# 线程池

## 1.预定义好的线程池

### 1.1.适合长任务的线程池CachedThreadPool

特点：

1.没有核心线程全部都是临时线程。

2.临时线程的数量是Integer.MAX\_VALUE，即2^31-1考虑到单台服务器所能承载的线程数量远远小于21亿，所以一般认为这个线程池能够处理无限多的请求。

3.临时线程用完之后最多存活60s。

4.工作队列是一个同步队列，实际生产过程中，一般在测试阶段就会利用空请求将这个工作队列填充，此时可以认为这个线程池没有工作队列。

5.大池子小队列，适用于高并发的短任务的场景，例如即时通讯，不适用于长任务场景。

package cn.tedu.threadpool;

import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class ExecutorServiceDemo {  
  
 public static void main(String[] args) {  
  
 ExecutorService cachedThreadPool =  
 Executors.*newCachedThreadPool*();  
  
 }  
}

### 1.2.适合短任务的线程池

特点：

1. 没有临时线程全部都是核心线程

2. 工作队列是LinkedBlockingQueue，默认是Integer.MAX\_VALUE一般认为能够存储无限多的请求

3.小池子大队列，适用于并发低的长任务场景，例如网盘下载，不适用于高并发的短任务场景

package cn.tedu.threadpool;  
  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class ExecutorServiceDemo {  
  
 public static void main(String[] args) {  
  
  
 ExecutorService fixedThreadPool =  
 Executors.*newFixedThreadPool*(5);  
 }  
  
}

## 1.ScheduledExecutorService - 定时调度执行器服务器

这个线程池起到"定时"效果。这个线程池是很多定时器的底层。

### 1.1.测试schedule()

此方法是普通的定时计划，在执行线程之后5秒钟才会执行具体的run()方法，在一下方法执行完hello之后会睡眠等待8秒然后结束当前线程。

package cn.tedu.threadpool.scheduledexecutorservice;

import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.TimeUnit;  
  
public class ScheduledExecutorServiceDemo01 {  
  
 public static void main(String[] args) {  
  
 //创建定时的线程池类，指定5个核心线程  
 ScheduledExecutorService scheduledExecutorService =  
 Executors.*newScheduledThreadPool*(5);  
  
 //传入runnable实现的类对象，  
 scheduledExecutorService  
 .schedule(new ScheduleThread01(),5, TimeUnit.*SECONDS*);  
  
 }  
}  
  
class ScheduleThread01 implements Runnable{  
  
 */\*\*  
 \* 创建一个线程，使用Runnable  
 \* 打印一句话然后线程等待8秒  
 \*/* @Override  
 public void run() {  
  
 try {  
 System.*out*.println("hello");  
 Thread.*sleep*(8000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

### 1.2.测试scheduleAtFixedRate()

具有定时不间断执行的方法，从上一次启动开始计算下一次的启动时间。这里的案例为第一个线程启动后是取时间间隔的最大值，即取类里面的8秒的阻塞，每8秒执行一次“hello”的打印。

package cn.tedu.threadpool.scheduledexecutorservice;

import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \* 测试定时线程池  
 \* scheduleAtFixedRate();  
 \*/*public class ScheduledExecutorServiceDemo02 {  
  
 public static void main(String[] args) {  
  
 //创建定时的线程池类，指定5个核心线程  
 ScheduledExecutorService scheduledExecutorService =  
 Executors.*newScheduledThreadPool*(5);  
  
 /\*  
 从上一次启动，开始计算下一次的启动时间  
 每隔5秒执行一次  
 间隔时间是取执行时间和执行时间的最大值  
 这里是取8秒  
 \*/  
 scheduledExecutorService  
 .scheduleAtFixedRate(new ScheduleThread02(),  
 0,  
 5, TimeUnit.*SECONDS*);  
  
 }  
}  
  
class ScheduleThread02 implements Runnable {  
  
 */\*\*  
 \* 创建一个线程，使用Runnable  
 \* 打印一句话然后线程等待8秒  
 \*/* @Override  
 public void run() {  
  
 try {  
 System.*out*.println("hello");  
 Thread.*sleep*(8000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

### 1.3.测试newScheduledThreadPool()

与上面的方法类似，不同之处在于上面的方法是在上一次线程启动时开始计时，而现在这个方法是在上一次启动线程结束后开始计时，即执行“hello”之后以及执行完类里面8秒的阻塞后再开始计算5秒的延迟，总共13秒的延迟。

package cn.tedu.threadpool.scheduledexecutorservice;

import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \* 测试定时执行线程  
 \* scheduleWithFixedDelay();  
 \*/*public class ScheduledExecutorServiceDemo03 {  
  
 public static void main(String[] args) {  
  
 //创建定时的线程池类，指定5个核心线程  
 ScheduledExecutorService scheduledExecutorService =  
 Executors.*newScheduledThreadPool*(5);  
  
 /\*  
 从上一次结束，开始计算下一次启动时间  
 每隔5s执行一次  
 \*/  
 scheduledExecutorService  
 .scheduleWithFixedDelay(new ScheduleThread02(),  
 0,  
 5, TimeUnit.*SECONDS*);  
  
 }  
}  
  
class ScheduleThread03 implements Runnable {  
  
 */\*\*  
 \* 创建一个线程，使用Runnable  
 \* 打印一句话然后线程等待8秒  
 \*/* @Override  
 public void run() {  
  
 try {  
 System.*out*.println("hello");  
  
