线程池

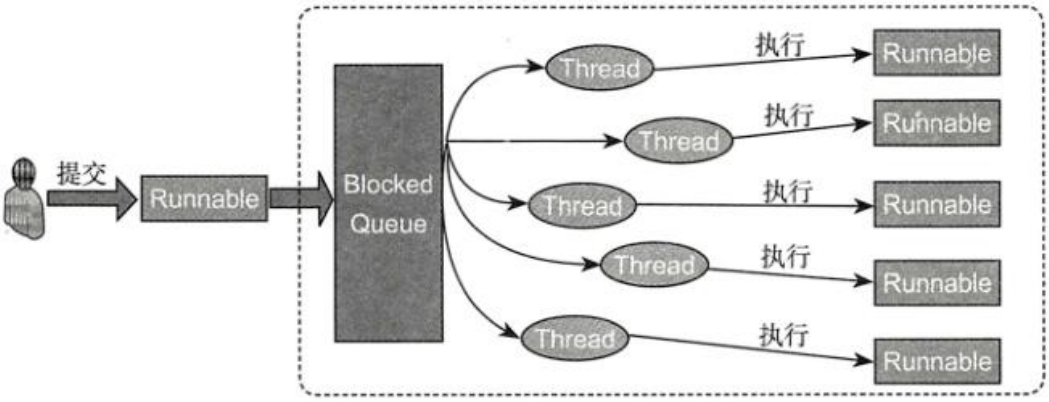
自 JDK1.5 起，util 包提供了 ExecutorService 线程池的实现，主要目的是为了重复利用线程，提高系统效率。通过前文的学习我们得知 Thread 是一个重量级的资源，创建、启动以及销毁都是比较耗费系统资源的，因此对线程的重复利用一种是非常好的程序设计习惯，加之系统中可创建的线程数量是有限的，线程数量和系统性能是一种抛物线的关系，也就是说当线程数量达到某个数值的时候，性能反倒会降低很多，因此对线程的管理，尤其是数量的控制更能直接决定程序的性能。

我们从原理入手，设计一个线程池，其目的并不是重复地发明轮子，而是为了帮助大家弄清楚一个线程池应该具备哪些功能，线程池的实现需要注意哪些细节。然后再讲解一下 ExecutorService 的使用。

二十四、自定义线程池

24.1、线程池原理

所谓线程池，通俗的理解就是有一个池子，里面存放着已经创建好的线程，当有任务提交给线程池执行时，池子中的某个线程会主动执行该任务。如果池子中的线程数量不够应付数量众多的任务时，则需要自动扩充新的线程到池子中，但是该数量是有限的，就好比池塘的水界线一样。当任务比较少的时候，池子中的线程能够自动回收，释放资源。为了能够异步地提交任务和缓存未被处理的任务，需要有一个任务队列，如图所示。

