

10 | 可视化编程：如何有效降低App前后端逻辑开发的技能门槛？

2022-04-04 陈旭

《说透低代码》

课程介绍 >



讲述：陈旭

时长 17:41 大小 16.21M



你好，我是陈旭。

今天我们来聊聊低代码平台实现可视化开发过程中一个难点功能：**可视化编程**。可视化编程解决的是应用开发三部曲（布局、交互、数据）中的交互环节。

但这样说有点狭隘，如果低代码平台同时支持开发后端 **Rest** 服务，那可视化编程的方法可以完全复用到后端的 **Rest** 服务开发中，而不仅限于前端交互逻辑的开发。因此，这一讲的内容实际上同时覆盖了前后端的低代码实现，**如果没有特别的说明，这讲的所有内容都适用于前后端低代码场合下使用。**

在开始之前，我想请你想一想这个问题：编码难在哪？

作为一个写了近 20 年代码的职业码农，面对这个问题，我的第一感觉是难点很多，数都数不清，但要列个一二三来，又觉得不好下手。仔细一想，编码就像艺术创作，比如绘画，虽然绘

画有一定的套路，但从开始到最终完成，有着巨大可自由发挥的空间，而填满这些自由发挥空间的，只能是作者的经验。并且，决定一幅画是否有灵魂的，也只能是作者的经验。

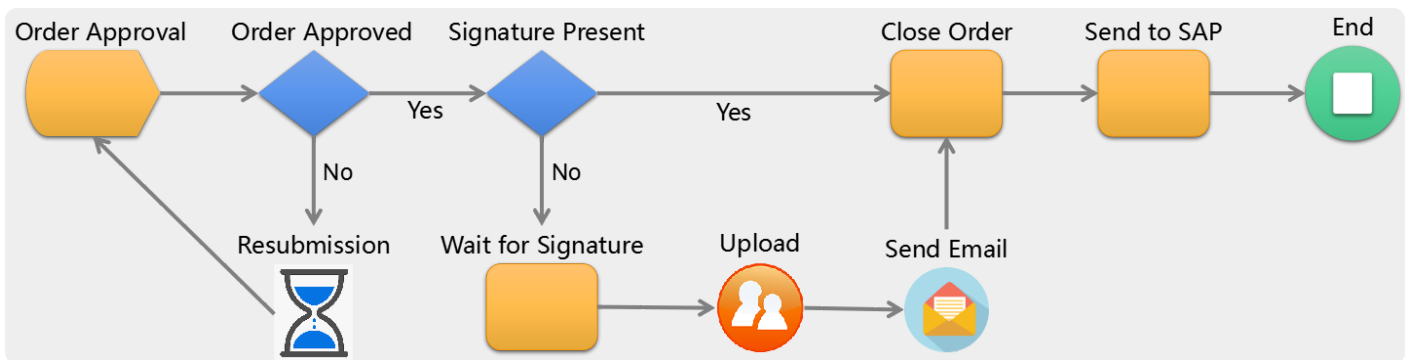
编码何尝不是这样呢？大概套路是有的，但细到每一个函数、每一个类如何编写，则完全由开发人员的经验决定。专家写的代码不仅性能好，**bug** 少，而且可读性非常高，反之，缺乏经验的开发人员能按预期把功能跑通就不错了，哪还顾得上可读性或性能。那有没有一种方法，可以让新手也可以写出专家级的代码呢？这正是我们今天要解决的问题。

可视化逻辑编排

代码具有非常强的流程逻辑，所以可视化编程要解决的第一个问题就是**如何进行逻辑流程的编排**。

编码时的流程逻辑是通过一行行代码自上而下来体现，可视化逻辑编排需要对逻辑有不同的组织方式。一种比较常见的逻辑可视化组织方式是流程图，通过流程图的形式来表达一个逻辑过程是非常自然的想法。

比如下面这个流程图，描述了一个订单审批的过程，看上去逻辑是比较清晰的：



你可以注意到，这个简单的流程图里包含了代码逻辑的三种基本控制结构：循环结构、选择结构、顺序结构，并且这三种结构在图中的呈现和融合都非常自然。关键是，**BOHM & Jacopini** 早在 **1966** 年就从理论上证明了，只要能同时支持这 **3** 种结构的流程，就可以表达任何复杂的程序逻辑。因此，至少在理论上我们不需要担心这个方式的可行性。

