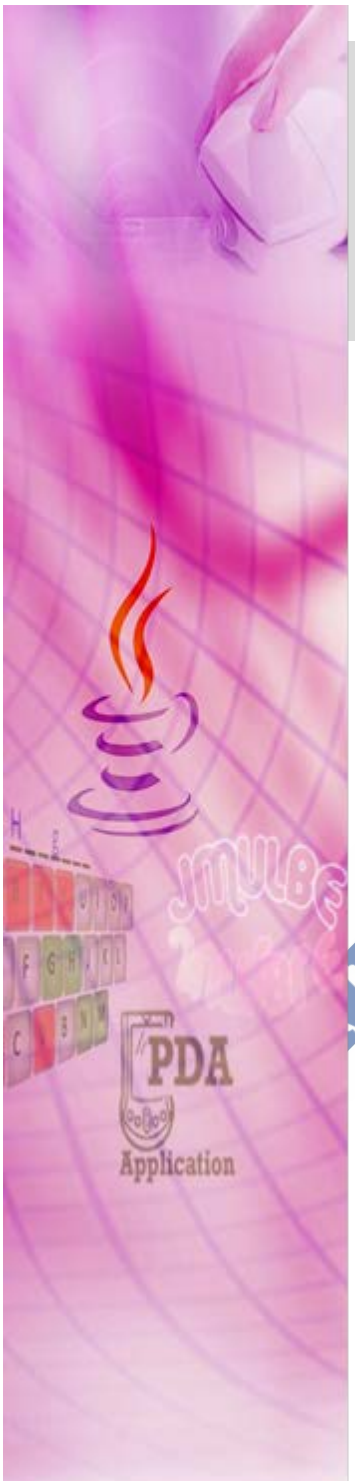


第 6 章

实现线程同步和并发

您已经学习了如何创建多线程 Java 应用程序。此外，还学习了在多线程应用程序中，一个资源可同时被多个线程访问。
然而，可能有这样的情况：您要求一个时间只有一个线程能够访问该资源。为此，Java 提供线程同步。

而且为了提高多线程应用程序的性能，Java 支持并发。
要实现并发，程序被分成若干任务，任务可以并行地执行。

本章重点讲述实现线程同步和并发。

目标

在本章中，您将学习：

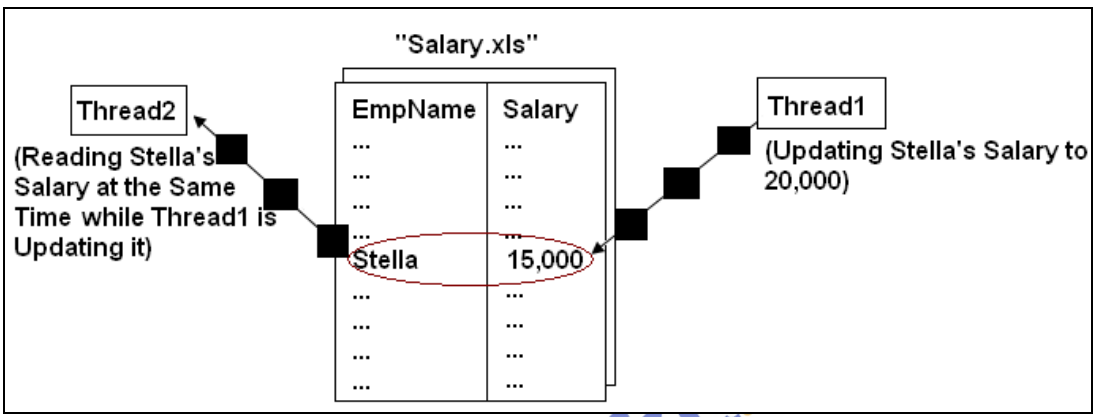
- 实现线程同步
- 实现并发


NIIT
training
china
.com


NIIT
training
china
.com

实现线程同步

在多线程环境中，当两个线程需要共享数据的时候，为了避免错误，您必须尝试协调它们的活动，以确保一个线程不会更改另一个线程使用的数据。例如，如果您有两个线程，一个读取文件中的薪水记录，另一个同时尝试更新薪水记录，那么读薪水记录的线程有可能未获得更新后的值。下图显示了两个线程同时访问一个文件的数据。



两个线程同时访问数据

在上图中，Thread1 正在将 Stella 的薪水更新为 $20,000。同时，另一个线程 Thread2, 正在读取更新前的 Stella 薪水。因此，Thread2 读取的信息是错误的。

为了避免这样的情况，Java 使您能够在给定时间协调和管理多个线程的操作，方法是使用同步线程并实现线程之间的通信。

同步线程

思考银行应用程序的场景：该应用程序允许您访问银行账户并执行存入和取出一定金额之类的操作。此外，这些操作也可使用因特网和自动柜员机 (ATM) 之类的介质执行。在此类场景中，存在这样的可能性：同时对某个账户执行多个操作。这可能导致数据丢失或者数据不一致。因此，为了避免此类问题，需要实现一种机制，通过该机制保证一个时间只能对一个账户执行一个操作。为此，可应用线程同步。

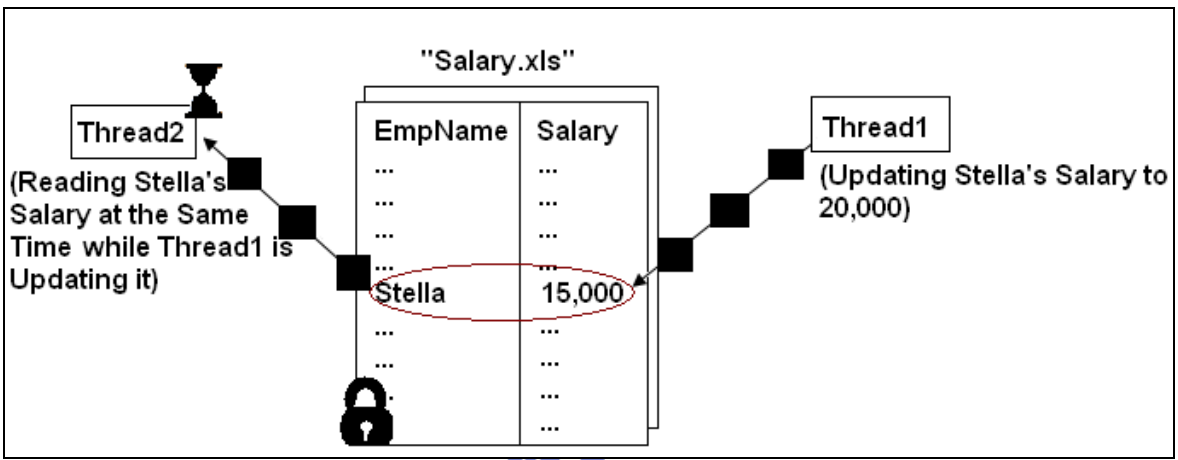
线程的同步确保当两个或多个线程需要访问共享资源时，一次仅一个线程使用该资源。可使用以下方法实现线程同步：

