组合模式

> 组合模式是一种结构性设计模式，可以使用它将对象组合成**树状结构**，并能像独立对象一样使用它们

应用场景

- 需要使用树状对象结构，可以使用组合模式
- 希望客户端代码以相同方式处理简单和复杂元素，可以使用组合模式

优缺点

优点：

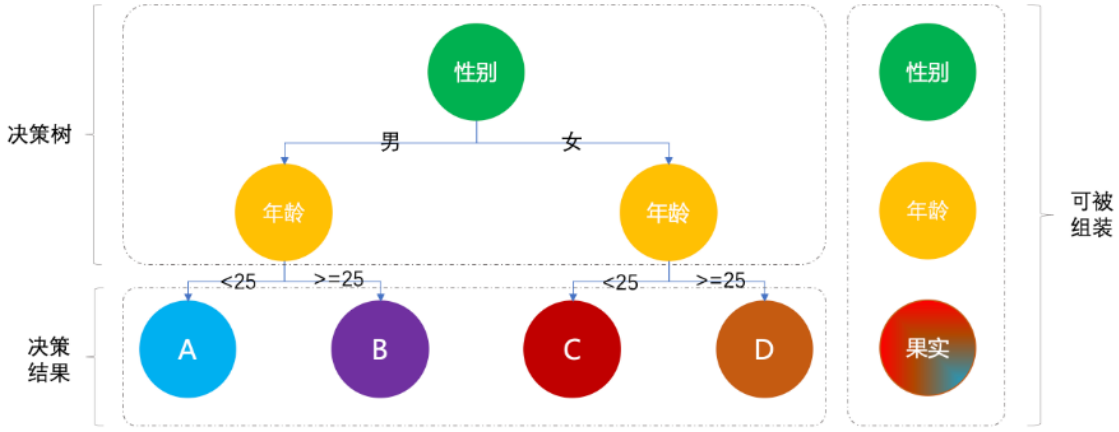
- 可以通过递归和多态的机制方便的使用复杂树形结构
- 开闭原则：无需更改现有代码，就可以在应用中添加新元素，使其成为对象树的一部分

缺点：

- 对于功能差异较大的类，提供公共接口会很困难，在特定情况下，需要过滤一般化组件的接口，令人难以理解

实例场景

精细化运营，使用决策树实现为不同人群推送内容



组合模式的实现：

实现思想是依托决策树的形式，思考一下为什么需要决策树，不用决策树行不行？

当然可以，没有什么是一个类解决不了的CRUD，我们只需要对年龄，性别做一个if-else，就能实现所有的逻辑

```
1  public class EngineController {
2
```

```
    // 直接使用大量if-else判断，后期太难维护了，只要调整一下年龄段就很容易导致系统崩盘
    private Logger logger = LoggerFactory.getLogger(EngineController.class);

    public String process(final String userId, final String userSex, final int userAge) {

        logger.info("ifelse实现方式判断用户结果。userId：{} userSex：{} userAge：{}"
                , userId, userSex, userAge);

        if ("man".equals(userSex)) {
            if (userAge < 25) {
                return "果实A";
            }

            if (userAge >= 25) {
                return "果实B";
            }
        }

        if ("woman".equals(userSex)) {
            if (userAge < 25) {
                return "果实C";
            }

            if (userAge >= 25) {
                return "果实D";
            }
        }
        return null;
    }
}
```

但这样的代码相当难维护，也很难拓展，所谓牵一发而动全身不过如此

什么是决策树?

> **决策树** 记录了当前根节点和全部节点列表

- **非叶子节点**：策略名，节点策略列表
- **节点策略**：下一个子节点，当前决策类型（大于，等于，小于条件）
- **叶子节点**：定义最终的决策，例如推送二次元文章，推送财经类文章

