带你一次搞定 Java 多线程（VI）

4 线程间的通信

4.4 ThreadLocal 的使用

除了控制资源的访问外，还可以通过增加资源来保证线程安全.

ThreadLocal 主要解决为每个线程绑定自己的值.

```
package com.wkcto.threadlocal;

/**
 * ThreadLocal 的基本使用
 */
public class Test01 {
    //定义 ThreadLocal 对象
    static  ThreadLocal threadLocal = new ThreadLocal();
    //定义线程类
    static  class  Subthread extends Thread{
        @Override
        public void run() {
            for (int i = 0; i < 20; i++) {
                //设置线程关联的的值
                threadLocal.set( Thread.currentThread().getName() + " - " + i);
                //调用 get()方法读取关联的值
                System.out.println(Thread.currentThread().getName() + " value = " +
threadLocal.get());
            }
        }
    }
    public static void main(String[] args) {
        Subthread t1 = new Subthread();
        Subthread t2 = new Subthread();
        t1.start();
        t2.start();

    }
}
```

```
package com.wkcto.threadlocal;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 *  在多线程环境中,把字符串转换为日期对象,多个线程使用同一个 SimpleDateFormat 对象
可能会产生线程安全问题,有异常
 *  为每个线程指定自己的 SimpleDateFormat 对象,使用 ThreadLocal
 */
public class Test02 {
    //定义 SimpleDateFormat 对象,该对象可以把字符串转换为日期
//      private static SimpleDateFormat sdf = new SimpleDateFormat("yyyy 年 MM 月 dd 日
HH:mm:ss");
    static ThreadLocal<SimpleDateFormat> threadLocal = new ThreadLocal<>();
    //定义 Runnable 接口的实现类
    static class ParseDate implements Runnable{
        private    int i = 0 ;
        public ParseDate(int i) {
            this.i = i;
        }

        @Override
        public void run() {
            try {
                String text = "2068 年 11 月 22 日  08:28:" + i%60;          //构建日期字符串
//                Date date = sdf.parse(text);              //把字符串转换为日期
                //先判断当前线程是否有 SimpleDateFormat 对象,如果当前线程没有 SimpleDateFormat 对象就创建一个,如果有就直接使用
                if (threadLocal.get() == null){
                    threadLocal.set(new SimpleDateFormat("yyyy 年 MM 月 dd 日 HH:mm:ss"));
                }
                Date date = threadLocal.get().parse(text);
                System.out.println(i + " -- " + date);
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        //创建 100 个线程
        for (int i = 0; i < 100; i++) {
            new Thread(new ParseDate(i)).start();
        }
    }
}
```

```
package com.wkcto.threadlocal;

import java.util.Date;
import java.util.Random;

/**
 * ThreadLocal 初始值，定义 ThreadLocal 类的子类,在子类中重写 initialValue()方法指定初始值,再第一次调用 get()方法不会返回 null
 */
public class Test03 {
    //1)  定义 ThreadLocal 的子类
    static class SubThreadLocal extends ThreadLocal<Date>{
        //  重写 initialValue 方法,设置初始值
        @Override
        protected Date initialValue() {
//                  return new Date();   //把当前日期设置为初始化
            return    new Date(System.currentTimeMillis() - 1000*60*15);
        }
    }

    //定义 ThreadLocal 对象
//      static ThreadLocal threadLocal = new ThreadLocal();
    //直接使用自定义的 SubThreadLocal 对象
    static SubThreadLocal threadLocal = new SubThreadLocal();

    //定义线程类
    static class SubThread extends    Thread{
        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                //第一次调用 threadLocal 的 get()方法会返回 null
                System.out.println("---------" + Thread.currentThread().getName() + " value="
+ threadLocal
                .get());
                //如果没有初始值就设置当前日期
                if ( threadLocal.get() == null ){
                    System.out.println("*****************");
                    threadLocal.set(new Date());
                }
                try {
                    Thread.sleep(new Random().nextInt(500));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
```

```
                        }
                }
        }

        public static void main(String[] args) {
                SubThread t1 = new SubThread();
                t1.start();
                SubThread t2 = new SubThread();
                t2.start();
        }
}
```

5 Lock 显示锁

在 JDK5 中增加了 Lock 锁接口，有 ReentrantLock 实现类,ReentrantLock 锁称为可重入锁，它功能比 synchronized 多.

