第 3 章

多线程特性

本章介绍线程之间的协作，synchronized 对象锁、线程死锁的产生、Object 对象监视器、线程等待机制、线程唤配机制、线程可见性重排序、线程生命周期状态。

3.1 引出 synchronized 对象锁

8min

11min

前 2 章的课程详细介绍了 Thread 类、ThreadGroup 类，以及它们常用的方法和它们之间的关联关系。在并发环境下线程之间的协作是必不可少的，在此前的课程中基本上没有运用线程之间的协作，线程对象. join()方法是线程协作的其中一种方式，但是它的使用场景较固定，不是很灵活。

在介绍 synchronized 对象锁之前，先参考一个单线程的案例，代码如下：

```
//第 3 章/one/CooperationRunnable.java
public class CooperationRunnable implements Runnable{

    public int num;
    @Override
    public void run() {
        for(int x = 0;x < 100000;x++){
            num++;
        }
    }

}
```

OneMain 类主方法，代码如下：

```
//第 3 章/one/OneMain.java
public class OneMain {

    public static void main(String[] args) throws InterruptedException {
        CooperationRunnable cooperationRunnable = new CooperationRunnable();
        Thread thread1 = new Thread(cooperationRunnable, "A");
```

```
            thread1.start();              //启动线程
            thread1.join();               //等待此线程对象销毁
            System.out.println(cooperationRunnable.num);
        }

    }
```

执行结果如下：

```
100000
```

单线程的执行结果，肯定是没有问题的，修改 OneMain 类，代码如下：

```
//第 3 章/one/OneMain.java
public class OneMain {

    public static void main(String[] args) throws InterruptedException {
        CooperationRunnable cooperationRunnable = new CooperationRunnable();

        Thread thread1 = new Thread(cooperationRunnable, "A");
        thread1.start();                  //启动线程

        Thread thread2 = new Thread(cooperationRunnable, "B");
        thread2.start();                  //启动线程

        Thread thread3 = new Thread(cooperationRunnable, "C");
        thread3.start();                  //启动线程

        thread1.join();                   //等待此线程对象销毁
        thread2.join();                   //等待此线程对象销毁
        thread3.join();                   //等待此线程对象销毁
        System.out.println(cooperationRunnable.num);
    }

}
```

多次执行代码后，观察结果。会发现每次的结果基本上都不相同，主要原因是 num++并非一个原子操作，如图 3-1 所示。

在多线程并发执行的情况下，可能出现例如 A 执行线程获得 num 值后，丢失了 CPU 的调度权，等到再次获得 CPU 调度权时，由于多线程并发执行的原因，此时的 num 值可能已经发生了变化，但是 A 执行线程使用的 num 值还是之前获得的旧值，在这种情况下就造成了数据的脏读，A 执行线程继续把 num 值增加 1，然后设置新的 num 值，这时又造成了数据的脏写或者覆盖。

可以使用 synchronized 对象锁来解决此问题，修改 CooperationRunnable 类，代码如下：

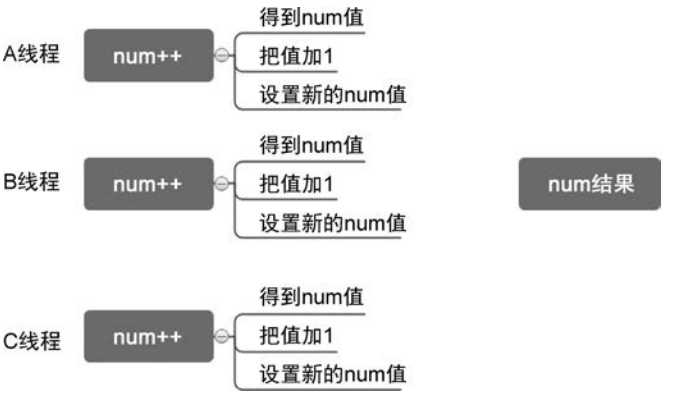

图 3-1　并发产生的脏读

```
//第 3 章/one/CooperationRunnable.java
public class CooperationRunnable implements Runnable{

    public int num;
    @Override
    public void run() {
        for(int x = 0;x < 100000;x++){
            synchronized (this){ //同一个对象锁,同时只能有一个线程进入执行
                num++;
            }
        }
    }

}
```

运行 OneMain 类主方法,执行结果如下:

```
300000
```

synchronized 对象锁,在锁同一个对象的情况下,同时只能有一个线程进入执行,其余阻塞等待拿锁,拿锁的过程是非公平的,并不是说阻塞等待时间越长拿锁概率越高,如图 3-2 所示。

图 3-2　阻塞等待拿锁

在使用 synchronized 对象锁时，需要注意锁的力度越小范围越好，毕竟是希望多线程并发执行提高效率，如果锁的范围过大，则会降低多线程并发执行的效果。

3.2 synchronized 对象锁

synchronized 对象锁按对象类型可划分为两类，即标准对象、class 对象。

3.2.1 标准对象

使用方式，可以把 synchronized 关键字标记在对象的方法上，此方法内所有代码块属于锁范围，也可以使用 synchronized(标准对象)直接锁定一个代码块区域，代码如下：


14min


7min


8min


6min

```
//第 3 章/two/CooperationTwo.java
public class CooperationTwo implements Runnable{
    public int num;

    @Override
    public void run() {
        for(int x = 0;x < 100000;x++){
            synchronized (this){ //同一个锁对象,同时只能有一个线程进入执行
                num++;
            }
        }
    }

    public void run1(){
        for(int x = 0;x < 100000;x++){
            run2(); //同一个锁对象,同时只能有一个线程进入执行
        }
    }
    //synchronized 关键字标记在对象的方法上
    public synchronized void run2(){
      /* synchronized (this){
            num++;
        } */
        num++;
    }
}
```

