JMM

1. 课程简介

1.1 JMM 研究的到底是什么？

JMM 即 Java 内存模型

- [ ] A. Java 内存结构，如栈、堆？
- [ ] B. JVM 调优？
- [ ] C. JVM 垃圾回收机制？
- [x] D. 多线程下 Java 代码的执行顺序，共享变量的读写？

1.2 学习目标

- 多线程下，读写共享变量会有哪些问题
- 解决这些问题的钥匙 - Java 内存模型
- 解决这些问题的手段 - 掌握同步方法
- 更多安全问题与解决方法

1.3 参考资料

- Java Language Specification Chapter 17. Threads and Locks
- JSR-133: JavaTM Memory Model and Thread Specification
- Doug Lea's JSR-133 cookbook
- Sutter’s Mill atomic Weapons: The C++ Memory Model and Modern Hardware
- Paul E. Mckenney's Is Parallel Programming Hard, And, If So, What Can You Do About It?

    - Appendix C - Why Memory Barriers?
- jcstress
- Aleksey Shipilëv's Java Memory Model Pragmatics (transcript)
- Java Concurrency in Practice
- The Art of Multiprocessor Programming

2. 多线程读写共享变量

2.1 Does your computer execute the program you wrote?

- [ ] Yes
- [x] No

2.2 澄清两个事实

- **你写的代码，未必是实际运行的代码**
- **代码的编写顺序，未必是实际执行顺序**

2.3 问题演示

以下问题演示，需要的条件：**共享变量 + 有读、至少一个写**

问题 - 永远的循环

```
boolean stop;

@Actor
public void a1() {
    while(!stop){
    }
}

@Signal
void a2() {
    stop = true;
}
```

问题 - 加加减减

```
int balance = 10;

@Actor
public void deposit() {
    balance += 5;
}

@Actor
public void withdraw() {
    balance -= 5;
}

@Arbiter
public void query(I_Result r) {
    r.r1 = balance;
}
```

问题 - 第四种可能

```
int a;
int b;

@Actor
public void actor1(II_Result r) {
    b = 1;
    r.r2 = a;
}

@Actor
public void actor2(II_Result r) {
    a = 2;
    r.r1 = b;
}
```

2.4 问题揭秘

永远的循环 - 揭秘

```
boolean stop;

@Actor
public void a1() {
    while(true){
        boolean b = stop;
        if(b) break;
```

```
    }
}

@Signal
void a2() {
    stop = true;
}
```

改写一下代码方便测试

```
static boolean stop = false;
public static void main(String[] args) {
    new Thread(() -> {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        stop = true;
        System.out.println(">>>>>>>>>> stop");
    }).start();

    foo();
}
static void foo() {
    while (true) {
        boolean b = stop;
        if(b) {
            break;
        }
    }
}
```

先用 `-XX:+PrintCompilation` 来查看即时编译情况

> % 的含义
>
> On-Stack-Replacement (OSR)

再尝试用 `-Xint` 强制解释执行

加加减减 - 揭秘

同样改写一下测试代码

```
static int balance = 10;

public static void withdraw() {
    balance += 5;
}

public static void deposit() {
    balance -= 5;
}

public static void main(String[] args) {

    List<Thread> threads = Arrays.asList(
        new Thread(TestAddSub::deposit),
        new Thread(TestAddSub::withdraw)
    );
    threads.forEach(Thread::start);
    for (Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println(balance);
}
```

这回用一下 ASM 工具，可以看到源码第 8 行的 `balance += 5` 的字节码如下

```
LINENUMBER 8 L0
    GETSTATIC TestAddSub.balance : I
    ICONST_5
    IADD
    PUTSTATIC TestAddSub.balance : I
```

而第 12 行的 `balance -= 5` 字节码如下

```
LINENUMBER 12 L0
    GETSTATIC TestAddSub.balance : I
    ICONST_5
    ISUB
    PUTSTATIC TestAddSub.balance : I
```

方法替换为伪码后

```
static int balance = 10;

public static void withdraw() {
    int b = balance;
    b += 5;
    balance = b;
}

public static void deposit() {
    int c = balance;
    c -= 5;
    balance = c;
}
```

可能的执行序列如下

case 1

```
int b = balance; // 线程1
b += 5;          // 线程1
balance = b;     // 线程1
int c = balance; // 线程2
c -= 5;          // 线程2
balance = c;     // 线程2
```

case 2

```
int c = balance; // 线程2
int b = balance; // 线程1
b += 5;          // 线程1
balance = b;     // 线程1
c -= 5;          // 线程2
balance = c;     // 线程2
```

case 3

```
int b = balance; // 线程1
int c = balance; // 线程2
c -= 5;          // 线程2
balance = c;     // 线程2
b += 5;          // 线程1
balance = b;     // 线程1
```

第四种可能 - 揭秘

```
int a;
int b;
```

```
@Actor
public void actor1(II_Result r) {
    b = 1;
    r.r2 = a;
}

@Actor
public void actor2(II_Result r) {
    a = 2;
    r.r1 = b;
}
```

可能的执行序列如下

case 1

```
b = 1;      // 线程1
r.r2 = a;   // 线程1

a = 2;      // 线程2
r.r1 = b;   // 线程2

// 结果 r1==1, r2==0
```

case 2

```
a = 2;      // 线程2
r.r1 = b;   // 线程2

b = 1;      // 线程1
r.r2 = a;   // 线程1

// 结果 r1==0, r2==2
```

case 3

```
a = 2;      // 线程2

b = 1;      // 线程1
r.r2 = a;   // 线程1

r.r1 = b;   // 线程2

// 结果 r1==1, r2==2
```

那么 case 4 的实际执行序列是？

```
r.r2 = a;   // 线程1

a = 2;      // 线程2
r.r1 = b;   // 线程2

b = 1;      // 线程1

// 结果 r1==0, r2==0
```

可能是编译器调整了指令执行顺序

💎 压测方能暴露问题

2.5 思考为什么

1. 如果让一个线程总是占用 CPU 是不合理的，任务调度器会让线程分时使用 CPU
2. 编译器以及硬件层面都会做层层优化，提升性能
    - Compiler/JIT 优化
    - Processor 流水线优化
    - Cache 优化

2.6 编译器优化

例 1

```
优化前
    x=1
    y="universe"
    x=2
优化后
    y="universe"
    x=2
```

例 2

```
优化前
    for(i=0;i<max;i++){
        z += a[i]
    }
优化后
    t = z
    for(i=0;i<max;i++){
        t += a[i]
    }
    z = t
```

例 3

```
优化前
    if(x>=0){
        y = 1;
        // ...
    }
优化后
    y = 1;
    if(x>=0){
        // ...
    }
```

2.7 Processor 优化

- 流水线在 CPU 的一个时钟周期内会执行多个指令的不同部分

非流水线操作

假设有三条指令

```
---|---|---|
1   2   3
```

每条指令执行花费 300ps 时间，最后将结果存入寄存器需要 20ps

一秒能运行的指令数为

$$1/(320 * 10^{-12}) = 3,125,000,000$$

流水线操作

仔细分析就会发现，可以把每个指令细分为三个阶段

```
A|B|C|          // 1
  A|B|C|        // 2
    A|B|C|      // 3
```

增加一些寄存器，缓存每一阶段的结果，这样就可以在执行 `指令1-C` 阶段时，同时执行 `指令2-B` 以及 `指令3-A`

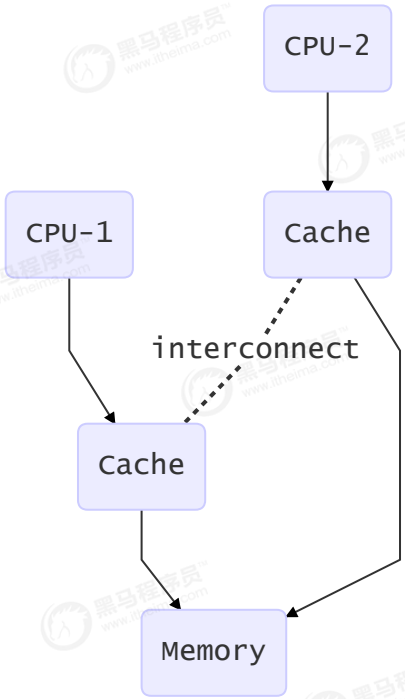
一秒能运行的指令数为

$$1/(120 * 10^{-12}) = 8,333,333,333$$

execute Out of Order

- 在按序执行中，一旦遇到指令依赖的情况，流水线就会停滞
- 如果采用乱序执行，就可以跳到下一个非依赖指令并发布它。这样，执行单元就可以总是处于工作状态，把时间浪费减到最少

2.8 缓存优化



> MESI 协议 引入缓存的副作用在于同一份数据可能保存了副本，一致性该如何保证呢？
>
> - Modified - 要向其它 CPU 发送 cache line 无效消息，并等待 ack
> - Exclusive - 独占、即将要执行修改
> - Shared - 共享、一般读取时的初始状态
> - Invalid - 一旦发现数据无效，需要重新加载数据

例子

a == b == 0

线程 1

```
b = 1;      // 线程1
r.r2 = a;   // 线程1
```

线程 2

```
a = 2;      // 线程2
r.r1 = b;   // 线程2
```

问 r1 和 r2 有没有可能同时为 0？

```
r.r1 = b;   // 线程2 与 a = 2 重排
r.r2 = a;   // 线程1 与 a = 1 重排
b = 1;      // 线程1
a = 2;      // 线程2
```

下面从缓存的角度分析，注意假定指令没有重排

```
b = 1;      // 线程1 - 写入 CPU-0 的 store buffer

a = 2;      // 线程2 - 写入 CPU-1 的 store buffer
r.r1 = b;   // 线程2 - 马上执行

r.r2 = a;   // 线程1 - 马上执行
            // 线程1 - 将 store buffer 中的 b = 1 写入 cache，晚了
            // 线程2 - 将 store buffer 中的 a = 2 写入 cache，晚了
```

？证明可能是缓存引起的

2.9 我们面对的问题

对于程序员而言，我们不应当关注究竟是

- 编译器优化
- Processor 优化
- 缓存优化

否则，就好像打开了潘多拉魔盒

3. JMM 内存模型

3.1 什么是 JMM

> *A memory model describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program. A high level, informal overview of the memory model shows it to be a set of rules for when writes by one thread are visible to another thread.*

> 多线程下，共享变量的**读写顺序**是头等大事，内存模型就是多线程下对共享变量的一组**读写规则**
>
> - 共享变量值是否在线程间同步
> - 代码可能的执行顺序

- 需要关注的操作就有两种 Load、Store
  - Load 就是从缓存读取到寄存器中，如果一级缓存中没有，就会层层读取二级、三级缓存，最后才是 Memory
  - Store 就是从寄存器运算结果写入缓存，不会直接写入 Memory，当 Cache line 将被 eject 时，会 writeback 到 Memory

3.2 JMM 规范

规则 1 - Race Condition

在**多线程**下，**没有依赖关系**的代码，在执行**共享变量读写操作**（至少有一个线程写）时，并不能保证以编写顺序（Program Order）执行，这称为发生了**竞态条件**（Race Condition）

例如

有共享变量 x，线程 1 执行

```
r.r1 = y;
r.r2 = x;
```

线程 2 执行

```
x = 1;
y = 1;
```

最终的结果可能是 `r1==1` 而 `r2==0`

竞争是为了更好的性能 -Data Race Free

规则 2 - Synchronization Order

若要保证**多线程**下，**每个线程**的**执行顺序**（Synchronization Order）按编写顺序（Program Order）执行，那么必须**使用 Synchronization Actions 来保证**，这些 SA 有

- lock，unlock
- volatile 方式读写变量
- VarHandle 方式读写变量

> *Synchronization Order 也称之为 Total Order*

例如

用 volatile 修饰共享变量 y，线程 1 执行

```
r.r1 = y;
r.r2 = x;
```

线程 2 执行

```
x = 1;
y = 1;
```

最终的结果就***不可能***是 `r1==1` 而 `r2==0`

□ SO 并不是阻止多线程切换

错误的认识，线程 1 执行

```
synchronized(LOCK) {
    r1 = x; //1 处
    r2 = x; //2 处
}
```

线程 2 执行

```
synchronized(LOCK) {
    x = 1
}
```

并不是说 //1 与 //2 处之间不能切换到线程 2，只是即使切换到了线程 2，因为线程 2 不能拿到 LOCK 锁导致被阻塞，执行权又会轮到线程 1

思考：如果线程 2 执行的代码不使用同一个 LOCK 对象呢？

□ volatile 只用了一半算 SO 吗?

