# Java工程师必备利器——并发框架的应用

火车票

网上购物

# 1．课程介绍

* 什么是并发
* 并发的优缺点
* Java线程
* executors线程池框架 Future executor（四种方式）

# 2.什么是并发

* 并发当有多个线程在操作时,如果系统只有一个CPU,则它根本不可能真正同时进行一个以上的线程，它只能把CPU运行时间划分成若干个时间段,再将时间 段分配给各个线程执行，在一个时间段的线程代码运行时，其它线程处于挂起状。.这种方式我们称之为并发(Concurrent)。
* 并行

同学们如果需往期视频资料，加西施老师QQ：295700008

## 2.1 Java并发性和多线程介绍

* 单CPU单任务
* 单CPU多任务
* 多CPU多任务

## 2.2多线程的优点

* 资源利用率更好
* 程序设计更简单
* 程序响应更快

异步：

同步：

## 2.3多线程的代价

* 设计更复杂（底层）
* 上下文切换的开销
* 增加资源消耗

软件

硬件

Java中怎么应用线程

Thread:

1. 就绪状态

Runnable：

1. 可运行状态
2. 可以实现资源共享

Servlet中的线程的不安全性

线程池的优点：

简化程序

提高效率

[java.util.concurrent](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/package-frame.html" \t "http://tool.oschina.net/uploads/apidocs/jdk-zh/packageFrame)

**Future**

1. 利用get获取计算结果
2. 可取消性(当运行结束后就不能取消)

## FutureTask

FutureTask(Callable<V> callable)

创建一个 FutureTask，一旦运行就执行给定的 Callable。

FutureTask(Runnable runnable, V result) 适配器模式

创建一个 FutureTask，一旦运行就执行给定的 Runnable，并安排成功完成时 get 返回给定的结果 。

## 3 Executor框架

* new Thread()的缺点

每次new Thread()耗费性能

调用new Thread()创建的线程缺乏管理，被称为野线程，而且可以无限制创建，之间相互竞争，会导致过多占用系统资源导致系统瘫痪。

不利于扩展，比如如定时执行、定期执行、线程中断

* 采用线程池的优点

重用存在的线程，减少对象创建、消亡的开销，性能佳可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞，提供定时执行、定期执行、单线程、并发数控制等功能。

ExecutorService 用它创建线程池

**[isShutdown](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ExecutorService.html" \l "isShutdown())**()

shutdown()

isTerminated()

Submit：执行任务并返回结果

**[execute](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/Executor.html" \l "execute(java.lang.Runnable))**:接受任务

## BlockingDeque

* Eexecutor作为灵活且强大的异步执行框架，其支持多种不同类型的任务执行策略，提供了一种标准的方法将任务的提交过程和执行过程解耦开发，基于生产者-消费者模式，其提交任务的线程相当于生产者，执行任务的线程相当于消费者，并用Runnable来表示任务，Executor的实现还提供了对生命周期的支持，以及统计信息收集，应用程序管理机制和性能监视等机制。
* Executor框架包括：线程池，Executor，Executors，ExecutorService，CompletionService，Future，Callable等。

Executor 接口定义了最基本的 execute 方法，用于接收用户提交任务。

ExecutorService 定义了线程池终止shutdown及submit提交 futureTask 任务支持的方法。（submit方法提交的是Runnable或Callable，返回了一个Future可引用的实例，可以认为是FutureTask的实例）。

AbstractExecutorService 是抽象类，提供ExecutorService 执行方法的默认实现。其内部实现的submit方法描述为：submit(Runnable) 的实现创建了一个关联RunnableFuture 类，该类将被执行并返回。子类可以重写 newTaskFor 方法，以返回 FutureTask 之外的RunnableFuture 实现。

ThreadPoolExecutor 是最核心的一个类，是线程池的内部实现。线程池的功能都在这里实现了，平时用的最多的基本就是这个了。其源码很精练，远没当时想象的多。

Executors工厂实现：

[Executors.newCachedThreadPool()](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/Executors.html" \l "newCachedThreadPool())（无界线程池，可以进行自动线程回收）

[Executors.newFixedThreadPool(int)](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/Executors.html" \l "newFixedThreadPool(int))（固定大小线程池）

[Executors.newSingleThreadExecutor()](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/Executors.html" \l "newSingleThreadExecutor())（单个后台线程）

### newSingleThreadScheduledExecutor（可调度的执行）

Worker ThreadPool

Runnable Callable

public class WorkerThreadDemo implements Runnable {

private String info;

public WorkerThreadDemo(String info){

this.info=info;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName()+" started. Information is:"+this.info);

try{

Thread.sleep(1000);

}catch(Exception e){

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+" finished.");

