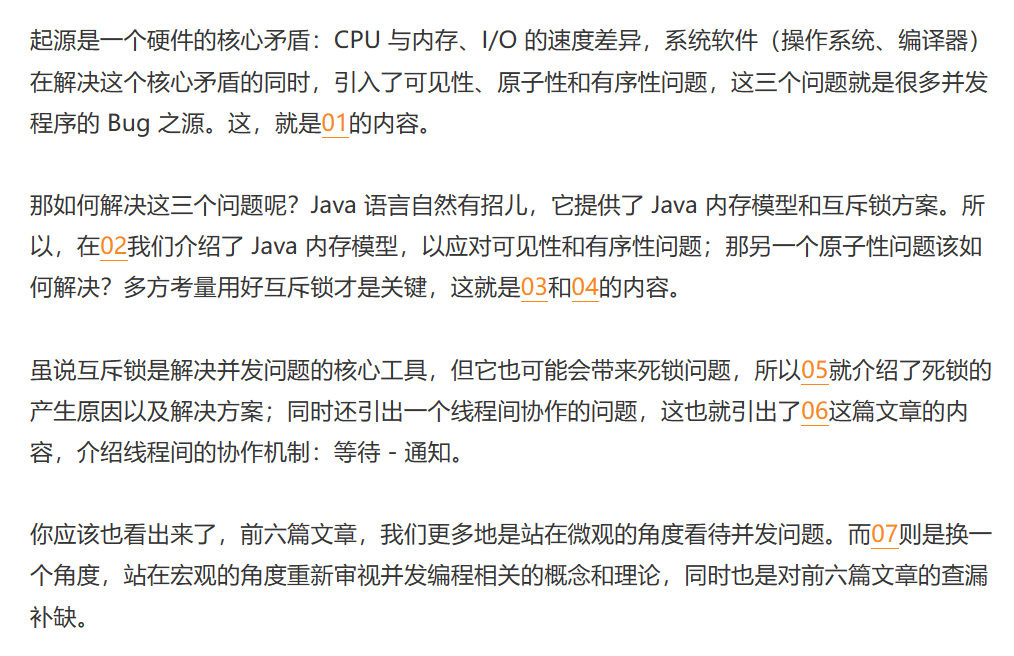
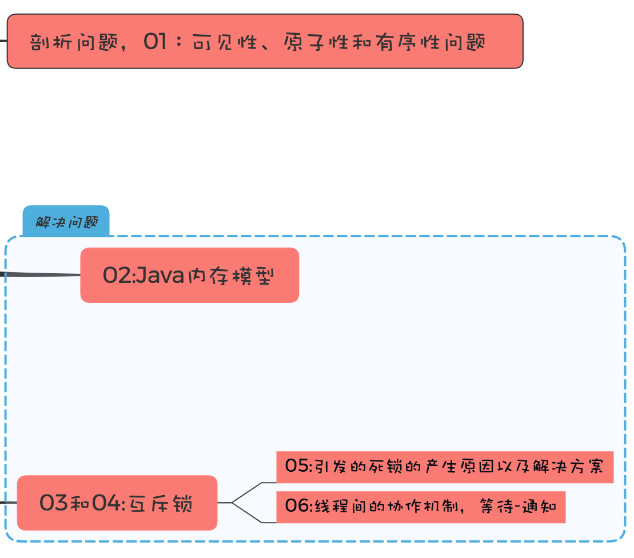
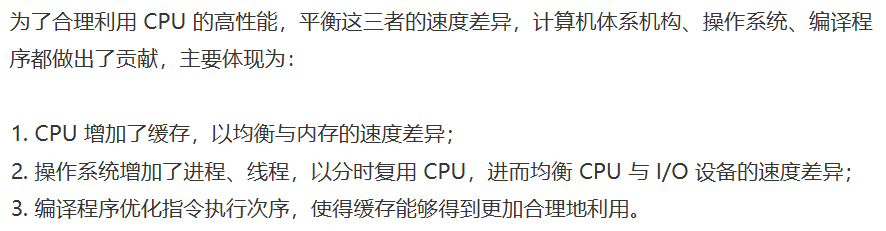
Java并发编程实战

# 总纲





# 01-问题源头

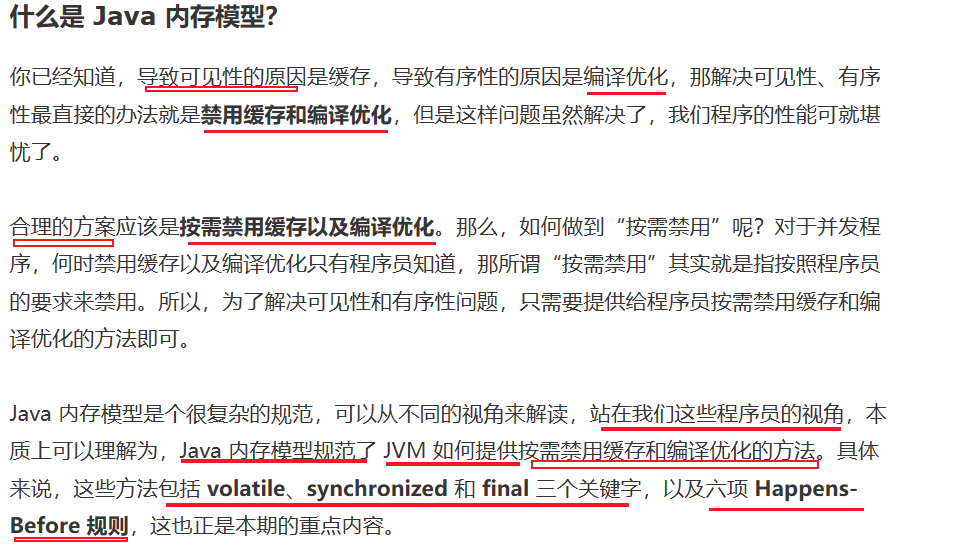


源头之一：缓存导致的可见性问题

源头之二：线程切换带来的原子性问题

源头之三：编译优化带来的有序性问题

# 02-Java内存模型



## Happens-Before 解决可见性

### volatile的本质

volatile 关键字并不是 Java 语言的特产，在C语言里也有，它最原始的意义就是**禁用 CPU 缓存。**

**声明一个 volatile 变量 volatile int x = 0，它表达的是：告诉编译器，对这个变量的读写，不能使用 CPU 缓存，必须从内存中读取或者写入。**

### Happens-Before的本质

Happens-Before规则要表达的是：**前面一个操作的结果对后续操作是可见的**。

也就是说Happens-Before 约束了编译器的优化行为，虽允许编译器优化，但是要求编译器优化后一定遵守 Happens-Before 规则。

在 Java 语言里面，Happens-Before 的语义本质上**是一种可见性**，A Happens-Before B 意味着 A 事件对 B 事件来说是可见的，无论 A 事件和 B 事件是否发生在同一个线程里。例如 A 事件发生在线程 1 上，B 事件发生在线程 2 上，Happens-Before 规则保证线程 2 上也能看到 A 事件的发生。