用例 1

```
int x;
volatile int y;
```

之后采用

```
x = 10; //1 处
y = 20; //2 处
```

此时 //1 处代码***绝不会***重排到 //2 处之后（只写了 volatile 变量）

用例 2

```
int x;
volatile int y;
```

执行下面的测试用例

```
@Actor
public void a1(II_Result r) {
    y = 1;      //1 处
    r.r2 = x;   //2 处
}

@Actor
public void a2(II_Result r) {
    x = 1;      //3 处
    r.r1 = y;   //4 处
}
```

//1 //2 处的顺序可以保证（只写了 volatile 变量），但 //3 //4 处的顺序却不能保证（只读了 volatile 变量），***仍会***出现 `r1==r2==0` 的问题

有时会很迷惑人，例如下面的例子

用例 3

```
@Actor
public void a1(II_Result r) {
    r.r2 = x;   //1 处
    y = 1;      //2 处
}

@Actor
public void a2(II_Result r) {
    r.r1 = y;   //3 处
    x = 1;      //4 处
}
```

这回 //1 //2（只写了 volatile 变量）//3 //4 处（只读了 volatile 变量）的顺序均能保证了，***绝不会***出现 `r1==r2==1` 的情况

此外将用例 2 中两个变量均用 volatile 修饰就不会出现 `r1==r2==0` 的问题，因此也把*全部都用 volatile 修饰称为 total order，部分变量用 volatile 修饰称为 partial order*

> *并不是说 partial order 不能用，只是，正确使用需要学明白后面的原理*

规则 3 - Happens-Before

若是变量读写时**发生线程切换**（例如，线程 1 写入 x，切换至线程 2，线程 2 读取 x）在这些边界的处理上**如果有 action1 先于 action 2 发生**，那么代码**可以按确定的顺序**执行，这称之为 **Happens-Before Order** 规则

> *Happens-Before Order 也称之为 Partial Order*

用公式表达为

$$action1 \overset{hb}{\longrightarrow} action2$$

含义为：如果 action1 先于 action2 发生，那么 action1 之前的共享变量的修改对于 action2 **可见**，且代码按 PO **顺序**执行

具体规则

其中 T_{n} 代表线程，而 x 未加说明，是普通共享变量，使用 volatile 会单独说明

1) 线程的启动和运行边界

$$\begin{aligned} T_1(x=10) &\overset{hb}{\longrightarrow} T_1(t2.start()) \\ &\overset{hb}{\longrightarrow} T_2(run()) \\ &\overset{hb}{\longrightarrow} T_2(x == 10) \end{aligned}$$

2) 线程的结束和 join 边界

$$\begin{aligned} T_1(x=10) &\overset{hb}{\longrightarrow} T_1(terminated) \\ &\overset{hb}{\longrightarrow} T_2(t1.join()) \\ &\overset{hb}{\longrightarrow} T_2(x == 10) \end{aligned}$$

3) 线程的打断和得知打断边界

$$\begin{aligned} T_1(x=10) &\overset{hb}{\longrightarrow} T_1(t2.interrupt()) \\ &\overset{hb}{\longrightarrow} T_2(this.isInterrupted()) \\ &\overset{hb}{\longrightarrow} T_2(x == 10) \end{aligned}$$

4) unlock 与 lock 边界

$$
\begin{aligned}
T_1(x=10) &\xrightarrow{hb} T_1(unlock(obj)) \\
&\xrightarrow{hb} T_2(lock(obj)) \\
&\xrightarrow{hb} T_2(x==10)
\end{aligned}
$$

5) volatile write 与 volatile read 边界

$$
T_1(x^{volatile}=10) \xrightarrow{hb} T_2(x^{volatile}==10)
$$

6) 传递性

$$
\begin{aligned}
T_{t1}(x=10) &\xrightarrow{hb} T_1(unlock(obj)) \\
&\xrightarrow{hb} T_2(lock(obj)) \\
&\xrightarrow{hb} T_2(x==10) \\
&\xrightarrow{hb} T_2(terminated) \\
&\xrightarrow{hb} T_3(t2.join()) \\
&\xrightarrow{hb} T_3(x==10)
\end{aligned}
$$

规则 4 - Causality

Causality 即因果律：代码之间如**存在依赖关系**，**即使没有加 SA** 操作，代码的执行顺序也是可以预见的

回顾一下

> ***多线程***下，没有***依赖关系***的代码，在***共享变量读写操作***（至少有一个线程写）时，并不能保证以编写顺序（Program Order）执行，这称为发生了***竞态条件***（Race Condition）

如果有一定的依赖关系呢？

比如

```
@JCStressTest
@Outcome(id = {"0, 0"}, expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(expect = Expect.FORBIDDEN, desc = "FORBIDDEN")
@State
public static class Case5 {
    int x;
    int y;

    @Actor
    public void a1(II_Result r) {
        r.r1 = x;
        y = r.r1;
    }

    @Actor
    public void a2(II_Result r) {
```

```
            r.r2 = y;
            x = r.r2;
        }
    }
```

x 的值来自于 y，y 的值来自于 x，而二者的初始值都是 0，因此没有可能有其他结果

规则 5 - 安全发布

若要安全构造对象，并将其共享使用，**需要用 final 或 volatile 修饰其成员变量**，并**避免 this 溢出**情况

> *静态成员变量可以安全地发布*

例如

```
class Holder {
    int x1;
    volatile int x2;

    public Holder(int v) {
        x1 = v;
        x2 = v;
    }
}
```

需要将它作为全局使用

```
Holder f;
```

两个线程，一个创建，一个使用

```
@Actor
public void a1() {
    f = new Holder(1);
}

@Actor
void a2(I_Result r) {
    Holder o = this.f;
```

```
        if (o != null) {
            r.r1 = o.x2 + o.x1;
        } else {
            r.r1 = -1;
        }
    }
```

可能会看到未构造完整的对象

4. 同步动作

前面没有详细展开从规则 2 之后的讲解，是因为要理解规则，还需理解底层原理，即内存屏障

4.1 内存屏障

> *共有四种内存屏障，具体实现与 CPU 架构相关，不必钻研太深，只需知道它们的效果*

LoadLoad

- 防止 B 的 Load 重排到 A 的 Load 之前

```
if(A) {
    LoadLoad
    return B
}
```

- 意义：A == true 时，再去获取 B，否则可能会由于重排导致 B 的值相对于 A 是过期的

LoadStore

- 防止 B 的 Store 被重排到 A 的 Load 之前

StoreStore

- 防止 A 的 Store 被重排到 B 的 Store 之后

```
A = x
StoreStore
B = true
```

- 意义：在 B 修改为 true 之前，其它线程别想看到 A 的修改
  - 有点类似于 sql 中更新后，commit 之前，其它事务不能看到这些更新（B 的赋值会触发 commit 并撤除屏障）

StoreLoad

- 意义：屏障前的改动都**同步到主存**[1]，屏障后的 Load **获取主存最新数据**
  - 防止屏障前所有的写操作，被重排序到屏障后的任何的读操作，可以认为此 `store -> load` 是连续的
  - 有点类似于 git 中先 commit，再远程 poll，而且这个动作是原子的

□ 如何记忆使用

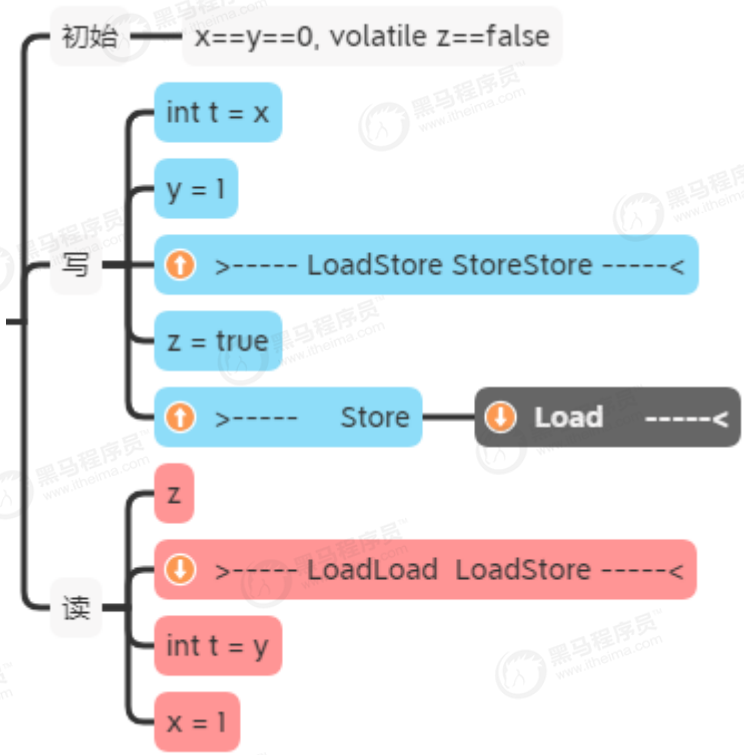
`LoadLoad + LoadStore = Acquire` 即让***同一线程***内***读操作之后的读写***上不去，第一个 Load 能读到主存最新值

`LoadStore + StoreStore = Release` 即让***同一线程***内***写操作之前的读写***下不来，后一个 Store 能将改动都写入主存

`StoreLoad` 最为特殊，还能用在***线程切换***时，对变量的***写操作 + 读操作***做同步，只要是对同一变量先写后读，那么屏障就能生效

4.2 volatile

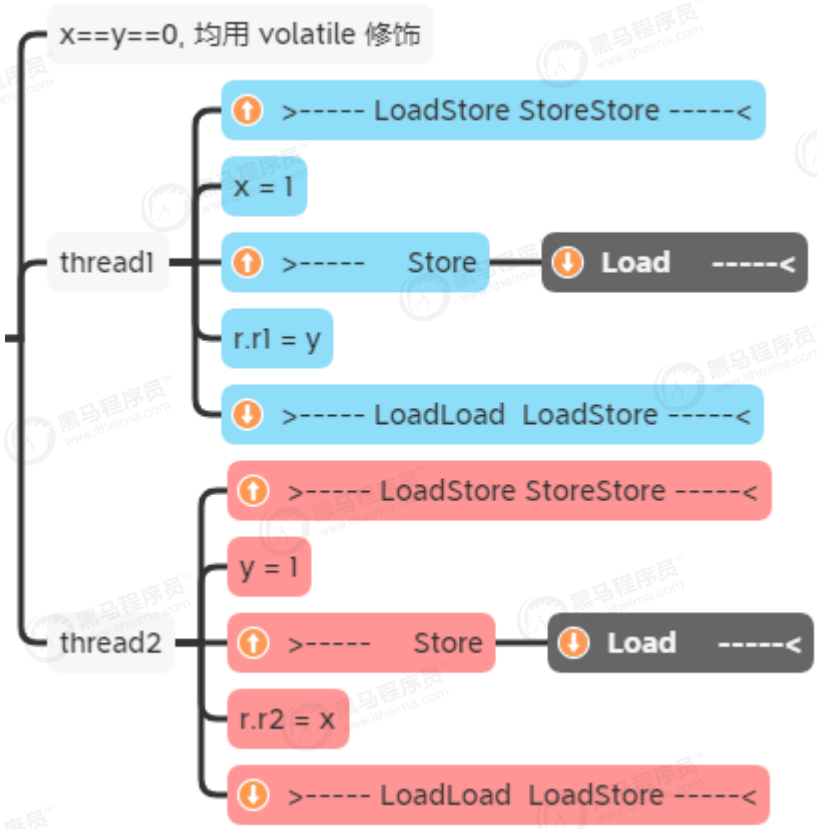
1. 本质



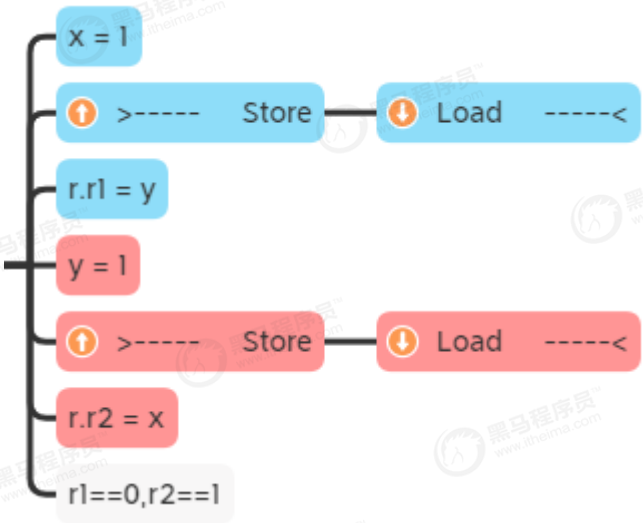
事实上对 volatile 而言 Store-Load 屏障最为有用，简化起见以后的分析省略部分其他屏障

