

G6-Editor 文档

G6-Editor 1.0 文档仓库

目录

[前言（必读）](#)

[快速上手](#)

[Demos](#)

[API](#)

[Editor](#)

[Command](#)

[Flow](#)

[Koni](#)

[Toolbar](#)

[Detailpanel](#)

[Minimap](#)

[Contextmenu](#)

[Itempanel](#)

前言（必读）

概述

`G6-Editor` 是 2018 年 2 月份立项。其目前的主要使命是支持阿里内部的，高交互的图编辑业务。项目至今才发展了 4 个月，虽说开发时间不长，但图编辑器的场景可以说是从 G6 诞生的头一天就频频遇到，其中踩了多少坑大概只有做过编辑器的人才知道。我们从中总结经验，吸取教训，最终借着本次 G6 开源的机会，开放 G6-Editor 1.0，和大家学习、讨论。在使用 G6-Editor 之前以下几点希望大家务必仔细阅读：

1. 不开源，仅供学习交流使用，不得商用，不再接受申请。（2018.09.21 前申请了使用权限，并得到使用许可的仍可以根据许可证协议进行商用。）
2. 不足够的配置、接口
3. 不足够的 Demo 及 文档
4. 特殊的升级策略：
 - a. 末尾版本号，如 1.0.x 用于修复 Bug 是兼容性升级
 - b. 中版本号，如 1.x.0 用于小范围调整接口及配置，**是不兼容升级**
 - c. 头版本号，如 x.0.0 用于重大框架调整，**是不兼容升级**

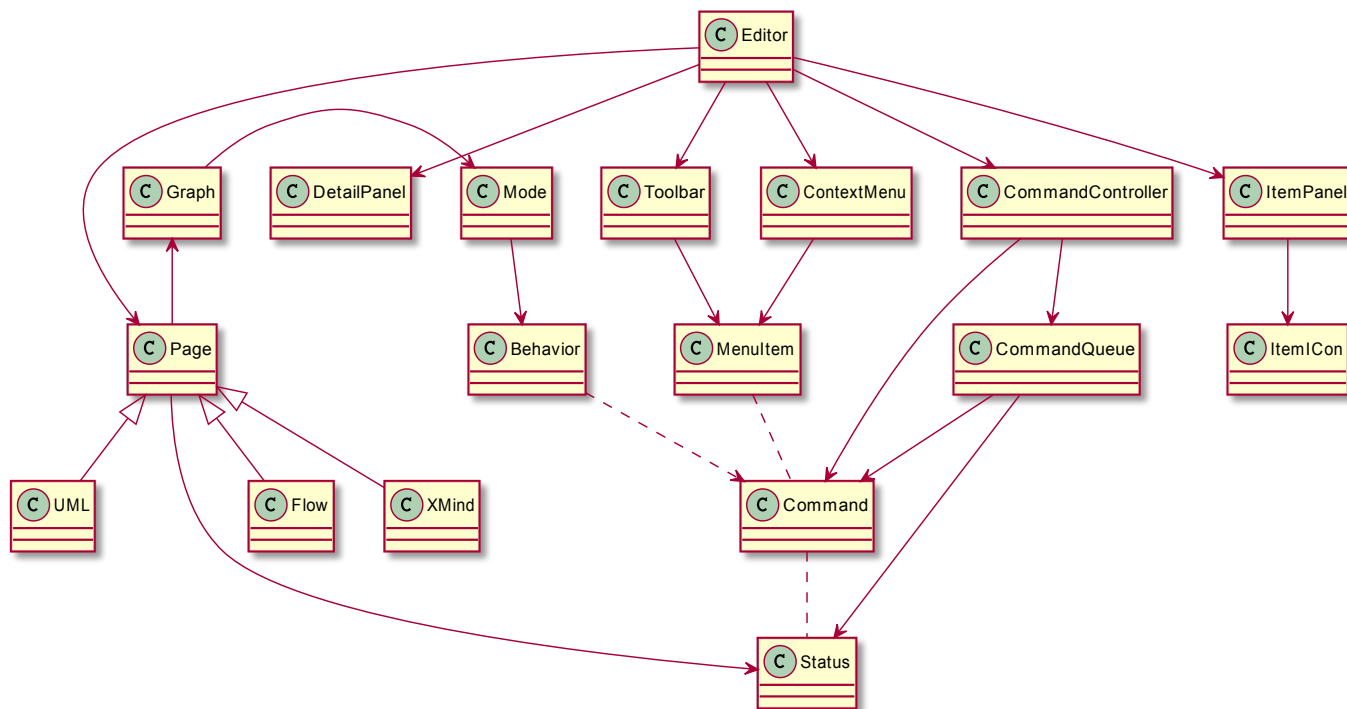
使用 G6-Editor 时 package.json 中应使用 `~` 作为依赖的前缀



浏览器支持

支持最新版的 Safari 及 Chrome。

整体框架



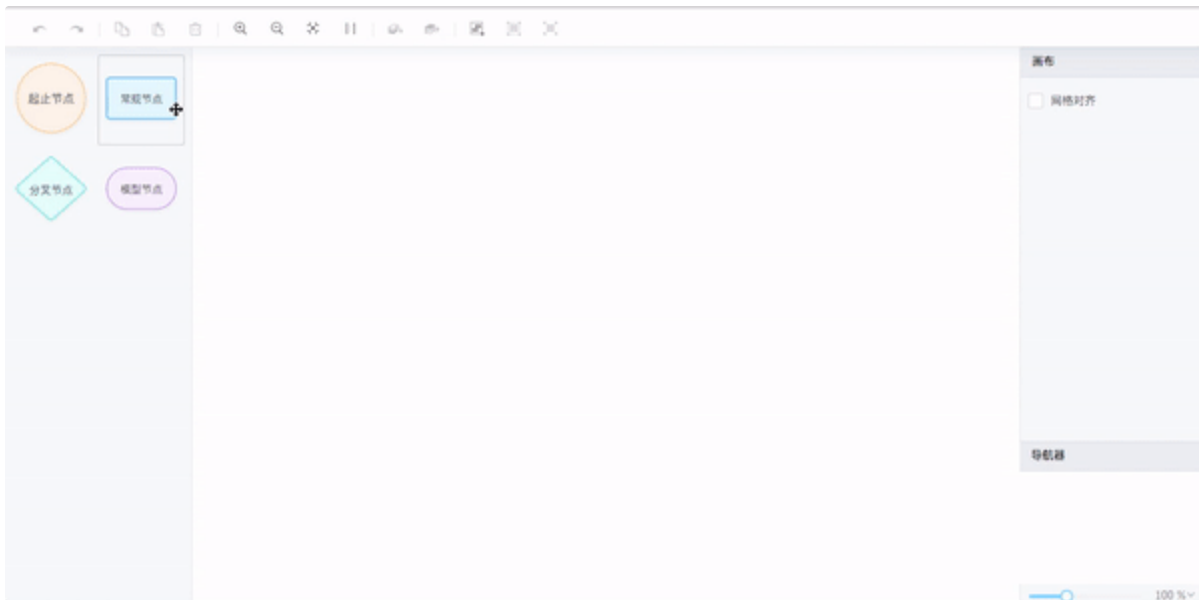
精心交互设计

图编辑器是一个非常复杂的人机交互的系统工程。打造一款体验优秀的前端图编辑器，不仅需要技术过硬，还需要融入优秀的交互设计。得益于体验技术部优质充足的设计储备，以及众多内部业务系统的共同磨砺，目前已经总结了两个完成度较高的图编辑器，分别是流图编辑器（Flow），脑图编辑器（Mind）。

