多线程那些事，硬核有趣

小林coding *2020-12-12*

01

起点

小白，坐在这间属于华夏国超一流互联网公司企鹅巴巴的小会议室里，等着技术面试官的到来。

02

突如其来的面试

Round 1

科学家路人S：

小伙子我看你简历上什么也没写，这次也是第一面，那我们就随便问点简单的多线程问题吧。先说说什么是Java的多线程吧，使用多线程有什么好处？有什么坏处？

小白

妈妈说专家的话不能信！果然，问个多线程还问好处坏处？我不想用不会用能进企鹅巴巴么？

但是作为打工人，我认真的回答道：

Java的多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务。

而使用多线程的好处是可以**提高 CPU 的利用率**。在多线程程序中，一个线程必须等待的时候，CPU 可以运行其它的线程而不是等待，这样就大大提高了程序的效率。也就是说允许单个程序创建多个**并行**执行的线程来完成各自的任务。

至于多线程的坏处么，主要有三点。第一点是线程也是程序，所以线程需要占用内存，线程越多占用内存也越多；第二点是多线程需要协调和管理，所以需要 CPU 时间跟踪线程；最后是线程之间对共享资源的访问会相互影响，必须解决竞用共享资源的问题。

Round 2

科学家路人S继续追问：

你刚才讲了“并行”这个词，那你说说并行和并发有什么区别？

小白

并发，英文单词是concurrency，就是多个任务在同一个 CPU 核上，按细分的时间片轮流(交替)执行，从逻辑上来看那些任务是同时执行。

并行，英文单词是parallelism，就是单位时间内，多个处理器或多核处理器同时处理多个任务，是真正意义上的“同时进行”。

这两句话，我相信99%的同学都知道！但是，如果想进企鹅巴巴，如果想应付P20的科学家！我就一定要自行的结合业务回答并发并行的优势！

现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力以及性能。面对复杂业务模型，并行程序会比串行程序更适应业务需求，而并发编程更能吻合这种业务拆分 。

Round 3

路人S和路人B果然都露出了**满意的笑容。**

路人B开始追问道：

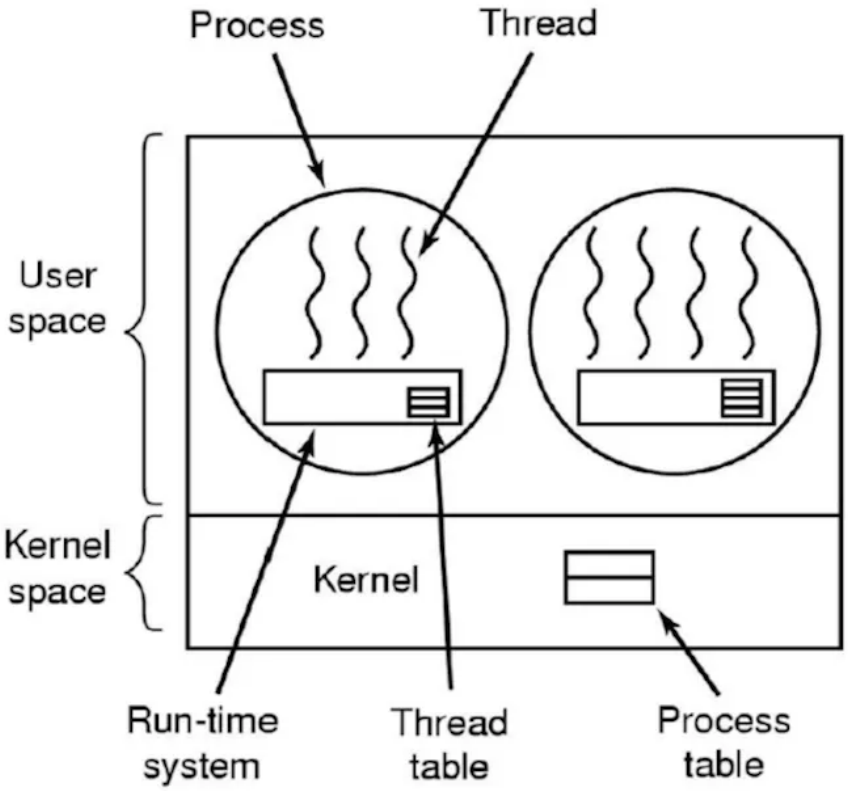
那你说说看，在操作系统中用户级线程和内核级线程是什么？这两个线程在多核CPU的计算机上是否都能并行？

