- 多线程总结
  - 基本概念
    - 线程概念
    - 生命周期及转换
    - 基本实现方法
    - 线程的基本控制
  - 线程同步
    - synchronized
      - 底层实现
      - 适用范围
      - synchronized状态变化
    - ReentranLock
      - ReetranLcok的基本使用方法
      - ReetranLcok特点
      - 非公平锁加锁过程
      - 公平锁加锁过程
      - 释放锁过程
      - 扩展
  - 线程池
    - 基本概念
      - 什么是线程池
      - 使用线程池的好处
      - 基本组成部分
    - 几个线程池的基本使用
      - newSingleThreadExecutor
      - newFixedThreadExecutor
      - newCacheThreadExecutor
      - newScheduleThreadExecutor
    - 线程池的执行策略
      - 源码分析
    - 线程池的拒绝策略
  - 锁分类
  - volitile（结合jvm第12章看）
    - 可见性
    - 原子性：
    - 有序性
    - volatile的应用场景
    - 实现机制
  - ThreadLocal
    - 关键内部类 ThreadLocalMap

多线程总结

基本概念

线程概念

线程是进程的执行单元，它可以拥有自己的堆栈、程序计数器和自己的局部变量，但不能拥有系统资源。

线程和进程的区别 进程和线程的主要差别在于它们是不同的操作系统资源管理方式。

- 进程是cpu分配系统资源的最小单位，有自己的独立地址。而线程是cpu调度的最小单位
- 线程是在进程下运行的，一个进程可以包含多个线程
- 在数据共享上，同一进程下的线程容易共享数据，而不同进程间比较麻烦

引入线程的优势

- 在进程内创建、终止线程比创建、终止进程要快。对进程的创建终止，涉及到资源的分配释放等.
- 同一进程内的线程间切换比进程间的切换要快,尤其是用户级线程间的切换
- 数据共享速度线程比进程更快。

生命周期及转换

新建和就绪状态

- 当使用new关键字创建线程后，该线程就处于新建状态，此时仅由JAVA虚拟机为其分配内存，并初始化其成员变量的值
- 当线程对象调用了start方法后，该线程就处于就绪状态（等待被CPU调用），java虚拟机会为其创建方法调用
- 调用start方法，会将run方法视为线程执行体来处理，而调用run方法，则run方法会被作为一个普通方法立即执行，而不是线程执行体。
- 调用run方法后，该线程就已经不再处于新建状态，此时不能再调用其start方法，否则会报IllegalThreadStateExceptino错误。

运行和阻塞状态

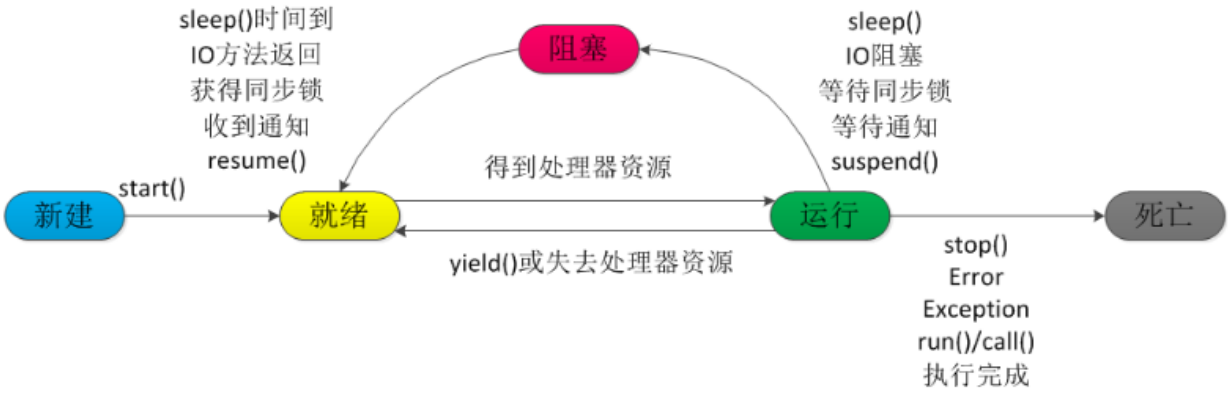
- 当处于就绪状态的线程获得了CPU，开始执行run方法的方法执行体，则该线程处于运行状态
    - 当发生如下情况时，线程将会进入阻塞状态
    - 线程调用sleep()方法主动放弃所占用的处理器资源,不会释放锁

- 线程调用了一个阻塞式IO方法，在该方法返回之前，该线程被阻塞
- 线程试图获得一个同步监视器，但该同步监视器正被其他线程所持有。关于同步监视器的知识、后面将存更深入的介绍
- 线程被wait了，在等待某个通知（notify）
- 程序调用了线程的suspend()方法将该线程挂起。但这个方法容易导致死锁，所以应该尽量避免使用该方法

- 线程解除阻塞，进入就绪状态
  - 调用sleep()方法的线程经过了指定时间。
  - 线程调用的阻塞式IO方法已经返回。
  - 线程成功地获得了试图取得的同步监视器。
  - 线程正在等待某个通知时，其他线程发出了个通知。
  - 处于挂起状态的线程被调甩了resdme()恢复方法

线程死亡

- run方法或者call方法执行结束，线程正常结束
- 线程抛出一个未捕获的Exception或者Error
- 直接调用线程的stop方法来结束线程---容易导致死锁。



基本实现方法

继承Thread类实现多线程，通过继承Thread的子类，即可以代表线程对象。

```
public class FirstThread extends Thread{
    private  int i ;
    public void  run(){
        for(; i<1000;i++){
            System.out.println(getName()+ " "+ i);
        }
    }

    public static void main(String[] args) {
        for(int j =0;j<100;j++){
            System.out.println(Thread.currentThread().getName()+ " " + j);
            if(j == 20){
```

