



人生是一场修行

博客园 首页 新随笔 订阅 管理

由浅入深理解Java线程池及线程池的如何使用

前言

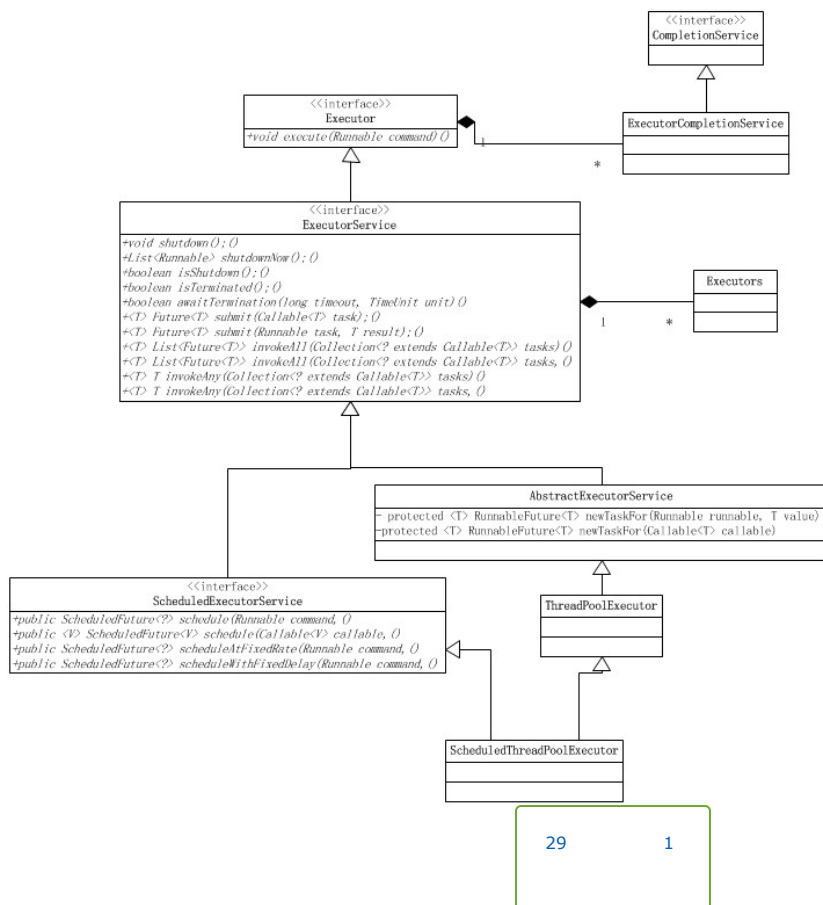
多线程的异步执行方式，虽然能够最大限度发挥多核计算机的计算能力，但是如果不加控制，反而会对系统造成负担。线程本身也要占用内存空间，大量的线程会占用内存资源并且可能会导致Out of Memory。即便没有这样的情况，大量的线程回收也会给GC带来很大的压力。

为了避免重复的创建线程，线程池的出现可以让线程进行复用。通俗点讲，当有工作来，就会向线程池拿一个线程，当工作完成后，并不是直接关闭线程，而是将这个线程归还给线程池供其他任务使用。

接下来从总体到细致的方式，来共同探讨线程池。

总体的架构

来看Executor的框架图：



接口：Executor, CompletionService, ExecutorService, ScheduledExecutorService

公告

昵称: Janti
园龄: 1年6个月
粉丝: 90
关注: 14
+加关注

随笔分类

- 1 Java基础(7)
- 1.1 +----集合
- 1.2 +----多线程(5)
- 1.3 +----网络与IO(4)
- 2 框架(1)
- 2.1 +----springboot(6)
- 2.2 +----Netty
- 2.3 +----Thrift
- 3 缓存redis与数据库
- 4 Java虚拟机(4)
- 5 分布式(4)
- 6 项目经验(4)

