附　录

辛鹏　荣浩　著

流程的永恒之道

工作流与BPM技术的理论、规范、模式及最佳实践

辛鹏
邮箱：xinpeng0618@163.com
新浪微博：http://weibo.com/opug
csdn博客：http://blog.csdn.net/snow_fox_yaya

荣浩
邮箱：ronghao100@gmail.com
新浪微博：http://weibo.com/ronghao100

版 权 声 明

附录A
工作流控制模式

模式最早出现在C. 亚历山大的《建筑的永恒之道》里。在这本建筑学的经典巨著里，亚历山大说：“永恒之道是一个唯有我们自己才能带来次序的过程，建筑或城市只有踏上了永恒之道，才会生机勃勃。”建筑模式，就是通往这条永恒之道的大门。亚历山大描述了253个建筑模式，这些模式和那里发生的事件一起赋予城市和建筑以特征。模式语言则是对应于那些使建筑美妙并加以深刻观察的模式的集合，它综合了我们对建造的认识。建筑的永恒之道正在于建筑模式，在于我们对建筑的认知达到一致即拥有共同的模式语言，在于我们真正遵循了自己的内心和顺其自然。

工作流模式亦是如此，它紧紧抓住了在我们周围发生的事件，赋予不同组织以特征。处处讲流程，处处明确职责，这是大企业；处处讲特色，处处要干预，这是政府机构；处处讲人，处处要求通才，这是创业公司。如何把某个创造价值的活动拆分成不同的活动，如何将这些活动协调整合起来，以便实现最终目标，不仅关系到流程，也关系到组织结构，更关系到人。工作流模式分为4种：控制模式、资源模式、数据模式和异常模式。

控制模式关注对流程进行建模，将商业目标的实现根据组织所进行的工作和现有的技术体系拆分成一系列的活动。资源模式关注组织内部资源的协调。数据模式关注流程中信息的传递。异常模式关注流程执行过程中出现偏离期望的情况。但不管是控制模式、资源模式、数据模式，还是异常模式，它们一致的核心都是人。人，永远是管理中的核心。

工作流控制模式共有43种，分为8组，分别是基本控制模式、高级分支和同步模式、多实例模式、状态模式、取消和强制完成模式、迭代模式、结束模式和触发模式，如图A-1所示。

- 基本控制模式关注流程里最基本的顺序、并发、条件和合并路由，是其他控制模式的基础；

- ❑ 高级分支和同步模式关注流程里更复杂一些的分支和同步场景；
- ❑ 多实例模式关注活动在一个流程实例中多次执行的场景；
- ❑ 状态模式关注流程实例状态对流程执行所产生的影响；
- ❑ 取消和强制完成模式关注流程实例/活动的取消，这通常与工作流异常相关；
- ❑ 迭代模式关注流程实例里的重复行为；
- ❑ 结束模式关注什么情况下流程实例执行结束；
- ❑ 触发模式关注外部环境变化对流程活动的影响。

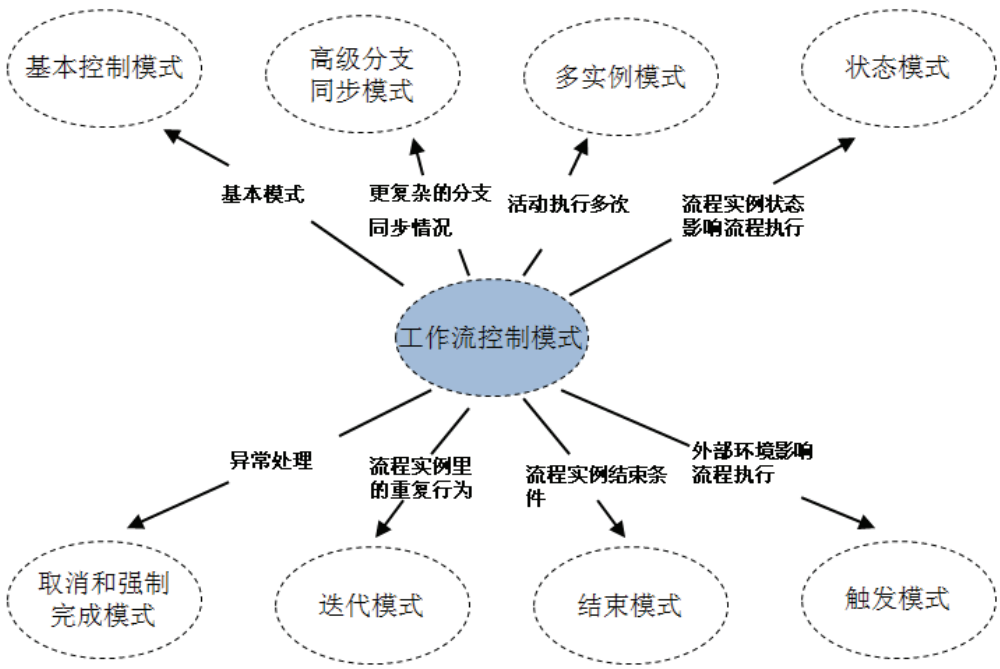


图 A-1　控制模式

工作流控制模式的出发点基于对实际业务的描述，而工作流系统对模式的支持程度则直接决定了该系统对业务的建模能力。所以，衡量一个工作流系统时，必须考虑其对工作流模式的支持程度。

基本控制模式

基本控制模式包括基本的顺序、并发、条件和合并路由，是其他控制模式的基础。基本控制模式有以下5种，如图A-2所示。

- ❑ 顺序：活动顺序执行。
- ❑ 并发分裂：分支分裂为两个或多个后续分支，所有后续分支都被同时触发执行。

- 同步：两个或多个分支合并为一个后续分支，只有所有分支都执行完毕后，后续分支才会被触发执行。
- 排他选择：分支分裂为两个或多个后续分支，只有一个后续分支被选择执行。
- 简单合并：两个或多个分支合并为一个后续分支，分支不需要同步，任何一个分支执行完毕后就会触发后续分支的执行。

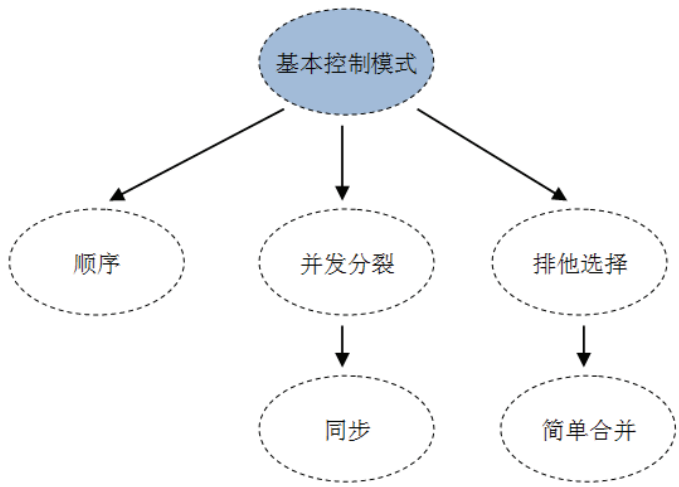


图 A-2　基本控制模式

顺序（WCP_1: Sequence）

描述

在一个流程实例里，活动在前续活动完成后顺序触发，如图A-3所示。

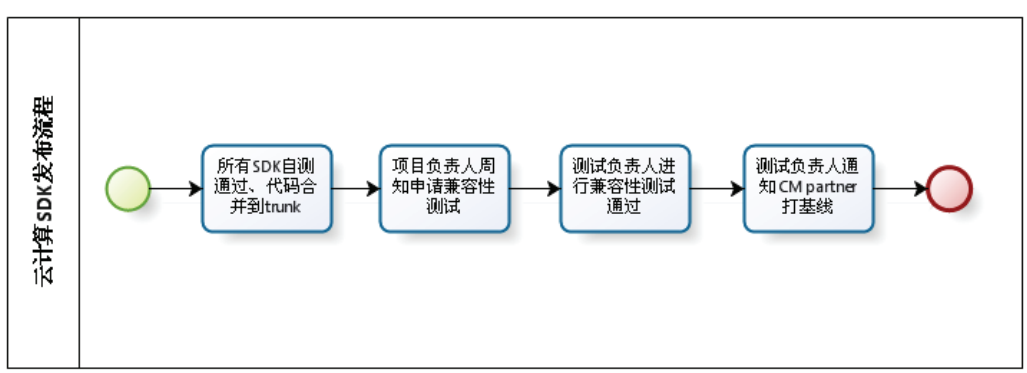


图 A-3　活动顺序执行

同义词

顺序执行、串行路由。

应用

顺序模式是工作流建模的基础，是流程定义里最基本的构建块，用以描述连续互相依赖的一系列活动。

并发分裂（WCP_2: Parallel Split）

描述

分支分裂为两个或多个后续分支，分支执行完毕后触发后续并发分支的同时执行，如图A-4所示。新员工入职时，先去人力资源部报道，接下来填写入职资料和签订合同，于此同时，IT部门帮忙开通RTX邮件开发环境权限、初始化机器，两个工作同时进行。

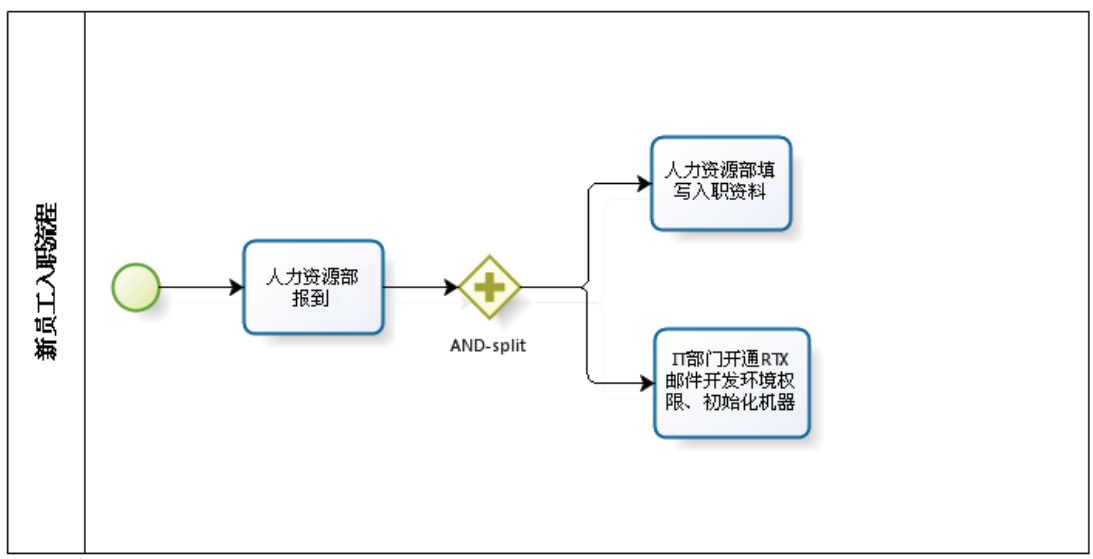


图 A-4　并发分裂

同义词

AND-split、Fork、并行路由、并行分裂。

应用

作为设计流程的一个基本原则，我们应该尽可能采用并行过程。信息化的一个重要作用就是加快信息的流动，让并行工作越来越成为可能。试想如果没有办公自动化系统，我们到人力资源部报到后，IT部门并没有及时收到消息，那么我们填完入职资料后还要去IT部门申请机器，这个

效率就非常低了。

对流程而言，流程实例执行时间是最重要的衡量指标：执行时间越短，对顾客越有效率，没有人喜欢等待。

同步（WCP_3: Synchronization）

描述

两个或多个分支合并为一个后续分支，被合并的分支都执行完毕后，后续分支才会触发，如图A-5所示。当资料填写完毕、IT部门初始化机器完毕并送至工位后，我们就可以收拾工位并和新同事打招呼了。

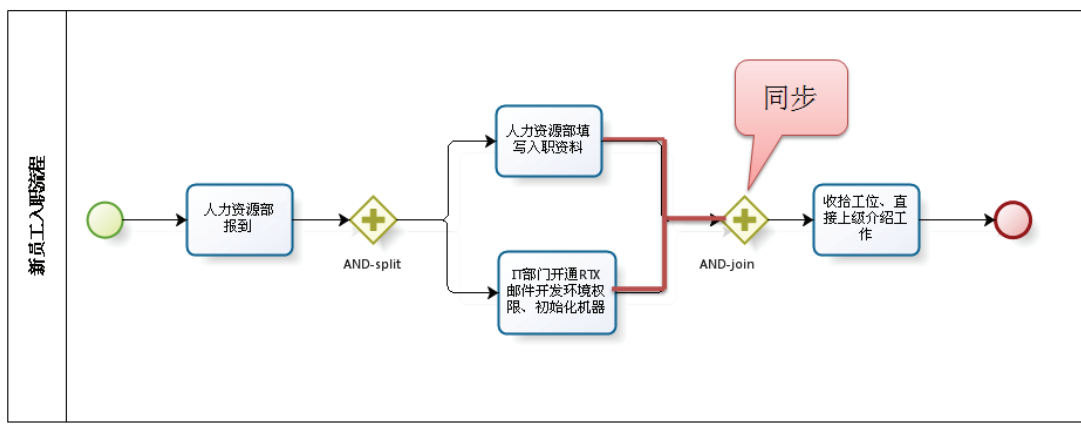


图 A-5　分支同步

同义词

AND-join、汇聚、同步。

排他选择（WCP_4: Exclusive Choice）

描述

分支分裂为两个或多个后续分支，分支执行完毕后，根据用户决策或流程数据选择触发一个后续分支执行，即多选一，如图A-6所示。在线购物时，确定订单后我们需要选择付款的方式，或者在线支付或者货到付款。

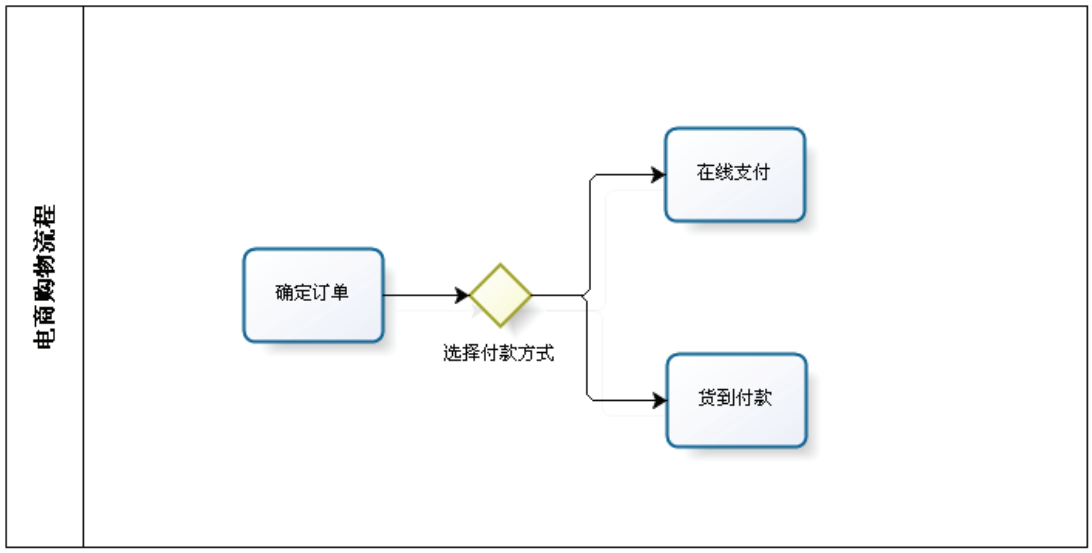


图 A-6　排他选择

同义词

XOR-split、排他OR-split、条件路由。

应用

流程实例中的路由条件判断，路由条件是前续活动的输出、流程数据、表达式或者规则引擎的计算结果。

简单合并（WCP_5: Simple Merge）

描述

两个或多个分支合并为一个后续分支，任何一个分支执行完毕后就会触发后续分支的执行，不需要同步，遵循先进先出的原则。该模式有个前提条件，即前续分支有且只有一个会执行，这一条件限定了后续分支只会被触发一次。

如图A-7所示，选择完付款方式后我们就可以提交订单，整个在线购物流程结束。

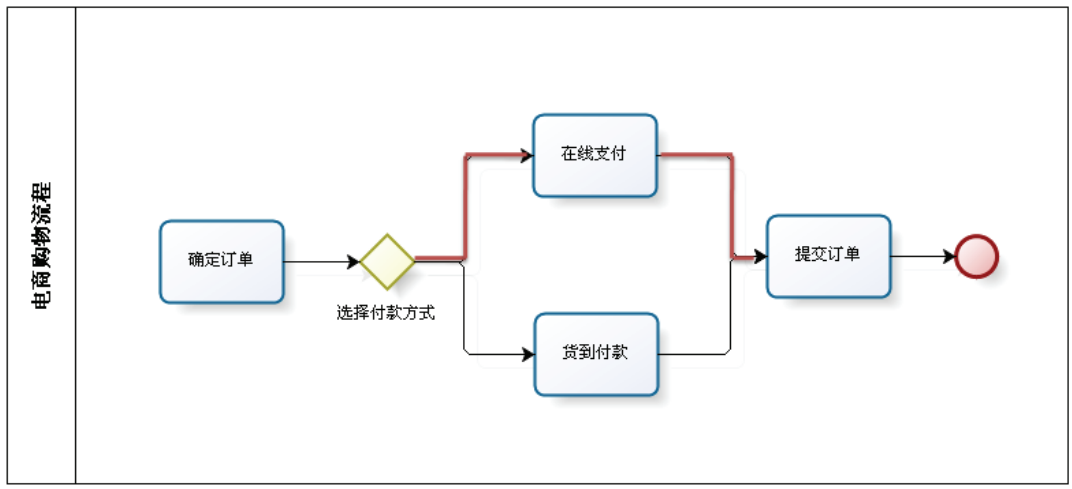


图 A -7　简单合并

同义词

XOR-join、排他OR-join、merge。

小结

基本控制模式非常简单，顺序、排他选择、简单合并模式组合的流程和我们编写程序的逻辑流程图非常相似，这三种模式同时能够对程序的逻辑流程图进行建模。

商业快速开发平台产品就使用了流程引擎来编排程序逻辑。他们的做法是将细粒度的代码逻辑封装为运算构件，然后再通过流程的可视化编辑器将这些运算构件粘合起来。这样，传统方式下采用代码实现业务逻辑的过程，变成了绘制流程图的过程。这样的实现看上去很美，其实存在严重的弊端。首先，编写代码变得复杂了，明明几十行代码能够实现的逻辑却需要经过编写构件、绘制程序流程图、部署、运行好多步才能实现，编程效率非常低下；其次是代码的执行效率低，程序的运行需要经过一次流程定义的解释才能执行；最后是这种实现完全牺牲了语言自身的特性，面向过程，很难提供代码级别的单元测试环境，只能提供有限的调试。该实现实际上是定义了一种简单的流程语言，通过该流程语言来进行功能函数（运算构件）调用的编排。活动编排没有问题，服务编排也没有问题，但是如果编排细粒度到功能函数，那么就超出了流程引擎的作用域。正如搭建房子，用积木搭建挺好，用木屑搭建就过犹不及了。提升编程效率的最好途径总是语言而不是工具。

高级分支和同步模式

高级分支和同步模式使得各个工作流产品在技术水平上拉开档次，技术上实现比较复杂。高级分支、同步模式共有14种，如图A-8所示。

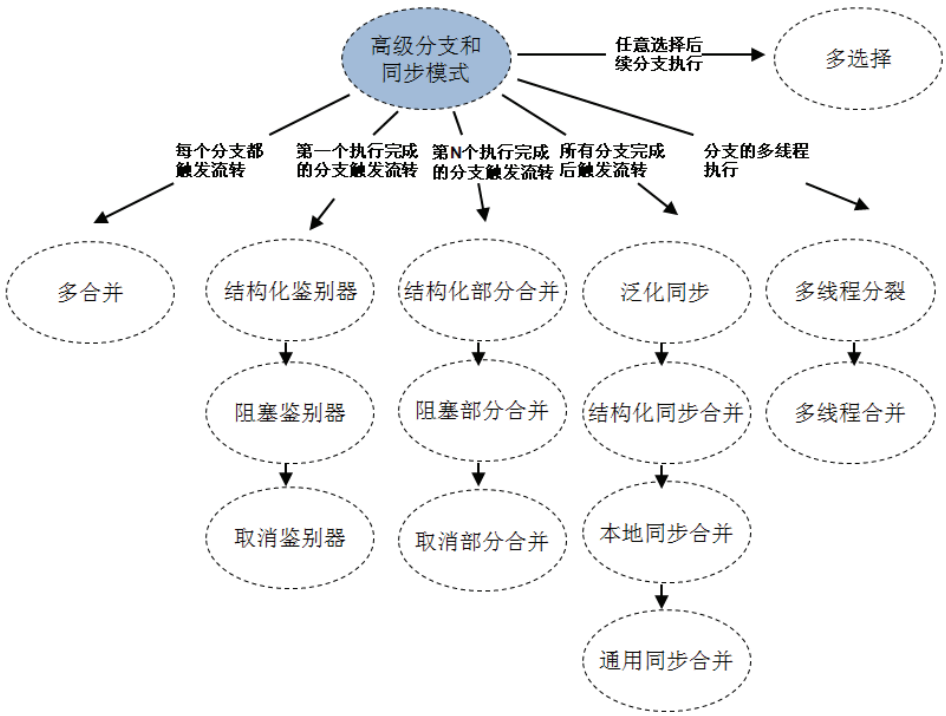


图 A-8　高级分支和同步模式

- 多选择：分支分裂为两个或多个后续分支，当分支执行完毕后会选择触发后续分支的一个或多个同时执行，M选N。
- 结构化同步合并：两个或多个分支合并为一个后续分支，只有当所有被实际触发的分支都执行完毕后才会触发后续分支的执行。
- 多合并：两个或多个分支合并为一个后续分支，每个实际触发的分支执行完毕后都会触发后续分支的执行。
- 结构化鉴别器：两个或多个分支合并为一个后续分支，第一个执行完毕的分支触发后续分支的执行，其他分支继续执行，但是被忽略，执行完毕后不再触发后续分支。
- 阻塞鉴别器：两个或多个分支合并为一个后续分支，第一个执行完毕的分支触发后续分支的执行，其他分支继续执行，但是被忽略，执行完毕后不再触发后续分支。如果分支有多个执行线程，那么在第一个执行线程被合并之前，后续线程被阻塞。
- 取消鉴别器：两个或多个分支合并为一个后续分支，第一个执行完毕的分支触发后续分支的执行，其他分支取消执行。
- 结构化部分合并：两个或多个分支合并为一个后续分支，假设流程实例中实际触发的分支为M个，第N个实际触发的分支执行完毕后触发后续分支的执行，N < M，其他分支继续执行，但是被忽略，执行完毕后不再触发后续分支。

- 阻塞部分合并：两个或多个分支合并为一个后续分支，第N个实际触发的分支执行完毕后触发后续分支的执行，其他分支继续执行，但是被忽略，执行完毕后不再触发后续分支。如果分支有多个执行线程，那么在第一个执行线程被合并之前，后续线程被阻塞。
- 取消部分合并：两个或多个分支合并为一个后续分支，第N个实际触发的分支执行完毕后触发后续分支的执行，其他分支取消执行。
- 泛化同步：同步模式的泛化，两个或多个分支合并为一个后续分支，等待所有分支都执行完毕后，后续分支才会触发，支持多个分支执行线程的同时合并。
- 本地同步合并：两个或多个分支合并为一个后续分支，该模式基于本地数据来决定需要同步的分支以及时机。
- 通用同步合并：两个或多个分支合并为一个后续分支，当所有被实际触发的分支都执行完毕并且未来也不会再被触发后才会触发后续分支的执行。与结构化同步合并的区别在于没有前提条件：不需要结构化建模，也不需要执行线程安全。
- 多线程合并：在流程的一个特定点，将一个分支的多个执行线程合并为一个执行线程。合并线程的数量在定义期确定。
- 多线程分裂：在流程的一个特定点，为一个分支初始化多个执行线程。执行线程的数量在定义期确定。

多选择（WCP_6: Multi-Choice）

描述

分支分裂为两个或多个后续分支，当分支执行完毕后，根据用户决策或流程数据选择触发后续分支的一个或多个同时执行，即M选N模式（多选多），后续有M个分支，根据运行时情况选择N个分支进行执行，1 <= N <= M。如图A-9所示，当110接到报警电话后，会根据情况，如是否有火情、是否有伤亡，选择是否同时出动消防和救护。

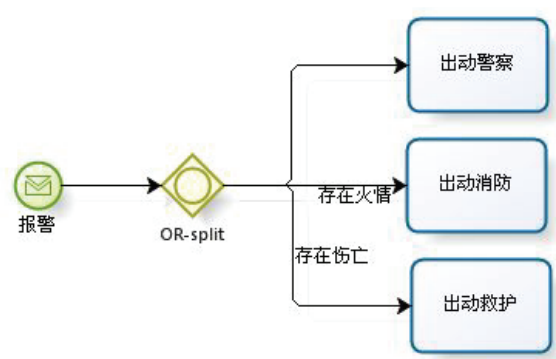


图 A-9　多选择

同义词

条件路由、多选、OR-split。

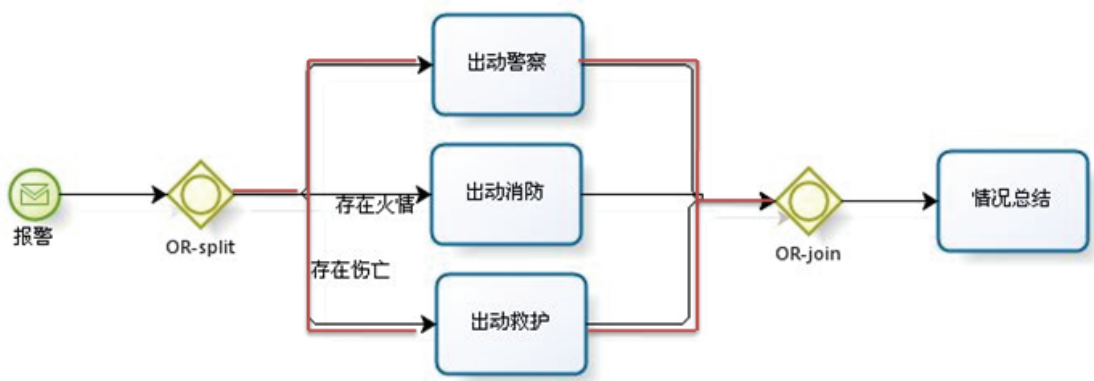
应用

从去年到今年，针对可能出现的重大灾害和突发事件，很多地方都建立起了联合紧急处理预案，围绕一个目标建立起来一整套处理机制，这套机制打破了原有职能部门的界限，减少了不必要的协调成本，是资源按照目标进行重组的一种形式。

结构化同步合并（WCP_7: Structured Synchronizing Merge）

描述

两个或多个分支合并为一个后续分支，只有当所有被实际触发的分支都执行完毕后才会触发后续分支的执行，分支需要同步。如图A-10所示，当110接到报警电话后，根据情况同时出动了警察和救护，当所有任务都执行完成后会进行出警情况的总结汇报。



图A-10　结构化同步合并

前提条件

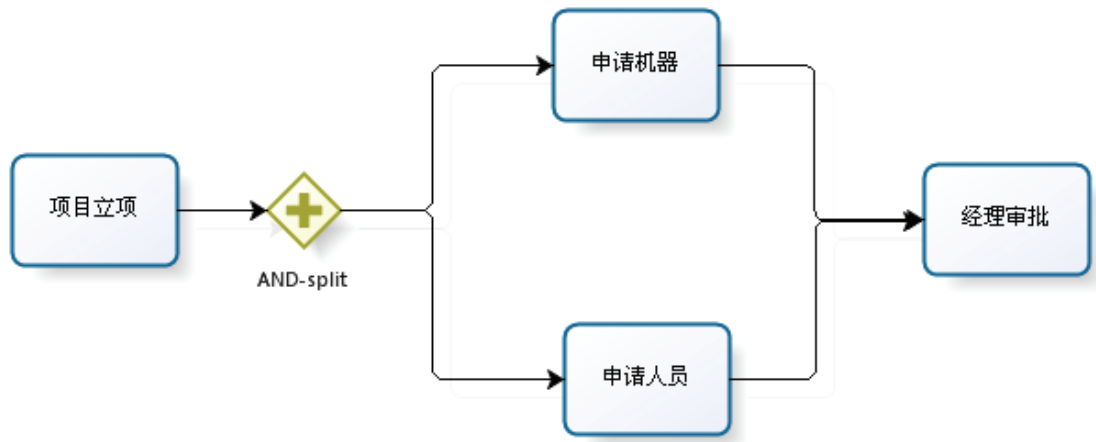
该模式存在两个前提条件：结构化和执行线程安全。

- 结构化：在流程定义里必须存在一个OR-split网关与该OR-join网关对应，OR-join网关合并从OR-split网关分裂出来的分支，在OR-split网关和OR-join网关之间的分支不允许存在任何的分裂和合并网关，如果存在，必须配对出现；
- 执行线程安全：在一个流程实例里，OR-join网关激活重置之前（合并完成当前所有分支执行线程之前），不允许OR-split网关再次激活产生需要合并的分支执行线程。

多合并（WCP_8: Multi-Merge）

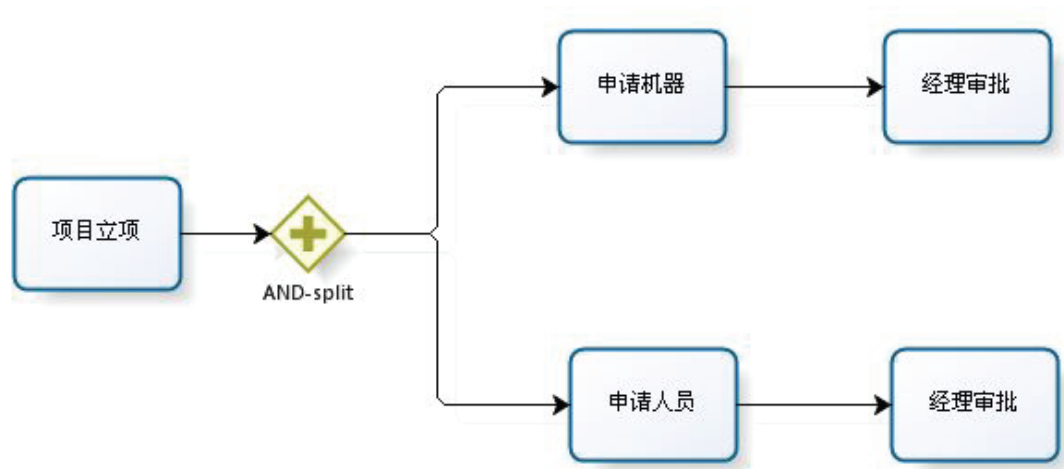
描述

两个或多个分支合并为一个后续分支，每个实际触发的分支执行完毕后都会触发后续分支的执行，如图A-11所示。项目立项后，项目经理需要申请服务器和申请项目人员，不管是申请服务器还是申请人员都需要经过一次部门经理的审批。



图A-11　多合并

图A-11的流程定义等效于图A-12所示的流程定义。



图A-12　经理需要分别审批两次

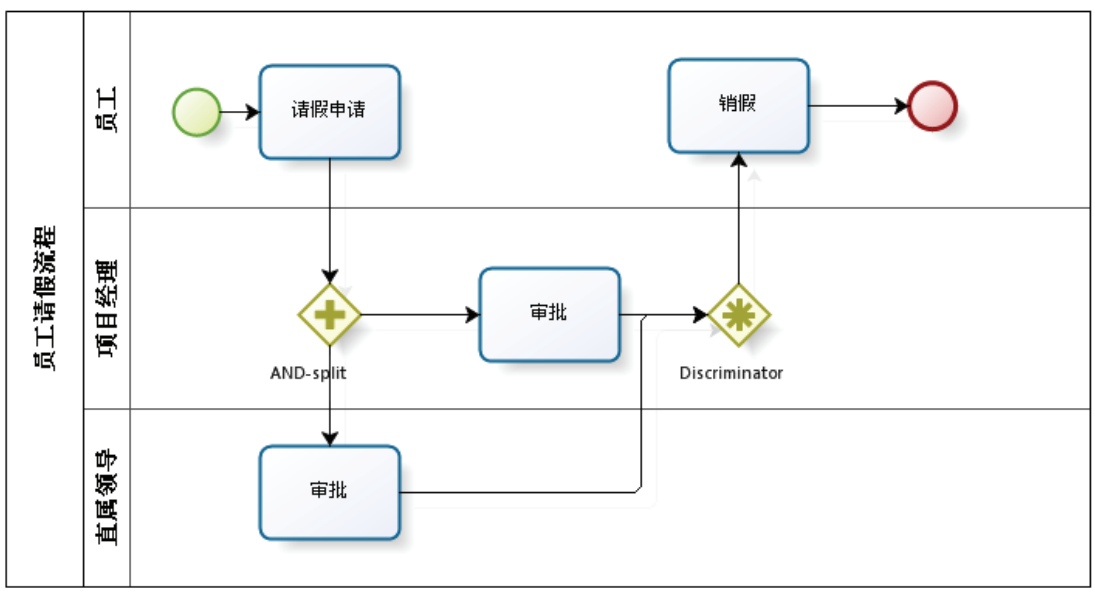
应用

多合并模式被用来简化建模。在存在很多并发分支且后续活动相似的情况下，这种模式能够减少流程定义的复杂度，类似于代码里的DRY原则。但应用该模式后，分支会产生多个执行线程，导致后续的合并节点处于一种线程不安全的状态。

结构化鉴别器（WCP_9: Structured Discriminator）

描述

两个或多个分支合并为一个后续分支，第一个执行完毕的分支触发后续分支的执行，其他分支继续执行，但是被忽略，执行完毕后不再触发后续分支。如图A-13所示，员工请假申请将同时发送给项目经理和直属领导审批，但只要一个审批通过即可。



图A-13　结构化鉴别器

前提条件

该模式存在两个前提条件：结构化和执行线程安全。

- 结构化：在流程定义里必须存在一个AND-split或OR-split网关与该Discriminator网关对应，Discriminator网关合并从AND-split或OR-split网关分裂出来的分支，在AND-split或OR-split网关和Discriminator网关之间的分支不允许存在任何的分裂和合并网关，如果存在，必须配对出现。

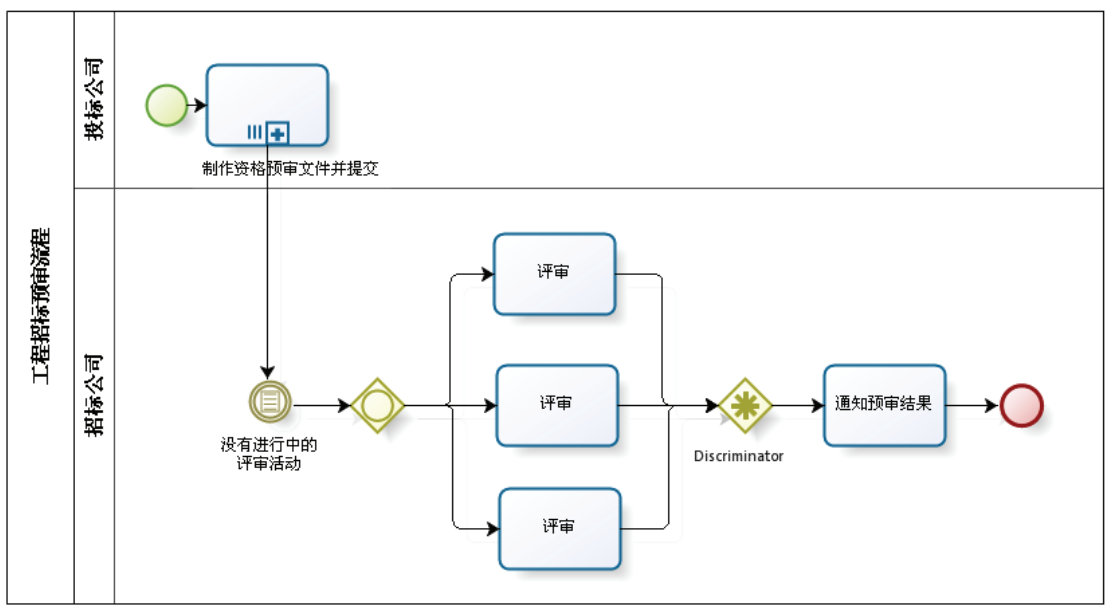
- 执行线程安全：在Discriminator网关被激活且未重置的情况下，不允许前续分支再次产生需要合并的执行线程。

阻塞鉴别器（WCP_28: Blocking Discriminator）

描述

两个或多个分支合并为一个后续分支，第一个执行完毕的分支触发后续分支的执行，其他分支继续执行，但是被忽略，执行完毕后不再触发后续分支。该模式的关键词是阻塞，如果分支有多个执行线程需要合并，那么在第一个执行线程被合并之前，后续线程被阻塞。与结构化鉴别器模式相比，该模式通过自身的实现机制保证Discriminator网关处于线程安全的状态。

如图A-14所示，工程招标预审流程，投标公司根据招标公告向招标公司报名，购买资格预审文件，接下来，根据资格预审文件要求，完成资格预审申请文件的制作，然后在规定时间内交到招标公司，进行预审。招标公司进行资格的预审，只要一个评委评审完成，我们即通知评审结果。我们使用多实例子流程对投标公司的投标流程进行建模；使用条件中间事件：没有进行中的评审活动，保证每次只对一家公司的申请进行审核。

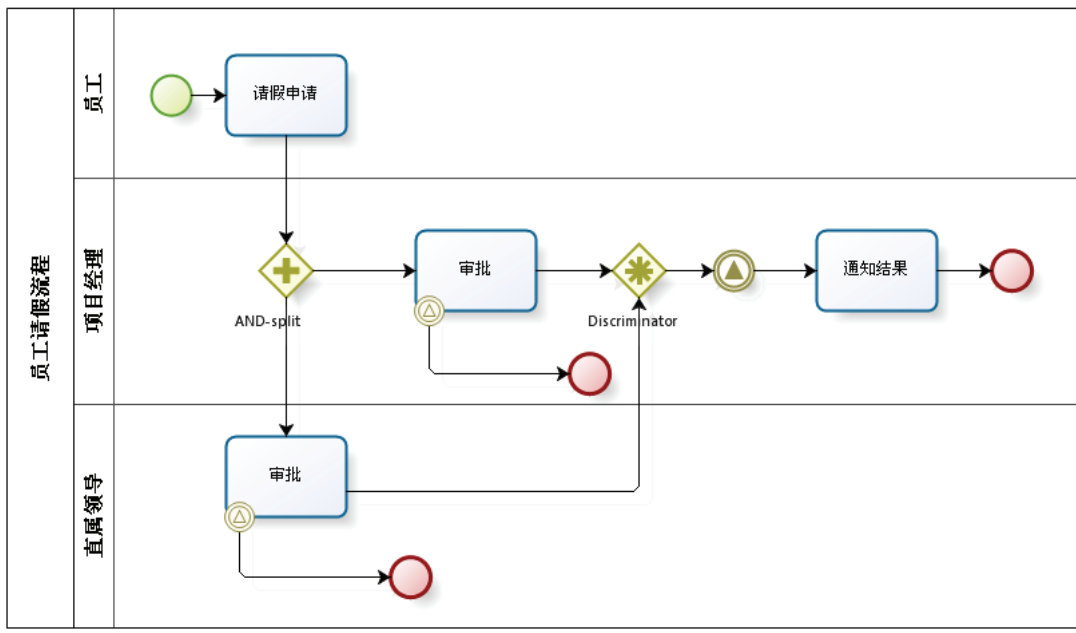


图A-14　阻塞鉴别器

取消鉴别器（WCP_29: Canceling Discriminator）

描述

和结构化鉴别器模式相似，第一个实际触发的分支执行完毕后触发后续分支的执行，区别是，其他分支不再继续执行，被取消。如图A-15所示，员工请假申请将同时发送给项目经理和直属领导审批，但只要一个审批通过即可，审批通过后我们发送审批通过的信号事件，另外一个进行中审批活动捕获该信号事件然后取消活动的执行。



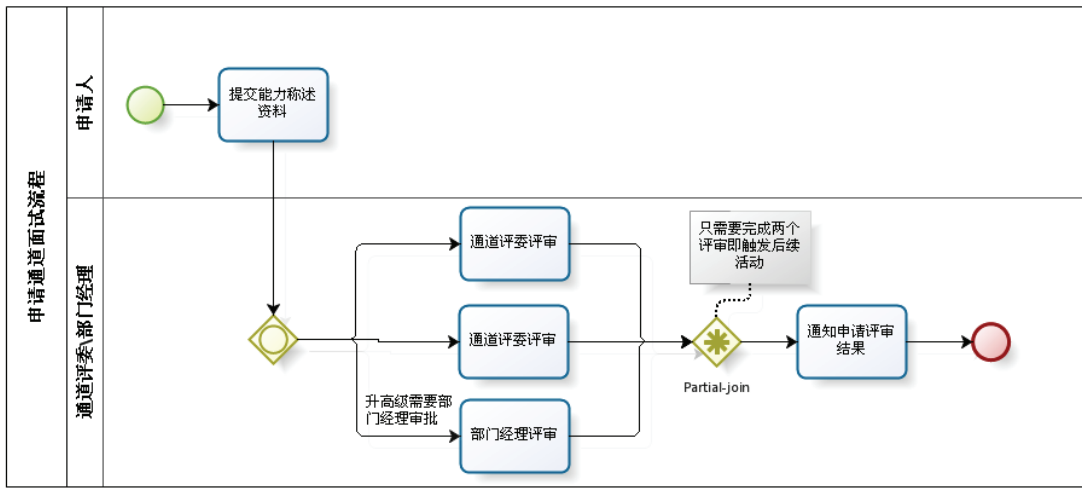
图A-15　取消鉴别器

结构化部分合并（WCP_30: Structured Partial Join）

描述

两个或多个分支合并为一个后续分支，假设流程实例中实际触发的分支为M个，第N个实际触发的分支执行完毕后即触发后续分支的执行，N <= M，其他分支继续执行，但是被忽略，执行完毕后不再触发后续分支。如图A-16所示，每年的晋级面试需要先提出申请，申请将发给两位通道评委评审，如果是升高级职称则还需要发送给部门经理评审，只需要两个人完成了评审，评审即完成，评审结果里只要有一个不通过则申请晋级不通过。

该模式是结构化鉴别器模式的泛化，在结构化鉴别器模式里，N=1。



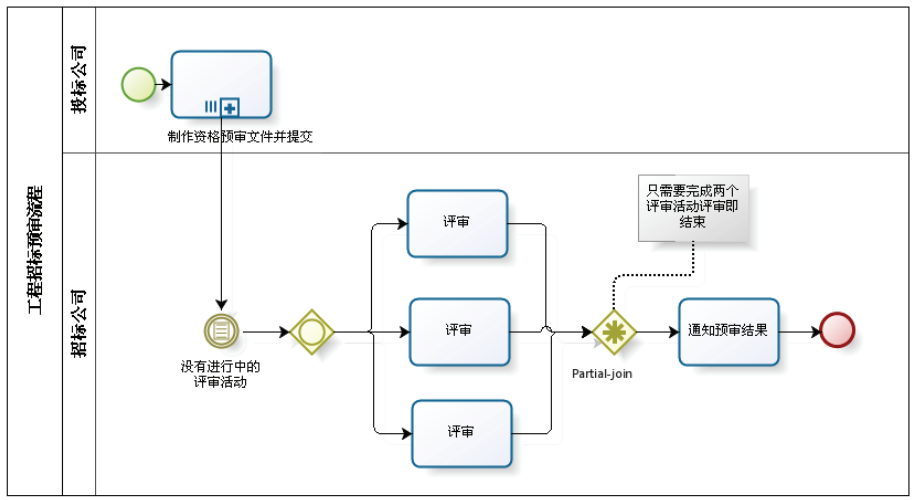
图A-16　结构化部分合并

阻塞部分合并（WCP_31: Blocking Partial Join）

描述

两个或多个分支合并为一个后续分支，假设流程实例中实际触发的分支为M个，第N个实际触发的分支执行完毕后即触发后续分支的执行，N<=M，其他分支继续执行，但是被忽略，执行完毕后不再触发后续分支。该模式的关键词是阻塞，如果分支有多个执行线程需要合并，那么在第一个执行线程被合并之前，后续线程被阻塞。与结构化部分合并模式相比，该模式通过自身的实现机制保证Partial-join网关处于线程安全的状态。

如图A-17所示，工程招标预审流程，招标公司进行资格的预审，需要两位评委完成评审，评审才结束，评审结果里只要有一个不通过则预审不通过。我们使用多实例子流程对投标公司的投标流程进行建模；使用条件中间事件：没有进行中的评审活动，保证每次只对一家公司的申请进行审核。



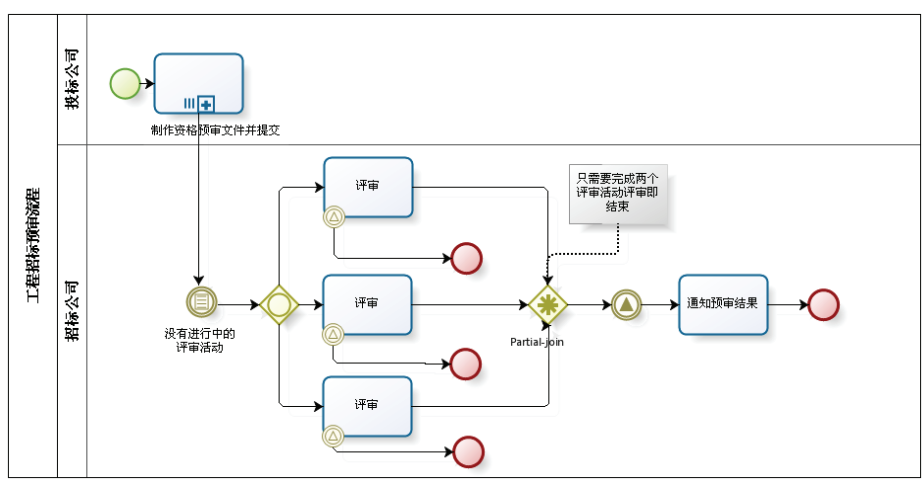
图A-17　阻塞部门合并

取消部分合并（WCP_32: Canceling Partial Join）

描述

和结构化部分合并模式相似，第N个实际触发的分支执行完毕后触发后续分支的执行，区别是，其他分支不再继续执行，被取消。

如图A-18所示，工程招标预审流程，招标公司进行资格的预审，需要两位评委完成评审，评审才结束，评审结果里只要有一个不通过则预审不通过。剩下一位评委的评审活动被取消。



