> 作者：JavaGieGie
>
> 微信公众号：Java开发零到壹

前言

上篇在synchronized精讲中，我们讲解了synchronized的用法、注意项等知识点，但知道用法只是基础，如果想要拿到更高的工资、更好的岗位，了解它的实现原理也是很重要的加分项，本章学习的意义非常重要。

本节主要讲解以下几个内容：

1. synchronized有哪些特性?
2. synchronized的实现原理?
3. synchronized的作用?
4. synchronized的缺陷?

正文

> **synchronized有哪些性质?**

- **可重入性**

定义：可重入是指同一个线程当外层函数获取到锁时，内层函数可以直接再次获取该锁。

看不懂没关系，我们先来看一个例子。

前一篇文章我们说过一妻多夫的生活案例（`没有看过的小伙伴建议先看前一篇`），小伙伴们想一下，假设兄弟四人中老大先拿到这块砖（锁），那么他当天就有资格进入房间（同步方法/代码块），这没毛病，毕竟早起的鸟儿有虫吃。在老大持有这块砖的过程中，他可以在房间随意走动，他甚至可以去厕所、浴室，卧室等等，并且都不需要再重新去获取那块砖（锁）。

这就叫**可重入**，一旦线程获取到对象锁（砖），再去访问该对象锁的所有同步方法（浴室、卧室）时，就不用再去重新竞争锁资源。

可能还有聪明的小伙伴觉得不够清晰，这里花Gie用 **代码进行演示**

```
//方法一
public synchronized void testSync1(){
    System.out.println("我进来了方法1");
    testSync1();
}

public static void main(String[] args) {
    new SyncDemo().testSync1();
}
```

运行过这段代码的小伙伴是不是惊喜的发现了什么呢，是不是一直到把IDEA跑崩了，它才结束程序。

```
我进来了方法1
我进来了方法1
我进来了方法1
Exception in thread "main" java.lang.StackOverflowError
```

这就可以得出可重入的第一个结论，线程获取到锁后，可以重复进入同一个方法。

那如果是调用 **不同的synchronized** 方法呢

```
public synchronized void testSync1(){
    System.out.println("我进来了方法1");
    testSync2();
}

public synchronized void testSync2(){
    System.out.println("我进来了方法2");
}
```

结果也很明显，依然可以 **调用成功**

```
我进来了方法1
我进来了方法2
```


从而得到的另一个结论：**获取到锁的线程，可以进入同一个对象的不同synchronized方法。**

- **不可中断性**

一旦锁被某个线程获取，其他线程再想进入`同步方法/代码块`，就只能等待，直到持有锁的那个线程释放这个锁，如果一直不释放这个锁，那么其他线程只能一直等待下去。

> **synchronized释放锁方式：1. 正常结束；2. 抛出异常**

这个比较容易理解，如例子中老大获取到锁后，只要他不主动让出那块砖，他就可以一直待在房间中，其他兄弟毫无办法，只能干瞪眼。

> **synchronized的实现原理?**

synchronized并不需要我们手动加锁、释放锁，因此想要分析原理的话，需要反汇编class文件进行查看。

> 反汇编操作流程拓展：
>
> 1. 找到java文件路径，使用 `javac 文件.java`，将java文件编译成class文件；
> 2. 使用`javap -c -v`命令进行反汇编，便可以查看到反汇编后的内容

先看一下第一个例子，使用synchronized修饰代码块

```
public synchronized void testSync2(){
    synchronized (this){

    }
}
```

反汇编后的内容如下：

从上面的内容我们可以看到，JVM采用`monitorenter`、`monitorexit`这两个指令来实现线程同步。

> 可以把monitorenter理解为 **加锁** 操作，monitorexit则是 **释放锁**。每个对象都会维护一个计数器，当一个线程获取到锁（执行monitorenter）后，计数器 **加1**；释放锁后（执行monitorexit指令），计数器 **减1**，当计数器的值为 0时，该对象锁被释放，其他等待中的线程开始竞争该锁。

从图中的标记我们可以看到，monitorexit被执行了两次，这是什么原因呢?

> 因为编译器要保证线程调用完`monitorenter`指令后，都要执行与其对应的 monitorexit 指令。为了保证在程序**发生异常**时二者仍然能够保持匹配，编译器会设置一个`异常处理器`，用于执行异常时的 monitorexit 指令。这样锁也能被正常释放，也不会导致死锁。图中最后一行`monitorexit`指令，就是异常结束时，被用来释放monitor的。

第二个例子，使用synchronized修饰方法

```
public synchronized void testSync1(){

}

public synchronized void testSync1();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED
    Code:
      stack=0, locals=1, args_size=1
         0: return
      LineNumberTable:
        line 14: 0
```

同样反汇编后看下是什么内容

```
public synchronized void testSync1();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED
    Code:
      stack=0, locals=1, args_size=1
         0: return
      LineNumberTable:
        line 14: 0
```

没想到吧，是不是有那么一瞬间你觉得自己啥都会了，monitorenter指令、monitorexit指令一个都没有

这是为什么，是不是百思不得其姐，就让花Gie科普一下吧

> 方法级的同步是隐式的，也就是说不需要通过字节码指令来控制的。JVM通过ACC_SYNCHRONIZED 访问标志来区分一个方法是否是同步方法。
>
> 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，说明该方法为同步方法，请求的线程需要先获取monitor（监视器锁），然后才能执行方法，在方法执行后再释放monitor。方法执行期间，其他任何线程因为无法获得monitor而处于阻塞状态。如果一个同步方法执行期间抛出了异常，并且在方法内部无法处理此异常，那么在异常被抛到方法外面之前监视器锁会被自动释放。

