多进程和多线程简介

多进程概念

- 当前的操作系统都是多任务OS，每个独立执行的任务就是一个进程。
- OS将时间划分为多个时间片（时间很短），每个时间片内将CPU分配给某一个任务，时间片结束，CPU将自动回收，再分配给另外任务。从外部看，所有任务是同时在执行。但是在CPU上，任务是按照串行依次运行（单核CPU）。如果是多核，多个进程任务可以并行。但是单个核上，多进程只能串行执行。
- 多进程的优点
    - 可以同时运行多个任务
    - 程序因IO堵塞时，可以释放CPU，让CPU为其他程序服务
    - 当系统有多个CPU时，可以为多个程序同时服务
        - 我们的CPU不再提高频率，而是提高核数
        - 2005年Herb Sutter的文章 The free lunch is over，指明多核和并行程序才是提高程序性能的唯一办法
- 多进程的缺点
    - 太笨重，不好管理
- 太笨重，不好切换

多线程概念

- 一个程序可以包括多个子任务，可串/并行，每个子任务可以称为一个线程
- 如果一个子任务阻塞，程序可以将CPU调度另外一个子任务进行工作。这样CPU还是保留在本程序中，而不是被调度到别的程序(进程)去。这样，提高本程序所获得CPU时间和利用率。

多进程和多线程对比

- 线程共享数据
- 线程通讯更高效
- 线程更轻量级，更容易切换
- 多个线程更容易管理

Java多线程实现

Java 多线程创建

java.lang.Thread：线程继承Thread类，实现run方法

```
public class Thread1 extends Thread{
    public void run()
    {
        System.out.println("hello");
    }
}
```

java.lang.Runnable接口：线程实现Runnable接口，实现run方法

```
public class Thread2 implements Runnable{
    public void run()
    {
        System.out.println("hello");
    }
}
```

> Java的四个主要接口：
> Clonable，用于对象克隆
> Comparable，用于对象比较
> Serializable，用于对象序列化
> Runnable，用于对象线程化

Java 多线程启动

> Thread方式：
>
> 可以提供过继承Thread类来创建线程。
> 通过start方法来启动线程的run方法。

```
public class Thread1 extends Thread{
    public void run()
    {
        System.out.println("hello");
    }
    public static void main(String[] a)
    {
        new Thread1().start();
    }
}
```

> Runnable方法：
>
> 可以通过实现Runnable接口来创建线程
> 实现Runnable的对象必须包装在Thread类里面，才可以启动；不能直接对Runnable的对象进行start方法。
> 通过start方法来启动线程的run方法

```
public class Thread2 implements Runnable{
    public void run()
    {
        System.out.println("hello");
    }
    public static void main(String[] a)
    {
        new Thread(new Thread2()).start();
    }
}
```

> 第一条规则：

> 调用run方法来启动run方法，将会是串行运行。
> 调用start方法来启动run方法，将会是并行运行。

```
public class ThreadDemo0
{
    public static void main(String args[]) throws Exception
    {
        //new TestThread0().run(); //串行
        new TestThread0().start();  //并行
        while(true)
        {
            System.out.println("main thread is running");
            Thread.sleep(10);
        }
    }
}
 class TestThread0
{
    public void run()
    {
        while(true)
        {
            System.out.println(" TestThread1 is running");
            try {
                Thread.sleep(1000); //1000毫秒
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

> 第二条规则：
>
> main线程可能早于子线程结束。
> main线程和子线程都结束了，整个程序才算终止。

```
public class ThreadDemo2
{
    public static void main(String args[]) throws InterruptedException
    {
        new TestThread2().start();
//      while(true)
//      {
//          System.out.println("main thread is running");
//          Thread.sleep(1000);
//      }
    }
}
 class TestThread2 extends Thread
{
    public void run()
    {
        while(true)
```

```
        {
            System.out.println("TestThread2" +
            "  is running");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

> 第三条规则：
>
> 实现Runnable的对象必须包装在Thread类里面，才可以启动。
> 不能直接对Runnable的对象进行start方法。

```
public class ThreadDemo3
{
    public static void main(String args[])
    {
        //new TestThread3().start();
        //Runnable对象必须放在一个Thread类中才能运行
        TestThread3 tt= new TestThread3();//创建TestThread类的一个实例
        Thread t= new Thread(tt);//创建一个Thread类的实例
        t.start();//使线程进入Runnable状态
        while(true)
        {
            System.out.println("main thread is running");
            try {
                Thread.sleep(1000); //1000毫秒
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
class TestThread3 implements Runnable //extends Thread
{
    //线程的代码段，当执行start()时，线程从此出开始执行
    public void run()
    {
        while(true)
        {
            System.out.println(Thread.currentThread().getName() +
            " is running");
            try {
                Thread.sleep(1000); //1000毫秒
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
```

```
        }
    }
}
```

> 第四条规则：
>
> 一个线程对象不能多次start，多次start将报异常。
> 多个线程对象都start后，哪一个先执行，完全由JVM/操作系统来主导，程序员无法指定。

```
public class ThreadDemo4
{
    public static void main(String [] args)
    {
        TestThread4 t=new TestThread4();
        t.start();
        //t.start();
        //t.start();
        //t.start();
        TestThread4 t1=new TestThread4();
        t1.start();
    }
}

class TestThread4 extends Thread
{
    public void run()
    {
        while(true)
        {
            System.out.println(Thread.currentThread().getName() +
            " is running");
            try {
                Thread.sleep(1000); //1000毫秒
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

Java 多线程实现对比

- Thread占据了父类的名额，不如Runnable方便
- Thread 类实现Runnable
- Runnable启动时需要Thread类的支持
- Runnable 更容易实现多线程中资源共享
- 结论：建议实现Runnable接口来完成多线程

Java多线程信息共享

- 线程类
  - 通过继承Thread或实现Runnable
  - 通过start方法，调用run方法， run方法工作
  - 线程run结束后，线程退出
- 粗粒度：子线程与子线程之间、和main线程之间缺乏交流
- 细粒度：线程之间有信息交流通讯
  - 通过共享变量达到信息共享
  - JDK原生库暂不支持发送消息 (类似MPI并行库直接发送消息)

static变量

同一个Runnable类的成员变量来达到共享。

示例代码（线程卖盘）

```
public class ThreadDemo0
{
    public static void main(String [] args)
    {
        new TestThread0().start();
        new TestThread0().start();
        new TestThread0().start();
        new TestThread0().start();
    }
}
class TestThread0 extends Thread
{
    //private int tickets=100;            //每个线程卖100张，没有共享
    private static int tickets=100;  //static变量是共享的，所有的线程共享
    public void run()
    {
        while(true)
        {
            if(tickets>0)
            {
                System.out.println(Thread.currentThread().getName() +
                " is selling ticket " + tickets);
                tickets = tickets - 1;
            }
            else
            {
                break;
            }
        }
    }
}
```

部分运行结果如下：

```
Thread-0 is selling ticket 100
Thread-0 is selling ticket 99
Thread-1 is selling ticket 100
Thread-1 is selling ticket 97
Thread-2 is selling ticket 100
Thread-3 is selling ticket 100
Thread-2 is selling ticket 94
Thread-1 is selling ticket 96
Thread-0 is selling ticket 98
Thread-1 is selling ticket 92
Thread-1 is selling ticket 90
Thread-1 is selling ticket 89
Thread-1 is selling ticket 88
Thread-1 is selling ticket 87
Thread-1 is selling ticket 86
Thread-1 is selling ticket 85
Thread-1 is selling ticket 84
Thread-1 is selling ticket 83
Thread-1 is selling ticket 82
Thread-1 is selling ticket 81
Thread-1 is selling ticket 80
Thread-2 is selling ticket 93
Thread-3 is selling ticket 94
Thread-3 is selling ticket 77
Thread-3 is selling ticket 76
Thread-3 is selling ticket 75
Thread-3 is selling ticket 74
```

普通成员变量

示例代码

```
public class ThreadDemo1
{
    public static void main(String [] args)
    {
        TestThread1 t=new TestThread1();
        new Thread(t).start();
        new Thread(t).start();
        new Thread(t).start();
        new Thread(t).start();
    }
}
class TestThread1 implements Runnable
{
    private int tickets=100;
    public void run()
    {
        while(true)
        {
            if(tickets>0)
            {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                tickets--;
                System.out.println(Thread.currentThread().getName() +" is
selling ticket " + tickets);
            }
            else
            {
                break;
```

```
            }


        }
    }

}
```

> TestThread1只被创建一次，就是t。
> 而new Thread（t）并没有创建TestThread1对象，而是把t包装成线程对象，然后启动。
> 第7行到第10行代码使用的是同一个TestThread1的对象t。

部分运行结果如下：

```
Thread-0 is selling ticket 35
Thread-3 is selling ticket 34
Thread-2 is selling ticket 34
Thread-0 is selling ticket 33
Thread-1 is selling ticket 32
Thread-3 is selling ticket 31
Thread-2 is selling ticket 31
Thread-1 is selling ticket 30
Thread-0 is selling ticket 29
Thread-2 is selling ticket 28
Thread-3 is selling ticket 27
Thread-1 is selling ticket 25
Thread-0 is selling ticket 26
Thread-2 is selling ticket 24
Thread-3 is selling ticket 23
Thread-1 is selling ticket 22
Thread-0 is selling ticket 21
Thread-3 is selling ticket 20
Thread-2 is selling ticket 19
Thread-1 is selling ticket 17
Thread-0 is selling ticket 17
Thread-3 is selling ticket 16
Thread-2 is selling ticket 15
Thread-0 is selling ticket 13
Thread-1 is selling ticket 13
Thread-3 is selling ticket 12
Thread-2 is selling ticket 12
Thread-1 is selling ticket 11
Thread-0 is selling ticket 10
Thread-3 is selling ticket 8
Thread-2 is selling ticket 8
Thread-1 is selling ticket 7
Thread-0 is selling ticket 6
Thread-2 is selling ticket 4
Thread-3 is selling ticket 4
Thread-1 is selling ticket 3
Thread-0 is selling ticket 2
Thread-3 is selling ticket 0
Thread-1 is selling ticket 0
Thread-2 is selling ticket 0
```

存在问题

- 工作缓存副本
    - 某线程修改了自己工作缓存中的值，其他线程并不知晓，继续用自己的工作缓存中的值，但该值不能反映最新的变量值，大家都是用的前一刻变量值。
- 关键步骤（临界区）缺乏加锁限制
  一次只允许一个线程对某一变量进行修改操作。

volatile关键字

采用volatile 关键字修饰变量，保证不同线程对共享变量操作时的可见性。

示例代码

