

# 11 | Java线程（下）：为什么局部变量是线程安全的？

王宝令 2019-03-23




00:00

07:38

讲述：王宝令 大小：7.01M

我们一遍一遍重复再重复地讲到，多个线程同时访问共享变量的时候，会导致并发问题。那在 Java 语言里，是不是所有变量都是共享变量呢？工作中我发现不少同学会给方法里面的局部变量设置同步，显然这些同学并没有把共享变量搞清楚。那 Java 方法里面的局部变量是否存在并发问题呢？下面我们就先结合一个例子剖析下这个问题。

比如，下面代码里的 `fibonacci()` 这个方法，会根据传入的参数 `n`，返回 1 到 `n` 的斐波那契数列，斐波那契数列类似这样：1、1、2、3、5、8、13、21、34.....第 1 项和第 2 项是 1，从第 3 项开始，每一项都等于前两项之和。在这个方法里面，有个局部变量：数组 `r` 用来保存数列的结果，每次计算完一项，都会更新数组 `r` 对应位置中的值。你可以思考这样一个问题，当多个线程调用 `fibonacci()` 这个方法的时候，数组 `r` 是否存在数据竞争（Data Race）呢？

 复制代码

```
1 // 返回斐波那契数列
2 int[] fibonacci(int n) {
3     // 创建结果数组
4     int[] r = new int[n];
5     // 初始化第一、第二个数
6     r[0] = r[1] = 1; // ①
7     // 计算 2..n
8     for(int i = 2; i < n; i++) {
```

```
9         r[i] = r[i-2] + r[i-1];
10     }
11     return r;
12 }
13
```


你自己可以在大脑里模拟一下多个线程调用 `fibonacci()` 方法的情景，假设多个线程执行到 ① 处，多个线程都要对数组 `r` 的第 1 项和第 2 项赋值，这里看上去感觉是存在数据竞争的，不过感觉再次欺骗了你。

其实很多人也是知道局部变量不存在数据竞争的，但是至于原因嘛，就说不清楚了。

那它背后的原因到底是怎样的呢？要弄清楚这个，你需要一点编译原理的知识。你知道在 CPU 层面，是没有方法概念的，CPU 的眼里，只有一条条的指令。编译程序，负责把高级语言里的方法转换成一条条的指令。所以你可以站在编译器实现者的角度来思考“怎么完成方法到指令的转换”。

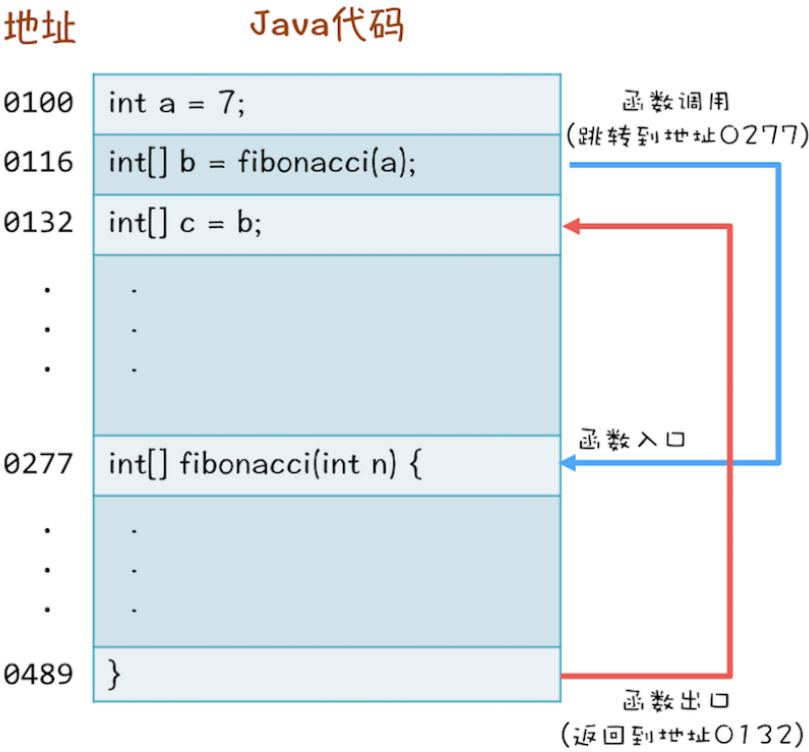
## 方法是如何被执行的

高级语言里的普通语句，例如上面的 `r[i] = r[i-2] + r[i-1]`；翻译成 CPU 的指令相对简单，可方法的调用就比较复杂了。例如下面这三行代码：第 1 行，声明一个 `int` 变量 `a`；第 2 行，调用方法 `fibonacci(a)`；第 3 行，将 `b` 赋值给 `c`。

