线程一共有七个状态：

1.初始化状态

2.准备运行状态（就绪状态）

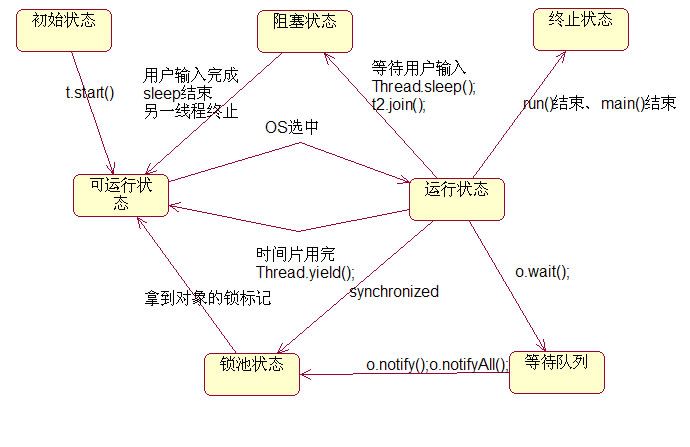
3.运行状态

4.终止状态

5.sleep睡眠状态

6.wait等待状态

7.IO阻塞状态



中断线程方法：

安全终止线程：interrupted()

stop()现在停止使用了，因为停止的时候并没有释放锁，一直在那停着

# 创建线程的方法

## 继承Thread类

重写run方法

## 实现Runnable接口

必须完成run函数

使用方式：MyThreadImp 为实现的Runable类

MyThreadImp myThreadImp = new MyThreadImp();

Thread thread = new Thread(myThreadImp);

## 匿名内部类方式

new Thread(){

public void run() {

System.out.println("hah");

};

}.start();

执行结果：hah

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("kaka");

}

}).start();

执行结果：kaka

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("runnable");

}

}){

@Override

public void run() {

System.out.println("sub");

}

}.start();

执行结果：sub 因为子类覆盖了父类的run方法

## 4.带返回值的线程

演示代码：

|  |
| --- |
| import java.util.concurrent.Callable;  import java.util.concurrent.FutureTask;  public class ThreadDemo2 implements Callable<Integer> {  public static void main(String[] args) throws Exception {  ThreadDemo2 demo = new ThreadDemo2();  FutureTask<Integer> task = new FutureTask<Integer>(demo);  Thread t = new Thread(task);  t.start();  System.out.println("我先干点别的");  Integer result = task.get();  System.out.println("线程打印的结果为：" + result);  }  @Override  public Integer call() throws Exception {  System.out.println("!!!");  Thread.sleep(3000);  return 1;  }  } |

## 5.定时器（quartz）

|  |
| --- |
| Timer timer = new Timer();  timer.schedule(new TimerTask() {  @Override  public void run() {  System.out.println("start");  }  }, 0, 1000); |

## 6.线程池实现

|  |
| --- |
| //创建10个线程  ExecutorService threadPool = Executors.newFixedThreadPool(10);  for(int i = 0; i < 10; i++){  threadPool.execute(new Runnable() {  @Override  public void run() {  System.out.println(Thread.currentThread().getName());  }  });  }  threadPool.shutdown(); |

## 7.Lambda表达式实现

|  |
| --- |
| public static void main(String[] args) {  List<Integer> values = Arrays.asList(10,20,30,40);  int res = new ThreadDemo5().add(values);  System.out.println(res);  }  public int add(List<Integer> values){  return values.parallelStream().mapToInt(a -> a).sum();  } |

## 8.spring实现多线程

# 线程安全问题

## 线程安全问题

有以下条件会出现线程安全问题：

1.多线程环境下

2.多个线程共享一个资源

3.对资源进行非原子性（读写）操作

从java字节码的角度看线程安全问题：

value ++;操作需要经过几个步骤才可以执行完成，多个线程下会出现脏读。

在字节码里只有一行字节码不会出现安全问题的

## 活跃性问题

死锁：一个人有另一个人的资源，另一个人有这个人的资源。互相不放锁，导致死锁。

饥饿：有一个线程一直占有资源

活锁：两个线程很谦让，同时释放资源，同时占有资源

饥饿与公平：

饥饿和公平遇到的情景：

高优先级吞噬所有低优先级的CPU时间片

线程被永久堵塞一个等待进入同步代码块的状态

等待的线程永远不被唤醒

如何避免饥饿问题：

设置合理的优先级

使用锁来代替synchronized

## 性能问题

任务切换即使上下文切换

# synchronized原理与使用

synchronized 放在方法上，内置锁就是当前类的实例

内置锁：即对象锁，java中每一个对象都可以成为锁

互斥锁：一个线程获得锁，其他线程不能获得锁

修饰普通方法：

修饰普通方法只对当前对象加锁。

修饰静态方法：

修饰静态方式对任何一个此类的对象加锁。

修饰代码块：

对当前代码块进行加锁。

## 字节码底层原理：

查看class文件，javap -v class文件名，查看synchronized在字节码中的实现

在javap的字节码文件中可以看到：

monitorenter

monitorexit

在这段字节码中就是synchronized包裹的代码。

任何对象都可以作为锁，那么锁信息又存在对象的什么地方呢？

存在对象头中：

对象头中的信息：

Mark Word

Class Metadata Address

Array Length（数组有的信息）

# synchronized锁实现的锁--偏向锁--轻量级锁--重量级锁

偏向锁：

偏向锁时，Mark Word包括（线程id，Epoch，对象的分代信息，是否是偏向锁，锁标志位）。每次获取锁和释放锁会浪费资源，很多情况下，竞争锁不是由多个线程，而是由一个线程在使用。偏向锁没有竞争的情况不释放锁，一旦出现竞争便可以释放锁。

轻量级锁：

当一个线程访问同步代码块是，会将Mark Word复制一份进行修改，如果还有线程来访问，也会将Mark Word复制一份进行修改，如果锁标志位标记着有线程占用同步代码块，会出现修改失败的操作，然后会继续循环修改锁的操作，直到线程释放锁。这个循环修改锁是自旋锁。轻量级锁的优点多个线程可以同时。

