多线程概述

2018年8月25日 11:38

1. 进程和线程

a. 进程

所谓的进程其实就是一个程序或服务运行的过程在操作系统中的体现，操作系统中一个独立运行的程序或服务就是一个进程。

b. 多进程

现代的操作系统都可以支持同时运行多个程序和服务，体现在进程上，就是多个进程可以并行的执行，这称之为操作系统支持多进程。

c. 线程

一个进程的内部还可以划分出多个并行执行的过程，称之为在进程内部存在线程。

d. 多线程

一个进程内部可以存在多个线程，且这些线程可以并行的执行，这样的机制称之为进程支持多线程。

**计算中只有一个cpu，实际上同一个时刻，只能处理一个运算，但是由于cpu的运算速度非常的快，通过不停的切换处理的任务，从而使多个进程 多个线程 依次都能得到处理，处理的很快，切换的也很快，在人看来似乎这些进程和线程都在同时执行

**jvm本身也是进程，可以在其中开辟多个线程执行并发的任务，这样的开发多个线程的技术就称之为java的多线程技术。

Java中的多线程

2018年8月25日 14:08

1. Thread类

java是面向对象的语言，万物皆对象，在java中也是用对象来代表底层的物理线程，来方便我们操作。这样的线程对象归属于Thread类。

java.lang

类 Thread

2. 重要API

构造方法：

构造方法摘要
Thread() 分配新的 Thread 对象。
Thread(Runnable target) 分配新的 Thread 对象。

重要方法：

void	**start**() 使该线程开始执行；Java 虚拟机调用该线程的 run 方法。

3. 启动线程的两种方式

   a. 启动线程的方式1：

      i. 创建一个类 实现Runnable接口 在其中的run()方法中编写启动的线程要执行的代码

ii. 创建改类的对象，传入Thread的构造方法 ，创建Thread对象
iii. 调用Thread对象的start()方法，启动线程

b. 启动线程的方式2：
   i. 写一个类继承Thread,覆盖其中的run()方法，在其中编写启动的线程要执行的代码
   ii. 创建该Thread类的子类的对象
   iii. 调用Thread类的子类的对象的start()方法，启动线程

**两种线程启动方式的比较：

java是单继承的，继承的方式创建线程，将会占用了extends关键字，这在类本身需要继承其他类的情况下无法使用

java是多实现的，实现接口的数量没有线程，所以实现接口创建线程的方式并不会受到单继承的线程

4. 线程并发的细节

a. 主线程和其他线程比起来，唯一特殊的地方是它是程序的入口，除此之外没有任何高低 先后 差别。
b. 多个线程并发的过程中，线程在不停的无序的争夺cpu，某一时刻哪个线程抢夺到，哪个线程就执行，由于cpu运行速度非常快，看起来似乎这些线程都在并发的执行。
c. 线程是在进程内部执行的，进程内部只有任意一个非守护线程存活，进程就不会结束。

5. 关闭线程

stop方法可以显示的命令线程立即停止，但此方法具有固有的不安全性，所以目前已经被过时 废弃掉了。不要去使用。

void	**stop**() **已过时。** *该方法具有固有的不安全性。*

在废弃掉stop方法之后，jdk中也没有提供任何其他类似的方法，官方的建议是，应该由程序本身提供相应的开关，来控制线程的运行和停止。通常由一个静态的布尔类型来作为这种开关。

6. Thread中的其他常用方法

线程优先级相关的字段，本质上是int类型的常量值，取值范围为1 - 10 ，值越大优先级越高

字段摘要	
static int	**MAX_PRIORITY** 线程可以具有的最高优先级。
static int	**MIN_PRIORITY** 线程可以具有的最低优先级。
static int	**NORM_PRIORITY** 分配给线程的默认优先级。

static Thread	**currentThread**() 返回对当前正在执行的线程对象的引用。
long	**getId**() 返回该线程的标识符。
String	**getName**() 返回该线程的名称。
void	**setName**(String name) 改变线程名称，使之与参数 name 相同。
int	**getPriority**() 返回线程的优先级。
void	**setPriority**(int newPriority)

	更改线程的优先级。
Thread.State	**getState**() 返回该线程的状态。
static void	**sleep**(long millis) 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。

