

- 虚拟的CPU，封装在java.lang.Thread类的一个实例中。
- 代码可以或不可以由多个线程共享，这和数据是独立的。两个线程如果执行同一个类的实例代码，则它们可以共享相同的代码。
- 数据可以或不可以由多个线程共享，这和代码是独立的。两个线程如果共享对一个公共对象的存取，则它们可以共享相同的数据

独立  
共享  
Trade off

## 线程体

- Java的线程是通过java.lang.Thread类来实现的。
- 每个线程都是通过某个特定Thread对象的方法run()来完成其操作的，方法run()称为线程体。

```
1 public class ThreadTester {  
2     public static void main(String args[]) {  
3         HelloRunner r = new HelloRunner();  
4         Thread t = new Thread(r);  
5         t.start();  
6     }  
7 }  
8  
9 class HelloRunner implements Runnable {  
10    int i;  
11  
12    public void run() {  
13        i = 0;  
14  
15        while (true) {  
16            System.out.println("Hello " + i++);  
17            if ( i == 50 ) {  
18                break;  
19            }  
20        }  
21    }  
22}
```

线程机

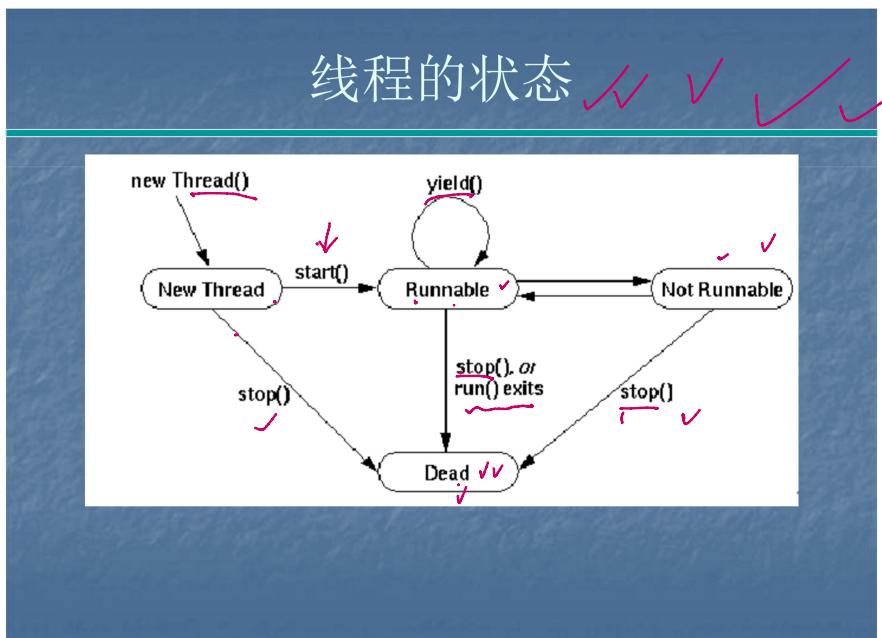
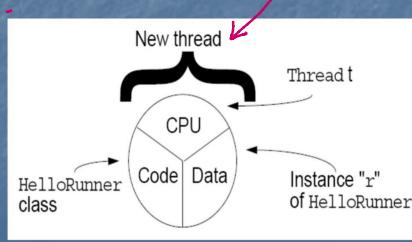
## 线程体

## 线程体

一个多线程编程环境允许创建基于同一个Runnable实例的多个线程。

```
Thread t1= new Thread(r);  
Thread t2= new Thread(r);  
此时，这两个线程共享数据和代码。
```