决策树遍历过程：

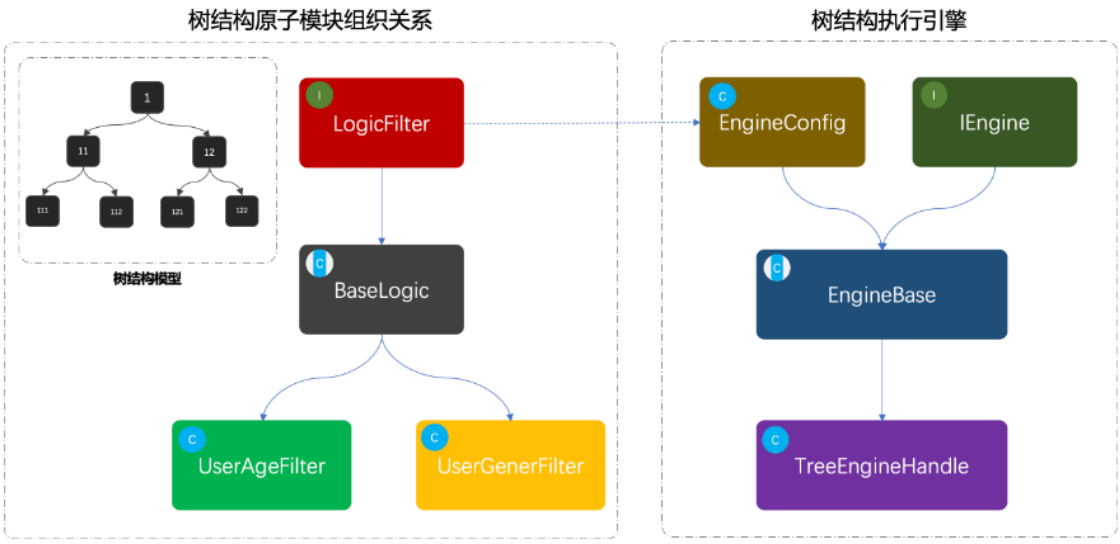
1. 发现当前节点为非叶子节点，根据策略名获取当前决策
2. 根据策略数据遍历节点策略列表，判断满足哪一种节点策略
3. 根据满足的节点策略获取下一个节点（节点移动）
4. 发现当前节点为叶子节点（果实），返回节点数据，遍历结束

其实本质上和树的遍历逻辑类似，只不过需要更多的业务逻辑来支撑维护

工程代码

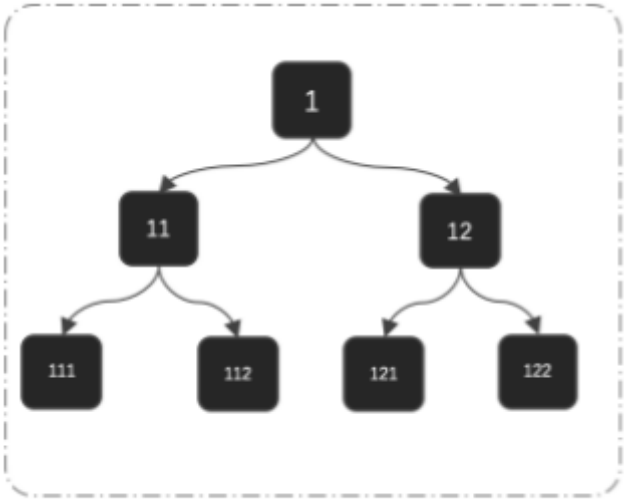
```
bantanger-demo-design-8-02
└── src
    ├── main
    |   └── java
    |       └── com.bantanger.demo.design.domain
    |           ├── model
    |           |   ├── aggregates
    |           |   |   └── TreeRich.java // 决策树聚合类，存储树根信息treeRoot和树节点Map<Long,TreeNode>
    |           |   └── vo
    |           |       ├── EngineResult.java // 执行引擎处理决策的结果类，也就是最后呈现结果打印出来的Vo
    |           |       ├── TreeNode.java // 实现逻辑树节点的类，通过规则Key和描述确定接口走向
    |           |       ├── TreeNodeLink.java // 实现逻辑树各个节点的连接和连接规则
    |           |       └── TreeRoot.java // 逻辑树树根，存储逻辑树ID，名称和树根节点的聚合类
    |           └── service
    |               ├── engine
    |               |   ├── impl
    |               |   |   └── TreeEngineHandle.java // 执行引擎实际处理类
    |               |   ├── EngineBase.java // 基础执行引擎，实现IEngine的部分方法
    |               |   ├── EngineConfig.java // 执行引擎配置类
    |               |   └── IEngine.java // 统一执行引擎接口规范
    |               └── logic
    |                   ├── impl
    |                   |   ├── UserAgeFilter.java // 逻辑走向年龄接口
    |                   |   └── UserGenderFilter.java // 逻辑走向性别接口
    |                   └── LogicFilter.java // 统一逻辑走向接口规范
    └── test
        └── java
            └── com.bantanger.demo.design.test
                └── ApiTest.java

