JMM

一、为什么需要JMM

我们知道C语言出现的时间是早于java的，因此C是没有类似于JMM这种机制的，这样会导致什么问题呢?

C语言是依赖于处理器的，没有一套JMM这样的规范，就导致了不同处理器，处理多线程的结果可能是不一样的。这是因为很多情况下CPU会进行乱序，所以就没有办法保证并发的安全。

在这种情况下，迫切需要一个标准，让多线程运行的结果可预期。所以说JMM本质其实就是一组规范。

有了这样的规范后，无论在什么处理器上运行，无论该处理器是支持缓存一致性还是不支持，我们都可以通过这样的内存模型抽象出来的这个规范，来做到包括我们普通java开发者、编译器开发者、JVM工程师、CPU工程师，这一系列环节达成一个统一。

二、什么是JMM

JMM实际上是一种规范，而且是一组规范，需要各个JVM的实现来遵守MM规范,以便于开发者可以利用这些规范,更方便地开发多线程程序

JMM是工具类和关键字的原理

如果没有JMM,那就需要我们自己指定什么时候用内存栅栏（工作内存和主内存之间的拷贝和同步）等,那是相当麻烦的,幸好有了JMM,让我们只需要用同步工具和关键字就可以开发并发程序。

最重要的三点内容：可见性、重排序、原子性

三、重排序

知识概览：

- 重排序的代码案例、什么是重排序
- 重排序的好处:提高处理速度
- 重排序的3种情况:编译器优化、CPU指令重排、内存的 "重排序"

重排序的代码案例：

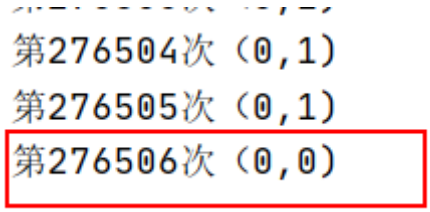
```
1  package jmm;
2
3  import java.util.concurrent.CountDownLatch;
4
5  /**
6   * 描述:       演示重排序的现象 “直到达到某个条件才停止”，测试小概率事件
7   */
8  public class OutOfOrderExecution {
9
10     private static int x = 0, y = 0;
11     private static int a = 0, b = 0;
12
13     public static void main(String[] args) throws InterruptedException {
14         int i = 0;
15         for (; ; ) {
```

```
            i++;
            x = 0;
            y = 0;
            a = 0;
            b = 0;

            // 该工具类可以让两个线程同时开始执行
            CountDownLatch latch = new CountDownLatch(3);

            Thread one = new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        latch.countDown();
                        latch.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    a = 1;
                    x = b;
                }
            });
            Thread two = new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        latch.countDown();
                        latch.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    b = 1;
                    y = a;
                }
            });
            two.start();
            one.start();
            latch.countDown();
            one.join();
            two.join();

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


}
```

如果不考虑重排序，上面代码之可能会出现三种结果（x, y） = （0, 1）(1, 0) (1, 1)

如果出现了（0, 0）即两个线程先执行了x = a, y =a 两行代码，即发生了重排序

结果：



重排序的三种情况：

1. 编译器优化

   编译器(**包括JVM,JIT编译器**等)出于优化的目的(例如当前有了数据a,那么如果把对a的操作放到一起效率会更高,避免了读取b后又返回来重新读取a的时间开销),在编译的过程中会进行一定程度的重排,导致生成的机器指令和之前的字节码的顺序不一致。

   在刚才的例子中,编译器将y=a和b=1这两行语句换了顺序(也可能是线程2的两行换了顺序,同理),因为它们之间没有数据依赖关系,那就不难得到x=0,y=0这种结果了。

2. 指令重排序

   CPU的优化行为,和编译器优化很类似,是通过乱序执行的技术,来提高执行放率。所以就算编译器不发生重排,CPU也可能对指令进行重排,所以我们开发中,一定要考虑到重排序带来的后果。

3. 内存的"重排序" 内存系统内不存在重排序,但是内存会带来看上去和重排序一样的效果,所以这里的重排序打了双引号。由于内存有缓存的存在,在JMM里表现为主存和本地内存,**由于主存和本地内存的不一致**,会使得程序表现出乱序的行为。

   在刚才的例子中,假设没编译器重排和指令重排,但是如果发生了内存缓存不致,也可能导致同样的情况:线程1修改了a的值,但是修改后并没有写回主存,所以线程2是看不到刚才线程1对a的修改的,所以线程2看到a还是等于0。同理,线程2对b的赋值操作也可能由于没及时写回主存,导致线程1看不到刚才线程2的修改。

