主讲老师： Fox

> 有道笔记地址：https://note.youdao.com/s/KiwUihrE

1. 并发三大特性

并发编程Bug的源头：原子性、可见性和有序性问题

1.1 原子性

一个或多个操作，要么全部执行且在执行过程中不被任何因素打断，要么全部不执行。在 Java 中，对基本数据类型的变量的读取和赋值操作是原子性操作（64位处理器）。不采取任何的原子性保障措施的自增操作并不是原子性的，比如i++操作。

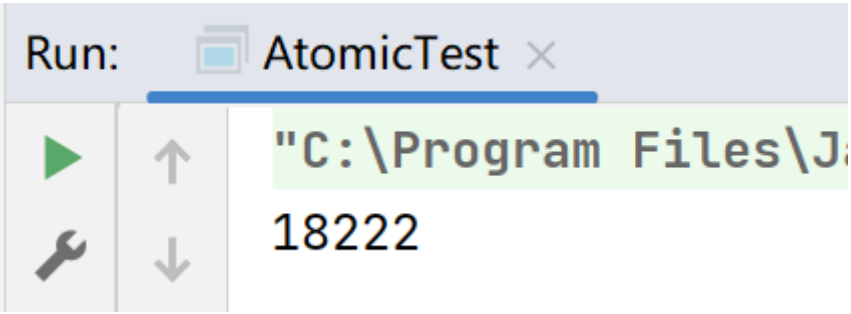
原子性案例分析

下面例子模拟多线程累加操作

```
public class AtomicTest {
    private static int counter = 0;

    public static void main(String[] args) {

        for (int i = 0; i < 10; i++) {
            Thread thread = new Thread(() -> {
                for (int j = 0; j < 10000; j++) {
                    counter++;
                }

            });
            thread.start();
        }

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

```

```
22          System.out.println(counter);
23
24      }
25
26  }
27
```

执行结果不确定， 与预期结果不符合，存在线程安全问题



如何保证原子性

- 通过 synchronized 关键字保证原子性
- 通过 Lock锁保证原子性
- 通过 CAS保证原子性

思考：在 32 位的机器上对 long 型变量进行加减操作是否存在并发隐患?
https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.7

1.2 可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

可见性案例分析

下面是模拟两个线程对共享变量操作的例子，用来分析线程间的可见性问题

```
1  public  class VisibilityTest {
2
3      private  boolean flag = true;
4
5      public void refresh() {
6          // 希望结束数据加载工作
```

```
        flag = false;
        System.out.println(Thread.currentThread().getName() + "修改flag:"+flag);
    }

    public void load() {
        System.out.println(Thread.currentThread().getName() + "开始执行.....");
        while (flag) {
            //TODO  业务逻辑：加载数据

        }
        System.out.println(Thread.currentThread().getName() + "数据加载完成，跳出循环");
    }


    public static void main(String[] args) throws InterruptedException {
        VisibilityTest test = new VisibilityTest();


        // 线程threadA模拟数据加载场景
        Thread threadA = new Thread(() -> test.load(), "threadA");
        threadA.start();

        // 让threadA先执行一会儿后再启动线程B
        Thread.sleep(1000);

        // 线程threadB通过修改flag控制threadA的执行时间，数据加载可以结束了
        Thread threadB = new Thread(() -> test.refresh(), "threadB");
        threadB.start();

    }

}
```

运行结果：threadA没有跳出循环，也就是说threadB对共享变量flag的更新操作对threadA不可见，存在可见性问题。

思考：**上面例子中为什么多线程对共享变量的操作存在可见性问题?**

如何保证可见性

- 通过 volatile 关键字保证可见性
- 通过 内存屏障保证可见性
- 通过 synchronized 关键字保证可见性
- 通过 Lock锁保证可见性

1.3 有序性

即程序执行的顺序按照代码的先后顺序执行。为了提升性能，编译器和处理器常常会对指令做重排序，所以存在有序性问题。

有序性案例分析

思考：下面的Java程序中x和y的最终结果是什么？

```
public class ReOrderTest {

    private static  int x = 0, y = 0;
    private  static  int a = 0, b = 0;

    public static void main(String[] args) throws InterruptedException {
        int i=0;
        while (true) {
            i++;
            x = 0;
            y = 0;
            a = 0;
            b = 0;

            /**
             *  x,y的值是多少:
             */
            Thread thread1 = new Thread(new Runnable() {
                @Override
                public void run() {
                    //用于调整两个线程的执行顺序
                    shortWait(20000);
                    a = 1;

