

# 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当前是否为关闭状态。

## boolean isTerminated()

判断ExectorService当前是否在关闭后，任务全部完成了。

## 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

## 所有构造方法全部调用init方法。

*/\*\*  
 \* 初始化一个线程  
 \*  
 \** ***@param*** *g 线程数组  
 \** ***@param*** *target 运行的任务对象  
 \** ***@param*** *name 线程名字  
 \** ***@param*** *stackSize 线程堆栈大小  
 \** ***@param*** *acc 系统资源访问决策  
 \** ***@param*** *inheritThreadLocals 是否继承线程池  
 \*/*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;  
 /\*返回当前线程的引用\*/  
 Thread parent = currentThread();  
 /\*Java安全管理器，这边不做详细解释，画个重点\*/  
 SecurityManager security = System.*getSecurityManager*();  
 if (g == null) {  
 if (security != null) {  
 /\*寻找存在的线程组\*/  
 g = security.getThreadGroup();  
 }  
 if (g == null) {  
 /\*自建\*/  
 g = parent.getThreadGroup();  
 }  
 }  
 /\*检查是否有修改线程的权限，否则抛出异常\*/  
 g.checkAccess();  
 /\*检查是否有必要的权限，否则抛出异常\*/

if (security != null) {  
 if (isCCLOverridden(getClass())) {  
 security.checkPermission

(SUBCLASS\_IMPLEMENTATION\_PERMISSION);  
 }  
 }  
 /\*线程组数量增加\*/  
 g.addUnstarted();  
 this.group = g;  
 /\*是否是守护线程\*/  
 this.daemon = parent.isDaemon();  
 /\*得到线程优先级\*/  
 this.priority = parent.getPriority();  
 /\*线程的类加载器设置，这点不太明白\*/  
 if (security == null || isCCLOverridden(parent.getClass()))  
 this.contextClassLoader = parent.getContextClassLoader();  
 else  
 this.contextClassLoader = parent.contextClassLoader;  
 /\*系统资源的访问决策\*/  
 this.inheritedAccessControlContext =  
 acc != null ? acc : AccessController.*getContext*();  
 this.target = target;  
 setPriority(priority);  
 if (inheritThreadLocals && parent.inheritableThreadLocals != null)  
 /\*拷贝一份\*/  
 this.inheritableThreadLocals = ThreadLocal.

*createInheritedMap*(parent.inheritableThreadLocals);  
 this.stackSize = stackSize;  
 /\*线程id设置\*/  
 tid = nextThreadID();  
}

整个源码看下来，就是一个放入线程组和检测能否创建线程的过程。关于类的加载器问题，也不是很明白。

## public static native Thread currentThread()

返回当前线程的引用。

## public static native void yield()

线程让步，当一个线程使用了这个方法，他就会把自己的cpu时间让掉，让自己或者其他线程运行，不一定会一定让步成功，也不一定按照优先级让步，并且所谓的优先级也是大量执行次数中才能体现的。

## public static native void sleep(long millis) throws InterruptedException

当前线程进行超时等待，时间结束自动开始运行。

## public void interrupt()

给此线程一个中断标志，但是线程还是会继续运行。

## public static boolean interrupted()

返回当前线程是否被中断，然后清除中断状态。

## public boolean isInterrupted()

查看此线程是否是中断状态。

## public final native boolean isAlive()

查看线程是否存活。

## public final void setPriority(int newPriority)

设置优先级。

## public final int getPriority()

得到优先级。

## public final void join() throws InterruptedException

等待当前线程die后再去执行新的任务。还有另外两个方法，一个参数代表等待多少毫秒，两个参数代表等待多少毫秒和多少纳秒。

## public static void dumpStack()

输出线程堆栈，在调试时可以用，可以看到是谁调用这个方法。

## public final void setDaemon(boolean on)

将线程设置为守护线程。什么是守护线程？-----守护线程拥有自动结束自己生命周期的特性，而非守护线程不具备这个特点。

用法：希望在 JVM 退出时，线程能够自动关闭，此时，守护线程是你的首选。

## public final boolean isDaemon()

测试是否为守护线程。

## public static native boolean holdsLock(Object obj)

查看线程是否拥有obj的锁。

## public StackTraceElement[] getStackTrace()

这个和上边的dumpStack()相似，但是这个把信息存在数组返回。比如一个项目所有需要信息传输的全部调用一个方法，但是不用写参数，而是通过这个方法获取谁调用的。

## public State getState()

public enum State {  
 /\*尚未启动的线程\*/  
 *NEW*,  
 /\*正在java虚拟机中执行的\*/  
 *RUNNABLE*,  
 /\*被阻塞，需要被唤醒\*/  
 *BLOCKED*,  
 /\*等待状态，join\*/  
 *WAITING*,  
 /\*时间等待状态，join（1000）\*/  
 *TIMED\_WAITING*,  
 /\*线程执行完毕\*/  
 *TERMINATED*;  
}

得到线程的状态，这个方法只能用来做监视，不用来做同步。

public static void setDefaultUncaughtExceptionHandler

(UncaughtExceptionHandler eh)

设置当线程由于未捕获的异常突然终止而调用的默认处理程序，相当于个全局catch。

## public static UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()

得到设置的默认程序。

## public UncaughtExceptionHandler getUncaughtExceptionHandler()

得到因为没有设置异常处理而突然中止引起的程序，否则为空。（发现和上边的名字一样，只是调用方式不一样，返回参数也就不一样了）

# ThreadGroup

内部组成：

## private final ThreadGroup parent; Thread threads[];

## ThreadGroup groups[];

表明当前组下可以有线程组和作为父子的概念，用于统一管理。

## public final ThreadGroup getParent()

得到父线程。

## public final boolean parentOf(ThreadGroup g)

查看是否为父线程或者祖父线程。

## public int activeCount()

返回当前线程组包括子线程的所有存活线程数量。

# Future

## boolean cancel(boolean mayInterruptIfRunning)

参数代表是否要中断线程，只有线程正在进行才返回true。

## boolean isCancelled()

如果任务在完成之前被取消则返回true。

## boolean isDone()

是否执行结束（结束、取消、异常）。

## V get() throws InterruptedException, ExecutionException

获取结果，若无结果会阻塞至异步计算完成再输出。

## V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException

在给定的时间内会阻塞获取结果，再输出。

# RunnableFuture

## void run();

只有这一个方法，问题是也不用写的呀，因为有继承Runnable，估计即兴创作可能是其中一个原因。

# FutureTask

状态：

private static final int *NEW* = 0;新建和执行中  
private static final int *COMPLETING* = 1;将要执行完毕  
private static final int *NORMAL* = 2;正常执行结束  
private static final int *EXCEPTIONAL* = 3;任务异常  
private static final int *CANCELLED* = 4;任务取消  
private static final int *INTERRUPTING* = 5;任务即将被中断  
private static final int *INTERRUPTED* = 6;任务线程已中断

构造方法：

## public FutureTask(Callable<V> callable)

## public FutureTask(Runnable runnable, V result)