小白

在操作系统的设计中，为了防止用户操作敏感指令而对OS带来安全隐患，我们把OS分成了用户空间（user space）和内核空间(kernel space)。

通过用户空间的库类实现的线程，就是**用户级线程（user-level threads，ULT）**。这种线程不依赖于操作系统核心，进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。

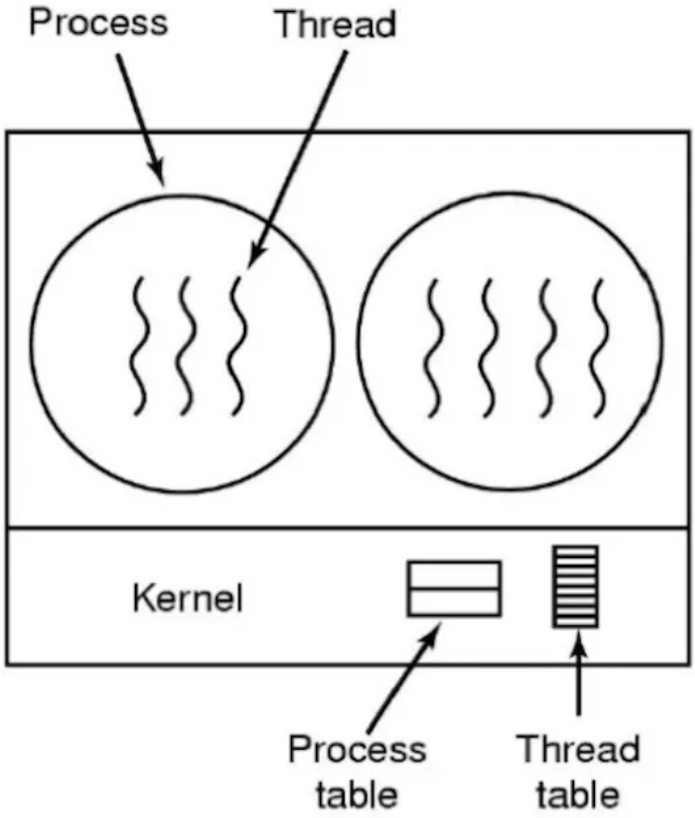
说着，我拿了一支笔，画了这么一张图：



在图里，我们可以清楚的看到，线程表（管理线程的数据结构）是处于进程内部的，完全处于用户空间层面，内核空间对此一无所知！**当然，用户线程也可以没有线程表！**

相应的，由OS内核空间直接掌控的线程，称为**内核级线程(kernel-level threads，KLT)**。其依赖于操作系统核心，由内核的内部需求进行创建和撤销。

接着，我画下了这张图：



同样的，在图中，我们看到内核线程的线程表(thread table)位于内核中，包括了线程控制块(TCB)，一旦线程阻塞，内核会从当前或者其他进程(process)中重新选择一个线程保证程序的执行。

对于用户级线程来说，其线程的切换发生在用户空间，这样的线程切换至少比陷入内核要快一个数量级。但是该种线程有个严重的缺点：如果一个线程开始运行，那么该进程中其他线程就不能运行，除非第一个线程自动放弃CPU。因为在一个单独的进程内部，没有时钟中断，所以不能用轮转调度（轮流）的方式调度线程。

