

## 一、Java JUC 简介

---

- 在 Java 5.0 提供了 **java.util.concurrent** ( 简称 JUC ) 包，在此包中增加了在并发编程中很常用的实用工具类，用于定义类似于线程的自定义子系统，包括线程池、异步 IO 和轻量级任务框架。提供可调的、灵活的线程池。还提供了设计用于多线程上下文中的 Collection 实现等。

## 二、volatile 关键字-内存可见性

---

### 1、内存可见性

---

- 内存可见性 (Memory Visibility) 是指当某个线程正在使用对象状态而另一个线程在同时修改该状态，需要确保当一个线程修改了对象状态后，其他线程能够看到发生的状态变化。
- 可见性错误是指当读操作与写操作在不同的线程中执行时，我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。
- 我们可以通过同步来保证对象被安全地发布。除此之外我们也可以使用一种更加轻量级的 volatile 变量。

### 2、volatile 关键字

---

- Java 提供了一种稍弱的同步机制，即 **volatile** 变量，用来确保将变量的更新操作通知到其他线程。可以将 **volatile** 看做一个轻量级的锁，但是又与锁有些不同：

- 对于多线程，不是一种互斥关系
- 不能保证变量状态的“原子性操作”

```
package com.wl.java;

/*
 * 一、volatile 关键字：当多个线程进行操作共享数据时，可以保证内存中的数据可见。
 *                      相较于 synchronized 是一种较为轻量级的同步策略。
 *
 * 注意：
 * 1. volatile 不具备“互斥性”
 * 2. volatile 不能保证变量的“原子性”
 */
public class TestVolatile {

    public static void main(String[] args) {
        ThreadDemo td = new ThreadDemo();
        new Thread(td).start();

        while(true){
            if(td.isFlag()){
                System.out.println("-----");
                break;
            }
        }
    }
}

class ThreadDemo implements Runnable {

    private volatile boolean flag = false;

    @Override
    public void run() {

        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
        }

        flag = true;
    }
}
```

```
        System.out.println("flag=" + isFlag());
    }

    public boolean isFlag() {
        return flag;
    }

    public void setFlag(boolean flag) {
        this.flag = flag;
    }
}
```

```
-----
flag=true
```

## 三、原子变量-CAS算法

### 1、CAS 算法

- CAS (Compare-And-Swap) 是一种硬件对并发的支持，针对多处理器操作而设计的处理器中的一种特殊指令，用于管理对共享数据的并发访问。
- CAS 是一种无锁的非阻塞算法的实现。
- CAS 包含了 3 个操作数：
  - 需要读写的内存值 V
  - 进行比较的值 A
  - 拟写入的新值 B
- 当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作。

### 2、原子变量

- 类的小工具包，支持在单个变量上解除锁的线程安全编程。事实上，此包中的类可将 `volatile` 值、字段和数组元素的概念扩展到那些也提供原子条件更新操作的类。
- 类 `AtomicBoolean`、`AtomicInteger`、`AtomicLong` 和 `AtomicReference` 的实例各自提供对相应类型单个变量的访问和更新。每个类也为该类型提供适当的实用工具方法。
- `AtomicIntegerArray`、`AtomicLongArray` 和 `AtomicReferenceArray` 类进一步扩展了原子操作，对这些类型的数组提供了支持。这些类在为其数组元素提供 `volatile` 访问语义方面也引人注目，这对于普通数组来说是不受支持的。
- **核心方法： `boolean compareAndSet(expectedValue, updateValue)`**
- `java.util.concurrent.atomic` 包下提供了一些原子操作的常用类：
  - `AtomicBoolean`、`AtomicInteger`、`AtomicLong`、`AtomicReference`
  - `AtomicIntegerArray`、`AtomicLongArray`
  - `AtomicMarkableReference`
  - `AtomicReferenceArray`
  - `AtomicStampedReference`

```
package com.wl.java.juc;

import java.util.concurrent.atomic.AtomicInteger;

/*
 * 一、i++ 的原子性问题：i++ 的操作实际上分为三个步骤“读-改-写”
 *      int i = 10;
 *      i = i++; //10
 *
 *      int temp = i;
 *      i = i + 1;
 *      i = temp;
 *
 * 二、原子变量：在 java.util.concurrent.atomic 包下提供了一些原子变量。
 *      1. volatile 保证内存可见性
 *      2. CAS (Compare-And-Swap) 算法保证数据变量的原子性
 *          CAS 算法是硬件对于并发操作的支持
 *          CAS 包含了三个操作数：
 *          ①内存值 V
 *          ②预估值 A
 *          ③更新值 B
 *          当且仅当 V == A 时，V = B；否则，不会执行任何操作。
 */
public class TestAtomicDemo {

    public static void main(String[] args) {
        AtomicDemo ad = new AtomicDemo();

        for (int i = 0; i < 10; i++) {
            new Thread(ad).start();
        }
    }
}

class AtomicDemo implements Runnable{
```

```
// private volatile int serialNumber = 0;

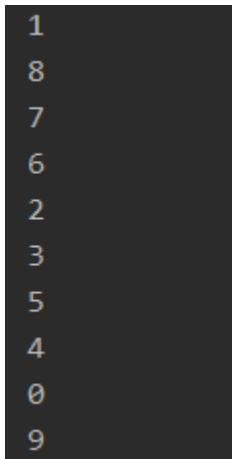
private AtomicInteger serialNumber = new AtomicInteger(0);

@Override
public void run() {

    try {
        Thread.sleep(200);
    } catch (InterruptedException e) {
    }

    System.out.println(getSerialNumber());
}

public int getSerialNumber(){
    return serialNumber.getAndIncrement();
}
}
```



