带你一次搞明白 Java 多线程（X）

6 线程管理

6.4 线程池

6.4.5 ThreadFactory

线程池中的线程从哪儿来的？答案就是 ThreadFactory.

ThreadFactory 是一个接口,只有一个用来创建线程的方法:

Thread newThread(Runnable r);

当线程池中需要创建线程时就会调用该方法

```
package com.wkcto.threadpool;

import java.util.Random;
import java.util.concurrent.*;

/**
 * 自定义线程工厂
 */
public class Test04 {
    public static void main(String[] args) throws InterruptedException {
        //定义任务
        Runnable r = new Runnable() {
            @Override
            public void run() {
                int num = new Random().nextInt(10);
                System.out.println(Thread.currentThread().getId() + "--" +
System.currentTimeMillis() + "开始睡眠:" + num + "秒");
```

```
                try {
                    TimeUnit.SECONDS.sleep(num);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        //创建线程池，使用自定义线程工厂，采用默认的拒绝策略是抛出异常
        ExecutorService executorService = new ThreadPoolExecutor(5, 5, 0,
TimeUnit.SECONDS, new SynchronousQueue<>(), new ThreadFactory() {
            @Override
            public Thread newThread(Runnable r) {
                //根据参数 r 接收的任务,创建一个线程
                Thread t = new Thread( r );
                t.setDaemon(true);  //设置为守护线程，当主线程运行结束,线程池中的
线程会自动退出
                System.out.println("创建了线程: " + t);
                return t ;
            }
        });

        //提交 5 个任务，当给当前线程池提交的任务超过 5 个时,线程池默认抛出异常
        for (int i = 0; i < 5; i++) {
            executorService.submit(r);
        }

        //主线程睡眠
        Thread.sleep(10000);
        //主线程睡眠超时，主线程结束，线程池中的线程会自动退出
    }
}
```

6.4.6 监控线程池

ThreadPoolExecutor 提供了一组方法用于监控线程池

int getActiveCount()　获得线程池中当前活动线程的数量

long getCompletedTaskCount()　返回线程池完成任务的数量

int getCorePoolSize()　线程池中核心线程的数量

int getLargestPoolSize()　返回线程池曾经达到的线程的最大数

int getMaximumPoolSize()　返回线程池的最大容量

int getPoolSize()　当前线程池的大小

BlockingQueue<Runnable> getQueue()　返回阻塞队列

long getTaskCount()　返回线程池收到的任务总数

```
package com.wkcto.threadpool;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * 监控线程池
 */
public class Test05 {
    public static void main(String[] args) throws InterruptedException {
        //先定义任务
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getId() + " 编号 的线程开始执
```

```
行: " + System.currentTimeMillis());
                    try {
                        Thread.sleep(10000);      //线程睡眠 20 秒,模拟任务执行时长
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            };

            //定义线程池
            ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(2, 5, 0, TimeUnit.SECONDS, new ArrayBlockingQueue<>(5), Executors.defaultThreadFactory(), new ThreadPoolExecutor.DiscardPolicy());

            //向线程池提交 30 个任务
            for (int i = 0; i < 30; i++) {
                poolExecutor.submit(r);
                System.out.println("当前线程池核心线程数量:" + poolExecutor.getCorePoolSize() + ",最大线程数:" + poolExecutor.getMaximumPoolSize() + ",当前线程池大小:" + poolExecutor.getPoolSize() + ",活动线程数量:" + poolExecutor.getActiveCount()+ ",收到任务数量:" + poolExecutor.getTaskCount() + ",完成任务数:" + poolExecutor.getCompletedTaskCount() + ",等待任务数:" + poolExecutor.getQueue().size()) ;
                TimeUnit.MILLISECONDS.sleep(500);
            }

            System.out.println("-----------------------------------------");
            while ( poolExecutor.getActiveCount() >= 0 ){
                System.out.println("当前线程池核心线程数量:" + poolExecutor.getCorePoolSize() + ",最大线程数:" + poolExecutor.getMaximumPoolSize() + ",当
```

```
前线程池大小:" + poolExecutor.getPoolSize() + ", 活动线程数量:" +
poolExecutor.getActiveCount()+ ",收到任务数量:" + poolExecutor.getTaskCount() + ",完成任务
数:" + poolExecutor.getCompletedTaskCount() + ", 等待任务数:" +
poolExecutor.getQueue().size()) ;
                Thread.sleep(1000);
            }
        }

}
```

6.4.7 扩展线程池

有时需要对线程池进行扩展,如在监控每个任务的开始和结束时间,或者自定义一些其他增强的功能.

