

第六讲

面向安全的多线程设计

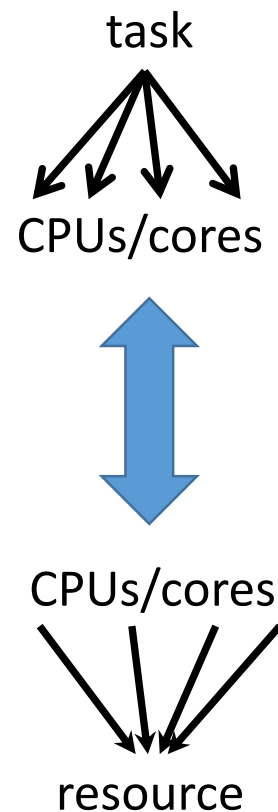
OO课程组

内容摘要

- 理解并发机制
- 多线程程序常见的问题
- 线程安全问题
- 线程安全设计
- 作业

并行与并发的概念差异

- **并行(parallel):** 使用多个处理器资源来同时进行计算，从而提高完成任务的效率
 - 例：google的Map Reduce就是使用多个分布的计算机来对搜索数据进行并行处理
 - 例：使用多个线程来对给定数据进行按段排序
- **并发(concurrency):** 多个执行流(即线程)需要共享访问某些公共资源(如数据)
 - 例：云计算系统中，多个用户(对应为线程)同时要访问云端资源，如数据文件
 - 例：论坛系统中，多用户同时发布帖子、回复帖子，帖子成为共享资源
 - 例：多个线程同时要访问共享的数据

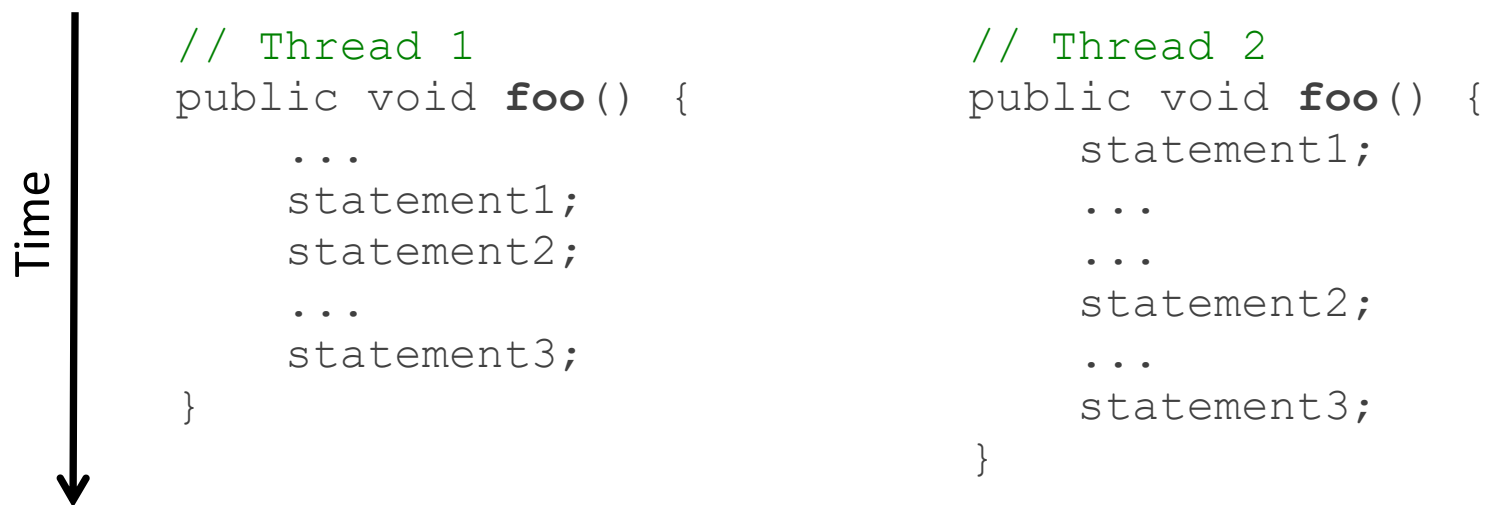


程序为什么需要并发机制

- 和并行机制不同，引入并发的目标并不总是获得更快的计算效率
 - 即便是单CPU、单核的平台也需要并发机制
- 并发机制可以
 - 提高程序的响应效率/性能
 - 例：可视化界面程序通过线程并发提高GUI界面的响应效率，从而获得更好的可用性，后台线程进行数据或任务计算
 - 例：网络通讯程序通过线程并发提高程序对网络请求的响应效率
 - 提高处理器的利用率，屏蔽I/O延迟
 - 如果一个线程在进行I/O慢速处理时，其他线程可以获得处理器资源，提高整体利用率
 - 有效隔离故障
 - 一个线程执行过程中出现了故障，不会影响其他线程的执行
 - 网络多用户访问情景

线程执行时序不确定性

- 使用线程并发机制会带来好处，同时也会带来潜在问题，即时序不确定性。如果两个线程同时执行一个方法：
 - 哪个线程先执行该方法是不确定的
 - 一个线程执行了多少指令后，另一个线程得到执行也是不确定的



