07 哪些场景需要额外注意线程安全问题?

在本课时我们主要学习哪些场景需要额外注意线程安全问题，在这里总结了四种场景。

访问共享变量或资源

第一种场景是访问共享变量或共享资源的时候，典型的场景有访问共享对象的属性，访问 static 静态变量，访问共享的缓存，等等。因为这些信息不仅会被一个线程访问到，还有可能被多个线程同时访问，那么就有可能在并发读写的情况下发生线程安全问题。比如我们上一课时讲过的多线程同时 i++ 的例子：

```
/**
 * 描述：        共享的变量或资源带来的线程安全问题
 */
public class ThreadNotSafe1 {
    static int i;
    public static void main(String[] args) throws InterruptedException {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                for (int j = 0; j < 10000; j++) {
                    i++;
                }
            }
        };
        Thread thread1 = new Thread(r);
        Thread thread2 = new Thread(r);
```

```
            thread1.start();
            thread2.start();
            thread1.join();
            thread2.join();
            System.out.println(i);
        }
    }
```

如代码所示，两个线程同时对 i 进行 i++ 操作，最后的输出可能是 15875 等小于20000的数，而不是我们期待的20000，这便是非常典型的共享变量带来的线程安全问题。

依赖时序的操作

第二个需要我们注意的场景是依赖时序的操作，如果我们操作的正确性是依赖时序的，而在多线程的情况下又不能保障执行的顺序和我们预想的一致，这个时候就会发生线程安全问题，如下面的代码所示：

```
if (map.containsKey(key)) {
    map.remove(obj)
}
```

代码中首先检查 map 中有没有 key 对应的元素，如果有则继续执行 remove 操作。此时，这个组合操作就是危险的，因为它是先检查后操作，而执行过程中可能会被打断。如果此时有两个线程同时进入 if() 语句，然后它们都检查到存在 key 对应的元素，于是都希望执行下面的 remove 操作，随后一个线程率先把 obj 给删除了，而另外一个线程它刚已经检查过存在 key 对应的元素，if 条件成立，所以它也会继续执行删除 obj 的操作，但实际上，集合中的 obj 已经被前面的线程删除了，这种情况下就可能导致线程安全问题。

类似的情况还有很多，比如我们先检查 x=1，如果 x=1 就修改 x 的值，代码如下所示：

```
if (x == 1) {
    x = 7 * x;
}
```

这样类似的场景都是同样的道理，“检查与执行”并非原子性操作，在中间可能被打断，而检查之后的结果也可能在执行时已经过期、无效，换句话说，获得正确结果取决于幸运的时序。这种情况下，我们就需要对它进行加锁等保护措施来保障操作的原子性。

不同数据之间存在绑定关系

第三种需要我们注意的线程安全场景是不同数据之间存在相互绑定关系的情况。有时候，我们的不同数据之间是成组出现的，存在着相互对应或绑定的关系，最典型的就是 IP 和端口号。有时候我们更换了 IP，往往需要同时更换端口号，如果没有把这两个操作绑定在一起，就有可能出现单独更换了 IP 或端口号的情况，而此时信息如果已经对外发布，信息获取方就有可能获取一个错误的 IP 与端口绑定情况，这时就发生了线程安全问题。在这种情况下，我们也同样需要保障操作的原子性。

对方没有声明自己是线程安全的

第四种值得注意的场景是在我们使用其他类时，如果对方没有声明自己是线程安全的，那么这种情况下对其他类进行多线程的并发操作，就有可能会发生线程安全问题。举个例子，比如说我们定义了 ArrayList，它本身并不是线程安全的，如果此时多个线程同时对 ArrayList 进行并发读/写，那么就有可能会产生线程安全问题，造成数据出错，而这个责任并不在 ArrayList，因为它本身并不是并发安全的，正如源码注释所写的：

```
Note that this implementation is not synchronized. If multiple threads

access an ArrayList instance concurrently, and at least one of the threads

modifies the list structurally, it must be synchronized externally.
```

这段话的意思是说，如果我们把 ArrayList 用在了多线程的场景，需要在外部手动用 synchronized 等方式保证并发安全。

所以 ArrayList 默认不适合并发读写，是我们错误地使用了它，导致了线程安全问题。所以，我们在使用其他类时如果会涉及并发场景，那么一定要首先确认清楚，对方是否支持并发操作，以上就是四种需要我们额外注意线程安全问题的场景，分别是访问共享变量或资源，依赖时序的操作，不同数据之间存在绑定关系，以及对方没有声明自己是线程安全的。

上一页 下一页