可视化拖拽编辑器(二)

一.实现拖拽辅助线

> 修改逻辑确保*mousedown*时至少有一个*block*节点是被选中的。

```
const blockMousedown = (e, block, index) => {
    e.preventDefault();
    e.stopPropagation();
    // block上我们规划一个属性 focus 获取焦点后就将focus变为true
    if (e.shiftKey) {
        if (focusData.value.focus.length <= 1) {
            block.focus = true
        } else {
            block.focus = !block.focus
        }
    } else {
        if (!block.focus) {
            clearBlockFocus();
            block.focus = true; // 要清空其他人foucs属性
        }
    }
    selectIndex.value = index;
    callback(e)
}
```

1.获取拖拽的节点

> 根据用户的选择找到最后点击的*block*，根据此*block*计算辅助线

```
import {computed} from 'vue'
export function useFocus(data,callback){ // 获取哪些元素被选中了
    // 记录最后选中的block的索引
+   const selectIndex = ref(-1);
    // 根据索引得到最后选中的block
+   const lastSelectBlock = computed(()=>
data.value.blocks[selectIndex.value])

    const containerMousedown = () => {
        clearBlockFocus();
+       selectIndex.value = -1; // 点击容器还原状态
    }
    const blockMousedown = (e, block,index) => {
        // ...
+       selectIndex.value = index; // 记录当前选中的索引
        callback(e)
    }
    return {
        blockMousedown,
        containerMousedown,
        focusData,
+       lastSelectBlock
    }
}
```

2.计算参考线位置

> 在初始化的时候给组件添加宽度

```
onMounted(() => {
    let { offsetWidth, offsetHeight } = blockRef.value;
    if (props.block.alignCenter) { // 说明是拖拽松手的时候才渲染的，其他的默认渲染到页面上的内容不需要居中
        props.block.left = props.block.left - offsetWidth / 2;
        props.block.top = props.block.top - offsetHeight / 2; // 原则上重新派发事件
        props.block.alignCenter = false; // 让渲染后的结果才能去居中
    }
    props.block.width = offsetWidth;
    props.block.height = offsetHeight;
})
```

```
import { reactive } from "vue"

export function useBlockDragger(focusData, lastSelectBlock) {
    let dragState = {
        startX: 0,
        startY: 0
    }
    let markLine = reactive({ // 计算参考线x和y
        x: null,
        y: null
    })
    const mousedown = (e) => {
+       const { width:BWidth, height:BHeight } = lastSelectBlock.value
        dragState = {
            startX: e.clientX,
            startY: e.clientY, // 记录每一个选中的位置
            startPos: focusData.value.focus.map(({ top, left }) => ({
top, left })),
+           startLeft: lastSelectBlock.value.left, // 记录block开始的left
和top
+           startTop: lastSelectBlock.value.top,
            lines: (() => {
                // 辅助线根据未选中的元素来计算
                const { unfocused } = focusData.value;
                let lines = { x: [], y: [] };
                unfocused.forEach(block => {
                    const { top: ATop, left: ALeft, width: AWidth,
height: Aheight } = block; // 未选中元素的信息
                    console.log(ATop,ALeft,AWidth,Aheight)

                    lines.y.push({ showTop: ATop, top: ATop }); // 头对头

                    lines.y.push({ showTop: ATop, top: ATop - BHeight
}); // 头对底
                    lines.y.push({ showTop: ATop + Aheight / 2 , top:
ATop + Aheight / 2 - BHeight / 2}); // 中对中
                    lines.y.push({ showTop: ATop + Aheight, top: ATop +
Aheight }); // 底对顶
                    lines.y.push({ showTop: ATop + Aheight, top: ATop +
Aheight - BHeight }) // 底对底
                    lines.x.push({ showLeft: ALeft, left: ALeft }); // 左
对左
                    lines.x.push({ showLeft: ALeft + AWidth, left: ALeft
+ AWidth }); // 右对左
                    lines.x.push({ showLeft:ALeft + AWidth / 2 , left:
ALeft + AWidth / 2 - BWidth / 2 }); // 中对中
                    lines.x.push({ showLeft: ALeft + AWidth, left: ALeft
+ AWidth - BWidth }); // 右对右
                    lines.x.push({ showLeft: ALeft, left: ALeft - BWidth
}); // 左对右
                });
                return lines
            })()
        }
        document.addEventListener('mousemove', mousemove);
        document.addEventListener('mouseup', mouseup)
    }
    const mousemove = (e) => {
        let { clientX: moveX, clientY: moveY } = e;

        let left = moveX - dragState.startX + dragState.startLeft; // 最
新的left值
```

```
        let top = moveY - dragState.startY + dragState.startTop; // 最新
的top值

        let x = null;
        let y = null;
        // 比较最新值和距离最近的参考线位置
        for (let i = 0; i < dragState.lines.y.length; i++) {
            const { top: t, showTop: s } = dragState.lines.y[i];
            if (Math.abs(t - top) < 5) { // 如果接近5像素
                y = s; // 要显示线的位置
                moveY = dragState.startY - dragState.startTop + t; // 容
器距离顶部的距离 + 目标位置
                break; // 直接改到目的地的位置，保证自动对齐
            }
        }
        for (let i = 0; i < dragState.lines.x.length; i++) {
            const { left: l, showLeft: s } = dragState.lines.x[i];
            if (Math.abs(l - left) < 5) {
                x = s; // 要显示线的位置
                moveX = dragState.startX - dragState.startLeft + l; // 容
器距离顶部的距离 + 目标位置
                break;
            }
        }
        markLine.x = x;
        markLine.y = y;
        let durX = moveX - dragState.startX;
        let durY = moveY - dragState.startY;
        focusData.value.focus.forEach((block, idx) => {
            block.top = dragState.startPos[idx].top + durY;
            block.left = dragState.startPos[idx].left + durX;
        })
    }
    const mouseup = (e) => {
        document.removeEventListener('mousemove', mousemove);
        document.removeEventListener('mouseup', mouseup);
+       markLine.x = null;
+       markLine.y = null;
    }
    return {
        mousedown,
        markLine
    }
}
```

```
{(markLine.x !== null) && <div class="line-x" style={{ left:
`${markLine.x}px` }}></div>}
{(markLine.y !== null) && <div class="line-y" style={{ top:
`${markLine.y}px` }}></div>}
```

```
.line-x{
    position: absolute;
    top: 0;
    bottom: 0;
    border-left: dashed 1px red
}
.line-y{
    position: absolute;
    left:0px;
    right:0px;
    border-top: dashed 1px red
}
```