四、可见性

知识概览

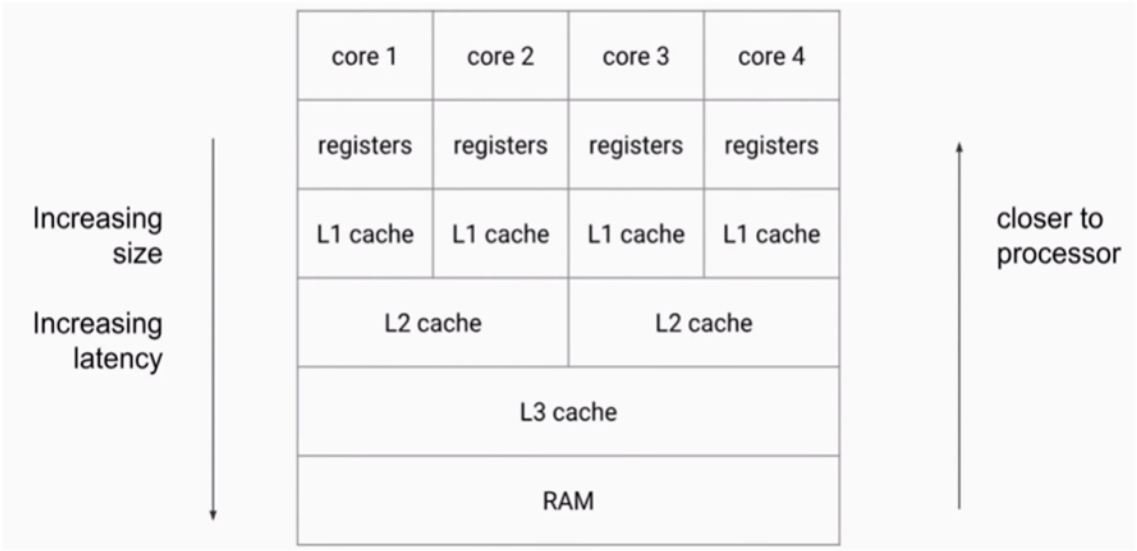
1. 案例:演示什么是可见性问题
2. 为什么会有可见性问题
3. JMM的抽象:主内存和本地内存
4. **Happens - Before**原则（重点）
5. **volatile**关键字，与synchronized关键字对比
6. 能保证可见性的**措施**
7. 升华:对 **synchronized可见性**的正确理解

案例:演示什么是可见性问题

```
1  package jmm;
2
3  /**
4   * 描述：        演示可见性带来的问题
5   */
6  public class FieldVisibility {
7
8      /*volatile */int a = 1;
9      /*volatile */int b = 2;
10
```

```
    private void change() {
        a = 3;
        b = a;
    }


    private void print() {
        System.out.println("b=" + b + ";a=" + a);
    }

    public static void main(String[] args) {
        while (true) {
            FieldVisibility test = new FieldVisibility();
            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        Thread.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    test.change();
                }
            }).start();

            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        Thread.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    test.print();
                }
            }).start();
        }

    }


}
```

为什么会有可见性问题



RAM：主内存

CPU有多级缓存,导致读的数据过期

- 高速缓存的容量比主内存小,但是速度仅次于寄存器,所以在CPU和主内存之间就多了 **Cache层**
- 线程间的对于共享变量的可见性问题不是直接由多核引起的,而是由**多缓存引起**的。
- 如果所有个核心**都只用一个缓存**,那么也就**不存在内存可见性**问题
- 每个核心都会将自己需要的数据读到**独占缓存**中,数据修改后也是写入到缓存中,然后**等待刷入到主存**中。所以会导致有些核心读取的值是一个**过期的值**。

五、JMM的抽象:主内存和本地内存

主内存和工作内存的关系

1. 所有的变量都存储在主内存中,同时每个线程也有自己独立的工作内存,工作内存中的变量内容是主内存中的拷贝
2. 线程不能直接读写主内存中的变量,而是只能操作自己工作内存中的变量,然后再同步到主内存中
3. 主内存是多个线程共享的,但线程间不共享工作内存如果线程间需要通信,必须借助主内存中转来完成