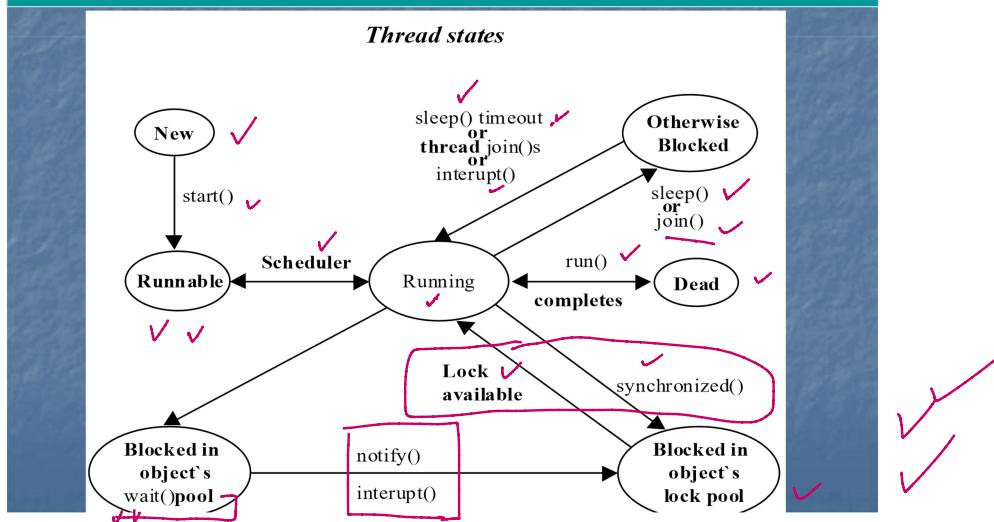
线程通过Thread对象的一个实例引用。线程从装入的Runnable实例的run()方法开始执行。线程操作的数据从传递给Thread构造函数的Runnable的特定实例处获得。

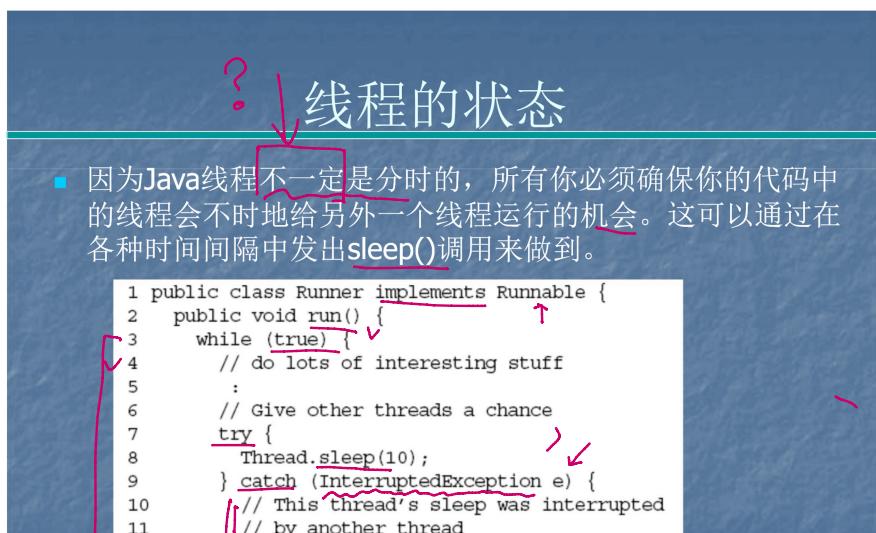
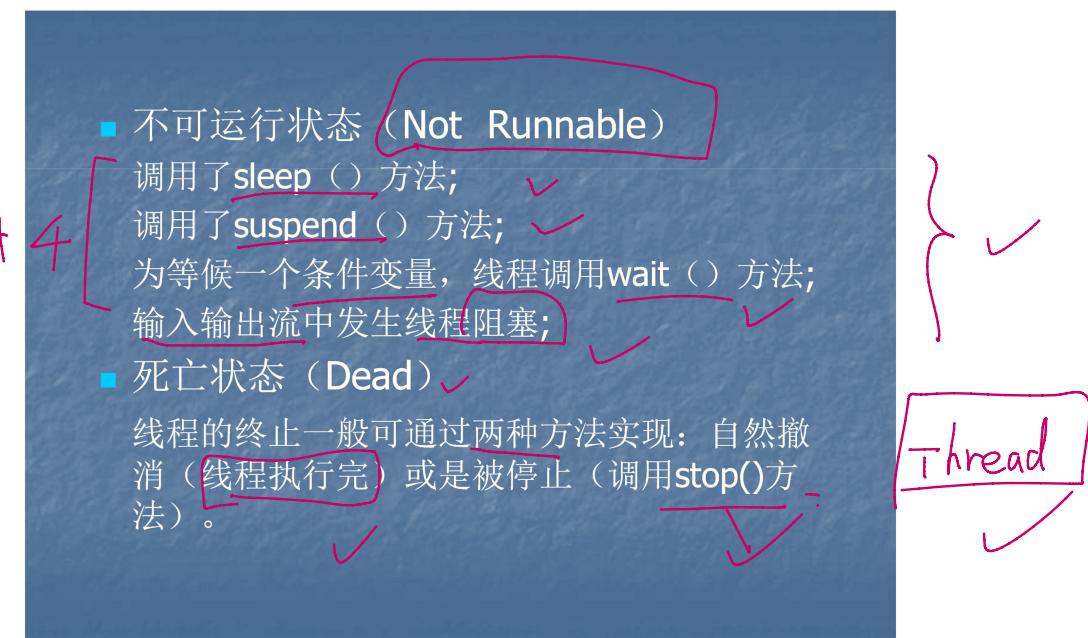
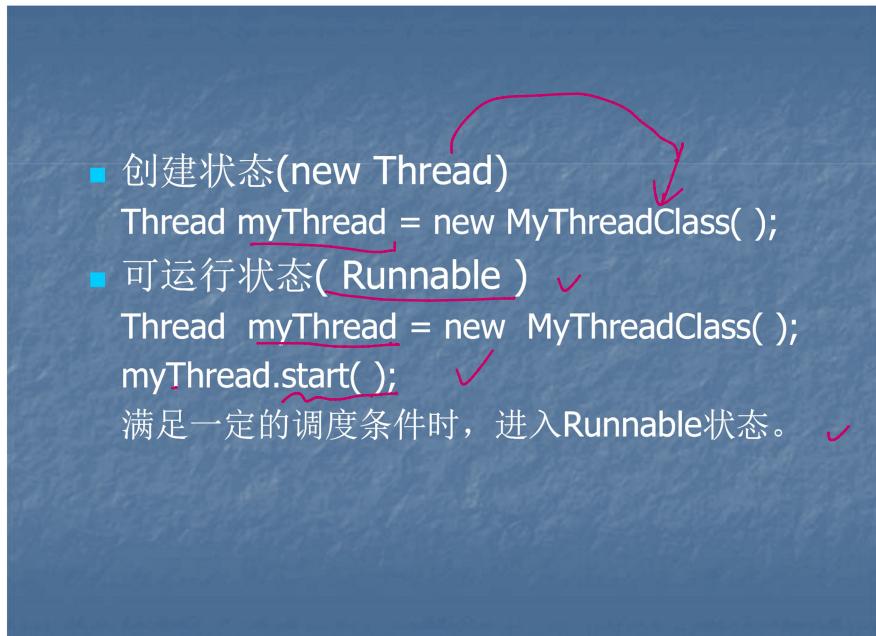


