让我们聊一聊Java并发之Synchronized

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：并发、Synchronized

题目描述

Java并发之Synchronized

题目解决

Synchronized简介

线程安全是并发编程中的至关重要的，造成线程安全问题的主要原因：

- 临界资源， 存在共享数据
- 多线程共同操作共享数据

而Java关键字synchronized，为多线程场景下防止临界资源访问冲突提供支持， 可以保证在同一时刻，只有一个线程可以执行某个方法或某个代码块操作共享数据。

即当要执行代码使用synchronized关键字时，它将检查锁是否可用，然后获取锁，执行代码，最后再释放锁。而synchronized有三种使用方式：

- synchronized方法： synchronized当前实例对象，进入同步代码前要获得当前实例的锁
- synchronized静态方法： synchronized当前类的class对象 ，进入同步代码前要获得当前类对象的锁
- synchronized代码块：synchronized括号里面的对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁

Synchronized方法

首先看一下没有使用synchronized关键字，如下：

```
public class ThreadNoSynchronizedTest {

    public void method1(){
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("method1");
    }

    public void method2() {
        System.out.println("method2");
    }
```

```
    public static void main(String[] args) {
        ThreadNoSynchronizedTest  tnst= new ThreadNoSynchronizedTest();

        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                tnst.method1();
            }
        });

        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                tnst.method2();
            }
        });
        t1.start();
        t2.start();
    }
}
```

在上述的代码中，method1比method2多了2s的延时，因此在t1和t2线程同时执行的情况下，执行结果：

> method2
>
> method1

当method1和method2使用了synchronized关键字后，代码如下：

```
public synchronized void method1(){
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("method1");
}

public synchronized void method2() {
    System.out.println("method2");
}
```

此时，由于method1占用了锁，因此method2必须要等待method1执行完之后才能执行，执行结果：

> method1
>
> method2

因此synchronized锁定是当前的对象，当前对象的synchronized方法在同一时间只能执行其中的一个，另外的synchronized方法需挂起等待，但不影响非synchronized方法的执行。下面的synchronized方法和synchronized代码块(把整个方法synchronized(this)包围起来）等价的。

```
public synchronized void method1(){

}

public  void method2() {
    synchronized(this){
    }
}
```

Synchronized静态方法

synchronized静态方法是作用在整个类上面的方法，相当于把类的class作为锁，示例代码如下：

```
public class TreadSynchronizedTest {

    public static synchronized void method1(){
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println("method1");
    }

    public static void method2() {
        synchronized(TreadTest.class){
            System.out.println("method2");
        }
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                TreadSynchronizedTest.method1();
            }
        });

        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                TreadSynchronizedTest.method2();
            }
        });
        t1.start();
        t2.start();
    }

}
```

由于将class作为锁，因此method1和method2存在着竞争关系，method2中synchronized(ThreadTest.class)等同于在method2的声明时void前面直接加上synchronized。上述代码的执行结果仍然是先打印出method1的结果：

> method1

> method2

Synchronized代码块

synchronized代码块应用于处理临界资源的代码块中，不需要访问临界资源的代码可以不用去竞争资源，减少了资源间的竞争，提高代码性能。示例代码如下：

```
    private Object obj = new Object();

    public void method1(){
        System.out.println("method1 start");
        synchronized(obj){
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System.out.println("method1 end");
         }
    }

    public void method2() {
        System.out.println("method2 start");


        // 延时10ms，让method1线获取到锁obj
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        synchronized(obj){
            System.out.println("method2 end");
        }
    }

    public static void main(String[] args) {
        TreadSynchronizedTest tst = new TreadSynchronizedTest();
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                tst.method1();
            }
        });

        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
```

```
                tst.method2();
            }
        });
        t1.start();
        t2.start();
    }
}
```

执行结果如下：

> method1 start
>
> method2 start
>
> method1 end
>
> method2 end

上述代码中，执行method2方法，先打印出 method2 start, 之后执行同步块，由于此时obj被method1获取到，method2只能等到method1执行完成后再执行，因此先打印method1 end，然后在打印method2 end。

