微内核工作流引擎
体系结构与部分解决方案参考

The Reference of Micro Kernel Workflow Engine Architecture and Some Solutions

版本：1.0

作者 ：胡长城 [银狐 999]
http://www.javafox.org
http://www.wfchina.org
http://blog.csdn.net/james999
完成日期：2005-7-24 version 1.0
联系信箱：james-fly@vip.sina.com
MSN ：fcxiao2000@hotmail.com

我们需要更多的原创……

更多工作流参考文档，请访问在 http://www.javafox.org

目录

1. 阅读本文的基础知识

本篇并不适合工作流的初学者，在阅读本篇之前，请您花一定的时间，弄懂下面所说的一些知识点，这样有助于您更加深刻的理解本文的内容。

首先您需要读懂 WfMC 的《工作流参考模型》。 其次，您需要有过一定的软件开发经验，对本文中所提到的一些应用模式能够迅速的理解，如：事件监听机制，状态机等等。

当然，如果您曾用过或了解 jbpm,osworkflow,shark,obe 这几个开源引擎，则更加有助

于明白本篇所谈及到的一些处理方式。

如果您能够对工作流常用应用需求能够有所了解，则能够更加深刻的理解，通用引擎体系的架构和用意

2. 前言

到目前为止，国内还没有一篇文章讲解引擎框架的。所以很多人对“引擎”很难有一个“结构上的认识”。一谈论到工作流引擎，很多人的脑海只有一个概念：哦，那是一个就像发动机样的东西。

虽然说引擎的实现方式是五花八门，但是从引擎的框架和内部层次角度来看，还是有很多共性的东西。今天我们就拿“微内核引擎”的通用体系框架来给大家做个讲解。

我想这篇的文章，对大家，不论是开发工作流引擎，还是应用工作流，都会有很多帮助。这也是这几年来，我第一次从一个“引擎的实现角度”来讲解工作流问题。早先我写的那些《工作流模型分析》《工作流授权控制模型》《工作流系统中的组织模型应用解决方案》《工作流系统功能列表系列》，以及《工作流之星光》等等文章，都主要是站在“理论、模型、原理、功能”的角度来讲解工作流一些普及知识。

说到这儿，插入几句额外的话：即使上次写的《工作流引擎核心调度算法与 PetriNet》也仅仅是分析了几种开源引擎的“调度算法”，虽然姑且也算是与实现接轨，但是估计能读懂得少之又少。**在那篇《工作流引擎核心调度算法与 PetriNet》的文章中，我隐去了有关“引擎内核组成”的内容，这部分内容只在我后来所提供的工作流培训中有所讲解**。在下面的文章中，我会多次提到“Engine Kernel”这一部分，至于 Engine Kernel 到底包含什么、如何调度、组成。本篇文章也不会谈及。这一部分内容将在我年底准备写那篇《工作流引擎内核揭秘》中和盘托出。

3. 我的工作流探索之路

在进入本文章正文之前，让我姑且浪费点笔墨来说说我的“工作流探索之路”，**虽然这不是一个述说经历的场合，但是希望通过分析我所经历的一些过程，能够让大家明白，在探索工作流的时候，需要积累些什么**，才能够真正明白这篇所介绍的“引擎架构层次”。——我不太希望明明确确长篇累牍地去写文字，告诉人们应该学习什么，那并不是一个 Preacher 所应该做的事情。希望有兴趣而又有探索钻研精神的开发者们，从这些含糊的文字中，自己摸索出什么—— 明晰而又豁朗的文字，将会在我后续的《工作流之魂》中详细介绍。

三年前我也是那么想的“哦，那是一个就像发动机样的东西”，三年后我已经很明白了，那么你呢？如果你现在刚刚接触工作流，而你又看到了这篇文档，那么你是个很幸运的人了——恭喜你，你将少走很多弯路。

是的，三年前我也是那样想的，那时候我已经是 JavaUnion 的总版主了。我不再一味只去

钻研干巴巴的 Java 技术（毕竟技术终究是技术），一头扎进了这个独特的圈子，一扎就是三年多。

最初的一年内，我几乎没有在理论方面去探究什么。那时候似乎也不能说是完全的工作流，用现在的话说，应该叫“审批流”（一套 OA 产品中的流程处理，2002 年的时候，能够支撑流程的 J2EE 的 OA 产品并不多，想来，我还是挺幸运的）。那时候没有什么可以求助的人，也没有什么可参考的文档，也主要是跟着公司内的老一辈们摸索着。

这一年虽然对工作流并没有积累多么有价值的经验（至少我们当时做得那个产品还没有 Engine 这个概念，内部的实现也过分依据于数据库的处理），但是对审批流的应用需求、模型、权限需求等方面却收获不小。这可以从我早期写的《工作流模型分析》和《工作流权限控制模型》中略窥一二。

中途有过半年去做了 Data Integration 方面的项目实施和开发。这领域虽然看似跟工作流没有任何关系，但是其很多有关“数据传输”的思想，却被整入了后来的很多工作流产品。有空大家看看最早期的支持流程的 EAI 产品，或者如今比较火爆的 BPM，都或多或少有这方面的影子。

这半年中，我还做了一些重要的事情：一是反思前一年做得公文流转产品；二是把 OBE 的源码翻了又翻（可惜那时候有很多朦胧不明白的地方）。—— 说到这里，也有必要感谢一下 “浆糊”和“踏冰”兄弟早期的的努力，他们于 2002 翻译的《工作流参考模型》和《XPDL》帮助了很多人。

半年之后我毅然返回工作流这个领域，并且也正是在那个时候真正开始从理论、模型角度来研究工作流。

然而，那时候国内工作流的可交流人太少了，自己坑坑洼洼的摸索，留下了很多朦胧的地方。带着这些的朦胧，后来主持过一款工作流的研发，可惜最终以失败告终（失败的原因是多方面的，但想来，那些朦胧之处，也成了研发中的瓶颈）。 虽然最终失败了，但是却也留下了很多探索中的成果，最明显的要数那篇《工作流系统中的组织模型应用解决方案》，从那篇文档中，你也许可以感到，当时我已经努力去多研究和接触现有的抽象模型和思想。—— 当然这样的研究对我以后的实施和开发工作流产品，起到很多基础性帮助和指导性作用。

但那时候，也隐约感觉“工作流”这个模型中还存在一个我还未探究的领域，但很可遗憾自己那时候却一直没有想通。后来有幸去了 Justep，在那两个多月的时间里，将很多精力投入到对“Process Methodology”研究中，才终于解决了自己的那个疑问。—— 从这一点上说，我挺佩服老宋（Justep CTO）的，至少我去研究“Process Methodology”，很大程度上被其逼的——至少在刚开始的时候，我总觉得像“Petri Net 之类的咚咚，对流程来说还是一个空洞的理论”。（可见那时候我得想法还是多么的幼稚，当然，很庆幸的是，我现在已经不这么认为了）

当时 Justep Biz5.0 刚刚开始研发，为了不泄密，所以没有没有写过什么专门的文档来阐述当时的收获，但是倒是写了很多零散的收获放在我的 Blog 上。

离开 Justep 之后，来到了现在的公司，又重构了一款流程产品。至少这时候自我感觉已经“积累的很敦实了”。很庆幸在这里碰到一位我很佩服的哥们“切尔斯基”。他在模式方面扎实的基本功，为这边的引擎奠定了一个很不错的基础。遗憾的是，我来后没有多久他就离开了公司，为此写了篇《送同仁别》。

如今，这边的引擎已经重构完了。而我基本上也将不再涉足 workflow 开发，主要把精力和心思转移到其它领域。如今写这篇文档，姑且算是前一段重构引擎的总结。也希望这篇文档，能够让更多的人“真真正正的理解工作流引擎”。

