<https://mp.weixin.qq.com/s/lKew-ID6zqP_QlN9j6XeHg>

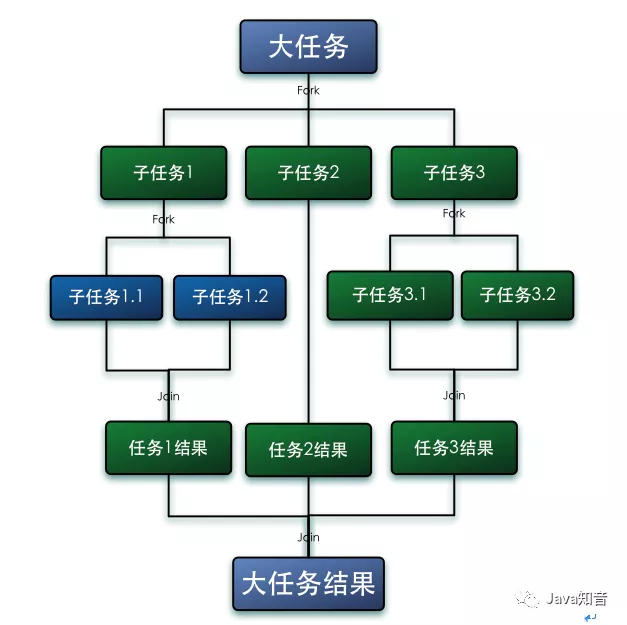
fork/join 全面剖析，你可以不用，但是不能不懂！

fork/join作为一个并发框架在jdk7的时候就加入到了我们的java并发包java.util.concurrent中，并且在java 8 的lambda并行流中充当着底层框架的角色。

这样一个优秀的框架设计，我自己想了解一下它的底层代码是如何实现的，所以我尝试的去阅读了JDK相关的源码。下面我打算分享一下阅读完之后的心得~。

## 1、fork/join的设计思路

了解一个框架的第一件事，就是先了解别人的设计思路!

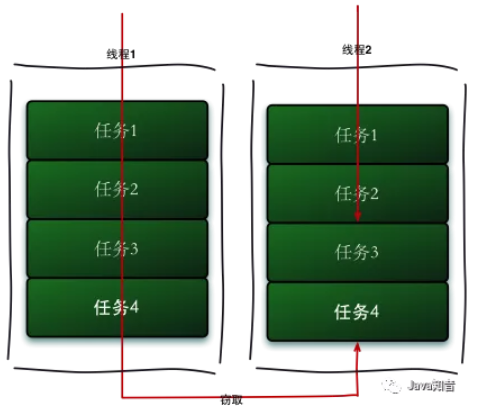


fork/join大体的执行过程就如上图所示，先把一个大任务分解(fork)成许多个独立的小任务，然后起多线程并行去处理这些小任务。处理完得到结果后再进行合并(join)就得到我们的最终结果。

显而易见的**这个框架是借助了现代计算机多核的优势并行去处理数据。**这看起来好像没有什么特别之处，这个套路很多人都会，并且工作中也会经常运用~。其实fork/join的最特别之处在于它还运用了一种叫work-stealing(工作窃取)的算法，这种算法的设计思路在于把分解出来的小任务放在多个双端队列中，而线程在队列的头和尾部都可获取任务。

当有线程把当前负责队列的任务处理完之后，它还可以从那些还没有处理完的队列的尾部窃取任务来处理，这连线程的空余时间也充分利用了！。

work-stealing原理图如下：



## 2、实现fork/join 定义了哪些角色？

了解设计原理，这仅仅是第一步！要了解别人整个的实现思路， 还需要了解别人为了实现这个框架定义了哪些角色，并了解这些角色的职责范围是什么的。因为知道谁负责了什么，谁做什么，这样整个逻辑才能串起来！在JAVA里面角色是以类的形式定义的,而了解类的行为最直接的方式就是看定义的公共方法~。

