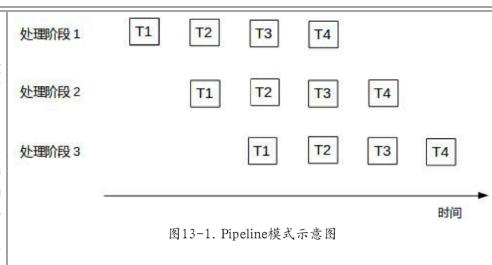
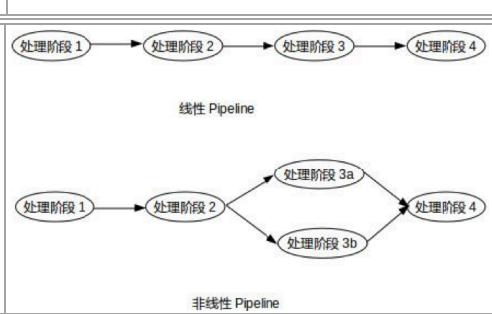
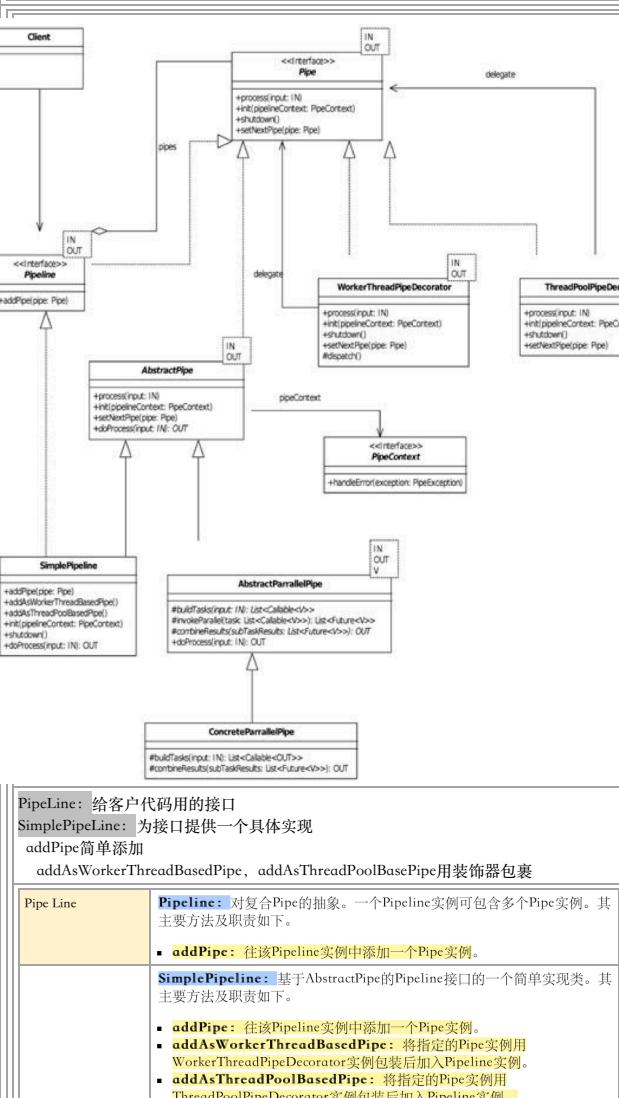


发件人: 方堃 fangkun119@icloud.com
 主题: CH13 Pipe Line
 日期: 2017年5月2日 下午12:03
 收件人:

方

CH13 Pipe Line

2017年4月30日 星期日
 下午4:51

功能	<p>Pipeline模式的核心思想是将一个任务处理分解为若干个处理阶段(Stage), 其中每个处理阶段的输出作为下一个处理阶段的输入, 并且各个处理阶段都有相应的工作者线程去执行相应的计算。因此, 处理一批任务时, 各个任务的各个处理阶段是平行(Parallel)的。通过并行计算, Pipeline模式使应用程序能够充分利用多核CPU资源, 提高其计算效率。</p> <p>假设有一批任务(T1, T2, T3, T4), 其中每个任务的处理可分解为3个处理阶段。虽然, 对于这一批任务中的某一个任务而言, 其处理仍然是串行的, 即完成一个任务的处理要依次执行各个处理阶段, 但从整体任务上看, 各个处理阶段的执行是平行的。比如, 处理阶段1的工作者线程执行完任务T1相应的计算后, 其处理结果会被提交给处理阶段2作为输入; 当处理阶段2的工作者线程正在执行任务T1的相应的计算时, 处理阶段1正在执行任务T2相应的计算, 此时这两个处理阶段是平行的, 如图13-1所示。</p>	 <p>图13-1. Pipeline模式示意图</p>																										
实现方式	<p>Pipeline模式将任务的处理分解为若干个处理阶段, 并将其中的每个阶段抽象为一个对象。这些表示处理阶段的对象都有其工作者线程负责对输入进行处理, 并将输出作为下一个处理阶段的输入。</p> <p>按照处理阶段中是否包含并发处理阶段, Pipeline模式可分为线性Pipeline和非线性Pipeline两种, 如图13-2所示。图13-2中的非线性Pipeline中的第3个处理阶段包含了两个可并发执行的操作。</p>	 <p>图13-2. Pipeline实现方式示意图</p>																										
类图	 <p>图13-3. Pipeline设计模式类图</p>	<table border="1"> <tr> <td data-bbox="896 999 976 1123">PIPE</td><td data-bbox="976 999 1388 1123"> Pipe: 对处理阶段的抽象, 负责对输入进行处理, 并将输出作为下一个处理阶段的输入。因此, 一个Pipe实例可以理解为包含输入、输出和处理的三元组。其主要方法及职责如下。 </td></tr> <tr> <td data-bbox="896 1123 976 1235">抽象Pipe</td><td data-bbox="976 1123 1388 1235"> <ul style="list-style-type: none"> process: 用于接收前一个处理阶段的处理结果, 作为该处理阶段的输入。 init: 初始化当前处理阶段对外提供的服务。 shutdown: 关闭当前处理阶段对外提供的服务。 setNextPipe: 设置当前处理阶段的下一个处理阶段。 </td></tr> <tr> <td data-bbox="896 1235 976 1347">Pipe接口</td><td data-bbox="976 1235 1388 1347"> AbstractPipe: Pipe接口的抽象实现类。其主要方法及职责如下。 </td></tr> <tr> <td data-bbox="896 1347 976 1459">Pipe装饰</td><td data-bbox="976 1347 1388 1459"> <ul style="list-style-type: none"> process: 接收前一个处理阶段的输入, 并调用其子类实现的doProcess方法对输入元素进行处理。相应的处理结果会被提交给下一个处理阶段作为输入。 init: 保存对其参数中指定的PipeContext实现的引用。子类可根据需要覆盖该方法以实现其服务的初始化。 shutdown: 默认实现什么也不做。子类可根据需要覆盖该方法实现服务停止。 setNextPipe: 设置当前处理阶段的下一个处理阶段。 doProcess: 留给子类实现的抽象方法。用于子类实现其对输入元素的处理逻辑。 </td></tr> <tr> <td data-bbox="896 1459 976 1572">Pipe装饰</td><td data-bbox="976 1459 1388 1572"> PipeContext: 对各个处理阶段的计算环境进行抽象, 主要用于异常处理。其主要方法及职责如下。 </td></tr> <tr> <td data-bbox="896 1572 976 1684">并行Pipe</td><td data-bbox="976 1572 1388 1684"> WorkerThreadPipeDecorator: 基于工作者线程的Pipe实现类。该Pipe实例会将接收到的输入元素存入队列, 由指定个体的工作者线程对队列中的输入元素进行处理。该类本身主要负责工作者线程的生命周期的管理。它通过调用delegate实例委派给指定的Pipe实例(以下简称委托Pipe实例)的相应方法实现Pipe接口中定义的各个方法。其主要方法及职责如下。 </td></tr> <tr> <td data-bbox="896 1684 976 1796"></td><td data-bbox="976 1684 1388 1796"> <ul style="list-style-type: none"> process: 接收前一个处理阶段的输入, 并将其存入队列, 由工作者线程运行时取出进行处理。 init: 启动工作者线程, 并调用委托Pipe实例的init方法。 shutdown: 停止工作者线程, 并调用委托Pipe实例的shutdown方法。 setNextPipe: 调用委托Pipe实例的setNextPipe方法。 dispatch: 取队列中的输入元素, 并调用委托Pipe实例的process方法对其进行处理。 </td></tr> <tr> <td data-bbox="896 1796 976 1909"></td><td data-bbox="976 1796 1388 1909"> ThreadPoolPipeDecorator: 基于线程池的Pipe实现类。该类主要实现用线程池中的工作线程去执行对各个输入元素的处理。它通过delegate实例调用所指定的Pipe实例(以下简称委托Pipe实例)的相应方法实现Pipe接口中定义的各个方法。其主要方法及职责如下。 </td></tr> <tr> <td data-bbox="896 1909 976 2021"></td><td data-bbox="976 1909 1388 2021"> <ul style="list-style-type: none"> process: 接收前一个处理阶段的输入, 并向线程池提交一个对该输入进行相应处理的任务。 init: 用委托Pipe实例的init方法。 shutdown: 关闭当前的Pipe实例对外提供的服务, 并调用委托Pipe实例的shutdown方法。 setNextPipe: 调用委托Pipe实例的setNextPipe方法。 </td></tr> <tr> <td data-bbox="896 2021 976 2097"></td><td data-bbox="976 2021 1388 2097"> AbstractParallelPipe: AbstractPipe的子类, 支持并行处理的Pipe实现类, 该类对每个输入元素(原始任务)生成相对应的一组子任务, 并以并行的方式去执行这些子任务。各个子任务的执行结果会被合并为相应原始任务的输出结果。其主要方法及职责如下。 </td></tr> <tr> <td data-bbox="976 2021 1388 2097"></td><td data-bbox="976 2021 1388 2097"> <ul style="list-style-type: none"> buildTasks: 留给子类实现的抽象方法。用于根据指定的输入构造一组子任务。 combineResults: 留给子类实现的抽象方法。对各个并行任务的处理结果进行合并, 形成相对输入元素的输出结果。 invokeParallel: 实现以并行的方式执行一组任务。 doProcess: 实现该类对输入的处理逻辑。 </td></tr> <tr> <td data-bbox="976 2021 1388 2097"></td><td data-bbox="976 2021 1388 2097"> ConcreteParallelPipe: 应用定义的AbstractParallelPipe的子类。其主要方法及职责如下。 </td></tr> <tr> <td data-bbox="976 2021 1388 2097"></td><td data-bbox="976 2021 1388 2097"> <ul style="list-style-type: none"> buildTasks: 根据指定的输入构造一组任务。 combineResults: 对各个并行任务的处理结果进行合并, 形成相对输入元素的输出结果。 </td></tr> </table>	PIPE	Pipe : 对处理阶段的抽象, 负责对输入进行处理, 并将输出作为下一个处理阶段的输入。因此, 一个Pipe实例可以理解为包含输入、输出和处理的三元组。其主要方法及职责如下。	抽象Pipe	<ul style="list-style-type: none"> process: 用于接收前一个处理阶段的处理结果, 作为该处理阶段的输入。 