- 使用同步方法
- 使用同步语句

同步方法

可使用 synchronized 关键字同步方法。同步基于监视程序的概念，监视器是一个用作锁的对象。所有 Java 对象都具有一个监视程序，任何给定时间只有一个线程持有对象的监视程序。要进入对象的监督程序，必须调用同步方法。监督程序控制同步方法访问对象或类的方式。当线程获取锁时，就认为它进入了监督程序。监督程序确保在任何给定时间只有一个线程有权访问资源。

当线程在同步方法内的时候，所有试图在同一个时间调用方法的其他线程必须等待。在同步方法的执行过程中，对象被锁定，因此无法调用任何其他同步方法。当方法完成其执行时会自动释放监视程序。下图显示了线程同步的一个示例。



线程同步的示例

在上图中，Thread1 正在更新文件中 Stella 的工资时，在 Salary.xls 文件上有一个锁。Thread2 要从 Salary.xls 文件读取 Stella 的工资，它正在等待 Thread1 释放文件上的锁。当线程在休眠或等待时，它临时释放它持有的锁。

以下代码段演示了如何使用同步方法对以上场景实现同步：

```
class EmployeeDetails
{
// Details of the employee
}
class EmployeeSalaryTransaction
{
public synchronized int processSalary()
{
// Code to process the salary
}
}
```

在上述代码段中，processSalary() 方法是同步方法。此方法不能在一个时间由多个线程访问。

同步语句

线程间的同步还可以通过使用同步语句实现。当类中未使用同步方法且您无权访问源代码时，使用同步语句。然后，如果方法只包含需要同步的一小段关键代码，那么可使用同步语句，而不是同步整个方法。您可以同步对此类的对象的访问，方法是将该对象定义的方法调用放置在 `synchronized` 块中。以下代码段显示了如何使用 `synchronized` 块：

```
synchronized(obj)
{
/* statements to be synchronized*/
}
```

在上述代码段中，`obj` 是对要同步的对象的引用。

以下代码段演示了如何使用同步语句在银行应用程序中实现同步：

```
class EmployeeDetails
{
   // Details of the employee
}
class EmployeeSalaryTransaction
{
public int processSalary()
   {
          synchronized(obj)
       {
          // Code to process the salary
       }
   }
}
```

在上述代码段中，`obj` 引用 `EmployeeDetails` 类的对象。处理工资的代码是在 `synchronized` 块内指定的，以确保它不能在一个时间被多个线程访问。

实现线程间通信

线程可以相互通信。多线程通过称为进程间通信的机制除去了轮询的概念。例如，有两个线程 thread1 和 thread2。thread1 生产的数据由 thread2 使用。thread1 是生产者，thread2 是使用者。使用者必须重复检查生产者是否已生成了数据。这是对 CPU 时间的浪费，因为使用者占用 CPU 时间来检查生产者是否已准备好数据。一个线程在其他线程完成其任务之前不能继续。

线程可以通知其他线程任务已完成。线程间的此通信称为线程间通信。线程间通信所使用的各种方法有：

- `wait()`：通知当前线程离开其对监视程序的控制并等待，直到其他线程调用 `notify()` 方法为止。
- `notify()`：唤醒正等待对象的监视程序得到执行的单个线程。如果多个线程正在等待，那么将随机选择其中之一。
- `notifyAll()`：唤醒正等待对象的监视程序的所有线程。

以下代码显示了生产者-使用者问题，但是不带线程间通信：

```
class SynchronizedMethods
{
int d;
synchronized void getData()
{
        System.out.println("Got data:"+ d);

}
synchronized void putData(int d)
{
        this.d = d;
        System.out.println("Put data:"+ d);
}
}
class Producer extends Thread
{
SynchronizedMethods t;
public Producer(SynchronizedMethods t)
{
   this.t = t;
}
public void run()
{
   int data =700;
   while(true)
{
     System.out.println("Put Called by producer");
     t.putData(data++);
}
}
}
class Consumer extends Thread
{
SynchronizedMethods t;
public Consumer(SynchronizedMethods t)
{
   this.t = t;
}
public void run()
{
while(true)
{
   System.out.println("Get Called by consumer");
   t.getData();
}
}
}
public class ProducerConsumer
{
public static void main(String args[])
{
      SynchronizedMethods obj1 = new SynchronizedMethods();
      Producer p = new Producer(obj1);
```

```
        Consumer c = new Consumer(obj1);
        p.start();
        c.start();
    }
    }
```

上述代码一旦执行就将无限运行下去，如以下输出所示：

```
    .
    .
    .
    .
    Put Called by producer
    Put data:29670
    Put Called by producer
    Put data:29671
    Put Called by producer
    Put data:29672
    Put Called by producer
    Put data:29673
    Put Called by producer
    Put data:29674
    Put Called by producer
    Put data:29675
    Put Called by producer
    Put Called by producer
    .
    .
    .
    .
```

要避免此类错误，请使用 `wait()`、`notify()` 和 `notifyAll()` 方法。所有三种方法都可以从同步方法内调用。

以下代码显示了使用 `wait()` 和 `notify()` 方法的线程间通信：

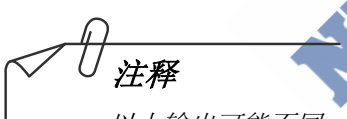
```
    class SynchronizedMethods
    {
    int d;
    boolean flag = false;
    synchronized int getData()
    {
    if(!flag)
    {
    try
    {
    wait();
    }

    catch(InterruptedException e)
    {
    System.out.println(" Exception caught");
    }
    }
    System.out.println("Got data:"+ d);
```

```
flag=false;
notify();
return d;
}
synchronized void putData(int d)
{

if(flag)
{

try
{
wait();
}
catch(InterruptedException e)
{
System.out.println(" Exception caught");
}
}
this.d = d;
flag=true;
System.out.println("Put data with value:"+ d);
notify();

}
}

class Producer implements Runnable
{
SynchronizedMethods t;
public Producer(SynchronizedMethods t)
{
this.t = t;
new Thread(this,"Producer").start();
System.out.println("Put Called by producer");
}
public void run()
{

int data =0;
while(true)
{
data=data+1;
t.putData(data);
}
}
}
class Consumer implements Runnable
{
SynchronizedMethods t;
public Consumer(SynchronizedMethods t)
{
this.t = t;
new Thread(this,"Consumer").start();
```

```
System.out.println("Get Called by consumer");
}
public void run()
{

while(true)
{
t.getData();
}
}
}
public class InterThreadComm
{
public static void main(String args[])
{
SynchronizedMethods obj1 = new SynchronizedMethods();
Producer p = new Producer(obj1);
Consumer c = new Consumer(obj1);

}
}
```

