带你一次搞明白 Java 多线程（IX）

6.2 捕获线程的执行异常

在线程的run方法中,如果有受检异常必须进行捕获处理,如果想要获得run()方法中出现的运行时异常信息,可以通过回调UncaughtExceptionHandler接口获得哪个线程出现了运行时异常.在Thread类中有关处理运行异常的方法有:

getDefaultUncaughtExceptionHandler() 获得全局的(默认的)UncaughtExceptionHandler

getUncaughtExceptionHandler() 获得当前线程的UncaughtExceptionHandler

setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) 设置全局的UncaughtExceptionHandler

setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) 设置当前线程的UncaughtExceptionHandler

当线程运行过程中出现异常,JVM会调用Thread类的dispatchUncaughtException(Throwable e)方法，该方法会调用getUncaughtExceptionHandler().uncaughtException(this, e); 如果想要获得线程中出现异常的信息，就需要设置线程的UncaughtExceptionHandler

```
package com.wkcto.threadexception;

import java.io.FileInputStream;

/**
 * 演示设置线程的 UnCaughtExceptionHandler 回调接口
 */
public class Test01 {
    public static void main(String[] args) {

        //1)设置线程全局的回调接口

        Thread.setDefaultUncaughtExceptionHandler(new  Thread.UncaughtExceptionHandler()
{
            @Override
            public void uncaughtException(Thread t, Throwable e) {

                //t 参数接收发生异常的线程, e 就是该线程中的异常

                System.out.println(t.getName() + "线程产生了异常: " + e.getMessage());

            }
        });

        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {

                System.out.println(Thread.currentThread().getName() + "开始运行");

                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {

                    //线程中的受检异常必须捕获处理

                    e.printStackTrace();
                }

                System.out.println(12 / 0 );       //会产生算术异常


            }
        });
        t1.start();
```

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                String txt = null;

                System.out.println( txt.length());    //会产生空指针异常

            }
        }).start();

        /*
            在实际开发中,这种设计异常处理的方式还是比较常用的,尤其是异常执行的方法

            如果线程产生了异常, JVM 会调用 dispatchUncaughtException()方法,在该方法中调用了 getUncaughtExceptionHandler().uncaughtException(this, e); 如果当前线程设置了 UncaughtExceptionHandler 回调接口就直接调用它自己的 uncaughtException 方法, 如果没有设置则调用当前线程所在线程组 UncaughtExceptionHandler 回调接口的 uncaughtException 方法,如果线程组也没有设置回调接口,则直接把异常的栈信息定向到 System.err 中
         */
    }
}
```

6.3 注入 Hook 钩子线程

现在很多软件包括 MySQL, Zookeeper, kafka 等都存在 Hook 线程的校验机制，目的是校验进程是否已启动,防止重复启动程序.

Hook 线程也称为钩子线程，当 JVM 退出的时候会执行 Hook 线程. 经常在程序启动时创建一个.lock 文件，用.lock 文件校验程序是否启

动,在程序退出(JVM 退出)时删除该.lock 文件，在 Hook 线程中除了防止重新启动进程外,还可以做资源释放，尽量避免在 Hook 线程中进行复杂的操作.

```
package com.wkcto.hook;

import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.concurrent.TimeUnit;

/**
 * 通过 Hook 线程防止程序重复启动
 */
public class Test {
    public static void main(String[] args) {
        //1)注入 Hook 线程,在程序退出时删除.lock 文件
        Runtime.getRuntime().addShutdownHook(new Thread(){
            @Override
            public void run() {
                System.out.println("JVM 退出,会启动当前 Hook 线程,在 Hook 线程中删除.lock 文件");
                getLockFile().toFile().delete();
            }
        });

        //2)程序运行时,检查 lock 文件是否存在,如果 lock 文件存在,则抛出异常
        if ( getLockFile().toFile().exists()){
            throw  new RuntimeException("程序已启动");
        }else {       //文件不存在,说明程序是第一次启动,创建 lock 文件
            try {
                getLockFile().toFile().createNewFile();
```

