并发编程专题，属于内功修炼的过程，有些东西不一定能用到，但是需要去理解他们背后的原理。积累越多，后续的学习过程就会越轻松，希望大家坚持。

咕泡学院 VIP 课：初识多线程及其原理分析

课程目标

1. 多线程的发展历史

2. 线程的应用

3. 并发编程的基础

4. 线程安全问题

什么情况下应该使用多线程

线程出现的目的是什么？解决进程中多任务的实时性问题？其实简单来说，也就是解决“阻塞”的问题，阻塞的意思就是程序运行到某个函数或过程后等待某些事件发生而暂时停止 CPU 占用的情况，也就是说会使得 CPU 闲置。还有一些场景就是比如对于一个函数中的运算逻辑的性能问题，我们可以通过多线程的技术，使得一个函数中的多个逻辑运算通过多线程技术达到一个并行执行，从而提升性能

所以，多线程最终解决的就是“等待”的问题，所以简单总结的使用场景

Ø 通过并行计算提高程序执行性能

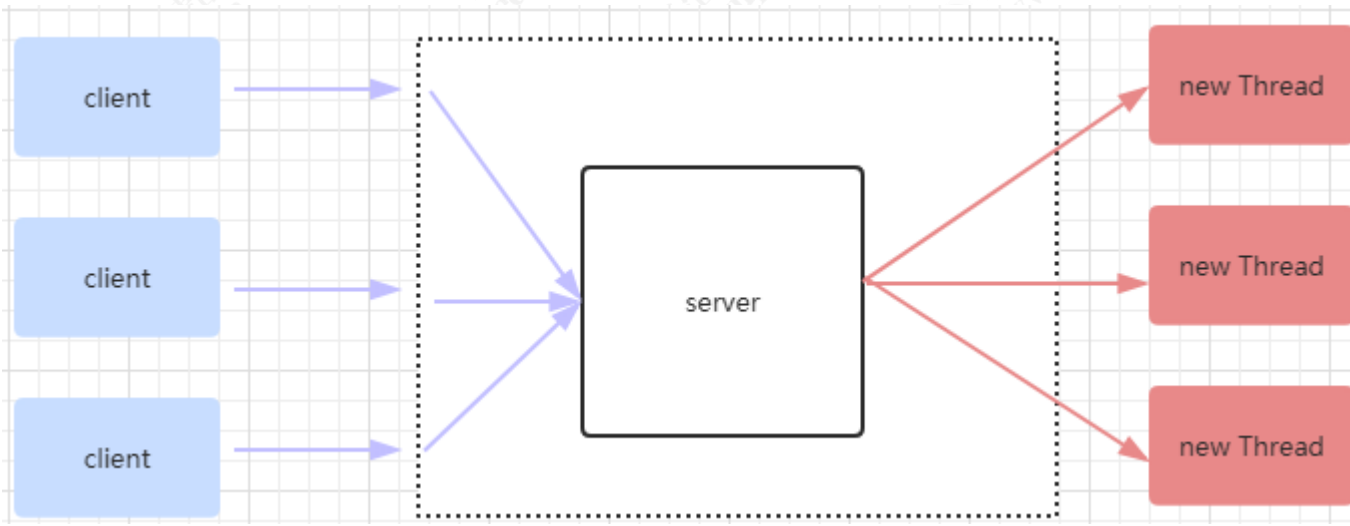
Ø 需要等待网络、I/O 响应导致耗费大量的执行时间，可以采用异步线程的方式来减少阻塞

tomcat7 以前的 io 模型

多线程的应用场景

- **客户端阻塞** 如果客户端只有一个线程，这个线程发起读取文件的操作必须等待 IO 流返回，线程（客户端）才能做其他的事
- **线程级别阻塞 BIO** 客户端一个线程情况下，一个线程导致整个客户端阻塞。

  那么我们可以使用多线程，一部分线程在等待 IO 操作返回其他线程可以继续做其他的事。此时从客户端角度来说，客户端没有闲着。



如何应用多线程

在 Java 中，有多种方式来实现多线程。继承 Thread 类、实现 Runnable 接口、使用 ExecutorService、Callable、Future 实现带返回结果的多线程。

