链滴

java 多线程及线程池的使用

作者：gongxiongzhuang

原文链接：https://ld246.com/article/1557481151073

来源网站：链滴

在项目里面为了提高性能往往会在主线程里面开启一个新线程去执行，这种做法最方便快捷，但是当户量数据上涨，很显然每次去开启新的线程服务器往往会吃不消，这时就需要线程池来管理和监控线的状态。

一、创建多线程的方式

java多线程很常见，如何使用多线程，如何创建线程，java中有两种方式，第一种是让自己的类实现Rnnable接口，第二种是让自己的类继承Thread类。其实Thread类自己也是实现了Runnable接口。

1.通过实现Runnable接口

```
public class Mytheard1 implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 30; i++) {
            System.out.println("thread#1===" + i);
            try {
                Thread.sleep(100L);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

2.通过继承Thread接口

```
public class Mytheard2 extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 30; i++) {
            System.out.println("thread#2===" + i);
            try {
                Thread.sleep(100L);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

启动上面两个线程

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

        long startTime = System.currentTimeMillis();
        // 通过主线程启动自己的线程

        // 通过实现runnable接口
```

```
        Thread thread1 = new Thread(new Mytheard1());
        //thread1.setDaemon(true);//守护线程
        thread1.start();

        // 通过继承thread类
        Thread thread2 = new Thread(new Mytheard2());
        //thread2.setDaemon(true);
        thread2.start();

        // 注意这里不是调运run()方法，而是调运线程类Thread的start方法，在Thread方法内部，会运本地系统方法，最终会自动调运自己线程类的run方法

        System.out.println("mainThread isDaemon:"
                + Thread.currentThread().isDaemon());
        System.out.println("thread1 isDaemon:" + thread1.isDaemon());
        System.out.println("thread2 isDaemon:" + thread2.isDaemon());

        // 让主线程睡眠
        Thread.sleep(1000L);
        System.out.println("主线程结束！用时："
                + (System.currentTimeMillis() - startTime));
        //System.exit(0);
    }
```

上面两种方式更推荐实现Runnable接口；Runnable面向接口，拓展性更好，而且可以防止java单继的限制。

二、线程类型的简单说明

java中线程一共有两种类型：守护线程（daemon thread）和用户线程（user thread）又叫非守护程

1.守护线程

可以通过thread.setDaemon(true)方法设置线程是否为守护线程，thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常。在守护线程中开启的新线也将是守护线程。守护线程顾名思义是用来守护的，是给所有得非守护进程提供服务的，所以在jvm行完所有的非守护进程之后，jvm就会停止，守护线程也不会再运行，最典型的守护线程就是java的圾回收机制(GC)。

2.非守护线程

java线程默认设置是非守护线程thread.setDaemon(false)。当主线程运行完之后，只要主线程里面非守护线程jvm就不会退出，直到所有的非守护线程执行完之后jvm才会退出。

总结：如果把一个线程设置成守护线程，则jvm的退出就不会关心当前线程的执行状态。

三、线程池的使用

上面代码中可以直接新起线程，如果100个并发同时访问主线程也就是短时间就启动了200个线程，20个线程同时工作，逻辑上是没有任何问题的，但是这样做对系统资源的开销很大。基于这样的考虑，要考虑启用线程池，线程池里有很多可用线程资源，如果需要就直接从线程池里拿就是。当不用的时

，线程池会自动帮我们管理。所以使用线程池主要有以下两个好处：**1、减少创建和销毁线程上所花的时间以及系统资源的开销 2、如不使用线程池，有可能造成系统创建大量线而导致消耗完系统内存 。**

1.自定义线程池

定义单例线程池

```
public class MyPool {

    private static MyPool myPool = null;
    //单例线程池中有两种具体的线程池
    private ThreadPoolExecutor threadPool = null;
    private ScheduledThreadPoolExecutor scheduledPool = null;

    public ThreadPoolExecutor getThreadPool() {
        return threadPool;
    }

    public ScheduledThreadPoolExecutor getScheduledPool() {
        return scheduledPool;
    }