4. 为什么要引入微内核

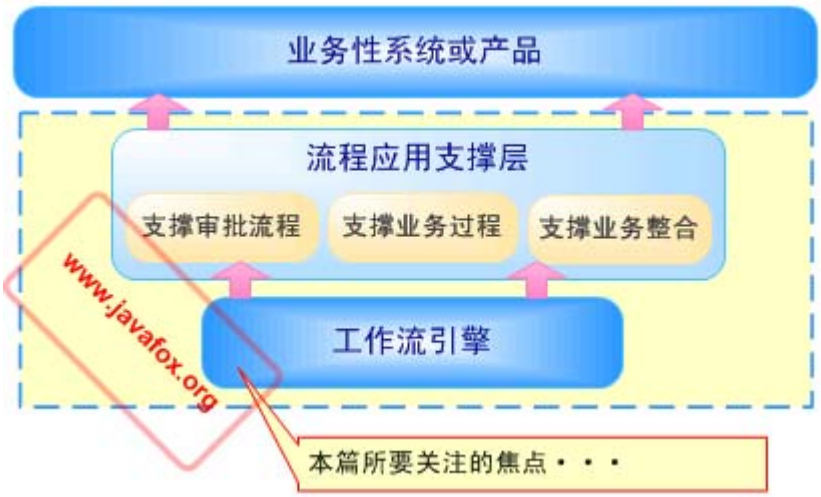
为什么要引入微内核引擎这个概念，大的道理就不说了，说点自己的感悟吧。前段时间重构这边的引擎，由于早期引擎内部与外部基本上娇柔在一起，弄得我重构之路上疲惫不堪，最大的问题是任何地方的修改都很容易“牵一发而动全身”。最终造成虽然引擎这边要重构的内容很多，但是基本上别人都没办法修改，基本上都落在我一人身上。最终我不得不首先把引擎所有部分全部打散，重新抽象和组装。

经过两轮的重构，这边的引擎逐渐在朝着“微内核”趋势发展。这样目前很多引擎所依赖的服务和组件，即使别人不懂引擎这一块构造，但是依据扩展接口和规则，可以较为容易的扩展和修改一些实现类。

其实微内核的思想，就是“降低组件耦合度”的延伸。当然后来的 IoC 发展也为“微内核”实现提供很好的支撑。当然微内核更加讲究的是一个“体系上、全局上”的设计思路。

5. 工作流引擎的位置

本篇的主旨是讲“引擎的内部架构体系”，微内核只是一种更加良性、易扩展的方式。但是在讲解引擎之前，先让我们给工作流引擎摆正一个位置：



所以平常我们所说的那些审批流产品，并不是真正的“工作流”，那只是在工作流之上一层的产物。好的审批流产品会内嵌“工作流引擎”负责流程的调度维护。当然也不得不说，部分国内的审批流产品，其内部实现与“工作流”是没有多大关系的。

6. 微内核工作流引擎的结构概述

微内核引擎的原则，就是尽量将“引擎所需的服务”与“引擎内核调度计算”剥离。将 Engine Kernel 部分尽量“疏剪”，尽可能将“一些处理操作”放置于外围扩展。我们纵观目前几个比较火爆的引擎：osworkflow，jBpm 等等，你会发现它们的内核部分都非常的简单，而是将大量的精力用于处理扩展上。

6.1. 七个层次划分

通常情况下，一个微内核工作流引擎（或者说，大部分的工作流引擎），都或多或少包含如下七个层次：

（1） **外设层**：这一层是不属于引擎，而是引擎与外界交互的一个扩展层。比如我们利用这一层，可以让外界通过 JMS/Web Service 等等方式与引擎交互。

（2） **对外接口层**：这是我们通常所说的 WAPI。

（3） **交互代理**：WAPI 如何获取与引擎的通信，由这一层负责解决

（4） **引擎内核处理**：这就是引擎真正的 Kernel 了，在一层最主要解决的问题就是：活动之间的调度

（5） **引擎运行服务层**：为引擎提供服务和资源

（6） **扩展实现层**：这一层是微内核最为繁琐地方，有各种扩展点供外围扩展。总结下来主要有三类：

a) 支撑流程运行的（Sustentive）：这些扩展点必须被实现，否则引擎无法运行。比如为引擎提供存储适配。

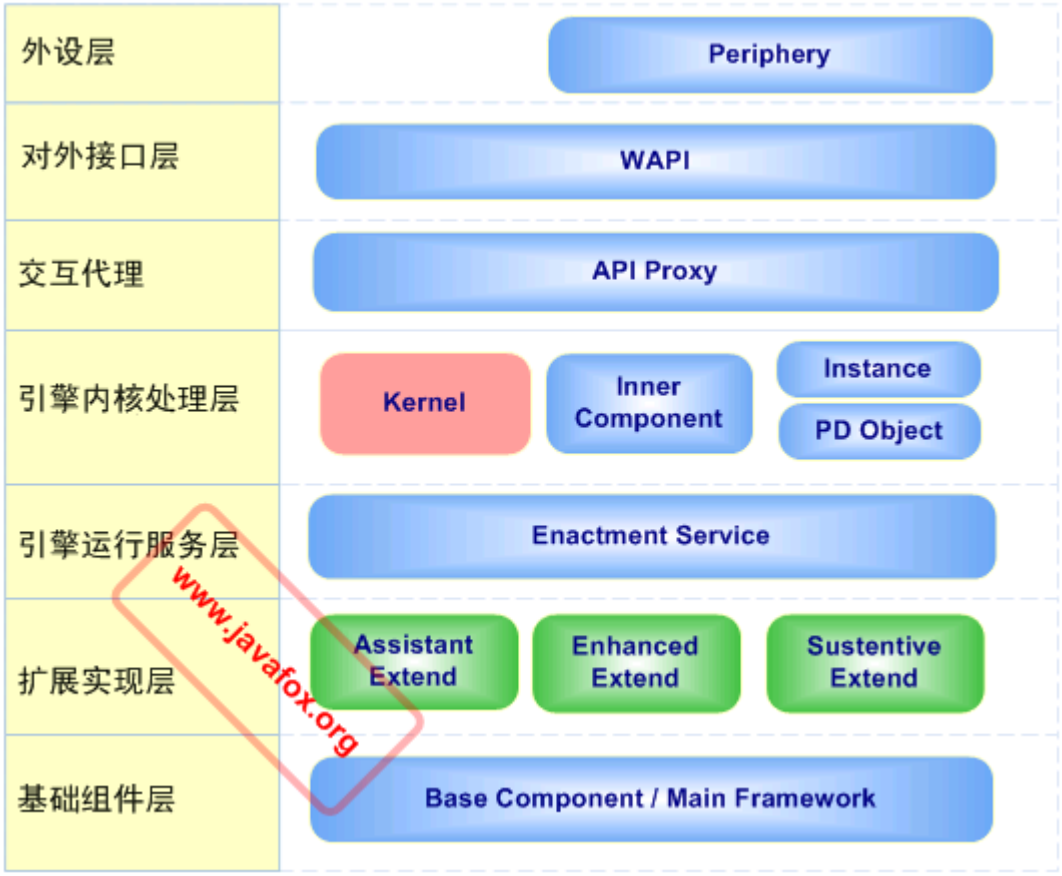
b) 辅助流程运行的（Assistant）：利用这些扩展点，可以让引擎进行更多的处理，比如一些 Function 接口、条件处理接口。

