1. 标识符由数字、字母、下划线\_、美元符$，不能以数字开头，不能是关键字，区分大小写。

2. byte、short、char在运算的时候，都会提升为int，再进行运算。

3. 常量计算 byte a = 3 + 4; java存在常量优化机制，编译时，会将3和4计算出一个7的结果，并且会自动判断该结果是否在byte取值范围内，在:编译通过，不在:编译失败。

4. 数组定义格式：int[] type 或 int type[]

5. 创建数组动态初始化：int[] arr = new int[3]，定义了三个长度的数组，下标从0开始，不赋值默认0，arr[0]=1进行赋值，int[] arr进栈并保存指向堆的引用，new int[3]进堆，并生成地址值。

静态初始化：int[] arr = new int[]{1, 2, 3} 或 int[] arr = {1, 2, 3}

6. 栈内存：存放局部变量及堆中的地址引用，如果是方法只有在调用时从方法区进入栈内存中，先进后出；

堆内存：new出来的内容都会进入堆内存，并且会存在地址值；

方法区：字节码文件（.class文件->类文件）加载时进入的内存及包含类中的方法；

7. 产生随机数：Random r = new Random(); int i = r.nextInt(5) + 5; 随机产生[0, 5)之间的数据，生成出来的再加5，为最终结果。

8. 方法重载：方法名相同，参数个数/类型/顺序不同，跟返回值无关。

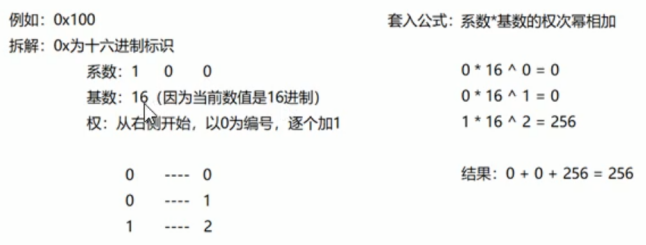
9. 十进制：Java中，数值默认都是10进制，不需要加任何修饰；

二进制：数值前面以0b开头，b大小写都可以；

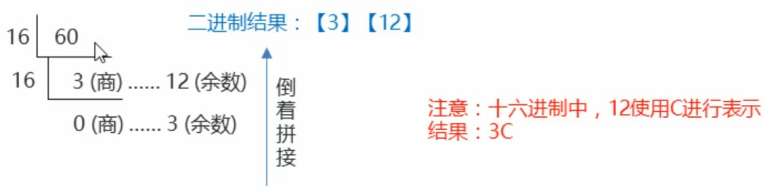
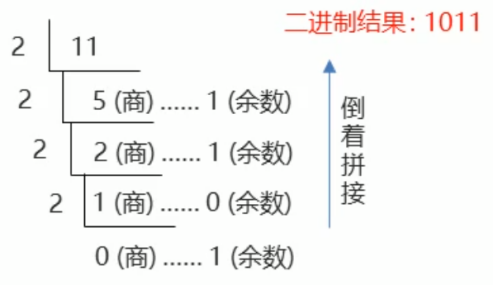
八进制：数值前面以0开头；

十六进制：数值前面以0x开头，x大小写都可以；

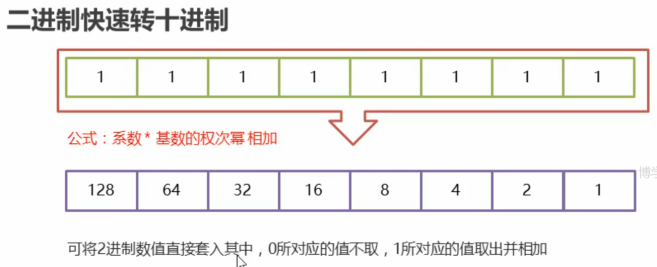
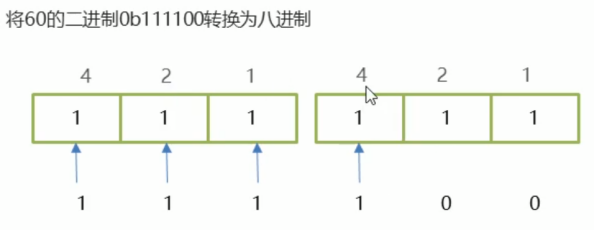
10. 任意进制转换为十进制：