    //设置线程池的各个参数的大小
    private int corePoolSize = 10;// 池中所保存的线程数，包括空闲线程。
    private int maximumPoolSize = 20;// 池中允许的最大线程数。
    private long keepAliveTime = 3;// 当线程数大于核心时，此为终止前多余的空闲线程等待新任
的最长时间。
    private int scheduledPoolSize = 10;

    private static synchronized void create() {
        if (myPool == null)
            myPool = new MyPool();
    }

    public static MyPool getInstance() {
        if (myPool == null)
            create();
        return myPool;
    }

    private MyPool() {
        //实例化线程池，这里使用的 LinkedBlockingQueue 作为 workQueue ， 使用 DiscardOldest
olicy 作为 handler
        this.threadPool = new ThreadPoolExecutor(corePoolSize, maximumPoolSize,
                keepAliveTime, TimeUnit.SECONDS,
                new LinkedBlockingQueue<>(),
                new ThreadPoolExecutor.CallerRunsPolicy());//不在新线程中执行任务，而是由调用者
在的线程来执行
        //实例化计划任务线程池
        this.scheduledPool = new ScheduledThreadPoolExecutor(scheduledPoolSize);
    }
}
```

创建线程池的主要参数说明

● corePoolSize（int）：线程池中保持的线程数量，包括空闲线程在内。也就是线程池释放的最小线数量界限。

● maximumPoolSize（int）:线程池中嫩容纳最大线程数量。

● keepAliveTime(long):空闲线程保持在线程池中的时间，当线程池中线程数量大于corePoolSize的候。

● unit(TimeUnit枚举类):上面参数时间的单位，可以是分钟，秒，毫秒等等。

● workQueue（BlockingQueue <Runnable>）:任务队列，当线程任务提交到线程池以后，首先入队列中，然后线程池按照该任务队列依次执行相应的任务。可以使用的workQueue有很多，比如：LnkedBlockingQueue等等。

● threadFactory(ThreadFactory类):新线程产生工厂类。

● handler（RejectedExecutionHandler类）:当提交线程拒绝执行、异常的时候，处理异常的类。类取值如下：(注意都是内部类)

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务，重复过程。

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务。

获取线程池并添加任务

```
    public void testThreadPool() {

        ThreadPoolExecutor pool1 = (ThreadPoolExecutor) Executors.newCachedThreadPool();

        pool1.execute(() -> System.out.println("快捷线程池中的线程！"));


        ThreadPoolExecutor pool2 = MyPool.getInstance().getThreadPool();
        pool2.execute(() -> {
            System.out.println("pool2 普通线程池中的线程！");
            try {
                Thread.sleep(30*1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        System.out.println("pool2 poolSize:"+pool2.getPoolSize());
        System.out.println("pool2 corePoolSize:"+pool2.getCorePoolSize());
        System.out.println("pool2 largestPoolSize:"+pool2.getLargestPoolSize());
        System.out.println("pool2 maximumPoolSize:"+pool2.getMaximumPoolSize());

        ScheduledThreadPoolExecutor pool3 = MyPool.getInstance().getScheduledPool();
        pool3.scheduleAtFixedRate(() -> System.out.println("计划任务线程池中的线程！"), 0, 5000,
TimeUnit.MILLISECONDS);
    }
```

2.jdk提供的常用线程池

java提供了几种常用的线程池，可以快捷的供程序员使用

- newFixedThreadPool 创建固定大小数量线程池，数量通过传入的参数决定。
- newSingleThreadExecutor 创建一个线程容量的线程池，所有的线程依次执行，相当于创建固定量为1的线程池。
- newCachedThreadPool 创建可缓存的线程池，没有最大线程限制（实际上是Integer.MAX_VALU）。如果用空闲线程等待时间超过一分钟，就关闭该线程。
- newScheduledThreadPool 创建计划(延迟)任务线程池,线程池中的线程可以让其在特定的延迟时之后执行，也可以以固定的时间重复执行（周期性执行）。相当于以前的Timer类的使用。
- newSingleThreadScheduledExecutor 创建单线程池延迟任务，创建一个线程容量的计划任务。

3.spring boot 中使用线程池

如果使用spring框架的朋友，可以直接使用spring封装的线程池，由spring容器管理。

spring boot中有两种方式配置线程池，一种是自定义配置，二种是修改原生spring异步线程池的装。

1.自定义线程池

