作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

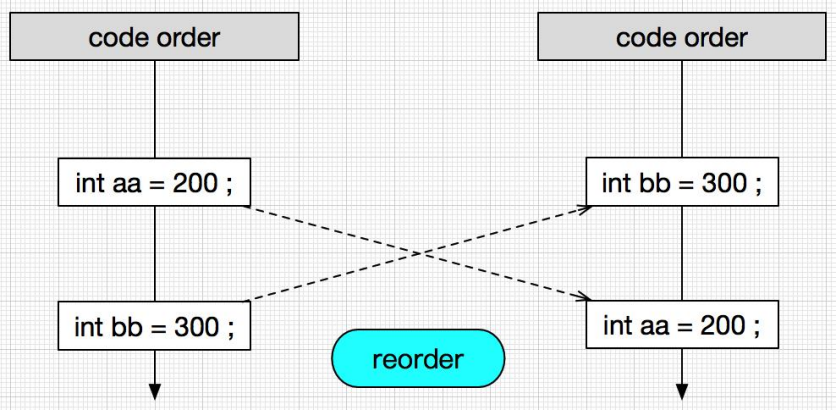
指令乱序的分类

程序的生命周期包括编写代码，编译代码，运行代码。编写代码的时候，前后的代码有顺序，编译代码可能改变代码顺序，运行代码也可能改变代码顺序。
改变代码顺序被称为指令乱序，作用为优化代码、提升性能。

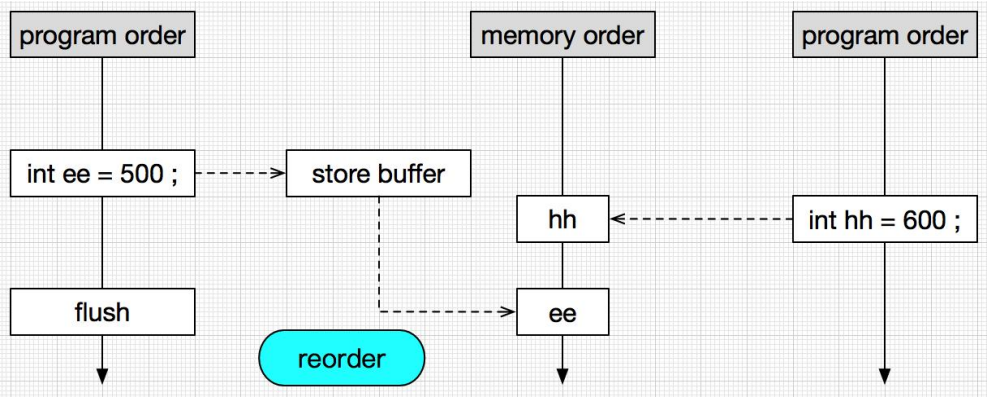
指令乱序，包括编译期指令乱序、运行期指令乱序。
编译期指令乱序：编译器使用优化策略，使得编译后的指令顺序与源码的代码顺序不一致。通过查看编译后的 ELF 文件，就可以分析这种乱序。
运行期指令乱序：CPU 使用流水线、超线程等并发机制改变指令的运行顺序，CPU 使用写缓冲区改变内存可见性。这种乱序发生在 CPU 运行期，分析难度大。

从影响的结果分类，指令乱序包含 2 部分，指令乱序、数据乱序。

指令乱序的示意图：
初始状态的顺序，`int aa = 200 ;` 在前面，`int bb = 300 ;` 在后面。
编译期，如果 `int bb = 300 ;` 的指令在前面，则发生编译期指令乱序。
运行期，如果 `int bb = 300 ;` 的指令先执行，则发生运行期指令乱序。



数据乱序的示意图：
查看纵向时间轴，`int ee = 500 ;` 先执行，`int hh = 600 ;` 后执行，预期 ee 先写到内存。
内存视角分析，hh 先写到内存，ee 后写到内存，发生数据乱序。
因为 ee 先被写到 store buffer，后续刷新到内存，时间有延迟。
store buffer 是写缓冲区，保存写操作的值，异步刷新，提供写性能。



