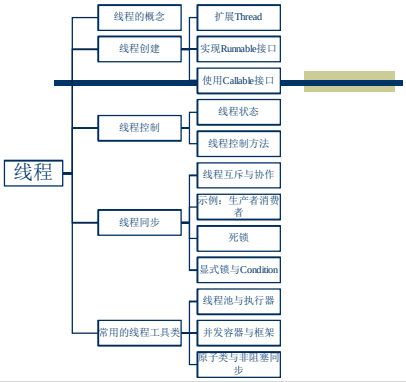
# 第9章 多线程编程

## 第9章 线程与并发编程




## 9.1 线程的概念

- **1、程序**

程序是一段静态代码，是指令与数据的集合。通常是外存上保存的可执行的二进制文件。

- **2、进程----Process**

进程（Process）是程序的一次运行活动。它对应从代码加载、执行到结束的一个过程，进程可以申请和拥有系统一整套资源，是系统进行资源分配和调度的基本单位。

- **3、线程---Thread**

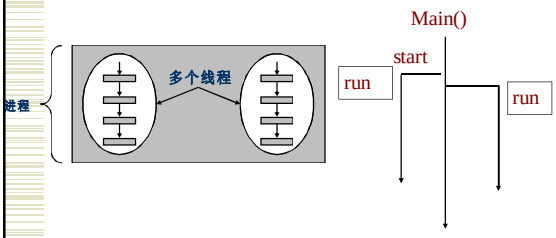
线程是进程中能够独立执行的执行序列。一个进程可以产生多个线程，线程也有创建、存活到消亡的的生命周期，每个线程都有独立的运行栈和程序计数器，是CPU调度的最小单位。

- **进程与线程关系**

一个进程中的所有线程共享相同的地址空间和这个进程所拥有的操作系统资源



## 线程的概念-----Java多线程程序设计






## 9.2 线程创建

Java提供了三种实现线程的方法：

- Ø 一、是扩展Thread类
- Ø 二、是实现Runnable接口
- Ø 三、是实现Callable接口。



## 9.2.1 扩展Thread类

**1、Thread类和Runnable接口**

java.lang.Thread类被用来封装线程执行机制。线程要执行的代码用Runnable接口定义，该接口是函数式接口，只包含一个run方法。

```
publlic void run()
```

Thread本身也实现了该接口。因此，创建线程的一个简单方法是扩展Thread类，并重写run() 方法。

**2、扩展Thread类定义线程**

```
class 子线程名 extends Thread {
    public void run() {
        /* 覆盖该方法*/
    }
}
```

当创建派生类的新对象后，可使用Thread的start()方法启动线程的run()方法。

```
public class ExtThread extends Thread{
	private int order ;
	public ExtThread(int order ){
		this.order=order;
		}
	public void run(){
		for(int i=1; i <=20; i+=2 ){
			System.out.print(order+",");
			}
		}
	}

class TestExtThread{
	public static void main(String arg[]) {
		ExtThread et1=new ExtThread(1); //创建线程
		et1.start();  //启动线程
		ExtThread et2=new ExtThread(2);//创建线程
		et2.start(); //启动线程
	}
}
```

【运行一次的结果】
1,2,2,2,2,2,2,1,2,1,2,1,1,1,1,1,2,1,1,2,

## 9.2.2 实现Runnable接口

- 向Thread的构造方法传递Runnable对象：该对象就是线程执行代码和处理数据的封装。Thread中有以下构造方法可以接收Runnable实例。

```
Public Thread(Runnable target)
Public Thread(Runnable target, String name)  //name为线程名
```

- 实现Runnable接口的语法如下：

```
public class  类名 [extends 父类]  implements Runnable{
public void run(){
    //线程执行的代码
}
}
```

## 9.2.2 实现Runnable接口

```
public class RunnableThread implements Runnable{
	private int order;
	public RunnableThread(int order) {
		this.order=order;
	}
	public void run() {
		for(int i=1; i <=20; i++ ){
			System.out.print(order+",");
			}
	}}
class TestThread2{
	public static void main(String arg[]) {
		RunnableThread r1=new RunnableThread(3);
		Thread thread1=new Thread(r1);
		thread1.start();
	 try {
		Thread.sleep(1);
	 }catch(Exception e) { }
	 System.out.print(" Done in main,");
	}
}
```

