# 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

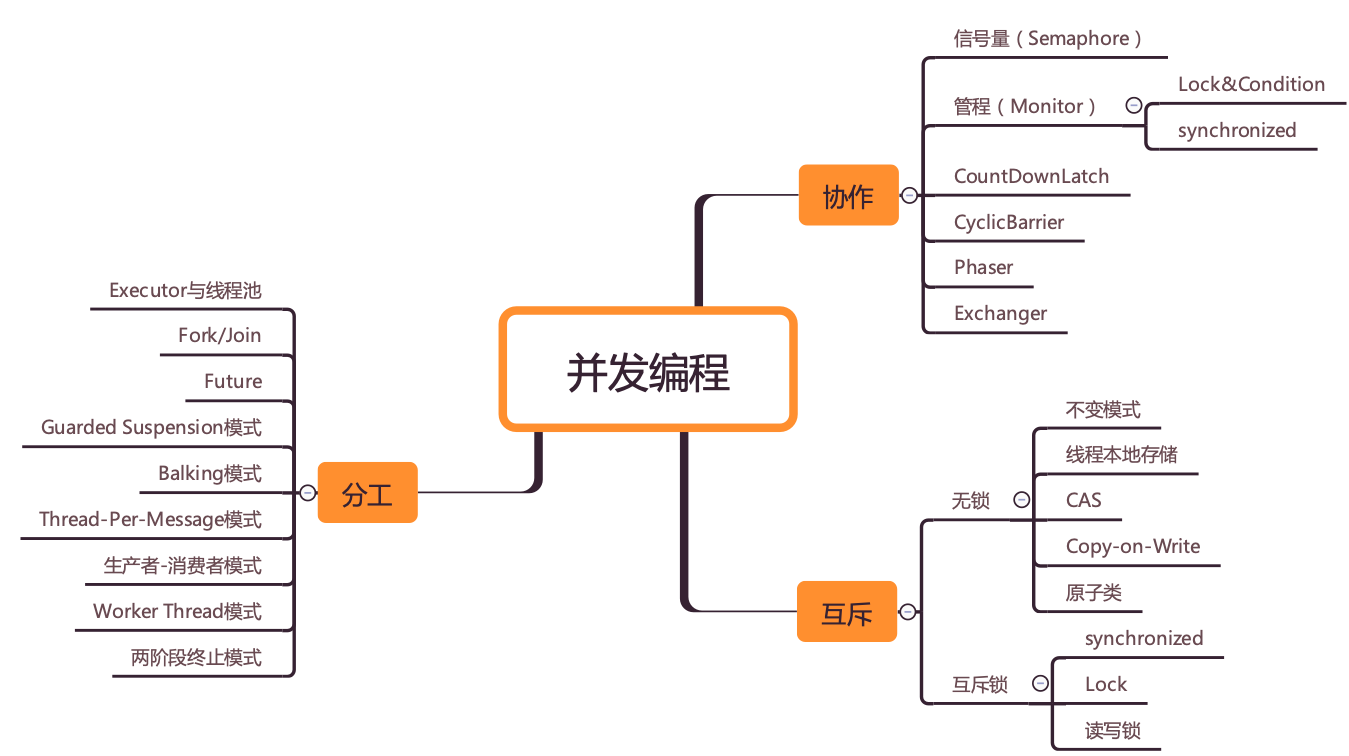
2.synchronized wait()/notify();SDK并发包condition

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();