```

组合模式模型结构



来自小傅哥

来看看具体树状结构图（真实节点）



树结构模型

- 首先可以看下黑色框框的模拟指导树结构；`1`、`11`、`12`、`111`、`112`、`121`、`122`，这是一组树结构的ID，并由节点串联组合出一棵关系树树。
- 接下来是类图部分，左侧是从`LogicFilter`开始定义适配的决策过滤器，`BaseLogic`是对接口的实现，提供最基本的通用方法。`UserAgeFilter`、`UserGenerFilter`，是两个具体的实现类用于判断`年龄`和`性别`。
- 最后则是对这颗可以被组织出来的决策树，进行执行的引擎。同样定义了引擎接口和基础的配置，在配置里面设定了需要的模式决策节点。

```
static {
     logicFilterMap = new ConcurrentHashMap<>();
     logicFilterMap.put("userAge", new UserAgeFilter());
     logicFilterMap.put("userGender", new UserGenderFilter());
}
```

基础对象

包路径	类	介绍
model.aggregates	TreeRich	聚合对象，包含组织树信息（传参对象，结合TreeRoot和TreeNodeMap）
model.vo	EngineResult	决策返回对象信息（最终结果走向打印返回信息）
model.vo	TreeNode	树节点；子叶节点（建立规则逻辑，节点值为null）、果实节点（通过节点值作为比较最终决策去向）
model.vo	TreeNodeLink	树节点链接链路
model.vo	TreeRoot	树根信息

上述类实现（get，set可以通过lombok生成，也可以自己实现）

```
public class TreeRich {
    private TreeRoot treeRoot;                  // 树根信息
    private Map<Long, TreeNode> treeNodeMap;    // 树节点ID -> 子节点
}
```

```
public class EngineResult {
    private boolean isSuccess;  // 执行结果
    private String userId;      // 用户ID
    private Long treeId;        // 规则树ID
    private Long nodeId;        // 果实节点ID
    private String nodeValue;   // 果实节点值
}
```

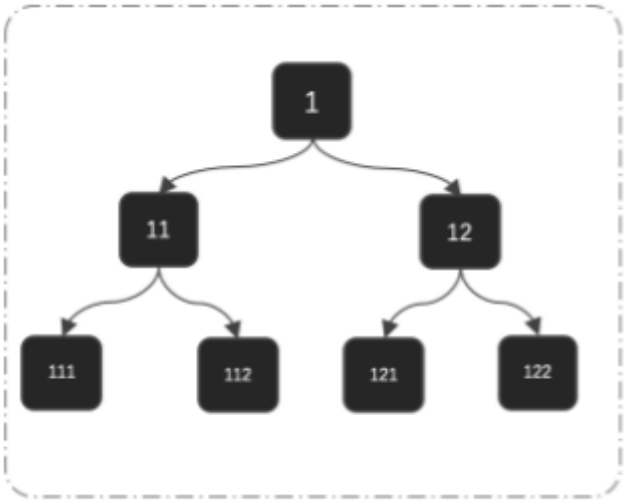
```
public class TreeNode {
    private Long treeId;        // 规则树ID(相当于根节点)
    private Long treeNodeId;    // 规则树节点ID
    private Integer nodeType;   // 节点类型: 1.子叶  2.果实
    private String nodeValue;   // 节点值[nodeType=2]:果实值
    private String ruleKey;     // 规则Key
    private String ruleDesc;    // 规则描述
    private List<TreeNodeLink> treeNodeLinkList; // 节点链路
}
```

```
public class TreeNodeLink {
    private Long nodeIdFrom;        // 节点From
    private Long nodeIdTo;          // 节点To
    private Integer ruleLimitType;  // 节点比对方式 1.= 2.> 3.< 4.<= 5.>= 6.enum[枚举范围]
    private String ruleLimitValue;  // 节点比对值 (决策类型 => 男女/年龄段)
}
```

```java
public class TreeRoot {
    private Long treeId;            // 规则树ID
    private Long treeRootNodeId;    // 规则树根ID
    private String treeName;        // 规则树名称
}
```

