本篇主要是对Java并发中synchronized关键字进行较为深入的探索，这些知识点结合博主对synchronized的个人理解以及相关的书籍的讲解(在结尾参考资料)，如有误处，欢迎留言。

* [synchronized的三种应用方式](https://blog.csdn.net/javazejian/article/details/72828483#synchronized的三种应用方式)
  + [synchronized作用于实例方法](https://blog.csdn.net/javazejian/article/details/72828483#synchronized作用于实例方法)
  + [synchronized作用于静态方法](https://blog.csdn.net/javazejian/article/details/72828483#synchronized作用于静态方法)
  + [synchronized同步代码块](https://blog.csdn.net/javazejian/article/details/72828483#synchronized同步代码块)
* [synchronized底层语义原理](https://blog.csdn.net/javazejian/article/details/72828483#synchronized底层语义原理)
  + [理解Java对象头与Monitor](https://blog.csdn.net/javazejian/article/details/72828483#理解java对象头与monitor)
  + [synchronized代码块底层原理](https://blog.csdn.net/javazejian/article/details/72828483#synchronized代码块底层原理)
  + [synchronized方法底层原理](https://blog.csdn.net/javazejian/article/details/72828483#synchronized方法底层原理)
  + [Java虚拟机对synchronized的优化](https://blog.csdn.net/javazejian/article/details/72828483#java虚拟机对synchronized的优化)
    - [偏向锁](https://blog.csdn.net/javazejian/article/details/72828483#偏向锁)
    - [轻量级锁](https://blog.csdn.net/javazejian/article/details/72828483#轻量级锁)
    - [自旋锁](https://blog.csdn.net/javazejian/article/details/72828483#自旋锁)
    - [锁消除](https://blog.csdn.net/javazejian/article/details/72828483#锁消除)
* [关于synchronized 可能需要了解的关键点](https://blog.csdn.net/javazejian/article/details/72828483#关于synchronized-可能需要了解的关键点)
  + [synchronized的可重入性](https://blog.csdn.net/javazejian/article/details/72828483#synchronized的可重入性)
  + [线程中断与synchronized](https://blog.csdn.net/javazejian/article/details/72828483#线程中断与synchronized)
    - [线程中断](https://blog.csdn.net/javazejian/article/details/72828483#线程中断)
    - [中断与synchronized](https://blog.csdn.net/javazejian/article/details/72828483#中断与synchronized)
  + [等待唤醒机制与synchronized](https://blog.csdn.net/javazejian/article/details/72828483#等待唤醒机制与synchronized)

线程安全是并发编程中的重要关注点，应该注意到的是，造成线程安全问题的主要诱因有两点，一是**存在共享数据(也称临界资源)**，二是存在多条线程**共同操作共享数据**。因此为了解决这个问题，我们可能需要这样一个方案，当存在多个线程操作共享数据时，需要保证**同一时刻有且只有一个线程在操作共享数据**，其他线程必须等到该线程处理完数据后再进行，这种方式有个高尚的名称叫**互斥锁**，即能达到互斥访问目的的锁，也就是说当一个共享数据被当前正在访问的线程加上互斥锁后，在同一个时刻，其他线程只能处于等待的状态，直到当前线程处理完毕释放该锁。在 Java 中，**关键字 synchronized可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块**(主要是对方法或者代码块中存在共享数据的操作)，同时我们还应该注意到synchronized另外一个重要的作用，synchronized可保证一个线程的变化(主要是共享数据的变化)被其他线程所看到**（保证可见性，完全可以替代Volatile功能）**，这点确实也是很重要的。

**synchronized的三种应用方式**

synchronized关键字最主要有以下3种应用方式，下面分别介绍

* **修饰实例方法**，作用于**当前实例加锁**，进入同步代码前要获得当前实例的锁
* **修饰静态方法**，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
* **修饰代码块**，指定加锁对象，对**给定对象加锁**，进入同步代码库前要获得给定对象的锁。

**synchronized作用于实例方法**

所谓的实例对象锁就是用**synchronized修饰实例对象中的实例方法**，注意是实例方法不包括静态方法，如下

public class AccountingSync implements Runnable{

//共享资源(临界资源)

static int i=0;

**/\*\***

**\* synchronized 修饰实例方法**

**\*/**

public synchronized void increase(){

i++;

}

@Override

public void run() {

for(int j=0;j<1000000;j++){

increase();

}

}

public static void main(String[] args) throws InterruptedException {

AccountingSync instance=new AccountingSync();

