I.线程与线程间通信

一、基本概念以及线程与进程之间的区别联系：

关于进程和线程，首先从定义上理解就有所不同  
  
1、进程是什么？  
是具有一定独立功能的程序、它是系统进行资源分配和调度的一个独立单位，重点在系统调度和单独的单位，也就是说进程是可以独 立运行的一段程序。  
2、线程又是什么？  
线程进程的一个实体，是CPU调度和分派的基本单位，他是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源。  
在运行时，只是暂用一些计数器、寄存器和栈 。  
  
他们之间的关系  
  
1、一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程（通常说的主线程）。  
2、资源分配给进程，同一进程的所有线程共享该进程的所有资源。  
3、线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。  
4、处理机分给线程，即真正在处理机上运行的是线程。  
5、线程是指进程内的一个执行单元，也是进程内的可调度实体。   
  
从三个角度来剖析二者之间的区别  
1、调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位。  
2、并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可以并发执行。  
3、拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。

二、多线程间通信方式：

1、共享变量

2、wait/notify机制

3、Lock/Condition机制

4、管道

三、共享变量

线程间发送信号的一个简单方式是在共享对象的变量里设置信号值。线程A在一个同步块里设置boolean型成员变量hasDataToProcess为true，线程B也在同步块里读取hasDataToProcess这个成员变量。这个简单的例子使用了一个持有信号的对象，并提供了set和check方法:

**[java]** [view plain](http://blog.csdn.net/ls5718/article/details/51878770) [copy](http://blog.csdn.net/ls5718/article/details/51878770)

1. **public** **class** MySignal{
3. **protected** **boolean** hasDataToProcess = **false**;
5. **public** **synchronized** **boolean** hasDataToProcess(){
6. **return** **this**.hasDataToProcess;
7. }
9. **public** **synchronized** **void** setHasDataToProcess(**boolean** hasData){
10. **this**.hasDataToProcess = hasData;
11. }
13. }

线程A和B必须获得指向一个MySignal共享实例的引用，以便进行通信。如果它们持有的引用指向不同的MySingal实例，那么彼此将不能检测到对方的信号。需要处理的数据可以存放在一个共享缓存区里，它和MySignal实例是分开存放的。

四、wait()/notify机制

为了实现线程通信，我们可以使用Object类提供的wait()、notify()、notifyAll()三个方法。调用wait()方法会释放对该同步监视器的锁定。这三个方法必须由同步监视器对象来调用，这可分成两种情况：

* 对于使用synchronized修饰的同步方法，因为该类的默认实例是(this)就是同步监视器，所以可以直接调用这三使用个方法。
* 对于synchronized修饰的同步代码块，同步监视器是synchronized括号里的对象，所以必须使用该对象调用这三个方法。

假设系统中有两条线程，这两条线程分别代表取钱者和存钱者。现在系统有一种特殊的要求，系统要求存款者和取钱者不断的实现存款和取钱动作，而且要求每当存款者将钱存入指定账户后，取钱者立即将钱取走.不允许存款者两次存钱，也不允许取钱者两次取钱。

我们通过设置一个旗标来标识账户中是否已有存款，有就为true,没有就标为false。具体代码如下:

首先我们定义一个Account类，这个类中有取钱和存钱的两个方法，由于这两个方法可能需要并发的执行取钱、存钱操作，所有将这两个方法都修改为同步方法.(使用synchronized关键字)。

