链滴

Fork/Join 框架 & CountDownLatch 与 CyclicBarrier

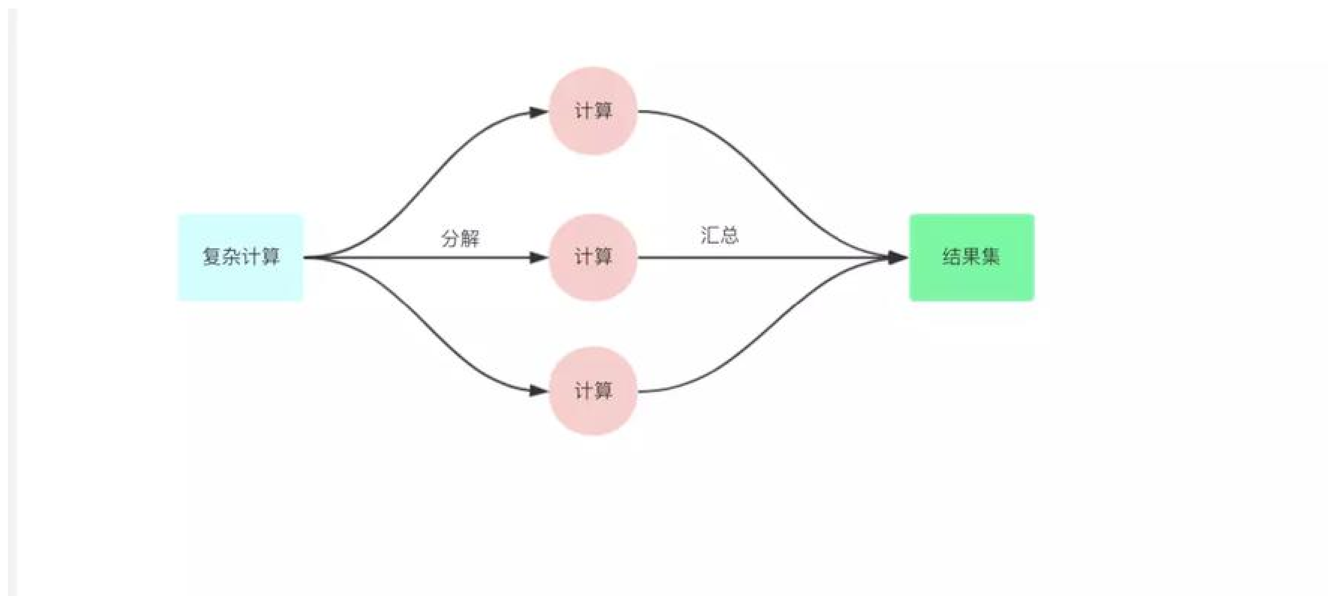
作者：DongXiaokai0819
原文链接：https://ld246.com/article/1571211710379
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

Java并发工具类

Fork/Join "分而治之"

“分而治之”：就是将一个复杂的计算，按照设定的阈值分解成多个小计算，然后将各个小计算的计结果进行汇总。

Fork/Join框架：就是在必要的情况下，将一个大任务，进行拆分（fork）成若干个小任务（拆到不可拆时，即达到阈值以下）再将一个个小任务计算的结果进行join汇总。

