

在编辑器界面通过拖拉，输入等做了一个海报，点击保存按钮后，到底保存了什么？

保存的就是一堆组件(html元素)的CSS或者非css的属性

组件(html元素)被单独抽离出来形成一个组件库

CSS或者非css的属性连同组件的类型被保存在数据库中

据组件类型去组件库中找组件，然后在组件上加上属性就会在编辑器中显示出海报的整体页面来

一:关于组件库的设计

两个项目怎样重用组件

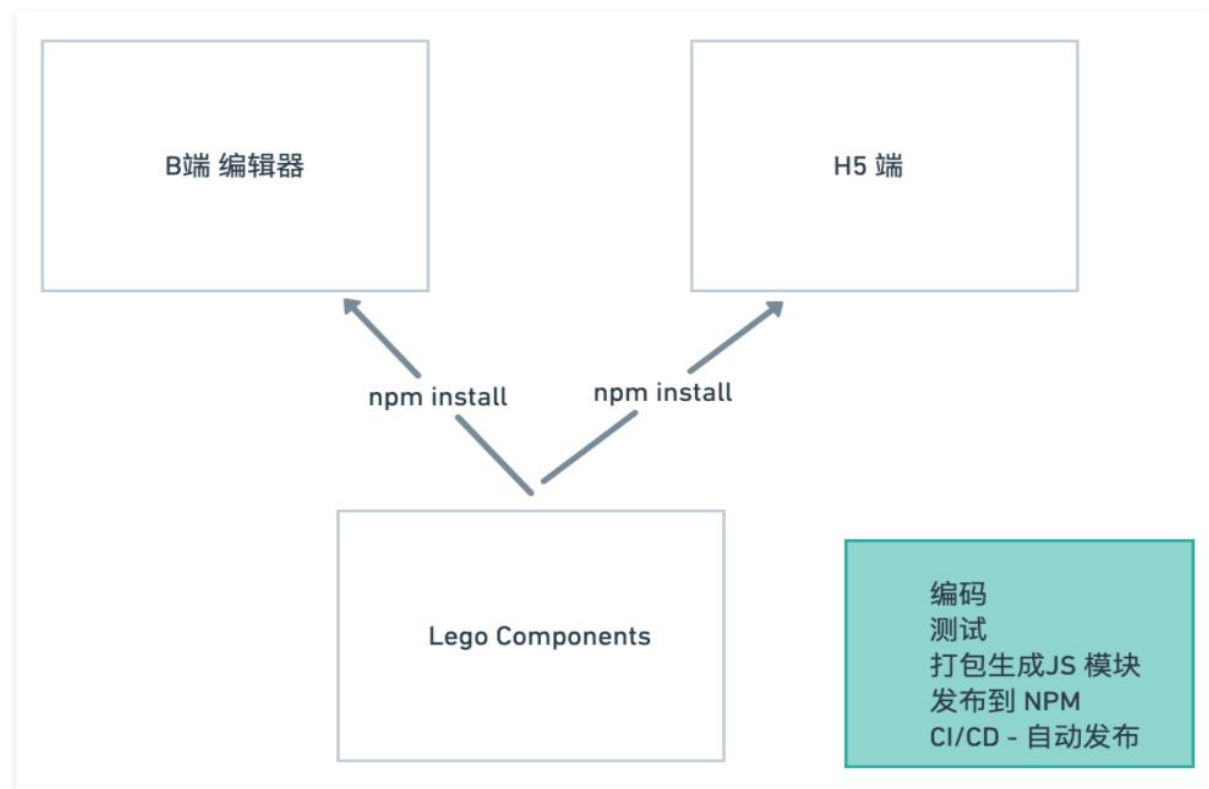
组件的属性应该怎样设计

组件的扩展性怎样保持

第一个问题：

在不同的项目中重用这些组件，所以把这些组件自然而然的抽取成一个代码库。降低和任何一个项目的耦合性，单独代码库让业务组件有独立的标准开发流程：

- 编码
- 测试
- 打包生成通用 JS 模块
- 发布至 NPM
- CI/CD - 自动发布



第二个问题:组件的属性应该怎样设计

原则

业务组件库大多数都是展示型组件，其实就是把对应的 template 加上属性（大部分是 css 属性）展示出来，会有少量行为，比如点击跳转等，而且这些组件会在多个不同的端进行展示，**所以业务组件库就是从简的原则，必须避免和编辑器编辑流程的耦合。**