重量级锁：

# 单例模式与线程安全

## 饿汉式：

public class Singleton {

//私有化构造方法

private Singleton(){}

private static Singleton instance = new Singleton();

public static Singleton getInstance(){

return instance;

}

}

是线程安全的单例模式。

## 懒汉式：

public class Singleton2 {

private Singleton2(){}

private static Singleton2 instance;

public synchronized static Singleton2 getInstance(){

if(instance == null){

instance = new Singleton2();

}

return instance;

}

}

加上synchronized是线程安全的，但是性能不能进行保证了，自旋锁比较浪费资源。

public class Singleton2 {

private Singleton2(){}

private static volatile Singleton2 instance;

public synchronized static Singleton2 getInstance(){

if(instance == null){

synchronized (Singleton2.class){

if(instance == null){

instance = new Singleton2();

}

}

}

return instance;

}

}

线程安全，性能好。

# 实现锁的方式--可重入锁--自旋锁--死锁

## 可重入锁：

### 1.方法的可重入锁

在同一个对象中，有两个方法都带有synchronized锁，当调用一个方法时，当中还调用了另一个带有synchronized方法，这就叫锁的重入。

public class KeCRS {

public synchronized void a(){

System.out.println(Thread.currentThread().getName() + "a");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

b();

}

public synchronized void b(){

System.out.println(Thread.currentThread().getName() + "b");

}

public static void main(String[] args) {

KeCRS k = new KeCRS();

new Thread(new Runnable() {

@Override

public void run() {

k.a();

}

}).start();

new Thread(new Runnable() {

@Override

public void run() {

k.b();

}

}).start();

}

}

运行结果：

Thread-0a

Thread-0b

Thread-1b

### 2.父类的可冲入锁

当存在父子类继承关系时，子类是完全可以通过“可重入锁”调用父类的同步方法。

## 自旋锁：

**自旋锁不改变线程状态，所以响应速度较快，但当并发量大时性能下降明显，因为其轮询时占用CPU，自旋锁等待时一直占用CPU。**

**public class ZXS {**

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

**new Thread(new Runnable() {**

**@Override**

**public void run() {**

**System.out.println(Thread.currentThread().getName() + "线程开始执行..");**

**try {**

**Thread.sleep(new Random().nextInt(2000));**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**System.out.println(Thread.currentThread().getName() + "线程执行完毕...");**

**}**

**}).start();**

**new Thread(new Runnable() {**

**@Override**

**public void run() {**

**System.out.println(Thread.currentThread().getName() + "线程开始执行..");**

**try {**

**Thread.sleep(new Random().nextInt(2000));**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**System.out.println(Thread.currentThread().getName() + "线程执行完毕...");**

**}**

**}).start();**

**new Thread(new Runnable() {**

**@Override**

**public void run() {**

**System.out.println(Thread.currentThread().getName() + "线程开始执行..");**

**try {**

**Thread.sleep(new Random().nextInt(2000));**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**System.out.println(Thread.currentThread().getName() + "线程执行完毕...");**

**}**

**}).start();**

**while(Thread.activeCount() != 1){**

**//自旋**

**}**

**System.out.println("所有的线程执行完毕了");**

**}**

**}**

**运行结果：**

**Thread-0线程开始执行..**

**Thread-1线程开始执行..**

**Thread-2线程开始执行..**

**Thread-1线程执行完毕...**

**Thread-2线程执行完毕...**

**Thread-0线程执行完毕...**

**所有的线程执行完毕了**

## **死锁：**

**两个线程互相拿着对方的锁，导致谁也进行不下去。**

**public class SS {**

**private Object obj1 = new Object();**

**private Object obj2 = new Object();**

**public void a(){**

**synchronized (obj1) {**

**try {**

**Thread.sleep(1000);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**synchronized (obj2) {**

**System.out.println("a方法...");**

**}**

**}**

**}**

**public void b(){**

**synchronized (obj2) {**

**try {**

**Thread.sleep(1000);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**synchronized (obj1) {**

**System.out.println("b方法...");**

**}**

**}**

**}**

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

**SS s = new SS();**

**new Thread(new Runnable() {**

**@Override**

**public void run() {**

**s.a();**

**}**

**}).start();**

**new Thread(new Runnable() {**

**@Override**

**public void run() {**

**s.b();**

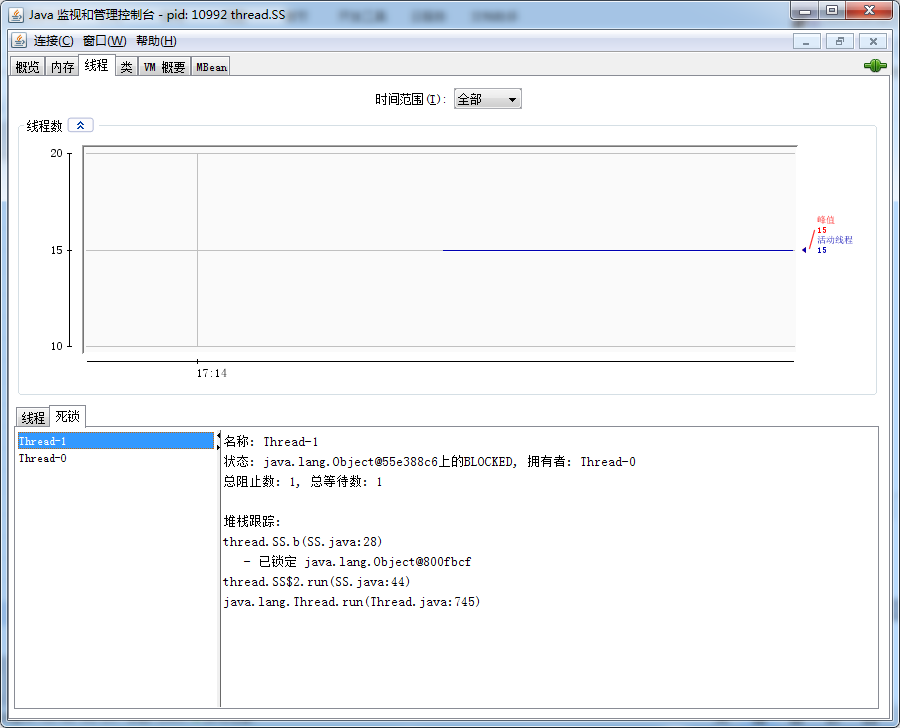
**}**

**}).start();**

**}**

**}**

**jconsole可以检测死锁**



# volatile原理和使用

volatile称之为轻量级锁，被vilatile修饰的变量，在线程之间是可见的。就是一个线程修改了这个变量，在另一个线程中能够读到这个修改之后的值。volatile能实现变量可见性，但是不能保证原子性操作。

