

ArrayList:

      ArrayList数组线性表的特点为:类似数组的形式进行存储，因此它的随机访问速度极快。

      ArrayList数组线性表的缺点为:不适合于在线性表中间需要频繁进行插入和删除操作。因为每次插入和删除都需要移动数组中的元素。

      可以这样理解ArrayList就是基于数组的一个线性表，只不过数组的长度可以动态改变而已。

      对于ArrayList的详细使用信息以及创建的过程可以查看jdk中ArrayList的源码，这里不做过多的讲解。

      对于使用ArrayList的开发者而言，下面几点内容一定要注意啦，尤其找工作面试的时候经常会被问到。

      注意啦！！！！！！！！

      a.如果在初始化ArrayList的时候没有指定初始化长度的话，默认的长度为10.

b.ArrayList在增加新元素的时候如果超过了原始的容量的话，ArrayList扩容ensureCapacity的方案为“原始容量\*3/2+1"

 c.ArrayList是线程不安全的，在多线程的情况下不要使用。如果一定在多线程使用List的，您可以使用Vector，因为Vector和ArrayList基本一致，区别在于Vector中的绝大部分方法都使用了同步关键字修饰，这样在多线程的情况下不会出现并发错误哦，还有就是它们的扩容方案不同，ArrayList是通过原始容量\*3/2+1,而Vector是允许设置默认的增长长度，Vector的默认扩容方式为原来的2倍。切记Vector是ArrayList的多线程的一个替代品。

d.ArrayList实现遍历的几种方法

LinkedList和ArrayList的区别和联系

      ArrayList数组线性表的特点为:类似数组的形式进行存储，因此它的随机访问速度极快。

      ArrayList数组线性表的缺点为:不适合于在线性表中间需要频繁进行插入和删除操作。因为每次插入和删除都需要移动数组中的元素。

       LinkedList的链式线性表的特点为: 适合于在链表中间需要频繁进行插入和删除操作。

       LinkedList的链式线性表的缺点为: 随机访问速度较慢。查找一个元素需要从头开始一个一个的找。速度你懂的。

Set接口的常见实现类有HashSet,LinedHashSet和TreeSet这三个。下面我们将分别讲解这三个类。

      1)HashSet

         HashSet是Set接口的最常见的实现类了。其底层是基于Hash算法进行存储相关元素的。

a.HashSet中存放null值

        HashSet中时允许出入null值的，但是在HashSet中仅仅能够存入一个null值哦。

     b.HashSet中存储元素的位置是固定的

        HashSet中存储的元素的是无序的，这个没什么好说的，但是由于HashSet底层是基于Hash算法实现的，使用了hashcode，

        所以HashSet中相应的元素的位置是固定

LinkedHashSet而言，它和HashSet主要区别在于LinkedHashSet中存储的元素是在哈希算法的基础上增加了

     链式表的结构。

TreeSet

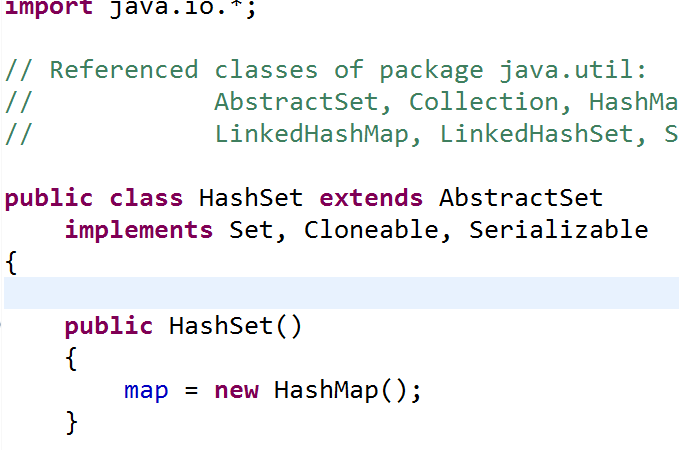
       TreeSet是一种排序二叉树。存入Set集合中的值，会按照值的大小进行相关的排序操作。底层算法是基于红黑树来实现的。

