线程

孙聪

网络与信息安全学院

2019-11-11

课程内容

- Java概述
- 面向对象程序设计概念
- Java语言基础
- Java面向对象特性
- Java高级特征
- 容器类
- 常用预定义类
- 异常处理
- 输入输出
- 线程

提要

提要

并发的目的和手段

- 并发的目的
  - 提高系统效率
  - 简化程序设计

并发的目的和手段

- 并发的目的
  - 提高系统效率
  - 简化程序设计
- 多线程是实现并发的一种有效手段
  - 多进程并发（多任务操作系统中）
  - 多线程并发：一个进程可以通过运行多个线程来并发地执行多项任务

线程 vs. 进程

- 进程：内核级的实体
  - 包含代码、数据、堆，PCB（进程管理、内存管理、文件管理信息）等
  - 进程结构存在于内核空间，用户程序须通过系统调用进行访问或改变
- 线程：用户级的实体
  - 线程结构驻留在用户空间，能够被普通的用户级函数组成的线程库直接访问
  - 线程独有：寄存器（栈指针，程序计数器）、栈等
  - 一个进程中的所有线程共享该进程的状态

- Java语言的重要特征是在语言级支持多线程的程序设计

什么是线程（程序设计角度）

- 线程：进程中的单个顺序执行流
  - “顺序”指逻辑上的“顺序”，即同一时刻仅执行一条语句，不是语法上的“顺序”，与多种程序流控制（分支、循环等）不矛盾
- 多线程：进程中包含多个顺序执行流

Java线程模型

- 线程模型：CPU、代码和数据的封装体
  - CPU：线程在占用CPU时的CPU状态
  - 代码：CPU所执行的代表线程行为的代码（可由多个线程共享）
  - 数据：CPU执行代码过程中操作的数据，包括线程独有数据（程序计数器、栈等）和共享数据（如堆上的对象）
- 代码与数据相互独立
- 代码和部分数据可多线程共享
- 代码 + 数据 = 线程体（决定线程的行为）

提要

Thread类

- 线程模型由java. lang. Thread类进行定义和描述
- 程序中的线程都是Thread类或其子类的实例

Thread类

- 线程模型由java.lang.Thread类进行定义和描述
- 程序中的线程都是Thread类或其子类的实例
- Thread类的构造方法
  - public Thread(ThreadGroup group, Runnable target, String name);
  - public Thread();
  - public Thread(Runnable target);
  - public Thread(ThreadGroup group, Runnable target);
  - public Thread(String name);
  - public Thread(ThreadGroup group, String name);
  - public Thread(Runnable target, String name);
  - 核心参数：
    - group：指明该线程所处的线程组（不建议使用）
    - name：线程名称
    - target：提供线程体的对象（由Runnable接口变量引用的实现了Runnable接口的类的对象）

线程的创建方法（一）

- 实现Runnable接口
  - 定义一个类实现Runnable接口，提供run()方法的实现
  - 将该类的一个实例作为参数传递给Thread构造方法，该实例对象提供线程体（代码：run()方法的实现；数据：对象状态）

```
public class CountDown implements Runnable{
    private static int idcnt=1;
    private final int threadid=idcnt++;
    int counter=3;
    public void run(){          //线程从Runnable实例的run()方法开始执行
        while(counter>=0){
            try{
                Thread.sleep(1000);
            } catch (Exception e){ e.printStackTrace(); }
            System.out.println("#" +threadid+(counter>0? "->" +counter: "->run!" ));
            counter--;
        }
    }
    public static void main(String[] args){
        Thread t1=new Thread(new CountDown());  //线程是Thread类的实例
        Thread t2=new Thread(new CountDown());
        t1.start();
        t2.start();
        System.out.println("waiting for run...");
    }
}
```

线程的创建方法（二）

- Thread类本身也实现了Runnable接口，因而可通过重写Thread类的子类中的run()方法定义线程行为，并通过创建Thread子类的实例创建线程
- 继承Thread类
  - 从Thread类派生子类，并重写其中的run()方法，以定义线程行为
  - 创建该子类的对象即创建线程

```
public class CountDown2 extends Thread{
    private static int idcnt=1;
    private final int threadid=idcnt++;
    int counter=3;
    public void run(){
        while(counter>=0){
            try{
                Thread.sleep(1000);
            } catch (Exception e){ e.printStackTrace(); }
            System.out.println("#" +threadid+(counter>0? "->" +counter: "->run!" ));
            counter--;
        }
    }
    public static void main(String[] args){
        CountDown2 t1=new CountDown2();
        Thread t2=new CountDown2();
        t1.start();
        t2.start();
        System.out.println("waiting for run...");
    }
}
```

两种线程创建方法比较

- 实现Runnable接口的优点：便于用extends继承其它类
- 继承Thread类的优点：程序代码更简单

```
public class CountDown3{ static int idcnt=1;
    public static void main(String[] args){
        new SubThread().start();
        new Thread(new Run()).start();
    }
}
class SubThread extends Thread{
    private final int threadid=CountDown3.idcnt++;
    public void run(){
        try{ Thread.sleep(1000);
            System.out.println("#" +threadid);
        } catch (Exception e){ e.printStackTrace(); }
    }
}
class Parent{
    public void doSth(){ System.out.println("do something ..."); }
}
class Run extends Parent implements Runnable{
    private final int threadid=CountDown3.idcnt++;
    public void run(){
        try{ Thread.sleep(1000);
            System.out.print("#" +threadid+ ": ");
            doSth();
        } catch (Exception e){ e.printStackTrace(); }
    }
```