交互设计录详见：[《G6-Editor-Flow 交互设计沉思录》](#)

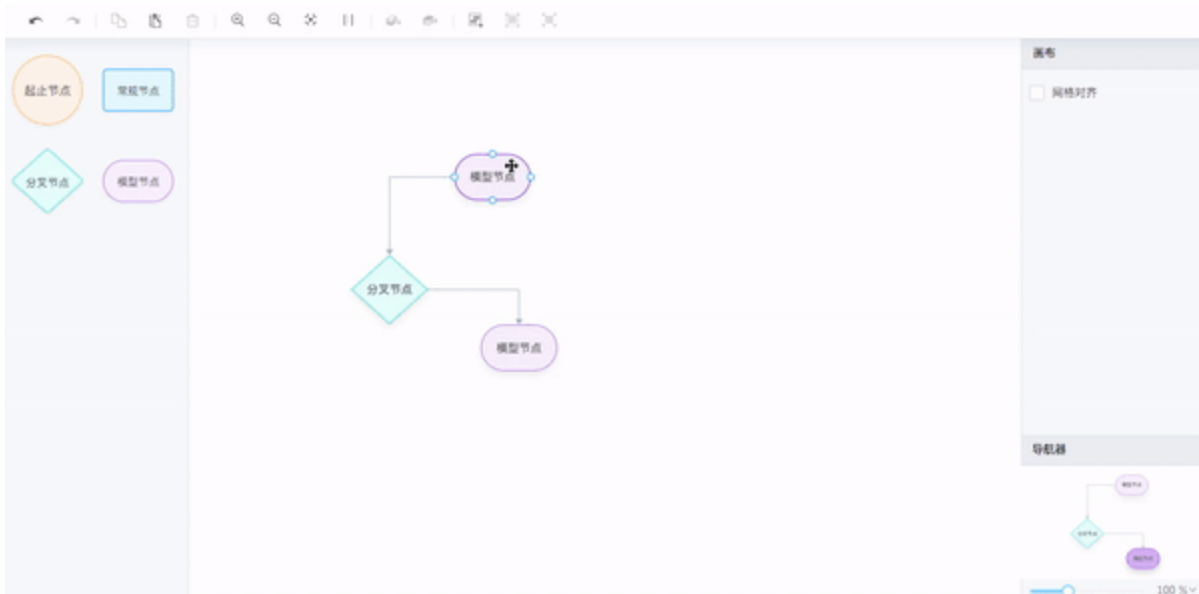
经典命令模式

图编辑器中有大量撤销重做、服务端指令通讯、甚至是多人协作的需求，在参考了传统软件架构后，发现在编辑器场景下定义一套命令模式，实现一套命令集几乎是**标配**。G6-Editor 也实现了一套自己的命令系统。得益于命令系统，G6-Editor 几乎所有的操作都是**可撤销、可重做的**。



贴心状态管理

在 G6-Editor 中我们精心定义并设计了页面的状态、命令的状态。G6-Editor 会帮你处理如：工具栏命令是否可用，不同图项面板栏的切换展示。在使用过程中，你只需要根据状态变更后，工具栏、右键、详情面板栏中相应变更的 dom class 属性作出相应响应，即可轻松实现。



地址

[文档链接](#)

[代码仓库](#)

体验改进计划说明

为了更好服务用户，G6Editor 会将 URL 和版本信息发送回 AntV 服务器：

```
https://kcart.alipay.com/web/bi.do
```

除了 URL 与 G6 版本信息外，不会收集任何其他信息。如有担心，可以通过下面的代码关闭：

```
// 关闭 G6Editor 的体验改进计划打点请求  
G6Editor.track(false)
```

快速上手

G6-Editor 是 AntV 官方提供的、专注于图可视化编辑器的类库，本文档将向大家介绍如何快速上手。

安装

```
npm i @antv/g6-editor --save
```

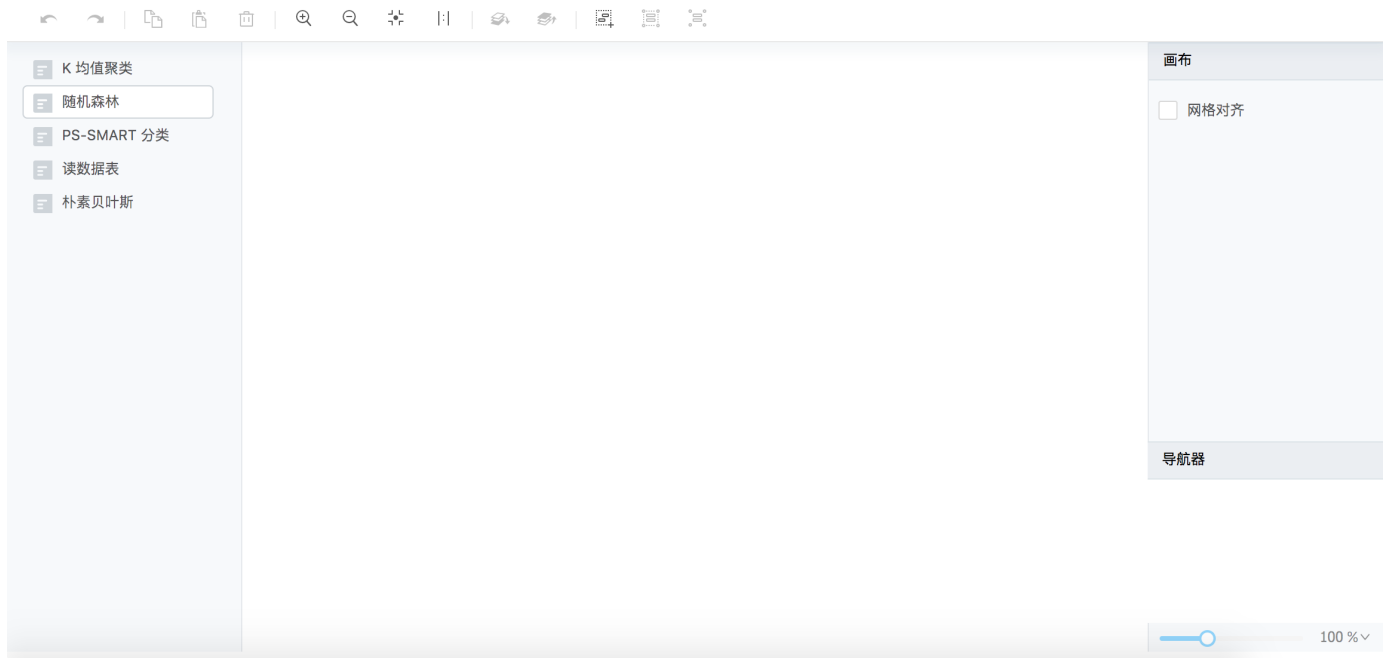
```
import G6Editor from '@antv/g6-editor';
```

