45 ThreadLocal 是用来解决共享资源的多线程访问的问题吗?

本课时主要讲解一个问题：ThreadLocal 是不是用来解决共享资源的多线程访问的。

这是一个常见的面试问题，如果被问到了 ThreadLocal，则有可能在你介绍完它的作用、注意点等内容之后，再问你：ThreadLocal 是不是用来解决共享资源的多线程访问的呢？假如遇到了这样的问题，其思路一定要清晰。这里我给出一个参考答案。

面试时被问到应如何回答

这道题的答案很明确——不是，ThreadLocal 并不是用来解决共享资源问题的。虽然 ThreadLocal 确实可以用于解决多线程情况下的线程安全问题，但其资源并不是共享的，而是每个线程独享的。所以这道题其实是有一定陷阱成分在内的。

ThreadLocal 解决线程安全问题的时候，相比于使用“锁”而言，换了一个思路，把资源变成了各线程独享的资源，非常巧妙地避免了同步操作。具体而言，它可以在 initialValue 中 new 出自己线程独享的资源，而多个线程之间，它们所访问的对象本身是不共享的，自然就不存在任何并发问题。这是 ThreadLocal 解决并发问题的最主要思路。

如果我们把放到 ThreadLocal 中的资源用 static 修饰，让它变成一个共享资源的话，那么即便使用了 ThreadLocal，同样也会有线程安全问题。比如我们对第 44 讲中的例子进行改造，如果我们在 SimpleDateFormat 之前加上一个 static 关键字来修饰，并且把这个静态对象放到 ThreadLocal 中去存储的话，代码如下所示：

```
public class ThreadLocalStatic {

    public static ExecutorService threadPool = Executors.newFixedThreadPool(16);

    static SimpleDateFormat dateFormat = new SimpleDateFormat("mm:ss");

    public static void main(String[] args) throws InterruptedException {

        for (int i = 0; i < 1000; i++) {

            int finalI = i;
```

```
                threadPool.submit(new Runnable() {
                    @Override
                    public void run() {
                        String date = new ThreadLocalStatic().date(finalI);
                        System.out.println(date);
                    }
                });
            }
            threadPool.shutdown();
        }
        public String date(int seconds) {
            Date date = new Date(1000 * seconds);
            SimpleDateFormat dateFormat = ThreadSafeFormatter.dateFormatThreadLocal.get
            return dateFormat.format(date);
        }
    }
    class ThreadSafeFormatter {
        public static ThreadLocal<SimpleDateFormat> dateFormatThreadLocal = new ThreadL
            @Override
            protected SimpleDateFormat initialValue() {
                return ThreadLocalStatic.dateFormat;
            }
        }
    }
```

那么在多线程中去获取这个资源并且同时使用的话，同样会出现时间重复的问题，运行结果如下。

```
00:15
00:15
```

```
00:05

00:16

...
```

可以看出，00:15 被多次打印了，发生了线程安全问题。也就是说，如果我们需要放到 ThreadLocal 中的这个对象是共享的，是被 static 修饰的，那么此时其实根本就不需要用到 ThreadLocal，即使用了 ThreadLocal 并不能解决线程安全问题。

相反，我们对于这种共享的变量，如果想要保证它的线程安全，应该用其他的方法，比如说可以使用 synchronized 或者是加锁等其他的方法来解决线程安全问题，而不是使用 ThreadLocal，因为这不是 ThreadLocal 应该使用的场景。

这个问题回答到这里，可能会引申出下面这个问题。

ThreadLocal 和 synchronized 是什么关系

面试官可能会问：你既然说 ThreadLocal 和 synchronized 它们两个都能解决线程安全问题，那么 ThreadLocal 和 synchronized 是什么关系呢?

我们先说第一种情况。当 ThreadLocal 用于解决线程安全问题的时候，也就是把一个对象给每个线程都生成一份独享的副本的，在这种场景下，ThreadLocal 和 synchronized 都可以理解为是用来保证线程安全的手段。例如，在第 44 讲 SimpleDateFormat 的例子中，我们既使用了 synchronized 来达到目的，也使用了 ThreadLocal 作为实现方案。但是效果和实现原理不同：

- ThreadLocal 是通过让每个线程独享自己的副本，避免了资源的竞争。
- synchronized 主要用于临界资源的分配，在同一时刻限制最多只有一个线程能访问该资源。

相比于 ThreadLocal 而言，synchronized 的效率会更低一些，但是花费的内存也更少。在这种场景下，ThreadLocal 和 synchronized 虽然有不同的效果，不过都可以达到线程安全的目的。

但是对于 ThreadLocal 而言，它还有不同的使用场景。比如当 ThreadLocal 用于让多个类能更方便地拿到我们希望给每个线程独立保存这个信息的场景下时（比如每个线程都会对应一个用户信息，也就是 user 对象），在这种场景下，ThreadLocal 侧重的是避免传参，所以此时 ThreadLocal 和 synchronized 是两个不同维度的工具。

以上就是本课时的内容。

在本课时中，首先介绍了 ThreadLocal 是不是用来解决共享资源的多线程访问的问题的，答案是“不是”，因为对于 ThreadLocal 而言，每个线程中的资源并不共享；然后我们又介绍了 ThreadLocal 和 synchronized 的关系。

上一页 下一页