1  
8  
7  
6  
2  
3  
5  
4  
0  
9

### 3、模拟CAS算法

```
package com.wl.java.juc;

/*
 * 模拟 CAS 算法
 */
public class TestCompareAndSwap {

    public static void main(String[] args) {
        final CompareAndSwap cas = new CompareAndSwap();

        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {

                @Override
                public void run() {
                    int expectedValue = cas.get();
                    boolean b = cas.compareAndSet(expectedValue, (int)
(Math.random() * 101));
                    System.out.println(b);
                }
            })
        }
    }
}
```

```
        }).start();
    }

}

}

class CompareAndSwap{
    private int value;

    //获取内存值
    public synchronized int get(){
        return value;
    }

    //比较
    public synchronized int compareAndSwap(int expectedValue, int newValue){
        int oldValue = value;

        if(oldValue == expectedValue){
            this.value = newValue;
        }

        return oldValue;
    }

    //设置
    public synchronized boolean compareAndSet(int expectedValue, int newValue){
        return expectedValue == compareAndSwap(expectedValue, newValue);
    }
}
```

## 四、ConcurrentHashMap 锁分段机制

---

- Java 5.0 在 `java.util.concurrent` 包中提供了多种并发容器类来改进同步容器的性能。
- `ConcurrentHashMap` 同步容器类是Java 5 增加的一个线程安全的哈希表。对与多线程的操作，介于 `HashMap` 与 `Hashtable` 之间。内部采用“锁分段”机制替代 `Hashtable` 的独占锁。进而提高性能。
- 此包还提供了设计用于多线程上下文中的 `Collection` 实现：  
`ConcurrentHashMap`、`ConcurrentSkipListMap`、`ConcurrentSkipListSet`、`CopyOnWriteArrayList` 和 `CopyOnWriteArraySet`。当期望许多线程访问一个给定 collection 时，`ConcurrentHashMap` 通常优于同步的 `HashMap`，`ConcurrentSkipListMap` 通常优于同步的 `TreeMap`。当期望的读数和遍历远远大于列表的更新数时，`CopyOnWriteArrayList` 优于同步的 `ArrayList`。

```
package com.wl.java.juc;

import java.util.Iterator;
import java.util.concurrent.CopyOnWriteArrayList;

/*
 * CopyOnWriteArrayList/CopyOnWriteArraySet : “写入并复制”
 * 注意：添加操作多时，效率低，因为每次添加时都会进行复制，开销非常的大。并发迭代操作多时可以选择。
 */
public class TestCopyOnWriteArrayList {

    public static void main(String[] args) {
        HelloThread ht = new HelloThread();

        for (int i = 0; i < 10; i++) {
            new Thread(ht).start();
        }
    }
}

class HelloThread implements Runnable{

    // private static List<String> list = Collections.synchronizedList(new
    ArrayList<String>()); // 报并发修改异常

    private static CopyOnWriteArrayList<String> list = new
    CopyOnWriteArrayList<>();

    static{
        list.add("AA");
        list.add("BB");
        list.add("CC");
    }

    @Override
```

```

public void run() {

    Iterator<String> it = list.iterator();

    while(it.hasNext()){
        System.out.println(it.next());

        list.add("AA");
    }

}
}