Thread t1=new Thread(instance);

Thread t2=new Thread(instance);

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println(i);

}

/\*\*

\* 输出结果:

\* 2000000

\*/

}

上述代码中，我们开启两个线程操作同一个**共享资源即变量i**，由于**i++;操作并不具备原子性**，该操作是先读取值，然后写回一个新值，相当于原来的值加上1，分两步完成，如果第二个线程在第一个线程读取旧值和写回新值期间读取i的域值，那么第二个线程就会与第一个线程一起看到同一个值，并执行相同值的加1操作，这也就造成了**线程安全失败**，因此对于**increase方法必须使用synchronized修饰，以便保证线程安全**。此时我们应该注意到**synchronized修饰的是实例方法increase**，在这样的情况下，**当前线程的锁便是实例对象instance**，注意Java中的线程同步锁可以是任意对象。从代码执行结果来看确实是正确的，倘若我们没有使用synchronized关键字，其最终输出结果就很可能小于2000000，这便是synchronized关键字的作用。这里我们还需要意识到，当**一个线程正在访问一个对象的 synchronized 实例方法，那么其他线程不能访问该对象的其他 synchronized 方法**，毕竟一个对象只有一把锁，当一个线程获取了该对象的锁之后，其他线程无法获取该对象的锁，**所以无法访问该对象的其他synchronized实例方法**，但是其他线程还是可以访问该实例对象的其他非synchronized方法，当然如果是一个线程 A 需要访问实例对象 obj1 的 synchronized 方法 f1(当前对象锁是obj1)，另一个线程 B 需要访问实例对象 obj2 的 synchronized 方法 f2(当前对象锁是obj2)，这样是允许的，因为两个实例对象锁并不同相同，此时如果两个线程操作数据并非共享的，线程安全是有保障的，遗憾的是如果两个线程操作的是共享数据，那么线程安全就有可能无法保证了，如下代码将演示出该现象

public class AccountingSyncBad implements Runnable{

static int i=0;//类变量

public synchronized void increase(){

i++;

}

@Override

public void run() {

for(int j=0;j<1000000;j++){

increase();

}

}

public static void main(String[] args) throws InterruptedException {

//new新实例

Thread t1=new Thread(new AccountingSyncBad());

//new新实例

Thread t2=new Thread(new AccountingSyncBad());

t1.start();

t2.start();

//join含义:当前线程A等待thread线程终止之后才能从thread.join()返回

t1.join();

t2.join();

System.out.println(i);

}

}

上述代码与前面不同的是我们同时创建了两个新实例AccountingSyncBad，然后启动两个不同的线程对共享变量i进行操作，但很遗憾操作结果是1452317而不是期望结果2000000，因为上述代码犯了严重的错误，虽然我们使用synchronized修饰了increase方法，但却**new了两个不同的实例对象，这也就意味着存在着两个不同的实例对象锁，因此t1和t2都会进入各自的对象锁，也就是说t1和t2线程使用的是不同的锁，因此线程安全是无法保证的**。解决这种困境的的方式是将synchronized作用于静态的increase方法，这样的话，对象锁就当前类对象，由于无论创建多少个实例对象，但**对于的类对象拥有只有一个，所有在这样的情况下对象锁就是唯一的**。下面我们看看如何使用将synchronized作用于静态的increase方法。

## synchronized作用于静态方法

当**synchronized作用于静态方法**时，其**锁就是当前类的class对象锁**。由于静态成员不专属于任何一个实例对象，**是类成员**，因此通过class对象锁可以控制静态 成员的并发操作。需要注意的是如果一个线程A调用一个实例对象的非static synchronized方法，而线程B需要调用这个实例对象所属类的静态 synchronized方法，是允许的，不会发生互斥现象，因为**访问静态 synchronized 方法占用的锁是当前类的class对象**，而访问非静态 synchronized 方法占用的锁是当前实例对象锁，看如下代码

public class AccountingSyncClass implements Runnable{

static int i=0;

/\*\*

\* 作用于静态方法,锁是当前class对象,也就是

\* AccountingSyncClass类对应的class对象

\*/

**public static synchronized void increase()**{

i++;

}

/\*\*

\* 非静态,访问时锁不一样不会发生互斥

\*/

**public synchronized void increase4Obj()**{

i++;

}

@Override

public void run() {

for(int j=0;j<1000000;j++){

increase();

increase**4Obj** ();

}

}

public static void main(String[] args) throws InterruptedException {

//new新实例

Thread t1=new Thread(new AccountingSyncClass());

