工作流分享

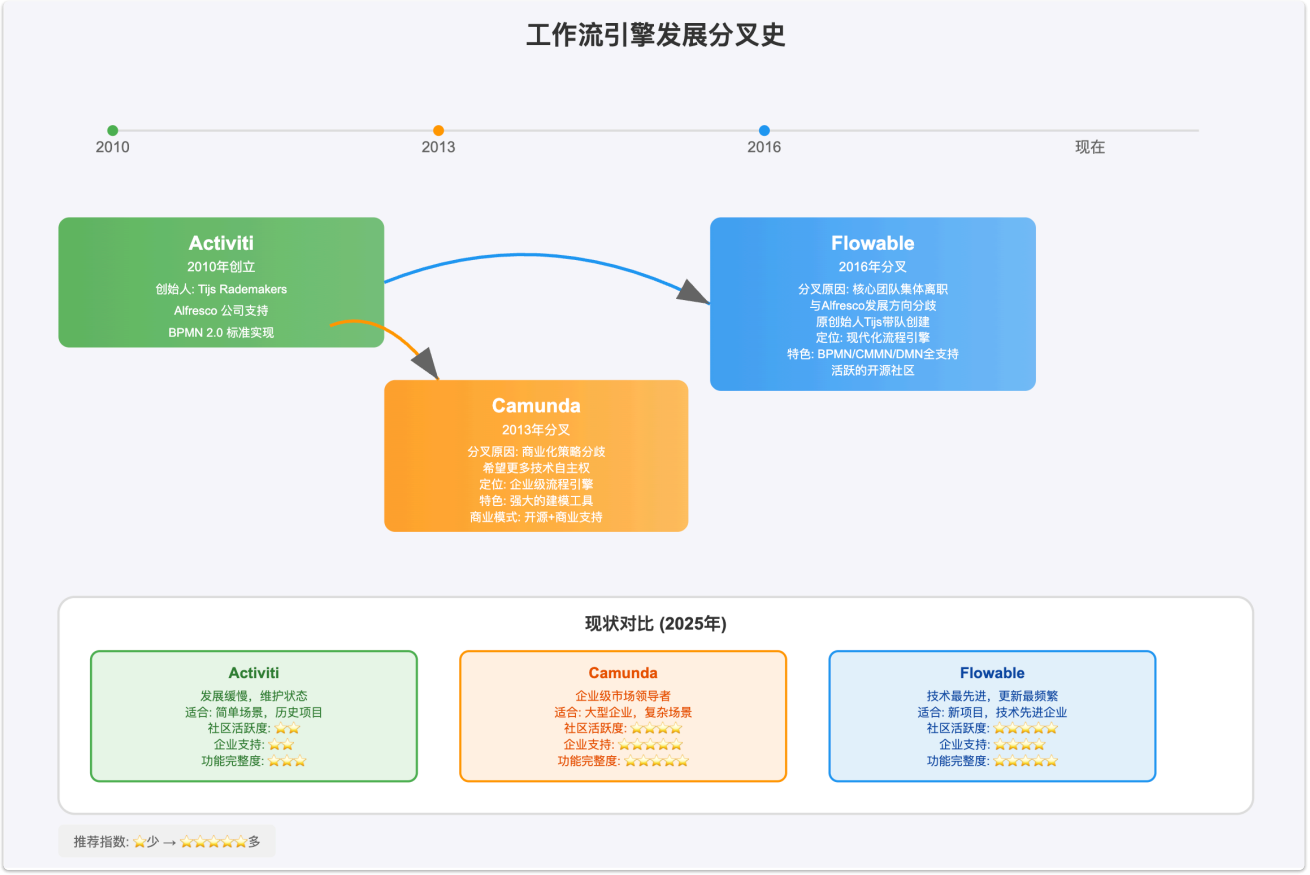
本文的主要的目的是分析如何基于当前的环境选择工作流框架以及版本。

第一部分是关于工作流的调研，第二部分是以前的实践经验。

提起工作流来说，比较流行的工作流有三种，Activiti，Camunda,以及 Flowable，为什么会有这三者呢？ 如何选择哪一个工作流与当前的环境，最匹配呢？

三者同根同源，均为 Activiti 分叉而来. 具体分叉的原因和三者发展方向，我们可以看以下内容。

1. 从分叉，看流派



🔥 Camunda 分叉 (2013 年)