继承 Thread 类创建线程

Thread 类本质上是实现了 Runnable 接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过 Thread 类的 start()实例方法。start()方法是一个 native 方法，它会启动一个新线程，并执行 run()方法。这种方式实现多线程很简单，通过自己的类直接 extend Thread，并复写 run()方法，就可以启动新线程并执行自己定义的 run()方法。

```
public class MyThread extends Thread {
    public void run() {
     System.out.println("MyThread.run()");
    }
}
MyThread myThread1 = new MyThread();
MyThread myThread2 = new MyThread();
myThread1.start();
myThread2.start();
```

实现 Runnable 接口创建线程

如果自己的类已经 extends 另一个类，就无法直接 extends Thread，此时，可以实现一个 Runnable 接口

```
public class MyThread extends OtherClass implements Runnable {
    public void run() {
     System.out.println("MyThread.run()");
    }
}
```

实现 Callable 接口通过 FutureTask 包装器来创建 Thread 线程

有的时候，我们可能需要让一步执行的线程在执行完成以后，提供一个返回值给到当前的主线程，主线程需要依赖这个值进行后续的逻辑处理，那么这个时候，就需要用到带返回值的线程了。Java 中提供了这样的实现方式 d

```
public class CallableDemo implements Callable<String> {
   public static void main(String[] args) throws ExecutionException,
InterruptedException {
       ExecutorService executorService=
Executors.newFixedThreadPool(1);
       CallableDemo callableDemo=new CallableDemo();
       Future<String> future=executorService.submit(callableDemo);
       System.out.println(future.get());
       executorService.shutdown();
  }
   @Override
   public String call() throws Exception {
       int a=1;
       int b=2;
       System.out.println(a+b);
       return "执行结果:"+(a+b);
  }
}
```

如何把多线程用得更加优雅

合理的利用异步操作，可以大大提升程序的处理性能，下面这个案例，如果看过 zookeeper 源码的同学应该都见过

通过阻塞队列以及多线程的方式，实现对请求的异步化处理，提升处理性能

Request

```
public class Request {

   private String name;

   public String getName() {
       return name;
```

```
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Request{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

RequestProcessor

```
public interface RequestProcessor {

    void processRequest(Request request);
}
```

PrintProcessor

```
public class PrintProcessor extends Thread implements
RequestProcessor{

    LinkedBlockingQueue<Request> requests = new
LinkedBlockingQueue<Request>();

    private final RequestProcessor nextProcessor;

    public PrintProcessor(RequestProcessor nextProcessor) {
        this.nextProcessor = nextProcessor;
    }

    @Override
    public void run() {
        while (true) {
```

```
            try {
                Request request=requests.take();
                System.out.println("print data:"+request.getName());
                nextProcessor.processRequest(request);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    //处理请求
    public void processRequest(Request request) {
        requests.add(request);
    }
}
```

SaveProcessor

```
public class SaveProcessor extends Thread implements
RequestProcessor{

    LinkedBlockingQueue<Request> requests = new
LinkedBlockingQueue<Request>();

    @Override
    public void run() {
        while (true) {
            try {
                Request request=requests.take();
                System.out.println("begin save request
info:"+request);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    //处理请求
    public void processRequest(Request request) {
        requests.add(request);
    }
```

```
}
```

Demo

```
public class Demo {

    PrintProcessor printProcessor;

    protected Demo(){
        SaveProcessor saveProcessor=new SaveProcessor();
        saveProcessor.start();
        printProcessor=new PrintProcessor(saveProcessor);
        printProcessor.start();
    }

    private void doTest(Request request){
        printProcessor.processRequest(request);
    }

    public static void main(String[] args) {
        Request request=new Request();
        request.setName("Mic");
        new Demo().doTest(request);

