并发编程（Concurrent Programming）

@M了个J
李明杰

https://github.com/CoderMJLee
http://cnblogs.com/mjios

码拉松

小码哥教育 SEEMYGO
实力IT教育 www.520it.com

进程（Process）

- 什么是进程?
  - 在操作系统中运行的一个应用程序

- 比如同时打开QQ、微信，操作系统就会分别启动2个进程

- 每个进程之间是独立的，每个进程均运行在其专用且受保护的内存空间内
- 在 Windows 中，可以通过“任务管理器”查看正在运行的进程

小码哥教育 SEEMYGO

线程（Thread）

- 什么是线程？
  - 1 个进程要想执行任务，必须得有线程（每 1 个进程至少要有 1 个线程）
  - 一个进程的所有任务都在线程中执行

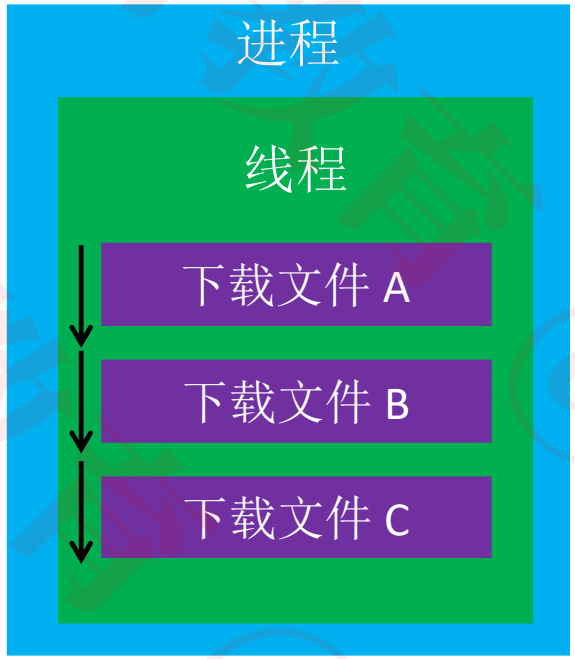
- 比如使用酷狗播放音乐、使用迅雷下载文件，都需要在线程中执行

线程的串行

- 1 个线程中任务的执行是串行的
  - 如果要在 1 个线程中执行多个任务，那么只能一个一个地按顺序执行这些任务
  - 在同一时间内，1 个线程只能执行 1 个任务

- 比如在 1 个线程中下载 3 个文件（分别是文件 A、文件 B、文件 C）

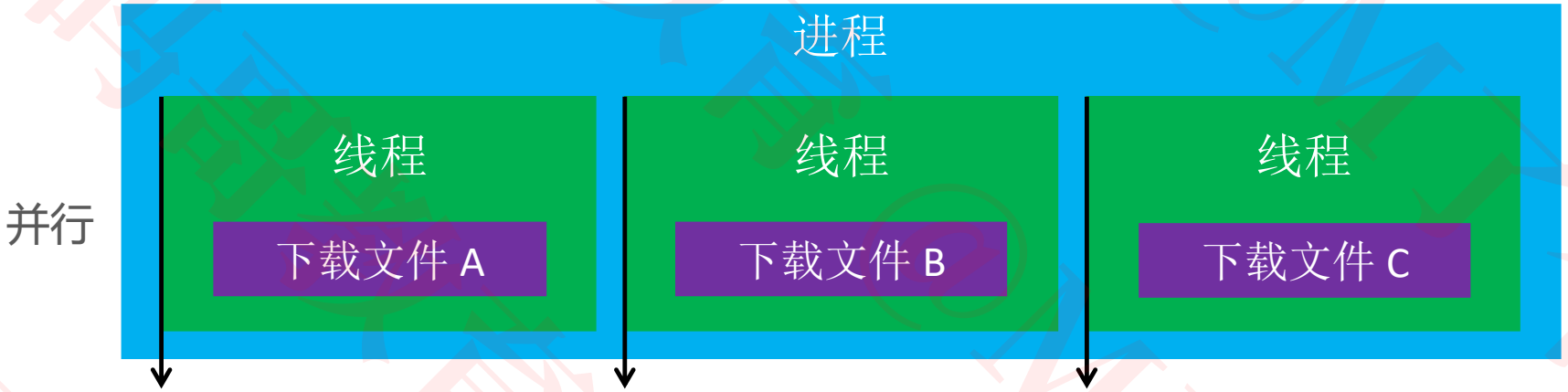
串行



多线程

- 什么是多线程
  - 1 个进程中可以开启多个线程，所有线程可以并行（同时）执行不同的任务
  - 进程 → 车间，线程 → 车间工人
  - 多线程技术可以提高程序的执行效率