> 所有的共享变量存在于主内存中,每个线程有自己的本地内存,而目线程读写共享数据也是**通过本地内存交换**的,所以才导致了可见性问题。

六、Happens-before

什么是happens-before

两种解释一个意思

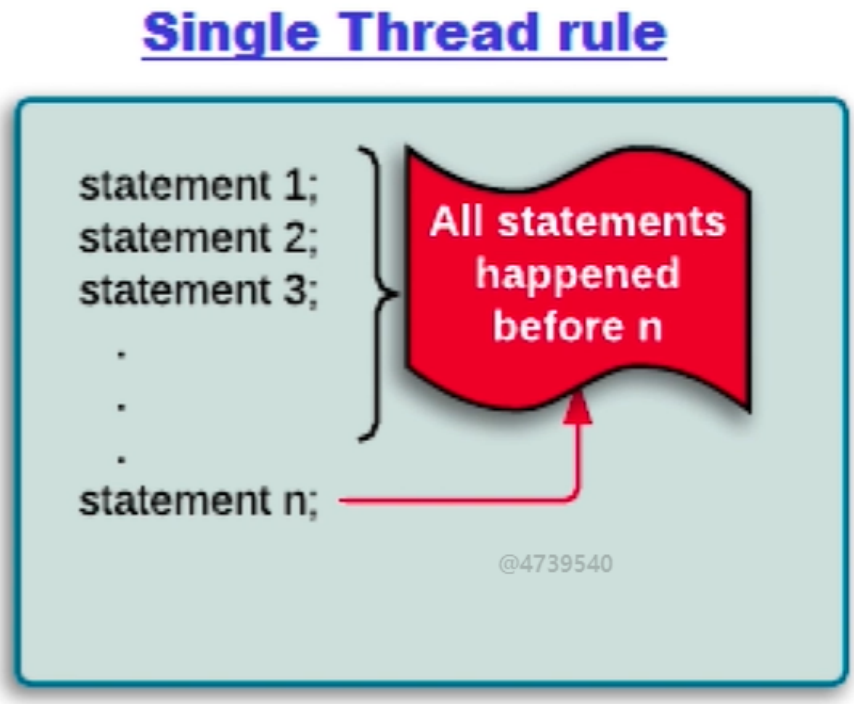
解释1：happens- before规则是用来解决可见性问题的:在时间上,动作A发生在动作B之前,B保证能看见A,这就是 happens- before。 解释2：两个操作可以用 happens- before来确定它们的执行顺序:如果一个操作 happens- before于另一个操作,那么我们说第一个操作对于第二个操作是可见的。