而且，你肯定听过，或者用过“一图胜千言”这句话，它指出了我们人类大脑对所处理的信息的“偏好”。人类的大脑和计算机不一样，人类对可视化的信息（比如一张图、一幅画）的处理效率要远高于其他信息（如文字）。实际上，人脑的 **80%** 功能都是用于处理视觉信息的，人们对接受视觉信息具有天生的敏感度。

所以，无论从计算机理论，还是从人脑认知的角度看，流程图式的逻辑编排是一种对大多数人都非常友好的方式，我们在实现可视化逻辑编排时，流程图式的逻辑编排是一种不错的备选。

接下来，再继续深入。我们日常写代码，有些逻辑是要复用的，我们常常使用函数和类来解决代码复用的问题。那么，流程图能搞得定吗？

在展开这个知识点之前，我要先提醒一下，可视化逻辑编排，无需要复刻任何一种编码技巧。不仅如此，实际上，可视化编程只需要实现极少量的编码技巧就可以了。

为啥？因为低代码平台的可视化开发手段多种多样，可视化编程只是其中的一种方法。实际上，低代码编辑器的主要职能就是实现各式各样的可视化开发方式，而这门课的主要内容就是在介绍低代码编辑器。比如前一讲我详细介绍了 App 的布局，这也是一种可视化开发手段。这样应该好理解吧？

那么，可视化编程必备的编码技巧是啥呢？我认为只需有函数定义和调用就足够了，连类都用不上。在低代码意义下的可视化编程，实际上只需要完成交互逻辑即可。

而一个 App 代码的架构、框架部分的代码是不需要低代码平台的使用人员来实现的，更无须通过可视化方式生成的这些代码，这部分的逻辑，低代码编译器会 100% 生成。通过可视化方式编排来的逻辑，会被低代码编译器添加到它自动生成的 App 的框架代码中去。因此，需要低代码的使用者通过可视化编程方式开发的逻辑，是简单、少量的。

那么，我们收回来，采用流程图式的逻辑编排如何来定义和使用函数呢？我认为有两种方法：第一种是显式定义，这是比较比较通用的方法，直接使用一个独立的流程来定义函数的逻辑，并允许在另一个流程里调用；第二种是融合式，也就是将函数“藏”到流程图里。

方法一很好理解，实现起来也简单，无需过多讨论，我们主要说说方法二。

你先想想，一个逻辑流程图能有几个入口和几个出口？一进一出？一进多出？这两种就是最普通的流程图了，对应类似下面这样的逻辑结构：

```
1 function f() {  
2     // ...  
3     if (xxx) {  
4         return 1;  
    }
```

 复制代码

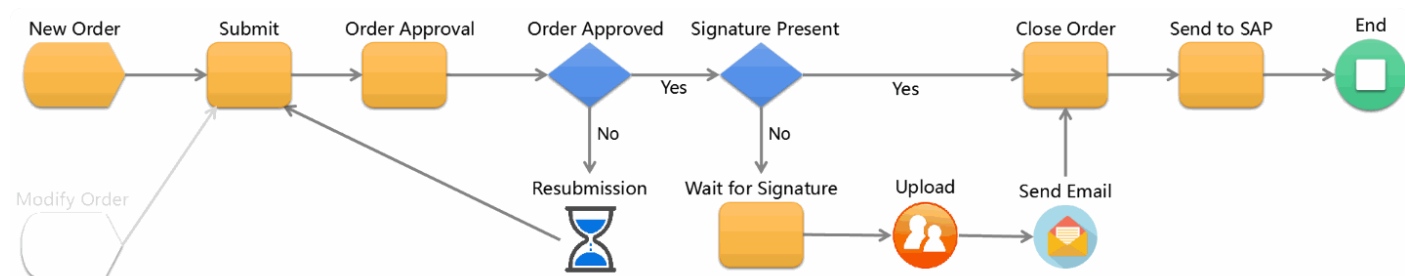
```

5     }
6     // ...
7     if (yyy) {
8         return 2;
9     }
10    // ...
11    if (zzz) {
12        return 3;
13    }
14    // ...
15    return 4;
16 }

```

那如果是多进多出呢？又是啥情况？

仔细一想，好像无法与任何代码逻辑结构对应上啊。其实，多进多出是有的，而且多进多出的流程同时把函数的定义和调用都描述出来了。似乎有点难理解？那我们先看看下面这个图：