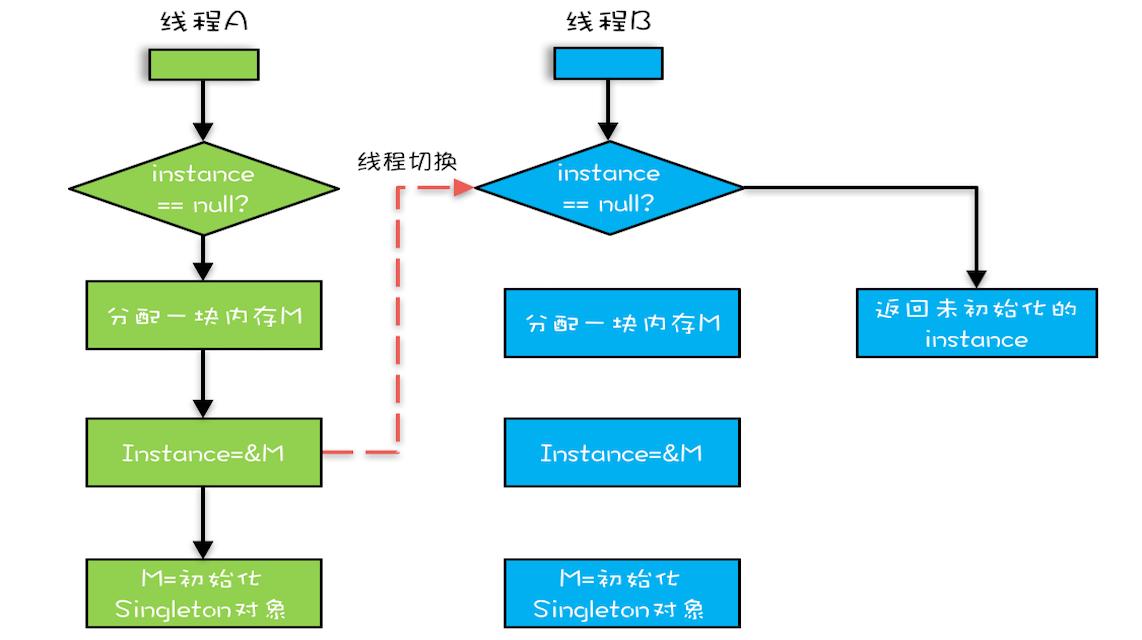
}

}

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

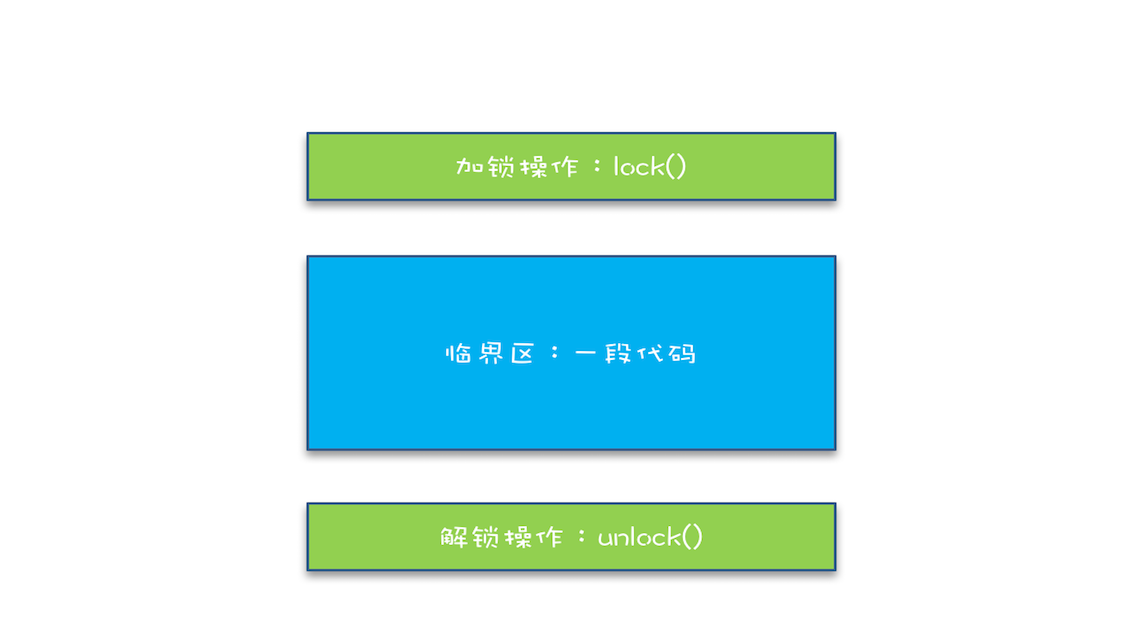


图 1简易锁模型

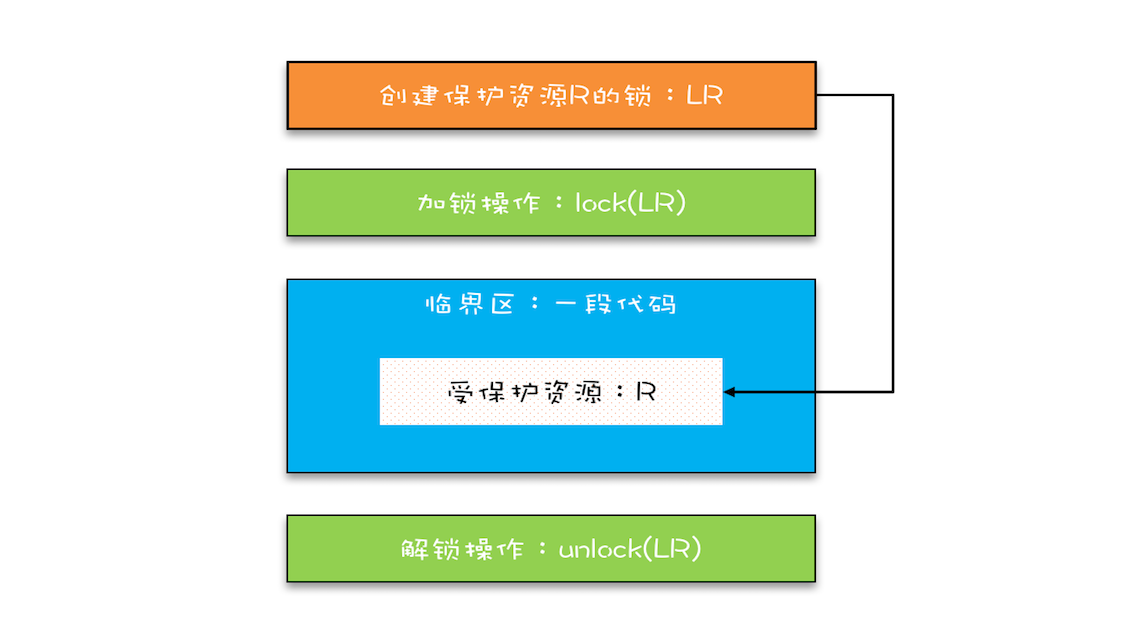


图 2改进后的锁模型

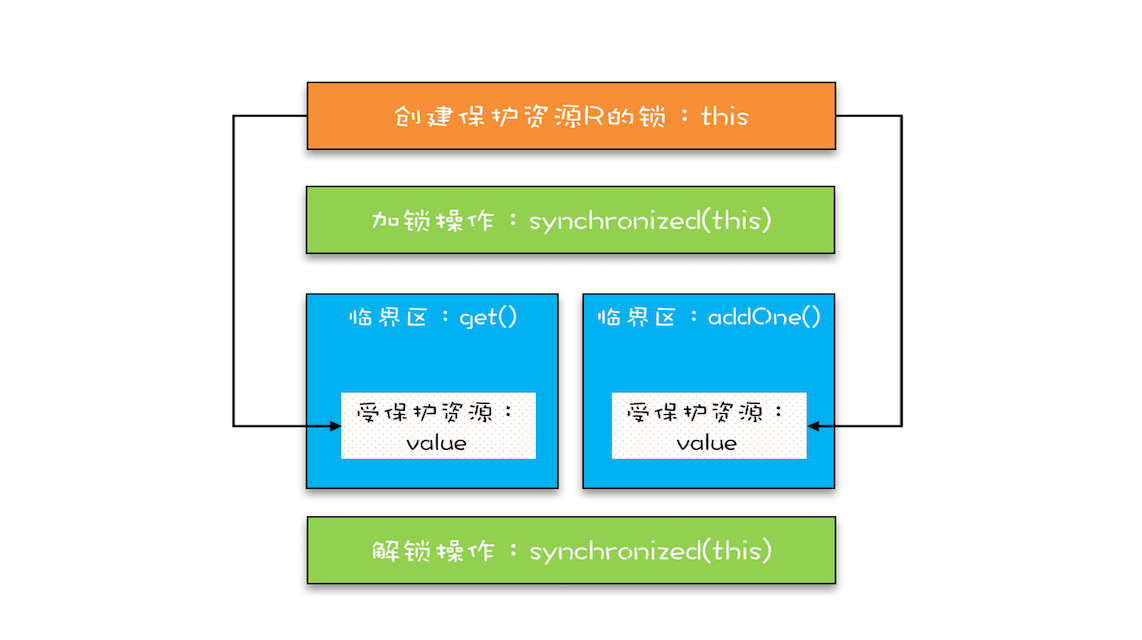


图 3保护临界区get()和addOne()的示意图

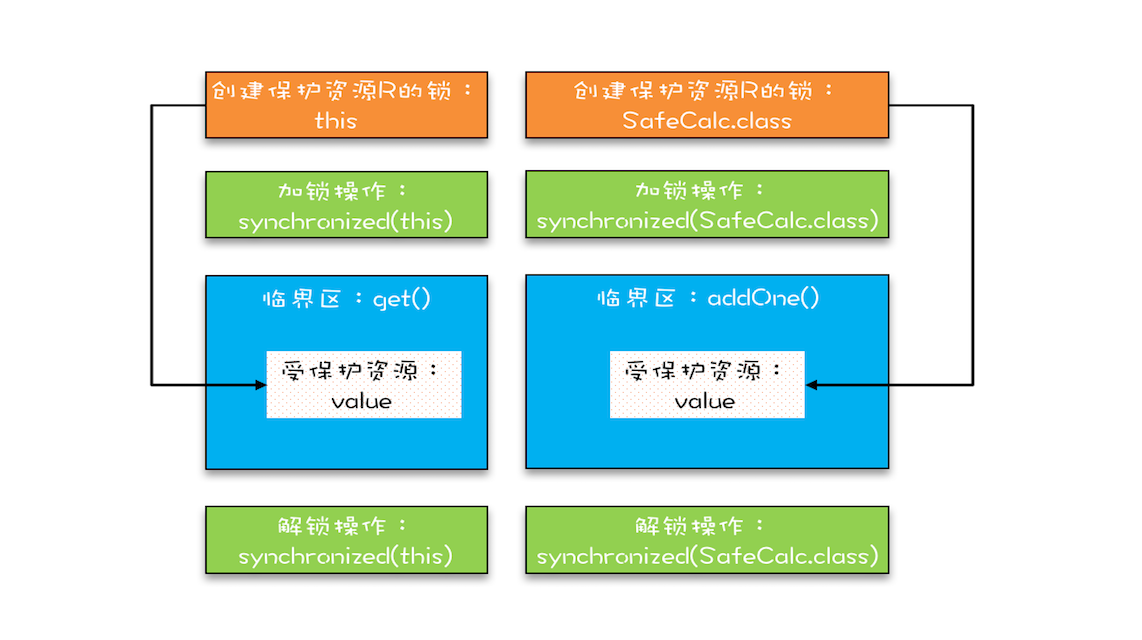


图 4两把锁保护一个资源的示意图

# 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：

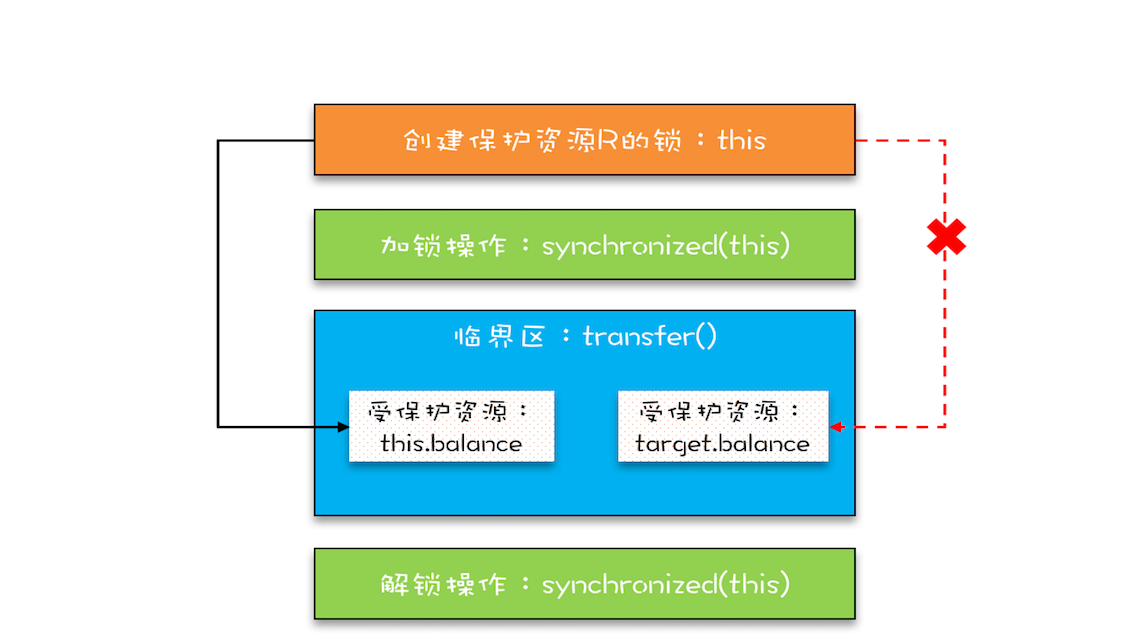


图 5锁this保护this.balance和target.balance

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;