```

## 五、CountDownLatch 闭锁

- Java 5.0 在 `java.util.concurrent` 包中提供了多种并发容器类来改进同步容器的性能。
- `CountDownLatch` 一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。
- 闭锁可以延迟线程的进度直到其到达终止状态，闭锁可以用来确保某些活动直到其他活动都完成才继续执行：
  - 确保某个计算在其需要的所有资源都被初始化之后才继续执行；
  - 确保某个服务在其依赖的所有其他服务都已经启动之后才启动；
  - 等待直到某个操作所有参与者都准备就绪再继续执行。

```

package com.wl.java.juc;

import java.util.concurrent.CountDownLatch;

/*
 * CountDownLatch : 闭锁，在完成某些运算时，只有其他所有线程的运算全部完成，当前运算才继续执行
 */
public class TestCountDownLatch {

    public static void main(String[] args) {
        final CountDownLatch latch = new CountDownLatch(50); // latch参数设置和线程数相同
        LatchDemo ld = new LatchDemo(latch);

        long start = System.currentTimeMillis();

        for (int i = 0; i < 50; i++) {
            new Thread(ld).start();
        }
    }
}

```



```

    }

    try {
        latch.await(); // 主线程等待
    } catch (InterruptedException e) {
    }

    long end = System.currentTimeMillis();

    System.out.println("耗费时间为: " + (end - start));
}

}

class LatchDemo implements Runnable {

    private CountdownLatch latch;

    public LatchDemo(CountdownLatch latch) {
        this.latch = latch;
    }

    @Override
    public void run() {

        try {
            for (int i = 0; i < 50000; i++) {
                if (i % 2 == 0) {
                    System.out.println(i);
                }
            }
        } finally {
            latch.countDown(); // latch的参数递减1
        }
    }
}

```

## 六、实现 Callable 接口

---

- Java 5.0 在 `java.util.concurrent` 提供了一个新的创建执行线程的方式： `Callable` 接口
- `Callable` 接口类似于 `Runnable`，两者都是为那些其实例可能被另一个线程执行的类设计的。但是 `Runnable` 不会返回结果，并且无法抛出经过检查的异常。
- `Callable` 需要依赖 `FutureTask`，`FutureTask` 也可以用作闭锁。

```
package com.wl.java.juc2;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

/*
 * 一、创建执行线程的方式三：实现 Callable 接口。 相较于实现 Runnable 接口的方式，方法可以有返回值，并且可以抛出异常。
 *
 * 二、执行 Callable 方式，需要 FutureTask 实现类的支持，用于接收运算结果。 FutureTask 是 Future 接口的实现类
 */
public class TestCallable {

    public static void main(String[] args) {
        ThreadDemo td = new ThreadDemo();

        //1.执行 Callable 方式，需要 FutureTask 实现类的支持，用于接收运算结果。
        FutureTask<Integer> result = new FutureTask<>(td);

        new Thread(result).start();

        //2.接收线程运算后的结果
        try {
            Integer sum = result.get(); //FutureTask 可用于闭锁，线程执行完才能接收线程运算后的结果
            System.out.println(sum); // 705082704
            System.out.println("-----");
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}

class ThreadDemo implements Callable<Integer>{

    @Override
    public Integer call() throws Exception {
        int sum = 0;
    }
}
```

```

        for (int i = 0; i <= 100000; i++) {
            sum += i;
        }

        return sum;
    }
}

/*class ThreadDemo implements Runnable{

    @Override
    public void run() {
    }

}*/

```

## 七、Lock 同步锁

### 1、显示锁Lock

#### 显示锁 Lock

- 在 Java 5.0 之前，协调共享对象的访问时可以使用的机制只有 `synchronized` 和 `volatile`。Java 5.0 后增加了一些新的机制，但并不是一种替代内置锁的方法，而是当内置锁不适用时，作为一种可选择的高级功能。
- `ReentrantLock` 实现了 `Lock` 接口，并提供了与 `synchronized` 相同的互斥性和内存可见性。但相较于 `synchronized` 提供了更高的处理锁的灵活性。

```

package com.wl.java.juc2;

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

/*
 * 一、用于解决多线程安全问题的方式：
 *
 * synchronized:隐式锁
 * 1. 同步代码块
 *
 */

```

```

* 2. 同步方法
*
* jdk 1.5 后:
* 3. 同步锁 Lock
* 注意: 是一个显示锁, 需要通过 lock() 方法上锁, 必须通过 unlock() 方法进行释放锁
*/
public class TestLock {

    public static void main(String[] args) {
        Ticket ticket = new Ticket();

        new Thread(ticket, "1号窗口").start();
        new Thread(ticket, "2号窗口").start();
        new Thread(ticket, "3号窗口").start();
    }
}

class Ticket implements Runnable{

    private int tick = 100;

    private Lock lock = new ReentrantLock();

    @Override
    public void run() {
        while(true){

            lock.lock(); //上锁

            try{
                if(tick > 0){
                    try {
                        Thread.sleep(200);
                    } catch (InterruptedException e) {
                    }

                    System.out.println(Thread.currentThread().getName() + " 完成售票, 余票为: " + --tick);
                }
            }finally{
                lock.unlock(); //释放锁
            }
        }
    }
}

