# 开篇词-为什么学并发编程

1.最开始数据库和tomact等中间件，基本上不用写并发程序；近几年多核发展64核，大型互联网厂商系统并发量轻松过百万，传统中间件和数据库成为瓶颈

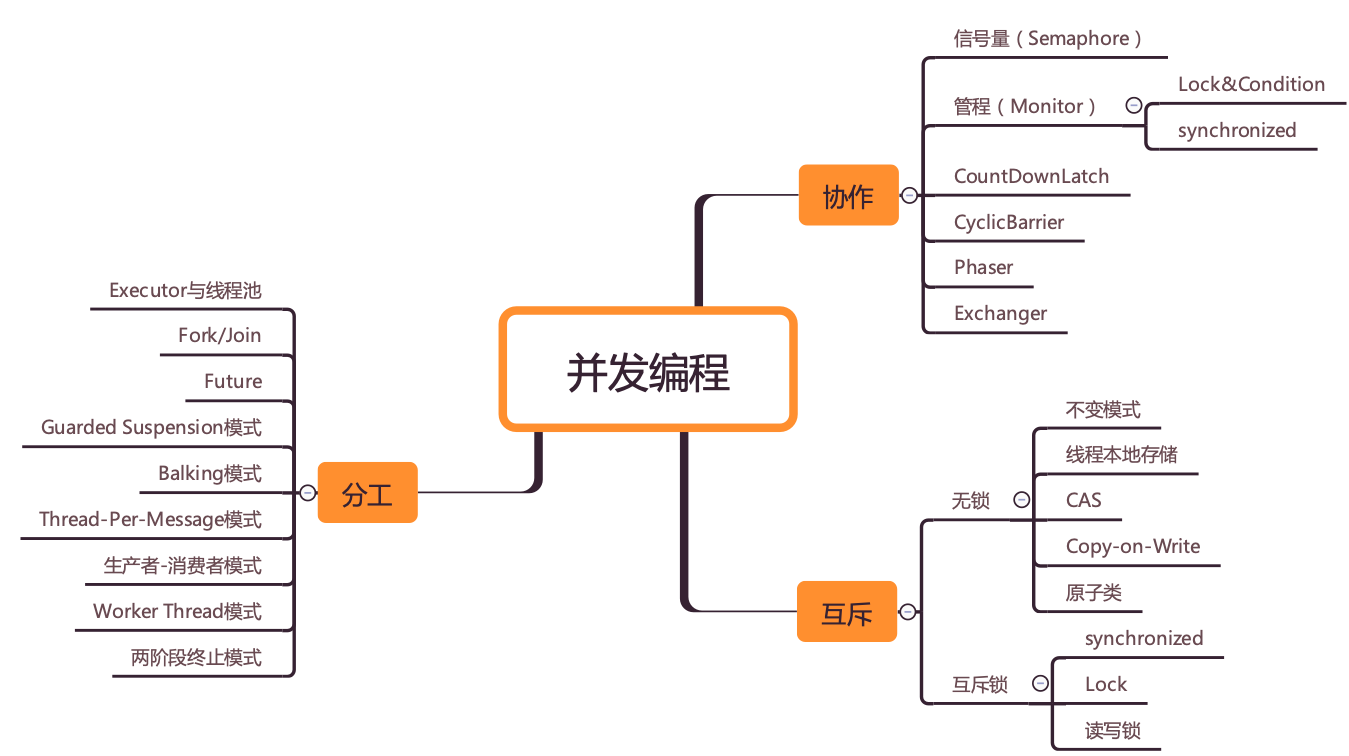
2.synchronized wait()/notify();SDK并发包condition

3.并发三个核心问题：分工，同步，互斥。SDK并发包很大部分内容这样组织。Fork/Join分工，countDownLatch同步，可重入锁互斥手段

4.理解并发，从理论到实现

# 学习攻略-如何才能学号并发编程

理解分工同步互斥概念，拿着java并发编程知识全景图去查漏补缺



# 一、可见性，原子性和有序性问题：并发编程bug的源头

源头之一：缓存导致的可见性问题

可见性：一个线程对并发变量的修改，另一个线程能够立刻看到

多核发展，cpu缓存机制。从缓存写内存的时机也不确定

例子：累加

public class Test {

private long count = 0;

private void add10K() {

int idx = 0;

while(idx++ < 10000) {

count += 1;

}

}

public static long calc() {

final Test test = new Test();

// 创建两个线程，执行 add() 操作

Thread th1 = new Thread(()->{

test.add10K();

});

Thread th2 = new Thread(()->{

test.add10K();

});

// 启动两个线程

th1.start();

th2.start();

// 等待两个线程执行结束

th1.join();

th2.join();

return count;

}

}

源头之二：线程切换带来的原子性问题

例子：可见性已经保证，count += 1，3个指令

源头之三：编译优化带来的有序性问题

例子：不加valitile关键字的双重校验锁：赋了地址，却还未初始化完毕

public class Singleton {

static Singleton instance;

static Singleton getInstance(){

if (instance == null) {

synchronized(Singleton.class) {

if (instance == null)

instance = new Singleton();

