Java 并发编程之美：并发编程基础晋级篇

借用 Java 并发编程实践中的话；编写正确的程序并不容易，而编写正常的并发程序就更难了;相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的并发编程必备基础晋级篇，通过通俗易懂的方式来和大家聊聊多线程并发编程中涉及到的高级基础知识（建议先阅读《Java 并发编程之美：线程相关的基础知识》），具体内容如下：

- 什么是多线程并发和并行。
- 什么是线程安全问题。
- 什么是共享变量的内存可见性问题。
- 什么是 Java 中原子性操作。
- 什么是 Java 中的 CAS 操作，AtomicLong 实现原理
- 什么是 Java 指令重排序。
- Java 中 Synchronized 关键字的内存语义是什么。
- Java 中 Volatile 关键字的内存语义是什么。
- 什么是伪共享，为何会出现，以及如何避免。
- 什么是可重入锁、乐观锁、悲观锁、公平锁、非公平锁、独占锁、共享锁。

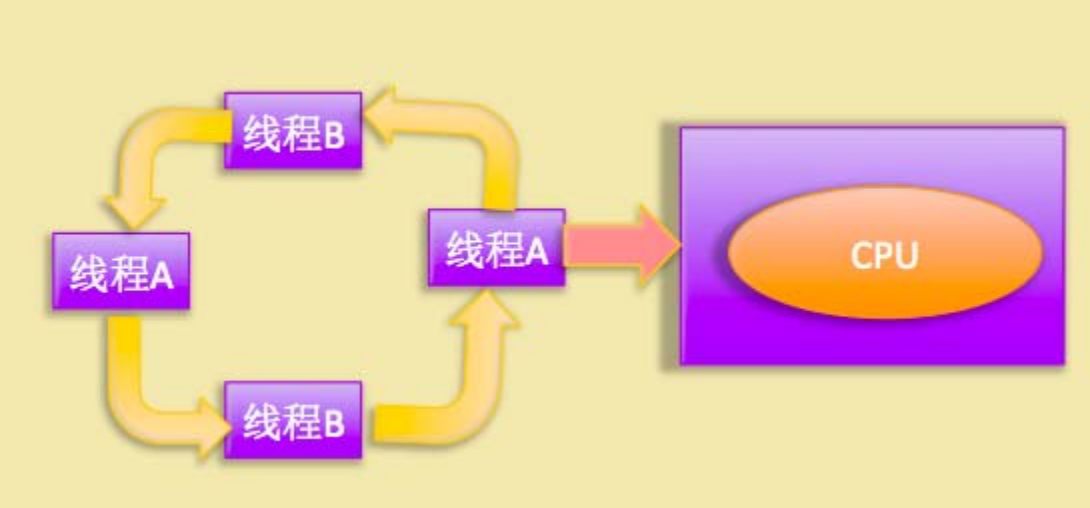
多线程并发与并行

首先要澄清并发和并行的概念，并发是指同一个时间段内多个任务同时都在执行，并且都没有执行结束；而并行是说在单位时间内多个任务同时在执行；并发任务强调在一个时间段内同时执行，而一个时间段有多个单位时间累积而成，所以说并发的多个任务在单位时间内不一定同时在执行。

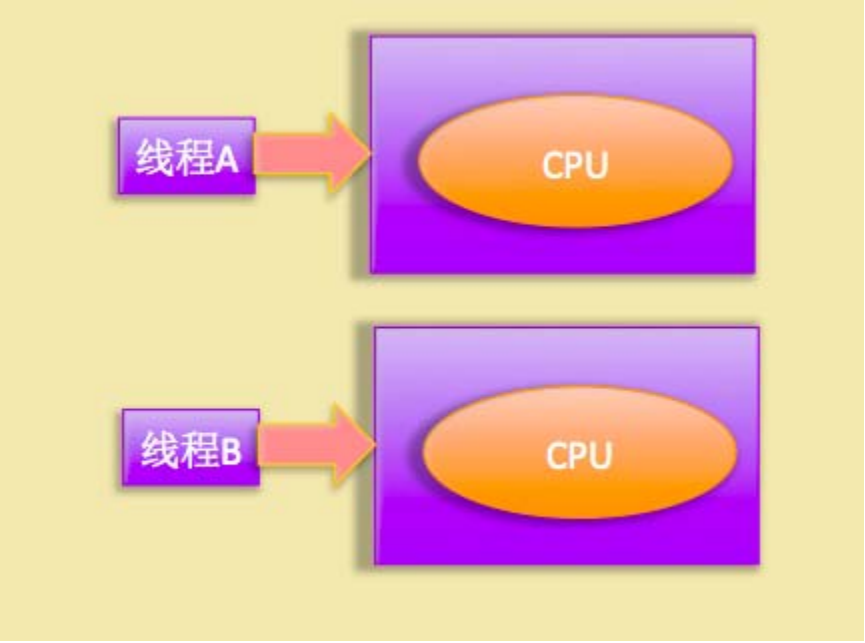
在单个 CPU 的时代多个任务同时运行都是并发，这是因为 CPU 同时只能执行一个任务，单个 CPU 时代多任务是共享一个 CPU 的，当一个任务占用 CPU 运行时候，其它任务就会被挂起，当占用 CPU 的任务时间片用完后，会把 CPU 让给其它任务来使用，所以在单 CPU 时代多线程编程的意义不大，并且线程间频繁的上下文切换还会带来开销。

如下图单个 CPU 上运行两个线程，可知线程 A 和 B 是轮流使用 CPU 进行任务处理的，也就是同时 CPU 只在执行一个线程上面的任务，当前线程 A 的时间片用完后会进行线程上

