

# JAVA并发编程实践

# 目录

第七章、应用线程池 .....	3
线程池的组成 .....	3
使用线程池经常遇到的问题 .....	4
□ 线程饥饿死锁 .....	4
□ 耗时操作 .....	5
定制线程池的大小 .....	5
配置 <b>ThreadPoolExecutor</b> .....	6
□ 线程的创建与销毁 .....	6
□ 管理队列任务 .....	7
阻塞队列类别 .....	7
□ 饱和策略 .....	8
□ 线程工厂 .....	9
<input checked="" type="checkbox"/> 定制的线程工厂 .....	9
<input checked="" type="checkbox"/> 自定义的线程基类 .....	10
<input checked="" type="checkbox"/> 构造后再定制 <i>ThreadPoolExecutor</i> .....	11
扩展 <b>ThreadPoolExecutor</b> .....	11
□ 给线程池加入统计信息功能 .....	12
并行递归算法 .....	13
<input checked="" type="checkbox"/> 顺序递归转化为并行递归 .....	13
<input checked="" type="checkbox"/> 走迷宫游戏 .....	14
小结 .....	17

# 第七章、应用线程池

本节主要关注在配置和调整线程池时用的高级选项，讲述了任务执行框架的过程中需要注意的危险。线程复用原理如下：

每一个 Thread 的类都有一个 start 方法。当调用 start 启动线程时 Java 虚拟机会调用该类的 run方法。那么该类的 run() 方法中就是调用了 Runnable 对象的 run() 方法。我们可以继承重写Thread 类，在其 start 方法中添加不断循环调用传递过来的 Runnable 对象。这就是线程池的实现原理。循环方法中不断获取 Runnable 是用 Queue 实现的，在获取下一个 Runnable 之前可以是阻塞的。



## 线程池的组成

Java 中的线程池是通过 Executor 框架实现的，该框架中用到了 Executor, Executors, ExecutorService, ThreadPoolExecutor , Callable 和 Future、FutureTask 这几个类。一般的线程池主要分为以下 4 个组成部分：

- 线程池管理器：用于创建并管理线程池
- 工作线程：线程池中的线程
- 任务接口：每个任务必须实现的接口，用于工作线程调度其运行
- 任务队列：用于存放待处理的任务，提供一种缓冲机制

## 使用线程池经常遇到的问题

由于任务和任务之间存在差异，所以Executor框架的任务执行策略需要根据需求来改变。比如：独立性任务、依赖性任务、线程封闭的任务、响应时间敏感的任务、ThreadLocal任务。

标准的Executor的实现是：在需求不高时回收空闲的线程，在需求增加时添加新的线程，如果任务抛出了异常，就会用一个全新的工作者线程取代出错的那个。只有是线程封闭任务时在池的某线程中使用ThreadLocal才有意义。

其它情况下，在线程池中，不应该用ThreadLocal传递任务间的数值。

当任务都是同类的，独立的时候，线程池才会有最佳的工作表现，如果将耗时的与短期的任务混合在一起，就需要扩大池。如果存在依赖就需要池无限大，否则有产生死锁的风险。

### □ 线程饥饿死锁

在线程池中如果一个任务依赖于其他任务的执行，就可能产生死锁。如果线程池不够大就有可能所有线程执行的任务都阻塞在线程池中，等待着仍然处于同一工作队列中的其他任务，这称为线程饥饿死锁。

```

public class ThreadDeadlock {
    ExecutorService exec = Executors.newSingleThreadExecutor();
    public class LoadFileTask implements Callable<String> {
        private final String fileName;

        public LoadFileTask(String fileName) {
            this.fileName = fileName;
        }

        public String call() throws Exception {
            // Here's where we would actually read the file
            return "";
        }
    }

    public class RenderPageTask implements Callable<String> {
        public String call() throws Exception {
            Future<String> header, footer;
            header = exec.submit(new LoadFileTask("header.html"));
            footer = exec.submit(new LoadFileTask("footer.html"));
            String page = renderBody();
            //线程池不够大时，会经常出现死锁
            return header.get() + page + footer.get();
        }

        private String renderBody() {
            return "";
        }
    }
}

```

## 口耗时操作

如果任务由于过长的时间周期而阻塞，那么即使不出现死锁，线程池的响应性也会变的很差，这时如果线程池的大小相当于耗时任务数来说太小，就会出现性能低下。

解决方式，类库的阻塞方法一般都会提供限时和非限时两个版本，就是限定任务等待资源的时间，而不要无限制地等下去。

如果线程池频频被阻塞的任务充满，也就意味着应该调整线程池的大小了。

## 定制线程池的大小

线程池合理的长度取决于未来提交的任务类型和所部署系统的特性。而不要硬编码，可以由配置参数或Runtime.availableProcessors的结果动态计算出来。也可以采用Runtime.getRuntime().availableProcessors()来计算出CPU的大小。