c) 增强流程运行的（Enactment）：利用这些扩展点，可以让引擎的内能变得更强大和完善，比如一些事件监听处理接口，客户自定义的策略实现等。

（7） **基础组建层**：为引擎运行提供一些基本的公共组件，比如脚本计算、时间计算等等

基本上所有的微内核引擎都近似这个层次来划分。经过不同层次的剥离，每一层次都有自己的职责和定位，从而让引擎职责变得更加清晰和简单。

下面让我们来看看这划分的层次图：

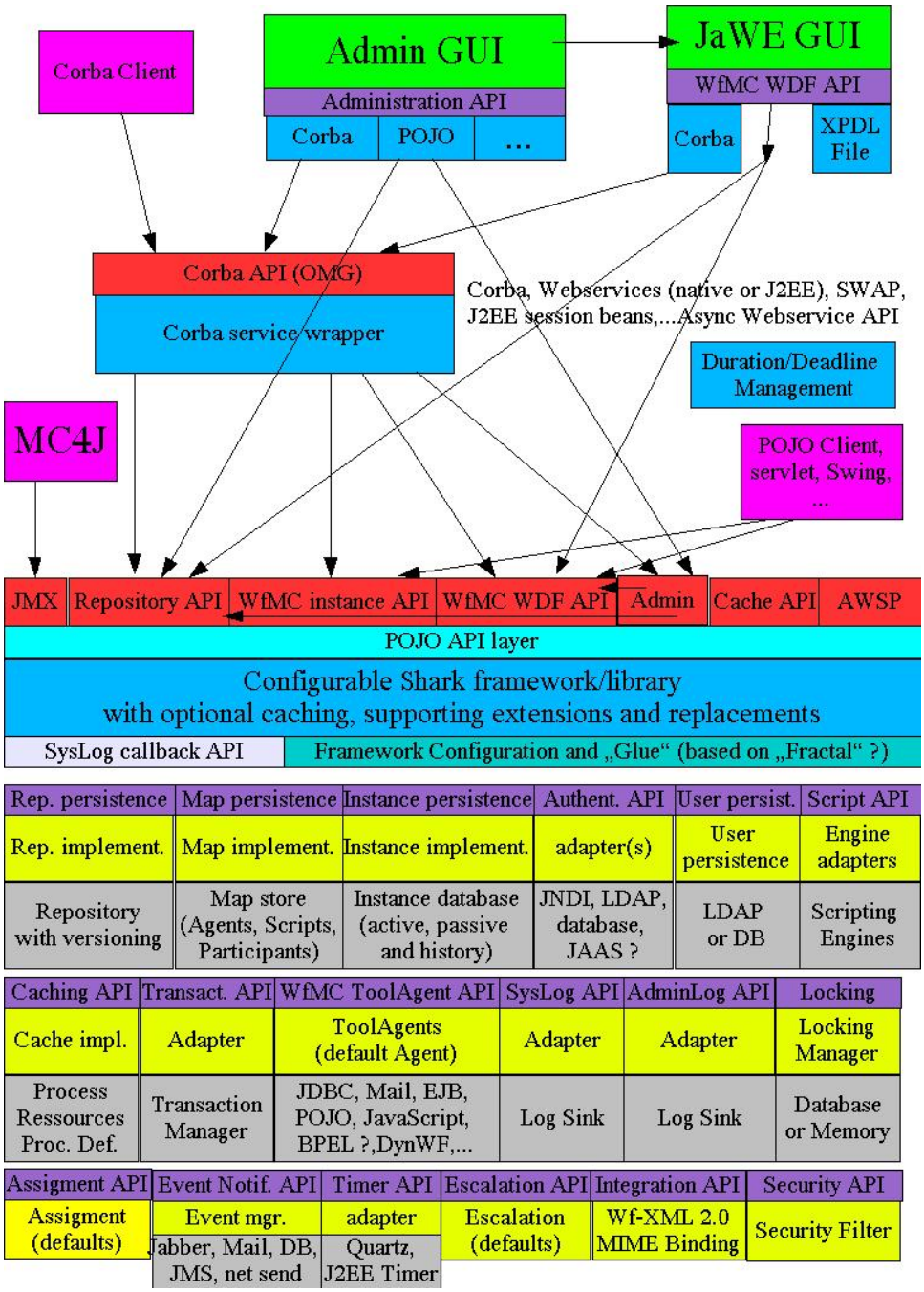


其中图中“红色”模块则代表“引擎内核（Engine Kernel）”。而图中“绿色”部分，则是对微内核引擎来说，代码量最多的地方。一个微内核引擎，大部分的代码都基本上放在了这一层。

在我们逐一详细讲解这个七个层次的内容的之前，先让我们来看看开源引擎的结构，看看是不是跟这个层次类似。

6.2. Shark 的架构简介

下面这张图，来自 Shark 的官方网站，也是 Shark 所公布的官方架构图。可惜的是，这张图并没有标明“引擎内核”所处的位置，而主要标明了各个对外扩展的组件的接口和实现。



详细有关 Shark 体系结构的介绍就不说了，有兴趣的可以去 http://shark.objectweb.org/doc/overview.html 读读 Shark 的官方帮助文档。但我们有必要简单说一说 Shark 的 Component。

Shark 主要有如下几种组件：

a、**Kernels**：引擎内核层

b、**Plugins**：包含了一些事件监听、存储服务等。内容属于了我所说的“服务层”和“扩展层”

c、**Wrappers**：对外包装层，类似是我所说的“外设层”

d、**Tool Agents**：一些应用程序的处理，内容基本上属于我所说的“扩展层”

e、**Applications**：这个应用层，不属于我所说的引擎范畴。其在 shark 中主要是一些

“流程设计器”、“web 应用”等外围的工作流组件。

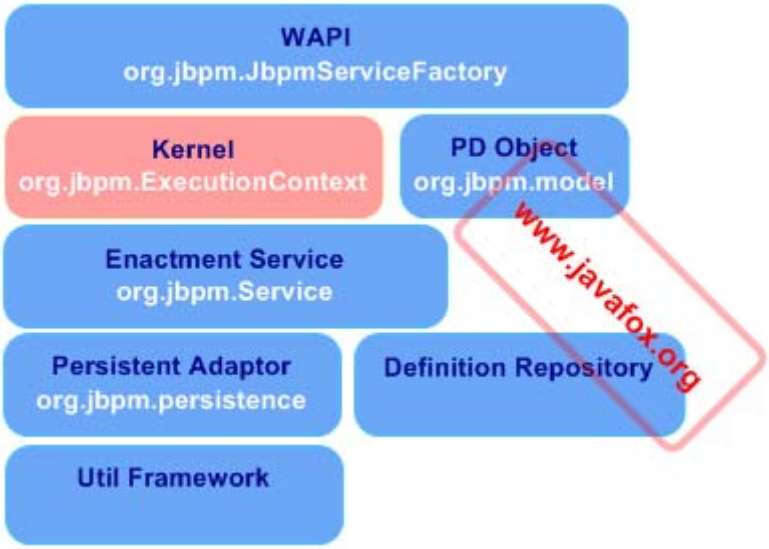
有关这几个组件的详细信息，可以访问

http://shark.objectweb.org/components.html 。

6.3. jBpm 的结构简介

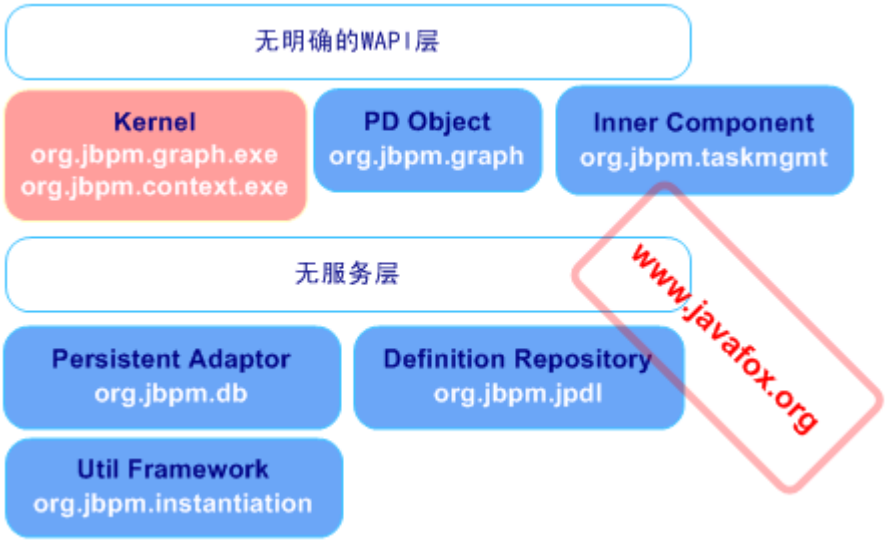
对于 jBpm 来说，其结构就不如 shark 那么体系完善。在早期的 jBpm2.0 中还能够窥探一些服务的影子，而到了 jBpm3.0，则其结构被更加的精简化。