团队构成：

- **核心人物**：Jakob Freund (CEO) 和 Bernd Ruecker (CTO) Camunda - 2025 Company Profile & Team
- **团队背景**：原本是 Activiti 的重要贡献者和咨询合作伙伴，约 20 人的团队 InfoQCamunda
- **角色定位**：商业化导向的技术团队

分叉原因：

主要分歧在于：

1. 技术方向和商业化策略不同
2. 应用服务器支持范围的分歧
3. 业务/IT 协调的理念差异
4. 希望吸引更多业务分析师参与社区 Camunda Forks Alfresco Activiti - InfoQ

特点和定位：

- **企业级导向**：专注大型企业市场
- **商业化成功**：2018 年获得 2500 万欧元 A 轮，2021 年获得 8200 万欧元 B 轮融资 Camunda - Wikipedia
- **技术创新**：在流程实例修改、流程实例迁移等核心功能方面领先 Camunda Engine Evolution since Activiti Fork | Camunda

💫 Flowable 分叉 (2016 年)

团队构成：

- **核心人物**：Tijs Rademakers (Activiti 原创始人和项目负责人) 和 Joram Barrez (Activiti 联合创始人和核心开发者) THE ORGECM Architect
- **团队背景**：Activiti 项目的 " 心脏和灵魂 "，是最核心的技术团队 Tijs Rademakers - VP Of Engineering at Flowable | The Org
- **角色定位**：开源技术纯粹主义者

分叉原因：

具体原因包括：

1. 与 Alfresco 在发展方向上的根本分歧
2. 感觉与自己创造的软件失去了连接
3. 担心项目失去应有的技术管理
4. 无法达成继续合作的协议 Tijs Rademakers - VP Of Engineering at Flowable | The Org

特点和定位：

- **开源纯粹性**：回归开源根本，采用精英管理制度 Tijs Rademakers - VP Of Engineering at Flowable | The Org
- **技术传承性**：由原始创造者领导，技术延续性最好
- **完全兼容**：设计为 Activiti 的直接替代品，就像 MariaDB 之于 MySQL Tijs Rademakers - VP Of Engineering at Flowable | The Org

关键差异：

维度	Camunda 团队	Flowable 团队
出身背景	外部合作伙伴和贡献者	Activiti 原始创造者
分叉时机	较早期 (2013 年)	核心团队集体离职后 (2016 年)
商业重点	企业级工具和服务	开源社区和技术创新
技术路线	重写核心引擎架构	继承和增强原有架构
市场策略	B2B 企业级销售	开源社区 + 商业服务
融资情况	大额风投融资	相对保守的商业化

简言之，两者分叉的原因都是因为方向和原公司不符，一个注重于业务与商业能力，一个注重于于技术与开源能力。但两者都对于 Activiti 有兼容。

2. 选型建议

🌱 Activiti

维护模式

版本	Java 版本	Spring Boot 版本	发布时间	状态说明
Activiti 5.x	Java 7+	Spring 3.x/4.x	2010-2016	已停止维护，不建议新项目使用
Activiti 6.x	Java 8+	Spring Boot 1.x/2.x	2016-2018	官方已停止维护，核心团队已离职
Activiti 7.x	Java 8+ / 11+	Spring Boot 2.x+	2017- 至今	云原生架构，但发展缓慢，强制集成 Spring Security
Activiti 8.x	Java 11+ / 17+	Spring Boot 3.x	2024- 至今	最新版本，支持 Spring Boot 3.x，但社区活跃度低

💫 Flowable

活跃开发

版本	Java 版本	Spring Boot 版本	发布时间	状态说明
Flowable 6.0-6.7	Java 8+	Spring Boot 2.x	2016-2022	稳定版本，仍可使用但建议升级
Flowable 6.8.x	Java 8+ / 11+	Spring Boot 2.6.x - 2.7.x	2022-2024	推荐生产使用，功能完善，性能稳定
Flowable 7.x	Java 17+	Spring Boot 3.x	2024- 至今	最新架构，包名变更为 com.flowable，需 JDK17+

🔥 Camunda

版本演进中

版本系列	Java 版本	Spring Boot 版本	支持状态	说明
Camunda 7.13-7.17	Java 8+ / 11+	Spring Boot 2.x	维护中	企业级功能丰富，但仅支持到 Spring Boot 2.x
Camunda 7.18-7.22	Java 8+ / 11+ / 17+	Spring Boot 2.7.x	当前稳定版	企业级首选，但不支持 Spring Boot 3.x
Camunda 7.23-7.24 LTS	Java 11+ / 17+ / 21+	Spring Boot 3.x	最新版本	7.24 为最后一个功能版本，2027 年支持到期
Camunda 8.x (Zeebe)	Java 11+ / 17+ / 21+	Spring Boot 3.x	云原生版本	基于 Zeebe 引擎，云原生架构，与 7.x 不兼容

