你必须要知道的锁原理、锁优化、CAS、AQS

景小财 Java技术驿站 2019-04-24

精品专栏

- 死磕 Java 并发
- 死磕 Sharding-jdbc
- 死磕 Spring 之 IOC

> 出处：https://www.jianshu.com/p/e674ee68fd3f

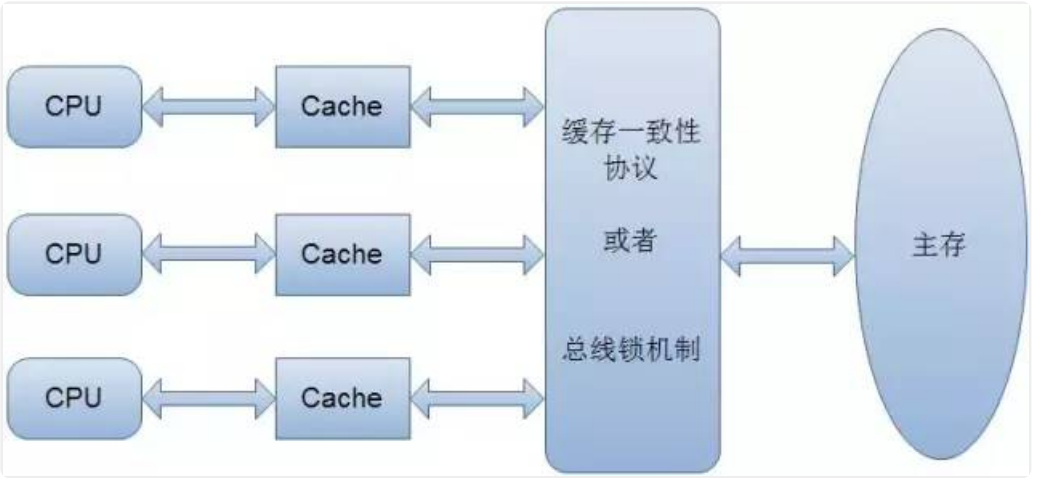
1、为什么要用锁？

锁-是为了解决并发操作引起的脏读、数据不一致的问题。

2、锁实现的基本原理

2.1、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得这个变量。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”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。



结论：如果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。

2.2、synchronized

> synchronized通过锁机制实现同步。

先来看下利用synchronized实现同步的基础：Java中的每一个对象都可以作为锁。

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。

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。

2.2.1 synchronized实现原理

> synchronized是基于Monitor来实现同步的。

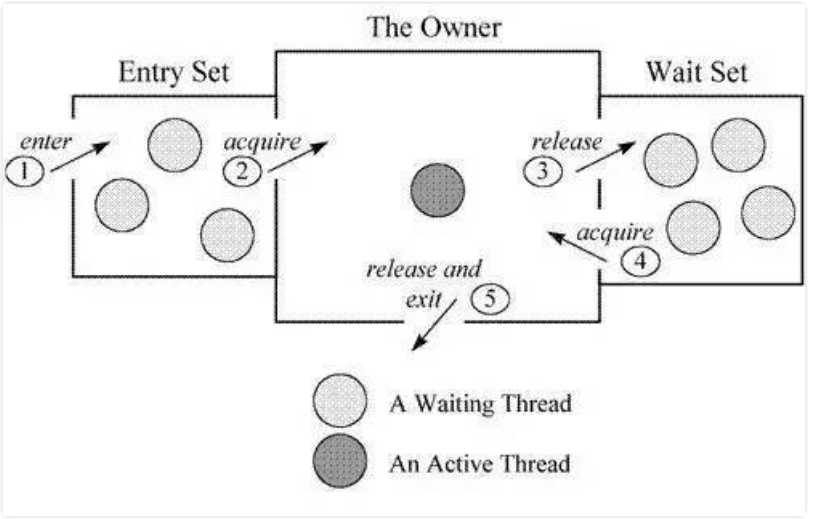
Monitor从两个方面来支持线程之间的同步：

- 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。



- 线程进入同步方法中。
- 为了继续执行临界区代码，线程必须获取 Monitor 锁。如果获取锁成功，将成为该监视者对象的拥有者。任一时刻内，监视者对象只属于一个活动线程（The Owner）
- 拥有监视者对象的线程可以调用 wait() 进入等待集合（Wait Set），同时释放监视锁，进入等待状态。
- 其他线程调用 notify() / notifyAll() 接口唤醒等待集合中的线程，这些等待的线程需要**重新获取监视锁后**才能执行 wait() 之后的代码。
- 同步方法执行完毕了，线程退出临界区，并释放监视锁。

