一、多线程

什么是多线程?

- 多线程（multithreading）是指从软件或者硬件上实现多个线程并发执行的技术。具有多线程能力的计算机因有硬件支持而能够在同一时间执行多于一个线程，进而提升整体处理性能。
- 并发编程：编写多线程代码，解决多线程带来的问题

为什么要学并发编程?

- 首先，来看一个案例：手写网站服务器案例。
- 高性能应用程序的一把钥匙，应用程序的翅膀，面试高频的考点
- 中间件几乎都是多线程应用：MySQL、ES、Redis，Tomcat，Druid，HikariCP...

怎么学并发编程?

- 多线程核心知识（概念、线程状态、线程安全问题、三大特性）
- 同步代码块Synchronized及其实现原理
- volatile关键字及其实现原理
- 多线程在JVM中的实现原理
- JUC概述
- 原子类Atomic-CAS及其实现原理
- 锁Lock-AQS核心原理剖析
- 并发工具类、并发容器、阻塞队列
- 线程池原理剖析
- 线程池案例-Web容器-压力测试

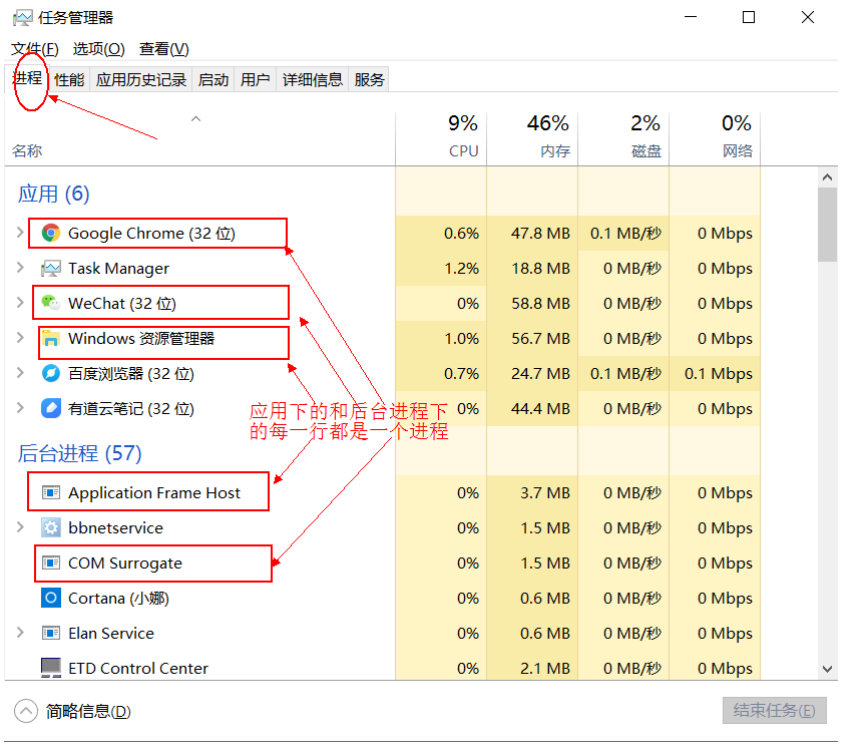
1. 多线程相关概念

1.1 线程和进程

- **进程**：是指内存中运行的一个应用程序，每个进程都有自己独立的内存空间；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**：是**进程中的一个执行单元**，负责当前进程中任务的执行。一个进程在其执行过程中，会产生很多个线程。

进程与线程区别:

- 进程：有独立**内存空间**，每个进程中的数据空间都是独立的。
- 线程：多线程之间**堆空间**与**方法区**是共享的，但每个线程的**栈空间**、**程序计数器**是独立的，线程消耗的资源比进程小的多。



1.1.1 什么是并发与并行?

- **并发（Concurrent）**：同一时间段，多个任务都在执行，单位时间内不一定同时执行。
- **并行（Parallel）**：单位时间内，多个任务同时执行，单位时间内一定是同时执行。并行上限取决于CPU核数（CPU时间片内50ms）

注意：并发是一种能力，而并行是一种手段。当我们的系统拥有了并发的能力后，代码如果跑在多核CPU上就可以并行运行。所以咱们会说高并发处理，而不会说高并行处理。并行处理是基于硬件CPU的是固定的，而并发处理的能力是可以通过设计编码进行提高的。

1.1.2 什么是线程上下文切换?

现在计算机一般都是多核CPU，且OS都能够同时支持远大于CPU内核数的线程运行。那么，**OS如何分配CPU资源与调度线程呢?**

以我的个人计算机为例：4核8线程

一个CPU内核，同一时刻只能被一个线程使用。为了提升CPU利用率，CPU采用了**时间片算法**将CPU时间片轮流分配给多个线程，每个线程分配了一个时间片（几十毫秒/线程），线程在时间片内，使用CPU执行任务。当时间片用完后，线程会被挂起，然后把 CPU 让给其它线程。

那么问题来了，线程再次运行时，系统是怎么知道线程之前运行到哪里了呢?

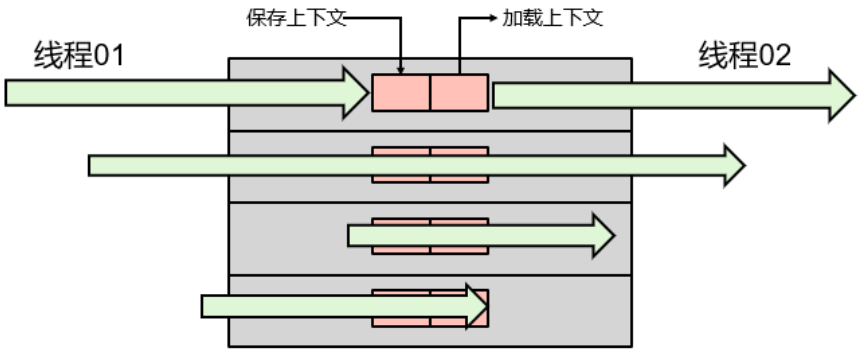
- CPU切换前会把当前任务状态保存下来，用于下次切换回任务时再次加载。
- **任务状态的保存及再加载的过程就叫做上下文切换。**

任务状态信息保存在哪里呢?

- 程序计数器：用来存储CPU正在执行的指令的位置，和即将执行的下一条指令的位置。
- 他们都是CPU在运行任何任务前，必须依赖的环境，被叫做**CPU上下文。**

上下文切换过程：

1. 挂起当前任务任务，将这个任务在 CPU 中的状态（上下文）存储于内存中的某处。
2. 恢复一个任务，在内存中检索下一个任务的上下文并将在 CPU 的寄存器中恢复。
3. 跳转到程序计数器所指定的位置（即跳转到任务被中断时的代码行）。



CPU（4核）时间片50ms

线程上下文切换会有什么问题呢?

过多的线程并行执行会导致CPU资源的争抢，产生频繁的上下文切换，常常表现为高并发执行时，RT延长。因此，合理控制上下文切换次数，可以提高多线程应用的运行效率。（也就是说**线程并不是越多越好，要合理的控制线程的数量。**）

- 直接消耗：指的是CPU寄存器需要保存和加载，系统调度器的代码需要执行
- 间接消耗：指的是多核的cache之间得共享数据，间接消耗对于程序的影响要看线程工作区操作数据的大小