线程执行时序不确定性

```
public class Producer extends Thread {  
    private Tray tray;          private int id;  
    public Producer(Tray t, int id) {  
        tray = t;          this.id = id;          }  
    public void run() {  
        for (int i = 0; i < 10; i++)  
            for(int j =0; j < 10; j++ ) {  
                tray.put(i, j);  
                System.out.println("Producer #" + this.id  + " put: (" +i +", "+j + ").");  
                try { sleep((int)(Math.random() * 100)); }  
                catch (InterruptedException e) { }  
            };  
    }  
}
```

```
public class ProducerConsumerTest {  
    public static void main(String[] args) {  
        Tray t = new Tray();  
        Producer p1 = new Producer(t, 1);  
        Consumer c1 = new Consumer(t, 1);  
        p1.start();  
        c1.start();  
    }  
}
```

线程执行时序不确定性

```
public class Consumer extends Thread {  
    private Tray tray;  
    private int id;  
    public Consumer(Tray t, int id) {  
        tray = t;    this.id = id;    }  
    public void run() {  
        int value = 0;  
        for (int i = 0; i < 10; i++) {  
            value = tray.get();  
            System.out.println("Consumer #" + this.id +  
" got: " + value);  
        }  
    }  
}
```

```
Consumer #1 got: 0  
Producer #1 put: (0,0).  
Consumer #1 got: 1  
Producer #1 put: (0,1).  
Producer #1 put: (0,2).  
Consumer #1 got: 2  
Consumer #1 got: 3  
Producer #1 put: (0,3).  
Consumer #1 got: 4  
Producer #1 put: (0,4).  
Producer #1 put: (0,5).  
Consumer #1 got: 5  
Producer #1 put: (0,6).  
Consumer #1 got: 6  
Consumer #1 got: 7  
Producer #1 put: (0,7).  
Consumer #1 got: 8  
Producer #1 put: (0,8).  
Producer #1 put: (0,9).  
Consumer #1 got: 9  
Producer #1 put: (1,0).
```

线程执行时序不确定性

```
public class Tray {  
    private int x,y;    private boolean available = false;  
    public synchronized int get() {  
        while (available == false) {  
            try { wait(); } catch (InterruptedException e) { }  
        }  
        available = false; // 此时available为真，确保所有其他消费者等待  
        notifyAll();  
        return x+y; }  
    public synchronized void put(int a, int b) {  
        while (available == true) {  
            try { wait(); } catch (InterruptedException e) { }  
        }  
        x= a; y = b;  
        available = true; // 唤醒等待队列中的其他消费者或生产者  
        notifyAll(); }  
}
```


线程安全对象

- Thread-safe class是关于对象如何被多个线程安全访问，与对象的具体行为无关。
- 一个线程安全的类，在多个线程并发访问该类的共享对象时，这个类行为的正确性不会受到线程调度、执行时间等因素影响，也不需要线程在调用该共享对象的方法时额外进行同步控制。
 - 共享对象才会出现线程安全问题
 - 线程的调度和执行时间具有不确定性
 - 类行为的正确性指什么？
- 不可变对象是否存在线程安全问题？

线程不安全的几个实例

- 无状态类与有状态类在安全上的差异

```
public class CountingFactorizer implements Servlet {  
    private long count = 0;  
    public long getCount() { return count; }  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        ++count;  
        encodeIntoResponse(resp, factors);  
    }  
}
```