【运行一次的结果】
3, Done in main,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,

## 9.2.2 实现Runnable接口

RunnableThread线程类如果只生成了一个对象，可以定义为匿名类或写成Lambda表达式。其代码如下所示：

```
//实现Runnable的匿名类对象
	new Thread(new Runnable() {
		public void run() {
			for(int i=1;i<=10;i++)
				System.out.print(5+",");
		} }).start();
//实现Runnable匿名类对象的Lambda表达式
	new Thread(()-> {
		for(int i=1;i<=10;i++)
			System.out.print(6+","); }).start();
```

## 9.2.3 使用Callable接口和FutureTask

- 前两种实现线程的方法有一个共同的特点：执行任务后不会返回执行结果，因为run()方法是没有返回值的。如果用户想获取线程的执行结果，需要利用共享变量或使用线程间通信来实现。这样操作起来比较繁琐。

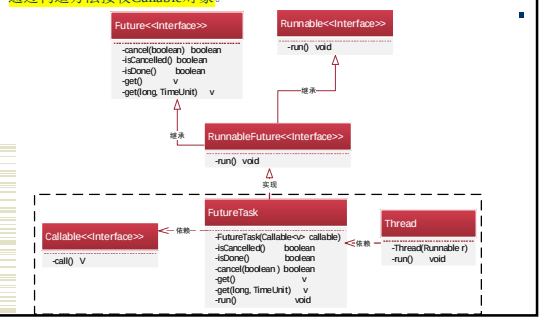
- 从Java5开始，Java就提供Callable接口和Future接口，Callable接口如下所示：

```
public interface Callable<V>{
        V call()throws Exception;
    }
```

方法call类似Runnable接口的run方法，表示要执行的代码

- Future接口表示异步计算的结果，它可以用get()方法来获得结果。

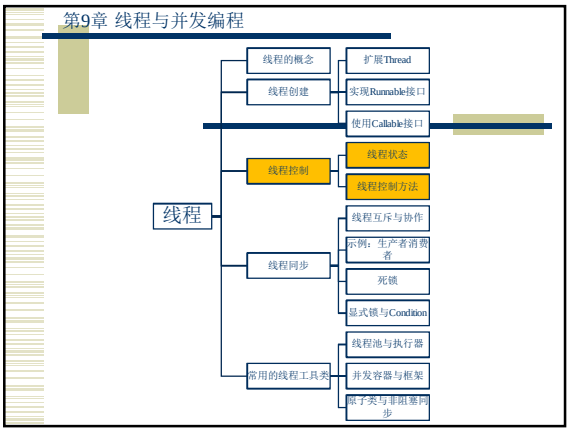
- 用Callable对象可以封装执行代码，不过Thread类并不接收Callable对象。解决方法是：用实现了Runnable接口和Future接口的类，比如FutureTask，通过构造方法接收Callable对象。



```
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;
public class CallableTest {
       public static void main(String arg[]) {
       1. MyCallable mc=new MyCallable();
       2. FutureTask<Integer> ft=new FutureTask<>(mc);
       3. new Thread(ft).start();
         try {
       4、      System.out.printf("\nthe result is:%d\n",ft.get());
          }catch(Exception e) {
               System.out.println(e.toString());
               }
    }}
class MyCallable implements Callable<Integer>{
       public Integer call() {
              int sum=0;
              for(int i=1;i<10;i++)
                     sum+=i;
              return sum;
       }}
```

【运行结果】 the result is:45

(1) 创建Callable接口的实现类，并实现Call方法。
(2) 使用FutureTask包装Callable的对象。
(3) 使用FutureTask对象作为Thread对象的target创建并启动线程。
(4) 调用FutureTask对象的get()来获取子线程执行结束的返回值。