```

## 2、生产者消费者案例-虚假唤醒(synchronized)

```

package com.wl.java.juc2;

```

```

/*
 * 生产者和消费者案例
 */
public class TestProductAndConsumer {

    public static void main(String[] args) {
        Clerk clerk = new Clerk();

        Productor pro = new Productor(clerk);
        Consumer cus = new Consumer(clerk);

        new Thread(pro, "生产者 A").start();
        new Thread(cus, "消费者 B").start();

        new Thread(pro, "生产者 C").start();
        new Thread(cus, "消费者 D").start();
    }
}

//店员
class Clerk{
    private int product = 0;

    //进货
    public synchronized void get(){//循环次数: 0
        while(product >= 1){//为了避免虚假唤醒问题, 应该总是使用在循环中
            System.out.println("产品已满! ");

            try {
                this.wait();
            } catch (InterruptedException e) {
            }
        }

        System.out.println(Thread.currentThread().getName() + " : " +
++product);
        this.notifyAll();
    }

    //卖货
    public synchronized void sale(){//product = 0; 循环次数: 0
        while(product <= 0){
            System.out.println("缺货! ");

            try {
                this.wait();
            } catch (InterruptedException e) {
            }
        }

        System.out.println(Thread.currentThread().getName() + " : " + --
product);
        this.notifyAll();
    }
}

```

//生产者

```
class Productor implements Runnable{
    private Clerk clerk;

    public Productor(Clerk clerk) {
        this.clerk = clerk;
    }

    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
            }

            clerk.get();
        }
    }
}
```

//消费者

```
class Consumer implements Runnable{
    private Clerk clerk;

    public Consumer(Clerk clerk) {
        this.clerk = clerk;
    }

    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            clerk.sale();
        }
    }
}
```

```
缺货!
缺货!
生产者 C : 1
消费者 B : 0
缺货!
缺货!
生产者 A : 1
消费者 D : 0
缺货!
缺货!
生产者 C : 1
消费者 B : 0
缺货!
缺货!
生产者 A : 1
消费者 D : 0
缺货!
缺货!
生产者 C : 1
消费者 B : 0
缺货!
缺货!
生产者 A : 1
消费者 D : 0
缺货!
缺货!
生产者 C : 1
```

### 3、生产者消费者案例-虚假唤醒(lock)

```
package com.wl.java.juc2;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/*
 * 生产者消费者案例:
 */
public class TestProductorAndConsumerForLock {

    public static void main(String[] args) {
        Clerk clerk = new Clerk();

        Productor pro = new Productor(clerk);
        Consumer con = new Consumer(clerk);

        new Thread(pro, "生产者 A").start();
        new Thread(con, "消费者 B").start();

        new Thread(pro, "生产者 C").start();
        new Thread(con, "消费者 D").start();
    }
}

class Clerk {
    private int product = 0;

    private Lock lock = new ReentrantLock();
```

```

private Condition condition = lock.newCondition();

// 进货
public void get() {
    lock.lock();

    try {
        while (product >= 1) { // 为了避免虚假唤醒，应该总是使用在循环中。
            System.out.println("产品已满！");

            try {
                condition.await();
            } catch (InterruptedException e) {
            }

        }
        System.out.println(Thread.currentThread().getName() + " : "
            + ++product);

        condition.signalAll();
    } finally {
        lock.unlock();
    }
}

// 卖货
public void sale() {
    lock.lock();

    try {
        while (product <= 0) {
            System.out.println("缺货！");

            try {
                condition.await();
            } catch (InterruptedException e) {
            }

        }

        System.out.println(Thread.currentThread().getName() + " : "
            + --product);

        condition.signalAll();
    } finally {
        lock.unlock();
    }
}

// 生产者
class Productor implements Runnable {

    private Clerk clerk;

    public Productor(Clerk clerk) {
        this.clerk = clerk;
    }
}

```



```

    }

    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            clerk.get();
        }
    }
}

// 消费者
class Consumer implements Runnable {

    private Clerk clerk;

    public Consumer(Clerk clerk) {
        this.clerk = clerk;
    }

    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            clerk.sale();
        }
    }
}