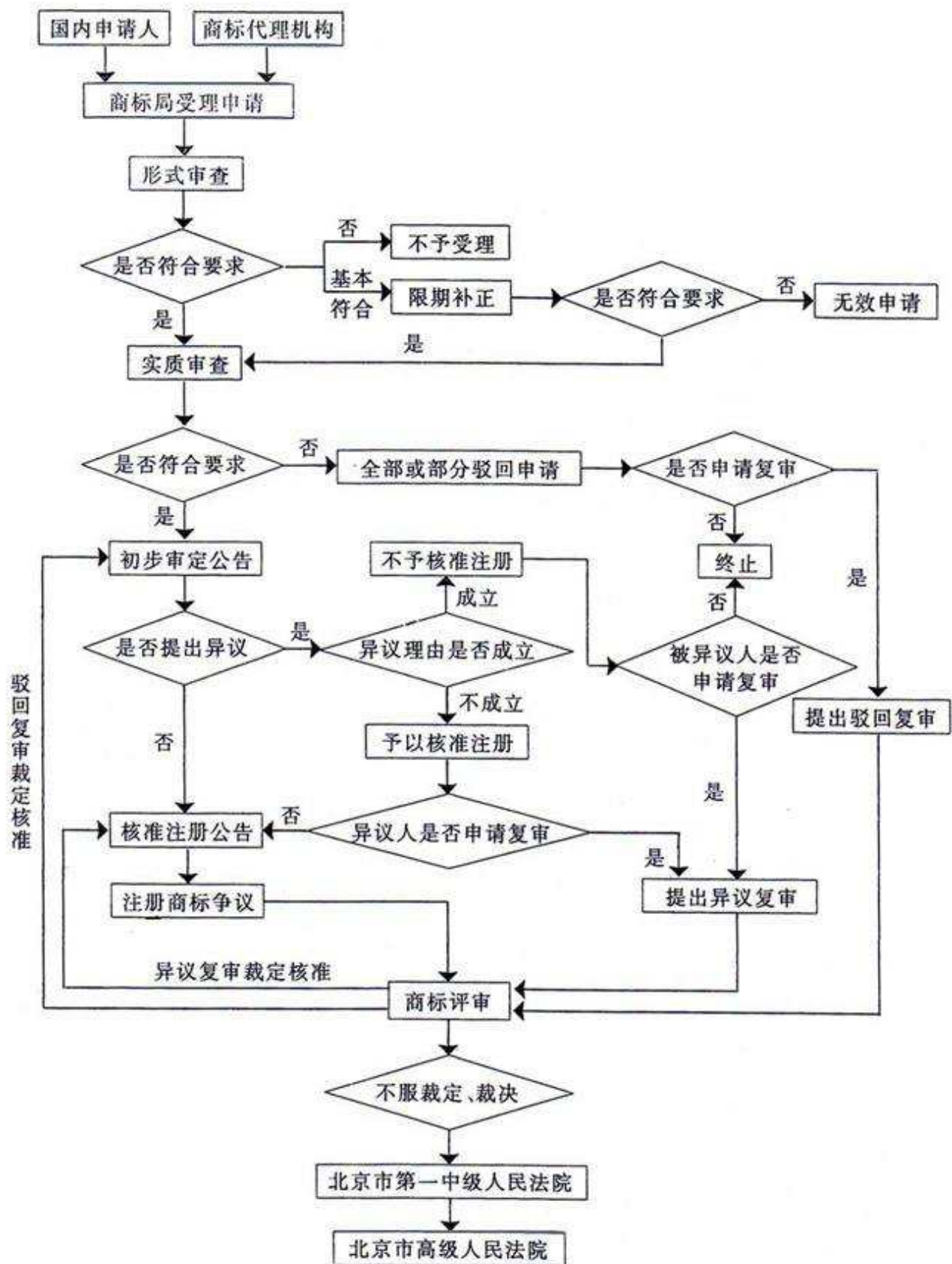
上面这个动画演示了一个多入口流程图。从上图可以看到，它包含了两个独立的工作流。你可以认为一个工作流就对应着一个事件回调函数，或者一个工作流对应着一个 **rest** 服务。在这个情况，两个工作流重叠部分实际上就是一个函数了。

是不是很哇塞？竟然还能有如此优雅自然的函数定义和复用方式！

如果是一个完全没有编程经验的人来用方法一，你是不是还要费一番口舌才能让他理解啥是函数，并且还要求他必须养成先定义后使用的习惯？而采用方法二呢，函数在哪？在哪调用？既在图上，又不在图上，这种方式对人脑非常友好。这就是可视化编程的魔力。