什么不是happens-before

两个线程没有相互配合的机制,所以代码X和Y的执行结果并不能保证总被对方看到的,这就不具备 happens- before。

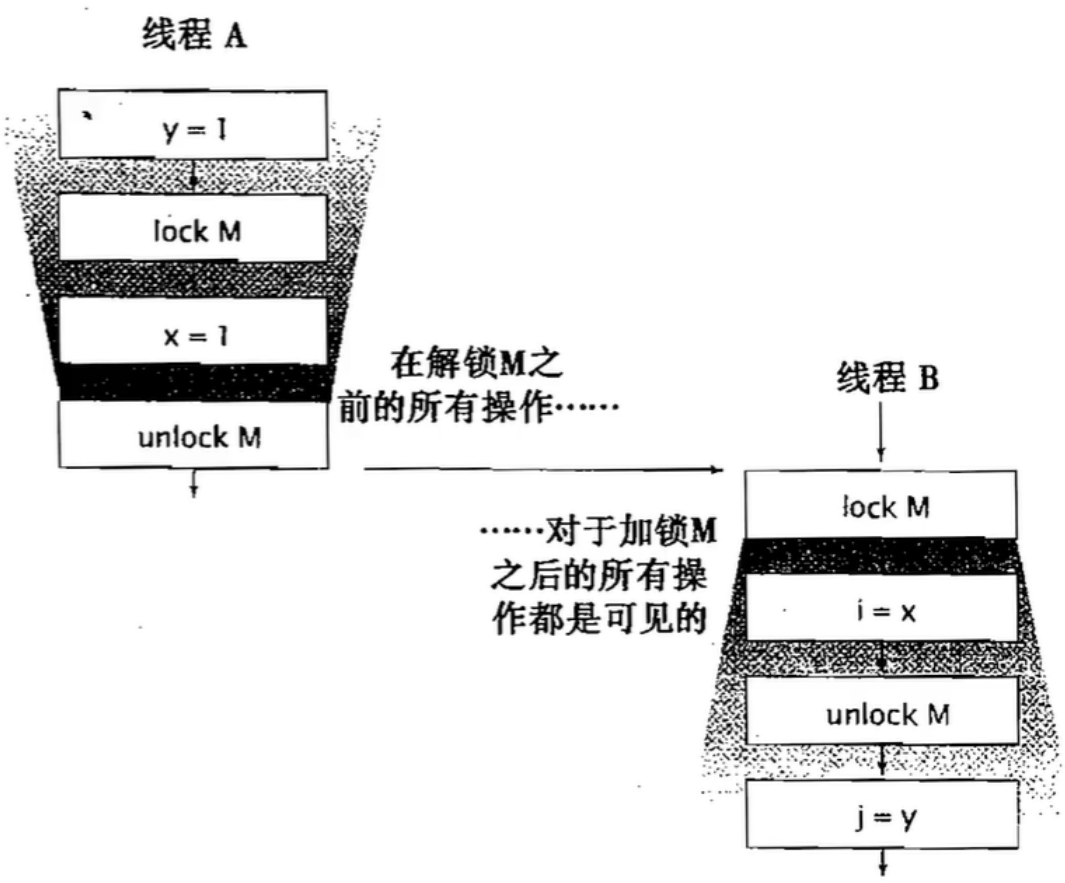
Happens-Before规则有哪些

1. 单线程原则

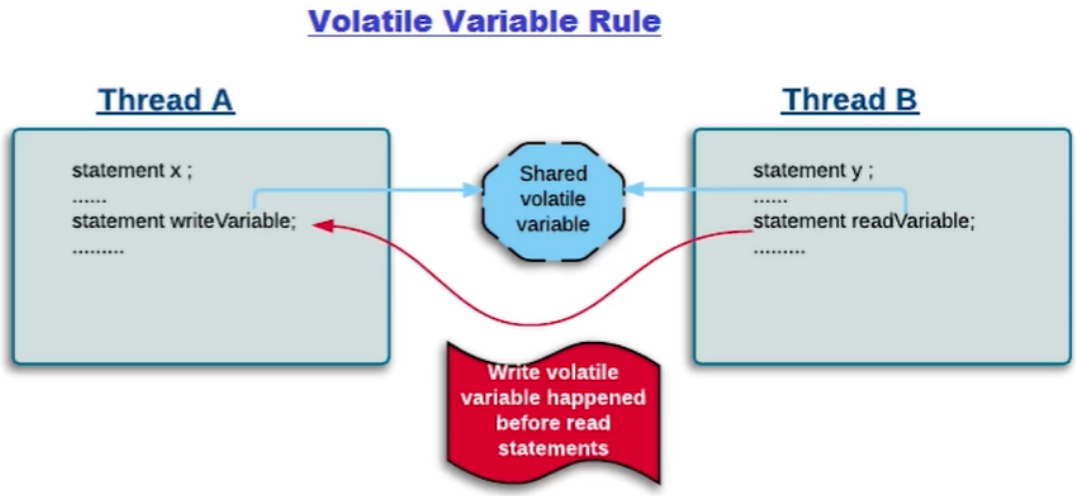


在一个线程内，后面的语句一定能看到前面的语句做了什么

2. **锁操作(synchronized和Lock)**

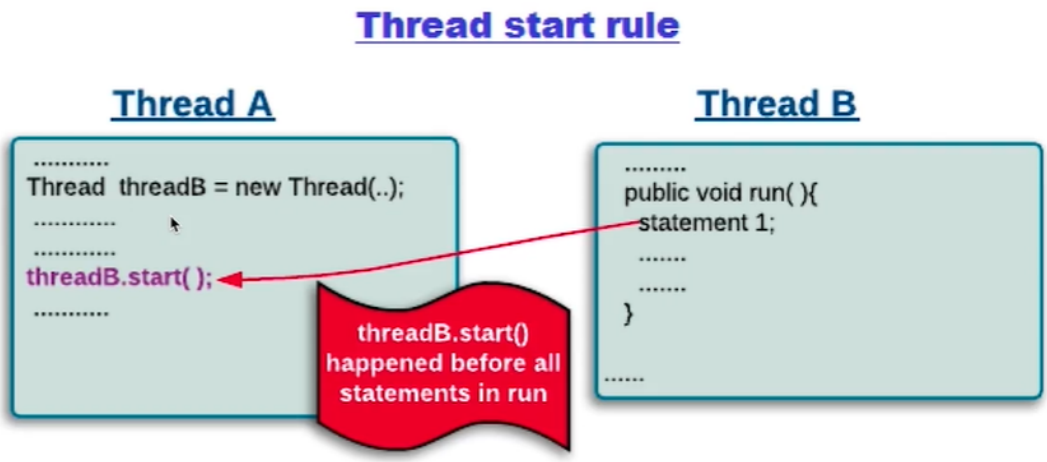


3. **volatile变量**



volatile修饰后，不同线程对这个变量就是可见的。

4. 线程启动



说明：子线程启动后，子线程都能看到子线程启动之前的主线程所以语句的修改。

5. 线程join
6. 传递性
7. 中断：一个线程被其他线程 Interrupt是,那么检测中断(isInterrupted)或者抛出InterruptedException一定能看到。
8. 构造方:对象构造方法的最后一行指令 happens- before于finalize() 方法的第一行指令
9. 工具类的 Happens- Before原贝
    1. 线程安全的容器get定能看到在此之前的put等存入动作
    2. CountdownLatch
    3. Semaphore
    4. Future
    5. 线程池
    6. Cyclicbarrier