TwoMain 类主方法，代码如下：

```
//第 3 章/two/TwoMain.java
public class TwoMain {
    public static void main(String[] args) throws InterruptedException {
        CooperationTwo cooperationTwo = new CooperationTwo();
        Thread thread1 = new Thread(cooperationTwo, "A");
        thread1.start();              //启动线程
```

```
        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                cooperationTwo.run1();
            }
        });
        thread2.start();            //启动线程

        for(int x = 0;x < 100000;x++){
            synchronized (cooperationTwo){
                                    //同一个锁对象,同时只能有一个线程进入执行
                cooperationTwo.num++;
            }
        }
        thread1.join();             //等待 thread1 线程对象销毁
        thread2.join();             //等待 thread2 线程对象销毁

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

执行结果如下：

```
300000
```

注意：还是同一个核心概念，只是使用方式不同。synchronized 对象锁，在锁同一个对象的情况下，同时只能有一个线程进入执行，其余阻塞等待拿锁，拿锁的过程是非公平的，并不是说阻塞等待时间越长，拿锁概率越高。

3.2.2　class 对象

使用方式，可以把 synchronized 关键字标记在静态方法上，此方法内的所有代码块属于锁范围，也可以使用 synchronized(class 对象)直接锁定一个代码块区域，代码如下：

```
//第 3 章/two/CooperationTwo.java
public class CooperationTwo implements Runnable{
    public static int num;

    @Override
    public void run() {
        for(int x = 0;x < 100000;x++){
            synchronized (CooperationTwo.class){
                    //同一个锁对象,同时只能有一个线程进入执行
                num++;
            }
        }
```

```
    }

    public void run1(){
        for(int x = 0;x < 100000;x++){
            run2(); //同一个锁对象,同时只能有一个线程进入执行
        }
    }

    public synchronized static void run2(){
        num++;
    }
}
```

TwoMain 类主方法，代码如下：

```
//第 3 章/two/TwoMain.java
public class TwoMain {
    public static void main(String[] args) throws InterruptedException {
        CooperationTwo cooperationTwo = new CooperationTwo();
        Thread thread1 = new Thread(cooperationTwo, "A");
        thread1.start();                //启动线程

        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                cooperationTwo.run1();
            }
        },"B");
        thread2.start();                //启动线程

        for(int x = 0;x < 100000;x++){
            synchronized (cooperationTwo){
                                        //同一个锁对象,同时只能有一个线程进入执行
                CooperationTwo.num++;
            }
        }

        thread1.join();                 //等待 thread1 线程对象销毁
        thread2.join();                 //等待 thread2 线程对象销毁

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

运行 TwoMain 类主方法，并观察执行结果，会发现结果基本上每次都不同，这是因为上面的代码有 3 个线程并行，使用了两把锁，A、B 线程使用了 CooperationTwo.class 对象锁，但是主线程使用的是 cooperationTwo 标准对象锁。修改 TwoMain 类，代码如下：

```
//第 3 章/two/TwoMain.java
public class TwoMain {
```

```
    public static void main(String[] args) throws InterruptedException {
        CooperationTwo cooperationTwo = new CooperationTwo();
        Thread thread1 = new Thread(cooperationTwo, "A");
        thread1.start();                    //启动线程

        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                cooperationTwo.run1();
            }
        },"B");
        thread2.start();                    //启动线程

        for(int x = 0;x < 100000;x++){
            synchronized (CooperationTwo.class){
                                            //同一个锁对象,同时只能有一个线程进入执行
                CooperationTwo.num++;
            }
        }

        thread1.join();                     //等待 thread1 线程对象销毁
        thread2.join();                     //等待 thread2 线程对象销毁

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

执行结果如下：

```
300000
```

3.2.3　锁特性

synchronized 对象锁具有的特性：异常自动释放锁、可重入、不响应中断、非公平锁。

1. 异常自动释放锁

拿锁、释放锁是一个关联性很强的逻辑，在正常情况下，拿锁后进入代码块执行，出了代码块就会自动释放锁。如果出现异常情况，则锁不能自动释放是一个很严重的问题，可以想象等待拿锁的线程永远拿不到锁，相当于这一块业务无法处理而造成线程阻塞，甚至引起整个应用软件的瘫痪。好在 synchronized 对象锁有异常自动释放的特性，代码如下：

```
//第 3 章/two/TwoMain2.java
public class TwoMain2 {
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                synchronized (this){
```

```
                    System.out.println(Thread.currentThread().getName());
                    try {
                        Thread.sleep(6000);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    int num = 10 / 0;              //模拟异常情况
                    System.out.println(Thread.currentThread().getName()
                                                + " - end");
                }
            }
        };
        Thread thread = new Thread(runnable,"A");
        thread.start();                            //启动线程

        Thread.sleep(100);
        synchronized (runnable){
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```

执行结果如下：

```
A
main
Exception in thread "A" java.lang.ArithmeticException: / by zero
    at cn.kungreat.book.three.two.TwoMain2$1.run(TwoMain2.java:15)
    at java.base/java.lang.Thread.run(Thread.java:833)
```

2. 可重入

相同的对象锁，在持有锁时可以直接进入，代码如下：

```
//第 3 章/two/TwoMain3.java
public class TwoMain3 {

    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new Runnable() {

            @Override
            public void run() {
                synchronized (this){
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    System.out.println(Thread.currentThread().getName());
                    reentry();              //可重入，同一个锁对象
                }
            }
```

```
            public synchronized void reentry(){
                System.out.println(Thread.currentThread().getName()
                                        + ":reentry");
            }
        };
        Thread thread = new Thread(runnable,"A");
        thread.start();

        Thread.sleep(100);              //睡眠 100ms
        synchronized (runnable){
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```

执行结果如下：

```
A
A:reentry
main
```

3. 不响应中断

synchronized 对象锁不响应中断。如果要用到中断，则需要自己设计实现。

4. 非公平锁

由系统调度分配资源，并不是等待拿锁时间越久拿锁概率越高，所以说它是非公平的。

3.3 线程死锁的产生

13min

由于 synchronized 对象锁的特性，以下代码会产生死锁，代码如下：