Java的多线程并发安全问题

2018年8月25日 15:30

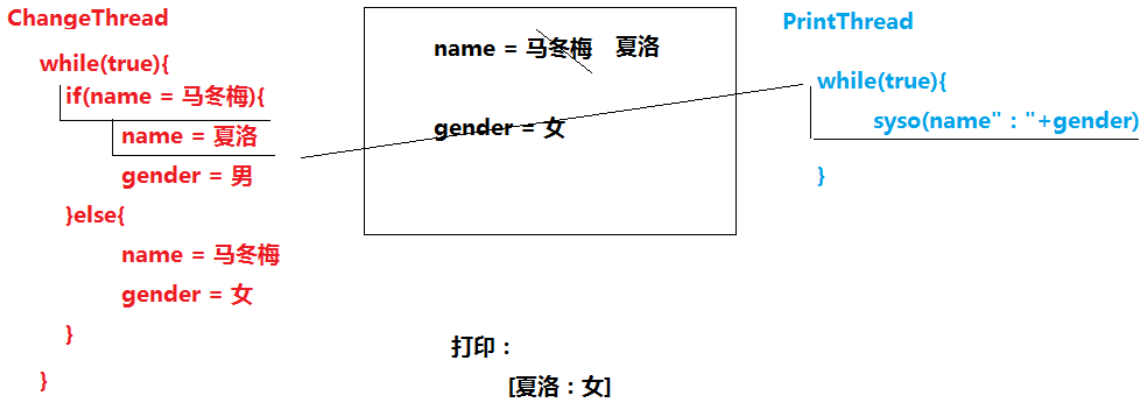
1. 多线程并发安全问题

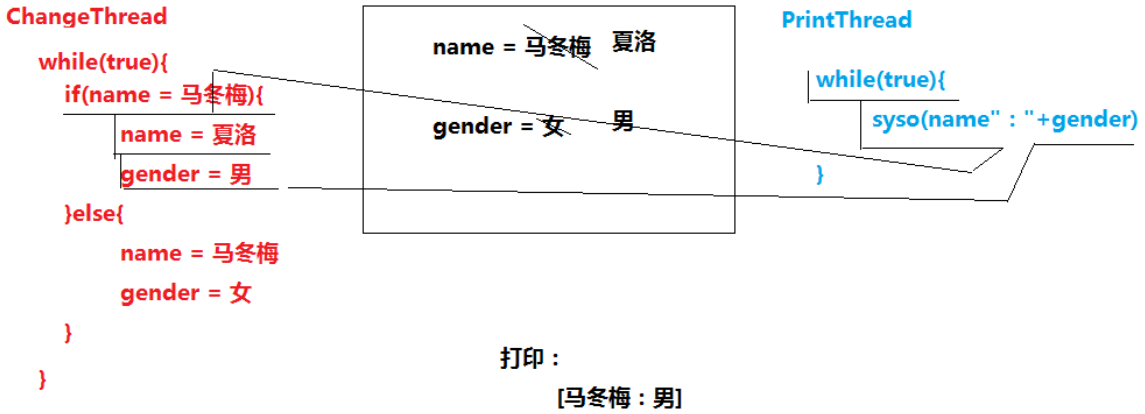
多线程环境下，多个线程是并发执行的，并且基于无序的cpu的争夺机制，线程的执行顺序是不确定的，此时如果多个线程同时去操作共享资源，就有可能因为线程的无序执行，产生一些意外的情况，这种问题就统称为多线程并发安全问题。

案例 - 修改并打印：

```
package cn.tedu.thread;

/**
 * 多线程并发安全问题
 */
public class ThreadDemo05 {
    public static String name = "马冬梅";
    public static String gender = "女";

    public static void main(String[] args) {
        new Thread(new PrintThread()).start();
        new Thread(new ChangeThread()).start();
    }
}

class ChangeThread implements Runnable{
    @Override
    public void run() {
        while(true){
            if("马冬梅".equals(ThreadDemo05.name)){
                ThreadDemo05.name = "夏洛";
                ThreadDemo05.gender = "男";
            }else{
                ThreadDemo05.name = "马冬梅";
                ThreadDemo05.gender = "女";
            }
```

```
            }
        }
    }

    class PrintThread implements Runnable{
        @Override
        public void run() {
            while(true){
                System.out.println("姓名："+ThreadDemo05.name+",性
                别："+ThreadDemo05.gender);
            }
        }
    }
```

