#### 设计器应该包含主要功能

1）资源维护

1. 组件选择器
2. 组件树查看功能
3. 画布
4. 工具栏

保存

撤销

恢复

删除

对齐功能

生成代码

1. 属性编辑栏

数据集定义

事件定义

包含定义与后台交互的规则、访问数据集的规则，访问事件的规则

其他基本属性定义

1. 快捷键定义栏
2. 使用说明书

#### 设计器原理：



设计器主要关注点

数据集定义：

动作定义：

组件定义：

样式定义(目前没有考虑)：

四者之间的关系：



组件只是数据状态的展示，所有的展示通过操作数据集的变化来反映。

Store 关键数据说明：

components //组件树,Map 类型

rootID //组件树的根节点

dataSet //数据集

actions //动作集

domRefs //封装的

history //历史记录信息

如何构造组件树

基本用法：

1. 在初始化store时定义一个根节点，并初始化actions和dataSet
2. 拖拽到画布时，设定选择的拖拽值和目标值
3. 初始化目标值，添加到组件树中

用例

1在store初始化时，定义根节点，并初始化dataSet和 actions

Class Stroe{

@observable actions = null; //动作

@observable rootID = null; //组件树根节点ID

@observable dataSet = null; //数据集

@observable components = null;//组件树

constructor() {

this.slide = new Slide({

parent:null,

id: id,

type:'Slide',

props: { style: {height:'100%',width:'100%'}, transition: ["slide"] },

children: [],

event:{

init:'init'

}

})

this.components = new Map();

this.actions = new Map();

//定义一个初始化方法，规定此方法为加载组件后调用的方法

this.actions.set("init",{name:'init',body:'',paramters:'',action:function(){},describe:''});

this.dataSet = new Map();

this.components.set(id,this.slide);

}

}

如何遍历组件树

如何绘制组件

基本用法

1. 定义组件属性
2. 定义组件，根据组件属性实现组件render
3. 将组件添加到组件树中

举例以Text组件为例

1. 定义组件属性

elements[ElementTypes.TEXT] = new Element({

type: ElementTypes.TEXT,

props: {//属性定义

style: {

wordBreak: "break-word",

width: 200,

height: 50,

left:70,

top:0,

},

type:'text',

maxLength:10000,

minLength:0,

placeholder:'',

disabled:false,

size:'',

rows:'',

readOnly:false,

resize:false,

binding:'',

prefix:'',

suffix:'',

prepend:'',

append:'',

},

event:{//事件定义

blur:'',

focus:'',

change:'',

},

binding:"",//绑定的数据集

children: null

}

});

1. 定义组件render方式

在每个组件当中一定要实现getSize方法，此方法用于拖拽和变化大小功能

@observer

export default class TextElement extends Component {

static propTypes = {

...CanvasElementPropTypes,

rect: PropTypes.object,

component: PropTypes.shape({

props: PropTypes.object,

children: PropTypes.node

})

}

getSize = () => {

const {width,height}=this.context.store.getDom(this.props.index).getBoundingClientRect();

return {

width,

height,

left: this.props.component.props.style.left,

top: this.props.component.props.style.top

}}

static contextTypes = {

store: PropTypes.object

};

constructor(props, context) {

super(props, context);

}

render() {

/\*\*this.props.comonent是从组件树种获取传递过来的\*\*/

const componentProps = this.props.component.props;

const {event,binding} = this.props.component;

const {actions,dataSet} = this.context.store;

const bingdingValue =dataSet.has(binding)?dataSet.get(binding).data:'';

let width = componentProps.style.width ? componentProps.style.width : "auto";

width = this.props.rect ? this.props.rect.width : width;

let height = this.props.rect ? this.props.rect.height : componentProps.style.height;

const eventObject = {};

for(var key in event){

const newKey ="on"+TilteUtil.titleCase(key);

eventObject[newKey]=actions.has(event[key])?actions.get(event[key]).action:null;

}

return (

<div

key={this.props.index}

className={this.props.classes}

onMouseDown={this.props.mouseDownAction}

onDragOver={this.props.dragOverAction} style={{top:this.props.postions.top,left:this.props.postions.left,width,height}}

>

<CanvasElement --此组件是用于组件在画布上可以拖动和变化大小每个组件都要使用此组件

{...pick(this.props, Object.keys(CanvasElementPropTypes))}

resizeVertical={false}

getSize={this.getSize}

>

<Input {...componentProps} {...eventObject} value={bingdingValue} style={{width:'100%',height:'100%'}} />

</CanvasElement>

</div>

);

}

}

如何获取某个组件的dom信息

Dom数据封装来 Store 中domRef，使用时可以通过context来获取

this.context.store.getDom(“domID”);

##### 组件树数据结构 （Map 对象）

{

Key: 组件ID

Value:compent

}

##### 组件数据结构（ object 对象）

{

Id:’’组件ID

,type:’’

