

## 多人协同编辑技术的演进 2021-10-24



pubuzhixing  
执剑天涯、快意江湖

184 人赞同了该文章

多人协同编辑一直是我们的 [PingCode Wiki](#) 不太敢触碰的一个功能，因为技术实现上有挑战。但协同编辑技术本身已经发展多年，解决方案已经相对成熟，我们团队也是在刚刚结束的 Q3 里完成了基于 PingCode Wiki 编辑器协同编辑的方案落地，所以这里想结合我们的技术选型及落地实践经验谈谈我对这块技术的理解。

主要内容以协同编辑技术为主，中间也会谈谈对技术发展演进的理解。

### 一个场景

一个常见的场景，页面发布冲突，这个交互在我们产品中真实存在过



两个用户基于相同的文章内容进行了修改，一个用户先发布，后一个用户在发布的时候就会有这样的提醒，虽然有提示，这其实对用户来说是不友好的。

通常产品的解决方案有以下三种：

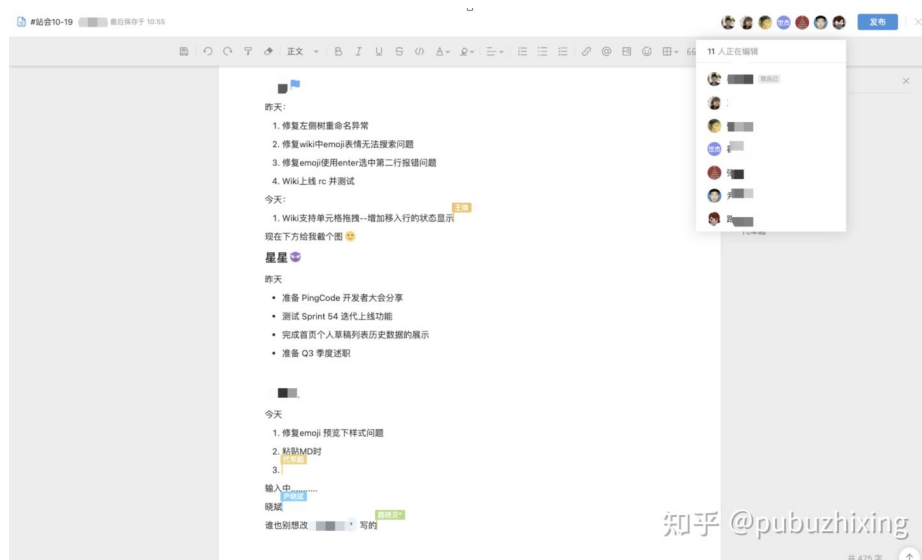
1. 悲观锁 - 一个文档只能同时有一个用户在编辑
2. 内容自动合并、冲突处理
3. 协同编辑

第二种方案也有国外产品在做就是 Gitbook



Gitbook 也是一种解决问题的方式。

然后下面我们产品协同编辑的最终交互截图：



主流的协同编辑交互就是这样，可以看到协作者列表以及每个协作者的正在输入的位置，实时看到他们输入了什么内容，我们甚至可以直接相互对话，这种方式可以有效避免冲突。

虽然协同编辑最终呈现给用户的就这一个界面，但是它背后却有复杂的技术作为支持，接下来就一起看看协同编辑是如何运作的。

## 认识协同编辑

**指导思想：** 系统不需要是正确的，它只需要保持一致，并且需要努力保持你的意图。

我觉得这句话可以作为协同编辑冲突处理的一个指导思想，它很简洁明了的阐述了一个事情，就是协同编辑的冲突处理不一定是完全正确的，因为冲突本身就意味着操作是互斥的，互斥双方的操作意图不可能完全保留。冲突处理最重要的是保证协同双方最终数据的一致性，然后在这个基础上努力保持各自的操作意图。

## 聊聊富文本数据模型

协同编辑是构建在富文本编辑器之上的技术，它的实现一定程度上依赖于富文本数据模型的设计，这里介绍两个比较有代表性的数据模型：

2012 年 Quill -> Delta

2016 年 Slate -> JSON

### Delta 数据模型

Quill 编辑器显示一段文字



它的数据表示是这样的

```
{
  "ops": [
    {
      "insert": "Hello "
    },
    {
      "attributes": {
        "bold": true
      },
      "insert": "Quill"
    },
    {
      "insert": "! "
    }
  ]
}
```

它定义三种操作（insert、retain、delete），编辑器产生的每一个操作记录都保存了对应的操作数据，然后用一些列的操作表达富文本内容，操作列表即最终的结果。

### Slate 数据模型（JSON）

模型定义:



编辑器中有一个图片类型的节点，对应的数据结构

```
[
  {
    "type": "image",
    "children": [
      {
        "text": ""
      }
    ],
    "width": 385,
    "height": 590,
    "thumbUrl": "https://atlas.pingcode.com/files/public/xxx",
    "originUrl": "https://atlas.pingcode.com/files/public/origin-url",
    "align": "center"
  }
]
```

知乎 @pubuzhixing

属性修改操作

```
[
  {
    "type": "set_node",
    "path": [
      0
    ],
    "properties": {
      "align": "center"
    },
    "newProperties": {
      "align": "right"
    }
  }
]
```

知乎 @pubuzhixing

我们可以看出虽然 Delta 和 Slate 数据的表现形式不同，但是他们都有一个共同点，就是针对数据的修改都可以由操作对象表达，这个操作对象可以在网络中传输，实现基于操作的内容更新，这个是协同编辑的一个基础。

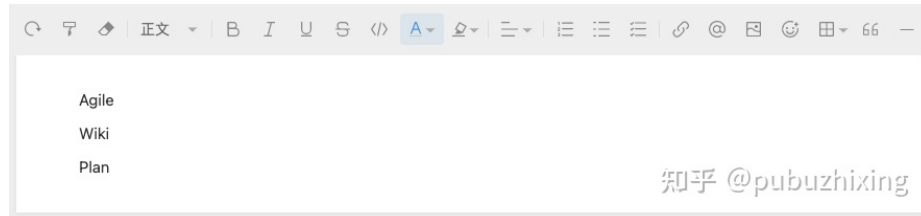
下面的部分我想聊聊在实现协同编辑时所面临的最核心的问题。

## 协同编辑面临的问题

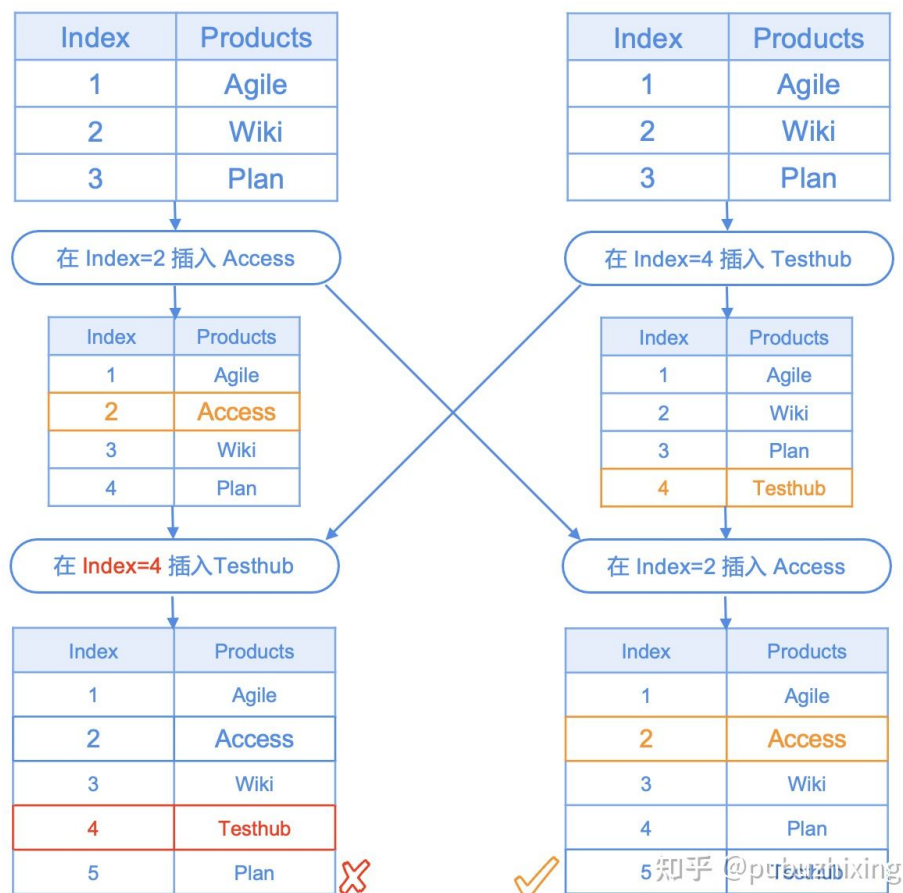
这里先抛出问题，带大家了解协同编辑所面临的问题的具体场景，从问题出发，而后再讨论解决方法。