对线程池的大小只要避免过大和过小两种情况就可以了，如果过大，会对CPU和内存资源造成浪费，如果过小，会影响吞吐量。为了正确配置，需要理解程序的计算环境、资源预算、任务自身的特点、CPU个数、内存大小等等。**如果你有不同类型的任务，它们之间还存在很大的差别，那么建议采用多个线程池，这样每个池可以根据不同任务的工作负载进行调节。**

一般池的大小定为CPU个数+1大小，如果存在I/O或是其他阻塞操作的任务，这个大小还可能要大。简单的计算公式如下：**线程池大小 = CPU个数×CPU使用率(0%-100%)×等待时间/任务计算时间。JVM会给每个线程分配固定的堆栈空间，用xss参数控制，默认64是1M，所以一台机器最大线程数=(机器本身内存-JVM堆xms值)/xss值。**

**另外还需要考虑linux系统的pid\_max, thread-max, max\_map\_count值**

**对于计算密集型的任务，一般会采用N+1的方式配置线程池大小，这样可以保证发生一个错误时不至于造成CPU周期中断工作。**

当任务需要使用池化的资源时，比如数据库连接，那么线程池的大小和资源池的大小就会相互影响，如果一个任务就需要一个数据库连接，那么资源池的大小就限制了线程池的大小。

## 配置ThreadPoolExecutor

ThreadPoolExecutor为一些Executor提供了基本的实现，这些Executor是由Executors中的工厂newCacheThreadPool、newFixedThreadPool返回的，它允许各种各样的用户定制。可以通过构造函数实例化一个ThreadPoolExecutor，个性化定制直到满意。

### □线程的创建与销毁

核心池的大小、最大池的大小和存活时间共同管理着线程的创建与销毁。核心池大小是目标的大小，线程池的实现试图维护池的大小，**池的大小等于核心池的大小**，并且直到工作队列充满前，池都不会创建更多的线程。最大池的大小是可同时活动的线程的上限，如果一个线程已经闲置的时间超过了存活时间，它将成为一个被回收的候选者，如果当前池的大小超过了核心池的大小，线程池会终止它。

通过调节核心大小和存活时间，可以促进线程池归还空闲线程占有的资源。如果把最大池大小设置为Integer.MAX\_VALUE。核心池大小设置为0，就可以定义一个无限大的线程池。

```

public ThreadPoolExecutor( int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit,
workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
}

```

## □管理队列任务

有限线程池限制了可以并发执行的任务的数量(单线程化的Executor是一个值得注意的特例，它们保证没有并发执行的任务，通过线程限制，提供了获得线程安全的可能性)。

ThreadPoolExecutor允许提供一个BlockingQueue来持有等待执行的任务，这些任务不会竞争CPU资源，任务排队有3种基本方式：**无限队列、有限队列和同步移交**。

**newFixedThreadPool**和**newSingleThreadExecutor**默认使用的是一个无限的**LinkedBlockingQueue**。如果所有的工作线程都处于忙碌状态，任务将会在队列中等候。如果任务持续地快速到达，超过了它们被执行的速度，队列也会无限制地增加。

一个稳妥的资源管理策略是使用有限队列，比如**ArrayBlockingQueue**、**LinkedBlockingQueue**、**PriorityBlockingQueue**。它可以控制资源耗尽的情况，但是当队列已满后，新的任务怎么办？**有很多的饱各策略可以处理这个问题**，对于一个有界队列，队列的长度与池的长度必须一起调节。一个大队列加一个小池，可以控制对内存和CPU的使用，但是吞吐量会受到限制。

对于庞大或无限的池，可以使用**SynchronousQueue**避开队列，将任务直接从生产移交工作线程。只有当池是无限的或者可以接受任务被拒绝**SynchronousQueue**才是一个有实际价值的选择，**newCacheThreadPool**工厂就使用了**SynchronousQueue**。

只有当任务彼此独立，才能使用有限线程或者有限工作队列的使用是合理的，否则有可能会引起死锁，这时不如使用**newCacheThreadPool**。

### 阻塞队列类别

- **ArrayBlockingQueue** :由数组结构组成的有界阻塞队列。FIFO原理
- **LinkedBlockingQueue** :由链表结构组成的有界阻塞队列。FIFO算法，采用了独立锁
- **PriorityBlockingQueue** :支持优先级排序的无界阻塞队列。
- **DelayQueue**:使用优先级队列实现的无界阻塞队列。只有到期的任务才能放行，可用于定时任务

