应二进制位：0001 0000)
                        // 如果hash & oldCap != 0。说明hash的形式为：... xxx1 xxxx,扩容后应该属于高
位
                        if ((e.hash & oldCap) == 0) {
                            // 同时维护低位的头部和尾部。如果loTail不为空，直接将新的节点放在尾部的
next
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            // 同时维护高位的头部和尾部。如果hiTail不为空，直接将新的节点放在尾部的
next
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);

                    // 将低位的头部放入数组中，index = j
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
```

```
                    // 将高位的头部放入数组中，index = j + oldCap
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
    }
    return newTab;
}

// 将TreeBin转成low、high两个TreeBin，不是先用Node，而是直接拆成TreeBin。
// 如果拆分后的TreeBin的长度<=6时，再转成链表。
final void split(HashMap<K,V> map, Node<K,V>[] tab, int index, int bit) {
    TreeNode<K,V> b = this;
    // Relink into lo and hi lists, preserving order
    TreeNode<K,V> loHead = null, loTail = null;
    TreeNode<K,V> hiHead = null, hiTail = null;
    int lc = 0, hc = 0;       // lowCount、highCount
    for (TreeNode<K,V> e = b, next; e != null; e = next) {
        next = (TreeNode<K,V>)e.next;
        e.next = null;
        if ((e.hash & bit) == 0) {
            if ((e.prev = loTail) == null)
                loHead = e;
            else
                loTail.next = e;
            loTail = e;
            ++lc;
        }
        else {
            if ((e.prev = hiTail) == null)
                hiHead = e;
            else
                hiTail.next = e;
            hiTail = e;
            ++hc;
        }
    }

    if (loHead != null) {
        if (lc <= UNTREEIFY_THRESHOLD)    // UNTREEIFY_THRESHOLD = 6
            tab[index] = loHead.untreeify(map);
        else {
            tab[index] = loHead;
            if (hiHead != null)           // (else is already treeified)
                loHead.treeify(tab);
        }
    }
    if (hiHead != null) {
        if (hc <= UNTREEIFY_THRESHOLD)

            tab[index + bit] = hiHead.untreeify(map);
```

```
            else {
                tab[index + bit] = hiHead;
                if (loHead != null)
                    hiHead.treeify(tab);
            }
        }
    }
```

2、synchronized

1. monitor对象

每个对象或实例都与一个monitor对象相关联，monitor的lock锁同一时间只会被一个线程获得。

- 如果monitor的计数为0，没有线程获得该monitor的锁
- 如果线程重入，monitor的计数+1
- 其他线程想获得monitor的所有权，会陷入阻塞状态直到monitor的计数为0，才能尝试获得。

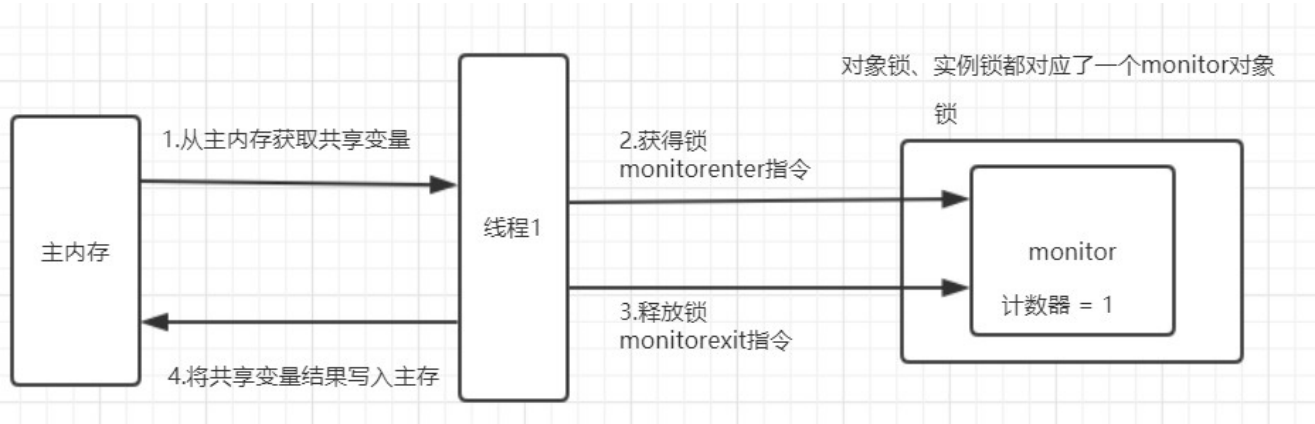
当Thread调用锁对象的wait方法后，该Thread就会被加入该对象monitor关联的`wait set`中，并释放monitor的所有权

- 锁对象调用notify方法：其中一个线程从wait set弹出（顺序不确定）
- 锁对象调用notifyAll方法：`wait set`所有线程都被弹出

2. synchronized的缺点

synchronized关键字包含两个JVM指令：`monitorenter`和`monitorexit`。

在monitor exit运行成功后，共享变量的值必须刷入主内存中，因此monitor enter成功之前都必须从主内存中获取数据，而不是从缓存中。



synchronized只能用来修饰方法和代码块，不能修饰类的变量。

缺点：

- 无法控制阻塞时长。无法设置超时的功能
- 一旦阻塞，便不可中断

```
public class SyncTest {
    public static void main(String[] args) throws InterruptedException {
        SyncTest syncTest = new SyncTest();
        Thread t1 = new Thread(syncTest::syncMethod, "t1");
        t1.start();
        t1.interrupt();   // t1持有monitor的锁。调用interrupt方法可以直接中断

        TimeUnit.MILLISECONDS.sleep(10);  // 确保t1先进入syncMethod

        Thread t2 = new Thread(syncTest::syncMethod, "t2");
        t2.start();
        t2.interrupt();   // t2为等待t1结束进入了阻塞状态。调用interrupt方法无效
    }

    public synchronized void syncMethod() {
        try {
            System.out.println("sleep 1 hour");
            TimeUnit.HOURS.sleep(1);
        } catch (InterruptedException ex) {
            System.out.println("被中断");
        }
    }
}

说明：
  1.T2线程何时获得执行syncMethod方法，完全取决于T1何时释放。无法实现T2最多等1分钟就放弃的功能
  2.一旦T2进入阻塞状态，他将无法被中断。因为synchronized阻塞无法向sleep和wait一样，捕捉到interrupt信号。所以现在争抢锁的过程中，无法获得锁的线程只能一直等待下去
```

3. CAS (Compare And Swap)

通过指令：cmpxchg

```
AtomicInteger count = new AtomicInteger(0);
// +1
int i = count.incrementAndGet();
```

```
// 获取当前对象的value字段，在这个对象内存中偏移量
// Unsafe类可以直接操作内存
valueOffset = unsafe.objectFieldOffset(AtomicInteger.class.getDeclaredField("value"));

public final int incrementAndGet() {
    // this:当前AtomicInteger对象
    // valueOffset
    return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}
```

```
public final int getAndAddInt(Object atomicObj, long valueOffset, int plusValue) {
    int lastValue;
    // 循环尝试CAS操作，直到成功。
    do {
        lastValue = this.getIntVolatile(atomicObj, valueOffset);
    } while(!this.compareAndSwapInt(atomicObj, valueOffset, lastValue, lastValue + plusValue));
    // 返回成功前，上一次的值
    return lastValue;
}
```

4. this和class锁的区别

this是实例锁，class是对象锁。
静态方法中，实例还没有创建，只能用对象锁
非静态方法或者代码块中，采用this时，相同实例只能有一个线程获得锁，不同实例可以同时获取锁。采用class时，所有线程只有一个线程获得锁

3、ConcurrentHashMap

1. jdk1.8之后的优化

- HashMap在并发情况下put会出现死循环，导致CPU使用率接近100%。
- HashTable虽然线程安全，但是由于所有线程都会竞争同一把锁，导致效率低下。

JDK1.8之前：

ConcurrentHashMap就是采用`分段锁Segment`，当访问的数据需要跨段的时候，按顺序获取锁，最后按顺序释放锁。这样能防止出现死锁。

```
ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。

Segment是一种可重入锁ReentrantLock，扮演锁的角色，HashEntry则用于存储键值对数据。

一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。
```

JDK1.8之后：

https://www.jianshu.com/p/c02a5627d0a5

主要设计上的变化有以下几点:

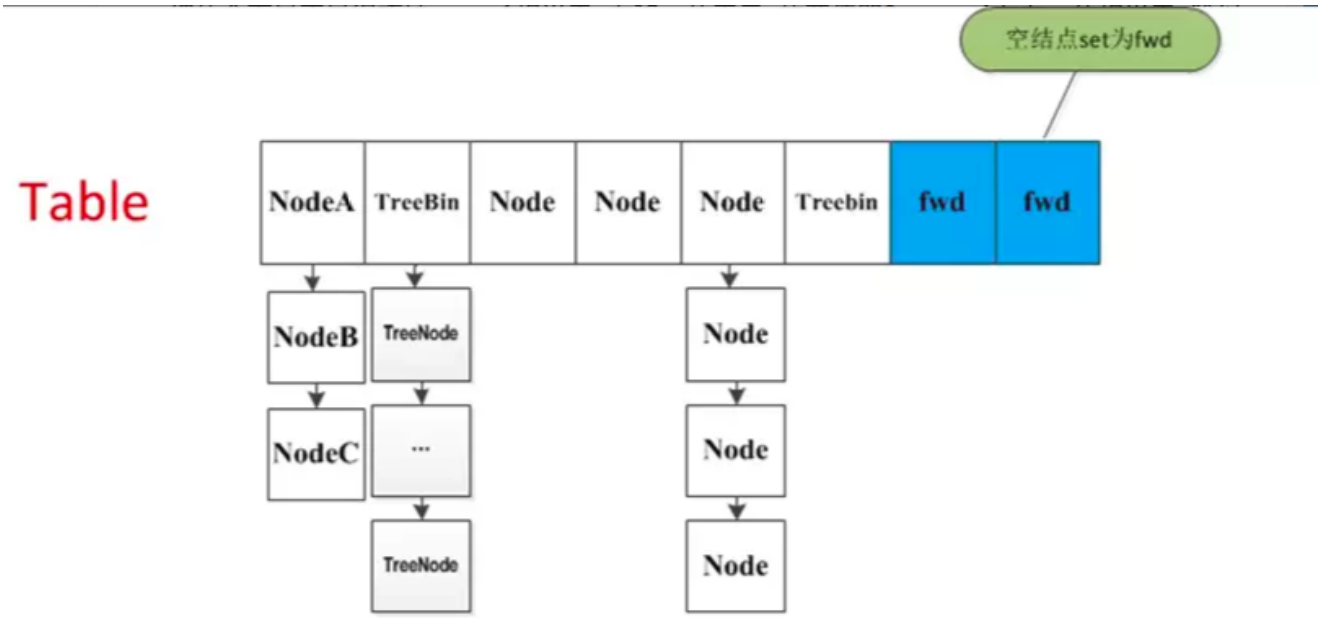
- 不采用segment而采用node，锁住node来实现减小锁粒度。
- 设计了MOVED状态，当resize的中过程中 线程2还在put数据，线程2会帮助resize。
- 使用3个CAS操作来确保node的一些操作的原子性，这种方式代替了锁。
- sizeCtl的不同值来代表不同含义，起到了控制的作用。volatile修饰，-1表示正在初始化
- 采用synchronized而不是ReentrantLock。

CorruentHashMap是采用`懒加载的方式`，new的时候是没有初始化的，而是在第一次put数据才调用initTable。

```java
// -1表示正在初始化，-n表示有n-1个线程正在进行扩容操作
private transient volatile int sizeCtl;
// U.compareAndXXXX函数实际就是利用CAS算法保证了线程安全性。
private static final Unsafe U = Unsafe.getUnsafe();

private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        if ((sc = sizeCtl) < 0)
            // 1. 保证只有一个线程正在进行初始化操作，等待其他线程完成初始化
            Thread.yield();
        // SIZECTL = U.objectFieldOffset(ConcurrentHashMap.class, "sizeCtl");
        // 这里就拿到了ConcurrentHashMap类中变量sizeCtl在内存中的位置(offset)
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    // 2. 得出数组的大小
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    // 3. 这里才真正的初始化数组
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    // 4. 计算数组中可用的大小：实际大小n*0.75（加载因子）
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    //1. 计算key的hash值
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        //2. 如果当前table还没有初始化先调用initTable方法将tab进行初始化
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        //3. tab中索引为i的位置的元素为null，则直接使用CAS将值插入即可
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;  // no lock when adding to empty bin
        }
        //4. MOVED表示当前正在扩容,当前线程帮助扩容
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    //5. 当前为链表，在链表中插入新的键值对
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,
                                                          value, null);
                                break;
                            }
                        }
                    }
                    // 6.当前为红黑树，将新的键值对插入到红黑树中
                    else if (f instanceof TreeBin) {
                        Node<K,V> p;
                        binCount = 2;
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                       value)) != null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)

                                p.val = value;
```

```
                        }
                    }
                }
            }
            // 7.插入完键值对后再根据实际大小看是否需要转换成红黑树
            if (binCount != 0) {
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    //8.对当前容量大小进行检查，如果超过了临界值（实际大小*加载因子）就需要扩容
    addCount(1L, binCount);
    return null;
}
```



2. 如何实现线程安全

多个线程在对数组的同一个Node进行put操作时，通过CAS控制只有一个线程更新成功，其他线程根据新的值重试。

对不同的Node进行put时，互相不影响。

3. 扩容时读写操作如何进行

- 对于get读操作，如果当前节点有数据，还没迁移完成，此时不影响读，能够正常进行。 如果当前链表已经迁移完成，那么头节点会被设置成fwd节点，此时get线程会帮助扩容。
- 对于put/remove写操作，如果当前链表已经迁移完成，那么头节点会被设置成fwd节点，此时写线程会帮助扩容，如果扩容没有完成，当前链表的头节点会被锁住，所以写线程会被阻塞，直到扩容完成。

为什么不用ReentrantLock而是Synchronzied呢？

synchronzied做了很多的优化，包括偏向锁，轻量级锁，重量级锁，可以依次向上升级锁状态。

因此，使用synchronized相较于ReentrantLock的性能会持平甚至在某些情况更优。

什么是CAS无锁？Compare And Swap（比较交换）

与锁相比，使用CAS会使程序看起来更加复杂一些。但由于其非阻塞性，它对死锁问题天生免疫，并且，线程间的相互影响也远远比基于锁的方式要小。更为重要的是，使用无锁的方式完全没有锁竞争带来的系统开销，也没有线程间频繁调度带来的开销，因此，它要比基于锁的方式拥有更优越的性能。

CAS算法的过程是这样：它包含三个参数CAS(V,E,N)：V表示要更新的变量，E表示预期值，N表示新值。仅当V值等于E值时，才会将V的值设为N，如果V值和E值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。最后，CAS返回当前V的真实值。

CAS操作是抱着乐观的态度进行的，它总是认为自己可以成功完成操作。当多个线程同时使用CAS操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理，CAS操作即使没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

CAS需要你额外给出一个期望值，也就是你认为这个变量现在应该是什么样子的。如果变量不是你想象的那样，那说明它已经被别人修改过了。你就重新读取，再次尝试修改就好了。