### 问题一：脏路径问题

假如编辑器中有三个段落，如下图所示



这里用数组简单模拟上面的三个段落，下图展示了两个用户同时对数据修改产生的操作序列



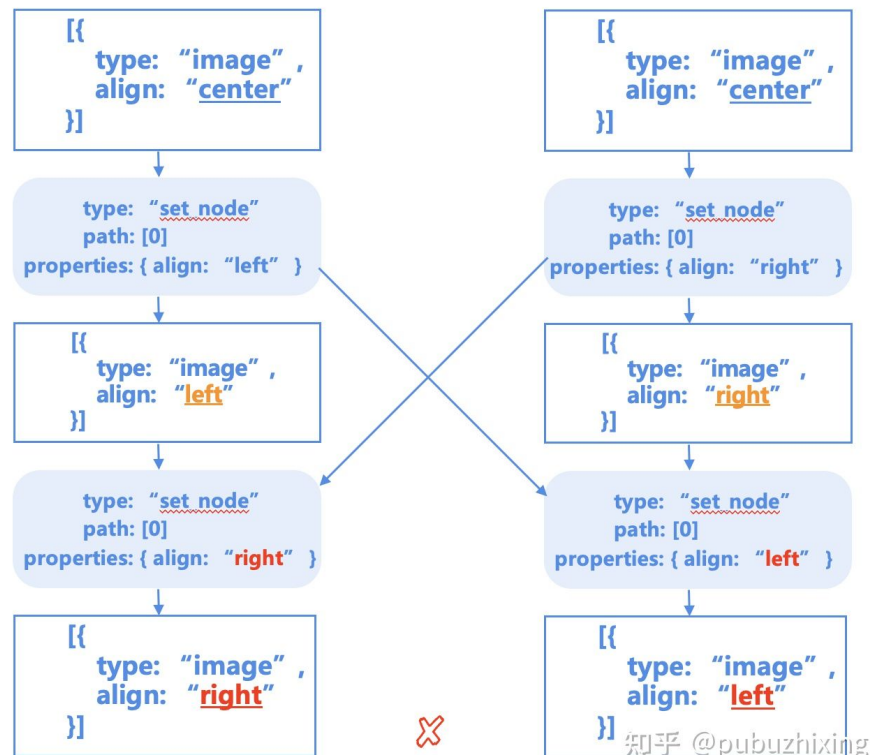
可以看到左边插入的段落「Testhub」插入位置是错误的

最上面的是原始的数据结构，左右两边代表两个用户的操作序列，开始时他们的状态一致。左边用户在 Index=2 的位置插入一个新的段落「Access」、右边用户在 Index=4 的位置插入一个新的段落「Testhub」，他们各自应用完自己的操作后，分别把操作通过消息服务传给对方，这个时候左边用户接收到右边用户同步过来的消息「在 Index=4 插入 Testhub」直接应用就会出现左边的结果，这个结果是与用户原本的意图是不一致的，而且与右边最终的数据不一致。

究其原因就是左边用户先进行的插入操作导致了它后面数据的索引发生变化，那么基于同步过来的操作直接应用就会出现上图的异常，我把这种情况称为脏路径问题。

### 问题二：并发冲突问题

这里以前面介绍的图片数据结构为例说明并发冲突的问题，下图展示问题出现的过程，为了方便表达，图片节点仅保留 type 和 align 两个字段



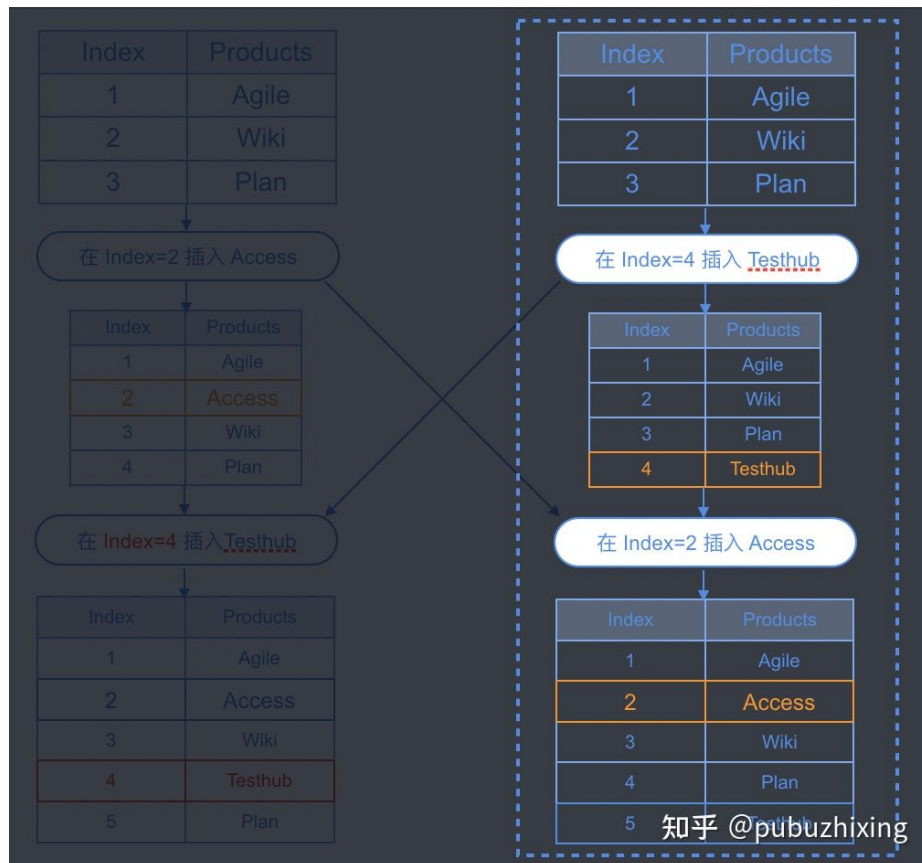
最上面的数据结构展示了两个用户开始时基于相同的状态，图片 align = 'center'。

左边用户修改 align 属性为 left、右边用户修改 align 属性为 right，按照默认处理他们把各自的操作通过消息服务传给对方，则会造成左边最终显示居右、右边最终显示居左，数据出现不一致，这种情况称为并发冲突，他们基于相同的位置修改了相同的属性。

### 问题三：undos/redos 问题

undos/redos 问题本质还是前面所说的「脏路径问题」+「并发冲突问题」，但是问题出现的场景有些不一样，又相对复杂，所以这里单独提出来了。

还是前面「脏路径问题」的数据操作，这里只看右边部分，分析它的撤回栈：



右边用户的操作列表：

序列	Origin	操作	反向操作
1	左边用户	在 Index=4 插入 Testhub	删除 Index=4 节点
2	右边用户	在 Index=2 插入 Access	删除 Index=2 节点