## Happens-Before 具体规则

### 1. 程序的顺序性规则

这条规则是指在一个线程中，按照程序顺序，前面的操作 Happens-Before 于后续的任意操作。例如：第 6 行代码 “x = 42;” Happens-Before 于第 7 行代码 “v = true;”，也就是说：程序前面对某个变量的修改一定是对后续操作可见的。

### 2. volatile 变量规则

这条规则是指对一个 volatile 变量的**写**操作， Happens-Before 于这个 volatile 变量的**读**操作。

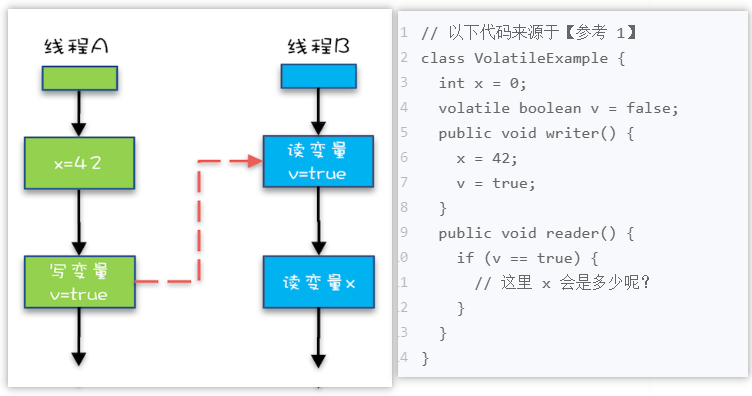
使用volatile为的是**禁用缓存**以及**编译优化。**

### 3. 传递性规则

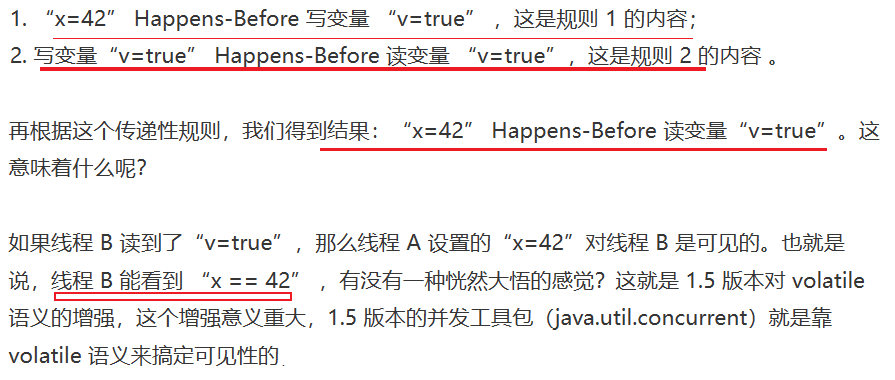
这条规则是指如果 A Happens-Before B，且 B Happens-Before C，那么 A Happens-Before C。

这就是 1.5 版本对 volatile 语义的增强。

### 案例-【顺序-volatile-传递性规则】



**推理过程**



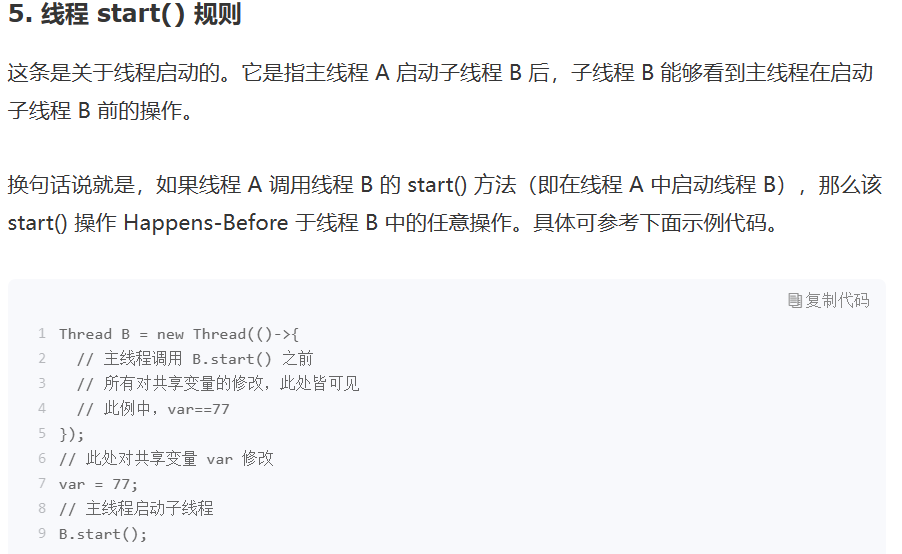
### 4. 管程中锁的规则

这条规则是指对一个锁的**解锁** Happens-Before 于后续对这个锁的**加锁**。

管程是一种通用的同步原语，在 Java 中指的就是 synchronized，synchronized 是 Java 里对管程的实现。

在Java里管程中的锁是隐式实现的。在进入同步块之前，会自动加锁，而在代码块执行完会自动释放锁，加锁以及释放锁都是编译器帮我们实现的。

### 5. 线程start()规则



### 6. 线程join()规则



### 7.final

**volatile** 为的是禁用缓存以及编译优化。

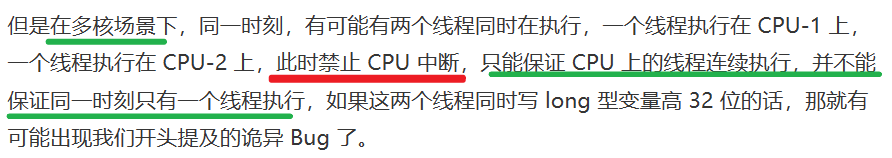
**final** 修饰变量时，初衷是告诉编译器：这个变量生而不变，可以使劲儿优化。