线程的运行

- 新创建的线程不会自动运行，必须调用线程的start()方法
- 该方法的调用把线程的CPU状态置为可运行（Runnable）状态
- Runnable状态意味着该线程可以参加调度，被JVM调度运行，但并不意味着线程一定会立即运行

习题10-1

实现了Runnable接口的类是否可以直接使用而不用于构造Thread对象？

java.util.concurrent.Executor接口

- J2SE 5.0引入的执行器接口
  - 帮助用户管理线程对象，简化编程开发
- 将主线程与所有任务线程分开，帮助主线程管理异步任务的执行
  - 主线程不再需要显式地管理任务线程的生命周期

java.util.concurrent.Executor接口

- 实际中使用Executor的子接口ExecutorService
  - 二者均包含接口方法： void execute(Runnable command)
  - ExecutorService接口还包含接口方法：void shutdown()
- 具体步骤
  - 调用Executors类的newCachedThreadPool()方法，得到一个线程池对象（该对象实现了ExecutorService接口）
  - 使用线程池对象调用execute()接口方法，参数传入Runnable对象，该方法将参数Runnable对象封装为线程执行
  - 使用线程池对象调用shutdown()接口方法：
    禁止向这个Executor提交新的任务（不是用于结束所有线程，整个程序会在Executor的所有任务都执行完后尽快退出）

```
public class CountDown4 implements Runnable{
    private static int idcnt=1;
    private final int threadid=idcnt++;
    int counter=3;
    public void run(){
        while(counter>=0){
            try{
                Thread.sleep(1000);
            } catch (Exception e){ e.printStackTrace(); }
            System.out.println( “#” +threadid+(counter>0? “->” +counter: “->run!” ));
            counter--;
        }
    }
    public static void main(String[] args){
        ExecutorService exec=Executors.newCachedThreadPool();
        exec.execute(new CountDown4());
        exec.execute(new CountDown4());
        exec.shutdown();
        System.out.println( “waiting for run...” );
    }
}
```

提要

线程优先级

- Thread类有三个有关线程优先级的静态常量
  - MIN_PRIORITY：最低优先级（1）
  - MAX_PRIORITY：最高优先级（10）
  - NORM_PRIORITY：普通优先级（5）
- MIN_PRIORITY≤线程优先级≤MAX_PRIORITY，数值越大优先级越高
- 线程有缺省优先级，主线程缺省为NORM_PRIORITY
- 子线程继承其父线程（创建子线程的语句执行时所在的线程）的优先级
- 获得/设定线程优先级
  - Thread类的getPriority() / setPriority()方法
- 注意：
  - 用户尽可能不要改变线程的优先级
  - 如需改变优先级，则一般在run()方法开始处进行设置
  - Java线程优先级与OS线程优先级难以映射，因此可移植的方式是仅使用MIN_PRIORITY，MAX_PRIORITY和NORM_PRIORITY设置优先级

线程调度策略

- 线程调度：在单个CPU上以某种顺序运行多个线程
- Java的线程调度策略
  - 规范：基于优先级的抢先式调度
  - 实际：平台相关，JVM相关

线程调度策略

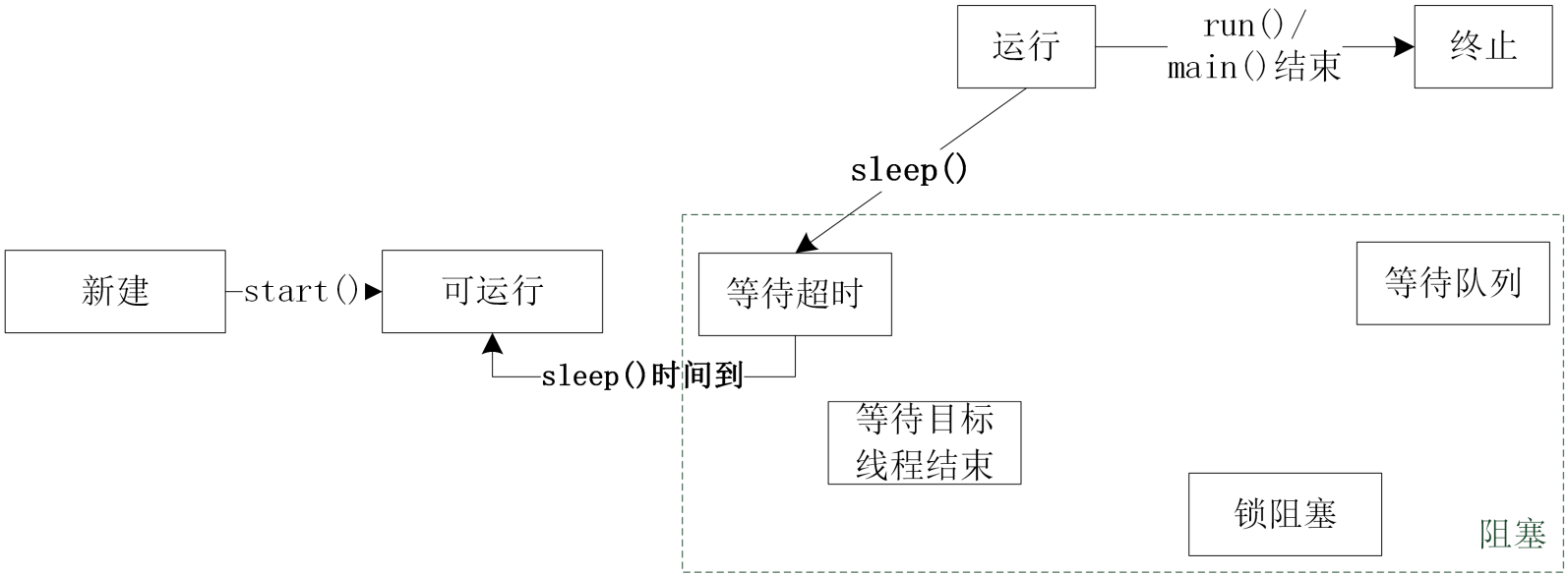
- 线程调度：在单个CPU上以某种顺序运行多个线程
- Java的线程调度策略
  - 规范：基于优先级的抢先式调度
  - 实际：平台相关，JVM相关