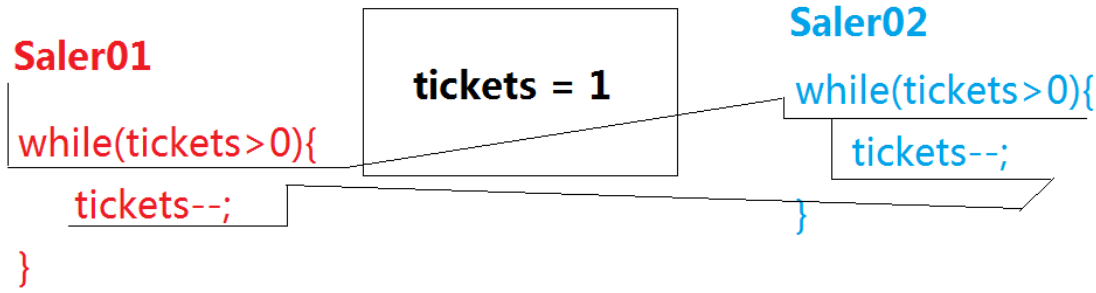




案例 - 卖火车票：

```
    package cn.tedu.thread;

    public class ThreadDemo06 {
        public static int tickets = 10;
        public static void main(String[] args) {
            new Thread(new Saler()).start();
```

```
            new Thread(new Saler()).start();
        }
    }

    class Saler implements Runnable{
        @Override
        public void run() {
            while(ThreadDemo06.tickets>0){
                ThreadDemo06.tickets--;
                System.out.println(Thread.currentThread().getId()+"窗口，卖出去了
                一张票。。剩余票数为"+ThreadDemo06.tickets);
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```



2. 多线程并发安全问题产生的条件

a. 有共享资源

b. 有多线程并发操作了共享资源

c. 有多线程并发操作了共享资源 且涉及到了修改操作

3. 解决多线程并发安全问题

解决多线程并发安全问题的关键，就是破坏产生多线程并发安全问题的条件

禁止共享资源 -- ThreadLocal

禁止多线程并发操作 -- Synchronized

禁止修改 -- ReadWriteLock

4. Synchronized同步代码块使用

原理：

通过控制并发线程无法同时操作共享资源，从而实现线程安全问题的解决。

细节：

在存在多线程并发安全问题的场景下，可以使用synchronized代码块，将产生多线程并发安全问题的代码包裹起来，并选择一个锁对象。

锁对象可以任意的选择，但是要保证，多个并发的线程操作的都是同一个锁对象。

锁对象并没有实际的作用，唯一的作用是在锁对象身上会标记一个锁的状态 - [关锁状态] [开锁状态]

在线程执行到同步代码块时，会在当前锁对上上检查，如果是锁是[开锁状态]，则进入同步代码块，同时将锁状态改为[关锁状态]

此时再有其他线程试图获取该对象上的锁时，锁已经是[关锁状态]，无法获得到锁，线程只能在锁下等待，进入阻塞状态。

直到之前得到锁的线程将synchronized代码块执行完，释放锁，重新将锁变为 [开锁状态]，等待的线程，才可以从阻塞状态中恢复，重新参与锁的争夺。

因此，同一时刻，只能有一个并发线程执行可能造成线程安全问题的代码，破坏了多线程并发的状态，从而解决多线程并发安全问题

同步代码块的结构：

单独声明同步代码块：

```
synchronized(锁对象){
    要同步的代码
}
```

在方法上声明同步，则称之为同步方法：

```
Public synchronized void mx(){
    ...
```

}

同步方法中的锁对象，普通方法是this,静态方法时当前类的字节码

锁对象的选择的原则：

任意对象都可以作为锁对象使用

但要保证，所有的并发线程都应该能够访问到该所对象，且访问的必须是同一个锁对象

常用的锁对象：

自己创建一个专用的对象

使用共享资源作为锁对象

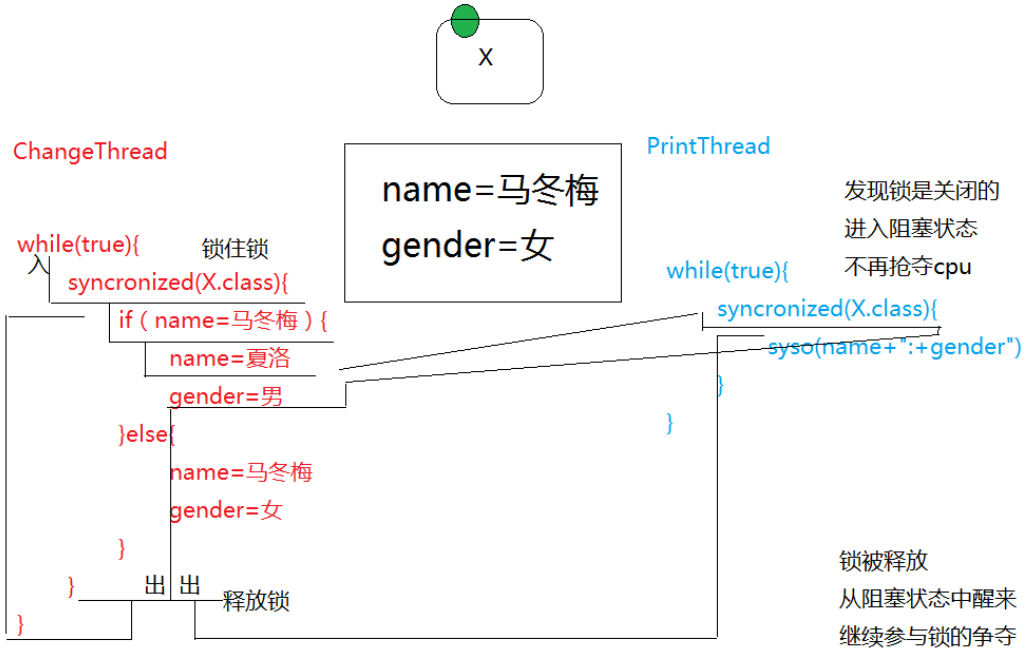
使用类的字节码对象作为锁对象

案例：利用同步代码块 解决上面的 修改打印案例中的 线程并发安全问题

代码：

