

JAVA 程序设计



第8章 多线程

第8章 多线程

Java程序设计



- 8.1 线程的创建
- 8.2 线程的控制
- 8.3 线程的同步
- 8.4 并发API
- 8.5 流式操作及并行流

8.1 线程的创建





线程的创建



Java中的线程



- 进程：一个程序的执行
- 线程：程序中单个顺序的流控制称为线程
- 一个进程中可以含有多个线程
 - 在操作系统中可以查看线程数
 - 如：在Windows中，在任务管理器，右键，选择列，选中“线程数”
- 一个进程中的多个线程
 - 分享CPU（并发的或以时间片的方式）
 - 共享内存（如多个线程访问同一对象）

Java支持多线程



- Java从语言级别支持多线程
 - 如：Object中wait(), notify()
- java.lang中的类 Thread



- 线程体---- **run()**方法来实现的。
- 线程启动后，系统就自动调用run()方法。
- 通常，run()方法执行一个时间较长的操作
 - 如一个循环
 - 显示一系列图片
 - 下载一个文件



创建线程的两种方法

- 1 . 通过继承Thread类创建线程
- class MyThread **extends Thread** {
- **public void run()** {
- for(int i=0;i<100;i++) {
- System.out.print (" " + i);
- }
- } 示例 : [Thread1.java](#)
- 2 . 通过向Thread()构造方法**传递Runnable对象**来创建线程
- class MyTask **implements Runnable** {
- **public void run()** { ... }
- }
- Thread thread = new Thread(mytask);
- **thread.start();**
- 示例 : [Thread2.java](#)



匿名类及Lambda表达式

- 多线程 TestThread3.java
- 可用匿名类来实现Runnable
- `new Thread(){`
- `public void run() {`
- `for(int i=0; i<10; i++)`
- `System.out.println(i);`
- `}`
- `}.start();`
- 或者用Lambda表达式 (Java8以上)
 - ▣ `new Thread(()-> {。。。}).start();`
- 示例：TestThread4Anonymous.java