5.1 锁的可重入性

锁的可重入是指,当一个线程获得一个对象锁后,再次请求该对象锁时是可以获得该对象的锁的.

```
package com.wkcto.lock.reentrant;

/**
 *  演示锁的可重入性
 */
public class Test01 {
        public synchronized void sm1(){

                System.out.println("同步方法 1");
                //线程执行 sm1()方法,默认 this 作为锁对象,在 sm1()方法中调用了 sm2()方法,注意
当前线程还是持有 this 锁对象的
                //sm2()同步方法默认的锁对象也是 this 对象，要执行 sm2()必须先获得 this 锁对象,
当前 this 对象被当前线程持有,可以 再次获得 this 对象，这就是锁的可重入性. 假设锁不可
重入的话,可能会造成死锁
                sm2();
        }

        private synchronized void sm2() {
                System.out.println("同步方法 2");
                sm3();
        }

        private synchronized void sm3() {
                System.out.println("同步方法 3");
        }

        public static void main(String[] args) {
                Test01 obj = new Test01();
                new Thread(new Runnable() {
                        @Override
                        public void run() {
                                obj.sm1();
                        }
                }).start();
        }
}
```

5.2 ReentrantLock

5.2.1 ReentrantLock 的基本使用

调用 lock()方法获得锁，调用 unlock()释放锁

```
package com.wkcto.lock.reentrant;

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

/**
 * Lock 锁的基本使用
 */
public class Test02 {
    //定义显示锁
    static Lock lock = new ReentrantLock();
    //定义方法
    public static void sm(){
        //先获得锁
        lock.lock();
        //for 循环就是同步代码块
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + " -- " + i);
        }
        //释放锁
        lock.unlock();
    }

    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                sm();
            }
        };
        //启动三个线程
        new Thread(r).start();
        new Thread(r).start();
        new Thread(r).start();
    }
}
```

```
package com.wkcto.lock.reentrant;

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

/**
 * 使用 Lock 锁同步不同方法中的同步代码块
 */
public class Test03 {
    static Lock lock = new ReentrantLock();         //定义锁对象
    public static void sm1(){
        //经常在 try 代码块中获得 Lock 锁，在 finally 子句中释放锁
        try {
            lock.lock();          //获得锁
            System.out.println(Thread.currentThread().getName() + "-- method 1 -- " + System.currentTimeMillis() );
            Thread.sleep(new Random().nextInt(1000));
            System.out.println(Thread.currentThread().getName() + "-- method 1 -- " + System.currentTimeMillis() );
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();          //释放锁
        }
    }

    public static void sm2(){
        try {
            lock.lock();            //获得锁
            System.out.println(Thread.currentThread().getName() + "-- method 22 -- " +
System.currentTimeMillis() );
            Thread.sleep(new Random().nextInt(1000));
            System.out.println(Thread.currentThread().getName() + "-- method 22 -- " +
System.currentTimeMillis() );
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();              //释放锁
        }
    }

    public static void main(String[] args) {
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                sm1();
            }
        };
        Runnable r2 = new Runnable() {
            @Override
            public void run() {
                sm2();
            }
        };

        new Thread(r1).start();
        new Thread(r1).start();
        new Thread(r1).start();
        new Thread(r2).start();
        new Thread(r2).start();
        new Thread(r2).start();
    }
}
```

5.2.2 ReentrantLock 锁的可重入性

```
package com.wkcto.lock.reentrant;

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

/**
 *   ReentrantLock 锁的可重入性
 */
public class Test04 {
    static class Subthread extends Thread{
        private static Lock lock = new ReentrantLock();         //定义锁对象

        public static int num = 0;        //定义变量
        @Override
        public void run() {
            for (int i = 0; i <10000 ; i++) {
                try {
                    //可重入锁指可以反复获得该锁
                    lock.lock();
                    lock.lock();
                    num++;
                }finally {
                    lock.unlock();
                    lock.unlock();
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Subthread t1 = new Subthread();
        Subthread t2 = new Subthread();
        t1.start();
        t2.start();
        t1.join();
        t2.join();
```

```
          System.out.println( Subthread.num );
     }
}
```