无状态对象的任何计算都只能使用局部变量，其引用只能在栈上，不会产生共享。

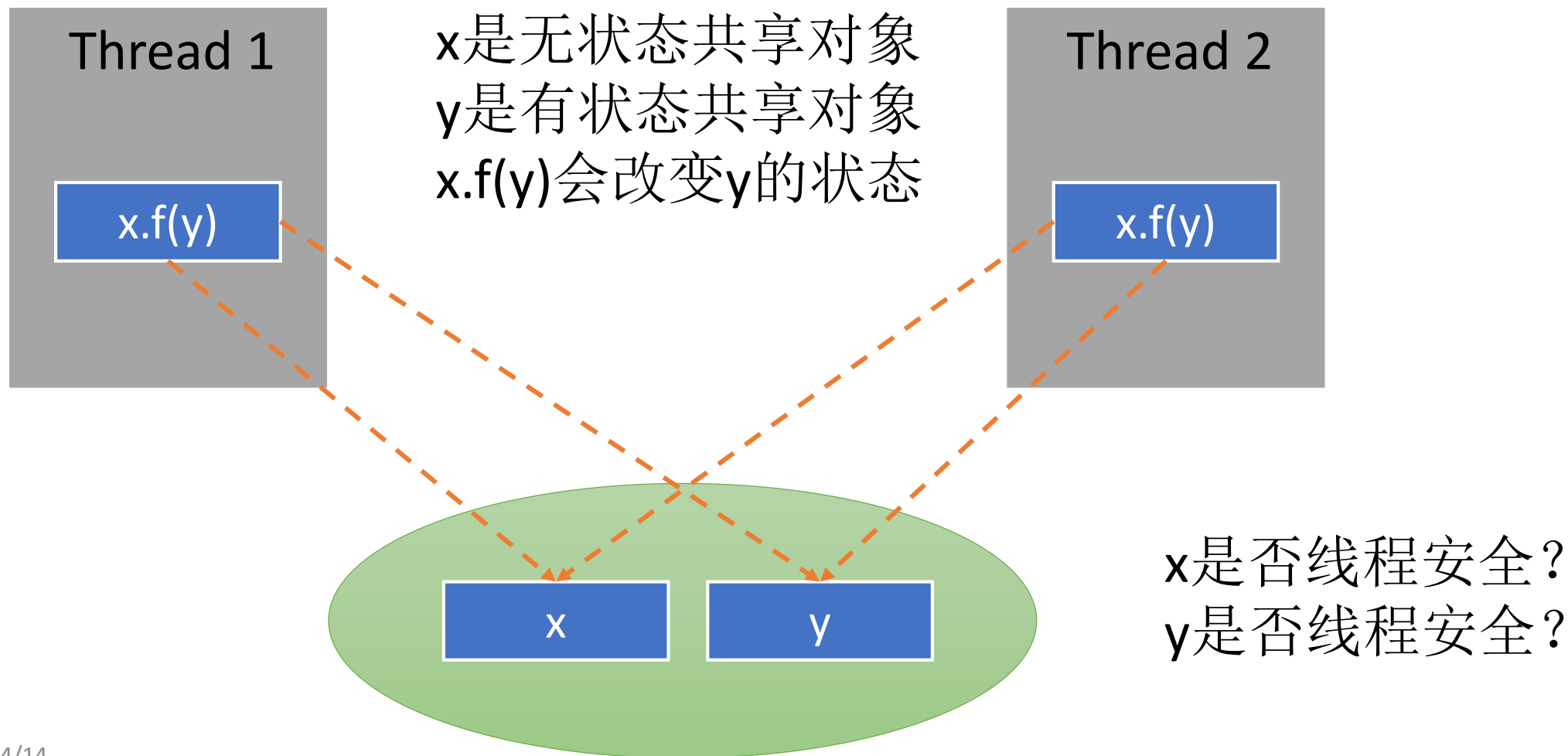
有状态对象的read-modify-write计算会导致出现不安全问题。

Read x: ...=...x...

Update x: x+1

Write to x: x=....

线程不安全的几个实例



线程不安全的几个实例

- 竞争条件(race conditions)

- 当一个计算的正确性依赖于多个线程之间的相对执行次序时，这几个线程就出现了竞争条件。最常出现竞争条件的计算模式是 **check-then-act**, 可能会导致某个线程使用了过时的值进行检查，从而做出错误的动作。

- **check-then-act:**

- **if (b_exp(x)){**
- **do something with x**
- **update x**
- **}**

```
public class LazyInitRace {  
    private ExpensiveObject instance = null;  
    public ExpensiveObject getInstance() {  
        if (instance == null)  
            instance = new ExpensiveObject();  
        return instance;  
    }  
}
```

线程不安全的几个实例

- Read-modify-write和check-then-act是基本的计算模式，大量使用。
 - 计算需要使用变量的当前值
 - 计算会改变变量的当前值
- 这两种计算模式出现线程安全隐患的主要原因是计算过程需要多步完成，无法保证计算的原子性
 - 原子性：计算时不会被中途中断
- 为了满足计算的原子性，需要使用互斥机制
 - Java提供了一系列保证计算原子性的数据类型
 - 锁(lock)

线程不安全的几个实例

- 如果read-modify-write和check-then-act计算只涉及单一变量，就可以通过java.util.concurrent.atomic包中提供的Atomic****类型来确保计算的原子性。

Class	Description
<code>AtomicBoolean</code>	A boolean value that may be updated atomically.
<code>AtomicInteger</code>	An int value that may be updated atomically.
<code>AtomicIntegerArray</code>	An int array in which elements may be updated atomically.
<code>AtomicIntegerFieldUpdater<T></code>	A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.
<code>AtomicLong</code>	A long value that may be updated atomically.
<code>AtomicLongArray</code>	A long array in which elements may be updated atomically.
<code>AtomicLongFieldUpdater<T></code>	A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.
<code>AtomicMarkableReference<V></code>	An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
<code>AtomicReference<V></code>	An object reference that may be updated atomically.
<code>AtomicReferenceArray<E></code>	An array of object references in which elements may be updated atomically.
<code>AtomicReferenceFieldUpdater<T,V></code>	A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.
<code>AtomicStampedReference<V></code>	An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.