```
/**
* 比较obj的offset处内存位置中的值和expect的值，
* 如果相同则更新为update的值，此更新是不可中断的。
*
* @param obj    需要更新的对象
* @param offset obj中整型field的偏移量
* @param expect 希望field中存在的值
* @param update 如果期望值expect与field的当前值相同，设置filed的值为这个新值
* @return 如果field的值被更改返回true
*/
public native boolean compareAndSwapInt(Object obj, long offset, int expect, int update);
```

CPU的汇编指令cmpxchg就是实现CAS功能。

4、AQS

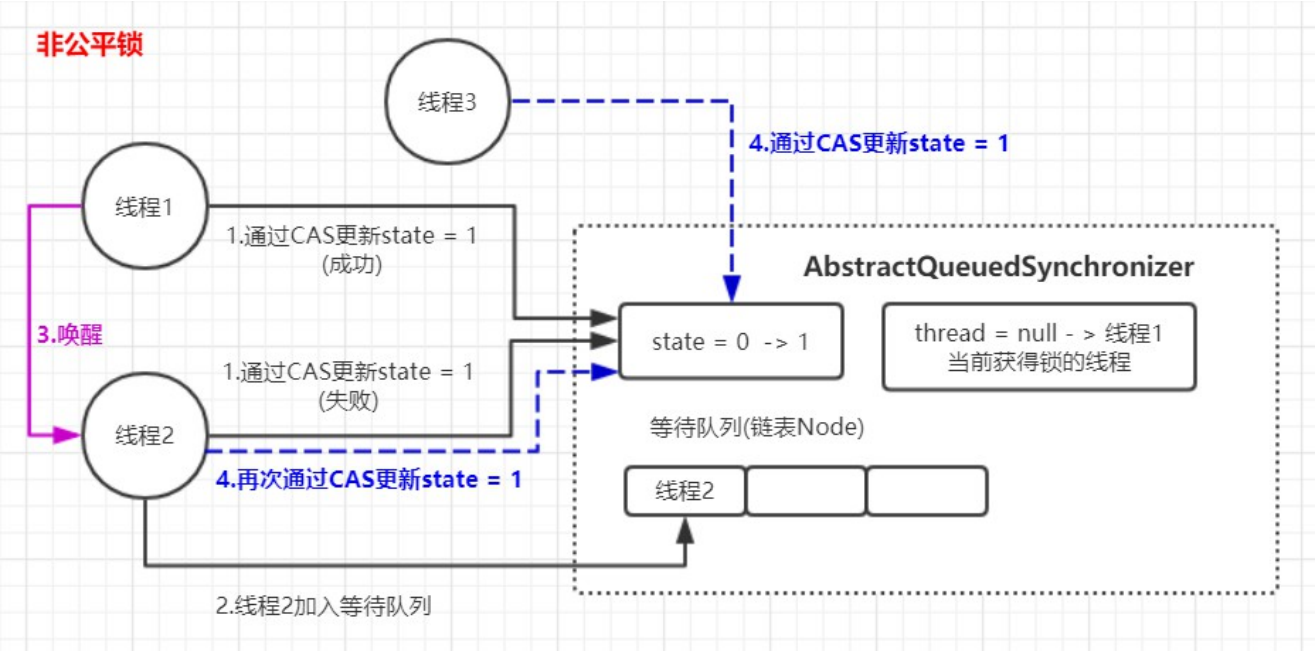
AbstractQueuedSynchronizer.class是并发包的一个类。抽象同步队列

```
// 底层就是利用了AQS
ReentrantLock lock = new ReentrantLock();        // 非公平锁
ReentrantLock lock = new ReentrantLock(true);    // 公平锁
lock.lock();
...
lock.unlock();
```

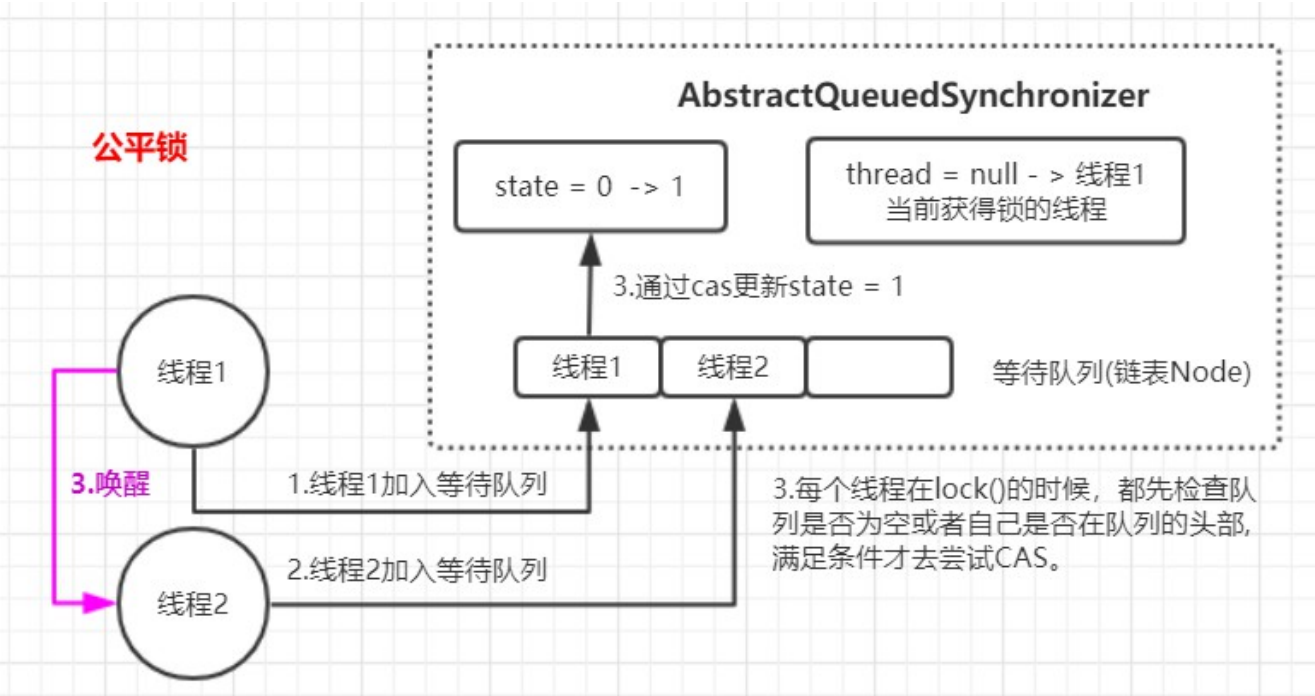
```
// ---------------- 公平锁 --------------------
final void lock() {
    acquire(1);
}

/**
 * Fair version of tryAcquire.  Don't grant access unless
 * recursive call or no waiters or is first.
 */
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) { // state=0,表示当前锁没有被占用
        // 队列为空 或 当前线程在队列的头部后，才尝试CAS操作
        if (!hasQueuedPredecessors() && compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

执行流程：

1. 多个线程都通过CAS操作，尝试更新`state = 1`。成功表示获得该锁
2. state = 1; thread = 获取锁成功的线程。失败的线程放入等待队列
3. 对于非公平锁：线程1执行结束后，会唤醒线程2，线程2再次尝试，但是如果线程3也在尝试。不保证线程2一定能抢到锁。失败的，再次加入等待队列
4. 对于公平锁：线程1结束后，线程3如果想获得锁，必须先去检查等待队列。队列不为空，然后线程2去获得锁。线程3加入等待队列。




```
// ---------------- 非公平锁 --------------------
final void lock() {
    // 直接CAS尝试获得锁
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}
```



总结：公平锁和非公平锁只有两处不同：

1. 非公平锁在调用 lock 后，**首先就会调用 CAS 进行一次抢锁**，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。

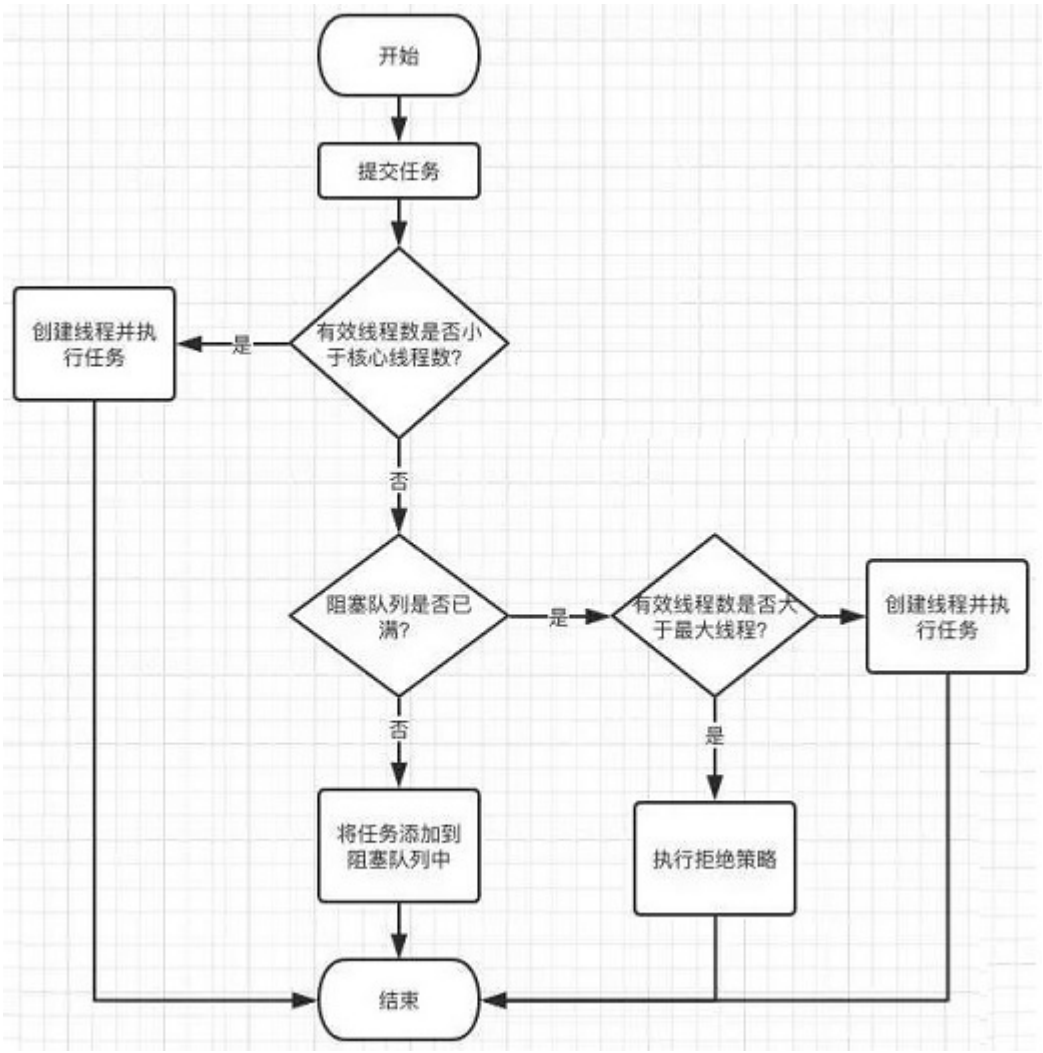
2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），**非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁**，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

5、线程池的原理

```
Q. 线程池是什么时候创建线程的？
A. 任务提交的时候

Q.线程池如何执行任务
（1）当线程数小于核心线程数时，创建线程。
（2）当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。
（3）当线程数大于等于核心线程数，且任务队列已满
    1）若线程数小于最大线程数，创建线程
    2）若线程数等于最大线程数，执行拒绝策略。

Q. 什么时候会触发reject策略？
A.队列满并且maxthread也满了， 还有新任务，默认策略是reject

Q. core到maxThread之间的线程什么时候会die?
A.  没有任务时，或者抛异常时。
   core线程也会die的，core到maxThread之间的线程有可能会晋升到core线程区间，
   core max只是个计数，线程并不是创建后就固定在一个区间了

Q. task抛出异常，线程池中这个work thread还能运行其他任务吗?
A. 不能。但是会创建新的线程，新线程可以运行其他task。有异常时旧的thread会被删除（GC回收），再创建新的thread， 即有异常时，旧thread不可能再执行新的任务。
```



5.1 java的四种线程池

Java通过Executors提供四种线程池，分别为：
1,newCachedThreadPool
创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
2,newFixedThreadPool
创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
3,newScheduledThreadPool
创建一个定长线程池，支持定时及周期性任务执行。
4,newSingleThreadExecutor
创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

5.2 拒绝策略

可以理解为达到最大线程数时，一个回调函数。

JDK默认有4个实现类，也可以自定义拒绝策略。

`AbortPolicy（默认）是抛出异常`，DiscardPolicy是直接将不能执行的任务删除。

```
从字面上理解为拒绝处理者,可以理解为当任务数大于maxmumPoolSize后的一个回调方法,RejectedExecutionHandler
本身是一个接口,jdk本身对他有4个实现类

// 线程调用运行该任务的 execute 本身。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度
CallerRunsPolicy

// 处理程序遭到拒绝将抛出运行时RejectedExecutionException;
AbortPolicy(默认)

// 不能执行的任务将被删除
DiscardPolicy

// 如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序（如果再次失败，则重复此过程）
DiscardOldestPolicy
```

建议：

自定义一个reject策略，如果线程池无法执行更多的任务了，此时建议可以把task信息持久化写入磁盘中，后续等待线程池的工作负载降低了。后台再启动一个线程，从磁盘读取task，从新提交到线程池中去执行。

