### 创建线程

##### 通过实现 Runnable 接口

class RunnableDemo implements Runnable {

private Thread t;

private String threadName;

RunnableDemo( String name) {

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

...

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

##### 通过继承 Thread 类本身

class ThreadDemo extends Thread {

private Thread t;

private String threadName;

ThreadDemo( String name) {

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

...

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

##### 通过 Callable 和 Future 创建线程

Test3 com.chen.testthread.TestCallable.java 代码来源

public class CallableThreadTest implements Callable<Integer> {

private String threadName;

public TestCallable(String name){

this.threadName = name;

}

@Override

public Integer call() throws Exception {

System.out.println("执行线程"+Thread.currentThread().getName());

return 2;

}

public Integer start() throws InterruptedException, ExecutionException{

FutureTask<Integer> task = new FutureTask<Integer>(this);

Thread thread = new Thread(task, this.threadName);

thread.start();

return task.get();

}

public static void main(String[] args){

TestCallable call = new TestCallable("有返回值的线程");

try {

System.out.println("返回结果："+call.start());

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

// 也可以使用Lambda表达式创建Callable对象

// 使用FutureTask类来包装Callable对象，该FutureTask对象封装了Callable对

// 象的call()方法的返回值

　　　 FutureTask<Integer> future=new FutureTask<Integer>(

　　　　 (Callable<Integer>)()->{

　　　　　　 return 5;

　　　　 }

　　 );

// 后续的调用方法一样

Thread thread2 = new Thread(futrue, “”);

thread2 .strat();

...

}

}

##### 对比三种方式：

1. 采用实现 Runnable、Callable 接口的方式创建多线程时，线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。

2. Callable接口提供了一个call（）

1). call()方法可以有返回值

2). call()方法可以声明抛出异常

3).Java5提供了Future接口来代表Callable接口里call()方法的返回值

并且为Future接口提供了一个实现类FutureTask

这个实现类既实现了Future接口，还实现了Runnable接口

因此可以作为Thread类的target

在Future接口里定义了几个公共方法来控制它关联的Callable任务：

>V get()：返回Callable里call方法的返回值，调用程序会阻塞，子线程结束后才会返回

>boolean cancel(boolean mayInterruptIfRunning)：视图取消该Future里面关联的

Callable任务

>V get(long timeout,TimeUnit unit)：返回Callable里call（）方法的返回值，最多阻塞

timeout时间，经过指定时间没有返回抛出TimeoutException

>boolean isDone()：若Callable任务完成，返回True

>boolean isCancelled()：如果在Callable任务正常完成前被取消，返回True

3. 使用继承 Thread 类的方式创建多线程时，编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread() 方法，直接使用 this 即可获得当前线程。

### 多线程的使用

有效利用多线程的关键是理解程序是并发执行而不是串行执行的。例如：程序中有两个子系统需要并发执行，这时候就需要利用多线程编程。

通过对多线程的使用，可以编写出非常高效的程序。不过请注意，如果你创建太多的线程，程序执行的效率实际上是降低了，而不是提升了。

请记住，上下文的切换开销也很重要，如果你创建了太多的线程，CPU 花费在上下文的切换的时间将多于执行程序的时间！

#### Thread方法

**常用重要方法**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public void start()** 使该线程开始执行；**Java** 虚拟机调用该线程的 run 方法。 |
| 2 | **public void run()** 如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。 |
| 3 | **public final void setName(String name)** 改变线程名称，使之与参数 name 相同。 |
| 4 | **public final void setPriority(int priority)**  更改线程的优先级。 |
| 5 | **public final void setDaemon(boolean on)** 将该线程标记为守护线程或用户线程。 |
| 6 | **public final void join(long millisec)** 等待该线程终止的时间最长为 millis 毫秒。 |
| 7 | **public void interrupt()** 中断线程。 |
| 8 | **public final boolean isAlive()** 测试线程是否处于活动状态。 |

**Thread类的静态方法**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public static void yield()** 暂停当前正在执行的线程对象，并执行其他线程。 |
| 2 | **public static void sleep(long millisec)** 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。 |
| 3 | **public static boolean holdsLock(Object x)** 当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。 |
| 4 | **public static Thread currentThread()** 返回对当前正在执行的线程对象的引用。 |
| 5 | **public static void dumpStack()** 将当前线程的堆栈跟踪打印至标准错误流。 |