    }
}
```

Java 并发编程的基础

线程作为操作系统调度的最小单元，并且能够让多线程同时执行，极大的提高了程序的性能，在多核环境下的优势更加明显。但是在使用多线程的过程中，如果对它的特性和原理不够理解的话，很容易造成各种问题。

线程的状态

Java 线程既然能够创建，那么也势必会被销毁，所以线程是存在生命周期的，那么我们接下来从线程的生命周期开始去了解线程。

线程一共有 6 种状态（NEW、RUNNABLE、BLOCKED、WAITING、TIME_WAITING、TERMINATED）

NEW：初始状态，线程被构建，但是还没有调用 start 方法

RUNNABLED：运行状态，JAVA 线程把操作系统中的就绪和运行两种状态统一称为“运行中”

BLOCKED：阻塞状态，表示线程进入等待状态,也就是线程因为某种原因放弃了 CPU 使用权，阻塞也分为几种情况

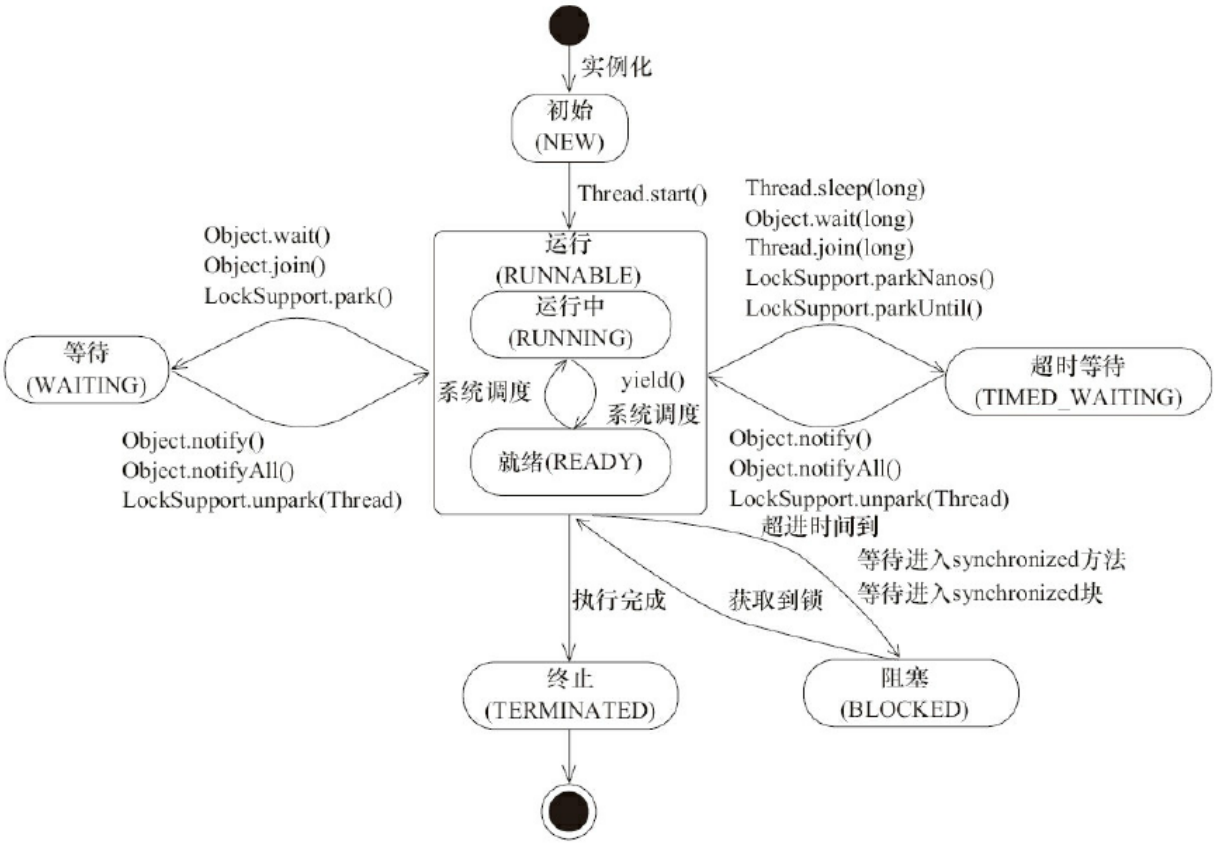
Ø 等待阻塞：运行的线程执行 wait 方法，jvm 会把当前线程放入到等待队列

Ø 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被其他线程锁占用了，那么 jvm 会把当前的线程放入到锁池中

Ø 其他阻塞：运行的线程执行 Thread.sleep 或者 t.join 方法，或者发出了 I/O 请求时，JVM 会把当前线程设置为阻塞状态，当 sleep 结束、join 线程终止、io 处理完毕则线程恢复

TIME_WAITING：超时等待状态，超时以后自动返回

TERMINATED：终止状态，表示当前线程执行完毕



通过代码演示线程的状态

编写如下代码

```
public class ThreadStatus {

    public static void main(String[] args) {
        //TIME_WAITING
        new Thread(()->{
            while(true){
                try {
                    TimeUnit.SECONDS.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"timewaiting").start();
```