ThreadPoolExecutor 线程池提供了两个方法:

protected void afterExecute(Runnable r, Throwable t)

protected void beforeExecute(Thread t, Runnable r)

在线程池执行某个任务前会调用 beforeExecute()方法,在任务结束后(任务异常退出)会执行 afterExecute()方法

查看 ThreadPoolExecutor 源码,在该类中定义了一个内部类 Worker,ThreadPoolExecutor 线程池中的工作线程就是 Worker 类的实例,Worker 实例在执行时会调用 beforeExecute()与 afterExecute()方法

```
package com.wkcto.threadpool;

import com.wkcto.produerstack.MyStack;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * 扩展线程池
 */
public class Test06 {

    //定义任务类
    private static class  MyTask implements  Runnable{
        String name;

        public MyTask(String name) {
            this.name = name;
        }

        @Override
        public void run() {

            System.out.println(name + "任务正在被线程 " + Thread.currentThread().getId() +
" 执行");

            try {

                Thread.sleep(1000);       //模拟任务执行时长

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {

        //定义扩展线程池，可以定义线程池类继承 ThreadPoolExecutor,在子类中重写
beforeExecute()/afterExecute()方法

        //也可以直接使用 ThreadPoolExecutor 的内部类
```

```
        ExecutorService executorService = new ThreadPoolExecutor(5, 5, 0,
TimeUnit.SECONDS, new LinkedBlockingQueue<>() ){

            //在内部类中重写任务开始方法

            @Override
            protected void beforeExecute(Thread t, Runnable r) {

                System.out.println(t.getId() + "线程准备执行任务: " + ((MyTask)r).name);

            }


            @Override
            protected void afterExecute(Runnable r, Throwable t) {

                System.out.println( ((MyTask)r).name + "任务执行完毕");

            }


            @Override
            protected void terminated() {

                System.out.println("线程池退出");

            }
        };


        //向线程池中添加任务

        for (int i = 0; i < 5; i++) {
            MyTask task = new MyTask("task-" + i);
            executorService.execute(task);
        }


        //关闭线程池

        executorService.shutdown();        //关闭线程池仅仅是说线程池不再接收新的任
务，线程池中已接收的任务正常执行完毕

    }
}
```

6.4.8 优化线程池大小

线程池大小对系统性能是有一定影响的,过大或者过小都会无法发挥最优的系统性能，线程池大小不需要非常精确,只要避免极大或者极小的情况即可，一般来说,线程池大小需要考虑 CPU 数量,内存大小等因素．在<Java Concurrency in Practice>书中给出一个估算线程池大小的公式:

线程池大小 = CPU 的数量 * 目标 CPU 的使用率*(1 + 等待时间与计算时间的比)

6.4.9 线程池死锁

如果在线程池中执行的 任务 A 在执行过程中又向线程池提交了任务 B，任务 B 添加到了线程池的等待队列中，如果任务 A 的结束需要等待任务 B 的执行结果．就有可能会出现这种情况：线程池中所有的工作线程都处于等待任务处理结果,而这些任务在阻塞队列中等待执行，线程池中没有可以对阻塞队列中的任务进行处理的线程,这种等待会一直持续下去,从而造成死锁.

适合给线程池提交相互独立的任务,而不是彼此依赖的任务．对于彼此依赖的任务,可以考虑分别提交给不同的线程池来执行.

6.4.10 线程池中的异常处理

在使用 ThreadPoolExecutor 进行 submit 提交任务时,有的任务抛出了异常,但是线程池并没有进行提示,即线程池把任务中的异常给吃掉了,可以把 submit 提交改为 execute 执行,也可以对 ThreadPoolExecutor 线程池进行扩展.对提交的任务进行包装:

```
package com.wkcto.threadpool;

import java.util.concurrent.*;

/**
 * 自定义线程池类,对 ThreadPoolExecutor 进行扩展
 */
public class Test08 {
    //自定义线程池类
    private static class  TraceThreadPollExecutor extends  ThreadPoolExecutor{
        public  TraceThreadPollExecutor(int  corePoolSize,  int  maximumPoolSize,  long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {
            super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
        }
        //定义方法,对执行的任务进行包装,接收两个参数,第一个参数接收要执行的任务,第二个参数是一个 Exception 异常
        public Runnable wrap( Runnable task, Exception exception){
            return  new Runnable() {
                @Override
                public void run() {
                    try {
                        task.run();
                    }catch (Exception e ){
                        exception.printStackTrace();
                        throw  e;
                    }
                }
```

