线程池

线程池简介

1. **线程池的概念**：

线程池就是首先创建一些线程，它们的集合称为线程池。使用线程池可以很好地提高性能，线程池在系统启动时即创建大量空闲的线程，程序将一个任务传给线程池，线程池就会启动一条线程来执行这个任务，执行结束以后，该线程并不会死亡，而是再次返回线程池中成为空闲状态，等待执行下一个任务。它的主要特点为：**线程复用、控制最大并发数、管理线程**。

2. 线程池的工作机制：

在线程池的编程模式下，任务是提交给整个线程池，而不是直接提交给某个线程，线程池在拿到任务后，就在内部寻找是否有空闲的线程，如果有，则将任务交给某个空闲的线程。

一个线程同时只能执行一个任务，但可以同时向一个线程池提交多个任务。

3. 使用线程池的原因

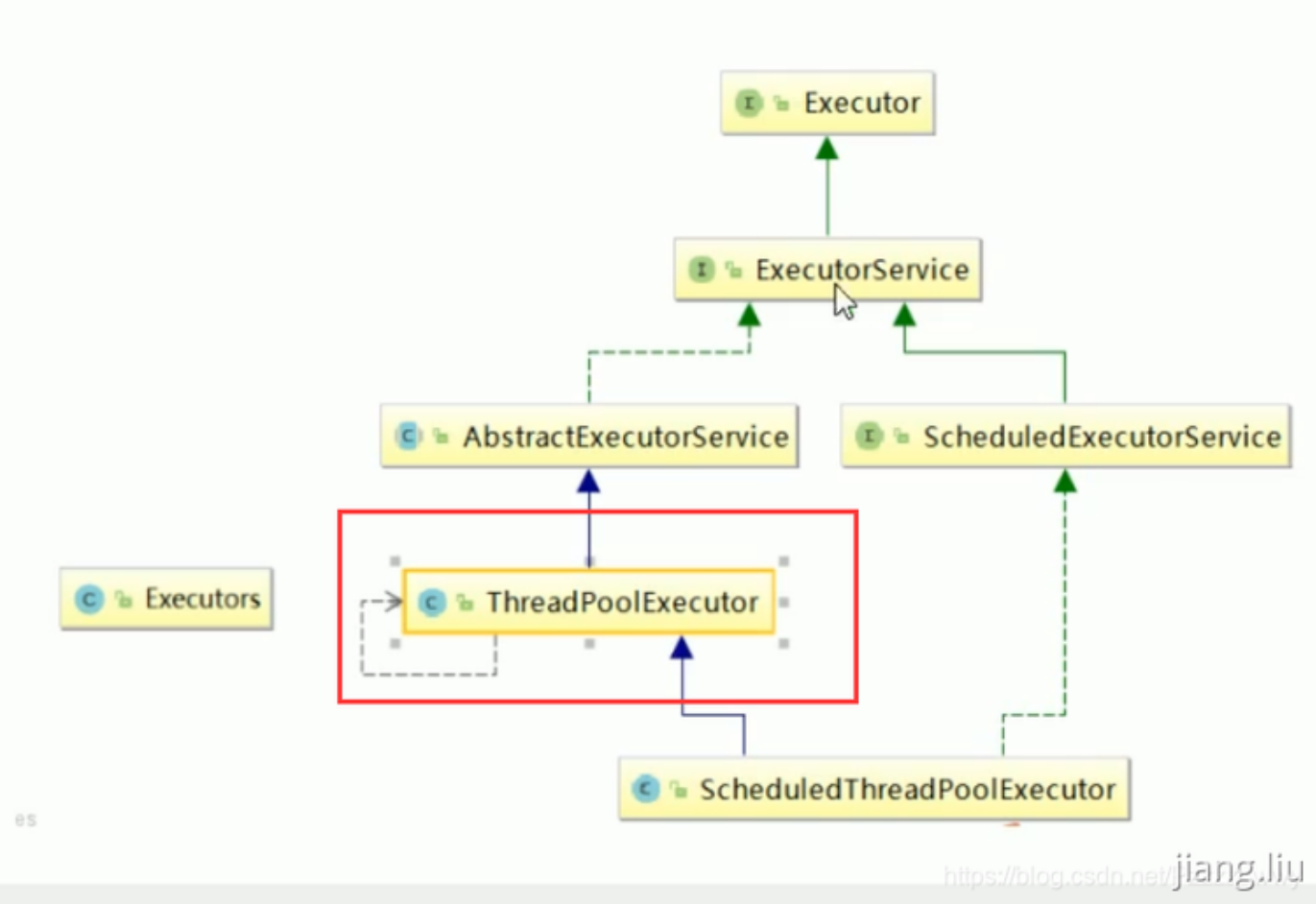
多线程运行时，系统不断的启动和关闭新线程，成本非常高，会过渡消耗系统资源，以及过度切换线程的危险，从而可能导致系统资源的崩溃。这时，线程池就是最好的选择了。

