# 多线程基础

## 相关概念

进程是一个应用程序在处理机上的一次执行过程，线程是进程的一部分。

多线程的优势：

进程之间不能共享内存，线程之间共享内存非常容易。

线程比进程代价小。

并发和并行

并行：多个CPU同时执行一段处理逻辑。是真正的同时。

并发：cpu调度。

线程安全：指并发时，代码经多线程使用，结果不受影响。

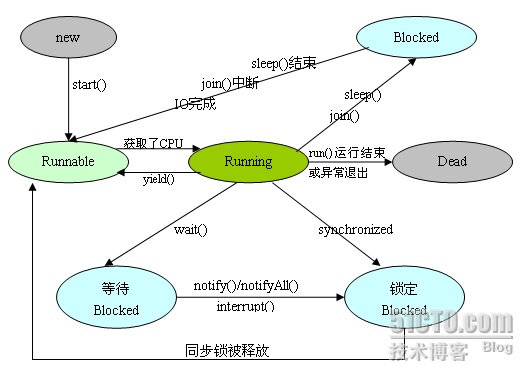
同步：人为控制和调度，保证共享资源时运行结果准确。

线程安全的优先级高于性能。

## 线程的状态

java.lang.Thread.State线程状态

1. NEW:  
   至今尚未启动的线程的状态。
2. RUNNABLE:  
   可运行线程的线程状态。处于可运行状态的某一线程正在 Java 虚拟机中运行，但它可能正在等待操作系统中的其他资源，比如处理器。
3. BLOCKED ：  
   受阻塞并且正在等待监视器锁的某一线程的线程状态。处于受阻塞状态的某一线程正在等待监视器锁，以便进入一个同步的块/方法，或者在调用 Object.wait 之后再次进入同步的块/方法。
4. WAITING ：  
   某一等待线程的线程状态。某一线程因为调用下列方法之一而处于等待状态：  
   -》不带超时值的 Object.wait  
   -》不带超时值的 Thread.join  
   -》LockSupport.park  
   处于等待状态的线程正等待另一个线程，以执行特定操作。 例如，已经在某一对象上调用了 Object.wait() 的线程正等待另一个线程，以便在该对象上调用 Object.notify() 或 Object.notifyAll()。已经调用了 Thread.join() 的线程正在等待指定线程终止。
5. TIMED\_WAITING  
   具有指定等待时间的某一等待线程的线程状态。某一线程因为调用以下带有指定正等待时间的方法之一而处于定时等待状态：  
   -》Thread.sleep  
   -》带有超时值的 Object.wait  
   -》带有超时值的 Thread.join  
   -》LockSupport.parkNanos  
   -》LockSupport.parkUntil
6. TERMINATED  
   已终止线程的线程状态。线程已经结束执行。



# 线程同步

## synchronized

synchronized, wait, notify 是任何对象都具有的同步工具, wait/notify必须存在于synchronized块中

## Volatile 关键字

是修饰 共享**变量**的，保证数据可见，但不保证不被修改，所以要结合CAS使用。

# 多线程的常用类

## ThreadLocal

为每一个线程提供独立的副本。不会相互影响。

用处：保存线程的独立变量。对一个线程类（继承自Thread)  
当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。常用于用户登录控制，如记录session信息。

实现：每个Thread都持有一个TreadLocalMap类型的变量（该类是一个轻量级的Map，功能与map一样，区别是桶里放的是entry而不是entry的链表。功能还是一个map。）以本身为key，以目标为value。  
主要方法是get()和set(T a)，set之后在map里维护一个threadLocal -> a，get时将a返回。ThreadLocal是一个特殊的容器。

ThreadLocal的接口方法

ThreadLocal类接口很简单，只有4个方法：

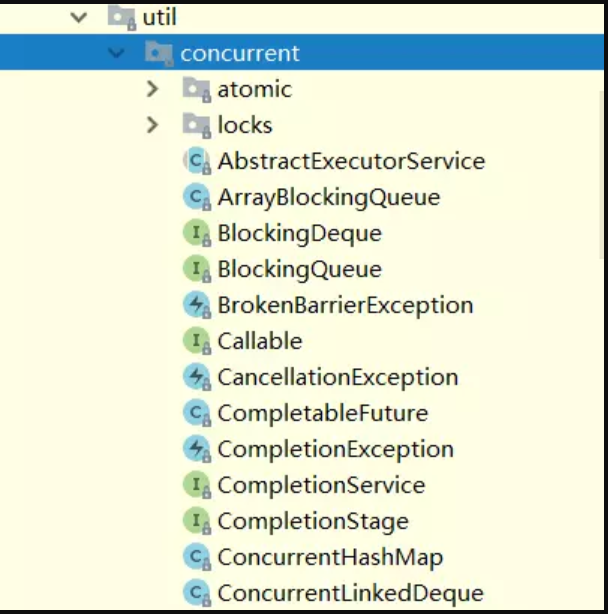
* void set(Object value)设置当前线程的线程局部变量的值。
* public Object get()该方法返回当前线程所对应的线程局部变量。
* public void remove()将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。
* protected Object initialValue()返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的缺省实现直接返回一个null。