```
package cn.tedu.thread;

/**
 * 多线程并发安全问题
 */
public class ThreadDemo05 {
    public static String name = "马冬梅";
    public static String gender = "女";

    public static void main(String[] args) {
        new Thread(new PrintThread()).start();
        new Thread(new ChangeThread()).start();
    }
}

class ChangeThread implements Runnable{
    @Override
    public void run() {
        while(true){
            synchronized (ThreadDemo05.class) {
                if("马冬梅".equals(ThreadDemo05.name)){
```

```
                ThreadDemo05.name = "夏洛";
                ThreadDemo05.gender = "男";
            }else{
                ThreadDemo05.name = "马冬梅";
                ThreadDemo05.gender = "女";
            }
        }
    }
}

class PrintThread implements Runnable{
    @Override
    public void run() {
        while(true){
            synchronized (ThreadDemo05.class) {
                System.out.println("姓名：" + ThreadDemo05.name + ",
                性别：" + ThreadDemo05.gender);
            }
        }
    }
}
```



案例：利用同步代码块 解决上面的 卖火车票案例中的 线程并发安全问题

代码：

```
package cn.tedu.thread;
```

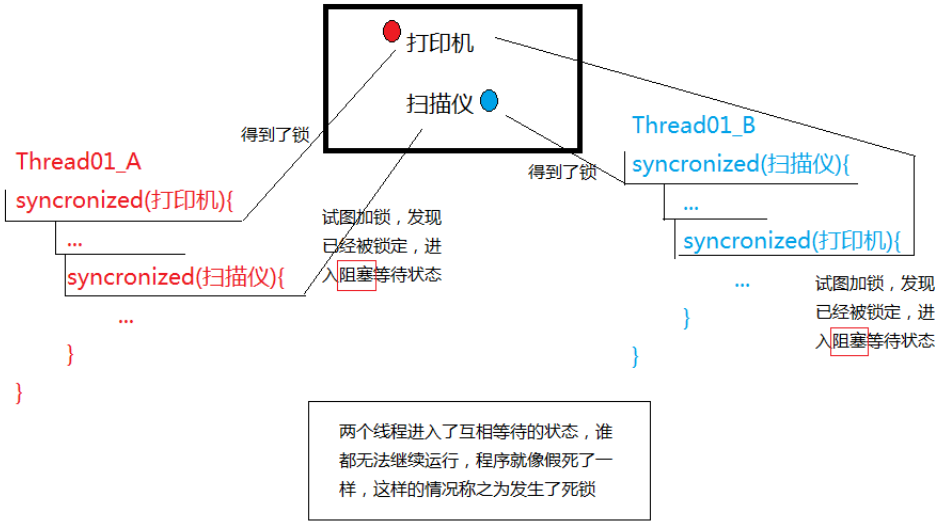
```
public class ThreadDemo06 {
    public static int tickets = 10;
    public static void main(String[] args) {
        new Thread(new Saler()).start();
        new Thread(new Saler()).start();
    }
}

class Saler implements Runnable{
    @Override
    public void run() {
        while(ThreadDemo06.tickets>0){
            synchronized (ThreadDemo06.class) {
                ThreadDemo06.tickets--;
                System.out.println(Thread.currentThread().getId() + "窗
                口，卖出去了一张票。。剩余票数为" +
                ThreadDemo06.tickets);
            }
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

