

笔记本：架构师之路
创建时间：2018\11\16 星期五 14:10
作者：804790605@qq.com
URL：about:blank

更新时间：2018\11\23 星期五 18:13

引言

在看分布式架构思想汇总的时候，碰到forkjoin的描述，百度搜索了一下，发现是一个很有趣的东西，感觉可以使用在优化首页信息查询里面，所以研究了一下。

文中描述截图如下：

计算分拆

计算的分拆有2种思路：

数据分拆：一个大的数据集，拆分成多个小的数据集，并行计算。

比如大规模数据归并排序

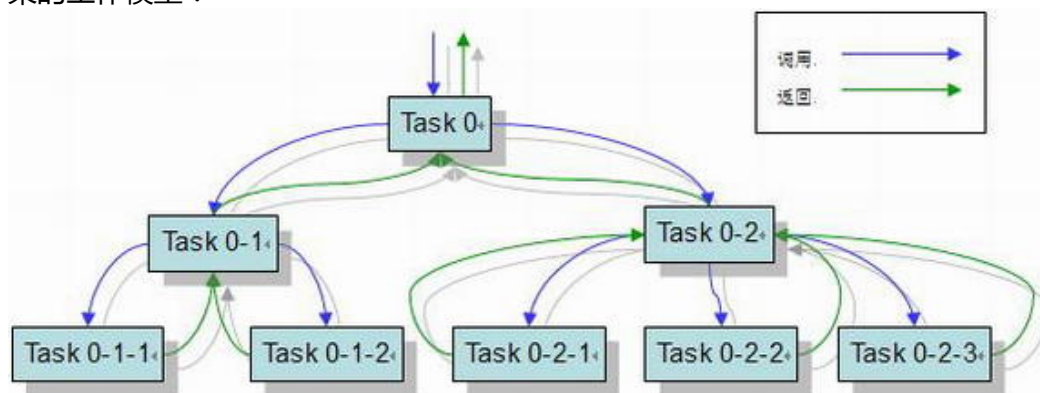
任务分拆：把一个长的任务，拆分成几个环节，各个环节并行计算。

Java中多线程的Fork/Join框架，Hadoop中的Map/Reduce，都是计算分拆的典型框架。其思路都是相似的，先分拆计算，再合并结果。

再比如分布式的搜索引擎中，数据分拆，分别建索引，查询结果再合并。

一、概述

Fork/Join 框架是java7中加入的一个并行任务框架，可以将任务分割成足够小的小任务，然后让不同的线程来做这些分割出来的小事情，然后完成之后再join，将小任务的结果组装成大任务的结果。下面的图片展示了这种框架的工作模型：



Fork/Join工作模型

使用Fork/Join并行框架的前提是我们的任务可以拆分成足够小的任务，而且可以根据小任务的结果来组装出大任务的结果

从JDK1.7开始，Java提供Fork/Join框架用于并行执行任务，它的思想就是讲一个大任务分割成若干小任务，最终汇总每个小任务的结果得到这个大任务的结果。

这种思想和**MapReduce很像 (input --> split --> map --> reduce --> output)**

主要有两步：

- 第一、任务切分；
- 第二、结果合并

它的模型大致是这样的：线程池中的每个线程都有自己的工作队列（PS：这一点和ThreadPoolExecutor不同，ThreadPoolExecutor是所有线程公用一个工作队列，所有线程都从这个工作队列中取任务），当自己队列中的任务都完成以后，会从其它线程的工作队列中偷一个任务执行，这样可以充分利用资源。

ForkJoin框架到了jdk1.8之后进一步做了优化,和jdk1.7的实现方法有很多不同。

ForkJoin框架要点

- fork/join框架是ExecutorService接口的一个实现，可以帮助开发人员充分利用多核处理器的优势，编写出并行执行的程序，提高应用程序的性能；设计的目的是为了处理那些可以被递归拆分的任务。
- fork/join框架与其它ExecutorService的实现类相似，会给线程池中的线程分发任务，不同之处在于它使用了工作窃取算法，所谓工作窃取，指的是对那些处理完自身任务的线程，会从其它线程窃取任务执行。
- fork/join框架的核心是ForkJoinPool类，该类继承了AbstractExecutorService类。ForkJoinPool实现了工作窃取算法并且能够执行 ForkJoinTask任务。

fork/join框架是用多线程的方式实现分治法来解决问题。fork指的是将问题不断地缩小规模，join是指根据子问题的计算结果，得出更高层次的结果。

fork/join框架的使用有一定的**约束条件**：

- 1. 除了fork() 和 join()方法外，线程不得使用其他的同步工具。线程最好也不要sleep()
- 2. 线程不得进行I/O操作
- 3. 线程不得抛出checked exception

- 想要了解核心的设计思路,可以提前阅读**Doug Lea**的论文:
 - [《A Java Fork/Join Framework》](https://link.jianshu.com/?t=http://gee.cs.oswego.edu/dl/papers/fj.pdf) <https://link.jianshu.com/?t=http://gee.cs.oswego.edu/dl/papers/fj.pdf>
 - [Java 并发编程笔记：如何使用 ForkJoinPool 以及原理](https://link.jianshu.com/?t=http://blog.dyngn.com/blog/2016/09/15/java-forkjoinpool-internals) <https://link.jianshu.com/?t=http://blog.dyngn.com/blog/2016/09/15/java-forkjoinpool-internals>

二、设计

在聊ForkJoin框架前，觉得很有必要先熟悉一下线程池。

1、线程池

什么是线程池

a、 第四种获取线程的方法：线程池，一个 ExecutorService，它使用可能的几个池线程之一执行每个提交的任务，通常使用 Executors 工厂方法配置。

b、 线程池可以解决两个不同问题：由于减少了每个任务调用的开销，它们通常可以在执行大量异步任务时提供增强的性能，并且还可以提供绑定和管理资源（包括执行任务集时使用的线程）的方法。每个ThreadPoolExecutor还维护着一些基本的统计数据，如完成的任务数。