组件命名

使用一个字母（L 代表乐高）加组件的名称：比如 LText 或者 l-text

组件分类

基础组件

- 1. 文本
- 2. 图片（用户主动上传的图片，支持 gif）
- 3. 形状

通用属性

这些组件都拥有的属性, 分为几组

尺寸：

- 长度 - 输入数字（同下面5项）
- 宽度
- 左边距
- 右边距
- 上边距
- 下边距

边框：

- 边框类型 - 无 | 实线 | 破折线 | 点状线 下拉菜单
- 边框颜色 - 颜色选择
- 边框宽度 - 滑动选择
- 边框圆角 - 滑动选择

阴影与透明度

- 透明度 - 滑动选择 100 - 0 倒排
- 阴影 - 滑动选择

位置


- X 坐标 - 输入数字
- Y 坐标 - 输入数字

事件功能

- 事件类型 - 无 | 跳转 URL 下拉菜单
- url 地址 - 输入框

特有属性

文本

- 文字内容 - 多行输入框
- 字号 - 输入数字
- 字体 - 宋体 | 黑体 | 楷体 | 仿宋 ... 下拉菜单
- 加粗 -  特殊 checkbox

- 斜体 - 同上
- 下划线 - 同上
- 行高 - slider
- 对齐 - 左 | 中 | 右 radio group
- 文字颜色 - 颜色选择
- 背景颜色 - 颜色选择

图片

- 上传图片 - 上传以及编辑控件

形状

- 背景颜色 - 颜色选择

高级组件

日期显示

特有属性

- 样式 - 下拉菜单 1 | 2
- 时间 - 默认为当前日期 日期选择器
- 字体颜色 - 颜色选择器

业务组件



```
1 // 方案一，将 css 作为一个统一的对象传入
2 <LText
3   css={{color: '#fff' ...}}
4   text="nihao"
5 />
6 // 内部实现比较简单
7 <p style={props.css}></p>
8
9 // 方案二，将 所有属性全部平铺传入
10 <LText
11   :text="nihao"
12   :color="#fff"
13   ...
14 />
15 // 内部实现会复杂一点
16 const styles = stylePick(props)
17 <p style={styles}></p>
18
19 // 方案一内部实现简单，但是保存的时候要多一层结构，并且更新数据的时候要知道是样式还是其他属性
20 // 方案二 内部实现稍微复杂一点，但是保存简单，更新数据不需要再做辨别
21 所以我觉得第二种解决方案是更好的，当然你也可能总结出更好的方案，就看大家自己怎样分析这个问题了
```


文本组件 独有属性

共有属性

图像组件 独有属性

另外一个维度：

这些组件目前有一些共有的属性，称之为公共属性。提到公共属性我们就要注意代码重用的问题。

<> 代码块

```
1 // 比如 在 Ltext 和 LImage 中都点击跳转的功能，属于公共属性的行为
2 // 抽象出一些通用的函数，在组件中完成通用的功能
3 import useClick from 'useClick'
4
5 useClick(props)
6
7 //这里只是分析可能遇到的问题，并不会写具体的代码，都用伪代码代替。
8
```

第三个问题：组件扩展性的问题

组件扩展性在业务组件库中不存在什么问题，因为每个组件都是独立的个体，它们的实现方案也相对独立，那么这里的扩展性是指在编辑器中是否能对它进行适配，包括展示和编辑的适配，所以这个问题可以合并到编辑器中的功能去讨论。

暂时还不用知道，后面会再来讨论这个问题

二， 关于组件属性数据结构的设计

1个海报是个整体，它由很多个不同的组件组成。

每个组件是一个object

每个组件都有一堆的属性，这些属性必须要放在object里面保存，因为它们都是键值对

然后每个组件都有id，跟组件的类型type

用伪代码整理思路

```
interface EditorStore {
  components: ComponentData[];
  //
  currentElement: string;
}

interface ComponentData {
  props: { [key: string] : any };
  id: string;
  type: string;
}

const components = [
  { id: '1', type: 'l-text',
    props: { text: 'hello', color: 'green' } },
  { id: '2', type: 'l-text',
    props: { text: 'hello2', color: 'purple' } },
]

components.map(component => <component.type { ...props } />)
```

