

《面向对象设计与构造》

Lec06-线程安全及其设计

OO课程组2023

北京航空航天大学计算机学院

提纲

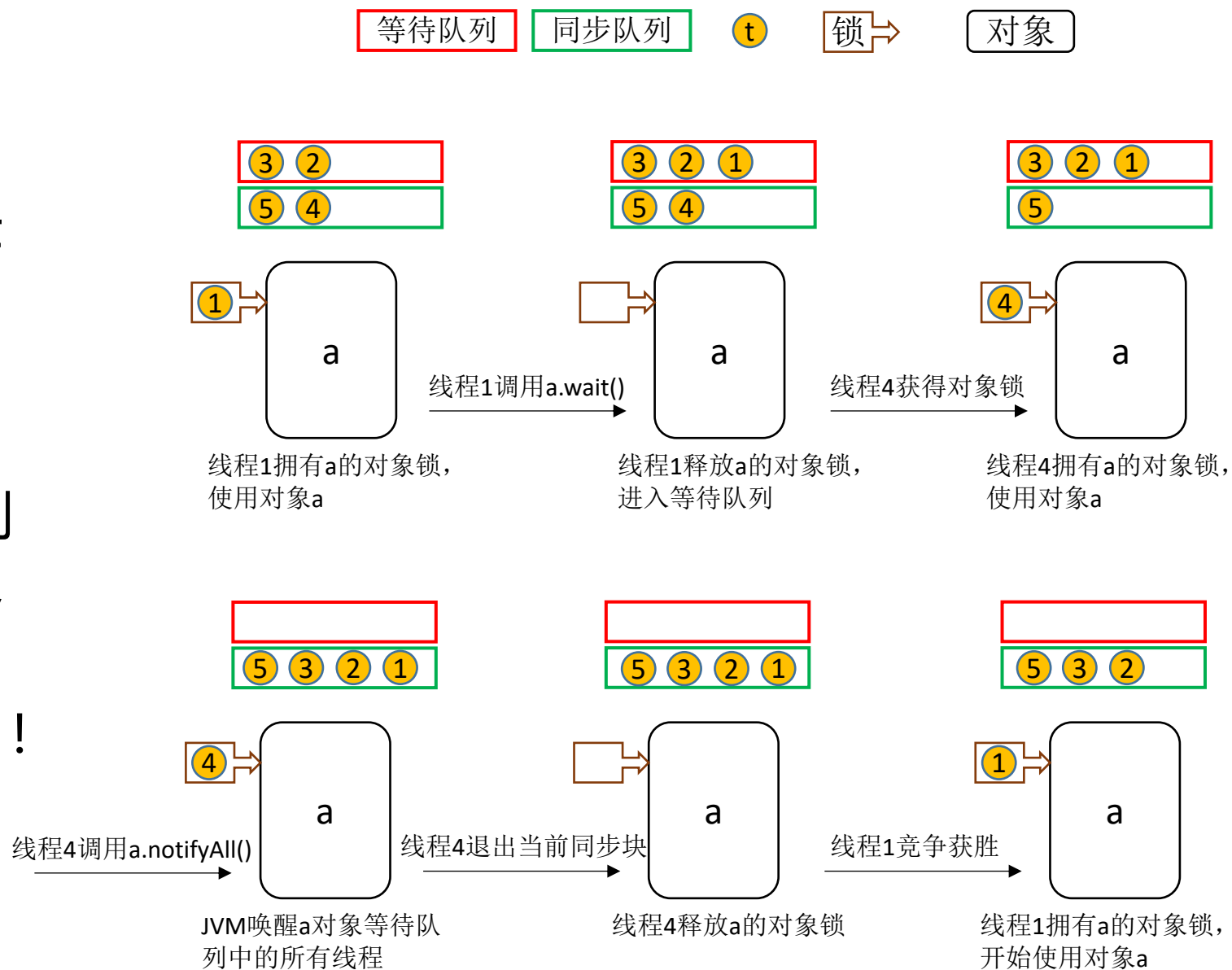
- 线程交互需要把握的要点
- 线程安全特性
- 导致不安全的计算模式
- 常见的不安全场景
- 线程安全设计
- Lock机制
- 线程交互设计架构
- 作业解析

线程交互需要把握的要点

- 任何时候当wait, notify, notifyAll执行时
 - 该语句的执行一定处于某个同步控制块
 - 该语句的执行一定处于某个对象的上下文
 - 该语句的执行一定处于某个线程的上下文(currentThread)
 - 该语句的执行一定表明currentThread获得了所在同步控制块的monitor
 - wait的含义是通知JVM
 - 把currentThread调度进入monitor所关联的等待队列
 - 释放monitor的锁
 - notify的含义是通知JVM
 - 从monitor等待队列中随机选择一个唤醒来调度执行
 - notifyAll的含义是通知JVM
 - 把monitor等待队列中的所有线程都唤醒，并按照某种策略调度一个来执行

Monitor是什么

- 可以由任意对象来担任
- Monitor内置
 - 唯一的一把锁
 - 一个等待队列
 - 一个同步队列/入口队列
- 共享对象与锁必须严格配合
 - 使用共享对象内置的锁！