```



- 编写一个程序，开启 3 个线程，这三个线程的 ID 分别为 A、B、C，每个线程将自己的 ID 在屏幕上打印 10 遍，要求输出的结果必须按顺序显示。

如：ABCABCABC..... 依次递归

```
package com.wl.java.juc2;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/*
 * 编写一个程序，开启 3 个线程，这三个线程的 ID 分别为 A、B、C，每个线程将自己的 ID 在屏幕上
 * 打印 10 遍，要求输出的结果必须按顺序显示。
 * 如：ABCABCABC..... 依次递归
 */
public class TestABCAIternate {

    public static void main(String[] args) {
        AlternateDemo ad = new AlternateDemo();

        new Thread(new Runnable() {
            @Override
            public void run() {

                for (int i = 1; i <= 20; i++) {
                    ad.loopA(i);
                }

            }
        }, "A").start();

        new Thread(new Runnable() {
            @Override
            public void run() {

                for (int i = 1; i <= 20; i++) {
                    ad.loopB(i);
                }

            }
        }, "B").start();

        new Thread(new Runnable() {
            @Override
            public void run() {

                for (int i = 1; i <= 20; i++) {
                    ad.loopC(i);

                    System.out.println("-----");
                }

            }
        }, "C").start();
    }
}
```

```

        }
    }, "C").start();
}

}

class AlternateDemo{

    private int number = 1; //当前正在执行线程的标记

    private Lock lock = new ReentrantLock();
    private Condition condition1 = lock.newCondition();
    private Condition condition2 = lock.newCondition();
    private Condition condition3 = lock.newCondition();

    /**
     * @param totalLoop : 循环第几轮
     */
    public void loopA(int totalLoop){
        lock.lock();

        try {
            //1. 判断
            if(number != 1){
                condition1.await();
            }

            //2. 打印
            for (int i = 1; i <= 1; i++) {
                System.out.println(Thread.currentThread().getName() + "\t" + i +
"\t" + totalLoop);
            }

            //3. 唤醒
            number = 2;
            condition2.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void loopB(int totalLoop){
        lock.lock();

        try {
            //1. 判断
            if(number != 2){
                condition2.await();
            }

            //2. 打印
            for (int i = 1; i <= 1; i++) {
                System.out.println(Thread.currentThread().getName() + "\t" + i +
"\t" + totalLoop);
            }
        }
    }
}

```

```

        //3. 唤醒
        number = 3;
        condition3.signal();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public void loopC(int totalLoop){
    lock.lock();

    try {
        //1. 判断
        if(number != 3){
            condition3.await();
        }

        //2. 打印
        for (int i = 1; i <= 1; i++) {
            System.out.println(Thread.currentThread().getName() + "\t" + i +
"\t" + totalLoop);
        }

        //3. 唤醒
        number = 1;
        condition1.signal();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
}

```

```
A 1 1
B 1 1
C 1 1
-----
A 1 2
B 1 2
C 1 2
-----
A 1 3
B 1 3
C 1 3
-----
A 1 4
B 1 4
C 1 4
-----
A 1 5
B 1 5
C 1 5
-----
A 1 6
B 1 6
C 1 6
-----
A 1 7
```

## 十、ReadWriteLock 读写锁

- ReadWriteLock 维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 **writer**，读取锁可以由多个 **reader** 线程同时保持。写入锁是独占的。。
- ReadWriteLock 读取操作通常不会改变共享资源，但执行写入操作时，必须独占方式来获取锁。对于读取操作占多数的数据结构。ReadWriteLock 能提供比独占锁更高的并发性。而对于只读的数据结构，其中包含的不变性可以完全不需要考虑加锁操作。

```
package com.wl.java.juc2;

import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/*
 * 1. ReadWriteLock : 读写锁
 *
 * 写写/读写 需要“互斥”
 * 读读 不需要互斥
 *
 */
public class TestReadWriteLock {

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

```

        ReadWriteLockDemo rw = new ReadWriteLockDemo();

        new Thread(new Runnable() {

            @Override
            public void run() {
                rw.set((int)(Math.random() * 101));
            }
        }, "Write:").start();

        for (int i = 0; i < 100; i++) {
            new Thread(new Runnable() {

                @Override
                public void run() {
                    rw.get();
                }
            }).start();
        }
    }

}

class ReadWriteLockDemo{

    private int number = 0;

    private ReadWriteLock lock = new ReentrantReadWriteLock();

    //读
    public void get(){
        lock.readLock().lock(); //上锁

        try{
            System.out.println(Thread.currentThread().getName() + " : " +
number);
        }finally{
            lock.readLock().unlock(); //释放锁
        }
    }

    //写
    public void set(int number){
        lock.writeLock().lock();

        try{
            System.out.println(Thread.currentThread().getName());
            this.number = number;
        }finally{
            lock.writeLock().unlock();
        }
    }
}

```