```
//第 3 章/three/ThreeMain.java
public class ThreeMain {

    public static void main(String[] args) {
        DeadRunnable deadRunnable = new DeadRunnable();
        Thread thread = new Thread(deadRunnable,"A");
        thread.start();

        deadRunnable.deadLock();
    }

    static class DeadRunnable implements Runnable{
        private Object objA = new Object();                    //objA 对象锁
        private Object objB = new Object();                    //objB 对象锁

        @Override
```

```
            public void run() {
                synchronized (objA){
                    System.out.println(Thread.currentThread().getName());
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    deadLock();
                }
            }

            public void deadLock(){
                synchronized (objB){
                    System.out.println(Thread.currentThread().getName());
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    run();
                }
            }
        }
    }
```

注意：以上代码只为了演示死锁，其本身也存在相互调用的缺陷。A 线程拿锁 objA 后再去调 deadLock()方法，阻塞等待拿锁 objB。main 线程拿锁 objB 后再去调 run()方法，阻塞等待拿锁 objA。

可以通过官方提供的工具 JConsole、jstack 检测死锁。

3.3.1 JConsole

在命令行直接运行 JConsole，如图 3-3 所示。

选择自己要查看的类，如图 3-4 所示。

连接以后，选择线程，并单击“检测死锁”按钮，如图 3-5 所示。

查看死锁情况，如图 3-6 所示。

3.3.2 jstack

通过 jps 查看 pid 信息，如图 3-7 所示。

使用 jstack-l pid，选择自己要查看的 pid 信息，如图 3-8 所示。

jstack 会打印出 pid 下所有的堆栈信息，内容比较多，这里关注核心的点，如图 3-9 所示。

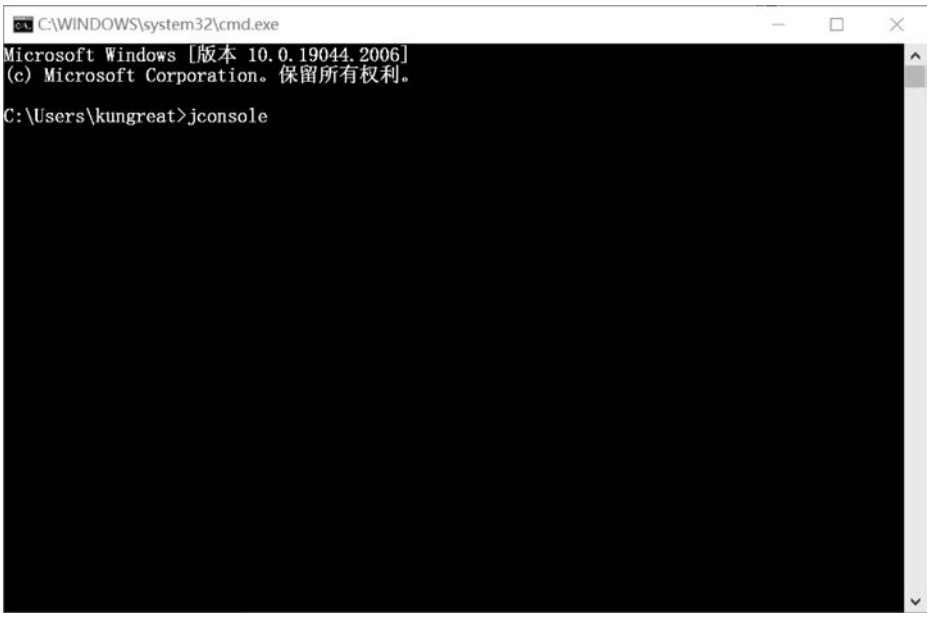

图 3-3　命令行

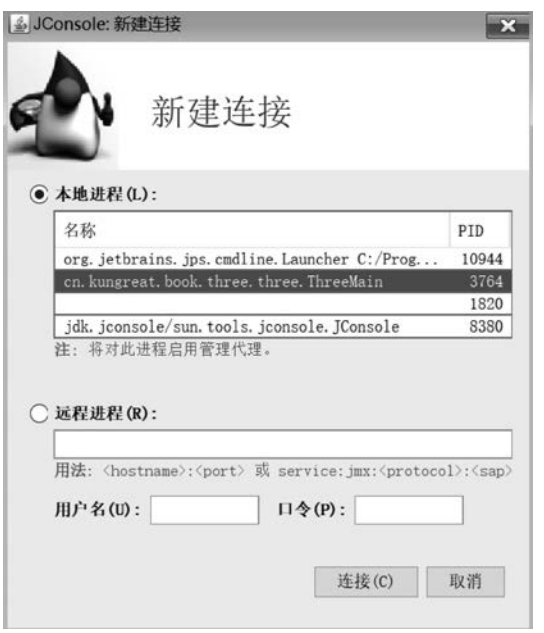

图 3-4　JConsole 首页

图 3-5　检测死锁

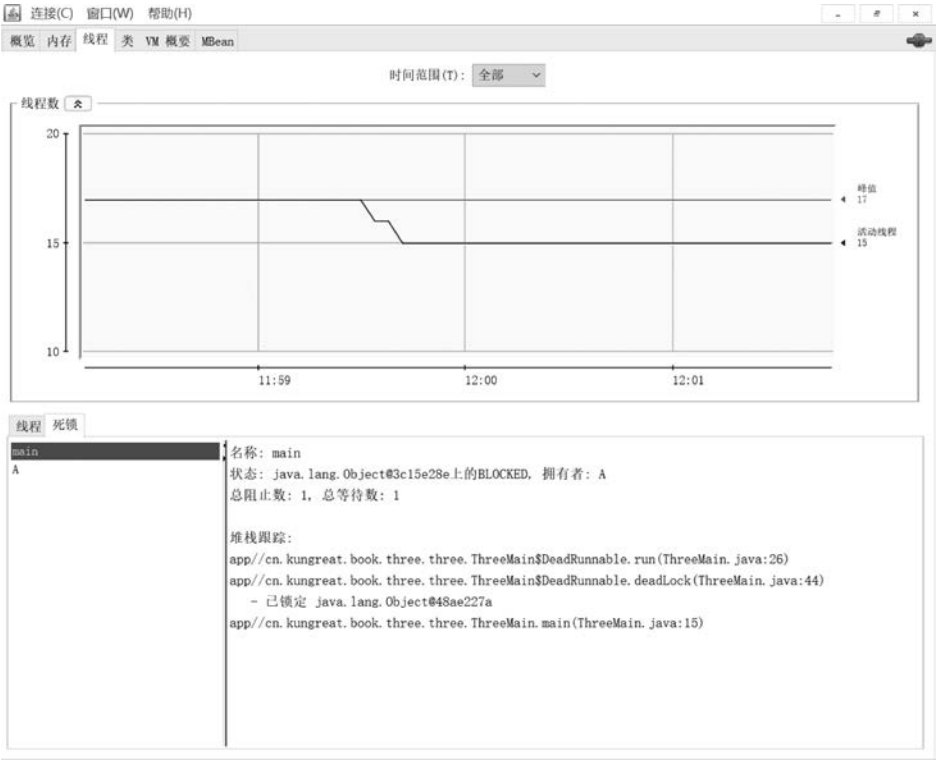

图 3-6　查看死锁

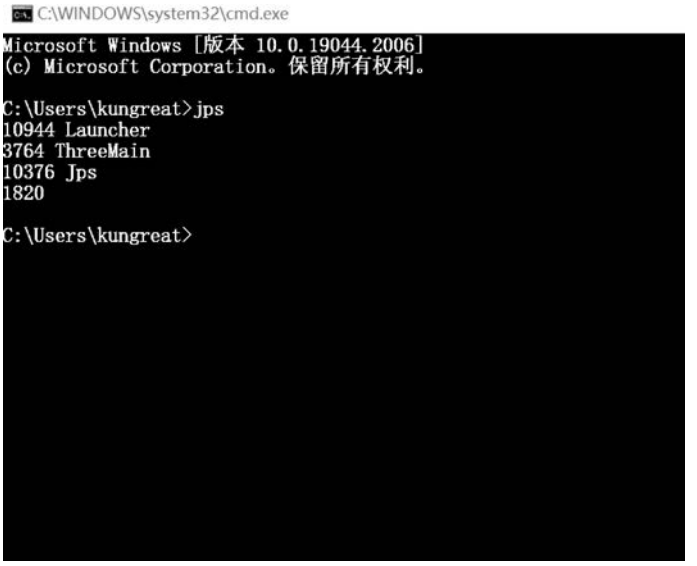

图 3-7　查看 pid

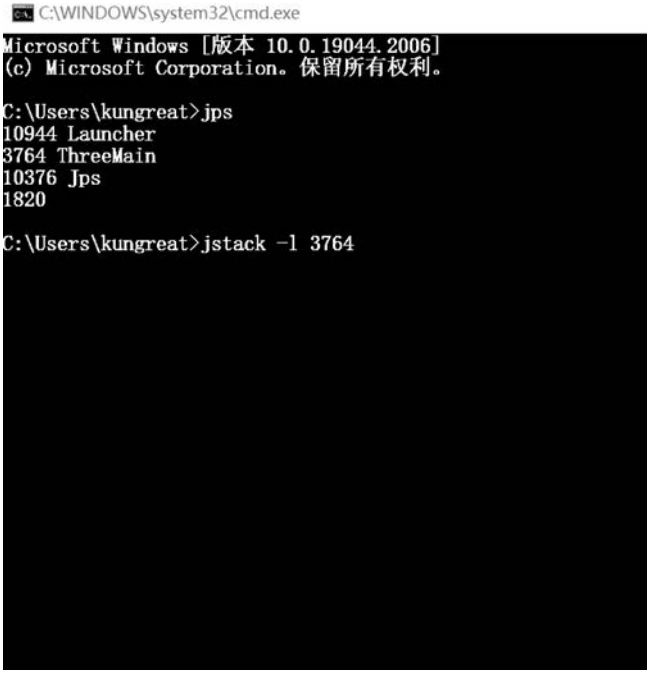

图 3-8　jstack 查看 pid

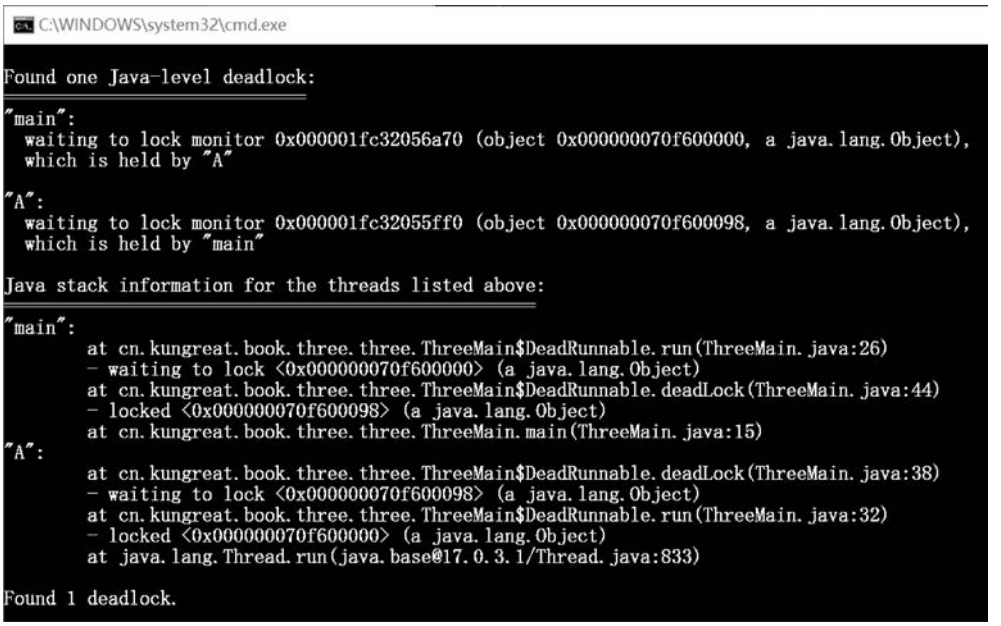

图 3-9　堆栈信息

3.4　对象监视器

对象监视器就是 synchronized 对象锁中锁定的那个对象，官方文档将其称为对象监视器。对象监视器提供了几个核心的功能，如 wait、notify、notifyAll。

3.4.1　wait()

使当前执行线程阻塞等待，直到它被唤醒或中断，具有释放当前锁的特性。如果没有被唤醒或中断，则一直阻塞等待，代码如下：

14min