理解多线程程序的特殊性

- wait的执行会立刻释放锁，并把当前线程置入等待队列
- notify/notifyAll不会立刻释放锁
 - 执行退出同步块时释放
 - notify/notifyAll后的语句会继续执行
- 如果wait所属对象与monitor不一致会如何？
 - `synchronized (obj1){...obj2.wait();...}`
- 如果notify/notifyAll处于一个循环中会如何？
- 如果notify/notifyAll后跟着一个循环会如何？

线程安全是一个重要的对象特性

- 线程安全（ thread-safe ）是关于对象是否能被多个线程安全访问的特性。如果一个对象是线程安全的，则无论多个线程以什么样的交叠次序来访问都不会影响该对象的行为结果
- 如果一个类是线程安全的，则其任意一个对象也都是线程安全的
- 任何时候访问一个线程安全对象，都无需做任何同步控制措施
 - 根据对象的任务性质，有可能需要让线程sleep或wait，避免CPU空转

Not(线程安全)

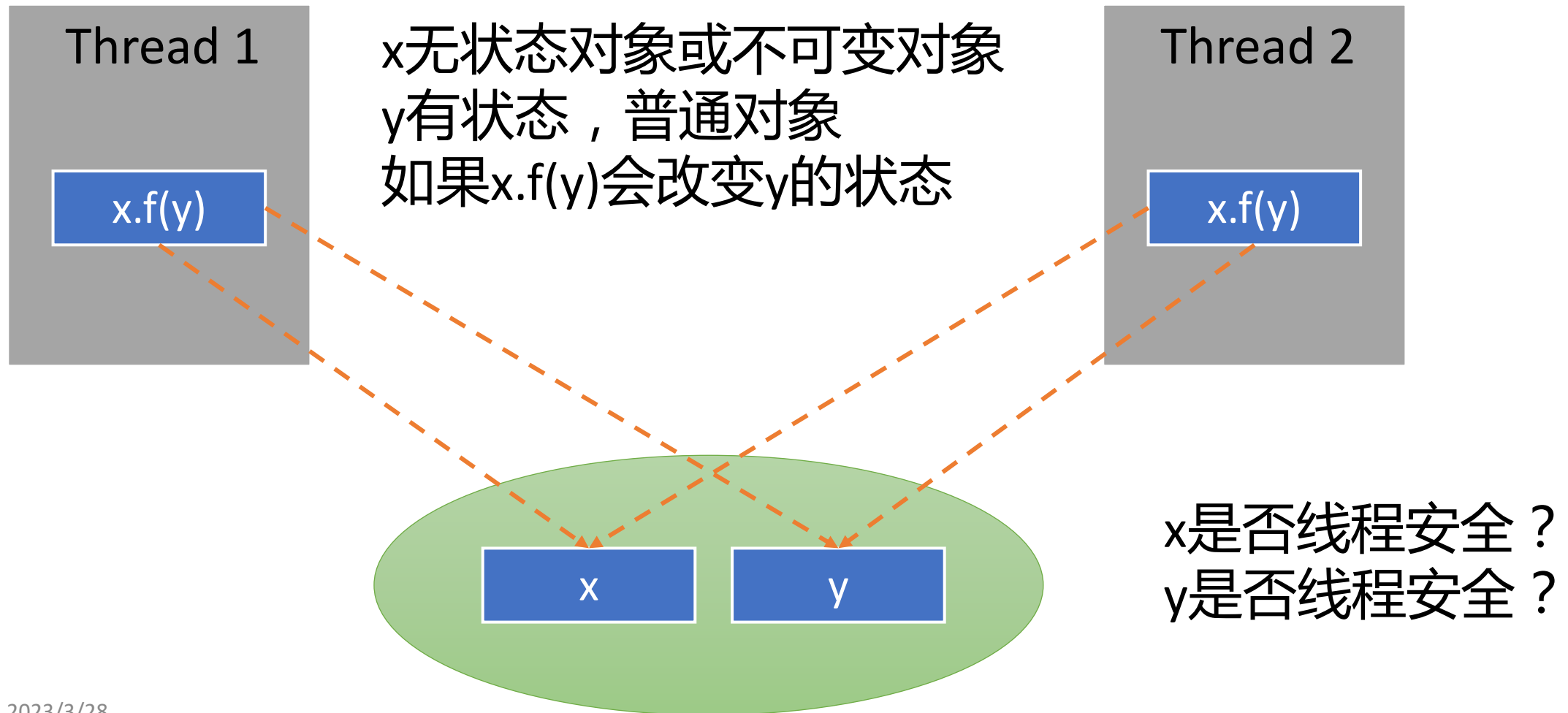
- 线程不安全对象：给定对象状态和访问该对象方法所用的参数，存在一个**线程调度执行序列**，使得这些线程在该对象上的执行结果与其他调度序列下执行产生的**结果不一致**

- 不可变对象是否有线程不安全问题？
- 非共享对象是否有线程不安全问题？
- 无状态对象是否有线程不安全问题？

线程不安全的**必要条件**：
多个线程**共享**、至少一个
线程**改变其状态**

- 线程安全是否意味着**行为正确**？
- 如果多个线程**只读取**共享对象的状态，是否有不安全问题？

如果参数传递的对象也共享会如何？



导致安全问题的两种经典计算模式

- Read-modify-write和check-then-act是经典的计算模式
 - 首先读取对象的状态 (optional : 检查对象状态是否满足某个条件)
 - 然后使用获取的状态做相关计算
 - 最后改变对象的状态

read-modify-write

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

Update **x**: **x**+1

Write to **x**: **x**=....

