**java线程池使用**

<https://www.jianshu.com/p/edd7cb4eafa0>

在Java1.5中提供了一个非常高效实用的多线程包：java.util.concurrent,提供了大量高级工具，可以帮助开发者编写高效易维护、结构清晰的Java多线程程序。

## 线程池

之前我们在使用多线程都是用Thread的start()来创建启动一个线程，但是在实际开发中，如果每个请求到达就创建一个新线程，开销是相当大的。服务器在创建和销毁线程上花费的时间和消耗的系统资源都相当大，甚至可能要比在处理实际的用请求的时间和资源要多的多。除了创建和销毁线程的开销之外，活动的线程也需要消耗系统资源。如果在一个jvm里创建太多的线程，可能会使系统由于过度消耗内存或“切换过度”而导致系统资源不足。这就引入了线程池概念。

线程池的原理其实就是对多线程的一个管理，为了实现异步机制的一种方法，其实就是多个线程执行多个任务，最终这些线程通过线程池进行管理…不用手动去维护…一次可以处理多个任务，这样就可以迅速的进行相应…比如说一个网站成为了热点网站，那么对于大量的点击量，就必须要对每一次的点击做出迅速的处理，这样才能达到更好的交互效果…这样就需要多个线程去处理这些请求，以便能够更好的提供服务…

在java.util.concurrent包下，提供了一系列与线程池相关的类。合理的使用线程池，可以带来多个好处：

（1） **降低资源消耗**。通过重复利用已创建的线程降低线程创建和销毁造成的消耗；

（2） **提高响应速度**。当任务到达时，任务可以不需要等到线程创建就能立即执行；

（3） **提高线程的可管理性**。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

线程池可以应对突然大爆发量的访问，通过有限个固定线程为大量的操作服务，减少创建和销毁线程所需的时间。

使用线程池：

* 1、创建线程池
* 2、创建任务
* 3、执行任务
* 4、关闭线程池

## 五种线程池的使用场景

* newSingleThreadExecutor：一个单线程的线程池，可以用于需要保证顺序执行的场景，并且只有一个线程在执行。
* newFixedThreadPool：一个固定大小的线程池，可以用于已知并发压力的情况下，对线程数做限制。
* newCachedThreadPool：一个可以无限扩大的线程池，比较适合处理执行时间比较小的任务。
* newScheduledThreadPool：可以延时启动，定时启动的线程池，适用于需要多个后台线程执行周期任务的场景。

newWorkStealingPool：一个拥有多个任务队列的线程池，可以减少连接数，创建当前可用cpu数量的线程来并行执行。

## 创建线程池

一般通过工具类Executors的静态方法来获取线程池或静态方法。介绍四种常用创建方法

ExecutorService service1 = Executors.newSingleThreadExecutor();

说明： **单例线程**，表示在任意的时间段内，线程池中只有一个线程在工作

ExecutorService service2 = Executors.newCacheThreadPool();

说明： **缓存线程池**，先查看线程池中是否有当前执行线程的缓存，如果有就resue(复用),如果没有,那么需要创建一个线程来完成当前的调用.并且这类线程池只能完成一些生存期很短的一些任务.并且这类线程池内部规定能resue(复用)的线程，空闲的时间不能超过60s,一旦超过了60s,就会被移出线程池

ExecutorService service3 = Executors.newFixedThreadPool(10);

说明： **固定型线程池**，和newCacheThreadPool()差不多，也能够实现resue(复用),但是这个池子规定了线程的最大数量，也就是说当池子有空闲时，那么新的任务将会在空闲线程中被执行，一旦线程池内的线程都在进行工作，那么新的任务就必须等待线程池有空闲的时候才能够进入线程池,其他的任务继续排队等待.这类池子没有规定其空闲的时间到底有多长.这一类的池子更适用于服务器.

ExecutorService service4 = Executors.newScheduledThreadPool(10);

说明： **调度型线程池**,调度型线程池会根据Scheduled(任务列表)进行延迟执行，或者是进行周期性的执行.适用于一些周期性的工作.

package com.reapal.brave.main;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

/\*\*

\* Created by jack-cooper on 2017/2/23.

