NGC-DAG工程化设计

名词	释义
DAG	Directed Acyclic Graph 有向无环图，任意一个节点出发，根据方向无法回到原节点的图
入度	有向图中某点作为图中边的终点的次数之和
出度	对于有向图来说出边条数即为该顶点的出度

一、Why DAG

👍
1. 优化模型的执行
    a. 支持节点的并行计算，提升性能
2. 可以灵活地支持不同语种、车型的对话架构
    a. 根据节点配置和组件池能够快速建立起对话全链路服务
3. 工作流任务存在执行顺序，运行较为复杂。
    a. 每一个工作流任务，是由一系列的子任务和任务与任务间的数据依赖关系构成，任务与任务之间存在数据上的交互，某些任务需要上一个任务的完成才能继续执行，由于这种依赖关系的复杂性，因此使用DAG来描述一个工作流任务

二、目标

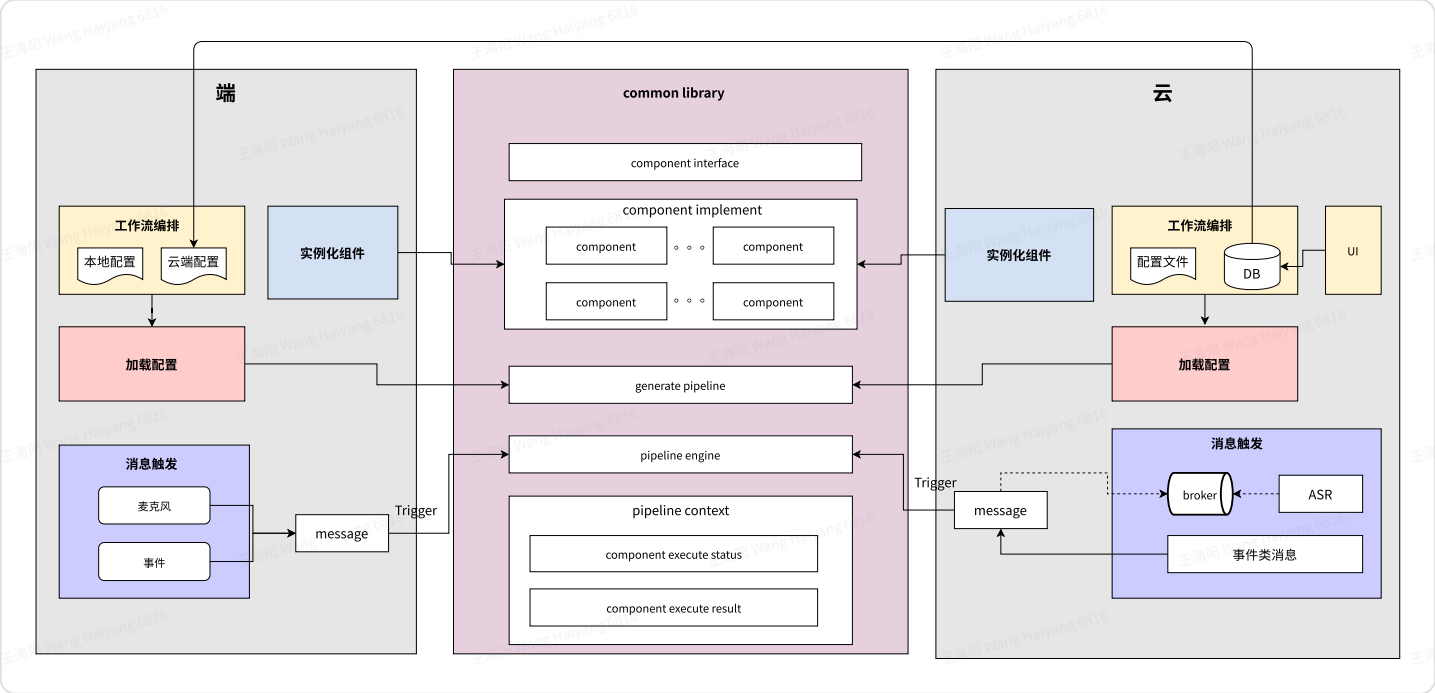
📌 短期：使用工作流引擎及默认的配置跑通云端全链路服务

长期：

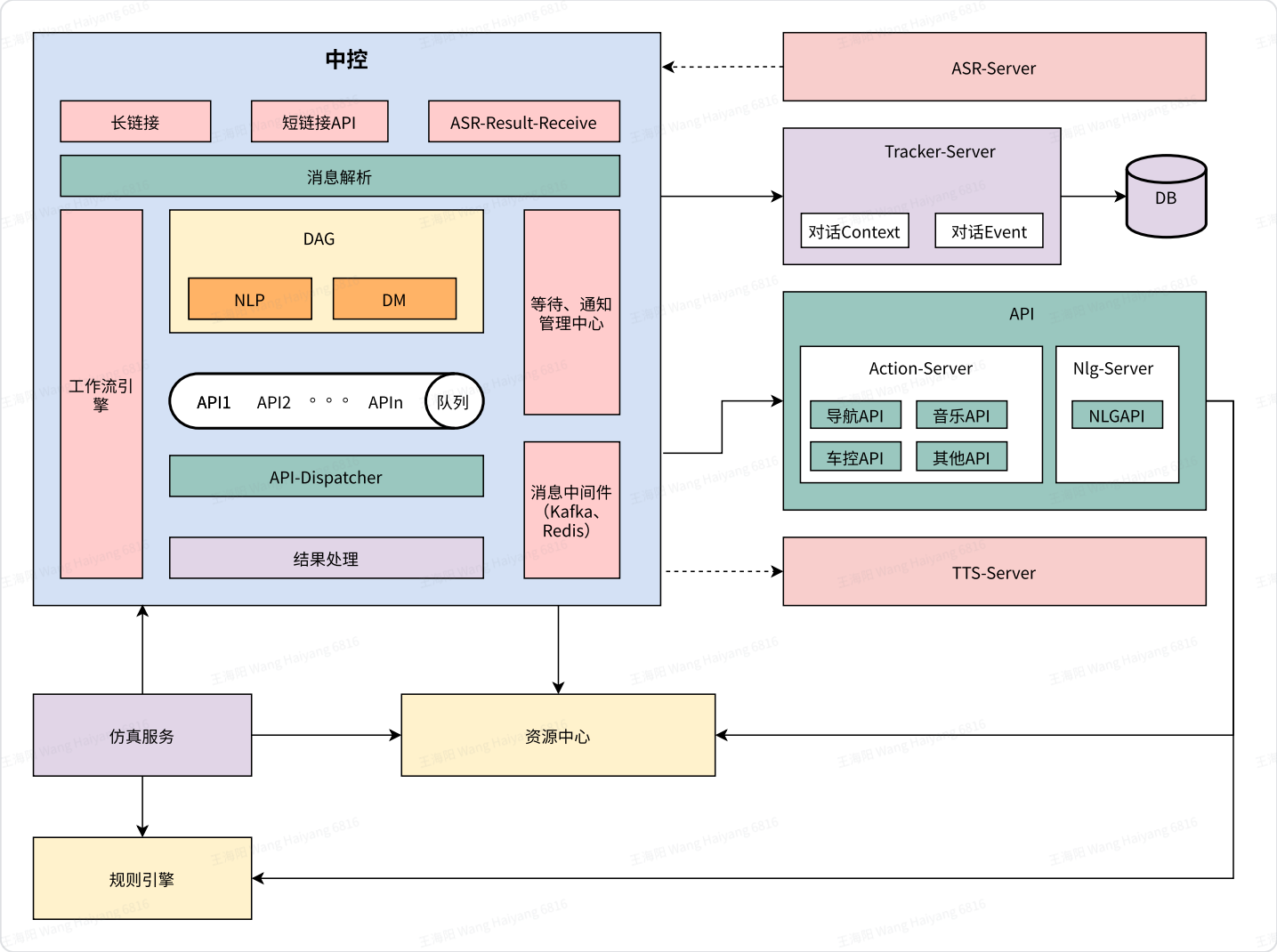
1. 支持平台化部署：通过编排组件池中的组件，全链路服务可以支持不同的语种和车型
2. 端云一体化部署