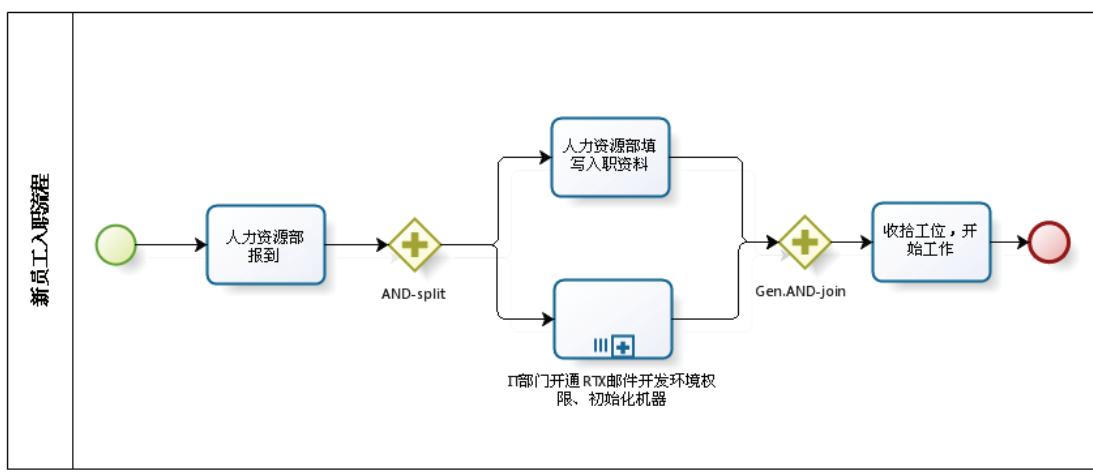
图A-18　取消部分合并

泛化同步（WCP_33: Generalized AND-Join）

描述

两个或多个分支合并为一个后续分支，等待所有分支都执行完毕后，后续分支才会触发。与同步模式的不同：允许与Gen.AND-join网关连接的前续分支产生多个执行线程，支持多个分支执行线程的同步合并。

如图A-19所示，新员工入职时，先去人力资源部报到，接下来填写入职资料和签订合同，于此同时，IT部门帮忙开通RTX邮件开发环境权限、初始化机器。初始化机器是一个多实例子流程，包含了多项工作：开通RTX、开通邮件、开通开发环境、安装软件、打包机器送至工位。只有这些工作都完成了，员工才能收拾工位开始工作。我们将IT部门的工作建模为多实例子流程，每一项工作都产生一个执行线程。



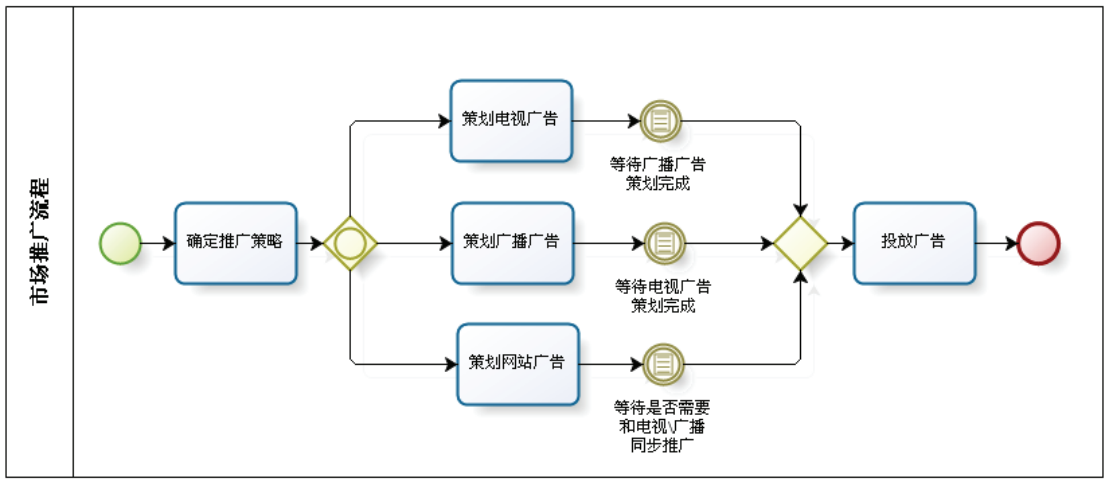
图A-19　泛化同步

本地同步合并（WCP_37: Local Synchronizing Merge）

描述

两个或多个分支合并为一个后续分支，该模式基于本地数据来决定需要同步的分支以及时机。

如图A-20所示，公司产品要进行市场推广，首先是确定推广策略：电视、广播和网站选择哪些渠道投放广告。如果选择电视和广播渠道，那么这两个渠道需要同步推广；如果选择了网站渠道，那么需要决定是否与电视广播渠道同步推广。我们是用条件事件决定需要同步的工作和时间。



图A-20　本地同步合并

通用同步合并（WCP_38: General Synchronizing Merge）

描述

两个或多个分支合并为一个后续分支，当所有被实际触发的分支都执行完毕并且未来也不会再被触发后才会触发后续分支的执行。该模式没有前提条件，不要求结构化，也不要求执行线程安全。

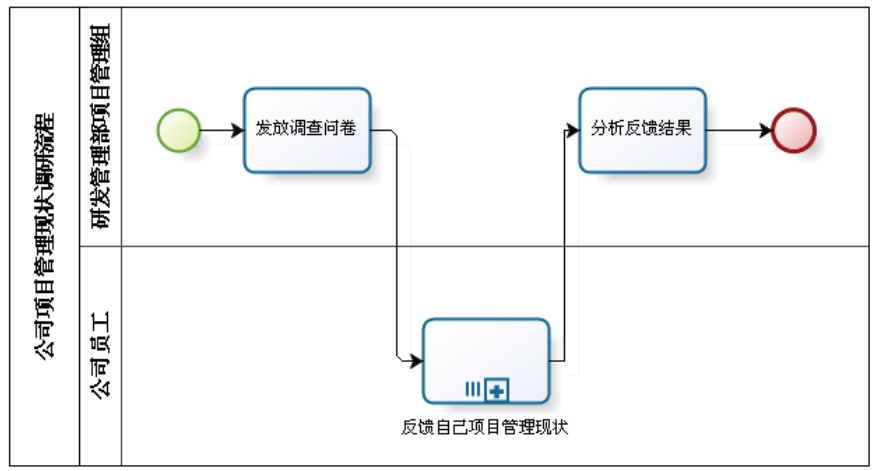
通用同步合并、结构化同步合并和本地同步合并模式解决的问题是一致的：对实际触发的前续分支进行同步合并。区别在于实现同步合并的机制：结构化同步合并通过建模的结构化判断需要同步的分支，本地同步合并基于本地数据分析实现同步合并，通用同步合并通过对流程实例的全面分析确定不再有需要合并的分支，触发流程的向后流转。

多线程合并（WCP_41: Thread Merge）

描述

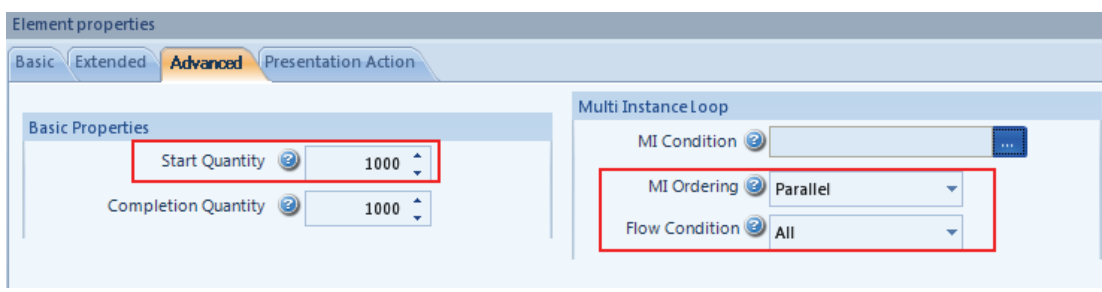
在流程的一个特定点，将一个分支的多个执行线程合并为一个执行线程。合并线程的数目在定义期确定。

如图A-21所示，公司计划对项目管理现状进行一次调研，将调查问卷发送到1000名员工手中，当所有人反馈后对反馈情况进行分析。



图A-21　多线程合并

我们使用多实例子流程对员工反馈情况进行建模，如图A-22所示，启动数量表明我们需要1000名员工的反馈，为每位员工启动了一个子流程实例；多实例循环顺序为并行，表明这些子流程实例并行执行；流转条件是All，表明需要将所有子流程实例都合并即所有员工都提交反馈后再触发后续分析活动。



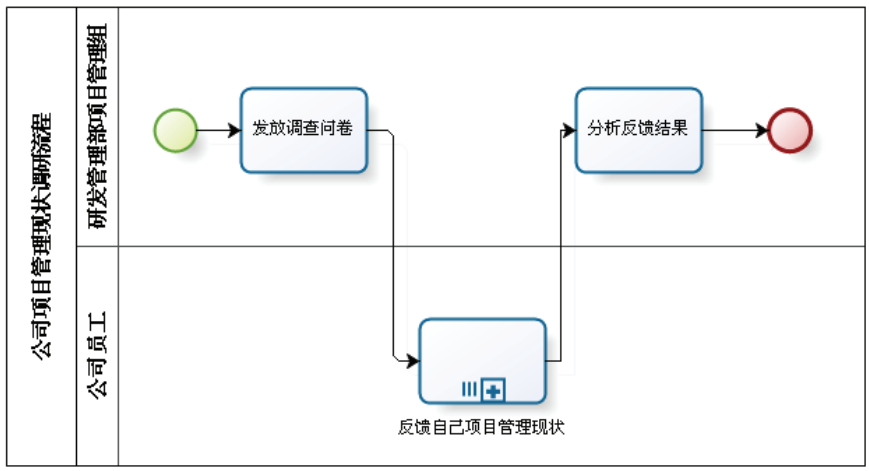
图A-22　使用多实例子流程合并多线程

多线程分裂（WCP_42: Thread Split）

描述

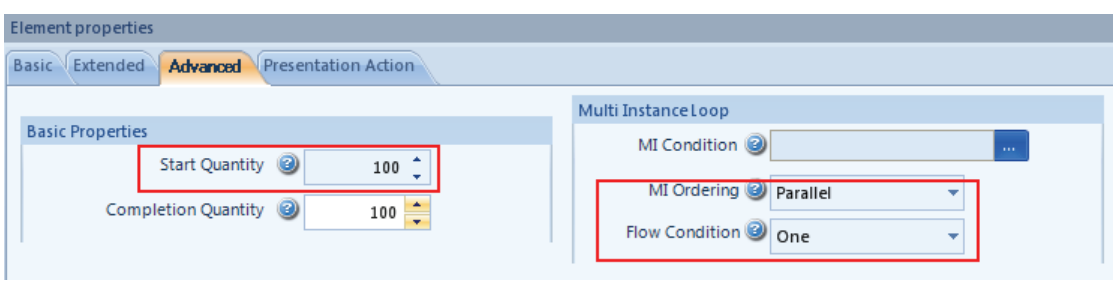
在流程的一个特定点，为一个分支初始化多个执行线程。执行线程的数量在定义期确定。

如图A-23所示，公司计划对项目管理现状进行一次调研，将调查问卷发送到100名员工手中，一旦有反馈就对反馈情况进行分析。



图A-23　多线程分裂

我们使用多实例子流程对员工反馈情况进行建模，如图A-24所示，启动数量表明我们需要100名员工的反馈，为每位员工启动了一个子流程实例；多实例循环顺序为并行，表明这些子流程实例并行执行；流转条件是One，表明任何一个子流程实例完成即任一员工有反馈后都会触发后续的分析活动。



图A-24　使用多实例子流程产生多线程

小结

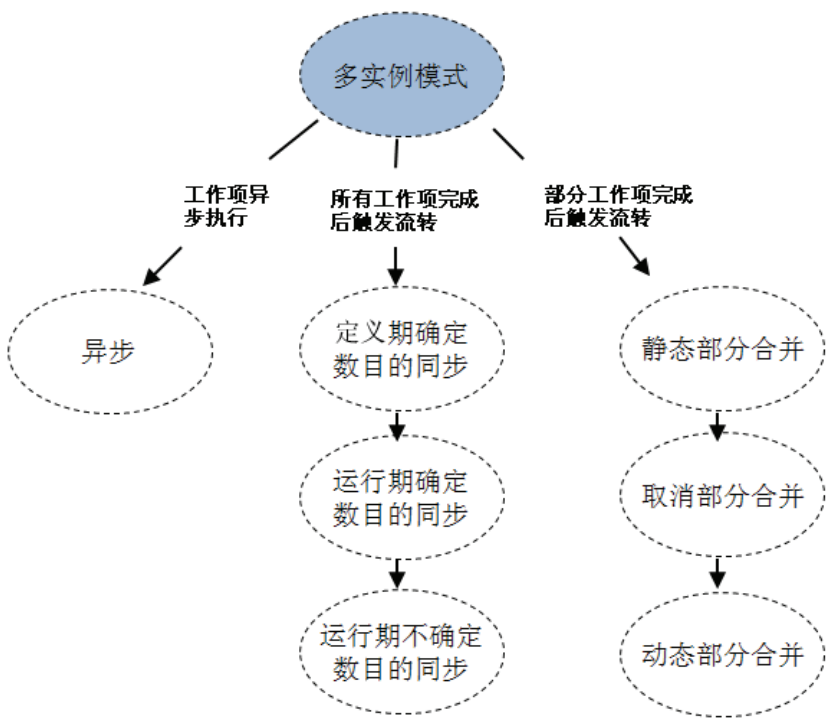
业务人员所建立的流程模型很难被直接执行，原因在于这些模型是对业务的直接描述，流程存在的目的在于沟通而不是机器执行。在本节所描述的模式里，很多模式实现起来非常困难，比如说非结构化的模式、非执行线程安全的模式，但是这些情况在实际业务中非常常见，原因就在于这些业务的处理者是人，人能够基于整个流程的执行情况具体情况具体分析，而工作流引擎则必须基于明确的输入进行计算，此时就需要系统分析人员在业务语言与计算机语言间进行转换。没有工作流产品能对上述所有模式进行支持，都有建模的各种约束。选择工作流产品时，很重要的一点就是看其所支持的模式是否与当前的业务相契合或者说做到最大程度上的适配。

同时，在应用工作流产品时，通过在流程定义里增加信息能够显著减少引擎运行期计算的难度，例如在应用结构化的合并模式时，我们在定义合并网关时就将与之对应的分裂网关信息保存到了它的属性定义中，避免运行期的计算。

多实例模式

在一个流程实例里，当一个活动存在多个工作项或活动实例时，我们称之为多实例。多实例产生于3种情况：活动在触发时产生多个工作项；活动在流程实例中被触发多次，产生了多个活动实例（循环、多实例合并）；两个或多个活动具有相同的工作内容，这些内容重复的活动被抽离成块活动或子流程多次执行。

多实例模式共有7种，如图A-25所示。



图A-25　多实例模式

- 异步多实例：活动创建多个工作项，这些工作项彼此独立同时执行，不需要同步。
- 定义期确定数目的同步多实例：活动创建多个工作项，创建数目在定义期建模时确定，这些工作项彼此独立同时执行，当它们都执行完毕后才触发后续活动的执行。
- 运行期确定数目的同步多实例：与定义期确定数目的同步多实例区别：创建工作项的数目在运行期、活动创建工作项前确定。

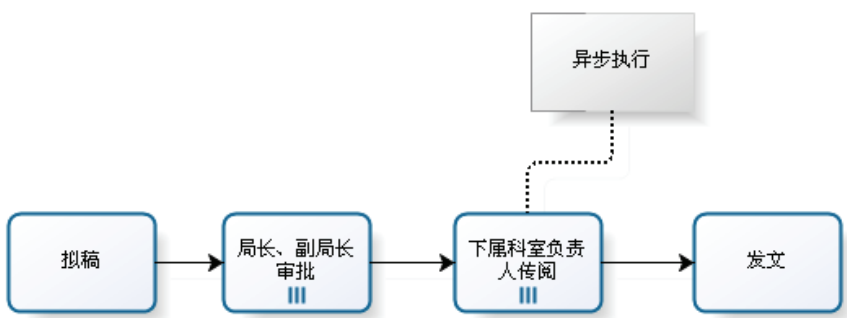
- 运行期不确定数目的同步多实例：活动创建多个工作项，创建数目依赖于运行时的一系列因素，这些因素包括流程实例状态、可用的资源、外部环境等，在最后一个工作项执行完成前都有可能产生新的工作项，这些工作项彼此独立同时执行，所有工作项都执行完毕后才触发后续活动的执行。
- 多实例的静态部分合并：扩展运行期确定数目的同步多实例模式。创建工作项的数目M在活动创建前确定，同时确定的还有必须完成的实例数目N，N < M。当N个工作项完成后即触发后续活动的执行，剩余的工作项继续执行，但是被忽略。
- 多实例的取消部分合并：与多实例的静态部分合并区别：当第N个工作项完成触发后续活动后，剩余工作项被取消不再执行。
- 多实例的动态部分合并：扩展运行期不确定数目的同步多实例模式，增加活动的完成条件，只要满足条件就触发后续活动，剩余的工作项继续执行，但被忽略。

异步多实例（WCP_12: Multiple Instances without Synchronization）

描述

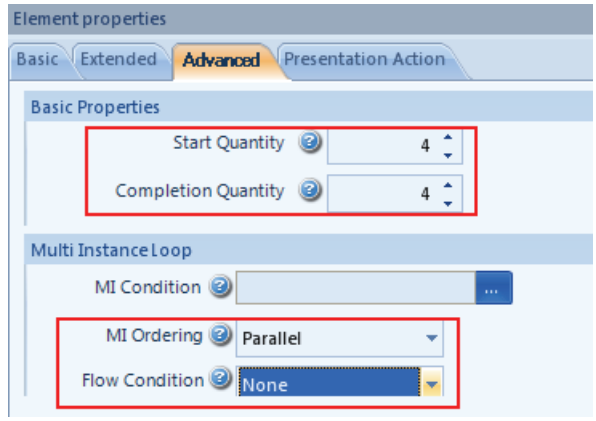
在一个流程实例里，一个活动创建多个工作项，这些工作项彼此独立同时执行，不需要同步。活动创建工作项的数目在定义期确定，是一个固定值。

如图A-26所示， 某单位的发文流程，拟稿后需要局长和副局长审批，然后下发到各科室负责人进行传阅，传阅这个行为不影响流程实例本身的执行，触发传阅活动后即触发发文活动。



图A-26　异步多实例

我们在活动定义期确定活动产生工作项的数目，如图A-27所示，下属有4个科室，所以我们设置启动数量为4，为每位科室负责人生成一个工作项；工作项的循环顺序为并行，表明这些工作项并行执行；流转条件是None，表明生成完工作项后就触发后续活动。



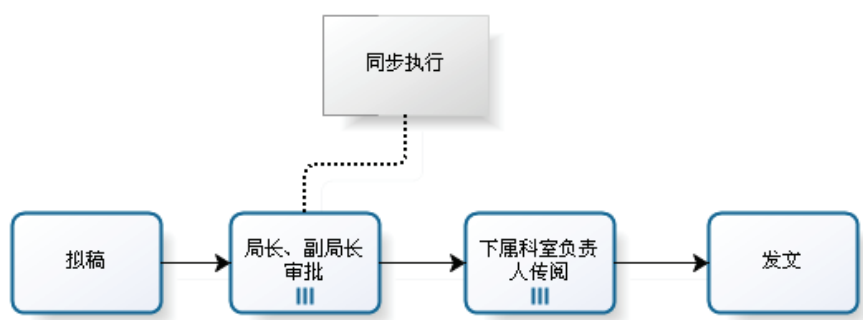
图A-27　定义期确定活动产生工作项的数目、异步执行

定义期确定数目的同步多实例（WCP_13: Multiple Instances with a Priori Design-Time Knowledge）

描述

在一个流程实例里，一个活动创建多个工作项，创建数目在定义期建模时确定，这些工作项彼此独立同时执行，当这些工作项都执行完毕后才触发后续活动的执行。

如图A-28所示，某单位的发文流程，拟稿后需要局长和副局长审批，必须都审批通过后才能下发到各科室负责人进行传阅。



图A-28　定义期确定数目的同步多实例

我们在活动定义期确定活动产生工作项的数目，如图A-29所示，有2个领导，所以我们设置启动数量为2，为每位领导生成一个工作项；工作项的循环顺序为并行，表明这些工作项并行执行；流转条件是All，表明所有工作项都执行完成后才能触发后续活动。

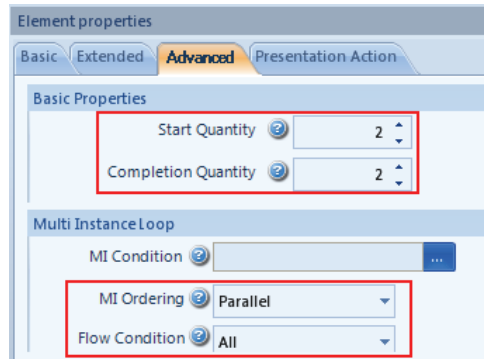


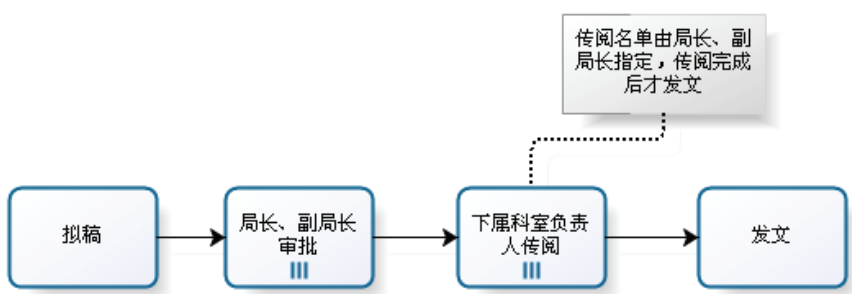
图 A-29　定义期确定活动产生工作项的数目

运行期确定数目的同步多实例（WCP_14: Multiple Instances with a Priori Run-Time Knowledge）

描述

在一个流程实例里，一个活动创建多个工作项，创建数目在运行期确定，这些工作项彼此独立同时执行，当这些工作项都执行完毕后才触发后续活动的执行。

如图A-30所示。传阅的名单由局长、副局长在审批时指定。



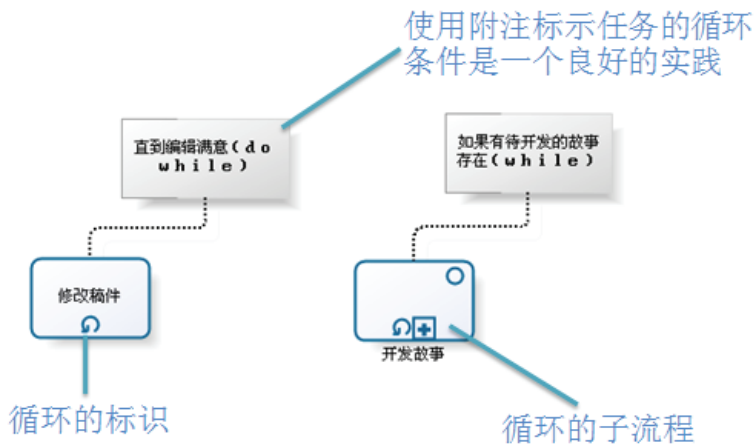
图A-30　运行时确定数目的同步多实例

运行期不确定数目的同步多实例（WCP_15: Multiple Instances without a Priori Run-Time Knowledge）

描述

在一个流程实例里，一个活动创建多个工作项，创建数目依赖于运行期的一系列因素，这些因素包括流程实例状态、可用的资源、外部环境变化等，在最后一个工作项执行完成前都有可能

产生新的工作项，这些工作项彼此独立同时执行，所有工作项都执行完毕后才触发后续活动的执行，如图A-31所示。



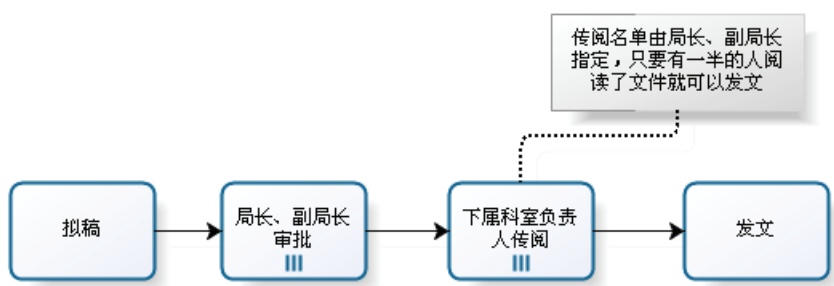
图A-31　运行期不确定数目的同步多实例

多实例的静态部分合并（WCP_34: Static Partial Join for Multiple Instances）

描述

在一个流程实例里，一个活动创建多个工作项，创建数目M在运行期确定，同时确定的还有必须完成的工作项数目N，N <= M，这些工作项彼此独立同时执行，当N个工作项完成后即触发后续活动的执行，其余的工作项继续执行，但是被忽略，不影响流程的路由。

如图A-32所示。传阅的名单由局长、副局长在审批时指定，只要有一半的人阅读了文件没有意见就可以发文。



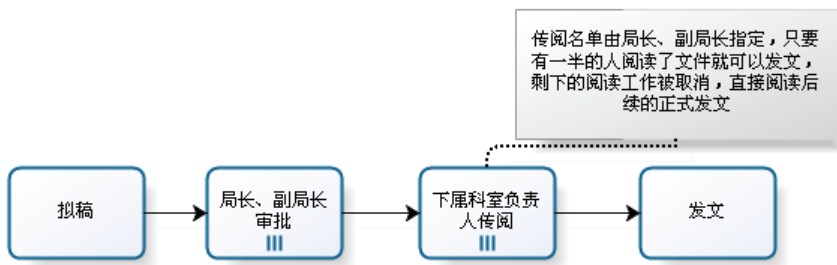
图A-32　多实例的静态部分合并

多实例的取消部分合并（WCP_35: Cancelling Partial Join for Multiple Instances）

描述

在一个流程实例里，一个活动创建多个工作项，创建数目M在运行期确定，同时确定的还有必须完成的工作项数目N，N <= M，这些工作项彼此独立同时执行，当N个工作项完成后即触发后续活动的执行，其余的工作项被取消。

如图A-33所示。传阅的名单由局长、副局长在审批时指定，只要有一半的人对文件没有意见就可以发文，剩下的传阅工作被取消，直接阅读正式文件。



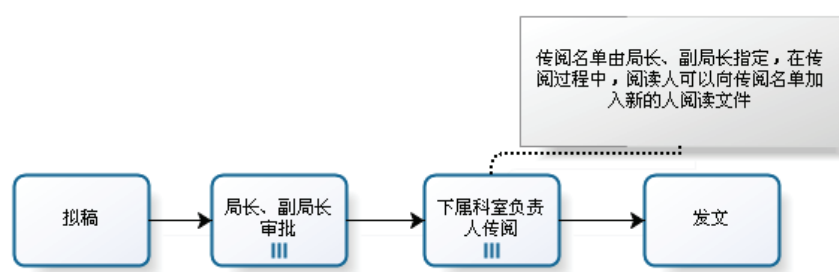
图A-33　多实例的取消部分合并

多实例的动态部分合并（WCP_36: Dynamic Partial Join for Multiple Instances）

描述

在一个流程实例里，一个活动创建多个工作项，创建数目依赖于运行期的一系列因素，这些因素包括流程实例状态、可用的资源、外部环境变化等，在最后一个工作项执行完成前都有可能产生新的工作项，这些工作项彼此独立同时执行，每个工作项完成时就对活动的完成条件进行验证，如果满足该条件那么就触发后续的活动,剩余的工作项将继续执行但被忽略,不再产生新的工作项。

如图A-34所示。传阅的名单由局长、副局长在审批时指定，阅读人在阅读过程中可以加入新的人阅读文件，只要名单中有一半的人阅读了文件并没有意见就可以发文。



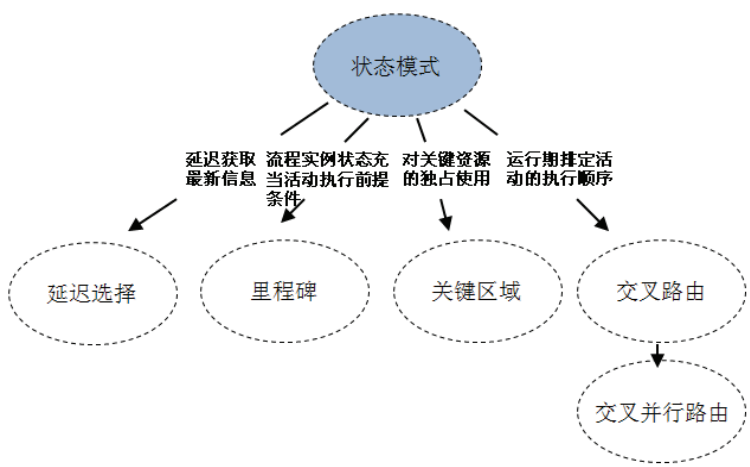
图A-34　多实例的动态部分合并

状态模式

当前流程实例的状态会影响流程实例的后续执行。这里的状态包括了当前流程实例正在执行的活动数量、内容、时间，正在执行活动的状态（挂起、超时）、与流程实例相关的数据以及当前资源状态等。状态模式讨论当前流程实例状态对流程实例后续执行所产生的影响。

状态模式共有5种，如图A-35所示。

- 延迟选择：当需要在多个分支中选择一个分支实际执行时，这个决定被尽可能的延后，以获得最新最充分的信息。
- 交叉并行路由：一系列的活动需要执行，在同一时间，只允许一个活动被执行。这些活动之间部分存在顺序，剩余活动任意执行，它们的顺序在运行时决定。
- 里程碑：只有当流程实例处于某一特定状态（里程碑）时，特定活动才有可能被激活。流程实例的特定状态充当活动执行的前提条件。
- 关键区域：两个或多个由互相连接活动构成的区域被标识为关键区域。在同一时间，这些关键区域只有一个能够激活执行，标示对资源的独占使用。
- 交叉路由：一系列的活动需要执行，在同一时间，只允许一个活动被执行。与交叉并行路由模式的区别：活动之间完全不存在任何预定的顺序，任意执行，在运行时决定它们的顺序。



图A-35　状态模式

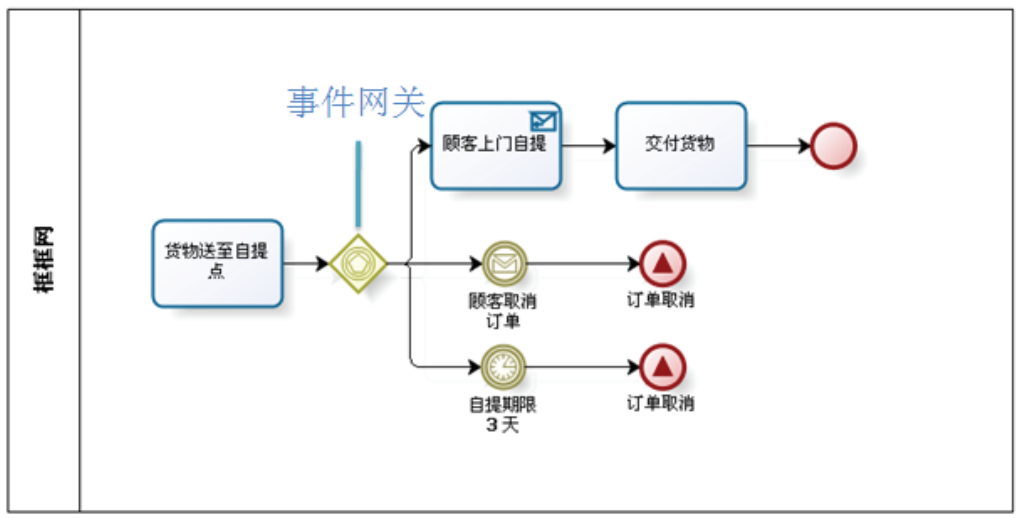
延迟选择（WCP_16: Deferred Choice）

描述

流程实例在某个点有多个分支可供选择，只能有一个分支被实际执行。

延迟选择与XOR-split的区别是：选择并不是在后续分支被触发之前，相反，这个决定被尽可能的延后，每个分支都有可能被执行，具体执行哪个分支取决于具体的流程实例运行环境。

如图A-36所示，某电商的货物自提流程，三个分支都被触发，一旦有一种分支情况发生，其他的分支将被取消。选择被延迟到第一个分支开始实际执行之前。



图A-36　延迟选择

同义词

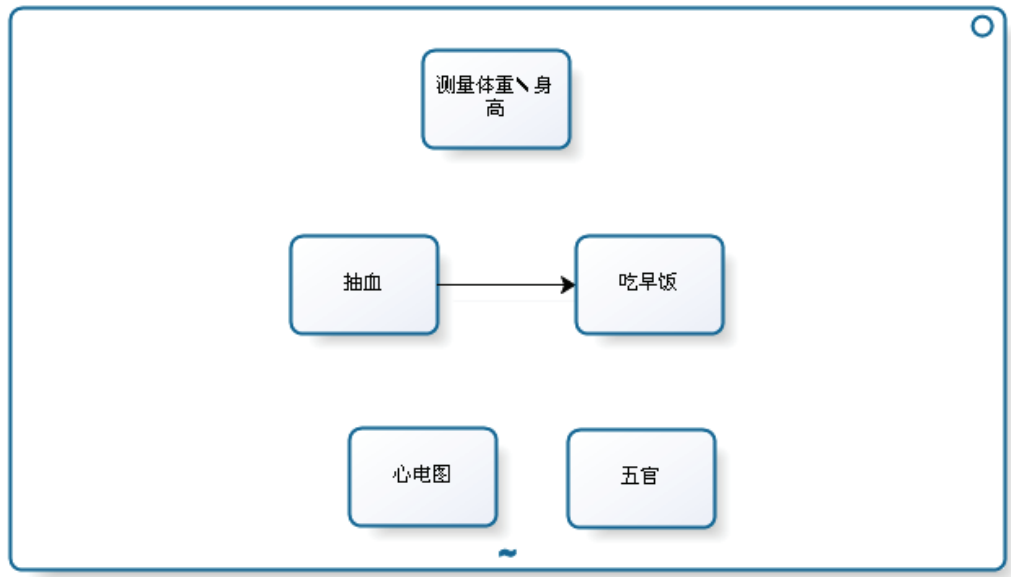
隐式选择、延迟XOR-split。

交叉并行路由（WCP_17: Interleaved Parallel Routing）

描述

一系列的活动需要执行，在同一时间，只允许一个活动被执行。与顺序模式不同，这些活动之间可能部分存在顺序，剩余活动则任意执行，它们的顺序在运行时决定。所有活动都执行完毕后触发后续活动的执行。

如图A-37所示，去医院检查身体，需要做常规检查、抽血和吃早饭，抽血一定要在吃早饭之前，其他项目则没有限制。那么可能的顺序有：常规检查→抽血→吃早饭（抽血处排队严重）、抽血→吃早饭→常规检查，以及抽血→常规检查→吃早饭。



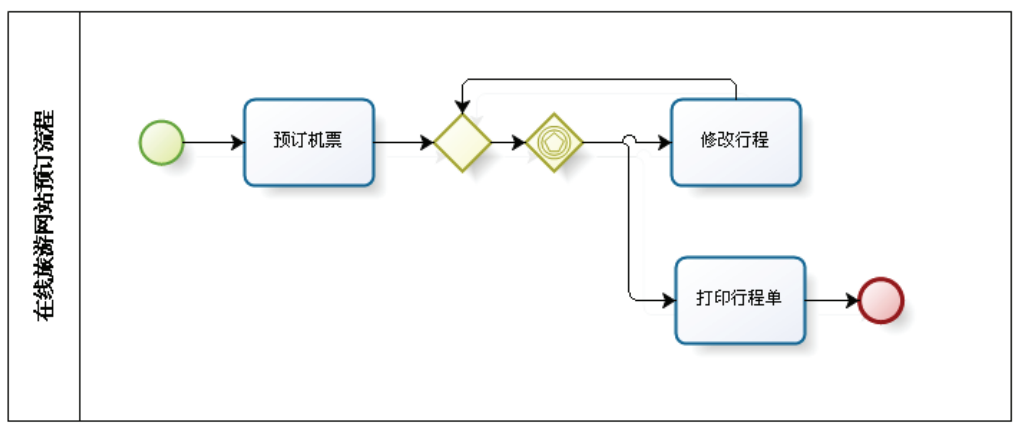
图A-37　交叉并行路由

里程碑（WCP_18: Milestone）

描述

只有当流程实例处于某一特定状态时，特定活动才有可能被激活。

如图A-38所示，我们去在线旅游网站预订行程，只要没有打印最终的行程单，我们就可以修改行程。这里我们使用事件网关做分支的延迟选择。



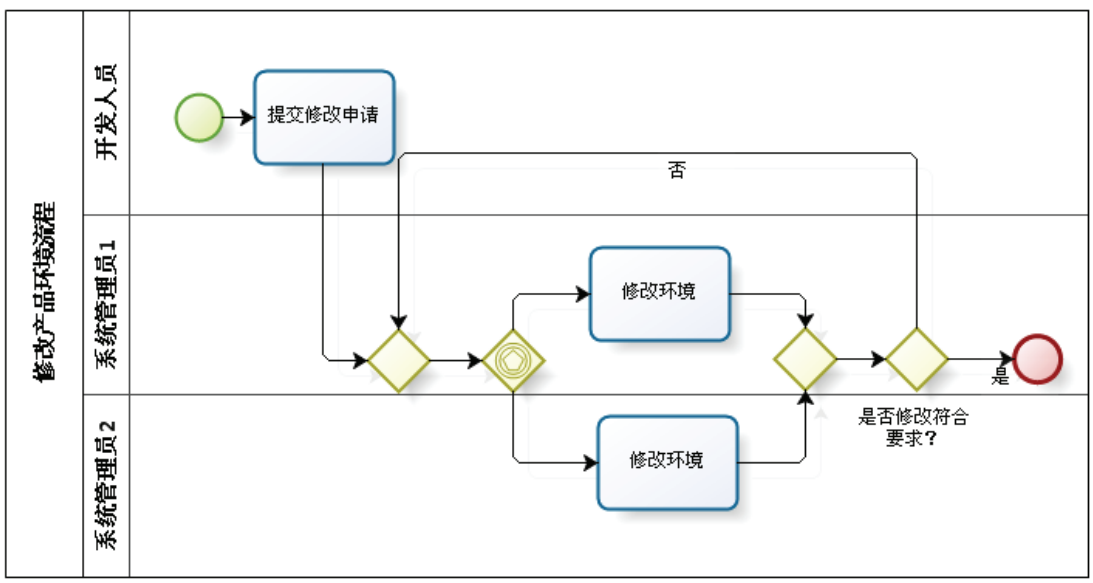
图A-38　里程碑

关键区域（WCP_39: Critical Section）

描述

两个或多个由互相连接活动构成的区域被标识为关键区域。在流程实例执行的任何时刻，这些关键区域只有一个能够被激活执行，不能同时执行。保证活动对某资源的独占式使用，该资源是物理资源、人力资源也可以是业务系统。

如图A-39所示，修改产品环境的流程，在任何时刻，都只能有一个系统管理员对环境进行修改。这里我们使用事件网关做分支的延迟选择。



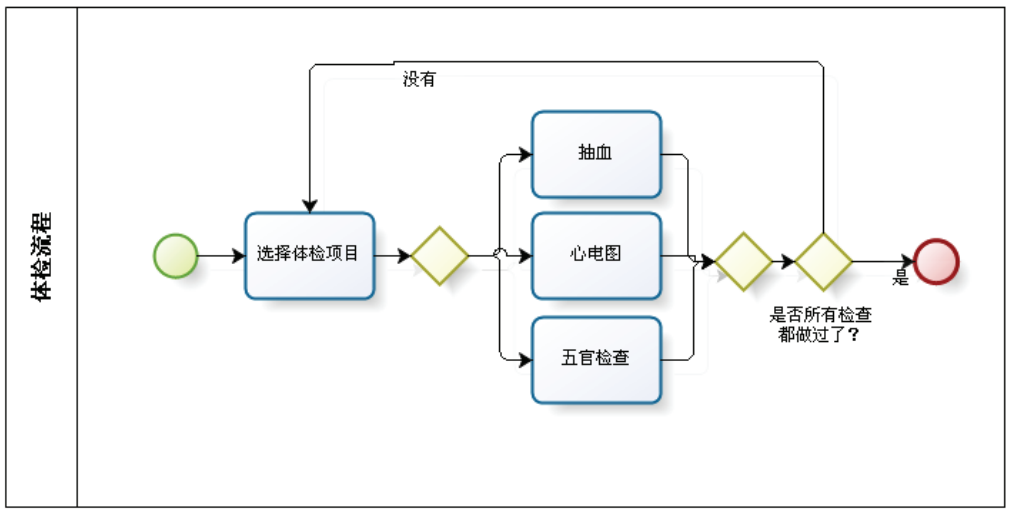
图A-39　关键区域

交叉路由（WCP_40: Interleaved Routing）

描述

一系列的活动需要执行，在同一时间，只允许一个活动被执行。交叉路由与交叉并行路由模式的区别是：活动之间完全不存在任何预定的顺序，可以任意执行，在运行时决定它们的顺序。所有活动都执行完毕后触发后续活动的执行。

如图A-40所示，体检流程，我们根据自己的情况选择检查项目的顺序，当所有项目都检查完毕后体检就结束了。

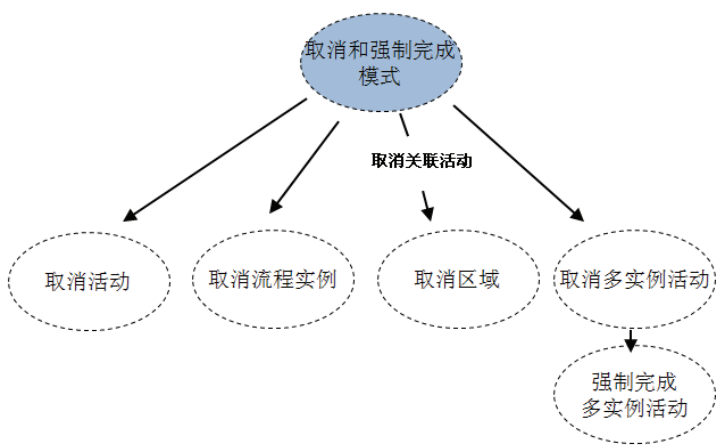


图A-40　交叉路由

取消和强制完成模式

流程实例执行的过程中，不免会产生异常情况，这些异常情况包括了超时、资源不可用、外部环境变化等，异常导致流程实例/活动执行的价值减少甚至浪费，在这种情况下，就涉及流程实例/活动执行的取消。

取消和强制完成模式共有5种，如图A-41所示。



图A-41　取消和强制完成模式

- 取消活动：取消流程实例中某一活动的执行。

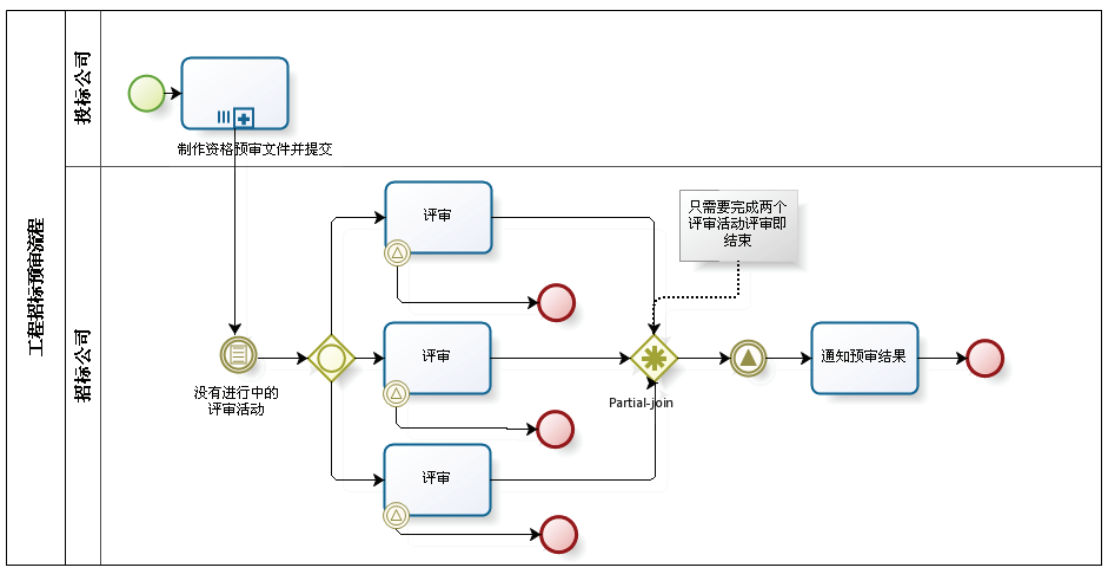
- ❑ 取消流程实例：取消整个流程实例的执行。
- ❑ 取消区域：取消某一区域里所有正在执行的活动。
- ❑ 取消多实例活动：取消流程实例中某一多实例活动的执行。
- ❑ 强制完成多实例活动：强制完成流程实例中某一多实例活动的执行，强制流程向后流转。

取消活动（WCP_19: Cancel Task）

描述

取消一个已激活的活动，如果已经开始执行，那么停止执行，并且可能的话，移除正在运行的工作项。

如图A-42所示，工程招标预审流程，招标公司进行资格的预审，需要两位评委完成评审评审才结束，评审结果里只要有一个不通过则预审不通过。剩下一位评委的评审活动被取消。



