

JUC多线程 (一)

学习目标：

- 掌握多线程的创建
- 掌握线程安全的处理
- 了解线程状态
- 掌握线程停止的两种方法
- 了解线程的原子性，可见性和有序性
- 理解内存可见性的原理
- 掌握synchronized解决内存可见性

1. 多线程基础

一个采用了多线程技术的应用程序可以更好地利用系统资源。其主要优势在于充分利用了CPU的空闲时间片，可以用尽可能少的时间来对用户的要求做出响应，使得进程的整体运行效率得到较大提高，同时增强了应用程序的灵活性。

更为重要的是，由于同一进程的所有线程是共享同一内存，所以不需要特殊的数据传送机制，不需要建立共享存储区或共享文件，从而使得不同任务之间的协调操作与运行、数据的交互、资源的分配等问题更加易于解决。

1.1 线程和进程

进程：

是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，一个应用程序可以同时运行多个进程；进程也是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创建、运行到消亡的过程。

线程：

进程内部的一个独立执行单元；一个进程可以同时并发的运行多个线程，可以理解为一个进程便相当于一个单 CPU 操作系统，而线程便是这个系统中运行的多个任务。

进程：有独立的内存空间，进程中的数据存放空间（堆空间和栈空间）是独立的，至少有一个线程。

线程：堆空间是共享的，栈空间是独立的，线程消耗的资源比进程小的多。

注意：

1. 因为一个进程中的多个线程是并发运行的，那么从微观角度看也是有先后顺序的，哪个线程执行完全取决于CPU的调度，程序员是不能完全控制的（可以设置线程优先级）。而这也就造成的多线程的随机性。
2. Java程序的进程里面至少包含两个线程，主线程也就是main()方法线程，另外一个垃圾回收机制线程。每
当使用java命令执行一个类时，实际上都会启动一个JVM，每一个JVM实际上就是在操作系统中启动了一个
线程，java本身具备了垃圾的收集机制，所以在Java运行时至少会启动两个线程。
3. 由于创建一个线程的开销比创建一个进程的开销小的多，那么我们在开发多任务运行的时候，通常考虑创建多线程，而不是创建多进程。

1.2 多线程的创建

创建Maven工程，编写测试类

1.2.1 继承Thread类

第一种继承Thread类 重写run方法

```
public class Demo1CreateThread extends Thread {  
  
    public static void main(String[] args) throws  
        InterruptedException {  
  
        System.out.println("-----多线程创建开始-----");  
        // 1. 创建一个线程  
        CreateThread createThread1 = new CreateThread();  
        CreateThread createThread2 = new CreateThread();  
        // 2. 开始执行线程 注意 开启线程不是调用run方法，而是start方法  
        System.out.println("-----多线程创建启动-----");  
    }  
}
```



```
        System.out.println("-----多线程创建结束-----");  
    }  
  
    static class CreateThread extends Thread {  
        public void run() {  
            String name = Thread.currentThread().getName();  
            for (int i = 0; i < 5; i++) {  
                System.out.println(name + "打印内容是:" + i);  
            }  
        }  
    }  
}
```

1.2.2 实现Runnable接口

实现Runnable接口,重写run方法

实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是继承Thread类还是实现Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的。

```
public class Demo2CreateRunnable {  
  
    public static void main(String[] args) {  
        System.out.println("-----多线程创建开始-----");  
        // 1.创建线程  
        CreateRunnable createRunnable = new CreateRunnable();  
        Thread thread1 = new Thread(createRunnable);  
        Thread thread2 = new Thread(createRunnable);  
        // 2.开始执行线程 注意 开启线程不是调用run方法，而是start方法  
        System.out.println("-----多线程创建启动-----");  
        thread1.start();  
        thread2.start();  
        System.out.println("-----多线程创建结束-----");  
    }  
  
    static class CreateRunnable implements Runnable {  
  
        public void run() {  
            String name = Thread.currentThread().getName();  
            for (int i = 0; i < 5; i++) {
```

```
}  
}  
}
```

实现Runnable接口比继承Thread类所具有的优势：

1. 适合多个相同的程序代码的线程去共享同一个资源。
2. 可以避免java中的单继承的局限性。
3. 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和数据独立。
4. 线程池只能放入实现Runnable或callable类线程，不能直接放入继承Thread的类

1.2.3 匿名内部类方式

使用线程的内匿名内部类方式，可以方便的实现每个线程执行不同的线程任务操作

```
public class Demo3Runnable {  
    public static boolean exit = true;  
  
    public static void main(String[] args) throws  
InterruptedException {  
        new Thread(new Runnable() {  
            public void run() {  
                String name = Thread.currentThread().getName();  
                for (int i = 0; i < 5; i++) {  
                    System.out.println(name + "执行内容: " + i);  
                }  
            }  
        }).start();  
  
        new Thread(new Runnable() {  
            public void run() {  
                String name = Thread.currentThread().getName();  
                for (int i = 0; i < 5; i++) {  
                    System.out.println(name + "执行内容: " + i);  
                }  
            }  
        }).start();  
    }  
}
```



```
}  
}}
```

1.2.4 守护线程

Java中有两种线程，一种是用户线程，另一种是守护线程。

用户线程是指用户自定义创建的线程，主线程停止，用户线程不会停止。

守护线程当进程不存在或主线程停止，守护线程也会被停止。

```
public class Demo4Daemon {  
    public static void main(String[] args) {  
        Thread thread = new Thread(new Runnable() {  
            public void run() {  
                for (int i = 0; i < 10; i++) {  
                    try {  
                        Thread.sleep(10);  
                    } catch (Exception e) {}  
                    System.out.println("子线程..." + i);  
                }  
            }  
        });  
  
        // 设置线程为守护线程  
        //thread.setDaemon(true);  
        thread.start();  
  
        for (int i = 0; i < 5; i++) {  
            try {  
                Thread.sleep(10);  
                System.out.println("主线程" + i);  
            } catch (Exception e) {}  
        }  
  
        System.out.println("主线程执行完毕!");  
    }  
}
```