, props:{

}

,event:{

}

,binding:’’--绑定的数据集

}

##### 数据集的数据结构（Map 对象）

{

Key : ’’-数据集名称

Value : {

Name:’’ --数据集名称

Describe:’’数据集描述信息

Type:’’ -- 数据集类型，arry,object,Boolean,string,number

initData: object--初始值,

data: --操作的值

}

}

##### 动作集的数据结构（Map 对象）

{

Key:动作名称

Value{

Name:’’动作名称

Describe:’’ 动作描述

parameters:’’传入参数，多个参数以逗号分隔

body:’’函数体

action:new Function(this.parameters,this.body)生成的js函数

}

}

##### 保存的数据结构

保存的数据结构为xml结构。举例

<?xml version="1.0" encoding="utf-8"?>

<root>

<Slide> --元件组定义

<parent>null</parent>

<type>Slide</type>

<id>231b8513-8c84-454f-ba1e-4161e132aa8b</id>

<props>

<style><![CDATA[{"height":"100%","width":"100%"}]]></style>

<transition><![CDATA[["slide"]]]></transition>

</props>

<event><![CDATA[{"init":"init"}]]></event>

<binding>undefined</binding>

<children>

<Button>

<parent>231b8513-8c84-454f-ba1e-4161e132aa8b</parent>

<type>Button</type>

<id>13cbfb6f-4e58-4ce0-ae34-a08b23bd7fba</id>

<props>

<style><![CDATA[{"width":100,"height":50,"left":121,"top":133}]]></style>

<label><![CDATA["按钮"]]></label>

<size><![CDATA["large"]]></size>

<type><![CDATA["primary"]]></type>

<plain><![CDATA[false]]></plain>

<loading><![CDATA[false]]></loading>

<disabled><![CDATA[false]]></disabled>

<icon><![CDATA[""]]></icon>

<nativeType><![CDATA["button"]]></nativeType>

</props>

<event><![CDATA[{"click":""}]]></event>

<binding>undefined</binding>

<children/>

</Button>

</children>

</Slide>

<Actions>--动作定义

<child>

<name>init</name>

<describe/>

<body><![CDATA[]]></body>

<paramters><![CDATA[]]></paramters>

<action><![CDATA[function action() {}]]></action>

</child>

</Actions>

<DataSet>

</DataSet>

</root>

##### 开发使用依赖的模块说明

Mobx ：状态管理的库，文档地址：http://cn.mobx.js.org/

React-dnd ：拖拽组件：https://react-dnd.github.io/react-dnd/docs-overview.html

Xmljs: xml解析 https://github.com/blackshadev/xmljs

###### 拖拽实现原理：

基本用法

1. 把应用的根组件包装在 DragDropContext 中
2. 把可以拖拽的组件包装在 DragSource 中
   1. 设置 type，DragSource 和 DropTarget 必须指定 type。只有在 type 相同的情况下，DragSource 才能放到 DropTarget 中
   2. 设置 spec，让组件可以响应拖拽事件
   3. 设置 collect，把拖拽过程中需要信息注入组件的 props
3. 把可以接受拖拽的组件包装在 DropTarget 中
   1. 设置 type，DragSource 和 DropTarget 必须指定 type。只有在 type 相同的情况下，DragSource 才能放到 DropTarget 中
   2. 设置 spec，让组件可以响应拖拽事件
   3. 设置 collect，把拖拽过程中需要信息注入组件的 props
4. 完

实例：

1. 将应用根包装在DragDropContxt中

import { DragDropContextProvider } from 'react-dnd'

import HTML5Backend from 'react-dnd-html5-backend'

class Desginer extends Component {

render(){

return (

<DragDropContextProvider backend={HTML5Backend}> // HTML5Backend不在在触摸屏下使用

<Canvas></Canvas>

</DragDropContextProvider>

</div>

)}

);

}

2、可拖拽元件包装在Dragsource中

import { DragSource } from 'react-dnd'

//定义spec

const boxSource = {

  beginDrag(props) {

    return {

      elementType: props.elementType,

    }

  },

  endDrag(props, monitor) {

const item = monitor.getItem()

const dropResult = monitor.getDropResult()//获取drop目标

     if (dropResult) {

//获取drop目标后执行想做的操作

}

)

   }

  },

}

@DragSource("element-types",

boxSource, (connect, monitor) => ({

  connectDragSource: connect.dragSource(),

  isDragging: monitor.isDragging(),

}))

class ElementItem extends Component {

/\*\*增加属性\*\*/

static propTypes = {

connectDragSource: PropTypes.func.isRequired,

    isDragging: PropTypes.bool.isRequired,

elementType: PropTypes.string.isRequired,

scale: PropTypes.number

};

 const { connectDragSource } = this.props

return connectDragSource(

<li>

文本框

</li>

);

}

3、把可以接受拖拽的组件包装在 DropTarget 中

import { DropTarget } from 'react-dnd'

//设置spec

const boxTarget = {

  drop(props,monitor) {

if(monitor.getDropResult()){

return {

id: monitor.getDropResult().id,

};