Synchronized除了线程之间互斥之外，还有一个非常大的作用，就是保证变量线程间可见性。

synchronized能完全替代volatile，而volatile不能完全替代synchronized。因为volatile不能保证原子性。而volatile实现变量可见性更轻量化。

volatile实现变量一致性：

public class Demo {

private volatile int a = 1;

public int getA() {

return a;

}

public void setA(int a) {

this.a = a;

}

public static void main(String[] args) {

Demo d = new Demo();

new Thread(new Runnable() {

@Override

public void run() {

d.setA(12);

}

}).start();

new Thread(new Runnable() {

@Override

public void run() {

System.out.println(d.getA());

}

}).start();

}

}

synchronized实现变量线程间可见性：

public class Demo {

private int a = 1;

public synchronized int getA() {

return a;

}

public synchronized void setA(int a) {

this.a = a;

}

public static void main(String[] args) {

Demo d = new Demo();

new Thread(new Runnable() {

@Override

public void run() {

d.setA(12);

}

}).start();

new Thread(new Runnable() {

@Override

public void run() {

System.out.println(d.getA());

}

}).start();

}

}

## volatile底层实现：

Lock指令：

在多处理器的系统上

将当前处理器存行的内容写回到系统内存

这个写回到内存的操作会使在其他CPU里缓存了该内存地址的数据失效

硬盘 --内存 --CPU的缓存

# jdk提供的原子类原理及使用（jdk5出现的）

在java.util.concurrent.atomic包中。

可大致分为四大类：

1.原子更新基本类型

2.原子更新数组

3.原子更新抽象类型

4.原子更新字段

AtomicInteger类的实现++实现：

do {

prev = get();

next = prev + 1;

} while (!compareAndSet(prev, next));

return prev;

# Lock接口（1.5版本）

## Lock接口中的方法：

void lock(); 获取锁

void lockInterruptibly() 如果当前线程未被[中断](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/lang/Thread.html" \l "interrupt())，则获取锁。

boolean tryLock() 仅在调用时锁为空闲状态才获取该锁。

boolean tryLock(long time, [TimeUnit](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/TimeUnit.html" \o "java.util.concurrent 中的枚举)unit) 如果锁在给定的等待时间内空闲，并且当前 线程未被[中断](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/lang/Thread.html" \l "interrupt())，则获取锁。

void unlock() 释放锁。

[Condition](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/Condition.html" \o "java.util.concurrent.locks 中的接口) newCondition() 返回绑定到此 Lock 实例的新 [Condition](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/Condition.html" \o "java.util.concurrent.locks 中的接口) 实例。

## 锁的使用：

public class Sequeuece {

private int value;

Lock lock = new ReentrantLock();

public int getNext(){

lock.lock();

int a = value++;

lock.unlock();

return a;

}

public static void main(String[] args) {

Sequeuece s = new Sequeuece();

new Thread(new Runnable() {

@Override

public void run() {

while(true){

System.out.println(Thread.currentThread().getName() + " " + s.getNext());

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

new Thread(new Runnable() {

@Override

public void run() {

while(true){

System.out.println(Thread.currentThread().getName() + " " + s.getNext());

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

}

}

## 使用Lock锁的好处：

1.灵活性：

Lock需要显示地获取和释放锁，繁琐，但是能让代码更灵活，可以在任意的地方去释放锁和获取锁。

synchronized不需要显示地获取和释放锁，简单，但不灵活。

2.使用Lock可以方便的实现公平性。

3.非阻塞的获取锁

4.能被中断的获取锁

5.超时获取锁

# 使用Lock接口实现一个自己的锁（暂时仅实现了lock()和unlock()方法）

## 普通锁（不可重入）：

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

public class MyLock implements Lock {

private boolean isLocked = false;

@Override

public synchronized void lock() {

while(isLocked){ //自旋锁

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

isLocked = true;

}

@Override

public synchronized void unlock() {

isLocked = false;

notify();

}

@Override

public void lockInterruptibly() throws InterruptedException {

}

@Override

public boolean tryLock() {

return false;

}

@Override

public boolean tryLock(long time, TimeUnit unit)

throws InterruptedException {

return false;

}

@Override

public Condition newCondition() {

// TODO Auto-generated method stub

return null;

}

}

## 可重入锁：

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

public class MyLock implements Lock {

private boolean isLocked = false; //是否有线程获取锁了

private Thread lockBy = null; //记录当前线程

private int lockCount = 0; //记录使用了几重锁

@Override

public synchronized void lock() {

Thread currentThread = Thread.currentThread();

while(isLocked && currentThread != lockBy){

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

isLocked = true;

lockBy = currentThread;

lockCount++;

}

@Override

public synchronized void unlock() {

if(lockBy == Thread.currentThread()){

lockCount--;

if(lockCount == 0){

isLocked = false;

notify();

}

}

}

@Override

public void lockInterruptibly() throws InterruptedException {

}

@Override

public boolean tryLock() {

return false;

}

@Override

public boolean tryLock(long time, TimeUnit unit)

throws InterruptedException {

return false;

}

@Override

public Condition newCondition() {

// TODO Auto-generated method stub

return null;

