# 设计同步器的意义

多线程编程中，有可能会出现多个线程同时访问同一个共享、可变资源的情况，这个资源我们称之其为临界资源；这种资源可能是：对象、变量、文件等。
共享：资源可以由多个线程同时访问
可变：资源可以在其生命周期内被修改

引出的问题：
**由于线程执行的过程是不可控的，所以需要采用同步机制来协同对对象可变状态的访问**

**那么我们怎么解决线程并发安全问题？**
实际上，所有的并发模式在解决线程安全问题时，采用的方案都是 **序列化访问临界资源**。即在同一时刻，只能有一个线程访问临界资源，也称作**同步互斥访问**。

**Java 中，提供了两种方式来实现同步互斥访问：synchronized 和 Lock**

**同步器的本质就是加锁**
加锁目的：**序列化访问临界资源**，即同一时刻只能有一个线程访问临界资源(**同步互斥访问**)
不过有一点需要区别的是：当多个线程执行一个方法时，该方法内部的局部变量并不是临界资源，因为这些局部变量是在每个线程的私有栈中，因此不具有共享性，不会导致线程安全问题。

# synchronized原理详解

synchronized内置锁是一种对象锁(锁的是对象而非引用)，作用粒度是对象，可以用来实现对临界资源的同步互斥访问，是可重入的。
加锁的方式:

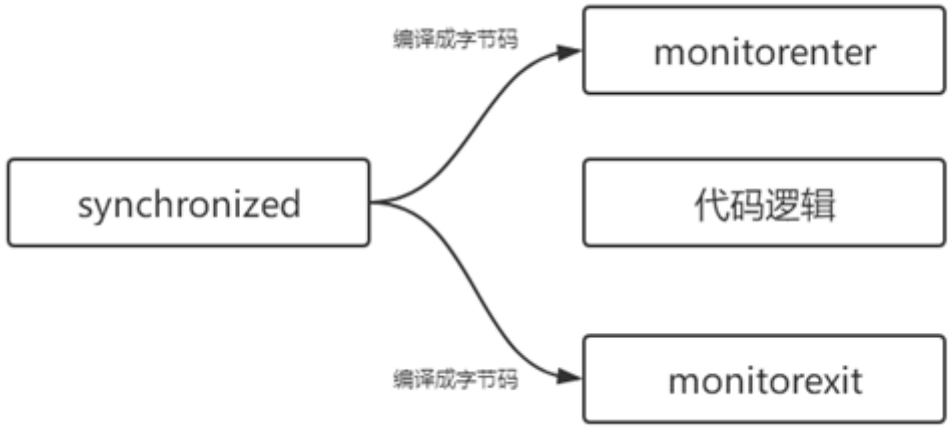
- 同步实例方法，锁是当前实例对象
- 同步类方法，锁是当前类对象

- 同步代码块，锁是括号里面的对象

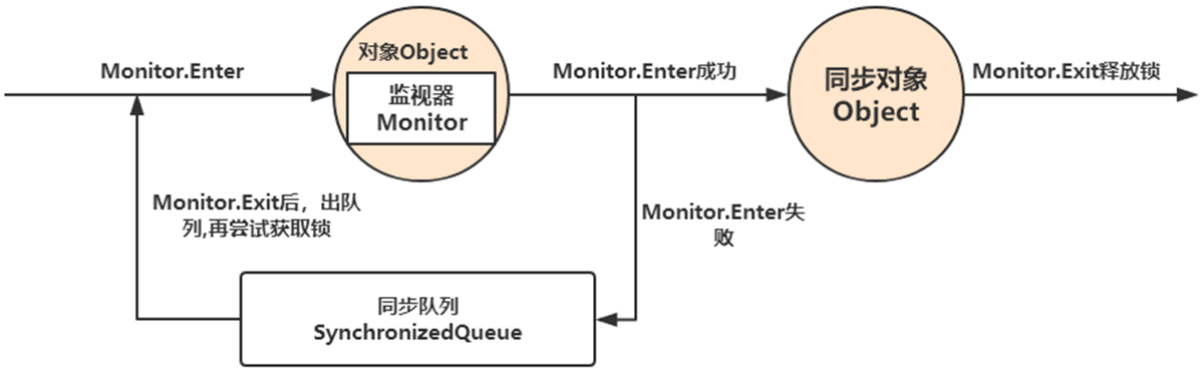
## synchronized底层原理

synchronized是基于JVM内置锁实现，通过内部对象Monitor(监视器锁)实现，基于进入与退出Monitor对象实现方法与代码块同步，监视器锁的实现依赖底层操作系统的Mutex lock（互斥锁）实现，它是一个重量级锁性能较低。当然，JVM内置锁在1.5之后版本做了重大的优化，如锁粗化（Lock Coarsening）、锁消除（Lock Elimination）、轻量级锁（Lightweight Locking）、偏向锁（Biased Locking）、适应性自旋（Adaptive Spinning）等技术来减少锁操作的开销，，内置锁的并发性能已经基本与Lock持平。

