01并发控制基本概念

由 黎金平创建, 最后修改于五月 21, 2019

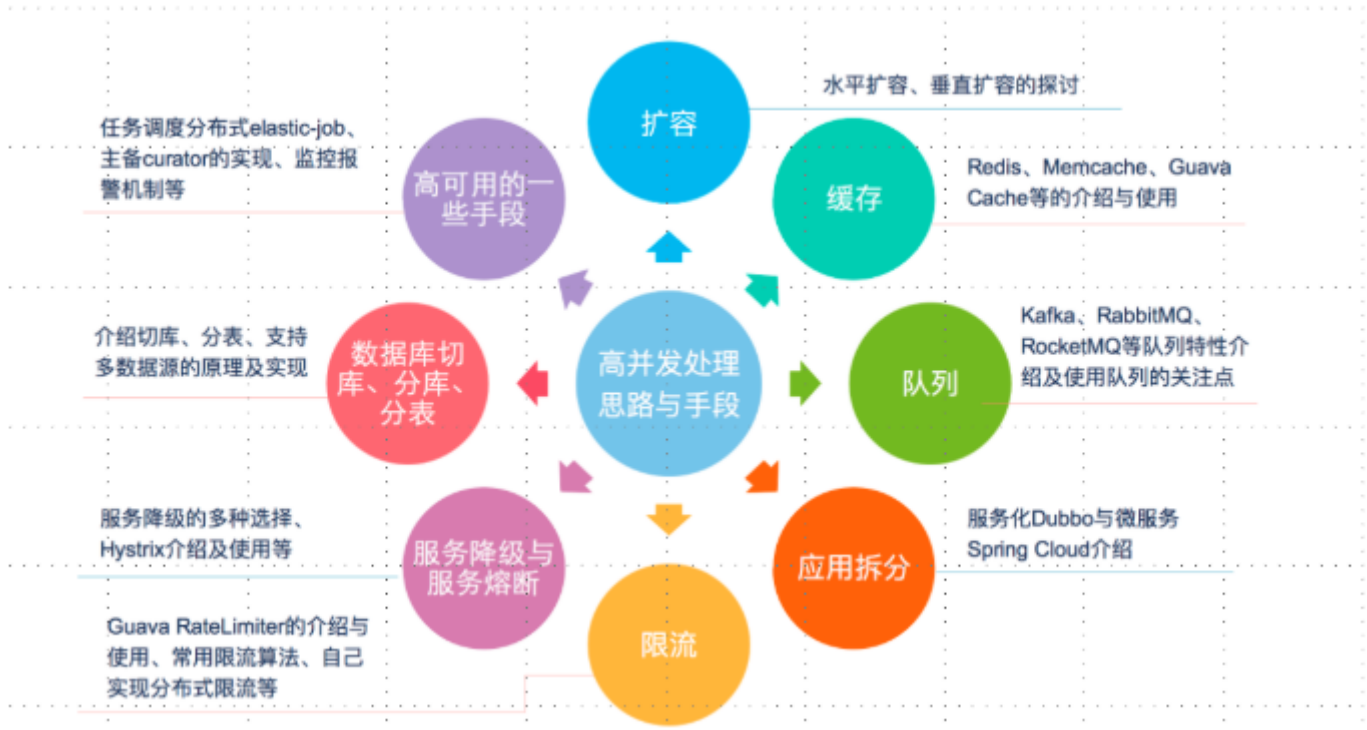
1并发相关基本概念

1.1并发

同时拥有两个或者多个线程，若程序在单核处理器上运行，多个线程将交替地换入或者换出内存，这些线程是同时“存在”的，每个线程都处于执行过程中的某个状态，高速切换感觉同时执行。如果运行多核处理器上，此时，程序中的每个线程将分配到一个处理器核上，因此可以真正的同时运行

1.2高并发

高并发（High Cuncurrency）通常是指，通过设计保证系统能够同时并发处理很多请求; 当我们说高并发时，我们谈的是是如何提高现有程序的性能，更多的是对高并发场景的一些解决方案、思路、手段等等。如果高并发处理不好，不仅仅降低了