## 9.3.1 线程状态

在Java中，线程从创建到结束通常要经历6种状态，线程状态可通过调用getState获得

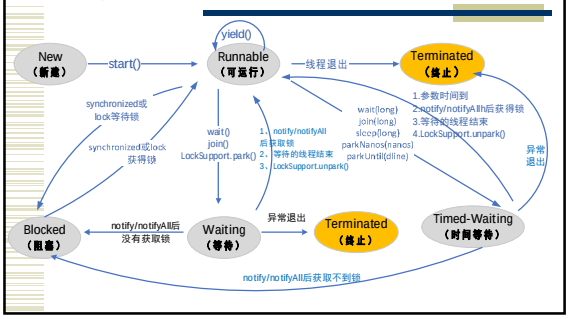



表 9-1 控制线程的常用方法

| 类型 | 方法 | 描述 |
|---|---|---|
| 启动线程 | start() | 启动线程，使线程从新建态进入可运行态 |
| 运行线程 | run() | 线程的执行代码，由系统自动调用。 |
| 设守护线程 | void setDaemon(boolean on) | 将此线程标记为守护线程或用户线程 |
| 修改优先级 | void setPriority(int newPri) | 将线程优先级改为 newPri |
| 线程让步 | static void yield() | 线程主动让出 CPU 使用权，转到就绪态（下一步还会参与 CPU 竞争） |
| 查询类操作 | | |
| | static Thread currentThread() | 返回当前执行线程的对象引用 |
| | String getName() | 返回线程名称 |
| | int getPriority() | 返回线程优先级 |
| | boolean isAlive() | 测试线程是否处于活动状态 |
| 暂停线程执行 | | |
| | static void sleep(long mils) | 让当前线程休眠 mils 毫秒 |
| | void join() | 如 th1.join()，表示挂起当前线程，等待 th1 线程运行结束再运行当前线程。 |
| | void join(long mil) | 表示当前线程最多等待 mil 毫秒后再运行 |
| | wait()(继承自 Object) | obj.wait()挂起当前线程，并释放目标对象 obj 的锁 |
| 唤醒等待线程 | | |
| 中断线程 | void interrupt() | 对于调用了 sleep,wait, join 等方法的休眠线程，该方法会结束休眠状态，并抛出异常；但运行态的线程不能被中断。对于非阻塞的线程，只是改变了线程中断状态，即 Thread.isInterrupted()将返回 true; |
| 唤醒线程 | notify()/notifyAll() | 唤醒通过 wait()操作处于等待态的线程 |

状态控制方法，可以自学

要求：

每种方法，后续请结合课本的示例(本课件18-30)和解释，

自行阅读学习

【运行结果】
Thread-0
5
Thread-1
5

**1、查询类操作**

【例9.4】调用线程的currentThread()、getName()、getPriority()。

```
public class QueryTest {
    public static void main(String arg[]) {
            new Thread() {
                    public void run() {
                    System.out.println(this.getName());
                    System.out.println(this.getPriority());
                    }
            }.start();

            new Thread(new Runnable() {
                    public void run() {
                            Thread th=Thread.currentThread();
                            System.out.println(th.getName());
                            System.out.println(th.getPriority());
                    }
            }).start();
    }}
```

## 2、sleep方法

sleep（long millisecond）方法可以让线程挂起指定长时间（单位为毫秒）。当时间到达后，线程将回到可运行状态。其基本使用方法是：

```
try {
        Thread.sleep(100);
    }catch(InterruptedException e) {
        处理代码
}
```

## 3、join方法

线程之间在执行顺序上可能有要求，比如，线程a要在线程b之前执行，那就可以在线程b中将线程a加入线程b（a.join()），join方法实际上是将两个线程合并成一个串行线程。



【例9.5】利用sleep和join方法暂停线程运行的示例代码

```
public class Sleep_Join {
   public static void main(String arg[]) {
        MyThread th1=new MyThread();
        th1.start();
        try {
                th1.join();
        }catch(InterruptedException e) {
                System.out.println("Join method is interrupted");
         }
                System.out.println("Done in main");
 }}

class MyThread extends Thread{
   public void run() {
        for (int i=0; i <=5; i++){
           try {
                Thread.sleep(500);
                System.out.print(i+" ");
            } catch(InterruptedException e) {
                e.printStackTrace();
                }
         }
        }}}
```

【运行结果】

```
0 1 2 3 4 5
Done in main
```

## 4、interrupt方法

目前，Java线程的中断设置是通过调用Thread.interrupt()方法来实现。

- 对于非阻塞线程，interrupt方法只是改变了中断状态，即Thread.isInterrupted()将返回true;
- 对于调用了sleep、wait、join方法的阻塞线程，这些线程不处于执行态是不可能给自己的中断状态置位的。但线程收到中断信号后，会抛出异常InterruptedException。