1. **public** **class** Account {
2. **private** String accountNo;
3. **private** **double** balance;
4. //标识账户中是否有存款的旗标
5. **private** **boolean** flag=**false**;
7. **public** Account() {
8. **super**();
9. }
11. **public** Account(String accountNo, **double** balance) {
12. **super**();
13. **this**.accountNo = accountNo;
14. **this**.balance = balance;
15. }
17. **public** **synchronized** **void** draw (**double** drawAmount){
19. **try** {
20. **if**(!flag){
21. **this**.wait();
22. }**else** {
23. //取钱
24. System.out.println(Thread.currentThread().getName()+" 取钱:"+drawAmount);
25. balance=balance-drawAmount;
26. System.out.println("余额 : "+balance);
27. //将标识账户是否已有存款的标志设为false
28. flag=**false**;
29. //唤醒其它线程
30. **this**.notifyAll();
31. }
32. } **catch** (Exception e) {
33. e.printStackTrace();
34. }
35. }

38. **public** **synchronized** **void** deposit(**double** depositAmount){
39. **try** {
40. **if**(flag){
41. **this**.wait();
42. }
43. **else**{
44. System.out.println(Thread.currentThread().getName()+"存钱"+depositAmount);
45. balance=balance+depositAmount;
46. System.out.println("账户余额为："+balance);
47. flag=**true**;
48. //唤醒其它线程
49. **this**.notifyAll();
50. }
51. } **catch** (Exception e) {
52. // TODO: handle exception
53. e.printStackTrace();
54. }
55. }
57. }

接下来创建两个线程类，分别为取钱和存钱线程！

取钱线程类：

**[java]** [view plain](http://blog.csdn.net/yulei_qq/article/details/8978456) [copy](http://blog.csdn.net/yulei_qq/article/details/8978456)

[在CODE上查看代码片](https://code.csdn.net/snippets/507489)

1. **public** **class** DrawThread **implements** Runnable {
3. **private** Account account;
4. **private** **double** drawAmount;

7. **public** DrawThread(Account account, **double** drawAmount) {
8. **super**();
9. **this**.account = account;
10. **this**.drawAmount = drawAmount;
11. }
13. **public** **void** run() {
14. **for**(**int** i=0;i<100;i++){
15. account.draw(drawAmount);
16. }
17. }
18. }

存钱线程类：

**[java]** [view plain](http://blog.csdn.net/yulei_qq/article/details/8978456) [copy](http://blog.csdn.net/yulei_qq/article/details/8978456)

[在CODE上查看代码片](https://code.csdn.net/snippets/507489)

1. **public** **class** depositThread **implements** Runnable{
2. **private** Account account;
3. **private** **double** depositAmount;
5. **public** depositThread(Account account, **double** depositAmount) {
6. **super**();
7. **this**.account = account;
8. **this**.depositAmount = depositAmount;
9. }

12. **public** **void** run() {
13. **for**(**int** i=0;i<100;i++){
14. account.deposit(depositAmount);
15. }
16. }
18. }

最后我们测试一下这个取钱和存钱的操作！

1. **public**  **class** TestDraw {
3. **public** **static** **void** main(String[] args) {
4. //创建一个账户
5. Account account=**new** Account();
6. **new** Thread(**new** DrawThread(account, 800),"取钱者").start();
7. **new** Thread(**new** depositThread(account, 800),"存款者甲").start();
8. **new** Thread(**new** depositThread(account, 800),"存款者乙").start();
9. **new** Thread(**new** depositThread(account, 800),"存款者丙").start();
11. }
13. }

大致的输出结果：

**[java]** [view plain](http://blog.csdn.net/yulei_qq/article/details/8978456) [copy](http://blog.csdn.net/yulei_qq/article/details/8978456)

[在CODE上查看代码片](https://code.csdn.net/snippets/507489)

