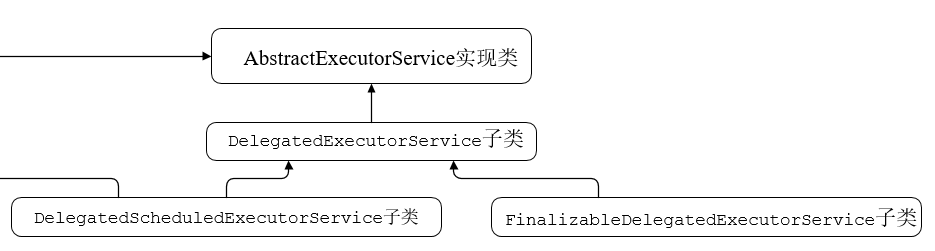
Executors工厂类

delegate 英[ˈdelɪgət] 美[ˈdɛlɪˌɡet]n. 代表，代表团成员; vt. 委派代表; 授权给; [法律] 债务转移;

# DelegatedExecutorService

**DelegatedExecutorService、DelegatedScheduledExecutorService、FinalizableDelegatedExecutorService**



**DelegatedExecutorService源码分析：**

/\*\*

\* **A wrapper class** that exposes only the ExecutorService methods

\* of an ExecutorService implementation.

\*/

static class **DelegatedExecutorService** extends **AbstractExecutorService** {

private final ExecutorService e;

DelegatedExecutorService(ExecutorService executor) { e = executor; }

public void execute(Runnable command) { e.execute(command); }

public void shutdown() { e.shutdown(); }

public List<Runnable> shutdownNow() { return e.shutdownNow(); }

public boolean isShutdown() { return e.isShutdown(); }

public boolean isTerminated() { return e.isTerminated(); }

public boolean awaitTermination(long timeout, TimeUnit unit)

throws InterruptedException {

return e.awaitTermination(timeout, unit);

}

public Future<?> submit(Runnable task) {

return e.submit(task);

}

public <T> Future<T> submit(Callable<T> task) {

return e.submit(task);

}

public <T> Future<T> submit(Runnable task, T result) {

return e.submit(task, result);

}

public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)

throws InterruptedException {

return e.invokeAll(tasks);

}

public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,

long timeout, TimeUnit unit)

throws InterruptedException {

return e.invokeAll(tasks, timeout, unit);

}

public <T> T invokeAny(Collection<? extends Callable<T>> tasks)

throws InterruptedException, ExecutionException {

return e.invokeAny(tasks);

}

public <T> T invokeAny(Collection<? extends Callable<T>> tasks,

long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException {

return e.invokeAny(tasks, timeout, unit);

}

}

**DelegatedScheduledExecutorService**源码分析：

/\*\*

\* A wrapper class that exposes only the ScheduledExecutorService

\* methods of a ScheduledExecutorService implementation.

\*/

static class **DelegatedScheduledExecutorService e**xtends **DelegatedExecutorService**

implements **ScheduledExecutorService** {

private final ScheduledExecutorService e;

**DelegatedScheduledExecutorService(ScheduledExecutorService executor) {**

**super(executor);**

**e = executor;**

**}**

public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit) {

return e.schedule(command, delay, unit);

}

public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit) {

return e.schedule(callable, delay, unit);

}

public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit) {

return e.scheduleAtFixedRate(command, initialDelay, period, unit);

}

public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit) {

return e.scheduleWithFixedDelay(command, initialDelay, delay, unit);

}

}

**FinalizableDelegatedExecutorService**源码:

static class **FinalizableDelegatedExecutorService** extends **DelegatedExecutorService** {

FinalizableDelegatedExecutorService(ExecutorService executor) {

super(executor);

}

protected void finalize() {

super.shutdown();

}

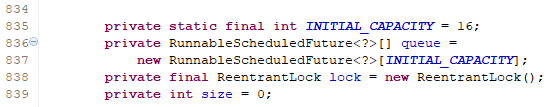
}

# DelayedWorkQueue

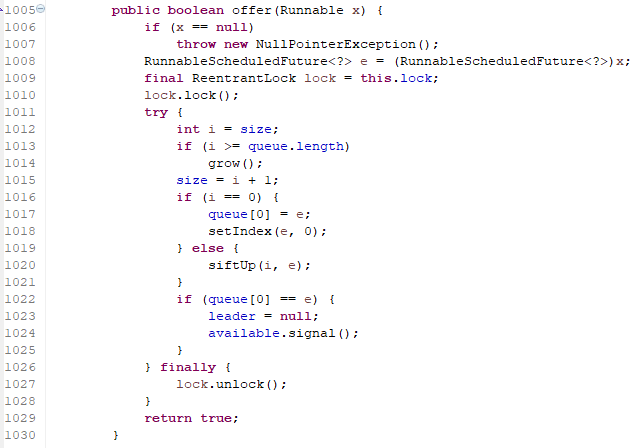
内部是一个**RunnableScheduledFuture**数组。

static class **DelayedWorkQueue** extends **AbstractQueue**<Runnable>

implements **BlockingQueue**<Runnable>



offer方法：



# ThreadFactory接口

## 继承关系

public interface **ThreadFactory** 线程工厂；存在于java.util.concurrent包中。

实现类：Executors内部的**DefaultThreadFactory**和**PrivilegedThreadFactory**(继承DefaultThreadFactory)。

## 功能介绍

An object that creates new threads on demand. Using **thread** factories removes hardwiring of calls to new Thread, enabling applications to use special thread subclasses, priorities, etc.

