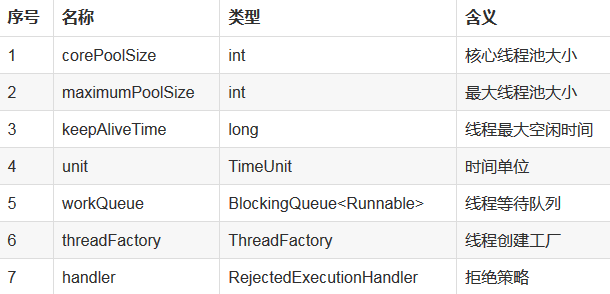
**线程池**

基本参数解析：



keepAliveTime :空闲线程存活的最长时间

拒绝策略：（任务数>（maximumPoolSize+等待队列长度））默认直接拒绝报,同时发出异常提醒 AbortHander

示例代码导入：

class MyTask implements Runnable{

private String name;

public MyTask(String name) {

this.name = name;

}

@Override

public void run() {

try {

System.out.println("task:"+name+"\t正在读取数据.....");

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("task:"+name+"\t正在读取数据完成");

}

}

上面代码是通过实现Runnable接口的方法创建了一个线程,通过睡眠一秒钟然后再执行下面的打印语句来表示完成了一个完整的线程，在进入睡眠的时候释放资源，使得别的线程可以开始抢占资源并运行。

public class ThreadPoolTest01 {

public static void main(String[] args) {

ThreadPoolExecutor threadPool = new ThreadPoolExecutor(2,

5,

300,

TimeUnit.MILLISECONDS,

new ArrayBlockingQueue<Runnable>(2)

);

for (int i = 0; i < 5; i++) {

//String.valueOf(i) 将整数转为字符串

MyTask myTask = new MyTask(String.valueOf(i));

//提交任务给线程池

threadPool.execute(myTask);

}

}

}

上述的代码块执行提交五个任务到线程池的操作，实际过程是这样的：

提交第1个任务到线程池：poolSize=0<corePoolSize 所以会为改任务创建一个线程对象并

启动线程。

提交第2个任务到线程池：poolSize=1<corePoolSize 所以会为改任务创建一个线程对象并

启动线程。

提交第3个任务到线程池：poolSize=2!<corePoolSize 此时会将任务放到等待队列（1）。

提交第4个任务到线程池：poolSize=2!<corePoolSize 没有空闲线程，此时会将任务放到等待队列（2）。

提交第5个任务到线程池：poolSize=2!<corePoolSize 没有空闲线程，判断此时

maximumPoolSize>poolSize>=corePoolSize。没有创建一个线程。

通过ThreadPoolExecutor 类new出线程池对象threadPool ，然后通过参数定义实际的线程池的各项属性，包括了：线程池初始大小，线程池最大线程数，空闲线程存活的最长时间，时间单位，等待队列；

ExecutorService（ThreadPoolExecutor的顶层接口）使用线程池中的线程执行每个提交的任务，通常我们使用Executors的工厂方法来创建ExecutorService。

