实战Java高并发程序设计第3周

法律声明

【声明】本视频和幻灯片为炼数成金网络课程的教学资料，所有资料只能在课程内使用，不得在课程以外范围散播，违者将可能被追究法律和经济责任。

课程详情访问炼数成金培训网站

http://edu.dataguru.cn

炼数成金逆向收费式网络课程

- Dataguru（炼数成金）是专业数据分析网站，提供教育，媒体，内容，社区，出版，数据分析业务等服务。我们的课程采用新兴的互联网教育形式，独创地发展了逆向收费式网络培训课程模式。既继承传统教育重学习氛围，重竞争压力的特点，同时又发挥互联网的威力打破时空限制，把天南地北志同道合的朋友组织在一起交流学习，使到原先孤立的学习个体组合成有组织的探索力量。并且把原先动辄成千上万的学习成本，直线下降至百元范围，造福大众。我们的目标是：低成本传播高价值知识，构架中国第一的网上知识流转阵地。
- 关于逆向收费式网络的详情，请看我们的培训网站 http://edu.dataguru.cn

内容提要

- 原子性
- 有序性
- 可见性
- Happen-Before
- 线程安全的概念

原子性

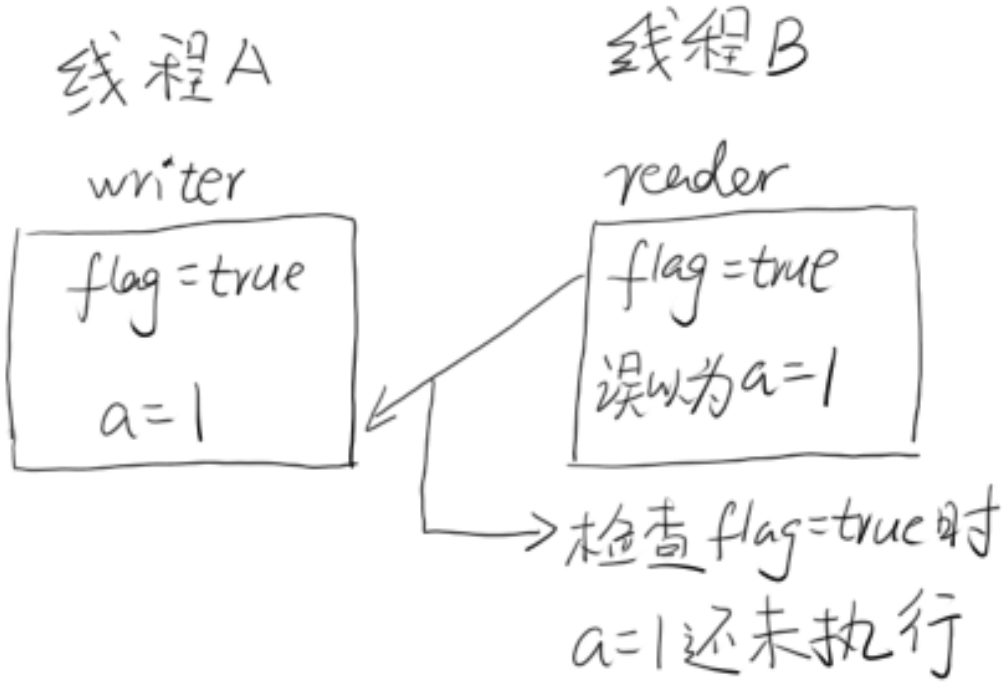
- 原子性是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其它线程干扰。

