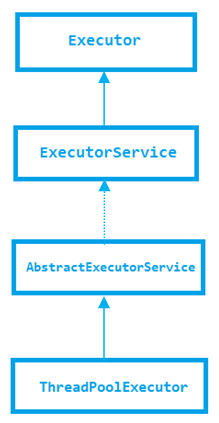
# **[java线程池01-ThreadPoolExecutor构造方法参数的使用规则](https://www.cnblogs.com/cdf-opensource-007/p/8769777.html)**

为了更好的使用多线程，JDK提供了线程池供开发人员使用，目的在于减少线程的创建和销毁次数，以此达到线程的重复利用。

其中ThreadPoolExecutor是线程池中最核心的一个类，我们先简单看一下这个类的继承关系。



 其中Executor是线程池的顶级接口，接口中只定义了一个方法  void execute(Runnable command)；线程池的操作方法都是定义子在ExecutorService子接口中的，所以说ExecutorService是线程池真正的接口。

ThreadPoolExecutor提供了四个构造方法，我们看一下参数最全的一个构造函数；

[IMG_257](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

public ThreadPoolExecutor(int corePoolSize,

　　　　　　　　　　　int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,  
　　　　　　　　　　　　　　　　　　ThreadFactory threadFactory,

RejectedExecutionHandler handler) {  
  
}

[IMG_258](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

函数的参数含义如下：

* **corePoolSize**： 线程池核心线程数
* **maximumPoolSize**：线程池最大数
* **keepAliveTime**： 空闲线程存活时间
* **unit**： 时间单位
* **workQueue**： 线程池所使用的缓冲队列
* **threadFactory**：线程池创建线程使用的工厂
* **handler**： 线程池对拒绝任务的处理策略

本节我们主要对前五个参数中的corePoolSize，maximumPoolSize及workQueue是如何配合使用做出说明（keepAliveTime，unit主要对空闲线程的存活时间做的定义，见名知意，不再做出说明），以此来引出线程池的一些特性。

threadFactory和handler这两个参数都有默认值，对于它们的用法将放到其它章节去做说明。

**特性一：当池中正在运行的线程数（包括空闲线程）小于corePoolSize时，新建线程执行任务。**

下面用实验来说明，代码如下：

[IMG_259](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

public class TestThreadPoolExecutor {

public static void main(String[] args) {

ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 60L, TimeUnit.SECONDS,new LinkedBlockingQueue<>(1));

//任务1

pool.execute(new Runnable() {

@Override

public void run() {

System.out.println("-------------helloworld\_001---------------" + Thread.currentThread().getName());

}

});

try {

//主线程睡2秒

Thread.sleep(2\*1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

//任务2

pool.execute(new Runnable() {

@Override

public void run() {

System.out.println("-------------helloworld\_002---------------" + Thread.currentThread().getName());

}

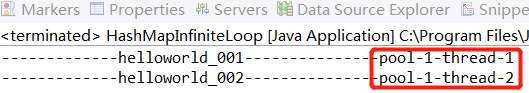
});

}

}

[IMG_260](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

实验结果如下：



实验结果分析：

从实验结果上可以看出，当执行任务1的线程（thread-1）执行完成之后，任务2并没有去复用thread-1而是新建线程（thread-2）去执行任务。

**特性二：当池中正在运行的线程数大于等于corePoolSize时，新插入的任务进入workQueue排队（如果workQueue长度允许），等待空闲线程来执行。**

下面用实验来说明，代码如下：

[IMG_262](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

public class TestThreadPoolExecutor {

public static void main(String[] args) {

ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 60L, TimeUnit.SECONDS, new LinkedBlockingQueue<>(1));

// 任务1

pool.execute(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(3 \* 1000);

System.out.println("-------------helloworld\_001---------------" + Thread.currentThread().getName());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

// 任务2

pool.execute(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(5 \* 1000);

System.out.println("-------------helloworld\_002---------------" + Thread.currentThread().getName());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

// 任务3

pool.execute(new Runnable() {

@Override

public void run() {

System.out.println("-------------helloworld\_003---------------" + Thread.currentThread().getName());

}

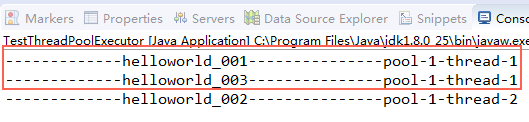
});