模版文本二

模版文本二

左侧模版

渲染左侧预设组件模版

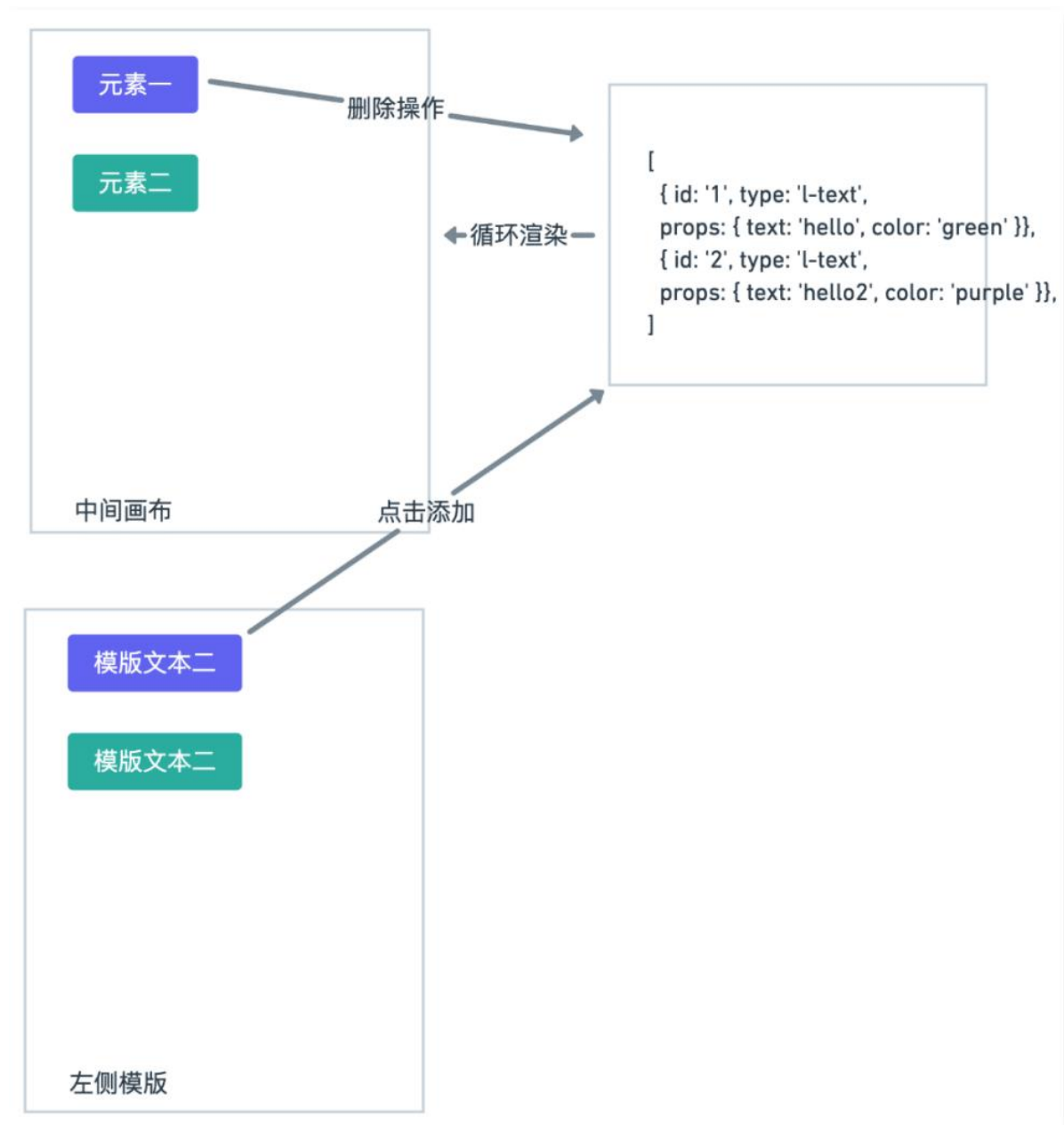
原理和上面一样的，只不过数据是预设好的，这个可以写死在本地，也可以从服务器端取得。他们和中间元素不一样的是，这些组件都有一个点击事件，我们可以添加一层 wrapper 来解决这个问题。这样也可以和内部的 lego components 做到隔离，互不影响。