积分与排名

积分 - 34413

排名 - 14591

阅读排行榜

抽象类: AbstractExecutorService

实现类: ExecutorCompletionService, ThreadPoolExecutor, ScheduledThreadPoolExecutor

从图中就可以看到主要的方法, 本文主要讨论的是ThreadPoolExecutor

研读ThreadPoolExecutor

看一下该类的构造器:

```
public ThreadPoolExecutor(int paramInt1, int paramInt2, long paramLong, TimeUnit paramTimeUnit,
    BlockingQueue<Runnable> paramBlockingQueue, ThreadFactory paramThreadFactory,
    RejectedExecutionHandler paramRejectedExecutionHandler) {
    this.cctl = new AtomicInteger(cctlOf(-536870912, 0));
    this.mainLock = new ReentrantLock();
    this.workers = new HashSet();
    this.termination = this.mainLock.newCondition();
    if ((paramInt1 < 0) || (paramInt2 <= 0) || (paramInt2 < paramInt1) || (paramLong < 0L))
        throw new IllegalArgumentException();
    if ((paramBlockingQueue == null) || (paramThreadFactory == null) || (paramRejectedExecutionHandler == null))
        throw new NullPointerException();
    this.corePoolSize = paramInt1;
    this.maximumPoolSize = paramInt2;
    this.workQueue = paramBlockingQueue;
    this.keepAliveTime = paramTimeUnit.toNanos(paramLong);
    this.threadFactory = paramThreadFactory;
    this.handler = paramRejectedExecutionHandler;
}
```

corePoolSize :线程池的核心池大小, 在创建线程池之后, 线程池默认没有任何线程。

当有任务过来的时候才会去创建创建线程执行任务。换个说法, 线程池创建之后, 线程池中的线程数为0, 当任务过来就会创建一个线程去执行, 直到线程数达到corePoolSize 之后, 就会被到达的任务放在队列中。(注意是到达的任务)。换句更精炼的话: corePoolSize 表示允许线程池中允许同时运行的最大线程数。

如果执行了线程池的prestartAllCoreThreads() 方法, 线程池会提前创建并启动所有核心线程。

maximumPoolSize :线程池允许的最大线程数, 他表示最大能创建多少个线程。maximumPoolSize肯定是大于等于corePoolSize。

keepAliveTime :表示线程没有任务时最多保持多久然后停止。默认情况下, 只有线程池中线程数大于corePoolSize 时, keepAliveTime 才会起作用。换句话说, 当线程池中的线程数大于corePoolSize, 并且一个线程空闲时间达到了keepAliveTime, 那么就是shutdown。

Unit:keepAliveTime 的单位。

workQueue : 一个阻塞队列, 用来存储等待执行的任务, 当线程池中的线程数超过它的corePoolSize的时候, 线程会进入阻塞队列进行阻塞等待。通过workQueue, 线程池实现了阻塞功能

threadFactory : 线程工厂, 用来创建线程。

handler :表示当拒绝处理任务时的策略。

任务缓存队列

在前面我们多次提到了任务缓存队列, 即workQueue, 它用来存放等待执行的任务。

workQueue的类型为BlockingQueue<Runnable>, 通常可以取下面三种类型:

1) 有界任务队列ArrayBlockingQueue: 基于数组的先进先出队列, 此队列创建时必须指定大小;

2) 无界任务队列LinkedBlockingQueue: 基于链表的先进先出队列, 如果创建时没有指定此队列大小, 则默认为Integer.MAX_VALUE;

3) 直接提交队列synchronousQueue: 这个队列比较特殊, 它不会保存提交的任务, 而是将直接新建一个线程来执行新来的任务。

拒绝策略

AbortPolicy: 丢弃任务并抛出RejectedExecutionException

CallerRunsPolicy: 只要线程池未关闭, 该策略直接在调用者线程中, 运行当前被丢弃的任务。显然这样做不会真的丢弃任务, 但是, 任务提交线程的性能极有可能急剧下降。

DiscardOldestPolicy: 丢弃队列中最老的一个请求, 也就是即将被执行的一个任务, 并尝试再次提交当前任务。

DiscardPolicy: 丢弃任务, 不做任何处理。

- 1. springboot之使用优雅地操作Redis(23203)
- 2. 由浅入深理解Java线程如何使用(19805)
- 3. 优化springboot(776)
- 4. 记一次内存溢出的分析(10)
- 5. spring boot之从零开始(6989)

评论排行榜

- 1. spring boot之从零开始(31)
- 2. 记一次内存溢出的分析(10)
- 3. 记一次线程池调优经历(10)
- 4. 由浅入深理解Java线程如何使用(10)
- 5. springboot之使用优雅地操作Redis(4)