三、架构设计

3.1、整体架构

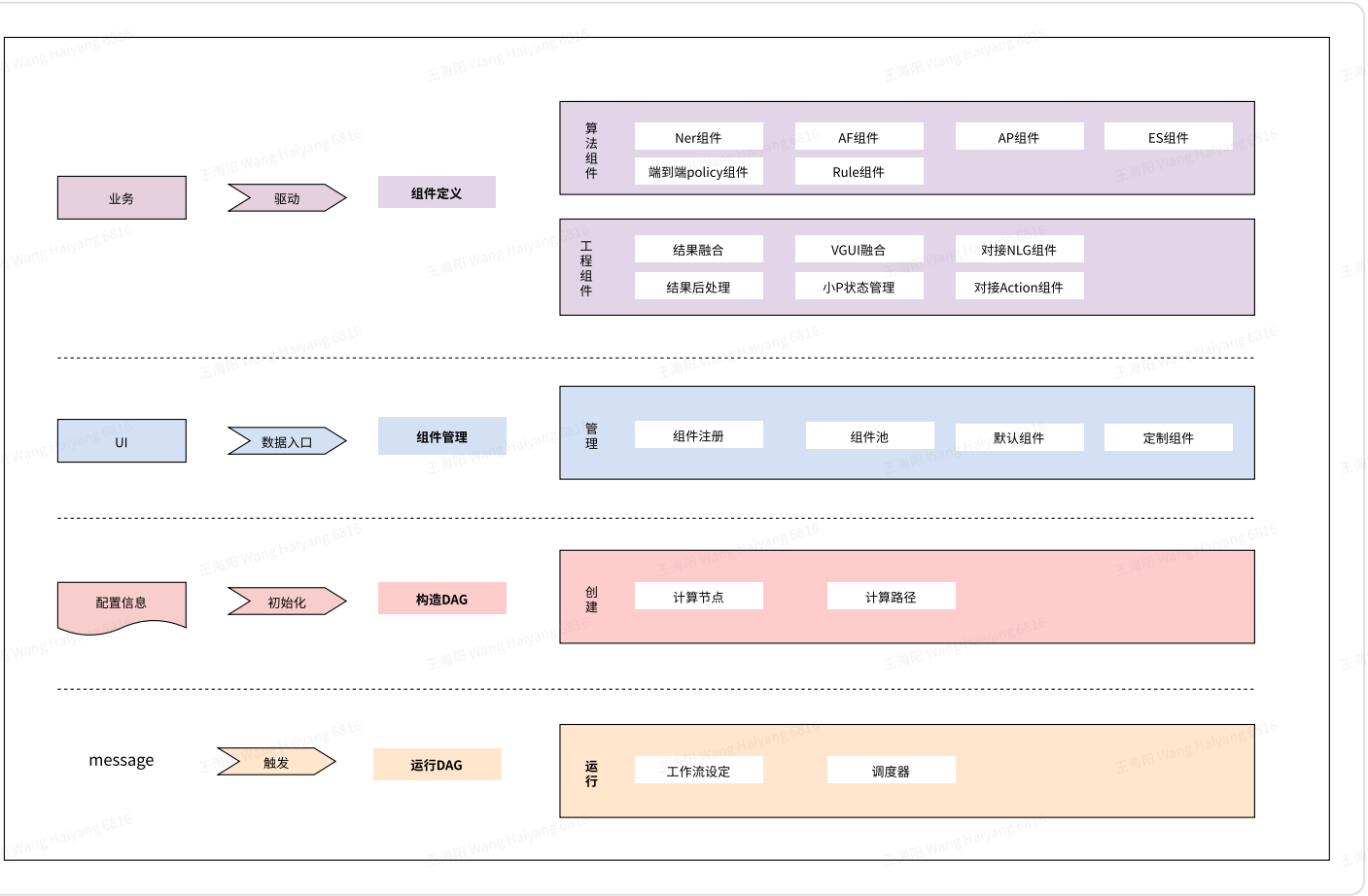


云端架构



待讨论点：ASR、TTS要不要作为组件进行编排？（当前ASR、TTS独立部署，客户端直连）

3.2、详细设计

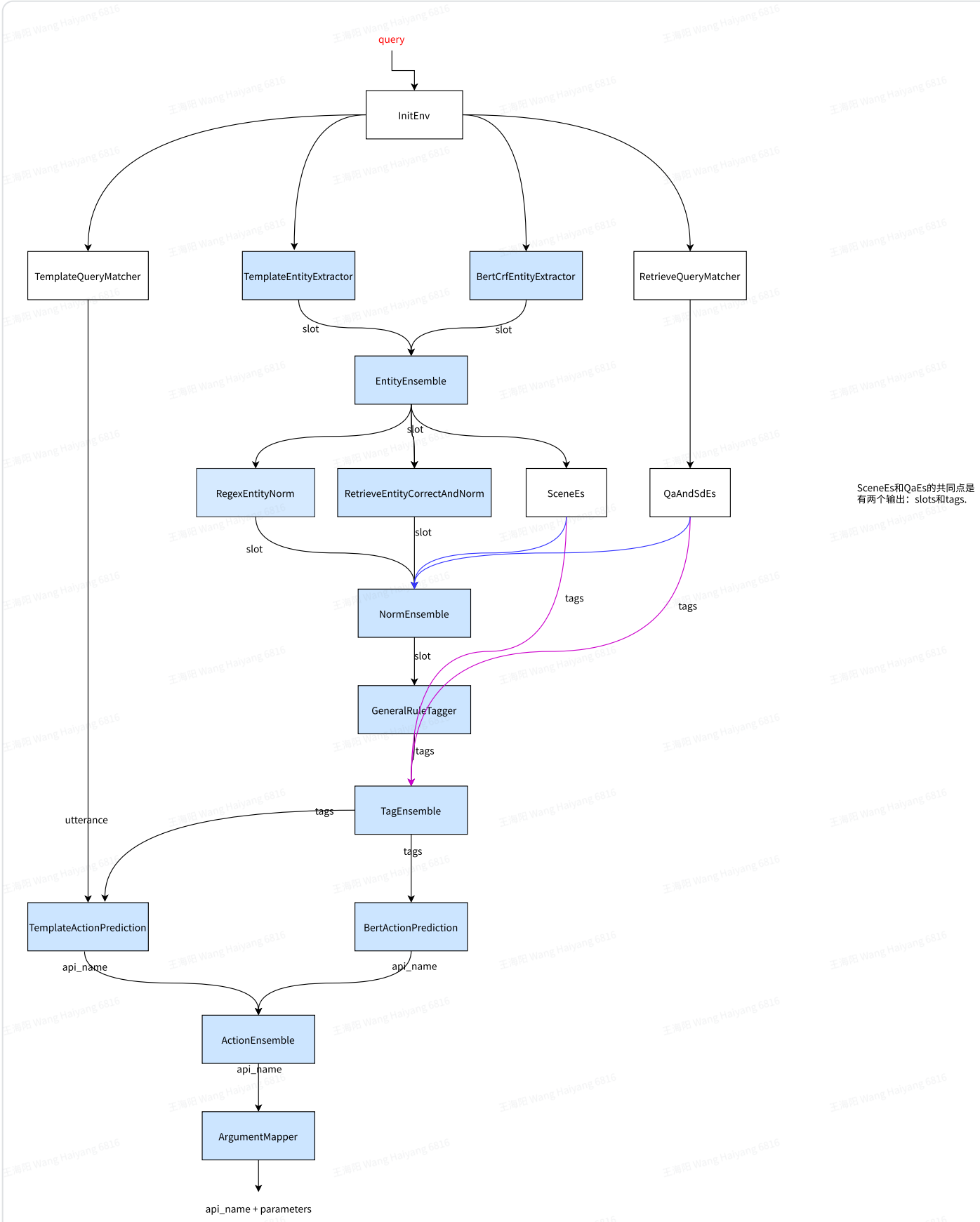


3.2.1、组件定义

3.2.1.1、组件梳理

初步划分出18个组件

（Tracker服务作为存储服务，对应交互没在图中体现，放到业务逻辑侧）



3.2.1.2、Schema定义

3.2.1.2.1、用户请求消息

UserQueryReq

字段	类型	必要数据	备注
query	string	yes	请求的文本
msgId	string	yes	
uid	long	no	小鹏账号的id 未登录： -1
city	string	no	
lon	double	no	经度小数点后6位 从导航客户端获取
lat	double	no	经度小数点后6位 从导航客户端获取
ts	long	yes	时间戳，如：1492397512117
activeApp	string	yes	当前app
activePage	string	yes	前台app的页面
status	string	yes	端上对话状态：start/continue/end/wait
preCmd	string	no	上一个指令码
preTs	long	no	上一个指令时间戳
pre_intent	string	no	上一个意图
vin	string	no	vin码， 大屏需要申请才有VIN码，真实车辆存在
switchData	string	no	服务器下发的保留字段，大屏回传
params	string	no	json格式，预留字段
originalText	string	no	前端asr识别的原始结果
speaking	boolean	no	true：大屏正在tts播报；false
sceneIds	List<String>	no	场景ID
eventType	string	no	大屏事件处理定义
eventData	string	no	
hardwareId	string	no	设备唯一编号
continuousDisplay	String	yes	连续上屏模式，loose/tight