<> 代码块

```
1      components.map(component => <Wrapper><component.name {...props} /></Wrapper>)
```

2

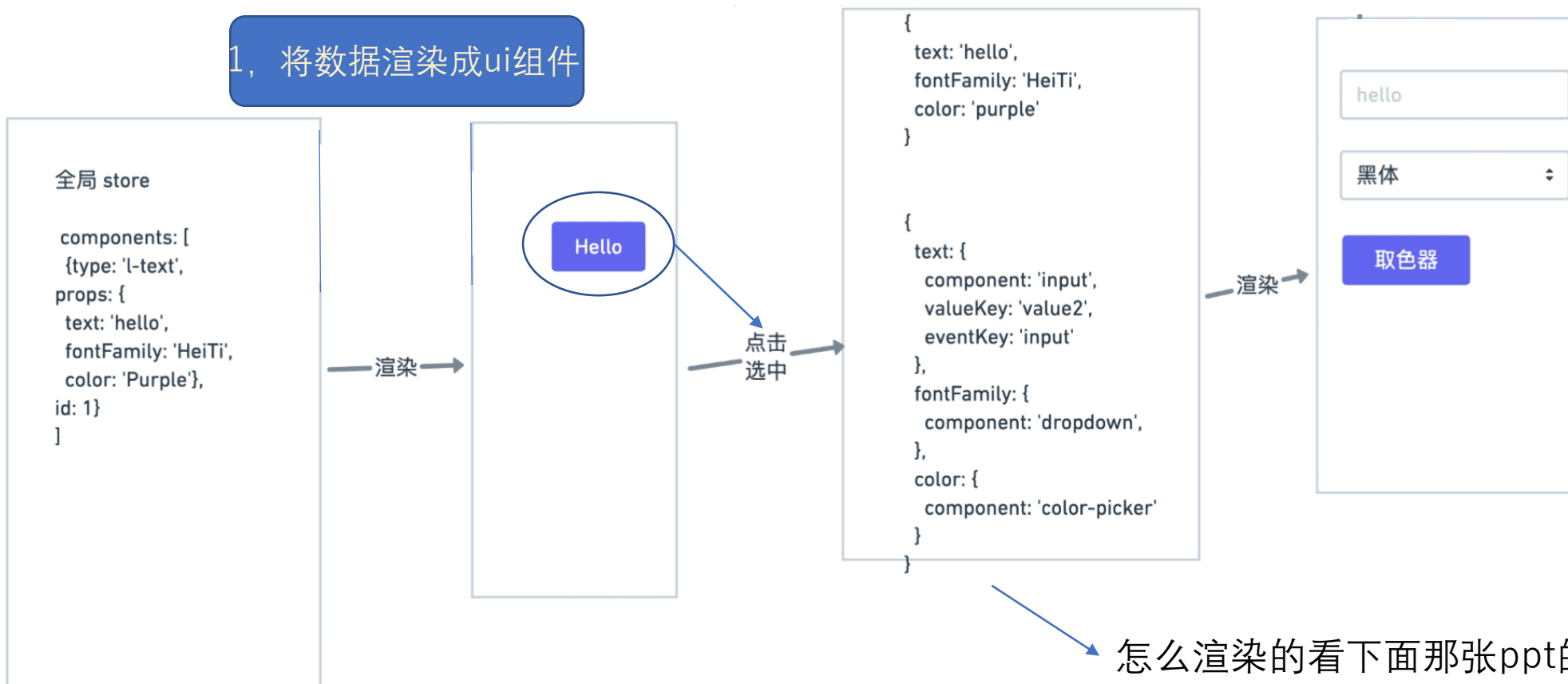
添加删除操作



Ui表单的渲染

2, 根据ui组件属性, 将属性渲染成ui的属性表单

1, 将数据渲染成ui组件



```
const textComponentProps = {  
  text: 'hello',  
  fontFamily: 'HeiTi',  
  color: '#fff'  
}
```

```
const propsMap = {  
  text: {  
    component: 'input'  
  },  
  fontFamily: {  
    component: 'dropdown',  
  },  
  color: {  
    component: 'color-picker'  
  }  
}
```

```
map(textComponentProps, (key, value) => {  
  <propsMap[key].component value={value} />  
})
```


组件属性的更新

表单里修改之后

