> 作者：JavaGieGie
>
> 微信公众号：Java开发零到壹

前言

涉猎广泛的小伙伴应该知道，我们国家有些少数民族至今保留着一妻多夫的制度，云南省德钦县有个村子雾浓顶村，全村有20户人家，有8户是几个兄弟娶一个妻子的，最多者有4兄弟娶一妻的。花Gie其不关心，只是比较关心同胞们的就寝情况，据官方消息称，每晚就寝时先到的兄弟会在门口放置一块砖，如果后面还有其他兄弟过来，看到这块砖后，就知晓了一切（握草，又来晚了）。

看到这里，小伙伴们是不是若有所思呢，这就是生活中解决多线程并发问题的经典实例，如果没有这块砖，5个人，十目相对是不是很尴尬呢。

本章主要讲解以下几个内容

1. 线程安全问题是什么?
2. 如何解决线程安全问题?
3. synchronized用法讲解!

正文

> **线程安全问题是什么?**

在日常业务开发过程中，很多小伙伴们可能没有考虑到线程安全问题，这是因为我们的开发场景大部分只是简单的CRUD，不存在资源竞争；而在多线程编程中，很大程度会出现多个线程同时访问同一个资源的场景，比如同时操作一个变量、一个对象、一个数据库表等。

举个常见的栗子：

这不还有两天就是618购物节，估计剁手党小伙伴已经磨刀霍霍，蓄势待发了。对于一些库存不足，较为稀缺的商品，各大网购平台都会将其设置为秒杀商品，这个时候就要展示大家单身几十年的手速了。

在秒杀过程中会可以简化为以下两个过程：

1. 查询商品库存数量
2. 如果库存充足，则扣减库存；库存不足，返回用户失败信息

假如某一时刻，只存在两个线程th_1、th_2，同时去查询库存数量，并获取到查询结果。

> 假设在th_1、th_2查询过程中，没有其他线程对库存进行扣减操作，所以这两个线程查询结果会是一致的。

th_1 根据数据库返回结果判断库存充足，可以进行下单操作，而线程2同样判断出库存充足。

最终两个用户都会下单成功，如果此时库存商品数量仅剩1个，那商家会出现多卖的现象。

这就是线程安全问题，简单说就是**多个线程同时访问一个资源时，会出现违背预期的结果。**这个资源被称为**临界资源**，或**共享资源。**

> **如何解决线程安全问题?**

从上面的案例我们可以看出，最终出现商品多卖的原因，是因为秒杀过程中查询和扣除库存不属于原子操作，一个线程在扣减库存且尚未完成数据库入库的情况下，其他线程此时能够去数据库查询库存，这就导致其他线程读取了不正确的数据。

那如何解决这一问题呢，通常的做法是**在同一时刻，只允许一个线程访问临界资源（查询和修改库存），在访问临界资源的代码前面加上一个锁，当访问完临界资源后释放锁，让其他线程继续访问。**

Java提供了两种方式解决这一问题：**synchronized**和**Lock**，Lock我们会在后续章节讲解。

> **synchronized分析讲解?**

介绍synchronized之前，我们要了解一个概念，**锁**，这个花Gie前面文章有介绍过一部分。

锁这个东西说起来很抽象, 你可以就把它想象成现实中的锁, 至于他的防盗锁、金锁、还是指纹锁并不重要，哪怕它就是一根草绳，一个自行车、甚至一坨那啥，都可以当做锁。 **锁**是什么外在形象并不重要，重要的是它代表的含义: 谁持有它, 谁就有独立访问临界资源的权利。

说完了锁，我下面用代码演示synchronized的几种用法。

1. **修饰普通方法**

```
public class SyncDemo{
    public static void main(String[] args) {
        PrintClass printClass = new PrintClass();
        new Thread(new Runnable() {
            @Override
            public void run() {
                printClass.print(Thread.currentThread());
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                printClass.print(Thread.currentThread());
            }
        }).start();
    }
}

class PrintClass {

    public void print(Thread thread){
        for(int i=0;i<5;i++){
            System.out.println(thread.getName()+"打印数据"+i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

输出结果为:

```
Thread-0打印数据0
Thread-1打印数据0
Thread-1打印数据1
Thread-0打印数据1
Thread-1打印数据2
Thread-0打印数据2
Thread-1打印数据3
Thread-0打印数据3
Thread-1打印数据4
Thread-0打印数据4
```

从结果中我们可以看到，两个线程能够同时执行 `print()` 方法，也就说明 `print()` 方法在多线程情况下是不安全的。

如果我们在 `print()` 方法前加上 `synchronized` 之后，我们看下：