       TreeSet和HashSet的主要区别在于TreeSet中的元素会按照相关的值进行排序~

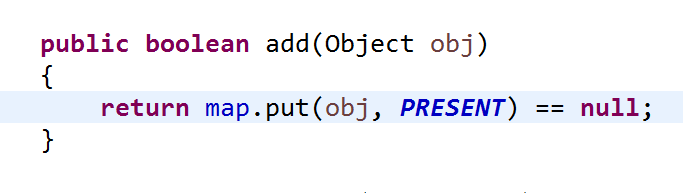
       TreeSet和HashSet的区别和联系

       1. HashSet是通过HashMap实现的,TreeSet是通过TreeMap实现的,只不过Set用的只是Map的key  
       2. Map的key和Set都有一个共同的特性就是集合的唯一性.TreeMap更是多了一个排序的功能.  
       3. hashCode和equal()是HashMap用的, 因为无需排序所以只需要关注定位和唯一性即可.  
           a. hashCode是用来计算hash值的,hash值是用来确定hash表索引的.  
           b. hash表中的一个索引处存放的是一张链表, 所以还要通过equal方法循环比较链上的每一个对象  
              才可以真正定位到键值对应的Entry.  
           c. put时,如果hash表中没定位到,就在链表前加一个Entry,如果定位到了,则更换Entry中的value,并返回旧value  
       4. 由于TreeMap需要排序,所以需要一个Comparator为键值进行大小比较.当然也是用Comparator定位的.  
           a. Comparator可以在创建TreeMap时指定  
           b. 如果创建时没有确定,那么就会使用key.compareTo()方法,这就要求key必须实现Comparable接口.  
           c. TreeMap是使用Tree数据结构实现的,所以使用compare接口就可以完成定位了.