#### Synchronized实现线程同步

#### 使用synchronized时释放锁总结

1、当前线程的同步方法、代码块执行结束的时候释放

2、当前线程在同步方法、同步代码块中遇到break 、return 终于该代码块或者方法的时候释放。

3、出现未处理的error或者exception导致异常结束的时候释放

4、程序执行了 同步对象 wait 方法 ，当前线程暂停，释放锁

**在以下两种情况不会释放锁：**

1、代码块中使用了 Thread.sleep()  Thread.yield() 这些方法暂停线程的执行，不会释放。

2、线程执行同步代码块时，其他线程调用 suspend 方法将该线程挂起，该线程不会释放锁 ，所以我们应该避免使用 suspend 和 resume 来控制线程 。

#### 线程数据的共享与否

下面是共享和不共享的案例，一个是创建原有类此此处的ThreadNoShareData，另一个是通过Thread来创建，就能实现线程的共享与否，所以区分线程是否共享数据，主要看他们是否同时操作同一个实例，如下代码

**线程不共享**

此处都有各自的count变量

package com.chen.testthread;

public class ThreadNoShareData extends Thread {

public ThreadNoShareData(String name){

super();

this.setName(name);

}

private Integer count = 4;

public void run(){

super.run();

count--;

System.out.println(this.currentThread().getName()+",count:"+count);

}

public static void main(String[] args){

ThreadNoShareData thread1 = new ThreadNoShareData("a1");

ThreadNoShareData thread2 = new ThreadNoShareData("a2");

ThreadNoShareData thread3 = new ThreadNoShareData("a3");

ThreadNoShareData thread4 = new ThreadNoShareData("a4");

thread1.start();

thread2.start();

thread3.start();

thread4.start();

}

}

线程共享 （同时操作一个实例）

此处共享的标量是count

package com.chen.testthread;

public class ThreadShareData extends Thread {

private Integer count = 4;

public void run(){

super.run();

count--;

System.out.println(this.currentThread().getName()+",count:"+count);

}

public static void main(String[] args){

ThreadShareData d = new ThreadShareData();

Thread thread1 = new Thread(d, "a1");

Thread thread2 = new Thread(d, "a2");

Thread thread3 = new Thread(d, "a3");

Thread thread4 = new Thread(d, "a4");

thread1.start();

thread2.start();

thread3.start();

thread4.start();

}

}

线程共享还有一个案例：

public class ThreadShare extends Thread{

public void run(){

super.run();

System.out.println(Thread.currentThread().getName()+" "+this.getName());

}

public static void main(String[] args){

ThreadYield t = new ThreadYield();

Thread thread = new Thread(t, "a");

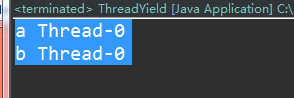
Thread threa2 = new Thread(t, "b");

thread.start();

threa2.start();

}

}



这里可以看出输出的this.getName() 都是“Thread-0”。说明此时的数据是共享的

#### 线程安全

当有多个线程同时操作同一个对象时就会产生了线程安全的问题，

即“非线程安全”，非线程安全主要指多个线程对同一个对象中的同一个实例变量进行操作时会出现值被更改、值不同步的情况，而影响程序的执行流程。

如下代码：

package com.chen.testthread;

public class ThreadSafe extends Thread {

public ThreadSafe(String name){

super();

this.setName(name);

}

public void run (){

super.run();

TestServlet.post("a", "aa");

}

public static void main(String[] args){

ThreadSafe t1 = new ThreadSafe("a1");

ThreadSafe2 t2 = new ThreadSafe2("a2");

t2.start();

t1.start();

}

}

class ThreadSafe2 extends Thread {

public ThreadSafe2(String name){

super();

this.setName(name);

}

public void run (){

super.run();

TestServlet.post("b", "bb");

}

}

class TestServlet {

private static String username;

private static String password;

public static void post(String username, String password){

TestServlet.username = username;

if(TestServlet.username.equals("a"));

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

TestServlet.password = password;

System.out.println(Thread.currentThread().getName()+";username:" + TestServlet.username + ", password: " + TestServlet.password);

}

}



如何才能使上面的多个线程访问一个对象的线程安全呢，在post方法中使用“synchronized”锁，数据将会被正常的显示，原因：加入关键字，那么这个方法被调用时就如同上了一把锁，在其他线程访问此方法时就会判断该方法是否上锁，上锁则等待锁的释放，因此这个处理方式是依次排队进行的，所以互相不会产生干扰，因此达到线程的安全