```
//第 3 章/four/FourMain.java
public class FourMain {
    public static void main(String[] args) throws Exception {
        MonitorRunnable monitorRunnable = new MonitorRunnable();
        Thread thread = new Thread(monitorRunnable, "A");
        thread.start();                 //启动线程
        Thread.sleep(100);              //睡眠 100ms
        synchronized (monitorRunnable){
            System.out.println(Thread.currentThread().getName());
        }
        System.out.println(Thread.currentThread().getName() + ":end");
    }

    static class MonitorRunnable implements Runnable{
```

```
        @Override
        public void run() {
            synchronized (this){
                System.out.println(Thread.currentThread().getName());
                try {
                    Thread.sleep(5000);        //睡眠 5s
                    this.wait();   //使当前线程阻塞等待,直到它被唤醒或中断[释放当前锁]
                    System.out.println("被唤醒了");
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            System.out.println(Thread.currentThread().getName() + ":end");
        }
    }
}
```

执行结果如下：

```
A
main
main:end
```

注意：以上代码 A 线程将会一直阻塞等待。

3.4.2　wait(long timeoutMillis)

使当前执行线程阻塞等待,直到它被唤醒、中断或者超过最大等待时间,具有释放当前锁的特性,接收 long 入参,作为最大等待时间的毫秒数,代码如下：

```
//第 3 章/four/FourMain.java
public class FourMain {
    public static void main(String[] args) throws InterruptedException {
        MonitorRunnable monitorRunnable = new MonitorRunnable();
        Thread thread = new Thread(monitorRunnable, "A");
        thread.start();
        Thread.sleep(100);              //睡眠 100ms
        synchronized (monitorRunnable){
            System.out.println(Thread.currentThread().getName());
        }
        System.out.println(Thread.currentThread().getName() + ":end");
    }

    static class MonitorRunnable implements Runnable{

        @Override
        public void run() {
            synchronized (this){
```

```
                System.out.println(Thread.currentThread().getName());
                try {
                    Thread.sleep(5000);            //睡眠 5s
                    this.wait(5000);
                    Thread.sleep(1000);            //睡眠 1s
                    System.out.println("被唤醒了");
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            System.out.println(Thread.currentThread().getName() + ":end");
        }
    }
}
```

在经过一定量的实时时间后，A 线程将正常结束生命周期，执行结果如下：

```
A
main
main:end
被唤醒了
A:end
```

3.4.3 notify()

6min

唤醒正在等待此对象监视器上的单个执行线程，代码如下：

```
//第 3 章/four/FourMain.java
public class FourMain {

    public static void main(String[] args) throws InterruptedException {
        MonitorRunnable monitorRunnable = new MonitorRunnable();
        Thread thread = new Thread(monitorRunnable, "A");
        thread.start();
        Thread.sleep(100);                    //睡眠 100ms
        synchronized (monitorRunnable){
            System.out.println(Thread.currentThread().getName());
            monitorRunnable.notify();        //唤醒正在等待此对象监视器上的单个线程
            System.out.println("唤醒正在等待此对象监视器上的单个线程");
        }

    }

    static class MonitorRunnable implements Runnable{

        @Override
        public void run() {
            synchronized (this){
                System.out.println(Thread.currentThread().getName());
```

```
                try {
                    Thread.sleep(5000);
                    this.wait();
                    System.out.println("被唤醒了");
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            System.out.println(Thread.currentThread().getName() + ":end");
        }
    }
}
```

执行结果如下：

```
A
main
唤醒正在等待此对象监视器上的单个线程
被唤醒了
A:end
```

注意：观察上方输出结果的第 3 行和第 4 行，并思考它们之间的先后顺序是怎么产生的。

当同一个对象监视器有多个执行线程 wait()时，只会唤醒其中的一个执行线程，代码如下：

```
//第 3 章/four/FourMain.java
public class FourMain {

    public static void main(String[] args) throws InterruptedException {
        MonitorRunnable monitorRunnable = new MonitorRunnable();
        Thread thread = new Thread(monitorRunnable, "A");
        thread.start();
        new Thread(monitorRunnable,"B").start();

        Thread.sleep(2000);              //睡眠 2s,保证 A、B 线程都能进入 wait()
        synchronized (monitorRunnable){
            System.out.println(Thread.currentThread().getName());
            monitorRunnable.notify();    //唤醒正在等待此对象监视器上的单个线程
            System.out.println("唤醒正在等待此对象监视器上的单个线程");
        }

    }

    static class MonitorRunnable implements Runnable{

        @Override
        public void run() {
```

```
                synchronized (this){
                    System.out.println(Thread.currentThread().getName());
                    try {
                        this.wait();   //使当前线程阻塞等待,直到它被唤醒或中断[释放当前锁]
                        System.out.println(Thread.currentThread().getName()
                                + "被唤醒了");
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                System.out.println(Thread.currentThread().getName() + ":end");
            }
        }
    }
```

注意：多次运行上方代码后，观察结果。

3.4.4 notifyAll()

6min

唤醒正在等待此对象监视器上的所有执行线程，代码如下：