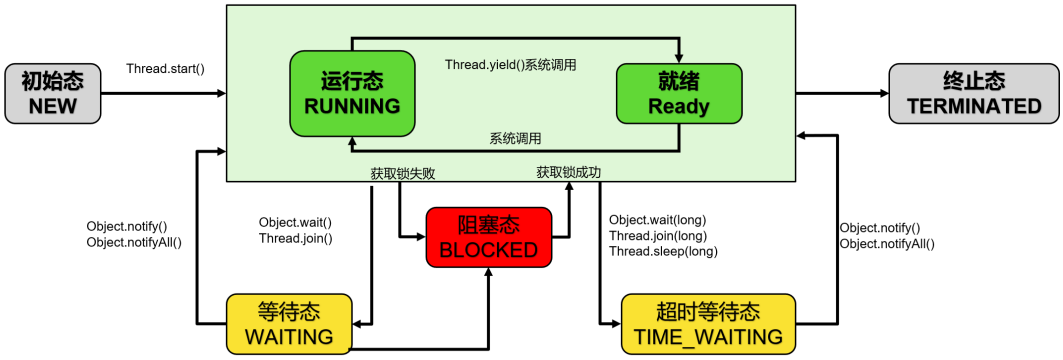
1.2.3 线程状态：一个线程的一生

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;
    }
```

- **NEW(新建)**：线程刚被创建，但是并未启动
- **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

线程状态图：



常用属性：

- 线程名称
- 线程ID：ThreadID = tid
- 线程优先级：Priority

常用方法：

- 线程让步：yield()
- 让线程休眠的方法：sleep()
- 等待线程执行终止的方法： join()
- 线程中断interrupt()
- 等待与通知系列函数wait()、notify()、notifyAll()

wait()与sleep()区别：

- 主要区别：**sleep()方法没有释放锁，而wait()方法释放了锁**

- 两者都可以暂停线程的执行
- wait()通常用于线程间的交互/通信，sleep()通常用于暂停线程执行
- wait()方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象的notify或notifyAll。sleep()方法执行完成后，线程会自动苏醒。或者可以使用wait(long)超时后，线程也会自动苏醒

1.2 多线程在JVM中的实现原理剖析

我们知道Java线程是通过行start()方法来启动的，线程启动后会执行run方法内的代码。

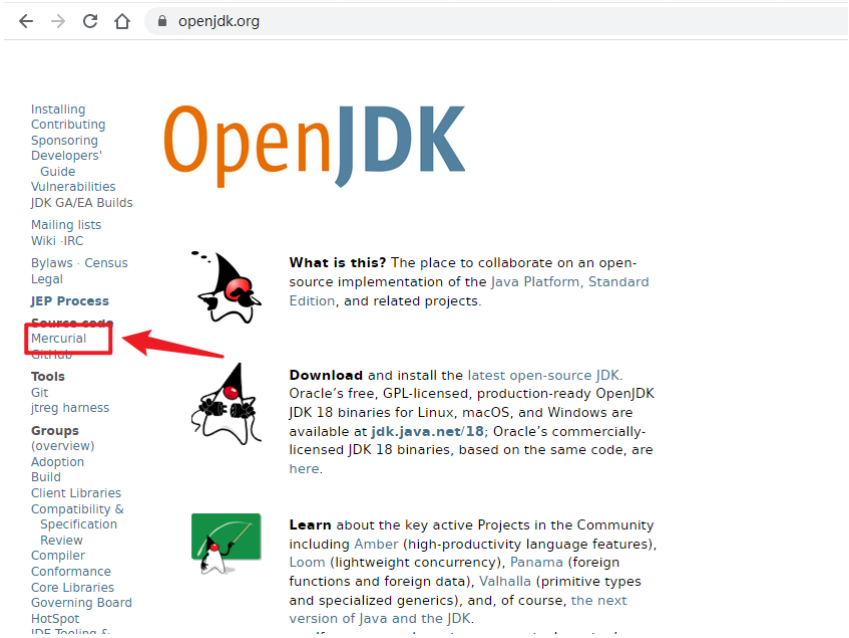
那Java线程是怎么实现run方法的执行的呢?

Java线程其实是“寄生”在操作系统线程上，通过操作系统的线程来实现Java线程的运行。接下来我们就深入源码来看看Java线程是怎么实现“寄生”在操作系统线程上来运行的。

```
1  package com.hero.multithreading;
2
3  public class ThreadDemo {
4      public static void main(String[] args) {
5          Thread thread =new Thread(()->{
6              System.out.println("线程");
7          });
8          thread.start();
9      }
10 }
```

1.2.1 下载源码

openjdk的源码是托管在 Mercurial（水银） 代码版本管理平台上的，可以使用Mercurial的代码管理工具直接从远程仓库（Repository）中下载获取源码。点击左侧菜单"Source Code" 下 “Mercurial” 进入远程仓库https://hg.openjdk.java.net



我们选用的是OpenJDK 8u，点击jdk8u进入https://hg.openjdk.java.net/jdk8u/jdk8u/，此网址为下载源。因为下载源码过程中需要执行mercurial脚本文件，所以需要在Linux平台下下载，获取源代码过程如下：

```
#需先安装代码版本管理工具
yum -y install mercurial
#安装成功后，可以用以下命令查看hg 版本信息
hg --version
#从远程仓库下载源码
hg clone https://hg.openjdk.java.net/jdk8u/jdk8u/
#赋予get_source.sh文件可执行权限
cd ./jdk8u/
chmod 755 get_source.sh
#执行文件获取源码
./get_source.sh
```

以上是下载方式，也可以直接用我下载好的源码！在课程资料中提供。

目录结构:

OpenJDK目录结构

```
jdk8u
|---corba               不常用的多语言、分布式通讯接口
|---hotspot             JVM的实现-HotSpot VM源代码
|---jaxp                用于处理XML的Java API
|---jaxws               一组XML Web Services的Java API
|---jdk                 JDK 实现
|---langtools           Java 语言工具
|---nashorn             JVM 上的 JavaScript 运行时
```

JDK实现 目录结构jdk8u/jdk

```
jdk                     JDK 实现
|---src                 源代码
|   |---share           与平台无关的公用代码
|   |   |---classes     Java API的实现
|   |   |---native      Java中相关的Native方法C++实现
#其它的目录如back、instrument、javavm、npt、transport等目录包含了实现Java的基础部分的C++源码，在这里可以从最底层理解 Java。
```

HotSpot VM目录结构：jdk8u\hotspot

```
hotspot
|---src                 HotSpot VM源代码
|   |---cpu             CPU相关代码
|   |---os              操作系统相关代码
|   |---os_cpu          操作系统+CPU组合的相关代码
|   |---share           与平台无关的公用代码
|   |   |---tools       工具
|   |   |---vm          HotSpot VM的核心代码
```

1.2.2 查看naive state0 方法

从入口开始，首先我们进入到Thread类的start方法内，可以看到有一个start0()方法的调用， 这里是真正启动Java线程的地方 。

```
public synchronized void start() {//线程安全的方法
    //线程状态判断
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    boolean started = false;
    try {
        start0();//关键步骤，主线
        started = true;
    } finally {
        try {
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
        }
    }
}
private native void start0();
```

start0是一个 native方法， 那么start0方法是在哪里实现的呢?

在openjdk源码share\native\java\lang\Thread.c 文件中我们可以找到start0的定义，Java 线程将start0方法和真正的实现方法JVM_StartThread进行了绑定。也就是说调用start0相当与调用了JVM_StartThread方法。

```
//定义了JNINativeMethod类型的数组，JNINativeMethod是一个C语言的结构体类型，在jni.h头文件中。
//数组中的在methods数组中定义了native方法，与JVM中线程方法的对应关系。
static JNINativeMethod methods[] = {
    {"start0",           "()V",        (void *)&JVM_StartThread},
    {"stop0",            "(" OBJ ")V", (void *)&JVM_StopThread},
    {"isAlive",          "()Z",        (void *)&JVM_IsThreadAlive},
    {"suspend0",         "()V",        (void *)&JVM_SuspendThread},
    {"resume0",          "()V",        (void *)&JVM_ResumeThread},
    {"setPriority0",     "(I)V",       (void *)&JVM_SetThreadPriority},
    {"yield",            "()V",        (void *)&JVM_Yield},
    {"sleep",            "(J)V",       (void *)&JVM_Sleep},
    {"currentThread",    "()" THD,     (void *)&JVM_CurrentThread},
    {"countStackFrames", "()I",        (void *)&JVM_CountStackFrames},
    {"interrupt0",       "()V",        (void *)&JVM_Interrupt},
    {"isInterrupted",    "(Z)Z",       (void *)&JVM_IsInterrupted},
    {"holdsLock",        "(" OBJ ")Z", (void *)&JVM_HoldsLock},
    {"getThreads",        "()[" THD,   (void *)&JVM_GetAllThreads},
    {"dumpThreads",      "([" THD ")[[" STE, (void *)&JVM_DumpThreads},
    {"setNativeName",    "(" STR ")V", (void *)&JVM_SetNativeThreadName},
};
//线程中registerNatives的native方法，对应的C++方法，使用了传统的JNI调用函数命名规则，
//java.lang.Thread.registerNative的方法对应的JNI函数。
//在这个方法内，完成了methods数组中的所有native方法与JVM中JNI函数的映射。
JNIEXPORT void JNICALL
```

```
Java_java_lang_Thread_registerNatives(JNIEnv *env, jclass cls)
{
    (*env)->RegisterNatives(env, cls, methods, ARRAY_LENGTH(methods));
}
```

JNINativeMethod类型的结构体变量，JNINativeMethod定义在jni.h中。定义了一个native方法和jni方法的映射关系，将Java中的native方法和JVM中真正的实现方法进行绑定。

```
typedef struct {
    char *name;//native方法
    char *signature;
    void *fnPtr;//真正的实现JNI方法
} JNINativeMethod;

```

那么这里就有一个问题，registerNatives方法具体是在哪里何时执行映射操作的呢?

在JVM首次加载Thread类的时候，在Thread类的静态初始化块中，调用了native registerNatives方法，它对应的Jni方法就是上面的Java_java_lang_Thread_registerNatives方法，就是在这里完成了state0和JVM_StartThread的绑定。

