

1. 关键API

1.1 生成固定大小的线程池

```
// 生成固定大小的线程池
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS, new
    LinkedBlockingQueue<Runnable>());
}

// corePoolSize — 最小线程数；如果allowCoreThreadTimeOut设置为true，线程池最后会减少到0；
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit,
    BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        Executors.defaultThreadFactory(), defaultHandler);
}
```

对于 `allowCoreThreadTimeOut`

If false (default), `core threads` stay alive even when idle.
If true, `core threads` use keepAliveTime to time out waiting for work.

- **corePoolSize:**

好好理解下面一段话：

线程池的基本大小，即在没有任务需要执行的时候线程池的大小，并且只有在 **工作队列满了** 的情况下才会创建超出这个数量的线程。

而**工作队列**指的是上面代码中的 `new LinkedBlockingQueue<Runnable>()`，它的默认容量是 `Integer.MAX_VALUE`，所以不可能（或很难）放满。你可以手动设置队列（`BlockingQueue`）的容量（比如：8），但是，仅仅这样就可以了吗？不！这个`BlockingQueue`和`maximumPoolSize`大有关系：

- **BlockingQueue + maximumPoolSize >= 最高突发任务数**

当所需线程数大于`BlockingQueue`容量的时候，多出的无法进入队列的任务就会创建新线程来执行，如果`maximumPoolSize`设置为有限值（如：`maximumPoolSize=8`），那么，当无法进入队列的任务数超过8（`maximumPoolSize`）时，就会抛出异常，所以，始终要保证：**`BlockingQueue + maximumPoolSize >= 最高突发任务数`**

- 特殊情况（`corePoolSize=0`）时，需要满足**`BlockingQueue + maximumPoolSize - 1 >= 最高突发任务数`**，否则抛出异常；但是，这种特殊情况对**`CachedThreadPool`**又没有影响：（不知道为啥）

示例代码：

```
@Test
public void test() throws InterruptedException {
    ExecutorService es = new ThreadPoolExecutor(1, 2, 60L,
        TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
    es.submit(runnable);
    es.submit(runnable);
    Thread.sleep(4000);
}
```

```

@Test
public void testLinked() throws InterruptedException {
    ExecutorService es = new ThreadPoolExecutor(1, 2, 60L,
        TimeUnit.SECONDS, new LinkedBlockingQueue<>(1));
    es.submit(runnable);
    es.submit(runnable);
    Thread.sleep(4000);
}

private Runnable runnable = () -> {
    System.out.println("开始线程" + Thread.currentThread().getId());
    try {
        Thread.sleep(3000);
        System.out.println("执行线程" +
            Thread.currentThread().getId());
    } catch (InterruptedException e) {
    }
    System.out.println("完成线程" + Thread.currentThread().getId());
};

```

1.2 CachedThreadPool

`newCachedThreadPool()` 生成一个初始大小为0，空闲时间为60s，没有上限的连接池：

```

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

```

`newCachedThreadPool(ThreadFactory threadFactory)` 同上：

- 关于 `SynchronousQueue` 可以参考[博客](#)

1.3 ScheduledThreadPool

`newScheduledThreadPool(int corePoolSize)` 和

`newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)` 生成延迟或定期执行任务的线程池：

- `ScheduledExecutorService#scheduleWithFixedDelay`：是 前一次执行结束 到 后一次执行开始 的间隔为 `delay`；
- `ScheduledExecutorService#scheduleAtFixedRate`：执行时刻是 `initialDelay`、`initialDelay + period`、`initialDelay + period * 2`、`initialDelay + period * 3`。。。等，如果任务执行时间大于 `period`，那么前一次执行完后立马执行下一个；

1.4 单线程

- `newSingleThreadExecutor()` 生成一个单线程并执行任务队列；
- 如果单线程因为执行任务而意外中断或关闭，那么会生成一个新的线程代替旧的线程继续执行接下来的任务；
- `newSingleThreadExecutor()` 和 `newFixedThreadPool(1)` 区别：

```

// final ExecutorService single = Executors.newSingleThreadExecutor();
final ExecutorService fixed = Executors.newFixedThreadPool(1);
ThreadPoolExecutor executor = (ThreadPoolExecutor) fixed;
executor.setCorePoolSize(4); // newFixedThreadPool(1)可以再设置大小

```

- 单线程 + 定时/延迟
`newSingleThreadScheduledExecutor`

1.5 newWorkStealingPool

作用：貌似是尽可能地利用所有处理器，生成一个线程池；

2. 原理与概念

- 在刚刚创建ThreadPoolExecutor的时候，线程并不会立即启动，而是要等到有任务提交时才会启动，除非调用了prestartCoreThread/prestartAllCoreThreads事先启动核心线程
- `largestPoolSize` - 该变量记录了线程池在整个生命周期中 曾经 出现的最大线程个数