链滴

Java 内存模型

作者：AlanSune

原文链接：https://ld246.com/article/1591530776491

来源网站：链滴

介绍

Java支持多线程执行，在语言层面使用Thread类表示。用户创建线程的唯一方式就是创建一个该类的象，每个线程都与这样一个对象相关联。在对应的Thread对象上调用start()方法将启动线程。

Java允许编译器和处理器进行优化，这会使未正确同步的程序表现出出人意料的结果。

Thread 1	**Thread 2**
1: r2 = A	3: r1 = B
2: B = 1	4: A = 2

考虑上图的例子，假设初始值A = B = 0，并且A和B是线程共享的，r1和r2是局部变量。可能会出现r1 == 1, r2 == 2这样的结果。从直觉上，要么指令1先执行，此时r2不应该看到指令4的结果；要么指令先执行，此时r1不应该看到指令2的结果。出现上述结果，那么应该有这样的执行顺序：**4 -> 1 -> 2 > 3 -> 4**，这样指令4既是第一条执行指令，也是最后一条执行指令，这自相矛盾。

由于Java允许编译器和处理器进行优化，那么如果指令4发生在指令3之前，即发生了**重排序**，一切就情合理了。从单线程的角度来看，只要重排序不影响线程的执行结果，Java就允许这样的操作。

as-if-serial语义

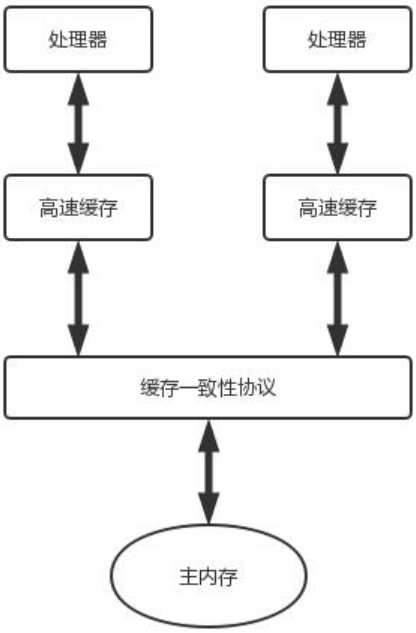
as-if-serial字面含义为**与串行似的**，其语义为编译器、运行时和硬件应该协同工作，以创建"as-if-serial"语义的假象，这意味着在单线程程序中，程序不应该能够观察重排序的效果。然而，在不正确同步多线程程序中，重新排序可能会发挥作用，一个线程能够观察到其他线程的影响，并且可能能够检测变量访问对其他线程以不同于执行或程序中指定的顺序变得可见。

物理平台的内存模型

在当前物理计算机中，多处理器体系架构已成为常态。在处理器运行的过程中，数据的获取和存储必

可少，然而由于存储设备的读取速度和处理器的运算速度相差较多，导致处理器不能充分的发挥自己性能，所以当前计算机都会在处理器和内存之间增加高速缓存。每个处理器都会拥自己的缓存，定期主内存进行协调。

增加缓存虽然有效的提高的处理器的效率，同时也为多处理器架构引入了新的问题。每个处理器的缓都只与主内存发生数据交换，而不能与其他缓存直接进行通信。如果多个处理器同时处理相同的内存那么可能导致每个缓存会出现不同的数据，这就是**缓冲一致性**。为了解决一致性问题，不同平台通过同的一致性协议来保证数据正确的同步回主内存。物理平台的交互关系如下图。



除了缓存的问题之外，当前处理器为了充分利用自己的性能，会对输入代码进行乱序执行。处理器会证最终的执行结果与顺序执行的结果一致，但不对执行顺序保证。例如针对代码：

```
a = 1;
b = 2;
c = a;
```

处理器为了优化性能，可能会按照以下顺序执行：

```
a = 1;
c = a;
b = 2;
```

在原顺序中，处理器需要读取a变量两次，这在性能上会造成很大的影响（想想处理器从主内存中读两次a的时间消耗）。如果处理器在执行中调整为重排序后的顺序，假设此时处理器执行完**a = 1**后，以将a的值缓存，这样就减少了性能消耗。从最终的结果上来看，结果保持了一致性，但是执行顺序原有代码并不相同。对于单线程来说，这样的顺序并不会引发问题，然而在多线程中，如果某个线程处理依赖其他线程的执行，那么就会出现严重的问题。

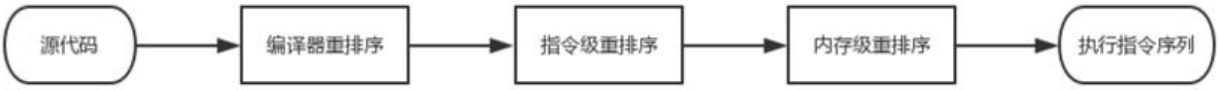
重排序

重排序即访问程序变量(对象实例字段、类静态字段和数组元素)的次序可能与程序指定的次序不同。译器可以自由地以优化的名义对指令进行排序。在某些情况下，处理器可能会无序地执行指令。数据

以在寄存器、处理器缓存和主存之间以不同于程序指定的顺序移动。

在上面已经描述了重排序可能导致的执行问题。例如，如果一个线程写字段**a**，然后写字段**b**，而**b**的不依赖于**a**的值，那么编译器可以自由地对这些操作重新排序，而缓存可以在**a**之前将**b**刷新到主存。许多重新排序的潜在来源，例如编译器、JIT编译器和缓存。