死锁问题

2018年8月27日 9:12

1. 死锁概述

死锁是一种并发锁定的特殊状态，指的是，当具有多个共享资源时 一部分线程持有一部分资源的锁 要求另外的线程持有的另外的资源的锁 形成了各自持有各自的锁而要求对方的锁的状态 这样 进入了一个互相等待的状态 都无法继续执行 则称之为产生了死锁



死锁并不是一种真正的锁，而是一种特殊状态，会造成程序无法继续运行或退出，所以要尽力的解决死锁

案例：

```
package cn.tedu.thread;

/**
 * 死锁
 */
class DYJ{}
class SMY{}

public class Demo01 {
    public static DYJ dyj = new DYJ();
    public static SMY smy = new SMY();

    public static void main(String[] args) {
        new Thread(new T01_A()).start();
```

```
            new Thread(new T01_B()).start();
        }
    }
    class T01_B implements Runnable{
        @Override
        public void run() {
            try{
                synchronized (Demo01.smy) {
                    System.out.println("B使用扫描仪"+Demo01.smy);
                    Thread.sleep(3000);
                    synchronized (Demo01.dyj) {
                        System.out.println("B使用打印机"+Demo01.dyj);
                        Thread.sleep(2000);
                    }
                }

            }catch( Exception e){
                e.printStackTrace();
            }
        }
    }
    class T01_A implements Runnable{
        @Override
        public void run() {
            try {
                synchronized (Demo01.dyj) {
                    System.out.println("A使用打印机"+Demo01.dyj);
                    Thread.sleep(3000);
                    synchronized (Demo01.smy) {
                        System.out.println("A使用扫描仪"+Demo01.smy);
                        Thread.sleep(2000);
                    }
                }
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

2. 死锁产生的条件

a. 多把锁
b. 多个线程
c. 同步嵌套

在Synchronized代码块中再包含Synchronized代码块，这就意味着，占用着一部分锁，再要求另一部分锁

3. 解决死锁

避免死锁

避免同步嵌套来避免死锁的产生

检测并打断死锁

有时无法进行避免死锁的操作，此时只能不停的检测是否有死锁产生，如果有死锁产生，则打断死锁，所谓的打断死锁，就是将造成死锁的某一线程错误退出，打断对锁互相要求的环，从而使程序可以正常运行下去。

线程之间的通信

2018年8月27日 9:52

1. 线程间通信概述

在程序运行过程中，线程是相对独立的单位，多个线程之间并行的执行，并不会有太多的沟通，每个线程都有属于自己的内存空间，且无法互相访问，所以可以认为多个线程之间是隔离的状态，并没有过多的信息传递。

而线程在并发运行的过程中，还会无序抢夺cpu，造成执行的顺序不确定，使执行的结果变得不可预期

有时我们希望能够实现 多个线程之间进行信息的传递 或 执行过程的协调 ，这样的技术称之为线程间的通信技术

线程间的通信技术：

共享内存机制 - 多个线程之间进行信息的传递

等待唤醒机制 - 多个线程执行过程的协调

2. 线程间的通信机制 - 共享内存机制 - 多个线程之间的信息传递

每个线程都各自有各自的内存空间，且无法互相访问，当多个线程需要进行信息的共享时，可以在多个线程都可以看到的公共的内存中保存数据，从而实现两者的通信

案例：某一个线程通过控制一个布尔类型的信号量 控制另一个线程执行的流程

```
package cn.tedu.thread;

/**
 * 线程间的通信 - 共享内存方式传递信息
 */
public class Demo03 {
    public static boolean canRun = true;
```

```
    public static void main(String[] args) {
        new Thread(new Thread03_Master()).start();
        new Thread(new Thread03_Slave()).start();
    }
}