例

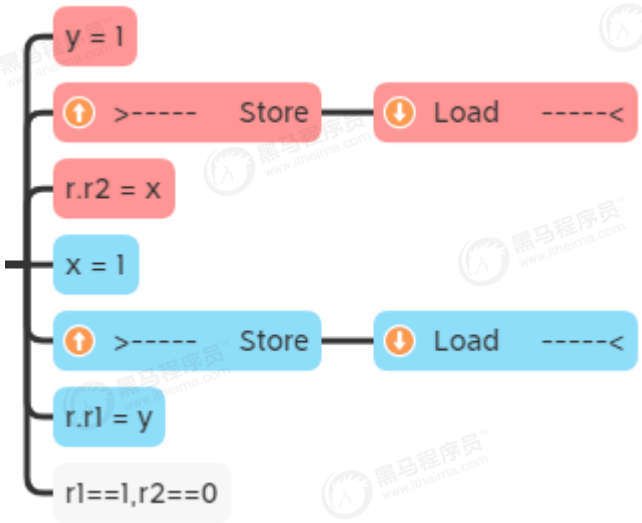
初始状态



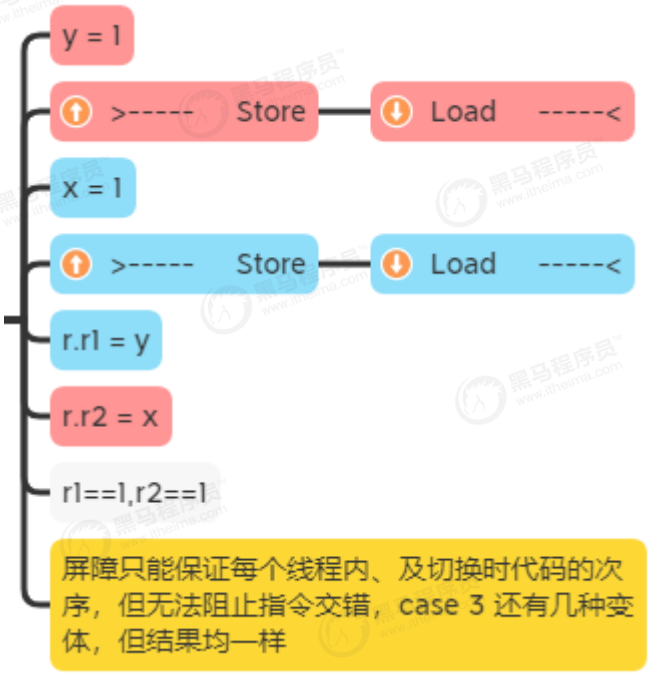
case 1



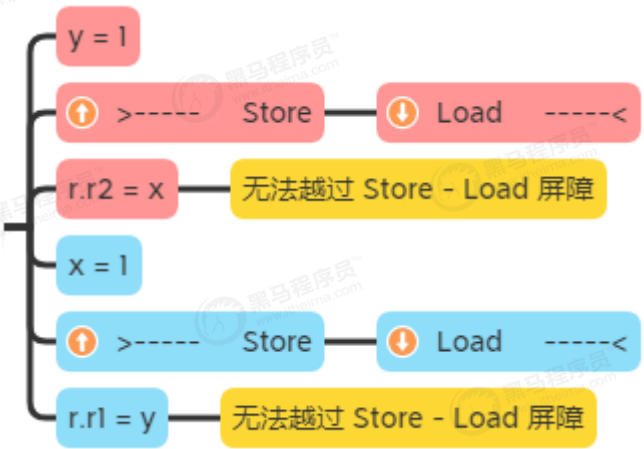
case 2



case 3



分析会出现 `r1==r2==0` 的情况吗？



💎 *单一变量的赋值原子性*

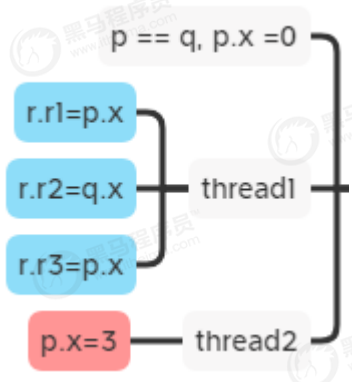
💎 *控制了可能的执行路径：线程内按屏障有序，线程切换时按 HB 有序*

💎 *可见性：线程切换时若发生了 `写 ->读` 则变量可见，顺带影响普通变量可见*

2. visibility

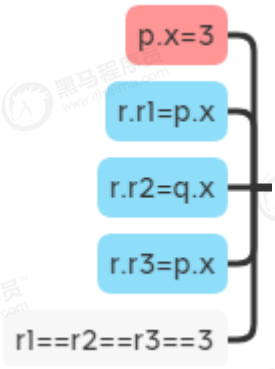
即使是多次读取同一变量，所得结果不合理

初始

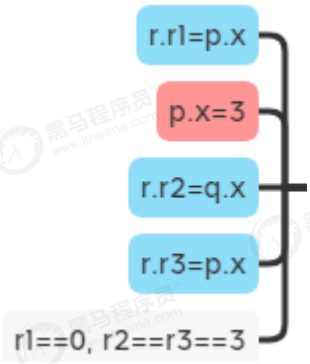


分析所有可能性

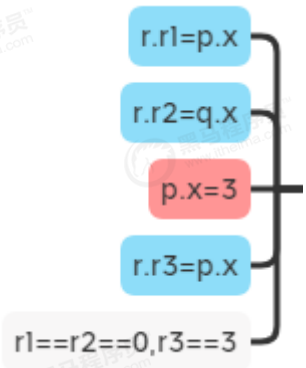
case 1



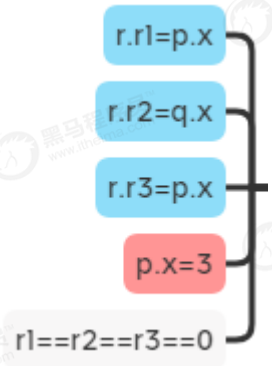
case 2



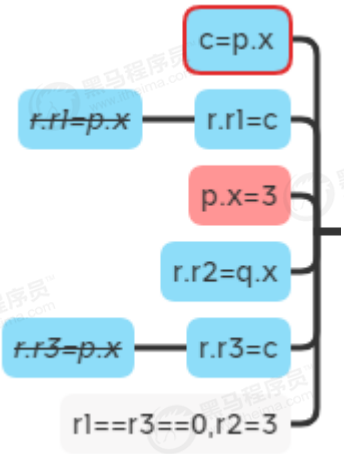
case 3



case 4



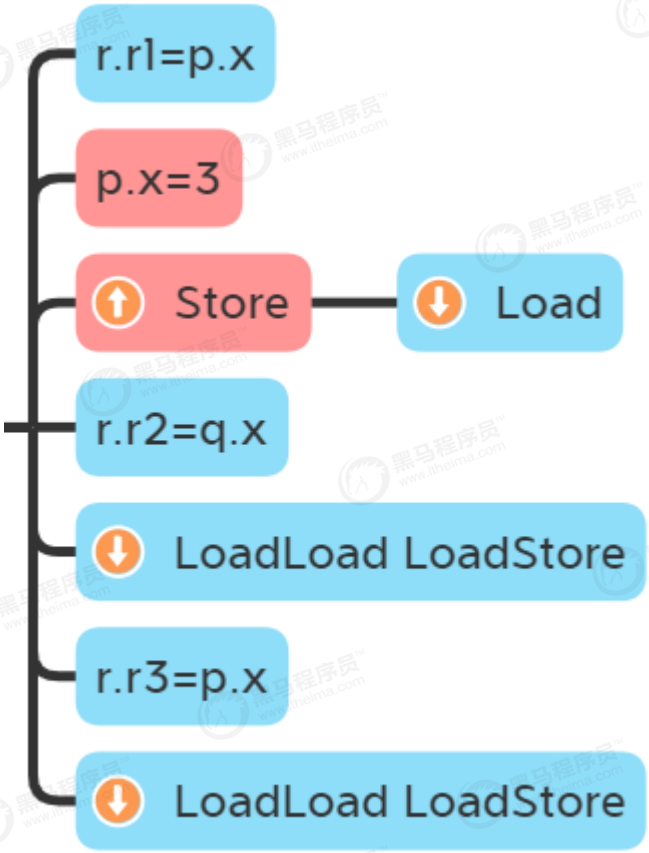
意外情况



使用 volatile 修饰 x 即可，测试用例如下

```
@JCStressTest
@Outcome(id = {"3, 3, 3", "0, 0, 0"}, expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(id = {"0, 3, 3", "0, 0, 3"}, expect = Expect.ACCEPTABLE_INTERESTING, desc =
"INTERESTING")
@Outcome(id = "0, 3, 0", expect = Expect.FORBIDDEN, desc = "FORBIDDEN")
@State
public static class Case2 {
    static class Foo {
        volatile int x = 0;
    }

    Foo p = new Foo();
    Foo q = p;

    @Actor
    public void actor1(III_Result r) {
        r.r1 = p.x;
        r.r2 = q.x;
        r.r3 = p.x;
    }

    @Actor
    public void actor2() {
        p.x = 3;
    }
}
```

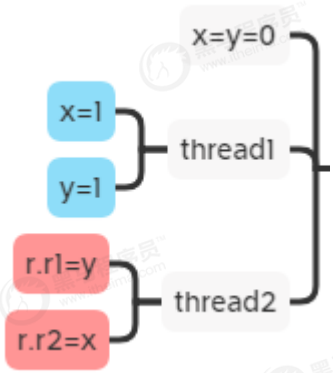
修复后



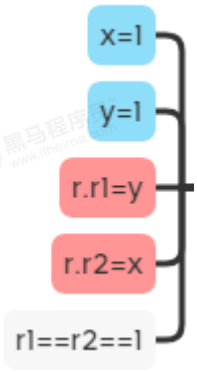
Store 屏障可以让红色线程对变量的修改同步到主存，而 Load 屏障可以让蓝色线程读到主存的最新值

3. partial ordering

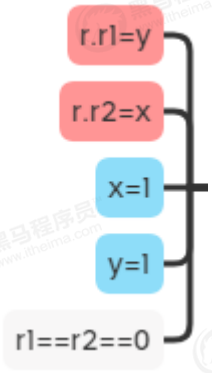
初始



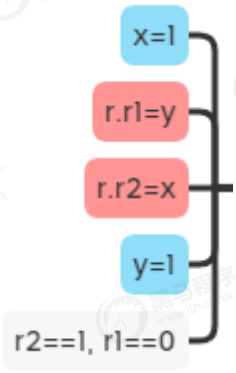
case 1



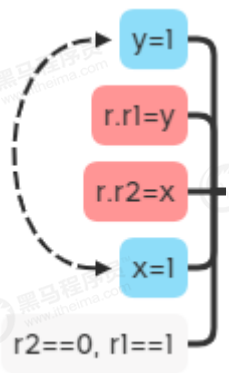
case 2



case 3

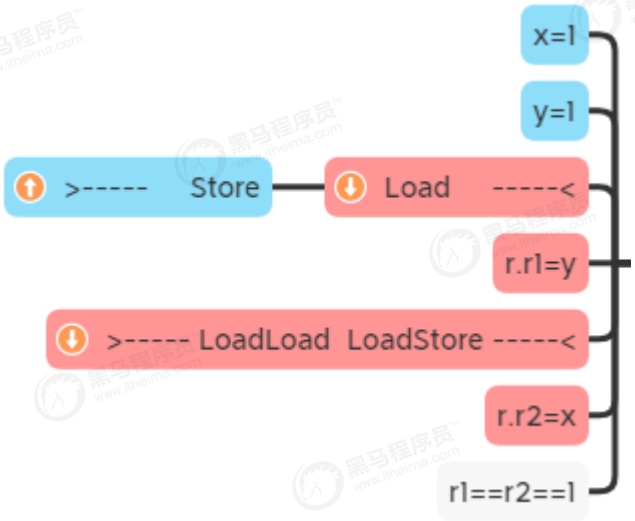


意外情况



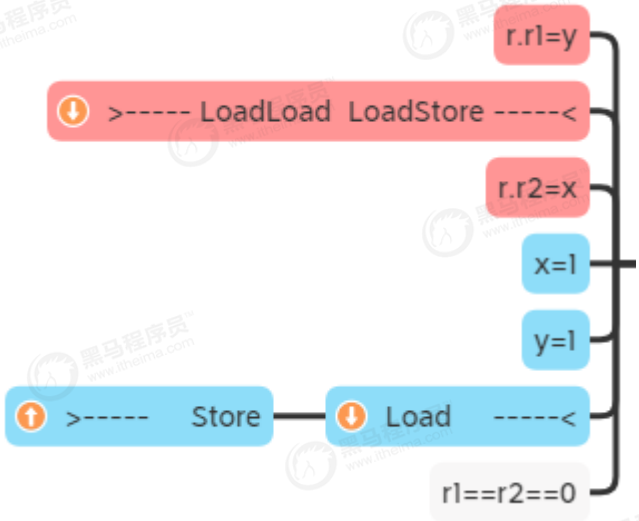
volatile 修饰 y

case 5，如果 y=1 先发生，那么前面的 Store 屏障会阻止 x=1 排下去，而后面的 Load 屏障会阻止后续的两个读操作排上来

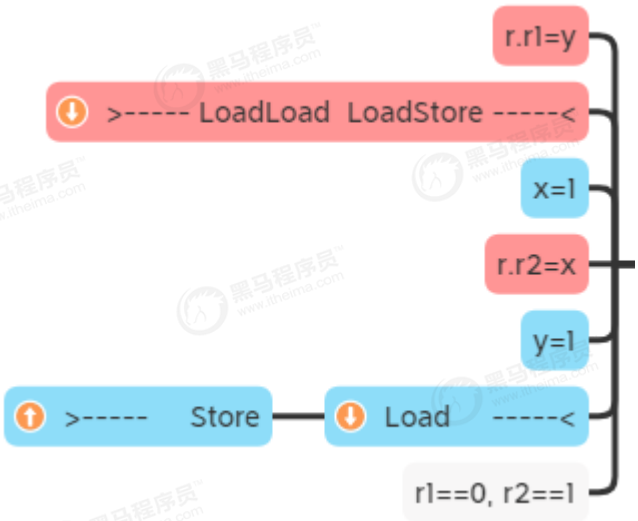


如果 y=1 后发生，前面的 Store 屏障会防止 x 排下去，但无法控制 r.r2=x 和 x=1 的执行顺序，（case 6 case 7），还有可能是 x=1 在 r.r1=y 之前，但结果与 case7 相同

case 6



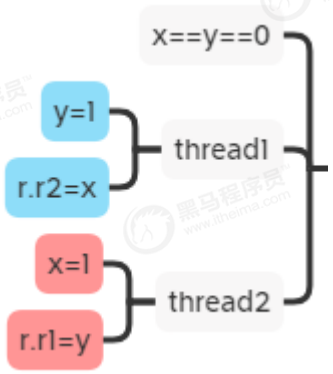
case 7



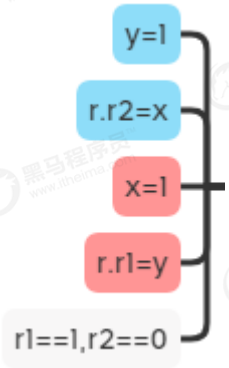
但观察 case 6 case 7，x=1 和 y=1 的位置被固定了，因此不会出现 case 4 的 `r2==0，r1==1` 情况