c、 为了便于跨大量上下文使用，此类提供了很多可调整的参数和扩展钩子 (hook)。但是，强烈建议程序员使用较为方便的 Executors 工厂方法

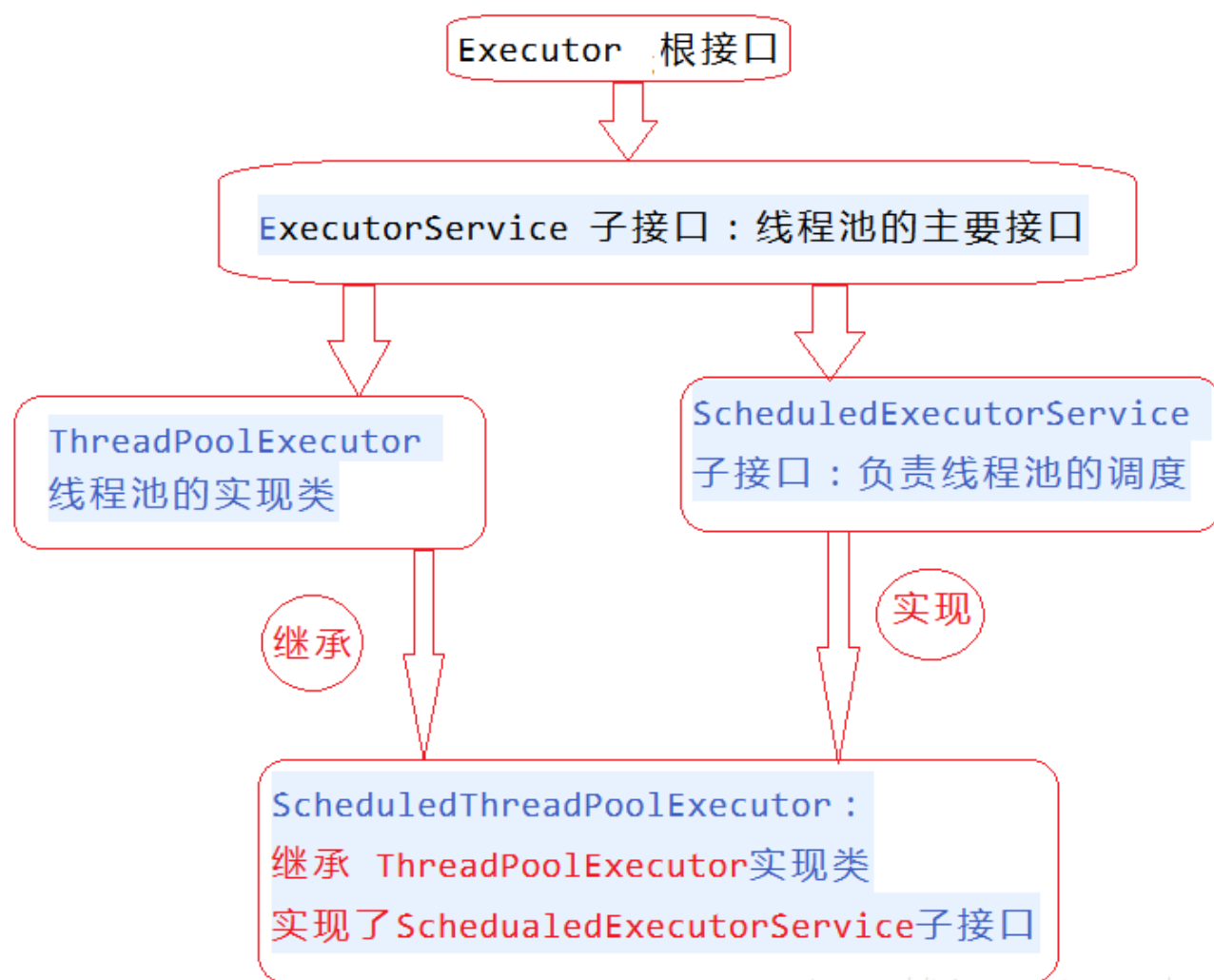
- Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）
- Executors.newFixedThreadPool(int)（固定大小线程池）
- Executors.newSingleThreadExecutor()（单个后台线程）

它们均为大多数使用场景预定义了设置。

线程池的体系结构：

java.util.concurrent.Executor：负责线程的使用与调度的根接口

- |->ExecutorService 子接口：线程池的主要接口
- |->ThreadPoolExecutor：线程池的实现类
- |->ScheduledExecutorService 子接口：负责线程池的调度
 - |->ScheduledThreadPoolExecutor：继承 ThreadPoolExecutor实现类，实现了 ScheduledExecutorService子接口



http://blog.csdn.net/rocky_03

工具类：java.util.concurrent.Executors

- `ExecutorService newFixedThreadPool()`：创建固定大小的线程池
- `ExecutorService newCachedThreadPool()`：缓存线程池，线程池的数量数量不固定，根据自己的需要更改大小
- `ExecutorService newSingleThreadExecutor()`：创建单个线程池，线程池中只有一个线程
- `ScheduledExecutorService newScheduledThreadPool()`：创建固定大小的线程，可以延迟或定时的执行任务

示例代码如下：

```
package main.zf.forkjoin;
```

```
/**
```

```

* @Author: Administrator
* @Date: 2018\11\16 0016 17:39
* @Description: 测试的继承Runnable接口的类
*/
class RunDemo implements Runnable{
@Override
public void run() {
for(int i=0;i<5;i++){
System.out.println(Thread.currentThread().getName()+"==>" +i);
}
}
}
}

```

```

package main.zf.forkjoin;

```

```

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

```

```

/**
* @Author: Administrator
* @Date: 2018\11\16 0016 17:41
* @Description:
*/

```

```

public class ThreadTest {
public static void main(String [] args){
test3();
}

```

```

public static void test1(){
ExecutorService pool = Executors.newFixedThreadPool(3);//创建指定三个线程的线程池
for(int i=0;i<3;i++){
pool.submit(new RunDemo());
}
pool.shutdown();//当前任务执行完之后关闭
// pool.shutdownNow();//强制关闭
}

```

```

/**
* 带返回值的Callable
*/
public static void test2(){

```

```

ExecutorService pool = Executors.newFixedThreadPool(3);//创建指定三个线程的线程池
//使用callable普通任务,创建匿名内部类,测试submit
List<Future<Integer>> list = new ArrayList<Future<Integer>>();
for (int i = 0; i < 10; i++) {
    Future<Integer> future = pool.submit(new Callable<Integer>() {
        @Override
        public Integer call() throws Exception {
            int sum=0;
            for(int j=0;j<100;j++){
                sum+=j;
            }
            return sum;
        }
    });
    list.add(future);
}
for(Future<Integer> future:list){
    try {
        System.out.println(future.get());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
pool.shutdown();
}