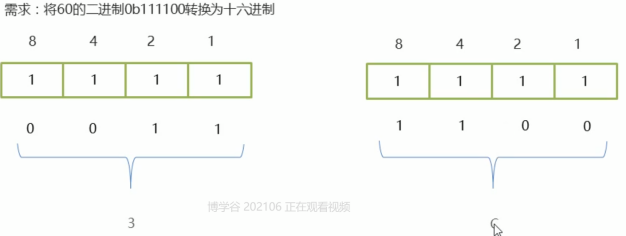


11. 十进制转换为任意进制：



12. 二进制转其他进制：



13. 位移运算：

<<：有符号左移运算，左边符号位丢弃，右边补齐0；

>>：有符号右移运算，根据符号位，补齐左边；

>>>：无符号右移，无论符号位是0还是1，都补0；

14. 异或^：一个数被另一个数异或两次，结果还是本身；如：10^20^20=10;

15. 二维数组：int[][] arr=new int[3][4]; 创建了包含3个一维数组并且数组元素4个的二维数组，赋值：int[0][0]=11，或int[][] arr={{11, 22, 33}, {44, 55, 66}}; 遍历：

**for** (**int** i=0; i<array.**length**; i++) {  
 **for** (**int** j=0; j<array[i].**length**; j++) {  
 System.***out***.println(array[i][j]);  
 }  
}

16. 集合和数组的区别：

①数组的长度不可变，集合的长度可变；

②数组可以存基本数据类型和引用数据类型；集合只能存引用数据类型，如果存基本数据类型，需要存对应的包装类；

17. Collection接口：包含List(有序，可重复)和Set(无序，不可重复)单列结合，继承了Iterator接口，可以使用增强for循环。

18. Iterator迭代器：next()获取当前位置的元素，将迭代器对象移向下一个索引的位置；

19. for使用场景：操作索引使用普通for; 删除元素使用迭代器; 只是遍历使用增强for;

20. 栈：先进后出，队列：先进先出；

21. 数组：通过地址值和索引定位，查询速度快；添加/删除元素时，后面每个元素后移/前移，增删效率低；

链表：每个元素称为结点，结点又是一个独立的对象，结点以存储具体的数据和下一个结点的地址组成，第一个结点称为head头结点(不存储值)，没有数据结点指向空地址，前一个结点记录后一个结点的地址值形成链表。增删快，查询需要head头结点往后查，所以查询慢；

22．ArrayList源码：当new创建时长度为0的数组，只有调用add方法才会创建一个长度为10的数组并且初始化为null，数组名为elementData，底层定义了变量size，功能：要操作的索引和元素的个数，当10元素的数组添加满之后，就需要自动扩容，扩容到1.5倍长度，会将旧数组的元素拷贝到新数组中，然后对新数组进行操作；

1. LinkedList特有方法：

①addFirst()：元素添加到第一行；

②addLast()：元素添加到最后一行；

③getFirst()：获取第一行的元素值；

④getLast()：获取最后一行的元素值；

⑤removeFirst()：删除第一行的元素，并返回元素值；

⑥removeLast()：删除最后一行的元素，并返回元素值；

24. LinkedList源码解析：LinkedList是一个双向链表，由前一个节点地址(Node<E> prev)、本节点应该存储的值(E item)和下一个节点地址(Node<E> next)组成，存在两个指针变量Node<E> first和Node<E> last。

第一次add，first和last指向同一个Node节点地址值，以后的新增，first指向第一个Node节点地址值，last会指向新添加的Node节点地址值，next会指向新增后数据的prev;

get()方法获取数据时，先判断索引和size>>1(size/2)的大小，小于的话离头近，就从first开始找(第一个往后找)，大于的话离尾近，就从last开始找(最后一个往前找)。

25. 泛型好处：把运行时期的问题提前到了编译时期；避免强制类型转换；

26. 泛型类：类后面有<E>，表示这个类是一个泛型类，创建泛型类的对象时，必须要给这个泛型确定具体的数据类型；

1. 泛型类的方法：

①Object[] objects = list.toArray()，但是返回的Object，后续可能会需要强转；

②String[] strings = list.toArray(new String[list.size()])，可能直接转换成泛型类型；

28. 泛型方法：修饰符 <类型> 返回值类型 方法名(类型 变量名){}; public <T> void show(T t){};

