

程序中的并发 多线程程序设计

主讲老师：申雪萍



2021/6/1

Xueping Shen



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

- 进程的概念
- 线程的概念
- 线程的调度
- 创建和启动线程
 - Thread线程类、
 - Runnable接口
- 多个线程的同步
- 线程之间的通信

程序（静态蓝本）和进程（动态执行）

- 程序（program）：是一段静态的代码，是对数据描述与操作的代码的集合，它是应用程序执行的蓝本。
- 进程（process）是**程序的一次执行过程**，是系统运行程序的基本单位。程序是静态的，进程是动态的。**系统运行一个程序即是一个进程从创建、运行到消亡的过程。**
- 多任务（multi task）在一个系统中可以同时运行多个程序，即有多个独立运行的任务，每个任务对应一个进程。

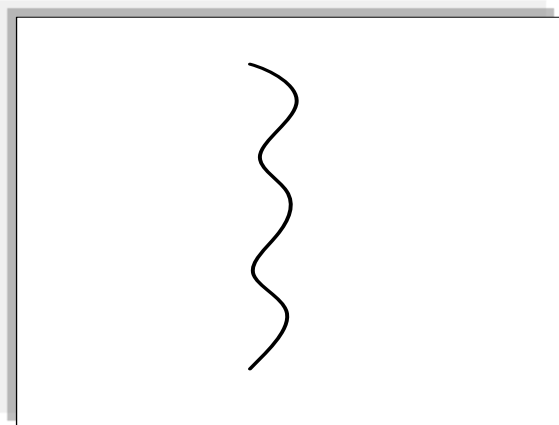
线程

- **线程**：比进程更小的执行单位，一个进程中可以包含多个线程。
 - 一个进程在执行过程中，为了同时完成多项操作，可以产生多个线程，形成多条执行线索。
 - 每个线程都有它自身的产生、存在和消亡的过程。
- 一个进程中的所有线程都在该进程的虚拟地址空间中，**使用该进程的全局变量和系统资源**。
- 操作系统给每个线程分配不同的CPU时间片，**在某一时刻，CPU只执行一个时间片内的线程，多个时间片中的相应线程在CPU内轮流执行**。

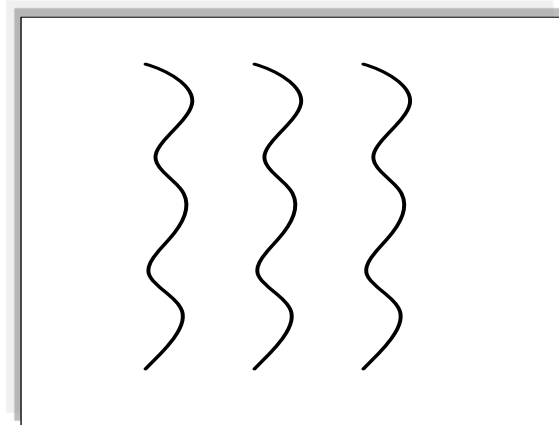
线程和进程之间的关系

- **多进程**环境中每一个进程既包括其所要执行的指令，也包括执行指令所需的任何系统资源，如CPU、内存空间、I/O端口等，**不同进程所占用的系统资源相对独立**；
- 线程是比进程单位更小的执行单位，多线程环境中每一个线程都隶属于某一进程，由进程触发执行，在系统资源的使用上，**属于同一进程的所有线程共享该进程的系统资源**；
 - 与进程不同的是线程本身即没有入口，也没有出口，其自身也不能独立运行，它栖身于某个进程之中，由进程启动运行，完成其任务后，自动终止，也可以由进程使之强制终止。

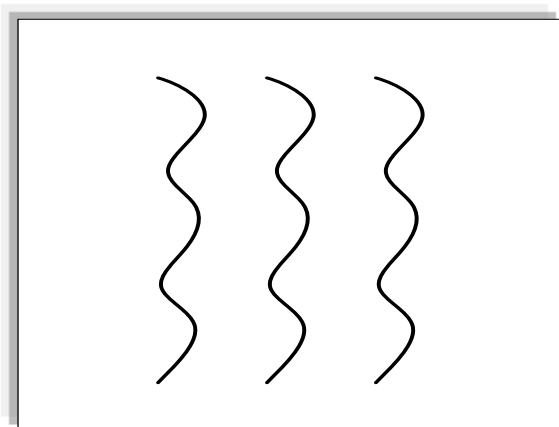
进程、线程示意图



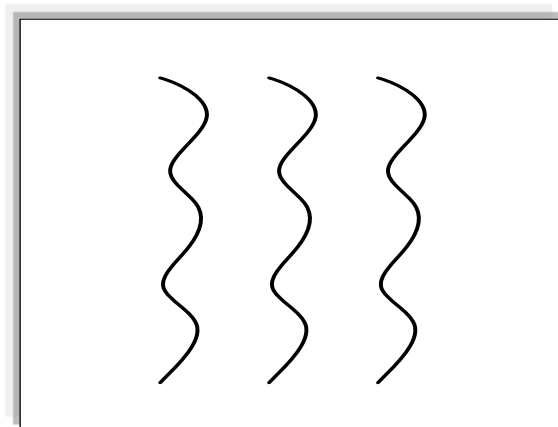
单进程 单线程



单进程 多线程



多进程 多线程



多线程程序设计

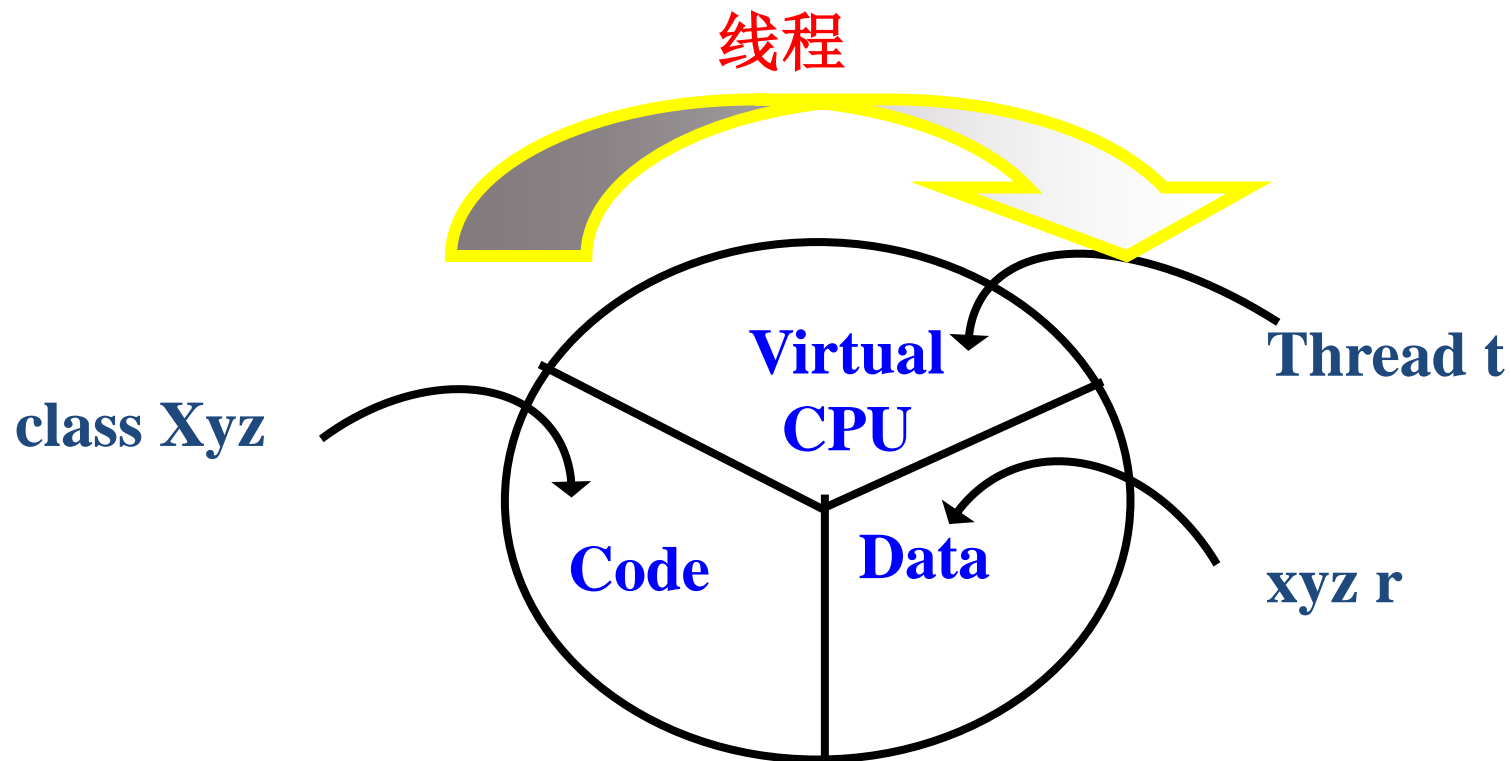
- 是指单个程序包含并发执行的多个线程。当多线程程序执行时，该程序对应的进程中就有多个控制流在同时执行，即具有并发执行的多个线程；
- 例如：
 - 影院售票问题
 - PV操作问题
 - Web Server接受客户端的请求问题
 - 银行问题
 - 网络聊天(一对多)程序

- 多位生产者生产产品给消费者消费，生产者将生产的产品放到产品线，多位消费者从产品线取走产品消费。
- 产品线上最多容纳100个产品；
- 当产品线上有空位时生产者才可以生产，否则要等待，同时给消费者发送消费的信号；
- 当产品线上有产品时消费者才可以消费，否则要等待，同时给生产者发送生产的信号；
- 产品线上同一时间只能有一个生产者或消费者；