synchronized关键字被编译成字节码后会被翻译成monitorenter 和 monitorexit 两条指令分别在同步块逻辑代码的起始位置与结束位置。




每个同步对象都有一个自己的Monitor(监视器锁)，加锁过程如下图所示：



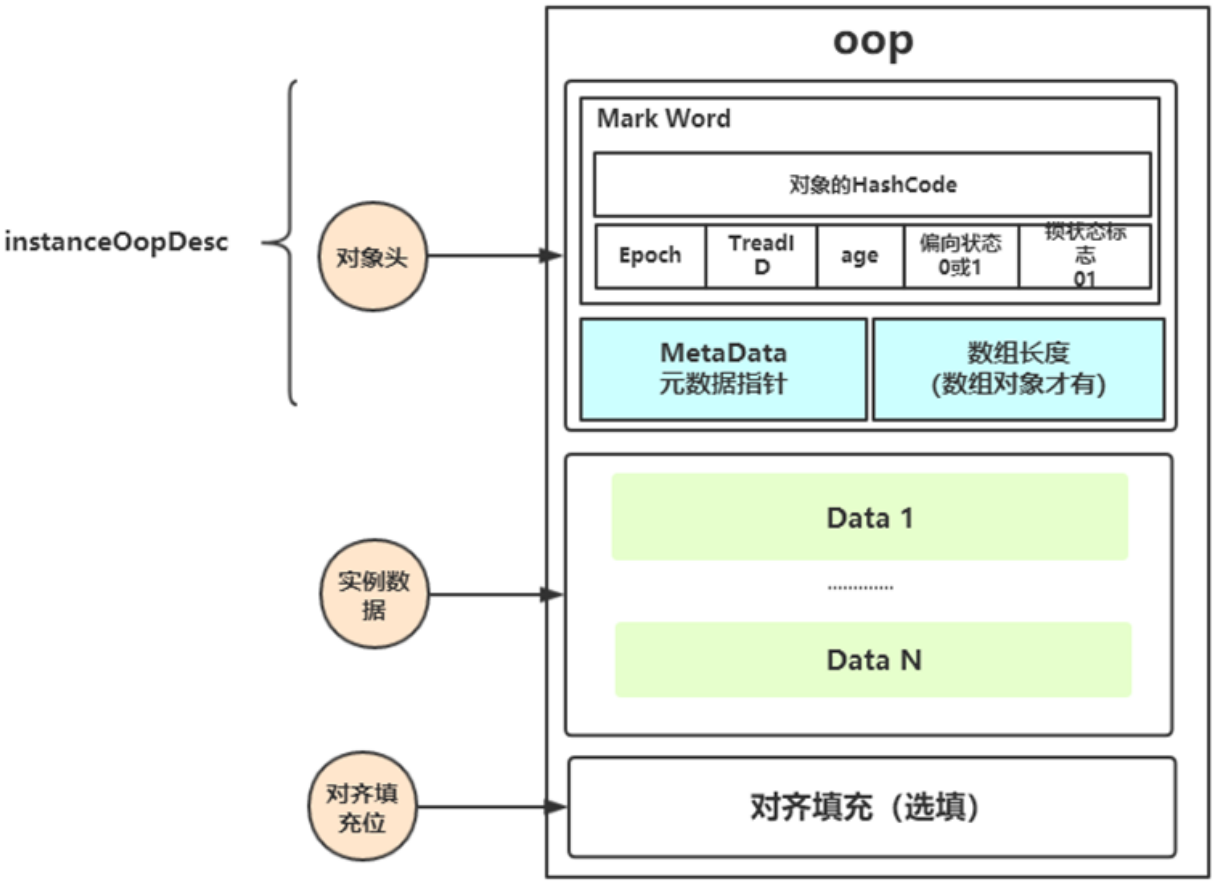

那么有个问题来了，我们知道synchronized加锁加在对象上，对象是如何记录锁状态的呢？

答案是锁状态是被记录在每个对象的对象头（Mark Word）中，下面我们一起认识一下对象的内存布局

## 对象的内存布局

HotSpot虚拟机中，对象在内存中存储的布局可以分为三块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

- 对象头：比如 hash码，对象所属的年代，对象锁，锁状态标志，偏向锁（线程）ID，偏向时间，数组长度（数组对象）等
- 实例数据：即创建对象时，对象中成员变量，方法等
- 对齐填充：对象的大小必须是8字节的整数倍




### 对象头

HotSpot虚拟机的对象头包括两部分信息，第一部分是“Mark Word”，用于存储对象自身的运行时数据， 如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等等，这部分数据的长度在32位和64位的虚拟机（暂 不考虑开启压缩指针的场景）中分别为32个和64个Bits，官方称它为“Mark Word”。对象需要存储的运行时数据很多，其实已经超出了32、64位Bitmap结构所能记录的限度，但是对象头信息是与对象自身定义的数据

无关的额 外存储成本，考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。例如在32位的HotSpot虚拟机 中对象未被锁定的状态下，Mark Word的32个Bits空间中的25Bits用于存储对象哈希码（HashCode），4Bits用于存储对象分代年龄，2Bits用于存储锁标志 位，1Bit固定为0，在其他状态（轻量级锁定、重量级锁定、GC标记、可偏向）下对象的存储内容如下表所示。

| 32位虚拟机 | | | |
|---|---|---|---|
| 25Bit | 4Bit | 1Bit | 2Bit |
| 对象的hashCode | 对象的分代年龄 | 是否是偏向锁 | 锁标志位 |

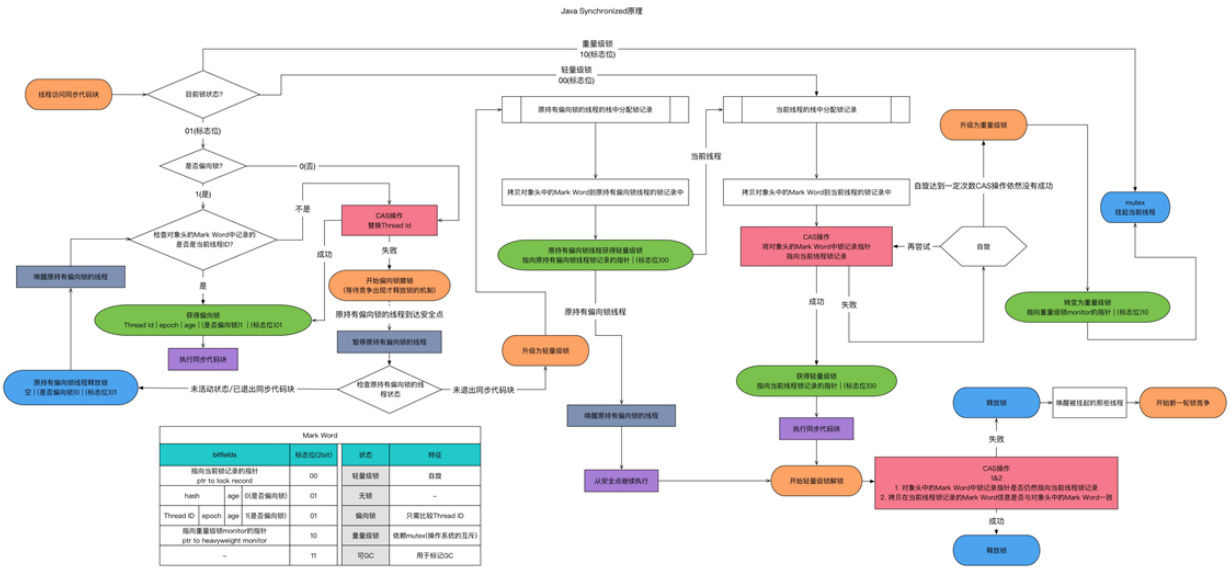
但是如果对象是数组类型，则需要三个机器码，因为JVM虚拟机可以通过Java对象的元数据信息确定Java对象的大小，但是无法从数组的元数据来确认数组的大小，所以用一块来记录数组长度。

