# 锁

## 锁对象

什么是锁对象?

每个java对象都有一个锁对象.而且只有一把钥匙.

如何创建锁对象：

可以使用this关键字作为锁对象，也可以使用所在类的字节码文件对应的Class对象作为锁对象

1. 类名.class

2. 对象.getClass()

Java中的每个对象都有一个内置锁，只有当对象具有同步方法代码时，内置锁才会起作用，当进入一个同步的非静态方法时，就会自动获得与类的当前实例（this）相关的锁，该类的代码就是正在执行的代码。获得一个对象的锁也成为获取锁、锁定对象也可以称之为监视器来指我们正在获取的锁对象。

因为一个对象只有一个锁，所有如果一个线程获得了这个锁，其他线程就不能获得了，直到这个线程释放（或者返回）锁。也就是说在锁释放之前，任何其他线程都不能进入同步代码（不可以进入该对象的任何同步方法）。释放锁指的是持有该锁的线程退出同步方法，此时，其他线程可以进入该对象上的同步方法。

1：只能同步方法（代码块），不能同步变量或者类

2：每个对象只有一个锁

3：不必同步类中的所有方法，类可以同时具有同步方法和非同步方法

4：如果两个线程要执行一个类中的一个同步方法，并且他们使用的是了类的同一个实例（对象）来调用方法，那么一次只有一个线程能够执行该方法，另一个线程需要等待，直到第一个线程完成方法调用，总结就是：一个线程获得了对象的锁，其他线程不可以进入该对象的同步方法。

5：如果类同时具有同步方法和非同步方法，那么多个线程仍然可以访问该类的非同步方法。

同步会影响性能（甚至死锁），优先考虑同步代码块。

6：如果线程进入sleep（） 睡眠状态，该线程会继续持有锁，不会释放。

## 死锁



经典的“哲学家就餐问题”，5个哲学家吃中餐，坐在圆卓子旁。每人有5根筷子（不是5双），每两个人中间放一根，哲学家时而思考，时而进餐。每个人都需要一双筷子才能吃到东西，吃完后将筷子放回原处继续思考，如果每个人都立刻抓住自己左边的筷子，然后等待右边的筷子空出来，同时又不放下已经拿到的筷子，这样每个人都无法得到1双筷子，无法吃饭都会饿死，这种情况就会产生死锁：每个人都拥有其他人需要的资源，同时又等待其他人拥有的资源，并且每个人在获得所有需要的资源之前都不会放弃已经拥有的资源。

当多个线程完成功能需要同时获取多个共享资源的时候可能会导致死锁。

1：两个任务以相反的顺序申请两个锁，死锁就可能出现

2：线程T1获得锁L1，线程T2获得锁L2，然后T1申请获得锁L2，同时T2申请获得锁L1，此时两个线程将要永久阻塞，死锁出现

如果一个类可能发生死锁，那么并不意味着每次都会发生死锁，只是表示有可能。要避免程序中出现死锁。

例如，某个程序需要访问两个文件，当进程中的两个线程分别各锁住了一个文件，那它们都在等待对方解锁另一个文件，而这永远不会发生。

3：要避免死锁

参看代码DeadLock

# 悲观锁/互斥锁（synchronized）

## 概述

在Java语言中，引入了对象互斥锁的概念，来保证共享数据操作的完整性。

每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。

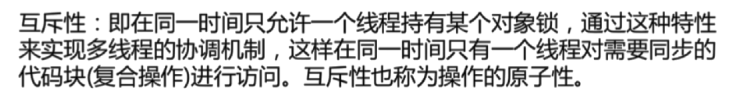
关键字synchronized 来与对象的互斥锁联系。当某个对象用synchronized修饰时，表明该对象在任一时刻只能由一个线程访问。