参考文档：https://www.ibm.com/developerworks/cn/java/j-lo-synchronized

2.2.2 synchronized具体实现

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

2、同步方法则使用ACC_SYNCHRONIZED标记符隐式的实现。

通过实例来看看具体实现：

```
public class SynchronizedTest {

   public synchronized void method1(){
   System.out.println("Hello World!");
   }

   public  void method2(){
   synchronized (this){
   System.out.println("Hello World!");
   }
   }
}
```

javap编译后的字节码如下：

```
  public synchronized void method1();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED//同步方法的实现
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field java/lang/System.out:Ljava/io
         3: ldc           #3                  // String Hello World!
         5: invokevirtual #4                  // Method java/io/PrintStream.println:
         8: return
      LineNumberTable:
        line 9: 0
        line 10: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Lsync/SynchronizedTest;

  public void method2();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=1
         0: aload_0
         1: dup
         2: astore_1
         3: monitorenter//同步代码块的实现
         4: getstatic     #2                  // Field java/lang/System.out:Ljava/io
         7: ldc           #3                  // String Hello World!
         9: invokevirtual #4                  // Method java/io/PrintStream.println:
        12: aload_1
        13: monitorexit//同步代码块的实现
        14: goto          22
        17: astore_2
        18: aload_1
        19: monitorexit
        20: aload_2
        21: athrow
        22: return
      Exception table:
       ...
      LineNumberTable:
       ...略
      LocalVariableTable:
       ...略
      StackMapTable: number_of_entries = 2
       ...略
}
SourceFile: "SynchronizedTest.java"
```

monitorenter

每一个对象都有一个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，获取规则如下：

- 如果monitor的进入数为0，则该线程可以进入monitor，并将monitor进入数设置为1，该线程即为monitor的拥有者。
- 如果当前线程已经拥有该monitor，只是重新进入，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。
- 如果monitor已被其他线程拥有，则当前线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取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。

2.2.3 锁存放的位置

锁标记存放在Java对象头的Mark Word中。

长　度	内　容	说　明
32/64bit	Mark Word	存储对象的 hashCode 或锁信息等
32/64bit	Class Metadata Address	存储到对象类型数据的指针
32/32bit	Array length	数组的长度（如果当前对象是数组）

锁状态	25bit	4bit	1bit 是否是偏向锁	2bit 锁标志位
无锁状态	对象的 hashCode	对象分代年龄	0	01

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC 标记	空				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

锁状态	25bit	31bit	1bit	4bit	1bit	2bit
			cms_free	分代年龄	偏向锁	锁标志位
无锁	unused	hashCode			0	01
偏向锁	ThreadID(54bit) Epoch(2bit)				1	01

