1、观察线程不安全

```
public class ThreadDemo12 {
 static class Counter{
 public int count = 0;
 public void increase(){
 count ++;
 }
 }
 static Counter counter = new Counter();
 public static void main(String[] args) {
  // 此处创建两个线程,分别针对 count 自增5万次
  Thread t1 = new Thread(){
  @Override
  public void run() {
  for(int i = 0;i < 50000; i ++){
  counter.increase();
```

```
    }
    }
    };

    Thread t2 = new Thread(){
    @Override
    public void run() {
    for(int i = 0;i < 50000; i ++){
    counter.increase();
    }
    }
    };
    t1.start();
    t2.start();
    try {
    t1.join();
    t2.join();
    } catch (InterruptedException e) {
    e.printStackTrace();
    }
    System.out.println(counter.count);
    }
}
```

预期能自增10万次，实际上自增的次数，无法确定，每次运行的结果都不一致。目前代码的结果是“不确定的”，因此，我们视这个代码位一个BUG!

为什么会产生这个情况？大概率是和并发执行相关。

由于多线程并发执行，导致代码中出现BUG，这样的情况就称为“线程不安全”。

2、线程安全的概念

如果多线程环境下代码运行的结果是符合我们预期的，即在单线程环境应该的结果，则说这个程序是线程安全的。

3、线程不安全的原因

1. 分析程序执行的过程

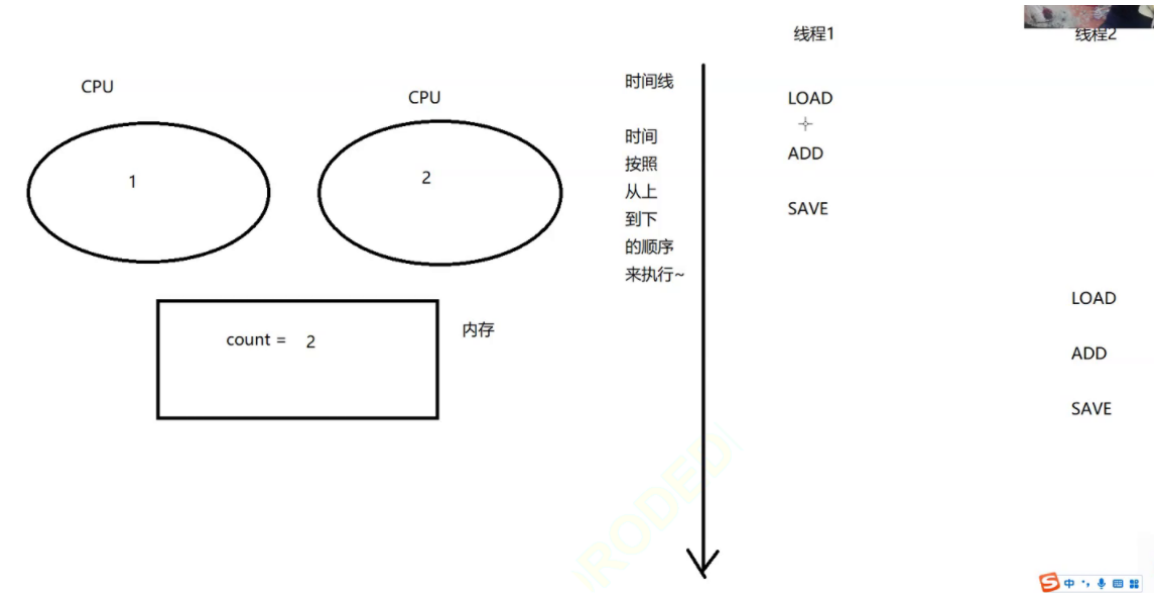
count ++ 的详细过程，分成三个步骤：

1.把内存中的值读取到 CPU 中 （LOAD）

2. 执行 ++ 操作 (ADD)