让我们首先来看看 jBpm2.0 的结构，其早期的设计思路“以服务为主导的”，结构上也基本朝向“微内核”发展的：



但是到了 jbpm3，受 Jboss 所控制后，整个结构进行了调整，将早期 jbpm2 种的服务体系彻底的抛弃了。这样的接口，让引擎内核的对象“职责泛化”，有些趋向于“Domain Model”的思想。

但是从 jbpm3 种我们还是可以分析出几个层次的：



从本质上说，jBpm3.0 已经不能算是“纯正的微内核工作流引擎”，但当然依然是嵌入式的。

纵观几个比较有名的开源引擎，我建议大家还是首先从 OBE 和 Shark 学起（OBE 的一些结构会在本篇文档后续内容逐渐介绍），首先这两个引擎的体系结构比较清晰和完备，其次者两个工作流所支撑的“概念体系”都是遵循 WfMC 规范的，对于初入门的工作流开发者来说，理解清楚一些基本的概念是非常重要的。至于 OSWorkflow 和 jBpm，虽然灵活，也有很多技巧，但是对于初学工作流而又准备研究工作流的开发者来说，刚开始还是不要碰为好（当然，以后是要好好研究研究的）。—— 当然，如果你仅仅只是为了应付客户或者上头的项目需求，抑或是满足自己的一点点对工作流的好奇心，那么刚开始去弄弄也无妨。

7. 微内核工作流引擎的七个层次介绍

微内核引擎的原则，就是尽量将“引擎所需的服务”与“引擎内核调度计算”剥离。将 Engine Kernel 部分尽量“疏剪”，尽可能将“一些处理操作”放置于外围扩展。我们纵观目前几个比较火爆的引擎：osworkflow，jBpm 等等，你会发现它们的内核部分都非常的简单，而是将大量的精力用于处理扩展上。

7.1. 外设层

既然是外设，其就是对引擎来说是“可有可无的外围设施（组件）”。这些外围组件会扩充引擎外围的能力，但本身不会提高引擎的性能、能力。—— 当然，站在另外一个角度来看待，这些外设也可以算是“引擎”的一个“客户”。

比如我们需要让引擎支持 JMS 或 Web Service，为引擎提供对移动设施的支持等等。这些外设可以让引擎的变得更加“绚丽”。事实上，通常我们所接触的工作流 web 应用，对工作流引擎来讲，就是一个“外设”。

7.2. 接口层

这也就是我们通常所说的“WAPI（Workflow Application Interface）”。说到这，我倒是建议大家有空多读读 WfMC 的《工作流参考模型（Workflow Reference Model）》。

当然现实应用中的 WAPI，要原比 WfMC 所提供的参考 WAPI 要丰富很多，几乎每一种引擎都有一堆自己的接口，也有一套自己的诠释方式。但是，不论 WAPI 多么丰富，也不论 WAPI 多么简单，它们迥异的身躯，却都拥有者一个共同的目的：提供与流程引擎（或者说流程服务器）的会话交互。外界通过这些接口来与引擎进行交互，对流程进行操纵、管理。

7.3. 交互代理层

这一层的目的只有一个，就是如何在“引擎”与“接口”建立通讯。而这个通讯对访问客户是透明的。比如客户无须关心“引擎在什么地方”，只需要通过这个代理层获取与引擎的通信即可。

比如 Shark 这个引擎，其有基于 Corba 的 WAPI，如何去获取这些 WAPI 及与引擎进行通讯呢？

```
/* Shark 的 Corba 客户端调用事例 */
CORBAProcStartClient cpsc=new CORBAProcStartClient(args[0]);
cpsc.startProcess(pkgId,pDefId,cntx);
```

在这样调用的时候，我们并不关心引擎在什么地方，采用什么通讯协议。只关心这些 WAPI 怎么调用。

7.4. 引擎内核处理层

这一层是引擎的内核关键层，其主要负责依据调度算法，调度流程的运转。这一层主要包含：“Engine Kernel”，相关的一些资源对象，相关的一些内部处理组件。

Engine Kernel 不是本篇介绍的范畴，这一部分我将在后续的《引擎内核揭秘》中详细书写。虽然各家的引擎五花八门，但是在 Engine Kernel 这一部分，其基本元素和逻辑关系都基本相似的。有关 Engine Kernel 中的调度算法，可以参考我写的《工作流引擎核心调度算法与 Petri Net》一文。

引擎在调度计算的时候，是需要一些实例对象和资源的。比如“流程实例（Process Instance）”和一些环境资源对象（ExecutionContext）等等，当然也需要一些负载数据的对象 —— 引擎的运转方向很大程度上有赖于这些数据。

至于所说的内部处理组件，是一个比较笼统地说法。各个引擎会因为自身的结构、功能等不同而不同。

7.5. 引擎运行服务层

这一层主要是为引擎内核的计算调度提供服务，这些服务主要包含：流程定义解析服务、流程实例存储服务、参与者解析服务、脚本计算服务、事件监听服务等等。Enactment Service 这一层会因引擎结构、功能的不同而不同。有的引擎体系比较庞大，相应的服务也比较多；当然有些引擎这一层很薄弱，甚至被逐渐淡化：比如 jbpm 这个开源引擎，在 jbpm2 的时候，这一层还是很丰富的，但到了 jbpm3，基本上已经没有这一层的踪影了。很多原本的服务，都被淡化成引擎内核对象对组件的直接访问。

有必要说明的一点是，这些服务很多程度上都是依赖于“扩展实现层”中的组件或适配器。

7.6. 扩展实现层

引擎主要的很多功能和资源都依赖于扩展实现层提供真正的实现。对于引擎来说，也主要有三种扩展：支撑流程运行的（Sustentive），辅助流程运行的（Assistant），增强流程运行的（Enactment）。