- SynchronousQueue:不存储元素的阻塞队列。
- LinkedTransferQueue:由链表结构组成的无界阻塞队列。
- LinkedBlockingDeque:由链表结构组成的双向阻塞队列

## □饱和策略

当一个有限队列充满后，饱各策略开始起作用，`ThreadPoolExecutor`的饱各策略可以通过调用`setRejectedExecutionHandler`来修改，JDK提供了几种不同的策略：`AbortPolicy`（默认）、`CallerRunsPolicy`、`DiscardPolicy`、`DiscardOldestPolicy`。

`abort`: 会抛出异常，调用者可以捕获这个异常，然后编写自己的处理代码。

`discard`: 当最新提交的任务不能进入队列等待执行时，会放弃这个任务。遗弃最旧的(`discard-oldest`)会丢弃本应该接下来就执行的任务，并会尝试去重新提交新任务。如果工作队列是优先级队列，这个策略正好是丢弃优先级最高的元素，**所以这种策略不适合这种队列。**

`caller-runs`: 调用者运行策略即不会丢弃哪个任务，也不会抛出任何异常，它会把一些任务推回到调用者那里，以减缓新任务流。当所有的池线程都被占用，而且工作队列已充满后，下一个任务会在**主线程中**运行。这段时间主线程不能提交任何任务，同时这也给了工作线程时间来追赶进度，主线程也不会调用`accept`这样的操作，所有的请求会堆积到TCP层。时间长了，当服务器过载时，也会抛弃任务，然后它的负荷会逐渐地外移-最终转嫁到用户端，这种机制使得服务器在高负载下可以平缓地劣化。

```
//创建一个可变长的线程池，使用受限队列和调用者运行饱和策略
ThreadPoolExecutor executor = new ThreadPoolExecutor(N_THREADS, n_THREADS,
    0L, TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>(CAPACITY));
executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy)
```

```

//使用信号量来遏制任务的提交
public class BoundedExecutor {
    private final Executor exec;
    private final Semaphore semaphore;

    public BoundedExecutor(Executor exec, int bound) {
        this.exec = exec;
        this.semaphore = new Semaphore(bound);
    }

    public void submitTask(final Runnable command)
        throws InterruptedException {
        semaphore.acquire();
        try {
            exec.execute(new Runnable() {
                public void run() {
                    try {
                        command.run();
                    } finally {
                        semaphore.release();
                    }
                }
            });
        } catch (RejectedExecutionException e) {
            semaphore.release();
        }
    }
}

```

## □线程工厂

线程池是通过线程工厂创建一个线程的。详细指明一个线程工厂，能允许你定制池线程的配置信息，`ThreadFactory`只有一个`newThread`方法，它会在线程池需要创建一个新线程时调用，我们可以根据需要自定义一个线程工厂。

### 定制的线程工厂

这个例子给每个线程池传递一个名字，这样就可以判断错误来自于哪个池。也可以自定义线程基类。

```

public class MyThreadFactory implements ThreadFactory {
    private final String poolName;

    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }

    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName);
    }
}

```

## ✓自定义的线程基类

这个例子给线程提供名字，设置自定义的UncaughtExceptionHandler，以此向Logger中写入信息，还能维护统计信息等。

```
public class MyAppThread extends Thread {
    public static final String DEFAULT_NAME = "MyAppThread";
    private static volatile boolean debugLifecycle = false;
    private static final AtomicInteger created = new AtomicInteger();
    private static final AtomicInteger alive = new AtomicInteger();
    private static final Logger log = Logger.getAnonymousLogger();

    public MyAppThread(Runnable r) {
        this(r, DEFAULT_NAME);
    }

    public MyAppThread(Runnable runnable, String name) {
        super(runnable, name + "-" + created.incrementAndGet());
        setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
            public void uncaughtException(Thread t,
                                          Throwable e) {
                log.log(Level.SEVERE,
                        "UNCAUGHT in thread " + t.getName(), e);
            }
        });
    }

    public void run() {
        // Copy debug flag to ensure consistent value throughout.
        boolean debug = debugLifecycle;
        if (debug) log.log(Level.FINE, "Created " + getName());
        try {
            alive.incrementAndGet();
            super.run();
        } finally {
            alive.decrementAndGet();
            if (debug) log.log(Level.FINE, "Exiting " + getName());
        }
    }

    public static int getThreadsCreated() {
        return created.get();
    }

    public static int getThreadsAlive() {
        return alive.get();
    }

    public static boolean getDebug() {
        return debugLifecycle;
    }

    public static void setDebug(boolean b) {
        debugLifecycle = b;
    }
}
```