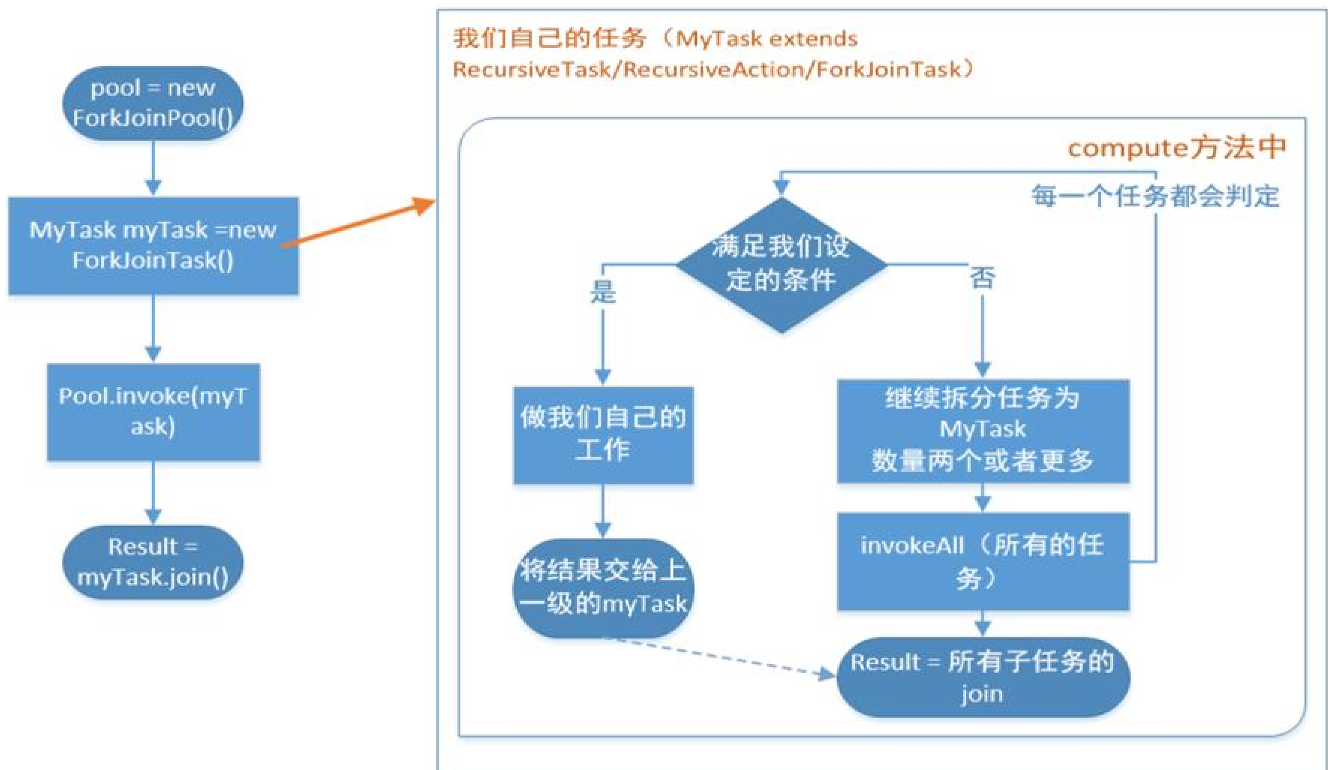


工作窃取特性

Fork/Join具有 工作窃取(workStealing) 特性, 即每个线程会维护一个双端队列, 默认是从尾部取任务, 当一个线程的工作队列为空时, 它会去其他线程的工作线程尾部窃取一个工作任务. 这种机制可以让线的限制时间减少, 提升程序效率, 并且减少获取工作任务的阻塞时间.

Fork/Join使用范式

原文链接：Fork/Join 框架 & CountDownLatch 与 CyclicBarrier



需要使用Fork/Join模式的类，需要继承自RecursiveTask（无返回值）、RecursiveAction（有返回），然后在compute方法内实现“分而治之”的逻辑就可以。

RecursiveTask（无返回值）、RecursiveAction（有返回值）这俩个类都是继承自ForkJoinTask类。

Fork/Join 使用步骤:

1. 编写RecursiveTask（无返回值）、RecursiveAction（有返回值）的ForkJoinTask任务类。
2. 在任务类的compute方法实现分儿治之的模式

- fork()：在任务执行过程中将大任务划分为多个小的子任务，调用子任务的fork()方法可以将任务放线程池中异步调度。其实这里执行子任务调用fork方法并不是最佳的选择，最佳的选择是invokeAll(lefTask,rightTask)方法。
- join()：调用子任务的join()方法等待任务返回的结果。这个方法类似于Thread.join()，区别在于前不受线程中断机制的影响。

3. 在主线程中声明ForkJoinPool线程池
4. 将任务类传入线程池对象，然后进行task调用，这里有三种调用方式

- forkJoinPool.execute(forkJoinTask) 异步执行tasks，无返回值
- forkJoinPool.invoke(forkJoinTask) 在当前线程同步执行该任务。该方法也不受中断机制影响。
- forkJoinPool.submit(forkJoinTask) 异步执行，且带Task返回值，可通过task.get 实现同步到主程