指令乱序的问题
编译器和 CPU 根据代码上下文实现指令乱序，比如一个方法模块的代码。编译器和 CPU 无法感知并发上下文，导致指令乱序可能引起并发场景错误。

用 C 和汇编直观理解指令乱序

假设有如下源码。给变量写值的代码顺序为 aa、bb、cc。

```
int aa = 1;
int bb = 2;
int cc = 3;

int main()
{
    aa = 100;
    bb = 200;
    cc = 300;

    return 0;
}
```

默认情况下，编译后的指令如下。mov 指令给变量写值的顺序为 aa、bb、cc。此时，编译后的指令顺序与源码的代码顺序一样。

```
    movl    $100, aa(%rip)
    movl    $200, bb(%rip)
    movl    $300, cc(%rip)
```

假设，编译器使用某种优化策略，编译后的指令如下。mov 指令给变量写值的顺序为 aa、cc、bb。此时，编译后的指令顺序与源码的代码顺序不一样，发生编译期指令乱序。

```
    movl    $100, aa(%rip)
    movl    $300, cc(%rip)
    movl    $200, bb(%rip)
```

假设，CPU 运行指令的顺序如下。3 个指令的运行顺序为 bb、aa、cc。此时，指令的运行顺序与编译后的指令顺序不一样，发生运行期指令乱序。

```
    movl    $200, bb(%rip)
```

```
    movl    $100, aa(%rip)
    movl    $300, cc(%rip)
```

用 C 和汇编分析 load 和 store

内存的基本操作包括读内存和写内存。读内存表示为 load，写内存表示为 store。load 和 store，可以组合出多种顺序。
loadload 表示第 1 个指令是读，第 2 个指令是读。
loadstore 表示第 1 个指令是读，第 2 个指令是写。
storestore 表示第 1 个指令是写，第 2 个指令是写。
storeload 表示第 1 个指令是写，第 2 个指令是读。

编写代码： loadstore.c

```
int ee = 100;
int ff = 200;

// loadload
int func_load_load()
{
    int tmp1 = ee;
    int tmp2 = ff;
    return tmp1 + tmp2;
}

// loadstore
int func_load_store()
{
    int tmp1 = ee;
    ff = 600;
    return tmp1;
}

// storestore
void func_store_store()
{
    ee = 500;
    ff = 600;
}

// storeload
int func_store_load()
{
    ee = 500;
    int tmp2 = ff;
    return ee + tmp2;
}
```

编译代码：

```
gcc loadstore.c -S -o loadstore.s
gcc loadstore.c -S -O2 -o loadstore.O2.s
```

分析结果：
编译时，使用参数-O2 优化策略，生成 loadstore.O2.s。这里展示关键代码。

源码	汇编代码	结果和分析

// loadload
int func_load_load()
{
int tmp1 = ee;
int tmp2 = ff;
return tmp1 + tmp2;
}
	func_load_load:
	movl ee(%rip), %eax
	addl ff(%rip), %eax
	ret
		汇编代码，首先 load(ee)，然后 load(ff)。
		没有发生编译期指令乱序。

// loadstore
int func_load_store()
{
int tmp1 = ee;
ff = 600;
return tmp1;
}
	func_load_store:
	movl $600, ff(%rip)
	movl ee(%rip), %eax
	ret
		汇编代码，首先 store(ff)，然后 load(ee)。
		与源码的代码顺序不一致。
		发生编译期指令乱序。

// storestore
void func_store_store()
{
ee = 500;
ff = 600;
}
	func_store_store:
	movl $500, ee(%rip)
	movl $600, ff(%rip)
	ret
		汇编代码，首先 store(ee)，然后 store(ff)。
		没有发生编译期指令乱序。

// storeload
int func_store_load()
{
ee = 500;
int tmp2 = ff;
return ee + tmp2;
}
	func_store_load:
	movl ff(%rip), %eax
	movl $500, ee(%rip)
	addl $500, %eax
	ret
		汇编代码，首先 load(ff)，然后 store(ee)。
		与源码的代码顺序不一致。
		发生编译期指令乱序。