}

}

}

}

}

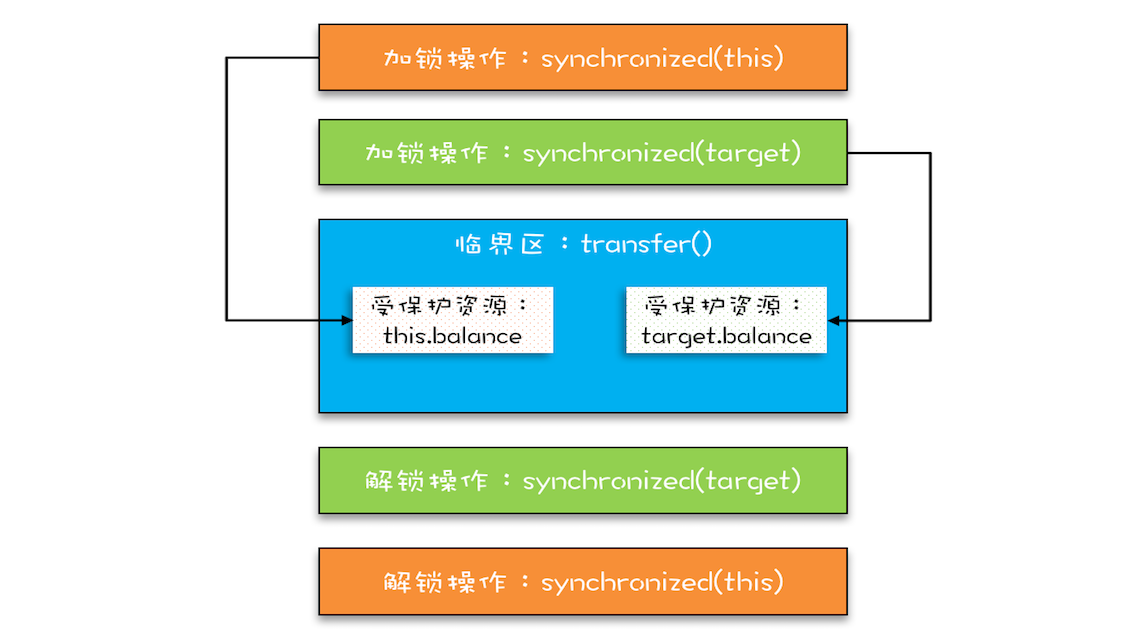


图 6两个转账操作并行示意图

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

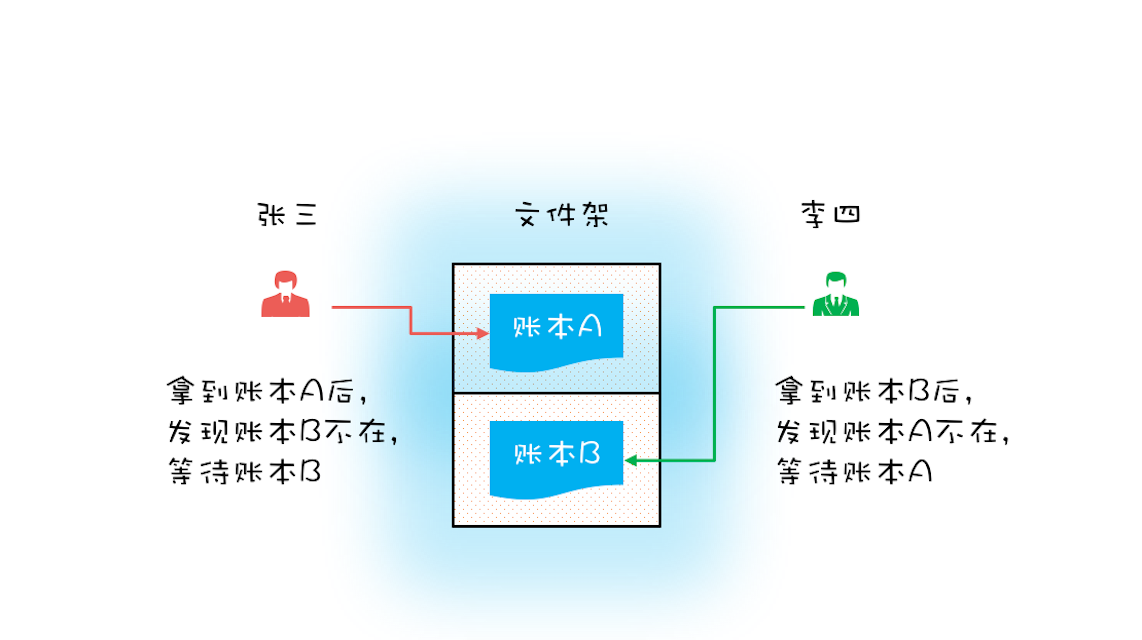


图 7转账业务中的“死等”

