

上代码之前，要先补充一下线程池构造的核心几个点

1. 线程池里的核心线程数与最大线程数
2. 线程池里真正工作的线程类worker
3. 线程池里用来存取任务的队列
4. 线程中的任务类task

**另外一个比较简单的方法，推荐面试答这种，逻辑会比较好吧~**

class ThreadExcutor{

//创建

private volatile boolean RUNNING = true;

//所有任务都放队列中，让工作线程来消费

private static BlockingQueue<Runnable> queue = null;

**//池里的workers**

private final HashSet<Worker> workers = new HashSet<Worker>();

private final List<Thread> threadList = new ArrayList<Thread>();**//放到worker里的工作线程**

//工作线程数

int poolSize = 0;

//核心线程数（创建了多少个工作线程）

int coreSize = 0;

boolean shutdown = false;

public ThreadExcutor(int poolSize){

this.poolSize = poolSize;

queue = new LinkedBlockingQueue<Runnable>(poolSize);//排队等待队列

}

public void exec(Runnable runnable) {

if (runnable == null) throw new NullPointerException();

if(coreSize < poolSize){

addThread(runnable);

}else{

//System.out.println("offer" + runnable.toString() + " " + queue.size());

try {

queue.put(runnable);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public void addThread(Runnable runnable){

coreSize ++;

Worker worker = new Worker(runnable);

workers.add(worker);

Thread t = new Thread(worker);

threadList.add(t);

try {

t.start();

}catch (Exception e){

e.printStackTrace();

}

}

public void shutdown() {

RUNNING = false;

if(!workers.isEmpty()){

for (Worker worker : workers){

worker.interruptIfIdle();

}

}

shutdown = true;

Thread.currentThread().interrupt();

}

//这里留个位置放内部类Worker

}

然后定义一个内部类Worker，这个内部类Worker是用来执行每个任务的，在创建线程池后，往线程里添加任务，每个任务都是由Worker一个一个来启动的。

/\*\*

\* 工作线程

\*/

class Worker implements Runnable{

public Worker(Runnable runnable){

queue.offer(runnable);

}

@Override

public void run() {

while (true && RUNNING){

if(shutdown == true){

Thread.interrupted();

}

Runnable task = null;

try {

task = getTask();

task.run();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public Runnable getTask() throws InterruptedException {

return queue.take();

}

public void interruptIfIdle() {

for (Thread thread :threadList) {

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

thread.interrupt();