## 线程的状态

- 在Java中，线程是抢占式的，但并不一定是分时的
- 抢占式调度模型是指可能有多个线程是可运行的，但只有一个线程在实际运行。这个线程会一直运行，直至它不再是可运行的，或者另一个具有更高优先级的线程成为可运行的。对于后面一种情形，低优先级线程被高优先级线程抢占了运行的机会。
- 一个线程可能因为各种原因而不再是可运行的。线程的代码可能执行了一个Thread.sleep()调用，要求这个线程暂停一段固定的时间。这个线程可能在等待访问某个资源，而且在这个资源可访问之前，这个线程无法继续运行。
- 所有可运行线程根据优先级保存在池中。当一个被阻塞的线程变成可运行时，它会被放回相应的可运行池，优先级最高的非空池中的线程会得到处理机时间(被运行)。

## 线程的状态





```
8     Thread.sleep(10);
9 } catch (InterruptedException e) {
10    // This thread's sleep was interrupted
11    // by another thread
12 }
13 }
14 }
15 }
```

## 线程的状态

- **Thread**类的另一个方法**yield()**, 可以用来使具有相同优先级的线程获得执行的机会。如果具有相同优先级的其它线程是可运行的, **yield()**将把调用线程放到可运行池中并使另一个线程运行。如果没有相同优先级的可运行进程, **yield()**什么都不做。
- **sleep()**调用会给较低优先级线程一个运行的机会。  
**yield()**方法只会给相同优先级线程一个执行的机会。

