JVM为每一个执行任务的线程分配一个独立的缓存，提高效率。共享变量的运算在缓存在执行之后刷新到主存中

每个线程对共享数据的操作彼此不可见---🡪出现了内存可见性问题

线程的三个特性：原子性，有序性，可见性

Volatile与synchronized 1.volatile不具备“互斥性” 2.volatile不能保证变量的原子性

3.volatile只是保证内存可见性

线程安全三要素：原子性，可见性，有序性

Volatile

Concurrent.atomic下面原子变量

1. 使用volatile 保存内存可见
2. CAS算法保存数据的原子性（CAS是硬件对并发操作共享数据的支持）

内存值V：相当于共享内存里面的值（可能由于其他线程进来，变量值已经改变，因此可能导致下一个线程读到的预估值与内存制不一样，因此下一个线程的V!=A，所以不进行赋值，----🡪相当于什么都不做）

预估值（旧值）A：相当于线程中变量的值

更新值B

当V==A时，才把B的值给V，否则不进行任何操作

一般来说我们会先得到内存的值，调用compareandset的方法会再次获取一次内存的值，然后我们将预估计与传入的值进行比较

ConcurrentHashMap

Hashtable：线程安全，效率低

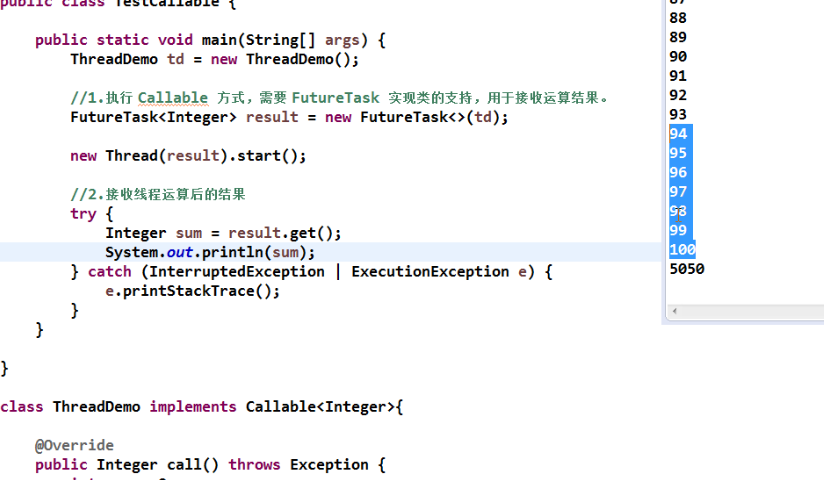
Hashmap：线程不安全，效率高

16个段segment,每一个段里面16长度的hash表（）

CountDownLatch（闭锁）:在完成某些运算时，只有其他所有运算全部完成，当前运算才继续执行

Callable接口（创建线程的第三种方式）：相较于Runnable 1.有返回值 2.抛出异常

需要FutureTask实现类的支持，FutureTask用于接收结果



线程没结束 futureTask.get（）操作不会执行 因此futureTask也可以用于闭锁

Lock

解决线程安全的方式：1.同步代码块 2.同步方法3.同步锁lock

Synchronize 存在虚假唤醒问题，因此需要用在循环中

线程八锁：某一个时刻只能有一个线程持有对象的锁

1.非静态的锁为this 静态方法的锁的对应为实例

2.某一个时刻内只能有一个线程持有锁

普通方法的锁为this，静态方法的锁是A.class 所以普通方法与静态方法无竞争关系

线程池：

Executor：

Executorservice：

ThreadpoolExecutor：线程池的实现类（3）

ScheduleExecutorService ：负责线程调度（4）

ScheduleThreadpoolExecutor ：实现了4继承了3

工具类：Executors --🡪用来创建线程池 （1.创建固定大小的线程池 2.创建无界线程池 3.创建单个线程的线程池4.创建固定大小的线程池，延迟或定时执行任务）

Fork/Join框架