链滴

Java 内存模型及其原理

作者：vcjmhg

原文链接：https://ld246.com/article/1622037323573

来源网站：链滴

概述

在《"Java并发知识梳理"》这篇文章中曾说道：在并发领域由于**可见性问题**、**原子性问题**、**有序性问题**从而会导致并发场景下，结果的不确定性，为了解决**可见性**和**有序性**导致的问题，Java构建出了一套存模型。因而本文就主要谈一谈Java内存模型的设计思路以及其原理。

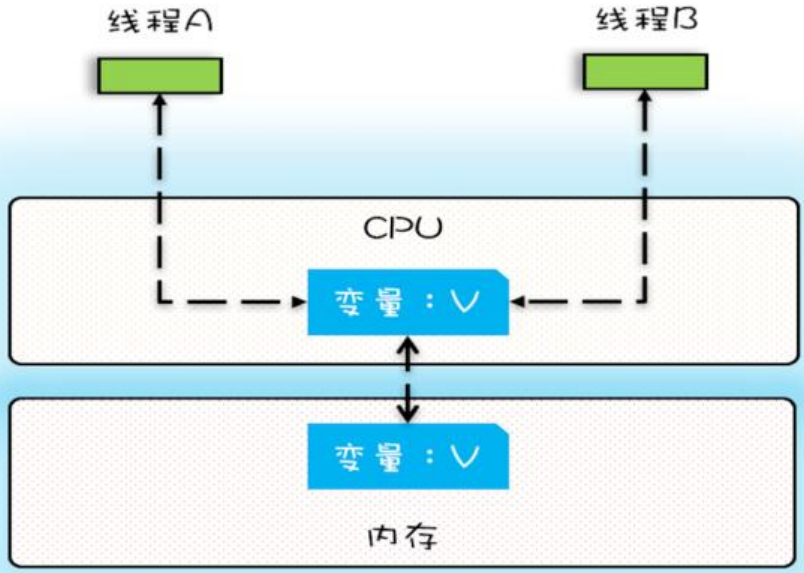
从三个问题说起

缓存导致可见性问题

什么是可见性呢?

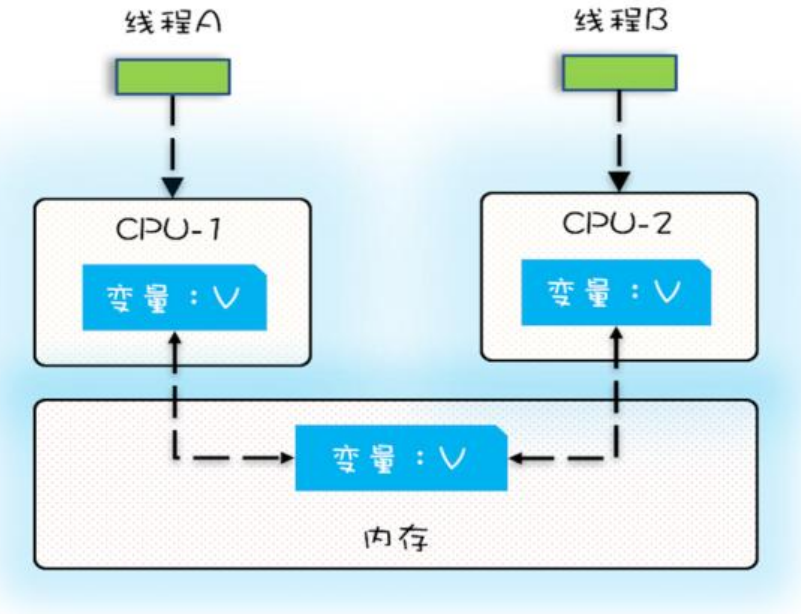
一个线程对共享变量的修改，另外一个线程能够立刻看到，我们称为可见性。

在单核时代，所有线程共用同一个CPU，CPU的缓存和主存一致性问题非常容易解决。例如下图：



由于两个线程操作的都是同一个CPU的缓存，因而线程A如果对变量V进行了修改，对线程B来说马上可以知晓，因而不存在可见性的问题。

但进入多核时代后，不同线程使用的可能是不同的CPU缓存，此时多个CPU缓存和主存之间的一致性题就没有那么容易解决。例如下图：



线程 A 操作的是 CPU-1 上的缓存，而线程 B 操作的是 CPU-2 上的缓存，很明显，这个时候线程 A 变量 V 的操作在没有同步到主存的情况下对变量B一定是不可见的。