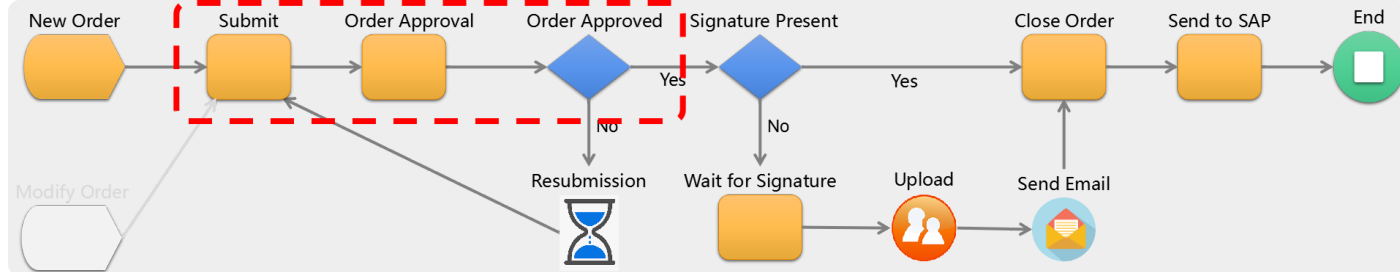
但是！没有一种方法是完美无瑕的。流程图方式编排逻辑方法的最大问题在于，**当逻辑复杂之后，它的可读性和可维护性会极大下降。**

比如下面这个申请商标的流程，你可以先看看：



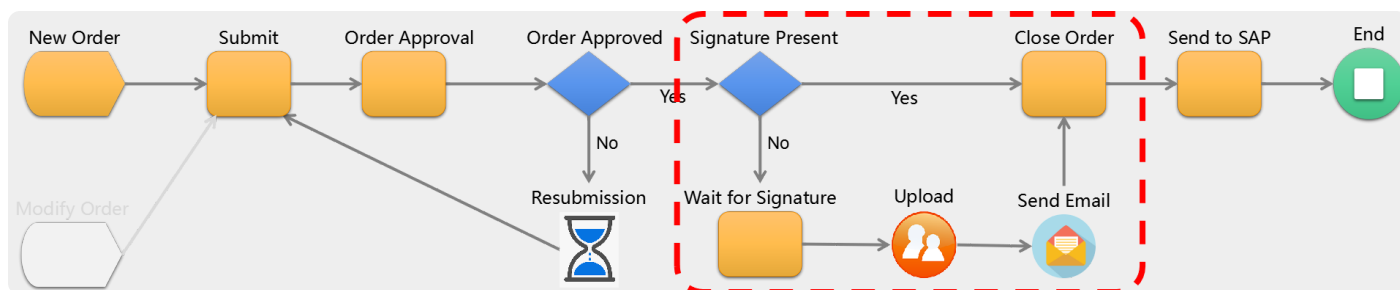
你看懂了吗？反正我没看懂。

那么这个问题应该怎么解决呢？有两个方法。首先我们想到的显然是折叠和展开了。之所以图会复杂，就是因为展示了过多细节，只要我们把不必要的细节隐藏起来，问题就可缓解了。目标确实非常明确，但是如何折叠？万一框选一部分节点，不一定能叠起来呢？