5.3 使用线程池的注意事项

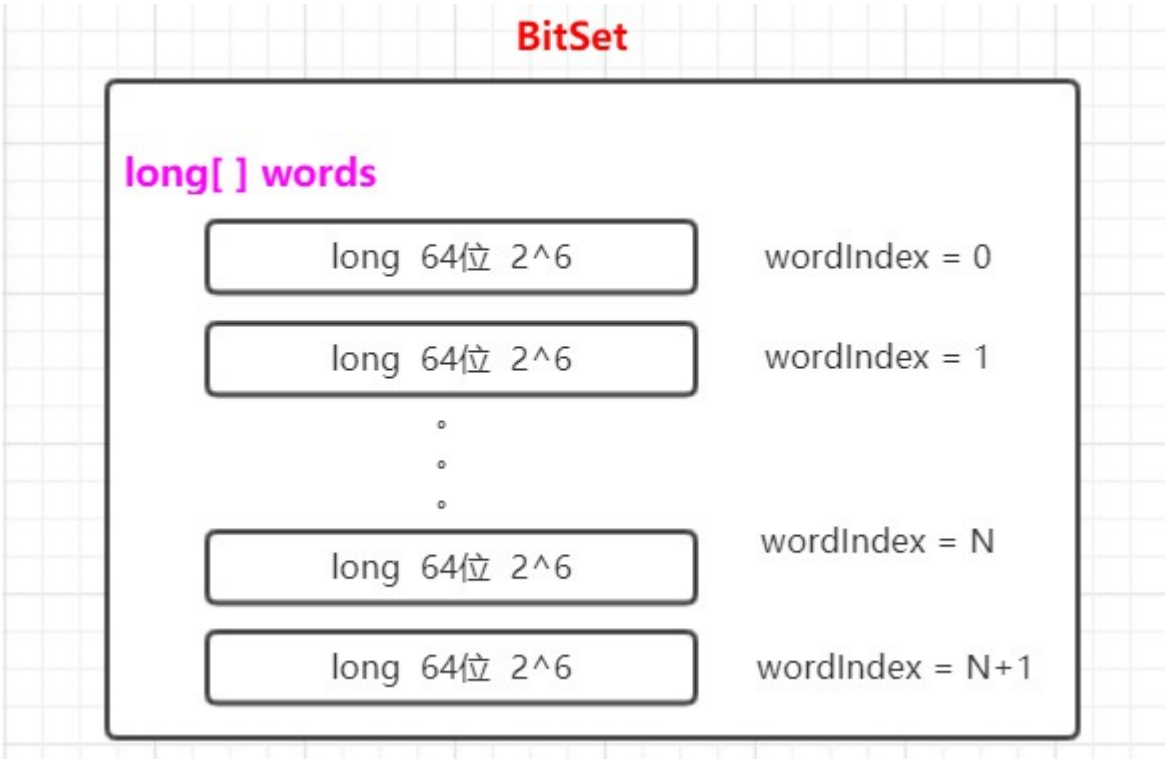
```
newFixedThreadPool => return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>());
阻塞队列基本是无界的。不会触发线程池的拒绝策略。
可能队列里面积累的任务数目，直接将内存撑爆了。。。

------------------------------------------------------------------------
newCachedThreadPool => return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>());
最大线程数为Integer.MAX_VALUE，而且阻塞队列是无界的，不会触发线程池的拒绝策略。
可能会创建很多线程池，造成浪费。而且单机线程数不能无限上升，在高并发情况下，如果线程调用的外部服务异常了，
每个都在等待，会导致新来的请求又重新创建线程。结果就有几千个线程。。。机器挂了
```

5.4 机器宕机导致线程队列丢失

当机器宕机的时候，必然会导致在线程池里面积压的任务丢失，所以在提交一个task到线程池之前，先将task信息持久化到数据库中，设置状态：`未提交`、`已提交`、`已完成`

系统重启的时候，从数据库扫描`未提交`和`已提交`的task，重新提交到线程池进行执行。

6、BitSet



1、set流程

```
//按位存储数据。比如65，就会将第65个bit位上的二进制数设置为1.
private long[] words;

// words是long类型的数组。long是64位，2^6.
// 所以每个需要存储的数据的wordIndex = bitIndex >> 6;
private final static int ADDRESS_BITS_PER_WORD = 6;

public void set(int bitIndex) {
    if (bitIndex < 0)
        throw new IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);
    // step1: 获取set的元素在words数组中的索引位置wordIndex
    int wordIndex = bitIndex >> ADDRESS_BITS_PER_WORD;
    // step2: words数组长度如果不包含该wordIndex，对words数组进行扩容
    expandTo(wordIndex);
    // step3: 将wordIndex对应位置的0设置为1
    words[wordIndex] |= (1L << bitIndex);

    checkInvariants();
}
```

2、Bloom过滤器

Bloom过滤器主要用于判断一个元素是否在一个集合中，它可以使用一个位数组简洁的表示一个数组。它的空间效率和查询时间远远超过一般的算法，但是它存在一定的误判的概率(所以通常会计算8个hash值)，适用于容忍误判的场景。如果布隆过滤器判断元素存在于一个集合中，那么大概率是存在在集合中，如果它判断元素不存在一个集合中，那么一定不存在于集合中。

通过这样的一个算法，可以无需将字符串的MD5值存储在内存中，只需要定义一定长度的Bitset即可，从而大大节约了空间。

Bloom过滤器的原理为：
将一个字符串通过一定算法映射为八个Hash值，将八个Hash值对应位置的Bitset位进行填充。在进行校验的时候，通过同样的算法计算八个Hash值，八个Hash值全部存在才可以认定为该字符串在集合中存在。

Bloom过滤器可以广泛应用于判断集合中是否存在某个元素的大量数据场景(大数据去重)，比如黑名单、爬虫访问记录。

```java
// 简单的布隆过滤器实现
public class BloomFilter {
    private static final int DEFAULT_SIZE = 2 << 24;
    // 基于8个不同的种子值，计算字符串的hash值
    private static final int[] seeds = new int[] { 5, 7, 11, 13, 31, 37, 61 };
    // 一个字符串对应8个hash值会存储在BitSet中
    private BitSet bits = new BitSet(DEFAULT_SIZE);
    // 8种就是那hash值的方法
    private SimpleHash[] func = new SimpleHash[seeds.length];

    public BloomFilter() {
        for (int i = 0; i < seeds.length; i++) {
            func[i] = new SimpleHash(DEFAULT_SIZE, seeds[i]);
        }
    }

    public void add(String value) {
        for (SimpleHash f : func) {
            bits.set(f.hash(value), true);
        }
    }

    public boolean contains(String value) {
        if (value == null) {
            return false;
        }
        boolean ret = true;
        for (SimpleHash f : func) {
          // 必须8个hash值都存在于BitSet中，才能返回true
            ret = ret && bits.get(f.hash(value));
        }
        return ret;
    }

    // 内部类，simpleHash
    public static class SimpleHash {
        private int cap;
        private int seed;

        public SimpleHash(int cap, int seed) {
            this.cap = cap;
            this.seed = seed;
        }

        public int hash(String value) {
            int result = 0;
            int len = value.length();
            for (int i = 0; i < len; i++) {
                result = seed * result + value.charAt(i);
            }
            return (cap - 1) & result;
        }

    }
```

```
    public static void main(String[] args) {
        BloomFilter bf = new BloomFilter();
        List<String> strs = new ArrayList<>();
        strs.add("123456");
        strs.add("hello word");
        strs.add("transDocId");
        strs.add("123456");
        strs.add("transDocId");
        strs.add("hello word");
        strs.add("test");
        for (int i=0;i<strs.size();i++) {
            String s = strs.get(i);
            boolean bl = bf.contains(s);
            if(bl){
                System.out.println(i+","+s);
            }else{
                bf.add(s);
            }
        }
    }
}
```

7、Unsafe类

Unsafe类使Java拥有了像C语言的指针一样操作内存空间的能力，同时也带来了指针的问题。过度的使用Unsafe类会使得出错的几率变大。

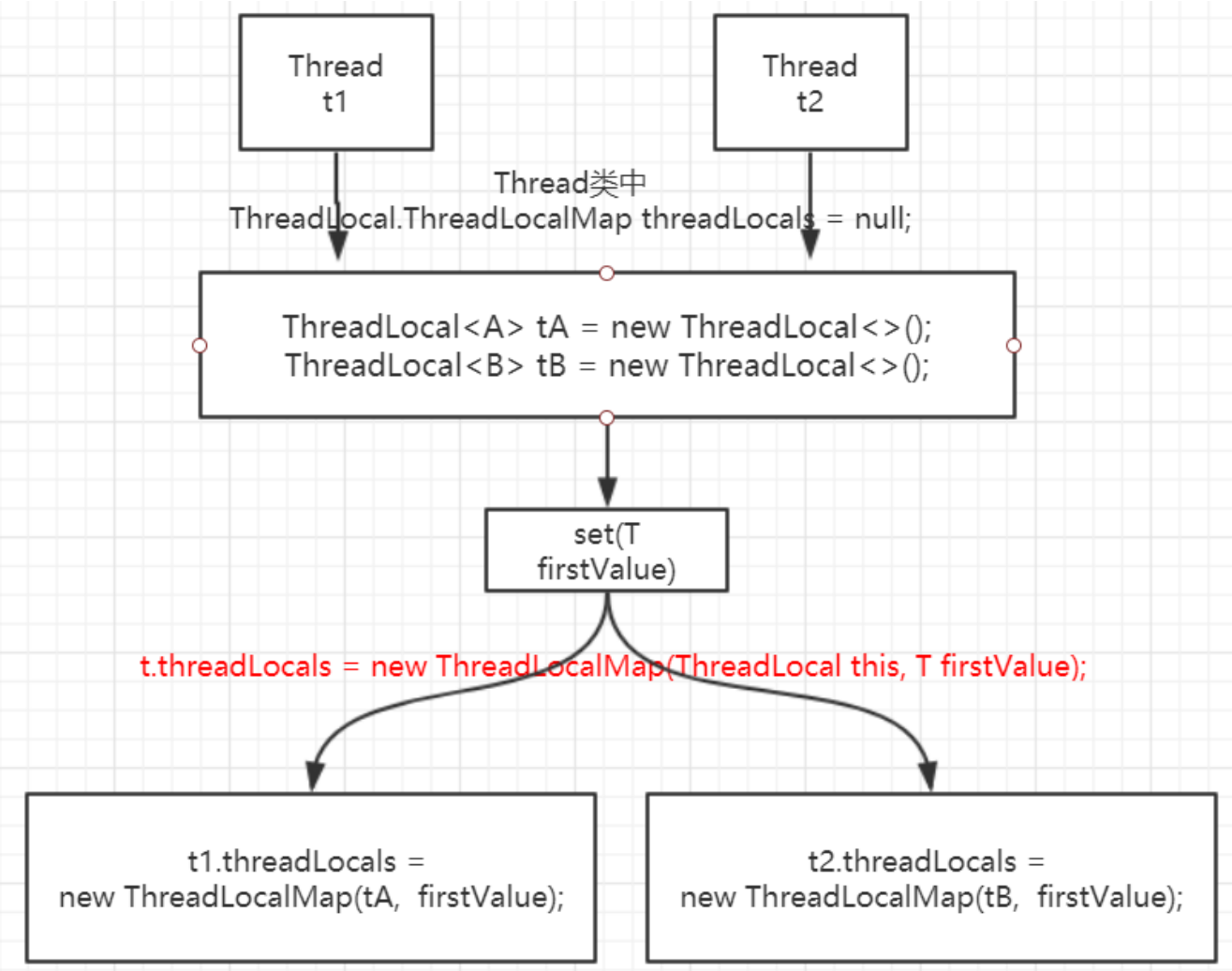
JDK1.5之后，java程序才可以使用CAS操作，该操作由Unsafe类里边的compareAndSwapInt等几个方法包装提供。

8、ThreadLocal类

http://www.sohu.com/a/297777807_753508

主要是存储线程内某个变量，永远不会被其他线程访问到。

Thread类中包含了变量threadLocals，是ThreadLocalMap类型。默认情况都是null，只有当ThreadLocal变量第一次set值的时候才会赋值。



注意：

- 一个线程里面可能有很多ThreadLocal对象，都是存储在t.threadLocals中。ThreadLocalMap的key值就是ThreadLocal对象。
- 因此如果当前线程的生命周期很长，一直存在，那么其内部的ThreadLocalMap对象也一直生存下来，这些null key就存在一条强引用链的关系一直存在：Thread --> ThreadLocalMap–>Entry–>Value，这条强引用链会导致Entry不会回收，Value也不会回收，但Entry中的Key却已经被回收的情况，造成内存泄漏。`比如在for循环中大量创建threadlocal变量`
- 为了尽量保证ThreadLocal不会内存泄漏： 在ThreadLocal的get()、set()、remove()方法调用的时候会通过getEntryAfterMiss方法，清除掉线程ThreadLocalMap中所有Entry中Key为null的Value，并将整个Entry设置为null，利于下次内存回收。
- ThreadLocal中使用了`斐波那契散列法`，来保证哈希表的离散度。而它选用的系数值`private static final int HASH_INCREMENT = 0x61c88647`，即是`2^32 * 黄金分割比`，(Math.sqrt(5) - 1)/2
- ThreadLocalMap中的Entry数组是环形数组，在遍历的时候，最后一个指向第一个。

`Entry继承WeakReference类后，会在内存不足的时候，对Entry进行回收。而ThreadLocal会在set的时候检查Entry，如果是null，则会删除该Entry。`

```
/*
     ThreadLocal values pertaining to this thread.
     This map is maintained by the ThreadLocal class.
*/
ThreadLocal.ThreadLocalMap threadLocals = null;

// key的哈希值采用斐波那契散列法获得。HASH_INCREMENT为系数，等于1640531527
private static final int HASH_INCREMENT = 0x61c88647;

private static int nextHashCode() {
  return nextHashCode.getAndAdd(HASH_INCREMENT);
}
```

```
void createMap(Thread t, T firstValue) {
    // this为ThreadLocal的实例化对象
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

ThreadLocalMap的构造方法