我们以一段代码为例:

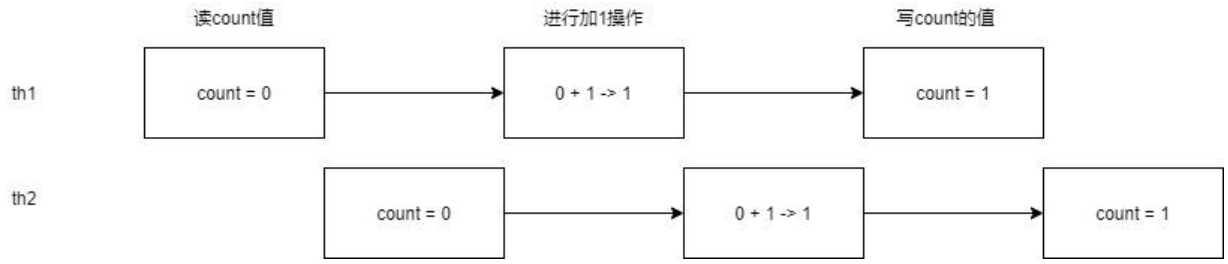
```
public class Test {
  private static long count = 0;
  private void add10K() {
    int idx = 0;
    while(idx++ < 10000) {
      count += 1;
    }
  }
  public static long calc() throws InterruptedException {
    final Test test = new Test();
    // 创建两个线程，执行add()操作
    Thread th1 = new Thread(()->{
      test.add10K();
    });
    Thread th2 = new Thread(()->{
      test.add10K();
    });
    // 启动两个线程
    th1.start();
    th2.start();
    // 等待两个线程执行结束
    th1.join();
    th2.join();
    return count;
  }

  public static void main(String[] args) throws InterruptedException {
    System.out.println(calc());
  }
}
```

直觉告诉我们应该是 20000，因为在单线程里调用两次 add10K() 方法，count 的值就是 20000，实际上 calc() 的执行结果是**个 10000 到 20000 之间的随机数。**我在本地测试了若干次结果如下：

```
第一次执行结果：10190
第二次执行结果：12223
第三册执行结果：10607
```

造成这样结果的原因是什么呢?

假设线程1（th1）和线程2（th2）同时启动，当第一次执行cout += 1操作时，可能会出现下边这种况：



在th1将count值读入自己CPU-1的缓存中，此时count值为0，此时th2也进行了读count值的操作，无疑问，此时count的值为0；而后th1进行了**加1操作**，count值变为1，此时由于该操作是在CPU-1缓存中进行，因而对与th2来说该操作是不可见的，因而th2在执行**加1操作**时，获取的count值仍然是，进行加法操作后，count值为1，接下来th1进行缓存和主存同步将count值改为1，而后th2进行缓

和主存同步将count值再次改为1。从而造成进行了两次count += 1操作，count的值仅仅增加1异常况发生。

然而想这样加1操作，我们共有**10000次**，因而难免会有若干次前面异常执行结果，最终就会造成最的结果值是位于10000到20000之间的随机数。

那怎么解决上边的问题呢?

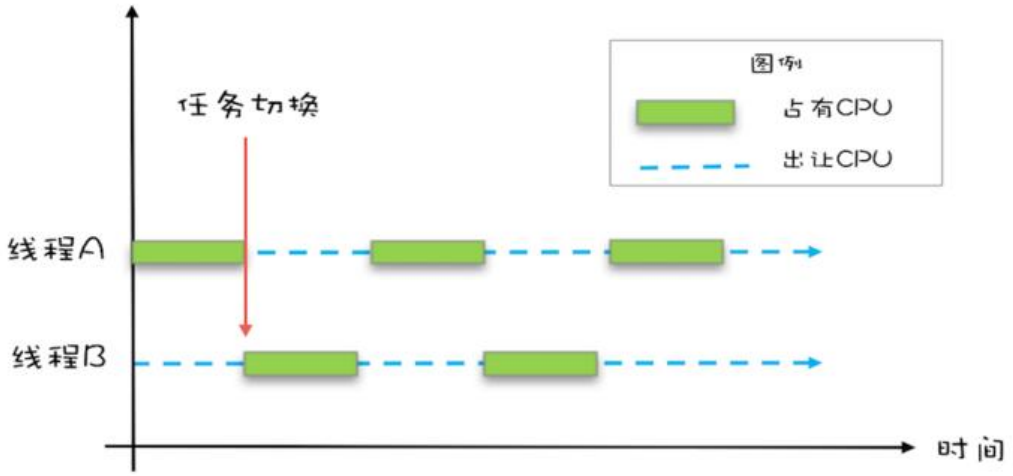
造成上述结果的原因主要就是缓存和主存的不一致造成的，那最暴力的方法当然可以通过禁用缓存来决，如果没有缓存那该问题自然就迎刃而解，具体来说可以使用volatile变量来修饰count变量，具体现和原理可以参考我的另一篇博客《"volatile的原理及应用"》

