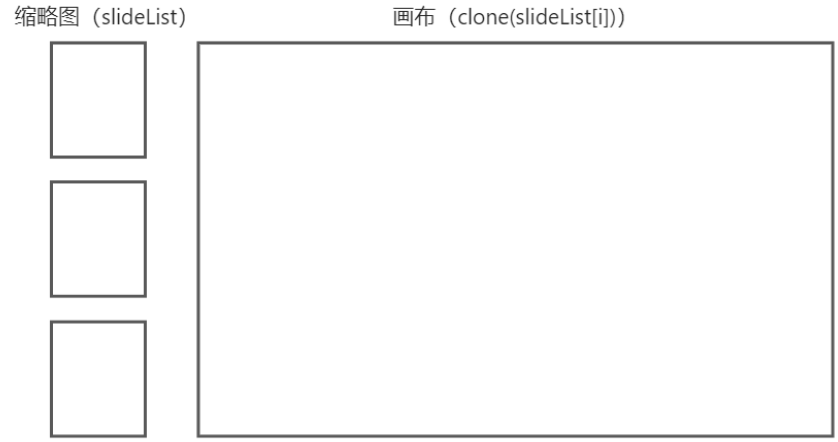
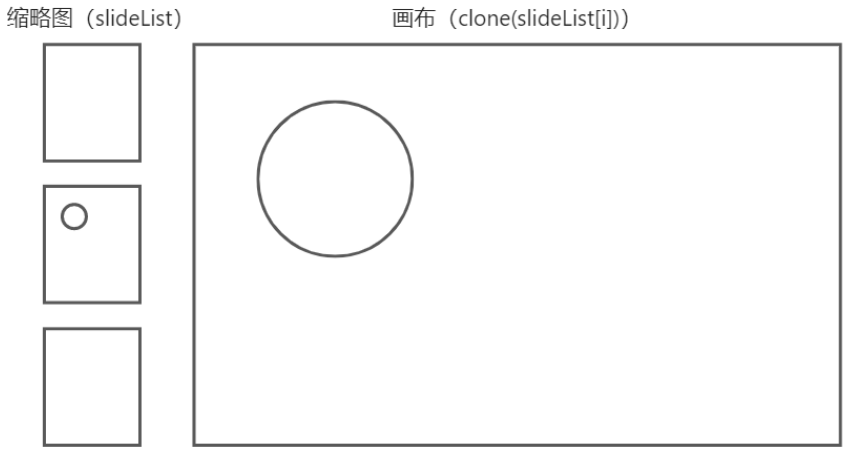
对画布、缩略图、快照更新机制的改进

现状

1. 现在画布是如何更新的，问题是啥？
2. 现在缩略图是如何更新的，问题是啥？
3. 现在快照是如何更新的，问题是啥？

1. 现在画布是如何更新的，问题是啥？





1. 更新画布
2. 更新 slideList: API.updateElement() -> get oldShape by id -> merge oldShape & newShape -> Object.assign(oldShape, merge)
3. watch slideList[i]
4. 更新画布 clone(slideList[i])

Q：为什么更新画布两次？因为撤销/重做的时候，只会更新 slideList，所以画布要 watch slideList[i]，而更新元素也要同步更新 slideList

Q：第一次更新画布是局部重绘，第二次更新画布 clone(slideList[i]) 是整体重绘

Q：画布可以不 clone 吗？或者可以 clone 缩略图吗？

1. 现在缩略图是如何更新的，问题是啥？

参照上面的过程，缩略图通过 Object.assign(oldShape, merge) 更新

Q：更新没有问题，除了在某些情况下，比如移动元素的时候，调用的是 update_slide_with_shapelist()，对 slideList[i] 进行了整体重绘，算是一个可优化项