## 构造后再定制ThreadPoolExecutor

大多数通过构造函数传递给ThreadPoolExecutor的参数都可以在创建后能过setters来进行修改，如果Executor是通过Executors中的某个工厂方法创建的(newSingleThreadExecutor除外)，也可以把结果转型为ThreadPoolExecutor，然后访问setters方法。

Executors有一个unconfigurableExecutorService的工厂方法，它返回一个现有的ExecutorService，并对它进行包装，它只暴露出ExecutorService的方法，所以不能再进一步进行配置。

可以把这种技术用在自己的Executor中，来防止执行策略被修改，可以用一个unconfigurableExecutorService包装它。

```
ExecutorService exec = Executors.newCachedThreadPool();
if (exec instanceof ThreadPoolExecutor)
    ((ThreadPoolExecutor)exec).setCorePoolSize(10);
else
    throw new AssertionError("error");
```

## 扩展ThreadPoolExecutor

ThreadPoolExecutor是可以扩展的，它提供了几个钩子让子类去覆写**beforeExecute**、**afterExecute**和**terminate**，用它们可以添加日志、时序、监视器或统计信息收集的功能，这些钩子方法是在执行任务的线程中被调用的。

任务正常还是异常结束afterExecute都会被调用，如果任务完成后抛出一个Error，则afterExecute不会被调用。如果beforeExecute抛出一个RuntimeException任务不会执行，afterExecute也不会被调用。

terminate会在线程池完成关闭动作后调用，也就是当所有任务和所有工作线程都已关闭，它就行执行，可以用来释放Executor在生命周期内分配到的资源，还可以发出通知，记录日志、统计信息等操作。

## □给线程池加入统计信息功能

```
public class TimingThreadPool extends ThreadPoolExecutor {

    public TimingThreadPool() {
        super(1, 1, 0L, TimeUnit.SECONDS, null);
    }

    private final ThreadLocal<Long> startTime = new
    ThreadLocal<Long>();
    private final Logger log = Logger.getLogger("TimingThreadPool");
    private final AtomicLong numTasks = new AtomicLong();
    private final AtomicLong totalTime = new AtomicLong();

    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        log.fine(String.format("Thread %s: start %s", t, r));
        startTime.set(System.nanoTime());
    }

    protected void afterExecute(Runnable r, Throwable t) {
        try {
            long endTime = System.nanoTime();
            long taskTime = endTime - startTime.get();
            numTasks.incrementAndGet();
            totalTime.addAndGet(taskTime);
            log.fine(String.format("Thread %s: end %s, time=%dns",
                t, r, taskTime));
        } finally {
            super.afterExecute(r, t);
        }
    }

    protected void terminated() {
        try {
            log.info(String.format("Terminated: avg time=%dns",
                totalTime.get() / numTasks.get()));
        } finally {
            super.terminated();
        }
    }
}
```

# 并行递归算法

当每次迭代都彼此独立，循环体如果包含重要的计算或要执行潜在的阻塞操作，那么这种循环都是并行化工作的目标。这节中提供了并行递归算法，用于解决类似推箱子这样的游戏的算法。

## ✓顺序递归转化为并行递归

```
public abstract class TransformingSequential {

    void processSequentially(List<Element> elements) {
        for (Element e : elements)
            process(e);
    }

    void processInParallel(Executor exec, List<Element> elements) {
        for (final Element e : elements)
            exec.execute(new Runnable() {
                public void run() {
                    process(e);
                }
            });
    }

    public abstract void process(Element e);

    public <T> void sequentialRecursive(List<Node<T>> nodes,
                                         Collection<T> results) {
        for (Node<T> n : nodes) {
            results.add(n.compute());
            sequentialRecursive(n.getChildren(), results);
        }
    }
}
```

```

public <T> void parallelRecursive(final Executor exec,
                                List<Node<T>> nodes,
                                final Collection<T> results) {
    for (final Node<T> n : nodes) {
        exec.execute(new Runnable() {
            public void run() {
                results.add(n.compute());
            }
        });
        parallelRecursive(exec, n.getChildren(), results);
    }
}
//等待并行运算的结果
public <T> Collection<T> getParallelResults(List<Node<T>> nodes)
    throws InterruptedException {
    ExecutorService exec = Executors.newCachedThreadPool();
    Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();
    parallelRecursive(exec, nodes, resultQueue);
    exec.shutdown();
    exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
    return resultQueue;
}

interface Element {
}

interface Node <T> {
    T compute();

    List<Node<T>> getChildren();
}
}