快速流程图编辑器

第一步：根据约定生成 DOM 容器

```
<div id="minimap"></div>      <!-- 缩略图 DOM 结构规约参考 Minimap API -->
<div id="toolbar"></div>      <!-- 工具栏 DOM 结构规约参考 Toolbar API -->
<div id="itempanel"></div>    <!-- 元素面板栏 DOM 结构规约参考 Itempanel API -->
>
<div id="detailpanel"></div>  <!-- 详情面板栏 DOM 结构规约参考 Detailpanel API -->
-->
<div id="contextmenu"></div>  <!-- 右键菜单栏 DOM 结构规约参考 Contextmenu API -->
-->
<div id="page"></div>        <!-- 参考 Flow、Mind API -->
```

第二步：设置 UI 排版及样式



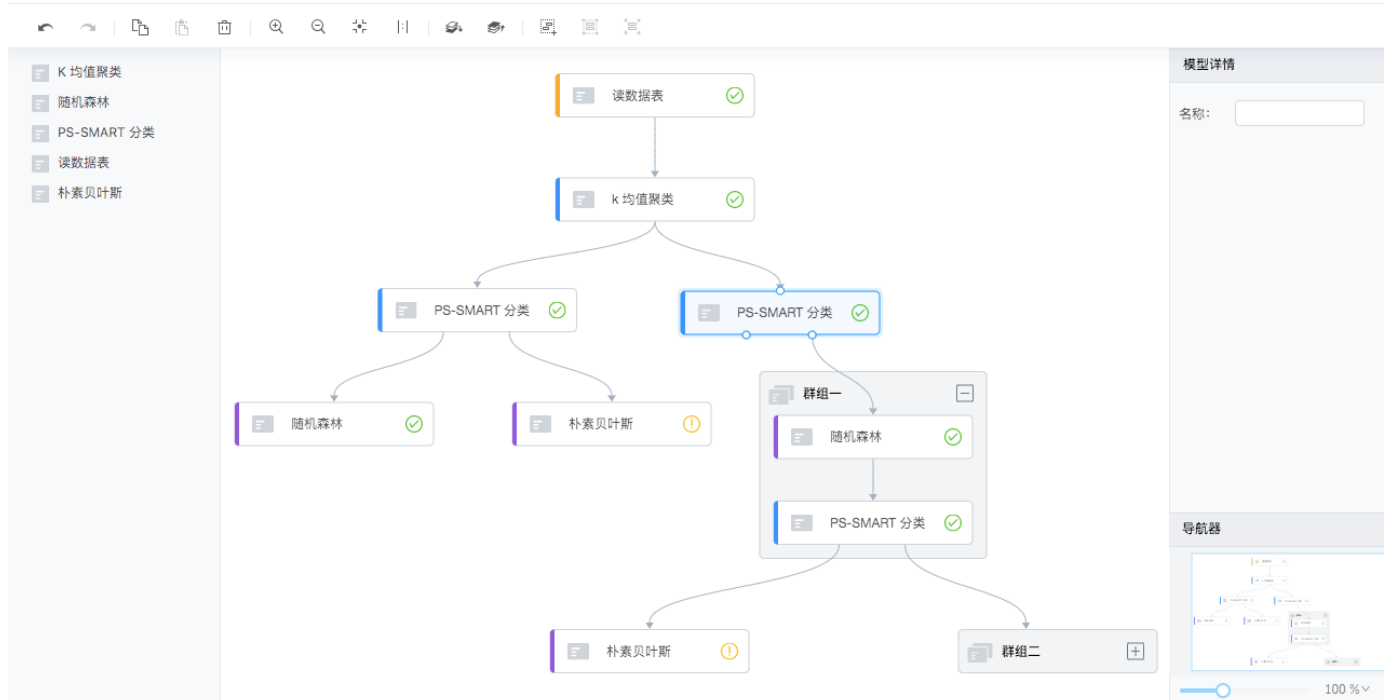
以上两步，配合一些成熟的 Web UI 框架（比如 AntD）就能快速搭建

第三步：编写 G6-Editor 脚本

```
const editor = new G6Editor();
const minimap = new G6Editor.Minimap({
  container: 'minimap',
});
const toolbar = new G6Editor.Toolbar({
  container: 'toolbar',
});
const contextmenu = new G6Editor.Contextmenu({
  container: 'contextmenu',
});
const itempanel = new G6Editor.Itempanel({
  container: 'itempanel',
});
const detailpanel = new G6Editor.Detailpanel({
  container: 'detailpanel',
});
const page = new G6Editor.Flow({
  graph: {
    container: 'page',
  },
});
editor.add(minimap);
```

```
editor.add(toolbar);
editor.add(contextmenu);
editor.add(itempanel);
editor.add(detailpanel);
editor.add(page);
```

搞定!

