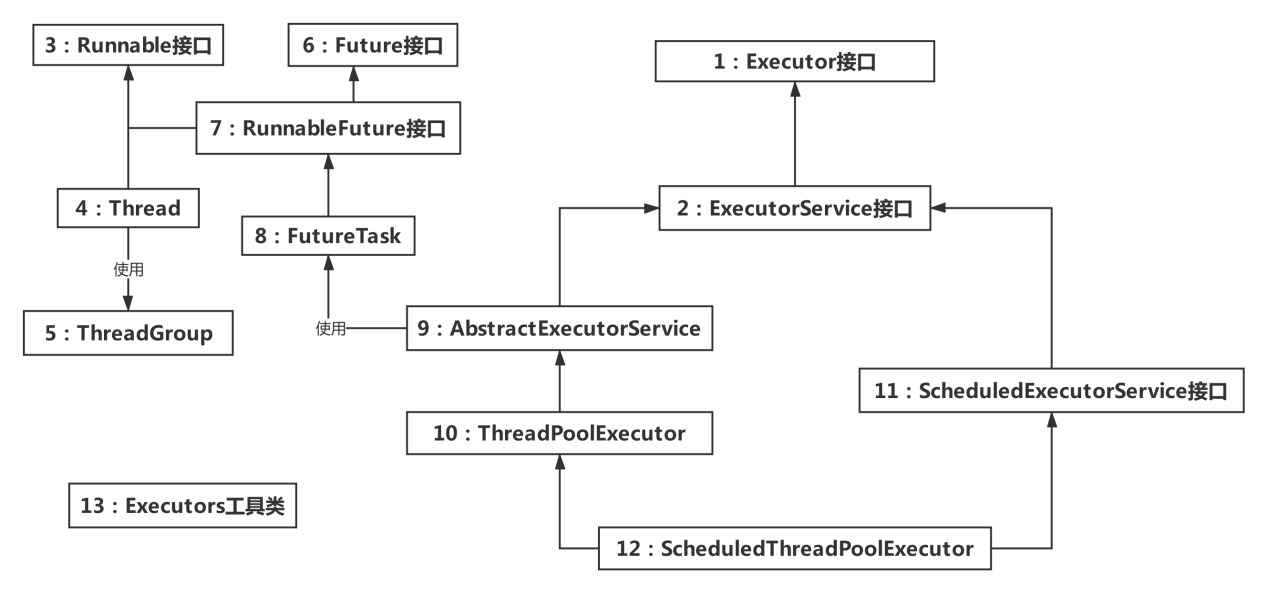
结构图入下，大标题对应图中的数字。

有重点标识，有代码链接和文章解释链接。

文章的红色部分意思是接口的方法，又粘贴了下，标记下。



# 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。（提问：？是啥意思）简单说就是T用于方法，而？用于参数。

## <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，比较高深。

## 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 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）

## public final int getPriority()

得到优先级。

# 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。只有线程正在进行或还未开始则返回true，代表已被取消。

## 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)

## public boolean isCancelled() { return state >= *CANCELLED*;}

判断任务是否已经被取消（中断）。

## public boolean isDone() { return state != *NEW*;}

判断任务是否已完成。

## public boolean cancel(boolean mayInterruptIfRunning)

很熟悉吧，因为是继承的Future。参数代表是否要中断线程，只有线程正在进行或者还未开始返回true，代表已经被取消。（重点）

## public 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。

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

在给定时间内获取结果，需要如果时间过期还没结果，会抛TimeoutException。

# AbstractExectorService

因为继承的ExecutorService，实现了三个提交任务的方法和四个异步查找的方法。如下，回顾下，想想用法和作用。

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

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

## Future<?> submit(Runnable task)

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

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

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

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

# ThreadPoolExecutor

（int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue）

（int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,

RejectedExecutionHandler handler）

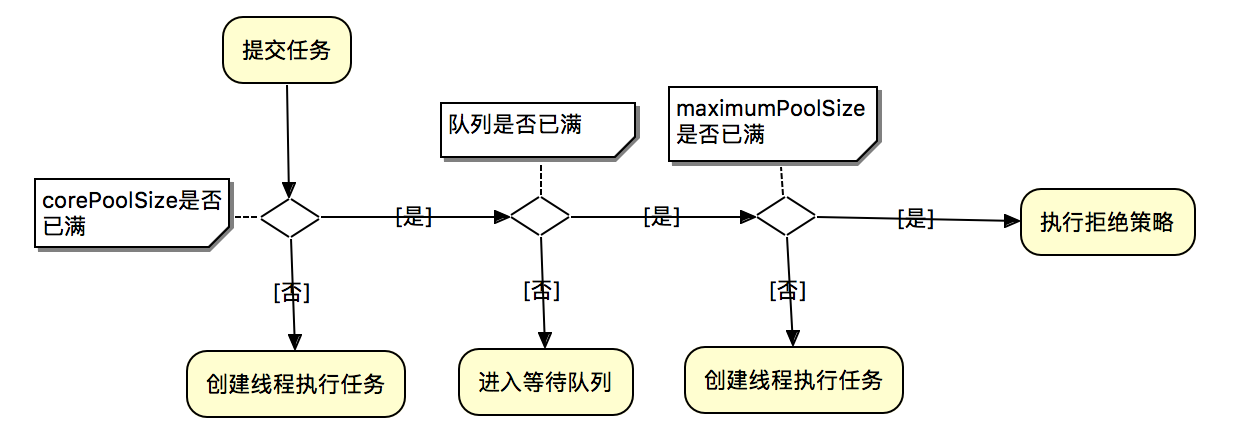