在此介绍下在Java体系中涉及的重排序类型。从java源码到实际执行的过程中，会经历一下三种重排：



- 编译器重排序：在不影响单线程执行过程的前提下，编译器重新安排执行顺序
- 指令级重排序：现代处理器采用指令并行执行，数据之间如果不存在数据依赖，那么处理器会通过令重排提高性能
- 内存级重排序：由于处理器缓存和读/写缓冲区的存在，会导致指令执行与看上去的顺序不一致。

其中编译器重排序，是由java编译器在编译过程中进行的指令重排，属于语言级别。指令级和内存级排序由硬件系统进行，不同的处理器会产生不同的处理结果。下面介绍上述重排序实例，给大家有个观的理解。

编译器重排序

当前JDK自带的javac工具在编译成字节码的过程中，不会对代码进行编译的优化。下面的示例使用 **hsis** 反编译工具，获取C2类型的JIT编译器生成的汇编指令，来展示JVM在运行中，由即时编译器造成重排序。该工具的使用，我会在其他文章中进行简单介绍。

源代码如下所示：

```
public class Test {
    int sum = 0;
    boolean flag = false;

    private void add(int param) {
        sum += 1;
        flag = true;
        sum += param;
    }

    public static void main(String[] args) {
        Test test = new Test();
        test.add(100);
    }

}
```

参考生成编译的指令（需安装 **hsdis** ）

```
javac Test.java
java -Xcomp -XX:CompileCommand=dontinline,Test.add -XX:CompileOnly=Test.add -XX:Co
```

```
pileCommand=print,Test.add Test
```

C2编译器编译后，**add** 方法的汇编指令如下，此处只展示部分重要指令，该指令使用jdk11生成。

其中前四行由 **hsdis** 工具生成，第二行表示 **Test** 的实例对象 **test** 的地址保存在 **rdx** 寄存器中，第行表示 **param** 参数值保存在 **r8** 寄存器中。

```
# {method} {0x000001b4f9910398} 'add' '(I)V' in 'Test'
# this:     rdx:rdx   = 'Test'
# parm0:    r8        = int
#           [sp+0x20]  (sp of caller)

sub    $0x18,%rsp
mov    %rbp,0x10(%rsp)
add    0xc(%rdx),%r8d   # 0xc(%rdx)表示test对象所在地址（rdx寄存器保存test对象的地址）移动0
c字节处的地址，此处为字段num的地址。该指令表示num和param相加，结果保存在r8寄存器中
movb   $0x1,0x10(%rdx)  # 将0x1（即十六进制的1）保存到test对象所在地址（rdx寄存器保存test
象的地址）移动0x10字节处的地址处，即变量flag赋值为true
inc    %r8d             # r8寄存器中的值自增加1，即sum += 1的部分操作
mov    %r8d,0xc(%rdx)   # 将r8寄存器的值写回est对象所在地址（rdx寄存器保存test对象的地址）
动0xc字节处的地址处，即将num + 1 + param的值写回内存
add    $0x10,%rsp
pop    %rbp
mov    0x108(%r15),%r10
test   %eax,(%r10)
retq
```

由上面的指令可知，**add** 方法在编译后，先执行 **num += param;**, 后执行 **num += 1;** 。此处虽然重排序，但是更重要的一点是，在执行完上述写回内存的操作前，**num** 的值都保存在寄存器中。这造成其他线程在获取 **num** 时，只能获取到初始值0或者add方法执行结束的值101，中间过程的值根没有保存回内存。

指令级重排序

CPU的基本工作是执行存储的指令序列，即程序。程序的执行过程实际上是不断地取出指令、分析指、执行指令的过程。一条CPU指令在执行中可以分为5个阶段：取指令、指令译码、执行指令、访存数和结果写回。

在串行的指令执行方式下，一个指令周期只能执行一条指令。如果在对第一条指令译码的时候，就取二条指令；第二条指令译码的时候，就取第三条指令。在完美的条件下，指令就可以像流水线一样进执行，这就是指令流水线技术。

然而由于数据之间存在相互依赖关系，所以上述的执行方式就存在一定的问题。比如如下的汇编指令：

```
指令1：ADD %r8d, %r10d  # 寄存器r8的值和寄存器r10的值相加，写入r10
指令2：inc %r10d        # 寄存器r10的值自增1
指令3：mov $0x10,(%rdx) # 10 写入寄存器rdx中指向的地址
```

由于指令2的操作数依赖指令1的执行结果，那么在指令1的执行完成前，指令2是不可以获取变量的值。假如将指令3提前到指令2之前，那么在指令2执行到取值的阶段，指令1的结果已写入寄存器 **r10**，么就可以完美实现流水线的执行过程。指令重排序的实际执行结果如下：

```
指令1：ADD %r8d, %r10d  # 寄存器r8的值和寄存器r10的值相加，写入r10
指令3：mov $0x10,(%rdx) # 10 写入寄存器rdx中指向的地址
```

指令2：inc %r10d　　# 寄存器r10的值自增1

可以看到，由于流水线技术，处理器可能乱序执行，造成重排序的结果。

内存级重排序

详情见编译器重排序，其中 **num** 在 **add** 方法中的中间操作值根本没有保存到内存中，而是保存在寄器中间。假如没有发生编译器重排序，在 **num += param;** 执行前，有其他线程想取得 **num += 1;** 后 **num** 的值，由于寄存器的存在，这个值在其他线程根本是不可见的。

总结

通过上述介绍可知，java在要求在单线程中保证**as-if-serial**,对多线程的执行并没有增加特殊的要求。java本意是为java虚拟机的实现者提供尽量大的自由度，保证java在运行时能最大限度的利用现代处理优化的功能。同时这也造成了java多线程在未正确同步的情况下，执行乱序的结果。本章通过一部分例，来演示java多线程执行的复杂情况，为下面的章节提供必要的前提知识。