【例9.6】线程中断interrupt()方法的示例代码

```
public class InterruptDemo extends Thread{
   public static void main(String arg[]) {
        Thread th1=new Thread() {
            public void run() {
                System.out.println("\t\t in th1 thread");
                try {
                        Thread.sleep(2);
                }catch(InterruptedException e) {
                        System.out.println(" Exception handle in th1");
                }}};
        Thread th2=new Thread() {
            public void run() {
                while(true) {
                   if(this.isInterrupted()) {
                        System.out.println("close the task in th2");
                        break;
                    }else
                        System.out.println("Th2 is going on");
                                }}};

        th1.start();
        th2.start();

        try {sleep(1);}catch(InterruptedException e){}

        th1.interrupt();
        th2.interrupt();
        }}
```

【运行结果】

```
                in th1 thread
Th2 is going on
Th2 is going on
……
Th2 is going on
close the task th2.
Exception handle in th1
```

## 5、Thread.yield()方法

```
class EvenOdd extends Thread
{ private int order;
  public EvenOdd(int order) {
       this.order=order;
    }
  public void run()
   { for (int i=0; i <= 10; i += 2 ) {
      if(order==1&&i==4) Thread.yield();
      System.out.println("in the "+order+"thread :"+i);
       } }}
```

```
public class ThreadTest{
  public static void main(String[] args){
       EvenOdd ot = new EvenOdd(1);
       EvenOdd et = new EvenOdd(2);
       ot.start();
       et.start();
      System.out.println("Main thread done"); } }
```

## 6、设置Daemon线程

- 如果我们对某个线程对象在启动（调用start方法）之前调用setDaemon(true)方法，这个线程就变成了后台线程。
- 对java程序来说，后台线程被用于完成支持型线程。只要还有一个前台线程在运行，这个进程就不会结束，如果一个进程中只有后台线程运行，这个进程就会结束。

25

```
public class DaemonTest {
  public static void main(String arg[]) {
      MyDemonTh mt=new MyDemonTh();
      mt.setDaemon(true);
      mt.start();
      try {
         Thread.sleep(500);
      }catch(InterruptedException e) {}

      System.out.println("Done in main");
      }
}
class MyDemonTh extends Thread{
    public void run() {
      for(int i=1;i<10;i++){
         try {
            Thread.sleep(100);
         }catch(InterruptedException e) { }

      System.out.println(this.getName()+" is alive");
       }
      }}
```

```
Thread-0 is alive
Thread-0 is alive
Thread-0 is alive
Done in main
```

## 7、调度——Scheduling

- java解释器根据线程的优先级设置执行所有就绪的线程
- 低优先级的线程要等到所有的高优先级的线程被阻塞后才可以运行
  - Priority from 1 to 10
    - MIN_PRIORITY :1
    - MAX_PRIORITY:10
    - NORM_PRIORITY:5（默认）

27

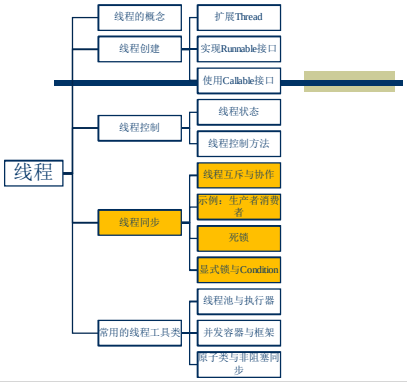
```
class PriThread extends Thread{
      private int id;
      public PriThread(int id) {
            this.id=id;
      }
      public void run() {
            for(int i=1;i<10;i++) {
                  System.out.print(id+" ");}
            System.out.println(" Done in thread"+id+" ");
      }
}
```

```
public class PriorityDemo {

   public static void main(String arg[]) {
       Thread[] priDemo=new Thread[3];

       System.out.println("线程的初始优先级");

       for(int i=0;i<3;i++) {
              priDemo[i]=new PriThread(i+1);
              System.out.print(priDemo[i].getPriority()+" ");
       }

       System.out.println("\n修改优先级，线程1=Max, 线程2=Normal,线程3=Min");
       priDemo[0].setPriority(Thread.MAX_PRIORITY);
       priDemo[2].setPriority(Thread.MIN_PRIORITY);
       for(int i=0;i<3;i++)
              priDemo[i].start();
       }
}
```

```
【一次运行结果】
线程的初始优先级
5 5 5
修改优先级，线程1=Max, 线程2=Normal,线程3=Min
1 1 1 1 1 1 1 2 1 1 3  Done in thread1
2 3 2 3 2 3 2 2 2 2 3 3 3  Done in thread2
3 3  Done in thread3
```