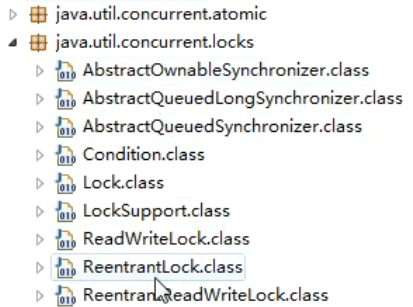
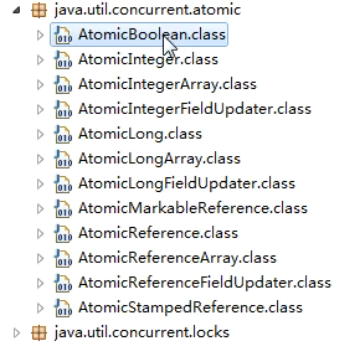
下面看个例子：

1. package com.derek.cultivate;
2. public class Bank {
3. //设置count为线程独立变量。
4. **private static ThreadLocal<Integer> count = new ThreadLocal<Integer>()**{
5. @Override
6. protected Integer initialValue() {
7. // TODO Auto-generated method stub
8. return 0;
9. }
10. };
11. // 存钱
12. public void addMoney(int money) {
13. count.set(count.get()+money);
14. System.out.println(System.currentTimeMillis() + "存进：" + money);
15. }
16. // 取钱
17. public void subMoney(int money) {
18. if (count.get() - money < 0) {
19. System.out.println("余额不足");
20. return;
21. }
22. count.set(count.get()- money);
23. System.out.println(+System.currentTimeMillis() + "取出：" + money);
24. }
25. // 查询
26. public void lookMoney() {
27. System.out.println("账户余额：" + count.get());
28. }
29. }
30. package com.derek.cultivate;
31. public class SyncThreadTest {
32. public static void main(String args[]){
33. final Bank bank=new Bank();
34. Thread tadd=new Thread(new Runnable() {
35. @Override
36. public void run() {
37. // TODO Auto-generated method stub
38. while(true){
39. try {
40. Thread.sleep(1000);
41. } catch (InterruptedException e) {
42. // TODO Auto-generated catch block
43. e.printStackTrace();
44. }
45. bank.addMoney(100);
46. bank.lookMoney();
47. System.out.println("\n");
48. }
49. }
50. });
51. Thread tsub = new Thread(new Runnable() {
52. @Override
53. public void run() {
54. // TODO Auto-generated method stub
55. while(true){
56. // bank.subMoney(100);
57. bank.lookMoney();
58. System.out.println("\n");
59. try {
60. Thread.sleep(1000);
61. } catch (InterruptedException e) {
62. // TODO Auto-generated catch block
63. e.printStackTrace();
64. }
65. }
66. }
67. });
68. tsub.start();
69. tadd.start();
70. }
71. }