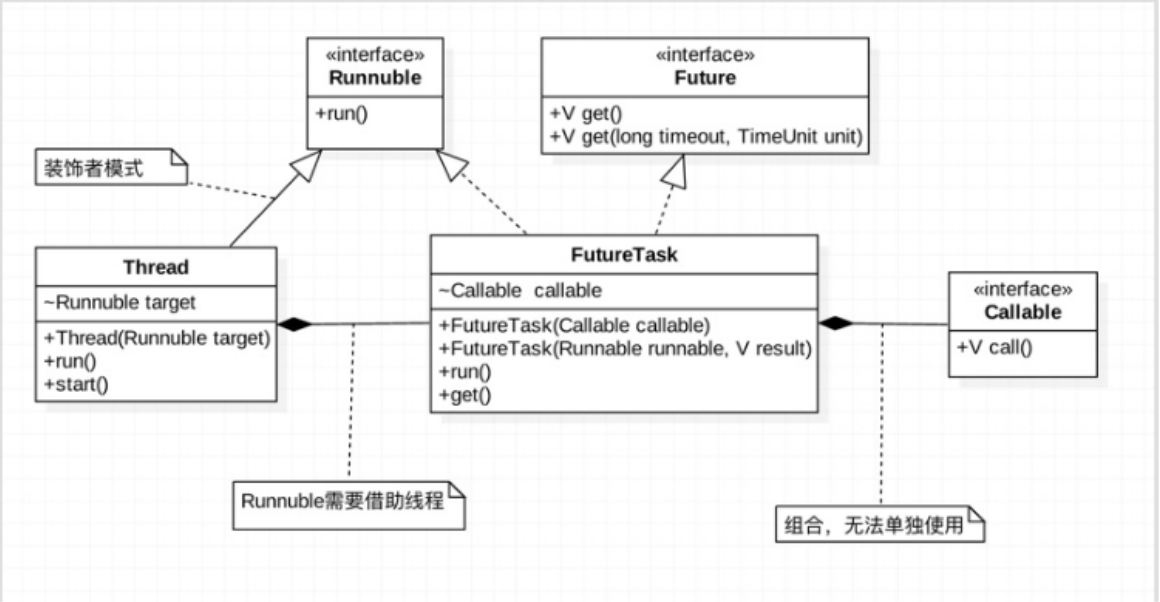
```
                new FirstThread().start();
                new FirstThread().start();
            }
        }
    }
}
```

实现Runnable接口实现多线程，而通过实现Runnable的类对象只能作为线程对象的target。

```
public class FirstRunnable implements  Runnable {
    private  int i;
    @Override
    public void run() {
        for(;i<100;i++){
            System.out.println(Thread.currentThread().getName()+ " "+ i);
        }
    }

    public static void main(String[] args) {
        for(int j =0 ;j<100; j++){
            System.out.println(Thread.currentThread().getName()+ " "+ j);
            if(j == 20){
                FirstRunnable firstRunnable = new FirstRunnable();
                new Thread(firstRunnable,"新线程1").start();
                new Thread(firstRunnable,"新线程2").start();
            }
        }
    }
}
```

先用lambda表达式创建Callable接口的实现类（也可以先创建callable实现类，再创建callable对象），并在其中重写call方法 用futureTask类来包装callable对象，该FutureTask对象封装了Callable对象的call方法返回值 通过futuretask实现类来实现线程，将futturetask的实例作为Thread对象的target来创建并启动线程。 调用futuretask对象的get方法来获取返回值。



Callable与Runnable的功能大致相似，Callable中有一个call()函数，**但是call()函数有返回值**，而Runnable的run()函数不能将结果返回给客户程序。

Future就是对Callable任务的执行结果进行取消、查询是否完成、获取结果、设置结果操作。其中的get()方法就是用来得到Callable的call()结果的。

FutureTask是Future的具体实现类，实现了get()等方法来对控制Callabel的行为，又因为Thread只能执行Runnable，所以FutureTask实现了Runnable接口。

因为FutureTask需要在Thread中执行，所以需要在run()方法中完成具体的实现：

线程的基本控制

join 是线程对象的方法，在某个程序执行流中调用其他线程的join方法，调用线程将被阻塞，直到被join的方法加入的join线程执行完为止。

```
public class JoinThread implements Runnable{
    @Override
    public void run() {
        for(int i = 0 ;i< 100;i++){
            System.out.println(Thread.currentThread().getName()+ " " + i);
        }
    }
    public static void main(String[] args) throws InterruptedException {
        new Thread(new JoinThread(),"线程一").start();
        for(int i = 0; i< 100; i++){
            System.out.println(Thread.currentThread().getName()+ " "+ i);
            if(i == 20){
                Thread thread = new Thread(new JoinThread(), "join  Thread");
                thread.start();
                thread.join();
            }
        }
```

```
    }
}
```

sleep sleep()是Thread类的静态方法，可通过调用，让当前正在执行的线程暂停一段时间，并进入阻塞状态。这时cpu通常会调用其他的就绪线程。该线程睡眠结束后进入就绪状态，也不会立马被执行。

yield yield()也是Thread类的静态方法，通过调用停止当前线程，但不会阻塞，而是直接进入就绪状态，让其他优先级相同或者更高的线程进入，如果没有，则还是自己运行。

-

```
public class YieldThread implements  Runnable{

    @Override
    public void run() {
        for(int i =0 ; i < 100; i++){
            System.out.println(Thread.currentThread().getName()+ "  "+ i);
            if(i == 10){
                Thread.yield();
            }
        }
    }

    public static void main(String[] args) {
        Thread a1 = new Thread(new YieldThread(),"low");
        a1.start();
        a1.setPriority(Thread.MIN_PRIORITY);
        Thread a2 = new Thread(new YieldThread(),"high");
        a2.setPriority(Thread.MAX_PRIORITY);
        a2.start();
    }
}
```

wait 它是Object类的方法(notify()、notifyAll() 也是Object对象)，必须放在循环体和同步代码块中，执行该方法的线程会释放锁，进入线程等待池中等待被再次唤醒(notify随机唤醒，notifyAll全部唤醒，线程结束自动唤醒)即放入锁池中竞争同步锁

- wait,notify,notifyAll.这三个方法都必须是在同步代码块中才可以运行，不然会报错
- 当线程执行wait时，会把当前的锁释放，然后让出CPU，进入等待状态
- 当执行notify/notifyAll方法时，会唤醒一个处于等待该 对象锁 的线程，然后继续往下执行，直到执行完退出对象锁锁住的区域（synchronized修饰的代码块）后再释放锁。（所以应该在执行notify后就立即退出临界区，否则还要等整个同步代码执行完，才会释放锁）

