

06-并发编程

刘亚雄

极客时间-Java 讲师



今日目标

1. 掌握多线程安全问题原因
2. 理解线程的生命周期和状态转换
3. 掌握Java内存模型JMM及多线程并发的三个特性
4. 掌握synchronized实现原理
5. 掌握volatile关键字实现原理

一、多线程

并发编程灵魂三问

01-什么是并发编程？

- 并发编程：编写多线程（MultiThreading）代码，解决多线程带来的线程安全问题
- 多线程是指从软件和硬件上实现多个线程并发执行的技术，多线程可以使计算机同一时间执行多个线程

02-为什么要学习？感觉上工作中很少用到

- 开启应用高性能的一把钥匙，应用程序的翅膀
- 面试高频考点
- 中间件几乎都是多线程应用：MySQL、ES、Redis、Tomcat、Druid、HikariCP等等

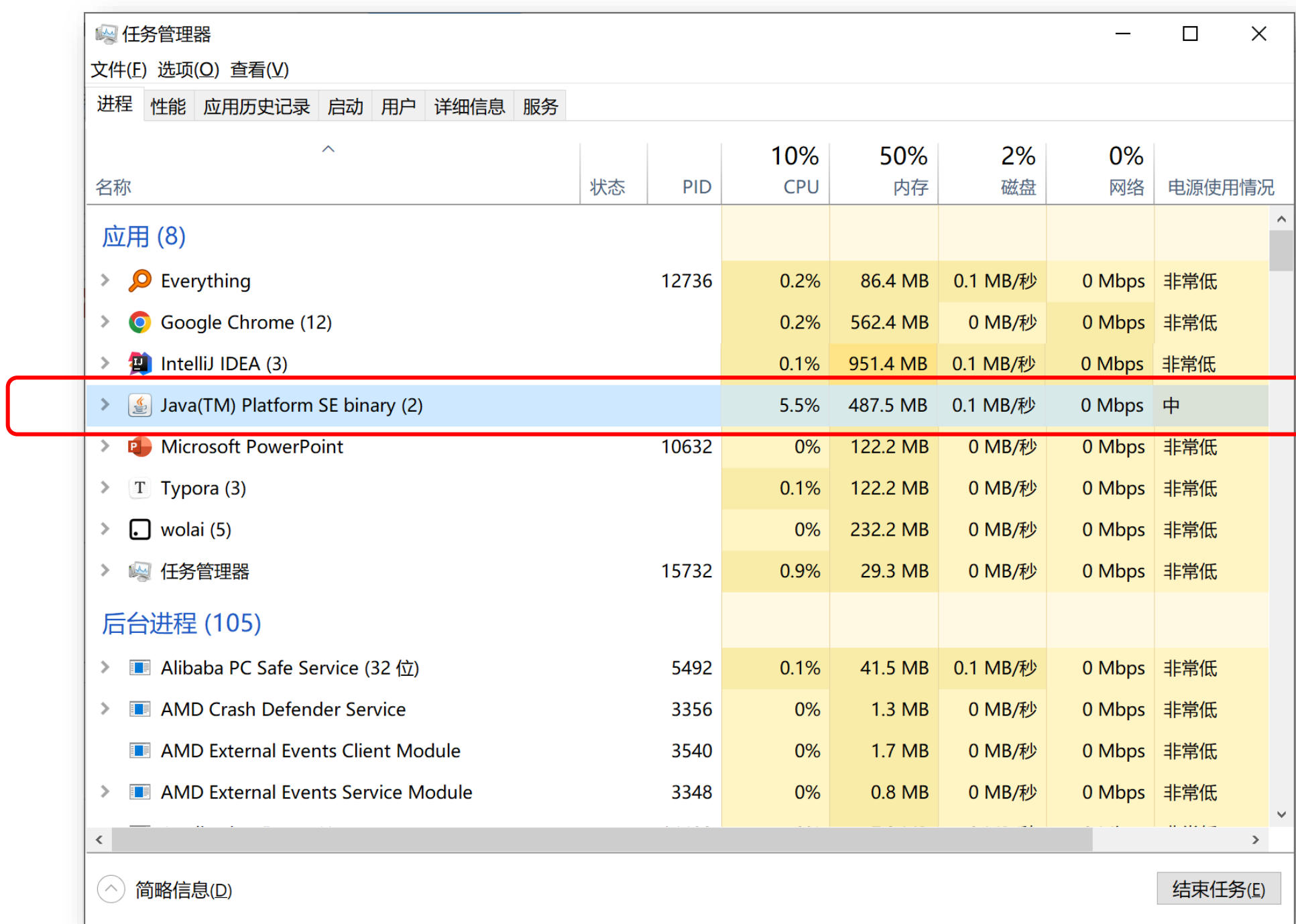
03-怎么学习并发编程？

- **多线程基础核心 → Synchronized实现原理 → Volatile实现原理 → JMM和指令重排**
- **JUC → 原子类与CAS实现原理 → 锁与AQS实现原理 → 并发工具类 → 并发容器 → 阻塞队列 → 线程池**

1.1 多线程重要概念

01-线程和进程

- **进程**：是指内存运行的一个应用程序，是**系统运行程序的基本单位**，是程序的一次执行过程
- **线程**：是**进程中的一个执行单元**，负责当前进程中的任务的执行，一个进程会产生很多线程
- **两者主要区别**：每个**进程都有独立内存空间**。线程之间的**堆空间和方法区共享**，**线程栈空间和程序计数器是独立的**。线程消耗资源比进程小的多



