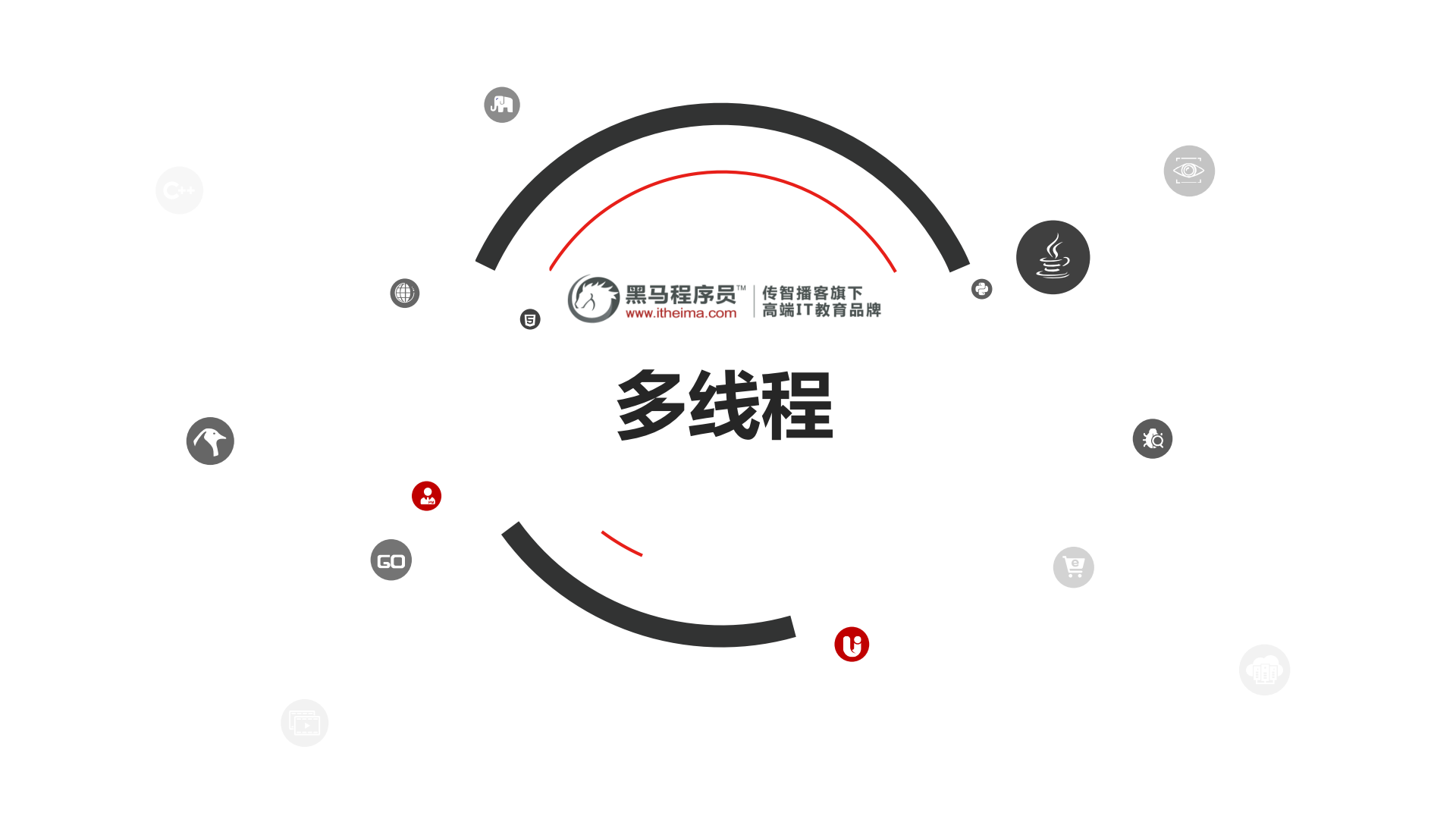

黑马程序员™
www.itheima.com
传智播客旗下
高端IT教育品牌
多线程

目录 Contents

实现多线程

简单了解多线程

是指从软件或者硬件上实现多个线程并发执行的技术。

具有多线程能力的计算机因有硬件支持而能够在同一时间执行多个线程，提升性能。

实现多线程

简单了解多线程

实现多线程

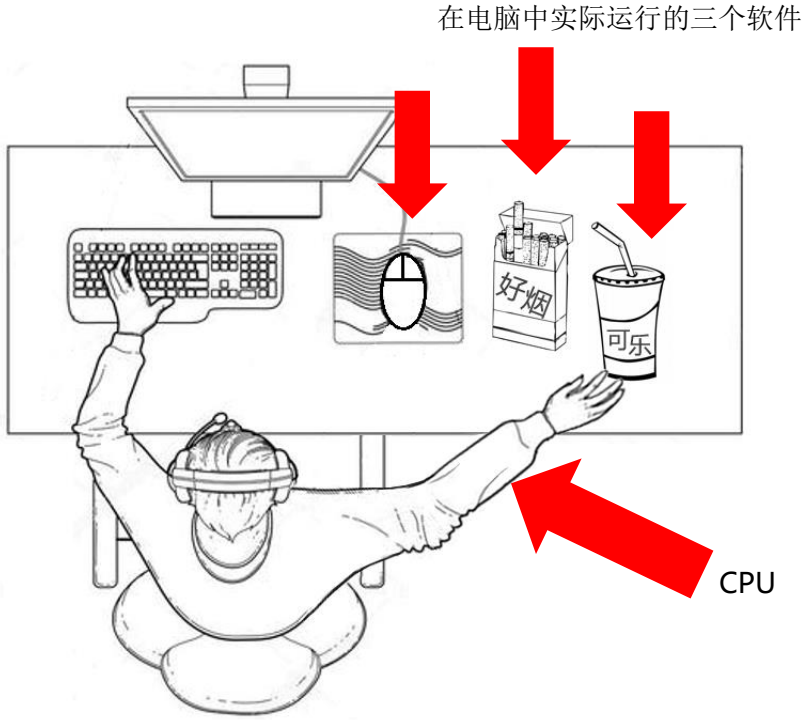
简单了解多线程

实现多线程

简单了解多线程

实现多线程

简单了解多线程



目录 Contents

实现多线程

并发和并行

- 并行：在同一时刻，有多个指令在多个CPU上**同时**执行。
- 并发：在同一时刻，有多个指令在单个CPU上**交替**执行。

实现多线程

并发和并行

并行：在同一时刻，有多个指令在多个CPU上**同时**执行。

并发和并行

并发：在同一时刻，有多个指令在单个CPU上**交替**执行。

并发和并行

并发：在同一时刻，有多个指令在单个CPU上**交替**执行。

并发和并行

并发：在同一时刻，有多个指令在单个CPU上**交替**执行。

实现多线程

进程和线程

进程：是正在运行的软件

- 独立性： 进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位。
- 动态性： 进程的实质是程序的一次执行过程，进程是动态产生，动态消亡的。
- 并发性： 任何进程都可以同其他进程一起并发执行

实现多线程

进程和线程

线程：是进程中的单个顺序控制流，是一条执行路径。

实现多线程

进程和线程

线程：是进程中的单个顺序控制流，是一条执行路径。

实现多线程

进程和线程

线程：是进程中的单个顺序控制流，是一条执行路径。

实现多线程

进程和线程

线程：是进程中的单个顺序控制流，是一条执行路径。