优质代码案例：happens-before演示

> happens- before?有一个原则是:如果A是对 volatile变量的写操作,B是对同一个变量的读操作,那么hb(A,B)。

近朱者赤:给b加了 **volatile**,不仅b被影响,也可以实现轻量级同步，b之前的写入(对应代码b=a)对读取b后的代码(print b)都可见, 所以在 writer Thread里对a的斌值,一定会对 reader Thread里的读取可见,所以这里的a即使不加 volatile,只要b读到是3,就可以由 happens- before原则保证了读取到的都是3而不可能读取到1。

代码参考上面的可见性代码：FieldVisibility

七、volatile关键字

volatile是什么

- volatile是一种**同步机制**,比 synchronized或者Lock相关类**更轻量**,因为使用 volatile并**不会发生上下文切换**等开销很大的行为。
- 如果一个变量别修饰成 volatile,那么JVM就知道了这个变量**可能会被并发修改。**
- 但是开销小,相应的能力也小,虽然说 volatile.是用来同步的保证线程安全的,但是 **volatile做不到synchronized那样的原子保护**, volatile仅在很有限的场景下才能发挥作用。

Volatile 的实现原理

见：https://www.infoq.cn/article/ftf-java-volatile/

volatile的使用场景

不适用：a++

适用场合1: boolean flag，如果一个共享变量自始至终只被各个线程赋值,而没有其他的操作，那么就可以用 volatile来代替synchronized或者代替原子变量,因为赋值自身是有原子性的,而volatile又保证了可见性,所以就足以保证线程安全。

使用场合2：作为触发器来使用，一般设置一个标志，如boolean flag = true，线程A在boolean flag设置为true前进行了一系列的操作，而线程B的操作又必须依赖于线程A boolean flag = true之前的一系列操作，这时候就用到了volatile ，使用volatile修饰flag后就可以保证当flag为true的时候，前面一系列操作对应线程B是可见的（Happens-Before的）

如下图所示

```
Map configOptions;
char[] configText;
volatile boolean initialized = false;

// Thread A
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;


// Thread B
while (!initialized)
  sleep();
// use configOptions
```

volatile的作用：

- 可见性：读一个 volatile变量之前,需要先**使相应的本地缓存失效**,这样就必须**到主内存读取最新值**,写一个 volatile属性会立即刷入到主内存。
- 禁止重排序：解决单例模式双重锁乱序问题

volatile和synchronized关系

volatile在这方面可以看做是轻量版的 synchronized：如果一个共享变量自始至终只被各个线程赋值,而没有其他的操作,那么就可以用 volatile来代替 synchronized或者代替原子变量,因为赋值自身是有原子性的,而 volatile又保证了可见性,所以就足以保证线程安全

学以致用：用volatile修正重排序问题

volatile 小结

1. volatile修饰符适用于以下场景：某个属性被多个线程共享,其中有一个线程修改了此属性，其他线程可以立即得到修改后的值,比如boolean flag；或者作为触发器，实现轻量级同步。
2. volatile属性的读写操作都是无锁的,它不能替代 synchronized因为它没有提供原子性和互斥性。因为无锁,不需要花费时间在获取锁和释放锁上,所以说它是低成本的。
3. volatile只能作用于属性,我们用 volatile修饰属性,这样compilers就不会对这个属性做指令重排序。
4. volatile提供了可见性,任何一个线程对其的修改将立马对其他线程可见。 volatile属性不会被线程存,始终从主存中读取。
5. volatile提供了 happens- before保证,对 volatile变量v的写入 happens- before所有其他线程后续对 v 的读操作。

6. volatile可以使得Long和 double的赋值是原子的,后面马上会讲long和 double的原子性。

八、可见性对 synchronized的升华、能保证可见性的措施、可见性总结

1. 能保证可见性的措施

除了 volatiles可以让变量保证可见性外, synchronized、Lock、并发集合、 Thread join() 和 Thread. start()等都可以保证的可见性,具体看happens-before原则的规定

2. 升华:对 synchronized可见性的正确理解

- synchronized不仅保证了原子性,还保证了可见性 即前面一个线程在synchronized里面的操作对于第二个线程是可见的
- synchronized不仅让被保护的代码安全,还近朱者赤 下个synchronized进入之前，不仅上个线程的整个synchronized代码块是可以被后面的线程看到的，synchronized之前的代码也可以被看到