任务管理器							
文件(F) 选项(O) 查看(V)							
进程 性能 应用历史记录 启动 用户 详细信息 服务							
名称	状态	PID	10% CPU	50% 内存	2% 磁盘	0% 网络	电源使用情况
应用 (8)							
Everything		12736	0.2%	86.4 MB	0.1 MB/秒	0 Mbps	非常低
Google Chrome (12)			0.2%	562.4 MB	0 MB/秒	0 Mbps	非常低
IntelliJ IDEA (3)			0.1%	951.4 MB	0.1 MB/秒	0 Mbps	非常低
Java(TM) Platform SE binary (2)			5.5%	487.5 MB	0.1 MB/秒	0 Mbps	中
Microsoft PowerPoint		10632	0%	122.2 MB	0 MB/秒	0 Mbps	非常低
Typora (3)			0.1%	122.2 MB	0 MB/秒	0 Mbps	非常低
wolai (5)			0%	232.2 MB	0 MB/秒	0 Mbps	非常低
任务管理器		15732	0.9%	29.3 MB	0 MB/秒	0 Mbps	非常低
后台进程 (105)							
Alibaba PC Safe Service (32 位)		5492	0.1%	41.5 MB	0.1 MB/秒	0 Mbps	非常低
AMD Crash Defender Service		3356	0%	1.3 MB	0 MB/秒	0 Mbps	非常低
AMD External Events Client Module		3540	0%	1.7 MB	0 MB/秒	0 Mbps	非常低
AMD External Events Service Module		3348	0%	0.8 MB	0 MB/秒	0 Mbps	非常低

进程

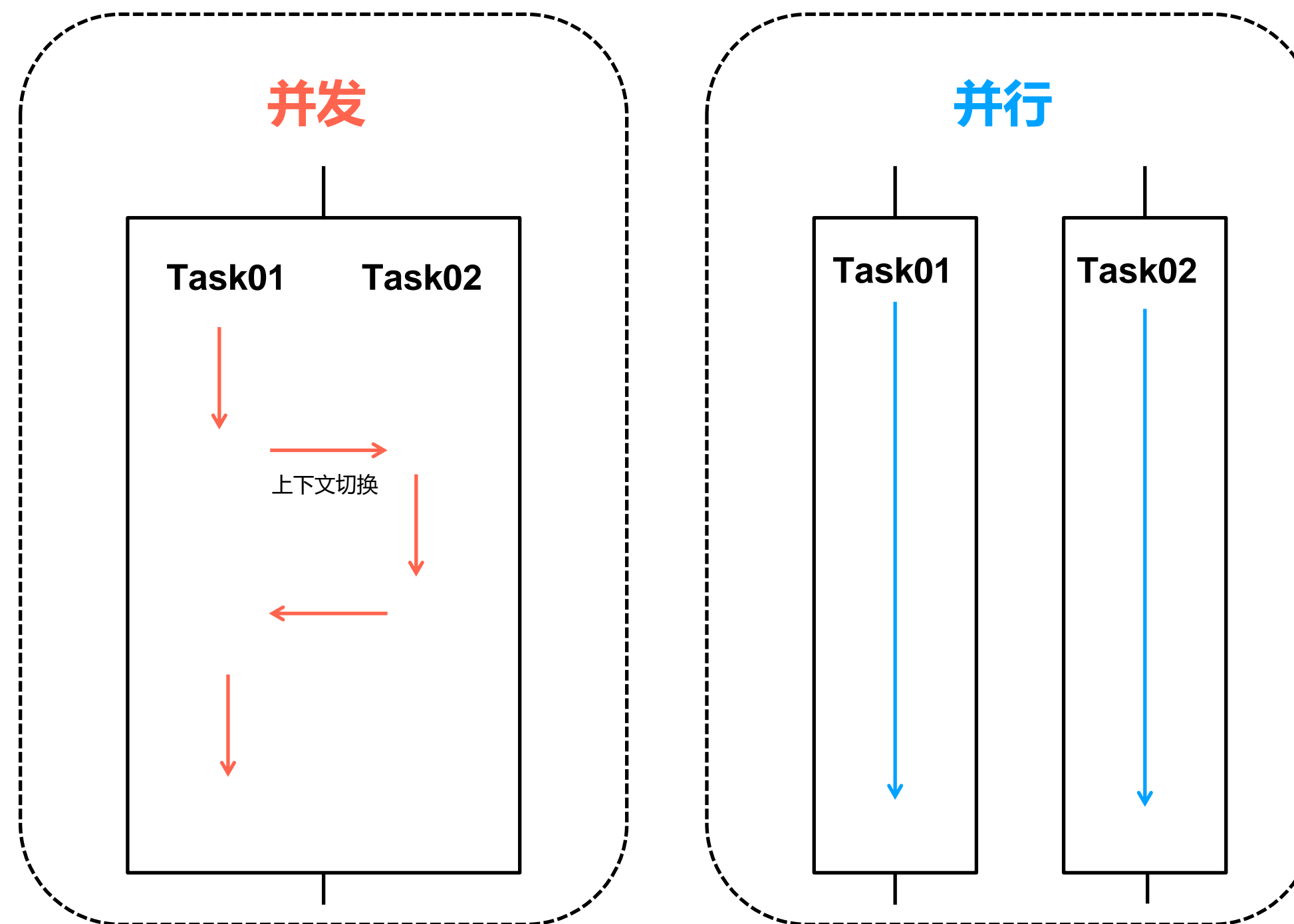


进程里面有多线程

1.1 多线程重要概念

02-并发和并行

- **并发Concurrency**: 同一时间段，多个任务都在执行，单位时间内不一定是同时执行
- **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**