 Thread.*sleep*(8000);  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

## 2.ForkJoinPool-分叉合并池

### 2.1.特性

将一个大任务进行拆分，拆分成多个小任务分配给多个线程执行，这个过程称之为分叉， 将分叉出来的任务结果进行汇总，这个过程称之为合并。

分叉合并能够非常有效的提高CPU的利用率。利用大量线程去抢占CPU，但是导致其他的线程被挤占，因此一般是把分叉合并放到凌晨进行。

数据量越小，循环的优势越明显；数据量越大，分叉合并的效率越高。分叉合并为了提高执行效率，会尽量均衡的将任务分配给每一个核。

在分叉合并中，如果一个核将它身上所有的任务执行完成之后，并不空闲下来，而是会去随机扫描一个核，然后从这个核的任务队列尾端来偷取一个任务回来执行，这种方式称之为work-stealing(工作窃取)策略 - 因为工作窃取策略涉及到CPU核之间的线程调度问题，所以底层是用C语言实现的。

### 2.2.测试代码

package cn.tedu.threadpool;

import java.util.concurrent.ExecutionException;  
import java.util.concurrent.ForkJoinPool;  
import java.util.concurrent.ForkJoinTask;  
import java.util.concurrent.RecursiveTask;  
  
public class ForkJoinPoolDemo {  
  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
  
 // 求1-100000000000L的和  
 // 主函数所在的类默认就是一个线程类  
 // ForkJoinPoolDemo是一个线程类  
 // 利用主线程求和，主线程会落在CPU的某一个核上执行  
 // long sum = 0;  
 // for (long i = 1; i <= 100000000000L; i++) {  
 // sum += i;  
 // }  
 // System.out.println(sum);  
 ForkJoinPool p = new ForkJoinPool();  
 ForkJoinTask<Long> f = p.submit(new Sum(1, 100000000000L));  
 System.out.println(f.get());  
 p.shutdown();  
 }  
  
}  
  
class Sum extends RecursiveTask<Long> {  
  
 private long start;  
 private long end;  
  
 public Sum(long start, long end) {  
 this.start = start;  
 this.end = end;  
 }  
  
 // 分叉合并的逻辑就是在这个方法中覆盖  
 @Override  
 protected Long compute() {  
 // 分叉：将大的拆成小的  
 // 拆的过程中得有限度  
 if (end - start <= 10000) {  
 // 如果范围内的数字个数不到10000个，此时认为数量已经比较少了  
 // 那么不能继续拆了，就要将这个范围内的数字来进行求和  
 long sum = 0;  
 for (long i = start; i <= end; i++) {  
 sum += i;  
 }  
 return sum;  
 } else {  
 // 说明范围内的数字依然比较多，那就继续分叉  
 long mid = (start + end) / 2;  
 Sum left = new Sum(start, mid);  
 Sum right = new Sum(mid + 1, end);  
 // 需要将拆分出来这两半各自定义成两个线程来执行  
 left.fork();  
 right.fork();  
 // 分叉完之后，最后需要将结果汇总  
 return left.join() + right.join();  
 }  
 }  
}

## 3.Callable

### 3.1.特性

通过泛型来定义返回结果类型。Callable的使用在实际开发中的场景和Runnable相似，如果需要执行结果那么使用Callable。

### 3.2.Callable和Runnable的区别

|  |  |  |
| --- | --- | --- |
|  | Runnable | Callable |
| 返回值 | 没有返回值 | 通过泛型来定义返回值的类型 |
| 启动方式 | 通过Thread直接启动  通过线程池的execute或者submit方法启动 | 通过FutureTask转化成Runnable之后再通过Thread启动  通过线程池的submit方法启动 |
| 异常处理 | 不允许抛出异常，那么就无法通过全局方式来进行处理 | 允许抛出异常，那么就可以通过全局方法来进行统一处理 |

### 3.3.测试案例

package cn.tedu.threadpool.callable;

import java.util.concurrent.\*;  
  
public class CallableDemo {  
  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
  
 /\*  
 如果想使用Thread直接启动线程则需要将其进行包装，  
 本质上还是使用Callable来传到Thread里面进行执行。  
 \*/  
 // Callable -> FutureTask -> RunnableFuture -> Runnable  
 // FutureTask<String> fs =  
 // new FutureTask<>(new CDemo());  
 // new Thread(fs).start();  
 // System.out.println(fs.get());  
  
 ExecutorService es =  
 Executors.*newCachedThreadPool*();  
 // 将结果封装成了Future对象  
 Future<String> f = es.submit(new CDemo());  
 System.*out*.println(f.get());  
 es.shutdown();  
 }  
  
}  
  
// 泛型表示的结果类型  
class CDemo implements Callable<String> {  
 @Override  
 public String call() throws Exception {  
 return "Success~";  
 }  
}

# 服务器线程承载量计算

## 1.Java内存结构

Stack：计算/执行代码块。栈内存是线程独享的

Heap：存储对象。堆内存是线程共享的

Method Area：存储类信息。方法区是线程共享的

Native Stack：执行本地方法。用native修饰但是没有方法体的方法称之为本地方法，本地方法的方法体是用其他语言来实现的。本地方法栈是线程独享的

Program Counter:对线程来进行计数。当PC计数器对哪一行计数的时候，任务就会执行哪一行。PC计数器是线程独享的。

## 2.承载计算

如果要计算一台服务器的线程承载量，要考虑独享内存的数量：栈内存、本地方法栈、PC计数器。其中，PC计数器只占几个字节大小，所以可以忽略不计；线程执行过程中除非出现本地方法，不然不会占用本地方法栈；栈内存大小是128K~8192K之间。

如果需要估计一台服务器的线程承载量，主要考虑栈内存，一般而言，一台服务器最多能允许将2/3的内存给栈内存使用，市面上主流的服务器内存是在64G~128G。

以128G为准，考虑极端情况，栈内存以128K来计算：

128G/128K\*2/3≈699050

实际开发过程中，一台服务器的线程数量大概是25W左右。