鉴于大家可能不太明白为什么要这样设计，我就先把之后测试类初始化挪过来帮助理解

严格对照这个图进行节点的创建于连接



树结构模型

```java
/**
     * 初始化，创建决策树所有逻辑节点并进行逻辑连接
     */
@Before
public void init() {
    // 节点：1 根节点，决策类型为性别，将人群分为男女两类
    TreeNode treeNode_01 = new TreeNode();
    treeNode_01.setTreeId(10001L);  // 规则树创建，记录ID为10001L
    treeNode_01.setTreeNodeId(1L);  // 规则树节点创建，记录ID为1L
    treeNode_01.setNodeType(1);     // 标记节点类型为子叶
    treeNode_01.setNodeValue(null); // 标记节点值为null
    treeNode_01.setRuleKey("userGender"); // 创建决策规则：用户性别
    treeNode_01.setRuleDesc("用户性别[男/女]"); // 增添决策规则详细描述

    // 链接：1 -> 11
    TreeNodeLink treeNodeLink_11 = new TreeNodeLink();
    treeNodeLink_11.setNodeIdFrom(1L);
    treeNodeLink_11.setNodeIdTo(11L);
    treeNodeLink_11.setRuleLimitType(1);
    treeNodeLink_11.setRuleLimitValue("man");

    // 链接：1 -> 12
    TreeNodeLink treeNodeLink_12 = new TreeNodeLink();
    treeNodeLink_12.setNodeIdFrom(1L);
    treeNodeLink_12.setNodeIdTo(12L);
    treeNodeLink_12.setRuleLimitType(1);
    treeNodeLink_12.setRuleLimitValue("woman");

    // 将这两个逻辑链接加入到节点1的决策列表中
    List<TreeNodeLink> treeNodeLinkList_1 = new ArrayList<>();
    treeNodeLinkList_1.add(treeNodeLink_11);
```

```
        treeNodeLinkList_1.add(treeNodeLink_12);
        treeNode_01.setTreeNodeLinkList(treeNodeLinkList_1);

        // 节点：11
        TreeNode treeNode_11 = new TreeNode();
        treeNode_11.setTreeId(10001L);
        treeNode_11.setTreeNodeId(11L);
        treeNode_11.setNodeType(1);
        treeNode_11.setNodeValue(null);
        treeNode_11.setRuleKey("userAge");
        treeNode_11.setRuleDesc("用户年龄");

        // 链接：11 -> 111
        TreeNodeLink treeNodeLink_111 = new TreeNodeLink();
        treeNodeLink_111.setNodeIdFrom(11L);
        treeNodeLink_111.setNodeIdTo(111L);
        treeNodeLink_111.setRuleLimitType(3); // <
        treeNodeLink_111.setRuleLimitValue("25"); // [0, 25)岁的人群

        // 链接：11 -> 112
        TreeNodeLink treeNodeLink_112 = new TreeNodeLink();
        treeNodeLink_112.setNodeIdFrom(11L);
        treeNodeLink_112.setNodeIdTo(112L);
        treeNodeLink_112.setRuleLimitType(5); // <=
        treeNodeLink_112.setRuleLimitValue("25"); // [25, 100]岁人群

        // 将这两个逻辑链接加入到节点11的决策列表中
        List<TreeNodeLink> treeNodeLinkList_11 = new ArrayList<>();
        treeNodeLinkList_11.add(treeNodeLink_111);
        treeNodeLinkList_11.add(treeNodeLink_112);
        treeNode_11.setTreeNodeLinkList(treeNodeLinkList_11);

        // 节点：12
        TreeNode treeNode_12 = new TreeNode();
        treeNode_12.setTreeId(10001L);
        treeNode_12.setTreeNodeId(12L);
        treeNode_12.setNodeType(1);
        treeNode_12.setNodeValue(null);
        treeNode_12.setRuleKey("userAge");
        treeNode_12.setRuleDesc("用户年龄");

        // 链接：12->121
        TreeNodeLink treeNodeLink_121 = new TreeNodeLink();
        treeNodeLink_121.setNodeIdFrom(12L);
        treeNodeLink_121.setNodeIdTo(121L);
        treeNodeLink_121.setRuleLimitType(3);
        treeNodeLink_121.setRuleLimitValue("35"); // 年龄 < 35

        // 链接：12->122
        TreeNodeLink treeNodeLink_122 = new TreeNodeLink();
        treeNodeLink_122.setNodeIdFrom(12L);
        treeNodeLink_122.setNodeIdTo(122L);
        treeNodeLink_122.setRuleLimitType(5);
        treeNodeLink_122.setRuleLimitValue("35"); // 年龄 >= 35

        // 将这两个逻辑链接加入到节点12的决策列表中
        List<TreeNodeLink> treeNodeLinkList_12 = new ArrayList<>();
        treeNodeLinkList_12.add(treeNodeLink_121);
```

```
        treeNodeLinkList_12.add(treeNodeLink_122);
        treeNode_12.setTreeNodeLinkList(treeNodeLinkList_12);

        // 节点：111 (果实节点)
        TreeNode treeNode_111 = new TreeNode();
        treeNode_111.setTreeId(10001L);
        treeNode_111.setTreeNodeId(111L);
        treeNode_111.setNodeType(2);
        treeNode_111.setNodeValue("果实A: 发送二次元内容 -> 男");

        // 节点：112 (果实节点)
        TreeNode treeNode_112 = new TreeNode();
        treeNode_112.setTreeId(10001L);
        treeNode_112.setTreeNodeId(112L);
        treeNode_112.setNodeType(2);
        treeNode_112.setNodeValue("果实B: 发送财经类内容 -> 男");

        // 节点：121
        TreeNode treeNode_121 = new TreeNode();
        treeNode_121.setTreeId(10001L);
        treeNode_121.setTreeNodeId(121L);
        treeNode_121.setNodeType(2);
        treeNode_121.setNodeValue("果实C: 发送美妆内容 -> 女");

        // 节点：122
        TreeNode treeNode_122 = new TreeNode();
        treeNode_122.setTreeId(10001L);
        treeNode_122.setTreeNodeId(122L);
        treeNode_122.setNodeType(2);
        treeNode_122.setNodeValue("果实D: 发送育儿内容 -> 女");

        // 创建树根
        TreeRoot treeRoot = new TreeRoot();
        treeRoot.setTreeId(10001L);
        treeRoot.setTreeRootNodeId(1L);
        treeRoot.setTreeName("规则决策树BanTanger");

        Map<Long, TreeNode> treeNodeMap = new HashMap<>();
        treeNodeMap.put(1L, treeNode_01);
        treeNodeMap.put(11L, treeNode_11);
        treeNodeMap.put(12L, treeNode_12);
        treeNodeMap.put(111L, treeNode_111);
        treeNodeMap.put(112L, treeNode_112);
        treeNodeMap.put(121L, treeNode_121);
        treeNodeMap.put(122L, treeNode_122);

        // 聚合树根和树节点
        treeRich = new TreeRich(treeRoot, treeNodeMap);
    }
```