```
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
    // ThreadLocalMap的底层是Entry数组，Entry对象是WeakReference的子类。
    table = new Entry[INITIAL_CAPACITY];
    // 根据 ThreadLocal 的散列值，查找对应元素在数组中的位置
    int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
    table[i] = new Entry(firstKey, firstValue);
    size = 1;
    setThreshold(INITIAL_CAPACITY);
}
```

Entry对象的定义

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
```

set方法详解

```
private void set(ThreadLocal<?> key, Object value) {
    Entry[] tab = table;
    int len = tab.length;
    // 计算出索引的位置
    int i = key.threadLocalHashCode & (len-1);

    // 从索引位置开始遍历,由于不是链表结构,因此通过nextIndex方法来寻找下一个索引位置
    for (Entry e = tab[i];
         e != null; // 当遍历到的Entry为空时结束遍历
         e = tab[i = nextIndex(i, len)]) {
        ThreadLocal<?> k = e.get(); // 拿到Entry的key，也就是ThreadLocal

        // 该Entry的key和传入的key相等，则用传入的value替换掉原来的value
        if (k == key) {
            e.value = value;
            return;
        }

        // 该Entry的key为空，则代表该Entry需要被清空，
        // 调用replaceStaleEntry方法
        if (k == null) {
            // 该方法会继续寻找传入key的安放位置，并清理掉key为空的Entry
            replaceStaleEntry(key, value, i);
            return;
        }
    }

    // 寻找到一个空位置， 则放置在该位置上
    tab[i] = new Entry(key, value);
    int sz = ++size;
    // cleanSomeSlots是用来清理掉key为空的Entry,如果此方法返回true,则代表至少清理
    // 了1个元素，则此次set必然不需要扩容，如果此方法返回false则判断sz是否大于阈值
    if (!cleanSomeSlots(i, sz) && sz >= threshold)
        rehash();   // 扩容
}
```

二、GC

1、垃圾回收算法

- 标记-清除：产生大量内存碎片，后期无法给大对象分配内存空间，提前触发FGC
- 复制：将内存分成两块，每次只用一块。GC后将一块所有存活对象复制到另一块。缺点：牺牲内存
- 标记-整理：但是后续的整理却不是直接将可回收的对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉边界以外的内存
- 分代收集算法：将堆分为新生代和老年代，然后采用不同的算法。

  新生代每次回收只会有少量对象存活，采用复制算法。

  老年代对象存活率高，没有额外的空间。采用标记清除或标记整理。

评估一个GC：

- GC的停顿时间：会stop the world使内存保持一个一致的状态进行回收。对应用程序会有影响
- GC的吞吐量：如果程序不面向用户，可能不关注停顿时间。而是关注吞吐量

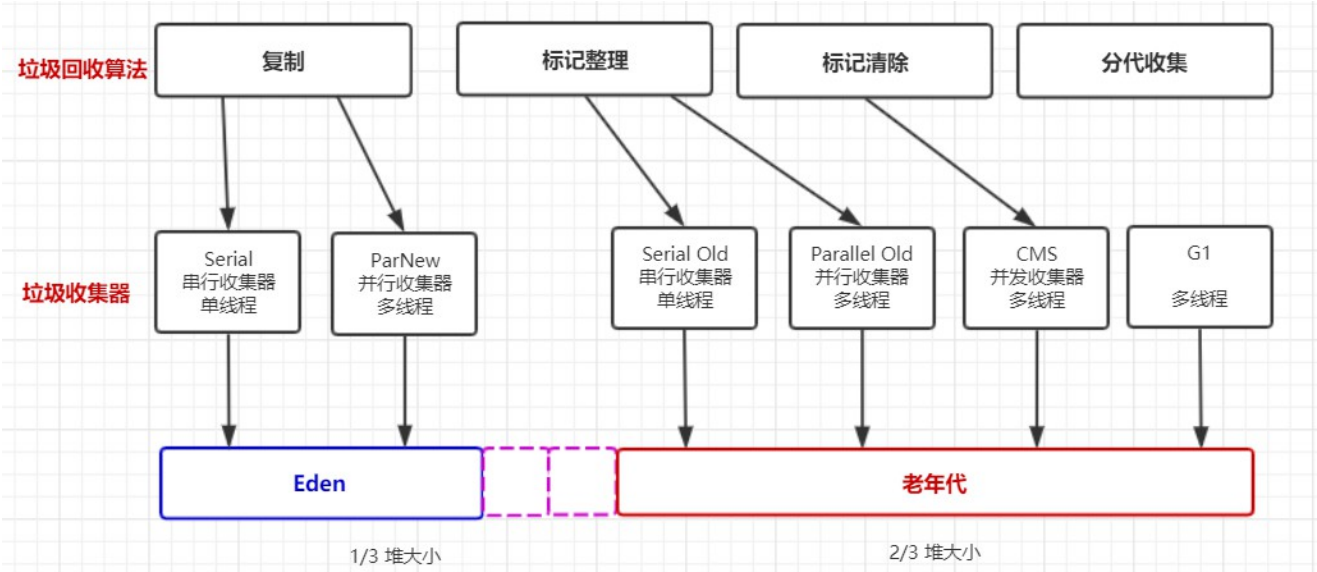
2、垃圾收集器

名称	工作区域	单/多线程	算法	特点
Serial	新生代	单	复制	1.stop the world 2.-XX:+UseSerialGC配置 3.client模式下默认新生代收集器
ParNew	新生代	多	复制	1.stop the world 2.Server模式下默认新生代收集器
Parallel scavenge	新生代	多	复制	吞吐量优先收集器，目标就是控制吞吐量 吞吐量=代码运行时间/（代码运行时间+垃圾收集时间） -XX:MaxGCPauseMillis　设置最大垃圾收集时间 -XX:+UserParallelGC
Serial Old	老年代	单	标记整理	1.client模式下默认新生代收集器 2.Server端作为CMS的后备方案。在并发收集发生Concurrent Mode Failure时使用
Parallel Old	老年代	多	标记整理	吞吐量优先的收集器组合, jdk1.6后，充分利用了多CPU：Parallel scavenge + Parallel Old
CMS	老年代	多	标记清除	优点：并发收集，低停顿，但吞吐量降低
G1	老年代	多		

2.1 图解垃圾回收

新生代：每次GC存活的对象很少，直接采用复制算法

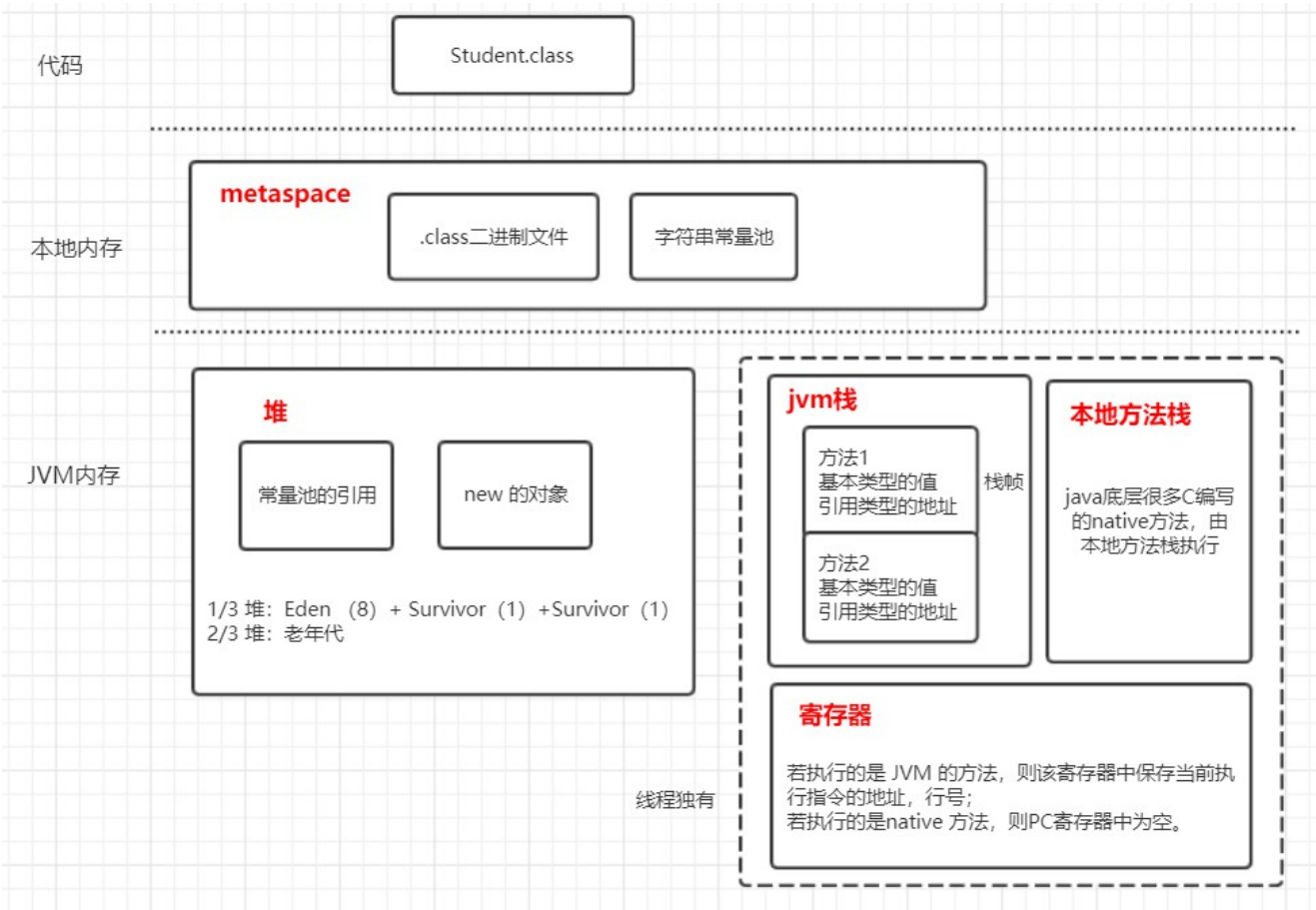
老年代：每次GC，会被GC的对象不多，所以一般采用标记整理、标记清除



CMS：(尽量要STW的时间短) 响应时间优先，减少垃圾收集停顿时间，但是吞吐量降低。
- CPU资源敏感：在并发标记和并发清除阶段，虽然不会导致线程停顿，但会占用部分CPU，导致程序缓慢，吞吐量降低
- 无法处理浮动垃圾：在并发清除阶段仍然会产生垃圾，无法被此次收集。
- 内存碎片问题：由于采用标记清除算法，没有整理

G1：
- 采用Region对堆进行划分
- 根据MaxGCPauseMillis，GC会优先对收益高的Region进行GC
- 对象的分配增加了Humongous区域

2.2 图解java内存结构

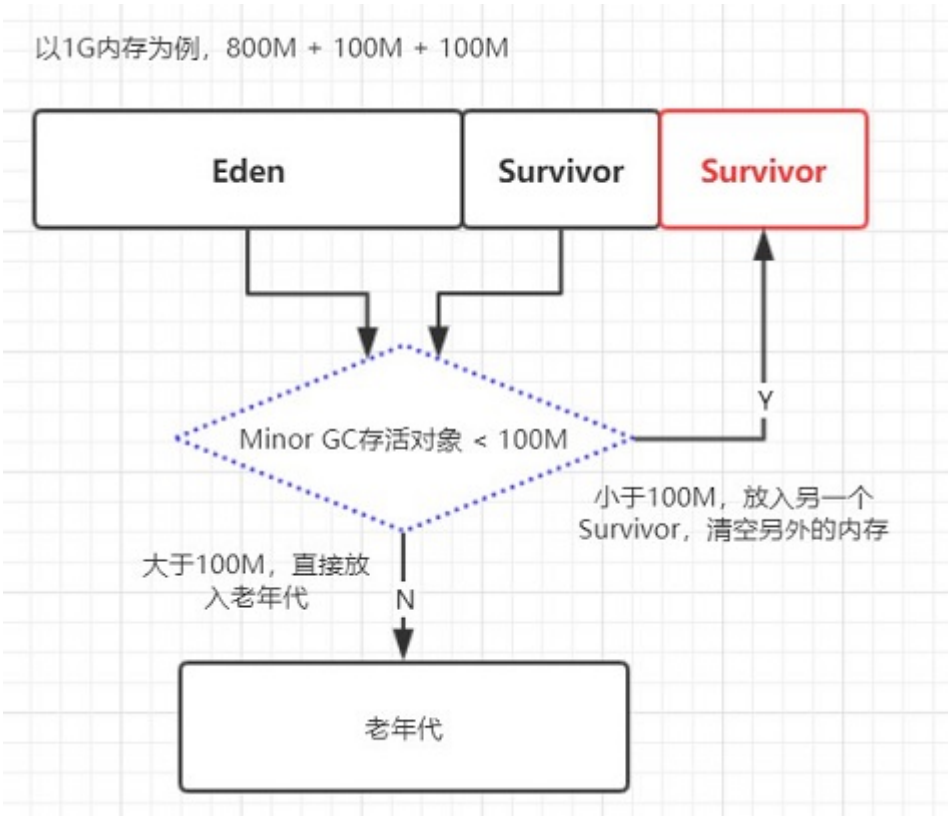


2.3 Minor GC

JVM的新生代采用的垃圾回收算法就是复制算法。

但是划分成了Eden：Survivor1：Survivor2 = 8:1:1

> 当Eden内存不足，触发Minor gc。
>
> 根据GC Roots进行可达性分析，将Eden +一个Survivor(他存放的是上一次Minor gc存活的对象)需要存活的对象放入另一个Survivor中。然后将Eden和原来的Survivor清空。
>
> 这样新生代90%的内存都被利用了。而且新生代的对象每次Minor GC存活的占比约为1%。
>
> 但是：如果Minor GC后存活的对象大于100M，一个Survivor放不下，



2.4 对象如何进入老年代

- 15次GC后，依旧存活的对象，会直接进入Old
- 动态对象年龄判断

  > 当前Survivor对象，按照年龄从小到大排序。如果年龄1 + 年龄2 + ... + 年龄n的总大小超过当前Survivor的50%，那么年龄大于n的对象都会移到Old中。

