多线程中的同步机制（线程安全）

1. **多线程的安全隐患**：

当多个线程是在**操作共享的数据**，且操作共享数据的线程代码有**多条语句**时，当其中一个线程在处理共享数据的多条代码过程中，其他线程参与了运算，就会导致**多线程安全问题的出现**。如果只有一句代码操作共享数据，就不会出现**多线程安全问题**。

1. **安全问题的解决方法**：将**操作共享数据的多条代码**封装起来，当有线程执行这块代码时，其他的线程**不准执行这块代码**，只有当此线程把这块代码都执行完毕后，其他线程才可以执行这块代码。在java中利用**同步**就可以实现上述需求。
2. **同步的前提**：同步中必须有多个线程使用**同一个锁（同步锁）**，此同步锁（对象）必须定义在run方法之外，否则就是一个线程一把锁了，没有任何意义。
3. **同步代码块**的定义格式：

**Synchronized（任意的一个对象）**

{

**需要被同步的代码块**

}

**其中的任意对象**称为**同步锁**；作用：**对同步进行监视**。

1. **同步方法**：

**（1）同步方法**：

如果把同步代码块封装成一个方法，此方法利用**synchronized关键字**修饰，则此方法称之为**同步方法**。同步方法默认的同步锁为**this所代表的对象**。同步方法的定义格式：

权限修饰符 **synchronized** 返回类型 方法名（）{

方法体

}//把**需要同步的代码块**封装成同步方法。

**（2）静态同步方法**：

如果同步方法是**静态的**，但是由于静态方法是不依赖于具体对象的，所以静态方法中不能使用this，此时静态同步方法的同步锁为**此类对应的字节码文件对象**，此对象可以通过两种方式获取:（1）**类名.class** ；（2） 通过该类的非静态方法getClass（）依赖于具体对象获取，例如**this.getClass（）**。建议使用**类名.class。任何一个类都具有class属性，通过类名.class**就可以获取此类的字节码文件对象**。**

1. **同步方法、静态同步方法与同步代码块的区别**：
   1. **同步方法的同步锁**是固定的**this所代表的对象**；**静态同步方法的同步锁**是**固定的此类的字节码文件对象**，而同步代码块的同步锁可以任意指定（任意对象），**也可以利用this或类名.class**。
   2. 如果想让同步方法和同步代码块使用相同的同步锁，那么**同步代码块必须使用this作为同步锁**。
   3. 如果想让静态同步方法和同步代码块使用**相同的同步锁**，那么**同步代码块必须使用此类的字节码文件对象作为同步锁，即一般写成类名.class**。
   4. 在实际开发中，**建议使用同步代码块**，因为可以**任意指定对象作为同步锁**，同步方法可以看做是**同步代码块的简写形式**。
2. 同步的利弊：
3. 同步的好处：**解决了多线程的安全问题**；（线程安全）。
4. 同步的弊端：降低了效率，因为同步外的线程**都会判断同步锁**，有时是一种没有必要的判断。有时可以利用**双重判断的方式**即同步锁内外都设置判断来尽量避免对同步锁的判断。如单例模式中的应用。
5. **多线程内存解析**：

不同的线程**有独自的内存区**，每个线程**有自己独自的路径**，相互不影响。只有当所有的线程都结束了，**JVM即Java虚拟机**才会结束。**任何一个线程发生异常与其他的线程没有影响。**

1. 面试题：为什么引入同步呢？

答：多个线程访问同一数据资源，可能会造成线程不安全问题。引入同步的目的就是解决多线程访问同一数据资源的安全性问题。

1. 示例1： **多窗口售票**：
2. 分析：**车票**本身就是一个对象，而多个窗口操作的必须是**同一个对象**，需要把**售车票**封装成一个**线程任务**。故**创建线程需要使用实现Runnable接口的方法**。（利用**继承Thread方法**是不太好，因为创建一个线程类就是一个新的任务，并不能实现多线程完成同一任务的目的，但是共享数据如果是静态的，也可以实现共享，这种方式不建议使用）

**多线程完成同一任务，需要操作共享数据时，就要考虑多线程安全问题**。多窗口售票，票号是共享的，因此**必须使用同步锁**。

代码如下：

**同步代码块方式：public class Tickets implements Runnable** {

int num;//票数

public Tickets(int num) {

super();

this.num = num;

}

Object obj = new Object();

public void run() {

while (true) {

synchronized (obj) {

if (num > 0)

System.out.println(Thread.currentThread().getName() + "...."+ this.num--);

else

break;

}}}}

**同步方法方式**：

public class Tickets implements Runnable {

private int num;

public Tickets(int num) {

super();

this.num = num;