## 线程体的构造

- **public Thread( ThreadGroup group, Runnable target, String name );**
- 任何实现接口**Runnable**的对象都可以作为一个线程的目标对象;

- 构造线程体的2种方法
  - 定义一个线程类, 它继承类**Thread**并重写其中的方法**run()**;

- 定义一个线程类，它继承类 Thread 并重写其中的方法 run();
- 提供一个实现接口 Runnable 的类作为线程的目标对象，在初始化一个 Thread 类或者 Thread 子类的线程对象时，把目标对象传递给这个线程实例，由该目标对象提供线程体 run();

通过继承类 Thread 构造线程体

```
class SimpleThread extends Thread {  
    public SimpleThread(String str) {  
        super(str);  
    }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i + " " + getName());  
            try {  
                sleep((int)(Math.random() * 1000));  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```

```
    System.out.println("DONE! " + getName());  
}  
}  
public class TwoThreadsTest {  
    public static void main (String args[]) {  
        new SimpleThread("First").start();  
        new SimpleThread("Second").start();  
    }  
}
```



0	First
0	Second
1	Second
1	First
2	First
2	Second
3	Second
3	First
4	First
4	Second
5	First
5	Second
<b>6</b>	<b>Second</b>

13

6	First
7	First
7	Second
8	Second
9	Second
8	First
DONE! Second	
9	First
<b>DONE! First</b>	

通过接口构造线程体

```
public class Clock extends java.applet.Applet  
implements Runnable {  
    Thread clockThread;  
    public void start() {  
        if (clockThread == null) {  
            clockThread = new Thread(this,  
"Clock");  
            clockThread.start();  
        }  
    }
```

雷电

AWT

Swing

```
public void run() {  
    while (clockThread != null) {  
        repaint();  
        try {  
            clockThread.sleep(1000);  
        } catch (InterruptedException e){}  
    }  
}  
  
public void paint(Graphics g) {  
    Date now = new Date();  
    g.drawString(now.getHours() + ":" +  
    now.getMinutes() + ":"  
    +now.getSeconds(), 5, 10);  
}
```

```
public void stop() {  
    clockThread.stop();  
    clockThread = null;  
}  
}
```

## 两种方法的比较

- 使用 **Runnable** 接口 ✓  
可以将 CPU, 代码和数据分开, 形成清晰的模型;  
还可以从其他类继承; ✓  
保持程序风格的一致性。
- 直接继承 **Thread** 类 ✓✓ 子  
不能再从其他类继承;  
编写简单, 可以直接操纵线程, 无需使用  
`Thread.currentThread()`。

## 线程的基本控制 ✓✓✓

- 当一个线程结束运行并终止时, 它就不能再运行了。✓
- 可以用一个指示 `run()` 方法必须退出的标志来停止一个线程
- 在一段特定的代码中, 可以使用静态 **Thread** 方法  
`currentThread()` 来获取对当前线程的引用

```
1 public class NameRunner implements Runnable {  
2     public void run() {  
3         while (true) {  
4             // lots of interesting stuff ✓✓✓  
5         }  
6         // Print name of the current thread  
7         System.out.println(  
8             "Thread " + Thread.currentThread().getName() + " completed");  
9     }  
10 }
```

```

1 } // end of implementing start vvvv
2     // Print name of the current thread
3     System.out.println(
4         "Thread " + Thread.currentThread().getName() + " completed");
5 }
6
7
8
9
10

```

```

1 public class Runner implements Runnable {
2     private boolean timeToQuit=false;
3
4     public void run() {
5         while ( ! timeToQuit ) {
6             ...
7         }
8         // clean up before run() ends
9     }
10
11    public void stopRunning() {
12        timeToQuit=true;
13    }
14 }

```