//new心事了

Thread t2=new Thread(new AccountingSyncClass());

//启动线程

t1.start();t2.start();

t1.join();t2.join();

System.out.println(i);

}

}

由于synchronized关键字修饰的是静态increase方法，与修饰实例方法不同的是，其锁对象是当前类的class对象。注意代码中的increase4Obj方法是实例方法，其对象锁是当前实例对象，如果别的线程调用该方法，将不会产生互斥现象，毕竟**锁对象不同**，但我们应该意识到这种情况下可能会发现线程安全问题(操作了共享静态变量i)。**(非静态方法的锁是归于实例对象，静态方法的锁是归于类对象，即操作同一个类所有对象都是共用这把锁的。)**

**所以i最后会小于20000.**

## synchronized同步代码块

除了使用关键字修饰实例方法和静态方法外，还可以使用同步代码块，在某些情况下，我们编写的方法体可能比较大，同时存在一些比较耗时的操作，而需要同步的代码又只有一小部分，如果直接对整个方法进行同步操作，可能会得不偿失，此时我们可以**使用同步代码块的方式对需要同步的代码进行包裹**，这样就无需对整个方法进行同步操作了，同步代码块的使用示例如下：

public class AccountingSync implements Runnable{

**static AccountingSync instance=new AccountingSync();**

static int i=0;

@Override

public void run() {

**//省略其他耗时操作....**

**//使用同步代码块对变量i进行同步操作,锁对象为instance**

**synchronized(instance){**

**for(int j=0;j<1000000;j++){**

**i++;**

**}**

**}**

}

public static void main(String[] args) throws InterruptedException {

Thread t1=new Thread(instance);

Thread t2=new Thread(instance);

t1.start();t2.start();

t1.join();t2.join();

System.out.println(i);

}

}

从代码看出，**将synchronized作用于一个给定的实例对象instance（静态的类对象）**，即当前实例对象就是锁对象，每次当线程进入synchronized包裹的代码块时就会要求当前线程持有**instance实例对象锁**，如果当前有其他线程正持有该对象锁，那么新到的线程就必须等待，这样也就保证了每次只有一个线程执行i++;操作。当然**除了instance作为对象外，我们还可以使用this对象(代表当前实例)或者当前类的class对象作为锁**，如下代码：

//this,当前实例对象锁

synchronized(this){

for(int j=0;j<1000000;j++){

i++;

}

}

//class对象锁

synchronized(AccountingSync.class){

for(int j=0;j<1000000;j++){

i++;

}

}

**以上方法经过实验后，**如果是new一个新的类对象，则生成的的是什么对象就锁住啥。

**锁住**this的话， 就跟第一种情况差不多，锁是属于各自的实例对象，锁住AccountingSync.class，则锁住了唯一的类对象。

**了解完synchronized的基本含义及其使用方式后，下面我们将进一步深入理解synchronized的底层实现原理。**

**synchronized底层语义原理**

Java 虚拟机中的同步(Synchronization)基于进入和退出管程(Monitor)对象实现， 无论是显式同步(有明确的 monitorenter 和 monitorexit 指令,即同步代码块)还是隐式同步都是如此。在 Java 语言中，**同步用的最多的地方可能是被 synchronized 修饰的同步方法**。同步方法 并不是由 monitorenter 和 monitorexit 指令来实现同步的，而是由方法调用指令读取运行时常量池中方法的 ACC\_SYNCHRONIZED 标志来隐式实现的，关于这点，稍后详细分析。

下面先来了解一个概念Java对象头，这对深入理解synchronized实现原理非常关键。

**理解Java对象头与Monitor**

在JVM中，对象在内存中的布局分为三块区域：对象头、实例数据和对齐填充。如下：



* **实例变量：**存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐。
* **填充数据：**由于虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐，这点了解即可。

而对于顶部，则是Java头对象，它**实现synchronized的锁对象的基础**，这点我们重点分析它，一般而言，**synchronized使用的锁对象是存储在Java对象头里**的，jvm中采用**2个字来存储对象头**(如果对象是数组则会分配3个字，多出来的1个字记录的是数组长度)，其主要结构是由Mark Word 和 Class Metadata Address 组成，其结构说明如下表：

| **虚拟机位数** | **头对象结构** | **说明** |
| --- | --- | --- |
| 32/64bit | Mark Word | **存储对象的hashCode、锁信息或分代年龄或GC标志等信息** |
| 32/64bit | Class Metadata Address | 类型指针指向对象的类元数据，JVM通过这个指针确定该对象是哪个类的实例。 |