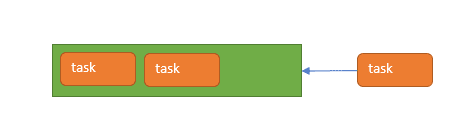
}

}

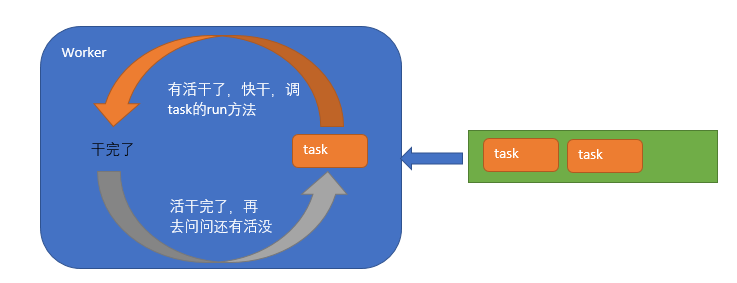
}

首先注意的一点，这个Worker是个内部类，是在线程池内声明的。

**exec方法**

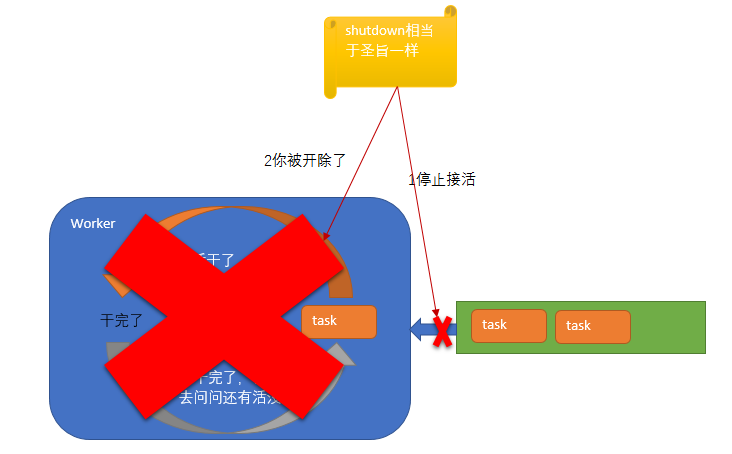


**Worker怎么工作**



这个工作线程实例化的时候就**先加入一个任务到队列中，也就是说在实例化这个工作线程时，这个工作线程也是一个任务被加入到线程池中。**然后就是run方法，这个run方法是线程调start方法生成的线程，而Worker调的run方法并没有生成新的线程。就是一个循环，一直在不停的从队列中取任务，然后执行。可以看到，取队列的方法是take()，这个方法意思如果队列为空了，取不到数据时就阻塞队列。

**然后看shutdown()**



你每天辛勤的劳动着，突然接收到上面的命令，说活暂时不要接了，先停下来，当你还没搞清楚状况时，接着你的领导又把你开除了，说公司要倒了，你先下岗吧，一会我也得下岗了。这就是shutdown做的事，shutdown必须是主线程才能停止工作线程。  
shutdown方法并不是用线程那种强制停止的搞法，而是先用一个标识符告诉工作线程，不要再接任务了。然后通知工作线程，你可以interrupt()了，当所有的线程停止后记得要把主线程也停掉，这样，一个简单任务的线程池就完成了。  
让我们来测试一下：

/\*\*

\* Created by wxwall on 2017/6/7.

\*/

public class TheadBlockedQ {

public static void main(String[] args) throws InterruptedException {

ThreadExcutor excutor = new ThreadExcutor(3);

for (int i = 0; i < 10; i++) {

excutor.exec(new Runnable() {

@Override

public void run() {

System.out.println("线程 " + Thread.currentThread().getName() + " 在帮我干活");

}

});

}

excutor.shutdown();

}

}

输出结果为：

线程 Thread-0 在帮我干活

线程 Thread-2 在帮我干活

线程 Thread-1 在帮我干活

线程 Thread-0 在帮我干活

线程 Thread-2 在帮我干活

线程 Thread-2 在帮我干活

线程 Thread-1 在帮我干活

线程 Thread-0 在帮我干活

Thread-0 interrupt

Thread-1 interrupt

Thread-2 interrupt

Thread-0 interrupt

Thread-1 interrupt

Thread-2 interrupt

Thread-0 interrupt

Thread-1 interrupt

Thread-2 interrupt

这当然是最简单实现，JDK的实现比这强大的多，而且还具备当工作线程处理不过来时，可以产生新的线程来处理任务，这个数量不能超过原先定义的最大线程数，而在本例中都没实现这些功能。  
我相信当想了解一个模块的功能时，如果一开始就了解其中最核心的点，然后向外慢慢扩展，那么学习这个模块时一定能省下不少时间，而且理解将很深刻。希望这个简单线程池实现能让你有所领悟，以更加简单的方式了解线程池，了解了线程池，对于其他池化技术，原理都是相通的。

另外个严谨点的方法：难一些。

**一，线程池的基本要素**

**线程池一般需要一个线程管理类： ThreadPoolManager，其作用有：**

  1）提供创建一定数量的线程的方法。主线程调用该方法，从而创建线程。创建的线程执行自己的例程，线程的例程阻塞在任务抓取上。

  2）提供对任务队列的操作的方法。主线程调用初始化任务队列的方法，然后在有任务的时候，调用提供的任务添加方法，将任务添入等待队列。当主线程调用任务的添加方法时，会触发等待的线程，从而使得阻塞的线程被唤醒，其抓取任务，并执行任务。

