NodeModel 文档

`NodeModel` 是用于定义工作流编辑器或绘图应用中节点的数据模型。
本模型设计为不可变（immutable），并与UI层解耦，通过提供序列化和反序列化方法便于持久化和网络传输。

一、NodeModel 基本信息

字段名	类型	说明
`id`	`String`	节点的唯一标识
`x`, `y`	`double`	节点左上角在画布上的坐标
`width`, `height`	`double`	节点尺寸（宽度、高度）
`dragMode`	`DragMode`	拖拽模式（`full` 整个节点拖拽 / `handle` 通过特定部位拖拽）
`type`	`String`	节点类型（例如："start", "end", "custom"）
`role`	`NodeRole`	节点角色（start, middle, end 等）
`title`	`String`	节点的标题（用于UI展示）
`anchors`	`List<AnchorModel>`	节点上可连接连线的锚点列表

二、扩展属性（可选）

字段名	类型	说明
`status`	`NodeStatus`	节点运行状态（none, running, completed, error）
`parentId`	`String?`	父节点ID（用于节点分组）
`zIndex`	`int`	节点的渲染层级（越大越靠上）
`enabled`	`bool`	节点是否启用
`locked`	`bool`	节点是否被锁定（锁定后无法编辑或拖拽）
`description`	`String?`	节点的描述信息（可选）
`style`	`NodeStyle?`	节点外观的纯数据定义

三、审计与版本信息（可选）

字段名	类型	说明
`version`	`int`	节点版本号（用于版本控制）

字段名	类型	说明
createdAt	DateTime?	节点创建的时间
updatedAt	DateTime?	节点最后更新的时间

四、扩展数据字段（自定义用途）

字段名	类型	说明
inputs	Map<String, dynamic>	存储节点的输入数据（自定义）
outputs	Map<String, dynamic>	存储节点的输出数据（自定义）
config	Map<String, dynamic>	节点配置项（自定义）
data	Map<String, dynamic>	存储业务相关的任意附加数据

五、核心方法说明

(1) rect

获取节点矩形区域（用于绘制或碰撞检测）：

```
Rect get rect => Rect.fromLTWH(x, y, width, height);

### (2) `getAnchorOffset`

获取指定锚点相对于节点中心的偏移量：

```dart
Offset getAnchorOffset(AnchorModel anchor) {
    final ratio = anchor.ratio.clamp(0.0, 1.0);
    final offset = anchor.position.getOffset(width, height);
    return Offset(x + offset.dx, y + offset.dy);
}
```
```

(3) copyWith

不可变更新方法，创建并返回新的节点实例：

```
final updatedNode = node.copyWith(
  x: 200,
  y: 300,
  status: NodeStatus.completed,
);
```

(4) `toJson` & `fromJson`

序列化和反序列化，支持网络传输或数据存储：

```
final json = node.toJson();
final node = NodeModel.fromJson(json);
```

六、常见使用场景示例

1. 节点位置更新

```
final newNode = node.copyWith(x: 300, y: 400);
```

2. 节点尺寸调整

```
final resizedNode = node.copyWith(width: 150, height: 100);
```

3. 节点状态更新

```
final updatedNode = node.copyWith(status: NodeStatus.completed);
```

4. 节点配置更新

```
final updatedNode = node.copyWith(config: {'key': 'value'});
```

5. 节点数据更新

```
final updatedNode = node.copyWith(data: {'key': 'value'});
```

6. 节点样式更新

```
final updatedNode = node.copyWith(style: NodeStyle(fillColor:
Colors.blue));
```

7. 节点配置更新

```
final updatedNode = node.copyWith(config: {'key': 'value'});
```

七、注意事项

- 所有字段定义为不可变（immutable），使用copyWith方法更新。
- 推荐将状态管理与节点模型解耦，使用单独的StateNotifier或其他状态管理工具维护节点列表。
- anchors、inputs、outputs、data 等动态数据更新时，必须重新创建对象，而非直接修改原有对象。

八、与其他模型的关系

1. 与AnchorModel的关系

AnchorModel 定义节点上的连接点，管理连接规则。

2. 与EdgeModel的关系

EdgeModel 定义节点之间的连接线，管理连接状态。

3. 与NodeStyle的关系

NodeStyle 定义节点的外观，与UI层解耦。

九、扩展与定制建议

- 扩展新字段时，可通过config或data字段直接加入，无需修改模型本身结构。
- 如果需要额外逻辑（如权限管理、多用户协作等），推荐通过插件或 Hook 机制扩展，而非直接在 NodeModel 内部实现。

十、枚举类型定义参考

```
enum DragMode { full, handle }

enum NodeRole { placeholder, start, middle, end, custom }

enum NodeStatus { none, running, completed, error }
```

十一、JSON 示例

序列化后的JSON示例

```
{
  "id": "node-123",
  "x": 150.0,
```

```
    "y": 200.0,
    "width": 100.0,
    "height": 60.0,
    "dragMode": "DragMode.full",
    "type": "custom",
    "role": "NodeRole.middle",
    "title": "Example Node",
    "anchors": [
      {
        "id": "anchor-1",
        "position": "left",
        "ratio": 0.5
      }
    ],
    "status": "NodeStatus.running",
    "zIndex": 1,
    "enabled": true,
    "locked": false,
    "description": "This is an example node",
    "style": {
      "fillColorHex": "#FF0000",
      "borderWidth": 2.0
    },
    "version": 3,
    "createdAt": 1712035800000,
    "updatedAt": 1712035900000,
    "inputs": {
      "key1": "value1"
    },
    "data": {
      "customKey": "customValue"
    }
  }
```

十二、单元测试建议

- 测试构造方法默认值与边界值
- 测试copyWith方法的不可变性
- 测试JSON序列化与反序列化是否一致
- 测试getAnchorOffset返回准确的坐标值

十三、总结

NodeModel以不可变的数据模型实现，便于维护和扩展。通过合理设计状态与数据的边界，能够很好地适用于复杂工作流或绘图应用。