实现多线程

进程和线程

线程：是进程中的单个顺序控制流，是一条执行路径。

实现多线程

进程和线程

线程：是进程中的单个顺序控制流，是一条执行路径。

- 单线程：一个进程如果只有一条执行路径，则称为单线程程序
- 多线程：一个进程如果有多条执行路径，则称为多线程程序

小结

并发和并行

- 并行：在同一时刻，有多个指令在多个CPU上**同时**执行。
- 并发：在同一时刻，有多个指令在单个CPU上**交替**执行。

进程和线程

- 进程：就是操作系统中正在运行的一个应用程序。
- 线程：就是应用程序中做的事情。比如：360软件中的杀毒，扫描木马，清理垃圾。

目录 Contents

实现多线程

多线程的实现方案

- 继承Thread类的方式进行实现
- 实现Runnable接口的方式进行实现
- 利用Callable和Future接口方式实现

实现多线程

多线程的实现方案

方案1：继承Thread类

- 定义一个类MyThread继承Thread类
- 在MyThread类中重写run()方法
- 创建MyThread类的对象
- 启动线程

实现多线程

两个小问题：

- 为什么要重写run()方法?

  因为run()是用来封装被线程执行的代码

- run()方法和start()方法的区别?

  run()：封装线程执行的代码，直接调用，相当于普通方法的调用，并没有开启线程。

  start()：启动线程；然后由JVM调用此线程的run()方法

多线程的实现方式

方式2：实现Runnable接口

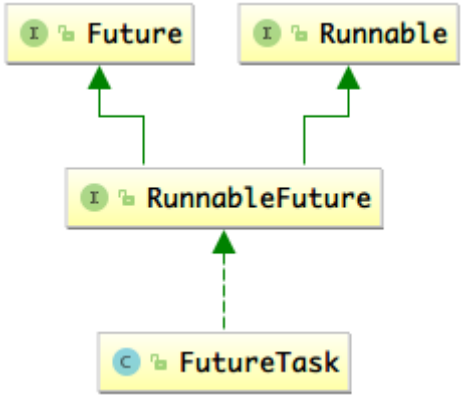
- 定义一个类MyRunnable实现Runnable接口
- 在MyRunnable类中重写run()方法
- 创建MyRunnable类的对象
- 创建Thread类的对象，把MyRunnable对象作为构造方法的参数
- 启动线程