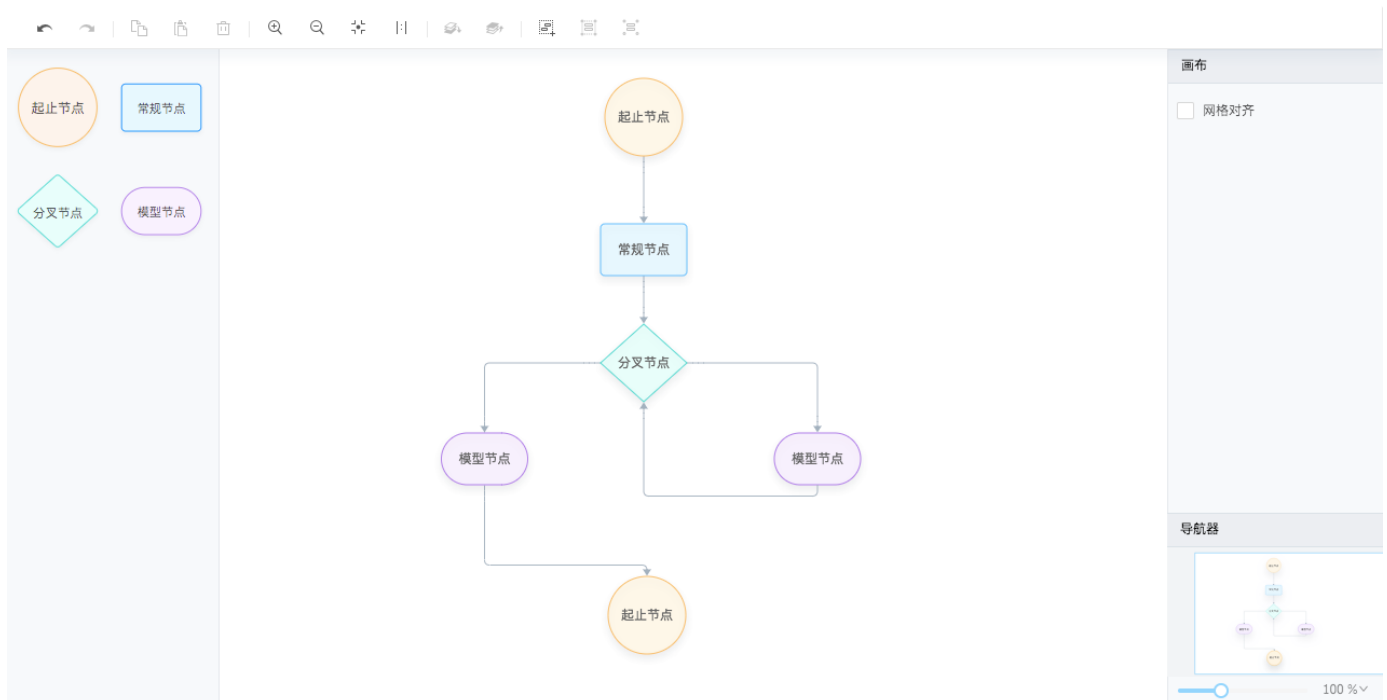


线上演示地
代码仓库地址

Demos

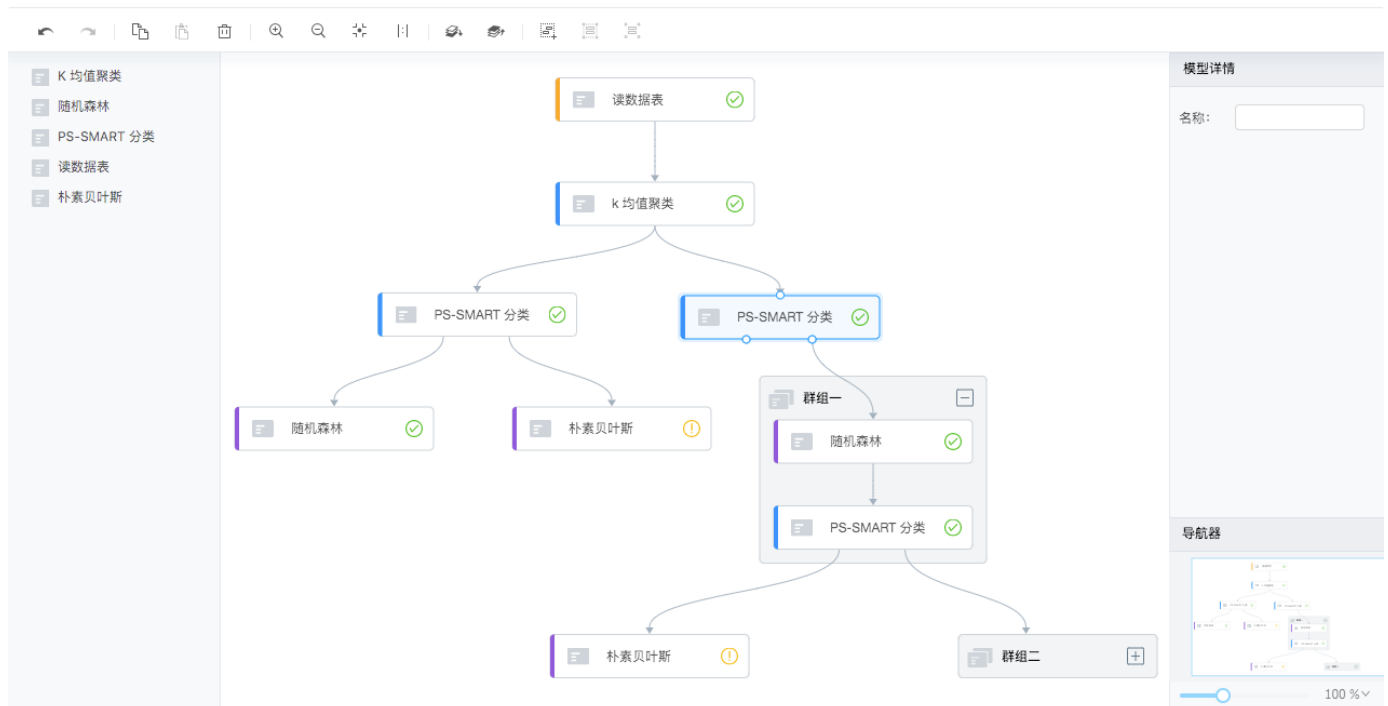
基础流程图

- [线上演示地址](#)
- [代码地址](#)