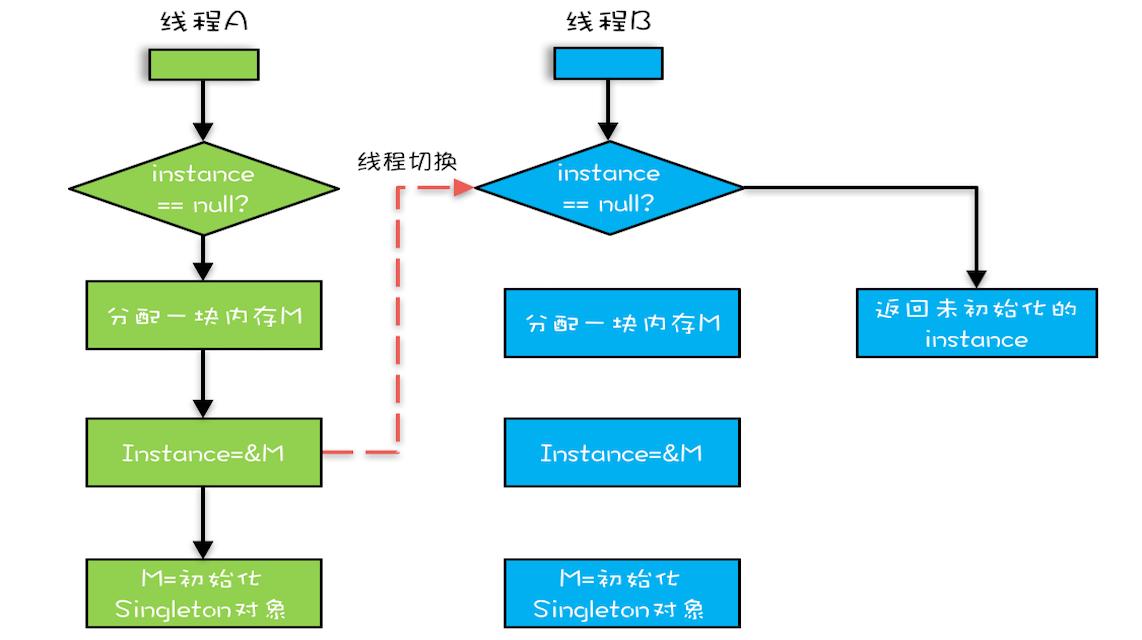
}

}

return instance;