用 C 和汇编分析避免指令乱序的方法

有多种方法避免指令乱序：
2 个指令前后依赖。为了保证代码逻辑正确，避免指令乱序。
2 个指令之间调用函数。函数可能修改指令使用的内存，为了保证数据正确，避免指令乱序。
使用内存屏障相关的关键字、标识符。比如，C 语言的 volatile 关键字、内联汇编的 memory 标识符，禁止编译期指令乱序。
使用内存屏障相关的指令。mfence 指令、lock 指令等，禁止运行期指令乱序。

编写代码： fix_loadstore.c

```
int ee = 100;
int ff = 200;

// loadstore
int func_load_store()
{
    int tmp1 = ee;
    ff = 600;
    return tmp1;
}

// 2 个指令前后依赖
int func_load_store_depend()
{
    int tmp1 = ee;
    ff = 600 + tmp1; // 指令前后依赖。
    return tmp1;
}

extern void func_empty();

// 2 个指令之间调用函数
int func_load_store_func()
{
    int tmp1 = ee;
    func_empty(); // 调用一个方法。
    ff = 600;
    return tmp1;
}

// 使用内存屏障相关的关键字、标识符
int func_load_store_barrier()
{
    int tmp1 = ee;
    asm("" ::: "memory"); // 内存屏障相关
    ff = 600;
    return tmp1;
}

// 使用内存屏障相关的指令
int func_load_store_barrier2()
{
    int tmp1 = ee;
    asm("mfence" ::: "memory"); // 内存屏障相关
    ff = 600;
    return tmp1;
}
```

编译代码：

```
gcc fix_loadstore.c -S -O2 -o fix_loadstore.O2.s
```

分析结果：

函数 func_load_store()发生编译期指令乱序。后续的几个函数，使用多种方法避免指令乱序。

编译时，使用参数-O2 优化策略，生成 fix_loadstore.O2.s。这里展示核心代码。

源码	汇编代码	分析

// loadstore
int func_load_store()
{
    int tmp1 = ee;
    ff = 600;
    return tmp1;
}
	func_load_store:
	    movl $600, ff(%rip)
	    movl ee(%rip), %eax
	    ret
		发生编译期指令乱序。

// 2 个指令前后依赖
int func_load_store_depend()
{
    int tmp1 = ee;
    ff = 600 + tmp1; // 指令前后依赖。
    return tmp1;
}
	func_load_store_depend:
	    movl ee(%rip), %eax
	    leal 600(%rax), %edx
	    movl %edx, ff(%rip)
	    ret
		2 个指令前后依赖，没有编译期指令乱序。

extern void func_empty();
 	
// 2 个指令之间调用函数
int func_load_store_func()
{
    int tmp1 = ee;
    func_empty(); // 调用一个方法。
    ff = 600;
    return tmp1;
}
	func_load_store_func:
	    pushq %rbx
	    movl ee(%rip), %ebx
	    xorl %eax, %eax
	    call func_empty
	    movl %ebx, %eax
	    movl $600, ff(%rip)
	    popq %rbx
	    ret
		2 个指令之间调用了某个方法，没有编译期指令乱序。

// 使用内存屏障相关的关键字、标识符
int func_load_store_barrier()
{
    int tmp1 = ee;
    asm("" ::: "memory"); // 内存屏障相关
    ff = 600;
    return tmp1;
}
	func_load_store_barrier:
	    movl ee(%rip), %eax
	    movl $600, ff(%rip)
	    ret
		使用 memory 标识符，没有编译期指令乱序。

// 使用内存屏障相关的指令
int func_load_store_barrier2()
{
    int tmp1 = ee;
    asm("mfence" ::: "memory"); // 内存屏障相关
    ff = 600;
    return tmp1;
}
	func_load_store_barrier2:
	    movl ee(%rip), %eax
	    mfence
	    movl $600, ff(%rip)
	    ret
		使用 mfence 关键字、memory 标识符，没有编译期指令乱序。