右边用户撤回栈（序列与操作列表相反）：

- ① 删除 Index=2 位置的 节点
- ② 删除 Index=4 位置的 节点

执行这种撤回逻辑其实是有问题，原因是撤回操作 ① 所对应的操作的触发者（Origin）是左边用户，如果按照这种撤回逻辑执行左边用户可能就蒙了：“我刚刚输入的内容怎么没了！”，虽然逻辑上可以解释，但它不符合用户的使用习惯，所以对于协同编辑场景：撤回应当只撤回自己的操作，协同者的操作应当被忽略。

右边用户撤回栈修复版：

- ① 删除 Index=4 位置的节点

可以看到撤回栈只包含右边的操作了，但是这又带来了另外一个问题，大家仔细观察可以发现现在 Index=4 对应的节点是「Plan」，这个时候撤回会把「Plan」删除掉，而右边用户在插入时插入的实际节点是「Testhub」，又出现了脏路径。

除了这种「脏路径」问题，「并发冲突」问题也会以类似的方式出现在，具体的逻辑就不再详细分析了。

撤回栈忽略协同者操作后，撤回栈中的操作路径会出现「脏路径」问题 + 「并发冲突」问题。

问题四：工程落地问题

这个问题比较好理解，就是协同编辑具体的落地问题：

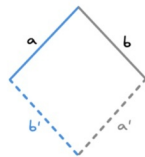
1. 操作的同步
2. 光标的同步
3. 网络不可知（网络抖动、网络延时、消息重连以及重连后的各种情况处理）
4. 文档版本历史
5. 离线编辑
6. ...

简单归纳下上面所提到问题，其实可以分为两类：

第一类：主要包含脏路径、并发冲突、Undos/Redos等，可以统称为**数据一致性问题**，它属于学术问题的范畴，因为并发冲突的处理结果需要保证**最终数据的一致性**，这个需要经过大量的学术研究、论证。

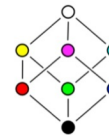
第二类：工程问题，重点是在解决「数据一致性」的基础上实现一套具体的落地方案，除了前面提到的具体落地开发的功能点，还要考虑性能问题、数据传输效率问题等，这块其实包含很大的工作量，是理论研究是否可以真正落地到生产实践的关键。

第一类学术问题的解决方案就是**数据一致性算法**，学术界主要有两个方面的研究：OT 算法 和 CRDT。



OT

操作变换



CRDT

无冲突复制数据类型

知乎 @pubuzhixing

下面我们简单介绍下这两种算法。

## 数据一致性算法

这里不会过多介绍算法的实现细节，只是提供它处理冲突的思路，以及从问题的本身出发去看待它处理问题的一个思路，至于具体的算法实现大家有兴趣可以去Github查找相关的资料去自己实践。

### OT

OT 全称是 Operational Transformation，它的核心思想是操作转换，通过转换数据修改操作解决协同编辑中的各种问题。

#### 发展历史

OT是最早（1989年）被提出的协同冲突处理算法

2006 年被应用到 Google docs

2011 年被应用到

Office 365



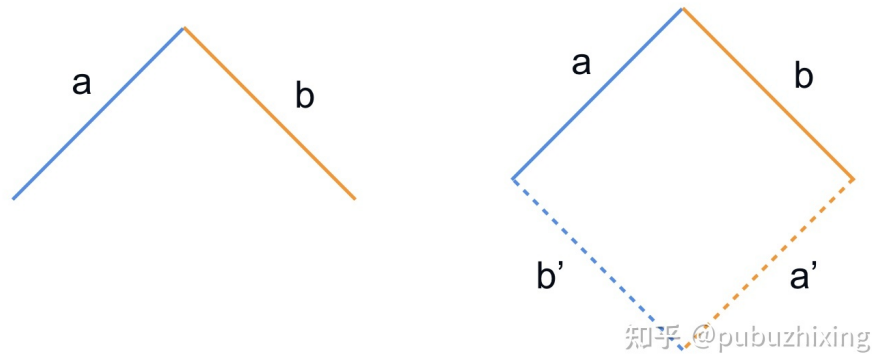
至今 OT 仍然是实现协同编辑的最主要的技术选择，Google docs 以及 Office 365 至今仍在采用 OT 的方案，国内近些年来的出现的一些文档类产品，包括石墨、钉钉、腾讯文档等等，他们的协同编辑技术也都是基于 OT 的。

### 核心思想

就像它的名称一样，它的核心思想是对用户协同编辑中产生的并发操作进行转换，通过转换对其产生的 **并发冲突** 和 **脏路径** 进行修正，然后把修正后的操作重新应用到文档中，保证操作的正确性和最终数据一致性。

### 原理图

可以用 diamond 图表示 OT 的核心原理



左图解释：

- a 标识左边用户的 operation
- b 表示右边用户的 operation
- 二者交叉的点表示文档基于相同的初始状态

左图状态：

两边用户分别应用操作 a 和 b 后，这时两边的文档内容都发生变化，且不一致；

操作转换：

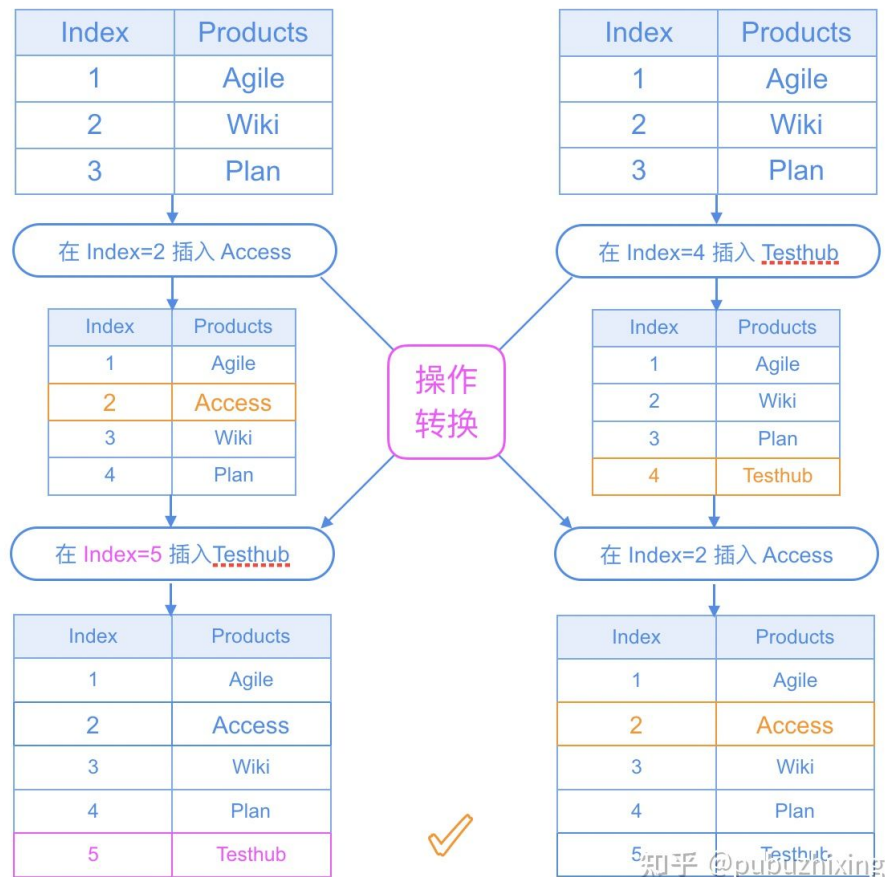
为了客户端和服务端的文档达到一致的状态，我们需要对 a 和 b 进行操作转换  $\text{transform}(a, b) \Rightarrow (a', b')$  得到两个衍生的操作作 a' 和 b'。