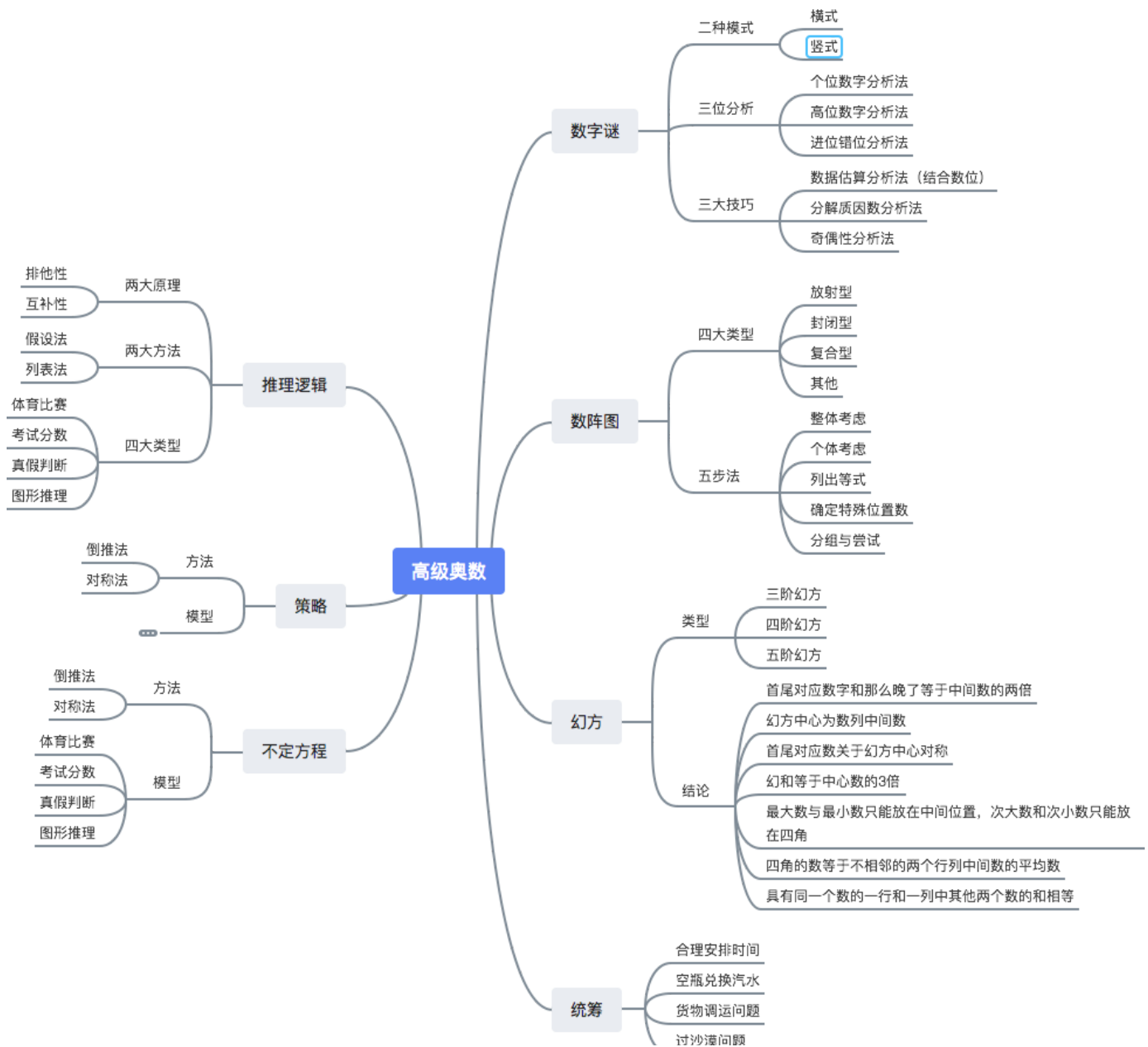
模型流程图

- [线上演示地址](#)
- [代码地址](#)



思维脑图

- [线上演示地址](#)
- [代码地址](#)



Editor

该类是整个编辑器的主控类，其主要职责是将编辑器的各个组件协同起来。

实例化方式：

```
import Editor from '@antv/g6-editor';  
const editor = new Editor();
```

实例方法

add

添加组件

```
editor.add(component);
```

参数：

`component` {object} 组件

getCurrentPage

获取当前页面

```
editor.getCurrentPage();
```

返回：

executeCommand

执行命令。编辑器中可回滚的操作，我们称之为命令。

```
editor.executeCommand(command)
```

参数

command {string || callback} 命令名称 或 回调函数

命令列表

命令英文名	命令中文名	Mac 快捷键	Win 快捷键	适用页面
clear	清空画布			All
selectAll	全选	⌘A	Ctrl+A	All
undo	撤销	⌘Z	Ctrl + Z	All
redo	重做	⇧⌘Z	Shift + Ctrl + Z	All
delete	删除	Delete	Delete	All
zoomIn	放大	⌘=	Ctrl + =	All
zoomOut	缩小	⌘-	Ctrl + -	All
autoZoom	自适应尺寸			All
resetZoom	实际尺寸	⌘0	Ctrl + 0	All
toFront	提升层级			All
toBack	下降层级			All
copy	复制	⌘C	Ctrl + C	Flow

paste	粘贴	⌘V	Ctrl + V	Flow
multiSelect	多选模式			Flow
addGroup	成组	⌘G	Ctrl + G	Flow
unGroup	取消组	⇧⌘G	Shift + Ctrl + G	Flow
append	添加相邻节点	Enter	Enter	Mind
appendChild	添加子节点	Tab	Tab	Mind
collapseExpand	折叠/展开	⌘/	Ctrl + /	Mind

on

事件监听

参数

eventName {string} 事件名

callback {function} 事件回调函数

事件对象

```
{
  // 命令对象
  command: {
    name,          // {string} 命令名称
    queue,         // {boolean} 是否进入命令队列（进入命令队列，意味着该命令可以撤销、重做）
  },
}
```

命令事件

```
editor.on('beforecommandexecute', ev=>{}); // 执行命令前  
editor.on('aftercommandexecute', ev=>{}); // 执行命令后
```

destroy

销毁编辑器

```
editor.destroy();
```

Command

命令是暴露给用户操作编辑器的最小接口，原则上所有的对图的操作都应该封装成命令。

对象获取方式：

```
const Command = G6Editor.Command;
```

静态方法

registerCommand

注册命令

```
Command.registerCommand(commandName, {  
  queue: true, // 命令是否进入队列，默认是 true  
  // 命令是否可用  
  enable(/* editor */) {  
  
  },  
  // 正向命令  
  execute(/* editor */) {  
  
  },  
  // 反向命令  
  back(/* editor */) {  
  
  }  
});
```

命令注册接口也可以用作重载已有命令的方法，比如：

```
Command.registerCommand('delete', {  
  enable() {
```



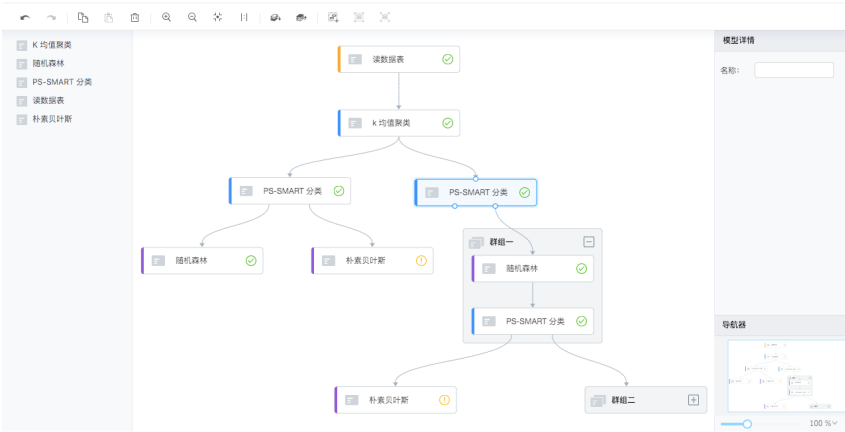
```
        return false;  
    }  
})
```