**线程池需要一个任务队列： List<Task>，其作用有：**

  提供任务的增删方法。而且该任务队列需要进行排他处理，防止多个工作线程对该任务队列进行同时的抓取操作或者主线程的加入与工作线程的抓取的并发操作。

**线程池需要一个类似信号量的通知机制：wait -notify：**

  工作线程调用wait在任务抓取上进行线程阻塞。主线程添加任务后，调用notify触发阻塞的线程。

**线程池需要一个线程类：WorkThread，其作用有：**

  提供线程的例程。创建线程WorkThread后，需要抓取任务，并执行任务。这是线程的例程。

**线程池需要一个任务类：Task，其作用有：**

  提供线程抓取并执行的任务目标。

**二，基础知识和策略选择**

  1，线程同步和通知机制

  Object类的wait()方法、notify()方法、notifyAll()方法。

  Object类的对象都有两个池： 监视池（monitor，或者称锁池）和等待池（等待队列）。

  监视池： 如果一个线程想要调用一个对象的synchronized的方法，或者对对象的操作被synchronized块所包含，则线程必须获得该对象的对象锁才能执行这个方法或者代码块。如果获取不到，则被加入到该对象的监视池中，等待锁资源。

  等待锁：如果一个线程调用对象的wait()方法，则线程会放弃持有的该对象的锁资源（因为wait()必须在获取到对象锁之后才能执行），并进入该对象的等待池。如果另外一个线程调用了相同对象的notify()/notifyAll()方法，就会唤醒在该对象的等待池中的一个或者所有线程。一旦被唤醒，线程就进入该对象的锁池，进行锁资源竞争，注意不是立即执行，也不能保证唤醒的是哪个线程。

  另外由于**wait()方法可能发生中断和虚假唤醒**，以及时间timeout（带时间的wait方法），因此，**应该在while循环中使用。**

//在同步块或者synchronized方法中

synchronized (obj) {

//在循环中

while (<condition does not hold>)

obj.wait();

... // Perform action appropriate to condition

}

注： 我个人理解，这里的“虚假唤醒”的具体含义，应该包含如下这种情况： **一个被唤醒的线程发现其需要获取的条件已经被别的被唤醒的线程导致的不再成立。** 比如在线程池的实现中，任务队列的任务： 当主线程向其中添加两个任务的时候，会唤醒第一个线程去抓取任务执行，而这个线程不会仅仅抓取一个，它执行完第一个立马抓取第二个。这个时候，第二个被唤醒的线程将无法抓取到，那么它不应该在被唤醒并获取锁后，对任务队列进行任务的获取并删除操作，即应该在获取锁后检查队列是否为空。由于获取锁后是继续wait()后的代码执行，因此必须使用while进行判断，而不能是if()判断，因为获取锁后，if的条件判断不会被再执行，而**while语句则必须再进行循环判断看是否跳出循环**。详细参见代码。

  synchronized用来修饰一个非静态方法，表示执行这个方法，必须获取该方法所属对象的锁；  synchronized用来修饰静态方法，表示要执行该方法必须获取该类的类锁；synchronized修饰代码块synchronized(obj) { //code.... }表示执行该代码块必须获取obj这个对象的对象锁。这样做的目的是减小锁的粒度，保证当不同块所需的锁不冲突时不用对整个对象加锁。利用零长度的byte数组对象做obj非常经济。

  atomic action（原子操作）：

  在Java中，以下两点操作是原子操作。

  1)，对引用变量和除了long和double之外的原始数据类型变量进行读写。

  2)，对所有声明为volatile的变量（包括long和double）的读写。  
  另外：在java.util.concurrent和java.util.concurrent.atomic包中提供了一些不依赖于同步机制的线程安全的类和方法。