```
                System.out.println("程序在启动时创建了 lock 文件");
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        //模拟程序运行
        for (int i = 0; i < 10; i++) {
            System.out.println("程序正在运行");
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private static Path getLockFile(){
        return Paths.get("", "tmp.lock");
    }
}
```

6.4 线程池

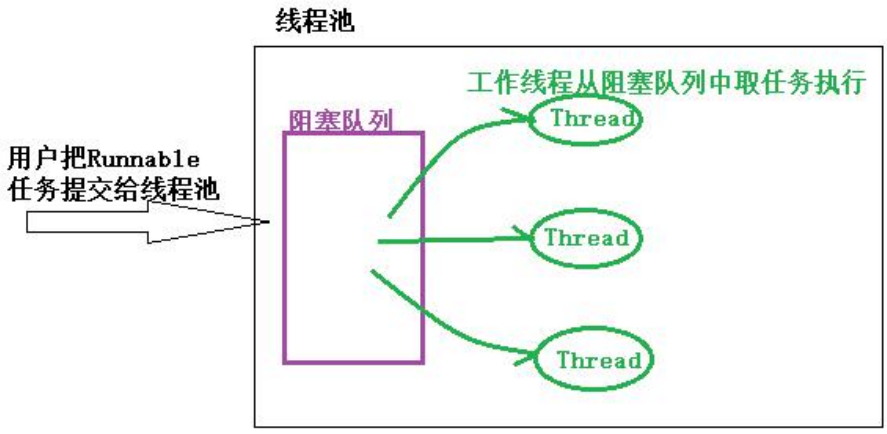
6.4.1 什么是线程池

可以以 new Thread(() -> { 线程执行的任务 }).start(); 这种形式开启一个线程. 当 run()方法运行结束,线程对象会被 GC 释放.

在真实的生产环境中,可能需要很多线程来支撑整个应用,当线程

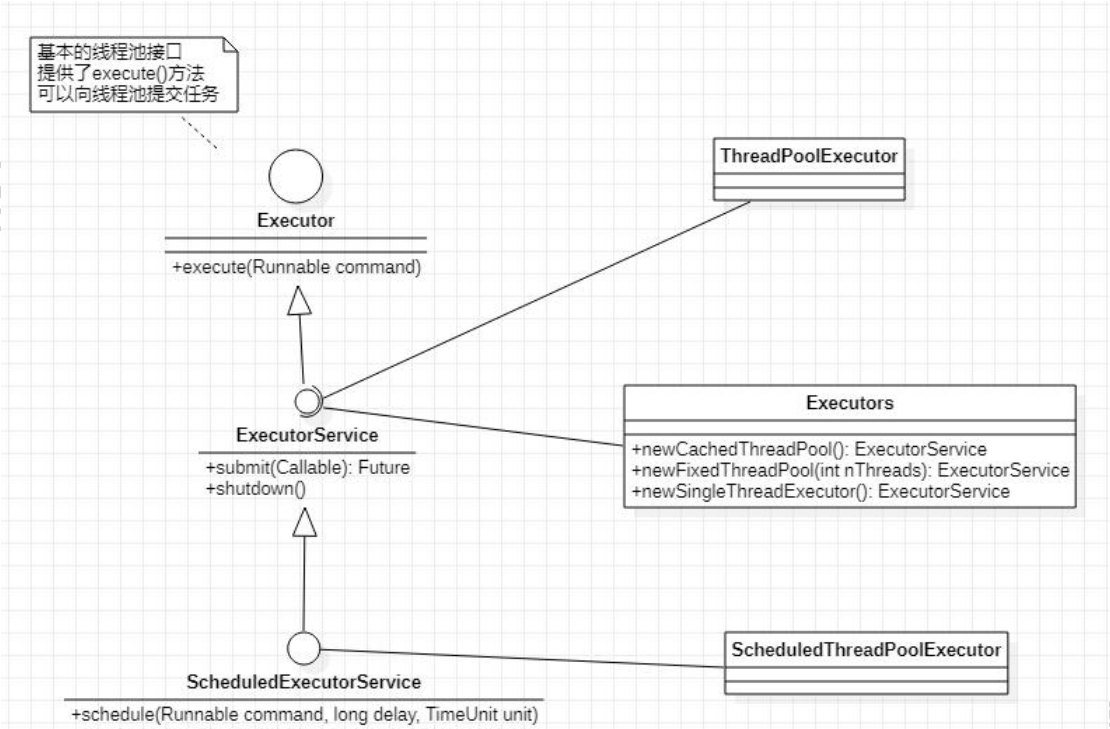
数量非常多时 ,反而会耗尽 CPU 资源. 如果不对线程进行控制与管理,反而会影响程序的性能. 线程开销主要包括: 创建与启动线程的开销;线程销毁开销; 线程调度的开销; 线程数量受限 CPU 处理器数量.

线程池就是有效使用线程的一种常用方式. 线程池内部可以预先创建一定数量的工作线程,客户端代码直接将任务作为一个对象提交给线程池, 线程池将这些任务缓存在工作队列中, 线程池中的工作线程不断地从队列中取出任务并执行.



6.4.2 JDK 对线程池的支持

JDK 提供了一套 Executor 框架,可以帮助开发人员有效的使用线程池




