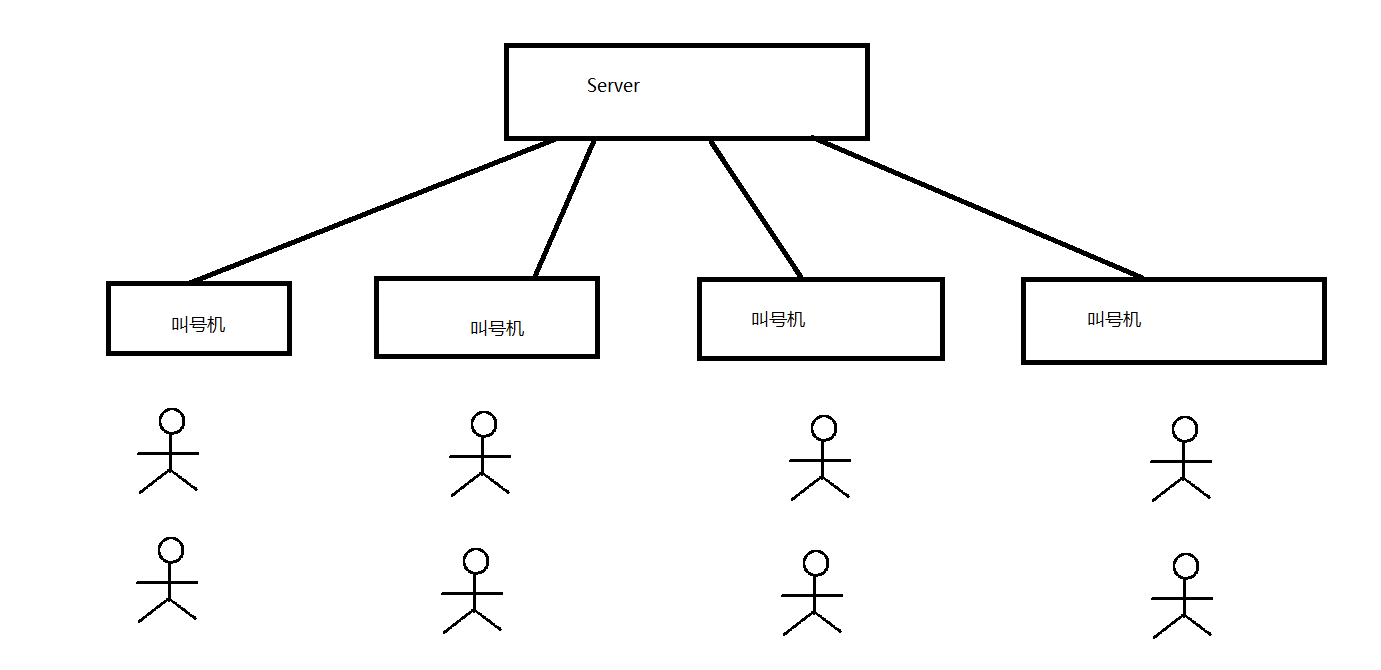
# synchronized



**并发叫号的程序**：

使用 static 解决并发问题

并发量比较大的时候会出现：跳号、重号、超过最大值

package com.luban;

import java.sql.SQLOutput;  
  
public class TicketDemo extends Thread{  
  
 // 叫到的号码牌  
 /\*  
 \* private int index = 1; 私有的变量，每一个便变量都有自己的index  
 \* private static int index = 1; 共享变量  
 \*/  
 private static int *index* = 1;  
  
 // 最多处理人数: 营业厅内最多只能处理50人  
 private static final int *MAX* = 50;  
  