1.3 线程安全

1.3.1 卖票案例

如果有多个线程在同时运行，而这些线程可能会同时运行这段代码。程序每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的，反之则是线程不安全的。

```
public class Demo5Ticket {

    public static void main(String[] args) {
        //创建线程任务对象
        Ticket ticket = new Ticket();
        //创建三个窗口对象
        Thread t1 = new Thread(ticket, "窗口1");
        Thread t2 = new Thread(ticket, "窗口2");
        Thread t3 = new Thread(ticket, "窗口3");

        //卖票
        t1.start();
        t2.start();
        t3.start();
    }

    static class Ticket implements Runnable {

        //Object lock = new Object();
        ReentrantLock lock = new ReentrantLock();
        private int ticket = 10;

        public void run() {
            String name = Thread.currentThread().getName();
            while (true) {
                sell(name);
                if (ticket <= 0) {
                    break;
                }
            }
        }

        private void sell(String name) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            lock.lock();
            try {
                ticket--;
            } finally {
                lock.unlock();
            }
        }
    }
}
```



```
        }  
        if (ticket > 0) {  
            System.out.println(name + "卖票: " + ticket);  
            ticket--;  
        }  
    }  
}
```

线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

1.3.2 线程同步

当我们使用多个线程访问同一资源的时候，且多个线程中对资源有写的操作，就容易出现线程安全问题。

要解决上述多线程并发访问一个资源的安全问题，Java中提供了同步机制(synchronized)来解决。

同步代码块

```
Object lock = new Object(); //创建锁  
synchronized(lock){  
    //可能会产生线程安全问题的代码  
}
```

同步方法

```
//同步方法  
public synchronized void method(){  
    //可能会产生线程安全问题的代码  
}
```

证明方式: 一个线程使用同步代码块(this明锁),另一个线程使用同步函数。如果两个线程抢票不能实现同步, 那么会出现数据错误。

```
//使用this锁的同步代码块
synchronized(this){
    //需要同步操作的代码
}
```

Lock锁

```
Lock lock = new ReentrantLock();
lock.lock();
    //需要同步操作的代码
lock.unlock();
```

1.3.2 死锁

多线程死锁: 同步中嵌套同步,导致锁无法释放。

死锁解决办法: 不要在同步中嵌套同步

```
public class Demo6DeadLock {

    public static void main(String[] args) {
        //创建线程任务对象
        Ticket ticket = new Ticket();
        //创建三个窗口对象
        Thread t1 = new Thread(ticket, "窗口1");
        Thread t2 = new Thread(ticket, "窗口2");
        Thread t3 = new Thread(ticket, "窗口3");

        //卖票
        t1.start();
        t2.start();
        t3.start();
    }

    static class Ticket implements Runnable {

        Object lock = new Object();
```




```
public void run() {
    String name = Thread.currentThread().getName();
    while (true) {
        if ("窗口1".equals(name)) {
            synchronized (lock) {
                sell(name);
            }
        } else {
            sell(name);
        }
        if (ticket <= 0) {
            break;
        }
    }
}

private synchronized void sell(String name) {
    synchronized (lock) {
        if (ticket > 0) {
            System.out.println(name + "卖票: " + ticket);
            ticket--;
        }
    }
}
}
```

1.4 线程状态

1.4.1 线程状态介绍

查看Thread源码，能够看到java的线程有六种状态：

```
public enum State {
    NEW,
    RUNNABLE,
    BLOCKED,
    WAITING,
    TIMED_WAITING,
    TERMINATED;
}
```

线程刚被创建，但是并不运行。

RUNNABLE(可运行)

线程可以在java虚拟机中运行的状态，可能正在运行自己代码，也可能没有，这取决于操作系统处理器。

BLOCKED(锁阻塞)

当一个线程试图获取一个对象锁，而该对象锁被其他的线程持有，则该线程进入Blocked状态；当该线程持有锁时，该线程将变成Runnable状态。

WAITING(无限等待)

一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入waiting状态。进入这个状态后是不能自动唤醒的，必须等待另一个线程调用notify或者notifyAll方法才能够唤醒。