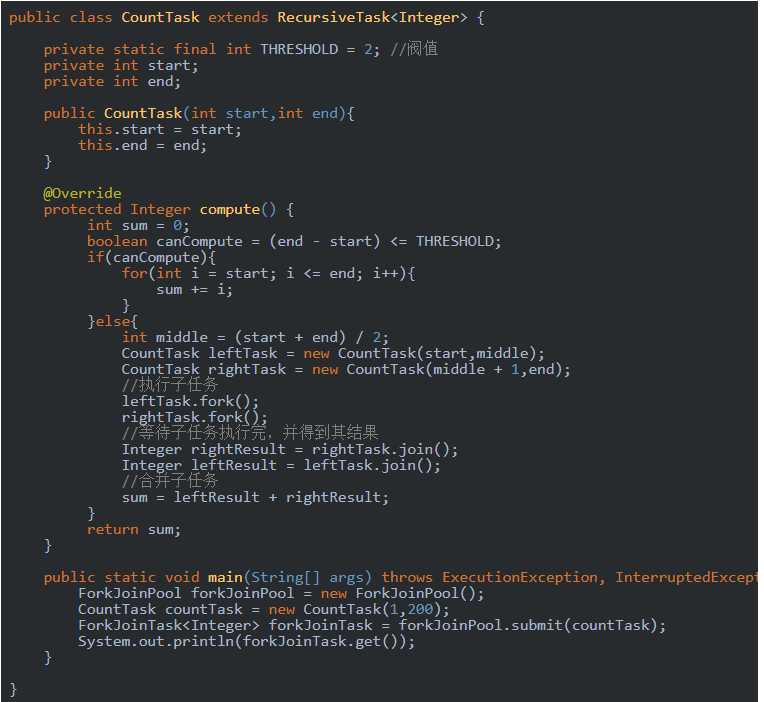
这里介绍JDK里面与fork/join相关的主要几个类：

* **ForkJoinPool：**充当fork/join框架里面的管理者，最原始的任务都要交给它才能处理。它负责控制整个fork/join有多少个workerThread，workerThread的创建，激活都是由它来掌控。它还负责workQueue队列的创建和分配，每当创建一个workerThread，它负责分配相应的workQueue。然后它把接到的活都交给workerThread去处理，它可以说是整个frok/join的容器。
* **ForkJoinWorkerThread：**fork/join里面真正干活的"工人"，本质是一个线程。里面有一个ForkJoinPool.WorkQueue的队列存放着它要干的活，接活之前它要向ForkJoinPool注册(registerWorker)，拿到相应的workQueue。然后就从workQueue里面拿任务出来处理。它是依附于ForkJoinPool而存活，如果ForkJoinPool的销毁了,它也会跟着结束。
* **ForkJoinPool.WorkQueue:**双端队列就是它，它负责存储接收的任务。
* **ForkJoinTask：**代表fork/join里面任务类型，我们一般用它的两个子类RecursiveTask、RecursiveAction。这两个区别在于RecursiveTask任务是有返回值，RecursiveAction没有返回值。任务的处理逻辑包括任务的切分都集中在compute()方法里面。

## 3、fork/join初始化时做了什么

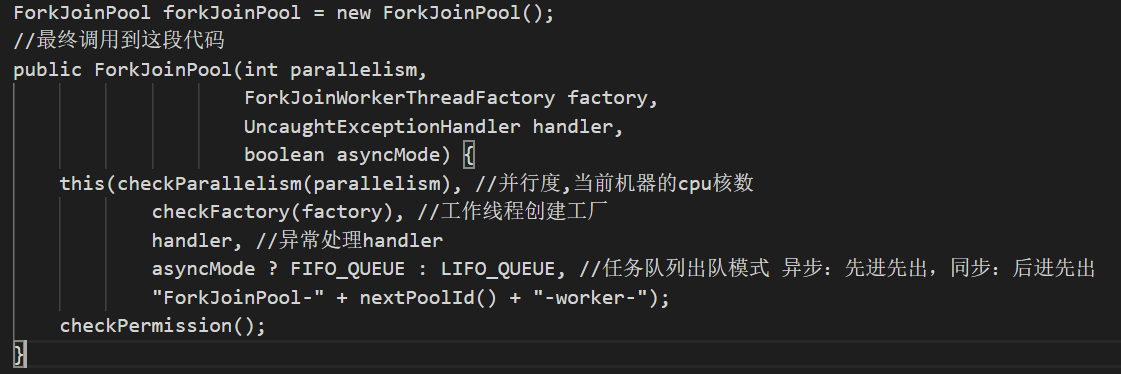
大到一个系统，小到一个框架，初始化工作往往是体现逻辑的一个重要地方！因为这是开始的地方，后面的逻辑会有依赖！所以把初始化看明白了，后面很多逻辑就容易理解多了。

下面上一段代码，(ps:这段代码是在网上找到的，并做了一小部分的修改)



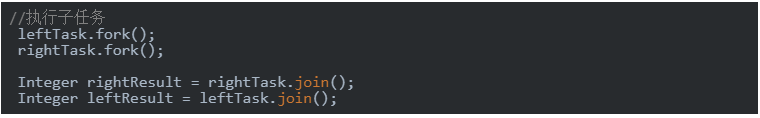
代码的执行过程解释起来也是很简单就是把[1,200]，分成[1,100],[101,200],然后再对每个部分进行一个递归分解最终分解成[1,2],[3,4],[5,6]…..[199,200]独立的小任务，然后两两求和合并。