线程池解决了两个不同的问题：

1. 提升性能：它们通常在执行大量异步任务时，由于减少了每个任务的调用开销，并且它们提供了一种限制和管理资源（包括线程）的方法，使得性能提升明显；

2. 统计信息：每个ThreadPoolExecutor保持一些基本的统计信息，例如完成的任务数量

 Executors.newCachedThreadPool（无界线程池，自动线程回收）

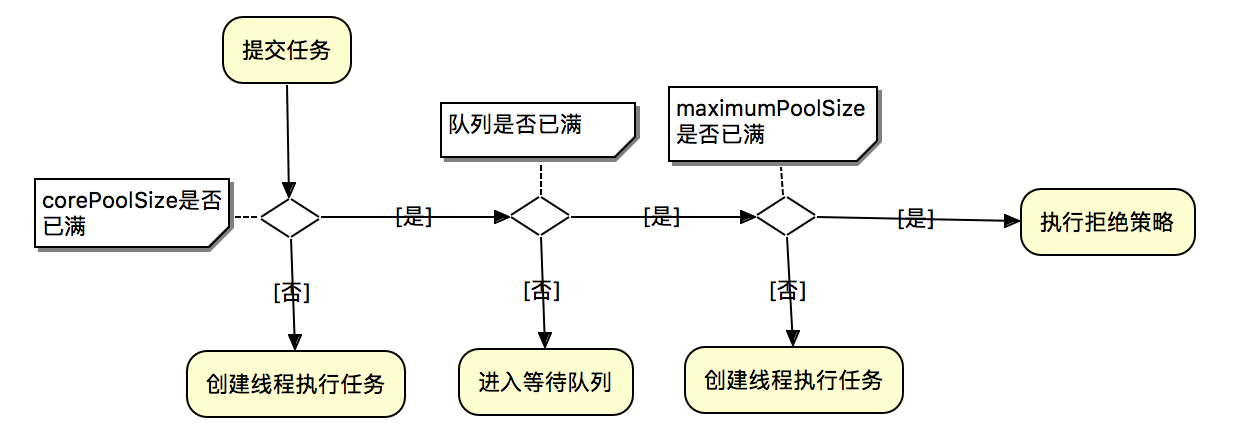
 Executors.newFixedThreadPool（固定大小的线程池）；

 Executors.newSingleThreadExecutor（单一后台线程）；

**一.Core and maximum pool sizes 核心和最大线程池数量**

线程池执行器将会根据corePoolSize和maximumPoolSize自动地调整线程池大小。

当在execute(Runnable)方法中提交新任务并且少于corePoolSize线程正在运行时，即使其他工作线程处于空闲状态，也会创建一个新线程来处理该请求。 如果有多于corePoolSize但小于maximumPoolSize线程正在运行，则仅当队列已满时才会创建新线程。 通过设置corePoolSize和maximumPoolSize相同，您可以创建一个固定大小的线程池。 通过将maximumPoolSize设置为基本上无界的值，例如Integer.MAX\_VALUE，您可以允许池容纳任意数量的并发任务。 通常，核心和最大池大小仅在构建时设置，但也可以使用setCorePoolSize和setMaximumPoolSize进行动态更改。



**二、prestartCoreThread 核心线程预启动**

prestartCoreThread() 和 prestartAllCoreThreads() 方法动态调整。  
 如果使用非空队列构建池，则可能需要预先启动线程。

**三、ThreadFactory 线程工厂**

新线程使用ThreadFactory创建。 如果未另行指定，则使用Executors.defaultThreadFactory默认工厂，使其全部位于同一个ThreadGroup中，并且具有相同的NORM\_PRIORITY优先级和非守护进程状态。

通过提供不同的ThreadFactory，您可以更改线程的名称，线程组，优先级，守护进程状态等。如果ThreadCactory在通过从newThread返回null询问时未能创建线程，则执行程序将继续，但可能无法执行任何任务。

线程应该有modifyThread权限。 如果工作线程或使用该池的其他线程不具备此权限，则服务可能会降级：配置更改可能无法及时生效，并且关闭池可能会保持可终止但尚未完成的状态。

**四、Keep-alive times 线程存活时间**

如果线程池当前拥有超过corePoolSize的线程，那么多余的线程在空闲时间超过keepAliveTime时会被终止 ( 请参阅getKeepAliveTime(TimeUnit) )。这提供了一种在不积极使用线程池时减少资源消耗的方法。

如果池在以后变得更加活跃，则应构建新线程。 也可以使用方法setKeepAliveTime(long，TimeUnit)进行动态调整。

**五、Queuing 队列**

# BlockingQueu用于存放提交的任务，队列的实际容量与线程池大小相关联。

 如果当前线程池任务线程数量小于核心线程池数量，执行器总是优先创建一个任务线程，而不是从线程队列中取一个空闲线程。

 如果当前线程池任务线程数量大于核心线程池数量，执行器总是优先从线程队列中取一个空闲线程，而不是创建一个任务线程。

 如果当前线程池任务线程数量大于核心线程池数量，且队列中无空闲任务线程，将会创建一个任务线程，直到超出maximumPoolSize，如果超时maximumPoolSize，则任务将会被拒绝。

**六、Rejected tasks 拒绝任务**

拒绝任务有两种情况：1. 线程池已经被关闭；2. 任务队列已满且maximumPoolSizes已满；无论哪种情况，都会调用RejectedExecutionHandler的rejectedExecution方法。