使用多个线程

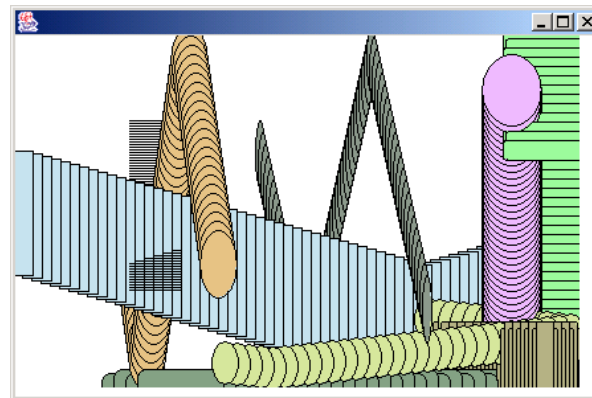


- 如 TestThread3.java

示例



- ThreadDrawJ.java 多线程绘图



示例:多线程下载

Java程序设计



- ThreadDownload.java

8.2 线程的控制

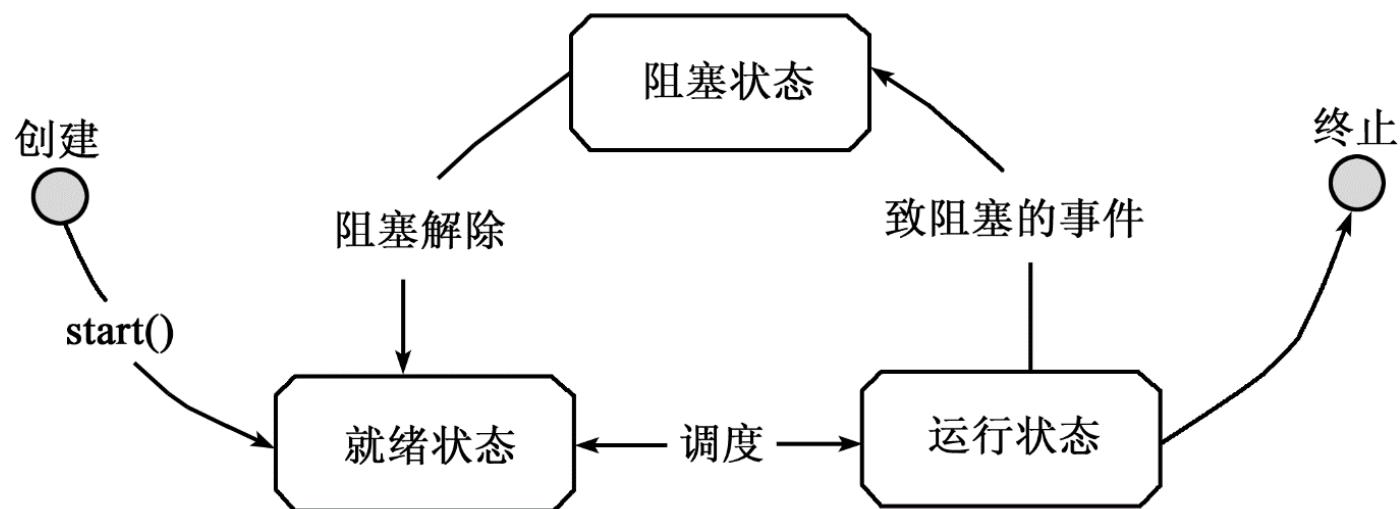




线程的控制



线程的状态与生命周期



对线程的基本控制



- 线程的启动

- `start()`

- 线程的结束

- 设定一个标记变量，以结束相应的循环及方法。

- 暂时阻止线程的执行

- `try{ Thread.sleep(1000);} catch(InterruptedException e){ }`

线程的优先级



- 设定线程的优先级

- `setPriority(int priority)`方法

- `MIN_PRIORITY` , `MAX_PRIORITY` , `NORM_PRIORITY`



- 线程有两种
 - 一类是普通线程（非Daemon线程）
 - 在Java程序中，若还有非Demon线程，则整个程序就不会结束
 - 一类是Daemon线程（守护线程，后台线程）
 - 如果普通线程结束了，则后台线程自动终止
 - 注：垃圾回收线程是后台线程
- 使用setDaemon(true);
- 示例：[TestThreadDaemon.java](#)

8.3 线程的同步





线程的同步



线程的不确定性



- 示例

- TestThreadCount.java

- 注cnt++实际编译为

```
0: getstatic      #2  
3: iconst_1  
4: iadd  
5: putstatic      #2  
8: return
```

```
0: getstatic      #2  
3: iconst_1  
4: iadd  
5: putstatic      #2  
8: return
```

多线程同步



- 同时运行的线程需要共享数据、
- 就必须考虑其它线程的状态与行为，这时就需要实现同步
- 示例：SyncCounter1.java



- Java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monitor（监视器），它上面一个称为“互斥锁（lock, mutex）”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。
 - 关键字synchronized 用来与对象的互斥锁联系。

synchronized



- synchronized的用法

- 对代码片断：

- `synchronized(对象){ }`

- 对某个方法：

- `synchronized` 放在方法声明中，
 - `public synchronized void push(char c){ }`
 - 相当于对`synchronized(this)`, 表示整个方法为同步方法。

- 示例：SyncCounter2.java



- 使用`wait()`方法可以释放对象锁
- 使用`notify()`或`notifyAll()`可以让等待的一个或所有线程进入就绪状态
- Java里面可以将`wait`和`notify`放在`synchronized`里面，是因为Java是这样处理的：
 - 在`synchronized`代码被执行期间，线程调用对象的`wait()`方法，会释放对象锁标志，然后进入等待状态，然后由其它线程调用`notify()`或者`notifyAll()`方法通知正在等待的线程。

生产者-消费者问题



示例：ProducerConsumerStack.java

```
class CubbyHole {
```

```
    private int index = 0;
```

```
    private int []data = new int[3];
```

```
    public synchronized void put(int value){
```

```
        while(index == data.length){
```

```
            try{
```

```
                this.wait();
```

```
            }catch(InterruptedException e){}
```

```
        }
```

```
        data[index] = value;
```

```
        index++;
```

```
        this.notify();
```

```
    }
```



```
• public synchronized int get(){  
•     while(index <=0){  
•         try{  
•             this.wait();  
•         }catch(InterruptedException e){}  
•     }  
•     index--;  
•     int val = data[index];  
•     this.notify();  
•     return val;  
• }  
• }
```