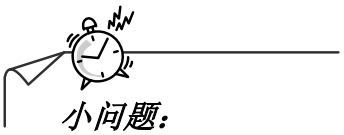
执行上述代码后，会显示以下输出：

```
Put Called by producer
Put data with value:1
Get Called by consumer
Got data:1
Put data with value:2
Got data:2
Put data with value:3
Got data:3
Put data with value:4
Got data:4
```



注释

以上输出可能不同。



小问题：

_____ 方法用于唤醒正等待对象的监视程序的所有线程。

答案：

notifyAll()

活动 6.1：实现同步

实现并发

在多线程环境中，实现同步以消除线程干扰并协调其活动是很重要的。然而，同步并不确保多线程应用程序正在高效率地利用系统资源。应用程序中可能存在这样的情况：多个线程正在等待获得已被锁定的同步方法或代码上的锁。因此，应用程序的效率受到影响。因此，提高多线程应用程序的效率并确保它们尽其所能使用系统的硬件资源（例如处理器），是很重要的。为此，Java 编程语言支持并发。

*并发*使 Java 程序员能够通过创建并发程序提高其应用程序的效率和资源利用率。当今大多数计算机都具有多个处理器以提供更快性能。因此，通过在 Java 应用程序中应用并发，各个程序任务可受管由具有多个处理器的系统同时执行。可通过将程序分为多个线程来实现并发，其中每个线程具有待执行的不同任务。这使处理时间的浪费降到最低，并提高了应用程序的执行速度。

要实现并发，Java 提供 `java.util.concurrent` 包。此包包含可用于实现不同方式以在 Java 应用程序中实现并发和高性能的必要类和接口。

实现原子变量和锁定

`java.util.concurrent` 包具有以下子包以支持并发编程：

- `java.util.concurrent.atomic`
- `java.util.concurrent.locks`

`java.util.concurrent.atomic` 包

思考多线程 Java 应用程序的场景，它处理数据的更新和读取，例如存储在变量中的整数值。您希望确保对数据的任何更改都立即得到更新以使访问这些变量的所有线程都得到最新值。您希望确保对变量的所有更新都成功实现，任何更改都不会因为电力中断之类的问题而丢失。而且您希望即使在发生饥饿等故障的情况下也能成功更新数据。为此，可使用原子变量。*原子变量*是单个变量，在其上执行的自增和自减之类的操作是一步完成的。此类操作称为原子操作。

*原子操作*是作为一个单元执行的操作。此外，原子操作在其过程中不能由任何其他操作停止。因此，当多个线程使用原子变量执行操作时，该操作要么在一个步骤中完全发生，要么根本就不发生。比起同步，原子变量的使用提高了性能，而且有助于避免数据不一致。

`java.util.concurrent.atomic` 包提供了多个类和方法，它们可用于创建原子变量以执行原子操作。

下表列出了 java.util.concurrent.atomic 包的各个类。

类	*描述*
AtomicBoolean	*可以原子方式进行更新的 boolean 值。*
AtomicInteger	*可以原子方式进行更新的 int 值。*
AtomicIntegerArray	*其元素可以原子方式进行更新的 int 数组。*
AtomicLong	*可以原子方式进行更新的 long 值。*
AtomicLongArray	*其元素可以原子方式进行更新的 long 数组。*

java.util.concurrent.atomic 包的类

思考以下代码，它演示原子整数变量的原子操作：

```
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicVariableDemo
{
public static void main(String a[])
{
AtomicInteger value = new AtomicInteger(5);
System.out.println("Intial value:" + value.get());
value.getAndIncrement();
System.out.println("After incrementing, the value:" + value.get());
value.set(40);
System.out.println("After setting, the value:" + value.get());
value.getAndDecrement();
System.out.println("After decrementing, the value:" + value.get());
value.compareAndSet(39, 45);
System.out.println("After updating, the value:" + value.get());
}
}
```

在上述代码中，创建了名为 value、值为 5 的 AtomicInteger。AtomicInteger 类的 get() 方法用于获取 value 变量的当前值。将使用 getAndIncrement() 方法以原子方式将引用变量的当前值自增 1。然后将使用 getAndDecrement() 方法以原子方式将当前值自减 1。compareAndSet() 方法在 value 变量的值为 39 的情况下将其设置为 45。

java.util.concurrent.locks 包

您已学习了如何在多线程应用程序中实现同步。然而，在使用同步线程的应用程序中，如果线程正在等待获得某个对象上的锁，但是锁不能获得，那么线程可能一直等待以获得锁。这可能导致死锁情况。而且，如果有多个线程在等待获得锁，就不可能确定在锁一旦可用的情况下哪个线程将获得它。有可能等待时间最长的那个线程一直不能获得锁。此外，使用同步时，如果线程在执行关键部分时引起了异常，那么锁可能一直不能被释放。这可能使应用程序的性能降级。为了克服以上缺点，Java 在 java.util.concurrent.locks 包中提供 ReentrantLock 类。该类实现

java.io.Serializable 接口。可重入锁用于线程中间的互锁。可重入锁可由同一线程获取任意多次。

此外，此类提供有用的方法，它们可用于以灵活且更有生产力的方式通过锁来执行锁定。锁是用于控制对共享资源访问的工具。

下表列出了 ReentrantLock 类的构造函数。

构造函数	*描述*
ReentrantLock()	*用于创建 ReentrantLock 类的实例。*
ReentrantLock(boolean fairness)	*用于创建 ReentrantLock 类的实例并设置公平性。公平性选项设置为 true 可以确保锁给予等待线程队列中等待时间最长的线程。*

ReentrantLock 类的构造函数

下表列出了 ReentrantLock 类的一些方法。

方法	*描述*
void lock()	*此方法用于获取锁。而且此方法不能被中断。*
boolean tryLock()	*此方法用于仅当锁可用时获取锁。如果锁可用，它将返回 true 值，否则它返回 false 值。*
boolean tryLock(long time, TimeUnit unit)	*此方法用于仅当锁在指定时间段内可用时获取锁。如果锁可用，它将返回 true 值，否则它返回 false 值。*
void unlock()	*此方法用于释放锁。*
boolean isFair()	*如果公平性设置为 true，此方法返回 true。*
boolean isLocked()	*此方法用于查询锁当前是否由任何线程持有。*