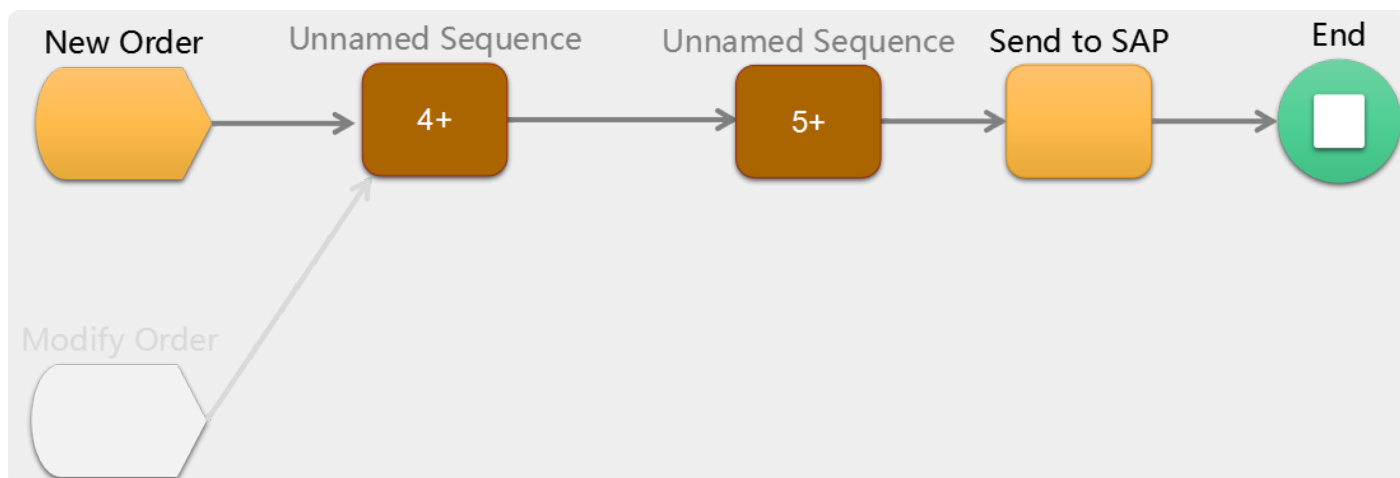
你看，现在上图选中的 3 个节点，就无法折叠，但如果我们把下面那个 **Resubmission** 一起框起来，貌似这块就可以折叠了。那能折叠和不能折叠的特征是啥，它们怎么区分呢？

很简单，就是数一下框住的所有节点有几进几出就好了。一进一出时就可以折叠，否则就不能折叠。你可以再看看下面框住的 5 个节点，一进一出，所以它们可以折叠。反之这 5 个少框了哪个，都不能折叠。