wait详解

```
public class Service {

    public void testMethod(Object lock) {
        try {
```

```
            synchronized (lock) {
                System.out.println("begin wait() ThreadName="
                        + Thread.currentThread().getName());

                //当ThreadA线程执行lock.wait();这条语句时，释放获得的对象锁lock，并放弃CPU，
进入等待队列。
                lock.wait();
                System.out.println("  end wait() ThreadName="
                        + Thread.currentThread().getName());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void synNotifyMethod(Object lock) {
        try {
            synchronized (lock) {
                System.out.println("begin notify() ThreadName="
                        + Thread.currentThread().getName() + " time="
                        + System.currentTimeMillis());

                //当另一个线程执行lock.notify();，会唤醒ThreadA，但是此时它并不立即释放锁，接
下来它睡眠了5秒钟(sleep()是不释放锁的，事实上sleep()也可以不在同步代码块中调用)，直到第28行，退
出synchronized修饰的临界区时，才会把锁释放。这时，ThreadA就有机会获得另一个线程释放的锁，并从等
待的地方起开始执行。

                lock.notify();
                Thread.sleep(5000);
                System.out.println("  end notify() ThreadName="
                        + Thread.currentThread().getName() + " time="
                        + System.currentTimeMillis());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

线程同步

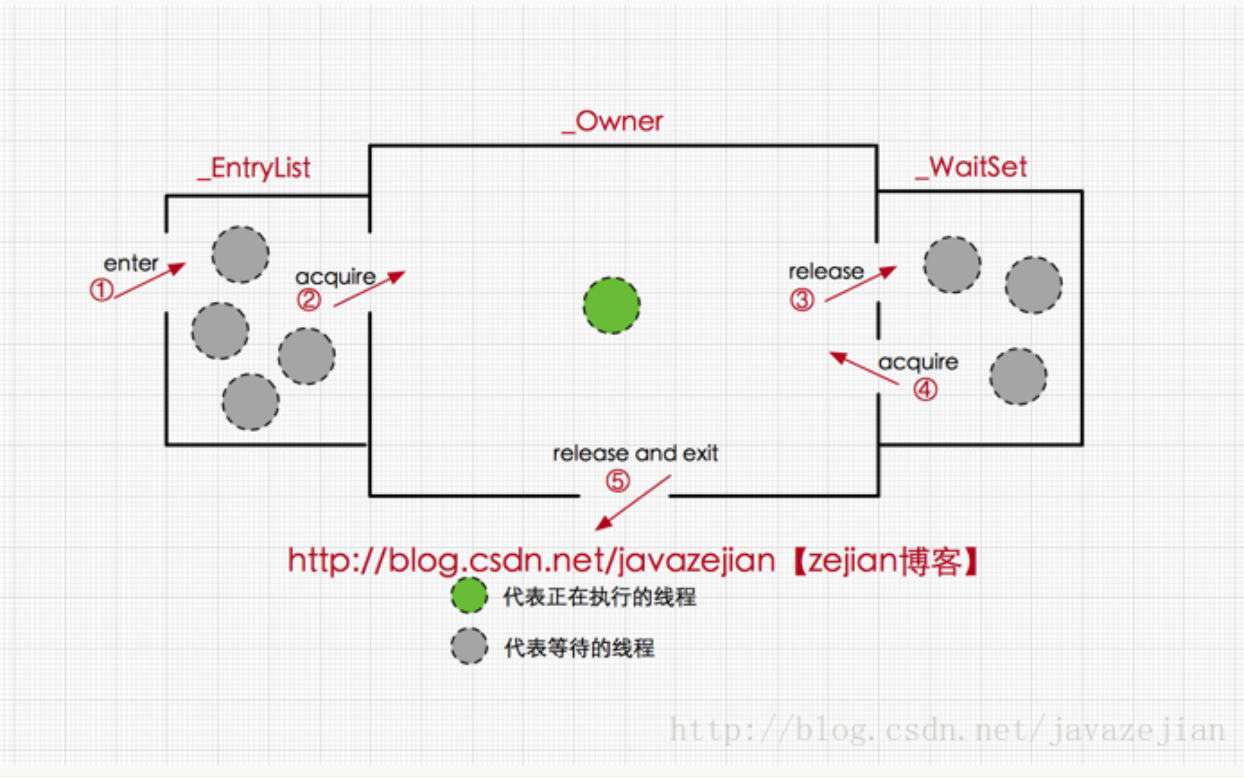
synchronized

底层实现

Synchronized的同步是基于进入和退出监视器对象（Monitor）来实现的。对于同步代码块，编译器会在自动在前后加上monitorenter和monitorexit指令定；对于同步方法，编译器会自动在方法标识上加上ACC_SYNCHRONIZED来表示这个方法为同步方法。
monitor是ObjectMonitor的具体实现。synchronized可以理解为是结合操作系统JVM的底层来实现的。

```
ObjectMonitor() {
    _header       = NULL;
    _count        = 0; //记录个数
    _waiters      = 0,
    _recursions   = 0;
    _object       = NULL;
    _owner        = NULL;
    _WaitSet      = NULL; //处于wait状态的线程，会被加入到_WaitSet
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ;
    FreeNext      = NULL ;
    _EntryList    = NULL ; //处于等待锁block状态的线程，会被加入到该列表
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
  }
```

ObjectMonitor中有两个队列，_WaitSet 和 _EntryList，用来保存ObjectWaiter对象列表(每个等待锁的线程都会被封装成ObjectWaiter对象)，_owner指向持有ObjectMonitor对象的线程，当多个线程同时访问一段同步代码时，首先会进入 _EntryList 集合，当线程获取到对象的monitor后进入_Owner区域并把monitor中的owner变量设置为当前线程同时monitor中的计数器count加1，若线程调用wait()方法，将释放当前持有的monitor，owner变量恢复为null，count自减1，同时该线程进入 WaitSet集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)



适用范围

根据修饰对象分类

- 修饰代码块
  - synchronized(this|Object){}
  - synchronized(XXX.class){}
- 修饰方法
  - 修饰静态方法
  - 修饰非静态方法

根据获取的锁分类

- 获得对象锁
  - synchronized(this|objcet){}
  - 修饰非静态方法
- 获取类锁
  - synchronized(xxx.class){}
  - 修饰静态方法

注意

- 当synchronized用来修饰静态方法或者类时，将会使得这个类的所有对象都是共享一把类锁，导致线程阻塞，所以这种写法一定要规避
- 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。
- 每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。
- 实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

synchronized状态变化

锁的释放

- 线程的同步方法或者同步代码块正常运行结束，释放同步监视器
- 在同步代码块，同步方法中遇到了break，return等终止该代码块，方法的执行。释放同步监视器
- 在同步代码块，同步方法中遇到了未经处理的error,Exception.导致该代码块，方法异常结束。
- 当前线程执行同步代码块或同步方法时，程序执行了同步监视器对象的wait()方法，则当前线程暂停，并释放同步监视器

不会释放锁

- 调用Thread的静态方法，sleep(),yield()不会释放同步监视器
- 线程执行同步代码块时，其他线程调用了该线程的suspend()方法将其挂起，该线程不会释放同步监视器。

ReentranLock

参考博客

是java为了实现同步锁而写入的方法。JAVA SE5.0之后并发包中新增了Lock接口用来实现锁的功能，它提供了与synchronized关键字类似的同步功能，只是在使用时需要显式地获取和释放锁，缺点就是缺少像synchronized那样隐式获取释放锁的便捷性，但是却拥有了锁获取与释放的可操作性，可中断的获取锁以及超时获取锁等多种synchronized关键字所不具备的同步特性。

ReetranLcok的基本使用方法

```
class X{
    private final ReentrantLock lock = new ReentrantLock();//参数默认false，属于不公平锁
    ...
    public void m(){
        lock.lock();//加锁
        try{
            //method body
        }finally{
            lock.unlock();//释放锁
        }
    }
}
```

ReetranLcok特点

可重入性

也就是说，一个线程可以对已被加锁的ReetranLock锁再次加锁，ReetranLock锁对象户会维持一个计数器来追踪lock()方法的嵌套调用，线程在每次调用lock()加锁后，必须显示地调用unlock()来释放锁，所以一段被锁保护的代码可以调用另一个被相同锁保护的方法。