对象头信息是与对象自身定义的数据无关的额外存储成本，但是考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存存储尽量多的数据，它会根据对象的状态复用自己的存储空间，也就是说，Mark Word会随着程序的运行发生变化，变化状态如下（32位虚拟机）：

| 锁状态 | 25 bit | | 4bit | 1bit | 2bit |
|---|---|---|---|---|---|
| | 23bit | 2bit | | 是否是偏向锁 | 锁标志位 |
| GC标记 | 空 | | | | 11 |
| 重量级锁 | 指向重量级锁Monitor的指针（依赖Mutex操作系统的互斥） | | | | 10 |
| 轻量级锁 | 指向线程栈中锁记录的指针<br>pointer to Lock Record | | | | 00 |
| 偏向锁 | 线程ID | Epoch | 对象分代年龄 | 1 | 01 |
| 无锁 | 对象的hashCode | | 对象分代年龄 | 0 | 01 |

## 锁的膨胀升级过程

锁的状态总共有四种，无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁，但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级。下图为锁的升级全过程：




偏向锁

偏向锁是Java 6之后加入的新锁，它是一种针对加锁操作的优化手段，经过研究发现，在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，因此为了减少同一线程获取锁(会涉及到一些CAS操作,耗时)的代价而引入偏向锁。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。下面我们接着了解轻量级锁。

轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)，此时Mark Word 的结构也变为轻量级锁的结构。轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

自旋锁

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

锁消除

消除锁是虚拟机另外一种锁的优化，这种优化更彻底，Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间，如下StringBuffer的append是一个同步方法，但是在add方法中的StringBuffer属于一个局部变量，并且不会被其他线程所使用，因此StringBuffer不可能存在共享资源竞争的情景，JVM会自动将其锁消除。

逃逸分析

使用逃逸分析，编译器可以对代码做如下优化:

一、同步省略。如果一个对象被发现只能从一个线程被访问到，那么对于这个对象的操作可以不考虑同步。

二、将堆分配转化为栈分配。如果一个对象在子程序中被分配，要使指向该对象的指针永远不会逃逸，对象可能是栈分配的候选，而不是堆分配。

三、分离对象或标量替换。有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分（或全部）可以不存储在内存，而是存储在CPU寄存器中。

是不是所有的对象和数组都会在堆内存分配空间?