```

*/***

** 带返回值的Callable的定时任务*

**/*

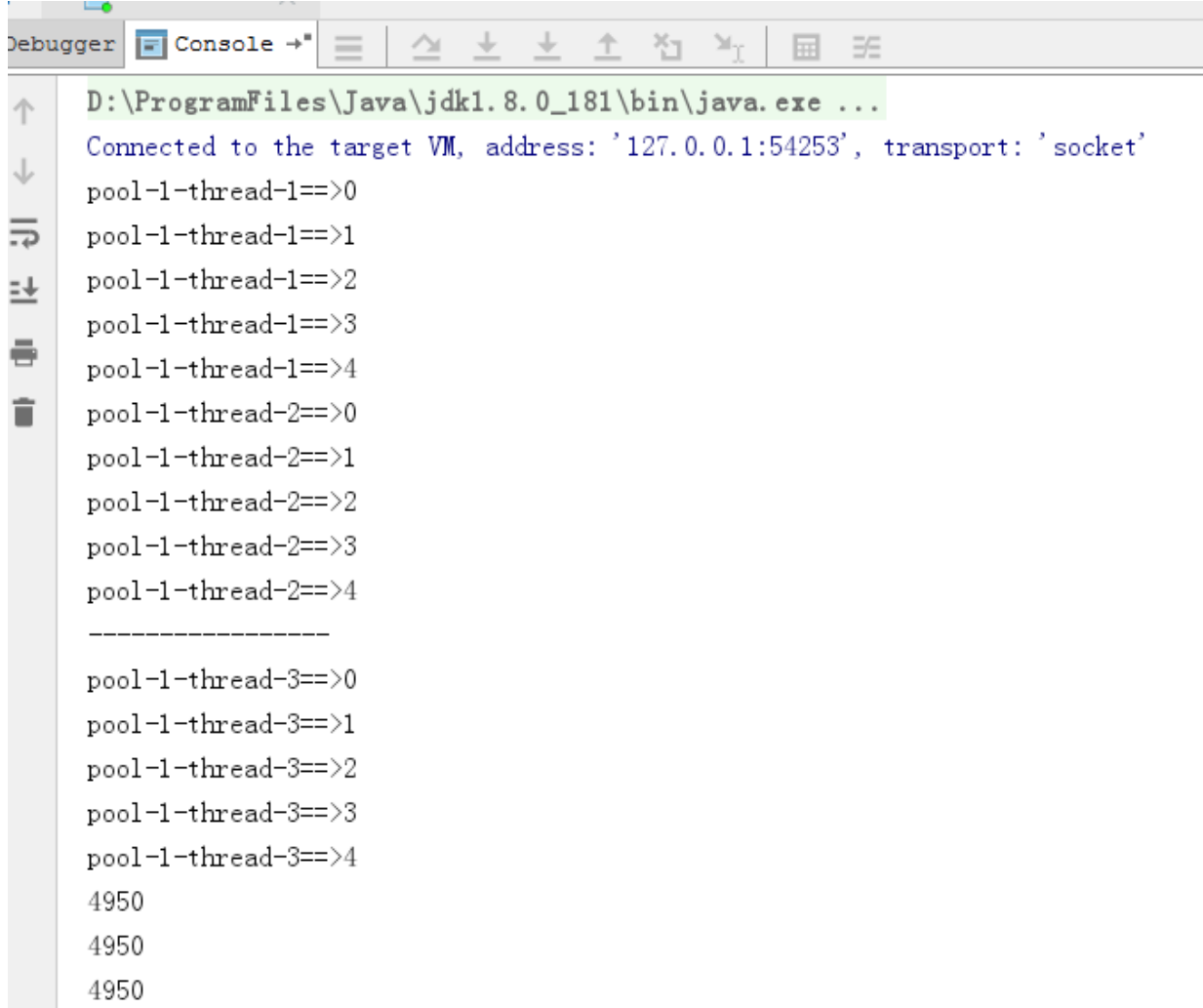
```

public static void test3(){
    ScheduledExecutorService pool2 = Executors.newScheduledThreadPool(3);
    //使用callable定时任务
    for(int i=0;i<5;i++){
        Future<Integer> result = pool2.schedule(new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                int num = new Random().nextInt(100);//100以内的随机数
                return num;
            }
        }, 1, TimeUnit.SECONDS);
        try {
            System.out.println(result.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
    pool2.shutdown();
}

```

```
}  
}  
}
```

对应的结果如下：



The screenshot shows a Java IDE's console window. The title bar includes 'Debugger' and 'Console'. The console output displays the following sequence of messages:

```
D:\ProgramFiles\Java\jdk1.8.0_181\bin\java.exe ...  
Connected to the target VM, address: '127.0.0.1:54253', transport: 'socket'  
pool-1-thread-1==>0  
pool-1-thread-1==>1  
pool-1-thread-1==>2  
pool-1-thread-1==>3  
pool-1-thread-1==>4  
pool-1-thread-2==>0  
pool-1-thread-2==>1  
pool-1-thread-2==>2  
pool-1-thread-2==>3  
pool-1-thread-2==>4  
-----  
pool-1-thread-3==>0  
pool-1-thread-3==>1  
pool-1-thread-3==>2  
pool-1-thread-3==>3  
pool-1-thread-3==>4  
4950  
4950  
4950
```

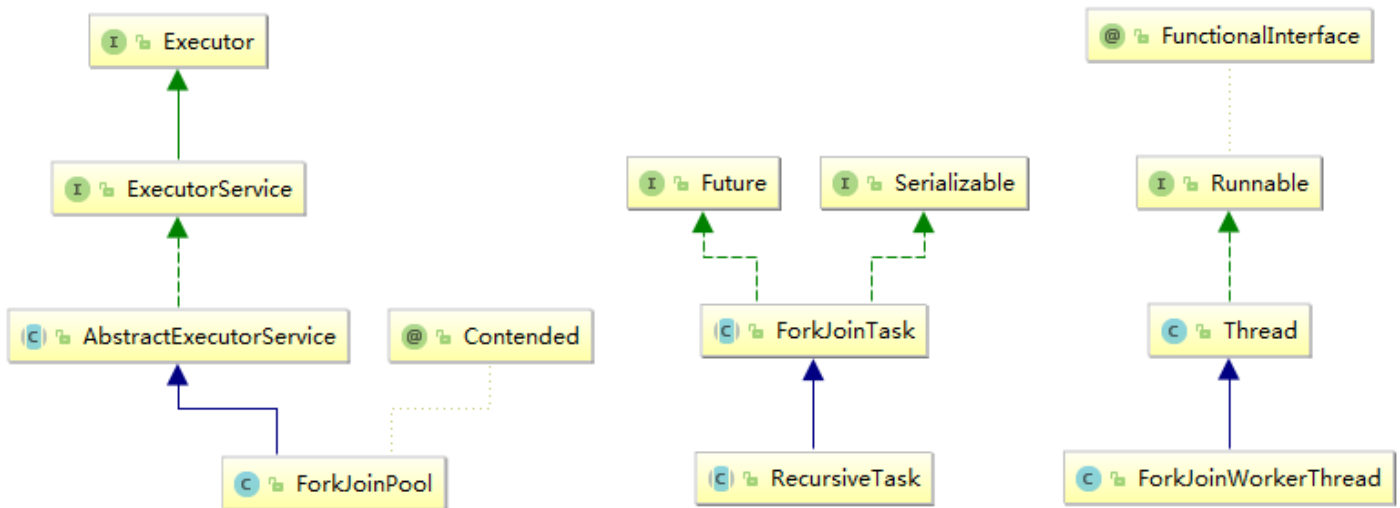
```
Debugger Console
↑
↓
↺
↻
pool-1-thread-3==>4
4950
4950
4950
4950
4950
4950
4950
4950
4950
4950
-----
53
3
25
5
89
Disconnected from the target VM, address: '127.0.0.1:54253', transport: 'socket'
Process finished with exit code 0
```