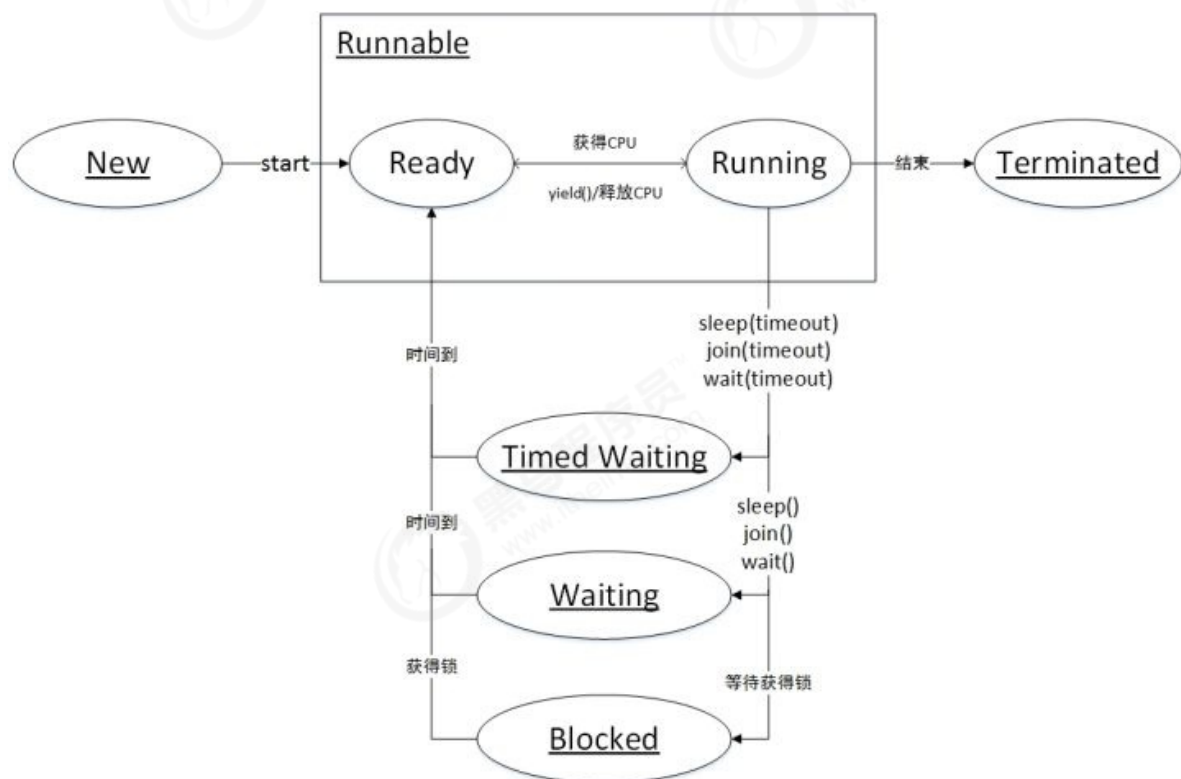
TIMED_WAITING(计时等待)

同waiting状态，有几个方法有超时参数，调用他们将进入Timed waiting状态。这一状态将一直保持到超时期满或者接收到唤醒通知。带有超时参数的常用方法有Thread.sleep、Object.wait。

TERMINATED(被终止)

因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡。

1.4.2 线程状态图



1.4.3 wait()、notify()



人心。

wait 方法会使持有该对象的线程把该对象的控制权交出去，然后处于等待状态。

notify 方法会通知**某个**正在等待这个对象的控制权的线程继续运行。

notifyAll 方法会通知**所有**正在等待这个对象的控制权的线程继续运行。

注意：一定要在线程同步中使用,并且是同一个锁的资源

wait和notify方法例子，一个人进站出站：

```
public class Demo7WaitAndNotify {  
    public static void main(String[] args) {  
        State state = new State();  
        InThread inThread = new InThread(state);  
        OutThread outThread = new OutThread(state);  
  
        Thread in = new Thread(inThread);  
        Thread out = new Thread(outThread);  
  
        in.start();  
        out.start();  
    }  
  
    // 控制状态  
    static class State {  
        //状态标识  
        public String flag = "车站外";  
    }  
  
    static class InThread implements Runnable {  
        private State state;  
  
        public InThread(State state) {  
            this.state = state;  
        }  
  
        public void run() {  
            while (true) {  
                synchronized (state) {  
                    if ("车站内".equals(state.flag)) {  
                        try {  
                            // 如果在车站内，就不用进站，等待，释放锁  
                            state.wait();  
                        } catch (Exception e) {  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```



```
        System.out.println("进站");
        state.flag = "站内";
        // 唤醒state等待的线程
        state.notify();
    }
}

static class OutThread implements Runnable {
    private State state;

    public OutThread(State state) {
        this.state = state;
    }

    public void run() {
        while (true) {
            synchronized (state) {
                if ("站外".equals(state.flag)) {
                    try {
                        // 如果在站外，就不用出站了，等待，释放锁
                        state.wait();
                    } catch (Exception e) {
                    }
                }
                System.out.println("出站");
                state.flag = "站外";
                // 唤醒state等待的线程
                state.notify();
            }
        }
    }
}
```

1.4.4 wait与sleep区别

- 对于sleep()方法，首先要知道该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。

在休眠状态等待，一旦满足条件，又会自动恢复运行状态。

wait()是把控制权交出去，然后进入等待此对象的等待锁定池处于等待状态，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备获取对象锁进入运行状态。

- 在调用sleep()方法的过程中，线程不会释放对象锁。而当调用wait()方法的时候，线程会放弃对象锁。

1.5 线程停止

结束线程有以下三种方法：

- (1) 设置退出标志，使线程正常退出。
- (2) 使用interrupt()方法中断线程。
- (3) 使用stop方法强行终止线程（不推荐使用Thread.stop, 这种终止线程运行的方法已经被废弃，使用它们是极端不安全的！）

1.5.1 使用退出标志

一般run()方法执行完，线程就会正常结束，然而，常常有些线程是伺服线程。它们需要长时间的运行，只有在外部的某些条件满足的情况下，才能关闭这些线程。使用一个变量来控制循环，例如：最直接的方法就是设一个boolean类型的标志，并通过设置这个标志为true或false来控制while循环是否退出，代码示例：

```
public class Demo8Exit {  
  
    public static boolean exit = true;  
  
    public static void main(String[] args) throws  
        InterruptedException {  
        Thread t = new Thread(new Runnable() {  
            public void run() {  
                while (exit) {  
                    try {  
                        System.out.println("线程执行！");  
                        Thread.sleep(1001);  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        })  
    }  
}
```

```
        t.start();