其中Mark Word在默认情况下存储着对象的HashCode、分代年龄、锁标记位等以下是32位JVM的Mark Word默认存储结构

其中轻量级锁和偏向锁是Java 6 对 synchronized 锁进行优化后新增加的，稍后我们会简要分析。这里我们主要**分析一下重量级锁也就是通常说synchronized的对象锁**，锁标识位为10，其中指针指向的是monitor对象（也称为管程或监视器锁）的起始地址。每个对象都存在着一个 monitor 与之关联，对象与其 monitor 之间的关系有**存在多种实现方式**，如**monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个 monitor 被某个线程持有后，它便处于锁定状态**。在Java虚拟机(HotSpot)中，monitor是由**ObjectMonitor**实现的，其主要数据结构如下（位于HotSpot虚拟机源码ObjectMonitor.hpp文件，C++实现的）

ObjectMonitor() {

\_header = NULL;

\_count = 0; //记录个数

\_waiters = 0,

\_recursions = 0;

\_object = NULL;

\_owner = NULL;

\_WaitSet = NULL; //处于wait状态的线程，会被加入到\_WaitSet

\_WaitSetLock = 0 ;

\_Responsible = NULL ;

\_succ = NULL ;

\_cxq = NULL ;

FreeNext = NULL ;

\_EntryList = NULL ; //处于等待锁block状态的线程，会被加入到该列表

\_SpinFreq = 0 ;

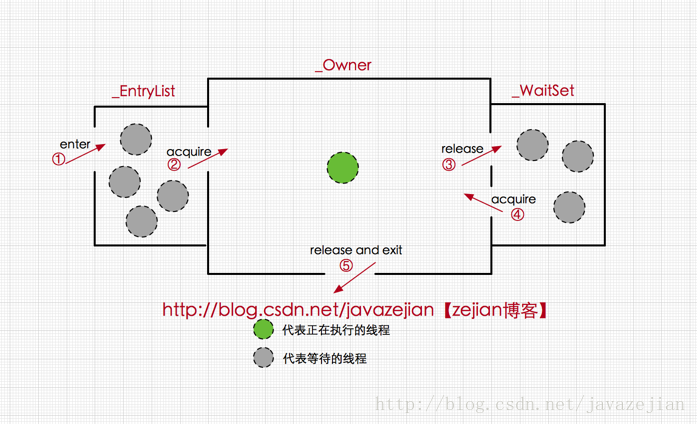
\_SpinClock = 0 ;

OwnerIsThread = 0 ;

}

**ObjectMonitor中有两个队列**，\_WaitSet**(等待队列)** 和 \_EntryList（**锁池）**，用来保存ObjectWaiter对象列表( **每个等待锁的线程都会被封装成ObjectWaiter对象**)，\_owner指向持有ObjectMonitor对象的线程，当多个线程同时访问一段同步代码时，**首先会进入 \_EntryList 集合**，当线程获取到对象的monitor 后进入 **\_Owner 区域**（正在执行状态）**并把monitor中的owner变量设置为当前线程**同时**monitor中的计数器count加1，**若线程调用 wait() 方法，将释放当前持有的monitor，**owner变量恢复为null**，count自减1，同时该线程进入 WaitSe t集合中等待被唤醒。

若当前线程执行完毕也将释放monitor(锁)并复位变量的值，唤醒其他在等待池里的线程，让其他线程进入获取monitor(锁)。如下图所示



由此看来，**monitor对象存在于每个Java对象的对象头中(存储的指针的指向)，**synchronized锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因，同时也是notify/notifyAll/wait等方法存在于顶级对象Object中的原因(关于这点稍后还会进行分析)，ok~，有了上述知识基础后，下面我们将进一步分析synchronized在字节码层面的具体语义实现。