2.2.3 synchronized的锁优化

JavaSE1.6为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”。

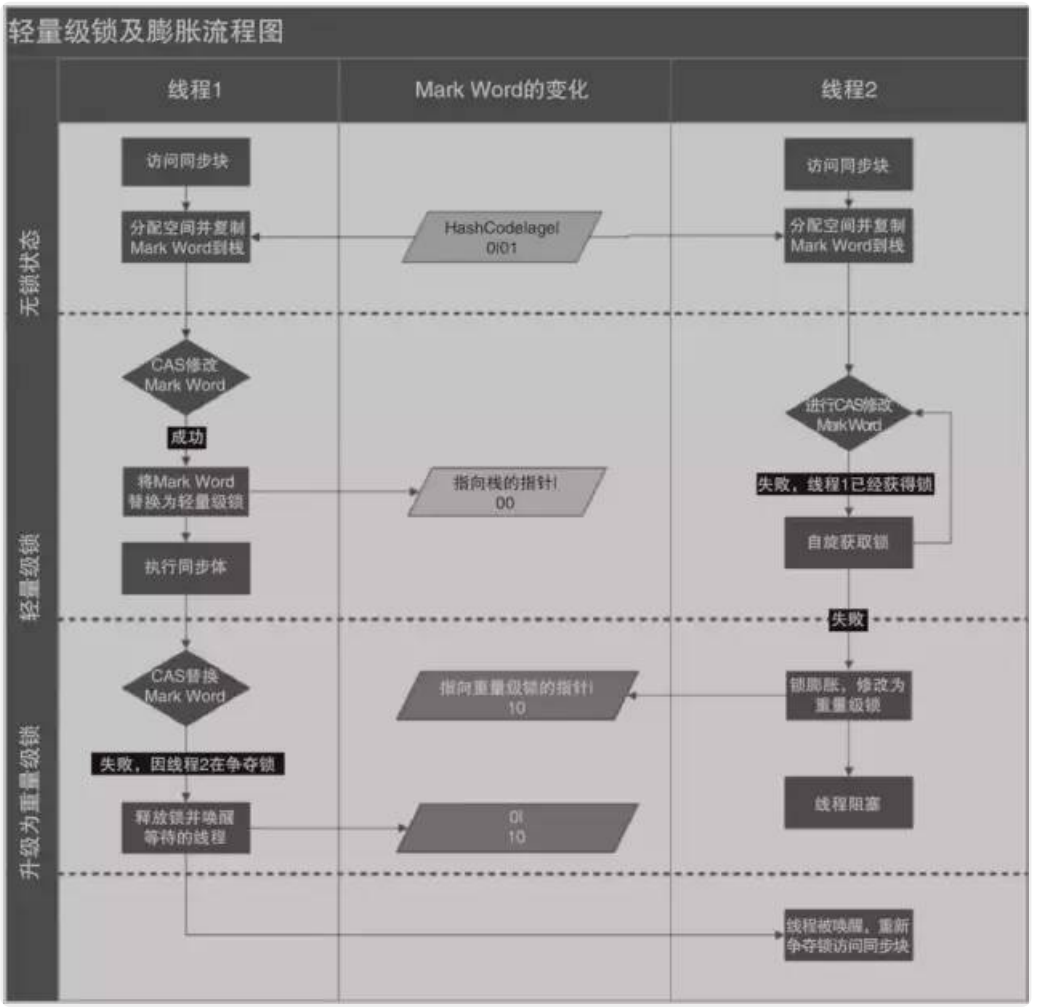
在JavaSE1.6中，锁一共有4种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态，这几个状态会随着竞争情况逐渐升级。

锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级

的策略，目的是为了提高获得锁和释放锁的效率。

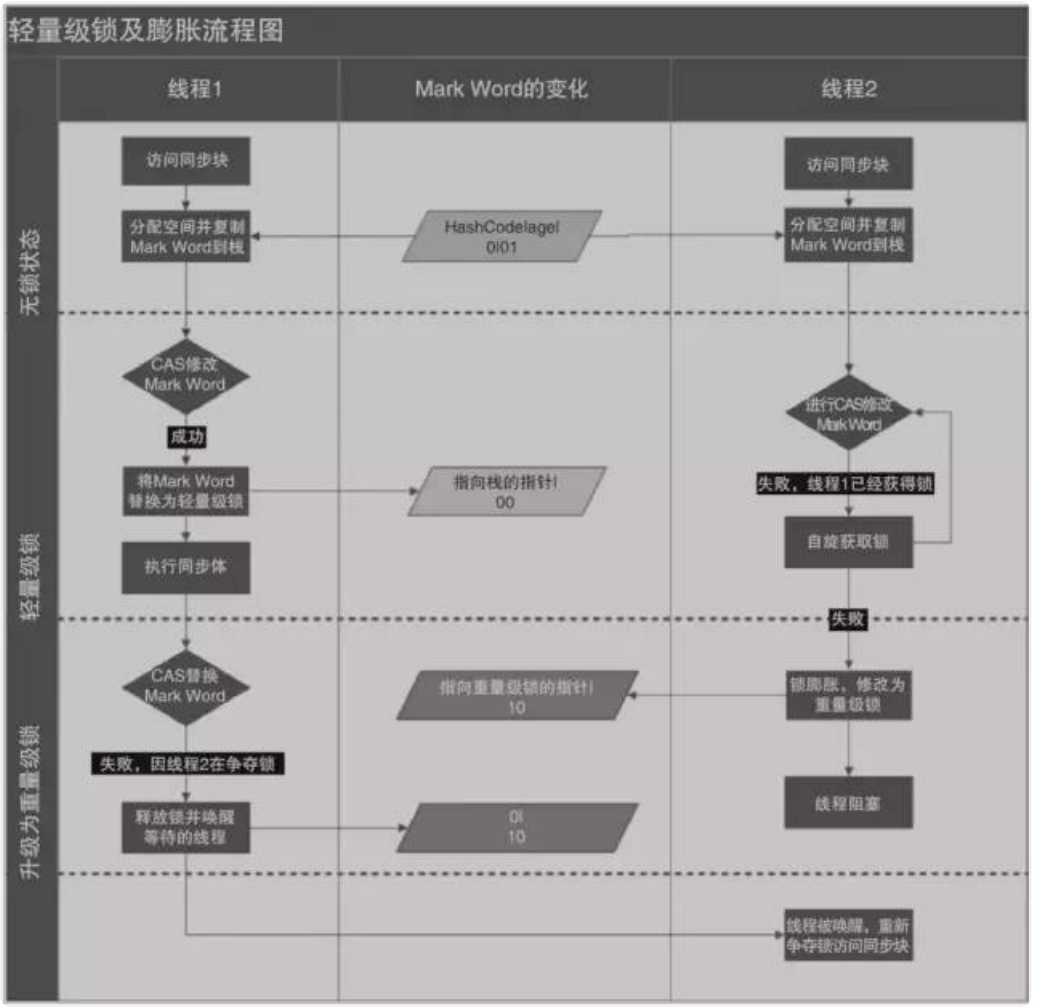
偏向锁：

> 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。



锁粗化（Lock Coarsening）： 也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（Lock Elimination）： 锁削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，但是被检测到不可能存在共享数据竞争的锁进行削除。

适应性自旋（Adaptive Spinning）： 自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间，比如100个循环。另一方面，如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。

2.2.4 锁的优缺点对比

锁	优　点	缺　点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗CPU	追求响应时间 同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块执行速度较长

2.3、CAS

> 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。

3、Java中的锁实现

3.1、队列同步器（AQS）

> 队列同步器AbstractQueuedSynchronizer（以下简称同步器），是用来构建锁或者其他同步组件的基础框架。

3.1.1、它使用了一个int成员变量表示同步状态。

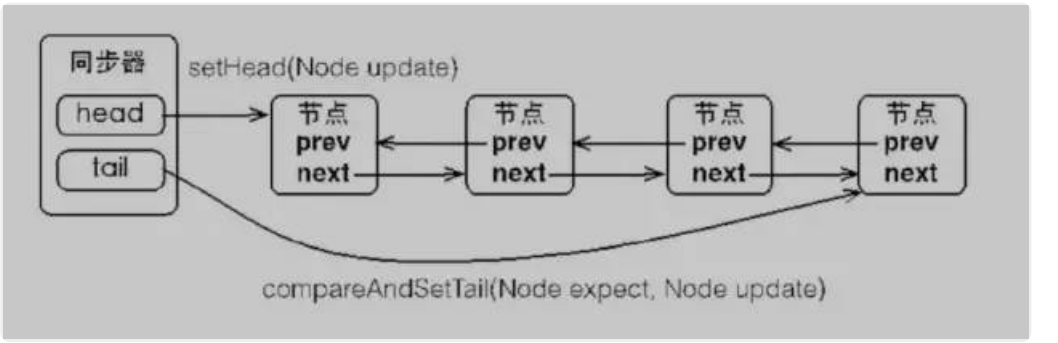
·getState()：获取当前同步状态。

·setState(int newState)：设置当前同步状态。

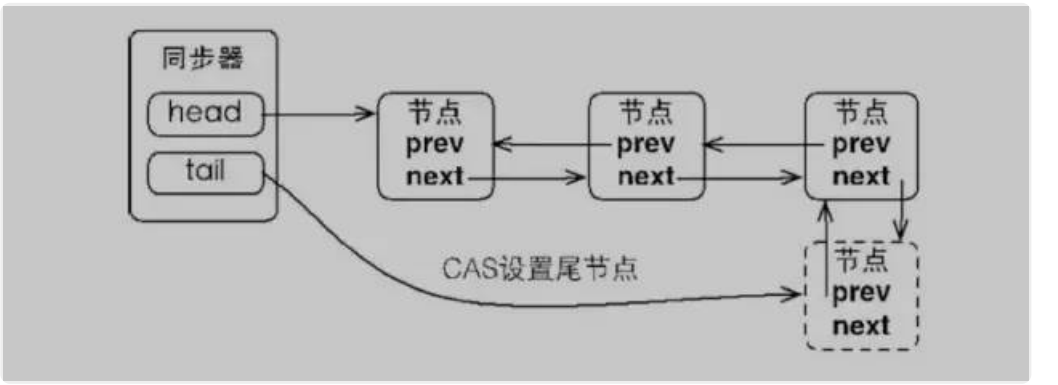
·compareAndSetState(int expect,int update)：使用CAS设置当前状态，该方法能够保证状态设置的原子性。