4. total ordering

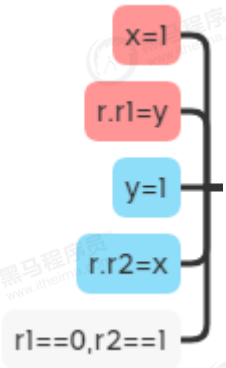
初始 - 先写再读



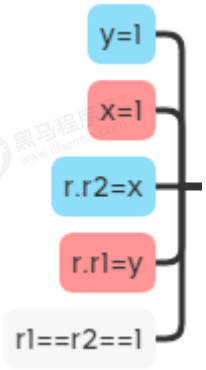
case 1



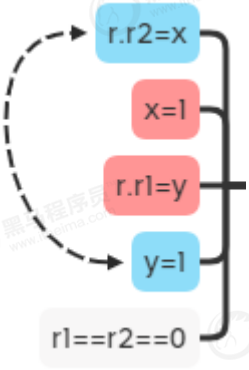
case 2



case 3

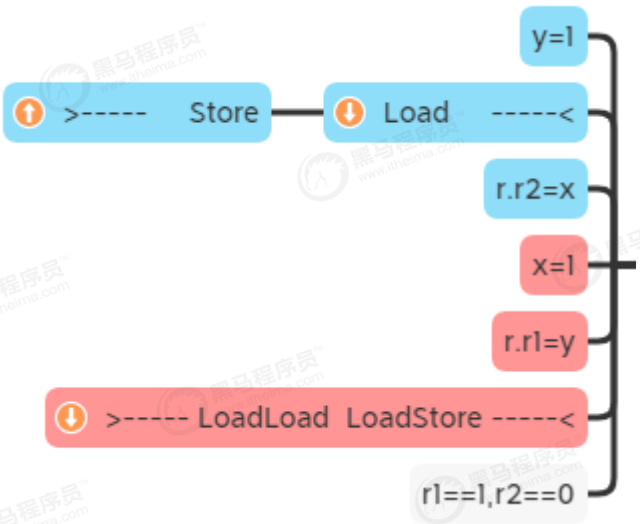


意外情况

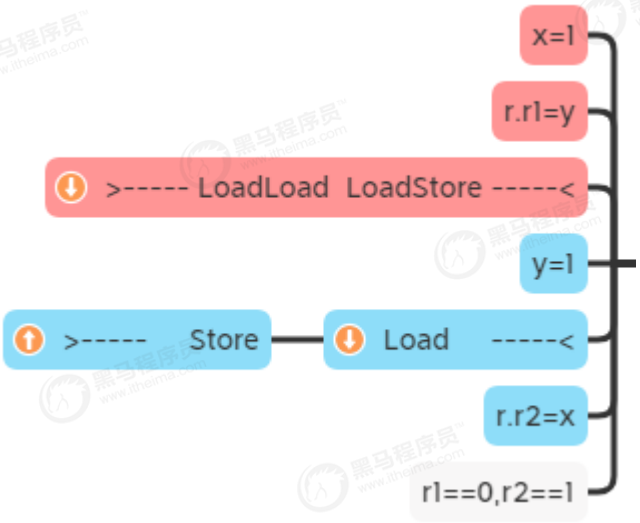


volatile 仅修饰 y - 先写再读

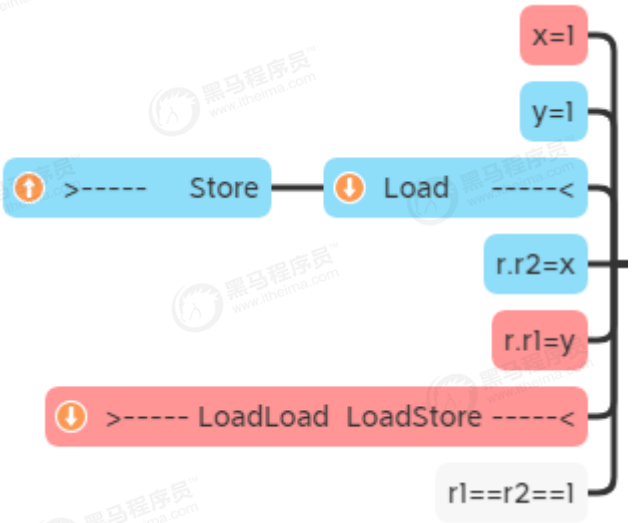
case 1



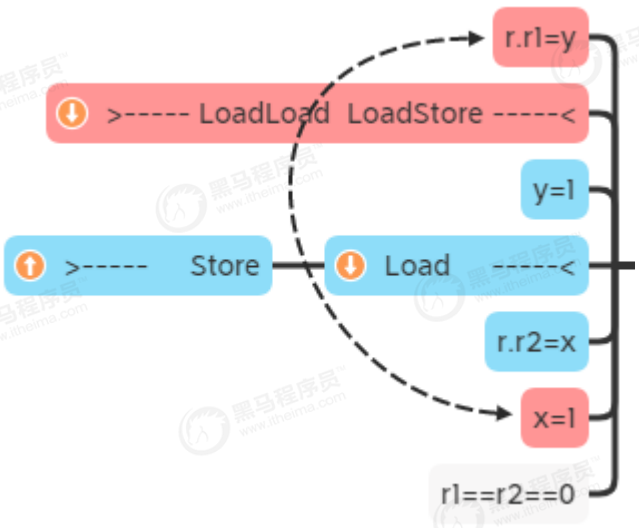
case 2



case 3

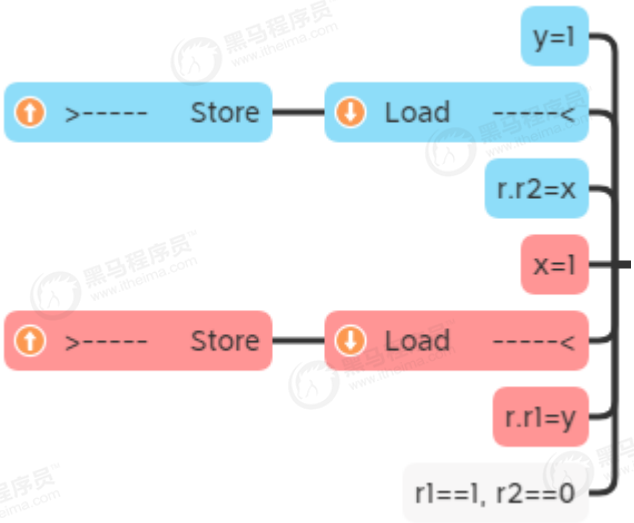


case 4，注意红色的 Load 屏障只能保证后续的读写不能排上去，但阻止不了 `x=1` 排下去，在 case 2的基础上稍加改动

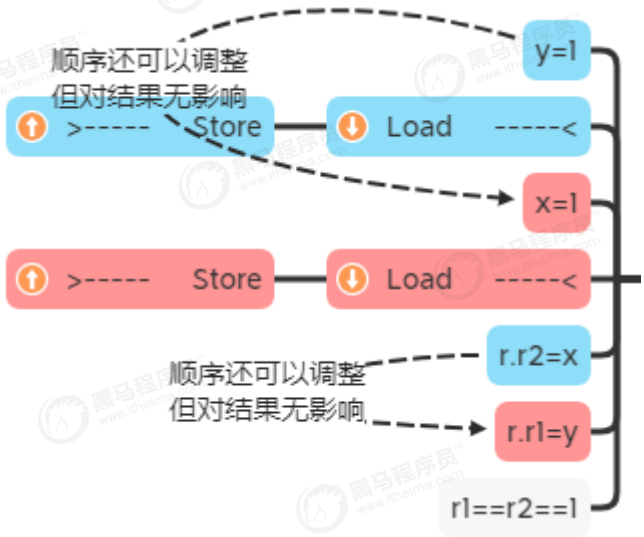


volatile 修饰 x 和 y - 先写再读

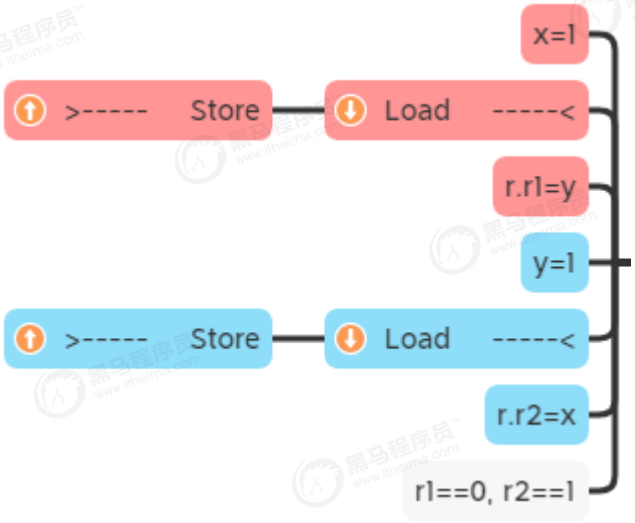
case 5，y=1 先于 r.r1=y，注意因为有 Store-Load 屏障的存在，蓝色相同线程的 r.r2=x 不会被排到 y=1 之前，同理，红色相同线程的 r.r1=y 不会被重排到 x=1 之前



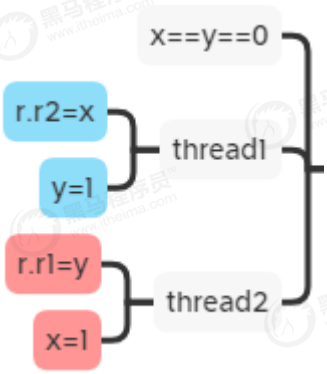
case 6，*还是y=1 先于r.r1=y，但是，线程间的两个操作r.r2=x 和x=1 的先后次序不固定，因此还可能是*



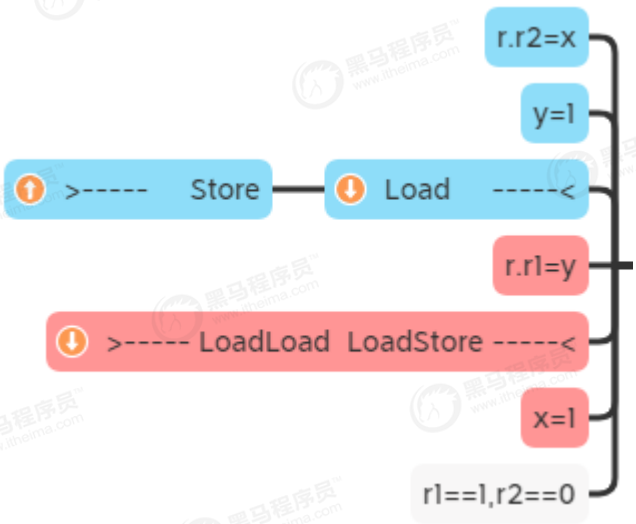
case 7，r.r1=y 先于 y=1，注意因为有 Store-Load 屏障的存在，蓝色相同线程的 r.r2=x 不会被排到 y=1 之前，同理，红色相同线程的 r.r1=y 不会被重排到 x=1 之前