多线程的实现方式

方式3：Callable和Future

- 定义一个类MyCallable实现Callable接口
- 在MyCallable类中重写call()方法
- 创建MyCallable类的对象
- 创建Future的实现类FutureTask对象，把MyCallable对象作为构造方法的参数
- 创建Thread类的对象，把FutureTask对象作为构造方法的参数
- 启动线程

- 再调用get方法，就可以获取线程结束之后的结果。



实现多线程

三种方式的对比

	优点	缺点
实现Runnable、Callable接口	扩展性强，实现该接口的同时还可以继承其他的类。	编程相对复杂，不能直接使用Thread类中的方法
继承Thread类	编程比较简单，可以直接使用Thread类中的方法	可以扩展性较差，不能再继承其他的类

目录 Contents

获取和设置线程名称

获取线程的名字

- String getName()：返回此线程的名称

Thread类中设置线程的名字

- void setName(String name)：将此线程的名称更改为等于参数 name
- 通过构造方法也可以设置线程名称

实现多线程

获得当前线程的对象

- public static Thread currentThread()：返回对当前正在执行的线程对象的引用

实现多线程

线程休眠

- public static void sleep(long time)：让线程休眠指定的时间，单位为毫秒。

线程调度

多线程的并发运行：

计算机中的CPU，在任意时刻只能执行一条机器指令。每个线程只有获得CPU的使用权才能执行代码。

各个线程轮流获得CPU的使用权，分别执行各自的任务。

线程调度

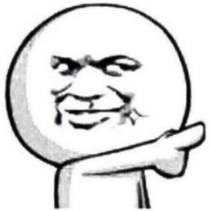
线程有两种调度模型

- 分时调度模型：所有线程**轮流**使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间片
- 抢占式调度模型：优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会**随机**选择一个，优先级高的线程获取的 CPU 时间片相对多一些

Java使用的是抢占式调度模型

实现多线程

CPU



从屏幕钻出来打我阿

顺着网线来打我啊

实现多线程

线程的优先级

- public final void setPriority(int newPriority)　　设置线程的优先级
- public final int getPriority()　　获取线程的优先级

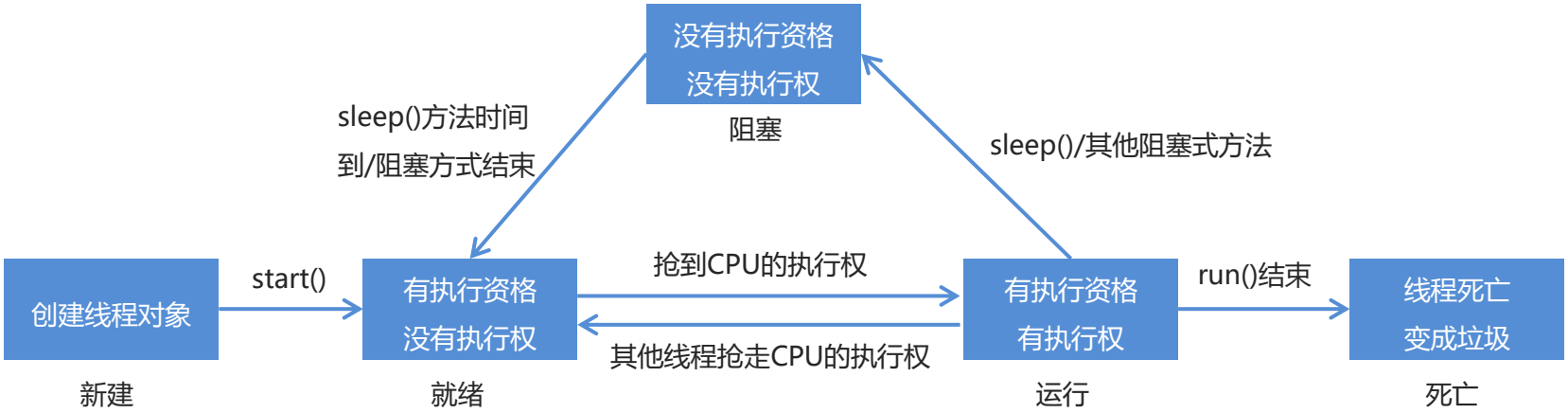
实现多线程

后台线程/守护线程

- public final void setDaemon(boolean on)：设置为守护线程

实现多线程

线程生命周期



目录 Contents

线程的安全问题

案例：卖票

需求：某电影院目前正在上映国产大片，共有100张票，而它有3个窗口卖票，请设计一个程序模拟该电影院卖票

思路：

① 定义一个类Ticket实现Runnable接口，里面定义一个成员变量：private int ticketCount = 100;