i++是原子操作吗？

有序性

- 在并发时，程序的执行可能就会出现乱序

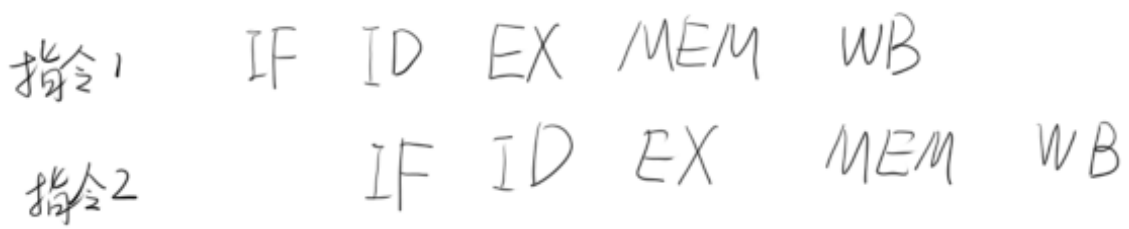
```
class OrderExample {
int a = 0;
boolean flag = false;
public void writer() {
    a = 1;
    flag = true;
}
public void reader() {
    if (flag) {
        int i =  a +1;
        ……
    }
}
}
```

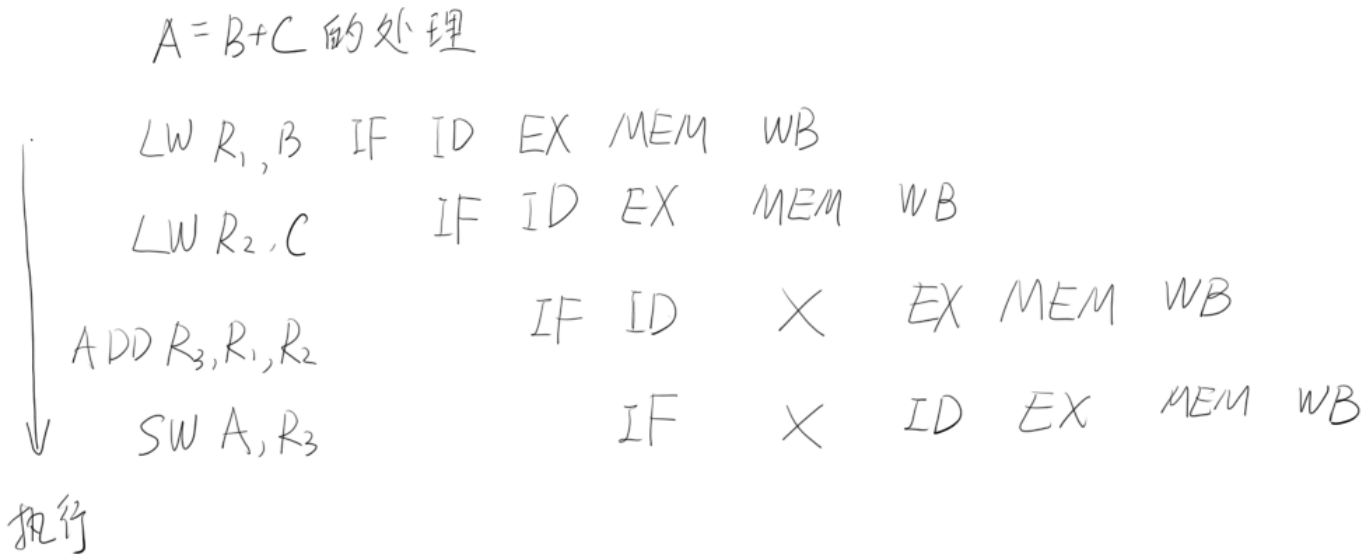


有序性

- 一条指令的执行是可以分为很多步骤的
  - 取指 IF
  - 译码和取寄存器操作数 ID
  - 执行或者有效地址计算 EX
  - 存储器访问 MEM
  - 写回 WB