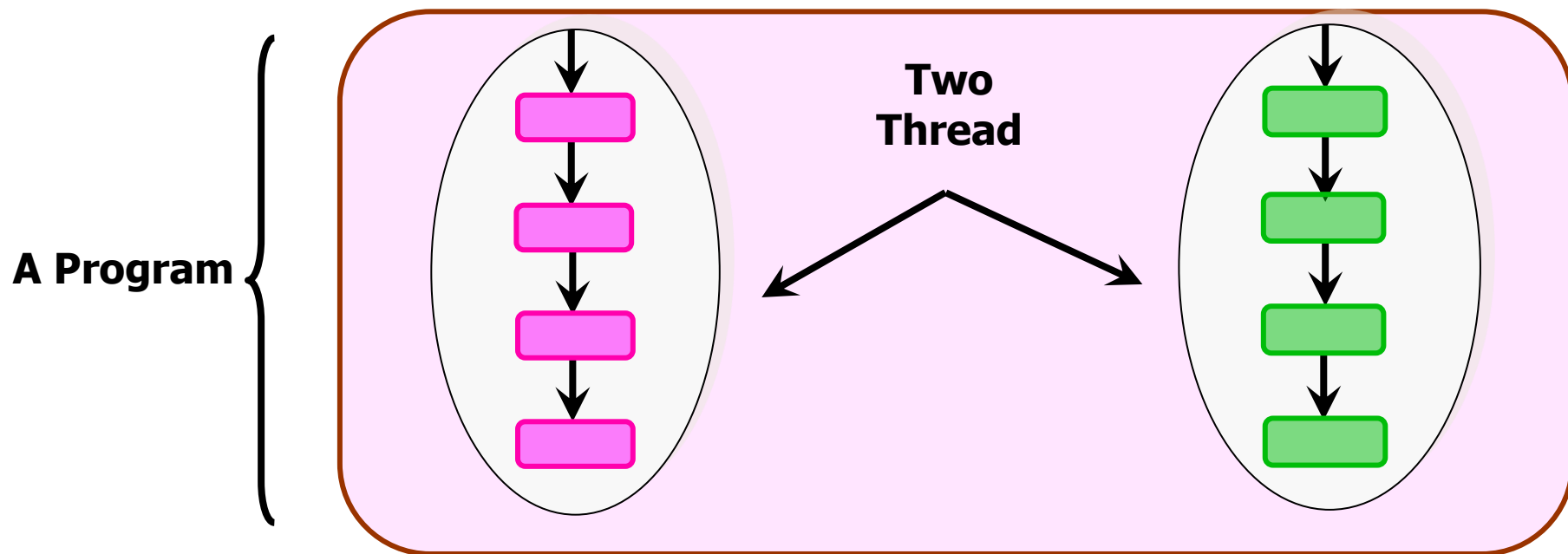
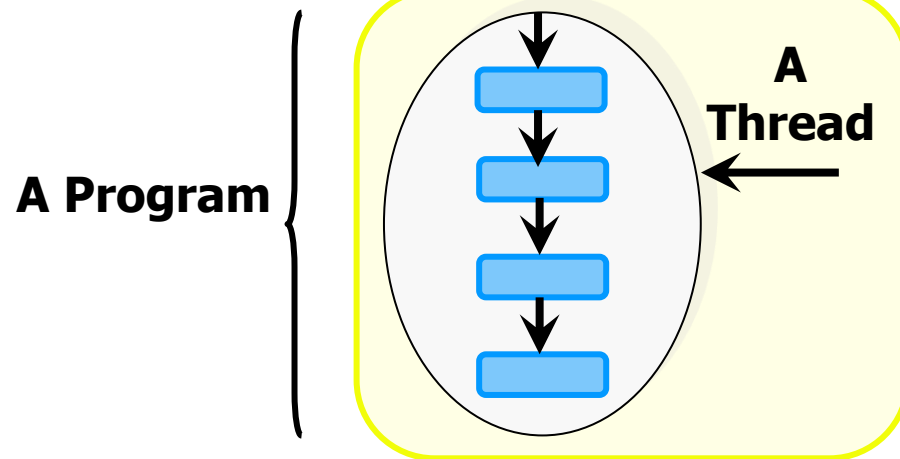
为什么要使用多线程？（系统开销）

- 线程之间共享相同的内存单元(代码和数据)，因此在线程间切换，不需要很大的系统开销，*所以线程之间的切换速度远远比进程之间快，线程之间的通信也比进程通信快的多。*
- 多个线程轮流抢占CPU资源而运行时，从*微观*上讲，一个时间里只能有一个作业被执行，在*宏观*上可使多个作业被同时执行，即等同于要让多台计算机同时工作，*使系统资源特别是CPU的利用率得到提高，从而提高整个程序的执行效率。*

线程运行环境



程序、进程和线程示意图



小结 (1)

1. 每一个进程就是一个应用程序。有自己的入口和出口。
2. 多进程环境中每一个进程既包括其所要执行的指令，也包括了执行指令所需的任何系统资源，如CPU、内存空间、I/O端口等；
3. 不同进程所占用的系统资源相对独立。
4. 线程之间切换的速度比进程切换要快得多（隶属于同一进程的线程）。

小结（2）

- 1.多个线程可共同存在于一个应用程序中。
- 2.多线程环境中每一个线程都隶属于某一进程,由进程触发执行（没有自己的入口和出口）；
- 3.在系统资源的使用上,属于同一进程的所有线程共享该进程的系统资源。

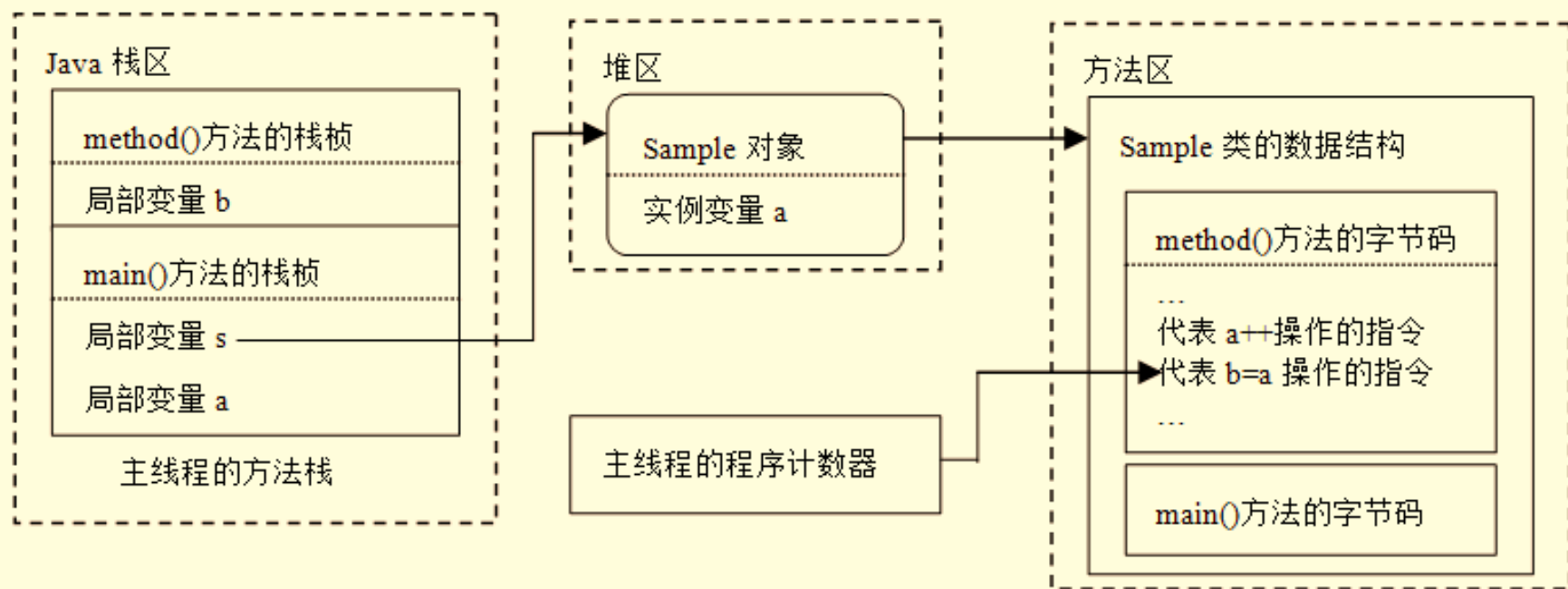
Java线程的运行机制

- 每个线程都有一个独立的程序计数器和方法调用栈（method invocation stack）：
 - 程序计数器：程序计数器（Program Counter Register）是一块较小的内存空间，它可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。
 - 方法调用栈：简称方法栈，用来跟踪线程运行中一系列的方法调用过程，栈中的元素称为栈帧。每当线程调用一个方法，就会向方法栈压入一个新帧，帧用来存储方法的参数、局部变量和运算过程中的临时数据。

