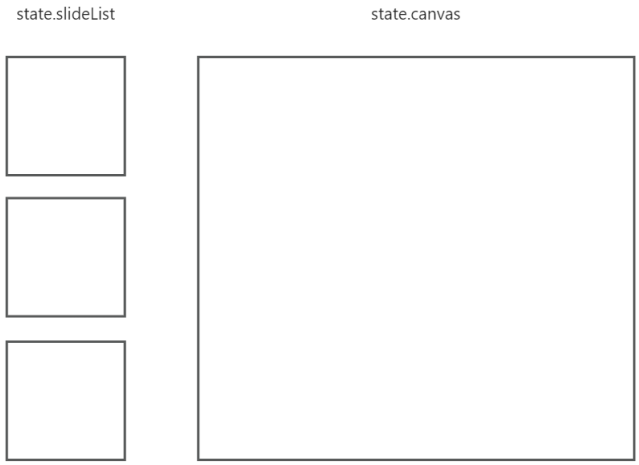
op改造FAQ

1. op是什么

op是operation的简称，中文叫操作。在新一轮的设计中，我们希望所有对课件的编辑行为，都能够被表达为一个或N个op的集合。

1. op是如何工作的

在举例之前，我们先搞清楚新的视图模型（view-model）是怎么样的

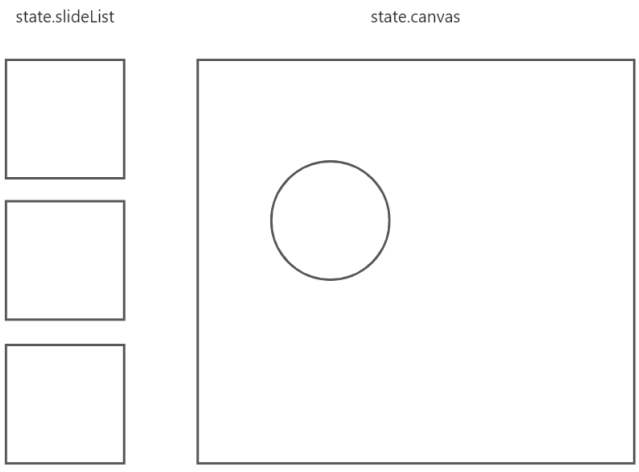


这里的state.canvas是一个新的数据模型，它的type是ISlide。

ps：在过去的设计中，画布的数据=watch(state.slideList, state.slideIndex) return deepClone(state.slideList[state.slideIndex])，这样做的目的是我们要始终保证画布的数据和state.slideList的数据一模一样。watch看似合理却也带来了其他问题，比如我们在上次评审中讲过的多次重绘的问题。除了watch，还有其他办法能始终保证两边的数据一模一样吗？是有的，假设我们想让A,B两个数据模型始终一模一样，只需要它们的初始值一样，然后对A的任何操作，在B上也操作一遍，就可以了。按照这个思路，我们在新版本里设计了state.canvas，画布的数据使用state.canvas（不再watch），和state.slideList隔离，以解决多次重绘的问题，对state.canvas的操作会通过op通知state.slideList，以保证两边的数据一模一样，这就是我们新一轮设计的核心。

搞清楚了新的视图模型，我们再来看几个例子。

举例一：新增一个形状

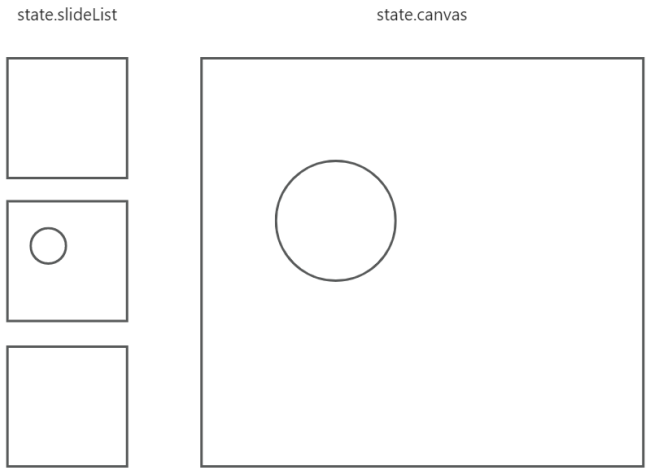


1. 第一步，当我们在state.canvas上新增一个形状的时候，state.canvas的数据首先被改变了（vue的mvvm）
2. 第二步，我们可以选择是否通过op通知state.slideList（某些情况下，我们可能不想通知state.slideList，比如连续移动/输入等）
3. 第三步，假设我们想通知state.slideList，可以通过api提交一个op，这个op大概是这样的

```
api.addDraftWithHistory([
  {
    type: 'insert_element',
    path: Path,
    value: PPTElement[]
  }
])
```