3.2.1.2.2、实体词提取组件

Input

字段	类型	必要数据	备注
query	string	yes	请求的文本
msgId	string	yes	消息Id

output

NerModelBO array结构

字段	类型	必要数据	备注
name	string	yes	实体词名称
pos	array	no	槽所在文本中的位置
rawvalue	string	no	原始槽值

3.2.1.2.3、特征提取归一化组件

Input

字段	类型	必要数据	备注
query	string	yes	请求的文本
msgId	string	yes	消息Id
slots	list<NerModelBO>	no	由ner model 输出作为输入

output

字段	类型	必要数据	备注
query	string	yes	实体词名称
msg	string	no	异常提示消息
slots	List<Slot>	no	归一化之后的槽值

Slot

字段	类型	必要数据	备注
name	string	yes	实体词名称
pos	array	no	槽所在文本中的位置
value	string	no	归一化之后的槽值
rawvalue	string	no	原始槽值
type	string	no	槽值的类别

3.2.1.2.4、AP组件

Input

字段	类型	必要数据	备注
query	string	yes	请求的文本
msgId	string	yes	消息Id
lastApi	string	no	上一轮的api名称
tag	string	no	上一轮的api标签

output

字段	类型	必要数据	备注
actionName	string	yes	需要请求的具体api名称

3.2.1.2.5、Rule组件

该组件暂未启动，待与 樊骏锋 Fan Junfeng 对齐

Input

字段	类型	必要数据	备注
query	string	yes	请求的文本
msgId	string	yes	消息Id

lastApi	string	no	上一轮的api名称
tag	string	no	上一轮的api标签

output

字段	类型	必要数据	备注
actionName	string	yes	需要请求的具体api名称

3.2.1.2.6、policy组件

目前还未启动，待与算法同学对齐 杨如栋 Yang Rudong 樊骏锋 Fan Junfeng 魏子兵 Wei Zibing 宁洪珂 Ning Hongke

字段	类型	必要数据	备注
query	string	yes	请求的文本
msgId	string	yes	消息Id
lastApi	string	no	上一轮的api名称
tag	string	no	上一轮的api标签

Output

字段	类型	必要数据	备注
actionName	string	yes	需要请求的具体api名称
endOfTurn	boolean	yes	该轮是否结束，需要和算法同学对齐，牵涉到多轮

3.2.1.2.7、AF组件

Input

字段	类型	必要数据	备注
msgId	string	yes	消息Id
actionName	string	yes	需要请求的具体api名称
msgId	string	yes	消息Id

slots	List<Slot>	no	归一化之后的槽值

output

字段	类型	必要数据	备注
actionName	string	yes	需要请求的具体api名称
params	list<ActionParamBO>	no	请求参数

ActionParamBO

字段	类型	必要数据	备注
name	string	yes	名称
value	string	yes	参数值
type	string	yes	类型

3.2.1.2.8、Action组件

Input

字段	类型	必要数据	备注
msgId	string	yes	消息Id
name	string	yes	需要请求的具体api名称
arguments	list<ActionParamBO>string	yes	参数

output

字段	类型	必要数据	备注
responses	list<ResponseBO>	yes	actionName对应的查询数据，支持批量查询
events	list<Event>	no	回调事件

ResponseBO

字段	类型	必要数据	备注
code	int	yes	code码，根据此值判定响应结果的正确性
msg	string	no	提示信息
text	string	no	tts文本
actionName	string	no	Action的名称
command	CommandBO	no	需要执行的指令
rule	RuleBO	no	待执行的规则
contentType	string	no	列表数据类型,默认值为none,即无列表
content	list	no	查询内容

CommandBO

字段	类型	必要数据	备注
code	string	yes	执行的指令
param	obj	no	执行参数
runSequence	int	no	执行顺序
delay	int	no	Action的名称

RuleBO

字段	类型	必要数据	备注
ruleId	string	no	规则Id
param	obj	no	执行参数

3.2.1.2.9、Nlg组件

Input

字段	类型	必要数据	备注

msgId	string	yes	消息Id
name	string	yes	需要请求的具体api名称
arguments	list<ActionParamBO>string	yes	参数

Output

确认输出 杨如栋 Yang Rudong 易晖 Yi Hui

字段	类型	必要数据	备注
nlg	string	yes	播报文本

3.2.1.2.10、结果融合组件

Input

字段	类型	必要数据	备注
msgId	string	yes	消息Id
actionOutput	ActionOutput	no	action结果
nlgOutput	NlgOutput	no	nlg结果

Output

WsResp

字段	类型	必要数据	备注
code	int	yes	code码
msg	string	no	异常时赋值
msgId	string	yes	消息Id
msgType	string	yes	消息类别
hardwareId	string	yes	设备ID
data	UserQueryResp	yes	数据体

UserQueryResp

字段	类型	必要数据	备注
nlg	string	no	播报文本
command	CommandBO	no	需要执行的指令
rule	RuleBO	no	待执行的规则
contentType	string	no	列表数据类型,默认值为none,即无列表
answer	list	no	查询内容
status	string	yes	对话状态
priority	int	yes	分值
query	string	no	用户说的话

3.2.1.2.11、结果后处理组件

input

WsResp

Output

WsResp

3.2.2、组件管理

3.2.2.1、注册

由于需要平台化部署，所以组件需要持久化。

Type：post

Url: {host}/v1/component/register

请求体：

字段	类型	允许为空	释义	备注
name	string	N	组件名称	比较重要
desc	string	Y	组件描述	
className	string	N	组件的类对象	
outputClassName	string	Y		

			组件的输出类对象	
inputClassName	string	Y	组件的输入类对象	
parents	list	N	父节点	依赖的节点，为空时表示头节点
childs	list	N	子节点	作为子节点的输入，为空时表示叶子节点
supportedLanguages	list	Y	支持的语种	若为空表示支持所有的语种
supportedCarTypes	list	Y	支持的车型	若为空表示支持所有的车型
modelPath	string	Y	模型存储路径	工程组件时为空

响应：

字段	**类型**	允许为空	**释义**	示例
code	int	N	code码	200

data	list	N	数据体	{
				"id": 1,
				"name": "ner",
				"desc": "实体词识别"
				}

msg	string	Y	消息提示	

3.2.2.2、修改

Type：post

Url: {host}/v1/component/update

请求体：

字段	**类型**	允许为空	**释义**	**备注**

id	long	N	数据id	
name	String	N	组件名称	比较重要
desc	String	Y	组件描述	
parents	list	N	父节点	依赖的节点，为空时表示头节点
childs	list	N	子节点	作为子节点的输入，为空时表示叶子节点
supportedLanguages	list	Y	支持的语种	若为空表示支持所有的语种
supportedCarTypes	list	Y	支持的车型	若为空表示支持所有的车型
path	string	Y	模型存储路径	