# 03-互斥-原子性

**我们把一个或者多个操作在 CPU 执行的过程中不被中断的特性称为原子性.**

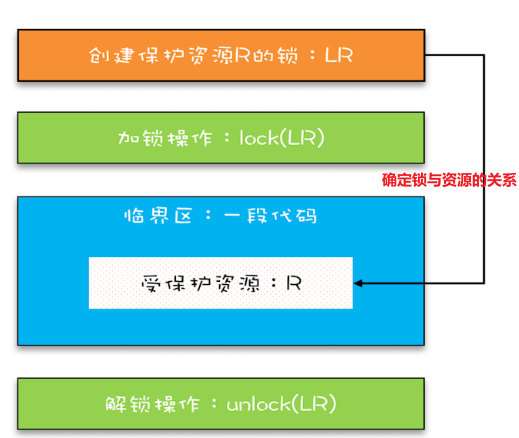
原子性问题的源头是线程切换，而操作系统做线程切换是依赖 CPU 中断的，所以禁止 CPU 发生中断 就能 禁止线程切换。

“**同一时刻只有一个线程执行**”我们称之为**互斥**。如果我们能够保证对共享变量的修改是互斥的，那么，无论是单核 CPU 还是多核 CPU，就都能保证原子性了。



## 锁模型

我们知道在现实世界里，锁和锁要保护的资源是有对应关系的，比如你用你家的锁保护你家的东西，我用我家的锁保护我家的东西。在并发编程世界里，锁和资源也应该有这个关系.

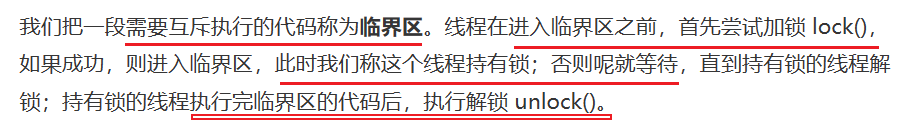


**首先**，我们要把临界区要保护的资源标注出来，如图中临界区里增加了一个元素：受保护的资源 R；

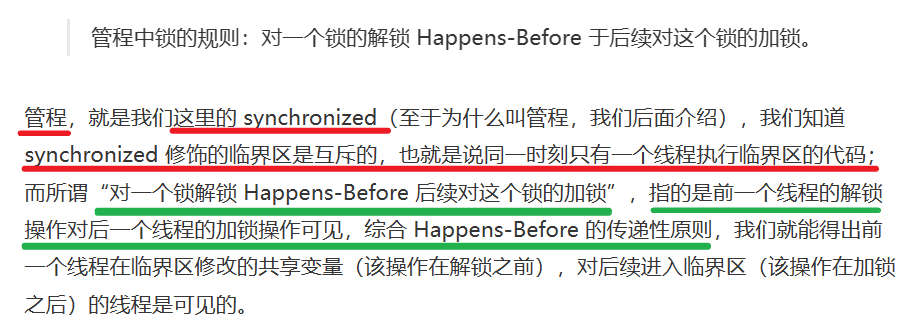
**其次**，我们要保护资源 R 就得为它创建一把锁 LR；

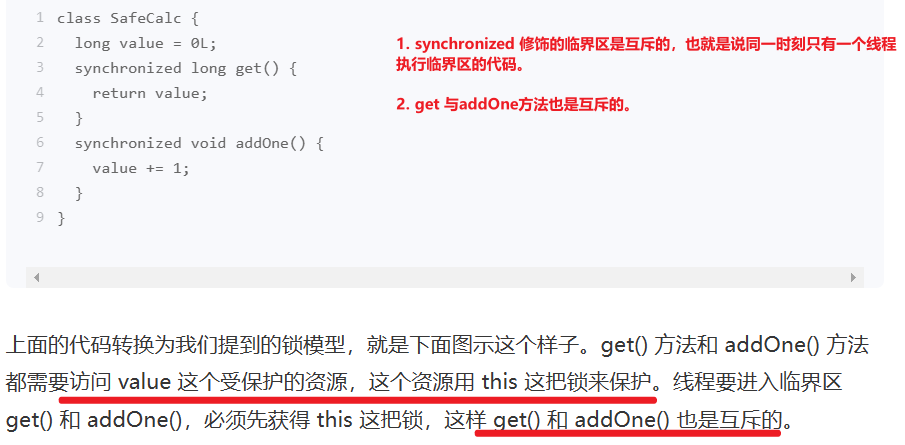
**最后**，针对这把锁 LR，我们还需在进出临界区时添上加锁操作和解锁操作。

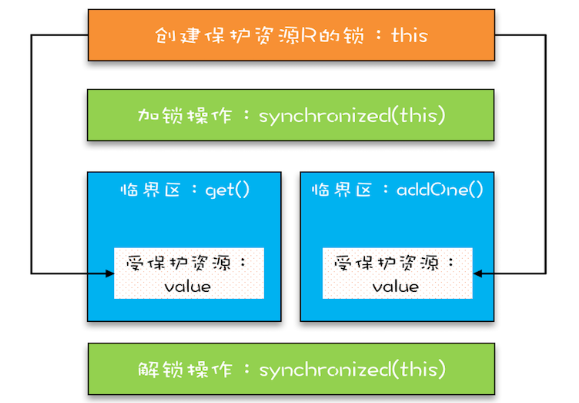
**另外**，在锁 LR 和受保护资源之间，我特地用一条线做了关联，这个关联关系非常重要。