为什么说并发是一种能力？

1.1 多线程重要概念

03-线程上下文切换

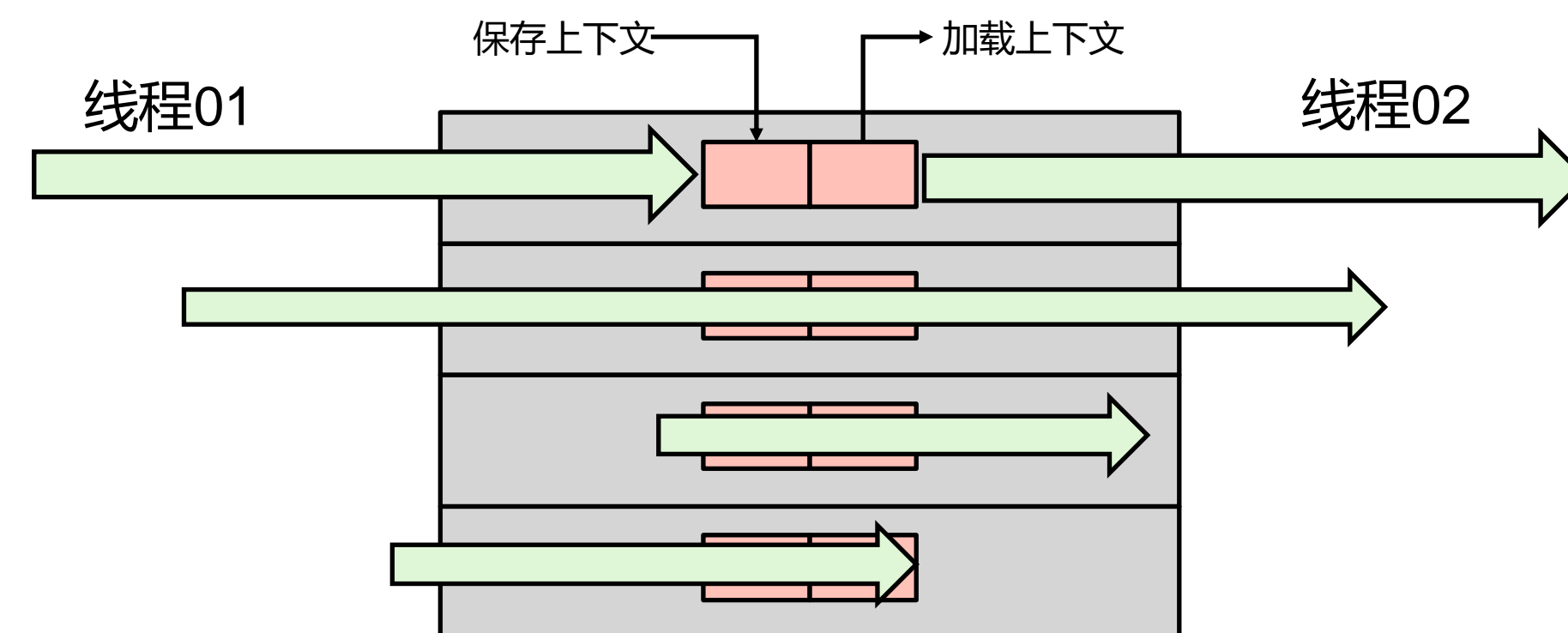
一个CPU同一时刻只能被一个线程使用，为了提升效率CPU采用**时间片算法**将CPU时间片轮流分配给多个线程。在分配的时间片内线程执行，如果没有执行完毕，则需要挂起然后把CPU让给其他线程。

那么问题来了：线程下次运行时，怎么知道上次运行到哪里呢？

- CPU切换线程，会把当前线程的执行位置记录下来，用于下次执行时找到准确位置
- 线程执行位置的记录与加载过程就叫做**上下文切换**
- **线程执行位置记录在程序计数器**

04-上下文切换过程：

- ① 挂起线程01，将线程在CPU的状态（上下文）存储在内存
- ② 恢复线程02，将内存中的上下文在CPU寄存器中恢复
- ③ 调转到程序计数器所指定的位置，继续执行之后的代码