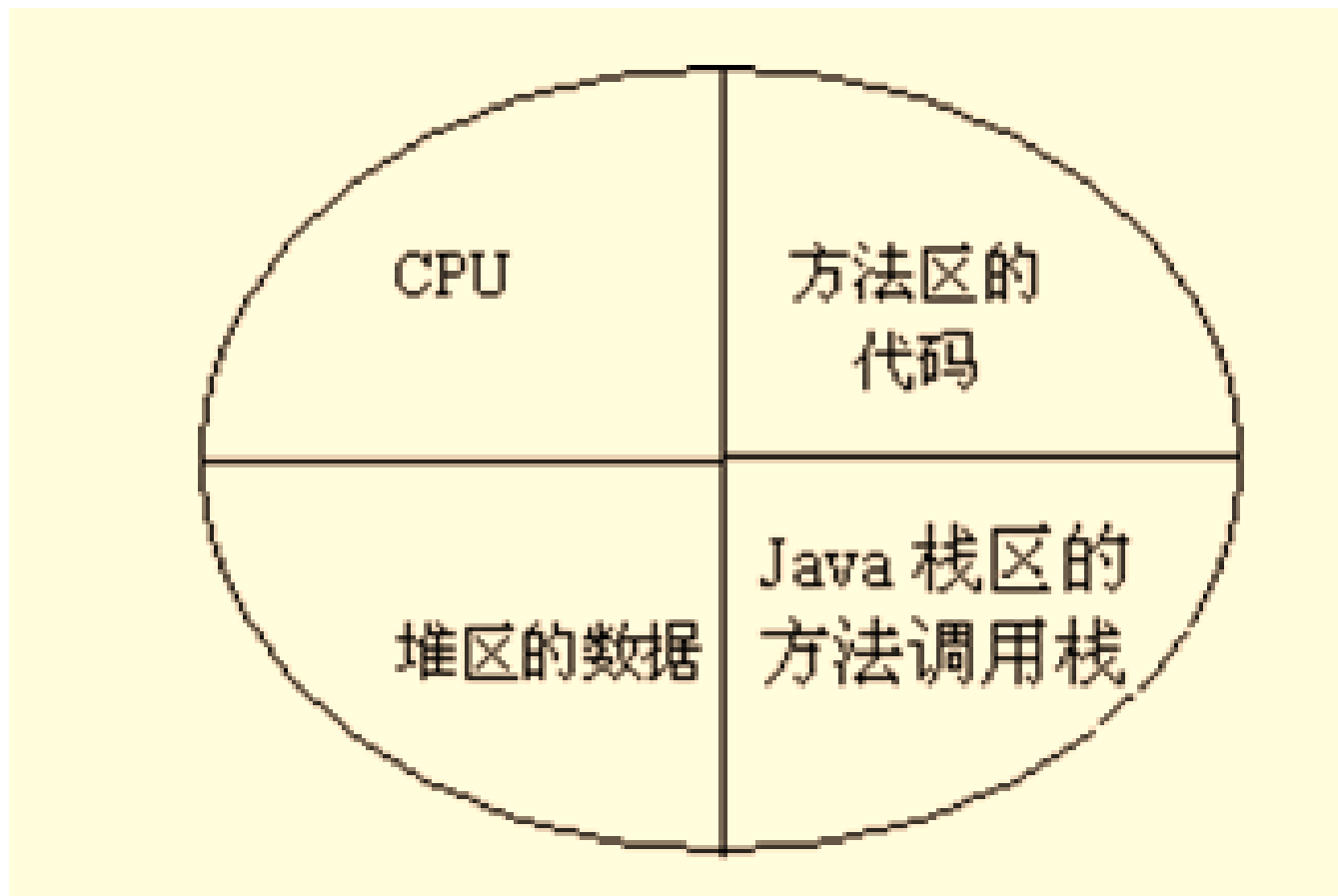
线程的栈帧

```
public class Sample{  
    private int a; //实例变量  
    public int method(){  
        int b=0; //局部变量  
        a++;  
        b=a;  
        return b;  
    }  
}
```

```
public static void main(String args[]){  
    Sample s=null; //局部变量  
    int a=0; //局部变量  
    s=new Sample();  
    a=s.method();  
    System.out.println(a);  
}
```



线程运行中需要的资源



主要内容

- 进程的概念
- 线程的概念
- **线程的调度**
- 创建和启动线程
 - Thread线程类、
 - Runnable接口
- 多个线程的同步
- 线程之间的通信

线程的调度 (1)

- 微观上，在一台只具有一个CPU的机器上，CPU在同一时间只能分配给一个线程做一件事。
- 当有多于一个的线程工作时，在Java中，线程调度通常是抢占式(即哪一个线程先抢到CPU资源则先运行)，而不是分时间片式。
- 一旦一个线程获得执行权，这个线程将持续运行下去，直到它运行结束或因为某种原因而阻塞，或者有另一个高优先级线程就绪（**这种情况称为低优先级线程被高优先级线程所抢占**）。

线程的调度 (2)

- 所有被阻塞的线程按次序排列，组成一个阻塞队列。例如：
 - 因为需要等待一个较慢的外部设备，例如磁盘或用户。
 - 让处于运行状态的线程调用 `Thread.sleep()` 方法。
 - 让处于运行状态的线程调用另一个线程的 `join()` 方法。

线程的调度 (3)

- 所有就绪但没有运行的线程则根据其优先级排入一个就绪队列。
- 当CPU空闲时，如果就绪队列不空，就绪队列中第一个具有最高优先级的线程将运行。
- 当一个线程被抢占而停止运行时，它的运行态被改变并放到就绪队列的队尾；
- 一个被阻塞（可能因为睡眠或等待I/O设备）的线程就绪后通常也放到就绪队列的队尾。

线程的调度与优先级 (1)

- 线程的调度是按：
 1. 其优先级的高低顺序执行的；
 2. 同样优先级的线程遵循“先到先执行的原则”；



线程的调度与优先级 (1)

- **线程优先级**：范围 1~10（10 级）。数值越大，级别越高
 - Thread 类定义的 3 个常数：
 - MIN_PRIORITY 最低(小)优先级（值为1）
 - MAX_PRIORITY 最高(大)优先级（值为10）
 - NORM_PRIORITY 默认优先级（值为5）
 - 线程创建时，继承父线程的优先级。
 - 常用方法：
 - getPriority()：获得线程的优先级
 - setPriority()：设置线程的优先级

ThreadPri.java

```
public class ThreadPri
{
    public static void main(String args[ ])
    {
        MyThread th[ ]=new MyThread[3];
        for(int i=0;i<3;i++){
            th[i]=new MyThread( );
            th[i].start( );
        }
        th[0].setPriority(Thread.MAX_PRIORITY);
        //th[1].setPriority(Thread.NORM_PRIORITY);
        th[2].setPriority(Thread.MIN_PRIORITY);
    }
}

class MyThread extends Thread
{
    public void run( )
    {
        for(int i=0;i<2;i++)
            System.out.println(getName( )+"线程的优先级是: \t"+getPriority( )+"。");
    }
}
```

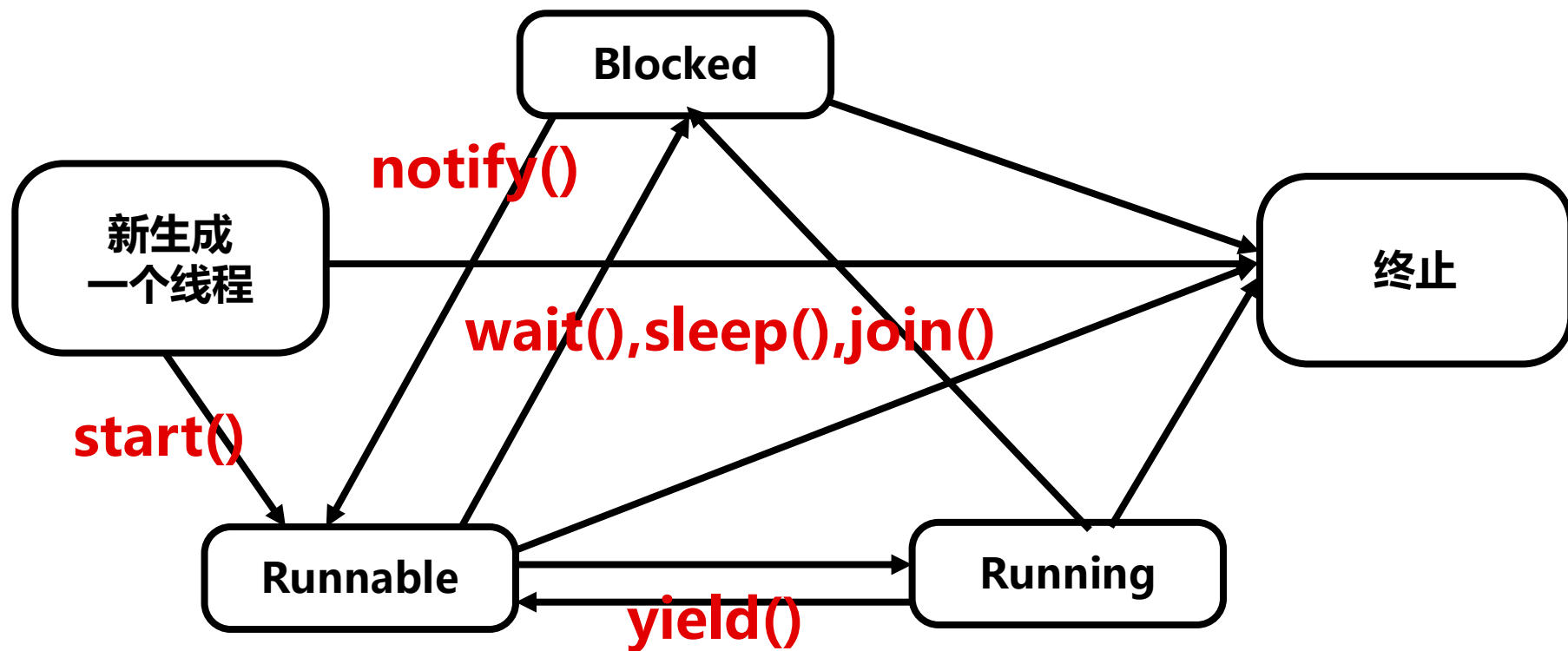
```
Thread-1线程的优先级是: 10。
Thread-1线程的优先级是: 10。
Thread-2线程的优先级是: 5。
Thread-2线程的优先级是: 5。
Thread-3线程的优先级是: 1。
Thread-3线程的优先级是: 1。
```

- **main() 方法——Application 应用程序**
 - 每当用java命令启动一个Java虚拟机进程（Application 应用程序），Java虚拟机就会创建一个主线程，该线程从程序入口main()方法开始执行。
- 当在主线程中创建 Thread 类或其子类对象时，就创建了一个线程对象。主线程就是上述创建线程的父线程。
- Programmer可以控制线程的启动、挂起与终止。

线程的状态

- 在一个多线程运行环境中运行的线程可以有多种状态,也就是说一个线程在不同时刻会处于下列状态之一:
 - 新建、就绪、运行、阻塞、终止;

线程的状态



线程生命周期中的5种状态(1)

- **新建**：当一个 Thread 类或其子类对象被创建时，新产生的线程处于新建状态，此时它已经有了相应的内存空间和其他资源。
如： Thread myThread=new Thread();
- **就绪**：调用 start() 方法来启动处于新建状态的线程后，将进入线程队列排队等待 CPU 服务，此时它已经具备了运行的条件，一旦轮到它来享用 CPU 资源时，就可以脱离创建它的主线程，开始自己的生命周期。

线程生命周期中的5种状态(2)

- **运行**：当就绪状态的线程被调度并获得处理器资源时，便进入运行状态。
- 每一个 Thread 类及其子类的对象都有一个重要的 run() 方法，当线程对象被调用执行时，它将自动调用本对象的 run() 方法，从第一句开始顺序执行。
- Run() 方法定义了这个线程的操作和功能。

线程生命周期中的5种状态(3)

- **阻塞：一个正在执行的线程暂停自己的执行而进入的状态。引起线程由运行状态进入阻塞状态的可能情况：**
 - 该线程正在等待 I/O 操作的完成
 - 网络操作
 - 为了获取锁而进入阻塞操作
 - 调用了该线程的 `sleep()` 方法
 - 调用了 `wait()` 方法
 - 让处于运行状态的线程调用另一个线程的 `join()` 方法