5.2.3 lockInterruptibly()方法

lockInterruptibly() 方法的作用:如果当前线程未被中断则获得锁,如果当前线程被中断则出现异常.

```
package com.wkcto.lock.reentrant;

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

/**
 * lockInterruptibly() 方法的作用:如果当前线程未被中断则获得锁,如果当前线程被中断则
出现异常.
 */
public class Test05 {
     static class Servier{
          private Lock lock = new ReentrantLock();          //定义锁对象
          public void serviceMethod(){
               try {
//                    lock.lock();          //获得锁定,即使调用了线程的 interrupt()方法,也没
有真正的中断线程
                    lock.lockInterruptibly();     //如果线程被中断了,不会获得锁,会产生异常
                    System.out.println(Thread.currentThread().getName() + "-- begin lock");
                    //执行一段耗时的操作
                    for (int i = 0; i < Integer.MAX_VALUE; i++) {
                         new StringBuilder();
```

```
                }
                System.out.println( Thread.currentThread().getName() + " -- end lock");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {

                System.out.println( Thread.currentThread().getName() + " *****  释放锁");

                lock.unlock();          //释放锁

            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Servier    s = new Servier();
        Runnable r = new Runnable() {
            @Override
            public void run() {
                s.serviceMethod();
            }
        };
        Thread t1 = new Thread(r);
        t1.start();

        Thread.sleep(50);
        Thread t2 = new Thread(r);
        t2.start();
        Thread.sleep(50);

        t2.interrupt();         //中断 t2 线程

    }
}
```

对于 synchronized 内部锁来说,如果一个线程在等待锁,只有两个结果:要么该线程获得锁继续执行;要么就保持等待.

对于 ReentrantLock 可重入锁来说,提供另外一种可能,在等待锁的

过程中,程序可以根据需要取消对锁的请求.

```
package com.wkcto.lock.reentrant;

import com.wkcto.pipestream.Test2;

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

/**
 *   通过 ReentrantLock 锁的 lockInterruptibly()方法避免死锁的产生
 */
public class Test06 {
    static   class   IntLock implements Runnable{
        //创建两个 ReentrantLock  锁对象
        public static ReentrantLock lock1 = new ReentrantLock();
        public static ReentrantLock lock2 = new ReentrantLock();
        int lockNum;            //定义整数变量,决定使用哪个锁

        public IntLock(int lockNum) {
            this.lockNum = lockNum;
        }

        @Override
        public void run() {
            try {
                if ( lockNum % 2 == 1){             //奇数,先锁 1,再锁 2
                    lock1.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName() + "获得锁 1,还需
要获得锁 2");
                    Thread.sleep(new Random().nextInt(500));
                    lock2.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName() + "同时获得了锁
1 与锁 2....");
                }else {         //偶数,先锁 2,再锁 1
                    lock2.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName() + "获得锁 2,还需
要获得锁 1");
                    Thread.sleep(new Random().nextInt(500));
                    lock1.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName() + "同时获得了锁
1 与锁 2....");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if ( lock1.isHeldByCurrentThread())             //判断当前线程是否持有该锁
                    lock1.unlock();
                if (lock2.isHeldByCurrentThread())
                    lock2.unlock();
                System.out.println( Thread.currentThread().getName() + "线程退出");
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        IntLock intLock1 = new IntLock(11);
        IntLock intLock2    = new IntLock(22);

        Thread t1 = new Thread(intLock1);
        Thread t2 = new Thread(intLock2);
        t1.start();
        t2.start();

        //在 main 线程,等待 3000 秒,如果还有线程没有结束就中断该线程
        Thread.sleep(3000);
        //可以中断任何一个线程来解决死锁, t2 线程会放弃对锁 1 的申请,同时释放锁 2, t1 线程会完成它的任务
        if (t2.isAlive()){ t2.interrupt();}
    }
}
```

5.2.4 tryLock()方法

tryLock(long time, TimeUnit unit) 的作用在给定等待时长内锁没有被另外的线程持有,并且当前线程也没有被中断,则获得该锁.通过该方法可以实现锁对象的限时等待.

```
package com.wkcto.lock.reentrant;

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

/**
 *tryLock(long time, TimeUnit unit) 的基本使用
 */
public class Test07 {
    static class TimeLock implements Runnable{
        private static ReentrantLock lock = new ReentrantLock();      //定义锁对象

        @Override
        public void run() {
            try {
                if ( lock.tryLock(3, TimeUnit.SECONDS) ){          //获得锁返回 true
                    System.out.println(Thread.currentThread().getName() + "获得锁,执行耗
时任务");
//                    Thread.sleep(4000);          //假设 Thread-0 线程先持有锁,完成任
务需要 4 秒钟,Thread-1 线程尝试获得锁,Thread-1 线程在 3 秒内还没有获得锁的话,Thread-1
线程会放弃
                    Thread.sleep(2000);          //假设 Thread-0 线程先持有锁,完成任
务需要 2 秒钟,Thread-1 线程尝试获得锁,Thread-1 线程会一直尝试,在它约定尝试的 3 秒内可
以获得锁对象
                }else {          //没有获得锁
                    System.out.println(Thread.currentThread().getName() + "没有获得锁");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if (lock.isHeldByCurrentThread()){
                    lock.unlock();
                }
            }
        }
    }

    public static void main(String[] args) {
        TimeLock timeLock = new TimeLock();

        Thread t1 = new Thread(timeLock);
        Thread t2 = new Thread(timeLock);
        t1.start();
        t2.start();
    }
}
```