1）降低资源消耗。重复利用已创建好的线程。

2）提高响应速度。不需要等待线程创建。

3）提高线程的管理性。统一分配、调优，监控。

继承图



线程池3大方法：

Java 复制代码

```
1  ExecutorService threadPool = Executors.newFixedThreadPool(5); //固定容量
2  ExecutorService threadPool = Executors.newSingleThreadExecutor();   //单例的、单个线程的线程池
3  ExecutorService threadPool = Executors.newCachedThreadPool();   //缓存的 即超出就自动创建线程的
```

/第一种，固定容量线程池，类似银行5个窗口，每次只能5个人办理，其他人阻塞等待。

```java
public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);// 看源码是LinkedBlockingQueue<Runnable>()

        for (int i = 1; i <= 10; i++) {
            final int ig = i;
            executor.execute(()-> {
                System.out.println(Thread.currentThread().getName() + "\t" + "办理业务");
            });
        }
        executor.shutdown();  // 关闭线程池

    }
```

第二种，单例线程池，线程池只有一个线程，类似银行只有1个窗口，每次只能1个人办理，其他人阻塞等待。

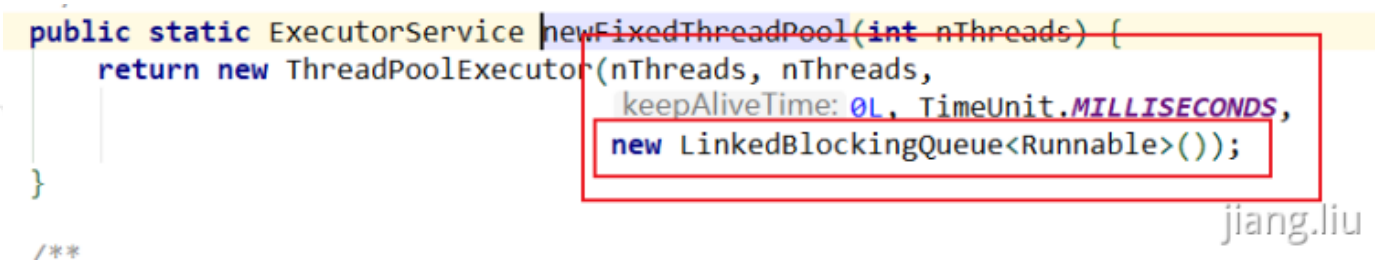
```java
ExecutorService executor = Executors.newSingleThreadExecutor(); //
```

第三种，缓存线程池，线程池随着线程的数量扩容，类似银行N个窗口，有N个线程，同时执行。线程多的时候自动扩容。

```java
ExecutorService executor = Executors.newCachedThreadPool();
```

接下来看看线程池的源码：

首先点开newFixedThreadPool()的源码可以看到：



接下来点进去newSingleThreadExecutor()的源码可以看到：

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor( corePoolSize: 1, maximumPoolSize: 1,
                                 keepAliveTime: 0L, TimeUnit.MILLISECONDS,
                                 new LinkedBlockingQueue<Runnable>()));
}

/**
```


接下来点进去newCachedThreadPool()的源码可以看到：

```
@NotNull()
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor( corePoolSize: 0, Integer.MAX_VALUE,
                                   keepAliveTime: 60L, TimeUnit.SECONDS,
                                   new SynchronousQueue<Runnable>());
}

/**
 * Creates a thread pool that creates new threads as needed, but
```


综上所述，返回的实际上只是一个`ThreadPoolExecutor`（可以看看继承图），利用构造器传入的不同的参数而已，而且我们也能发现底层是阻塞队列。

同时说明我们也可以通过ThreadPoolExecutor`来创建线程池，Executors只是一个创建线程池的工具类，实际上返回的还是ThreadPoolExecutor。

接着我们继续点开`ThreadPoolExecutor`：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
}

/**
```