上述代码会重载删除（delete）命令的是否可用（enable）的方法。

Flow

简介

流程图页面类，继承自 Page 专用于构建 有向 的流程图编辑器。



获取方式：Editor.Flow，创建 Flow 实例的方式如下：

```
new Editor.Flow(cfg);
```

配置项

graph

G6 图配置项，参考 [G6.Graph API](#) 。 必选

注意：svg 渲染暂时有比较严重的性能问题，暂时不推荐使用。

align

对齐配置项 {object}，可以配置如下属性： 可选

```
align: {
  line: object, // 对齐线
  样式
  item: true || false || 'horizontal' || 'vertical' || 'center', // 图项对齐
  grid: true || false || 'cc' || 'tl' // 网格对齐
}
```

grid

网格线配置 {object}，可以配置如下属性： 可选

```
grid: {
  cell: number, // 网孔尺寸
  line: object, // 网格线样式
}
```

shortcut

快捷键配置，命令快捷键对照表参考 Editor API 中的命令列表 {object} 可选

示例：

```
const flow = new Editor.Flow({
  shortcut: {
    zoomIn: true, // 开启放大快捷键
    zoomOut: false, // 开启视口缩小快捷键
  },
})
```

noEndEdge

是否支持悬空边 {boolean}

静态方法

registerNode

注册一个流程图节点

```
Flow.registerNode(name, {  
  // 参考: https://www.yuque.com/antv/g6/detail-anchor  
  anchor,  
  // 绘制  
  draw(item) {  
    return keyShape;  
  },  
}, extendShape);
```

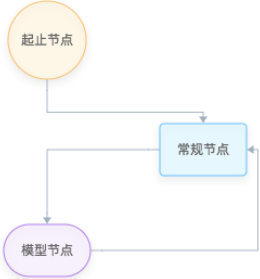
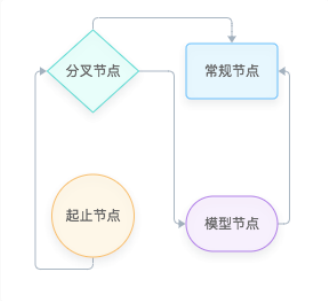

registerEdge

注册一个流程图边

```
Flow.registerEdge(name, {  
  // 绘制  
  draw(item) {  
    return keyShape;  
  },  
}, extendShape);
```

内部注册的边:

名称	解释	示例
----	----	----

flow-polyline	流程图折现	
flow-polyline-round	流程图圆角折现	
flow-smooth	流程图曲线	

registerGroup

注册一个流程图的组

```
Flow.registerGroup(name, {
  // 绘制
  draw(item) {
    return keyShape;
  },
}, extendShape);
```

registerGuide

注册一个流程图的导引

```
Flow.registerGuide(name, {  
  // 绘制  
  draw(item) {  
    return keyShape;  
  },  
}, extendShape);
```

实例方法

save

保存数据

```
const data = flow.save();
```

返回

`data` {object} 图数据

read

读数据

```
const data = flow.read(data);
```

参数

`data` {object} 图数据（一般情况下由所对应图类 `save` 导出）

导入的数据模型，以下键名在 Flow 数据中有特定含义，是保留字段，用户在设置自有数据时应 `避免使用`。

节点的数据模型（属性值为样例数据）

```
{
  id: 'node1',                // id 必须唯一
  color: '#333',              // 颜色（该颜色被认为是 ant-design-palettes 的6阶色，激活颜色、选中会根据该色值自动设置）
  size: 10 || [10, 10],       // 尺寸 || [宽, 高]
  shape: 'circle',            // 所用图形
  style: {                     // 关键形样式（优先级高于color）
    fill: 'red',
    stroke: 'blue'
  },
  label: '文本标签' || {      // 文本标签 || 文本图形配置
    text: '文本标签',
    fill: 'green'
  },
  parent: 'group1',           // 所属组
  index: 1,                   // 渲染层级
}
```

边的数据模型（属性值为样例数据）

```
{
  id: 'edge1',                // id 必须唯一
  source: 'node1',            // 源节点 id
  target: 'node2',            // 目标节点 id
  controlPoints: [{           // 控制点
    x: 10,
    y: 10
  }],
  sourceAnchor: 0,             // 源节点锚点
  targetAnchor: 2,            // 目标节点锚点
  shape: 'line',              // 所用图形
  style: {                     // 关键形样式（优先级高于color）
    fill: 'red',
    stroke: 'blue'
  },
  label: '文本标签' || {      // 文本标签 || 文本图形配置
    text: '文本标签',
    fill: 'green'
  },
}
```

```

    },
    labelRectStyle: {           // 文本矩形底的样式
        fill: 'blue'
    },
    parent: 'group1',          // 所属组
    index: 1,                   // 渲染层级
}

```

参数

`data` {object} 图数据

on

事件监听

参数

`eventName` {string} 事件名

`callback` {function} 事件回调函数

事件对象

```

{
    currentItem,           // drag 拖动子项
    currentShape,          // drag 拖动图形
    shape,                 // 图形对象
    item,                  // 事件目标图项
    domEvent,              // 原生的 dom 事件
    x,                     // 图横坐标
    y,                     // 图纵坐标
    domX,                  // dom横坐标
    domY,                  // dom纵坐标
    anchor: {              // 锚点
        index,             // 锚点索引
        x,                 // 锚点横坐标
        y,                 // 锚点纵坐标
    }
}

```



```

    ...cfg          // 锚点自定义配置
  },
  source,           // 边的源节点
  target,           // 边的目标节点
  sourceAnchor,     // 源锚点索引
  targetAnchor,     // 目标锚点索引
  dragEndPointType, // 被拖动的边的端点，可取值为：'source' || 'target'
  action,           // 数据变更动作 add、update、remove、changeData
  toShape,          // mouseleave、dragleave 到达的图形
  toItem,           // mouseleave、dragleave 到达的子项
}

```

鼠标事件

这部分事件由 `graph` 提供，这类事件可以与前缀 ”（空即任意），'node'，'edge'，'group'，'anchor' 自由组合使用，例如：

```

const graph = flow.getGraph();
graph.on('click', ev=>{});           // 任意点击事件
graph.on('node:click', ev=>{});      // 节点点击事件
graph.on('edge:click', ev=>{});      // 边点击事件
graph.on('group:click', ev=>{});     // 组点击事件
graph.on('anchor:click', ev=>{});    // 锚点击事件

```

数变更事件

```

flow.on('beforechange', ev=>{});      // 图数据变更前
flow.on('afterchange', ev=>{});       // 图数据变更后

```

状态变更事件

```

flow.on('beforeitemselected', ev=>{}); // 选中前
flow.on('afteritemselected', ev=>{});  // 选中后
flow.on('beforeitemunselected', ev=>{}); // 取消选中前
flow.on('afteritemunselected', ev=>{}); // 取消选中后