当然上述方法比较暴力，而且效率较低，后续会介绍一些其他方案，来高效的解决该问题。

线程切换带来的原子性问题

由于IO速度远远小于CPU处理速度，因而为了提高CPU的利用率，引入了多进程技术，使得即便是单环境下也可以一边听歌，一边上网。

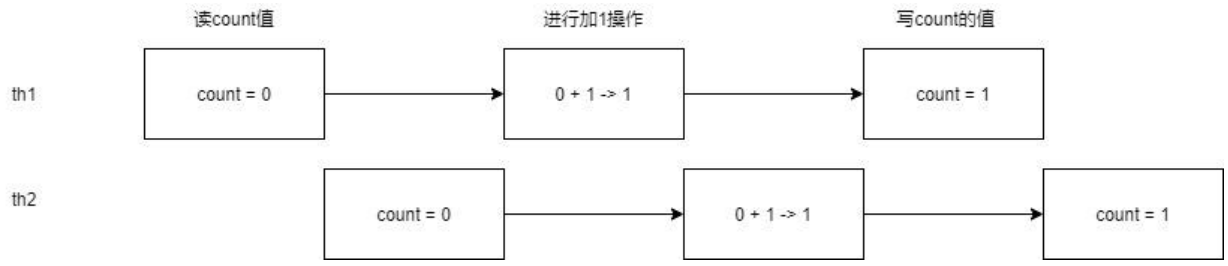
而单核情况下的多进程技术，本质上是通过**时间片轮转**来实现的，也就是说先允许一个线程运行一个间片的时间（比如50毫秒），然后切换另一个进程运行一个时间片的时间，循环往复，其示意图如下示:



同样的，多线程本质上也是通过**时间片轮转**技术来实现的，因而也涉及到任务切换。而任务切换大多情况下，是在时间片结束的时候发生，但Java语言某一行语句的执行时间可能不止一个时间片，比如边所提到的count += 1这条指令，就必须要三个CPU指令来完成:

1. 指令 1(第一个时间片)：首先，需要把变量 count 从内存加载到 CPU 的寄存器;
2. 指令2（第二个时间片）：在缓存中将count做加1操作
3. 指令2（第三个时间片）：将count的值同步到主存中

操作系统做任务切换，可以发生在任何一条 CPU 指令执行完，是的，是 CPU 指令，而不是高级语言的一条语句。对于上面的三条指令来说，我们假设 count=0，如果线程 A 在指令 1 执行完后做线程换，线程 A 和线程 B 按照下图的序列执行，那么我们会发现两个线程都执行了 count+=1 的操作，是得到的结果不是我们期望的 2，而是 1。



我们潜意识里面觉得 count+=1 这个操作是一个不可分割的整体，就像一个原子一样，但实际上它由三条指令组成，因而并非是原子操作，基于以上原因，我们在编写多线程程序时可能会出现错误。

编译优化带来的有序性问题

我们知道，在计算机中，CPU都是通过指令流的形式来执行指令的，为了提高指令流水的执行效率，编译阶段，编译器会对指令进行重排（指令重调度）（在不影响执行结果的情况下将不相关的两个指放在一起）。但这样也会造成一个问题，**代码执行的顺序可能并不会按照代码编写时的先后顺序来执行**某些情况下，顺序的调整可能并不会影响最后的执行结果比如：

```
a = 1;
b = 2;
```

