基于JDK的ForkJoin构建一个简单易用的并发组件

2018 年 04 月 09 日

基于ForkJoin构建一个简单易用的并发组件

在实际的业务开发中，需要用到并发编程的知识，实际使用线程池来异步执行任务的场景并不是特别多，而且一般真的遇到了需要并发使用的时候，可能更加常见的就是直接实现Runnable/Callable接口，丢到Thread中执行了；或者更高级一点，定义一个线程池，扔进去执行；本片博文，将从另一个角度，借助JDK提供的ForkJoin，来设计一个简单易用的并发框架

I. 背景

实际项目中，使用并发的一个case就是商品详情页的展示了，一个详情页的展示，除了基本的商品数据之外，还有销量，地址，评价，推荐，店铺信息，装饰信息等，用一段伪代码来描述拼装整个详情数据的过程

```
// 获取商品基本信息
ItemInfo itemInfo = itemService.getInfo(itemId);

// 获取销量
int sellCount = sellService.getSellCount(itemId);

// 获取评价信息
RateInfo rateInfo = rateService.getRateInfo(itemId);


// 获取店铺信息
ShopInfo shopInfo = shopService.getShopInfo(shopId);


// 获取装饰信息
DecorateInfo decoreateInfo = decorateService.getDecorateInfo(itemId);

// 获取推荐商品
RecommandInfo recommandInfo = recommandService.getRecommand(itemId);
```

如果是正常的执行过程，那么就是上面的6个调用，串行的执行下来，假设每个服务的rt是10ms，那么光是这里六个服务执行下来，耗时就>60ms了，

但从业务角度出发，上面6个服务调用，彼此之间没有什么关联，即一个服务的调用，并不依赖另一个服务返回的结果，她们完全可以并发执行，这样六个服务执行下来，耗时就是六个服务中耗时最久的一个了，可能也就10ms多一点了

两个一对比，发现这种场景下，使用并发的优势非常明显了，接下来的问题是，我们希望以最简单的方式，将上面的代码改成并发的

II. 设计与实现

以上面的case为例，如果我们采用线程池的方式，可以怎么实现呢?

1. 线程池方式

因为线程池方式不是重点，所以就简单的演示以下，可以怎么实现，以及实现之后的效果如何

```
// 1. 创建线程池
ExecutorService alarmExecutorService = new ThreadPoolExecutor(3, 5, 60,
                TimeUnit.SECONDS,
                new LinkedBlockingDeque<>(10),
                new DefaultThreadFactory("service-pool"),
                new ThreadPoolExecutor.CallerRunsPolicy());


// 2. 将服务调用，封装到线程任务中执行
Future<ItemInfo> itemFuture = alarmExecutorService.submit(new Callable<Item
    @Override
    public ItemInfo call() throws Exception {
        return itemService.getInfo(itemId);
    }
});

// ... 其他的服务依次类推


// 3. 获取数据
ItemInfo = itemFutre.get(); // 阻塞，直到返回
```

上面这个实现可以说是一个非常清晰明了的实现方式了，我们接下来看一下，用Fork/Join框架可以怎么玩，又会有什么好处

2. ForkJoin方式

首先可能需要简单的介绍下，这是个什么东西，Fork/Join框架是Java7提供了的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架

简单来说，就是讲一个复杂的任务，拆分成很多小任务，并发去执行的机制，任务与任务的执行，可能并不会独占线程，采用了一种名为工作窃取的手段，详情可以参考

ForkJoin 学习使用笔记

借助ForkJoin的方式，又可以怎么支持上面的场景呢？一个简单的方案如下

```
// 1. 创建池
ForkJoinPool pool = new ForkJoinPool(10);
```

```
// 2. 创建任务并提交
ForkJoinTask<ItemInfo> future = joinPool.submit(new RecursiveTask<ItemInfo>
    public ItemInfo compute() {
        return itemService.getItemInfo(itemId);
    }
});
```

```
// 3. 获取结果
future.join();
```

这样一对比，两者之间并没有什么区别，而且也没有用到传说中的任务拆解

3. 进阶

如何能够充分的利用ForkJoin的任务拆解的思想来解决问题呢？

将上面的实例，我们稍微变通一下，将整个详情页的数据返回，看做是一个任务，对于内部的服务调用，根据不同的应用提供放，再进行任务划分，假

设可以变成如下的层次结构

从上图可以看出，前面的服务调用，还可以继续划分，比如我们常见的商品信息，就可以区分为基本商品信息，sku信息，库存信息，而这三个又是可以并发执行的，也就是说从，借助forjoin的任务拆解，我们完全可以做到更细粒度的并发场景

那么现在的目标就是，如何实现上面这个任务拆分的场景需求，而且还希望对既有的代码改动不太大，关键还在于写出来后，得容易看懂+维护（这点其实很重要，笔者接触过一个封装得特别好，导致业务交接的维护成本太大，以及排查问题难度飙升的情况）