线程不安全的几个实例

- 如果read-modify-write和check-then-act计算只涉及单一变量，就可以通过java.util.concurrent.atomic包中提供的Atomic****类型来确保计算的原子性。

```
import java.util.concurrent.atomic;  
public class CountingFactorizer implements Servlet {  
    private final AtomicLong count = new AtomicLong(0);  
    public long getCount() { return count.get(); }  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        count.incrementAndGet();  
        encodeIntoResponse(resp, factors);  
    }  
}
```

线程不安全的几个实例

- 如果一个类里有多多个需要确保计算原子性的属性怎么办？

```
import java.util.concurrent.atomic;  
public class UnsafeCachingFactorizer implements Servlet {  
    private final AtomicReference<BigInteger> lastNumber = new AtomicReference<BigInteger>();  
    private final AtomicReference<BigInteger[]> lastFactors = new AtomicReference<BigInteger[]>();  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        if (i.equals(lastNumber.get()))  
            encodeIntoResponse(resp, lastFactors.get());  
        else {  
            BigInteger[] factors = factor(i);  
            lastNumber.set(i);  
            lastFactors.set(factors);  
            encodeIntoResponse(resp, factors);  
        }  
    }  
}
```

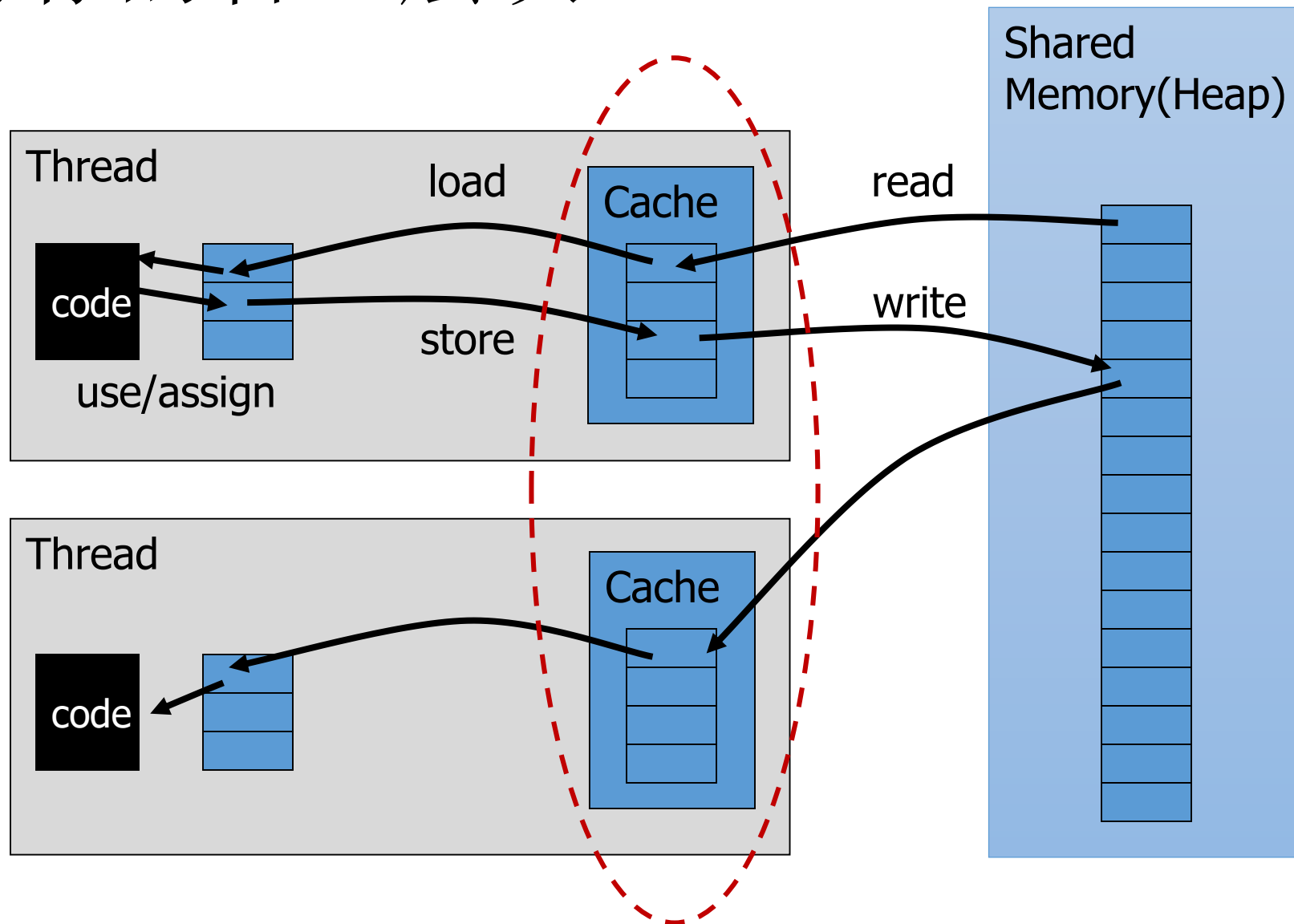
lastNumber和lastFactors之间具有关系：
lastFactors = factor(lastNumber)
==》使用lock来解决！

假设lastNumber和lastFactors之间没有
关系，是否仍然有问题？

Java内存模型

- 管理多个线程对共享内存(对象)的访问方式
 - 多个线程共享对象，因而具有竞争关系
 - 通过同步控制来避免出现不可预料的问题
 - 计算结果被覆盖
 - 读取过时的结果
 - 相互等待—死锁
 - 对象状态被破坏
 - 计算结果不能被其他线程及时获得
 - 等等