- 基于优先级的抢先式调度
  - 允许多个线程可运行，但只有一个线程在运行
  - 根据线程优先级，选择高优先级的可运行线程
  - 当前线程持续运行，直到以下两种情况之一
    - 该线程自行进入非可运行状态（如执行Thread.sleep()调用，或等待访问共享的资源）
    - 出现更高优先级线程成为可运行（这时CPU被高优先级线程抢占运行）
  - 同一优先级的多个可运行线程可分时轮流运行，也可逐个运行，由JVM而定

线程的基本控制

- static Thread currentThread()
  - Thread类的静态方法，返回对当前正在执行的线程的引用
- boolean isAlive()
  - 线程状态未知时，用以确定线程是否活着
  - 返回true：线程已启动，且尚未运行结束
- static void sleep(...)
  - 使当前线程阻塞（睡眠）一段固定的时间。在线程睡眠时间内，将运行其他线程
  - sleep()结束后，线程从阻塞状态进入Runnable状态
  - sleep()可能抛出InterrputedException

线程的基本控制

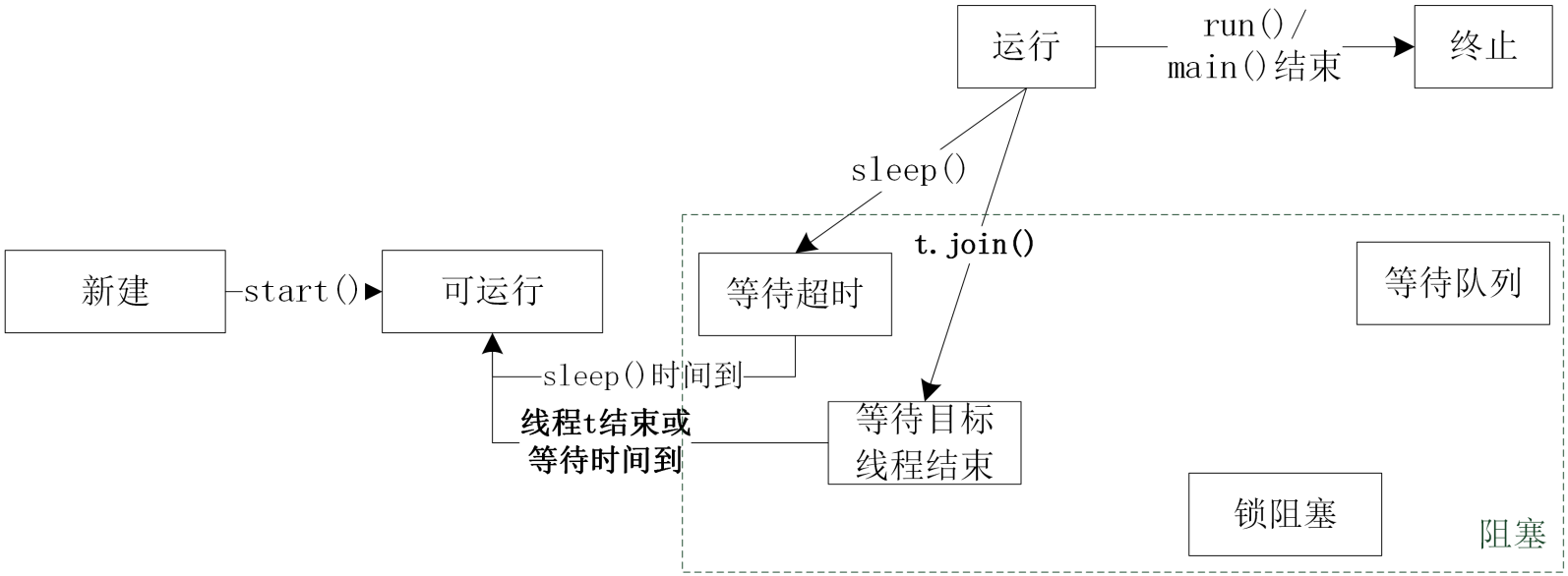

运行
run()/
main()结束
终止
sleep()
新建
start()
可运行
等待超时
等待队列
sleep()时间到
等待目标
线程结束
锁阻塞
阻塞

线程的基本控制：join()

在当前线程中执行t.join()方法使当前线程等待，直到线程t结束为止，线程恢复到Runnable状态

```
public void doTask(){
    TimerThread  t = new TimerThread(100);
    t.start();
    ...
    // 与线程t并发运行
    ...
    try{
        t.join();    // 在这里等待线程t执行结束
        ...          // 当前线程继续执行
    } catch( InterruptedException e) {
        e.printStackTrace();
    }
    ...
}
```