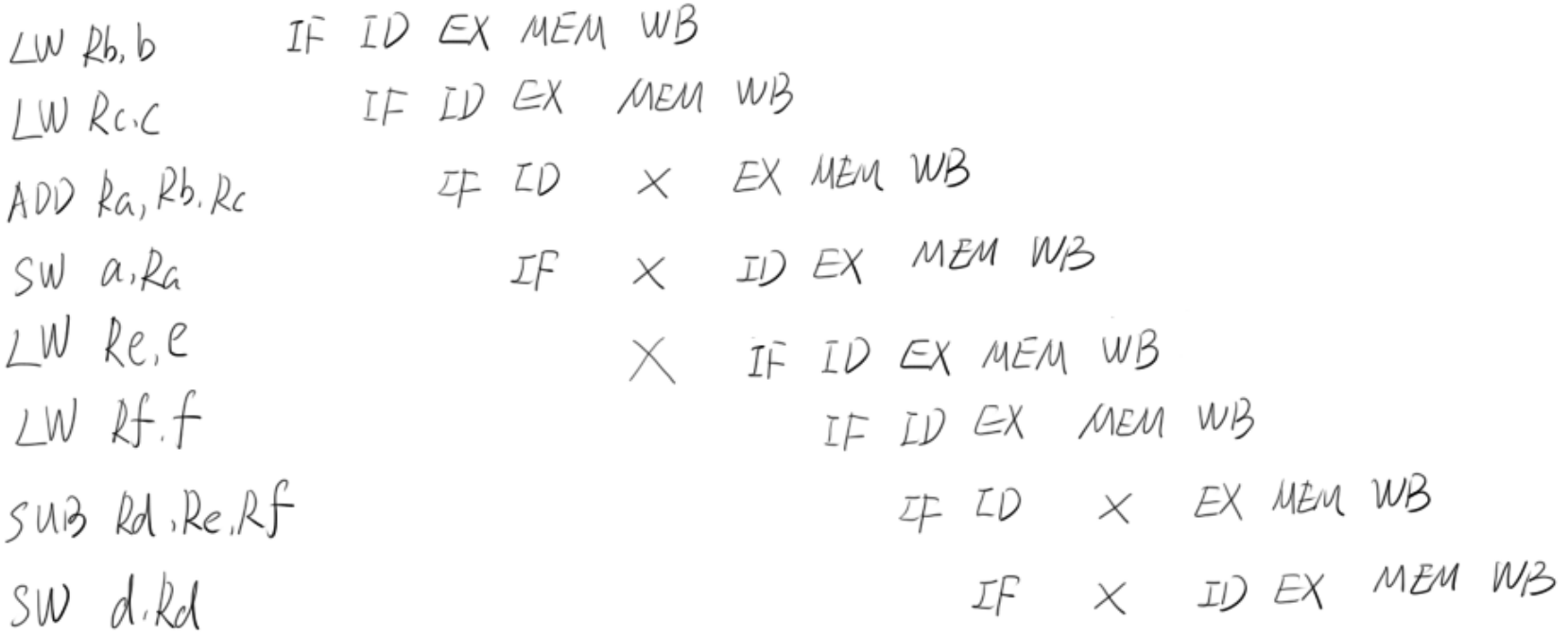
有序性


指令1 IF ID EX MEM WB
指令2 IF ID EX MEM WB


A=B+C的处理
LW R1,B IF ID EX MEM WB
LW R2,C IF ID EX MEM WB
ADD R3,R1,R2 IF ID X EX MEM WB