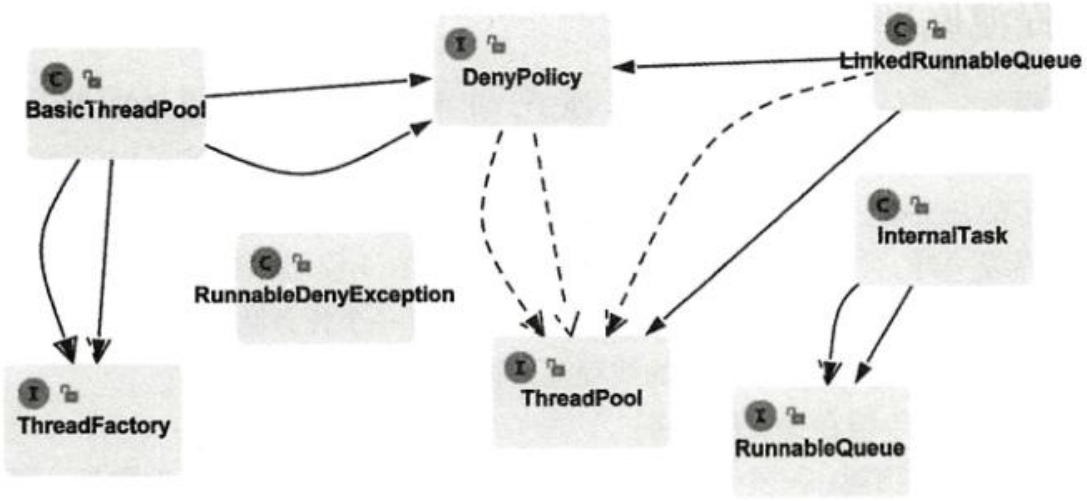


通过上面的描述可知，一个完整的线程池应该具备如下要素。

- 任务队列：用于缓存提交的任务。
- 线程数量管理功能：一个线程池必须能够很好地管理和控制线程数量，可通过如下三个参数来实现，比如创建线程池时初始的线程数量 init；线程池自动扩充时最大的线程数量 max；在线程池空闲时需要释放线程但是也要维护一定数量的活跃数量或者核心数量 core。有了这三个参数，就能够很好地控制线程池中的线程数量，将其维护在一个合理的范围之内，三者之间的关系是 init<=core<=max。
- 任务拒绝策略：如果线程数量已达到上限且任务队列已满，则需要有相应的拒绝策略来通知任务提交者。
- 线程工厂：主要用于个性化定制线程，比如将线程设置为守护线程以及设置线程名称等。
- QueueSize：任务队列主要存放提交的 Runnable，但是为了防止内存溢出，需要有 limit 数量对其进行控制。
- Keepedalive 时间：该时间主要决定线程各个重要参数自动维护的时间间隔。

24.2、线程池实现

下面我们实现一个比较简单的 ThreadPool，虽然比较简单，但是该有的功能基本上都具备，对大家学习和掌握 JUC 中的 ExecutorService 也有一定的帮助。如图所示为线程池实现类图。



ThreadPool

ThreadPool 主要定义了一个线程池应该具备的基本操作和方法，下面是 ThreadPool 接口定义的方法：

```
package com.bjsxt.chapter24;

/**
 * 线程池接口
```

```
 */
public interface ThreadPool {

    // 提交任务到线程池
    void execute(Runnable runnable);

    // 关闭线程池
    void shutdown();

    // 获取线程池的初始化大小
    int getInitSize();

    // 获取线程池最大的线程数
    int getMaxSize();

    // 获取线程池的核心线程数量
    int getCoreSize();

    // 获取线程池中用于缓存任务队列的大小
    int getQueueSize();

    // 获取线程池中活跃线程的数量
    int getActiveCount();

    // 查看线程池是否已经被 shutdown
    boolean isShutdown();

}
```

RunanbleQueue

RunanbleQueue 主要用于存放提交的 Runnable，该 Runnable 是一个 BlockedQueue，并且有 limit 的限制，示例代码如下。

```
package com.bjsxt.chapter24;

/**
 * 任务队列，主要用于缓存提交到线程池中的任务
 */
public interface RunnableQueue {

    // 当有新的任务进来时首先会 offer 到队列中
    void offer(Runnable runnable);

    // 工作线程通过 take 方法获取 Runnable
    Runnable take();

    // 获取任务队列中任务的数量
    int size();

}
```

ThreadFactory

ThreadFactory 提供了创建线程的接口，以便于个性化地定制 Thread，比如 Thread 应该被加到哪个 Group 中，优先级、线程名字以及是否为守护线程等，示例代码如下。

```java
package com.bjsxt.chapter24;

/**
 * 创建线程的工厂
 */
@FunctionalInterface
public interface ThreadFactory {

    Thread createThread(Runnable runnable);

}
```