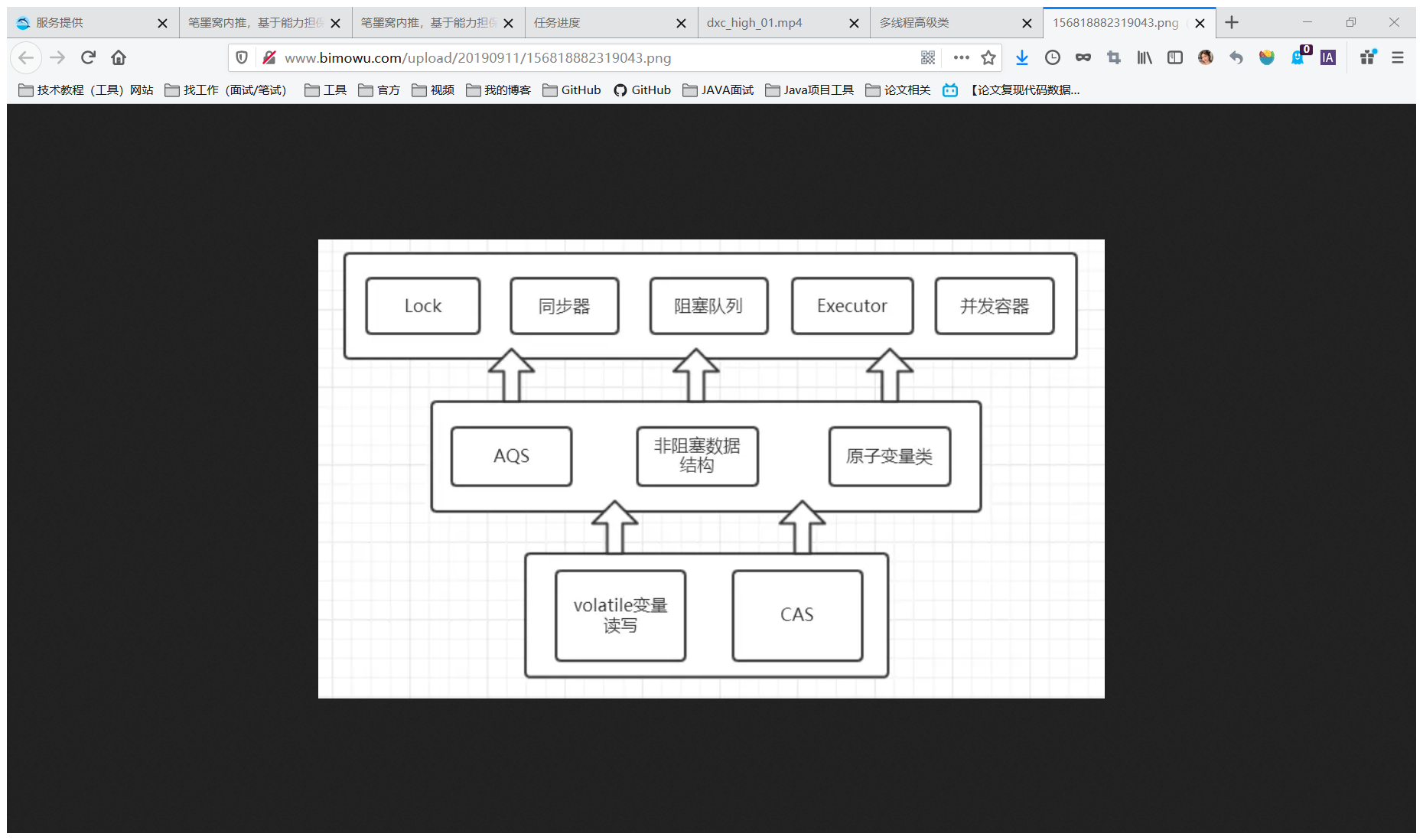
运行之后发现，有个线程账户一直为0，是的，这就是ThreadLocal的数据隔离  
ThreadLocal与同步机制  
a.ThreadLocal与同步机制都是为了解决多线程中相同变量的访问冲突问题。  
b.前者采用以”空间换时间”的方法，后者采用以”时间换空间”的方式

## java.util.concurrent





这些类都是依赖于volatile变量和CAS实现的



原子类：

原子变量类相当于一种泛化的volatile变量，能够支持原子的和有条件的读-改-写操作。

* 标量类（Scalar）：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference
* 数组类：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray
* 更新器类：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater
* 复合变量类：AtomicMarkableReference，AtomicStampedReference

AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference这四种基本类型用来处理布尔，整数，长整数，

其内部实现不是简单的使用synchronized，而是一个更为高效的方式CAS (compare and swap) + volatile和native方法，从而避免了synchronized的高开销，执行效率大为提升。

### [详解java并发原子类AtomicInteger（基于jdk1.8源码分析）](https://zhuanlan.zhihu.com/p/87141904)

### 比较和交换 CAS

compareAndSwapInt又叫做CAS，如果将来找工作，这个不清楚的话，基本上可以告别java这个方向了。

**CAS操作包含三个操作数——内存位置、预期原值及新值。执行CAS操作的时候，将内存位置的值与预期原值比较，如果相匹配，那么处理器**

**会自动将该位置值更新为新值，否则，处理器不做任何操作。**

**for(;;){**

**//先取出AtomicInteger的当前值**

**int current=get();**

**//对当前值加1操作**

**int next=current+1;**

**//通过compareAndSet方法比较当前值有没有被其它线程修改过，若修改过返回false则再次进入compareAndSet方法判断**

**if(compareAndSet(current,next))**

**return current;**

**}**

问题： ABA 问题。（解决：每次改动的时候加一个版本号）

### ReentrantLock类

实现了Lock接口的锁。 与使用synchronized方法和代码块具有相同的基本行为和语义，并且扩展了其能力。主要应用 lock() : 获得锁； unlock() : 释放锁 这两个方法来同步操作。

1. private int account = 100;
2. private ReentrantLock lock = new ReentrantLock();
3. public int getAccount() {
4. return account;
5. }
6. //同步方法
7. public void save(int money) {
8. lock.lock();
9. try {
10. account+=money;
11. } finally {
12. lock.unlock();
13. }
14. }

synchronized和ReentrantLock对比；  
1.synchronized是java自带的关键字，直接作用与jvm层面，跟obeject类紧密相关，不会生产死锁；ReentrantLock是以一个java类，加锁和解锁必须成对出现，否者会出现死锁的可能。  
2.Lock是可以中断锁，Synchronized是非中断锁，必须等待线程执行完成释放锁。  
3.lock比synchronized更灵活，lock可实现公平锁，读写锁等等应用场景更灵活。