- 比如同时开启 3 个线程分别下载 3 个文件（分别是文件 A、文件 B、文件 C）



小码哥教育 SEEMYGO

多线程的原理

- 同一时间，CPU 的 1 个核心只能处理 1 个线程（只有 1 个线程在工作）
- 多线程并发（同时）执行，其实是 CPU 快速地在多个线程之间调度（切换）

- 如果 CPU 调度线程的速度足够快，就造成了多线程并发执行的假象
- 如果是多核 CPU，才是真正地实现了多个线程同时执行
- 思考：如果线程非常非常多，会发生什么情况？
  - CPU 会在 N 个线程之间调度，消耗大量的 CPU 资源，CPU 会累死
  - 每条线程被调度执行的频次会降低（线程的执行效率降低）

多线程的优缺点

- 优点
  - 能适当提高程序的执行效率
  - 能适当提高资源利用率（CPU、内存利用率）

- 缺点
  - 开启线程需要占用一定的内存空间，如果开启大量的线程，会占用大量的内存空间，降低程序的性能
  - 线程越多，CPU 在调度线程上的开销就越大
  - 程序设计更加复杂
    - ✓ 比如线程之间的通信问题、多线程的数据共享问题

小码哥教育 SEEMYGO

默认线程

- 每一个 Java 程序启动后，会默认开启一个线程，称为主线程（main 方法所在的线程）

- 每一个线程都是一个 java.lang.Thread 对象，可以通过 Thread.*currentThread* 方法获取当前的线程对象

```
public static void main(String[] args) {
    // Thread[main,5,main]
    System.out.println(Thread.currentThread());
}
```

小码哥教育 SEEMYGO

开启新线程的第 1 种方法

```
// 传入一个Runnable实例，在run方法中编写子线程需要执行的任务
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("开启了新线程：" + Thread.currentThread());
    }
});
thread.start();
```

```
Thread thread = new Thread(() -> {
    System.out.println("开启了新线程：" + Thread.currentThread());
});
thread.start();
```

小码哥教育 SEEMYGO

开启新线程的第 2 种方法

```
// 继承Thread，重写run方法
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("开启了新线程：" + Thread.currentThread());
    }
}
```

```
MyThread thread = new MyThread();
thread.start();
```

- 注意
  - 直接调用线程的 run 方法并不能开启新线程
  - 调用线程的 start 方法才能成功开启新线程

- Thread 类实现了 Runnable 接口

小码哥教育 SEEMYGO

多线程的内存布局

- PC 寄存器（Program Counter Register）：每一个线程都有自己的 PC 寄存器
- Java 虚拟机栈（Java Virtual Machine Stack）：每一个线程都有自己的 Java 虚拟机栈
- 堆（Heap）：多个线程共享堆
- 方法区（Method Area）：多个线程共享方法区
- 本地方法栈（Native Method Stack）：每一个线程都有自己的本地方法栈

线程的状态

- 可以通过 Thread.getState 方法获得线程的状态（线程一共有 6 种状态）

- NEW（新建）：尚未启动

- RUNNABLE（可运行状态）：正在 JVM 中运行
  - ✓ 或者正在等待操作系统的其他资源（比如处理器）

- BLOCKED（阻塞状态）：正在等待监视器锁（内部锁）

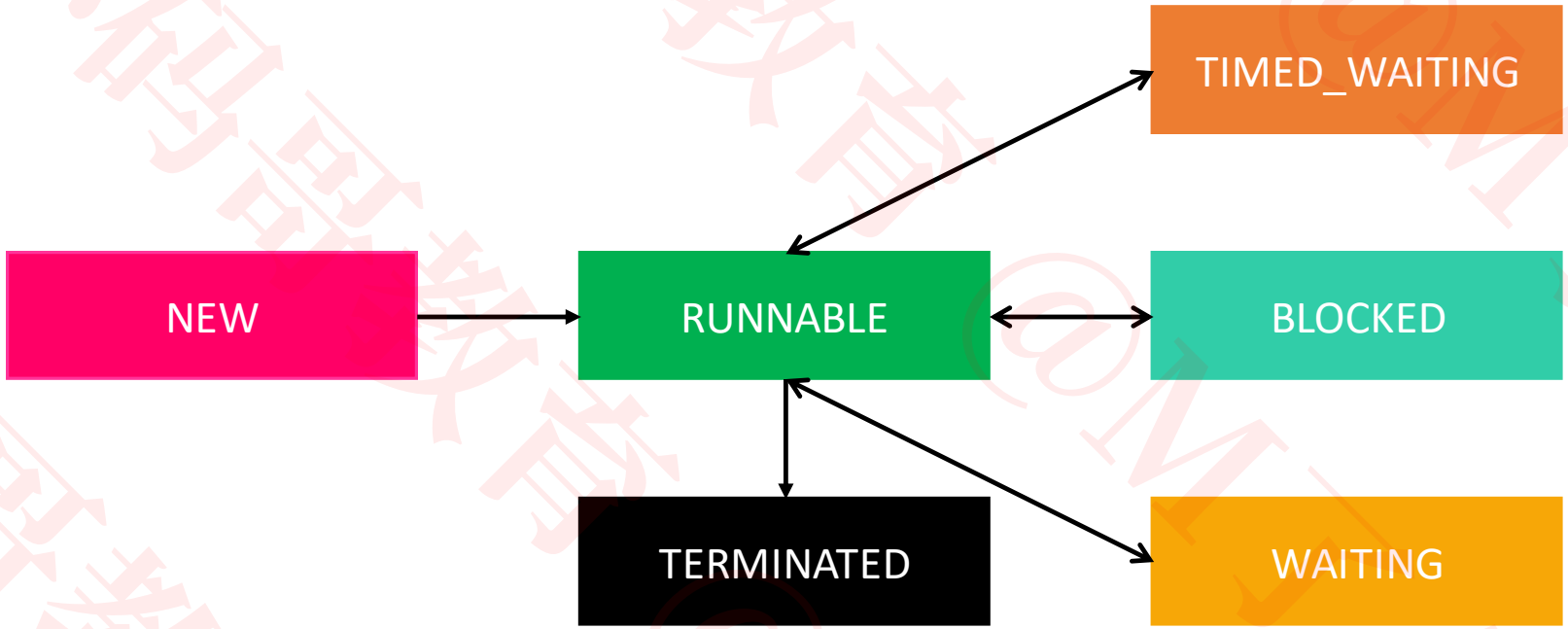
- WAITING（等待状态）：在等待另一个线程
  - 调用以下方法会处于等待状态
  - ✓ 没有超时值的 Object.wait
  - ✓ 没有超时值的 Thread.join
  - ✓ LockSupport.*park*