        Thread.sleep(10001);
        exit = false;
        System.out.println("退出标识位设置成功");
    }
}
```

1.5.2 使用interrupt()方法

使用interrupt()方法来中断线程有两种情况：

1)线程处于阻塞状态

如使用了sleep,同步锁的wait,socket中的receiver,accept等方法时，会使线程处于阻塞状态。当调用线程的interrupt()方法时，会抛出InterruptedException异常。阻塞中的那个方法抛出这个异常，通过代码捕获该异常，然后break跳出循环状态，从而让我们有机会结束这个线程的执行。

2)线程未处于阻塞状态

使用isInterrupted()判断线程的中断标志来退出循环。当使用interrupt()方法时，中断标志就会置true，和使用自定义的标志来控制循环是一样的道理。

```
public class Demo9Interrupt {

    public static boolean exit = true;

    public static void main(String[] args) throws
InterruptedException {
        Thread t = new Thread(new Runnable() {
            public void run() {
                while (exit) {
                    try {
                        System.out.println("线程执行！");

                        //判断线程的中断标志来退出循环
                        if
(Thread.currentThread().isInterrupted()) {
                            break;
                        }
                    }
                }
            }
        });
    }
}
```



时，

```
        } catch (InterruptedException e) {  
            e.printStackTrace();  
            //线程处于阻塞状态,当调用线程的interrupt()方法  
  
            //会抛出InterruptedException异常,跳出循环  
            break;  
        }  
    }  
}  
});  
t.start();  
  
Thread.sleep(10001);  
//中断线程  
t.interrupt();  
System.out.println("线程中断了");  
}  
}
```

1.6 线程优先级

1.6.1 优先级priority

现今操作系统基本采用分时的形式调度运行的线程，线程分配得到时间片的多少决定了线程使用处理器资源的多少，也对应了线程优先级这个概念。

在JAVA线程中，通过一个int priority来控制优先级，范围为1-10，其中10最高，默认值为5。

```
public class Demo10Priorityt {  
  
    public static void main(String[] args) {  
        PrioritytThread prioritytThread = new PrioritytThread();  
  
        // 如果8核CPU处理3线程，无论优先级高低，每个线程都是单独一个CPU执行，就无法体现优先级  
        // 开启10个线程，让8个CPU处理，这里线程就需要竞争CPU资源，优先级高的能分配更多的CPU资源  
        for (int i = 0; i < 10; i++) {  
            Thread t = new Thread(prioritytThread, "线程" + i);  
            if (i == 1) {
```



```
        if (i == 2) {
            t.setPriority(1);
        }
        t.setDaemon(true);
        t.start();
    }

    try {
        Thread.sleep(10001);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("线程1总计: " + PrioritytThread.count1);
    System.out.println("线程2总计: " + PrioritytThread.count2);
}

static class PrioritytThread implements Runnable {
    public static Integer count1 = 0;
    public static Integer count2 = 0;

    public void run() {
        while (true) {
            if ("线程
1".equals(Thread.currentThread().getName())) {
                count1++;
            }
            if ("线程
2".equals(Thread.currentThread().getName())) {
                count2++;
            }

            if (Thread.currentThread().isInterrupted()) {
                break;
            }
        }
    }
}
```

1.6.2 join()方法



因此，线程A的线性工作顺序对于线程B的线性工作顺序而言，线程A的线性工作顺序是线性的，直到线程A执行完毕后，才会继续执行线程B。

```
public class Demo11Join {
    public static void main(String[] args) {
        JoinThread joinThread = new JoinThread();
        Thread thread1 = new Thread(joinThread, "线程1");
        Thread thread2 = new Thread(joinThread, "线程2");
        Thread thread3 = new Thread(joinThread, "线程3");
        thread1.start();
        thread2.start();
        thread3.start();

        try {
            thread1.join();
        } catch (Exception e) {

        }

        for (int i = 0; i < 5; i++) {
            System.out.println("main ---i:" + i);
        }
    }

    static class JoinThread implements Runnable {
        private Random random = new Random();

        public void run() {
            String name = Thread.currentThread().getName();
            for (int i = 0; i < 5; i++) {
                try {
                    Thread.sleep(random.nextInt(10));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(name + "内容是:" + i);
            }
        }
    }
}
```

1.6.3 yield方法

13XX未)

yield()让当前正在运行的线程回到可运行状态，以允许具有相同优先级的其他线程获得运行的机会。因此，使用yield()的目的是让具有相同优先级的线程之间能够适当的轮换执行。但是，实际中无法保证yield()达到让步的目的，因为，让步的线程可能被线程调度程序再次选中。

查看源码介绍：

```
/**
 * A hint to the scheduler that the current thread is willing to yield
 * its current use of a processor. The scheduler is free to ignore this
 * hint. 向调度程序提示当前线程愿意产生当前对处理器的使用。
 * 调度程序可以忽略此提示。
 *
 * <p> Yield is a heuristic attempt to improve relative progression
 * between threads that would otherwise over-utilise a CPU. Its use
 * should be combined with detailed profiling and benchmarking to
 * ensure that it actually has the desired effect.
 *
 * <p> It is rarely appropriate to use this method. It may be useful
 * for debugging or testing purposes, where it may help to reproduce
 * bugs due to race conditions. It may also be useful when designing
 * concurrency control constructs such as the ones in the
 * {@link java.util.concurrent.locks} package. 很少适合使用这种方法。它可能对调试或测试有用，
 * 因为在调试或测试中，它可能有助于根据竞争条件
 * 重现错误。在设计并发控制结构（如java.util.
 * concurrent.locks包中的结构）时，它也可能很有用。
 */
public static native void yield()
```

结论：大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。

2. 多线程并发的3个特性

多线程并发开发中，要知道什么是多线程的原子性，可见性和有序性，以避免相关的问题产生。

2.1 原子性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行

一个很经典的例子就是银行账户转账问题：

比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。



1000元之后，银行大额中止。这件事云云转账厂家虽然喊云J 1000元，但是厂家D没有收到这个转过来的1000元。

所以这2个操作**必须要具备原子性**才能保证不出现一些意外的问题。

2.2 可见性

可见性：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

举个简单的例子，看下面这段代码：