预定义了四种处理策略：

1. **AbortPolicy**：默认测策略，抛出RejectedExecutionException运行时异常；

2. **CallerRunsPolicy**：这提供了一个简单的反馈控制机制，可以减慢提交新任务的速度；

3. **DiscardPolicy**：直接丢弃新提交的任务；

4. **DiscardOldestPolicy**：如果执行器没有关闭，队列头的任务将会被丢弃，然后执行器重新尝试执行任务（如果失败，则重复这一过程）；  
 我们可以自己定义RejectedExecutionHandler，以适应特殊的容量和队列策略场景中

# 三种主要的队列策略：

1. **Direct handoffs 直接握手队列**  
 Direct handoffs 的一个很好的默认选择是 SynchronousQueue，它将任务交给线程而不需要保留。这里，如果没有线程立即可用来运行它，那么排队任务的尝试将失败，因此将构建新的线程。  
 此策略在处理可能具有内部依赖关系的请求集时避免锁定。Direct handoffs 通常需要无限制的maximumPoolSizes来避免拒绝新提交的任务。 **但得注意，当任务持续以平均提交速度大余平均处理速度时，会导致线程数量会无限增长问题。**

2. **Unbounded queues 无界队列**  
 当所有corePoolSize线程繁忙时，使用无界队列（例如，没有预定义容量的LinkedBlockingQueue）将导致新任务在队列中等待，从而导致maximumPoolSize的值没有任何作用。当每个任务互不影响，完全独立于其他任务时，这可能是合适的; 例如，在网页服务器中， 这种队列方式可以用于平滑瞬时大量请求。**但得注意，当任务持续以平均提交速度大余平均处理速度时，会导致队列无限增长问题。**

3. **Bounded queues 有界队列**  
 一个有界的队列（例如，一个ArrayBlockingQueue）和有限的maximumPoolSizes配置有助于防止资源耗尽，但是难以控制。队列大小和maximumPoolSizes需要 **相互权衡**：

 使用大队列和较小的maximumPoolSizes可以最大限度地减少CPU使用率，操作系统资源和上下文切换开销，但会导致人为的低吞吐量。如果任务经常被阻塞（比如I/O限制），那么系统可以调度比我们允许的更多的线程。

 使用小队列通常需要较大的maximumPoolSizes，这会使CPU更繁忙，但可能会遇到不可接受的调度开销，这也会降低吞吐量。

代码示例02：

public class ThreadPoolTest02 {

public static void main(String[] args) {

//创建线程池的方式二：Executors类

//1）等待队列是一个0容量的队列，当提交大批量任务，提交的速度比处理任务的速度快

//会导致系统资源被耗光。从而使得系统奔溃。

ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();

//2)使用一个无界队列，队列大小是无穷大。堆积大量的任务，从而导致oom （out of memery）

// ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(3);

for (int i = 0; i < 5; i++) {

//String.valueOf(i) 将整数转为字符串

MyTask2 myTask = new MyTask2(String.valueOf(i));

//提交任务给线程池

newCachedThreadPool.execute(myTask);

}

如果线程任务有返回值如何处理？

示例代码导入：

class MyTask3 implements Callable<Integer> {//Callable这个接口和 call() 方法可以使得Integer得到返回值；

private String name;

public MyTask3(String name) {

this.name = name;

}

@Override

public Integer call() throws Exception {

Thread.sleep(3000);

int sum = 0;

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

sum += i;

}

return sum;

}

}

public class ThreadPoolTest03 {

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

// 创建线程池的方式二：Executors类

// 1）等待队列是一个0容量的队列，当提交大批量任务，提交的速度比处理任务的速度快

// 会导致系统资源被耗光。从而使得系统奔溃。

ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();

// 2)使用一个无界队列，队列大小是无穷大。堆积大量的任务，从而导致oom （out of memery）

// ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(3);

// String.valueOf(i) 将整数转为字符串

MyTask3 myTask = new MyTask3(String.valueOf(1));

// 提交任务给线程池

Future<Integer> result = newCachedThreadPool.submit(myTask);

//可以去做其他事情

System.out.println("我去洗脚，洗完脚再过来取和的值");

Thread.sleep(1000);

//得到返回值

Integer sum = result.get();

System.out.println(sum);

}

}