29. 泛型通配符<?>：元素可以匹配任何的类型

<? extends Number>：类型通配符上限，Number或子类型；

<? super Number>：类型通配符下限，Number或父类型；

30. Set集合：是没有索引的，所以不能使用通过索引获取元素的方法，不能使用普通for;

31. Set之TreeSet：不包含重复元素的集合；没有带索引的方法；可以将元素按照规则进行排序；

32. TreeSet之自然排序：存储的自定义对象需要自定义排序规则，对象需要实现Comparable接口，重写compareTo()方法，原理：

①如果返回值是负数，表示当前存入的元素是较小值，存左边；

②如果返回值为0，表示当前存入的元素跟集合中元素重复了，不存；

③如果返回值为正数，表示当前存入的元素是较大值，存右边。

33. TreeSet之比较器排序：创建TreeSet时，使用带参构造方法，接收Comparator的实现类对象，重写compara(T o1，T o2)方法；

在使用的时候，默认使用自然排序，当自然排序不满足现在的需求时，使用比较器排序。

1. 二叉树：由父节点地址、值、左子节点地址和右子节点地址组成；

度：每一个节点的子节点数量，二叉树中，任意一个节点的度要小于等于2；

普通二叉树：整个树的数据毫无规律；

二叉查找树：又称二叉排序树或者二叉搜索树，每一个节点上最多有两个子节点；每一个节点的左子节点都是小于自己的，每一个节点的右子节点都是大于自己的；

平衡二叉树：二叉树左右两个子树的高度差不超过1；任意节点的左右两个子树都是一颗平衡二叉树；

左旋：就是将根节点的右侧往左拉，原先的右子节点变成新的父节点，并把多余的左子节点出让，给已经降级的根节点当右子节点。

右旋：将根节点的左侧往右拉，左子节点变成了新的父节点，并把多余的右子节点出让，给已经降级根节点当左子节点。

左左：当根节点左子树的左子树有节点插入，导致二叉树不平衡，整体进行右旋；

左右：当根节点左子树的右子树有节点插入，导致二叉树不平衡，需要先将左子树左旋，

再整体进行右旋，达到平衡；

右右：当根节点右子树的右子树有节点插入，导致二叉树不平衡，整体进行左旋；

右左：当根节点右子树的左子树有节点插入，导致二叉树不平衡，需要先将右子树右旋，再整体进行左旋，达到平衡；

红黑树：是一种自平衡的二叉查找树；根节点必须是黑色的；叶节点(Nil)是黑色的；两个相连节点不能是红色；对每个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点。

红黑树：如果两个红色相连，其父节点是红色，叔叔节点也是红色：将父节点设为黑色，叔叔节点设为黑色；将祖父节点设为红色；如果祖父节点为根节点，则将根节点再次变为黑色；

红黑树：如果两个红色相连，其父节点是红色，叔叔节点也是黑色：将父节点设为黑色；将祖父节点设为红色；以祖父节点为支点进行旋转；

1. HashSet：底层是哈希表；无序；没有索引，不能使用普通for；元素唯一。

哈希值：没有重写hashCode()是根据对象的地址值，算出来的int类型的整数；重写了hashCode()方法，一般都是通过对象的属性值计算出的哈希值。

36. HashSet源码解析：

1.7版本：

①底层结构：哈希表。（数组 + 链表）

②数组的长度默认是16，加载因子为0.75，当数组里边存了16\*0.75=12个元素的时候，数组就会扩容为原来的两倍。

③存入位置：会先获取元素的哈希值和数组长度计算应存入的索引，判断该索引处是否为null，如果是null，直接添加；如果不是null，则与链表中所有的元素，通过equals方法比较属性值，只要有一个相同，就不存，如果都不一样，才会存入集合，老元素挂在新元素的下面。

1.8版本：

①底层结构：哈希表。（数组 + 链表 + 红黑树）

②当链表长度为8，再添加自动转为红黑树。

③当存入的元素，已形成红黑树，继续用equals进行比较，但不是全部比较，小的跟左边比，大的跟右边比。

1. Map集合：键不能重复，值可以重复；键+值为Entry对象；
2. Map遍历两种方式：

①keySet()：先获取map所有的key，再遍历所有的key，获取值；

②entrySet()：获取所有的entry，再通过entry获取getKey()和getValue();

1. HashMap原理解析：

①HashMap跟HashSet一样底层是哈希表结构；

②HashMap在put时，会创建一个Entry对象，记录键和值，再通过hashCode方法根据键计算出Hash值，计算出索引，当存入的数组为空，直接存进去，不为空，就通过equal比较键的属性值，一样就会覆盖旧的，不一样，老元素就会挂载新元素下面，形成链表，链表长度大于8并且表的长度大于64的时候会转化为红黑树；

1. TreeMap原理解析：TreeMap跟TreeSet一样底层是红黑树结构的。
2. 可变参数：修饰符 返回值类型 方法名(数据类型... 变量名){} public static int sum(int... a){};
3. 可变参数：其实就是一个数组；如果一个方法有多个参数，包含可变参数，可变参数要放在最后。
4. File类：在读写数据时告诉虚拟机要操作的（文件/文件夹）在哪；对（文件/文件夹）本身进行操作。包括创建、删除等；
5. File三种构造方法：