需要注意同步的还有 ‘i--’此类问题

#### 注意i - - 和System.out.println()

这两个都会产生安全问题，i--需要同步。Syso内部有同步，在需要释放锁的时候可能不一定能成功释放锁例如suspend中，详情看后续记录

#### API

currentThread()

isAlive() 判断线程的活动状态（线程已经启动且尚未终止）

Sleep()

getId() 获取线程的唯一标识

详情请看上文的Thread方法

#### 停止线程

**为什么要停止**

任务已经完成，需要放弃当前操作

**怎么停**

1）Thread.stop() 这个方法已经被废弃，不推荐使用

**强制停止的坏处**：

强制停止则有可能使一些清理性的工作得不到完成。对锁进行了解锁，导致数据得不到同步的处理，出现数据不一致的问题。代码如下：

public class ThreadStop extends Thread {

private POJOBean bean;

public ThreadStop(POJOBean bean){

this.bean = bean;

}

public void run(){

super.run();

bean.print("b", "bb");

}

public static void main(String[] args){

POJOBean bean = new POJOBean();

ThreadStop t = new ThreadStop(bean);

Thread thread = new Thread(t, "A");

thread.start();

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

thread.stop();

System.out.println(bean.getName()+" "+bean.getPass());

}

}

class POJOBean{

private String name = "a";

private String pass = "aa";

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getPass() {

return pass;

}

public void setPass(String pass) {

this.pass = pass;

}

public void print(String name, String pass){

this.name = name;

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

e.printStackTrace();

}

this.pass = pass;

}

}

从结果看出现数据有问题



2）Thread.interrupt() 中断线程，并且需要加入一个判断才能完成线程的停止

在使用该方法进行停止线程时并没有马上进行停止，只是打了一个停止的标记，所以说想让线程终止，那就得判断停止标志来作出相应的动作

获取终止的方式有两种：

Public static boolean **interrupted**()

Public boolean **isInterrupted**()

两者的区别是：

前者第一次调用之后会清除标记，所以第二次调用就不是当前的标记了

，后者调用之后不会清除

1. 使用退出标志，使线程正常退出
2. 利用异常停止线程

public class ThreadStopByException extends Thread {

public void run(){

super.run();

try {

for(int i = 0; i < 50000; i++){

if(Thread.interrupted()){

throw new **InterruptedException**("通过异常来停止线程 ");

}

System.out.println(this.currentThread().getName()+",i:"+i);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public static void main(String[] args){

ThreadStopByException t = new ThreadStopByException();

t.start();

try {

Thread.sleep(2000);

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

t.interrupt();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

1. return停止线程

和interrupted一起使用

if(Thread.interrupted()){

return;

}

#### 暂停线程

暂停的线程可以进行恢复

**为什么要暂停**

**怎么暂停**

1. **suspend与resume方法暂停线程**
   1. **缺点 -- 独占**

在使用这两个方法时，如果使用不当，极易造成公共的同步对象的独占，使得其他线程无法访问公共同步对象，如下代码所示：

package com.chen.testthread;

public class ThreadSuspendAndResume extends Thread{

synchronized public void run(){

super.run();

System.out.println(Thread.currentThread().getName()+":begin");

if(Thread.currentThread().getName().equals("a")){

System.out.println("a线程将永远独占此锁");

Thread.currentThread().**suspend**();

}

System.out.println(Thread.currentThread().getName()+":end");

}

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

ThreadSuspendAndResume thread = new ThreadSuspendAndResume();

Thread a = new Thread(thread, "a");

Thread b = new Thread(thread, "b");

System.out.println("a启动线程");

a.start();

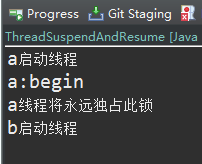
Thread.sleep(2000);

b.start();

System.out.println("b启动线程");

}

}



另一种是因为System.out.println()方法

public class ThreadSuspendAndResume2 extends Thread{

private long i = 0;

public void run(){

super.run();

System.out.println(Thread.currentThread().getName()+":begin");

while(true){

i++;

System.out.println(i);

}

}

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

ThreadSuspendAndResume2 thread = new ThreadSuspendAndResume2();

thread.start();

Thread.sleep(1);

System.out.println("suspend");

thread.suspend();

System.out.println("main end");

}

}

批注那一句不一定输出是因为System.out.println()这个println方法是带锁的，而调用suspend方法间接暂停了一个带锁的方法，导致和上面案例一样锁独占的问题

* 1. **缺点 -- 不同步**
  2. **xxx**

1. **xx**

#### 方法yield

作用是放弃当前的CPU资源，将它让给其他的任务去占用。但放弃的时间不确定，有可能刚放弃就马上获取CPU时间片，如下代码演示：

package com.chen.testthread;

public class ThreadYield extends Thread{

public void run(){

super.run();

long count = 0;

long start = System.currentTimeMillis();

for(long i = 0; i < 50000000; i++){

Thread.yield();

count = count + i;

}

System.out.println("所用时间：" + (System.currentTimeMillis() - start) + "毫秒");