响应：

字段	**类型**	允许为空	**释义**	示例
code	int	N	code码	200

data	list	N	数据体	{
				"id": 1,
				"name": "ner",
				"desc": "实体词识别"
				}

msg	string	Y	消息提示	

3.2.2.3、注销

Type：get

Url: {host}/v1/component/delete?id=xxx

请求体：

字段	**类型**	允许为空	**释义**	**备注**

id	long	N	数据id	

响应：

字段	**类型**	允许为空	**释义**	示例
code	int	N	code码	200

data	list	N	数据体	{
				"id": 1
				"isDelete": 1
				}

msg	string	Y	消息提示	

3.2.2.4、查询

Type：post

Url: {host}/v1/component/list

请求体：

字段	**类型**	允许为空	**释义**	**备注**
id	long	Y	数据id	
pageIndex	int	Y		
pageSize	int	Y		

响应：

字段	**类型**	允许为空	**释义**	示例
code	int	N	code码	200
data	list	N	数据体	[{ "id": 1 "name": "ner", "desc": "实体词识别"

				"parents": [] "childs": [] "supportedLanguages": cn "supportedCarTypes":H93 "path":"" }]
msg	string	Y	消息提示	

3.2.3、构造DAG

3.2.3.1、概述

1. 对模块化功能进行组件化，一个组件可以作为一个图节点进行编排
2. 通过指定的配置项进行DAG的构造
   a. 短期：仅通过配置 车型和语种 两个维度的数据即可，走默认的配置节点及工作流
   b. 长期：由于后期可能会引入更多的组件，将提供一个UI界面把所有的组件进行展示供编排
3. 由于对话场景的特殊性，构造的DAG，其入度、出度为0的节点各一个
4. 在编排时初始化默认的节点：

节点	名称	备注
ner	命名实体识别	
featurizer	特征提取、归一化	
ap	action预测	
apRule	ap配套的规则	
policy	对话策略	决策turn及session的结束
af	参数填充	
action	action服务	
nlg	nlg服务	看后期规划，若后期被action直接引用了，那就可以摘除了
resultFusion	结果融合	

resultPost	结果后处理	当前先按一个组件处理，后期按需拆分。里面包含了 小P状态管理、VGUI融合、耗时埋点、结果下发、结果广播等

3.2.3.2、图

图schema：

字段名词	类型	允许为空	释义	备注
nodes	list	N	所有参与的节点名称集合	
edges	list<Edge>	N	边集合	根据组件名称和childNodes

边定义

字段名词	类型	允许为空	释义	备注
source	GraphNode	N	边起点	
destination	GraphNode	N	边终点	

3.2.3.3、节点

GraphNode

字段名词	类型	允许为空	释义	备注
name	string	N	节点名词	一般和组件名词保持一致
uses	string	N	对应的组件类路径	用于构造执行实例
parentNodes	List<GraphNode>	Y	依赖的父节点	
childNodes	List<GraphNode>	Y	子节点	
eager	boolean	Y	是否需要加载（默认为true）	特指模型是否立即加载初始化

isTarget	boolean	N	是否是目标节点	如果target节点执行完毕，相当于整个计算结束。面向编排时使用
input	InputDescriptor	Y	节点的输入	
output	OutputDescriptor	Y	节点的输出	扩展用，可以灵活支持数据的存储和应用，本地缓存 or 分布式缓存