①File file = new File(String path); 将字符串路径转换为抽象路径名来创建新的File;

②File file = new File(String parent, String child); 以父路径字符串和子路径字符串拼接创建File;

③File file = new File(File parent, String child); 以父抽象路径和子路径字符串拼接创建File;

1. File绝对路径和相对路径：

①绝对路径：从盘符开始；

②相对路径：相对当前项目下的路径；分两种情况：File file = new File(“a.txt”)文件的位置和项目平级；File file = new File(“模块名\\a.txt”)在当前项目模块下面。

1. File方法：

①createNewFile()：创建一个空的文件，返回true/false；

②mkdir()：创建一个单级文件夹，返回true/false；

③mkdirs()：创建一个多级文件夹，返回true/false；

④delete()：只能删除文件和空文件夹；

⑤listFiles()：获取文件夹下所有的文件及文件夹信息；

1. IO：可以将数据从硬盘中读取出来；可以将数据从内存保存到硬盘；
2. IO：input是数据从硬盘进内存的过程，称之为读；

output是数据从内存到硬盘的过程，称之为写；

1. IO字节流：操作所有类型文件；

IO字符流：只能操作纯文本文件，能用记事本打开不乱码就可以用字符流，否则用字节流；

1. FileOutputStream：字节流写数据，操作byte[]数组；

①创建字节输出流对象：如果文件不存在，就创建；如果文件存在就清空；

②写数据：如果写出的是整数，写出的是整数在码表上对应的字母；write()

③释放资源：每次使用完流必须要释放资源；close();

1. FileInputStream：字节流读数据，也是分为三步：

FileInputStream fileInputStream = **new** FileInputStream(**"F:\\aaa.xlsx"**);  
FileOutputStream fileOutputStream = **new** FileOutputStream(**"C:\\Desktop\\aaa.xlsx"**);  
byte[] bytes = new byte[1024]; //一次性读多长

**int** len = 0; //读到的有效个数  
**while**((len=fileInputStream.read(bytes)) != -1) {  
 fileOutputStream.write(bytes, 0, len);  
}

fileOutputStream.close();

fileInputStream.close();

1. BufferedInputStream字节缓冲输入流和BufferOutputStream字节缓冲输出流原理：缓

冲流都是为了提高读写效率，底层就是创建了一个长度为byte[] buf = new byte[8192]数组，硬盘到内存存在缓冲区，一次性读写8192个字节到缓冲区，其实整个流程都是在内存中进行的，比起字节流从硬盘到内存，效率非常快。

1. 字符流比起字节流的好处：字节流在操作汉字可能会出现中文乱码问题；

①windows默认使用码表：GBK，一个中文两个字节；idea和以后工作默认使用Unicode的UTF-8编码解码格式，一个中文三个字节。

②字节流一次读一个字节，在读中文时，每次只能读一部分所以出现了乱码。

③字符流读取中文时，字符流 = 字节流 + 编码表，不管在哪张码表中，中文的第一个字节一定是负数，读到一个字节为负数时，再判断码表(获取到UTF-8编码格式)，会一次读三个负数合成一个汉字。

1. FileReader和FileWriter：字符输入流和字符输出流，底层是字节流，一个字节一个字节进行读。可操作char[]数组。
2. BufferedWriter newLine()：字符输出缓冲流下换行；底层\r\n

56. BufferedReader readLine()：字符输入缓冲流，一次读一行；原理：一直读，读到回车换行为止。判断返回行的字符串 !=null;

57.转换流：

InputStreamReader：是从字节流到字符流的桥梁；

OutputStreamWriter：是从字符流到字节流的桥梁；

58.对象操作流：

ObjectInputStream：对象操作输入流（对象反序列化流），把写到本地文件中的对象读到内存中，或者接受网络传输的对象；readObject();

ObjectOutputStream：对象操作输出流（对象序列化流），把对象写到本地文件中，或者在网络中传输对象；writeObject();

59. Serializable接口：标记性接口，只要类实现了这个接口，就表示这个类的对象可以被序列化；

serialVersionUID序列号：如果我们没有定义，那么虚拟机会根据类中的信息会自动的计算出一个序列号。存在的问题：如果我们修改了类中的信息。那么虚拟机会再次计算出一个序列号，导致本地中的序列号和类中的序列号不一致，反序列化失败，解决方法：可以手动定义个序列号。

transient：给成员变量加transient关键字修饰，成员变量不参与序列化过程。private transient Integer age;