Hashset的底层和hashmap类似。



在设置value值时是设置成了固定值





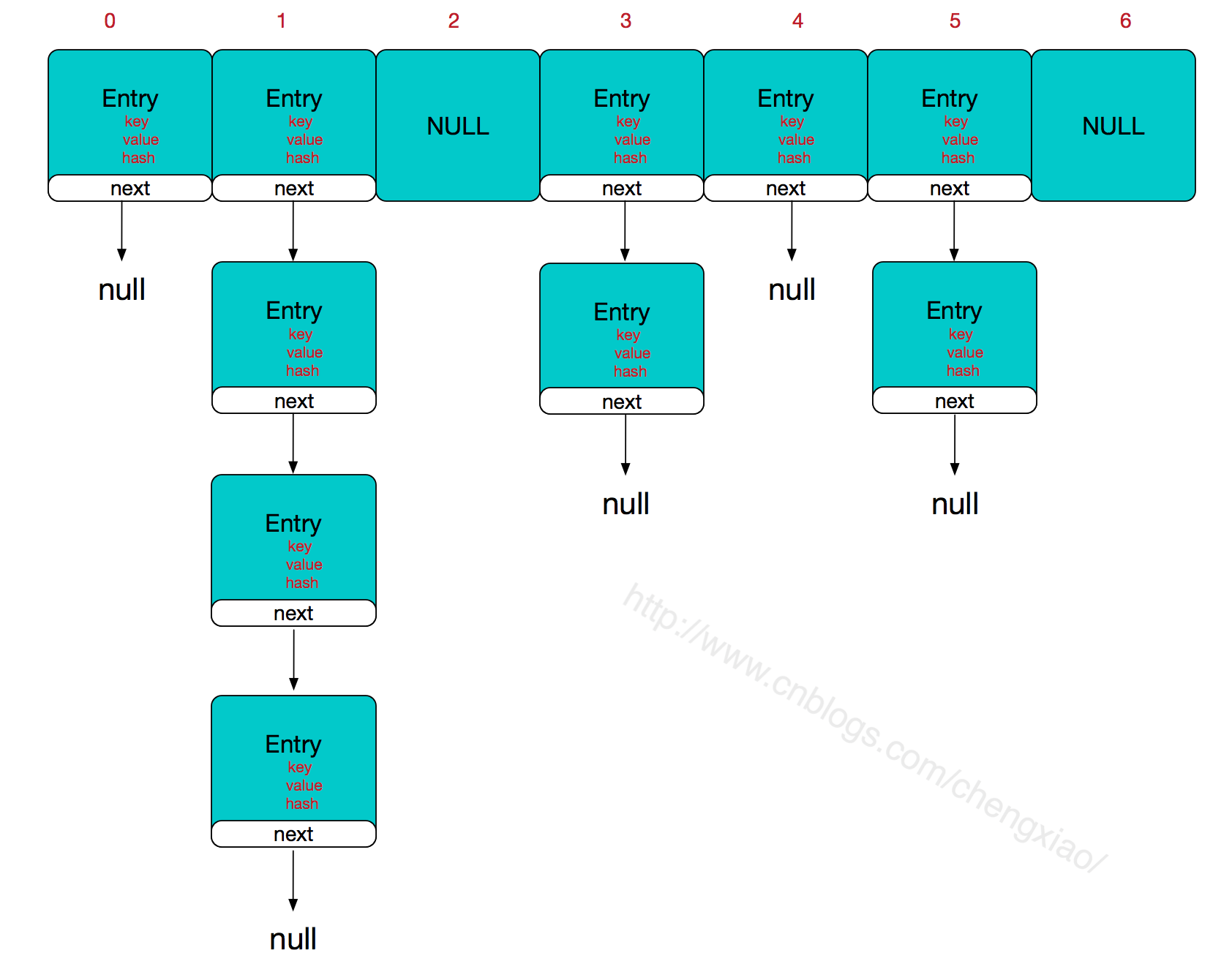
List和Set的区别

1. set不允许重复，list可以重复
2. set没有顺序，list有顺序

List：是存储单列数据的集合，存储的数据是有序并且是可以重复的   
Map：存储双列数据的集合，通过键值对存储数据，存储 的数据是无序的，Key值不能重复，value值可以重复

哈续表

HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对。



ConcurrentHashMap

 CAS、ReentrantLock、UNSAFE 、红黑树

简单理解就是，ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的意思，所以很多地方都会将其描述为分段锁。

**扩容: rehash**

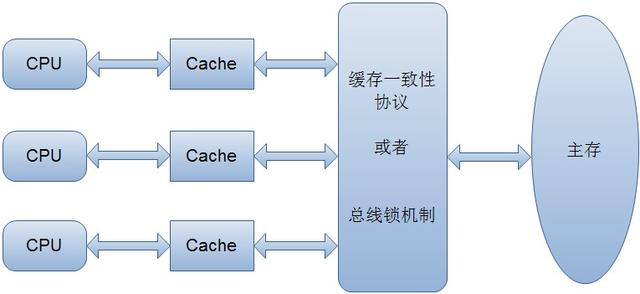
重复一下，segment 数组不能扩容，扩容是 segment 数组某个位置内部的数组 HashEntry\[] 进行扩容，扩容后，容量为原来的 2 倍。

锁

Volatile

Java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致地更新，线程应该确保通过排他锁单独获取这个变量。Java语言提供了volatile。在某些情况下比锁要更加方便。