也就是说，同一进程中的用户级线程，在不考虑调起多个内核级线程的基础上，是没有办法利用多核CPU的，其实质是**并发而非并行。**

对于内核级线程来说，其线程在内核中创建和撤销线程的开销比较大，需要考虑上下文切换的开销。

但是，**内核级线程是可以利用多核CPU的，即可以并行**！

这回答的累死我了，不过为了能进企鹅巴巴，走向人生巅峰，一切都值了！

Round 4

路人B点了点头说：

嗯，小伙子基础还是比较牢靠的！那你说说Java里的多线程是用户级线程还是内核级线程呢？

小白

是...当我要脱口而出的时候，发现不对，这面试官在套路我！堂堂科学家，套路还没入职的孩子么？

Java里的多线程，既不是用户级线程，也不是内核级线程！

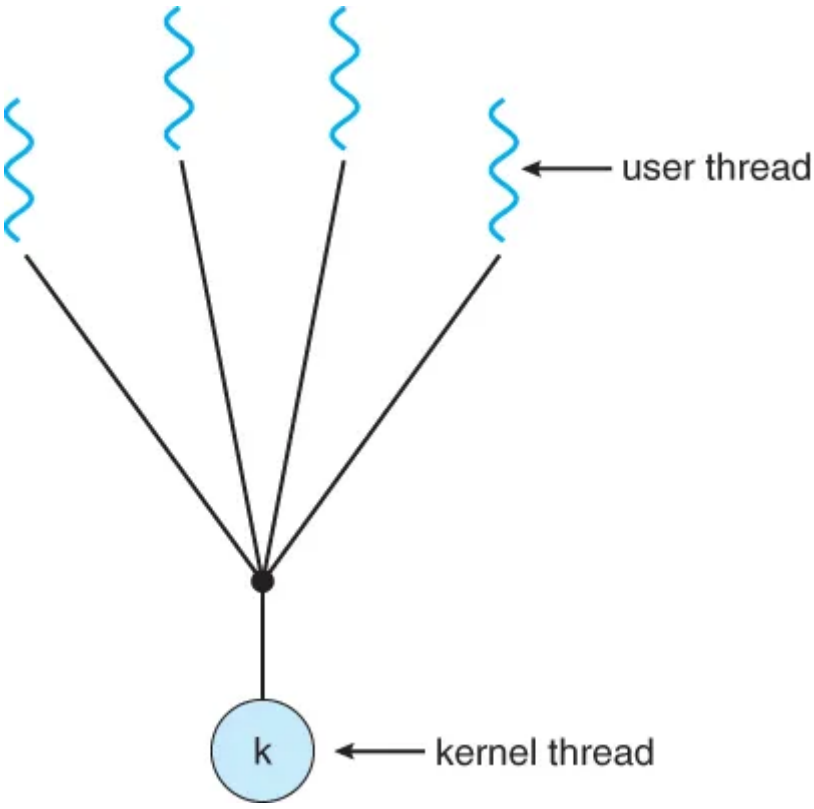
首先，Java是跨操作平台的语言，是使用JVM去运行编译文件的。不同的JVM对线程的实现不同，相同的JVM对不同操作平台的线程实现方式也有区别！

其次，要讲明白程序级别实现多线程，就必须先说一下多线程模型。

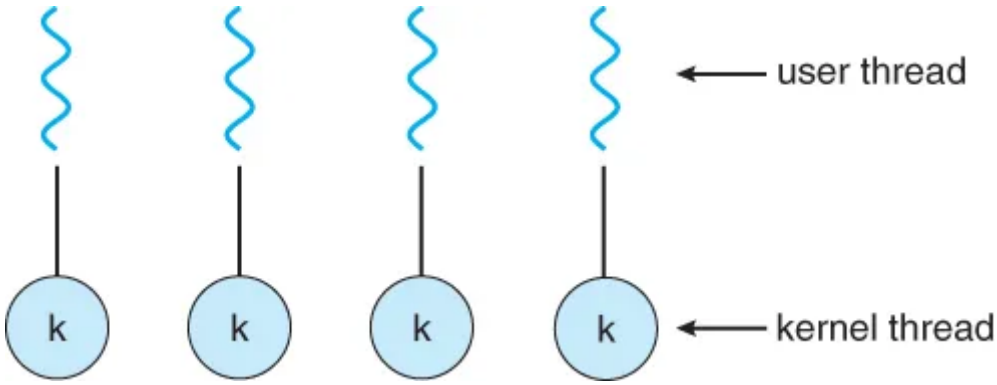
裂开！怎么感觉这又是一道大题啊！B是操作系统的科学家吧！感觉问的都是很底层的东西了啊，现在程序员内卷成这样了么？实习生都问这么底层的问题了？虽然百般不爽，但是为了拿下美女HR，不！是横扫offer。我要给路人B讲明白这个线程模型！

上面我说过OS上的线程分为ULT和KLT，我们写程序的代码只能是在用户空间里写代码！而程序运行中，基本上都会进入内核运行，所以我们在实现程序级别多线程的时候，必须让ULT映射到KLT上去。在程序级别的多线程设计里，有以下三种多线程模型。