check-then-act

```
If (b_exp(x)){  
    do something with x;  
    update x;  
}
```

导致安全问题的两种经典计算模式

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

```
import java.util.concurrent.atomic;  
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);  
    }  
}
```

导致安全问题的两种经典计算模式

- 不安全问题的表征
 - 多个修改发生覆盖
 - 读取不到最新的结果
- Read-modify-write和check-then-act出现线程安全问题的本质：
 - 读取+修改
 - 计算过程需要多步完成
 - 线程调度**无法提供原子性保证**
- 互斥机制是一种建立**计算原子性**的技术手段
 - 保证计算原子性的数据类型
 - 保证计算原子性的同步块

提供计算原子性的Java类型

- 如果read-modify-write和check-then-act计算只涉及单一变量，可以使用java.util.concurrent.atomic包中提供的原子类型

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

多个原子类型对象之间是否有依赖关系？

```
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)
==》必须扩大同步控制范围，否则会出现数据一致性问题

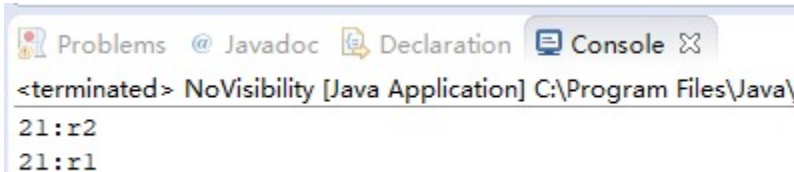
假设lastNumber和lastFactors之间没有关系，是否仍然有问题？

共享对象的读写冲突

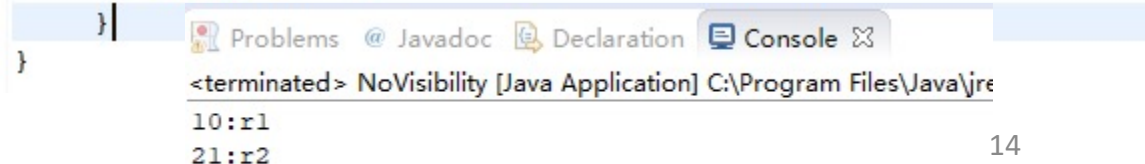
- 问题表现：共享对象的状态更新被覆盖；读取的状态不完整等
 - 线程A “in writing access”，线程B “start the writing or reading access”
 - 线程A “in reading access”，线程B “start the writing access”
- 状态更新延迟
 - 没有读写冲突
 - 读操作无法及时获得状态的更新

```
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;
    }
}
```



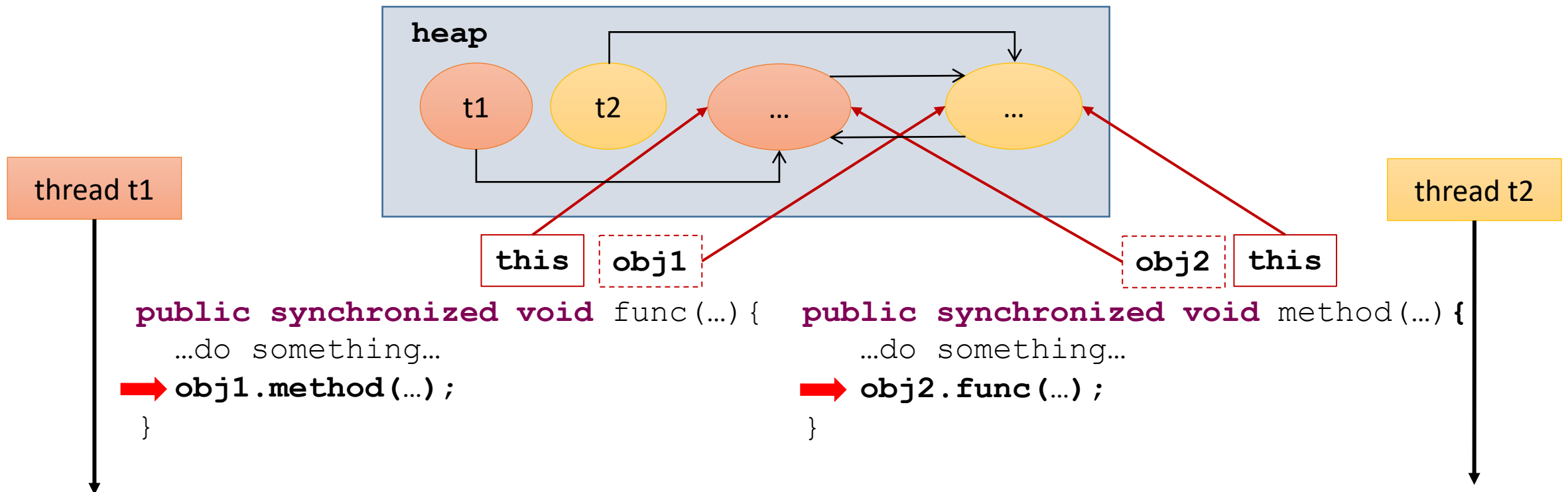
```
<terminated> NoVisibility [Java Application] C:\Program Files\Java\
21:r2
21:r1
```