### 说一下Monitor

什么是Monitor？我们可以把它理解为一个同步工具，也可以描述为一种同步机制，它通常被描述为一个对象。  
与一切皆对象一样，所有的Java对象是天生的Monitor，每一个Java对象都有成为Monitor的潜质，因为在Java的设计中 ，每一个Java对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者Monitor锁。  
Monitor 是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局的可用列表。**每一个被锁住的对象都会和一个monitor关联（对象头的MarkWord中的LockWord指向monitor的起始地址）**，同时monitor中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。其结构如下：  
[](http://cmsblogs.qiniudn.com/wp-content/uploads/2017/02/44444-1.png)  
Owner：初始时为NULL表示当前没有任何线程拥有该monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为NULL；  
**EntryQ:**关联一个系统互斥锁（semaphore），阻塞所有试图锁住monitor record失败的线程。**(锁池)**  
RcThis:表示blocked或waiting在该monitor record上的所有线程的个数。  
Nest:**用来实现重入锁的计数。**  
HashCode:保存从对象头拷贝过来的HashCode值（可能还包含GC age）。  
Candidate:用来避免不必要的阻塞或等待线程唤醒，因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换（从阻塞到就绪然后因为竞争锁失败又被阻塞）从而导致性能严重下降。Candidate只有两种可能的值0表示没有需要唤醒的线程1表示要唤醒一个继任线程来竞争锁。

## synchronized代码块底层原理

现在我们重新定义一个synchronized修饰的同步代码块，在代码块中操作共享变量i，如下

public class SyncCodeBlock {

public int i;

public void syncTask(){

//同步代码库

synchronized (this){

i++;

}

}

}

编译上述代码并使用javap反编译后得到字节码如下(这里我们省略一部分没有必要的信息)：

Classfile /Users/zejian/Downloads/Java8\_Action/src/main/java/com/zejian/concurrencys/SyncCodeBlock.class

Last modified 2017-6-2; size 426 bytes

MD5 checksum c80bc322c87b312de760942820b4fed5

Compiled from "SyncCodeBlock.java"

public class com.zejian.concurrencys.SyncCodeBlock

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

//........省略常量池中数据

//构造函数

public com.zejian.concurrencys.SyncCodeBlock();

descriptor: ()V

flags: ACC\_PUBLIC

Code:

stack=1, locals=1, args\_size=1

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: return

LineNumberTable:

line 7: 0

//===========主要看看syncTask方法实现================

public void syncTask();

descriptor: ()V

flags: ACC\_PUBLIC

Code:

stack=3, locals=3, args\_size=1

0: aload\_0

1: dup

2: astore\_1

3: monitorenter //注意此处，进入同步方法

4: aload\_0

5: dup

6: getfield #2 // Field i:I

9: iconst\_1

10: iadd

11: putfield #2 // Field i:I

14: aload\_1

15: monitorexit //注意此处，退出同步方法

16: goto 24

19: astore\_2

20: aload\_1

21: monitorexit //注意此处，退出同步方法

22: aload\_2

23: athrow

24: return

Exception table:

//省略其他字节码.......

}

SourceFile: "SyncCodeBlock.java"

我们主要关注字节码中的如下代码

3: monitorenter //进入同步方法

//..........省略其他

15: monitorexit //退出同步方法

16: goto 24

//省略其他.......

21: monitorexit //退出同步方法

从字节码中可知同步语句块的实现使用的是monitorenter 和 monitorexit 指令，其中monitorenter指令指向同步代码块的开始位置，monitorexit指令则指明同步代码块的结束位置，当执行monitorenter指令时，当前线程将试图获取 objectref(即对象锁) 所对应的 monitor 的持有权，当 **objectref 的 monitor 的进入计数器为 0，那线程可以成功取得 monitor，并将计数器值设置为 1，取锁成功**。如果当前线程已经拥有 objectref 的 monitor 的持有权，那它可以重入这个 monitor (关于重入性稍后会分析)，重入时计数器的值也会加 1。倘若其他线程已经拥有 objectref 的 monitor 的所有权，那当前线程将被阻塞，直到正在执行线程执行完毕，即monitorexit指令被执行，**执行线程将释放 monitor(锁)并设置计数器值为0 ，其他线程将有机会持有 monitor** 。值得注意的是编译器将会确保无论方法通过何种方式完成，方法中调用过的每条 monitorenter 指令都有执行其对应 monitorexit 指令，而无论这个方法是正常结束还是异常结束。为了保证在方法异常完成时 monitorenter 和 monitorexit 指令依然可以正确配对执行，编译器会自动产生一个异常处理器，这个异常处理器声明可处理所有的异常，它的目的就是用来执行 monitorexit 指令。从字节码中也可以看出多了一个monitorexit指令，它就是异常结束时被执行的释放monitor 的指令。