```
package com.wkcto.threadpool;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 线程池的基本使用
 */
public class Test01 {
    public static void main(String[] args) {

        //创建有 5 个线程大小的线程池,
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(5);

        //向线程池中提交 18 个任务,这 18 个任务存储到线程池的阻塞队列中，线程池中这
5 个线程就从阻塞队列中取任务执行
        for (int i = 0; i < 18; i++) {
            fixedThreadPool.execute(new Runnable() {
                @Override
                public void run() {
```

```
                    System.out.println(Thread.currentThread().getId() + " 编号的任务在执
行任务,开始时间: " + System.currentTimeMillis());
                    try {
                        Thread.sleep(3000);        //模拟任务执行时长
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }

    }
}
```

```
package com.wkcto.threadpool;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

/**
 * 线程池的计划任务
 */
public class Test02 {
    public static void main(String[] args) {
        //创建一个有调度功能的线程池
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(10);

        //在延迟 2 秒后执行任务, schedule( Runnable 任务，延迟时长，时间单位)
        scheduledExecutorService.schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getId() + " -- " +
System.currentTimeMillis() );
            }
```

```
        }, 2, TimeUnit.SECONDS);

        //以固定的频率执行任务,开启任务的时间是固定的，在3秒后执行任务,以后每隔5秒重新执行一次
        /*        scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getId() + "----在固定频率开启任务---" + System.currentTimeMillis());
                try {
                    TimeUnit.SECONDS.sleep(3);  //睡眠模拟任务执行时间 ,如果任务执行时长超过了时间间隔,则任务完成后立即开启下个任务
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, 3, 2, TimeUnit.SECONDS);*/

//在上次任务结束后,在固定延迟后再次执行该任务,不管执行任务耗时多长,总是在任务结束后的 2 秒再次开启新的任务
        scheduledExecutorService.scheduleWithFixedDelay(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getId() + "----在固定频率开启任务---" + System.currentTimeMillis());
                try {
                    TimeUnit.SECONDS.sleep(3);  //睡眠模拟任务执行时间 ,如果任务执行时长超过了时间间隔,则任务完成后立即开启下个任务
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                    }
                }
            }, 3, 2, TimeUnit.SECONDS);

        }
}
```

6.4.3 核心线程池的底层实现

查看 Executors 工具类中 newCachedThreadPool(), newSingleThreadExcecutor(), newFixedThreadPool()源码:

```
    public static ExecutorService newCachedThreadPool() {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>());
    }
```

该线程池在极端情况下,每次提交新的任务都会创建新的线程执行. 适合用来执行大量耗时短并且提交频繁的任务

```
    public static ExecutorService newFixedThreadPool(int nThreads) {
        return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>());
    }
```