调整后：

```
b = 2;
a = 1;
```

但在某些情况下，这种调整可能是致命的，会成为BUG的来源。

下边我们以双重检测锁实现单例设计模式为例，来说是说有序性可能会导致的问题,具体代码如下所示：

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

假设有两个线程 A、B 同时调用 getInstance() 方法，他们会同时发现 instance == null，于是同时对 Singleton.class 加锁，此时 JVM 保证只有一个线程能够加锁成功（假设是线程 A），另外一个线程会处于等待状态（假设是线程 B）；线程 A 会创建一个 Singleton 实例，之后释放锁，锁释放后，线 B 被唤醒，线程 B 再次尝试加锁，此时是可以加锁成功的，加锁成功后，线程 B 检查 instance == n ll 时会发现，已经创建过 Singleton 实例了，所以线程 B 不会再创建一个 Singleton 实例。

整个过程，看上去似乎很完美，但实际上这个getInstance()方法是存在问题的。问题在呢？主要是在**ew**操作上，我们想当然的认为new操作应该是这个样子的：

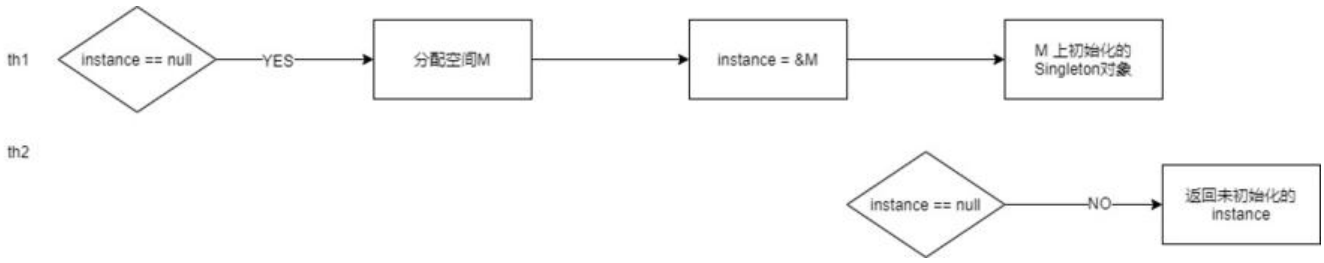
1. 分配一块空间M
2. 然后在M上初始化Singleton()对象
3. 将M地址赋值给instance变量

但实际的优化路径确实这样的：

1. 分配一块内存 M；
2. 将 M 的地址赋值给 instance 变量；
3. 最后在内存 M 上初始化 Singleton 对象。

但这样（**优化后**）就会带了一个问题：

我们假设第一个线程先执行getInstance()方法，首先对instance进行判断此时明显为空，而后进行instance对象的初始化操作，分配空间M、将M的地址赋值给变量instance。如果此时进行线程切换线程开始对instance进行判断，instance已经被赋了M的地址，因而此时为非空的，会直接返回instance但由于instance并未被初始化，此时访问 instance的成员变量就可能触发空指针异常。整个过程流程，如下图所示：



同样的要解决该问题也可以通过禁用指令重排来实现，具体可以参考《"volatile的原理及应用"》

内存模型

正如前面所说，由于缓存导致了可见性的问题，由于编译优化带来了有序性的问题，而解决这两个问的最有效方案就是**禁用缓存和编译优化。** 这样虽然说解决了问题，但随之带来的是效率和性能的降低因而最合理的方案应该是**按需禁用缓存和编译优化。**

那么，如何做到“按需禁用”呢？对于并发程序，何时禁用缓存以及编译优化只有程序员知道，那所“按需禁用”其实就是指按照程序员的要求来禁用。所以，为了解决可见性和有序性问题，**只需要提给程序员按需禁用缓存和编译优化的方法**即可。

Java内存模型是一个比较复杂的概念，从不同的角度看，它有不同的内容：

从内存功能划分角度看，它由主内存和工作内存的组成以及内存间的交互操作两部分构成。从程序员程角度看，Java 内存模型规范了 JVM 如何提供按需禁用缓存和编译优化的方法。具体来说，这些方包括 volatile、synchronized 和 final 三个关键字，以及六项 Happens-Before 规则。