Volatile在多处理器开发中保证了共享变量的可见性。可见性的意思就是当一个线程修改一个共享变量时，另一个线程能读到这个修改的值。



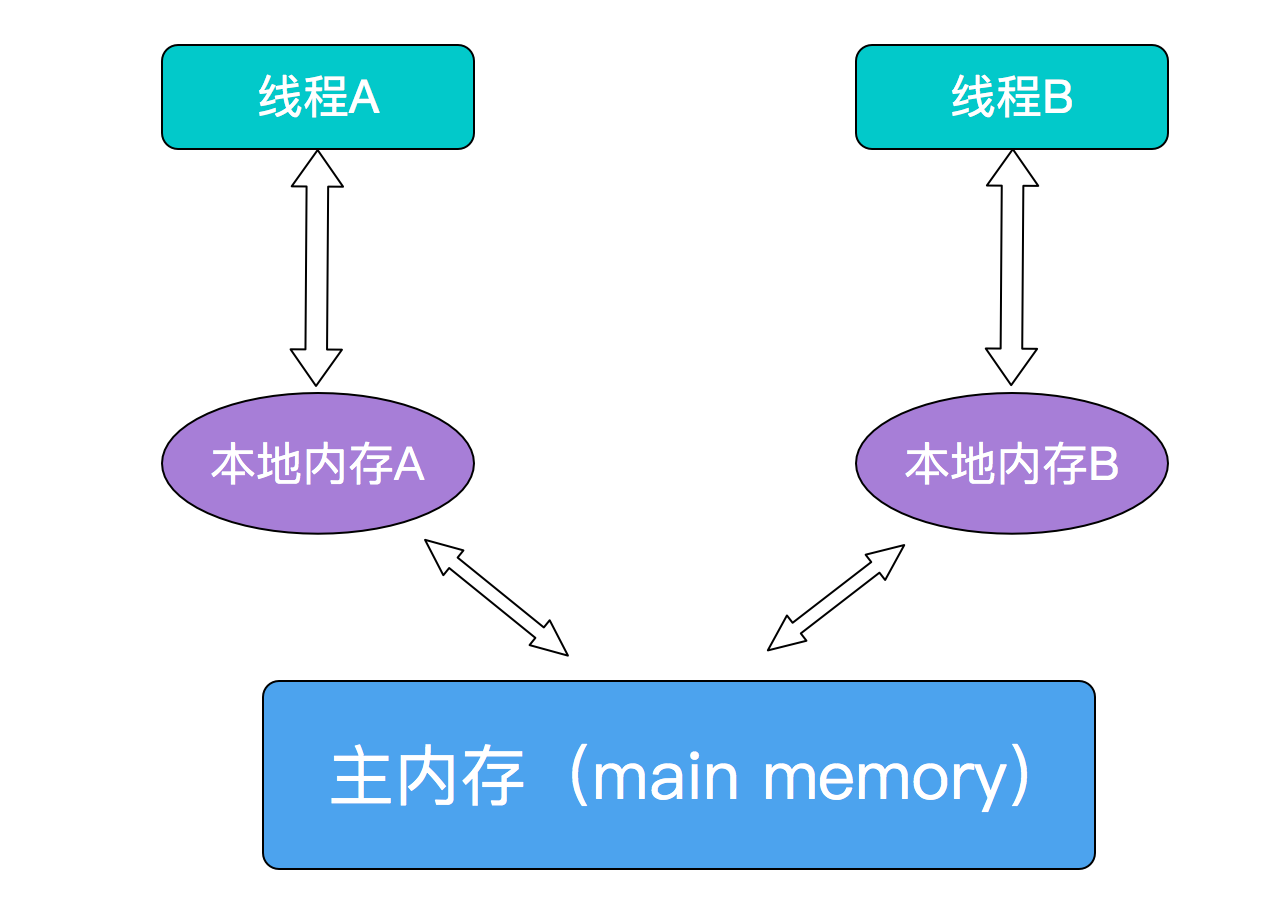
Volatile变量修饰符使用恰当的话，它比synchronized的使用和执行成本更低，因为它不会引起线程上下文的切换和调度。

**Java内存模型**

为什么出现这种情况呢，我们需要先了解一下**JMM（java内存模型）**

**java虚拟机有自己的内存模型（Java Memory Model，JMM），JMM可以屏蔽掉各种硬件和操作系统的内存访问差异，以实现让java程序在各种平台下都能达到一致的内存访问效果。**

**JMM决定一个线程对共享变量的写入何时对另一个线程可见，JMM定义了线程和主内存之间的抽象关系：共享变量存储在主内存(Main Memory)中，每个线程都有一个私有的本地内存（Local Memory），本地内存保存了被该线程使用到的主内存的副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。这三者之间的交互关系如下**



　需要注意的是，JMM是个抽象的内存模型，所以所谓的本地内存，主内存都是抽象概念，并不一定就真实的对应cpu缓存和物理内存。当然如果是出于理解的目的，这样对应起来也无不可。

volatile不能保证原子性

++操作是个符合操作：读取、加一、赋值。

|  |
| --- |
| package test;  import java.util.concurrent.CountDownLatch;  /\*\*  \* Created by chengxiao on 2017/3/18.  \*/  public class Counter {  public static volatile int num = 0;  //使用CountDownLatch来等待计算线程执行完  static CountDownLatch countDownLatch = new CountDownLatch(30);  public static void main(String []args) throws InterruptedException {  //开启30个线程进行累加操作  for(int i=0;i<30;i++){  new Thread(){  public void run(){  for(int j=0;j<10000;j++){  num++;//自加操作  }  countDownLatch.countDown();  }  }.start();  }  //等待计算线程执行完  countDownLatch.await();  System.out.println(num);  }  } |