}

}



思考题：long 32位。多条指令，无法保证原子性

# 二、java内存模型：看java如何解决可见性和有序性问题

java内存模型：volatile,synchronized,final,六项happens-before规则

Happens-Before规则：

1.程序的顺序性规则

2.volatile变量规则

3.传递性

4.管程中锁的规则：实现-强制写内存

5.线程start规则

6.线程join规则

被忽视的final：

对象this逸出，构造函数中将this值传给全局变量，此时this可能还没初始化完毕

# 互斥锁（上）：解决原子性问题

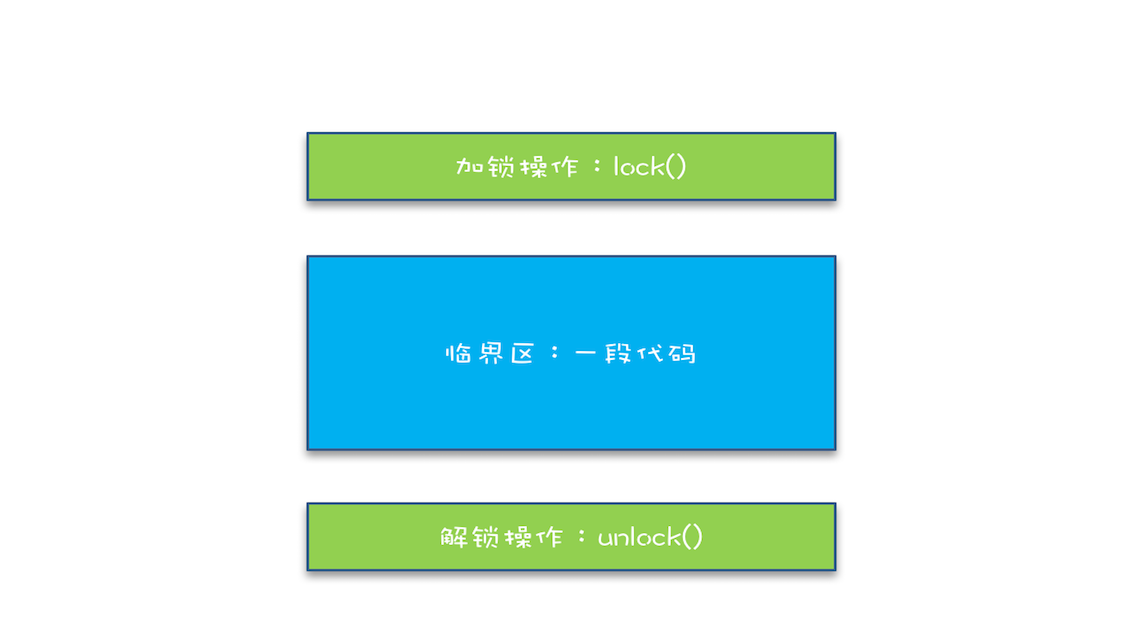


图 1简易锁模型