DenyPolicy

DenyPolicy 主要用于当 Queue 中的 Runnable 达到了 limit 上限时，决定采用何种策略通知提交者。该接口中定义了三种默认的实现，示例代码如下。

```java
package com.bjsxt.chapter24;

/**
 * 拒绝策略
 */
@FunctionalInterface
public interface DenyPolicy {

    void reject(Runnable runnable, ThreadPool threadPool);

    // 该拒绝策略会直接将任务丢弃
    class DiscardDenyPolicy implements DenyPolicy {
        @Override
        public void reject(Runnable runnable, ThreadPool threadPool) {
            // ...
        }
    }

    // 该拒绝策略会向任务提交者抛出异常
    class AbortDenyPolicy implements DenyPolicy {
        @Override
        public void reject(Runnable runnable, ThreadPool threadPool) {
            throw new RunnableDenyException("任务 " + runnable + "将被终止。");
        }
    }

    // 该拒绝策略会使任务在提交者所在的线程中执行任务
    class RunnerDenyPolicy implements DenyPolicy {
        @Override
        public void reject(Runnable runnable, ThreadPool threadPool) {
            if (!threadPool.isShutdown())
                runnable.run();
        }
    }

}
```

RunnableDenyException

RunnableDenyException 是 RuntimeException 的子类，主要用于通知任务提交者，

任务队列已无法再接收新的任务。示例代码如下。

```
package com.bjsxt.chapter24;

/**
 * 自定义拒绝策略异常类
 */
public class RunnableDenyException extends RuntimeException {

    public RunnableDenyException(String message) {
        super(message);
    }

}
```

InternalTask

InternalTask 是 Runnable 的一个实现，主要用于线程池内部，该类会使用到 RunnableQueue，然后不断地从 queue 中取出某个 Runnable，并运行 Runnable 的 run 方法，除此之外，代码还对该类增加了一个开关方法 stop，主要用于停止当前线程，一般在线程池销毁和线程数量维护的时候会使用到。示例代码如下。

```
package com.bjsxt.chapter24;

/**
 * 任务类
 */
public class InternalTask implements Runnable {

    private final RunnableQueue runnableQueue;
    private volatile boolean running = true;

    public InternalTask(RunnableQueue runnableQueue) {
        this.runnableQueue = runnableQueue;
    }

    @Override
    public void run() {
        // 如果当前任务为 running 并且没有被中断，则将其不断的从 queue 中获取 runnable，然后执行 run 方法
        while (running && !Thread.currentThread().isInterrupted()) {
            Runnable task = runnableQueue.take();
            task.run();
        }
    }

    // 停止当前任务，主要会在线程池的 shutdown 方法中使用
    public void stop() {
        this.running = false;
    }

}
```

24.3、线程池详细实现

本节将对线程池进行详细的实现，其中会涉及很多同步的技巧和资源竞争，我们将结合并发基础部分大多数知识灵活使用，也算是一个综合练习。

```
package com.bjsxt.chapter24;

import java.util.LinkedList;

public class LinkedRunnableQueue implements RunnableQueue {

    // 任务队列的最大容量，在构造时传入
    private final int limit;

    // 若任务队列中的任务已经满了，则需要执行拒绝策略
    private final DenyPolicy denyPolicy;

    // 存放任务的队列
    private final LinkedList<Runnable> runnableQueue = new LinkedList<>();
    private final ThreadPool threadPool;

    public LinkedRunnableQueue(int limit, DenyPolicy denyPolicy, ThreadPool threadPool)
{
        this.limit = limit;
        this.denyPolicy = denyPolicy;
        this.threadPool = threadPool;
    }

    @Override
    public void offer(Runnable runnable) {

    }

    @Override
    public Runnable take() {
        return null;
    }

    @Override
    public int size() {
        return 0;
    }

}
```

在 LinkedRunnableQueue 中有几个重要的属性，第一个是 limit，也就是 Runnable 队列的上限；当提交的 Runnable 数量达到 limit 上限时，则会调用 DenyPolicy 的 reject 方法；RunnableList 是一个双向循环列表，用于存放 Runnable 任务，示例代码如下：

```
@Override
public void offer(Runnable runnable) {
    synchronized (runnableList) {
        if (runnableList.size() >= limit) {
            // 无法容纳新的任务时执行拒绝策略
            denyPolicy.reject(runnable, threadPool);
        } else {
            // 将任务加入到队尾，并且唤醒阻塞中的线程
```

```
            runnableList.addLast(runnable);
            runnableList.notifyAll();
        }
    }
}
```