  2，线程和任务

  线程：

  线程可以赋予名字、优先级、标识是否是守护线程等。

  在Java中有两类线程：**User Thread(用户线程)、Daemon Thread(守护线程)。**

  守护线程和用户线程唯一的区别就是守护线程会随着用户线程的（被守护的）结束而结束。

  使用方法：

  setDaemon(true);

  这里有几点需要注意：   
    (1) setDaemon(true)必须在start()之前设置，否则会抛出一个IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。   
    (2) 在Daemon线程中产生的新线程也是Daemon的。    
    (3) 不要认为所有的应用都可以分配给Daemon来进行服务，比如读写操作或者计算逻辑。有可能在守护线程跑了一半的时候主线程跑完了，然后守护线程就也会停止了。

  Java中的线程可以通过两种方式进行创建：继承Thread类，或者实现Runnable接口。继承Thread类应该重写其run()方法。实现Runnable类应该实现run()方法，并且将其作为Thread(Runnable target)的参数，即Thread的执行目标，创建线程，并开启其例程。例如：

  1）继承Thread:

class PrimeThread extends Thread {

long minPrime;

PrimeThread(long minPrime) {

this.minPrime = minPrime;

}

public void run() {

// compute primes larger than minPrime

 . . .

}

}

PrimeThread p = new PrimeThread(143);

//开启线程的例程

 p.start();

  2)实现Runnable：

class PrimeRun implements Runnable {

long minPrime;

PrimeRun(long minPrime) {

this.minPrime = minPrime;

}

public void run() {

// compute primes larger than minPrime

 . . .

}

}

PrimeRun p = new PrimeRun(143);

//作为Thread的参数，并start例程

new Thread(p).start();

  任务类：

**任务类应该实现Runnable接口并提供run()方法。因为其是打算通过线程执行其实例的**。Runnable为希望在活动时（线程被开启）执行代码的对象提供了一个公共协议。其为非Thread类的子类的类提供了一种激活方式，即无需继承Thread类便可以进行激活。即通过实例化某个Thread类的子类，并通过将此实现Runnable接口的类的实例作为线程的目标传递给线程，就可运行Runnable类而无需继承Thread。这在大多数情况下，如果只想重写run() 方法，而不重写其他Thread 方法，就应使用 Runnable 接口。因为除非程序员打算修改或增强类的基本行为，否则不应为该类创建子类。

  因此任务类应该是实现Runnable接口的抽象类并提供run的基本实现。

 3，**任务队列的数据结构选择**

  ArrayList和LinkedList ：

  ArrayList和LinkedList是两个集合类，可以用于存储一系列的对象引用。ArrayList是基于动态数组的实现，而LinkedList是基于链表的。

  对于随机访问操作（获取第n个元素的值，或者查找指定值的某个元素），ArrayList比LinkedList占优势，因为LinkedList需要指针的移动。

  对于ArrayList，进行随机方法，其每次访问都是一个常数，而对于ArrayList则是于ArrayList的长度（要访问元素所在索引）成比例的。

  对于增删操作，由于ArrayList需要数据的移动，因而LinkedList占优势。对于ArrayList与插入的索引成比例，而LinkedList则是固定的。

   因而，对于需要开始、中间插入删除的并且是顺序访问，而不是随机访问的时候，可以采用LinkedList；不会进行开始中间插入删除，

而是末尾操作，并且多是随机访问则应该使用ArrayList。整体来说，ArrayList查询效率高，增删操作效率低；LinkedList查询效率低，增删操作效率较高。

综上所述，我们设计的线程池要考虑**线程和任务类**，以及对于**任务队列访问的排他处理和wait-notify机制的细节**，同时要考虑**任务队列的存储数据结构的选择**。

**三，实现**

  1）线程池管理类

package poolmanager;  
  