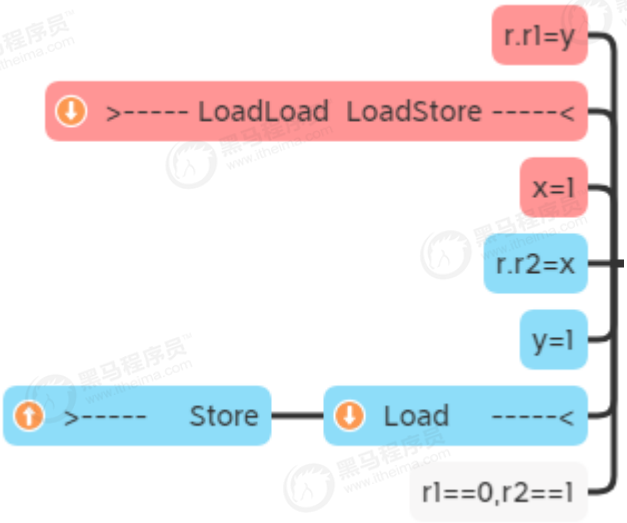
volatile 仅修饰 y - 先读再写



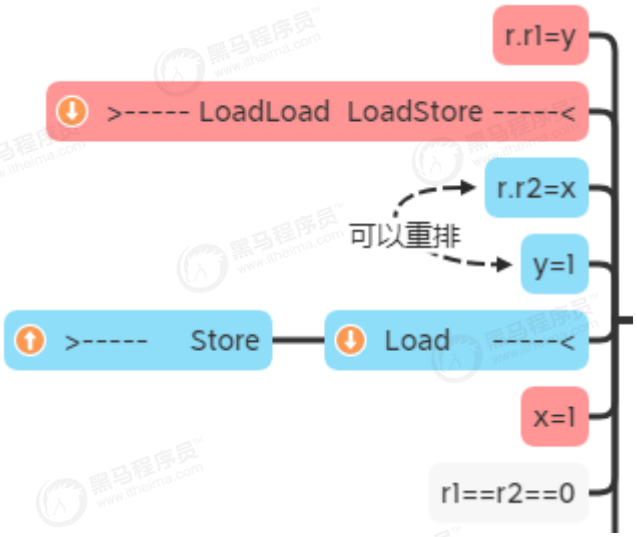
case 1



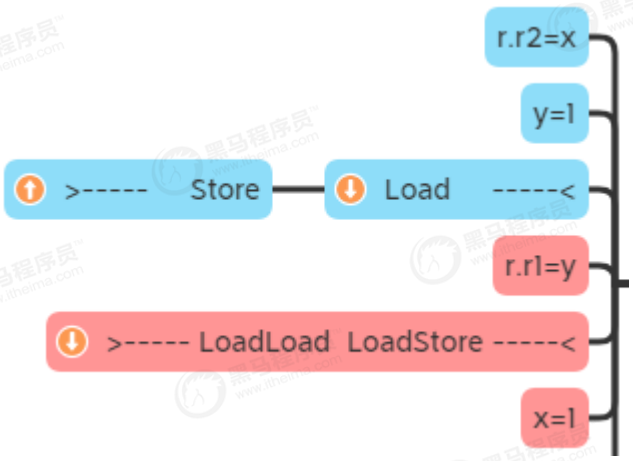
case 2



case 3，Store-Load 屏障不会生效，因为 y=1 仅会让前面的写操作不要重排到下面，对读操作阻止不了重排，不过结果是一样的



case 4，这回有用的是 y 读取动作导致的读屏障，它能够禁止后续的读写重排上去，因此在 y==1 的情况下，x 不能先等于 1



5. 源码体现

凡是需要 cas 操作的地方

代码片段 1 - AtomicInteger

```
public class AtomicInteger
    extends Number implements java.io.Serializable {

    private static final Unsafe U = Unsafe.getUnsafe();
    private static final long VALUE
        = U.objectFieldOffset(AtomicInteger.class, "value");

    private volatile int value;

    // ...

    public final boolean compareAndSet(int expectedVal, int newVal) {
        return U.compareAndSetInt(this, VALUE, expectedVal, newVal);
    }

    // ...
}
```

代码片段 2 - AbstractQueuedSynchronizer

```
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer
    implements java.io.Serializable {

    private transient volatile Node head;

    private transient volatile Node tail;

    private volatile int state;

    protected final int getState() {
        return state;
    }

    protected final boolean compareAndSetState(int e, int n) {
        return U.compareAndSetInt(this, STATE, e, n);
    }

    final void enqueue(Node node) {
        if (node != null) {
            for (;;) {
                Node t = tail;
                node.setPrevRelaxed(t);
                if (t == null)
                    tryInitializeHead();
```

```
                else if (casTail(t, node)) {
                    t.next = node;
                    if (t.status < 0)
                        LockSupport.unpark(node.waiter);
                    break;
                }
            }
        }
    }

    private void tryInitializeHead() {
        Node h = new ExclusiveNode(); // 头
        if (U.compareAndSetReference(this, HEAD, null, h))
            tail = h;
    }

    private boolean casTail(Node c, Node v) {
        return U.compareAndSetReference(this, TAIL, c, v);
    }
}
```

代码片段 3 - ConcurrentHashMap

```
public class ConcurrentHashMap<K,V> extends AbstractMap<K,V>
    implements ConcurrentMap<K,V>, Serializable {
    /**
     * Table initialization and resizing control.  When negative, the
     * table is being initialized or resized: -1 for initialization,
     * else -(1 + the number of active resizing threads).  Otherwise,
     * when table is null, holds the initial table size to use upon
     * creation, or 0 for default. After initialization, holds the
     * next element count value upon which to resize the table.
     */
    private transient volatile int sizeCtl;

    /**
     * The array of bins. Lazily initialized upon first insertion.
     * Size is always a power of two. Accessed directly by iterators.
     */
    transient volatile Node<K,V>[] table;

    private final Node<K,V>[] initTable() {
        Node<K,V>[] tab; int sc;
        while ((tab = table) == null || tab.length == 0) {
            if ((sc = sizeCtl) < 0)
                Thread.yield();
            else if (U.compareAndSetInt(this, SIZECTL, sc, -1)) {
                try {
                    if ((tab = table) == null || tab.length == 0) {
                        int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
```

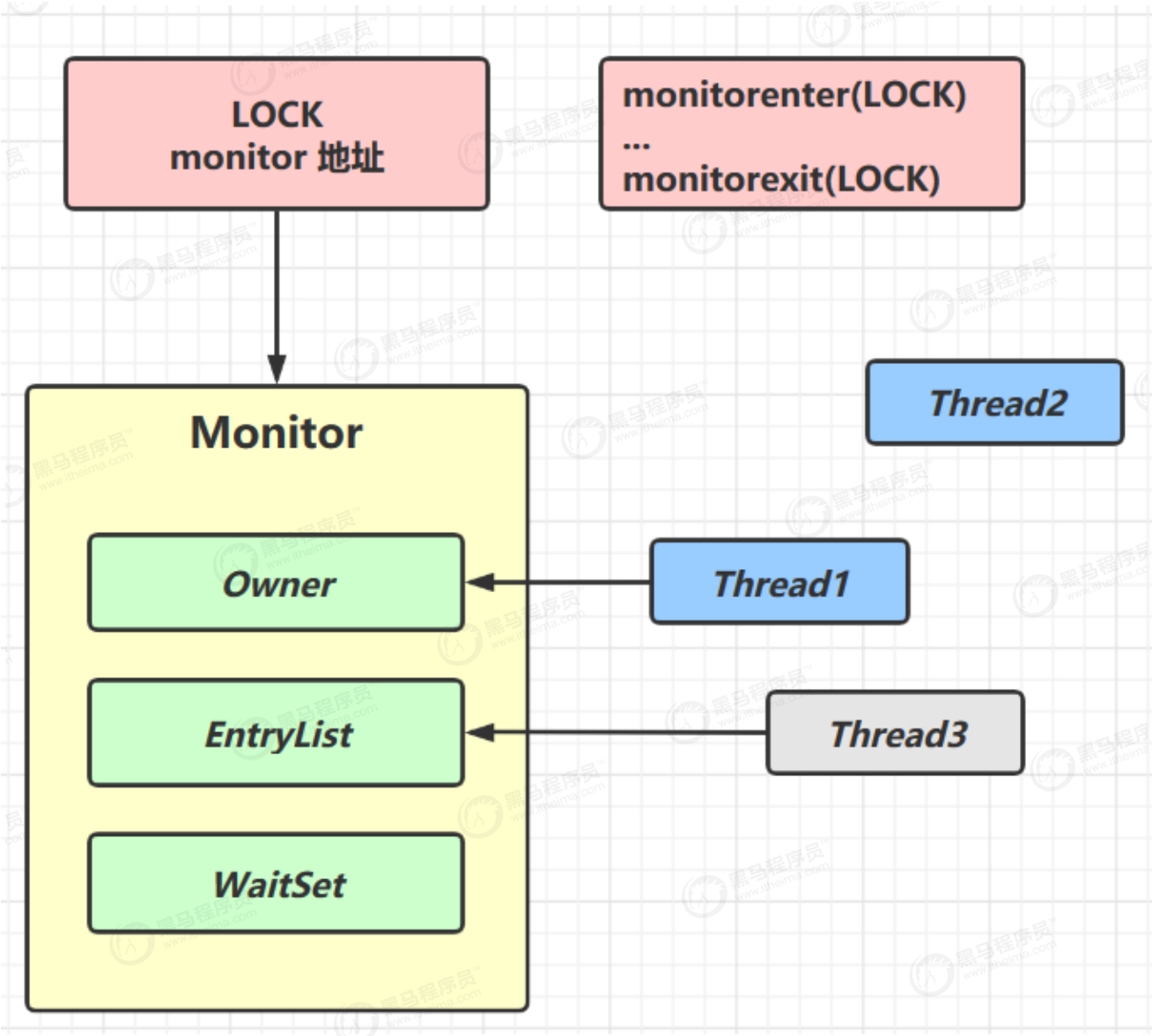
```
                        table = tab = nt;
                        sc = n - (n >>> 2);
                    }
                } finally {
                    sizeCtl = sc;
                }
                break;
            }
        }
        return tab;
    }

    // ...
}
```

💎 volatile 来负责可见、cas 来保障原子

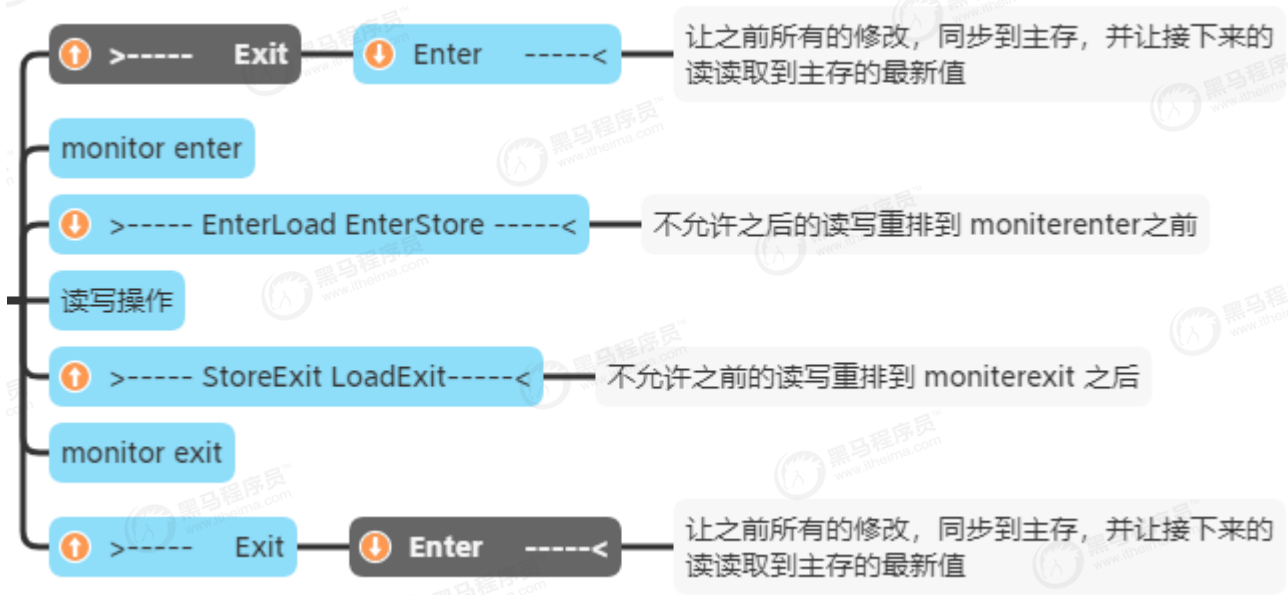
4.3 synchronized

1. 本质

monitorenter 与 monitorexit 工作原理


LOCK
monitor 地址
monitorenter(LOCK)
...
monitorexit(LOCK)
Monitor
Owner
EntryList
WaitSet
Thread2
Thread1
Thread3

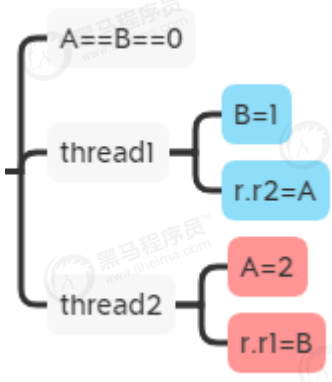
相关的内存屏障


>----- Exit
Enter -----<
让之前所有的修改，同步到主存，并让接下来的读读取到主存的最新值
monitor enter
>----- EnterLoad EnterStore -----<
不允许之后的读写重排到 moniterenter之前
读写操作
>----- StoreExit LoadExit-----<
不允许之前的读写重排到 moniterexit 之后
monitor exit
>----- Exit
Enter -----<
让之前所有的修改，同步到主存，并让接下来的读读取到主存的最新值