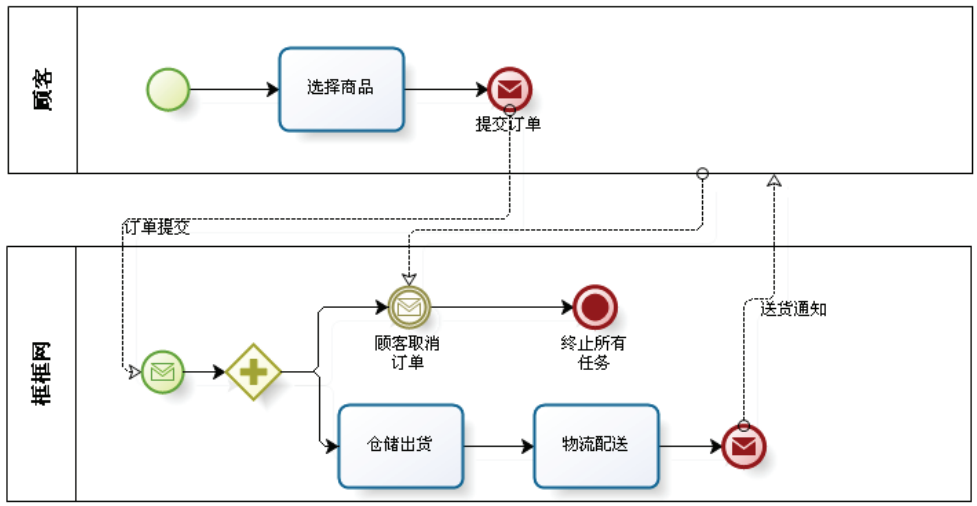
图A-42　取消活动

取消流程实例（WCP_20: Cancel Case）

描述

取消一个正在执行中的流程实例。该流程实例被标识为未成功完成。

如图A-43所示，订单处理过程中顾客突然取消订单，那么停止该流程实例的执行。我们使用终止结束事件结束流程实例。



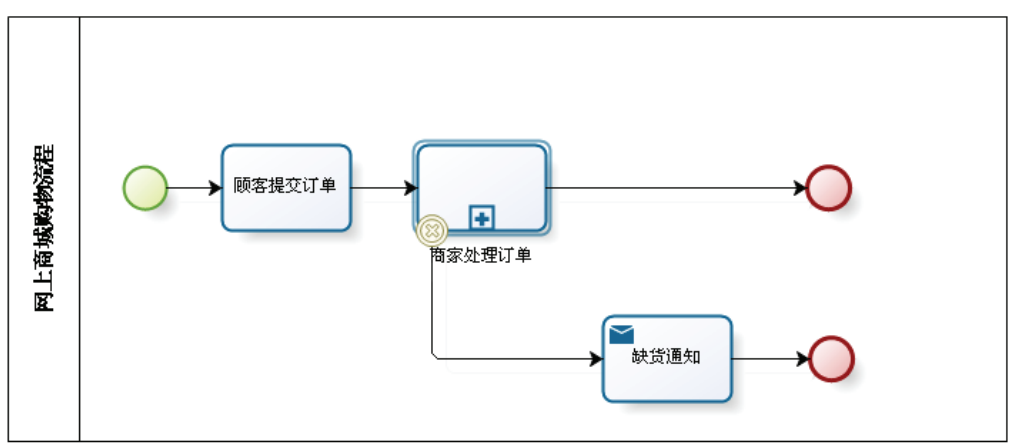
图A-43　取消流程实例

取消区域（WCP_25: Cancel Region）

描述

一系统的活动建模成一个区域，流程实例执行时，可以对整个区域进行取消，区域中正在执行或处于激活状态的活动被取消。这些活动可以位于不同分支上并互不连接。

如图A-44所示，双十一期间，很多商家处理订单的过程中发现缺货，于是取消整个订单处理的子流程，给顾客发送消息通知。



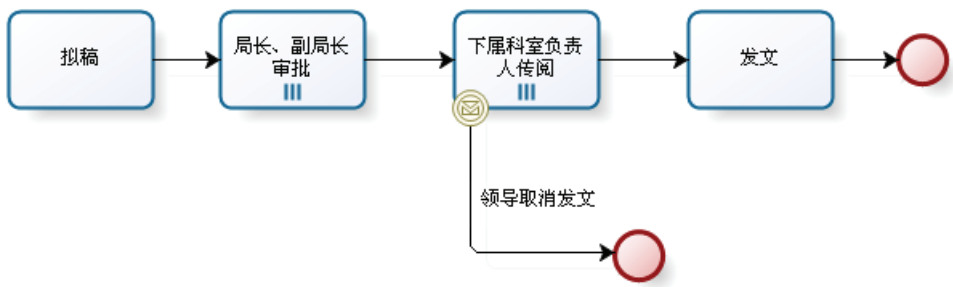
图A-44　取消区域

取消多实例活动（WCP_26: Cancel Multiple Instance Activity）

描述

在一个流程实例里，可以对多实例活动进行取消：未完成的工作项被取消，已经完成的工作项不受影响。

如图A-45所示，领导可以在发文传阅过程中取消发文，此时阅读文件的工作被取消，已经阅读过的不受影响。



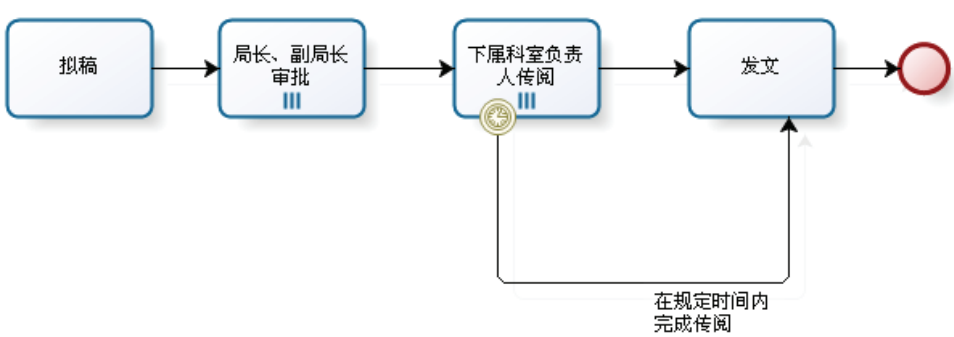
图A-45　取消多实例活动

强制完成多实例活动（WCP_27: Complete Multiple Instance Activity）

描述

在一个流程实例里，可以对多实例活动强制完成：未完成的工作项被取消，已经完成的工作项不受影响，后续活动被触发。

如图A-46所示，领导要求科室负责人在当天完成文件的传阅，第二天早上进行正式发文，此时阅读文件的工作被取消，已经阅读过的不受影响。



图A-46　强制完成多实例活动

小结

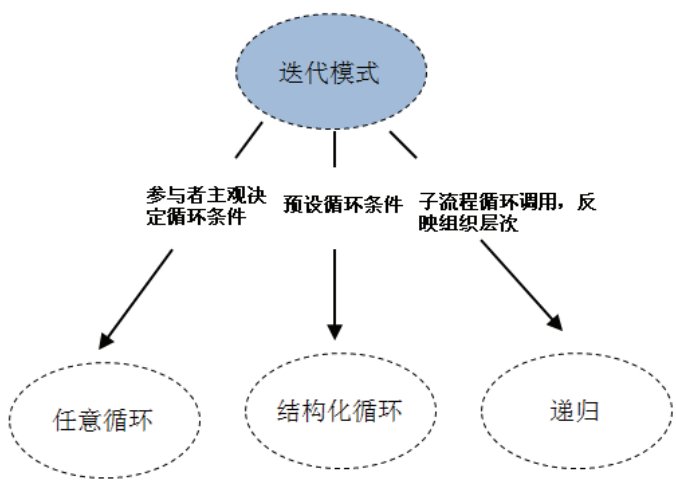
取消流程实例/活动经常与工作流异常相关，此时执行过的工作已经产生了一定的影响，为了使流程实例能够继续执行（或通过其他路径继续执行）或正常停止，往往需要对已执行工作所产生的影响进行消除，这就需要通过恢复动作（回滚与业务补偿）完成，我们将在附录D工作流异常模式里详细讨论这些情况。

迭代模式

在流程实例执行时，因为各种原因，我们需要重复执行一些活动或路径。迭代模式讨论流程实例里的重复行为。

迭代模式共有3种，如图A-47所示。

- ❑ 任意循环：能够在流程里建立有多个入口和出口的循环。
- ❑ 结构化循环：能够重复执行活动或子流程。循环只有一个单一的入口和出口。
- ❑ 递归：活动在执行时能够递归调用它自己。



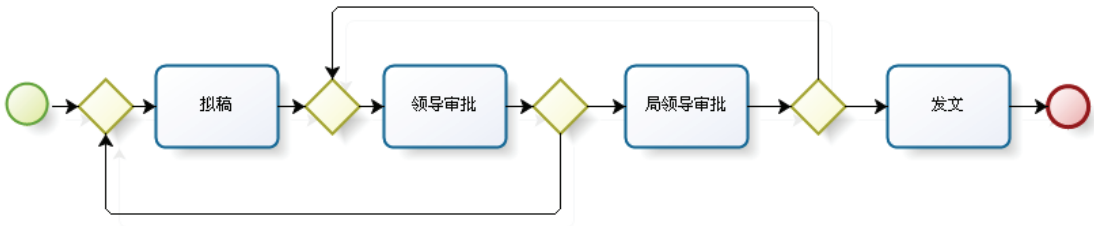
图A-47　迭代模式

任意循环（WCP_10: Arbitrary Cycles）

描述

流程里的循环具有多个入口和出口。

如图A-48所示，发文流程，如果有领导审批不通过，就返回上一活动重新修改，这样多次往复修改直到领导满意为止。



图A-48　任意循环

同义词

非结构化循环。

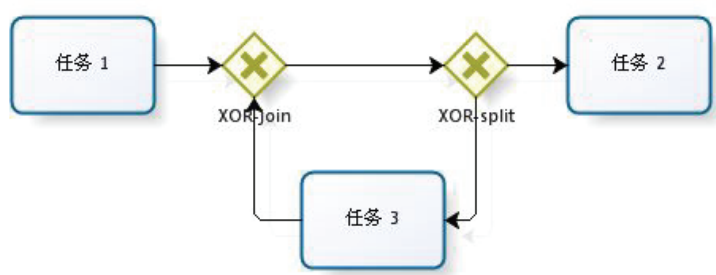
应用

该模式类似于程序里的“goto”，虽然很自然，但是给维护带来噩梦。在我们的项目里，当用户第一次使用工作流系统进行流程建模时，最常见的情况就是任意循环大量出现，因为这种模式最适合用户的自然思维，另一方面则是因为很多公司的流程人为干预太过严重，用户甚至要求能够从任意活动回退到任意前续活动。我们建议这一模式尽量少用。

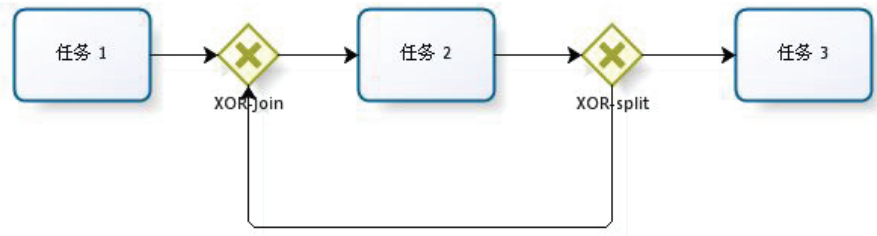
结构化循环（WCP_21: Structured Loop）

描述

能够重复执行活动或子流程。循环只有单一的入口和出口。该模式有图A-49和图A-50所示的两种形式。



图A-49　结构化循环：while



图A-50　结构化循环：do while

前提条件

同一时间只能有一个该循环的实例运行。

应用

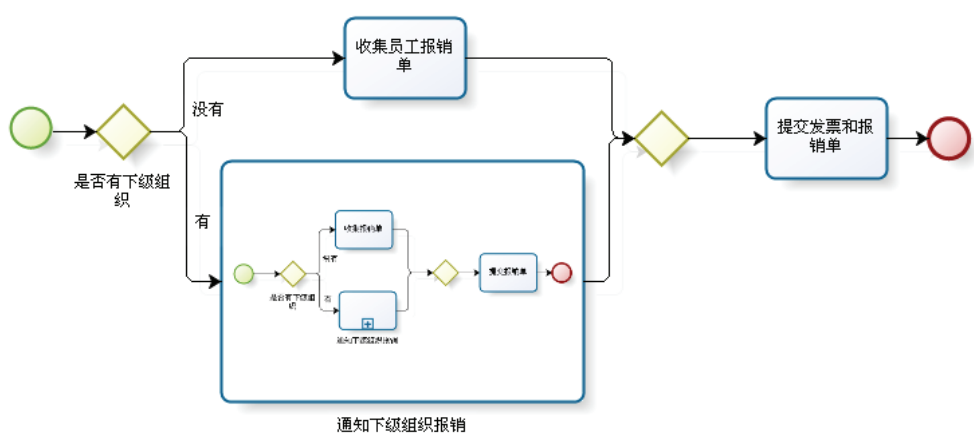
重复的执行活动直至满足一定的条件为止。与任意循环模式相比，该模式更加贴合计算机语言，也更加容易被工作流系统所支持。

递归（WCP_22: Recursion）

描述

活动在执行时能够递归调用它自己。该模式反映出流程执行的层次性。

如图A-51所示，每个月底，公司统一对所有出差费用进行报销，财务部门发起一个报销流程，同时，作为对应，每个部门都发起一系列的报销活动，这些报销活动具体到每个项目团队，每个团队也都执行一系列的报销活动，这样就构成了一个多层次的递归调用关系，这个层次关系是组织管理层次的映射。



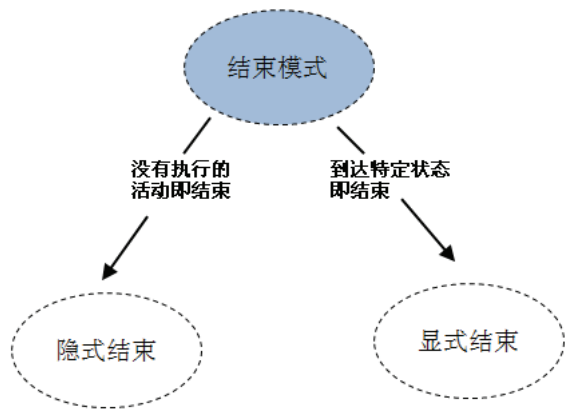
图A-51　递归

结束模式

结束模式讨论什么情况下流程实例执行结束。

结束模式共有2种，如图A-52所示。

- ❑ 隐式结束：没有活动执行，流程实例即算结束。
- ❑ 显式结束：流程实例到达某个状态即算结束。



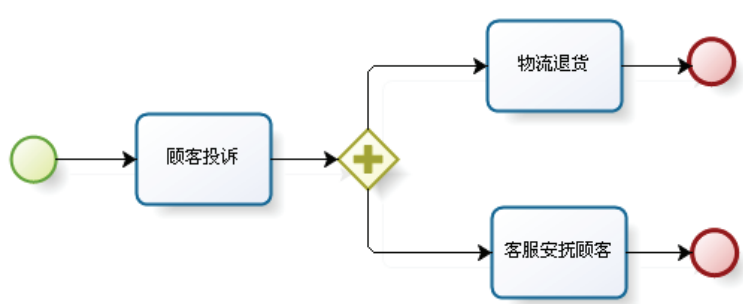
图A-52　结束模式

隐式结束（WCP_11: Implicit Termination）

描述

当流程实例中所有的活动都执行完毕，不会产生新的执行活动，且流程实例没有死锁，那么该流程实例就算成功结束了。

如图A-53所示，公司受到顾客投诉要求退货，一方面我们需要尽快退货，另一方面我们需要安抚顾客找出他不满意的原因以便进一步改进，这两项工作都完成了流程就结束。



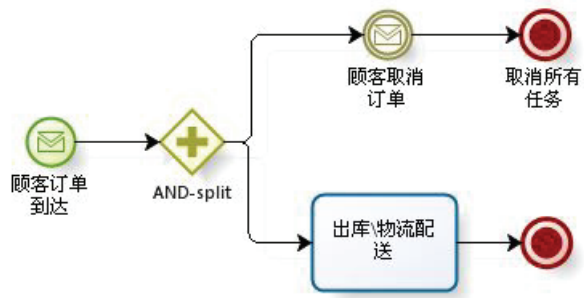
图A-53　隐式结束

显式结束（WCP_43: Explicit Termination）

描述

当流程实例到达某个状态即意味着流程实例执行结束了，我们使用终止结束事件表示这个状态。流程定义允许存在多个终止结束事件，只要到达其中一个终止结束事件流程实例即告结束，剩余未完成的活动被取消。

如图A-54所示，不管是顾客取消订单还是物流配送完成，订单处理流程都完成。

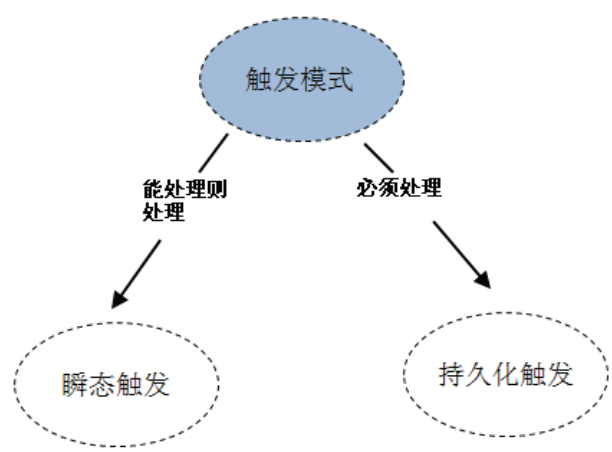


图A-54　显式结束

触发模式

在流程实例执行过程中，我们总是会受到各种因素的影响，这其中就包括了组织外部的影响。触发模式讨论外部环境变化对流程实例执行的影响。

触发模式共有2种，如图A-55所示。



图A-55　触发模式

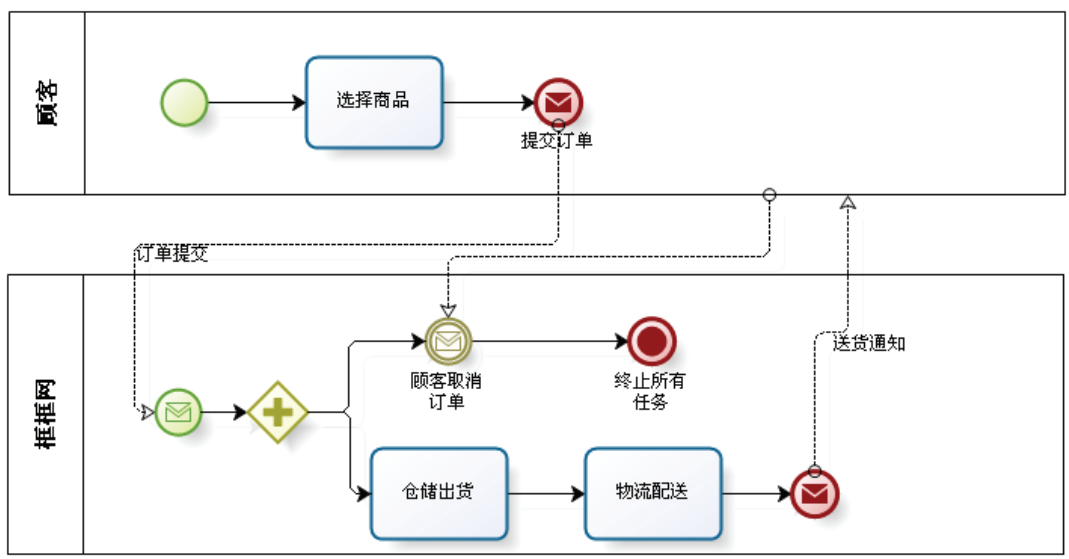
- ❑ 瞬态触发：外部事件触发流程实例做出反应，如果此刻流程实例不能处理，则丢弃。
- ❑ 持久化触发：外部事件触发流程实例做出反应，如果此刻流程实例不能处理，则持久化，后续处理。

瞬态触发（WCP_23: Transient Trigger）

描述

活动能够被外部环境发出的信号触发，这些信号表现为消息或事件。事件是瞬态的，如果此时流程实例能够对事件做出反应那么迅速处理，否则丢弃。流程实例只有处于一定的状态才能对外部事件进行处理。

如图A-56所示，网上商店购物，在货物配送之前都可以取消订单，一旦配送则不能取消。



图A-56　瞬态触发

持久化触发（WCP_24: Persistent Trigger）

描述

活动能够被外部环境发出的信号触发，这些信号表现为消息或事件。事件被持久化，如果此时流程实例能够对事件做出反应那么迅速处理，否则等待，直至被处理。

如图A-57所示，装修公司施工前一定会等待顾客确认设计方案。

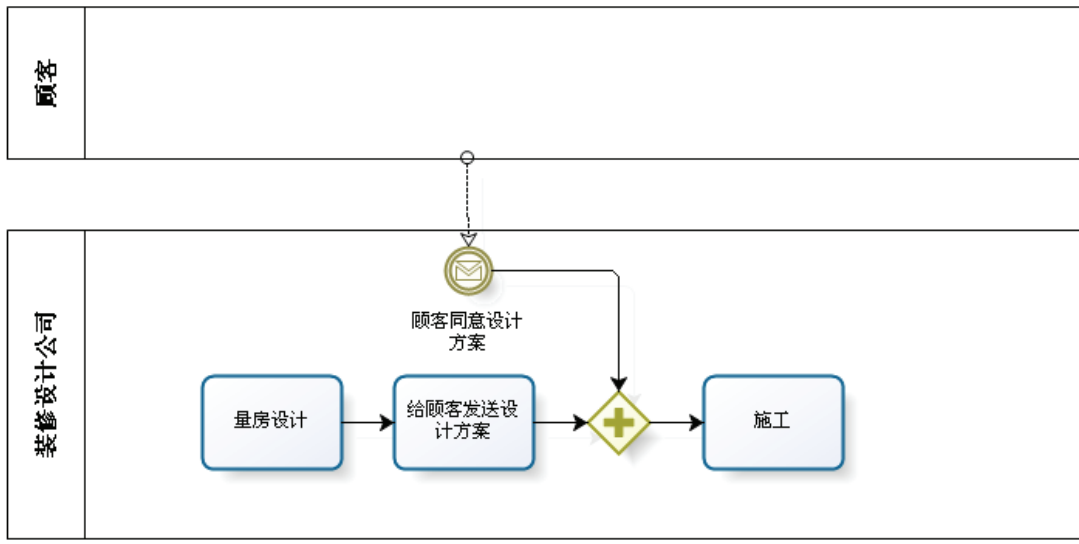


图 A-57　持久化触发

附录B
工作流资源模式

组织结构涉及两个基本要求：一方面要把某个创造价值的活动拆分成不同的活动，另一方面又要将各项活动协调整合起来，以便实现最终目标。在附录A里，我们讨论了工作流控制模式，即组织结构的第一个基本要求，强调对业务流程进行建模，将商业目标的实现根据组织所进行的工作和现有的技术体系拆分成一系列的活动。这里将接着介绍工作流的资源模式，活动协调的本质其实是组织内部资源的协调，即满足组织机构的第二个基本要求。活动的执行需要资源，资源的协调对业务流程执行的效率非常重要，对顾客而言，流程执行的时间越短则越有效率。什么是资源呢？人是最重要的资源，除了人之外，还有其他的非人力资源，如机器、设备、计算机等。资源最根本的特征是：*它能够执行特定的活动*。

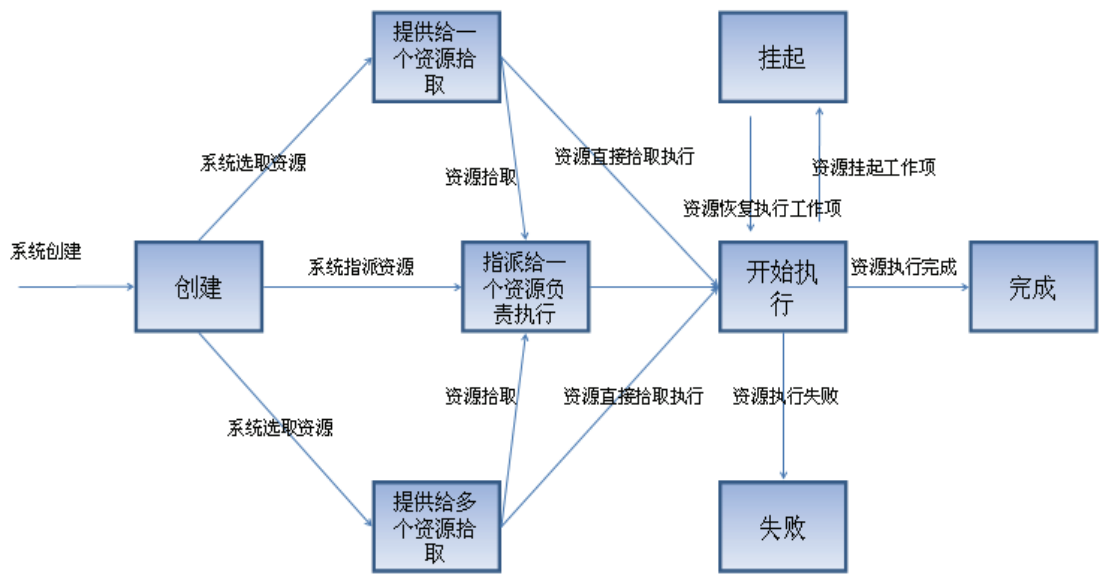
一个资源只能执行有限种类的活动。反过来，一个活动通常也只能被有限种类的资源所执行，这涉及资源的分组。组织对资源的分组具有多种形式，最常见的是部门和角色：部门体现资源在组织中的位置，角色体现资源的业务职能。对跨国跨地域的组织而言，还有地域机构的划分，此外还有临时组，例如以交付为核心的软件开发公司里的项目组。

为保证流程中每个活动都由合适的资源执行，我们在活动建模时定义分配规则，该规则说明执行该活动资源所需要符合的前提条件和约束。分配规则可以直接指定到特定资源（直接分配到具体的人执行），也可以是多个资源分组的交集（例如销售部的经理），还可能依赖于活动/流程实例本身属性（例如对于北京地区提交的货物订单会交由北京配送中心进行处理），此外，还有其他一些考虑，例如为安全考虑，银行任何职员不能执行同一流程实例中两个连续的活动，会计学里称之为职能分离。我们会在后面的资源模式里详细讨论这些情况。

在工作流系统里，活动在运行期被映射为工作项（work item），活动的执行被映射为工作项的执行。一般情况下，一个活动在一个流程实例中只对应一个工作项，但是存在一个活动需要多

人完成的情况，这个时候一个活动就会对应着多个工作项。工作项是工作流系统中最小的工作单元，其代表着一个单一资源对某一活动的执行。工作流系统里，资源与工作项的交互通过工作项管理器进行管理，即我们通常所见的工作项列表（任务列表），我们通过这一列表拾取工作、处理工作以及管理工作的状态。

工作项的生命周期具有8个状态，分别是创建（Created）、提供给一个资源拾取（Offered Single Resource）、提供给多个资源拾取（Offered Multiple Resources）、指派给一个资源负责执行（Allocated）、执行（Started）、完成（Completed）、失败（Failed）和挂起（Suspended），如图B-1所示。



图B-1　工作项的生命周期

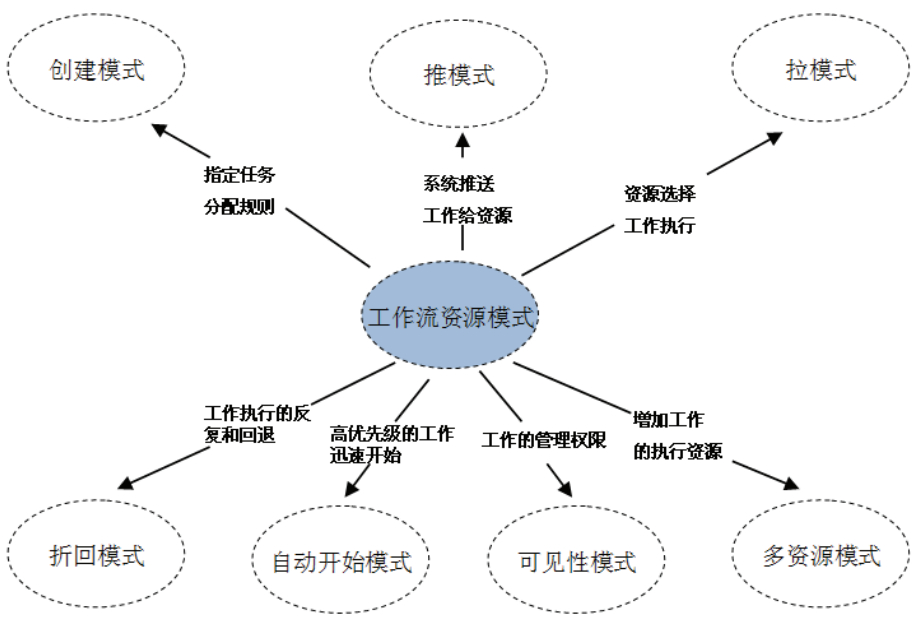
工作项被系统创建完毕后即处于创建状态，接下来系统会选取资源来执行该工作项。有两种状态：一种是被提供状态，一种是被指派状态，这两者的区别在于对资源来说一个是可选的一个是必须的。如果系统提供一个工作项给资源执行，这仅仅意味着资源符合执行该工作的前提条件，资源不必为该工作负责，即资源可以选择执行该工作也可以选择拒绝，资源只是该工作的合适候选者；而如果系统指派一个工作项给资源执行，则意味着资源必须为该工作负责，该工作必须由该资源执行。因为一个工作项只能由一个资源来执行，所以如果是指派的话，那么只能指定一个资源；而提供，则可以提供给一个资源也可以提供给多个资源来候选。工作项管理器提供两种列表来区分这两种状态，分别是可拾取列表和待办列表，一旦资源对可拾取列表里的工作项进行拾取，工作项即进入到资源的待办列表，状态成为被指派状态。

工作项进入被指派状态意味着执行该工作的资源已确定，那么接下来就可以由资源开始执行该工作，执行的过程中可以将工作暂时挂起中断处理，后续可以再恢复对该工作的执行。如果工

作成功完成，则工作项成为完成状态；如果工作因为各种原因没有成功完成，则工作项置为失败状态。

工作流资源模式共有43种，根据工作项所处的不同阶段以及状态变迁，分为7组，即创建模式、推模式、拉模式、折回模式、自动开始模式、可见性模式和多资源模式，如图B-2所示。

- 创建模式位于工作项生命周期的创建阶段，作为流程模型的一部分在流程定义期指定活动的分配规则，限定执行该活动的资源范围；
- 推模式将创建完毕的工作项与满足分配规则的资源进行匹配，将工作项推送给资源，资源本身不做出选择；
- 拉模式则是资源把工作项与自身进行匹配，考察其能够执行的工作项并选择执行，资源是主动的；
- 折回模式对应着由于各种原因所导致的工作项状态的反复和回退；
- 自动开始模式提供一种系统驱动工作项执行的方式，表明工作项的高优先级，需要马上开始执行；
- 可见性模式讨论各种不同资源对工作项的可见性，与管理权限相关；
- 多资源模式讨论一个资源执行多个工作项和多个资源执行同一个工作项的特殊情况。



图B-2　工作流资源模式的分类

分组是协调组织内部工作的一种不可或缺的手段，其最重要的作用就是建立起一套普遍的监督体系，每个单位都会指定一名管理者，由其对该单位的所有行为负责，这些管理者又会相互联系，从而建立起组织的权力体系；其次，分组通常要求单位里的人员共享相同的资源，如硬件机

器；最后，分组可以鼓励同一单位内人员的相互调节（即通过非正式的简单沟通实现对工作的协调），因为大家在同一个地点工作，又共用公用设施，如厕所，使得大家彼此接近，促进了经常性的非正式接触。

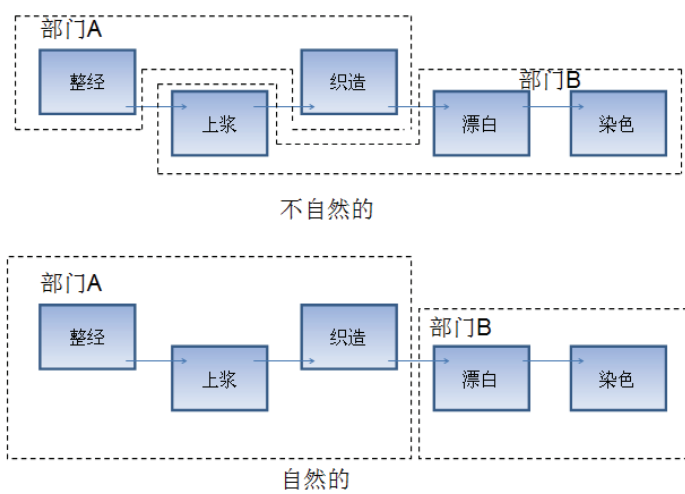
分组带来的最大问题就是：促进了组内协调，却牺牲了组外协调。步兵瑞科就曾愤愤地抱怨：他们就知道在天上飞，我们却在下面送死。（《银河舰队》）作为开发人员的我们也曾经抱怨过：他们就知道提需求，反正也不用自己开发。

那么，组织分组的标准有哪些呢？有以下4个：

- ❑ 工作流相依性；
- ❑ 工作方法相依性；
- ❑ 工作规模相依；
- ❑ 社会相依性。

● 工作流相依性

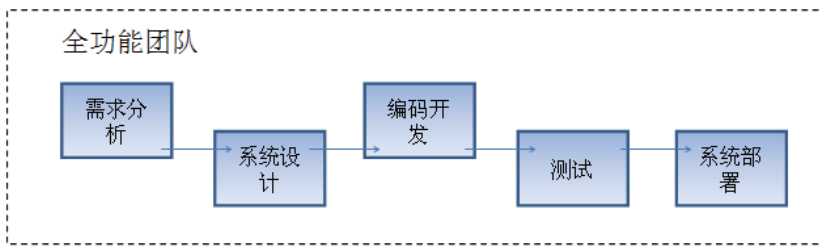
许多针对特定操作活动之间关系的研究，都着重指出了这样一个结论：对操作活动的分组应该反映出工作流的自然相依性。例如，图B-3所示是一位作者对纺织厂中连续生产工序“自然”和“不自然”的看法。以工作流相依性为基础的分组，单位成员会有一种领土完整的感觉，他们支配着一个定义明确的工作流程，工作中所出现的大多数问题，都可以通过彼此的相互调节而得到轻易的解决。相反，如果一个定义明确的工作流程分解到若干不同单位来完成，那么协调起来就困难了。组织要求各单位之间能够相互合作，可实际上，单位之间很难进行良好的合作，所以，一旦出现问题，必须呈交给远离工作流程的上级管理者来解决，而这些上级管理者由于远离实际的工作流程，往往会根据下级汇报做出决策，于是决策的有效性可想而知（参见明茨伯格的《卓有成效的组织》）。



图B-3　根据工作流对纺织厂的“自然”与“不自然”分组

那么，根据工作流相依性分组的最优解和最差解分别是怎样的呢？我们以软件开发流程来进行说明。

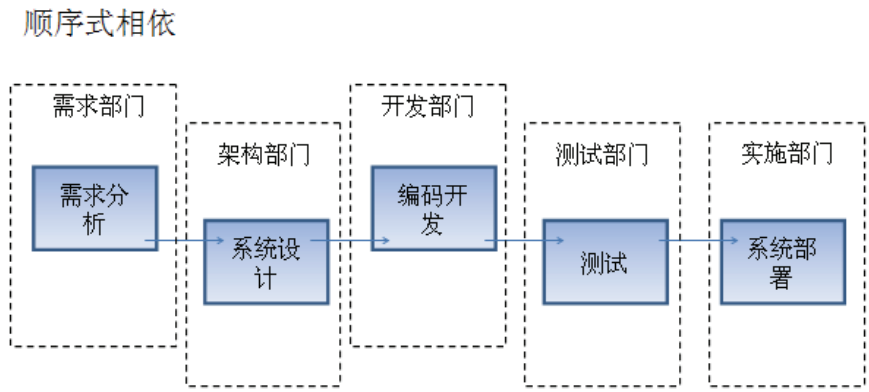
如果一个单位的职能能够涵盖整个完整的工作流程，则是最优解。在这种情况下，工作中的大部分问题都能在单位内得到解决。如图B-4所示，开发流程中的大部分工作都能在一个团队内完成，这个团队包含了BA、开发人员、测试人员等多种角色的成员，所以也被称为全功能团队或闭环团队。



图B-4　工作流相依性分组最优解

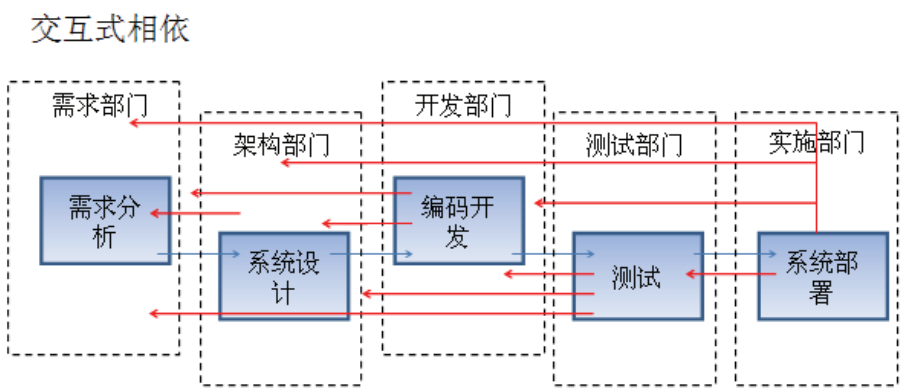
由工作流相依性重新思考组织分组由来已久。1990 年，迈克尔 • 哈默在《哈佛商业评论》上发表了题为“再造：不是自动化改造而是推倒重来”的文章，文中提出的再造思想开创了一场新的管理革命。以此为标志，形成了新的业务流程理念，并伴随着对传统企业金字塔式组织理念和管理模式的反思，新的理念强调企业以业务流程为中心进行运作、打破传统的部门隔阂、增加客户价值和企业效益（降低成本）。以业务流程为中心取代职能分工，成为管理的首要原则，围绕流程建立起来的组织具有更高的敏捷性、效率和效益，呈现出扁平化、网络化的特征。然而，重新思考图B-4所示的全功能团队我们会发现，在很多情况下，在最低层级组建上图所示的全功能团队并不现实（什么是最低层级？意思是该单位不会再有下级单位），出于沟通效率的考虑，一个单位的成员不能够无限扩大，在传统的管理书籍中（法约尔），这个约束甚至被建议为5人。在很多制造型企业里，这个人数实际上是大大超出这个限制的，原因就在于标准化。然而，在知识密集型企业里，因为并没有一致的标准能够遵循，单位成员之间必须面对面沟通以协调彼此的工作，那么单位规模必须足够小，小到便于所有成员能够做到适宜、频繁和非正式的沟通。所以，对于一个软件项目而言，一个小于10人的全功能团队是最适合的，一旦团队规模超过20人，那么就必须进行再分组。对很多软件开发而言，他们需要的人数远超20人，那么这种最低层级上的全功能团队就不再适用。

如果工作流程上的各个单位构成顺序依赖的关系，则是次优解。在这种情况下，每个单位仅仅对其上一个单位产生依赖，单位之间的协调较少。如图B-5所示，可以看出这是一个典型的以职能进行分组的组织，这样的分组至少看上去并不坏，但是现实却是：这是一个相当没效率的分组。原因就在于该分组基于一个重要的假设：开发流程中的活动是可以分阶段完成的即瀑布开发模型。现实中，这个假设却是完全不成立的，这些活动联系的如此之紧密，以致于在这些单位之间不得不时时发生大量的协调。于是该分组实际是图B-6的样子，最差解！



图B-5　工作流相依性分组次优解

如果工作流程上的各个活动需要跨越多个单位进行反复协调沟通，那么则是最差解，称之为交互式相依，如图B-6所示。在我们观察过的一个组织里边，测试人员发现软件缺陷后的第一反应不是走过仅仅一屏风之隔的开发小组里进行沟通，而是先填写在线的缺陷跟踪系统，然后再打开即时消息工具，给开发人员发消息：有缺陷，缺陷号是xxx。组织在进行分组时，必须寻求将协调和沟通的成本降至最低。



图B-6　工作流相依性分组最差解

- 工作方法相依性

即使用相同工作方法的人员分到一个单位，通常也就是职能分组。这种分组的好处在于能够激发方法的互相交流，也就是专业性，同类专家分到一起之后，他们能够互相交流，提高各自的专业水平。在现在公司里，经常能够看到不同团队成员之间的非正式交流，这里，其实是公司整体的文化氛围为这种交流提供了便利。实际分组时，需要在工作流相依性和工作方法相依性间做出权衡。

- 规模相依性

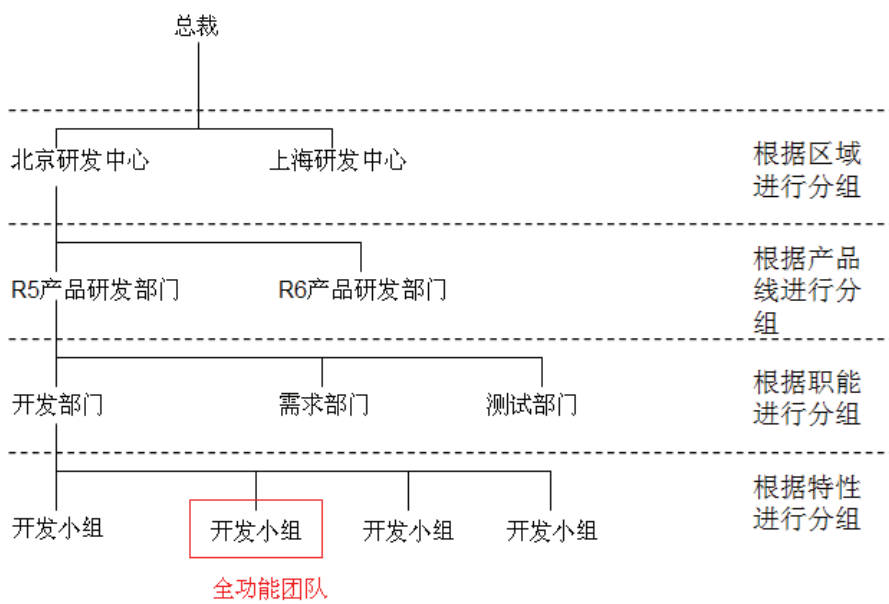
第三个标准与经济规模有关。考虑这样一个例子，软件的测试需要真实的硬件环境进行模拟，而这些硬件比较昂贵，那么一个最经济的方式就是成立专门的测试部，统一购买一批硬件，统一对所有的软件进行上线前测试。同理，由于DBA比较昂贵，公司不可能为每个团队都配备一名，所以DBA不属于任何团队，其是共享的。

- 社会相依性

第四个标准与具体的工作没有关系，与人的社会性有关系。如果领导没有头晕，他是绝对不可能将两个水火不容的人放置到一个单位内的（帝王除外，那叫帝王术）。

以上就是组织进行分组的4种标准。归纳一下：如果工作流相依性意义重大而又难以纳入标准的话，那么组织就应该尝试以市场（项目）为基础进行分组，这样便于相互调节和直接监督；如果工作流不规则，标准化能够涵盖工作流相依性，如果方法和规模相依性意义重大，那么组织应该积极寻求专业化，以职能进行分组。

最后，我们讨论一下大规模软件团队的分组.在上面我们提到，一旦人员规模超过20人，那么最低层级上的全功能团队就不再适用，就有必要进行再分组。如何进行再分组呢？图B-7所示是某IT企业的多层级分组，实际上最重要的是开发部门的按特性分组，每个开发小组都必须能够独立交付产品的一个特性。注意，这里是交付，既然是交付那么就不仅仅包含开发一个活动，还需要包括需求分析与测试，这样，从某种意义上，该开发小组实际构成了全功能团队，实际中，每个开发小组都包括了系统分析人员、开发人员与测试人员。

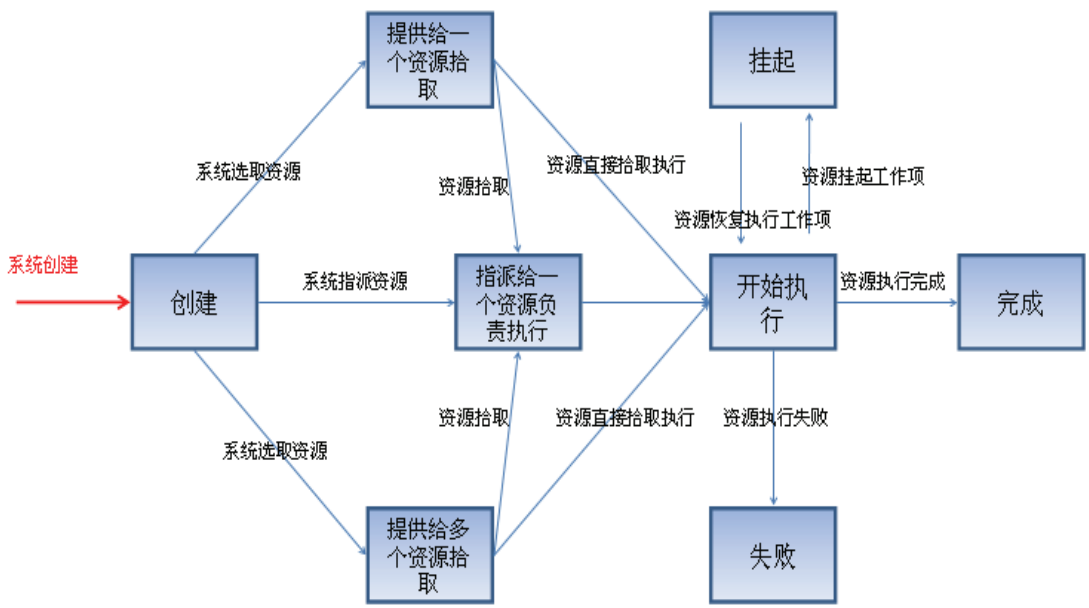


图B-7　某IT企业的多层级分组

开发部门按照特性交付分为多个开发小组后，整个产品由一个个模块构成，新的问题出现，就是系统的集成问题，这里的集成问题实际反映出各个开发小组之间的协调问题。此时，一个独立的测试部门和持续集成就是必须的了，从某种意义上理解，测试部门实际上着重解决的是各个模块间的相互影响以及系统作为一个整体的完整测试，从持续集成的角度考虑，此时最重要的自动化测试应该应用在各个模块之间交互的部分。