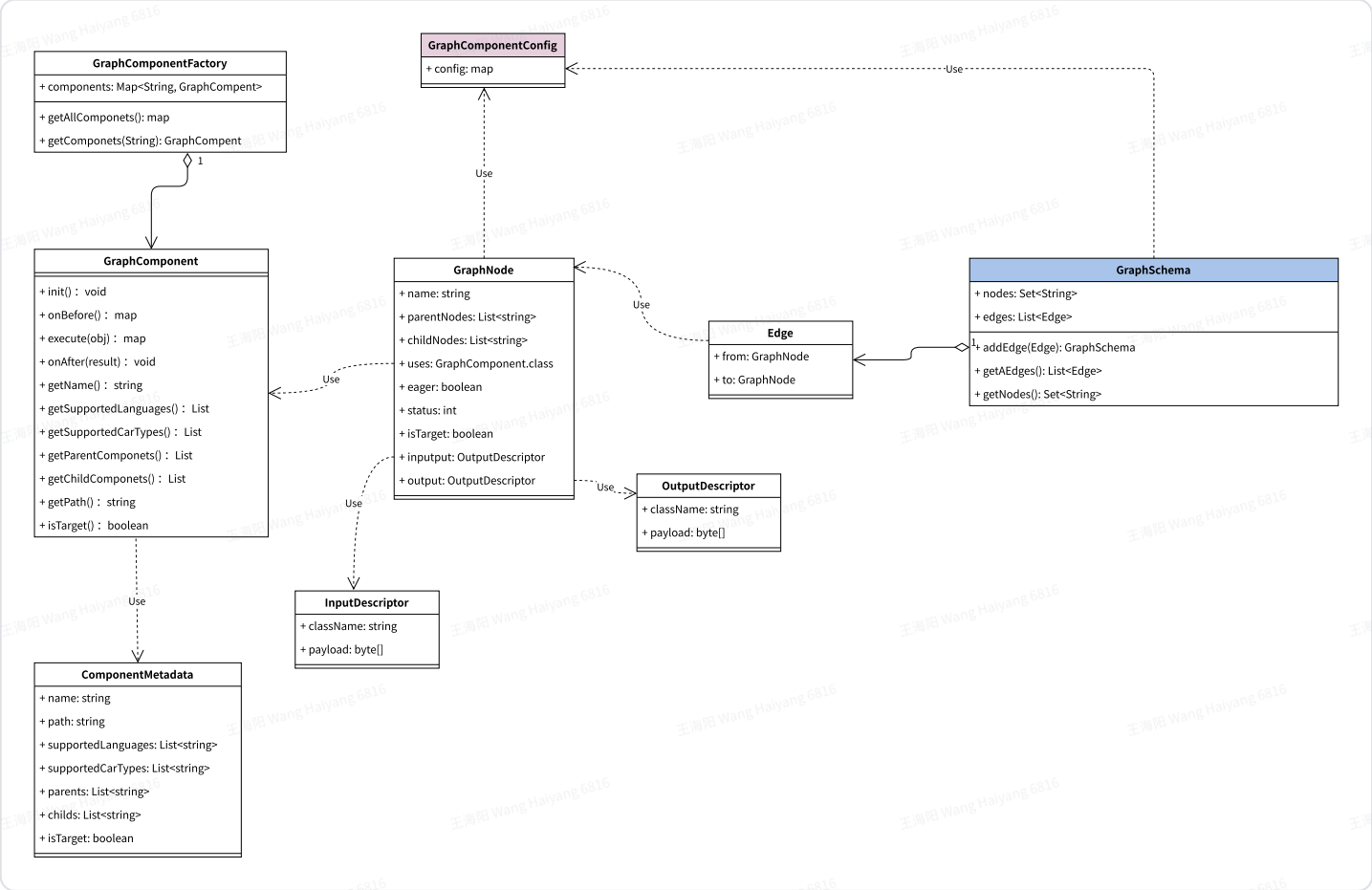
InputDescriptor

字段名词	**类型**	允许为空	**释义**	**备注**
className	string	N	类名	
payload	byte[]	N	数据体	

OutputDescriptor

字段名词	**类型**	允许为空	**释义**	**备注**
className	string	N	类名	
payload	byte[]	N	数据体	

3.2.3.4、核心类图

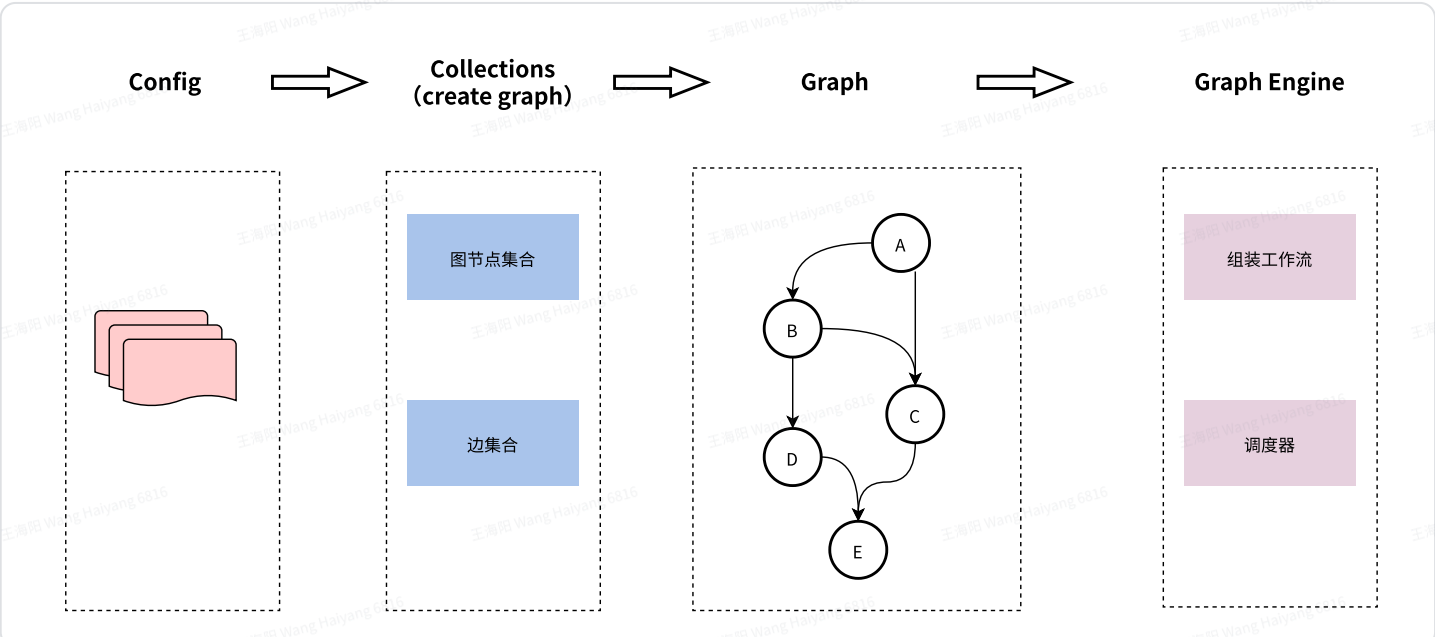


3.2.4、DAG Workflow Engine

基于开源easy-flow框架进行二次开发

调研：工作流引擎方案对比

整体流程图如下：



3.2.4.1、配置文件

当前V2.0阶段使用配置文件进行支持，主要关于两部分信息：组件的依赖关系和分组信息，如下示例：

```yaml
components:
   - name: A
     dependency:
   - name: B
     dependency: A
   - name: C
     dependency: A,B
   - name: D
     dependency: B
   - name: E
     dependency: C,D
```

3.2.4.2、图数据

根据配置文件中的组件信息在服务启动时进行构图，主要生成节点和边

核心代码如下：

```java
public static Graph composeGraph(List<ComponentInfo> components) {
    Set<GraphNode> nodes = new HashSet<>(components.size());
    Map<String, GraphNode> nodeMap = new HashMap<>(components.size());
    for (ComponentInfo component : components) {
        GraphNode node = GraphNode.builder()
                .name(component.getName())
                .className(component.getClassName())
                .parentNodes(component.getDependency())
                .groupName(component.getGroupName())
                .build();
        nodes.add(node);
        nodeMap.put(component.getName(), node);
    }
    Set<Edge> edges = new HashSet<>();
    for (GraphNode node : nodes) {
        final List<String> parentNodes = node.getParentNodes();
        for (String resource : parentNodes) {
            if (StringUtils.isBlank(resource)) {
                continue;
            }
            Edge edge = Edge.builder().sourceNodeName(resource).destinationNodeN
            edges.add(edge);
        }
```

```
    }

    return Graph.builder().nodes(nodes).edges(edges).build();
}
```

示例如下：

```
{
    "edges":[
        {
            "destinationNodeName":"C",
            "sourceNodeName":"B"
        },
        {
            "destinationNodeName":"D",
            "sourceNodeName":"B"
        },
        {
            "destinationNodeName":"B",
            "sourceNodeName":"A"
        },
        {
            "destinationNodeName":"C",
            "sourceNodeName":"A"
        },
        {
            "destinationNodeName":"E",
            "sourceNodeName":"D"
        },
        {
            "destinationNodeName":"E",
            "sourceNodeName":"C"
        }
    ],
    "nodes":[
        {
            "name":"E",
            "parentNodes":[
                "C",
                "D"
            ]
        },
        {
            "name":"A",
            "parentNodes":[
```

```
            ]
        },
        {
            "name":"B",
            "parentNodes":[
                "A"
            ]
        },
        {
            "groupName":"norm",
            "name":"C",
            "parentNodes":[
                "A",
                "B"
            ]
        },
        {
            "groupName":"norm",
            "name":"D",
            "parentNodes":[
                "B"
            ]
        }
    ]
}
```

3.2.4.3、生成执行工作流

根据图数据生成对应的执行工作流，目的是生成工作流引擎需要的执行代码。主要思想包含两部分：

1. 根据图数据为每个节点生成对应的执行优先级（用权重进行标识，权重越小执行优先级越高）

   权重计算公式：所有父节点的权重 + 当前节点的入度

2. 根据分组信息为相关节点打上组信息，根据组的执行属性进行相关代码逻辑的生成

核心代码如下：

```
/**
 * 按图顺序从header节点到tail节点 组装工作流
 * 执行步骤:
 * 1.根据边获取到各个节点的依赖和孩子节点
 * 2.根据边上的各个节点及对应的孩子节点计算各个节点的权重，权重越小计算的优先级越高
 * 2.1 设置被指向节点优先级=所有父节点的和+入度
 * 2.2.遍历边表，先生成一个父节点和子节点的映射map
 * 2.3.再次遍历边表，如果是头节点，默认值设0，如果是尾节点默认设0，已存在=节点原有值+1
 * 2.4.递归给尾节点的所有下游节点+1
```

```java
     * 3.根据权重进行节点的排序
     * 4.根据有序的节点集合构造对应的工作流节点集合
     *
     * @param graph 图
     * @param groupInfos 分组信息
     * @return 带执行舒徐的工作流节点集合
     */
    public static AIWorkFlow composeWorkFlow(Graph graph, Map<String, GroupInfo> gro
        Set<Edge> edges = graph.getEdges();
        //存放孩子节点
        HashMultimap<String, String> childMap = HashMultimap.create();
        //存放父节点
        HashMultimap<String, String> parentMap = HashMultimap.create();
        //获取所有的节点的孩子和依赖节点
        for (Edge s : edges) {
            String sourceNodeName = s.getSourceNodeName();
            String destinationNodeName = s.getDestinationNodeName();
            childMap.put(sourceNodeName, destinationNodeName);
            parentMap.put(destinationNodeName, sourceNodeName);
        }
        //根据边上的各个节点及对应的孩子节点计算各个节点的权重，权重越小执行的优先级越高
        TreeMap<String, Integer> priorityMap = new TreeMap<>();
        for (Edge edge : edges) {
            String sourceNodeName = edge.getSourceNodeName();
            String destinationNodeName = edge.getDestinationNodeName();
            priorityMap.putIfAbsent(sourceNodeName, 0);
            priorityMap.putIfAbsent(destinationNodeName, 0);
            priorityMap.computeIfPresent(destinationNodeName, (k, v) -> v = v + 1);
            //更新目标节点+递归更新自己的孩子节点
            updateChild(priorityMap, childMap, destinationNodeName);
        }
        //取工作流中所有的节点集合
        List<WorkFlowNode> workFlowNodes = getWorkFlowNodes(priorityMap, childMap, p
        return AIWorkFlow.builder().workFlowNodes(workFlowNodes).groupInfos(groupInf
    }
```

生成工作流示例如下：

```json
{
    "groupInfos":{
        "norm":{
            "components":[
                "C",
                "D"
            ],
```

```
            "executeType":"CONCURRENT",
            "name":"norm"
        }
    },
    "workFlowNodes":[
        {
            "childNodes":[
                "B",
                "C"
            ],
            "nodeName":"A",
            "parentNodes":[

            ],
            "priority":0
        },
        {
            "childNodes":[
                "D",
                "C"
            ],
            "nodeName":"B",
            "parentNodes":[
                "A"
            ],
            "priority":1
        },
        {
            "childNodes":[
                "E"
            ],
            "groupName":"norm",
            "nodeName":"D",
            "parentNodes":[
                "B"
            ],
            "priority":2
        },
        {
            "childNodes":[
                "E"
            ],
            "groupName":"norm",
            "nodeName":"C",
            "parentNodes":[
                "A",
                "B"
```