同步的局限性：导致程序的执行效率要降低

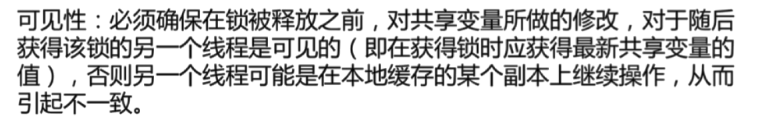
同步方法（非静态的）的锁为this。同步方法（静态的）的锁为当前类本身。

## 特性

### 互斥形

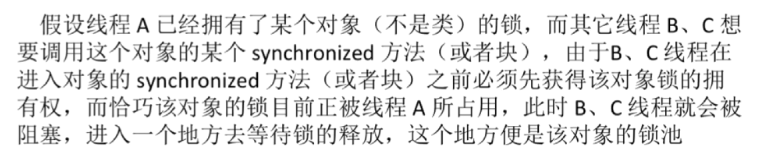


### 可见性



## 锁池和等待池

锁池EntryList



等待池WaitSet

调用进入等待池

## 释放锁的操作

### 会释放

1.当前线程的同步方法、同步代码块执行结束

2.当前线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行。

3.当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致异常结束

4.当前线程在同步代码块、同步方法中执行了线程对象的wait()方法，当前线程暂停，并释放锁。

### 不会释放

1.线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行

2.线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁（同步监视器）。应尽量避免使用suspend()和resume()来控制线程

## 互斥锁应用

### 单例设计模式之懒汉式

//关于懒汉式的线程安全问题：使用同步机制

//对于一般的方法内，使用同步代码块，可以考虑使用this。

//对于静态方法而言，使用当前类本身充当锁。

**class** Singleton{

**private** Singleton() {

// **TODO** Auto-generated constructor stub

}

**private** **static** Singleton *instance* = **null**;

**public** **static** Singleton getInstance(){

**if** (*instance* == **null**) {

**synchronized** (Singleton.**class**) {

**if** (*instance* != **null**) {

*instance* = **new** Singleton();

}

}

}

**return** *instance*;

}

}

**public** **class** TestSingleton {

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

Singleton s1 = Singleton.*getInstance*();

Singleton s2 = Singleton.*getInstance*();

System.*out*.println(s1 == s2);

}

}

### 银行取钱

**银行有一个账户。**

**有两个储户分别向同一个账户存3000元，每次存1000，存3次。每次存完打印账户余额。**

问题：该程序是否有安全问题，如果有，如何解决？

【提示】

1，明确哪些代码是多线程运行代码，须写入run()方法

2，明确什么是共享数据。

3，明确多线程运行代码中哪些语句是操作共享数据的

**拓展问题：可否实现两个储户交替存钱的操作。需要使用线程通信！**

/\*\*

\* 银行有一个账户。有两个储户分别向同一个账户存3000元，每次存1000，存3次。每次存完打印账户余额。

\* 1.是否涉及多线程？是。有2个储户（2种方式）

\* 2.是否共享数据？有，同一个账户

\* 3.考虑线程同步（2种方式处理线程安全）

\*/

**class** Account {

**double** balance; //余额

**public** Account() {

// **TODO** Auto-generated constructor stub

}

//存钱

**public** **synchronized** **void** deposit(**double** amt){

balance += amt;

**try** {

Thread.*currentThread*().*sleep*(10);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.*out*.println(Thread.*currentThread*().getName() +":" +balance);

}

}

**class** Customer **extends** Thread{

Account account;

**public** Customer(Account account) {

**this**.account = account;

}

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

**for** (**int** i = 0; i < 3; i++) {

account.deposit(1000);

}

}

}

**public** **class** TestAccount {

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

Account account = **new** Account();

Customer customer1 = **new** Customer(account);

Customer customer2 = **new** Customer(account);

customer1.setName("甲");

customer2.setName("乙");

customer1.start();

customer2.start();

}

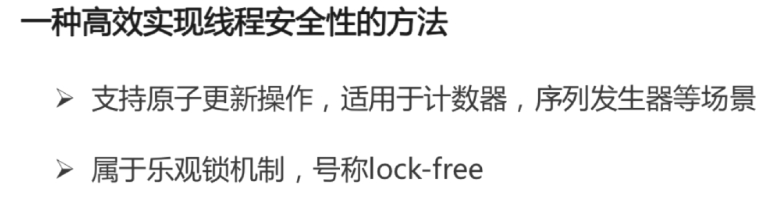
}

(

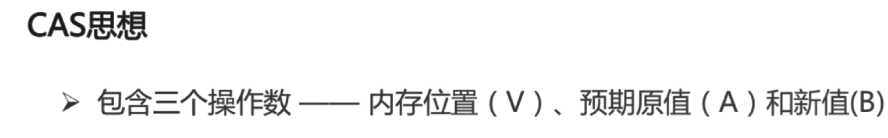
# 乐观锁（cas）

## Cas

### 概念



### 思想



### Unsale类

### Variable

### 缺点