（int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory）

所有构造方法指向这个，总共七个参数。重点。其他方法有调用。

public ThreadPoolExecutor(int corePoolSize,核心线程池大小  
 int maximumPoolSize,线程池最大数量  
 long keepAliveTime,线程最大空闲时间  
 TimeUnit unit,空闲时间单位  
 BlockingQueue<Runnable> workQueue,等待队列  
 ThreadFactory threadFactory,创建的工厂  
 RejectedExecutionHandler handler)拒绝策略

## 关于核心线程数量和线程池最大数量：



注意顺序，先看核心线程数量，再看队列大小，最后看总线程数量。所以只有队列是有界的时候才会往下走。

## 关于线程最大空闲时间和空闲单位：

当线程池数量超过核心线程池数量后，那么多余的线程在特定的时间和单位下被终止。

## 关于创建的工厂：

未指定，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，并且可以统一设置名字的。

## 关于拒绝策略：

分为线程池关闭和线程池数量达到上限两种情况会触发拒绝策略。在ThreadPoolExecutor里实现了四个类继承于RejectedExecutionHandler。

AbortPolicy：直接抛错！！！

CallerRunsPolicy：在调用者的线程里执行这个任务，为了减缓提交任务量。

DiscardOldestPolicy：先执行poll，把头元素移出，再执行这个线程。

DiscardPolicy：这个方法是空的，啥都不干，相当于直接把多的线程给丢掉了。

## retry:

就是个标记，功能类似goto，命名不是固定的，只要合法就行。

## 关于等待队列：

我看继承BlockingQueue接口的有八个类，这边牵扯到队列上了，就简单说几个。

SynchronousQueue：直接握手队列，生产者put数据后（如果已经有数据也会阻塞直到数据被消费），消费者take会进行等待只要有数据可以消费。（是个重点，设计的很有意思，分为公平和不公平，公平队列的FIFO，不公平栈的LIFO），缺点：生产者速度过快会导致线程无限增长。[看代码](https://github.com/xiaobaobao007/PracticeCode/blob/master/src/Collection/Queue/SynchronousQueueTest.java)。

LinkedBlockingQueue：无界的FIFO队列。同样导致无限增长。

ArrayBlockingQueue：有界的FIFO队列。有助于资源耗尽，但是难以控制，队列大小和线程池大小需要权衡设计。

DelayedWorkQueue：无界的堆。数据结构堆需要了解（其实就是个数组，按照树的结构思维进行排序）[讲解](https://www.cnblogs.com/java-chen-hao/archive/2019/01/16/10275910.html" \o "讲解)。

## break retry;

直接跳出多层循环，而不只是跳出一次循环。

## continue retry;

回到标记处。

## addWorker(Runnable firstTask, boolean core)

添加任务。Core（if (wc >= *CAPACITY* ||wc >= (core ? corePoolSize : maximumPoolSize))  
return false;）意思是，true往核心线程添加，false往线程池添加，如果添加不了则false。

