### 1、线程与进程

一个程序最少需要一个进程，而一个进程最少需要一个线程。

关系是线程 -> 进程 -> 程序的大致组成结构

所以线程是程序执行流的最小单位，而进程是系统进行资源分配和调度的一个独立单位。以下我们所有讨论的都是建立在线程基础之上。

### 2、Thread的几个重要方法

我们先了解一下Thread的几个重要方法。

* start()方法，调用该方法开始执行该线程；
* stop()方法，调用该方法强制结束该线程执行；
* join方法，调用该方法等待该线程结束。
* sleep()方法，调用该方法该线程进入等待。
* run()方法，调用该方法直接执行线程的run()方法，但是线程调用start()方法时也会运行run()方法，区别就是一个是由线程调度运行run()方法，一个是直接调用了线程中的run()方法！！

看到这里，可能有些人就会问啦，那wait()和notify()呢？要注意，其实wait()与notify()方法是Object的方法，不是Thread的方法！！同时，wait()与notify()会配合使用，分别表示线程挂起和线程恢复。

这里还有一个很常见的问题，顺带提一下：wait()与sleep()的区别，简单来说wait()会释放对象锁而sleep()不会释放对象锁。

### 3、线程状态



线程总共有5大状态：

* 新建状态：新建线程对象，并没有调用start()方法之前
* 就绪状态：调用start()方法之后线程就进入就绪状态，但是并不是说只要调用start()方法线程就马上变为当前线程，在变为当前线程之前都是为就绪状态。值得一提的是，线程在睡眠和挂起中恢复的时候也会进入就绪状态哦。
* 运行状态：线程被设置为当前线程，开始执行run()方法。就是线程进入运行状态
* 阻塞状态：线程被暂停，比如说调用sleep()方法后线程就进入阻塞状态
* 死亡状态：线程执行结束

### 4、锁类型

* 可重入锁：在执行对象中所有同步方法不用再次获得锁
* 可中断锁：在等待获取锁过程中可中断
* 公平锁：按等待获取锁的线程的等待时间进行获取，等待时间长的具有优先获取锁权利
* 读写锁：对资源读取和写入的时候拆分为2部分处理，读的时候可以多线程一起读，写的时候必须同步地写

### 5、synchronized与Lock的区别



### 6、Lock详细介绍与demo

Lock接口源码：

public interface Lock {

*/\*\*  
 \* Acquires the lock.  
 \*/* void lock();  
  
 */\*\*  
 \* Acquires the lock unless the current thread is  
 \* {****@linkplain*** *Thread#interrupt interrupted}.  
 \*/* void lockInterruptibly() throws InterruptedException;  
  
 */\*\*  
 \* Acquires the lock only if it is free at the time of invocation.  
 \*/* boolean tryLock();  
  
 */\*\*  
 \* Acquires the lock if it is free within the given waiting time and the  
 \* current thread has not been {****@linkplain*** *Thread#interrupt interrupted}.  
 \*/* boolean tryLock(long time, TimeUnit unit) throws InterruptedException;  
  
 */\*\*  
 \* Releases the lock.  
 \*/* void unlock();  
  
 */\*\*  
 \* Returns a new {****@link*** *Condition} instance that is bound to this  
 \* {****@code*** *Lock} instance.  
 \*/* Condition newCondition();  
}

接口中的方法：

* lock()：获取锁，如果锁被占用则一直等待
* unlock()：释放锁
* tryLock(): 注意返回类型是boolean，如果获取锁的时候锁被占用就返回false，否则返回true
* tryLock(long time, TimeUnit unit)：比起tryLock()就是给了一个时间期限，保证等待参数时间
* lockInterruptibly()：用该锁的获得方式，如果线程在获取锁的阶段进入了等待，那么可以中断此线程，先去做别的事

通过以上的解释，大致可以解释在上个部分中“锁类型(lockInterruptibly())”，“锁状态(tryLock())”等问题，还有就是前面子所获取的过程我所写的“大致就是可以尝试获得锁，线程可以不会一直等待”用了“可以”的原因。