下文切换，也就是保存当前线程的执行线程，然后切换线程 B 占用 CPU 运行任务。



如下图双 CPU 时候，线程 A 和线程 B 在自己的 CPU 上执行任务，实现了真正的并行运行。

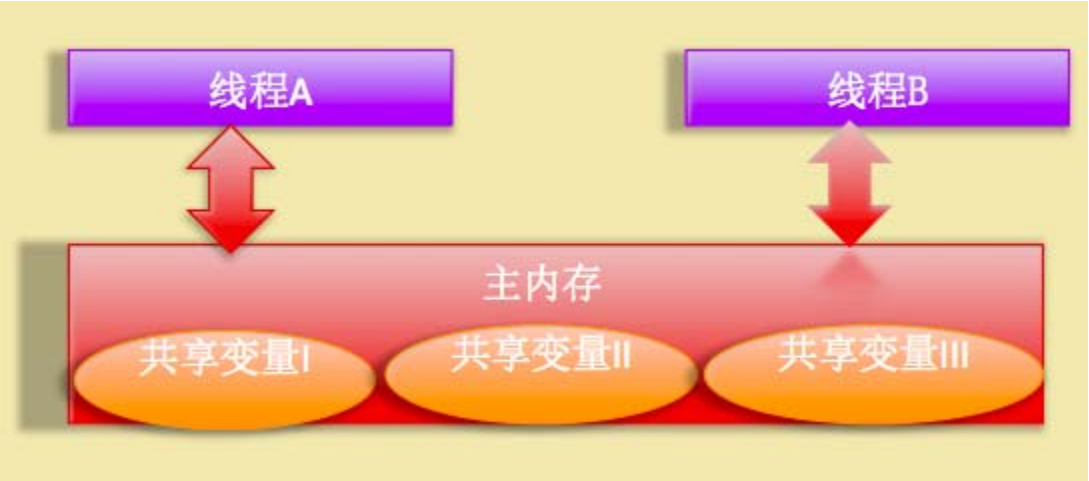


而在多线程编程实践中线程的个数往往多于 CPU 的个数，所以平时都是称多线程并发编程而不是多线程并行编程。

线程安全问题

谈到线程安全问题不得不先说说什么是共享资源，所谓共享资源是说多个线程都可以去访问的资源。

线程安全问题是指当多个线程同时读写一个共享资源并且没有任何同步措施的时候，导致脏数据或者其它不可预见的结果的问题。



如上图，线程 A 和线程 B 可以同时去操作主内存中的共享变量，是不是说多个线程共享了资源，都会产生线程安全问题呢？答案是否定的，如果多个线程都是只读取共享资源，而不去修改，那么就不会存在线程安全问题。

只有当至少一个线程修改共享资源时候才会存在线程安全问题。最典型的就是计数器类的实现，计数 count 本身是一个共享变量，多个线程可以对其进行增加一，如果不使用同步的话，由于递增操作是获取 -> 加1 -> 保存三步操作，所以可能导致导致计数不准确，如下表：

	t1	t2	t3	t4
线程A	从内存读取count值到本线程	递增本线程count的值	写回主内存	
线程B		从内存读取count值到本线程	递增本线程count的值	写回主内存

假如当前 count=0，t1 时刻线程 A 读取了 count 值到本地变量 countA。

然后 t2 时刻递增 countA 值为1，同时线程 B 读取 count 的值0放到本地变量 countB 值为 0（因为 countA 还没有写入主内存）。

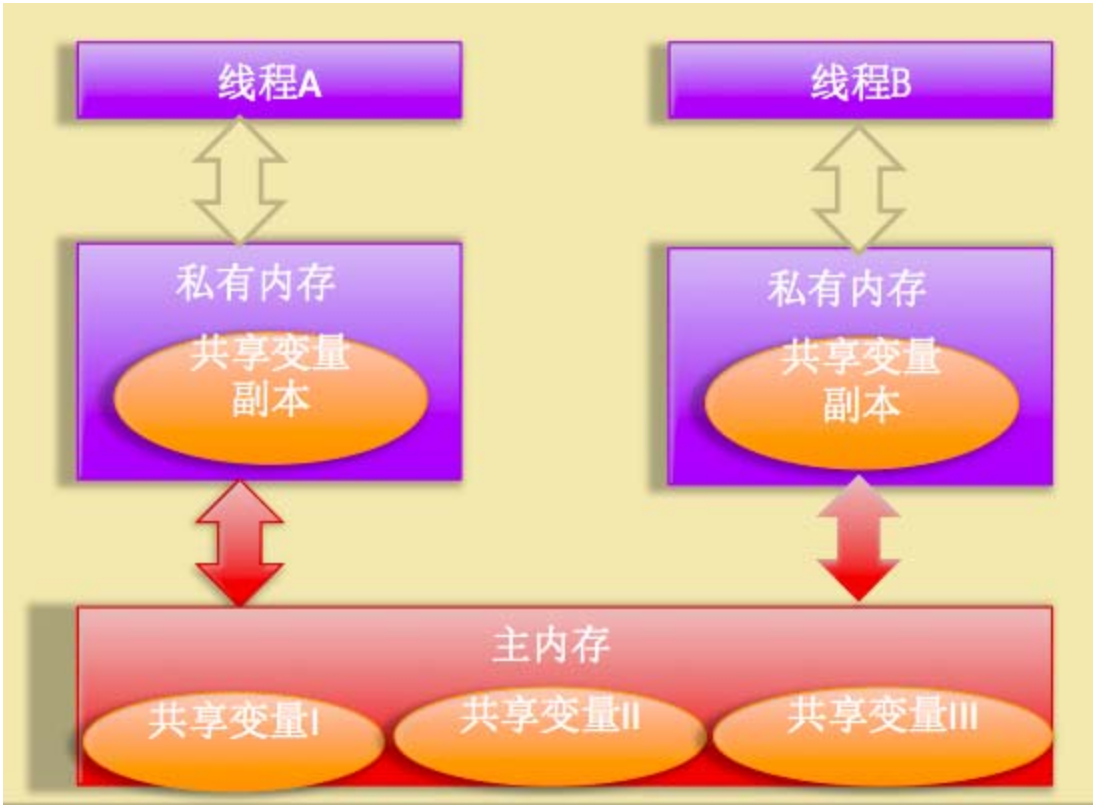
t3 时刻线程 A 才把 countA 为1的值写入主内存，至此线程 A 一次计数完毕，同时线程 B 递增 CountB 值为1。

t4 时刻线程 B 把 countB 值1写入内存，至此线程 B 一次计数完毕。

先不考虑内存可见性问题，明明是两次计数哇，为啥最后结果还是1而不是2呢？其实这就是共享变量的线程安全问题。那么如何解决？这就需要在线程访问共享变量时候进行适当的同步，Java 中首屈一指的是使用关键字 Synchronized 进行同步，这个下面会有具体介绍。