}

}

# AbstractQueuedSynchronized(AQS)详解（jdk1.5）

<https://www.cnblogs.com/waterystone/p/4920797.html>

<https://blog.csdn.net/u012152619/article/details/74977570>

AQS是模板方法模式

为实现依赖于先进先出 (FIFO) 等待队列的阻塞锁和相关同步器（信号量、事件，等等）提供一个框架。

## AQS底层源码自己理解分析：

AQS会维护一个双向列表。当一个线程申请资源时，如果当前线程别占用，就会往维护的链表上添加一个Node结点。

## 使用此类：

为了将此类用作同步器的基础，需要适当地重新定义以下方法，这是通过使用 [getState()](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "getState())、[setState(int)](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "setState(int)) 和/或 [compareAndSetState(int, int)](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "compareAndSetState(int, int)) 方法来检查和/或修改同步状态来实现的：

[tryAcquire(int)](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "tryAcquire(int)) 试图在独占模式下获取对象状态。

[tryRelease(int)](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "tryRelease(int)) 试图设置状态来反映独占模式下的一个释放。

[tryAcquireShared(int)](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "tryAcquireShared(int)) 试图在共享模式下获取对象状态。

[tryReleaseShared(int)](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "tryReleaseShared(int)) 试图设置状态来反映共享模式下的一个释放。

[isHeldExclusively()](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "isHeldExclusively()) 如果对于当前（正调用的）线程，同步是以独占方式进行的， 则返回 true。

默认情况下，每个方法都抛出 [UnsupportedOperationException](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/concurrent/locks/../../../../java/lang/UnsupportedOperationException.html" \o "java.lang 中的类)。这些方法的实现在内部必须是线程安全的，通常应该很短并且不被阻塞。定义这些方法是使用此类的唯一 受支持的方式。其他所有方法都被声明为 final，因为它们无法是各不相同的。

## 实现AQS的数据结构：

底层是一个链表

static final class Node {

static final Node SHARED = new Node();

static final Node EXCLUSIVE = null;

static final int CANCELLED = 1; //线程已取消状态

static final int SIGNAL = -1;

static final int CONDITION = -2;

static final int PROPAGATE = -3;

volatile int waitStatus; //等待状态

volatile Node prev;

volatile Node next;

volatile Thread thread;

Node nextWaiter;

final boolean isShared() {

return nextWaiter == SHARED;

}

final Node predecessor() throws NullPointerException {

Node p = prev;

if (p == null)

throw new NullPointerException();

else

return p;

}

Node() { // Used to establish initial head or SHARED marker

}

Node(Thread thread, Node mode) { // Used by addWaiter

this.nextWaiter = mode;

this.thread = thread;

}

Node(Thread thread, int waitStatus) { // Used by Condition

this.waitStatus = waitStatus;

this.thread = thread;

}

}

## 使用AQS实现一个自己的Lock类：

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.AbstractQueuedSynchronizer;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

public class MyLock2 implements Lock {

private Helper helper = new Helper();

private class Helper extends AbstractQueuedSynchronizer{

@Override

protected boolean tryAcquire(int arg) {

/\*\*

\* 如果第一个线程进来，可以拿到锁，因此我们可以返回true

\* 如果第二个线程进来，则拿不到锁，返回false有种特例，如果当前进来的线程和当前线程时同一个线程，则可以拿到锁，但是有代价的，更新状态值

\* 如何判断是第一个线程进来还是其他线程进来？

\*/

int state = getState();

Thread t = Thread.currentThread();

if(state == 0){

if(compareAndSetState(0,arg)){

setExclusiveOwnerThread(Thread.currentThread());

return true;

}

} else if(getExclusiveOwnerThread() == t) {

setState(state + 1);

return true;

}

return false;

}

@Override

protected boolean tryRelease(int arg) {

//锁的获取和释放肯定是一一对应的,那么调用此方法的线程一定是当前线程

if(Thread.currentThread() != getExclusiveOwnerThread()){

throw new RuntimeException();

}

int state = getState() - arg;

boolean flag = false;

if(state == 0){

setExclusiveOwnerThread(null);

flag = true;

}

setState(state);

return flag;

}

Condition newCondition(){

return new ConditionObject();

}

}

@Override

public void lock() {

helper.acquire(1);

}

@Override

public void lockInterruptibly() throws InterruptedException {

helper.acquireInterruptibly(1);

}

@Override

public boolean tryLock() {

return helper.tryAcquire(1);

}

@Override

public boolean tryLock(long time, TimeUnit unit)

throws InterruptedException {

return helper.tryAcquireNanos(1, unit.toNanos(time));

}

@Override

public void unlock() {

helper.release(1);

}

@Override

public Condition newCondition() {

return helper.newCondition();

}

}

# 实现锁的方式--公平锁--非公平锁

AQS（abstarctQueueSynchronized）下的ReentrantLock实现类，实现了公平锁和非公平锁

## 公平锁和非公平锁只有两处不同：

1.非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。

2.非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

总结：公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

## 公平锁

公平是针对锁的获取而言的，如果一个锁是公平的，那么锁的获取顺序就应该符合请求的绝对时间顺序。

手动实现一个公平锁：

public class QueueObject {

private boolean isNotified = false;

public synchronized void doWait() throws InterruptedException {

while(!isNotified){

this.wait();

}

this.isNotified = false;

}

public synchronized void doNotify(){

this.isNotified = true;

this.notify();

}

public boolean equals(Object o){

return this == o;

}

}

import java.util.ArrayList;

import java.util.List;

public class FairLock {

private boolean isLocked = false;

private Thread lockingThread = null;

private List<QueueObject> waitingThreads = new ArrayList<QueueObject>();

public void lock() {

QueueObject queueObject = new QueueObject();

synchronized (this) {

waitingThreads.add(queueObject);

}

try {

queueObject.doWait();

} catch (InterruptedException e) {

synchronized (this) {

waitingThreads.remove(queueObject);