```
public class ThreadDemo2
{
    public static void main(String args[]) throws Exception
    {
        TestThread2 t = new TestThread2();
        t.start();
        Thread.sleep(2000);
        t.flag = false;
        System.out.println("main thread is exiting");
    }
}

class TestThread2 extends Thread
{
    //boolean flag = true;    //子线程不会停止
    volatile boolean flag = true;  //用volatile修饰的变量可以及时在各线程里面通知
    public void run()
    {
        int i=0;
        while(flag)
        {
            i++;
        }
        System.out.println("test thread3 is exiting");
    }
}
```

运行结果如下：

```
main thread is exiting
test thread3 is exiting
```

关键步骤加锁

- 关键步骤加锁限制
  - 互斥：某一个线程运行一个代码段(关键区)，其他线程不能同时运行这个代码段
  - 同步：多个线程的运行，必须按照某一种规定的先后顺序来运行
  - 互斥是同步的一种特例
- 互斥的关键字是synchronized
  - synchronized代码块/函数，只能一个线程进入
  - synchronized加大性能负担，但是使用简便

示例代码

```
public class ThreadDemo3 {
    public static void main(String[] args) {
```

```
        TestThread3 t = new TestThread3();
        new Thread(t, "Thread-0").start();
        new Thread(t, "Thread-1").start();
        new Thread(t, "Thread-2").start();
        new Thread(t, "Thread-3").start();
    }
}

class TestThread3 implements Runnable {
    private volatile int tickets = 100; // 多个 线程在共享的
    String str = new String("");

    public void run() {
        while (true) {
            synchronized(str){   //同步代码块
                sale();
            }
            try {
                Thread.sleep(100);
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
            if (tickets <= 0) {
                break;
            }
        }

    }

    public synchronized void sale() { // 同步函数
        if (tickets > 0) {
            System.out.println(Thread.currentThread().getName() + " is saling
ticket " + tickets--);
        }
    }

}
```

部分运行结果如下：

```
Thread-0 is saling ticket 36
Thread-3 is saling ticket 35
Thread-2 is saling ticket 34
Thread-1 is saling ticket 33
Thread-3 is saling ticket 32
Thread-0 is saling ticket 31
Thread-2 is saling ticket 30
Thread-1 is saling ticket 29
Thread-3 is saling ticket 28
Thread-0 is saling ticket 27
Thread-2 is saling ticket 26
Thread-1 is saling ticket 25
Thread-0 is saling ticket 24
Thread-3 is saling ticket 23
Thread-2 is saling ticket 22
Thread-1 is saling ticket 21
Thread-3 is saling ticket 20
Thread-0 is saling ticket 19
Thread-2 is saling ticket 18
Thread-1 is saling ticket 17
Thread-3 is saling ticket 16
Thread-0 is saling ticket 15
Thread-2 is saling ticket 14
Thread-1 is saling ticket 13
Thread-2 is saling ticket 12
Thread-0 is saling ticket 11
Thread-3 is saling ticket 10
Thread-1 is saling ticket 9
Thread-2 is saling ticket 8
Thread-3 is saling ticket 7
Thread-0 is saling ticket 6
Thread-1 is saling ticket 5
Thread-2 is saling ticket 4
Thread-3 is saling ticket 3
Thread-0 is saling ticket 2
Thread-1 is saling ticket 1
```

Java多线程管理

线程状态

- NEW 刚创建(new)
- RUNNABLE 就绪态(start)
- RUNNING 运行中(run)
- BLOCK 阻塞(sleep)
- TERMINATED 结束

- Thread的部分API已经废弃
  - 暂停和恢复 suspend/resume
  - 消亡 stop/destroy
- 线程阻塞和唤醒
  - sleep，时间一到，自己会醒来
  - wait/notify/notifyAll，等待，需要别人来唤醒（不被唤醒则一直等待）
  - join，等待另外一个线程结束
  - interrupt，向另外一个线程发送中断信号，该线程收到信号，会触发InterruptedException(可解除阻塞)，并进行下一步处理

生产者与消费者问题

> 生产者不断的往仓库中存放产品，消费者从仓库中消费产品。
> 其中生产者和消费者都可以有若干个。
> 仓库规则：容量有限，库满时不能存放，库空时不能取产品 。

主类

```
package product;
public class ProductTest {
    public static void main(String[] args) throws InterruptedException {
        Storage storage = new Storage();


        Thread consumer1 = new Thread(new Consumer(storage));
        consumer1.setName("消费者1");
        Thread consumer2 = new Thread(new Consumer(storage));
        consumer2.setName("消费者2");
        Thread producer1 = new Thread(new Producer(storage));
        producer1.setName("生产者1");
        Thread producer2 = new Thread(new Producer(storage));
        producer2.setName("生产者2");

        producer1.start();
        producer2.start();
        Thread.sleep(1000);
        consumer1.start();
        consumer2.start();
    }

}
```

仓库

```
package product;
/**
 *仓库
 */
class Storage {
    // 仓库容量为10
    private Product[] products = new Product[10];
    private int top = 0;

    // 生产者往仓库中放入产品
    public synchronized void push(Product product) {
        while (top == products.length) {
            try {
                System.out.println("producer wait");
                wait();//仓库已满，等待
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        //把产品放入仓库
```

```
        products[top++] = product;
        System.out.println(Thread.currentThread().getName() + " 生产了产品"
                + product);
        System.out.println("producer notifyAll");
        notifyAll();//唤醒等待线程
    }

    // 消费者从仓库中取出产品
    public synchronized Product pop() {
        while (top == 0) {
            try {
                System.out.println("consumer wait");
                wait();//仓库空，等待
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        //从仓库中取产品
        --top;
        Product p = new Product(products[top].getId(), products[top].getName());
        products[top] = null;
        System.out.println(Thread.currentThread().getName() + " 消费了产品" + p);
        System.out.println("comsumer notifyAll");
        notifyAll();//唤醒等待线程
        return p;
    }

}
```

产品类

```
package product;

/**

 * 产品类
   */
   class Product {
   private int id;// 产品id
   private String name;// 产品名称

   public Product(int id, String name) {
    this.id = id;
    this.name = name;
   }
   public String toString() {
    return "(产品ID: " + id + " 产品名称: " + name + ")";
   }
   public int getId() {
    return id;
   }
   public void setId(int id) {
    this.id = id;
   }
   public String getName() {
```

```
    return name;
   }
   public void setName(String name) {
    this.name = name;
   }
   }
```

生产者

```
package product;

import java.util.Random;

/**

 * 生产者
   */
   class Producer implements Runnable {
   private Storage storage;

   public Producer(Storage storage) {
    this.storage = storage;
   }

   @Override
   public void run() {
    int i = 0;
    Random r = new Random();
    while(i<10)
    {
        i++;
        Product product = new Product(i, "电话" + r.nextInt(100));
        storage.push(product);
    }
   }
   }
```

消费者

```
package product;

/**

 * 消费者
   */
   class Consumer implements Runnable {
   private Storage storage;

   public Consumer(Storage storage) {
    this.storage = storage;
   }

   public void run() {
    int i = 0;
```

```
    while(i<10)
    {
        i++;
        storage.pop();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

   }
   }
```

该示例中两个生产者分别生产10个产品，两个消费者分别消费10个产品。

部分运行结果如下：

```
comsumer notifyAll
生产者1 生产了产品(产品ID: 9 产品名称: 电话50)
producer notifyAll
producer wait
producer wait
消费者2 消费了产品(产品ID: 9 产品名称: 电话50)
comsumer notifyAll
生产者1 生产了产品(产品ID: 10 产品名称: 电话95)
producer notifyAll
producer wait
消费者1 消费了产品(产品ID: 10 产品名称: 电话95)
comsumer notifyAll
生产者2 生产了产品(产品ID: 6 产品名称: 电话39)
producer notifyAll
producer wait
消费者1 消费了产品(产品ID: 6 产品名称: 电话39)
comsumer notifyAll
生产者2 生产了产品(产品ID: 7 产品名称: 电话61)
producer notifyAll
producer wait
消费者2 消费了产品(产品ID: 7 产品名称: 电话61)
comsumer notifyAll
生产者2 生产了产品(产品ID: 8 产品名称: 电话48)
producer notifyAll
producer wait
消费者1 消费了产品(产品ID: 8 产品名称: 电话48)
comsumer notifyAll
生产者2 生产了产品(产品ID: 9 产品名称: 电话68)
producer notifyAll
producer wait
消费者2 消费了产品(产品ID: 9 产品名称: 电话68)
comsumer notifyAll
生产者2 生产了产品(产品ID: 10 产品名称: 电话92)
producer notifyAll
```

线程要做自己的主

- 线程被动地暂停和终止
  - 依靠别的线程来拯救自己 ☹☹☹
  - 没有及时释放资源
- 线程主动暂停和终止
  - 定期监测共享变量
  - 如果需要暂停或者终止，先释放资源，再主动动作 😋😋😋
  - 暂停：Thread.sleep()，休眠
  - 终止：run方法结束，线程终止

示例代码