 /\*\*  
 \* ThreadPoolManager 线程池管理类  
 \* @version 1.0.0  
 \* @see 线程池定义式样书  
 \* @date    2011/09/30 IterZebra  
 \*/  
//import java.util.Collections;  
import java.util.LinkedList;  
import java.util.List;  
  
import systemconfig.SystemConfig;  
  
public class ThreadPoolManager extends ThreadGroup {  
  
    /\*\*线程池是否开启标识\*/  
    int flagThreadPoolValid = 0;  
  
    /\*\*线程池中线程的数量，从系统配置类中获取\*/  
   **int threadSize = SystemConfig.getThreadDefaultCount();**  
    /\*\*任务队列\*/  
     /\*\*  
     Java 2 SE Documentation.  
     Returns a synchronized (thread-safe) list backed by the specified list.  
     In order to guarantee serial access, it is critical that all access to  
     the backing list is accomplished through the returned list.  
     根据指定的参数list，返回一个线程安全的同步的list。  
     为了保证有序访问，所有的对于参数list的访问都应该通过返回的这个list去进行。  
     It is imperative that the user manually synchronize  
     on the returned list when iterating over it.  
     如果进行迭代访问的时候，对于返回的队列应用程序本身进行同步处理是必要的。  
     注：即使用迭代器访问应该使用synchronized(){}修饰代码块，使得其对list的访问进行同步；  
     而如果是采用返回的list的add等方法，则应用程序本身不需要进行同步处理。  
  
   **另外由于本类中对ThreadPoolManager的访问方法都进行了同步操作，  
     因此对本List的同步不是必要的。**  
     当然，由于加锁顺序一致性，使用对List的同步，也不会导致线程死锁。  
    List<Task> TaskList= Collections.synchronizedList(new LinkedList<Task>());  
  
    \*/  
**List<Task> TaskList= new LinkedList<Task>();**  
  
    /\*\*线程池管理类构造方法\*/  
    public ThreadPoolManager(String threadpoolname) {  
  
         //ThreadGroup的名称  
        super(threadpoolname);  
  
        //继承自父类的方法，设置是否是守护线程  
        setDaemon(true);  
  
    }  
  
    /\*\*  
     \* @brief 开启线程池  
     \* @param null  
     \* @return void  
     \*/  
    public synchronized void threadPoolStart(){  
  
        if(threadSize ==0 || flagThreadPoolValid !=0){  
  
            try {  
                throw new Exception();  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
  
            return ;  
        }  
  
        if( TaskList ==null ){  
  
            try {  
                throw new Exception();  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
  
            return;  
        }  
  
        // 创建并开启线程例程  
        for (int i = 0;i<threadSize;i++){  
  
             new WorkThread(i).start();  
  
        }  
  
        flagThreadPoolValid = 1;  
  
    }  
  
    /\*\*  
     \* @brief 关闭线程池  
     \* @param null  
     \* @return void  
     \*/  
    public synchronized void threadPoolEnd(){  
  
        if(threadSize ==0 || flagThreadPoolValid !=1){  
  
            try {  
                throw new Exception();  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
  
            return;  
  
        }  
        if(TaskList ==null ){  
  
            try {  
                throw new Exception();  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
  
            return;  
        }  
  
        TaskList.clear();  
        threadSize = 0;  
        flagThreadPoolValid = 0;  
  
        // 继承自父类，中断ThreadGroup中添加的所有线程  
        interrupt();  
  
    }  
  
    /\*\*  
     \* @brief 向线程池管理类的**任务队列中添加任务**  
     \* @param newTask Task任务类  
     \* @return void  
     \*/  
   **public synchronized void addTask(Task newTask){**  
        if(TaskList == null ){  
  
            try {  
                throw new Exception();  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
  
            return;  
        }  
  
        TaskList.add(newTask);  
  
        //唤醒一个正在getTask()方法中等待任务的工作线程  
        notify();  
  