右图应用操作转换的结果：

左边用户的 a 操作的衍生操作 a' 在右边用户端应用，b' 在左边用户端应用，最终文档内容达到一致。

这里说明的只是最基础的 OT 模型，每个客户端只有一个操作的情况 (1:1)，还有每个客户端对应多个操作的情况 (M:N)，还有 OT 控制算法等等。并且在真正实现 OT 时有可能每一次操作转换只得到一个衍生操作 (ottypes 定义的操作变换就是这样)，跟前面的 transforms 有些不一样，但这些不是特别重要，具体实现的时候在仔细理解，这里描述的只是 OT 算法的最基础思路。

用 OT 解决「脏路径」问题



如上图所示 OT 在操作同步的过程中增加一层操作转换的逻辑，用于纠正并发操作产生的脏路径。

左边同步右边操作时索引由 4 转换为 5。

操作转换逻辑分析：

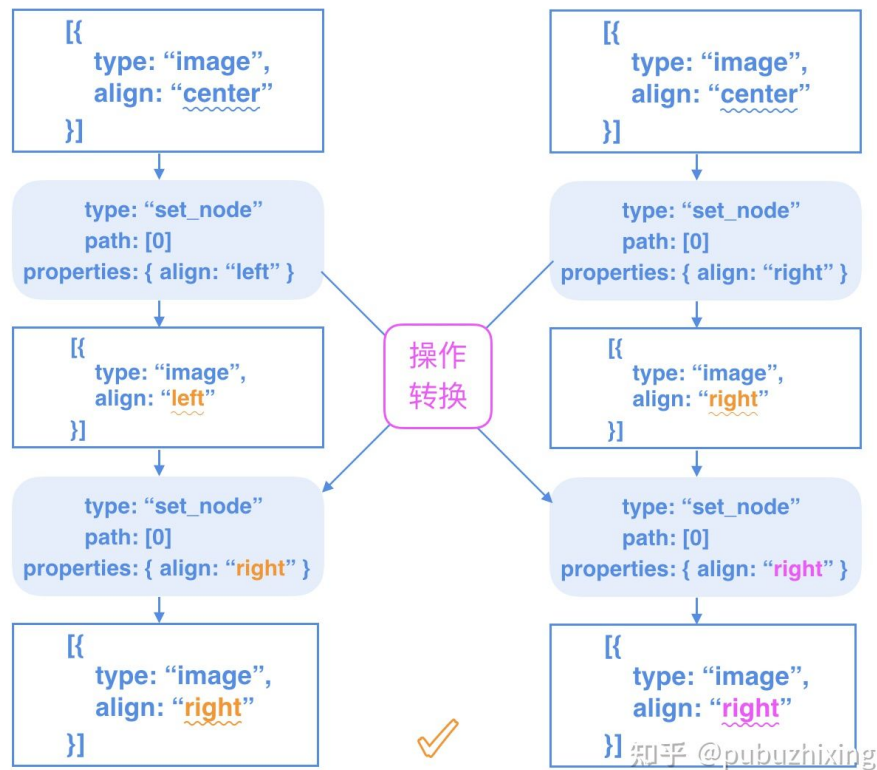
对于左边用户：

因为在协同操作「在 Index= 4 插入 Testhub」到达之前，已经执行了本地操作「在 Index=2 插入 Access」，而本地操作的索引 Index=2 小于协同操作的索引 Index=4，所以协同操作的索引路径应当加上本地新增的节点长度，也就是1，索引发生变化由 4 变成 5。

对于右边用户：

因为协同操作的索引路径小于本地操作的索引路径，本地操作不对协同操作产生影响，所以不需要做任何转换，直接应用源操作即可。

用 OT 解决「并发冲突」问题



可以看到基于 OT 解决「并发冲突」同样是使用操作转换逻辑，只不过这次的操作转换并不转换脏路径，而是协调冲突的属性修改，上图的处理结果是假定右边操作后到达服务器的，最终结果收拢到居右显示。

从上面的两种场景分析可以看出这个操作转换过程并没有太复杂，虽然真实的场景下要考虑的情况会比这要多，但是也就是一层逻辑转换。还有就是真实的场景需要对每一种操作类型做交叉操作转换，比如 Delta 支持三种操作，那么可能要支持 3 3 种操作变换，Slate 支持 9 种原子操作，可能要实现 99 种操作变换，复杂度大概就是这样。

### OT 解决 undos/reds 问题

前面已经说过 undos/reds 问题 本质就是「脏路径」+「并发冲突」问题，所以 OT 的处理方案就是当编辑器接收到协同操作时，需要对 Undo 栈、Redo 栈中的所有操作循环执行操作转换逻辑，undo 或者 redo 时最终执行的是转换后的操作，具体的逻辑不再意义赘述。

### 算法说明

可以看出 OT 是对编辑器的数据操作进行转换，所以 OT 算法的实现依赖于编辑器数据模型的设计，不同的数据模型需要实现不同的操作转换算法。

OT 算法大概就说到这里，下面看看 CRDT 是如何处理数据一致性问题的。

### CRDT

CRDT (Conflict-free Replicated Data Type)即“无冲突复制数据类型”，它主要被应用在分布式系统中，保证分布式应用的数据一致性，文档协同编辑可以理解成分布式应用的一种，它的本质是数据结构，通过数据结构的设计保证并发操作数据的最终一致性。

CRDT 于 2011 年正式被提出。基于 CRDT 的协同编辑框架 Yjs 大概在 2015 年开源，Yjs 是专门为在 web 上构建协同应用程序而设计的。

### 核心思想

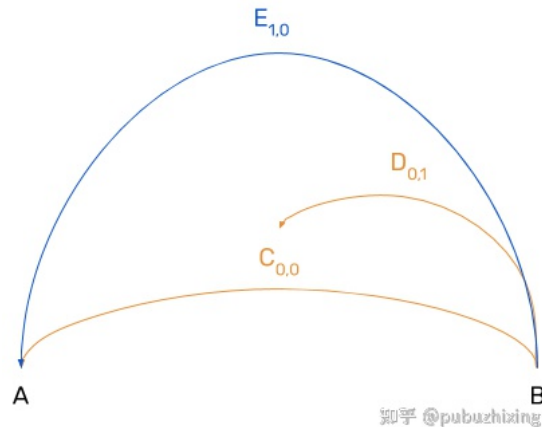
大多数的 CRDT 为在文档中创建的每个字符分配一个唯一的标识符。

为了确保文档始终能够收敛，CRDT 模型即使在删除字符时也会保留元数据。

CRDT 最初是为了解决分布式系统最终数据一致性而提出的，它支持各个主机副本之间数据修改的直接同步，而且数据修改的同步顺序以及同步的次数不影响最终结果，只要修改操作一致，数据的最终状态就是一致的，也就是通常大家说的 CRDT 数据的满足交换性和幂等性。

### 简单介绍 CRDT 是如何处理冲突的

下图描述了 Yjs 中处理冲突的算法模型，它是一个支持点对点传输的冲突处理模型。



上图基础说明

- 最下面的“AB”标识初始状态
- 上面的每一根线代表一个插入操作
- 每一个操作都有一个唯一标识符  
比如 C0,0 操作中的 0,0 就是一个标识符  
第一个 0 指示用户编号  
第二个 0 指示操作序列

例如，以下标识符表示 user 0 插入“C”在“A”和“B”之间

C0,0

相同的用户 user 0 插入“D”在“B”和“C”之间，可以使用下面的操作

D0,1

这时候另外一个用户期望插入“E”在“A”和“B”之间，但是这个操作是与前面插入“C”的操作（C0,0）是并发操作。

此时用户的唯一标识应该与前面的不同，但是 clock 应该是与前面的插入操作类似：

E1,0

由于存在并发冲突，Yjs 执行与 OT 相同的冲突解决，并比较各自插入的用户标识符。

由于用户标识符 1 大于 0，因此生成的文档为：

ACDEB

以上就是 Yjs 处理并发冲突的算法介绍，其实也不难理解，首先它的插入操作是基于已有字符的相对位置，在 OT 中使用的相当于是基于索引的绝对位置，然后就是冲突的处理，主要是比较用户标识符，标识符小的先应用，标识符大的后应用。

