

1. 四种线程池

使用Executors类中的这四个方法可以创建四个线程池，这四个线程池都是ExecutorService类型的，关闭这四种线程池都是调用ExecutorService类中的shutdown方法

- newCachedThreadPool 具有缓存性质的线程池,该线程池中的线程最大空闲时间60s，在某个线程空闲的60s内，该线程可重复利用(缓存特性)，没有最大线程数限制。一个没有容量的等待队列（因为不需要，根据创建ThreadPoolExecutor时指定的参数可以知道这个线程池没有核心线程，应急线程可以有Integer.MAX个，如果一个任务P被提交到这个线程池中，但是这个线程池中沒有空闲的线程，那么会直接创建一个新的线程存放在线程池中，用这个新创建的线程运行任务P）。

该线程池适用场景：高并发情况下 && 任务执行时间短的场景

- newFixedThreadPool 具有固定数量的线程池，核心线程数等于最大线程数，线程池中空闲线程的最大空闲时间为0，超出最大线程数的任务进入等待队列。（优先使用这个线程池）

该线程池适用场景：高并发下控制性能

- newScheduledThreadPool 具有时间调度特性的线程池，必须初始化核心线程数，最大线程数为Integer.MAX，线程池中线程的最大空闲时间为0，底层使用DelayedWorkQueue实现延迟特性。

该线程池适用场景：

- newSingleThreadExecutor 核心线程数与最大线程数均为1；这个方法在创建类的时候和使用newFixedThreadPool方法创建的类的时候底层都是使用ThreadPoolExecutor类，并且参数除了core和max之外全都相同，区别只是core和max都是1；

该线程池适用场景：不需要并发，但需要一定执行顺序的场景

- 这四个线程池的底层参数都有四个，只是这四个参数的值不同：这四个线程底层都是创建ThreadPoolExecutor类的实例对象（区别只是这些参数不同），该类的构造器中有六个参数，下面列举出的四个是这四个线程池用到的，还有参数：timeout的单位、拒绝策略。
 - core 核心线程数，线程池初始化的时候指定的线程数量，如果核心线程数量为0，那么每一个任务都会创建一个新的线程；核心线程不会自动结束。
 - max 最大线程数，线程池在初始化之后可以最多可以运行的线程数量是：应急线程数量+核心线程数【max-核心线程数=应急线程数】，最大线程数不能小于核心线程数，最少应该是相等的
 - timeout 超时时间，线程池中应急线程的的最大空闲时间（如果在创建了线程池之后调用allowsCoreThreadTimeOut方法并且传递参数为true，那么

这个时间也是核心线程的超时时间)：线程池中的线程（应急+核心线程）如果空闲时间超过 `keepAliveTime`，将被终止。这提供了一种在池没有被积极使用时减少资源消耗的方法。如果池稍后变得更加活跃（有新的任务提交），则将构造新线程。

- `queue` 等待队列（存放任务），当正在运行的线程数等于最大线程数之后，再次向线程池提交任务，这个时候这个任务就会进入等待队列，当线程池中又线程空闲的时候，等待队列中的任务才会被执行

- 示例：

`CachedThreadPoolTest`（在代码中两处`sleep`的地方用于对比，当注释掉外部的`sleep`，打开`run`方法中的`sleep`方法之后由于每个线程运行的时间不超过1S，导致在1S后下一个线程开始的时候会使用上线程池中空闲的线程而不是新创建一个线程）

`FixedThreadPoolTest`

```
//得到运行时环境下可用的线程数量，这个数量可以作为使用
newFixedThreadPool创建线程时的核心线程数量
System.out.println(Runtime.getRuntime().availableProcessors());
ExecutorService fixedThreadPool =
Executors.newFixedThreadPool(4*20);
for (int i = 0; i < 10; i++) {
    final int index = i;
    fixedThreadPool.execute(new Runnable() {
        public void run() {
            try {

                System.out.println(Thread.currentThread().getName()+">>"+index);

                /*
                    * run方法中的sleep的作用是为了模拟线程运行所需要的时间，当当前正在运行
                    * 的线程的数量超过核心线程数之后就会等待，直到正在运行的核心线程运行完毕才
                    * 能有新的线程创建并运行
                    */
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
}
```

`ScheduledThreadPoolTest`（有三种方式）

```
public static void test1() {
    ScheduledExecutorService scheduledThreadPool
= Executors.newScheduledThreadPool(1);

    for (int i = 0; i < 10; i++) {
```

```

        final int index = i;
        Runnable task = new Runnable() {
            public void run() {

                System.out.println(Thread.currentThread().getNam
e() + ">> delay " + index + " seconds run....");
            }
        };
        //此处的意思是：task指定的任务要延迟i秒后执行
        ScheduledFuture<?> schedule =
scheduledThreadPool.schedule(task, i,
TimeUnit.SECONDS);
    }
    scheduledThreadPool.shutdown();
}

public static void test2() {
    ScheduledExecutorService scheduledThreadPool
= Executors.newScheduledThreadPool(12);
    Runnable task = new Runnable() {
        public void run() {
            try {

                System.out.println(Thread.currentThread().getNam
e() + ">> sleep..." +
System.currentTimeMillis());
                //将这里的参数由3秒换成1秒，可以看出来
                //当任务的运行时间大于执行scheduleAtFixedRate方法时指定的
                //周期

                //时，一个任务两次运行的状况
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(Thread.currentThread().getNam
e() + ">> run....." +
System.currentTimeMillis());
        }
    };

    //第三个参数是第二个参数的单位，第二个参数是一个任务周
    //期性运行的固定频率
    // 此处的意思是：一个任务在0毫秒之后执行（立即执行），
    // 之后的该任务每隔2秒执行1次（如果任务执行的时间小于2秒），如
    // 果任务本身运行的时间比2秒长，那么这个任
    // 务就是在第一次运行完之才会再次运行，而不是在【从第一
    // 个任务开始运行2秒后，第一个任务运行中】的情况下执行第二个任
    // 务。
    ScheduledFuture<?> scheduleAtFixedRate =
scheduledThreadPool.scheduleAtFixedRate(task, 0,
2, TimeUnit.SECONDS);

```

```

    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    //对定时任务进行停止操作，参数为true意味着当第一次任务开始执行10秒之后，无论该任务是否还在运行都会停止
    //传递参数为false的时候，意味着10秒之后，如果还有线程在运行，那么会等待该线程运行完毕
    scheduleAtFixedRate.cancel(true);
    //关闭线程池
    scheduledThreadPool.shutdown();
}

public static void test3() {
    ScheduledExecutorService scheduledThreadPool
= Executors.newScheduledThreadPool(10);
    Runnable task = new Runnable() {
        public void run() {
            try {

                System.out.println(Thread.currentThread().getName() + ">> sleep..." +
                System.currentTimeMillis());
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(Thread.currentThread().getName() + ">> run....." +
            System.currentTimeMillis());
        }
    };

    //第三个参数是第二个参数的单位，第二个参数是一个任务周期性运行的固定延迟时间
    //此处的意思是：一个任务在0毫秒之后执行（立即执行），之后该任务执行完毕之后等待2秒再次执行
    ScheduledFuture<?> scheduleWithFixedDelay =
scheduledThreadPool.scheduleWithFixedDelay(task,
0, 2, TimeUnit.SECONDS);

    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    //对定时任务进行停止操作，参数为true意味着当第一次任务开始执行10秒之后，无论该任务是否还在运行都会停止