}

@Override

public String toString() {

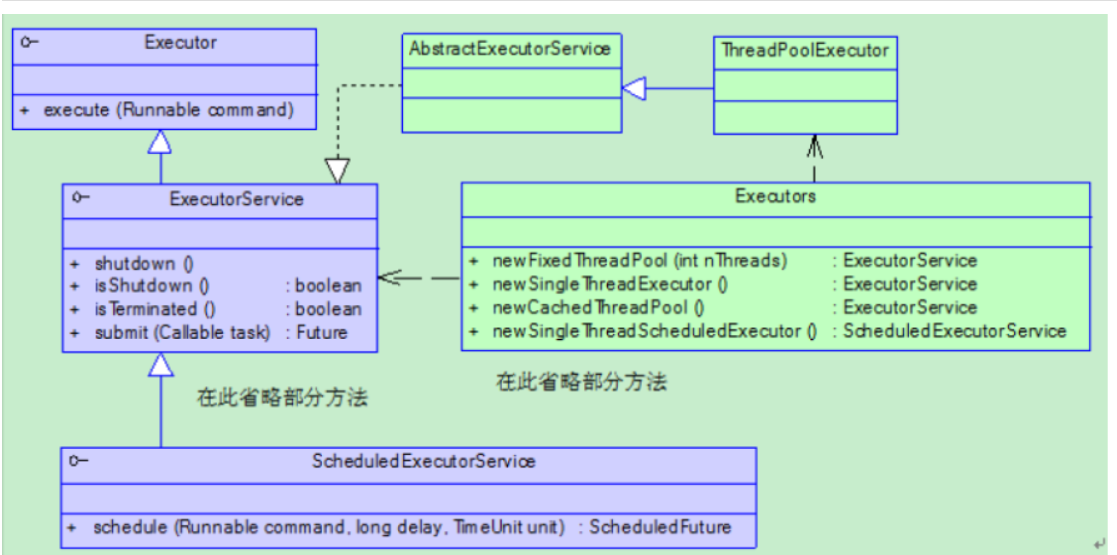
return this.info;

}

}

ScheduledExecutorService接口具有为预定执行或重复执行任务而设计的方法。可以预定Runnable或Callable任务在初始的延迟之后只运行一次，也可以预定一个Runnable对象周期性的运行。

ScheduledThreadPoolExecutor 在 ThreadPoolExecutor 的基础上提供了支持定时调度的功能。线程任务可以在一定延时时间后才被触发执行。



* **Executor** 执行器接口，该接口定义执行Runnable任务的方式。
* **ExecutorService** 该接口定义提供对Executor的服务。
* **ScheduledExecutorService** 定时调度接口。
* **AbstractExecutorService** 执行框架抽象类。
* **ThreadPoolExecutor** JDK中线程池的具体实现。
* **Executors** 线程池工厂类。

Executors方法介绍

Executors工厂类

通过Executors提供四种线程池，

* newFixedThreadPool
* newCachedThreadPool
* newSingleThreadExecutor
* newScheduledThreadPool

Work线程：

public class WorkThread implements Runnable{

private String command;

public WorkThread(String s){

this.command=s;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName()+" start.command="+command);

processCommand();

System.out.println(Thread.currentThread().getName()+" end.");

}

private void processCommand() {

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public String toString() {

return this.command;

}

}

### **1.public static ExecutorService newFixedThreadPool(int nThreads)**

创建固定数目线程的线程池。

public static void main(String[] args) {

ExecutorService pool= Executors.newFixedThreadPool(5);

for (int i = 0; i <10 ; i++) {

Runnable worker=new WorkThread("Thread"+i);

pool.execute(worker);

}

pool.shutdown();

while (!pool.isTerminated()){

}

System.out.println("Thread is finished");

}

### **2.public static ExecutorService newCachedThreadPool()**

创建一个可缓存的线程池，调用execute将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线 程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

public static void main(String[] args) {

ExecutorService pool=Executors.newCachedThreadPool();

for (int i = 0; i < 10; i++) {

Runnable worker=new WorkThread("Thread "+i);

pool.execute(worker);

}

pool.shutdown();

while(!pool.isTerminated()){

}

System.out.println("Finished all Thread!");

}

### **3.public static ExecutorService newSingleThreadExecutor()**

创建一个单线程化的Executor。如果该线程因为异常而结束就新建一条线程来继续执行后续的任务

public static void main(String[] args) {

ExecutorService pool=Executors.newSingleThreadScheduledExecutor();

for (int i = 0; i <10; i++) {

Runnable worker=new WorkThread("Thread "+i);

pool.execute(worker);

}

pool.shutdown();

while(!pool.isTerminated()){

}

System.out.println("Finished all Thread ");

}

### **4.public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)**

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

public static void main(String[] args) {

ScheduledExecutorService pool=Executors.newScheduledThreadPool(3);

for (int i = 0; i <10 ; i++) {

Runnable worker=new WorkThread("Thread "+i);

pool.execute(worker);

pool.schedule(worker,10,TimeUnit.MILLISECONDS);

}

pool.shutdown();

while(!pool.isTerminated()){

}

System.out.println("Finished all Thread.");

}

### **5.ScheduledExecutorService pool=Executors.newSingleThreadScheduledExecutor();**

public static void main(String[] args) {

ScheduledExecutorService pool=Executors.newSingleThreadScheduledExecutor();

for (int i = 0; i < 10; i++) {

Runnable worker=new WorkThread("Thread "+i);

pool.execute(worker);

}

pool.shutdown();

while(!pool.isTerminated()){

}

System.out.println("Finished all Thread!");

}

案例：自定义连接池

public static void main(String[] args) {

BlockingQueue queue=new ArrayBlockingQueue(10);

ThreadPoolExecutor pool=new ThreadPoolExecutor(3,5,2,TimeUnit.MILLISECONDS,queue);

for (int i = 0; i < 20; i++) {

Runnable worker=new WorkThread("Thread "+i);

pool.execute(worker);

}

pool.shutdown();

while (!pool.isTerminated()){

}

System.out.println("Finished all Thread");

}