接着再点进这this，我们可以看到它有七个参数：

```java
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

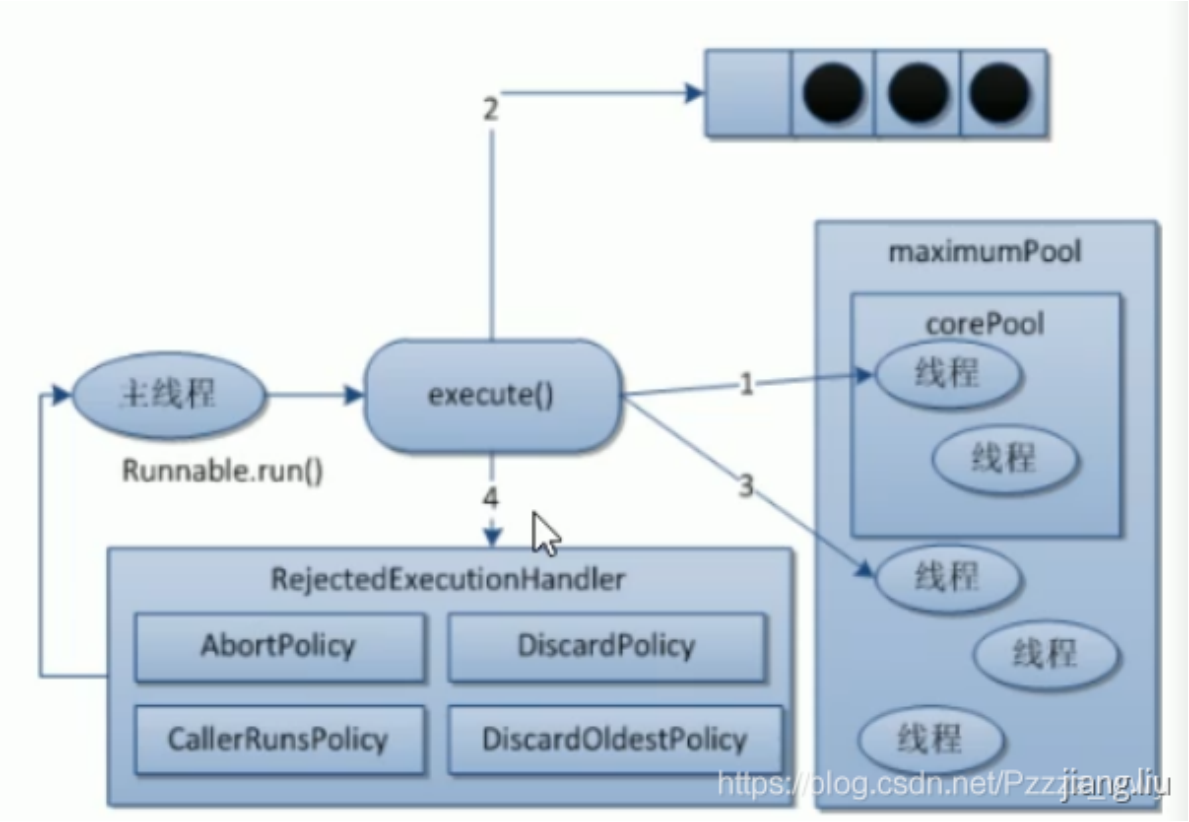

threadPoolExecutor七大参数

七大参数的解释：

1. corePoolSize：线程池中的常驻核心线程数。
2. maximumPoolSize：线程池中能够容纳同时执行的最大线程数，此值必须大于等于1。
3. keepAliveTime：多余的空闲线程的存活时间；当前池中线程数超过corePoolSize时，当空闲时间达到keepLiveTime时，多余线程会被销毁直到剩下corePoolSize个线程为止。
4. unit：keepLiveTime的单位（年月日时分秒）。
5. workQueue：任务队列，被提交但尚未被执行的任务。
6. threadFactory：表示生成线程池中工作线程的线程工厂，用于创建线程，一半默认的即可。
7. handler：拒绝策略，表示当队列满了，并且工作线程大于等于线程池的最大线程数（maximumPoolSize）时如何来拒绝请求执行的runnable的策略。

通俗理解为：线程池相当于一家银行，窗口相当于线程，银行有最多有5个受理窗口，但是常用的却只有2个，而候客区就相当于我们的阻塞队列（**BlockingQueue**），那当我们的阻塞队列满了之后，**handle**拒绝策略出来了，相当于银行门口立了块牌子，上面写着不办理业务了！然后当客户都办理的差不多了，此时多出来(在corePool的基础上扩容的窗口)的窗口在经过**keepAliveTime**的时间后就关闭了，重新恢复到**corePool**个受理窗口。

线程池的底层工作原理图



线程池的工作流程

首先线程池接收到任务，先判断核心线程数是否满了，没有满接客，满了就放到阻塞队列，如果阻塞队列没满，这些任务放在阻塞队列，如果满了，就扩容线程数到最大线程数，如果最大线程数也满了，就是我们的拒绝策略。

这就是线程池四大步骤。 **接客、放入队列，扩容线程，拒绝策略**!

1、在创建了线程池后，开始等待请求。
2、当调用`execute()`方法添加一个请求任务时，线程池会做出如下判断：
2.1如果正在运行的线程数量小于`corePoolSize`，那么马上创建线程运行这个任务；
2.2如果正在运行的线程数量大于或等于`corePoolSize`，那么将这个任务放入队列；