```

```

//传递参数为false的时候，意味着10秒之后，如果还有现成
在运行，那么会等待该线程运行完毕
scheduleWithFixedDelay.cancel(true);
//关闭线程池
scheduledThreadPool.shutdown();
}

public static void main(String[] args) {
    //test1();
    test2();
    //test3();
}

```

、
SingleThreadExecutorTest

1.1 向线程池中提交任务的方式

- 向ThreadPoolExecutor类构造的线程池中提交任务的两种方式：
execute()和submit()

<T> Future<T>	submit(Callable<T> task)	提交一个返回值的任务用于执行，返回一个表示任务的未决结果的 Future。
Future<?>	submit(Runnable task)	提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。
<T> Future<T>	submit(Runnable task, T result)	提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。 Future 博客
void	execute(Runnable command)	在未来某个时间执行给定的命令。 @51CTO博客

两种方式提交的区别：

- execute只能提交Runnable类型的任务，无返回值。
- submit既可以提交Runnable类型的任务，也可以提交Callable类型的任务，会有一个类型为Future的返回值，但当任务类型为Runnable时，返回值为null。
- execute在执行任务时，如果遇到异常会直接抛出
- submit不会直接抛出，只有在使用Future的get方法获取返回值时，才会抛出异常。
- 向ScheduledThreadPoolExecutor类（ThreadPoolExecutor类的子类）构造的线程池中提交任务的三种方式：（有关这三种方式可以去看上面的示例中的注释，示例中的三个方法分别使用了下表的方法）

方法摘要

<code><V> ScheduledFuture<V></code>	<code>schedule(Callable<V> callable, long delay, TimeUnit unit)</code> 创建并执行在给定延迟后启用的 <code>ScheduledFuture</code> 。
<code>ScheduledFuture<?></code>	<code>schedule(Runnable command, long delay, TimeUnit unit)</code> 创建并执行在给定延迟后启用的 <code>Runnable</code> 的一次性操作。注意和下面两个方法的区别
<code>ScheduledFuture<?></code> 这两个方法是循环执行同一个任务，区别只是在循环的机制不同，前者是按照周期循环执行，后者是按照间隔循环执行	<code>scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)</code> 创建并执行一个在给定初始延迟后首次启用的定期操作，后续操作具有给定的周期；也就是将在 <code>initialDelay</code> 后开始执行，然后在 <code>initialDelay+period</code> 后执行，接着在 <code>initialDelay + 2 * period</code> 后执行，依此类推。
<code>ScheduledFuture<?></code>	<code>scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)</code> 创建并执行一个在给定初始延迟后首次启用的定期操作，随后，在每一次执行终止和下一次执行开始之间都存在给定的延迟。

对scheduleAtFixedRate方法的补充：

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

创建并执行在给定的初始延迟之后，随后以给定的时间段首先启用的周期性动作；那就是执行将在 `initialDelay` 之后开始，然后 `initialDelay+period`，然后是 `initialDelay + 2 * period`，等等。如果任务的执行遇到异常，则后续的执行被抑制。否则，任务将仅通过取消或终止执行人终止。如果任务执行时间比其周期长，则后续执行可能会迟到，但不会同时执行。

参数：

command - 要执行的任务
initialDelay - 首次执行的延迟时间
period - 连续执行之间的周期
unit - `initialDelay` 和 `period` 参数的时间单位

返回：

表示挂起任务完成的 `ScheduledFuture`，并且其 `get()` 方法在取消后将抛出异常

抛出：

[RejectedExecutionException](#) - 如果无法安排执行该任务
[NullPointerException](#) - 如果 `command` 为 `null`
[IllegalArgumentException](#) - 如果 `period` 小于等于 0

- 关于这四个方法的返回值的类型：ScheduledFuture

```
public interface
ScheduledFuture<V>
extends Delayed, Future<V>
```

一个延迟的、结果可接受的操作，可以将其取消。通常已安排的 `future` 是用 `ScheduledExecutorService` 安排任务的结果（官网描述，但我没看懂）。

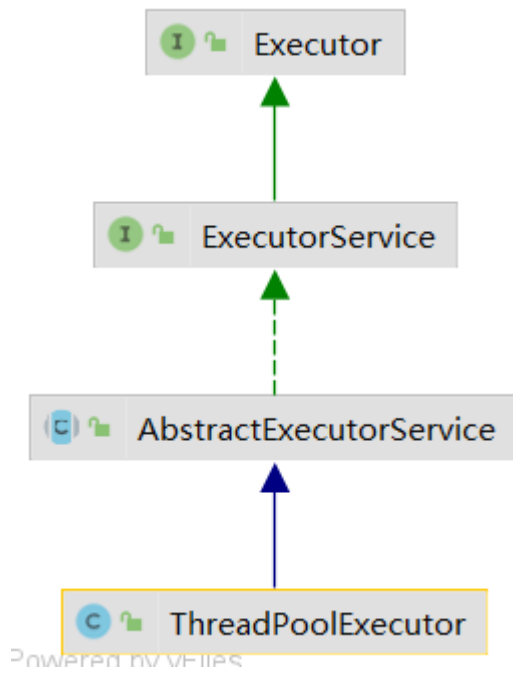
- `cancel(boolean mayInterruptIfRunning)`：继承自 `Future`；试图取消对此任务的执行。如果任务已完成、或已取消，或者由于某些其他原因而无法取消，则此尝试将失败。当调用 `cancel` 时，如果调用成功，而此任务尚未启动，则此任务将永不运行。如果任务已经启动，则 `mayInterruptIfRunning` 参数确定是否应该以试图停止任务的方式来中断执行此任务的线程。

2. ThreadPoolExecutor

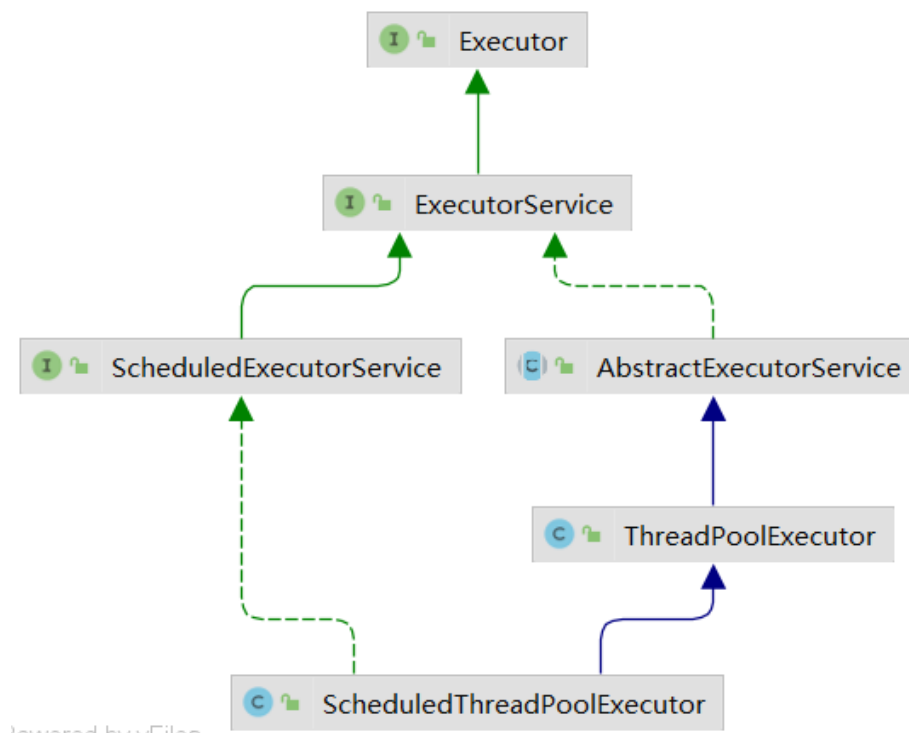
- 四种线程池都是通过 `Executors` 类（工厂类）创建的，底层创建的都是 `ThreadPoolExecutor` 类，可以构建自己需要的线程类。