```
<terminated> NoVisibility [Java Application] C:\Program Files\Java\jre
10:r1
21:r2
```

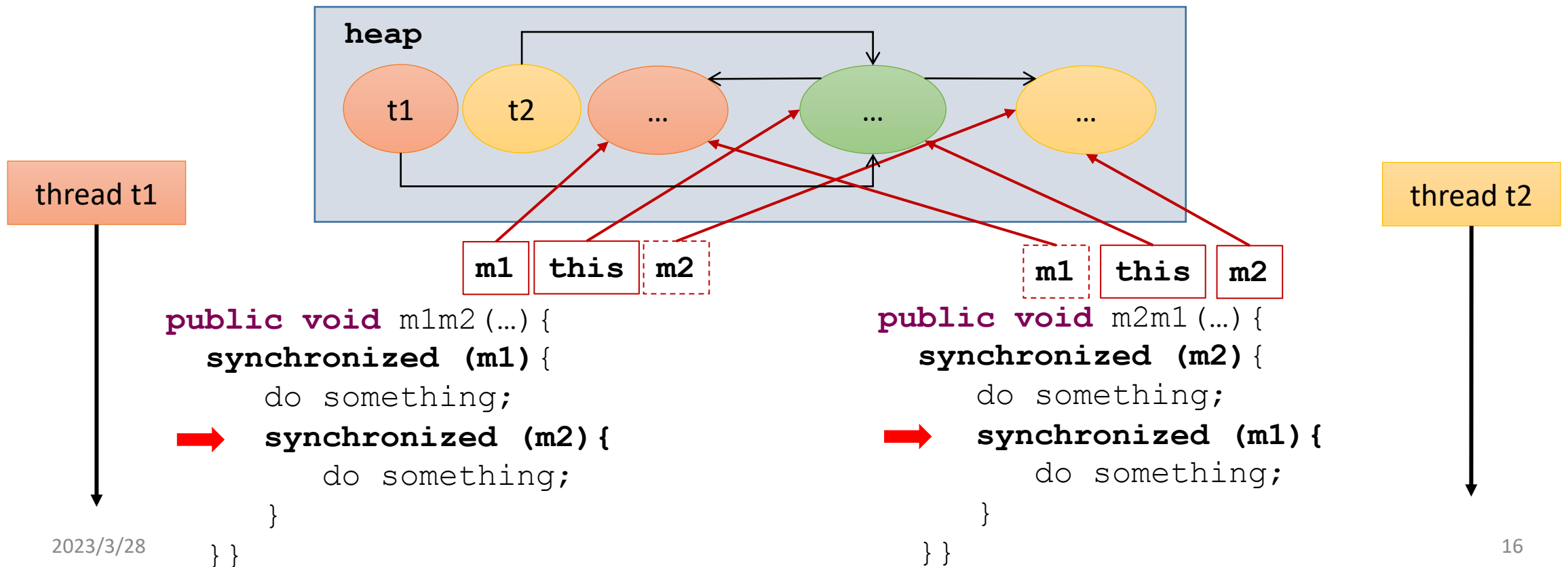
共享对象的循环引用导致的死锁

- 两个线程所配置的共享对象相互引用（循环引用）
 - 执行时需要互相访问对方的共享对象，发生死锁



Monitor的循环依赖导致的死锁

- 两个线程的共享对象使用了两个循环依赖的monitor
 - 执行需要对方的monitor，发生死锁



对象共享是产生线程安全问题的根本原因

- 程序中有多重操作会产生对象共享
 - 参数传递对象
 - 返回对象
 - 对象引用赋值
 - 把对象存入外部可共享访问的容器
- 在分析一个类是否存在线程安全问题时，首先是分析它会被哪些线程共享
 - 线程内共享
 - 线程间共享

不受控的线程间对象共享

```
public class Student{
    private ArrayList<Course> courseList;
    private int id;
    public Student(int stuID){
        courseList = new ArrayList<Course>();
        id = stuID;
    }
    public synchronized ArrayList getCourseList() {
        return courseList;
    }
    public synchronized void append(Course c){
        if(!exist(c)) courseList.append(c);
    }
}
```

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

如果Student的某个对象在线程间共享，courseList以及其中管理的所有Course对象都是线程间共享对象

线程安全的设计考虑

- 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
 - 避免不受控的对象发布
 - 设置**范围适当**的临界区
 - 使用**合适**的监控器monitor

```
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...
    }
}
```

线程安全的设计考虑

- 同步控制范围越大，其他线程等待时间就会越长
- 临界区和监控器的**匹配**
 - 经验原则：临界区中代码围绕监控器对象开展工作
 - 如果临界区代码不访问监控器（而访问其他共享对象），则无法获得安全保护的效果
- 临界区**最小化**
 - 如果一个临界区中某行代码不访问monitor
 - 如果临界区中某行代码仅访问monitor中的不可变数据
 - 应把相应的代码移出临界区

基于锁的线程安全设计

- Java语言同时提供了锁机制来控制同步访问
 - ReentrantLock：可随处使用
 - ReentrantReadWriteLock：单写、多读
 - 比Monitor机制更加灵活