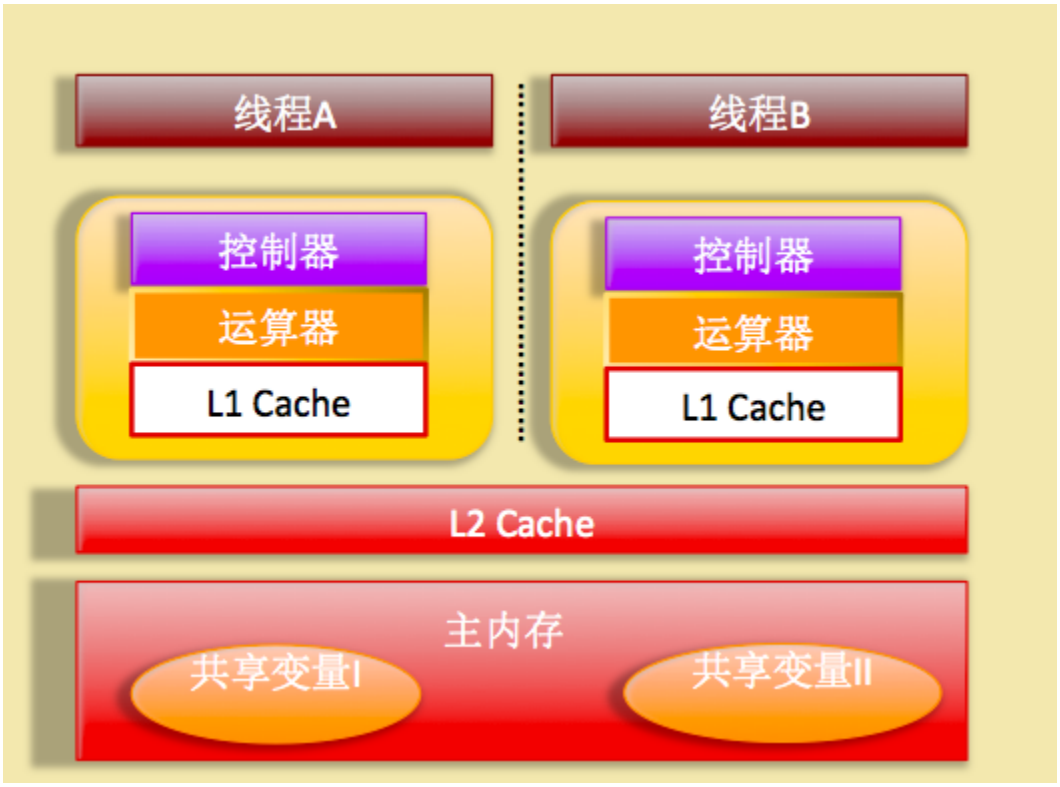
共享变量的内存可见性问题

要谈内存可见性首先需要介绍下 Java 中多线程下处理共享变量时候的内存模型。



如上图，Jav a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。

Java 内存模型是个抽象的概念，那么在实际实现中什么是线程的工作内存呢?



如上图是双核 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。

那么 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 CPU 的寄存器。

假如线程 A 和 B 同时去处理一个共享变量，会出现什么情况呢?

使用上图 CPU 架构，假设线程 A和 B 使用不同 CPU 进行去修改共享变量 X，假设 X 的初始化为0，并且当前两级 Cache 都为空的情况，具体看下面分析:

- 假设线程 A 首先获取共享变量 X 的值，由于两级 Cache 都没有命中，所以到主内存加载了 X=0，然后会把 X=0 的值缓存到两级缓存，假设线程 A 修改 X 的值为1，然后写入到两级 Cache，并且刷新到主内存（注：如果没刷新会主内存也会存在内存不可见问题）。这时候线程 A 所在的 CPU 的两级 Cache 内和主内存里面 X 的值都是 1；
- 然后假设线程 B 这时候获取 X 的值，首先一级缓存没有命中，然后看二级缓存，二级缓存命中了，所以返回 X=1；然后线程 B 修改 X 的值为2；然后存放到线程2所在的一级 Cache 和共享二级 Cache，最后更新主内存值为2；
- 然后假设线程 A 这次又需要修改 X 的值，获取时候一级缓存命中获取 X=1，到这里问题就出现了，明明线程 B 已经把 X 的值修改为了2，为啥线程 A 获取的还是1呢？这就是共享变量的内存不可见问题，也就是线程 B 写入的值对线程 A 不可见。

那么对于共享变量内存不可见问题如何解决呢？ Java 中首屈一指的 Synchronized 和 Volatile 关键字就可以解决这个问题，下面会有讲解。

Java 中 Synchronized 关键字

Synchronized 块是 Java 提供的一种原子性内置锁，Java 中每个对象都可以当做一个同步锁的功能来使用，这些 Java 内置的使用者看不到的锁被称为内部锁，也叫做监视器锁。

线程在进入 Synchronized 代码块前会自动尝试获取内部锁，如果这时候内部锁没有被其他线程占有，则当前线程就获取到了内部锁，这时候其它企图访问该代码块的线程会被阻塞挂起。

拿到内部锁的线程会在正常退出同步代码块或者异常抛出后或者同步块内调用了该内置锁资源的 wait 系列方法时候释放该内置锁；内置锁是排它锁，也就是当一个线程获取这个锁后，其它线程必须等待该线程释放锁才能获取该锁。

上一节讲了多线程并发修改共享变量时候会存在内存不可见的问题，究其原因是因为 Java 内存模型中线程操作共享变量时候会从自己的工作内存中获取而不是从主内存获取或者线程写入到本地内存的变量没有被刷新会主内存。

下面讲解下 Synchronized 的一个内存语义，这个内存语义就可以解决共享变量内存不可见性问题。

线程进入 Synchronized 块的语义是会把在 Synchronized 块内使用到的变量从线程的工作内存中清除，在 Synchronized 块内使用该变量时候就不会从线程的工作内存中获取了，

而是直接从主内存中获取；退出 Synchronized 块的内存语义是会把 Synchronized 块内对共享变量的修改刷新到主内存。对应上面一节讲解的假如线程在 Synchronized 块内获取变量 X 的值，那么线程首先会清空所在的 CPU 的缓存，然后从主内存获取变量 X 的值；当线程修改了变量的值后会把修改的值刷新回主内存。

其实这也是加锁和释放锁的语义，当获取锁后会清空本地内存中后面将会用到的共享变量，在使用这些共享变量的时候会从主内存进行加载；在释放锁时候会刷新本地内存中修改的共享变量到主内存。

除了可以解决共享变量内存可见性问题外，Synchronized 经常被用来实现原子性操作，另外注意，Synchronized 关键字会引起线程上下文切换和线程调度的开销。