这里我留个小问题给你，如果不选择图中的 **Upload** 节点，那么此时被框住的 4 个节点一共是几进几出？欢迎在评论区留言。

把这 5 块折叠以后，它们就变成了一个子流程了，然后我们再把前文的 4 个节点也叠起来，此时的逻辑图是这样的，看起来就简洁多了：

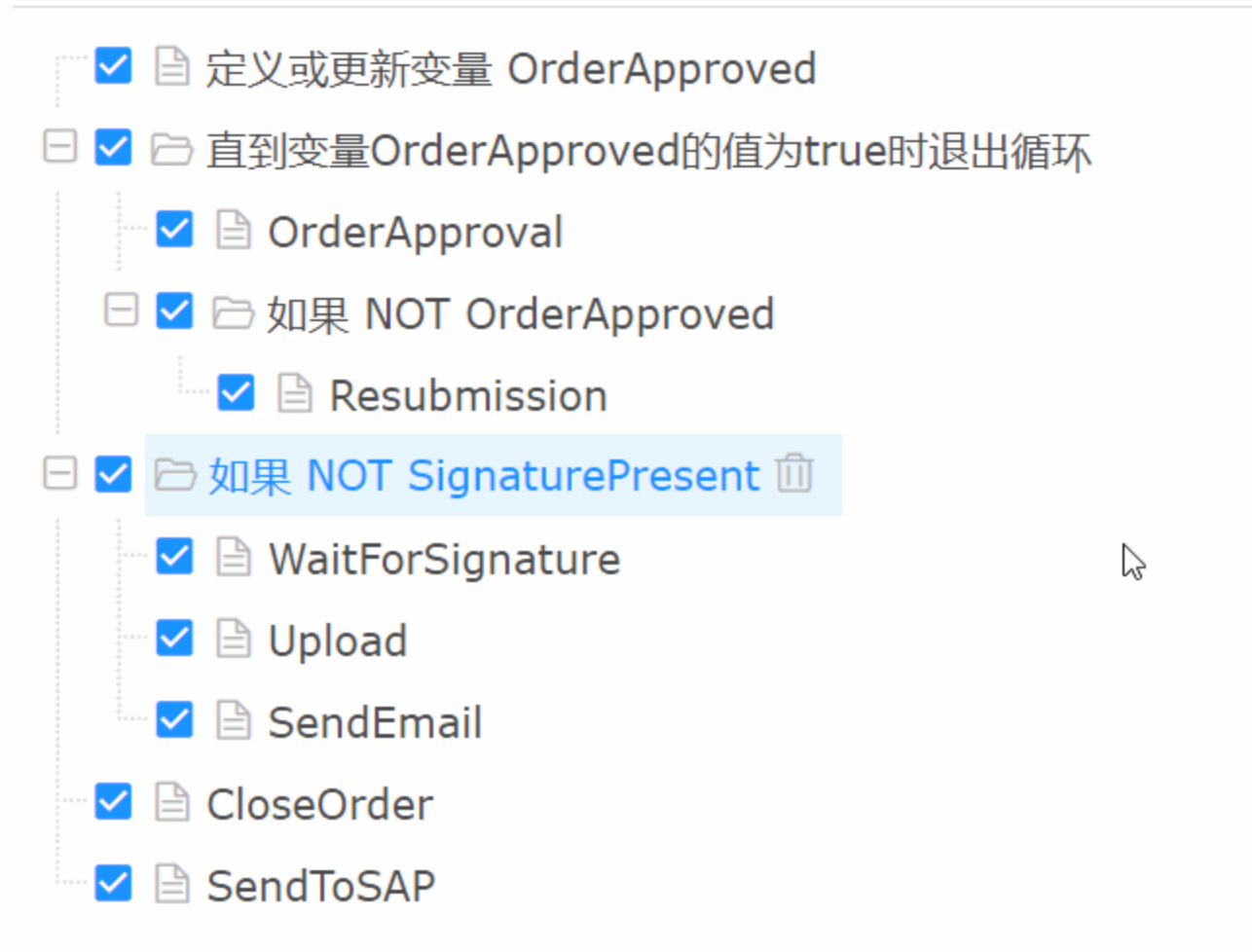


要提升复杂逻辑的可读性，其实还有另一种方法，就是采用树状形式来组织逻辑。比如，把例子中的流程图改写为树状形式，是这的：



显然，采用树状形式来组织逻辑没有流程图那么“酷”，但好处是不挑逻辑复杂度：几乎任何复杂度的逻辑采用树来呈现后，都差不多，而且折叠展开也非常方便。而且我们可以通过拖放树节点的方式，做到对已有逻辑的大调整，这是流程图无法比拟的。

支持鼠标拖拽、Ctrl+C/X/V、Del快捷键；支持Ctrl+点击多选；支持跨事件跨模块拷贝动作；



那么这两种逻辑组织方式，是否只能非此即彼呢？

不是的，满足一定条件时，两种方式是可以相互转换。比如，在选中的节点只有一个入口（出口数量不限）时，这两种逻辑组织方式是等价的，可以相互转换。

不过，到这里，我们都在讨论逻辑流程如何编排。接下来我们更进一步，请你把眼光聚焦到流程（或逻辑树）中的单个节点，我们来看看如何实现可视化开发单个节点的逻辑。

填空即开发

如果说，可视化逻辑编排是编码时的编辑器输入框，那么编排界面上的各个节点就是代码输入框里的一行行代码。当然，这句话说得不全对，一个节点不一定对应着一行代码，有可能对应着几行甚至更多的代码，这是节点与代码行之间表面的差异，它们的实质差异是：节点是功能层面的概念，而代码则是实现层面的概念。

为了更好理解，我们做一个类比：如果说代码是一个个原子，节点则是一个个的分子。高中化学告诉我们，分子都是由原子通过不同的排列顺序组合而成的，原子不具有化学性质，而分子是有化学性质的。一堆氧原子不能用于你呼吸，而两个氧原子结合在一起之后成了氧气，氧气是可以拿来呼吸的。

代码和节点的关系，与原子和分子的关系非常相似。单独一行代码可能没有任何业务意义，但是多行代码按一定顺序组合在一起，就可以形成特定功能。因此，一个节点是由一行或多行代码组成的，具有特定功能。所以我说代码是实现层面的概念，而基于代码组合而成的节点，则功能层面的概念。再延伸一点，如果非要在代码层面上为功能节点找一个位置，你可以把它当做一个函数或者一个类来看待。

理解了代码和节点的关系之后，设计单个节点的方法就明确了：**找功能（而不非找代码）**。比如创建订单功能，或是订单审批功能，或是发送电邮功能，等等，还有很多很多。这样看节点的数量就没完没了了。

节点的功能有大有小。功能越大，意味着使用的条件就越苛刻，所需的功能数量也就越多；反之，功能越小，单个功能的可用场景就越多，通过他们的组合就能形成新的功能，因此，功能节点的数量需求就越少。

那么，你一定会追问，功能多小算小呢？这个问题需要分通用和业务两个层面来回答。先说业务层面，这方面没有标准，只能根据实际业务情况而定。与业务无关的通用功能，则相对明确，我们来详细说说。