```
@Configuration
@EnableAsync//开启线程池
public class TaskExecutePool {

    @Autowired
    private TaskThreadPoolConfig config;

    @Bean
    public Executor myTaskAsyncPool() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        //核心线程池大小
        executor.setCorePoolSize(config.getCorePoolSize());
        //最大线程数
        executor.setMaxPoolSize(config.getMaxPoolSize());
        //队列容量
        executor.setQueueCapacity(config.getQueueCapacity());
        //活跃时间
        executor.setKeepAliveSeconds(config.getKeepAliveSeconds());
        //线程名字前缀
        executor.setThreadNamePrefix("MyExecutor-");

        // setRejectedExecutionHandler：当pool已经达到max size的时候，如何处理新任务
        // CallerRunsPolicy：不在新线程中执行任务，而是由调用者所在的线程来执行
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
        executor.initialize();
        return executor;
    }
}
```

2.修改原生spring异步线程池的装配

```
@Configuration
```

```
@EnableAsync
public class NativeAsyncTaskExecutePool implements AsyncConfigurer {
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    //注入配置类
    @Autowired
    TaskThreadPoolConfig config;

    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        //核心线程池大小
        executor.setCorePoolSize(config.getCorePoolSize());
        //最大线程数
        executor.setMaxPoolSize(config.getMaxPoolSize());
        //队列容量
        executor.setQueueCapacity(config.getQueueCapacity());
        //活跃时间
        executor.setKeepAliveSeconds(config.getKeepAliveSeconds());
        //线程名字前缀
        executor.setThreadNamePrefix("MyExecutor2-");

        // setRejectedExecutionHandler：当pool已经达到max size的时候，如何处理新任务
        // CallerRunsPolicy：不在新线程中执行任务，而是由调用者所在的线程来执行
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
        executor.initialize();
        return executor;
    }


    /**
     *  异步任务中异常处理
     * @return
     */
    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
        return (ex, method, objects) -> {
            logger.error("==========================="+ex.getMessage()+"=======
==============", ex);
            logger.error("exception method:"+method.getName());
        };
    }
}
```

线程池配置类

```
@Component
public class TaskThreadPoolConfig {
    @Value("${task.pool.corePoolSize}")
    private int corePoolSize;
    @Value("${task.pool.maxPoolSize}")
    private int maxPoolSize;
    @Value("${task.pool.keepAliveSeconds}")
    private int keepAliveSeconds;
    @Value("${task.pool.queueCapacity}")
```

```
    private int queueCapacity;

    ......//省略get()，set()方法
}
```

配置文件配置线程池大小

```
# spring 线程池
task:
  pool:
    #核心线程池
    corePoolSize: 500
    #最大线程池
    maxPoolSize: 1000
    #活跃时间
    keepAliveSeconds: 300
    #队列容量
    queueCapacity: 50
```

需要异步线程执行的任务

```
@Component
public class AsyncTask {
    protected final Logger logger = LoggerFactory.getLogger(this.getClass());

    @Async("myTaskAsyncPool")  //myTaskAsynPool即配置线程池的方法名，此处如果不写自定线程池的方法名，会使用默认的线程池
    public void doTask1(int i) {
        logger.info("Task"+i+" started.");
    }

    @Async//使用默认的线程池
    public void doTask2(int i) {
        if (i == 0) {
            throw new NullPointerException();
        }
        logger.info("Task2-Native"+i+" started.");
    }

    @Async//使用默认的线程池并返回参数
    public ListenableFuture<String> doTask3(int i) {
        logger.info("Task3-返回值"+i+" started.");
        return new AsyncResult<>(i + "");
    }
}
```

获取线程池，并执行任务

```
@Test
public void AsyncTaskTest() {
    for (int i = 0; i < 10000; i++) {
        try {
            //自定义线程池
            asyncTask.doTask1(i);
            //spring异步线程池
```

```
            asyncTask.doTask2(i);

            String text = asyncTask.doTask3(i).get();//阻塞调用
            System.out.println(text);
            String context = asyncTask.doTask3(i).get(1, TimeUnit.SECONDS);//限时调用
            System.out.println(context);
        } catch (InterruptedException | ExecutionException | TimeoutException e) {
            e.printStackTrace();
        }
    }
    logger.info("All tasks finished.");
}
```