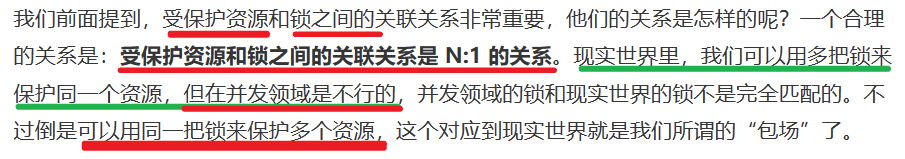
## Synchronized-锁

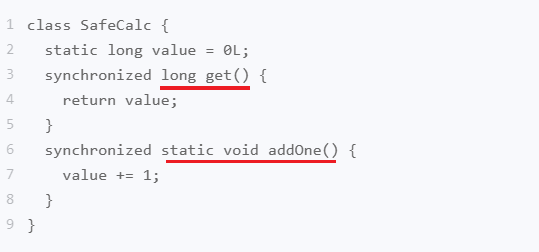






## 锁和受保护资源的关系



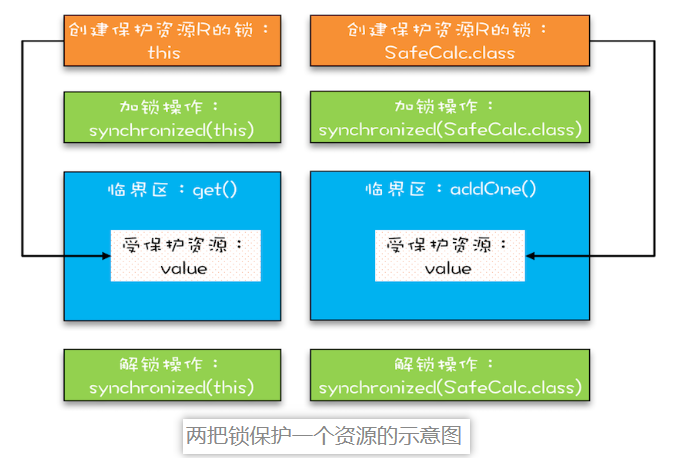


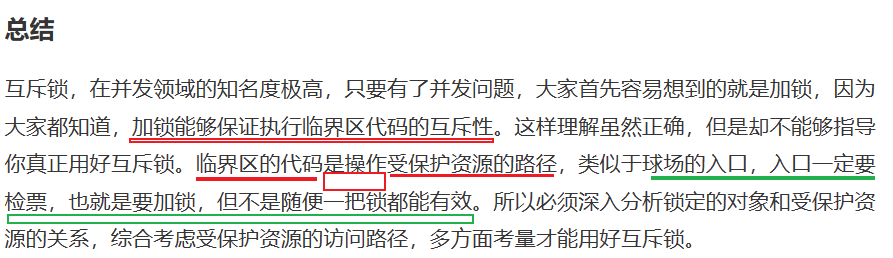
**上面代码是用两个锁保护一个资源。这个受保护的资源就是静态变量 value，两个锁分别是 this 和 SafeCalc.class**。

由于临界区 get() 和 addOne() 是用两个锁保护的，因此这两个临界区没有互斥关系，

**临界区addOne()对value的修改** 对 **临界区get()也没有可见性保证**，这就导致并发问题了。

下面这幅图来形象描述这个关系。





**加锁本质就是在锁对象的对象头中写入当前线程id**

# 04一把锁保护多个资源

## 锁的类型

**// 1修饰非静态方法**

synchronized void foo() {

锁是this,也就是调用的这个方法的对象，因此锁保护的范围是 这个对象内的资源，

无法对其他对象的字样进行保护。

}

**// 2修饰静态方法**

synchronized static void bar() {

该锁是当前调用这个方法的类，因此该锁的保护范围就是整个类。

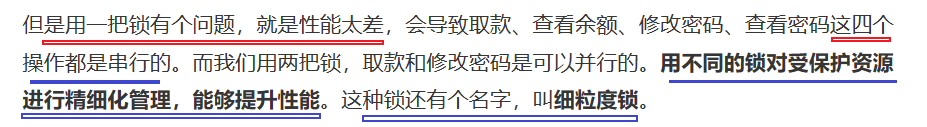
}

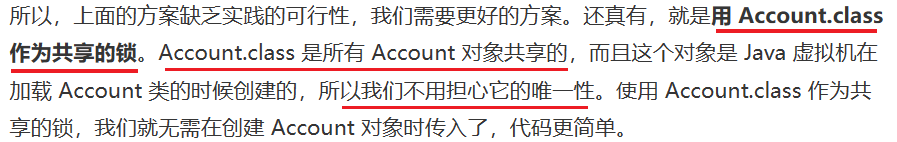
**//3代码块同步**

synchronized(lock) {

该同步代码可以实现以上两种的同步，关键在于lock是实例对象 **还是** 是class类

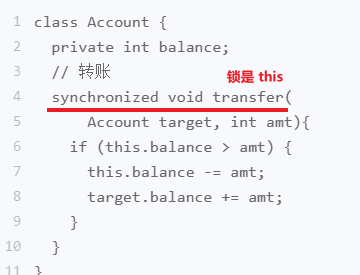
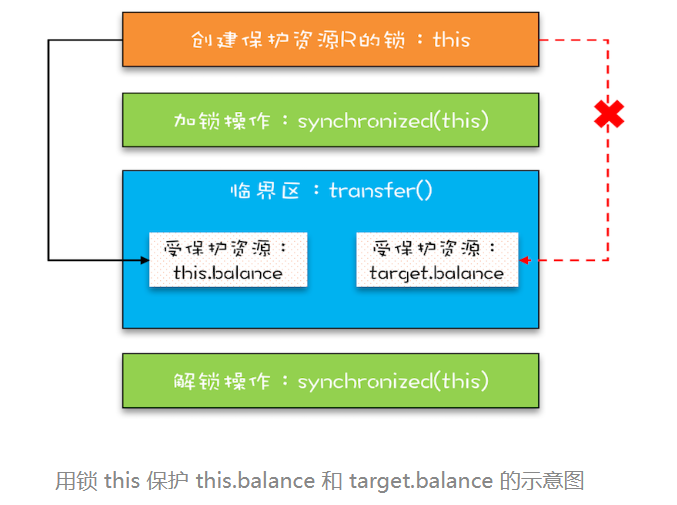
}





## 保护有关联关系的多个资源

### 错误方式

每创建一个 Account对象的时候,都有一个this，每个对象的this是不一样的。

原因是：锁不是唯一的，因为每个对象调用 transfer方法的时候 锁都是不一样的。

代码案例：

Account a = New Account();

Account b = New Account();