所谓的通用功能节点，指的是与编程语言、组件操作等紧密相关的那些功能。比如下面这个图基本就把大多数通用功能节点分类都列举出来了：

程序控制	▼
事件与数据	▼
动画	▼
弹出	▼
数组操作	▼
字符串操作	▼
表单组件操作	▼
Tab组件操作	▼
复杂事件操作	▼
Table组件操作	▼
图形操作	▼
时间操作	▼
进度条操作	▼
步骤条操作	▼
上传操作	▼
组件其它操作	▼
表单操作	▼
高级	▼

你可以根据上面这个图为线索来设计你的通用功能节点。

接下来，我就以条件判断这个功能节点为例，说明功能节点如何实现。因为我们这一讲专注的是可视化编程，那么功能节点的使用当然也是可视化的了。我们可以把一个功能节点粗糙地理解为一个函数，类似的，功能也是有输入参数和返回值的。

我们先来看看**输入条件可视化配置**。

即使是一个条件判断功能，也是要有输入参数的，比如下面这个图，展示了条件判断功能的配置界面：

条件判断 帮助文档 携带数据说明 ☒ 是否执行

根据设定的条件，判断是否要执行归属于此动作的内部动作。

描述 如果 NOT SignaturePresent x

在满足以下 任意一个条件 时，则认为当前的判断条件成立。在计算发生异常时，取 false 作为当前条件的判断依据。

NOT SignaturePresent x +

方法 表达式 v

表达式* 1 SignaturePresent

条件取反 ☒ ?

是不是没想到一个看似简单的条件判断功能节点的输入条件，也能玩出这么多花样来？这样的设计主要有两方面考虑：

1. 判断方法为“表达式”是为方便有少量编程技能的用户使用，直接写一个表达式可以避免啰嗦界面配置，注意这里我把“表达式”设置成了默认值；
 2. 其他判断方法，基本都是给无编程技能者使用的，它覆盖了 JavaScript 常见的条件判断方法。
- 基本上，其他功能节点都可以采用类似方式来处理输入参数的选择。

接下来我们说说**功能节点的结果可视化配置**。

无论功能节点生成的代码有多少，总要通过它的结果对外产生影响，而且这个结果必须交到开发者手里，进而开发者可以将上一个功能节点的结果作为下一个功能节点的输入，从而最终形成一串逻辑。想象一下，逻辑编排出来的逻辑就像一根管子一样，数据就是管子里的水，水从管子的一头流向另一头，中间穿过一个个节点，每一个节点都对数据产生一点影响，最终所有影响叠加到一起，就是开发者所需的最终结果。

那么，如何可视化配置呢？其实超级简单，比如下图的最后一行，有两个输入框，它们都接受一个字符串作为变量名来接收功能节点的结果：

报表数据

维度数据

数据加工

☒ 启用桩数据

☒ 是否支持域

集群

localhost

域

VoLTE

选择报表

联合附着_位置

时间粒度

时间

参数	操作符	值
是否成功*		
\$isSuccess		
返回结果*		
\$result		

与函数的返回值不同，我要求开发者直接提供变量名给我，动作节点对应的代码计算好了后，把结果直接赋值给变量，这样一个配置就可以解决许多问题。

好了，我们现在可以来解释一下这部分的小标题为啥是填空即开发了。每个功能节点都有输入条件和结果需要配置，因此我们为每个功能节点都定制了一个表单，接收开发者对当前功能节点的配置，这就是“填空”的过程。

那填空后呢？填好后，功能节点就根据开发者填写的内容，把代码正确生成出来，这便完成了 **Pro Code** 模式下的“开发”过程。我把这个过程简洁地概括为：填空即开发。

兜底策略

接下来，我们需要讨论一下兜底策略在可视化编程中的应用。

无论可视化编排多么强大，无论功能节点多么丰富，无论这两者多么易用、讨喜，但是你必须承认，它们有可能无法解决所有问题，或者在特定复杂场景下，它们的效率比不上 **Pro Code** 方式。这个时候，我们就需要启用兜底功能了。

先别慌，其实兜底功能实现起来非常简单。先回顾一下咱这个专栏所说的低代码的一个重要特点：它和 **Pro Code** 一样，也是基于代码来构建的应用的。这个特点在今天这讲的内容中有多处体现，比如功能节点在填空背后，实际是把代码生成出来了。