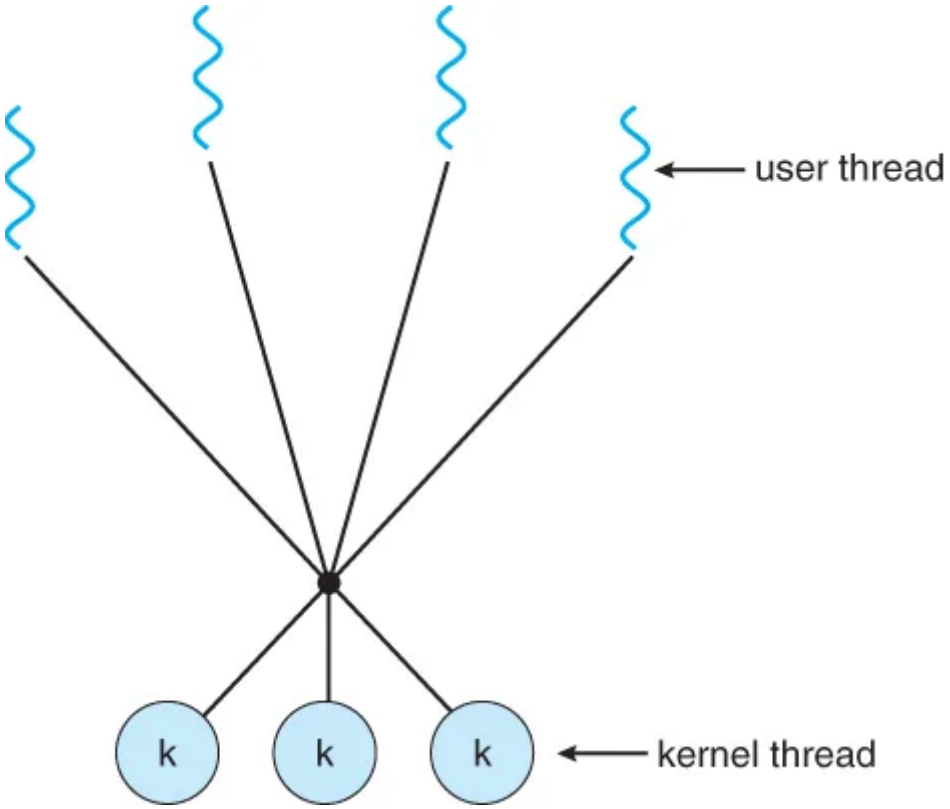
多对1模型：在多对一模型中，多个ULT映射到1个KLT上去，此时ULT的进程表处于进程之中。



1对1模型：在一对一模型中，1个ULT对应1个KLT。自己不在进程中创建线程表来管理，几行代码之后直接通过**系统调用**调起KLT就能实现。



多对多模型：在多对多模型中，N个ULT对应小于等于N个的KLT。这种模型结合了1对1和多对1的优点，用户创建线程没有限制，阻塞内核系统的命令不会阻塞整个进程。



最后，就拿最热门的HotSpot VM来说吧，他在Solaris上就有两种线程实现方式，可以让用户选择一对一或多对多这两种模型；而在Windows和Linux下，使用的都是一对一的多线程模型，Java的线程通过一一映射到Light Weight Process（轻量级进程，LWP）从而实现了和KLT的一一对应。

Round 5

路人B听到这个回答，眼睛都亮了！直接追问道：

ULT如何映射到KLT？怎么调起的？

小白

ULT在执行的过程中，如果执行的指令需要进入内核态，则ULT会通过**系统调用**调起一个KLT！

所谓系统调度，就是在OS中分割用户空间和内核空间的API。

Round 6

路人B继续追问道：

ULT的执行过程中可以不调起KLT么？举个例子。

小白

可以不调起，比如ULT中就只有sleep这个指令，就不会进入内核态执行，更不会调起KLT。

问到这里，我有点吐血了都！看着B对我的回答很满意，我心中却把B已经问候了一百遍！

Round 7

路人S总算接过了话题：

看来同学对于底层的知识理解还凑合，那你有没有看过HotSpot的源码？能不能简单说说看Java的线程是怎么运行的？

小白

这问的还上瘾了？P20的问题咋这么“简单”呢！说实话，自从前几天发生了灵异事件之后，我确实技术突飞猛进，这个源代码我好像还真的瞄了一眼，不过我不能暴露自己拥有金手指的秘密啊！

于是我挠了挠头，思考了1分钟，然后说道：

源码以前看过，只能记得一个大概。

1、在Java中，使用java.lang.Thread的构造方法来构建一个java.lang.Thread对象，此时只是对这个对象的部分字段(例如线程名，优先级等)进行初始化；