上面是以 Yjs 为例介绍 CRDT 的冲突处理模型，下面看看 CRDT 是如何解决前面所提出的问题的。

### 用 CRDT 的思想解决脏路径问题

首先我们使用类似于 CRDT 的方式描述刚才的数组：

List			List	
Index	Products		Unique Id	Products
1	Agile	+ 唯一标识 →	111	Agile
2	Wiki		222	Wiki
3	Plan		333	Plan

可以看到右边的列表使用唯一 Id 替换了原本数组的索引，然后描述内容修改的操作也相应的做一下调整

#### 左边操作：

在 Index=2 的位置插入 Access -> 在 111 之后插入 Access

#### 右边操作：

在 Index=4 的位置插入 Testhub -> 在 333 之后插入 Testhub

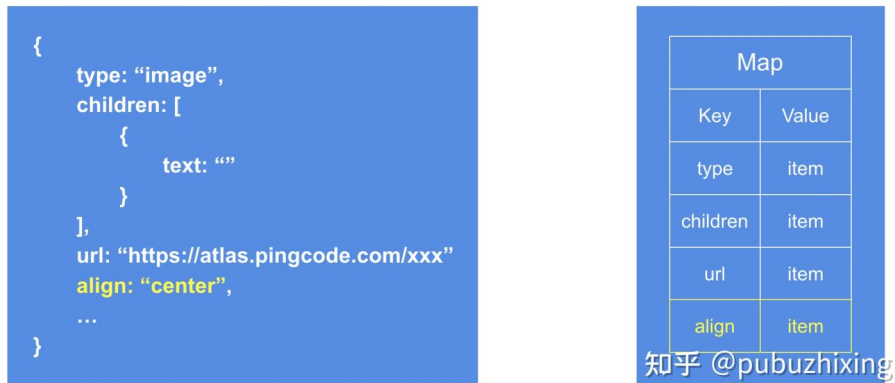
同步操作之后左边和右边最终的数据结构应该都是一样的：

Unique ID	Products
111	Agile
112	Access
222	Wiki
333	Plan
334	Testhub

因为这里只是模拟 CRDT，解释 CRDT 的思想，真实的 CRDT 通常是使用双向链表，这里为了好理解所以仍然沿用数组，只是给数组中的每一个段落节点数据增加一个唯一标识。

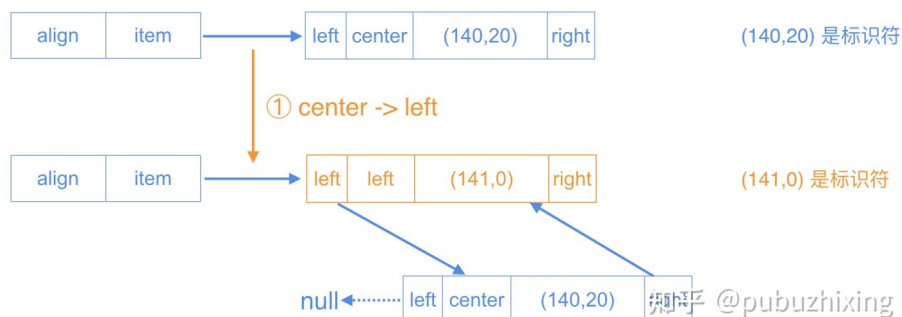
### CRDT 解决并发冲突

这里还是以图片设置 align 属性为例介绍，首先看看 CRDT 如何描述对象属性及属性修改：



左边是图片数据模型，右边是模拟 CRDT 对应的数据结构，图片对象中的每一个字段都使用结构对象去描述内容及内容的修改，这里以 align 字段的代表看它的表达

操作①：

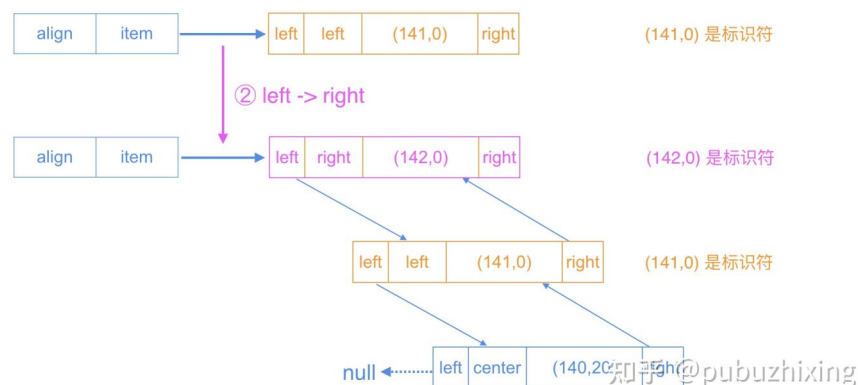


最上面蓝色部分表示 align 的初始值是 center，(140, 20) 是这个初始数据结构的标识，它也是基于某一个用户的操作产生的。

这个时候一个用户执行了操作①，把 align 属性修改为 left，产生了一个新的结构对象，就是图中橙色部分的表示。操作完成后，Map 中的 align 字段指向了新产生的结构对象上，标识符是 (141,0)，因为 (141,0) 这个结构对象是基于 (140,20) 的修改，所以它的 left 指向 (140,20) 这个结构对象。

这个示例会有一些歧义，就是链表的数据结构本身会有 left、right 两个指针（在结构对象左右两边），然后中间部分其实是内容，但是我的内容存储的是图片的 align 属性，它的值可能是 left、center、right，跟链表在 left、right 指针在一起可能产生混淆，这里标记下，就是结构对象中的第二个块描述的是属性内容。

操作②：



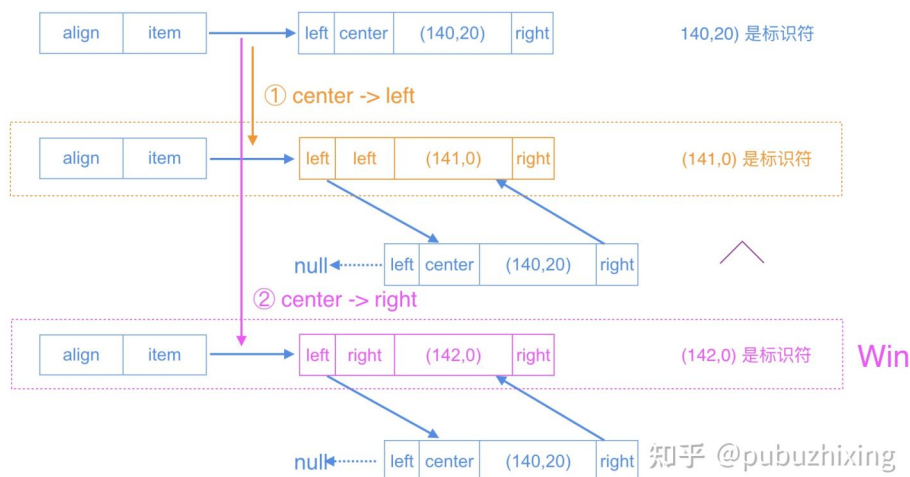
这个时候另外一个用户基于刚刚产生的结构对象 (141,0) 进行了操作②，把 align 属性修改为

right，产生了一个新的结构对象，就是图中橙红色部分的表示。

图片下半部分是这两个操作之后最终的数据结构，它是一个双向链表的表达（这种表达已经很接近 Yjs 真实的数据结构了），它不仅描述最终的数据状态（right），还可以表达出数据修改的顺序：center -> left -> right。

这个示例其实描述的是顺序操作，每一个操作基于的状态都是最新状态，两个用户执行的操作是有确定先后顺序的。

下面看看两个用户并发的执行属性修改时产生的数据结构：



与前面最大的不同就是执行操作 ② 和执行操作 ① 所基于的状态是一致的，都是基于 align = 'center' 进行修改的，这种情况表达的就是并发数据的修改。接下来就是并发处理的逻辑了，跟前面介绍的一致，这个时候操作 ① 的对应的用户标识 141 小于操作 ② 对应用户标识 142，所以先应用操作 ①，后应用操作 ②，所以最终图片的 align 属性状态是 right。