3.实现画布对齐

```
[
    ...unfocused,
    {
        top:0,
        left:0,
        width:data.value.container.width,
        height:data.value.container.height
    }
]
```

> 将画布也计入参考物中即可

二.按钮操作的设计

1.按钮布局及样式

> 引入字体图标

```
@import "../iconfont/iconfont.css";
```

> 实现撤销及重做功能

```
const buttons = [
    {label:'撤销',icon:'icon-back',handler:()=>console.log('撤销')},
    {label:'重做',icon:'icon-forward',handler:()=>console.log('重做')}
];
```

```
 <div class="editor-top">
    {buttons.map((btn,index)=>{
        return <div class="editor-top-button" onClick={btn.handler}>
            <i class={`iconfont ${btn.icon}`}></i>
            <span> {btn.label}</span>
        </div>
    })}
</div>
```

```
&-top{
    position: absolute;
    right:280px;
    left:280px;
    height:80px;
    display: flex;
    justify-content: center;
    align-items: center;
    &-button {
        width:60px;
        height:60px;
        display: flex;
        flex-direction: column;
        align-items: center;
        justify-content: center;
        background-color: rgb(0, 0, 0,.3);
        color:#fff;
        & + & {
            margin-left: 1px;
        }
        span{
            font-size: 14px;;
        }
    }
}
```

2.实现命令注册

```
let {commands} = useCommand(data);
 const buttons = [
     { label: '撤销', icon: 'icon-back', handler: () => commands.undo()
},
     { label: '重做', icon: 'icon-forward', handler: () =>
commands.redo() }
 ];

export function useCommand(data){
    const state = {
        current: -1, // 前进后退的一个索引值
        queue: [], // 存放操作队列
        commands: {}, // 注册的所有命令  映射表
        commandArray: [], // 注册命令的数组
    }
    const registry = (command) => {
        state.commandArray.push(command);
        state.commands[command.name] = (...args) => {
            const { redo } = command.execute(...args);
            redo();
        }
    }
    // 撤销
    registry({
        name: 'undo',
        keyboard: 'ctrl+z',
        execute() { // 调用对应的快捷键或者点击时会执行execute函数
            return {
                redo: () => { // 默认会调用redo方法
                    console.log('后退')
                }
            }
        }
    })
    // 重做
    registry({
        name: 'redo',
        keyboard: 'ctrl+y',
        execute() {
            return {
                redo: () => {
                    console.log('前进')
                }
            }
        }
    })
    return state
}
```