## synchronized方法底层原理

**方法级的同步是隐式，即无需通过字节码指令来控制的，它实现在方法调用和返回操作之中。**JVM可以从方法常量池中的方法表结构(method\_info Structure) 中的 ACC\_SYNCHRONIZED 访问**标志区分一个方法是否同步方法**。当方法调用时，调用指令将会 检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程**将先持有monitor**（虚拟机规范中用的是管程一词）， **然后再执行方法**，最后再方法完成(无论是正常完成还是非正常完成)时释放monitor。在方法执行期间，执行线程持有了monitor，其他任何线程都无法再获得同一个monitor。如果**一个同步方法执行期间抛 出了异常**，并且在方法内部无法处理此异常，那这个同步方法所持有的monitor将在异常抛到同步方法之外时自动释放。下面我们看看字节码层面如何实现：

public class SyncMethod {

public int i;

public synchronized void syncTask(){

i++;

}

}

使用javap反编译后的字节码如下：

Classfile /Users/zejian/Downloads/Java8\_Action/src/main/java/com/zejian/concurrencys/SyncMethod.class

Last modified 2017-6-2; size 308 bytes

MD5 checksum f34075a8c059ea65e4cc2fa610e0cd94

Compiled from "SyncMethod.java"

public class com.zejian.concurrencys.SyncMethod

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool;

//省略没必要的字节码

//==================syncTask方法======================

public synchronized void syncTask();

descriptor: ()V

//方法标识ACC\_PUBLIC代表public修饰，ACC\_SYNCHRONIZED指明该方法为同步方法

flags: ACC\_PUBLIC, ACC\_SYNCHRONIZED

Code:

stack=3, locals=1, args\_size=1

0: aload\_0

1: dup

2: getfield #2 // Field i:I

5: iconst\_1

6: iadd

7: putfield #2 // Field i:I

10: return

LineNumberTable:

line 12: 0

line 13: 10

}

SourceFile: "SyncMethod.java"

从字节码中可以看出，synchronized**修饰的方法并没有monitorenter指令和monitorexit指令，**取得代之的确实是ACC\_SYNCHRONIZED标识，该标识指明了该方法是一个同步方法，JVM通过**该ACC\_SYNCHRONIZED访问标志来辨别一个方法是否声明为同步方法**，从而执行相应的同步调用。这便是synchronized锁在同步代码块和同步方法上实现的基本原理。同时我们还必须注意到的是在Java早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的Mutex Lock来实现的，而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，这也是为什么早期的synchronized效率低的原因。庆幸的是在Java 6之后Java官方对从JVM层面对synchronized较大优化，所以现在的synchronized锁效率也优化得很不错了，Java 6之后，为了减少获得锁和释放锁所带来的性能消耗，引入了轻量级锁和偏向锁，接下来我们将简单了解一下Java官方在JVM层面对synchronized锁的优化。

## Java虚拟机对synchronized的优化

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，关于重量级锁，前面我们已详细分析过，下面我们将介绍**偏向锁和轻量级锁以及JVM的其他优化手段**，这里并不打算深入到每个锁的实现和转换过程更多地是阐述Java虚拟机所提供的每个锁的核心优化思想，毕竟涉及到具体过程比较繁琐，如需了解详细过程可以查阅《深入理解Java虚拟机原理》。

### 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**（锁标志会改变）。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，**但是比如i++，这个i是全局变量共享资源。那么该变量所对应的方法的锁则不会被消除。**

/\*\*

\* Created by zejian on 2017/6/4.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\* 消除StringBuffer同步锁

\*/

public class StringBufferRemoveSync {

public void add(String str1, String str2) {

//StringBuffer是线程安全,由于sb只会在append方法中使用,不可能被其他线程引用

//因此sb属于不可能共享的资源,JVM会自动消除内部的锁

StringBuffer sb = new StringBuffer();

sb.append(str1).append(str2);

}

public static void main(String[] args) {

StringBufferRemoveSync rmsync = new StringBufferRemoveSync();

for (int i = 0; i < 10000000; i++) {

rmsync.add("abc", "123");

}

}

}

# 关于synchronized 可能需要了解的关键点

## synchronized的可重入性

从互斥锁的设计上来说，当一个线程试图操作一个由其他线程持有的对象锁的临界资源时**，将会处于阻塞状态（进入锁池等待取得锁），**但当一个线程**再次请求自己持有对象锁**的临界资源时，这种情况属于**重入锁**，请求将会成功，在java中synchronized是基于原子性的内部锁机制，是可重入的，因此在一个线程调用synchronized方法的同时在其方法体内部调用该对象另一个synchronized方法，也就是说**一个线程得到一个对象锁后再次请求该对象锁，是允许的**，这就是synchronized的可重入性。如下：

public class AccountingSync implements Runnable{

static AccountingSync instance=new AccountingSync();

static int i=0;

static int j=0;