```
public synchronized  void print(Thread thread){
    for(int i=0;i<5;i++){
        System.out.println(thread.getName()+"打印数据"+i);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

代码执行结果为：

```
Thread-0打印数据0
Thread-0打印数据1
Thread-0打印数据2
Thread-0打印数据3
Thread-0打印数据4
Thread-1打印数据0
Thread-1打印数据1
Thread-1打印数据2
Thread-1打印数据3
Thread-1打印数据4
```

结果很明显 `Thread-0` 执行完毕后，`Thread-1` 才开始执行。

这是因为两个线程thread_0 和thread_1 持有同一个对象 `printClass` 的锁, 因此同一时刻, 只有一个线程能够访问 `print` 方法，线程 `thread_0` 在执行完方法后，会自动释放锁，然后 `thread_1` 才能获取这把锁, 执行同步方法中的代码。

没听懂？那我这里再举个反例，使用不同的对象调用方法，会有什么情况呢：

```
public class SyncDemo{
    public static void main(String[] args) {
        PrintClass printClass1 = new PrintClass();
        PrintClass printClass2 = new PrintClass();
        new Thread(new Runnable() {
            @Override
            public void run() {
```

```
                printClass1.print(Thread.currentThread());
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                printClass2.print(Thread.currentThread());
            }
        }).start();
    }
}
public synchronized  void print(Thread thread){
    for(int i=0;i<5;i++){
        System.out.println(thread.getName()+"打印数据"+i);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

执行结果也很明显，两个线程同时执行`print()`方法：

```
Thread-0打印数据0
Thread-1打印数据0
Thread-1打印数据1
Thread-0打印数据1
Thread-1打印数据2
Thread-0打印数据2
Thread-1打印数据3
Thread-0打印数据3
Thread-1打印数据4
Thread-0打印数据4
```

这里对`synchronized`修饰的普通方法，我们得出以下几点：

- 当一个线程正在访问一个对象的synchronized方法时，其他线程不能访问该对象的任意一个`synchronized`方法。因为一个对象只有一把锁，当该锁被占用后，其他线程无法获取该对象的锁，所以无法访问该对象任意一个`synchronized`方法。
- 当一个线程正在访问一个对象的`synchronized`方法，那么其他线程可以访问该对象的非`synchronized`方法。这是因为访问非`synchronized`方法不需要获得该对象的锁，因此可以被其他线程所访问。
- 如果一个线程1访问`printClass1`对象的`print`方法，另一个线程2依然可以访问`printClass2`的`print`方法。虽然`printClass1`、`printClass2`属于同一个类型，但是他们访问的是不同的对象，所以不存在互斥问题。
- synchronized方法不能继承。父类中用synchronized 修饰的方法，子类在重写时，默认情况下不是同步的，必须显式的使用 synchronized 关键字修饰。

---

2. **synchronized修饰静态方法**

把上面的例子稍加修改一下，在print方法前加上static ，大家先思考一下，会出现什么情况。

```
public synchronized static void print(Thread thread){
    for(int i=0;i<5;i++){
        System.out.println(thread.getName()+"打印数据"+i);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

执行结果如下：

```
Thread-0打印数据0
Thread-0打印数据1
Thread-0打印数据2
Thread-0打印数据3
Thread-0打印数据4
Thread-1打印数据0
Thread-1打印数据1
Thread-1打印数据2
Thread-1打印数据3
Thread-1打印数据4
```

不要急，不要慌，年轻人要沉得住气，且听花Gie娓娓道来。

> 拓展：java有两种对象：实例对象和Class对象。每一个类都有一个Class对象，每当编译一个新类就产生一个Class对象，基本类型 (boolean, byte, char, short, int, long, float, and double)有Class对象，数组有Class对象，就连关键字void也有Class对象（void.class）。Class对象对应着java.lang.Class类，如果说类是对象抽象和集合的话，那么Class类就是对类的抽象和集合。

这是因为静态方法是属于类而不属于对象，如果使用`synchronized`修饰静态方法，这就是拿Class对象作为锁，而我们通过上面介绍知道Class对象只有一个，所以无论创建多少个实例对象(new出来的)来进行调用，使用得都是类锁（只有一把），所以就实现了线程间的互斥（不能同时访问）。

对于synchronized修饰的静态方法，需要注意点：

- 两个线程（线程1、线程2），如果线程1执行一个对象的static synchronized方法，另外一个线程需要执行这个对象的fi非static synchronized方法，两者不会发生互斥现象（能够同时访问），因为访问static synchronized方法占用的是类锁，而访问非static synchronized方法占用的是对象锁，所以不存在互斥现象；
- 每个类的类对象只有一个。