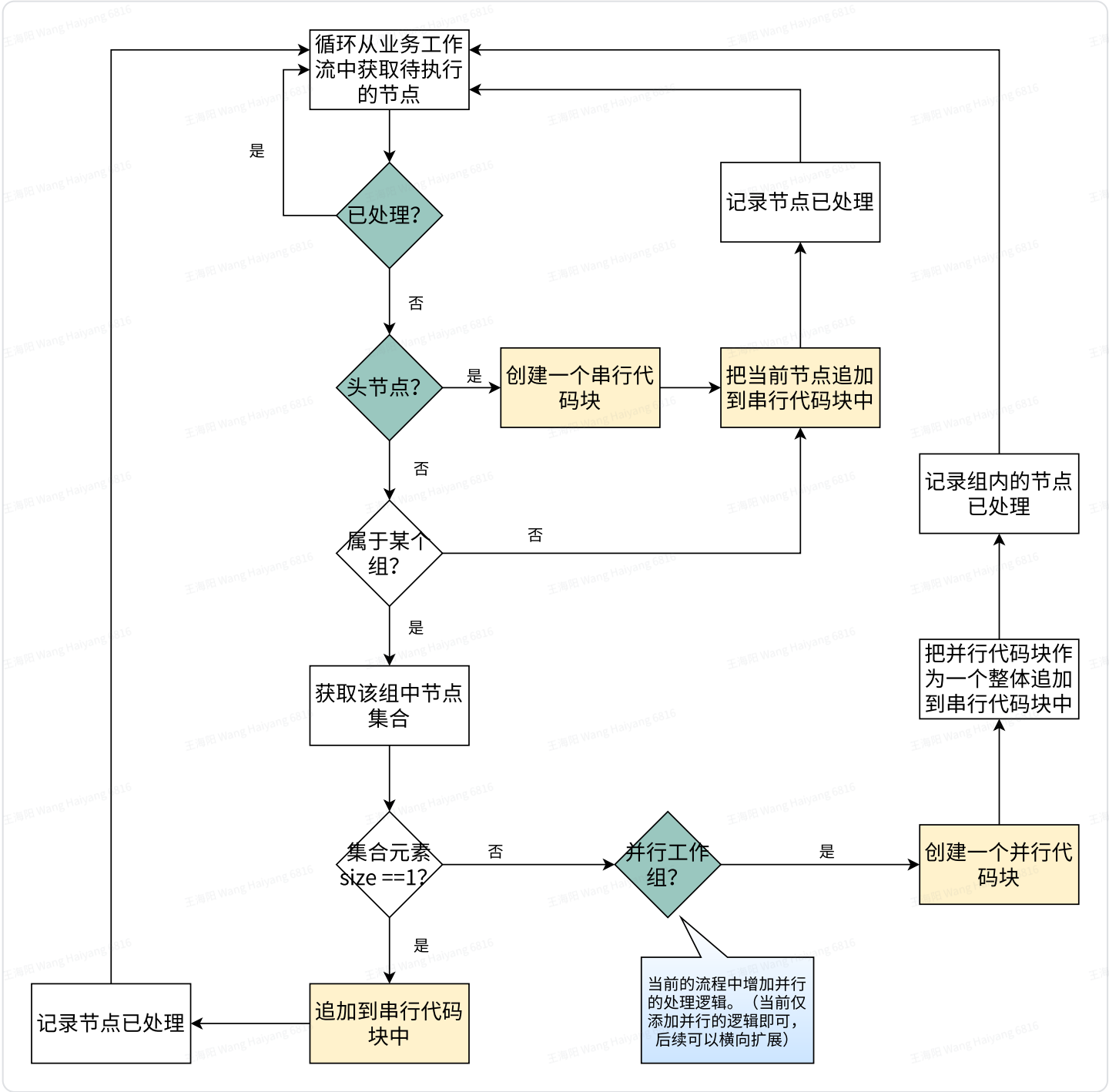
```
            ],
            "priority":3
        },
        {
            "childNodes":[

            ],
            "nodeName":"E",
            "parentNodes":[
                "D",
                "C"
            ],
            "priority":7
        }
    ]
}
```

3.2.4.4、动态生成引擎执行的代码

流程如下：

版本：V1.0



核心代码如下：

```
1  private WorkFlow getDynamicWorkflow(AIWorkFlow workFlow, Map<String, GraphCompon
2      final List<WorkFlowNode> workFlowNodes = workFlow.getWorkFlowNodes();
3      final Map<String, GroupInfo> groupInfos = workFlow.getGroupInfos();
4      //记录节点的处理记录，是否已处理；1-是，（0 or null）-否
5      Map<String, Integer> nodeProcessRecordMap = new HashMap<>();
6      SequentialFlow.Builder.NameStep nameStep = SequentialFlow.Builder.aNewSequer
7      SequentialFlow.Builder.ThenStep thenStep = null;
8      final int nodeSize = workFlowNodes.size();
9      for (int i = 0; i < nodeSize; i++) {
10         final WorkFlowNode node = workFlowNodes.get(i);
11         final String nodeName = node.getNodeName();
```

```
12          //如果节点已处理则pass
13          if (hadProcessed(nodeName, nodeProcessRecordMap)) {
14              continue;
15          }
16          final GraphComponent component = componentMap.get(nodeName);
17          //头节点,由于业务的特殊性，需要初始化运行上下文，所以用初始化环境组件作为了头节点。
18          // 这里头节点的判定未使用GraphUtil.findHeadNodes()函数进行更多复杂的操作，直接
19          if (thenStep == null) {
20              thenStep = nameStep.named("start dynamic workflow").execute(componen
21              markNodeHadProcessed(nodeProcessRecordMap, nodeName);
22          } else {
23              final String groupName = node.getGroupName();
24              //不属于任何一个组，独立运行即可
25              if (null == groupName) {
26                  thenStep.then(component);
27                  markNodeHadProcessed(nodeProcessRecordMap, nodeName);
28              } else {
29                  final GroupInfo groupInfo = groupInfos.get(groupName);
30                  //组中只包含自己一个组件（理论上不推荐这么配置），增加groupInfo == nul
31                  if (groupInfo == null || onlyContainsSelf(groupInfo.getComponent
32                      thenStep.then(component);
33                      markNodeHadProcessed(nodeProcessRecordMap, nodeName);
34                  } else {
35                      final List<String> components = groupInfo.getComponents();
36                      Work[] works = new Work[components.size()];
37                      String[] workNames = new String[components.size()];
38                      int k = 0;
39                      for (int j = 0; j < components.size(); j++) {
40                          final String name = components.get(j);
41                          //防止组件多次执行的情况-处理配置出现错误的场景
42                          if (hadProcessed(name, nodeProcessRecordMap)) {
43                              continue;
44                          }
45                          works[k] = componentMap.get(name);
46                          workNames[k] = name;
47                          k++;
48                      }
49                      //若组内的组件需并行操作（目前业务上仅有并行这一种情况，后续其他的逻
50                      if (ExecuteTypeEnum.CONCURRENT.name().equals(groupInfo.getEx
51                          final ParallelFlow parallelFlow = ParallelFlow.Builder.a
52                          thenStep.then(parallelFlow);
53                          markNodeHadProcessed(nodeProcessRecordMap, workNames);
54                      }
55                  }
56              }
57
58          }
```