1. 存款者甲存钱800.0
2. 账户余额为：800.0
3. 取钱者 取钱:800.0
4. 余额 : 0.0
5. 存款者丙存钱800.0
6. 账户余额为：800.0
7. 取钱者 取钱:800.0
8. 余额 : 0.0
9. 存款者甲存钱800.0
10. 账户余额为：800.0
11. 取钱者 取钱:800.0
12. 余额 : 0.0
13. 存款者丙存钱800.0
14. 账户余额为：800.0
15. 取钱者 取钱:800.0
16. 余额 : 0.0
17. 存款者甲存钱800.0
18. 账户余额为：800.0
19. 取钱者 取钱:800.0
20. 余额 : 0.0
21. 存款者丙存钱800.0
22. 账户余额为：800.0
23. 取钱者 取钱:800.0
24. 余额 : 0.0
25. 存款者甲存钱800.0
26. 账户余额为：800.0
27. 取钱者 取钱:800.0
28. 余额 : 0.0
29. 存款者丙存钱800.0
30. 账户余额为：800.0
31. 取钱者 取钱:800.0
32. 余额 : 0.0
33. 存款者甲存钱800.0
34. 账户余额为：800.0
35. 取钱者 取钱:800.0
36. 余额 : 0.0

五、Lock/Condition机制

如何程序不使用synchronized关键字来保持同步，而是直接适用Lock对像来保持同步，则系统中不存在隐式的同步监视器对象，也就不能使用wait()、notify()、notifyAll()来协调线程的运行.

当使用LOCK对象保持同步时，JAVA为我们提供了Condition类来协调线程的运行。关于Condition类，JDK文档里进行了详细的解释.，再次就不啰嗦了。

我们就拿Account类进行稍微的修改 一下吧！

1. **import** java.util.concurrent.locks.Condition;
2. **import** java.util.concurrent.locks.Lock;
3. **import** java.util.concurrent.locks.ReentrantLock;
5. **public** **class** Account {
7. //显示定义Lock对象
8. **private** **final** Lock lock=**new** ReentrantLock();
9. //获得指定Lock对象对应的条件变量
10. **private** **final**  Condition con=lock.newCondition();
12. **private** String accountNo;
13. **private** **double** balance;
14. //标识账户中是否有存款的旗标
15. **private** **boolean** flag=**false**;
17. **public** Account() {
18. **super**();
19. }
21. **public** Account(String accountNo, **double** balance) {
22. **super**();
23. **this**.accountNo = accountNo;
24. **this**.balance = balance;
25. }
27. **public** **void** draw (**double** drawAmount){
29. //加锁
30. lock.lock();
31. **try** {
32. **if**(!flag){
33. //            this.wait();
34. con.await();
35. }**else** {
36. //取钱
37. System.out.println(Thread.currentThread().getName()+" 取钱:"+drawAmount);
38. balance=balance-drawAmount;
39. System.out.println("余额 : "+balance);
40. //将标识账户是否已有存款的标志设为false
41. flag=**false**;
42. //唤醒其它线程
43. //               this.notifyAll();
44. con.signalAll();
45. }
46. } **catch** (Exception e) {
47. e.printStackTrace();
48. }
49. **finally**{
50. lock.unlock();
51. }
52. }

55. **public** **void** deposit(**double** depositAmount){
56. //加锁
57. lock.lock();
58. **try** {
59. **if**(flag){
60. //              this.wait();
61. con.await();
62. }
63. **else**{
64. System.out.println(Thread.currentThread().getName()+"存钱"+depositAmount);
65. balance=balance+depositAmount;
66. System.out.println("账户余额为："+balance);
67. flag=**true**;
68. //唤醒其它线程
69. //                this.notifyAll();
70. con.signalAll();
71. }
72. } **catch** (Exception e) {
73. // TODO: handle exception
74. e.printStackTrace();
75. }**finally**{
76. lock.unlock();
77. }
78. }
80. }

输出结果和上面是一样的！ 只不过这里 显示的使用Lock对像来充当同步监视器,使用Condition对象来暂停指定线程，唤醒指定线程！

六、管道

管道流是JAVA中线程通讯的常用方式之一，基本流程如下：

1）创建管道输出流PipedOutputStream pos和管道输入流PipedInputStream pis

2）将pos和pis匹配，pos.connect(pis);