60. Properties：是一个Map体系的集合类，继承了HashTable；Map的方法它都能用，外加setProperty() = put()，getProperty() = get()，stringPropertyNames() = keySet();

①load(Reader reader)：将本地文件中的键值对数据读取到集合中；

②store(Writer writer, String comments) 将集合中的数据以键值对形式保存在本地；comments作用为第一行的注释。

61. 并行：在同一时刻，有多个指令在多个CPU上同时执行；

并发：在同一时刻，有多个指令在单个CPU上交替执行；

62. 进程：是正在运行的软件；

线程：是进程中的单个顺序控制流，是一个执行路径；

63. Thread类：

①run()：封装线程执行的代码，直接调用，相当于普通方法的调用，并没有开启线程；

②start()：启动线程；然后由JVM调用此线程的run()方法；

64. Runnable接口：可以操作共享数据。

①创建实现Runnable接口参数对象； MyRunnable mr = new MyRunnable();

②创建一个线程对象，并把参数传递给这个线程； Thread t1 = new Thread(mr);

③在线程启动之后，执行的就是参数里边的run方法；t1.start();

65. Callable + FuctureTask接口：可以有返回值

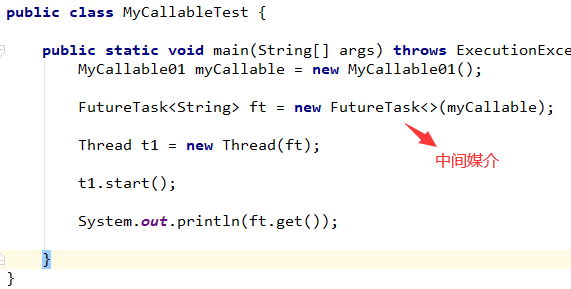
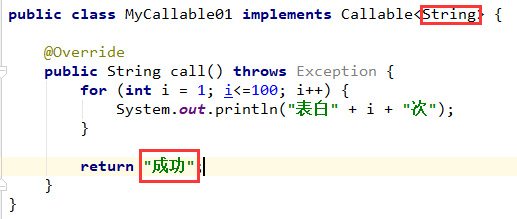
①定义一个类MyCallable实现Callable接口，重写call()方法，泛型设置为返回值得类型；

②测试：创建MyCallable类对象，再创建FuctureTask对象，把MyCallable对象 作为构造方法的参数，FuctureTask作为中间媒介，既可以获取线程执行完毕之后的结果，也可以作为参数传递给Thread对象；

③创建Thread类的对象，把FutureTask对象作为构造方法的参数；

④启动线程

⑤使用FutureTask对象调用get方法，就可以获取线程结束之后的结果，一定要放在启动线程之后，否则会一致等待获取数据；



66. getName()：获取线程名称；

setName(String name)：自定义设置线程名称；

Thread.currentThread().getName()：获取当前线程名称；

sleep()：线程睡眠；

setPriority()：线程调度：分时调度模型和抢占式调用模型(默认)，抢占式的如果线程优先级相同，那么会随机选择一个，优先级高的线程获取的CPU时间片相对多一些，可设置值1 - 10，默认是5，数字越大越先获取CPU时间片；

setDaemon(true)：当普通线程执行完之后，那么守护线程也没有继续运行下去的必要了；

67. synchronized(任意对象){} 同步代码：

锁一定要唯一，否则会出现重复消费数据，可以用static修饰，存入方法区保证只有一个副本。

同步方法放在非静态方法上是this锁；

同步方法放在静态方法上是class锁；

同步代码块可以用this锁或class锁；

68. Lock锁的使用：采用实现类ReentrantLock来实例化；

进入多线程共享代码是进行lock()加锁，当前线程消费后，unlock()解锁。

69. 死锁：是指两个或者多个线程互相持有对方所需要的资源，导致这些线程处于等待状态。

70. 等待和唤醒方法：体现生产和消费过程中的等待和唤醒，这几个方法在Object中；

wait()：导致当前线程等待，直到另一个线程调用该对象的notify()方法或notifyAll()方法；

notify()：唤醒正在等待对象监视器的单个线程；

notifyAll()：唤醒正在等待对象监视器的所有线程；

71. 阻塞队列实现等待唤醒机制，BlockingQueue的核心方法；

put(anObject)：将参数放入队列，如果放不进去会阻塞，底层有lock锁；

take()：取出第一个数据，取不到会阻塞，底层也有lock锁；

72. 常见BlockingQueue:

ArrayBlockingQueue：底层是数组，有界；

LinkedBlockingQueue：底层是链表，无界。但不是真正的无界，最大为int的最大值；

73. 虚拟机中线程的六种状态：

新建状态(NEW) 🡪 创建线程对象；

就绪状态(RUNNABLE) 🡪 调用start()方法；

阻塞状态(BLOCKED) 🡪 无法获得锁对象；

等待状态(WAITING) 🡪 wait()方法；

计时状态(TIMED\_WAITING) 🡪 sleep()方法；

结束状态(TERMINATED) 🡪 全部代码运行完毕；

74. 线程池：使用Executors类中的静态方法，有任务需要执行时，线程池会自动的帮我们创建对象，任务执行完毕，也会自动把线程对象归还池子；

submit()方法：利用方法将要执行的任务交给线程池；

shutdown()方法：任务全部执行完毕，关闭连接池；

75. Executors类中的静态方法，调用方法返回ExecutorService可操作submit()、shutdown()方法：

newCachedThreadPool()：创建一个默认的线程池对象；

newFixedThreadPool(int nThreads)：创建一个最多容纳nThreads个线程；

76. 自定义创建线程池对象ThreadPoolExecutor参数：

①核心线程数量；

②最大线程数；（大于等于核心线程数）

③空闲线程最大存活时间；（值）

④时间单位；（承接第三项，单位，使用TimeUnit枚举）

⑤任务队列；（任务比较多大于最大线程数，会放入阻塞队列）

⑥创建线程工厂；（按照默认的方式创建线程对象，底层就是new Thread(参数)）

⑦任务的拒绝策略；（任务过多，绝决服务策略）

78. ThreadPoolExecutor拒绝策略：

①什么时候拒绝任务？ 当提交的任务 > 池子最大线程数量 + 队列容量；

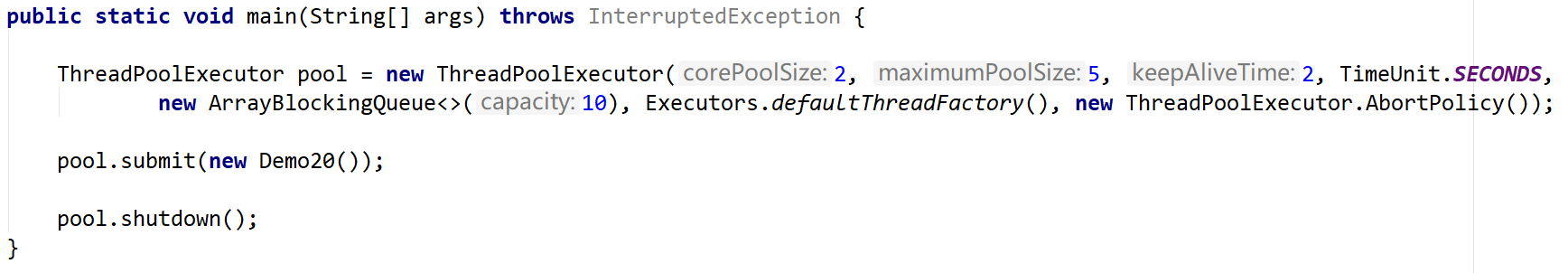
②ThreadPoolExecutor.AbortPolicy：丢弃任务并抛出异常。是默认的策略；

ThreadPoolExecutor.DiscardPolicy：丢弃任务，但是不抛出异常；

ThreadPoolExecutor.DiscardOldestPolicy：抛弃队列中等待最久的任务，然后把当前任务加入队列中；

ThreadPoolExecutor.CallerRunsPolicy：调用任务的run()方法绕过线程池直接执行。

77. ThreadPoolExecutor代码实现：



78. JMM:

①堆内存是唯一的，每一个线程都有自己的线程栈；

②每一个线程在使用堆里面变量的时候，都会先拷贝一份到变量的副本中；

③在线程中，每一次使用是从变量的副本中获取的。

④导致问题，其他线程修改了堆中共享变量的值，当前线程不一定能及时使用最新的值；

⑤使用Volatile解决问题，强制线程每次在使用的时候，都会看一下共享区域最新的值；

79. Synchronized同步代码块(悲观锁)对共享变量的操作：

①线程获得锁；

②清空变量副本；

③拷贝共享变量最新的值到变量副本中；

④执行代码；

⑤将修改后变量副本中的值赋值给共享数据；

⑥释放锁；

80. 原子性：是指在一次操作或多次操作中，要么所有的操作全部都得到了执行并且不会受到任何因素的干扰而中断，要么所有的操作都不执行，多个操作是一个不可分割的整体；

