

转载请注明链接:

Java内存模型

版本号:2018/10/02-(0:00)

- [Java内存模型](#)
 - [实例](#)
 - [重排序](#)
 - [内存模型](#)
 - [happen-before](#)
 - [线程内](#)
 - [线程间](#)
 - [volatile](#)
 - [问题汇总](#)
 - [参考资料](#)
 - [交流群](#)



实例

1、下面的代码，在单线程和多线程中执行，(r1、r2)会有哪些可能？

```
int a=0, b=0;

public void method1() {
    int r1 = b;
    a = 1;
}

public void method2() {
    int r2 = a;
    b = 2;
}
```

1. 单线程: (0、1)、(2、0)
2. 多线程: (0、1)、(2、0)、(0、0)、(2、1)

2、为什么会出现这种情况(2、1)?

原因有三个:

1. 即时编译器的重排序
2. 处理器的乱序执行
3. 内存系统的重排序。由于后两种原因涉及具体的体系架构

[Java 语言规范第 17.4 小节](#)

3、处理器的乱序执行和内存系统的重排序会涉及具体的体系架构

重排序

1、即时编译器的重排序是什么?即时编译器（和处理器）需要遵守的 as-if-serial原则是什么?

1. 即时编译器的重排序是: 对指令执行顺序重新排序。
2. as-if-serial原则就是在单线程中，具有顺序执行的假象。
3. 即时编译器必须要保证:
 1. 经过重排序的执行结果要与顺序执行的结果保持一致。

2、什么情况下即时编译器和cpu不会对指令重排序?为什么?

1. 如果两个操作之间存在数据依赖，那么即时编译器（和处理器）不能调整它们的顺序
2. 否则将会造成程序语义的改变。

3、下面的代码即时编译器会如何处理?

```
int a=0, b=0;

public void method1() {
    int r2 = a;
    b = 1;
    .. // Code uses b
    if (r2 == 2) {
        ..
    }
}
```

即时编译器有两种选择。

1. 第一: 一开始便将a加载至某一寄存器中，并且在接下来b的相关代码中 避免使用该寄存器。
2. 第二: 一开始不加载a，在真正使用r2时才将a加载至寄存器中。在执行使用 b 的代码时，不再额外占用一个通用寄存器，从而减少需要借助栈空间的情况。

4、栈空间是什么?

5、下面代码的场景中，即时编译器会怎么做？

```
int a=0, b=0;

public void method1() {
    for (..) {
        int r2 = a;
        b = 1;
        .. // Code uses r2 and rewrites a
    }
}
```

1. b的赋值和循环无关，会移到循环外部。
2. r2的赋值和使用，会保留在循环内

6、即时编译器的重排序为什么在多线程中会出问题？

1. 这种情况叫做数据竞争(data race)
2. 需要应用进行恰当的同步操作

内存模型

7、java内存模型是什么？

1. 为了避免数据竞争导致的干扰
2. java 5引入java内存模型
3. 引入的重要概念是happen-before关系

happen-before

8、happen before关系是指什么？

1. 用于描述两个操作的内存可见性
2. 如果操作x，happen before 操作y，x对于y就具有可见性

9、happens-before 关系还具备传递性。

1. 如果操作 X happens-before 操作 Y
2. 并且操作 Y happens-before 操作 Z
3. 那么操作 X happens-before 操作 Z

线程内

9、JVM规范如何定义的线程内的happen before关系？

1. 在同一个线程中，字节码的先后顺序（program order）也暗含了 happens-before 关系：

2. 在程序控制流路径中靠前的字节码happens-before 靠后的字节码。
3. 然而，这并不意味着前者一定在后者之前执行
4. 如果后者没有数据依赖于前者，那么可能会被重排序。

线程间

10、Java 内存模型定义了哪些程间的 happensbefore 关系？

1. 解锁操作 happens-before 之后（这里指时钟顺序先后）对同一把锁的加锁操作。
2. volatile 字段的写操作 happens-before 之后（这里指时钟顺序先后）对同一字段的读操作。
3. 线程的启动操作（即 Thread.starts()）happens-before 该线程的第一个操作。
4. 线程的后一个操作 happens-before 它的终止事件（即其他线程通过 Thread.isAlive() 或 Thread.join() 判断该线程是否中止）。
5. 线程对其他线程的中断操作 happens-before 被中断线程所收到的中断事件（即被中断线程的 InterruptedException 异常，或者第三个线程针对被中断线程的 Thread.interrupted 或者 Thread.isInterrupted 调用）。
6. 构造器中的后一个操作 happens-before 析构器的第一个操作。