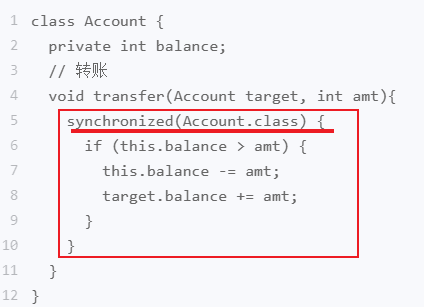
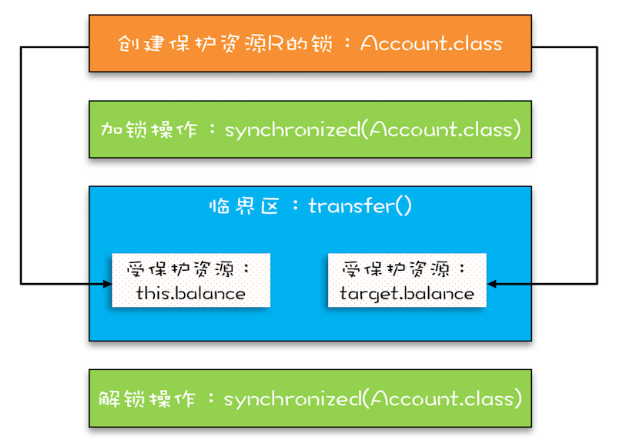
Account c = New Account();

a.transfer(b, 100) a账户向b账户转100

b.transfer(c, 100) b账户向c账户转100

当两个转账动作同时执行时，虽然transfer方法都是一样的，但是每个对象transfer方法的锁是不一样的。这就导致 当前对象a的this锁  **无法限定** 另一个对象b的transfer方法 不执行转账操作。