```
//线程1执行的代码
int i = 0;
i = 10;

//线程2执行的代码
j = i;
```

当线程1执行 `int i = 0` 这句时，`i` 的初始值0加载到内存中，然后再执行 `i = 10`，那么在内存中 `i` 的值变为10了。

如果当线程1执行到 `int i = 0` 这句时，此时线程2执行 `j = i`，它读取 `i` 的值并加载到内存中，注意此时内存当中 `i` 的值是0，那么就会使得 `j` 的值也为0，而不是10。

这就是可见性问题，线程1对变量 `i` 修改了之后，线程2没有立即看到线程1修改的值。

2.3 有序性

有序性：程序执行的顺序按照代码的先后顺序执行

```
int count = 0;
boolean flag = false;
count = 1; //语句1
flag = true; //语句2
```

文里进行赋值且打印。从代码顺序上看，语句1肯定在语句2前面，那么JVM在执行这段代码的时候会保证语句1一定会在语句2前面执行吗？不一定，为什么呢？这里可能会发生指令重排序（Instruction Reorder）。

什么是重排序？一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致。

as-if-serial:无论如何重排序，程序最终执行结果和代码顺序执行的结果是一致的。Java编译器、运行时和处理器都会保证Java在单线程下遵循as-if-serial语义)

上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。但是要注意，虽然处理器会对指令进行重排序，但是它会保证程序最终结果会和代码顺序执行结果相同，那么它靠什么保证的呢？

再看下面一个例子：

```
int a = 10; //语句1
int b = 2;  //语句2
a = a + 3;  //语句3
b = a*a;    //语句4
```

这段代码有4个语句，那么可能的一个执行顺序是： 语句2 语句1 语句3 语句4

不可能是这个执行顺序： 语句2 语句1 语句4 语句3

因为处理器在进行重排序时是会考虑指令之间的数据依赖性，如果一个指令 Instruction 2必须用到Instruction 1的结果，那么处理器会保证Instruction 1会在Instruction 2之前执行。虽然重排序不会影响单个线程内程序执行的结果，但是多线程会有影响

下面看一个例子：