1.3什么是线程安全问题

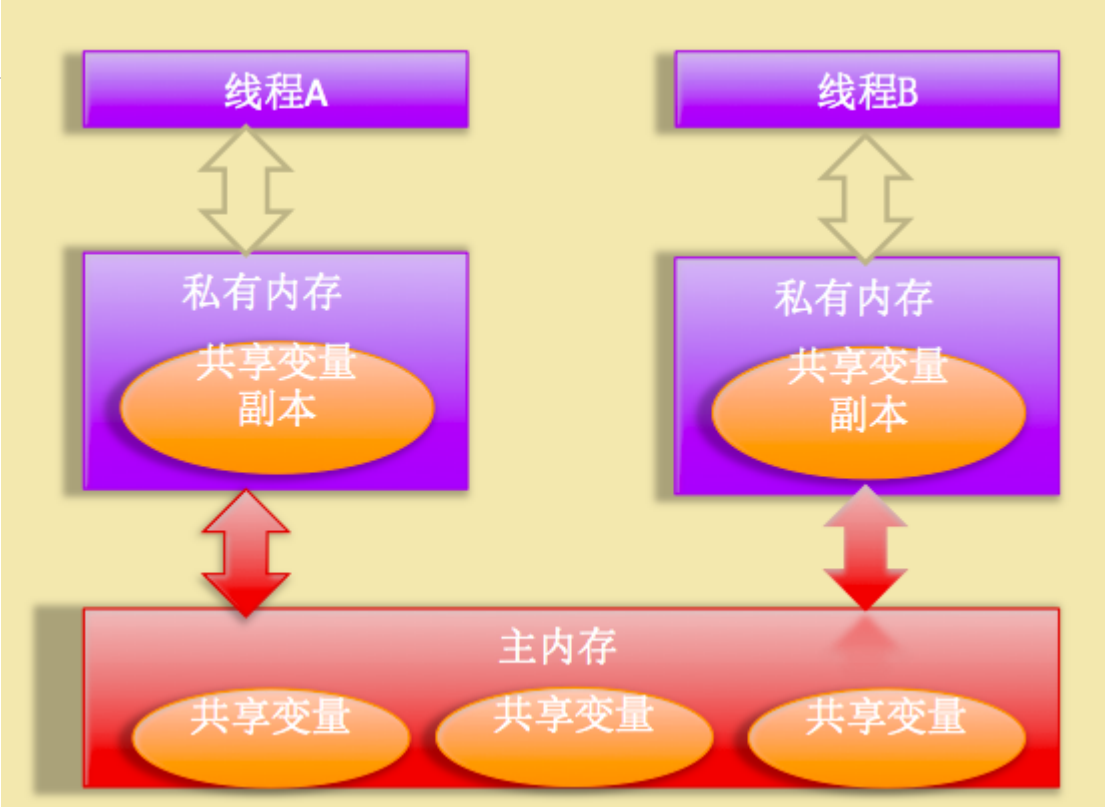
线程安全问题是指当多个线程同时读写一个状态变量，并且没有任何同步措施时候，导致脏数据或者其他不可预见的结果的问题。Java中首要的同步策略是使用Synchronized关键字，它提供了可重入的独占锁。

1.4什么是共享变量可见性问题

- 堆内存（Heap）：存放实例域, 静态域, 数组元素. **在线程间共享**.

- 栈内存（Stack）：存放局部变量, 方法定义参数和异常处理器参数.
- 可见性:一个线程对共享变量值的修改,能够及时地被其他线程看到
- 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

要谈可见性首先需要介绍下多线程处理共享变量时候的Java中内存模型。



Java内存模型规定了所有的变量都存放在主内存中，当线程使用变量时候都是把主内存里面的变量拷贝到了自己的工作空间或者叫做工作内存。

JMM两条规定：

1. 线程对共享变量的所有操作都必须在自己的工作内存中进行,不能直接从主内存中读取
2. 不同线程之间无法直接访问其他线程工作内存中的变量,线程间变量值的传递需要通过主内存来完成

如图是双核CPU系统架构，每核有自己的控制器和运算器，其中控制器包含一组寄存器和操作控制器，运算器执行算术逻辑运算，并且有自己的一级缓存，并且有些架构里面双核还有个共享的二级缓存。对应Java内存模型里面的工作内存，在实现上这里是指L1或者L2缓存或者自己cpu的寄存器。

- 线程首先从主内存拷贝共享变量到自己的工作空间
- 然后对工作空间里的变量进行处理
- 处理完后更新变量值到主内存

那么假如线程A和B同时去处理一个共享变量，会出现什么情况那?

线程A 先从主内存中获取共享变量（a=2），然后在自己本地内存中计算（a+2）但是还没有更新会主内存（结果可能目前存放在当前cpu的寄存器或者高速缓存）。
但此时B也从主内存获取（a=2），在本地内存改变（a+2）,处理后，线程A才把自己的处理结果更新到主内存或者缓存，可知线程B处理的并不是线程A处理后的结果，也就是说线程A处理后的变量值对线程B不可见，这就是共享变量的不可见性问题。

构成共享变量内存不可见原因是因为三步流程不是原子性操作，

共享变量可见性实现原理:

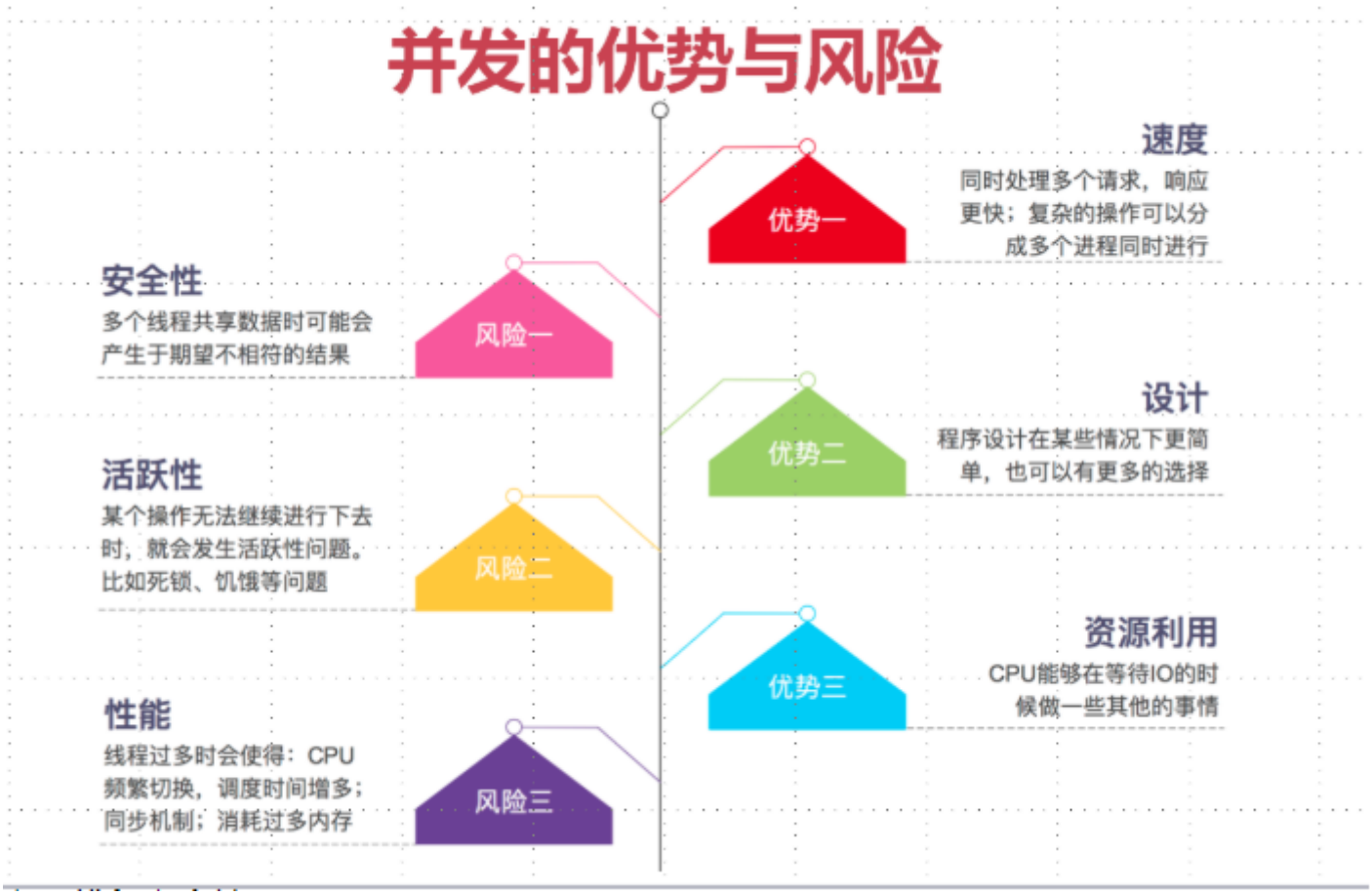
1. **把线程A中的工作内存中更新过的共享变量刷新到主内存中**
2. **将主内存中最新的共享变量的值更新到线程B中的工作内存中**

Java语言层面支持的可见性实现方式:

1. **synchronized;详见2.2可见性的讲解**
2. **volatile;详见2.2可见性的讲解**

3. **final也可以保证内存可见性，详见final**

1.5并发的优势和风险



重排序通常是编译器或运行时环境为了优化程序性能而采取的对指令进行重新排序执行的一种手段。重排序分为两类：编译期重排序和运行期重排序，分别对应编译时和运行时环境

编译期重排：编译源代码时，编译器依据对上下文的分析，对指令进行重排序，以之更适合于CPU的并行执行。

运行期重排：CPU在执行过程中，动态分析依赖部件的效能，对指令做重排序优化。

重排序可以保证最终执行的结果是与程序顺序执行的结果一致，并且只会对不存在数据依赖性的指令进行重排序，这个重排序在单线程下对最终执行结果是没有影响的，但是在多线程下就会存在问题。

一个例子:

demo1 折叠源码

```
int a = 1;(1)
int b = 2;(2)
int c= a + b;(3)
```

如上c的值依赖a和b的值,所以重排序后能够保证（3）的操作在(2)(1)之后，但是(1)(2)谁先执行就不一定了，这在单线程下不会存在问题，因为并不影响最终结果。