2、ForkJoin框架

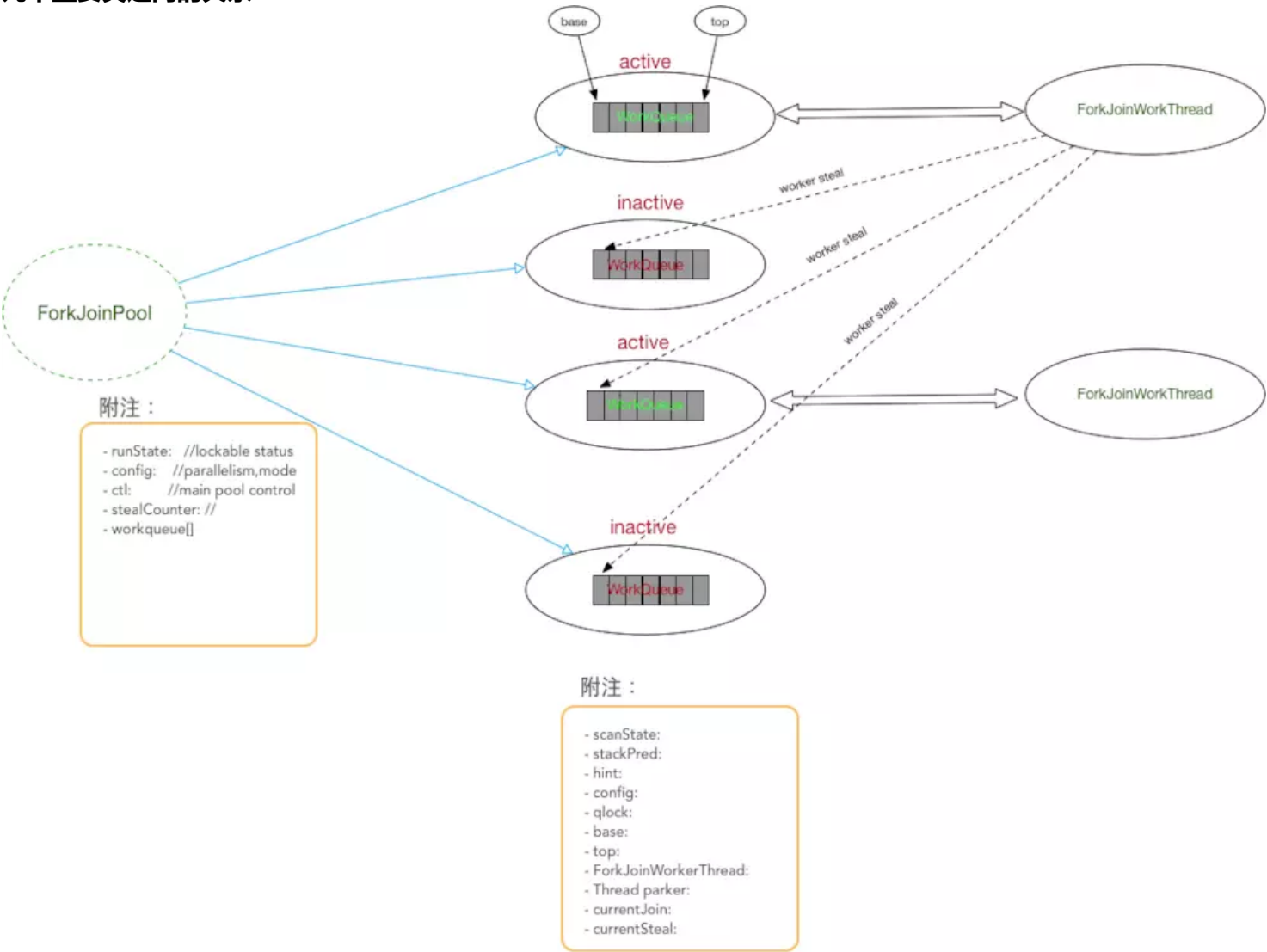
和传统的线程池使用AQS的实现逻辑不同,ForkJoin引入全新的结构来标识:

- ForkJoinPool: 用于执行ForkJoinTask任务的执行池,不再是传统执行池 **Worker+Queue** 的组合模式,而是维护了一个队列数组WorkQueue,这样在提交任务和线程任务的时候大幅度的减少碰撞。
- WorkQueue: 双向列表,用于任务的有序执行,如果WorkQueue用于自己的执行线程Thread,线程默认将会从top端选取任务用来执行 - LIFO。因为只有owner的Thread才能从top端取任务,所以在设置变量时, int top; 不需要使用 volatile。
- ForkJoinWorkThread: 用于执行任务的线程,用于区别使用非ForkJoinWorkThread线程提交的task;启动一个该Thread,会自动注册一个WorkQueue到Pool,这里规定,拥有Thread的WorkQueue只能出现在 **WorkQueue数组的奇数位**
- ForkJoinTask: 任务, 它比传统的任务更加轻量, 不再对是RUNNABLE的子类,提供fork/join方法用于分割任务以及聚合结果。
- 为了充分施展并行运算,该框架实现了复杂的 **worker steal**算法,当任务处于等待中,thread通过一定策略,不让自己挂起,充分利用资源,当然,它比其他语言的协程要重一些。
- ForkJoin采用 **“工作窃取” 模式 (work-stealing)** : 当执行新的任务时它可以将其拆分分成更小的任务执行,并将小任务加到线程队列中,然后再从一个随机线程的队列中偷一个并把它放在自己的队列中。
- ForkJoin相对于一般的线程池实现, fork/join框架的优势体现在对其中包含的任务的处理方式上.在一般的线程池中, 如果一个线程正在执行的任务由于某些原因无法继续运行, 那么该线程会处于等待状态.而在 fork/join框架实现中,如果某个子问题由于等待另外一个子问题的完成而无法继续运行. 那么处理该子问题的线程会主动寻找其他尚未运行的子问题来执行.这种方式**减少了线程的等待时间**,提高了性能。

几个重要类的类图



几个重要类之间的关系：



废话不多说，先来个实例。代码如下：

```

package main.zf.forkjoin;

import java.util.concurrent.RecursiveTask;

/**

```



```
* @Author: Administrator
* @Date: 2018\11\19 0019 09:37
* @Description:
*/
```

```
class CaculatorForkAndJoin extends RecursiveTask<Long> {
```

```
/**
 * 创建serialVersionUID
 */
```

```
private static final long serialVersionUID = 1L;
```

```
private long start;
private long end;
private static final long THURSHOLD = 10000L; //临界值
```

```
CaculatorForkAndJoin(long start,long end){
    this.start = start;
    this.end = end;
}
```

```
//重写方法
```

```
@Override
```

```
protected Long compute() {
    long length = end - start;
    if(length <= THURSHOLD){
        long sum = new Long(0);
        for(long i = start;i<=end;i++){
            sum+=i;
        }
    }
}
```

```
return sum;
```

```
}else{
```

```
//中间值
```

```
long mid = (start + end ) / 2;
```

```
CaculatorForkAndJoin left = new CaculatorForkAndJoin(start, mid);
```

```
left.fork();//进行拆分，同时压入现线程队列
```

```
CaculatorForkAndJoin right = new CaculatorForkAndJoin(mid+1, end);
```

```
right.fork();//进行拆分，同时压入现线程队列
```

```
return left.join()+right.join();
```

```
}
```

```
}
```

```
}
```

```
package main.zf.forkjoin;
```

```
import java.util.concurrent.ForkJoinPool;
```

```
/**
```

```
* @Author: Administrator
```

```
* @Date: 2018\11\19 0019 09:38
```