创建模式

创建模式位于工作项生命周期的创建阶段，对工作项可能的执行方式进行限定，限定执行该活动的资源范围。我们在活动建模时定义分配规则，说明执行该活动资源所需要符合的前提条件和约束，如图B-8所示。创建模式的重要性在于确保流程实例的执行符合预定的设计原则，通过模型增加活动执行的确定性。

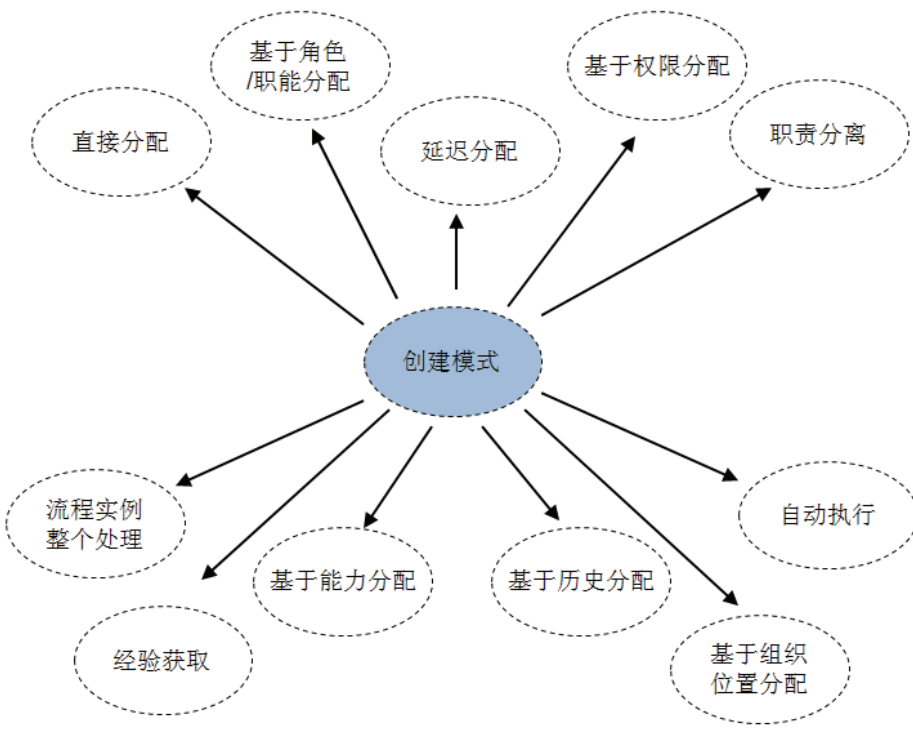


图B-8　创建模式位于工作项的创建阶段

创建模式有11种，如图B-9所示。

- 直接分配：直接为活动指定特定的资源。
- 基于角色/职能的分配：为活动指定资源执行所需要的职能，我们使用角色在组织中定义职能。
- 延迟分配：资源的确定延迟至运行期。
- 基于权限分配：为活动指定资源执行所需要的权限，从而对资源做出限定。

- ❑ 职责分离：为了安全或避免错误，同一流程实例的两个活动必须由两个不同的资源执行。
- ❑ 流程实例整个处理：整个流程实例由一个资源执行，资源对整个流程实例负责，更加投入。
- ❑ 经验获取：活动交与有经验的资源执行，提高活动执行效率。
- ❑ 基于能力的分配：为活动指定资源执行所需要的能力，从而对资源做出限定。
- ❑ 基于历史的分配：基于资源先前工作的历史决定活动的分配，找出最适合的资源。
- ❑ 基于组织位置分配：基于资源在组织中所处的位置以及与其他资源的关系分配活动。
- ❑ 自动执行：活动的执行自动完成，不需要人的参与，例如调用数据库、Web服务。

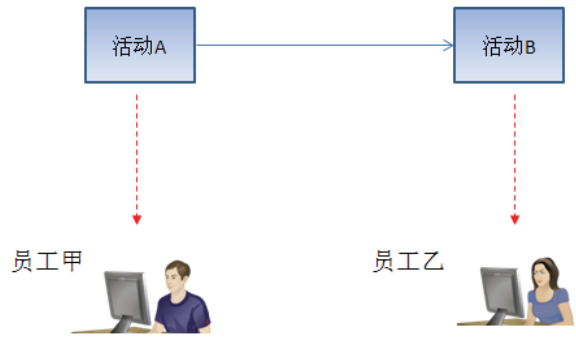


图B-9　创建模式

直接分配（WRP_1: Direct Distribution）

描述

能够在定义期直接为活动指定特定的资源，该活动的工作项在运行期分配给他。示例见图B-10。



图B-10　直接分配

应用

该模式应用于流程里的关键路径。在中小企业里，该模式是应用最多的分配模式，因为人员少，管理扁平，所以每个人的职责都非常清晰。该模式也是执行效率最高的资源模式，因为人和活动直接绑定，所以不会产生推诿等情况，便于管理和追究责任。随着企业规模的扩大，管理层次的增加，一个活动需要交由特定的部门、岗位或角色来执行，这样无形中影响活动的执行效率。

该模式的缺点在于，一旦关键资源因为各种原因不能及时处理活动，那么将造成整个流程实例的挂起等待。

基于角色的分配（WRP_2: Role-Based Distribution）

描述

能够在定义期为活动指定一个或多个角色，该活动的工作项在运行期分配给属于这些角色的资源。示例见图B-11。

图B-11　基于角色的分配

应用

企业达到一定规模（能同时处理多个流程实例），必然需要对人员进行分组，角色是典型的职能分组方式，将具有相同职能特征的人员定义为一个特定的角色，这些属性与工作的内容和资源的技能相关，例如开发人员、项目经理、总经理等，而角色又会与权限产生关联。

将活动分配给角色意味着将会有多个资源可以执行该活动，需要协调，执行效率相比直接分配会下降，这也是企业扩大后管理成本增大的一种表现。

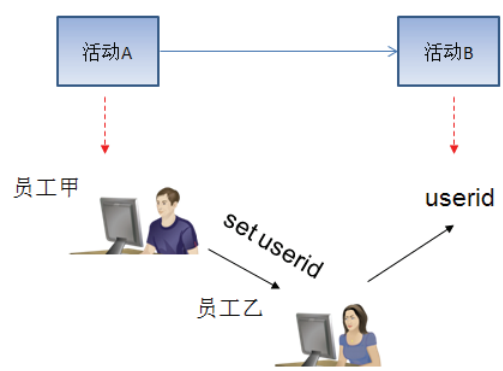
如果我们必须在两个资源间进行选择，而他们在执行该工作项上是等价的，那么最好选择那个相对只能处理少量其他种类工作的资源。换句话说，当还有其他的资源可以选择时，尽量让通用性好的资源空闲，为将来尽可能预留弹性资源。

延迟分配（WRP_3: Deferred Distribution）

描述

能够在定义期为活动指定工作流数据变量，定义期并不确定资源，资源的确定被延迟至运行期，系统运行期从该数据变量里读取该活动工作项所要分配的资源。

如图B-12所示，活动B在定义分配规则时指向数据变量userid，当在一个流程实例中实际执行时，活动A由员工甲执行，其指定下一活动的执行者为员工乙即设置了userid这一数据变量为员工乙的ID，这样当活动B工作项创建时，系统访问userid，发现该数据变量指向员工乙的ID，于是将该工作项分配给员工乙。根据具体的分配策略，运行期该数据变量不仅可以指定人员，也可以指定角色、部门。



图B-12　延迟分配

应用

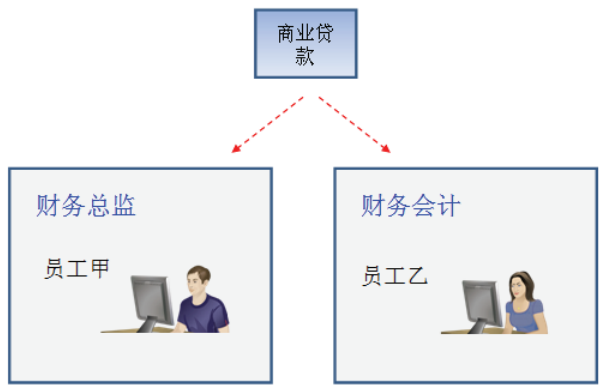
该模式给资源的分配引入灵活性。资源的确定延迟至运行期，即活动可以根据具体的流程实例执行情况确定执行资源。例如定义后续活动的执行者为当前活动执行者的部门经理、北京地区提交的货物订单交由北京配送中心的业务员进行处理、下一活动的执行由上一活动的办理者指定

等等。具体实施时，这个指定的数据变量可以通过一系列的规则运算得出。

基于权限分配（WRP_4: Authorization）

描述

能够在定义期为活动指定资源执行所需要的权限，只有具有权限的资源才能执行该活动。示例见图B-13。



图B-13　基于权限分配

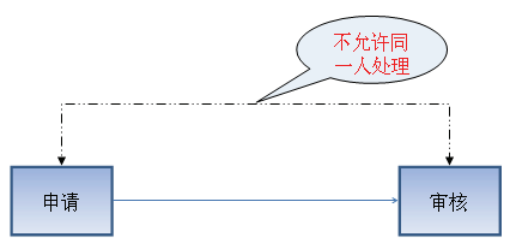
应用

该模式在流程模型里引入组织权限。权限代表着对同一件事情不同资源执行工作内容的差别。权限越大能执行的操作越多意味着需要负的责任越大。

职责分离（WRP_5: Separation of Duties）

描述

在一个流程实例里，能够指定两个活动必须由不同的资源执行。示例见图B-14。



图B-14　职责分离

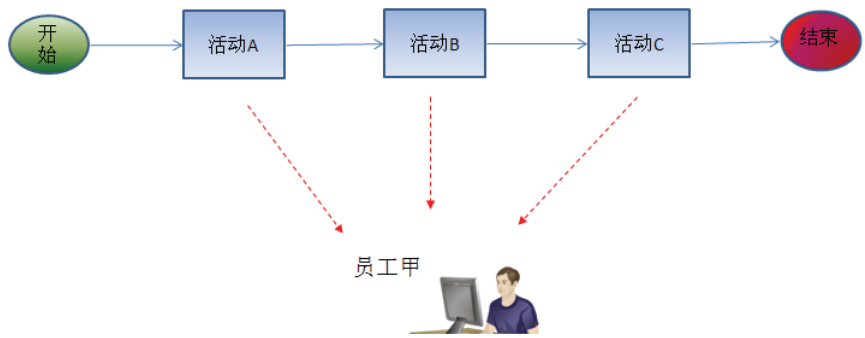
应用

职责分离，不能既当运动员又当裁判员，不能又当跳水队领队又当全运会裁判长。避免滥用权力。

流程实例整个处理（WRP_6: Case Handing）

描述

在一个流程实例里，所有的活动都能够分配给同一个资源执行。示例见图B-15。



图B-15　流程实例整个处理

应用

在应用该模式的流程里，流程里的活动都是紧密关联的。流程里的活动执行在一个紧密相关的上下文里，如果所有的工作都由一个人执行，那么在其了解该上下文的情况下连贯执行这些活动会具有更高的效率；而如果执行活动的过程中换人，那么新换的人无疑还需要时间对该流程实例相关的上下文进行熟悉，这样无疑会多付出执行成本。此外，一个资源对整个流程实例全权负责会更加投入，能够给顾客提供更优质快捷的服务。

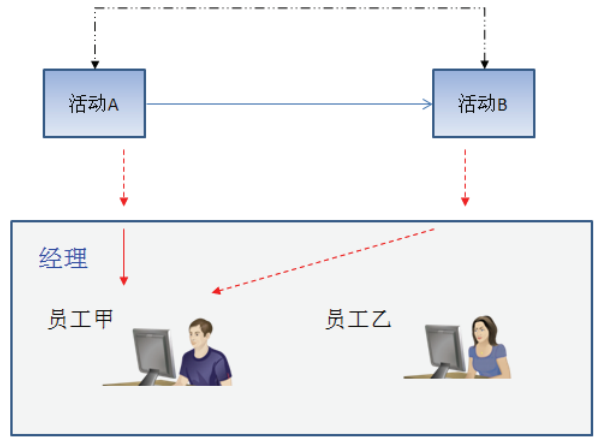
即使一个资源不能执行流程实例里的所有活动，也需要设立流程实例管理员来对整个流程实例负责。

经验获取（WRP_7: Retain Familiar）

描述

在一个流程实例里，当存在多个资源都能执行某一活动时，能够将工作项优先分配给执行了同一流程实例前某一活动的资源。

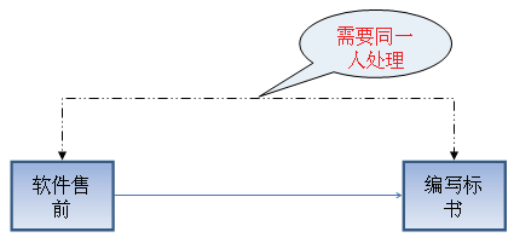
如图B-16所示，活动A和活动B在定义期被指定由同一个角色执行，那么在运行期，在一个流程实例里，如果活动A被分配给了员工甲执行，那么在进行活动B的分配时，活动B依旧由员工甲执行。



图B-16　经验获取

应用

该模式加快活动的执行，这些活动之间存在着紧密关联，如果执行了其中一个，那么就会熟悉这些相关联活动的背景上下文，积累一定经验，那么后续活动的执行相对其他人来说会变得容易。提高活动执行的效率。

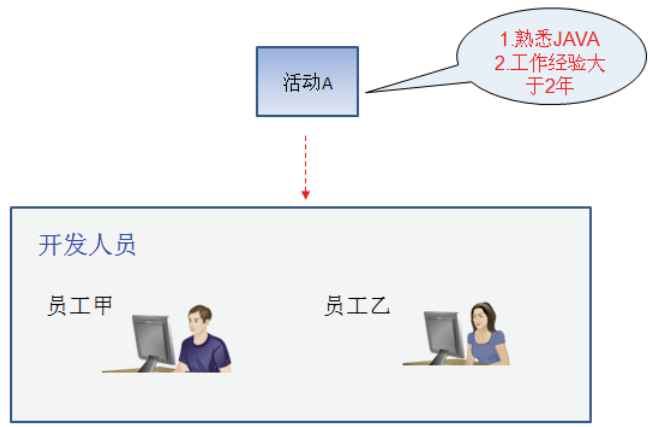


图B-17　尽量使用同一资源执行关联活动

基于能力的分配（WRP_8: Capability-Based Distribution）

描述

能够在定义期为活动指定执行所需要的能力，基于资源的能力进行工作项的分配，能力作为组织模型的一部分建模为资源的属性。示例见图B-18。



图B-18　基于能力分配

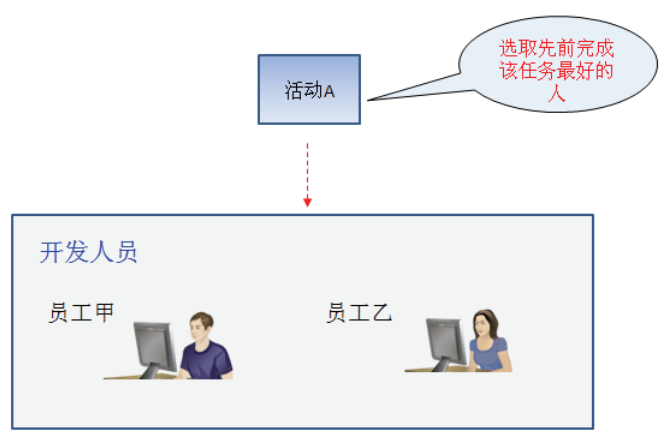
应用

人力资源管理中很重要的一点就是要做到人尽其用，每个人的能力都能得到最大的发挥，其实也就是根据能力将人放置到最合适的工作中去。从这一点看，该模式是对人尽其能的实现，让资源发挥自己的专长。但是如何定义各个人员的能力，这本身就比较的主观，执行各项工作需要具备的能力也具有主观因素，更何况很多时候能力并不能与表现划上等号，模型是固定的但人是变化的受环境影响的，工具本身就是工具，工具的应用最后还是依赖于人。

基于历史的分配（WRP_9: History-Based Distribution）

描述

能够基于资源先前的工作历史决定活动的分配。示例见图B-19。



图B-19　基于历史分配

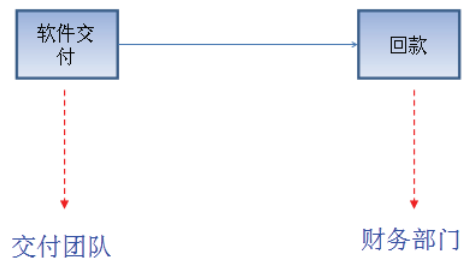
应用

找出最适合的资源。考虑历史时，有很多因素要考虑，例如完成类似活动最好的员工、完成类似活动最快的员工、完成类似活动出差错最少的员工等。这些考虑因素依赖于具体的活动/流程实例属性和背景。

基于组织位置分配（WRP_10: Organizational Distribution）

描述

能够基于资源在组织机构中的位置以及与其他资源的关系分配活动。示例见图B-20。



图B-20　基于组织位置分配

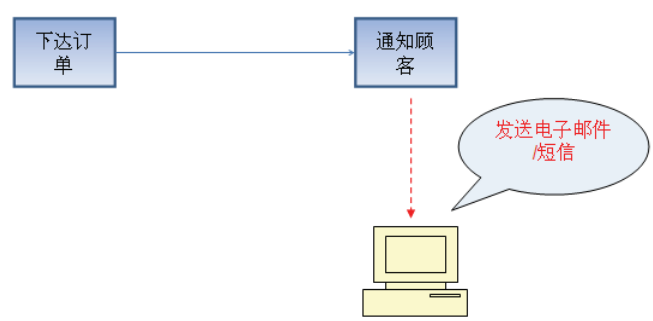
应用

确保活动在组织的正确位置中得到执行，基于组织结构而不是职能特征分配活动。组织位置最典型的建模就是部门。

自动执行（WRP_11: Automatic Execution）

描述

活动的执行能够自动完成，不需要人的参与。示例见图B-21。



图B-21　自动执行

应用

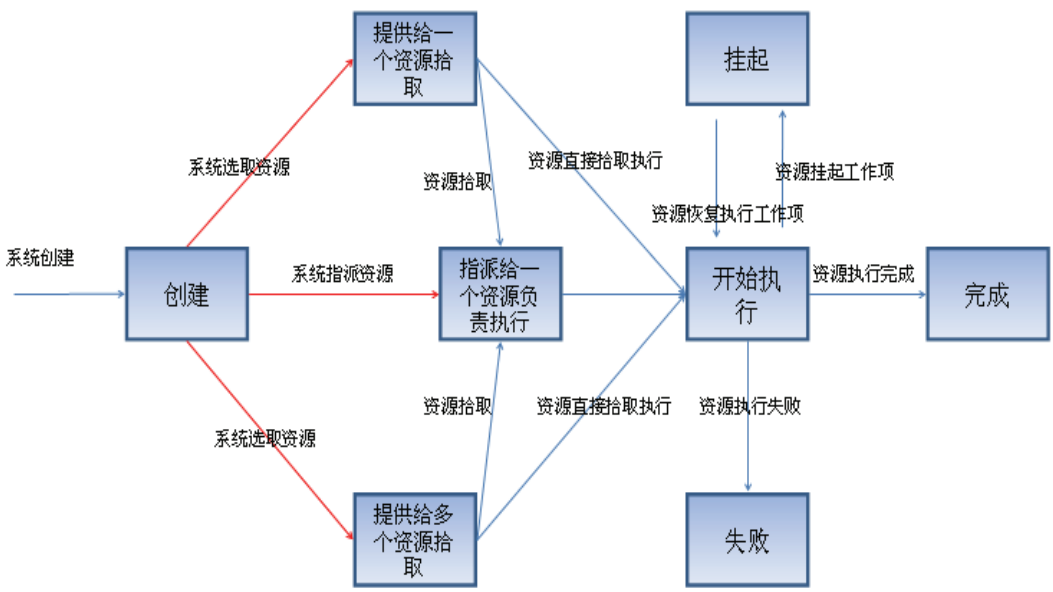
自动化使得越来越多的工作可以交由计算机或相应的机器设备直接完成。流程中的自动执行节点逐渐增加。工作流系统对该模式的支持即是支持各种的企业集成方式，不管是通过Web服务还是消息，工作流需要驱动相应的设备机器或应用系统进行工作并传递给必须的数据。随着信息化程度的提高，目前流程应用越来越趋向于企业集成领域。这也是为什么基于Web服务编排的BPEL会逐渐取代XPDL成为流程执行标准的原因之一。

推模式

在创建阶段，工作流系统根据不同的创建模式为活动产生了工作项，并为工作项限定了资源范围。接下来，系统将工作项与已限定的资源进行匹配，推送给相关的资源执行，资源本身不做选择，我们把这种方式称为“推式驱动”，系统将工作项推送给资源。

在前面我们已经了解到，工作流系统通过工作项管理器即不同类型的工作项列表与用户进行交互，所以这里的推送也可以理解为系统将生成的工作项推送至相应资源的工作项列表里。

如图B-22所示，推模式对应着工作项生命周期里三种状态的变迁，即提供给一个资源拾取、提供给多个资源拾取（这些资源中只会有一个会实际执行，属于竞争关系）、指派给一个资源负责执行。

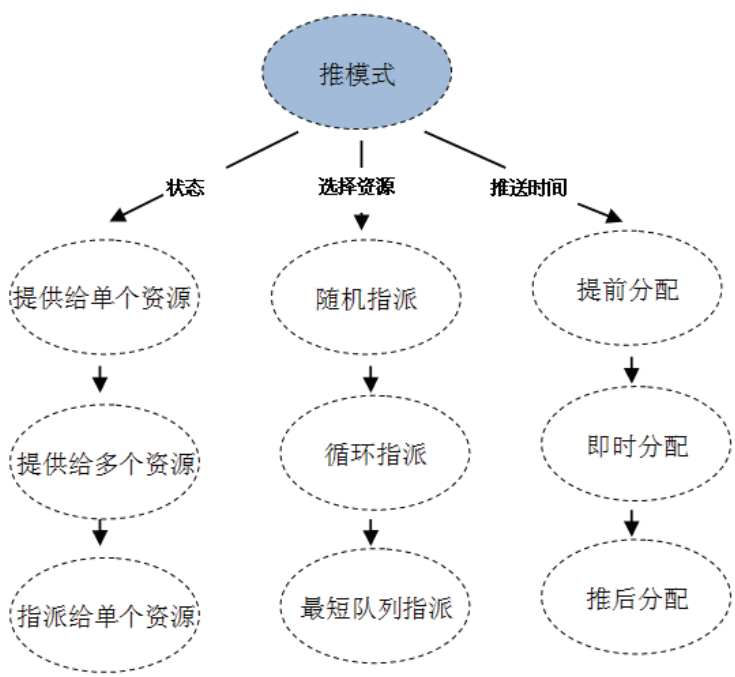


图B-22　工作项生命周期里的推模式

推模式共有9种，分为3组，如图B-23所示。第一组包括提供给单个资源、提供给多个资源和指派给单个资源，关注工作项推送的最终分配状态；第二组包括随机指派、循环指派和最短队列

指派，关注当工作项分配给角色、部门等包含多个资源的资源组时，如何从中确定最终的一个资源并进行指派；第三组包括提前分配、即时分配和推后分配，关注将工作项推送给用户的时机。

- 提供给单个资源：在非绑定的基础上将工作项推送给单个资源。
- 提供给多个资源：在非绑定的基础上将工作项推送给多个资源。
- 指派给单个资源：在绑定的基础上将工作项推送给单个资源。
- 随机指派：当存在多个资源可供选择时，从中随机选择一个资源进行工作项的指派。
- 循环指派：当存在多个资源可供选择时，循环选择其中一个资源进行工作项的指派。
- 最短队列指派：当存在多个资源可供选择时，选择其中一个具有最少待办工作即最短工作队列的资源进行工作项的指派。
- 提前分配：在工作项实际可以执行之前即将该工作项通知或潜在的分配给资源，提前预热。
- 即时分配：在工作项实际可以执行时将该工作项分配给资源。
- 推后分配：在工作项实际可以执行后的某个时间才将该工作项分配给资源，减少资源负载，提高流程实例吞吐量。



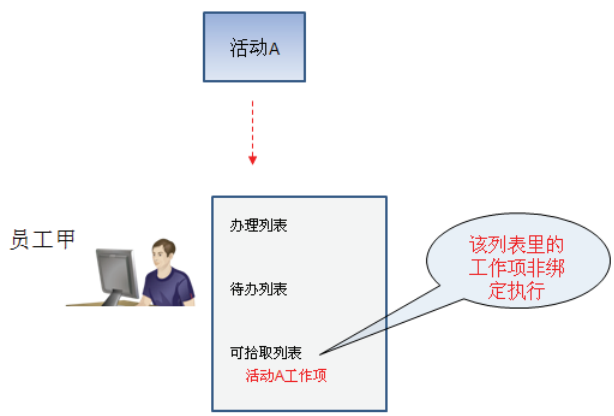
图B-23　推模式

提供给单个资源（WRP_12: Distribution by Offer - Single Resource）

描述

能够在非绑定的基础上将工作项推送给单个资源。

如图B-24所示，活动A工作项被系统推送至员工甲的可拾取列表。这意味着员工甲不必为该工作负责，他可以选择执行该工作也可选择忽略或拒绝。如果他选择拒绝或忽略且工作项超时，那么会导致系统对该工作项的重新分配。如果他选择执行该工作，那么他首先需要拾取该工作项，这会使该工作项进入他的待办列表，意味着其必须对该工作负责。



图B-24　提供给单个资源

应用

资源能够将工作与自己进行匹配，选择执行工作。

提供给多个资源（WRP_13: Distribution by Offer – Multiple Resource）

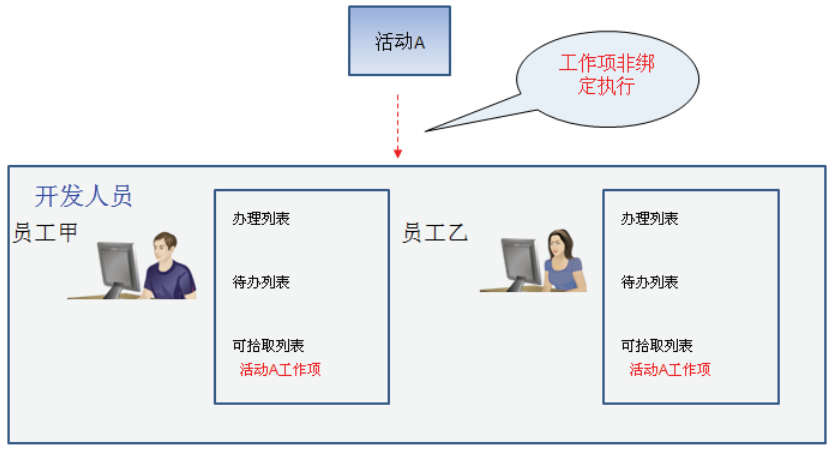
描述

能够在非绑定的基础上将工作项推送给多个资源。

如图B-25所示，活动A所生成的工作项被推送给多个员工的可拾取列表。这些员工不必为该工作负责，他们可以选择执行该工作也可选择忽略或拒绝。如果他们都选择拒绝或忽略且工作项超时，那么会导致系统对该工作项的重新分配。如果有一名员工选择执行该工作，那么该工作项进入他的待办列表，其他员工将不再具有拾取该工作项的机会。

应用

资源能够将工作与自己进行匹配，选择执行工作。多个资源选择，竞争执行。



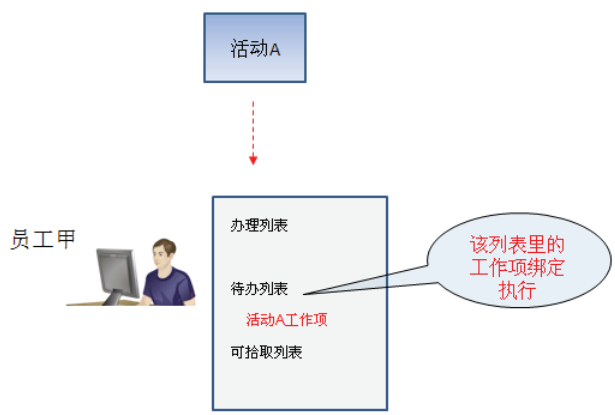
图B-25　提供给多个资源

指派给单个资源（WRP_14: Distribution by Allocation – Single Resource）

描述

能够在绑定的基础上将工作项推送给单个资源。

如图B-26所示，活动A工作项被系统推送至员工甲的待办列表。这意味着员工甲必须为该工作负责。



图B-26　指派给单个资源

应用

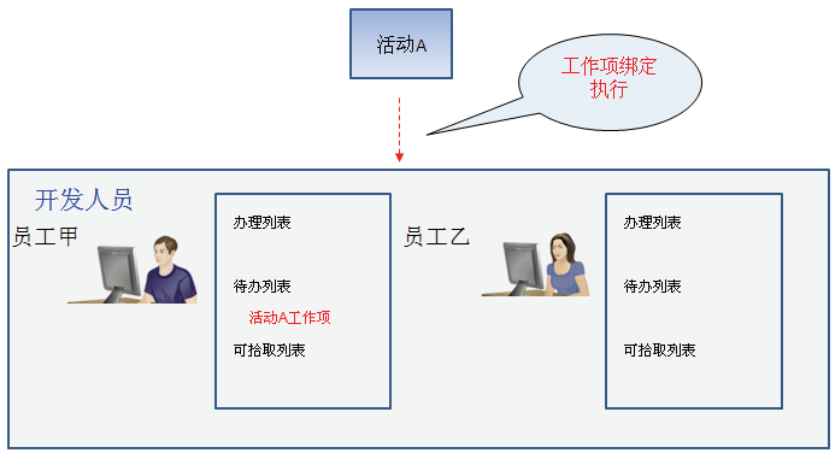
直接指定工作责任人。

随机指派（WRP_15: Random Allocation）

描述

当存在多个资源可供选择时，能够从中随机选择一个资源进行工作项的指派。

如图B-27所示，活动A所生成的工作项在创建阶段分配给了开发人员这一角色，在推送阶段，系统会随机选取一名开发人员负责该工作项的执行。



图B-27　随机指派

应用

提供了一种指派资源的非确定性机制。

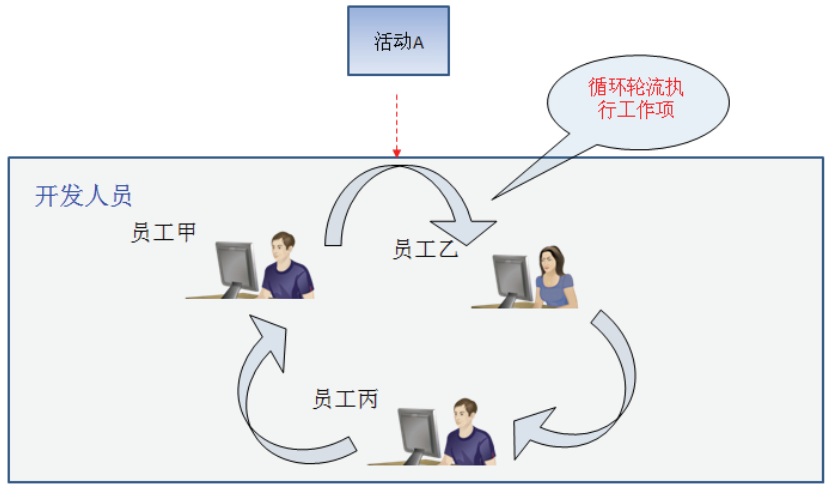
循环指派（WRP_16: Round Robin Allocation）

描述

当存在多个资源可供选择时，能够循环选择其中一个资源进行工作项的指派。示例见图B-28。

应用

不患贫而患不均，平等/平均地分配工作。

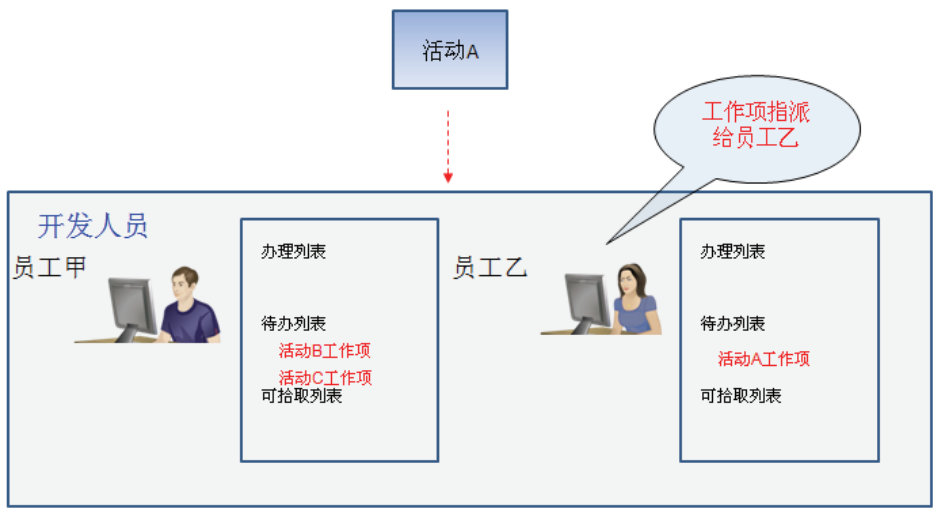


图B-28　循环指派

最短队列指派（WRP_17: Shortest Queue）

描述

当存在多个资源可供选择时，能够选择其中一个具有最少待办工作即最短工作队列的资源进行工作项的指派。示例见图B-29。



图B-29　最短队列指派

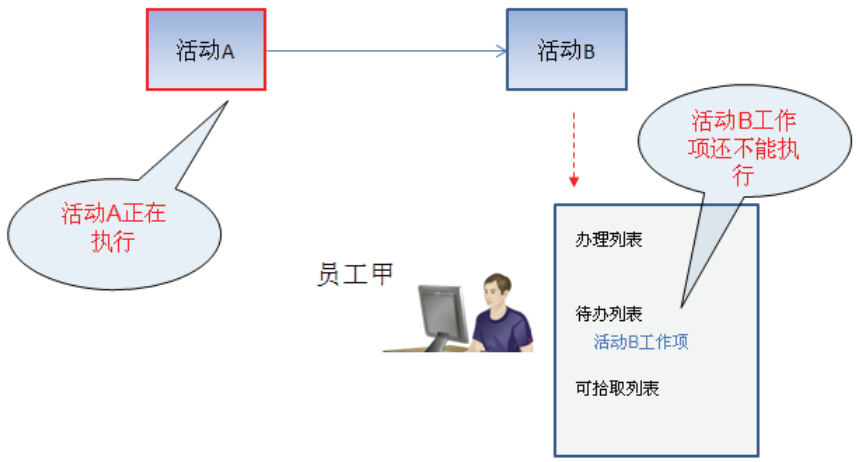
应用

该模式的目的在于能够最快开始工作的执行，找出相比而言最为空闲的资源迅速开始工作。但是实际应用中，仅仅依靠工作的数量来判断资源是否空闲是不可靠的，因为不同工作甚至同一种工作在不同时间都存在着难易之分。

提前分配（WRP_18: Early Distribution）

描述

能够在工作项实际可以执行之前即将该工作项通知或潜在的分配给资源。示例见图B-30。



图B-30提前分配

应用

该模式强调的是预先计划，即管理的计划性。从某种意义上说，稍微复杂一点的工作都应该做到提前通知、提前准备，做到提前预热。

即时分配（WRP_19: Distribution on Enablement）

描述

能够在工作项实际可以执行时将该工作项分配给资源。

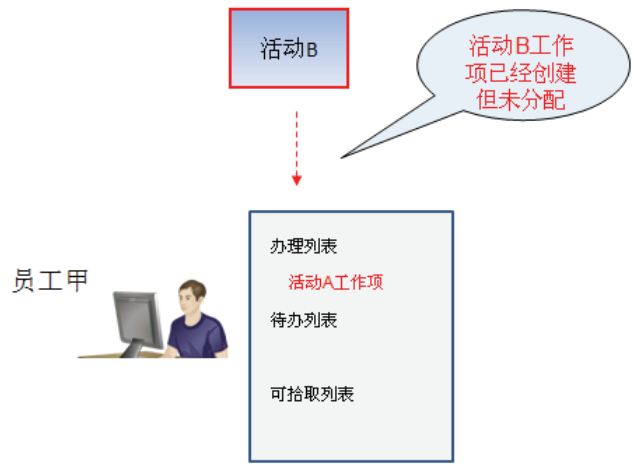
应用

机械性的工作（生产流水线），重复单一高度标准化的工作，无计划性的工作，如各种突发情况的处理。

推后分配（WRP_20: Late Distribution）

描述

能够在工作项实际可以执行后的某个时间才将该工作项分配给资源。示例见图B-31。



图B-31　推后分配

应用

在实际触发工作项的执行之前，考虑其他一些因素：当前组织是否有足够的资源，当前组织的负载等，保证组织和资源对工作的负载处于一种良好的状态，避免出现图B-32所示的情况。同时，加快对正处理流程实例的执行速度。

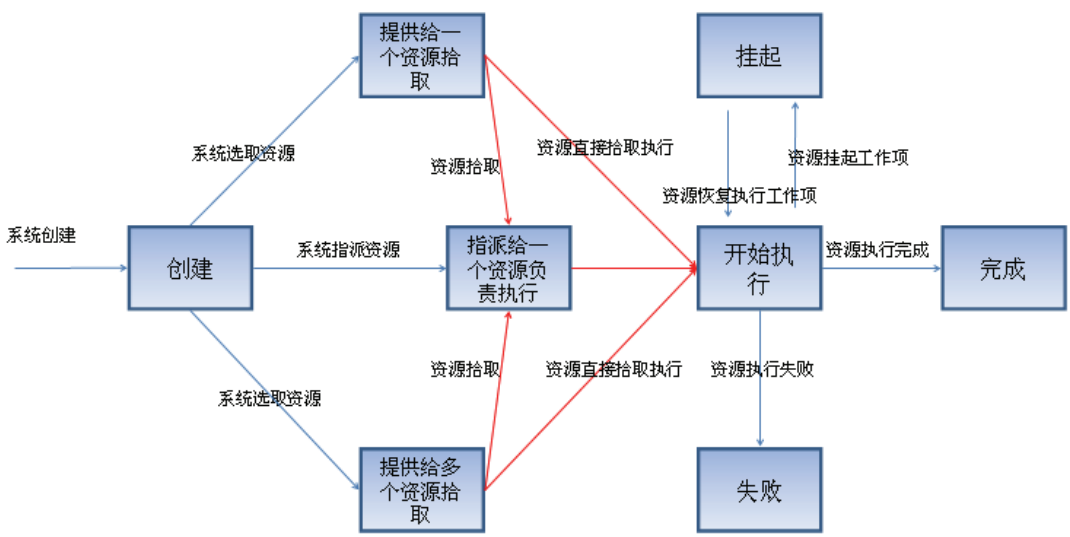
图B-32　工作负荷超出组织能力

拉模式

与推模式相比，拉模式动作的主语发生了变化：推模式的主语是工作流系统，由系统将工作项推送给资源；拉模式的主语是资源，资源把工作项与自己进行匹配，考虑自己能够执行的工作项，从中选择一个，资源拉动工作项。

拉模式对应着工作项生命周期里的5种状态变迁，如图B-33所示。

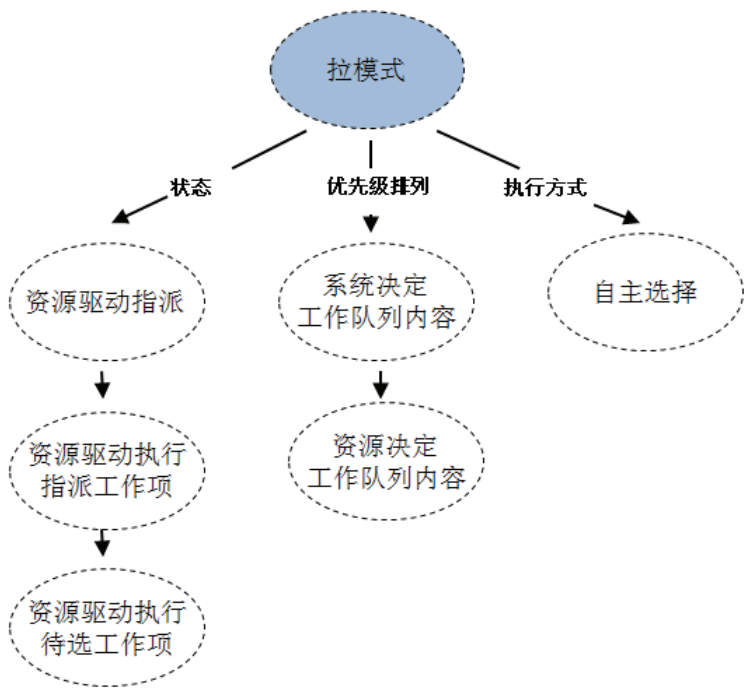
- 由提供给一个资源拾取到指派给一个资源负责执行：这意味着该资源拾取了该工作项，其将负责该工作项的执行，并将在未来的某个时候执行该工作项；
- 由提供给多个资源拾取到指派给一个资源负责执行：这意味着多个资源中的一个资源拾取了该工作项，其将负责该工作项的执行，并将在未来的某个时候执行该工作项，余下的资源将不再有机会执行该工作项；
- 由提供给一个资源拾取到开始执行：这意味着该资源拾取了该工作项，其将负责该工作项的执行，并立即开始执行该工作项；
- 由指派给一个资源负责执行到开始执行：这意味着该资源开始执行该工作项；
- 由提供给多个资源拾取到开始执行：这意味着多个资源中的一个资源拾取了该工作项，其将负责该工作项的执行，并立即开始执行该工作项，余下的资源将不再有机会执行该工作项；



图B-33　工作项生命周期里的拉模式

拉模式共有6种，分为3组，即资源驱动指派、资源驱动执行指派工作项和资源驱动执行提供工作项关注资源驱动工作项的状态变迁；系统决定工作队列内容和资源决定工作队列内容关注工作项优先级的排列；自主选择关注资源选择执行工作项的方式，如图B-34所示。

- 资源驱动指派：资源将工作项指派给自己，负责该工作项的执行。
- 资源驱动执行指派工作项：资源开始执行指派给其的工作项。
- 资源驱动执行提供工作项：资源选取提供给其的工作项，并马上开始执行该工作项。
- 系统决定工作队列内容：工作流系统排定资源工作项列表里工作项的执行顺序。
- 资源决定工作队列内容：资源排定工作项列表里工作项的执行顺序。
- 自主选择：资源根据自己个人的情况选择执行工作项。



图B-34　拉模式

资源驱动指派（WRP_21: Resource-Initiated Allocation）

描述

资源能够将工作项指派给自己，负责该工作项的执行，但是不必马上开始执行该工作项。示例见图B-35。

该模式对应着工作项的两种状态变迁：由提供给一个资源拾取到指派给一个资源负责执行、由提供给多个资源拾取到指派给一个资源负责执行。

图B-35　资源驱动指派

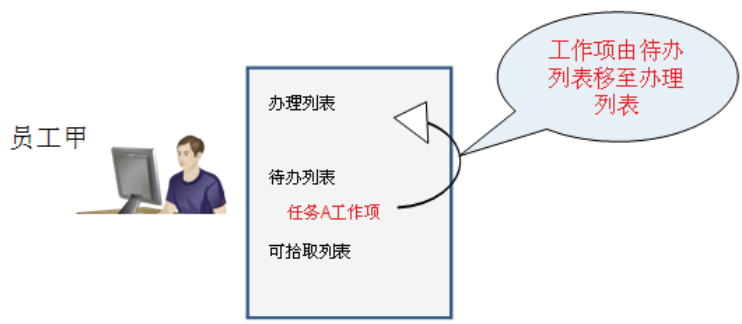
应用

资源能够将工作项与自己匹配并选择执行。

资源驱动执行指派工作项（WRP_22: Resource-Initiated Execution – Allocated Work Item）

描述

资源能够开始执行指派给其的工作项。示例见图B-36。



图B-36　资源驱动执行指派的工作

该模式对应着工作项的一种状态变迁：由指派给一个资源负责执行到开始执行。

应用

资源标识工作已经开始执行。

资源驱动执行提供工作项（WRP_23: Resource-Initiated Execution – Offered Work Item）

描述

资源能够选取提供给其的一个工作项，并马上开始执行该工作项。示例见图B-37。

图B-37　资源驱动执行可选工作

该模式对应着工作项的两种状态变迁：由提供给一个资源拾取到开始执行、由提供给多个资源拾取到开始执行。

应用

该模式强制要求资源一旦拾取了可选的工作项就必须马上开始执行，基于两点的考虑：一是工作项需要尽快执行；二是工作项能够指派给当前最为空闲的资源，不出现该工作项被繁忙资源卡住，造成等待和阻塞。

在日常开发里，我们使用看板/故事卡管理项目的开发。每天早上由开发人员在看板上挑选移动故事卡，一旦故事卡由可开发状态移动至开发状态，则必须进行该卡的开发工作，必须展示项目的真实进度，同时不允许一个开发人员同时进行多张故事卡的开发。

在工作流系统里，实现上述3个模式只是在不同的工作项列表里移动这些工作项，以反映工作项不同的状态和变迁策略，对IT系统而言这很简单，困难在于如何能保证人确实是这么做的，例如说一旦拾取就必须开始执行，工作项的跳转很简单，但无法保证的是拾取该工作项的人一定会按照要求马上开始执行该工作项，也就是说流程项目的实施不仅仅包含技术实施，也包含了一套与之相应的管理实施。那种期望上一套流程系统就能马上提高生产效率和管理水平是不现实的，其中一定需要包含管理方式和组织机构的相应变化。