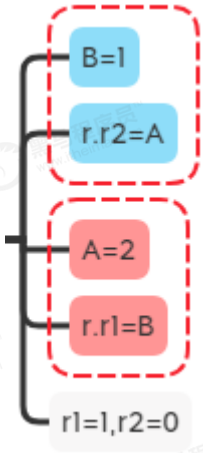
2. Atomicity

synchronized - 正确同步

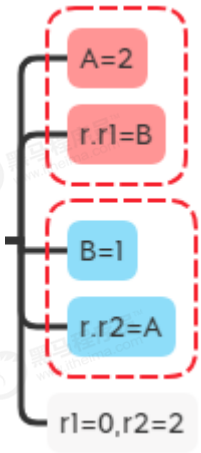
初始



case 1

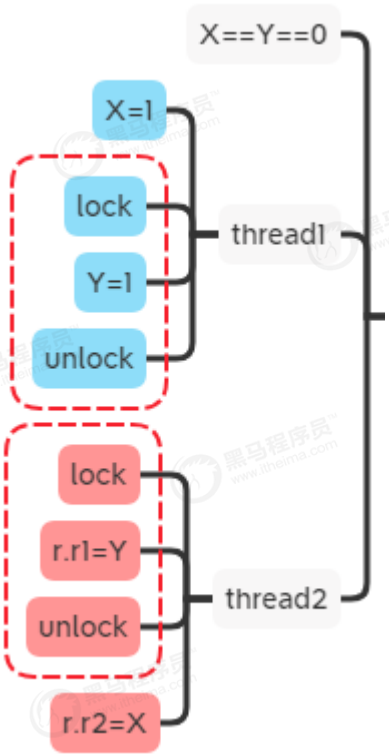


case 2

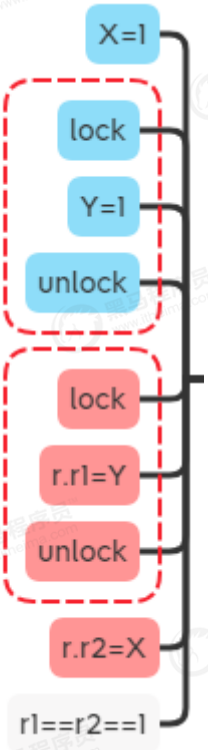


synchronized - 未正确同步

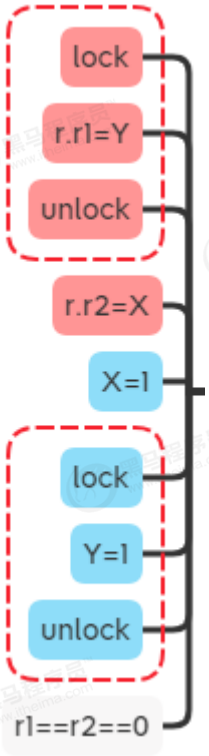
初始


X==Y==0
X=1
lock
Y=1
unlock
thread1
lock
r.r1=Y
unlock
r.r2=X
thread2

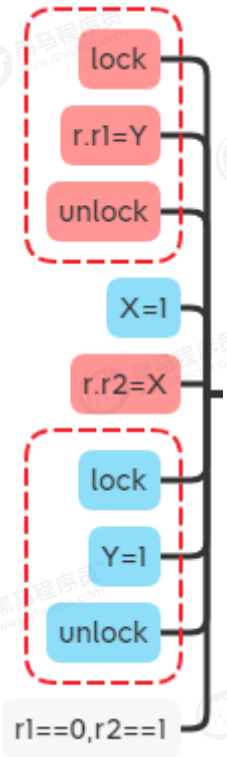
case1


X=1
lock
Y=1
unlock
lock
r.r1=Y
unlock
r.r2=X
r1==r2==1

case 2



case 3



3. 优化

- 重量级
  - 当有竞争时，仍会向系统申请 Monitor 互斥锁

- 轻量级锁
  - 如果线程加锁、解锁时间上刚好是错开的，这时候就可以使用轻量级锁，只是使用 cas 尝试将对象头替换为该线程的锁记录地址，如果 cas 失败，会锁重入或触发重量级锁升级
- 偏向锁
  - 打个比方，轻量级锁就好比用课本占座，线程每次占座前还得比较一下，课本是不是自己的（cas），频繁 cas 性能也会受到影响
  - 而偏向锁就好比座位上已经刻好了线程的名字，线程【专用】这个座位，比 cas 更为轻量
  - 但是一旦其他线程访问偏向对象，那么比较麻烦，需要把座位上的名字擦去，这称之为**偏向锁撤销**，锁也升级为轻量级锁
  - 偏向锁撤销也属于昂贵的操作，怎么减少呢，JVM 会记录**这一类对象**被撤销的次数，如果超过了 20 这个阈值，下次新线程访问偏向对象时，就不用撤销了，而是刻上新线程的名字，这称为**重偏向**
  - 如果撤销次数进一步增加，超过 40 这个阈值，JVM 会认为**这一类对象**不适合采用偏向锁，会对它们**禁用偏向锁**，下次新建对象会直接加轻量级锁

4. 无锁 vs 有锁

- synchronized 更为重量，申请锁、锁重入都要**发起系统调用**，频繁调用性能会受影响
- synchronized 如果无法获取锁时，线程会陷入阻塞，引起的**线程上下文切换**成本高
- 虽然做了一系列优化，但**轻量级锁**、**偏向锁**都是针对**无数据竞争场景**的

- 如果数据的原子操作时间较长，仍应该让线程阻塞，无锁适合的是**短频快的共享数据修改操作**主要用于**计数器**、**停止标记**、或是**阻塞前的有限尝试**

4.4 VarHandle

1. 目前无锁实现问题

目前 Java 中的无锁技术主要体现在以 AtomicInteger 为代表的的原子操作类，它的底层使用 Unsafe 实现，而 Unsafe 的问题在于安全性和可移植性

此外，volatile 主要使用了 Store-Load 屏障来控制顺序，这个屏障还是太强了，有没有更轻量级的解决方法呢？

2. VarHandle 快速上手

在 Java9 中引入了 VarHandle，来提供更细粒度的内存屏障，保证共享变量读写可见性、有序性、原子性。提供了更好的安全性和可移植性，替代 Unsafe 的部分功能

创建

```
public class TestVarHandle {
    int x;

    static VarHandle X;

    static {
        try {
            X = MethodHandles.lookup()
                .findVarHandle(TestVarHandle.class, "x", int.class);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
]
```

读写

方法名	作用	说明
get	获取值	与普通变量取值一样，会重排、有不可见现象
set	设置值	
getOpaque	获取值	对其保护的变量，保证其不重排和可见性，但不使用屏障，不阻碍其它变量
setOpaque	设置	
getAcquire	获取值	相当于 get 之后加 LoadLoad + LoadStore
setRelease	设置值	相当于 set 之前加 LoadStore + StoreStore
getVolatile	获取值	语义同 volatile，相当于获取之后加 LoadLoad + LoadStore
setVolatile	设置值	语义同 volatile，相当于设置之前加 LoadStore + StoreStore，设置之后加 StoreLoad
compareAndSet	原子赋值	原子赋值，成功返回 true，失败返回 false

3. visibility - 标记位

```
@JCStressTest(value = Mode.Termination)
@Outcome(id = {"TERMINATED"}, expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(expect = Expect.FORBIDDEN, desc = "FORBIDDEN")
public static class Case5 {
    boolean stop;
    static VarHandle STOP;

    static {
        try {
            STOP = MethodHandles.lookup()
                .findVarHandle(Case5.class, "stop", boolean.class);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    @Actor
    public void a1() {
        while (true) {
            if (((boolean) STOP.getOpaque(this))) {
                break;
            }
        }
    }

    @Signal
```

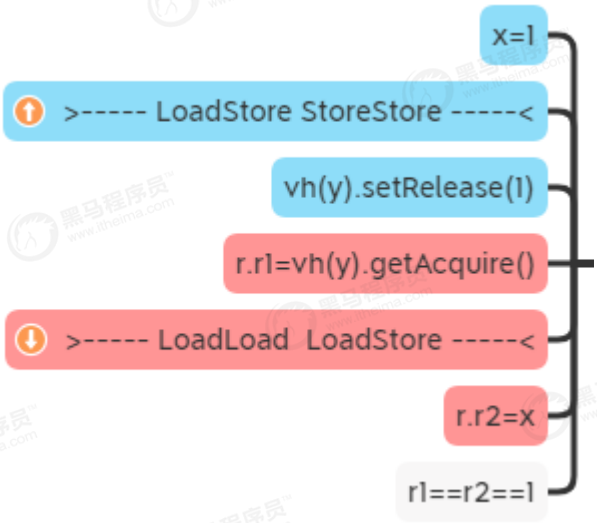
```
    void a2() {
        STOP.setOpaque(this, true);
    }
}
```

4. visibility - 连贯性

```
@JCStressTest
@Outcome(id = {"3, 3, 3", "0, 0, 0"}, expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(id = {"0, 3, 3", "0, 0, 3"}, expect = Expect.ACCEPTABLE_INTERESTING, desc =
"INTERESTING")
@Outcome(id = "0, 3, 0", expect = Expect.FORBIDDEN, desc = "FORBIDDEN")
@State
public static class Case3 {
    static VarHandle X;

    static {
        try {
            X = MethodHandles.lookup()
                .findVarHandle(Foo.class, "x", int.class);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }

    static class Foo {
        int x = 0;
    }

    Foo p = new Foo();
    Foo q = p;

    @Actor
    public void actor1(III_Result r) {
        r.r1 = (int) X.getOpaque(p);
        r.r2 = (int) X.getOpaque(q);
        r.r3 = (int) X.getOpaque(p);
    }

    @Actor
    public void actor2() {
        X.setOpaque(p, 3);
    }
}
```

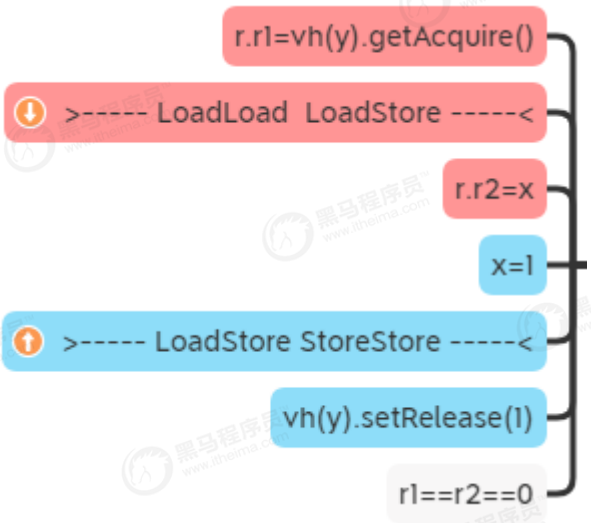
5. partial ordering

case 1，如果 y=1 先发生，那么前面的 Store 屏障会阻止 x=1 排下去，而后面的 Load 屏障会阻止后续的 r.r2=x 排上来

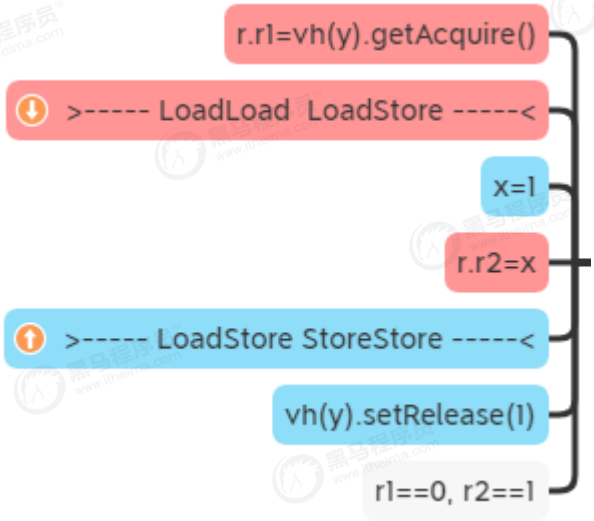


如果 y=1 后发生，那么 Store 屏障会阻止 x=1 排下去，而 Load 屏障会阻止后续的 r.r2=x 排上去，同样，无法控制 r.r2=x 和 x=1 的执行顺序（case 2 case 3）还有可能是 x=1 在 r.r1=y 之前，但结果与 case 3 相同