决策逻辑部分

好的，现在树型结构已经创建完毕了，下面开始书写决策逻辑部分

树节点逻辑过滤器接口LogicFilter.java

```
public interface LogicFilter {

    /**
     * 逻辑决策器
     * @param matterValue            决策值
     * @param treeNodeLineInfoList   决策节点
     * @return 下一个节点Id
     */
    Long filter(String matterValue, List<TreeNodeLink> treeNodeLineInfoList);

    /**
     * 获取当前节点的决策值
     * @param treeId
     * @param userId
     * @param decisionMatter 决策物料
     * @return 决策值
     */
    String matterValue(Long treeId, String userId, Map<String, String> decisionMatter);

}
```

定义统一了适配的通用接口，逻辑决策器，获取决策值，每一个提供决策能力的节点（非果实节点）都必须实现此接口，保证统一性，在上文提到的初始化节点里表现就是

```
treeNode_01.setNodeValue(null); // 标记节点值为null
treeNode_01.setRuleKey("userGender"); // 创建决策规则: 用户性别
treeNode_01.setRuleDesc("用户性别[男/女]"); // 增添决策规则详细描述
```

决策抽象类提供基础服务BaseLogic.java

```
public abstract class BaseLogic implements LogicFilter {

    @Override
    public Long filter(String matterValue, List<TreeNodeLink> treeNodeLinkList) {
        for (TreeNodeLink nodeLine : treeNodeLinkList) {
            if (decisionLogic(matterValue, nodeLine)) return nodeLine.getNodeIdTo();
        }
        return 0L;
    }

    @Override
    public abstract String matterValue(Long treeId, String userId, Map<String, String> decisionMatter);