针对num++这类复合类的操作，可以使用java并发包中的原子操作类原子操作类是通过循环CAS的方式来保证其原子性的。

volatile还有一个特性：**禁止指令重排序优化。**

重排序是指编译器和处理器为了优化程序性能而对指令序列进行排序的一种手段。但是重排序也需要遵守一定规则：

volatile禁止指令重排序也有一些规则，简单列举一下：

**1.当第二个操作是voaltile写时，无论第一个操作是什么，都不能进行重排序**

**2.当地一个操作是volatile读时，不管第二个操作是什么，都不能进行重排序**

**3.当第一个操作是volatile写时，第二个操作是volatile读时，不能进行重排序**

Synchronized通过锁机制实现同步。

具体表现为以下3种形式。

* 对于普通同步方法，锁是当前实例对象。
* 对于静态同步方法，锁是当前类的Class对象。
* 对于同步方法块，锁是Synchonized括号里配置的对象。

ynchronized是基于Monitor来实现同步的。

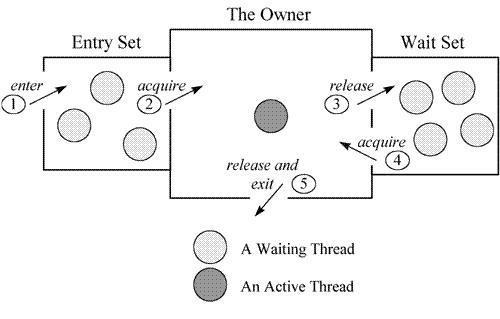
Monitor从两个方面来支持线程之间的同步：

* 互斥执行
* 协作

1、Java 使用对象锁 ( 使用 synchronized 获得对象锁 ) 保证工作在共享的数据集上的线程互斥执行。

2、使用 notify/notifyAll/wait 方法来协同不同线程之间的工作。

3、Class和Object都关联了一个Monitor。



同步代码块采用monitorenter、monitorexit指令显式的实现。

同步方法则使用ACC\_SYNCHRONIZED标记符隐式的实现。

Monitorenter

每个对象都用一个monitor，一个monitor只能被一个线程拥有。当一个线程执行到monitorenter指令时会尝试获取相应对象的monitor对象，获取规则如下：

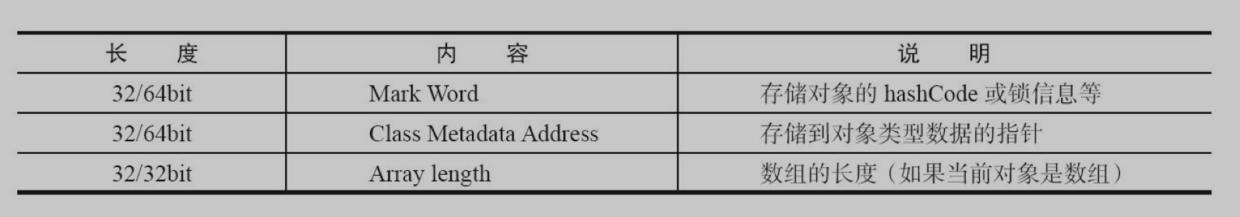
1. 如果monitor的进入数为0，则该线程可以进入monitor，并将monitor的进入数设置为1,该线程即为monitor的拥有者。
2. 如果当前线程已经拥有该monitor，只能重新进入，则进入monitor的进入数加1，所以synchronized关键字实现的锁是可重入的锁。
3. 如果monitor已被其他线程拥有，则当前线程进入阻塞状态，直到进入数为0，重新尝试获取monitor。

Monitorexit

只有拥有相应对象的monitor的线程才能执行monitorexit指令。每执行一次该指令monitor进入数减1，当进入数为0时当前线程释放monitor，此时其他阻塞的线程将可以尝试获取该monitor。

