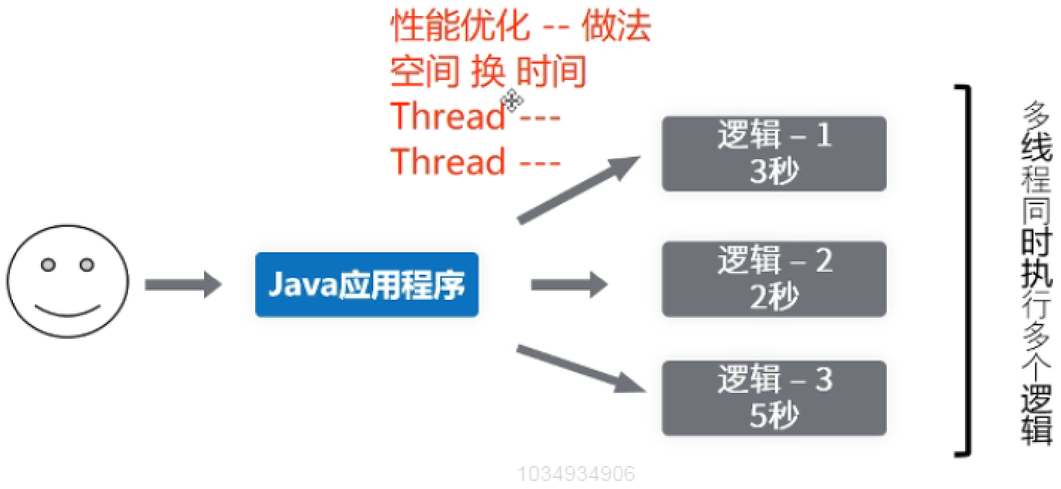
多线程知识扩展

多线程应用



总的执行时间，取决于执行最慢的逻辑。

逻辑之间无依赖关系，可同时执行，则可以应用多线程技术进行优化。

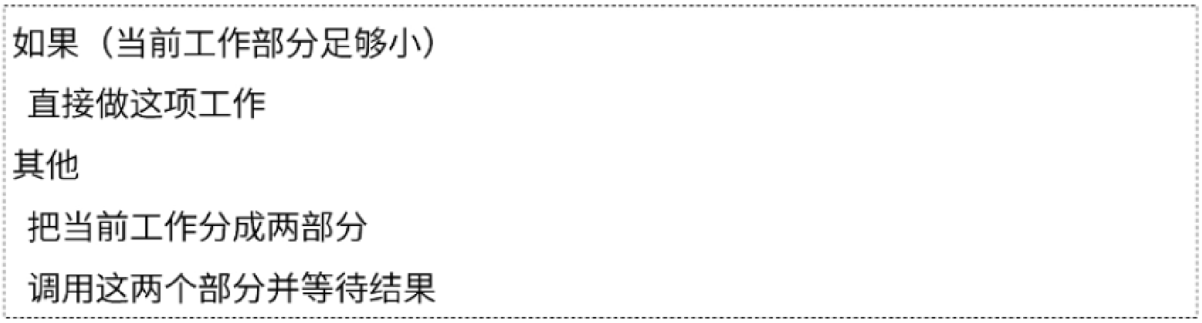
自己实现一个FutureTask

```
public class MyFutureTask<T> implements Runnable {

    private Callable<T> callable;

    private T result;

    private volatile String state = "NEW";

    private LinkedBlockingQueue<Thread> waiters = new LinkedBlockingQueue<>();

    public MyFutureTask(Callable<T> callable) {
        this.callable = callable;
    }

    @Override
    public void run() {
        try {
            result = callable.call();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            state = "OVER";
        }

        System.out.println(Thread.currentThread() + " 生产者生产，唤醒消费者，拿到结果");
//        while (true) {
//            Thread thread = waiters.poll();
//            if (null == thread) {
//                break;
//            }
//            LockSupport.unpark(thread);
//        }
        Thread waiter = waiters.poll();
        while (null != waiter) {
            LockSupport.unpark(waiter);
            waiter = waiters.poll();
        }
    }

    public T get() {
        Thread thread = Thread.currentThread();
        waiters.add(thread);
        while (!"OVER".equals(state)) {
            System.out.println(Thread.currentThread() + " 消费者线程被阻塞");
            LockSupport.park(thread);
        }
        return result;
    }
}

public class MyFutureTaskDemo {

    private static ExecutorService executorService = Executors.newSingleThreadExecutor();

    public static void main(String[] args) throws ExecutionException, InterruptedException
{
        Callable<Integer> callable = new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                Thread.sleep(3000);
                return 1;
            }
        };

        MyFutureTask<Integer> myFutureTask = new MyFutureTask<>(callable);

        new Thread(myFutureTask).start();
        System.out.println("结果: " + myFutureTask.get());
    }
}
```