```
Write:
Thread-0 : 12
Thread-2 : 12
Thread-3 : 12
Thread-4 : 12
Thread-1 : 12
Thread-11 : 12
Thread-6 : 12
Thread-7 : 12
Thread-8 : 12
Thread-10 : 12
Thread-5 : 12
Thread-12 : 12
Thread-13 : 12
Thread-14 : 12
Thread-17 : 12
Thread-18 : 12
Thread-15 : 12
Thread-16 : 12
Thread-19 : 12
Thread-24 : 12
Thread-20 : 12
Thread-9 : 12
Thread-21 : 12
Thread-22 : 12
```

## 十一、线程八锁

- 一个对象里面如果有多个synchronized方法，某一个时刻内，只要一个线程去调用其中的一个synchronized方法了，其它的线程都只能等待，换句话说，某一个时刻内，只能有唯一一个线程去访问这些synchronized方法
- 锁的是当前对象this，被锁定后，其它的线程都不能进入到当前对象的其它的synchronized方法
- 加个普通方法后发现和同步锁无关
- 换成两个对象后，不是同一把锁了，情况立刻变化。
- 都换成静态同步方法后，情况又变化
- 所有的非静态同步方法用的都是同一把锁——实例对象本身，也就是说如果一个实例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁，可是别的实例对象的非静态同步方法因为跟该实例对象的非静态同步方法用的是不同的锁，所以毋须等待该实例对象已获取锁的非静态同步方法释放锁就可以获取他们自己的锁。
- 所有的静态同步方法用的也是同一把锁——类对象本身，这两把锁是两个不同的对象，所以静态同步方法与非静态同步方法之间是不会有竞态条件的。但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，而不管是同一个实例对象的静态同步方法之间，还是不同的实例对象的静态同步方法之间，只要它们同一个类的实例对象！

```
package com.wl.java.juc2;

/*
 * 题目：判断打印的 "one" or "two" ?
 *
 * 1. 两个普通同步方法，两个线程，标准打印， 打印? //one two
 * 2. 新增 Thread.sleep() 给 getOne() ,打印? //one two
 * 3. 新增普通方法 getThree() , 打印? //three one two
 * 4. 两个普通同步方法，两个 Number 对象，打印? //two one
 * 5. 修改 getOne() 为静态同步方法，打印? //two one
 * 6. 修改两个方法均为静态同步方法，一个 Number 对象? //one two
 * 7. 一个静态同步方法，一个非静态同步方法，两个 Number 对象? //two one
 * 8. 两个静态同步方法，两个 Number 对象? //one two
 *
 * 线程八锁的关键：
```



- \* ① 非静态方法的锁默认为 **this**，静态方法的锁为 对应的 **Class** 实例
- \* ② 某一个时刻内，只能有一个线程持有锁，无论几个方法。

```
*/
public class TestThread8Monitor {

    public static void main(String[] args) {
        Number number = new Number();
        Number number2 = new Number();

        new Thread(new Runnable() {
            @Override
            public void run() {
                number.getOne();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
//                number.getTwo();
                number2.getTwo();
            }
        }).start();

        /*new Thread(new Runnable() {
            @Override
            public void run() {
                number.getThree();
            }
        }).start();*/
    }
}

class Number{

    public static synchronized void getOne(){//Number.class
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }

        System.out.println("one");
    }

    public synchronized void getTwo(){//this
        System.out.println("two");
    }

    public void getThree(){
        System.out.println("three");
    }
}
```

## 十二、线程池

- 第四种获取线程的方法：线程池，一个 `ExecutorService`，它使用可能的几个池线程之一执行每个提交的任务，通常使用 `Executors` 工厂方法配置。
- 线程池可以解决两个不同问题：由于减少了每个任务调用的开销，它们通常可以在执行大量异步任务时提供增强的性能，并且还可以提供绑定和管理资源（包括执行任务集时使用的线程）的方法。每个 `ThreadPoolExecutor` 还维护着一些基本的统计数据，如完成的任务数。
- 为了便于跨大量上下文使用，此类提供了很多可调整的参数和扩展钩子 (hook)。但是，强烈建议程序员使用较为方便的 `Executors` 工厂方法：

- `Executors.newCachedThreadPool()`（无界线程池，可以进行自动线程回收）
- `Executors.newFixedThreadPool(int)`（固定大小线程池）
- `Executors.newSingleThreadExecutor()`（单个后台线程）

它们均为大多数使用场景预定义了设置。

```
package com.wl.java.juc2;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