前面已经简要说了这三种扩展的定位和用途，下面我们来分析这一种扩展都大致包含些什么：

7.6.1. 支撑流程运行的（Sustentive）

这一种扩展对于引擎来说，必须有实现，否则引擎不能正常运行。

通常来说，属于这种扩展的主要有以下几种：

（1） 组织模型适配

（2） 流程实例存储服务

（3） 应用适配

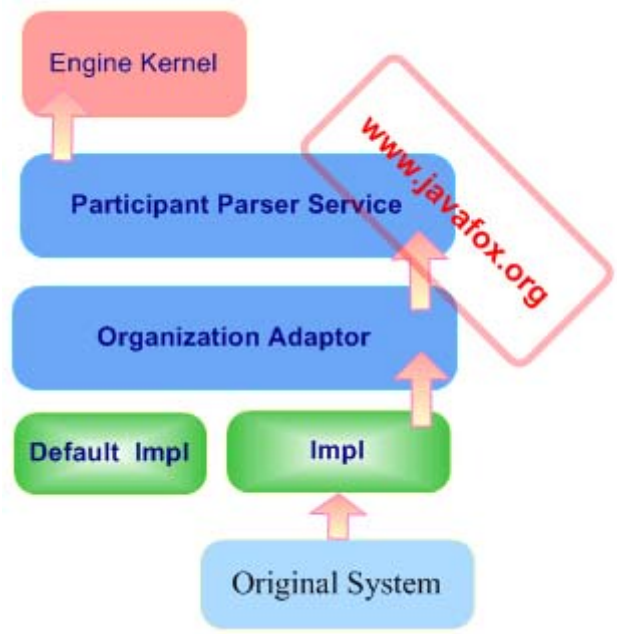
（4） 流程定义操纵和扩展

当然还有也姑且算是这种，比如“任务分配处理”等等，特别是很多嵌入式开源引擎，其本身没有提供组织模型的适配，但是也为了支持针对手工任务的分配，提供了相应的处理接口

下面主要详细说一下其中三个：组织模型（包括任务分配处理）、流程实例存储、应用适配

7.6.1.1. 组织模型适配

对于嵌入式工作流引擎来说，如何解决对“组织模型”的支持是一个很头痛的问题。各种系统都回涉及到组织模型和权限。大多商用的嵌入式工作流引擎都会提供对通用组织模型的“组织模型适配器”解决方案；



有关 Participant Parser Servic 与 Organization Adaptor 的接口内容，不是本篇的重点。我会在后续的《工作流系统中组织模型应用解决方案和实现 2005 版》中详细叙述。（这里我采用了 WfMC 概念中的 Participant 概念）

但是开源引擎很少采用这种方式，而且基本上都无一例外的屏蔽了这一层，只提供最为简单的“任务分配处理接口”，根据相应的系统，特定的实现。

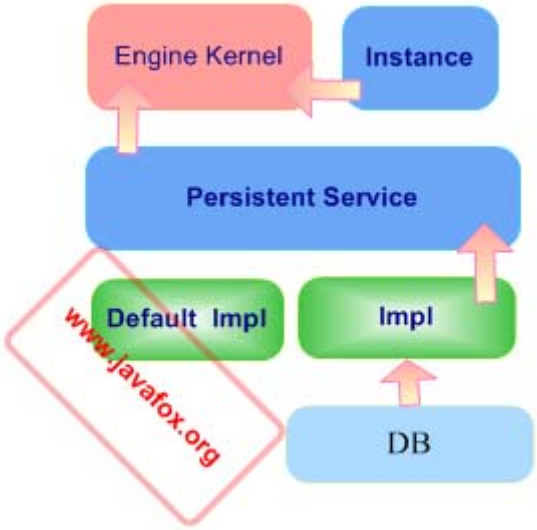
下面我们简单来看看 jBpm 这个开源引擎是如何处理这一层的：

针对人工型处理（在 jBpm 是 Task-Node 节点），你必须实现 org.jbpm.taskmgmt.def. AssignmentHandler 这个接口。

```
/*以下是 jBpm 任务分配的扩展配置*/
<process-definition name="the baby process">
  <start-state>
    <transition name="baby cries" to="t" />
  </start-state>
  <task-node name="t">
    <task name="change nappy">
      <assignment class="org.jbpm.tutorial.taskmgmt.NappyAssignmentHandler" />
    </task>
    <transition to="end" />
  </task-node>
  <end-state name="end" />
</process-definition>
```

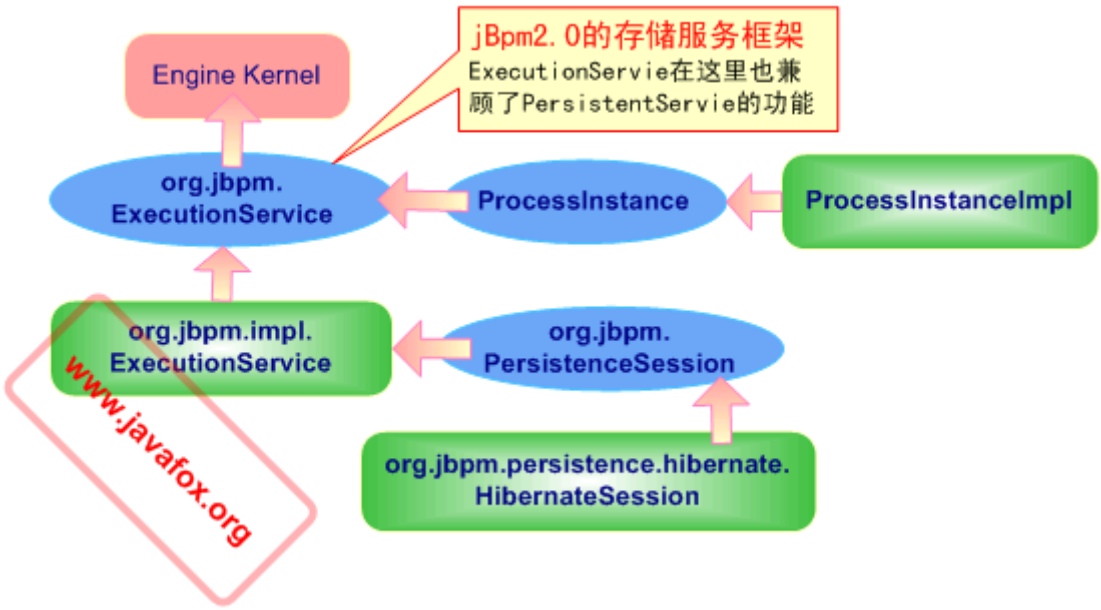
7.6.1.2. 流程实例存储服务

流程实例的存储是另外一个必须的服务，其本身对引擎来说是在一种支撑扩展，同时也是流程执行服务（Enactment Service）的一种。



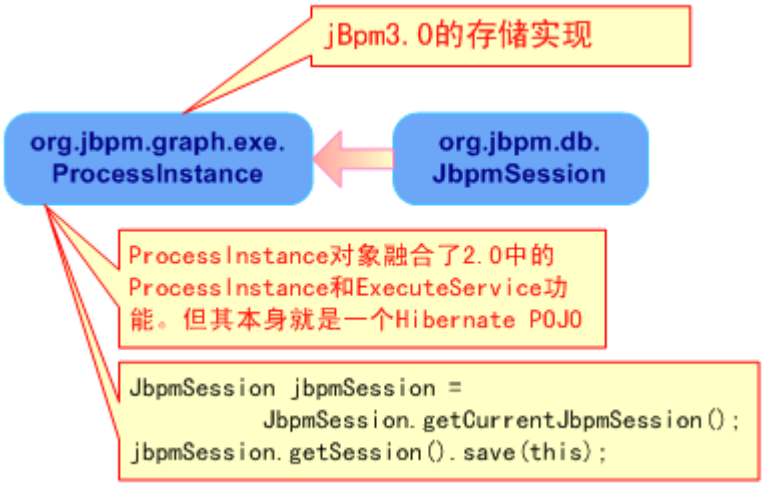
这种存储服务结构在商业工作流引擎中是比较流行的，即使在开源引擎中，也有很多采用的是这种结构，比如早期的 jbpm2.0（但 jbpm3.0 抛弃了这种结构，jbpm3.0 将原本在 2.0 中的 persistent 层彻底抛弃了）。