Synchronized原理

synchronized 是JVM实现的一种锁，其中锁的获取和释放分别是monitorenter 和 monitorexit 指令。

加了 synchronized 关键字的代码段，生成的字节码文件会多出 monitorenter 和 monitorexit 两条指令，并且会多一个 ACC_SYNCHRONIZED 标志位，

当方法调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。

在方法执行期间，其他任何线程都无法再获得同一个monitor对象。其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。

在Java1.6之后，sychronized在实现上分为了偏向锁、轻量级锁和重量级锁，其中偏向锁在 java1.6 是默认开启的，轻量级锁在多线程竞争的情况下会膨胀成重量级锁，有关锁的数据都保存在对象头中。

- 偏向锁：在只有一个线程访问同步块时使用，通过CAS操作获取锁
- 轻量级锁：当存在多个线程交替访问同步快，偏向锁就会升级为轻量级锁。当线程获取轻量级锁失败，说明存在着竞争，轻量级锁会膨胀成重量级锁，当前线程会通过自旋（通过CAS操作不断获取锁），后面的其他获取锁的线程则直接进入阻塞状态。
- 重量级锁：锁获取失败则线程直接阻塞，因此会有线程上下文的切换，性能最差。

锁优化-适应性自旋（Adaptive Spinning）

从轻量级锁获取的流程中我们知道，当线程在获取轻量级锁的过程中执行CAS操作失败时，是要通过自旋来获取重量级锁的。问题在于，自旋是需要消耗CPU的，如果一直获取不到锁的话，那该线程就一直处在自旋状态，白白浪费CPU资源。

其中解决这个问题最简单的办法就是指定自旋的次数，例如让其循环10次，如果还没获取到锁就进入阻塞状态。但是JDK采用了更聪明的方式——适应性自旋，简单来说就是线程如果自旋成功了，则下次自旋的次数会更多，如果自旋失败了，则自旋的次数就会减少。

锁优化-锁粗化（Lock Coarsening）

锁粗化的概念应该比较好理解，就是将多次连接在一起的加锁、解锁操作合并为一次，将多个连续的锁扩展成一个范围更大的锁。举个例子：

```
public class StringBufferTest {
     StringBuffer stringBuffer = new StringBuffer();
     public void append(){
         stringBuffer.append("a");
         stringBuffer.append("b");
         stringBuffer.append("c");
     }
 }
```

这里每次调用stringBuffer.append方法都需要加锁和解锁，如果虚拟机检测到有一系列连串的对同一个对象加锁和解锁操作，就会将其合并成一次范围更大的加锁和解锁操作，即在第一次append方法时进行加锁，最后一次append方法结束后进行解锁。

锁优化-锁消除（Lock Elimination）

锁消除即删除不必要的加锁操作。根据代码逃逸技术，如果判断到一段代码中，堆上的数据不会逃逸出当前线程，那么可以认为这段代码是线程安全的，不必要加锁。看下面这段程序：

```
public class SynchronizedTest02 {

     public static void main(String[] args) {
         SynchronizedTest02 test02 = new SynchronizedTest02();
         for (int i = 0; i < 10000; i++) {
             i++;
         }
         long start = System.currentTimeMillis();
         for (int i = 0; i < 100000000; i++) {
             test02.append("abc", "def");
         }
         System.out.println("Time=" + (System.currentTimeMillis() - start));
     }

     public void append(String str1, String str2) {
         StringBuffer sb = new StringBuffer();
         sb.append(str1).append(str2);
     }
}
```

虽然StringBuffer的append是一个同步方法，但是这段程序中的StringBuffer属于一个局部变量，并且不会从该方法中逃逸出去，所以其实这过程是线程安全的，可以将锁消除。

Sychronized缺点

Sychronized会让没有得到锁的资源进入Block状态，争夺到资源之后又转为Running状态，这个过程涉及到操作系统用户模式和内核模式的切换，代价比较高。

Java1.6为 synchronized 做了优化，增加了从偏向锁到轻量级锁再到重量级锁的过度，但是在最终转变为重量级锁之后，性能仍然较低。