**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

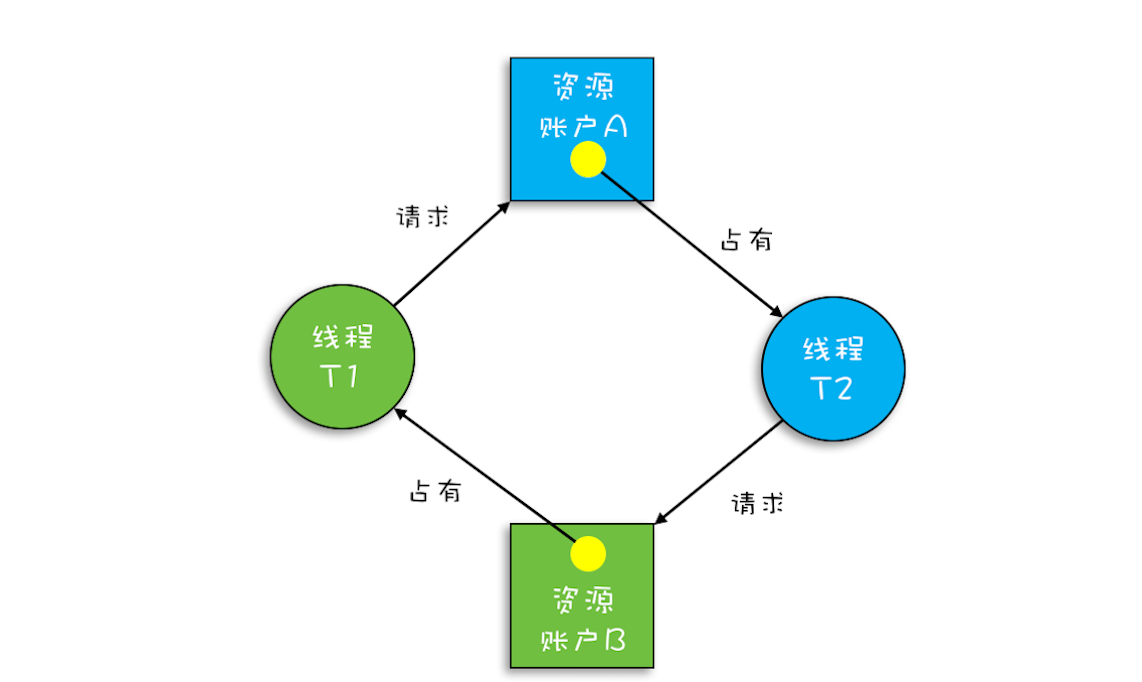


图 8转账发生死锁时的资源分配图

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

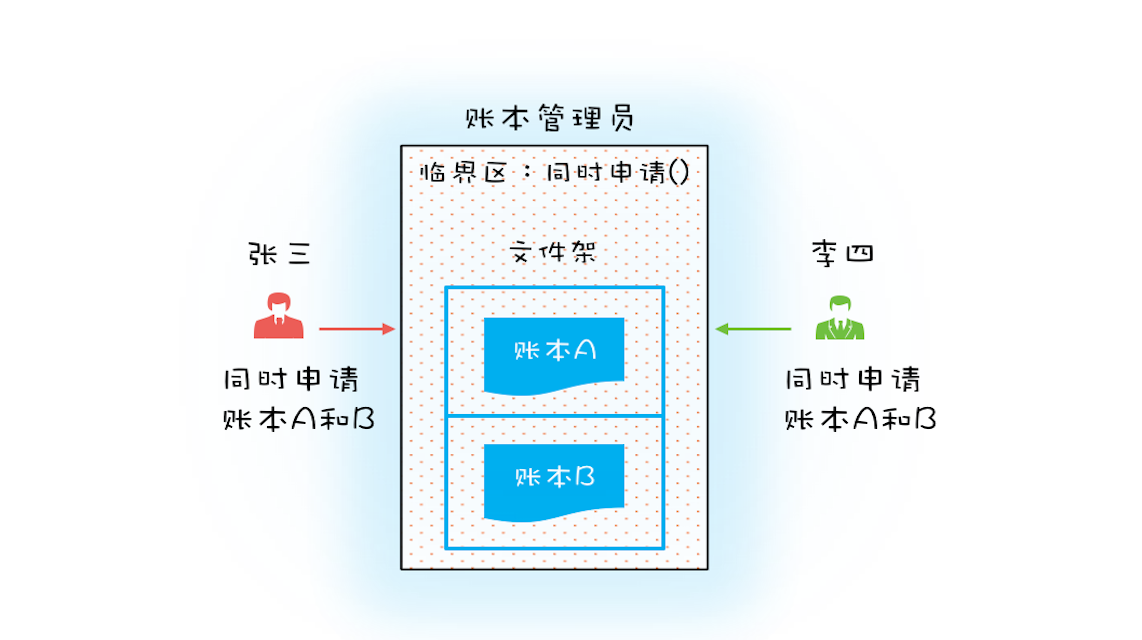


图 9通过账本管理员拿账本

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**

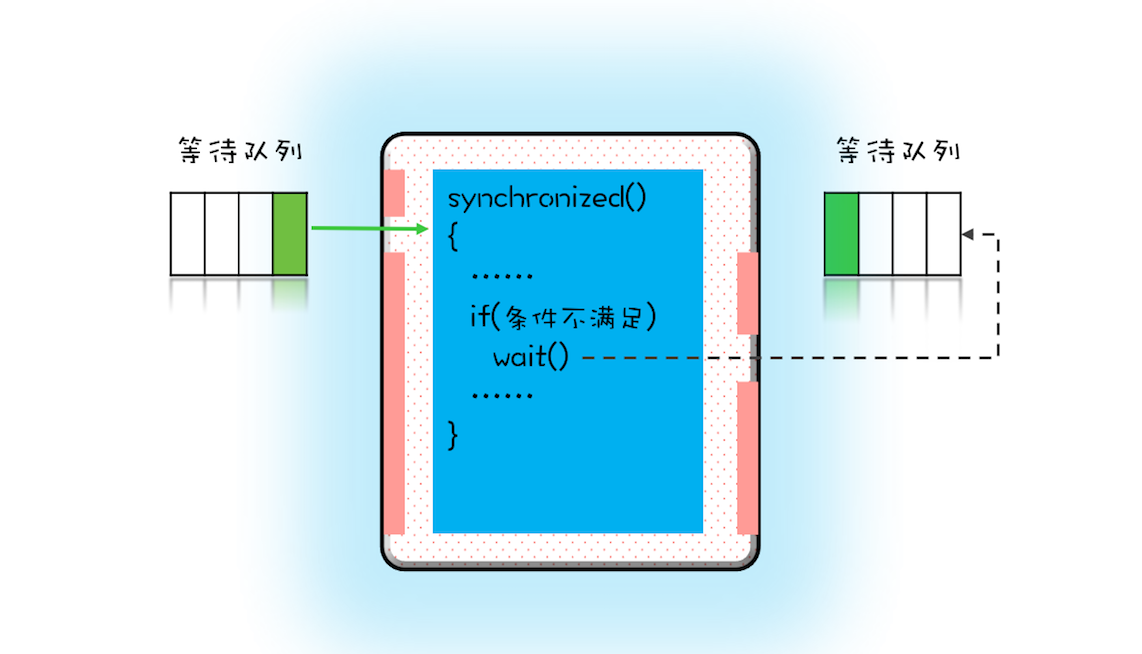
****

图 10wait()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。

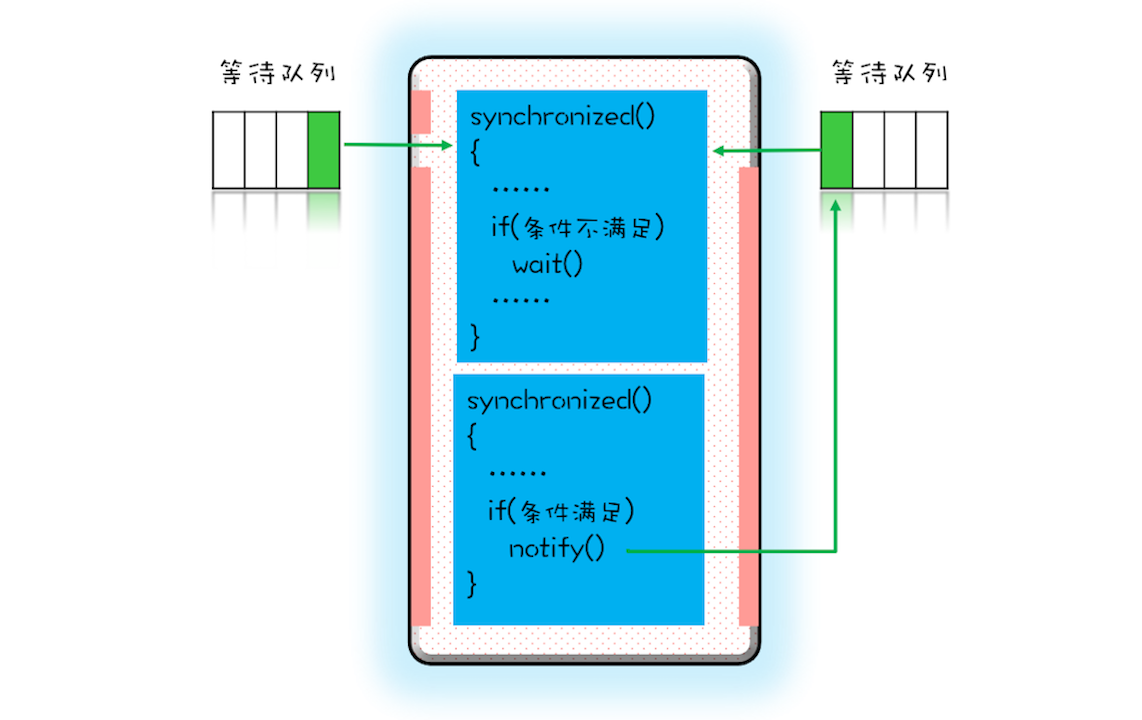


图 11 notyfy()操作工作原理图

**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();

}

}

# 安全性，活跃性，性能问题

**安全性问题：**

线程安全：本质上就是正确性，按照我们期望的执行（有并发bug的都是没按照我们期望的执行）

数据竞争：当多个线程同时访问同一数据，并且至少一个线程会写这个数据，不采取任何保护措施导致了并发bug

竞态条件：程序的执行结果依赖线程的执行顺序

**活跃性问题：**

死锁：

活锁：线程活跃彼此谦让，一直不能获得所有资源执行

饥饿：一直无法访问资源而无法继续执行（公平分配资源比较靠谱）

**性能问题：**

Java SDK并发包里之所以有那么多东西，有很大部分原因是要提升特定领域的性能

。方案分两种：1、使用无锁的算法和数据结构（线程本地存储，copy on write，乐观锁）；

1. 减少锁持有的时间。减少串行化提高并行度（使用细粒度的锁ConcurrentHashMap）

总结：

并发编程微观上涉及原子性，可见性，有序性问题，宏观上表现为安全性、活跃性和性能问题

课后思考：

void addIfNotExist(Vector v,

Object o){

if(!v.contains(o)) {

v.add(o);

}

}

代码出现了竞态条件。尽管vector是线程安全的容器，意味着方法本身原子性，可见性，有序性问题不会出现，但是contains和add方法的简单组合并不线程安全，contains方法执行后，可能某线程已经将该对象添加进去了，即竞态条件