ReentrantLock 类的方法

思考显示如何实现可重入锁的以下代码：

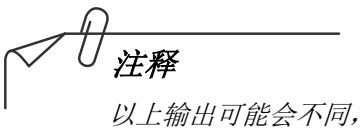
```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockImplementation implements Runnable {
final Lock lock = new ReentrantLock();
String name;
Thread t;
```

```
public void createThreads(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("Creating New Thread:" + t.getName());
t.start();
}

public static void main(String args[]) {
ReentrantLockImplementation obj = new ReentrantLockImplementation();
obj.createThreads("Thread 1");
obj.createThreads("Thread 2");
obj.createThreads("Thread 3");
}

public void run() {
do {
try {
if (lock.tryLock(400, TimeUnit.MILLISECONDS)) {
try {
System.out.println(Thread.currentThread().getName()+" acquired the lock."
);
Thread.sleep(1000);
} finally {
lock.unlock();
System.out.println(Thread.currentThread().getName()+" released the lock."
);
}
break;
}
else
{
System.out.println( Thread.currentThread().getName() + " could not acquire
the lock.Need to try for the lock again.");
}
}
catch (InterruptedException e)
{
System.out.println(e);
}
} while (true);
}
}
```

执行上述代码后，会显示以下输出：

```
Creating New Thread:Thread 1
Creating New Thread:Thread 2
Creating New Thread:Thread 3
Thread 1 acquired the lock.
Thread 3 could not acquire the lock.Need to try for the lock again.
Thread 2 could not acquire the lock.Need to try for the lock again.
Thread 3 could not acquire the lock.Need to try for the lock again.
Thread 2 could not acquire the lock.Need to try for the lock again.
Thread 3 acquired the lock.
```

```
Thread 1 released the lock.
Thread 2 could not acquire the lock.Need to try for the lock again.
Thread 2 could not acquire the lock.Need to try for the lock again.
Thread 2 acquired the lock.
Thread 3 released the lock.
Thread 2 released the lock.
```



注释

以上输出可能会不同，因为无法预测线程执行的顺序。

在上述代码中，创建了三个线程 Thread 1、Thread 2 和 Thread 3。还创建了新的 ReentrantLock lock。然后在执行 run() 方法时，线程在 try 块中获取锁。lock.tryLock(400, TimeUnit.MILLISECONDS) 语句用于检查锁在 400 毫秒的时间段内是否可用。如果是，将获取它，否则线程需要再次尝试获取它。一旦任务完成，线程在 finally 块中释放锁，这确保在线程完成执行时锁将最终被释放。

此外，为了分别对读和写操作实现不同的锁，Java 在 java.util.concurrent.locks 包中提供 ReentrantReadWriteLock 类。此类使您能够对于同一个锁具有不同的读和写版本，方法是在给定时间只有一个线程能够写，而其他线程能够读。ReentrantReadWriteLock 类实现 ReadWriteLock 接口。

下表列出了 ReentrantReadWriteLock 类的构造函数。

构造函数	*描述*
ReentrantReadWriteLock()	*此方法用于创建 ReentrantReadWriteLock 类的实例。*
ReentrantReadWriteLock (boolean fairness)	*此方法用于创建 ReentrantReadWriteLock 类的实例并设置公平性。公平性选项设置为 true 可以确保锁给予等待线程队列中等待时间最长的线程。*

ReentrantReadWriteLock 类的构造函数

下表列出了 ReentrantReadWriteLock 类的一些方法。

方法	*描述*
Lock readLock()	*此方法返回用于读操作的锁。*
Lock writeLock()	*此方法返回用于写操作的锁。*

ReentrantReadWriteLock 类的方法

思考显示如何实现 ReentrantReadWriteLock 的以下代码：

```
import java.util.Random;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class ThreadLock extends Thread {

private static final ReentrantReadWriteLock rwlock = new
ReentrantReadWriteLock();
@Override
public void run()
{
try {
readData();
sleep(2000);
writeData();
} catch (final Exception e)
{
System.out.println(e);
}
}
public void readData()
{
try {
rwlock.readLock().lock();
System.out.println(Thread.currentThread().getName() + " is reading and the
value is 5");
} finally {
System.out.println(Thread.currentThread().getName() + " is exiting after
reading.");
rwlock.readLock().unlock();
}
}

public void writeData()
{
try {
rwlock.writeLock().lock();
Random rand = new Random();
int n = rand.nextInt(50);
System.out.println(Thread.currentThread().getName() + " has the write lock
and is writing.");
System.out.println("The new value is:" + (5 + n));
}
finally
{
System.out.println(Thread.currentThread().getName() + " is releasing the lock
and exiting after writing.");
rwlock.writeLock().unlock();
}
}
}

public class ReentrantLockTest {
public static void main(final String[] args) throws Exception {
ThreadLock obj = new ThreadLock();
```

```
ThreadLock obj2 = new ThreadLock();
obj.start();
obj2.start();
}
}
```

在上述代码中，在 ThreadLock 类中创建了新的 ReentrantReadWriteLock 锁 rwlock，用于扩展 Thread 类。在同一个类中创建了方法 readData()。当线程开始执行此方法时，它在 try 块中获取读操作锁，并在执行任务后在 finally 块中释放它。readData() 方法可由多个线程访问以进行读操作。然后当线程进入 writeData() 方法时，它获取写操作锁，只有获得了写操作锁的线程才能进入 writeData() 方法的关键部分。最后，执行任务后，线程释放写操作锁，以使它能够被另一个等待线程获得。

执行上述代码后，会显示以下输出：

```
Thread-1 is reading and the value is 5
Thread-0 is reading and the value is 5
Thread-1 is exiting after reading.
Thread-0 is exiting after reading.
Thread-1 has the write lock and is writing.
The new value is:9
Thread-1 is releasing the lock and exiting after writing.
Thread-0 has the write lock and is writing.
The new value is:54
Thread-0 is releasing the lock and exiting after writing.
```

注释

以上输出可能会不同，因为无法预测线程执行的顺序。

了解并发同步器

思考多线程应用程序的场景：您希望在应用程序的线程中间实现协调和并发以使数据不一致和错误减到最低。例如，您希望应用程序的多个线程交换数据或者您希望一个或多个线程在应用程序中的另一个线程执行了特定操作之后执行。在此类场景中，可使用 java.util.concurrent 包中提供的 synchronizer 类。提供了以下 synchronizer 类：

- Semaphore
- CountdownLatch
- CyclicBarrier
- Phaser
- Exchanger

Semaphore

semaphore 是具有内部计数器的锁。它用于限制多个线程同时访问共享资源。此外，semaphore 也可用于限制可访问共享资源的线程数目。Semaphore 包含应该由希望访问共享资源的线程获得的许可。一旦线程完成其对共享资源的处理，线程必须释放该许可。acquire() 和 release() 方法分别用于获取和释放锁。如果许可对于某个线程不可用，请求线程必须等待直到可以获得许可。比起 synchronized 块，semaphore 提供公平性能力。semaphore 的公平性能力可用于控制向请求线程给予许可的顺序。如果公平性设置为 true，那么 semaphore 将按照线程请求的顺序给予许可。例如，许可将给予第一个请求许可的线程。如果公平性设置为 false，许可将给予任何一个等待获得许可的线程。使用公平性设置为 true 的 semaphore 有助于避免并发环境中线程中间的饥饿问题。