}

}

[IMG_263](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

 实验结果如下：



实验结果分析：

从实验结果上看，任务3会等待任务1执行完之后，有了空闲线程，才会执行。并没有新建线程执行任务3，这时**maximumPoolSize=3**这个参数不起作用**。**

**特性三：当队列里的任务数达到上限，并且池中正在运行的线程数小于maximumPoolSize，对于新加入的任务，新建线程。**

下面用实验来说明，代码如下：

[IMG_265](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

public class TestThreadPoolExecutor {

public static void main(String[] args) {

ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 60L, TimeUnit.SECONDS, new LinkedBlockingQueue<>(1));

// 任务1

pool.execute(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(3 \* 1000);

System.out.println("-------------helloworld\_001---------------" + Thread.currentThread().getName());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

// 任务2

pool.execute(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(5 \* 1000);

System.out.println("-------------helloworld\_002---------------" + Thread.currentThread().getName());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

// 任务3

pool.execute(new Runnable() {

@Override

public void run() {

System.out.println("-------------helloworld\_003---------------" + Thread.currentThread().getName());

}

});

// 任务4

pool.execute(new Runnable() {

@Override

public void run() {

System.out.println("-------------helloworld\_004---------------" + Thread.currentThread().getName());

}

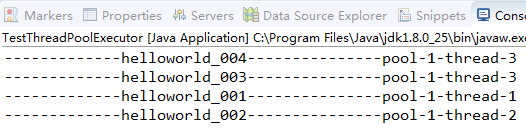
});

}

}

[IMG_266](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

实验结果如下：



实验结果分析：

当任务4进入队列时发现队列的长度已经到了上限，所以无法进入队列排队，而此时正在运行的线程数（2）小于maximumPoolSize所以新建线程执行该任务。

**特性四：当队列里的任务数达到上限，并且池中正在运行的线程数等于maximumPoolSize，对于新加入的任务，执行拒绝策略（线程池默认的拒绝策略是抛异常）。**

下面用实验来说明，代码如下：

[IMG_268](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

public class TestThreadPoolExecutor {

public static void main(String[] args) {

ThreadPoolExecutor pool = new ThreadPoolExecutor(2, 3, 60L, TimeUnit.SECONDS, new LinkedBlockingQueue<>(1));

// 任务1

pool.execute(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(3 \* 1000);

System.out.println("-------------helloworld\_001---------------" + Thread.currentThread().getName());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

// 任务2

pool.execute(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(5 \* 1000);

System.out.println("-------------helloworld\_002---------------" + Thread.currentThread().getName());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

// 任务3

pool.execute(new Runnable() {

@Override

public void run() {

System.out.println("-------------helloworld\_003---------------" + Thread.currentThread().getName());

}

});

// 任务4

pool.execute(new Runnable() {

@Override

public void run() {

try {

Thread.sleep(2 \* 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("-------------helloworld\_004---------------" + Thread.currentThread().getName());

}

});

// 任务5

pool.execute(new Runnable() {

@Override

public void run() {

System.out.println("-------------helloworld\_005---------------" + Thread.currentThread().getName());

}

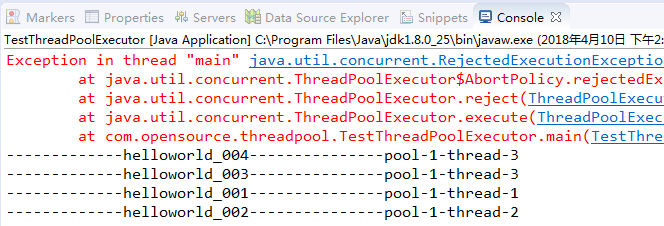
});

}

}

[IMG_269](https://www.cnblogs.com/cdf-opensource-007/p/javascript:void(0);)

实验结果如下：



实验结果分析：

当任务5加入时，队列达到上限，池内运行的线程数达到最大，故执行默认的拒绝策略，抛异常。

本文中使用到的队列类型虽然仅限于LinkedBlockingQueue这一种队列类型，但总结出来的特性，对与常用ArrayBlockingQueue 和 SynchronousQueue同样适用