线程生命周期中的5种状态(4)

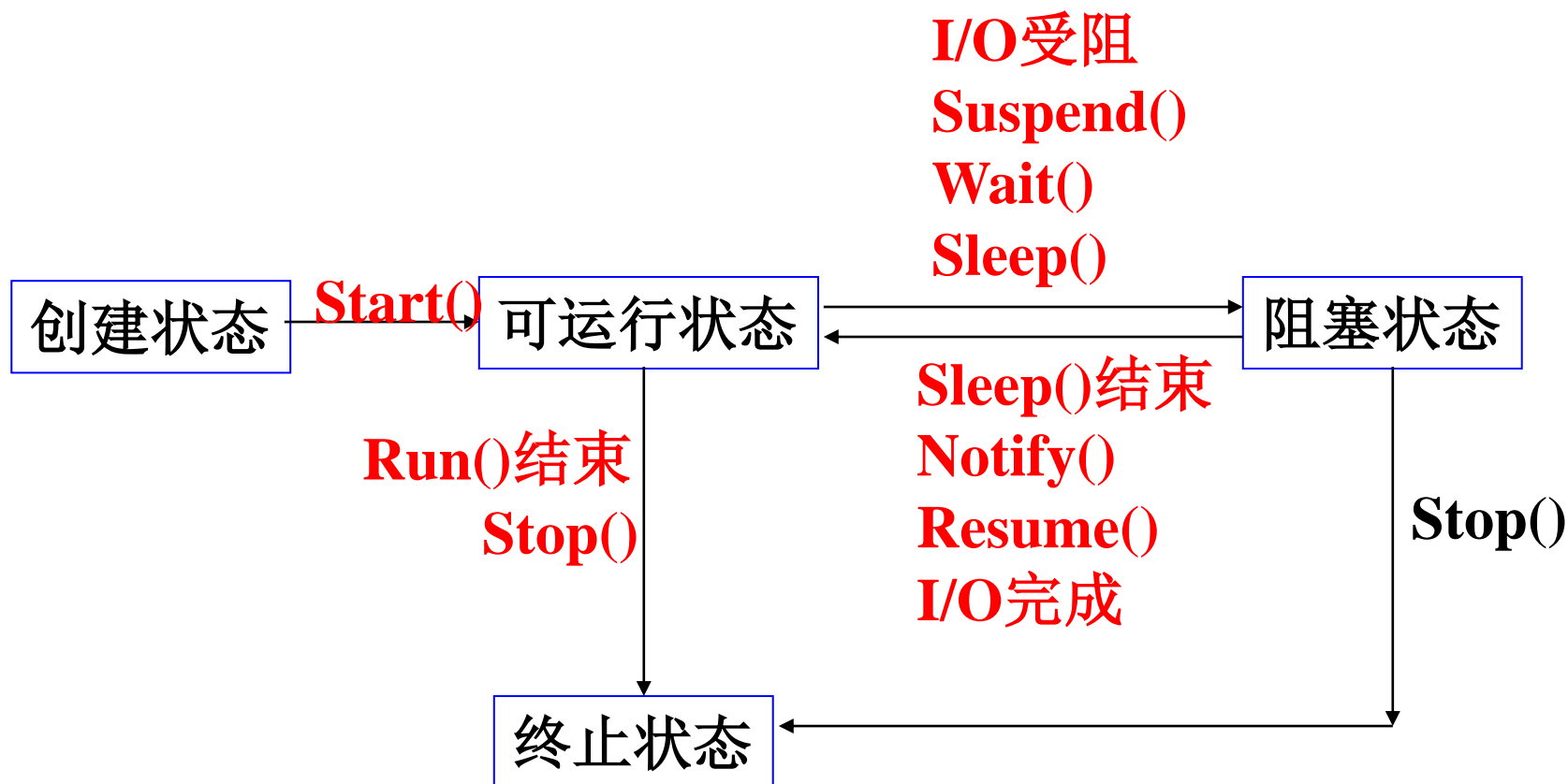
- 对应于不同进入阻塞状态的情况，采取不同的方法使其回到就绪状态：
 - I/O 操作：等待这个 I/O 操作完成后；
 - sleep() 方法：等待其指定的休眠事件结束后，自动脱离阻塞状态；
 - 调用 wait() 方法：调用 notify() 或 notifyAll() 方法；

线程生命周期中的5种状态(5)

- **终止:**

- **自然终止:** 线程完成了自己的全部工作
- **强制终止:** 在线程执行完之前, 调用 `stop()` 或 `destroy()` 方法终止线程

线程的状态及转换



主要内容

- 进程的概念
- 线程的概念
- 线程的调度
- **创建和启动线程**
 - Thread线程类、
 - Runnable接口
- 多个线程的同步
- 线程之间的通信



创建一个新的线程

- Java中的多线程是建立在**Thread类,Runnable接口**的基础上的，通常有两种办法让我们来创建一个新的线程：
 - 创建一个Thread类,或者一个Thread子类的对象；
 - 创建一个实现Runnable接口的类的对象；

Thread类介绍

Thread的构造方法

Thread();

Thread(Runnable target);

Thread(Runnable target, String name);

Thread(String name);

Thread(ThreadGroup group, Runnable target);

**Thread(ThreadGroup group, Runnable target,
String name);**

**Thread(ThreadGroup group, Runnable target,
String name, long stackSize);**

Thread(ThreadGroup group, String name);

name:线程的名称, **target:**要运行的线程对象, **group:**线程组的名称, **stackSize:**堆栈的大小

Thread构造方法

- Thread的构造方法一共有八个，这里根据命名方式分类，使用默认命名的构造方法如下
 - Thread()
 - Thread(Runnable target)
 - Thread(ThreadGroup group,Runnable target)
- 命名线程的构造方法如下：
 - Thread(String name)
 - Thread(Runnable target,String name)
 - Thread(ThreadGroup group,String name)
 - Thread(ThreadGroup group,Runnable target,String name)
 - Thread(ThreadGroup group,Runnable target,String name,long stackSize)

Thread构造方法

- 一个线程的创建肯定是由另一个线程完成的；
- 被创建线程的父线程是创建它的线程；
- 而main线程由JVM创建，而main线程又可以成为其他线程的父线程；
- 如果一个线程创建的时候没有指定ThreadGroup，那么将会和父线程同一个ThreadGroup。main线程所在的ThreadGroup称为main；

关于stackSize（了解）

- Thread构造方法中有一个stackSize参数，该参数指定了JVM分配线程栈的地址空间的字节数，对平台依赖性较高，在一些平台上：
 - 设置较大的值：可以使得线程内调用递归深度增加，降低StackOverflowError出现的概率
 - 设置较低的值：可以使得创建的线程数增多，可以推迟OutOfMemoryError出现的时间
- 但是，在一些平台上该参数不会起任何作用。另外，如果设置为0也不会起到任何作用。

Thread类介绍

Thread的主要方法(共**34**个,其中**5**个不再使用了)

static Thread currentThread();	int getPriority();
---------------------------------------	---------------------------

String getName();	ThreadGroup getThreadGroup();
--------------------------	--------------------------------------

void interrupt();	boolean isInterrupted();
--------------------------	---------------------------------

void join();	join(long millis);	join(long millis, int nanos);
---------------------	---------------------------	--------------------------------------

static void yield();	void setName(String name);
-----------------------------	-----------------------------------

void setPriority(int priority);	void start();
--	----------------------

boolean isAlive();	static void sleep(long millis, int nanos);
---------------------------	---

void run();	static void sleep(long millis);
--------------------	--

countStackFrames();	suspend();	resume();
stop();	stop(Throwable obj);	

Thread类的常用方法(1)

1. **currentThread(): 返回当前运行的Thread对象。**
2. **setName(): 设置线程的名字。**
3. **getName(): 返回该线程的名字。**
4. **setPriority(): 设置线程优先级。**
5. **getPriority(): 返回线程优先级。**
6. **start(): 启动一个线程。**
7. **run(): 线程体,由start()方法调用,当run()方法返回时,当前的线程结束。**
8. **stop(): 使调用它的线程立即停止执行。**
9. **isAlive(): 如果线程已被启动并且未被终止,那么isAlive(): 返回true。如果返回false, 则该线程是新创建或是已被终止的。**

Thread类的常用方法(1)

- 10. **sleep(int n):** 使线程睡眠n毫秒,n毫秒后,线程可以再次运行。
- 11. **yield():** 将CPU控制权主动移交到下一个可运行线程。
- 12. **join():** 方法join()将引起现行线程等待,直至方法join所调用的线程结束。
- 13. **suspend():** 使线程挂起,暂停运行。
- 14. **resume()** 恢复挂起的线程,使其处于可运行状态 (Runnable) 。
- 15. **wait():**
- 16. **notify():**
- 17. **notifyAll():**

suspend();

resume();

Stop();

目前已经不再使用。为什么? 请看API.