- 类图：

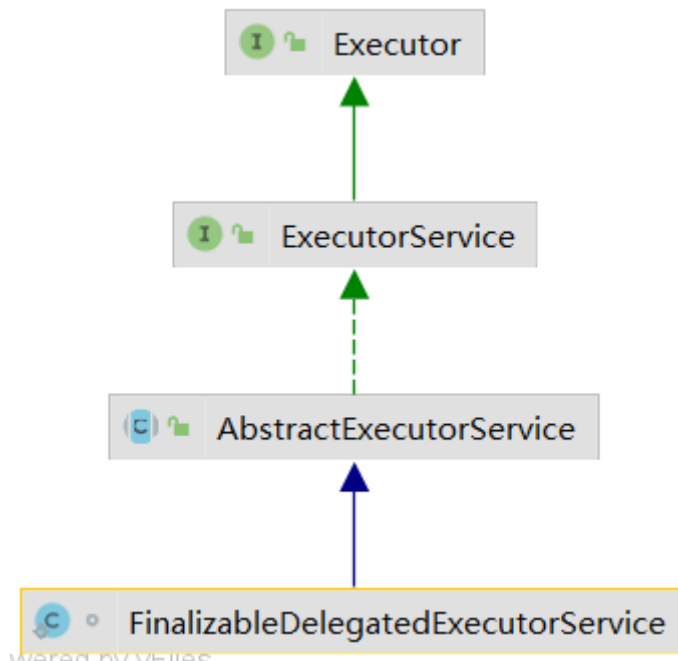
创建第一二种线程池的时候使用的ThreadPoolExecutor类的类图



创建第三种线程池的时候使用的ScheduledThreadPoolExecutor类的类图



创建第四种线程池使用的类图



3. 线程池使用拒绝策略

- 拒绝策略是和等待队列相关的；如果等待队列是有限的队列，如果等待的线程的数量超过了等待队列的容量，这个时候就需要使用拒绝策略对这些超出【最大线程数量 + 等待队列长度】个数的线程进行处理，JDK为我么提供了四种默认的拒绝策略：
 - AbortPolicy 抛出异常,不影响其他线程运行
 - CallerRunsPolicy 使用主调用当前任务，会优先运行该任务
 - DiscardOldestPolicy 丢弃最老的任务（将线程池中第一个线程开始运行的时候执行的任务丢弃，之后如果还需要执行拒绝策略，则按照线程池中线程开始运行的顺序逐个丢弃这些线程对应的任务）
 - DiscardPolicy 直接丢弃,什么也不做

这些决绝策略都是实现了RejectedExecutionHandler接口的类，通过实现该接口中的rejectedExecution方法来实现拒绝策略，我们也可以定义自己的拒绝策略。

- 示例：ThreadPoolExecutorTest
示例代码中构建自己需要的简单的线程类的方式（没有使用拒绝策略）：

```
class MyTask implements Runnable{

    int index = 0;

    public MyTask(int index) {
        this.index = index;
    }
}
```

```

@Override
public void run() {
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println(Thread.currentThread().getName()
        + ">>run " + index);
}
}

public class test{
    public static void main(String[] args) {
        ThreadPoolExecutor executor = new
        ThreadPoolExecutor(7,7,60L,TimeUnit.SECONDS,new
        LinkedBlockingQueue<Runnable>());
        for(int i=0;i<8;i++){
            MyTask task = new MyTask(i);
            executor.submit(task);
        }
        executor.shutdown();
    }
}

```

示例代码中使用拒绝策略部分的代码：

```

ThreadPoolExecutor executor = new
ThreadPoolExecutor(3,3,0,TimeUnit.SECONDS,new
LinkedBlockingQueue<Runnable>(1),new
AbortPolicy());

ThreadPoolExecutor executor = new
ThreadPoolExecutor(3,3,0,TimeUnit.SECONDS,new
LinkedBlockingQueue<Runnable>(1),new
CallerRunsPolicy());

ThreadPoolExecutor executor = new
ThreadPoolExecutor(3,3,0,TimeUnit.SECONDS,new
LinkedBlockingQueue<Runnable>(1),new
DiscardOldestPolicy());

ThreadPoolExecutor executor = new
ThreadPoolExecutor(3,3,0,TimeUnit.SECONDS,new
LinkedBlockingQueue<Runnable>(1),new
DiscardPolicy());

```

示例代码中自定义拒绝策略的方式以及使用的代码：

```

class MyPolicy implements
RejectedExecutionHandler{
    //参数中的r指的是被拒绝策略处理的任务
    @Override
    public void rejectedExecution(Runnable r,
ThreadPoolExecutor executor) {
        System.out.println(r+"被拒绝执
行"+System.currentTimeMillis());
    }
}
public class test{
    //使用自定义的拒绝策略创建自定义的线程池
    ThreadPoolExecutor executor = new
ThreadPoolExecutor(3,3,0,TimeUnit.SECONDS,new
LinkedBlockingQueue<Runnable>(1),new MyPolicy());
    //使用这个线程池对任务（任何实现了Runnable接口、
Callable接口的实现类的实例对象）进行处理

}

```

4. ThreadFactory

改变线程池中，线程创建的行为

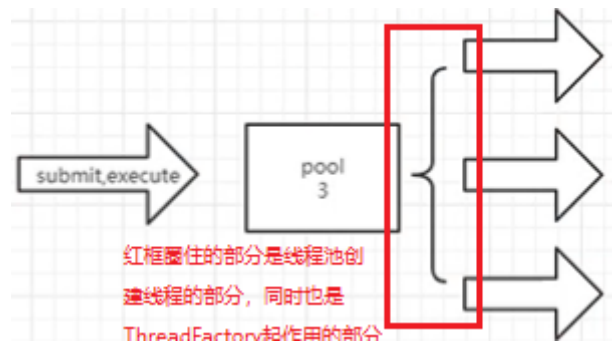
知识点

我们通常使用线程池的submit方法将任务提交到线程池内执行。

如果此时线程池内有空闲的线程，则会立即执行该任务，如果没有则需要根据线程池的类型选择等待，或者新建线程。

所以线程池内的线程并不是线程池对象初始化（**new**）的时候就创建好的，而是当有任务被提交进来之后才创建的，而创建线程的过程在线程池创建的时候不指定**ThreadFactory**参数时是无法干预的。

如果我们想在每个线程创建时记录一些日志，或者推送一些消息那怎么做？使用ThreadFactory接口。



使用ThreadFactory接口的步骤：

1. 编写ThreadFactory接口的实现类
 - a. 该接口中只有一个newThread方法，参数是任务（Runnable接口的实现类）的实例对象，该方法的作用是将传入进来的任务转成线程对象
2. 创建线程池时传入ThreadFactory对象