3.1.2、通过内置的FIFO双向队列来完成获取锁线程的排队工作。

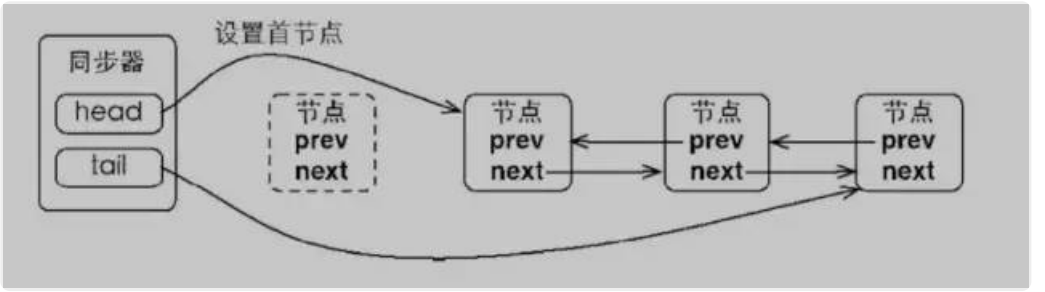
- 同步器包含两个节点类型的应用，一个指向头节点，一个指向尾节点，未获取到锁的线程会创建节点线程安全（compareAndSetTail）的加入队列尾部。同步队列遵循FIFO，首节点是获取同步状态成功的节点。



