二

全面了解 JDK 线程池实现原理

前言

线程池，顾名思义就是存放线程的池子，池子中存放了很多可复用的线程。使用线程池，具有以下优点：

- 通过可复用的线程，避免线程频繁创建和销毁，降低系统资源消耗
- 提高系统任务处理速度
- 增加线程的可管理性，使用线程池可以对线程进行统一监控、分配等

基础构造

回顾 JDK 线程池中有核心线程和非核心线程之分，当线程数未达到**核心线程数**时，添加一个任务就会创建一个线程，当达到核心线程数时，为了控制线程的数量，多余的任务将会放入**阻塞队列**中，当队列也满了的话，此时为了提高系统的执行任务的效率，便会创建非核心线程，当然非核心线程也不能无限制创建，当达到**最大线程数**时，便会执行**拒绝策略**。

这里有核心线程和非核心线程，其含义有：

- 核心和非核心只是线程在不同阶段的称呼，两者的本质是完全一致的
- 区分核心非核心主要是为了控制线程的数量

因此从上面我们能够归纳出一个线程池的构造结构有核心线程数、阻塞队列、最大线程数、拒绝策略，通过代码可以表示为：

```
public class LimynlThreadPool{

    /**
     * 线程池名称
     */
    private String name;

    /**
     * 核心线程数
```

```
     */
    private int coreSize;

    /**
     * 最大线程数
     */
    private int maxSize;

    /**
     * 阻塞队列
     */
    private BlockingQueue<Runnable> taskQueue;

    /**
     * 拒绝策略
     */
    private RejectPolicy rejectPolicy;

    public LimynlThreadPoolExecutor(String name, int coreSize, int maxSize, Blockin
        this.name = name;
        this.coreSize = coreSize;
        this.maxSize = maxSize;
        this.taskQueue = taskQueue;
        this.rejectPolicy = rejectPolicy;
    }
}
```

继续分析，对于普通用户来说使用线程池的目的就是将需要执行的任务提交到线程池中去执行，因此我们还需要定义一个提交任务的入口，为了实现面向接口编程，我们抽象一个接口：

```
public interface Executor {
    /**
     * 任务执行入口
     */
    void execute(Runnable task);
}
```

执行无返回值任务

从上面分析可知，我们的线程池的的整体流程可化分为两大部分：即任务执行部分、线程创建部分。

任务执行部分

对于任务执行部分主要流程为：

- 如果运行的线程数（runningCount）小于核心线程数，则创建一个线程来执行任务

- 如果达到核心线程数，则将任务提交到阻塞队列
- 如果添加队列失败，即队列已满，创建非核心线程来执行任务
- 如果线程数达到最大线程数，则执行拒绝策略

注意这里线程数量计数器 runningCount 为了保证多线程环境下的原子性和可见性，我们可以使用 AtomicInteger 来修饰，因为 AtomicInteger 底层就是使用 CAS 来保证原子性，使用 volatile 来保证可见性。

此时我们的线程池为：

```
public class LimynlThreadPool implements Executor {

    private String name;
    private int coreSize;
    private int maxSize;
    private BlockingQueue<Runnable> taskQueue;
    private RejectPolicy rejectPolicy;

    /**
     * 当前正在运行的线程数(底层采用 CAS+volatile 实现)
     */
    private AtomicInteger runningCount = new AtomicInteger(0);

    // 线程计数器
    private AtomicInteger threadCounter = new AtomicInteger(0);

    public LimynlThreadPoolExecutor(String name, int coreSize, int maxSize, Blockin
        this.name = name;
        this.coreSize = coreSize;
        this.maxSize = maxSize;
        this.taskQueue = taskQueue;
        this.rejectPolicy = rejectPolicy;
    }

    @Override
    public void execute(Runnable task) {
        // 线程池的线程总数
        int count = runningCount.get();
        // 如果正在运行的线程数小于核心线程数，直接创建一个线程
        if (count < coreSize) {
            if (addWorker(task, true)) {
                return;
            }
        }

        // 如果达到核心线程数，则将任务提交到阻塞队列
        if(!taskQueue.offer(task)){
            // 如果队列已满，创建非核心线程来执行任务
            if (!addWorker(task, false)) {
                // 线程数达到最大线程数，执行拒绝策略
                rejectPolicy.reject(task, this);
```

```
            }
        }
    }
}
```