 @Override  
 public void run() {  
  
 */\*\*  
 \* 可能出现两种情况：跳号 和 重号  
 \* i = 50; i+1; System.out.println(i);  
 \* 1.重号：  
 \* 线程1 从内存中获取i的到工作区间，执行加1操作，没来得及写入到内存中，  
 \* 线程2 就进行了同样的操作，并且写入到内存中，此时线程2输出结果是51，线程1输出结果也是51  
 \* 2.跳号：  
 \* 线程1 从内存中获取i的到工作区间，执行加以操作，写入到内存中，  
 \* 没有来得及执行写出操作到控制台，线程2就进行了同样的操作，此时线程2输出结果是52，线程1输出结果也是52  
 \*/*// while(index <= MAX){  
// System.out.println(Thread.currentThread().getName() + "叫到的号码是：" + (index++));  
// }  
  
 */\*\*  
 \* 使用sychronize封锁当前对象,解决上述出现的两种问题（重号和跳号）  
 \* 将 sychronize(this){ 代码块 } 中的代码块作为一个整体：  
 \* 1. 具体有原子性，不能只执行其中一部分代码，要么全执行  
 \* 2. 具有排他性，有一条线程在执行代码块时，即使其他的线程获取了CPU时间片，执行到该处就不能执行了，必须等上次的线程执行完成。  
 \*/* synchronized (this) {  
 while(*index* <= *MAX*){  
 System.*out*.println(Thread.*currentThread*().getName() + "叫到的号码是：" + (*index*++));  
 }  
 }  
  
 }  
  
 // main方法缩写方法： psvm  
 public static void main(String[] args) {  
 // 基于线程创建  
 TicketDemo t1 = new TicketDemo();  
 TicketDemo t2 = new TicketDemo();  
 TicketDemo t3 = new TicketDemo();  
 TicketDemo t4 = new TicketDemo();  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 }  
}

## 一、概念

是利用锁的机制来实现同步的。

锁机制有如下两种特性：

**互斥性：**即在同一时间只允许一个线程持有某个对象锁，通过这种特性来实现多线程中的协调机制，这样在同一时间只有一个线程对需同步的代码块(复合操作)进行访问。互斥性我们也往往称为操作的原子性。

**可见性：**必须确保在锁被释放之前，对共享变量所做的修改，对于随后获得该锁的另一个线程是可见的（即在获得锁时应获得最新共享变量的值），否则另一个线程可能是在本地缓存的某个副本上继续操作从而引起不一致。

## 二、synchronized的用法

**根据修饰对象分类**

1. 同步方法
   1. 同步非静态方法

Public synchronized void methodName(){

……

}

* 1. 同步静态方法

Public synchronized static void methodName(){

……



**synchronized(this|object) {}**

**synchronized(类.class) {}**

Private final Object MUTEX =new Object();

Public void methodName(){

Synchronized(MUTEX ){

……

}

}

根据获取的锁分类

1. 获取对象锁

synchronized(this|object) {}

修饰非静态方法

在 Java 中，每个对象都会有一个 monitor 对象，这个对象其实就是 Java 对象的锁，通常会被称为“内置锁”或“对象锁”。类的对象可以有多个，所以每个对象有其独立的对象锁，互不干扰。

1. 获取类锁

synchronized(类.class) {}

修饰静态方法

在 Java 中，针对每个类也有一个锁，可以称为“类锁”，类锁实际上是通过对象锁实现的，即类的 Class 对象锁。每个类只有一个 Class 对象，所以每个类只有一个类锁。