offer 方法是一个同步方法，如果队列数量达到了上限，则会执行拒绝策略，否则会将 Runnable 存放至队列中，同时唤醒 take 任务的线程：

```
@Override
public Runnable take() throws InterruptedException {
    synchronized (runnableList) {
        while (runnableList.isEmpty()) {
            try {
                // 如果任务队列中没有可执行的任务，则当前线程将会挂起
                runnableList.wait();
            } catch (InterruptedException e) {
                throw e;
            }
        }
        // 从任务队列头部移除一个任务
        return runnableList.removeFirst();
    }
}
```

take 方法也是同步方法，线程不断从队列中获取 Runnable 任务，当队列为空的时候工作线程会陷入阻塞，有可能在阻塞的过程中被中断，为了传递中断信号需要在 catch 语句块中将异常抛出以通知上游（InternalTask）。

size 方法用于返回 runnablelist 的任务个数。

```
@Override
public int size() {
    synchronized (runnableList) {
        // 返回当前任务队列中的任务数
        return runnableList.size();
    }
}
```

24.3.1、初始化线程池

根据前面的讲解，线程池需要有数量控制属性、创建线程工厂、任务队列策略等功能，线程池初始化代码如下所示。

```
public class BasicThreadPool extends Thread implements ThreadPool {

    // 初始化线程数量
    private final int initSize;
    // 线程池最大线程数量
    private final int maxSize;
    // 线程池核心线程数量
    private final int coreSize;
    // 当前活跃的线程数量
    private int activeCount;
    // 创建线程所需的工厂
    private final ThreadFactory threadFactory;
    // 任务队列
    private final RunnableQueue runnableQueue;
    // 线程池是否已经被 shutdown
    private volatile boolean isShutdown = false;
```

```
    // 工作线程队列
    private final Queue<ThreadTask> threadQueue = new ArrayDeque<>();
    // 拒绝策略
    private final static DenyPolicy DEFAULT_DENY_POLICY = new
DenyPolicy.DiscardDenyPolicy();
    private final static ThreadFactory DEFAULT_THREAD_FACTORY = new
DefaultThreadFactory();
    private final long keepAliveTime;
    private final TimeUnit timeUnit;

    public BasicThreadPool(int initSize, int maxSize, int coreSize, int queueSize) {
        this(initSize, maxSize, coreSize,
                DEFAULT_THREAD_FACTORY, queueSize,
                DEFAULT_DENY_POLICY, 10, TimeUnit.SECONDS);
    }

    public BasicThreadPool(int initSize, int maxSize, int coreSize,
                           ThreadFactory threadFactory, int queueSize,
                           DenyPolicy denyPolicy, long keepAliveTime, TimeUnit timeUnit)
{
        this.initSize = initSize;
        this.maxSize = maxSize;
        this.coreSize = coreSize;
        this.threadFactory = threadFactory;
        this.keepAliveTime = keepAliveTime;
        this.timeUnit = timeUnit;
        this.runnableQueue = new LinkedRunnableQueue(queueSize, denyPolicy, this);
        this.init();// 初始化线程池
    }

    // 初始化线程池
    private void init() {
        start();
        for (int i = 0; i < initSize; i++)
            newThread();
    }

...

}
```

24.3.2、提交任务

提交任务非常简单，只是将 Runnable 插人 runnableQueue 中即可。示例代码如下：

```
// 提交任务
@Override
public void execute(Runnable runnable) {
    if (this.isShutdown)
        throw new IllegalStateException("线程池已销毁。");
    // 提交任务只是简单的往任务队列中插入 Runnable
    this.runnableQueue.offer(runnable);
}
```