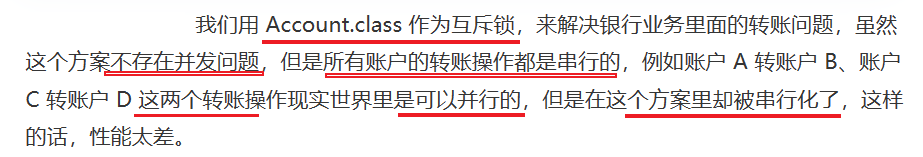
### 正确方式

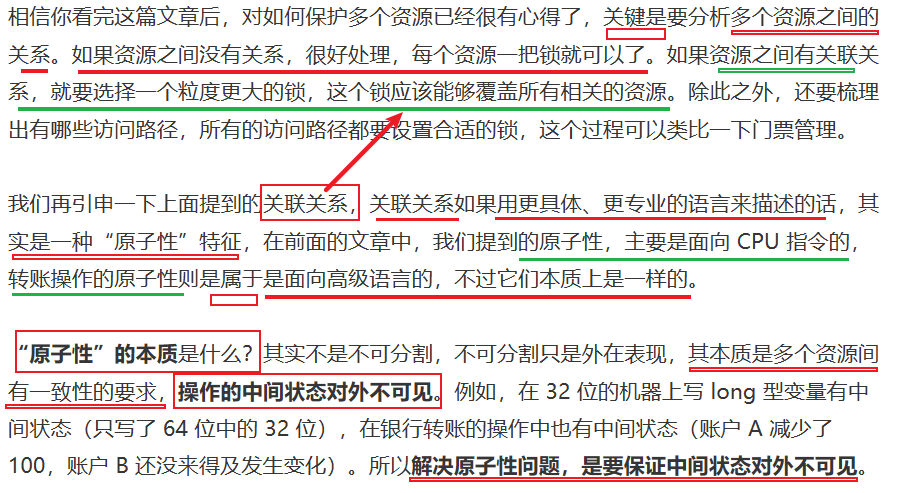
**this 是对象级别的锁，每个对象的this都不同，指定当前对象共享。**

**xx.class 是共享锁, 该class对每个对象的是一样的，对所有 xxx对象共享。**

### 产生的效果

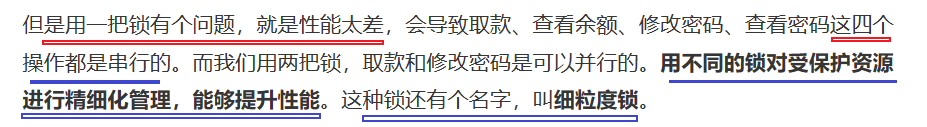


## 总结



# 05-死锁

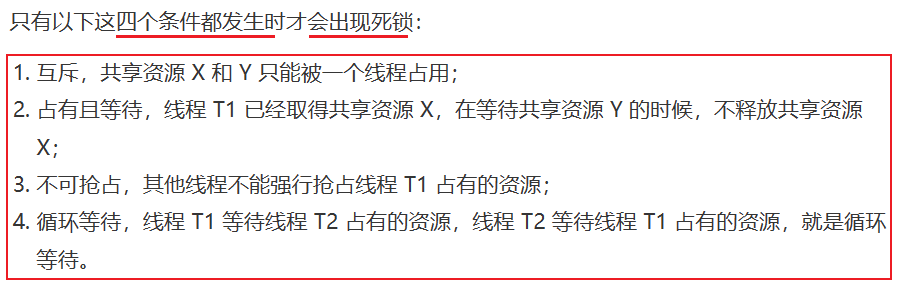
## 细粒度锁



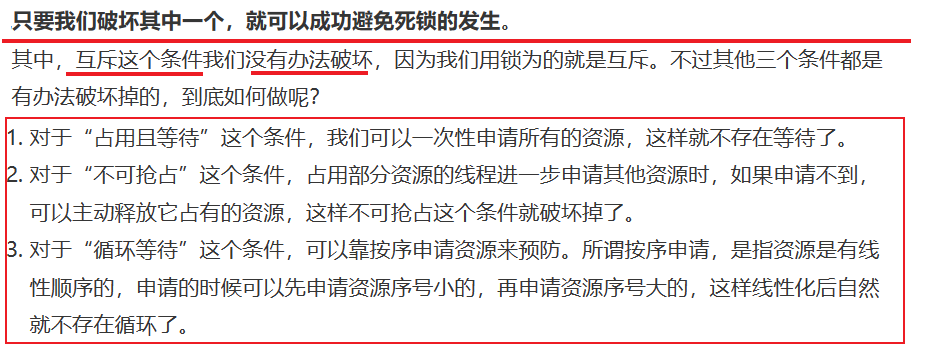
**使用细粒度锁可以提高并行度，是性能优化的一个重要手段。**

**但是使用细粒度锁是有代价的，代价就是可能会导致死锁。**

## 产生死锁的条件



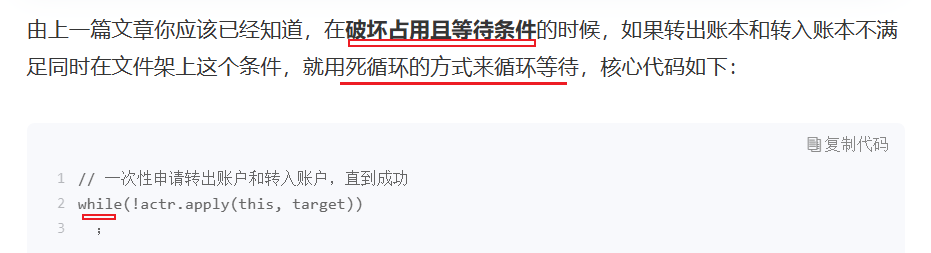
## 避免死锁方式



# 06-“等待-通知”机制

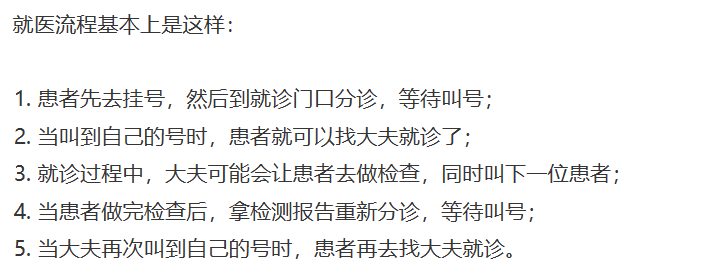
## 为啥产生

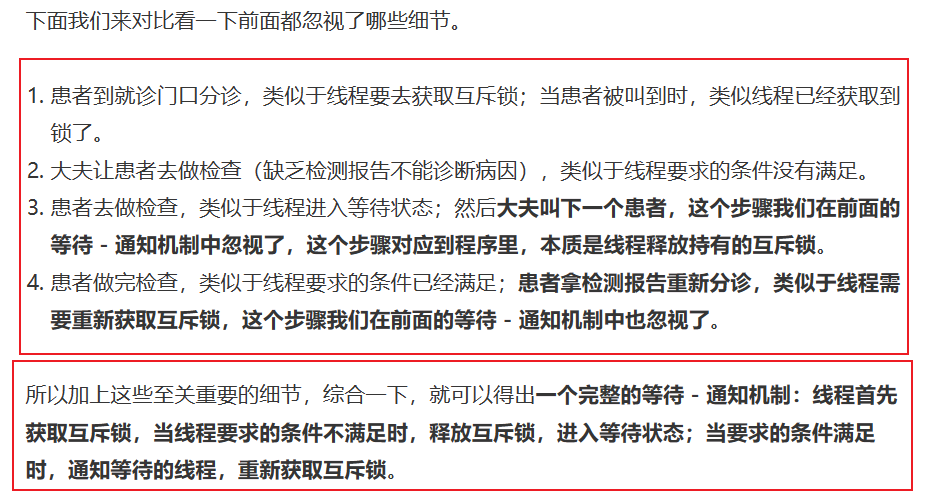




用“等待-通知”**机制 优化循环等待，解决死锁的问题**

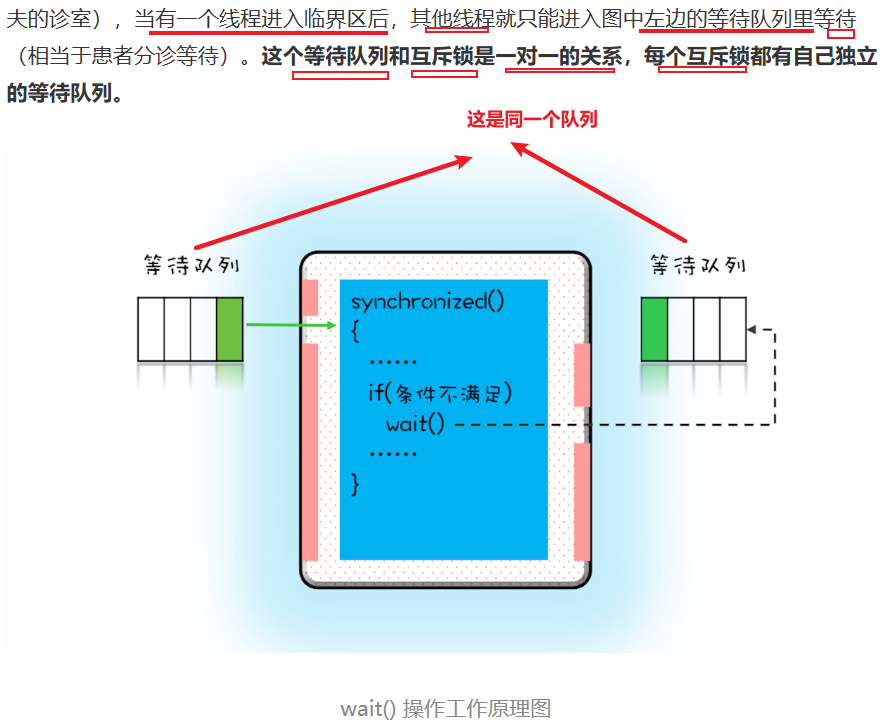
## 流程

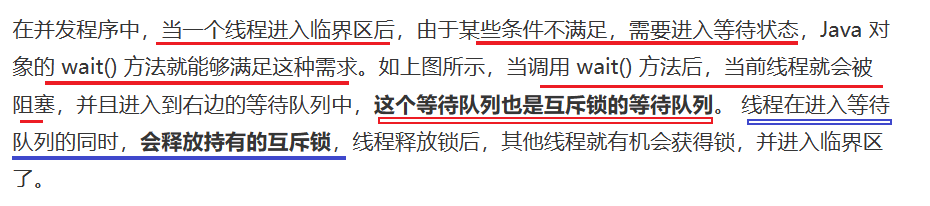




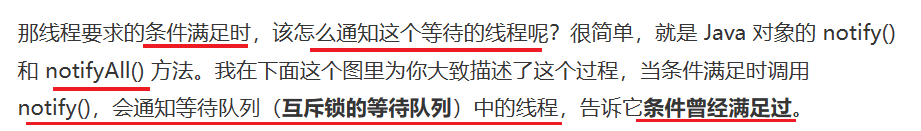
Java 语言内置的 synchronized 配合 wait()、notify()/ notifyAll() 这三个方法就能实现 **等待-通知机制**。