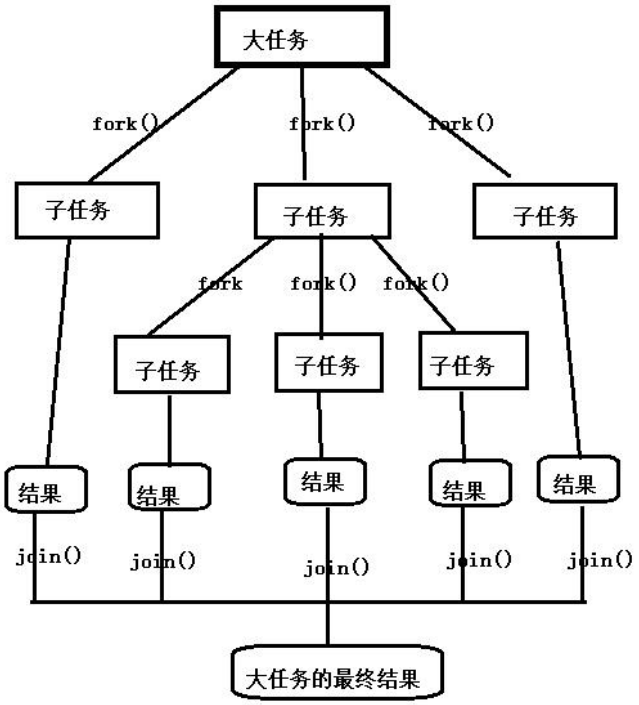
```
                };
            }

            //重写 submit 方法
            @Override
            public Future<?> submit(Runnable task) {
                return super.submit(wrap(task, new Exception("客户跟踪异常")));
            }

            @Override
            public void execute(Runnable command) {
                super.execute(wrap(command, new Exception("客户跟踪异常")));
            }
        }

        //定义类实现 Runnable 接口,用于计算两个数相除
        private static class    DivideTask implements    Runnable{
            private    int x;
            private    int y;

            public DivideTask(int x, int y) {
                this.x = x;
                this.y = y;
            }

            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + "计算:" + x + " / " + y + " =
" + (x/y));
            }
        }
        public static void main(String[] args) {
            //创建线程池
//              ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(0, Integer.MAX_VALUE,
0, TimeUnit.SECONDS, new SynchronousQueue<>());
```

```
        //使用自定义的线程池
        ThreadPoolExecutor poolExecutor = new TraceThreadPollExecutor(0, Integer.MAX_VALUE, 0, TimeUnit.SECONDS, new SynchronousQueue<>());

        //向线程池中添加计算两个数相除的任务
        for (int i = 0; i < 5; i++) {
            poolExecutor.submit(new DivideTask(10, i));
//            poolExecutor.execute(new DivideTask(10, i));
        }

    }
}
```

6.4.11 ForkJoinPool 线程池

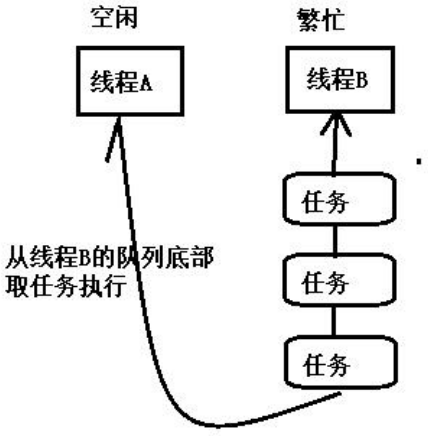
“分而治之”是一个有效的处理大数据的方法,著名的 MapReduce 就是采用这种分而治之的思路. 简单点说,如果要处理的 1000 个数据,但是我们不具备处理 1000 个数据的能力,可以只处理 10 个数据, 可以把这 1000 个数据分阶段处理 100 次,每次处理 10 个,把 100 次的处理结果进行合成,形成最后这 1000 个数据的处理结果.