Q：缩略图的问题在于初始渲染（比如 100 页的课件），目前我们采用的策略是只渲染可视区域，但依然有一些问题，比如上下滑动的时候，会重新加载新的缩略图，加载较慢

1. 现在快照是如何更新的，问题是啥？

在更新 slideList 的时候，会调 add_snapshot()，现在的 snapshot 是这么管理的：

1. 对整个 slideList 缓存：先把每个 slideList[i] 转成字符串，然后计算字符串的 hash，然后存储 { key: hash, value: JSON.stringify(slideList[i])}
2. add_snapshot() 的时候，再执行一遍上诉过程，如果 hash 没变，证明 slideList[i] 没变，如果 hash 变了，就新增一个 { key: hash, value: JSON.stringify(slideList[i])}
3. 用一个数组保存每步的 hash 集合
4. 撤销/重做的时候，先找到 hash 集合，再根据 key 取出来 value，最后根据类型（新增/删除/修改）修改 slideList

Q：计算 hash 的耗时（20-200ms）：目前我们把 add_snapshot() 从同步改为异步，缓解了计算 hash 会阻塞渲染的问题，但没有彻底去掉这个耗时

Q：内存问题：现在的策略已经考虑到对整个 slideList 缓存是非常耗内存的，所以用 { key: hash, value } 的方式来存储数据，但依然不得不至少 clone 一份 slideList 数据

Q：现在撤销/重做的时候，是对 slideList[i] 的整体重绘（新增/删除可以，但修改不需要）

改进

我们希望：

1. 更新画布的时候，不需要更新两次，也不要整体重绘
2. 缩略图的初始渲染，保留只渲染可视区域的策略，同时，解决加载较慢的问题
3. 去掉计算 hash 的耗时
4. 解决 clone 一份 slideList 的内存问题
5. 撤销/重做的时候，不需要对 slideList[i] 整体重绘

设计

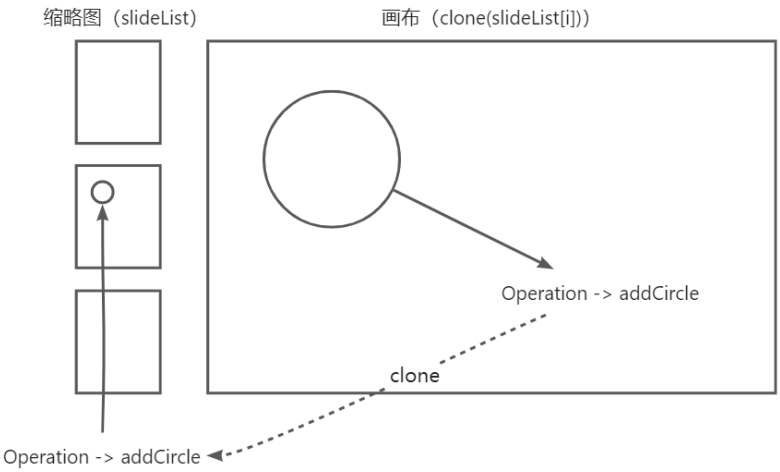
增加以下设计：

1. 操作-Operation

把操作作为更新 slideList（和其他 state 下的相关数据）的最小抽象

type	payload
新增幻灯片：insert_slide	节点：path，幻灯片：slide
删除幻灯片：delete_slide	节点：path，幻灯片：slide
修改幻灯片：update_slide	节点：path，修改前：slide，修改后：newSlide
移动幻灯片：move_slide	旧节点：path，新节点：newPath
...others	