```
init = false
context = loadContext(); //语句1
init = true; //语句2

//线程2:
while(!init){//如果初始化未完成，等待
    sleep();
}
execute(context);//初始化完成，执行逻辑
```

上面代码中，由于语句1和语句2没有数据依赖性，因此可能会被重排序。假如发生了重排序，在线程1执行过程中先执行语句2，而此时线程2会以为初始化工作已经完成，那么就会跳出while循环，去执行execute(context)方法，而此时context并没有被初始化，就会导致程序出错。

从上面可以看出，**重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。**

要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能导致程序运行不正确。

3. Java内存可见性

3.1 了解Java内存模型

JVM内存结构、Java对象模型和Java内存模型，这就是三个截然不同的概念，而这三个概念很容易混淆。这里详细区别一下

3.1.1 JVM内存结构

我们都知道，Java代码是要运行在虚拟机上的，而虚拟机在执行Java程序的过程中会把所管理的内存划分为若干个不同的数据区域，这些区域都有各自的用途。其中有些区域随着虚拟机进程的启动而存在，而有些区域则依赖用户线程的启动和结束而建立和销毁。

在《Java虚拟机规范（Java SE 8）》中描述了JVM运行时内存区域结构如下：



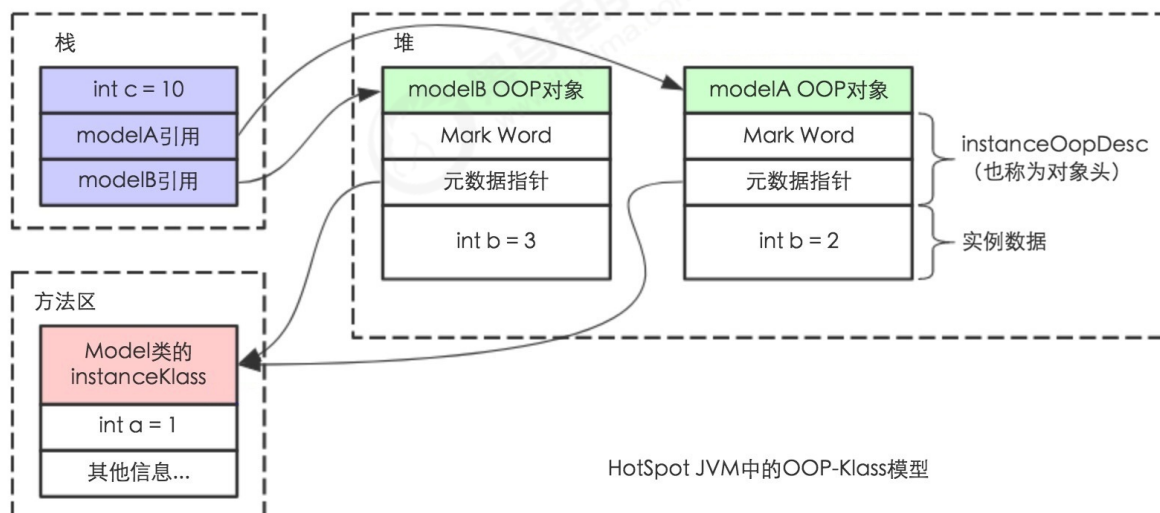
JVM内存结构，由Java虚拟机规范定义。描述的是Java程序执行过程中，由JVM管理的不同数据区域。各个区域有其特定的功能。

3.1.2 Java对象模型

Java是一种面向对象的语言，而Java对象在JVM中的存储也是有一定的结构的。而这个关于Java对象自身的存储模型称之为Java对象模型。

HotSpot虚拟机中（Sun JDK和OpenJDK中所带的虚拟机，也是目前使用范围最广的Java虚拟机），设计了一个OOP-Klass Model。OOP（Ordinary Object Pointer）指的是普通对象指针，而Klass用来描述对象实例的具体类型。

每一个Java类，在被JVM加载的时候，JVM会给这个类创建一个 `instanceKlass` 对象，保存在方法区，用来在JVM层表示该Java类。当我们在Java代码中，使用new创建一个对象的时候，JVM会创建一个 `instanceOopDesc` 对象，这个对象中包含了对象头以及实例数据。



HotSpot JVM中的OOP-Klass模型

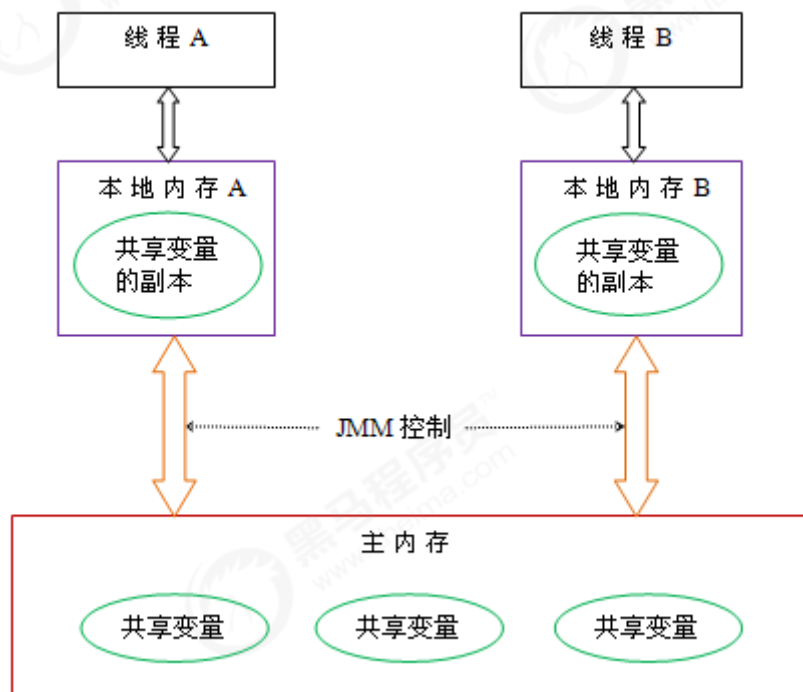
3.1.3 内存模型

Java内存模型就是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。

有兴趣详细了解Java内存模型是什么，为什么要有Java内存模型，Java内存模型解决了什么问题的学员，参考：<https://www.hollischuang.com/archives/2550>。

Java内存模型是根据英文Java Memory Model (JMM) 翻译过来的。其实JMM并不像JVM内存结构一样是真实存在的。他只是一个抽象的概念。JSR-133: [Java Memory Model and Thread Specification](#)中描述了，JMM是和多线程相关的，他描述了一组规则或规范，这个规范定义了一个线程对共享变量的写入时对另一个线程是可见的。

简单总结下，Java的多线程之间是通过共享内存进行通信的，而由于采用共享内存进行通信，在通信过程中会存在一系列如可见性、原子性、顺序性等问题，而JMM就是围绕着多线程通信以及与其相关的一系列特性而建立的模型。JMM定义了一些语法集，这些语法集映射到Java语言中就是volatile、synchronized等关键字。



JMM线程操作内存的基本的规则：

第一条关于线程与主内存：线程对共享变量的所有操作都必须在自己的工作内存（本地内存）中进行，不能直接从主内存中读写

里，线程间变量值的传递需要经过主内存才能完成。

• 主内存

主要存储的是Java实例对象，所有线程创建的实例对象都存放在主内存中，不管该实例对象是成员变量还是方法中的本地变量(也称局部变量)，当然也包括了共享的类信息、常量、静态变量。由于是共享数据区域，多条线程对同一个变量进行访问可能会发现线程安全问题。

• 本地内存

主要存储当前方法的所有本地变量信息(本地内存中存储着主内存中的变量副本拷贝)，每个线程只能访问自己的本地内存，即**线程中的本地变量对其它线程是不可见的**，就算是两个线程执行的是同一段代码，它们也会各自在自己的工作内存中创建属于当前线程的本地变量，当然也包括了字节码行号指示器、相关Native方法的信息。注意由于工作内存是每个线程的私有数据，线程间无法相互访问工作内存，因此存储在工作内存的数据不存在线程安全问题。

3.1.4 小结

JVM内存结构，和Java虚拟机的运行时区域有关。Java对象模型，和Java对象在虚拟机中的表现形式有关。**Java内存模型，和Java的并发编程有关。**

3.2 内存可见性

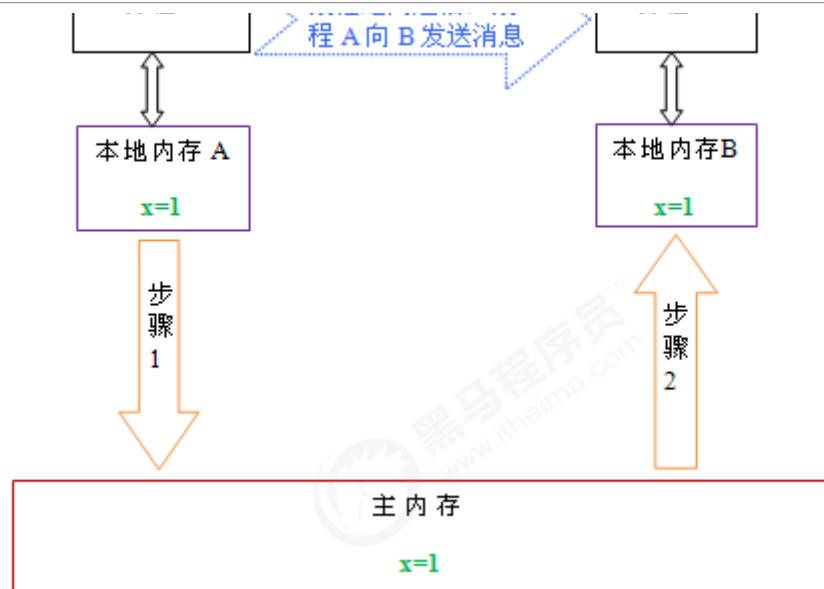
3.2.1 内存可见性介绍

可见性：一个线程对共享变量值的修改，能够及时的被其他线程看到

共享变量：如果一个变量在多个线程的工作内存中都存在副本，那么这个变量就是这几个线程的共享变量

线程 A 与线程 B 之间如要通信的话，必须要经历下面 2 个步骤：

1. 首先，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去。
2. 然后，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。



如上图所示，本地内存 A 和 B 有主内存中共享变量 x 的副本。假设初始时，这三个内存中的 x 值都为 0。线程 A 在执行时，把更新后的 x 值（假设值为 1）临时存放在自己的本地内存 A 中。当线程 A 和线程 B 需要通信时，线程 A 首先会把自己本地内存中修改后的 x 值刷新到主内存中，此时主内存中的 x 值变为了 1。随后，线程 B 到主内存中去读取线程 A 更新后的 x 值，此时线程 B 的本地内存的 x 值也变为了 1。

从整体来看，这两个步骤实质上是线程 A 在向线程 B 发送消息，而且这个通信过程必须要经过主内存。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 程序员提供内存可见性保证。

3.3.2 可见性问题

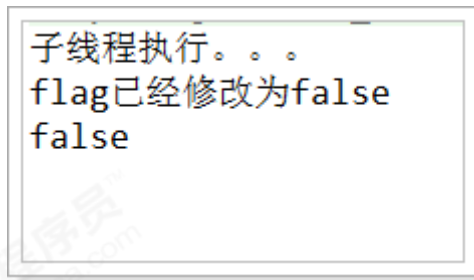
前面讲过多线程的内存可见性，现在我们写一个内存不可见的问题。

案例如下：

