# 一、线程、同步、异步、线程池

多线程并发时，多个线程同时请求同一个资源，必然导致此资源的数据不安全，A线程修改了B线程的处理的数据，而B线程又修改了A线程处理的数理。显然这是由于全局资源造成的，有时为了解决此问题，优先考虑使用局部变量，退而求其次使用同步代码块，出于这样的安全考虑就必须牺牲系统处理性能，加在多线程并发时资源挣夺最激烈的地方，这就需要实现线程的同步机制。

同步：A线程要请求某个资源，但是此资源正在被B线程使用中，因为同步机制存在，A线程请求不到，怎么办，A线程只能等待下去。

异步：A线程要请求某个资源，但是此资源正在被B线程使用中，因为没有同步机制存在，A线程仍然请求的到，A线程无需等待。

显然，同步最最安全，最保险的。而异步不安全，容易导致死锁，这样一个线程死掉就会导致整个进程崩溃，但没有同步机制的存在，性能会有所提升。

## 1.1、Java中实现多线程

（1）、继承Thread,重写里面的run方法

（2）、实现runnable接口

比较推荐后者，第一，java没有单继承的限制第二，还可以隔离代码

## 1.2、Java同步机制的4种实现方式

（1）、ThreadLocal

（2）、synchronized( )

（3）、wait( )与notify( )

（4）、volatile

目的：都是为了解决多线程中的对同一变量的访问冲突

### 1.2.1、ThreadLocal

ThreadLocal 保证不同线程拥有不同实例，相同线程一定拥有相同的实例，即为每一个使用该变量的线程提供一个该变量值的副本，每一个线程都可以独立改变自己的副本，而不是与其它线程的副本冲突。

优势：提供了线程安全的共享对象

与其它同步机制的区别：同步机制是为了同步多个线程对相同资源的并发访问，是为了多个线程之间进行通信；而ThreadLocal是隔离多个线程的数据共享，从根本上就不在多个线程之间共享资源，这样当然不需要多个线程进行同步了。

## 1.3、线程池

要知道在计算机中任何资源的创建，包括线程，都需要消耗系统资源的。在WEB服务中，对于web服务器的响应速度必须要尽可能的快，这就容不得每次在用户提交请求按钮后，再创建线程提供服务。为了减少用户的等待时间，线程必须预先创建，放在线程池中，线程池可以用HashTable这种数据结构来实现，看了Apach HTTP服务器的线程池的源代码，用是就是HashTable,KEY用线程对象，value 用ControlRunnable，ControlRunnable是线程池中唯一能干活的线程，是它指派线程池中的线程对外提供服务。出于安全考虑，Apach HTTP服务器的线程池它是同步的。听说weblogic有异步的实现方式，没有研究过，不敢确定

一、关键字：

thread（线程）、thread-safe(线程安全)、intercurrent（并发的）

synchronized(同步的)、asynchronized(异步的)、

volatile（易变的）、atomic（原子的）、share（共享）

二、总结背景：

一次读写共享文件编写，嚯，好家伙，竟然揪出这些零碎而又是一路的知识点。于是乎，Google和翻阅了《Java参考大全》、《Effective Java Second Edition》，

特此总结一下供日后工作学习参考。

三、概念：

1、 什么时候必须同步？什么叫同步？如何同步？

要跨线程维护正确的可见性，只要在几个线程之间共享非 final 变量，就必须使用synchronized（或 volatile）以确保一个线程可以看见另一个线程做的更改。为