```
package interrupt;

public class InterruptTest {

    public static void main(String[] args) throws InterruptedException {
        TestThread1 t1 = new TestThread1();
        TestThread2 t2 = new TestThread2();

        t1.start();
        t2.start();

        // 让线程运行一会儿后中断
        Thread.sleep(2000);
        t1.interrupt();
        t2.flag = false;
        System.out.println("main thread is exiting");
    }

}

class TestThread1 extends Thread {
    public void run() {
        // 判断标志，当本线程被别人interrupt后，JVM会被本线程设置interrupted标记
        while (!interrupted()) {
            System.out.println("test thread1 is running");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
                break;
            }
        }
        System.out.println("test thread1 is exiting");
    }
}

class TestThread2 extends Thread {
    public volatile boolean flag = true;
    public void run() {
        // 判断标志，当本线程被别人interrupt后，JVM会被本线程设置interrupted标记
        while (flag) {
```

```
            System.out.println("test thread2 is running");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("test thread2 is exiting");
    }
}
```

> interrupted() 是Thread类的方法，用来测试当前线程是否收到一个INTERRUPT信号。
> 如果收到，该方法返回true，否则返回false。

运行结果如下：

```
test thread1 is running
test thread2 is running
test thread1 is running
test thread2 is running
main thread is exiting
java.lang.InterruptedException: sleep interrupted
test thread1 is exiting
test thread2 is exiting
```

两种方式比较：

用interrupt这个标志来中断异常的话，需要自己去添加异常处理，并且此处的异常可能会让你来不及释放资源。
定期去监测flag变量，当变量被修改了，就可以很优雅地释放所有资源，然后主动退出。

多线程死锁

- 每个线程互相持有别人需要的锁(哲学家吃面问题)
- 预防死锁，对资源进行等级排序

> 死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。

示例代码

```
package deadlock;

import java.util.concurrent.TimeUnit;

public class ThreadDemo5
{
    public static Integer r1 = 1;
    public static Integer r2 = 2;
    public static void main(String args[]) throws InterruptedException
    {
        TestThread51 t1 = new TestThread51();
        t1.start();
```

```
        TestThread52 t2 = new TestThread52();
        t2.start();
    }
}

class TestThread51 extends Thread
{
    public void run()
    {
        //先要r1,再要r2
        synchronized(ThreadDemo5.r1)
        {
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized(ThreadDemo5.r2)
            {
                System.out.println("TestThread51 is running");
            }
        }
    }
}
class TestThread52 extends Thread
{
    public void run()
    {
        //先要r2,再要r1
        synchronized(ThreadDemo5.r2)
        {
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized(ThreadDemo5.r1)
            {
                System.out.println("TestThread52 is running");
            }
        }
    }
}
```

> TimeUnit是JDK 5引入的新类
> 位于java.util.concurrent包中。
> 它提供了时间单位粒度和一些时间转换、计时和延迟等函数。

该代码段中t1拿到r1，t2拿到r2，但双发下一步需要取得的锁被对方持有从而无法进行下去，由此产生了死锁。
若两个进程都先拿r1，再拿r2则可以避免死锁

守护（后台）线程

- 普通线程的结束，是run方法运行结束
- 守护线程的结束，是run方法运行结束，或main函数结束
- 守护线程永远不要访问资源，如文件或数据库等

示例代码

```
package daemon;

public class ThreadDemo4
{
    public static void main(String args[]) throws InterruptedException
    {
        TestThread4 t = new TestThread4();
        t.setDaemon(true);
        t.start();
        Thread.sleep(2000);
        System.out.println("main thread is exiting");
    }
}
 class TestThread4 extends Thread
{
    public void run()
    {
        while(true)
        {
            System.out.println("TestThread4" +
            "  is running");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

运行结果如下：

```
<terminated> ThreadDemo4 [Java Application]
TestThread4  is running
TestThread4  is running
main thread is exiting
TestThread4  is running
```

Java并发框架Executor

并行计算

- 并行模式
  - 主从模式 (Master-Slave)
  - Worker模式(Worker-Worker)
- Java并发编程
  - Thread/Runnable/Thread组管理
  - Executor
  - Fork-Join框架

线程组管理

线程组 T h r e a d G r o u p ThreadGroupThreadGroup

- 线程的集合
- 树形结构，大线程组可以包括小线程组
- 可以通过enumerate方法遍历组内的线程，执行操作
- 能够有效管理多个线程，但是管理效率低
- 任务分配和执行过程高度耦合
- 重复创建线程、关闭线程操作，无法重用线程

示例代码

```
package threadgroup;

import java.util.concurrent.TimeUnit;

public class Main {

    public static void main(String[] args) {

        // 创建线程组
        ThreadGroup threadGroup = new ThreadGroup("Searcher");
        Result result=new Result();

        // 创建一个任务，10个线程完成
        Searcher searchTask=new Searcher(result);
        for (int i=0; i<10; i++) {
            Thread thread=new Thread(threadGroup, searchTask);
            thread.start();
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("========华丽丽0=======");

        // 查看线程组消息
        System.out.printf("active 线程数量: %d\n",threadGroup.activeCount());
        System.out.printf("线程组信息明细\n");
        threadGroup.list();
```

```
        System.out.println("========华丽丽1=======");

        // 遍历线程组
        Thread[] threads=new Thread[threadGroup.activeCount()];
        threadGroup.enumerate(threads);
        for (int i=0; i<threadGroup.activeCount(); i++) {
            System.out.printf("Thread %s:
%s\n",threads[i].getName(),threads[i].getState());
        }
        System.out.println("========华丽丽2=======");

        // Wait for the finalization of the Threadds
        waitFinish(threadGroup);

        // Interrupt all the Thread objects assigned to the ThreadGroup
        threadGroup.interrupt();
    }

    public static void waitFinish(ThreadGroup threadGroup) {
        while (threadGroup.activeCount()>9) {
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

```
package threadgroup;

/**

 * 搜索结果类
   *
    */
   public class Result {

   private String name;
   public String getName() {
    return name;
   }
   public void setName(String name) {
    this.name = name;
   }

}
```

```
package threadgroup;

import java.util.Date;
import java.util.Random;
import java.util.concurrent.TimeUnit;

public class Searcher implements Runnable {
```

```
    private Result result;

    public Searcher(Result result) {
        this.result=result;
    }

    @Override
    public void run() {
        String name=Thread.currentThread().getName();
        System.out.printf("Thread %s: 启动\n",name);
        try {
            doTask();
            result.setName(name);
        } catch (InterruptedException e) {
            System.out.printf("Thread %s: 被中断\n",name);
            return;
        }
        System.out.printf("Thread %s: 完成\n",name);
    }

    private void doTask() throws InterruptedException {
        Random random=new Random((new Date()).getTime());
        int value=(int)(random.nextDouble()*100);
        System.out.printf("Thread %s:
%d\n",Thread.currentThread().getName(),value);
        TimeUnit.SECONDS.sleep(value);
    }

}
```

> activeCount，返回线程组中还处于active的线程数
> enumerate，将线程组中active的线程拷贝到数组中
> interrupt，对线程组中所有的线程发出interrupt信号
> list，打印线程组中所有的线程信息

运行结果如下

```
Thread Thread-0: 启动
Thread Thread-0: 36
Thread Thread-1: 启动
Thread Thread-1: 25
Thread Thread-2: 启动
Thread Thread-2: 52
Thread Thread-3: 启动
Thread Thread-3: 43
Thread Thread-4: 启动
Thread Thread-4: 71
Thread Thread-5: 启动
Thread Thread-5: 62
Thread Thread-6: 启动
Thread Thread-6: 43
Thread Thread-7: 启动
Thread Thread-7: 32
Thread Thread-8: 启动
Thread Thread-8: 60
Thread Thread-9: 启动
Thread Thread-9: 51
========华丽丽0=======
active 线程数量: 10
线程组信息明细
java.lang.ThreadGroup[name=Searcher,maxpri=10]
    Thread[Thread-0,5,Searcher]
    Thread[Thread-1,5,Searcher]
    Thread[Thread-2,5,Searcher]
    Thread[Thread-3,5,Searcher]
    Thread[Thread-4,5,Searcher]
    Thread[Thread-5,5,Searcher]
    Thread[Thread-6,5,Searcher]
    Thread[Thread-7,5,Searcher]
    Thread[Thread-8,5,Searcher]
    ========华丽丽1=======
    Thread Thread-0: TIMED_WAITING
    Thread Thread-1: TIMED_WAITING
    Thread Thread-2: TIMED_WAITING
    Thread Thread-3: TIMED_WAITING
    Thread Thread-4: TIMED_WAITING
    Thread Thread-5: TIMED_WAITING
    Thread Thread-6: TIMED_WAITING
    Thread Thread-7: TIMED_WAITING
    Thread Thread-8: TIMED_WAITING
    Thread Thread-9: TIMED_WAITING
    ========华丽丽2=======
    Thread Thread-1: 完成
    Thread Thread-3: 被中断
    Thread Thread-8: 被中断
    Thread Thread-2: 被中断
    Thread Thread-6: 被中断
    Thread Thread-0: 被中断
    Thread Thread-5: 被中断
    Thread Thread-7: 被中断
    Thread Thread-4: 被中断
```

并发框架Executor

- 从JDK 5开始提供Executor FrameWork (java.util.concurrent.*)
  - 分离任务的创建和执行者的创建
  - 线程重复利用(new线程代价很大)
- 理解共享线程池的概念
  - 预设好的多个Thread，可弹性增加
  - 多次执行很多很小的任务
  - 任务创建和执行过程解耦
  - 程序员无需关心线程池执行任务过程
- 主要类
  - Executors.newCachedThreadPool/newFixedThreadPool 创建线程池
  - ExecutorService 线程池服务
  - Callable 具体的逻辑对象(线程类)
  - Future 返回结果

> Callable和Runnable是等价的，可以用来执行一个任务。
> Runnab的run方法没有返回值，而Callable的call方法可以有返回值。

示例代码1