```
public class Demo1Jmm {  
  
    public static void main(String[] args) throws  
        InterruptedException {  
  
        JmmDemo demo = new JmmDemo();  
        Thread t = new Thread(demo);  
        t.start();  
        Thread.sleep(100);  
        demo.flag = false;  
        System.out.println("已经修改为false");  
        System.out.println(demo.flag);  
    }  
}
```

```
static class JmmDemo implements Runnable {  
    public boolean flag = true;  
  
    public void run() {  
        System.out.println("子线程执行。。。");  
        while (flag) {  
        }  
        System.out.println("子线程结束。。。");  
    }  
}
```

执行结果



按照main方法的逻辑，我们已经把flag设置为false，那么从逻辑上讲，子线程就应该跳出while死循环，因为这个时候条件不成立，但是我们可以看到，程序仍旧执行中，并没有停止。

原因:线程之间的变量是不可见的，因为读取的是副本，没有及时读取到主内存结果。

解决办法：强制线程每次读取该值的时候都去“主内存”中取值

4 synchronized

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个线程执行synchronized声明的代码块。还可以保证共享变量的内存可见性。同一时刻只有一个线程执行，这部分代码块的重排序也不会影响其执行结果。也就是说使用了synchronized可以保证并发的原子性，可见性，有序性。

4.1 解决可见性问题

JMM关于synchronized的两条规定：

线程解锁前（退出同步代码块时）：必须把自己工作内存中共享变量的最新值刷新到主内存中



子文里时需安从工内付中里机读取机的且（加块习群块定问一扣块）

做如下修改，在死循环中添加同步代码块

```
while (flag) {  
    synchronized (this) {  
    }  
}
```

synchronized实现可见性的过程

1. 获得互斥锁（同步获取锁）
2. 清空本地内存
3. 从主内存拷贝变量的最新副本到本地内存
4. 执行代码
5. 将更改后的共享变量的值刷新到主内存
6. 释放互斥锁

4.2 同步原理

synchronized的同步可以解决原子性、可见性和有序性的问题，那是如何实现同步的呢？

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

1. 普通同步方法，锁是当前实例对象this
2. 静态同步方法，锁是当前类的class对象
3. 同步方法块，锁是括号里面的对象

当一个线程访问同步代码块时，它首先是需要得到锁才能执行同步代码，当退出或者抛出异常时必须释放锁。

synchronized的同步操作主要是monitorenter和monitorexit这两个jvm指令实现的，先写一段简单的代码：

```
public void test2() {  
    synchronized (this) {  
    }  
}
```

在cmd命令行执行javac编译和javap -c Java 字节码的指令

```
javac Demo2Synchronized.java  
javap -c Demo2Synchronized.class
```

从结果可以看出，同步代码块是使用monitorenter和monitorexit这两个jvm指令实现的：