下面是Lock一般使用的例子，注意ReentrantLock是Lock接口的实现。

lock()：

public class LockTest {

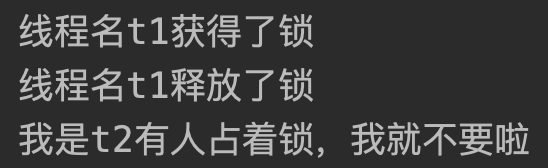
private Lock lock = new ReentrantLock();  
  
 */\*\*  
 \* 需要参与同步的方法  
 \*/* private void method(Thread thread) {  
 lock.lock();  
 try {  
 System.*out*.println("线程名" + thread.getName() + "获得了锁");  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 System.*out*.println("线程名" + thread.getName() + "释放了锁");  
 }  
 }  
  
 public static void main(String[] args) {  
 LockTest lockTest = new LockTest();  
  
 //线程t1  
 Thread t1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 lockTest.method(Thread.*currentThread*());  
 }  
 }, "t1");  
  
 //线程t2  
 Thread t2 = new Thread(() -> lockTest.method(Thread.*currentThread*()), "t2");  
  
 t1.start();  
 t2.start();  
 }  
}

tryLock()：

public class TryLockTest {

private Lock lock = new ReentrantLock();  
  
 */\*\*  
 \* 需要参与同步的方法  
 \*/* private void method(Thread thread) {  
 if (lock.tryLock()) {  
 try {  
 System.*out*.println("线程名" + thread.getName() + "获得了锁");  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 System.*out*.println("线程名" + thread.getName() + "释放了锁");  
 lock.unlock();  
 }  
 } else {  
 System.*out*.println("我是" + Thread.*currentThread*().getName() + "有人占着锁，我就不要啦");  
 }  
 }  
  
 public static void main(String[] args) {  
 TryLockTest tryLockTest = new TryLockTest();  
  
 //线程1  
 Thread t1 = new Thread(() -> tryLockTest.method(Thread.*currentThread*()), "t1");  
  
 Thread t2 = new Thread(() -> tryLockTest.method(Thread.*currentThread*()), "t2");  
  
 t1.start();  
 t2.start();  
 }  
}

结果：



看到这里相信大家也都会使用如何使用Lock了吧，关于tryLock(long time, TimeUnit unit)和lockInterruptibly()不再赘述。前者主要存在一个等待时间，在测试代码中写入一个等待时间，后者主要是等待中断，会抛出一个中断异常，常用度不高，喜欢探究可以自己深入研究。

前面比较重提到“公平锁”，在这里可以提一下ReentrantLock对于平衡锁的定义，在源码中有这么两段：

*/\*\**

*\* Sync object for non-fair locks  
 \*/*static final class NonfairSync extends Sync {  
 private static final long *serialVersionUID* = 7316153563782823691L;  
  
 */\*\*  
 \* Performs lock. Try immediate barge, backing up to normal  
 \* acquire on failure.  
 \*/* final void lock() {  
 if (compareAndSetState(0, 1))  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 else  
 acquire(1);  
 }  
  
 protected final boolean tryAcquire(int acquires) {  
 return nonfairTryAcquire(acquires);  
 }  
}

*/\*\*  
 \* Sync object for fair locks  
 \*/*static final class FairSync extends Sync {  
 private static final long *serialVersionUID* = -3000897897090466540L;  
  
 final void lock() {  
 acquire(1);  
 }  
  
 */\*\*  
 \* Fair version of tryAcquire. Don't grant access unless  
 \* recursive call or no waiters or is first.  
 \*/* protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 if (!hasQueuedPredecessors() &&  
 compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0)  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 return false;  
 }  
}

从以上源码可以看出在Lock中可以自己控制锁是否公平，而且，默认的是非公平锁，以下是ReentrantLock的构造函数：

*/\*\**

*\* Creates an instance of {****@code*** *ReentrantLock}.  
 \* This is equivalent to using {****@code*** *ReentrantLock(false)}.  
 \*/*public ReentrantLock() {  
 sync = new NonfairSync(); // 默认非公平锁  
}  
  
*/\*\*  
 \* Creates an instance of {****@code*** *ReentrantLock} with the  
 \* given fairness policy.  
 \*/*public ReentrantLock(boolean fair) {  
 sync = fair ? new FairSync() : new NonfairSync();  
}