Demo:

com.mkevin.demo13.DemoThreadFactory

```
class DemoThreadFactory implements ThreadFactory {
    // 控制线程创建时是否记录日志
    private boolean saveLog;
    // 工场名称
    private String factoryName;

    public DemoThreadFactory(String factoryName, boolean
saveLog) {
        this.factoryName = factoryName;
        this.saveLog = saveLog;
    }

    public Thread newThread(Runnable r) {
        if (saveLog) {
            //动态输出日志
            P.l(System.currentTimeMillis() +
this.factoryName + " create start");
        }
        //创建执行指定任务r的线程
        Thread thread = new Thread(r);
        //自定义线程名字
        thread.setName("Kevin-Thread-" + thread.getName()
+ ":" + thread.getId());
        try {
            //模拟线程初始化时间

            Thread.sleep(ThreadLocalRandom.current().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if (saveLog) {
```

```
        P.l(System.currentTimeMillis() +
this.factoryName + " " + thread.getName() + " create
end");
    }
    return thread;
}
}
```

com.mkevin.demo13.ThreadFactoryDemo0

5. 线程池内异常的优雅处理

优雅的处理线程池内未捕获异常

线程池状态

- 线程池内运行的线程如果发生异常，一定要捕获，要养成习惯。
- 可以采用更优雅的方式处理所有线程的异常，例如记录日志，发送预警消息等。
 - 结合ThreadFactory以及线程的setUncaughtExceptionHandler方法来处理最为优雅（在newThread方法中调用Thread类中的这个方法设置未捕获异常处理器）
 - 这种方式对execute提交的任务有效，对submit提交的任务无效，巨坑！

Demo:

com.mkevin.demo15.ThreadExceptionDemo1

6. 关闭线程池

shutdown和shutdownNow的作用和区别

shutdown 与 shutdownNow

知识点

- shutdown让线程池内的任务继续执行完毕，但是不允许新的任务提交
- shutdown方法不阻塞, 等所有线程执行完毕后，销毁线程
- shutdown之后提交的任务会抛出RejectedExecutionException异常，代表拒绝接收

- `shutdownNow`之后会引发`sleep`、`join`、`wait`方法的`InterruptedException`异常，并且该线程执行的任务不会执行完毕
- 如果正在运行的任务中没有捕获`InterruptedException`异常的条件，则任务会继续运行直到结束，此时`shutdownNow`不起任何作用；如果运行中的任务中有处理`InterruptedException`异常的条件，那么会直接销毁线程（不管线程是否执行完毕）
- `shutdownNow`之后提交的任务会抛出`RejectedExecutionException`异常，代表拒绝接收
- 我大概看了一下对应的源码，按照源码中的使用方式我感觉这个方法的作用是：给当前正在运行的线程设置中断标志为`true`（中断状态），如果正在执行的任务中有处理`InterruptedException`异常的操作，并且该操作会结束任务的执行，那么才会真正结束任务的执行；但是如果正在执行的任务中没有处理`InterruptedException`的条件（或者仅仅是`try`了该异常`catch`语句中是空的）也不会中断该任务。

Demo:

`com.mkevin.demo14.ThreadPoolDemo1`

```
public static void main(String[] args) throws
InterruptedException {

    // 创建线程池，而不提交任何任务，则无任何线程被创建，程序直接运行结束
    /*ExecutorService executorService =
Executors.newCachedThreadPool();
    P.l("main is over");*/

    // 创建线程池，并向线程池提交任务，则创建线程，任务执行完毕，而线程池中刚刚创建的线程不销毁
    // 毁，JVM继续运行等待60秒后，自动销毁空闲线程，JVM退出
    //Runner runner = new Runner();
    //ExecutorService executorService =
Executors.newCachedThreadPool();
    //executorService.submit(runner);
    //P.l("main is over");

    // 创建线程池，并向线程池提交任务，则创建线程，任务执行完毕而线程不销毁，JVM继续运行
    // 始终保持有1个线程存活，因为这个线程是核心线程，核心线程不会随着超时时间的到达而关闭
    // ，所以JVM不会退出，应该使用shutdown方法结束这个线程
    /*Runner runner = new Runner();
    ExecutorService executorService =
Executors.newFixedThreadPool(1);
    executorService.submit(runner);
    P.l("main is over");*/

    //shutdown让线程池内的任务继续执行完毕，但是不允许新的任务提交，shutdown方法不阻塞
```

```

//等所有线程执行完毕后，销毁线程，JVM退出
/*Runner runner = new Runner();
    ExecutorService executorService =
Executors.newFixedThreadPool(1);
    executorService.submit(runner);
    executorService.shutdown();
    P.l("main is over");*/

//shutdown之后提交的任务会抛出RejectedExecutionException
异常，代表拒绝接收
/*Runner runner = new Runner();
    ExecutorService executorService =
Executors.newFixedThreadPool(3);
    executorService.submit(runner);
    executorService.shutdown();
    P.l("main is over");
    executorService.submit(runner);*/

//shutdownNow之后提交的任务会抛出
RejectedExecutionException异常，代表拒绝接收
//shutdownNow之后会引发sleep、join、wait方法的
InterruptedException异常，并且
//线程池中的线程执行的任务不会执行完毕
/*ExecutorService executorService =
Executors.newFixedThreadPool(5);
    executorService.submit(new Runner());
    executorService.shutdownNow();
    P.l("main is over");
    executorService.submit(new Runner());*/

//如果线程池中正在执行的任务中没有触发InterruptedException的
条件，则任务会继续运行直
//到结束，这种情况下shutdownNow方法不会对已经提交并且正在运行
的任务进行处理
/*ExecutorService executorService =
Executors.newFixedThreadPool(5);
    executorService.submit(new Runner1());
    executorService.shutdownNow();
    P.l("main is over");*/

//可以在任务中判断Thread.currentThread().isInterrupted()
来规避shutdownNow的问题
    ExecutorService executorService =
Executors.newFixedThreadPool(5);
    executorService.submit(new Runner2());
    executorService.shutdownNow();
    P.l("main is over");
}

static class Runner implements Runnable {
    public void run() {
        try {

```

```

        P.l(Thread.currentThread().getName()+"
begin");
        Thread.sleep(2000);
        P.l(Thread.currentThread().getName()+" end");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

//不做特殊处理的任务，无法规避shutdownNow的问题
static class Runner1 implements Runnable {
    public void run() {
        while(true){
            P.l(Thread.currentThread().getName()+ " " +
System.currentTimeMillis());
        }
    }
}

//判断线程是否Interrupted的程序，规避shutdownNow的问题
static class Runner2 implements Runnable {
    public void run() {
        while(true){
            P.l(Thread.currentThread().getName()+ " " +
System.currentTimeMillis());
            if(Thread.currentThread().isInterrupted()){
                P.l(Thread.currentThread().getName()+ "
interrupted");
                break;
            }
        }
    }
}
}

```

7. 线程池的结束状态

线程池内线程运行结束的标志

线程池状态

- **isShutdown:** 用来判断线程池是否已经关闭，只要执行了shutdown方法这个方法就会返回true，但是并不代表任务已经都执行完毕了
- **isTerminated:** 任务全部执行完毕，并且线程池已经关闭，才会返回true
- **awaitTermination** 阻塞（阻塞参数指定的时间去看线程池内线程运行的状态），直到所有任务在关闭请求后完成执行，或发生超时，或当