系统决定工作队列内容（WRP_24: System-Determined Work Queue Content）

描述

工作流系统能够排定资源工作项列表里工作项的执行顺序。示例见图B-38。

图B-38　系统排定工作顺序

应用

工作项的排序规则非常多，其目的是将最重要或优先级最高的工作项排在最前面，引起资源的注意优先执行。

- 先进先出：按照工作项创建的顺序进行排序，也可以按照整个流程实例被创建的时间进行排序。先进先出是一个简单但是很有效的分配规则，在实践中被广泛采用。
- 后进先出：与先进先出相反，最近创建的工作项要优先处理，在某些情况下，能够提高平均服务水平。
- 最短处理时间：根据流程实例的一些属性，区分容易的和困难的流程实例、简单的和耗时的活动。优先选择那些耗时最少的工作项，能够降低流程实例平均处理时间。
- 最短剩余处理时间：对剩余处理时间最短的流程实例优先处理，能够减少正在执行的工作数量并提高服务水平。
- 最早截止期限：考虑流程实例执行的上下文，根据流程实例的截止时间决定工作顺序，今天要完成的工作比本周要完成的工作具有更高的优先级。

每一种排序规则所需要的信息量存在很大的不同，先进先出不需要信息，最短剩余处理时间需要流程实例路由信息和流程实例属性。事实上，还存在更高级的排序规则，这些规则充分考虑正在进行的工作、预期到来的工作、组织资源的可用性等。

资源决定工作队列内容（WRP_25: Resource-Determined Work Queue Content）

描述

资源能够排定其工作项列表里工作项的执行顺序。

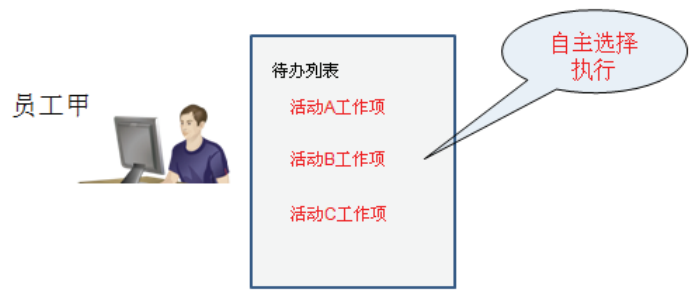
应用

为资源提供一定程度上排序工作项的灵活性。每个人关注的视角和侧重点不同，就会产生不同的工作排序和内容过滤。工作流系统被赋予顾问的角色，资源保留自由度。

自主选择（WRP_26: Selection Autonomy）

描述

资源能够根据自己个人的情况选择执行工作项。示例见图B-39。



图B-39　自主选择执行工作

应用

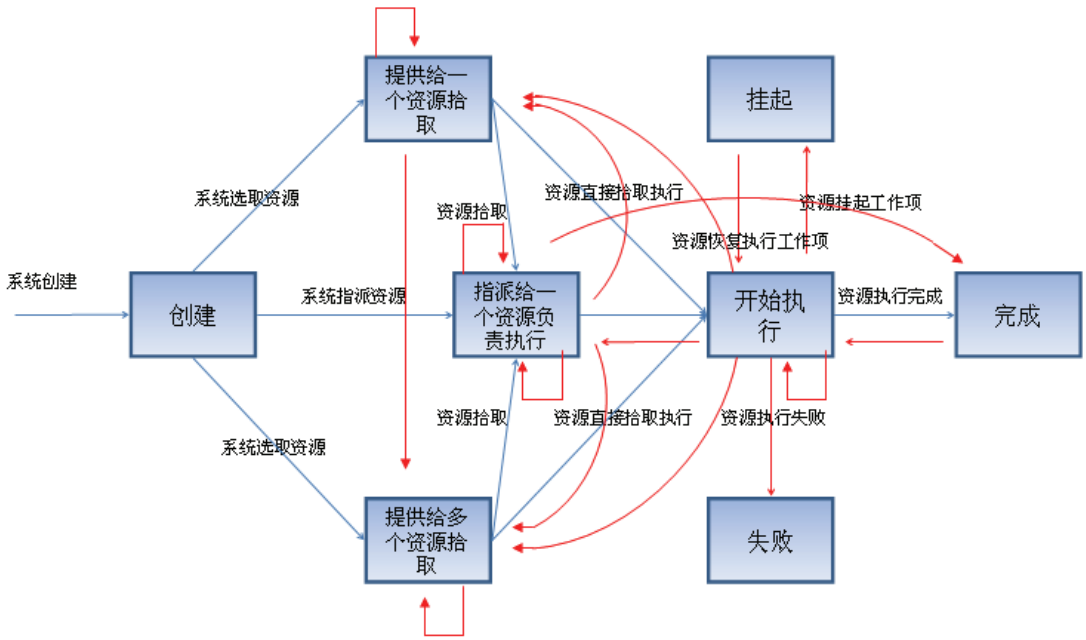
资源能够自主安排自己的工作优先次序和个人的工作序列。

折回模式

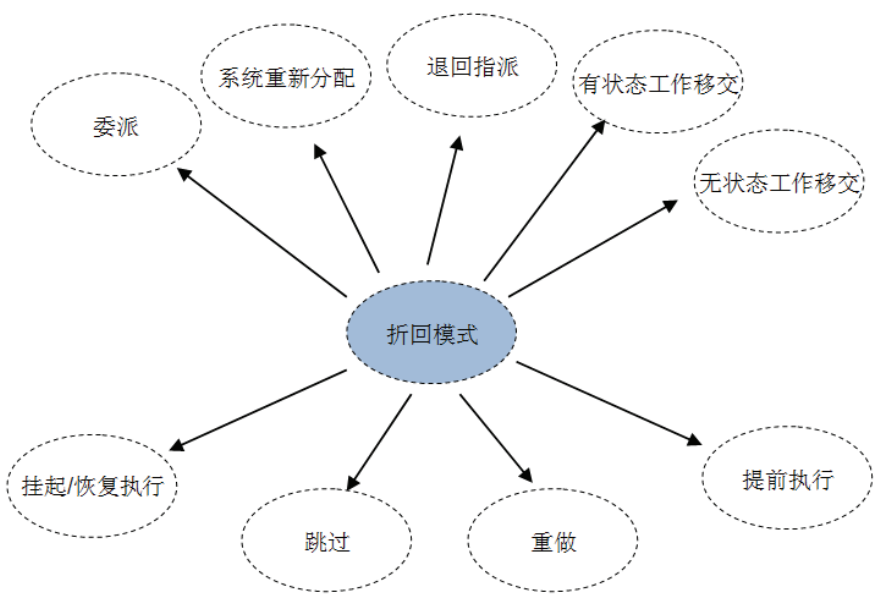
实际工作中，工作的执行状态不可能总是与预想相符，总会出现各种各样的情况，例如原本分配给员工甲的活动由于甲要请假不得不重新分配，由于需要处理新的紧急活动，员工乙当前的工作需要挂起一段时间等等。折回模式对应着这些情况，折回代表着工作项状态的反复和回退，如图B-40所示。

折回模式共有9种，如图B-41所示。

- 委派：资源将先前指派给他的工作项委派给他人执行。
- 系统重新分配：系统将没有完成的工作项重新提供或指派给其他资源执行。
- 退回指派：资源撤销指派给他的工作项，工作项重新指派给其他资源。
- 有状态工作移交：资源将其已经开始执行的工作项移交给他人执行，工作项保持状态。
- 无状态工作移交：资源将其已经开始执行的工作项移交给他人重新开始执行。
- 挂起/恢复执行：资源临时挂起当前执行的工作项，并在某一个时候重新恢复执行该工作项。
- 跳过：资源选择跳过指派给他的工作项，不执行该工作项同时将工作项置为完成。
- 重做：资源重新执行先前已经完成的工作项。
- 提前执行：资源在流程实例实际触发该工作前提前执行该工作。



图B-40　工作项生命周期里的折回模式

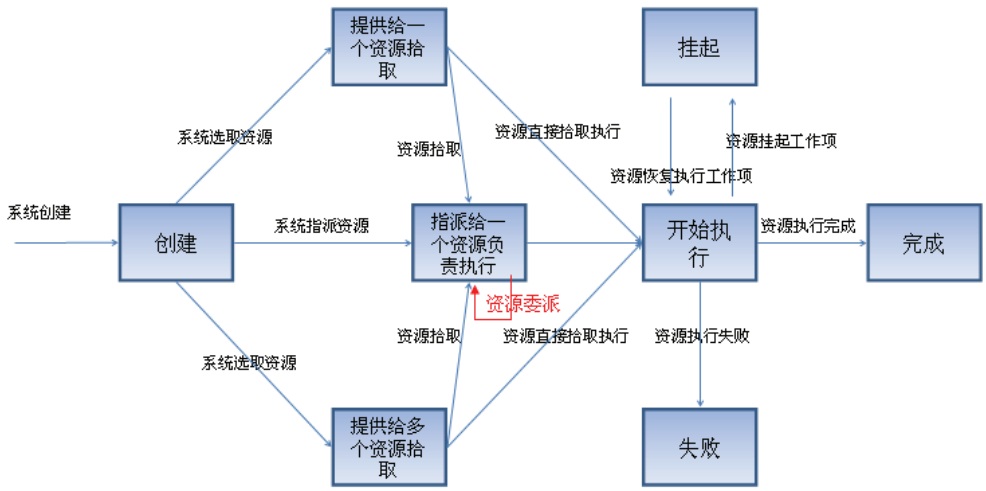


图B-41　折回模式

委派（WRP_27: Delegation）

描述

资源能够将先前指派给他的工作项指派给另外的资源执行，如图B-42所示。



图B-42　委派

应用

委派在工作中非常常见，例如员工请假/出差/繁忙，需要将他的工作委派给其他同事执行、领导将相关工作委派给下属执行等。

实际应用中，委派按照粒度分为了两种：一种是工作项的委派，这是一种细粒度的委派，指单一活动的委派，与某一特定的流程实例关联；另一种是业务的委派，这是一种粗粒度的委派，例如，资源将其负责的某类业务的工作全部委派给他人，这意味着属于这类业务的所有工作都将由委派人执行。业务的委派与权限紧密关联。

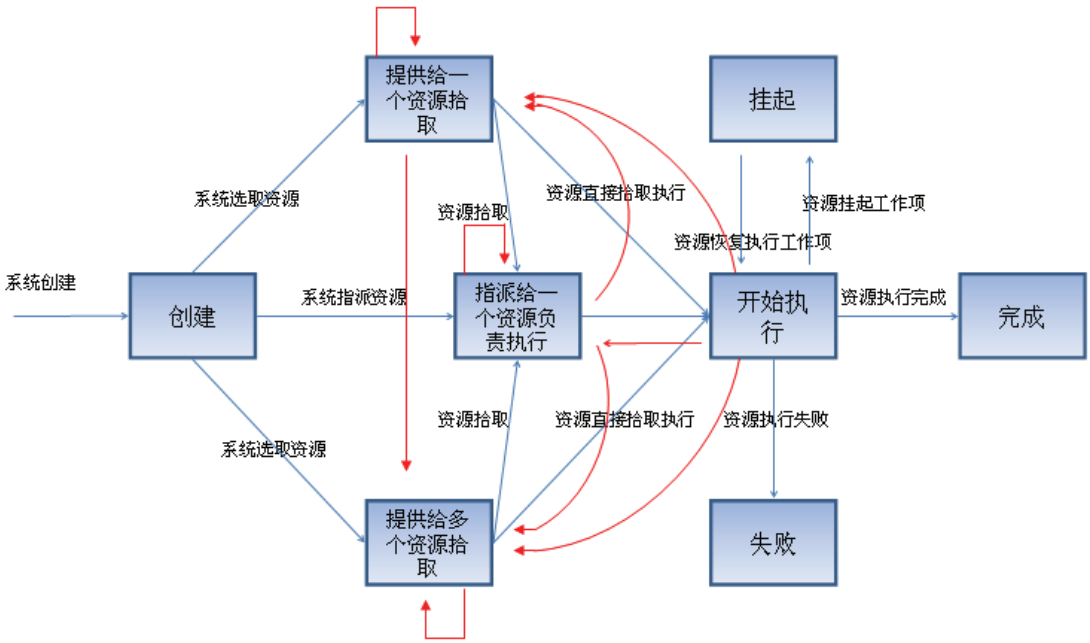
注意：委派意味着原先指派的资源还必须对该工作负责。例如，员工甲将某项工作委派给员工乙执行，尽管员工乙实际执行了该工作，但该工作仍然必须由员工甲负责，所以在实现中，员工甲必须能够保持对委派工作项的追踪和控制。

系统重新分配（WRP_28: Escalation）

描述

系统能够重新分配已经分配的工作项，以加快工作项的执行。

如图B-43所示，工作项原先提供或指派给了一个或多个资源执行，现在由于各种原因，需要优化该工作项的执行，所以将该工作项收回重新分配，提供或指派给其他的资源。



图B-43　系统重新分配

应用

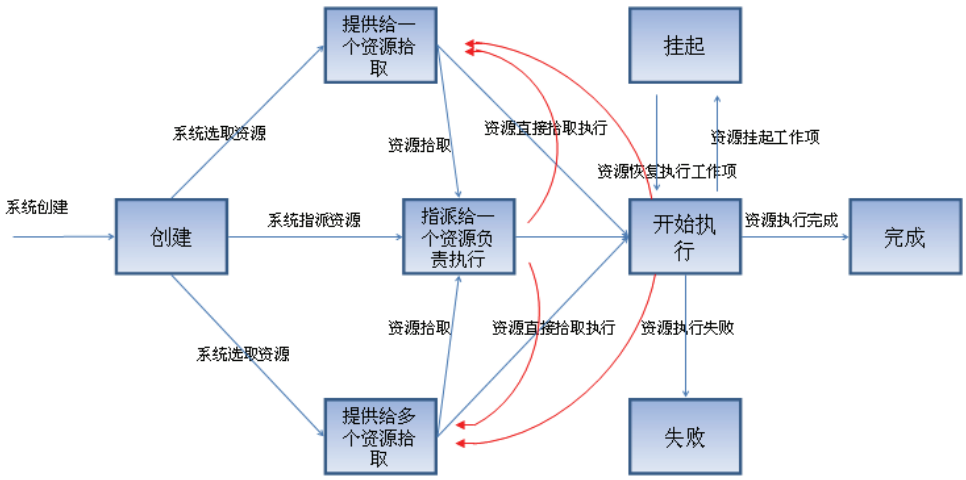
系统驱动工作分配的优化。很多时候，计划跟不上变化，工作也是这样，分配工作前有许多的考虑因素，如个人能力、工作经验、技能要求等，但在实际工作中会发现原先的资源分配并不合理，或者有些人承担了太多的职责，或者有人能力超出其目前担承的职责等等，在这种时候就需要对工作进行灵活的重新分配以到达最高的执行效率。

对流程的优化始终是一个对人的命题，而不是对机器和工具的命题，工具所能做到的只是尽可能多的提供可供参考的数据模型，例如各种报表、数据分析等，最后做出决策的还是人。所以该模式的实现也以提供给流程管理员重新分配工作项的能力为主，同时提供工作项超时的提示为辅。

退回指派（WRP_29: Deallocation）

描述

资源能够撤销指派给他的工作项，工作项可以重新分配给其他资源，如图B-44所示。



图B-44　退回指派

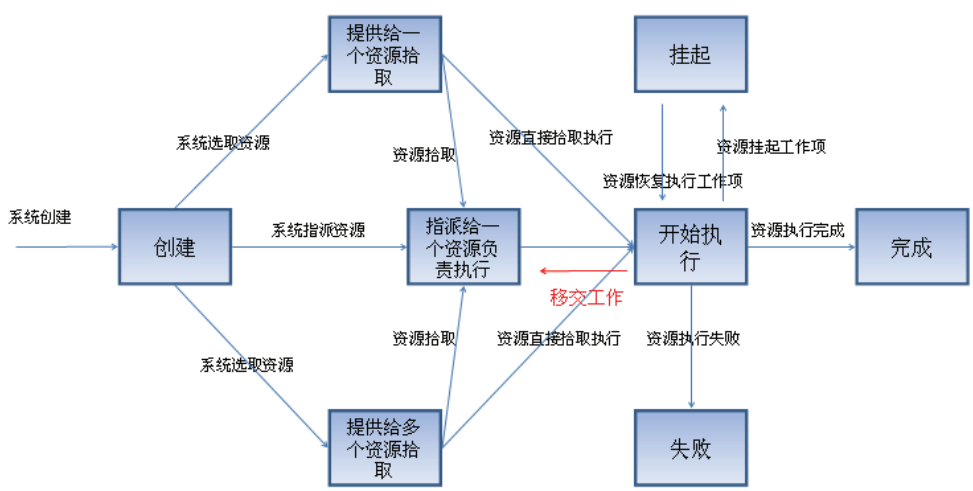
应用

资源驱动工作分配的优化。

有状态工作移交（WRP_30: Stateful Reallocation）

描述

资源能够将正在执行的工作项移交给其他资源执行，该工作的状态将得到保存，如图B-45所示。



图B-45　工作移交

应用

该模式与委派模式很相似，差别就在于委派模式是将未开始执行的工作进行重新指派执行，而该模式则是将已开始执行的工作进行重新指派执行。委派模式中的委派者仍需要为委派出去的工作负责，而移交则意味着责任的移交。

无状态工作移交（WRP_31: Stateless Reallocation）

描述

资源能够将正在执行的工作项移交给其他资源执行，该工作的状态不会得到保存。

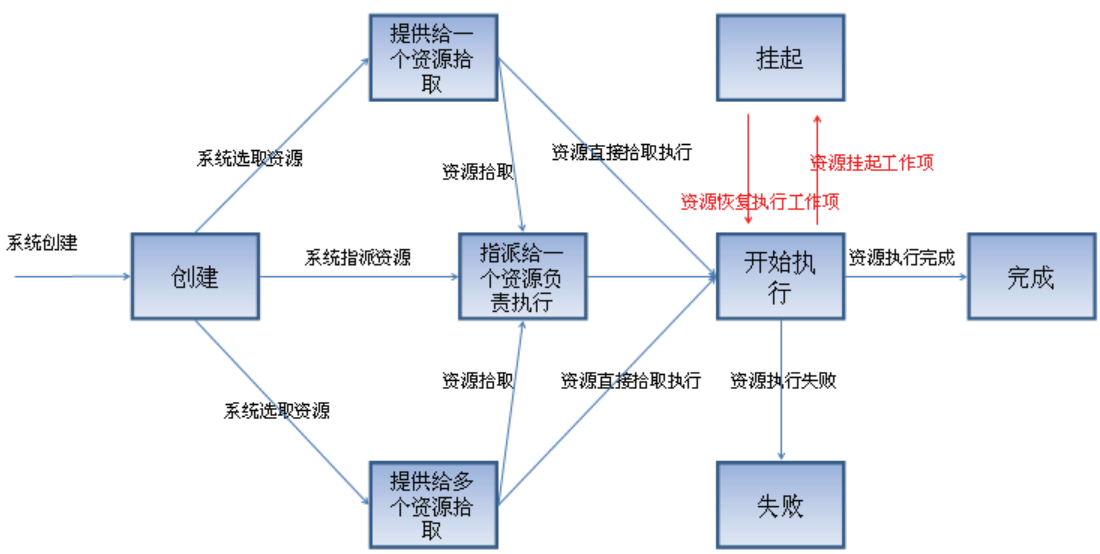
应用

工作的无状态移交意味着该工作的重新执行，原有工作对重启的工作而言没有价值。

挂起/恢复执行（WRP_32: Suspension/Resumption）

描述

资源能够挂起当前执行的工作项，并在某一个时候重新恢复执行该工作项，如图B-46所示。



图B-46　挂起/恢复执行工作

应用

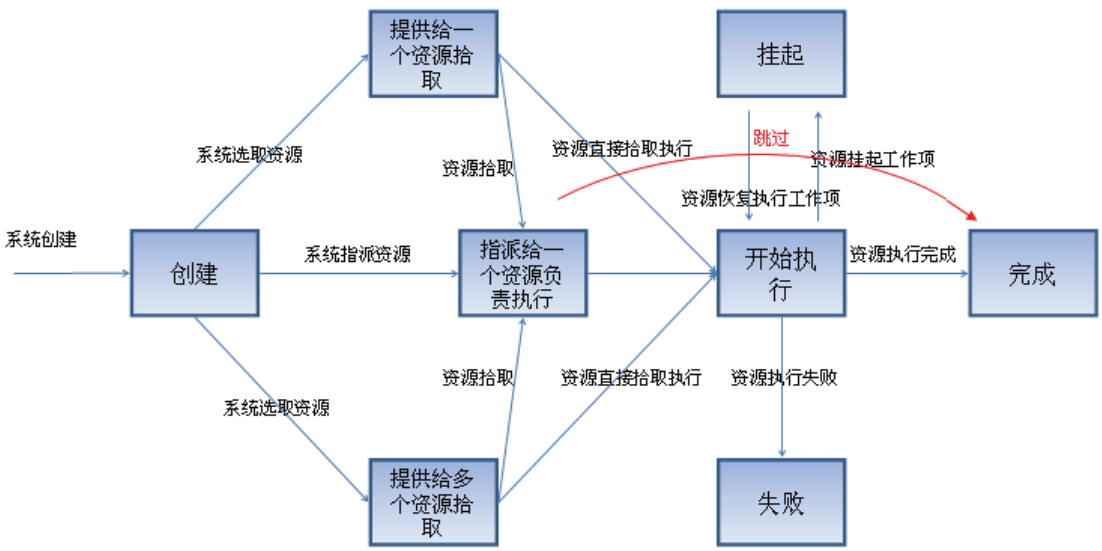
资源对分配给其的工作进行优化执行，能够根据自己和当前流程实例的实际情况合理的安排

工作执行，挂起正在执行的工作，执行当前最重要或效率最高的工作，然后再返回执行该工作。

跳过（WRP_33: Skip）

描述

资源能够选择跳过指派给他的工作项的执行，不执行该工作项，并将该工作项标识为完成，如图B-47所示。



图B-47　跳过工作的执行

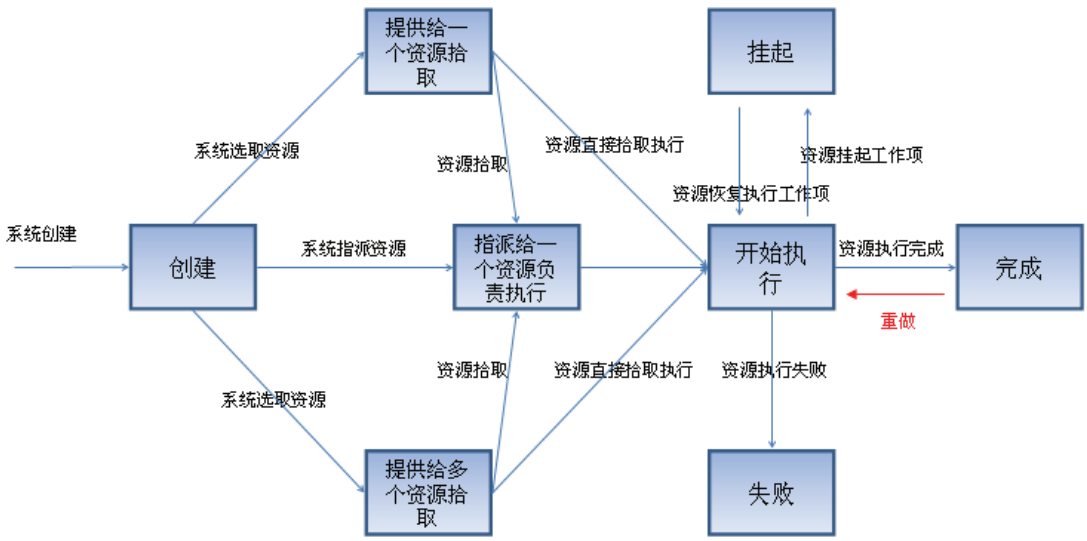
应用

因为变化，当前工作不再具有价值，选择跳过继续执行后续工作。

重做（WRP_34: Redo）

描述

资源能够对先前完成的工作项重新处理，同时，该工作的后续工作项（后续活动所对应的工作项）也将被重新处理，如图B-48所示。



图B-48　重新执行工作

应用

对已完成的工作进行重新处理并不少见，但该模式最为重要的部分还是在于要求所有后续工作的重新处理，所以该模式应用在极其重要的关键活动里。例如，非常重要的决策工作，因为后续的活动严重依赖于该工作所作出的决策，所以一旦决策发生变化，那么相应的后续工作必须都要做出变化。这也是业务敏捷性的一种反映。

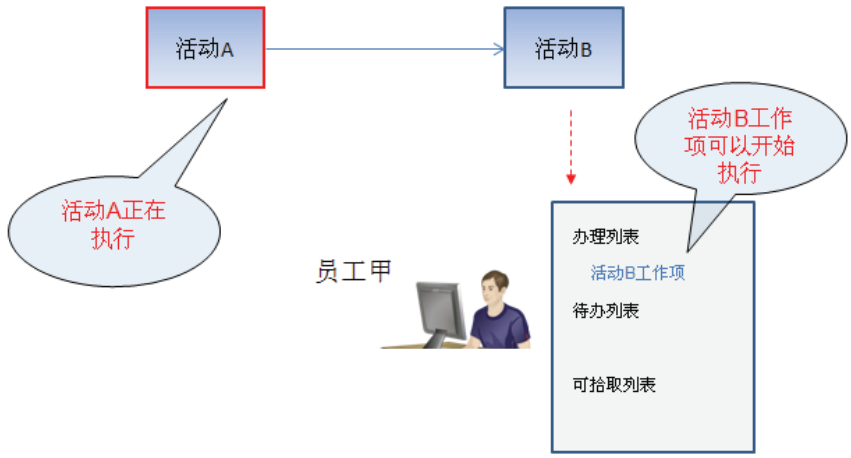
注意，该模式是一种代价高昂的应用，因为这意味着该流程实例中的所有后续工作都需要重新处理，所以如何在业务处理中尽早发现可能的环境变化并及时作出决策的调整并避免成本高昂的返工才是最重要的一点。

提前执行（WRP_35: Pre-Do）

描述

在工作实际提供或指派给资源执行之前，资源能够提前执行该工作。示例见图B-49。

该模式需要一个前提条件：活动不能依赖于前续活动的处理输出。该模式与推模式里的提前分配模式非常相似，所不同的是：提前分配强调一种通知机制，强调预先准备；而提前执行则已经可以开始实际的执行工作。



图B-49　提前执行工作

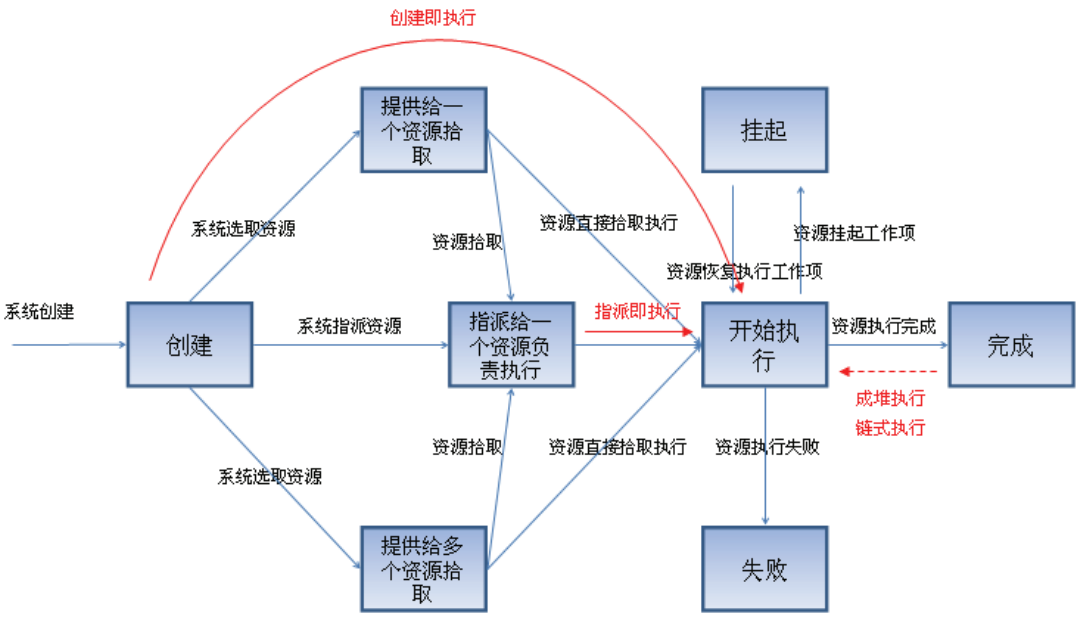
应用

和提前分配模式不同，该模式提供了一种流程活动执行的灵活机制，在预先定义的流程里，活动的执行是具有一定顺序的，在大多数情况下，这种顺序是合理的，但是在某些具体的流程实例里，某些串行执行的活动可以并行的执行以达到最好的执行效率和负载均衡，在这种情况下，就应该应用该模式并行执行部分活动。

注意：该模式仅仅引入了一种实际执行活动的灵活性，是对流程定义固化的补偿，如果在实际流程实例中频繁应用到该模式，则意味着流程定义本身需要作出调整。

自动开始模式

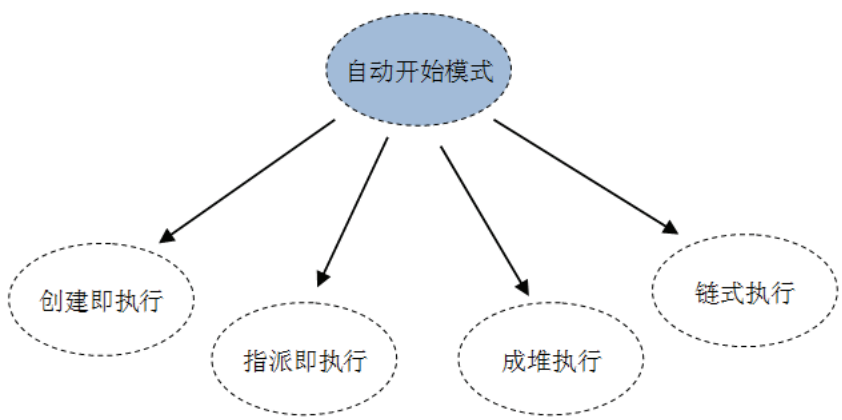
前面我们讨论了创建模式、推模式和拉模式，它们对应着工作项的一个正常生命周期：创建、提供/指派、资源选取开始执行。在前面的讨论里，工作项的执行都是由资源驱动的（从工作项待办列表里选取执行），而自动开始模式则提供了一种系统驱动工作项执行的方式，系统直接驱动工作项执行表明了该工作项的最高优先级，需要马上开始执行，如图B-50所示。



图B-50　工作项生命周期里的自动开始模式

自动开始模式共有4种如图B-51所示。

- 创建即执行：资源在工作项一创建完毕就开始执行。
- 指派即执行：资源在工作项一指派完毕就开始执行。
- 成堆执行：资源成堆执行同一活动处于不同流程实例中的不同工作项。
- 链式执行：当前一个活动的工作项执行完毕后，资源自动开始执行同一流程实例中的下一活动工作项。

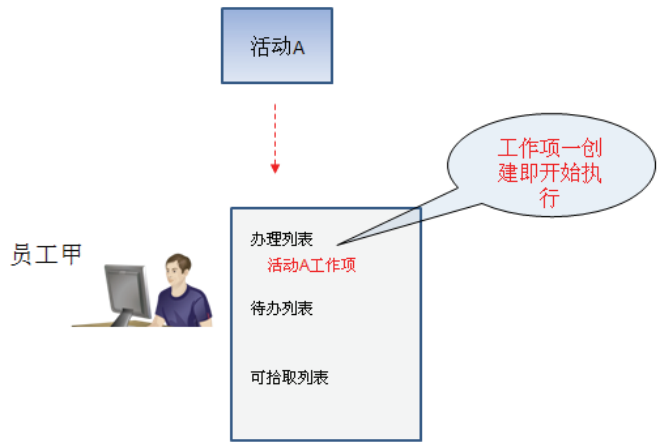


图B-51　自动开始模式

创建即执行（WRP_36: Commencement on Creation）

描述

资源能够在工作项一创建完毕就开始执行。示例见图B-52。



图B-52　创建即开始执行

应用

该模式应用在关键的优先级高的活动/流程实例里，通过系统推送，强制资源优先执行该活动/流程实例，省去活动/流程实例的等待时间。

指派即执行（WRP_37: Commencement on Allocation）

描述

资源能够在工作项一指派完毕就开始执行。示例见图B-53。

图B-53　指派即开始执行

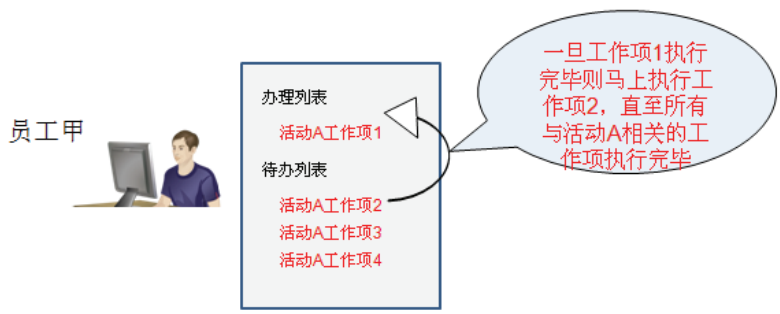
应用

该模式跳过了工作项的指派状态，是对创建即开始执行模式的扩展，在创建即开始执行模式里，工作项必须预先确定明确的执行人，不能分配给角色、岗位等资源分组，而在该模式里除了支持创建即开始执行模式里的情况，同时也提供了对这种情况的支持：工作项提供给多个资源拾取，一旦一个资源拾取则必须马上开始执行（从这个角度看，该模式与资源驱动执行-提供工作项模式是相同的）。

成堆执行（WRP_38: Piled Execution）

描述

资源能够成堆集中执行同一活动处于不同流程实例中的不同工作项。示例见图B-54。



图B-54　成堆执行工作

应用

某开发人员熟悉持续集成工具，此时同时有多个软件开发项目需要搭建持续集成环境。一旦他为某个项目组搭建了持续集成环境，那么处于执行效率的考虑，最好的方式是他一鼓作气将所有的持续集成环境都搭建完毕。

相同/相似的工作交由同一资源一并执行，这些工作具有完全或大部分相似的执行上下文（相同的知识、能力要求），从这个角度能够达到最高的执行效率。

链式执行（WRP_39: Chained Execution）

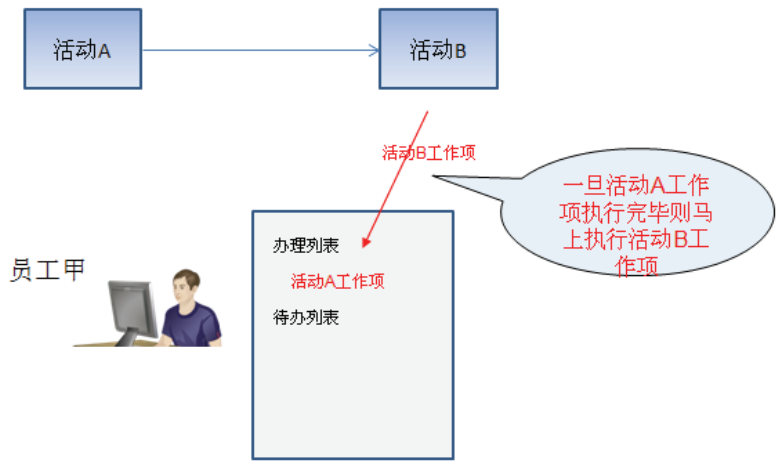
描述

当前一个活动的工作项执行完毕后，资源能够自动开始执行同一流程实例中的下一活动工作项。示例见图B-55。

应用

该模式将资源胶黏在一个流程实例上，同样是出于执行效率的考虑（两项工作位于同一流程

实例里，具有相同的执行上下文）。该模式的应用具有前提条件，即流程建模时，连续的活动由相同的资源进行处理。



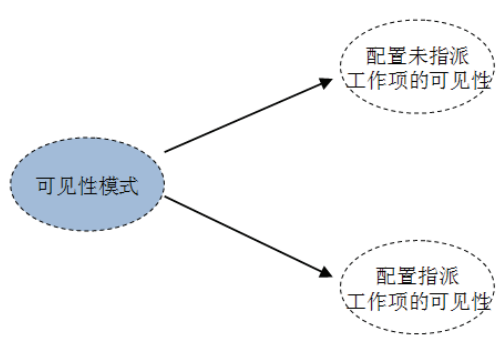
图B-55　连续执行同一流程实例中的工作

可见性模式

可见性模式讨论各种资源对工作项的可见性，不同资源由于权限的不同，对工作项拥有不同的可见范围。由于涉及权限，那么根据不同的组织机构设置，必然会出现不同的工作项权限，这里不讨论具体的工作项权限分配，仅从工作项的状态来讨论区分工作项可见性的必要性。

可见性模式包括2种，即未指派状态工作项的可见性和指派状态工作项的可见性，如图B-56所示。实际上，工作项处于执行状态或完成状态对不同资源也存在不同的可见性。

- 未指派状态工作项的可见性：能够配置未指派工作项对不同资源的可见性。
- 指派状态工作项的可见性：能够配置已指派工作项对不同资源的可见性。

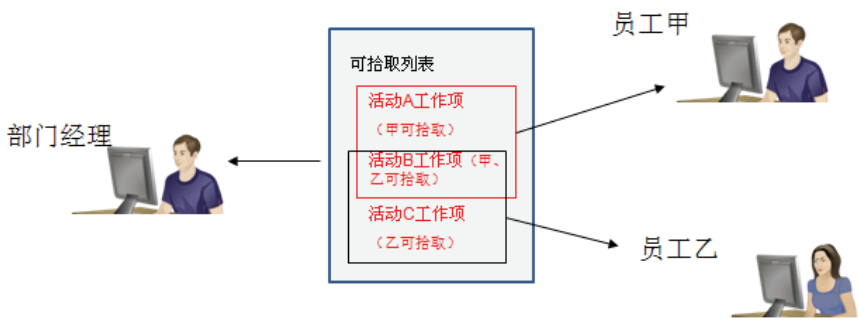


图B-56　可见性模式

配置未指派工作项的可见性（WRP_40: Configurable Unallocated Work Item Visibility）

描述

能够配置未指派工作项对不同资源的可见性。活动A工作项、活动B工作项和活动C工作项。员工甲可拾取的工作项包括：活动A和活动B工作项；员工乙可拾取的工作项包括：活动B和活动C工作项，那么由此产生的可见性是：员工甲只能看到活动A和活动B工作项，员工乙只能看到活动B和活动C工作项。而作为员工甲和员工乙的部门经理，他需要了解每个属下的工作情况，所以他可以看见所有甲乙可见的工作项，见图B-57所示。

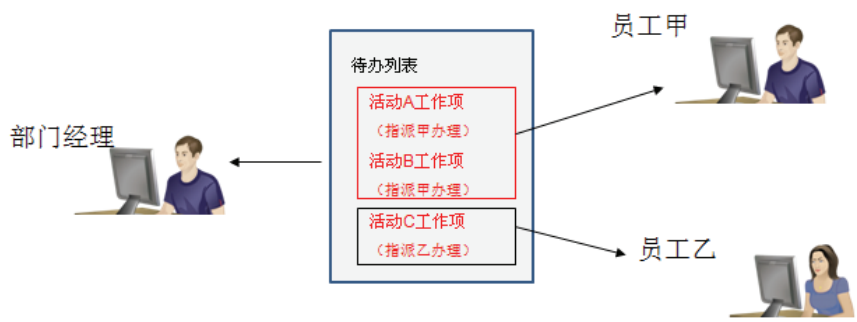


图B-57　配置未指派工作的可见性

配置指派工作项的可见性（WRP_41: Configurable Allocated Work Item Visibility）

描述

能够配置已指派工作项对不同资源的可见性。示例见图B-58。



图B-58　配置已指派工作的可见性

应用

随着企业规模的发展，几乎所有企业的组织模型都会形成金字塔型的结构，一方面是出于分工的需要，另一方面则是出于管理的需要，每一层级的人员都需要对上一级负责，同时管理下一层级的人员。处于管理的需要，管理者需要了解下属的工作情况，这样权限就自然产生了，具体到工作流的活动里，管理者需要对其所管理下属的工作具有可见性。

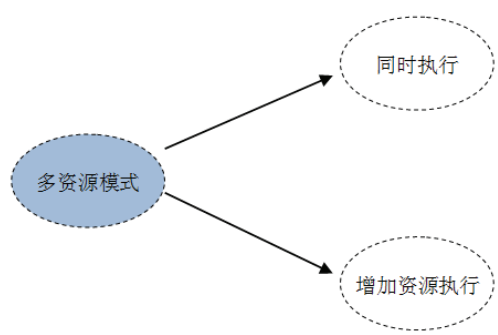
不仅仅是对于工作项，对于流程实例本身也具有可见性的权限分配。对流程实例负责的人必然具备最大的可见性和权限，流程根据活动分解，如果仅仅只对某一活动负责，那么则只对该活动的工作项具有可见性，而如果需要对多个活动负责，那么就需要对多个活动的工作项具有可见性，最直接的责任人就是具体执行该活动工作项的人员，但是引入管理的层级后，职责的承担也会形成层级的关系，从上至下层层承担，此时担负最大职责的人员不再是具体的工作执行人员，而是管理人员。

多资源模式

到目前为止，我们讨论的工作项都是与某一特定资源一一对应的，即一个工作项只能由一个单一资源执行，或者严格来说，一个工作项在任何时间段都只能由一个单一资源执行（考虑到工作移交的情况）；同时，一个资源在任何一个时间段都只能处理一个工作项。

多资源模式将会讨论两种不同的情况：一个资源同时执行多个工作项、多个资源执行同一个工作项。多资源模式包括2种，如图B-59所示。

- 同时执行：资源同时执行多个工作。
- 增加资源执行：为正在执行的工作增加更多的资源。

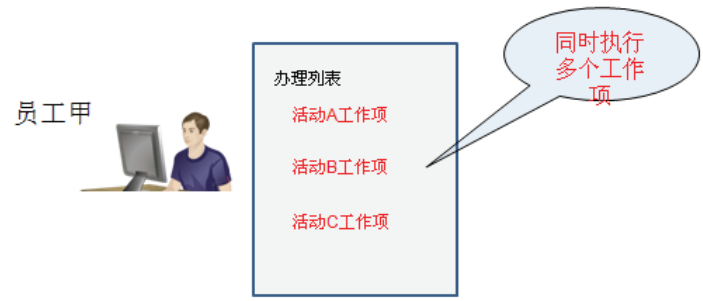


图B-59　多资源模式

同时执行（WRP_42: Simultaneous Execution）

描述

资源能够同时执行多个工作项。示例见图B-60。



图B-60　同时执行多个工作

应用

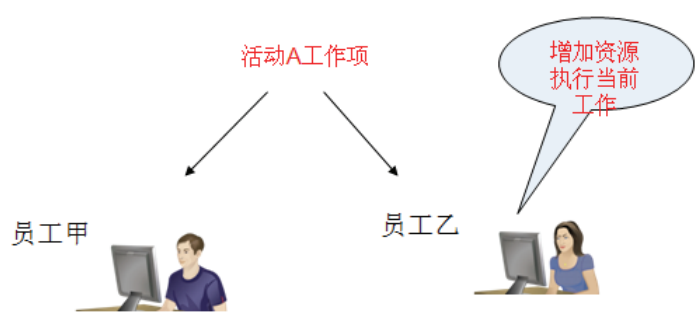
和计算机一样，虽然在任何时刻都只能处理一项工作，但是通过将多项工作切分成多个线程交替执行，从某个时间段看，资源能够同时处理多项工作。资源能够选取相关联的多个工作，同时开始执行，在执行的过程中，合理安排这些工作的执行时机和顺序。

尽管该模式引入了选择工作执行的灵活性，但我们的观点和温伯格一致：让开发人员最没效率的事就是让他同时做许多事。

增加资源执行（WRP_43: Additional Resources）

描述

资源能够要求增加资源来处理他正在执行的工作项。示例见图B-61。



图B-61　增加资源执行工作

应用

根据工作的难易和执行情况，动态的增加资源。

小结

在本章里，我们讨论了工作流的43种资源模式，这些模式分为7类，分别是创建模式、推模式、拉模式、折回模式、自动开始模式、可见性模式和多资源模式。

创建模式在系统创建工作项时生效，其位于工作项生命周期的创建阶段，创建模式作为流程模型的组成部分在流程定义期、在活动节点的定义里进行定义，与资源关联，用来限定可执行该活动的资源范围。系统根据创建模式限定的资源范围生成工作项。

接下来，系统需要将工作项推送给相关的资源进行执行，这个推送的过程即是推模式所包含的内容。工作流系统通过工作项管理器即不同类型的工作项列表与用户进行交互，这里的推送可以理解为系统将生成的工作项推送至相应资源的工作项列表里。