- TIMED_WAITING（定时等待状态）
  - 调用以下方法会处于定时等待状态
  - ✓ Thread.*sleep*
  - ✓ 有超时值的 Object.wait
  - ✓ 有超时值的 Thread.join
  - ✓ LockSupport.*parkNanos*
  - ✓ LockSupport.*parkUntil*

- TERMINATED（终止状态）：已经执行完毕

小码哥教育 SEEMYGO

线程的状态切换



sleep、interrupt

- 可以通过 Thread.*sleep* 方法暂停当前线程，进入 WAITING 状态
- 在暂停期间，若调用线程对象的 interrupt 方法中断线程，会抛出 java.lang.InterruptedException 异常

```
Thread thread = new Thread(() -> {
    System.out.println("begin");
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        System.out.println("interrupt");
    }
    System.out.println("end");
});
thread.start();
Thread.sleep(1000);
thread.interrupt();
```

```
begin
interrupt
end
```

join、isAlive

- A.join 方法：等线程 A 执行完毕后，当前线程再继续执行任务。可以传参指定最长等待时间

- A.isAlive 方法：查看线程 A 是否还活着

```
Thread t1 = new Thread(() -> {
    System.out.println("t1 - begin");
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("t1 - end");
});
t1.start();
```

小码哥教育 SEEMYGO

join、isAlive

```
Thread t2 = new Thread(() -> {
    System.out.println("t2 - begin");
    System.out.println("t1.isAlive - " + t1.isAlive());
    try {
        t1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("t1.state - " + t1.getState());
    System.out.println("t1.isAlive - " + t1.isAlive());
    System.out.println("t2 - end");
});
t2.start();
```

```
t1 - begin
t2 - begin
t1.isAlive - true
t1 - end
t1.isAlive - false
t2 - end
```