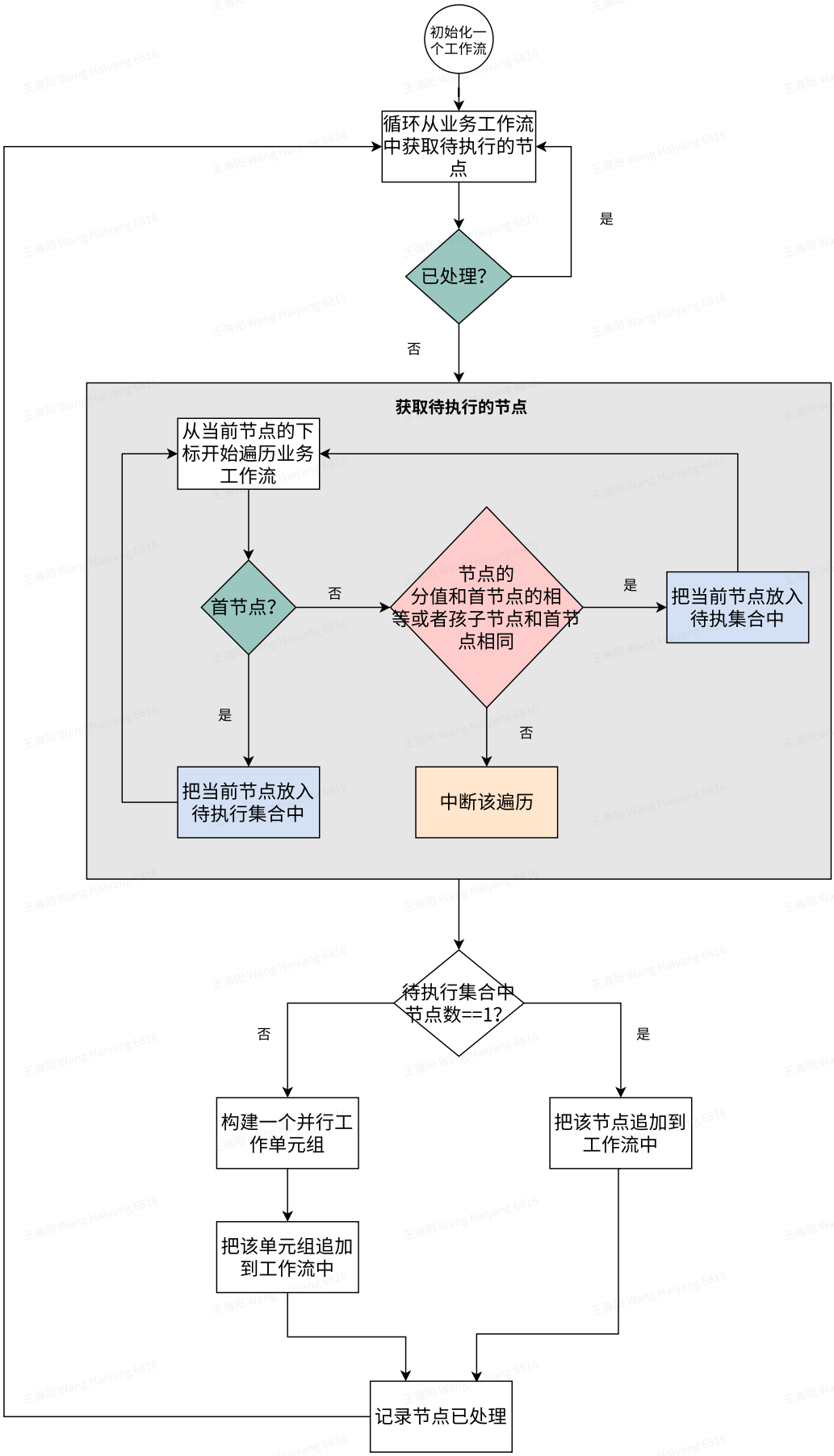
```
59      }
60      if (thenStep == null) {
61          throw new RuntimeException("an exception occurred during get dynamic wor
62      }
63      return thenStep.build();
64  }
```

感兴趣的同学可以看下对应的mr：

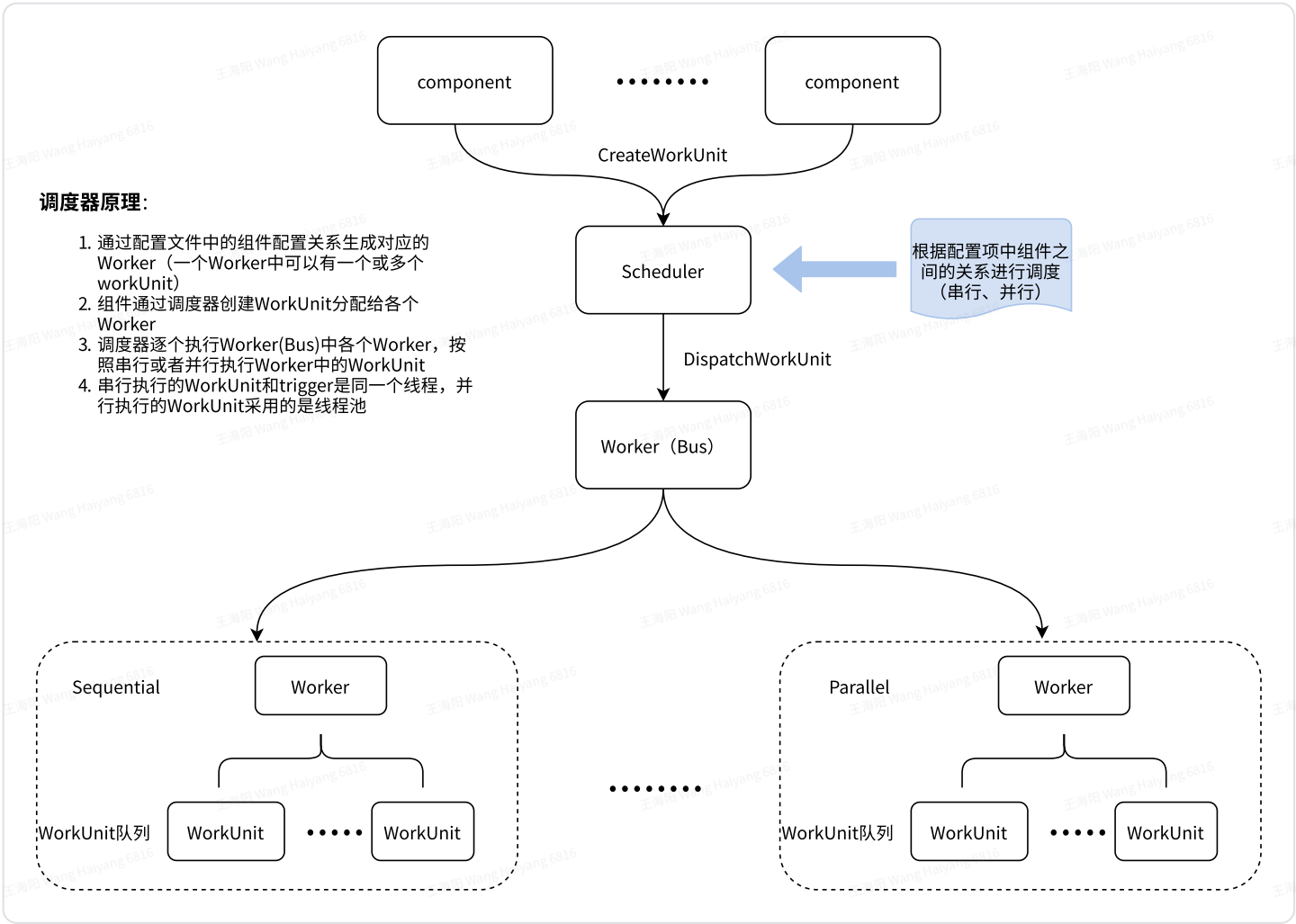
http://gitlab.xiaopeng.local:18080/xai/ngc/xp-ngc-library/merge_requests/94/diffs （端云公共lib）

http://gitlab.xiaopeng.local:18080/xai/ngc/xp-ngc-console/merge_requests/55/diffs （中控）

版本：V2.0（当前使用的是该版本）


初始化一个工作流
循环从业务工作流中获取待执行的节点
已处理?
是
否
获取待执行的节点
从当前节点的下标开始遍历业务工作流
首节点?
否
节点的分值和首节点的相等或者孩子节点和首节点相同
是
把当前节点放入待执执集合中
是
把当前节点放入待执行集合中
否
中断该遍历
待执行集合中节点数==1?
否
是
构建一个并行工作单元组
把该节点追加到工作流中
把该单元组追加到工作流中
记录节点已处理