Java 中 Volatile 关键字

上面介绍了使用锁的方式可以解决共享变量内存可见性问题，但是使用锁太重，因为它会引起线程上下文的切换开销，对于解决内存可见性问题，Java 还提供了一种弱形式的同步，也就是使用了 volatile 关键字。

一旦一个变量被 volatile 修饰了，当线程获取这个变量值的时候会首先清空线程工作内存中该变量的值，然后从主内存获取该变量的值；当线程写入被 volatile 修饰的变量的值的时候，首先会把修改后的值写入工作内存，然后会刷新到主内存。这就保证了对一个变量的更新对其它线程马上可见。

下面看一个使用 volatile 关键字解决内存不可见性的一个例子，如下代码的共享变量 value 是线程不安全的，因为它没有进行适当同步措施。

```
public class ThreadNotSafeInteger {

    private int value;

    public int get() {
        return value;
    }

    public void set(int value) {
        this.value = value;
    }
}
```

首先看下使用 synchronized 关键字进行同步方式如下：

```
public class ThreadSafeInteger {

    private int value;
```

```
    public synchronized int get() {
        return value;
    }

    public synchronized  void set(int value) {
        this.value = value;
    }
}
```

然后看下使用 volatile 进行同步如下：

```
public class ThreadSafeInteger {

    private volatile int value;

    public int get() {
        return value;
    }

    public void set(int value) {
        this.value = value;
    }
}
```

这里使用 synchronized 和使用 volatile 是等价的，都解决了共享变量 value 的内存不可见性问题；但是前者是独占锁，同时只能有一个线程调用 get() 方法，其它调用线程会被阻塞；并且会存在线程上下文切换和线程重新调度的开销；而后者是非阻塞算法，不会造成线程上下文切换的开销。

这里使用 synchronized 和使用 volatile 是等价的，但是并不是所有情况下都是等价的，这是因为 volatile 虽然提供了可见性保证，但是并没有保证操作的原子性。

那么一般什么时候才使用 volatile 关键字修饰变量呢？

- 当写入变量值时候不依赖变量的当前值。因为如果依赖当前值则是获取 -> 计算 -> 写入操作，而这三步操作不是原子性的，而 volatile 不保证原子性。
- 读写变量值时候没有进行加锁。因为加锁本身已经保证了内存可见性，这时候不需要把变量声明为 volatile。

另外变量被声明为 volatile 还可以避免重排序的发生，这个后面会讲到。

Java 中原子性操作

所谓原子性操作是指当执行一系列操作时候，这些操作那么全部被执行，那么全部不被执行，不存在只执行其中一部分的情况。

在设计计数器时候一般都是先读取当前值，然后+1，然后更新，这个过程是读 -> 改 -> 写的过程，如果不能保证这个过程是原子性，那么就会出现线程安全问题。如下代码是线程不安全的，因为不能保证 `++value` 是原子性操作。

```
public class ThreadNotSafeCount {

    private  Long value;

    public Long getCount() {
        return value;
    }

    public void inc() {
        ++value;
    }
}
```

通过使用 `Javap -c` 查看汇编代码如下：

```
public void inc();
   Code:
      0: aload_0
      1: dup
      2: getfield      #2                  // Field value:J
      5: lconst_1
      6: ladd
      7: putfield      #2                  // Field value:J
     10: return
```

可知简单的 `++value` 有 2，5，6，7 组成，其中2是获取当前 value 的值并放入栈顶，5是把常量1放入栈顶，6是把当前栈顶中2个值相加并把结果放入栈顶，7则是把栈顶结果赋值会 value 变量，可知 Java 中简单的一句 ++value 转换为汇编后就不具有原子性了。

那么如何才能保证多个操作完成原子性呢，最简单的是使用 Synchronized 进行同步，修改代码如下：

```
public class ThreadSafeCount {

    private  Long value;

    public synchronized Long getCount() {
        return value;
    }

    public synchronized void inc() {
        ++value;
```

```
        }
    }
```

使用 Synchronized 的确可以实现线程安全，即实现内存可见性和同步，但是 Synchronized 是独占锁，同时只有一个线程可以调用 getCount 方法，其他没有获取内部锁的线程会被阻塞掉；而这里 getCount 方法只是读操作，多个线程同时调用不会存在线程安全问题，但是加了关键字 Synchronized 后同时就只能有一个线程可以调用了，这显然大大降低了并发性。

也许你会问既然是只读操作那么为何不去掉 getCount 方法上的 Synchronized 关键字呢？其实是不能去掉的，别忘了这里要靠 Synchronized 的内存语义来实现 value 的内存可见性。

那么有没有更好的实现呢？答案是肯定的，下面会讲到的内部使用非阻塞 CAS 算法实现的原子性操作类 AtomicLong 就是不错选择。

Java 中的 CAS 操作和 AtomicLong 实现原理

CAS 来源

在 Java 中锁在并发处理中占据了一席之地，但是使用锁不好的地方是当一个线程没有获取到锁后会被阻塞挂起，这会导致线程上下文的切换和重新调度的开销。

Java 中提供了非阻塞的 volatile 关键字来解决共享变量的可见性问题，这在一定程度上弥补了锁所在带来的开销，但是 volatile 只能保证共享变量的可见性问题，但是还是不能解决例如读 -> 改 -> 写等的原子性问题。

CAS 即 Compare And Swap，是 JDK 提供的非阻塞原子性操作，它通过硬件保证了比较-更新操作的原子性，JDK 里面的 Unsafe 类提供了一些列的 `compareAndSwap*` 方法，下面以 compareAndSwapLong 为例进行简单介绍。

- boolean compareAndSwapLong(Object obj,long valueOffset,long expect, long update) 方法。

compareAndSwap 的意思也就是比较并交换，CAS 有四个操作数分别为：对象内存位置，对象中的变量的偏移量，变量预期值 expect，新的值 update。

操作含义是如果对象 obj 中内存偏移量为 valueOffset 位置的变量值为 expect 则使用新的值 update 替换旧的值 expect。这个是处理器提供的一个原子性指令。

AtomicLong 的原理

并发包中原子性操作类都有 AtomicInteger，AtomicLong，AtomicBoolean，原理类似，本节讲解下 AtomicLong 类。AtomicLong 是原子性递增或者递减类，其内部使用 Unsafe 来