小码哥教育 SEEMYGO

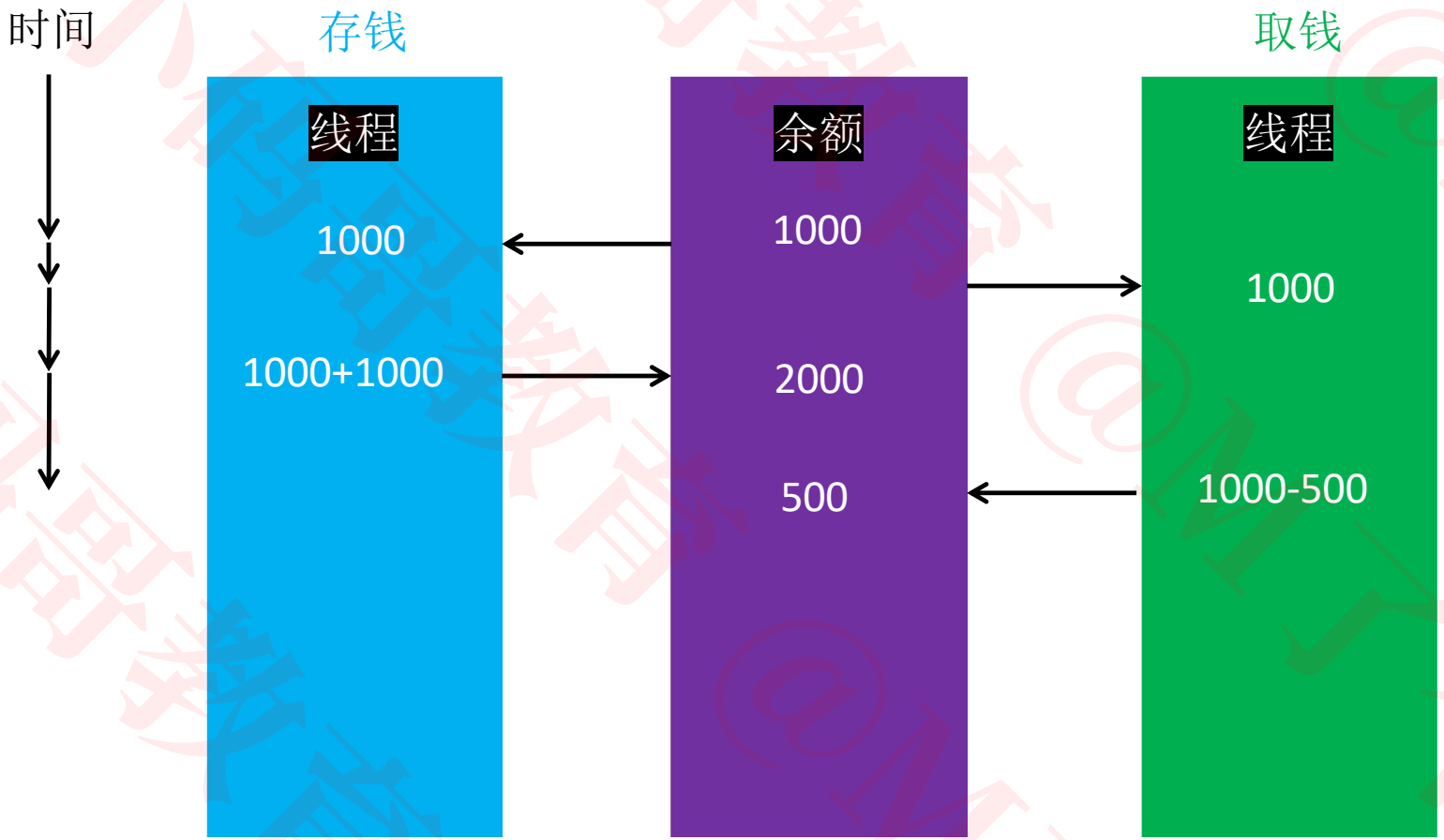
线程安全问题

- 多个线程可能会共享（访问）同一个资源
  - 比如访问同一个对象、同一个变量、同一个文件

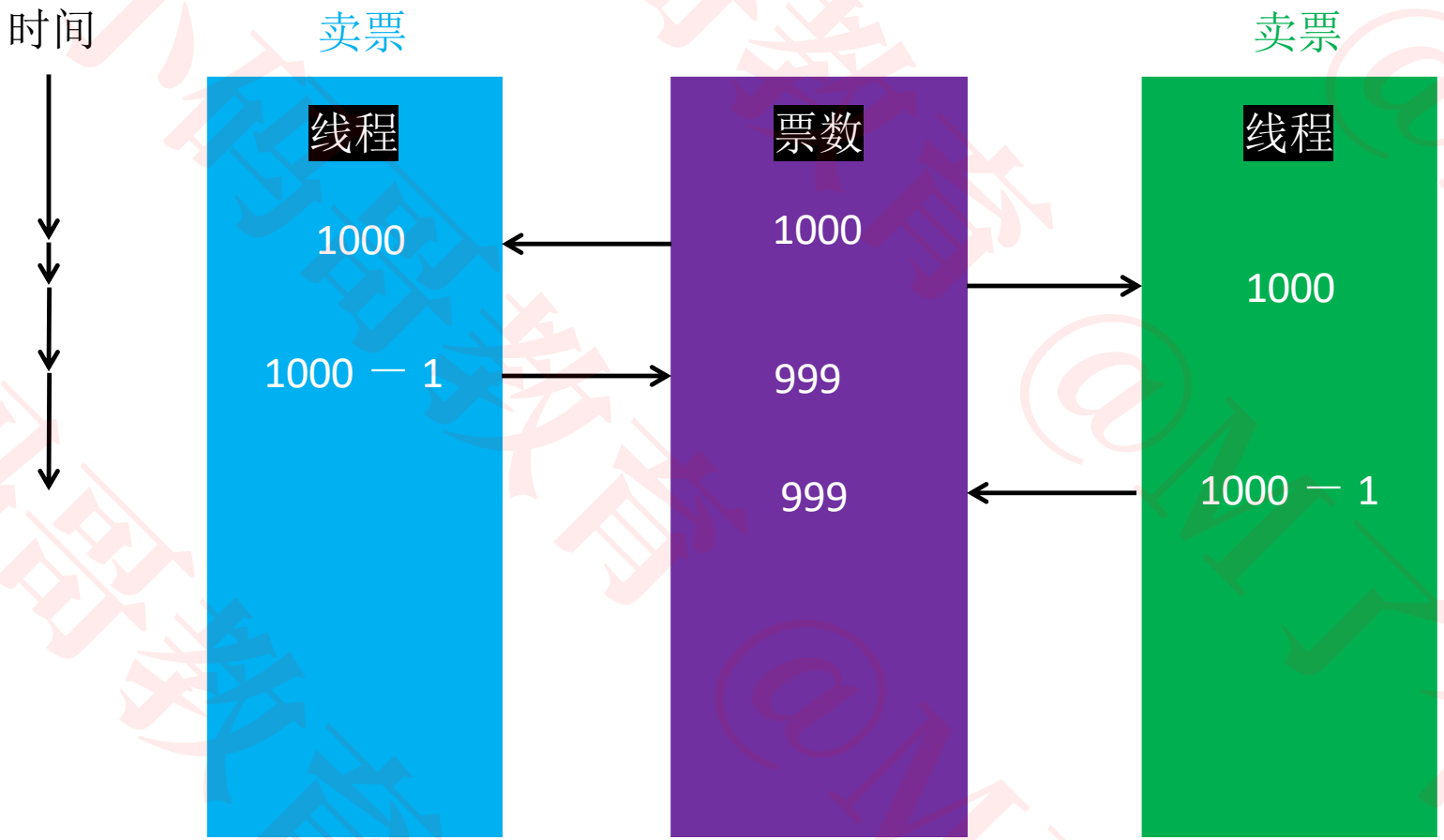
- 当多个线程访问同一块资源时，很容易引发数据错乱和数据安全问题，称为线程安全问题