注意事项

- 因为Java线程的调度不是分时的，所以你必须确保你的代码中的线程会不时地给另外一个线程运行的机会。
- 有三种方法可以做到一点：
 - ①让处于运行状态的线程调用Thread.sleep()方法。
 - ②让处于运行状态的线程调用Thread.yield()方法。
 - ③让处于运行状态的线程调用另一个线程的join()方法。

sleep()和yield()的区别 (1)

- ① 这两个方法都是静态的实例方法。
- ② sleep()使线程转入阻塞状态，而yield()使线程转入runnable状态。
- ③ yield()给相同优先级或更高的线程运行机会，如果当前没有存在相同优先级的线程，则yield()什么都不做。而sleep()不会考虑线程的优先级，会给其他线程运行的机会，因此也会给相同或更低优先级线程运行机会。

sleep()和yield()的区别 (2)

- ④ **sleep()会有中断异常抛出，而yield()不抛出任何异常。**
- ⑤ **sleep()方法具有更好的可移植性，因为yield()的实现还取决于底层的操作系统对线程的调度策略。**
- ⑥ **对于yield()的主要用途是在测试阶段，人为的提高程序的并发性能，以帮助发现一些隐藏的并发错误，当程序正常运行时，则不能依靠yield方法提高程序的并发性能。**

方法wait()与sleep()方法的对比 (1)

- 相同点
 - 方法wait()与sleep()方法一样，都能使线程等待而停止运行

方法wait()与sleep()方法的对比 (2)

- 其区别在于：
 - **sleep()方法不会释放对象的锁，而wait()方法进入等待时，可以释放对象的锁，因而别的线程能对这些加锁的对象进行操作。**
 - **所以，wait,notify和notifyAll都是与同步相关联的方法,只有在synchronized方法中才可以用。在不同步的方法或代码中则使用sleep()方法使线程暂时停止运行。**

sleep()方法

- sleep()有两个重载方法：
 - sleep(long mills)
 - sleep(long mills,int nanos)
- 但是在JDK1.5后，引入了TimeUnit，其中对sleep()方法提供了很好的封装，建议使用TimeUnit.XXXX.sleep()去代替Thread.sleep():
 - TimeUnit.SECONDS.sleep(1);
 - TimeUnit.MINUTES.sleep(3);

join() 方法:

- **作用：使当前正在运行的线程暂停下来，等待指定的时间后或等待调用该方法的线程结束后，再恢复运行**

1. **public final void join()throws InterruptedException;**
2. **public final void join(long millis,int nanos)throws InterruptedException;**
3. **public final void join(long millis)throws InterruptedException;**

Deprecated. *This method has been deprecated, as it is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see [Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?](#).*

String **toString()**

Returns a string representation of this thread, including the thread's name, priority, and thread group.

static void yield()

Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Methods inherited from class java.lang.Object

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

应用线程类Thread创建线程

1. 将一个类定义为Thread的子类,那么这个类就可以用来创建线程。
2. 这个类中有一个**至关重要的方法**——**public void run**, 这个方法称为**线程体**, 它是整个线程的**核心**, **线程**所要完成任务的代码都定义在**线程体**中, 实际上**不同功能的线程之间的区别**就在于它们线程体的不同

应用线程类创建线程

创建线程对象

线程子类名 线程对象名
=new 线程子类名 (实际参数)

启动一个线程

线程对象名.start();

线程体的执行

Run()方法的执行

示例代码

```
import java.util.*;
class TimePrinter extends Thread {
    int pauseTime;
    String name;
    public TimePrinter(int x, String n) {
        pauseTime = x;
        name = n;
    }

    public void run() {
        while(true) {
            try {
                System.out.println(name + ":" + new Date(System.currentTimeMillis()))
                Thread.sleep(pauseTime);
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
```

Thread.sleep()方法的使用，注意异常的抛出

示例代码

```
static public void main(String args[]) {  
    TimePrinter tp1 = new TimePrinter(1000, "Fast Guy");  
    tp1.start();  
    TimePrinter tp2 = new TimePrinter(3000, "Slow Guy");  
    tp2.start();  
}
```


运行 结果

Fast	Guy:Mon	May	31	15:29:21	CST	2021
Slow	Guy:Mon	May	31	15:29:21	CST	2021
Fast	Guy:Mon	May	31	15:29:22	CST	2021
Fast	Guy:Mon	May	31	15:29:23	CST	2021
Slow	Guy:Mon	May	31	15:29:24	CST	2021
Fast	Guy:Mon	May	31	15:29:24	CST	2021
Fast	Guy:Mon	May	31	15:29:25	CST	2021
Fast	Guy:Mon	May	31	15:29:26	CST	2021
Slow	Guy:Mon	May	31	15:29:27	CST	2021
Fast	Guy:Mon	May	31	15:29:27	CST	2021
Fast	Guy:Mon	May	31	15:29:28	CST	2021
Fast	Guy:Mon	May	31	15:29:29	CST	2021
Slow	Guy:Mon	May	31	15:29:30	CST	2021
Fast	Guy:Mon	May	31	15:29:30	CST	2021
Fast	Guy:Mon	May	31	15:29:31	CST	2021
Fast	Guy:Mon	May	31	15:29:32	CST	2021



示例代码

```
import java.util.*;
class TimePrinter3 extends Thread {
    int pauseTime;
    String name;
    public TimePrinter3(int x, String n) {
        pauseTime = x;
        name = n;
    }
    public void run() {
        while(true) {
            System.out.println(name + ":" + new Date(System.currentTimeMillis()))
            yield();
        }
    }
    static public void main(String args[]) {
        TimePrinter tp1 = new TimePrinter(1000, "Fast Guy");
        tp1.start();
        TimePrinter tp2 = new TimePrinter(3000, "Slow Guy");
        tp2.start();
    }
}
```



Thead.yield()
方法

```
public class TestSubThread3{  
    public static void main(String args[]){  
        SubThread t1=new SubThread("线程_1");  
        SubThread t2=new SubThread("线程_2");  
        t1.start();  
        t2.start();  
        System.out.println("main()方法运行完毕!");  
    }  
}  
class SubThread extends Thread{  
    String str;  
    public SubThread(String s){  
        str=s;  
    }  
    public void run(){  
        for(int i=1;i<=3;i++){  
            for(long j=1;j<1000000000;j++)  
                ;  
            System.out.println(str+": 第"+i+"次运行");  
            System.out.println(str+": 结束!");  
        }  
    }  
}
```

主线程和其他
线程共存

运行结果

```
main()方法运行完毕!  
线程_2: 第1次运行  
线程_1: 第1次运行  
线程_2: 第2次运行  
线程_1: 第2次运行  
线程_1: 第3次运行  
线程_1: 结束!  
线程_2: 第3次运行  
线程_2: 结束!
```

示例代码

```
public class TestSubThread1{
    public static void main(String[] args)throws InterruptedException{
        SubThread t1=new SubThread("First 线程");
        SubThread t2=new SubThread("Second 线程");
        t1.start();
        t2.start();
        Thread.sleep(5000);
        System.out.println("Main()方法运行完毕! ");
    }
}
class SubThread extends Thread{
    String str;
    public SubThread(String s){
        str=s;
    }
    public void run(){
        for(int i=1;i<=3;i++){
            for(long j=1;j<1000000000;j++)
                ;
            System.out.println(str+": 第"+i+"次运行");
        }
        System.out.println(str+": 结束! ");
    }
}
```

Thread.sleep()方法的使用，注意异常的抛出

运行结果

```
First 线程: 第1次运行
Second 线程: 第1次运行
main()方法运行完毕!
First 线程: 第2次运行
Second 线程: 第2次运行
First 线程: 第3次运行
First 线程: 结束!
Second 线程: 第3次运行
Second 线程: 结束!
```

示例代码

```
public class TestSubThread{
    public static void main(String args[]) {
        SubThread t1=new SubThread("线程_1");
        SubThread t2=new SubThread("线程_2");
        t1.start();
        t2.start();
        try{
            t1.join();
            t2.join();
        }
        catch (InterruptedException e) {
            System.out.println(e.toString());
        }
        System.out.println("main()方法运行完毕！");
    }
}
```

join()方法的使用，注意异常的抛出。保证了主线程最后结束

```

class SubThread extends Thread{
    String str;
    public SubThread(String s){
        str=s;
    }
    public void run(){
        for(int i=1;i<=3;i++){
            for(long j=1;j<1000000000;j++)
                ;
            System.out.println(str+"： 第"+i+"次运行");
        }
        System.out.println(str+"： 结束！");
    }
}

```

```

线程_2： 第1次运行
线程_1： 第1次运行
线程_2： 第2次运行
线程_1： 第2次运行
线程_2： 第3次运行
线程_2： 结束！
线程_1： 第3次运行
线程_1： 结束！
main()方法运行完毕！

```

示例代码

```
public class TestMyThread{
    public static void main(String args[]){
        MyThread t=new MyThread(5);
        System.out.println("线程名:"+t.getName());//得到线程的名字
        //线程t没有启动, t.isAlive()得值为false
        System.out.println("线程启动?    "+t.isAlive());
        t.start();//线程t已启动, t.isAlive()得值为true
        System.out.println("线程启动?    "+t.isAlive());
        System.out.println("线程是否中断?" +t.isInterrupted());
        t.interrupt();
        System.out.println("线程是否中断?" +t.isInterrupted());
    }
}
```

```

class MyThread extends Thread{
    MyThread(int priority){
        setPriority(priority);
        setName("MyThread-"+priority);
    }
    public void run(){
        System.out.println(getName()+"运行");
        try{
            sleep(1000);
        }
        catch(InterruptedException e){
            System.out.println("线程休眠时调用interrupt()方法会抛出异常");
        }
        System.out.println(getName()+"结束!");
    }
}

```

```

线程名:MyThread-5
线程启动?      false
线程启动?      true
线程是否中断?false
线程是否中断?true
MyThread-5运行
线程休眠时调用interrupt()方法会抛出异常
MyThread-5结束!

```

应用Runnable接口创建线程

1. Runnable是Java中用以实现线程的接口，从根本上讲，任何实现线程功能的类都必须实现该接口。

- **Thread(Runnable target);**
- **Thread(Runnable target, String name);**

2. Runnable接口中只定义了一个方法就是run()方法，也就是线程体。

示例代码

```
import java.util.*;

class TimePrinter1 implements Runnable {
    int pauseTime;
    String name;
    public TimePrinter1(int x, String n) {
        pauseTime = x;
        name = n;
    }
    public void run() {
        while(true) {
            try {
                System.out.println(name + ":" + new
                    Date(System.currentTimeMillis()));
                Thread.sleep(pauseTime);
            }
            catch(Exception e) {
                System.out.println(e);
            }
        }
    }
    static public void main(String args[]) {
        Thread t1 = new Thread(new TimePrinter1(1000, "Fast Guy"));
        t1.start();
        Thread t2 = new Thread(new TimePrinter1(3000, "Slow Guy"));
        t2.start();
    }
}
```

示例代码

```
class MyThread implements Runnable{
    String str;
    int count;
    public MyThread(String s,int count){
        str=s;
        this.count=count;
    }
    public void run(){
        for(int i=1;i<=5;i++){
            for(long j=1;j<1000000000;j++)
                ;
            System.out.println(Thread.currentThread().getName()+"： 第"+i+"次运行");
        }
        System.out.println(Thread.currentThread().getName()+"： 结束！");
    }
}
```

```
public class TestMyThread{
    public static void main(String args[])throws InterruptedException{
        MyThread parent1=new MyThread("线程_1",100);
        Thread t1=new Thread(parent1,"t1");
        Thread t2=new Thread(parent1,"t2");

        MyThread parent2=new MyThread("线程_2",200);
        Thread t3=new Thread(parent2,"t3");
        Thread t4=new Thread(parent2,"t4");

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t1.join();
        t2.join();
        t3.join();
        t4.join();
        System.out.println(Thread.currentThread().getName()+"方法运行完毕！");
    }
}
```