实现，下面看下代码：

```
public class AtomicLong extends Number implements
java.io.Serializable {
    private static final long serialVersionUID =
1927816293512124184L;

    //  （1）获取Unsafe实例
    private static final Unsafe unsafe = Unsafe.getUnsafe();

    //（2）存放变量value的偏移量
    private static final long valueOffset;


     //（3）判断JVM是否支持Long类型无锁CAS
    static final boolean VM_SUPPORTS_LONG_CAS = VMSupportsCS8();
    private static native boolean VMSupportsCS8();

    static {
        try {
            //（4）获取value在AtomicLong中偏移量
            valueOffset = unsafe.objectFieldOffset
                (AtomicLong.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    //（5）实际变量值
    private volatile long value;

    public AtomicLong(long initialValue) {
        value = initialValue;
    }
    ....
}
```

- 代码（1）创建了通过 Unsafe.getUnsafe（）方式获取到 Unsafe 类实例，这里你可能会疑问为何这里能通过 Unsafe.getUnsafe() 方式获取到 Unsafe 类实例？其实这是因为 AtomicLong 类也是在 rt.jar 包里面，AtomicLong 类的加载就是通过 BootStarp 类加载器进行加载的（关于 Unsafe 后面高级篇会具体讲解，这里先了解）
- 代码（5）中 value 声明为 volatile 是为了多线程下保证内存可见性，value 是具体存放计数的变量。
- 代码（2） （4）获取 value 变量在 AtomicLong 类中偏移量。

下面重点看下 AtomicLong 中主要函数：

- 递增和递减操作代码。

```
//（6）调用unsafe方法，原子性设置value值为原始值+1，返回值为递增后的值
public final long incrementAndGet() {
    return unsafe.getAndAddLong(this, valueOffset, 1L) + 1L;
}

//（7)调用unsafe方法，原子性设置value值为原始值-1，返回值为递减之后的值
public final long decrementAndGet() {
    return unsafe.getAndAddLong(this, valueOffset, -1L) - 1L;
}

//(8)调用unsafe方法，原子性设置value值为原始值+1,返回值为原始值
public final long getAndIncrement() {
    return unsafe.getAndAddLong(this, valueOffset, 1L);
}


//(9)调用unsafe方法，原子性设置value值为原始值-1，返回值为原始值
public final long getAndDecrement() {
    return unsafe.getAndAddLong(this, valueOffset, -1L);
}
```

如上代码内部都是调用 Unsafe 的 getAndAddLong 方法实现，这个函数是个原子性操作，这里第一个参数是 AtomicLong 实例的引用，第二个参数是 value 变量在 AtomicLong 中的偏移值，第三个参数是要设置第二个变量的值。

其中 getAndIncrement 方法在 JDK 7 的实现逻辑为：

```
public final long getAndIncrement() {
        while (true) {
            long current = get();
            long next = current + 1;
            if (compareAndSet(current, next))
                return current;
        }
    }
```

如上代码可知每个线程是先拿到变量的当前值（由于是 value 是 volatile 变量所以这里拿到的是最新的值），然后在工作内存对其进行增一操作，然后使用 CAS 修改变量的值，如果设置失败，则循环继续尝试，直到设置成功。

而 JDK 8 逻辑为：

```
public final long getAndIncrement() {
    return unsafe.getAndAddLong(this, valueOffset, 1L);
}
```

其中JDK8中unsafe.getAndAddLong代码为：

```
public final long getAndAddLong(Object paramObject, long
paramLong1, long paramLong2)
  {
    long l;
    do
    {
      l = getLongVolatile(paramObject, paramLong1);
    } while (!compareAndSwapLong(paramObject, paramLong1, l, l +
paramLong2));
    return l;
  }
```

可知 JDK 7 的 AtomicLong 中的循环逻辑已经被 JDK 8 的原子操作类 UNsafe 内置了，之所以内置应该是考虑到这种函数会在其它地方也会用到，内置可以提高复用性。

- boolean compareAndSet(long expect, long update)方法

```
public final boolean compareAndSet(long expect, long update) {
    return unsafe.compareAndSwapLong(this, valueOffset, expect,
update);
}
```

如上代码可知道内部还是调用了 unsafe.compareAndSwapLong 方法。如果原子变量中 value 的值等于 expect 则使用 update 值更新该值并返回 true，否者返回 false。

下面通过一个多线程使用 AtomicLong 统计0的个数的例子来加深对 AtomicLong 的理解：

```
/**
  统计0的个数
 */
public class Atomic
{
   //(10)创建Long型原子计数器
   private static  AtomicLong atomicLong = new AtomicLong();
   //(11)创建数据源
   private static Integer[] arrayOne = new Integer[]
{0,1,2,3,0,5,6,0,56,0};
   private static Integer[] arrayTwo = new Integer[]
{10,1,2,3,0,5,6,0,56,0};

    public static void main( String[] args ) throws
InterruptedException
    {
     //（12）线程one统计数组arrayOne中0的个数
```

```
            Thread threadOne = new Thread(new Runnable() {

                @Override
                public void run() {

                 int size = arrayOne.length;
                 for(int i=0;i<size;++i){
                     if(arrayOne[i].intValue() == 0){

                         atomicLong.incrementAndGet();
                     }
                 }

                }
            });
            //（13）线程two统计数组arrayTwo中0的个数
            Thread threadTwo = new Thread(new Runnable() {

                @Override
                public void run() {

                     int size = arrayTwo.length;
                     for(int i=0;i<size;++i){
                         if(arrayTwo[i].intValue() == 0){

                             atomicLong.incrementAndGet();
                         }
                     }

                }
            });

            //(14)启动子线程
            threadOne.start();
            threadTwo.start();

            //(15)等待线程执行完毕
            threadOne.join();
            threadTwo.join();

            System.out.println("count 0:" + atomicLong.get());

        }
    }
```

输出结果：count 0:7。

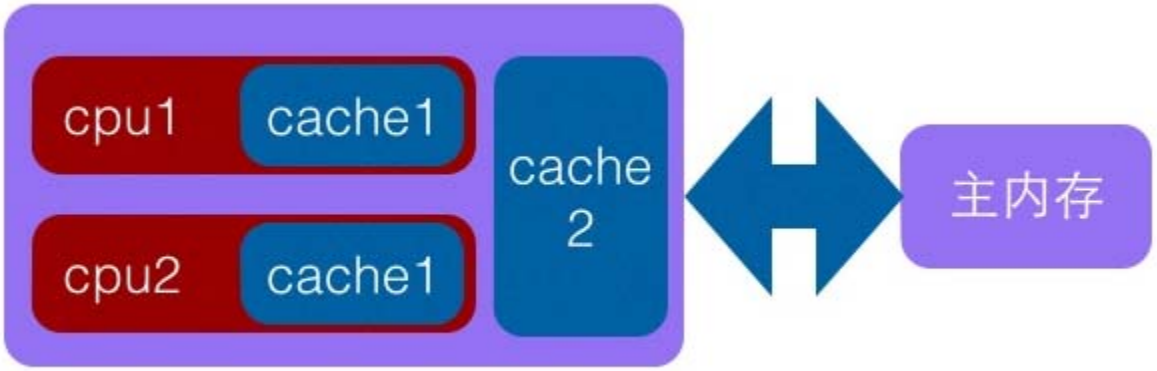
如上代码两个线程各自统计自己所在数据中0的个数，每当找到一个0就会调用AtomicLong的原子性递增方法。