```
public void put(int i) {  
    locker.lock();  
    try {  
        list.add(new Integer(i));  
    } finally {  
        locker.unlock();  
    }  
}
```

```
public int get() {  
    locker.lock();  
    Integer i;  
    try {  
        i = list.remove(0);  
    } finally {  
        locker.unlock();  
    }  
    return i.intValue();  
}
```

```
public int size() {  
    locker.lock();  
    int i;  
    try {  
        i = list.size();  
    } finally {  
        locker.unlock();  
    }  
    return i;  
}
```

```
public void run() {  
    int i;  
    while (true) {  
        if(tray.size() > 0) {  
            i = tray.get();  
            System.out.println(tname + " consumed: "+i);  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) { }  
        }  
    }  
}  
  
public void run() {  
    while (tray.size() < 10) {  
        int i = (int)(Math.random()*100);  
        tray.put(i);  
        System.out.println(tname + " produced: "+i);  
        try {  
            Thread.sleep(100);  
        } catch (InterruptedException e) {}  
    }  
}
```

读写同步控制锁

- Monitor使用JVM内置的锁机制
 - 简洁、易用，效果等同于ReentrantLock
- 读-写冲突和写-写冲突是最突出的问题，既避免冲突，同时又获得高性能
 - 写互斥，读共享
- Read Write Lock
 - 记录读写线程
 - #readingReaders, #writingWriters, #waitingWriters
 - 控制规则
 - (writingWriters == 0) → 获得read lock
 - (writingWriters == 0 && readingReaders == 0) → 获得write lock

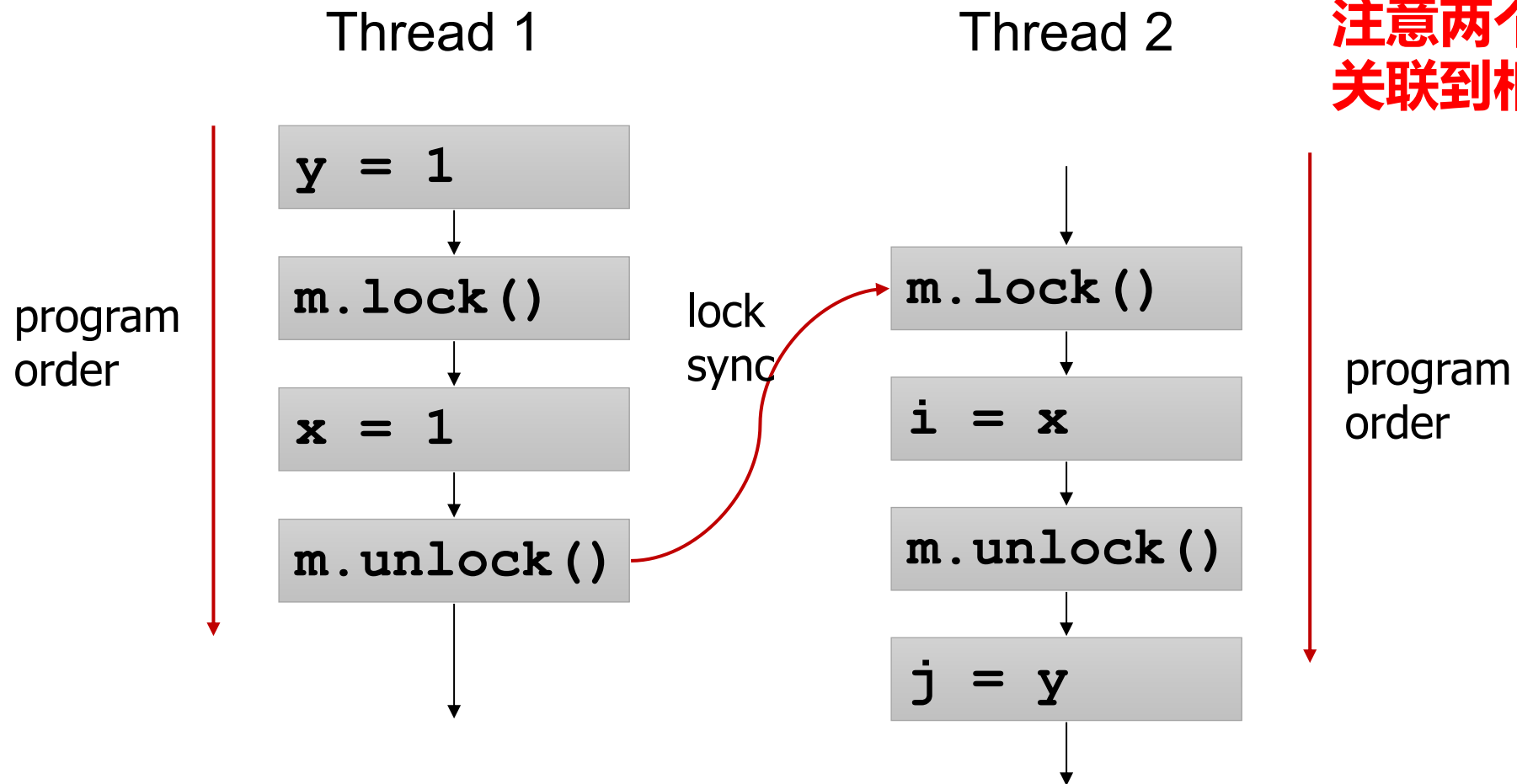
<http://tutorials.jenkov.com/java-concurrency/read-write-locks.html>