```
package executor.example1;

public class Main {

    public static void main(String[] args) throws InterruptedException {
        // 创建一个执行服务器
        Server server=new Server();

        // 创建100个任务，并发给执行器，等待完成
        for (int i=0; i<100; i++){
            Task task=new Task("Task "+i);
            Thread.sleep(10);
            server.submitTask(task);
        }
        server.endServer();
    }

}
```

```
package executor.example1;

import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
```

```
/**

 * 执行服务器
   *
    */
   public class Server {

   //线程池
   private ThreadPoolExecutor executor;
```

```
   public Server(){
    executor=(ThreadPoolExecutor)Executors.newCachedThreadPool(); //弹性容量
    //executor=(ThreadPoolExecutor)Executors.newFixedThreadPool(5);
   }

   //向线程池提交任务
   public void submitTask(Task task){
    System.out.printf("Server: A new task has arrived\n");
    executor.execute(task); //执行  无返回值


    System.out.printf("Server: Pool Size: %d\n",executor.getPoolSize());
    System.out.printf("Server: Active Count: %d\n",executor.getActiveCount());
    System.out.printf("Server: Completed Tasks:
%d\n",executor.getCompletedTaskCount());

   }

   public void endServer() {
    executor.shutdown();
   }
   }

   package executor.example1;
```

```
import java.util.Date;
import java.util.concurrent.TimeUnit;
/**

 * Task 任务类

 * @author Tom
   *
    */
   public class Task implements Runnable {

   private String name;

   public Task(String name){
    this.name=name;
   }

   public void run() {
    try {
        Long duration=(long)(Math.random()*1000);
        System.out.printf("%s: Task %s: Doing a task during %d
seconds\n",Thread.currentThread().getName(),name,duration);
        Thread.sleep(duration);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }


    System.out.printf("%s: Task %s: Finished on:
%s\n",Thread.currentThread().getName(),name,new Date());
```

```
    }
    }
```

部分运行结果如下：

```
pool-1-thread-2: Task Task 1: Doing a task during 960 seconds
Server: Active Count: 2
Server: Completed Tasks: 0
Server: A new task has arrived
Server: Pool Size: 3
pool-1-thread-3: Task Task 2: Doing a task during 722 seconds
Server: Active Count: 3
Server: Completed Tasks: 0
Server: A new task has arrived
Server: Pool Size: 4
Server: Active Count: 4
pool-1-thread-4: Task Task 3: Doing a task during 851 seconds
Server: Completed Tasks: 0
Server: A new task has arrived
Server: Pool Size: 5
pool-1-thread-5: Task Task 4: Doing a task during 354 seconds
Server: Active Count: 5
Server: Completed Tasks: 0
Server: A new task has arrived
Server: Pool Size: 6
pool-1-thread-6: Task Task 5: Doing a task during 388 seconds
Server: Active Count: 6
Server: Completed Tasks: 0
Server: A new task has arrived
Server: Pool Size: 7
pool-1-thread-7: Task Task 6: Doing a task during 939 seconds
Server: Active Count: 7
Server: Completed Tasks: 0
Server: A new task has arrived
Server: Pool Size: 8
pool-1-thread-8: Task Task 7: Doing a task during 887 seconds
Server: Active Count: 8
Server: Completed Tasks: 0
Server: A new task has arrived
Server: Pool Size: 9
```

示例代码2

```
package executor.example2;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadPoolExecutor;


public class SumTest {

    public static void main(String[] args) {

        // 执行线程池
```

```
        ThreadPoolExecutor executor=
(ThreadPoolExecutor)Executors.newFixedThreadPool(4);

        List<Future<Integer>> resultList=new ArrayList<>();

        //统计1-1000总和，分成10个任务计算，提交任务
        for (int i=0; i<10; i++){
            SumTask calculator=new SumTask(i*100+1, (i+1)*100);
            Future<Integer> result=executor.submit(calculator);
            resultList.add(result);
        }

        // 每隔50毫秒，轮询等待10个任务结束
        do {
            System.out.printf("Main: 已经完成多少个任务:
%d\n",executor.getCompletedTaskCount());
            for (int i=0; i<resultList.size(); i++) {
                Future<Integer> result=resultList.get(i);
                System.out.printf("Main: Task %d: %s\n",i,result.isDone());
            }
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } while (executor.getCompletedTaskCount()<resultList.size());

        // 所有任务都已经结束了，综合计算结果
        int total = 0;
        for (int i=0; i<resultList.size(); i++) {
            Future<Integer> result=resultList.get(i);
            Integer sum=null;
            try {
                sum=result.get();
                total = total + sum;
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }
        System.out.printf("1-1000的总和:" + total);

        // 关闭线程池
        executor.shutdown();
    }

}
```

```
package executor.example2;

import java.util.Random;
import java.util.concurrent.Callable;

public class SumTask implements Callable<Integer> {
    //定义每个线程计算的区间
    private int startNumber;
```

```
    private int endNumber;


    public SumTask(int startNumber, int endNumber){
        this.startNumber=startNumber;
        this.endNumber=endNumber;
    }

    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for(int i=startNumber; i<=endNumber; i++)
        {
            sum = sum + i;
        }

        Thread.sleep(new Random().nextInt(1000));

        System.out.printf("%s: %d\n",Thread.currentThread().getName(),sum);
        return sum;
    }

}
```

部分运行结果如下：

```
Main: Task 7: false
Main: Task 8: true
Main: Task 9: true
Main: 已经完成多少个任务: 9
Main: Task 0: true
Main: Task 1: true
Main: Task 2: true
Main: Task 3: true
Main: Task 4: true
Main: Task 5: true
Main: Task 6: true
Main: Task 7: false
Main: Task 8: true
Main: Task 9: true
Main: 已经完成多少个任务: 9
Main: Task 0: true
Main: Task 1: true
Main: Task 2: true
Main: Task 3: true
Main: Task 4: true
Main: Task 5: true
Main: Task 6: true
Main: Task 7: false
Main: Task 8: true
Main: Task 9: true
Main: 已经完成多少个任务: 9
Main: Task 0: true
Main: Task 1: true
Main: Task 2: true
Main: Task 3: true
Main: Task 4: true
Main: Task 5: true
Main: Task 6: true
Main: Task 7: false
Main: Task 8: true
Main: Task 9: true
pool-1-thread-2: 75050
1-1000的总和:500500
```

Java并发框架Fork-Join

- Java 7 提供另一种并行框架：分解、治理、合并(分治编程)
- 适合用于整体任务量不好确定的场合(最小任务可确定)

关键类

ForkJoinPool 任务池
RecursiveAction
RecursiveTask

示例代码

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;

//分任务求和
public class SumTest {


    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        //创建执行线程池
        ForkJoinPool pool = new ForkJoinPool();
        //ForkJoinPool pool = new ForkJoinPool(4);

        //创建任务
        SumTask task = new SumTask(1, 10000000);

        //提交任务
        ForkJoinTask<Long> result = pool.submit(task);

        //等待结果
        do {
            System.out.printf("Main: Thread Count:
%d\n",pool.getActiveThreadCount());
            System.out.printf("Main: Paralelism: %d\n",pool.getParallelism());
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } while (!task.isDone());

        //输出结果
        System.out.println(result.get().toString());
    }

}
```

```
import java.math.BigInteger;
import java.util.concurrent.RecursiveTask;

//分任务求和
public class SumTask extends RecursiveTask<Long> {


    private int start;
    private int end;

    public SumTask(int start, int end) {
        this.start = start;
        this.end = end;
    }

    public static final int threadhold = 5;

    @Override
    protected Long compute() {
```

```
        Long sum = 0L;

        // 如果任务足够小，就直接执行
        boolean canCompute = (end - start) <= threadhold;
        if (canCompute) {
            for (int i = start; i <= end; i++) {
                sum = sum + i;
            }
        } else {
            // 任务大于阈值，分裂为2个任务
            int middle = (start + end) / 2;
            SumTask subTask1 = new SumTask(start, middle);
            SumTask subTask2 = new SumTask(middle + 1, end);

            invokeAll(subTask1, subTask2);

            Long sum1 = subTask1.join();
            Long sum2 = subTask2.join();

            // 结果合并
            sum = sum1 + sum2;
        }
        return sum;
    }

}
```

运行结果如下：

```
Main: Thread Count: 1
Main: Paralelism: 12
Main: Thread Count: 11
Main: Paralelism: 12
50000005000000
```

Java并发数据结构

- 常用的数据结构是线程不安全的
    - ArrayList, HashMap, HashSet 非同步的
    - 多个线程同时读写，可能会抛出异常或数据错误
- 传统Vector，Hashtable等同步集合性能过差
- 并发数据结构：数据添加和删除
    - 阻塞式集合：当集合为空或者满时，等待
    - 非阻塞式集合：当集合为空或者满时，不等待，返回null或异常

List

- Vector 同步安全，写多读少
- ArrayList 不安全
- Collections.synchronizedList(List list) 基于synchronized，效率差