/*
 * 一、线程池：提供了一个线程队列，队列中保存着所有等待状态的线程。避免了创建与销毁额外开销，提高了响应的速度。
 *
 * 二、线程池的体系结构：
 *   java.util.concurrent.Executor : 负责线程的使用与调度的根接口
 *   |--**ExecutorService 子接口：线程池的主要接口
 *   |--ThreadPoolExecutor 线程池的实现类
 *   |--ScheduledExecutorService 子接口：负责线程的调度
 *   |--ScheduledThreadPoolExecutor : 继承 ThreadPoolExecutor， 实现 ScheduledExecutorService
 *
 * 三、工具类 : Executors
 *   ExecutorService newFixedThreadPool() : 创建固定大小的线程池
 *   ExecutorService newCachedThreadPool() : 缓存线程池，线程池的数量不固定，可以根据需求自动的更改数量。
 *   ExecutorService newSingleThreadExecutor() : 创建单个线程池。线程池中只有一个线程
 *
 *   ScheduledExecutorService newScheduledThreadPool() : 创建固定大小的线程，可以延迟或定时的执行任务。
 */
public class TestThreadPool {

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

```

//1. 创建线程池
ExecutorService pool = Executors.newFixedThreadPool(5);

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

for (int i = 0; i < 10; i++) {
    Future<Integer> future = pool.submit(new Callable<Integer>(){

        @Override
        public Integer call() throws Exception {
            int sum = 0;

            for (int i = 0; i <= 100; i++) {
                sum += i;
            }

            return sum;
        }

    });

    list.add(future);
}

pool.shutdown();

for (Future<Integer> future : list) {
    System.out.println(future.get());
}

/*ThreadPoolDemo tpd = new ThreadPoolDemo();

//2. 为线程池中的线程分配任务
for (int i = 0; i < 10; i++) {
    pool.submit(tpd);
}

//3. 关闭线程池
pool.shutdown();*/
}

// new Thread(tpd).start();
// new Thread(tpd).start();

}

class ThreadPoolDemo implements Runnable{

    private int i = 0;

    @Override
    public void run() {
        while(i <= 100){
            System.out.println(Thread.currentThread().getName() + " : " + i++);
        }
    }
}

```

```
}
```

```
5050
5050
5050
5050
5050
5050
5050
5050
5050
5050
```

## 十三、线程调度

### ScheduledExecutorService

- 一个 `ExecutorService`，可安排在给定的延迟后运行或定期执行的命令。

```
package com.wl.java.juc2;

import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

/*
 * 一、线程池：提供了一个线程队列，队列中保存着所有等待状态的线程。避免了创建与销毁额外开销，提高了响应的速度。
 *
 * 二、线程池的体系结构：
 *   java.util.concurrent.Executor  ： 负责线程的使用与调度的根接口
 *       |--**ExecutorService 子接口：线程池的主要接口
 *           |--ThreadPoolExecutor 线程池的实现类
 *           |--ScheduledExecutorService 子接口：负责线程的调度
 *               |--ScheduledThreadPoolExecutor  ： 继承 ThreadPoolExecutor， 实现 ScheduledExecutorService
 *
 * 三、工具类  ： Executors
 *   ExecutorService newFixedThreadPool()  ： 创建固定大小的线程池
 *   ExecutorService newCachedThreadPool()  ： 缓存线程池，线程池的数量不固定，可以根据需求自动的更改数量。
 *   ExecutorService newSingleThreadExecutor()  ： 创建单个线程池。线程池中只有一个线程
 */
```

\* `ScheduledExecutorService newScheduledThreadPool()` : 创建固定大小的线程, 可以延迟或定时的执行任务。

```
*/
public class TestScheduledThreadPool {

    public static void main(String[] args) throws Exception {
        ScheduledExecutorService pool = Executors.newScheduledThreadPool(5);

        for (int i = 0; i < 5; i++) {
            Future<Integer> result = pool.schedule(new Callable<Integer>(){

                @Override
                public Integer call() throws Exception {
                    int num = new Random().nextInt(100); //生成随机数
                    System.out.println(Thread.currentThread().getName() + " : "
+ num);

                    return num;
                }

            }, 1, TimeUnit.SECONDS);

            System.out.println(result.get());
        }

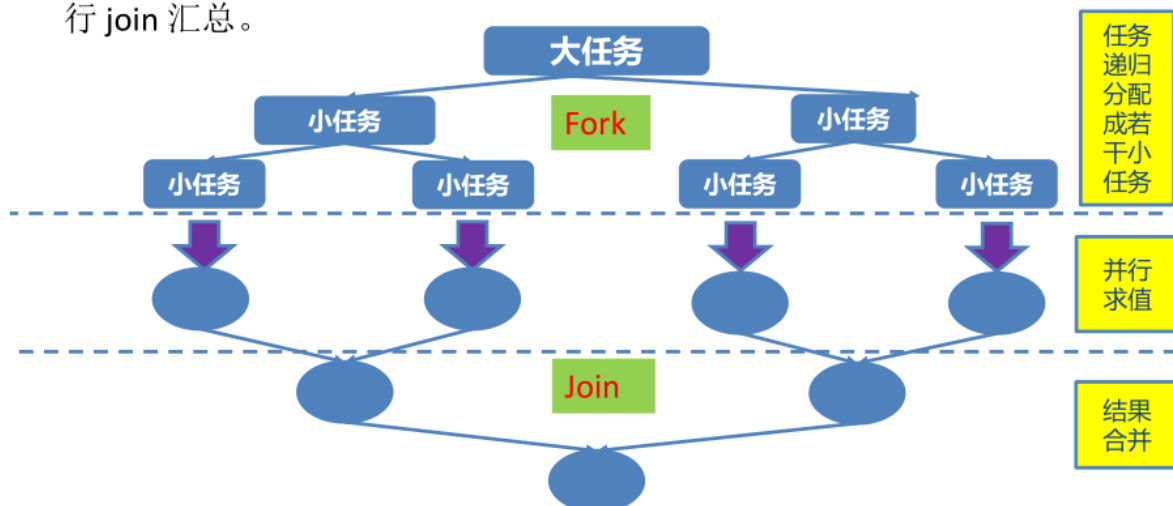
        pool.shutdown();
    }
}
```

```
pool-1-thread-1 : 75
75
pool-1-thread-1 : 11
11
pool-1-thread-2 : 54
54
pool-1-thread-1 : 24
24
pool-1-thread-3 : 83
83
```