- 大对象直接进入老年代

  > 因为一个大对象，在被Minor GC的时候，会被反复地在Eden -> Survivor <-> Survivor之间移动。这种copy很耗费时间。

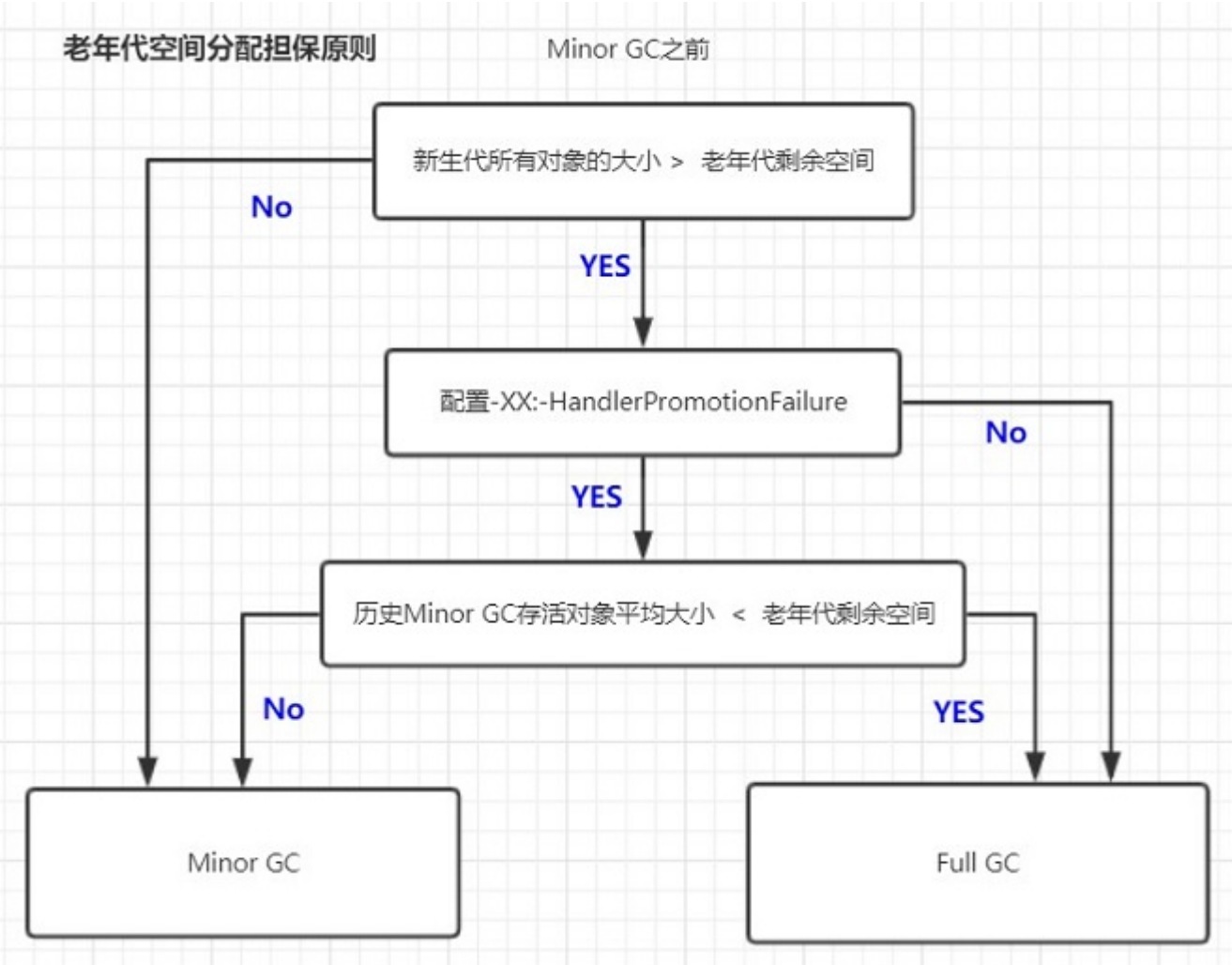
2.5 Full GC

触发Full GC的情况有两种：

第一种：Minor GC后，存活对象太多，老年代放不下。触发Full GC

第二种：Minor GC前会对老年代的空间大小做检查（老年代空间分配担保原则）。

发现空间不足，会触发一次Full GC + Minor GC



3、CMS垃圾收集器

Concurrent Mark Sweep。在老年代的整个过程分4部分

```
1.初始标记（stop the world）
仅仅标记GC Roots能直接关联到的对象。速度很快

2.并发标记
与用户线程并发执行，标记所有可达对象

3.重新标记（stop the world）
修正并发标记阶段因用户程序继续运行而导致标记发生变动的那一部分标记记录。

4.并发清除
```

CMS的缺点：

- CPU资源敏感：在并发标记和并发清除阶段，虽然不会导致线程停顿，但会占用部分CPU，导致程序缓慢，吞吐量降低

- 无法处理浮动垃圾：在并发清除阶段仍然会产生垃圾，无法被此次收集。

  预留一部分内存给用户程序用，一旦内存不够，发生Concurrent Mode Failure，JVM采用备用方案Serial Old对老年代垃圾收集

- 内存碎片问题：由于采用标记清除算法，没有整理

  -XX:CMSFullGCsBeforeCompation设置多少次不带压缩的FGC后，执行一次带压缩的FGC。(默认0,每次FGC都进行碎片整理)

4、G1(JDK11默认)

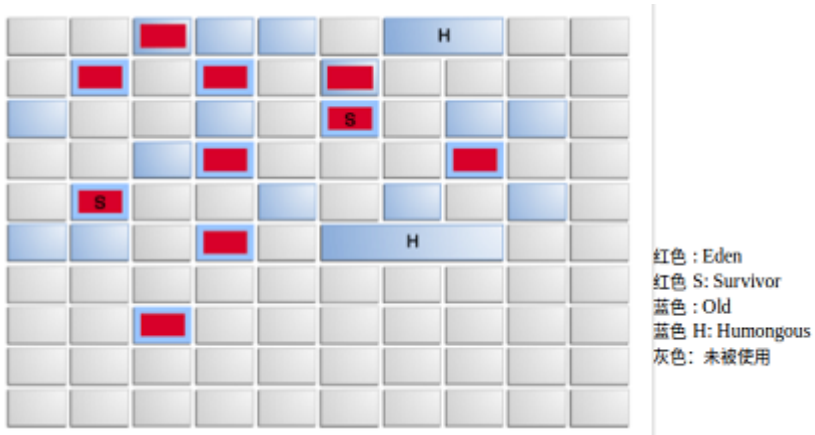
特点:

- 采用Region对堆进行划分
- 根据MaxGCPauseMillis，GC会优先对收益高的Region进行GC
- 对象的分配增加了Humongous区域

https://blog.csdn.net/goldenfish1919/article/details/82911948

G1垃圾回收器是主要针对**多处理器**以及**大内存**的机器，以极高的概率满足预测GC停顿时间要求的同时，还具备高吞吐量性能特征。是基于标记整理的垃圾回收器。

堆内存被划分为多个大小相等的逻辑heap 区，其中一部分区域被当成老一代收集器相同的角色(eden, survivor, old), 但每个角色的区域个数都不是固定的。



每个堆Region 的大小在JVM启动时就确定了，JVM通常生成2000个Region，每个Region 大小在1M-32M之间。这些Region会被逻辑映射成Eden, Survivor, 和 old generation(老年代)空间。

4.1 对象分配策略

它分为3个阶段：

1. TLAB(Thread Local Allocation Buffer)线程本地分配缓冲区
2. Eden区中分配
3. Humongous区分配

TLAB为线程本地分配缓冲区，它的目的为了使对象尽可能快的分配出来。如果对象在一个共享的空间中分配，我们需要采用一些同步机制来管理这些空间内的空闲空间指针。在Eden空间中，每一个线程都有一个固定的分区用于分配对象，即一个TLAB。分配对象时，线程之间不再需要进行任何的同步。

对TLAB空间中无法分配的对象，JVM会尝试在Eden空间中进行分配。如果Eden空间无法容纳该对象，就只能在老年代中进行分配空间。

G1提供了两种GC模式:Young GC和Mixed GC，两种都是Stop The World(STW)

4.2 MaxGCPauseMillis调优

```
-XX:+UseG1GC -Xmx32g -XX:MaxGCPauseMillis=200(默认200ms)
```

这个MaxGCPauseMillis参数就是允许的GC最大的暂停时间。G1尽量确保每次GC暂停的时间都在设置的MaxGCPauseMillis范围内。 那G1是如何做到最大暂停时间的呢？这涉及到另一个概念，CSet(collection set)。它的意思是在一次垃圾收集器中被收集的区域集合。

- Young GC：选定所有新生代里的region。通过控制新生代的region个数来控制young GC的开销。
- Mixed GC：选定所有新生代里的region，外加根据global concurrent marking统计得出收集收益高的若干老年代region。在用户指定的开销目标范围内尽可能选择收益高的老年代region。

在吞吐量跟MaxGCPauseMillis之间做一个平衡。

如果MaxGCPauseMillis设置的过小，那么GC就会频繁，吞吐量就会下降。

如果MaxGCPauseMillis设置的过大，应用程序暂停时间就会变长。

4.3 springboot启动设置GC参数

程序启动的start.sh

```
# !/bin/sh

script=$0
if [[ ${script:0:1} == "/" ]]; then
    bin=`dirname $script`
else
    bin=`pwd`/`dirname $script`
fi

root=${bin}/..
app=digital-human-auth-service

APPLICATION=${bin}/${app}.jar
SPRING_CONFIG_FILE=${root}/conf/application.yaml
MAX_MEMORY=1024M
INFO_LOG_FILE_PATH=${root}/log/info/${app}.info.log
ERROR_LOG_FILE_PATH=${root}/log/error/${app}.error.log
WARN_LOG_FILE_PATH=${root}/log/warn/${app}.warn.log
DEBUG_LOG_FILE_PATH=${root}/log/debug/${app}.debug.log
ACCESS_LOG_FILE_PATH=${root}/log/access/${app}.access.log
ACCESS_DEBUG_LOG_FILE_PATH=${root}/log/access_debug/${app}.access_debug.log
LOGBACK_FILE_PATH=${root}/conf/logback.xml
LOGBACK_ACCESS_FILE_PATH=${root}/conf/logback-access.xml


AGGRESSIVE_GC=false
USE_OLD=false
#"CMS" or "G1"
COLLECTOR=G1"
MIN_MEMORY=384M
MAX_MEMORY=384M
NEW_MEMORY=384M

if[[ $USEOLD == true ]]; then
    TENURE_OPTIONS="-XX: MaxTenuringThreshold-15"
else
    TENURE_OPTIONS="-XX: +NeverTenure"
fi

 MEMORY_SIZE_OPTIONS="-XMS$MIN_MEMORY
                      -XmX$MAX_MEMORY
                      -Xmn$NEW_MEMORY

 MEMORY_THRESHOLD_OPTIONS="-XX:MinHeapFreeRatio=10
                           -XX:MaxHeapFreeRatio=20""

 CMS_OPTIONS="-XX:+UseConcMarkSweepGC
        -XX:InitiatingHeapOccupancyPercent=15"

 G1_OPTIONS=-XX:+UseG1GC
        -XX:-G1UseAdaptiveIHOP
        -XX:InitiatingHeapOccupancyPercent=15
        -XX:+UnlockExperimentalVMOptions
```

```
        -XX:G1MixedGCLiveThresholdPercent=95"

if [[ $COLLECTOR == "CMS" ]]; then
   COLLECTOR_OPTIONS=$CMS_OPTIONS
else
   COLLECTOR_OPTIONS=$G1_OPTIONS
fi

if [[ $AGGRESSIVE_GC == true ]];then
   VM_OPTIONS="$COLLECTOR_OPTIONS
               $TENURE_OPTIONS
               $MEMORY_SIZE_OPTIONS
               $MEMORY_THRESHOLD_OPTIONS"
else
   VM_OPTIONS="-Xmx1024M"
fi

echo "JVM options: $VM_OPTIONS"


java -Dspring.config.location=${SPRING_CONFIG_FILE} -Dfile.encoding=UTF-8 \
    -Dlogging.config=$LOGBACK_FILE_PATH -Dlogging.access.config=$LOGBACK_ACCESS_FILE_PATH\
    -Dlogging.info_log_file_path=$INFO_LOG_FILE_PATH -
Dlogging.error_log_file_path=$ERROR_LOG_FILE_PATH \
    -Dlogging.warn_log_file_path=$WARN_LOG_FILE_PATH -
Dlogging.debug_log_file_path=$DEBUG_LOG_FILE_PATH \
    -Dlogging.access_log_file_path=$ACCESS_LOG_FILE_PATH -
Dlogging.access_debug_log_file_path=$ACCESS_DEBUG_LOG_FILE_PATH \
    -Dlogging.info_log_max_history_in_hours=168 -Dlogging.error_log_max_history_in_days=30 \
    -Dlogging.warn_log_max_history_in_days=30 -Dlogging.debug_log_max_history_in_days=7 \
     ${VM_OPTIONS}-jar ${APPLICATION}  > /dev/null 2>&1 &

tail -F ${root}/log/info/${app}.info.log
```

5、哪些对象应该被回收

1.引用计数法

缺点：循环引用

> 给每个对象添加一个引用计数器。
> 但是难以解决对象的循环引用问题

2.可达性分析（JVM采用）

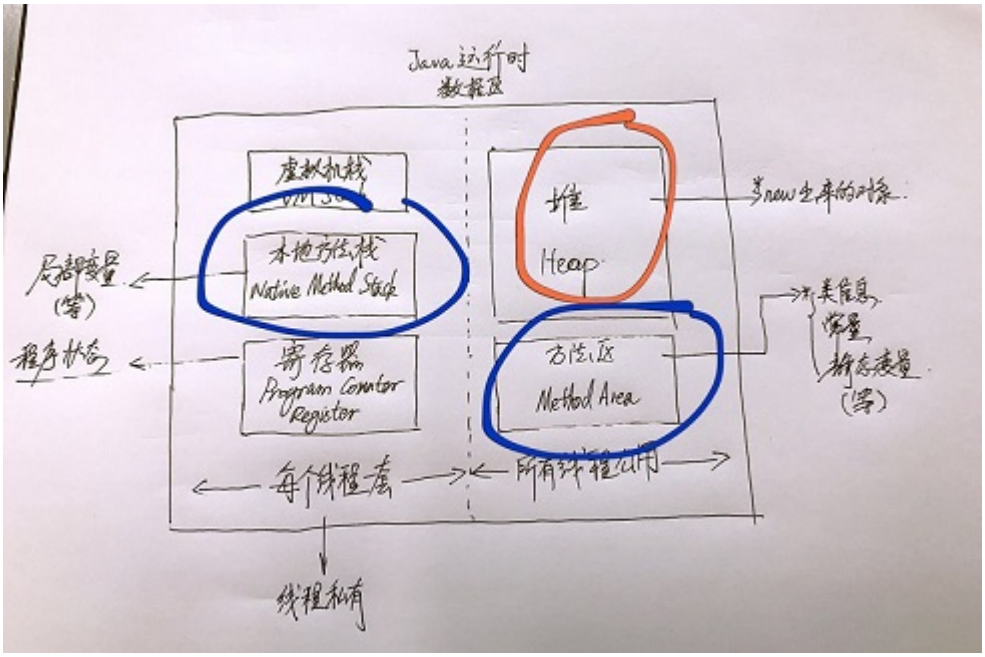
该算法有一个GC Roots对象集合，仍然存活的对象会以一个GC Roots对象作为引用链的起始节点，而没有与GC Roots对象建立引用关系则判断为可回收对象