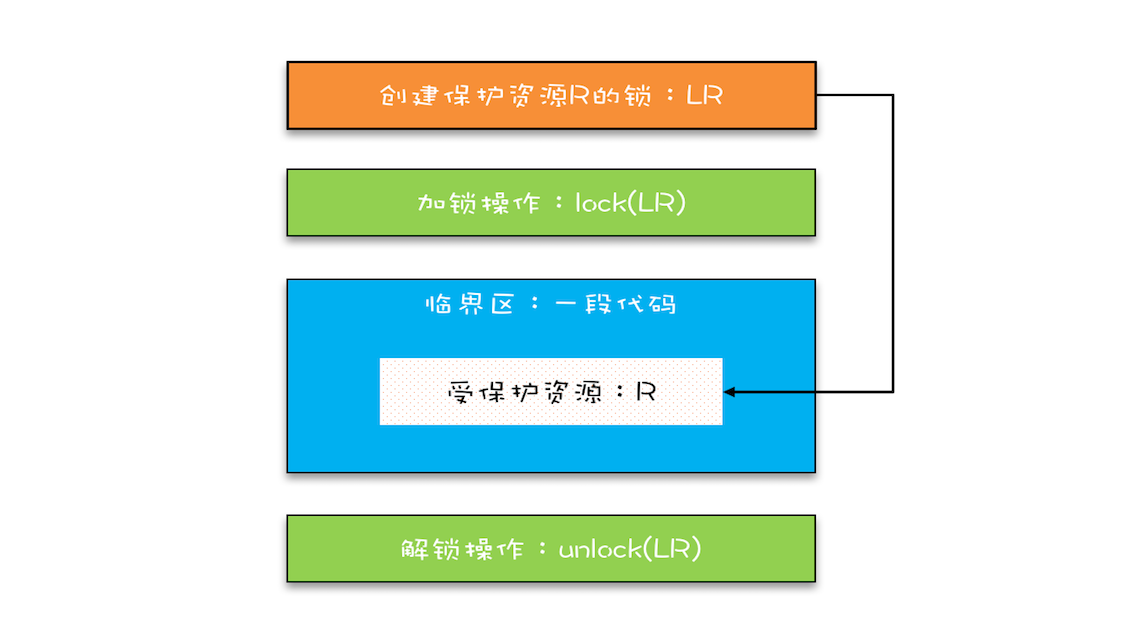


图 2改进后的锁模型

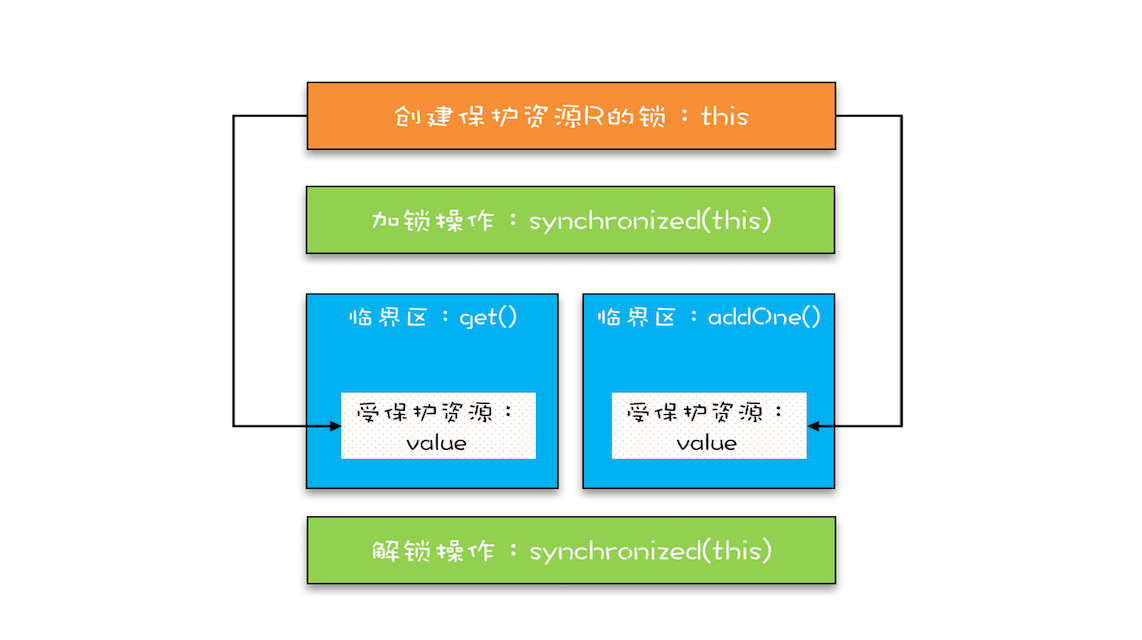


图 3保护临界区get()和addOne()的示意图

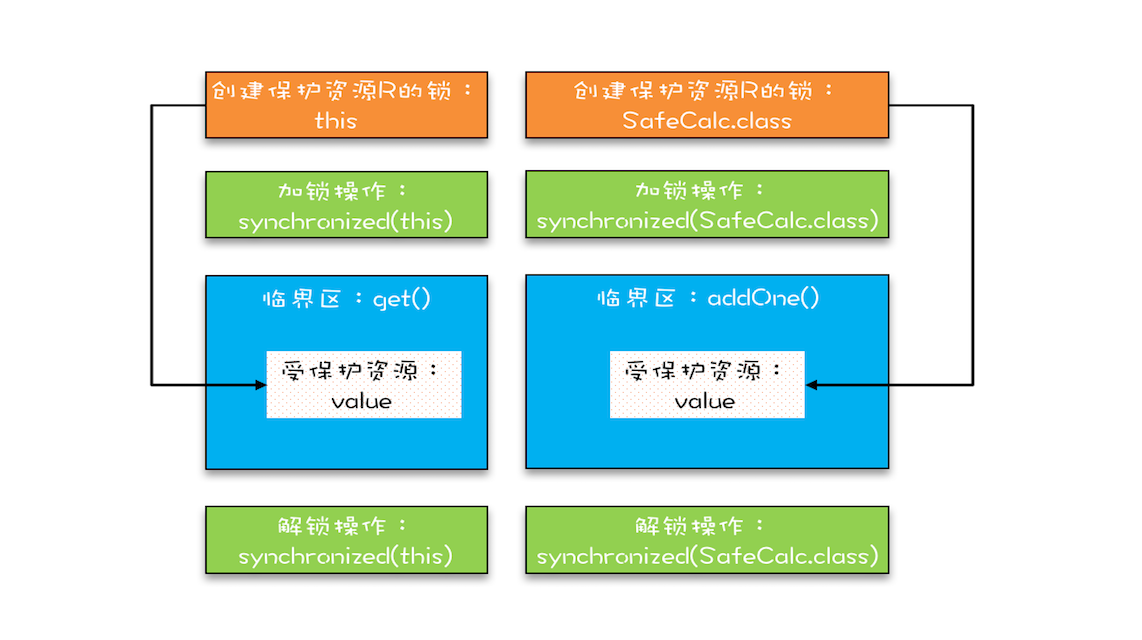


图 4两把锁保护一个资源的示意图

# 互斥锁（下）：如何用一把锁保护多个资源？

一把锁可以保护一个以上资源（1:N），但是一个资源用多把锁是严重错误的

保护没有关联关系的的多个资源：一个资源一把锁

保护有关联关系的多个资源：

**注意**锁要覆盖所有受保护资源

错误姿势：

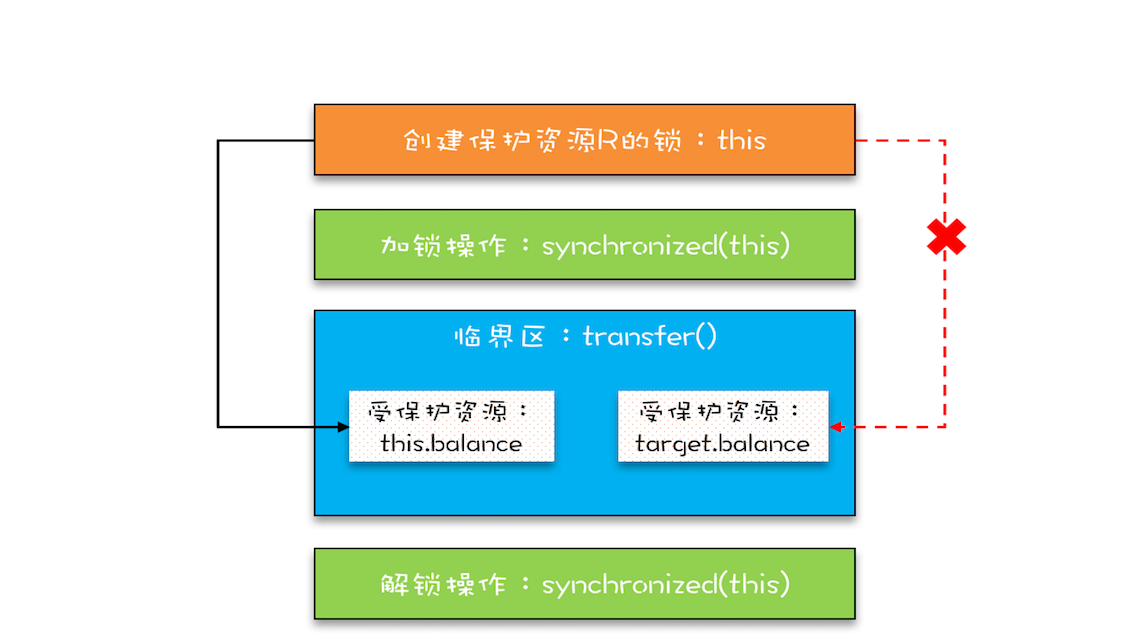


图 1锁this保护this.balance和target.balance

正确姿势：

New 账户时，每个账户共有一把锁。（传错就完啦）

用账户类对象作为锁（粒度太大）

按顺序获取this，target对象锁（AB,BA转账可能死锁）

# 一不小心死锁了怎么办？

死锁例子：

class Account{

private int balance;

//转账

void transfer(Account target,int amt){

//锁定转出账户

synchronized(this){

//锁定转入账户

synchronized(target){

if(this.balance > amt){

this.balance -= amt;

targer.balance += amt;