}

public static void main(String[] args){

ThreadYield t = new ThreadYield();

Thread thread = new Thread(t, "a");

thread.start();

}

}

#### 线程优先级

1. 通过此方法设置优先级setPriority
2. 有三个预置的量

public final static int MIN\_PRIORITY = 1;

public final static int NORE\_PRIORITY = 5;

public final static int MAX\_PRIORITY = 10;

1. 优先级具有继承性，优先级会跟随创建它的线程的优先级
2. 高优先级总数大部分先执行完，并不是搞优先级线程全部先执行完

#### 守护线程

在java线程中有两种线程，一种是用户线程，另一种是守护线程

守护线程时一种特殊的线程，它有陪伴的含义，当进程中不存在非守护线程了，则守护线程自动销毁。典型的守护线程就是垃圾回收线程（GC），创建守护线程如下代码：

public class ThreadDaemon extends Thread{

private int i = 0;

public void run(){

super.run();

while(true){

i++;

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(i);

}

}

public static void main(String[] args){

ThreadDaemon d = new ThreadDaemon();

Thread thread = new Thread(d, "d");

thread.setDaemon(true);

thread.start();

try {

Thread.sleep(6000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("停止主线程，那么守护线程也会停止");

}

}

### 对象及变量的并发访问

**synchronized同步方法**

**为什么要同步**

数据的脏读使程序的执行不正常

**什么导致不同步或者不同步的位置在哪**

非线程安全问题存在于实例变量，方法内部私有变量不存在安全问题

所以要考虑的就是实例变量

有两个代码案例：

如果有两个线程同时操作业务对象中的实例变量则有可能会出现“非线程安全”

**怎么同步**

最后用一个案例非线程相关的方法修饰了synchronized，说明了怎么同步

**术语** 同步 synchroized 异步 asynchronized

共享,只有共享资源的读写访问才需要同步化，如果不是共享资源，那么就没有同步的必要

线程锁是作用于对象上的，而不是方法

**可重入锁** 自己可以获取自己的内部锁

同步不具有继承性

#### synchronized同步语句块

Synchronized声明方法会比较耗时，

所以建议使用最小同步块synchronized(this)

Synchronized使用的是“对象监视器”

当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对同一个object中所有其他synchronized(this)同步代码块的访问将被阻塞

Synchronized(非this)

不予其他锁this同步方法争抢this锁，这么看来是可以大大提高效率的

锁this和锁非this（任意其他的对象）

对象监视器不同，运行结果会是异步，如同一个类中的synchronized方法和synchronized非this，两个线程同时操作这个对象的两个方法时是异步的执行

三个结论

1. 使用同一个对象监视器和使用不同监视器的效果：前者同步，后者异步
2. 为了和a（a中有同步块）对象同步，特定在b对象中使用a来作为对象监视器（b中用了锁非this），达到了同步
3. 把上面的第二点的a对象中的同步块换成锁this，同样成立