@Override

public void run() {

for(int j=0;j<1000000;j++){

//this,当前实例对象锁

synchronized(this){

i++;

increase();//synchronized的可重入性

}

}

}

public synchronized void increase(){

j++;

}

public static void main(String[] args) throws InterruptedException {

Thread t1=new Thread(instance);

Thread t2=new Thread(instance);

t1.start();t2.start();

t1.join();t2.join();

System.out.println(i);

}

}

正如代码所演示的，在获取当前实例对象锁后进入synchronized代码块执行同步代码，并在代码块中调用了当前实例对象的另外一个synchronized方法，再次请求当前实例锁时，将被允许，进而执行方法体代码，这就是重入锁最直接的体现，需要特别注意另外一种情况，当子类继承父类时，子类也是可以通过可重入锁调用父类的同步方法。**注意由于synchronized是基于monitor实现的，因此每次重入，monitor中的计数器仍会加1。**

## 线程中断与synchronized（在锁池，正在运行中的线程无法直接被interrupt方法中断，而wait（），sleep（）和join（）等则可以）

### 线程中断

正如中断二字所表达的意义，在线程**运行(run方法)中间打断它**，在Java中，**提供了以下3个有关线程中断的方法**

//中断线程（实例方法）

public void Thread.interrupt();

//判断线程是否被中断（实例方法）

public boolean Thread.isInterrupted();

//判断是否被中断并清除当前中断状态（静态方法）

public static boolean Thread.interrupted();

当一个线程**处于被阻塞状态或者试图执行一个阻塞操作时，使用Thread.interrupt()方式中断该线程，注意此时将会抛出一个InterruptedException的异常，**同时中断状态将会被复位(由中断状态改为非中断状态)，如下代码将演示该过程：

public class InterruputSleepThread3 {

public static void main(String[] args) throws InterruptedException {

Thread t1 = new Thread() {

@Override

public void run() {

//while在try中，通过异常中断就可以退出run循环

try {

while (true) {

//当前线程处于阻塞状态，异常必须捕捉处理，无法往外抛出

TimeUnit.SECONDS.sleep(2);

}

} catch (InterruptedException e) {

System.out.println("Interruted When Sleep");

boolean interrupt = this.isInterrupted();

//中断状态被复位，**false**

System.out.println("interrupt:"+interrupt);

}

}

};

t1.start();

TimeUnit.SECONDS.sleep(2);

//中断处于阻塞状态的线程

t1.interrupt();

/\*\*

\* 输出结果:

Interruted When Sleep

interrupt:false

\*/

}

}

如上述代码所示，我们创建一个线程，并在线程中调用了sleep方法从而使用线程进入阻塞状态，启动线程后，**调用线程实例对象的interrupt方法中断阻塞异常，并抛出InterruptedException异常，此时中断状态也将被复位**。这里有些人可能会诧异，为什么不用Thread.sleep(2000);而是用TimeUnit.SECONDS.sleep(2);其实原因很简单，前者使用时并没有明确的单位说明，而后者非常明确表达秒的单位，事实上后者的内部实现最终还是调用了Thread.sleep(2000);，但为了编写的代码语义更清晰，建议使用TimeUnit.SECONDS.sleep(2);的方式，注意TimeUnit是个枚举类型。ok~，除了阻塞中断的情景，我们还可能会遇到**处于运行期且非阻塞的状态的线程，这种情况下，直接调用Thread.interrupt()中断线程是不会得到任响应的**，如下代码，将无法中断非阻塞状态下的线程：

public class InterruputThread {

public static void main(String[] args) throws InterruptedException {

Thread t1=new Thread(){

@Override

public void run(){

while(true){

System.out.println("未被中断");

}

}

};

t1.start();

TimeUnit.SECONDS.sleep(2);

t1.interrupt();

/\*\*

\* 输出结果(无限执行):

未被中断

未被中断

未被中断

......

\*/

}

}

虽然我们调用了interrupt方法，**但线程t1并未被中断，因为处于非阻塞状态的线程需要我们手动进行中断检测并结束程序，**改进后代码如下：

public class InterruputThread {

public static void main(String[] args) throws InterruptedException {

Thread t1=new Thread(){

@Override

public void run(){

while(true){

//判断当前线程是否被中断

if (this.isInterrupted()){

System.out.println("线程中断");

break;

}

}

System.out.println("已跳出循环,线程中断!");

}

};

t1.start();

TimeUnit.SECONDS.sleep(2);

t1.interrupt();

/\*\*

\* 输出结果:

线程中断

已跳出循环,线程中断!