让我们先在来看看 jBpm2.0 的存储处理结构：



从上面的 jBpm2.0 的存储实现图中，我们看到，jBpm2.0 的存储服务框架基本是符合我们上面说的框架结构。

原本这套结构是非常支持扩展，可以扩展流程实例对象的实现，扩展存储服务的实现。但是在 jBpm3.0 中，这套结构被彻底的推翻了。

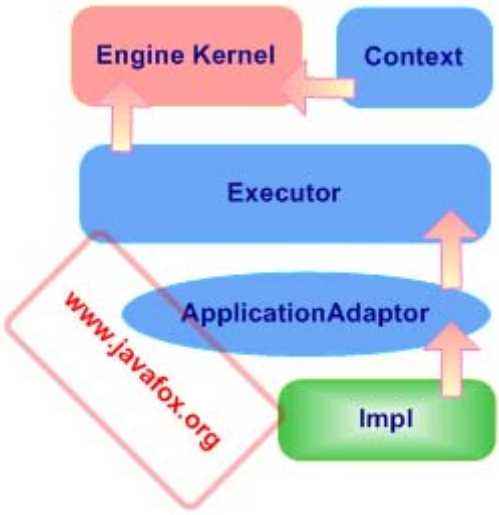


jBpm3.0 采用这样的存储服务方式，使得结构更加简单（只有两个主类）。但我个人认为，采用这样的结构，并不利于 jBpm 的引擎发展。因为 jBpm3.0 不仅抛弃了存储服务，而且将 2.0 中非常良好的执行服务（ExecutionService）也抛弃了，而娇柔在流程实例（ProcessInstance）对象中。

7.6.1.3. 应用适配

工作流引擎难免会与其他应用系统大交道，调用其它应用系统进行一些处理，比如我们最常用的要数“邮件系统”了。

通常的应用适配的结构如下图所示：



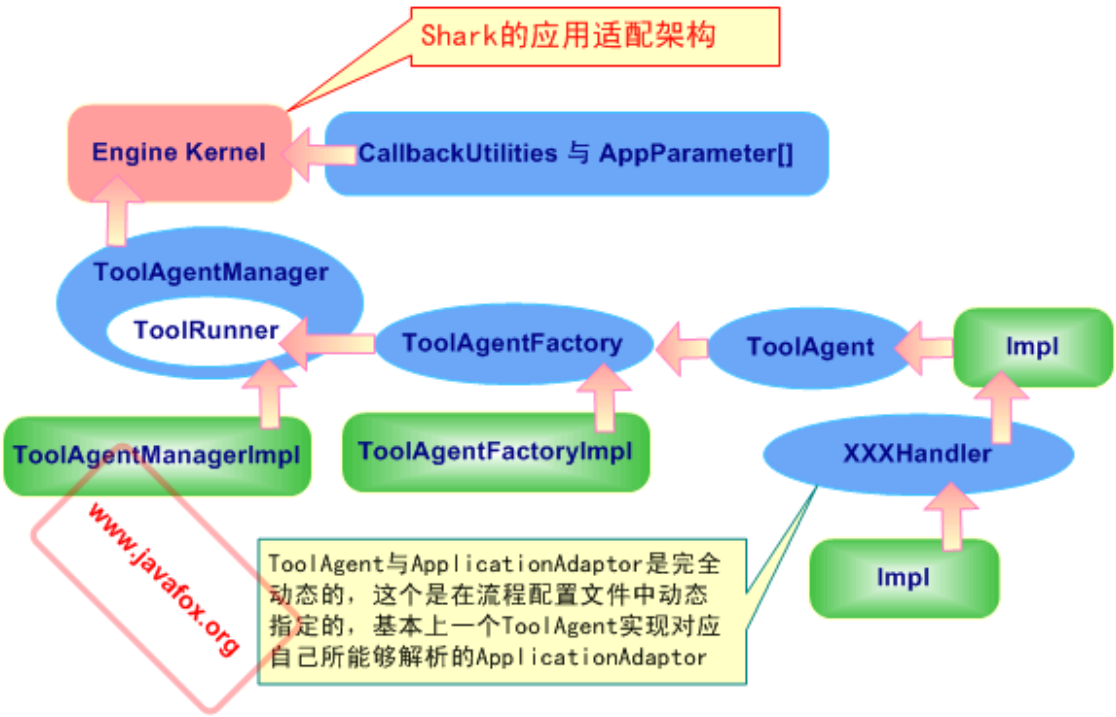
应用适配架构主要通常都包含三个主要要素：

（1） 环境资源对象：应用实现中可以获取并处理的对象资源

（2） 执行器：运行应用适配器的执行体

（3） 应用适配实现扩展接口：共对外扩展的接口

接下来让我们来看看 Shark 这个引擎是如何实现应用适配的：



Shark 的实现方式是复杂的，当然也是非常灵活的，其可以任意的扩展 ToolAgent 以及此 ToolAgent 所应对的应用处理类。

但是唯一不足的就是，Shark 因为完全遵循 XPDL 规范，所以对“环境资源对象”的操作和管理，显得较为苍白和不足。

相比较而以，jBpm 和 OSWorkflow 则实现的非常简单，但是 jBpm 和 OSWorkflow 从本质上说并没有“应用适配”这个概念，但是两种引擎为了便于扩展应用，都提供彼此可以处理应用的接口。jBpm 提供了 org.jbpm.graph.def.ActionHandler 接口，OSWorkflow 提供了 com.opensymphony.workflow.FunctionProvider 接口。

至于 jBpm 和 OSWorkflow 的“应用处理接口”就不再详细叙述，有兴趣的可以去参考它们的帮助文档。

7.6.2. 辅助流程运行的（Assistant）

这一种扩展对于工作流引擎来说，不是必须实现的。即使实现，主要是辅助引擎做一些处理性的工作。

主要有如下几种：

（1） 条件处理，或额外条件实现类

（2） 内部后台功能处理

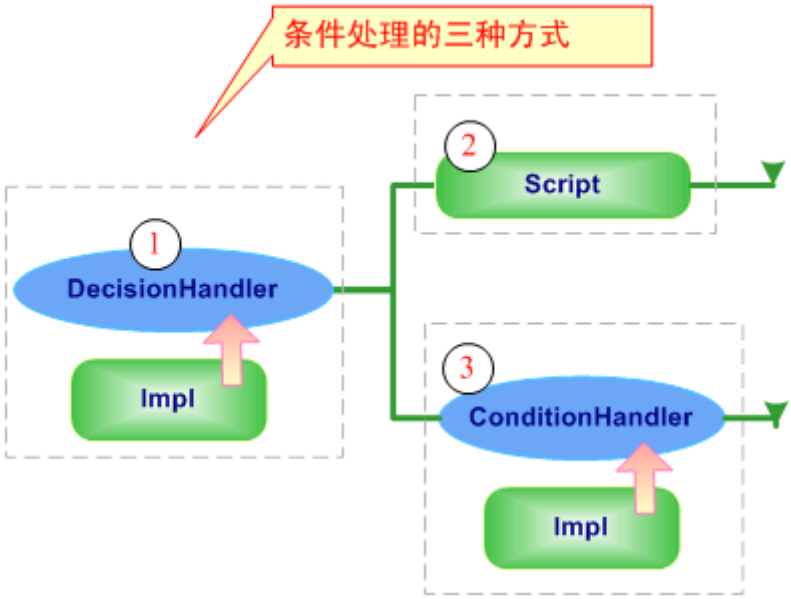
（3） 客户额外操作处理

下面来详细说说这三种扩展：

7.6.2.1. 条件处理

现在没有哪个引擎不支持条件计算的，即使最简单的引擎，也会支持对“脚本”语言的解析计算。当然，复杂的，就会额外引入一些条件处理接口来满足额外的客户需求。

通常情况下，处理条件的方式有如下三种：



（1） 采用一个“决策处理接口”，来判断后序可执行的分支
（2） 每一个分支上可以负载“脚本条件”，来判断此分支是否可以被运行
（3） 每一个分支上可以负载“条件处理接口”，来判断此分支是否可以被运行