```

1 public class ThreadController {
2     private Runner r = new Runner();
3     private Thread t = new Thread(r);
4
5     public void startThread() {
6         t.start();
7     }
8
9     public void stopThread() {
10        // use specific instance of Runner
11        r.stopRunning();
12    }
13 }

```

## 线程的调度

- Java 提供一个线程调度器来监控程序中启动后进入就绪状态的所有线程。线程调度器按照线程的优先级决定应调度哪些线程来执行。
  - setPriority(int) 方法设置优先级
- 多数线程的调度是抢先式的。
  - 时间片方式
  - 非时间片方式

- 下面几种情况下，当前线程会放弃CPU：
  - 线程调用了**yield()**, **suspend()**或**sleep()**方法主动放弃；
  - 由于当前线程进行I/O访问，外存读写，等待用户输入等操作，导致线程阻塞；
  - 为等候一个条件变量，线程调用**wait()**方法；
  - 抢先式系统下，有高优先级的线程参与调度；  
时间片方式下，当前时间片用完，有同优先级的线程参与调度。

## 线程的优先级

- 线程的优先级用数字来表示，范围从1到10，即**Thread.MIN\_PRIORITY**到**Thread.MAX\_PRIORITY**。一个线程的缺省优先级是5，即**Thread.NORM\_PRIORITY**。
  - **int getPriority();**
  - **void setPriority(int newPriority);**

```
class ThreadTest{  
    public static void main( String args [] ) {  
        ✓ Thread t1 = new MyThread("T1");  
        t1.setPriority( Thread.MIN_PRIORITY ); ✓  
        ✓ t1.start( );  
        ✓ Thread t2 = new MyThread("T2");  
        t2.setPriority( Thread.MAX_PRIORITY ); ✓  
        ✓ t2.start( );  
        ✓ Thread t3 = new MyThread("T3");  
        t3.setPriority( Thread.MAX_PRIORITY );  
        ✓ t3.start( );  
    }  
}
```

```
class MyThread extends Thread {  
    String message;  
    MyThread ( String message ) {  
        this.message = message;  
    }  
    public void run(){  
        for ( int i=0; i<3; i++ )  
            System.out.println( message+ " " +getPriority() );  
    }  
}
```

运行结果:

```

T2 10
T2 10
T2 10
T3 10
T3 10
T3 10
T1 1
T1 1
T1 1

```

注意: 并不是在所有系统中运行Java程序时都采用时间片策略调度线程, 所以一个线程在空闲时应该主动放弃CPU, 以使其他同优先级和低优先级的线程得到执行。



```

1 public class ThreadDemo extends Thread {
2     //2种不同的方法生成线程t1和t2
3     public void run(){
4         for(int i = 1; i<5;i++) compute(); vvvv
5     }
6     public static void main(String args[]){
7         ThreadDemo t1 = new ThreadDemo();
8         Thread t2 = new Thread(new Runnable() { ✓①
9             public void run() { ✓②
10                for(int i=1;i<5;i++) compute(); vvvv
11            }
12        });
13        if (args.length >=1) t1.setPriority(Integer.parseInt(args[0]));
14        if (args.length >=2) t2.setPriority(Integer.parseInt(args[1]));
15        t1.start();t2.start();
16        for(int i=0; i<5; i++) compute(); vvvv
17    }
18    static ThreadLocal<Integer> numberOfCalls = new ThreadLocal<Integer> ();
19    //ThreadLocal记录了每个线程调用compute()的次数。
20    static synchronized void compute(){ ✓
21        //计算当前线程被调用的次数
22        Integer n = (Integer) numberOfCalls.get();
23        if (n == null) n = new Integer(1);
24        else n = new Integer(n.intValue()+1);
25        numberOfCalls.set(n);
26        System.out.println(Thread.currentThread().getName() + ":" + n);
27        for(int i=0,j=0;i<1000000;i++) j+= i;

```

```

25     if (n == null) n = new Integer(1);
26     else n = new Integer(n.intValue() + 1);
27     numberOfCalls.set(n);
28     System.out.println(Thread.currentThread().getName() + ": " + n);
29     for (int i=0,j=0;i<1000000;i++) j+= i;
30     try {
31         Thread.sleep((int)(Math.random()*100+1));
32     } catch (InterruptedException e){}
33     Thread.yield(); //线程间的谦让
}

```

‘让渡’

## 基本的线程控制

### ■ 终止线程 ✓

■ 线程执行完其run()方法后，会自然终止。

■ 通过调用线程的实例方法stop()来终止线程。✓

### ■ 测试线程状态

■ 可以通过Thread中的isAlive()方法来获取线程是否处于活动状态；

■ 线程由start()方法启动后，直到其被终止之间的任何时刻，都处于‘Alive’状态。

### ■ 线程的暂停和恢复 ✓

■ sleep()方法 ✓

■ suspend()和resume() ✓

可以由线程自身调用suspend()方法暂停自己，也可以由其它线程调用suspend()方法暂停其执行，但是要恢复由suspend()方法挂起的线程，只能由其它线程来调用resume()方法。

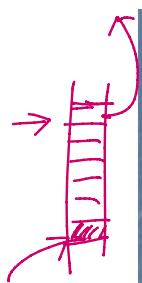
■ join() ✓

使当前线程停下来等待，直至另一个调用join方法的线程终止。