init: 初始化当前处理阶段对外提供的服务。 shutdown: 关闭当前处理阶段对外提供的服务。 setNextPipe: 设置当前处理阶段的下一个处理阶段。 	Pipe接口	AbstractPipe : Pipe接口的抽象实现类。其主要方法及职责如下。	Pipe装饰	<ul style="list-style-type: none"> process: 接收前一个处理阶段的输入, 并调用其子类实现的doProcess方法对输入元素进行处理。相应的处理结果会被提交给下一个处理阶段作为输入。 init: 保存对其参数中指定的PipeContext实现的引用。子类可根据需要覆盖该方法以实现其服务的初始化。 shutdown: 默认实现什么也不做。子类可根据需要覆盖该方法实现服务停止。 setNextPipe: 设置当前处理阶段的下一个处理阶段。 doProcess: 留给子类实现的抽象方法。用于子类实现其对输入元素的处理逻辑。 	Pipe装饰	PipeContext : 对各个处理阶段的计算环境进行抽象, 主要用于异常处理。其主要方法及职责如下。	并行Pipe	WorkerThreadPipeDecorator : 基于工作者线程的Pipe实现类。该Pipe实例会将接收到的输入元素存入队列, 由指定个体的工作者线程对队列中的输入元素进行处理。该类本身主要负责工作者线程的生命周期的管理。它通过调用delegate实例委派给指定的Pipe实例(以下简称委托Pipe实例)的相应方法实现Pipe接口中定义的各个方法。其主要方法及职责如下。		<ul style="list-style-type: none"> process: 接收前一个处理阶段的输入, 并将其存入队列, 由工作者线程运行时取出进行处理。 init: 启动工作者线程, 并调用委托Pipe实例的init方法。 shutdown: 停止工作者线程, 并调用委托Pipe实例的shutdown方法。 setNextPipe: 调用委托Pipe实例的setNextPipe方法。 dispatch: 取队列中的输入元素, 并调用委托Pipe实例的process方法对其进行处理。 		ThreadPoolPipeDecorator : 基于线程池的Pipe实现类。该类主要实现用线程池中的工作线程去执行对各个输入元素的处理。它通过delegate实例调用所指定的Pipe实例(以下简称委托Pipe实例)的相应方法实现Pipe接口中定义的各个方法。其主要方法及职责如下。		<ul style="list-style-type: none"> process: 接收前一个处理阶段的输入, 并向线程池提交一个对该输入进行相应处理的任务。 init: 用委托Pipe实例的init方法。 shutdown: 关闭当前的Pipe实例对外提供的服务, 并调用委托Pipe实例的shutdown方法。 setNextPipe: 调用委托Pipe实例的setNextPipe方法。 		AbstractParallelPipe : AbstractPipe的子类, 支持并行处理的Pipe实现类, 该类对每个输入元素(原始任务)生成相对应的一组子任务, 并以并行的方式去执行这些子任务。各个子任务的执行结果会被合并为相应原始任务的输出结果。其主要方法及职责如下。		<ul style="list-style-type: none"> buildTasks: 留给子类实现的抽象方法。用于根据指定的输入构造一组子任务。 combineResults: 留给子类实现的抽象方法。对各个并行任务的处理结果进行合并, 形成相对输入元素的输出结果。 invokeParallel: 实现以并行的方式执行一组任务。 doProcess: 实现该类对输入的处理逻辑。 		ConcreteParallelPipe : 应用定义的AbstractParallelPipe的子类。其主要方法及职责如下。		<ul style="list-style-type: none"> buildTasks: 根据指定的输入构造一组任务。 combineResults: 对各个并行任务的处理结果进行合并, 形成相对输入元素的输出结果。
PIPE	Pipe : 对处理阶段的抽象, 负责对输入进行处理, 并将输出作为下一个处理阶段的输入。因此, 一个Pipe实例可以理解为包含输入、输出和处理的三元组。其主要方法及职责如下。																											
抽象Pipe	<ul style="list-style-type: none"> process: 用于接收前一个处理阶段的处理结果, 作为该处理阶段的输入。 init: 初始化当前处理阶段对外提供的服务。 shutdown: 关闭当前处理阶段对外提供的服务。 setNextPipe: 设置当前处理阶段的下一个处理阶段。 																											
Pipe接口	AbstractPipe : Pipe接口的抽象实现类。其主要方法及职责如下。																											
Pipe装饰	<ul style="list-style-type: none"> process: 接收前一个处理阶段的输入, 并调用其子类实现的doProcess方法对输入元素进行处理。相应的处理结果会被提交给下一个处理阶段作为输入。 init: 保存对其参数中指定的PipeContext实现的引用。子类可根据需要覆盖该方法以实现其服务的初始化。 shutdown: 默认实现什么也不做。子类可根据需要覆盖该方法实现服务停止。 setNextPipe: 设置当前处理阶段的下一个处理阶段。 doProcess: 留给子类实现的抽象方法。用于子类实现其对输入元素的处理逻辑。 																											
Pipe装饰	PipeContext : 对各个处理阶段的计算环境进行抽象, 主要用于异常处理。其主要方法及职责如下。																											
并行Pipe	WorkerThreadPipeDecorator : 基于工作者线程的Pipe实现类。该Pipe实例会将接收到的输入元素存入队列, 由指定个体的工作者线程对队列中的输入元素进行处理。该类本身主要负责工作者线程的生命周期的管理。它通过调用delegate实例委派给指定的Pipe实例(以下简称委托Pipe实例)的相应方法实现Pipe接口中定义的各个方法。其主要方法及职责如下。																											
	<ul style="list-style-type: none"> process: 接收前一个处理阶段的输入, 并将其存入队列, 由工作者线程运行时取出进行处理。 init: 启动工作者线程, 并调用委托Pipe实例的init方法。 shutdown: 停止工作者线程, 并调用委托Pipe实例的shutdown方法。 setNextPipe: 调用委托Pipe实例的setNextPipe方法。 dispatch: 取队列中的输入元素, 并调用委托Pipe实例的process方法对其进行处理。 																											
	ThreadPoolPipeDecorator : 基于线程池的Pipe实现类。该类主要实现用线程池中的工作线程去执行对各个输入元素的处理。它通过delegate实例调用所指定的Pipe实例(以下简称委托Pipe实例)的相应方法实现Pipe接口中定义的各个方法。其主要方法及职责如下。																											
	<ul style="list-style-type: none"> process: 接收前一个处理阶段的输入, 并向线程池提交一个对该输入进行相应处理的任务。 init: 用委托Pipe实例的init方法。 shutdown: 关闭当前的Pipe实例对外提供的服务, 并调用委托Pipe实例的shutdown方法。 setNextPipe: 调用委托Pipe实例的setNextPipe方法。 																											
	AbstractParallelPipe : AbstractPipe的子类, 支持并行处理的Pipe实现类, 该类对每个输入元素(原始任务)生成相对应的一组子任务, 并以并行的方式去执行这些子任务。各个子任务的执行结果会被合并为相应原始任务的输出结果。其主要方法及职责如下。																											
	<ul style="list-style-type: none"> buildTasks: 留给子类实现的抽象方法。用于根据指定的输入构造一组子任务。 combineResults: 留给子类实现的抽象方法。对各个并行任务的处理结果进行合并, 形成相对输入元素的输出结果。 invokeParallel: 实现以并行的方式执行一组任务。 doProcess: 实现该类对输入的处理逻辑。 																											
	ConcreteParallelPipe : 应用定义的AbstractParallelPipe的子类。其主要方法及职责如下。																											
	<ul style="list-style-type: none"> buildTasks: 根据指定的输入构造一组任务。 combineResults: 对各个并行任务的处理结果进行合并, 形成相对输入元素的输出结果。 																											