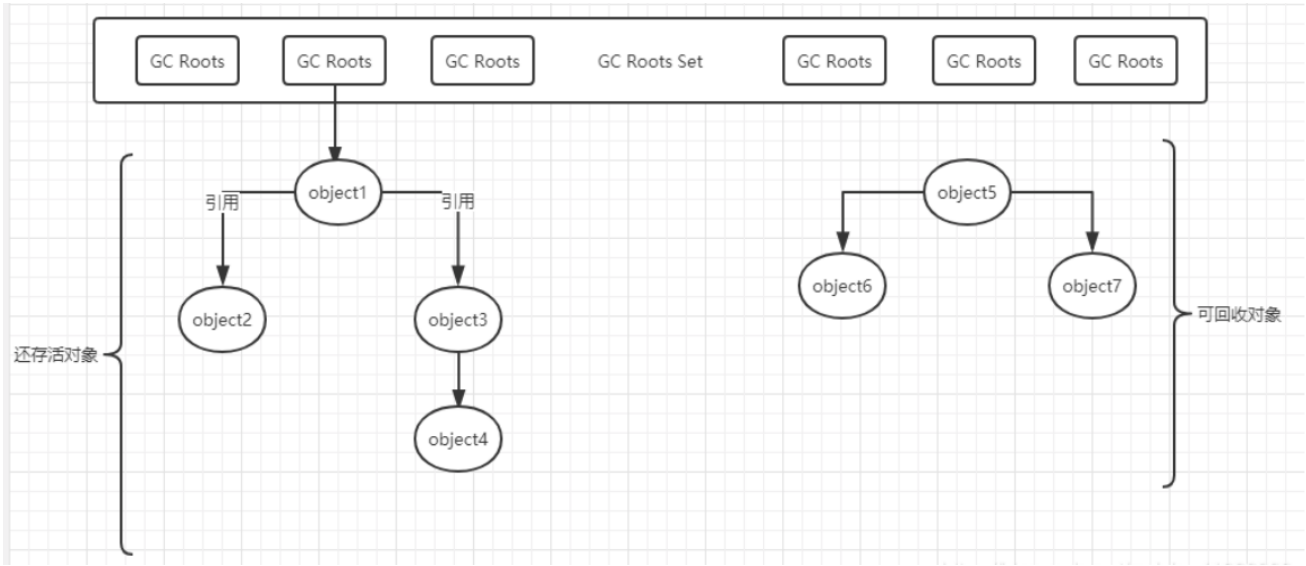
在Java语言中，可作为GC Roots的对象包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 方法区中类静态属性引用的对象。
- 方法区中常量引用的对象。

- 本地方法栈中JNI（即一般说的Native方法）引用的对象。

> 总结就是：局部变量、常量、静态属性都可以作为GC Roots的对象。





- 强引用： 即对象实例化（Object o = new Object()）只要强引用存在，垃圾收集器不会回收被引用对象。
- 软引用： `SoftReference`，一些还有用但是暂时没用到又不是必须的对象，在系统将要发生内存溢出前，这些对象会放进回收名单进行第二次回收，如果回收了这部分内存还是内存不足，则抛出异常。
- 弱引用： `WeakReference`，GC时无论当前内存是否足够，都会回收被弱引用的对象。`WeakHashMap`和`ThreadLocal`类中都用了弱引用。
- 虚引用： 设置虚引用的唯一目的就是能在这个对象被收集前收到一个系统通知。

6、哪些类应该被回收

加载到metaspace里面的类，何时被回收？满足以下几个条件类就可以被回收

- 该类的所有对象实例，在堆内存中都被回收了
- 加载这个类的类加载器ClassLoader被回收了。
- 对该类的class对象没有任何引用

7、JVM优化

```
JVM优化的核心：

第一：尽可能让对象在新生代分配合回收，尽量别让太多对象频繁进入老年代，避免频繁对Old年代进行Full GC。
第二：同时给系统充足的内存，避免新生代频繁进行Minor GC
```

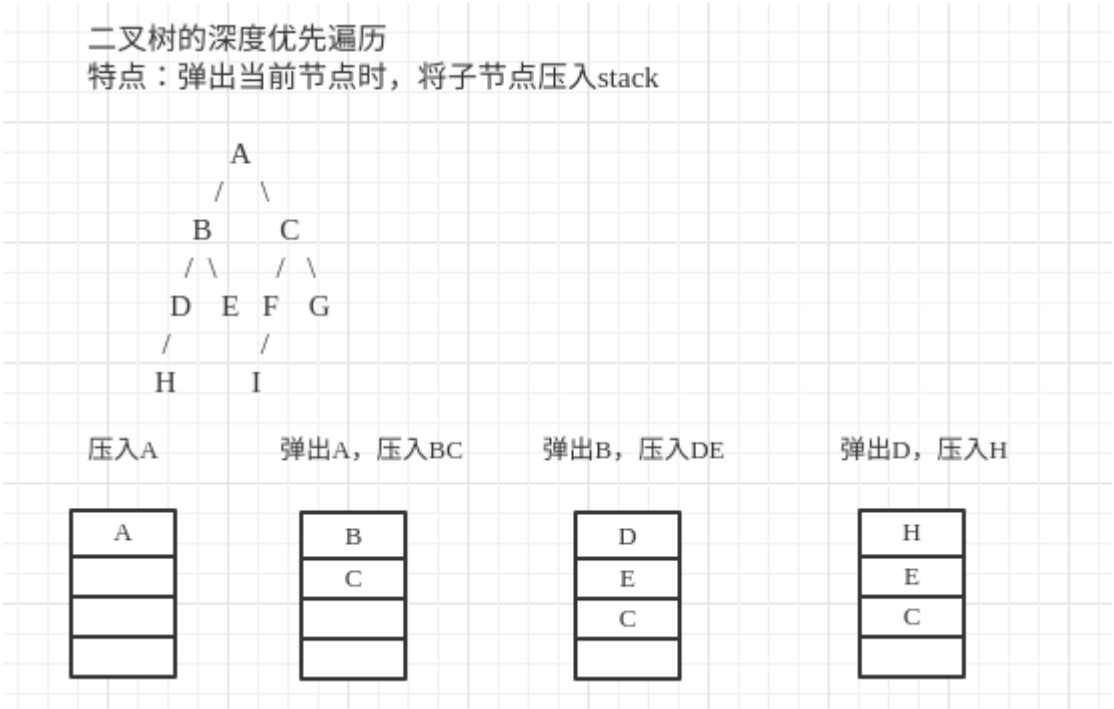
三、树

1.二叉树

不平衡，如果根节点选取不合适，会导致一侧节点特别多，一侧很少

二叉树的遍历

- 深度优先：利用stack的先进后出，从顶部弹出一个节点时，将其左右子节点压入stack顶部
- 广度优先：利用Queue先进先出，从头部拿出一个节点时，将其左右子节点加入queue尾部



```java
import java.util.LinkedList;
import java.util.Stack;

/**
 *                   A
 *                 /   \
 *               B       C
 *             /  \     /  \
 *           D     E   F    G
 *          /         /
 *         H         I
 */
public class Test1 {
    public static void main(String[] args) {
        MyTreeNode head = new MyTreeNode("A");

        head.setLeft(new MyTreeNode("B"));
        head.setRight(new MyTreeNode("C"));

        head.getLeft().setLeft(new MyTreeNode("D"));
        head.getLeft().setRight(new MyTreeNode("E"));
        head.getRight().setLeft(new MyTreeNode("F"));
        head.getRight().setRight(new MyTreeNode("G"));

        head.getLeft().getLeft().setLeft(new MyTreeNode("H"));
        head.getRight().getLeft().setLeft(new MyTreeNode("I"));

        breadthFirstSearch(head);
        System.out.println("\n---------------------");
        depthFirstSearch(head);
        System.out.println("\n---------------------");
        depthFirstSearchRecursive(head);
    }

    /**
     * 广度优先遍历二叉树 (利用Queue先进先出)
     * 从头部拿出一个MyTreeNode时，将其左右子节点加入queue尾部
     *
     * @param head
     */
    public static void breadthFirstSearch(MyTreeNode head) {
        if (head == null)
            return;
        LinkedList<MyTreeNode> queue = new LinkedList<>();
        queue.offer(head);
        while (!queue.isEmpty()) {
            MyTreeNode myTreeNode = queue.poll();
            if (myTreeNode.getLeft() != null)
                queue.offer(myTreeNode.getLeft());
            if (myTreeNode.getRight() != null)
                queue.offer(myTreeNode.getRight());
            System.out.print(myTreeNode.getData() + " ");

        }
```

```
    }

    /**
     * 深度优先遍历二叉树 （利用stack先进后出）
     * 从stack弹出一个MyTreeNode时，将其左右子节点压入stack中
     * （如先遍历左节点，压栈时就先右节点，再左节点
     *
     * @param head
     */
    public static void depthFirstSearch(MyTreeNode head) {
        if (head == null)
            return;
        Stack<MyTreeNode> stack = new Stack<>();
        stack.push(head);
        while (!stack.isEmpty()) {
            MyTreeNode treeNode = stack.pop();
            if (treeNode.getRight() != null)
                stack.push(treeNode.getRight());
            if (treeNode.getLeft() != null)
                stack.push(treeNode.getLeft());
            System.out.print(treeNode.getData() + " ");
        }
    }

    /**
     * 深度优先遍历二叉树 （递归实现）:其实就是先序遍历
     * @param head
     */
    public static void depthFirstSearchRecursive(MyTreeNode head) {
        if (head != null) {
            System.out.print(head.getData() + " ");
            depthFirstSearchRecursive(head.getLeft());
            depthFirstSearchRecursive(head.getRight());
        }
    }
    /**
     * 先序遍历二叉树
     * @param head
     */
    public static void perOrder(MyTreeNode node) {

        if (node == null)
            return;
        System.out.print(node.getData() + " ");
        perOrder(node.getLeft());
        perOrder(node.getRight());
    }
    /**
     * 中序遍历二叉树
     * @param head
     */
    public static void midOrder(MyTreeNode head) {

        if (head == null)
```

```
            return;
        midOrder(head.getLeft());
        System.out.print(head.getData() + " ");
        midOrder(head.getRight());
    }
}
```

2.红黑树

应用:由于树的深度过大而造成磁盘IO读写过于频繁，进而导致效率低下的情况下，采用红黑树

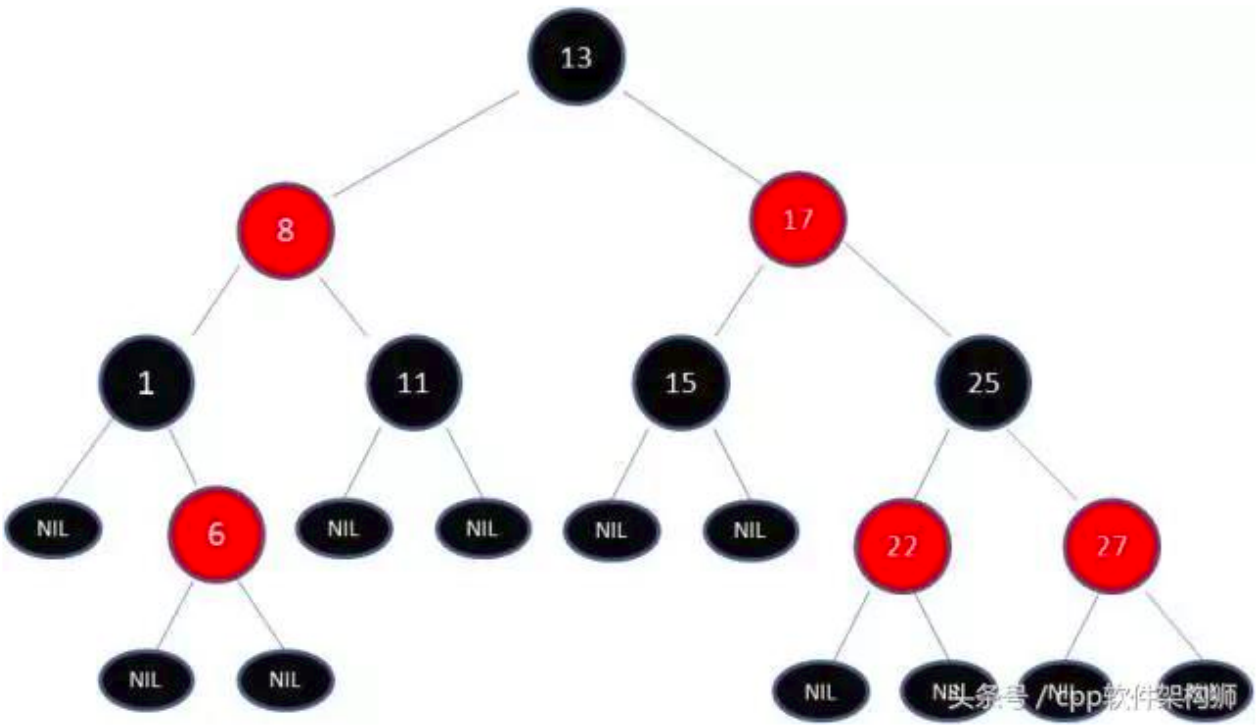
- Linux进程调度
- HashMap的底层。

```
性质：
1.所有的节点都是黑色或者红色
2.根节点是黑色
3.每个叶子节点都是黑色的空节点
4.每个红色节点的两个子节点都是黑色
5.从任意节点到其子节点的所有路径都经过相同的黑色节点
```

从性质5可以看出，红黑树不是完美平衡的二叉树，树的深度不一定是一样的，只是黑色节点的层数是相同的。

红黑树如何实现自平衡？左旋，右旋，变色。

```
左旋只影响旋转结点和其右子树的结构，把右子树的结点往左子树挪了。
右旋只影响旋转结点和其左子树的结构，把左子树的结点往右子树挪了。
所以旋转操作是局部的。
```