24.3.3、线程池自动维护

线程池中线程数量的维护主要由 run 负责，这也是为什么 BasicThreadPool 继承自

Thread 了，不过不推荐使用直接继承的方式，线程池自动维护代码如下：

```
// 创建任务线程，并启动
private void newThread() {
    InternalTask internalTask = new InternalTask(runnableQueue);
    Thread thread = this.threadFactory.createThread(internalTask);
    ThreadTask threadTask = new ThreadTask(thread, internalTask);
    threadQueue.offer(threadTask);
    this.activeCount++;
    thread.start();
}

// 从线程池中移除某个线程
private void removeThread() {
    ThreadTask threadTask = threadQueue.remove();
    threadTask.internalTask.stop();
    this.activeCount--;
}

@Override
public void run() {
    while (!isShutdown && !isInterrupted()) {
        try {
            timeUnit.sleep(keepAliveTime);
        } catch (InterruptedException e) {
            isShutdown = true;
            break;
        }

        synchronized (this) {
            if (isShutdown)
                break;
            // 当前的队列中有任务尚未处理，并且 activeCount < coreSize 则继续扩容
            if (runnableQueue.size() > 0 && activeCount < coreSize) {
                for (int i = initSize; i < coreSize; i++)
                    newThread();
                continue;
            }
            // 当前的队列中有任务尚未处理，并且 activeCount < maxSize 则继续扩容
            if (runnableQueue.size() > 0 && activeCount < maxSize) {
                for (int i = coreSize; i < maxSize; i++)
                    newThread();
            }
            // 当前的队列中没有任务，则需要回收，回收至 coreSize 即可
            if (runnableQueue.size() == 0 && activeCount > coreSize) {
                for (int i = coreSize; i < activeCount; i++)
                    newThread();
            }
        }
    }
}
```

下面重点来解说线程自动维护的方法，自动维护线程的代码块是同步代码块，主要是为了阻止在线程维护过程中线程池销毁引起的数据不一致问题。

任务队列中若存在积压任务，并且当前活动线程少于核心线程数，则新建 coreSize-initSize 数量的线程，并且将其加入到活动线程队列中，为了防止马上进行 maxSize-coreSize 数量的扩充，建议使用 continue 终止本次循环。

任务队列中有积压任务，并且当前活动线程少于最大线程数，则新建 max Size- core Size 数量的线程，并且将其加入到活动队列中。

当前线程池不够繁忙时，则需要回收部分线程，回收到 coreSize 数量即可，回收时调用 removeThread 方法，在该方法中需要考虑的一点是，如果被回收的线程恰巧从 Runnable 任务取出了某个任务，则会继续保持该线程的运行，直到完成了任务的运行为止，详见 InternalTask 的 run 方法。

24.3.4、线程池销毁

线程池的销毁同样需要同步机制的保护，主要是为了防止与线程池本身的维护线程引起数据冲突，线程池销毁代码如下：

```
@Override
public void shutdown() {
    synchronized (this) {
        if (isShutdown)
            return;
        isShutdown = true;
        threadQueue.forEach(threadTask -> {
            threadTask.internalTask.stop();
            threadTask.thread.interrupt();
        });
        this.interrupt();
    }
}
```

销毁线程池主要为了是停止 BasicThreadPool 线程，停止线程池中的活动线程并且将 isShutdown 开关变量更改为 true。

24.3.5、线程池的其他方法

```
@Override
public int getInitSize() {
    if (isShutdown)
        throw new IllegalStateException("线程池已销毁。");
    return this.initSize;
}

@Override
public int getMaxSize() {
    if (isShutdown)
        throw new IllegalStateException("线程池已销毁。");
    return this.maxSize;
}

@Override
public int getCoreSize() {
    if (isShutdown)
        throw new IllegalStateException("线程池已销毁。");
    return this.coreSize;
}

@Override
public int getQueueSize() {
    if (isShutdown)
        throw new IllegalStateException("线程池已销毁。");
```

```
    return this.runnableQueue.size();
}

@Override
public int getActiveCount() {
    if (isShutdown)
        throw new IllegalStateException("线程池已销毁。");
    return this.activeCount;
}

@Override
public boolean isShutdown() {
    if (isShutdown)
        throw new IllegalStateException("线程池已销毁。");
    return this.isShutdown;
}
```