}

boolean flag = true;

public void run() {

while(flag)

shouTickets();

}

**private synchronized void shouTickets() {**

if (num > 0)

System.out.println(Thread.currentThread().getName() + "...."+ this.num--);

else

flag = false;

}}

**主函数中**：

Tickets t1 = **new** Tickets(100);//创建任务

Thread th1 = **new** Thread(t1, "窗口1");

Thread th2 = **new** Thread(t1, "窗口2");

Thread th3 = **new** Thread(t1, "窗口3");

Thread th4 = **new** Thread(t1, "窗口4");

**th1.start();th2.start();th3.start();th4.start();**

1. 示例2：**单例设计模式与同步安全**：

单例设计模式分为两种：（1）饿汉式，即**单例模式**；（2）懒汉式，又称为**延迟加载的单例模式**。

创建步骤：

1. 构造器私有化，避免外界直接创建对象；
2. 声明一个私有的静态变量：饿汉式：直接new出一个对象；懒汉式为null；
3. 创建一个公共的静态的方法getInstance，饿汉式直接return 对象；懒汉式，判断是否为null，如果为null，就new一个，最后返回对象。
4. 饿汉式：

public class Student {

**private static final Student s = new Student();//静态常量**

private Student() **{//构造方法私有化，禁止外界创建对象**

super();

}

**public static Student getInstance() {**

**return s;**

**}** }//由于操作s对象只有一句代码，return s；**故不存在同步安全问题**。

1. 懒汉式：（**延迟加载的单例模式）面试的时候经常考到这种情况**

**public** **class** Student {

**private** Student() {

**super**();

}

**private** **static** Student *student* = **null**;

public static Student getInstance() {

if(*student* == null) {

try {

Thread.*sleep*(1000);

} catch (InterruptedException e) {

}

*student* = new Student();

}

**return** *student*; **//因为操作s有多条语句，故存在多线程安全问题**

}

}

**public** **static** **void** main(String[] args) {

Thread th1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

Student student = Student.*getInstance*();

System.***out***.println(student);

}

});

Thread th2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

Student student = Student.*getInstance*();

System.***out***.println(student);

}

});

th1.start();

th2.start();

}

结果：

**并非单例，出现了两个对象，也就是并发问题。**

简化：

public class Student {

**private static Student s = null;//不能用final修饰，为一个静态常量**

private Student() {

super();

}

**public static Student getInstance() {**

**if(s == null) {**

**s = new Student();**

**}**

**return s;**

**}**

}

解决办法：1是利用**同步代码块**，2是利用**静态同步方法**。

* 1. 静态同步代码块：这样效率更高，建议使用这种方式。**双重判断或叫做双重检查**。

**第一次检查：目的是提高效率，避免过多的等待；**

**第二次检查：目的是保证线程安全。**

public static Student getInstance() {

if (s == null) {//**作用是提高效率，以免过多的判断同步锁**

**synchronized (Student.class) {//利用同步锁是该类的Class对象**

**if (s == null)//安全保障**

**s = new Student();**

**}**

}

return s;

}

* 1. **静态同步方法：**

**public static synchronized Student getInstance() {//利用的同步锁为该类的Class对象**

if (s == null)

s = new Student();

return s;

}

针对**单例设计模式的多线程安全问题**，通常会有一些面试题：

**面试问题**：（1） 懒汉式中的**静态同步方法**的同步锁是什么？

答：**类的字节码文件对象**。

（2）饿汉式和懒汉式那个效率会更高些？

答：饿汉式会更好，因为只要s赋予对象之后，就**不会进行同步锁的判断**，不会再进入**同步代码块**中；然而**懒汉式同步方法**，每次都要进行同步锁的判断，而且如果是**同一个同步锁**，就会进入同步方法中，进而使效率降低。

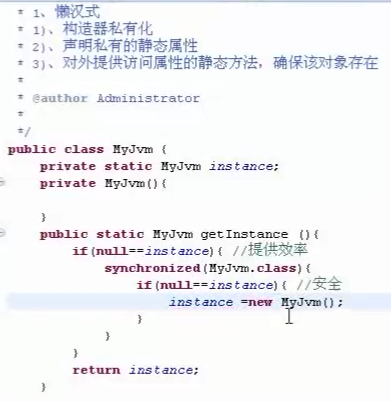
总结：





**提高效率所用：采用内部类存储单例对象。**

**类只有使用时才会加载，所以通过创建一个内部类存放单例对象，当调用到方法getInstance（）时，才会加载内部类，同时返回对象，也可以提高效率。**



**Runtime类**就是采用的饿汉式单例设计模式：

源代码：