81. count++:

①从共享数据中读取数据到本线程栈中；

②修改本线程栈中变量副本的值；

③会把本线程栈中变量副本的值赋值给共享数据；

④多线程在执行的过程中，有可能被其他线程打断，不是一个原子性操作；

82. volatile:

①只能保证线程每次在使用共享数据的时候是最新值；

②但是不能保证原子性；

③Synchorized可以保证原子性；

83. AtomicInteger可以保证int数据的原子性：

①int get(); 获取值；

②int getAndIncrement()：以原子方式将当前值加1，注意，这里返回的是自增前的值；

③int incrementAndGet()：以原子方式将当前值加1，注意，这里返回的是自增后的值；

④int addAndGet(int data)：以原子方式将参数与对象中的值相加，并返回的结果；

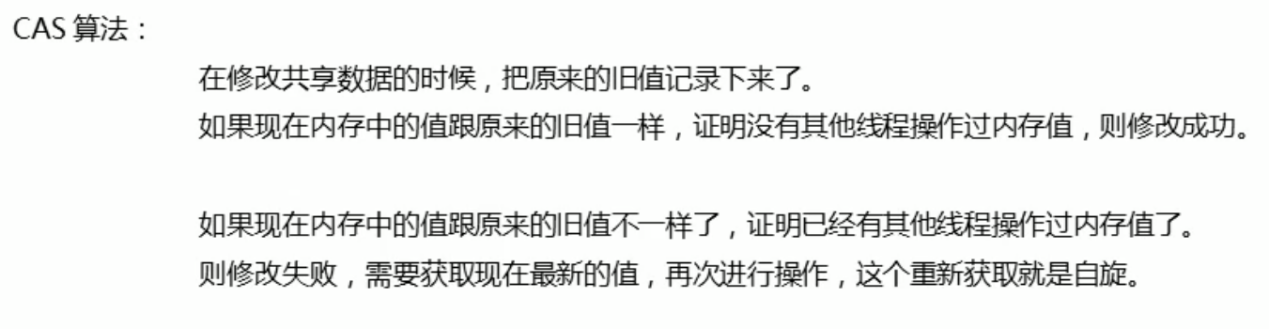
⑤int getAndSet(int value)：以原子方式设置为value的值，并返回旧值。

84. AtomicInteger原理解析：自旋锁 + CAS算法

CAS(乐观锁)算法：有3个操作数(内存值V，旧的预期值A，要修改的值B);

当旧的预期值A ==内存值 此时修改成功，将V改为B;

当旧的预期值A !=内存值 此时修改失败，不做任何操作，并重新获取现在的最新值(这个重新获取的动作就是自旋)



85. HashMap：多线程情况下，线程不安全的；

HashTable：采用悲观锁synchronized的形式保证数据的安全性；只要有线程访问，会将整个表全部锁起来，所以HashTable的效率低；

ConcurrentHashMap也是线程安全的，效率较高；

86. ConcurrentHashMap原理：

1.7版本：

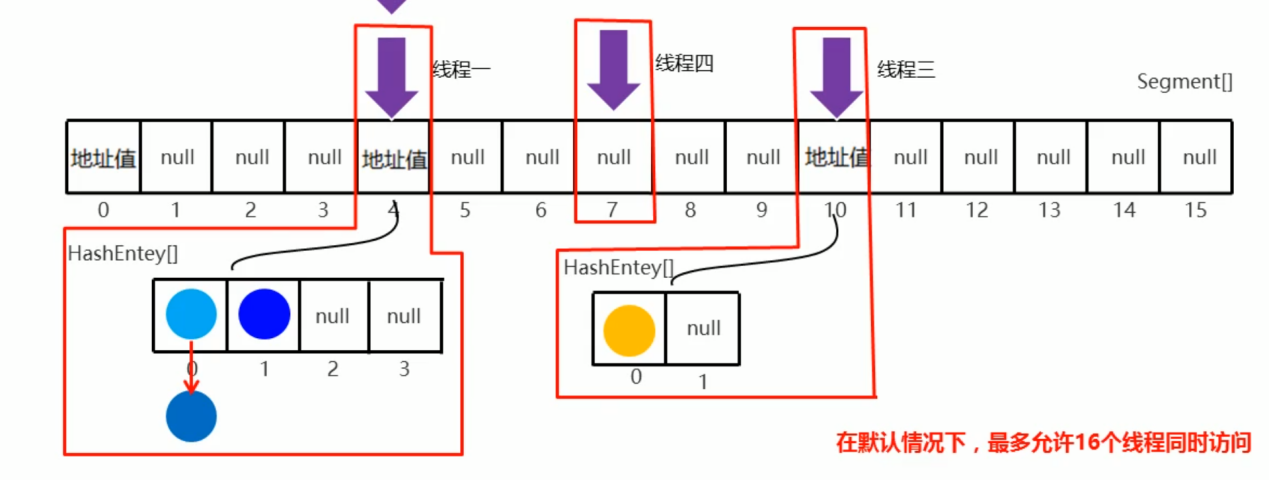
创建对象：默认创建一个长度16，加载因子为0.75的大数组(Segment[])，这个大数组一旦创建无法扩容；