Fork/Join实战

有返回值，同步调用forkJoinTask方法，统计数组和

```
public class SumArray {
```

```
    private static class SumTask extends RecursiveTask<Integer>{

        private final static int THRESHOLD = MakeArray.ARRAY_LENGTH/10;//阈值
        private int[] src; //表示我们要实际统计的数组
        private int fromIndex;//开始统计的下标
        private int toIndex;//统计到哪里结束的下标

        public SumTask(int[] src, int fromIndex, int toIndex) {
            this.src = src;
            this.fromIndex = fromIndex;//开始下标
            this.toIndex = toIndex;//结束下标
        }

        @Override
        protected Integer compute() {
            if(toIndex-fromIndex < THRESHOLD) {//如果结束下标-开始下标<阈值的话，就不用再进
拆分了
                int count = 0;
                for(int i=fromIndex;i<=toIndex;i++) {
                    count = count + src[i];
                }
                return count;
            }else {//大于阈值，继续拆分
                int mid = (fromIndex+toIndex)/2;
                SumTask left = new SumTask(src,fromIndex,mid);//左边的任务
                SumTask right = new SumTask(src,mid+1,toIndex);//右边的任务
                invokeAll(left,right);//调用子任务
                return left.join()+right.join();//递归调用
            }
        }
    }


    public static void main(String[] args) {

        ForkJoinPool pool = new ForkJoinPool();
        int[] src = MakeArray.makeArray();//获得一个很大的满是数字的数组
        SumTask innerFind = new SumTask(src,0,src.length-1);

        long start = System.currentTimeMillis();
        pool.invoke(innerFind);//同步调用，启动forkJoinTask
        System.out.println("Task is Running.....");
        System.out.println("The count is "+innerFind.join()
                +" spend time:"+(System.currentTimeMillis()-start)+"ms");
    }
}
```

CountDownLatch 闭锁

- 一组线程等待其他的相关线程完成工作以后再执行，加强版join

```
public CountDownLatch(int count) { if (count < 0) throw new IllegalArgumentException("coun
 < 0"); this.sync = new Sync(count); }
```

原文链接：Fork/Join 框架 & CountDownLatch 与 CyclicBarrier

具体使用步骤

1. 初始化CountDownLatch（int count） 传入int值，表明有几个扣除点

2. 在需要等待其它线程完成工作后才可继续执行任务的线程处，调用countDownLatch.await()方法该方法的作用是等待count为0时，该线程才可继续执行任务。可在多个线程内部调用countDownLatch.await()方法，则这些线程都要等待count扣除点值为0时才可继续执行任务

3. 在相关初始化线程或者业务线程内，调用countDownLatch.countDown()来使得初始化countDownLatch时的count扣除点-1。

下面来看一下这个例子，这个例子有四个扣除点，有俩个线程设置了await()

```
public class UseCountDownLatch {
    static CountDownLatch countDownLatch = new CountDownLatch(4);//初始化CountDownLach，并设置了四个扣除点

    private static class initClass extends Thread{
        private CountDownLatch countDownLatch;
        public initClass(CountDownLatch countDownLatch){
            this.countDownLatch = countDownLatch;
        }

        @Override
        public void run() {
            System.out.println("初始化相关工作。。。。。");
            try {
                Thread.sleep(2000);
                countDownLatch.countDown();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private static class businessClass extends Thread{
        private CountDownLatch countDownLatch;
        public businessClass(CountDownLatch countDownLatch){
            this.countDownLatch = countDownLatch;
        }

        @Override
        public void run() {
            try {
                System.out.println("完成相关业务，需要其它线程的初始化完成");
                countDownLatch.await();;//等待点
                System.out.println("其它线程工作完成，业务线程继续工作。。。");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        new Thread(()->{
```

```
            try {
                System.out.println("干了一些事情。。。。");
                Thread.sleep(1000);
                countDownLatch.countDown();
                System.out.println("又干了一些事情。。。。");
                Thread.sleep(1000);
                countDownLatch.countDown();//每完成一步工作，扣减一次
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
        new Thread(new businessClass(countDownLatch)).start();
        for (int i=0; i<2; i++){
            new Thread(new initClass(countDownLatch)).start();
        }
        countDownLatch.await();//等待点
        System.out.println("主线程接着开始工作。。。。");
    }
}
```

