

# Executor接口

## void execute(Runnable command)

异步执行command，避免重复的new Thread消耗资源，更便于统一管理。

# ExecotorService接口

## void shutdown()

不再接受新任务，等待所有任务结束后关闭ExectorService。

## List<Runnable> shutdownNow()

直接关闭ExectorService，正在执行的线程interrupt并把列表返回。

## boolean isShutdown()

判断ExectorService是否为关闭状态。

## boolean isTerminated()

判断ExectorService关闭后，任务是否全部完成了。

## boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException

等待单位是unit的timeout后返回ExectorService是否全部执行完。

## <T> Future<T> submit(Callable<T> task)

提交task任务，得到Future对象，可以得到任务执行的结果。

## <T> Future<T> submit(Runnable task, T result)

提交task任务，成功后调用future.get()将返回result。

## Future<?> submit(Runnable task)

提交task任务，成功后调用future.get()将返回null。

## <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException

批量提交任务并获得他们的future，Task列表与Future列表一一对应（全部执行完返回）。（适用于需要查找到所有信息的需求）

## <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,long timeout, TimeUnit unit) throws InterruptedException

批量提交任务并获得他们的future，并限定处理所有任务的时间。

<T> T invokeAny(Collection<? extends Callable<T>> tasks)  
 throws InterruptedException, ExecutionException

批量提交任务并获得一个已经成功执行的任务的结果(只要成功执行一个就返回)。（适用于只要查找到一条信息就返回的需求）

<T> T invokeAny(Collection<? extends Callable<T>> tasks,long timeout, TimeUnit unit)  
 throws InterruptedException, ExecutionException, TimeoutException

批量提交任务，并限定处理所有任务的时间。返回一个已经成功执行的任务的结果(只要成功执行一个就返回)。

# Runnable接口

## public abstract void run()

创建一个线程重写run方法。

# Thread

初始化全部调用的init方法。

private void init(ThreadGroup g, Runnable target, String name,  
 long stackSize, AccessControlContext acc,  
 boolean inheritThreadLocals) {  
 if (name == null) {  
 throw new NullPointerException("name cannot be null");  
 }  
  
 this.name = name;  
  
 Thread parent = *currentThread*();  
 SecurityManager security = System.*getSecurityManager*();  
 if (g == null) {  
 /\* Determine if it's an applet or not \*/  
  
 /\* If there is a security manager, ask the security manager  
 what to do. \*/  
 if (security != null) {  
 g = security.getThreadGroup();  
 }  
  
 /\* If the security doesn't have a strong opinion of the matter  
 use the parent thread group. \*/  
 if (g == null) {  
 g = parent.getThreadGroup();  
 }  
 }  
  
 /\* checkAccess regardless of whether or not threadgroup is  
 explicitly passed in. \*/  
 g.checkAccess();  
  
 /\*  
 \* Do we have the required permissions?  
 \*/  
 if (security != null) {  
 if (*isCCLOverridden*(getClass())) {  
 security.checkPermission(*SUBCLASS\_IMPLEMENTATION\_PERMISSION*);  
 }  
 }  
  
 g.addUnstarted();  
  
 this.group = g;  
 this.daemon = parent.isDaemon();  
 this.priority = parent.getPriority();  
 if (security == null || *isCCLOverridden*(parent.getClass()))  
 this.contextClassLoader = parent.getContextClassLoader();  
 else  
 this.contextClassLoader = parent.contextClassLoader;  
 this.inheritedAccessControlContext =  
 acc != null ? acc : AccessController.*getContext*();  
 this.target = target;  
 setPriority(priority);  
 if (inheritThreadLocals && parent.inheritableThreadLocals != null)  
 this.inheritableThreadLocals =  
 ThreadLocal.*createInheritedMap*(parent.inheritableThreadLocals);  
 /\* Stash the specified stack size in case the VM cares \*/  
 this.stackSize = stackSize;  
  
 /\* Set thread ID \*/  
 tid = *nextThreadID*();  
}