 复制代码

```
1 int a = 7;
2 int[] b = fibonacci(a);
3 int[] c = b;
4
```

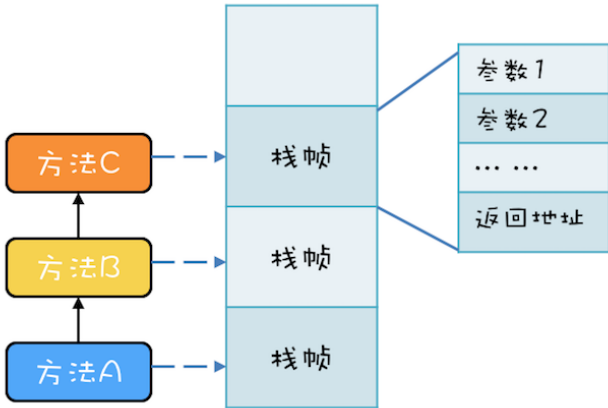
当你调用 `fibonacci(a)` 的时候，CPU 要先找到方法 `fibonacci()` 的地址，然后跳转到这个地址去执行代码，最后 CPU 执行完方法 `fibonacci()` 之后，要能够返回。首先找到调用方法的下一条语句的地址：也就是 `int[] c=b;` 的地址，再跳转到这个地址去执行。你可以参考下面这个图再加深一下理解。



方法的调用过程

到这里，方法调用的过程想必你已经清楚了，但是还有一个很重要的问题，“CPU 去哪里找到调用方法的参数和返回地址？”如果你熟悉 CPU 的工作原理，你应该会立刻想到：**通过 CPU 的堆栈寄存器**。CPU 支持一种栈结构，栈你一定很熟悉了，就像手枪的弹夹，先入后出。因为这个栈是和方法调用相关的，因此经常被称为**调用栈**。

例如，有三个方法 A、B、C，他们的调用关系是 A->B->C（A 调用 B，B 调用 C），在运行时，会构建出下面这样的调用栈。每个方法在调用栈里都有自己的独立空间，称为**栈帧**，每个栈帧里都有对应方法需要的参数和返回地址。当调用方法时，会创建新的栈帧，并压入调用栈；当方法返回时，对应的栈帧就会被自动弹出。也就是说，**栈帧和方法是共生共死的**。



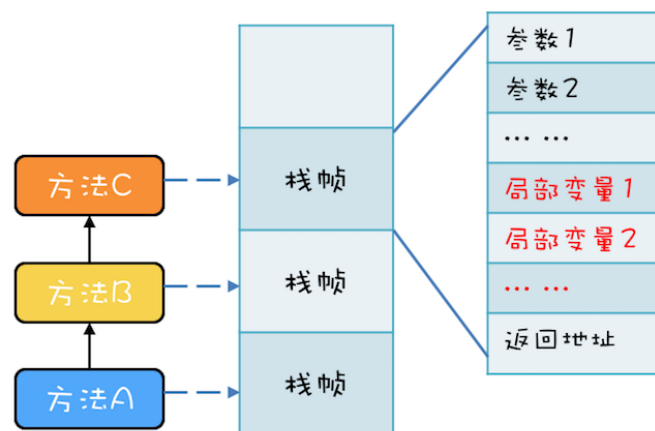
调用栈结构

利用栈结构来支持方法调用这个方案非常普遍，以至于 CPU 里内置了栈寄存器。虽然各家编程语言定义的方法千奇百怪，但是方法的内部执行原理却是出奇的一致：都是**靠栈结构解决**的。Java 语言虽然是靠虚拟机解释执行的，但是方法的调用也是利用栈结构解决的。