3）将pos赋给信息输入线程，pis赋给信息获取线程，就可以实现线程间的通讯了

**[java]** [view plain](http://blog.csdn.net/ls5718/article/details/51878770) [copy](http://blog.csdn.net/ls5718/article/details/51878770)

1. **import** java.io.IOException;
2. **import** java.io.PipedInputStream;
3. **import** java.io.PipedOutputStream;
5. **public** **class** testPipeConnection {
7. **public** **static** **void** main(String[] args) {
8. /\*\*
9. \* 创建管道输出流
10. \*/
11. PipedOutputStream pos = **new** PipedOutputStream();
12. /\*\*
13. \* 创建管道输入流
14. \*/
15. PipedInputStream pis = **new** PipedInputStream();
16. **try** {
17. /\*\*
18. \* 将管道输入流与输出流连接 此过程也可通过重载的构造函数来实现
19. \*/
20. pos.connect(pis);
21. } **catch** (IOException e) {
22. e.printStackTrace();
23. }
24. /\*\*
25. \* 创建生产者线程
26. \*/
27. Producer p = **new** Producer(pos);
28. /\*\*
29. \* 创建消费者线程
30. \*/
31. Consumer1 c1 = **new** Consumer1(pis);
32. /\*\*
33. \* 启动线程
34. \*/
35. p.start();
36. c1.start();
37. }
38. }
40. /\*\*
41. \* 生产者线程(与一个管道输入流相关联)
42. \*
43. \*/
44. **class** Producer **extends** Thread {
45. **private** PipedOutputStream pos;
47. **public** Producer(PipedOutputStream pos) {
48. **this**.pos = pos;
49. }
51. **public** **void** run() {
52. **int** i = 0;
53. **try** {
54. **while**(**true**)
55. {
56. **this**.sleep(3000);
57. pos.write(i);
58. i++;
59. }
60. } **catch** (Exception e) {
61. e.printStackTrace();
62. }
63. }
64. }
66. /\*\*
67. \* 消费者线程(与一个管道输入流相关联)
68. \*
69. \*/
70. **class** Consumer1 **extends** Thread {
71. **private** PipedInputStream pis;
73. **public** Consumer1(PipedInputStream pis) {
74. **this**.pis = pis;
75. }
77. **public** **void** run() {
78. **try** {
79. **while**(**true**)
80. {
81. System.out.println("consumer1:"+pis.read());
82. }
83. } **catch** (IOException e) {
84. e.printStackTrace();
85. }
86. }
87. }

程序启动后，就可以看到producer线程往consumer1线程发送数据

**[java]** [view plain](http://blog.csdn.net/ls5718/article/details/51878770) [copy](http://blog.csdn.net/ls5718/article/details/51878770)

1. consumer1:0
2. consumer1:1
3. consumer1:2
4. consumer1:3
5. ......

管道流虽然使用起来方便，但是也有一些缺点

1）管道流只能在两个线程之间传递数据

线程consumer1和consumer2同时从pis中read数据，当线程producer往管道流中写入一段数据后，每一个时刻只有一个线程能获取到数据，并不是两个线程都能获取到producer发送来的数据，因此一个管道流只能用于两个线程间的通讯。不仅仅是管道流，其他IO方式都是一对一传输。

2）管道流只能实现单向发送，如果要两个线程之间互通讯，则需要两个管道流

 可以看到上面的例子中，线程producer通过管道流向线程consumer发送数据，如果线程consumer想给线程producer发送数据，则需要新建另一个管道流pos1和pis1，将pos1赋给consumer1，将pis1赋给producer，具体例子本文不再多说。

II.进程与进程间通信

一、进程间通信方式

（1）管道（Pipe）：管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。  
（2）命名管道（named pipe）：命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关 系 进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令mkfifo或系统调用mkfifo来创建。  
（3）信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送 信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）。  
（4）消息（Message）队列：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺  
（5）共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。  
（6）内存映射（mapped memory）：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。  
（7）信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。

（8）套接口（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。