举个栗子：在画布上新增一个形状，产生了一个操作，把这个操作应用到 slideList 上，缩略图也新增了这个形状

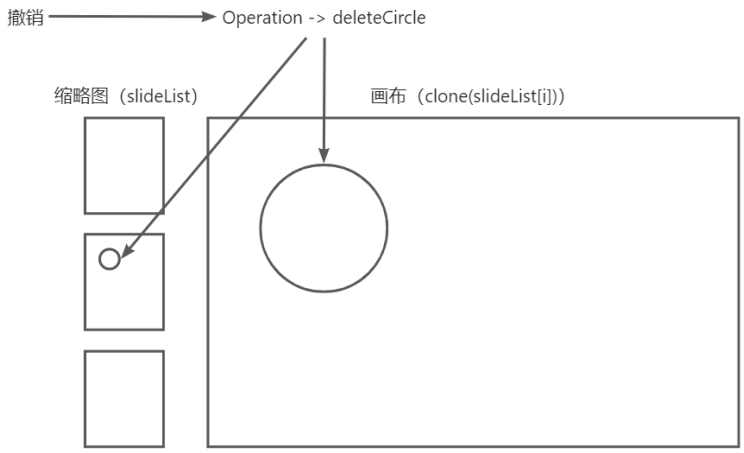


除了对更新视图友好外，对撤销/重做也非常友好（不需要计算 hash，不需要 clone slideList），此外，对协同也非常友好（如果以文档模型为最小抽象，协同时就绕不开“传输整个文档对象”和“计算本地文档和远端文档的 diff”两个过程）

packages > core > draft > src > TS operation.ts > OperationTransform

```ts
import { ISlide } from '../../types'
import { Path } from './path'

type InsertSlideOperation = {
  type: 'insert_slide'
  path: Path
  slide: ISlide
}
type DeleteSlideOperation = {
  type: 'delete_slide'
  path: Path
  slide: ISlide
}
type UpdateSlideOperation = {
  type: 'update_slide'
  path: Path
  slide: ISlide
  newSlide: ISlide
}
export type Operation = InsertSlideOperation | DeleteSlideOperation | UpdateSlideOperation

type OperationTransform = {
  inverse: (op: Operation) => Operation
}
export const operationTransform: OperationTransform = {
  inverse: (op: Operation) => {
    switch (op.type) {
      case 'insert_slide':
        return { ...op, type: 'delete_slide' }
      case 'delete_slide':
        return { ...op, type: 'insert_slide' }
      case 'update_slide': {
        const { slide, newSlide } = op
        return { ...op, slide: newSlide, newSlide: slide }
      }
    }
  },
}
```

为什么要提供一个 inverse 方法？因为在撤销的时候，我们需要执行逆操作



1. 节点寻址-Path

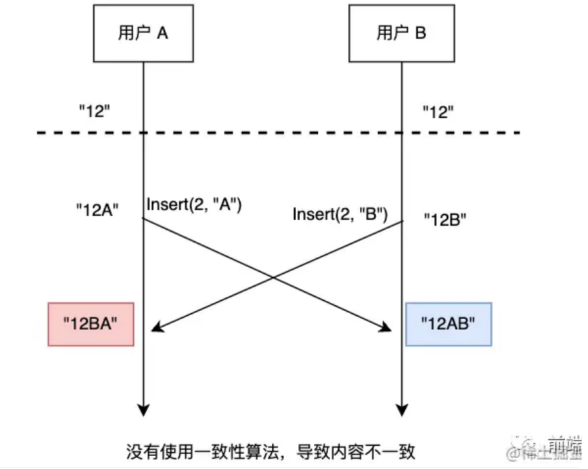
我们在新增/删除/修改元素的时候，只知道元素的 id，它【没有反应节点位置的能力】，如果我们要根据 id 来寻找节点，就要全量遍历当前的节点树，查询开销是非常大的，为此，我们设计了【Path】这个模型去描述节点位置，Path 是一个（string | number)[]

```
packages > core > draft > src > TS path.ts > ...
1  export type Path = (string | number)[]
2
3  type PathTransform = {
4    parent: (path: Path) => Path
5    last: (path: Path) => string | number
6  }
7  export const pathTransform: PathTransform = {
8    parent: (path: Path) => {
9      if (path.length == 0) throw new Error('path: cannot get the parent')
10     return path.slice(0, -1)
11   },
12   last: (path: Path) => {
13     if (path.length == 0) throw new Error('path: cannot get the last')
14     return path[path.length - 1]
15   },
16 }
17
```