}

}

}

}

}

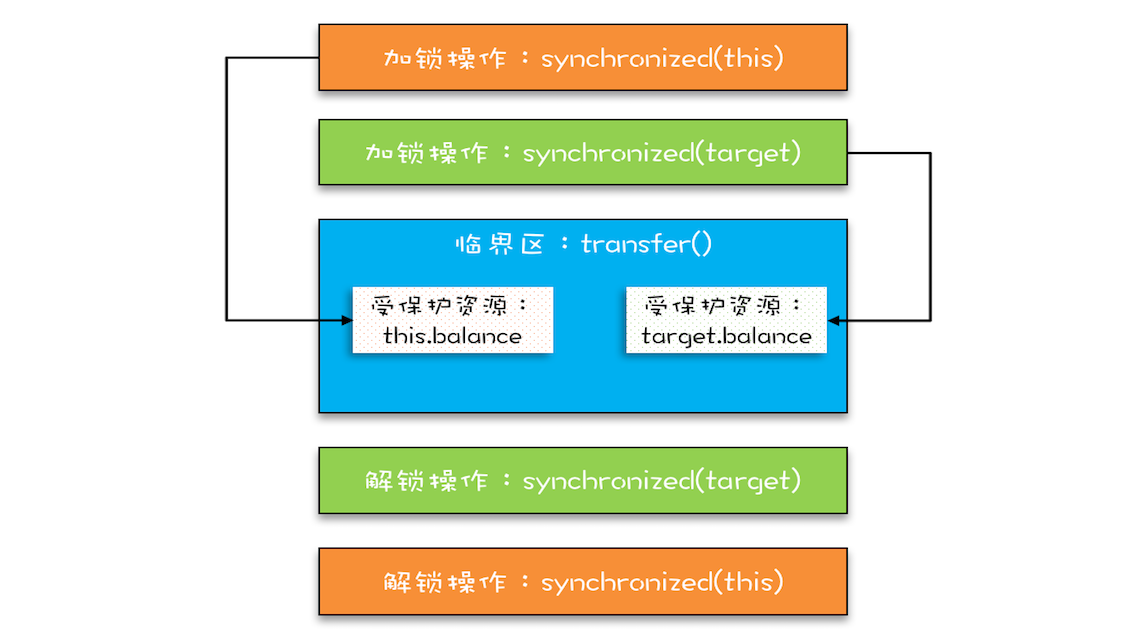


图 2两个转账操作并行示意图

细粒度锁可以提高并行度，是性能优化的一个重要手段，代价是可能造成死锁

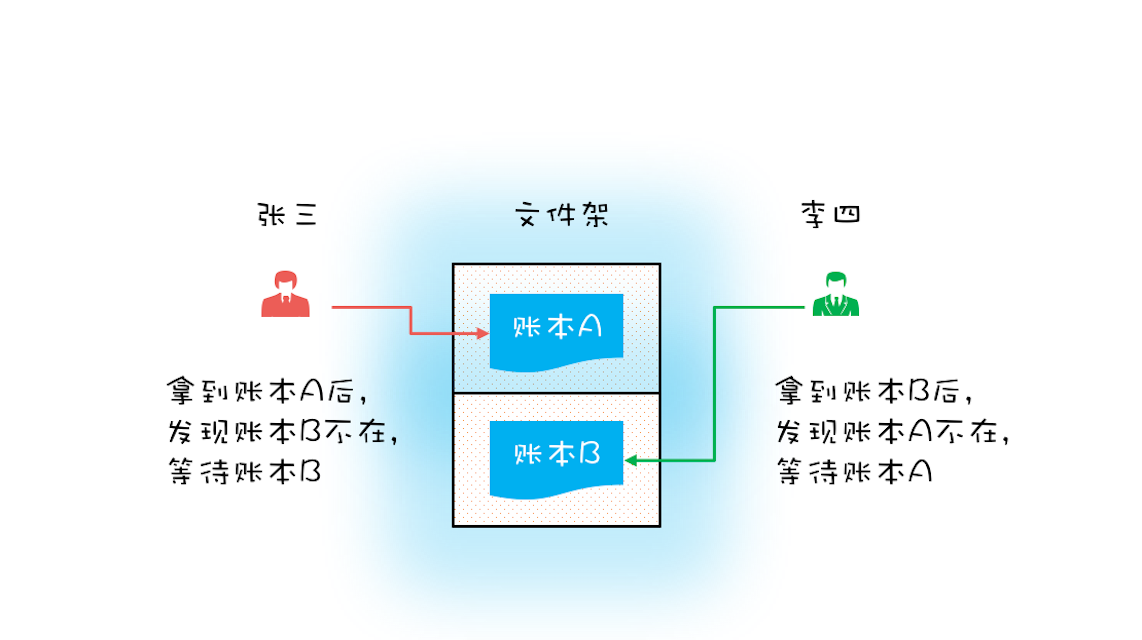


图 3转账业务中的“死等”