* @Description:

*/

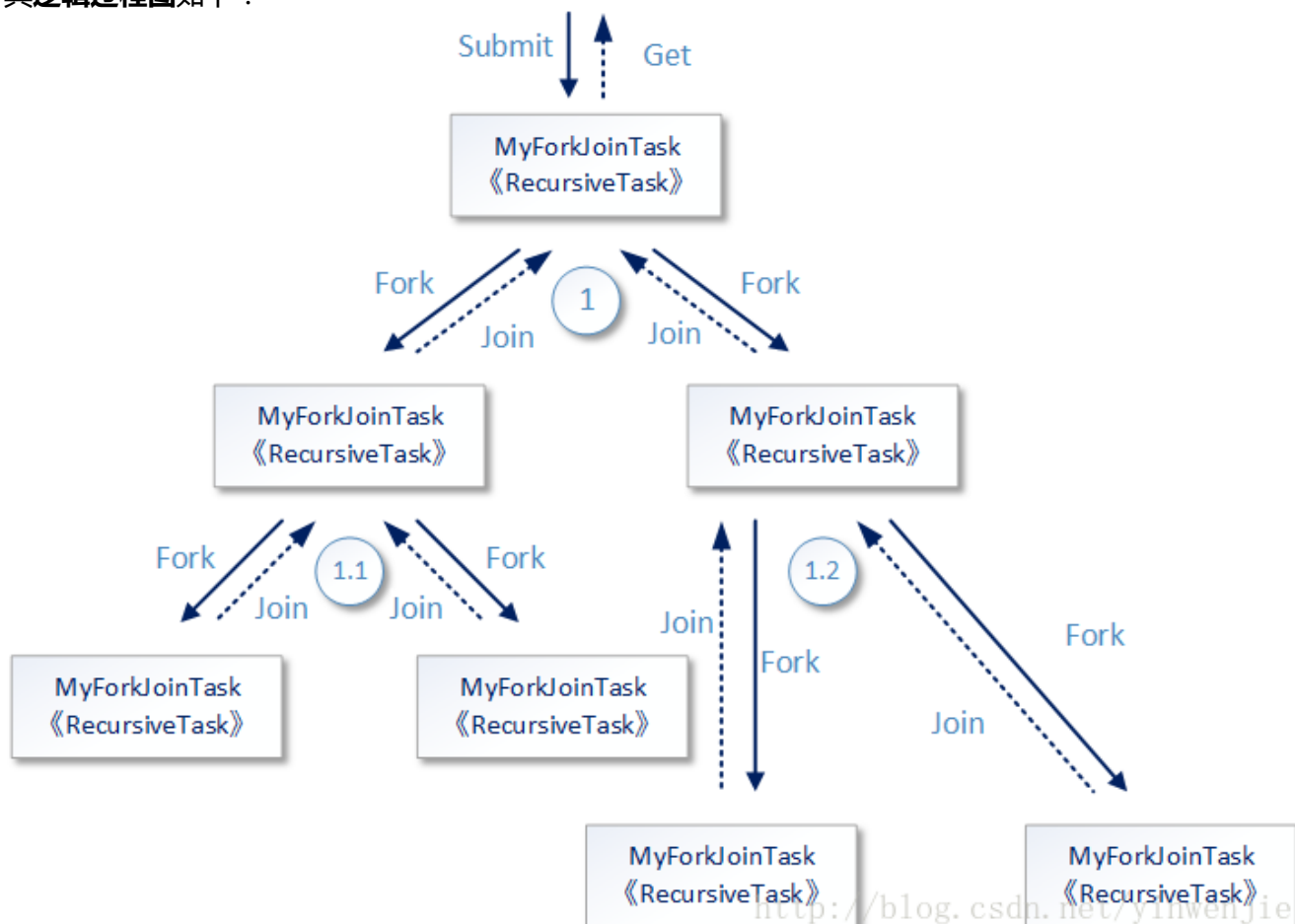
```
public class ForkJoinTest {  
    public static void main(String[] args) {  
        long start = System.currentTimeMillis();  
        //创建 线程池  
        ForkJoinPool pool = new ForkJoinPool();  
        //创建任务  
        CaculatorForkAndJoin task = new CaculatorForkAndJoin(0L,1000000000L);  
        //添加任务到线程池，获得返回值  
        long sum = pool.invoke(task);  
        long end = System.currentTimeMillis();  
        System.out.println(sum+"spend:"+(end - start));  
    }  
}
```

运行结果如下：

```
D:\ProgramFiles\Java\jdk1.8.0_181\bin\java.exe ...  
Connected to the target VM, address: '127.0.0.1:51798', transport: 'socket'  
500000000500000000spend:50  
Disconnected from the target VM, address: '127.0.0.1:51798', transport: 'socket'
```