SW A,R3 IF X ID EX MEM WB
执行

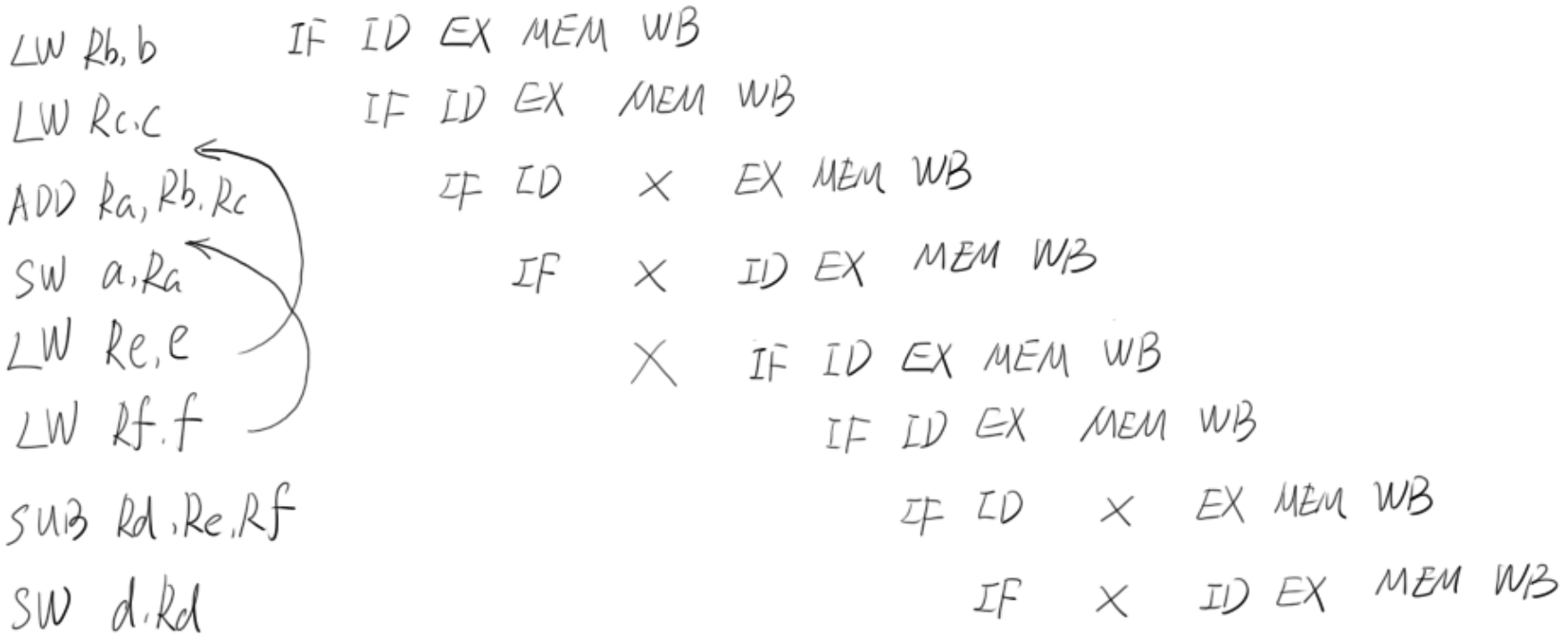
有序性

a=b+c
d=e-f


LW Rb,b
LW Rc,c
ADD Ra,Rb,Rc
SW a,Ra
LW Re,e
LW Rf,f
SUB Rd,Re,Rf
SW d,Rd
IF ID EX MEM WB