case 2

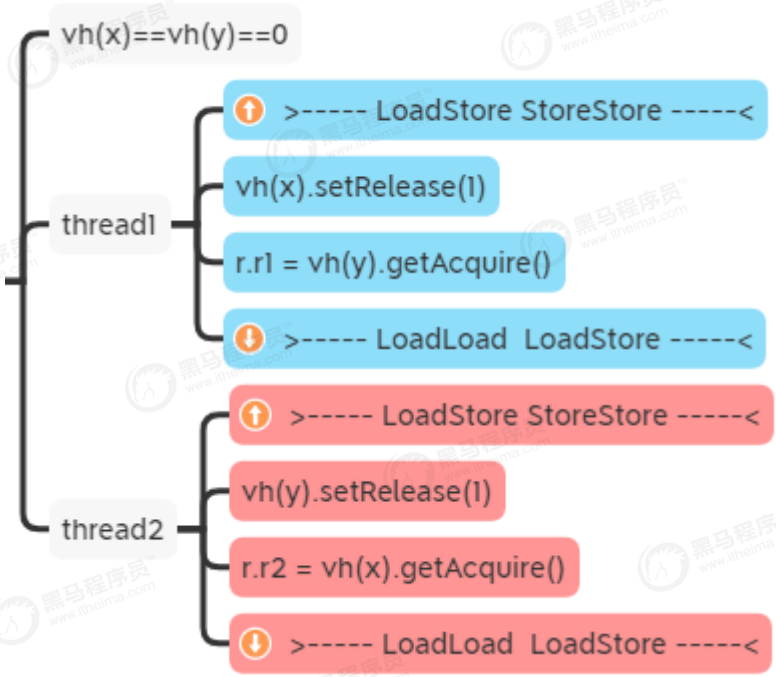


case 3

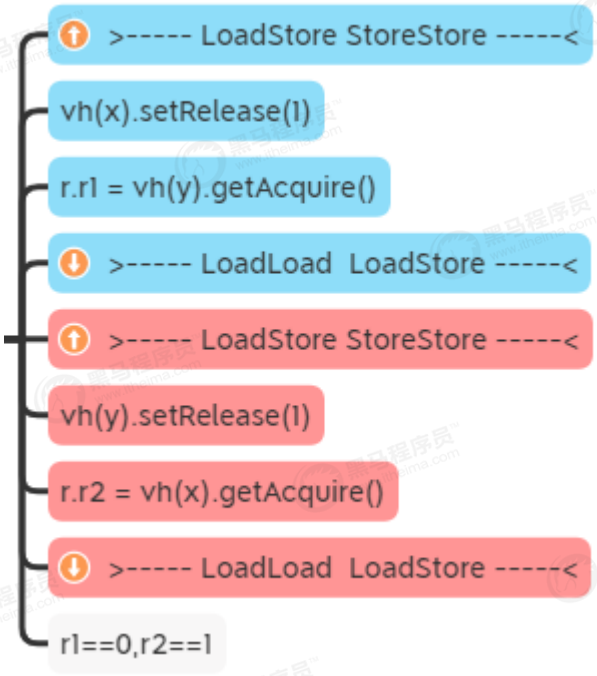


6. total ordering

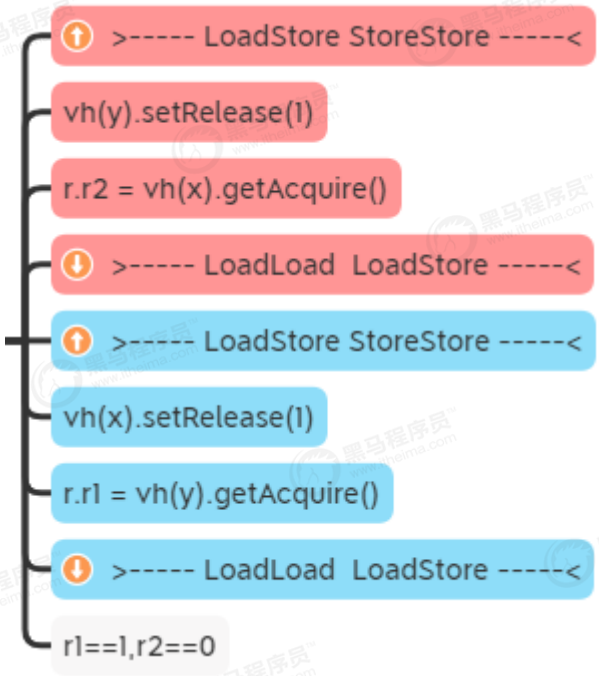
初始



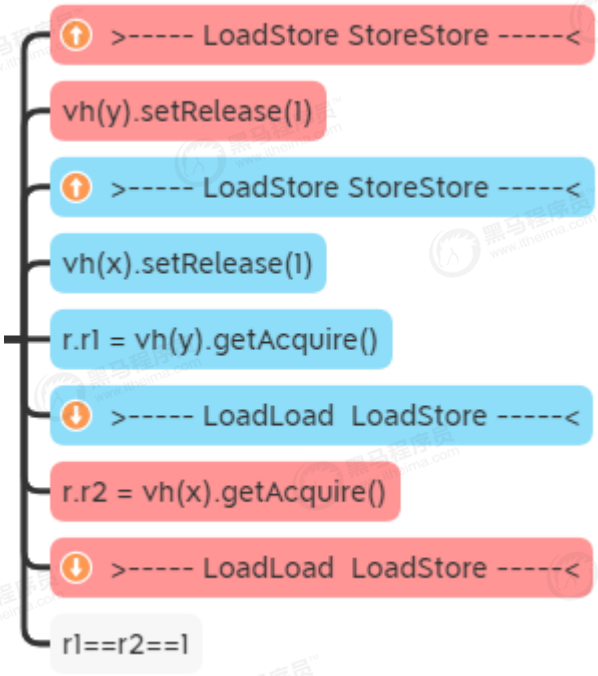
case 1


>----- LoadStore StoreStore -----<
vh(x).setRelease(1)
r.r1 = vh(y).getAcquire()
>----- LoadLoad LoadStore -----<
>----- LoadStore StoreStore -----<
vh(y).setRelease(1)
r.r2 = vh(x).getAcquire()
>----- LoadLoad LoadStore -----<
r1==0,r2==1

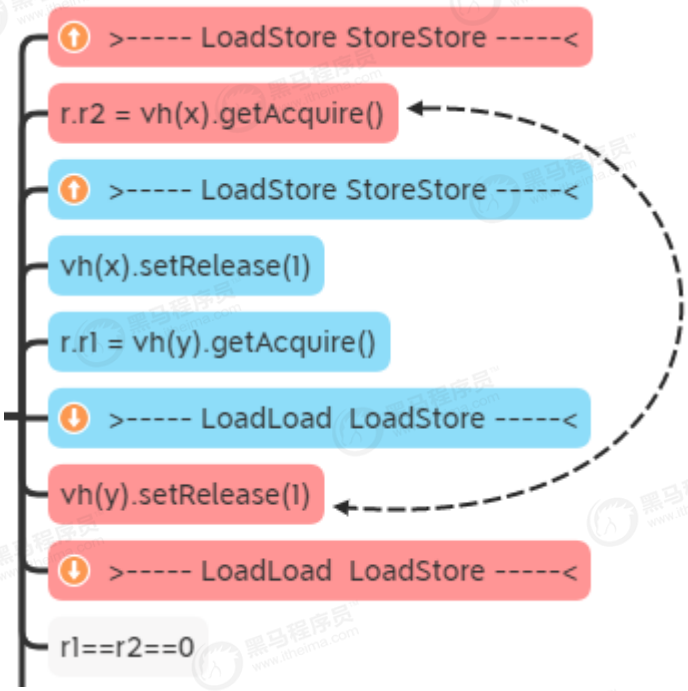
case 2


>----- LoadStore StoreStore -----<
vh(y).setRelease(1)
r.r2 = vh(x).getAcquire()
>----- LoadLoad LoadStore -----<
>----- LoadStore StoreStore -----<
vh(x).setRelease(1)
r.r1 = vh(y).getAcquire()
>----- LoadLoad LoadStore -----<
r1==1,r2==0

case 3



case 4，注意看红色的屏障之间的代码仍然可以被重排、蓝色的也一样，因为 case 3 中，红色 Load 屏障只能阻止下面的代码上不来，Store 屏障只能阻止上面的代码下不去，但在此之间，代码顺序无法保证



5. 更多安全问题

5.1 单个变量读写原子性

单个变量的读写原子性

- 64 位系统 vs 32 位系统
  - 如果需要保证 long 和 double 在 32 位系统中原子性，需要用 volatile 修饰
- JMM9 之前
- JMM9 32 位系统下 double 和 long 的问题，double 没有问题，long 在 -server -XX:+UnlockExperimentalVMOptions -XX:-AlwaysAtomicAccesses 才有问题

Long 类型变量测试

```
@JCStressTest
@Outcome(id = {"0", "-1"}, expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(expect = Expect.FORBIDDEN, desc = "FORBIDDEN")
@State
public static class Case3 {
    long x;

    @Actor
    public void actor1() {
        x = 0xFFFF_FFFF_FFFF_FFFFL;
    }

    @Actor
    public void actor2(L_Result r) {
        r.r1 = x;
    }
}
```

32 long double 字 4 个字节

64 字 8 个字节

Object alignment

你或许听说过对象对齐，它的一个主要目的就是为了单个变量读写的原子性，可以使用 jol 工具查看 java 对象的内存布局

```
<dependency>
    <groupId>org.openjdk.jol</groupId>
    <artifactId>jol-core</artifactId>
    <version>0.10</version>
</dependency>
```

测试类

```
public class TestJol {
    public static void main(String[] args) {
        String layout = ClassLayout.parseClass(Test.class).toPrintable();
        System.out.println(layout);
    }

    public static class Test {
        private byte a;
        private byte b;
        private byte c;
        private long e;
    }
}
```

开启对象头压缩（默认）输出

```
com.itheima.test.TestJol$Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0    12        (object header)                           N/A
     12     1   byte Test.a                                    N/A
     13     1   byte Test.b                                    N/A
     14     1   byte Test.c                                    N/A
     15     1        (alignment/padding gap)
     16     8   long Test.e                                    N/A
Instance size: 24 bytes
Space losses: 1 bytes internal + 0 bytes external = 1 bytes total
```

不开启对象头压缩 `-XX:-UseCompressedOops` 输出

```
com.itheima.test.TestJol$Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0    16        (object header)                           N/A
     16     8   long Test.e                                    N/A
     24     1   byte Test.a                                    N/A
     25     1   byte Test.b                                    N/A
     26     1   byte Test.c                                    N/A
     27     5        (loss due to the next object alignment)
Instance size: 32 bytes
Space losses: 0 bytes internal + 5 bytes external = 5 bytes total
```

5.2 字分裂

前面也看到了，Java 能够保证单个共享变量**读写**是原子的，类似的数组元素的读写，也会提供这样的保证

```
byte[8]

[0][1][2][3]

[0][1][2][3]
```

如果上述效果不能保证，则称之为发生了字分裂现象，java 中没有字分裂[2]，但 Java 中某些实现会有类似字分裂现象，例如 BitSet、Unsafe 读写等

数组元素读写测试

```
@JCStressTest
@Outcome(id = {"0", "-1"}, expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(expect = Expect.FORBIDDEN, desc = "FORBIDDEN")
@State
public static class Case4 {
    byte[] b = new byte[256];
    int off = ThreadLocalRandom.current().nextInt(256);

    @Actor
    public void actor1() {
```

```
        b[off] = (byte) 0xFF;
    }

    @Actor
    public void actor2(I_Result r) {
        r.r1 = b[off];
    }
}
```

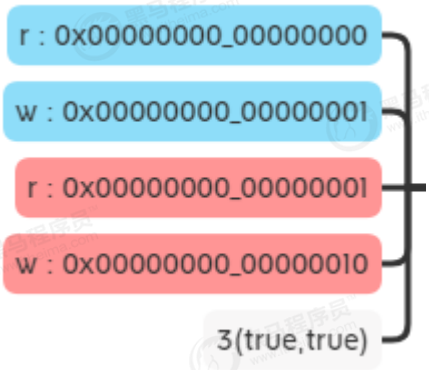
BitSet 读写测试

```
boolean 1 个字节

long[]

long
```

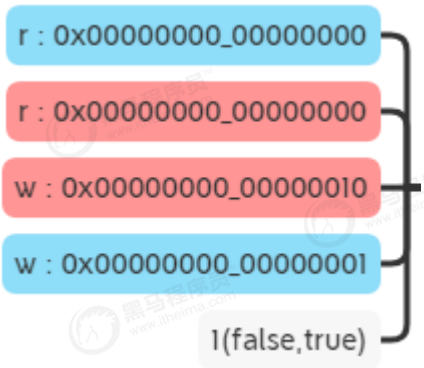
```
@JCStressTest
@Outcome(id = "true, true", expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(expect = Expect.ACCEPTABLE_INTERESTING, desc = "INTERESTING")
@State
public static class Case6 {
    BitSet b = new BitSet();

    @Actor
    public void a() {
        b.set(0);
    }

    @Actor
    public void b() {
        b.set(1);
    }

    @Arbiter
    public void c(ZZ_Result r) {
        r.r1 = b.get(0);
        r.r2 = b.get(1);
    }
}
```

分析

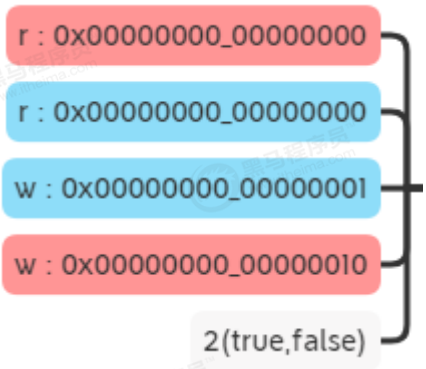
case 1



case 2



case 3



Unsafe 直接操作内存

```
public class TestUnsafe {
    public static final long ARRAY_BASE_OFFSET =
UnsafeHolder.U.arrayBaseOffset(byte[].class);

    static byte[] ss = new byte[8];
    public static void main(String[] args) {
        System.out.println(ARRAY_BASE_OFFSET);

        UnsafeHolder.U.putInt(ss, ARRAY_BASE_OFFSET, 0xFFFFFFFF);
        System.out.println(Arrays.toString(ss));
    }
}
```

输出

```
16
[-1, -1, -1, -1, 0, 0, 0, 0]
```

来个压测

```
@JCStressTest
@Outcome(id = "0", expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(id = "-1", expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(expect = Expect.ACCEPTABLE_INTERESTING, desc = "INTERESTING")
@State
public static class Case5 {
    byte[] ss = new byte[256];
    long base = UnsafeHolder.U.arrayBaseOffset(byte[].class);
    long off = base + ThreadLocalRandom.current().nextInt(256 - 4);

    @Actor
    public void writer() {
        UnsafeHolder.U.putInt(ss,  off, 0xFFFF_FFFF);
    }

    @Actor
    public void reader(I_Result r) {
        r.r1 = UnsafeHolder.U.getInt(ss, off);
    }
}
```

某次结果

```
Observed state   Occurrences              Expectation  Interpretation
            -1    25,591,098               ACCEPTABLE  ACCEPTABLE
      -16777216           877   ACCEPTABLE_INTERESTING  INTERESTING
          -256           923   ACCEPTABLE_INTERESTING  INTERESTING
         -65536           925   ACCEPTABLE_INTERESTING  INTERESTING
             0     5,093,890               ACCEPTABLE  ACCEPTABLE
       16777215         1,673   ACCEPTABLE_INTERESTING  INTERESTING
           255         1,758   ACCEPTABLE_INTERESTING  INTERESTING
         65535         1,707   ACCEPTABLE_INTERESTING  INTERESTING
```