**死锁定义**：一组相互竞争资源的线程因相互等待，导致“永久”阻塞的现象

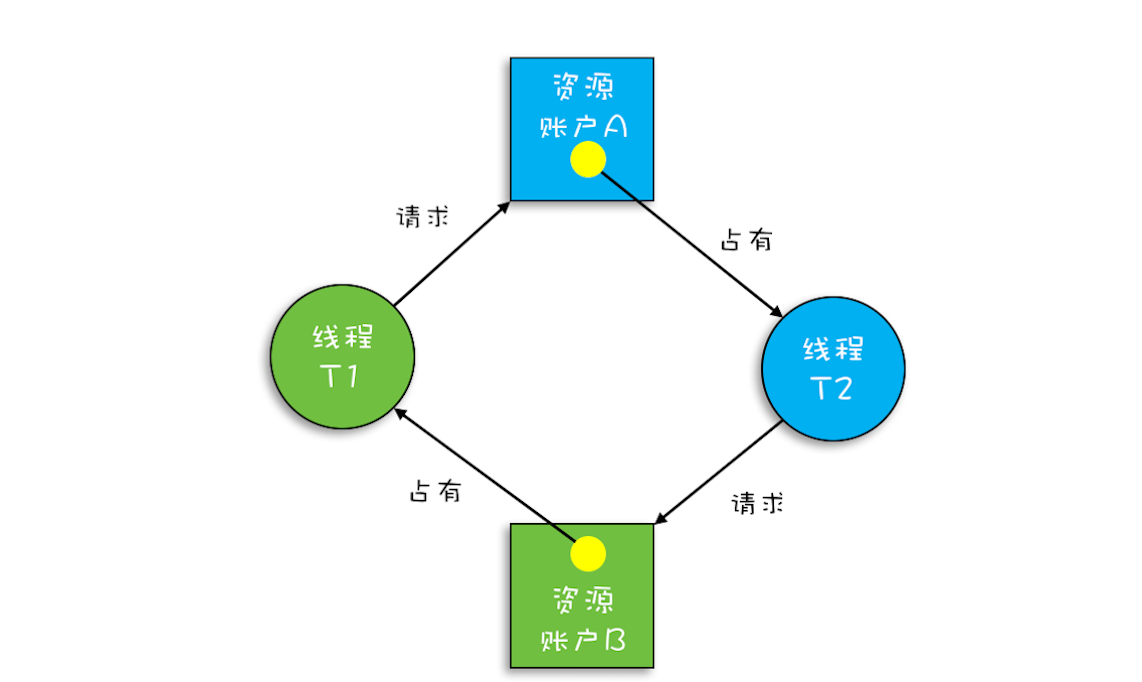


图 4转账发生死锁时的资源分配图

如何**预防死锁**

以下**四个条件都发生才会造成死锁**：

1、互斥：共享资源X和Y只能被一个线程占用（锁天然保证，管程模型）

2、占有且等待：线程T1已经取得共享资源X，在等待共享资源Y的时候，不释放共享资源X（占有了资源不主动释放）

3、不可抢占：其它线程不能强行抢占线程T1占有的资源（不能抢别人占有的资源）

4、循环等待：线程T1等待线程T2占有的资源，线程T2等待线程T1占有的资源（很直观）

**破坏其中一个就可以避免死锁**

互斥没办法破坏

1. 占有且等待。一次申请完所有，就不存在等待

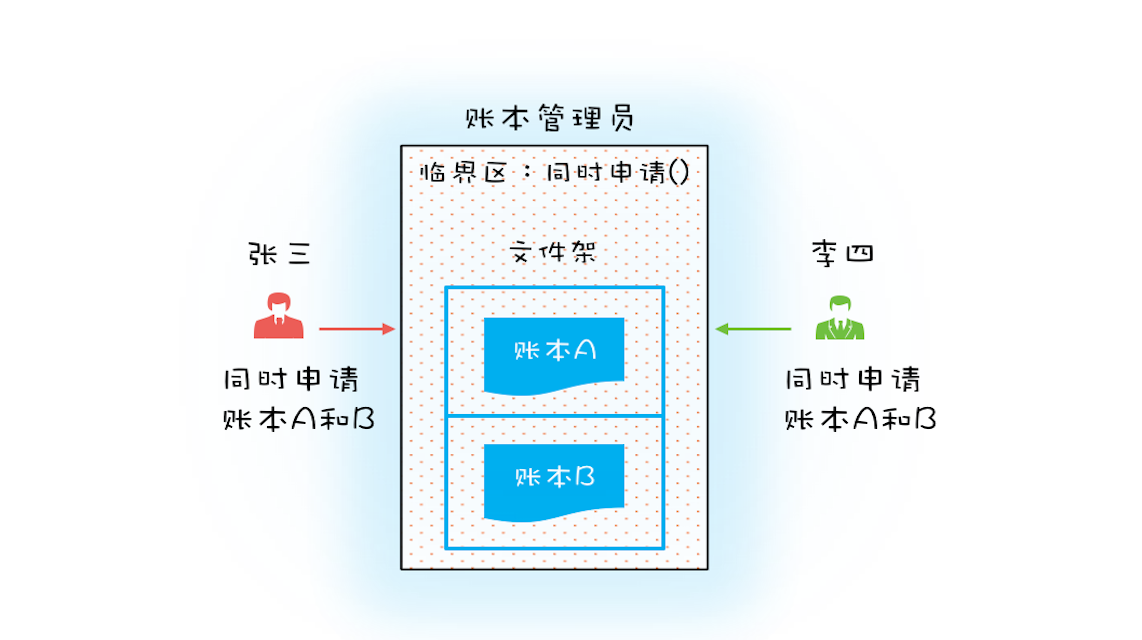


图 5通过账本管理员拿账本

class Allocator{

private List<Object> als = new ArrayList<>();

//一次性申请所有资源

synchronized boolean apply(Object from,Object to){

if(als.contains(from) || als.contains(to)){

return false;

}else{

als.add(from);

als.add(to);

}

return true;

}

//归还资源

synchronized void free(Object from,Object to){

als.remove(from);

als.remove(to);

}

}

class Account{

//actr应该为单例

private Allocator actr;

private int balance;

//转账

void transfer(Account target,int amt){

//一次性申请转出账户和转入账户，直到成功

while(!actr.apply(this,target));

try{

//锁定转出账户

synchronized(this){

//锁定转入账户

synchronized(target){

if(this.balance > amt){

this.balance -= amt;

target.balance += amt;

}

}

}

}finally{

actr.free(this,target);

}

}

}

1. 不可抢占。占有部分资源的线程申请其它资源如果申请不到就主动释放，这样其它线程就可以获取

Lock锁支持超时，后面再说

1. 循环等待。按序申请资源。线性化就不会存在循环

class Account{

private int id;

private int balance;

//转账

void transfer(Account target,int amt){

Account left = this;

Account right = target;

if(this.id > target.id){

left = target;

right = this;

}

//锁定号小的账户

synchronized(left){

//锁定号大的账户

synchronized(right){

if(this.balance > amt){

this.balance -= amt;

target.balance += amt;

}

}

}

}

}

总结：

解决死锁最简单的方案：按序申请资源破坏循环等待条件；超时，一次性获取所有稍微麻烦。要评估性能

# 用“等待-通知”机制优化循环等待

**破坏占用且等待条件**

// 一次性申请转出账户和转入账户，直到成功

while(!actr.apply(this, target))