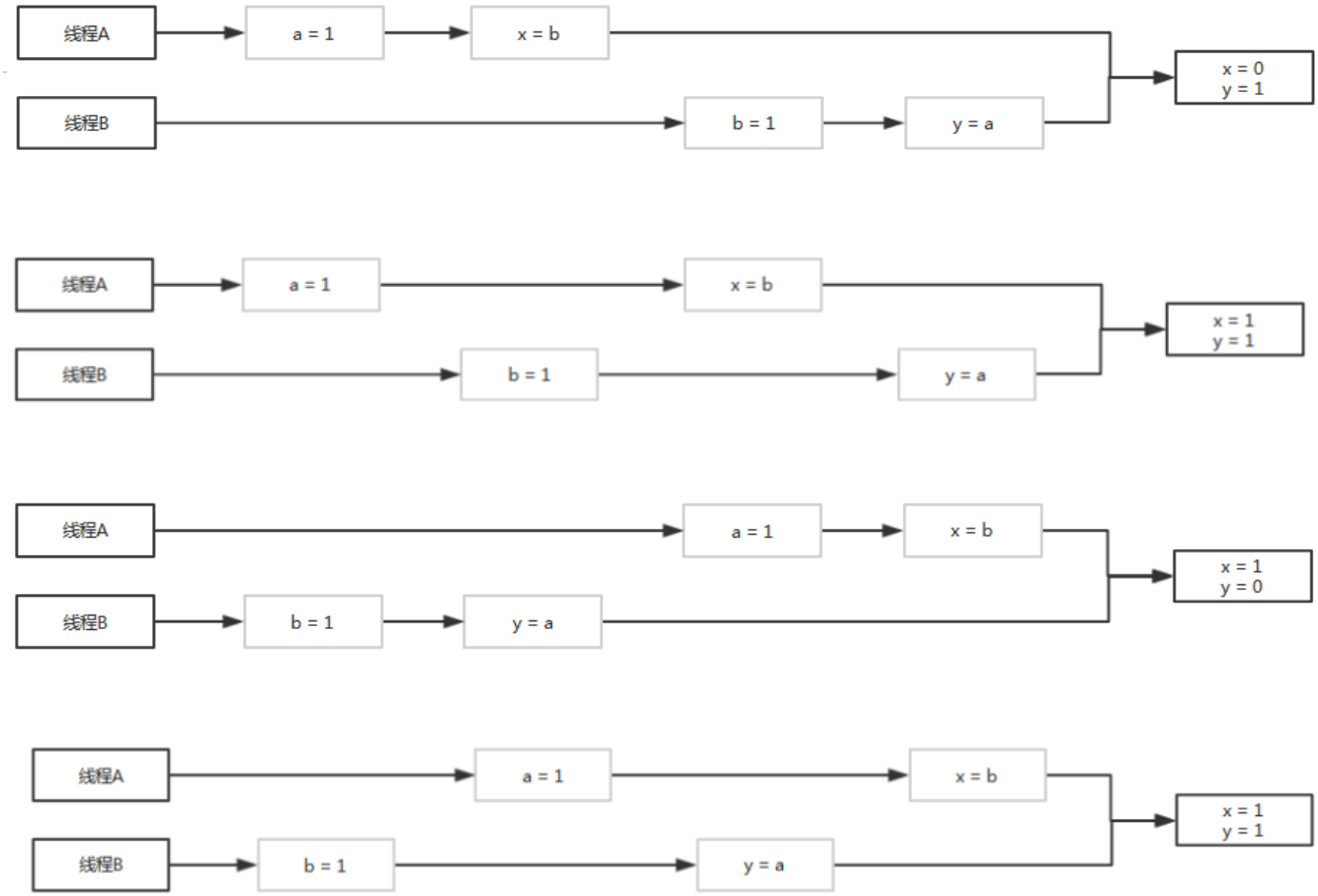
一个多线程例子:

demo2 折叠源码

```
public class Demo2 {
    private int x = 0;
private int y = 0;
private int a = 0;
private int b = 0;

   public void awrite() {

    a = 1;//(1)
    x = b;//(2)
}

public void bwrite() {

    b = 1;// (3)
    y = a;// (4)
}
    public static class AThread extends Thread{

        private Demo2 demo2;

        public AThread(Demo2 demo2) {

            this.demo2 = demo2;
        }

        @Override
        public void run() {
```

```
            super.run();

            this.demo2.awrite();
        }
    }
    public static class BThread extends Thread{

        private Demo2 demo2;

        public BThread(Demo2 demo2) {

            this.demo2 = demo2;
        }

        @Override
        public void run() {
            super.run();

            this.demo2.bwrite();
        }
    }

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

        Demo2 demo2 = new Demo2();

        for (int i = 0; i < 100000; i++) {
            AThread aThread = new AThread(demo2);
            BThread bThread = new BThread(demo2);

            aThread.start();
            bThread.start();

            aThread.join();
            bThread.join();
            if (demo2.x == 0 && demo2.y == 0) {
                System.out.println("resort");
            }

            demo2.x = demo2.y = demo2.a = demo2.b = 0;

        }

        System.out.println("end");
    }
}
```