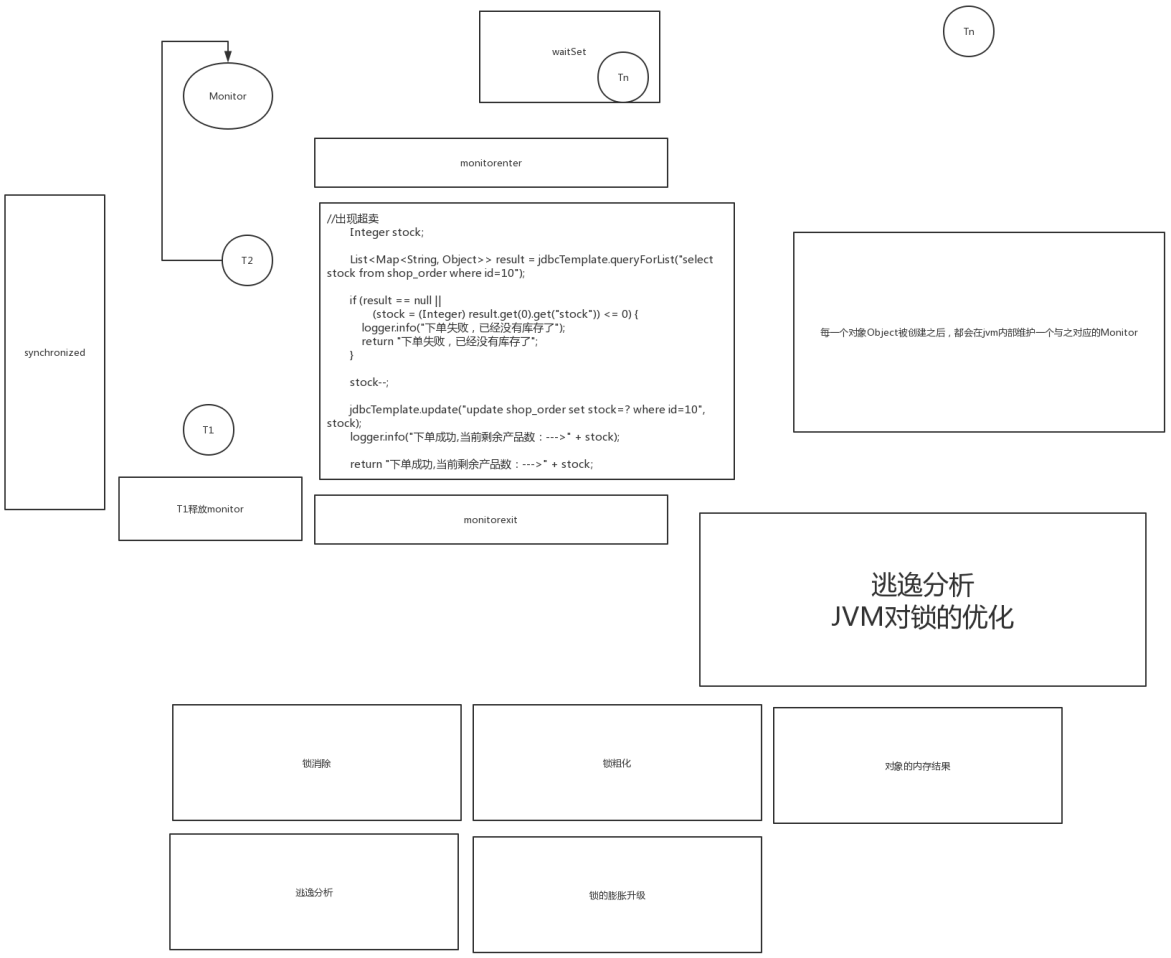
不一定

在Java代码运行时，通过JVM参数可指定是否开启逃逸分析，`-XX:+DoEscapeAnalysis`： 表示开启逃逸分析 `-XX:-DoEscapeAnalysis`： 表示关