```

1 public static void main(String[] args) {
2     Thread t = new Thread(new Runner());
3     t.start();
4     ...
5     // Do stuff in parallel with the other thread for a while
6     ...
7     // Wait here for the timer thread to finish
8     try {
9         t.join(); ✓
}

```



```
7 // Wait here for the timer thread to finish
8 try {
9     t.join(); ✓
10 } catch (InterruptedException e) {
11     // t came back early
12 }
13 ...
14 // Now continue in this thread
15 ...
16 }
```

void join(long timeout);

## 多线程的互斥与同步

- 关键字 **synchronized**
- 提供 Java 编程语言一种机制，允许程序员控制共享数据的线程。
- 下面的堆栈的例子是一个当多线程共享数据时会经常发生的问题的一个简单范例。说明需要有机制来保证共享数据在任何线程使用它完成某一特定任务之前是一致的。

## 多线程的互斥与同步

- 临界资源问题

```
class stack{
    int idx=0; ✓
    char[ ] data = new char[6];
    public void push(char c){
        data[idx] = c; ←
        idx++; ←
    }
}
```

```
public char pop(){  
    idx--;  
    return data[idx];  
}
```

两个线程A和B在同时使用Stack的同一个实例对象，  
A正在往堆栈里push一个数据，B则要从堆栈中pop  
一个数据。

- 1) 操作之前 data = | p | q | | | | | idx=2
- 2) A执行push中的第一个语句，将r推入堆栈；

data = | p | q | r | | | | | idx=2

3) A还未执行idx++语句，A的执行被B中断，B执行  
pop方法，返回q:

data = | p | q | r | | | | | idx=1 ← A  
 B

4) A继续执行push的第二个语句：

data = | p | q | r | | , | | | idx=2 ←

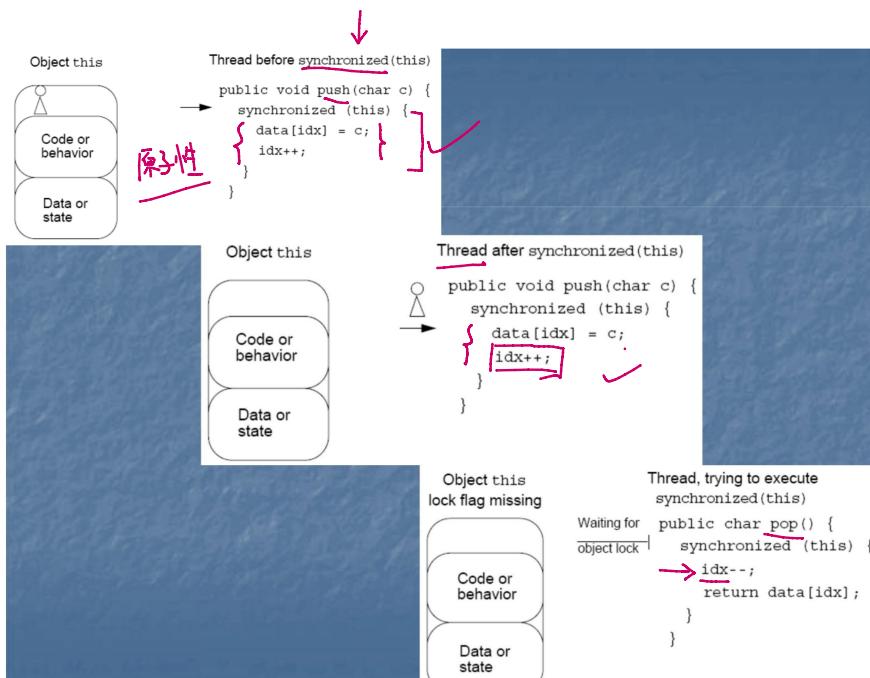
最后的结果相当于r没有入栈。

■ 产生这种问题的原因在于对共享数据访问的操作的不完整性。

在Java语言中，引入了对象互斥锁的概念，来保证共享数据操作的完整性。

每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。  $10^{-9}$  second

关键字synchronized 来与对象的互斥锁联系。当某个对象用synchronized修饰时，表明该对象在任一时刻只能由一个线程访问。