\*/

public class Test {

public static void main(String[] args) {

ExecutorService service = Executors.newCachedThreadPool();

service.submit(new Runnable() {

@Override

public void run() {

while(true){

System.out.println("hello world !");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

});

System.out.println(" ===> main Thread execute here ! " );

}

}

### 自定义线程池

构造函数的定义

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) ;

* corePoolSize  
  线程池大小，决定着新提交的任务是新开线程去执行还是放到任务队列中，也是线程池的最最核心的参数。一般线程池开始时是没有线程的，只有当任务来了并且线程数量小于corePoolSize才会创建线程。
* maximumPoolSize  
  最大线程数，线程池能创建的最大线程数量。
* keepAliveTime  
  在线程数量超过corePoolSize后，多余空闲线程的最大存活时间。
* unit  
  时间单位
* workQueue  
  存放来不及处理的任务的队列，是一个BlockingQueue。
* threadFactory  
  生产线程的工厂类，可以定义线程名，优先级等。
* ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()
* .setNameFormat("demo-pool-%d").build();
* handler  
  拒绝策略，当任务来不及处理的时候，如何处理, 前面有讲解。  
  1、 AbortPolicy -- 当任务添加到线程池中被拒绝时，它将抛出 RejectedExecutionException 异常。  
  2、 CallerRunsPolicy -- 当任务添加到线程池中被拒绝时，会在线程池当前正在运行的Thread线程池中处理被拒绝的任务。  
  3、 DiscardOldestPolicy -- 当任务添加到线程池中被拒绝时，线程池会放弃等待队列中最旧的未处理任务，然后将被拒绝的任务添加到等待队列中。  
  4、 DiscardPolicy -- 当任务添加到线程池中被拒绝时，线程池将丢弃被拒绝 的任务。  
  参考： [http://www.cnblogs.com/skywang12345/p/3512947.html](https://links.jianshu.com/go?to=http%3A%2F%2Fwww.cnblogs.com%2Fskywang12345%2Fp%2F3512947.html)

例子

ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()

.setNameFormat("demo-pool-%d").build();

ExecutorService pool = new ThreadPoolExecutor(5, 200,0L, TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>(1024), namedThreadFactory, new ThreadPoolExecutor.CallerRunsPolicy());

for(int i=0;i<1000;i++) {

pool.execute(() -> {

//todo 此处写业务

});

}

## 创建任务

任务分为两种:一种是有返回值的（ **callable** ），一种是没有返回值的（ **runnable** ）. Callable与 Future 两功能是Java在后续版本中为了适应多并法才加入的，Callable是类似于Runnable的接口，实现Callable接口的类和实现Runnable的类都是可被其他线程执行的任务。

* 无返回值的任务就是一个实现了runnable接口的类.使用run方法.
* 有返回值的任务是一个实现了callable接口的类.使用call方法.

### Callable和Runnable的区别如下：

* Callable定义的方法是call，而Runnable定义的方法是run。
* Callable的call方法可以有返回值，而Runnable的run方法不能有返回值。
* Callable的call方法可抛出异常，而Runnable的run方法不能抛出异常。

#### Future 介绍

Future表示异步计算的结果，它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。Future的cancel方法可以取消任务的执行，它有一布尔参数，参数为 true 表示立即中断任务的执行，参数为 false 表示允许正在运行的任务运行完成。Future的 get 方法等待计算完成，获取计算结果。

package com.reapal.brave.main;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class CallableAndFuture {

public static class MyCallable implements Callable{

private int flag = 0;

public MyCallable(int flag){

this.flag = flag;

}

public String call() throws Exception{

if (this.flag == 0){

return "flag = 0";

}

if (this.flag == 1){

try {

while (true) {

System.out.println("looping.");

Thread.sleep(2000);

}

} catch (InterruptedException e) {

System.out.println("Interrupted");

}

return "false";

} else {

throw new Exception("Bad flag value!");

}

}

}

public static void main(String[] args) {

// 定义3个Callable类型的任务

MyCallable task1 = new MyCallable(0);

MyCallable task2 = new MyCallable(1);

MyCallable task3 = new MyCallable(2);

// 创建一个执行任务的服务

ExecutorService es = Executors.newFixedThreadPool(3);

try {

// 提交并执行任务，任务启动时返回了一个Future对象，

// 如果想得到任务执行的结果或者是异常可对这个Future对象进行操作

Future future1 = es.submit(task1);

// 获得第一个任务的结果，如果调用get方法，当前线程会等待任务执行完毕后才往下执行

System.out.println("task1: " + future1.get());

Future future2 = es.submit(task2);

// 等待5秒后，再停止第二个任务。因为第二个任务进行的是无限循环

Thread.sleep(5000);

System.out.println("task2 cancel: " + future2.cancel(true));

// 获取第三个任务的输出，因为执行第三个任务会引起异常

// 所以下面的语句将引起异常的抛出

Future future3 = es.submit(task3);

System.out.println("task3: " + future3.get());

} catch (Exception e){

System.out.println(e.toString());

}

// 停止任务执行服务

es.shutdownNow();

}

}

## 执行任务

通过java.util.concurrent.ExecutorService接口对象来执行任务，该对象有两个方法可以执行任务execute和submit。execute这种方式提交没有返回值，也就不能判断是否执行成功。submit这种方式它会返回一个Future对象，通过future的get方法来获取返回值，get方法会阻塞住直到任务完成。

execute与submit区别：

* 接收的参数不一样
* submit有返回值，而execute没有
* submit方便Exception处理
* execute是Executor接口中唯一定义的方法；submit是ExecutorService（该接口继承Executor）中定义的方法