注：在没有原子类的情况下例如最开始一节中自己做计数器的话，需要使用一定的同步措施，比如使用 Synchronized 关键字等，但是这些都是阻塞算法，对性能有一定损耗，而本节介绍的这些原子操作类都是使用 CAS 非阻塞算法，性能会更好。但是在高并发情况下 AtomicLong 还是会存在性能问题，后期高级篇会讲到 JDK 8 中提供了一个在高并发下性能更好的 LongAdder 类。

伪共享

什么是伪共享

计算机系统中为了解决主内存与 CPU 运行速度的差距，在 CPU 与主内存之间添加了一级或者多级高速缓冲存储器（Cache），这个 Cache 一般是集成到 CPU 内部的，所以也叫 CPU Cache，如下图是两级 Cache 结构：

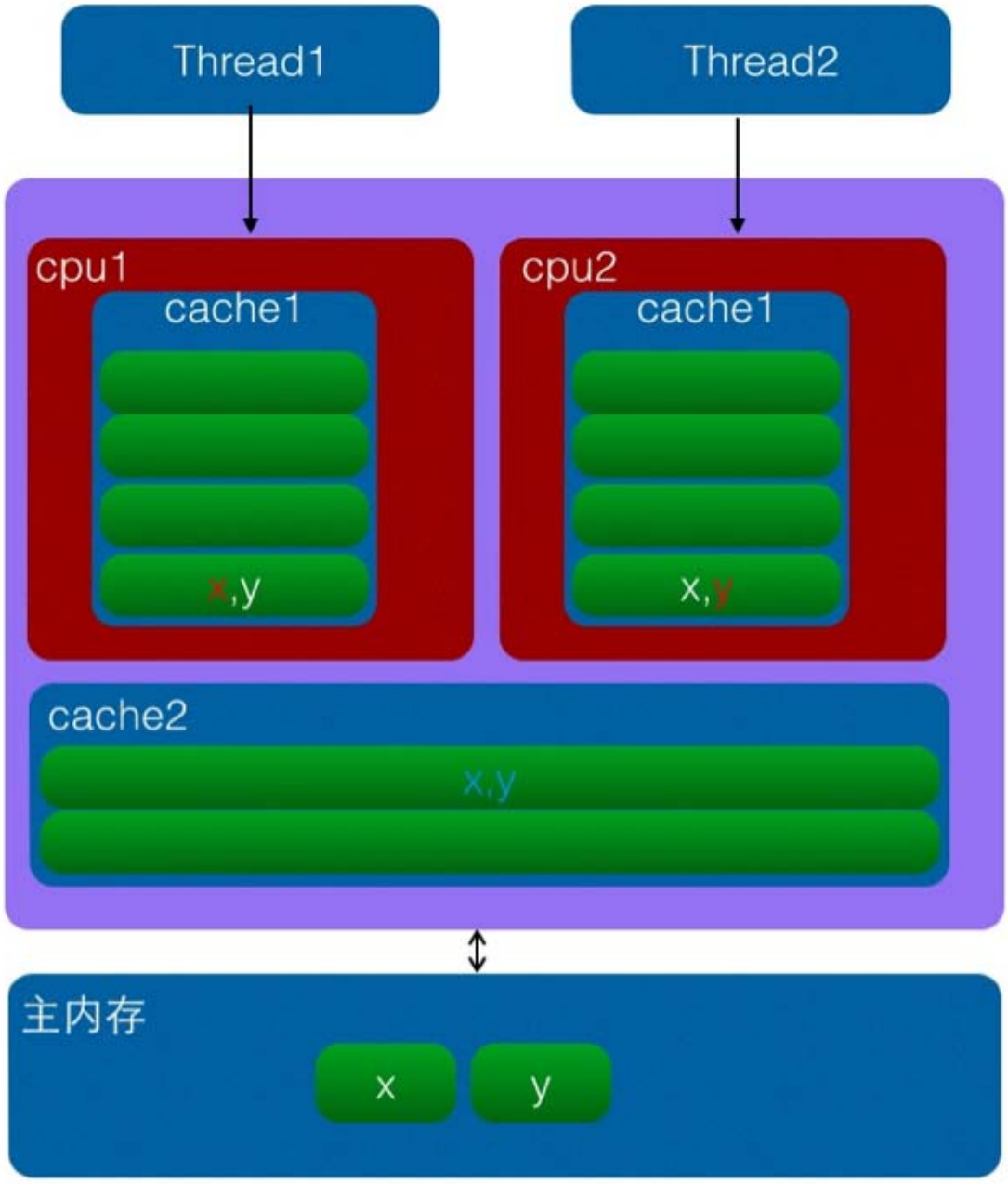


Cache 内部是按行存储的，其中每一行称为一个 Cache 行，Cache 行是 Cache 与主内存进行数据交换的单位，Cache 行的大小一般为2的幂次数字节。

当 CPU 访问某一个变量时候，首先会去看 CPU Cache 内是否有该变量，如果有则直接从中获取，否者就去主内存里面获取该变量，然后把该变量所在内存区域的一个 Cache 行大小的内存拷贝到 Cache（Cache 行是 Cache 与主内存进行数据交换的单位）。

由于存放到 Cache 行的的是内存块而不是单个变量，所以可能会把多个变量存放到了一个 Cache 行。当多个线程同时修改一个缓存行里面的多个变量时候，由于同时只能有一

个线程操作缓存行，所以相比每个变量放到一个缓存行性能会有所下降，这就是伪共享。



如上图变量 x，y 同时被放到了 CPU 的一级和二级缓存，当线程1使用 CPU 1对变量 x 进行更新时候，首先会修改 CPU 1 的一级缓存变量 x 所在缓存行，这时候缓存一致性协议会导致 CPU 2 中变量 x 对应的缓存行失效。