a. api.addDraftWithHistory() 是提交op的方法之一，它不仅会更改state.slideList的内容，还会将这个op存到history里，以便撤销/重做，撤销/重做的具体实现会在后面的FAQ里说明；在某些情况下，你可能想要提交一个不能被撤销/重做的op，需要使用api.addDraftWithoutHistory()
b. 有些编辑行为可能需要同时改变多个状态（比如框选多个元素移动），所以addDraftWithHistory()的传参是一个op数组
c. 新增一个形状的op，有三个携带信息，分别是type,path和value，type标明op的【类型】，path标明数据的【位置】，value标明数据的【内容】。op被提交之后，我们会“依样画葫芦”，对state.slideList执行一遍insert_element，以保证两边的数据一模一样，依的“样”就是path，画的“葫芦”就是value

完成以上三步以后，我们就能在缩略图上看到这个形状了

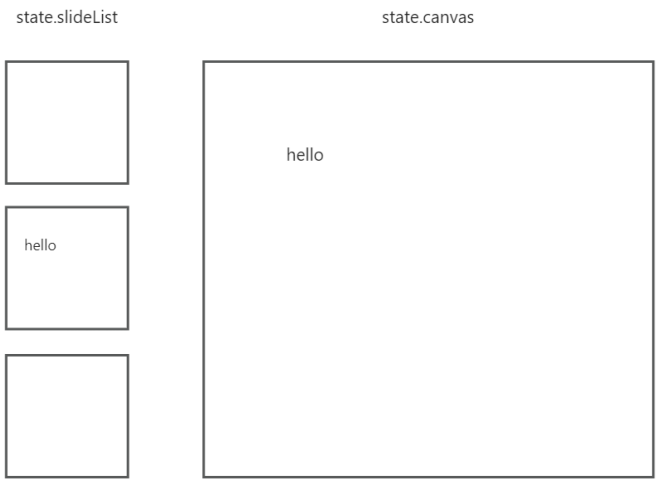


我们再来梳理一下关键点：

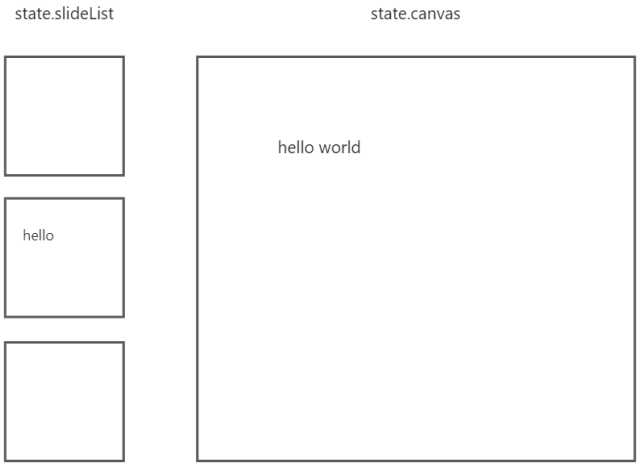
1. 在过去的设计中，我们总在操心state.slideList是否正确以及该如何和何时更改它，因为画布的数据是watch和clone state.slideList的，而在新的设计中，我们不再需要思考这些“繁重”的内容，而是回归到编辑本身，思考一个编辑行为发生的时候，它该如何被表达出来即可，这是一种新的看问题的角度。
2. 新的设计带来了很大的心理减负，我们不再需要调用各种api，例如updateElement、addSnapshot等，修改数据的过程都被封装在op内部，对组件而言，只需要专注自己的逻辑和在适当的时候提交op即可。（ps：在我看来，提交op更像是一种附属操作，它不会影响组件的逻辑，就算不提交op，也丝毫不影响对组件的编辑，它只会影响在重新加载该页的时候，是否符合我们的预期）。
3. 新的设计带来了更高的性能，比如编辑一段文字，我们可以将path定位到要修改的段落，将value定位到要修改的内容，这样一个op在被patch到dom的时候，js的计算时长是微秒级的（1s = 1000ms = 1000 * 1000us）。同时，新的设计也给协同铺垫了基础，简单的说，协同就是从远端接收op。