```
//第 3 章/four/FourMain.java
public class FourMain {

    public static void main(String[] args) throws InterruptedException {
        MonitorRunnable monitorRunnable = new MonitorRunnable();
        Thread thread = new Thread(monitorRunnable, "A");
        thread.start();
        new Thread(monitorRunnable,"B").start();

        Thread.sleep(2000);              //睡眠 2s,保证 A、B 线程都能进入 wait()
        synchronized (monitorRunnable){
            System.out.println(Thread.currentThread().getName());
            monitorRunnable.notifyAll();
            System.out.println("唤醒正在等待此对象监视器上的所有线程");
        }

    }

    static class MonitorRunnable implements Runnable{

        @Override
        public void run() {
            synchronized (this){
                System.out.println(Thread.currentThread().getName());
                try {
                    this.wait();   //使当前线程阻塞等待,直到它被唤醒或中断[释放当前锁]
                    System.out.println(Thread.currentThread().getName()
                            + "被唤醒了");
```

```
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            System.out.println(Thread.currentThread().getName() + ":end");
        }
    }
}
```

执行结果如下：

```
A
B
main
唤醒正在等待此对象监视器上的所有线程
A被唤醒了
B被唤醒了
B:end
A:end
```

对象监视器的等待、唤醒机制配合使用可以很大程度地提高程序的灵活性，此处是模拟生产和消费的案例，代码如下：

12min

```
//第3章/four/ConsumeMain.java
public class ConsumeMain {
    public static void main(String[] args) {
        ConsumeRunnable consumeRunnable = new ConsumeRunnable();
        new Thread(consumeRunnable,"A").start();
        consumeRunnable.consume();
    }

    static class ConsumeRunnable implements Runnable{
        private Boolean consume = false;              //生产、消费状态标识
        private Object data = null;                   //存放消费的数据

        @Override
        public void run() {
            for (int x = 0;x < 10;x++){
                synchronized (this){
                    if(consume){
                        try {
                            this.wait();
                            Thread.sleep(3000);
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                    data = x;
                    consume = true;
                    this.notify();                    //有没有线程 wait 都可以调用
```

```
                    System.out.println(Thread.currentThread().getName()
                        +":生产了"+x);
                }
            }
        }

        public void consume(){
            for(int x=0;x<10;x++){
                synchronized (this){
                    if(!consume){
                        try {
                            this.wait();
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                    System.out.println(Thread.currentThread().getName()
                        +":消费了"+data);
                    data=null;
                    consume=false;
                    this.notify();          //有没有线程 wait 都可以调用
                }
            }
        }
    }
}
```

执行结果如下：

```
A:生产了 0
main:消费了 0
A:生产了 1
main:消费了 1
A:生产了 2
main:消费了 2
A:生产了 3
main:消费了 3
A:生产了 4
main:消费了 4
A:生产了 5
main:消费了 5
A:生产了 6
main:消费了 6
A:生产了 7
main:消费了 7
A:生产了 8
main:消费了 8
A:生产了 9
main:消费了 9
```

注意：notify()唤醒线程后，此线程一样需要执行拿锁的过程。

3.5　线程的可见性和重排序

可见性和重排序跟CPU、系统、Java底层的设计相关，这里主要关注怎么使用。可见性表示的是当某个线程修改了数据时，对于其他线程，是否能够立即感知数据被修改了。重排序表示的是在代码编译期间，对代码进行了顺序的改变。

3.5.1　可见性

8min

在极端的情况下，某个线程修改了数据，对其他线程不可见，代码如下：

```
//第3章/five/FiveMain.java
public class FiveMain {
    static boolean running = true;              //下次会修改的代码

    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                while (running){ //极端的一种演示情况

                }
                System.out.println(Thread.currentThread().getName()
                                                                + ":end");
            }
        },"A");
        thread.start();
        Thread.sleep(100);
        running = false;                        //修改数据
        System.out.println(Thread.currentThread().getName() + ":end");
    }
}
```

执行结果如下：

```
main:end
```

注意：由于A线程并没有感知到running数据被修改了，所以一直没有结束。

修改FiveMain类，代码如下：

```
//第3章/five/FiveMain.java
public class FiveMain {
    static volatile boolean running = true;
```

```
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                while (running){ //极端的一种演示情况

                }
                System.out.println(Thread.currentThread().getName()
                                                                    + ":end");
            }
        },"A");
        thread.start();
        Thread.sleep(100);
        running = false;
        Thread.sleep(100);
        System.out.println(Thread.currentThread().getName() + ":end");
    }
}
```

执行结果如下：

```
A: end
main: end
```

使用 volatile 关键字可以解决可见性问题。

3.5.2 重排序

7min

2min

表示的是在代码编译期间，对代码进行了顺序的改变。很难真实地验证，大家理解概念即可，代码如下：

```
//第 3 章/five/PossibleReordering.java
public class PossibleReordering {
    static int x, y = 0;                    //下次会修改的代码
    static int a, b = 0;                    //下次会修改的代码

    public static void main(String[] args) throws Exception {
        for (int num = 0;;) {
            num++;
            x = 0;y = 0;a = 0;b = 0;        //重置数据
            Thread threadA = new Thread(new Runnable() {
                @Override
                public void run() {
                    a = 1;
                    x = b;
                    /* 重排序后
                     * x = b;
                     * a = 1;
                     */
```