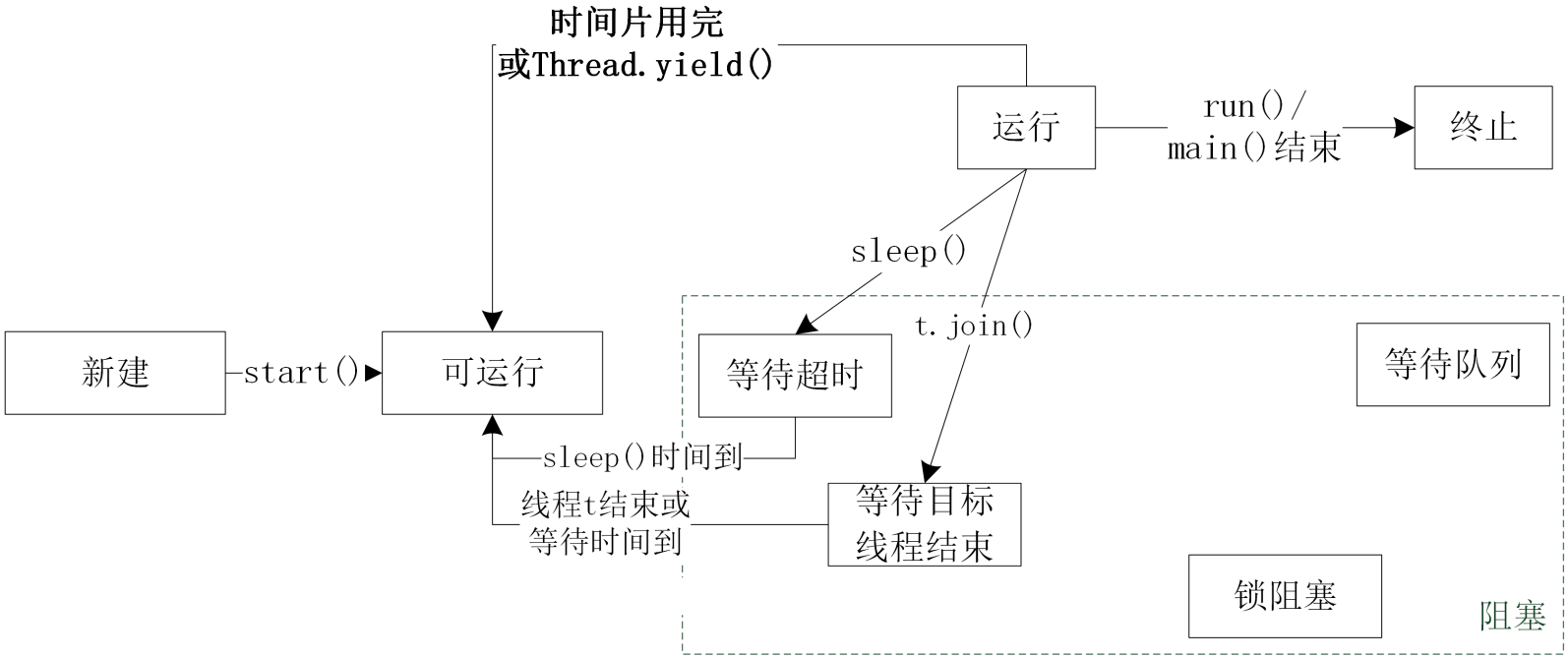
线程的基本控制：join()



```
class ToJoin extends Thread{
    public ToJoin(String nm){ super(nm); }
    public void run(){
        try{ Thread.sleep(2000); } catch (InterruptedException e){ }
        System.out.println(Thread.currentThread().getName()+" awake!");
    }
}
class Joiner implements Runnable{
    private ToJoin tojoin;
    public Joiner(ToJoin t){ this.tojoin=t; }
    public void run(){
        try{ this.tojoin.join(); } catch (InterruptedException e){ }
        System.out.println(this.tojoin.getName()+" join finished");
    }
}
public class TestJoin{
    public static void main(String[] args){
        ToJoin t1=new ToJoin("t1");
        t1.start();
        new Thread(new Joiner(t1)).start();
    }
}
```

线程的基本控制：yield()

- 调用该方法告诉调度器当前线程愿意将CPU让给具有相同优先级的线程



线程的基本控制：结束线程

- 线程完成运行并结束后，将不能再运行
- 线程除正常运行结束（run()方法执行完成）外，还可用其他方法控制使其停止
  - stop()：强行终止线程，易造成线程不一致
  - 使用标志flag：通过设置flag 指明run()方法应该结束

```
public class TestStop {
    public static void main(String[] args){
        Tick t=new Tick();
        new Thread(t).start();
        try{
            Thread.sleep(3000);
        } catch (Exception e){}
        System.out.println("quiting Task ...");
        t.stopRunning();
    }
}
class Tick implements Runnable {
    private boolean timeToQuit = false;
    public void stopRunning() { timeToQuit = true; }
    public void run( ){
        while(!timeToQuit){
            try {
                Thread.sleep(1000);
                System.out.println("tick ...");
            } catch (Exception e) {}
        }
        System.out.println("Tick finished.");
    }
}
```

线程的基本控制

- 以下控制方法已废止，会导致死锁
  - stop()
  - suspend()
  - resume()

提要

线程并发中的问题

- 多个线程相对执行的顺序不确定，导致执行结果不确定
- 多个线程操作共享数据时，执行结果不确定导致共享数据的一致性被破坏
- 对共享数据操作的并发控制机制称为线程同步