}

return { id: props.index}

},

}

@DropTarget("element-types", boxTarget, (connect, monitor) => ({

  connectDropTarget: connect.dropTarget(),

  isOver: monitor.isOver(),

  canDrop: monitor.canDrop(),

}))

class Slide extends Component {

//定义属性

static propTypes = {

scale: PropTypes.number.isRequired,

connectDropTarget: PropTypes.func.isRequired,

    isOver: PropTypes.bool.isRequired,

    canDrop: PropTypes.bool.isRequired,

};

render() {

const { canDrop, isOver, connectDropTarget } = this.props;

return connectDropTarget(

<div canDrop={canDrop} isOver={isOver} >

</div>

);

}

}

###### 状态管理实现原理：

基本用法

1、定义页面数据和相关的数据操作的一个store。

2、定义一个Provider 组件可以将store通过react的context机制传递给子组件

3、将根组件包裹在Provider中

3、定义static contextTypes 中增加属性 store 使子组件获取 store。

实例

1、定义store

export default class Store{

@observable number = 1;// observable,用于标注这个对象是可以观察的

addNumber()

{

this.number++;

}

}

1. 定义Provider

export default class Provider extends React.Component {

constructor(props, context) {

super(props, context);

this.store = props.store;

}

getChildContext() {

return { store: this.store };

}

render() {

return React.Children.only(this.props.children);

}

}

Provider.childContextTypes = {

store: PropTypes.object

};

Provider.propTypes = {

store: PropTypes.object.isRequired,

children: PropTypes.any

};

1. 将根组件用Provider包裹

class App xtends Component {

render() {

let store = new Store(); //store实例化

return (

<Provider store={store}>

<Canvas></Canvas>

</Provider>

);

}

}

1. 组件中获取store

import { observer } from "mobx-react";

@observer //用于监听store中数据变化

class Canvas extend Component{

static contextTypes = {

store: PropTypes.object

};

\_onClick = {

this.context.store.addNumber():

}

render() {

const {number} = this.context.store; //获取store

return (

<div OnClick={this.\_onClick}>{ number}</div>

);

}

}

关键地方说明：

每一个组件的序列化和反序列化方式有可能不一样，扩展了js的object对象，定义了一个基础对象类型，定义的元素属性需要通过Element进行包装。

export default class Element extends Object{

constructor(...items) {

super(...items);

}

serializeStart=(compent)=>{

let xml = new Array();

const {id,type,props,children,parent,event,binding} = compent;

xml.push('<'+type+">")

xml.push('<parent>'+parent+'</parent>')

xml.push('<type>'+type+'</type>')

xml.push('<id>'+id+'</id>')

xml.push('<props>')

for(var key in props ){

xml.push('<'+key+'><![CDATA['+JSON.stringify( props[key] )+']]></'+key+'>')

}

xml.push('</props>')

xml.push('<event><![CDATA['+JSON.stringify(event )+']]></event>')

xml.push('<binding>'+binding+'</binding>')

return xml;

}

serializeEnd=(type)=>{

return '</'+type+">"

}

deserialize=(xmlNode)=>{

const {type,props,id,parent,children} = xmlNode;

let typetemp = this.deserializeType(type);

const element = elementMap[typetemp] ;

element.id = this.deserializeID(id);

element.type = typetemp;

element.parent = this.deserializeParent(parent);

element.props = this.deserializePropety(props);

element.children = this.deserializeChildren(children);

return element;

}

deserializeID=(idArray)=>{

if(idArray == null || idArray.length==0) return null;

return idArray[0];

}

deserializeParent=(parentArray)=>{

if(parentArray == null || parentArray.length==0) return null;

return parentArray[0];

}

deserializeType=(typeArray)=>{

if(typeArray == null || typeArray.length==0) return null;

return typeArray[0];

}

deserializeChildren=(nodes)=>{

const children = new Array();

if(nodes==null || nodes.length==0) return;

for(let key in nodes[0]){

let nodeArr = nodes[0][key];

for(let i=0,length=nodeArr.length;i<length;i++ ){

let node = nodeArr[i];

const {id} = node;

children.push(id[0])

}

}

return children;

}

deserializePropety=(propsArr)=>{

if(propsArr == null || propsArr.length==0) return {};

const props = {};

const temp = propsArr[0];

const {style,transition,isQuote,listType,size} = temp;

if(style){

props.style = JSON.parse(style[0]);

}

if(transition){

props.transition = transition[0];

}

if(isQuote){

props.isQuote = isQuote[0];

}

if(listType){

props.isQuote = listType[0];

}

if(size){

props.size = size[0];

}

return props;

}

}

定义的规则：

访问后端服务的规则

引入了fetch.js

访问数据集的规则

#{data.数据集的名称}

访问方法集的规则

#{action.方法集的名称}

后期需要的内容

1）深入组件的定义

1）权限

2）路由

3）静态文件的存储