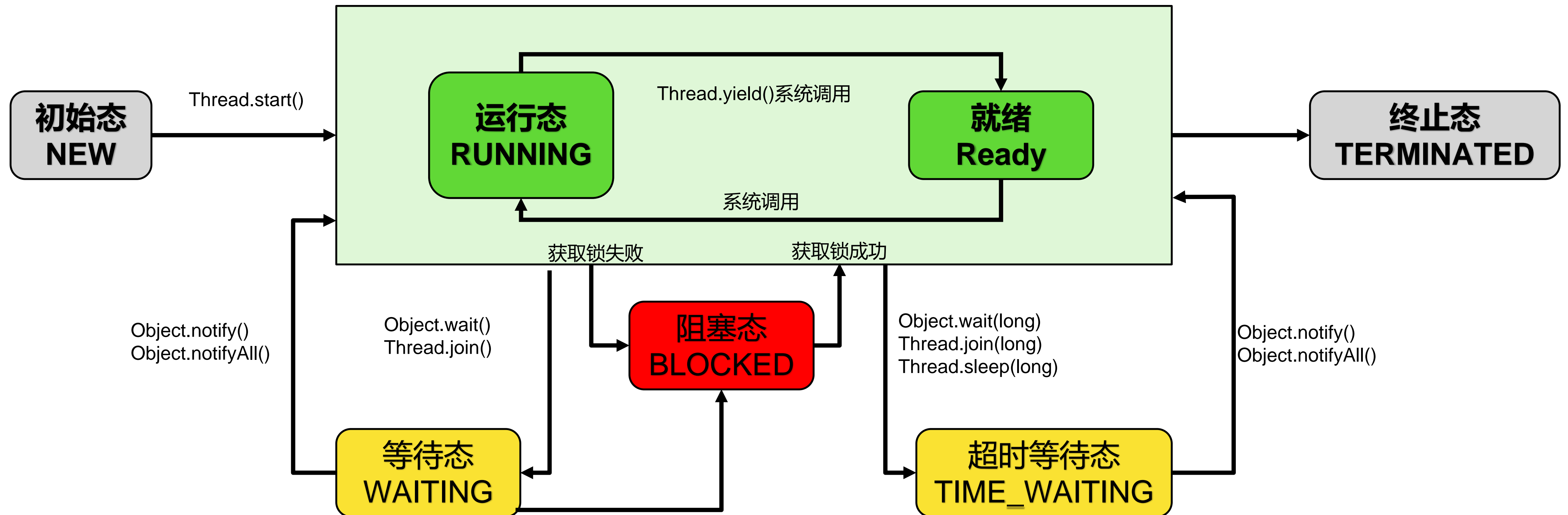


CPU (4核) 时间片50ms

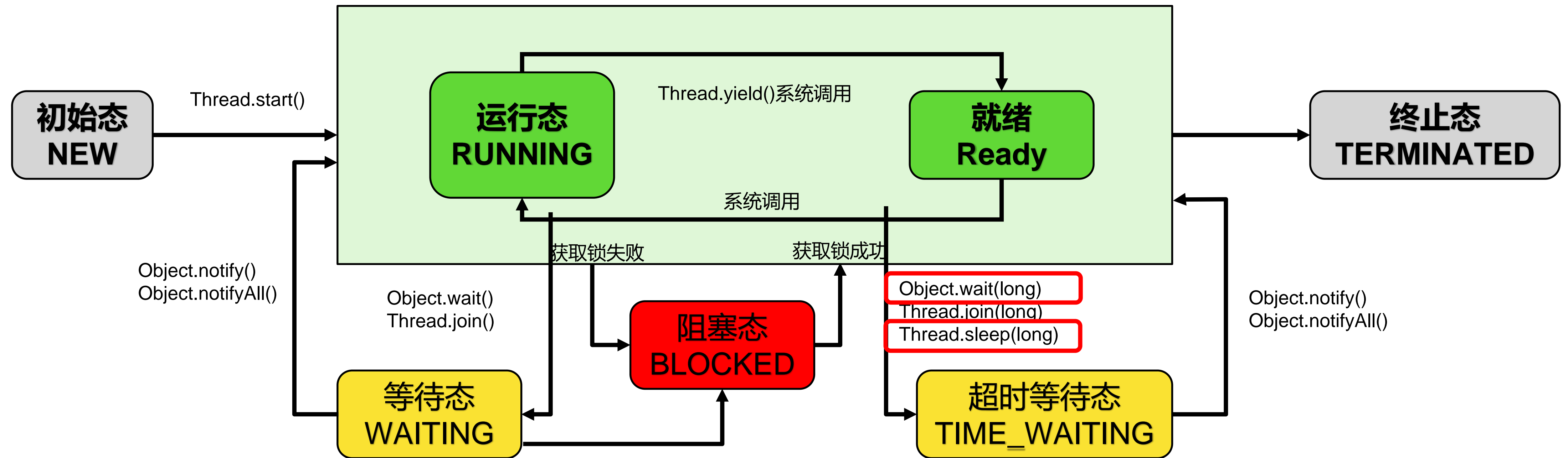
1.2 线程的一生

01-线程从出生到死亡会出现六种状态：

- ①New（新建）、②Runnable（可运行）、③Terminated（终止）
- ④Blocked（锁阻塞）、⑤Waiting（无限等待）、⑥Timed_Waiting（超时等待）



