**为什么要线程同步**

因为当我们有多个线程要同时访问一个变量或对象时，如果这些线程中既有读又有写操作时，就会导致变量值或对象的状态出现混乱，从而导致程序异常。举个例子，如果一个银行账户同时被两个线程操作，一个取100块，一个存钱100块。假设账户原本有0块，如果取钱线程和存钱线程同时发生，会出现什么结果呢？取钱不成功，账户余额是100.取钱成功了，账户余额是0.那到底是哪个呢？很难说清楚。因此多线程同步就是要解决这个问题。

举例：

不同步代码：java

1. **public** **class** Bank {
3. **private** **int** count =0;//账户余额
5. //存钱
6. **public**  **void** addMoney(**int** money){
7. count +=money;
8. System.out.println(System.currentTimeMillis()+"存进："+money);
9. }
11. //取钱
12. **public**  **void** subMoney(**int** money){
13. **if**(count-money < 0){
14. System.out.println("余额不足");
15. **return**;
16. }
17. count -=money;
18. System.out.println(+System.currentTimeMillis()+"取出："+money);
19. }
21. //查询
22. **public** **void** lookMoney(){
23. System.out.println("账户余额："+count);
24. }
25. }

**使用同步时的代码**

**（1）同步方法：**

即有synchronized关键字修饰的方法。 由于java的每个对象都有一个内置锁，当用此关键字修饰方法时，内置锁会保护整个方法。在调用该方法前，需要获得内置锁，否则就处于阻塞状态。

修改后的Bank.java

1. **public** **class** Bank {
3. **private** **int** count =0;//账户余额
5. //存钱
6. **public**  **synchronized** **void** addMoney(**int** money){
7. count +=money;
8. System.out.println(System.currentTimeMillis()+"存进："+money);
9. }
11. //取钱
12. **public**  **synchronized** **void** subMoney(**int** money){
13. **if**(count-money < 0){
14. System.out.println("余额不足");
15. **return**;
16. }
17. count -=money;
18. System.out.println(+System.currentTimeMillis()+"取出："+money);
19. }
21. //查询
22. **public** **void** lookMoney(){
23. System.out.println("账户余额："+count);
24. }
25. }

**（2）同步代码块**

即有synchronized关键字修饰的语句块。被该关键字修饰的语句块会自动被加上内置锁，从而实现同步

Bank.java代码如下：

1. **public** **class** Bank {
3. **private** **int** count =0;//账户余额
5. //存钱
6. **public**   **void** addMoney(**int** money){
8. **synchronized** (**this**) {
9. count +=money;
10. }
11. System.out.println(System.currentTimeMillis()+"存进："+money);
12. }
14. //取钱
15. **public**   **void** subMoney(**int** money){
17. **synchronized** (**this**) {
18. **if**(count-money < 0){
19. System.out.println("余额不足");
20. **return**;
21. }
22. count -=money;
23. }
24. System.out.println(+System.currentTimeMillis()+"取出："+money);
25. }
27. //查询
28. **public** **void** lookMoney(){
29. System.out.println("账户余额："+count);
30. }
31. }

**（3）使用重入锁实现线程同步**

    在JavaSE5.0中新增了一个java.util.concurrent包来支持同步。ReentrantLock类是可重入、互斥、实现了Lock接口的 锁， 它与使用synchronized方法和快具有相同的基本行为和语义，并且扩展了其能力。

     ReenreantLock类的常用方法有：

         ReentrantLock() : 创建一个ReentrantLock实例

         lock() : 获得锁

         unlock() : 释放锁

     注：ReentrantLock()还有一个可以创建公平锁的构造方法，但由于能大幅度降低程序运行效率，不推荐使用

1. **public** **class** Bank {
3. **private**  **int** count = 0;// 账户余额
5. //需要声明这个锁
6. **private** Lock lock = **new** ReentrantLock();
8. // 存钱
9. **public** **void** addMoney(**int** money) {
10. lock.lock();//上锁
11. **try**{
12. count += money;
13. System.out.println(System.currentTimeMillis() + "存进：" + money);
15. }**finally**{
16. lock.unlock();//解锁
17. }
18. }
20. // 取钱
21. **public** **void** subMoney(**int** money) {
22. lock.lock();
23. **try**{
25. **if** (count - money < 0) {
26. System.out.println("余额不足");
27. **return**;
28. }
29. count -= money;
30. System.out.println(+System.currentTimeMillis() + "取出：" + money);
31. }**finally**{
32. lock.unlock();
33. }
34. }
36. // 查询
37. **public** **void** lookMoney() {
38. System.out.println("账户余额：" + count);
39. }
40. }

**(4)使用局部变量实现线程同步**

1. **public** **class** Bank {
3. **private** **static** ThreadLocal<Integer> count = **new** ThreadLocal<Integer>(){
5. @Override
6. **protected** Integer initialValue() {
7. // TODO Auto-generated method stub
8. **return** 0;
9. }
11. };

14. // 存钱
15. **public** **void** addMoney(**int** money) {
16. count.set(count.get()+money);
17. System.out.println(System.currentTimeMillis() + "存进：" + money);
19. }
21. // 取钱
22. **public** **void** subMoney(**int** money) {
23. **if** (count.get() - money < 0) {
24. System.out.println("余额不足");
25. **return**;
26. }
27. count.set(count.get()- money);
28. System.out.println(+System.currentTimeMillis() + "取出：" + money);
29. }
31. // 查询
32. **public** **void** lookMoney() {
33. System.out.println("账户余额：" + count.get());
34. }
35. }

ThreadLocal的原理：

如果使用ThreadLocal管理变量，则每一个使用该变量的线程都获得该变量的副本，副本之间相互独立，这样每一个线程都可以随意修改自己的变量副本，而不会对其他线程产生影响。现在明白了吧，原来每个线程运行的都是一个副本，也就是说存钱和取钱是两个账户，知识名字相同而已。所以就会发生上面的效果。

ThreadLocal与同步机制

a.ThreadLocal与同步机制都是为了解决多线程中相同变量的访问冲突问题

b.前者采用以"空间换时间"的方法，后者采用以"时间换空间"的方式