静态同步synchronized方法时对当前的java文件的class类进行持锁

静态锁对任何实例都起作用

锁class和静态锁一样

不用String作为锁对象，为什么呢。

因为String常量池的问题。通过书上代码测试得，两个相同字符串，在作为锁对象的时候，只有一个线程能获得锁，是同步运行的

将不同字符串作为锁对象时候，两个线程是异步的

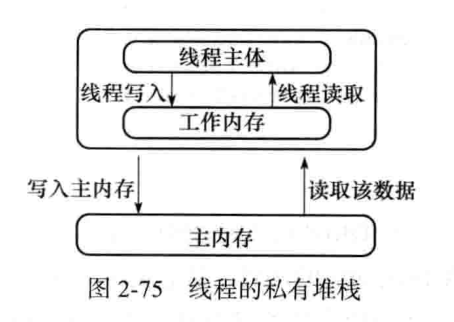
解决线程死锁

使用JDK自带工具检测是否有死锁现象

使用任何任何类型的对象作为锁时，只要获取锁对象相同就是同步，获取不同对象锁则异步

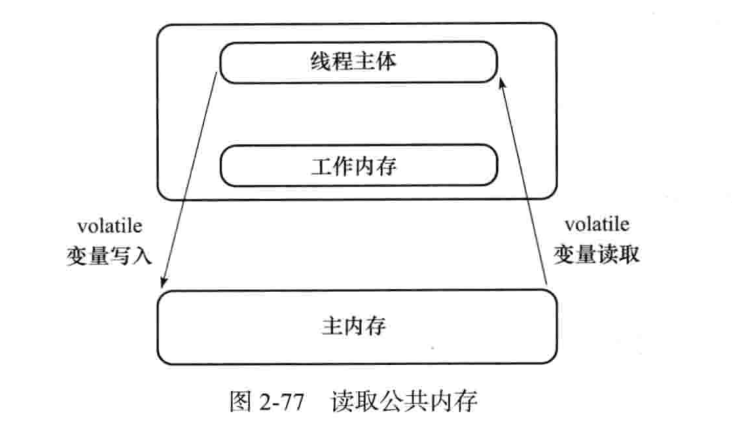
#### volatile关键字

通常变量在私有堆栈中的值和公共堆栈的值并不同步



关键字主要作用是使变量在多个线程间可见。

使用volatile关键字时强制的从公共内存中读取变量值，统一读取的方式，不再从私有数据栈中取。这也是可见性的表现



关键字volatile解决的是多个线程之间访问资源的同步性

Synchronized和volatile的比较（两者都是实现同步）

1. Volatile是线程同步的轻量级实现，性能相对synchronized要好，volatile只能修饰变量，synchronized可以修饰方法、代码块
2. Volatile不会发生阻塞，而synchronized会出现阻塞
3. Volatile能保证数据的的可见性，但不能保证原子性，而synchronized可以保证原子性，也可以间接保证可见性，因为他会将私有内存和公共内存中的数据做同步
4. Volatile解决的是变量在多个线程之间的可见性；而synchronized解决的是多个线程之间访问资源的同步性

线程的安全包含原子性和可见性。Java同步机制都是围绕这两个方面来确保线程安全

volatile非原子特性，不具备操作的同步性（虽然读取公共内存访问同步，但在一些例如i++的多步操作中，是需要加锁让其同步的）

关键字的使用主要作用是多线程读取共享变量时，可以获得最新值

使用AtomicInteger原子类进行i++操作（另一种同步方式）

这个类保证了i++的原子操作，但是在多个线程同时引用此实例变量时，依然需要加锁，让众多的线程操作同步

Synchronized也能具备多个线程访问资源的同步性

### 线程间的通信

#### 等待、通知机制

循环轮询判断结果其实也能实现线程之间的协作，但是这样会浪费cpu资源

只能在同步方法或同步块中调用wait方法，因为在调用wait方法时，线程必须获得该对象的对象锁（notify也是如此）

调用notify时不会马上释放锁，退出synchronized块后才释放锁，只有这样wait才能获取需要的锁对象。（而wait是即刻释放锁）

调用notify只随机唤醒一个线程

Wait（long） 达到最大时间限定，自动唤醒

**生产者消费者模式**

**什么是假死**

不管是消费者，还是生产者，都处于等待状态

**如何解决假死**

使用notifyAll，不仅可以唤醒异类，也可唤醒同类，避免“假死的情况”

使用while来判断条件的改变

通过管道进行线程间通信：字节流

PipedInputStream和PipedOutputStream

通过管道进行线程间通信：字符流

PipedReader和PipedWriter

使用：流对象.connecte(流对象) ，以此达到两个流的关联，实现通信

#### Join的使用

在主线程结束了，但是子线程还没运行完，这时候可能需要等待这个子线程运行完，程序的数据才不会丢失，此时就要用到join方法了。Join的作用是等待线程对象销毁。

Join具有使线程同步的效果（主线程等待join线程运行完就是一种同步效果了）

Join和synchronized的区别是，join内部使用wait方法进行等待，而synchronized使用的是对象监视器作为同步

Join和interrupt方法遇到时，会出现异常

Join（long）和sleep（long） 区别是，是否释放锁

对于书上的陷阱一说不太理解

#### 类ThreadLocal的使用

变量值的共享可以使用public static形式，但是我们只是需要在一个线程内有自己的共享变量呢，可以使用ThreadLocal类，可以比喻成全局存放数据的盒子，存储每个线程的私有数据