把一个大任务调用 fork()方法分解为若干小的任务,把小任务的处理结果进行 join()合并为大任务的结果



系统对 ForkJoinPool 线程池进行了优化,提交的任务数量与线程的数量不一定是一对一关系.在多数情况下,一个物理线程实际上需要处理多个逻辑任务.

线程A把自己的任务执行完毕
线程B的任务队列中还有若干的任务等待执行
线程A会从线程B的等待队列中取任务帮助线程B完成
线程A在帮助线程B执行任务时,总是从线程B的等待队列底部开始取任务,



ForkJoinPool 线程池中最常用 的方法是:

<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)　向线程池提交一个 ForkJoinTask 任务. ForkJoinTask 任务支持 fork()分解与 join()等待

的任务．ForkJoinTask 有两个重要的子类:RecursiveAction 和 RecursiveTask ,它们的区别在于 RecursiveAction 任务没有返回值，RecursiveTask 任务可以带有返回值

```
package com.wkcto.threadpool;

import java.util.ArrayList;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.concurrent.RecursiveTask;

/**
 * 演示 ForkJoinPool 线程池的使用
 *   使用该线程池模拟数列求和
 */
public class Test09 {
    //计算数列的和，需要返回结果,可以定义任务继承 RecursiveTask
    private static class CountTask extends RecursiveTask<Long>{
        private static final int THRESHOLD = 10000;        //定义数据规模的阈值,允许计算10000 个数内的和,超过该阈值的数列就需要分解
        private static final int TASKNUM = 100;        //定义每次把大任务分解为 100 个小任务
        private long start;        //计算数列的起始值
        private long end;          //计算数列的结束值

        public CountTask(long start, long end) {
            this.start = start;
            this.end = end;
        }
```

```
        //重写 RecursiveTask 类的 compute()方法,计算数列的结果
        @Override
        protected Long compute() {
            long sum = 0 ;          //保存计算的结果
            //判断任务是否需要继续分解,如果当前数列 end 与 start 范围的数超过阈值 THRESHOLD,就需要继续分解
            if ( end - start < THRESHOLD){
                //小于阈值可以直接计算
                for (long i = start ; i <= end; i++){
                    sum += i;
                }
            }else {         //数列范围超过阈值,需要继续分解
                //约定每次分解成 100 个小任务,计算每个任务的计算量
                long step = (start + end ) / TASKNUM;
                //start = 0 , end = 200000, step = 2000,  如果计算[0,200000]范围内数列的和,  把该范围的数列分解为 100 个小任务,每个任务计算 2000 个数即可
                //注意,如果任务划分的层次很深,即 THRESHOLD 阈值太小,每个任务的计算量很小,层次划分就会很深,可能出现两种情况:一是系统内的线程数量会越积越多,导致性能下降严重;   二是分解次数过多,方法调用过多可能会导致栈溢出
                //创建一个存储任务的集合
                ArrayList<CountTask> subTaskList = new ArrayList<>();
                long pos = start;           //每个任务的起始位置
                for (int i = 0; i < TASKNUM; i++) {
                    long lastOne = pos + step;          //每个任务的结束位置
                    //调整最后一个任务的结束位置
                    if ( lastOne > end ){
```

```
                lastOne = end;
            }
            //创建子任务
            CountTask task = new CountTask(pos, lastOne);
            //把任务添加到集合中
            subTaskList.add(task);
            //调用 for()提交子任务
            task.fork();
            //调整下个任务的起始位置
            pos += step + 1;
        }

        //等待所有的子任务结束后,合并计算结果
        for (CountTask task : subTaskList) {
            sum += task.join();          //join()会一直等待子任务执行完毕返回执行结果
        }
    }

    return sum;
  }
}

public static void main(String[] args) {
    //创建 ForkJoinPool 线程池
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    //创建一个大的任务
    CountTask task = new CountTask(0L, 200000L);
    //把大任务提交给线程池
    ForkJoinTask<Long> result = forkJoinPool.submit(task);
    try {
```

```
            Long res = result.get();        //调用任务的 get()方法返回结果

            System.out.println("计算数列结果为:" + res);

        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }


        //验证
        long s = 0L;
        for (long i = 0; i <= 200000 ; i++) {
            s += i;
        }
        System.out.println(s);
    }
}
```