### CRDT 解决 undso/redos问题

CRDT 可以理解为完全没有「脏路径」问题，然后并发冲突问题也完全可以基于 CRDT 的标识符（时间戳）去解决，那么基于 CRDT 的方案中，实现 undos/redos 应该就比较简单了，只需要根据 CRDT 的数据结构的新增或者删除去实现 undos/redos 栈就可以有效解决问题。假如进行了一个生成结构对象的操作，那么撤回的时候可能就把它标记删除。

假如进行一个删除结构对象的操作，在执行撤回操作时可能就对应于重新执行结构对象的插入操作。

### CRDT 算法说明

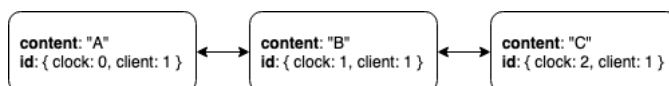
与 OT 不同，CRDT 是一种全新的解决方案，它不依赖于编辑器实现，对于任何的编辑器数据模型都可以使用一套 CRDT 数据结构去处理冲突，也是因为数据结构的性质，它也可以不依赖中心化的服务器，而且稳定性非常高，这区别于 OT，OT 可以理解为是通过算法控制保证数据一致性，CRDT 通过数据结构设计保证数据一致性，它在复杂的网络环境中的处理是更稳健的，CRDT 的代价就是要保存更多的元数据，这会带来一定内存消耗，但是这是可优化的，事实证明这个代价在协同编辑场景是完全可忽略不计的。

### Yjs 优化

其实基于 CRDT 的协同编辑方案一直是被质疑的，而且质疑的声音到现在都一直还在，Yjs 也受其影响。尽管基于 CRDT 实现的 Yjs 已经如此强大了，大家还总是拿 CRDT 的内存开销、性能开销说事，以我目前的了解：内存开销、性能问题对于 Yjs 来说早已不是问题，所以这里简单介绍一下 Yjs 的优化，这部分内容的整理基于官方对 Yjs 优化的介绍，性能问题和内存占用问题每一个点都有大量的基准测试去验证，这里只对优化方式进行一些简单的介绍。

## 一、结构表示优化

当用户从左到右键入内容“ABC”时，它将执行以下操作： $\text{insert}(0, "A") \cdot \text{insert}(1, "B") \cdot \text{insert}(2, "C")$ 。对文本内容建模的 YATA CRDT 的链表将如下所示：



插入内容“ABC”的CRDT模型（假设用户具有唯一的客户端标识符“1”）所有的 CRDT 都会为每个字符分配某种唯一的 ID 和附加的元数据，这对于大型文档来说非常消耗内存。我们不能删除元数据，因为它是解决冲突的必要条件。

Yjs 也唯一地标识每个字符和分配元数据，有效地表示了这些信息。较大的文档插入表示为单个 Item 对象，使用字符偏移量唯一地单独标识每个字符。

然后这块是有优化空间，下面的 Item 也可以将字符“A”唯一标识为 {client:1,clock:0}，字符“B”为 {client:1,clock:1}，依此类推.....

```

Item {
  id: { client: 1, clock: 0 },
  content: 'ABC',
  length: 3,
  ...
}
  
```

如果用户将大量内容复制/粘贴到文档中，则插入的内容由单个 Item 表示。此外，从左到右写入的单字符插入可以合并为单个 Item。重要的是，我们能够在不丢失任何元数据的情况下拆分和合并项。

这就是 Yjs 对于数据表示的优化，通过这种方式可以有效减少 Yjs 数据结构中结构对象的数量，从而有效减少内存的占用。

然而，这种方法最重要的缺点是处理单个字符变得更加复杂（也没关系，因为这是 Yjs 框架做的事情）。

当另一个用户希望在“B”和“C”之间插入一个字符时，需要将操作的“BC”部分拆分为两个单独的操作。我们不能重新组合这些操作，因为在 CRDT 中我们永远不能删除字符或从文档树中删除它们。

## 二、删除优化

我们可以指示需要删除字符的唯一方法是将其标记为已删除。虽然如此，这块还是有优化空间，以 Slate 的段落结构为例，当你将段落标记为删除时，你也可以将段落下的所有文本结构标记为删除。

比如，一个段落包含文本“ABC”，当标记段落删除时：

(Paragraph) D

相当于将以下所有文本节点（字符）也标记为删除：

AD    BD    CD

这是我们可以完全从内存中删除所有字符节点对应的结构，因为字符节点是被删除段落的子节点。

基于这种方式也可以有效减少 Yjs 的内存占用。

## 三、操作定义



这块其实是从 V8 的角度去优化 Yjs 结构对象的创建，整体思路就是让 Yjs 创建对象的过程能够被浏览器优化，无论是内存占用还是对象创建速度。

#### 四、查询优化

大家应该都知道使用双向链表最大的弊端就是查询性能，因为每一个操作你都需要遍历整个链表去查询某一个结构对象，当 Yjs 结构对象数据非常巨大时，执行的每一个操作有可能会因此损耗一定的时间，Yjs 对此也是有优化措施的，目前我从源代码中看到，Yjs 会对用户经常操作的结构对象进行缓存（其实就是缓存位置），查找过程中优先重缓存中去匹配，通过如果缓存命中则可以有效提高数据的查询速度。

#### 五、编码优化

Yjs 会对网络中传输以及存储在数据库中结构对象进行统一的二进制编码，当然也会提供相应的解码操作，通过二进制编码可以有效的提高数据的传输效率。

#### OT vs CRDT

	优势	劣势
OT	1. 高性能 2. 保留原始的操作意图 3. 容易理解	1. 需要中心化服务器 2. 需要 OT 控制算法 3. 不同数据模型 OT 算法需单独实现
CRDT	1. 去中心化 2. 天然支持离线编辑 3. 稳定性高	1. 损失操作意图（比如 yjs 就不支持 split_node、move_node 操作的同步） 2. 损耗内存及性能 3. 基础数据结构实现难度大

OT 和 CRDT 算法的部分就到这里，下面介绍下基于 OT 和 CRDT 算法在实际开发中的工程落地方案。

#### 开源解决方案

这里主要介绍两种方案，一种是基于 OT 的 ShareDB 方案，另外一种是基于 CRDT 的 Yjs 方案。



#### ShareDB 方案



#### Yjs 方案

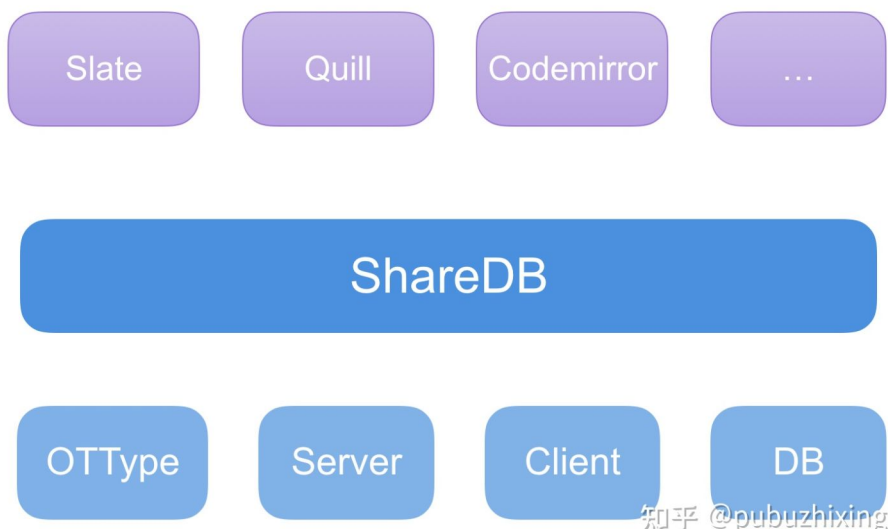
知乎 @pubuzhixing

#### ShareDB 方案

针对 OT 其实社区一直有一个对应的解决方案 - sharedb，只是比较遗憾的是 slate 和 sharedb 该怎么结合缺少明确方案，我在 Github 上搜索发现也有人研究过，只不过是针对的是 slate 比较旧的版本，也不怎么维护了，但是它的实现给了我一些思路，加上原本的理解就有了现在的方案：slate + [ottype-slate](#) + sharedb。