```

```
    private boolean decisionLogic(String matterValue, TreeNodeLink nodeLink) {
        switch (nodeLink.getRuleLimitType()) {
            case 1:
                return matterValue.equals(nodeLink.getRuleLimitValue());
            case 2:
                return Double.parseDouble(matterValue) > Double.parseDouble(nodeLink.getRuleLimitValue());
            case 3:
                return Double.parseDouble(matterValue) < Double.parseDouble(nodeLink.getRuleLimitValue());
            case 4:
                return Double.parseDouble(matterValue) <= Double.parseDouble(nodeLink.getRuleLimitValue());
            case 5:
                return Double.parseDouble(matterValue) >= Double.parseDouble(nodeLink.getRuleLimitValue());
            default:
                return false;
        }
    }
}
```

这里实现基本决策方法：`1、2、3、4、5`，`等于、小于、大于、小于等于、大于等于`的判断逻辑。

同时matterValue方法(获取决策值)应该是放在具体节点逻辑中实现的，所以这里将baseLogic定义成抽象类，将matterValue设置成抽象接口方法交给下一层实现

树节点逻辑实现类(年龄节点和性别节点)

年龄节点

```
public class UserAgeFilter extends BaseLogic {

    @Override
    public String matterValue(Long treeId, String userId, Map<String, String> decisionMatter) {
        return decisionMatter.get("age");
    }

}
```

性别节点

```
public class UserGenderFilter extends BaseLogic {

    @Override
    public String matterValue(Long treeId, String userId, Map<String, String> decisionMatter) {
        return decisionMatter.get("gender");
    }

}
```

- 以上两个决策逻辑的节点获取值的方式都非常简单，只是获取用户的入参即可。实际的业务开发可以**从数据库、RPC接口、缓存运算**等各种方式获取。

决策引擎部分

决策引擎接口定义（统一接口）

```
public interface IEngine {

    EngineResult process(
            final Long treeId,
            final String userId,
            TreeRich treeRich,
            final Map<String, String> decisionMatter
    );

}
```

- 对于使用方来说也同样需要定义统一的接口操作，这样的好处非常方便后续拓展出不同类型的决策引擎，也就是可以建造不同的决策工厂。

决策节点配置

```
public class EngineConfig {

    /**
     * 对于这样的map接口，可以抽取到数据库中，就可以非常方便管理
     */
    static Map<String, LogicFilter> logicFilterMap; // 将所有接口决策节点配置到map结构中

    static {
        logicFilterMap = new ConcurrentHashMap<>(); // 使用ConcurrentHashMap保证线程安全，本质是靠分段锁保证安全
        logicFilterMap.put("userAge", new UserAgeFilter());
        logicFilterMap.put("userGender", new UserGenderFilter());
    }

    public Map<String, LogicFilter> getLogicFilterMap() {
        return logicFilterMap; // 单例模式，饿汉式
    }

    public void setLogicFilterMap(Map<String, LogicFilter> logicFilterMap){
        this.logicFilterMap = logicFilterMap;
    }

}
```

- 在这里将可提供服务的决策节点配置到`map`结构中，对于这样的`map`结构可以抽取到数据库中，那么就可以非常方便的管理。

基础决策引擎功能（遍历决策树）

```
public abstract class EngineBase extends EngineConfig implements IEngine {

    private Logger logger = LoggerFactory.getLogger(EngineBase.class);

    @Override
    public abstract EngineResult process(Long treeId, String userId, TreeRich treeRich, Map<String, String> decisionMatter);

    protected TreeNode engineDecisionMaker(
            TreeRich treeRich, Long treeId, String userId, Map<String, String> decisionMatter) {
        TreeRoot treeRoot = treeRich.getTreeRoot();
        Map<Long, TreeNode> treeNodeMap = treeRich.getTreeNodeMap();
        // 规则树根ID
        Long treeRootNodeId = treeRoot.getTreeRootNodeId();
        TreeNode treeNodeInfo = treeNodeMap.get(treeRootNodeId);
        // 判断节点类型[NodeType]：1 子叶， 2 果实
        while(treeNodeInfo.getNodeType().equals(1)) {
            String ruleKey = treeNodeInfo.getRuleKey();
            LogicFilter logicFilter = logicFilterMap.get(ruleKey);
            String matterValue = logicFilter.matterValue(treeId, userId, decisionMatter); // 获得决策值
            Long nextNode = logicFilter.filter(matterValue, treeNodeInfo.getTreeNodeLinkList()); // 根据决策值确定下一节点走向
            treeNodeInfo = treeNodeMap.get(nextNode); // 节点移动
            logger.info("决策树引擎=>{} userId: {} treeId: {} treeNode: {} ruleKey: {} matterValue: {}",
                    treeRoot.getTreeName(), userId, treeId, treeNodeInfo.getTreeNodeId(), ruleKey, matterValue);
        }
        return treeNodeInfo;
    }
}
```

