MyRunnable.class：获取的是MyRunnable在内存中的运行时类，每个运行时类只有一份。

单例模式

整个系统中，无论做怎样的业务操作，保证内存中只有一个对象实例。

数据保存在主内存中的，线程在访问该数据时，并不是直接访问主内存中的数据，而是从主内存中拷贝一份数据出来放入工作内存，线程对工作内存中的数据进行操作，操作完成之后再将工作内存中的数据拷贝到主内存中。

使用volatile关键字来解决因为工作内存-主内存机制导致可能会发生的单例的错误。

使用volatile关键字修饰的变量，主内存对线程可见。

线程甲修改了其工作内存中的数据，线程乙工作内存中的数据会同步更新的。

double-check

```
class Singleton{
    private volatile static Singleton instance = null;
    private Singleton() {
        System.out.println("创建了Singleton对象");
    }
    public static Singleton getInstance() {
        if(instance == null) {
            synchronized (Singleton.class) {
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

死锁

```
package com.southwind.thread;

public class Test3 {
    public static void main(String[] args) {
        DeadLockRunnable d1 = new DeadLockRunnable();
        d1.flag = 1;
        DeadLockRunnable d2 = new DeadLockRunnable();
        d2.flag = 2;
        new Thread(d1,"张三").start();
```

```
        //避免出现死锁
        try {
            Thread.currentThread().sleep(1000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        new Thread(d2,"李四").start();
    }
}

class DeadLockRunnable implements Runnable{
    public int flag;
    private static Object o1 = new Object();
    private static Object o2 = new Object();
    @Override
    public void run() {
        // TODO Auto-generated method stub
        if(flag == 1) {
            System.out.println(Thread.currentThread().getName()+"获取了资源
o1，等待获取资源o2");
            synchronized (o1) {
                try {
                    Thread.currentThread().sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                synchronized (o2) {
                    System.out.println(Thread.currentThread().getName()+"执
行完毕");
                }
            }
        }
        if(flag == 2) {
            System.out.println(Thread.currentThread().getName()+"获取了资源
o2，等待获取资源o1");
            synchronized (o2) {
                try {
                    Thread.currentThread().sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                synchronized (o1) {
                    System.out.println(Thread.currentThread().getName()+"执
行完毕");
                }
            }
```

```
        }
    }

}
```

线程实际应用

1.多线程模拟两个跑步，每个线程代表一个人，可设置每个人跑步的速度，每跑完100米给出响应的信息，跑到终点之后给出响应的提示。

```
package com.southwind.thread;

public class Test4 {
    public static void main(String[] args) {
        RunRunnable r1 = new RunRunnable(2000, 1);
        RunRunnable r2 = new RunRunnable(500, 1);
        new Thread(r1,"张三").start();
        new Thread(r2,"李四").start();
    }
}

class RunRunnable implements Runnable{
    //跑100米需要的时间
    private int time;
    //已跑完的100米
    private int num;

    public RunRunnable(int time,int km) {
        this.time = time;
        this.num = km*1000/100;
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
        while(num > 0) {
            System.out.println(Thread.currentThread().getName()+"跑了100
米！");
            try {
                Thread.currentThread().sleep(this.time);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            num--;
        }
        System.out.println(Thread.currentThread().getName()+"到达了终点！");
    }
}
```