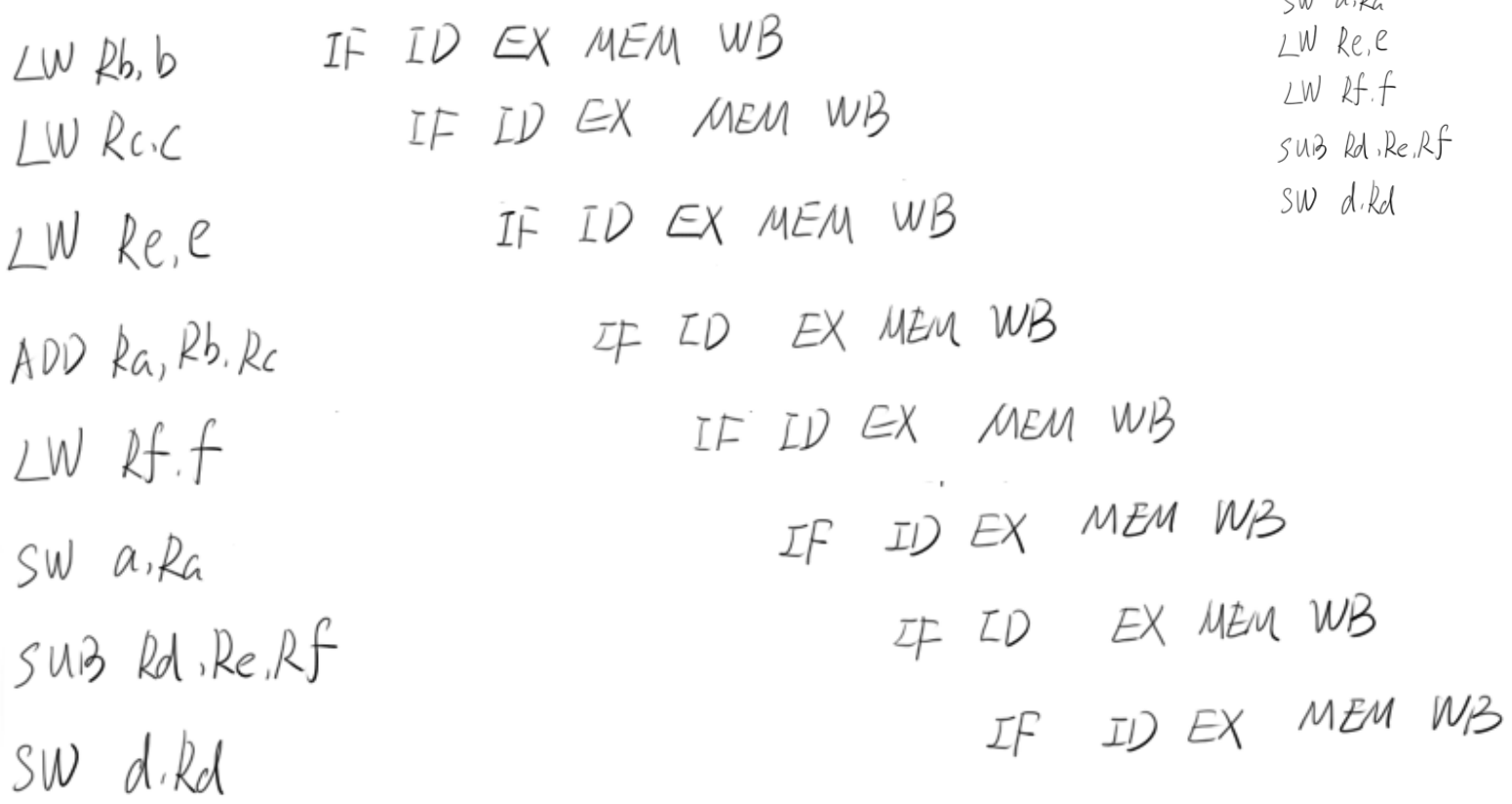
有序性

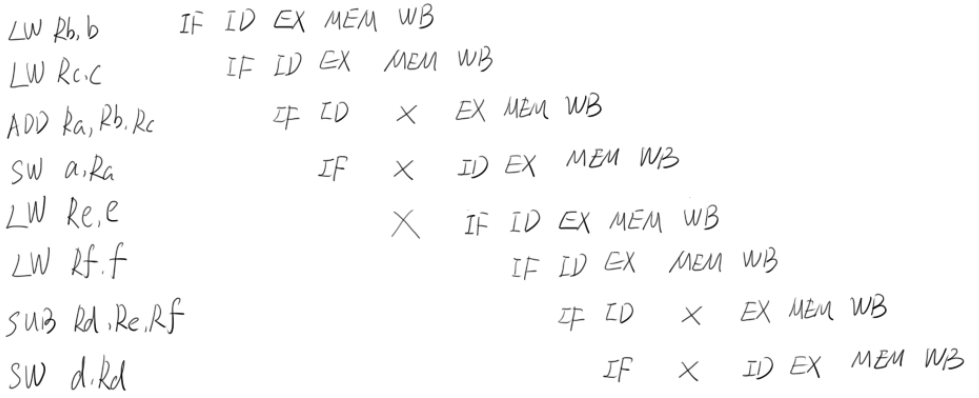
a=b+c
d=e-f


LW Rb, b
LW Rc, c
ADD Ra, Rb, Rc
SW a, Ra
LW Re, e
LW Rf, f
SUB Rd, Re, Rf
SW d, Rd
IF ID EX MEM WB
IF ID EX MEM WB
IF ID × EX MEM WB
IF × ID EX MEM WB
× IF ID EX MEM WB
IF ID EX MEM WB
IF ID × EX MEM WB
IF × ID EX MEM WB

有序性

a=b+c
d=e-f

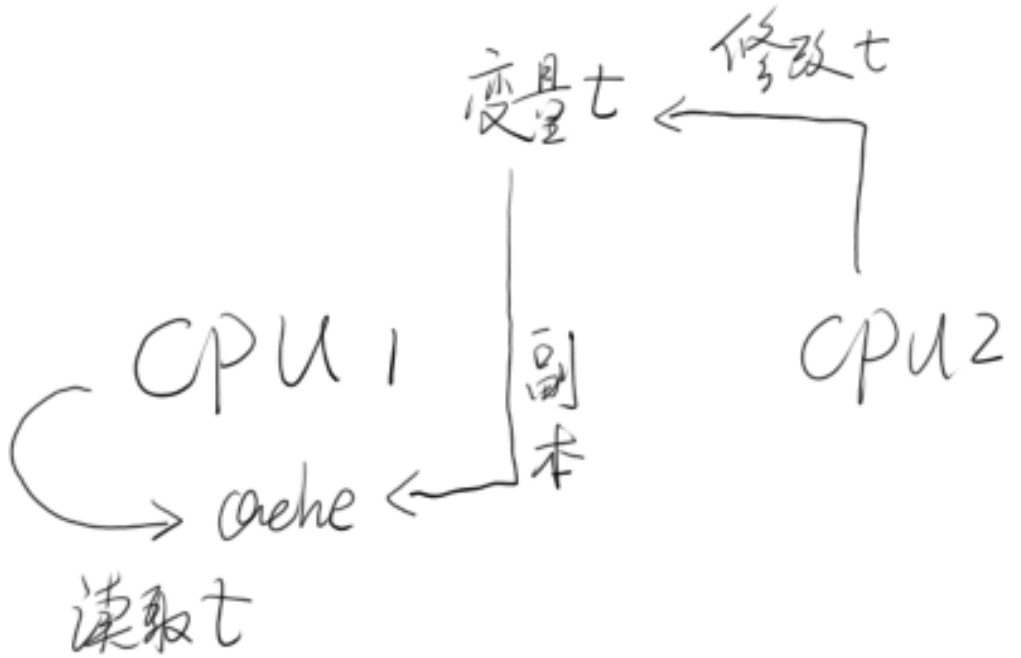




指令重排可以使流水线更加顺畅

可见性

- 可见性是指当一个线程修改了某一个共享变量的值，其他线程是否能够立即知道这个修改。
  - 编译器优化
  - 硬件优化（如写吸收，批操作）



可见性

- Java虚拟机层面的可见性
  - http://hushi55.github.io/2015/01/05/volatile-assembly

```
public class VisibilityTest extends Thread {
    private boolean stop;
    public void run() {
        int i = 0;
        while(!stop) {
            i++;
        }
        System.out.println("finish loop,i=" + i);
    }

    public void stopIt() {
        stop = true;
    }

    public boolean getStop(){
        return stop;
    }
```

```
    public static void main(String[] args) throws Exception {
        VisibilityTest v = new VisibilityTest();
        v.start();

        Thread.sleep(1000);
        v.stopIt();
        Thread.sleep(2000);
        System.out.println("finish main");
        System.out.println(v.getStop());
    }
```