选型建议总结

最佳选择: `Java 17+ + Spring Boot 3.x + Flowable 7.x` (技术最先进)
企业级: `Java 17+ + Spring Boot 3.x + Camunda 7.24 LTS` (商业支持)
云原生: `Java 17+ + Spring Boot 3.x + Camunda 8.x` (微服务编排)

场景	首选方案	备选方案	不推荐	特别说明
🎯 Java 8 项目	Flowable 6.8.x	Camunda 7.18-7.22	Activiti (任何版本)	Java 8 兼容性最佳
☕ Java 11+ & Spring Boot 2.x	Flowable 6.8.x	Camunda 7.22	-	稳定的组合
☕ Java 11+ & Spring Boot 3.x	Flowable 7.x	Camunda 7.23+	-	新技术栈适配
🚀 Java 17+ 技术领先	Flowable 7.x	-	-	必须 Spring Boot 3.x
🏢 Java 17+ 企业级	Camunda 7.24 LTS	-	-	商业支持
🌐 Java 17+ 微服务	Camunda 8.x	-	-	云原生架构
☁️ 云原生 (任意 Java 版本)	Camunda 8.x (Zeebe)	-	-	分布式微服务编排

3. 实践方案

Activiti 和 Flowable 主要使用的模块如下：

- RepositoryService: 流程定义管理，负责流程定义的存储、部署、查询和管理。 关联表格： `ACT_RE_*`

- RuntimeService: 流程实例管理，管理正在运行的流程实例，相当于流程的＂执行中心＂关联表格：`ACT_RU_*`
- TaskService: 任务管理，管理用户任务的整个生命周期，相当于任务的＂工作台＂关联表格 `ACT_RU_*`
- HistoryService: 历史数据查询，提供已完成流程的历史数据查询，相当于流程的＂档案馆＂关联表格: `ACT_HI_*`

不过，根据我的过往经验，用户相关的和历史相关的流程，一般会以单独的业务表格来做存储，比如会创建 `template_basic_info` ，以及 `template_manager` ，`template_submit_permission` 等表格，用来记录模板的基本信息以及模板的管理人员，提交人员。 以及会使用创建的 `workflow_history_instance` 以及 `workflow_history_task` 来简化历史业务查询。而且工作流任务，可以在执行这个任务之前配置对应的审批用户，因此在外部系统或者独立的表格维护用户系统即可。这样可以降低耦合性。

相关配置如下，字段的其他情况，请查看附录:

```
activiti:
  database-schema-update: false  # 不自动更新数据库
  history-level: none  # 不记录历史数据
  check-process-definitions: false  # 不检查流程定义
  deployment-mode: never-fail # 部署永不失败
```

而且，使用工作流框架的时候，选择符合当前项目的工作流系统，并可对工作流的工作做一定的裁剪，而且不同的工作流系统新增的功能不一样。比如增加的 DMN（Decision Model and Notation， 决策表系统，比如 VIP 用户，购买物品超过 1000,打 8 折， 普通用户，超过 1000,打 9 折，普通用户不超过 1000,打 95 折）可以用来相对应添加业务决策的功能。

附录:

工作流配置

1. database-schema-update

数据库架构更新策略

yaml

```yaml
database-schema-update: false  # 不自动更新数据库
```

可选值及含义:

值	说明	使用场景
false	不做任何数据库架构检查和更新	生产环境（手动管理数据库）
true	启动时检查并自动更新数据库架构	开发环境
create	启动时创建数据库表（如果不存在）	首次部署
create-drop	启动时创建表，关闭时删除表	单元测试
drop-create	启动时先删除旧表再创建新表	测试环境重置

java

```java
// 代码示例
ProcessEngineConfiguration config = ProcessEngineConfiguration
    .createStandaloneProcessEngineConfiguration()
    .setDatabaseSchemaUpdate("false");  // 生产环境设置
```