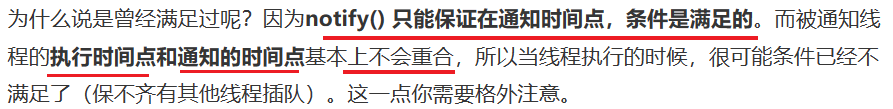
## Wait

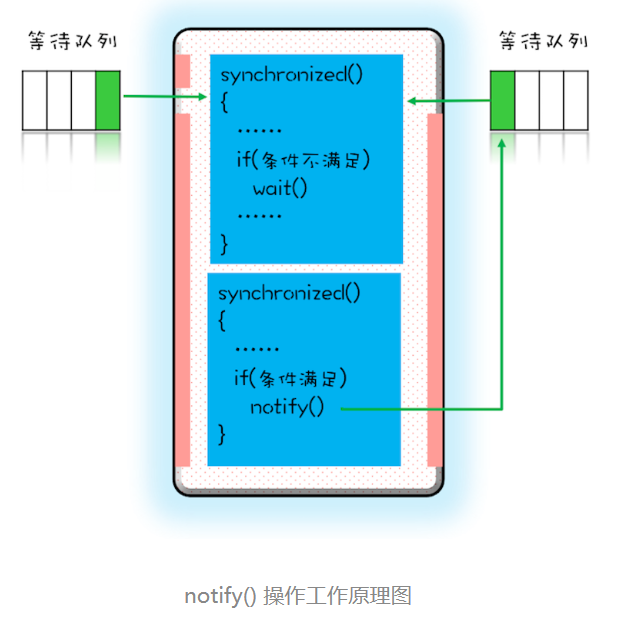




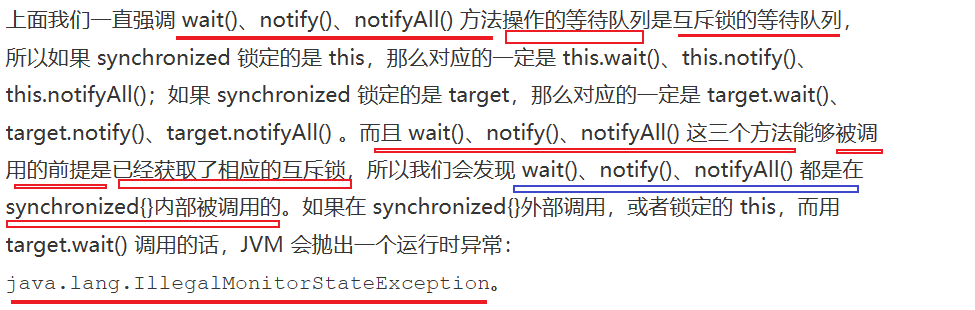
## notifyAll



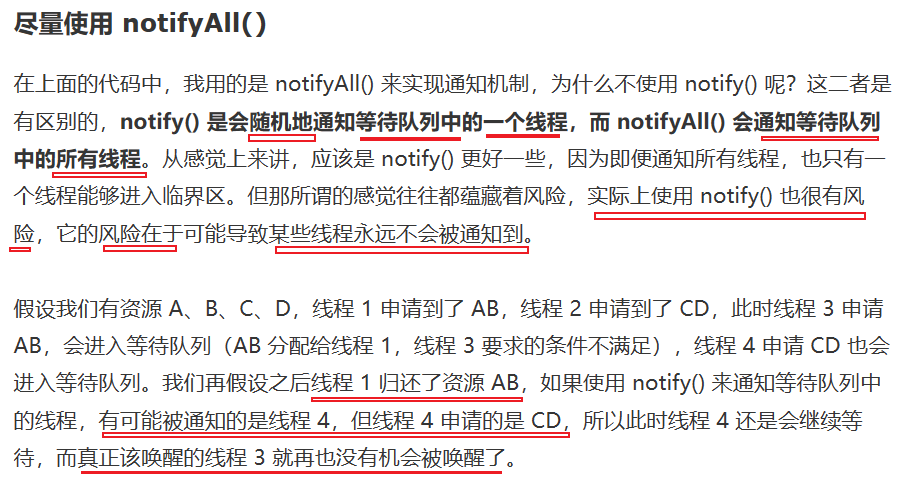




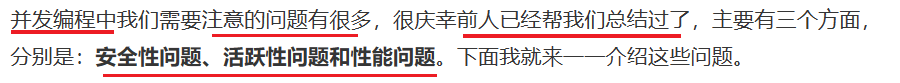
## 注意



## Notify与notifyAll区别



# 07- 前六章的一个总结



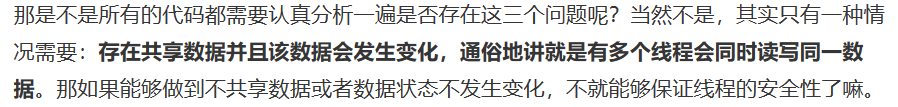
## 安全性

### 1.线程安全产生的原因

那什么是线程安全呢？**其实本质上就是正确性**，而正确性的含义就是程序按照我们期望的执行，不要让我们感到意外

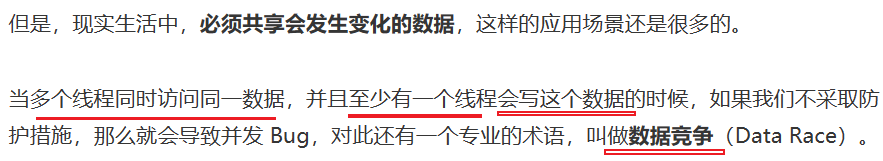
并发 Bug 的三个主要源头**：原子性问题、可见性问题和有序性问题**。也就是说，理论上线程安全的程序，就要避免出现原子性问题、可见性问题和有序性问题。