可见性

- -server模式运行上述代码，永远不会停止

```
0x0193bf8c: mov     dword ptr [esp+0ffffc000h],eax
0x0193bf93: push    ebp
0x0193bf94: sub     esp,8h           ;*synchronization entry
                                     ; - edu.hushi.jvm.VisibilityTest::run@-1 (line 13)
0x0193bf97: mov     ebp,ecx
0x0193bf99: movzx   eax,byte ptr [ecx+64h]  ;*getfield stop
                                     ; - edu.hushi.jvm.VisibilityTest::run@9 (line 14)
0x0193bf9d: test    eax,eax
0x0193bf9f: jne     193bfafh         ;*ifeq
                                     ; - edu.hushi.jvm.VisibilityTest::run@12 (line 14)
0x0193bfa1: mov     ebx,1h           ; OopMap{ebp=Oop off=38}
                                     ;*ifeq
                                     ; - edu.hushi.jvm.VisibilityTest::run@12 (line 14)
0x0193bfa6: test    dword ptr [0a0000h],edi  ;*ifeq  这里开始是循环
                                     ; - edu.hushi.jvm.VisibilityTest::run@12 (line 14)
                                     ;   {poll}
0x0193bfac: inc     ebx              ;*iinc
                                     ; - edu.hushi.jvm.VisibilityTest::run@5 (line 15)
0x0193bfad: jmp     193bfa6h
0x0193bfaf: mov     ecx,14h
0x0193bfb4: nop
0x0193bfb7: call    191dd00h         ; OopMap{ebp=Oop off=60}
                                     ;*getstatic out
                                     ; - edu.hushi.jvm.VisibilityTest::run@15 (line 17)
                                     ;   {runtime_call}
```

```
public void run() {
    int i = 0;
    while(!stop) {
        i++;
    }
    System.out.println("finish loop,i=" + i);
}
```

可见性

Example 17.4-1. Incorrectly Synchronized Programs May Exhibit Surprising Behavior

The semantics of the Java programming language allow compilers and microprocessors to perform optimizations that can interact with incorrectly synchronized code in ways that can produce behaviors that seem paradoxical. Here are some examples of how incorrectly synchronized programs may exhibit surprising behaviors.

Consider, for example, the example program traces shown in Table 17.1. This program uses local variables `r1` and `r2` and shared variables `A` and `B`. Initially, `A == B == 0`.

Table 17.1. Surprising results caused by statement reordering - original code

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

It may appear that the result `r2 == 2` and `r1 == 1` is impossible. Intuitively, either instruction 1 or instruction 3 should come first in an execution. If instruction 1 comes first, it should not be able to see the write at instruction 4. If instruction 3 comes first, it should not be able to see the write at instruction 2.

If some execution exhibited this behavior, then we would know that instruction 4 came before instruction 1, which came before instruction 2, which came before instruction 3, which came before instruction 4. This is, on the face of it, absurd.

However, compilers are allowed to reorder the instructions in either thread, when this does not affect the execution of that thread in isolation. If instruction 1 is reordered with instruction 2, as shown in the trace in Table 17.2, then it is easy to see how the result `r2 == 2` and `r1 == 1` might occur.

Table 17.2. Surprising results caused by statement reordering - valid compiler transformation

Thread 1	Thread 2
B = 1;	r1 = B;
r2 = A;	A = 2;

可见性

Another example of surprising results can be seen in Table 17.3. Initially, `p == q` and `p.x == 0`. This program is also incorrectly synchronized; it writes to shared memory without enforcing any ordering between those writes.