```
public class ReentrantLockTest {
    private Lock lock = new ReentrantLock();
    public void method1() {
        lock.lock();
        try {
            System.out.println("方法1获得ReentrantLock锁运行了");
            method2();
        } finally {
            lock.unlock();
        }
    }
    public void method2() {
        lock.lock();
        try {
            System.out.println("方法1里面调用的方法2重入ReentrantLock锁,也正常运行了");
        } finally {
            lock.unlock();
        }
    }
    public static void main(String[] args) {
        new ReentrantLockTest().method1();
```

```
    }
}
```

Node

Reetranlock,有3个内部类Sync,FairSync,NonFairSyn,公平锁和非公平锁都继承自Sync，而Sync是继承自AbstractQueuedSynchronizer类，在AbstractQueuedSynchronizer类中主要实现了用来存储等待结点的双向队列，以及队列插入线程结点，获取当前结点状态，强行插入等操作逻辑。

公平和非公平锁的队列都基于锁内部维护的一个双向链表，表结点Node的值就是每一个请求当前锁(失败)的线程。公平锁则在于每次都是依次从队首取值。非公平锁与公平锁的区别在于新晋获取锁的进程会有多次机会去抢占锁。

公平锁和非公平锁在说的获取上都使用到了 volatile 关键字修饰的state字段， 这是保证多线程环境下锁的获取与否的核心。 但是当并发情况下多个线程都读取到 state == 0时，则必须用到CAS技术，一门CPU的原子锁技术，可通过CPU对共享变量加锁的形式，实现数据变更的原子操作。
volatile 和 CAS的结合是并发抢占的关键。

属性类型与名称	描　　述
int waitStatus	等待状态。 包含如下状态。 ① CANCELLED，值为 1，由于在同步队列中等待的线程等待超时或者被中断，需要从同步队列中取消等待，节点进入该状态将不会变化 ② SIGNAL，值为 -1，后继节点的线程处于等待状态，而当前节点的线程如果释放了同步状态或者被取消，将会通知后继节点，使后继节点的线程得以运行 ③ CONDITION，值为 -2，节点在等待队列中，节点线程等待在 Condition 上，当其他线程对 Condition 调用了 signal() 方法后，该节点将会从等待队列中转移到同步队列中，加入到对同步状态的获取中 ④ PROPAGATE，值为 -3，表示下一次共享式同步状态获取将会无条件地被传播下去 ⑤ INITIAL，值为 0，初始状态
Node prev	前驱节点，当节点加入同步队列时被设置（尾部添加）
Node next	后继节点
Node nextWaiter	等待队列中的后继节点。如果当前节点是共享的，那么这个字段将是一个 SHARED 常量，也就是说节点类型（独占和共享）和等待队列中的后继节点共用同一个字段
Thread thread	获取同步状态的线程

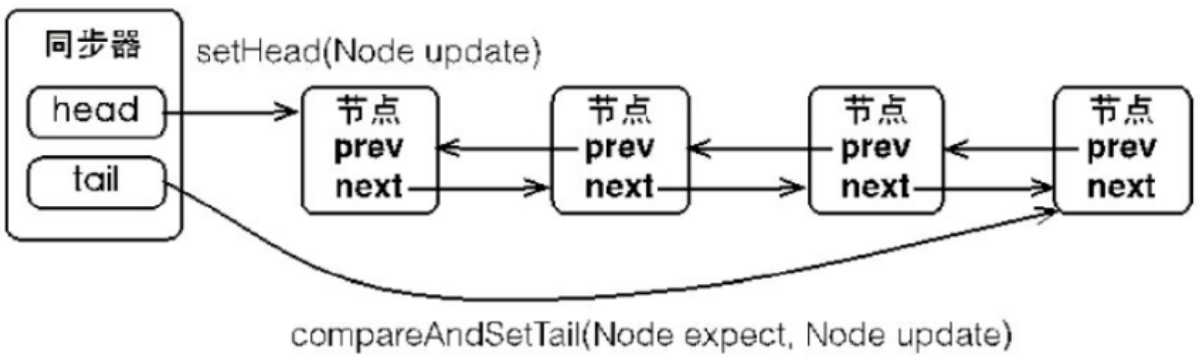


图5-1　同步队列的基本结构

非公平锁加锁过程

多个线程调用lock方法占有锁，利用compareAndSetState()判断当前锁状态，若果是0，则利用CAS让某个线程独占锁，如果锁已经被占有，判断状态是1。其他线程调用acquir()方法来竞争锁（后续会全部进入同步队列或者自悬挂起）。当有其他不在等待队列里的线程A又进来想要调用锁，恰好此前锁被释放，那么A会在同步队列中所有等待获取锁的线程之前抢先获取锁。

```
static final class NonfairSync extends Sync {
        private static final long serialVersionUID = 7316153563782823691L;

        /**
         * Performs lock.  Try immediate barge, backing up to normal
         * acquire on failure.
         */
        final void lock() {
            if (compareAndSetState(0, 1))
                setExclusiveOwnerThread(Thread.currentThread());
            else
                acquire(1);
        }

        protected final boolean tryAcquire(int acquires) {
            return nonfairTryAcquire(acquires);
        }
    }
```

acquire方法：仍然尝试通过tryAcquire()方法获取锁，失败则返回false。根据且规则，会调用&&后面的方法acquireQueued(),其中addWaiter()方法将线程包装Node加入同步队列

```
    public final void acquire(int arg) {
    // tryAcquire()方法也是让新来的线程进行第二次插队的机会！！
    //如果再次获取锁还不成功才会放到队列
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

addWaiter()是对线程进行包装成Node，对node的双向队列状态进行判断，是否为空，不为空直接CAS添加到队尾，如果为空或者CAS添加失败，则用enq()方法强制入队。

acquireQueued的逻辑是：failed 标记最终acquire是否成功， interrupted标记是否曾被挂起过。注意到for(; 跳出的唯一条件就是if (p == head && tryAcquire(arg)) 即当前线程结点是头结点且获取锁成功。从这里我们应该看到，这是一个线程第三次又想着尝试快速获取锁：虽然此时该节点已被加入等待队列，在进行睡眠之前又通过p == head && tryAcquire(arg)方法看看能否获取锁。也就是说只有该线程结点的所有 有效的前置结点都拿到过锁了，当前结点才有机会争夺锁，如果失败了那就通过shouldParkAfterFailedAcquire方法判断是否可以挂起当前结点，等待响应中断。观察 每次调用tryAcquire方法的时机，可以看出作者优化意图：