Public static ThreadLocal local = new ThreadLocal() // 这个对象供所有线程调用

Set（放入数据）、get（）取得数据。每个线程都这么操作一个ThreadLocal静态实例，但是在各自调用get方法时确是各自set的值，达到了隔离的效果

### Lock的使用

#### ReentrantLock类的使用

ReentrantLock也能达到同步互斥的效果，并且具有嗅探锁定、多路分支通知等功能，使用上比Synchronized更加灵活

**使用ReentrantLock实现同步**

.lock 方法持有对象监视器

.unlock 方法释放对象监视器

ReentrantLock结合Condition可以实现选择性通知，不同与notify随机通知

.await() 方法前必须先获取对象监视器

一个消费者和一个生产者的交替作业，通过condition不再有假死的现象

多消费者和多生产者的交替作业，通过condition 使用signAll通知所有 没有假死

公平锁表示线程获取锁的顺序是按照线程加锁的顺序来，即FIFO先进先出

非公平锁就是一种获取锁的强占机制，是随机的

如何设置公平和非公平的获取机制呢？

在new ReentrantLock(boolean) true 为公平 false 为非公平

Lock.getHoldCount() 当前线程保持此锁定的个数

Lock.getQueueLength() 等待线程个数

Lock.getWaitQueueLength() 等待某一个Condition的线程数

Lock.hasQueuedThread(Thread) 查询指定的线程是否在等待获取此锁定

Lock.hasWaiters(Condition) 查询是否有线程正在等待与此锁定有关的condition

Lock.isFair() 判断是不是公平锁

Lock.isHeldByCurrentThread() 查询当前线程是否保持此锁定

Lock.isLocked() 是否被锁定

Lock.tryLock() 判断是否获得锁

Lock.tryLock(3, TimeUnit.SECONDS) 指定时间没获得锁，当前线程将被中断

#### ReentrantReadWriteLock类的使用

此类是读写锁，使用它可以加快运行效率，

在不需要操作示例变量的方法中，使用它可以提升代码运行效率

读操作锁--共享锁

写操作锁--排他锁

多个读锁之间不互斥

读锁和写锁、写锁和写锁都互斥

Lock.readLock().lock() 获取读锁

Lock.readLock().unlock() 解锁

Lock.writeLock().lock() 获取写锁

Lock.writeLock().unLock() 解锁

#### 与synchronized的区别

看java基础

### 定时器Timer的使用

**是什么**

负责计划任务，在指定时间开始执行某一任务

Timer 作为执行者

TimerTask 作为被执行的一个线程

**怎么用**

Timer.schedule(TimerTask, Date) 指定日期之后执行

TimerTask是以队列的方式一个一个呗顺序执行的，如果前面耗时太长，那么可能会导致后面延迟执行

Timer.schedule(TimerTask, Date, long period) 指定日期之后执行，并按指定的时间间隔循环执行

TimerTask.cancel() 将自身从人物队列中清除

Timer.cancel() 将队列清空

Timer.schedule(TimerTask, long delay) 在当前时间 延迟指定时间执行一次

Timer.schedule(TimerTask, long delay, long period) 在当前时间，延迟delay毫秒数，以period毫秒数为时间间隔循环执行

Timer.secheduleAtFixedRate(TimerTask, Date, period) 和上面方法有些许不同，可以测试进行比较，应该是没有时间延迟 循环执行

**有什么用**

在什么时候使用定时任务，如在Android中，实现类似于轮询动画等常见功能

### 单例模式与多线程

**为什么**

如何使单例模式遇到多线程是安全的、正确的

**怎么做**

#### 立即加载、饿汉模式

-- 使用类的时候就已经创建完毕

Public static MyObject object = new MyObject();

Private MyObject(){}

Public static MyObject getInstance(){

Return object;

}

这么做在多线程的环境下确实是安全的，都是同一个实例

#### 延迟加载、懒汉模式

-- 需要的时候才创建

public class MyObject{

private static MyObject myObject;

private MyObject(){};

public static MyObject getInstance(){

if(myObject == null)

myObject = new MyObject();

return myObject;

}

}

这在多线程并不安全，产生了多个实例。

此处解决这个问题的方法可以是同步方法，同步块，但是为了提升性能，这里采用DCL双检查锁机制

public class MyObject{

private static MyObject myObject;

private MyObject(){};

public static MyObject getInstance(){

if(myObject != null){}

else {

synchronized(MyObject.class){

if(myObject == null){

myObject = new MyObject();

}

}

}

return myObject;

}

}

可以从代码中看到这个是很优秀的，如果使用同步方法，多个线程不管需不需要创建此对象都会进行资源锁的获取，这是耗费cpu资源的，而采用DCL检查锁呢，就需要的时候会同步，其他时候一律异步。

#### 静态内置类

public class MyObject{

private static class MyObjectHandler {

private static MyObject myObject = new MyObject();

}

public static MyObject getInstance(){

return MyObjectHandler.myObject;

}

}

静态内置类也实现了多线程情况下的安全

#### 序列化与反序列化

也能实现线程安全

#### 使用static代码块

#### 使用枚举数据类型

public enum MyObject{

connectionFactory;

private Connection connection;

private MyObject() {

try{

System.out.println("调用了MyObject 的构造");

String url = "";

String username = "";

String password = "";

String driverName = "";

Class.forName("");

connection = DriverManager.getConnection(url, username, password);

}catch(ClassNotFoundException | SQLException e){

e.printStackTrace();