- 什么情况下会出现线程安全问题？
  - 多个线程共享同一个资源
  - 且至少有一个线程正在进行写的操作

线程安全问题 – 存钱取钱



线程安全问题 – 卖票



小码哥教育 SEEMYGO

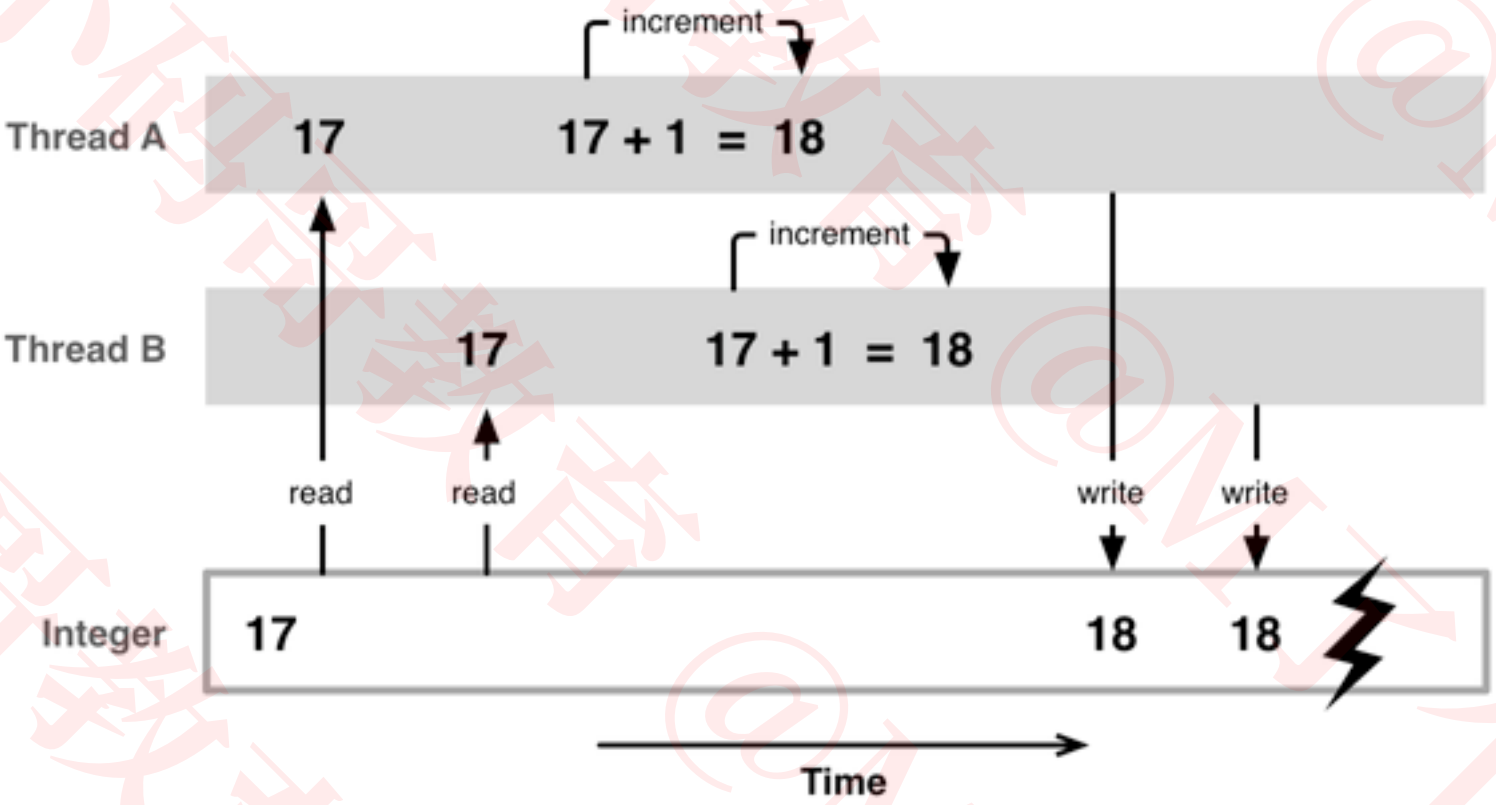
线程安全问题 – 示例

```
public class Station implements Runnable {
    private int tickets = 100;
    /**
     * 卖一张票
     * @return 是否还有票可以卖
     */
    public boolean saleTicket() {
        if (tickets < 1) return false;
        tickets--;
        String name = Thread.currentThread().getName();
        System.out.println(name + "卖了1张，剩" + tickets + "张");
        return tickets > 0;
    }
    @Override
    public void run() {
        while (saleTicket());
    }
}
```

```
Station station = new Station();
for (int i = 1; i <= 4; i++) {
    Thread thread = new Thread(station);
    thread.setName("" + i);
    thread.start();
}
```