}

}

}

public void unlock(){

if(this.lockingThread != Thread.currentThread()){

throw new IllegalMonitorStateException("Calling thread has not locked this lock");

}

isLocked = false;

lockingThread = null;

if(waitingThreads.size() > 0){

waitingThreads.get(0).doNotify();

}

}

}

非公平锁

ReentrantLock中的NonfairSync内部类实现的就是非公平锁

# 读写锁--共享锁--排它锁

读写锁既是排它锁也是共享锁，读锁就是共享锁，写锁就是排它锁。

写——写 互斥

写——读 互斥

读——读 共享

读写锁在jdk中的实现类private ReadWriteLock rwl = new ReentrantReadWriteLock();

## 使用在ReentrantReadWriteLock的中读写锁实现类

public class Demo {

private Map<String, Object> map = new HashMap<>();

private ReadWriteLock rwl = new ReentrantReadWriteLock();

private Lock r = rwl.readLock();

private Lock w = rwl.writeLock();

public Object get(String key){

r.lock();

try {

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

return map.get(key);

} finally{

r.unlock();

}

}

public void put(String key, Object value){

w.lock();

try {

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

map.put(key, value);

} finally{

w.unlock();

}

}

}

## 读写锁需要保存的状态

int值就表示锁重入的次数

写锁重入的次数

读锁的个数

每个读锁重入的个数

## ReentrantReadWriteLock中实现排它锁和共享锁

ReentrantReadWriteLock中实现排它锁和共享锁需要重写AbstractQueuedSynchronized中的方法：

排它锁需要重写：tryAcquire(int)进行加锁，tryRelease(int)进行释放锁

tryAcquire(int)方法需要记录写的线程重入个数和排斥其他锁

tryRelease(int)方法需要减少写的重入个数和释放写的排它锁

共享锁需要重写：tryAcquireShared(int)进行加锁，tryReleaseShared(int)进行释放锁

tryAcquireShared(int)方法记录每一个读的线程的重入锁的个数

tryReleaseShared(int)方法减少写的重入个数和释放共享锁

## ReentrantReadWriteLock锁降级详解

锁降级就是指写锁降级为读锁。

在写锁没有释放的时候，获取到读锁，在释放写锁

private Map<String, Object> map = new HashMap<>();

private ReadWriteLock rwl = new ReentrantReadWriteLock();

private Lock r = rwl.readLock();

private Lock w = rwl.writeLock();

private volatile boolean isUpdate;

public void readWrite(){

r.lock();

if(isUpdate){

r.unlock();

w.lock();

map.put("xxx", "xxx");

r.lock();

w.unlock();

}

Object obj = new Object();

System.out.println();

r.unlock();

}

锁升级就是指读锁升级为写锁

在读锁没有释放的时候，获取到写锁，在释放读锁

# 线程安全问题总结

## 出现线程安全的条件

有以下条件会出现线程安全问题：

1.多线程环境下

2.多个线程共享一个资源

3.对资源进行非原子性（读写）操作

## 解决线程安全性问题的途径

1.synchroinzied锁（偏向锁，轻量级锁，重量级锁）

2.volatile锁，只能保证数据可见性，不能保证原子性操作

3.JDK提供的原子类

4.使用Lock锁（共享锁，排它锁）

## 认识的（\*锁）

偏向锁

轻量级锁

重量级锁

重入锁

自旋锁

死锁

共享锁

独占锁

排他锁

读写锁

公平锁

非公平锁

# wait notify

wait()方法：使当前线程释放锁

notify()方法：使处于等待状态的线程随机叫醒一个

notifyAll()方法：使处于等待状态的所有线程都处于唤醒状态

sleep不会释放锁

## 生产者消费者模式

资源类

public class P {

private int count;

public final int MAX\_COUNT = 10;

public synchronized void push (){

while(count >= MAX\_COUNT){

try {

System.out.println(Thread.currentThread().getName() + " 库存数量达到上限，停止生产");

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

count ++;

System.out.println(Thread.currentThread().getName() + " 生产者生产，当前库存为：" + count);

}

public synchronized void take(){

while(count <= 0){

try {

System.out.println(Thread.currentThread().getName() + " 库存数量为0，消费者等待");

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

count --;

System.out.println(Thread.currentThread().getName() + " 消费者消费，当前库存为：" + count);

notifyAll();

}

}

生产者：

public class PushTarget implements Runnable {

private P p;

public PushTarget(P p){

this.p = p;

}

@Override

public void run() {

while(true){

p.push();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

消费者：

public class TakeTarget implements Runnable {

private P p;

public TakeTarget(P p) {

this.p = p;

}

@Override

public void run() {

while(true){

p.take();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

测试类：

public class Main {

public static void main(String[] args) {

P p = new P();

PushTarget push = new PushTarget(p);

TakeTarget take = new TakeTarget(p);

new Thread(push).start();

new Thread(push).start();

new Thread(push).start();

new Thread(push).start();

new Thread(push).start();

new Thread(take).start();

new Thread(take).start();

new Thread(take).start();

new Thread(take).start();

new Thread(take).start();

}

}

# 实现三个线程之间进行依次查数0，1，2，3....

## 方法一：使用信号量，wait(),notifyAll()等方法

public class ConditionDemo {

volatile int a = 0;

private int signal;

public synchronized void a(){

while(signal != 0){

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(a++);

signal++;

notifyAll();

}

public synchronized void b(){

while(signal != 1){

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(a++);

signal++;

notifyAll();

}

public synchronized void c(){

while(signal != 2){

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(a++);

signal = 0;

notifyAll();

}

public static void main(String[] args) {

ConditionDemo d = new ConditionDemo();

A a = new A(d);

B b = new B(d);

C c = new C(d);

new Thread(a).start();

new Thread(b).start();

new Thread(c).start();

}

}

class A implements Runnable{

private ConditionDemo demo;

public A(ConditionDemo demo) {

this.demo = demo;

}

@Override

public void run() {

while(true){

demo.a();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class B implements Runnable{

private ConditionDemo demo;

public B(ConditionDemo demo) {

this.demo = demo;