- 优先级不能作为程序正确性的依赖，它高度依赖于宿主机平台的实现系统。
- Java优先级被映射到宿主机平台的优先级上，优先级个数可能更多，也可能更少，还可能被忽略（linuxJava虚拟机）。

第9章 线程与并发编程



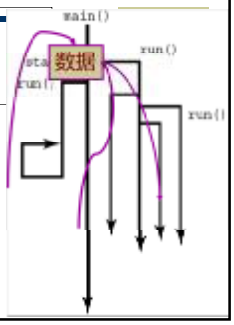

## 9.4线程同步

- **异步**（asynchronization）：是指每个线程包含了运行时所需要的数据和方法，不需要外部的资源和方法，也不会被其他线程的状态和行为所影响。前面提到的线程都是独立且异步执行的。（示例）
- 但在大多数多线程应用中，同时运行的线程需要共享数据，这时就需要考虑两个线程的访问次序，否则会导致数据的不一致，基本上，并发线程在解决共享冲突问题时，都是采用同步机制来解决该问题。
- **同步**（Synchronization）：用来保证进程之间执行顺序协调有序，并确保当两个或多个线程都要访问共享数据时，任何时刻只能有一个线程占用该共享资源，即**序列化访问共享资源**

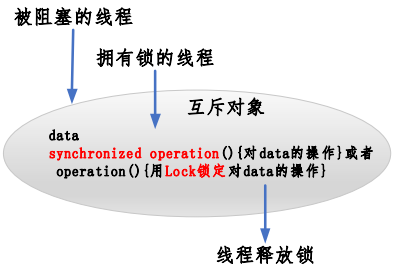
32

## 9.4线程同步

- 同步
  - 互斥-----Mutual exclusion
  - 协作-----Coordination




## 9.4.1线程互斥----Mutual exclusion




34

## 1、互斥对象

共享资源一般是以对象的形式存在。如果想控制对共享资源的访问，需要在共享资源类中把所有要访问这个资源的操作标记为synchronized(或用Lock)，该类生成的对象就是互斥对象。

```
class Guard{
        private int value; //共享数据（资源）
        public Guard(int e) {
                value=e;
        }
        public synchronized int getV() {  //互斥方法
                return value;
        }
        public synchronized void setV(int e) { //互斥方法
                value=e;
        }
}
```

## 2、Java中的锁

- 内置对象锁：synchronized关键字关联
- volatile域
- Lock锁/Condition条件

## 2、内置锁对象锁

Java中的对象和类都有对应的内置锁。

（1）Java中**每个对象**：都包含单一的锁（也称为监视器）是当前对象this，它自动成为对象的一部分。而锁与synchronized 关键字是相关联的。

（2）**每一个类**：有一个锁（作为类的Class对象的一部分）。所以，一个synchronized static方法可以锁定一个类，内置锁是这个类，防止同一个类的其他同步静态方法对static数据的并发访问。

比如类型为Person，静态锁为Person.class。所以，实例同步方法和static同步方法之间因为锁不同而互不干扰。

## 3、synchronized关键字

**（1）同步方法（synchronized method）**

总结一下，线程的互斥可以通过对他们共享数据的互斥封装来实现，互斥对象提供了同步方法去访问共享数据。同步方法的一般形式如下：

```
synchronized 返回值  方法名([参数列表]){
  //代码
  }
```

**（2）同步语句块（synchronized statement）**

为了提供更多的弹性，Java允许在方法内部定义同步语句块，其作用与修饰方法类似，只是作用范围不一样。该同步语句块相当并发编程中的于临界区。形式如下：

```
synchronized (其他对象/this/类){ //需要显式指明锁
  //代码
  }
```