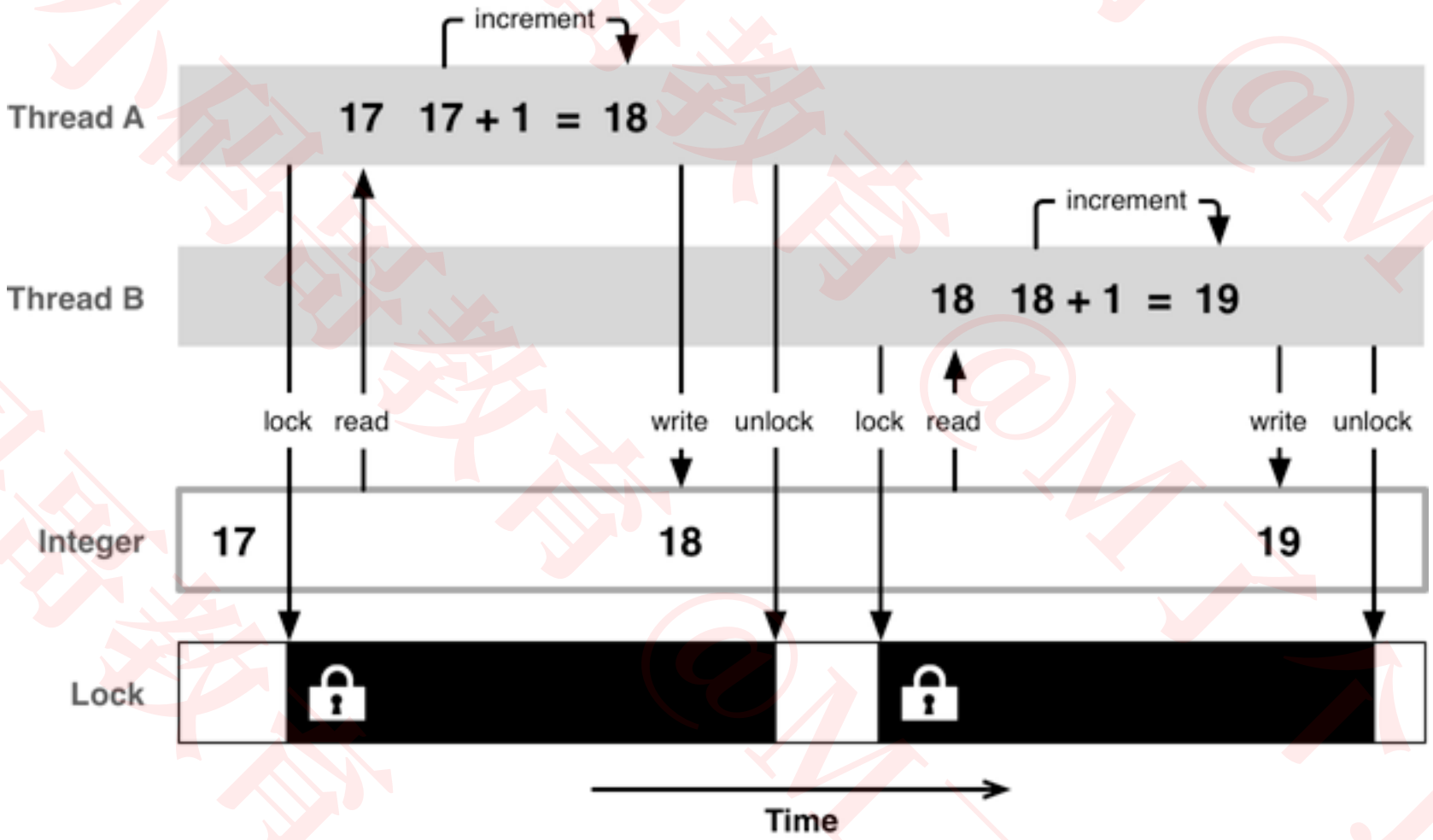
小码哥教育 SEEMYGO

线程安全问题 – 分析问题



小码哥教育 SEEMYGO

线程安全问题 – 解决方案



小码哥教育 SEEMYGO

线程同步

- 可以使用线程同步技术来解决线程安全问题
  - 同步语句（Synchronized Statement）
  - 同步方法（Synchronized Method）