1.2 线程的一生



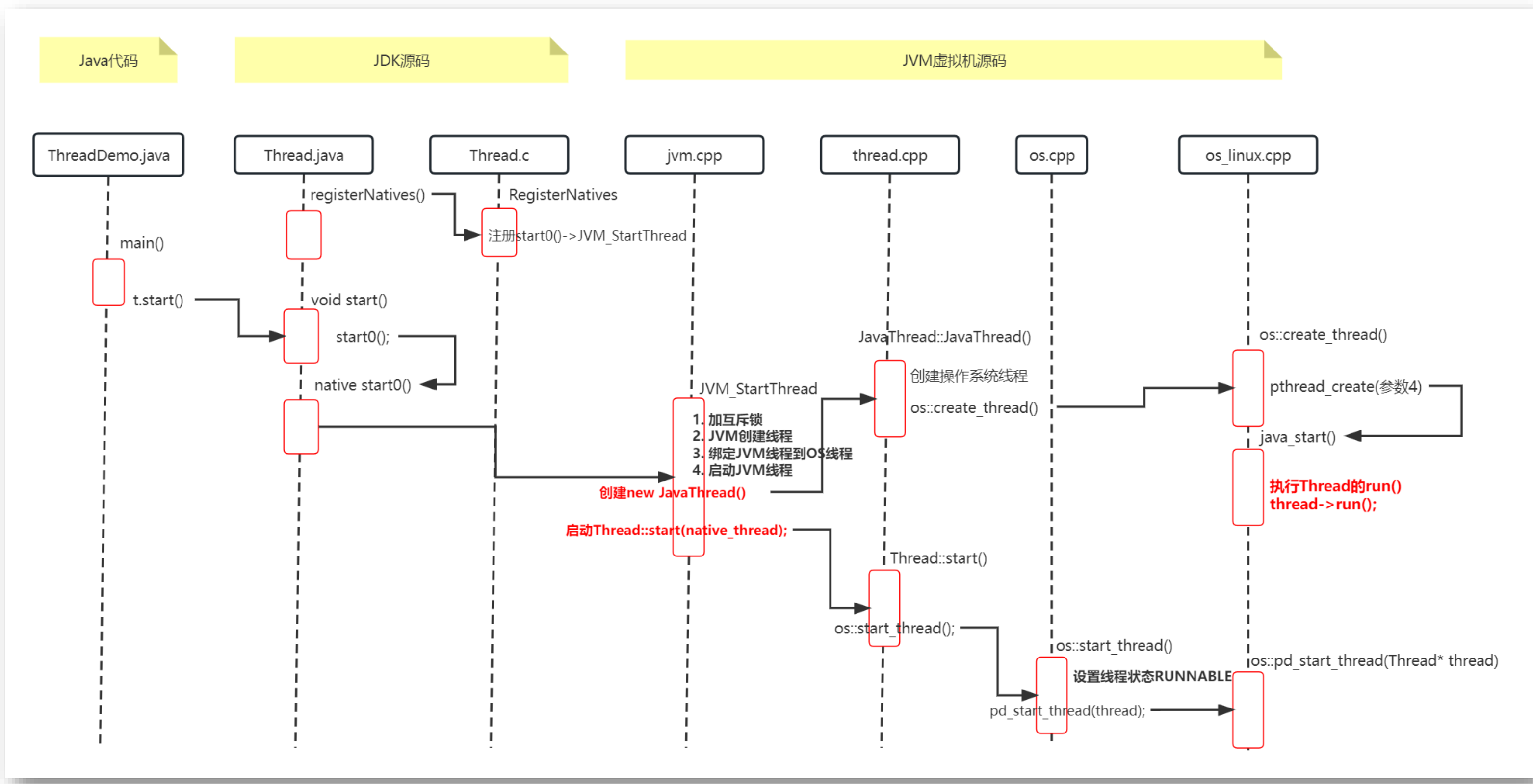
02-wait与sleep()的区别：

- **主要区别：** `sleep()` 方法没有释放锁，`wait()` 方法释放了锁
- 两者都可以暂停线程执行：`wait()` 常用于线程间交互/通信，`sleep()` 用于暂停线程执行
- `wait()` 方法被调用后，需要别的线程调用同一个对象的 `notify` 和 `notifyAll`。超时苏醒使用 `wait(long)` 方法
- `sleep()` 方法执行完成后，线程会自动苏醒。

1.3 多线程源码剖析-图解

Java线程是通过start()方法启动，启动后会执行run()方法
Thread究竟是如何执行run()方法呢？

```
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 }
```



1.3 多线程源码剖析-小结

```
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 }
```

流程小结：

- ① 线程类被JVM加载时会绑定native方法与对应的C++方法
- ② start()方法执行：
 - **start()→native start0()→JVM_Thread→ 创建线程JavaThread::JavaThread**
- ③ 创建OS线程，指定OS线程运行入口：
 - **创建线程构造方法→ 创建OS线程→指定OS线程执行入口，就是线程的run()方法**
- ④ 启动OS线程，运行时会调用指定的运行入口run()方法。至此，实现一个的线程运行
- ⑤ 创建线程的过程是线程安全的，基于操作系统互斥量（MutexLocker）保证互斥，所以说**创建线程性能很差**

1.4 线程安全问题

01-什么是线程安全问题？

- 多个线程同时执行，可能会运行同一行代码，如果程序每次运行结果与单线程执行结果一致，且变量的预期值也一样，就是线程安全的，反之则是线程不安全。



案例：电影院卖票《阿凡达2》

02-引发线程安全问题的根本原因：**多个线程共享变量**

- 如果多个线程对共享变量只有读操作，无写操作，那么此操作是线程安全的
- 如果多个线程同时执行共享变量的写和读操作，则操作不是线程安全的



怎么解决线程安全问题？



1.4 解决线程安全问题

03-为了解决线程安全问题，Java给出了各种办法：

- 同步机制Synchronized
- Volatile关键字：内存屏障
- 原子类：CAS
- 锁：AQS
- 并发容器

案例：牛刀小试



知其然知其所以然！接下来咱们看看多线程的三个核心特点

1.5 线程并发三大特性

01-并发编程最重要的三个特性：必须掌握

- **原子性**：一个系列指令代码，要么全执行，要么都不执行，执行过程不能被打断
- **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，一个线程修改了共享变量的值，其他线程能够立即看到
 - 为什么会出现不可见问题呢？因为Java内存模型（JMM）

1.5 线程并发三大特性-有序性

02-什么是指令重排？

编译器和处理器会对执行指令进行重排序优化，**目的是提高程序运行效率**。现象是，我们编写的Java代码语句的先后顺序，不一定是按照我们写的顺序执行。

举个栗子：

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

上述代码在执行过程中：语句1一定在语句2之前执行吗？不一定，因为有指令重排

- 按顺序执行不好么，**为什么要指令重排执行？** 同步变异步，系统指令层面的优化
- 无论如何重排，不会影响最终执行结果，因为大部分指令并没有严格的前后执行顺序
- 在单线程情况下，程序执行遵循**as-if-serial**语义