推荐排行榜

- 1. 记一次内存溢出的分析(10)
- 2. 由浅入深理解Java线程如何使用(29)
- 3. spring boot之从零开始(27)
- 4. 记一次线程池调优经历(10)
- 5. 长连接、短连接、心跳连(6)

291

线程池的任务处理策略：

如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

线程池的关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：


shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

源码分析

首先来看最核心的execute方法，这个方法在AbstractExecutorService中并没有实现，从Executor接口，直到ThreadPoolExecutor才实现了改方法，

ExecutorService中的submit(),invokeAll(),invokeAny()都是调用的execute方法，所以execute是核心中的核心,源码分析将围绕它逐步展开。



```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    /*
     * Proceed in 3 steps:
     *
     * 1. If fewer than corePoolSize threads are running, try to
     * start a new thread with the given command as its first
     * task. The call to addWorker atomically checks runState and
     * workerCount, and so prevents false alarms that would add
     * threads when it shouldn't, by returning false.
     * 如果正在运行的线程数小于corePoolSize，那么将调用addWorker 方法来创建一个新的线程，并将该任务作为新线程的第一个任务来执
     * 行。
     *
     * 当然，在创建线程之前会做原子性质的检查，如果条件不允许，则不创建线程来执行任务，并返回false。
     *
     * 2. If a task can be successfully queued, then we still need
     * to double-check whether we should have added a thread
     * (because existing ones died since last checking) or that
     * the pool shut down since entry into this method. So we
     * recheck state and if necessary roll back the enqueueing if
     * stopped, or start a new thread if there are none.
     * 如果一个任务成功进入阻塞队列，那么我们需要进行一个双重检查来确保是我们已经添加一个线程（因为存在着一些线程在上次检查后他已经死
     * 亡）或者
     *
     * 当我们进入该方法时，该线程池已经关闭。所以，我们将重新检查状态，线程池关闭的情况下则回滚入队列，线程池没有线程的情况则创建一个
     * 新的线程。
     *
     * 3. If we cannot queue task, then we try to add a new
     * thread. If it fails, we know we are shut down or saturated
     * and so reject the task.
     * 如果任务无法入队（队列满了），那么我们将尝试新开启一个线程（从corepoolsize到扩充到maximum），如果失败了，那么可以确定原
     * 因，要么是
     *
     * 线程池关闭了或者饱和了（达到maximum），所以我们执行拒绝策略。
     *
     */
    // 1.当前线程数量小于corePoolSize，则创建并启动线程。
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            // 成功，则返回
        return;
        c = ctl.get();
    }
    // 2.步骤1失败，则尝试进入阻塞队列，
    if (isRunning(c) && workQueue.offer(command)) {
        // 入队成功，检查线程池状态，如果状态部署RUNNING而且remove成功，则拒绝任务
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        // 如果当前worker数量为0，通过addWorker(null, false)创建一个线程，其任务为null
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
}
```

29

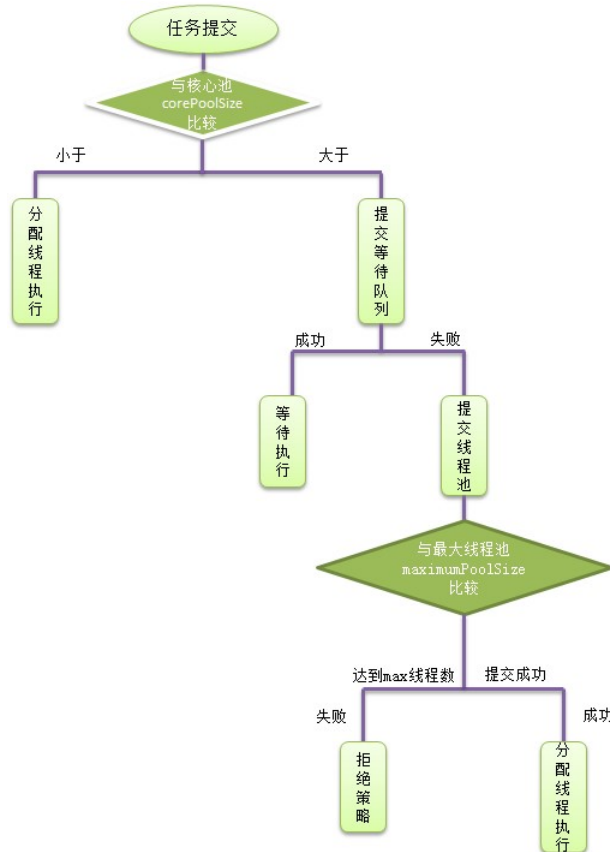
1

```

        addWorker(null, false);
    }
    // 3. 步骤1和2失败, 则尝试将线程池的数量有corePoolSize扩充至maxPoolSize, 如果失败, 则拒绝任务
    else if (!addWorker(command, false))
        reject(command);
}