前线程中断（以先发生者为准）。

Demo:

com.mkevin.demo14.ThreadPoolDemo2

8. 允许核心线程超时策略

核心线程也允许超时销毁，但是需要我们手动设置，默认是没有超时销毁的。

在调用`ThreadPoolExecutor`的构造方法创建线程池的时候，给这个类传递的参数中的`core`指的就是 **核心线程数**

核心线程 + 应急线程：这样的线程是常驻线程池内部的，是不会被销毁的，这样的线程处理完各自的任务之后会依然停留在线程池内部，如果没有新的任务提交进来，那么它们依旧保持空闲状态；如果创建好线程池之后（创建`ThreadPoolExecutor`实例对象时指定的参数是：`3,4,120,queue`），有一个任务被提交进来，那么线程池中的3个核心线程中的某个线程就会运行这个任务，如果又一次性提交了2个任务，导致线程池中的核心线程都在运行，这时一个新的任务（记作Q）被提交进来，就需要去看构造`ThreadPoolExecutor`的时候传递的第二个参数`max`（上面的4），如果 $3 + N(Q)$ 的个数没有超过`max`（上面的4）这个时候就会在线程池中再创建一个线程（称之为应急线程），而这个新创建的线程有空闲的指定时间：`keepAliveTime`（也就是上面的120），当这个新创建的应急线程执行完Q任务之后的`keepAliveTime`时间内没有被分配到别的任务，那么该线程就会被销毁。

以上说的核心线程的属性是默认的，再这一节中我们可以让核心线程也像应急线程一样有超时策略

允许核心线程超时策略

- 核心线程也允许销毁，`allowsCoreThreadTimeOut`就用来做这个事
 - 设置控制核心线程是否可能超时的策略，如果在保持活动时间内没有任务到达，则该策略将在新任务到达时根据需要被替换。下面是参数值对应的功能：
 - 如果为`false`，则不会由于缺少传入任务而终止核心线程。
 - 如果为`true`，则应用于非核心线程的相同保持活动策略也适用于核心线程。
 - 为避免连续更换线程，设置为`true`时保持活动时间必须大于零。
- 通常应该在池被激活（提交任务）之前调用此方法。

Demo:

com.mkevin.demo16.allowCoreThreadTimeOutDemo

9. 核心线程预启动策略

核心线程预启动

核心线程预启动策略

- 默认情况下，核心线程只有在任务提交的时候才会创建；而预启动策略，可以让核心线程提前启动，从而增强最初提交的线程运行性能
- **prestartCoreThread**: 调用一次该方法启动1个核心线程，返回值为**true**和**false**，分别代表预启动成功是失败。覆盖仅在执行新任务时启动核心线程的默认策略。如果所有核心线程都已启动，则此方法将返回**false**。
- **prestartAllCoreThreads**: 调用一次该方法启动所有核心线程，返回值为启动成功的线程的数量。覆盖仅在执行新任务时启动核心线程的默认策略。如果核心线程全部启动后再次调用，则会返回0

Demo:

com.mkevin.demo17.prestartAllCoreThreadsDemo

10. 自定义线程及线程池切面

切面

线程及线程池切面

- 在线程执行前、执行后增加切面，在线程池关闭时执行某段程序。
- 需要实现自己的线程池类（继承ThreadPoolExecutor），并覆写beforeExecute、afterExecute、terminated方法

Demo:

com.mkevin.demo19.BeforAfterTerminatedDemo

11. 移除线程池中的任务

怎样删除线程池中的任务

移除线程池中的任务

- 使用线程池的**remove**方法
- 已经正在运行中的任务不可以删除，该方法会返回**false**
- **execute**方法提交的，未运行的任务可以删除
- **submit**方法提交的，未运行任务也不可以删除，小心采坑！

Demo:

com.mkevin.demo20.TaskRemoveDemo

12. 获取各种线程池状态数据

大量的get方法怎么玩？

获取各种线程池状态数据

可以获取线程池的各种动态和静态数据，用于程序控制。

- 返回当前线程池在创建时指定的核心线程数：getCorePoolSize
- 返回当前线程池中的线程数：getPoolSize
- 返回最大允许的线程数：getMaximumPoolSize
- 返回池中同时存在的最大线程数：getLargestPoolSize
- 返回预定执行的任务总和：getTaskCount 【这个方法得到的是一个预估值，不一定准确】
- 返回当前线程池已经完成的任务数：getCompletedTaskCount
- 返回正在执行任务的线程的大致数目：getActiveCount
- 返回线程池空闲时间：getKeepAliveTime 【调用的时候传入一个参数：时间的单位】

Demo:

com.mkevin.demo18.ThreadPoolGetDemo

13. 设计模式-单例模式

- 饿汉模式：类加载的时候，就进行对象的创建，系统开销较大，但是不存在线程安全问题。

饿汉示例： DemoThread22

示例化简后的形式

```
class Singleton1 {  
    private static Singleton1 singleton = new  
    Singleton1(); // 建立对象  
    private Singleton1() {}  
}
```

- 懒汉模式：多数采用饿汉模式，在使用时才真正的创建单例对象，但是存在线程安全问题
 - 懒汉示例： DemoThread23 （线程安全问题和解决方案）
化简后的存在线程安全问题的示例：

```

class Singleton2 {
    private static Singleton2 singleton =
null; // 不建立对象
    private Singleton2() {
    }
    /*synchronized 可以解决线程安全问题,但是
存在性能问题,即使singleton!=null也需要先获得锁
*/
    public /*synchronized*/ static
Singleton2 getInstance() {
        if (singleton == null) { // 先判断
是否为空
            //一系列操作...
            singleton = new Singleton2();
        } // 懒汉式做法
        return singleton;
    }
}

```

线程安全问题存在的原因可以去看《Java核心技术卷I 第十二章 并发.xmind》中对volatile关键字的介绍，导致多个线程分别创建多个不同的实例对象。

解决示例中线程安全问题最简单的方式就是直接在方法上使用synchronized关键字

- 懒汉模式： DemoThread24 （线程安全的性能优化），在这个示例中没有使用volatile关键字保证对象在创建的时候保持指令的有序性，可能会在使用这个对象的时候出现问题，相关东西可以去看《Java核心技术卷I 第十二章 并发.xmind》中堆volatile关键字的介绍。
- 静态内部类单例：兼具懒汉模式和饿汉模式的优点
 - 静态内部类单例： DemoThread25

```

/**静态内部类-单例对象构建方法
(
利用JDK的特性：类级内部类只有在第一次被使用的时候才会被装载
（被初始化），
这样可以保证单例对象只有在第一次被使用的时候初始化一次，
并且不需要加锁，性能得到大大提高，并且保证了线程安全。
)
*/
class Singleton4 {
    private static class InnerSingleton {
        private static Singleton4 single = new
Singleton4();
    }

    private Singleton4(){}
}

