> 作者：JavaGieGie
>
> 微信公众号：Java开发零到壹

前言

假期结束，开始上班了，又是丧气满满的一天，还好有花Gie的文章陪伴，不然我会寂寞到死的（臭不要脸）。

多线程已经介绍了三篇多线程的很多知识点了，还没有看过的小伙伴记得花点时间看下，本系列都是从浅到深，逐步递进，不要想一口吃个胖子哦，小心虚胖（手动狗头）。

> 狗剩子：花GieGie~，来这么早。
>
> 我：文章还没写完呢，还有一堆小伙伴等着我呢，肯定来得早啊。
>
> 狗剩子：那你还在这蹲坑
>
> 我：.....

本章结束就要结束Java内存模型的讲解了，虽说结束，但是后面会一直贯穿在各个知识点中，多线程比较基础的知识点讲解的差不多了,本文正式引入了对volatile的分析，接下来的几章也会对`线程池`、`CAS`、`ThreadLocal`、`原子类`、`AQS`、`并发集合`等逐步讲解，看完这一系列，谁还能与你争锋（依旧狗头护体）。

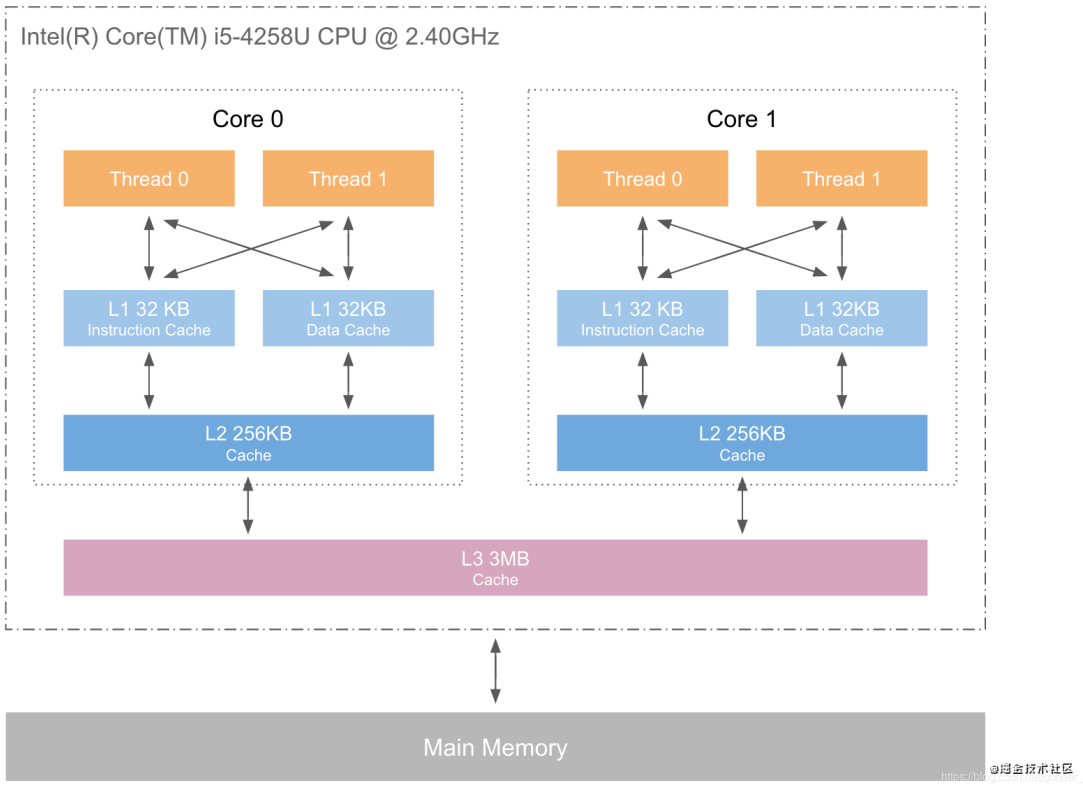
啪啪啪...看我如何打脸

正文

> **我：狗儿，你昨天提及了主内存和工作内存，上次介绍有点太糙了，今天可以再仔细说一下嘛**

好的呀，我都依你。

说到JMM主内存和工作内存，我们就必须要先了解一下`CPU缓存结构`。



> Core0、Core1代表两个核
>
> L1：每个核上有两个L1, 一个用于存数据（Data Cache），一个用于存指令（Instruction Cache）