}

@Override

public void run() {

while(true){

demo.b();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class C implements Runnable{

private ConditionDemo demo;

public C(ConditionDemo demo) {

this.demo = demo;

}

@Override

public void run() {

while(true){

demo.c();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

## 方法二：使用Condition

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class ConditionDemo1 {

volatile int a1 = 0;

private int signal;

Lock lock = new ReentrantLock();

Condition a = lock.newCondition();

Condition b = lock.newCondition();

Condition c = lock.newCondition();

public void a(){

lock.lock();

while(signal != 0){

try {

a.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(a1++);

signal++;

b.signal();

lock.unlock();

}

public void b(){

lock.lock();

while(signal != 1){

try {

b.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(a1++);

signal++;

c.signal();

lock.unlock();

}

public void c(){

lock.lock();

while(signal != 2){

try {

c.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(a1++);

signal = 0;

a.signal();

lock.unlock();

}

public static void main(String[] args) {

ConditionDemo1 d = new ConditionDemo1();

A a = new A(d);

B b = new B(d);

C c = new C(d);

new Thread(a).start();

new Thread(b).start();

new Thread(c).start();

}

}

class A implements Runnable{

private ConditionDemo1 demo;

public A(ConditionDemo1 demo) {

this.demo = demo;

}

@Override

public void run() {

while(true){

demo.a();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class B implements Runnable{

private ConditionDemo1 demo;

public B(ConditionDemo1 demo) {

this.demo = demo;

}

@Override

public void run() {

while(true){

demo.b();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class C implements Runnable{

private ConditionDemo1 demo;

public C(ConditionDemo1 demo) {

this.demo = demo;

}

@Override

public void run() {

while(true){

demo.c();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

# condition实现一个有界队列

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class MyQueue<E> {

private Object[] obj;

private int addIndex;

private int removeIndex;

private int queueSize;

private Lock lock = new ReentrantLock();

Condition addCondition = lock.newCondition();

Condition removeCondition = lock.newCondition();

public MyQueue(int count){

obj = new Object[count];

}

public void add(E e){

lock.lock();

while(queueSize == obj.length){

try {

addCondition.await();

} catch (InterruptedException e1) {

e1.printStackTrace();

}

}

obj[addIndex] = e;

if(++addIndex == obj.length){

addIndex = 0;

}

queueSize++;

removeCondition.signal();

lock.unlock();

}

public void remove(){

lock.lock();

while(queueSize == 0){

try {

removeCondition.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

obj[removeIndex] = null;

if(++removeIndex == obj.length){

removeIndex = 0;

}

queueSize--;

addCondition.signal();

lock.unlock();

}

}

# condition实现原理--等待队列

ReentrentLock是一个同步队列

condition维护了一个等待队列，是一个单向链表。

condition与wait()、notify()的区别：

wait(),notify()只能创建一个消息队列

condition可以为每一个线程创建一个消息队列

# ThreadLock原理与使用

1.2就出现了，1.5时进行了改写。

ThreadLocal用于保存某个线程共享变量：对于同一个static ThreadLocal，不同线程只能从中get，set，remove自己的变量，而不会影响其他线程的变量。

1、ThreadLocal.get: 获取ThreadLocal中当前线程共享变量的值。

2、ThreadLocal.set: 设置ThreadLocal中当前线程共享变量的值。

3、ThreadLocal.remove: 移除ThreadLocal中当前线程共享变量的值。

4、ThreadLocal.initialValue: ThreadLocal没有被当前线程赋值时或当前线程刚调用remove方法后调用get方法，返回此方法值。

## ThreadLock底层原理

ThreadLock底层是一个map的数据结构，以当前线程为key，以值为value

## 使用：

public class Demo {

private ThreadLocal<Integer> count = new ThreadLocal<Integer>(){

protected Integer initialValue() {

return new Integer(0);

};

};

public int getNext(){

Integer value = count.get();

value++;

count.set(value);

return value;

}

public static void main(String[] args) {

Demo d = new Demo();

new Thread(new Runnable() {

@Override

public void run() {

while(true){

System.out.println(Thread.currentThread().getName() + " " + d.getNext());

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

new Thread(new Runnable() {

@Override

public void run() {

while(true){

System.out.println(Thread.currentThread().getName() + " " + d.getNext());

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

new Thread(new Runnable() {

@Override

public void run() {

while(true){

System.out.println(Thread.currentThread().getName() + " " + d.getNext());

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

new Thread(new Runnable() {

@Override

public void run() {

while(true){

System.out.println(Thread.currentThread().getName() + " " + d.getNext());

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}).start();

}

}

# 并发工具类CountDownLatch

https://blog.csdn.net/zzg1229059735/article/details/61191679

1.5出现的：

CountDownLatch是一个同步工具类，用来协调多个线程之间的同步，或者说起到线程之间的通信（而不是用作互斥的作用）。

CountDownLatch能够使一个线程在等待另外一些线程完成各自工作之后，再继续执行。使用一个计数器进行实现。计数器初始值为线程的数量。当每一个线程完成自己任务后，计数器的值就会减一。当计数器的值为0时，表示所有的线程都已经完成了任务，然后在CountDownLatch上等待的线程就可以恢复执行任务。

CountDownLatch方法使用CountDownLatch.countDown()方法，使得减到0的时候会唤醒线程

## 一个案例：对一个文本中所有的数字并行求和

### 正常解决方法：

import java.io.BufferedReader;

import java.io.FileReader;

import java.util.ArrayList;

import java.util.List;

public class Demo {

private int[] nums;

public Demo(int line){

nums = new int[line];

}

public void calc(String line, int index){

String[] nus = line.split(","); //切分出分一个数字

int total = 0;

for(String num : nus){

total += Integer.parseInt(num);

}

nums[index] = total;

System.out.println(Thread.currentThread().getName() + " 执行计算任务..." + line + " 结果为：" + total);

}

public void sum(){

int total = 0;

for(int i = 0; i < nums.length; i++){

total += nums[i];