Table 17.3. Surprising results caused by forward substitution

Thread 1	Thread 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3;`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r1.x;`	

One common compiler optimization involves having the value read for `r2` reused for `r5`: they are both reads of `r1.x` with no intervening write. This situation is shown in Table 17.4.

Table 17.4. Surprising results caused by forward substitution

Thread 1	Thread 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3;`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r2;`	

Now consider the case where the assignment to `r6.x` in Thread 2 happens between the first read of `r1.x` and the read of `r3.x` in Thread 1. If the compiler decides to reuse the value of `r2` for the `r5`, then `r2` and `r5` will have the value `0`, and `r4` will have the value `3`. From the perspective of the programmer, the value stored at `p.x` has changed from `0` to `3` and then changed back.

Happen-Before规则

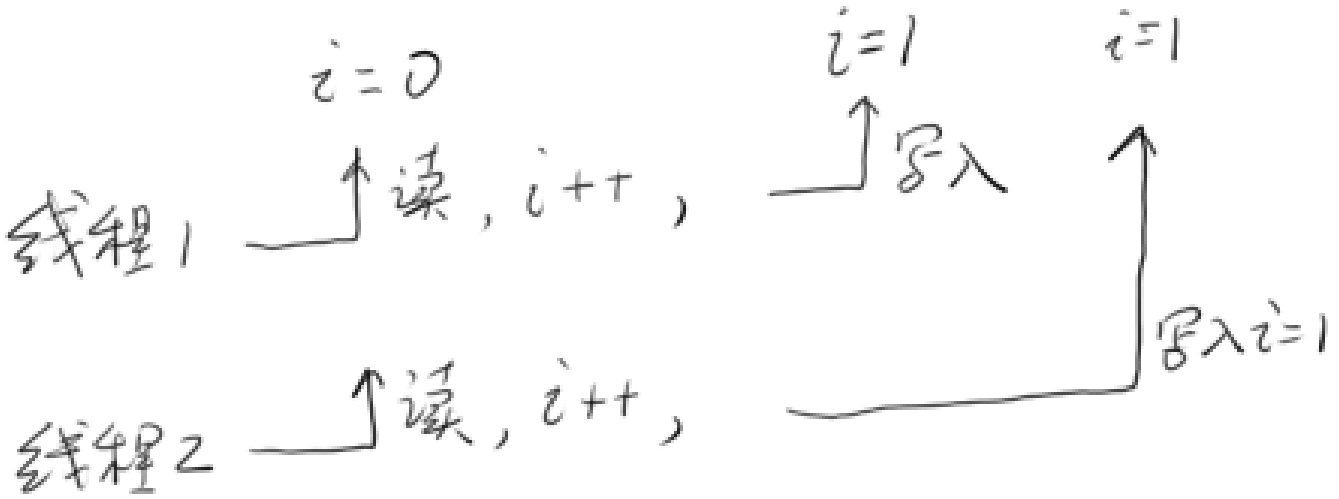
```
a=1;
b=a+1;
```

- 程序顺序原则：一个线程内保证语义的串行性
- volatile规则：volatile变量的写，先发生于读，这保证了volatile变量的可见性
- 锁规则：解锁（unlock）必然发生在随后的加锁（lock）前
- 传递性：A先于B，B先于C，那么A必然先于C
- 线程的start()方法先于它的每一个动作
- 线程的所有操作先于线程的终结（Thread.join()）
- 线程的中断（interrupt()）先于被中断线程的代码
- 对象的构造函数执行结束先于finalize()方法

线程安全的概念

- 指某个函数、函数库在多线程环境中被调用时，能够正确地处理各个线程的局部变量，使程序功能正确完成。

i++在多线程下访问的情况



线程安全的概念

```
public class AccountingSync implements Runnable{
        static AccountingSync instance=new AccountingSync();
        static int i=0;
        @Override
        public void run() {
                for(int j=0;j<10000000;j++){
                        synchronized(instance){
                                i++;
                        }
                }
        }
```

Thanks

FAQ时间