```

## 走迷宫游戏

```

//P代表位置, M代表移动
public interface Puzzle <P, M> {
    P initialPosition();

    boolean isGoal(P position);

    Set<M> legalMoves(P position);

    P move(P position, M move);
}

```

```

@Immutable
public class PuzzleNode <P, M> {
    final P pos;
    final M move;
    final PuzzleNode<P, M> prev;

    public PuzzleNode(P pos, M move, PuzzleNode<P, M> prev) {
        this.pos = pos;
        this.move = move;
        this.prev = prev;
    }

    List<M> asMoveList() {
        List<M> solution = new LinkedList<M>();
        for (PuzzleNode<P, M> n = this; n.move != null; n = n.prev)
            solution.add(0, n.move);
        return solution;
    }
}

```

```

/*可携带结果的闭锁*/
@ThreadSafe
public class ValueLatch <T> {
    @GuardedBy("this") private T value = null;
    private final CountDownLatch done = new CountDownLatch(1);

    public boolean isSet() {
        return (done.getCount() == 0);
    }

    public synchronized void setValue(T newValue) {
        if (!isSet()) {
            value = newValue;
            done.countDown();
        }
    }

    public T getValue() throws InterruptedException {
        done.await();
        synchronized (this) {
            return value;
        }
    }
}

```

```

public class ConcurrentPuzzleSolver <P, M> {
    private final Puzzle<P, M> puzzle;//P代表位置, M代表移动
    private final ExecutorService exec;
    private final ConcurrentMap<P, Boolean> seen;
    protected final ValueLatch<PuzzleNode<P, M>> solution = new
ValueLatch<PuzzleNode<P, M>>();

    public ConcurrentPuzzleSolver(Puzzle<P, M> puzzle) {
        this.puzzle = puzzle;
        this.exec = initThreadPool();
        this.seen = new ConcurrentHashMap<P, Boolean>();
        if (exec instanceof ThreadPoolExecutor) {
            ThreadPoolExecutor tpe = (ThreadPoolExecutor) exec;
            tpe.setRejectedExecutionHandler(new
ThreadPoolExecutor.DiscardPolicy());
        }
    }

    private ExecutorService initThreadPool() {
        return Executors.newCachedThreadPool();
    }

    public List<M> solve() throws InterruptedException {
        try {
            P p = puzzle.initialPosition();
            exec.execute(newTask(p, null, null));
            // 阻塞直到发现一个方案
            PuzzleNode<P, M> solnPuzzleNode = solution.getValue();
            return (solnPuzzleNode == null) ? null :
solnPuzzleNode.asMoveList();
        } finally {
            exec.shutdown();
        }
    }

    protected Runnable newTask(P p, M m, PuzzleNode<P, M> n) {
        return new SolverTask(p, m, n);
    }

    protected class SolverTask extends PuzzleNode<P, M> implements
Runnable {
        SolverTask(P pos, M move, PuzzleNode<P, M> prev) {
            super(pos, move, prev);
        }

        public void run() {
            if (solution.isSet()
                || seen.putIfAbsent(pos, true) != null)
                return; // 已找到一个方案, 或该位置曾经到达过
            if (puzzle.isGoal(pos))
                solution.setValue(this);
            else
                for (M m : puzzle.legalMoves(pos))
                    exec.execute(newTask(puzzle.move(pos, m), m,
this));
        }
    }
}

```

```

/*能够感知任务不存在的解决者*/
public class PuzzleSolver <P, M> extends ConcurrentPuzzleSolver<P, M> {
    PuzzleSolver(Puzzle<P, M> puzzle) {
        super(puzzle);
    }

    private final AtomicInteger taskCount = new AtomicInteger(0);

    protected Runnable newTask(P p, M m, PuzzleNode<P, M> n) {
        return new CountingSolverTask(p, m, n);
    }

    class CountingSolverTask extends SolverTask {
        CountingSolverTask(P pos, M move, PuzzleNode<P, M> prev) {
            super(pos, move, prev);
            taskCount.incrementAndGet();
        }

        public void run() {
            try {
                super.run();
            } finally {
                if (taskCount.decrementAndGet() == 0)
                    solution.setValue(null);
            }
        }
    }
}

```

## 小结

对于并发执行的任务，Executor框架是强大且灵活的，它提供了大量的可调节的选项，比如创建和关闭线程的策略，处理队列任务的策略，处理过多任务的策略，并且提供了几个钩子函数用于扩展它的行为。然而，和大多数强大的框架一样，草率地将一些设定组合在一起，并不能很好地工作，一些类型的任务需要特定的执行策略，而一些调节参数组合在一起后可能产生意外的结果。