- 未获取到锁的线程将创建一个节点，设置到尾节点。如下图所示：



- 首节点的线程在释放锁时，将会唤醒后继节点。而后继节点将会在获取锁成功时将自己设置为首节点。如下图所示：

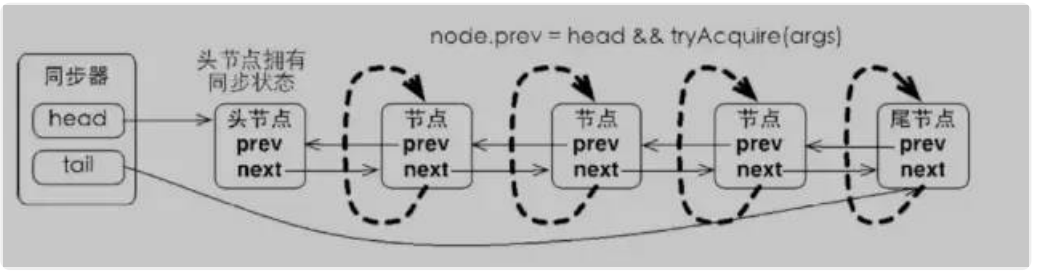


3.1.3、独占式/共享式锁获取

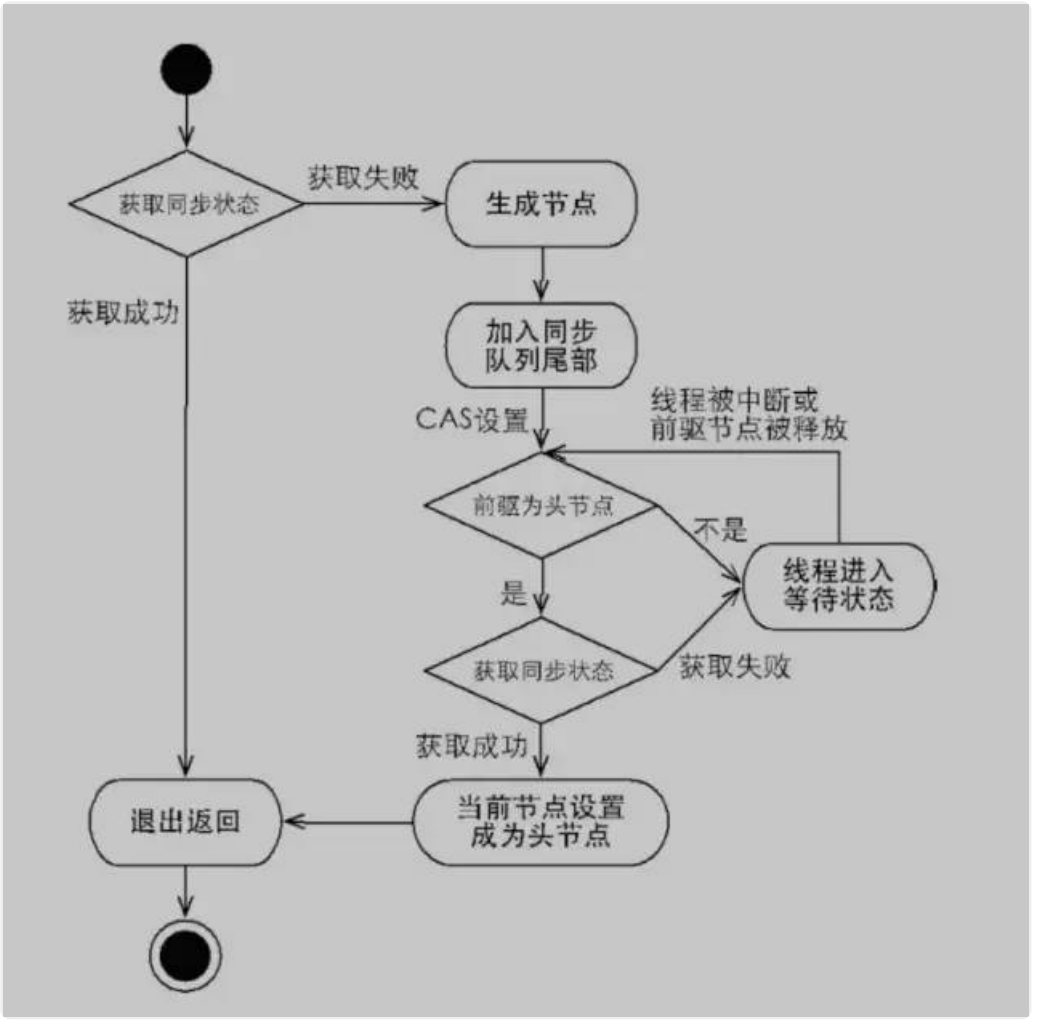
> 独占式：有且只有一个线程能获取到锁，如：ReentrantLock。
>
> 共享式：可以多个线程同时获取到锁，如：CountDownLatch