Java内存的管理层次



共享对象访问的关键问题

- 读写访问冲突

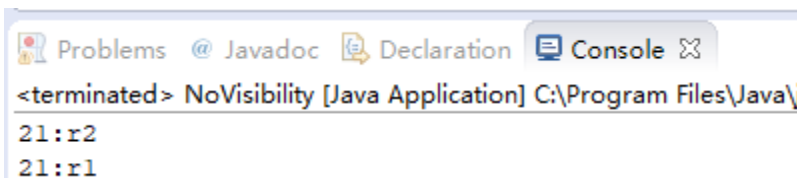
- 问题表现：共享对象的状态更新被覆盖；读取的状态不完整等
- 根本原因：一个线程的“写访问”还未做完，另一个线程进行“写访问”或者“读访问”

- 状态更新延迟

- 没有读写冲突（比如写动作本身就是原子动作），但是读操作无法及时获得状态的更新

```
public class NoVisibility {
    private static boolean ready=false;
    private static int number=0;
    private static class ReaderThread extends Thread
    {
        public ReaderThread(String name){
            super(name);
        }
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number+": "+this.getName());
        }
    }

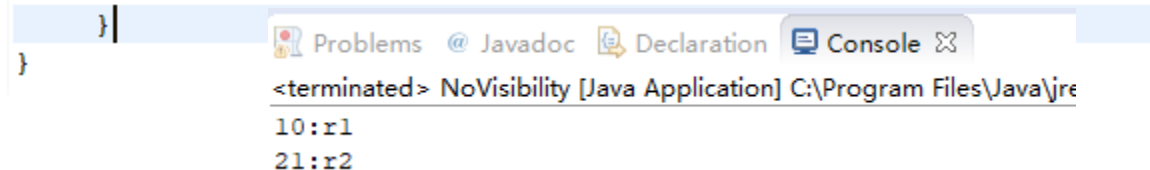
    public static void main(String[] args) {
        ReaderThread r1 = new ReaderThread("r1");
        r1.start();
        ready = true;
        number = 10;
        //try{
        //    r1.sleep(50);
        //}catch (InterruptedException e){};
        ReaderThread r2 = new ReaderThread("r2");
        r2.start();
        number = 21;
    }
}
```