```

相信看了代码也是一脸懵,接下来用一个流程图来讲一讲, 他究竟干了什么事:



结合上面的流程图来逐行解析, 首先前面进行空指针检查,

wonrkerCountOf()方法能够取得当前线程池中的线程的总数, 取得当前线程数与核心池大小比较,

- 如果小于, 将通过addWorker()方法调度执行。
- 如果大于核心池大小, 那么就提交到等待队列。
- 如果进入等待队列失败, 则会将任务直接提交给线程池。
- 如果线程数达到最大线程数, 那么就提交失败, 执行拒绝策略。

excute()方法中添加任务的方式是使用addWorker () 方法, 看一下源码, 一起学习一下。

```

private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    // 外层循环, 用于判断线程池状态
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
                firstTask == null &&
                ! workQueue.isEmpty()))
            return false;

        // 内层的循环, 任务是将worker数量加1
        for (;;) {
            int wc = workerCountOf(c);

```

```

        if (wc >= CAPACITY ||
            wc >= (core ? corePoolSize : maximumPoolSize))
            return false;
        if (compareAndIncrementWorkerCount(c))
            break retry;
        c = ctl.get(); // Re-read ctl
        if (runStateOf(c) != rs)
            continue retry;
        // else CAS failed due to workerCount change; retry inner loop
    }
}
// worker加1后, 接下来将worker添加到HashSet<Worker>中, 并启动worker
boolean workerStarted = false;
boolean workerAdded = false;
Worker w = null;
try {
    final ReentrantLock mainLock = this.mainLock;
    w = new Worker(firstTask);
    final Thread t = w.thread;
    if (t != null) {
        mainLock.lock();
        try {
            // Recheck while holding lock.
            // Back out on ThreadFactory failure or if
            // shut down before lock acquired.
            int c = ctl.get();
            int rs = runStateOf(c);

            if (rs < SHUTDOWN ||
                (rs == SHUTDOWN && firstTask == null)) {
                if (t.isAlive()) // precheck that t is startable
                    throw new IllegalThreadStateException();
                workers.add(w);
                int s = workers.size();
                if (s > largestPoolSize)
                    largestPoolSize = s;
                workerAdded = true;
            }
        } finally {
            mainLock.unlock();
        }
        // 如果往HashSet<Worker>添加成功, 则启动该线程
        if (workerAdded) {
            t.start();
            workerStarted = true;
        }
    }
} finally {
    if (! workerStarted)
        addWorkerFailed(w);
}
return workerStarted;
}

```

addWorker(Runnable firstTask, boolean core)的主要任务是创建并启动线程。

他会根据当前线程的状态和给定的值 (core or maximum) 来判断是否可以创建一个线程。

addWorker共有四种传参方式。execute使用了其中三种, 分别为:

1.addWorker(paramRunnable, true)

线程数小于corePoolSize时, 放一个需要处理的task进Workers Set。如果Workers Set长度超过corePoolSize, 就返回false。

2.addWorker(null, false)

放入一个空的task进workers Set, 长度限制是maximumPoolSize。这样一个task为空的worker在线程执行的时候会去任务队列里拿任务, 这样就相当于创建了一个新的线程, 只是没有马上分配任务。

3.addWorker(paramRunnable, false)

当队列被放满时, 就尝试将这个新来的task直接放入Workers Set, 而此时Workers Set的长度限制是maximumPoolSize。如果线程池也满了的话就返回false。

29

1

还有一种情况是execute()方法没有使用的

`addWorker(null, true)`

这个方法就是放一个null的task进Workers Set，而且是在小于corePoolSize时，如果此时Set中的数量已经达到corePoolSize那就返回false，什么也不干。实际使用中是在

```
startAllCoreThreads()
```