② 在Ticket类中重写run()方法实现卖票，代码步骤如下

A：判断票数大于0，就卖票，并告知是哪个窗口卖的

B：票数要减1

C：卖光之后，线程停止

③ 定义一个测试类TicketDemo，里面有main方法，代码步骤如下

A：创建Ticket类的对象

B：创建三个Thread类的对象，把Ticket对象作为构造方法的参数，并给出对应的窗口名称

C：启动线程

线程的安全问题

卖票案例的思考

刚才讲解了电影院卖票程序，好像没有什么问题。但是在实际生活中，售票时出票也是需要时间的，所以，在出售一张票的时候，需要一点时间的延迟，接下来我们去修改卖票程序中卖票的动作：
每次出票时间100毫秒，用sleep()方法实现

卖票出现了问题

- 相同的票出现了多次
- 出现了负数的票

问题原因：

- 线程执行的随机性导致的

卖票案例数据安全问题的解决

为什么出现问题?(这也是我们判断多线程程序是否会有数据安全问题的标准)

- 多线程操作共享数据

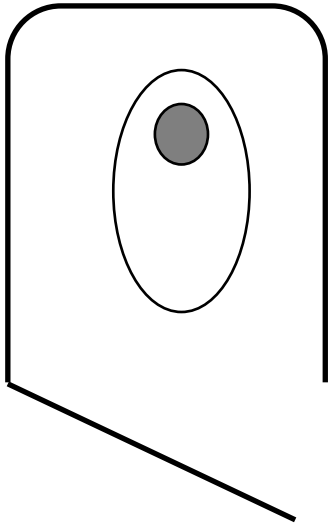
如何解决多线程安全问题呢?

- 基本思想：让程序没有安全问题的环境

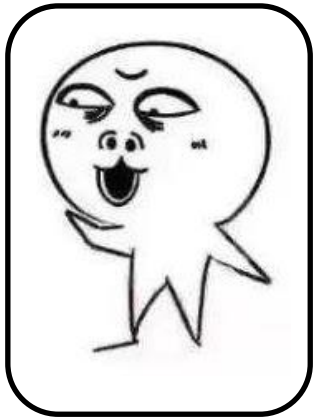
怎么实现呢?

- 把多条语句操作共享数据的代码给锁起来，让任意时刻只能有一个线程执行即可
- Java提供了同步代码块的方式来解决

线程的安全问题



线程的安全问题



线程的安全问题

线程的安全问题

线程的安全问题

同步代码块

锁多条语句操作共享数据，可以使用同步代码块实现

- 格式：

```
synchronized(任意对象) {
        多条语句操作共享数据的代码
}
```

- 默认情况是打开的，只要有一个线程进去执行代码了，锁就会关闭
- 当线程执行完出来了，锁才会自动打开

同步的好处和弊端

- 好处：解决了多线程的数据安全问题
- 弊端：当线程很多时，因为每个线程都会去判断同步上的锁，这是很耗费资源的，无形中会降低程序的运行效率

线程的安全问题

同步方法

同步方法：就是把synchronized关键字加到方法上

- 格式：

  修饰符 **synchronized** 返回值类型 方法名(方法参数) {　}

同步代码块和同步方法的区别:

- 同步代码块可以锁住指定代码,同步方法是锁住方法中所有代码
- 同步代码块可以指定锁对象,同步方法不能指定锁对象

同步方法的锁对象是什么呢?

- **this**

线程的安全问题

同步方法

同步静态方法：就是把synchronized关键字加到静态方法上

- 格式：

  修饰符 static **synchronized** 返回值类型 方法名(方法参数) {　}

同步静态方法的锁对象是什么呢?

- **类名.class**

线程的安全问题

Lock锁

虽然我们可以理解同步代码块和同步方法的锁对象问题，但是我们并没有直接看到在哪里加上了锁，在哪里释放了锁，为了更清晰的表达如何加锁和释放锁，JDK5以后提供了一个新的锁对象Lock