生产者-消费者问题



示例：ProducerConsumerStack.java

```
class CubbyHole {
```

```
    private int index = 0;
```

```
    private int []data = new int[3];
```

```
    public synchronized void put(int value){
```

```
        while(index == data.length){
```

```
            try{
```

```
                this.wait();
```

```
            }catch(InterruptedException e){}
```

```
        }
```

```
        data[index] = value;
```

```
        index++;
```

```
        this.notify();
```

```
    }
```

生产者-消费者问题（续）



```
• public synchronized int get(){  
•     while(index <=0){  
•         try{  
•             this.wait();  
•         }catch(InterruptedException e){}  
•     }  
•     index--;  
•     int val = data[index];  
•     this.notify();  
•     return val;  
• }  
• }
```

线程的死锁



- 示例

- ▣ DeadLockDemo.java



并发API中增加了更多的类

- JDK1.5中增加了更多的类，以便更灵活地使用锁机制
- java.util.concurrent.locks包
- Lock接口、ReentrantLock类
 - ▣ lock() tryLock() unlock()
- ReadWriteLock接口、ReentrantReadWriteLock类
 - ▣ .writeLock().lock(), .readLock().unlock()

8.4 并发API





并发API



并发API



- java.util.concurrent包及其子包
 - 从JDK1.5开始
 - 提供了一系列的工具，更好、更方便地使用线程
- 这里介绍几个实用的类
 - 单变量、集合、Timer、线程池

原子变量



- java.util.concurrent.atomic 包
 - AtomicInteger 类
 - getAndIncrement () 方法
- 示例 AtomicIntegerDemo.java



- 在JDK1.5以前
 - ArrayList/HashMap不是线程安全的
 - Vector及Hashtable是线程安全的
 - 产生一个线程安全的集合对象
 - Collections.synchronizedArrayList(list)



并发的集合类

- `java.util.concurrent`包中增加了一些方便的类
- `CopyOnWriteArrayList`、`CopyOnWriteArraySet`
 - 适合于很少写入而读取频繁的对象
- `ConcurrentHashMap`
 - `putIfAbsent()`, `remove()`, `replace()`
- `ArrayBlockingQueue`
 - 生产者与消费者，使用`put()`及`take()`
 - 示例：[BlockingQueueDemo.java](#)

使用线程池



- 线程池相关的类
 - ExecutorService 接口、ThreadPoolExecutor 类
 - Executors 工具类
- 常见的用法
 - ExecutorService pool = Executors.newCachedThreadPool();
 - 使用其execute(Runnable r)方法
- 示例 : [ThreadPoolDemo.java](#)

使用Timer



- 使用 `java.util.Timer` 类
重复某件事
 - 示例：[TimerTest.java](#)
- 使用 `javax.swing.Timer` 类
 - 重复执行 `ActionListener`
 - 示例：[TimerSwing.java](#)

特别注意



- 在线程中更新图形化界面，要调用
 - `SwingUtilites.invokeLater`
 - 示例 [ThreadDrawJ.java](#)



- java.util.concurrent.locks包
- Lock接口、ReentrantLock类
 - lock() tryLock() unlock()
 - 例：NoDeadLockDemo.java
- ReadWriteLock接口、ReentrantReadWriteLock类
 - .writeLock().lock(), .readLock().unlock()
 - 例：ArrayList2.java

Executor与Future



- 执行与任务分开，使用线程池
- Future异步取得结果

异步编程



- Executor执行与任务分开，使用线程池
- Future异步取得结果
- 在一定意义上实现异步编程
- 示例：ExcecutorAndFuture.java

进一步参考



- 《Java并发编程实践》 B.Goetz等著(433页)

8.5 流式操作及并行的流





流式操作及并行的流





起因：

- 既然集合是常见的任务，何不抽取出来
- `List<Integer> nums = Arrays.asList(1,2,3);`
- `nums. stream()`
- `.forEach(x->{System.out.println(x);});`
- 将常见的集合上的操作抽取出来，并能连续地进行操作
- 从Java8开始，提供了“流(stream)”操作

流 (stream)