先看上图，CPU存在三级缓存`L1`/`L2`/`L3`，你可能会想CPU是闲着没事干，设计那么多层内存干啥，直接从主存（物理内存）中读写数据他不香吗，这样就省事多了。但是我们要思考一下，CPU办事效率非常高，处理速度和物理内存相比不在同一个层级，如果CPU每次的读写都直接和主存交互，这样会大大降低指令的执行速递，这也就引出了这三级缓存。

```
a = a + 1
```

举个简单的栗子，当线程执行这个语句时，会先从主存当中读取`变量a`的值，拷贝到高速缓存中，然后CPU执行指令对`变量a`进行`加1`操作，并将数据写入缓存，最后将高速缓存中`变量a`修改后的值刷新到主存当中。

> **拓展**：线程在获取数据时首先会在最快的缓存中（L1）找数据，如果缓存没有命中(Cache miss)则往下一级找, 直到三级缓存都找不到时，那只有向内存（Main Memoy）查找数据了，未命中的次数越多，耗时也就越长。
>
> **我：那这个和我们JMM的内存结构有什么关系嘞?**

Java作为一门高级语言，屏蔽了这些底层细节，而是JMM定义了一套读写内存的规范。在JMM中，主内存和工作内存并非真正意义上的物理划分，而是JMM的一种抽象，它将`L1`、`L2`以及`寄存器`抽象成工作内存，每个处理器只能进行独享，而`L3`、`RAM`抽象成主内存，在处理器之间进行共享。

JMM关于主内存/工作内存的约束：

- 所有变量存储在主内存中，每个线程拥有自己的工作内存，工作内存中的变量是主内存中拷贝的**副本**；
- 线程不能直接操作主内存，只能通过修改本地内存，然后本地内存同步到主内存；
- 线程之间不能直接进行通信，只能通过主内存进行中转；
- 正是因为线程间这种通信方式，加上线程之间通信是有延时的，这也就导致了**可见性问题**。

> **我：我们有什么办法可以解决可见性问题吗?**

我们可以通过`happens-before`原则来解决可见性问题。

> **我：（草率了，居然没有听过）那...那可以说说这个具体是指什么吗？**

`happens-before`具体是指什么呢，我举个栗子：动作A发生在动作B之前，那动作B肯定能够看见动作A，这就是happens-before原则。

如果还是觉得很抽象，那我们再看个反例：两个线程（`线程1`、`线程2`），对于线程1执行的东西，线程2**有时可以看到但有时候又看不到**，这种情况下就不具备happens-before。这里看过花Gie蹲坑系列文章的小伙伴，应该可以想到上篇文章我们讲解过可见性的一个案例，出现第四种情况，`b=3，a=1`的情况时[《蹲坑也能进大厂》多线程系列-Java内存模型精讲]，正是因为不具备happens-before原则。

> **我：说了happens-before，那它都有哪些应用呢？**

这里先简单的列举一下，小伙伴们大致了解一下happens-before涉及到的范围有哪些就足够了,后面会对每一项单独进行讲解。

它的应用非常广，看下面这些分类，小伙伴们估计大部分都了解过的吧：

- **单线程原则：**

  > 单个线程中，按照程序的顺序，后面的操作一定可以看到前面的操作内容。

- **start()：**

  > 主线程A启动线程B，线程B中可以看到主线程启动B之前的操作。

- **join()：**

  > 主线程A等待子线程B完成，当子线程B执行完毕后，主线程A可以看到线程B的所有操作。

- **volatile**
- **synchronized、Lock**
- **工具类：**
  - 线程安全容器：例如CurreentHashMap
  - CountDownLatch
  - Semaphore
  - 线程池
  - Future
  - CyclicBarrier

`synchronized`、`线程池`等知识点，这里由于篇幅限制，后面都会一一讲解，逐步更新，有兴趣的小伙伴们可以关注一下（花Gie，今天的广告帮你打了，工资结一下吧）。

> **我：（老脸一粉）工资那个再说，你先说说volatile，我还等着去搬砖呢。**

首先`volatile`是一种同步机制，一旦一个共享变量（成员变量、静态成员变量）被`volatile`修饰之后，那么就具备了以下两个作用：

- 可见性：就是说一个线程修改了某个变量的值，其他线程能够立马感知到该变量已被修改；
- 禁止指令重排序。

- **对于可见性，我这里举个栗子：**

```
static boolean flag = true;

public static void main(String[] args) throws InterruptedException {
```

```
    //线程1
    new Thread(new Runnable() {
        @Override
        public void run() {
            while (flag){
                System.out.println("啥也不是！！！！");
            }
        }
    }).start();
    Thread.sleep(10);
    //线程2
    new Thread(new Runnable() {
        @Override
        public void run() {
            flag = false;
        }
    }).start();
}
```