## 关闭线程池

线程池使用完毕，需要对其进行关闭，有两种方法

shutdown()

说明：shutdown并不是直接关闭线程池，而是不再接受新的任务…如果线程池内有任务，那么把这些任务执行完毕后，关闭线程池

shutdownNow()

说明：这个方法表示不再接受新的任务，并把任务队列中的任务直接移出掉，如果有正在执行的，尝试进行停止

## **综合使用案例(FutureTask)**

import java.util.concurrent.\*;

/\*\*

\* Author : Slogen

\* AddTime : 17/6/4

\* Email : huangjian13@meituan.com

\*/

public class CallDemo {

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

/\*\*

\* 第一种方式:Future + ExecutorService

\* Task task = new Task();

\* ExecutorService service = Executors.newCachedThreadPool();

\* Future<Integer> future = service.submit(task1);

\* service.shutdown();

\*/

/\*\*

\* 第二种方式: FutureTask + ExecutorService

\* ExecutorService executor = Executors.newCachedThreadPool();

\* Task task = new Task();

\* FutureTask<Integer> futureTask = new FutureTask<Integer>(task);

\* executor.submit(futureTask);

\* executor.shutdown();

\*/

/\*\*

\* 第三种方式:FutureTask + Thread

\*/

// 2. 新建FutureTask,需要一个实现了Callable接口的类的实例作为构造函数参数

FutureTask<Integer> futureTask = new FutureTask<Integer>(new Task());

// 3. 新建Thread对象并启动

Thread thread = new Thread(futureTask);

thread.setName("Task thread");

thread.start();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Thread [" + Thread.currentThread().getName() + "] is running");

// 4. 调用isDone()判断任务是否结束

if(!futureTask.isDone()) {

System.out.println("Task is not done");

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

int result = 0;

try {

// 5. 调用get()方法获取任务结果,如果任务没有执行完成则阻塞等待

result = futureTask.get();

} catch (Exception e) {

e.printStackTrace();

}

System.out.println("result is " + result);

}

// 1. 继承Callable接口,实现call()方法,泛型参数为要返回的类型

static class Task implements Callable<Integer> {

@Override

public Integer call() throws Exception {

System.out.println("Thread [" + Thread.currentThread().getName() + "] is running");

int result = 0;

for(int i = 0; i < 100;++i) {

result += i;

}

Thread.sleep(3000);

return result;

}

}

}

## 综合使用案例一

需求：从数据库中获取url，并利用httpclient循环访问url地址，并对返回结果进行操作

分析：由于是循环的对多个url进行访问并获取数据，为了执行的效率，考虑使用多线程，url数量未知如果每个任务都创建一个线程将消耗大量的系统资源，最后决定使用线程池。

public class GetMonitorDataService {

private Logger logger = LoggerFactory.getLogger(GetMonitorDataService.class);

@Resource

private MonitorProjectUrlMapper groupUrlMapper;

@Resource

private MonitorDetailBatchInsertMapper monitorDetailBatchInsertMapper;

public void sendData(){

//调用dao查询所有url

MonitorProjectUrlExample example=new MonitorProjectUrlExample();

List<MonitorProjectUrl> list=groupUrlMapper.selectByExample(example);

logger.info("此次查询数据库中监控url个数为"+list.size());

//获取系统处理器个数，作为线程池数量

int nThreads=Runtime.getRuntime().availableProcessors();

//定义一个装载多线程返回值的集合

List<MonitorDetail> result= Collections.synchronizedList(new ArrayList<MonitorDetail>());

//创建线程池，这里定义了一个创建线程池的工具类，避免了创建多个线程池，ThreadPoolFactoryUtil可以使用单例模式设计

ExecutorService executorService = ThreadPoolFactoryUtil.getExecutorService(nThreads);

//遍历数据库取出的url

if(list!=null&&list.size()>0) {

for (MonitorProjectUrl monitorProjectUrl : list) {

String url = monitorProjectUrl.getMonitorUrl();

//创建任务

ThreadTask threadTask = new ThreadTask(url, result);

//执行任务

executorService.execute(threadTask);

try {//等待直到所有任务完成

executorService.awaitTermination(Long.MAX\_VALUE, TimeUnit.MINUTES);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

//注意区分shutdownNow

executorService.shutdown();

//对数据进行操作

saveData(result);

}

}

任务

public class ThreadTask implements Runnable{

//这里实现runnable接口

private String url;

private List<MonitorDetail> list;

public ThreadTask(String url,List<MonitorDetail> list){

this.url=url;

this.list=list;