```
        //WAITING，线程在 ThreadStatus 类锁上通过 wait 进行等待
        new Thread(()->{
            while(true){
                synchronized (ThreadStatus.class){
                    try {
                        ThreadStatus.class.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        },"Waiting").start();
        //线程在 ThreadStatus 加锁后，不会释放锁
        new Thread(new BlockedDemo(),"BlockDemo-01").start();
        new Thread(new BlockedDemo(),"BlockDemo-02").start();
    }
    static class BlockedDemo extends Thread{
        public void run(){
            synchronized (BlockedDemo.class){
                while(true){
                    try {
                        TimeUnit.SECONDS.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

通过相应命令显示线程状态

- 打开终端或者命令提示符，键入“jps”，（JDK1.5 提供的一个显示当前所有 java 进程 pid 的命令），可以获得相应进程的 pid
- 根据上一步骤获得的 pid，继续输入 jstack pid（jstack 是 java 虚拟机自带的一种堆栈跟踪工具。jstack 用于打印出给定的 java 进程 ID 或 core file 或远程调试服务的 Java 堆栈信息）

线程的停止

线程的启动过程大家都非常熟悉，但是如何终止一个线程，我相信绝大部分人在面试的时候被问到这个问题时，也会不知所措，不知道怎么回答。

记住，线程的终止，并不是简单的调用 stop 命令去。虽然 api 仍然可以调用，

但是和其他的线程控制方法如 suspend、resume 一样都是过期了的不建议使

用，就拿 stop 来说，stop 方法在结束一个线程时并不会保证线程的资源正常释放，因此会导致程序可能出现一些不确定的状态。

要优雅的去中断一个线程，在线程中提供了一个 interrupt 方法

interrupt 方法

当其他线程通过调用当前线程的 interrupt 方法，表示向当前线程打个招呼，告诉他可以中断线程的执行了，至于什么时候中断，取决于当前线程自己。

线程通过检查资深是否被中断来进行相应，可以通过 isInterrupted()来判断是否被中断。

通过下面这个例子，来实现了线程终止的逻辑

```
public class InterruptDemo {
   private static int i;
   public static void main(String[] args) throws
InterruptedException {
        Thread thread=new Thread(()->{
            while(!Thread.currentThread().isInterrupted()){
                i++;
           }
            System.out.println("Num:"+i);
       },"interruptDemo");
        thread.start();
        TimeUnit.SECONDS.sleep(1);
        thread.interrupt();
  }
```

```
}
```

这种通过标识位或者中断操作的方式能够使线程在终止时有机会去清理资源，而不是武断地将线程停止，因此这种终止线程的做法显得更加安全和优雅

Thread.interrupted

上面的案例中，通过 interrupt，设置了一个标识告诉线程可以终止了，线程中还提供了静态方法 Thread.interrupted()对设置中断标识的线程复位。比如在上面的案例中，外面的线程调用 thread.interrupt 来设置中断标识，而在线程里面，又通过 Thread.interrupted 把线程的标识又进行了复位

```
public class InterruptDemo {
   public static void main(String[] args) throws
InterruptedException{
       Thread thread=new Thread(()->{
           while(true){
               boolean ii=Thread.currentThread().isInterrupted();
               if(ii){
                   System.out.println("before:"+ii);
                   Thread.interrupted();//对线程进行复位，中断标识为
false
                   System.out.println("after:"+Thread.currentThread()
.isInterrupted());
               }
           }
       });
        thread.start();
        TimeUnit.SECONDS.sleep(1);
        thread.interrupt();//设置中断标识,中断标识为 true
  }
}
```

其他的线程复位

除了通过 Thread.interrupted 方法对线程中断标识进行复位以外，还有一种被动复位的场景，就是对抛出 InterruptedException 异常的方法，在 InterruptedException 抛出之前，JVM 会先把线程的中断标识位清除，然后才会抛出 InterruptedException，这个时候如果调用 isInterrupted 方法，将会返回 false

```
public class InterruptDemo {
   public static void main(String[] args) throws
InterruptedException{
      Thread thread=new Thread(()->{
          while(true){
              try {
                  Thread.sleep(10000);
              } catch (InterruptedException e) {
                  //抛出该异常，会将复位标识设置为 false
                  e.printStackTrace();
              }
          }
      });
      thread.start();
      TimeUnit.SECONDS.sleep(1);
      thread.interrupt();//设置复位标识为 true
      TimeUnit.SECONDS.sleep(1);
      System.out.println(thread.isInterrupted());//false
  }
}
```