4. 实现

a. 设计思路

首先是定义一个最基本的执行单元，也就是封装具体的业务逻辑，也就是我们常说的Task（最终的效果也就是一个一个的task进行执行任务）

因为考虑到任务的拆解的情况，所以我们需要一个特殊的task，这个task可以是多个task的集合（也就是大任务，先称为bigTask）

然后就是使用时，所有的task都封装在一个bigTask中，直接丢给forkJoinPool来执行（支持同步获取结果的invoke调用方式和异步获取结果的execute方式）

那么，核心就在与如何设计这个BigTask了，以及在执行时，将bigTask拆解

成更细粒度的bigTask或者task，并最终将所有的task执行结果合并起来并返回

b. 实现

基本task接口

```
/**
 * Created by yihui on 2018/4/8.
 */
public interface IDataLoader<T> {


    /**
     * 具体的业务逻辑，放在这个方法里面执行，将返回的结果，封装到context内
     *
     * @param context
     */
    void load(T context);

}
```

一个抽象的实现类，继承forkjoin的RecuriAction，这个就对应上我们前面定义的基本Task了

```
public abstract class AbstractDataLoader<T> extends RecursiveAction impleme

    // 这里就是用来保存返回的结果，由业务防自己在实现的load()方法中写入数据
    protected T context;

    public AbstractDataLoader(T context) {
        this.context = context;
    }

    public void compute() {
        load(context);
    }


    /**
     * 获取执行后的结果，强制等待执行完毕
     * @return
     */
    public T getContext() {
```

```
        this.join();
        return context;
    }

    public void setContext(T context) {
        this.context = context;
    }
}
```

然后就是BigTask的实现了，也比较简单，内部维持一个List

```
public class DefaultForkJoinDataLoader<T> extends AbstractDataLoader<T> {
    /**
     * 待执行的任务列表
     */
    private List<AbstractDataLoader> taskList;


    public DefaultForkJoinDataLoader(T context) {
        super(context);
        taskList = new ArrayList<>();
    }

public DefaultForkJoinDataLoader<T> addTask(IDataLoader dataLoader) {
    taskList.add(new AbstractDataLoader(this.context) {
        @Override
        public void load(Object context) {
            dataLoader.load(context);
        }
    });
    return this;
}


// 注意这里，借助fork对任务进行了拆解
@Override
public void load(Object context) {
    this.taskList.forEach(ForkJoinTask::fork);
}


/**
 * 获取执行后的结果
 * @return
 */
public T getContext() {
```

```
        this.taskList.forEach(ForkJoinTask::join);
        return this.context;
    }
}
```

接下来就是比较简单的线程池的设计了，因为我们需要提供同步获取结果，和异步获取结果的两种姿势，所以对ForkJoinPool需要做个扩展

```
public class ExtendForkJoinPool extends ForkJoinPool {

    public ExtendForkJoinPool() {
    }

    public ExtendForkJoinPool(int parallelism) {
        super(parallelism);
    }

    public ExtendForkJoinPool(int parallelism, ForkJoinWorkerThreadFactory
        super(parallelism, factory, handler, asyncMode);
    }

    // 同步阻塞调用时，需要对每个task执行join，确保执行完毕
    public <T> T invoke(ForkJoinTask<T> task) {
        if (task instanceof AbstractDataLoader) {
            super.invoke(task);
            return (T) ((AbstractDataLoader) task).getContext();
        } else {
            return super.invoke(task);
        }
    }
}
```

然后就是创建Pool的工厂类，没什么特别的了

```
public class ForkJoinPoolFactory {

    private int parallelism;

    private ExtendForkJoinPool forkJoinPool;

    public ForkJoinPoolFactory() {
        this(Runtime.getRuntime().availableProcessors() * 16);
    }
```

```
    public ForkJoinPoolFactory(int parallelism) {
        this.parallelism = parallelism;
        forkJoinPool = new ExtendForkJoinPool(parallelism);
    }

    public ExtendForkJoinPool getObject() {
        return this.forkJoinPool;
    }

    public int getParallelism() {
        return parallelism;
    }

    public void setParallelism(int parallelism) {
        this.parallelism = parallelism;
    }


    public void destroy() throws Exception {
        this.forkJoinPool.shutdown();
    }

}
```