- 这里主要提供决策树流程的处理过程，有点像通过链路的关系(`性别`、`年龄`)在二叉树中寻找果实节点的过程。
- 同时提供一个抽象方法，执行决策流程的方法供外部去做具体的实现。

决策引擎的实现

```java
public class TreeEngineHandle extends EngineBase {

    @Override
    public EngineResult process(Long treeId, String userId, TreeRich treeRich, Map<String, String> decisionMatter) {
        // 决策流程
        TreeNode treeNode = engineDecisionMaker(treeRich, treeId, userId, decisionMatter);
        // 决策结果
        return new EngineResult(userId, treeId, treeNode.getTreeNodeId(), treeNode.getNodeValue());
    }

}
```

- 这里对于决策引擎的实现就非常简单了，通过传递进来的必要信息；决策树信息、决策物料值，来做具体的树形结构决策。

测试验证

组装树的关系

为了体现基础类的设计思想放在前面去了，可以自行观看

- **重要**，这一部分是组合模式非常重要的使用，在我们已经建造好的决策树关系下，可以创建出树的各个节点，以及对节点间使用链路进行串联。
- 及时后续你需要做任何业务的扩展都可以在里面添加相应的节点，并做动态化的配置。
- 关于这部分手动组合的方式可以提取到数据库中，那么也就可以**扩展到图形界面**的进行配置操作。

测试类编写

```
@Test
public void test_tree() {
    logger.info("决策树组合结构信息：\r\n" + JSON.toJSONString(treeRich));

    IEngine treeEngineHandle = new TreeEngineHandle();
    Map<String, String> decisionMatter = new HashMap<>();
    decisionMatter.put("gender", "man");
    decisionMatter.put("age", "29");

    EngineResult result = treeEngineHandle.process(10001L, "Oli09pLkdjh", treeRich, decisionMatter);

    logger.info("测试结果：{}", JSON.toJSONString(result));
}
```

- 在这里提供了调用的通过组织模式创建出来的流程决策树，调用的时候传入了决策树的ID，那么如果是业务开发中就可以方便的解耦决策树与业务的绑定关系，按需传入决策树ID即可。
- 此外入参我们还提供了需要处理；`男`(man)、`年龄`(29岁)，的参数信息。

测试结果

```
23:35:05.711 [main] INFO  o.i.d.d.d.service.engine.EngineBase - 决策树引擎=>规则决策树 userId：Oli09pLkdjh treeId：10001 treeNode：11 ruleKey：userGender matterValue：man
23:35:05.712 [main] INFO  o.i.d.d.d.service.engine.EngineBase - 决策树引擎=>规则决策树 userId：Oli09pLkdjh treeId：10001 treeNode：112 ruleKey：userAge matterValue：29
23:35:05.715 [main] INFO  org.itstack.demo.design.test.ApiTest - 测试结果：{"nodeId":112,"nodeValue":"果实B：发送财经类内容 -> 男","success":true,"treeId":10001,"userId":"Oli09pLkdjh"}

Process finished with exit code 0
```

- 从测试结果上看这与我们使用`ifelse`是一样的，但是目前这与的组合模式设计下，就非常方便后续的拓展和修改。

之后想要继续细化粒度，就可以直接再创建接口实现类继承LogicFilter，同时实现子集的决策规则

也不必使用if-else这种丑陋的设计了

总结

- 从以上的决策树场景来看，组合模式的主要解决的是一系列简单逻辑节点或者扩展的复杂逻辑节点在不同结构的组织下，对于外部的调用是仍然可以非常简单的。
- 这部分设计模式保证了开闭原则，无需更改模型结构你就可以提供新的逻辑节点的使用并配合组织出新的关系树。但如果是一些功能差异化非常大的接口进行包装就会变得比较困难，但也不是不能很好的处理，只不过需要做一些适配和特定化的开发。
- 很多时候因为你的极致追求和稍有倔强的工匠精神，即使在面对同样的业务需求，你能完成出最好的代码结构和最易于扩展的技术架构。`不要被远不能给你指导提升能力的影响到放弃自己的追求！`

参考资料：

为什么使用final作为入参

初学 Java 设计模式（九）：实战组合模式「决策树实现精准化运营」

深入设计模式

小博哥重学设计模式