至于条件的处理架构，到不用再的说了。大抵与上面的应用处理有些类似。

7.6.2.2. 功能处理

这和上一节中说的应用处理有些类似。只是上面的应用处理是大抵比较明确的（基本上针对所有流程都有可能会涉及到），而此处说的功能，则是跟每个流程实施相关的，毕竟每一个流程都有自己的特定业务逻辑处理，所以需要很广泛的“功能处理”或者说“事件处理”支持。

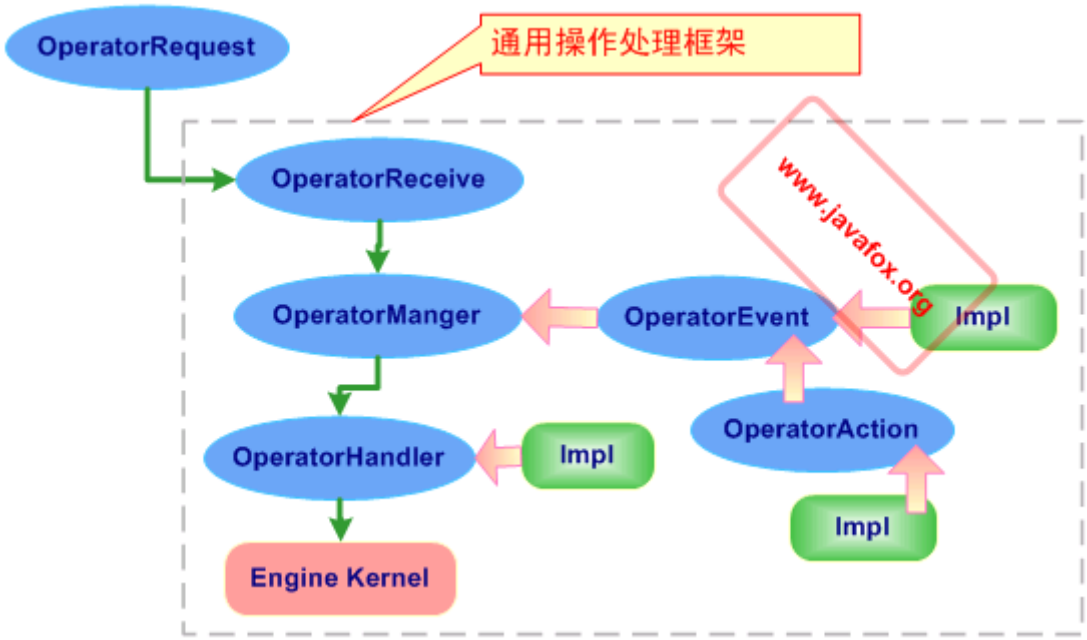
可以说，jBpm 的 Event 设计，OSWorkflow 的 Function 设计，都大抵是用于解决这类问题的。

7.6.2.3. 客户操作处理

在现实应用中，客户的操作行为和操作习惯是丰富多样的。开源的工作流引擎都基本上没有这一层次的扩展支持，但是商用工作流引擎，却不得不在这一层上做一些应用性的支持。

我们可以将客户的一些行为抽象为“操作”。比如在国内流程处理过程中，比较常见的额外操作是“退回”“取回”“跳转”等等。

这对一些外围的操作，我们需要提供一套机制来满足多变的需求，如下图所示：



7.6.3. 增强流程运行的（Enactment）

这种类型的扩展，即不是引擎所必需的，也不是简间单单的外围处理辅助。一旦实现，必然会增强引擎内部的处理能力。

当然，这类的扩展并不是很多。我们平常所接触到的，主要是一些策略性的扩展、事件监听、超时处理、代理人处理、工作日历处理等等。策略性的扩展也主要是用在“工作项分配”“工作项执行”“工作项提交”等几个方面（采用 WfMC 的工作流术语）；而事件监听则主要针对流程处理中所发生的行为进行监听，当某一种行为发生的时候，可以额外的执行什么事情；超时处理则主要针对流程实例、活动实例超时的时候执行什么处理。

这一类的扩展实现，一般在引擎发布的时候都会提供一些默认的实现类，但这些默认实现类大多为“空实现”。

7.6.3.1. 策略扩展

有关策略，到是很难抽象出一个通用的框架结构。基本上每个引擎都会有自己的实现规则和方式。

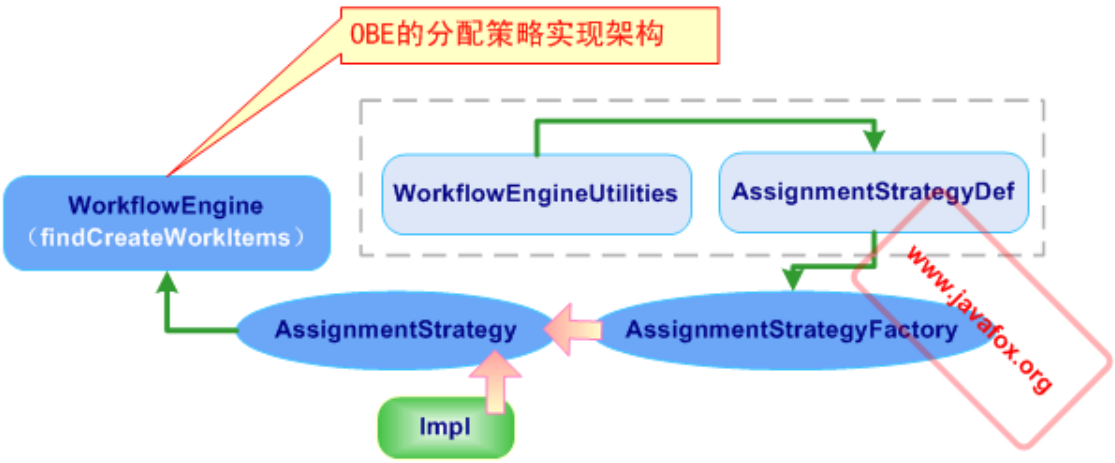
首先让我们来看看 Shark 这个引擎的实现方式。

作为一个庞大的开源工作流引擎，在“策略”的实现机制上显得却不够完美，整个结构过分的单一。

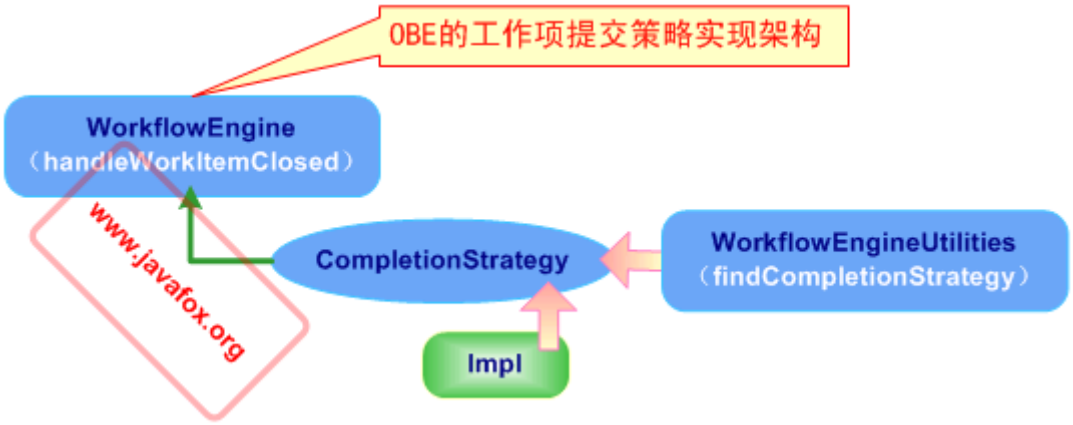
其实，在 Shark 引擎中是没有“策略（Strategy）”这个含义的。上图显示的是 Shark 在分配工作项的时候得处理机制和实现（在 Shark 中也没有明确的 WorkItem 概念，实现的替代产物是 WfAssignment）。