输出如下：

```
干了一些事情。。。。
初始化相关工作。。。。。
初始化相关工作。。。。。
完成相关业务，需要其它线程的初始化完成
又干了一些事情。。。。
主线程接着开始工作。。。。
其它线程工作完成，业务线程继续工作。。。
```

可以看出，只有当count值减为0之后，俩个等待的线程才继续开始工作

CyclicBarrier 循环栅栏

● 让一组线程达到某个栅栏，被阻塞，一直到组内最后一个线程达到栅栏时，屏障开放，所有被阻塞线程会继续运行。

CyclicBarrier 有俩个构造方法

1. CyclicBarrier(int parties) 传入通过栅栏所需的一组线程的数量

2. CyclicBarrier(int parties, Runnable barrierAction)传入通过栅栏所需的一组线程的数量，和到达定的数量之后，新开一个线程完成相应工作

具体使用步骤：

1. 初始化CyclicBarrier (int parties, Runnable barrierAction))

2. 在需要这组线程共同等待的栅栏处调用cyclicBarrier.await()方法

3. 编写通过栅栏后所需要完成什么工作的线程类

下面来看一下这个例子，这个例子将parties设置为5，意为只有当五个线程都到达cyclicBarrier.await(处时，五个线程才可以继续往下执行。当栅栏放行后，栅栏会重置！

```
public class UseCyclicBarrier {

    private static CyclicBarrier barrier
        = new CyclicBarrier(5,new CollectThread());

    private static ConcurrentHashMap<String,Long> resultMap
            = new ConcurrentHashMap<>();//存放子线程工作结果的容器

    public static void main(String[] args) {
        for(int i=0;i<=4;i++){
            Thread thread = new Thread(new SubThread());
            thread.start();
        }

        for(int i=0;i<=4;i++){
            Thread thread = new Thread(new SubThread());
            thread.start();
        }

    }

    //负责屏障开放以后的工作
    private static class CollectThread implements Runnable{

        @Override
        public void run() {
            System.out.println("栅栏放行！，重置栅栏！");
            StringBuilder result = new StringBuilder();
            for(Map.Entry<String,Long> workResult:resultMap.entrySet()){
                result.append("["+workResult.getValue()+"]");
            }
            System.out.println(" the result = "+ result);
            System.out.println("do other business........");
        }
    }

    //工作线程
    private static class SubThread implements Runnable{

        @Override
        public void run() {
            long id = Thread.currentThread().getId();//线程本身的处理结果
            resultMap.put(Thread.currentThread().getId()+"",id);
            Random r = new Random();//随机决定工作线程的是否睡眠
            try {
                if(r.nextBoolean()) {
                    Thread.sleep(2000+id);
                    System.out.println("Thread_"+id+" ....do something ");
                }
                System.out.println(id+"....is await");
                barrier.await();
                Thread.sleep(1000+id);
                System.out.println("Thread_"+id+" ....do its business ");
            } catch (Exception e) {
```

```
                e.printStackTrace();
            }

        }
    }
}
```

使用场景：

CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个Excel保存用户所有银行流水，每个Sheet保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日均银行流水，最后再用barrierAction用这些线程的计算结果，计算出整个Excel的日均银行流水。

辨析CountDownLatch和CyclicBarrier

1. Countdownlatch放行由第三者控制，CyclicBarrier放行由一组线程本身控制
2. Countdownlatch放行条件 >= 线程数，CyclicBarrier放行条件 = 线程数
3. Countdownlatch放行后的动作实施者是第三者线程组，且具有不可重复性。CyclicBarrier放行后动作实施者还是这个线程组本身，且可以反复执行。

Semaphore

控制同时访问某个特定资源的线程数量，主要用于流量控制。

构造方法

```
public Semaphore(int permits) {//可同时许可多少个线程来用 sync = new NonfairSync(permits); }
```