```
public class Thread implements Runnable {
    private static native void registerNatives();
    static {
        registerNatives();
    }
    ...
}
```

1.2.3 JVM_StartThread 方法

至此，我们知道执行state0方法就是执行JVM_StartThread方法，它定义在hotspot JVM源码文件`src\share\vm\prims\jvm.cpp`中。

```
JVM_ENTRY(void, JVM_StartThread(JNIEnv* env, jobject jthread))
  JVMWrapper("JVM_StartThread");
  JavaThread *native_thread = NULL;
  bool throw_illegal_thread_state = false;
  {
    //获取互斥锁
    MutexLocker mu(Threads_lock);

    //判断Java线程是否已启动，如果已启动，则抛异常。
    if (java_lang_Thread::thread(JNIHandles::resolve_non_null(jthread)) !=
NULL) {
      throw_illegal_thread_state = true;//抛出非法线程状态异常
    } else {

     //如果Java线程没有启动，创建JVM中的JavaThread
      jlong size
=java_lang_Thread::stackSize(JNIHandles::resolve_non_null(jthread));
      size_t sz = size > 0 ? (size_t) size : 0;
      native_thread = new JavaThread(&thread_entry, sz); //JVM中的JavaThread
构造函数在文件thread.cpp
```

```
18
19        if (native_thread->osthread() != NULL) {
20          //将Java中的Thread和Jvm中的Thread进行绑定。
21          native_thread->prepare(jthread);
22        }
23      }
24    }
25  ......
26    //开始启动执行JVM Thread线程。
27    Thread::start(native_thread);
28  JVM_END
```

进入java线程构造函数，在`src\share\vm\runtime\thread.cpp`中

```
1  JavaThread::JavaThread(ThreadFunction entry_point, size_t stack_sz) :
    Thread()
2  {
3    if (TraceThreadEvents) {
4      tty->print_cr("creating thread %p", this);
5    }
6    initialize();
7    _jni_attach_state = _not_attaching_via_jni;
8    set_entry_point(entry_point);
9    os::ThreadType thr_type = os::java_thread;
10   thr_type = entry_point == &compiler_thread_entry ? os::compiler_thread :
11                                                      os::java_thread;
12   //创建操作系统线程,这个才是真正映射到系统层面。
13   os::create_thread(this, thr_type, stack_sz);
14
15   _safepoint_visible = false;
16 }
```

我们着重关注os::create_thread(this, thr_type, stack_sz)，它的作用是创建Java线程对应的操作系统线程。

1.2.4 创建操作系统线程

JVM在所有的操作系统中都实现了os::create_thread，我们看linux操作系统的实现在`src\os\linux\vm\os_linux.cpp`中

```
1  bool os::create_thread(Thread* thread, ThreadType thr_type, size_t
   stack_size) {
2  ......
3      pthread_t tid;
4      //创建os级别的线程，pthread_create是linux操作系统创建线程的函数。
5      //成功则返回0，否则返回出错编号
6      //第一个参数：指向线程标识符的指针。
7      //第二个参数：用来设置线程属性。
8      //第三个参数：java_start是新建的线程运行的初始函数地址；
9      //第四个参数：java_start函数的参数。
10
11     //java_start就是新创建的线程启动入口，它会等待一个信号来调用Java中的run()方法，
```

```
    int ret = pthread_create(&tid, &attr, (void* (*)(void*)) java_start,
thread);
......
  return true;
}

//新建的操作系统线程从java_start方法开始执行。而在此函数中执行java线程对象的run方法。
static void *java_start(Thread *thread) {
......
  thread->run();//此处的作用是执行java线程对象的run方法
  return 0;
}
```

1.2.5 操作系统线程执行

至此一个操作系统线程创建及初始化完毕了，我们返回到步骤1.2.3中的JVM_StartThread 方法中，最后一行`Thread::start(native_thread);`开始执行操作系统线程。

```
//hotspot\src\share\vm\runtime\thread.cpp
void Thread::start(Thread* thread) {
  trace("start", thread);
  // Start is different from resume in that its safety is guaranteed by
context or
  // being called from a Java method synchronized on the Thread object.
  if (!DisableStartThread) {
    if (thread->is_Java_thread()) {
      // Initialize the thread state to RUNNABLE before starting this
thread.
      // Can not set it after the thread started because we do not know the
      // exact thread state at that time. It could be in MONITOR_WAIT or
      // in SLEEPING or some other state.
      //线程状态更新为RUNNABLE
      java_lang_Thread::set_thread_status(((JavaThread*)thread)-
>threadObj(),
                                          java_lang_Thread::RUNNABLE);
    }
    os::start_thread(thread);
  }
}

//hotspot\src\share\vm\runtime\os.cpp
void os::start_thread(Thread* thread) {
  // guard suspend/resume
  MutexLockerEx ml(thread->SR_lock(), Mutex::_no_safepoint_check_flag);
  OSThread* osthread = thread->osthread();
  //操作系统线程状态置为RUNNABLE
  osthread->set_state(RUNNABLE);
  pd_start_thread(thread);
}

//hotspot\src\os\linux\vm\os_linux.cpp
void os::pd_start_thread(Thread* thread) {
  OSThread * osthread = thread->osthread();
  assert(osthread->get_state() != INITIALIZED, "just checking");
  Monitor* sync_with_child = osthread->startThread_lock();
```

```
  MutexLockerEx ml(sync_with_child, Mutex::_no_safepoint_check_flag);
  sync_with_child->notify();
}
```

至此，操作系统线程为就绪状态，等待被CPU选中运行时，就会调用执行入口函数java_start，调用Java线程的run方法，至此Java线程也就同时运行起来了。

小结一下

1. 线程类被JVM加载时，完成线程所有native方法和C++中的对应方法绑定。
2. **Java线程调用start方法：**start方法 ==> native state0方法 ==> JVM_StartThread ==> 创建JavaThread::JavaThread线程
3. **创建OS线程，并指定OS线程的运行入口：**创建JavaThread::JavaThread线程 ==> 创建OS线程 os::create_thread ==> 指定OS线程执行入口Java线程的run方法
4. **启动OS线程：**运行时会调用Java线程的run方法，至此实现了Java线程的运行。
5. 创建线程的时候使用的是互斥锁MutexLocker操作系统（互斥量），所以说创建线程是一个性能很差的操作！

1.3 线程安全问题【关键】

什么是线程安全?

如果有多个线程在同时执行，而多个线程可能会同时运行一行代码。如果程序每次运行结果和单线程运行的结果一样，且其他的变量的值也和预期一样，就是线程安全的，反之则是线程不安全的。

举个栗子:

1.3.1 经典案例：卖票

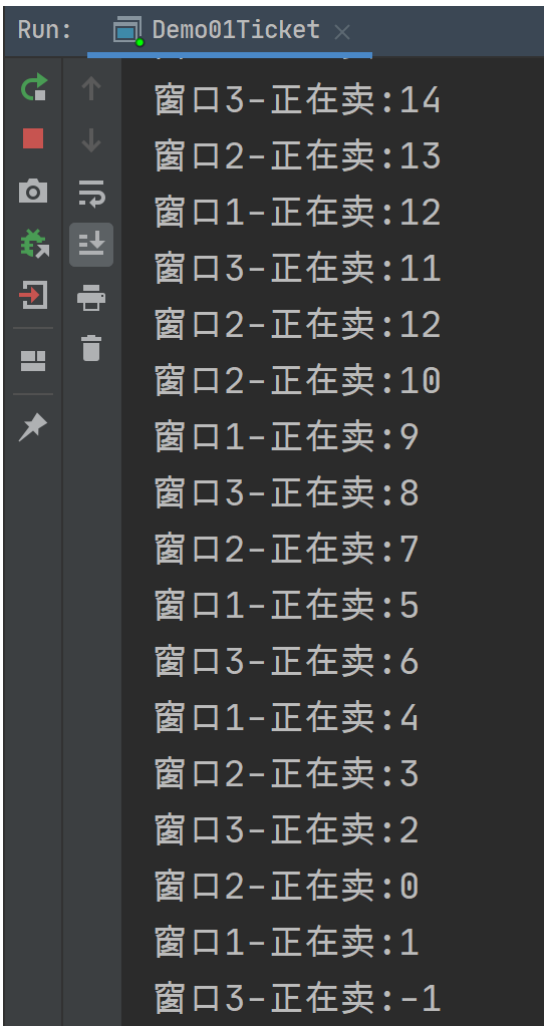
假设电影院要播放的电影是 “独行月球”，电影院要卖电影票，我们采用多线程程序模拟电影院卖票过程。一场电影的座位共100个，每个座位一张票。公有三个电影票售票窗口。要求实现多个窗口同时卖“独行月球”这场电影票。卖完为止，不可多卖也不可以有余票。

代码实现:

```
package com.hero.multithreading;

public class Demo01Ticket {

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

        //同时卖票
        t1.start();
        t2.start();
        t3.start();
    }
```

```
18
19
20  }
21  class SellTicketTask implements Runnable {
22      //电影票100张
23      private int tickets = 100;
24      /*
25       * 每个窗口执行相同的卖票操作
26       * 窗口永远开启，所有窗口卖完100张票为止
27       */
28      @Override
29      public void run() {
30          while (true) {
31              //有票 可以卖
32              if (tickets > 0){
33                  //模拟出票时间：使用sleep模拟一下出票时间
34                  try {
35                      Thread.sleep(20);
36                  } catch (InterruptedException e) {
37                      e.printStackTrace();
38                  }
39                  //获取当前线程对象的名字
40                  String name = Thread.currentThread().getName();
41                  System.out.println(name + "-正在卖:" + tickets--);
42              }
43          }
44      }
45  }
```

结果中有一部分这样现象：



发现程序出现了两个问题：

1. 相同的票数：比如12这张票被卖了两回。
2. 不存在的票：比如0票与-1票，是不存在的。

这种问题，几个窗口(线程)票数不同步了，这种问题称为线程不安全。

引发线程安全问题：线程安全问题都是由全局变量及静态变量【共享】引起的。

- 如果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；
- 如果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问题。

怎么解决这个问题呢?

- 线程同步
- volatile
- JUC
    - 原子类（CAS）
    - 锁（AQS）

1.3.2 线程同步

为了保证不出现线程安全问题，Java引入了线程同步机制（**synchronized**）。**那么怎么完成同步操作呢?**

> 这个是JUC的起源也是因为同步机制不太行！JUC的出现就是因为它！

1. **同步代码块Synchronized-重量级锁**
2. **同步方法Synchronized-重量级锁**
3. **锁机制【JUC】**

同步代码块

```
synchronized(同步锁){
    需要同步操作的代码
}
```

举例:

```
class SellTicketTaskSynchronized implements Runnable {
    //电影票
    private int tickets = 100;
    private final Object lock = new Object();//锁对象，可以是任意类型数据
    /*
     * 每个窗口执行相同的卖票操作
     * 窗口永远开启，所有窗口卖完100张票为止
     */
    @Override
    public void run() {
        while (true) {
            synchronized (lock){
                //有票 可以卖
                if (tickets > 0){
                    //模拟出票时间：使用sleep模拟一下出票时间
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    //获取当前线程对象的名字
                    String name = Thread.currentThread().getName();
                    System.out.println(name + "-正在卖:" + tickets--);
                }
```

```
            }
        }
    }
}
```

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
}
```

Lock锁

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

1.4 多线程并发的3个特性

并发编程中，三个非常重要的特性：**原子性，有序性和可见性**

1. **原子性**：即一个操作或多个操作，要么全部执行，要么就都不执。执行过程中，不能被打断
2. **有序性**：程序代码按照先后顺序执行
    - **为什么会出现无序问题呢?** 因为**指令重排**
3. **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
    - **为什么出现不可见性问题呢?** 可以说是因为Java内存模型【JMM】

1.5 有序性案例：指令重排序

什么是指令重排序?

重排序是编译器和处理器为了提高程序运行效率，会对输入代码进行优化的一种手段。它不保证程序中，各个语句执行先后顺序的一致。

举个栗子：

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

在单线程中以上代码从顺序上看，语句1是在语句2前面的，那么**JVM在真正执行这段代码的时候，会保证语句1一定在语句2前面执行吗?**

- 还真不一定！**为什么呢?** 这里可能会发生**指令重排序（Instruction Reorder）。**
- 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，在**单线程下遵循as-if-serial语义。**

按顺序执行不好么，为什么要重排序去执行?

- 不进行指令重排，就相当于没有编译优化，那么程序执行效率打折扣
- 当前线程获取CPU时间片（几十毫秒），如果按照先后顺序执行，并不能把CPU性能发挥完全，上一个指令执行完执行下一个，CPU会出现空挡期。

什么是as-if-serial语义?

- 不管编译器和处理器怎么优化字节码指令，怎样进行指令重排，单线程所执行的结果不能受影响。
- 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。

再举个单线程栗子：

```
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**
- 处理器在进行重排序时，会考虑指令之间的数据依赖性，如果一个 **指令2** 必须用到 **指令1** 的结果，那么处理器会保证**指令1**会在**指令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。

再举个多线程的栗子：

```
//线程1:
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，而此是线程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。**

要想多线程程序正确地执行，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，就有可能会导致程序运行不正确。

1.6 可见性案例：Java内存模型（JMM）

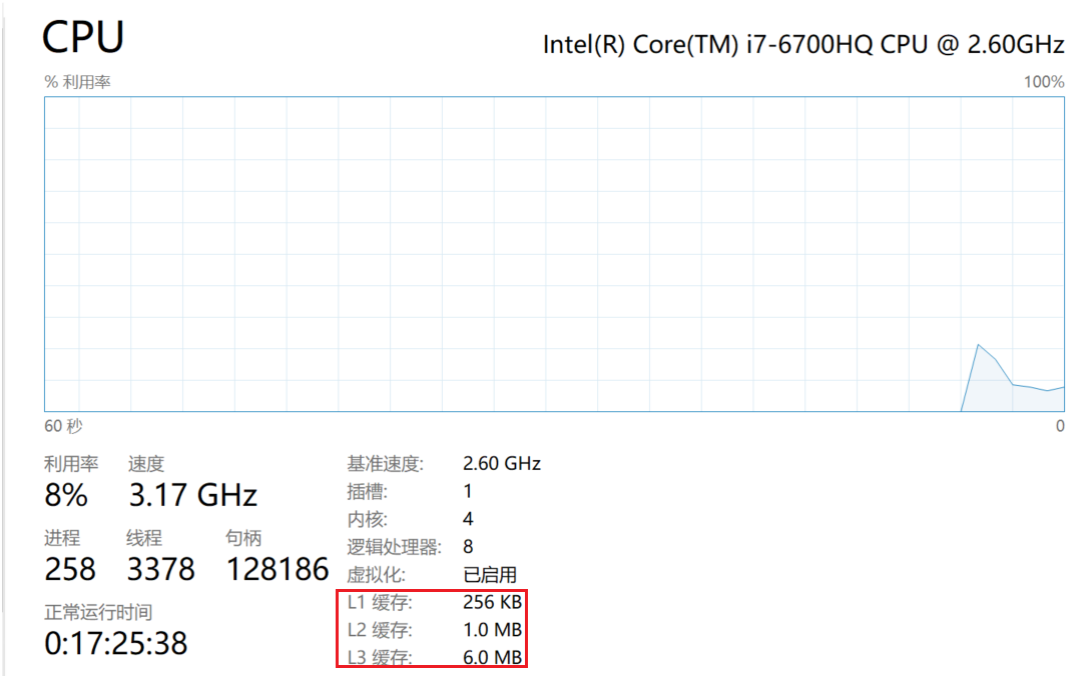
1.6.1 CPU和缓存一致性

在多核 CPU 中，每个核的自己的缓存，关于同一个数据的缓存内容可能不一致。

计算机在执行程序的时候，每条指令都是在 CPU 中执行的，而执行的时候，又免不了和数据打交道，而计算机上面的数据，是存放在计算机的物理内存上的。

当内存的读取速度和CPU的执行速度相比差别不大的时候，这样的机制是没有任何问题的，可是随着CPU的技术的发展，CPU的执行速度和内存的读取速度差距越来越大，导致CPU每次操作内存都要耗费很多等待时间。

为了解决这个问题，在**CPU和物理内存上新增高速缓存**，这样程序的执行过程也就发生了改变，变成了程序在运行过程中，会将运算所需要的数据从主内存复制一份到CPU的高速缓存中，当CPU进行计算时就可以直接从高速缓存中读数据和写数据了，当运算结束再将数据刷新到主内存就可以了。



随着技术的发展，CPU开始出现了多核的概念，每个核都有一套自己的缓存，并且随着计算机能力不断提升，还开始支持多线程，最终演变成，多个线程访问进程中的某个共享内存，且这多个线程分别在不同的核心上执行，则每个核心都会在各自的 Cache 中保留一份共享内存的缓冲，我们知道多核是可以并行的，这样就会出现多个线程同时写各自的缓存的情况，**导致各自的 Cache 之间的数据不一致性问题。**