Lock实现提供比使用synchronized方法和语句可以获得更广泛的锁定操作
Lock中提供了获得锁和释放锁的方法

- void lock()：获得锁
- void unlock()：释放锁

Lock是接口不能直接实例化，这里采用它的实现类ReentrantLock来实例化
ReentrantLock的构造方法

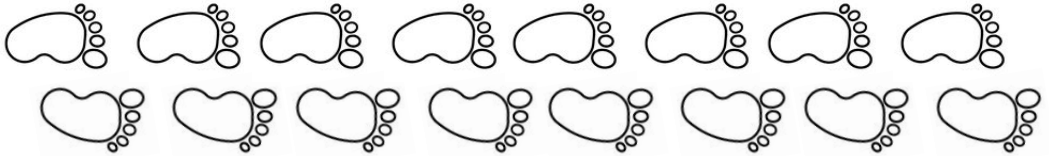
- ReentrantLock()：创建一个ReentrantLock的实例

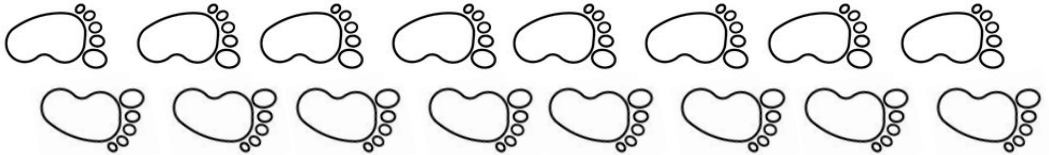
目录 Contents

线程的安全问题

死锁

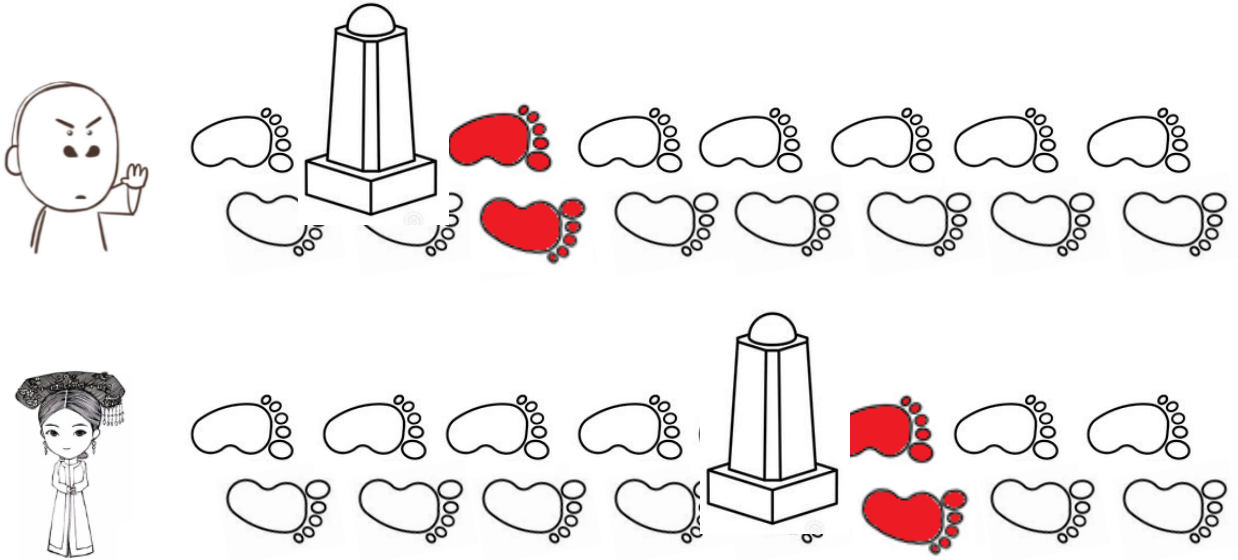
线程死锁是指由于两个或者多个线程互相持有对方所需要的资源，导致这些线程处于等待状态，无法前往执行。