> 调用注册的命令。

3.实现拖拽菜单后撤回、重做功能

> 需要记录拖拽的前后状态，采用发布订阅模式

```
npm install mitt
```

```
import mitt from "mitt";
export const events = mitt()
```

```
const dragstart = (e, component) => {
    containerRef.value.addEventListener('dragenter', dragenter)
    containerRef.value.addEventListener('dragover', dragover)
    containerRef.value.addEventListener('dragleave', dragleave)
    containerRef.value.addEventListener('drop', drop)
    currentComponent = component
    events.emit('start');  // 拖拽前发射事件
}
const dragend = (e)=>{
    containerRef.value.removeEventListener('dragenter', dragenter)
    containerRef.value.removeEventListener('dragover', dragover)
    containerRef.value.removeEventListener('dragleave', dragleave)
    containerRef.value.removeEventListener('drop', drop)
    events.emit('end');   // 拖拽后发射事件
}
```

注册拖拽命令

```
registry({
    name: 'drag',
    pushQueue: true,
    init() {
        this.before = null;
        const start = () => this.before = deepcopy(data.value.blocks);
// 记录拖拽前的状态
        const end = () => state.commands.drag(); // 拖拽后调用指令
        events.on(start);
        events.on(end);
        return () => { // 返回解除绑定事件
            events.off(start);
            events.off(end);
        }
    },
    execute() {
        let before = this.before; // 之前
        let after = data.value.blocks // 之后
        return {
            redo: () => { // 默认向后
                data.value = { ...data.value, blocks: after }
            },
            undo: () => { // 向前事件
                data.value = { ...data.value, blocks: before }
            }
        }
    }
});
// 初始化
;(() => {
    state.commandArray.forEach(command => command.init &&
state.destroyList.push(command.init()))
})();
onUnmounted(()=>{
    state.destroyList.forEach(fn => fn && fn()); // 卸载绑定的事件
});
```

```
const registry = (command) => {
    state.commandArray.push(command);
    state.commands[command.name] = (...args) => {
        const { redo, undo } = command.execute(...args);
        redo();
        if (!command.pushQueue) {
            return
        }
        let { queue, current } = state;
        if (queue.length > 0) { // 如果添加的过程中有撤销去掉不需要的内容
            queue = queue.slice(0, current + 1); //
            state.queue = queue;
        }
        queue.push({ undo, redo }); // 记住操作
        state.current = current + 1; // 累加1
    }
}
```

前进/后退实现

```
registry({ // 撤销
    name: 'undo',
    keyboard: 'ctrl+z',
    execute(data) { // 调用对应的快捷键或者点击时会执行execute函数
        return {
            redo: () => { // 默认会调用redo方法
                if (state.current === -1) { return }
                const queueItem = state.queue[state.current]
                if (queueItem) {
                    queueItem.undo && queueItem.undo()
                    state.current--
                }
            }
        }
    }
})
registry({ // 重做
    name: 'redo',
    keyboard: 'ctrl+y',
    execute() {
        return {
            redo: () => {
                const queueItem = state.queue[state.current + 1]
                if (queueItem) {
                    queueItem.redo()
                    state.current++
                }
            }
        }
    }
})
```

4.实现组件拖拽撤回、重做功能

```
export function useBlockDragger(focusData, lastSelectBlock, data) {
    let dragState = {
        startX: 0,
        startY: 0,
+       dragging: false // 默认不是正在拖拽中
    }
    const mousedown = (e) => {
        const { width: BWidth, height: BHeight } = lastSelectBlock.value
        dragState = {
+           dragging:false,
            // ...
        }
        events.emit('start')
    }
    const mousemove = (e) => {
+      if (!dragState.dragging) {
            dragState.dragging = true; // 确实开始拖拽了
            events.emit('start');
        }
    }
    const mouseup = (e) => {
        document.removeEventListener('mousemove', mousemove);
        document.removeEventListener('mouseup', mouseup)
        markLine.x = null;
        markLine.y = null;
+      if (dragState.dragging) { // 触发end事件
            events.emit('end');
        }
    }
    return {
        mousedown,
        markLine
    }
}
```

5.快捷键设计

```
const keyboardEvent = (() => {
    const keyCodes = {
        89: 'y',
        90: 'z'
    }
    const onKeydown = (e) => {
        const { keyCode, ctrlKey } = e;
        let keyString = [];
        if(ctrlKey) keyString.push('ctrl'); // 如果按住ctrl
        keyString.push(keyCodes[keyCode]); // 按住的keycode是谁
        keyString = keyString.join('+'); // 用+拼接
        state.commandArray.forEach(({keyboard,name})=>{ // 循环注册的命令
            if(!keyboard) return; // 没有按键的命令跳过
            if(keyboard == keyString){
                state.commands[name](); // 执行对应的命令
                e.preventDefault();
            }
        })
    }
    const init = () => {
        window.addEventListener('keydown',onKeydown); // 绑定事件
        return () => { // 卸载事件
            window.removeEventListener('keydown', onKeydown)
        }
    }
    return init
})();
;(() => {
+   state.destroyList.push(keyboardEvent());
    state.commandArray.forEach(command => command.init &&
state.destroyList.push(command.init()))
})();
```