方法，这个方法用来为线程池预先启动corePoolSize个worker等待从workQueue中获取任务执行。

执行流程：

- 判断线程池当前是否为可以添加worker线程的状态，可以则继续下一步，不可以return false：
 - 线程池状态>shutdown，可能为stop、tidying、terminated，不能添加worker线程
 - 线程池状态==shutdown，firstTask不为空，不能添加worker线程，因为shutdown状态的线程池不接收新任务
 - 线程池状态==shutdown，firstTask==null，workQueue为空，不能添加worker线程，因为firstTask为空是为了添加一个没有任务的线程再从workQueue获取task，而workQueue为空，说明添加无任务线程已经没有意义
- 线程池当前线程数量是否超过上限（corePoolSize 或 maximumPoolSize），超过了return false，没超过则对workerCount+1，继续下一步
- 在线程池的ReentrantLock保证下，向Workers Set中添加新创建的worker实例，添加完成后解锁，并启动worker线程，如果这一切都成功了，return true，如果添加worker入Set失败或启动失败，调用addWorkerFailed()逻辑

常见的四种线程池

newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int var0) {
    return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue());
}

public static ExecutorService newFixedThreadPool(int var0, ThreadFactory var1) {
    return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue(), var1);
}
```

固定大小的线程池，可以指定线程池的大小，该线程池corePoolSize和maximumPoolSize相等，阻塞队列使用的是LinkedBlockingQueue，大小为整数最大值。

该线程池中的线程数量始终不变，当有新任务提交时，线程池中有空闲线程则会立即执行，如果没有，则会暂存到阻塞队列。对于固定大小的线程池，不存在线程数量的变化。同时使用无界的LinkedBlockingQueue来存放执行的任务。当任务提交十分频繁的时候，LinkedBlockingQueue

迅速增大，存在着耗尽系统资源的问题。而且在线程池空闲时，即线程池中沒有可运行任务时，它也不会释放工作线程，还会占用一定的系统资源，需要shutdown。

newSingleThreadExecutor

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

public static ExecutorService newSingleThreadExecutor(ThreadFactory var0) {
    return new Executors.FinalizableDelegatedExecutorService(new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue(), var0));
}
```

单个线程线程池，只有一个线程的线程池，阻塞队列使用的是LinkedBlockingQueue,若有多余的任务提交到线程池中，则会被暂存到阻塞队列，待空闲时再去执行。按照先入先出的顺序执行任务。

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, 2147483647, 60L, TimeUnit.SECONDS, new SynchronousQueue());
}

public static ExecutorService newCachedThreadPool(ThreadFactory var0) {
    return new ThreadPoolExecutor(0, 2147483647, 60L, TimeUnit.SECONDS, new SynchronousQueue(), var0);
}
```

缓存线程池，缓存的线程默认存活60秒。线程的核心池corePoolSize大小为0，核心池最大为Integer.**MAX_VALUE**,阻塞队列使用的是SynchronousQueue。是一个直接提交的阻塞队列， 他总会迫使线程池增加新的线程去执行新的任务。在没有任务执行时，当线程的空闲时间超过keepAliveTime（60秒），则工作线程将会终止被回收，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销。如果同时又大量任务被提交，而且任务执行的时间不是特别快，那么线程池便会新增出等量的线程池处理任务，这很可能会很快耗尽系统的资源。

newScheduledThreadPool

```
public static ScheduledExecutorService newScheduledThreadPool(int var0) {  
    return new ScheduledThreadPoolExecutor(var0);  
}  
  