2. history-level

历史数据记录级别

yaml

```yaml
history-level: none  # 不记录历史数据
```

可选值及含义：

级别	说明	记录内容	性能影响
none	不记录历史	无	最高性能
activity	记录活动级别	流程实例、活动实例	中等性能
audit	审计级别（默认）	+ 任务、变量	较低性能
full	完整记录	+ 变量更新细节	最低性能

影响的表：

sql

```sql
-- history-level: none 时，这些表都是空的
SELECT * FROM ACT_HI_PROCINST;  -- 历史流程实例
SELECT * FROM ACT_HI_ACTINST;   -- 历史活动实例
SELECT * FROM ACT_HI_TASKINST;  -- 历史任务实例
SELECT * FROM ACT_HI_VARINST;   -- 历史变量
SELECT * FROM ACT_HI_DETAIL;    -- 历史详情
```

使用示例：

java

```java
// 根据不同环境设置不同级别
@Configuration
public class ActivitiConfig {

    @Value("${spring.profiles.active}")
    private String profile;

    @Bean
    public ProcessEngineConfiguration processEngineConfiguration() {
        ProcessEngineConfiguration config = ...;

        // 根据环境设置历史级别
        if ("prod".equals(profile)) {
            config.setHistoryLevel(HistoryLevel.NONE);  // 生产环境不记录
        } else if ("dev".equals(profile)) {
            config.setHistoryLevel(HistoryLevel.FULL);  // 开发环境全记录
        } else {
            config.setHistoryLevel(HistoryLevel.AUDIT); // 默认审计级别
        }

        return config;
    }
}
```

3. check-process-definitions

是否检查流程定义

yaml

```yaml
check-process-definitions: false  # 不检查流程定义
```

含义：

- true：启动时检查并验证所有流程定义的正确性
- false：跳过流程定义验证，加快启动速度

检查内容：

java

```java
// 当设置为 true 时，会检查：
// 1. BPMN XML 格式是否正确
// 2. 流程定义是否有语法错误
// 3. 服务任务的类是否存在
// 4. 表达式是否合法

// 示例：会被检查出的错误
<serviceTask id="task1"
    flowable:class="com.example.NotExistClass"/>  // 类不存在

<sequenceFlow sourceRef="task1"
    targetRef="notExistTask"/>  // 目标任务不存在
```

4. deployment-mode

部署模式

yaml

```yaml
deployment-mode: never-fail  # 部署永不失败
```

可选值：

值	说明	行为
default	默认模式	部署失败会抛出异常，应用启动失败
single-resource	单资源模式	每个资源单独部署，一个失败不影响其他
never-fail	永不失败模式	部署失败只记录日志，不影响应用启动

使用场景示例：

java

```java
@Component
public class ProcessDeployer {

    @PostConstruct
    public void deployProcesses() {
        try {
            // 部署多个流程定义
            repositoryService.createDeployment()
                .addClasspathResource("processes/process1.bpmn")  // 可能有错
```

```
                .addClasspathResource("processes/process2.bpmn")  // 正确的
                .addClasspathResource("processes/process3.bpmn")  // 可能有错
                .deploy();
        } catch (Exception e) {
            // deployment-mode: default - 这里会抛异常，应用启动失败
            // deployment-mode: never-fail - 只记录日志，继续启动
            log.error("流程部署失败: ", e);
        }
    }
}
```

决策表用例

传统代码:

```
public double getDiscount(String userType, double amount) {
    if ("VIP".equals(userType) && amount > 1000) {
        return 0.8;
    } else if ("NORMAL".equals(userType) && amount > 1000) {
        return 0.9;
    } else if ("NORMAL".equals(userType) && amount <= 1000) {
        return 0.95;
    }
    return 1.0;
}
```

DMN 决策表方式:

用户类型	购买金额	→ 折扣率
VIP	> 1000	0.8
NORMAL	> 1000	0.9
NORMAL	≤ 1000	0.95

DMN 代码:

```
@Service
public class DiscountService {

    @Autowired
    private DmnRuleService dmnRuleService;


    public double getDiscount(String userType, double amount) {
        Map<String, Object> variables = new HashMap<>();
        variables.put("userType", userType);
        variables.put("amount", amount);

        Map<String, Object> result = dmnRuleService
            .createExecuteDecisionBuilder()
            .decisionKey("discountDecision")
            .variables(variables)
            .execute();

        return (Double) result.get("discount");
    }
```

```
    public double getFinalPrice(String userType, double amount) {
        double discount = getDiscount(userType, amount);
        return amount * discount;
    }
}
```