相比较而言，OBE 这个引擎就要丰富多了，明确地提出了“分配策略”和“提交策略”概念，并提供两个供扩展的接口：AssignmentStrategy 和 CompletionStrategy。当然有一点还是比较遗憾的，OBE 的策略实现类，只针对 Process 和 Package，而不是 Activity。

下图显示的是 OBE 的分配策略实现架构：



下图显示的是 OBE 提交策略的实现架构：



对于开源引擎来说，OBE 能够明确提出“分配策略”和“提交策略”已经很不容易。虽然这些策略的实现方式大致雷同：在定义文件或配置文件中注册不同的实现类，但对于这些嵌入式的开源引擎来说，已经足以应付需求。

7.6.3.2. 事件监听扩展

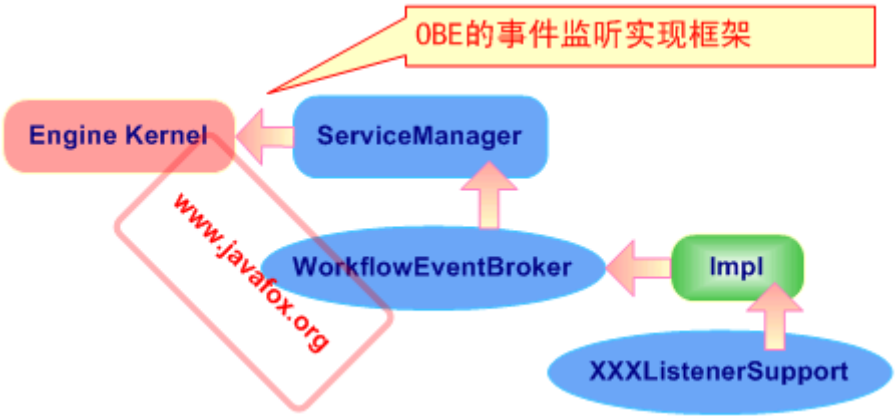
事件监听扩展基本上是非常经典的一种扩展体系，但是在很多“微型”或“轻量型”嵌入式工作流引擎中是很少出现的，这主要是因为“事件监听”的实现架构是比较简单和明确，但是“需要实现的方法”和“实现的接口”过于繁杂，并且在实际应用中，这种“监听机制”却很少真正有扩展的必要。

有必要说明的是，“事件监听”和“事件驱动链（EPC）”、“ECA（Event-Condition-Action）”是不同的。比如在 jBpm 的 Event 处理机制，有些类似 ECA 的思想。

至于“Event-Listener”通用结构就没有必要介绍了，这样的文章铺天盖地都是。下面我来来简要看看 OBE 的事件监听的实现架构。

OBE 的事件监听实现

OBE 的事件监听是繁杂的，在传统的 Support 实现框架下，又封装了一层 Broker 代理，见下图。

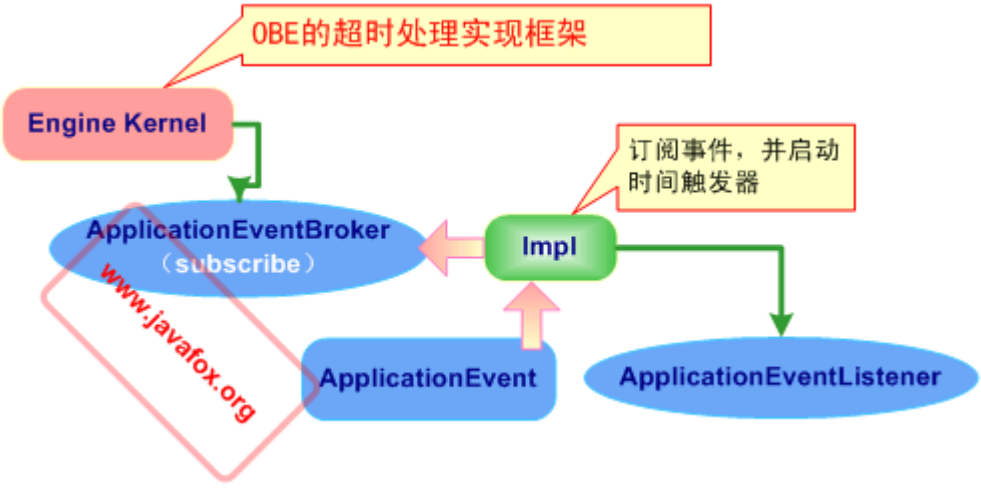


在 OBE 事件监听的框架中，还有一个 ApplicationEventBroker 代理，这个我们将在下面的“超时处理”体系中提及

7.6.3.3. 超时处理

超时处理本身跟引擎关系不大，但是在现实应用处理中，又是不得不支持的功能。我们无法预见客户最终会采用什么样的通知方式，所以工作流引擎必须提供一个扩展的机制，让二次实施的时候扩展实现。

明确提出了超时处理的引擎只有 OBE，可惜 OBE 所提供的处理机制并不是很完美，其采用不够灵活的 Application Event Listener 机制来处理。



在 OBE 的超时处理思想中，用到了“订阅”“时间触发器”“事件监听”三个实现模式来协作完成。这三者的结合，也是比较常见常用的方式。

这种实现方式，是非常值得商业引擎借鉴的。但是商业引擎的底层实现可以借鉴这种机制，但是要在应用角度超越这种方式。

至于“超时处理”的通用框架及解决方案不是本篇的内容，我会在后续的《工作流之魂》中详细介绍，这里只稍微提一下，以供参考。

超时处理主要注意如下几点：

A、 什么时间超时—— 计算超时时间点，这个与下面说的“工作日历”有或多或少的关系

B、超时的监听方式——实时监听，还是隔时监听，还是固定时间监听

C、超时后的处理对策（继续等待发通知，还是直接往下进行）

D、 超时后的处理工具（邮件通知，还是别的）

E、通知人

以上几点就不详细叙述，有兴趣的可以自己思考思考。

7.6.3.4. 代理人处理

代理人的处理扩展接口不复杂，一般应用都几乎支持。其作用域也主要是在“人员分配的时候”进行代理过滤。至于实现方式不再累赘的叙述。

7.6.3.5. 工作日历

工作日历跟代理人一样，本身不属于工作流范畴，但是任何现实应用的引擎都必须提供对其的支持。

工作日历最主要的功能，也主要是用于协助处理“任务的办理期限”。其框架和结构也是非常简单，实现方式上也不再累赘的叙述。

7.7. 基础组件层

任何应用组件都必然会依赖一些工作组件包或基础框架。这一层也就是用于解决这个问题的。对于整个应用产品来说，这一层是被所有应用组件所共有的。

8. 结尾

终于到结尾了，中间杂事太多，断断续续写了一个多星期才完成。写完之后，自己读了读，发现还是有很多地方写的不够细，却也懒的去修饰了。其实，这篇文章，最主要的隐含意思，是告诉大家“**如何从一个结构上，去分析引擎**”。

当你去读一个引擎的时候，当你去开发一个引擎的时候，最主要是了解或规划出这个引擎的体系结构。

计划写三篇有关工作流引擎内部实现的文章，从三个方面阐述：

角度	文章	完成情况
从引擎内核调度算法角度	《工作流引擎核心调度算法与 PetriNet》	2005-4-17

从引擎体系结构角度	《微内核工作流引擎体系与部分解决方案参考》	2005-7-24
从引擎内核角度	《工作流引擎内核揭秘》	未完成

如今已经完成两篇，计划今年 11 月份发布第三篇。到时候，希望通过这三篇文档，给大家揭示出一个完整的工作流引擎内核。

胡长城（银狐 999）
2005 年 7 月 24 日星期日 晚