会发现，运行所用的时间非常短。

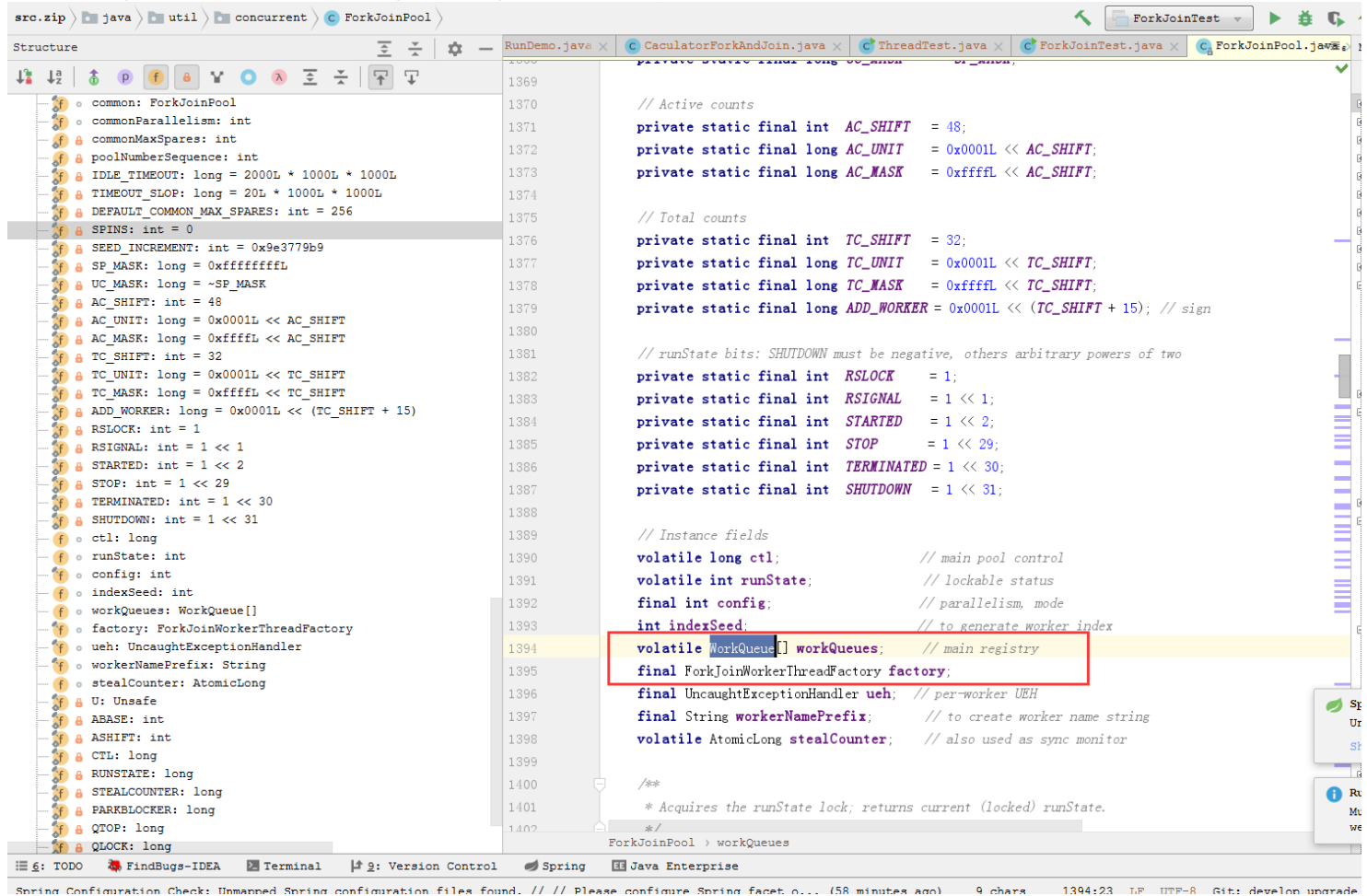
其逻辑过程图如下：



源码解析

通过ForkJoinPool的类关系图，可以看到本质上它就是一个Executor。

通过源码，在ForkJoinPool里面，有两个特别重要的成员如下：



可以看到红框里面的两个变量，

- **workQueues** 用于保存向ForkJoinPool提交的任务，而具体的执行有ForkJoinWorkerThread执行。
- **ForkJoinWorkerThreadFactory**可以用于生产出ForkJoinWorkerThread。可以看一下ForkJoinWorkerThread，可以发现每一个ForkJoinWorkerThread会有一个pool和一个workQueue，和我们上面描述的是一致的，每个线程都被分配了一个任务队列，而执行这个任务队列的线程由pool提供。

```
protected ForkJoinWorkerThread(ForkJoinPool pool) {
    // Use a placeholder until a useful name can be set in registerWorker
    super( name: "aForkJoinWorkerThread");
    this.pool = pool;
    this.workQueue = pool.registerWorker( wt: this);
}
```

下面看下关键的fork和join方法

fork

源码如下：

```

/**
 * Arranges to asynchronously execute this task in the pool the
 * current task is running in, if applicable, or using the {@link
 * ForkJoinPool#commonPool()} if not {@link #inForkJoinPool}. While
 * it is not necessarily enforced, it is a usage error to fork a
 * task more than once unless it has completed and been
 * reinitialized. Subsequent modifications to the state of this
 * task or any data it operates on are not necessarily
 * consistently observable by any thread other than the one
 * executing it unless preceded by a call to {@link #join} or
 * related methods, or a call to {@link #isDone} returning {@code
 * true}.
 *
 * @return {@code this}, to simplify usage
 */

```

```

@ public final ForkJoinTask<V> fork() {
    Thread t;
    if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)
        ((ForkJoinWorkerThread)t).workQueue.push(task: this);
    else
        ForkJoinPool.common.externalPush(task: this);
    return this;
}

```

看上面的fork代码，可以看到

- 首先**取**到了当前线程，
- 然后**判断**是否是我们的ForkJoinPool专用线程，
 - 如果是，则强制类型转换（**向下转换**）成ForkJoinWorkerThread，然后将任务**push**到这个线程负责的队列里面去。
 - 如果当前线程不是ForkJoinWorkerThread类型的线程，那么就会走else之后的逻辑，
 - 大概的意思是首先尝试将任务提交给当前线程，如果不成功，则使用例外的处理方法，关于底层实现较为复杂，和我们使用Fork/Join关系也不太大，如果希望搞明白具体原理，可以看源码。

接下来看下**join**

```

/**
 * Returns the result of the computation when it {@link #isDone} is
 * done}. This method differs from {@link #get\(\)} in that
 * abnormal completion results in {@code RuntimeException} or
 * {@code Error}, not {@code ExecutionException}, and that
 * interrupts of the calling thread do not cause the
 * method to abruptly return by throwing {@code}
 * InterruptedException.
 *
 * @return the computed result
 */

```

```

public final V join() {
    int s;
    if ((s = doJoin() & DONE_MASK) != NORMAL)
        reportException(s);
    return getRawResult();
}

```

```

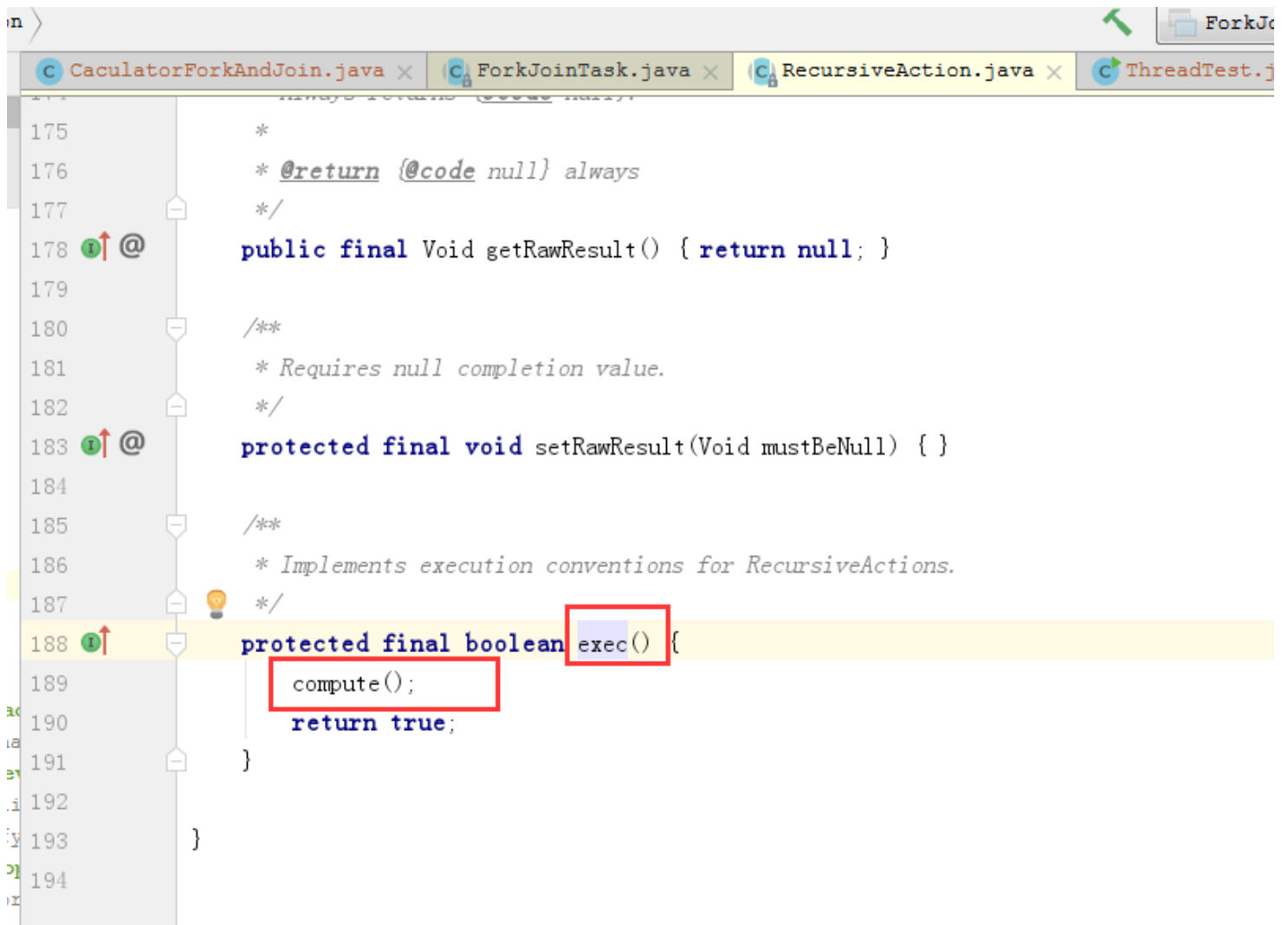
/**
 * Implementation for join, get, quietlyJoin. Directly handles
 * only cases of already-completed, external wait, and
 * unfork+exec. Others are relayed to ForkJoinPool.awaitJoin.
 *
 * @return status upon completion
 */