啥是个as-if-serial语义？

1.5 线程并发三大特性-有序性

03-什么是as-if-serial语义？

不管编译期和处理器怎么重排指令，单线程执行结果不受影响

举个栗子：

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

上面代码执行的顺序：语句2 ==> 语句1 ==> 语句3 ==> 语句4

不可能是：语句2 ==> 语句1 ==> 语句4 ==> 语句3

为什么？因为处理器在进行指令重排时，会考虑指令之间的数据依赖性

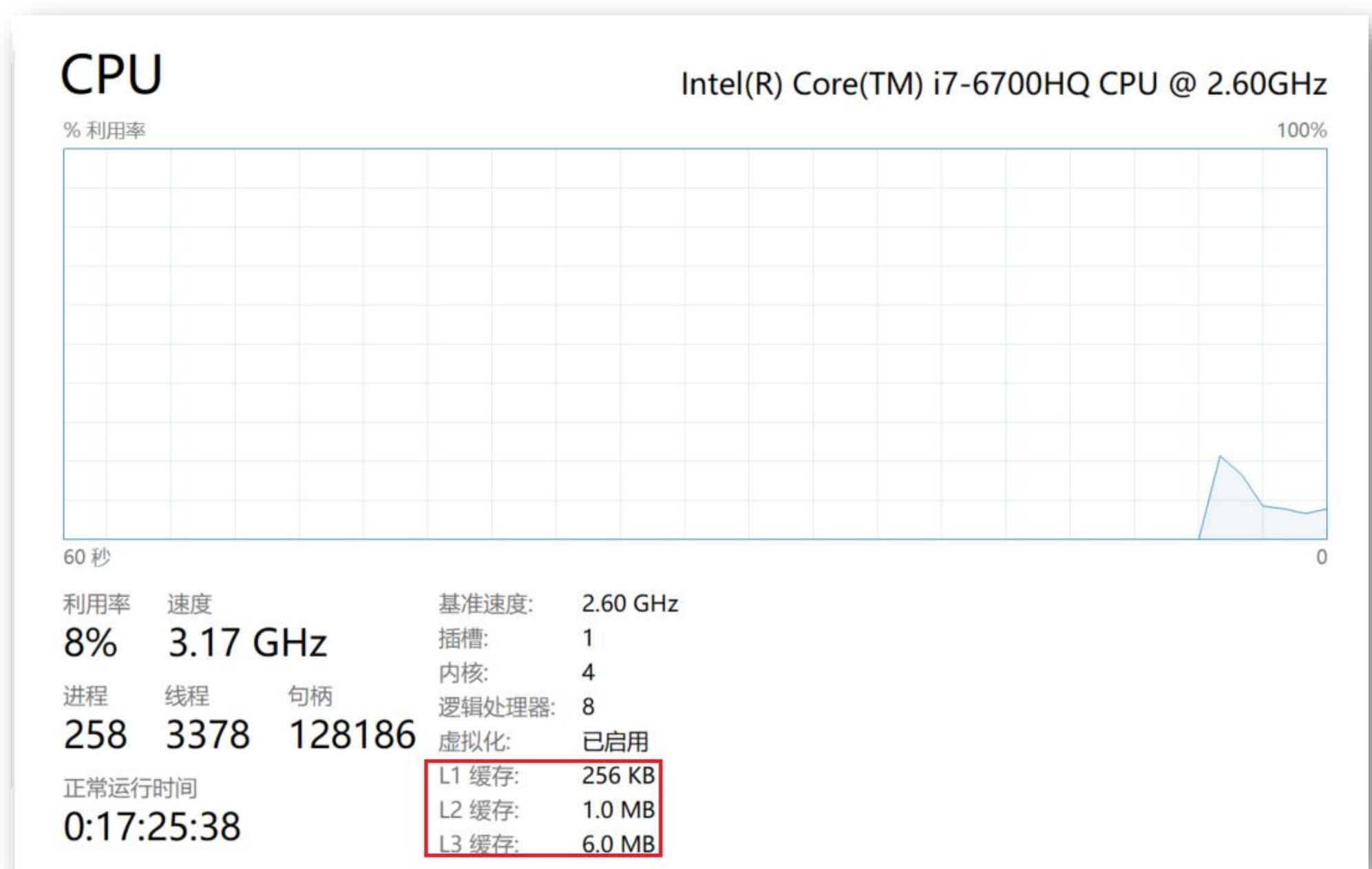
虽然重排序不会影响单线程程序正确执行，但是会影响多线程。我再举个栗子：

```
1 //线程1:
2 init = false
3 context = loadContext(); //语句1
4 init = true; //语句2
5
6 //线程2:
7 while(!init){//如果初始化未完成，等待
8     sleep();
9 }
10 execute(context); //初始化完成，执行逻辑
```


1.5 线程并发三大特性-可见性

01-CPU和缓存一致性

- 在多核 CPU 中每个核都有自己的缓存，同一个数据的缓存与内存可能不一致
- **为什么需要CPU缓存？** 随着CPU技术发展，CPU执行速度和内存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。
- 程序在运行过程中会将运算所需数据**从主内存复制到CPU高速缓存**，当CPU计算直接操作高速缓存数据，**运算结束将结果刷回主内存**。



多线程中，如何保证有序性？及数据的一致性（可见性）？

1.5 线程并发三大特性-可见性

02-Java内存模型 (Java Memory Model)

- Java为了保证满足原子性、可见性及有序性，诞生了一个重要的规范JSR133，Java内存模型简称JMM
- JMM定义了共享内存系统中**多线程应用读写操作行为的规范**
- JMM规范定义的规则，规范了内存的读写操作，从而保证指令执行的正确性
- **JMM规范解决了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规范因此有了Synchronized、Volatile、锁等概念
- JMM的实现屏蔽各种硬件和操作系统的差异，在各种平台下对内存的访问都能保证效果一致