闭逃逸分析 从jdk 1.7开始已经默认开始逃逸分析，如需关闭，需要指定`-XX:-DoEscapeAnalysis`

关于逃逸分析的案例论证见Git课程源码

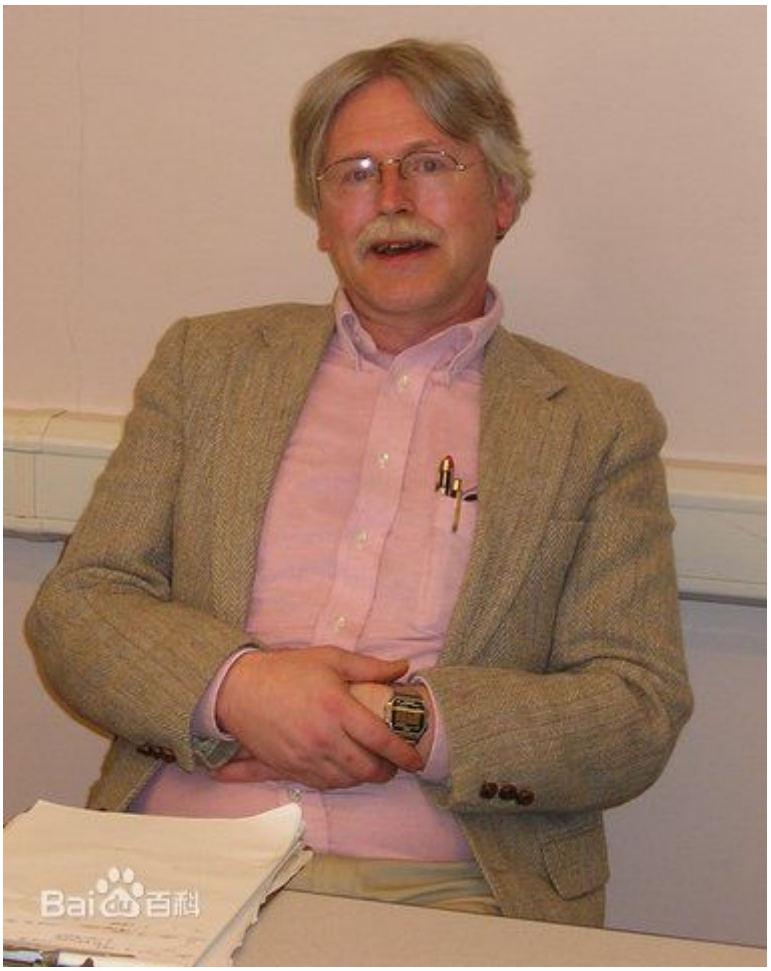
```
/**
 * 进行两种测试
 * 关闭逃逸分析，同时调大堆空间，避免堆内GC的发生，如果有GC信息将会被打印出来
 * VM运行参数: -Xmx4G -Xms4G -XX:-DoEscapeAnalysis -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError
 *
 * 开启逃逸分析
 * VM运行参数: -Xmx4G -Xms4G -XX:+DoEscapeAnalysis -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError
 *
 * 执行main方法后
 * jps 查看进程
 * jmap -histo 进程ID
 *
 */
```




# AbstractQueuedSynchronizer(AQS)

## 并发之父

*Doug Lea*




*生平不识Doug Lea，学懂并发也枉然*

Java并发编程核心在于java.concurrent.util包而juc当中的大多数同步器实现都是围绕着共同的基础行为，比如等待队列、条件队列、独占获取、共享获取等，而这个行为的抽象就是基于AbstractQueuedSynchronizer简称AQS，AQS定义了一套多线程访问共享资源的同步器框架，是一个依赖状态(state)的同步器。

AQS具备特性

- 阻塞等待队列
- 共享/独占
- 公平/非公平
- 可重入
- 允许中断

例如Java.concurrent.util当中同步器的实现如Lock,Latch,Barrier等，都是基于AQS框架实现

- 一般通过定义内部类Sync继承AQS

- 将同步器所有调用都映射到Sync对应的方法

AQS内部维护属性volatile int state (32位)

- state表示资源的可用状态

State三种访问方式

getState()、setState()、compareAndSetState()

AQS定义两种资源共享方式

- Exclusive-独占，只有一个线程能执行，如ReentrantLock
- Share-共享，多个线程可以同时执行，如Semaphore/CountDownLatch