}

}

public Connection getConnection(){

return connection;

}

}

采用枚举类也能达到线程安全，上面代码时最常见的jdbc工厂类的创建，自始至终返回同一个实例。

### 线程生命周期

线程的生命周期

线程的生命周期包含5个阶段，包括：新建、就绪、运行、阻塞、销毁。

**新建**：就是刚使用new方法，new出来的线程；

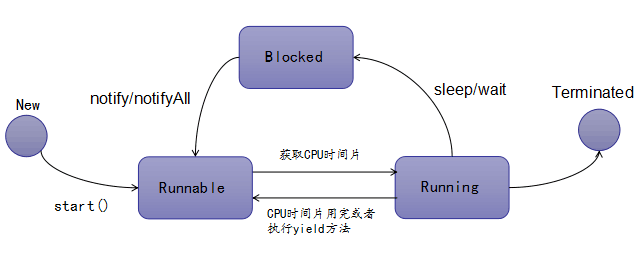
**就绪**：就是调用的线程的start()方法后，这时候线程处于等待CPU分配资源阶段，谁先抢的CPU资源，谁开始执行;

**运行**：当就绪的线程被调度并获得CPU资源时，便进入运行状态，run方法定义了线程的操作和功能;

**阻塞**：在运行状态的时候，可能因为某些原因导致运行状态的线程变成了阻塞状态，比如sleep()、wait()之后线程就处于了阻塞状态，这个时候需要其他机制将处于阻塞状态的线程唤醒，比如调用notify或者notifyAll()方法。唤醒的线程不会立刻执行run方法，它们要再次等待CPU分配资源进入运行状态;

**销毁**：如果线程正常执行完毕后或线程被提前强制性的终止或出现异常导致结束，那么线程就要被销毁，释放资源;

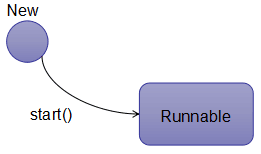
**完整的生命周期图如下**：



#### 新建状态

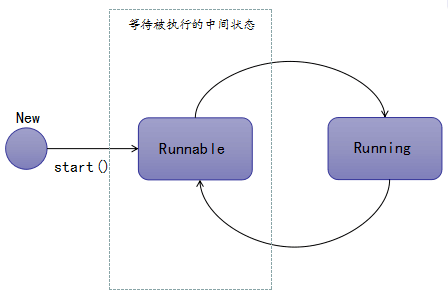
Thread t1 = new Thread();

以上代码代表Thread被创建，而在操作系统层面，真正的线程还没有被创建。只有当我们调用了 start() 方法之后，该线程才会被创建出来，进入Runnable状态。只有当我们调用了 start() 方法之后，该线程才会被创建出来。

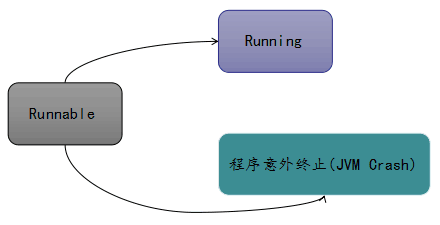


#### 就绪状态

调用start()方法后，JVM 进程会去创建一个新的线程，而此线程不会马上被 CPU 调度运行，进入Running状态，这里会有一个中间状态，就是Runnable状态，你可以理解为等待被 CPU 调度的状态



那么处于Runnable状态的线程能发生哪些状态转变？

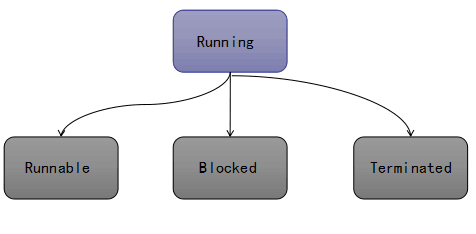


Runnable状态的线程无法直接进入Blocked状态和Terminated状态的。只有处在Running状态的线程，换句话说，只有获得CPU调度执行权的线程才有资格进入Blocked状态和Terminated状态，Runnable状态的线程要么能被转换成Running状态，要么被意外终止。

#### 运行状态

当CPU调度发生，并从任务队列中选中了某个Runnable线程时，该线程会进入Running执行状态，并且开始调用run()方法中逻辑代码。

那么处于Running状态的线程能发生哪些状态转变？



* 被转换成Terminated状态，比如调用 stop() 方法;
* 被转换成Blocked状态，比如调用了sleep, wait 方法被加入 waitSet 中；
* 被转换成Blocked状态，如进行 IO 阻塞操作，如查询数据库进入阻塞状态；
* 被转换成Blocked状态，比如获取某个锁的释放，而被加入该锁的阻塞队列中；
* 被转换成Runnable状态，该线程的时间片用完，等待CPU 再次调度；
* 被转换成Runnable状态，线程主动调用 yield 方法，让出 CPU 资源；