如果不进行重排序，程序的执行顺序有四种可能：



如代码由于(1)(2)(3)(4) 之间不存在依赖，所以写线程(3)(4)可能被重排序为先执行（4）在执行（3),那么执行（4）后，A线程可能已经执行了（1）操作，并且在（2）执行前开始执行（3）操作，这时候x,y都为0.


```
"C:\Program Files\Java\jdk1.7.0_79\bin\java.exe" ...
resort
resort
resort
resort
resort
end

Process finished with exit code 0
```

解决：使用volatile 修饰四个变量可以避免重排序，或者使用synchronized修饰两个写方法。

1.7happens-before

下面是happens-before原则规则：

1.7.1 程序次序原则（Program Order Rule）

指在**一个线程**内，按照程序代码的控制流顺序（考虑到循环等结构），编写在前面的操作先行发生于后面的操作。需要注意这里注重的是单线程内。eg:

demo3

```
user.setName("test");
user.getName();
// 在一个线程内，上述代码中的set方法要先行于get方法执行
```

1.7.2监视器锁定原则（Monitor Lock Rule）

对于同一个锁，一个unlock操作要先行于下一次的lock操作

1.7.3volatile 原则（Volatile Variable Rule）

对一个volatile修饰变量的写操作要先行发生于下一次的读操作

1.7.4 传递性（Transitivity）

如果A先行发生于B，B先行发生于C，那么A先行发生于C。

1.7.5线程启动原则（Thread Start Rule）

Thread对象的start()方法先行发生于此线程的每一个操作；等同于：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作

1.7.6 线程中断原则（Thread Interruption Rule）

对线程interrupt方法的调用先行发生于被中断线程的代码检测到中断事件的发生。

注：Java的中断是一种协作机制，中断操作并不能直接中断一个线程，仅仅只是标识该线程接收到一个中断请求，至于实际什么时候中断，要中断线程自己检测处理。

1.7.7线程终止原则（Thread Termination Rule）

线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行；等同于：假定线程A在执行的过程中，通过指定ThreadB.join()等待线程B终止，那么线程B在终止之前对共享变量的修改在线程A等待返回后可见

1.7.8 对象终结原则（Finalize Rule）

一个对象的初始化完成先行发生于其finalize()方法的开始。

注：如果对象覆盖了finalize()方法，那么在GC的第一次标记后，会放入一个叫做F-Queue的队列中，由一个Finalizer线程触发该方法的调用。

指望利用它来进行资源回收

上面八条是原生Java满足Happens-before关系的规则，但是我们可以对他们进行推导出其他满足happens-before的规则：

1. 将一个元素放入一个线程安全的队列的操作Happens-Before从队列中取出这个元素的操作
2. 将一个元素放入一个线程安全容器的操作Happens-Before从容器中取出这个元素的操作
3. 在CountDownLatch上的倒数操作Happens-Before CountDownLatch#await()操作
4. 释放Semaphore许可的操作Happens-Before获得许可操作
5. Future表示的任务的所有操作Happens-Before Future#get()操作
6. 向Executor提交一个Runnable或Callable的操作Happens-Before任务开始执行操作

这里再说一遍happens-before的概念：如果两个操作不存在上述（前面8条 + 后面6条）任一一个happens-before规则，那么这两个操作就没有顺序的保障，JVM可以对这两个操作进行重排序。如果操作A happens-before操作B，那么操作A在内存上所做的操作对操作B都是可见的。

happens-before原则demo（这段代码的作用是两个线程间隔打印出 0 – 100 的数字）：

demo4 折叠源码

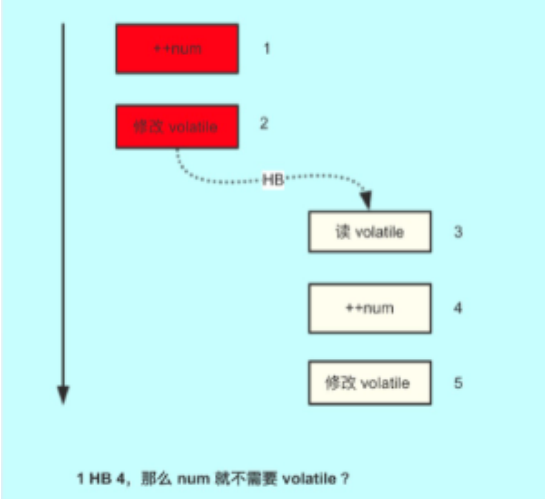
```
public class Demo4 {

    static int num = 0;
    static volatile boolean flag = false;

    public static void main(String[] args){
        Thread t1 = new Thread("t1"){
            @Override
            public void run() {
                for (; 100 > num; ) {
                    if (!flag && (num == 0 || ++num % 2 == 0)) {
                        System.out.println(Thread.currentThread().getName() + ":" +
                        flag = true;
                    }
                }
            }
        };

        Thread t2 = new Thread("t2"){
            @Override
            public void run() {
                for (; 100 > num; ) {
                    if (flag && (++num % 2 != 0)) {
                        System.out.println(Thread.currentThread().getName() + ":" +
                        flag = false;
                    }
                }
            }
        };
        t1.start();
        t2.start();
    }
}
```