# 管程：并发编程的万能钥匙

**管程**：管理共享变量以及对共享变量的操作过程，让他们支持并发。

管程和信号量是等价的，用管程能实现信号量，用信号量能实现管程

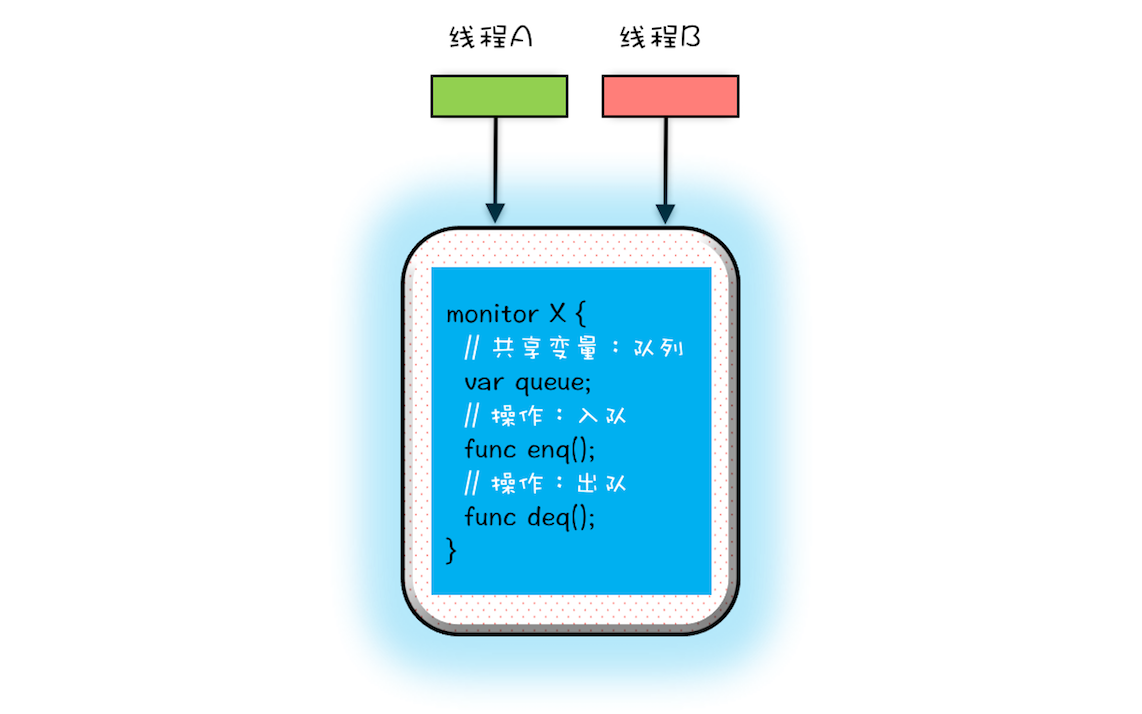


图 12 管程模型代码化语义

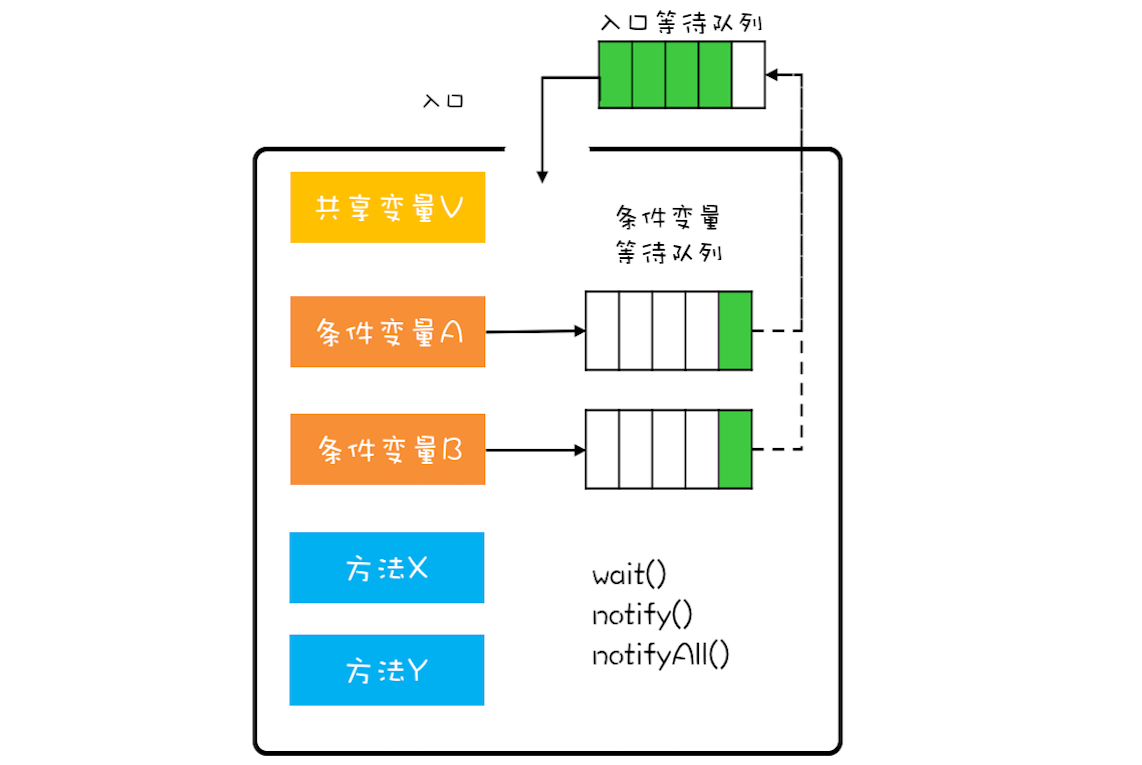


图 13MESA管程模型

阻塞队列伪代码

public class BlockingQueue<T> {

private final Lock lock = new ReentrantLock();

private final Condition notFull = lock.newCondition();

private final Condition notEmpty = lock.newCondition();

void enq(T x){

lock.lock();

try {

while (队列已满) {

notFull.await();

}

//入队操作

//入队后从非空等待队列中唤醒一个线程，出队

notEmpty.signal();

}finally {

lock.unlock();

}

}

void deq(){

lock.lock();

try {

while(队列已空){

notEmpty.await();

}

//出队操作

//出队后从不满等待队列里唤醒一个线程，入队

notFull.signal();

}finally {

lock.unlock();

}

}

}

Wait()正确使用姿势

While(条件不满足){

Wait();

}

尽量使用notifyAll();