九、原子性

- 什么是原子性
- Java中的原子操作有哪些?
- long和 double的原子性
- 原子操作+原子操作!=原子操作

1. 什么是原子性

系列的操作,要么全部执行成功,要么全部不执行,不会出现执行一半的情况,是不可分割的。

2. Java中的原子操作有哪些

1. 除long和 doubler之外的基本类型(int, byte, boolean, short. char,float)的赋值操作
2. 所有引用 reference的斌值操作,不管是32位的机器还是64位的机器
3. java.concurrent. Atomic.*包中所有类的原子操作

3. long和 double的原子性

问题描述：官方文档、对于64位的值的写入,可以分为两个32位的操作进行写入、读取错误、使用volatile解决

结论：在32位上的JVM上,long和 double的操作不是原子的,但是在64位的JVM上是原子的

实际开发中：商用Java虚拟机中不会出现。我们开发中所使用的虚拟机通常已经考虑到

十、JMM面试常见问题

1. JMM应用实例:单例模式8种写法、单例和并发的关系(真实面试超高频考点)

- 单例模式的作用
    - 节省内存和计算
    - 保证结果正确
    - 方便管理
- 单例模式使用场景

1. 无状态的工具类:比如日志工具类,不管是在哪里使用,我们需要的只是它帮我们记录日志信息,除此之外,并不需要在它的实例对象上存储任何状态,这时候我们就只需要一个实例对象即可。
2. 全局信息类:比如我们在一个类上记录网站的访问次数,我们门不希望有的访问被记录在对象A上,有的却记录在对象B上,这时候我们就让这个类成为单例。

2. 单例模式的8种写法

（1）饿汉式——静态常量（可用）

```
package singleton;

/**
 * 描述:       饿汉式（静态常量）（可用）
 */
public class Singleton1 {

    private final static Singleton1 INSTANCE = new Singleton1();
   //构造方法
    private Singleton1() {

    }

    public static Singleton1 getInstance() {
        return INSTANCE;
    }

}
```

（2）饿汉式——（静态代码块）（可用）

```
package singleton;

/**
 * @desc: // 饿汉式单例模式（静态代码块）（可用）  实际上与第一种相同，优点和缺点和第一类一样
 * @author: Mr.Han
 */
public class Singleton2 {

    private static final Singleton2 INSTANCE;

    static {
        INSTANCE  = new Singleton2();
    }

    private Singleton2(){}

    public static Singleton2 getInstance() {
        return INSTANCE;
    }
}
```

(3) 懒汉式——没有加synchronized （线程不安全，不可用）

```
package singleton;

/**
 * @desc:  饿汉式 没有加synchronized 会有线程安全问题 （不可用）
 * @author: Mr.Han
 */
public class Singleton3 {

    private static Singleton3 singleton3 = null;

    private Singleton3() {

    }

    private static Singleton3 getInstance() {

        if (singleton3 == null) {
            singleton3 = new Singleton3();
        }

        return singleton3;
    }

}
```

(4) 懒汉式 —— 创建的方法名上使用synchronized （不推荐，效率太差）

```
package singleton;

/**
 * @desc:  饿汉式 使用加synchronized 无线程安全问题 （不推荐） => 效率太低
 * @author: Mr.Han
 */
public class Singleton4 {

    private static Singleton4 singleton3 = null;

    private Singleton4() {

    }

    private synchronized static Singleton4 getInstance() {

        if (singleton3 == null) {
            singleton3 = new Singleton4();
        }

        return singleton3;
    }

}
```

（5）懒汉式——只把`instance = new Singleton5();`用`synchronized`包住（线程不安全）（不可用）

```
package singleton;

/**
 * 描述：       懒汉式（线程不安全）（不可用）
 */
public class Singleton5 {

    private static Singleton5 instance;

    private Singleton5() {

    }

    public static Singleton5 getInstance() {
        if (instance == null) {
            synchronized (Singleton5.class) {
                instance = new Singleton5();
            }
        }
        return instance;
    }
}
```

（6）懒汉式 —— 双重检查，使用volatile关键字，面试用

```
package singleton;

/**
 * 描述：       懒汉式使用双重检查（线程安全）（面试推荐推荐）
 */
public class Singleton6 {

    private static volatile Singleton6 instance;

    private Singleton6() {

    }

    public static Singleton6 getInstance() {
        if (instance == null) {
            synchronized (Singleton6.class) {
                // 双重检查即在同步中再检查一次
                if (instance == null) {
                    instance = new Singleton6();
                }
            }
        }
        return instance;
    }
}
```