查找

查找最坏时间复杂度为O(2lgN)，也即整颗树刚好红黑相隔的时候。能有这么好的查找效率得益于红黑树自平衡的特性

插入

https://www.jianshu.com/p/e136ec79235c

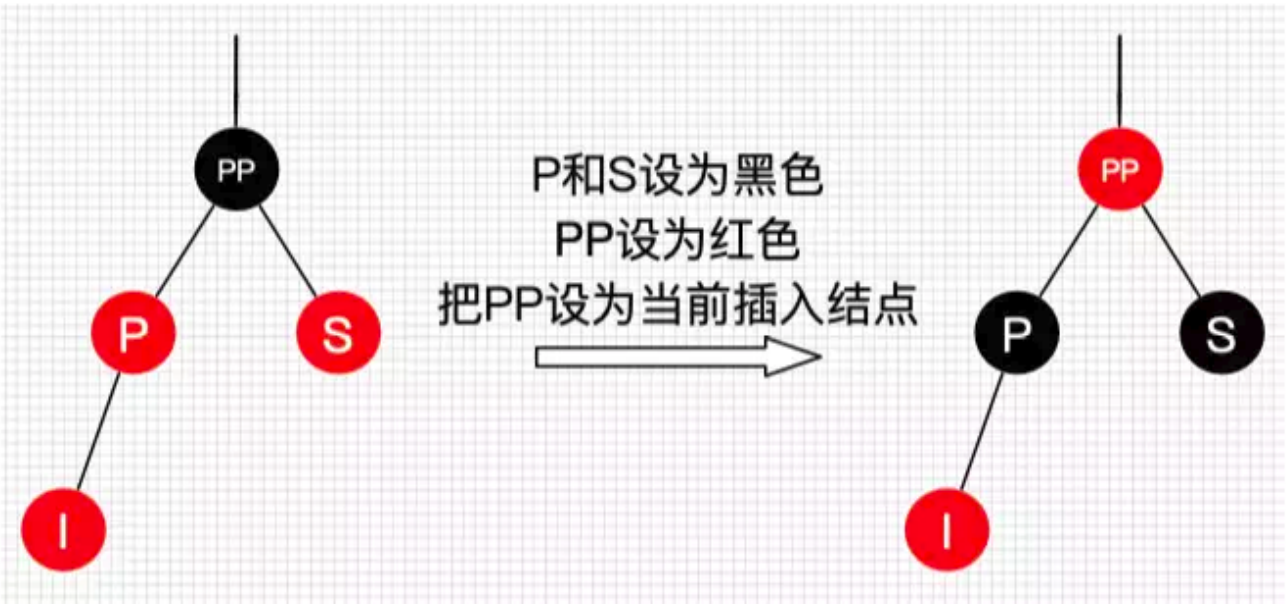
先根据大小确定插入的位置，但插入结点是应该是什么颜色呢？答案是红色。

理由很简单，红色在父结点（如果存在）为黑色结点时，红黑树的黑色平衡没被破坏，不需要做自平衡操作。但如果插入结点是黑色，那么插入位置所在的子树黑色结点总是多1，必须做自平衡。

> 插入情景1：红黑树为空树
> 最简单的一种情景，直接把插入结点作为根结点就行。
> 根据红黑树性质2：根节点是黑色。还需要把插入结点设为黑色。
>
> 插入情景2：插入结点的Key已存在
> 插入结点的Key已存在，既然红黑树总保持平衡，在插入前红黑树已经是平衡的，那么把插入结点设置为将要替代结点的颜色(颜色不变)，再把结点的值更新就完成插入。
>
> 插入情景3：插入结点的父结点为黑结点
> 由于插入的结点是红色的，当插入结点的黑色时，并不会影响红黑树的平衡，直接插入即可，无需做自平衡。
>
> 插入情景4：插入结点的父结点为红结点
> 再次回想下红黑树的性质2：根结点是黑色。如果插入的父结点为红结点，那么该父结点不可能为根结点，所以插入结点总是存在祖父结点。这点很重要，因为后续的旋转操作肯定需要祖父结点的参与。

插入的节点为I，父亲节点为P，叔叔节点为S，祖父节点为PP

P为红，S为红

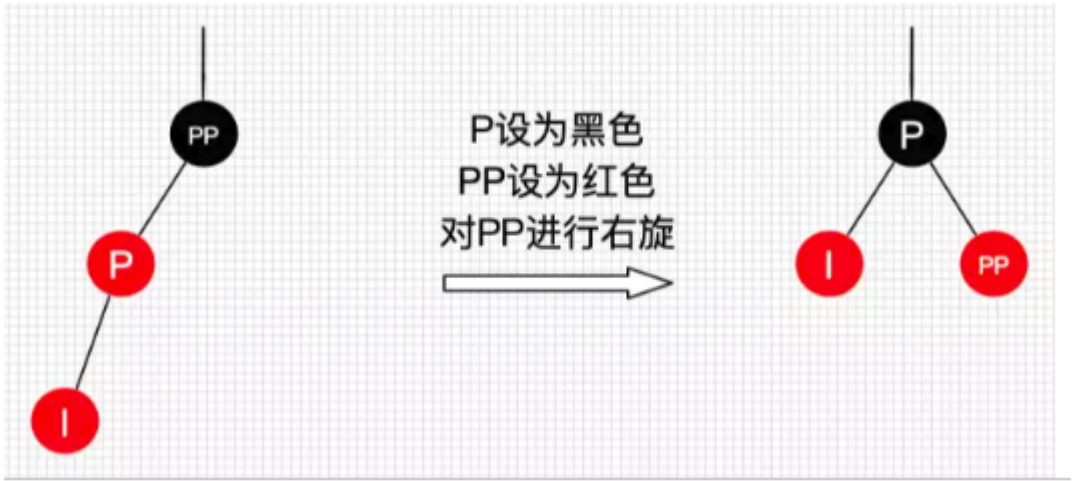


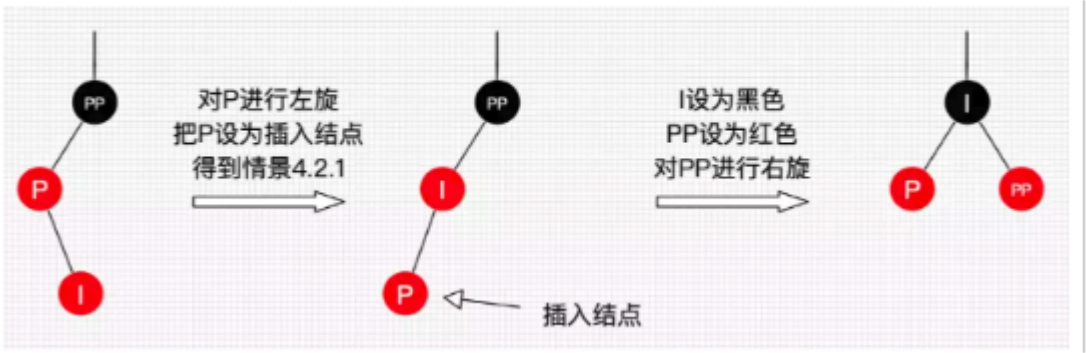
PP为红，并且为当前节点。继续判断直到树平衡。

注意：
如果PP刚好为根结点时，那么根据性质2，我们必须把PP重新设为黑色，那么树的红黑结构变为：黑黑红。换句话说，从根结点到叶子结点的路径中，黑色结点增加了。这也是唯一一种会增加红黑树黑色结点层数的插入情景。

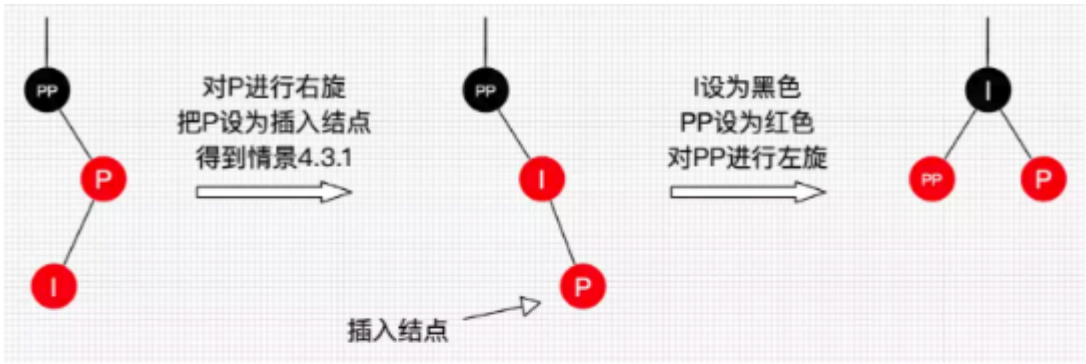
我们还可以总结出另外一个经验：红黑树的生长是自底向上的。这点不同于普通的二叉查找树，普通的二叉查找树的生长是自顶向下的。

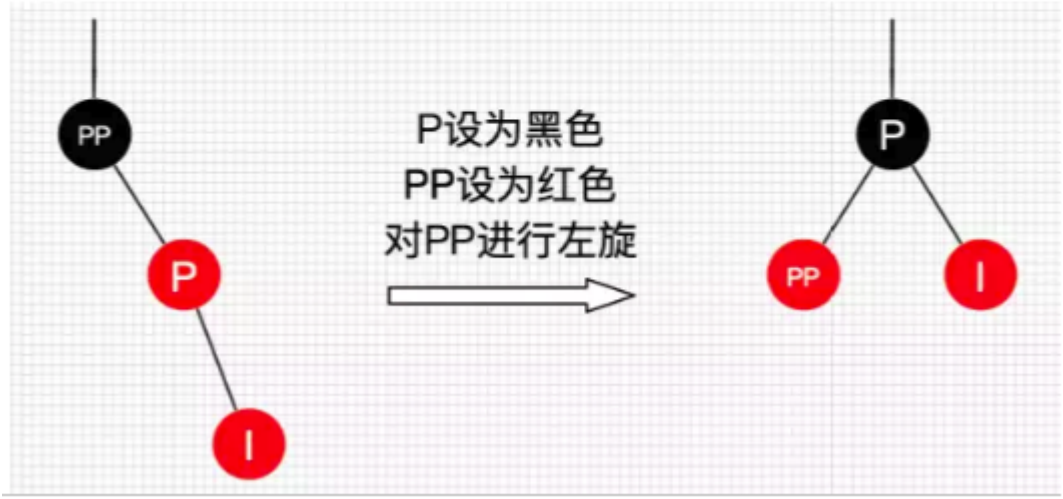
P为左红，S没有





P为右红，S没有





删除

红黑树的删除操作也包括两部分工作：一查找目标结点；而删除后自平衡。查找目标结点显然可以复用查找操作，当不存在目标结点时，忽略本次操作；当存在目标结点时，删除后就得做自平衡处理了。删除了结点后我们还需要找结点来替代删除结点的位置，不然子树跟父辈结点断开了，除非删除结点刚好没子结点，那么就不需要替代。

二叉树删除结点找替代结点有3种情情景：

- 情景1：若删除结点无子结点，直接删除
- 情景2：若删除结点只有一个子结点，用子结点替换删除结点
- 情景3：若删除结点有两个子结点，用后继结点（大于删除结点的最小结点）替换删除结点

将所有节点往X轴投影，删除节点的后面第一个节点为后继节点。

删除结点被替代后，在不考虑结点的键值的情况下，对于树来说，可以认为删除的是替代结点（后继节点）！

3.B树

应用：B树大量应用在数据库和文件系统当中。MongoDB中

它的设计思想是，将相关数据尽量集中在一起，以便一次读取多个数据，减少硬盘操作次数。B树算法减少定位记录时所经历的中间过程，从而加快存取速度。

4.B+树

5.B树和B+树的应用

B-树和B+树最重要的一个区别：**B+树只有叶节点存放数据，其余节点用来索引，而B-树是每个索引节点都会有Data域**。这就决定了B+树更适合用来存储外部数据，也就是所谓的磁盘数据。

> B类树的特定就是每层节点数目非常多，层数很少，目的就是为了就少磁盘IO次数，当查询数据的时候，最好的情况就是很快找到目标索引，然后读取数据。

从Mysql（Inoodb）的角度来看：

B+树是用来充当索引的，一般来说索引非常大，尤其是关系性数据库这种数据量大的索引能达到亿级别，所以为了减少内存的占用，索引也会被存储在磁盘上。

那么Mysql如何衡量查询效率呢？

- 优点一：磁盘IO次数，B+树除了叶子节点其它节点并不存储数据，节点小，磁盘IO次数就少(相对二叉树，深度小)。
- 优点二：如果非叶子节点保存数据，这样导致在非叶子节点中能保存的指针数量变少，指针少的情况下要保存大量数据，只能增加树的高度，导致IO操作变多，查询性能变低。
- 优点三：B+树所有的Data域在叶子节点，一般来说都会进行一个优化，就是将所有的叶子节点用指针串起来。这样遍历叶子节点就能获得全部数据，这样就能进行**区间访问**啦。（相对于B树，可以区间查询，因为是关系型数据库）

MongoDB为什么使用B-树而不是B+树？

- 它并不是传统的关系性数据库，而是以Json格式作为存储的nosql，目的就是高性能，高可用，易扩展。
- MongoDB使用B-树，所有节点都有Data域，只要找到指定索引就可以进行访问，无疑单次查询平均快于Mysql（但侧面来看Mysql至少平均查询耗时差不多）。

总体来说，Mysql选用B+树和MongoDB选用B-树还是以自己的需求来选择的。

四、并发

五、设计模式

设计模式的分类

总体来说设计模式分为三大类：

创建型模式，共五种：`工厂方法模式`、抽象工厂模式、`单例模式`、建造者模式、原型模式。

结构型模式，共七种：`适配器模式`、`装饰器模式`、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：`策略模式`、模板方法模式、`观察者模式`、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

哪些设计模式体现了开闭原则？

开闭原则：对扩展开放，对修改关闭

1.抽象工厂模式和代理模式

1.1.工厂模式

```
public class BeanFactory {
    // 普通工程模式
    public Animal getAnimal(String type) {
        switch (type.toUpperCase()) {
            case "DOG":
                return new Dog();
            case "CAT":
                return new Cat();
            default:
                return new Animal();
        }
    }
    // 普通工程模式改进版，防止type输入错误，导致无法创建实际类
    public Animal getDog() {
        return new Dog();
    }

    public Animal getCat() {
        return new Cat();
    }

    // 静态工厂模式。不需要实例化工厂类(最常用)
    public static Animal getStaticDog() {
        return new Dog();
    }

    public static Animal getStaticCat() {
        return new Cat();
    }
}
```