demo4_description 折叠源码

熟悉并发编程的同学肯定要说了，这个 num 变量没有使用 **volatile**，会有可见性问题，即：t1 线程更新了 n

刚开始本人也是这么认为的，但最近通过研究 HB 规则，我发现，去掉 num 的 **volatile** 修饰也是可以的

我们分析一下，如下图：



我们分析这个图：

1. 首先，红色和淡黄色表示不同的线程操作。
2. 红色线程对 num 变量做 ++，然后修改了 volatile 变量，这个是符合 **程序次序规则**的。也就是 1 HB 2.
3. 红色线程对 volatile 的写 HB 黄色线程对 volatile 的读，也就是 2 HB 3.
4. 黄色线程读取 volatile 变量，然后对 num 变量做 ++，符合 **程序次序规则**，也就是 3 HB 4.
5. 根据传递性规则，1 肯定 HB 4. 所以，**1 的修改对 4来说都是可见的。**

注意：HB 规则保证上一个操作的结果对下一个操作都是可见的。

所以，上面的小程序中，线程 A 对 num 的修改，线程 B 是完全感知的 —— 即使 num 没有使用 volatile 修饰。

这样，我们就借助 HB 原则实现了对一个变量的同步操作，也就是在多线程环境中，保证了并发修改共享变量的安全性。并且没有对这个变量使用 Java 的原语：volatile 和 synchronized ；

这可能看起来不安全（实际上安全），也好像不太容易理解。因为这一切都是 HB 底层的 缓存一致性协议（cache protocol） 和 内存屏障（memory barrier ）实现的。

线程终止规则demo:

demo5 折叠源码

```
private static int a = 1;

public static void main(String[] args){
    final Thread tb = new Thread("tb"){
        @Override
        public void run() {
            a = 2;
        }
    };
    Thread ta = new Thread("ta"){
        @Override
```

```
        public void run() {
            try {
                tb.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 输出结果会是什么呢?
            System.out.println( Thread.currentThread().getName() + ":" + a);
        }
    };
    ta.start();
    tb.start();
}
```

线程start规则demo:

demo6 折叠源码

```
public class Demo6 {
    private static int a = 1;

    public static void main(String[] args){
        final Thread tb = new Thread("tb"){
            @Override
            public void run() {
                // a的值是多少?
                System.out.println(Thread.currentThread().getName() + ":" + a);
            }
        };
        Thread ta = new Thread(){
            @Override
            public void run() {
                tb.start();
                a = 2;
            }
        };
        ta.start();
    }
}
```

这demo5,demo6两个操作，也可以保证变量 a 的可见性。

确实有点颠覆之前的观念。之前的观念中，如果一个变量没有被 volatile 修饰或 final 修饰，那么他在多线程下的读写肯定是不安全的 —— 因为会有缓存，导致读取到的不是最新的。

然而，通过借助 HB，我们可以实现

总结：一个操作“时间上的先发生”不代表这个操作先行发生；一个操作先行发生也不代表这个操作在时间上是先发生的（重排序的出现）。
时间上的先后顺序对先行发生没有太大的关系，所以衡量并发安全问题的时候不要受到时间顺序的影响，一切以先行发生原则为准。

归纳为一句话：happens-before规则保证了单线程和正确同步的多线程的执行结果不会被改变。

概念：当多个线程访问某个类时，不管运行时环境采用**何种调度方式**或者这些进程将如何交替进行，并且在主调代码中**不需要任何额外的同步或协同**，这个类都能**表现出正确的行为**，那么就称这个类是线程安全的。

线程安全主要体现在三个方面：原子性、可见性、有序性

2.1原子性

原子是世界上的最小单位，具有不可分割性；提供了互斥访问，**同一时刻**只能有**一个线程**对他进行操作

eg:比如 a=0；（a非long和double类型）这个操作是不可分割的，那么我们说这个操作时原子操作。再比如：a++；这个操作实际是a = a + 1；是可分割的，所以他不是一个原子操作。非原子操作都会存在线程安全问题，需要我们使用同步技术（sychronized）来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法。比如：AtomicInteger、AtomicLong、AtomicReference等

说到原子性，一共有两个方法，一个是JDK中已经提供好的Atomic包，他们均使用了CAS完成线程的原子性操作，另一个是使用锁的机制来处理线程之间的原子性。锁包括：synchronized、Lock