- void acquire() 拿许可
- void release() 还许可
- boolean tryAcquire() 尝试的去拿许可
- int availablePermits() 查询当前还有多少许可
- boolean hasQueuedThreads() 是否有线程在等待许可
- int getQueueLength() 查询当前在等待许可的线程数量

我们接下来用Semaphore，简单实现一个数据库连接池。

首先先写一个数据库连接类来实现Connection接口

```
public class SqlConnectionImpl implements Connection
```

接下来，我们来限制一下数据库连接池的连接数量

```
public class DBPoolSemaphore {

    private final static int POOL_SIZE = 10;//数据库连接池最大连接数量
    private final Semaphore useful, useless;//useful表示可用的数据库连接，useless表示已用的数库连接

    public DBPoolSemaphore() {
```

```
        this.useful = new Semaphore(POOL_SIZE);
        this.useless = new Semaphore(0);
    }

    //存放数据库连接的容器
    private static LinkedList<Connection> pool = new LinkedList<Connection>();

    //初始化池
    static {
        for (int i = 0; i < POOL_SIZE; i++) {
            pool.addLast(SqlConnectImpl.fetchConnection());
        }
    }

    /*归还连接*/
    public void returnConnect(Connection connection) throws InterruptedException {
        if (connection != null) {
            System.out.println("当前有" + useful.getQueueLength() + "个线程等待数据库连接！！"
                    + "可用连接数:" + useful.availablePermits());
            useless.acquire();//已用数据库连接-1
            synchronized (pool) {
                pool.addLast(connection);
            }
            useful.release();//可用数据库连接+1
        }
    }

    /*从池子拿连接*/
    public Connection takeConnect() throws InterruptedException {
        useful.acquire();//可用数据库连接-1
        Connection conn;
        synchronized (pool) {
            conn = pool.removeFirst();
        }
        useless.release();//已用数据库连接+1
        return conn;
    }
}
```

测试类

```
public class AppTest {

    private static DBPoolSemaphore dbPool = new DBPoolSemaphore();

    //业务线程
    private static class BusiThread extends Thread{
        @Override
        public void run() {
            Random r = new Random();//让每个线程持有连接的时间不一样
            long start = System.currentTimeMillis();
            try {
                Connection connect = dbPool.takeConnect();
                System.out.println("Thread_"+Thread.currentThread().getId()
```

```
                +"_获取数据库连接共耗时【"+(System.currentTimeMillis()-start)+"】ms.");
            SleepTools.ms(100+r.nextInt(100));//模拟业务操作，线程持有连接查询数据
            System.out.println("查询数据完成，归还连接！");
            dbPool.returnConnect(connect);
        } catch (InterruptedException e) {
        }
    }
}

public static void main(String[] args) {
    for (int i = 0; i < 30; i++) {
        Thread thread = new BusiThread();
        thread.start();
    }
}
}
```

输出如下

```
Thread_13_获取数据库连接共耗时【1】ms.
Thread_15_获取数据库连接共耗时【0】ms.
Thread_14_获取数据库连接共耗时【0】ms.
Thread_18_获取数据库连接共耗时【1】ms.
Thread_19_获取数据库连接共耗时【0】ms.
Thread_17_获取数据库连接共耗时【0】ms.
Thread_12_获取数据库连接共耗时【3】ms.
Thread_16_获取数据库连接共耗时【1】ms.
Thread_22_获取数据库连接共耗时【0】ms.
Thread_21_获取数据库连接共耗时【0】ms.
查询数据完成，归还连接！
当前有20个线程等待数据库连接！！可用连接数:0//当连接池为空时，需要获取连接的线程就会进入
个等待队列中等待，直到有连接释放掉
Thread_24_获取数据库连接共耗时【105】ms.
查询数据完成，归还连接！
当前有19个线程等待数据库连接！！可用连接数:0
Thread_25_获取数据库连接共耗时【132】ms.
查询数据完成，归还连接！
查询数据完成，归还连接！
当前有18个线程等待数据库连接！！可用连接数:0
Thread_23_获取数据库连接共耗时【164】ms.
查询数据完成，归还连接！
当前有17个线程等待数据库连接！！可用连接数:0
Thread_26_获取数据库连接共耗时【164】ms.
```