```

```
                    x = b;
                }
            });
            Thread thread2 = new Thread(new Runnable() {
                @Override
                public void run() {
                    b = 1;
                    y = a;
                }
            });

            thread1.start();
            thread2.start();
            thread1.join();
            thread2.join();

            System.out.println("第" + i + "次（" + x + "," + y + "）");
            if (x==0&&y==0){
                break;
            }
        }
    }

    public static void shortWait(long interval){
        long start = System.nanoTime();
        long end;
        do{
            end = System.nanoTime();
        }while(start + interval >= end);
    }

}
```

执行结果：x,y出现了0，0的结果，程序终止。出现这种结果有可能是重排序导致的

如何保证有序性

- 通过 volatile 关键字保证有序性
- 通过 内存屏障保证有序性
- 通过 synchronized关键字保证有序性
- 通过Lock锁保证有序性

2. Java内存模型详解

在并发编程中，需要处理的两个关键问题：

1）多线程之间如何通信（线程之间以何种机制来交换数据）。

2）多线程之间如何同步 （控制不同线程间操作发生的相对顺序）。

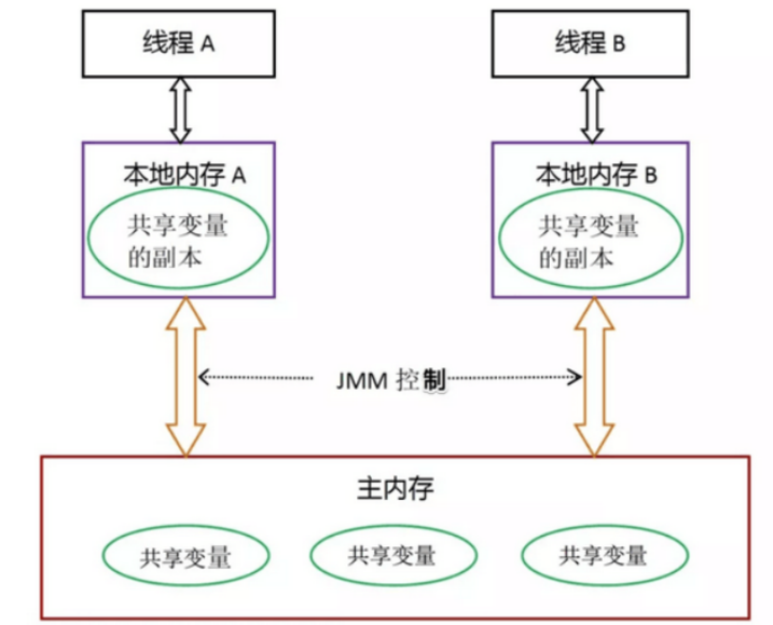
线程之间常用的通信机制有两种：共享内存和消息传递，Java采用的是共享内存模型。

2.1 Java内存模型的抽象结构

Java线程之间的通信由Java内存模型（Java Memory Model，简称JMM）控制，JMM决定一个线程对共享变量的写入何时对另一个线程可见。

从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存中，每个线程都有一个私有的本地内存，本地内存中存储了共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在，它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。

根据JMM的规定，**线程对共享变量的所有操作都必须在自己的本地内存中进行，不能直接从主内存中读取。**



从上图看，线程A和线程B之间要通信的话，必须经历以下两个步骤：

1）线程A把本地内存A中更新过的共享变量刷新到主内存中

2）线程B到主内存中去读取线程A之前已更新过的共享变量

所以，线程A无法直接访问线程B的工作内存，线程间通信必须经过主内存。**JMM通过控制主内存与每个线程的本地内存之间的交互，来为Java程序提供内存可见性的保证。**