时序	初始化		第1步：客户端代码创建Pipeline参与者实例。 第2~4步：客户端代码创建其所需的各个Pipe参与者实例。 第5步：客户端代码调用Pipeline实例的add方法添加其创建的Pipe实例。 第6步：客户端代码创建PipeContext参与者实例。 第7步：客户端代码调用Pipeline实例的init方法。 第8~13步：init方法调用其所属Pipeline实例所包含的各个Pipe实例的init方法，以初始化这些Pipe实例所提供的服务。 第14步：init方法调用返回。
	添任务		第1步：客户端代码调用Pipeline参与者实例的process方法提交一个任务。 第2、3步：process方法调用当前Pipeline实例所包含的第一个Pipe实例的process方法。 第4步：process方法返回。 第5步：Pipeline实例中的第一个Pipe实例开始其任务处理，其dispatch方法被工作者线程（或者线程池中的工作者线程）调用。 第6步：dispatch方法调用pipe1的委托Pipe实例delegate1的process方法。 第7步：delegate1的process方法调用其doProcess方法对输入元素inputA进行处理，相应处理结果为outA。 第8、9步：delegate1的process方法获取其下一个Pipe实例（同时也是pipe1的下一个Pipe实例）pipe2，并调用pipe2的process方法，将outA作为输入元素提交给pipe2处理。 第10步：pipe1的process方法返回。
使用	点	<p>Pipeline模式可以对有依赖关系的任务实现并行处理。并行和并发编程中，为了提高并发性我们往往需要将规模较大的任务分解成若干个规模较小的子任务，这些子任务常没有依赖关系。而Pipeline模式则在允许子任务间存在依赖关系的条件下实现并行计算：Pipeline模式中的每个Pipe的输入元素可以看成提交给该Pipe的一个任务。这样，前Pipeline实例的任务处理结果就是下一个Pipe实例的输入任务。所以，从整体任务上看Pipeline模式对其他任务的处理是顺序的。由于每个Pipe实例都有其工作者线程负责任务处理，当一个Pipe实例处理其上游Pipe实例提交的某个任务时，其上游Pipe实例已经在处理其接收到的其他任务。因此，从整体任务上看，Pipeline对任务的处理又是并行的。</p> <p>Pipeline模式为使用单线程模型编程提供了便利。多线程编程总的来说是复杂的，不仅代码编写比较复杂，出现问题时也不好定位。多线程程序出现非预期结果时，开不仅要考虑算法是否正确，还要考虑是否是多线程相关问题（如线程安全），导致非预期的结果。相反，单线程编程就显得相对简单。Pipeline模式非常便于我们采用单线程实现对子任务的处理。比如，子任务的处理涉及非线程安全对象或者涉及阻塞I/O（Blocking I/O）操作时，我们不希望引入锁从而避免其增加上下文切换。此时单线程编程，本章案例就是这样一个个例子。Pipeline模式中，要对某种子任务采用单线程处理，开发人员只需要调用SimplePipeline的addAsWorkerThreadBasedPipe方法将负责该子任务的Pipe实例加入Pipeline即可，如清单13-1所示。</p> <p>Pipeline模式中，每个Pipeline实例都是一个Pipe实例。因此，我们不仅可以往一个Pipeline实例中添加Pipe实例，也可以添加其他Pipeline实例。这就允许应用复用Pipeline实例组合（Pipeline实例），或者改变一个Pipeline实例内部各个Pipe实例的编排而外界不受此影响。因此，Pipeline模式带来了一定的灵活性。</p> <p>当然，Pipeline模式提供的优势的背后也隐藏着代价。Pipeline模式中各个处理阶段所使用的工作者线程或者线程池、表示各个处理阶段的输入/输出对象的创建和移动队列）都有其自身的时间和空间开销。因此，Pipeline模式适合于处理规模较大的任务，否则可能得不偿失：Pipeline模式可能带来的好处无法抵消其背后的代价。当一个Pipe实例处理其上游Pipe实例提交的某个任务时，其上游Pipe实例已经在处理其接收到的其他任务。因此，从整体任务上看，Pipeline对任务的处理又是并行的。</p> <p>Pipeline模式为使用单线程模型编程提供了便利。多线程编程总的来说是复杂的，不仅代码编写比较复杂，出现问题时也不好定位。多线程程序出现非预期结果时，开不仅要考虑算法是否正确，还要考虑是否是多线程相关问题（如线程安全），导致非预期的结果。相反，单线程编程就显得相对简单。Pipeline模式非常便于我们采用单线程实现对子任务的处理。比如，子任务的处理涉及非线程安全对象或者涉及阻塞I/O（Blocking I/O）操作时，我们不希望引入锁从而避免其增加上下文切换。此时单线程编程，本章案例就是这样一个个例子。Pipeline模式中，要对某种子任务采用单线程处理，开发人员只需要调用SimplePipeline的addAsWorkerThreadBasedPipe方法将负责该子任务的Pipe实例加入Pipeline即可，如清单13-1所示。</p> <p>Pipeline模式中，每个Pipeline实例都是一个Pipe实例。因此，我们不仅可以往一个Pipeline实例中添加Pipe实例，也可以添加其他Pipeline实例。这就允许应用复用Pipeline实例组合（Pipeline实例），或者改变一个Pipeline实例内部各个Pipe实例的编排而外界不受此影响。因此，Pipeline模式带来了一定的灵活性。</p> <p>当然，Pipeline模式提供的优势的背后也隐藏着代价。Pipeline模式中各个处理阶段所使用的工作者线程或者线程池、表示各个处理阶段的输入/输出对象的创建和移动队列）都有其自身的时间和空间开销。因此，Pipeline模式适合于处理规模较大的任务，否则可能得不偿失：Pipeline模式可能带来的好处无法抵消其背后的代价。</p>	
注意事项	管道	<p>管道：下面以线性Pipeline为例讨论Pipeline的深度问题，非线性Pipeline同样可参考之。一个Pipeline实例中包含的Pipe实例的个数被称为Pipeline的深度。通常，我们需要Pipeline的深度与JVM宿主机的CPU个数（以下简称N_{CPU}）之间的关系。如果当前Pipeline实例所处理的任务多属CPU密集型，那么Pipeline的深度最好不超过N_{CPU}。如果Pipeline所处理的任务多属I/O密集型，则Pipeline的深度最好不超过$2 \times N_{CPU}$。</p> <p>按照上述规则，如果Pipeline的深度太大，或者Pipeline的深度并未“超标”但Pipeline中有不少Pipe实例需要使用多个工作者线程，那么此时我们可以考虑使用线程池来处理Pipeline实例的任务处理。实现这点，我们只需要调用SimplePipeline实例的addAsThreadPoolBasedPipe方法添加相关Pipe实例即可。另外，使用基于线程池的Pipe实例需要注意到某些问题，下文会提到这点。</p>	<p>输出</p> <pre> Task->[pipe1, pool-1-thread-1] Task->[pipe1, pool-1-thread-2] Task->[pipe1, pool-1-thread-3] Task->[pipe1, pool-1-thread-4] Task->[pipe1, pool-1-thread-5]-->[pipe2, pool-1-thread-8] Task->[pipe1, pool-1-thread-6]-->[pipe2, pool-1-thread-3] Task->[pipe1, pool-1-thread-7]-->[pipe2, pool-1-thread-5] Task->[pipe1, pool-1-thread-8]-->[pipe3, pool-1-thread-8] Task->[pipe1, main]-->[pipe2, main] Task->[pipe1, main]-->[pipe2, pool-1-thread-1]-->[pipe2, pool-1-thread-5] Task->[0]-->[pipe1, pool-1-thread-1]-->[pipe2, pool-1-thread-7] 此省略部分输出 Task->[4]-->[pipe1, pool-1-thread-8]-->[pipe2, pool-1-thread-8] Task->[5]-->[pipe1, pool-1-thread-1]-->[pipe2, pool-1-thread-1] Task->[6]-->[pipe1, main]-->[pipe2, main] Task->[7]-->[pipe1, main]-->[pipe2, main]-->[pipe3, pool-1-thread-5] Task->[8]-->[pipe1, pool-1-thread-1]-->[pipe2, pool-1-thread-1]-->[pipe3, pool-1-thread-8] Task->[9]-->[pipe1, pool-1-thread-8]-->[pipe2, pool-1-thread-8]-->[pipe3, pool-1-thread-8] Task->[10]-->[pipe1, main]-->[pipe2, main]-->[pipe3, pool-1-thread-4] </pre>