上述代码为线程池的其他方法，主要是用来获取线程池各个参数的值（当线程池已经被关闭时，调用查询方法将会抛出异常信息），至此线程池的全部实现已经完成。

下面我们将对线程池的应用进行简单的测试。

24.4、线程池的应用

本节将写一个简单的程序分别测试线程池的任务提交、线程池线程数量的动态扩展，以及线程池的销毁功能，代码如下所示。

```
package com.bjsxt.chapter24;

import java.util.concurrent.TimeUnit;

public class ThreadPoolTest {

    public static void main(String[] args) throws InterruptedException {
        // 定义线程池，初始化线程数为 2，核心线程数为 4，最大线程数为 6，任务队列最多允许 1000
个任务
        final ThreadPool threadPool = new BasicThreadPool(2, 6, 4, 1000);

        // 定义 20 个任务并且提交给线程池
        for (int i = 0; i < 20; i++) {
            threadPool.execute(() -> {
                try {
                    TimeUnit.SECONDS.sleep(10);
                    System.out.println(Thread.currentThread().getName() + " 正在运行。
");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        while (true) {
            // 不断输出线程池的信息
            System.out.println("getActiveCount: " + threadPool.getActiveCount());
            System.out.println("getQueueSize: " + threadPool.getQueueSize());
            System.out.println("getCoreSize: " + threadPool.getCoreSize());
            System.out.println("getMaxSize: " + threadPool.getMaxSize());
            System.out.println("-----------------------------------------");
            TimeUnit.SECONDS.sleep(5);
        }
```

```
    }

}
```

上述测试代码中，定义了一个 Basic 线程池，其中初始化线程数量为 2,核心线程数量为 4,最大线程数量为 6,最大任务队列数量为 1000,同时提交了 20 个任务到线程池中，然后在 main 线程中不断地输出线程池中的线程数量信息监控变化，运行上述代码，截取的部分输出信息如下：

```
getActiveCount: 2
getQueueSize: 18
getCoreSize: 4
getMaxSize: 6
------------------------------------------------
getActiveCount: 2
getQueueSize: 18
getCoreSize: 4
getMaxSize: 6
------------------------------------------------
thread-pool-0 正在运行。
thread-pool-1 正在运行。
getActiveCount: 4
getQueueSize: 14
getCoreSize: 4
getMaxSize: 6
------------------------------------------------
getActiveCount: 4
getQueueSize: 14
getCoreSize: 4
getMaxSize: 6
------------------------------------------------
thread-pool-0 正在运行。
thread-pool-3 正在运行。
thread-pool-2 正在运行。
thread-pool-1 正在运行。
getActiveCount: 6
getQueueSize: 8
getCoreSize: 4
getMaxSize: 6
------------------------------------------------
getActiveCount: 6
getQueueSize: 8
getCoreSize: 4
getMaxSize: 6
------------------------------------------------
thread-pool-2 正在运行。
thread-pool-3 正在运行。
thread-pool-0 正在运行。
thread-pool-1 正在运行。
thread-pool-4 正在运行。
thread-pool-5 正在运行。
getActiveCount: 6
getQueueSize: 2
getCoreSize: 4
getMaxSize: 6
------------------------------------------------
getActiveCount: 6
```

```
getQueueSize: 2
getCoreSize: 4
getMaxSize: 6
-------------------------------------------
thread-pool-3 正在运行。
thread-pool-2 正在运行。
thread-pool-0 正在运行。
thread-pool-1 正在运行。
thread-pool-5 正在运行。
thread-pool-4 正在运行。
getActiveCount: 5
getQueueSize: 0
getCoreSize: 4
getMaxSize: 6
-------------------------------------------
getActiveCount: 5
getQueueSize: 0
getCoreSize: 4
getMaxSize: 6
-------------------------------------------
thread-pool-2 正在运行。
thread-pool-3 正在运行。
getActiveCount: 4
getQueueSize: 0
getCoreSize: 4
getMaxSize: 6
-------------------------------------------
```

通过上述输出信息可以看出，线程池中线程的动态扩展状况以及任务执行情况，在输出的最后会发现 activeCount 停留在了 coreSize 的位置，这也符合我们的设计。