CountDownLatch

此 synchronizer 类用于一个或多个线程需要等待直到执行了一组操作或事件的情况。并且它还可用于需要通过一组事件或操作的执行来同步多个线程的情况。实现此 synchronizer 类后，将以整数对它进行初始化。此数字是表示操作数目的计数器值，在此数目的操作之后等待线程将开始执行。此计数器值可使用 getCount() 方法检索。需要等待以上操作完成的线程将使用方法 await()。此方法将该线程置于休眠状态，直到那些操作完成。一个操作完成时，将使用另一个方法 countDown() 使计数的数目减小一。最后，当 countdown 达到 0 时，将释放所有休眠线程。

CyclicBarrier

此 synchronizer 类用于多个线程在称为 barrier 的预定义执行点必须等待的情况。为此，使用 await() 方法。线程使用此方法等待彼此到达 barrier。一旦所有线程到达了同一 barrier，任务将被执行。此类实例化后，可指定将参与以达到执行点的线程数目。指定的线程数目不能随时间而变化。此外，在实例化期间，可指定要在达到等待状态时执行的任务。然后，线程可完成其部分并达到同一执行等待点。当所有线程达到执行等待点时，它们可开始处理其任务的下一部分。

Phaser

此 synchronizer 类用于独立线程需要分阶段或步骤运行以完成任务的情况。此类提供的功能类似于 CyclicBarrier 类中所提供的功能。然而，使用此类时，需要使用 Phaser 同步的线程数目可动态更改。要使用 Phaser 类实现同步，将创建 Phaser 的对象。Phaser 对象创建时可带有线程，也可以不带任何线程。然后，register() 方法用于注册每个线程，arriveAndDeregister() 方法用于取消注册已注册的线程。bulkRegister() 方法也可用于批量注册多个线程。arriveAndAwaitAdvance() 方法可用于通知每个线程等待直到其他线程达到执行点。一旦所有线程达到执行点，可使用 onAdvance() 方法指定要执行的任务。

Exchanger

此 synchronizer 类用于需要在两个并发运行任务之间执行同步和数据交换的情况。此类允许两个线程等待公共交换点。如果一个线程较早地到达了交换点，它将等待另一个线程到达此公共交换点。当两个线程都到达了此公共点时，它们将交换共享数据。用于在线程之间交换数据的方法称为 exchange()。当线程准备好交换数据时，它调用 exchange() 方法。如果另一个线程未就绪，调用 exchange 方法的线程将被置于休眠状态，直到另一个线程准备好进行数据交换。

了解并发集合

思考以下场景：您需要创建并发程序以实现存储和管理数据的集合。集合框架中的一些类（例如 ArrayList）不能用于处理多个线程，因为无法控制并发程序的数据访问。例如在某个程序中，如果多个线程对未同步对象集合（例如 ArrayList）执行并发任务，会导致数据不一致错误，该错误进而又会影响应用程序的所需功能。为了避免此情况，Java 在 java.util.concurrent 包中提供并发 Collection 类。使用 Collection 类通过允许多个线程共享数据集合提高并发应用程序的性能和可伸缩性。java.util.concurrent 包中提供的一些可用于并发编程的并发集合如下：

- ArrayBlockingQueue：这是基于 FIFO 的阻塞队列，其中元素是通过数组维护的。在 FIFO 队列中，新元素插入到队列的末尾，将从队列头部除去元素。可在创建此类实例时指定可存储在队列中的元素数目容量。而且，可存储在队列中的元素数目存在上限。此上限只能在实例化时指定，不能在以后更改。如果线程尝试在此队列已满时向其添加元素，线程将被阻塞，直到从队列中除去了某个现有元素。
- LinkedBlockingQueue：这是基于 FIFO 的队列，其中可以指定要存储在列表中的元素的最大容量。此队列是使用节点集合的链接表实现的。如果线程尝试在此队列已满时向其添加元素，线程将被阻塞，直到从队列中除去了某个现有元素。此队列的吞吐量高于 ArrayBlockingQueue。
- PriorityBlockingQueue：这是一个阻塞队列，它根据优先级对元素排序。如果线程尝试在此队列已满时向其添加元素，线程将被阻塞，直到从队列中除去了某个现有元素。此外，当线程尝试除去元素并且队列为空时，将阻塞线程。
- SynchronousQueue：这是一个阻塞队列，它阻止将数据添加到队列的请求，直到收到了从队列检索数据的请求，反之亦然。此队列不具有存储任何数据元素的容量。
- DelayQueue：此队列根据剩余时间执行元素排序，在此时间之前可从队列中除去这些元素。在此类队列中，仅当延迟时间为 0 时才能从队列中检索对象。只有实现 Delayed 接口的那些对象才能用于此队列。
- ConcurrentHashMap：此类是提供并发的 java.util.Map 接口的实现。此类适用于多线程环境，因为它将内部表分段。这允许多个线程执行并发读操作。然而，只能对需要更新的段应用锁。比起 Hashtable 类，此类不允许键具有空值。并且也不允许重复值。
- ConcurrentLinkedQueue：这是基于链接节点的基于 FIFO 的队列。此队列可用于并发访问。在此队列中，队列的头部表示在队列中时间最长的元素。另一方面，尾部表示在队列中时间最短的元素。此队列不允许在队列中输入空元素。
- ConcurrentSkipListMap：这是 NavigableMap 接口的并发实现。您可使用 ConcurrentSkipListMap 映射的对象存储“键－值”对，它们按照键的自然顺序进行排序，或者由映射创建时提供的比较器进行排序，具体取决于使用哪个构造函数。此外，ConcurrentSkipListMap 集合中的插入、除去、更新和访问操作可安全地由多个线程并发执行。
- ConcurrentSkipListSet：类似于 ConcurrentSkipListMap，可使用 ConcurrentSkipListSet 类存储 Set 集合中的元素，其中插入、除去、更新和访问操作可安全地由多个线程并发执行。ConcurrentSkipListSet 类实现 java.util 包的 NavigableSet 接口以使导航方法报告给定搜索目标的最接近匹配。此外，类似于 ConcurrentSkipListMap 类，ConcurrentSkipListSet 中存储的元素按照自然顺序排序或者由比较器进行排序。