5.3 安全发布

构造也不安全

```
@JCStressTest
@Outcome(id = {"16", "-1"}, expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(expect = Expect.ACCEPTABLE_INTERESTING, desc = "INTERESTING")
@State
public static class Case1 {
    Holder f;

    int v = 1;

    @Actor
    public void a1() {
        f = new Holder(v);
    }

    @Actor
    void a2(I_Result r) {
        Holder o = this.f;
        if (o != null) {
            r.r1 = o.x8 + o.x7 + o.x6 + o.x5 + o.x4 + o.x3 + o.x2 + o.x1;
            r.r1 += o.y8 + o.y7 + o.y6 + o.y5 + o.y4 + o.y3 + o.y2 + o.y1;
        } else {
            r.r1 = -1;
        }
    }

    static class Holder {
        int x1, x2, x3, x4;
        int x5, x6, x7, x8;
        int y1, y2, y3, y4;
        int y5, y6, y7, y8;
```

```
        public Holder(int v) {
            x1 = v;
            x2 = v;
            x3 = v;
            x4 = v;
            x5 = v;
            x6 = v;
            x7 = v;
            x8 = v;
            y1 = v;
            y2 = v;
            y3 = v;
            y4 = v;
            y5 = v;
            y6 = v;
            y7 = v;
            y8 = v;
        }
    }
}
```

原因分析，看字节码

使用 final 改进

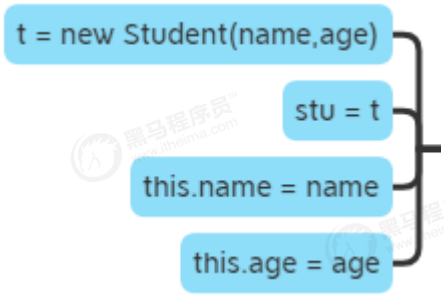
```
@JCStressTest
@Outcome(id = {"16", "-1"}, expect = Expect.ACCEPTABLE, desc = "ACCEPTABLE")
@Outcome(expect = Expect.FORBIDDEN, desc = "FORBIDDEN")
@State
public static class Case2 {
    Holder f;

    int v = 1;

    @Actor
    public void a1() {
        f = new Holder(v);
    }

    @Actor
    void a2(I_Result r) {
        Holder o = this.f;
        if (o != null) {
            r.r1 = o.x8 + o.x7 + o.x6 + o.x5 + o.x4 + o.x3 + o.x2 + o.x1;
            r.r1 += o.y8 + o.y7 + o.y6 + o.y5 + o.y4 + o.y3 + o.y2 + o.y1;
```

```
            } else {
                r.r1 = -1;
            }
        }

        static class Holder {
            final int x1;
            int x2, x3, x4;
            int x5, x6, x7, x8;
            int y1, y2, y3, y4;
            int y5, y6, y7, y8;

            public Holder(int v) {
                x1 = v;
                x2 = v;
                x3 = v;
                x4 = v;
                x5 = v;
                x6 = v;
                x7 = v;
                x8 = v;
                y1 = v;
                y2 = v;
                y3 = v;
                y4 = v;
                y5 = v;
                y6 = v;
                y7 = v;
                y8 = v;
            }
        }
    }
```

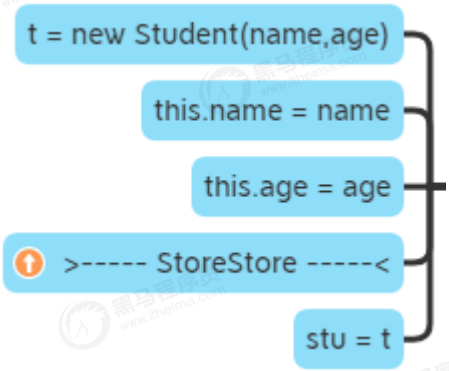
原因分析

初始

name 如果没有 final 修饰

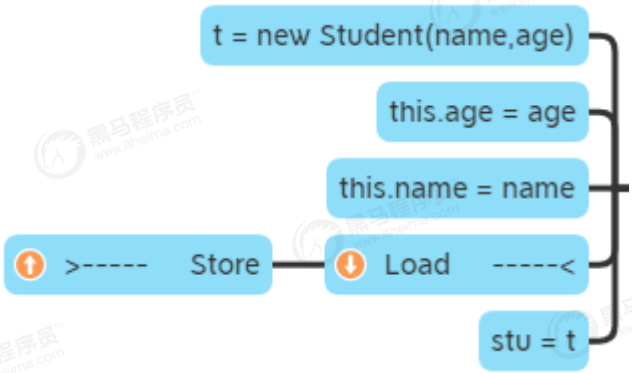


name 有 final 修饰，位置任意



使用 volatile 改进

name 有 volatile 修饰，注意位置必须在最后



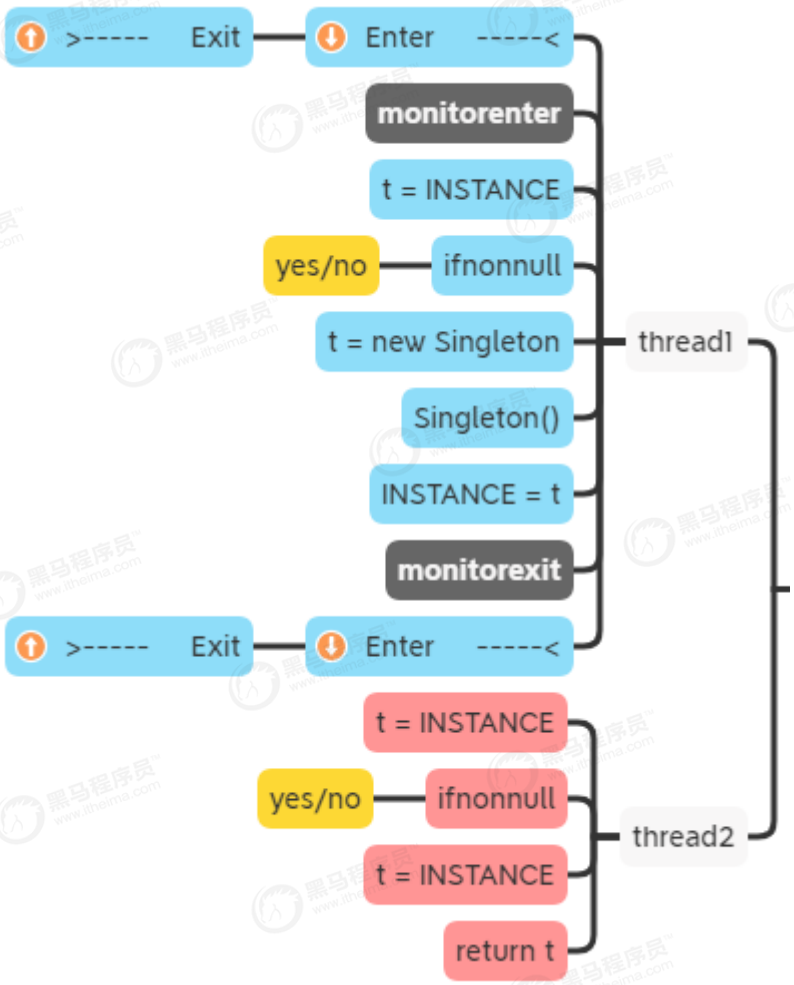
DCL 安全单例

```
public class Singleton {
    private static Singleton INSTANCE = null;

    public static Singleton getInstance() {
        if (INSTANCE == null) {
            synchronized (Singleton.class) {
                if (INSTANCE == null) {
                    INSTANCE = new Singleton();
                }
            }
        }
        return INSTANCE;
```

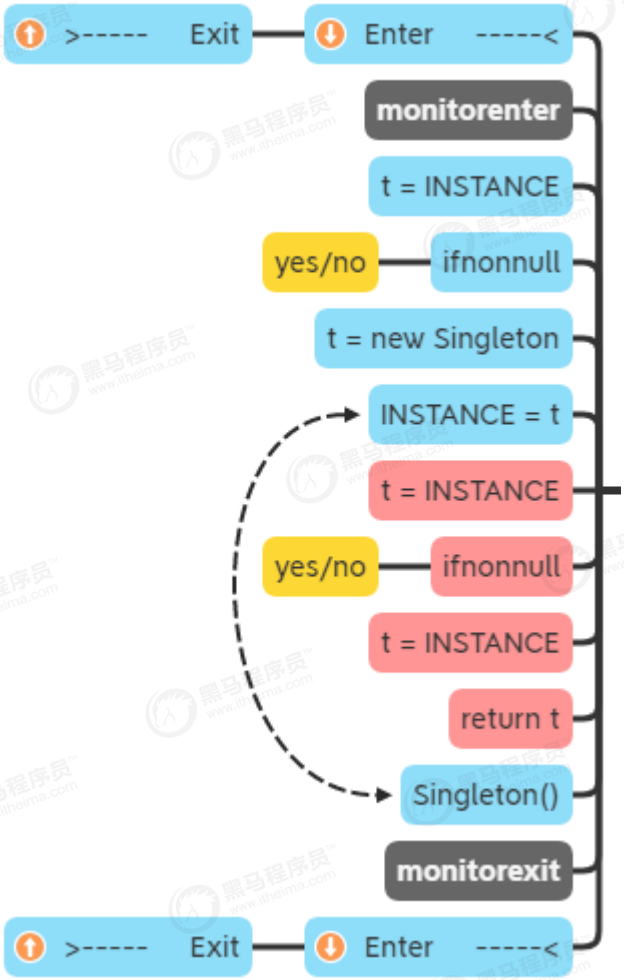
```
    }
}
```

试用今天所学知识，分析为什么上述写法不正确

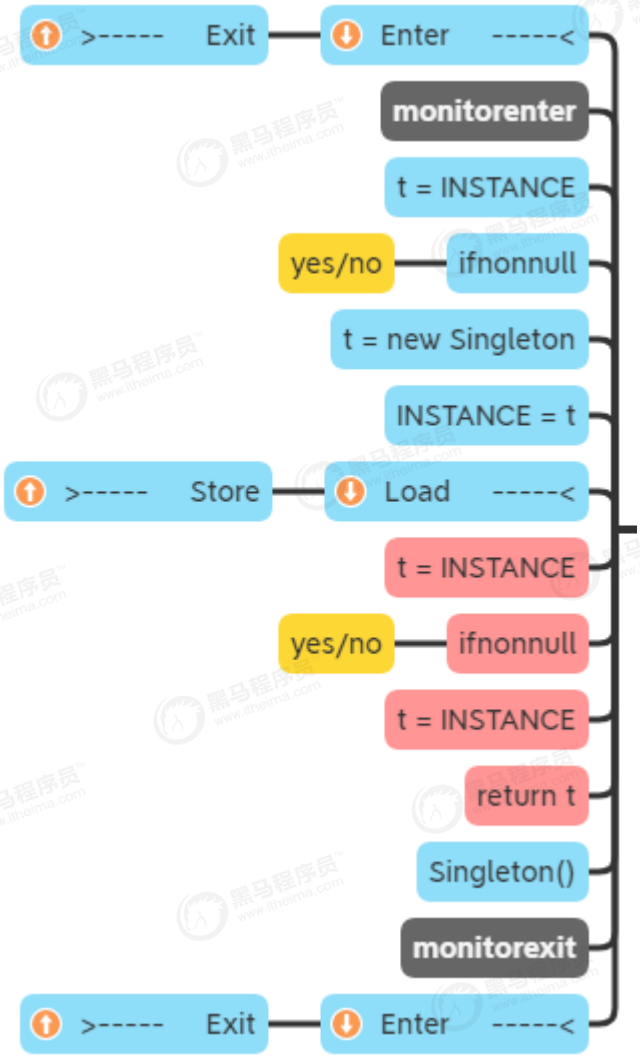
初始



有问题的情况



使用 volatile 解决



练习 - 实现一个无锁单例

提示：可以使用 volatile 配合 cas，或者 VarHandle 配合 cas 来实现

6. 总结

1. JMM 是研究的是
    - 多线程下 Java 代码的执行顺序，实际代码的执行顺序与你编写的代码顺序不同
    - 共享变量的读写操作，在竞态条件下，需要考虑共享变量读写的原子性、可见性、有序性
2. 共享变量的问题起因
    - 原子性是由于操作系统的分时机制，线程切换所致
    - *有序性和可见性可能来自于编译器优化、处理器优化、缓存优化*
3. JMM 制定了一些规则，理解这些规则，才能写出正确的线程安全代码
    - 竞态条件会导致代码顺序被重排
    - 利用 synchronized、volatile 一些 SA，可以控制线程内代码的执行顺序
    - 线程切换时的执行顺序与可见性，遵守 HB 规则
    - HB 规则还不足够，需要因果律作为补充
    - 可以通过 final 或 volatile 实现对象的安全发布
4. 从底层理解 volatile 与 synchronized
    - 内存屏障
    - synchronized 是如何解决原子性、可见性、有序性问题的，有哪些优化
    - volatile 是如何解决可见性、有序性问题的，与 cas 结合的威力
    - VarHandle 是如何解决可见性、有序性问题的
5. 更多安全问题
    - 单个变量、数组元素的读写原子性
    - 能够列举字分裂的几个相关例子
    - 构造方法什么情况下会线程不安全，如何改进
    - 彻底掌握 DCL 安全单例

---

1. 屏障改动是同步到主存吗？前面讲解 CPU 缓存一致性时分析过，真正引起指令重排的其实是 StoreBuffer，而不是缓存，缓存真正地位相当于我们平时说的主存。因此，严格的说屏障的改动是**同步到缓存**就 OK 了，但如果都这么做会增大分析与理解难度和不必要的争论，因此仍然保留网上大部分的说法屏障前的改动都**同步到主存**。↩

2. 某些处理器没有单个 byte 级别的读写，以 int 为单位读写↩