AQS定义两种队列

- 同步等待队列
- 条件等待队列

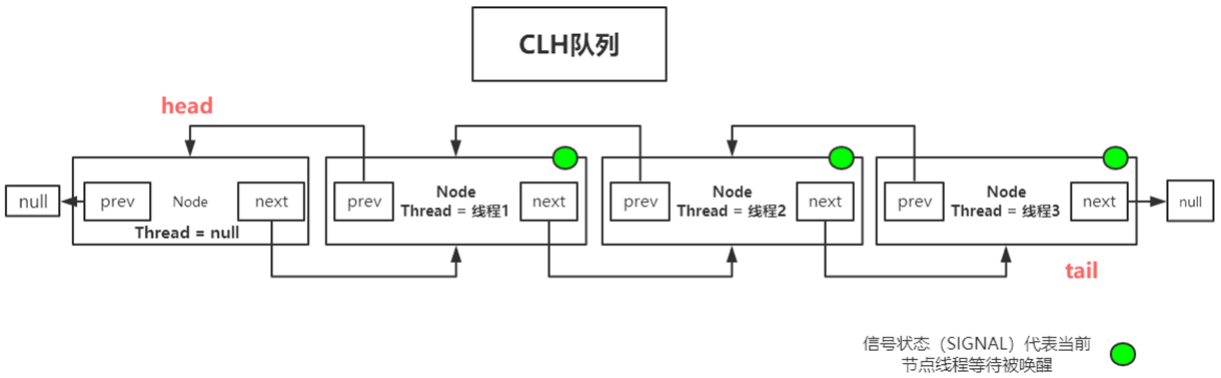
不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

- isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
- tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
- tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
- tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

## 同步等待队列

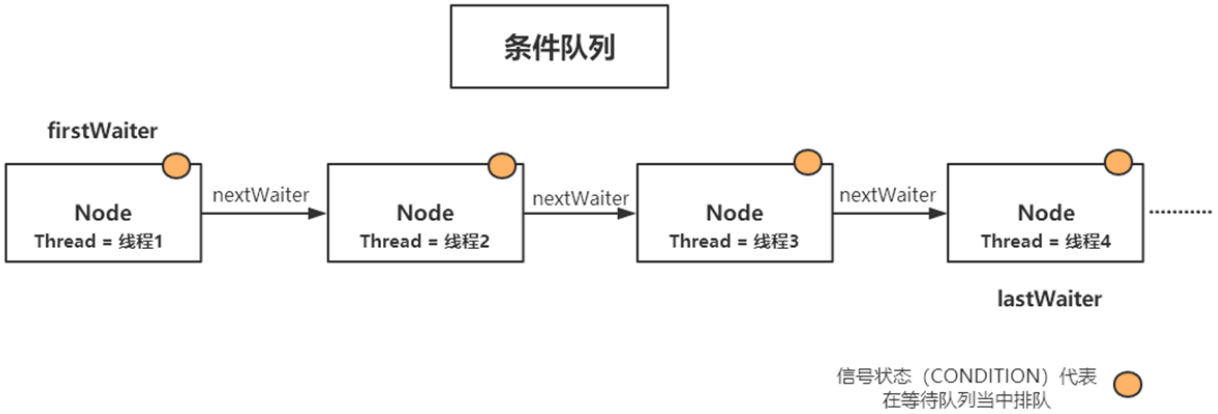
AQS当中的同步等待队列也称CLH队列，CLH队列是Craig、Landin、Hagersten三人发明的一种基于双向链表数据结构的队列，是FIFO先入先出线程

等待队列，Java中的CLH队列是原CLH队列的一个变种,线程由原自旋机制改为阻塞机制。




## 条件等待队列

Condition是一个多线程间协调通信的工具类，使得某个，或者某些线程一起等待某个条件（Condition）,只有当该条件具备时，这些等待线程才会被唤醒，从而重新争夺锁




**VIP学员注意：文档为课程的主要知识要点，尽量上课前预习，所有知识点会在课堂上深入讲解，源码部分在课堂深入分析。**