class Thread03_Master implements Runnable{
    @Override
    public void run() {
        try {
            Thread.sleep(2000);
            Demo03.canRun = false;
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
class Thread03_Slave implements Runnable{
    @Override
    public void run() {
        while(Demo03.canRun){
            System.out.println("小弟在运行。。。");
        }
        System.out.println("小弟告辞了。。。");
    }
}
```

3. 线程之间的通信技术 - 等待唤醒机制 - 协调线程执行的先后顺序

多个线程在并发的过程中，互相抢夺cpu，造成执行的先后顺序是不确定的，如果需要控制线程的执行先后顺序，可以采用等待唤醒的机制

在锁对象身上，具有等待、唤醒的方法，这些方法其实是定义在Object类上的，所以任意对象作为锁对象时，都可以有等待 唤醒的方法使用

void	**wait**() 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。
void	**wait**(long timeout)

	在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。
void	**wait**(long timeout, int nanos) 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。
void	**notify**() 唤醒在此对象监视器上等待的单个线程。
void	**notifyAll**() 唤醒在此对象监视器上等待的所有线程。

wait()

在Synchronized代码块中，调用锁对象的wait方法将会使当前线程进入进入阻塞状态 不再争夺cpu 释放锁 阻塞的状态会一直持续下去 直到被其他线程 调用锁对象的notify()或notifyAll()方法唤醒

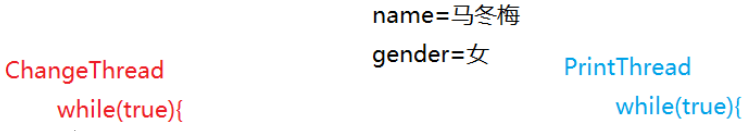
notify()

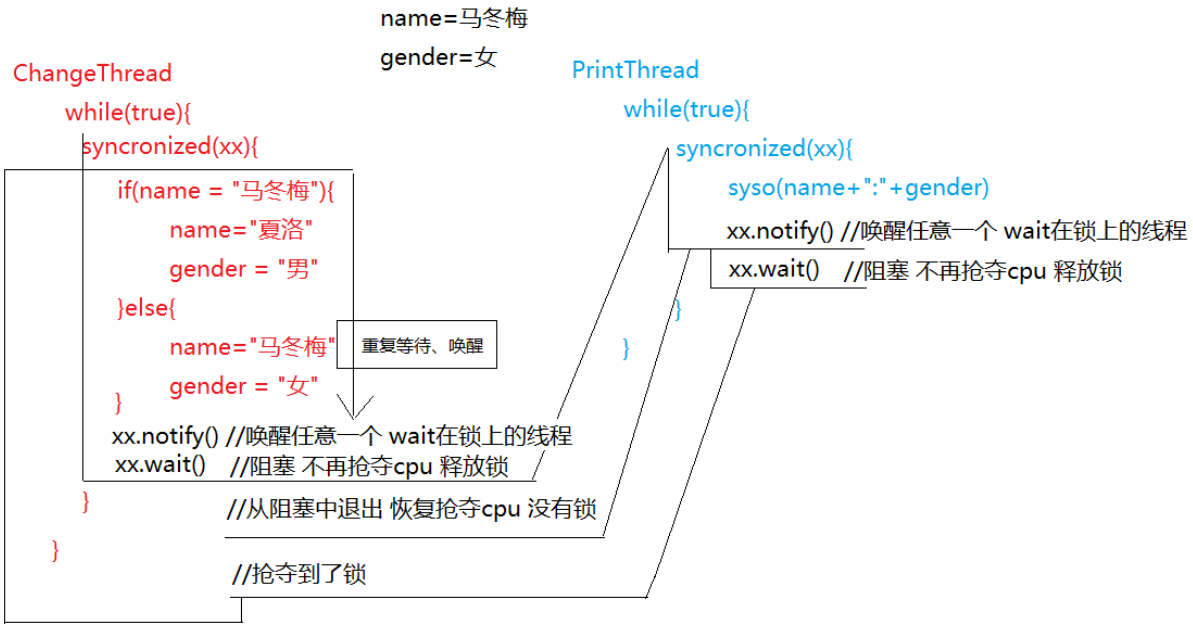
在Synchronized代码块中，调用锁对象的notify()方法，将会唤醒之前在这个锁对象上进入wait()状态的某一个线程，使其退出阻塞状态，退出阻塞状态后，恢复对cpu的争夺，但仍然需要得到锁，才可以继续运行。至于唤醒的是哪一个线程是不确定的

notifyAll()

在Synchronized代码块中，调用锁对象的notifyAll()方法，将会唤醒之前在这个锁对象上进入wait()状态的所有线程，使这些线程退出阻塞状态，退出阻塞状态后，恢复对cpu的争夺，但仍然需要得到锁，才可以继续运行。

案例：修改之前的修改打印案例，实现修改和打印依次执行






```
package cn.tedu.thread;

/**
 * 多线程并发安全问题
 */
public class Demo02 {
    public static String name = "马冬梅";
    public static String gender = "女";

    public static void main(String[] args) {
        new Thread(new PrintThread()).start();
        new Thread(new ChangeThread()).start();
    }
}

class ChangeThread implements Runnable{
    @Override
    public void run() {
        try {
            while(true){
                synchronized (Demo02.class) {
                    if ("马冬梅".equals(Demo02.name)) {
                        Demo02.name = "夏洛";
                        Demo02.gender = "男";
                    } else {
                        Demo02.name = "马冬梅";
```