Problems @ Javadoc Declaration Console

<terminated> NoVisibility [Java Application] C:\Program Files\Java\jre

21:r2
21:r1

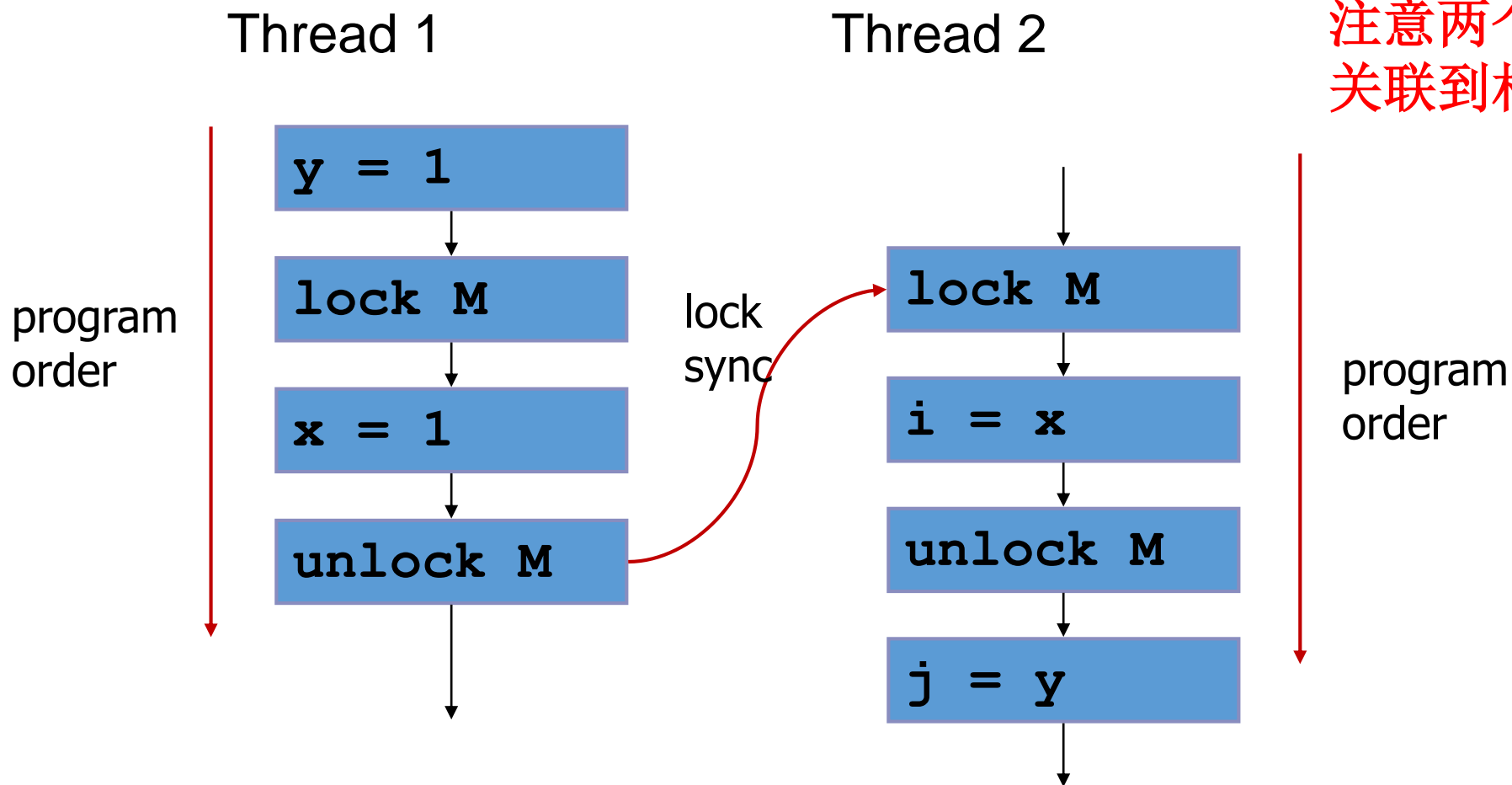


Problems @ Javadoc Declaration Console

<terminated> NoVisibility [Java Application] C:\Program Files\Java\jre

10:r1
21:r2

通过锁解决共享对象访问问题

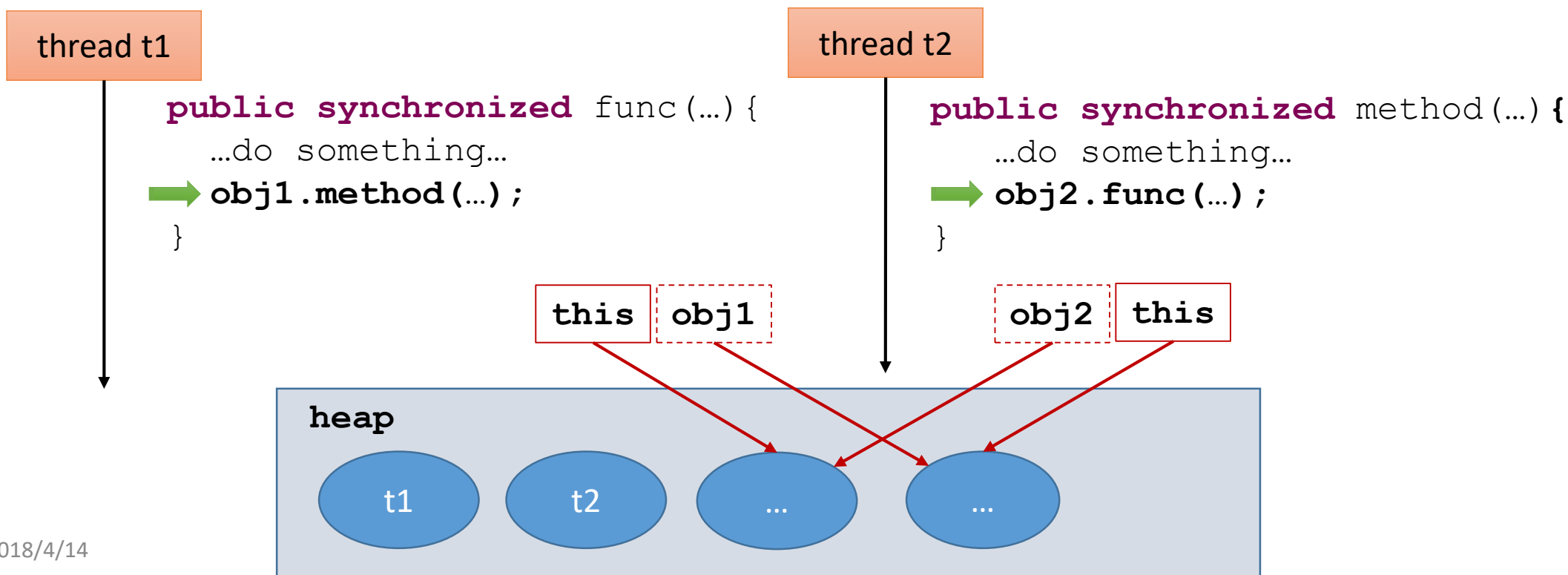


注意两个锁**M**必须
关联到相同的对象

unlock M之前的x取值在lock M之后能直接获得

死锁问题

- 两个线程互相等待对方持有的锁来往下执行，导致相互锁住
 - 不一定必然导致程序不响应，但是会导致某些功能不响应



共享对象的线程安全问题

- 共享是产生安全问题的根本
- 程序中有多重操作会产生对象共享
 - 参数传递
 - 返回值
 - 对象引用赋值
- 在分析一个类是否存在线程安全问题时，需要分析它会在哪些线程中被使用和共享
- 把一个私有对象向外共享的过程也称为对象发布