> synchronized所包围的代码一次只能由一个线程来执行

- CopyOnWriteArrayList 读多写少，基于复制机制，非阻塞

示例代码

```
package list;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class ListTest {

    public static void main(String[] args) throws InterruptedException{

        //线程不安全
        List<String> unsafeList = new ArrayList<String>();
        //线程安全
        List<String> safeList1 = Collections.synchronizedList(new
ArrayList<String>());
        //线程安全
        CopyOnWriteArrayList<String> safeList2 = new
CopyOnWriteArrayList<String>();

        ListThread t1 = new ListThread(unsafeList);
        ListThread t2 = new ListThread(safeList1);
        ListThread t3 = new ListThread(safeList2);

        for(int i = 0; i < 10; i++){
            Thread t = new Thread(t1, String.valueOf(i));
            t.start();
        }
        for(int i = 0; i < 10; i++) {
            Thread t = new Thread(t2, String.valueOf(i));
            t.start();
        }
        for(int i = 0; i < 10; i++) {
            Thread t = new Thread(t3, String.valueOf(i));
            t.start();
        }

        //等待子线程执行完
        Thread.sleep(2000);

        System.out.println("listThread1.list.size() = " + t1.list.size());
        System.out.println("listThread2.list.size() = " + t2.list.size());
        System.out.println("listThread3.list.size() = " + t3.list.size());
```

```
        //输出list中的值
        System.out.println("unsafeList: ");
        for(String s : t1.list){
            if(s == null){
                System.out.print("null  ");
            }
            else
            {
                System.out.print(s + "  ");
            }
        }
        System.out.println();
        System.out.println("safeList1: ");
        for(String s : t2.list){
            if(s == null){
                System.out.print("null  ");
            }
            else
            {
                System.out.print(s + "  ");
            }
        }
        System.out.println();
        System.out.println("safeList2: ");
        for(String s : t3.list){
            if(s == null){
                System.out.print("null  ");
            }
            else
            {
                System.out.print(s + "  ");
            }
        }
    }

}

class ListThread implements Runnable{
    public List<String> list;

    public ListThread(List<String> list){
        this.list = list;
    }

    @Override
    public void run() {
        int i = 0;
        while(i<10)
        {
            try {
                Thread.sleep(10);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
            //把当前线程名称加入list中
            list.add(Thread.currentThread().getName());
            i++;
        }
```

```
    }

}
```

运行结果如下：

```
listThread1.list.size() = 69
listThread2.list.size() = 100
listThread3.list.size() = 100
unsafeList:
1 6 5 3 8 7 9 6 2 0 8 2 5 4 8 7 9 2 6 4 1 8 7 1 4 3 2 0 8 9 7 0 2 8 9 2 5 3 0 1 4 8
safeList1:
6 0 5 4 9 3 1 2 7 8 8 2 5 4 1 3 7 6 0 9 5 6 8 4 9 3 0 1 7 2 6 5 1 9 3 4 7 8 2 0 6 5
safeList2:
0 1 2 4 3 8 6 7 9 5 2 4 1 3 0 5 6 9 7 8 0 2 1 3 4 5 6 7 8 9 1 0 4 2 3 5 8 6 9 7 0 1
```

Set

- HashSet 不安全
- Collections.synchronizedSet(Set set) 基于synchronized，效率差
- CopyOnWriteArraySet (基于CopyOnWriteArrayList实现) 读多写少，非阻塞

示例代码

```
package set;

import java.util.*;
import java.util.concurrent.CopyOnWriteArraySet;

public class SetTest{

    public static void main(String[] args) throws InterruptedException{

        //线程不安全
        Set<String> unsafeSet = new HashSet<String>();
        //线程安全
        Set<String> safeSet1 = Collections.synchronizedSet(new HashSet<String>
());
        //线程安全
        CopyOnWriteArraySet<String> safeSet2 = new CopyOnWriteArraySet<String>
();

        SetThread t1 = new SetThread(unsafeSet);
        SetThread t2 = new SetThread(safeSet1);
        SetThread t3 = new SetThread(safeSet2);

        //unsafeSet的运行测试
        for(int i = 0; i < 10; i++){
            Thread t = new Thread(t1, String.valueOf(i));
            t.start();
        }
        for(int i = 0; i < 10; i++) {
            Thread t = new Thread(t2, String.valueOf(i));
            t.start();
        }
        for(int i = 0; i < 10; i++) {
            Thread t = new Thread(t3, String.valueOf(i));
            t.start();
        }
```

```
        //等待子线程执行完
        Thread.sleep(2000);

        System.out.println("setThread1.set.size() = " + t1.set.size());
        System.out.println("setThread2.set.size() = " + t2.set.size());
        System.out.println("setThread3.set.size() = " + t3.set.size());

        //输出set中的值
        System.out.println("unsafeSet: ");
        for(String element:t1.set){
            if(element == null){
                System.out.print("null  ");
            }
            else
            {
                System.out.print(element + "  ");
            }
        }
        System.out.println();
        System.out.println("safeSet1: ");
        for(String element:t2.set){
            if(element == null){
                System.out.print("null  ");
            }
            else
            {
                System.out.print(element + "  ");
            }
        }
        System.out.println();
        System.out.println("safeSet2: ");
        for(String element:t3.set){
            if(element == null){
                System.out.print("null  ");
            }
            else
            {
                System.out.print(element + "  ");
            }
        }
    }

}

class SetThread implements Runnable{
    public Set<String> set;

    public SetThread(Set<String> set){
        this.set = set;
    }

    @Override
    public void run() {
        int i = 0;
        while(i<10)
        {
            i++;
```

```
            try {
                Thread.sleep(10);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
            //把当前线程名称加入list中
            set.add(Thread.currentThread().getName() + i);
        }
    }

}
```

输出结果如下：

```
setThread1.set.size() = 98
setThread2.set.size() = 100
setThread3.set.size() = 100
unsafeSet:
88  01  89  02  03  04  05  06  07  08  09  110  510  91  910  92  93  94  95  96  97  98  11  99  12  13  14  15  16  17  18
safeSet1:
88  01  89  02  03  04  05  06  07  08  09  110  510  91  910  92  93  94  95  96  97  98  11  99  12  13  14  15  16  17  18
safeSet2:
11  01  21  51  61  71  31  91  81  41  02  62  22  52  12  92  72  82  32  42  23  53  63  13  03  33  83  73  93  43  14  04
```

Map

- Hashtable 同步安全，写多读少
- HashMap 不安全
- Collections.synchronizedMap(Map map) 基于synchronized，效率差
- ConcurrentHashMap 读多写少，非阻塞

示例代码

```
package map;

import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class MapTest{

    public static void main(String[] args) throws InterruptedException{

        //线程不安全
        Map<Integer,String> unsafeMap = new HashMap<Integer,String>();
        //线程安全
        Map<Integer,String> safeMap1 = Collections.synchronizedMap(new
HashMap<Integer,String>());
        //线程安全
        ConcurrentHashMap<Integer,String> safeMap2 = new
ConcurrentHashMap<Integer,String>();

        MapThread t1 = new MapThread(unsafeMap);
        MapThread t2 = new MapThread(safeMap1);
        MapThread t3 = new MapThread(safeMap2);

        //unsafeMap的运行测试
        for(int i = 0; i < 10; i++){
            Thread t = new Thread(t1);
```

```
            t.start();
        }
        for(int i = 0; i < 10; i++) {
            Thread t = new Thread(t2);
            t.start();
        }
        for(int i = 0; i < 10; i++) {
            Thread t = new Thread(t3);
            t.start();
        }

        //等待子线程执行完
        Thread.sleep(2000);

        System.out.println("mapThread1.map.size() = " + t1.map.size());
        System.out.println("mapThread2.map.size() = " + t2.map.size());
        System.out.println("mapThread3.map.size() = " + t3.map.size());

        //输出set中的值
        System.out.println("unsafeMap: ");
        Iterator iter = t1.map.entrySet().iterator();
        while(iter.hasNext()) {
            Map.Entry<Integer,String> entry =
(Map.Entry<Integer,String>)iter.next();
            // 获取key
            System.out.print(entry.getKey() + ":");
            // 获取value
            System.out.print(entry.getValue() + " ");
        }
        System.out.println();

        System.out.println("safeMap1: ");
        iter = t2.map.entrySet().iterator();
        while(iter.hasNext()) {
            Map.Entry<Integer,String> entry =
(Map.Entry<Integer,String>)iter.next();
            // 获取key
            System.out.print(entry.getKey() + ":");
            // 获取value
            System.out.print(entry.getValue() + " ");
        }

        System.out.println();
        System.out.println("safeMap2: ");
        iter = t3.map.entrySet().iterator();
        while(iter.hasNext()) {
            Map.Entry<Integer,String> entry =
(Map.Entry<Integer,String>)iter.next();
            // 获取key
            System.out.print(entry.getKey() + ":");
            // 获取value
            System.out.print(entry.getValue() + " ");
        }
        System.out.println();
        System.out.println("mapThread1.map.size() = " + t1.map.size());
        System.out.println("mapThread2.map.size() = " + t2.map.size());
        System.out.println("mapThread3.map.size() = " + t3.map.size());
    }
```

```
}

class MapThread implements Runnable
{
    public Map<Integer,String> map;

    public MapThread(Map<Integer,String> map){
        this.map = map;
    }

    @Override
    public void run() {
        int i=0;

        while(i<100)
        {
            //把当前线程名称加入map中
            map.put(i++,Thread.currentThread().getName());
            try {
                Thread.sleep(10);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }

}
```

运行结果如下：

```
mapThread1.map.size() = 141
mapThread2.map.size() = 100
mapThread3.map.size() = 100
unsafeMap:
2:Thread-4 4:Thread-7 5:Thread-7 7:Thread-4 10:Thread-4 11:Thread-4 12:Thread-4 13:Thread-4 14:Thread-4 16:Thread-4 17:Thread-
safeMap1:
0:Thread-19 1:Thread-10 2:Thread-12 3:Thread-17 4:Thread-17 5:Thread-11 6:Thread-11 7:Thread-11 8:Thread-10 9:Thread-18 10:Thr
safeMap2:
0:Thread-29 1:Thread-28 2:Thread-25 3:Thread-24 4:Thread-29 5:Thread-28 6:Thread-28 7:Thread-25 8:Thread-28 9:Thread-25 10:Thr
mapThread1.map.size() = 141
mapThread2.map.size() = 100
mapThread3.map.size() = 100
```