```

控制事件

这类事件能决定某些行为是否能成功触发

```
// 鼠标悬浮节点后显示锚点前，可用作控制，锚点是否显示
flow.on('hoovernode:beforeshowanchor', ev=>{
  // ev.item      子项
  ev.cancel = true; // 若设置为 true 则取消显示锚点
});

// 从锚点拖出边后，显示锚点前，可用作控制，要去连的锚点是否显示
flow.on('dragedge:beforeshowanchor', ev=>{
  // ev.sourceAnchor 源锚点
  // ev.source       源子项
  // ev.targetAnchor 目标锚点
  // ev.target       目标子项
  ev.cancel = true; // 若设置为 true 则取消显示要去连接的锚点
});

// 鼠标悬浮锚点后，开启添加边模式前，可用作控制，哪些锚点可添加边
flow.on('hoveranchor:beforeadddedge', ev=>{
  // ev.anchor 锚点
  // ev.item   子项
  ev.cancel = true; // 若设置该锚点不会触发连接模式
});
```

setSelected

将图项目，设置为选中或非选中状态

```
flow.setSelected(item, bool);
```

参数

`item` 图项 或 图项 id

`bool` true 为选中, false 为取消选中

getSelected

获取选中图项集

```
const items = flow.getSelected();
```

返回

`items` 选中图项集

clearSelected

取消所有图项的选中状态

```
flow.clearSelected();
```

getGraph

获取 G6.Graph 实例

```
const graph = flow.getGraph();
```

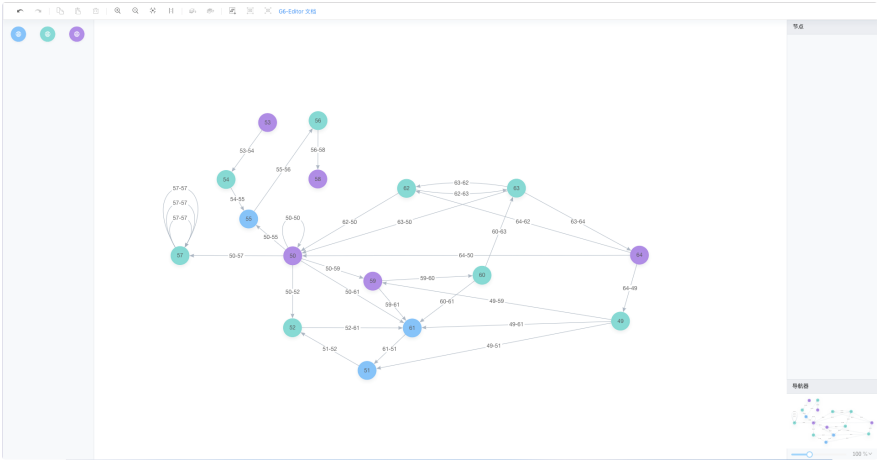
返回

`graph` G6.Graph 实例

Koni

简介

Koni 是一款针对一般网络图、拓扑图制作的图编排画板。



命名故事：图论起源于哥尼斯堡（Konigsberg）七桥问题，为纪念图论的诞生，取 Konigsberg 前四个字母命名该模板

获取方式：Editor.Koni，创建 Koni 实例的方式如下：

```
new Editor.Koni(cfg);
```

配置项

graph

G6 图配置项，参考 [G6.Graph API](#) 。 **必选**

注意：svg 渲染暂时有比较严重的性能问题，暂时不推荐使用。

align

对齐配置项 {object}，可以配置如下属性： **可选**

```
align: {
  line: object, // 对齐线
  样式
  item: true || false || 'horizontal' || 'vertical' || 'center', // 图项对齐
  grid: true || false || 'cc' || 'tl' // 网格对齐
}
```

grid

网格线配置 {object}，可以配置如下属性： 可选

```
grid: {
  cell: number, // 网孔尺寸
  line: object, // 网格线样式
}
```

shortcut

快捷键配置，命令快捷键对照表参考 Editor API 中的命令列表 {object} 可选

示例：

```
const koni = new Editor.Koni({
  shortcut: {
    zoomIn: true, // 开启放大快捷键
    zoomOut: false, // 开启视口缩小快捷键
  },
})
```

noEndEdge

是否支持悬空边 {boolean}

静态方法

registerNode

注册一个流程图节点

```
Koni.registerNode(name, {  
  // 参考: https://www.yuque.com/antv/g6/detail-anchor  
  anchor,  
  // 绘制  
  draw(item) {  
    return keyShape;  
  },  
}, extendShape);
```

registerEdge

注册一个流程图边

```
Koni.registerEdge(name, {  
  // 绘制  
  draw(item) {  
    return keyShape;  
  },  
}, extendShape);
```

registerGroup

注册一个流程图的组

```
Koni.registerGroup(name, {  
  // 绘制  
  draw(item) {  
    return keyShape;  
  },  
}, extendShape);
```

```
    },  
    }, extendShape);
```

registerGuide

注册一个流程图的导引

```
Koni.registerGuide(name, {  
    // 绘制  
    draw(item) {  
        return keyShape;  
    },  
    }, extendShape);
```

实例方法

save

保存数据

```
const data = koni.save();
```

返回

`data` {object} 图数据

read

读数据

```
const data = koni.read(data);
```

参数

`data {object}` 图数据（一般情况下由所对应图类 `save` 导出）

导入的数据模型，以下键名在 Koni 数据中有特定含义，是保留字段，用户在设置自有数据时应 **避免使用**

。

节点的数据模型（属性值为样例数据）

```
{
  id: 'node1',                // id 必须唯一
  color: '#333',              // 颜色（该颜色被认为是 ant-design-palettes 的6阶色，激活颜色、选中会根据该色值自动设置）
  size: 10 || [10, 10],       // 尺寸 || [宽, 高]
  shape: 'circle',            // 所用图形
  style: {                    // 关键形样式（优先级高于color）
    fill: 'red',
    stroke: 'blue'
  },
  label: '文本标签' || {      // 文本标签 || 文本图形配置
    text: '文本标签',
    fill: 'green'
  },
  parent: 'group1',           // 所属组
  index: 1,                   // 渲染层级
}
```

边的数据模型（属性值为样例数据）

```
{
  id: 'edge1',                // id 必须唯一
  source: 'node1',            // 源节点 id
  target: 'node2',            // 目标节点 id
  controlPoints: [{           // 控制点
    x: 10,
    y: 10
  }],
}
```



```

    sourceAnchor: 0,          // 源节点锚点
    targetAnchor: 2,         // 目标节点锚点
    shape: 'line',           // 所用图形
    style: {                  // 关键形样式（优先级高于color）
      fill: 'red',
      stroke: 'blue'
    },
    label: '文本标签' || {    // 文本标签 || 文本图形配置
      text: '文本标签',
      fill: 'green'
    },
    labelRectStyle: {         // 文本矩形底的样式
      fill: 'blue'
    },
    parent: 'group1',         // 所属组
    index: 1,                 // 渲染层级
  }

```

参数

`data` {object} 图数据

on

事件监听

参数

`eventName` {string} 事件名

`callback` {function} 事件回调函数

事件对象

```

{
  currentItem,          // drag 拖动子项
  currentShape,         // drag 拖动图形
  shape,                // 图形对象
  item,                 // 事件目标图项