- 内存模型
  - 内存划分角度
    - 主内存与工作内存的划分
    - 内存间的交互操作
  - 从程序员使用角度

- volatile
- synchronized
- final
- 六项Happens-Before规则

首先我们了解一下Java内存模型中主内存和工作内存的划分问题：

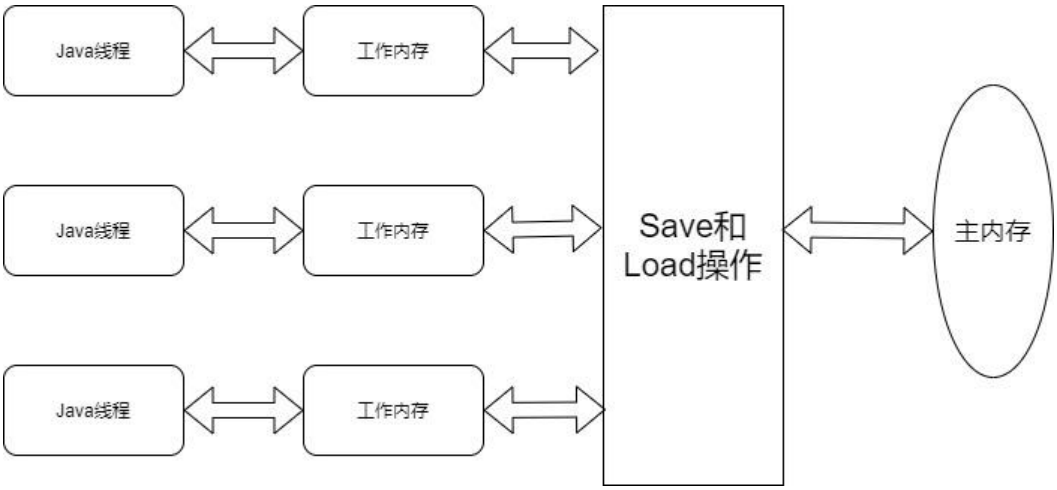
主内存和工作内存的划分

首先，我们思考一个问题，为何需要重新定义一种内存模型，而不是用操作系统本身具有的内存模型？

问题其实也很好回答，不同操作系统在实现线程时，可能方案不同的，硬件细节可能也是有差异的，Java本身作为跨平台语言，必须有一套统一的内存模型，来屏蔽各种硬件和操作系统的内存访问差异让Java程序在各个平台下都能够达到一致的内存访问效果。

从内存角度来看，Java内存的**主要目的**是**定义程序中各种变量的访问规则**，即关注虚拟机中把变量值储到内存和从内存中去除变量值这样的细节。

具体来说，Java内存模型规定了所有的变量都存储在主内存（Main Memory）中（此处的主内存跟绍物理硬件时提到的主内存的名字是一样的，两者也可以类比，但从物理层面看，它属于虚拟机内存一部分）。每条线程都有自己的工作内存（Working Memory）。线程中的工作内存保存了该线程用的变量的主内存副本，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中数据。不同线程之间也无法直接访问对方工作内存中的变量，**不同线程变量值的传递都是通过主内存完成。**线程、主内存和工作内存三者之间的关系如下图所示：



在明白了工作内存和主内存的关系之后，接下来我们需要搞清楚的是工作内存和主内存之间的一个同的问题即**内存之间的交互操作。**

内存间的交互操作

关于主内存和工作内存之间的交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存步回主内存这一类的实现细节，Java内存模型定义了8种操作来完成。Java虚拟机在实现这8中操作时保证每一种操作都是**原子的、不可拆分的。**

- **lock**（锁定）：作用于主内存变量，把一个变量标识为线程独占的状态。
- **unlock**（解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才以被其他线程锁定。

● **read**（读取）：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便续load动作使用

● **load**（载入）：作用于工作内存的变量，把read操作从主内存中得到的**变量值**放到工作内存的**变量本中**。

● **use**（使用）：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟遇到一个**使用变量的值的指令**时将执行这个操作。

● **assign**（赋值）：作用于工作内存的变量，它把一个执行引擎接受的值赋给工作内存的变量，每当拟机遇到**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，它把工作内存中一个变量的值传送到主内存中，便于随后write作使用。

● **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到主内存的量中。

如果要把一个变量从内存中拷贝到工作内存中则需要顺序的执行read和load操作，如果要把变量从工内存中同步回主内存，则需要按顺序执行store和write操作。但需要注意的是上边的操作Java虚拟机求必须是**顺序的**，但并**不要求是连续的。** 也就是说read之后的下一个操作必须是load，但它们之间以存在一定的间隔时间。

为了保证操作正确性，针对上边的八种操作，定义了如果规则：

比如：

● 不允许read和load、sotre和write操作单独出现，即不允许一个变量从主内存中读取了但工作内存接受，或者工作内存发起回写，但主内存不接受的情况发生。

● 不允许一个线程丢弃它最近的assign操作，即变量在工作内存中改变了之后必须把该变化的同步回内存。

● 不允许一个线程无原因的把数据同步回工作内存。

● .....