- 尽量在没入队的时候拿到锁，避免过多队列操作维护成本
- 尽量在睡眠前拿到锁，避免过多上下文切换

```
final boolean acquireQueued(final Node node, int arg) {
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {// 获取前置结点
                final Node p = node.predecessor();
        //如果第一次循环就获取成功那么返回的interrupted是false，不需要自我中断。
        //否则 说明在获取到同步状态之前经历过挂起（返回true）那么就需要自我中断，可以 看acquire
方法中的代码。
                if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return interrupted;
                }
                //如果当前node线程是可挂起的，就调用parkAndCheckInterrupt挂起 ，interrupted
设置为true，表示曾经被中断过
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)//如果tryAcquire出现异常那么取消当前结点的获取
                cancelAcquire(node);
        }
    }
```

公平锁加锁过程

CPU在调度线程的时候是在等待队列里随机挑选一个线程，由于这种随机性所以是无法保证线程先到先得的（synchronized控制的锁就是这种非公平锁）。但这样就会产生饥饿现象，即有些线程（优先级较低的线程）可能永远也无法获取CPU的执行权，优先级高的线程会不断的强制它的资源。

公平锁可以保证线程按照时间的先后顺序执行，避免饥饿现象的产生。但公平锁的效率比较低，因为要实现顺序执行，需要维护一个有序队列。

lock方法对比非公平锁， 没有了if else 也就意味着新来的线程没有插队的机会， 所有来的线程必须扔到队列尾部， acquire方法也会像非公平锁一样首先调用tryAcquire插队试试，但是只有队列为空或着本身就是head，那么才可能成功，如果 队列非空那么肯定被扔到队列尾部去了, 插个毛线。

```
private static ReentrantLock lock=new ReentrantLock(true);
//公平锁的加锁过程
static final class FairSync extends Sync {
        private static final long serialVersionUID = -3000897897090466540L;

        final void lock() {
            acquire(1);
        }

        /**
         * Fair version of tryAcquire.  Don't grant access unless
         * recursive call or no waiters or is first.
```

```
         */
        protected final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
            //对于has方法，若当前线程是头部或者队列为空返回false，然后设置当前线程独占锁，若当前线程有前一个node返回true。保证没有线程插队。
                if (!hasQueuedPredecessors() &&
                    compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            //判断重入锁
            else if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }
    }
```

释放锁过程

公平锁和非公平锁的释放都采用realease，但有两个需要注意的

- 其释放是逐级释放的，也就是说在可重入的场景中，必须等到场景内所有加锁的方法都释放锁后，当前线程才会释放锁
- 释放完后就通过trylease中的unparkSuccessor唤醒后续结点或者新加入的结点来获取锁。

扩展

- synchronized主要依赖JVM底层实现，而Lock是通过编码方式实现，其实现方式差别还是比较大，可以参考之前的一个文章
- synchronized由于其简单方便，只需要声明在方法、代码块上即可，主要是不需要关心锁释放问题，在一般的编程中使用量还是比较大的，但是在真正的并发编程系统中，Lock方式明显优于synchronized：(在之后已经通过自旋锁等对synchronized进行优化，单纯从性能上讲差不多了已经)
- synchronized最致命的缺陷是：synchronized不支持中断和超时，也就是说通过synchronized一旦被阻塞住，如果一直无法获取到所资源就会一直被阻塞，即使中断也没用，这对并发系统的性能影响太大了；Lock支持中断和超时、还支持尝试机制获取锁，对synchronized进行了很好的扩展，所以从灵活性上Lock是明显优于synchronized的

CAS概念 CAS(compareAndSwap)的基本实现原理，它包含3个参数，CAS（V，E，N），V表示要更新变量的值，E表示预期值，N表示新值。仅当V值等于E值时，才会将V的值设为N，如果V值和E值不同，则说明已经有其他线程做两个更新，则当前线程则什么都不做。最后，CAS 返回当前V的真实值。CAS 操作时抱着乐观的态度进行的，它总是认为自己可以成功完成操作。

当多个线程同时使用CAS 操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会挂起，仅是被告知失败，并且允许再次尝试，当然也允许实现的线程放弃操作。基于这样的原理，CAS 操作即使没有锁，也可以发现其他线程对当前线程的干扰。

与锁相比，使用CAS会使程序看起来更加复杂一些，但由于其非阻塞的，它对死锁问题天生免疫，并且，线程间的相互影响也非常小。更为重要的是，使用无锁的方式完全没有锁竞争带来的系统开销，也没有线程间频繁调度带来的开销，因此，他要比基于锁的方式拥有更优越的性能。

但底层的CPU和java其实是实现了原子操作，保证CAS的比较和交换是一个原子操作，不被打断

- cpu通过总线锁定。或者通过缓存锁定(当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。)
- java通过unsafe这个类的compareAndSwapInt来实现，它本质是一个.class的字节码文件，通过getClassLoader反射来实现，具体不了解了

缺点：最著名的就是 ABA 问题，假设一个变量 A ，修改为 B之后又修改为 A，CAS 的机制是无法察觉的，但实际上已经被修改过了。如果在基本类型上是没有问题的，但是如果是引用类型呢？这个对象中有多个变量，我怎么知道有没有被改过？聪明的你一定想到了，加个版本号啊。每次修改就检查版本号，如果版本号变了，说明改过，就算你还是 A，也不行。

```
//伪代码
void function(int b) {
   int backup = a;
   int c = a + b;
   compareAndSwap(a, backup, c);
}

void compareAndSwap(int backup ,int c ){
       if (a == backup) {
           a = c;
       }
}
```

线程池

基本概念

什么是线程池

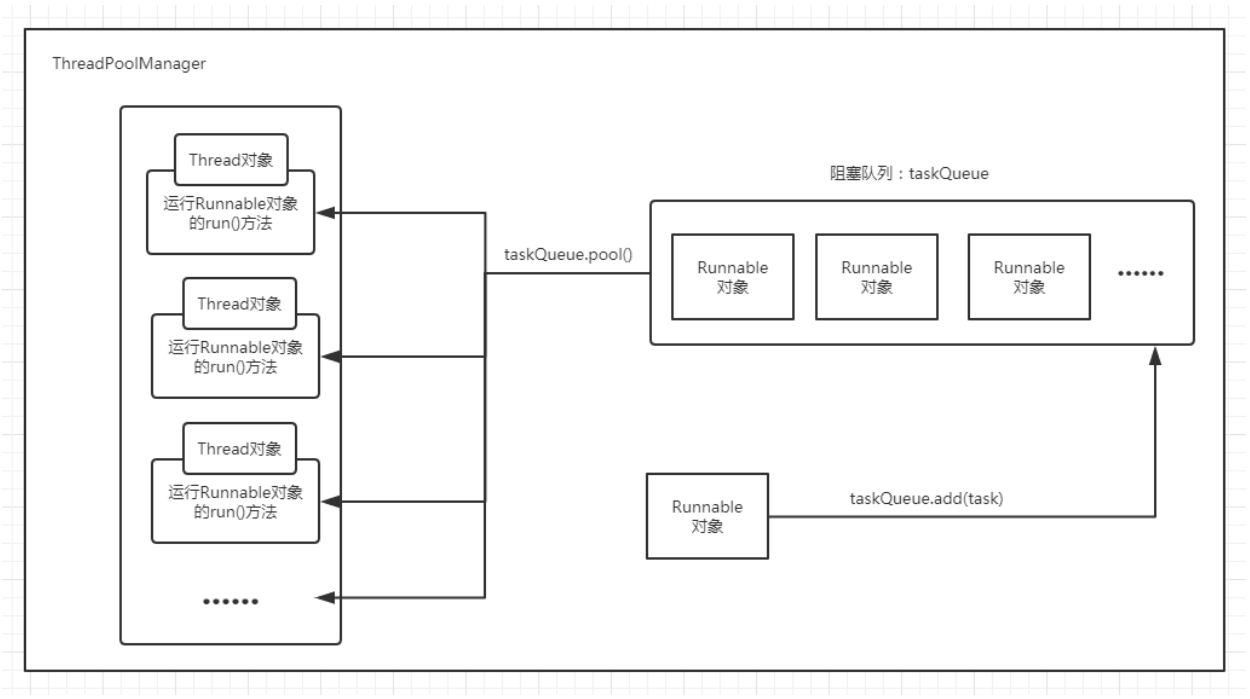
线程池的基本思想是一种对象池，在程序启动时就开辟一块内存空间，里面存放了众多(未死亡)的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源