使用notify情况：

1. 所有等待线程拥有相同的等待条件
2. 所有等待线程被唤醒后，执行相同的操作
3. 只需要唤醒一个线程

Synchronized属于内置管程，只支持一个条件变量

# Java线程（线程的生命周期）

**通用的线程生命周期：**

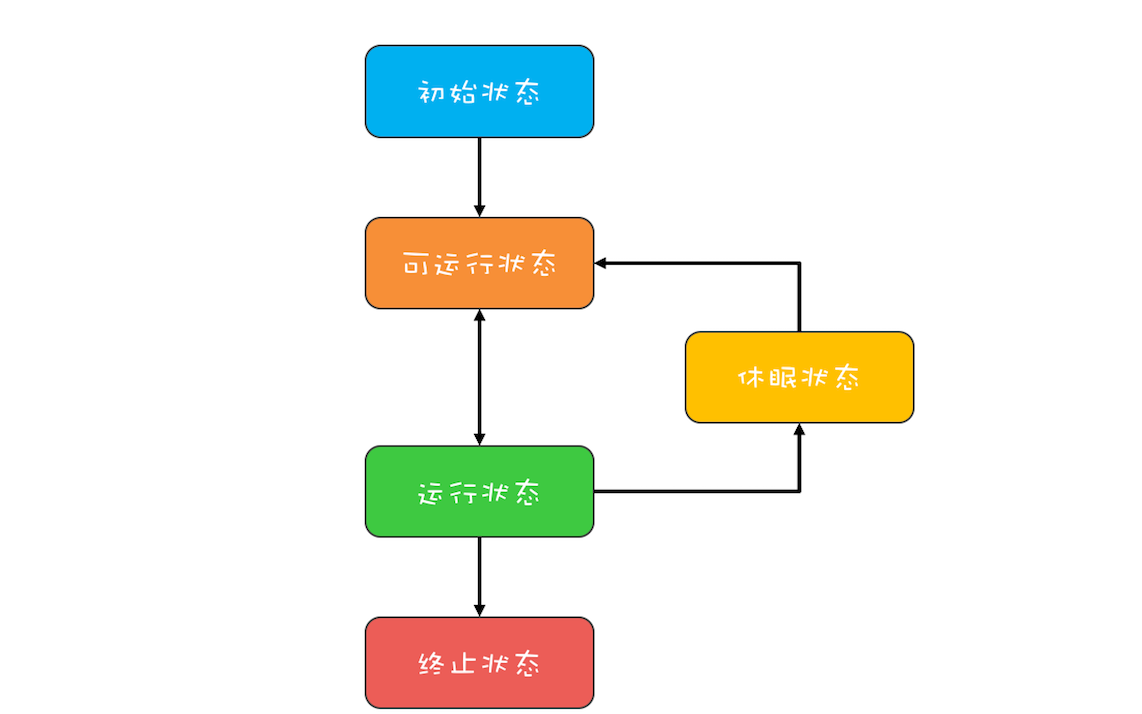


图 14 通用线程状态转换图-五态模型

Java中的线程生命周期

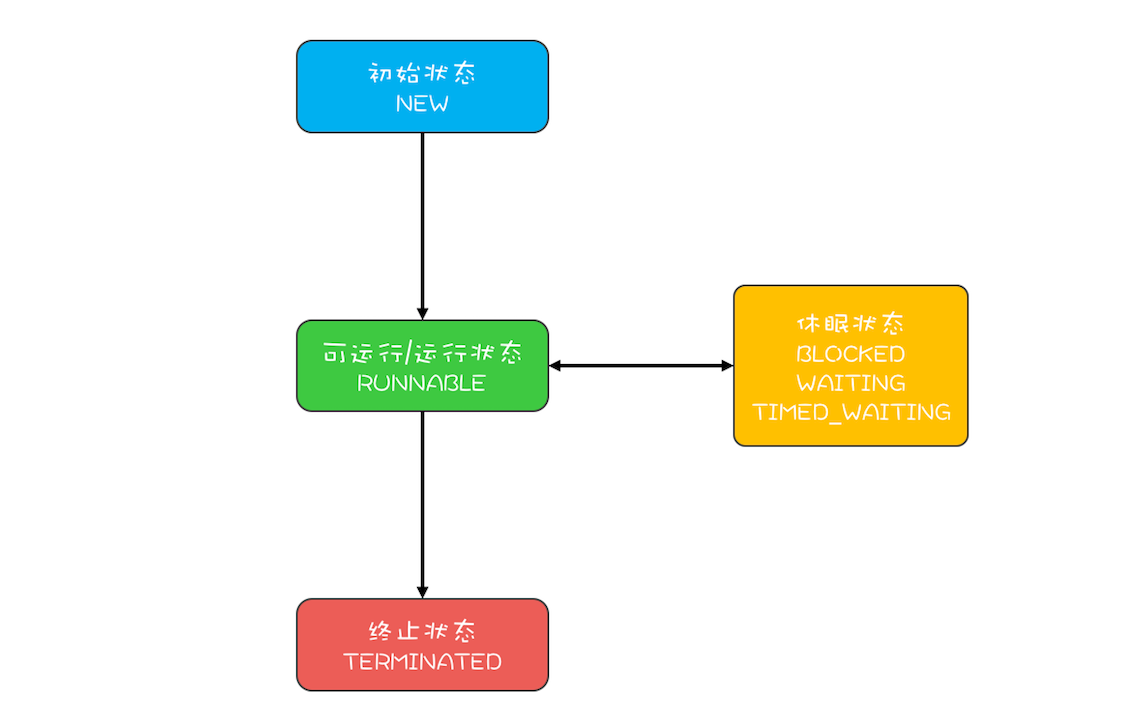


图 15java中线程状态转换图

**RUNNABLE与BLOCKED的状态转换**

RUNNABLE->BLOCKED：线程等待synchronized的隐式锁；

说明：调用阻塞式API时，操作系统层面会转到BLOCKED，java层面认为和等待CPU资源一样，都是RUNNABLE状态

**RUNNABLE与WAITING的状态转换**

1、synchronized，无参wait方法

2、线程无参join方法

3、LockSupport.park()

**RUNNABLE与TIMED\_WAITING的状态转换**

1. 带参数sleep方法
2. Synchronized隐式锁，超时参数wait方法
3. 超时参数join方法
4. 超时参数LockSupport.parkNanos方法
5. 超时参数LockSupport.parkUntil方法

**NEW到RUNNABLE状态**

New出来是NEW状态

调用start方法即可

**RUNNABLE到TERMINATED状态**

1. 正常执行完run方法
2. Stop方法（不再使用，显示锁不能释放）
3. Interrupt方法：

知识通知线程，线程有机会执行一些后续操作，也可以无视这个通知

异常手段：

线程处于waiting，timed\_waiting,先变成runnable，再抛出Interrrupt异常

线程处于RUNNABLE状态，阻塞IO

Java.nio.channels.InterruptibleChannel -> java.nio.channels.ClosedByInterruptException

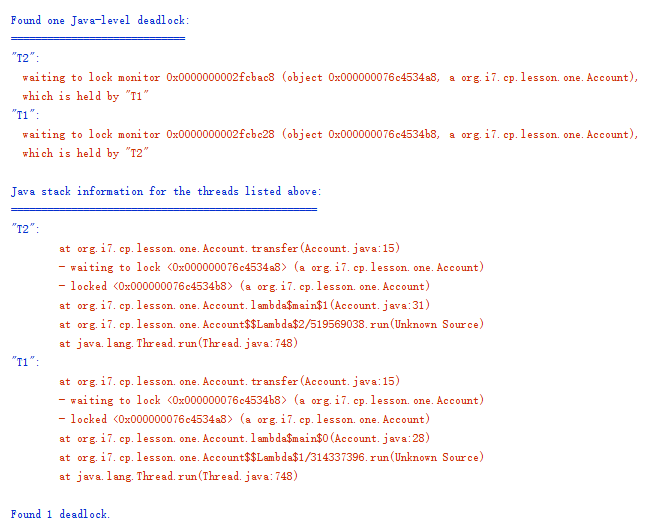
Java.nio.channels.Selector selector立即返回

主动监测：

线程处于RUNNABLE状态，并且没有阻塞在I/O操作上，isInterrupted方法监测

**总结：**

可以用jstack 或者java VisualVM可视化工具将线程栈导出来



# Java线程（创建多少个线程合适）

使用多线程，是为了提高CPU和I/O的综合利用率。

CPU密集型：理论上多少个CPU核多少个线程是最合适的，工程上一般要多一个

I/O密集型：单核：最佳线程数 =1 +（I/O 耗时 / CPU 耗时）；多核：最佳线程数 =核数\*（1 +（I/O 耗时 / CPU 耗时））

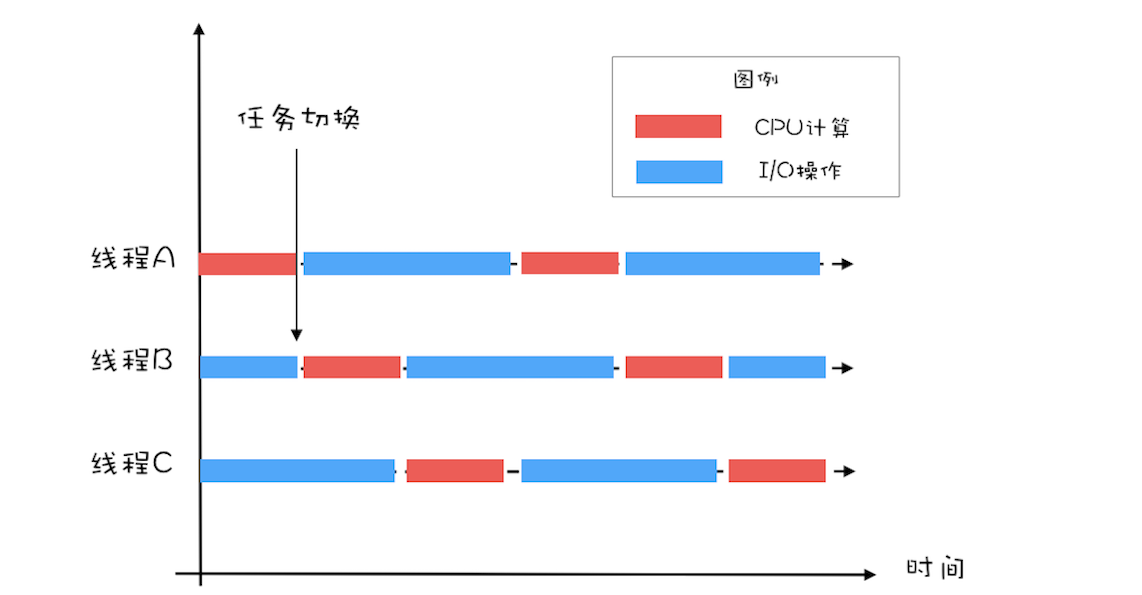


图 16单核三线程切换

# java线程（为什么局部变量是线程安全的）

了解下线程栈，以及为什么子线程不能访问方法里的局部变量

# 如何用面向对象思想写好并发程序

java语言，面向对象让并发编程变得简单

1. 封装共享变量

将共享变量作为对象属性封装在内部，对所有公共方法制定并发访问策略

public class Counter {

private long value;

synchronized long get(){

return value;

}

synchronized long addOne(){

return ++value;

}

}

对于不会发生变化的共享变量，建议用final修饰（避免并发问题，也可以表明设计意图）

二、识别共享变量间的约束条件

public class SafeWM {

// 库存上限

private final AtomicLong upper =

new AtomicLong(0);

// 库存下限

private final AtomicLong lower =

new AtomicLong(0);

// 设置库存上限

void setUpper(long v){

upper.set(v);

}

// 设置库存下限

void setLower(long v){

lower.set(v);

}

// 省略其他业务代码

}

代码问题：忽视了上下限的关系

public class SafeWM {

// 库存上限

private final AtomicLong upper =

new AtomicLong(0);

// 库存下限

private final AtomicLong lower =

new AtomicLong(0);

// 设置库存上限

void setUpper(long v){

// 检查参数合法性

if (v < lower.get()) {

throw new IllegalArgumentException();

}

upper.set(v);

}

// 设置库存下限

void setLower(long v){

// 检查参数合法性

if (v > upper.get()) {

throw new IllegalArgumentException();

}

lower.set(v);

}

// 省略其他业务代码

}

竞态条件出现

三、制定并发访问策略

方案：

1. 避免共享：利用线程本地存储，以及为每个任务分配独立的线程
2. 不变模式：Actor、CSP、函数式编程
3. 管程及其他同步工具：管程是万能的，但是对于很多场景使用并发包里的读写锁、并发容器等同步工具更好