对象发布与状态失控

- 对象发布
 - 把对象“丢进”外部直接可访问的集合中
 - 把对象传递入另外一个类的方法中
 - 通过方法返回值把对象返回到外部
- 在单线程程序中，对象发布不会影响执行结果
- 多线程程序必须跟踪发布出去的每个对象
 - 确保对其访问的线程之间的执行顺序得到控制
 - 否则出现状态失控

```
class UnsafePublication {  
    private String[] states = new String[] {"AK", "AL" ...};  
    public String[] getStates() { return states; }  
}
```

对象发布与状态失控

- 我们必须做对象引用分析

- 数据流分析的一种
- 得到对象传递引用链：方法代码区块
- 处于同一条传递引用链上的代码区块如果跨越多个线程，则都需要进行同步控制，否则就会产生共享对象状态失控--》线程不安全

```
public class Student{
    private Vector<Course> courseList;
    private int id;
    public Student(int stuID){
        courseList = new Vector<Course>();
        id = stuID;
    }
    public Vector<Course> get() {
        return courseList;
    }
    public boolean append(Course c){
        if(!exist(c)) courseList.append(c);
        return true;
    }
    ...other methods...
}
```

```
public class Manager{
    private Vector<Student> studentList;
    public void manage(int stuID){
        ...retrieve the student by the stuID
        list = student.get();
        for (Course c: list){
            if(c.matches()) c.update(...);
        }
    }
    public boolean selectCourse(int stuID, Course c){
        ...retrieve the student by the stuID
        return student.append(c);
    }
}
```


如何确保线程安全

- 使用不可变对象
 - 使用**final**来强制限制对属性成员的修改
 - 要小心对可变对象的引用
- 使用可变对象
 - 操作的原子性
 - 共享对象要始终处于严密控制之下
 - 不能直接**return**出去
 - 读、写访问的互斥控制
 - 更改的及时可见
 - 设置同步控制范围

```
public class Board{ //推箱子面板
    private final Set<Box> boxList;
    public Board(){
        boxList = new ...
        while(...){
            Box box = new Box(...);
            boxList.add(box);}
    }
    public Box get(Position p) {
        box = .... //retrieve a box according to p
        return box;
    }
    public void draw()
    {
        ... draw all the boxes in the board...
    }
}
```

如何确保线程安全

- 同步控制范围
 - 方法中代码块同步: `synchronized(e){...}`
 - 每个对象默认都有唯一的一个锁
 - 整个方法同步: `public synchronized xxx method(...){...}`
 - 使用的是什么锁?
- 同步控制范围越大, 其他线程等待时间就会越长
 - 性能问题
 - 尽量减少锁的使用, 尽量缩小锁控制的范围

如何确保线程安全

- 无法修改一个非thread-safe类的实现时怎么办？
- 使用thread-safe wrapper--Monitor Pattern
 - 设计一个线程安全Wrapper类，管理对目标类对象的访问
 - Wrapper类提供线程安全的方法来访问所管理的对象(delegation)

```
public class A{  
    private int value;  
    private boolean odd;  
    public A(){  
        value = 0; odd = false;}  
    public void increase(int x) {  
        value += x;  
        if(x%2 ==1 || x%2 == -1) odd = true;  
        else odd = false;  
    }  
}
```



```
public class SafeA{  
    private A a;  
    public SafeA(){  
        a = new A();  
    }  
    public synchronized void increase(int x) {  
        a.increase(x);  
    }  
}
```