Queue & Deque

- ConcurrentLinkedQueue 非阻塞
- ArrayBlockingQueue/LinkedBlockingQueue 阻塞

示例代码

```
package queue;

import java.util.*;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ConcurrentLinkedDeque;

public class QueueTest {

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

```
        //线程不安全
        Deque<String> unsafeQueue = new ArrayDeque<String>();
        //线程安全
        ConcurrentLinkedDeque<String> safeQueue1 = new
ConcurrentLinkedDeque<String>();

        ArrayBlockingQueue<String> safeQueue2 = new ArrayBlockingQueue<String>
(100);

        QueueThread t1 = new QueueThread(unsafeQueue);
        QueueThread t2 = new QueueThread(safeQueue1);
        QueueThread t3 = new QueueThread(safeQueue2);

        for(int i = 0; i < 10; i++){
            Thread thread1 = new Thread(t1, String.valueOf(i));
            thread1.start();
        }
        for(int i = 0; i < 10; i++) {
            Thread thread2 = new Thread(t2, String.valueOf(i));
            thread2.start();
        }
        for(int i = 0; i < 10; i++) {
            Thread thread3 = new Thread(t3, String.valueOf(i));
            thread3.start();
        }

        //等待子线程执行完
        Thread.sleep(2000);

        System.out.println("queueThread1.queue.size() = " + t1.queue.size());
        System.out.println("queueThread2.queue.size() = " + t2.queue.size());
        System.out.println("queueThread3.queue.size() = " + t3.queue.size());

        //输出queue中的值
        System.out.println("unsafeQueue: ");
        for(String s:t1.queue)
        {
            System.out.print(s + " ");
        }
        System.out.println();
        System.out.println("safeQueue1: ");
        for(String s:t2.queue)
        {
            System.out.print(s + " ");
        }
        System.out.println();
        System.out.println("safeQueue2: ");
        for(String s:t3.queue)
        {
            System.out.print(s + " ");
        }
    }

}

class QueueThread implements Runnable{
    public Queue<String> queue;
```

```
    public QueueThread(Queue<String> queue){
        this.queue = queue;
    }

    @Override
    public void run() {
        int i = 0;
        while(i<10)
        {
            i++;
            try {
                Thread.sleep(10);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
            //把当前线程名称加入list中
            queue.add(Thread.currentThread().getName());
        }
    }

}
```

运行结果如下：

```
queueThread1.queue.size() = 86
queueThread2.queue.size() = 100
queueThread3.queue.size() = 100
Exception in thread "main" java.util.ConcurrentModificationException
        at java.base/java.util.ArrayDeque.nonNullElementAt(ArrayDeque.java:270)
        at java.base/java.util.ArrayDeque$DeqIterator.next(ArrayDeque.java:700)
        at queue.QueueTest.main(QueueTest.java:44)
unsafeQueue:
0 8 7 6 5 4 3 2 1 9 5 0 6 3 1 2 4 8
```

Java并发协作控制

线程协作

- Thread/Executor/Fork-Join
  - 线程启动，运行，结束
  - 线程之间缺少协作
- synchronized 同步
  - 限定只有一个线程才能进入关键区
  - 简单粗暴，性能损失有点大

Lock

- Lock也可以实现同步的效果
  - 实现更复杂的临界区结构
  - tryLock方法可以预判锁是否空闲
  - 允许分离读写的操作，多个读，一个写
  - 性能更好
- ReentrantLock类，可重入的互斥锁

- ReentrantReadWriteLock类，可重入的读写锁
- lock和unlock函数

示例代码

> 有家奶茶店，点单有时需要排队
> 假设想买奶茶的人如果看到需要排队，就决定不买
> 又假设奶茶店有老板和多名员工，记单方式比较原始，只有一个订单本
> 老板负责写新订单，员工不断地查看订单本得到信息来制作奶茶，在老板写新订单时员工不能看订单本
> 多个员工可同时看订单本，在员工看时老板不能写新订单

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;public class
LockExample {

    private static final ReentrantLock queueLock = new ReentrantLock(); //可重入锁
    private static final ReentrantReadWriteLock orderLock = new
ReentrantReadWriteLock(); //可重入读写锁

    /**
     * 有家奶茶店，点单有时需要排队
     * 假设想买奶茶的人如果看到需要排队，就决定不买
     * 又假设奶茶店有老板和多名员工，记单方式比较原始，只有一个订单本
     * 老板负责写新订单，员工不断地查看订单本得到信息来制作奶茶，在老板写新订单时员工不能看订单
本
     * 多个员工可同时看订单本，在员工看时老板不能写新订单
     * @param args
     * @throws InterruptedException
     */
    public static void main(String[] args) throws InterruptedException {
        //buyMilkTea();
        handleOrder(); //需手动关闭
    }

    public void tryToBuyMilkTea() throws InterruptedException {
        boolean flag = true;
        while(flag)
        {
            if (queueLock.tryLock()) {
                //queueLock.lock();
                long thinkingTime = (long) (Math.random() * 500);
                Thread.sleep(thinkingTime);
                System.out.println(Thread.currentThread().getName() + ":  来一杯珍
珠奶茶，不要珍珠");
                flag = false;
                queueLock.unlock();
            } else {
                //System.out.println(Thread.currentThread().getName() + ": " +
queueLock.getQueueLength() + "人在排队");
                System.out.println(Thread.currentThread().getName() + ":  再等
等");
            }
```

```
            if(flag)
            {
                Thread.sleep(1000);
            }
        }

    }

    public void addOrder() throws InterruptedException {
        orderLock.writeLock().lock();
        long writingTime = (long) (Math.random() * 1000);
        Thread.sleep(writingTime);
        System.out.println("老板新加一笔订单");
        orderLock.writeLock().unlock();
    }

    public void viewOrder() throws InterruptedException {
        orderLock.readLock().lock();

        long readingTime = (long) (Math.random() * 500);
        Thread.sleep(readingTime);
        System.out.println(Thread.currentThread().getName() + ": 查看订单本");
        orderLock.readLock().unlock();

    }

    public static void buyMilkTea() throws InterruptedException {
        LockExample lockExample = new LockExample();
        int STUDENTS_CNT = 10;

        Thread[] students = new Thread[STUDENTS_CNT];
        for (int i = 0; i < STUDENTS_CNT; i++) {
            students[i] = new Thread(new Runnable() {

                @Override
                public void run() {
                    try {
                        long walkingTime = (long) (Math.random() * 1000);
                        Thread.sleep(walkingTime);
                        lockExample.tryToBuyMilkTea();
                    } catch(InterruptedException e) {
                        System.out.println(e.getMessage());
                    }
                }

            }
            );

            students[i].start();
        }

        for (int i = 0; i < STUDENTS_CNT; i++)
            students[i].join();

    }
```

```
    public static void handleOrder() throws InterruptedException {
        LockExample lockExample = new LockExample();


•


        Thread boss = new Thread(new Runnable() {

            @Override
            public void run() {
                while (true) {
                    try {
                        lockExample.addOrder();
                        long waitingTime = (long) (Math.random() * 1000);
                        Thread.sleep(waitingTime);
                    } catch (InterruptedException e) {
                        System.out.println(e.getMessage());
                    }
                }
            }
        });
        boss.start();

        int workerCnt = 3;
        Thread[] workers = new Thread[workerCnt];
        for (int i = 0; i < workerCnt; i++)
        {
            workers[i] = new Thread(new Runnable() {

                @Override
                public void run() {
                    while (true) {
                        try {
                                lockExample.viewOrder();
                                long workingTime = (long) (Math.random() *
5000);
                                Thread.sleep(workingTime);
                            } catch (InterruptedException e) {
                                System.out.println(e.getMessage());
                            }
                        }
                }

            });

            workers[i].start();
        }

    }

}
```

> readLock，读锁
> 可以多个线程共享
> writeLock，写锁
> 排他的，只能一个线程拥有

运行结果如下：

buyMilkTea()

```
Thread-9：来一杯珍珠奶茶，不要珍珠
Thread-1：来一杯珍珠奶茶，不要珍珠
Thread-3：再等等
Thread-7：再等等
Thread-0：来一杯珍珠奶茶，不要珍珠
Thread-4：再等等
Thread-6：再等等
Thread-5：来一杯珍珠奶茶，不要珍珠
Thread-8：来一杯珍珠奶茶，不要珍珠
Thread-2：来一杯珍珠奶茶，不要珍珠
Thread-7：再等等
Thread-3：来一杯珍珠奶茶，不要珍珠
Thread-4：来一杯珍珠奶茶，不要珍珠
```

handleOrder()

```
Thread-3: 查看订单本
老板新加一笔订单
Thread-3: 查看订单本
老板新加一笔订单
Thread-2: 查看订单本
Thread-1: 查看订单本
老板新加一笔订单
Thread-2: 查看订单本
老板新加一笔订单
Thread-2: 查看订单本
老板新加一笔订单
Thread-1: 查看订单本
Thread-2: 查看订单本
Thread-3: 查看订单本
老板新加一笔订单
老板新加一笔订单
Thread-1: 查看订单本
老板新加一笔订单
Thread-1: 查看订单本
Thread-2: 查看订单本
老板新加一笔订单
Thread-3: 查看订单本
老板新加一笔订单
Thread-2: 查看订单本
老板新加一笔订单
Thread-3: 查看订单本
```

Semaphore

- 信号量，由1965年Dijkstra提出的
- 信号量：本质上是一个计数器
- 计数器大于0，可以使用，等于0不能使用
- 可以设置多个并发量，例如限制10个访问
- Semaphore
  - acquire获取
  - release释放
- 比Lock更进一步，可以控制多个同时访问关键区