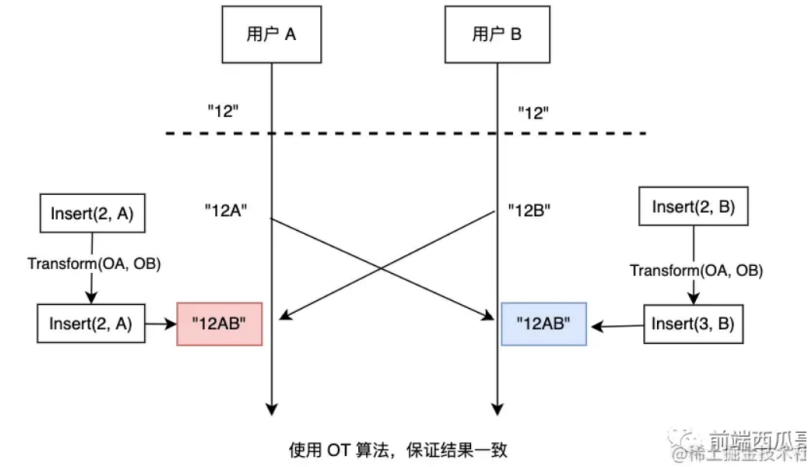
举个栗子：给定一个 Path，比如 ['slideList', 1]，我们从根节点 state 开始，沿着这个路径 state['slideList'][1]，就能很快的找到 slideList[1] 节点，同时我们还很容易获取它的祖先（state, state['slideList']）和兄弟（state['slideList'][2]）节点

我们会给 Path 提供一系列的工具函数，来处理各类复杂问题，比如：

1. 处理路径关系
2. 变更路径
3. 路径转换：路径转换非常重要，在协同时，如果两个用户 A 和 B 在同时操作一份数据，就有可能产生问题



怎么解决呢？



`Transfrom(OA, OB) = Path.normalize(OA, OB)`

Path 怎么获取呢？

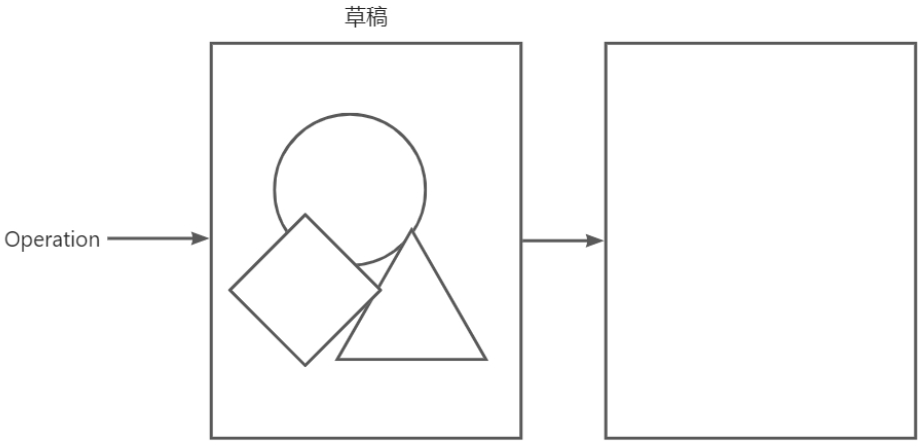
我们提供一个 get 方法来实现 id 与 path 的转换（暂时还没写），这个方法类似现在的 get OldShape by id，不过我们会做 memorize（函数缓存），以相同的 id 查询路径，将返回最近一次的结果

1. 草稿-draft

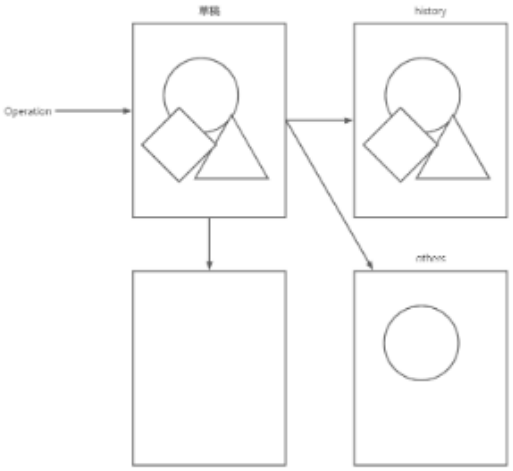
有些操作需要在更新视图的同时，记录它的状态，以便撤销/重做，有些操作只需要更新视图，不需要撤销/重做，我们该如何设计？

packages > core > draft > src > TS draft.ts > ...

```ts
import { Operation } from './operation'

export type Draft = {
  ops: Operation[]
  onChange: () => void
  apply: (op: Operation) => void
}

export function createDraft(): Draft {
  const draft: Draft = {
    ops: [],
    onChange: () => {},
    apply: (op: Operation) => {
      if (draft.ops.length == 0) {
        Promise.resolve().then(() => {
          draft.onChange()
          draft.ops = []
        })
      }
      draft.ops.push(op)
    },
  }
  return draft
}