}

System.out.println("最终的结果为：" + total);

}

public static void main(String[] args) {

List<String> contents = readFile();

int lineCount = contents.size();

Demo d = new Demo(lineCount);

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

final int j = i;

new Thread(new Runnable() {

@Override

public void run() {

d.calc(contents.get(j), j);

}

}).start();

}

while(Thread.activeCount() > 1){

}

d.sum();

}

private static List<String> readFile() {

List<String> contents = new ArrayList<String>();

String line = null;

BufferedReader br = null;

try {

br = new BufferedReader(new FileReader("D:\\nums.txt"));

while((line = br.readLine()) != null){

contents.add(line);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

if(br != null){

try {

br.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return contents;

}

}

### CountDownLatch解决方法：

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.CountDownLatch;

public class Demo2 {

private int[] nums;

public Demo2(int line){

nums = new int[line];

}

public void calc(String line, int index, CountDownLatch latch){

String[] nus = line.split(","); //切分出分一个数字

int total = 0;

for(String num : nus){

total += Integer.parseInt(num);

}

nums[index] = total;

System.out.println(Thread.currentThread().getName() + " 执行计算任务..." + line + " 结果为：" + total);

latch.countDown();

}

public void sum(){

int total = 0;

for(int i = 0; i < nums.length; i++){

total += nums[i];

}

System.out.println("最终的结果为：" + total);

}

public static void main(String[] args) {

List<String> contents = readFile();

int lineCount = contents.size();

CountDownLatch latch = new CountDownLatch(lineCount);

Demo2 d = new Demo2(lineCount);

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

final int j = i;

new Thread(new Runnable() {

@Override

public void run() {

d.calc(contents.get(j), j, latch);

}

}).start();

}

try {

latch.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

d.sum();

}

private static List<String> readFile() {

List<String> contents = new ArrayList<String>();

String line = null;

BufferedReader br = null;

try {

br = new BufferedReader(new FileReader("D:\\nums.txt"));

while((line = br.readLine()) != null){

contents.add(line);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

if(br != null){

try {

br.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return contents;

}

}

# 并发工具类CyclicBarrier

CyclicBarrier:可看成是个障碍，所有的线程必须到齐后才能一起通过这个障碍

使用CyclicBarrier.await方法的线程达到一定数量就可以全部唤醒执行以下内容

## 使用：

import java.util.Random;

import java.util.concurrent.CyclicBarrier;

public class Demo3 {

Random random = new Random();

public void meeting(CyclicBarrier barrier){

try {

Thread.sleep(random.nextInt(4000));

} catch (InterruptedException e1) {

e1.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " 到达会议室，等待开会");

try {

barrier.await();

} catch (Exception e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName());

}

public static void main(String[] args) {

Demo3 demo = new Demo3();

CyclicBarrier barrier = new CyclicBarrier(10, new Runnable() {

@Override

public void run() {

System.out.println("开始开会。。。");

}

});

for(int i = 0; i < 10; i++){

new Thread(new Runnable() {

@Override

public void run() {

demo.meeting(barrier);

}

}).start();

}

}

}

# 并发工具类Semaphore

可以定义一个方法同一时刻有多少线程访问

Semaphore的实现和线程池不同点：

Semaphore是有很多的线程等待这个固定的位置，而线程池只有这些线程等待执行这些位置

# 并发工具类Exchanger

# 提前完成任务Future的使用

Future如何实现的：

Callable

FutureTask

Thread

Callable和Runnable在代码使用上的区别

Runnable的run方法是被线程调用的，在run方法是异步执行的

Callable的call方法，不是异步执行的，是由Future的run方法执行的

# Future设计模式

## 需要实例化的类：

public class Product {

private int id;

private String name;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Product(int id, String name) {

System.out.println("开始生产 " + name);

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

e.printStackTrace();

}

this.id = id;

this.name = name;

System.out.println(name + "生产完毕");

}

@Override

public String toString() {

return "Product [id=" + id + ", name=" + name + "]";

}

}

## 实例化的工厂类：

public class Future {

private Product product;

private boolean down;

public synchronized void setProduct(Product product){

if(down){

return;

}

this.product = product;

this.down = true;

notifyAll();

}

public synchronized Product getProduct(){

while(!down){

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

return this.product;

}

}

## 创建另一个线程进行实例化类：

import java.util.Random;

public class ProductFactory {

public Future createProduct(String name){

Future f = new Future(); //创建一个订单

System.out.println("下单成功，你可以去上班了");

//生产产品

new Thread(new Runnable() {

@Override

public void run() {

Product p = new Product(new Random().nextInt(), name);

f.setProduct(p);

}

}).start();

return f;

}

}

## 模拟使用Future设计模式：

public class Demo {

public static void main(String[] args) {

ProductFactory pf = new ProductFactory();

//下单交钱

Future f = pf.createProduct("蛋糕");

System.out.println("我去上班了，下了班我来取蛋糕...");

System.out.println("我拿着蛋糕回家" + f.getProduct());

}

}

# Fork/Join框架

* 多线程的目的不仅仅是提高程序运行的性能，
* 但是可以充分利用CPU资源

Fork/Join的出现是为了多核时代，提高CPU利用率，利用Fork将任务分开，利用Join将人物合并

ForkJoinTask类就实现了这个问题

使用这个操作：

import java.util.concurrent.ForkJoinPool;

import java.util.concurrent.Future;

import java.util.concurrent.RecursiveTask;

public class Demo extends RecursiveTask<Integer> {

private int begin;

private int end;

public Demo(int begin, int end) {

this.begin = begin;

this.end = end;

}

@Override

protected Integer compute() {

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

int sum = 0;

//拆分任务

if(end - begin <= 2){

//计算

for(int i = begin; i <= end; i++){

sum += i;

}

} else {

Demo d1 = new Demo(begin, (begin + end) / 2);

Demo d2 = new Demo((begin + end) / 2, end);

//执行自任务

d1.fork();

d2.fork();