确保线程安全的设计

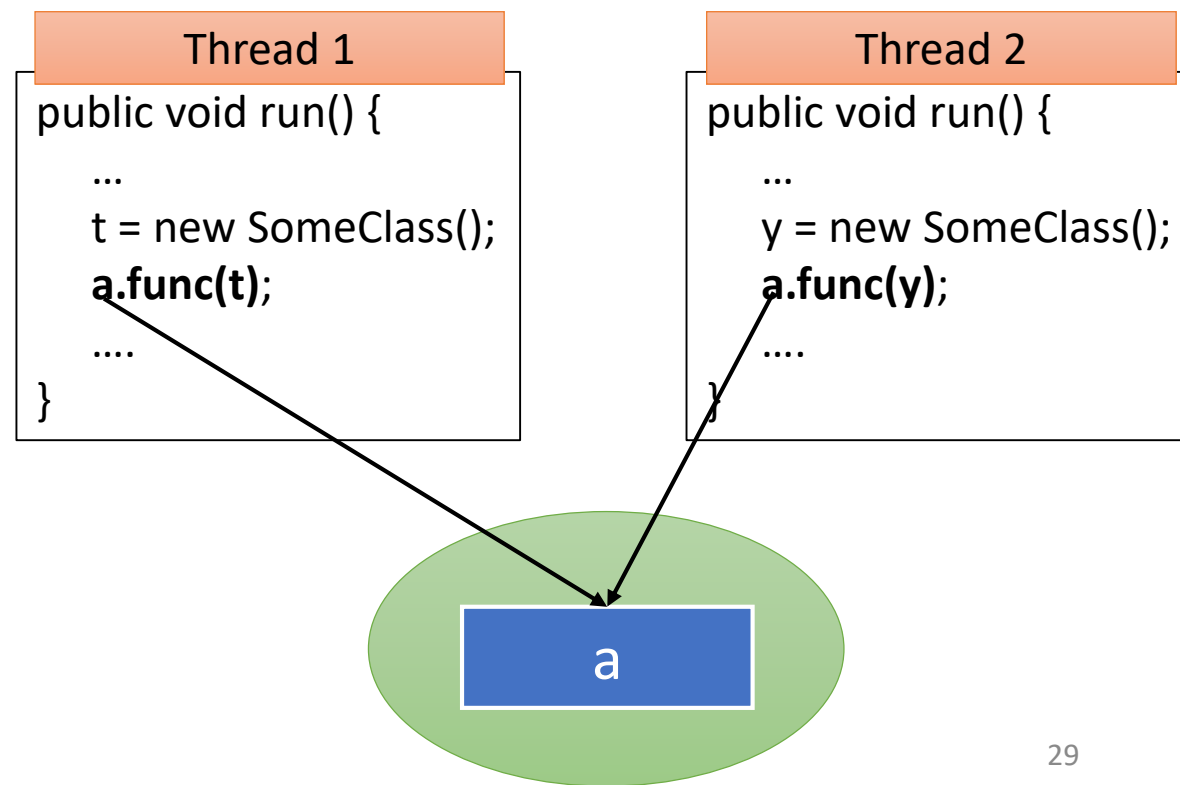
- 识别形成对象状态的域变量
 - 关注所属关系(ownership): 即哪些类拥有/管理相应的对象
 - 一个对象可能由不同的类拥有, 形成shared ownership→产生共享
- 识别对象状态需满足的约束
 - 状态变量独立
 - 状态变量不独立
- 定义同步策略来管理对该对象的并发访问
 - Backward analysis: 从对象出发, 循着ownership逆向分析, 直至thread
 - 确保对象的状态约束在任何线程的访问中都能满足

确保线程安全的设计

- 同步策略

- 同步控制哪些对象的访问
 - 被多个线程共享的任何可变对象
- 谁来负责同步控制
 - 共享对象自身
 - 使用共享对象的对象
- 使用什么锁
 - 使用共享对象的this
 - 使用共享对象所管理的局部对象锁
 - **使用传递参数对象锁**
- 同步控制范围
 - 方法级别
 - 语句块级别

```
public class A{  
    ...  
    public void func(Object x) {  
        ...  
        synchronized(x){  
            ...do something...  
        }  
    }  
}
```



确保线程安全的设计

- 归结起来，三个关键要素
 - 控制对象发布和共享
 - 复杂的对象引用和共享是导致程序死锁或数据竞争的主要原因
 - 共享对象设计为线程安全类
 - 尽可能减少同步范围，尽量保持在共享对象内部
 - 使用共享对象的类无需采取针对共享对象的同步控制措施
 - 保持简洁的线程类
 - `run`方法只负责顶层的控制逻辑
 - 尽量不要管理业务数据，更加不要让一个线程对象去访问另一个线程对象
 - 尽量不提供业务方法

多线程程序设计总结

- 关键一：哪些类该设计为线程？
 - 根据软件需求/任务要求
 - 在问题域中具有并发行为的类
- 关键二：该构造和启动多少个线程？
 - 根据运行时的数据量变化（如数据处理应用）
 - 根据运行时数据产生的独立性/并发性特征（如服务器应用中考虑的客户请求处理）
- 关键三：线程之间需要共享哪些对象？
 - 保守策略：除非必要，尽量不要在线程之间产生共享
- 关键四：线程共享对象如何确保安全访问？
 - 设计为线程安全类

作业

- IFTTT是互联网的一种应用形态，它支持以IF X THEN Y的方式来定义任务，并能够在后台自动执行任务，比如：
 - IF {news.163.com has new message} THEN {drag the message to my blog}
- 针对桌面操作系统的文件系统，支持如下任务
 - 触发器(IF): renamed, modified, path-changed, size-changed
 - 目标文件的规模计算为直接下层非目录文件的规模之和
 - 任务(THEN): record-summary, record-detail, recover
- 变化summary: 按照所定义触发器类别的变化数目统计
- 变化detail: 按照所定义触发器类别的变化详情，包括发生变化的文件夹、文件等

<https://ifttt.com/recipes>

作业

- 要求使用线程安全设计
 - 设计线程安全的文件访问类和其他有可能被共享的类，使用多线程进行检测和处理
 - 能够支持任意深度的目录，能够同时监视不少于5个，不多于8个目录
 - 要求文件访问类（线程安全类）提供文件修改方法供测试使用
- 开展自动化测试
 - 每个同学按照提供的测试线程样例来对所分配程序进行测试
 - 需要构造单独的专用于测试的main方法
 - 测试线程可以使用被测程序中提供的文件访问类,也可以自己编写相关的类来实施测试.如果是后者,则需要保证自己的类必须线程安全类,否则导致的问题不能算作被测程序的bug.
 - 不允许手工方式在操作系统层次来开展测试,只能通过测试线程实施测试