38

```
class RunDemo implements Runnable{
   private int stack=5;
  public void run(){     //也可以定义为public synchronized run()

        String name=Thread.currentThread().getName();
        synchronized(this){
           for(int i=0;i<5;i++)
              System.out.println(name+"--stack is: "+(stack--));
        }
              }}

public class RunnableTest {
        public static void main(String arg[]){
              RunDemo rd=new RunDemo(); //共享对象
              Thread t1=new Thread(rd);
              Thread t2=new Thread(rd);
              t1.start();
              t2.start();
              }
}
```

## 运行结果

【运行结果】

```
Thread-0--stack is: 5
Thread-0--stack is: 4
Thread-0--stack is: 3
Thread-0--stack is: 2
Thread-0--stack is: 1
Thread-1--stack is: 0
Thread-1--stack is: -1
Thread-1--stack is: -2
Thread-1--stack is: -3
Thread-1--stack is: -4
```

【不加同步的一次运行结果】

```
Thread-1--stack is: 5
Thread-0--stack is: 4
Thread-1--stack is: 3
Thread-1--stack is: 1
Thread-0--stack is: 2
Thread-0--stack is: -1
Thread-0--stack is: -2
Thread-0--stack is: -3
Thread-1--stack is: 0
Thread-1--stack is: -4
```

## 9.4.2 线程协作

- 同步的目的之一是保持共享资源的一致性，同时它也可以使线程之间实现协作。
- 通过synchronized只实现了较低层次的互斥同步，这解决了共享数据一致性的问题，
- 下面我们需要进一步解决线程同步中如何使任务之间可以协调工作的线程协作问题。

## 举例：

考虑一下并发编程中典型的“生产者和消费者”协作问题。

- 线程A用烤箱烘焙面包，
- 然后线程B从烤箱中取走烤好的面包吃，
- 接着线程A又开始烘焙第二个面包。

很显然：

（1）两个线程需要互斥的访问烤箱，

（2）但另外一点也很清晰：线程B需要在线程A考好面包后才能吃到面包。

我们需要提供“面包已准备好”、“烤箱为空”这样的条件信息用于交流，同时要协调线程间的吃面包和烤面包的操作顺序。

## 9.4.2 线程协作

- 为了实现这种协作，线程通常需要检查一个包含在互斥方法中的条件，通常，线程进入临界区后，如果检查某一条件不满足，该线程会中断并等待。后续如果条件满足，线程会被唤醒继续执行。
- Java实现线程协作最简单的方式，是通过继承自Object类的wait()和notify/notifyAll方法来实现线程挂起和唤醒。

## 1、synchronized等待/通知机制


## 1、synchronized等待/通知机制

**2、wait方法**

wait方法是等待方执行的操作，等待方遵循如下的原则：

1）首先获取共享对象O的锁

2）查询条件，如果条件不满足，则调用对象O的wait方法，当前运行线程被阻塞，线程由运行态变为等待态，并将当前线程放置到对象O的等待队列（WaitQueue）。

3）如果条件满足，线程执行对应的逻辑。

对应的伪代码如下：

```
synchronized(对象) {
      while(条件不满足) {
            对象.wait();
      }
      应用处理逻辑
}
```