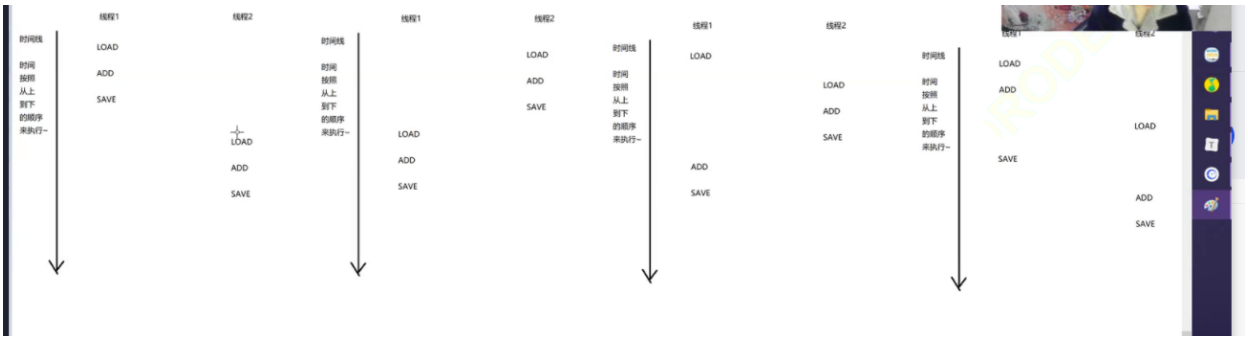
3. 把 CPU 的值写回到内存中 (SAVE)

假设两个线程分别在不同的 CPU 上执行，这样操作出来的结果是符合预期的；如果是两个线程的并发的执行 count ++，就容易出现问题。



如图所示，两个线程真的是按照图中的顺序执行的吗？不确定！操作系统调度线程的时候，采用的是”抢占式执行‘的方式。

某个线程什么时候能上 CPU 执行，什么时候会切换出 CPU 是完全不确定的，而且另一方面，两个线程在两个不同的 CPU 也可以完全并行执行，因此，两个线程执行的具体执行的顺序，是完全不可预期的。