2、调用java.lang.Thread对象的start()方法，开始此线程。此时，在start()方法内部，调用start0() 本地方法来开始此线程；

3、start0()在VM中对应的是JVM_StartThread，也就是，在VM中，实际运行的是JVM_StartThread方法(宏),在这个方法中，创建了一个JavaThread对象；

4、在JavaThread对象的创建过程中，会根据运行平台创建一个对应的OSThread对象，且JavaThread保持这个OSThread对象的引用；

5、在OSThread对象的创建过程中，创建一个平台相关的底层级线程，如果这个底层级线程失败，那么就抛出异常；

6、在正常情况下，这个底层级的线程开始运行，并执行java.lang.Thread对象的run方法；

7、当java.lang.Thread生成的Object的run()方法执行完毕返回后，或者抛出异常终止后，终止native thread;

8、最后就是释放相关的资源(包括内存、锁等)

大概就是以上这么个步骤吧。

回答完这个，我要跪谢我的金手指了！我看见路人S在电脑上敲着什么，估计他也比较懵，没想到我居然能答得上来吧！

Round 8

路人S对此不置可否，说道：

那你说说什么是上下文切换吧。

小白

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。

时间片是CPU分配给各个线程的时间，因为时间非常短，所以CPU不断通过切换线程，让我们觉得多个线程是同时执行的，时间片一般是几十毫秒。

当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。

概括来说就是：**当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。任务从保存到再加载的过程就是一次上下文切换。**

Round 9

路人S继续问道：

频繁切换上下文会有什么问题？

小白

上下文切换通常是计算密集型的，每次切换时，需要保存当前的状态起来，以便能够进行恢复先前状态，而这个切换时非常损耗性能。

也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

Round 10

S继续问：

减少上下文切换的方式有哪些?

小白

通常减少上下文切换的方式有：

1、**无锁并发编程**：可以参照concurrentHashMap锁分段的思想，不同的线程处理不同段的数据，这样在多线程竞争的条件下，可以减少上下文切换的时间。

2、**CAS算法**：利用Atomic下使用CAS算法来更新数据，使用了乐观锁，可以有效的减少一部分不必要的锁竞争带来的上下文切换。

3、**使用最少线程**：避免创建不需要的线程，比如任务很少，但是创建了很多的线程，这样会造成大量的线程都处于等待状态。

4、**协程**：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。

Round 11

路人B听了，眼睛一亮，立刻追问道：

协程是什么？和用户线程有什么区别?

小白

我听了真想抽自己几个嘴巴子，怎么又来了！B是只会OS吧！

协程的英文单词是Coroutine，这是一个程序组件，它既不是线程也不是进程。它的执行过程更类似于一个方法，或者说不带返回值的函数调用。

我看到过stack overflow和很多博客里，都认为这两者是一个东西。但是，在我的理解中，这两者还是有区别的。

不可否认的是，**协程和ULT做的是同一个事情。**所以从某种角度上讲，他们确实是**等价**的！

但是，**ULT这个概念被提出的时候，其背后的思想本质是讲ULT是个本机线程**，也就是使用了OS的用户空间内提供的库类直接创建的线程。这个时候，你不需要在OS上面添加一些其他第三方的库类。

而协程这个概念是康威定律的提出者Melvin Edward Conway在1958年提出的一个概念，其背后的思想是**不直接使用OS本身的库类，自己做一些库类去实现并发。**在那个年代，OS上面的第三方库类并不像现在这么流行，OS本身的库类和其他第三方库类的结合也并不像今天这么容易。所以协程并不是本机线程，他是需要借助一些其他不属于OS的第三方库类调用OS用户空间的库类来实现达到ULT的效果。

当然，这个概念在今天来看，就会显得很让人混淆了。因为到底哪些库类算是OS本机的库类，哪些算是第三方库类？这和1960年的时候已经有绝大的区别了！所以大家认为这两者是一个东西，其实也不能说他说的不对，只能说可能对这个思想本身背后代表的东西不明白。

Round 12

路人B听了，立刻坐直了身体，继续追问道：

那你知道fiber么？这个和上面两个名词有什么区别？

小白

fiber也是一种**本机线程**，其本质是一种**特殊的ULT**，即更轻量级的ULT。说白了就是**这种ULT的线程表一定存于进程之中。**

而我们在构建一对一多线程模型的时候，ULT的线程表其实还是交给内核了！这是两者之间最直接的差别。所以我们经常称fiber就是协同调度的ULT，在win32中可以调用fiber来构建多对多的多线程模型。