错误	<p>错误：Pipe实例对其实任务进行处理过程中抛出的异常可能需要在相应Pipe实例之外进行处理。此时，异常处理通常有两种方式。一种是本章案例所使用的方法，即各个Pipe实例到异常后调用PipeContext实例的handleError进行错误处理。另一种方法是创建一个专门负责错误处理的Pipe实例，其他Pipe实例捕获异常后提交相关数据给该Pipe实例处理。</p>		
可配置	<p>可配置：本章案例中，我们以代码的方式将若干个Pipe实例添加到Pipeline实例中。如果有必要的话，我们也可以借助配置文件等可配置的方式实现以动态的方式创建Pipeline实例，往其中添加所需的Pipe实例。</p>		
线程池	<p>线程池：当Pipeline的深度比较大，或者其深度不大但不少Pipe实例需要多个工作者线程负责任务处理时，为了避免创建过多的线程而增加资源的消耗以及上下文切换，我们可以考虑使用基于线程池的Pipe，即一个Pipeline实例中的多个Pipe实例共用一个（或者多个）线程池负责各个Pipe的任务处理。清单13-4展示了一个基于线程池的Pipe例子。同一个线程池负责执行提交给不同Pipe实例的任务，其本质是一个Pipeline实例中的多个Pipe实例作为相应线程池的客户端向线程池提交任务。因此，此情形下，无论是实现Pipe还是使用Pipeline模式都要注意各个Pipe实例给线程池提交的任务之间是否存在依赖关系。这是因为同一个线程池实例执行的各个任务若存在依赖关系，则可能导致线程有关线程池死锁的进一步信息，请参考第9章。</p>		
	<pre> public class ThreadPoolBasedPipeExample { public static void main(String[] args) { final ThreadPoolExecutor executorService; executorService = new ThreadPoolExecutor(1, Runtime.getRuntime() .availableProcessors() * 2, 60, TimeUnit.MINUTES, new SynchronousQueue<Runnable>(), new ThreadPoolExecutor.CallerRunsPolicy()); } final SimplePipeline<String, String> pipeline = new SimplePipeline<String, String>(); Pipe<String, String> pipe = new AbstractPipe<String, String>() { @Override protected String doProcess(String input) throws PipeException { String result = input + "->" + pipe.getName() + Thread.currentThread().getMethodName() + "!"; System.out.println(result); return result; } }; } </pre>	输出	

```

    ...
    pipeline.addAsThreadPoolBasedPipe(pipe, executorService);
    pipe = new AbstractPipe<String, String>() {
        @Override
        protected String doProcess(String input) throws PipeException {
            String result = input + "->[pipe2," +
                Thread.currentThread().getName() + "]";
            System.out.println(result);
            try {
                Thread.sleep(new Random().nextInt(100));
            } catch (InterruptedException e) {
                ;
            }
            return result;
        }
    };
    pipeline.addAsThreadPoolBasedPipe(pipe, executorService);
    pipe = new AbstractPipe<String, String>() {
        @Override
        protected String doProcess(String input) throws PipeException {
            String result = input + "->[pipe3," +
                Thread.currentThread().getName() + "]";
            System.out.println(result);
            try {
                Thread.sleep(new Random().nextInt(200));
            } catch (InterruptedException e) {
                ;
            }
            return result;
        }
    };
    @Override
    public void shutdown(long timeout, TimeUnit unit) {
        // 在最后一个Pipe中关闭线程池
        executorService.shutdown();
        try {
            executorService.awaitTermination(timeout, unit);
        } catch (InterruptedException e) {
            ;
        }
    }
}

pipeline.addAsThreadPoolBasedPipe(pipe, executorService);
    pipeline.init(pipeline.newDefaultPipeContext());
    int N = 100;
    try {
        for (int i = 0; i < N; i++) {
            pipeline.process("Task-" + i);
        }
    } catch (IllegalStateException e) {
        ;
    } catch (InterruptedException e) {
        ;
    }
    pipeline.shutdown(10, TimeUnit.SECONDS);
}
}