运行结果

```
t1: 第1次运行  
t4: 第1次运行  
t3: 第1次运行  
t2: 第1次运行  
t1: 第2次运行  
t4: 第2次运行  
t3: 第2次运行  
t2: 第2次运行  
t1: 第3次运行  
t4: 第3次运行  
t2: 第3次运行  
t3: 第3次运行  
t1: 第4次运行  
t4: 第4次运行  
t3: 第4次运行  
t2: 第4次运行  
t1: 第5次运行  
t1: 结束!  
t3: 第5次运行  
t3: 结束!  
t2: 第5次运行  
t2: 结束!  
t4: 第5次运行  
t4: 结束!  
main方法运行完毕!
```



适用于采用实现Runnable接口方法的情况

- ① 因为Java只允许单继承，如果一个类已经继承了Thread，就不能再继承其他类。
- ② 特别是在除了run()方法以外，并不打算重写Thread类的其它方法的情况下，以实现Runnable接口的方式生成新线程就显得更加合理了。

终止线程

- 当线程执行完run()方法，它将自然终止运行。
- Thread有一个stop()方法，可以强制结束线程，但这种方法是不安全的。因此，在stop()方法已经被废弃。
- 实际编程中，一般是定义一个标志变量，然后通过程序来改变标志变量的值，从而控制线程从run()方法中自然退出。

示例代码

```
public class MyThreadStop extends Thread{
    int a; boolean flag=false;
    public void run(){
        while(!flag){

            System.out.println(a++);
        }
    }
    public void setFlag(boolean _flag){
        flag=_flag;
    }
    public static void main(String args[]){
        MyThreadStop t=new MyThreadStop();
        t.start();
        try{ Thread.sleep(1000);}catch(Exception e){}
        t.setFlag(true);
    }
}
```

小结：创建用户多线程的步骤（1）

1. 创建一个Thread类的子类
2. 在子类中将希望该线程做的工作写到run()里面
3. 生成该子类的一个对象
4. 调用该对象的start()方法

```
class MyThread extends Thread{  
    public void run(){... ..}  
    //其它方法等  
}
```

省略号代表的是我们想让这个线程完成的工作

```
class MyClass{  
    public static void main(String[] args){  
        MyThread mt = new MyThread();  
        mt.start();  
    }  
    //其它方法等  
}
```

调用start(),就会生成一个新的线程,并开始执行run()里规定的任务

小结：创建用户多线程的步骤(2)

1. 创建一个实现Runnable接口的类
2. 在该类中将希望该线程做的工作写到run()里面
3. 生成该类的一个对象
4. 用上述对象去生成Thread类的一个对象
5. 调用Thread类的对象的start()方法

```
class MyRunnable implements Runnable{  
    public void run(){... ...}  
    //其它方法等  
}  
class MyClass{  
    public static void main(String[] args){  
        MyRunnable mr = new MyRunnable();  
        Thread t = new Thread(mr);  
        t.start();  
    }  
    //其它方法等  
}
```

省略号代表的是我们想让这个线程完成的工作

调用Thread对象的start(),就会生成一个新的线程,并开始执行MyRunnable类的run()里规定的任务

小结：创建用户多线程的步骤(3)

1. 创建一个实现Runnable接口的类
2. 在该类中将希望该线程做的工作写到run()里面
3. 写一个start()方法,在里面创建Thread,调用start()
4. 生成Runnable类的一个对象
5. 调用该类对象的start()方法

```
class MyRunnable implements Runnable{
```

```
    public void run(){... ...}
```

```
    public void start(){
```

```
        new Thread(this).start();
```

```
    }
```

```
    //其它方法等
```

```
}
```

```
class MyClass{
```

```
    public static void main(String[] args){
```

```
        MyRunnable mr = new MyRunnable();
```

```
        mr.start();
```

```
    }
```

```
    //其它方法等
```

```
}
```

省略号代表的是我们想让这个线程完成的工作

以this为参数生成一个Thread类的对象,并调用它的start()方法

调用start(),就会生成一个新的线程,并开始执行run()里规定的任务

线程的互斥锁和线程同步 (synchronized)



2021/6/1

Xueping Shen



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

背景

- 在多线程环境中, 两个或者两个以上的线程访问一个共享对象的情况是很普遍的
- 比如某公司在银行里有一个帐号, 其在北京和上海的营业部都可以使用, 那么如果北京的营业部在存款的时候, 上海的营业部正好在取款, 会发生什么事呢?

- 带着这个问题, 我们再来看看多线程是怎样运行的:
 - ① 各个线程是通过竞争CPU时间而获得运行机会的;
 - ② 各线程什么时候得到CPU时间, 占用多久, 是不可预测的;
 - ③ 一个正在运行着的线程在什么地方被暂停是不确定的。
- 共享资源对象, 如果不协调这些操作的话, 就不能保证运行结果的正确。

示例代码:[ThreadDemo.java](#)

```
public class ThreadDemo{
    public static void main(String[] args){
        TestThread tt=new TestThread();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
    }
}
class TestThread implements Runnable
{
    int tickets =100;
    public void run(){
        while(true){
            if(tickets>0){
                try{Thread.sleep(10);}
                catch(Exception e){}
                System.out.println(Thread.currentThread().getName()+"is sailing ticket"+tickets--);
            }
        }
    }
}
```



运行结果

```
Thread-4is sailing ticket21
Thread-1is sailing ticket20
Thread-2is sailing ticket19
Thread-3is sailing ticket18
Thread-4is sailing ticket17
Thread-1is sailing ticket16
Thread-2is sailing ticket15
Thread-3is sailing ticket14
Thread-4is sailing ticket13
Thread-1is sailing ticket12
Thread-2is sailing ticket11
Thread-3is sailing ticket10
Thread-4is sailing ticket9
Thread-1is sailing ticket8
Thread-2is sailing ticket7
Thread-3is sailing ticket6
Thread-4is sailing ticket5
Thread-1is sailing ticket4
Thread-2is sailing ticket3
Thread-3is sailing ticket2
Thread-4is sailing ticket1
Thread-1is sailing ticket0
Thread-2is sailing ticket-1
Thread-3is sailing ticket-2
```

线程的互斥锁

- 同步是一种保证共享资源完整性的手段；
- 具体作法是在代表原子操作的程序代码前加上synchronized标记，这样的代码被称为同步代码块。
- Java中用关键字synchronized为共享资源加锁，在任何一个时刻只能有一个线程能取得加锁对象的钥匙，对加锁对象进行操作。从而达到了对共享资源访问时的保护。

线程的互斥锁

- synchronized关键字可以使用在：
 - ① 1. 一个成员方法上
 - ② 2. 一个静态方法上
 - ③ 3. 一个语句块上

示例代码: [ThreadDemo1.java](#)

```
public class ThreadDemo1{
    public static void main(String[] args){
        TestThread tt=new TestThread();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
    }
}
class TestThread implements Runnable{
    int tickets =100;
    String str=new String("hello");
    public void run(){
// String str=new String("hello"); //这样不可以, 因为str每次是一个新的对象所以要使用全局的str
        while(true){
            synchronized(str){
                if(tickets>0){
                    try{Thread.sleep(10);}
                    catch(Exception e){}
                    System.out.println(Thread.currentThread().getName()+"is sailing ticket"+tickets--);
                }
            }
        }
    }
}
```

运行结果

```
Thread-3 is sailing ticket24
Thread-4 is sailing ticket23
Thread-1 is sailing ticket22
Thread-2 is sailing ticket21
Thread-3 is sailing ticket20
Thread-4 is sailing ticket19
Thread-1 is sailing ticket18
Thread-2 is sailing ticket17
Thread-3 is sailing ticket16
Thread-4 is sailing ticket15
Thread-1 is sailing ticket14
Thread-2 is sailing ticket13
Thread-3 is sailing ticket12
Thread-4 is sailing ticket11
Thread-1 is sailing ticket10
Thread-2 is sailing ticket9
Thread-3 is sailing ticket8
Thread-4 is sailing ticket7
Thread-1 is sailing ticket6
Thread-2 is sailing ticket5
Thread-3 is sailing ticket4
Thread-4 is sailing ticket3
Thread-1 is sailing ticket2
Thread-2 is sailing ticket1
```

示例代码: [ThreadDemo2.java](#)

```
public class ThreadDemo2{
    public static void main(String[] args){
        TestThread tt=new TestThread();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
        new Thread(tt).start();
    }
}
class TestThread implements Runnable{
    int tickets =100;
    public void run(){
        while(true){
            sale();
        }
    }
    public synchronized void sale(){
        if(tickets>0){
            try{Thread.sleep(10);}
            catch(Exception e){}
            System.out.println(Thread.currentThread().getName()+"is sailing ticket"+tickets--);
        }
    }
}
```

运行结果

```
Thread-3is sailing ticket24
Thread-4is sailing ticket23
Thread-1is sailing ticket22
Thread-2is sailing ticket21
Thread-3is sailing ticket20
Thread-4is sailing ticket19
Thread-1is sailing ticket18
Thread-2is sailing ticket17
Thread-3is sailing ticket16
Thread-4is sailing ticket15
Thread-1is sailing ticket14
Thread-2is sailing ticket13
Thread-3is sailing ticket12
Thread-4is sailing ticket11
Thread-1is sailing ticket10
Thread-2is sailing ticket9
Thread-3is sailing ticket8
Thread-4is sailing ticket7
Thread-1is sailing ticket6
Thread-2is sailing ticket5
Thread-3is sailing ticket4
Thread-4is sailing ticket3
Thread-1is sailing ticket2
Thread-2is sailing ticket1
```

关键字synchronized的使用方法(1)

- `public synchronized void write();`
 - 由synchronized关键字锁定的是一个对象,而不是一个方法,
 - 一个线程调用某个对象的synchronized方法,就锁定了这个对象,直到它从这个方法中退出,才释放了这个锁定。因此,一旦一个对象被某个线程锁定了,它的所有synchronized方法就都不允许别的线程调用了。
 - 如果一个线程调用该方法,则方法体中访问的对象都被锁定,不允许其他线程访问,以解决共享资源的访问冲突问题。

关键字synchronized的使用方法(2)