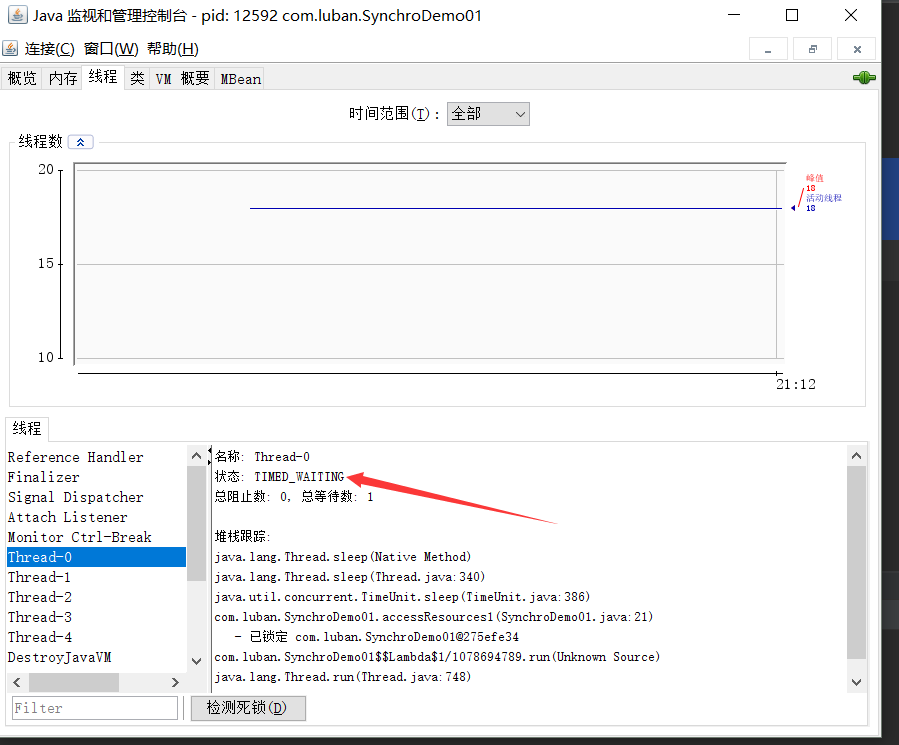
在 Java 中，每个对象都会有一个 monitor 对象，监视器。

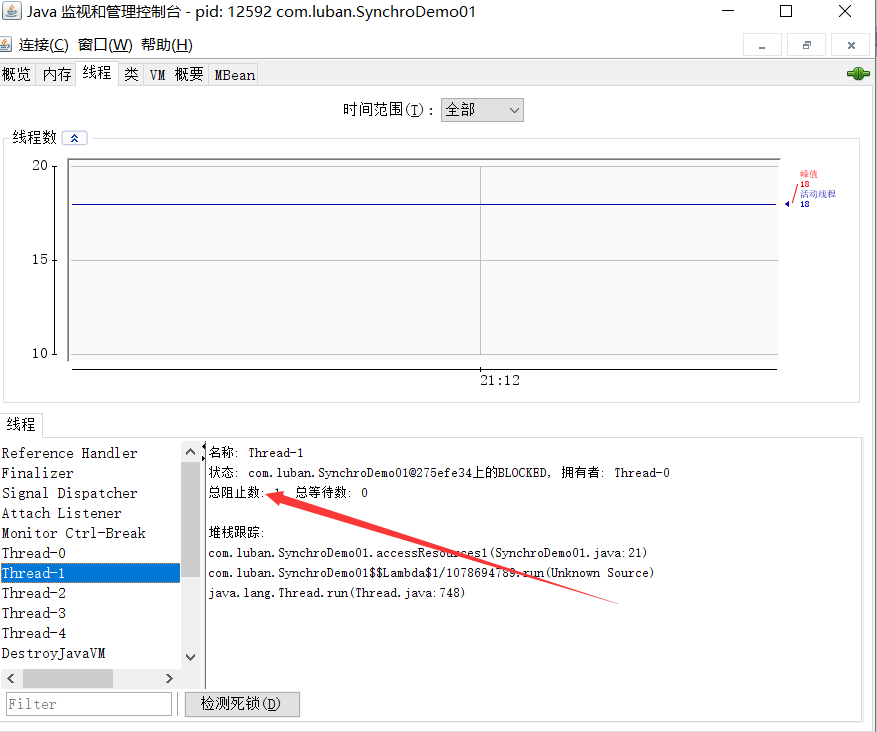
1. 某一线程占有这个对象的时候，先看monitor 的计数器是不是0，如果是0表示还没有线程占有该对象，这个时候此线程占有这个对象，并且对这个对象的monitor+1；如果不为0，表示这个对象已经被其他线程占有，这个线程等待。当线程释放占有权的时候，monitor-1；
2. 同一线程可以对同一对象进行多次加锁，+1，+1，重入性