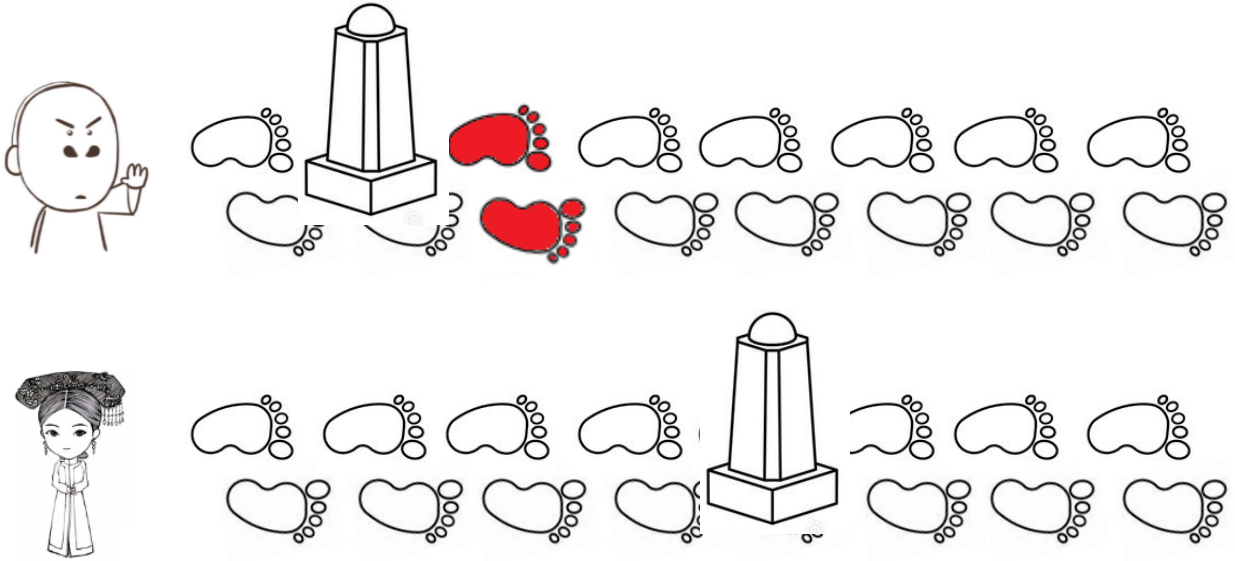
线程的安全问题

死锁



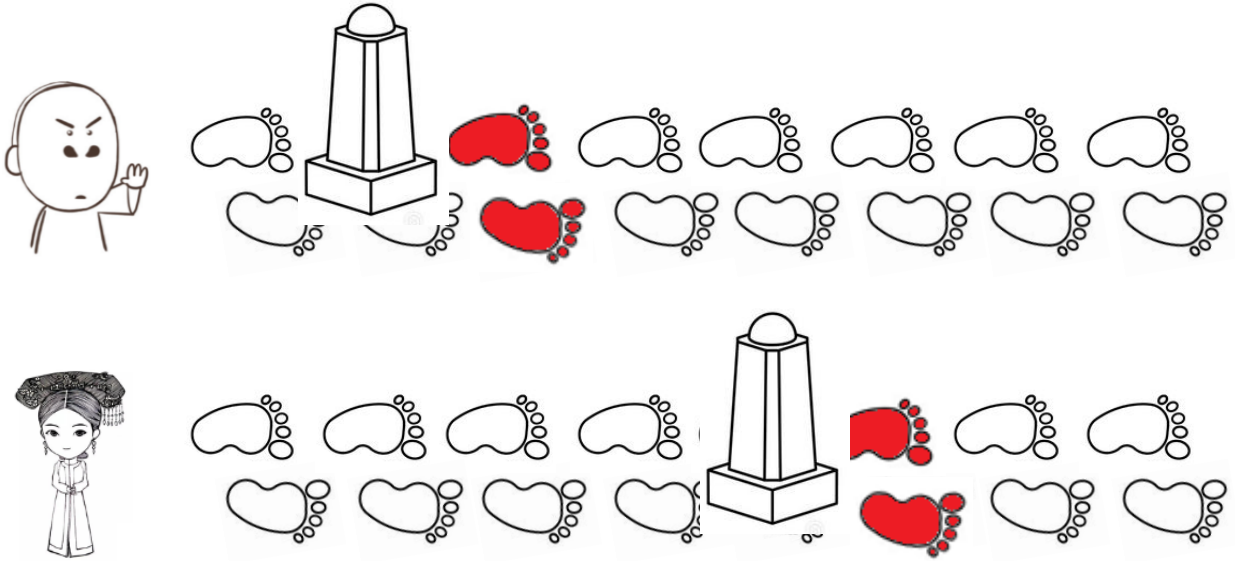
线程的安全问题

死锁



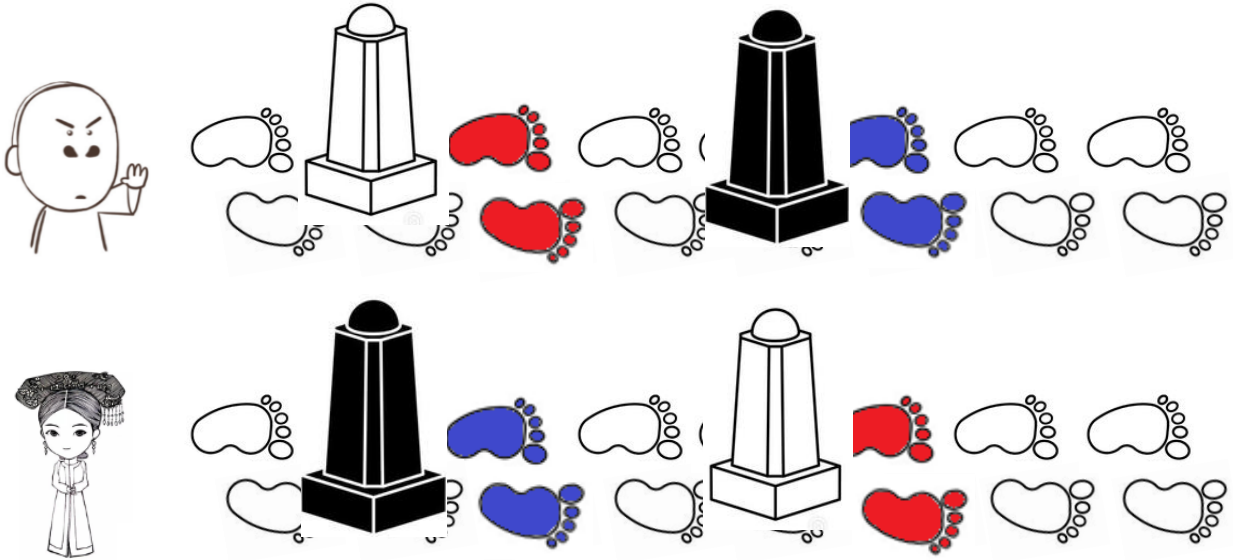
线程的安全问题

死锁



线程的安全问题

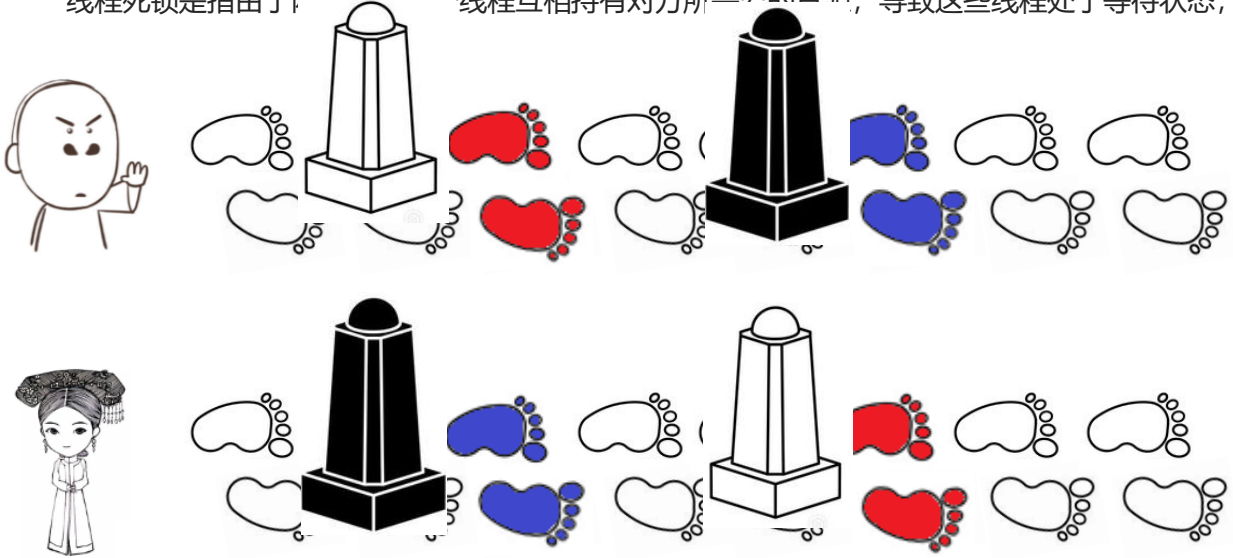
死锁



线程的安全问题

死锁(需要构思回程关闭机关)

线程死锁是指由于两个或者多个线程互相持有对方所需要的资源，导致这些线程处于等待状态，无法前往执行。



目录 Contents

生产者消费者

生产者消费者模式概述

生产者消费者模式是一个十分经典的多线程协作的模式，弄懂生产者消费者问题能够让我们对多线程编程的理解更加深刻

吃货线程

桌子上的食物用来控制线程

厨师线程

生产者消费者

生产者消费者理想情况

抢到执行权

抢到执行权

生产者消费者

生产者消费者理想情况

抢到执行权

抢到执行权

生产者消费者

消费者等待过程

抢到执行权

大兄弟，做好 了。
来吃吧。

等待中。。。

抢到执行权