- CopyOnWriteArrayList：这是 ArrayList 类的线程安全实现。在此列表中，添加和设置之类的操作是通过对数组进行全新的复制而执行的。如果要执行的读操作数目大于要执行的更新操作数目，此类列表比同步 ArrayList 更理想。
- CopyOnWriteArraySet：此类表示将 CopyOnWriteArrayList 用于其操作的集合。由于它将 CopyOnWriteArrayList 用于其操作，因此它是线程安全的。并且，要执行的读操作数多于要执行的更新操作数时最好使用它。

实现 ExecutorService

思考以下场景：您需要构建 Java 应用程序，其中需要执行若干并发任务。为了创建此类并发应用程序，需要创建和管理若干线程。这是繁重的任务，因为需要编写很多程序代码。而且，由于使用多个线程，高效管理计算机资源和应用程序性能就成为一个难题。为了避免以上问题，可使用执行器。执行器是用于执行任务的机制。

java.util.concurrent 包提供执行器功能。可有效地使用执行器管理线程任务，例如它们的实例化和执行，这样可提高应用程序处理大量线程时的性能。通过使用执行器，线程的创建和管理可与应用程序的其余部分分离。java.util.concurrent 包中定义的执行器接口如下：

- Executor
- ExecutorService

Executor

Executor 接口使用执行器执行任务。*任务*是实现 Runnable 或 Callable 接口的对象。实现 Runnable 接口的任务是可运行任务，实现 Callable 接口的任务是可调用任务。可运行任务和可调用任务之间的区别是后者会返回结果和抛出检查到的异常。可运行任务的功能由 run() 方法定义，而可调用任务的功能由 call() 方法定义。比起 run() 方法，call() 方法返回结果并且还允许它在无法计算结果的情况下抛出异常。

Executor 接口提供 execute() 方法，该方法执行 Runnable 任务（而不是使用 start() 方法）启动可运行任务。该方法不执行 Callable 任务。execute() 方法的签名是：

```
public void execute(Runnable task)
```

在以上签名中，task 是在未来某个时间执行的可运行任务。任务是使用线程池中的线程执行的，该池是可用于执行工作的线程的集合。此类线程又称为工作程序线程。工作程序线程不会在完成任务时死亡。它们将变为空闲，可被重用以执行任务。这样就不必创建不同线程，从而提高了效率。

使用 Executor 接口的一些限制如下：

- 执行器不能用于执行一组可运行任务。
- 执行器执行可运行任务，因为 Runnable 的 run() 方法不返回值，因此可运行任务不能将值返回给其调用方法。
- 执行器不能用于取消和执行可运行任务之类的操作或者确定可运行任务是否完成执行。
- 无法关闭执行器。

以上限制可通过 ExecutorService 接口克服。

ExecutorService

ExecutorService 扩展 Executor 接口。它还用于使用线程池提交和执行任务的用途。ExecutorService 可用于大型线程池以执行可运行或可调用任务。ExecutorService 提供附加方法，这些方法添加了更多管理执行器任务的功能，例如将它关闭。下表列出 ExecutorService 接口的一些方法。

方法	*描述*
boolean isShutdown()	*此方法在执行器已关闭的情况下返回 true。*
void shutdown()	*此方法用于启动执行器服务的关闭。*
boolean isTerminated()	*此方法用于检查执行器服务是否处于终止状态。如果所有未决任务都已在关闭后完成执行，它返回 true。*
List<Runnable> shutdownNow()	*此方法用于停止所有执行中的任务，并且返回等待执行的任务列表。*
Future<?> submit(Runnable task)	*此方法用于提交可运行任务以供执行。它还返回 Future 对象，该对象提供关于任务状态和结果的信息。*
Future<?> submit(Callable<T> task)	*此方法用于提交可调用任务并返回该任务的 Future 对象。*

ExecutorService 接口的方法

可通过 newFixedThreadPool() 方法使用固定线程池 ExecutorService。newFixedThreadPool() 方法具有以下签名：

```
public static ExecutorService newFixedThreadPool(int numberofThreads)
```

在以上签名中，numberofThreads 指定池中线程数目。

可使用以下代码段创建包含两个线程的线程池，这些线程用于执行已提交执行的任务：

```
ExecutorService name = Executors.newFixedThreadPool(2);
```

在上述代码段中，name 是要创建的固定线程池执行器服务的名称。为了创建具有固定数量线程的线程池，使用 Executors 类的 newFixedThreadPool() 方法。Executors 类是一个实用程序类。它提供功能以创建 Executor 和 ExceutorService 对象。

也可通过 newCachedThreadPool() 方法创建缓存线程池 ExecutorService。此方法具有以下签名：

```
public static ExecutorService newCachedThreadPool()
```

缓存线程池允许：

- 在需要时创建新线程。
- 重用线程，因为池中使用的线程可在完成任务之后被重用。
- 终止空闲时间段为 60 秒的线程。

可使用以下代码段创建新的缓存线程池执行器服务：

```
ExecutorService threadpool = Executors.newCachedThreadPool();
```

在上述代码段中，创建了新的缓存线程池执行器服务 `threadpool`。思考以下代码，它显示如何使用执行器服务并执行 `Runnable` 任务：

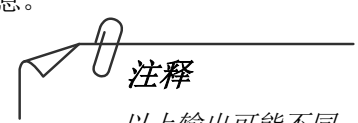
```
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Task implements Runnable {

String taskname;
public Task(String name)
{
taskname = name;
}
public void run()
    {
System.out.println("The task name is " +taskname +" executed by "
+Thread.currentThread().getName());
}
}
class TaskExecutor {
public static void main(String a[])
{
Task task1 = new Task("Task1");
Task task2 = new Task("Task2");
Task task3 = new Task("Task3");
ExecutorService threadexecutor = Executors.newCachedThreadPool();
System.out.println("Executor started");
threadexecutor.execute(task1);
threadexecutor.execute(task2);
threadexecutor.execute(task3);
threadexecutor.shutdown();
if(threadexecutor.isShutdown())
{
System.out.println("All tasks completed.The executor is shutting down.");
}
}
}
```

上述代码的输出是：

```
Executor started
```

```
The task name is Task2 executed by pool-1-thread-2
The task name is Task1 executed by pool-1-thread-1
The task name is Task3 executed by pool-1-thread-3
All tasks completed.The executor is shutting down.
```

在上述代码中，完成了三个任务 task1、task2 和 task3。然后创建了新的缓存线程池执行器服务 threadexecutor，所有任务都由此执行器服务执行。然后将检查执行器服务是否关闭并显示相应消息。



注释

以上输出可能不同。

思考以下代码，它显示如何使用执行器服务并执行 Callable 任务：

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
class Task implements Callable {