```

```

private int doJoin() {
    int s; Thread t; ForkJoinWorkerThread wt; ForkJoinPool.WorkQueue w;
    return (s = status) < 0 ? s :
        ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) ?
            (w = (wt = (ForkJoinWorkerThread)t).workQueue).
                tryUnpush(t: this) && (s = doExec()) < 0 ? s :
            wt.pool.awaitJoin(w, task: this, deadline: 0L) :
        externalAwaitDone();
}

```



```
175      *
176      * @return {@code null} always
177      */
178      @ public final Void getRawResult() { return null; }
179
180      /**
181       * Requires null completion value.
182       */
183      @ protected final void setRawResult(Void mustBeNull) { }
184
185      /**
186       * Implements execution conventions for RecursiveActions.
187       */
188      protected final boolean exec() {
189          compute();
190          return true;
191      }
192
193  }
194
195  }
```

上面展示了主要的调用链路，我们发现最后落到了我们在代码里编写的`compute`方法（其他要么是`run`或是`call`之类的线程方法），也就是执行它。

所以，我们需要知道的一点是，`fork`仅仅是分割任务，只有当我们执行`join`的时候，我们的任务才会被执行。

应用

刚好要在项目中统计他人发起的签约合同数量，觉得可以使用上，接下来就把代码逻辑贴出来，如下：

根据当前用户获取其所有接入者，查询在每个接入者下签约的数量（这个可以作为原子操作，即可以`fork`，`join`操作），当然由于统计的东西比较多，我把获取每一个接入者的操作也作为原子操作（里面每一个统计都可以作为`fork`，`join`）。

接下来看代码如下：

1、处理URI的controller

```
package com.ancun.netsign.manage.controller;

/**
 * @Author: zhengfeng
 * @Date: 2018\11\20 0020 20:18
 * @Description:
 */
@RequestMapping("/statistical")
@RestController
public class StatisticalController {
    private static final Logger logger = LoggerFactory.getLogger(StatisticalController.class);
```

@Autowired

private StatisticalService **statisticalService**;

@Autowired

private PartnerService **partnerService**;

/**

* 运营支撑平台登录后可统计相关接入者使用数据：

* 合同发起量、

* 完成签署量（我发起的+他人发起的）、

* 二要素、三要素、四要素、

* 人脸认证使用数

* @param **statisticRequest**

* @return

*/

@RequestMapping(value = **"/statisCounts"**, method = RequestMethod.**POST**)

@RequiresPermissions(**"statistical:statisCounts"**)

public RespBody<PageInfo<StatisticResponseVO>> statisCount(StatisticRequest statisticRequest) {

List list = **new** ArrayList<>();

PageInfo<StatisticResponseVO> resultList = **new** PageInfo<>();

if(**null** != statisticRequest.getPartnerId()){

StatisticResponseVO statisticResponse = **new** StatisticResponseVO();

PartnerPO partnerPO = **partnerService**.findOneById(statisticRequest.getPartnerId());

statisticResponse.setPartnerName(partnerPO.getPartnerName());

statisticResponse.setPartnerCreateTime(**new** Timestamp(partnerPO.getCreateTime().getTime()));

// 获取当前接入者发起的合同数量

Integer applyCount = **statisticalService**.getPartnerApplyContractCount(statisticRequest);

Long signedCount = **statisticalService**.getPartnerSignCount(statisticRequest);

Integer personalIdentifyTwoCount =

statisticalService.getPartnerPersonIdentifyCountTwo(statisticRequest);

Integer personalIdentifyThreeCount =

statisticalService.getPartnerPersonIdentifyCountThree(statisticRequest);

Integer personalIdentifyFourCount =

statisticalService.getPartnerPersonIdentifyCountFour(statisticRequest);

Long faceCount = **statisticalService**.getPartnerFaceIdentifyCount(statisticRequest);

statisticResponse.setApplyCount(applyCount.longValue());

statisticResponse.setSignedCount(signedCount);

statisticResponse.setPersonTwoCount(Long.*valueOf*(personalIdentifyTwoCount));

statisticResponse.setPersonThreeCount(Long.*valueOf*(personalIdentifyThreeCount));

statisticResponse.setPersonFourCount(Long.*valueOf*(personalIdentifyFourCount));

statisticResponse.setFaceCount(faceCount);

list.add(statisticResponse);

resultList = **new** PageInfo<>(list);

else {

// 统计全部接入者

PartnerRequest partnerRequest = **new** PartnerRequest();

if(**null** != statisticRequest){

partnerRequest.setPage(statisticRequest.getPage());

partnerRequest.setRows(statisticRequest.getRows());

}

PageInfo<PartnerPO> pagePartner = **partnerService**.queryPartner(partnerRequest);


```

List<PartnerPO> partnerPOList = pagePartner.getList();
list = statisticalService.getPartnersStatistical(partnerPOList,statisticRequest);
BeanUtils.copyProperties(pagePartner,resultList);
resultList.setList(list);
}
return new RespBody<>(resultList);
}
}