```
                    Demo02.gender = "女";
                }
                //唤醒
                Demo02.class.notify();
                //等待
                Demo02.class.wait();
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}

class PrintThread implements Runnable{
    @Override
    public void run() {
        try {
            while(true){
                synchronized (Demo02.class) {
                    System.out.println("姓名：" + Demo02.name + ",性
                    别：" + Demo02.gender);
                    Demo02.class.notify();
                    Demo02.class.wait();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

案例：利用等待唤醒机制 实现一个阻塞式队列

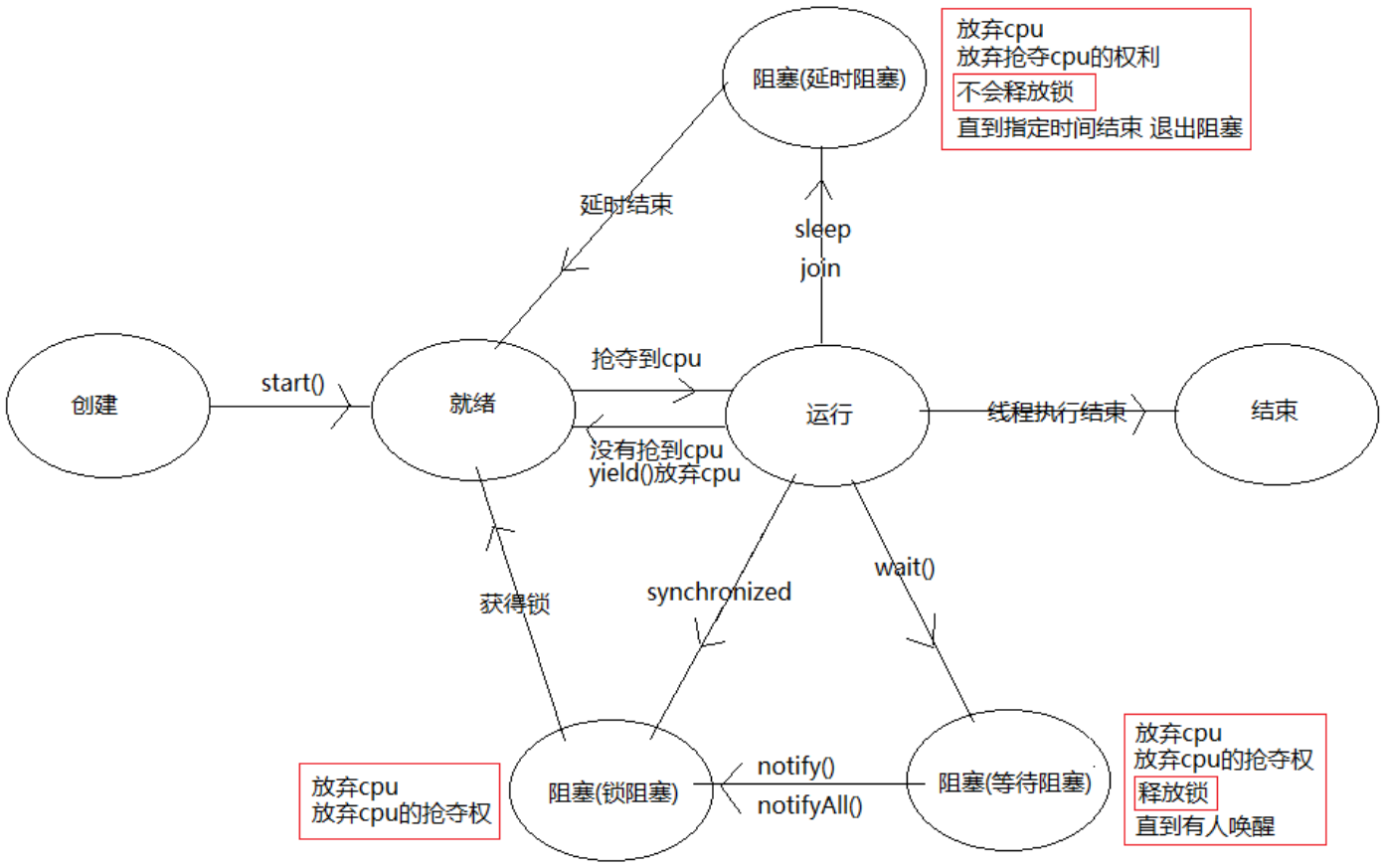
```
package cn.tedu.thread;

import java.util.LinkedList;

/**
 * 等待唤醒机制 案例 - 实现阻塞式队列
 * @author Administrator
```

```
 *
 */
class MyBlockingQueue{
    private LinkedList queue = new LinkedList<>();
    private int MaxSize = 0;
    public MyBlockingQueue(int size) {
        this.MaxSize = size;
    }

    //--如果满了还要加 要阻塞当前操作线程
    public synchronized void add(Object obj){
        try {
            if(queue.size() >= this.MaxSize){
                //--不能往里加，阻塞当前线程，直到有人取走，队列变为非满
                this.wait();
            }
            queue.add(obj);
            this.notify();
        } catch (InterruptedException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }

    //--如果空了还要取 要阻塞当前操作线程
    public synchronized Object take(){
        try {
            if(queue.size() <=0 ){
                this.wait();
            }
            Object obj = queue.poll();
            this.notify();
            return obj;
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }
}
```

```
public class Demo04 {
    public static void main(String[] args) {
        MyBlockingQueue bq = new MyBlockingQueue(3);
        new Thread(new Runnable() {
            @Override
            public void run() {
                bq.add("aaa");
                bq.add("bbb");
                bq.add("ccc");
                bq.add("ddd");
                bq.add("eee");
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(bq.take());
                System.out.println(bq.take());
                System.out.println(bq.take());
                System.out.println(bq.take());
                System.out.println(bq.take());
            }
        }).start();
    }
}
```

