- synchronized 方法

这个方法可以是静态方法和非静态方法，但是不能是抽象类的抽象方法，也不能是接口中的接口方法。通过在方法声明中加入 synchronized关键字来声明，语法如下：

```
1    public  synchronized  void accessVal(int newVal);
```

synchronized 方法的缺陷：若将一个大的方法声明为synchronized 将会大大影响效率。

- synchronized块

synchronized代码块也能保证代码块中所有变量都将会从主存中读，当线程退出代码块时，对所有变量的更新将会flush到主存，不管这些变量是不是volatile类型的。

```
synchronized(syncObject)
    {
     //允许访问控制的代码
    }
```

锁定的对象就是同步方法所属的主体**对象自身**。如果这个方法是静态同步方法呢？那么线程锁定的就不是这个类的对象了，也不是这个类自身，而是这个类对应的**java.lang.Class类型的对象**。同步方法和同步块之间的相互制约只限于同一个对象之间，所以静态同步方法只受它所属类的其它静态同步方法的制约，而跟这个类的实例（对象）没有关系。synchronized的目的是使**同一个对象**的**多个线程**

```
    Runnable r1 = new ThreadTest(); //也可写成ThreadTest r1 = new ThreadTest();
    Runnable r2 = new ThreadTest();
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
--------------------------------------------------------------------------------
---
   Runnable r = new ThreadTest();
    Thread t1 = new Thread(r);
    Thread t2 = new Thread(r);
    t1.start();
    t2.start();
```

多线程操作同一个对象(使用线程同步)

```
public class TestSync {
    public static void main(String[] args) {
        Account a1 = new Account(100, "高");
        Drawing draw1 = new Drawing(80, a1);
        Drawing draw2 = new Drawing(80, a1);
        draw1.start(); // 你取钱
        draw2.start(); // 你老婆取钱
    }
}
/*
 * 简单表示银行账户
 */
class Account {
    int money;
    String aname;
    public Account(int money, String aname) {
        super();
        this.money = money;
        this.aname = aname;
    }
}
/**
 * 模拟提款操作
 *
 * @author Administrator
 *
 */
class Drawing extends Thread {
    int drawingNum; // 取多少钱
    Account account; // 要取钱的账户
    int expenseTotal; // 总共取的钱数

    public Drawing(int drawingNum, Account account) {
        super();
        this.drawingNum = drawingNum;
        this.account = account;
    }

    @Override
    public void run() {
        draw();
    }

    void draw() {
        synchronized (account) {                //一个接一个account
            if (account.money - drawingNum < 0) {
                System.out.println(this.getName() + "取款，余额不足！");
                return;
            }
            try {
                Thread.sleep(1000); // 判断完后阻塞。其他线程开始运行。
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            account.money -= drawingNum;
            expenseTotal += drawingNum;
        }
        System.out.println(this.getName() + "--账户余额：" + account.money
```

```
60              System.out.println(this.getName() + "--总共取了：" + expenseTotal)
61          }
        }
```

“synchronized (account)” 意味着线程需要获得account对象的“锁”才有资格运行同步块中的代码。 Account对象的“锁”也称为“互斥锁”，在同一时刻只能被一个线程使用。

```
public class SynchronizedThread {

    class Bank {
        private int account = 100;

        public int getAccount() {
            return account;
        }

        /**
         * 用同步方法实现
         *
         * @param money
         */
        public synchronized void save(int money) {
            account += money;
        }

        /**
         * 用同步代码块实现
         *
         * @param money
         */
        public void save1(int money) {
            synchronized (this) {
```

```
                account += money;
            }
        }
    }
```

- **使用特殊域变量(volatile)**

a.volatile关键字为域变量的访问提供了一种免锁机制，

b.使用volatile修饰域相当于告诉虚拟机该域可能会被其他线程更新，

c.因此每次使用该域就要**重新计算**，而不是使用寄存器中的值 。保证变量会直接从主存读取，而对变量的更新也会直接写到主存。

d.volatile不会提供任何原子操作，它也不能用来修饰final类型的变量

```
class Bank {
    //需要同步的变量加上volatile
    private volatile int account = 100;

    public int getAccount() {
        return account;
    }
    //这里不再需要synchronized
    public void save(int money) {
        account += money;
    }
}
```

注：多线程中的非同步问题主要出现在对域的读写上，如果让域自身避免这个问题，则就不需要修改操作该域的方法。

用final域，有锁保护的域和volatile域可以避免非同步的问题。

- **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，

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

```
class Bank {

    private int account = 100;
    //需要声明这个锁
    private Lock lock = new ReentrantLock();
    public int getAccount() {
        return account;
    }
    //这里不再需要synchronized
    public void save(int money) {
        lock.lock();
        try{
            account += money;
        }finally{
            lock.unlock();//注意及时释放锁，否则会出现死锁，通常在finally代码释放锁
        }

    }
}
```

- **使用局部变量实现线程同步**

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。

ThreadLocal 类的常用方法：
ThreadLocal()：创建一个线程本地变量
get()：返回此线程局部变量的当前线程副本中的值
initialValue()：返回此线程局部变量的当前线程的"初始值"

set(T value)：将此线程局部变量的当前线程副本中的值设置为value

```
      public class Bank{
            //使用ThreadLocal类管理共享变量account
            private static ThreadLocal<Integer>
account = new ThreadLocal<Integer>(){
                @Override
                protected Integer initialValue(){
                    return 100;
                }
            };
            public void save(int money){
                account.set(account.get()+money);
            }
            public int getAccount(){
                return account.get();
            }
        }
```

注：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