生产者消费者

消费者等待步骤

抢到执行权

消费者步骤：

1，判断桌子上是否有汉堡包。

2，如果没有就等待。

等待中。。。

抢到执行权

生产者步骤：

1，生产汉堡包。

2，把汉堡包放在桌子上。

3，叫醒等待的消费者开吃。

生产者消费者

生产者等待

消费者步骤：
1，判断桌子上是否有汉堡包。
2，如果没有就等待。

生产者步骤：
1，生产汉堡包。
2，把汉堡包放在桌子上。
3，叫醒等待的消费者开吃。

生产者消费者

生产者等待

消费者步骤：
1，判断桌子上是否有汉堡包。
2，如果没有就等待。

等待中。。。

生产者步骤：
1，生产汉堡包。
2，把汉堡包放在桌子上。
3，叫醒等待的消费者开吃。

生产者消费者

生产者等待

消费者步骤：
1，判断桌子上是否有汉堡包。
2，如果没有就等待。

生产者步骤：
1，判断桌子上是否有汉堡包
如果有就等待，如果没有才生产。
2，把汉堡包放在桌子上。
3，叫醒等待的消费者开吃。

生产者消费者

生产者等待

抢到执行权

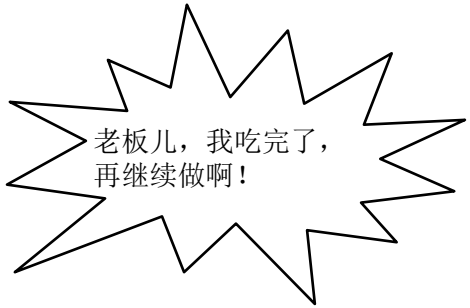
消费者步骤：
1，判断桌子上是否有汉堡包。
2，如果没有就等待。
3，如果有就开吃

