更加详细的开发与测试周期表

下表为一个更加精细的开发计划示例，提供了每个阶段的**主要任务**、**详细子任务**、**测试策略**、以及**验收标准与时间预估**。此表格可用于更准确地进行项目排期与进度把控。

阶段	主要任务	详细子任务	测试策略	验收标准	时间预估
1. 环境准备	- 项目基础结构/依赖配置 - CI/CD 流水线搭建	1. 创建 Flutter 项目并配置 `pubspec.yaml` 2. 安装/配置 Riverpod、测试库等基础依赖 3. 在 GitHub/GitLab Actions 上配置 CI流水线 (自动化构建、测试) 4. 拉取/合并主分支，验证流水线无报错	- 手动执行 `flutter run` 以检查最基础的可运行性 - CI/CD 提交测试：仅**空项目**编译能否通过	- 项目能在本地/CI 环境成功运行 - 无关键报错或依赖缺失 - CI 构建流水线绿灯	**2~3天**
2. 数据与状态层	- Node/Edge/Anchor 的数据结构 - StateManagement初步完成	1. 设计 `NodeModel`, `EdgeModel`, `AnchorModel` 并编写文档注释 2. 使用 Riverpod/StateNotifier 管理节点、边、锚点列表 3. 提供初步的增删改方法(如 `addNode`, `deleteEdge`, `updateAnchor`) 4. 撰写数据层单元测试 (模型序列化、状态更新)	- **单元测试**： - Model 序列化反序列化验证 - Provider 状态更新验证 (add/delete是否正确)	- 节点、边、锚点数据结构清晰且可运行 - 对象间关系(如锚点属于节点)一致 - 单元测试全部通过	**3~4天**
3. 画布Canvas核心	- 画布渲染/交互框架 (平移、缩放) - InteractionStateMachine初步	1. 实现 `CanvasRenderer`：支持网格背景或基本画布 2. `CanvasController`：维护缩放(scale)、平移(offset)等属性 3. 编写 `canvas_interaction_manager.dart` 处理鼠标滚轮/拖拽空白 => 平移或缩放 4. 初始化 `InteractionStateMachine`(仅PanTool/ZoomTool)	- **Widget测试**： - 拖拽空白 => 检查offset 变化 - 滚轮滚动 => 检查scale 变化 - 测试InteractionState 切换(normal=>Pan=>zoom)	- 画布可基础缩放/平移且流畅 - 交互状态机初具雏形，能进行基本模式切换 - 测试用例绿灯通过，性能无明显卡顿	**4~5天**

阶段	主要任务	详细子任务	测试策略	验收标准	时间预估
4. 节点Node功能	- 节点渲染、NodeWidget - 节点拖拽、单/多选	1. `NodeRenderer`绘制节点形状/文字 2. `node_widget.dart` 实现节点UI (可拖拽等) 3. 结合 `SelectionManager`，完成单选/多选 4. 在 `node_interaction_manager.dart` 内处理节点拖拽和点击事件 5. 撰写节点测试 (拖拽后的坐标，选中状态的更新)	- **Widget测试**： - 模拟单节点点击 => isSelected? - 多选(Shift/ctrl) => selection里多个id? - 拖拽 => x,y 变化正确 - 贴近真实拖拽场景(Ctrl+拖等)	- 节点能在画布正确渲染 - 单/多选切换正常 - 节点拖拽顺畅，测试用例覆盖80%+交互场景	**5~6天**
5. 连线Edge功能	- 连线的绘制与锚点连接 - QuickConnectionPlugin初版	1. `EdgeRenderer` 绘制贝塞尔曲线/箭头 2. `edge_interaction_manager.dart` 处理锚点拖拽 => 连接Edge 3. **可选**：开发 `QuickConnectionPlugin` => 单击锚点快速连线 4. 写Edge单元测试(连线数据正确性)、Widget测试(拖拽锚点 => 成功创建边)	- **单元测试**： - EdgeModel(sourceNodeId, targetNodeId)更新 - QuickConnection调用时自动生成边 - **Widget测试**：实际拖锚点 => 看edgesProvider里是否多了一条connected edge	- 边能正确绘制与连接 - 快速连线功能可选启用 - 测试通过，错误场景(拖拽到空白)有保护处理	**4~5天**
6. 框选与键盘支持	- BoxSelection(Shift+拖空白) - 快捷键(删除、复制等)	1. `BoxSelectionHandler` => 在画布拖拽框选节点 2. `keyboard_event_dispatcher.dart` => 处理 Del、Ctrl+C、Ctrl+V 等快捷键 3. NodeCopyHandler、NodeEdgeDeletionHandler 调用 provider 执行复制/删除 4. 测试：框选节点后多选状态 + 快捷键删除	- **Widget测试**： - Shift+空白拖拽 => selection 里出现多个节点 - 按 Del => node/edge 被清除 - Ctrl+拖 => NodeCopy? 结果provider里出现复制节点	- 多选框选稳定 - 删除/复制操作无异常 - 键盘按键冲突场景有处理(如cmd/ctrl区分)	**3~4天**
7. 插件系统	- GraphPlugin接口完善 - 插件加载/卸载流程	1. 在 `graph_plugin.dart` 定义标准接口(onLoad, onUnload, onNodeTap...) 2. `plugin_manager.dart` 提供 registerPlugin / removePlugin 3. 调整已有功能(如QuickConnection/ContextMenu)迁移到插件机制 4. 测试：加载插件 => 触发插件逻辑；卸载插件 => 不再响应	- **集成测试**： - 同时加载多个插件 => 看事件是否冲突 - 卸载某插件 => 其功能立即失效 - 插件间优先级处理?(先后顺序)	- 插件可动态启用与禁用 - 无核心代码修改即可扩展功能 - 互斥插件能正常优先级处理	**4~5天**