其实，fiber、coroutine和ULT在用户层面能看到的效果是**基本等价**的。

其中ULT是描述OS库本身提供的功能；fiber描述的是OS提供的协同调度的ULT；coroutine描述的是第三方实现的并发并行功能。

这些名词很多都是历史原因的问题，同时也是深入研究需要了解的事情，我们普通程序员在使用的时候，更多的关心的是应用层方面的东西。而这些名词的理解已经深入到源码层了。

Round 13

路人S估计被我秀的脑壳痛了，立刻说道：

还是讲讲看在 Java 程序中怎么保证多线程的运行安全吧。

小白

Java的线程安全在三个方面体现：

原子性：提供互斥访问，同一时刻只能有一个线程对数据进行操作，在Java中使用了atomic和synchronized这两个关键字来确保原子性；

可见性：一个线程对主内存的修改可以及时地被其他线程看到，在Java中使用了synchronized和volatile这两个关键字确保可见性；

有序性：一个线程观察其他线程中的指令执行顺序，由于指令重排序，该观察结果一般杂乱无序，在Java中使用了happens-before原则来确保有序性。

Round 14

路人S继续问道：

你刚才讲了有序性，那你说说代码为什么会重排序?

小白

在执行程序时，为了提高性能，处理器和编译器常常会对指令进行重排序。

Round 15

路人S继续追问：

重排序是想怎么重排就重排么？

小白

这面试官也很难缠啊，怎么一直在追问，是需要我给他孝敬一根华子么？要不是看着旁边有个美女HR，我早就孝敬S他老人家了！

当然不是！不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

1、在单线程环境下不能改变程序运行的结果；

2、存在数据依赖关系的不允许重排序。

所以重排序不会对单线程有影响，只会破坏多线程的执行语义。

Round 16

路人S继续追问道：

那你讲讲看在Java中如何保障重排序不影响单线程的吧。

小白

保障这一结果是因为在编译器，runtime 和处理器都必须遵守as-if-serial语义规则。

为了遵守as-if-serial语义，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，这些操作可能被编译器和处理器重排序。

我来举个例子吧

说着我拿着笔在纸上写了三行简单的代码：

```
1   double width  = 15.67;          //A
2   double height = 14.32;          //B
3   double area = width * height;   //C
```

我们看这个例子，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，如果C排到A和B的前面，那么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。

这就是as-if-serial语义。

Round 17

路人S继续问道：

那你说说看你刚才讲的happens-before原则吧。

小白

happens-before说白了就是谁在谁前面发生的一个关系。

HB规则是Java内存模型（JMM）向程序员提供的跨线程内存可见性保证。

说的直白一点，就是如果A线程的写操作a与B线程的读操作b之间存在happens-before关系，尽管a操作和b操作在不同的线程中执行，但JMM向程序员保证a操作将对b操作可见。

具体的定义为：

1、如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。

2、两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行。如果重排序之后的执行结果，与按happens-before关系来执行的结果一致，那么这种重排序并不非法。

具体的规则有8条：

1、程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。

2、监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。

3、volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

4、传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

5、start()规则：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作。

6、Join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回。

7、程序中断规则：对线程interrupted()方法的调用先行于被中断线程的代码检测到中断时间的发生。

8、对象finalize规则：一个对象的初始化完成（构造函数执行结束）先行于发生它的finalize()方法的开始。

Round 18

路人S接着追问：

你刚才说HB规则不代表最终的执行顺序，能不能举个例子。

小白

就拿讲as-if-serial提到的例子举例吧，例子很简单就是面积=宽*高。

利用HB的程序顺序规则，存在三个happens-before关系：

1、A happens-before B；

2、B happens-before C;

3、A happens-before C。

这里的第三个关系是利用传递性进行推论的。这里的第三个关系是利用传递性进行推论的。

A happens-before B，定义1要求A执行结果对B可见，并且A操作的执行顺序在B操作之前；但与此同时利用HB定义中的第二条，A、B操作彼此不存在数据依赖性，两个操作的执行顺序对最终结果都不会产生影响。

在不改变最终结果的前提下，允许A，B两个操作重排序，即happens-before关系并不代表了最终的执行顺序。

推荐阅读

[你不好奇 Linux 是如何收发网络包的?](#)

[小小的 float，藏着大大的学问](#)

我以前写过一篇学习经验，你可以看看：https://mp.weixin.qq.com/s/yopn5PqC7ESQcxWOj0eBoQ