锁存放的位置

锁标记存放在Java对象头的Mark Word中。





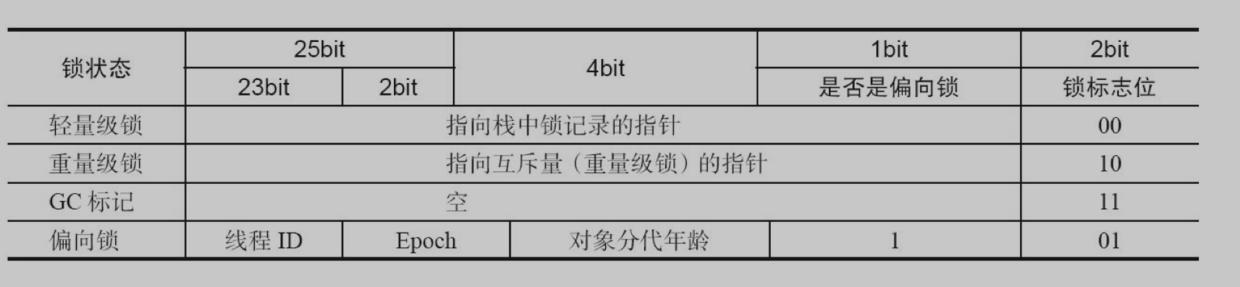
锁分为无锁状态、偏向锁状态、轻量级锁和重量级锁状态（jdk1.6）,锁只能升级不能降级。