- `public static synchronized int getValue();`
- 一个线程调用一个类的static synchronized方法, 就锁定了这个类对象, 也就控制了所有static synchronized方法。即类中所有的静态同步方法在每个时刻只有一个可以执行

关键字synchronized的使用方法(3)

- `synchronized (obj) { ... }`
- 一个线程进入由`synchronized (obj) { ... }`修饰的程序块, 就锁定了由`obj`指定的那个对象。

注意事项

- ① 一个对象在某一时刻只能被一个线程锁定；
- ② 一个类可以有多个对象, 不同的对象可以被不同的线程锁定
- ③ 一个线程可以锁定多个对象

误区

- 关键字synchronized可以作为Java方法修饰符，也可以作为JAVA方法内的语句。
- 被它修饰的代码部分往往被描述为临界区。这使很多人认为，由于代码被synchronized保护着，因此同一时刻只能有一个线程访问它。但这种观点是错误的。

- 对于Java类中的方法，关键字 `synchronized` 其实并不锁定该方法或该方法的部分代码，它只是锁定对象。
- `synchronized` 方法或 `synchronized` 区段内的代码在同一时刻下可有多个线程执行，只要是对不同的对象调用该方法就可以。

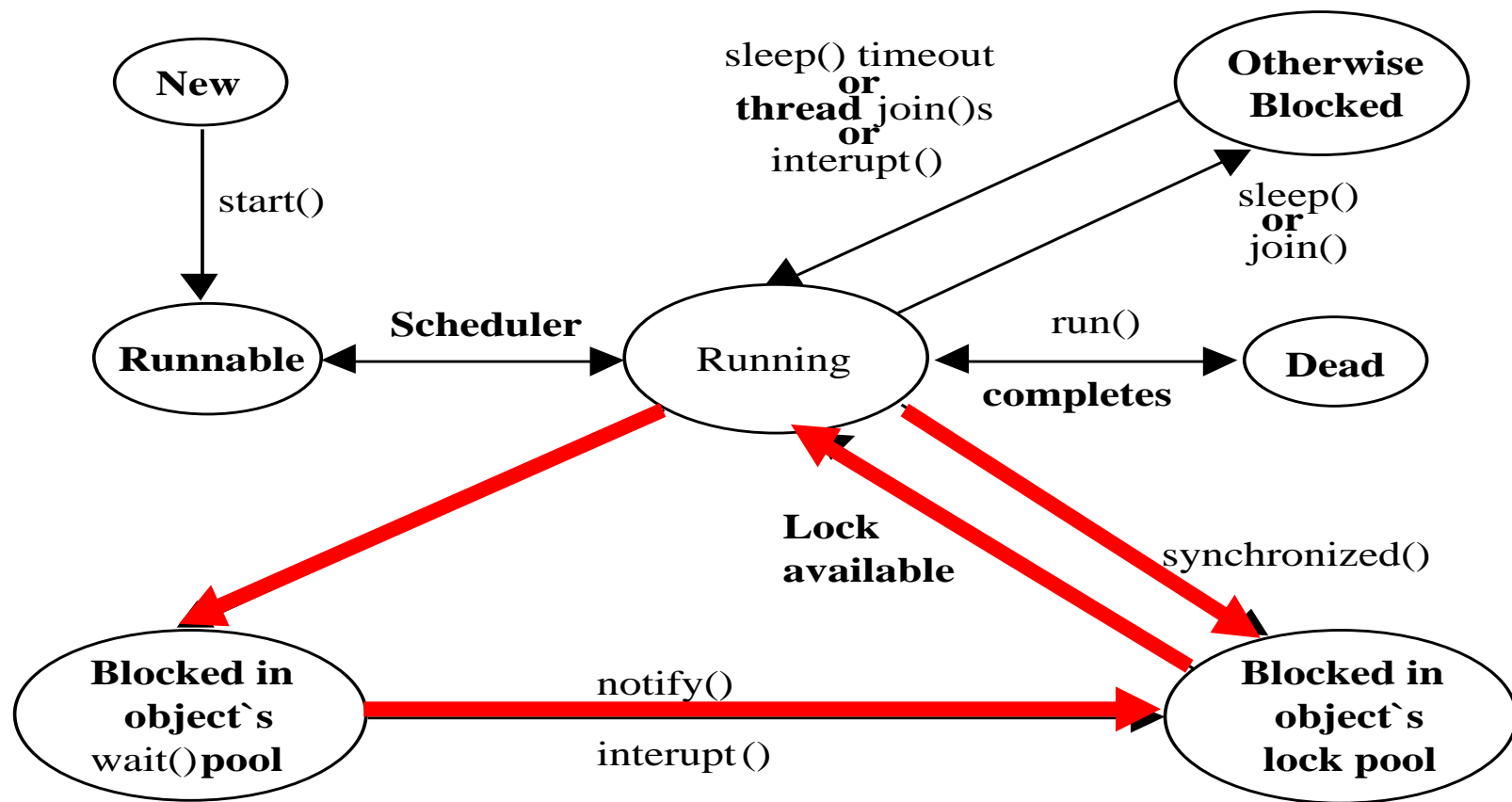
- 不同的线程执行不同的任务，如果这些任务有某种联系，线程之间必须能够通信，协调完成工作。
- 例如生产者和消费者共同操作堆栈，当堆栈为空时，消费者无法取出产品，应该先通知生产者向堆栈中加入产品。当堆栈已满时，生产者无法继续加入产品，应该先通知消费者从堆栈中取出产品。

☛ java.lang.Object 类中提供了两个用于线程通信的方法：

- wait()：执行该方法的线程释放对象的锁，Java 虚拟机把该线程放到该对象的等待池中。该线程等待其他线程将它唤醒。
- notify()：执行该方法的线程唤醒在对象的等待池中等待的一个线程。Java 虚拟机从对象的等待池中随机的选择一个线程，把它转到对象的锁池中。

线程状态

Thread states



线程状态说明

- 一般来说，每个共享对象的互斥锁存在两个队列，一个是锁等待队列，另一个是锁申请队列，锁申请队列中的第一个线程可以对该共享对象进行操作，而锁等待队列中的线程在某些情况下将移入到锁申请队列。

wait()、notify()和notifyAll()方法

- wait, notify, notifyAll 必须在已经持有锁的情况下执行, 所以它们只能出现在 synchronized 作用的范围内, 也就是出现在 synchronized 修饰的方法中。
- wait 的作用: 释放已持有的锁, 进入等待队列。
- notify 的作用: 随机唤醒 wait 队列中的一个线程并把它移入锁申请队列
- notifyAll 的作用: 唤醒 wait 队列中的所有线程并把它它们移入锁申请队列。

wait()、notify()和notifyAll()方法

- wait, notify和notifyAll都是与同步相关的方法, 只有在synchronized方法中才可以用
- 一个“在同步方法中等待”的线程自己是没有办法恢复运行的, 只能等另一个也进入了该对象的synchronized方法的线程调用notify和notifyAll后才有可能恢复运行

例如:

如果银行帐户里的金额比要转帐的金额少,就等待从别的地方存入足够的金额后再继续处理。

这种情况也被称为“在同步方法中等待”

```
public synchronized void get(){  
    ... ..  
    if(account.amount<amount){  
        ...  
        wait();  
    }  
    ... ..  
}
```

方法wait()与sleep() 方法对比

- ① 方法wait()与sleep() 方法一样，都能使线程等待而停止运行
- ② 其区别在于sleep()方法不会释放对象的锁，而wait()方法进入等待时，可以释放对象的锁，因而别的线程能对这些加锁的对象进行操作。所以，wait, notify和notifyAll都是与同步相关联的方法，只有在synchronized方法中才可以用。
- ③ 在不同步的方法或代码中则使用sleep()方法使线程暂时停止运行。

示例代码

```
class Account{
    String name;
    long id;
    private double sum,saveMoney,getMoney;
    public Account(long id,String name,double sum){
        this.id=id;
        this.name=name;
        this.sum=sum;
    }
    public synchronized double get(double d){
        while(sum<50000){
            try{
                this.wait();
            }
            catch(InterruptedException e){
            }
        }
        this.notify();
        getMoney=d;
        sum-=getMoney;
        return getMoney;
    }
}
```

示例代码

```
public synchronized double save(double d){
    while(sum>=125000){
        try{
            this.wait();
        }
        catch(InterruptedException e){

        }
    }
    this.notify();
    saveMoney=d;
    sum+=saveMoney;
    return saveMoney;
}

public double getSum(){
    return sum;
}

public String toString(){
    return "账号: "+id+": 姓名"+name+": 存款余额"+sum;
}

}
```

示例代码

```
class GetMoney implements Runnable{
    Account account1;
    public GetMoney(Account a){
        account1=a;
    }
    public void run(){
        for(int i=0;i<4;i++){
System.out.print("取钱: "+account1.get(50000)+"\t账户余额
sum="+account1.getSum()+"\n");
            try{
                Thread.sleep(500);
            }
            catch(InterruptedException e){}
        }
    }
}
```

示例代码

```
class SaveMoney implements Runnable{
    Account account1;
    public SaveMoney(Account a){
        account1=a;
    }
    public void run(){
        for(int i=0;i<8;i++){
System.out.print("存钱: "+account1.save(25000)+"\t账户余额 : sum="+account1.getSum()+"\n");
            try{
                Thread.sleep(50);
            }
            catch(InterruptedException e){

            }
        }
    }
}
```

案例：对同一个账户存钱，取钱并行

```
public class Test{  
    public static void main(String[] args){  
        //建立一个账户，存款为0  
        Account ac=new Account(123456789,"Tom",0.0);  
        System.out.println(ac.toString()+"：存取款操作如下");  
        Thread p=new Thread(new GetMoney(ac));  
        Thread c=new Thread(new SaveMoney(ac));  
        p.start();  
        c.start();  
    }  
}
```


账号: 123456789: 姓名Tom: 存款余额0.0: 存取款操作如下

存钱: 25000.0	账户余额: sum=25000.0
存钱: 25000.0	账户余额: sum=0.0
取钱: 50000.0	账户余额: sum=0.0
存钱: 25000.0	账户余额: sum=25000.0
存钱: 25000.0	账户余额: sum=50000.0
存钱: 25000.0	账户余额: sum=75000.0
存钱: 25000.0	账户余额: sum=100000.0
存钱: 25000.0	账户余额: sum=125000.0
取钱: 50000.0	账户余额: sum=75000.0
存钱: 25000.0	账户余额: sum=100000.0
存钱: 25000.0	账户余额: sum=125000.0
取钱: 50000.0	账户余额: sum=100000.0
存钱: 25000.0	账户余额: sum=100000.0
存钱: 25000.0	账户余额: sum=125000.0
取钱: 50000.0	账户余额: sum=100000.0