这段代码用于停止一个线程，相信有不少小伙伴用到过，但是这并不是一个正确停止线程的方法，因为这里存在极小概率会停止线程失败。当`线程1`更改了`flag`变量后，还没来得及将内容回写到主存当中，就被安排做其他事情去了，此时`线程1`并不能感知到线程2已经对`flag`变量进行修改，因此会继续执行下去。

如果用`volatile`修饰flag变量，那就完全可以避免这种情况出现，**原因有以下几点**：

- 使用volatile关键字会强制将修改的值立即写入主存；
- 使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量flag的缓存行无效（也就是上文提到的CPU中L1或者L2缓存中对应的缓存行无效）；
- 由于线程1的工作内存中缓存变量flag的缓存行无效，所以线程1再次读取变量flag的值时会去主存读取。

因此`线程2`修改stop值时（修改`线程2`工作内存中的值，并将修改后的值写入内存），会使得`线程1`的工作内存中缓存变量stop的缓存行无效，然后`线程1`读取时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新后，去对应的主存读取最新的值。

- **禁止指令重排序**

前面章节我们提到编译器在解释代码时，实际执行的顺序和我们代码编写的顺序很可能是不同的，直白的说就是编译器只保证执行结果和你想要的一致，但至于先执行哪句代码、后执行哪一句代码，我说了算。但这里仅仅是在单线程下比较好用，一旦引入了多线程，就会出现各种奇怪的问题。

这里举个简单的栗子：

```
//a、b为非volatile变量
//flag为volatile变量

a = 2;         //语句1
b = 0;         //语句2
flag = true;   //语句3
c = 4;          //语句4
d = -1;        //语句5
```

由于`flag`变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。**但是**要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

并且`volatile`关键字能保证，执行到语句3时，语句1和语句2必定是执行完毕了的，且语句1和语句2的执行结果对语句3、语句4、语句5是可见的。

> **我：这么说我就懂了，那 a++ 问题是不是也能依赖volatile解决呢?**

这个咱们先看下面一段代码。

```
import java.util.concurrent.atomic.AtomicInteger;

public class volatileDemo implements Runnable {

   volatile int a;
   //只要知道该类在并发状态下进行自增或自减，是线程安全的即可
   AtomicInteger realCount = new AtomicInteger();

   public static void main(String[] args) throws InterruptedException {
      Runnable r =  new volatileDemo();
      Thread thread1 = new Thread(r);
      Thread thread2 = new Thread(r);
      thread1.start();
      thread2.start();
      thread1.join();
      thread2.join();
      //使用a++得到的结果
      System.out.println(((volatileDemo) r).a);
      //线程安全类得到的结果
      System.out.println(((volatileDemo) r).realCount.get());
   }
   @Override
   public void run() {
      for (int i = 0; i < 1000; i++) {
         a++;
         //realCount增加1
         realCount.incrementAndGet();
      }
   }
}
```