线程之间的状态转换

2018年8月27日 11:24



多线程-其他方法

2018年8月27日 11:40

1. setDaemon()

线程从是否可以独立执行的角度，可以分为用户线程和守护线程

通常创建出来的线程都是用户线程，但可以通过setDaemon(true)将用户线程转换为守护线程

用户线程可以独立运行 守护线程不可以 也就是说 如果进程中 已经没有用户线程 只剩下守护线程 则守护线程会自动退出

void	**setDaemon**(boolean on) 将该线程标记为守护线程或用户线程。
boolean	**isDaemon**() 测试该线程是否为守护线程。

案例：王者荣耀

```
package cn.tedu.thread;

/**
 * 守护线程 - 王者荣耀 游戏开发
 */
public class Demo05 {
    public static void main(String[] args) throws Exception {
        new Thread(new ShuiJing()).start();

        Thread t1 = new Thread(new Hero("安其拉"));
        Thread t2 = new Thread(new Hero("孙悟空"));
        Thread t3 = new Thread(new Hero("诸葛亮"));
        Thread t4 = new Thread(new Hero("吕布"));
        Thread t5 = new Thread(new Hero("亚瑟"));
```

```
            t1.setDaemon(true);
            t2.setDaemon(true);
            t3.setDaemon(true);
            t4.setDaemon(true);
            t5.setDaemon(true);

            t1.start();
            t2.start();
            t3.start();
            t4.start();
            t5.start();

            while(ShuiJing.blood>0){
                Thread.sleep(1000);
                ShuiJing.blood -= 8;
            }
        }
    }

    class Hero implements Runnable{
        private String name = null;
        public Hero(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            while(true){
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(name+"说：为了水晶，冲啊~~~碾碎
                他们。。");
            }
        }
```

```
}

class ShuiJing implements Runnable{
	public static int blood = 100;
	@Override
	public void run() {
		while(this.blood>0){
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println("全军出击，守卫水晶。。血量剩余"+blood);
		}
		System.out.println("水晶被打爆了。。游戏结束。。。");
	}
}
```

2. join()

在当前线程执行时可以调用另一个线程的join方法 ，则当前线程立即进入阻塞模式，直到被join进来的线程执行完成为止

void	**join**() 等待该线程终止。

案例:

```
package cn.tedu.thread;

/**
 * join
 * @author Administrator
 *
 */
public class Demo06 {
	public static void main(String[] args) throws Exception {
```

```
            System.out.println("上数学课。。。");
            System.out.println("上数学课。。。");
            System.out.println("上数学课。。。");
            System.out.println("上数学课。。。");
            Thread t = new Thread(new Thread_06());
            t.start();
            t.join();//将进入阻塞模式 直到 t执行完成
            System.out.println("上数学课。。。");
            System.out.println("上数学课。。。");
            System.out.println("上数学课。。。");
            System.out.println("上数学课。。。");
        }
    }

    class Thread_06 implements Runnable{
        @Override
        public void run() {
            System.out.println("做广播体操开始。。。");
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("做广播体操结束。。。");
        }
    }
```

3. yield()

static void	**yield**() 暂停当前正在执行的线程对象，并执行其他线程。