The simplest implementation of this interface is just:

class **SimpleThreadFactory** implements **ThreadFactory** {

public Thread newThread(Runnable r) {

return new Thread(r);

}

}

## 只定义一个方法：

**Thread newThread(Runnable r)**

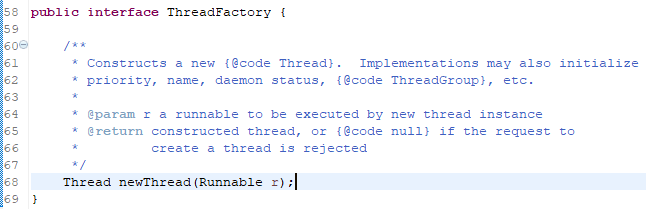
Constructs a new Thread. Implementations may also initialize priority, name, daemon status, ThreadGroup, etc.

创建一个新的线程，可以对线程的优先级、名称、是否守护线程、线程组等进行初始化操作。

**Parameters**: r - a runnable to be executed by new thread instance

**Returns**: constructed thread, or null if the request to create a thread is rejected

## ThreadFactory接口的源代码：



## Executors.defaultThreadFactory() 默认的线程工厂

The **Executors.defaultThreadFactory()** method provides a more useful simple implementation, that sets the created thread context to known values before returning it.

public static **ThreadFactory** **defaultThreadFactory**()

Returns a default thread factory used to create new threads. This factory creates all new threads used by an Executor in the same ThreadGroup. If there is a **SecurityManager**, it uses the group of **System.getSecurityManager()**, else the group of the thread invoking this defaultThreadFactory method. Each new thread is created as a non-daemon thread with priority set to the smaller of **Thread.NORM\_PRIORITY** and the maximum priority permitted in the thread group. New threads have names accessible via **Thread.getName()** of **pool-N-thread-M**, *where N is the sequence number of this factory, and M is the sequence number of the thread created by this factory*.

**Executors.defaultThreadFactory()：源代码**



## DefaultThreadFactory默认线程工厂的源代码(很简单)

**DefaultThreadFactory**是Executors类中的一个内部类(default的,对外不暴露)。

实现了ThreadFactory接口，并实现了newThread方法，内部对线程的名字、线程组、优先级、守护性进行了初始设置。其中线程名字时利用线程池编号和当前线程编号组合而成：

pool-1-thread-2 ；这里面线程池编号和线程编号都是利用原子类AtomicInteger实现的。

/\*\*

\* The default thread factory

\*/

static class **DefaultThreadFactory** implements **ThreadFactory** {

**private static final AtomicInteger poolNumber = new AtomicInteger(1);**

**private final ThreadGroup group;**

**private final AtomicInteger threadNumber = new AtomicInteger(1);**

**private final String namePrefix;**

DefaultThreadFactory() {

SecurityManager s = System.getSecurityManager();//线程安全管理

group = (s != null) ? s.getThreadGroup() :

Thread.currentThread().getThreadGroup();

**namePrefix**=**"pool-"+ poolNumber.getAndIncrement() +"-thread-";**

}

public Thread newThread(Runnable r) {

Thread t = new Thread(**group**, **r**, **namePrefix + threadNumber.getAndIncrement()**,0);

if (t.isDaemon())

t.setDaemon(false);//默认非守护

if (t.getPriority() != Thread.NORM\_PRIORITY)

t.setPriority(Thread.NORM\_PRIORITY);//默认优先级为5

return t;

}

}

System.out.println(**Thread.currentThread().getName());线程输出的名字都是来自默认线程工厂的初始化：namePrefix + threadNumber.getAndIncrement()**

pool-1-thread-2

pool-1-thread-4

pool-1-thread-3

pool-1-thread-1

## PrivilegedThreadFactory源代码分析

**PrivilegedThreadFactory**继承于DefaultThreadFactory；覆写了newThread方法。

/\*\*

\* Thread factory capturing access control context and class loader

\*/

static class **PrivilegedThreadFactory** extends **DefaultThreadFactory** {

private final **AccessControlContext** acc;

private final **ClassLoader** ccl;

PrivilegedThreadFactory() {

super();

SecurityManager sm = System.getSecurityManager();

if (sm != null) {

// Calls to getContextClassLoader from this class

// never trigger a security check, but we check

// whether our callers have this permission anyways.

sm.checkPermission(SecurityConstants.GET\_CLASSLOADER\_PERMISSION);

// Fail fast

sm.checkPermission(new RuntimePermission("setContextClassLoader"));

}

this.acc = AccessController.getContext();

this.ccl = Thread.currentThread().getContextClassLoader();

}

public Thread **newThread**(final Runnable r) {

return super.newThread(new Runnable() {

public void run() {

AccessController.doPrivileged(new PrivilegedAction<Void>() {

public Void run() {

Thread.currentThread().setContextClassLoader(ccl);

r.run();

return null;

}

}, acc);

}

});

}

}