线程创建部分

对于线程创建部分主要是看线程池中创建的线程是否达到了最大线程数，如果达到了最大线程数直接返回，如果没有达到最大线程数，直接创建线程，并且执行新任务，以后从阻塞队列中获取任务执行。

```
private boolean addWorker(Runnable newTask, boolean isCore) {
    while(true){
        int count = runningCount.get();
        int maxNum = isCore ? coreSize : maxSize;
        // 如果当前线程数量大于核心线程数或者非核心线程数，直接返回执行下一步
        if (count >= maxNum) {
            return false;
        }

        // 原子修改线程数，如果失败，说明其他线程在修改，通过外层的 while 实现重试
        if (runningCount.compareAndSet(count, count + 1)) {
            String threadName = name + threadCounter.incrementAndGet();
            // 创建线程并启动
            new Thread(() -> {
                System.out.println("thread name : " + Thread.currentThread().getNam
                Runnable task = newTask;
                // 第一次执行完新任务后，以后直接从阻塞队列中获取任务并执行，
                // 如果阻塞队列中没有任务，将会一直阻塞到这里，直到有任务可执行为止
                // 注意：从这里可以看出来对于线程池来说，为什么使用的是阻塞队列而不是普通
                while (task != null || (task = getTask()) != null) {
                    try {
                        // 执行真正的任务
                        task.run();
                    } finally {
                        // 任务执行完成，置为 null
                        task = null;
                    }
                }
            }, threadName).start();
            break;
        }
    }
    return true;
}

private Runnable getTask() {
    try {
        // 一直阻塞线程知道取到任务为止，维持核心线程的存活
        return taskQueue.take();
    } catch (InterruptedException e) {
        // 如果线程中断，说明当前线程已终止，线程池中的总数量应该减 1
```

```
            runningCount.decrementAndGet();
            return null;
        }
    }
```

拒绝策略

到这里一个线程池的雏形已经构建出来了，但是对于线程池任务队列已经满了，并且最大线程数也达到了最大值，因此再次提交的任务需要执行拒绝策略，常见的拒绝策略有：直接抛弃、调用线程自己处理、抛出异常、丢弃队列中最古老的任务等等。这里我们以直接抛弃为例，看看如果自定义拒绝策略。

首先定义线程池的公共接口：

```
public interface RejectPolicy {
    void reject(Runnable task, LimynlThreadPool limynlThreadPool);
}
```

实现自定义拒绝策略：

```
public class DiscardRejectPolicy implements RejectPolicy {
    @Override
    public void reject(Runnable task, LimynlThreadPool limynlThreadPool) {
        System.out.println("discard one task");
    }
}
```

到此我们基于 Runnable 实现不带返回值的任务的线程池，下面通过测试用例验证。

```
public class Main {

    public static void main(String[] args) {
        Executor threadPool = new LimynlThreadPool("test", 5, 8, new ArrayBlockingQ
        AtomicInteger counter = new AtomicInteger(0);

        IntStream.range(0, 15).forEach(index -> threadPool.execute(() -> {
            try {
                Thread.sleep(1000);
                System.out.println("running: " + System.currentTimeMillis() + ": "
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }));
    }
}
```

执行结果为：

```
thread name : test1
thread name : test6
thread name : test7
discard one task
thread name : test5
thread name : test4
thread name : test3
thread name : test2
discard one task
thread name : test8
running: 1601343696818: 1
running: 1601343696818: 5
running: 1601343696818: 7
running: 1601343696818: 3
running: 1601343696818: 6
running: 1601343696818: 2
running: 1601343696818: 4
running: 1601343696818: 8
running: 1601343697819: 9
running: 1601343697819: 13
running: 1601343697819: 10
running: 1601343697819: 12
running: 1601343697819: 11
```

执行有返回值任务

从上一节知道，我们使用 Runnable 实现了无返回值的任务线程池执行能力。但是有时我们需要获取任务的执行结果。因此当主线程将任务提交到线程池中后，需要等待任务执行完毕后才能获取到任务的结果。因此我们应该清楚，当任务未执行或者还在执行过程中，当前线程如果需要获取该任务的执行结果，需要阻塞，直到任务执行完成。 对于无返回值任务的执行，我们使用的 Runnable，对于有返回值任务，我们使用 Callable 实现，因此对于线程执行接口，我们需要进行改造，以至于能够返回任务的执行结果。

```
public interface FutureExecutor extends Executor {
    // 将任务的执行结果包装到 Future 中
    <T> Future<T> submit(Callable<T> task);
}

public interface Future<T> {
    /**
     * 获取任务的结果
     */
    T get();
}
```

任务定义

接下来我们需要一种新任务，这种任务既能够执行，又能够返回执行结果，因此我们可以同时实现 Runnabl、Future 接口，为了表示任务所处的阶段，我们需要定义任务的状态，因此新任务的定义如下：

```
public class FutureTask<T> implements Runnable, Future {
    /**
     * 任务执行状态：0 未开始 1 正常完成 2 异常完成
     */
    private static final int NEW = 0;
    private static final int FINISHED = 1;
    private static final int EXCEPTION = 2;

    /**
     * 真正需要执行的任务
     */
    private Callable<T> task;

    // 通过原子类去更新线程状态
    private AtomicInteger state = new AtomicInteger(NEW);

    /**
     * 保存任务的执行结果
     */
    private Object result;

    public FutureTask(Callable<T> task) {
        this.task = task;
    }

    @Override
    public T get() {
        return null;
    }

    @Override
    public void run() {
    }
}
```