示例代码

> 现有一地下车库，共有车位5个，由10辆车需要停放，每次停放时，去申请信号量

```
import java.util.concurrent.Semaphore;

public class SemaphoreExample {

    private final Semaphore placeSemaphore = new Semaphore(5);

    public boolean parking() throws InterruptedException {
        if (placeSemaphore.tryAcquire()) {
            System.out.println(Thread.currentThread().getName() + ": 停车成功");
            return true;
        } else {
            System.out.println(Thread.currentThread().getName() + ": 没有空位");
            return false;
        }

    }

    public void leaving() throws InterruptedException {
        placeSemaphore.release();
        System.out.println(Thread.currentThread().getName() + ": 开走");
    }

    /**
     * 现有一地下车库，共有车位5个，由10辆车需要停放，每次停放时，去申请信号量
     * @param args
     * @throws InterruptedException
     */
    public static void main(String[] args) throws InterruptedException {
        int tryToParkCnt = 10;

        SemaphoreExample semaphoreExample = new SemaphoreExample();

        Thread[] parkers = new Thread[tryToParkCnt];

        for (int i = 0; i < tryToParkCnt; i++) {
            parkers[i] = new Thread(new Runnable() {

                @Override
```

```
                public void run() {
                    try {
                        long randomTime = (long) (Math.random() * 1000);
                        Thread.sleep(randomTime);
                        if (semaphoreExample.parking()) {
                            long parkingTime = (long) (Math.random() * 1200);
                            Thread.sleep(parkingTime);
                            semaphoreExample.leaving();
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });

            parkers[i].start();
        }

        for (int i = 0; i < tryToParkCnt; i++) {
            parkers[i].join();
        }
    }

}
```

输出结果如下：

```
Thread-2: 停车成功
Thread-5: 停车成功
Thread-7: 停车成功
Thread-0: 停车成功
Thread-1: 停车成功
Thread-2: 开走
Thread-8: 停车成功
Thread-4: 没有空位
Thread-6: 没有空位
Thread-9: 没有空位
Thread-7: 开走
Thread-3: 停车成功
Thread-8: 开走
Thread-5: 开走
Thread-1: 开走
Thread-0: 开走
Thread-3: 开走
```

Latch

- 等待锁，是一个同步辅助类
- 用来同步执行任务的一个或者多个线程
- 不是用来保护临界区或者共享资源
- CountDownLatch

- countDown() 计数减1
- await() 等待latch变成0

> 设想百米赛跑比赛 发令枪发出信号后选手开始跑，全部选手跑到终点后比赛结束

示例代码

```
import java.util.concurrent.CountDownLatch;

public class CountDownLatchExample {

    /**
     * 设想百米赛跑比赛 发令枪发出信号后选手开始跑，全部选手跑到终点后比赛结束
     *
     * @param args
     * @throws InterruptedException
     */
    public static void main(String[] args) throws InterruptedException {
        int runnerCnt = 10;
        CountDownLatch startSignal = new CountDownLatch(1);
        CountDownLatch doneSignal = new CountDownLatch(runnerCnt);

        for (int i = 0; i < runnerCnt; ++i) // create and start threads
            new Thread(new Worker(startSignal, doneSignal)).start();

        System.out.println("准备工作...");
        System.out.println("准备工作就绪");
        startSignal.countDown(); // let all threads proceed
        System.out.println("比赛开始");
        doneSignal.await(); // wait for all to finish
        System.out.println("比赛结束");
    }

    static class Worker implements Runnable {
        private final CountDownLatch startSignal;
        private final CountDownLatch doneSignal;

        Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
            this.startSignal = startSignal;
            this.doneSignal = doneSignal;
        }

        public void run() {
            try {
                startSignal.await();
                doWork();
                doneSignal.countDown();
            } catch (InterruptedException ex) {
            } // return;
        }

        void doWork() {
            System.out.println(Thread.currentThread().getName() + ": 跑完全程");
        }
    }
```

```
}
```

> Latch变成0以后将唤醒所有在此Latch上await的线程，解锁它们的await等待。

运行结果如下：

```
准备工作...
准备工作就绪
比赛开始
Thread-9: 跑完全程
Thread-2: 跑完全程
Thread-7: 跑完全程
Thread-8: 跑完全程
Thread-4: 跑完全程
Thread-0: 跑完全程
Thread-1: 跑完全程
Thread-5: 跑完全程
Thread-6: 跑完全程
Thread-3: 跑完全程
比赛结束
```

Barrier

- 集合点，也是一个同步辅助类
- 允许多个线程在某一个点上进行同步
- CyclicBarrier
  - 构造函数是需要同步的线程数量
  - await等待其他线程，到达数量后，就放行

示例代码

> 假定有三行数，用三个线程分别计算每一行的和，最终计算总和

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierExample {


    /**
     * 假定有三行数，用三个线程分别计算每一行的和，最终计算总和
     * @param args
     */
    public static void main(String[] args) {
        final int[][] numbers = new int[3][5];
        final int[] results = new int[3];
        int[] row1 = new int[]{1, 2, 3, 4, 5};
        int[] row2 = new int[]{6, 7, 8, 9, 10};
        int[] row3 = new int[]{11, 12, 13, 14, 15};
        numbers[0] = row1;
        numbers[1] = row2;
        numbers[2] = row3;
```

```
        CalculateFinalResult finalResultCalculator = new
CalculateFinalResult(results);
        CyclicBarrier barrier = new CyclicBarrier(3, finalResultCalculator);
        //当有3个线程在barrier上await，就执行finalResultCalculator

        for(int i = 0; i < 3; i++) {
            CalculateEachRow rowCalculator = new CalculateEachRow(barrier,
numbers, i, results);
            new Thread(rowCalculator).start();
        }
    }

}

class CalculateEachRow implements Runnable {

    final int[][] numbers;
    final int rowNumber;
    final int[] res;
    final CyclicBarrier barrier;

    CalculateEachRow(CyclicBarrier barrier, int[][] numbers, int rowNumber,
int[] res) {
        this.barrier = barrier;
        this.numbers = numbers;
        this.rowNumber = rowNumber;
        this.res = res;
    }

    @Override
    public void run() {
        int[] row = numbers[rowNumber];
        int sum = 0;
        for (int data : row) {
            sum += data;
            res[rowNumber] = sum;
        }
        try {
            System.out.println(Thread.currentThread().getName() + ": 计算第" +
(rowNumber + 1) + "行结束，结果为: " + sum);
            barrier.await(); //等待！只要超过3个(Barrier的构造参数)，就放行。
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }

}


class CalculateFinalResult implements Runnable {
    final int[] eachRowRes;
    int finalRes;
    public int getFinalResult() {
        return finalRes;
    }


    CalculateFinalResult(int[] eachRowRes) {
```

```
        this.eachRowRes = eachRowRes;
    }

    @Override
    public void run() {
        int sum = 0;
        for(int data : eachRowRes) {
            sum += data;
        }
        finalRes = sum;
        System.out.println("最终结果为: " + finalRes);
    }

}
```

> 当在Barrier上await的线程数量达到预定的要求后，所有的await的线程不再等待，全部解锁。而且Barrier将执行预定的回调动作（在本程序中，回调动作就是CalculateFinalResult）。

运行结果如下：

```
Thread-0: 计算第1行结束，结果为: 15
Thread-2: 计算第3行结束，结果为: 65
Thread-1: 计算第2行结束，结果为: 40
最终结果为: 120
```

Phaser

- 允许执行并发多阶段任务，同步辅助类
- 在每一个阶段结束的位置对线程进行同步，当所有的线程都到达这步，再进行下一步
- Phaser
  - arrive()
  - arriveAndAwaitAdvance()

示例代码

> 假设举行考试，总共三道大题，每次下发一道题目，等所有学生完成后再进行下一道

```
import java.util.concurrent.Phaser;

public class PhaserExample {

    /**
     * 假设举行考试，总共三道大题，每次下发一道题目，等所有学生完成后再进行下一道
     *
     * @param args
     */
    public static void main(String[] args) {

        int studentsCnt = 5;
        Phaser phaser = new Phaser(studentsCnt);

        for (int i = 0; i < studentsCnt; i++) {
            new Thread(new Student(phaser)).start();
```

```
        }
    }

}

class Student implements Runnable {

    private final Phaser phaser;

    public Student(Phaser phaser) {
        this.phaser = phaser;
    }

    @Override
    public void run() {
        try {
            doTesting(1);
            phaser.arriveAndAwaitAdvance(); //等到5个线程都到了，才放行
            doTesting(2);
            phaser.arriveAndAwaitAdvance();
            doTesting(3);
            phaser.arriveAndAwaitAdvance();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void doTesting(int i) throws InterruptedException {
        String name = Thread.currentThread().getName();
        System.out.println(name + "开始答第" + i + "题");
        long thinkingTime = (long) (Math.random() * 1000);
        Thread.sleep(thinkingTime);
        System.out.println(name + "第" + i + "道题答题结束");
    }

}
```

运行结果如下：

```
Thread-3开始答第1题
Thread-4开始答第1题
Thread-2开始答第1题
Thread-0开始答第1题
Thread-1开始答第1题
Thread-3第1道题答题结束
Thread-1第1道题答题结束
Thread-0第1道题答题结束
Thread-4第1道题答题结束
Thread-2第1道题答题结束
Thread-2开始答第2题
Thread-3开始答第2题
Thread-1开始答第2题
Thread-0开始答第2题
Thread-4开始答第2题
Thread-3第2道题答题结束
Thread-4第2道题答题结束
Thread-1第2道题答题结束
Thread-0第2道题答题结束
Thread-2第2道题答题结束
Thread-2开始答第3题
Thread-3开始答第3题
Thread-4开始答第3题
Thread-0开始答第3题
Thread-1开始答第3题
Thread-4第3道题答题结束
Thread-1第3道题答题结束
Thread-2第3道题答题结束
Thread-0第3道题答题结束
Thread-3第3道题答题结束
```

Exchanger

- 允许在并发线程中互相交换消息
- 允许在2个线程中定义同步点，当两个线程都到达同步点，它们交换数据结构
- Exchanger
    - exchange(), 线程双方互相交互数据
    - 交换数据是双向的

示例代码

> 本例通过Exchanger实现学生成绩查询，简单线程间数据的交换