## 1、synchronized等待/通知机制

**notify/notifyAll方法**

- **通知方遵循如下原则：**

**1）获取对象的锁**

**2）改变条件**

**3）通知所有等待在对象上的线程**

**其伪代码如下：**

```
synchronized(对象){
  改变条件
  对象.notify()/ 对象.notifyAll();
 }
}
```

## Example 生产者/消费者


47

```
class CirBuf{//1、定义支持协作的互斥对象
      private int index=0; //缓冲区指针
      private int[] buf=new int[3]; //缓冲区，最大长度为3

      public synchronized void put(int value) {
         while(index==buf.length) { //查询条件，缓冲区已满
              try {
                    wait(); //生产者挂起等待
              }catch(InterruptedException e) {}
              }
         buf[index++]=value;//生产产品
         System.out.println("in:  producer "+ value);
         notify();//唤醒等待线程
      }
      public synchronized int get() {
          while(index==0) { //查询条件，缓冲区中无数据
              try {
                    wait();  //消费者挂起等待
              }catch(InterruptedException e) {}
              }
          index--; //改变条件，指针向下移动
1.        notify();//唤醒等待线程
2.        System.out.println("out: consumer "+ buf[index]);
3.        return buf[index]; //返回消费数据
      }}
```

```
class Consumer extends Thread{
       private CirBuf sb;
       public Consumer(CirBuf temp) {
              sb=temp;
       }
       public void run() {
              for(int i=0;i<2;i++) { //消费两次
                     sb.get();//消费
                     try {
                            Thread.sleep(10);
                     }catch(InterruptedException e) {}
       }
}

}
```

```
class Producer extends Thread{ //3、定义生产者
       private CirBuf sb;
       public Producer(CirBuf temp) {
              sb=temp;
       }
       public void run() {
              for(int i=1;i<=4;i++) { //生产四个产品
                     sb.put(i);//生产
                     try {
                            Thread.sleep(10);
                     }catch(InterruptedException e) {}
       }
}}
```

```
public class ThreadCooperation { //4、测试程序
       public static void main(String arg[]) {
              CirBuf  cb=new CirBuf();
              Thread pro=new Producer(cb);
              Thread con=new Consumer(cb);
              Thread con2=new Consumer(cb);
    //一个生产线程，两个消费线程并发执行。
              pro.start();
              con.start();
        con2.start();
       }}
```

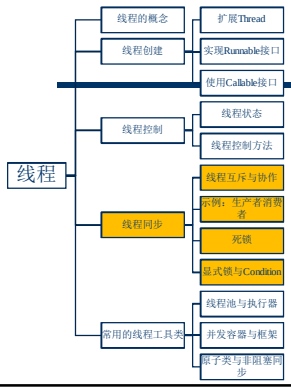
【运行结果】

```
in:  producer 1
out: consumer 1
in:  producer 2
out: consumer 2
in:  producer 3
out: consumer 3
in:  producer 4
out: consumer 4
```

## 9.4.4 死锁

- 锁是非常有用的工具，运用场景非常多，但同时它也会带来困扰，那就是可能会引发死锁，一旦产生死锁，就会造成系统功能不可用。
- 多个线程如果各自占有共享资源，同时又互相等待对方资源，在得到对方资源前不会释放自己的资源，从而导致都想得到资源而又都得不到的状态，这就是死锁。

## 第9章 线程与并发编程




## 9.4.5 显式锁:Lock(后续显式锁和条件Condition 自学教材相关知识。)

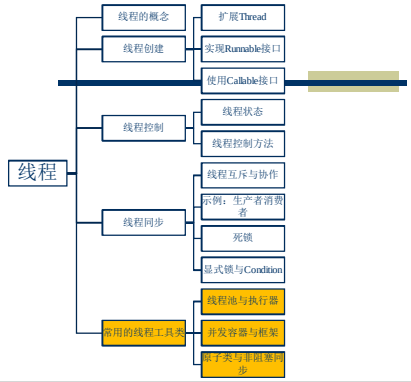
前面我们用synchronized关键字隐式地获取锁。这种方式虽然简化了同步的管理，但依然存在一些问题:

（1）如果涉及到多个共享资源交叉地加锁和解锁时，synchronized就不那么容易实现。

（2）线程由于某些特定原因发生阻塞，但没有释放锁，其他线程只能继续等待。

（3）共享资源读操作之间是不冲突的。但使用synchronized后，却只能互斥访问。

第9章 线程与并发编程




9.5 常用线程工具类

8.4 线程池

- 1 为什么需要线程池
- 2 创建、关闭、监控线程池
- 3 提交任务到线程池
- 4 使用线程池

57

1 为什么需要线程池

- 构建服务器应用程序时，如果每个请求到达就创建一个新的线程，处理请求，服务结束后就销毁线程，这会存在严重的性能缺陷。
  - 创建和销毁线程花费时间和系统资源
  - 创建太多线程会导致过度消耗内存或调度切换过度
- 服务器程序需要限制给定时刻处理请求的数目
  - 通过让多个任务重用线程，降低线程创建开销
  - 适当调整线程池中线程数目，请求数超过某个阈值，就强制新请求等待。
- java.util.concurrent.ThreadPoolExecutor
  - 管理线程的生命周期 和执行过程，维护一定量线程

58

- 小结
  - 线程定义：三种方法
  - 线程状态
  - 线程互斥：synchronized, Lock
  - 线程协作:Object的wait/notify或者notifyAll结合synchronized
  - 线程协作Condition的await/signal结合Lock