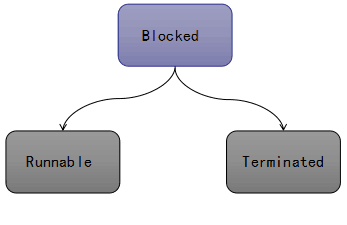
#### 阻塞状态

如果一个线程执行了sleep（睡眠）、suspend（挂起）等方法，失去所占用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。

**阻塞可以分为三种**

* 等待阻塞：运行状态中的线程执行 wait() 方法，使线程进入到等待阻塞状态
* 同步阻塞：线程在获取 synchronized 同步锁失败(因为同步锁被其他线程占用)。
* 其他阻塞：通过调用线程的 sleep() 或 join() 发出了 I/O 请求时，线程就会进入到阻塞状态。当sleep() 状态超时，join() 等待线程终止或超时，或者 I/O 处理完毕，线程重新转入就绪状态。

**Blocked状态的线程能够发生哪些状态改变？**



* 被转换成Terminated状态，比如调用 stop() 方法，或者是 JVM 意外 Crash;
* 被转换成Runnable状态，阻塞时间结束，比如读取到了数据库的数据后；
* 完成了指定时间的休眠，进入到Runnable状态；
* 正在wait中的线程，被其他线程调用notify/notifyAll方法唤醒，进入到Runnable状态；
* 线程获取到了想要的锁资源，进入Runnable状态；
* 线程在阻塞状态下被打断，如其他线程调用了interrupt方法，进入到Runnable状态；

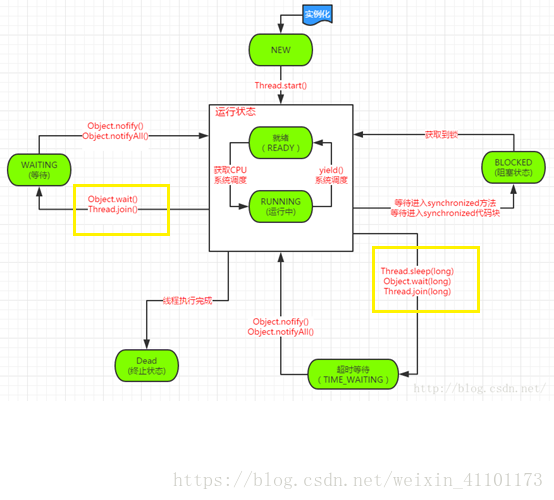
#### 终止状态

一旦线程进入了Terminated状态，就意味着这个线程生命的终结，哪些情况下，线程会进入到Terminated状态呢？

* 线程正常运行结束，生命周期结束；
* 线程运行过程中出现意外错误；

JVM 异常结束，所有的线程生命周期均被结束。

#### Sleep和wait方法的区别



**Sleep和wait的阻塞场景图**

①sleep()实现线程阻塞的方法，我们称之为“**线程睡眠**”，方式是超时等待，怎么理解？就是sleep()通过传入“**睡眠时间**”作为方法的参数，时间一到就从“睡眠”中“醒来”；

②**wait()方法实现线程阻塞的方法**，我们称之为“**线程等待**”，方式有两种：

1）和sleep()方法一样，通过传入“睡眠时间”作为参数，时间到了就“醒了”；

1. 不传入时间，进行一次“无限期的等待”，只用通过notify()方法来“唤醒”。

通过上面图的展示和笔者的相关描述，相信读者你已经有几分清楚了，sleep()和wait()方法的区别之一，就是实现线程阻塞的方式不一样。

那么sleep()和wait()二者还有一个很大的区别就是，二者“**是否释放同步锁**”不一样。我们都知道，多线程开发中，为了实现不同线程间的同步会采用同步锁的方式——synchronized即在线程使用一个资源时为其加锁，这样其他的线程便不能访问那个资源了，直到解锁后才可以访问。感兴趣的读者可以参考笔者之前文章《利用synchronized实现线程同步》里面详细介绍了多线程开发中利用synchronized实现线程同步的方式。

而使用sleep()和wait()两种方法对于“CPU执行权”和“同步锁”的方式不同：

**①sleep()释放CPU执行权，但不释放同步锁；**

**②wait()释放CPU执行权，也释放同步锁，使得其他线程可以使用同步控制块或**者方法。

以上，就是sleep()和wait()方法的**两个关键性区别**。

总结：综上我们利用下表展示sleep()和wait()的所有区别：