```
public void push(char c){  
    synchronized(this){  
        data[idx]=c;  
        idx++;  
    }  
}  
public char pop(){  
    synchronized(this){  
        idx--;  
        return data[idx];  
    }  
}
```

```
    }  
    return data[idx];  
}
```

## 多线程的互斥与同步

- 线程执行到 synchronized() 代码块末尾时释放
- synchronized() 代码块抛出中断或异常时自动释放
  - 由于等待一个对象的锁标志的线程在得到标志之前不能恢复运行，所以让持有锁标志的线程在不再需要的时候返回标志是很重要的。
  - 锁标志将自动返回给它的对象。持有锁标志的线程执行到 synchronized() 代码块末尾时将释放锁。Java 技术特别注意了保证即使出现中断或异常而使得执行流跳出 synchronized() 代码块，锁也会自动返回。此外，如果一个线程对同一个对象两次发出 synchronized 调用，则在跳出最外层的块时，标志会正确地释放，而最内层的将被忽略。

try  
catch

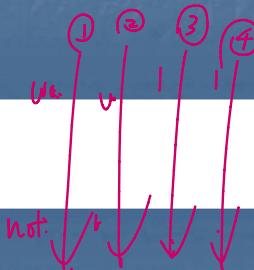
- synchronized 除了象上面讲的放在对象前面限制一段代码的执行外，还可以放在方法声明中，表示整个方法为同步方法。

```
public synchronized void push(char c){  
    ...  
}
```
- 如果 synchronized 用在类声明中，则表明该类中的所有方法都是 synchronized 的。
- 由 synchronized 保护的数据应当是 private 的。

## 多线程的同步

```
class SyncStack{  
    private int index = 0;  
    private char []buffer = new char[6];
```

```
private int index = 0;  
private char []buffer = new char[6];  
  
public synchronized void push(char c){  
    while(index == buffer.length){  
        try{  
            this.wait();  
        }catch(InterruptedException e){}  
    }  
}
```



```
this.notify();  
buffer[index] = c;  
index++;  
}  
  
public synchronized char pop(){  
    while(index == 0){  
        try{  
            this.wait();  
        }catch(InterruptedException e){}  
    }  
}
```

```
this.notify();  
index--;  
return buffer[index];  
}  
}
```

1 1 1

# 死锁

- 如果程序中有多个线程竞争多个资源，就可能会产生死锁。当一个线程等待由另一个线程持有的锁，而后者正在等待已被第一个线程持有的锁时，就会发生死锁。在这种情况下，除非另一个已经执行到 `synchronized` 块的末尾，否则没有一个线程能继续执行。由于没有一个线程能继续执行，所以没有一个线程能执行到块的末尾。
- Java 技术不监测也不试图避免这种情况。因而保证不发生死锁就成了程序员的责任。避免死锁的一个通用的经验法则是：决定获取锁的次序并始终遵照这个次序。按照与获取相反的次序释放锁。



```
1 //该程序导致2个同时试图获取2个相同资源的线程之间的死锁
2 //所有线程都应该按照相同顺序请求锁，才能避免这种死锁。
3 public class DeadlockDemo {
4     public static void main(String [] args){
5         final Object res1 = "resource1";
6         final Object res2 = "resource2";
7         //线程1先拿锁res1，再拿锁res2
8         Thread t1 = new Thread(){
9             public void run(){
10                 synchronized(res1){
11                     System.out.println("Thread1: locked res1");
12                     try {Thread.sleep(50);}
13                     catch (InterruptedException e){}
14
15                     synchronized(res2){
16                         System.out.println("Thread1: locked res2");
17                     }
18
19                 }
20             };
21
22             //线程2的写法和线程1正好相反，它试图加锁res2，再加锁res1
23             Thread t2 = new Thread(){
24
25                 t1.start(); t2.start();
26
27             }
28
29     }
30 }
```

dead lock