因此到这里我们的准备工作都完成了，接下里看如何执行任务，以及如何获取任务执行结果。

任务执行部分

- 对于线程的执行部分首先我们需要检测线程是否是新建状态，如果不是直接返回
- 如果是新建任务直接执行任务
- 原子更新当前任务状态
- 判断调用者是否为空，如果不为空，则唤醒

```
@Override
public void run() {
    // 任务不是新建状态，说明执行过了
    if (state.get() != NEW) {
        return;
    }
    try {
        // 执行真正的任务
        T t = task.call();

        /**
         * 原子更新任务状态
         */
        if (state.compareAndSet(NEW, FINISHED)) {
            this.result = t;
            // 检查是否有调用者
            finish();
        }
    } catch (Exception e) {
        // 任务异常，直接返回异常
        if (state.compareAndSet(NEW, EXCEPTION)) {
            this.result = e;
            finish();
        }
    }
}

private void finish() {
    for (Thread c; (c = caller.get()) != null; ) {
        if (caller.compareAndSet(c, null)) {
            // 使用 unpark 唤醒调用线程
            LockSupport.unpark (c);
        }
    }
}
```

获取任务执行结果

获取执行结果主要流程为：

- 如果任务执行过，直接返回结果
- 如果任务执行过程发生异常，直接抛出异常，此时异常将会抛到子线程外
- 如果任务处于执行过程中，需要判断调用者线程是否需要阻塞

```
@Override
public T get() {
    int s = state.get();
    // 如果线程还在执行，需要判断
    if (s == NEW) {
        // 为了判断当前调用者是否已经阻塞，我们需要一个标识
```

```
            boolean flag = false;
            for (; ; ) {
                s = state.get();
                // 说明当前任务执行完成，跳出循环
                if (s > NEW) {
                    break;
                } else if (!flag) {
                    marked = caller.compareAndSet(null, Thread.currentThread());
                } else {
                    // 将当前调用者线程阻塞，直到任务执行完成，被唤醒
                    LockSupport.park();
                }
            }
        }

        // 如果线程执行过，直接返回结果
        if (s == FINISHED) {
            return (T) result;
        }

        throw new RuntimeException((Throwable) result);
    }
```

线程提交入口

上面我们把具有执行返回值的任务整体流程梳理完成，接下来就是如何吧任务提交给线程池，以及如何把我们定义的任务包装成具有执行能力和返回结果。

为了实现之前代码的复用，我们直接继承 LimynlThreadPool 并且实现 FutureExecutor：

```
public class LimynlThreadPoolFuture extends LimynlThreadPool implements FutureExecu

    public LimynlThreadPoolFutureExecutor(String name,
                                          int coreSize,
                                          int maxSize,
                                          BlockingQueue<Runnable> taskQueue,
                                          RejectPolicy rejectPolicy) {
        super(name, coreSize, maxSize, taskQueue, rejectPolicy);
    }

    @Override
    public <T> Future<T> submit(Callable<T> task) {
        // 将任务进行包装
        FutureTask<T> futureTask = new FutureTask<>(task);
        // 将任务提交到线程池
        execute(futureTask);
        // 返回 Future，通过访问 get 方法就能够获取任务的执行结果
        return futureTask;
    }
}
```

接下来我们验证我们的代码：

```
public class Main {

    public static void main(String[] args) {
        FutureExecutor executor = new LimynlThreadPoolFutureExecutor("test", 2, 4,
                new ArrayBlockingQueue<>(6), new DiscardRejectPolicy());
        List<Future<Integer>> list = new ArrayList<>();
        IntStream.range(0, 10).forEach(i -> {
            Future<Integer> future = executor.submit(() -> {
                Thread.sleep(1000);
                System.out.println("running: " + i);
                return i;
            });

            list.add(future);
        });

        list.forEach(item -> System.out.println(item.get()));
    }
}
```

运行结果：

```
thread name : core_test1
thread name : core_test2
thread name : test3
thread name : test4
running: 0
0
running: 1
1
running: 8
running: 9
running: 2
2
running: 3
3
running: 4
4
running: 5
5
running: 6
6
running: 7
7
8
9
```

回顾

以上我们实现了基于 Runnable 实现不带返回值的任务，基于 Callable 实现带返回值的任务，通过原子类实现状态更新。整个过程也是比较好理解的，整体框架原型其实就是 JDK 线程池 ThreadPoolExecutor 的大体流程。通过手写线程池，回头再去看看线程池源码相信会有不一样的收获。

上一页 下一页