}

//把获取的数据进行处理

@Override

public void run() {

MonitorDetail detail = HttpClientUtil.send(url, MonitorDetail.class);

list.add(detail);

}

}

## 综合使用案例二(countDownLatch)

package com.br.lucky.utils;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.\*;

/\*\*

\* @author 10400

\* @create 2018-04-19 20:38

\*/

public class FatureTest {

//1、配置线程池

private static ExecutorService es = Executors.newFixedThreadPool(20);

//2、封装响应Feature

class BizResult{

public String orderId;

public String data;

public String getOrderId() {

return orderId;

}

public void setOrderId(String orderId) {

this.orderId = orderId;

}

public String getData() {

return data;

}

public void setData(String data) {

this.data = data;

}

}

//3、实现Callable接口

class BizTask implements Callable {

private String orderId;

private Object data;

//可以用其他方式

private CountDownLatch countDownLatch;

public BizTask(String orderId, Object data, CountDownLatch countDownLatch) {

this.orderId = orderId;

this.data = data;

this.countDownLatch = countDownLatch;

}

@Override

public Object call() {

try {

//todo business

System.out.println("当前线程Id = " + this.orderId);

BizResult br = new BizResult();

br.setOrderId(this.orderId);

br.setData("some key about your business" + this.getClass());

return br;

}catch (Exception e){

e.printStackTrace();

}finally {

//线程结束时，将计时器减一

countDownLatch.countDown();

}

return null;

}

}

/\*\*

\* 业务逻辑入口

\*/

public List<Future> beginBusiness() throws InterruptedException {

//模拟批量业务数据

List<String> list = new ArrayList<>();

for (int i = 0 ; i < 1000 ; i++) {

list.add(String.valueOf(i));

}

//设置计数器

CountDownLatch countDownLatch = new CountDownLatch(list.size());

//接收多线程响应结果

List<Future> resultList = new ArrayList<>();

//begin thread

for( int i = 0 ,size = list.size() ; i<size; i++){

//todo something befor thread

resultList.add(es.submit(new BizTask(list.get(i), null, countDownLatch)));

}

//wait finish

countDownLatch.await();

return resultList;

}

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

FatureTest ft = new FatureTest();

List<Future> futures = ft.beginBusiness();

System.out.println("futures.size() = " + futures.size());

//todo some operate

System.out.println(" ==========================end========================= " );

}

}

## 综合使用案例三(future.get())

package com.br.lucky.utils;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.\*;

/\*\*

\* @author 10400

\* @create 2018-04-19 20:38

\*/

public class FatureTest {

/\*\*

ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()

.setNameFormat("demo-pool-%d").build();

ExecutorService pool = new ThreadPoolExecutor(5, 200,0L, TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>(1024), namedThreadFactory, new ThreadPoolExecutor.CallerRunsPolicy());

for(int i=0;i<1000;i++) {

pool.execute(() -> {

//todo 业务逻辑在此

});

}

\*/

//1、配置线程池

private static ExecutorService es = Executors.newFixedThreadPool(20);

//2、封装响应Feature

class BizResult{

public String orderId;

public String data;

public String getOrderId() {

return orderId;

}

public void setOrderId(String orderId) {

this.orderId = orderId;

}

public String getData() {

return data;

}

public void setData(String data) {

this.data = data;

}

}

//3、实现Callable接口

class BizTask implements Callable {

private String orderId;

private Object data;

public BizTask(String orderId, Object data) {

this.orderId = orderId;

this.data = data;

}

@Override

public Object call() {

try {

//todo business

System.out.println("当前线程Id = " + this.orderId);

BizResult br = new BizResult();

br.setOrderId(this.orderId);

br.setData("some key about your business" + this.getClass());

Thread.sleep(3000);

return br;

}catch (Exception e){

e.printStackTrace();

}

return null;

}

}

/\*\*

\* 业务逻辑入口

\*/

public List<Future> beginBusiness() throws InterruptedException, ExecutionException {

//模拟批量业务数据

List<String> list = new ArrayList<>();

for (int i = 0 ; i < 100 ; i++) {

list.add(String.valueOf(i));

}

//接收多线程响应结果

List<Future> resultList = new ArrayList<>();

//begin thread

for( int i = 0 ,size = list.size() ; i<size; i++){

//todo something befor thread

Future future = es.submit(new BizTask(list.get(i), null));

resultList.add(future);

}

for (Future f : resultList) {

f.get();

}

System.out.println(" =====多线程执行结束====== ");

//wait finish

return resultList;

}

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

FatureTest ft = new FatureTest();

List<Future> futures = ft.beginBusiness();

System.out.println("futures.size() = " + futures.size());

//todo some operate

System.out.println(" ==========================end========================= " );

}

}