原则：

1. 优先使用成熟的工具类
2. 迫不得已再使用低级同步原语，synchronized，lock，semphere等。没那么简单
3. 避免过早优化

# 理论基础模块热点答疑

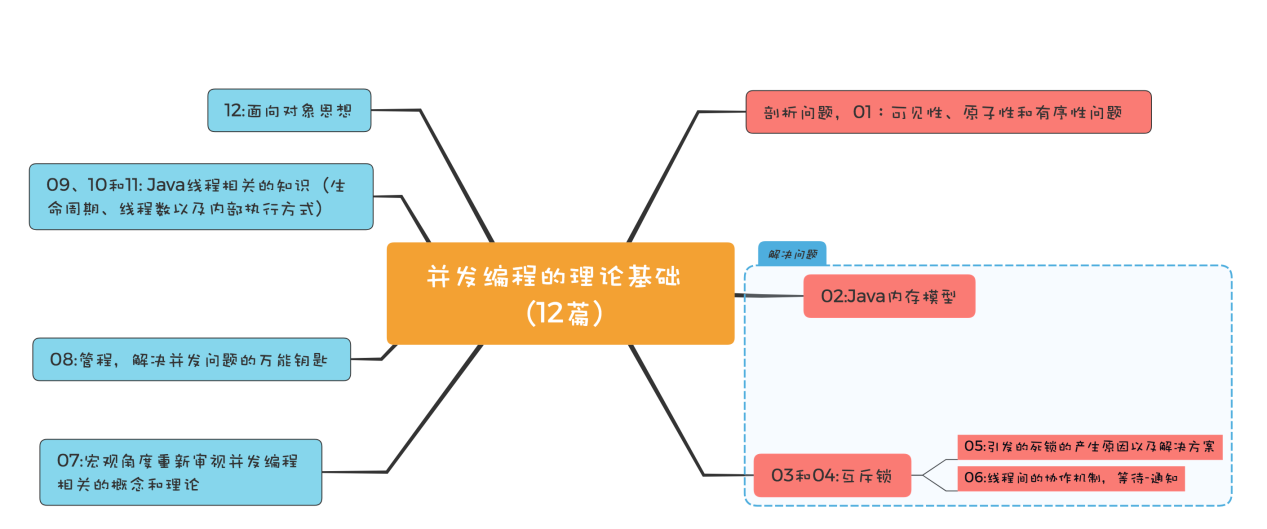


图 17并发编程理论基础模块思维导图

1. 用锁的最佳实践

锁与资源的关系应该是1:N

2、锁的性能要看场景

While(!actr.apply(this, target))和synchronized（Acount.class）

如果转账业务耗时很长，那么while循环可以提高并行度

3、注意竞态条件

4、方法调用是先计算参数

5、InterruptionException需要注意处理

6、理论值or经验值

# Lock和Condition（并发包中的管程）

**再造管程的理由：**

1、能响应中断

2、支持超时

3、非阻塞的获取锁

**如何保证可见性：**

1、顺序性

2、Valatile原则

3、传递性

可重入锁概念：

公平锁与非公平锁

**用锁的最佳实践：**

1、永远只在更新对象成员变量的地方加锁

2、永远只在访问可变的成员变量加锁

3、永远不在调用其他对象的方法时加锁

课后思考：

class Account {

private int balance;

private final Lock lock

= new ReentrantLock();

// 转账

void transfer(Account tar, int amt){

while (true) {

if(this.lock.tryLock()) {

try {

if (tar.lock.tryLock()) {

try {

this.balance -= amt;

tar.balance += amt;

} finally {

tar.lock.unlock();

}

}//if

} finally {

this.lock.unlock();

}

}//if

}//while

}//transfer

}

注意活锁

# Lock和Condition(Dubbo如何用管程实现异步转同步)

**异步与同步**：调用方是否需要等待结果？需要就是同步，不需要就是异步

同步，java默认的处理方式，让程序支持**异步有两种方式实现**：

1. 调用方创建子线程，在子线程中执行方法调用（异步调用）
2. 方法实现的时候，创建一个新的线程执行主要逻辑，主线程直接return（异步方法）

Dubbo源码分析：

// 创建锁与条件变量

private final Lock lock

= new ReentrantLock();

private final Condition done

= lock.newCondition();

// 调用方通过该方法等待结果

Object get(int timeout){

long start = System.nanoTime();

lock.lock();

try {

while (!isDone()) {

done.await(timeout);

long cur=System.nanoTime();

if (isDone() ||

cur-start > timeout){

break;

}

}

} finally {

lock.unlock();

}

if (!isDone()) {

throw new TimeoutException();

}

return returnFromResponse();

}

// RPC 结果是否已经返回

boolean isDone() {

return response != null;

}

// RPC 结果返回时调用该方法

private void doReceived(Response res) {

lock.lock();

try {

response = res;

if (done != null) {

done.signal();

}

} finally {

lock.unlock();

}

}

# Semphore

**信号量模型**：一个计数器、一个等待队列、三个方法

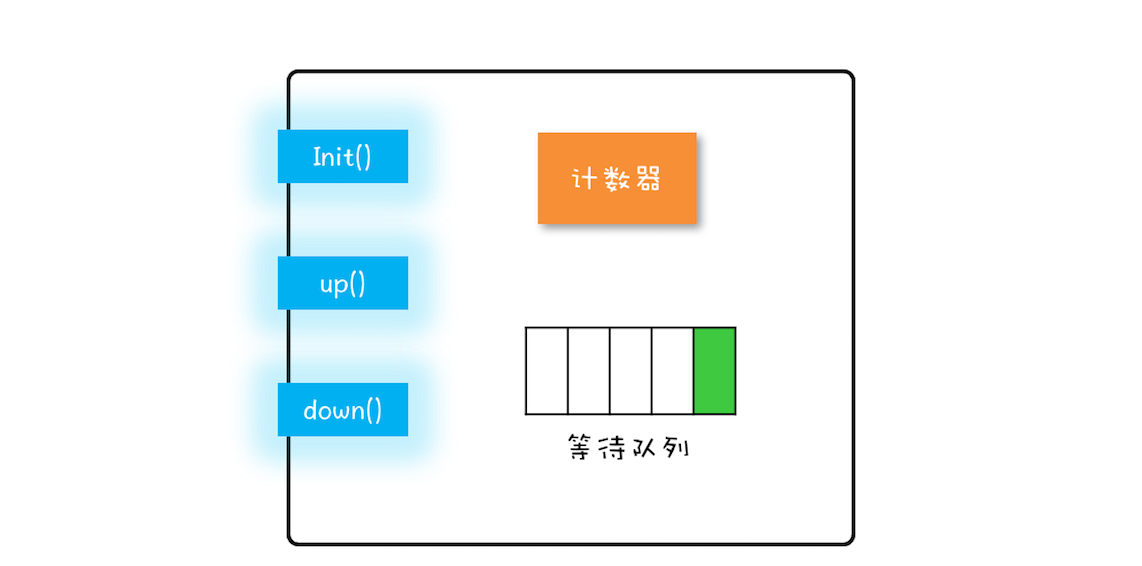


图 18信号量模型图

class Semaphore{

// 计数器

int count;

// 等待队列

Queue queue;

// 初始化操作

Semaphore(int c){

this.count=c;

}

//

void down(){

this.count--;

if(this.count<0){

// 将当前线程插入等待队列

// 阻塞当前线程

}

}

void up(){

this.count++;

if(this.count<=0) {

// 移除等待队列中的某个线程 T

// 唤醒线程 T

}

}

}

**如何使用信号量：**

static int count;

// 初始化信号量

static final Semaphore s

= new Semaphore(1);

// 用信号量保证互斥

static void addOne() {

s.acquire();

try {

count+=1;

} finally {

s.release();

}

}

**快速实现一个限流器：**

Semphore可以允许多个线程访问一个临界区

class ObjPool<T, R> {

final List<T> pool;

// 用信号量实现限流器

final Semaphore sem;

// 构造函数

ObjPool(int size, T t){

pool = new Vector<T>(){};

for(int i=0; i<size; i++){

pool.add(t);

}

sem = new Semaphore(size);

}

// 利用对象池的对象，调用 func

R exec(Function<T,R> func) {

T t = null;

sem.acquire();

try {

t = pool.remove(0);

return func.apply(t);

} finally {

pool.add(t);

sem.release();

}

}

}

// 创建对象池

ObjPool<Long, String> pool =

new ObjPool<Long, String>(10, 2);

// 通过对象池获取 t，之后执行

pool.exec(t -> {

System.out.println(t);

return t.toString();

});