```
public void put(int i) {
    locker.writeLock().lock();
    try {
        list.add(new Integer(i));
    } finally {
        locker.writeLock().unlock();
    }
}

public int get() {
    locker.readLock().lock();
    Integer i;
    try {
        i = list.remove(0);
    } finally {
        locker.readLock().unlock();
    }
    return i.intValue();
}

public int size() {
    locker.readLock().lock();
    int i;
    try {
        i = list.size();
    } finally {
        locker.readLock().unlock();
    }
    return i;
}
```

锁的同步控制问题

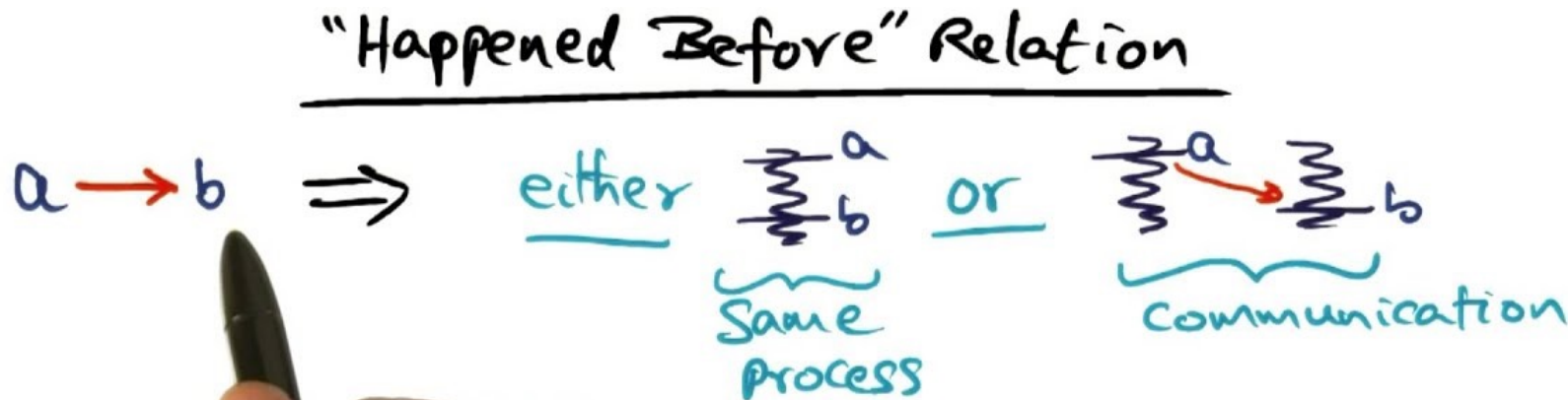


**注意两个锁m必须
关联到相同的对象**

确保m.lock之后的代码能获得m.unlock之前的x最新取值(happened-before)

Happened-Before关系

- 如果happened-before关系得到满足，则不会发生读写不一致问题
 - 任意两个操作a和b，如果满足happened-before关系，假设a先于b执行，则b执行时可以看到a操作产生的最新结果
 - 如果a和b所读写的对象非共享，则无所谓happened-before关系是否满足
 - read-modify-write模式中的happened-before?
 - check-then-act模式中的happened-before?



Leslie Lamport . Time, Clocks and the Ordering of Events in a Distributed System. 1978

线程安全的关键是保护共享数据的读写

- 识别对象数据的管理层次
 - 梳理数据管理层次(ownership, part-of)
 - 一个对象可能由多个对象**拥有**，产生**共享**
- 识别对象状态需满足的约束
 - 状态变量独立：修改时顺序不敏感，独立保护即可
 - 状态变量不独立：修改时顺序敏感，需要联合保护
- 确定同步控制范围
 - 确定共享对象集合
 - 共享对象的所有访问语句都需要进行同步控制

问题：对象是运行时创建的，无法静态确定对象集合

确保线程安全的设计

- 从设计上确认用来构造共享对象的类
 - 设计为线程安全类
 - 自己管理好同步控制
 - 外部无需额外保护
- 确定同步控制范围：以共享数据为中心
 - 方法级别
 - 语句块级别

Safety first



- 确定同步控制机制
 - Monitor机制：Java内置
 - Lock机制：需额外构造锁对象
- 确定使用什么锁
- **在monitor或locker对象与被保护数据之间建立语义关联，应封装在一起！**

非线程安全类的封装处理

- 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)
 - 选择Monitor时要注意被保护类中是否有static属性！

```
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{  
    final 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

- 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
 - 访问共享对象的类无需额外采取同步控制措施
 - 同步控制范围不宜过大，性能与安全的平衡
 - 读写锁一般可以获得更好的性能
 - 保持简洁的线程类
 - run方法只负责顶层的控制逻辑
 - 不去管理具体业务数据
 - 不要让一个线程对象去访问另一个线程对象