### 7、补充

#### 7.1 两种锁的底层实现方式

synchronized：我们知道java是用字节码指令来控制程序（这里不包括热点代码编译成机器码）。在字节指令中，存在有synchronized所包含的代码块，那么会形成2段流程的执行。

查看SyncDemo.class源码：

$ javap -v SyncDemo.class

Classfile /Users/chance/IdeaProjects/java-

basis/target/classes/com/chance/basis/thread/sync/SyncDemo.class

Last modified 2020-7-13; size 601 bytes

MD5 checksum 2c30158d651b1d3ec3e3c901805159a3

Compiled from "SyncDemo.java"

public class com.chance.basis.thread.sync.SyncDemo

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

#1 = Methodref #5.#20 // java/lang/Object."<init>":()V

#2 = Class #21 // com/chance/basis/thread/sync/SyncDemo

#3 = Fieldref #22.#23 // java/lang/System.out:Ljava/io/PrintStream;

#4 = Methodref #24.#25 // java/io/PrintStream.println:()V

#5 = Class #26 // java/lang/Object

#6 = Utf8 <init>

#7 = Utf8 ()V

#8 = Utf8 Code

#9 = Utf8 LineNumberTable

#10 = Utf8 LocalVariableTable

#11 = Utf8 this

#12 = Utf8 Lcom/chance/basis/thread/sync/SyncDemo;

#13 = Utf8 sync

#14 = Utf8 StackMapTable

#15 = Class #21 // com/chance/basis/thread/sync/SyncDemo

#16 = Class #26 // java/lang/Object

#17 = Class #27 // java/lang/Throwable

#18 = Utf8 SourceFile

#19 = Utf8 SyncDemo.java

#20 = NameAndType #6:#7 // "<init>":()V

#21 = Utf8 com/chance/basis/thread/sync/SyncDemo

#22 = Class #28 // java/lang/System

#23 = NameAndType #29:#30 // out:Ljava/io/PrintStream;

#24 = Class #31 // java/io/PrintStream

#25 = NameAndType #32:#7 // println:()V

#26 = Utf8 java/lang/Object

#27 = Utf8 java/lang/Throwable

#28 = Utf8 java/lang/System

#29 = Utf8 out

#30 = Utf8 Ljava/io/PrintStream;

#31 = Utf8 java/io/PrintStream

#32 = Utf8 println

{

public com.chance.basis.thread.sync.SyncDemo();

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 11: 0

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 this Lcom/chance/basis/thread/sync/SyncDemo;

public void sync();

descriptor: ()V

flags: ACC\_PUBLIC

Code:

stack=2, locals=3, args\_size=1

0: ldc #2 // class com/chance/basis/thread/sync/SyncDemo

2: dup

3: astore\_1

4: monitorenter

5: getstatic #3 // Field java/lang/System.out:Ljava/io/PrintStream;

8: invokevirtual #4 // Method java/io/PrintStream.println:()V

11: aload\_1

12: monitorexit // 执行完释放

13: goto 21// 如果正常运行结束会跳转到21行执行

16: astore\_2

17: aload\_1

18: monitorexit // 发生异常，虚拟机释放

19: aload\_2

20: athrow

21: return

Exception table:

from to target type

5 13 16 any

16 19 16 any

LineNumberTable:

line 14: 0

line 15: 5

line 16: 11

line 17: 21

LocalVariableTable:

Start Length Slot Name Signature

0 22 0 this Lcom/chance/basis/thread/sync/SyncDemo;

StackMapTable: number\_of\_entries = 2

frame\_type = 255 /\* full\_frame \*/

offset\_delta = 16

locals = [ class com/chance/basis/thread/sync/SyncDemo, class java/lang/Object ]

stack = [ class java/lang/Throwable ]

frame\_type = 250 /\* chop \*/

offset\_delta = 4

}

SourceFile: "SyncDemo.java"