**ShareDB** ShareDB 是基于 OT 实现协同编辑的一套解决方案，提供协同消息转发、光标同步、数据持久化、OT 控制算法等等。

ShareDB 架构图如下



下边浅蓝色部分是 ShareDB 包含的主要模块，ShareDB 会提供基于 WebScket 的消息服务实现以及对应的前端链接消息服务的 SDK，可以同步操作和光标，ShareDB 也包含数据持久化部分的实现。最左边的 OTType 是核心的操作转换的部分，因为不同编辑器的数据模型需要实现单独 OT 的算法，所以 ShareDB 本身不包含 OT 的实现，而是提供了标准的接入接口，任何数据类型只要基于这个接口实现了对应的操作转换算法，那么它就可以通过注册的方式接入到 ShareDB 中，这个标准接口的定义可以参考 [ottypes](#) 中的实现。

上面紫色部分是 ShareDB 可以支持的编辑器，编辑器想要接入最终的任务就是基于编辑器的数据模型实现一个自己的 OTType 就可以，然后 Quill 编辑器的 Delta 数据模型本身就实现了操作转换的逻辑，所以 Quill 是最容易接入的。

### ottypes

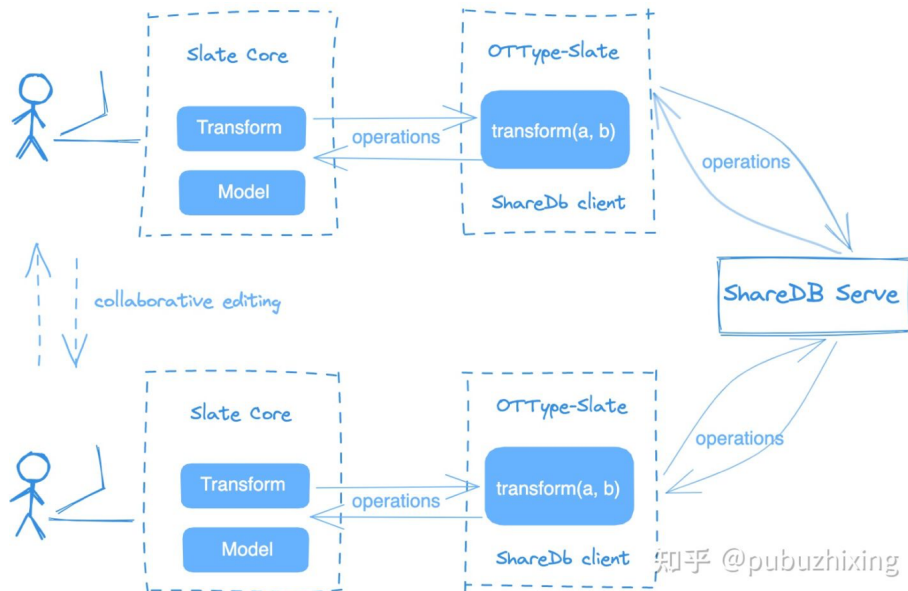
前面有提到的 otypes 其实是定了一种标准的 OT 的接口，根据这种标准实现的的类型转换都可以完美的与 ShareDB 配合使用，共同完成数据的协同编辑，前面方案中提到的 otype-slate 其实就是 otypes 的一种实现。

### otype-slate

个人感觉 slate 中定义的数据模型以及数据变换可读性非常高，它的表达方式以及提供的工具函数式非常清晰且完善，并且每种原子操作都是可逆的，我大概看了 sharedb 默认支持的基于 JSON 的操作变换实现(ot-json0)，ot-json 针对数据修改的表达，可读性还是非常差的，所以我感觉可以自己写一个针对 slate 数据模型的 OTType 实现，所以就有了 [otype-slate](#)。

otype-slate 当前只是初步实现了部分操作变换函数，然后结合 slate-angular 和 sharedb 搭建了一个协同编辑的测试 Demo，剩余的部分操作变换函数后续慢慢补充。

### ShareDB 方案流程图



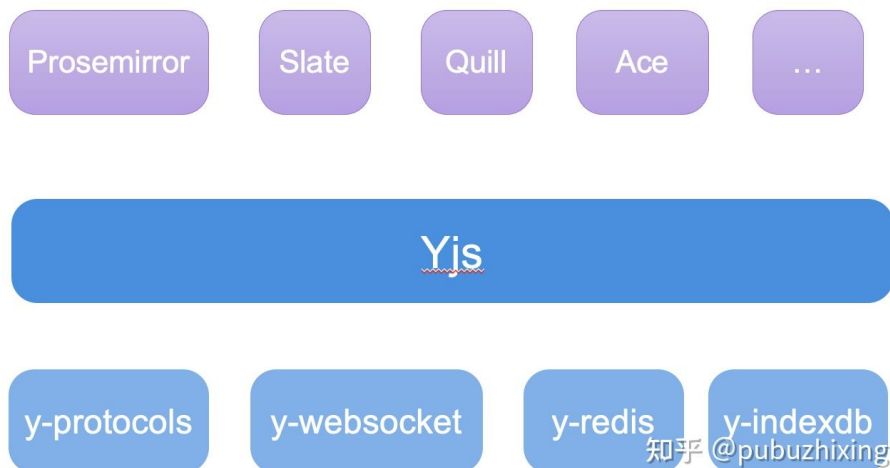
从上面开始看，假如用户在基于 Slate 编辑器进行协同编辑，可以看到用户内容修改产生的 operations 在传递给 ShareDB Serve 之前可能会经过操作转换，这取决于操作所基于的文档版本和服务器的文档版本是否一致，不一致就需要计算出两个版本差异的部分操作，拿差异的操作与新产生的操作进行操作转换，基于操作转换的结果去同步内容的修改，这个过程之后就是把最终的操作通过消息服务转发给其它客户端，其它客户端在应用这个操作，实现协同编辑。

从这个流程可以看出操作转换最终有可能是在服务端进行，也有可能是在客户端进行。因为操作转换的过程需要通过 OT 控制算法实现多客户端的操作变换的协调，这个过程必须走一个中心化的服务器，否则过程很难控制，所以基于 OT 算法这个方案是不能实现点对点通讯的。

### Yjs 方案

Yjs 是基于 CRDT 的开源解决方案，它提供了比较完善的生态，在2020年的时候社区也出现了基于 Slate 编辑器的中间绑定层。

Yjs 架构图



y-websocket - 提供协同编辑时的消息通讯，包含服务端实现和前端集成的SDK

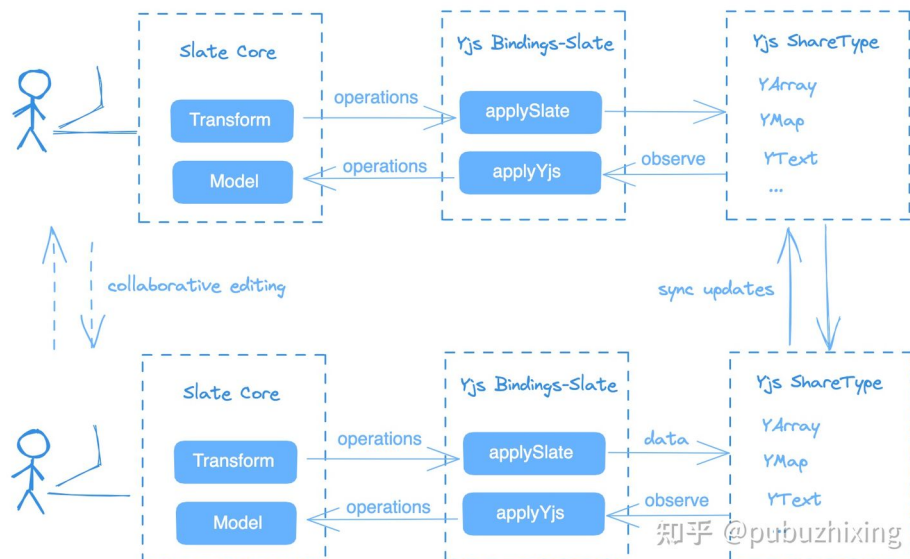
y-protocols - 定义消息通讯协议，包括消息服务初始化、内容更新、鉴权、感知系统等