```

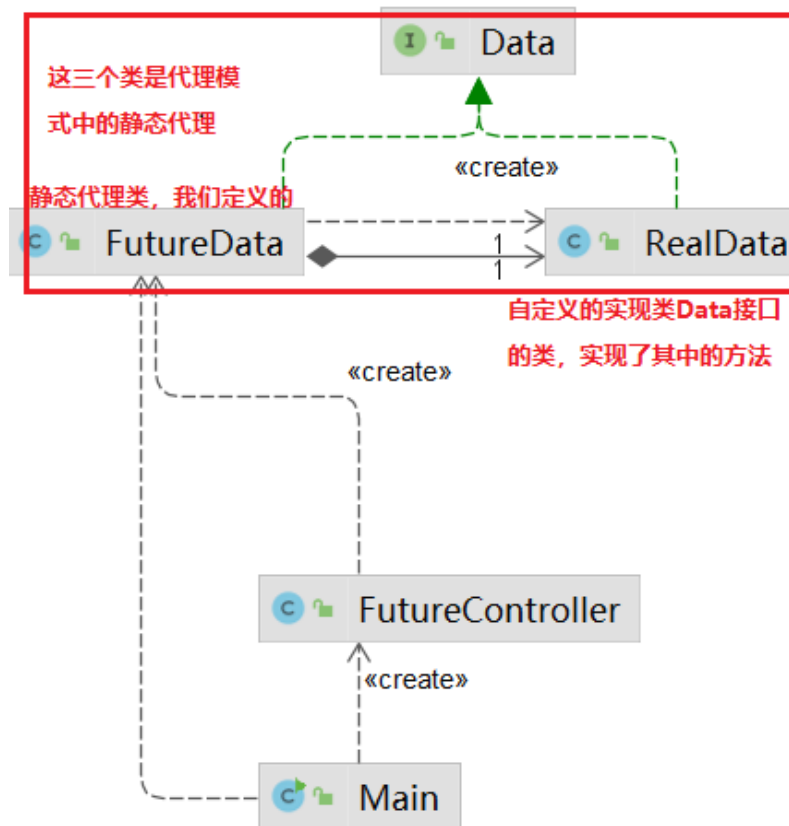
```

public static Singleton4 getInstance(){
    return InnerSingleton.single;
}
}

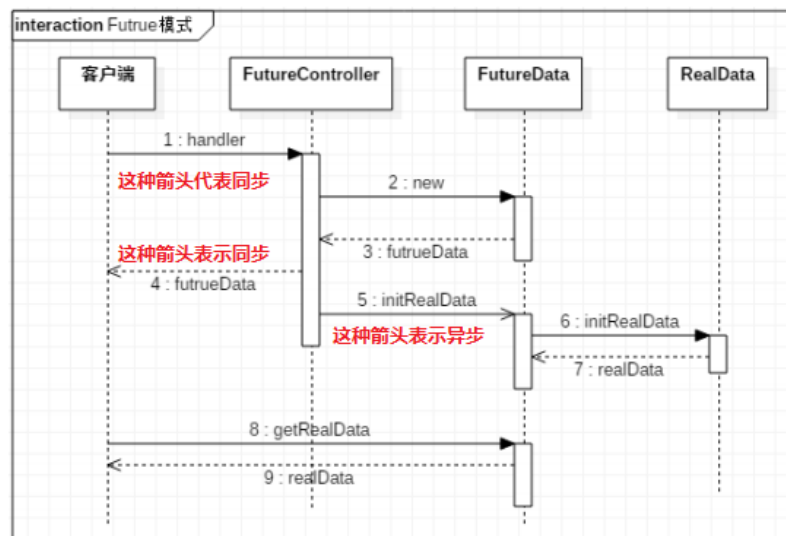
```

14. 设计模式-Future

- 简单来说,客户端请求之后,先返回一个应答结果,然后异步的去准备数据,客户端可以先去处理其他事情,当需要最终结果的时候再来获取,如果此时数据已经准备好,则将真实数据返回;如果此时数据还没有准备好,则阻塞等待。
- 模拟Future设计模式的示例: com.mimaxueyuan.demo.high.futtrue包下的Main类是程序的入口
在该包下对于Future设计模式的模拟的类图:



这是Main类执行的时序图:



- JDK的Concurrent包提供了Futrue模式的实现，可以直接使用。
- 使用Futrue模式需要实现Callable接口，并使用FutureTask进行封装，使用线程池进行提交。
- 示例：com.mimaxueyuan.demo.high.futrue.JdkFuture
在这个类中使用JDK提供的Future模式，最终的表现结果和上面模拟Future模式一样；
代码：

```

public class JdkFuture implements
Callable<String>{
    private String para;

    public JdkFuture(String para){
        this.para = para;
    }

    /**
     * 这里是真实的业务逻辑，其执行可能很慢
     */
    @Override
    public String call() throws Exception {
        //模拟执行耗时
        Thread.sleep(5000);
        String result = this.para + "处理完成";
        return result;
    }

    //主控制函数
    public static void main(String[] args) throws
Exception {
        String queryStr = "zhangsan";

        //构造FutureTask，并且传入需要真正进行业务逻辑
        处理的类,该类一定是实现了Callable接口的类
        FutureTask<String> future = new
FutureTask<String>(new JdkFuture(queryStr));
        FutureTask<String> future2 = new
FutureTask<String>(new JdkFuture(queryStr));
  
```

//创建一个固定线程的线程池且线程数为1,上面创建的
FutureTask任务可以放在这个线程池中。

```
ExecutorService executor =  
Executors.newFixedThreadPool(2);
```

//这里提交任务future,则开启线程执行JdkFuture的
call()方法执行

//submit和execute的区别: 第一点是submit可以
传入实现Callable接口的实例对象, 第二点是submit方法有返
回值

```
Future f1 = executor.submit(future);
```

//单独启动一个线程去执行的

```
Future f2 = executor.submit(future2);
```

```
System.out.println("请求完毕");
```

```
try {
```

//这里可以做额外的数据操作,也就是主程序执行
其他业务逻辑

```
System.out.println("处理实际的业务逻辑...");
```

```
Thread.sleep(1000);
```

```
} catch (Exception e) {
```

```
e.printStackTrace();
```

```
}
```

//调用获取数据方法,如果call()方法没有执行完成,则
依然会进行等待

```
System.out.println("数据: " +  
future.get());
```

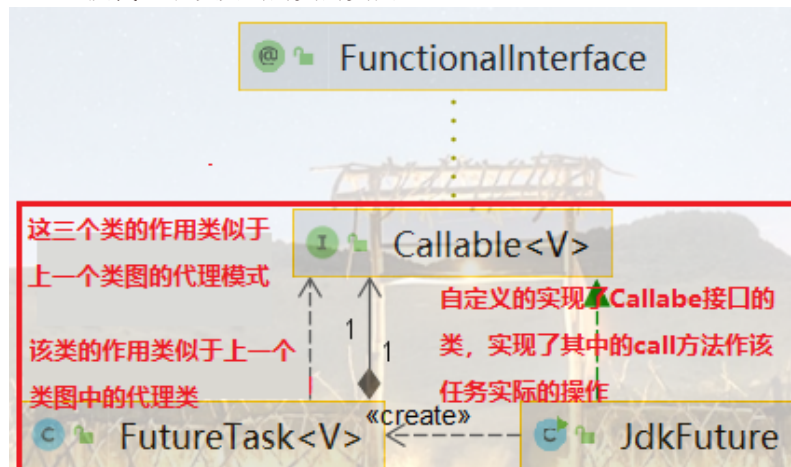
```
System.out.println("数据: " +  
future2.get());
```

```
executor.shutdown();
```

```
}
```

```
}
```

这段代码中涉及到的类的类图:

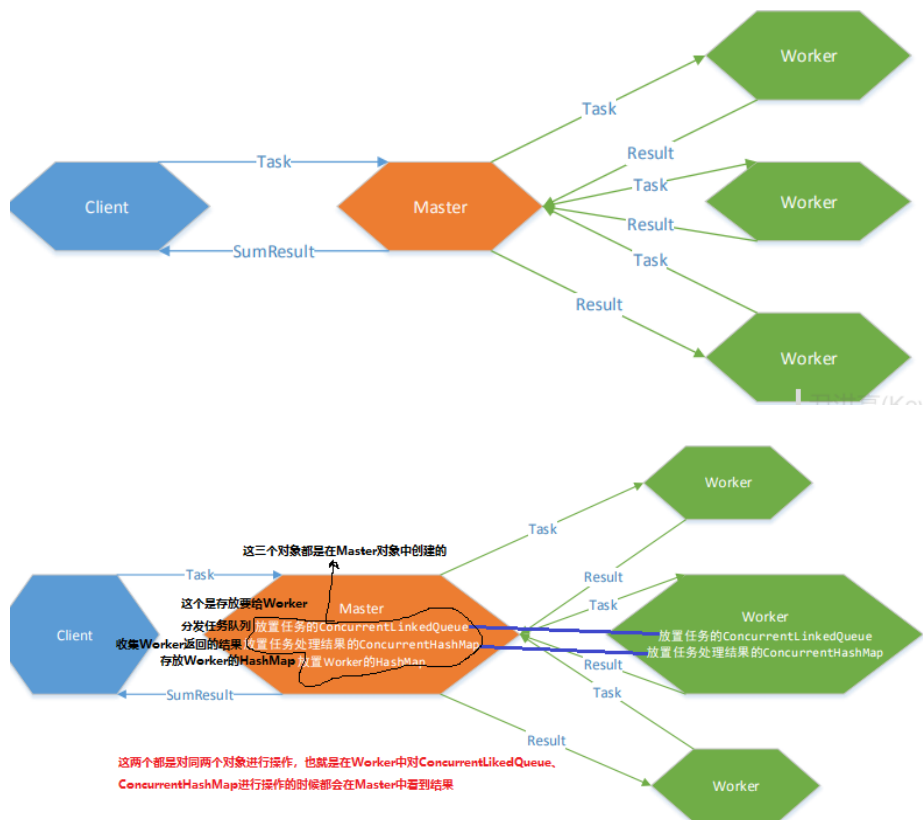


15. 设计模式-Producer-Consumer

- Producer-Consumer称为生产者消费者模式，是消息队列中间件的核心实现模式，ActiveMQ、RocketMQ、Kafka、RabbitMQ。
- 示例： `com.mimaxueyuan.demo.high.mq.Main`

16. 设计模式-Master-Worker

- Master-Worker模式是一种将串行任务并行化的方案，被分解的子任务在系统中可以被并行处理，同时，如果有需要，Master（这个Master不是一个任务，没有实现Runnable/Callable接口）进程不需要等待所有子任务都完成计算，就可以根据已有的部分结果集计算最终结果集。
- 客户端将所有任务提交给Master，Master分配Worker去并发处理任务，并将每一个任务的处理结果返回给Master，所有的任务处理完毕后，由Master进行结果汇总再返回给Client
- Master和Worker共享两个对象，通过这两个对象可以让它们进行交互；这两个对象分别是：一个存放Task的CurrentLinkedQueue队列、一个存放结果的CurrentHashMap；
责任：
 - a. Master:
 - i. 将任务（注意：这里说的任务并不是实现了一个Runnable/Callable接口的类的实例对象）存放在CurrentLinkedQueue队列中；
 - ii. 同时将用户传递进来的Worker封装到不同的Thread中【注意：在代码中创建Master时只传递了一个Worker的实例对象，但是创建了多个Thread，它们都封装的是同一个Worker】，并把这个Thread和这个线程对应的名字封装到一个HashMap中，线程名作为Map的key，线程的实例对象作为Map的value；
 - iii. 从Worker存放任务结果的ConcurrentHashMap中取出结果
 - b. Worker:
 - i. 从Master存放任务（注意：这个任务并不是实现了Runnable/Callable接口的类的实例对象）的CurrentLinkedQueue队列中取出任务，并调用自己的方法处理这个任务
 - ii. 向ConcurrentHashMap中存放当前Worker实例对象对已获得的任务的处理结果
- 示例： `com.mimaxueyuan.demo.high.masterworker.Main`



17. CompletionService

轻松搞定Master Worker模式

CompletionService接口



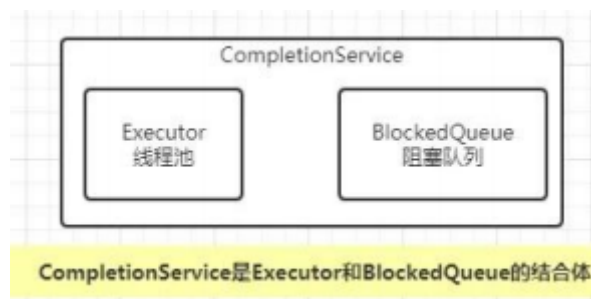
这张图说明了CompletionService的使用模式

你可用它不断的提交任务（线程）给Executor处理

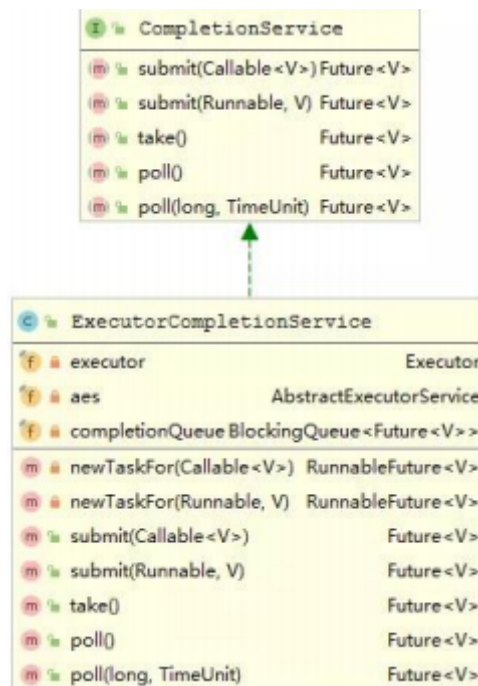
处理后的结果都会自动放入BlockedQueue另外一个线程不断的从队列里取得处理结果

好处是，哪个任务先处理完就能先得到哪个结果最后做汇总处理

从而轻松完成MasterWorker模式相同的功能



这张图说明了CompletionService的本质：就是线程池Executor加上阻塞队列BlockedQueue，线程池Executor用来处理任务，BlockedQueue用来存储每个线程的运行结果。



- submit用于提交任务
- take用于获取处理结果（阻塞式），得到一个Future对象
- poll也用于获取处理结果（非阻塞式）

Demo :

com.mkevin.demo11.CompletionServiceDemo0

com.mkevin.demo11.CompletionServiceDemo1（这个示例的目的是：证明在使用take获取结果的时候，即使提交的任务的call方法中抛出了异常，也会将每个线程的运行结果存放在阻塞队列中，只是在调用take方法返回的Future对象的get方法获得值的时候才会报异常）

com.mkevin.demo11.CompletionServiceDemo2（这个示例的目的是：展示在使用submit(Runnable,V)的时候怎么获得返回结果，因为Runnable接口中的run方法是没有返回值的；可以通过在Runnable接口的实现类中加入一个“存放结果对象（可以使是们自定义的对象）”，在run方法运行结束的时候，会把运行的结果存放在这个“存放结果”的对象中）

```

/**
 * 让Runnable也具有获得结果的特性
 */