```

可复用代码	接口	<pre> public interface Pipe<IN, OUT> { /** * 设置当前Pipe实例的下一个Pipe实例。 * * @param nextPipe * 下一个Pipe实例 */ void setNextPipe(Pipe<?, ?> nextPipe); /** * 初始化当前Pipe实例对外提供的服务。 * * @param pipeCtx */ void init(PipeContext pipeCtx); /** * 停止当前Pipe实例对外提供的服务。 * * @param timeout * @param unit */ void shutdown(long timeout, TimeUnit unit); /** * 对输入元素进行处理，并将处理结果作为下一个Pipe实例的输入。 */ void process(IN input) throws InterruptedException; } </pre>
-------	----	--

```
    }

抽象 PIPE public abstract class AbstractPipe<IN, OUT> implements Pipe<IN, OUT> {
    protected volatile Pipe<?, ?> nextPipe = null;
    protected volatile PipeContext pipeCtx;

    @Override
    public void init(PipeContext pipeCtx) {
        this.pipeCtx = pipeCtx;
    }

    @Override
    public void setNextPipe(Pipe<?, ?> nextPipe) {
        this.nextPipe = nextPipe;
    }

    @Override
    public void shutdown(long timeout, TimeUnit unit) {
        // 什么也不做
    }

    /**
     * 留给子类实现。用于子类实现其任务处理逻辑。
     *
     * @param input
     *         输入元素（任务）
     * @return 任务的处理结果
     * @throws PipeException
     */
    protected abstract OUT doProcess(IN input) throws PipeException;
    @SuppressWarnings("unchecked")
    public void process(IN input) throws InterruptedException {
        try {
            OUT out = doProcess(input);
            if (null != nextPipe) {
                if (null != out) {
                    ((Pipe<OUT, ?>) nextPipe).process(out);
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } catch (PipeException e) {
            pipeCtx.handleError(e);
        }
    }
}

PIPE 线程装饰 public class WorkerThreadPipeDecorator<IN, OUT> implements Pipe<IN, OUT> {
    protected final BlockingQueue<IN> workQueue;
    private final Set<AbstractTerminatableThread> workerThreads = new
    HashSet<AbstractTerminatableThread>();
    private final TerminationToken terminationToken = new TerminationToken();

    private final Pipe<IN, OUT> delegate;

    public WorkerThreadPipeDecorator(Pipe<IN, OUT> delegate, int workerCount) {
        this(new SynchronousQueue<IN>(), delegate, workerCount);
    }

    public WorkerThreadPipeDecorator(BlockingQueue<IN> workQueue,
                                    Pipe<IN, OUT> delegate, int workerCount) {
        if (workerCount <= 0) {
            throw new IllegalArgumentException("workerCount should be positive!");
        }

        this.workQueue = workQueue;
        this.delegate = delegate;
        for (int i = 0; i < workerCount; i++) {
            workerThreads.add(new AbstractTerminatableThread(terminationToken) {
                @Override
                protected void doRun() throws Exception {
                    try {
                        dispatch();
                    } finally {
                        terminationToken.reservations.decrementAndGet();
                    }
                }
            });
        }
    }

    protected void dispatch() throws InterruptedException {
        IN input = workQueue.take();
        delegate.process(input);
    }

    @Override
    public void init(PipeContext pipeCtx) {
        delegate.init(pipeCtx);
        for (AbstractTerminatableThread thread : workerThreads) {
            thread.start();
        }
    }

    @Override
    public void process(IN input) throws InterruptedException {
        workQueue.put(input);
        terminationToken.reservations.incrementAndGet();
    }
}
```

```

        @Override
        public void shutdown(long timeout, TimeUnit unit) {
            for (AbstractTerminatableThread thread : workerThreads) {
                thread.terminate();
                try {
                    thread.join(TimeUnit.MILLISECONDS.convert(timeout, unit));
                } catch (InterruptedException e) {
                }
            }
            delegate.shutdown(timeout, unit);
        }

        @Override
        public void setNextPipe(Pipe<?, ?> nextPipe) {
            delegate.setNextPipe(nextPipe);
        }
    }
}

PIPE线程池
public class ThreadPoolPipeDecorator<IN, OUT> implements Pipe<IN, OUT> {
    private final Pipe<IN, OUT> delegate;
    private final ExecutorService executorService;

    //线程池停止标志。
    private final TerminationToken terminationToken;
    private final CountDownLatch stageProcessDoneLatch = new CountDownLatch(1);

    public ThreadPoolPipeDecorator(Pipe<IN, OUT> delegate,
                                   ExecutorService executorService) {
        this.delegate = delegate;
        this.executorService = executorService;
        this.terminationToken = TerminationToken.newInstance(executorService);
    }

    @Override
    public void init(PipeContext pipeCtx) {
        delegate.init(pipeCtx);
    }

    @Override
    public void process(final IN input) throws InterruptedException {
        Runnable task = new Runnable() {
            @Override
            public void run() {
                int remainingReservations = -1;
                try {
                    delegate.process(input);
                } catch (InterruptedException e) {
                }
            } finally {
                remainingReservations = terminationToken.reservations
                        .decrementAndGet();
            }
            if (terminationToken.isToShutdown() && 0 == remainingReservations) {
                stageProcessDoneLatch.countDown();
            }
        };
        executorService.submit(task);
        terminationToken.reservations.incrementAndGet();
    }

    @Override
    public void shutdown(long timeout, TimeUnit unit) {
        terminationToken.setIsToShutdown();

        if (terminationToken.reservations.get() > 0) {
            try {
                if (stageProcessDoneLatch.getCount() > 0) {
                    stageProcessDoneLatch.await(timeout, unit);
                }
            } catch (InterruptedException e) {
            }
        }
        delegate.shutdown(timeout, unit);
    }

    @Override
    public void setNextPipe(Pipe<?, ?> nextPipe) {
        delegate.setNextPipe(nextPipe);
    }

    /**
     * 线程池停止标志。
     * 每个ExecutorService实例对应唯一的一个TerminationToken实例。
     * 这里使用了Two-phase Termination模式（第5章）的思想来停止多个Pipe实例所共用的
     * 线程池实例。
     * @author Viscent Huang
     */
    private static class TerminationToken extends
        io.github.viscent.mtpattern.tpt.TerminationToken {
        private final static ConcurrentHashMap<ExecutorService, TerminationToken>
            INSTANCES_MAP = new ConcurrentHashMap<ExecutorService, TerminationToken>();
    }
}