小码哥教育 SEEMYGO

线程同步 - 同步语句

```
public boolean saleTicket() {
    synchronized (this) {
        if (tickets < 1) return false;
        tickets--;
        String name = Thread.currentThread().getName();
        System.out.println(name + "卖了1张，剩" + tickets + "张");
        return tickets > 0;
    }
}
```

- synchronized （obj）的原理
  - 每个对象都有一个与它相关的内部锁（intrinsic lock）或者叫监视器锁（monitor lock）
  - 第一个执行到同步语句的线程可以获得 obj 的内部锁，在执行完同步语句中的代码后释放此锁
  - 只要一个线程持有了内部锁，那么其它线程在同一时刻将无法再获得此锁
    - ✓ 当它们试图获取此锁时，将会进入 BLOCKED 状态
- 多个线程访问同一个 synchronized （obj）语句时
  - obj 必须是同一个对象，才能起到同步的作用

小码哥教育 SEEMYGO

线程同步 - 同步方法

```
public synchronized boolean saleTicket() {
    if (tickets < 1) return false;
    tickets--;
    String name = Thread.currentThread().getName();
    System.out.println(name + "卖了1张，剩" + tickets + "张");
    return tickets > 0;
}
```

- synchronized 不能修饰构造方法

- 同步方法的本质
  - 实例方法：synchronized （this）
  - 静态方法：synchronized （Class对象）

- 同步语句比同步方法更灵活一点
  - 同步语句可以精确控制需要加锁的代码范围

- 使用了线程同步技术后
  - 虽然解决了线程安全问题，但是降低了程序的执行效率
  - 所以在真正有必要的时候，才使用线程同步技术

小码哥教育 SEEMYGO

单例模式（懒汉式）改进

```
public class Rocket {
    private static Rocket instance = null;
    private Rocket() {}
    public static synchronized Rocket getInstance() {
        if (instance == null) {
            instance = new Rocket();
        }
        return instance;
    }
}
```

几个常用类的细节

- 动态数组
  - ArrayList：非线程安全
  - Vector：线程安全

- 动态字符串
  - StringBuilder：非线程安全
  - StringBuffer：线程安全

- 映射（字典）
  - HashMap：非线程安全
  - Hashtable：线程安全

小码哥教育 SEEMYGO

死锁（Deadlock）

- 什么是死锁?
  - 两个或者多个线程永远阻塞，相互等待对方的锁

```
new Thread(() -> {
    synchronized ("1") {
        System.out.println("1 - 1");
        try {
            Thread.sleep(100);
        } catch (Exception e) {
            e.printStackTrace();
        }
        synchronized ("2") {
            System.out.println("1 - 2");
        }
    }
}).start();
```

```
new Thread(() -> {
    synchronized ("2") {
        System.out.println("2 - 1");
        try {
            Thread.sleep(100);
        } catch (Exception e) {
            e.printStackTrace();
        }
        synchronized ("1") {
            System.out.println("2 - 2");
        }
    }
}).start();
```

小码哥教育 SEEMYGO

死锁 – 示例

```
class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public synchronized void hello(Person p) {
        System.out.format("[%s] hello to [%s]%n", name, p.name);
        p.smile(this);
    }
    public synchronized void smile(Person p) {
        System.out.format("[%s] smile to [%s]%n", name, p.name);
    }
}
```

```
Person jack = new Person("Jack");
Person rose = new Person("Rose");
new Thread(() -> { jack.hello(rose); }).start();
new Thread(() -> { rose.hello(jack); }).start();
```

线程间通信

- 可以使用 Object.wait、Object.notify、Object.notifyAll 方法实现线程之间的通信

- 若想在线程 A 中成功调用 obj.wait、obj.notify、obj.notifyAll 方法
  - 线程 A 必须要持有 obj 的内部锁

- obj.wait： 释放 obj 的内部锁，当前线程进入 `WAITING` 或 `TIMED_WAITING` 状态

- obj.notifyAll： 唤醒所有因为 obj.wait 进入 `WAITING` 或 `TIMED_WAITING` 状态的线程

- obj.notify： 随机唤醒 1 个因为 obj.wait 进入 `WAITING` 或 `TIMED_WAITING` 状态的线程

小码哥教育 SEEMYGO

线程间通信 – 示例 – Drop

```
public class Drop {
    private String food;
    private boolean empty = true;
}
```

```
public synchronized String get() {
    while (empty) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    empty = true;
    notifyAll();
    return food;
}
```

```
public synchronized void add(String food) {
    while (!empty) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    empty = false;
    this.food = food;
    notifyAll();
}
```