阶段	主要任务	详细子任务	测试策略	验收标准	时间预估
8. Hooks机制	- Hooks的生命周期钩子 - Node/Edge/Canvas Hooks	1. `use_node_hook.dart` / `use_edge_hook.dart` / `use_canvas_hook.dart` 2. 增加`onNodeCreate`, `onEdgeConnect`等 3. 测试：编写一个自定义Hook => onNodeCreate时自动给node添加tag => 观察node是否有此tag	- **单元+集成测试**： - Hook触发时机是否正确(前/后) - 覆盖主要生命周期测试（node被删除时onNodeDelete?）	- Hook编写简单 - 测试Hook触发顺序无错 - 开发者可用此机制扩展业务	**3~4天**
9. 性能优化	- 空间索引(Quadtree) - 性能监控插件	1. 在 `geometry_utils` 中增加Quadtree加速碰撞检测 2. 开发 `performance_utils.dart` 监控帧率/渲染耗时 3. 测试：节点数量增至上千，观察是否卡顿	- **性能测试**： - 多节点场景下，帧率>50fps - DevTools CPU/内存Profile 检查无明显瓶颈	- 大数量场景依旧流畅 - Profiling无明显性能瓶颈	**3~4天**
10. 最终功能整合	- 全局回归测试 - 文档与示例编写	1. 整合前所有功能：节点/边/插件/框选/键盘等依次测试 2. 撰写`basic_workflow_example.dart`, `advanced_workflow_example.dart` 3. 编写文档：`getting_started.md`, `plugins_development_guide.md` 4. 人工演示 + 用户试用(内部Alpha)	- **回归测试**： - 全功能交互是否互相冲突 - 文档是否清晰、示例可否正常运行 - 可做手动冒烟测试(常用操作)	- 全部集成功能稳定运行 - 示例项目可成功展示常用交互 - 文档覆盖常见问题与开发流程	**4~5天**

详细说明

1. **基础阶段**：集中解决“项目初始化”、“数据结构和状态管理”问题，让后续开发更顺畅。
2. **中期阶段**：逐步完善画布、节点、边、锚点、选择、快捷键等核心交互功能，每个功能都写足够多的单元/Widget测试。
3. **后期阶段**：重点于**插件化**与**Hooks**，为长远扩展打基础；再进行**性能优化**，确保大量节点/边场景下流畅。
4. **最终整合**：进行**全局回归测试**与**文档/示例**完备，保证对内外部开发者都易上手。

小结

通过这样分期分批的开发和测试安排，可以保证你的工作流编辑器在**功能**、**性能**、**可扩展性**各方面稳步提升，并在每个阶段都能及早发现并解决问题。