有同学在问线程为什么要复位？首先我们来看看线程执行 interrupt 以后的源码是做了什么？

thread.cpp

```
void Thread::interrupt(Thread* thread) {
 trace("interrupt", thread);
 debug_only(check_for_dangling_thread_pointer(thread);)
```

```
 os::interrupt(thread);
}
```

os_linux.cpp

```
void os::interrupt(Thread* thread) {
 assert(Thread::current() == thread ||
Threads_lock->owned_by_self(),
   "possibility of dangling Thread pointer");

 OSThread* osthread = thread->osthread();

 if (!osthread->interrupted()) {
   osthread->set_interrupted(true);
   // More than one thread can get here with the same value of
osthread,
   // resulting in multiple notifications. We do, however, want the
store
   // to interrupted() to be visible to other threads before we
execute unpark().
   OrderAccess::fence();
   ParkEvent * const slp = thread->_SleepEvent ;
   if (slp != NULL) slp->unpark() ;
}

 // For JSR166. Unpark even if interrupt status already was set
 if (thread->is_Java_thread())
  ((JavaThread*)thread)->parker()->unpark();

 ParkEvent * ev = thread->_ParkEvent ;
 if (ev != NULL) ev->unpark() ;

}
```

其实就是通过 unpark 去唤醒当前线程，并且设置一个标识位为 true。 并没有所谓的中断线程的操作，所以实际上，线程复位可以用来实现多个线程之间的通信。

线程的停止方法之 2

除了通过 interrupt 标识为去中断线程以外，我们还可以通过下面这种方式，定义一个 volatile 修饰的成员变量，来控制线程的终止。这实际上是应用了 volatile 能够实现多线程之间共享变量的可见性这一特点来实现的。

```
public class VolatileDemo {
   private volatile static boolean stop=false;
   public static void main(String[] args) throws
InterruptedException {
       Thread thread=new Thread(()->{
           int i=0;
           while(!stop){
               i++;
          }
      });
       thread.start();
       System.out.println("begin start thread");
       Thread.sleep(1000);
       stop=true;
  }
}
```

线程的安全性问题

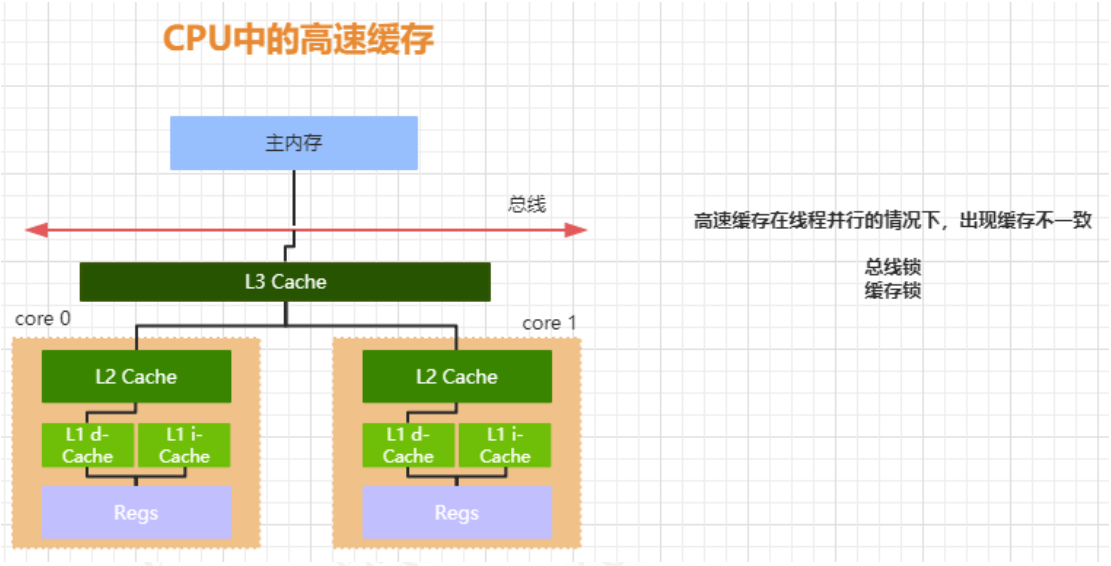
大家都知道，线程会存在安全性问题，那接下来我们从原理层面去了解线程为什么会存在安全性问题，并且我们应该怎么去解决这类的问题。