```
D:\itcast\class_thread\class_01\src\main\java\com\itheima\demo0525>javap -c SynchronizedDemo.class  
Compiled from "SynchronizedDemo.java"  
public class com.itheima.demo0525.SynchronizedDemo {  
    public com.itheima.demo0525.SynchronizedDemo();  
        Code:  
        0: aload_0  
        1: invokespecial #1          // Method java/lang/Object."<init>":()V  
        4: return  
  
    public synchronized void test1();  
        Code:  
        0: return  
  
    public void test2();  
        Code:  
        0: aload_0  
        1: dup  
        2: astore_1  
        3: monitorenter             // monitor进入，获取锁  
        4: aload_1  
        5: monitorexit              // monitor退出，释放锁  
        6: goto          14  
        9: astore_2  
       10: aload_1  
       11: monitorexit  
       12: aload_2  
       13: athrow  
       14: return  
    Exception table:  
        from    to    target type  
         4       6      9     any  
         9      12      9     any  
}
```

4.3 锁优化

synchronized是重量级锁，效率不高。但在jdk 1.6中对synchronize的实现进行了各种优化，使得它显得不是那么重了。jdk1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。



块内存，它们会随有兄子的数量而逐渐升级。

注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

4.3.1 自旋锁

线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒对CPU来说是一件负担很重的工作，势必会给系统的并发性能带来很大的压力。同时我们发现许多应用上面，对象锁的锁状态只会持续很短一段时间，为了这一段很短的时间频繁地阻塞和唤醒线程是非常不值得的。所以引入自旋锁。

所谓自旋锁，就是让该线程等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁。怎么等待呢？执行一段无意义的循环即可（自旋）。

自旋等待不能替代阻塞，虽然它可以避免线程切换带来的开销，但是它占用了处理器的时间。如果持有锁的线程很快就释放了锁，那么自旋的效率就非常好，反之，自旋的线程就会白白消耗掉处理的资源，它不会做任何有意义的工作，典型的占着茅坑不拉屎，这样反而会带来性能上的浪费。所以说，自旋等待的时间（自旋的次数）必须要有一个限度，如果自旋超过了定义的时间仍然没有获取到锁，则应该被挂起。

自旋锁在JDK 1.4.2中引入，默认关闭，但是可以使用-XX:+UseSpinning开启，在JDK1.6中默认开启。同时自旋的默认次数为10次，可以通过参数-XX:PreBlockSpin来调整；

如果通过参数-XX:preBlockSpin来调整自旋锁的自旋次数，会带来诸多不便。假如我将参数调整为10，但是系统很多线程都是等你刚刚退出的时候就释放了锁（假如你多自旋一两次就可以获取锁），你是不是很尴尬。于是JDK1.6引入自适应的自旋锁，让虚拟机会变得越来越聪明。

4.3.2 适应自旋锁

JDK 1.6引入了更加聪明的自旋锁，即自适应自旋锁。所谓自适应就意味着自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。它怎么做呢？线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能再次成功，那么它就会允许自旋等待持续的次数更多。反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源。

人/元/贝/测/云/越/不/越/庄/明，应/以/机/云/交/付/越/不/越/聪/明。

4.3.3 锁消除

为了保证数据的完整性，我们在进行操作时需要对这部分操作进行同步控制，但是在有些情况下，JVM检测到不可能存在共享数据竞争，这是JVM会对这些同步锁进行锁消除。锁消除的依据是逃逸分析的数据支持。

如果不存在竞争，为什么还需要加锁呢？所以锁消除可以节省毫无意义的请求锁的时间。变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是对于我们程序员来说这还不清楚么？我们会在明明知道不存在数据竞争的代码块前加上同步吗？但是有时候程序并不是我们所想的那样？我们虽然没有显示使用锁，但是我们在一些JDK的内置API时，如StringBuffer、Vector、HashTable等，这个时候会存在隐形的加锁操作。比如StringBuffer的append()方法，Vector的add()方法：

```
public void test(){
    Vector<Integer> vector = new Vector<Integer>();
    for(int i = 0 ; i < 10 ; i++){
        vector.add(i);
    }
    System.out.println(vector);
}
```

在运行这段代码时，JVM可以明显检测到变量vector没有逃逸出方法vectorTest()之外，所以JVM可以大胆地将vector内部的加锁操作消除。

4.3.4 锁粗化

在使用同步锁的时候，需要让同步块的作用范围尽可能小，仅在共享数据的实际作用域中才进行同步，这样做的目的是为了使需要同步的操作量尽可能缩小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。

在大多数的情况下，上述观点是正确的。但是如果一系列的连续加锁解锁操作，可能会导致不必要的性能损耗，所以引入锁粗化的概念。

锁粗话概念比较好理解，就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。如上面实例：vector每次add的时候都需要加锁操作，JVM检测到对同一个对象（vector）连续加锁、解锁操作，会合并一个更大范围的加锁、解锁操作，即加锁解锁操作会移到for循环之外。

轻量级锁的加锁解锁操作是需要依赖多次CAS原子指令的。而偏向锁只需要检查是否为偏向锁、锁标识为以及ThreadID即可，可以减少不必要的CAS操作。

4.3.6 轻量级锁

引入轻量级锁的主要目的是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁。轻量级锁主要使用CAS进行原子操作。

但是对于轻量级锁，其性能提升的依据是“对于绝大部分的锁，在整个生命周期内都是不会存在竞争的”，如果打破这个依据则除了互斥的开销外，还有额外的CAS操作，因此在有多线程竞争的情况下，轻量级锁比重量级锁更慢。

4.3.7 重量锁

重量级锁通过对象内部的监视器（monitor）实现，其中monitor的本质是依赖于底层操作系统的Mutex Lock（互斥锁）实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高。