public static ScheduledExecutorService newScheduledThreadPool(int var0, ThreadFactory var1) {  
    return new ScheduledThreadPoolExecutor(var0, var1);  
}
```

定时线程池，该线程池可用于周期性地去执行任务，通常用于周期性的同步数据。

scheduleAtFixedRate:是以固定的频率去执行任务，周期是指每次执行任务成功执行之间的间隔。

schedultWithFixedDelay:是以固定的延时去执行任务，延时是指上一次执行成功之后和下一次开始执行的之前的时间。

使用实例

newFixedThreadPool实例：

View Code

newCachedThreadPool实例：

View Code

这里没用调用shutDown方法，这里可以发现过60秒之后，会自动释放资源。

newSingleThreadExecutor

View Code

这里需要注意一点，newSingleThreadExecutor和newFixedThreadPool一样，在线程池中没有任务时可执行，也不会释放系统资源的，所以需要shutdown。

newScheduledThreadPool

View Code

最后杂谈

如何选择线程池数量

线程池的大小决定着系统的性能，过大或者过小的线程池数量都无法发挥最优的系统性能。

当然线程池的大小也不需要做的太过于精确，只需要避免过大和过小的情况。一般来说，确定线程池的大小需要考虑CPU的数量，内存大小，任务是计算密集型还是IO密集型等因素

NCPU = CPU的数量

UCPU = 期望对CPU的使用率 0 ≤ UCPU ≤ 1

W/C = 等待时间与计算时间的比率

如果希望处理器达到理想的使用率，那么线程池的最优大小为：

线程池大小=NCPU *UCPU(1+W/C)

在Java中使用

```
int ncpus = Runtime.getRuntime().availableProcessors();
```

获取CPU的数量。

线程池工厂

Executors的线程池如果不指定线程工厂会使用Executors中的DefaultThreadFactory,默认线程池工厂创建的线程都是非守护线程。

使用自定义的线程工厂可以做很多事情,比如可以跟踪线程池在何时创建了多少线程,也可以自定义线程名称和优先级。如果将

新建的线程都设置成守护线程,当主线程退出后,将会强制销毁线程池。

下面这个例子,记录了线程的创建,并将所有的线程设置成守护线程。

View Code

扩展线程池

ThreadPoolExecutor是可以拓展的,它提供了几个可以在子类中改写的方法: beforeExecute,afterExecute和terminated。

在执行任务的线程中将调用beforeExecute和afterExecute,这些方法中还可以添加日志,计时,监视或统计收集的功能,

还可以用来输出有用的调试信息,帮助系统诊断故障。下面是一个扩展线程池的例子:

View Code

线程池的正确使用

以下阿里编码规范里面说的一段话:

线程池不允许使用Executors去创建,而是通过ThreadPoolExecutor的方式,这样的处理方式让写的同学更加明确线程池的运行规则,规避资源耗尽的风险。说明: Executors各个方法的弊端:

- 1) newFixedThreadPool和newSingleThreadExecutor:
主要问题是堆积的请求处理队列可能会耗费非常大的内存,甚至OOM。
- 2) newCachedThreadPool和newScheduledThreadPool:
主要问题是线程数最大数是Integer.MAX_VALUE,可能会创建数量非常多的线程,甚至OOM。

手动创建线程池有几个注意点

- 1.任务独立。如何任务依赖于其他任务,那么可能产生死锁。例如某个任务等待另一个任务的返回值或执行结果,那么除非线程池足够大,否则将发生线程饥饿死锁。
- 2.合理配置阻塞时间过长的任务。如果任务阻塞时间过长,那么即使不出现死锁,线程池的性能也会变得很糟糕。在Java并发包里可阻塞方法都同时定义了限时方式和不限时方式。例如

Thread.join,BlockingQueue.put,CountDownLatch.await等,如果任务超时,则标识任务失败,然后中止任务或者将任务放回队列以便随后执行,这样,无论任务的最终结果是否成功,这种办法都能够保证任务总能继续执行下去。
- 3.设置合理的线程池大小。只需要避免过大或者过小的情况即可,上文的公式**线程池大小=NCPU *UCPU(1+W/C)**。
- 4.选择合适的阻塞队列。newFixedThreadPool和newSingleThreadExecutor都使用了无界的阻塞队列,无界阻塞队列会有消耗很大的内存,如果使用了有界阻塞队列,它会规避内存占用过大的问题,但是当任务填满有界阻塞队列,新的任务该怎么办?在使用有界队列是,需要选择合适的拒绝策略,队列的大小和线程池的大小必须一起调节。对于非常大的或者无界的线程池,可以使用SynchronousQueue来避免任务排队,以直接将任务从生产者提交到工作者线程。

下面是Thrift框架处理socket任务所使用的一个线程池,可以看一下FaceBook的工程师是如何自定义线程池的。