    }  
  
    /\*\*  
     \* @brief 获取线程池中任务队列中的任务  
     \* @param null  
     \* @return Task类  
     \*/  
   **public synchronized Task getTask()**{  
  
        if(TaskList ==null ){  
  
            try {  
                throw new Exception();  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
  
            return null;  
        }  
        /\*\*  
         Java 2 SE Documentation.  
          As in the one argument version,  
          interrupts and spurious wakeups are possible,  
          and this method should always be used in a loop.  
          正如具有一个参数的函数版本，  
          由于wait()函数调用过程中可能发生中断、虚假唤醒，  
          因此这个方法必须在检测循环中使用。  
          另外，在这里，如果不使用while循环，存在如下一种问题：  
          抛出地址越界异常。  
          例如：  
          主线程添加了一个任务，这样会唤醒一个线程。在这个线程被唤醒执行。  
          此时主线程添加另外一个任务，又唤醒一个线程。  
          第一个线程其抓取任务后，继续抓取下一个任务执行，也就是抓取添加的第二个任务。  
          这样队列中已经不存在任务。第二个被唤醒的线程必然在执行remove(0)的时候抛出上述异常。  
  
          而且也不能仅仅是if判断，因为唤醒获取锁后，  
          会直接执行wait()后的代码，只有while可以达到进行判断的目的。  
        \*/  
        while(TaskList.size() == 0 ){  
  
            try {  
                wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
  
        }  
  
       **return  TaskList.remove(0);**  
  
    }  
  
    /\*\*  
     \* @brief 内部类，**工作线程类**     \*/  
 **private class WorkThread extends Thread{**  
  
        public WorkThread(int threadID) {  
  
            //父类构造方法,将线程加入到ThreadGroup中  
            super(ThreadPoolManager.this,""+threadID);  
        }  
  
        /\*\*  
          \* @brief 重写父类Thread的run方法  
          \* @param null  
          \* @return void  
          \*/  
        public void run(){  
  
            //isInterrupted()方法继承自Thread类，判断线程是否被中断  
            while(! isInterrupted()){  
  
                Task runTask = getTask();  
  
                //getTask()返回null或者线程执行getTask()时被中断  
                if(runTask == null)    break ;  
  
                runTask.run();  
  
            }  
  
        }  
  
    }  
}

  2）任务类

package poolmanager;  
  
public abstract class Task implements Runnable {  
  
    public void run() {  
    }  
  
}

//系统配置类

package systemconfig;  
  
public class SystemConfig {  
      
    static final int THREAD\_DEFAULT\_COUNT = 10;  
      
    public static int getThreadDefaultCount() {  
        return THREAD\_DEFAULT\_COUNT;  
    }  
      
  
}

  3）测试

package poolmanager.test;  
  
import poolmanager.Task;  
  
public class TestTask extends Task {  
  
    private   int i ;  
  
    public TestTask(int i) {  
        this.i = i;  
    }  
  
    public void run() {  
        System.out.println("Hello world");  
        System.out.println(i+"");  
    }  
  
}

package poolmanager.test;  
  
import poolmanager.Task;  
import poolmanager.ThreadPoolManager;  
  
public class PoolManagerTest {  
  
    /\*\*  
     \* @param args  
     \*/  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        ThreadPoolManager threadpoolmanager = new ThreadPoolManager("SimpleThreadPoll");  
        threadpoolmanager.threadPoolStart();  
        Task newTask = new TestTask(1);  
        threadpoolmanager.addTask(newTask);  
         newTask = new TestTask(2);  
        threadpoolmanager.addTask(newTask);  
         newTask = new TestTask(3);  
        threadpoolmanager.addTask(newTask);  
         newTask = new TestTask(4);  
        threadpoolmanager.addTask(newTask);  
         newTask = new TestTask(5);  
        threadpoolmanager.addTask(newTask);  
         newTask = new TestTask(6);  
        threadpoolmanager.addTask(newTask);  
        //threadpoolmanager.threadPoolEnd();  
  
    }  
}