了在线程之间进行可靠的通信，也为了互斥访问，同步是必须的。这归因于java语言规范的内存模型，它规定了：一个线程所做的变化何时以及如何变成对其它线程

可见。

因为多线程将异步行为引进程序，所以在需要同步时，必须有一种方法强制进行。例如：如果2个线程想要通信并且要共享一个复杂的数据结构，如链表，此时需要确

保它们互不冲突，也就是必须阻止B线程在A线程读数据的过程中向链表里面写数据（A获得了锁，B必须等A释放了该锁）。

为了达到这个目的，java在一个旧的的进程同步模型——监控器（Monitor）的基础上实现了一个巧妙的方案：监控器是一个控制机制，可以认为是一个很小的、只能

容纳一个线程的盒子，一旦一个线程进入监控器，其它的线程必须等待，直到那个线程退出监控为止。通过这种方式，一个监控器可以保证共享资源在同一时刻只可

被一个线程使用。这种方式称之为同步。（一旦一个线程进入一个实例的任何同步方法，别的线程将不能进入该同一实例的其它同步方法，但是该实例的非同步方法

仍然能够被调用）。

错误的理解：同步嘛，就是几个线程可以同时进行访问。

同步和多线程关系：没多线程环境就不需要同步;有多线程环境也不一定需要同步。

锁提供了两种主要特性：互斥（mutual exclusion）和可见性（visibility）。

互斥即一次只允许一个线程持有某个特定的锁，因此可使用该特性实现对共享数据的协调访问协议，这样，一次就只有一个线程能够使用该共享数据。

可见性要更加复杂一些，它必须确保释放锁之前对共享数据做出的更改对于随后获得该锁的另一个线程是可见的 —— 如果没有同步机制提供的这种可见性保证，线

程看到的共享变量可能是修改前的值或不一致的值，这将引发许多严重问题

小结：为了防止多个线程并发对同一数据的修改，所以需要同步，否则会造成数据不一致（就是所谓的：线程安全。如java集合框架中Hashtable和Vector是线程安全

的。我们的大部分程序都不是线程安全的，因为没有进行同步，而且我们没有必要，因为大部分情况根本没有多线程环境）。

2、 什么叫原子的（原子操作）？

Java原子操作是指：不会被打断地的操作。（就是做到互斥和可见性？！）

那难道原子操作就可以真的达到线程安全同步效果了吗？实际上有一些原子操作不一定是线程安全的。

那么，原子操作在什么情况下不是线程安全的呢？也许是这个原因导致的：java线程允许线程在自己的内存区保存变量的副本。允许线程使用本地的私有拷贝进行工

作而非每次都使用主存的值是为了提高性能（本人愚见：虽然原子操作是线程安全的，可各线程在得到变量（读操作）后，就是各自玩弄自己的副本了，更新操作（

写操作）因未写入主存中，导致其它线程不可见）。

那该如何解决呢？因此需要通过java同步机制。

在java中，32位或者更少位数的赋值是原子的。在一个32位的硬件平台上，除了double和long型的其它原始类型通常都是使用32位进行表示，而double和long通常使