```



看起来没什么用？



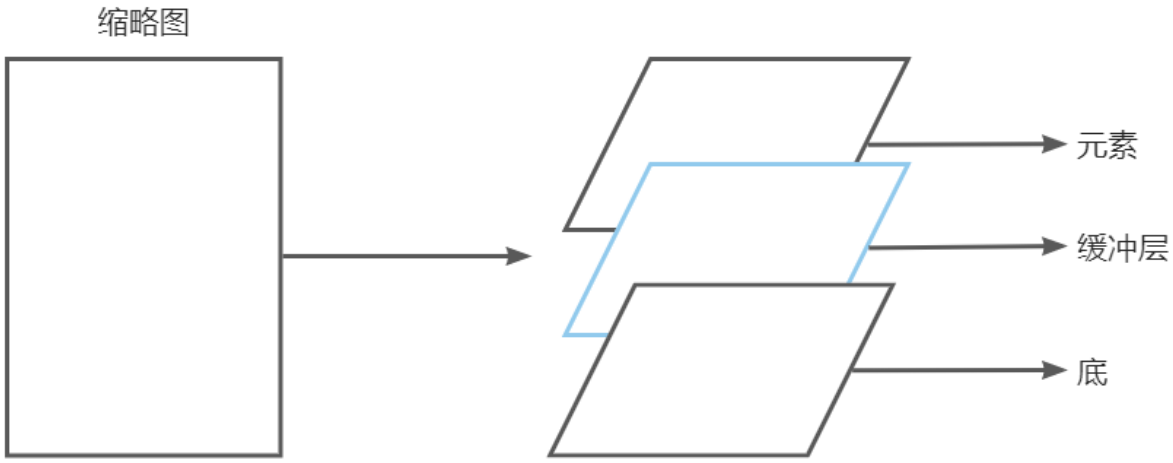
withHistory

packages > core > draft > src > TS history.ts > ...

```ts
import { Draft } from './draft'
import { Operation, operationTransform } from './operation'

type History = {
  history: {
    redos: Operation[][]
    undos: Operation[][]
  }
  clear: () => void
  redo: () => void
  undo: () => void
}

export function withHistory<T extends Draft>(draft: T): T & History {
  const d = draft as T & History
  const { apply } = d
  d.history = { redos: [], undos: [] }
  d.clear = () => {
    const { history } = d
    history.redos = []
    history.undos = []
  }
  d.redo = () => {
    const { history } = d
    const { redos } = history
    if (redos.length > 0) {
      const batch = redos[redos.length - 1]
      batch.forEach(op => apply(op))
      history.redos.pop()
      history.undos.push(batch)
    }
  }
  d.undo = () => {
    const { history } = d
    const { undos } = history
    if (undos.length > 0) {
      const batch = undos[undos.length - 1]
      const inverseOps = batch.map(operationTransform.inverse).reverse()
      inverseOps.forEach(op => apply(op))
      history.redos.push(batch)
      history.undos.pop()
    }
  }
  d.apply = (op: Operation) => {
    const { ops, history } = d
    const { undos } = history
    const lastBatch = undos[undos.length - 1]
    if (lastBatch && ops.length !== 0) {
      lastBatch.push(op)
    } else {
      undos.push([op])
    }
    while (undos.length > 21) undos.shift()
    history.redos = []
    apply(op)
  }
  return d
}

```

1. 缩略图的缓冲层

增加缓冲层（暂定 canvas，只绘制 shapeData 的形状），缓冲层基于脏路径（即 Operation 影响的路径）更新


缩略图
元素
缓冲层
底