\*/

}

}

是的，我们在代码中使用了**实例方法isInterrupted判断线程是否已被中断**，如果被中断将跳出循环以此结束线程,注意非阻塞状态调用interrupt()并不会导致中断状态重置。综合所述，可以简单总结一下中断两种情况，**一种是当线程处于阻塞状态或者试图执行一个阻塞操作时，我们可以使用实例方法interrupt()进行线程中断，执行中断操作后将会抛出interruptException异常(该异常必须捕捉无法向外抛出)并将中断状态复位，另外一种是当线程处于运行状态时，我们也可调用实例方法interrupt()进行线程中断，但同时必须手动判断中断状态，并编写中断线程的代码(其实就是结束run方法体的代码)。**有时我们在编码时可能需要兼顾以上两种情况，那么就可以如下编写：

public void run(){

try {

//判断当前线程是否已中断,注意interrupted方法是静态的,执行后会对中断状态进行复位

while (!Thread.interrupted()) {

TimeUnit.SECONDS.sleep(2);

}

} catch (InterruptedException e) {

}

}

### 中断与synchronized

事实上线程的中断操作对于正在等待获取的锁对象的synchronized方法或者代码块并不起作用，也就是对于synchronized来说，如果一个线程在等待锁（锁池），那么结果只有两种，**要么它获得这把锁继续执行，要么它就保存等待，即使调用中断线程的方法，也不会生效**。演示代码如下

/\*\*

\* Created by zejian on 2017/6/2.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

public class SynchronizedBlocked implements Runnable{

public synchronized void f() {

System.out.println("Trying to call f()");

while(true) // Never releases lock

Thread.yield();

}

/\*\*

\* 在构造器中创建新线程并启动获取对象锁

\*/

public SynchronizedBlocked() {

//该线程已持有当前实例锁

new Thread() {

public void run() {

f(); // Lock acquired by this thread

}

}.start();

}

public void run() {

//中断判断

while (true) {

if (Thread.interrupted()) {

System.out.println("中断线程!!");

break;

} else {

f();

}

}

}

public static void main(String[] args) throws InterruptedException {

SynchronizedBlocked sync = new SynchronizedBlocked();

Thread t = new Thread(sync);

//启动后调用f()方法,无法获取当前实例锁处于等待状态

t.start();

TimeUnit.SECONDS.sleep(1);

//中断线程,无法生效

t.interrupt();

}

}

我们在SynchronizedBlocked构造函数中**创建一个新线程并启动获取调用f()获取到当前实例锁**，由于**SynchronizedBlocked自身也是线程，启动后在其run方法中也调用了f()，但由于对象锁被其他线程占用，导致t线程只能等到锁**，此时我们调用了t.interrupt();**（即锁池里的线程不会被中断）**但并不能中断线程。

 Thread.interrupt()方法不会中断一个正在运行的线程。它的作用是，在线程受到阻塞时抛出一个中断信号，这样线程就得以退出阻塞的状态。更确切的说，如果线程被**Object.wait()、Thread.join()以及Thread.sleep()三种方法之一阻塞**，那么，它将接收到一个中断异常（InterruptedException），从而提早地终结被阻塞状态。

## 等待与唤醒机制与synchronized

所谓等待唤醒机制本篇主要指的是notify/notifyAll（进入锁池）和wait（进入等待队列），在使用这3个方法时，必须处于synchronized代码块或者synchronized方法中，否则就会抛出IllegalMonitorStateException异常，这是因为调用这几个方法前**必须拿到当前对象的监视器monitor对象，也就是说notify/notifyAll和wait方法依赖于monitor对象（首先得知道竞争哪个锁）**，在前面的分析中，我们知道monitor 存在于对象头的Mark Word 中**(存储monitor引用指针)**，而**synchronized关键字可以获取 monitor** ，这也就是为什么notify/notifyAll和wait方法必须在synchronized代码块或者synchronized方法调用的原因。

synchronized (obj) {

obj.wait();

obj.notify();

obj.notifyAll();

}

需要特别理解的一点是，与sleep方法不同的是wait方法调用完成后，线程将被暂停，但wait方法将会释放当前持有的监视器锁(monitor)，直到有线程调用notify/notifyAll方法后方能继续执行，而**sleep方法只让线程休眠并不释放锁。**同时**notify/notifyAll方法调用后（只是进入锁池），并不会马上释放监视器锁，而是在相应的synchronized(){}/synchronized方法执行结束后才自动释放锁。**