独占式

- 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。

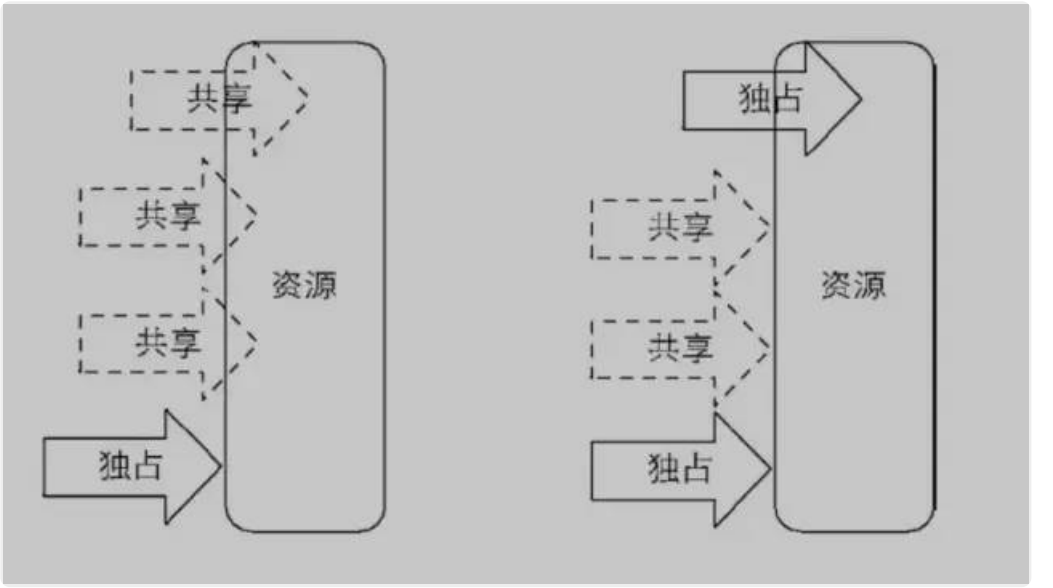


- 独占式锁获取流程：

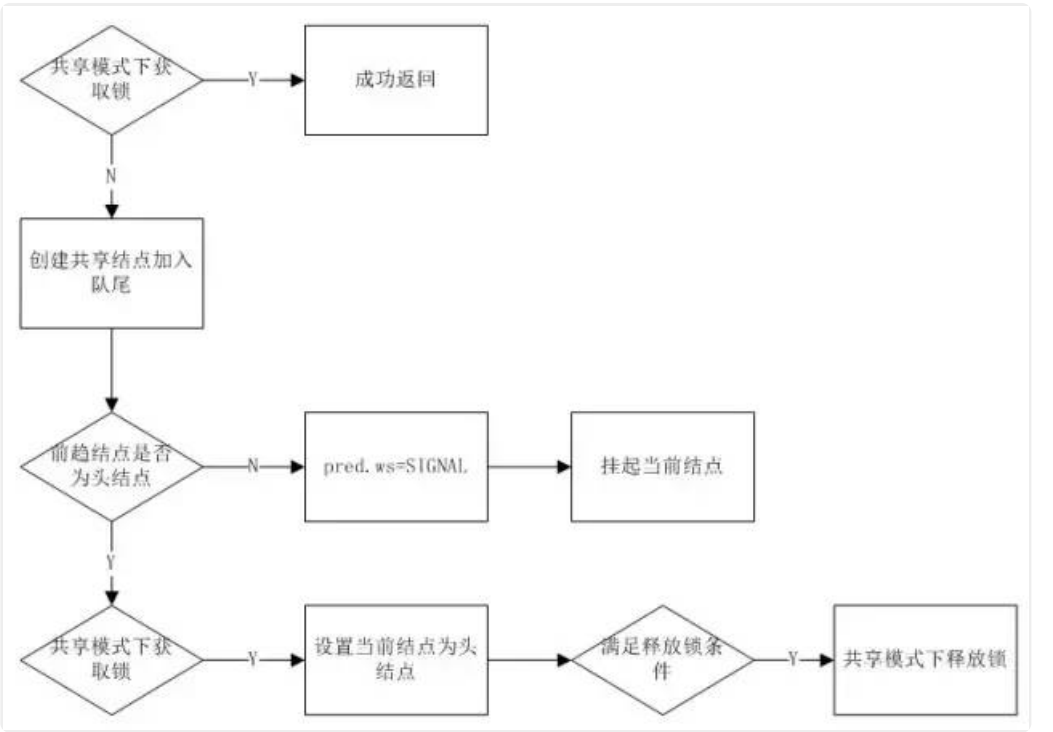


共享式：

- 共享式与独占式的区别：

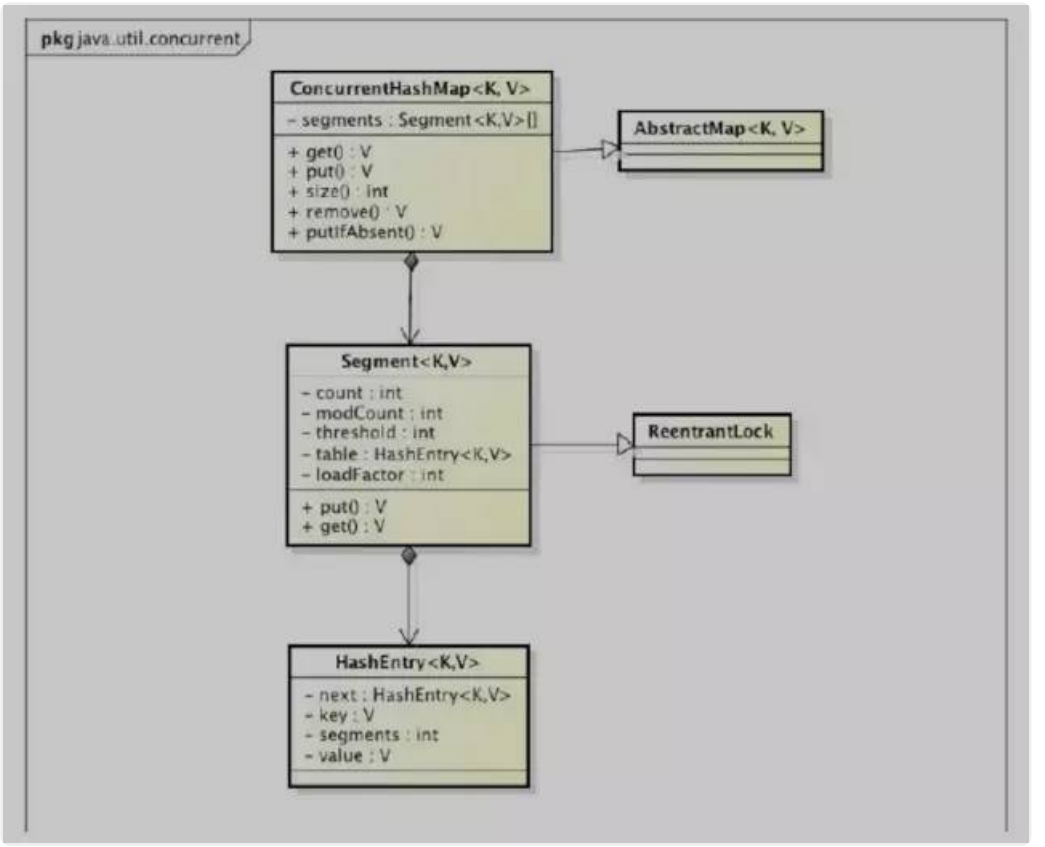


- 共享锁获取流程：

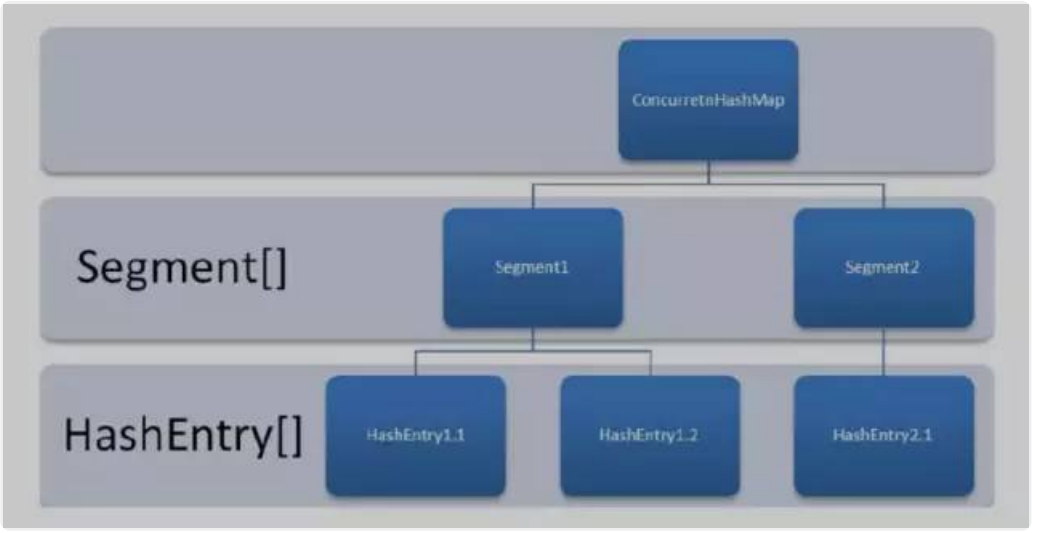


4、锁的使用用例

4.1、ConcurrentHashMap的实现原理及使用（1.7）



ConcurrentHashMap类图



ConcurrentHashMap数据结构

结论：ConcurrentHashMap使用的锁分段技术。首先将数据分成一段一段地存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

往 期 精 彩

你真的理解零拷贝吗？
关于Mybaits,我总结了 10 种通用的写法
Redis 基础、高级特性与性能调优 | 高薪必备
你的接口，真的能承受高并发吗？|文末签到福利
Redis查漏补缺：最易错过的技术要点大扫盲|文末签到福利
NIO相关基础篇
Java 大神面试经验
源码阅读技巧篇

END

>>>>>> 加群交流技术 <<<<<<

看到这里，说明你喜欢本文
你的转发，好看的人都点了好看↓↓↓

吾爱破解论坛