> 温馨提醒： 面试期间，有同学会把Java内存模型误解为Java内存结构，然后答到堆，栈，GC垃圾回收，最后和面试官想问的问题相差甚远。实际上一般问到Java内存模型都是想问多线程，Java并发相关的问题。

主内存与工作内存交互协议

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间的实现细节，Java内存模型定义了以下八种**原子操作**来完成：

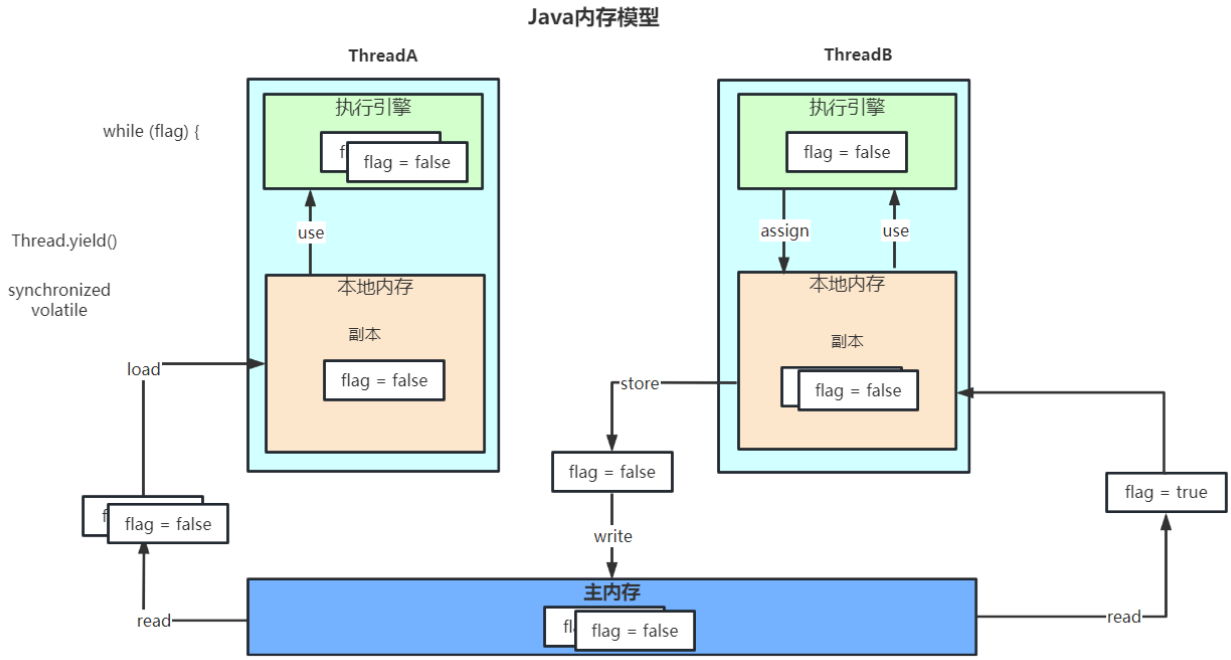
- lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。
- unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
- read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用
- load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。
- use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。
- assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。
- write（写入）：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中。

Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

- 如果要把一个变量从主内存中复制到工作内存，就需要按顺序地执行read和load操作， 如果把变量从工作内存中同步回主内存中，就要按顺序地执行store和write操作。但Java内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行。
- 不允许read和load、store和write操作之一单独出现
- 不允许一个线程丢弃它的最近assign的操作，即变量在工作内存中改变了之后必须同步到主内存中。
- 不允许一个线程无原因地（没有发生过任何assign操作）把数据从工作内存同步回主内存中。
- 一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量。即就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。
- 一个变量在同一时刻只允许一条线程对其进行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。lock和unlock必须成对出现
- 如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行load或assign操作初始化变量的值
- 如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量。
- 对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）。