规则一共有8条，比较复杂感兴趣可以参考周志华老师的《深入理解JVM虚拟机》第三版，443页上的详细描述，此处不再详述。

因为上边的整个定义是比较琐碎的，就开发者来说是没法直接使用的，它更多面向的是虚拟机开发人来看，因而就程序员使用而言，我们更加关注的点是**三个关键字**和六项**Happen-Before**规则。

三个关键字

volatile

volatile是Java虚拟机提供的最轻量级的同步机制，当一个变量被定义为volatile之后，它将具有两个性：

第一项是**保证此变量对所有线程的可见性**，这里的**看见行**是指一个线程修改了这个变量的值，新值对他线程来说是立即可知的。而普通变量并不能做到这一点，普通变量的值在线程传递时均需要通过主存来完成。比如，线程A修改了一个普通变量的值，然后向主内存中进行回写，另一条线程B在线程A写完成之后再对主内存进行读取操作，新变量值才会对线程B可见。

另一项作用**禁止指令重排序优化**，普通的变量仅会保证在该方法的执行过程中所有依赖赋值的结果的方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码的执行顺序一致。

关于volatile的具体使用和原理可以参考另一篇博客《"volatile的原理及应用" 》

synchronized

Java 语言提供的 synchronized关键字，就是锁的一种实现。synchronized关键字可以用来修饰方法也可以用来修饰代码块，通过对代码加上synchronized修饰，可以保证其原子操作解决原子性问题。

关于其具体原理可以参考：《"synchronized的原理及应用"》

final

final关键字就比较简单，final 修饰变量时，初衷是告诉编译器：**这个变量生而不变，可以可劲儿优**。

因而被final修饰的关键字是绝对线程安全，不会有线程安全的问题。

Happen-Before规则

如何理解 Happens-Before 呢?

Happens-Before 并不是说前面一个操作发生在后续操作的前面，它真正要表达的是：**前面一个操作结果对后续操作是可见的。**就像有心灵感应的两个人，虽然远隔千里，一个人心之所想，另一个人都得到。Happens-Before 规则就是要保证线程之间的这种“心灵感应”。所以比较正式的说法是：**Happens-Before 约束了编译器的优化行为，虽允许编译器优化，但是要求编译器优化后一定遵守 Happns-Before 规则。**

Happen-Before规则主要有六项，主要包含以下内容：

- Happen-Before规则
  - 程序的顺序性规则
  - volatile 变量规则
  - 传递性
  - 管程中锁的规则
  - 线程 start() 规则
  - 线程 join() 规则

1. 程序的顺序性规则

这条规则是指在一个线程中，按照程序顺序，**前面的操作 Happens-Before 于后续的任意操作。**这规则比较容易容易理解，我们以下边一个代码为例：

```
class VolatileExample {
  int x = 0;
  volatile boolean v = false;
  public void writer() {
    x = 42;
    v = true;
  }
  public void reader() {
    if (v == true) {
      // 这里x会是多少呢?
    }
```

```
  }
}
```

第六行代码x = 42 对于第七行代码v = true是可见的。

这也比较符合单线程里面的思维：**程序前面对某个变量的修改一定是对后续操作可见的。**

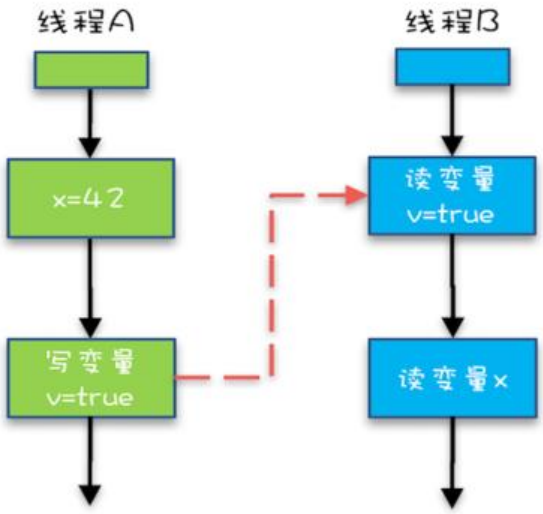
2. volatile变量规则

这条规则是指对一个 volatile 变量的**写操作**， Happens-Before 于后续对这个 volatile 变量的**读操作**
这个其实就是有点像禁用缓存的意思，似乎跟JDK1.5中的定义没有啥区别，但是和后序传递性相结合话，就会有不同的含义。