## synchronized原理分析

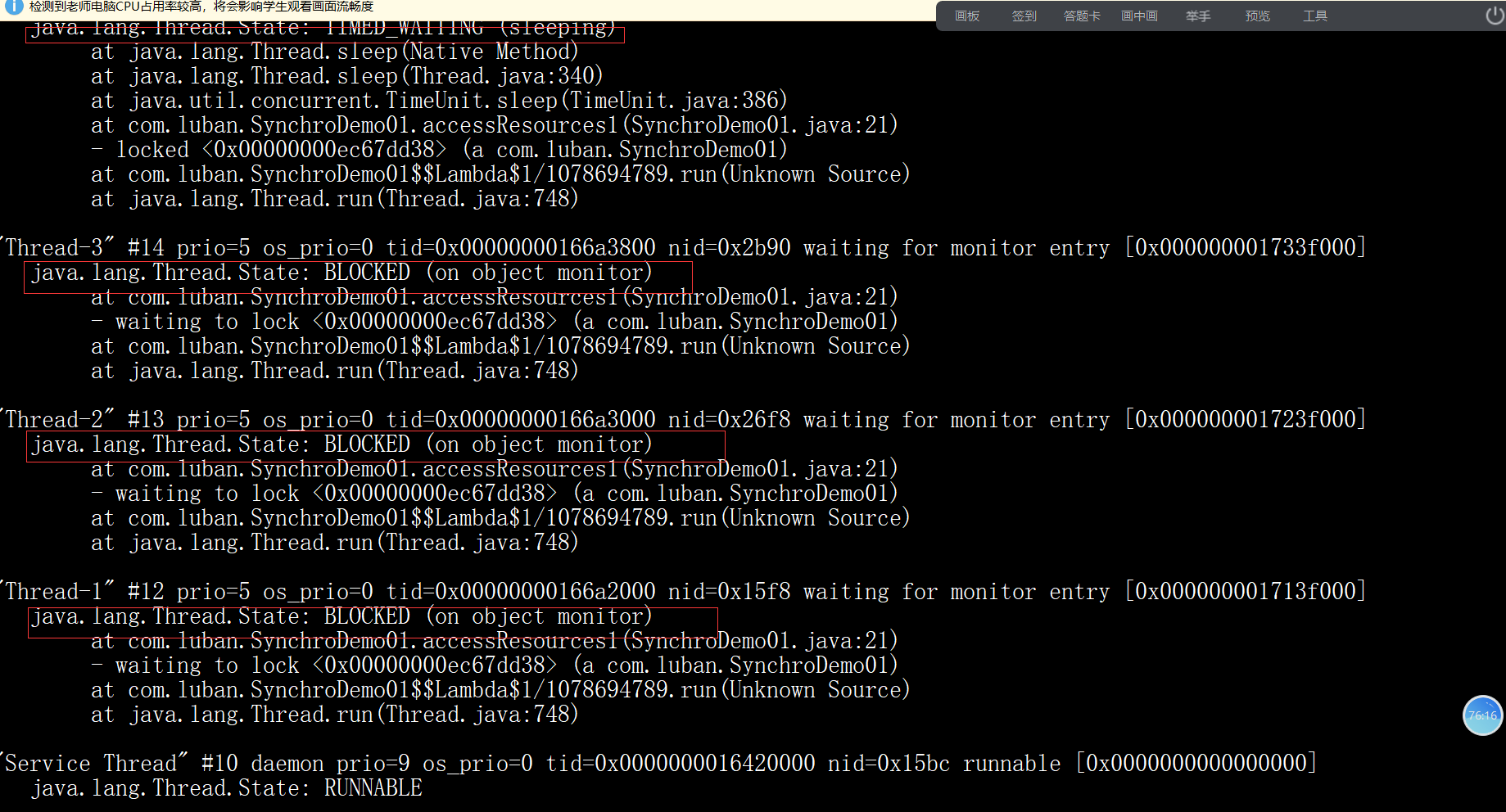
1. 线程堆栈分析（互斥）

Jconsole





Jstack pid

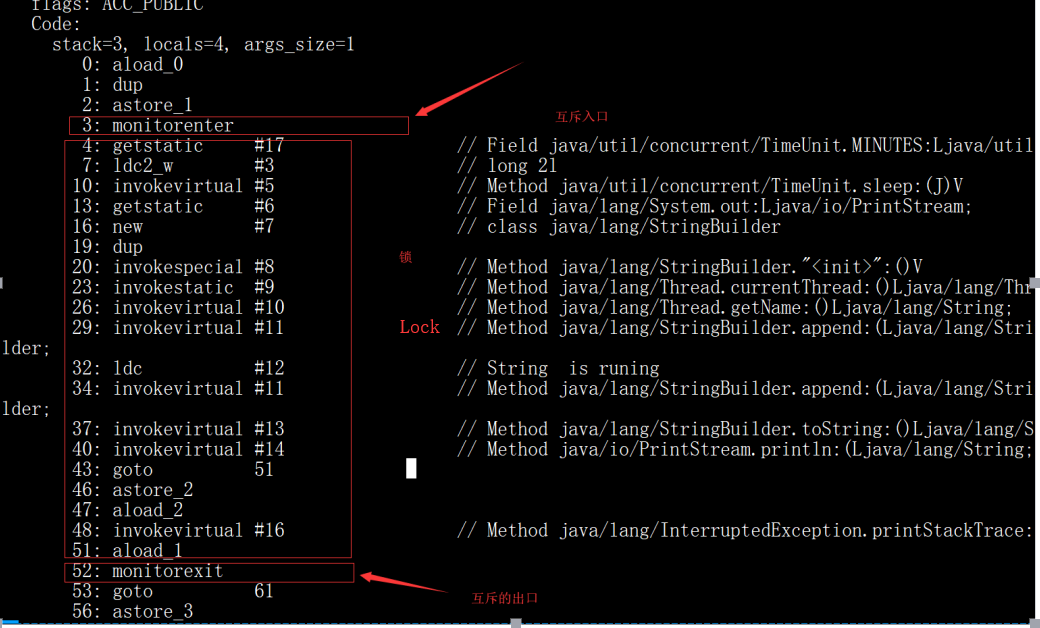


1. JVM指令分析

Javap -V 反编译

Monitorenter (互斥入口)

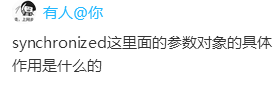
Monitorexit (互斥出口)



Getstatic

astore\_n

aload\_n



Monitor

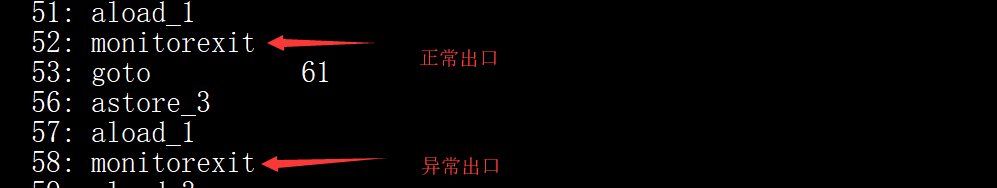
(1)0,lock

1. 重入
2. monitor一个线程占有，其他线程请求时会进入BOLCK,直到monitor为0

Monitorexit

计数器减一，为0时为解锁

Javap -v



以上是代码块的加锁monitorenter和monitorExit配合使用

对方法的加锁（同步方法、互斥方法）

ACC\_SYNCHRONIZED



1.6版本的JDK以前只有重量锁

ACC\_SYNCHRONIZED

1. 使用synchronized注意的问题
2. 与moniter关联的对象不能为空
3. synchronized作用域太大
4. 不同的monitor企图锁相同的方法
5. 多个锁的交叉导致死锁