推模式的主语是系统，由系统将工作项推送至资源的工作项列表，那么，接下来的主动权交由单个资源本身，由其拉动工作项的执行，这是拉模式所包含的内容。

实际工作中，工作的执行状态不可能总是与预想相符的，总会出现各种各样的情况，例如重新分配、重做、挂起等。折回模式对应着这些情况，折回代表着工作项状态的反复、回退。

自动开始模式提供了一种系统驱动工作项执行的方式，系统直接驱动工作项执行表明了该工作项的高优先级，需要马上开始执行。

可见性模式讨论各种不同资源对工作项的可见性，工作项自身作为资源与权限相关。

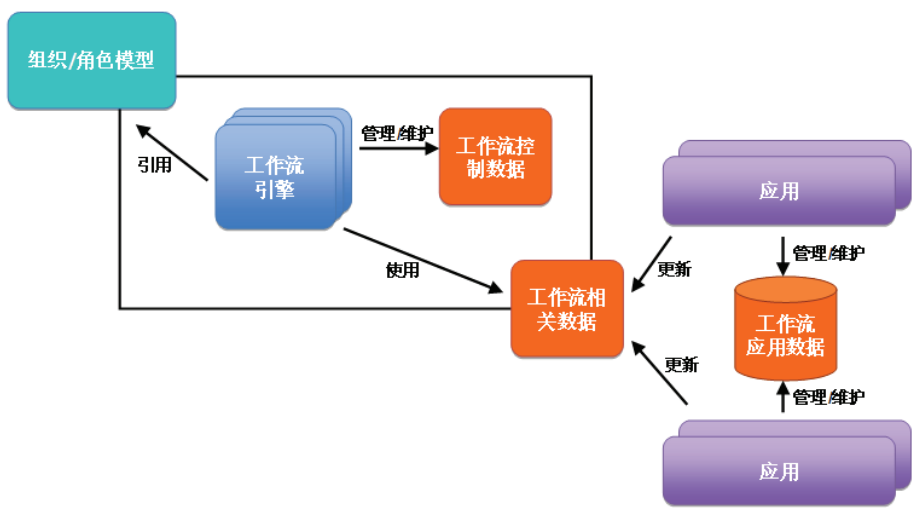
多资源模式讨论一个资源执行多个工作项和多个资源执行同一个工作项的情况。

从这些模式的讨论可以看出，这些模式关注的是组织内部资源的协调，关注通过合理分配工作和调配工作的执行为组织带来最高的执行效率。但组织内部资源协调所包含的内容远远超出这些模式，如何让组织内所有人目标一致向着一个方向努力，如何将最合适的人放到最合适的岗位，如何建设闭环团队避免跨团队的沟通成本，这些都是实际工作中我们所要不断思考的。人，永远是管理中最重要的因素。

附录C
工作流数据模式

正如语言是人与人之间的沟通方式一样，数据是IT系统之间的沟通方式，语言之间的沟通总是有效，数据交互却未必，因为除了让计算机理解之外，数据还需要让人理解，IT系统是对现实生活的映射，也正因为如此，现在数据之间的沟通也在向语言靠拢即语义化（REST/语义网）。

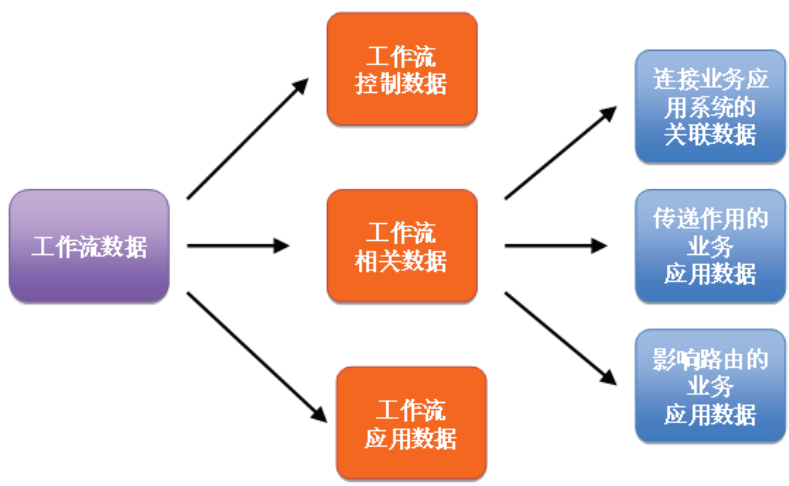
在WfMC的工作流模型里，工作流数据被分为了3类，如图C-1所示。



图C-1　WfMC的工作流数据分类

- 工作流控制数据：工作流系统管理的内部控制数据，这些数据包括了与流程实例和活动实例相关的执行数据和状态数据，例如流程实例的状态、执行时间、工作项的执行者、执行时间、状态、紧急程度等。
- 工作流相关数据：工作流系统使用工作流相关数据确定流程实例的流转条件，并选择下一个将执行的活动，这些数据由业务系统访问并修改。例如报销流程中的“报销金额”，这个数据会决定该流程的审批路径；再例如为活动设置的超时时间，这个数据会触发活动的取消。这些数据是工作流系统需要依赖进行流程流转的业务应用数据。
- 工作流应用数据：业务系统管理的业务数据，工作流系统不能访问。

我们遵循WfMC的工作流数据分类，区别是将工作流相关数据根据应用场景进一步细化为3类并重新定义，如图C-2所示。

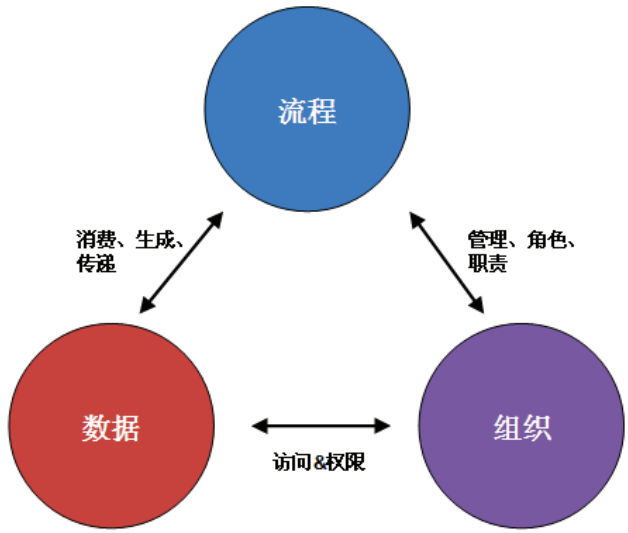


图C-2　工作流数据分类

我们将工作流相关数据泛化为为工作流系统能够访问并使用的业务应用数据，分为3类：

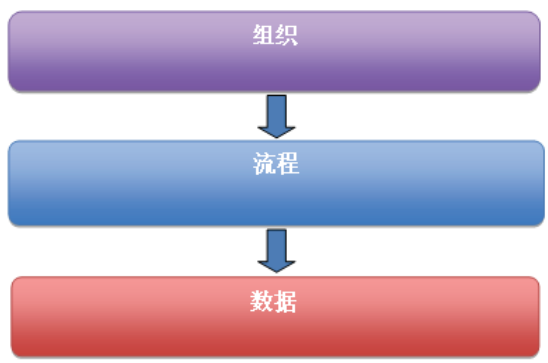
- 连接业务系统的关联数据：工作流系统与业务系统进行关联的数据，例如特定于Web系统，工作流系统会在每个流程/活动实例里保持有导航至对应业务表单的URL。
- 传递作用的业务应用数据：当流程跨越多个业务模块时，需要在模块间传递数据，此时会利用工作流系统进行传递，在工作流系统里暂时存储或转换这些业务数据。在面向服务的软件架构中（SOA），工作流系统作为重要的中间件负责服务之间的调用编排，业务应用数据被封装为SDO通过工作流系统在不同Web服务（业务系统）间传递。
- 影响路由的业务应用数据：和WfMC对工作流相关数据的定义一致。

流程、数据和组织是工作流应用中最重要的三个方面，它们紧紧地依赖在一起，如图C-3所示。



图C-3　流程、数据与组织

在以流程为核心模型的软件架构里，系统功能通过流程进行协调，流程面向组织，如图C-4所示。



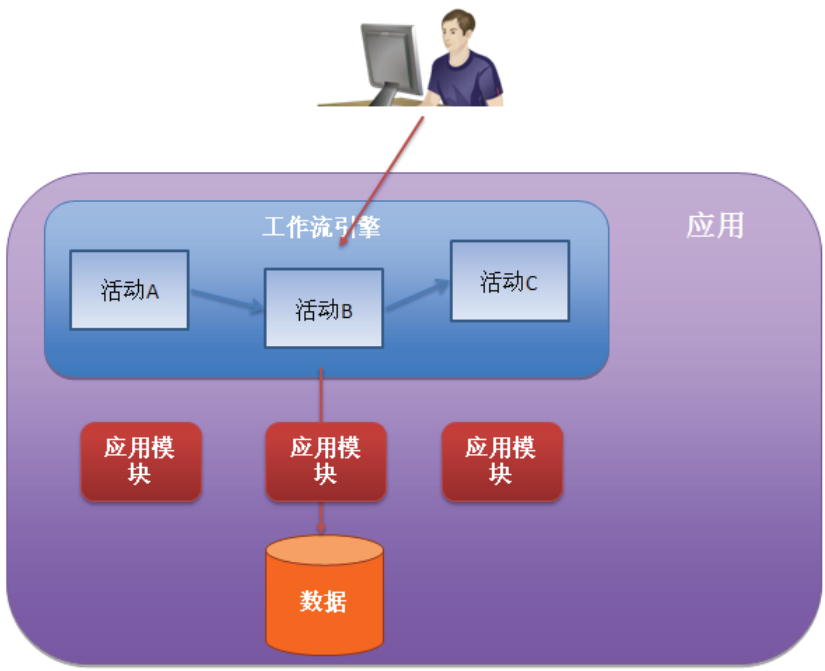
图C-4　面向组织的流程

流程作为整个应用的入口，流程模型中包含组织模型（部门/人员/角色），流程通过活动定义界定出应用的各个业务上下文，并由此界定组织中部门/人员/角色的职责、权限和互相协作的规则。

以流程为核心的软件架构有2种应用场景：独立业务应用系统的工作流嵌入式使用和业务应用系统的企业内集成。

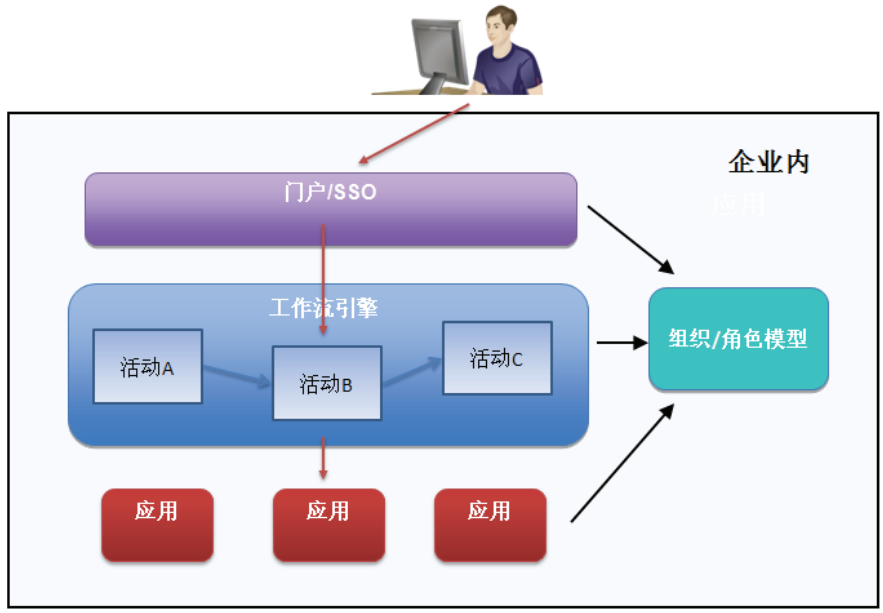
在独立业务应用系统的工作流嵌入式使用中，工作流系统作为内部组件对应用系统进行流程逻辑的横切。试想一个需要多人处理的电力缴费流程，在引入工作流系统之前，我们需要为每个

业务表单设置一个状态位，以此来进行业务处理状态的跟踪。如果流程固定，那么这样做并没有什么不好，例如财务软件、海关报关软件等，它们的流程虽然复杂但是不常改变，此时就没有必要引入工作流系统。但是对于另外一些情况，例如制造业的订单处理、库存管理、政府的协同办公等，流程经常需要定制修改，此时如果继续由业务应用系统自己处理流程逻辑那么成本将会很高，如图C-5所示。



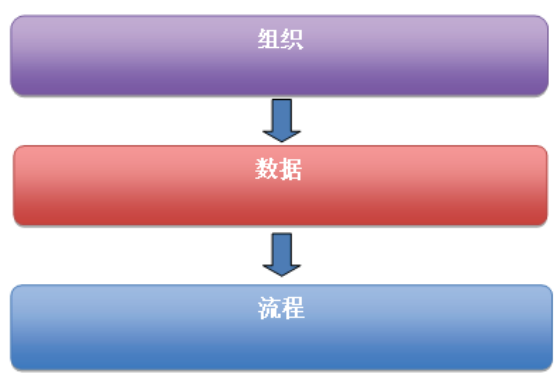
图C-5　嵌入工作流的独立应用系统

在使用工作流进行企业内应用系统的集成时，工作流系统作为独立的服务部署。在上规模的企业里，很多流程会涉及不同的业务功能，例如报价、订单审核、资产核准、绩效评估等，这些流程经常会跨越不同的部门和业务应用系统。工作流系统此时扮演的就是集成角色，由其通过定制流程将这些业务应用系统撮合起来，实现企业内各部门的信息流动和协作。此时，用户登录门户，通过工作流系统推送的统一活动列表导航到各个业务应用系统进行工作，所有应用系统使用统一的组织模型，如图C-6所示。



图C-6　使用工作流进行企业内应用系统的集成

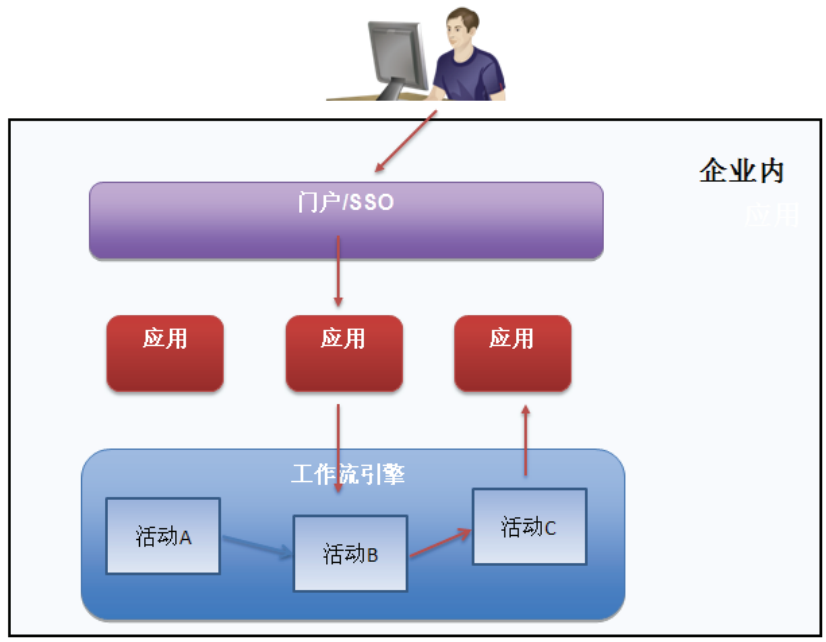
在以数据为核心模型的软件架构里，流程面向数据，系统功能分散在各个应用系统中，流程通过编排服务，在不同应用系统间传递/同步数据，以数据进行业务的驱动。流程模型不包含组织模型，流程的本质是数据流，反映数据的流向，如图C-7所示。



图C-7　面向数据的流程

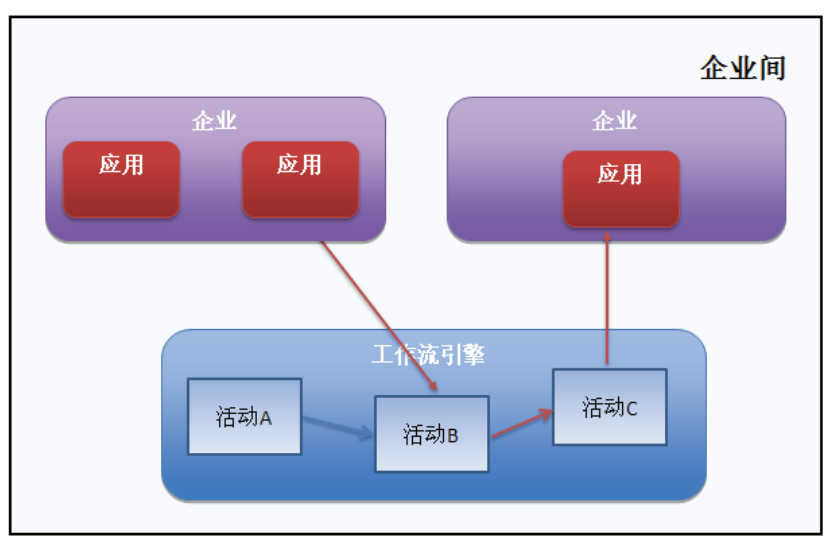
以数据为核心的软件架构同样有两种应用场景：业务应用系统的企业内集成和企业间集成。

用户登录门户，进入各个业务应用系统进行工作，应用系统之间的数据同步由工作流系统完成，活动的触发由各个应用系统自己负责，如图C-8所示。



图C-8　使用工作流同步数据进行企业内应用系统的集成

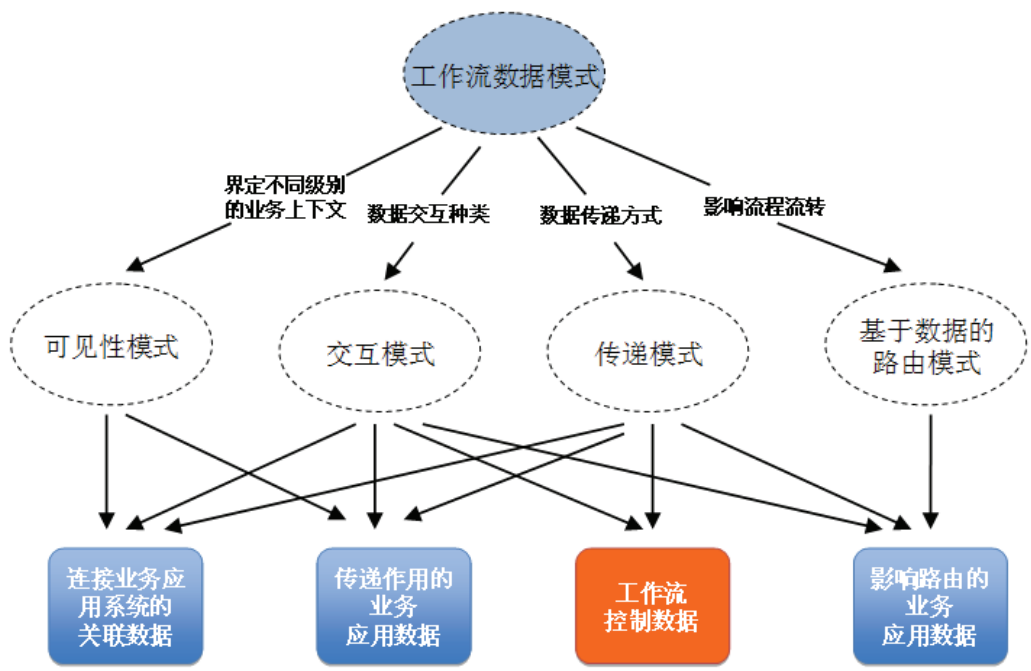
企业间进行应用系统集成，如B2C、B2B流程，需要在企业间同步传递数据。此时，数据的传递是核心，如图C-9所示。



图C-9　使用工作流进行企业间的应用系统集成

以流程为核心模型和以数据为核心模型之间最大的区别在于流程面向的对象。流程面向组织反映出组织职责的可视化，出发点是企业的管理；流程面向数据反映出跨不同应用系统的业务数据流，在处理企业间的业务流程时是唯一的选择。实际应用时，经常混搭两种架构风格，例如使用工作流同步数据进行企业内应用系统的集成，同时在单个应用系统里使用嵌入式工作流。

工作流数据模式共有40种，根据关注点的不同，分为4组：数据可见性模式、数据交互模式、数据传递模式和基于数据的路由模式。数据可见性模式关注工作流数据的作用范围，讨论到的数据包括连接业务系统的关联数据和传递作用的业务应用数据；交互模式关注工作流系统各组件以及外部环境之间可能存在的数据交互种类，讨论到的数据包括工作流控制数据和工作流相关数据；传递模式关注数据在工作流系统各组件以及外部环境之间的传递方式，讨论到的数据包括工作流控制数据和工作流相关数据；基于数据的路由模式关注数据影响工作流流程执行的方式，讨论的数据为影响路由的业务应用数据，如图C-10所示。



图C-10　工作流数据模式的分类

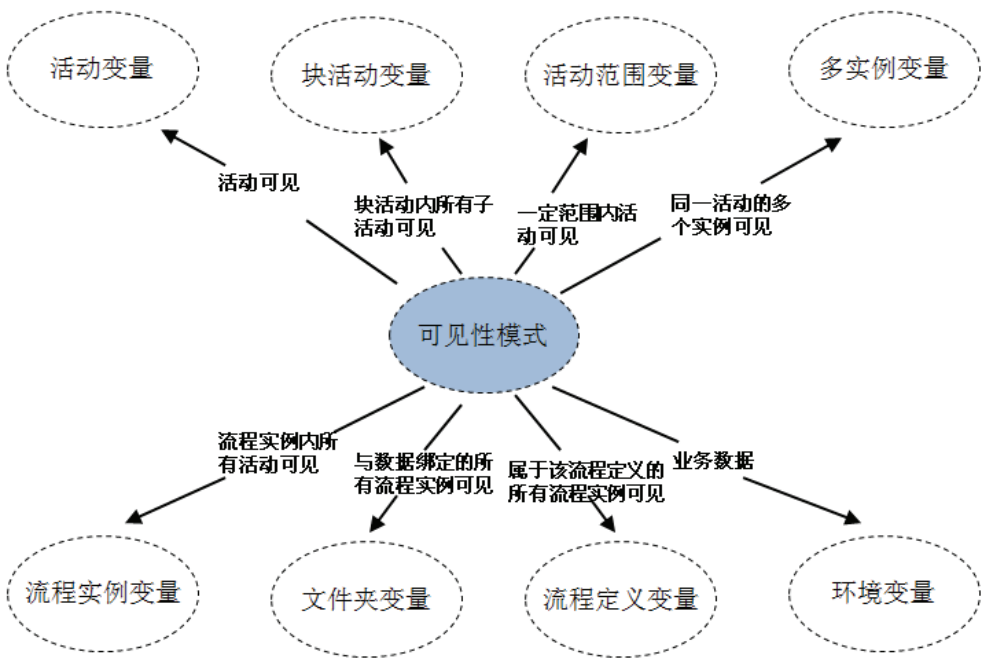
出于统一的目的，我们对工作流数据进行约定：我们使用def var ${变量名}定义变量，同时def var ${变量名}的声明位置决定了该变量的作用范围。图C-11中，我们在活动B上定义了一个名为M的变量，表明它的作用范围为活动B，即在一个流程实例里仅活动B的实例可见。我们使用use(${变量名})表明对该变量的使用；使用pass(${变量名})表明该变量值的传递。图C-11里，变量M的值被从活动B传递至活动C。

图C-11　工作流数据的约定

对于数据类型，典型的有string、integer、float、boolean、date和time，很多工作流系统使用序列化和反序列化支持存储任意类型的数据类型，如数组、集合和对象。

可见性模式

共有8种可见性模式，它们建立起不同工作流组件级别的变量作用范围。



图C-12　工作流数据可见性模式

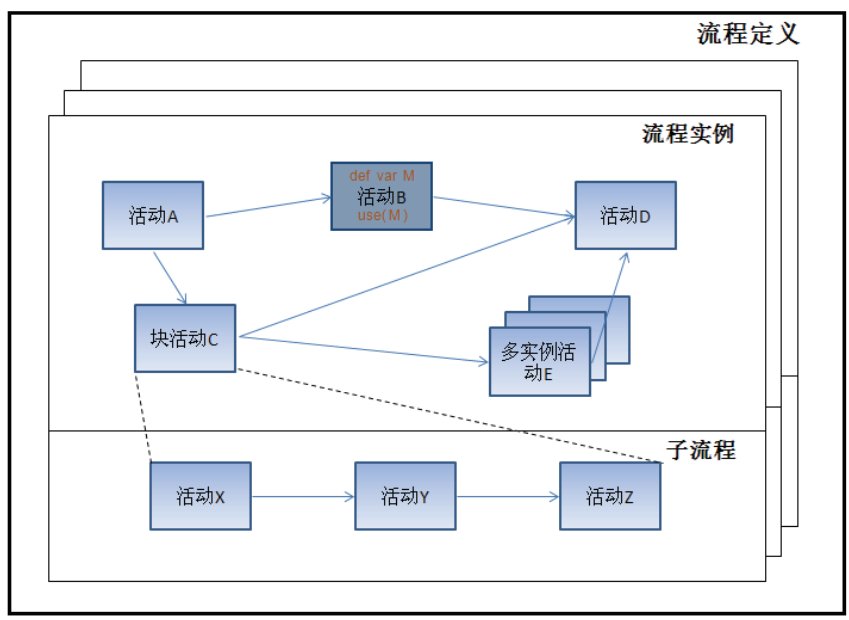
- 活动变量：在单个活动上定义变量，变量的作用范围为该活动。
- 块活动变量：在块活动上定义变量，变量的作用范围为块活动所包含的所有子活动。
- 活动范围变量：为一定范围内的活动定义变量，变量的作用范围为范围内的所有活动。
- 多实例变量：在多实例活动上定义变量，变量的作用范围为所有属于该活动的工作项。
- 流程实例变量：为流程实例定义变量，变量的作用范围为该流程实例里所有的活动。
- 文件夹变量：定义变量集合，当启动流程实例时，流程实例与该变量集合绑定，变量的作用范围为所有共享该变量集合的流程实例。

- ❑ 流程定义变量：为流程定义定义变量，变量的作用范围为所有属于该流程定义的流程实例。
- ❑ 环境变量：变量是应用级别的业务数据、跨业务系统的数据。

活动变量（WDP_1: Task Data）

描述

活动能够定义变量，在一个流程实例里，该变量只能被其活动实例所使用。



图C-13　活动级别的数据可见性

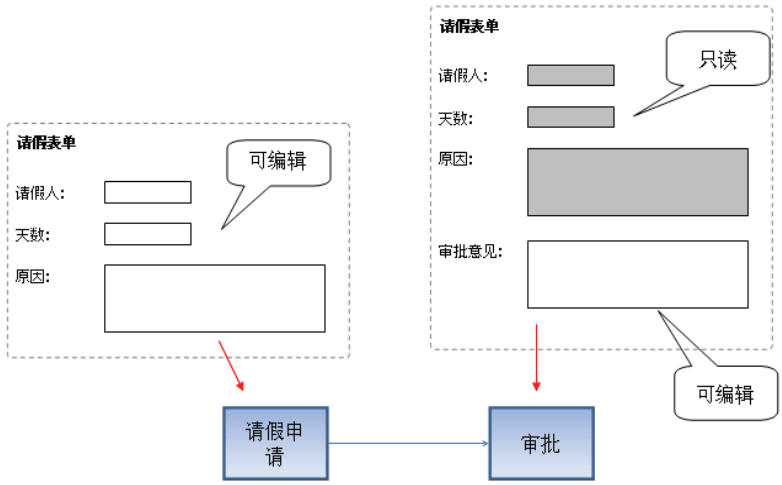
活动变量与业务数据权限控制

在应用工作流系统时，我们经常碰到这样的问题：一个流程实例中的不同活动对业务数据拥有不同的权限，如图C-14所示。

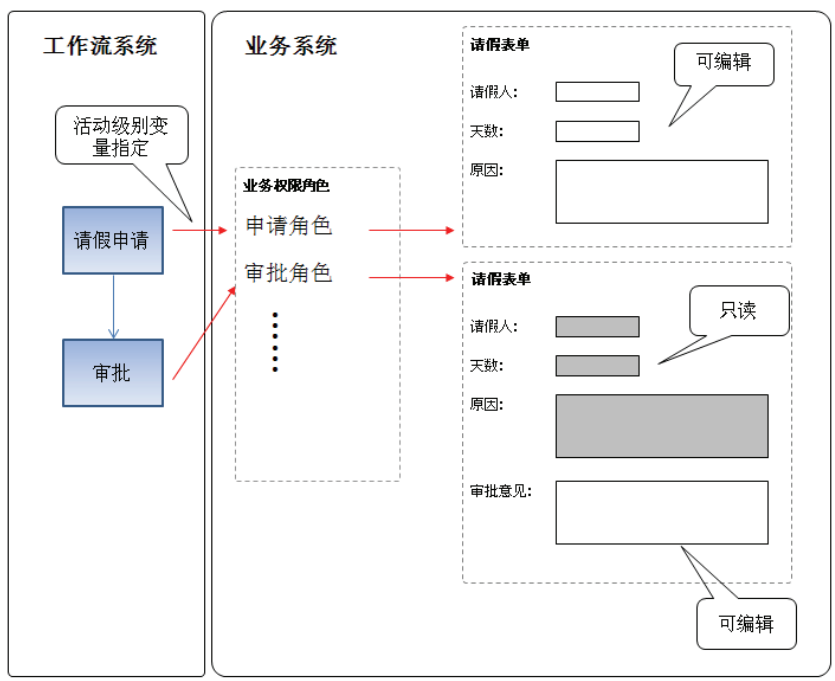
在执行请假申请活动时，申请者编辑请假人、天数和原因3个字段；而到审批活动时，审批者增加了一个可编辑的审批意见字段，但其余3个字段变为只读字段。我们将这类问题统称为与流程相关的业务数据权限控制。

此时，我们在业务系统里引入业务权限角色的概念，通过该角色隔离开工作流系统与业务数据权限，即业务数据权限的管理属于业务系统范围（由业务系统实现）。在定义好业务系统的权限角色后，我们通过活动级别的变量将流程中的具体活动与业务权限角色绑定，这样就实现了流

程活动与业务数据权限的挂接，同时又保持了工作流系统的单一职责。



图C-14　与流程相关的业务数据权限

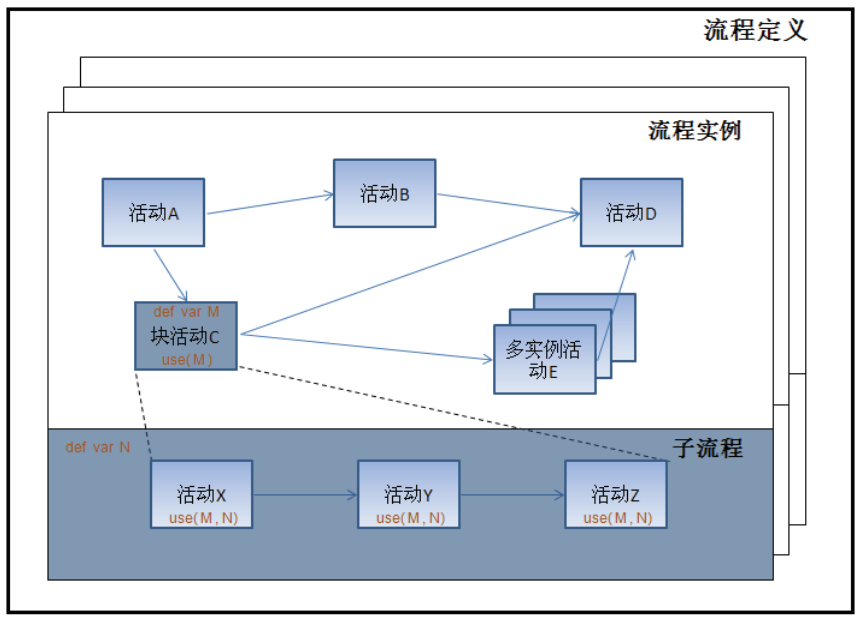


图C-15　流程相关的业务数据权限控制

块活动变量（WDP_2: Block Data）

描述

块活动能够定义变量，在一个流程实例里，其所包含的子活动实例能够使用这些变量。



图C-16　块活动级别的数据可见性

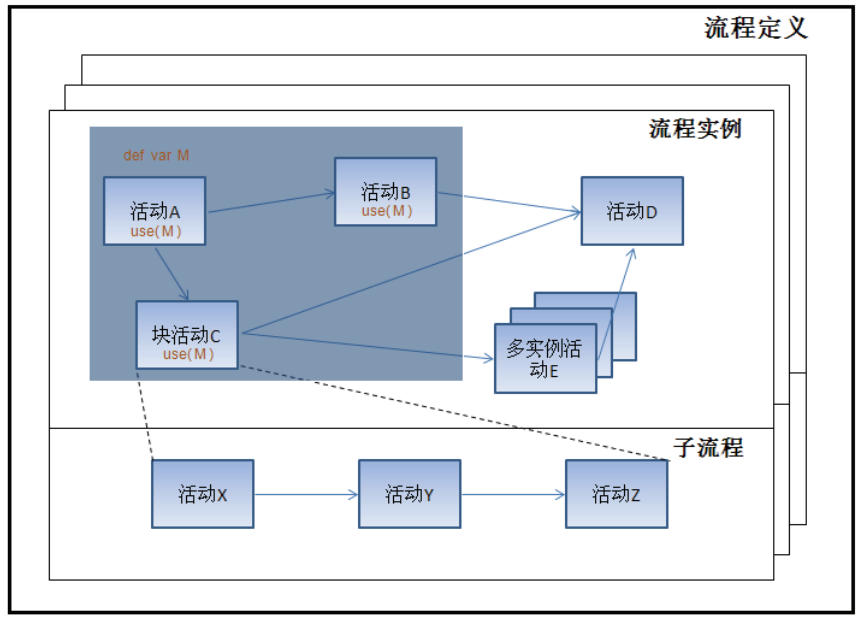
应用

在块活动为子流程的情况下，块活动级别变量最重要的职责就是初始化子流程实例：在父流程实例里为子流程实例的第一个活动指定参与者，传递子流程实例所必需的业务数据。

活动范围变量（WDP_3: Scope Data）

描述

一定的活动范围能够定义变量，在一个流程实例里，该范围所包含的活动实例能够使用该变量。



图C-17　活动范围级别的数据可见性

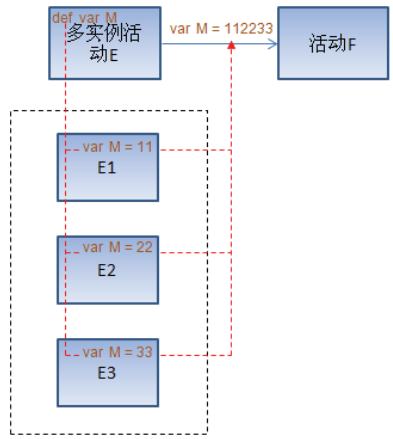
活动范围和子流程在概念上比较相似，都是包含一系列的子活动，它们之间的差别在于：子流程具有比较独立的上下文（例如子流程由另外一个部门执行）和执行环境，能够复用，而活动范围则是对流程中活动的一种分组，脱离开当前流程的上下文，这些活动无法执行。

多实例变量（WDP_4: Multiple Instance Data）

描述

多实例活动能够定义变量，在一个流程实例里，该活动的每一个工作项都能够初始化该变量，并独立使用。

什么情况下会应用到多实例活动？当一个活动需要多人共同参与并且实际执行过程中互不影响时，我们会使用多实例活动对该业务场景进行建模。重要的有两点：一是每个工作项**一定**具有相同的业务上下文；二是执行过程各自独立。对多实例活动而言，最重要的就是在各个工作项执行完毕后进行数据的汇总。根据不同的业务场景，汇总策略不同。例如在企业决策里，当多人同时决策时，采取少数服从多数；而在进行员工某项关键技能考核时，采用一票否决制。

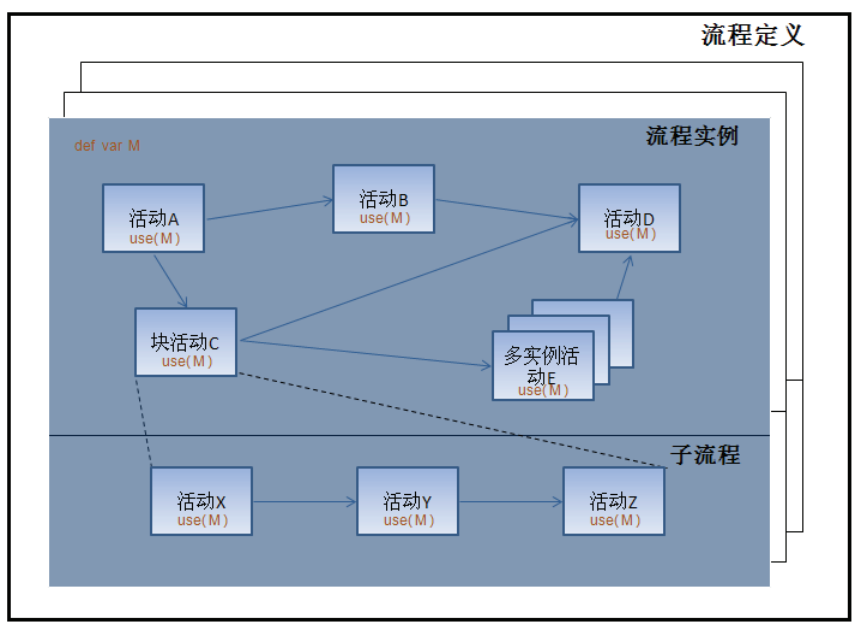


图C-18　多实例活动级别的数据可见性

流程实例变量（WDP_5: Case Data）

描述

流程定义能够定义变量，在一个流程实例里，该流程实例中的所有活动实例都能够使用这些变量。



图C-19　流程实例级别的数据可见性

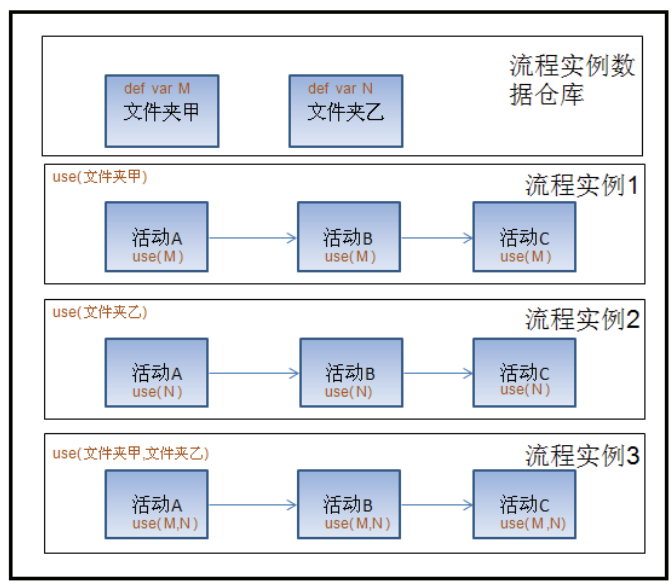
应用

流程实例变量是应用最广泛的工作流变量，一方面是因为很多工作流系统只支持流程实例级别的工作流变量，另一方面是整个流程实例所共享的上下文依赖于流程实例变量的建立，存在两个最重要的流程实例变量：一个是业务领域模型的唯一标识符ID，另外一个是该领域模型对应业务表单的URL。

文件夹变量（WDP_6: Folder Data）

描述

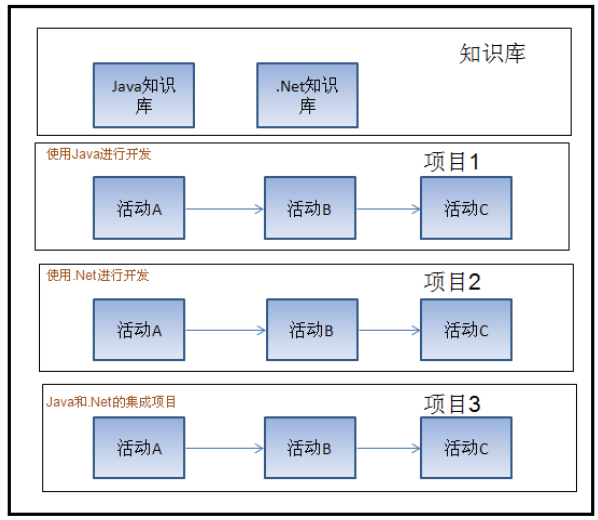
流程定义能够定义一组变量，我们把这些变量的集合称为文件夹，当启动一个流程实例时，流程实例能够与这些变量绑定，一旦绑定，该流程实例里的所有活动实例就能够使用该文件夹里的变量。



图C-20　文件夹级别的数据可见性

应用

不同的流程实例之间能够选择性的共享数据。如建立与流程相关的知识库应用。如图C-21所示。

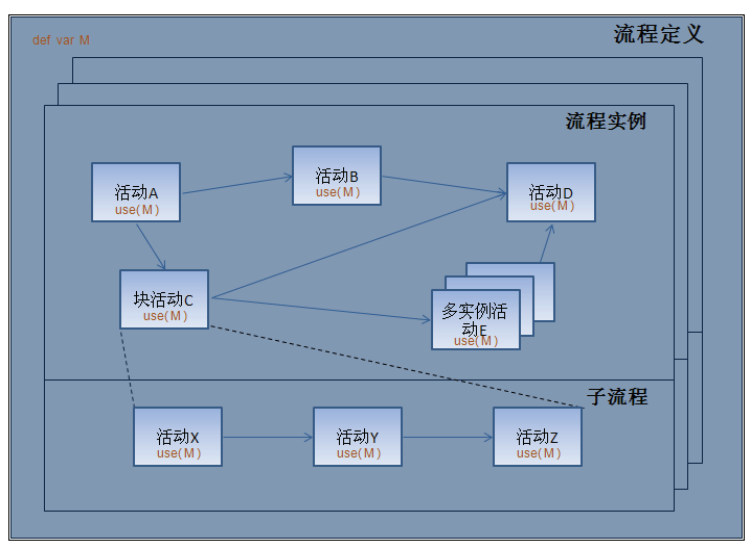


图C-21　使用文件夹变量建立活动与知识库的关联

流程定义变量（WDP_7: Workflow Data）

描述

流程定义能够定义变量，所有属于该流程定义的流程实例都能使用这些变量，这些变量在其所有流程实例的活动间是共享的。

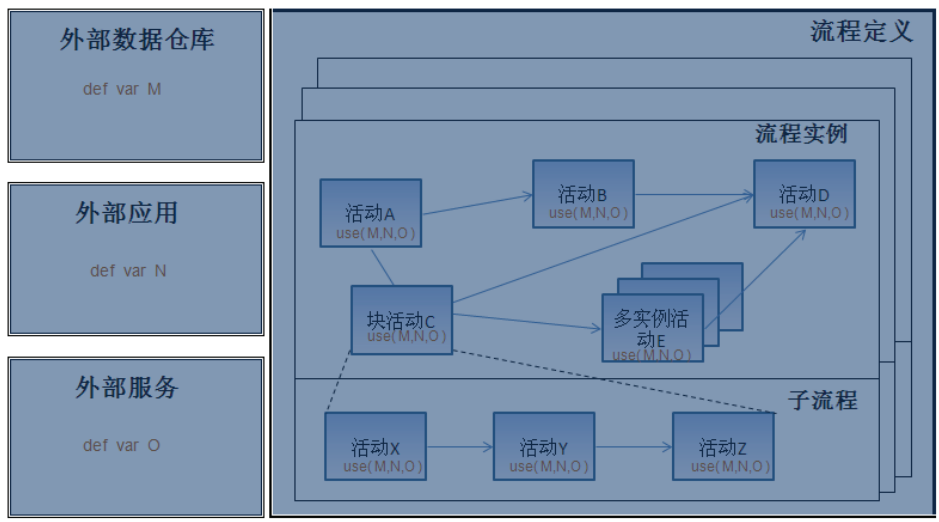


图C-22　流程定义级别的数据可见性

环境变量（WDP_8: Environment Data）

描述

流程实例里的活动能够在运行期使用外部环境的变量，这里的外部环境包括了外部数据仓库（数据库）、外部应用和外部服务等。



图C-23　环境级别的数据可见性

应用

该模式强调工作流系统的集成能力，工作流系统在执行时需要具备从外部获取和交换数据的能力。

交互模式

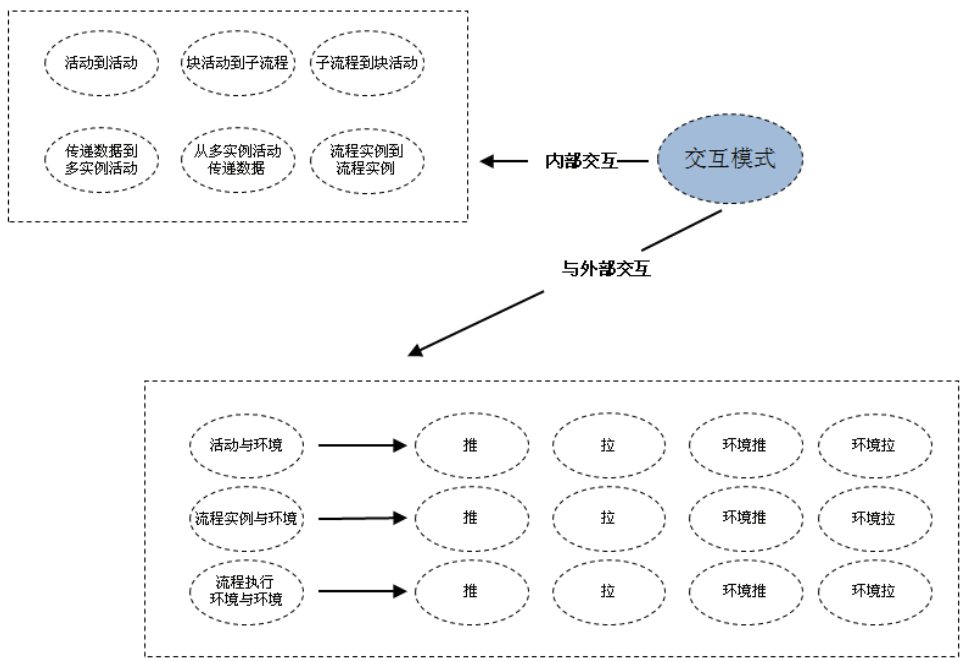
交互模式共有18种，讨论工作流系统各组件以及外部环境之间可能存在的数据交互类型。交互模式分为两组：内部数据交互和外部数据交互。

内部数据交互有如下6种。

- ❑ 活动到活动模式：活动之间传递数据。
- ❑ 块活动到子流程模式：块活动给子流程传递数据。
- ❑ 子流程到块活动模式：子流程给块活动传递数据。
- ❑ 传递数据到多实例活动模式：活动传递数据给后续多实例活动。
- ❑ 从多实例活动传递数据模式：多实例活动给后续活动传递数据。
- ❑ 流程实例到流程实例模式：流程实例之间传递数据。