```
private static ExecutorService createDefaultExecutorService(Args args) {
    SynchronousQueue executorQueue = new SynchronousQueue();

    return new ThreadPoolExecutor(args.minWorkerThreads, args.maxWorkerThreads, 60L, TimeUnit.SECONDS,
        executorQueue);
}
```

总结:

本文是作者在平时的工作学习中总结出来的,如果不足之处欢迎批评斧正。

参考资料

《实战Java》高并发程序设计

《Java Concurrency in Practice》

Java线程池ThreadPoolExecutor使用和分析(二)

我的博客即将入驻“云栖社区”，诚邀技术同仁一同入驻。

个人博客网站 <http://www.janti.cn>

分类： 1.2 +----多线程

标签： 线程池

好文置顶

关注我

收藏该文



Janti

关注 - 14

粉丝 - 90

+加关注

« 上一篇: Docker学习笔记——制作容器与容器概念

» 下一篇: 记一次线程池调优经历

posted @ 2018-01-07 23:35 Janti 阅读(19804) 评论(10) 编辑 收藏

评论列表

#1楼 2018-08-07 00:30 鱼一鱼

可以可以，很详细

支持(0) 反对(0)

#2楼 2018-10-29 18:48 堉堉堉堉堉

难得的好文章居然没有人分享，顶一个

支持(0) 反对(0)

#3楼 2018-10-30 10:12 白色程序猿

朋友 看到你github上的springboot项目很不错 我也是搞java的 之前在移动工作 现在在一家金融公司上班 最近有点时间 可不可以开权限一起开发 放心 我单拉分支 master你进行管理合并怎么样？

支持(0) 反对(0)

#4楼[楼主] 2018-10-30 10:20 Janti

@ 白色程序猿

你指的是哪一个项目

支持(0) 反对(0)

#5楼 2018-10-30 10:40 白色程序猿

Jantent 这个不错 其他的我也有看 类似myblog的都可以 我想搞一个这样的项目做做 平常都是公司的期货业务有些枯燥 如果不方便就算了 没关系的

支持(0) 反对(0)

#6楼[楼主] 2018-10-30 11:29 Janti

@ 白色程序猿

这个项目 只是用来练手的，没啥价值的，你直接fork过去，自己改造就好了，最近太忙了，jantent已经不维护了

支持(0) 反对(0)

#7楼 2018-10-30 11:41 白色程序猿

29

1

感觉比较简单好上手 不知道博主有没有好一点的项目推荐一下 也想深入学习研究一下

支持(0) 反对(0)

#8楼[楼主] 2018-10-30 11:44 Janti

@ 白色程序猿

你在GitHub上面搜一下 springboot, 一大把, 很多开源的

支持(0) 反对(0)

#9楼 2018-11-02 21:53 溢性循环

问下 为什么我操作的时候 执行数据库查询的时候 总是失败啊 到了查询数据库那一步就跳过去了 不再执行下面的语句

支持(0) 反对(0)

#10楼 2018-12-05 10:19 江上船儿

当提交第(corePoolSize+workQueue.size()+1)个任务且当前线程数小于maximumPoolSize时, 会创建线程, 执行任务。

有个问题, 和workQueue中的任务的执行相比, 哪个先执行?

求大佬解惑呀

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论, 请 [登录](#) 或 [注册](#), [访问网站首页](#)。

【推荐】超50万VC++源码: 大型组态工控、电力仿真CAD与GIS源码库!

【活动】华为云12.12会员节全场1折起 满额送Mate20

【活动】华为云会员节云服务特惠1折起

【活动】腾讯云+社区开发者大会12月15日首都北京盛大起航!



腾讯云
腾讯云AMD云服务器
节省IT成本30%
1核1G AMD机型0.57元/天起
立即抢购

相关博文:

- Java 进阶7 并行优化 JDK多任务执行框架技术
- java中的多线程
- 多线程编程几个误区
- Java 多线程 线程池
- 线程池全面总结



intel × 英特尔 AI护老虎, 智护生态
英特尔®, 用人工智能解决大问题

最新新闻:

- 登陆不到两周, InSight探测器意外捕捉到火星的风声
 - 从天上到地上, 无人驾驶终于航行到了海上
 - Gmail推销邮件过滤器疑似出现故障: 大量培根邮件涌入用户主邮箱
 - 支付宝公布8年前视频: 马云亲自体验扫码支付
 - ofo被告: 至少遭9家公司起诉 多地运维疑似停滞
- » 更多新闻...

Copyright ©2018 Janti