偏向锁：

无锁竞争的情况下为了减少锁竞争的资源开销，引入偏向锁。

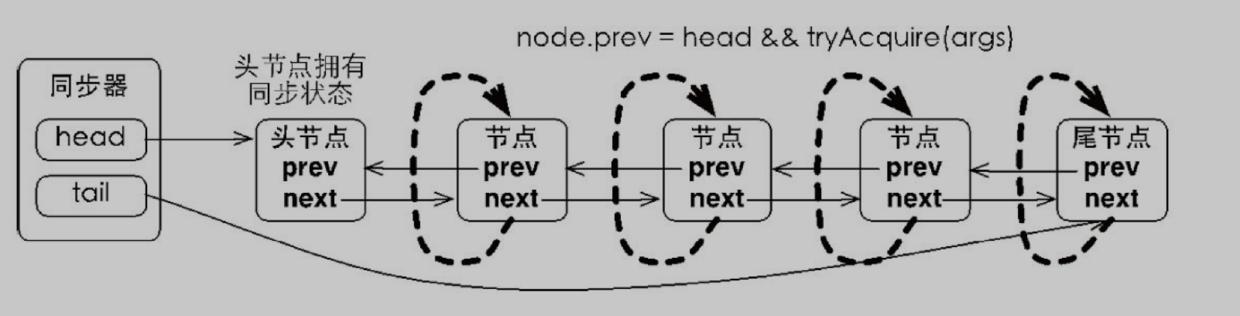
轻量级锁：

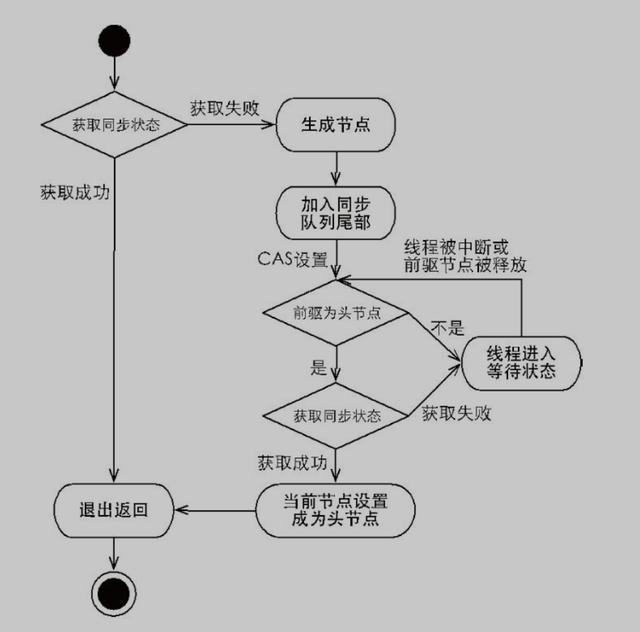
轻量级锁所适应的场景是线程交替执行同步块的情况。



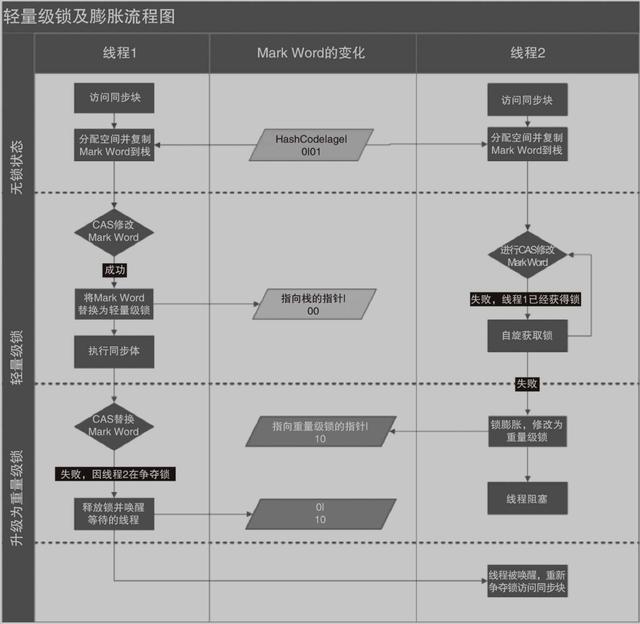
独占式：有且只有一个线程能获取到锁，如：ReentrantLock。

每个节点自旋观察自己的前一节点是不是Header节点，如果是，就去尝试获取锁。



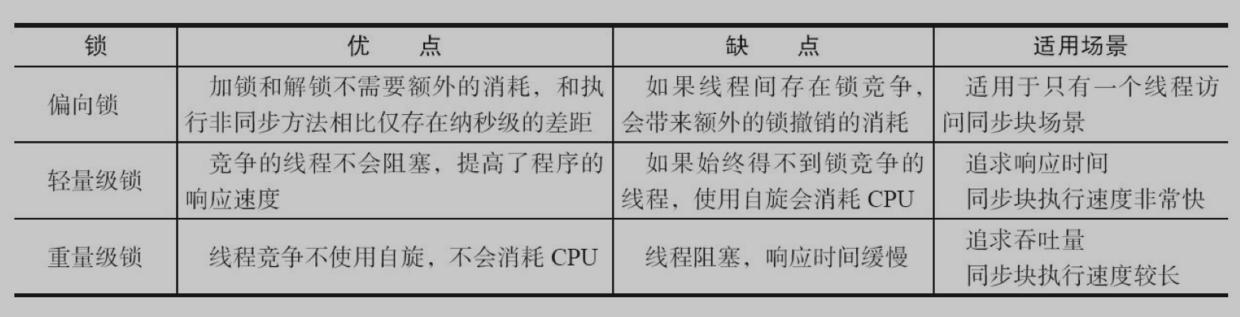


共享式：可以多个线程同时获取到锁，如：CountDownLatch



锁粗化：就是减少不必要的紧连在一起的unlock、lock操作，将多个连续的锁扩展成一个范围更大的锁

锁消除：锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不能存在共享数据竞争的锁进行消除。



CAS，在Java并发应用中通常指CompareAndSwap或CompareAndSet，即比较并交换。

1、CAS是一个原子操作，它比较一个内存位置的值并且只有相等时修改这个内存位置的值为新的值，保证了新的值总是基于最新的信息计算的，如果有其他线程在这期间修改了这个值则CAS失败。CAS返回是否成功或者内存位置原来的值用于判断是否CAS成功。

2、JVM中的CAS操作是利用了处理器提供的CMPXCHG指令实现的。

优点：

* 竞争不大的时候系统开销小。

缺点：

* 循环时间长开销大。
* ABA问题。
* 只能保证一个共享变量的原子操作。

ReentrantLock：重入锁