tryLock()仅在调用时锁定未被其他线程持有的锁,如果调用方法时,

锁对象对其他线程持有,则放弃. 调用方法尝试获得没,如果该锁没有被其他线程占用则返回 true 表示锁定成功; 如果锁被其他线程占用则返回 false,不等待.

```
package com.wkcto.lock.reentrant;

import java.util.concurrent.locks.ReentrantLock;

/**
 *tryLock()
 *   当锁对象没有被其他线程持有的情况下才会获得该锁定
 */
public class Test08 {
    static class Service{
        private ReentrantLock lock = new ReentrantLock();
        public void serviceMethod(){
            try {
                if (lock.tryLock()){
                    System.out.println(Thread.currentThread().getName() + "获得锁定");
                    Thread.sleep(3000);     //模拟执行任务的时长
                }else {
                    System.out.println(Thread.currentThread().getName() + "没有获得锁定
");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if (lock.isHeldByCurrentThread()){
                    lock.unlock();
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Service service = new Service();
        Runnable r = new Runnable() {
            @Override
            public void run() {
                service.serviceMethod();
            }
        };

        Thread t1 = new Thread(r);
        t1.start();
        Thread.sleep(50);          //睡眠 50 毫秒,确保 t1 线程锁定
        Thread t2 = new Thread(r);
        t2.start();
    }
}
```

```
package com.wkcto.lock.reentrant;

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

/**
 * 使用 tryLock()可以避免死锁
 */
public class Test09 {
    static class   IntLock implements Runnable{
        private static ReentrantLock lock1 = new ReentrantLock();
        private static ReentrantLock lock2 = new ReentrantLock();
        private int lockNum;           //用于控制锁的顺序


        public IntLock(int lockNum) {
            this.lockNum = lockNum;
        }

        @Override
        public void run() {
            if ( lockNum % 2 == 0 ){      //偶数先锁 1,再锁 2
                while (true){
                    try {
                        if (lock1.tryLock()){
                            System.out.println(Thread.currentThread().getName() + "获得
锁 1，还想获得锁 2");
                            Thread.sleep(new Random().nextInt(100));
                            try {
                                if (lock2.tryLock()){
System.out.println(Thread.currentThread().getName() + "同时获得锁 1 与锁 2 ----完成任务了");
                                    return;          //结束 run()方法执行,即当前线程
结束
                                }
                            } finally {
                                if (lock2.isHeldByCurrentThread()){
                                    lock2.unlock();
                                }
                            }
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        if (lock1.isHeldByCurrentThread()){
                            lock1.unlock();
                        }
                    }
                }
            }else {       //奇数就先锁 2,再锁 1
                while (true){
                    try {
                        if (lock2.tryLock()){
                            System.out.println(Thread.currentThread().getName() + "获得
锁 2，还想获得锁 1");
                            Thread.sleep(new Random().nextInt(100));
                            try {
                                if (lock1.tryLock()){

System.out.println(Thread.currentThread().getName() + "同时获得锁 1 与锁 2 ----完成任务了");
                                    return;          //结束 run()方法执行,即当前线程

结束
                                }
                            } finally {
                                if (lock1.isHeldByCurrentThread()){
                                    lock1.unlock();
                                }
                            }
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        if (lock2.isHeldByCurrentThread()){
                            lock2.unlock();
                        }
                    }
                }
            }
        }
    }
    public static void main(String[] args) {
        IntLock intLock1 = new IntLock(11);
        IntLock intLock2 = new IntLock(22);
        Thread t1 = new Thread(intLock1);
        Thread t2 = new Thread(intLock2);
        t1.start();
        t2.start();
        //运行后,使用 tryLock()尝试获得锁,不会傻傻的等待,通过循环不停的再次尝试,如果

等待的时间足够长,线程总是会获得想要的资源
    }
}
```