那么线程2写入变量 x 的时候就只能去二级缓存去查找，这就破坏了一级缓存，而一级缓存比二级缓存更快，这里也说明了多个线程不可能同时去修改自己所使用的 CPU 中缓存行中相同缓存行里面的变量。更坏的情况下如果 CPU 只有一级缓存，那么会导致频繁的直接访问主内存。

为何会出现伪共享

伪共享的产生是因为多个变量被放入了一个缓存行，并且多个线程同时去写入缓存行中不同变量。那么为何多个变量会被放入一个缓存行那。其实是因为 Cache 与内存交换数

据的单位就是 Cache 行，当 CPU 要访问的变量没有在 Cache 命中时候，根据程序运行的局部性原理会把该变量在内存中大小为 Cache 行的内存放如缓存行。

```
long a;
long b;
long c;
long d;
```

如上代码，声明了四个 long 变量，假设 Cache 行的大小为32个字节，那么当 CPU 访问变量 a 时候发现该变量没有在 Cache 命中，那么就会去主内存把变量 a 以及内存地址附近的 b、c、d 放入缓存行。

也就是地址连续的多个变量才有可能会被放到一个缓存行中，当创建数组时候，数组里面的多个元素就会被放入到同一个缓存行。那么单线程下多个变量放入缓存行对性能有影响？其实正常情况下单线程访问时候由于数组元素被放入到了一个或者多个 Cache 行对代码执行是有利的，因为数据都在缓存中，代码执行会更快，可以对比下面代码执行：

代码（1）：

```
public class TestForContent {

    static final int LINE_NUM = 1024;
    static final int COLUM_NUM = 1024;
    public static void main(String[] args) {

        long [][] array = new long[LINE_NUM][COLUM_NUM];

        long startTime = System.currentTimeMillis();
        for(int i =0;i<LINE_NUM;++i){
            for(int j=0;j<COLUM_NUM;++j){
                array[i][j] = i*2+j;
            }
        }
        long endTime = System.currentTimeMillis();
        long cacheTime = endTime - startTime;
        System.out.println("cache time:" + cacheTime);

    }
}
```

代码（2）：

```
public class TestForContent2 {

    static final int LINE_NUM = 1024;
    static final int COLUM_NUM = 1024;
```

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

        long [][] array = new long[LINE_NUM][COLUM_NUM];

        long startTime = System.currentTimeMillis();
        for(int i =0;i<COLUM_NUM;++i){
            for(int j=0;j<LINE_NUM;++j){
                array[j][i] = i*2+j;
            }
        }
        long endTime = System.currentTimeMillis();

        System.out.println("no cache time:" + (endTime -
startTime));

    }

}
```

我 Mac 电脑上执行代码（1）多次耗时均在10ms一下，执行代码（2）多次耗时均在10ms以上。

总的来说代码（1）比代码（2）执行的快，这是因为数组内数组元素之间内存地址是连续的，当访问数组第一个元素时候，会把第一个元素后续若干元素一块放入到 Cache 行，这样顺序访问数组元素时候会在 Cache 中直接命中，就不会去主内存读取，后续访问也是这样。

总结下也就是当顺序访问数组里面元素时候，如果当前元素在 Cache 没有命中，那么会从主内存一下子读取后续若干个元素到 Cache，也就是一次访问内存可以让后面多次直接在 Cache 命中。而代码（2）是跳跃式访问数组元素的，而不是顺序的，这破坏了程序访问的局部性原理，并且 Cache是有容量控制的，Cache 满了会根据一定淘汰算法替换 Cache 行，会导致从内存置换过来的 Cache 行的元素还没等到读取就被替换掉了。

所以单个线程下顺序修改一个 Cache 行中的多个变量，是充分利用了程序运行局部性原理，会加速程序的运行，而多线程下并发修改一个 Cache 行中的多个变量而就会进行竞争 Cache 行，降低程序运行性能。

如何避免伪共享

JDK 8 之前一般都是通过字节填充的方式来避免，也就是创建一个变量的时候使用填充字段填充该变量所在的缓存行，这样就避免了多个变量存在同一个缓存行，如下代码：

```
    public final static class FilledLong {
        public volatile long value = 0L;
        public long p1, p2, p3, p4, p5, p6;
    }
```

假如 Cache 行为64个字节，那么我们在 FilledLong 类里面填充了6个 long 类型变量，每个 long 类型占用8个字节，加上 value 变量的8个字节总共56个字节，另外这里 FilledLong 是一个类对象，而类对象的字节码的对象头占用了8个字节，所以当 new 一个 FilledLong 对象时候实际会占用64个字节的内存，这个正好可以放入 Cache 的一个行。

在 JDK 8 中提供了一个 sun.misc.Contended 注解，用来解决伪共享问题，上面代码可以修改为如下：

```
@sun.misc.Contended
  public final static class FilledLong {
        public volatile long value = 0L;
    }
```

上面是修饰类的，当然也可以修饰变量，比如 Thread 类中的使用：

```
    /** The current seed for a ThreadLocalRandom */
    @sun.misc.Contended("tlr")
    long threadLocalRandomSeed;

    /** Probe hash value; nonzero if threadLocalRandomSeed
initialized */
    @sun.misc.Contended("tlr")
    int threadLocalRandomProbe;

    /** Secondary seed isolated from public ThreadLocalRandom
sequence */
    @sun.misc.Contended("tlr")
    int threadLocalRandomSecondarySeed;
```

Thread 类里面这三个变量是在 ThreadLocalRandom（Chat：《Java 并发编程之美：并发编程高级篇之一》中对其进行了讲解）中为了实现高并发下高性能生成随机数时候使用的，这三个变量默认是初始化为0。

需要注意的是默认情况下 @Contended 注解只用到 Java 核心类，比如 rt 包下的类，如果需要在用户 classpath 下的类使用这个注解需要添加 JVM 参数：`-XX:-RestrictContended`，另外默认填充的宽度为128，如果你想要自定义宽度可以设置 `-XX:ContendedPaddingWidth` 参数。

注：本节讲述了伪共享如何产生，以及如何避免，并证明多线程下访问同一个 Cache 行的多个的变量时候才会出现伪共享，当单个线程访问一个 Cache 行里面的多个变量时候反而对程序运行起到加速作用。这里为后面高级篇讲解 LongAdder 的实现提供了基础。

Java 中的指令重排序

Java 内存模型允许编译器和处理器对指令进行重排序以提高运行性能，并且重排序只会对不存在数据依赖性的指令进行重排序；在单线程下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，但是在多线程下就会存在问题。

下面看一个例子

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

如上代码变量 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。

下面看一个多线程的例子：