```

```

domEvent,          // 原生的 dom 事件
x,                 // 图横坐标
y,                 // 图纵坐标
domX,              // dom横坐标
domY,              // dom纵坐标
anchor: {          // 锚点
  index,           // 锚点索引
  x,               // 锚点横坐标
  y,               // 锚点纵坐标
  ...cfg           // 锚点自定义配置
},
source,            // 边的源节点
target,            // 边的目标节点
sourceAnchor,      // 源锚点索引
targetAnchor,      // 目标锚点索引
dragEndPointType,  // 被拖动的边的端点，可取值为: 'source' || 'target'
action,            // 数据变更动作 add、update、remove、changeData
toShape,           // mouseleave、dragleave 到达的图形
toItem,            // mouseleave、dragleave 到达的子项
}

```

鼠标事件

这部分事件由 `graph` 提供，这类事件可以与前缀 ”（ `空即任意` ）, 'node', 'edge', 'group' 自由组合使用，例如：

```

const graph = koni.getGraph();
graph.on('click', ev=>{});           // 任意点击事件
graph.on('node:click', ev=>{});      // 节点点击事件
graph.on('edge:click', ev=>{});      // 边点击事件
graph.on('group:click', ev=>{});     // 组点击事件

```

状态变更事件

```

koni.on('beforeitemselected', ev=>{}); // 选中前
koni.on('afteritemselected', ev=>{});  // 选中后
koni.on('beforeitemunselected', ev=>{}); // 取消选中前
koni.on('afteritemunselected', ev=>{}); // 取消选中后

```

setSelected

将图项目，设置为选中或非选中状态

```
koni.setSelected(item, bool);
```

参数

`item` 图项 或 图项 id

`bool` true 为选中，false 为取消选中

getSelected

获取选中图项集

```
const items = koni.getSelected();
```

返回

`items` 选中图项集

clearSelected

取消所有图项的选中状态

```
koni.clearSelected();
```

Toolbar

工具栏类，负责工具栏按钮的命令绑定、可用禁用状态控制。

获取方式 `Editor.Toolbar`，以下方式实例化。

```
new Editor.Toolbar(cfg);
```

配置项

container

dom 容器 或 dom 容器 id;

容器标识约定

容器内：

1. `class` 含 **command** 的 dom 节点为命令容器
2. `data-command` 标识该 dom 节点具体所对应的命令
3. `class` 含 **disable** 表示该命令当前不可用

容器模版示例

```
<div id="container">
  <button data-command="copy" class="command">复制</button>
  <button data-command="paste" class="command disable">粘贴</button>
  <button data-command="custom" class="command" >自定义命令</button>
</div>
```


Detailpanel

属性栏类，负责属性栏显示隐藏的控制。

获取方式 `Editor.Detailpanel`，以下方式实例化。

```
new Editor.Detailpanel(cfg);
```

配置项

container

dom 容器 或 dom 容器 id;

容器标识约定

1. `data-status` 标识不同页面状态下，各个右键菜单容器的显示隐藏

容器模版示例

```
<div id="container">
  <div data-status="node-selected">节点属性栏</div>
  <div data-status="edge-selected">边属性栏</div>
  <div data-status="group-selected">群组属性栏</div>
  <div data-status="canvas-selected">画布属性栏</div>
  <div data-status="multi-selected">多选时属性栏</div>
</div>
```

Minimap

缩略图类，负责绘制缩略图及双图联动。

获取方式 `Editor.Minimap`，以下方式实例化。

```
new Editor.Minimap(cfg);
```

配置项

container

dom 容器 或 dom 容器 id {object || string} 必填

width

宽度 {number} 必填

height

高度 {number} 必填

viewportWindowStyle

缩略图可视区域视窗样式 {object} 参考 [G 绘图属性](#)

viewportBackStyle

缩略图背景样式 {object} 参考 [G 绘图属性](#)

Contextmenu

右键菜单类，负责处理右键菜单的显示隐藏、命令绑定、可用禁用状态控制。

获取方式 `Editor.Contextmenu`，以下方式实例化。

```
new Editor.Contextmenu(cfg);
```

配置项

container

dom 容器 或 dom 容器 id;

容器标识约定

容器内:

1. `class` 含 **menu** 的 dom 节点为菜单容器
2. `data-status` 标识不同页面状态下，各个右键菜单容器的显示隐藏
3. `class` 含 **command** 的 dom 节点为命令容器
4. `data-command` 标识该 dom 节点具体所对应的命令
5. `class` 含 **disable** 表示该命令当前不可用

容器模版示例

```
<div id="container">
  <div data-status="node-selected" class="menu">
    <button data-command="copy" class="command">复制</button>
    <button data-command="paste" class="command">粘贴</button>
    <button data-command="delete" class="command">删除</button>
  </div>
  <div data-status="edge-selected" class="menu">
```



```
    <button data-command="delete" class="command">删除</button>
</div>
<div data-status="group-selected" class="menu">
    <button data-command="copy" class="command">复制</button>
    <button data-command="paste" class="command">粘贴</button>
    <button data-command="unGroup" class="command">解组</button>
    <button data-command="delete" class="command">删除</button>
</div>
<div data-status="canvas-selected" class="menu">
    <button data-command="undo" class="command">撤销</button>
    <button data-command="redo" class="command disable">重做</button>
</div>
<div data-status="multi-selected" class="menu">
    <button data-command="copy" class="command">复制</button>
    <button data-command="paste" class="command">粘贴</button>
    <button data-command="addGroup" class="command">归组</button>
</div>
</div>
```

Itempanel

元素面板栏，负责处理元素添加的通讯。

获取方式 `Editor.Itempanel`，以下方式实例化。

```
new Editor.Itempanel(cfg);
```

配置项

container

dom 容器 或 dom 容器 id;

容器标识约定

容器内：

1. `class` 含 `getItem` 的 dom 元素可以被当作用于添加图节点
2. `data-*` 所有 `*` 都会被设置进添加图项的数据模型。

如：

```
<div data-size="50*50" data-id="aaa" data-shape="commonShape" data-type="node" data-label="原型">圆形</div>
```

添加到图中的元素 model 为

```
{
  id: "aaa",
  shape: "commonShape",
```

```
    label: "原型"  
  }
```

3. `data-type` 是保留字段，取值 'node' \ 'edge'，决定元素类型。
4. 特别留意：`data-size` 的写法是：

```
<div data-size="50*50" >圆形</div>
```

并且这个属性会直接决定委托图形（画布内拖拽节点时的虚框）尺寸。

容器模版示例

```
<div id="container">  
  <div class="getItem" data-type="node" data-shape="circle">圆形</div>  
  <div class="getItem" data-type="node" data-shape="rect">  
      
  </div>  
  <div class="getItem" data-type="node" data-shape="custom">自定义节点</div>  
</div>
```