原子性demo:

demo7 折叠源码

```
public class Demo7 {
        public int sharedState;
        public void nonSafeAction() {
            while (sharedState < 100000) {
                int former = sharedState++;
                int latter = sharedState;
                if (former != latter - 1) {
                    System.out.printf("former is " +
                            former + ", " + "latter is " + latter);
                }
            }
        }
        public static void main(String[] args) throws InterruptedException {
            final Demo7 demo7 = new Demo7();
            Thread threadA = new Thread(){
                @Override
                public void run(){
                    demo7.nonSafeAction();
                }
            };
            Thread threadB = new Thread(){
                @Override
                public void run(){
                    demo7.nonSafeAction();
                }
            };
            threadA.start();
            threadB.start();
            threadA.join();
            threadB.join();
        }
```

```
    }
```

结果：

```
"C:\Program Files\Java\jdk1.7.0_79\bin\java.exe" ...
former is 20596, latter is 20612
Process finished with exit code 0
```

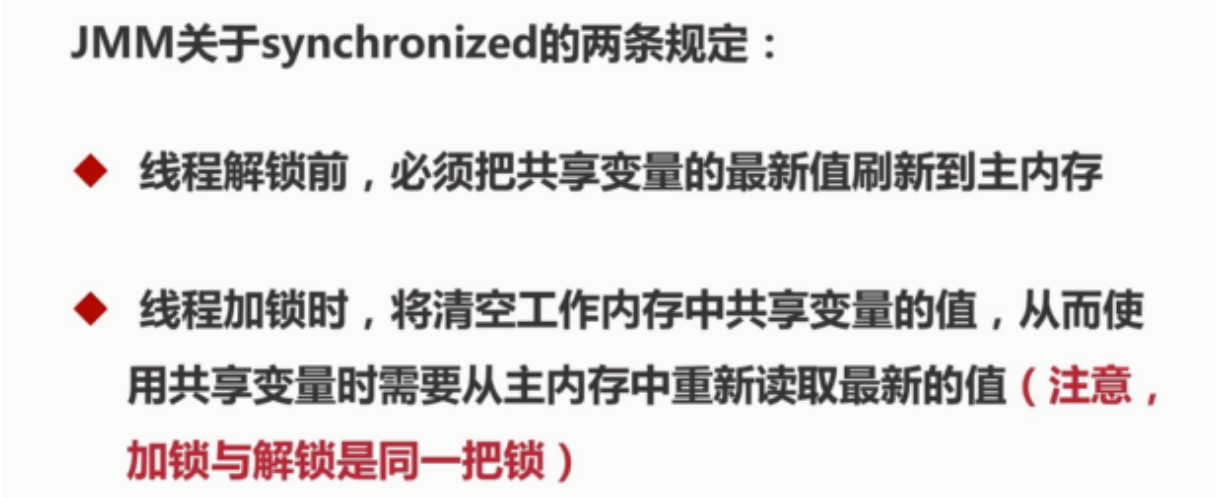
解决：将两次的赋值过程使用synchronized保护起来，使用this作为互斥单元，即可避免别的线程去修改sharedState

```
synchronized (this) {
int former = sharedState ++;
int latter = sharedState;
// …
}
```