小码哥教育 SEEMYGO

线程间通信 – 示例 – Producer

```
public class Producer implements Runnable {
    private Drop drop;
    public Producer(Drop drop) {
        this.drop = drop;
    }
    public void run() {
        String foods[] = { "beef", "bread", "apple", "cookie" };
        for (int i = 0; i < foods.length; i++) {
            drop.add(foods[i]);
        }
        drop.add(null);
    }
}
```

小码哥教育 SEEMYGO

线程间通信 – 示例 – Consumer

```
public class Consumer implements Runnable {
    private Drop drop;
    public Consumer(Drop drop) {
        this.drop = drop;
    }
    public void run() {
        String food = null;
        while ((food = drop.get()) != null) {
            System.out.println("接收到食物: " + food);
        }
    }
}
```

```
Drop drop = new Drop();
new Thread(new Consumer(drop)).start();
new Thread(new Producer(drop)).start();
```

ReentrantLock（可重入锁）

- ReentrantLock ，译为“可重入锁”
  - 类的全名是：java.util.concurrent.locks.ReentrantLock
  - 具有跟同步语句、同步方法一样的一些基本功能，但功能更加强大

- 什么是可重入?
  - 同一个线程可以重复获取同一个锁
  - 其实 `synchronized` 也是可重入的

ReentrantLock – lock、tryLock

- ReentrantLock.lock ： 获取此锁
  - 如果此锁没有被另一个线程持有，则将锁的持有计数设为 1，并且此方法立即返回
  - 如果当前线程已经持有此锁，则将锁的持有计数加 1，并且此方法立即返回
  - 如果此锁被另一个线程持有，并且在获得锁之前，此线程将一直处于休眠状态，此时锁的持有计数被设为 1

- ReentrantLock.tryLock ： 仅在锁未被其他线程持有的情况下，才获取此锁
  - 如果此锁没有被另一个线程持有，则将锁的持有计数设为 1，并且此方法立即返回 true
  - 如果当前线程已经持有此锁，则将锁的持有计数加 1，并且此方法立即返回 true。
  - 如果锁被另一个线程持有，则此方法立即返回 false

ReentrantLock – unlock、isLocked

- ReentrantLock.unlock ： 尝试释放此锁
  - 如果当前线程持有此锁，则将持有计数减 1
  - 如果持有计数现在为 0，则释放此锁
  - 如果当前线程没有持有此锁，则抛出 java.lang.IllegalMonitorStateException

- ReentrantLock.isLocked ： 查看此锁是否被任意线程持有

小码哥教育 SEEMYGO

ReentrantLock 在卖票示例中的使用

```
private Lock lock = new ReentrantLock();
private int tickets = 100;
public boolean saleTicket() {
    try {
        lock.lock();
        if (tickets < 1) return false;
        tickets--;
        String name = Thread.currentThread().getName();
        System.out.println(name + "卖了1张，剩" + tickets + "张");
        return tickets > 0;
    } finally {
        lock.unlock();
    }
}
```

小码哥教育 SEEMYGO

ReentrantLock – tryLock使用注意

```
Lock lock = new ReentrantLock();
```

```
new Thread(() -> {
    try {
        lock.lock();
        System.out.println("1");
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}).start();
```

```
new Thread(() -> {
    boolean locked = false;
    try {
        locked = lock.tryLock();
        System.out.println("2");
    } finally {
        if (locked) {
            lock.unlock();
        }
    }
}).start();
```

线程池（Thread Pool）

- 线程对象占用大量内存，在大型应用程序中，频繁地创建和销毁线程对象会产生大量内存管理开销

- 使用线程池可以最大程度地减少线程创建、销毁所带来的开销

- 线程池由工作线程（Worker Thread）组成
  - 普通线程：执行完一个任务后，生命周期就结束了
  - 工作线程：可以执行多个任务（任务没来就一直等，任务来了就干活）
    - ✓ 先将任务添加到队列（Queue）中，再从队列中取出任务提交到池中

- 常用的线程池类型是固定线程池（Fixed Thread Pool）
  - 具有固定数量的正在运行的线程

小码哥教育 SEEMYGO

线程池 – 基本使用

```
// 创建拥有5条工作线程的固定线程池
ExecutorService pool = Executors.newFixedThreadPool(5);
// 执行任务
pool.execute(() -> {
    // 11_pool-1-thread-1
    System.out.println(11 + "_" + Thread.currentThread().getName());
});
pool.execute(() -> {
    // 22_pool-1-thread-2
    System.out.println(22 + "_" + Thread.currentThread().getName());
});
pool.execute(() -> {
    // 33_pool-1-thread-3
    System.out.println(33 + "_" + Thread.currentThread().getName());
});
// 关闭线程池
pool.shutdown();
```