使用线程池的好处

- 减少了创建和销毁线程的次数，每个工作线程都可以被重复利用个，可以执行多个任务
- 能有效控制线程的最大并发数，根据系统的可承受能力，调整线程池中工作线程的数目，防止因消耗过多，而把服务器累趴下（每个线程大约需要1MB的内存）
- 对线程进行简单的管理，比如：延时执行、定时循环执行等。可以通过线程池很好的进行实现。

基本组成部分

- 线程池管理器：用于创建并管理创建线程池，包括创建线程池，新建线程池，添加新任务
- 工作线程：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务
- 任务接口：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等
- 任务队列：用于存放没有处理的任务，提供一种缓冲机制



几个线程池的基本使用

newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

可用理解为newFixedThreadPool(1)。

newFixedThreadExecutor

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个

新线程。

```
    //通过给线程池submit  Runnalbe对象来执行
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(3);
        Runnable target = () ->{
            for(int i = 0; i <100; i++){
                System.out.println(Thread.currentThread().getName()+ " "+ i);
            }
        };
        executorService.submit(target);
        executorService.submit(target);
        executorService.shutdownNow();
    }

    //通过execute方法中加入Runnable对象来执行，这个时候默认一次性调用所有可用的线程你来执行。
     public static void main(String[] args) {
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 10; i++) {
            final int index = i;
            fixedThreadPool.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        System.out.println(Thread.currentThread().getName()+ " "+ index);
                        Thread.sleep(2000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            });
        }
    }
```

newCacheThreadExecutor

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

newScheduleThreadExecutor

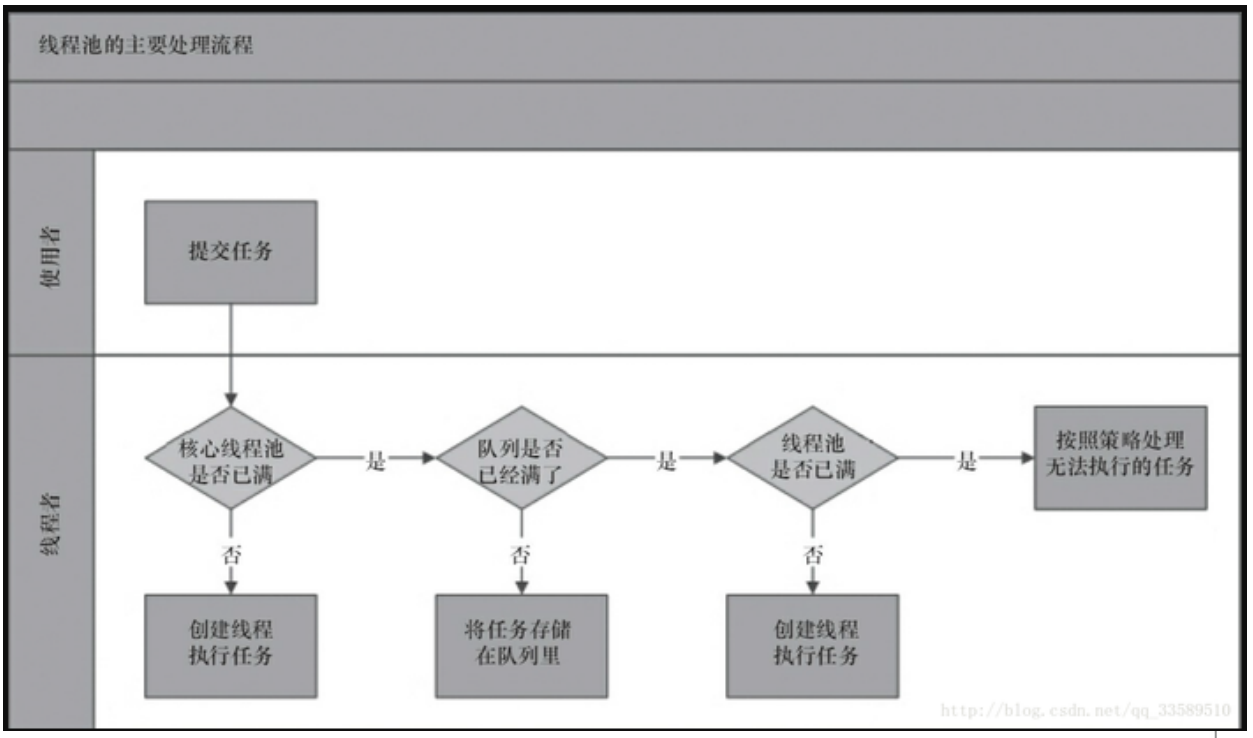
创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