可见性案例深入分析

本节课重点：结合可见性案例理解主内存和工作内存的交互过程



Java中可见性底层有两种实现：

1. 内存屏障 （synchronized Threed.sleep(10) volatile)

```
lock addl $0x0,(%rsp)
```

2. cup上下文切换 （Threed.yield() Threed.sleep(0) ）

2.2 锁的内存语义

锁获取和释放的内存语义：

- 当线程获取锁时，JMM会把该线程对应的本地内存置为无效。
- 当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中。

synchronized关键字的作用是确保多个线程访问共享资源时的互斥性和可见性。在获取锁之前，线程会将共享变量的最新值从主内存中读取到线程本地的缓存中，释放锁时会将修改后的共享变量的值刷新到主内存中，以保证可见性。

2.3 volatile内存语义

volatile写的内存语义：

当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内存。
volatile读的内存语义：
当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效，线程接下来将从主内存中读取共享变量。

volatile内存语义的实现原理

JMM属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。

volatile禁止重排序规则

为了实现volatile的内存语义，JMM会限制编译器重排序，JMM针对编译器制定了volatile重排序规则表。

由表中可以看出，**volatile禁止重排序场景**：
1. 当第二个操作是volatile写时，不管第一个操作是什么，都不能重排序。
2. 当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。
3. 当第一个操作是volatile写，第二个操作是volatile读时，不能重排序。

有序性案例深入分析

本节课重点：结合课上例子深入理解volatile禁止重排序的规则

案例：在Java多线程程序中，有时候需要采用延迟初始化来降低初始化类和创建对象的开销。双重检查锁定是常用的延迟初始化技术，但它有一个错误的用法。

```
1  public class Singleton {
2
3      private static Singleton singleton;
4      private Singleton() {
5      }
6      /**
7       * 双重检查锁定（Double-checked Locking）实现单例对象的延迟初始化
8       *
9       * @return
10      */
11     public static Singleton getSingleton() {
12         if (singleton == null) {
13             synchronized (Singleton.class) {
```

```
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }

}
```

正确的用法应该是使用volatile修饰singleton

```
 private volatile static Singleton singleton;
```

原因就在于singleton = new Singleton()这行代码，创建了一个对象。这行代码可以分解为三行伪代码

```
memory = allocate(); //1. 分配对象内存空间
ctorInstance(memory);  //2.初始化对象
instance = memory; //3.设置instance指向刚刚分配的内存地址
```

上面2和3之间可能会被重排序，重排序之后的执行时序如下：

```
memory = allocate(); //1. 分配对象内存空间
instance = memory; //3.设置instance指向刚刚分配的内存地址
                   //注意，此时对象还没有被初始化
ctorInstance(memory);  //2.初始化对象
```

JMM内存屏障插入策略

为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。**JMM内存屏障插入策略**：

1. 在每个volatile写操作的**前面**插入一个StoreStore屏障
2. 在每个volatile写操作的**后面**插入一个StoreLoad屏障
3. 在每个volatile读操作的**后面**插入一个LoadLoad屏障
4. 在每个volatile读操作的**后面**插入一个LoadStore屏障

上述内存屏障的插入策略非常保守，但它可以保证在任意处理器平台，任意程序中都能得到正确的volatile内存语义。

由于不同的处理器有不同的松紧度的处理器内存模型，内存屏障的插入还可以根据具体的处理器内存模型继续优化。以x86处理器为例，x86不会对读-读、读-写、写-写操作做重排序，因此在x86处理器中会省略这3类操作对应的内存屏障，仅会对写-读操作做重排序。

不同硬件实现内存屏障的方式不同，Java内存模型屏蔽了这种底层硬件平台的差异，由JVM来为不同的平台生成相应的机器码。

拓展：处理器级别内存屏障指令

拿X86处理器来说，有几种主要的内存屏障：

1. lfence，是一种Load Barrier 读屏障
2. sfence, 是一种Store Barrier 写屏障
3. mfence, 是一种全能型的屏障，具备lfence和sfence的能力
4. Lock前缀，Lock不是一种内存屏障，但是它能完成类似内存屏障的功能。Lock会对CPU总线和高速缓存加锁，可以理解为CPU指令级的一种锁。