其实线程安全问题可以总结为: 可见性、原子性、有序性这几个问题，我们搞懂了这几个问题并且知道怎么解决，那么多线程安全性问题也就不是问题了

CPU 高速缓存

线程是 CPU 调度的最小单元，线程涉及的目的最终仍然是更充分的利用计算机处理的效能，但是绝大部分的运算任务不能只依靠处理器“计算”就能完成，处

理器还需要与内存交互，比如读取运算数据、存储运算结果，这个 I/O 操作是很难消除的。而由于计算机的存储设备与处理器的运算速度差距非常大，所以现代计算机系统都会增加一层读写速度尽可能接近处理器运算速度的高速缓存来作为内存和处理器之间的缓冲：将运算需要使用的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步到内存之中。



高速缓存从下到上越接近 CPU 速度越快，同时容量也越小。现在大部分的处理器都有二级或者三级缓存，从下到上依次为 L3 cache, L2 cache, L1 cache. 缓存又可以分为指令缓存和数据缓存，指令缓存用来缓存程序的代码，数据缓存用来缓存程序的数据

L1 Cache，一级缓存，本地 core 的缓存，分成 32K 的数据缓存 L1d 和 32k 指令缓存 L1i，访问 L1 需要 3cycles，耗时大约 1ns；

L2 Cache，二级缓存，本地 core 的缓存，被设计为 L1 缓存与共享的 L3 缓存之间的缓冲，大小为 256K，访问 L2 需要 12cycles，耗时大约 3ns；

L3 Cache，三级缓存，在同插槽的所有 core 共享 L3 缓存，分为多个 2M 的段，访问 L3 需要 38cycles，耗时大约 12ns；

缓存一致性问题

CPU-0 读取主存的数据，缓存到 CPU-0 的高速缓存中，CPU-1 也做了同样的事情，而 CPU-1 把 count 的值修改成了 2，并且同步到 CPU-1 的高速缓存，但是这个修改以后的值并没有写入到主存中，CPU-0 访问该字节，由于缓存没有更新，所以仍然是之前的值，就会导致数据不一致的问题

引发这个问题的原因是因为多核心 CPU 情况下存在指令并行执行，而各个 CPU 核心之间的数据不共享从而导致缓存一致性问题，为了解决这个问题，CPU 生产厂商提供了相应的解决方案

总线锁

当一个 CPU 对其缓存中的数据进行操作的时候，往总线中发送一个 Lock 信号。其他处理器的请求将会被阻塞，那么该处理器可以独占共享内存。总线锁相当于把 CPU 和内存之间的通信锁住了，所以这种方式会导致 CPU 的性能下降，所以 P6 系列以后的处理器，出现了另外一种方式，就是缓存锁。

缓存锁

如果缓存在处理器缓存行中的内存区域在 LOCK 操作期间被锁定，当它执行锁操作回写内存时，处理不在总线上声明 LOCK 信号，而是修改内部的缓存地址，然后通过缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改被两个以上处理器缓存的内存区域的数据，当其他处理器回写已经被锁定的缓存行的数据时会导致该缓存行无效。