如何保证？JMM模型

1.6.2 Java内存模型（JMM）

背景：

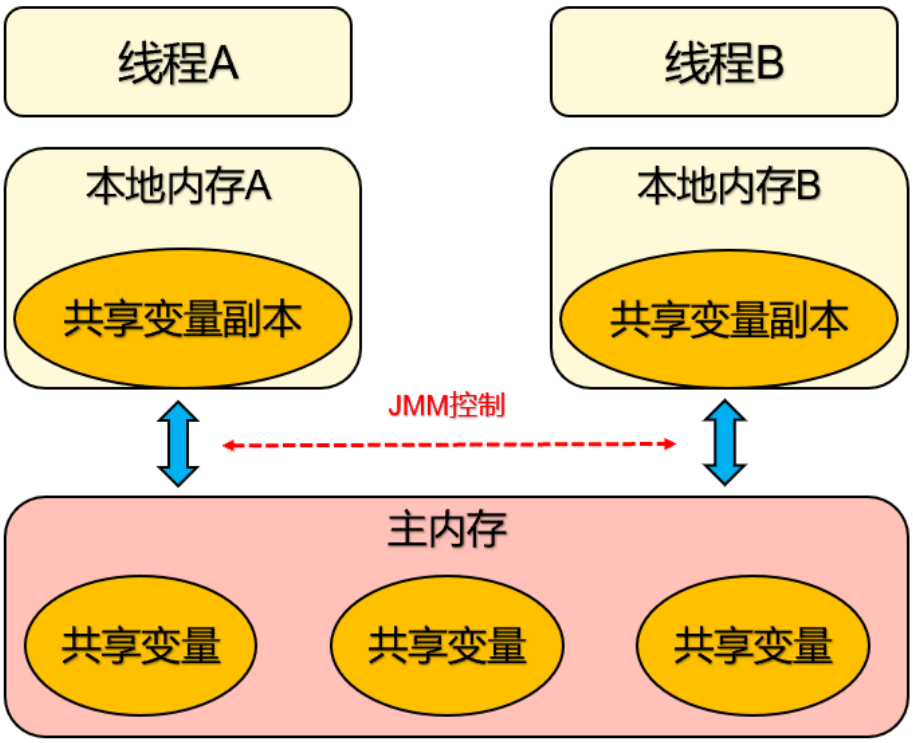
- **CPU和缓存一致性**
- **处理器优化和指令重排**

Java为了保证并发编程中可以满足原子性、可见性及有序性，诞生出了一个重要的概念，那就是**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。**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。

Java实现了JMM规范保证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，JMM在Java中的实现**屏蔽了各种硬件和操作系统的访问差异。**

内存模型怎么解决并发问题的?

- 内存模型解决并发问题主要采取两种方式，分别是**限制处理器优化**，另一种是**使用了内存屏障。**
- 而对于这两种方式，Java底层其实已经封装好了一些关键字，我们只需要用起来就可以了，不需要关注底层具体如何实现。
- 关于解决并发编程中的原子性问题，Java底层封装了Synchronized的方式，来保证方法和代码块内的操作都是原子性的；
- 而至于可见性问题，Java底层则封装了Volatile的方式，将被修饰的变量在修改后立即同步到主内存中。
- 至于有序性问题，其实也就是我们所说的重排序问题，Volatile关键字也会禁止指令的重排序，而Synchroinzed关键字由于保证了同一时刻只允许一条线程操作，自然也就保证了有序性。



- **JMM定义一个共享变量何时写入，何时对另一个线程可见**
- 线程之间的共享变量存储在主内存：

    - 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**）。
    - 由于主内存是共享数据区，多条线程对同一个变量访问会出现线程安全问题。
- 每个线程都有一个私有的本地内存，本地内存存储共享变量的副本
    - 主要存储**当前方法的所有本地变量**，每个线程只能访问自己的本地内存。
    - **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同一段代码，它们也会在自己的本地内存中，创建属于自己线程的本地变量。
- **本地内存是抽象概念涵盖：缓存，写缓冲区，寄存器等**

JMM线程操作内存的基本规则：

- 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本地内存中进行，不能直接从主内存中读写
- 第二条，关于线程间本地内存：不同线程之间无法直接访问其他线程本地内存中的变量，线程间变量值的传递需要经过主内存

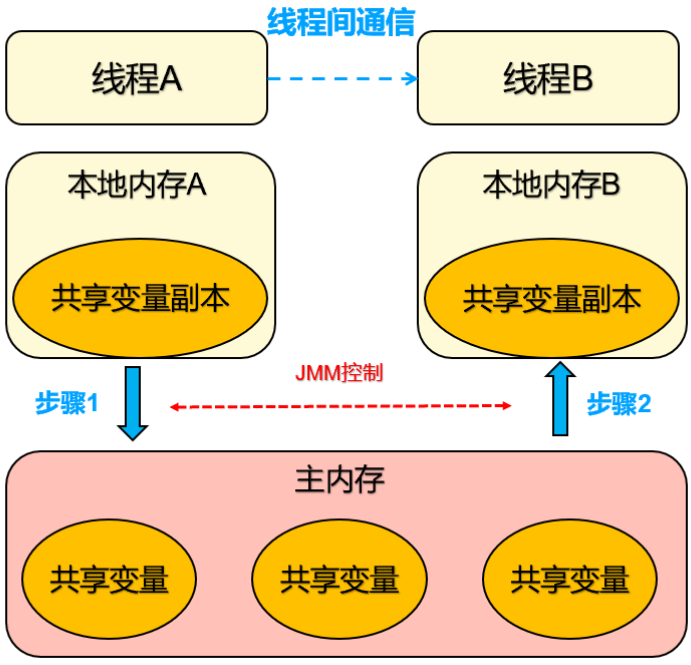
1.6.3 什么是内存可见性?