既然填空 = 代码，那是不是可以给开发者一个代码编辑器，让开发者自己把代码填上去，就能解决所有可能出现的问题呢？

答案是肯定的！所以说兜底策略在这里的实现其实非常简单。我这里放了两张图，是我们低代码平台 **Awade** 现在用的方法，你可以参考一下：

高级

打印日志 将给定的数据，在控制台打印出来 (console.log)

自定义代码块 执行一段自定义代码，可用于处理一些复杂问题

页面状态导航 将当前视图切换到另一个状态

解析URL参数 将url参数区里的参数提取出来，并转为一个对象

依赖引入 任何库或类只要引入一次就可全局使用，为利于维护，建议在onInit事件中统一完成引入

休眠/等待 休眠一段时间，再执行后续动作

客户端数据缓存 用于在当前客户端存储一笔数据，可选择会话期有效

+ 添加动作

自定义代码块 [帮助文档](#) [携带数据说明](#)

执行一段自定义代码，可用于处理一些复杂问题

描述一下这段代码都干了啥

```
1 const pluckDeep = key => obj => key.split('.').reduce((accum, key) => accum[key]
2
3 const compose = (...fns) => res => fns.reduce((accum, next) => next(accum), res
4
5 const unfold = (f, seed) => {
6   const go = (f, seed, acc) => {
7     const res = f(seed)
8     return res ? go(f, res[1], acc.concat([res[0]])) : acc
9   }
10  return go(f, seed, [])
11 }
```

简单粗暴，但是好用！

虽然功能实现确实比较简单，但写过代码的人都知道，一个 IDE 绝对不仅仅是能敲代码，能跑就可以的。这些功能可能还占不到 IDE 功能的 10%，其他的如智能提示、出错提示、编译错误等功能是非常影响编码体验和开发效率的。虽然我们这里只是一个代码输入框，但往大了说，你应该把它当做一个 WebIDE 来理解。

但是这个功能点的实现弹性是非常大的，你可以根据你手里的资源，决定要把这个代码编辑器做到哪个程度。

总结

可视化编程是可视化开发中的一个难点，总体可以分成两个主要功能点：**可视化逻辑编排**和**功能节点**。限于篇幅，这一讲我们没有从代码实现层面讨论具体如何实现，但是我已经从方案和特性层面将这两个功能点基本很透了。后续在动态更新部分，我会从代码实现层面来说说这两个功能落地过程中的难点。

我介绍了两种可视化逻辑编排的方法，分别是流程图式和逻辑树式，这两种方式可以相互结合，而不是非此即彼。可视化逻辑编排的方式，不仅可以很好地解决程序逻辑的编排场景，实际上，这个方式还可以独立出来作为一个独立的 App 场景：流程审批场景。我们只要再加上泳道，事件等功能的话，再采用这讲给出的方法，几乎可以直接实现一个符合 BPMN 规范的场景。以后有机会我们展开讨论这个话题。

在功能节点配置界面上，开发者填的是数据，而代码是自动生成的。不仅如此，功能节点的实现靠的不是前后端的语言专家，就是业务专家，功能节点将这些专家的经验凝聚在了它生出的代码中。所以，即使是一个无技能者，只要他正确填写了表单，就可以得到专家级的代码。这就是低代码平台的能力！

这一讲中我只详细讨论了通用型节点的实现，对业务功能型节点则是一带而过。其实，对于一个通用型低代码平台来说，其职责也主要在通用型节点的实现，至于业务功能型节点，我建议你统统移到插件中来定制。

这么做有两个原因，一是业务功能节点数量非常多，胡子眉毛一把抓地全放到平台上，会非常乱；二是恐怕平台团队不知道如何去实现业务功能节点，甚至连要哪些功能节点都不知道，这本就是业务团队才知晓的知识。所以平台团队应该与应用团队合作来开发插件，这样才能利用好业务专家的经验。


思考题

1. 你认为哪些功能节点是比较常用，需要第一优先级实现的？条件判断、数组操作、字符串操作，还有其他哪些呢？
2. 有的功能节点是异步的，可视化逻辑编排如何自动处理异步功能节点？这是一个有点难度的问题，欢迎挑战一下，把你的方案留在评论区。

下一节课我们将要讨论可视化编程中的高低代码混合开发模式，你可以做些准备。我是陈旭，我们下节课见。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 09 | 属性编辑器：如何解除Web组件属性与编辑器的耦合？

[下一篇](#) 11 | 亦敌亦友：Low Code与Pro Code混合使用怎样实现？

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。