```
public static class ReadThread extends Thread {
        public void run() {

            while(!Thread.currentThread().isInterrupted()){
                if(ready){//(1)
                    System.out.println(num+num);//(2)
                }
                System.out.println("read thread....");
            }

        }
    }

public static class Writethread extends Thread {
   public void run() {
        num = 2;//(3)
        ready = true;//(4)
        System.out.println("writeThread set over...");
   }
}

private static int num =0;
private static boolean ready = false;

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

   ReadThread rt = new ReadThread();
   rt.start();

   Writethread  wt = new Writethread();
   wt.start();
```

```
        Thread.sleep(10);
        rt.interrupt();
        System.out.println("main exit");
    }
```

首先这段代码里面的变量没有声明为 volatile 也没有使用任何同步措施，所以多线程下存在共享变量内存可见性问题，这里先不谈内存可见性问题，因为通过把变量声明为 volatile 本身就可以避免指令重排序问题。

这里先看看指令重排序会造成什么影响，如上代码不考虑内存可见性问题的情况下 程序一定会输出4？答案是不一定，由于代码（1）（2）（3）（4）之间不存在依赖，所以写线程的代码（3）（4）可能被重排序为先执行（4）在执行（3），那么执行（4）后，读线程可能已经执行了（1）操作，并且在（3）执行前开始执行（2）操作，这时候打印结果为0而不是4。

这就是重排序在多线程下导致程序执行结果不是我们想要的了，这里使用 volatile 修饰 ready 可以避免重排序和内存可见性问题。

当写 volatile 变量时候，可以确保 volatile 写之前的操作不会被编译器重排序到 volatile 写之后。
当读 volatile 读变量时候，可以确保 volatile 读之后的操作不会被编译器重排序到 volatile 读之前。

锁的概述

乐观锁与悲观锁

乐观锁和悲观锁是在数据库中使用的名词，本节这里也提下。

悲观锁

悲观锁指对数据被外界修改持保守态度，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制，数据库中实现是对数据记录操作前给记录加排它锁。如果获取锁失败，则说明数据正在被其它线程修改，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。

使用悲观锁的一个常用的例子： `select * from 表 where .. for update;`。

乐观锁

乐观锁是相对悲观锁来说的，它认为数据一般情况下不会造成冲突，所以在访问记录前不会加排它锁，而是在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测。具体说是根据 update 返回的行数让用户决定如何去做。

例如： `update 表 set comment='***',status='operator',version=version+1 where version = 1 and id = 1;`

乐观锁并不会使用数据库提供的锁机制，一般在表添加 version 字段或者使用业务状态来做。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。

公平锁与非公平锁

根据线程获取锁的抢占机制锁可以分为公平锁和非公平锁，公平锁表示线程获取锁的顺序是按照线程请求锁的时间长短来分决定的的，也就是最早获取锁的线程将最早获取到锁，也就是先来先得的 FIFO 顺序。而非公平锁则运行时候闯入，也就是先来不一定先得。

ReentrantLock 提供了公平和非公平锁的实现：

- 公平锁： `ReentrantLock pairLock = new ReentrantLock(true);`
- 非公平锁： `ReentrantLock pairLock = new ReentrantLock(false);`

如果构造函数不传递参数，则默认是非公平锁。

具体来说假设线程 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它干涉，如果使用公平锁则需要把 C 挂起，让 B 获取当前锁。

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。

独占锁与共享锁

根据锁只能被单个线程持有还是能被多个线程共同持有，锁分为独占锁和共享锁。

独占锁保证任何时候都只有一个线程能得到锁，ReentrantLock 就是以独占方式实现的。共享锁则同时有多个线程可以持有，例如 ReadWriteLock 读写锁，它允许一个资源可以被多线程同时进行读操作。

独占锁是一种悲观锁，每次访问资源都先加上互斥锁，这限制了并发性，因为读操作并不会影响数据一致性，而独占锁只允许同时一个线程读取数据，其它线程必须等待当前线程释放锁才能进行读取。

共享锁则是一种乐观锁，它放宽了加锁的条件，允许多个线程同时进行读操作。

什么是可重入锁

当一个线程要获取一个被其它线程持有的独占锁时候，该线程会被阻塞，那么当一个线程再次获取它自己已经获取的锁时候是否会被阻塞那?如果不被阻塞，那么我们说该锁是

可重入的，也就是只要该线程获取了该锁，那么可以无限制次数（高级篇我们会知道严格来说是有限次数）进入被该锁锁住的代码。

下面看一个例子看看什么情况下会用可重入锁。

```
public class Hello{
     public Synchronized void helloA(){
        System.out.println("hello");
     }

     public Synchronized void helloB(){
        System.out.println("hello B");
        helloA();
     }

}
```

如上面代码当调用 helloB 函数前会先获取内置锁，然后打印输出，然后调用 helloA 方法，调用前会先去获取内置锁，如果内置锁不是可重入的那么该调用就会导致死锁了，因为线程持有并等待了锁导致调用 helloA 时候永远不会获取到锁。

实际上 synchronized 内部锁是可重入锁，可重入锁的原理是在锁内部维护了一个线程标示，用来标示该锁目前被那个线程占用，然后关联一个计数器。一开始计数器值为0，说明该锁没有被任何线程占用，当一个线程获取了该锁，计数器会变成1，其它线程在获取该锁时候发现锁的所有者不是自己就会被阻塞挂起。

但是当获取该锁的线程再次获取锁时候发现锁拥有者是自己，就会把计数器值+1， 当释放锁后计数器会-1，当计数器为0时候，锁里面的线程标示重置为 null，这时候阻塞的线程会获取被唤醒来竞争获取该锁。

总结

本章主要介绍了并发编程的基础知识，为后面高级篇讲解并发包源码提供了基础，通过图形结合讲述了为什么要使用多线程编程，多线程编程存在的线程安全问题，以及什么是内存可见性问题。然后讲解了 synchronized 和 volatile 关键字，并且强调了前者既保证了内存可见性同时也保证了原子性，而后者则主要做到了内存可见性，但是它们的内存语义还是很相似的，最后讲解的什么是 CAS 和线程间同步以及各种锁的概念，都为后面讲解 JUC 包源码奠定了基础。