> **synchronized的作用?**

分析完原理，现在再来看下synchronized的作用，理解起来就有据可依了，synchronized的作用主要有以下三个：

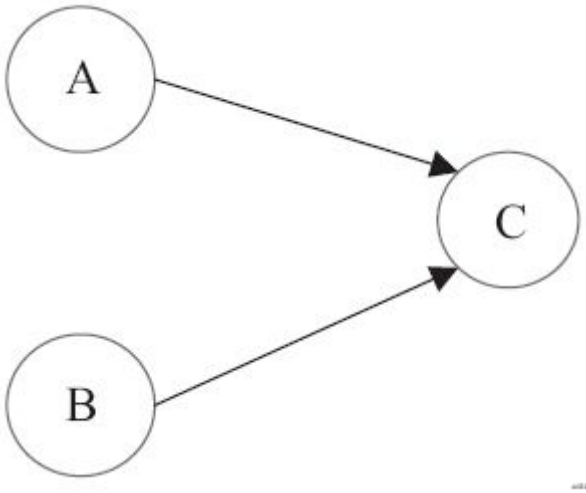
1. 原子性：一个操作或者多个操作 要么**全部执行完成**，要么**都不执行**。Java内存模型提供了`monitorenter`和`monitorexit`指令来隐式的实现原子操作，从而实现线程之间互斥的访问同步代码块，将多个小原子操作合成大原子。
2. 可见性：当多个线程访问同一个变量时，一个线程修改了该变量的值，其他线程能够立即看得到修改后的值。synchronized **解锁之前**，必须将工作内存中的数据同步到主内存，因此在解锁后，其它线程操作该变量时每次都可以看到被修改后的值。
3. 有序性：程序执行的顺序按照代码的先后顺序执行。这其实和 as-if-serial 有关。

> **拓展：来自《Java并发编程的艺术》书籍：**

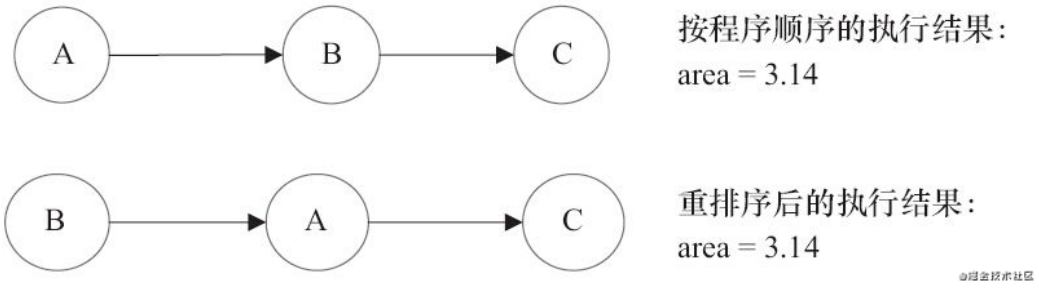
> 什么是as-if-serial语义：不管怎么重排序，单线程程序的执行结果不能被改变。编译器、runtime和处理器都必须遵守as-if-serial语义。所以编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但是，如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。

```
double pi  = 3.14;            // A
double r   = 1.0;             // B
double area = pi * r * r;     // C
```

上面3个操作的数据依赖关系如图所示：



A和C之间存在数据依赖关系，同时B和C之间也存在数据依赖关系。因此在最终执行的指令序列中，C不能被重排序到A和B的前面(因为C排到A和B的前面，程序的结果将会被改变)。但A和B之间没有数据依赖关系，编译器和处理器可以重排序A和B之间的执行顺序。



as-if-serial语义把单线程程序保护了起来，遵守as-if-serial语义的编译器、runtime和处理器共同为编写单线程程序的程序员创建了一个幻觉：单线程程序是按程序的顺序来执行的。

> **synchronized的缺陷?**

1. 效率低。
    - 等待获取锁的线,不能主动退出等待。
    - 持有锁的线程只能在执行完毕、抛出异常释放锁，除此之外不能主动释放锁。
2. 无法知道是否成功获取到锁。

总结

本章提到了一些新的概念，比如`监视器锁`，`monitorenter`，`monitorexit`以及`反汇编`这类的词语，新入手的小伙伴可能会有一丝丝的难受，但是对于概念性的词汇，我们只能试着去理解，不能过于捉急，有什么不理解的可以在**下方留言**，花Gie肯定有问必答哦，共同进步。

下期预告

本章讲解了synchronized的全部内容，接下来会在`ThreadLocal`，`线程池`中进行分享，加油！冲鸭~ **希望大家持续关注，为了大厂梦，我们继续肝。**

点关注，防走丢

以上就是本期全部内容，**如有纰漏之处，请留言指教，非常感谢。**我是花GieGie，有问题大家随时留言讨论，我们下期见👊。

> 文章持续更新，可以微信搜一搜「**Java开发零到壹**」第一时间阅读，后续会持续更新Java面试和各类知识点，有兴趣的小伙伴欢迎关注，一起学习，一起哈🐯🥃。