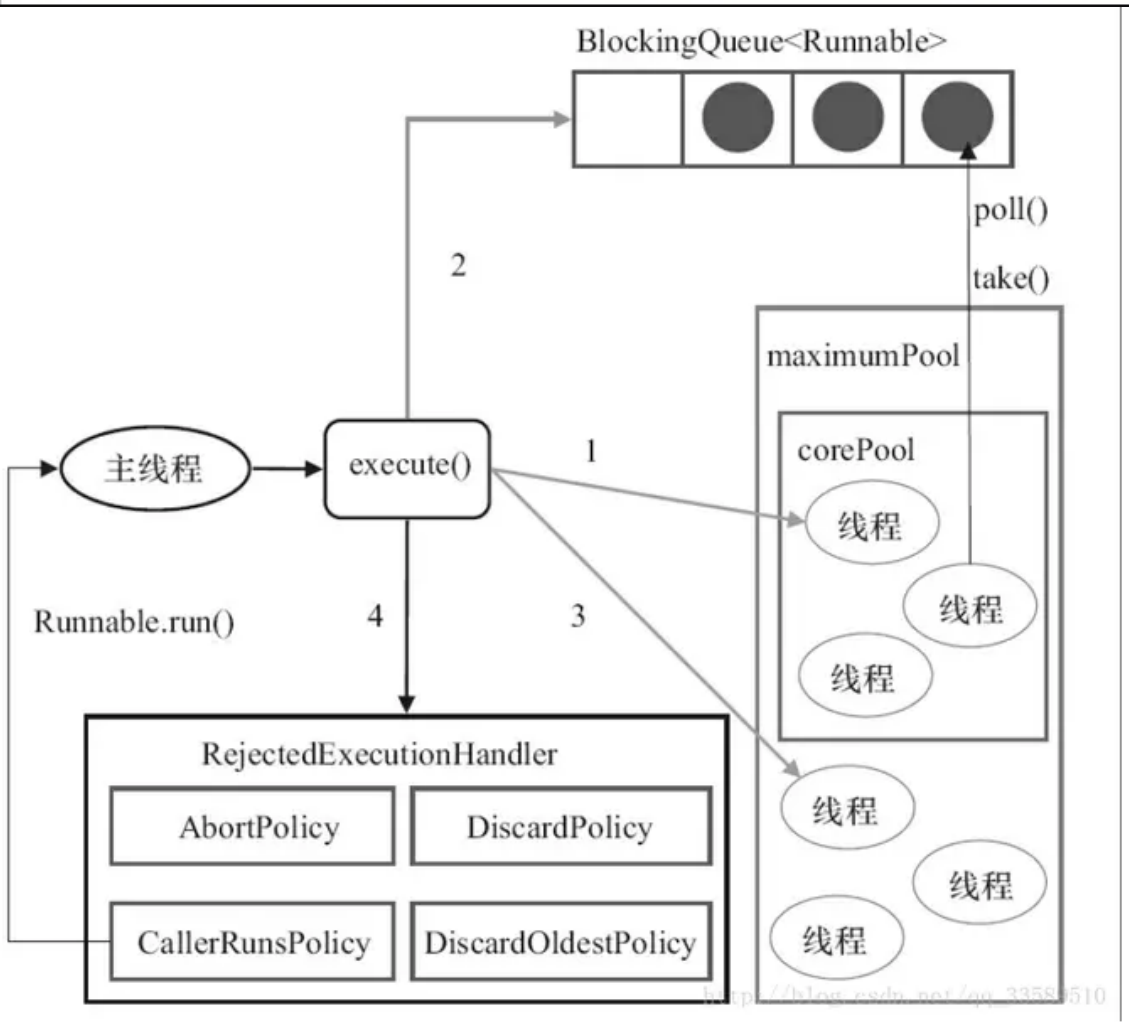
注意这里不是利用execute方法，而是shceduleAtFixedRate();

```
public static void main(String[] args) {
        ScheduledExecutorService scheduledThreadPool =
Executors.newScheduledThreadPool(5);
```

```
        scheduledThreadPool.scheduleAtFixedRate(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName() + ": delay 1 seconds,
and excute every 3 seconds");
            }
        }, 1, 3, TimeUnit.SECONDS);// 表示延迟1秒后每3秒执行一次
    }
```

线程池的执行策略





1、提交线程，线程数量未达到corePoolSize，则新建一个线程(核心线程)来执行任务 2、线程数量达到了corePools，则将任务移入队列BlockingQueue等待 3、blockingqueue队列已满，无法加入，新建线程(非核心线程)执行任务（执行这一步需要全局锁） 4、队列已满，总线程数达到maximumPoolSize，就会触发拒绝策略（抛出异常）。

源码分析

```
public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        /*
         * Proceed in 3 steps:
         *
         * 1. If fewer than corePoolSize threads are running, try to
         * start a new thread with the given command as its first
         * task.  The call to addWorker atomically checks runState and
         * workerCount, and so prevents false alarms that would add
         * threads when it shouldn't, by returning false.
         *
         * 2. If a task can be successfully queued, then we still need
         * to double-check whether we should have added a thread
         * (because existing ones died since last checking) or that
         * the pool shut down since entry into this method. So we
         * recheck state and if necessary roll back the enqueuing if
         * stopped, or start a new thread if there are none.
         *
         * 3. If we cannot queue task, then we try to add a new
         * thread.  If it fails, we know we are shut down or saturated
         * and so reject the task.
         */
        int c = ctl.get();
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            if (! isRunning(recheck) && remove(command))
                reject(command);
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        else if (!addWorker(command, false))
            reject(command);
    }
```

其中addWorker()方法， 就是根据当前池的状态和给定的core和maximum值判断是否添加新的工作线程。 如果是这样,工作线程数量会相应调整，如果可能的话,一个新的工作线程创建并启动，将firstTask作为其运行的第一项任务。 如果池已停止此方法返回false，如果线程工厂在被访问时未能创建线程,也返回false。 如果线程创建失败，或者是由于线程工厂返回null，或者由于异常（通常是在调用Thread.start（ ）后的OOM））， 我们干净地回滚。

线程池的拒绝策略

- abortPolicy:不执行任务，直接抛出异常，提示线程池已满，线程池默认策略

```
public static class AbortPolicy implements RejectedExecutionHandler {
    /**
     * Creates an {@code AbortPolicy}.
     */
    public AbortPolicy() { }

    /**
     * Always throws RejectedExecutionException.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     * @throws RejectedExecutionException always
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        throw new RejectedExecutionException("Task " + r.toString() +
                                             " rejected from " +
                                             e.toString());
    }
}
```

- DiscardPolicy:不执行新任务，也不抛出异常，基本上为静默模式。

```
public static class DiscardPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code DiscardPolicy}.
         */
        public DiscardPolicy() { }

        /**
         * Does nothing, which has the effect of discarding task r.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        }
    }
```

- DisCardOldPolicy:将消息队列中的第一个任务替换为当前新进来的任务执行

```
    public static class DiscardOldestPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code DiscardOldestPolicy} for the given executor.
         */
        public DiscardOldestPolicy() { }

        /**
         * Obtains and ignores the next task that the executor
         * would otherwise execute, if one is immediately available,
         * and then retries execution of task r, unless the executor
         * is shut down, in which case task r is instead discarded.
         *
         * @param r the runnable task requested to be executed
```

```
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                e.getQueue().poll();
                e.execute(r);
            }
        }
    }
}
```

- CallerRunPolicy用于被拒绝任务的处理程序，它直接在 execute 方法的调用线程中运行被拒绝的任务；如果执行程序已关闭，则会丢弃该任务。

```
public static class CallerRunsPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code CallerRunsPolicy}.
         */
        public CallerRunsPolicy() { }

        /**
         * Executes task r in the caller's thread, unless the executor
         * has been shut down, in which case the task is discarded.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                r.run();
            }
        }
    }
```