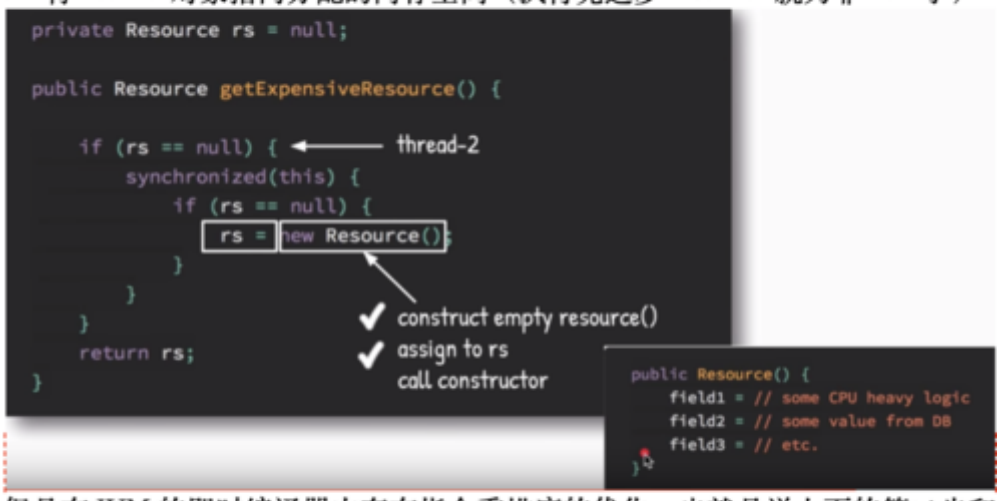
- 优点:线程安全;延迟加载;效率较高。

- 为什么要 double- check
    1. 线程安全
    2. 单 check行不行?
    3. 性能问题
- 为什么要用 volatile
    1. 新建对象实际上有3个步骤

       在这里的volatile想要防止的,是这种特殊情况：“在第一个线程退出 synchronized之前,里面的操作执行了一部分,比如执行了new却还没执行构造函数,然后第一个线程被切换走了,这个时候第二个线程刚刚到第一重检查,所以看到的对象就是非空,就跳过了整个 synchronized代码块,获取到了这个单例对象,但是使用其中的属性的时候却不是想要的值。”

       主要在于instance = new Singleton()这句，这并非是一个原子操作，事实上在 JVM 中这句话做了下面 3 件事情。

       1. 给 instance 分配内存
       2. 调用 Singleton 的构造函数来初始化成员变量
       3. 将instance对象指向分配的内存空间（执行完这步 instance 就为非 null 了）

       但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，已经线程一被调度器暂停，此时线程二刚刚进来第一重检查，看到的 instance 已经是非 null 了（但却没有初始化，里面的值可能是null/false/0，总之不是构造函数中指定的值），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错或者是看到了非预期的值（因为此时属性的值是默认值而不是所需要的值）。

    2. 重排序会导致NPE（空指针异常）
    3. 防止重排序

(7) 懒汉式使用静态内部类（线程安全）

```
package singleton;

/**
 * 描述:        懒汉式使用静态内部类（线程安全）
 *              必须加volatile关键字
 */
public class Singleton7 {
    private Singleton7() {

    }

```

```
    public static Singleton7 getInstance() {
        return SingletonInstance.INSTANCE;
    }

    private static class SingletonInstance {
       private static final Singleton7 INSTANCE = new Singleton7();
    }
}
```

（8）懒汉式——枚举（可用）

- 生产实践中最佳的写法

```
package singleton;

/**
 * @desc: 懒汉式使用枚举
 * @author: Mr.Han
 */
public enum Singleton8 {
    INSTANCE;

    public void whatever() {
        System.out.println("I am Enum Singleton type");
    }

}
```

调用：

```
package singleton;

/**
 * @desc:
 * @author: Mr.Han
 */
public class TestSingleton8 {
    public static void main(String[] args) {
        Singleton8.INSTANCE.whatever();
    }
}

// sout:   I am Enum Singleton type
```

3. 哪种单例模式最好?

- Joshua Bloch大神在《 Effective Java》中明确表达过的观点:“使用枚举实现单例的方法虽然还没有广泛采用,但是单元素的枚举类型已经成为实现 Singleton的最佳方法。
- 写法简单
- 线程安全有保障

  经过反编译我们可以发现，枚举实际上会被编译成一个final Class,继承了Enum类，并且Enum这个父类中的各个实例基本都是static定义的，所以枚举的本质经过反编译后就是一个静态的对象。第一次在使用到枚举这个实例的时候才会被加载进来，是一种懒加载。

```
public final enum singleton/Singleton8 extends java/lang/Enum {
```

- 避免反序列化破坏单例：其他的实现单例的方法，是可以通过反射/反序列化这种方式绕过去的。比如说用反射就可以把私有的构造方法给绕过去。反序列化同样可以反序列化多个实例。如果使用枚举就可以避免这种情况的发生。