11、时钟顺序的先后是什么？

12、JVM规范定义的线程内，线程间的happen-before关系表明前一个指令一定在后一个指令前执行？

1. 不是，如果后一个操作没有数据依赖于前一个操作，就可能会重排序。

volatile

1、第一个实例中happen before的关系如何？解释下为什么会出现 1 2的结果？

1. 根据JVM规范线程内，字节码的先后顺序暗含了happens-before关系
 1. method1: r1 = b, happens-before, a = 1
 2. method2: r2 = a, happens-before, b = 2
2. 这两对关系中的操作没有数据依赖，因此可以重排序:

Thread1	Thread2
b=1	
	r1=b
	a=2
r2=a	

2、如何解决这种问题？

使用volatile关键字来修饰变量

3、为什么volatile能起效果？

1. 这种跨线程的读写操作，即时编译器只能保证单线程的语义正常。可是多线程就无法保证了。先读后写，先写后读都可能被重排序。
2. 使用volatile关键字，等于定义了额外的happen before关系：
 - volatile规定写操作、读操作不进行重排序(无论原来是读写、写读，都不会再改变顺序)。
3. 根据happens-before的传递性：
 1. 如果是先读(r1=b)、后写(b=1), 那么r2=a, 一定在r1=b之后执行。a=2, 一定在b=1之前执行。
 2. 如果是先写(b=1)、后读(r1=b), 那么r2=a, 一定在r1=b之前执行。a=2, 一定在b=1之后执行。

// 1-先读(r1=b)、后写(b=1)

Thread1	Thread2
	r1=b
	a=2
b=1	
r2=a	

// 2-先写(b=1)、后读(r1=b)

Thread1	Thread2
b=1	
r2=a	
	r1=b
	a=2

4、解决数据竞争的关键技巧是什么？

1. 构造一种跨线程的happens-before关系，操作X happens-before Y
2. 使得 操作X 之前的字节码的结果，对 操作Y 之后的字节码可见。

5、如何判断两个操作之间是否有数据依赖关系？

1. 改变操作的顺序，会导致语义的改变。

6、Java内存模型的底层实现?采用什么实现的禁止重排序？

1. 通过内存屏障（memory barrier）来禁止重排序的。

7、即时编译器是如何使用内存屏障的？

1. 即时编译器，会针对每一个 happens-before 关系
2. 向正在编译的目标方法中插入相应的读读、读写、写读以及写写内存屏障。

8、内存屏障的作用？

1. 会限制即时编译器的重排序操作。
2. 以 volatile 字段访问为例, 所插入的内存屏障:
 1. 将不允许 volatile 字段写操作之前的内存访问被重排序至其之后;
 2. 将不允许 volatile 字段读操作之后的内存访问被重排序至其之前。

内存屏障在底层操作系统上的具体实现?

1. 即时编译器将根据具体的底层体系架构, 将这些内存屏障替换成具体的 CPU 指令。
2. 以 X86_64 架构为例, 读读、读写以及写写内存屏障是空操作 (no-op),
3. x86_64 中只有写读内存屏障会被替换成具体指令 [2]。

x86-64 中的写读内存屏障的具体指令是什么?

例子中的读写分析

1. method1 和 method2 之中的代码均属于先读后写 (假设 r1 和 r2 被存储在寄存器之中)。
2. x86-64, 不能将读操作重排序到写操作后面
3. 因此这是即时编译器, 进行的重排序

cpu 的重排序?

1. 和即时编译器的理念是一样的, 不同 cpu 处理不同
2. X86_64 架构的处理器并不能将读操作重排序至写操作之后, 具体可参考 Intel Software Developer Manual Volume 3, 8.2.3.3 小节。

volatile 字段, 即时编译器和 cpu 是怎么处理的?

1. 即时编译器将在 volatile 字段的读写操作前后各插入一些内存屏障。
2. 在 X86_64 架构上, 只有 volatile 字段写操作之后的写读内存屏障需要用具体指令来替代。(HotSpot 所选取的具体指令是 lock add DWORD PTR [rsp], 0x0, 而非 mfence[3]。)
3. 该具体指令的效果, 可以简单理解为强制刷新处理器的写缓存。
4. 写缓存是处理器用来加速内存存储效率的一项技术。

刷新写缓存是什么?

在碰到内存写操作时, 处理器并不会等待该指令结束, 而是直接开始下一指令, 并且依赖于写缓存将更改的数据同步至主内存 (main memory) 之中。

强制刷新写缓存, 将使得当前线程写入 volatile 字段的值 (以及写缓存中已有的其他内存修改), 同步至主内存之中。

由于内存写操作同时会无效化其他处理器所持有的、指向同一内存地址的缓存行, 因此可以认为其他处理器能够立即见到该 volatile 字段的最新值。

锁, volatile 字段

问题汇总

参考资料

1. [Java内存模型](#)
2. [Java 语言规范第 17.4 小节](#)

交流群