；

Apply操作耗时短并发量不大方案挺合适（循环几次几十次），反之上万次太耗CPU

解决方案：**等待-通知机制**

**用synchronized实现等待-通知机制**

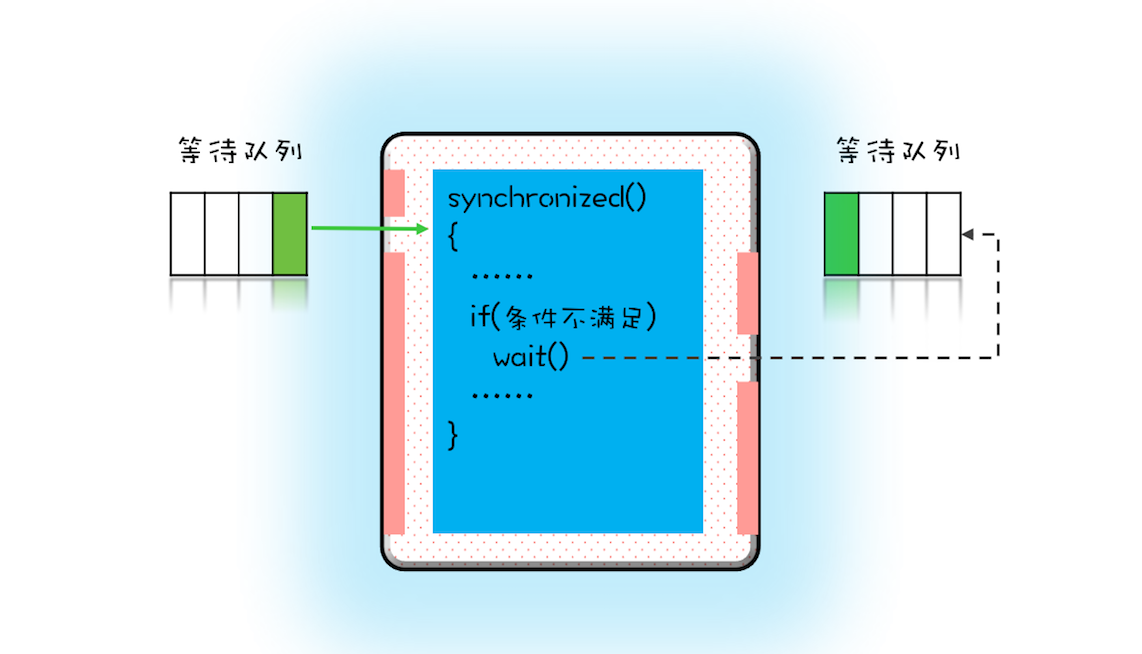
****

图 6wait()操作工作原理图

左边的等待队列和互斥锁一对一，每个互斥锁都有自己独立的等待队列。右边也是互斥锁的等待队列。

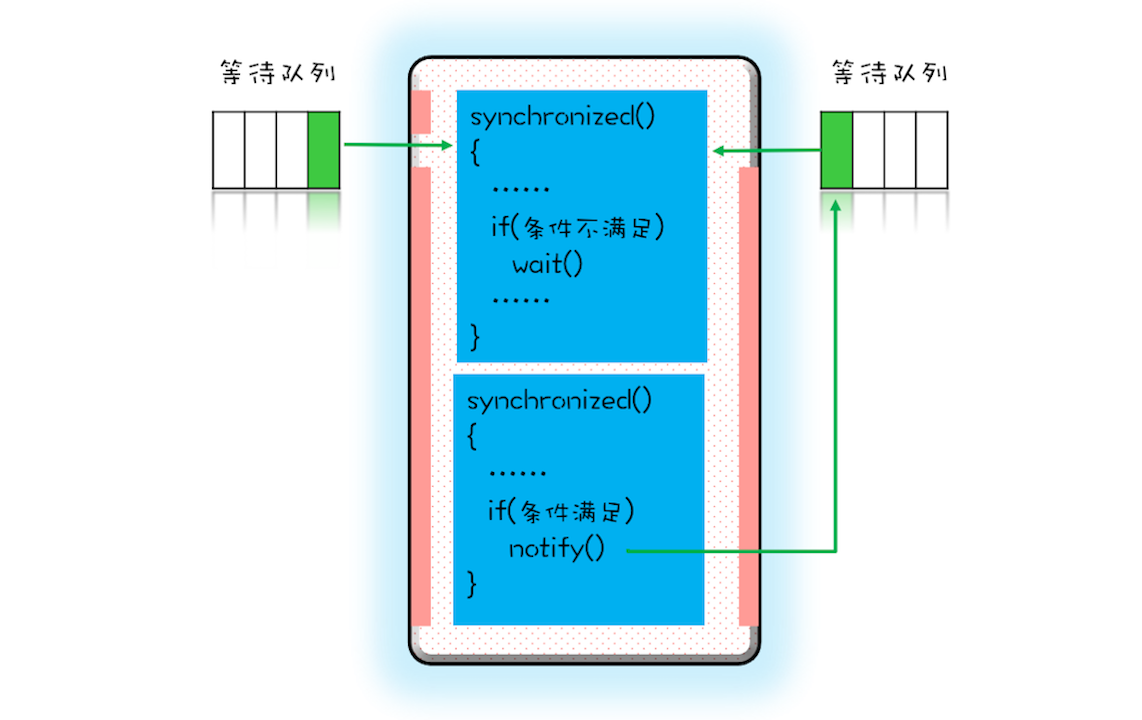


图 7 notyfy()操作工作原理图

**Notify和notifyAll的区别**：尽量使用notifyAll。Notify是随机唤醒一个，notifyAll是唤醒所有。Notify有风险，某些线程可能永远不会被通知到

class Allocator{

private List<Object> als = new ArrayList<Object>();

//一次性申请所有资源

synchronized void apply(Object from,Object to){

//经典写法

while(als.contains(from) || als.contains(to)){

try{

this.wait();

}catch(Exception e){

}

als.add(from);

als.add(to);

}

}

//归还资源

synchronized void free(Object from,Object to){

als.remove(from);

als.remove(to);

this.notifyAll();

}

}