> 而对于序列化这件事情，Java专门对枚举的序列化做了规定，在序列化时仅仅是将枚举对象的name属性输出到结果中，反序列化的时候，就是通过java.lang.Enum的valueOf方法，来根据名字查找枚举对象，而不是创建新的对象，所以这就防止了反序列化导致的单例破坏问题的出现。
>
> 对于通过反射破坏单例而言，枚举类同样有防御措施。反射在通过newInstance创建对象时，会检查这个类是否是枚举类，如果是，就抛出 IllegalArgumentException("Cannot reflectively create enum objects") 异常，反射创建对象失败。
>
> 可以看出，枚举这种方式，能够防止序列化和反射破坏单例，在这一点上，与其他的实现方式比，有很大的优势。安全问题不容小视，一旦生成了多个实例，单例模式就彻底没用了。

总结：

- 最好的方法是利用枚举,因为还可以防止反序列化重新创建新的对象;
- 非线程同步的单例模式的实现方法不能使用;
- 如果程序一开始要加载的资源太多,那么就应该使用懒加载;
- 饿汉式如果是对象的创建，对象创建前需要一些配置文件，这种情况下饿汉式就不适用。
- 懒加载虽然好,但是静态内部类这种方式会引入编程复杂性

4. 说一说JMM

1. 是什么
    1. 首先**JMM实际上是一种规范，而且是一组规范**，需要各个JVM的实现来遵守MM规范,以便于开发者可以利用这些规范,更方便地开发多线程程序
    2. JMM是工具类和关键字的原理
    3. 如果没有JMM,那就需要我们自己指定什么时候用内存栅栏（工作内存和主内存之间的拷贝和同步）等,那是相当麻烦的,幸好有了JMM,让我们只需要用同步工具和关键字就可以开发并发程序。
2. 聊重排、可见性、原子性
    1. 重点讲可见性：哪些可以保证可见性，引出volitile
    2. happens-before原则：哪些必须实现happens-before原则
    3. 原子性：java中自身的原子操作

5.什么是原子操作?Java中有哪些原子操作?生成对象的过程是不是原子操作?

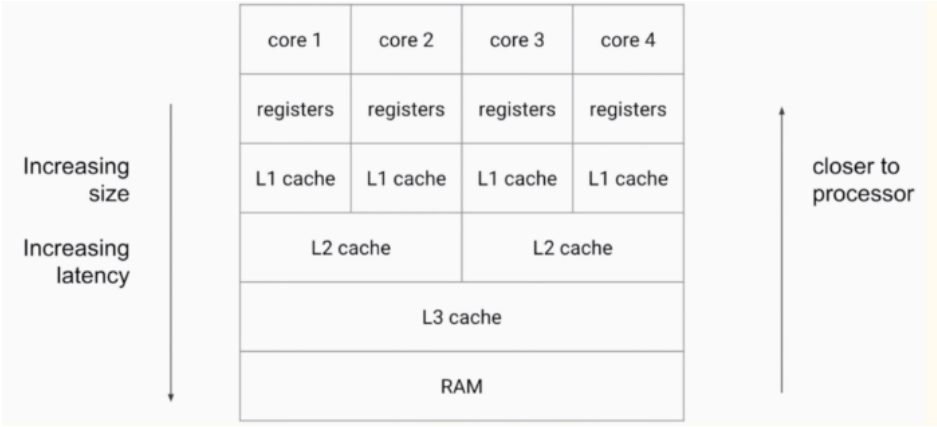
>

6. 什么是内存可见性

> 为了提高CPU的运行效率,CPU内加入了高速缓存,高速缓存的容量比主内存小,但是速度仅次于寄存器,所以在CPU和主内存之间就多了 Cache层,导致了多线程时很多问题的发生线程间的对于共享变量的可见性问题不是直接由多核引起的,而是由多缓存引起的。如果所有个核心都只用一个缓存,那么也就不存在内存可见性问题了。现代多核CPU中每个核心拥有自己的一级缓存或一级缓存加上二级缓存等,问题就发生在每个核的独占缓存上。每个核心都会将自己需要的数据读到独占缓存中,数

> 据修改后也是写入到缓存中,然后等待刷入到主存中。所以会导致有些核心读取的值是一个过期的值

CPU缓存结构图:



Java作为高级语言,屏蔽了这些底层细节,用JMM定义了一套读写内存数据的规范,虽然我们不再需要关心一级缓存和二级缓存的问题,但是,JMM抽象了主内存和本地内存的概念。这里说的本地内存并不是真的是一块给每个线程分配的内存,而是JMM的一个抽象,是对于寄存 器、一级缓存、二级缓存等的抽象。