线程并发中的问题

一个堆栈类

```
public class MyStack{
    private int idx = 0;
    private char[] data = new char[6];
    public void push( char c ){
        data[idx] = c;
        idx ++;    // 栈顶指针指向下一个空单元
    }
    public char pop( ){
        idx--;
        return data[idx];
    }
}
```

线程a（压栈）	线程b（出栈）	堆栈s状态

线程a（压栈）	线程b（出栈）	堆栈s状态
		|p|q| | | | |, idx=2

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2

时刻t1	//调用s.push('r');
	data[idx]='r';
			|p|q|r| | | |, idx=2

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2

时刻t1	//调用s.push('r');
	data[idx]='r';
	/*线程a被抢占，idx++;
	未执行*/
			|p|q|r| | | |, idx=2

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2

时刻t1	//调用s.push('r');
	data[idx]='r';
	/*线程a被抢占，idx++;
	未执行*/
			|p|q|r| | | |, idx=2

时刻t2		//调用s.pop() idx−−; return data[idx];	|p|q|r| | | |, idx=1

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2

时刻t1	//调用s.push('r');
	data[idx]='r';
	/*线程a被抢占，idx++;
	未执行*/
			|p|q|r| | | |, idx=2

时刻t2		//调用s.pop() idx——; return data[idx];	|p|q|r| | | |, idx=1
时刻t3	//恢复运行 idx++;		|p|q|r| | | |, idx=2

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2

时刻t1	//调用s.push('r');
	data[idx]='r';
	/*线程a被抢占，idx++;
	未执行*/
			|p|q|r| | | |, idx=2

时刻t2		//调用s.pop() idx——; return data[idx];	|p|q|r| | | |, idx=1
时刻t3	//恢复运行 idx++;		|p|q|r| | | |, idx=2

- 线程a压入的数据'r'丢失。t4时刻pop()会怎样？

对象锁与锁语句

- 对象锁：Java中每个标记为synchronized的对象都含有的一个排他锁
- synchronized：对共享数据的排他操作的标记方法
  - synchronized标记在特定的代码片段（方法）上，这样的代码片段又称临界区
  - 共享数据包含在对象中，对象锁与对象一一对应，对象的所有synchronized方法（代码片段）共享对象锁
- 排他
  - 线程获得对象锁后才能执行临界区代码片段（即拥有对该对象的操作权）
  - 这时，其他任何线程无法获得对象锁，也无法执行临界区代码

MyStack类的改进

栈的push(), pop()操作都采用这种线程同步机制

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

多线程对共享数据进行操作

线程a（压栈）	线程b（出栈）	堆栈s状态

多线程对共享数据进行操作

线程a（压栈）	线程b（出栈）	堆栈s状态
		|p|q| | | | |, idx=2

多线程对共享数据进行操作

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2
时刻t1	/*调用s.push('r'), 获得s的对象锁后运行*/ data[idx]='r';		|p|q|r| | | |, idx=2

多线程对共享数据进行操作

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2
时刻t1	/*调用s.push('r'), 获得s的对象锁后运行*/ data[idx]='r'; /*线程a被抢占，idx++; 未执行*/		|p|q|r| | | |, idx=2

多线程对共享数据进行操作

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2
时刻t1	/*调用s.push('r'), 获得s的对象锁后运行*/ data[idx]='r'; /*线程a被抢占，idx++; 未执行*/		|p|q|r| | | |, idx=2
时刻t2		/*调用s.pop(), 未获得s的对象锁, b到s的lock pool中 等待s的对象锁*/	|p|q|r| | | |, idx=2

多线程对共享数据进行操作

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2
时刻t1	/*调用s.push('r'), 获得s的对象锁后运行*/ data[idx]='r'; /*线程a被抢占，idx++; 未执行*/		|p|q|r| | | |, idx=2
时刻t2		/*调用s.pop(), 未获得s的对象锁, b到s的lock pool中 等待s的对象锁*/	|p|q|r| | | |, idx=2