```
                }
            },"A");
            Thread threadB = new Thread(new Runnable() {
                @Override
                public void run() {
                    b = 1;
                    y = a;
                }
            },"B");
            threadA.start();          //启动线程
            threadB.start();          //启动线程
            threadA.join();
            threadB.join();
            if(x == 0 && y == 0) {
                System.out.println(num);
                return;
            }
        }
    }
}
```

运行 PossibleReordering 类主方法，等待执行结果。分析上面的代码，可能产生的结果有 3 种。

（1）A 线程先执行完：x==0、y==1。

（2）B 线程先执行完：x==1、y==0。

（3）一起执行完：x==1、y==1。

如果出现 x==0、y==0 的情况，即产生了异常数据，则可能就是由重排序造成的，如图 3-10 所示。

图 3-10　重排序后的效果

可以使用 volatile 关键字解决重排序问题，代码如下：

```
public class PossibleReordering {
    static volatile int x, y = 0;
    static volatile int a, b = 0;

    public static void main(String[] args) throws Exception {
        for (int num = 0;;) {
            num++;
            x = 0;y = 0;a = 0;b = 0;              //重置数据
            Thread threadA = new Thread(new Runnable() {
                @Override
                public void run() {
                    a = 1;
                    x = b;
                    /* 重排序后
                     * x = b;
                     * a = 1;
                     */
```

```
                }
            },"A");
            Thread threadB = new Thread(new Runnable() {
                @Override
                public void run() {
                    b = 1;
                    y = a;
                }
            },"B");
            threadA.start();            //启动线程
            threadB.start();            //启动线程
            threadA.join();
            threadB.join();
            if(x == 0 && y == 0) {
                System.out.println(num);
                return;
            }
        }
    }
}
```

注意：volatile 关键字可以解决可见性、重排序的问题。思考 volatile、synchronized 之间的区别。

3min

3.6 线程生命周期状态

Thread.State 是一个枚举类。一个线程在给定的时间点只能处于一种状态，这些状态是不反映任何操作系统线程状态的虚拟机线程状态。

3.6.1 NEW

7min

尚未启动的线程处于此状态，代码如下：

```
//第 3 章/six/SixMain.java
public class SixMain {

    public static void main(String[] args) throws InterruptedException {
        StateRunnable stateRunnable = new StateRunnable();
        Thread thread = new Thread(stateRunnable,"A");

        System.out.println(thread.getState()); //线程状态
    }

    static class StateRunnable implements Runnable{

        @Override
```

```
        public void run() {
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```

执行结果如下：

```
NEW
```

3.6.2　RUNNABLE

在执行中的线程处于此状态，代码如下：

```
//第 3 章/six/SixMain.java
public class SixMain {

    public static void main(String[] args) throws InterruptedException {
        StateRunnable stateRunnable = new StateRunnable();
        Thread thread = new Thread(stateRunnable,"A");
        thread.start();
        Thread.sleep(100);                          //睡眠 100ms
        System.out.println(thread.getState());  //线程状态
    }

    static class StateRunnable implements Runnable{

        @Override
        public void run() {
            //模拟线程一直在执行中的状态
            while (true){

            }
        }
    }
}
```

执行结果如下：

```
RUNNABLE
```

3.6.3　BLOCKED

阻塞等待拿锁的执行线程处于此状态，代码如下：

```
//第 3 章/six/SixMain.java
public class SixMain {
```

```
    public static void main(String[] args) throws InterruptedException {
        StateRunnable stateRunnable = new StateRunnable();
        Thread thread = new Thread(stateRunnable,"A");
        thread.start();
        synchronized (stateRunnable){//主线程先拿锁
            System.out.println(Thread.currentThread().getName());
            Thread.sleep(1000);                          //睡眠 1000ms
            System.out.println(thread.getState());   //线程状态
        }
    }

    static class StateRunnable implements Runnable{

        @Override
        public void run() {
            try {
                Thread.sleep(100);                       //睡眠 100ms
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            synchronized (this){
                System.out.println(Thread.currentThread().getName());
            }
        }
    }
}
```

执行结果如下：

```
main
BLOCKED
A
```

3.6.4 WAITING

无限期等待的执行线程处于此状态，代码如下：

```
//第 3 章/six/SixMain.java
public class SixMain {

    public static void main(String[] args) throws InterruptedException {
        StateRunnable stateRunnable = new StateRunnable();
        Thread thread = new Thread(stateRunnable,"A");
        thread.start();
        Thread.sleep(100);                          //睡眠 100ms
        synchronized (stateRunnable){
            System.out.println(thread.getState());  //线程状态
            stateRunnable.notify();                 //唤醒正在等待此对象监视器上的单个线程
        }
    }
```

```
    static class StateRunnable implements Runnable{

        @Override
        public void run() {
            synchronized (this){
                try {
                    this.wait();
            //使当前线程阻塞等待,直到它被唤醒或中断,具有释放当前锁的特性
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                System.out.println(Thread.currentThread().getName());
            }
        }
    }
}
```

执行结果如下：

```
WAITING
A
```

3.6.5　TIMED_WAITING

最大等待指定时间的执行线程处于此状态，代码如下：

```
//第 3 章/six/SixMain.java
public class SixMain {

    public static void main(String[] args) throws InterruptedException {
        StateRunnable stateRunnable = new StateRunnable();
        Thread thread = new Thread(stateRunnable,"A");
        thread.start();
        Thread.sleep(100);                          //睡眠 100ms
        synchronized (stateRunnable){
            System.out.println(thread.getState());  //线程状态
            stateRunnable.notify();                 //唤醒正在等待此对象监视器上的单个线程
        }
    }

    static class StateRunnable implements Runnable{

        @Override
        public void run() {
            synchronized (this){
                try {
                    this.wait(2000);
            //使当前线程阻塞等待,直到它被唤醒、中断或者直到经过一定量的实时时间
            //具有释放当前锁的特性
```

```
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                System.out.println(Thread.currentThread().getName());
            }
        }
    }
}
```