## 局部变量存哪里？

我们已经知道了方法间的调用在 CPU 眼里是怎么执行的，但还有一个关键问题：方法内的局部变量存哪里？

局部变量的作用域是方法内部，也就是说当方法执行完，局部变量就没用了，局部变量应该和方法同生共死。此时你应该会想到调用栈的栈帧，调用栈的栈帧就是和方法同生共死的，所以局部变量放到调用栈里那儿是相当的合理。事实上，的确是这样的，**局部变量就是放到了调用栈里**。于是调用栈的结构就变成了下图这样。

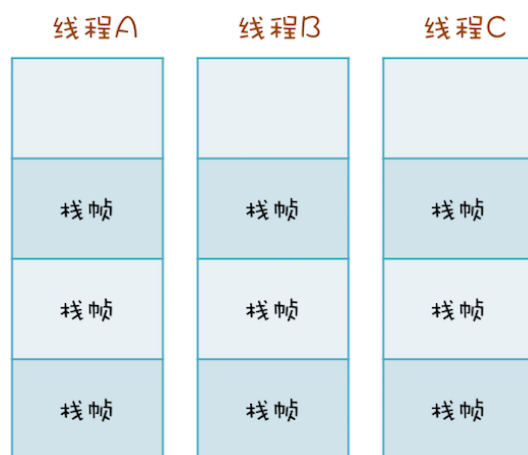


保护局部变量的调用栈结构

这个结论相信很多人都知道，因为学 Java 语言的时候，基本所有的教材都会告诉你 new 出来的对象是在堆里，局部变量是在栈里，只不过很多人并不清楚堆和栈的区别，以及为什么要区分堆和栈。现在你应该很清楚了，局部变量是和方法同生共死的，一个变量如果想跨越方法的边界，就必须创建在堆里。

## 调用栈与线程

两个线程可以同时用不同的参数调用相同的方法，那调用栈和线程之间是什么关系呢？答案是：**每个线程都有自己独立的调用栈**。因为如果不是这样，那两个线程就互相干扰了。如下面这幅图所示，线程 A、B、C 每个线程都有自己独立的调用栈。



线程与调用栈的关系图

现在，让我们回过头来再看篇首的问题：Java 方法里面的局部变量是否存在并发问题？现在你应该很清楚了，一点问题都没有。因为每个线程都有自己的调用栈，局部变量保存在线程各自的调用栈里面，不会共享，所以自然也就没有并发问题。再次重申一遍：没有共享，就没有伤害。

## 线程封闭

方法里的局部变量，因为不会和其他线程共享，所以没有并发问题，这个思路很好，已经成为解决并发问题的一个重要技术，同时还有个响当当的名字叫做**线程封闭**，比较官方的解释是：**仅在单线程内访问数据**。由于不存在共享，所以即便不同步也不会有并发问题，性能杠杠的。

采用线程封闭技术的案例非常多，例如从数据库连接池里获取的连接 Connection，在 JDBC 规范里并没有要求这个 Connection 必须是线程安全的。数据库连接池通过线程封闭技术，保证一个 Connection 一旦被一个线程获取之后，在这个线程关闭 Connection 之前的这段时间里，不会再分配给其他线程，从而保证了 Connection 不会有并发问题。

## 总结

调用栈是一个通用的计算机概念，所有的编程语言都会涉及到，Java 调用栈相关的知识，我并没有花费很大的力气去深究，但是靠着那点 C 语言的知识，稍微思考一下，基本上也就推断出来了。工作了十几年，我发现最近几年和前些年最大的区别是：很多技术的实现原理我都是靠推断，然后看源码验证，而不是像以前一样纯粹靠看源码来总结了。

建议你也多研究原理性的东西、通用的东西，有这些东西之后再学具体的技术就快多了。

## 课后思考

常听人说，递归调用太深，可能导致栈溢出。你思考一下原因是什么？有哪些解决方案呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。



© 版权归极客邦科技所有，未经许可不得转载



由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表

0/2000字

提交留言

## 精选留言(1)



西西弗与卡夫卡

因为调用方法时局部变量会进线程的栈帧，线程的栈内存是有限的，而递归没控制好容易造成太多层次调用，最终栈溢出。

解决思路一是开源节流，即减少多余的局部变量或扩大栈内存大小设置，减少调用层次涉及具体业务逻辑，优化空间有限；二是改弦更张，即想办法消除递归，比如说能否改造成尾递归（Java会优化掉尾递归）

👍 3    2019-03-23