Exchanger <V>

Exchanger适用于俩个线程间的数据交换。

俩个线程通过调用exchange()方法交换数据，如果第一个线程先到达同步点（exchange()），它则一直等待第二个线程，直到第二个线程到达同步点，然后俩个线程交换数据后，继续执行任务。

- V exchange(V v)：等待另一个线程到达此交换点（除非当前线程被中断），然后将给定的对象传给该线程，并接收该线程的对象
- V exchange(V v, long timeout, TimeUnit unit)：等待另一个线程到达此交换点（除非当前线程

中断或超出了指定的等待时间），然后将给定的对象传送给该线程，并接收该线程的对象。

```
    private static class ExchangerA extends Thread{
        private Exchanger<String> message = new Exchanger<>();

        public ExchangerA(Exchanger<String> exchanger){
            this.message = exchanger;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(1000);
                System.out.println("等待接收ExchangerB的消息");
                String exchange = message.exchange("Hello ,I am " + Thread.currentThread().getN
me());
                System.out.println("ExchangerB 发送的消息为：/n" + exchange);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private static class ExchangerB extends Thread{
        private Exchanger<String> message = new Exchanger<>();

        public ExchangerB(Exchanger<String> exchanger){
            this.message = exchanger;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(1000);
                System.out.println("等待接收ExchangerA的消息");
                String exchange = message.exchange("你好啊 ,I am " + Thread.currentThread().get
ame());
                System.out.println("ExchangerA 发送的消息为：/n" + exchange);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        Exchanger<String> exchanger = new Exchanger<>();//俩个线程传入同一个Exchanger
        new ExchangerA(exchanger).start();
        new ExchangerB(exchanger).start();
    }
```