## 十四、ForkJoinPool 分支/合并框架 工作窃取

### 1、Fork/Join框架

- Fork/Join 框架：就是在必要的情况下，将一个大任务，进行拆分(fork)成若干个小任务（拆到不可再拆时），再将一个个的小任务运算的结果进行 join 汇总。



## 2、Fork/Join框架与线程池的区别

- 采用“工作窃取”模式（work-stealing）：

当执行新的任务时它可以将其拆分分成更小的任务执行，并将小任务加到线程队列中，然后再从一个随机线程的队列中偷一个并把它放在自己的队列中。

- 相对于一般的线程池实现，fork/join框架的优势体现在对其中包含的任务的处理方式上.在一般的线程池中，如果一个线程正在执行的任务由于某些原因无法继续运行，那么该线程会处于等待状态。而在fork/join框架实现中，如果某个子问题由于等待另外一个子问题的完成而无法继续运行。那么处理该子问题的线程会主动寻找其他尚未运行的子问题来执行.这种方式减少了线程的等待时间，提高了性能。

```
package com.wl.java8.test;
import java.util.concurrent.RecursiveTask;
public class ForkJoinCalculate extends RecursiveTask<Long>{
    /**
     *
     */
    private static final long serialVersionUID = 13475679780L;
    private long start;
    private long end;
    private static final long THRESHOLD = 10000L; //临界值

    public ForkJoinCalculate(long start, long end) {
        this.start = start;
        this.end = end;
    }

    @Override
```

```

        protected Long compute() {
            long length = end - start;
            if (length <= THRESHOLD) {
                long sum = 0;
                for (long i = start; i <= end; i++) {
                    sum += i;
                }
                return sum;
            } else {
                long middle = (start + end) / 2;
                ForkJoinCalculate left = new ForkJoinCalculate(start, middle);
                left.fork(); // 拆分, 并将该子任务压入线程队列
                ForkJoinCalculate right = new ForkJoinCalculate(middle + 1, end);
                right.fork();
                return left.join() + right.join();
            }
        }
    }
}

/**
 * TestForkJoin 计算0-10000000000L的累加
 */
// java 7 ForkJoin框架
@Test
public void test131(){
    long start = System.currentTimeMillis();
    ForkJoinPool pool = new ForkJoinPool();
    ForkJoinTask<Long> task = new ForkJoinCalculate(0L, 10000000000L);
    long sum = pool.invoke(task);
    System.out.println(sum); // -5340232216128654848
    long end = System.currentTimeMillis();
    System.out.println("耗费的时间为: " + (end - start)); //3153-1923-2397
}

// 普通 for
@Test
public void test132(){
    long start = System.currentTimeMillis();
    long sum = 0L;
    for (long i = 0L; i <= 10000000000L; i++) {
        sum += i;
    }
    System.out.println(sum); // -5340232216128654848
    long end = System.currentTimeMillis();
    System.out.println("耗费的时间为: " + (end - start)); //3023-3086-4719
}

// java8 提供并行流,效率更大
@Test
public void test133(){
    long start = System.currentTimeMillis();
    Long sum = LongStream.rangeClosed(0L, 10000000000L)
        .parallel()
        .sum();
    System.out.println(sum); // -5340232216128654848
    long end = System.currentTimeMillis();
    System.out.println("耗费的时间为: " + (end - start)); //1294-2188-1883
}

```