# ReadWriteLock：快速实现一个完备的缓存

**什么是读写锁？**

1. 允许多个线程同时读共享变量；
2. 只允许一个线程写共享变量
3. 如果一个线程正在进行写操作，此时禁止读线程读共享变量

**实现一个缓存的按需加载**

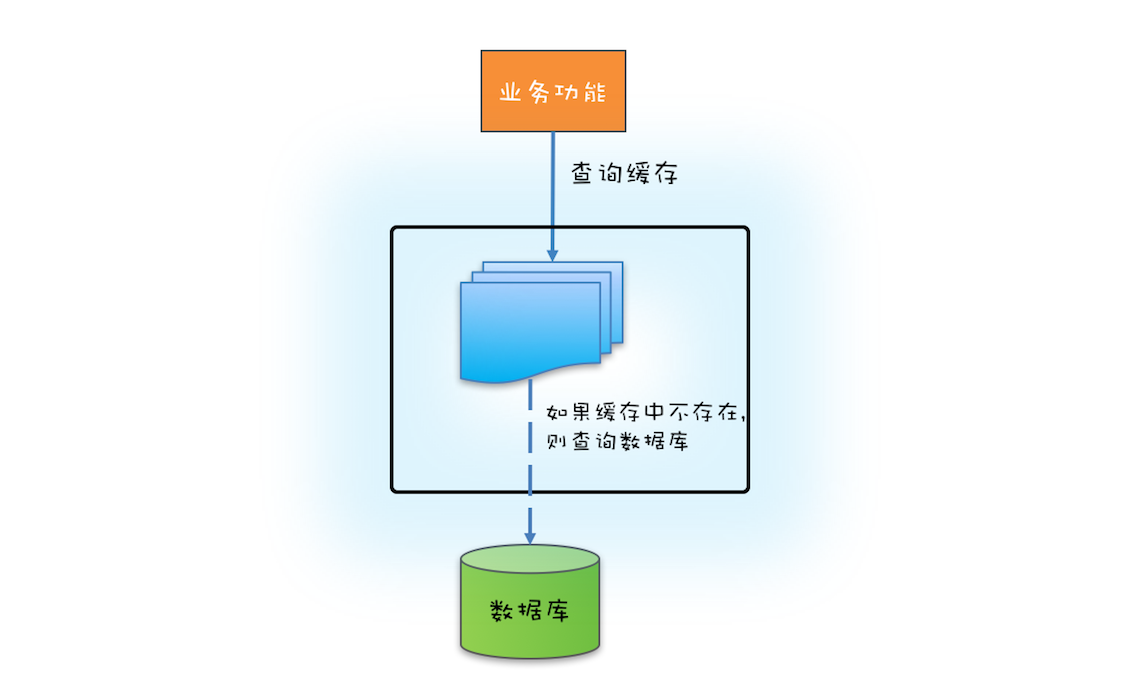


图 19 按需加载示意图

public class Cache<K,V> {

private final Map<K,V> m = new HashMap<>();

private final ReadWriteLock rwl = new ReentrantReadWriteLock();

private final Lock r = rwl.readLock();

private final Lock w = rwl.writeLock();

public V get(K key){

V v = null;

//读缓存

r.lock();

try {

v = m.get(key);

}finally{

r.unlock();

}

//缓存中存在，返回

if(v!=null){

return v;

}

//缓存中不存在，查询数据库

w.lock();

try{

//再次验证，可能已经有其它线程查询过数据库

v = m.get(key);

if(v == null){

//查询数据库

//v=----

m.put(key,v);

}

}finally {

w.unlock();

}

return v;

}

}

**锁的升级**：先获取读锁，然后再升级为写锁

**锁的降级**：先获取写锁，再降级为读锁

**读写锁不允许升级，但是降级确实允许的**

class CachedData {

Object data;

volatile boolean cacheValid;

final ReadWriteLock rwl =

new ReentrantReadWriteLock();

// 读锁

final Lock r = rwl.readLock();

// 写锁

final Lock w = rwl.writeLock();

void processCachedData() {

// 获取读锁

r.lock();

if (!cacheValid) {

// 释放读锁，因为不允许读锁的升级

r.unlock();

// 获取写锁

w.lock();

try {

// 再次检查状态

if (!cacheValid) {

data = ...

cacheValid = true;

}

// 释放写锁前，降级为读锁

// 降级是可以的

r.lock(); ①

} finally {

// 释放写锁

w.unlock();

}

}

// 此处仍然持有读锁

try {use(data);}

finally {r.unlock();}

}

}

**总结：**

读写锁类似ReentrantLock，支持公平非公平。只有写锁有条件变量，读锁不支持条件变量；

记得引入超时机制

# StampedLock:有没有比读写锁更快的锁

**StampedLock支持的三种模式：**

写锁、悲观读锁，乐观读；乐观读这个操作无锁

**先乐观读，再悲观读**

public class Point {

private int x,y;

private StampedLock sl = new StampedLock();

//计算到原点的距离

public double distanceFromOrigin(){

//乐观读

long stamp = sl.tryOptimisticRead();

//读入局部变量

//读入过程可能被修改

int curX = x;

int curY = y;

if(!sl.validate(stamp)){

//升级为悲观读锁

stamp = sl.readLock();

try{

curX = x;

curY = y;

}finally {

sl.unlockRead(stamp);

}

}

return Math.sqrt(curX\*curX +curY\*curY);

}

}

**StampedLock使用注意事项**

1、StampedlLock的功能仅仅是ReadWriteLock的子集

2、StampedLock不支持重入

3、StampedLock的悲观读写锁都不支持条件变量

4、如果阻塞在StampedLock的readLock()或writeLock()，此时调用interrupt方法，会导致CPU飙升。如果需要支持中断功能，一定使用readLockInterruptibly,writeLockInterruptibly

final StampedLock lock

= new StampedLock();

Thread T1 = new Thread(()->{

// 获取写锁

lock.writeLock();

// 永远阻塞在此处，不释放写锁

LockSupport.park();

});

T1.start();

// 保证 T1 获取写锁

Thread.sleep(100);

Thread T2 = new Thread(()->

// 阻塞在悲观读锁

lock.readLock()

);

T2.start();

// 保证 T2 阻塞在读锁

Thread.sleep(100);

// 中断线程 T2

// 会导致线程 T2 所在 CPU 飙升

T2.interrupt();

T2.join();

**StampedLock读模板**：

final StampedLock sl =

new StampedLock();

// 乐观读

long stamp =

sl.tryOptimisticRead();

// 读入方法局部变量

......

// 校验 stamp

if (!sl.validate(stamp)){

// 升级为悲观读锁

stamp = sl.readLock();

try {

// 读入方法局部变量

.....

} finally {

// 释放悲观读锁

sl.unlockRead(stamp);

}

}

// 使用方法局部变量执行业务操作

......

**StampedLock写模板：**

long stamp = sl.writeLock();

try {

// 写共享变量

......

} finally {

sl.unlockWrite(stamp);

}

**StampedLock支持锁的降级和升级**

private double x, y;

final StampedLock sl = new StampedLock();

// 存在问题的方法

void moveIfAtOrigin(double newX, double newY){

long stamp = sl.readLock();

try {

while(x == 0.0 && y == 0.0){

long ws = sl.tryConvertToWriteLock(stamp);

if (ws != 0L) {

x = newX;

y = newY;

break;

} else {

sl.unlockRead(stamp);

stamp = sl.writeLock();

}

}

} finally {

sl.unlock(stamp);

}

成功升级为写锁后有bug

# CountDownLatch和CycliBarrier：如何让多线程步调一致

**用countdownlatch实现线程等待**

// 创建 2 个线程的线程池

Executor executor =

Executors.newFixedThreadPool(2);

while(存在未对账订单){

// 计数器初始化为 2

CountDownLatch latch =

new CountDownLatch(2);

// 查询未对账订单

executor.execute(()-> {

pos = getPOrders();

latch.countDown();

});

// 查询派送单

executor.execute(()-> {

dos = getDOrders();

latch.countDown();

});

// 等待两个查询操作结束

latch.await();

// 执行对账操作

diff = check(pos, dos);

// 差异写入差异库

save(diff);

}

**用cyclicbarrier实现线程同步**

// 订单队列

Vector<P> pos;

// 派送单队列

Vector<D> dos;

// 执行回调的线程池

Executor executor =

Executors.newFixedThreadPool(1);

final CyclicBarrier barrier =

new CyclicBarrier(2, ()->{

executor.execute(()->check());

});

void check(){

P p = pos.remove(0);

D d = dos.remove(0);

// 执行对账操作

diff = check(p, d);

// 差异写入差异库

save(diff);

}

void checkAll(){

// 循环查询订单库

Thread T1 = new Thread(()->{

while(存在未对账订单){

// 查询订单库

pos.add(getPOrders());

// 等待

barrier.await();

}

});

T1.start();

// 循环查询运单库

Thread T2 = new Thread(()->{

while(存在未对账订单){

// 查询运单库

dos.add(getDOrders());

// 等待

barrier.await();

}

});

T2.start();

}

CyclicBarrier的回调函数执行在一个回合里最后执行await()的线程上，而且同步调用回调函数check()，调用完check之后，才会开始第二回合。所以check如果不另开一线程异步执行，就起不到性能优化的作用了。