```
    public static ExecutorService newSingleThreadExecutor() {
        return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1,
                                    0L, TimeUnit.MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>()));
    }
```

Excutors 工具类中返回线程池的方法底层都使用了 ThreadPoolExecutor 线程池,这些方法都是 ThreadPoolExecutor 线程池的封装.

ThreadPoolExecutor 的构造方法:

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

各个参数含义:

corePoolSize, 指定线程池中核心线程的数量

maxinumPoolSize,指定线程池中最大线程数量

keepAliveTime,当线程池线程的数量超过 corePoolSize 时,多余的空闲线程的存活时长,即空闲线程在多长时长内销毁

unit, 是 keepAliveTime 时长单位

workQueue,任务队列,把任务提交到该任务队列中等待执行

threadFactory,线程工厂,用于创建线程

handler 拒绝策略,当任务太多来不及处理时,如何拒绝

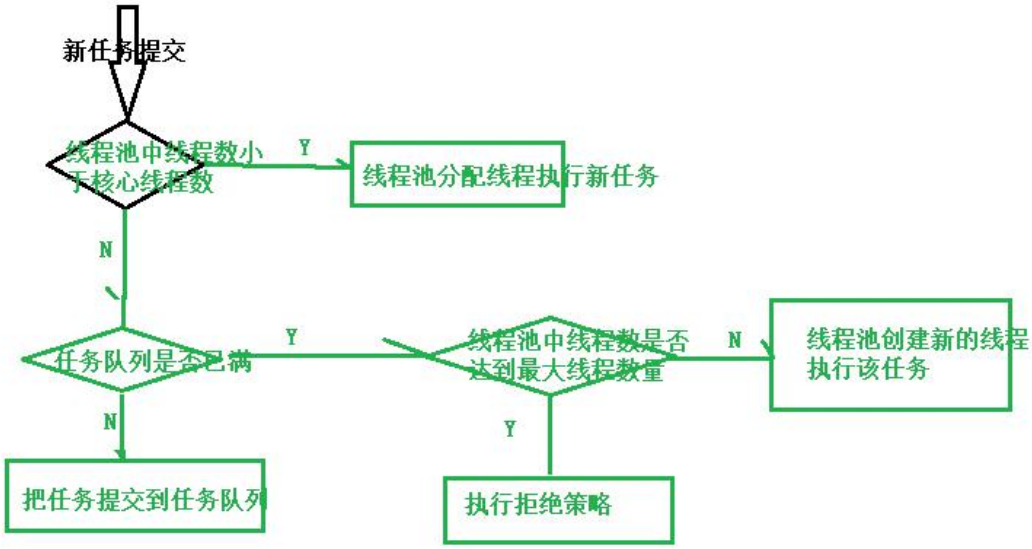
说明:

workQueue 工作队列是指提交未执行的任务队列,它是 BlockingQueue 接口的对象,仅用于存储 Runnable 任务.根据队列功能分类,在 ThreadPoolExecutor 构造方法中可以使用以下几种阻塞队列:

1) 直接提交队列,由 SynchronousQueue 对象提供,该队列没有容量,提交给线程池的任务不会被真实的保存,总是将新的任务提

交给线程执行,如果没有空闲线程,则尝试创建新的线程,如果线程数量已经达到 maxinumPoolSize 规定的最大值则执行拒绝策略.

2) 有界任务队列,由 ArrayBlockingQueue 实现,在创建 ArrayBlockingQueue 对象时,可以指定一个容量. 当有任务需要执行时,如果线程池中线程数小于 corePoolSize 核心线程数则创建新的线程;如果大于 corePoolSize 核心线程数则加入等待队列.如果队列已满则无法加入,在线程数小于 maxinumPoolSize 指定的最大线程数前提下会创建新的线程来执行,如果线程数大于 maxinumPoolSize 最大线程数则执行拒绝策略



3) 无界任务队列,由 LinkedBlockingQueue 对象实现,与有界队列相比,除非系统资源耗尽,否则无界队列不存在任务入队失败的情况. 当有新的任务时,在系统线程数小于 corePoolSize 核心线程数则创建新的线程来执行任务;当线程池中线程数量大于

corePoolSize 核心线程数则把任务加入阻塞队列

4) 优先任务队列是通过 PriorityBlockingQueue 实现的,是带有任务优先级的队列,是一个特殊的无界队列.不管是 ArrayBlockingQueue 队列还是 LinkedBlockingQueue 队列都是按照先进先出算法处理任务的.在 PriorityBlockingQueue 队列中可以根据任务优先级顺序先后执行.

6.4.4 拒绝策略

ThreadPoolExecutor 构造方法的最后一个参数指定了拒绝策略.当提交给线程池的任务量超过实际承载能力时,如何处理? 即线程池中的线程已经用完了,等待队列也满了,无法为新提交的任务服务,可以通过拒绝策略来处理这个问题. JDK 提供了四种拒绝策略:

AbortPolicy 策略,会抛出异常

CallerRunsPolicy 策略,只要线程池没关闭,会在调用者线程中运行当前被丢弃的任务

DiscardOldestPolicy 将任务队列中最老的任务丢弃,尝试再次提交新任务

DiscardPolicy 直接丢弃这个无法处理的任务

Executors 工具类提供的静态方法返回的线程池默认的拒绝策略是 AbortPolicy 抛出异常,如果内置的拒绝策略无法满足实际需求,可以扩

展 RejectedExecutionHandler 接口

```
package com.wkcto.threadpool;

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

/**
 * 自定义拒绝策略
 */
public class Test03 {
    public static void main(String[] args) {
        //定义任务
        Runnable r = new Runnable() {
            @Override
            public void run() {
                int num = new Random().nextInt(5);
                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，自定义拒绝策略
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5, 5, 0, TimeUnit.SECONDS, new LinkedBlockingQueue<>(10), Executors.defaultThreadFactory(), new RejectedExecutionHandler(){
            @Override
            public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
                //r 就是请求的任务, executor 就是当前线程池
                System.out.println(r + " is discarding..");
            }
        });
```

```
        //向线程池提交若干任务
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            threadPoolExecutor.submit(r);
        }
    }
}
```