可见性是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。

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。

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 程序提供内存可见性的保证。



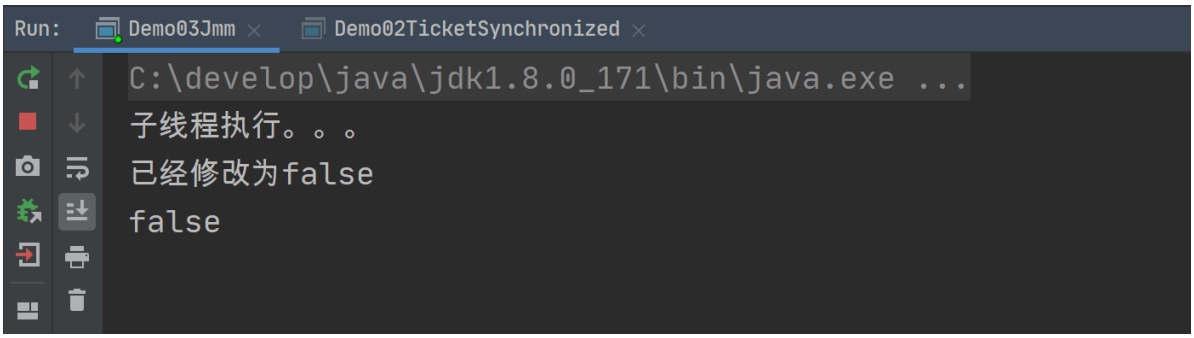
1.6.4 可见性案例

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代码

案例如下：

```
public class Demo02Jmm {

    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。

原因是：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

怎么解决?

- 使用Synchronized同步代码块
- 彻底禁止JMM？禁止重排序和读取本地内存副本
- happens-before规则：按需使用重排序和本地内存副本，前提是需要满足**happens-before规则**

1.6.5 happens-before规则

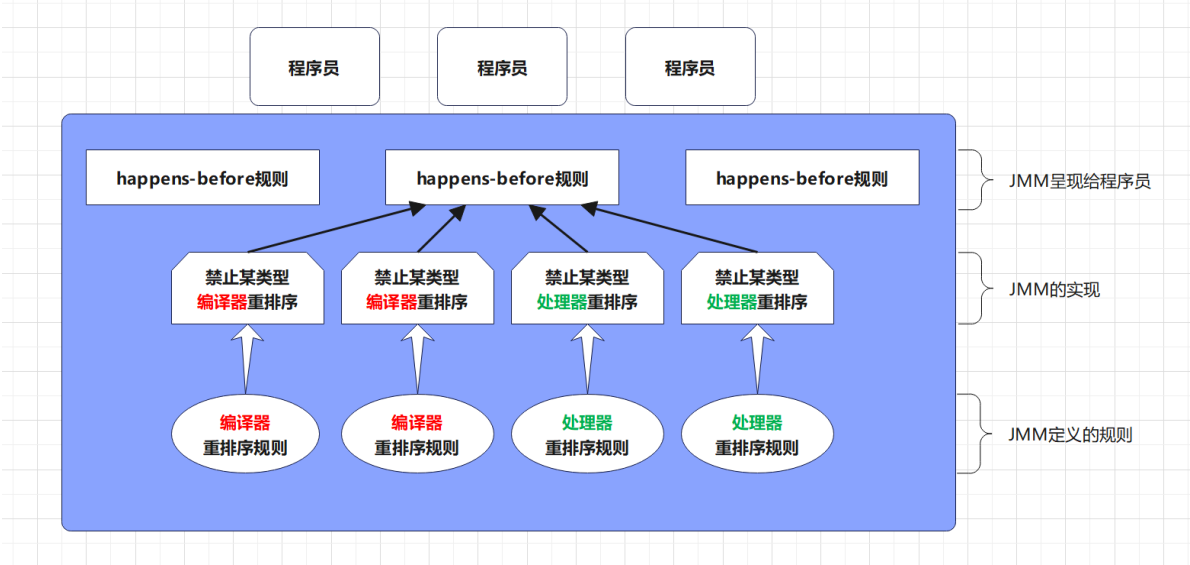
在JMM中，使用happens-before规则来约束编译器的优化行为，允许编译期优化，但需要遵守一定的Happens-Before规则。**如果一个操作执行的结果需要对另一个操作可见**，那么这两个操作之间必须要存在happens-before的关系！

程序员需关注的happens-before规则：

- 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。
- 锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。
- volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。
- 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

happens-before的实现：1.处理器重排序规则，2.编译器重排序规则

> 注意：对于程序员来说，理解以上happens-before规则即可，JMM设计happens-before的目标就是屏蔽编译器和处理器重排序规则的复杂性。



2. synchronized

在并发编程中，synchronized大家都肯定用过，一般情况下，我们会把synchronized称为重量级锁。主要原因，是因为JDK1.6之前，synchronized是一个重量级锁相比于JUC的锁显得非常笨重，存在性能问题。JDK1.6及之后，Java对synchronized进行的了一系列优化，性能与JUC的锁不相上下

synchronized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代码块，还可以保证共享变量的内存可见性，也可以保证修饰的代码块重排序也不会影响其执行结果。

一句话：synchronized可以保证并发程序的**原子性，可见性，有序性。**

synchronized可以修饰方法和代码块。

- 方法：可修饰静态方法和非静态方法
- 代码块：同步代码块的锁对象可以为当前实例对象、字节码对象（class）、其他实例对象

2.1 如何解决可见性问题？

JMM关于synchronized的两条规定：

- 线程解锁前：必须把自己本地内存中共享变量的最新值刷新到主内存中
- 线程加锁时：将清空本地内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值

在可见性案例中，做如下修改：

```
while (flag) {
    //在死循环中添加同步代码块，可以解决可见性问题
    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

2.2 同步原理剖析

synchronized是如何实现同步的呢?

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 class Demo05Synchronized {
    public synchronized void increase(){
        System.out.println("synchronized 方法");
    }
    public void syncBlock(){
        synchronized (this){
            System.out.println("synchronized 块");
        }
    }
}

```

在cmd命令行执行javac编译和`javap -c ***.class`生成class文件对应的字节码指令

```
javac Demo05Synchronized.java
javap -c Demo05Synchronized.class
```

从结果可以看出，同步代码块使用的是monitorenter和monitorexit这两个jvm指令

```

```

```
//同步方法
public synchronized void increase();
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED //ACC_SYNCHRONIZED标记
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2  // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3  // String synchronized 方法
         5: invokevirtual #4  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
         8: return

 //同步块
  public void syncBlock();
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=1
         0: aload_0
         1: dup
         2: astore_1
         3: monitorenter      //monitorenter指令进入同步块
         4: getstatic     #2  // Field
java/lang/System.out:Ljava/io/PrintStream;
         7: ldc           #5  // String synchronized 块
         9: invokevirtual #4  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
        12: aload_1
        13: monitorexit       //monitorexit指令退出同步块
        14: goto          22
        17: astore_2
        18: aload_1
        19: monitorexit       //monitorexit指令退出同步块
        20: aload_2
        21: athrow
        22: return
```

从上述字节码指令看的到，**同步代码块**和**同步方法**的字节码是不同的

- 对于synchronized同步块，对应的monitorenter和monitorexit指令分别对应synchronized同步块的进入和退出。
    - **为什么会多一个monitorexit?** 编译器会为同步块添加一个隐式的try-finally，在finally中会调用monitorexit命令释放锁
- 对于synchronized方法，对应ACC_SYNCHRONIZED关键字，JVM进行方法调用时，发现调用的方法被ACC_SYNCHRONIZED修饰，则会先尝试获得锁，方法调用结束了释放锁。在JVM底层，对于这两种synchronized的实现大致相同。**都是基于monitorenter和monitorexit指令实现**，底层还是使用 标记字段MarkWord和**Monitor（管程）**来实现重量级锁。

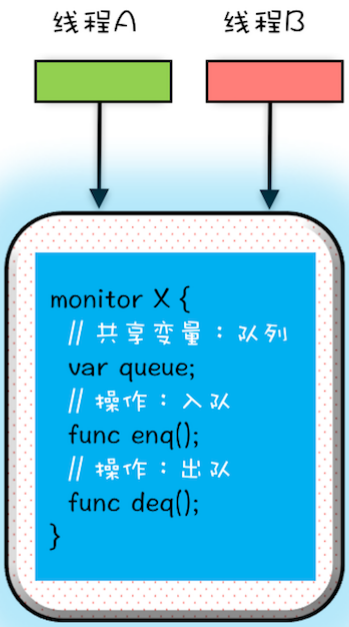
2.3 什么是Monitor?

- Monitor中文翻译为管程，也有人称之为“监视器”，管程指的是管理共享变量以及对共享变量的操作过程，让他们支持并发。
- Java中的所有对象都可以作为锁，每个对象都与一个 monitor 相关联，线程可以对 monitor 执行 lock 和 unlock 操作。
- Java并没有把lock和unlock操作直接开放给用户使用，但是却提供了两个指令来隐式地使用这两个操作：moniterenter和moniterexit。moniterenter对应lock操作，moniterexit对应unlock操作，通过这两个指锁定和解锁 monitor 对象来实现同步。
- 当一个monitor对象被线程持有后，它将处于锁定状态。对于一个 monitor 而言，同时只能有一个线程能锁定monitor，其它线程试图获得已被锁定的 monitor时，都将被阻塞。当monitor被释放后，阻塞中的线程会尝试获得该 monitor锁。一个线程可以对一个 monitor 反复执行 lock 操作，对应的释放锁时，需要执行相同次数的 unlock 操作。

> 详细了解Monitor请参考：极客时间专栏《Java并发编程实战-王宝令》管程：并发编程的万能钥匙
> 文章：https://time.geekbang.org/column/article/86089

Monitor如何解决了线程安全问题?

管程解决互斥问题的思路：就是将共享变量及其对共享变量的操作统一封装起来。



2.4 什么是锁优化?

- 在JDK 1.6之前，synchronized使用传统的锁（重量级锁）实现。它依赖于操作系统（互斥量）的同步机制，涉及到用户态和内核态的切换、线程的上下文切换，性能开销较高，所以给开发者留下了synchronized关键字性能不好的印象。
- 如果**只有一个线程运行时**并没有发生资源竞争、或**两个线程交替执行**，使用传统锁机制无疑效率是会比较低的。
- JDK1.6中为了减少这两个场景，获得锁和释放锁带来的性能消耗，同步锁进行优化引入：**偏向锁和轻量级锁。**

- 同步锁一共有四种状态，级别从低到高依次是：**无锁，偏向锁，轻量级锁，重量级锁。**这四种状态会随着竞争激烈情况逐渐升级。

偏向锁

偏向锁则是基这样一个想法：只有一个线程访问锁资源（无竞争）的话，偏向锁就会把整个同步措施都消除，并记录当前持有锁资源的线程和锁的类型。

轻量级锁

轻量级锁是基于这样一个想法：只有两个线程交替运行时，如果线程竞争锁失败了，先不立即挂起，而是让它飞一会儿（自旋），在等待过程中，可能锁就被释放了，这时该线程就可以重新尝试获取锁，同时记录持有锁资源的线程和锁的类型。

那锁信息存储在哪?

例如：锁类型，当前持有线程

偏向锁标记	锁状态标识	锁状态
0	01	无锁
1	01	偏向锁
无	00	轻量锁
无	10	重量锁
无	11	GC标记

同步锁锁定资源是对象，那无疑存储在对象信息中，由对象直接携带，是最方便管理和操作的。

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁状态	对象Hashcode、对象GC分代年龄				01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空，不需要记录信息				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01

> 32位操作系统的Markword

3. volatile

通过前面的内容，咱们了解了synchronized同步代码块，同步代码块在多线程场景下存在性能问题。接下来介绍一个轻量级的线程安全问题解决方案 **volatile** ，它比使用synchronized的成本更加低。

Java语言对volatile的定义：**Java允许线程访问共享变量，为了确保共享变量能被准确和一致地更新，线程应该确保通过排他锁单独获得这个变量。**

说人话：volatile可以保证多线程场景下变量的**可见性和有序性。**如果某变量用volatile修饰，则可以确保所有线程看到变量的值是一致的。

- 可见性：保证此变量的修改对所有线程的可见性。
- 有序性：禁止指令重排序优化，编译器和处理器在进行指令优化时，不能把在volatile变量操作(读/写)后面的语句放到其前面执行，也不能将volatile变量操作前面的语句放在其后执行。**遵循了JMM的happens-before规则**

> 注：volatile虽然看起来比较简单，无非就是在某个变量前加上volatile，但要用好并不容易！

3.1 解决内存可见性问题

在可见性案例中，做如下修改：