---

3. **synchronized代码块**

代码形式如下：

```
synchronized(object) {    //由锁保护的代码}
```

当某个线程执行到这段代码块时，该线程会尝试获取对象object的锁，然后才能进入代码块执行。object有两种形式：如果是this，代表需要获取当前对象的锁（`printClass`）；如果是类中的一个属性，代表需要获取该属性的锁。

还是用刚刚的栗子修改后进行演示：

```
public class SyncDemo{    public static void main(String[] args) {
 PrintClass printClass = new PrintClass();        new Thread(new Runnable() {
         @Override            public void run() {
 printClass.print(Thread.currentThread());            }        }).start();
 new Thread(new Runnable() {            @Override            public void run() {
               printClass.print(Thread.currentThread());            }
}).start();    }}//方式1 通过使用this（printClass对象锁）class PrintClass {
public void print(Thread thread){        synchronized (this){            for(int
i=0;i<5;i++){                System.out.println(thread.getName()+"打印数据"+i);
             try {                    Thread.sleep(100);                } catch
(InterruptedException e) {                    e.printStackTrace();
}            }        }    }}//方式2 通过使用object属性锁class PrintClass2 {
private Object object = new Object();        public void print(Thread thread){
     synchronized (object){            for(int i=0;i<5;i++){
System.out.println(thread.getName()+"打印数据"+i);                try {
         Thread.sleep(100);                } catch (InterruptedException e) {
               e.printStackTrace();                }            }        }    }}
```

执行结果如下：

```
Thread-0打印数据0Thread-0打印数据1Thread-0打印数据2Thread-0打印数据3Thread-0打印数据
4Thread-1打印数据0Thread-1打印数据1Thread-1打印数据2Thread-1打印数据3Thread-1打印数据4
```

这里可以看到，由于两个线程`thread_0`和`thread_1`持有同一个对象`printClass`（或object属性）的锁，因此同一时刻，只有一个线程能访问`print`方法，线程`thread_0`在执行完方法后，会自动释放锁，然后`thread_1`才能获取这把锁，进入同步方法。

....是不是觉得这句话在哪看过，这里的结论和上面使用synchronized修饰普通方法得出的结果是一致的。小伙伴们还可以尝试使用`不同对象`（`或属性`）进行调用，篇幅太长这里就不一一列举了。

下面这两种方式是等价的

```
//写法1public synchronized void print() {    // 由锁保护的代码}//写法2public void
print() {    synchronized(this) {        // 由锁保护的代码    }}
```

针对同步代码块的概括：

- 当一个线程访问`printClass`对象的一个synchronized(this)同步代码块时，其他线程对`printClass`中所有其它synchronized(this)同步代码块的访问将被阻塞。因为同一个对象只有一把锁。
- 当一个线程访问`printClass`对象的一个synchronized(this)同步代码块时，其他线程对`printClass`中非synchronized(this)同步代码块依然可以访问。只要对象中的代码没有加锁，就不需要获取锁资源，所以能够自由访问。

总结

这里花Gie对`synchronized`做一个**总结**吧：

- 任何java对象都可以作为锁；
- 一把锁只能同时被一个线程获取，没有拿到锁的线程会进行等待；
- 每个实例都对应有自己的一把锁，不通实例之间互不影响（如printClass1、printClass2）；
- 作用于静态方法时，相当于以Class对象作为锁（类锁），此时对象的所有实例只能争抢同一把锁；
- 无论方法正常执行完毕或者抛出异常，都会释放锁。

文章比较长，原本想把synchronized的底层原理也一块讲解了，但是写着写着就太多了，**所以决定另起一篇专门讲解，花Gie觉得非常有必要掌握**，无论使用还是原理，都需要小伙伴们慢慢咀嚼，千万不要囫囵吞枣，看了一下总结，就自以为自己已经掌握了，还是那句话，`纸上得来终觉浅，绝知此事要躬行`，多动手，多实践。

下期预告

知道了怎么用，还远远不够，针对`synchronized`同步代码块究竟是怎么获得锁的?、`JAVA中任何对象都可以作为锁，那么锁信息是怎么被记录和存储的?`、`释放锁又是怎么释放的呢？`这些问题，你又是有怎样的理解呢?

带着这些问题，下一章花Gie会，我们下一章会深入原理，带小伙伴们更深一步了解。**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🐯🥞。

原创不易，你怎忍心白嫖，如果你觉得这篇文章对你有点用的话，感谢老铁为本文**点个赞、评论或转发一下**，因为这将是我输出更多优质文章的动力，感谢！