时刻t3	//恢复运行
	idx++;
	/*完成push()并
	交回s的对象锁*/
			|p|q|r| | | |, idx=3

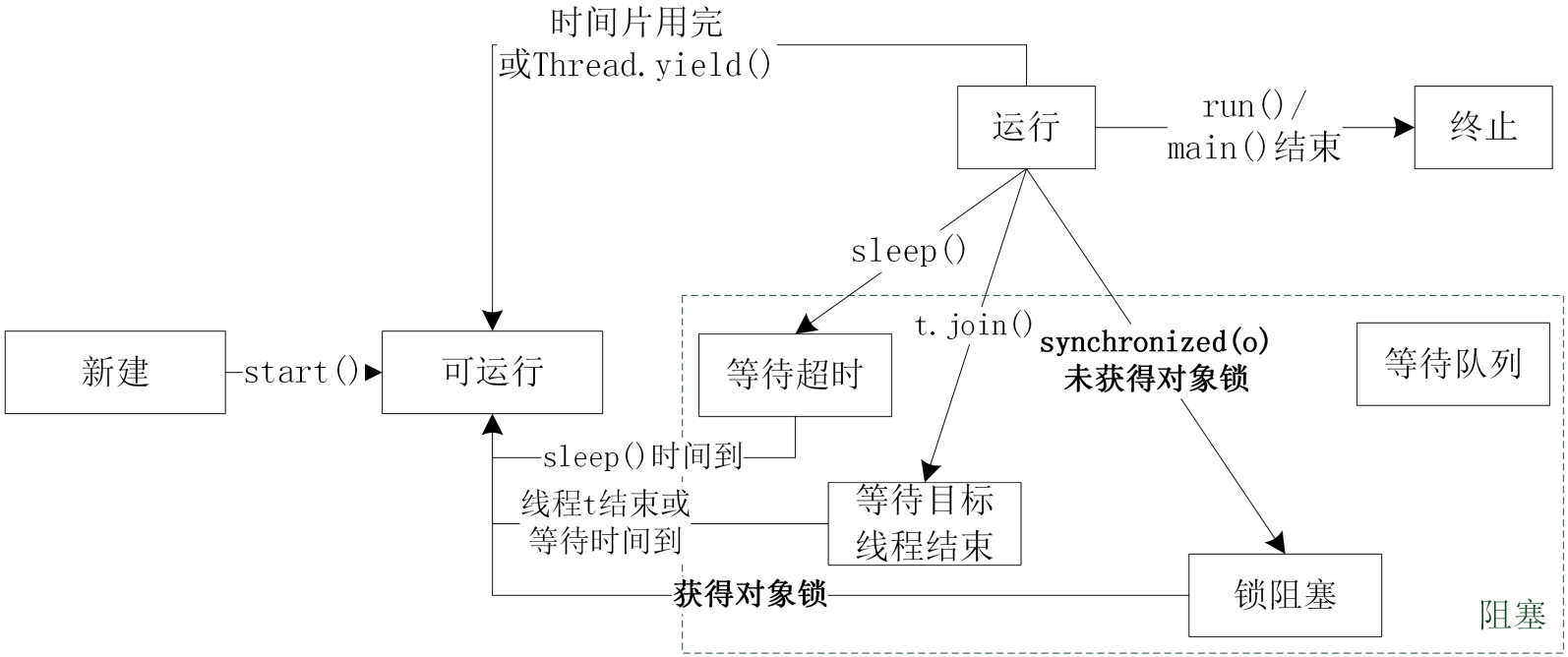
多线程对共享数据进行操作

	线程a（压栈）	线程b（出栈）	堆栈s状态
			|p|q| | | | |, idx=2
时刻t1	/*调用s.push('r'), 获得s的对象锁后运行*/ data[idx]='r'; /*线程a被抢占，idx++; 未执行*/		|p|q|r| | | |, idx=2
时刻t2		/*调用s.pop(), 未获得s的对象锁, b到s的lock pool中 等待s的对象锁*/	|p|q|r| | | |, idx=2

时刻t3	//恢复运行
	idx++;
	/*完成push()并
	交回s的对象锁*/
			|p|q|r| | | |, idx=3

时刻t4		/*运行pop(), 返回'r'*/	|p|q|r| | | |, idx=2

线程进入锁等待队列


时间片用完
或Thread.yield()
运行
run()/
main()结束
终止
sleep()
t.join()
synchronized(o)
未获得对象锁
新建
start()
可运行
等待超时
等待队列
sleep()时间到
线程t结束或
等待时间到
等待目标
线程结束
获得对象锁
锁阻塞
阻塞

几点说明

- 如果一个方法的整体都在synchronized块中，则可将synchronized关键字置于方法声明中，即：

```
public synchronized void mtd() {/*方法体*/ }
```

等价于

```
public void mtd(){ synchronized(this){ /*方法体*/ }}
```

- 考虑，以上两种方式哪种效率更高？

几点说明

- 对象锁具有可重入性：允许已拥有某对象锁的线程再次请求并获得该对象锁

```
public class GetLockAgain implements Runnable{
    public synchronized void mtd(){
        System.out.println("Entered Critical Section in mtd()");
        mtd2();
    }
    public synchronized void mtd2(){
        System.out.println("Entered Critical Section in mtd2()");
    }
    public void run(){
        mtd();
    }
    public static void main(String[] args){
        new Thread(new GetLockAgain()).start();
    }
}
```

几点说明

- 对共享资源加锁，是通过限制代码对共享资源的访问方式来实现的——不要留给用户绕过临界区代码访问共享资源的机会
  - 将作为成员变量的共享数据设为private的
  - 对共享数据的所有访问都必须使用synchronized
- 何时返还对象锁？
  - synchronized语句块执行完毕后
  - 在synchronized语句块中出现exception时
  - 持有锁的线程调用该对象的wait()方法，将该线程放入对象的wait pool中，等待某事件的发生