3. 传递性

这条规则是指如果**A Happens-Before B**，且 **B Happens-Before C**，那么**A Happens-Before C**
这个就比较容易理解，很像是数学中学习的传递性。

我们将该条规则与规则"1. 程序的顺序性规则"、 规则"2. volatile变量规则" 结合一下会发生什么呢uestion question



从图中，我们可以看到：

1. 根据规则"1. 程序的顺序性规则"， x = 42 Happen-Before于 写变量v = true。

2. 由于x由 volatile关键字修饰，根据规则"2. volatile变量规则"，写变量v = true先发生于读变量v = ture

3. 根据规则"3. 传递性"，我们可以推知 x = 42Happen-Before于读变量v==true。

而这就意味对于线程B执行方法reader()，运行到 v == true时，对于x的结果一定会是42。这就是 1.5 版本对 volatile语义的增强，这个增强意义重大，1.5 版本的并发工具包（java.util.concurrent）就

靠 volatile 语义来搞定可见性的，这个在后面的内容中会详细介绍。

4. 管程中锁的规则

这条规则是指，对一个锁的**解锁操作**Happen-Before于后续**对这个变量的加锁操**作。

在了解这条规则之前，我们要首先了解一下管程，什么是管程呢?

简单来说**管程**就是一种**同步原语**，在Java中synchronized就是管程的一种实现。

管程的锁在Java中是隐式的，例如下边代码，进入代码块时，会自动加锁，出代码块时会自动解锁：

```
synchronized (this) { //此处自动加锁
  // x是共享变量,初始值=10
  if (this.x < 12) {
    this.x = 12;
  }
} //此处自动解锁
```

结合规则4，当两个线程A和B都执行该代码块时，线程A进入代码块后，将x的值由原来的10改成12
在此过程中B如果执行到该代码块时，线程A加的锁对线程B是可见的，因而线程B会先进入阻塞队列
等到线程A执行完之后，线程B继续执行，因为线程A的执行结果对想成B可见，因而此时x的值已经是1
了。

5. 线程start()规则

这条是关于线程启动的。**它是指主线程 A 启动子线程 B 后，子线程 B 能够看到主线程在启动子线程 B 前的操作。**

也就是说，主线程在启动子线程之前的操作Happen-Before于线程B中的任意操作。以代码为例：

```
Thread B = new Thread(()->{
  // 主线程调用B.start()之前
  // 所有对共享变量的修改，此处皆可见
  // 此例中，var==77
});
// 此处对共享变量var修改
var = 77;
// 主线程启动子线程
B.start();
```

变量var = 77在线程B的代码主体中都是可见的。

6. 线程join()规则

这条是关于线程等待的。它是指主线程 A 等待子线程 B 完成（主线程 A 通过调用子线程 B 的 join() 法实现），当子线程 B 完成后（主线程 A 中 join() 方法返回），主线程能够看到子线程的操作。当所谓的“看到”，指的是对共享变量的操作。

换句话就是说，如果在线程 A 中，调用线程 B 的 join()并成功返回，那么线程 B 中的任意操作 Happ ns-Before 于该join()操作的返回。以代码为例：

```
int var;
```

```
Thread B = new Thread(()->{
  // 此处对共享变量var修改
  var = 66;
});
// 例如此处对共享变量修改，
// 则这个修改结果对线程B可见
// 主线程启动子线程
B.start();
B.join()
// 子线程所有对共享变量的修改
// 在主线程调用B.join()之后皆可见
// 此例中，var==66
```

在主线程执行完B的join()方法之后，对于变量var = 66这一结果均是可见的。

总结

本文主要讲了Java内存模型的原理及其构成，Java内存模型按照不同角度的划分，可以看到不同的内，从JVM设计者的角度看，我们可以看到"主内存和工作内存的划分"以及"内存间的交互操作"。从程员角度看，我们能够看到三个常用的关键字：volatile、synchronized以及final，并且简单介绍了这三个关键字的作用，后续也会分别写出对应详细的文章，从原理角度进行阐述volatile和synchronize的原理以及使用。同时，对于Happens-Before 规则我们也有详细的介绍，分成了六部分就行了详细阐述。

参考

1. 《深入理解JVM虚拟机》
2. 《Java并发变成实战》
3. 《Java并发编程实战》