```
// 添加volatile关键词
private volatile boolean flag = true;
```

volatile实现内存可见性的过程

线程写volatile变量的过程：

1. 改变线程本地内存中volatile变量副本的值；
2. 将改变后的副本的值从本地内存刷新到主内存

线程读volatile变量的过程：

1. 从主内存中读取volatile变量的最新值到线程的本地内存中
2. 从本地内存中读取volatile变量的副本

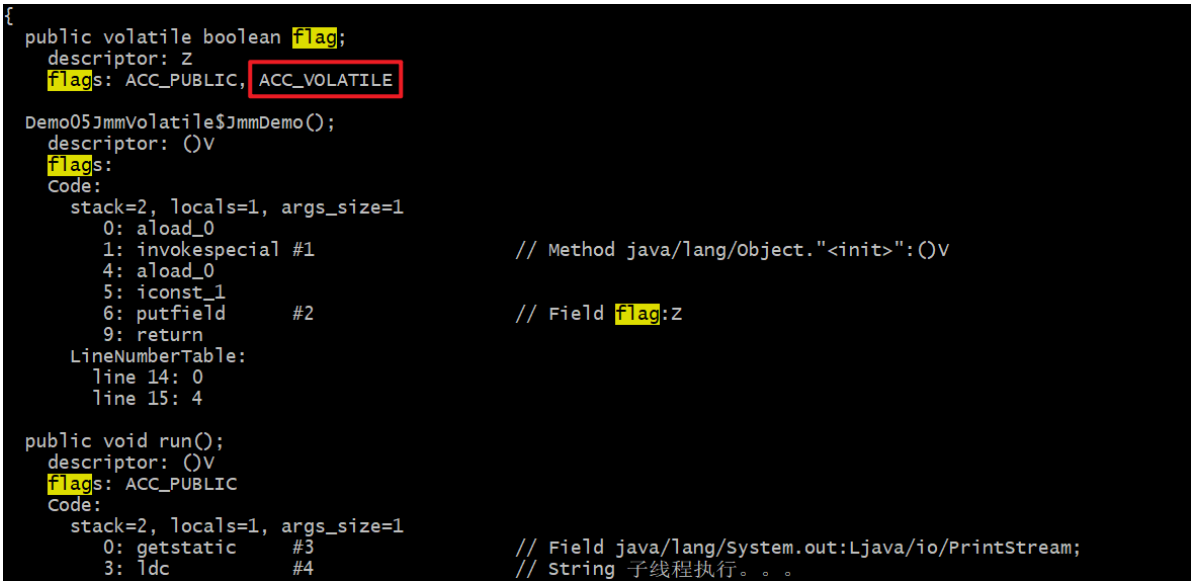
3.2 volatile实现原理-源码分析

volatile实现内存可见性原理：内存屏障（Memory Barrier）

内存屏障（Memory Barrier）是一种**CPU指令**，用于控制特定条件下的重排序和内存可见性问题。Java编译器也会根据内存屏障的规则禁止重排序

- 写操作时，通过在写操作指令后加入一条store屏障指令，让本地内存中变量的值能够刷新到主内存中
- 读操作时，通过在读操作前加入一条load屏障指令，及时读取到变量在主内存的值

我们可以从源码角度，来理解volatile的可见性和有序性。



有volatile修饰的变量，通过javap可以看到volatile字节码有个关键字ACC_VOLATILE，通过这个关键字定位到hotspot JVM源码文件`src\share\vm\utilities\accessFlags.hpp`文件，代码如下：

```
 public:
  // Java access flags
  bool is_public      () const         { return (_flags & JVM_ACC_PUBLIC
) != 0; }
  bool is_private     () const         { return (_flags & JVM_ACC_PRIVATE
  ) != 0; }
  bool is_protected   () const         { return (_flags & JVM_ACC_PROTECTED
  ) != 0; }
  bool is_static      () const         { return (_flags & JVM_ACC_STATIC
) != 0; }
  bool is_final       () const         { return (_flags & JVM_ACC_FINAL
  ) != 0; }
  bool is_synchronized() const         { return (_flags &
JVM_ACC_SYNCHRONIZED) != 0; }
  bool is_super       () const         { return (_flags & JVM_ACC_SUPER
  ) != 0; }
  bool is_volatile    () const         { return (_flags & JVM_ACC_VOLATILE
) != 0; }
  bool is_transient   () const         { return (_flags & JVM_ACC_TRANSIENT
  ) != 0; }
  bool is_native      () const         { return (_flags & JVM_ACC_NATIVE
) != 0; }
  bool is_interface   () const         { return (_flags & JVM_ACC_INTERFACE
  ) != 0; }
  bool is_abstract    () const         { return (_flags & JVM_ACC_ABSTRACT
) != 0; }
  bool is_strict      () const         { return (_flags & JVM_ACC_STRICT
) != 0; }
```

可以看到

```
bool is_volatile () const { return (_flags & JVM_ACC_VOLATILE ) != 0; }
```

再根据关键字is_volatile搜索，在`src\share\vm\interpreter\bytecodeInterpreter.cpp`可以看到如下代码：

```
//
// Now store the result
//
int field_offset = cache->f2_as_index();
if (cache->is_volatile()) {
    if (tos_type == itos) {
        obj->release_int_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == atos) {
        VERIFY_OOP(STACK_OBJECT(-1));
        obj->release_obj_field_put(field_offset, STACK_OBJECT(-1));
        OrderAccess::release_store(&BYTE_MAP_BASE[(uintptr_t)obj >>
CardTableModRefBS::card_shift], 0);
    } else if (tos_type == btos) {
        obj->release_byte_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == ltos) {
        obj->release_long_field_put(field_offset, STACK_LONG(-1));
    } else if (tos_type == ctos) {
        obj->release_char_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == stos) {
        obj->release_short_field_put(field_offset, STACK_INT(-1));
    } else if (tos_type == ftos) {
        obj->release_float_field_put(field_offset, STACK_FLOAT(-1));
    } else {
        obj->release_double_field_put(field_offset, STACK_DOUBLE(-1));
    }
    OrderAccess::storeload();
}
```

在这段代码中，会先判断tos_type（volatile变量类型），后面有不同的基础类型的调用，比如int类型就调用release_int_field_put，byte就调用release_byte_field_put等等。

我们可以看到后面执行的语句是

```
OrderAccess::storeload();
```

可以在`src\share\vm\runtime\orderAccess.hpp`找到对应的实现方法：

```
static void storeload();
```

实际上这个方法的实现针对不同的CPU有不同的实现的，在`src/os_cpu`目录下可以看到不同的实现，以`src\os_cpu\linux_x86\vm\orderAccess_linux_x86.inline.hpp`为例，是这么实现的：

```
inline void OrderAccess::loadload()   { acquire(); }
inline void OrderAccess::storestore() { release(); }
inline void OrderAccess::loadstore()  { acquire(); }
inline void OrderAccess::storeload()  { fence(); }
```

fence()函数的实现：