```java
import java.util.Scanner;
import java.util.concurrent.Exchanger;

public class ExchangerExample {


    /**
     * 本例通过Exchanger实现学生成绩查询，简单线程间数据的交换
     * @param args
     * @throws InterruptedException
```

```
     */
    public static void main(String[] args) throws InterruptedException {
        Exchanger<String> exchanger = new Exchanger<String>();
        BackgroundWorker worker = new BackgroundWorker(exchanger);
        new Thread(worker).start();

        Scanner scanner = new Scanner(System.in);
        while(true) {
            System.out.println("输入要查询的属性学生姓名：");
            String input = scanner.nextLine().trim();
            exchanger.exchange(input); //把用户输入传递给线程
            String value = exchanger.exchange(null); //拿到线程反馈结果
            if ("exit".equals(value)) {
                break;
            }
            System.out.println("查询结果：" + value);
        }
        scanner.close();
    }

}

class BackgroundWorker implements Runnable {

    final Exchanger<String> exchanger;
    BackgroundWorker(Exchanger<String> exchanger) {
        this.exchanger = exchanger;
    }
    @Override
    public void run() {
        while (true) {
            try {
                String item = exchanger.exchange(null);
                switch (item) {
                case "zhangsan":
                    exchanger.exchange("90");
                    break;
                case "lisi":
                    exchanger.exchange("80");
                    break;
                case "wangwu":
                    exchanger.exchange("70");
                    break;
                case "exit":
                    exchanger.exchange("exit");
                    return;
                default:
                    exchanger.exchange("查无此人");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

> 当两个线程都同时执行到同一个exchanger的exchange方法，两个线程就相互交换数据，交换是双向的。

运行结果如下：

```
输入要查询的属性学生姓名：
zhangsan
查询结果：90
输入要查询的属性学生姓名：
lisi
查询结果：80
输入要查询的属性学生姓名：
wangwu
查询结果：70
输入要查询的属性学生姓名：
zhaosi
查询结果：查无此人
输入要查询的属性学生姓名：
exit
```

Java定时任务执行

定时任务

- Thread/Executor/Fork-Join 多线程
  - 立刻执行
  - 框架调度
- 定时执行
  - 固定某一个时间点运行
  - 以某一个周期

简单定时器机制（Timer）

- 设置计划任务，也就是在指定的时间开始执行某一个任务。
- TimerTask 封装任务
- Timer类 定时器

示例代码

> TimerTask也是继承于Runnable这个接口
> schedule(TimerTask task, long d‘elay) 延迟 delay 毫秒执行
> schedule(TimerTask task, Date time) 特定时间执行
> schedule(TimerTask task, long delay, long period) 延迟 delay 执行并每隔period 执行一次

```
package timer;

import java.util.Calendar;
import java.util.Date;
import java.util.Timer;
import java.util.TimerTask;

public class TimerTest {
    public static void main(String[] args) throws InterruptedException {
            MyTask task = new MyTask();
            Timer timer = new Timer();
```

```
            System.out.println("当前时间："+new Date().toLocaleString());
            //当前时间1秒后，每2秒执行一次
            timer.schedule(task, 1000, 2000);

            Thread.sleep(10000);
            task.cancel();  //取消当前的任务

            System.out.println("================================");

            Calendar now = Calendar.getInstance();
            now.set(Calendar.SECOND,now.get(Calendar.SECOND)+3);
            Date runDate = now.getTime();
            MyTask2 task2 = new MyTask2();
            timer.scheduleAtFixedRate(task2,runDate,3000); //固定速率


•


            Thread.sleep(20000);
            timer.cancel();  //取消定时器
    }

}

class MyTask extends TimerTask {
    public void run() {
        System.out.println("运行了！时间为：" + new Date());
    }
}

class MyTask2 extends TimerTask {
    public void run() {
        System.out.println("运行了！时间为：" + new Date());
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

运行结果如下：

```
当前时间：2020年11月7日 下午7:32:49
运行了！时间为：Sat Nov 07 19:32:50 CST 2020
运行了！时间为：Sat Nov 07 19:32:52 CST 2020
运行了！时间为：Sat Nov 07 19:32:54 CST 2020
运行了！时间为：Sat Nov 07 19:32:56 CST 2020
运行了！时间为：Sat Nov 07 19:32:58 CST 2020
================================
运行了！时间为：Sat Nov 07 19:33:02 CST 2020
运行了！时间为：Sat Nov 07 19:33:06 CST 2020
运行了！时间为：Sat Nov 07 19:33:10 CST 2020
运行了！时间为：Sat Nov 07 19:33:14 CST 2020
运行了！时间为：Sat Nov 07 19:33:18 CST 2020
```

> 一个Timer对象可以执行多个计划任务，但是这些任务是串行执行的。如果有一个任务执行很慢，将会影响后续的任务准点运行。

Executor +定时器机制

ScheduledExecutorService

- 定时任务
- 周期任务

示例代码

```
package schedule;

import java.util.Date;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorTest {


    public static void main(String[] a) throws Exception
    {
        //executeAtFixTime();
        //executeFixedRate();  //3s
        executeFixedDelay();  //4s
    }

    public static void executeAtFixTime() throws Exception {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
        executor.schedule(
                new MyTask(),
                1,
                TimeUnit.SECONDS);

        Thread.sleep(20000);
        executor.shutdown();
    }

    /**
```

```
     * 周期任务 固定速率 是以上一个任务开始的时间计时，period时间过去后，检测上一个任务是否执行完毕，
     * 如果上一个任务执行完毕，则当前任务立即执行，如果上一个任务没有执行完毕，则需要等上一个任务执行完毕后立即执行。
     * @throws Exception
     */
    public static void executeFixedRate() throws Exception {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
        executor.scheduleAtFixedRate(
                new MyTask(),
                1,
                3000,
                TimeUnit.MILLISECONDS);

        Thread.sleep(20000);
        executor.shutdown();
    }

    /**
     * 周期任务 固定延时 是以上一个任务结束时开始计时，period时间过去后，立即执行。
     * @throws Exception
     */
    public static void executeFixedDelay() throws Exception {
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
        executor.scheduleWithFixedDelay(
                new MyTask(),
                1,
                3000,
                TimeUnit.MILLISECONDS);

        Thread.sleep(20000);
        executor.shutdown();
    }

}

class MyTask implements Runnable {
    public void run() {
        System.out.println("时间为: " + new Date());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //System.out.println("时间为: " + new Date());
    }
}
```

运行结果如下：

executeAtFixTime()

```
时间为：Sat Nov 07 19:40:59 CST 2020
```

executeFixedRate()

```
时间为：Sat Nov 07 19:42:39 CST 2020
时间为：Sat Nov 07 19:42:42 CST 2020
时间为：Sat Nov 07 19:42:45 CST 2020
时间为：Sat Nov 07 19:42:48 CST 2020
时间为：Sat Nov 07 19:42:51 CST 2020
时间为：Sat Nov 07 19:42:54 CST 2020
时间为：Sat Nov 07 19:42:57 CST 2020
```

executeFixedDelay()

```
时间为：Sat Nov 07 19:43:24 CST 2020
时间为：Sat Nov 07 19:43:28 CST 2020
时间为：Sat Nov 07 19:43:32 CST 2020
时间为：Sat Nov 07 19:43:36 CST 2020
时间为：Sat Nov 07 19:43:40 CST 2020
```

Quartz

- Quartz是一个较为完善的任务调度框架
- 解决程序中Timer零散管理的问题
- 功能更加强大
- Timer执行周期任务，如果中间某一次有异常，整个任务终止执行
- Quartz执行周期任务，如果中间某一次有异常，不影响下次任务执行

示例代码

```
package quartz;

import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.Trigger;
import org.quartz.impl.StdSchedulerFactory;

import static org.quartz.JobBuilder.newJob;
import static org.quartz.SimpleScheduleBuilder.simpleSchedule;
import static org.quartz.TriggerBuilder.newTrigger;

public class QuartzTest {

    public static void main(String[] args) {
        try {
            //创建scheduler
            Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();

            //定义一个Trigger
            Trigger trigger = newTrigger().withIdentity("trigger1", "group1") //
定义name/group
                    .startNow()//一旦加入scheduler，立即生效
                    .withSchedule(simpleSchedule() //使用SimpleTrigger
                            .withIntervalInSeconds(2) //每隔2秒执行一次
                            .repeatForever()) //一直执行
```

```
                .build();

        //定义一个JobDetail
        JobDetail job = newJob(HelloJob.class) //定义Job类为HelloQuartz类
                .withIdentity("job1", "group1") //定义name/group
                .usingJobData("name", "quartz") //定义属性
                .build();

        //加入这个调度
        scheduler.scheduleJob(job, trigger);

        //启动
        scheduler.start();

        //运行一段时间后关闭
        Thread.sleep(10000);
        scheduler.shutdown(true);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

}

package quartz;

import org.quartz.Job;
import org.quartz.JobDetail;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

import java.util.Date;

public class HelloJob implements Job {
    public void execute(JobExecutionContext context) throws
JobExecutionException {
        JobDetail detail = context.getJobDetail();
        String name = detail.getJobDataMap().getString("name");
        System.out.println("hello from " + name + " at " + new Date());
    }
}
```

运行结果如下：

```
hello from quartz at Sat Nov 07 19:51:03 CST 2020
hello from quartz at Sat Nov 07 19:51:05 CST 2020
hello from quartz at Sat Nov 07 19:51:07 CST 2020
hello from quartz at Sat Nov 07 19:51:09 CST 2020
hello from quartz at Sat Nov 07 19:51:11 CST 2020
hello from quartz at Sat Nov 07 19:51:13 CST 2020
```