其实显然易见负责整个fork/join初始化工作的就是ForkJoinPool！初始化代码就是那一行 ForkJoinPool forkJoinPool = new ForkJoinPool(),点进去查看源码。



看完初始化的代码我们可以知道原来创建ForkJoinPool创建workerThread的工作都是统一由一个叫ForkJoinWorkerThreadFactory的工厂去创建，创建出来的线程都有一个统一的前辍名称"ForkJoinPool-" + nextPoolId() + "-worker-"。队列出队模式是LIFO(后进先出)，那这样后面的入队的任务是会被先处理的。

所以上面提到对代码做了一些修改就是先处理rightTask，再处理leftTask。这其实是对代码的一种优化！



## 4、任务的提交逻辑？

fork/join其实大部分逻辑处理操作都集中在提交任务和处理任务这两块，了解任务的提交基本上后面就很容易理解了。

### fork/join提交任务主要分为两种：

第一种:第一次提交到forkJoinPool



第二种:任务切分之后的提交



### 提交到forkJoinPool :

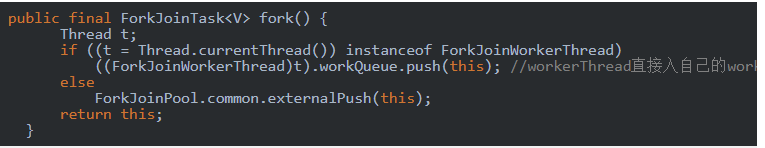
代码调用路径 submit(ForkJoinTask<T> task) -> externalPush(ForkJoinTask<?> task) -> externalSubmit(ForkJoinTask<?> task)

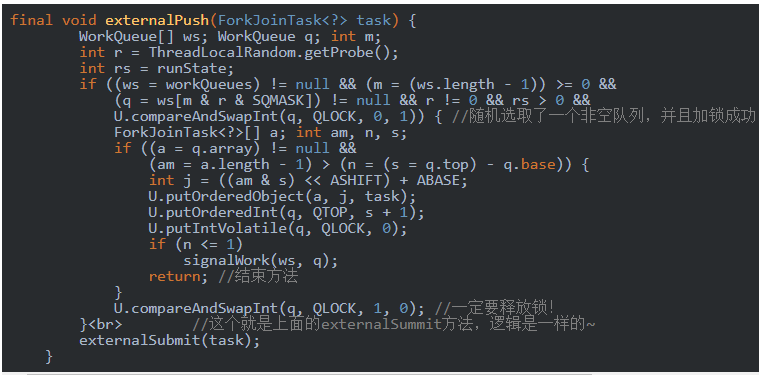
下面贴上externalSubmit的详细代码，着重留意注释的部分。

通过对externalSubmit方法的代码进行分析，我们知道了第一次提交任务给forkJoinPool时是在无限循环for (;;)中入队。第一步先检查workQueues是不是还没有创建，如果没有，则进行创建。之后跳到外层for循环并随机选取workQueues里面一个队列，并判断队列是否已创建。没有创建，则进行创建！后又跳到外层for循环直到选到一个非空队列并且加锁成功！这样最后才把任务入队~。

所以我们知道fork/join的任务队列workQueues并不是初始化的时候就创建好了，而是在有任务提交的时候才创建！并且每次入队时都需要利用cas操作来进行加锁和释放锁！

任务切分之后的提交：

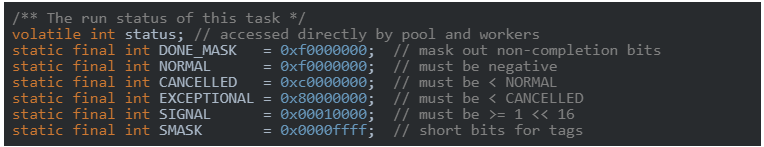


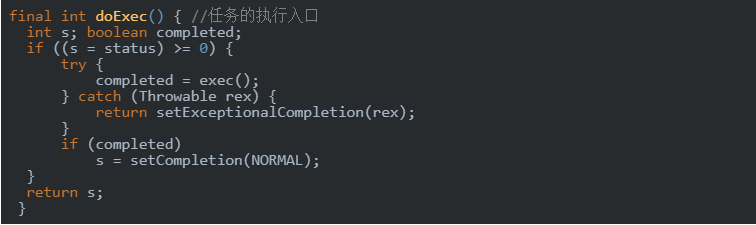


从代码中我们知道了提交一个fork任务的过程和第一次提交到forkJoinPool的过程是大同小异的。主要区分了提交任务的线程是不是workerThread，如果是，任务直接入workerThread当前的workQueue，不是则尝试选中一个workQueue q。如果q非空并且加锁成功则进行入队，否则执行与第一次任务提交到forkJoinPool差不多的逻辑~。