```

2、具体逻辑service层以及操作fork/join操作的RecursiveTask类

```

/**
 * 对传入的接入者列表的每一个接入者进行统计
 * @param partnerList
 * @param statisticRequest
 * @return
 */
@Override
public List<StatisticResponseVO> getPartnersStatistical(List partnerList, StatisticRequest
statisticRequest){
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    return forkJoinPool.invoke(new PartnerListStatisticalTask(partnerList, statisticRequest));
}

/**
 * 单独处理一个接入者的统计情况
 */
private class PartnerStatisticalTask extends RecursiveTask<StatisticResponseVO> {
    private final StatisticRequest statisticRequest;
    PartnerStatisticalTask(StatisticRequest statisticRequest) {
        super();
        this.statisticRequest = statisticRequest;
    }
    @Override
    protected StatisticResponseVO compute() {
        StatisticResponseVO statisticResponse = new StatisticResponseVO();
        PartnerPO partnerPO = partnerService.findOneById(statisticRequest.getPartnerId());
        statisticResponse.setPartnerName(partnerPO.getPartnerName());
        statisticResponse.setPartnerCreateTime(partnerPO.getCreateTime());
        // 获取当前接入者发起的合同数量
        Integer applyCount = getPartnerApplyContractCount(statisticRequest);
        Long signedCount = getPartnerSignCount(statisticRequest);
        Integer personalIdentifyTwoCount = getPartnerPersonIdentifyCountTwo(statisticRequest);
        Integer personalIdentifyThreeCount = getPartnerPersonIdentifyCountThree(statisticRequest);
        Integer personalIdentifyFourCount = getPartnerPersonIdentifyCountFour(statisticRequest);
        Long faceCount = getPartnerFaceIdentifyCount(statisticRequest);
        statisticResponse.setApplyCount(applyCount.longValue());
        statisticResponse.setSignedCount(signedCount);
        statisticResponse.setPersonTwoCount(Long.valueOf(personalIdentifyTwoCount));
        statisticResponse.setPersonThreeCount(Long.valueOf(personalIdentifyThreeCount));
    }
}

```

```

    statisticResponse.setPersonFourCount(Long.valueOf(personalIdentifyFourCount));
    statisticResponse.setFaceCount(faceCount);
    return statisticResponse;
}
}

/**
 * 接入者列表的处理
 */
private class PartnerListStatisticalTask extends RecursiveTask<List<StatisticResponseVO>> {
    private final List<PartnerPO> partnerList;
    private final StatisticRequest statisticRequest;
    PartnerListStatisticalTask(List<PartnerPO> partnerList, StatisticRequest statisticRequest) {
        super();
        this.partnerList = partnerList;
        this.statisticRequest = statisticRequest;
    }
    @Override
    protected List<StatisticResponseVO> compute() {
        List<StatisticResponseVO> list = new ArrayList<>();
        List<RecursiveTask<StatisticResponseVO>> forks = new LinkedList<>();
        for (PartnerPO partner : partnerList) {
            StatisticRequest statisticRequestTemp = new StatisticRequest();
            BeanUtils.copyProperties(statisticRequest, statisticRequestTemp);
            statisticRequestTemp.setPartnerId(partner.getId());
            PartnerStatisticalTask task = new PartnerStatisticalTask(statisticRequestTemp);
            forks.add(task);
            task.fork();
        }
        for (RecursiveTask<StatisticResponseVO> task : forks) {
            list.add(task.join());
        }
        return list;
    }
}

```

3、应用结果如下：

192.168.0.182:7000/statistic/statisCounts

应用 小白一键重装 百度 淘宝 京东 天猫 系统之家 电影大全 intelliJ git ancun 前阵 springboot project tars JPA my 从任务中导入 vue JWT jenkins bigdata pm 缓存 shiro 其他书签

安存 ANCUN 运营支撑管理平台 admin

数据统计 用量统计 / 数据统计

刷新 搜索

接入者: 全部 开始时间: 请选择开始时间 结束时间: 请选择结束时间

序号	接入者	接入时间	发起合同数量	合同签署数	二要素认证次数	三要素认证次数	四要素认证次数	人脸认证次数
1	奥巴马测试3	2018-11-22 21:17:21	4	1	1	0	0	0
2	宁波慈溪农村商业银行股份有限公司	2018-11-22 22:08:45	2	6	1	3	1	0
3	糯米Test	2018-08-29 16:40:59	345	359	4	2	4	19
4	笑歌江湖	2018-11-22 21:17:51	0	0	0	0	0	0
5	奥巴马测试2	2018-11-22 20:46:41	0	0	0	0	0	0
6	mulan	2018-08-29 16:31:56	226	175	1	5	4	12
7	彭清彪号	2018-10-08 18:16:11	96	57	0	0	0	0
8	千禧测试	2018-08-29 16:46:15	143	79	0	0	0	2
9	奥巴马测试	2018-08-29 16:36:54	398	278	8	17	4	6
10	安存	2018-03-20 17:03:00	0	0	0	20	6	0

6 requests | 12.3 KB transferred

Console Filter Default levels Group similar 0.2s 0.1s 69%

Network Headers Preview Response Cookies Timing

Request URL: http://192.168.0.182:10010/statistical/statisCounts

Request Method: POST

Status Code: 200

Remote Address: 192.168.0.182:10010

Referrer Policy: no-referrer-when-downgrade

Response Headers

Access-Control-Allow-Credentials: true

Access-Control-Allow-Methods: GET, POST, OPTIONS, PUT, DELETE

Access-control-Allow-Origin: http://192.168.0.182:7000

Content-Type: application/json; charset=UTF-8

Date: Fri, 23 Nov 2018 10:11:19 GMT

Set-Cookie: rememberMe=deleteMe; Path=/; Max-Age=0; Expires=Thu, -Nov-2018 10:11:20 GMT

Transfer-Encoding: chunked

Request Headers

Accept: application/json, text/plain, */*

参考资料：

分布式架构 - 基本思想汇总 <https://mp.weixin.qq.com/s/ZoSXhTri2unIQ6OQh9JTBg>

Java Fork/Join框架 <http://ifeve.com/java-fork-join-framework/amp/>

Java并发（十）线程池&fork/join框架 https://blog.csdn.net/rocky_03/article/details/71304319

Java线程之fork/join框架 <https://blog.csdn.net/z69183787/article/details/70183540>

Java Fork/Join 框架 <https://www.cnblogs.com/cjsblog/p/9078341.html>

java多线程解说【拾贰】_并发框架：Fork/Join <https://blog.csdn.net/xinzun/article/details/79372795>

JAVA中的Fork/Join框架 <http://www.cnblogs.com/chenpi/p/5581198.html>

JAVA并行框架：Fork/Join <http://www.cnblogs.com/dongguacai/p/6021859.html>

线程基础：多任务处理（12）——Fork/Join框架（基本使用） <https://blog.csdn.net/yinwenjie/article/details/71524140>

jdk1.8-ForkJoin框架剖析 https://www.jianshu.com/p/f777abb7b251?utm_campaign=maleskine&utm_content=note&utm_medium=pc_all_hots&utm_source=recommendation

Java Fork/Join并行框架 <https://www.jianshu.com/p/ac9e175662ca>