如上就是这段代码段字节码指令。我们可以清晰段看到，其实synchronized映射成字节码指令就是增加来两个指令：monitorenter和monitorexit。当一条线程进行执行的遇到monitorenter指令的时候，它会去尝试获得锁，如果获得锁那么锁计数+1（为什么会加一呢，因为它是一个可重入锁，所以需要用这个锁计数判断锁的情况），如果没有获得锁，那么阻塞。当它遇到monitorexit的时候，锁计数器-1，当计数器为0，那么就释放锁。

那么有的朋友看到这里就疑惑了，那图上有2个monitorexit呀？马上回答这个问题： synchronized锁释放有两种机制，一种就是执行完释放；另外一种就是发送异常，虚拟机释放。图中第二个monitorexit就是发生异常时执行的流程，这就是我开头说的“会有2个流程存在“。而且，从图中我们也可以看到在第13行，有一个goto指令，也就是说如果正常运行结束会跳转到21行执行。

接下来我们再聊一聊Lock。

Lock：Lock实现和synchronized不一样，后者是一种悲观锁，它胆子很小，它很怕有人和它抢吃的，所以它每次吃东西前都把自己关起来。而Lock底层其实是CAS乐观锁的体现，它无所谓，别人抢了它吃的，它重新去拿吃的就好啦，所以它很乐观。如果面试问起，你就说底层主要靠volatile和CAS操作实现的。

#### 7.2 尽可能去使用synchronized而不要去使用LOCK

什么概念呢？我和大家打个比方：你叫jdk，你生了一个孩子叫synchronized，后来呢，你领养了一个孩子叫LOCK。起初，LOCK刚来到新家的时候，它很乖，很懂事，各个方面都表现的比synchronized好。你很开心，但是你内心深处又有一点淡淡的忧伤，你不希望你自己亲生的孩子竟然还不如一个领养的孩子乖巧。这个时候，你对亲生的孩子教育更加深刻了，你想证明，你的亲生孩子synchronized并不会比领养的孩子LOCK差。（博主只是打个比方）

那如何教育呢？

在jdk1.6~jdk1.7的时候，也就是synchronized16、7岁的时候，你作为爸爸，你给他优化了，具体优化在哪里呢：

（1）线程自旋和适应性自旋

我们知道，java’线程其实是映射在内核之上的，线程的挂起和恢复会极大的影响开销。并且jdk官方人员发现，很多线程在等待锁的时候，在很短的一段时间就获得了锁，所以它们在线程等待的时候，并不需要把线程挂起，而是让他无目的的循环，一般设置10次。这样就避免了线程切换的开销，极大的提升了性能。

而适应性自旋，是赋予了自旋一种学习能力，它并不固定自旋10次一下。他可以根据它前面线程的自旋情况，从而调整它的自旋，甚至是不经过自旋而直接挂起。

（2）锁消除

什么叫锁消除呢？就是把不必要的同步在编译阶段进行移除。

那么有的小伙伴又迷糊了，我自己写的代码我会不知道这里要不要加锁？我加了锁就是表示这边会有同步呀？

并不是这样，这里所说的锁消除并不一定指代是你写的代码的锁消除，我打一个比方：

在jdk1.5以前，我们的String字符串拼接操作其实底层是StringBuffer来实现的（这个大家可以用我前面介绍的方法，写一个简单的demo，然后查看class文件中的字节码指令就清楚了），而在jdk1.5之后，那么是用StringBuilder来拼接的。我们考虑前面的情况，比如如下代码：

String str1="qwe";

String str2="asd";

String str3=str1+str2;

底层实现会变成这样：

StringBuffer sb = new StringBuffer();

sb.append("qwe");

sb.append("asd");

我们知道，StringBuffer是一个线程安全的类，也就是说两个append方法都会同步，通过指针逃逸分析（就是变量不会外泄），我们发现在这段代码并不存在线程安全问题，这个时候就会把这个同步锁消除。

（3）锁粗化

在用synchronized的时候，我们都讲究为了避免大开销，尽量同步代码块要小。那么为什么还要加粗呢？

我们继续以上面的字符串拼接为例，我们知道在这一段代码中，每一个append都需要同步一次，那么我可以把锁粗化到第一个append和最后一个append（这里不要去纠结前面的锁消除，我只是打个比方）

（4）轻量级锁

待完善。。。

（5）偏向锁

待完善。。。