到此，整个基本上算是完了，每个类都很简单，就那么点东西，接下来就是需要看怎么用了

III. 测试验证

先来一个简单的case，演示下，应该怎么用

```
 @Data
static class Context {
    public int addAns;

    public int mulAns;

    public String concatAns;

    public Map<String, Object> ans = new ConcurrentHashMap<>();
}


@Test
public void testForkJoinFramework() {
```

```
ForkJoinPool forkJoinPool = new ForkJoinPoolFactory().getObject();

Context context = new Context();
DefaultForkJoinDataLoader<Context> loader = new DefaultForkJoinDataLoad
loader.addTask(new IDataLoader<Context>() {
    @Override
    public void load(Context context) {
        context.addAns = 100;
        System.out.println("add thread: " + Thread.currentThread());
    }
});
loader.addTask(new IDataLoader<Context>() {
    @Override
    public void load(Context context) {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        context.mulAns = 50;
        System.out.println("mul thread: " + Thread.currentThread());
    }
});
loader.addTask(new IDataLoader<Context>() {
    @Override
    public void load(Context context) {
        context.concatAns = "hell world";
        System.out.println("concat thread: " + Thread.currentThread());
    }
});


DefaultForkJoinDataLoader<Context> subTask = new DefaultForkJoinDataLoa
subTask.addTask(new IDataLoader<Context>() {
    @Override
    public void load(Context context) {
        System.out.println("sub thread1: " + Thread.currentThread() + "
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        context.ans.put(Thread.currentThread().getName(), System.curren

    }
});
subTask.addTask(new IDataLoader<Context>() {
    @Override
```

```
        public void load(Context context) {
            System.out.println("sub thread2: " + Thread.currentThread() + "
            context.ans.put(Thread.currentThread().getName(), System.curren
        }
    });

    loader.addTask(subTask);


    long start = System.currentTimeMillis();
    System.out.println("------- start: " + start);

    // 提交任务，同步阻塞调用方式
    forkJoinPool.invoke(loader);


    System.out.println("------- end: " + (System.currentTimeMillis() - star

    // 输出返回结果，要求3s后输出，所有的结果都设置完毕
    System.out.println("the ans: " + context);
}
```

使用起来就比较简单了，简单的四步骤即可：

- 创建Pool
- 指定保存结果的容器类ContextHolder
- 创建任务
    - 创建根任务 `new DefaultForkJoinDataLoader<>(context);`
    - 添加子任务
- 提交

上面这个实现中，对于需要将Task进行再次拆分，会变得非常简单，看下上面的输出

```
------- start: 1523200221827
add thread: Thread[ForkJoinPool-1-worker-50,5,main]
concat thread: Thread[ForkJoinPool-1-worker-36,5,main]
sub thread2: Thread[ForkJoinPool-1-worker-29,5,main] | now: 1523200222000
sub thread1: Thread[ForkJoinPool-1-worker-36,5,main] | now: 1523200222000
mul thread: Thread[ForkJoinPool-1-worker-43,5,main]
------- end: 3176
the ans: ForJoinTest.Context(addAns=100, mulAns=50, concatAns=hell world, a
```

- 首先是各个子任务执行的线程输出可以看出确实是不同线程执行的任务（并发）
- 3s后，输出结果，即invoke之后，会阻塞直到所有的任务执行完毕
- subTask进行了任务拆解，两个子任务的执行时间相同，但是一个sleep，另一个则不受影响（子任务也是并行执行）

对于希望异步执行的情况，也比较简单了，仅仅是在提交任务的地方，稍微改动一下即可，然后在需要获取数据的时候，通过loader来获取结果即可

```
@Test
public void testForkJoinFramework2() {
    ForkJoinPool forkJoinPool = new ForkJoinPoolFactory().getObject();

    Context context = new Context();
    DefaultForkJoinDataLoader<Context> loader = new DefaultForkJoinDataLoad
    loader.addTask(new IDataLoader<Context>() {
        @Override
        public void load(Context context) {
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            context.addAns = 100;
            System.out.println("add thread: " + Thread.currentThread());
        }
    });
    loader.addTask(new IDataLoader<Context>() {
        @Override
        public void load(Context context) {
            context.mulAns = 50;
            System.out.println("mul thread: " + Thread.currentThread());
        }
    });
    loader.addTask(new IDataLoader<Context>() {
        @Override
        public void load(Context context) {
            context.concatAns = "hell world";
            System.out.println("concat thread: " + Thread.currentThread());
        }
    });


    long start = System.currentTimeMillis();
    System.out.println("------- start: " + start);
```

```
    // 如果暂时不关心返回结果，可以采用execute方式，异步执行
    forkJoinPool.execute(loader);

    // .... 这里可以做其他的事情 此时，不会阻塞，addAns不会被设置
    System.out.println("context is: " + context);
    System.out.println("------- then: " + (System.currentTimeMillis() - sta


    loader.getContext(); // 主动调用这个，表示会等待所有任务执行完毕后，才继续下去
    System.out.println("context is: " + context);
    System.out.println("------- end: " + (System.currentTimeMillis() - star
}
```

IV. 其他

源码

相关源码可在git上查看，主要在Quick-Alarm项目中

- QuickAlarm
- 并发相关代码

个人博客： 一灰灰Blog

个人博客，记录所有学习和工作中的博文，欢迎大家前去逛逛

声明

尽信书则不如，已上内容，纯属一家之言，因本人能力一般，见识有限，如发现bug或者有更好的建议，随时欢迎批评指正

- 微博地址: 小灰灰Blog
- QQ： 一灰灰/3302797840

扫描关注