锁分类

volitile（结合jvm第12章看）

可见性

当一条线程对volatile变量进行了修改操作时，其他线程能立即知道修改的值，即当读取一个volatile变量时总是返回最近一次写入的值

```
public class VolatileTest extends  Thread{
    private boolean isRunning = true;
    public boolean isRunning(){
        return isRunning;
    }
    public void setRunning(boolean isRunning){
```

```
        this.isRunning= isRunning;
    }
    public void run(){
        System.out.println("进入了run.............");
        while (isRunning){
            System.out.println("thred is running");
        }
        System.out.println("isRunning的值被修改为为false,线程将被停止了");
    }
    public static void main(String[] args) throws InterruptedException {
        VolatileTest volatileThread = new VolatileTest();
        volatileThread.start();
        Thread.sleep(1000);
        volatileThread.setRunning(false);    //停止线程
    }
}
```

在对isRunning进行setfalse的操作后，不会打印线程停止这段话。

因为Java内存模型(JMM)规定了所有的变量都存储在主内存中，主内存中的变量为共享变量，而每条线程都有自己的工作内存，线程的工作内存保存了从主内存拷贝的变量，所有对变量的操作都在自己的工作内存中进行，完成后再刷新到主内存中，代码主线程(线程main)虽然对isRunning的变量进行了修改且有刷新回主内存中（《深入理解java虚拟机》中关于主内存与工作内存的交互协议提到变量在工作内存中改变后必须将该变化同步回主内存），但volatileThread线程读的仍是自己工作内存的旧值导致出现多线程的可见性问题，解决办法就是给isRunning变量加上volatile关键字。

volatile内存语义

- 当线程对volatile变量进行写操作时，会将修改后的值刷新回主内存
- 当线程对volatile变量进行读操作时，会先将自己工作内存中的变量置为无效，之后再通过主内存拷贝新值到工作内存中使用。

原子性：

对于单个voatile变量其具有原子性(能保证long double类型的变量具有原子性)，但对于i ++ 这类复合操作其不具有原子性(见下面分析) 不保证源自性

有序性

volatile关键字禁止指令重排序有两层意思：

1）当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

2）在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行

volatile的应用场景

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

1）对变量的写操作不依赖于当前值（运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值）

2）该变量没有包含在具有其他变量的不变式中（变量不需要与其他的状态变量共同参与不变约束）

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

事实上，我的理解就是上面的2个条件需要保证操作是原子性操作，才能保证使用volatile关键字的程序在并发时能够正确执行。

实现机制

“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”

lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

2）它会强制将对缓存的修改操作立即写入主存；

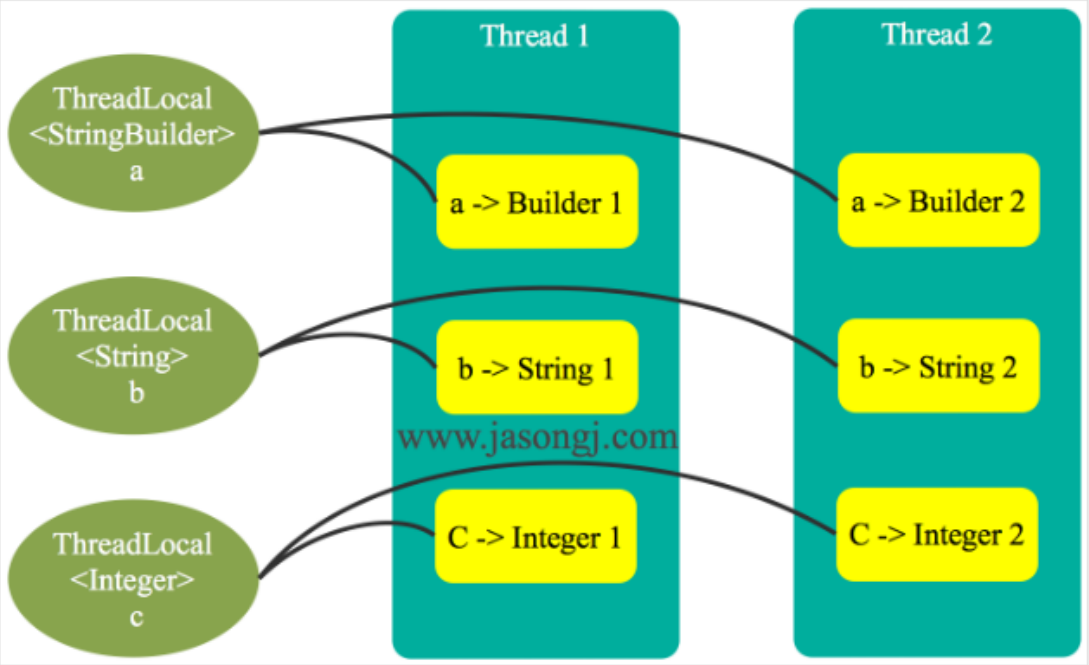
3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

ThreadLocal

产生线程安全问题的根源在于多线程之间的数据共享。如果没有数据共享，其实就没有多线程并发的安全问题。ThreadLocal就是通过实现避免多线程数据共享从而避免线程的并发安全问题。它为每一个线程都保留一个对象的副本，以ThreadLocalMap的形式保存(底层是一个Entry)，其中key是ThreadLocal类型的变量，value是要保存的副本变量，然后通过get方法来获取当前线程中的副本变量。

关键内部类 ThreadLocalMap

每一个线程都需要维护一个ThreadLocalMap，用来存储对应的key和value、



源码基本方法

构造器

```
public ThreadLocal() {
    }
```

get()

获取当前线程的副本变量值。根据线程找map，如果map存在，则根据threadLocal(key)去找value，也就是副本变量，如果变量存在，则返回。否则返回初始化的设置副本变量。

```
public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }
```

setInitialValue()

获取副本变量值，和当前线程，如果map存在，直接set值进去，如果不存在，创建map.

```
    private T setInitialValue() {
        T value = initialValue();
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
        return value;
    }
```

value = initialValue的初始值为null，所以如果是直接初始化的值，直接get，会报空指针异常的错误。解决办法是重写initialValue的方法。

```
    ThreadLocal<Long> longLocal = new ThreadLocal<Long>(){
        protected Long initialValue() {
            return Thread.currentThread().getId();
        };
    };
    ThreadLocal<String> stringLocal = new ThreadLocal<String>(){;
        protected String initialValue() {
            return Thread.currentThread().getName();
        };
    };
```

使用场景

最常见的ThreadLocal使用场景主要用来解决数据库连接，session管理等

```
private static ThreadLocal<Connection> connectionHolder
= new ThreadLocal<Connection>() {
public Connection initialValue() {
    return DriverManager.getConnection(DB_URL);
}
};

public static Connection getConnection() {
return connectionHolder.get();
}
```

内存泄漏问题(还未学)

ConcurrenHashMap