```

```

// 私有构造器
private TerminationToken() {
}

void setIsToShutdown() {
    this.toShutdown = true;
}

static TerminationToken newInstance(ExecutorService executorService) {
    TerminationToken token = INSTANCES_MAP.get(executorService);
    if (null == token) {
        token = new TerminationToken();
        TerminationToken existingToken = INSTANCES_MAP.putIfAbsent(
            executorService, token);
        if (null != existingToken) {
            token = existingToken;
        }
    }
    return token;
}
}

抽象Pipe public abstract class AbstractParallelPipe<IN, OUT, V> extends
    AbstractPipe<IN, OUT> {
    private final ExecutorService executorService;

    public AbstractParallelPipe(BlockingQueue<IN> queue,
        ExecutorService executorService) {
        super();
        this.executorService = executorService;
    }

    /**
     * 留给子类实现。用于根据指定的输入元素input构造一组子任务。
     *
     * @param input
     *      输入元素
     * @param <V>
     *      子任务的处理结果类型
     * @return 一组子任务
     */
    protected abstract List<Callable<V>> buildTasks(IN input) throws Exception;

    /**
     * 留给子类实现。对各个子任务的处理结果进行合并，形成相应输入元素的输出结果。
     *
     * @param subTaskResults
     *      子任务处理结果列表
     * @return 相应输入元素的处理结果
     * @throws Exception
     */
    protected abstract OUT combineResults(List<Future<V>> subTaskResults)
        throws Exception;

    /**
     * 以并行的方式执行一组子任务。
     *
     * @param tasks
     *      一组子任务
     * @return 一组可以借以获取并行任务中各个任务处理结果的Promise（参见第6章，Promise模式）
     * 实例。
     */
    protected List<Future<V>> invokeParallel(List<Callable<V>> tasks)
        throws Exception {
        return executorService.invokeAll(tasks);
    }

@Override
protected OUT doProcess(final IN input) throws PipeException {
    OUT out = null;
    try {
        out = combineResults(invokeParallel(buildTasks(input)));
    } catch (Exception e) {
        throw new PipeException(this, input, "Task failed", e);
    }
    return out;
}

}

PipeLine * @author Viscent Huang
*
* @param <IN>
*      输入类型
* @param <OUT>
*      输出类型
*/
public interface Pipeline<IN, OUT> extends Pipe<IN, OUT> {

    /**
     * 往该Pipeline实例中添加一个Pipe实例。
     *
     * @param pipe
     *      Pipe实例
     */
    void addPipe(Pipe<?, ?> pipe);
}

SimplePip public class SimplePipeline<T, OUT> extends AbstractPipe<T, OUT> implements
    Pipeline<T, OUT> {

    private final Queue<Pipe<?, ?>> pipes = new LinkedList<Pipe<?, ?>>();

    private final ExecutorService helperExecutor;

    public SimplePipeline() {
        this(Executors.newSingleThreadExecutor(new ThreadFactory() {
            @Override
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r, "SimplePipeline-Helper");
                t.setDaemon(true);
            }
        }));
    }
}