## public boolean prestartCoreThread()

启动核心线程，进行一个预启动操作。空闲等待任务的到来。如果方法前所有核心线程已经启动则返回false，如果成功启动所有核心线程则返回true。

# ScheduledExecutorService接口

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

多少秒延迟后执行线程任务。（只执行一次）

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

多少秒延迟后执行线程任务。（只执行一次）

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

在initialDelay延迟后开始第一次执行，随后每次间隔时间=initialDelay+n\*period。

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

在initialDelay延迟后开始第一次执行，随后每次间隔时间=delay。

# ScheduledThreadPoolExecutor

有四个构造方法，都是调用的ThreadPoolExecutor的构造方法。一定要把他的七个参数搞明白。

## public ScheduledThreadPoolExecutor(int corePoolSize)

super(corePoolSize, Integer.*MAX\_VALUE*, 0, *NANOSECONDS*,new DelayedWorkQueue())

指定核心线程的数量，默认线程池数量，0等待，默认DelayedWorkQueue队列。

## public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory)

super(corePoolSize, Integer.*MAX\_VALUE*, 0,*NANOSECONDS*,new DelayedWorkQueue(), threadFactory)指定核心线程的数量，默认线程池数量，0等待，默认DelayedWorkQueue队列，指定的线程池工厂。

## public ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHandler handler)

super(corePoolSize,Integer.*MAX\_VALUE*,0,*NANOSECONDS*,new DelayedWorkQueue(),

handler)指定核心线程的数量，默认线程池数量，0等待，默认DelayedWorkQueue队列，指定的拒绝策略。

## public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory, RejectedExecutionHandler handler)

super(corePoolSize, Integer.*MAX\_VALUE*, 0, *NANOSECONDS*,new DelayedWorkQueue(), threadFactory, handler)指定核心线程的数量，默认线程池数量，0等待，默认DelayedWorkQueue队列，指定的线程工厂，指定的拒绝策略。

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

多少秒延迟后执行线程任务。（只执行一次）

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

多少秒延迟后执行线程任务。（只执行一次）

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

在initialDelay延迟后开始第一次执行，随后每次间隔时间=initialDelay+n\*period。

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

在initialDelay延迟后开始第一次执行，随后每次间隔时间=delay。

# Executors工具类

就像是Collection和Collections的关系，这个类提供很多满足大部分需求的线程池和一些线程池的操作。不再分析源码直接说用法（ThreadPoolExecutor的构造方法。一定要把他的七个参数）。

核心线程数量=最大线程池数量=nThreads，无界的队列方式。特点：运行数量的限制和等待队列无限制并且不会被清除。

newFixedThreadPool(int nThreads)

newFixedThreadPool(int nThreads, ThreadFactory threadFactory)

核心线程数量=0，最大线程池数量=Integer.MAX，60S计时，SynchronousQueue直接握手队列。特点：时间超时回收。

newCachedThreadPool()

newCachedThreadPool(ThreadFactory threadFactory)

核心线程数量=corePoolSize，最大线程池数量=Integer.MAX，DelayedWorkQueue延时队列。特点：运行数量的限制、延时的特点和无限制不清除。

newScheduledThreadPool(int corePoolSize)

newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)

核心线程数量=最大线程池数量=1，LinkedBlockingQueueFIFO队列。特点：一次只有一个线程运行，并且所有任务按照FIFO原则排序。

newSingleThreadExecutor()

newSingleThreadExecutor(ThreadFactory threadFactory)

提供默认的线程工厂，统一设置非守护线程和统一的默认优先级。

defaultThreadFactory()

# 注意事项

## execute和submit的区别与联系

execute和submit都属于线程池的方法，execute只能提交Runnable类型的任务，而submit既能提交Runnable类型任务也能提交Callable类型任务。

execute会直接抛出任务执行时的异常，submit会吃掉异常，可通过Future的get方法将任务执行时的异常重新抛出。

execute所属顶层接口是Executor,submit所属顶层接口是ExecutorService，实现类ThreadPoolExecutor重写了execute方法,抽象类AbstractExecutorService重写了submit方法。