活跃性以

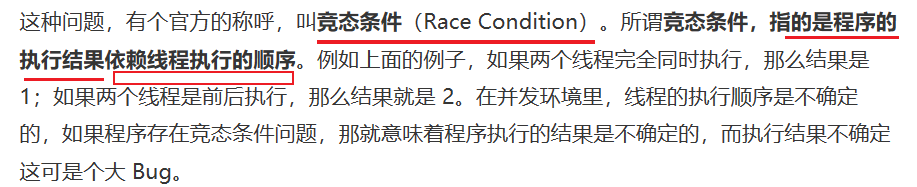


### 2.会发生变化的数据

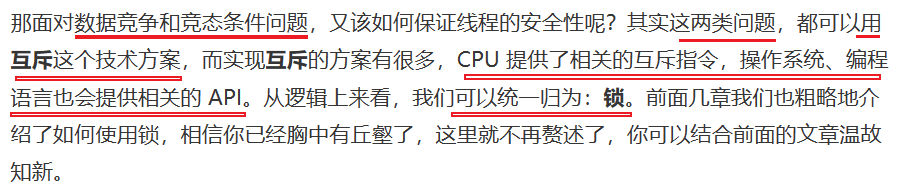
#### 数据竞争



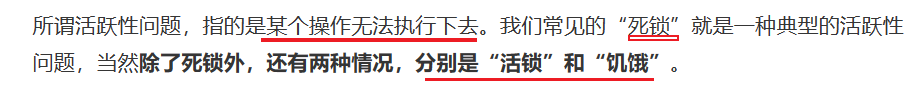
#### 竞态条件



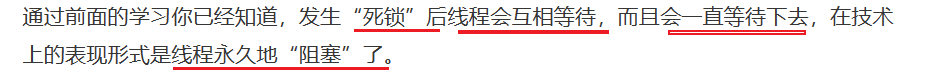
### 3.解决方法



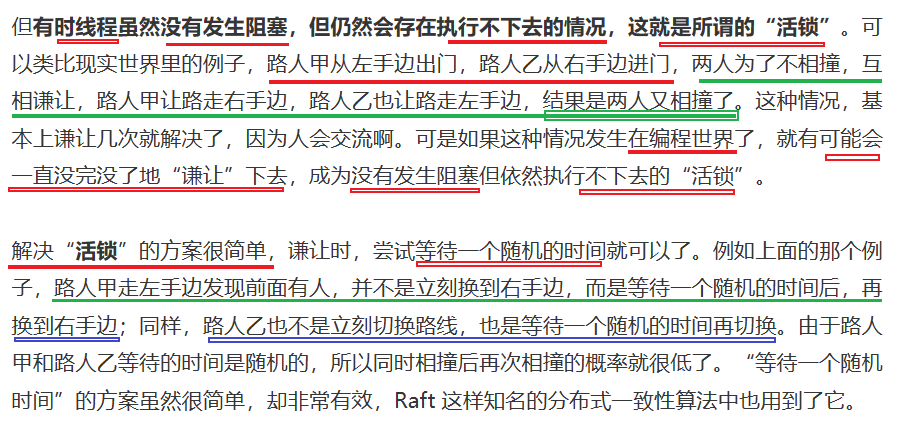
## 活跃性



### 死锁

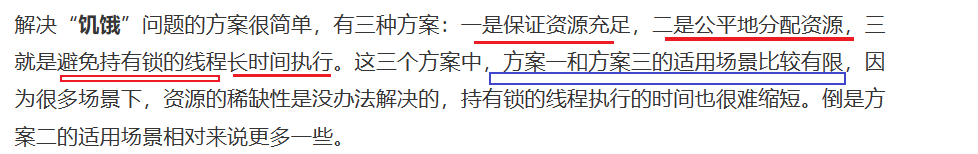


### 活锁



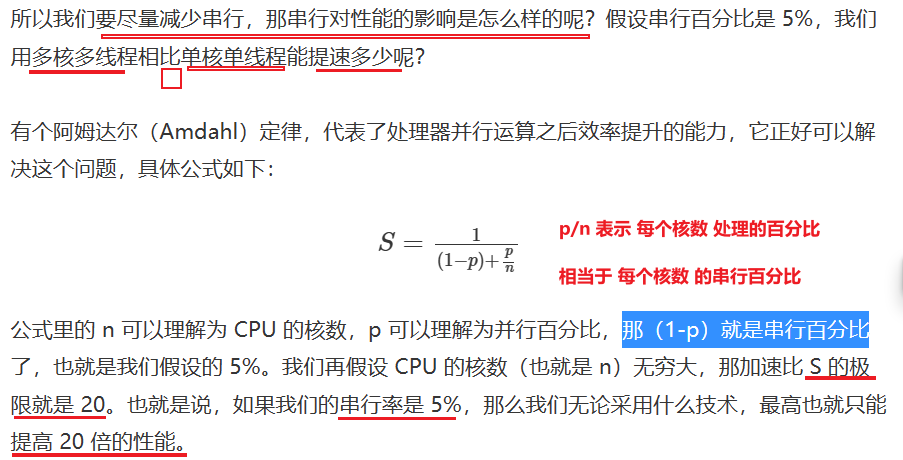
### 饥饿

**所谓“饥饿”指的是线程因无法访问所需资源而无法执行下去的情况。**

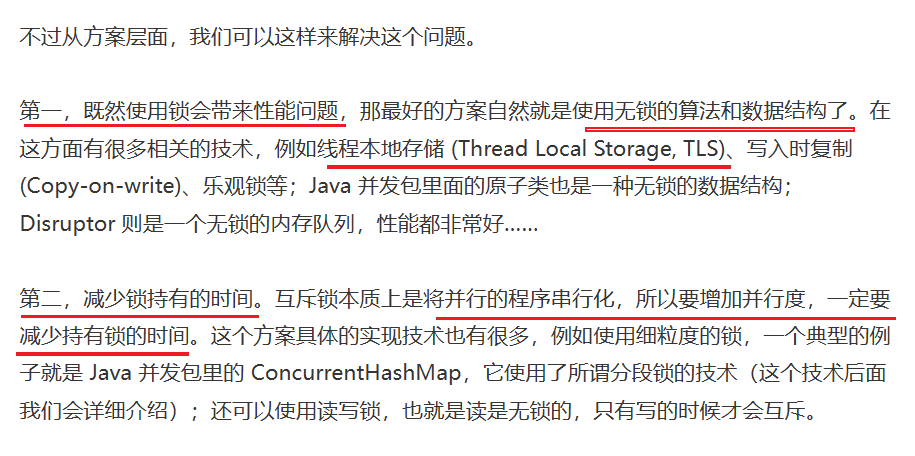


## 性能问题

### 计算公式



### 解决方案



### 指标