```

```

public class CompletionServiceDemo2 {
    public static void main(String[] args) {
        try {
            //创建结果对象，用于获取结果
            Result result = new Result();
            ExecutorService executorService =
Executors.newCachedThreadPool();
            CompletionService cs = new
ExecutorCompletionService(executorService);
            //创建Runnable对象，在创建SalaryRunner2类的实例对
象的时候，会把存放结果的Result的对象
            //的引用传递到SalaryRunner2类中，在SalaryRunner2
类的run方法执行完毕的时候会对这个对象
            //进行赋值
            SalaryRunner2 runner1 = new
SalaryRunner2(result, 10, 1000);
            SalaryRunner2 runner2 = new
SalaryRunner2(result, 20, 2000);
            //提交对象，并使用result对象接收结果，这个对象在
SalaryRunner2类中也有定义
            Future<Result> f1 = cs.submit(runner1,result);
            Future<Result> f2 = cs.submit(runner2,result);
            //获取结果
            System.out.println("f1:" +
f1.get().getValue());
            System.out.println("f2:" +
f2.get().getValue());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

/**
 * 结果计算
 */
public class SalaryRunner2 implements Runnable {

    //结果,因为这个类实现的Runnable接口中的run方法没有返回值
    //所以，可以在这个类中增加一个存放结果的对象，这个对象是
    //通过参数被初始化的，在run方法执行完的时候会给这个对象的
    //属性赋值
    private Result result;
    //工资
    private long salary;
    //耗时
    private long costTime;

    public SalaryRunner2(Result result,long salary, long
costTime){
        this.result = result;
    }
}

```

```

        this.costTime = costTime;
        this.salary=salary;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(this.costTime);
            this.result.setValue(this.salary*1000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

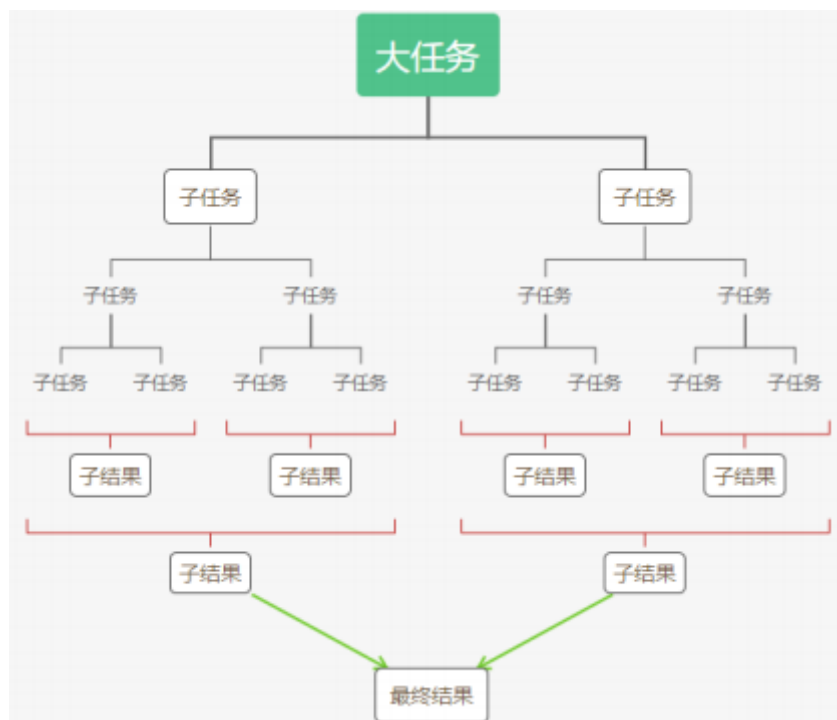
```

com.mkevin.demo11.CompletionServiceDemo3（这个示例中展示了：一个线程向CompletionService的线程池中不断地提交任务，另一个线程不断地从CompletionService地阻塞队列中获取）

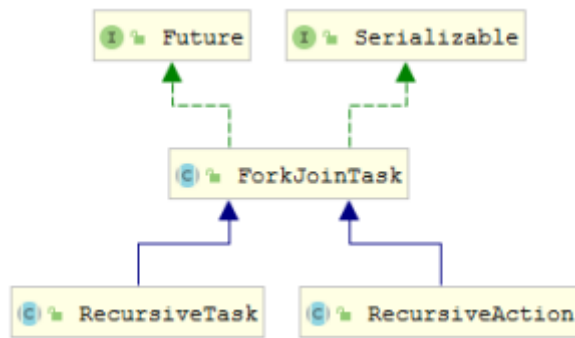
18. ForkJoin

ForkJoin模式的使用

ForkJoin思想



ForkJoin使用



- 两个重要的实现类，又同时是抽象类
- `java.util.concurrent.RecursiveTask` 递归任务类
- `java.util.concurrent.RecursiveAction` 递归活动类
- ForkJoin的使用方式：
 - 先创建一个ForkJoinPool
 - 之后创建RecursiveTask或RecursiveAction的子类的实例对象提交到ForkJoinPool中

Demo:

`com.mkevin.demo12.ForkJoinDemo0` 至 `ForkJoinDemo7`

ForkJoinDemo0: 展示了简单的ForkJoinTask的子类RecursiveAction的使用

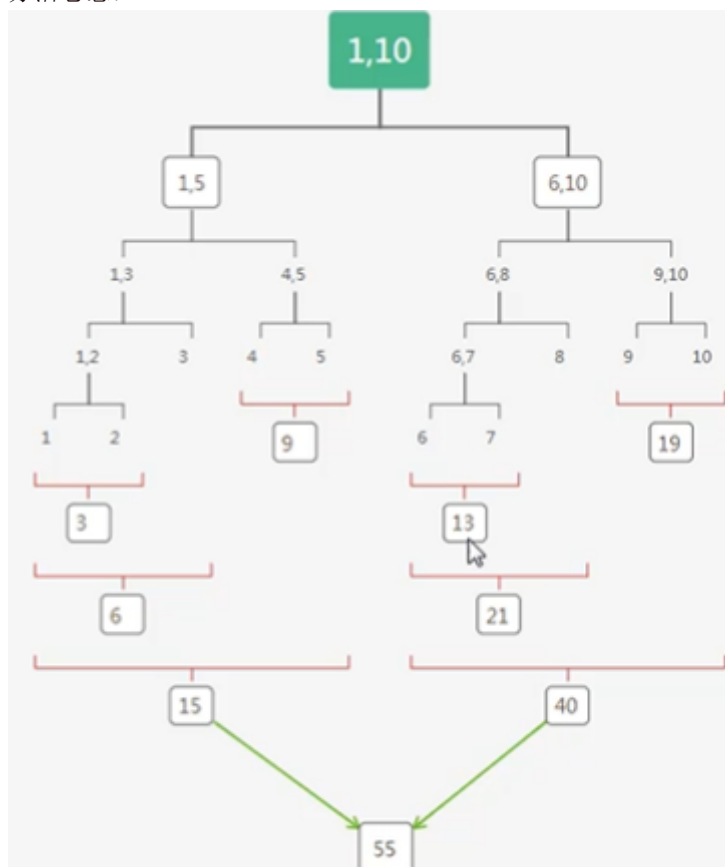
ForkJoinDemo1: 展示了怎么获得RecursiveTask类中compute方法的执行结果

ForkJoinDemo2: 展示了提交到ForkJoinPool中的任务可以是不同的任务

ForkJoinDemo3: 没有这个类

ForkJoinDemo4: 分治模型的典型例子，展示了计算 $1+2+3+4+\dots+10$ 使用ForkJoinTask的情况

分治思想：



窃取算法：别的线程没做完的事情，我在做完了我自己的事情之后去帮他们做。

ForkJoinDemo5: ForkJoinPool可以直接执行Runnable类型的任务（需要获取返回结果）

ForkJoinDemo6

ForkJoinDemo7: 说明ForkJoinPool中也有shutdown方法