课堂讨论

- 右边的代码基本给出了ReentrantLock的实现逻辑
 - 如果一个线程调用了lock之后，连续两次调用unlock会发生情况？如何修复这个问题？
 - 假设某个线程在获得了lock之后的工作出现了长时间不结束，希望Lock能够提供一个挤占方法，让被阻塞线程获得lock，并终止长时间未unlock的线程，该怎么修改？

```
public class Lock{
    boolean isLocked = false;
    Thread lockedBy = null;
    int    lockedCount = 0;
    public synchronized void lock() {
        Thread callingThread = Thread.currentThread();
        while(isLocked && lockedBy != callingThread) wait();
        isLocked = true;
        lockedCount++;
        lockedBy = callingThread;
    }
    public synchronized void unlock() {
        if(Thread.currentThread() == this.lockedBy) {
            lockedCount--;
            if(lockedCount == 0){
                isLocked = false;
                notifyAll();
            }
        }
    }
}
```

线程交互的架构设计

- 线程交互是并发场景设计的一个关键问题，往往和业务有交织
- 有三种不同的显著特征或模式
 - 多种请求、集中处理模式：服务器架构
 - 多种请求、分段处理模式：流水线架构
 - 多种请求、特化处理模式：事件驱动架构
- 归根结底还是服务于业务处理
 - 请求类型
 - 处理要求
 - 不同处理之间的协调与配合

服务器架构的线程交互

- 多种请求、集中处理
- 一个线程充当Server（集中处理），按照请求类型建立一到多个共享的请求队列
 - bankingServer（银行窗口服务线程）
 - cashQueue；loanQueue；investQueue；
- Server线程按照策略从多个queue中提取请求来进行处理
 - 不可抢占的调度策略：请求等待时间、请求到达时间等因素
 - 可抢占的调度策略：队列优先级

流水架构的线程交互

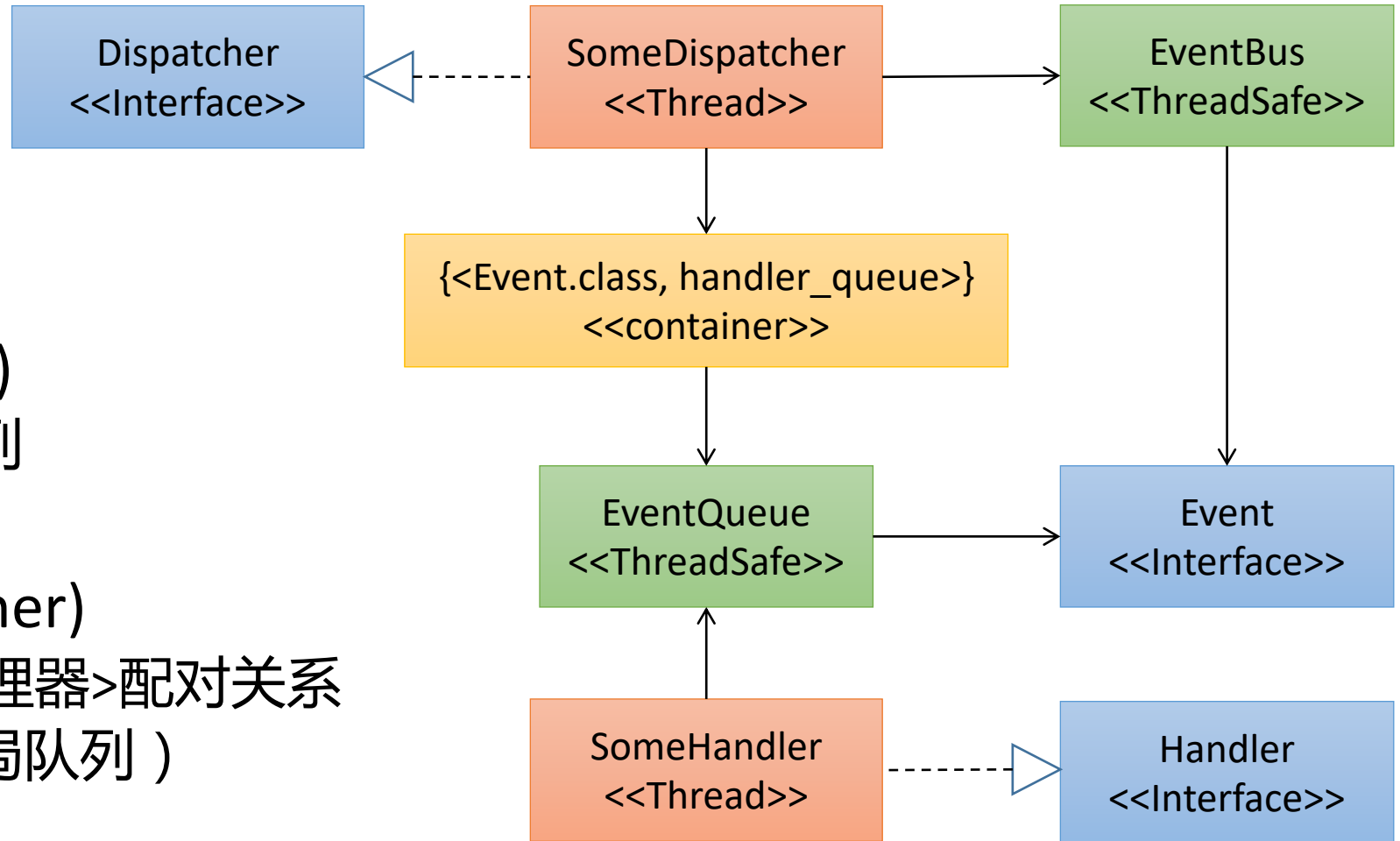
- 对请求的处理具有分段流水特征
 - 如贷款申请的处理
 - 不同类型的请求可能需要不同的分段处理
- 每一段处理都是一个线程，从输入队列取出请求，处理完毕后放入输出队列
 - 处理线程之间互不知晓，通过输入输出队列进行交互
- 如何为分段处理线程配置队列？
 - 流水线控制器为每个分段处理线程配置输入和输出队列
 - ➔配置了分段处理线程之间的工作顺序关系
 - 流水线控制器是个协调者