String taskname;
public Task(String name)
{
taskname = name;
}
public String call()
{
System.out.println("The task name is " +taskname +" executed by "
+Thread.currentThread().getName());
return taskname;
}
}
public class CallableTaskDemo {
public static void main(String a[])
{
Task task1 = new Task("Task1");
Task task2 = new Task("Task2");
Task task3 = new Task("Task3");
ExecutorService threadexecutorPool1 = Executors.newFixedThreadPool(3);
ExecutorService threadexecutorPool2 = Executors.newFixedThreadPool(3);
Future<String> f1=threadexecutorPool1.submit(task1);
Future<String> f2=threadexecutorPool2.submit(task2);
Future<String> f3=threadexecutorPool2.submit(task3);

}
}
```

上述代码的输出是：

```
The task name is Task1 executed by pool-1-thread-1
```

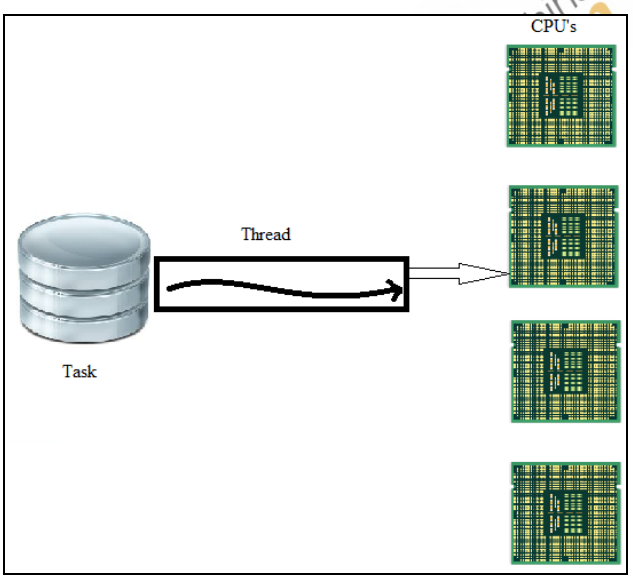
```
The task name is Task3 executed by pool-2-thread-2
The task name is Task2 executed by pool-2-thread-1
```

在上述代码中，创建了类 Task。Task 类实现 Callable 接口。此类还重写 call()。然后在类 CallableTaskDemo 中创建了两个执行器服务 threadexecutorPool1 和 threadexecutorPool2。然后创建了 Task 类的任务对象 task1、task2 和 task3。task1 任务提交给 threadexecutorPool1 池以供执行，task2 和 task3 提交给 threadexecutorPool2 以供执行。

实现分岔/汇合框架

思考以下情况：您需要创建大规模并发应用程序，假设它运行在具有多个处理器的计算机上。对于此类应用程序，可能发生这样的情况：应用程序未利用所有可用处理器来执行其任务。这将导致计算能力的浪费和效率损失。为了克服此问题并实现最大的 CPU 利用率，需要确保多个任务以并行方式在多个处理器上执行。为了满足以上需求，Java 提供分岔/汇合框架。

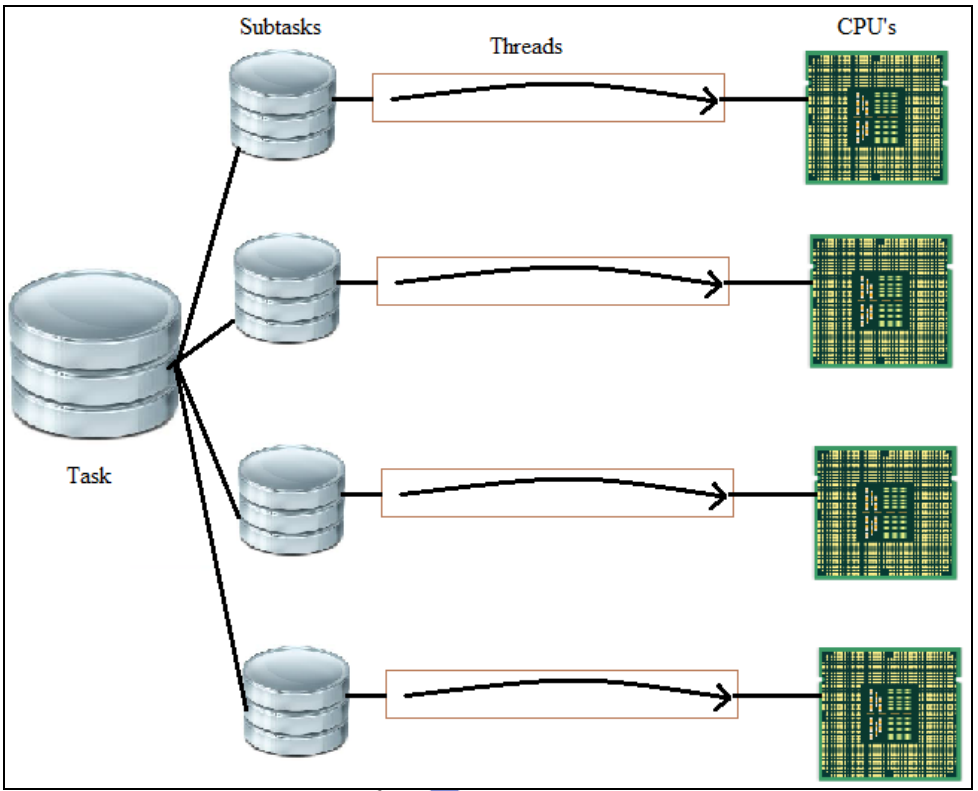
下图显示在应用程序未利用所有可用处理器执行其任务时是如何浪费计算能力的。



CPU 利用率的浪费

分岔/汇合基于以下原理：将大任务分解为小任务直到它不能进一步分解为止。然后，将以并行方式执行较小的任务，将合并它们的结果以获得最终结果。

下图显示如何通过分岔/汇合框架实现有效的 CPU 利用率。



有效 CPU 利用率

分岔/汇合框架利用线程池执行一个任务的各个子任务。在此框架中，如果线程在完成其任务后一直处于空闲状态，可使用该线程窃取另一线程的未决工作。这称为工作窃取。工作窃取提高了应用程序的执行速度和性能。然而，如果只有一个 CPU，分岔/汇合框架是无益的。

下表列出可用于实现分岔/汇合框架的 java.util.concurrent 包的类。

类	*描述*
ForkJoinPool	*此方法实现 ExecutorService 接口。它用于创建线程池以执行任务并实现工作窃取。*
ForkJoinTask	*此方法是将在 ForkJoinPool 中执行的所有任务的超类。这些任务在量级上比线程要轻。ForkJoinTask 在被提交给 ForkJoinPool 时启动执行。*
RecursiveTask	*此方法用于要通过分岔/汇合框架执行的任务需要返回结果的情况。*
RecursiveAction	*此方法用于要通过分岔/汇合框架执行的任务不需要返回结果的情况。*

java.util.concurrent 包的类

思考以下场景：您需要从一大列整数中找出最大值。在此类场景中，可实现分岔/汇合框架以并行执行计算并实现最大处理效率。以下代码显示如何实现分岔/汇合编程以找出一列整数中的最大值：