再谈Callable、Future和FutureTask

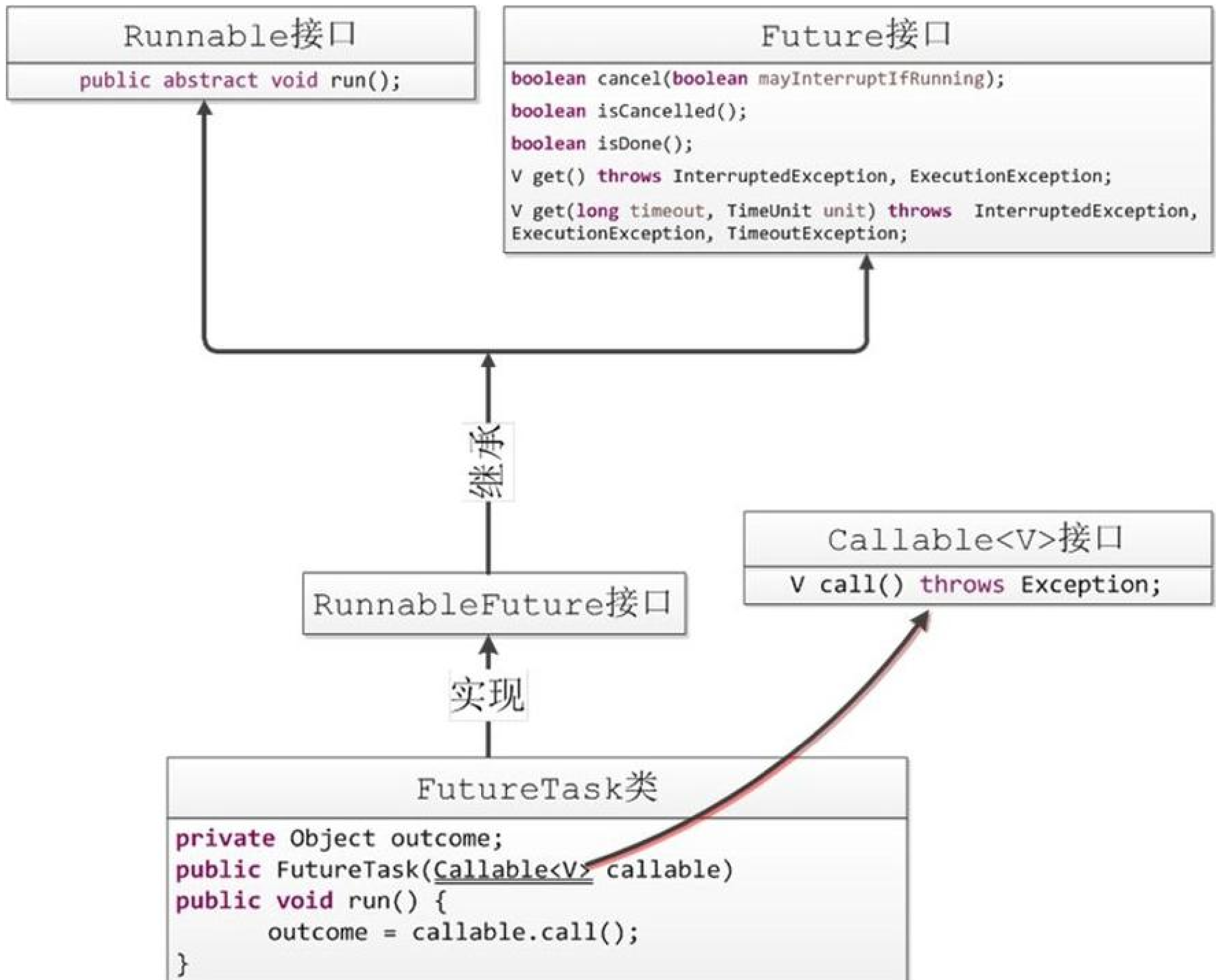
callable接口使用步骤：

1. 创建callable接口实现类，并实现call()方法，该call方法将作为线程执行体，并且有返回值。

2. 创建callable实现类的实例，使用FutureTask类来包装callable对象，该FutureTask对象封装了callable对象的call方法的返回值。

3. 使用FutureTask对象作为Thread对象的target创建并启动新线程。

4. 调用FutureTask对象的get()方法来获取子线程执行结束后的返回值。



由图，我们可以看到，FutureTask实现了继承自Runnable接口和Future接口的RunnableFuture接。FutureTask类的构造函数需要传入一个Callable类型，

下面我们来先讲解一下Future接口的这些方法。

- Boolean isDone() 任务是否执行结束，执行结束 return true； 不管是正常还是异常结束或者自己消只要执行结束都return true；
- Boolean isCancelled() 任务完成前被取消 return true； 其它情况return false；
- Boolean cancel(Boolean)

1. 任务还没有开始，return true；
2. 任务已经开始的话，调用cancel(true)则中断正在运行的任务，中断成功return true； 失败 return alse；
3. 任务已经开始的话，调用cancel(false)不会去中断已经执行的任务,但可返回true

原文链接：Fork/Join 框架 & CountDownLatch 与 CyclicBarrier

4. 任务已经结束，return false；

● get() 获取执行结果，在这个过程中线程会一直阻塞，直到任务执行完毕，如果在此过程中，线程中断则直接抛出异常。

● get(long timeout,TimeUnit unit) get()的超时模式，超时则抛出TimeoutException

接下来我们测试一下这几个方法

```
    /*实现Callable接口，允许有返回值*/
    private static class UseCallable implements Callable<Integer> {
        private int sum;
        @Override
        public Integer call() throws Exception {
            System.out.println("Callable子线程开始计算");
            Thread.sleep(3000);
            for(int i=0;i<5000;i++) {
                sum = sum+i;
            }
            System.out.println("Callable子线程计算完成，结果="+sum);
            return sum;
        }

    }

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

        UseCallable useCallable = new UseCallable();
        FutureTask<Integer> futureTask = new FutureTask<Integer>(useCallable);
        new Thread(futureTask).start();
        Thread.sleep(1000);
        System.out.println("中断计算");
//        boolean cancel = futureTask.cancel(true);
//        boolean cancel = futureTask.cancel(false);//虽然调用cancel但不会中断，响应的isCancelle也是 return true的，这里大家可以测试一下
//        System.out.println("线程中断结果为"+cancel);
        System.out.println("线程是否完成前被取消" + futureTask.isCancelled());
        System.out.println("等待线程运行结果。。。get阻塞中。。。");
        System.out.println("线程运行结果："+ futureTask.get());
    }
```

原文链接：Fork/Join 框架 & CountDownLatch 与 CyclicBarrier