内存屏障有两个能力：

1. 阻止屏障两边的指令重排序
2. 刷新处理器缓存

Hotspots源码中内存屏障的实现

orderAccess_linux_x86.inline.hpp

```
inline void OrderAccess::storeload()  { fence(); }
inline void OrderAccess::fence() {
  if (os::is_MP()) {
    // always use locked addl since mfence is sometimes expensive
#ifdef AMD64
    __asm__ volatile ("lock; addl $0,0(%%rsp)" : : : "cc", "memory");
#else
    __asm__ volatile ("lock; addl $0,0(%%esp)" : : : "cc", "memory");
#endif
  }
}
```

x86处理器中利用lock前缀指令实现类似内存屏障的效果。

> **lock前缀指令的作用**
> 1. 确保后续指令执行的原子性。在Pentium及之前的处理器中，带有lock前缀的指令在执行期间会锁住总线，使得其它处理器暂时无法通过总线访问内存，很显然，这个开销很大。在新的处理器中，Intel使用缓存锁定来保证指令执行的原子性，缓存锁定将大大降低lock前缀指令的执行开销。
> 2. LOCK前缀指令具有类似于内存屏障的功能，禁止该指令与前面和后面的读写指令重排序。
> 3. LOCK前缀指令会等待它之前所有的指令完成、并且所有缓冲的写操作写回内存(也就是将store buffer中的内容写入内存)之后才开始执行，并且根据缓存一致性协议，刷新store buffer的操作会导致其他cache中的副本失效

2.4 happens-before

happens-before的定义

JSR-133使用happens-before的概念来指定两个操作之间的执行顺序。由于这两个操作可以在一个线程之内，也可以在不同的线程之内。因此，JMM可以通过happens-before关系向程序员提供跨线程的内存可见性保证。

JSR-133规范对happens-before关系的定义如下：

1）如果一个操作happens-before 另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。 这是JMM对程序员的承诺， 注意，这只是JMM向程序员做出的保证。

2）两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行。如果重排序之后的执行结果，与按happens-before关系来执行的结果一致，那么这种排序并不非法，也就是说，JMM允许这种排序。这是JMM对编译器和处理器重排序的约束原则

JMM遵循一个基本原则：只要不改变程序的执行结果，编译器和处理器怎么优化都行。

- as-if-serial语义保证单线程内程序的执行结果不被改变
- happens-before关系保证正确同步的多线程程序的执行结果不被改变。

这么做的目的是为了在不改变程序执行结果的前提下，尽可能地提高程序执行的并行度。

happens-before规则

JSR-133规范定义了如下happens-before规则：

1）程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作；

2）锁定规则：对一个锁的解锁，happens-before于随后对这个锁的加锁；

3）volatile变量规则：对一个volatile变量的写操作，happens-before于任意后续对这个volatile变量的读操作；

4）传递规则：如果A happens-before B，并且B happens-before C，则A happens-before C；

5）线程启动规则：如果线程A调用线程B的start()方法来启动线程B，则start()操作happens-before于线程B中的任意操作；

6）线程中断规则：对线程interrupt()方法的调用happens-before于被中断线程的代码检测到中断事件的发生；

7）线程终结规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回；

8）对象终结规则：一个对象的初始化完成happens-before于它的finalize()方法的开始。

2.5 总结

Java中的volatile关键字可以保证多线程操作共享变量的可见性以及禁止指令重排序，synchronized关键字不仅保证可见性，同时也保证了原子性（互斥性）。在更底层，JMM通过内存屏障来实现内存的可见性以及禁止重排序。为了程序员的方便理解，提出了happens-before，它更加的简单易懂，从而避免了程序员为了理解内存可见性而去学习复杂的重排序规则以及这些规则的具体实现方法。