等待中。。。

生产者步骤：
1，判断桌子上是否有汉堡包
如果有就等待，如果没有才生产。
2，把汉堡包放在桌子上。
3，叫醒等待的消费者开吃。

生产者消费者

生产者等待

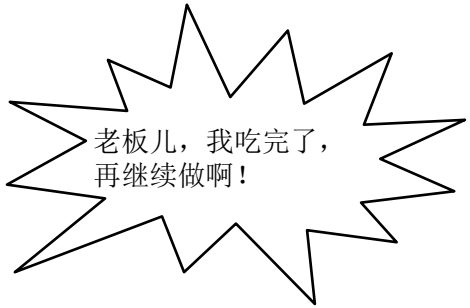


消费者步骤：
1，判断桌子上是否有汉堡包。
2，如果没有就等待。
3，如果有就开吃
4，**吃完之后，桌子上的汉堡包就没有了**
叫醒等待的生产者继续生产
汉堡包的总数量减一

生产者步骤：
1，判断桌子上是否有汉堡包
如果有就等待，如果没有才生产。
2，把汉堡包放在桌子上。
3，叫醒等待的消费者开吃。

生产者消费者

小结



消费者步骤：
1，判断桌子上是否有汉堡包。
2，如果没有就等待。
3，如果有就开吃
4，吃完之后，桌子上的汉堡包就没有了
叫醒等待的生产者继续生产
汉堡包的总数量减一

生产者步骤：
1，判断桌子上是否有汉堡包
如果有就等待，如果没有才生产。
2，把汉堡包放在桌子上。
3，叫醒等待的消费者开吃。

生产者消费者

等待和唤醒的方法

为了体现生产和消费过程中的等待和唤醒，Java就提供了几个方法供我们使用，这几个方法在Object类中

Object类的等待和唤醒方法：

方法名	说明
void wait()	导致当前线程等待，直到另一个线程调用该对象的 notify()方法或 notifyAll()方法
void notify()	唤醒正在等待对象监视器的单个线程
void notifyAll()	唤醒正在等待对象监视器的所有线程

生产者消费者

阻塞队列实现等待唤醒机制

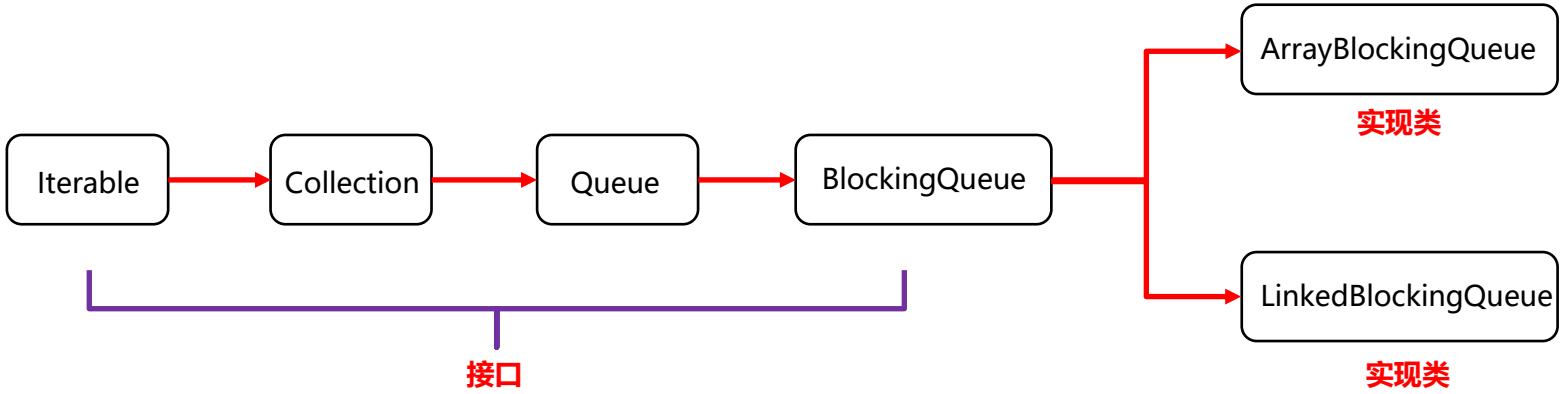
take方法

阻塞队列

put方法

生产者消费者

阻塞队列继承结构



生产者消费者

阻塞队列实现等待唤醒机制

BlockingQueue的核心方法：

put(anObject):将参数放入队列，如果放不进去会阻塞。

take():取出第一个数据，取不到会阻塞。

阻塞队列实现等待唤醒机制

常见BlockingQueue：

ArrayBlockingQueue：底层是数组，有界。

LinkedBlockingQueue：底层是链表，无界。但不是真正的无界，最大为int的最大值。

传智播客旗下高端IT教育品牌