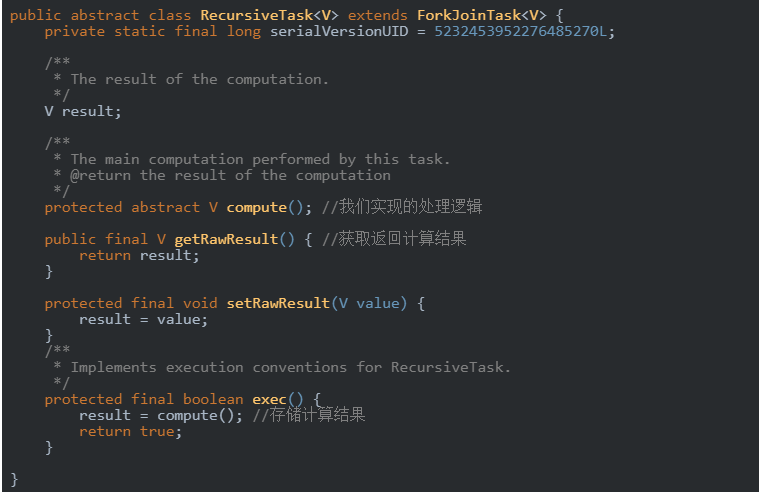
## 5、任务的消费

提交到任务的最终目的，是为了消费任务并最终获取到我们想要的结果。介绍任务消费之前我们先了解一个我们的任务ForkJoinTask有哪些关键属性和方法。





再看一下RecursiveTask的定义



在代码中我们看到任务的真正执行链路是 doExec -> exec -> compute -> 最后设置status 和 result。既然定义状态status并且还是volatile类型我们可以推断出workerThread在获取到执行任务之后都会先判断status是不是已完成或者异常状态，才决定要不要处理该任务。

下面看一下任务真正的处理逻辑代码！



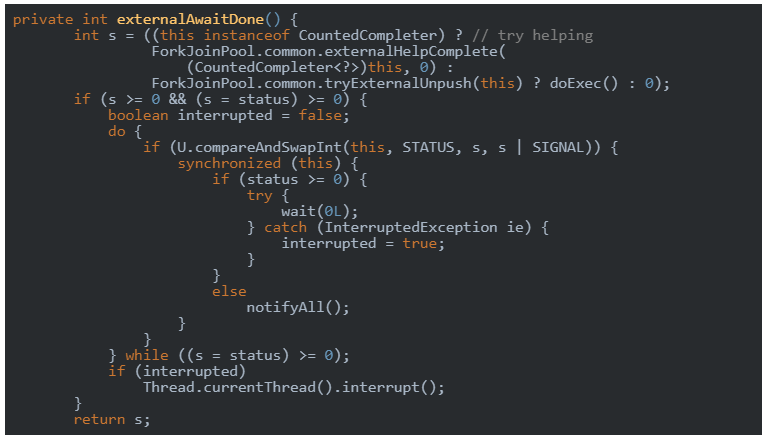
### 代码的调用链是从上到下。整体处理逻辑如下：

线程是workerThread：

先判断任务是否已经处理完成，任务完成直接返回，没有则直接尝试出队tryUnpush(this) 然后执行任务处理doExec()。如果没有出队成功或者处理成功，则执行wt.pool.awaitJoin(w, this, 0L)。wt.pool.awaitJoin(w, this, 0L)的处理逻辑简单来说也是在一个for(;;）中不断的轮询任务的状态是不是已完成，完成就直接退出方法。否就继续尝试出队处理。直到任务完成或者超时为止。

线程不是workerThread:

直接进行入externalAwaitDone()



externalAwaitDone的处理逻辑其实也比较简单，当前线程自己先尝试把任务出队ForkJoinPool.common.tryExternalUnpush(this) ? doExec()然后处理掉，如果不成功就交给workerThread去处理，然后利用object/wait的经典方法去监听任务status的状态变更。

## 6、任务的窃取

一直说fork/join的任务是work-stealing(工作窃取)，那任务究竟是怎么被窃取的呢。我们分析一下任务是由workThread来窃取的，workThread是一个线程。线程的所有逻辑都是由run()方法执行，所以任务的窃取逻辑一定在run()方法中可以找到！



其实只要看下面的英文注释就知道了大概scan(WorkQueue w, int r)就是用来窃取任务的！

所以我们知道任务的窃取从workerThread运行的那一刻就已经开始了！先随机选中一条队列看能不能窃取到任务，取不到则窃取下一条队列，直接遍历完一遍所有的队列，如果都窃取不到就返回null。