执行结果如下：

```
TIMED_WAITING
A
```

3.6.6 TERMINATED

已退出的执行线程处于此状态，代码如下：

```
//第 3 章/six/SixMain.java
public class SixMain {

    public static void main(String[] args) throws InterruptedException {
        StateRunnable stateRunnable = new StateRunnable();
        Thread thread = new Thread(stateRunnable,"A");
        thread.start();
        Thread.sleep(100);                          //睡眠 100ms
        System.out.println(thread.getState());  //线程状态
    }

    static class StateRunnable implements Runnable{

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```

执行结果如下：

```
A
TERMINATED
```

小结

通过本章的学习，相信读者已经掌握了多线程协作的核心知识点。合理地利用这些知识点，已经足够在大多数情况下去处理多线程的并发问题。后续章节还会介绍其他协作方

式，充分理解线程之间的协作概念，也是为学习后续课程奠定坚实的基础。

习题

1. 判断题

(1) Thread.sleep(2000)有释放当前锁的特性。(　　)

(2) 对象监视器.wait()有释放当前锁的特性。(　　)

(3) 对象监视器.notify()唤醒正在等待此对象监视器上的单个执行线程。(　　)

(4) volatile 关键字可以解决线程之间的可见性、重排序问题。(　　)

(5) 线程的生命周期共有 6 种状态，在给定时间点只能处于其中的一种状态。(　　)

(6) synchronized 对象锁可以锁任何对象。(　　)

(7) 对象监视器.wait(6000)可以在等待时间结束前唤醒。(　　)

2. 选择题

(1) 可以检测死锁的官方工具有(　　)。(多选)

A. JConsole　　B. keytool　　C. jstack　　D. jar

(2) 获得 Thread 当前执行线程对象正确的方法是(　　)。(单选)

A. this

B. Thread.currentThread()

C. this.getName()

D. Thread.currentThread().getName()

(3) Thread 类中有降低 CPU 调度执行效果的方法有(　　)。(多选)

A. 线程对象.setPriority(int newPriority)　B. Thread.yield()

C. 线程对象.getPriority()　　D. 线程对象.interrupt()

(4) volatile 关键字解决的问题有(　　)。(多选)

A. 可见性　　B. 并发脏读　　C. 并发脏写　　D. 重排序

3. 填空题

(1) 查看执行结果，并补充代码，代码如下：

```
//第 3 章/answer/SixMain.java
public class SixMain {

    private ________ static boolean running = true;
    public static void main(String[] args) throws InterruptedException {
        StateRunnable stateRunnable = new StateRunnable();
        Thread thread = new Thread(stateRunnable,"A");
        thread.start();
        Thread.sleep(100);                          //睡眠 100ms

        System.out.println(thread.getState());    //线程状态
```

```
        ____________________________;
    }

    static class StateRunnable implements Runnable{

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

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

执行结果如下：

```
RUNNABLE
A:end
```

（2）根据业务要求补全代码，A、B 两个执行线程轮流按照 1～100 的顺序打印数字，代码如下：

```
//第 3 章/answer/LoopPrint.java
public class LoopPrint {
    public static void main(String[] args) throws InterruptedException {
        LoopRunnable loopRunnable = new LoopRunnable();
        Thread threadA = new Thread(loopRunnable, ____);
        Thread threadB = new Thread(loopRunnable, ____);
        threadA.start();
        threadB.start();
    }

    static class LoopRunnable implements Runnable{
        private volatile int num = 1;
        @Override
        public void run() {
            for (;num <= 100;){
                synchronized (this){
                    System.out.println(Thread.currentThread().getName()
                    + ":" + num);
                    num++;
                    ____________________;
                    try {
                        ____________________;
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
```

```
                }
                synchronized (this){
                    ________________;
                }
            }
        }
    }
```

（3）根据业务要求补全代码，启动 3 个执行线程 A、B、C，每个执行线程将自己的名称轮流打印 5 遍，打印顺序是 ABCABC…，代码如下：

```
//第 3 章/answer/ThreePrint.java
public class ThreePrint {

    public static void main(String[] args) {
        LoopRunnable loopRunnable = new LoopRunnable();
        new Thread(loopRunnable, "A").start();
        new Thread(loopRunnable, "B").start();
        new Thread(loopRunnable, "C").start();
    }

    static class LoopRunnable implements Runnable {
        private volatile int loopIndex = 0;
        private final String[] loopNames = {"A", "B", "C"};

        @Override
        public void run() {
            for (int x = 0; x < 5; x++) {
                synchronized (this) {
                    String name = Thread.currentThread().getName();
                    //名称不匹配时一直循环
                    while (!name.equals(loopNames[loopIndex])) {
                        try {
                            ________________;
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                    System.out.print(name);              //消费名称
                    loopIndex++;
                    if (loopIndex == loopNames.length) {
                        loopIndex = 0;                   //重置指针
                    }
                    ____________________;
                }
            }
        }
    }
}
```

（4）根据业务要求补全代码，包括生产和消费，生产的总量为 10，代码如下：

```
//第 3 章/answer/ConsumeMain.java
public class ConsumeMain {
    public static void main(String[] args) {
        ConsumeRunnable consumeRunnable = new ConsumeRunnable();
        new Thread(consumeRunnable,"A").start();
        consumeRunnable.consume();
    }

    static class ConsumeRunnable implements Runnable{
        private Boolean consume = false;                    //生产、消费状态标识
        private Object data = null;                         //存放消费的数据

        @Override
        public void run() {
            for (int x = 0;x < 10;x++){
                synchronized (this){
                    if(consume){
                        try {
                            ____________________;
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                    data = x;
                    consume = true;
                    ____________________;                   //有没有线程 wait 都可以调用
                    System.out.println(Thread.currentThread().getName()
                                                    + ":生产了" + x);
                }
            }
        }

        public void consume(){
            for(int x = 0;x < 10;x++){
                synchronized (this){
                    if(!consume){
                        try {
                            ____________________;
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                    System.out.println(Thread.currentThread().getName()
                         + ":消费了" + data);
                    data = null;
```

```
                    consume = false;
                    _____________; //有没有线程 wait 都可以调用
                }
            }
        }
    }
}
```