用64位表示。另外，对象引用使用本机指针实现，通常也是32位的。对这些32位的类型的操作是原子的。

这些原始类型通常使用32位或者64位表示，这又引入了另一个小小的神话：原始类型的大小是

由语言保证的。这是不对的。java语言保证的是原始类型的表数范围而非JVM中的存储大小。因此int型总是有相同的表数范围。在一个JVM上可能使用32位实现，而在

另一个JVM上可能是64位的。在此再次强调：在所有平台上被保证的是表数范围，32位以及更小的值的操作是原子的。

3、 不要搞混了：同步、异步

举个例子：普通B/S模式（同步）AJAX技术（异步）

同步：提交请求->等待服务器处理->处理完返回这个期间客户端浏览器不能干任何事

异步：请求通过事件触发->服务器处理（这是浏览器仍然可以作其他事情）->处理完毕

可见，彼“同步”非此“同步”——我们说的java中的那个共享数据同步（synchronized）

一个同步的对象是指行为（动作），一个是同步的对象是指物质（共享数据）。

4、 Java同步机制有4种实现方式：（部分引用网上资源）

① ThreadLocal ② synchronized( ) ③ wait() 与 notify() ④ volatile

目的：都是为了解决多线程中的对同一变量的访问冲突

ThreadLocal

ThreadLocal 保证不同线程拥有不同实例，相同线程一定拥有相同的实例，即为每一个使用该

变量的线程提供一个该变量值的副本，每一个线程都可以独立改变自己的副本，而不是与其它线程

的副本冲突。

优势：提供了线程安全的共享对象

与其它同步机制的区别：同步机制是为了同步多个线程对相同资源的并发访问，是为了多个线程之

间进行通信；而 ThreadLocal 是隔离多个线程的数据共享，从根本上就不在多个线程之间共享资源

，这样当然不需要多个线程进行同步了。

volatile

volatile 修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。

而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。

优势：这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

缘由：Java 语言规范中指出，为了获得最佳速度，允许线程保存共享成员变量的私有拷贝，而

且只当线程进入或者离开同步代码块时才与共享成员变量的原始值对比。这样当多个线程同时与某

个对象交互时，就必须要注意到要让线程及时的得到共享成员变量的变化。而 volatile 关键字就

是提示 VM ：对于这个成员变量不能保存它的私有拷贝，而应直接与共享成员变量交互。

使用技巧：在两个或者更多的线程访问的成员变量上使用 volatile 。当要访问的变量已在

synchronized 代码块中，或者为常量时，不必使用。

线程为了提高效率，将某成员变量(如A)拷贝了一份（如B），线程中对A的访问其实访问的

是B。只在某些动作时才进行A和B的同步，因此存在A和B不一致的情况。volatile就是用来避免这种

情况的。 volatile告诉jvm，它所修饰的变量不保留拷贝，直接访问主内存中的（读操作多时使用

较好；线程间需要通信，本条做不到）

Volatile 变量具有 synchronized 的可见性特性，但是不具备原子特性。这就是说线程能够自

动发现 volatile 变量的最新值。Volatile 变量可用于提供线程安全，但是只能应用于非常有限的

一组用例：多个变量之间或者某个变量的当前值与修改后值之间没有约束。

您只能在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理

想的线程安全，必须同时满足下面两个条件：

对变量的写操作不依赖于当前值；该变量没有包含在具有其他变量的不变式中。

sleep() vs wait()

sleep是线程类（Thread）的方法，导致此线程暂停执行指定时间，把执行机会给其他线程，但是监

控状态依然保持，到时后会自动恢复。调用sleep不会释放对象锁。

wait是Object类的方法，对此对象调用wait方法导致本线程放弃对象锁，进入等待此对象的等待锁

定池，只有针对此对象发出notify方法（或notifyAll）后本线程才进入对象锁定池准备获得对象锁

进入运行状态。

（如果变量被声明为volatile，在每次访问时都会和主存一致；如果变量在同步方法或者同步块中

被访问，当在方法或者块的入口处获得锁以及方法或者块退出时释放锁时变量被同步。）

四、例子：

Demo1:

package test.thread;

class SynTest{

//非同步

static void method(Thread thread){

System.out.println("begin "+thread.getName());

try{

Thread.sleep(2000);

}catch(Exception ex){

ex.printStackTrace();

}

System.out.println("end "+thread.getName());

}

//同步方式一：同步方法

synchronized static void method1(Thread thread){//这个方法是同步的方法，每次只有一

个线程可以进来

System.out.println("begin "+thread.getName());

try{

Thread.sleep(2000);

}catch(Exception ex){

ex.printStackTrace();

}

System.out.println("end "+thread.getName());

}

//同步方式二：同步代码块

static void method2(Thread thread){

synchronized(SynTest.class) {

System.out.println("begin "+thread.getName());

try{

Thread.sleep(2000);

}catch(Exception ex){

ex.printStackTrace();

}

System.out.println("end "+thread.getName());

}

}

//同步方式三：使用同步对象锁

private static Object \_lock1=new Object();

private static byte \_lock2[]={};//据说，此锁更可提高性能。源于：锁的对象越小越好

static void method3(Thread thread){

synchronized(\_lock1) {

System.out.println("begin "+thread.getName());

try{

Thread.sleep(2000);

}catch(Exception ex){

ex.printStackTrace();

}

System.out.println("end "+thread.getName());

}

}

public static void main(String[] args){

//启动3个线程，这里用了匿名类

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

new Thread(){

public void run(){

method(this);

//method1(this);

//method2(this);

//method3(this);

}

}.start();

}

}

}

Demo2:

package test.thread;

import com.util.LogUtil;

public class SynTest2 {

public static void main(String[] args){

Callme target=new Callme();

Caller ob1=new Caller(target,"Hello");

Caller ob2=new Caller(target,"Synchronized");

Caller ob3=new Caller(target,"World");

}

}

class Callme{

synchronized void test(){

LogUtil.log("测试是否是：一旦一个线程进入一个实例的任何同步方法，别的线程将不能

进入该同一实例的其它同步方法，但是该实例的非同步方法仍然能够被调用");

}

void nonsynCall(String msg){

LogUtil.log("["+msg);

LogUtil.log("]");

}

synchronized void synCall(String msg){

LogUtil.logPrint("["+msg);

LogUtil.log("]");

}

}

class Caller implements Runnable{

String msg;

Callme target;

Thread t;

Caller(Callme target,String msg){

this.target=target;

this.msg=msg;

t=new Thread(this);

t.start();

}

public void run() {

// TODO Auto-generated method stub

//target.nonsynCall(msg);

target.synCall(msg);

target.test();

}

}