2.3如果这个时候队列满了且正在运行的线程数量还小于`maximumPoolSize`，那么还是要创建非核心线程立刻运行这个任务；
2.4如果队列满了且正在运行的线程数量大于或等于`maximumPoolSize`，那么线程池会启动饱和拒绝策略来执行。
3、当一个线程完成任务时，它会从队列中取下一个任务来执行。
4、当一个线程无事可做超过一定的时间（`keepAliveTime`）时，线程会判断：
如果当前运行的线程数大于`corePoolSize`，那么这个线程就被停掉。
所以线程池的所有任务完成后，它最终会收缩到`corePoolSize`的大小。

那我们实际开发中怎么进行配置呢?

为什么呢?

对于newSingleThreadExecutor(); 而言LinkedBlockQueue的长度是Integer.MAX_VALUE,

对于newCachedThreadPool()而言，maximumPool的值竟然为Integer.MAX_VALUE！！

两者均会导致OOM异常!

自定义线程池

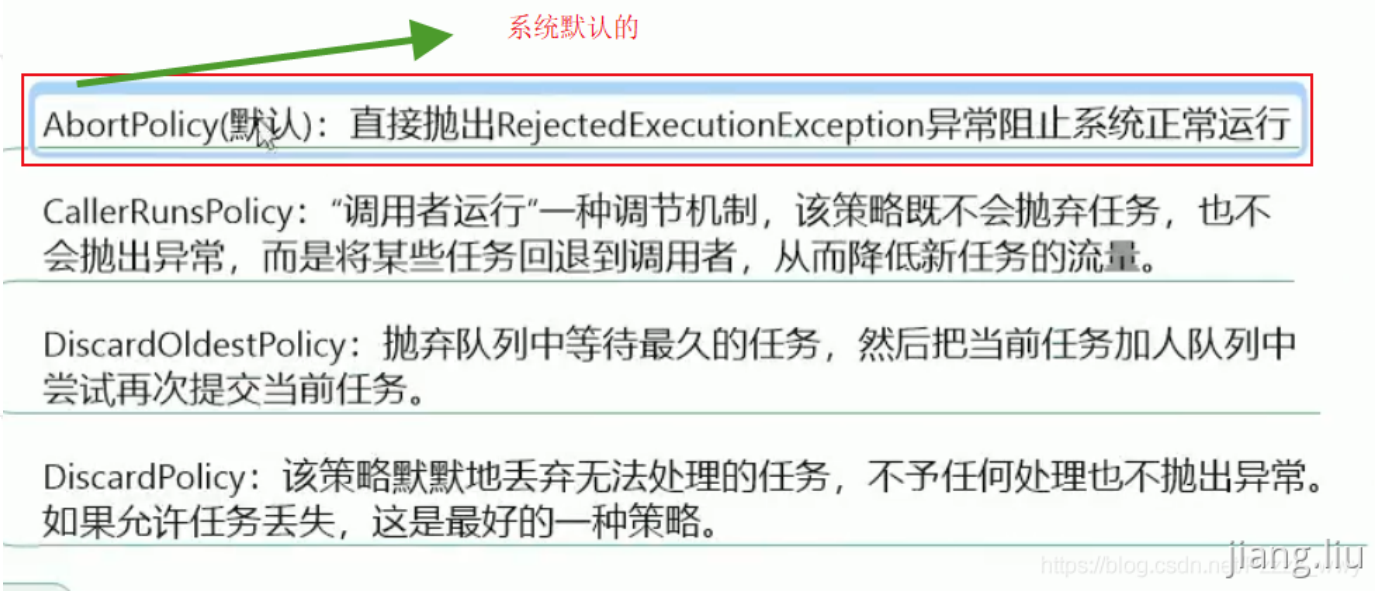
```java
    public static void main(String[] args) {
        ExecutorService threadPool = new ThreadPoolExecutor(
                2, // 常驻线程数
                5, // 最大线程数
                2L, // 空闲回收线程的时间
                TimeUnit.SECONDS, // 空闲时间单位
                new LinkedBlockingQueue<>(3), // 使用Executors工具类自带的线程池实现中的阻塞队列，
                                              // 阻塞队列的数量不定义则使用默认值Integer.MAX_VALUE。
                Executors.defaultThreadFactory(), // 使用Executors工具类自带的线程池实现中的工厂
                new ThreadPoolExecutor.AbortPolicy()); //JDK默认的拒绝策略