外部数据交互有如下12种。

- ❑ 活动推数据到环境模式：活动给外部环境传递数据。
- ❑ 活动从环境拉数据模式：活动从外部环境获取数据。
- ❑ 环境推数据到活动模式：活动接受并使用外部环境主动传递给它的数据。
- ❑ 环境从活动拉数据模式：活动接受外部环境的数据请求，并传递数据给外部环境。
- ❑ 流程实例推数据到环境模式：流程实例给外部环境传递数据。
- ❑ 流程实例从环境拉数据模式：流程实例从外部环境获取数据。
- ❑ 环境推数据到流程实例模式：流程实例接受并使用外部环境主动传递给它的数据。
- ❑ 环境从流程实例拉数据模式：流程实例接受外部环境的数据请求，并传递数据给外部环境。
- ❑ 流程执行环境推数据到环境模式：流程执行环境（流程定义级别的数据）给外部环境传递数据。
- ❑ 流程执行环境从环境拉数据模式：流程执行环境从外部环境获取数据。
- ❑ 环境推数据到流程执行环境模式：流程执行环境接受并使用外部环境主动传递给它的数据。
- ❑ 环境从流程执行环境拉数据模式：流程执行环境接受外部环境的数据请求，并传递数据给外部环境。



图C-24　工作流数据交互模式

活动到活动（WDP_9: Task to Task）

描述

在一个流程实例里，活动实例能够给后续的活动实例传递数据。

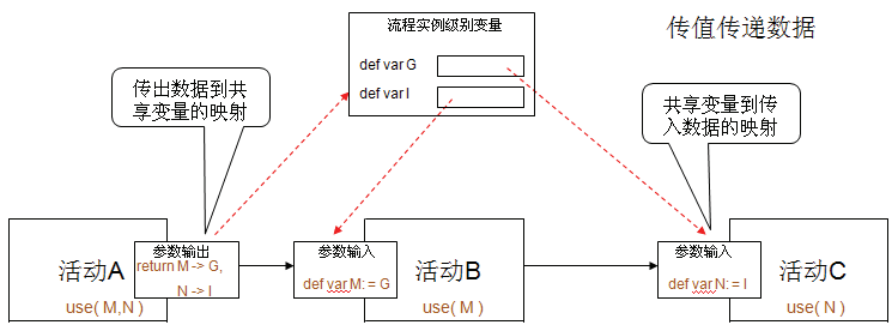
图C-25　活动到活动的数据交互

应用

后续活动需要使用前续活动所产生的数据。例如，当前活动参与者指定后续活动的参与者、紧急程度和期望完成时间。

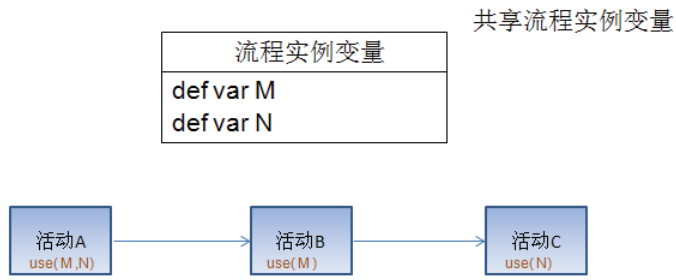
活动之间的数据交互属于工作流系统里最基本的功能，存在两种主要的实现方式：传值和共享数据。

传值：每个活动都有自己的活动变量，使用流程实例变量进行活动变量之间的传值，如图C-26所示。



图C-26　通过传值在活动间传递数据

共享数据：活动没有自己的活动变量，直接使用流程实例变量，避免数据的显式传递。这种实现是最简单也是最常用的方式，但是当流程实例中存在两个或以上同时执行的活动实例时，会存在数据并发操作的问题，如图C-27所示。

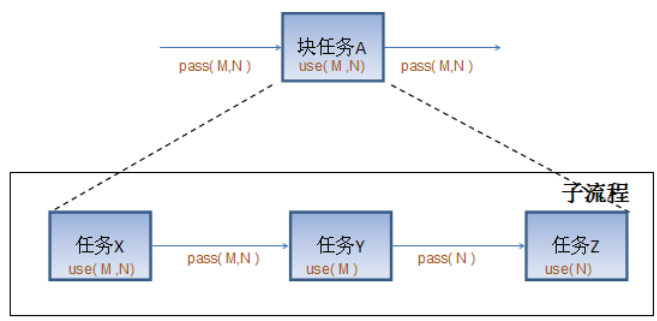


图C-27　共享流程实例变量

块活动到子流程（WDP_10: Block Task to Sub-Workflow Decomposition）

描述

在一个流程实例里，块活动实例能够给与之对应的子流程实例传递数据。



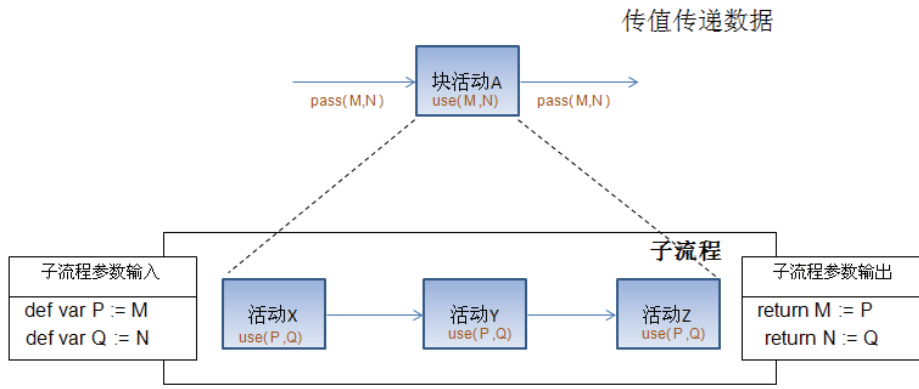
图C-28　块活动到子流程的数据传递

应用

作为独立建模的流程模型，子流程在多个流程里复用。因为任何复用复用的都是行为而不是数据，所以子流程实例的执行上下文需要父流程实例传递业务数据或者业务关联数据进行初始化。

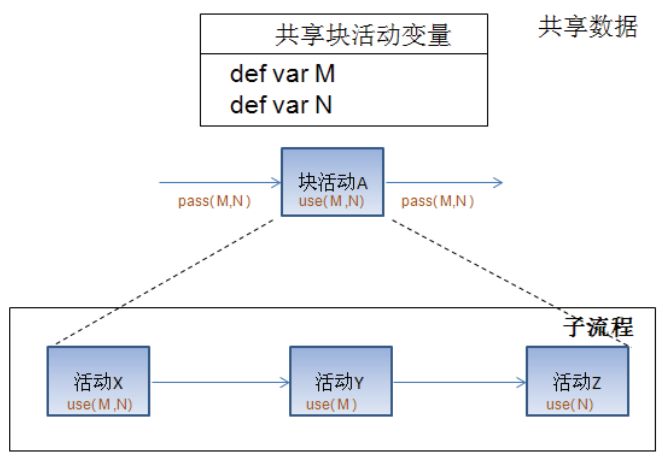
和活动间的数据交互一样，存在两种主要的实现方式：传值和共享数据。

传值：我们在块活动定义里进行数据映射，将块活动中的变量与子流程中的变量进行一一映射，在运行期，父流程实例会将相应的变量传值到子流程中的对应变量中，子流程实例执行完毕再将值传回，如图C-29所示。



图C-29　通过传值给子流程传递数据

共享数据：子流程实例直接使用块活动变量，避免数据的显式传递，如图C-30所示。



图C-30　共享块活动变量

子流程到块活动（WDP_11: Sub-Workflow Decomposition to Block Task）

描述

在一个流程实例里，子流程实例能够给与之对应的块活动实例传递数据。

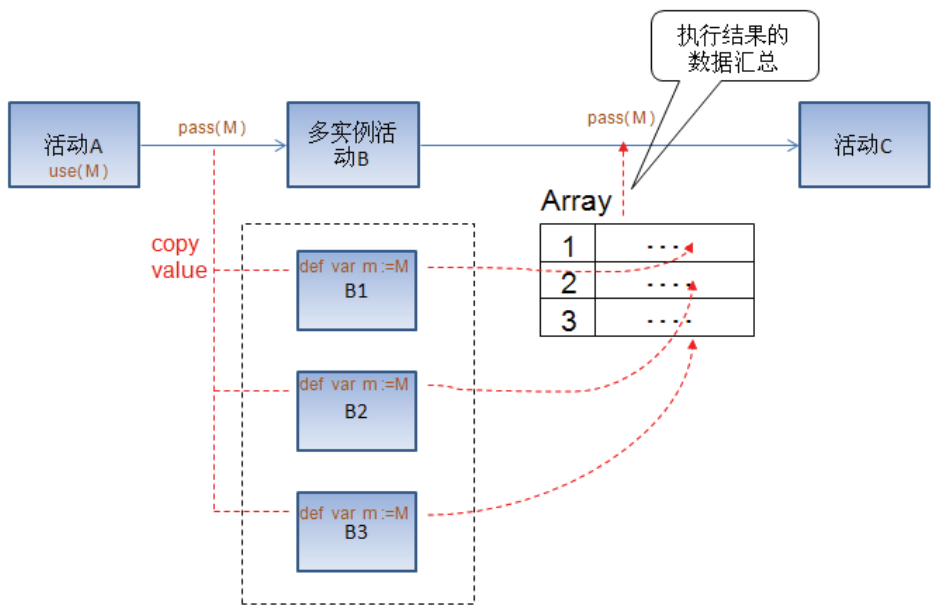
应用

子流程实例执行结束后给调用其的块活动实例返回执行结果。

传递数据到多实例活动（WDP_12: to Multiple Instance Task）

描述

在一个流程实例里，前续活动实例能够给后续多实例活动的每一个工作项传递数据。



图C-31　前续活动到多实例活动的数据传递

从多实例活动传递数据（WDP_13: from Multiple Instance Task）

描述

在一个流程实例里，多实例活动实例能够给后续活动实例传递数据。

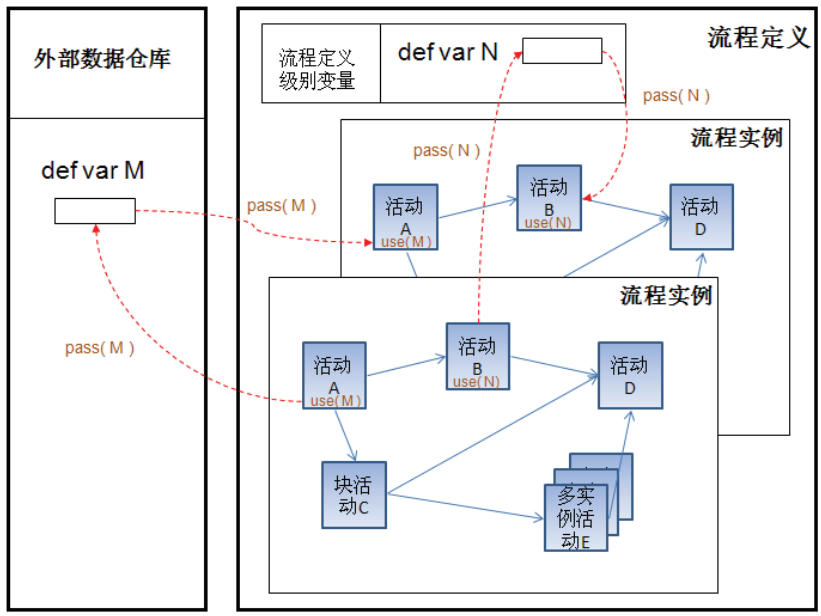
应用

多实例活动的各个工作项汇总执行结果并影响后续活动实例的执行。

流程实例到流程实例（WDP_14: Case to Case）

描述

流程实例在执行过程中能够给其他正在执行的流程实例传递数据。

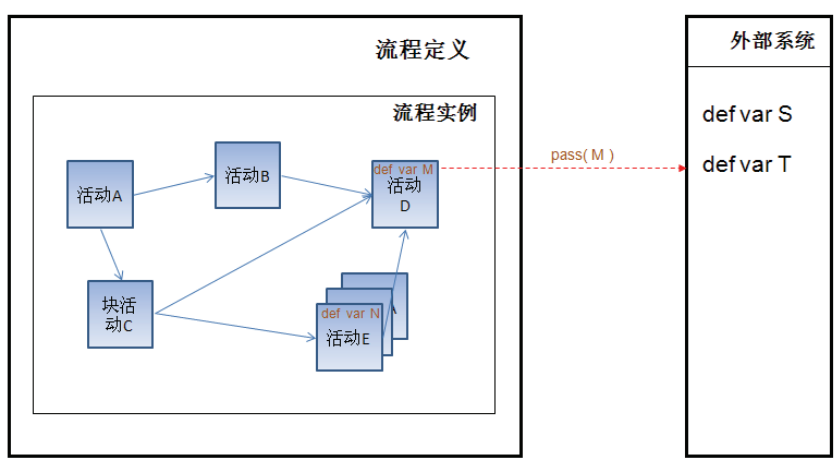


图C-32　流程实例之间的数据交互

活动推数据到环境（WDP_15: Task to Environment - Push-Oriented）

描述

活动实例能够给外部环境传递数据，外部环境包括了外部数据仓库（数据库）、外部应用和外部服务等。



图C-33　活动实例推数据到外部环境

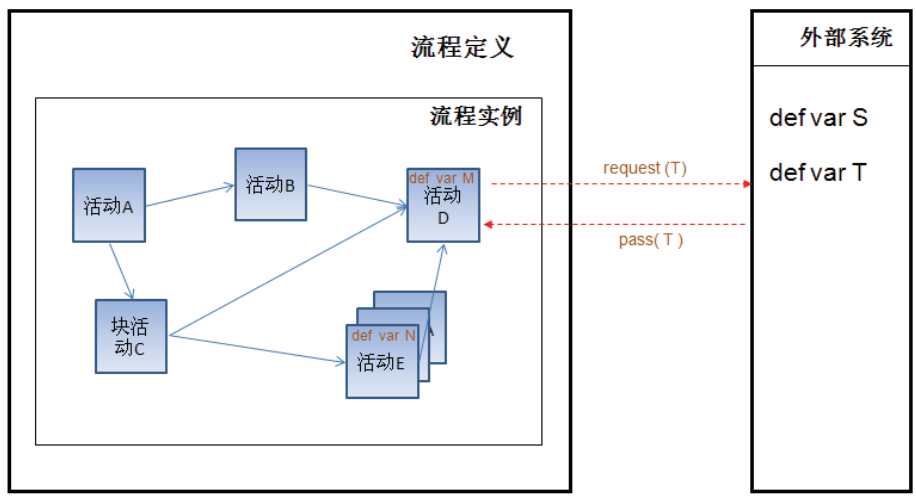
应用

当一个流程实例跨越多个业务系统时，业务系统之间的数据通过工作流系统完成传递。

活动从环境拉数据（WDP_16: Environment to Task – Pull-Oriented）

描述

活动实例能够从外部环境获取数据。

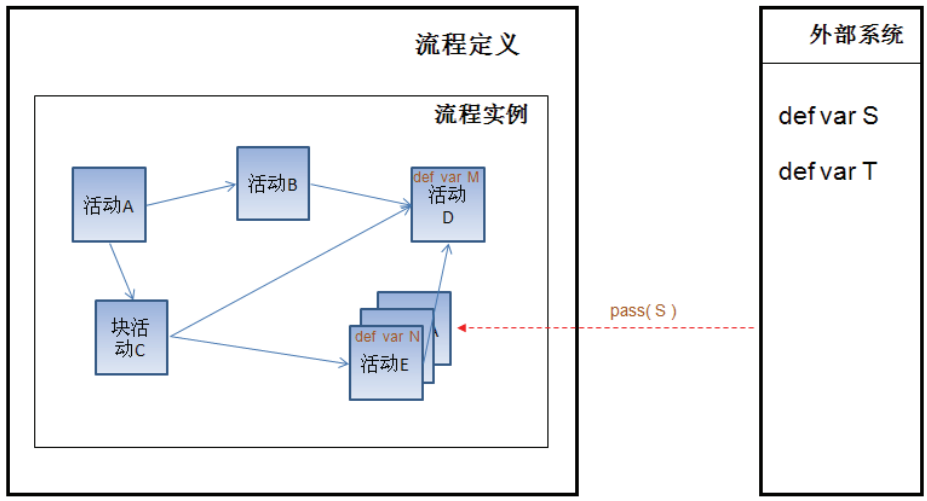


图C-34　活动实例从外部环境拉数据

环境推数据到活动（WDP_17: Environment to Task – Push-Oriented）

描述

活动实例能够接受并使用外部环境主动传递给它的数据。

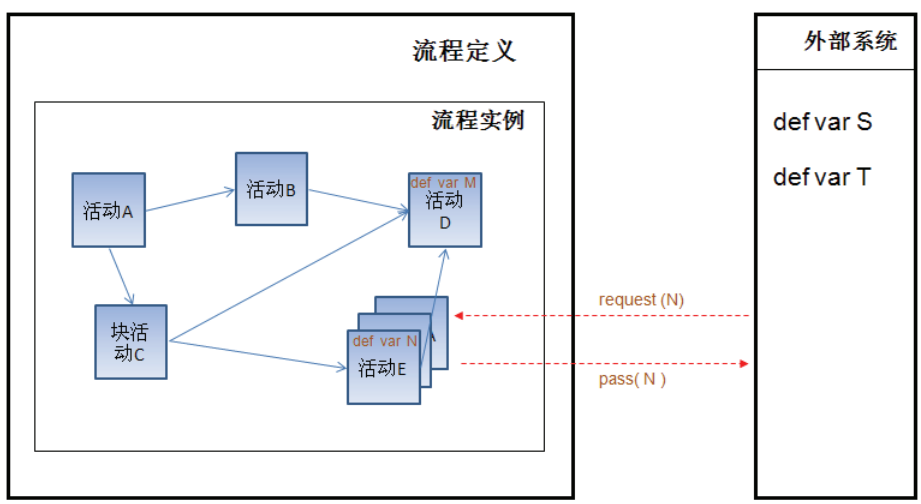


图C-35　外部环境推数据到活动实例

环境从活动拉数据（WDP_18: Task to Environment - Pull-Oriented）

描述

活动实例能够接受外部环境的数据请求，并传递数据给外部环境。

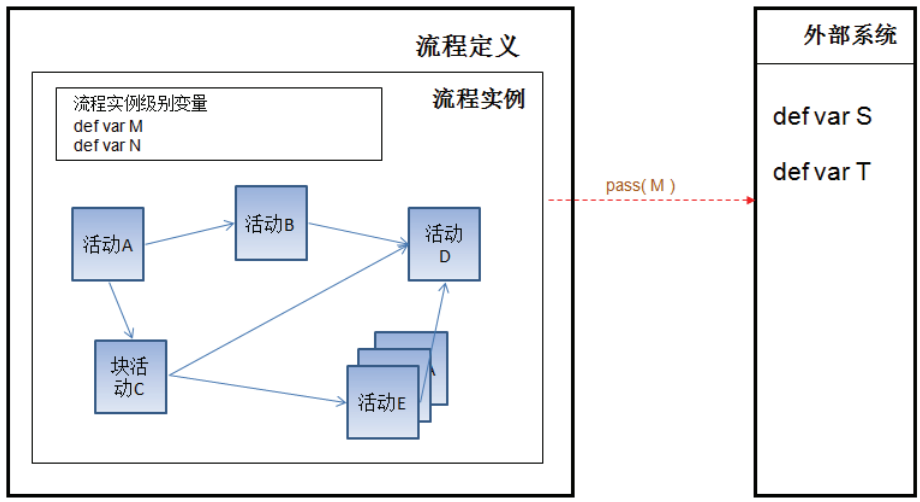


图C-36　外部环境从活动实例拉数据

流程实例推数据到环境（WDP_19: Case to Environment - Push-Oriented）

描述

流程实例能够给外部环境传递数据。



图C-37　流程实例推数据到外部环境

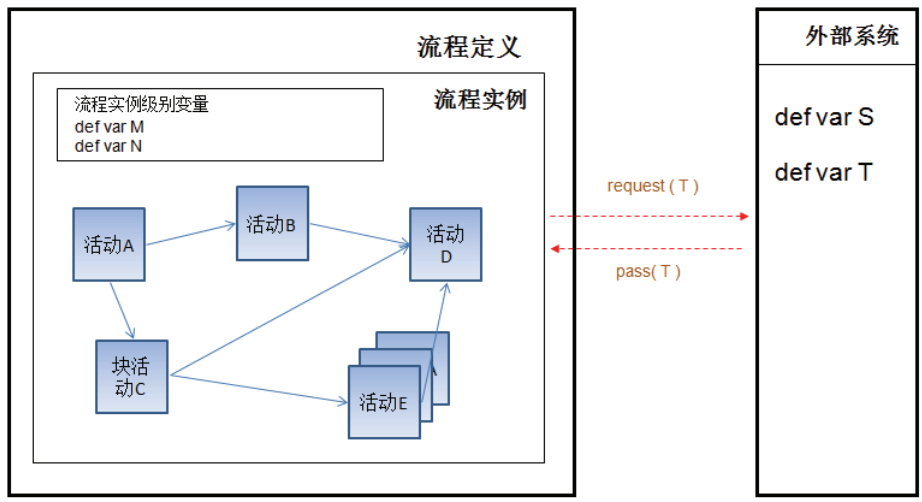
应用

在对报销流程进行分析时，我们发现大部分的报销金额都低于500元，然而这些报销流程却都要经过很多环节，在与客户确认后，我们将低于500元的报销限定于部门内部审批即可，由此整个报销过程大大加快，同时对公司省下很多办公成本。

流程实例从环境拉数据（WDP_20: Environment to Case - Pull-Oriented）

描述

流程实例能够从外部环境获取数据。

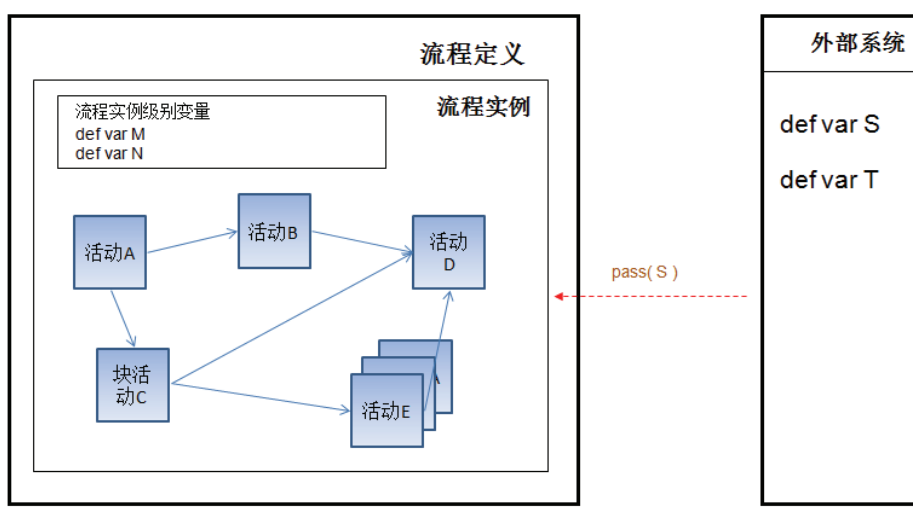


图C-38　流程实例从外部环境拉数据

环境推数据到流程实例（WDP_21: Environment to Case - Push-Oriented）

描述

流程实例能够接受并使用外部环境主动传递给它的数据。如中间消息事件。

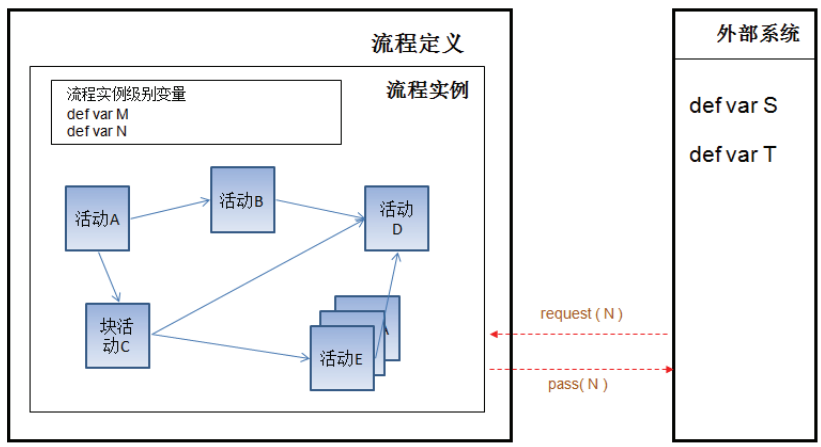


图C-39　外部环境推数据到流程实例

环境从流程实例拉数据（WDP_22: Case to Environment - Pull-Oriented）

描述

流程实例能够接受外部环境的数据请求，并传递数据给外部环境。

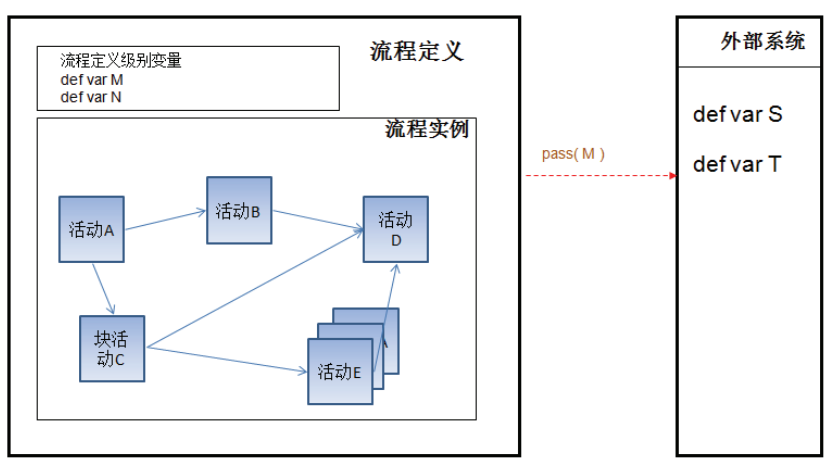


图C-40　外部环境从流程实例拉数据

流程执行环境推数据到环境（WDP_23: Workflow to Environment - Push-Oriented）

描述

流程执行环境能够给外部环境传递数据。

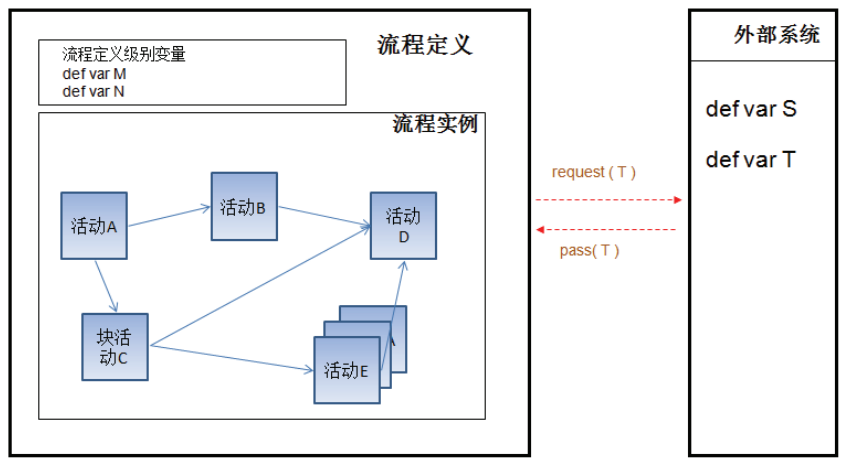


图C-41　流程执行环境推数据到外部环境

流程执行环境从环境拉数据（WDP_24: Environment to Workflow - Pull-Oriented）

描述

流程执行环境能够从外部环境获取数据。

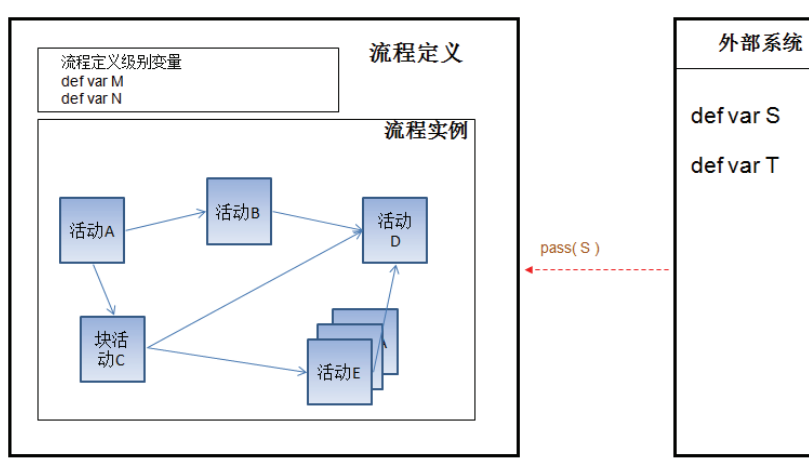


图C-42　流程执行环境从外部环境拉数据

环境推数据到流程执行环境（WDP_25: Environment to Workflow - Push-Oriented）

描述

流程执行环境能够接受并使用外部环境主动传递给它的数据。

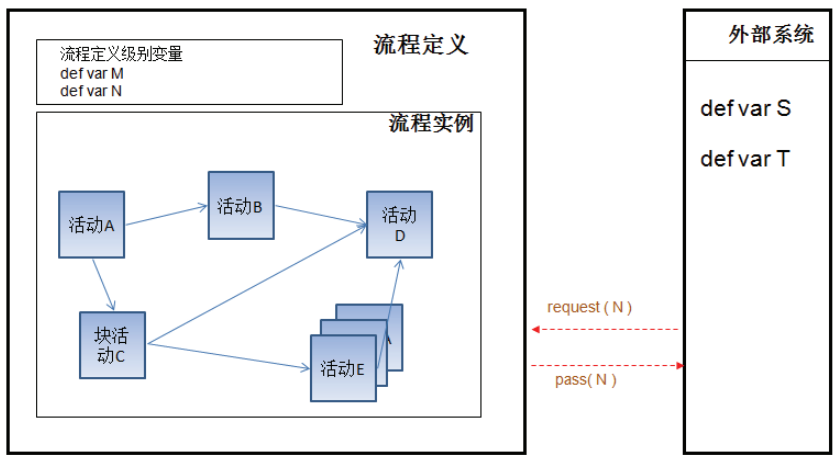


图C-43　外部环境推数据到流程执行环境

环境从流程执行环境拉数据（WDP_26: Workflow to Environment - Pull-Oriented）

描述

流程执行环境能够接受外部环境的数据请求，并传递数据给外部环境。

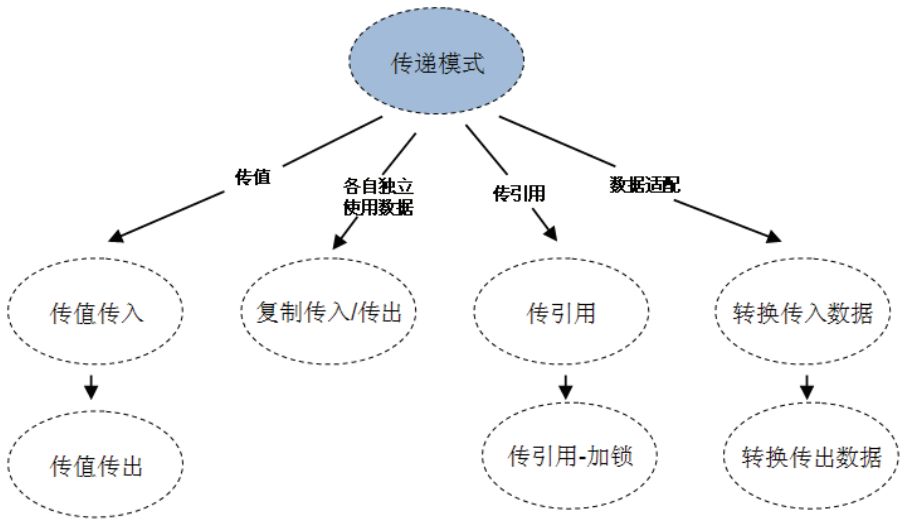


图C-44　外部环境从流程执行环境拉数据

传递模式

数据传递模式共有7种，讨论数据在工作流系统各组件以及外部环境之间的传递方式。

- 通过传值传入数据：当流程组件之间不共享一个通用的数据存储仓库时，我们通过传值传入数据。
- 通过传值传出数据：当流程组件之间不共享一个通用的数据存储仓库时，我们通过传值传出数据。
- 复制传入/传回数据：为了使流程组件能够各自独立的使用数据，我们通过复制传入/传回数据传递数据。
- 通过传引用传递数据——不加锁：当流程组件共享一个通用的数据存储仓库时，我们通过传引用直接使用数据。
- 通过传引用传递数据——加锁：当通过传引用传递数据时，我们通过加锁来避免并发的读写冲突。
- 转换传入数据：当传递的数据与目标数据类型不一致时，我们应用数据转换函数对传入数据进行转换。
- 转换传出数据：当传递的数据与目标数据类型不一致时，我们应用数据转换函数对传出数据进行转换。



图C-45　工作流数据传递模式

通过传值传入数据（WDP_27: Data Transfer by Value - Incoming）

描述

流程组件能够通过传值接受传入数据。

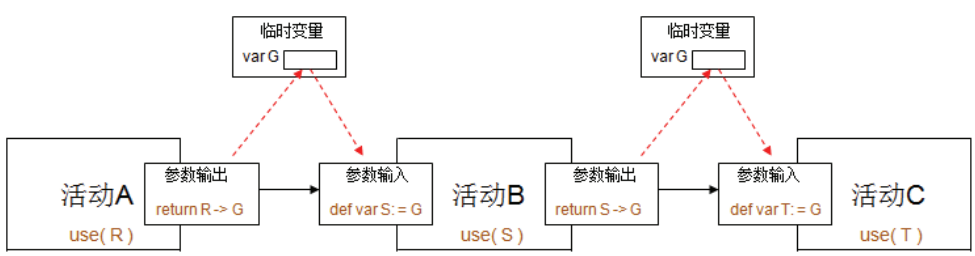
应用

在流程组件不共享一个通用的数据存储仓库时传递数据。

通过传值传出数据（WDP_28: Data Transfer by Value - Outgoing）

描述

流程组件能够通过传值传出数据，如图C-46所示。

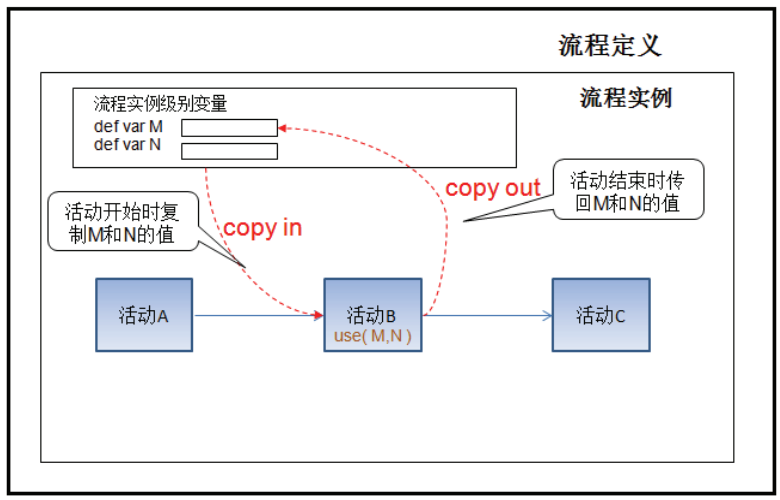


图C-46　通过传值传递数据

复制传入/传回数据（WDP_29: Data Transfer – Copy In/Copy Out）

描述

流程组件能够在开始执行时从外部资源（包括工作流系统内部的数据，也包括外部环境的数据）复制数据，并在执行结束时将数据传回。



图C-47　复制传入、传回数据

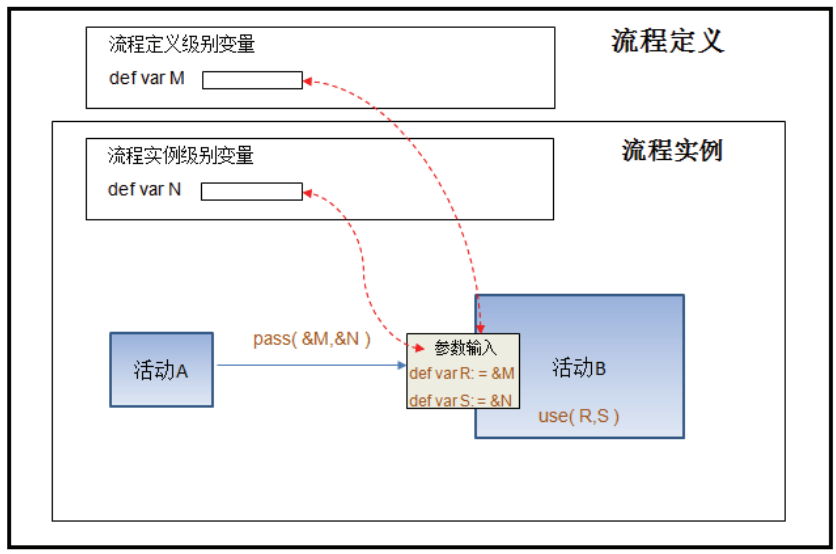
应用

父子流程实例之间的数据传递以及给多实例活动传递数据。

通过传引用传递数据——不加锁（WDP_30: Data Transfer by Reference - Unlocked）

描述

流程组件之间能够通过传递数据的引用实现数据传递，该数据处于这些流程组件都能访问的位置。对数据没有并发读写控制。

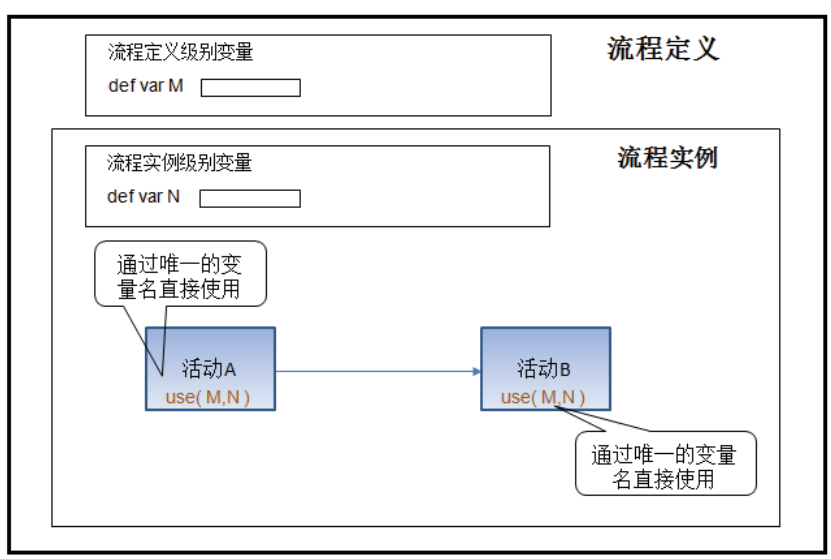


图C-48　通过传引用传递数据

应用

当流程组件共享一个通用的数据存储仓库时，不显式的传递数据。

该模式是应用最为广泛的数据传递模式，因为它足够简单。如图C-49所示，通过唯一的变量名直接使用数据。



图C-49　通过唯一变量名直接使用数据

该模式最大的问题正如它的名字：不加锁。因为数据没有并发的读写控制，所以如果存在多个执行中的活动实例同时修改同一数据或者读写同时发生时，就会产生冲突。

通过传引用传递数据——加锁（WDP_31: Data Transfer by Reference – With Lock）

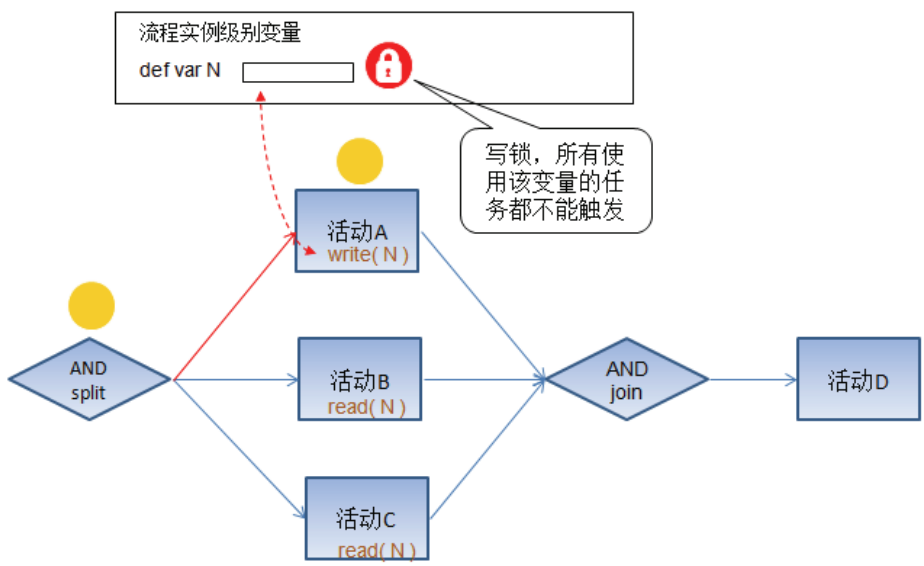
描述

流程组件之间能够通过传递数据的引用实现数据传递，该数据处于这些流程组件都能访问的位置。为了避免读写冲突，通过读写锁对这些数据进行并发读写控制。

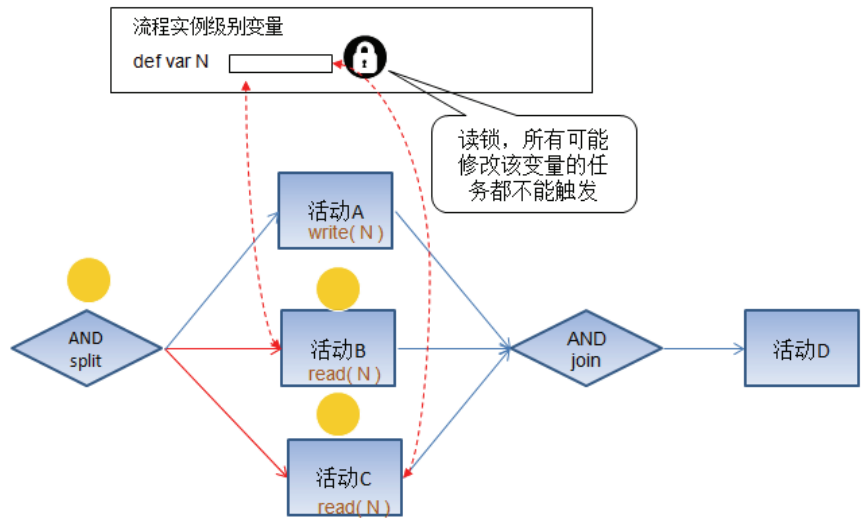
应用

当通过传引用传递数据时，避免并发的数据读写冲突。

活动实例在使用数据时，我们对变量进行加锁。如果是只读，我们加读锁；如果是写，我们加写锁。我们在进行活动定义时在活动节点上声明活动对该变量是只读还是写。



图C-50　使用写锁控制活动的触发

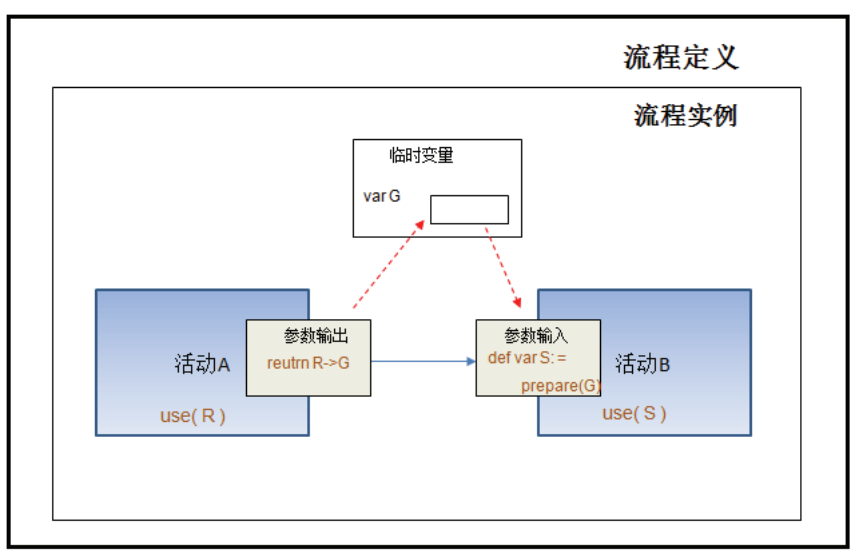


图C-51　使用读锁控制活动的触发

转换传入数据（WDP_32: Data Transformation - Input）

描述

在将数据传递给流程组件前能够应用转换函数对数据进行转换。



图C-52　转换传入数据

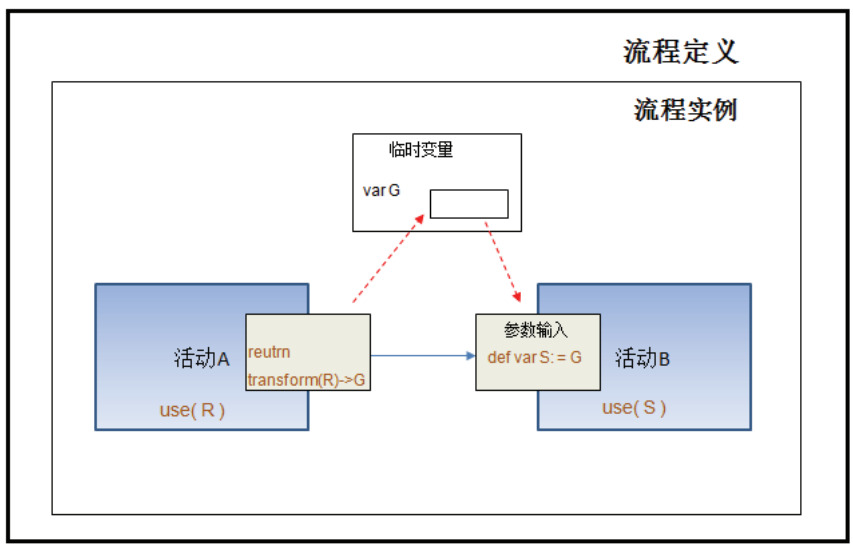
应用

当传递的数据与目标数据类型不一致时，对数据进行转换。

转换传出数据（WDP_33: Data Transformation - Output）

描述

在将数据从流程组件传出前能够应用转换函数对数据进行转换。



图C-53　转换传出数据

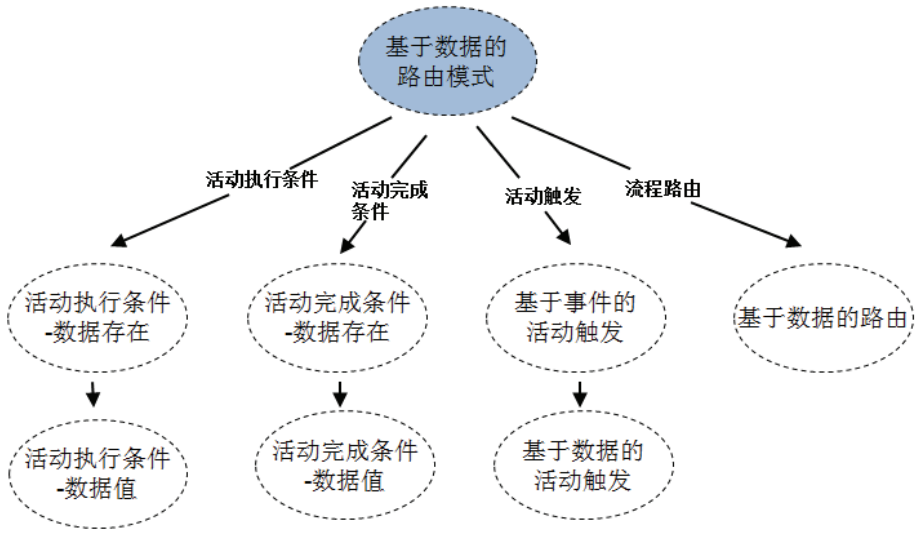
应用

当传递的数据与目标数据类型不一致时，对数据进行转换。

基于数据的路由模式

基于数据的路由模式共有7种，讨论数据对流程实例执行所产生的影响。

- ❑ 活动执行条件——数据存在：根据数据是否可用决定活动是否可以执行。
- ❑ 活动执行条件——数据值：根据数据是否等于指定值决定活动是否可以执行。
- ❑ 活动完成条件——数据存在：根据数据是否可用决定活动是否能够执行完成。
- ❑ 活动完成条件——数据值：根据数据是否等于指定值决定活动是否能够执行完成。
- ❑ 基于事件的活动触发：外部事件能够触发活动的执行并传递数据。
- ❑ 基于数据的活动触发：通过数据反映的流程实例状态能够触发活动的执行。
- ❑ 基于数据的路由：数据能够决定流程的路由。