## Java虚拟机对synchronized的优化

偏向锁

轻量级锁

重量级锁（等待时间长）

对象头与monitor

一个对象实例包含：对象头、实例变量、填充数据



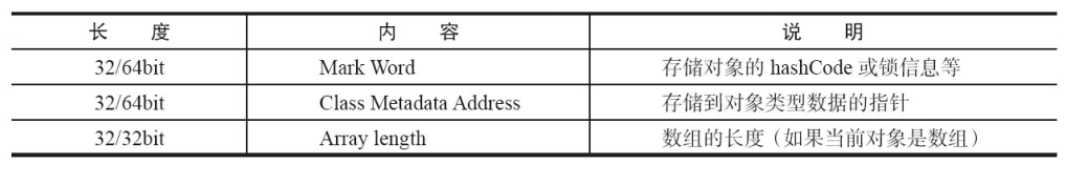
对象头：加锁的基础

实例变量：属性变量的信息

填充数据：

2个字：

hashCode的作用：HashSet





无锁状态:没有加锁

偏向锁：在对象第一次被某一线程占有的时候，是否偏向锁置1，锁表志位置01，写入线程号，当其他的线程访问的时候，竞争。竞争失败，锁会升级为轻量级锁

很多次被第一次占有它的线程获取次数多，竞争成功

CAS算法 compare and set（CAS）

无锁状态时间非常接近

竞争不激烈的时候适用

轻量级锁：线程有交替适用，互斥性不是很强，CAS失败，00

重量级锁：强互斥，10，等待时间长

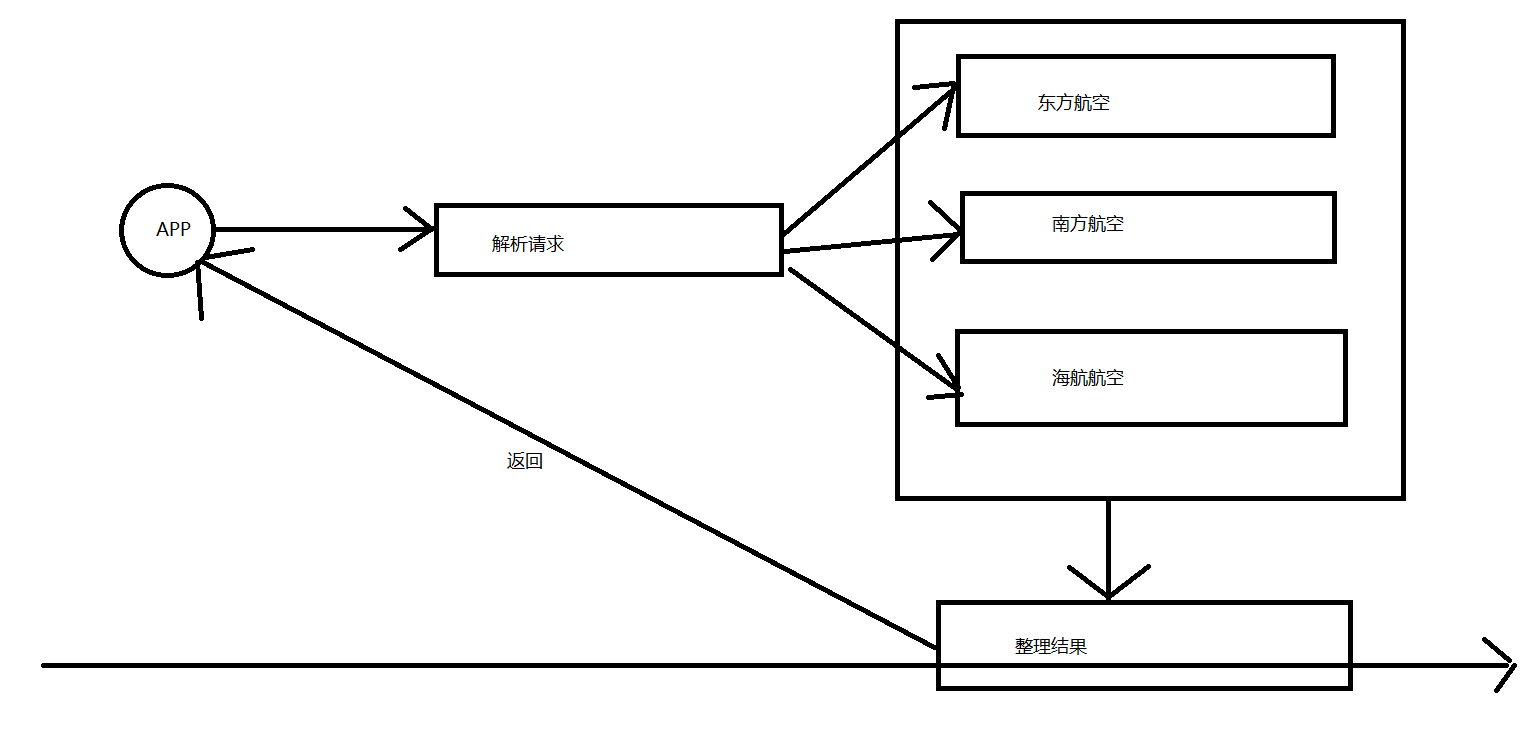
自旋锁：竞争失败的时候，不是马上转化级别，而是执行几次空循环5 10

锁消除：JIT在编译的时候把不必要的锁去掉

package com.luban;

import com.sun.org.apache.xerces.internal.parsers.CachingParserPool;  
  
import java.util.concurrent.TimeUnit;  
  
public class SynchronizeDemo {  
  
 // 1、同步方法  
 // 1.1 修饰静态方法  
 public synchronized static void accessResources0(){  
 try {  
 TimeUnit.*SECONDS*.sleep(2);  
 System.*out*.println(Thread.*currentThread*().getName() + " is running");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 // 1.2 非静态方法  
 public synchronized void accessResources1(){  
 try {  
 TimeUnit.*MINUTES*.sleep(2);  