接下来，看一下JMM内存模型**抽象**结构图：

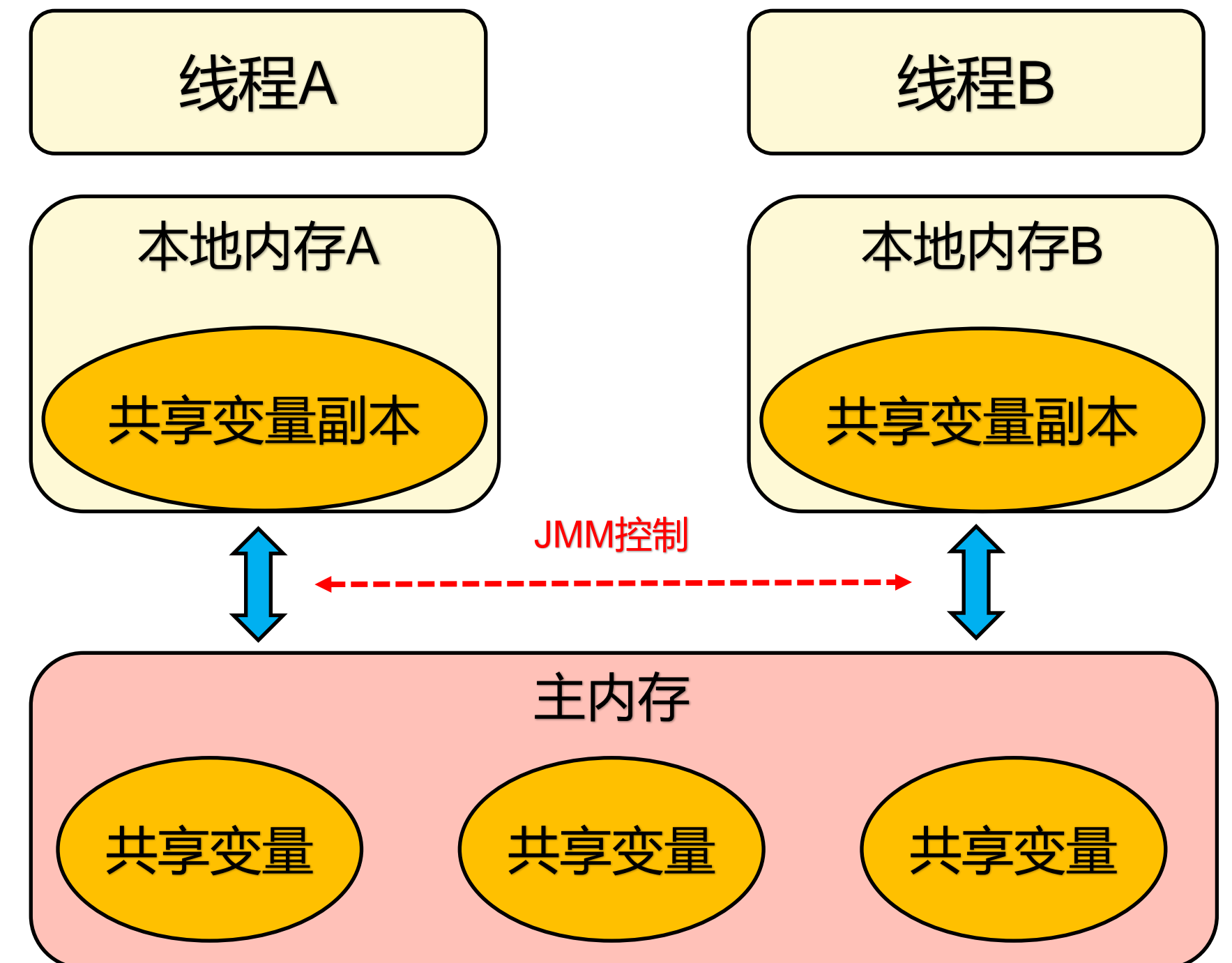
1.5 线程并发三大特性-可见性

03-JMM内存模型抽象结构示意图：

- **JMM定义共享变量何时写入，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
- 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
- **本地内存是抽象的，不真实存在，涵盖：缓存，写缓冲区，寄存器等**