```

```

        return t;
    }

}));


}

public SimplePipeline(final ExecutorService helperExecutor) {
    super();
    this.helperExecutor = helperExecutor;
}

@Override
public void shutdown(long timeout, TimeUnit unit) {
    Pipe<?, ?> pipe;
    while (null != (pipe = pipes.poll())) {
        pipe.shutdown(timeout, unit);
    }
    helperExecutor.shutdown();
}

@Override
protected OUT doProcess(T input) throws PipeException {
    // 什么也不做
    return null;
}

@Override
public void addPipe(Pipe<?, ?> pipe) {
    // Pipe间的关联关系在init方法中建立
    pipes.add(pipe);
}

public <INPUT, OUTPUT> void addAsWorkerThreadBasedPipe(
    Pipe<INPUT, OUTPUT> delegate, int workerCount) {
    addPipe(new WorkerThreadPipeDecorator<INPUT, OUTPUT>(delegate,
    workerCount));
}

public <INPUT, OUTPUT> void addAsThreadPoolBasedPipe(
    Pipe<INPUT, OUTPUT> delegate, ExecutorService executorService) {
    addPipe(new ThreadPoolPipeDecorator<INPUT, OUTPUT>(delegate,
    executorService));
}

@Override
public void process(T input) throws InterruptedException {
    @SuppressWarnings("unchecked")
    Pipe<T, ?> firstPipe = (Pipe<T, ?>) pipes.peek();
    firstPipe.process(input);
}

@Override
public void init(final PipeContext ctx) {
    LinkedList<Pipe<?, ?>> pipesList = (LinkedList<Pipe<?, ?>>) pipes;
    Pipe<?, ?> prevPipe = this;
    for (Pipe<?, ?> pipe : pipesList) {
        prevPipe.setNextPipe(pipe);
        prevPipe = pipe;
    }

    Runnable task = new Runnable() {
        @Override
        public void run() {
            for (Pipe<?, ?> pipe : pipes) {
                pipe.init(ctx);
            }
        }
    };
    helperExecutor.submit(task);
}

public PipeContext newDefaultPipeContext() {
    return new PipeContext() {
        @Override
        public void handleError(final PipeException exp) {
            helperExecutor.submit(new Runnable() {
                @Override
                public void run() {
                    exp.printStackTrace();
                }
            });
        }
    };
}
}

PipeContext
public interface PipeContext {
    /**
     * 用于对处理阶段抛出的异常进行处理.
     *
     * @param exp
     */
    void handleError(PipeException exp);
}

```

```

PipeException public class PipeException extends Exception {
    private static final long serialVersionUID = -2944728968269016114L;
    /**
     * 抛出异常的Pipe实例。
     */
    public final Pipe<?, ?> sourcePipe;
    /**
     * 抛出异常的Pipe实例在抛出异常时所处理的输入元素。
     */
    public final Object input;

    public PipeException(Pipe<?, ?> sourcePipe, Object input, String message) {
        super(message);
        this.sourcePipe = sourcePipe;
        this.input = input;
    }

    public PipeException(Pipe<?, ?> sourcePipe, Object input, String message,
        Throwable cause) {
        super(message, cause);
        this.sourcePipe = sourcePipe;
        this.input = input;
    }
}

```

使用上述代码

利用本章的可复用代码实现Pipeline模式，应用程序只需要完成以下几件事情。

1. 【必需】创建Pipeline实例。

2. 【必需】创建表示各个处理阶段的Pipe实例，并将其添加到Pipeline实例中。

3. 【必需】初始化Pipeline实例。

例子

背景 某系统需要一个数据同步的定时任务。该定时任务将数据库中符合指定条件的记录数据以文件的形式FTP传输（同步）到指定的主机上。该定时任务需要满足以下几个要求。

1. 每个数据文件最多只包含N（如10000，具体可配置）条记录；当一个数据文件被写满时，其他待写记录会被写入新的数据文件。
2. 每个数据文件可能需要被传输到多台主机上。
3. 本地要保留同步过的数据文件的备份。

因此，该定时任务所做的事情主要包括：查询数据库（以下称为Stage1）、根据数据库查询结果集生成本地数据文件（以下称为Stage2）、FTP传输各个数据文件到指定的主机（支持多台主机，以下称为Stage3）、备份传输完毕（或者失败）的数据文件（以下称为Stage4）。显然，该任务处理涉及比较多的I/O操作：查询数据库和传输数据文件均涉及网络I/O和文件I/O，备份传输过的文件涉及文件I/O。所以，该任务的处理逻辑不适宜用单线程实现。因为采用单线程模型上述处理步骤只能是按顺序串行执行，而这会使各个处理步骤所涉及的I/O等待的负面影响放大。另外，如果采用多个线程，每个线程中仍然是按顺序串行处理（每个线程先后执行Stage2、Stage3和Stage4）也是不适合的，这样会导致执行Stage2时多个线程需要征用同一个数据文件（因为这个文件还没有被写满）。

这里，Pipeline模式可以派上用场：采用一个线程去负责Stage1的执行。其余处理步骤（Stage2、Stage3和Stage4）中的每一个步骤都有专门的工作者线程去负责处理。这样，从个体任务上看，上述处理步骤虽然仍然是按顺序串行处理，但从整体任务上看几个步骤却是并行的，从而提升了计算效率。该定时任务的代码如清单13-1所示。

用到的Pipe

Pipe 实例名称	输入	处理	输出	线程模型
stageSaveFile	一组数据库记录（类型为RecordSaveTask）	将输入的一组数据库记录存入数据文件	若当前数据文件已写满，则输出该写满的文件（类型为File）；否则，输出null	单工作者线程。若使用多线程模型，必然导致多个线程征用同一个数据文件（文件未写满），而这又需要引入锁。使用单线程模型则可以避免锁的使用
stageTransferFile	已写满的数据文件（类型File）	将输入的数据文件以并行方式FTP 传输至指定的多台FTP 主机	若文件传输成功，则输出已传输过的数据文件（类型为File）；否则输出null	单工作者线程。该 Pipe 处理过程会用到一款非线程安全的开源FTP 客户端组件，使用单线程模型可以在不引入锁的情况下确保使用该组件时的线程安全
stageBackupFile	已传输成功的文件（类型File）	将输入的文件移动到备份目录	null	单工作者线程

代码 电子书 3160

使用 Microsoft OneNote for Mac 创建。