```
import java.util.Random;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class MaxFind extends RecursiveTask<Integer> {

private static final int SEQUENTIAL_THRESHOLD = 100;

private final int[] data;
private final int start;
private final int end;

public MaxFind(int[] data, int start, int end) {
this.data = data;
this.start = start;
this.end = end;
}

public MaxFind(int[] data) {
this(data, 0, data.length);
}

protected Integer compute() {
final int length = end - start;
if (length < SEQUENTIAL_THRESHOLD) {
return computeDirectly();
}
final int split = length / 2;
final MaxFind left = new MaxFind(data, start, start + split);
left.fork();
```

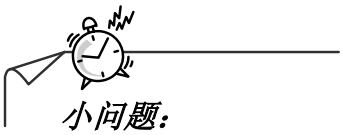
```
final MaxFind right = new MaxFind(data, start + split, end);
return Math.max(right.compute(), left.join());
}

private Integer computeDirectly() {
System.out.println(Thread.currentThread() + " is searching array index:" +
start   + " to " + end);
int max = Integer.MIN_VALUE;
for (int i = start; i < end; i++) {
if (data[i] > max) {
max = data[i];
}
}
return max;
}
public static void main(String[] args) {
// create a random data set
final int[] data = new int[200];
final Random random = new Random();
for (int i = 0; i < data.length; i++) {
data[i] = random.nextInt(350);
}
// submit the task to the pool
final ForkJoinPool pool = new ForkJoinPool(4);
final MaxFind finder = new MaxFind(data);
System.out.println("The max value is:"+pool.invoke(finder));

}
}
```

上述代码的输出是：

```
Thread[ForkJoinPool-1-worker-1,5,main] is searching array index:150 to 200
Thread[ForkJoinPool-1-worker-4,5,main] is searching array index:0 to 50
Thread[ForkJoinPool-1-worker-2,5,main] is searching array index:50 to 100
Thread[ForkJoinPool-1-worker-3,5,main] is searching array index:100 to 150
The max value is:349
```

在上述代码中，创建了类 `MaxFind`。此类扩展 `RecursiveTask` 类以实现分岔/汇合框架。`RecursiveTask` 类的使用使程序员能够将较大的任务分为较小的任务，并以并发方式运行它们以实现效率。在 `main` 方法中，创建了包含 `200` 个整数的数组，并在其中插入了随机值，最大范围为 `350`。然后，创建了 `ForkJoinPool` 对象，它带有 `4` 个可用于执行计算的工作程序线程。然后创建了分岔/汇合任务 `finder` 并使用 `invoke()` 方法将它提交给分岔/汇合池。将使用 `invoke()` 方法返回分岔/汇合任务的结果。

在 `MaxFind` 类中，方法 `compute` 包含执行计算和将任务分成子任务（如果需要）的逻辑。`SEQUENTIAL_THRESHOLD` 参数用于指定阈值，该阈值用于确定任务是否大到可以分成子任务。在此例中，`SEQUENTIAL_THRESHOLD` 的值为 `100`，它指示如果数组大小等于或小于 `100` 个元素，那么任务将以顺序方式作为单个任务执行，而不分成多个任务。然而，如果数组大小大于 `100`，任务将分成多个任务。`fork()` 方法用于将任务分成子任务 `left` 和 `right` 并以并行方式计算它们。然后将使用 `join()` 方法获取并行执行的 `left` 和 `right` 子任务的计算结果。`computeDirectly()` 方法用于搜索数组的最大数字并将它返回。



小问题：

_____类是将在 ForkJoinPool 中执行的所有任务的超类。

答案：

ForkJoinTask

练习问题

1. 指出用于同步方法的关键字。
   a. `synchronize`
   b. `synchronized`
   c. `lock`
   d. `monitor`

2. 指出用于临时释放锁的方法。
   a. `wait()`
   b. `release()`
   c. `synchronize()`
   d. `notify()`

3. 指出用于唤醒正等待执行中对象的监督程序的单个线程的方法。
   a. `wait()`
   b. `release()`
   c. `synchronize()`
   d. `notify()`

4. 指出提供了可用于创建原子变量以执行原子操作的类和方法的包。
   a. `java.util.concurrent.atomic`
   b. `java.util.atomic`
   c. `java.util.concurrent.atomicInteger`
   d. `java.util.concurrent.Integer`

5. 指出用于线程只能在锁可用时获取锁的情况的方法。
   a. `unlock()`
   b. `tryLock()`
   c. `lock()`
   d. `checklock()`

小结

在本章中，您学习了：

- 线程的同步确保当两个或多个线程需要访问共享资源时，一次仅一个线程使用该资源。
- 可使用以下方法实现线程同步：
  - 使用同步方法
  - 使用同步语句
- 同步基于监视器的概念，监视器是一个用作锁的对象。
- 当线程在同步方法内的时候，所有试图在同一个时间调用该方法的其他线程必须等待。
- 当类中未使用同步方法且您无权访问源代码时，使用同步语句。
- 多线程通过称为进程间通信的机制除去了轮询的概念。
- 线程间通信所使用的各种方法有：
  - `wait()`
  - `notify()`
  - `notifyAll()`
- 并发使 Java 程序员能够通过创建并发程序提高其应用程序的效率和资源利用率。
- 要实现并发，Java 提供 `java.util.concurrent` 包。
- `java.util.concurrent` 包具有以下子包以支持并发编程：
  - `java.util.concurrent.atomic`
  - `java.util.concurrent.locks`
- 原子变量是单个变量，在其上执行的自增和自减之类的操作是一步完成的。此类操作称为原子操作。
- 原子操作是作为一个单元执行的操作。
- 可重入锁用于线程中间的互锁。
- 此外，为了分别对读和写操作实现不同的锁，Java 在 `java.util.concurrent.locks` 包中提供 `ReentrantReadWriteLock` 类。
- 提供了以下 `synchronizer` 类：
  - `Semaphore`
  - `CountdownLatch`
  - `CyclicBarrier`
  - `Phaser`
  - `Exchanger`
- 使用 `Collection` 类通过允许多个线程共享数据集合提高并发应用程序的性能和可伸缩性。
- 可有效地使用执行器管理线程任务，例如它们的实例化和执行，这样可提高应用程序处理大量线程时的性能。
- `Executor` 接口使用执行器执行任务。
- `ExecutorService` 提供附加方法，这些方法添加了更多管理执行器任务的功能，例如将它关闭。

- 分岔/汇合基于以下原理：将大任务分解为小任务直到它不能进一步分解为止。