举例二：修改一段文本

我们再来举一个修改的例子，它和新增形状差不多



1. 第一步，我们先修改state.canvas的数据，让它变成hello world（vue的mvvm），在输入过程中，我们是不需要提交op的



1. 第二步，输入完成，提交一个op，这个op大概是这样的

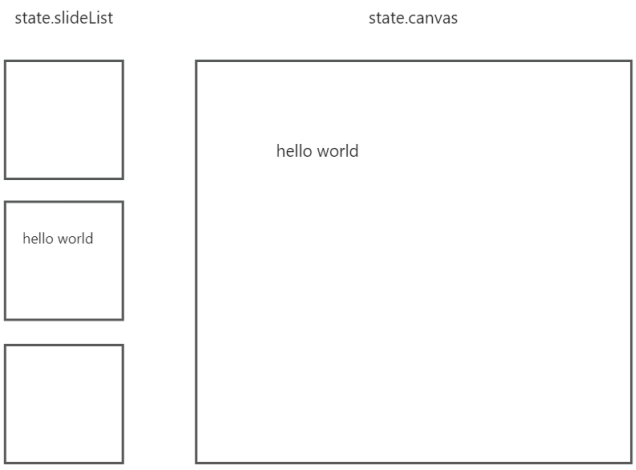
```
api.addDraftWithHistory([
  {
    type: 'set_element',
    path: Path,
    value: Partial<PPTElement>,
    newValue: Partial<PPTElement>
  }
])
```

a. 和上面稍有不同，这个op的携带信息除了type, path和value，多了一个newValue，它标明path【位置】的value【内容】被修改为了newValue【新内容】。似乎没什么难理解的，这里想说的是value和newValue分别从哪来？很简单，在state.slideList里面按path取值就是value，在state.canvas里面按path取值就是newValue，可以想一下为什么？
b. api提供了根据root和path取值的方法，大概是这样的

```
const value = api.getNode(state.slideList, path)
const newValue = api.getNode(state.canvas, path)

api.addDraftWithHistory([
  {
    type: 'set_element',
    path: Path,
    value,
    newValue
  }
])
```

通过以上简单的几行代码，我们就能在缩略图上看到文本被修改了



1. Path是什么

在上次评审中我们讲过，为了更快的找到需要被修改的节点，我们设计了Path模型，它是一个简单的string|number[]，默认[]表示state根节点，['slideList']就是state.slideList节点，['slideList', 0]就是state.slideList[0]节点，['slideList', 0, 'slideData']就是state.slideList[0].slideData节点。

1. 撤销/重做是怎么做的

撤销/重做有两种思路，一种叫快照，记录某一时间的系统状态，恢复上一时间的系统状态；另一种叫逆向操作，例如新增一个元素，撤销就是删除这个元素。在过去的设计中，我们一直使用快照的方式，不过它有两个弊端，一是内存问题，我们的系统状态（也就是state数据）可能很大，每次保存系统状态都需要大片的内存开销（我们也解释过，已经利用hash算法做过一轮优化，不过依旧至少要copy一份state数据），二是不利于协同，协同时总要传输和diff整个系统状态。

在新的设计中，我们使用逆向操作的方式，具体表现为：

a. 我们规定op要么是互逆的（比如insert和remove），要么是自逆的（比如set要传oldValue和newValue，move要传oldPath和newPath）
b. 使用addDraftWithHistory提交op以后，op会被存到history里，在执行撤销的时候，会先取op的逆（op的逆还是一个op，insert就remove，set就调换oldValue和newValue的值），再执行op的逆，重做是一样的道理

op的内存开销比系统状态要小的多。

1. 我们目前有哪些op——广源
2. 我们目前有哪些api——平姐
3. 代码示例——小乔