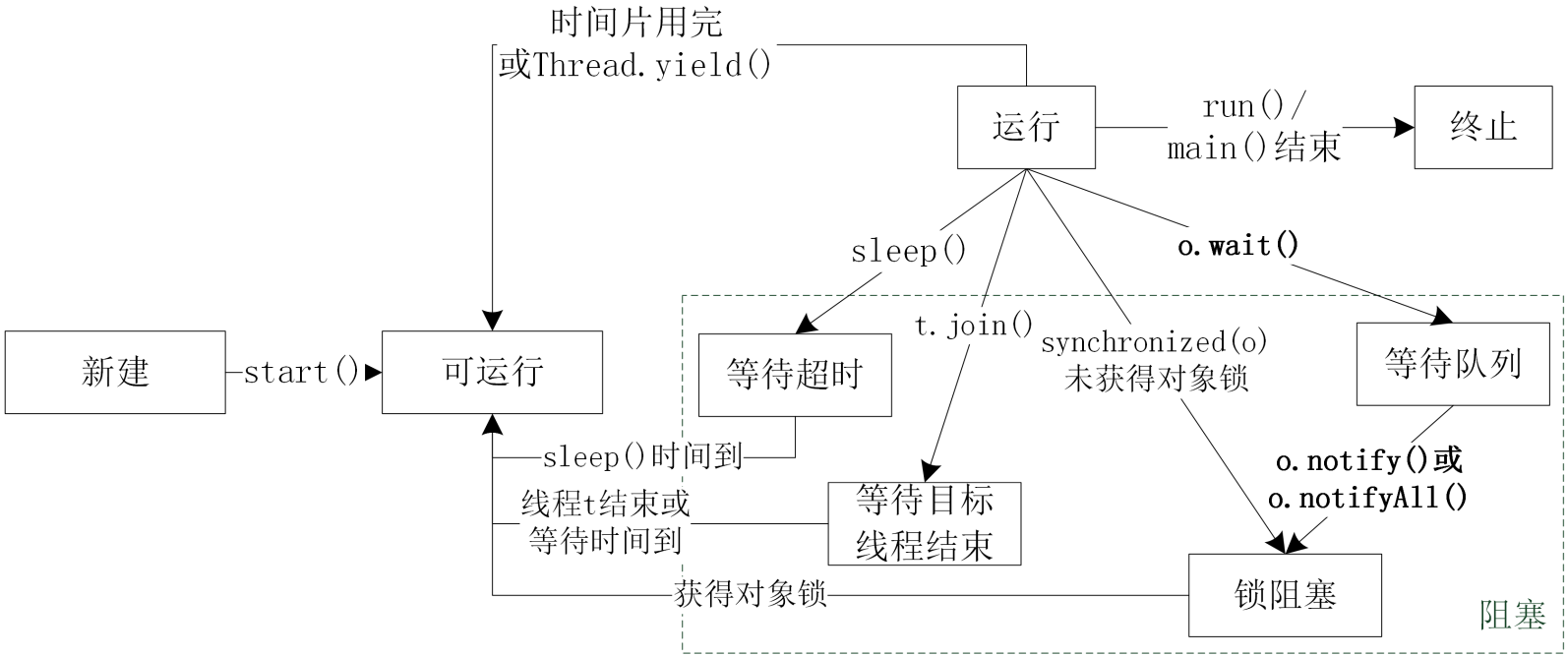
避免死锁

- 死锁是指两个线程同时等待对方持有的锁
- 死锁的避免完全由程序控制
- 可以采用的方法——资源排序
  - 在访问多个共享数据对象时，从全局考虑定义一个获得锁的顺序，并在整个程序中都遵守此顺序；释放锁时，要按加锁的反序释放

线程间的交互

- 多个线程间不仅需要同步使用共享数据，还需要进行某种协作，一种典型的协作方式是使用共享资源的wait()和notify()方法
- wait()和notify()
  - 线程在synchronized块中调用R.wait()释放共享对象R的对象锁，将自身阻塞并进入R的wait pool
  - 线程调用R.notify()，将共享对象R的wait pool中的一个线程唤醒（移入lock pool），等待R的对象锁
  - notifyAll()将共享对象wait pool中的所有线程都移入lock pool

线程间的交互


时间片用完
或Thread.yield()
运行
run()/
main()结束
终止
sleep()
o.wait()
t.join()
synchronized(o)
未获得对象锁
新建
start()
可运行
等待超时
等待队列
sleep()时间到
线程t结束或
等待时间到
等待目标
线程结束
o.notify()或
o.notifyAll()
获得对象锁
锁阻塞
阻塞

示例--Producer/Consumer

Producer线程：每隔300ms产生一个字母压入theStack栈中，共200个

```
class Producer implements Runnable {
    private SyncStack theStack;
    ...
    public void run() {
        char c;
        for (int i = 0; i < 200; i++) {
            c = (char) (Math.random() * 26 + 'A');
            theStack.push(c);
            System.out.println("Producer" + num + ":" + c);
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {}
        }
    }
}
```

示例--Producer/Consumer

Consumer线程：每隔300ms从theStack栈中取出一个字符，共200个

```
class Consumer extends Thread {
    private SyncStack theStack;
    ...
    public void run() {
        char c;
        for (int i = 0; i < 200; i++) {
            c = theStack.pop();
            System.out.println("Consumer" + num + ":" + c);
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {}
        }
    }
}
```