案例：PV操作

- 多个生产者生产产品，多个消费者消费产品，产品要提供一个含有标识的id属性，另外需要在生产或消费时打印产品的详细内容；
- 店员一次最多只能持有10份产品，如果生产者生产的产品多于10份，则会让生产者线程等待；
- 当店员没有产品消费时，消费线程等待；

- 多个生产者线程和多个消费者线程都需要面向店员，生产者线程生产的产品交给店员消费，消费者线程从店员那里获得产品消费；
- 店员在操作产品时，需要做到互斥；
- 产品类：Products
- 店员类：Clerk
- 生产者线程：Producer
- 消费者线程：Consumer

```
public class Products {  
    int id;  
    public Products() {  
        super();  
    }  
    public Products(int id) {  
        super();  
        this.id = id;  
    }  
    public String toString() {  
        return "Products [id=" + id + "]";  
    }  
}
```

```
public class Clerk {  
    int index = 0; // 默认为0个产品  
    Products[] pro = new Products[10];  
    // 生产者生产出来的产品交给店员  
    public synchronized void addProduct(Products pd) {  
        while (index == pro.length) {  
            try {  
                this.wait(); // 产品已满, 请稍后再生产  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        this.notify(); // 通知等待区的消费者可以取产品了  
        pro[index] = pd;  
        index++;  
    }  
}
```

// 消费者从店员处取产品

```
public synchronized Products getProduct() {  
    while (index == 0) {  
        try {  
            this.wait(); // 缺货, 请稍后再取。  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
    this.notify(); // 通知等待区的生产者可以生产产品了  
    index--;  
    return pro[index];  
}
```

```

public class Producer implements Runnable {
    private Clerk clerk;
    public Producer() {
        super();
    }
    public Producer(Clerk clerk) {
        super();
        this.clerk = clerk;
    }
    public void run() {
        System.out.println("生产者开始生产产品");
        for (int i = 0; i < 15; i++) {
            // 注意此处的循环次数一定要大于pro数组的长度 (10)
            Products pd = new Products(i);
            clerk.addProduct(pd); // 生产产品
            System.out.println("生产了: " + pd);
            try { // 睡眠时间用随机产生的值来设置
                Thread.sleep((int) (Math.random() * 10) * 100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```
public class Consumer implements Runnable {
    private Clerk clerk;
    public Consumer() {
        super();
    }
    public Consumer(Clerk clerk) {
        super();
        this.clerk = clerk;
    }
    public void run() {
        System.out.println("消费者开始取走产品");
        for (int i = 0; i < 15; i++) {
            // 注意此处的循环次数一定要大于pro数组的长度(10)
            // 取产品
            Products pd = clerk.getProduct();
            System.out.println("消费了: " + pd);
            try { // 睡眠时间用随机产生的值来设置
                Thread.sleep((int) (Math.random() * 10) * 100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```




```
public class ProducerAndConsumerTest {  
    public static void main(String[] args) {  
        Clerk clerk = new Clerk();  
        Thread producerThread = new Thread(new Producer(clerk));  
        Thread consumerThread = new Thread(new Consumer(clerk));  
        Thread producerThread1 = new Thread(new Producer(clerk));  
        Thread consumerThread2 = new Thread(new Consumer(clerk));  
        producerThread.start();  
        consumerThread.start();  
        producerThread1.start();  
        consumerThread2.start();  
    }  
}
```

```
消费了: Products [id=4]
生产了: Products [id=3]
消费了: Products [id=3]
消费了: Products [id=2]
消费了: Products [id=1]
生产了: Products [id=5]
消费了: Products [id=5]
生产了: Products [id=4]
消费了: Products [id=4]
```

- 在数据结构上（stack）实现生产者和消费者问题
- PV.java

思考题：

- 在数据结构上（stack）实现生产者和消费者问题，生产者生产计算机放入Stack中，消费者从Stack中取出计算机消费。
- 计算机可以定制等
- ○ ○ ○

小结

- 线程之间交换信息称为线程通信，可以认为，wait()和notify()方法是java采用的一种简单的线程间通信机制，利用它们，彼此间只传送一个信号。
- 线程间要传送的数据较多时，必须使用其它的像共享主存、管道流等通信方式。
 - ① 主存读/写通信:Java输入输出流
 - ② 管道流通信:

- 管道用来把一个程序输出连接到另一程序输入。java.io中提供了类PipedInputStream和PipedOutputStream作为管道的输入/输出部件。
 - 构造方法中，对于管道输入/输出流来说，对应管道输出/输入流作为参数以进行连接；
 - 管道输入/输出流还提供了方法connect()以进行相应的连接。

- [TestPipedStream.java](#)

死锁 (1)

- ✓ 在多线程竞争使用多资源的程序中，有可能出现死锁的情况。死锁不是由于资源短缺造成的，而是由于程序设计不合理而造成程序中所有的线程都陷入等待态或阻塞态。
- ✓ 这种情况发生在一个线程等待另一个线程所持有的锁，而那个线程又在等待第一个线程持有的锁的时候。

死锁 (2)

- ✓ 每个线程都不能继续运行，除非另一线程运行完同步程序块。
- ✓ 因为哪个线程都不能继续运行，所以哪个线程都无法运行完同步程序块。
- ✓ 另一种情况是在wait, notify结构中, 如果所有的线程都wait了, 就不会有线程来notify它们, 因此要特别注意;

后台线程

- 后台线程是指为其他线程提供服务的线程，也称为守护线程（**Daemon Threads**）。
- 守护线程是一类特殊的线程，它和普通线程的区别在于它并不是应用程序的核心部分，当一个应用程序的所有非守护线程终止运行时，**即使仍然有守护线程在运行，应用程序也将终止。**
- **反之，只要有一个非守护线程在运行，应用程序就不会终止。**
- 主线程默认情况下是前台线程，由前台线程创建的线程默认情况下也是前台线程。



守护线程(Daemon Threads)

- 可以通过调用方法 `isDaemon()` 来判断一个线程是否是守护线程
- 程，也可以调用方法 `setDaemon()` 来将一个线程设为守护线程。
 - *Garbage collector is a daemon thread*

守护线程(Daemon Threads)

```
public class Machine extends Thread {  
    private int a;  
    private static int count;  
    public void start() {  
        super.start();  
  
        // 匿名线程类  
        Thread daemon = new Thread() {  
            public void run() {  
                while (true) { // 无限循环  
                    reset();  
                    try {  
                        sleep(50);  
                    } catch (InterruptedException e) {  
                        throw new RuntimeException(e);  
                    }  
                }  
            }  
        };  
        daemon.setDaemon(true);  
        daemon.start();  
    }  
}
```

```
public void reset() {  
    a = 0;  
}  
public void run() {  
    while (true) {  
        System.out.println(getName() + ":" + a++);  
        if (count++ == 100)  
            break;  
        yield();  
    }  
}  
public static void main(String args[]) throws Exception {  
    Machine machine = new Machine();  
    machine.start();  
}  
}
```

定时器Timer

- 在JDK的java.util包中提供了一个实用类Timer，它能够定时执行特定的任务。
- TimerTask类表示定时器执行的一项任务。
- Timer类的`schedule(TimerTask task, long delay, long period)`方法用来设置定时器需要定时执行的任务。task参数表示任务，delay参数表示延迟执行的时间，以毫秒为单位，period参数表示每次执行任务的间隔时间，以毫秒为单位。



定时器

- `timer.schedule(task,10,50);`
- 以上代码表明定时器将在10毫秒以后开始执行task任务（即执行TimerTask实例的run()方法），以后每隔50毫秒重复执行一次task任务。



定时器

```
import java.util.Timer;
import java.util.TimerTask;
public class Machine1 extends Thread {
    private int a;
    public void start() {
        super.start();
        // 把与Timer关联的线程设为后台线程
        Timer timer = new Timer(true);
        // 匿名类
        TimerTask task = new TimerTask() {
            public void run() {
                reset();
            }
        };
        // 设置定时任务
        timer.schedule(task, 10, 50);
    }
}
```




```
public void reset() {  
    a = 0;  
}  
  
public void run() {  
    for (int i = 0; i < 1000; i++) {  
        System.out.println(getName() + ":" + a++);  
        yield();  
    }  
}  
  
public static void main(String args[]) throws Exception {  
    Machine1 machine = new Machine1();  
    machine.start();  
}  
}
```

思考：多线程问题

- 对多线程程序本身来说，它会对系统产生以下影响：
 - 线程需要占用内存。
 - 线程过多，会消耗大量CPU时间来跟踪线程。
 - 必须考虑多线程同时访问共享资源的问题，如果没有协调好，就会产生令人意想不到的问题，例如可怕的死锁和资源竞争。
 - 因为同一个任务的所有线程都共享相同的地址空间，并共享任务的全局变量，所以程序也必须考虑多线程同时访问全局变量的问题。

小结 (1)

- 可以采用 Thread 类的子类或实现 Runnable 接口，实现多线程
- 两个关键性操作：
 - 定义用户线程的操作，即定义用户线程的run()方法；
 - 在适当的时候建立用户线程的实例。

- 继承 Thread 类

- 用户创建自己的 Thread 类的子类，并在子类中重新定义自己的 run() 方法，run() 方法中包含了用户线程的操作。
- 在需要建立自己的线程时，只需创建一个已定义好的 Thread 子类对象就可以了。

小结 (3)

- 实现 Runnable 接口
 - 编写实现 Runnable 接口的线程类，重写 Runnable 接口中的 run() 方法。
 - 在实现 Runnable 接口的同时还可以继承其它类。

小结（4）

- Java中用关键字 **synchronized** 为共享资源加锁，在任何一个时刻只能有一个线程能取得加锁对象的钥匙，对加锁对象进行操作。从而达到了对共享资源访问时的保护。
- synchronized关键字可以使用在：
 1. 一个成员方法上
 2. 一个静态方法上
 3. 一个语句块上

小结 (5)

- 线程间传递信号：
 - wait, notify和notifyAll都是与同步相关联的方法, 只有在synchronized方法中才可以用
- 主存读/写通信
- 管道流通信