还会创建一个长度为2的小数组(HashEntey[])，把地址值给0索引处，其他索引位置的元素都是null；

添加：第一次会根据键的哈希值来计算出在大数组中应存入的位置；

如果为null，则按照模板创建小数组，创建完毕会二次哈希，计算出在小数组中应存入的位置，直接存入；

如果不为null，就会根据记录的地址值找到小数组。二次哈希，计算出在小数组中应存入的位置，如果需要扩容，则将小数组扩容两倍；如果不需要扩容，则会判断小数组的这个位置有没有元素，如果没有元素，则直接存，如果有元素，就会调用equals方法，比较属性值，如果equals为true，则不存，如果equals为false，形成链表；



1.8版本：

①如果使用空参构造创建ConcurrentHashMap对象，则什么事情都不做，在第一次添加元素的时候创建哈希表；

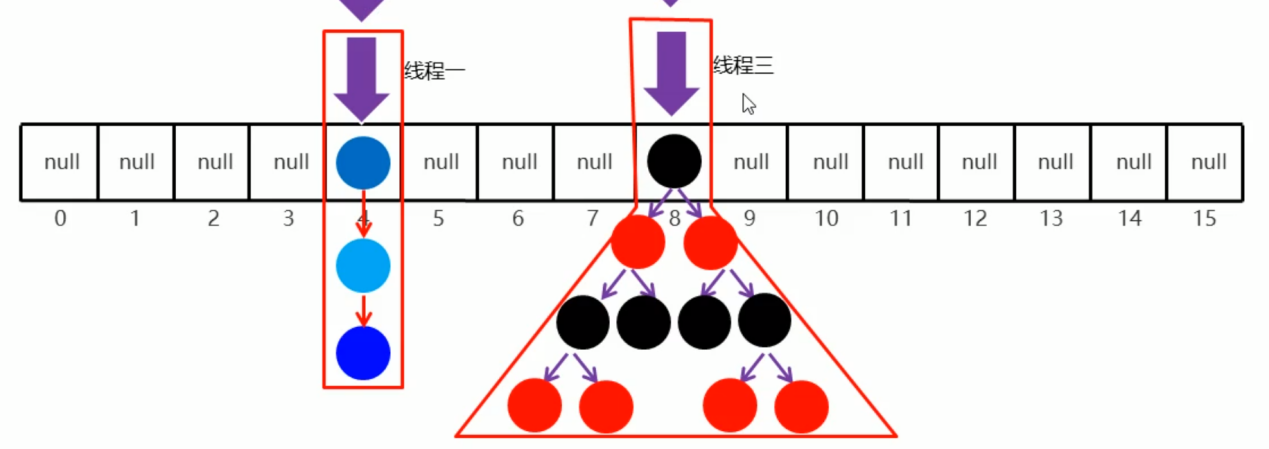
②计算当前元素应存入的索引；

③如果该索引位置为null，则利用cas算法，将本结点添加到数组中；

④如果该索引位置不为null，则利用volatile关键字获得当前位置最新的结点地址，挂在他下面，变成链表。

⑤当链表的长度大于等于8时，自动转换成红黑树；

⑥以链表或者红黑树头结点为锁对象，配合悲观锁保证多线程操作集合时数据的安全性。



87. CountDownLatch并发工具类：让某一条线程等待其他线程执行完毕之后再执行；

CountDownLatch(int count)：参数写等待线程的数量。并定义了一个计数器；

await()：让线程等待，当计数器为0时，会唤醒等待的线程；

countDown()：线程执行完毕时调用，会将计数器-1。

88. Semaphore并发工具类：可控制访问特定资源的线程数量。

Semaphore(int permits)：构造方法限制同时访问线程数量。

acquire()：实例调用，获得通行证；

release()：实例调用，访问完资源后收回许可证；