 System.*out*.println(Thread.*currentThread*().getName() + " is running");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 // 2、同步代码块  
 // 2.1 代码块 (对象)  
 public void accessResources3(){  
 synchronized(this){ // this指的是当前对象  
 try {  
 TimeUnit.*SECONDS*.sleep(2);  
 System.*out*.println(Thread.*currentThread*().getName() + " is running");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 // 2.2 代码块 (CLASS类)  
 public void accessResources4(){  
 */\*\*  
 \* 使用ClassLoader(类加载器)加载类(SynchrenizeDemo)到方法区时，会生成了一个Class对象到堆中  
 \* Class对象的所有对象都公用一个锁  
 \*/* synchronized(SynchronizeDemo.class){  
 try {  
 TimeUnit.*SECONDS*.sleep(2);  
 System.*out*.println(Thread.*currentThread*().getName() + " is running");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
  
 public static void main(String[] args) {  
  
// for(int i=0; i<5; i++) {  
// new Thread(SynchronizeDemo::accessResources0).start();  
// }  
  
 final SynchronizeDemo demo1 = new SynchronizeDemo();  
 for(int i=0; i<5; i++) {  
 new Thread(demo1::accessResources1).start();  
 }  
  
 }  
}

作业：模式购买飞机票的程序（北京到上海）



Java的层次：

1. java语法 和 面向对象的思想
2. JVM
3. JMM
4. 并发