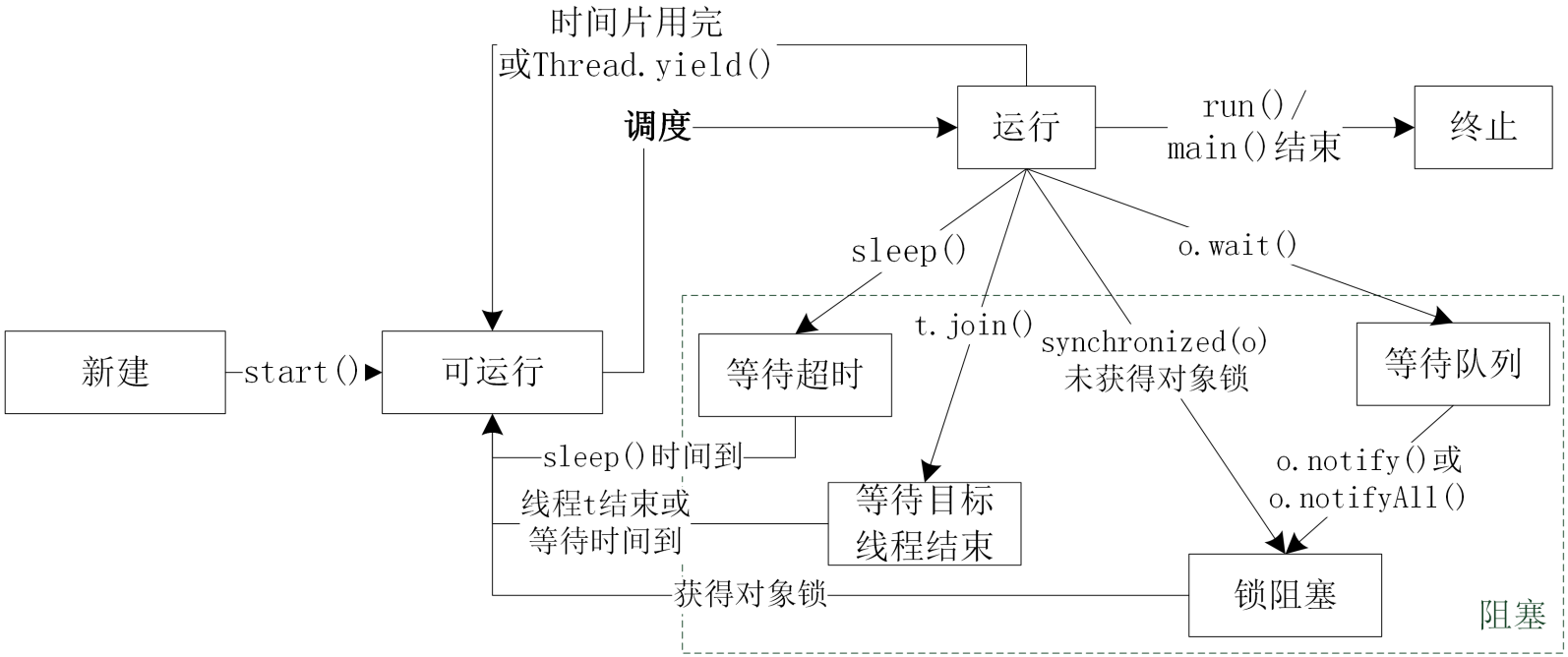
示例--Producer/Consumer

堆栈类SyncStack：为保证共享数据一致性，push()与pop()定义为synchronized

```
public class SyncStack {
    private Vector<Character> buffer = new Vector<Character>(400, 200);
    public synchronized char pop() {
        char c;
        while (buffer.size() == 0) {
            try {
                this.wait();          //pop()中加入wait()
            } catch (InterruptedException e) {}
        }
        c = ((Character) buffer.remove(buffer.size() - 1)).charValue();
        return c;
    }
    public synchronized void push(char c) {
        this.notify();                //push()中加入notify()
        Character charObj = new Character(c);
        buffer.addElement(charObj);
    }
}
```

提要

线程状态与生命周期


时间片用完
或Thread.yield()
调度
运行
run()/
main()结束
终止
sleep()
o.wait()
t.join()
synchronized(o)
未获得对象锁
新建
start()
可运行
等待超时
等待队列
sleep()时间到
线程t结束或
等待时间到
等待目标
线程结束
o.notify()或
o.notifyAll()
获得对象锁
锁阻塞
阻塞

线程状态总结

- New：线程被创建，尚未start()
- Runnable：线程可以被调度器选中执行
- Running：线程正在占用CPU执行
- Dead：run()执行结束，不会再被调度
- Blocked：线程此后还能运行，但某种条件阻止其运行
  - 当前线程调用sleep()进入睡眠状态
  - 当前线程调用t.join()等待线程t执行结束
  - 当前线程试图执行临界区代码片段但未获得对象锁
  - 当前线程调用共享对象的wait()方法

提要

管道流

- 用以实现线程间数据的直接传输
  - 管道输入流： PipedReader/PipedInputStream
  - 管道输出流： PipedWriter/PipedOutputStream
- 传输前，需要将一个线程的输出流与另一线程的输入流连接

```
PipedInputStream pin= new PipedInputStream( );
PipedOutputStream pout = new PipedOutputStream(pin);
```

或

```
PipedInputStream pin= new PipedInputStream( );
PipedOutputStream pout = new PipedOutputStream( );
pin.connect(pout); //或pout.connect(pin);
```

管道流示例（1）

- 发送者任务每次输出一句“hello”到管道流，接收者任务逐行地从管道流中读出数据并在控制台输出

```
public class PipedIO{
    public static void main(String[] args) throws Exception{
        Sender s=new Sender();
        Receiver r=new Receiver(s.getPipedWriter());
        ExecutorService exec=Executors.newCachedThreadPool();
        exec.execute(s);
        exec.execute(r);
        Thread.sleep(1000);
        exec.shutdownNow();
    }
}
```

管道流示例（1）

```
class Sender implements Runnable{
    private PipedWriter out=new PipedWriter();
    public PipedWriter getPipedWriter(){ return this.out; }
    public void run(){
        try{
            while(true)
                out.write("hello\n");
        } catch (Exception e){ ... }
    }
}
class Receiver implements Runnable{
    private PipedReader in=new PipedReader();
    public Receiver(PipedWriter o) throws IOException{ in.connect(o); }
    public void run(){
        BufferedReader br=new BufferedReader(in);
        try{
            while(true)
                System.out.println(br.readLine());
        } catch (IOException e){ ... }
    }
}
```

管道流示例（2）

- 输入一组单词，先将每个单词逆序，再将所有单词排序，最后将这些单词逆序输出
  - Rhymingwords. java
  - 程序处理流程：