04-JMM线程操作内存基本规则：

- ① 线程操作共享变量必须在本地内存中，不能直接操作主内存的
- ② 线程间无法直接访问对方的共享变量，需经过主内存传递



JMM、CPU缓存一致性与可见性有什么关系？

1.5 线程并发三大特性-可见性

05-什么是内存可见性？

➤ 可见性是一个线程对共享变量的修改，能够及时被其他线程看到

举个栗子：线程A和线程B保证共享变量共享

① 线程A把本地内存A的共享变量副本值更新到主内存

② 线程B到主内存读取最新的共享变量

JMM通过控制线程与本地内存之间的交互，来保证内存可见性



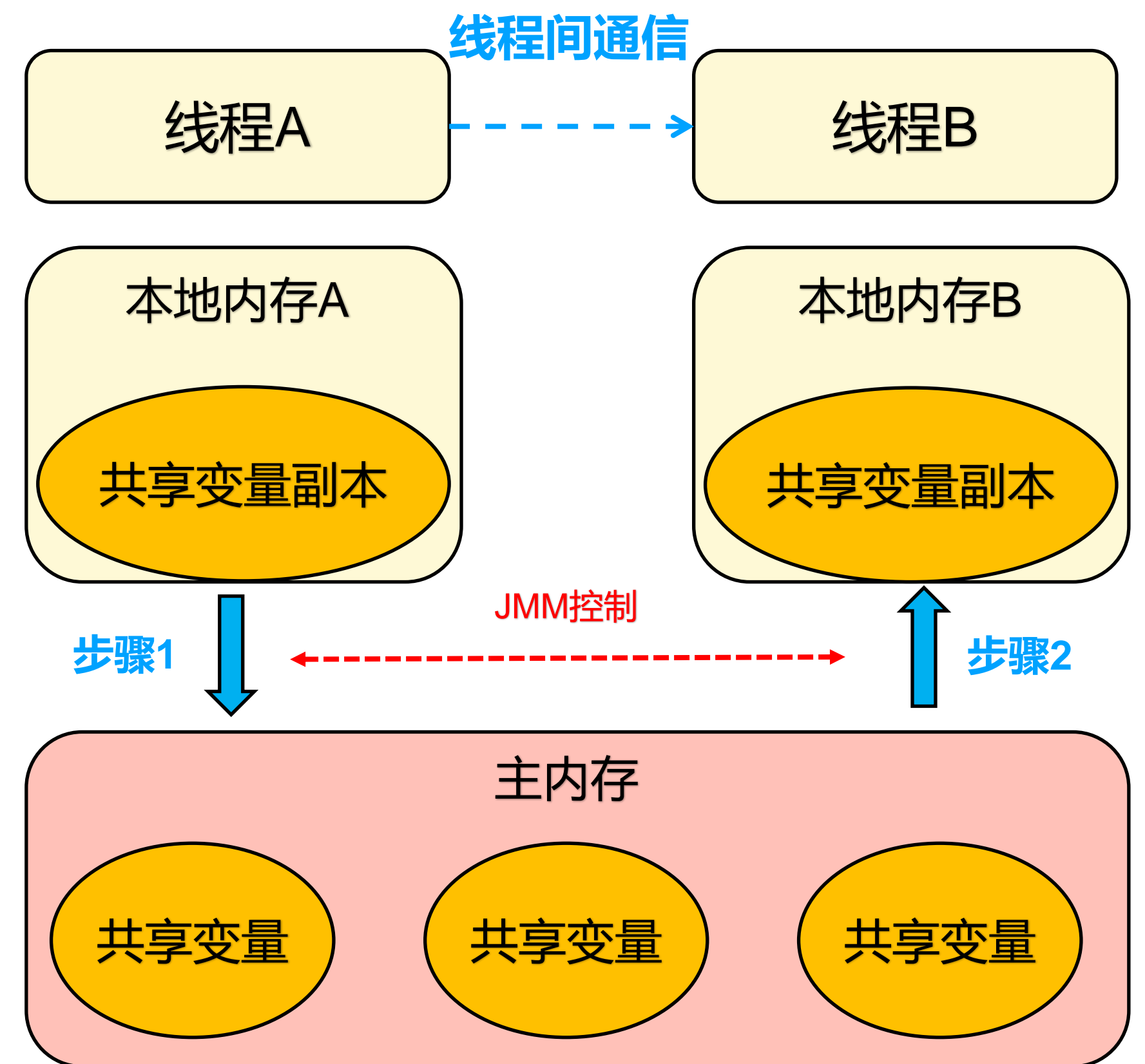
案例：写段代码看下可见性问题

06-怎么解决可见性问题？

➤ 推翻JMM，直接读取主内存共享变量？

➤ 使用JMM：Synchronized，volatile

➤ 都遵循了**happens-before**规则



1.5 线程并发三大特性-可见性

07-happens-before规则

在JMM中使用happens-before规则约束编译器优化行为，Java允许编译器优化，但是不能无条件优化。

如果一个操作的执行结果需要对另一个操作可见，那么这两个操作必须存在happens-before的关系！

程序员需要关注的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与随后对这个锁的加锁
- **Volatile变量规则：**对一个volatile修饰的变量的写，happens-before与任意后续对这个变量的读
- **传递性：**如果A happens-before B，B happens-before C，那么A happens-before C



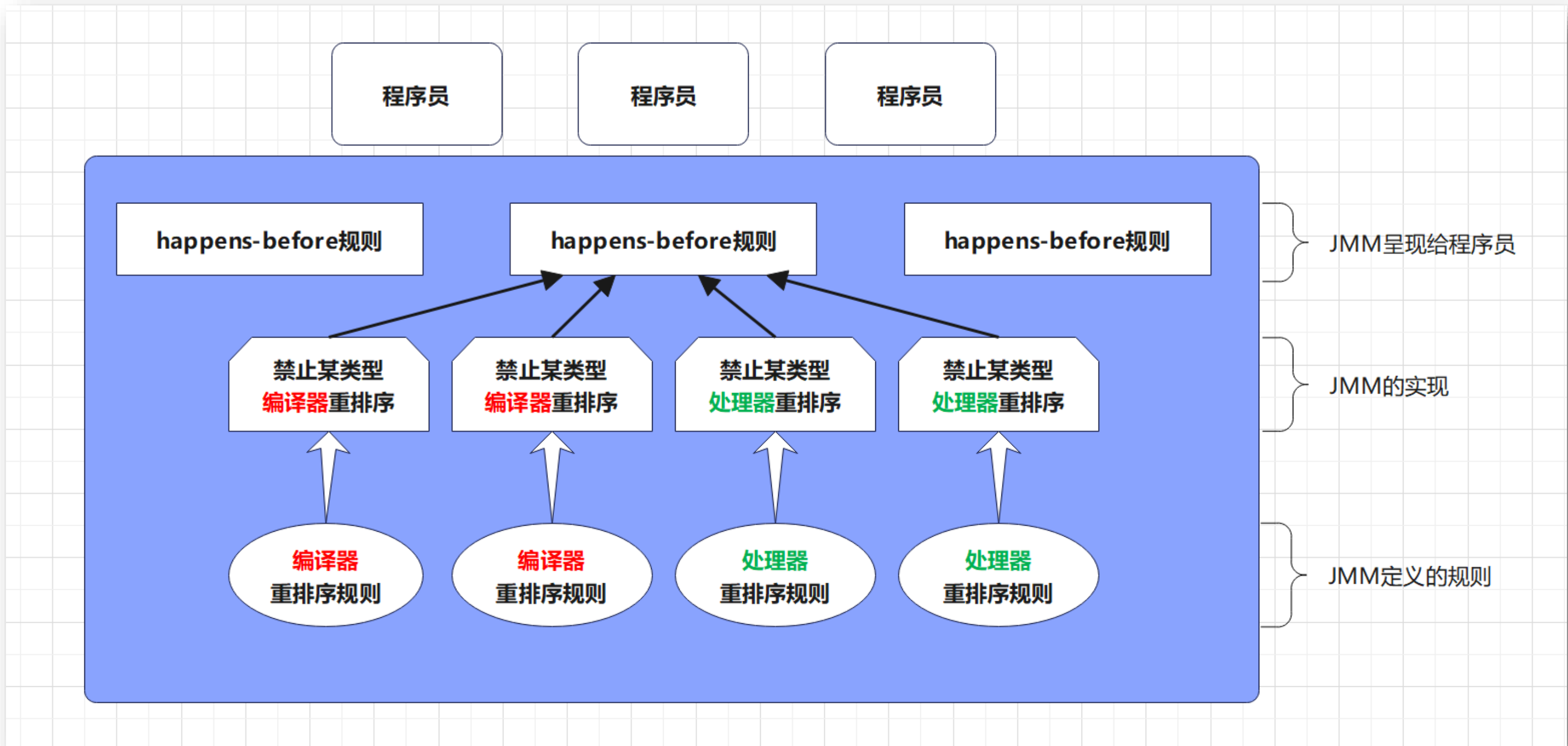
happens-before是怎么实现的呢？

1.5 线程并发三大特性-可见性

08-happens-before规则-实现

- 处理器重排序规则
- 编译器重排序规则

注意：对于程序员，理解上面讲的happens-before规则就可以了。JMM设计happens-before规则的目标就是**屏蔽编译器和处理器重排序规则的复杂性**。不同的CPU架构，不同的OS，不同的虚拟机实现皆不相同！



接下来看一下Synchronized实现原理

THANKS

 极客时间 | 训练营

教育不是注满一桶水，而是点燃一把火