- The new java.util.stream package provides utilities "to support functional-style operations on streams of values"
- 支持在流上的函数式风格的操作
- 得到流
 - ▣ `Stream<T> stream = collection.stream();`
- 操作流
 - ▣ `int sumOfWeights = blocks.stream()`
 - ▣ `.filter(b -> b.getColor() == RED)`
 - ▣ `.mapToInt(b -> b.getWeight())`
 - ▣ `.sum();`

示例：数组进行流化



- `Arrays.stream(a)`
- `.filter(i -> i>20)`
- `.map(i -> i*i)`
- `.sorted()`
- `.distinct()`
- `.limit(10)`
- `.max();`



- `Collection People = ...;`
- `people.stream()`
- `.filter(p -> p.age>20)`
- `.sorted(Comparator.comparing(Person::`
`getName))`
- `.limit(5)`
- `.mapToDouble(p -> p.score)`
- `.average();`



- `myOrders.stream()`
 - ▣ `.filter(t -> t.getBuyer().getAge()>65)`
 - ▣ `.map(t -> t.getSeller())`
 - ▣ `.distinct()`
 - ▣ `.sort(Compator.comparing(s->s.getName())`
 - ▣ `.forEach(s -> System.out.println(s.getName()));`
- 可以说：Lambda实现了函数式编程
- 是一种全新的思考问题的方法

stream的操作种类



- 流操作分成两类

- 中间的 - 中间的操作保持流打开状态，并允许后续的操作。

- 如：filter sorted limit map

- 末端的 - 末端的操作必须是对流的最终操作。

- 如：max min count forEach findAny

流操作的步骤



- 流涉及了这些步骤：
 - 从某个源头获得一个流。
 - 执行一个或更多的中间的操作。
 - 执行一个末端的操作。



如何得到Stream

- 对于数组
 - `Arrays.stream(ary)`
- 对于collection (包括List)
 - 用 `list.stream()`
- 对于Map
 - 没有流，但提供了类似的方法
 - 如 `map.putIfAbsent`
 - `map.computeIfPresent`
 - `map.merge`

Stream的子接口



- DoubleStream,
- IntStream,
- LongStream,
- Stream<T>

流的并行计算



- 只需将.stream()
- 换成 . parallelStream()
- 其他都不变，就可以实现并行计算
- stream就是为并行运算而生的



并行的流式操作

- 例 : UseParallelStream.java
- `List<Integer> a = Arrays.asList(1,2,5,7,3);`
- `System.out.println(`
- `a.parallelStream()`
- `.mapToInt(i->(int)i)`
- `.filter(i -> i>2)`
- `.map(i -> i*i)`
- `.sorted()`
- `.distinct()`
- `.limit(10)`
- `.max()`
- `);`



中间的操作

- filter - 排除所有与断言不匹配的元素。
- map - 通过Function对元素执行一对一的转换。
- flatMap - 通过FlatMapper将每个元素转变为无或更多的元素。
- peek - 对每个遇到的元素执行一些操作。主要对调试很有用。
- distinct - 根据.equals行为排除所有重复的元素。这是一个有状态的操作。
- sorted - 确保流中的元素在后续的操作中，按照比较器（Comparator）决定的顺序访问。这是一个有状态的操作。
- limit - 保证后续的操作所能看到的最大数量的元素。
- substream - 确保后续的操作只能看到一个范围的（根据index）元素。
- skip- 忽略一些元素
- mapToDouble mapToInt mapToLong 类型转换



末端的操作

- `forEach` - 对流中的每个元素执行一些操作。
- `toArray` - 将流中的元素倾倒入一个数组。
- `min` - 根据一个比较器找到流中元素的最小值。
- `max` - 根据一个比较器找到流中元素的最大值。
- `count` - 计算流中元素的数量。
- `anyMatch` - 判断流中是否至少有一个元素匹配断言。这是一个短路的操作。
- `allMatch` - 判断流中是否每一个元素都匹配断言。这是一个短路的操作。
- `noneMatch` - 判断流中是否没有一个元素匹配断言。这是一个短路的操作。
- `findFirst` - 查找流中的第一个元素。这是一个短路的操作。
- `findAny` - 查找流中的任意元素，可能对某些流要比`findFirst`代价低。这是一个短路的操作。
- 注：子接口还有更多的操作，如 `average` 等等