```
inline void OrderAccess::fence() {
  if (os::is_MP()) {
    // always use locked addl since mfence is sometimes expensive
#ifdef AMD64
    __asm__ volatile ("lock; addl $0,0(%%)" : : : "cc", "memory");
#else
    __asm__ volatile ("lock; addl $0,0(%%esp)" : : : "cc", "memory");
#endif
  }
}
```

通过这面代码可以看到`lock; addl`，其实这个就是内存屏障。`lock; addl $0,0(%%esp)`作为cpu的一个内存屏障。

`addl $0,0(%%rsp)`表示：将数值0加到rsp寄存器中，而该寄存器指向栈顶的内存单元。加上一个0，rsp寄存器的数值依然不变。即这是一条无用的汇编指令。在此利用addl指令来配合lock指令，用作cpu的内存屏障。

Java编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。为保证在任意处理器平台下能得到正确的 volatile 内存操作语义，JMM 采取保守策略，下面是基于保守策略的 JMM 内存屏障插入策略：

- 在每个 volatile 写前，插入StoreStore 屏障。
- 在每个 volatile 写后，插入StoreLoad 屏障。
- 在每个 volatile 读后，插入LoadLoad 屏障。
- 在每个 volatile 读后，插入LoadStore 屏障。

结合JMM中四类内存屏障的作用，我们可以得出下面的结论：

屏障类型	示例	说明
LoadLoad	Load1;LoadLoad;Load2	确保装载动作Load1，先于 Load2及其后所有Load操作。 对于屏障前后的Store操作并无影响。
StoreStore	Store1;StoreStore;Store2	确保Store1刷新数据到内存(使数据对其他处理器可见)的操作，先于Store2及其后所有Store指令的执行。 对于屏障前后的Load操作并无影响。
LoadStore	Load1;LoadStore;Store2	确保屏障指令之前的所有Load操作，先于屏障之后所有Store操作(刷新数据到主存)。
StoreLoad	Store1;StoreLoad;Load2	确保屏障之前的所有内存访问操作(包括Store和Load)完成之后，才执行屏障之后的内存访问操作。 全能型屏障，会屏蔽屏障前后所有指令的重排。

重排序规则表：

是否允许重排序	第二个操作		
第一个操作	普通读/写	Volatile读	Volatile写
普通读	Yes	Yes	Yes
普通写	Yes	Yes	No
Volatile读	No	No	No
Volatile写	No	No	No

1. 当第一个操作是**volatile读**时，不管第二个操作是什么，都不能重排序
    - 确保 volatile 读到的是最新值：volatile 读之后的操作不会被编译器重排序到 volatile 读之前
2. 当第一个操作是**volatile写**时，不管第二个操作是什么，都不能重排序
    - 确保 volatile 写操作对之后的操作可见
3. 当第二个操作是**volatile写**时，第一个操作是普通写时，不能重排序

3.3 volatile缺陷：原子性Bug

原子性的问题：虽然volatile可以保证可见性，但是不能满足原子性

```
package com.hero.multithreading;

public class Demo06Volatile {
    public static void main(String[] args) throws InterruptedException {
        VolatileDemo demo = new VolatileDemo();

        for (int i = 0; i < 2; i++) {
            Thread t = new Thread(demo);
            t.start();
        }

        Thread.sleep(1000);
        System.out.println("count = "+demo.count);
    }

    static class VolatileDemo implements Runnable {
        public volatile int count;
        //public volatile AtomicInteger count = new AtomicInteger(0);

        public void run() {
            addCount();
        }

        public void addCount() {
            for (int i = 0; i < 10000; i++) {
                count++;//但是实际情况是三条汇编指令
            }
        }
    }
}

```

结果：count = 12205

不应该是20000吗？问题分析：

以上出现原子性问题的原因是count++并不是原子性操作。

count = 5 开始，流程分析：

1. 线程1读取count的值为5
2. 线程2读取count的值为5
3. 线程2加1操作
4. 线程2最新count的值为6
5. 线程2写入值到主内存的最新值为6
6. 线程1执行加1count=6，写入到主内存的值是6。
7. 结果：对count进行了两次加1操作，主内存实际上只是加1一次。结果为6

解决方案：

1. 使用synchronized
2. 使用ReentrantLock（可重入锁）
3. 使用AtomicInteger（原子操作）

使用synchronized

```
public synchronized void addCount() {
    for (int i = 0; i < 10000; i++) {
        count++;
    }
}
```

使用ReentrantLock（可重入锁）

```
//可重入锁
private Lock lock = new ReentrantLock();

public void addCount() {
    for (int i = 0; i < 10000; i++) {
        lock.lock();
        count++;
        lock.unlock();
    }
}
```

使用AtomicInteger（原子操作）

```
public static AtomicInteger count = new AtomicInteger(0);
public void addCount() {
    for (int i = 0; i < 10000; i++) {
        //count++;
        count.incrementAndGet();
    }
}
```

3.4 volatile适合使用场景

变量真正独立于其他变量和自己以前的值，在单独使用的时候，适合用volatile

- 对变量的写入操作不依赖其当前值：例如++和--运算符的场景则不行
- 该变量没有包含在具有其他变量的不变式中

3.5 synchronized和volatile比较

- volatile不需要加锁，比synchronized更轻便，不会阻塞线程
- synchronized既能保证可见性，又能保证原子性，而volatile只能保证可见性，无法保证原子性
- 与synchronized相比volatile是一种非常简单的同步机制

总结

01-多线程相关概念

- **并发（Concurrent）：**同一时间段，多个任务都在执行，单位时间内不一定同时执行
- **并行（Parallel）：**单位时间内，多个任务同时执行，单位时间内一定是同时执行
  - 并发是一种能力，并行是一种手段
- **线程上下文切换：**CPU为了提升利用率，**采用时间片算法**将CPU时间片轮流分配给多个线程，多个线程在一个时间片中未执行完毕就会进行线程上下文切换，切换的过程中会记录当前线程在上一个时间片中执行的位置然后挂起待下次执行。
- **线程生命周期：**NEW（新建）、RUNNABLE（可运行）、TERMINATED（终止态）、BLOCKED（锁阻塞）、WAITING（无限等待）、TIMED_WAITING（计时等待）
- **多线程在JVM中的实现原理剖析：**Java线程是怎么实现run方法的执行的
  - 线程类被JVM加载时，完成线程所有native方法和C++中的对应方法绑定。
  - **Java线程调用start方法：**start方法 ==> native state0方法 ==> JVM_StartThread ==> 创建JavaThread::JavaThread线程
  - **创建OS线程，并指定OS线程的运行入口：**创建JavaThread::JavaThread线程 ==> 创建OS线程os::create_thread ==> 指定OS线程执行入口Java线程的run方法
  - **启动OS线程：**运行时会调用Java线程的run方法，至此实现了Java线程的运行。
  - 创建线程的时候使用的是互斥锁MutexLocker操作系统（互斥量），所以说创建线程是一个性能很差的操作！
- **线程安全问题：**如果有多个线程在同时执行，而多个线程可能会同时运行一行代码。如果程序每次运行结果和单线程运行的结果一样，且其他的变量的值也和预期一样，就是线程安全的，反之则是线程不安全的。
- **多线程并发的三个特性：原子性、有序性、可见性**
  - **原子性：**即一个操作或多个操作，要么全部执行，要么就都不执。执行过程中，不能被打断
  - **有序性：**程序代码按照先后顺序执行
    - **为什么会出现无序问题呢？**因为**指令重排**
  - **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

        - **为什么出现不可见性问题呢？**可以说是因为Java内存模型【JMM】
- **指令重排优化：**重排序是编译器和处理器为了提高程序运行效率，会对输入代码进行优化的一种手段。它不保证程序中，各个语句执行先后顺序的一致。
- **JMM内存模型：**诞生背景是因为CPU的缓存一致性、指令重排优化
    - Java为了保证并发编程中可以满足原子性、可见性及有序性，诞生出了一个重要的概念，那就是**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。**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。
- **happens-before规则：**在JMM中，使用happens-before规则来约束编译器的优化行为，允许编译期优化，但需要遵守一定的Happens-Before规则。**如果一个操作执行的结果需要对另一个操作可见**，那么这两个操作之间必须要存在happens-before的关系！
    - **程序员需关注的happens-before规则：**
        - 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。
        - 锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。
        - volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。
        - 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

02-线程异步变同步Synchronized

Synchronized解决内存可见性问题

- 线程解锁前：必须把自己本地内存中共享变量的最新值刷新到主内存中
- 线程加锁时：将清空本地内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值

Synchronized同步实现原理：同步操作主要是monitorenter和monitorexit两个指令实现，背后原理是Monitor（管程）

管程(Monitor)：是**管**理共享变量及对共享变量操作的过**程**，**将共享变量和对共享变量的操作统一封装起来**，让这个过程可以并发执行。

为什么所有对象都可以作为锁？因为每个对象都都有一个Monitor对象与之关联。然后线程对monitor执行lock和unlock操作，相当于对对象执行上锁和解锁操作。

锁优化：同步锁一共有四个状态：无锁，偏向锁，轻量级锁，重量级锁，JVM会视情况来逐渐升级锁，而不是上来就加重量级锁

- **偏向锁：只有一个线程访问锁资源**，偏向锁就会把整个同步措施消除
- **轻量级锁：只有两个线程交替竞争锁资源**，如果线程竞争锁失败了不立即挂起，而是让它飞一会（自旋），在等待过程中可能锁就会被释放出来，这时尝试重新获取锁

03-volatile关键字

volatile可以保证多线程场景下共享变量的可见性、有序性。

- **可见性：**保证对此共享变量的修改，所有线程的可见性
- **有序性：**禁止指令重排序的优化，遵循JMM的happens-before规则

内存屏障（Memory Barrier）是CPU的一种指令，用于控制特定条件下的重排序和内存可见性问题。Java编译器会根据内存屏障的规则禁止重排序。

volatile缺陷：原子性问题

适合场景：变量真正独立于其他变量和自己以前的值，在单独使用的时适合用volatile

- 对变量的写入操作不依赖其当前值：例如++和--运算符的场景则不行
- 该变量没有包含在具有其他变量的不变式中