如果线程1的 SAVE 在线程2 的 LOAD 之后，那么此时就会出现当前的这个线程不安全的情况。或者说，必须是“串行执行”的时候，才不会有这样的问题。

在这个代码中，两个线程并发的自增了5w次，在这5w次中，有多少次触发了类似于上面的“线程不安全的问题”，这是不确定的，自增结果是多少，也是

不确定的，但是最终的结果肯定是在5w~10w之间

极端情况下：

如果每次自增都触发了线程安全问题（并列执行），结果就是5w

如果每次自增没有触发线程安全问题（串行执行），结果就是10w

2. 产生线程不安全的原因

1. 线程之间是抢占式执行的（根本原因，不安全的万恶之源）

抢占式执行，导致线程里面的操作顺序无法确定，这样的随机性，是导致线程安全性问题的根本所在

2. 多个线程上修改同一变量

1. 一个线程修改同一个变量，没有线程安全问题！不涉及并发，结果是确定的

2. 多个线程读取同一变量，也没有线程安全问题！读只是单纯把数据从内存放到了 CPU 中，不涉及到修改，内存中的数据始终不变

3.多个线程修改不同的变量，也没有线程安全问题！类似于第1条。

所以为了规避线程安全问题，就可以尝试变换代码的组织形式，达到一个线程只改一个变量

3. 原子性

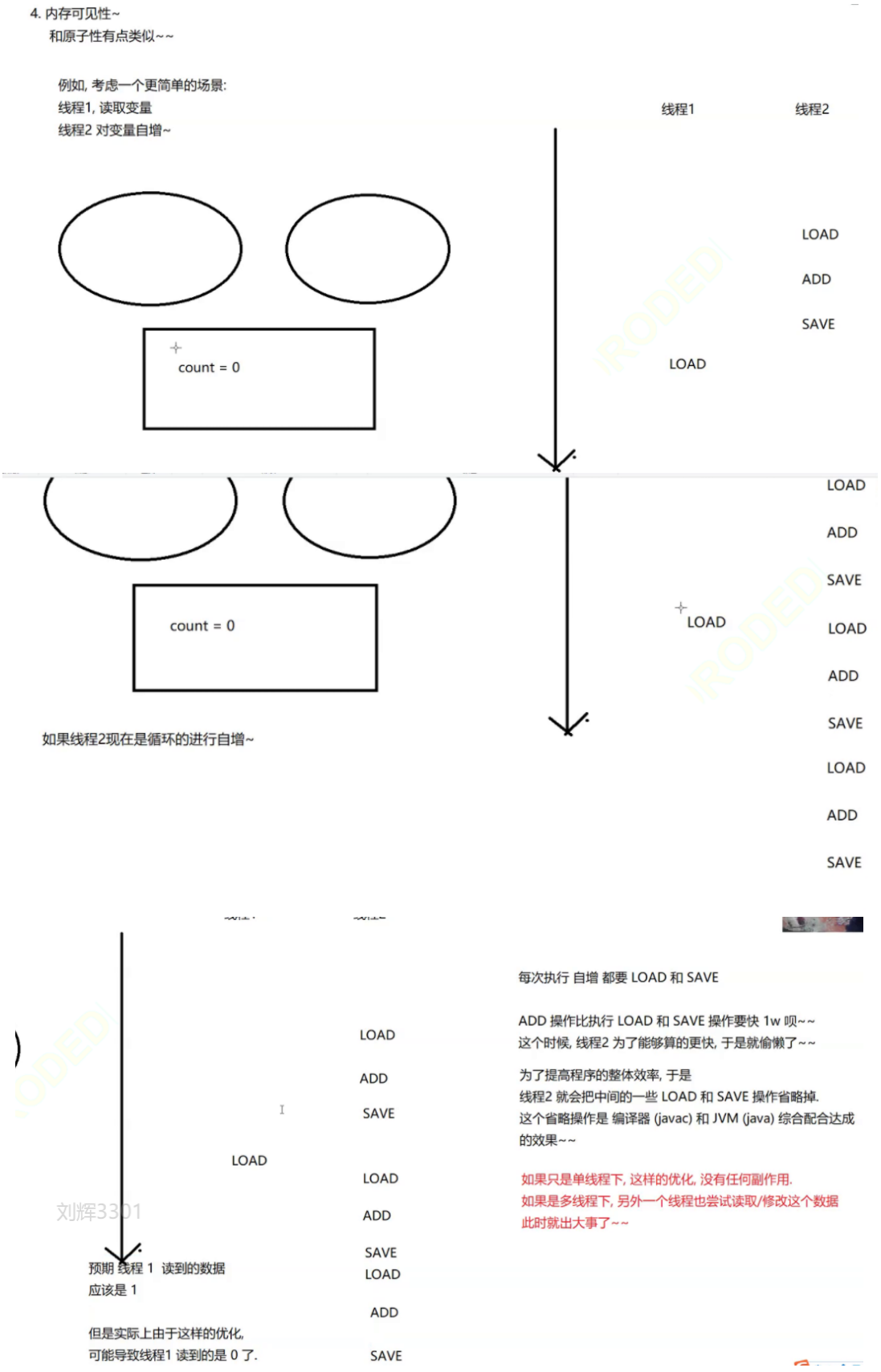
像“++” 这样的操作，本质上三个步骤，是一个“非原子’ 的操作。

像” = “ 操作，本质上是一个步骤，认为是一个” 原子 “的操作

像当前，” ++ “ 操作本身不是原子的，可以通过” 加锁 “ 的方式，把这个操作变成原子的

4. 内存可见性

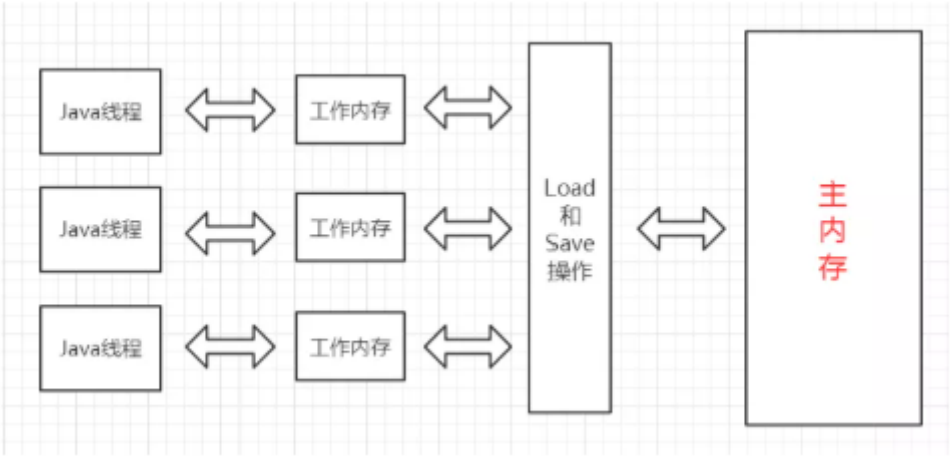
可见性指, 一个线程对共享变量值的修改，能够及时地被其他线程看到.



volatile 关键字，就是用来解决这个问题的

Java 内存模型 (JMM): Java虚拟机规范中定义了Java内存模型.

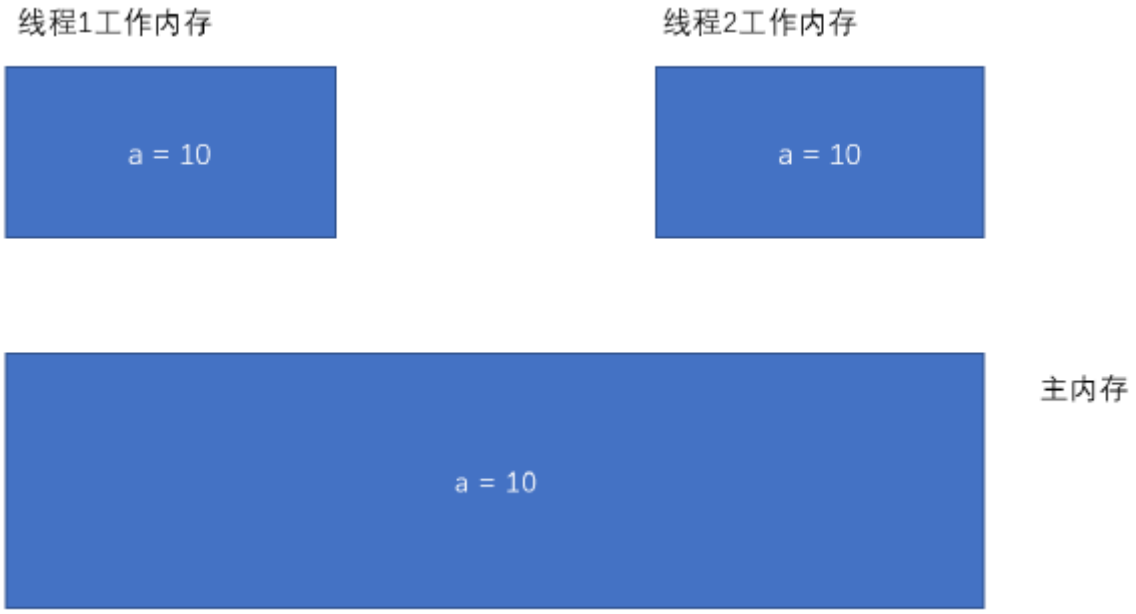
目的是屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的并发效果.