Integer a = d1.join();

Integer b = d2.join();

sum = a + b;

}

//拆分

return sum;

}

public static void main(String[] args) throws Exception {

ForkJoinPool pool = new ForkJoinPool();

Future<Integer> future = pool.submit(new Demo(1, 100));

System.out.println("....");

System.out.println("计算的值为：" + future.get());

}

}

# 并发容器和同步容器

同步容器：以前遇到的都是同步容器，有的容器是线程安全的，不过效率不好，有的是线程不安全的

并发容器：在并发容器中使用的容器

## 同步容器：

Vector-->ArrayList

Hashtable-->HashMap

## 并发容器：（jdk1.5实现）

Vector-->ArrayList-->CopyOnWriteArrayList

Hashtable-->HashMap-->ConcurrentHashMap

## CopyOnWriteArrayList并发容器:

实现原理：在ArrayList基础上进行了修改，他对会出现线程安全的方法上做了改变

在修改的时候会复制出一份，然后再进行修改，再将这个新数组设置在原数组上。

使用场景：读多写少的操作，读少写多的操作用同步操作也可以

## ConcurrentLinedQueue非阻塞队列

添加时，队列尾指针tail如果next为null，那么将新添加的元素添加到tail的next元素上，如果tail的next不为null，那么tail将指向新添加的元素

## BlockingQueue阻塞队列（多看所有的实现）

1.5出现的

阻塞队列并不是吧所有的方法都进行了阻塞，而是以下的两个方法：

put()

take()

# 实现简单的消息队列

## 消息队列的两种形式：

点对点

发布-订阅

## 消息队列有哪些：

RabbitMQ（Erlang编写）

Redis

ZeroMQ

ActiveMQ

Jafka/Kafka

# 线程池

## 什么是线程池：

线程池是指在初始化一个多线程应用程序过程中创建一个线程集合，然后在需要执行新的任务时重用这些线程而不是新建一个线程。线程池中线程的数量通常完全取决于可用内存数量和应用程序的需求。然而，增加可用线程数量是可能的。线程池中的每个线程都有被分配一个任务，一旦任务已经完成了，线程回到池子中并等待下一次分配任务。

## 为什么使用线程池：

基于以下几个原因在多线程应用程序中使用线程是必须的：

  1. 线程池改进了一个应用程序的响应时间。由于线程池中的线程已经准备好且等待被分配任务，应用程序可以直接拿来使用而不用新建一个线程。

  2. 线程池节省了CLR 为每个短生存周期任务创建一个完整的线程的开销并可以在任务完成后回收资源。

  3. 线程池根据当前在系统中运行的进程来优化线程时间片。

  4. 线程池允许我们开启多个任务而不用为每个线程设置属性。

  5. 线程池允许我们为正在执行的任务的程序参数传递一个包含状态信息的对象引用。

  6. 线程池可以用来解决处理一个特定请求最大线程数量限制问题。

## 线程池的优势：

1.降低资源消耗。通过重复利用以创建的线程降低线程和销毁造成的消耗。

2.提高响应速度。当任务到达时，如果可以不需要等到线程创建就能立即执行。

3.提高线程的可管理性。线程时稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

## 线程池的饱和策略：

CallerRunsPolicy：只用调用者所在线程来运行任务

DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务

DiscardPolicy：不处理，丢弃掉

## 线程池源码解析：

### 参数认识：

1.corePoolSize：线程池的基本大小，当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本路线能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不在创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。

2.runnableTaskQueue：任务队列，用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。

* ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按FIFO（先进先出）原则对元素进行排序。
* LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO（先进先出）排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。
* SychronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常高于LinkedBlockingQueue。静态工厂方法Executors.newCachedThreadPool使用了这个队列。
* PriorityBlockingQueue：一个具有优先级的无限阻塞队列

3.maximumPoolSize：线程池最大大小，线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会在创建新的线程执行任务，值得注意的是如果使用了无界的任务队列这个参数就没什么效果。

4.ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字，Debug和定位问题时非常有帮助。

5.RejectedExecutionHandle（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交新的任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。

* CallerRunsPolicy：只用调用者所在线程来运行任务
* DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务
* DiscardPolicy：不处理，丢弃掉
* 当然也可以根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化不能处理的任务。

6.keepAliveTime：线程活动保持时间，线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。

7.TimeUnit：线程活动保持时间的单位，可选单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒（MILLISECONDS），微妙（MICROSECONDS，千分之一秒）和毫微秒（MANOSECONDS，千分之一微妙）

### 线程池的其他变量

//线程池的控制状态：用来表示线程池的运行状态（整型的高3位）和运行的worker数量（低29位）

private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

//29位的偏移量

private static final int COUNT\_BITS = Integer.SIZE - 3;

//最大容量（2^29 - 1）

private static final int CAPACITY = (1 << COUNT\_BITS) - 1;

### 线程池的运行状态：

//接受新任务并且处理已经进入阻塞队列的任务

private static final int RUNNING = -1 << COUNT\_BITS;

//不接受任务，但是处理已经进入阻塞队列的任务

private static final int SHUTDOWN = 0 << COUNT\_BITS;

//不接受新任务，不处理已经进入阻塞队列的任务并且中断正在运行的任务

private static final int STOP = 1 << COUNT\_BITS;

//所有任务都已经终止，workerCount为0，线程转化为TIDYING状态并且调用terminated钩子函数

private static final int TIDYING = 2 << COUNT\_BITS;

//terminated钩子函数已经运行完成

private static final int TERMINATED = 3 << COUNT\_BITS;

# Executor框架

Executors.newCachedThreadPool()

Executors.newFixedThreadPool(10)

Executors.newSingleThreadExecutor() //只执行一个任务，一个线程挂掉，线程池在启动一个线程进行执行

Executors.newScheduledThreadPool(10) //延时一段时间在执行

Executors.newWorkStealingPool(); //ForkJoin的线程池

# 简易web服务器

http协议：<http://www.cnblogs.com/ranyonsue/p/5984001.html#4006792>