2.2可见性

一个内存对主内存的修改可以及时地被其他线程观察到

可见性-synchronized

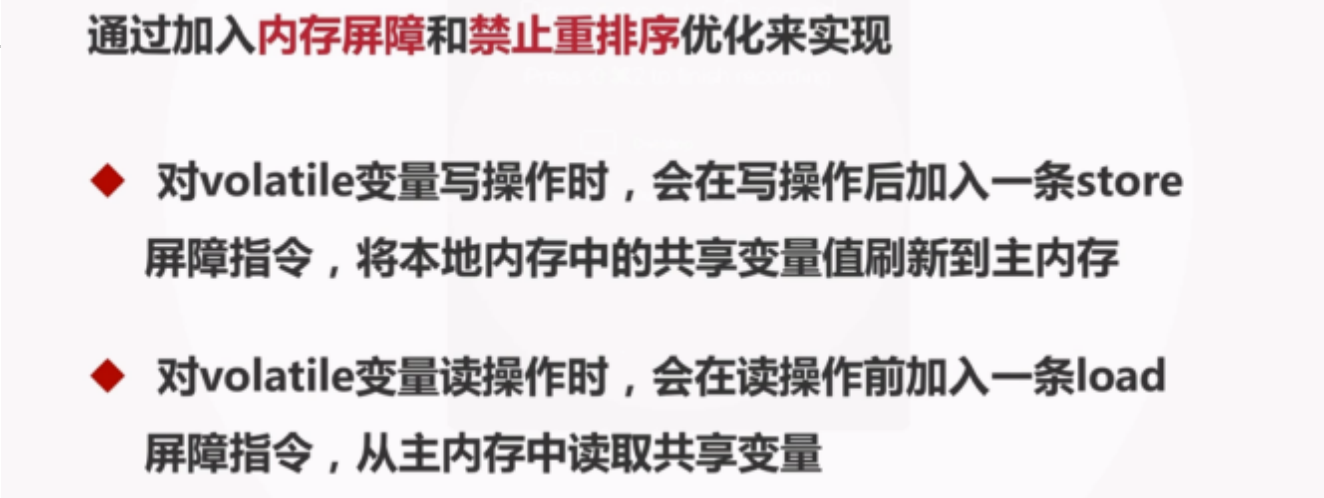


synchronized:可以同时保证**可见性**和**原子性**;但是会引起线程上下文切换和线程调度

导致共享变量在线程间不可见的原因：

1. 线程的交叉执行（保证**原子性**，使用**synchronized**关键字）
2. 重排序结合线程交叉执行（**原子性**）
3. 共享变量更新后的值没有在工作内存与主内存间及时更新（**可见性**）

可见性-Volatile



Volatile:号称轻量级的synchroinzed; 只保证变量值的**可见性**；不会引起线程上下文切换和线程调度；还用来解决重排序问题

不能保证volatile变量复合操作的原子性；eg:

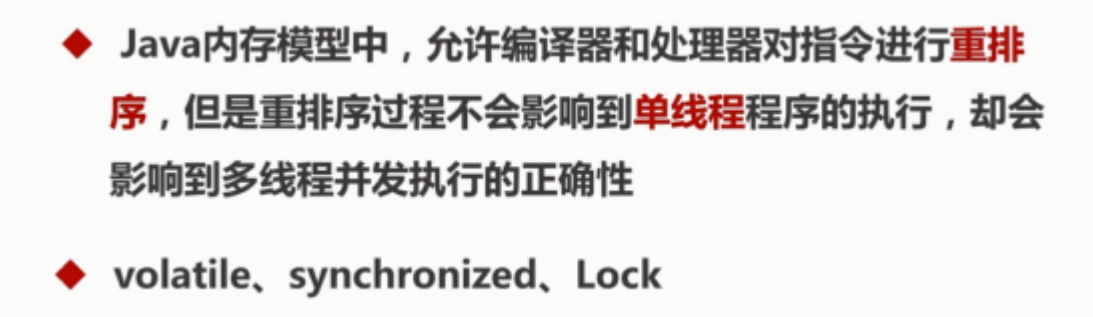
```
int num = 0;
num++;//++操作非原子操作，分3步执行；等同于num = num + 1;
保证原子性的方法： synchronized关键字、ReentrantLock可传入锁对象、AtomicInterger对象
```

Synchronized和Volatile比较

	Synchronized	Volatile
保证可见性	是	是
保证原子性	是	-
阻塞线程	是	否
使用范围	变量、方法	变量
禁止指令重排序	否	是

2.3有序性

一个线程观察其他线程的指令执行顺序，由于存在指令重排，该观察结果一般杂乱无序



3.锁

3.1乐观锁

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write_condition机制的其实都是提供的乐观锁;

乐观锁直到提交的时候才去锁定，所以不会产生任何锁和死锁

3.2悲观锁

悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁

数据库中实现是对数据记录进行操作前，先给记录加排它锁，如果获取锁失败，则说明数据正在被其他线程修改，则等待或者抛出异常。如果加锁成功，则获取记录，对其修改，然后事务提交后释放排它锁。
一个例子：`select * from 表 where .. for update;`

悲观锁是先加锁再访问策略，处理加锁会让数据库产生额外的开销，还有增加产生死锁的机会，另外在多个线程只读情况下不会产生数据不一致行问题，没必要使用锁，只会增加系统负载，降低并发性，因为当一个事务锁定了该条记录，其他读该记录的事务只能等待。

比较：两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适

3.3公平锁

公平锁表示线程获取锁的顺序是按照线程加锁的时间多少来决定的，也就是最早加锁的线程将最早获取锁，也就是先来先得的FIFO顺序

3.4非公平锁

非公平锁则运行闯入，也就是先来不一定先得到锁

非公平锁 ReentrantLock pairLock = new ReentrantLock(false);

10个并发的公平锁：Semaphore semaphore = new Semaphore(10,true);

10个并发的非公平锁：Semaphore semaphore = new Semaphore(10,false);

如果构造函数不传递参数，则默认是非公平锁。

在没有公平性需求的前提下**尽量使用非公平锁**，因为**公平锁会带来性能开销**。
假设线程A已经持有了锁，这时候线程B请求该锁将会被挂起，当线程A释放锁后，假如当前有线程C也需要获取该锁，如果采用非公平锁方式，则根据线程调度策略线程B和C两者之一可能获取锁，这时候不需要任何其他干涉，如果使用公平锁则需要把C挂起，让B获取当前锁

4.示例代码

concurrencydemo.zip

无标签