事件驱动架构的线程交互

- 多种请求、**特化**处理
- 事件驱动架构
 - 按照事件类型来**注册**处理器
 - 按照事件类型在发生时**分派**给相应的处理器
 - 事件、处理器、分派器（也称为转发器）均可通过接口来定义
- 线程角色
 - 处理器：专门处理某一类事件的线程
 - 分派器：针对事件类型来把相应的事件分派给合适的处理器
- 优势
 - 用户程序无需关心谁处理当前事件，只需投进（与分派器共享）指定的队列
 - 用户程序可以根据需要扩展事件、处理器及其配对关系

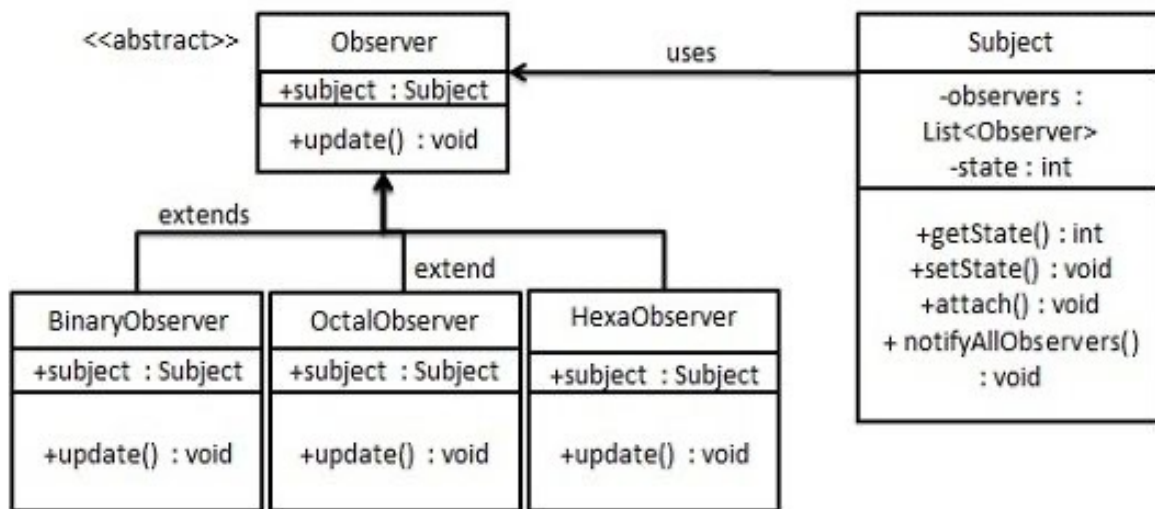
事件驱动架构的线程交互

- 事件接口(Event)
 - 获取事件类型
 - 获取事件ID
 - 获取事件数据
- 处理器接口(Handler)
 - 设置局部的事件队列
 - 处理到达的事件
- 分派器接口(Dispatcher)
 - 注册<eventType, 处理器>配对关系
 - 配置事件总线 (全局队列)



流水线架构中的请求状态变化处理

- 分段处理线程在多个队列间“搬运”请求
 - 请求状态发生变化
 - 外部如何知道处理到什么阶段了？
- 如何通知关心者的问题
 - 提供一个接口让观察者自己来查询
 - 状态变化时通知观察者
- 观察者查询方式
 - 多线程场景下建立很多线程交互，复杂
 - 单线程场景下依赖于很多不同的具体观察者
- 推送方式（观察者模式）
 - 多线程场景下也无需建立额外的线程交互
 - 可以针对观察者建立抽象层次，统一处理



流水线控制器接受观察者注册，按照观察者感兴趣的请求类型或者特定请求状态来建立通知关系：事件驱动的观察！

作业解析

- 功能迭代
 - 可以动态增加电梯
 - 可以动态维护某部电梯（即不再响应新的乘客请求）
- 线程安全设计
 - 让线程间共享对象自己管理好对自己的访问
 - 使用更加高效率的读写锁
- 线程协同架构
 - 识别共享对象：多级队列（全局、局部）
 - 识别线程角色：用户请求模拟、电梯、调度器
- 分层的调度策略
 - 首先决定分配给哪部电梯，依据？
 - 然后针对一部电梯看如何提高运载效率，依据？