        try {
            //模拟10个用户办理业务，但是只有5个受理窗口
            for (int i = 0; i < 9; i++) {
                threadPool.execute(() -> {
                    System.out.println(Thread.currentThread().getName() + "\t" + "办理业务");
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            threadPool.shutdown(); //关闭线程池
        }
    }
```

threadPool 是我们自定义的线程池，连接过上面的参数的应该都知道，该线程池最大支持的并发量就应该是maximumPool+Queue的大小，即5+3=8，而超过了大小之后就会报错：`java.util.concurrent.RejectedExecutionException` 拒绝执行异常。

```
pool-1-thread-1 办理业务
pool-1-thread-4 办理业务
pool-1-thread-3 办理业务
pool-1-thread-2 办理业务
pool-1-thread-3 办理业务
pool-1-thread-4 办理业务
pool-1-thread-1 办理业务
pool-1-thread-5 办理业务
java.util.concurrent.RejectedExecutionException: Task juc.MyThreadPoolDemo$$Lambda$1/1078694789@3b9a45b3 rejected from java.util.concurrent.ThreadPoolExecutor@7699a589[Running, pool size = 5, active
    at juc.MyThreadPoolDemo.main(MyThreadPoolDemo.java:28)
```

拒绝策略（4种）

当线程池达到最大线程数，并且等待队列也全部满了，无法处理新的任务请求，就需要合理的拒绝策略来处理。



注，第四种应该是直接丢弃提交的这个新任务。

将上面代码的拒绝策略改成第二种`new ThreadPoolExecutor.CallerRunsPolicy()`，返回结果如下：

```
pool-1-thread-1 办理业务
pool-1-thread-5 办理业务
pool-1-thread-4 办理业务
main    办理业务
pool-1-thread-3 办理业务
pool-1-thread-2 办理业务
pool-1-thread-4 办理业务
pool-1-thread-5 办理业务
pool-1-thread-1 办理业务
```

将多余的无法处理的任务回退给调用者。

第三种`new ThreadPoolExecutor.DiscardOldestPolicy()`：

```
pool-1-thread-1 办理业务
pool-1-thread-2 办理业务
pool-1-thread-3 办理业务
pool-1-thread-2 办理业务
pool-1-thread-3 办理业务
pool-1-thread-1 办理业务
pool-1-thread-2 办理业务
pool-1-thread-4 办理业务
pool-1-thread-5 办理业务
```

丢弃无法处理的任务，不报错。

第四种`new ThreadPoolExecutor.DiscardPolicy()`：

```
pool-1-thread-1 办理业务
pool-1-thread-2 办理业务
pool-1-thread-3 办理业务
pool-1-thread-1 办理业务
pool-1-thread-2 办理业务
pool-1-thread-4 办理业务
pool-1-thread-3 办理业务
pool-1-thread-5 办理业务
```

以上策略均继承自`RejectedExecutionHandler`接口。

最后提一句怎么设置maximumPoolSize合理，

Plain Text 复制代码

```
// 打印机器cpu核数
System.out.println(Runtime.getRuntime().availableProcessors()); //8核
```

一般设置为CPU核数加1，例如8核的话，设置为9