1.1抽象工厂模式

工厂模式的缺点在于一旦有新的类加入，就需要修改工厂类。不太符合开闭原则。

解决方法：将工厂类抽象成一个interface，一旦有新的类加入，只需要增加interface的实现类（创建新的工厂类），这样就不用修改原来的代码。

2.单例模式

写出三种实现单例模式的方法

第一种：synchronized+锁

```
public class Singleton{

  private static Singleton instance = null;

  private Singleton(){};

  public static Singleton getInstance(){
    if (instance == null ){
      synchronized(Singleton.class){
        if (instance == null){
          instance = new Singleton();
          // java指令中对象的创建和赋值是分开的（并不保证先后顺序）。
          // 所以 instance = new Singleton()分两步执行。
          // 可能jvm先为Singleton分配空间并直接赋值给instance，然后在初始化Singleton这个实例。
          // 此时，线程B进来，instance不为null，但是又没有初始化Singleton实例    ----> 报错
        }
      }
    }
    return instance;
  }
}
```

第二种：内部类

```
public class Singleton2 {
    private Singleton2() { }
    private static class SingletonFactory {
        private static Singleton2 singleton2 = new Singleton2();
    }

    public static Singleton2 getInstance() {
        return SingletonFactory.singleton2;
    }
}
```

第三种：枚举

```
public class SingletonDemo {

    private SingletonDemo() { }

    private enum EnumHolder {
        INSTANCE;
        private SingletonDemo instance;
        // 枚举的构造方法
        EnumHolder() {
            instance = new SingletonDemo();
        }
        private SingletonDemo getInstance() {
            return instance;
        }
    }

    public static SingletonDemo getInstance() {
        return EnumHolder.INSTANCE.getInstance();
    }
}
```

3.代理模式

特点：

- 代理类和委托类实现相同的接口
- 委托类实例化对象作为代理类的成员变量

为什么要用代理模式？

- 中介隔离作用：在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是`代理类和委托类实现相同的接口`。
- 开闭原则，增加功能：代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。

代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。

代理类本身并不真正实现服务，而是同过调用委托类的相关方法，来提供特定的服务。

真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要打开已经封装好的委托类。

静态代理：就是程序运行前已经编写好代理类，编译成.class文件了

动态代理：在程序运行过程中，通过反射机制动态创建

3.1 静态代理

优点：可以做到在符合开闭原则的情况下对目标对象进行功能扩展。

缺点：我们得为每一个服务都得创建代理类，工作量太大，不易管理。同时接口一旦发生改变，代理类也得相应修改。

```
public interface BuyHouse {
    void buyHouse();
}
```

```
public class BuyHouseImpl implements BuyHouse {
    @Override
    public void buyHouse() {
        System.out.println("i wanna buy house");
    }
}
```

```
public class BuyHouseProxy implements BuyHouse {
    // 委托类实例化对象作为代理类的成员变量
    private BuyHouse buyHouse;

    public BuyHouseProxy(){
        this.buyHouse = new BuyHouseImpl();
    }
    public BuyHouseProxy(BuyHouse buyHouse){
        this.buyHouse = buyHouse;
    }

    @Override
    public void buyHouse() {
        System.out.println("before");
        buyHouse.buyHouse();
        System.out.println("after");
    }
}
```

```
public class BuyHouseTest {
    public static void main(String[] args) {
        // 上层只接触到代理类，而不直接获取被代理类
        BuyHouse proxy = new BuyHouseProxy();
        proxy.buyHouse();
    }
}
```

3.2 JDK动态代理

JDK动态代理：基于接口

动态代理中只需要编写一个动态处理器就可以了，不再需要再手动的创建代理类。真正的代理对象由JDK再运行时为我们动态的来创建。

```
public class DynamicProxyHandler implements InvocationHandler {

    private Object object;

    public DynamicProxyHandler(final Object object) {
        this.object = object;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("买房前准备");
        Object result = method.invoke(object, args);
        System.out.println("买房后装修");
        return result;
    }
}
```

```
public class DynamicProxyTest {
    public static void main(String[] args) {
        BuyHouse buyHouse = new BuyHouseImpl();
        BuyHouse proxyBuyHouse = (BuyHouse) Proxy.newProxyInstance(
                                                BuyHouse.class.getClassLoader(),
                                                new Class[]{BuyHouse.class},
                                                new DynamicProxyHandler(buyHouse));
        proxyBuyHouse.buyHosue();
    }
}
```

注意*Proxy.newProxyInstance()*方法接受三个参数：

- *ClassLoader loader:*指定当前目标对象使用的类加载器,获取加载器的方法是固定的
- *Class<?>[] interfaces:*指定目标对象实现的接口的类型,使用泛型方式确认类型
- *InvocationHandler:* `指定动态处理器，` 执行目标对象的方法时,会触发事件处理器的方法

4.CGLIB动态代理

基于类的继承。

JDK实现动态代理需要实现类通过接口定义业务方法。

对于没有接口的类，如何实现动态代理呢，这就需要CGLib了。CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。

因为采用的是继承，所以不能对final修饰的类进行代理。

JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

六、JVM内存模型

1.类加载过程

2.和类加载器

jvm的内存分配

new一个对象时发生具体步骤

https://www.cnblogs.com/JackPn/p/9386182.html

java在new一个对象的时候，会先查看对象所属的类有没有被加载到内存，如果没有的话，就会先通过类的全限定名来加载。加载并初始化类完成后，再进行对象的创建工作。

我们先假设是第一次使用该类，这样的话new一个对象就可以分为两个过程：**加载并初始化类**和**创建对象**。

第一步：类加载

> java是使用双亲委派模型来进行类的加载的

1、加载

由类加载器负责根据一个类的全限定名来读取此类的二进制字节流到JVM内部，并存储在运行时内存区的方法区，然后将其转换为一个与目标类型对应的java.lang.Class对象实例

2、验证

格式验证：验证是否符合class文件规范

3、准备

为类中的所有静态变量分配内存空间，并为其设置一个初始值
被final修饰的static变量（常量），会直接赋值。

4、解析

将常量池中的符号引用转为直接引用（得到类或者字段、方法在内存中的指针或者偏移量，以便直接调用该方法），这个可以在初始化之后再执行。
解析需要静态绑定的内容。

以上2、3、4三个阶段又合称为链接阶段：
链接阶段要做的是将加载到JVM中的二进制字节流的类数据信息合并到JVM的运行时状态中。

5、初始化（先父后子）

5.1 为静态变量赋值

5.2 执行static代码块

> 注意：static代码块只有jvm能够调用
> 如果是多线程需要同时初始化一个类，仅仅只能允许其中一个线程对其执行初始化操作，其余线程必须等待，只有在活动线程执行完对类的初始化操作之后，才会通知正在等待的其他线程。

因为子类存在对父类的依赖，所以**类的加载顺序是先加载父类后加载子类，初始化也一样。**不过，父类初始化时，子类静态变量的值也有有的，是默认值。

最终，方法区会存储当前类类信息，包括类的**静态变量、类初始化代码（定义静态变量时的赋值语句** 和 **静态初始化代码块）、实例变量定义、实例初始化代码（定义实例变量时的赋值语句实例代码块**和**构造方法）**和**实例方法**，还有**父类的类信息引用。**

第二步、创建对象

1、在堆区分配对象需要的内存

分配的内存包括本类和父类的所有实例变量，但不包括任何静态变量

2、对所有实例变量赋默认值

将方法区内对实例变量的定义拷贝一份到堆区，然后赋默认值

3、执行实例初始化代码

初始化顺序是先初始化父类再初始化子类，初始化时先执行**实例代码块**然后是**构造方法**

4、如果有类似于Child c = new Child()形式的c引用的话，在栈区定义Child类型引用变量c，然后将堆区对象的地址赋值给它

多线程，线程池

spring的IOC、AOP

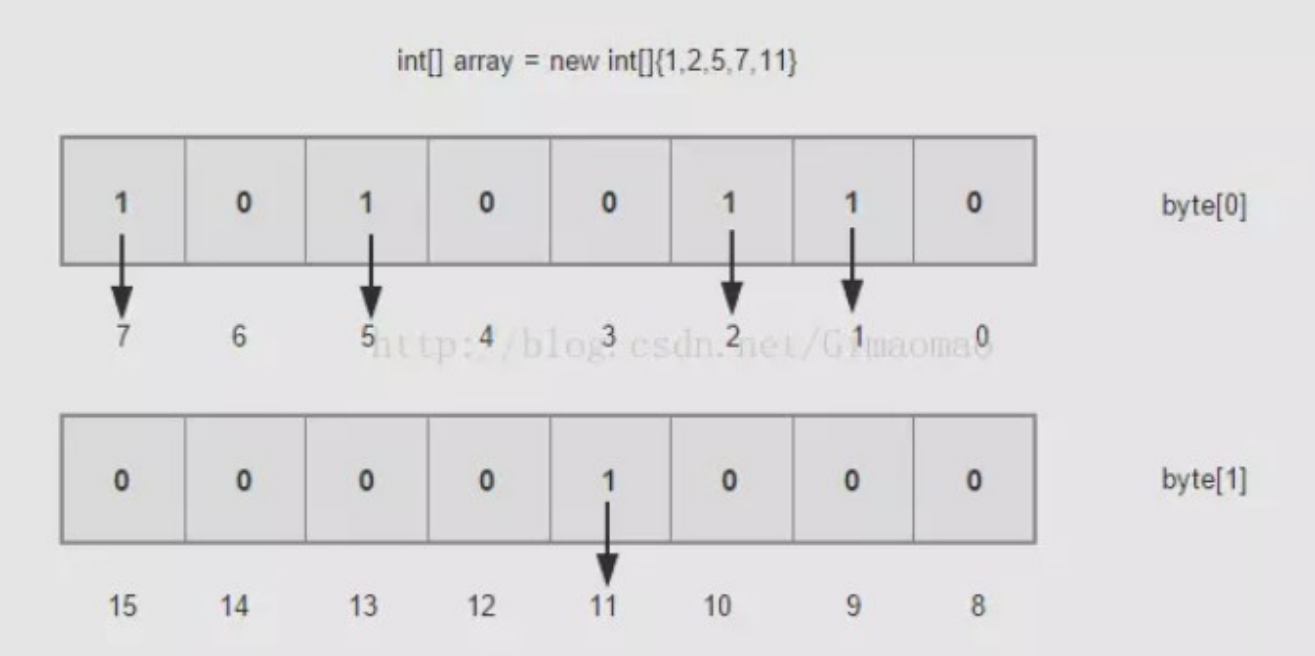
经典题

BitMap

```
针对问题：
        排序、去重
在真正很大数据量的时候，可以先分桶 + BitMap

PS: 2.7亿个整数大约占用1G内存
```

所谓的bitmap：就是用一个bit位来标记某个元素，而数组下标是该元素。

bitmap经常用在大数据的题中，比如10亿个int类型的数，如果用int数组存储的话，那么需要大约4G内存，浪费内存。如果用bitmap解决，就比较方便。bitmap可以用int来模拟，也可以用byte来模拟，它只是逻辑上的概念，在java语言中写不出来，我们采用byte。一个byte占8个bit，如果每一个bit的值是有或者没有，即1或0，则如下图所示：



方法重点：

- 位运算
- 与运算
- 或运算

```
public class BitMapTest {
    private static final int CAPACITY = 1_000_000;// 数据容量

    public static void main(String[] args) {
        testMyFullBitMap();
    }

    public static void testMyFullBitMap() {
        MyFullBitMap ms = new MyFullBitMap();
        long startTime = System.currentTimeMillis();
        // byte[] bytes = mockData(ms);
        byte[] bytes = mockBigData(ms);
        long endTime = System.currentTimeMillis();
        System.out.printf("存入%d个数，用时%dms\n", CAPACITY, endTime - startTime);

        startTime = System.currentTimeMillis();
        ms.output(bytes);
        endTime = System.currentTimeMillis();
        System.out.printf("取出%d个数，用时%dms\n", CAPACITY, endTime - startTime);
    }

    public static byte[] mockData(MyFullBitMap ms) {
        ms.setBit(2); ms.setBit(10);
        ms.setBit(8); ms.setBit(4);
        ms.setBit(14); ms.setBit(11);
        return ms.getDataBytes();
    }

    public static byte[] mockBigData(MyFullBitMap ms) {
        Random random = new Random();
        for (int i = 0; i < CAPACITY; i++) {
            int num = random.nextInt();
            ms.setBit(num);
        }
        return ms.getDataBytes();
    }
}


/**
 * 整数              bit数组                 byte数组
 * -2^31     -->     bit[0]          -->     byte[0]
 *   0       -->     bit[2^31]       -->     byte[2^28]
 *  2^31     -->     bit[2^32]       -->     byte[2^29]
 *
 * int范围:  [-2^31, 2^31-1]
 * byte范围:[-128, 127] [-2^8, 2^8-1]
 */
class MyFullBitMap {
    // 定义一个byte数组表示所有的int数据，1 bit对应一个，共2^32b = 2^29B = 512MB
    private byte[] dataBytes = new byte[1 << 29];


    public byte[] getDataBytes() {
```

```
        return dataBytes;
    }

    /**
     * 读取数据，并将对应数数据的 到对应的bit中，并返回byte数组。
     * @param num 读取的数据
     * @return byte数组 dataBytes
     */
    public byte[] setBit(int num) {
        // 获取num数据对应bit数组（虚拟）的索引,考虑有负数，0在bit[2^31]处
        long bitIndex = num + (1l << 31);
        // bit数组（虚拟）在byte数组中的索引
        int index = (int) (bitIndex / 8);
        // bitIndex 在byte[]数组索引index 中的具体位置
        int innerIndex = (int) (bitIndex % 8);

        System.out.println("byte[" + index + "] 中的索引： " + innerIndex);
        // 找出重复元素
        if (b == 1 << innerIndex) {
            System.out.println(num + "是重复元素");
        }

        // 采用或运算，只要有“1”，结果就是“1”。
        // 注意由于(1 << innerIndex)可知bit数组不是严格按照顺序排列的，在byte数组内部，1还是在最右侧。
        // 否则就是(1 << (8 - innerIndex))
        dataBytes[index] = (byte) (dataBytes[index] | (1 << innerIndex));
        return dataBytes;
    }

    /**
     * 输出数组中的数据
     * @param bytes byte数组
     */
    public void output(byte[] bytes) {
        int count = 0;
        System.out.println("bytes = " + bytes);
        for (int i = 0; i < bytes.length; i++) {
            for (int j = 0; j < 8; j++) {
                // 采用与运算获取对于bit位上的值，都为“1”，才返回“1”
                if (((bytes[i]) & (1 << j)) != 0) {
                    count++;
                    int number = (int) ((((long) i * 8 + j) - (1l << 31)));
                    System.out.println("取出的第 " + count + "\t个数: " + number);
                }
            }
        }
    }
}
```