ForkJoinPool

1. ForkJoinPool是ExecutorService接口的实现，它专为可以递归分解成小块的工作而设计。Fork/Join框架将任务分配给线程池的工作线程，充分利用了多处理器的优势，提高程序性能。

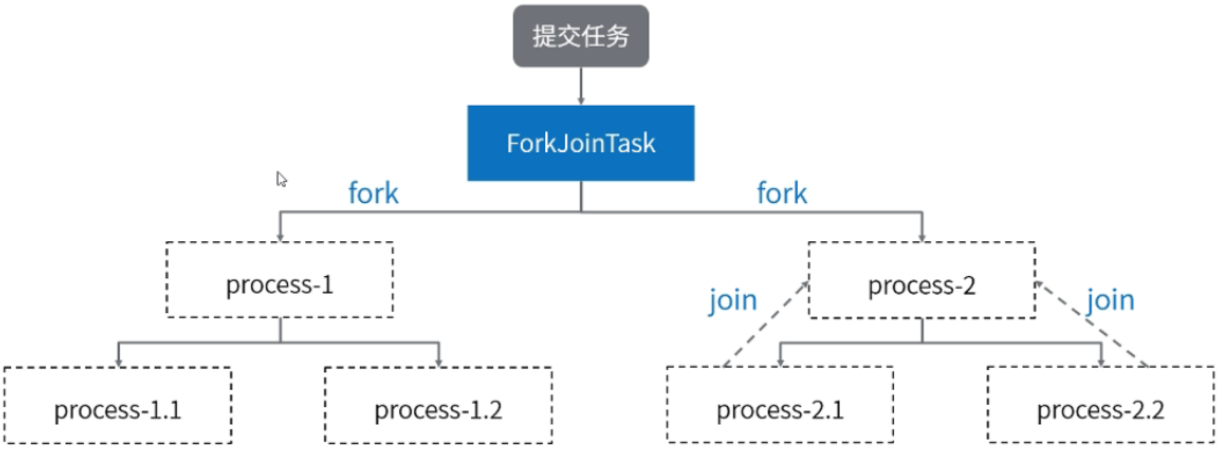
2. 使用Fork/Join框架的第一步是编写执行一部分工作的代码。类似的伪代码如下：



```
如果（当前工作部分足够小）
 直接做这项工作
其他
 把当前工作分成两部分
 调用这两个部分并等待结果
```

将此代码包装在ForkJoinTask子类中，通常是RecursiveTask(可以返回结果)，或RecursiveAction。

3. 关键是分解任务fork出新任务，汇集join任务执行结果。



```
   public class ForkJoinDemo {
       public static void main(String[] args) throws ExecutionException,
InterruptedException {
           ForkJoinDemo forkJoinDemo = new ForkJoinDemo();
           List<Integer> list = new ArrayList<>();
           for (int i = 1; i <= 100; i++) {
               list.add(i);
           }

           ForkJoinPool forkJoinPool = new ForkJoinPool();
           Future<Integer> future = forkJoinPool.submit(forkJoinDemo.new SumTask(list, 1,
list.size()));
           long start = System.currentTimeMillis();
           int result = future.get();
           System.out.println("结果: " + result + "耗时: " + (System.currentTimeMillis() -
start));
           System.out.println();
       }

       class SumTask extends RecursiveTask<Integer> {

           private List<Integer> elements;
```

```
        private int start;

        private int end;

        public SumTask(List<Integer> emelemts, int start, int end) {
            this.elements = emelemts;
            this.start = start;
            this.end = end;
        }

        @Override
        protected Integer compute() {
            int step = end - start;
            // 任务足够小则直接执行
            if (step == 24) {
                System.out.println(Thread.currentThread() + "任务足够小直接执行");
                int result = 0;
                for (int i = start; i <= end; i++) {
                    result += i;
                }
                return result;
            } else {
                // 继续拆分任务
                System.out.println(Thread.currentThread() + "任务不够小拆分一次");
                int x = (start + end) / 2;
                SumTask subTask = new SumTask(elements, start, x);
                subTask.fork();

                SumTask subTask2 = new SumTask(elements, x + 1, end);
                subTask2.fork();

                int result = 0;
                result += subTask.join();
                result += subTask2.join();
                return result;
            }
        }
    }
}

Thread[ForkJoinPool-1-worker-1,5,main]任务不够小拆分一次
Thread[ForkJoinPool-1-worker-3,5,main]任务不够小拆分一次
Thread[ForkJoinPool-1-worker-3,5,main]任务足够小直接执行
Thread[ForkJoinPool-1-worker-3,5,main]任务足够小直接执行
Thread[ForkJoinPool-1-worker-2,5,main]任务不够小拆分一次
Thread[ForkJoinPool-1-worker-2,5,main]任务足够小直接执行
Thread[ForkJoinPool-1-worker-2,5,main]任务足够小直接执行
结果：5050耗时：7
```