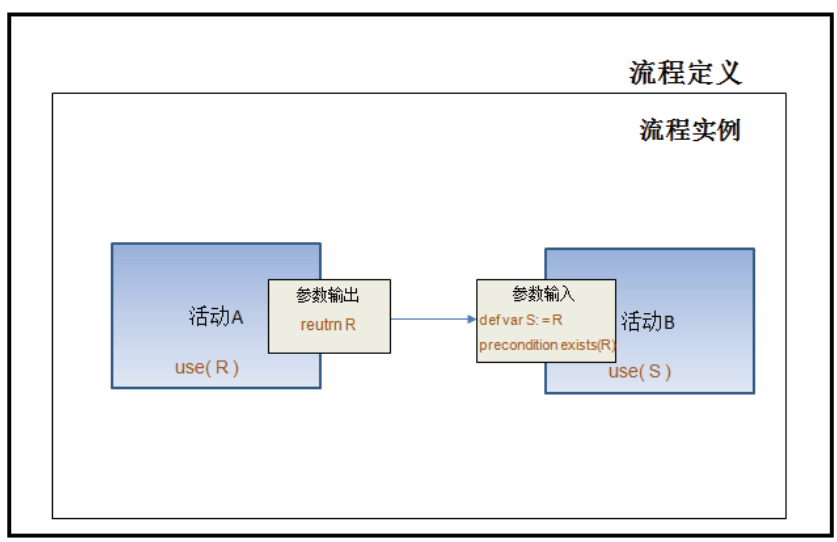


图C-54　基于数据的路由模式

活动执行条件——数据存在（WDP_34: Task Precondition – Data Existence）

描述

数据是否可用能够作为活动的执行条件。只有满足执行条件活动才可执行。



图C-55　活动执行条件——数据存在

应用

在执行活动时，设定活动执行的前提条件，只有满足条件时才能执行该活动。这个条件反映出活动对前续活动执行结果、流程实例状态的依赖。当无法满足活动的执行条件时，有5种行为可以选择：推迟活动的执行直至满足条件、为活动设置不满足条件时的默认行为、跳过该活动的执行、与该活动参与者交互由参与者做出决定和终止当前流程实例的执行。

活动执行条件——数据值（WDP_35: Task Precondition – Data Value）

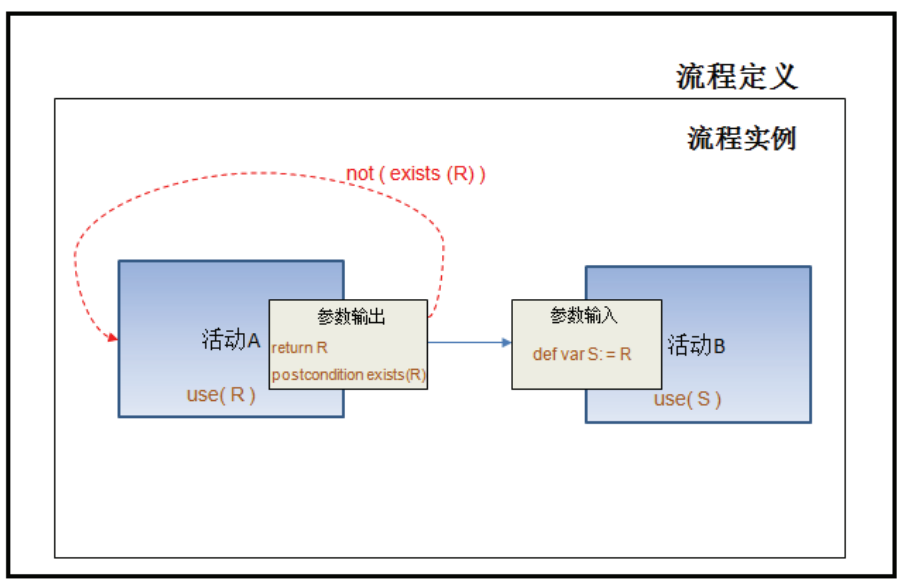
描述

数据是否等于指定值能够作为活动的执行条件。只有满足执行条件的活动才可执行。

活动完成条件——数据存在（WDP_36: Task Postcondition – Data Existence）

描述

数据是否可用能够作为活动的完成条件。只有满足完成条件的活动才表明执行完毕。



图C-56　活动完成条件——数据存在

应用

在执行活动时，设定活动执行的完成条件，只有满足条件时该活动才算完成。这个条件反映出对活动执行结果的期望。当无法满足活动的完成条件时，有2种行为可以选择：重新/继续执行该活动直至满足条件为止；将该活动标识为未完成挂起并继续执行后续活动。

活动完成条件——数据值（WDP_37: Task Postcondition – Data Value）

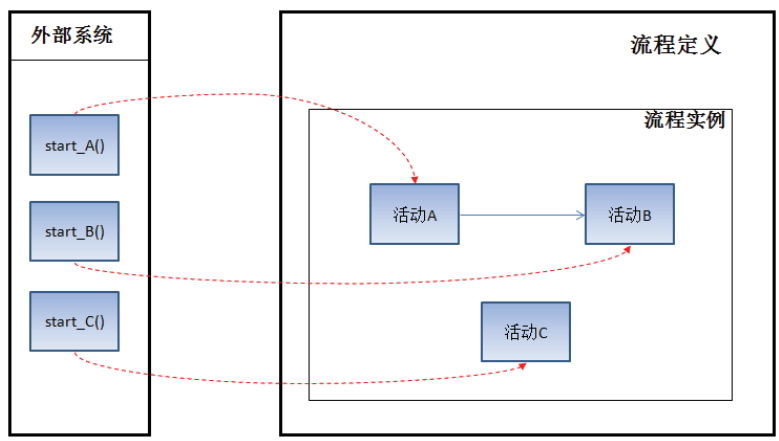
描述

数据是否等于指定值能够作为活动的完成条件。只有满足完成条件的活动才表明执行完毕。

基于事件的活动触发（WDP_38: Event-based Task Trigger）

描述

外部事件能够触发活动实例的执行并给活动实例传递数据。



图C-57　基于事件的活动触发

应用

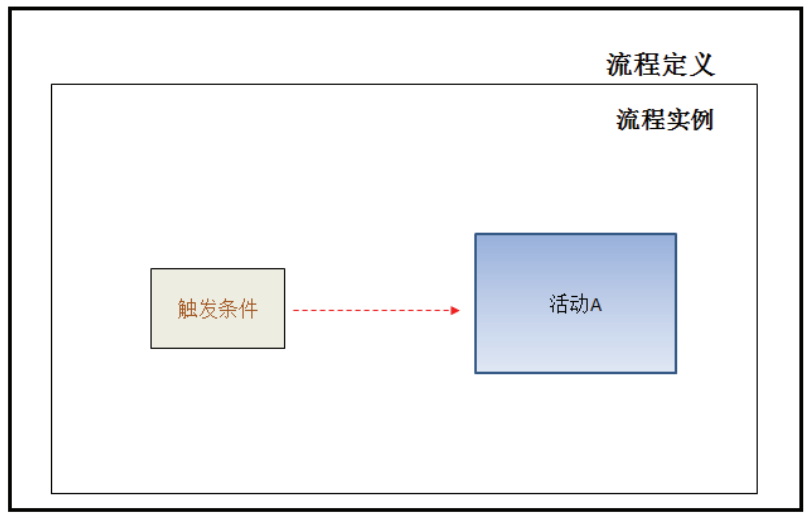
流程实例的执行能够响应外部环境发生的变化，并对变化作出反应。

该模式对应于控制模式里的触发模式（WCP_23和WCP_24）。存在3种应用场景，如图C-57所示。触发流程的第一个活动（活动A），其等价于触发一个新的流程实例；触发流程实例中一个等待中的活动（活动B），若活动不处于等待状态那么有两种情况：事件丢失（瞬态触发）、事件持久化等待消费（持久化触发）；触发流程中与主流程隔离的单独活动（活动C），设计该活动的目的是消费特定的外部事件。

基于数据的活动触发（WDP_39: Data-based Task Trigger）

描述

数据能够作为活动的触发条件。我们为活动指定基于数据的表达式，当该表达式求值为真时即触发活动的执行。

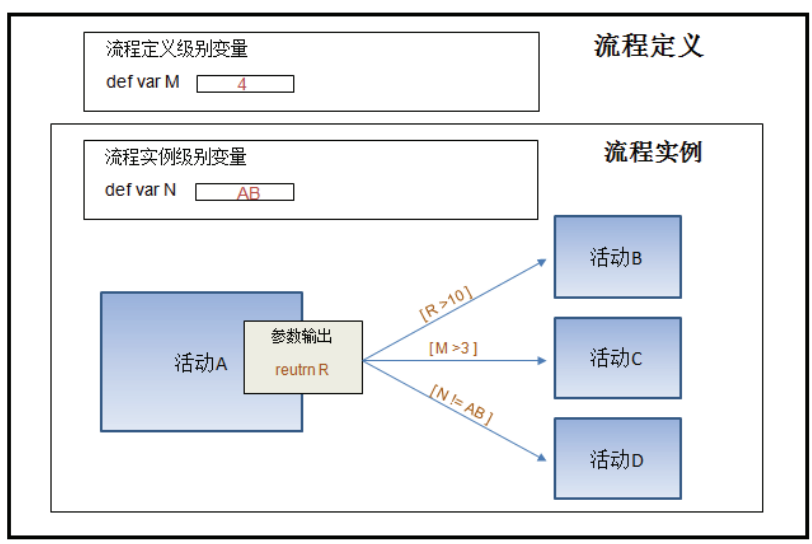


图C-58　基于数据的活动触发

基于数据的路由（WDP_40: Data-based Routing）

描述

数据能够影响流程的路由。我们为XOR-split或OR-split的后续分支指定基于数据的路由表达式，当表达式求值为真时即触发对应分支的路由。



图C-59　基于数据的路由

附录D
工作流异常处理模式

在软件开发里，我们将不在自己控制范围内因素所造成的问题和没有预料到的情况称为异常。工作流异常和软件开发里异常的概念一致，将流程实例执行过程中出现的问题和错误称为异常，这些异常是由各种不确定因素造成的，从而使流程实例执行偏离了流程设计者最初的期望。引起工作流异常的因素有很多，流程定义描述的不准确或不完整，执行环境的变化，不能获取资源等都会引起流程执行偏离预期。这些因素涉及系统异常：硬件、软件、通讯、工作流模型、相关应用程序、流程逻辑约束、工作流相关数据约束、时间约束以及执行算法；涉及业务异常：工作人员请假离职、资源紧张、突发事件、用户中止合同、项目目标发生变化等。

在Java和C++里，当程序运行过程中出现异常时，会中止当前程序的运行，并将异常层层抛出。工作流系统的异常处理与之相同，如果在当前的运行上下文里，我们得不到足够的信息来处理这个问题，我们就会停止运行，并将这个问题交给上层进行处理，上层拥有更多的信息。在软件开发里，异常处理的一个原则是：如果不知道该如何处理这个异常，那么就别去捕捉它。这个原则同样适用于流程实例的执行过程，如果不知道如何处理一个问题，那么就不要处理，把它留给能处理这个问题的人去处理。

程序里，异常处理的目标在于让我们能用比现在更少的代码，以一种更简单的方式来开发大型、可靠的程序。通过将处理异常的代码从异常发生的地方移开，我们能够在一个地方集中精力去解决想解决的问题，然后再到另一个地方集中处理这些异常问题。程序的主线不会被异常处理这类枝节问题给搞得支离破碎，程序也更易于理解和维护。这一目标同样也是工作流系统异常处理的目标，当应用工作流系统时，我们首先会进行流程建模，复杂的流程定义会导致流程的不可管理，作为一个原则，流程定义必须能够被执行工作的人所理解。当对实际业务流程进行建模时，如果只建模流程执行的乐观路径，模型会直接易懂，但如果将流程执行过程中的各种异常情况和

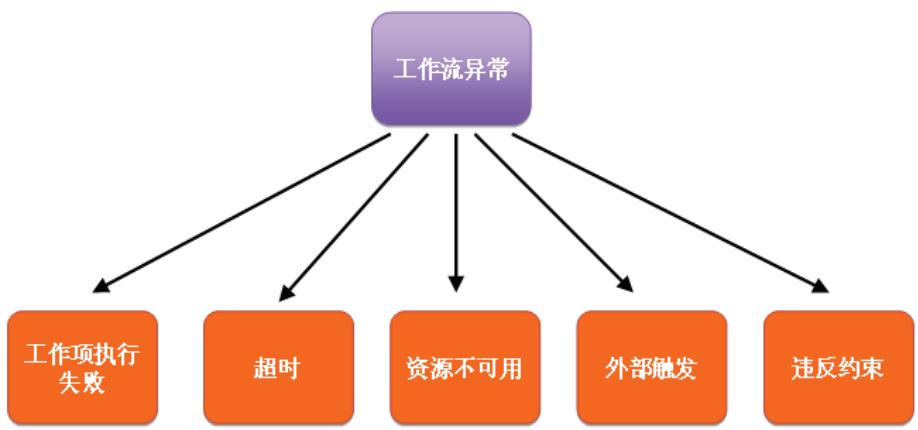
如何处理都进行建模，那么肯定会带来巨大的复杂性，使得流程定义非常难以维护。此时一是需要工作流系统提供异常处理机制；二是需要应用人员进行适当的异常分类与建模，并在合适的位置设置异常处理器。

在前面的章节里，我们分别讨论了工作流的控制模式、资源模式和数据模式，这三种模式分别对工作流的不同方面进行了描述，异常处理模式与这些模式都紧密关联，因为在工作流的执行中，流程定义、流转、资源分配与数据约束都会引起异常。

在本章里，我们首先对工作流执行过程中可能出现的异常进行分类，一般来说，异常都会与一个正在执行的工作项关联，即当前正在执行的工作产生了异常，但外部环境的变化也会影响到当前正在执行的工作。接下来我们将讨论在不同的层次上对异常进行处理，这些层次包括工作项级别和流程实例级别（包括了流程定义级别），以及异常发生时我们可以采取的恢复动作。当讨论完这些点之后，我们将会看到它们如何互相结合构成我们的工作流异常处理模式即面。

异常分类

流程执行过程中的异常分为五类，如图D-1所示。



图D-1　工作流异常分类

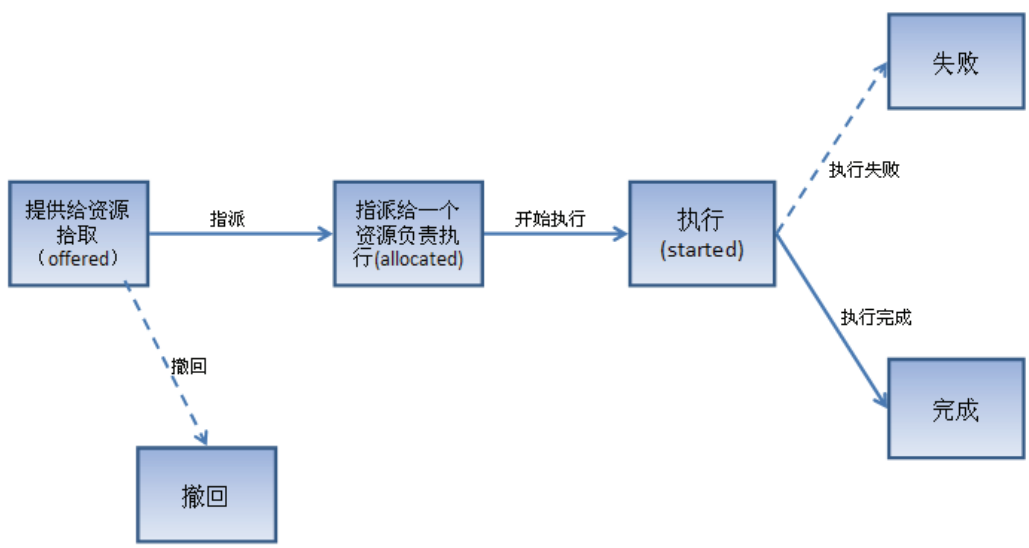
- 工作项执行失败：工作项所代表的工作不能继续执行或无法按照期望完成。导致工作项执行失败的原因有很多种，与该工作项相关的硬件故障、软件故障或网络故障都会导致活动参与者无法正常执行该工作项，同时活动参与者也可能会自己中止该工作项的执行或者直接放弃。
- 超时：工作项未在指定的时间点完成或未在指定的时间点开始执行。
- 资源不可用：没有可用的资源执行或完成工作项。有两种情况，一是分配工作项时系统找不到满足执行该工作条件的资源（人手不够，资源被占用），二是工作项执行过程中，先前指定的资源不能继续或无法执行该工作项（生病、离职、调动）。

- 外部触发：外部触发通常表现为事件，组织外部的事件影响正在执行中的工作项。例如用户突然取消订单会导致订单处理流程中所有工作项的中止，并伴随着业务回滚（收回发货）。
- 违反约束：工作流约束包括流程流转的约束（流程死锁，不能继续执行，进入死循环）、数据的约束、资源的约束（超出当前组织资源所能达到的能力）以及业务约束。工作流系统运行过程中需要保证流程执行的合理性和一致性，需要对流程执行状态进行持续的监控。

工作项级别的异常处理

一般来说，异常都会与一个正在执行的工作项关联，即当前正在执行的工作产生了异常，外部触发是个例外，此时异常并不由流程自身执行产生，但是影响到当前正在执行的工作项。当发生异常时，工作项级别有很多种可能的处理方式，这些方式与当前工作项状态息息相关，所以在讨论具体的处理方式之前，我们先简单回顾一下工作项的生命周期，然后讨论工作项在其生命周期的不同状态时可以选择的异常处理方式。

从资源的角度，工作项的生命周期具有6个状态（相比资源模式里的讨论，进行了简化），分别是：提供给资源拾取（Offered）、指派给一个资源执行（Allocated）、执行（Started）、完成（Completed）、失败（Failed）和撤回（Withdrawn）。如图D-2所示。



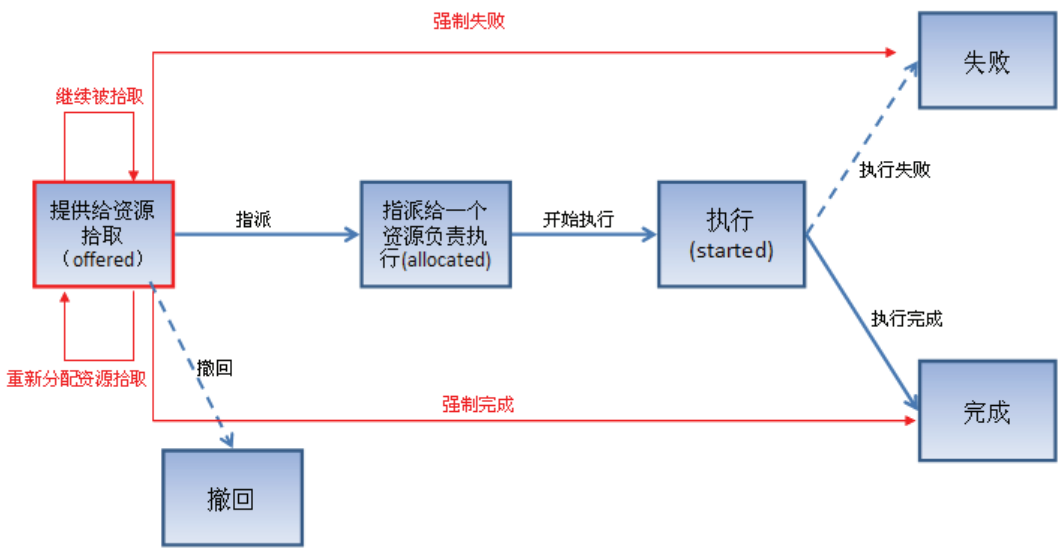
图D-2　工作项的生命周期

最开始，工作流系统创建工作项并将其提供给资源拾取，这里的资源可以是单个资源也可以是多个资源，工作项此时处于提供给资源拾取状态。接下来，其中一个资源向工作流系统发送一

个要求指派的请求，要求系统将该工作项指派给他进行执行，然后系统就会将工作项指派给该资源执行，工作项此时处于指派给一个资源执行的状态，同时，对于前一步工作项被提供的其他资源来说，工作项被撤回了。接下来，资源向系统发送开始执行的请求，工作项即进入执行状态。最后，如果资源正常完成活动，会给系统发送请求，告诉活动完成，系统将工作项的状态置为完成，工作项的生命周期即到此结束；而如果资源没有按照期望完成活动或无法完成活动，工作项的状态置为失败，工作项的生命周期结束。

当工作项处于提供给资源拾取的被拾取状态时，可能的异常处理方式包括以下4种。

- 继续被拾取（OCO，continue-offer）：工作项继续保持被拾取的状态，不发生任何变化。
- 重新分配资源拾取（ORO，re-offer）：工作项重新分配给新的资源进行拾取。
- 强制完成（OFC，force-complete）：工作项从资源的待拾取列表撤回，状态变为完成，触发后续活动产生新的工作项。
- 强制失败（OFF，force-fail）：工作项从资源的待拾取列表撤回，状态变为失败，不触发后续活动。

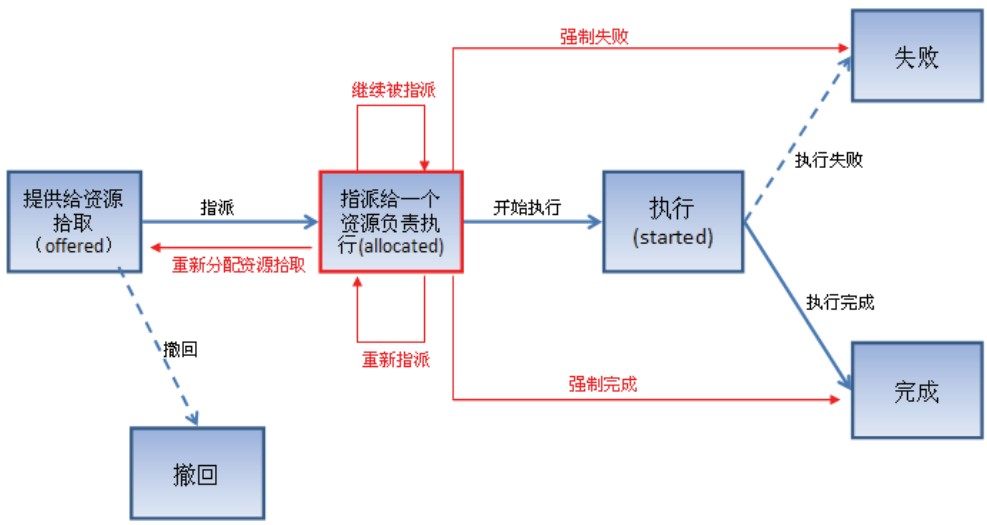


图D-3　工作项处于被拾取状态的异常处理

当工作项处于指派给一个资源执行的被指派状态时，可能的异常处理方式包括以下5种。

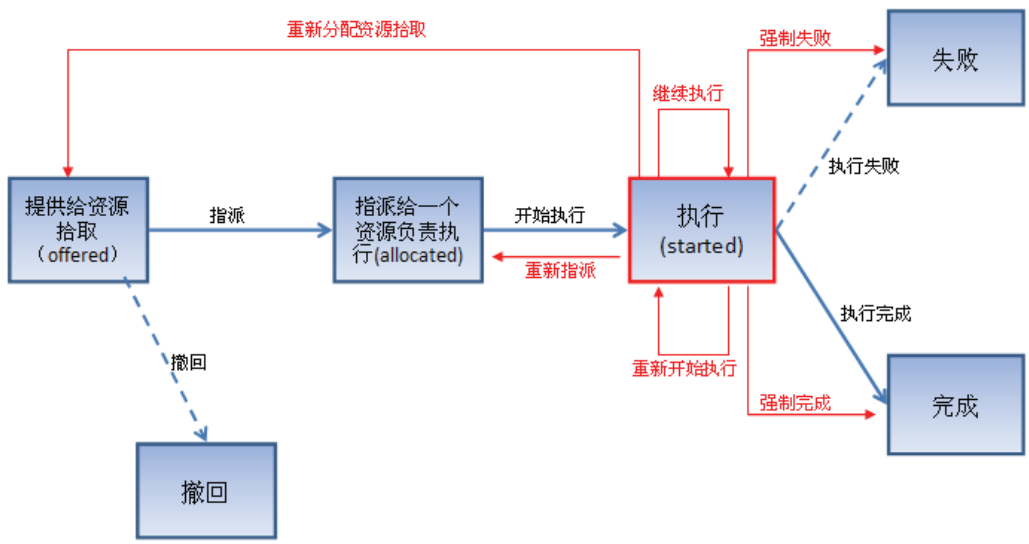
- 继续被指派（ACA，continue-allocation）：工作项继续保持被指派的状态，不发生任何变化。
- 重新指派（ARA，re-allocation）：工作项从资源的待办列表撤回，工作项重新分配给新的资源执行。
- 重新分配资源拾取（ARO，re-offer）：工作项从资源的待办列表撤回，工作项重新分配给新的资源进行拾取。

- 强制完成（AFC，force-complete）：工作项从资源的待办列表撤回，状态变为完成，触发后续活动产生新的工作项。
- 强制失败（AFF，force-fail）：工作项从资源的待办列表撤回，状态变为失败，不触发后续活动。



图D-4　工作项处于被指派状态的异常处理

当工作项处于执行状态时，可能的异常处理方式包括以下6种。



图D-5　工作项处于执行状态的异常处理

- 继续执行（SCE，continue-execution）：工作项继续保持执行的状态，不发生任何变化。
- 重新开始执行（SRS，re-start）：当前正在执行的工作中止，由同一个资源重新执行该工作项。
- 重新指派（SRA，re-allocation）：当前正在执行的工作中止，工作项从资源的办理列表撤回，工作项重新分配给新的资源执行。
- 重新分配资源拾取（SRO，re-offer）：当前正在执行的工作中止，工作项从资源的办理列表撤回，工作项重新分配给新的资源进行拾取。
- 强制完成（SFC，force-complete）：当前正在执行的工作中止，工作项从资源的办理列表撤回，状态变为完成，触发后续活动产生新的工作项。
- 强制失败（SFF，force-fail）：当前正在执行的工作中止，工作项从资源的办理列表撤回，状态变为失败，不触发后续活动。

流程实例级别的异常处理

异常不仅会影响到与其关联的工作，往往还会影响到同一个流程实例里正在执行的其他工作甚至同一个流程定义里其他正在执行的工作，所以我们还需要在更高的一个级别进行处理。

可能的异常处理方式包括以下3种。

- 继续执行（CWC，continue with case）：当前流程实例中其他工作不受影响，流程实例继续执行。异常在工作项级别已经得到完全的处理。
- 当前流程实例中止执行（RCC，remove current case）：当前流程实例中的部分或所有工作中止执行，如果所有工作都中止执行，那么即流程实例中止执行。异常所影响的范围限于当前流程实例内。
- 所有属于同一流程定义的流程实例都中止执行（RAC，remove all cases）：属于同一流程定义的流程实例中的部分或所有工作中止执行，如果所有工作都中止执行，那么即所有流程实例都中止执行。异常影响所有属于同一流程定义的流程实例。

恢复动作

当异常发生时，执行过的工作已经产生了一定的影响，为了使流程实例能够继续执行或正常停止，必须对已执行工作所产生的影响进行消除,这是通过恢复动作完成的，该动作对流程中可补偿的工作进行补偿。

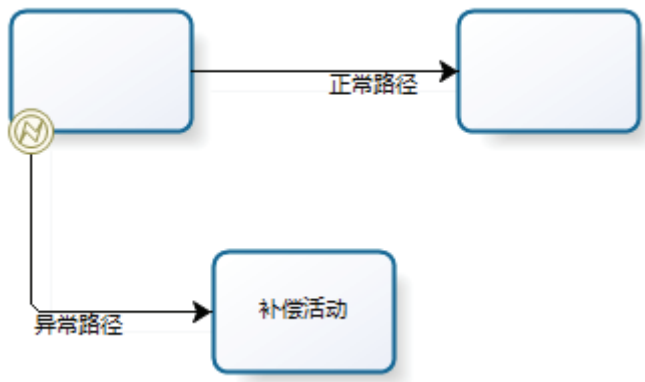
可能的恢复动作包括以下3种。

- 什么都不做（NIL，no action）：什么都不做。
- 回滚（RBK，rollback）：流程实例状态重置至异常发生前的一个回滚点。
- 补偿（COM，compensate）：额外的补偿动作，消除异常所产生的影响。

对于回滚，我们需要记录一个回滚点，通过执行日志记录流程实例执行过程中的事件，当回

滚动作发生时，我们将执行日志中回滚点后续的执行事件全部撤销。我们并不需要将流程实例回滚后的状态与流程实例当时位于回滚点的状态完全一致，我们需要的只是消除异常所产生的影响，所以执行日志不需要详细记录流程实例中的每一个执行事件。同时，有一些操作也是回滚动作无法恢复的，例如资源浪费的时间、已经发送的文档/报告等。

补偿动作相对回滚来说有很多的灵活性，我们不仅可以将发生过的事件撤回，还可以额外进行一些操作（例如通过异常路径继续执行流程实例），来消除异常所带来的影响。



图D-6　BPMN里的异常流属于补偿动作

异常处理模式

对每一种的异常而言，都可能存在不同的处理方式，我们把每种针对特定异常类型可能的处理方式称为异常处理模式。一个异常处理模式包括4方面的内容：

- 异常的类型；
- 发生异常的活动如何处理即工作项级别对异常的处理；
- 发生异常的流程实例以及关联流程实例如何处理即流程实例级别对异常的处理；
- 采取何种恢复动作。

共有135种可能的异常处理模式。

工作执行失败	工作超时	资源不可用	外部触发	违反约束
OFF-CWC-NIL	OCO-CWC-NIL	ORO-CWC-NIL	OCO-CWC-NIL	SCE-CWC-NIL
OFF-CWC-COM	ORO-CWC-NIL	OFF-CWC-NIL	OFF-CWC-NIL	SRS-CWC-NIL
OFC-CWC-NIL	OFF-CWC-NIL	OFF-RCC-NIL	OFF-RCC-NIL	SRS-CWC-COM
OFC-CWC-COM	OFF-RCC-NIL	OFC-CWC-NIL	OFC-CWC-NIL	SRS-CWC-RBK
AFF-CWC-NIL	OFC-CWC-NIL	ARO-CWC-NIL	ACA-CWC-NIL	SFF-CWC-NIL
AFF-CWC-COM	ACA-CWC-NIL	ARA-CWC-NIL	AFF-CWC-NIL	SFF-CWC-COM
AFC-CWC-NIL	ARA-CWC-NIL	AFF-CWC-NIL	AFF-RCC-NIL	SFF-CWC-RBK
AFC-CWC-COM	ARO-CWC-NIL	AFF-RCC-N1L	AFC-CWC-NIL	SFF-RCC-NIL
SRS-CWS-NIL	AFF-CWC-NIL	AFC-CWC-NIL	SCE-CWC-NIL	SFF-RCC-COM
SRS-CWC-COM	AFF-RCC-NIL	SRA-CWC-NIL	SRS-CWC-NIL	SFF-RCC-RBK
SRS-CWC-RBK	AFC-CWC-NIL	SRA-CWC-COM	SRS-CWC-COM	SFF-RAC-NIL
SFF-CWC-NIL	SCE-CWC-NIL	SRA-CWC-RBK	SRS-CWC-RBK	SFC-CWC-NIL
SFF-CWC-COM	SCE-CWC-COM	SRO-CWC-NIL	SFF-CWC-NIL	SFC-CWC-COM
SFF-CWC-RBK	SRS-CWC-NIL	SRO-CWC-COM	SFF-CWC-COM	
SFF-RCC-NIL	SRS-CWC-COM	SRO-CWC-RBK	SFF-CWC-RBK	
SFF-RCC-COM	SRS-CWC-RBK	SFF-CWC-NIL	SFF-RCC-NIL	
SFF-RCC-RBK	SRA-CWC-NIL	SFF-CWC-COM	SFF-RCC-COM	
SFC-CWC-NIL	SRA-CWC-COM	SFF-CWC-RBK	SFF-RCC-RBK	
SFC-CWC-COM	SRA-CWC-RBK	SFF-BCC-NIL	SFF-RAC-NIL	
SFC-CWC-RBK	SRO-CWC-NIL	SFF-BCC-COM	SFC-CWC-NIL	
	SRO-CWC-COM	SFF-RCC-RBK	SFC-CWC-COM	
	SRO-CWC-RBK	SFF-RAC-NIL		
	SFF-CWC-NIL	SFC-CWC-NIL		
	SFF-CWC-COM	SFC-CWC-COM		
	SFF-CWG-RBK			
	SFF-RCC-NIL			
	SFF-RCC-COM			
	SFF-RCC-RBK			
	SFC-CWC-NIL			
	SFC-CWC-COM			

其他的异常分类和处理

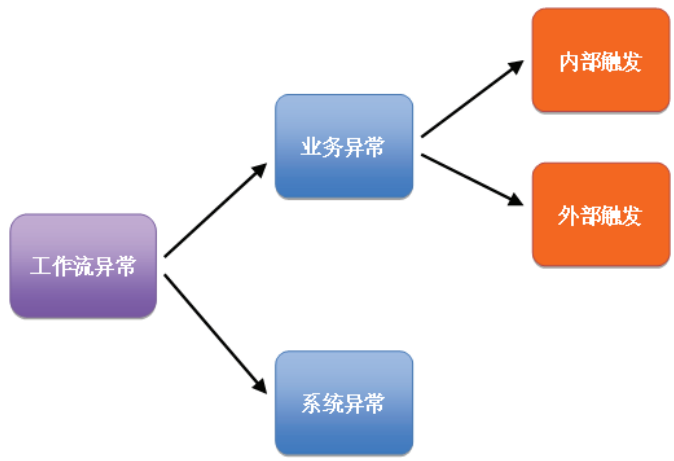
Bruce Silver的异常处理模式

Bruce Silver（http://brsilver.com/）的异常处理基于BPMN，关注如何建模以显式的捕获异常，异常通过异常路径进行处理。这种定义存在的局限：异常处理过于单一，例如，很多时候我们并不需要定义异常路径，我们需要做得仅仅是暂停流程实例并通知流程实例负责人；并未涉及异常处理的细节，例如，对工作项的各种可能异常处理都没有涉及；所捕获的异常属于可预知的异常（所以可以进行建模），对不可预知异常没有涉及。

Bruce Silver根据异常产生的原因进行了分类，如图D-7所示。

- 业务异常：工作项能够执行完成，但是返回异常的结果（排除系统异常）。根据异常产生的原因又分为组织内部原因触发和组织外部原因触发。内部触发包括了活动参与者做出的非期望行为（取消活动）、超时、业务规则约束（资源不可用），外部触发包括了用户突然取消或改变决定。

- 系统异常：与工作项相关的硬件故障、软件故障或网络故障所导致的工作项无法执行。例如工作项调用的Web服务连接超时、调用失败。

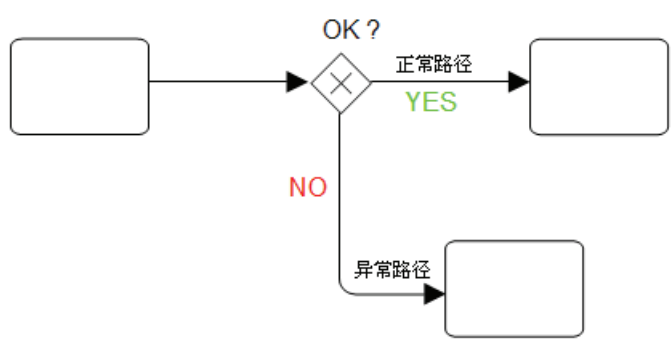


图D-7　根据原因所进行的异常分类

Bruce Silver将异常处理模式分为了7种：内部业务异常处理、重新抛出异常处理、活动超时处理、系统异常处理、非请求的外部异常处理、请求等待响应的外部异常处理和事务。

内部业务异常处理

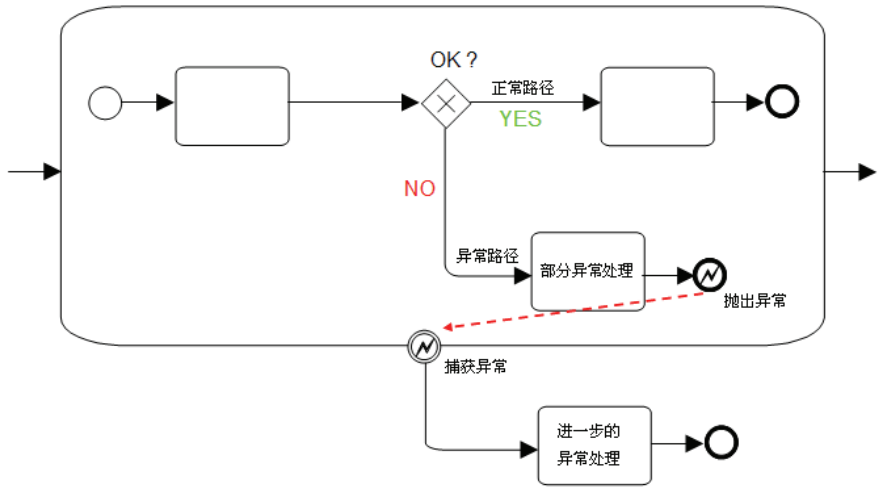
组织内部原因导致的业务异常，包括活动参与者做出的非期望行为和违反业务规则约束，例如活动参与者对订单审批不通过、业务数据不完整、送货前发现缺货。此时使用网关进行建模，不使用异常事件。



图D-8　内部业务异常处理

重新抛出异常处理

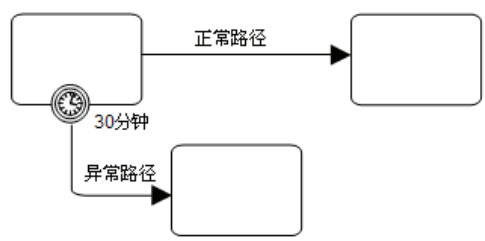
下层流程无法完全处理的异常继续抛出，上层流程继续处理。



图D-9　抛出/捕获异常

活动超时处理

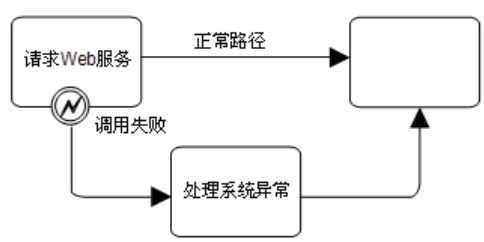
活动处理超时。



图D-10　活动超时处理

系统异常处理

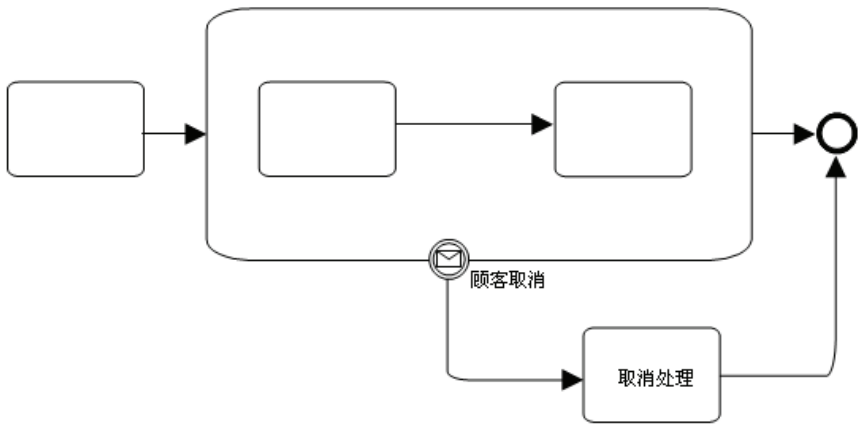
调用的Web服务连接超时、调用失败，与人无关。



图D-11　系统异常处理

非请求的外部异常处理

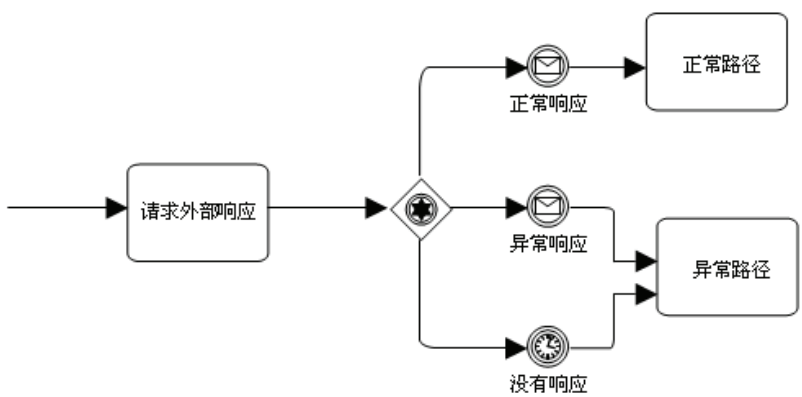
非请求的外部触发意味着流程实例执行过程并未向外部请求响应，事件是由外部驱动触发的。使用消息事件捕获外部触发，将外部触发可能影响的活动建模为块活动或者子流程（建立起上下文），忽略过早或过晚的外部触发。



图D-12　BPMN建模的外部触发异常处理

请求等待响应的外部异常处理

请求等待响应意味着流程实例执行过程中需要与外部交互，向外部发送请求并等待外部的响应。使用事件网关进行建模。



图D-13　请求外部响应异常处理

事务

事务由事务协议所支持，如 WS-Transaction。

Ziv Ben-Eliahu和Michael Elhadad的语义流程建模

Ziv Ben-Eliahu和Michael Elhadad的论文（http://www.cs.bgu.ac.il/~bpmn/wiki.files/Paper.pdf）从Java的Checked异常获取灵感，扩展了BPMN的活动，给活动增加了3种元数据：语义属性、异常和异常处理器。异常定义活动运行期可能产生的异常（异常事件），异常处理器定义活动对应的异常处理器，语义属性对活动进行语义描述。

实际应用时，业务人员快速对乐观路径进行建模，定义活动时输入可能产生的异常然后迅速跳过。稍后，建模工具对活动定义进行检查（类似于Java里的编译检查），如果找不出活动异常对应的处理器，那么报错，强制业务分析人员返回来继续异常处理的建模。如果业务分析人员没有输入异常，那么建模工具会根据活动具有业务语义的关键词或活动定义本身推断出活动运行期可能出现的异常，依据推断异常进行检查。例如，一个通知顾客的活动，语义关键词是通知，那么工具推断出通知包括了3步：发送数据、接受返回数据和对返回数据进行处理，那么可能的异常就包括：发送数据失败、返回数据超时和返回数据非法。基于活动语义的异常推断需要我们建立起相应的语义仓库，实现关键词的检索和推断。

Ziv Ben-Eliahu和Michael Elhadad强制业务分析人员对异常进行建模处理的原因在于：他们认为对于复杂流程来说，业务分析人员往往关注于乐观路径而忘记对异常进行处理；存在不熟悉业务的建模人员，这些人需要从语义推断中获得建议。

我们认为作者还有一层考虑，即认为捕获异常的最佳时机是在建模的时候，也就是在流程执行之前，这和Java的基本哲学一致：糟糕的代码根本就得不到执行。

但Ziv Ben-Eliahu和Michael Elhadad的异常处理忽略了两种情况：一是对所有异常都建模处理会增加流程模型的复杂性，这违反了异常处理的目标：保持流程模型的简单、所有执行者都可理解；二是作为一个通用的实践：大部分流程执行过程中的异常处理都是脱离于工作流系统之外的，是由人来处理的，工作流系统只是通知。工具只是工具。