得到的结果为：

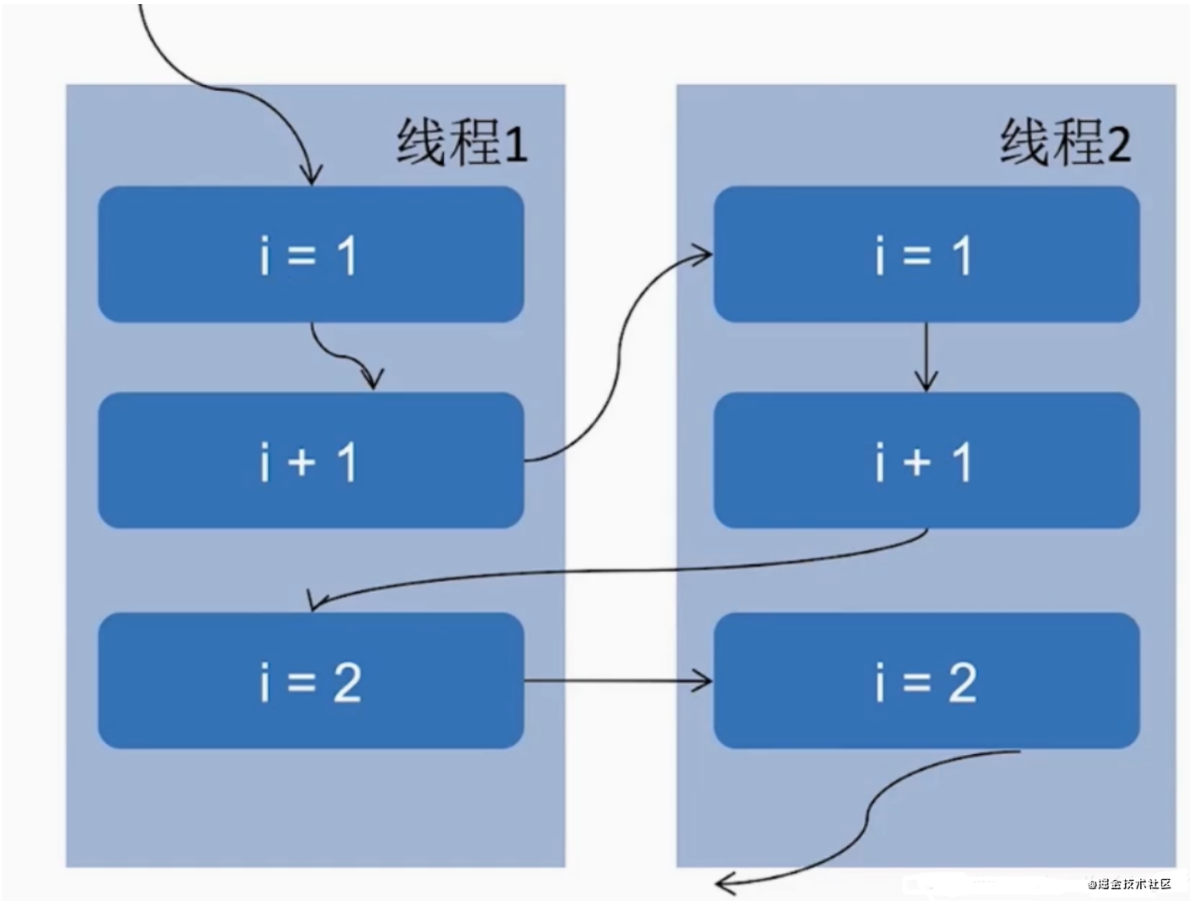
这个是为什么呢，我明明加上`volatile`了呀，你这个咋不好使了呢，骗子，退钱。

淡定.....先别急，我们先来扒开外表，深入的了解一下 `a++`，这其实并非原子操作，而是包含了几个步骤：读取a的值，进行加1操作，最后把加好的值赋值给a。

看到这里是不是就不奇怪了，因为`volatile`并不能保证**原子性**。

比如下面这个流程：

- 线程1读取到a的值并执行完+1动作（还未执最后的赋值）
- 此时另一个线程2也读取到a的值并执行+1动作；
- 线程1、线程2完成赋值，并将新值写回主存；
- 可以看出线程2用于计算的a值依旧为修改前的，所以等到线程2执行完毕后，a的值会少增加一次。



> 我：讲的很棒，必须加鸡腿，那你把volatile给小伙伴们总结一下吧?

总结下来有以下几点：

- volatile提供可见性。用于被多个线程共享的变量，保证被任意一个线程修改后，其他线程能立马获取到修改后的值；
- volatile不能替代`synchronized`，它不具备原子性和互斥性；
- volatile只作用于属性，可以禁止该属性被重排序；

- volatile提供了`happens-before`保证，也就是说对volatile变量进行修改后，其他线程都能获取到修改后的值。

总结

今天这章又和大家进一步探讨了JMM，你是不是对它也有了一个新的认识呢，除此之外我们还引入了新的知识点`volatile`，这个也是多线程中比较基础且非常常用的，非常有必要掌握，肝了一天，篇幅有点长，小伙伴们一定要耐下心看看。

下一章花Gie会继续介绍大家非常熟悉的`synchronized`，会不会和你认知的不一样呢，我们下一章见。**希望大家持续关注，为了大厂梦，我们继续肝。**

点关注，防走丢

以上就是本期全部内容，**如有纰漏之处，请留言指教，非常感谢。**我是花GieGie ，有问题大家随时留言讨论 ，我们下期见🐒。

> 文章持续更新，可以微信搜一搜「 **Java开发零到壹** 」第一时间阅读，后续会持续更新Java面试和各类知识点，有兴趣的小伙伴欢迎关注，一起学习，一起哈🐯🥃。

原创不易，你怎忍心白嫖，如果你觉得这篇文章对你有点用的话，感谢老铁为本文**点个赞、评论或转发一下**，因为这将是我输出更多优质文章的动力，感谢！