3.2.4.5、调度器



串行处理逻辑

```
1  public WorkReport execute(WorkContext workContext) {
2      WorkReport workReport = null;
3      for (Work work : workUnits) {
4          workReport = work.execute(workContext);
5          if (workReport != null && FAILED.equals(workReport.getStatus())) {
6              break;
7          }
8      }
9      return workReport;
10 }
```

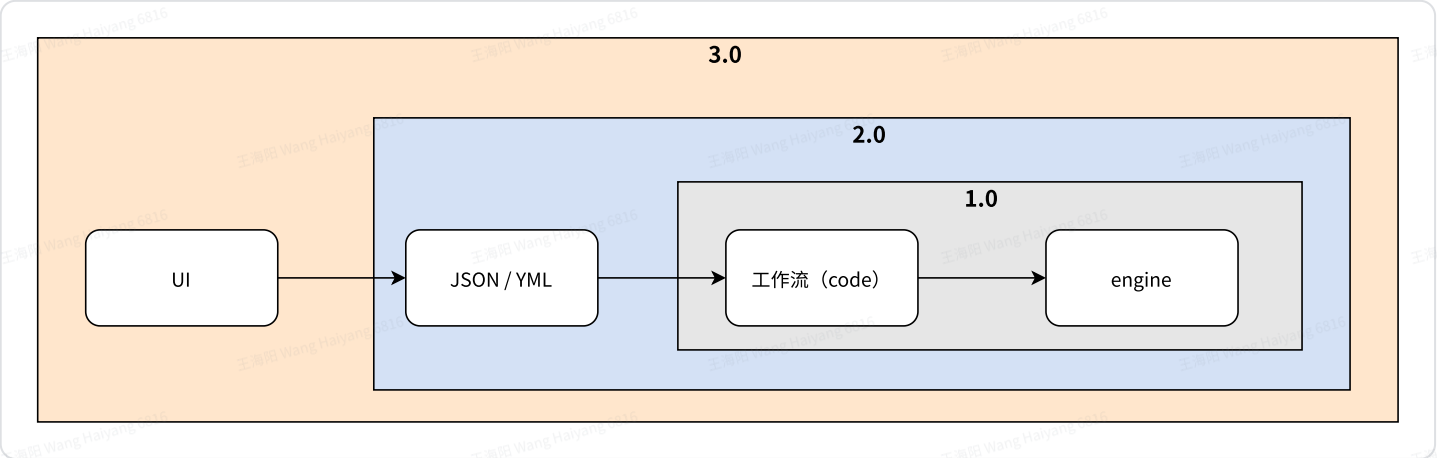
并行处理逻辑

```
1  public ParallelFlowReport executeInParallel(List<Work> workUnits, WorkContext wo
2      List<Callable<WorkReport>> tasks = new ArrayList<>(workUnits.size());
3      workUnits.forEach(work -> tasks.add(() -> work.execute(workContext)));
```

```
    List<Future<WorkReport>> futures = this.workExecutor.invokeAll(tasks);
    Map<Work, Future<WorkReport>> workToReportFuturesMap = new HashMap<>();
    for (int index = 0; index < workUnits.size(); index++) {
        workToReportFuturesMap.put(workUnits.get(index), futures.get(index));
    }
    List<WorkReport> workReports = new ArrayList<>();
    for (Map.Entry<Work, Future<WorkReport>> entry : workToReportFuturesMap.entr
        workReports.add(entry.getValue().get());
    }
    ParallelFlowReport workFlowReport = new ParallelFlowReport();
    workFlowReport.addAll(workReports);
    return workFlowReport;
}
```

3.2.4.6、工作流演进方案

分三个阶段进行推进，当前处于V2.0阶段



V1.0、硬编码引擎所需的技术工作流，使用引擎跑通全链路

V2.0、通过配置项生成业务工作流，通过业务工作流生成引擎需要的技术工作流，再把技术工作流灌入引擎执行

V3.0、通过UI提供编排组件的能力，通过编排生成业务工作流，通过业务工作流生成引擎需要的技术工作流，再把技术工作流灌入引擎执行

四、难点

1. 各个组件schema定义，组件包括：
    a. 模型组件
    b. 工程组件
2. java工程加载模型及部署
3. 工作流编排

4. 工作流引擎的执行
5. 端云一体化

五、工作项-List

工作项

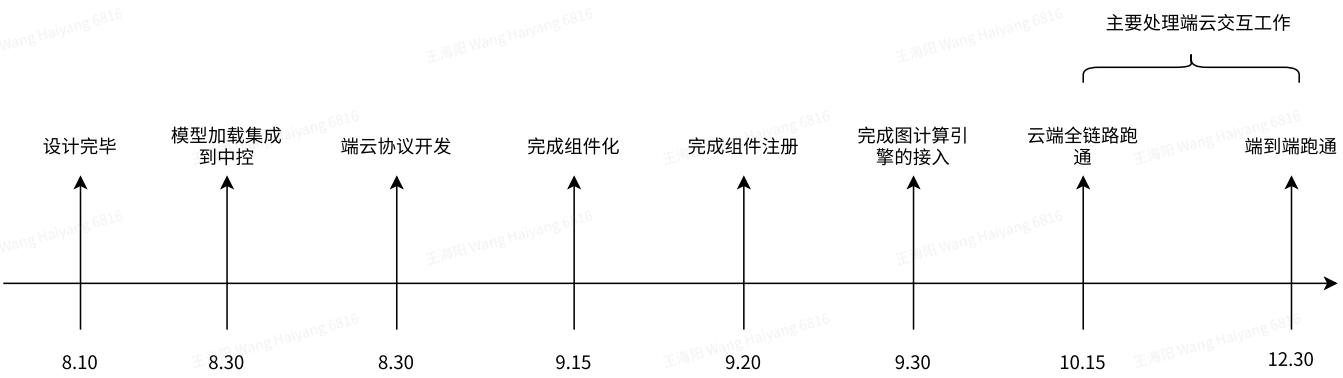
	工作项	日期	人员	版本规划
1.0	10 条记录			
1	整体设计	2022/08/10	杨如栋	1.0
2	模型加载集成到中控	2022/08/30	杨如栋	1.0
3	端云协议开发	2022/08/30	杨如栋	1.0
4	完成组件化	2022/09/15	杨如栋	1.0
5	java加载模型部署到GPU服务节点	2022/08/30	杨如栋	1.0
6	组件注册	2022/09/20	杨如栋	1.0
7	工作流编写	2022/09/21	杨如栋	1.0
8	完成工作流引擎的接入	2022/09/30	杨如栋	1.0
9	云端全链路集成	2022/10/15	杨如栋	1.0
10	集成客户端	2022/12/30	杨如栋	1.0
2.0	5 条记录			
1	输出工作流library	2023/01/05	杨如栋	2.0
2	客户端组件化开发			2.0
3	客户端基于工作流引擎跑通本地对话			2.0
4	本地加载模型实验			2.0
5	基于配置自动转化为工作流	2022/11/30	杨如栋	2.0
3.0	3 条记录			
1	工作流编排提供UI界面			3.0
2	基于UI自动完成工作流的编写	2023/03/30	杨如栋	3.0
3	云端代码进行裁剪客户端运行实验	2023/04/28	杨如栋	3.0

18 条记录

ps： 人员分配TBD

六、里程碑

短期：~ 12.30



七、远期规划

1. 端云一体化，工作流共享：【library形式输出】
2. 提供UI支持组件编排
3. 日志国际化
    a. 不同语种输出的日志进行国际化配置

八、组件化收益

组件化

	组件名称	语种	车型
1	实体词提取	n	m
2	ES(特征提取、归一化)	n	m
3	AP组件	n	m
4	Rule组件	n	m
5	policy	1	1
6	AF组件	1	1
7	Action组件	n	1~m
8	Nlg组件	1	1
9	结果融合组件	1	1
10	结果后处理组件	1	1~m

10 条记录

九、相关调研

1. Rasa-DAG
2. Apache tez https://tez.apache.org/index.html
    a. 基于DAG的数据处理框架
    b. 优化了MapReduce计算
    c. https://github.com/apache/tez
3. 分布式工作流引擎 https://github.com/okayrunner/piper
4. 工作流引擎 https://github.com/j-easy/easy-flows