所以如果声明了 CPU 的锁机制，会生成一个 LOCK 指令，会产生两个作用

1. Lock 前缀指令会引起引起处理器缓存回写到内存，在 P6 以后的处理器中，LOCK 信号一般不锁总线，而是锁缓存

2. 一个处理器的缓存回写到内存会导致其他处理器的缓存无效

缓存一致性协议

处理器上有一套完整的协议，来保证 Cache 的一致性，比较经典的应该就是 MESI 协议了，它的方法是在 CPU 缓存中保存一个标记位，这个标记为有四种状态

Ø M(Modified) 修改缓存，当前 CPU 缓存已经被修改，表示已经和内存中的数据不一致了

Ø I(Invalid) 失效缓存，说明 CPU 的缓存已经不能使用了

Ø E(Exclusive) 独占缓存，当前 cpu 的缓存和内存中数据保持一直，而且其他处理器没有缓存该数据

Ø S(Shared) 共享缓存，数据和内存中数据一致，并且该数据存在多个 cpu 缓存中

每个 Core 的 Cache 控制器不仅知道自己的读写操作，也监听其它 Cache 的读写操作，嗅探（snooping）"协议

CPU 的读取会遵循几个原则

1. 如果缓存的状态是 I，那么就从内存中读取，否则直接从缓存读取

2. 如果缓存处于 M 或者 E 的 CPU 嗅探到其他 CPU 有读的操作，就把自己的缓存写入到内存，并把自己的状态设置为 S

3. 只有缓存状态是 M 或 E 的时候，CPU 才可以修改缓存中的数据，修改后，缓存状态变为 MC

CPU 的优化执行

除了增加高速缓存以为，为了更充分利用处理器内内部的运算单元，处理器可能会对输入的代码进行乱序执行优化，处理器会在计算之后将乱序执行的结果充足，保证该结果与顺序执行的结果一直，但并不保证程序中各个语句计算的先后顺序与输入代码中的顺序一致，这个是处理器的优化执行；还有一个就是编程语言的编译器也会有类似的优化，比如做指令重排来提升性能。

并发编程的问题

前面说的和硬件有关的概念你可能听得有点蒙，还不知道他到底和软件有啥关系，其实原子性、可见性、有序性问题，是我们抽象出来的概念，他们的核心本质就是刚刚提到的缓存一致性问题、处理器优化问题导致的指令重排序问题。

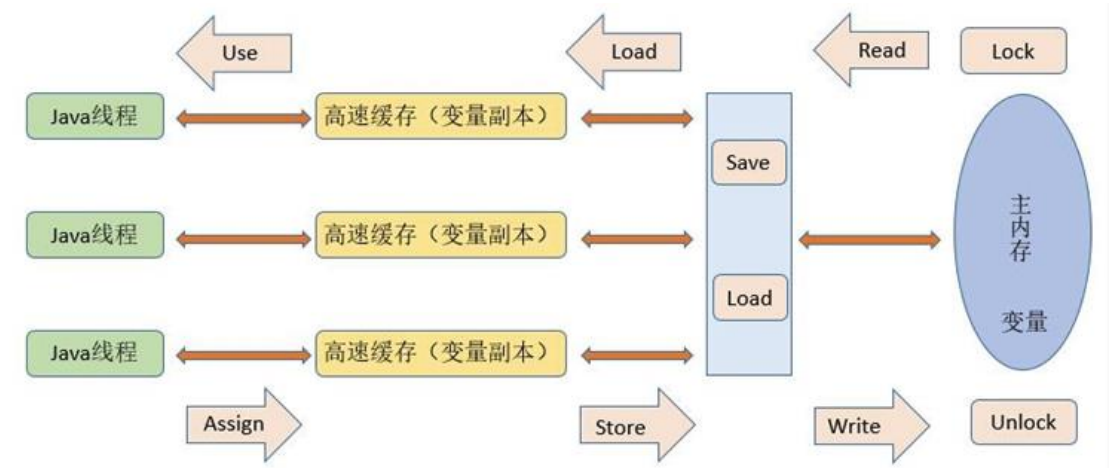
比如缓存一致性就导致可见性问题、处理器的乱序执行会导致原子性问题、指令重排会导致有序性问题。为了解决这些问题，所以在 JVM 中引入了 JMM 的概念

内存模型

内存模型定义了共享内存系统中多线程程序读写操作行为的规范，来屏蔽各种硬件和操作系统的内存访问差异，来实现 Java 程序在各个平台下都能达到一致的内存访问效果。Java 内存模型的主要目标是定义程序中各个变量的访问规则，也就是在虚拟机中将变量存储到内存以及从内存中取出变量（这里的变量，指的是共享变量，也就是实例对象、静态字段、数组对象等存储在堆内存中的变量。而对于局部变量这类的，属于线程私有，不会被共享）这类的底层细节。通过这些规则来规范对内存的读写操作，从而保证指令执行的正确性。

它与处理器有关、与缓存有关、与并发有关、与编译器也有关。他解决了 CPU 多级缓存、处理器优化、指令重排等导致的内存访问问题，保证了并发场景下的可见性、原子性和有序性，。内存模型解决并发问题主要采用两种方式：限制处理器优化和使用内存屏障

Java 内存模型定义了线程和内存的交互方式，在 JMM 抽象模型中，分为主内存、工作内存。主内存是所有线程共享的，工作内存是每个线程独有的。线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，不能直接读写主内存中的变量。并且不同的线程之间无法访问对方工作内存中的变量，线程间的变量值的传递都需要通过主内存来完成，他们三者的交互关系如下



所以，总的来说，JMM 是一种规范，目的是解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题。目的是保证并发编程场景中的原子性、可见性和有序性