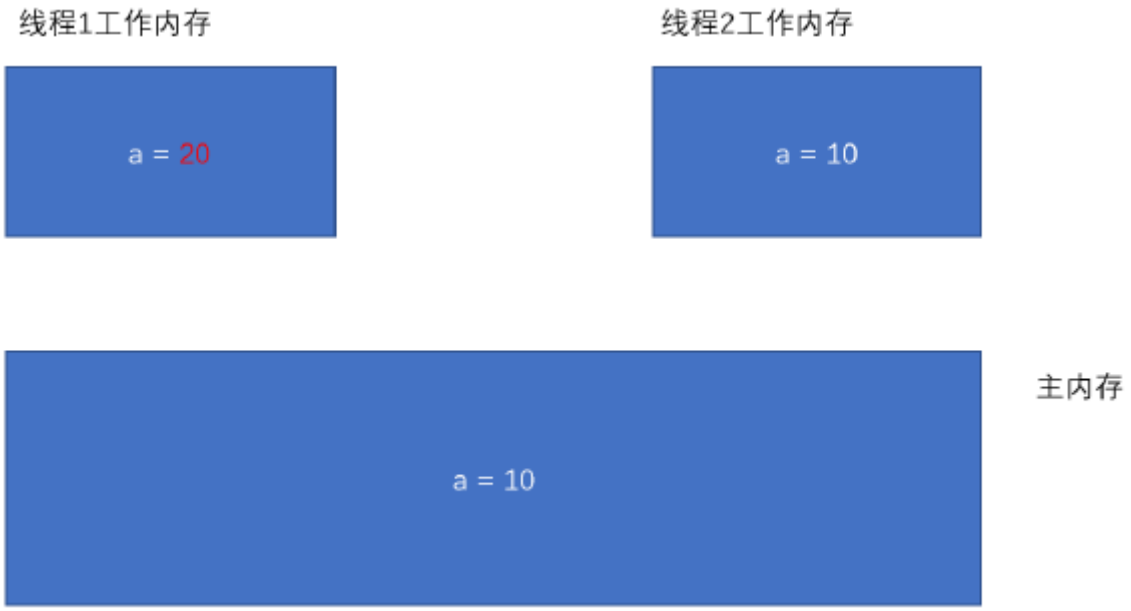
- 线程之间的共享变量存在 主内存 (Main Memory).
- 每一个线程都有自己的 "工作内存" (Working Memory) .
- 当线程要读取一个共享变量的时候, 会先把变量从主内存拷贝到工作内存, 再从工作内存读取数据.
- 当线程要修改一个共享变量的时候, 也会先修改工作内存中的副本, 再同步回主内存.

由于每个线程有自己的工作内存, 这些工作内存中的内容相当于同一个共享变量的 "副本". 此时修改线程1 的工作内存中的值, 线程2 的工作内存不一定会及时变化.

1) 初始情况下, 两个线程的工作内存内容一致.



2) 一旦线程1 修改了 a 的值, 此时主内存不一定能及时同步. 对应的线程2 的工作内存的 a 的值也不一定能及时同步.



这个时候代码中就容易出现问题.

5. 指令重排序

4、解决之前的线程不安全问题

```
public class ThreadDemo12 {
 static class Counter{
 public int count = 0;

// synchronized public void increase(){
// count ++;
// }

 public void increase(){
  synchronized (this){
  count ++;
  }
  }
  }
  static Counter counter = new Counter();
  public static void main(String[] args) {
  // 此处创建两个线程,分别针对 count 自增5万次
  Thread t1 = new Thread(){
  @Override
  public void run() {
```

```
21  for(int i = 0;i < 50000; i ++){
22  counter.increase();
23  }
24  }
25  };
26
27  Thread t2 = new Thread(){
28  @Override
29  public void run() {
30  for(int i = 0;i < 50000; i ++){
31  counter.increase();
32  }
33  }
34  };
35  t1.start();
36  t2.start();
37
38  try {
39  t1.join();
40  t2.join();
41  } catch (InterruptedException e) {
42  e.printStackTrace();
43  }
44  System.out.println(counter.count);
45  }
46 }
```