线程不安全最有效的方法及时同步控制

synchronized如果遇到异常自动释放锁,即解决可见性又解决了原子性问题

volatile实现线程间的通信,只解决了可见性问题

同一个线程内的不同方法可以多次拿到同一个锁，并且不会死锁。

synchronized获得的锁是可重入锁，可重入锁就是可以被一个线程多次获得的锁

比较简单的原子性问题,可以用AtomXXX,例如AtomInteger来解决

都必须在锁里面定义🡪wait等待,会释放锁 notify唤醒 , 唤醒一个拥有相同类型锁的代码,但本身不释放锁🡪notifyAll叫醒全部

countdownlatch/cyclicbarrier/semaphore

门闩🡪 countdownlatch🡪 Countdownlatch latch = new Countdownlatch(1);

latch.await(); 使用门闩,线程等待. 其他线程调用latch.countDown();使数字变为零,门闩打开

reentrantlock重入锁

Lock lock = new Reentrantlock();

lock.lock();手工锁,类似于synchronized(this),但必须要手动释放,最后一定要在finally里释放锁lock.unlock();

boolean locked = lock.tryLock();尝试拿到锁,拿到锁返回true.拿不到锁,返回false,方法内可以设置等待时间参数,最后根据返回值判断是否解锁

lock.lockInterruptibly();可以被其他线程的lock.interrupt();方法打断的锁

在new ()里面传true,就变为公平锁,公平锁会轮流获得锁

与synchronized不同点🡪可以自动释放/乐观悲观锁/公平锁非公平锁🡪性能上没有区别

条件调用

Lock lock = new Reentrantlock();

Condition condition = lock.newCondition();

condition.await();等待该线程

condition.signalAll();通知叫醒拥有该锁的线程

线程局部变量,与volatile相反🡪

ThreadLocal threadLocal = new ThreadLocal();

threadLocal.set();

threadLocal.get();

让一个线程改变的值只能在该线程功能中生效,无法线程通讯,类似于每个线程拷贝一份

Runnable的run()和Callable的call()方法都是设计出给新线程调用的，run()方法没有返回值，call()方法是可以有返回值的，并且返回值自己指定。

run重写方法没有定义异常，不可以抛异常，但是call方法可以抛异常。

接口

Executor,ExecutorService;

工具类：例如Arrays是操作数组的工具类，Collections是操作集合的工具类

Executors;操作Executor的工具类

线程池ThreadPool：

ExecutorService service = Executors.newFixThreadPool(5);

创建一个固定个数的线程池，

service.execute();

service.submit();

线程池的作用，线程执行完任务后，并不会关闭，而是直接区执行新的内容。节省了线程开启和关闭所消耗的资源。java内核态线程管理。

如果线程数超过线程池容纳数，线程就会被放入队列，等待线程池空闲后进入线程池。

线程池关闭方法：service.shutdown();

service.shutdownnow();该方法会立即关闭线程池，不等线程做完。

当线程正常关闭时，线程的信息会显示关闭，但未执行完的线程也会继续执行直到全部结束。

任务列表队列🡪线程池🡪结束任务队列

Future 未来:

未来任务，在未来获取任务的返回值

FutureTask<> task = new FutureTask<>(传入Callable类型的参数);

new Thread(task).start();//线程执行；

task.get();//这个方法会使当前线程阻塞，直到task的线程执行完并拿到返回值

第二种线程池：

弹性线程池，最开始线程池内线程数为0，线程不够启新线程，有空余线程使用老线程，线程默认超过60秒空闲关闭

ExecutorService service = Executors.newCachedThreadPool();

第三种线程池：线程池内永远只有一个线程，作用保证线程按顺序执行

ExecutorService service = Executors.newSingleThreadExcutor();

第四种线程池：定时器线程池

ExecutorService service = Executors.newScheduledThreadPool(4);

线程是可以复用，而timmer每次都new一个新的线程

以固定频路执行某个任务，有四个参数

service.scheduleAtFixedRate(实现线程接口的对象，第一个任务在几秒后自行时间，每个隔多长时间重复执行，时间单位)

第五种线程池：工作窃取线程池

ExecutorService service = Executors.newWorkStealingPool();

每一个线程池都有一个队列，窃取线程池空闲时会去其他线程池里拿未执行的线程执行，主动找活干

第六种线程：

ForkJoinPool

最后线程池的原理：所有线程池最后都是调用下面这个类

ThreadPoolExecutor

高并发的容器：