y-redis - 持久化数据到 Redis

y-indexeddb - 持久化数据到 IndexedDB

在上层 Yjs 支持任何大部分主流编辑器的接入，因为 Yjs 也可以理解为一套独立的数据模型，它与每种编辑器本身的数据模型是不同的，所以每种编辑器想要接入 Yjs 都必须实现一个中间绑定层，用于编辑器数据模型与 Yjs 数据模型转换，这个转换是双向的，官方目前提供了 Prosemirror、Quill、Ace 等编辑器的中间绑定层，基于 Slate 编辑器的中间绑定层是由社区开发者提供的。

Yjs 方案流程图



从上到下描述一下用户操作的同步过程，假如上面用户在基于 Slate 编辑器进行一些数据的修改，它产生的 operations 需要先经 Yjs Bindings 把基于 Slate 的操作转换为 Yjs 的数据修改（使用 applySlate），更新本地 Yjs 的数据结构，当 Yjs 的数据结构被修改后它可以通过一种网络传输协议把数据结构的变更同步给协作者，协作者直接应用这个远程的数据同步到本地的 Yjs 数据结构上，然后 Yjs Bindings 中还有一个订阅操作，就是订阅远程的 Yjs 数据修改，然后通过 applyYjs 方法把 Yjs 数据修改的表达转化成 Slate 的 operations，最终 Slate 应用这个 operations 实现内容的同步，中间并发冲突的问题完全交给 Yjs 数据结构去处理，转化到 Slate 的操作永远跟 Yjs 的处理结果一致。

从流程图可以看出每一个客户端都维护了一个 Yjs 数据结构的副本，这个数据结构副本所表达的内容与 Slate 编辑器数据所表达的内容完全一样，只是它们承担职责不同，Slate 数据供编辑器及其插件渲染使用，然后 Yjs 数据结构用于处理冲突、保证数据一致性，数据的修改最终是通过 Yjs 的数据结构来进行同步的。

值得一提的是 Yjs 数据结构本身支持端端数据的直接同步，可以不借助中心化的服务器。

## PingCode Wiki 协同方案选择

2021年了，技术应该变一变，协同编辑方案不应该只有 OT，下面简单谈谈我们做技术选型时的考量。

今年 Q3 我们团队正式开始做协同编辑，我们的编辑器是基于 Slate 框架实现的，虽然在这之前我对协同编辑有一些调研，但都不成体系，所以在 Q3 开始的时候我们又重新进行了一次调研，核心问题还是选 OT 还是 CRDT，下面是我们当时掌握的一些情况：

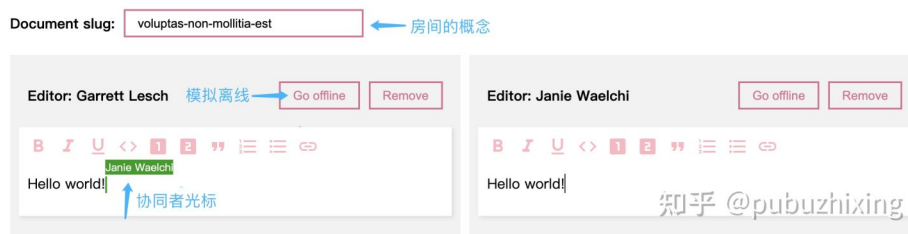
### OT 方案

- TinyMCE 编辑器基于 Slate 模型 + OT 实现协同编辑，但是他们的不开源
- 大厂产品的协同编辑方案都是基于 OT 实现的
- 对于 OT 当时只是了解思路，不知道如何落地，准确的说都不知道协同编辑应该包含哪些基础模块

## CRDT 方案

- 社区对于 CRDT 一直有一些质疑的声音
- CRDT 缺少商业产品上的应用案例（文档类）
- Yjs 生态比较完善 基于Slate编辑器有成熟的Demo
- 翻译了部分 Yjs 技术资料、对 Yjs 印象不错
- 基于我们的编辑器搭建了Yjs的协同编辑Demo，可以跑通

当时调研的 slate-yjs 提供的 Demo 截图如下



这个Demo可以说功能非常完善，而且技术栈跟我们基本是完全吻合。虽然对于 CRDT 社区有一些质疑的声音，但是事实总要验证一下，因为 Yjs 完善的 Demo 以及对它的初步印象，我们决定按照 Yjs 的方案试一试。

这基本上是我们选型的过程了，因为之后的过程就很顺利，首先是我们基于 Yjs 的生态快速在测试环境上搭建了协同编辑的初步版本，逐渐的我们在官方提供的消息服务的基础上重新实现了一个我们自己的消息服务，加上鉴权，然后基于就是逐步排查和修复协同编辑的一些细节问题，包括消息服务连接的控制、undos/redos 的问题、弹框处理等等，总之就是没有太大的问题，而且性能上基本没有损耗，大文档的加载（大概5-6万字的内容）Yjs 基本可以在毫秒级去处理完成。

现在重新来看Yjs方案的选择，我觉得我们这套方案的选择非常正确，在这个过程中没有浪费一点团队的时间，而且在Q3实现协同编辑的过程中，大家都很轻松，而且在 Yjs 上我们还可以学到很多东西，下面是我总结的 Yjs 在功能以及设计上的一些优势：

### 功能上：

- 设计了完善的感知体系，用户同步用户在线状态、光标位置等
- 支持离线编辑
- 网络不可知，可以非常稳健的处理网络抖动、网络延时等问题
- 提供 undos/redos 的管理
- 版本历史

### 设计上：

- 模块职责划分清楚，尤其是抽取独立的协议库 y-protocols，让复杂的消息同步变得非常的清晰可控
- 网络协议/数据持久化 实现松耦合，网络协议支持接入 y-websocket、y-webrtc，持久化 y-redis、社区有提供 y-mongodb
- 可以很快的与任意编辑器集成

可以这么说现在 Yjs 对于我们的意义，就之于两年前 Slate 对我们的意义，是我们这个阶段了解和学习协同编辑的重要支柱，实现协同编辑到底包含哪些东西、都有什么问题、Yjs 是怎么解决的、Yjs 有什么缺点、它是如何优化的等等，就像一个老师帮助你完成你的工作，然后让你在这个过程中有所进步。

## 谈谈技术的演进



1989 年 OT 算法正式提出，代表着协同编辑技术的开始，但是当时编辑器的架构设计远不能达到现在的水平，它的理念在那个时期一定是非常超前的，现在协同编辑数据模型的演变我觉得一定程度上也有受 OT 算法的影响。

2006 年 Google 把 OT 真正带到了商业产品中，这个过程经历大概十多年，然后就是 2011 微软紧接着基于 OT 实现了协同编辑，这中间也经历了大概 5 年的时间，我觉得这个时间跨度一定跟当时的编辑器技术背景有关系，这个时期其实协同编辑技术也只是在这些顶尖科技公司得到发展和应用。

2011 年 CRDT 算法提出代表着一种新的协同编辑方案的出现。

2012 年 Quill 编辑器开源，它的数据模型 Delta 就是基于 OT 算法设计的，个人觉得 Quill 编辑器的开源对于协同编辑以及 OT 的发展是一个重要的里程碑，在以前协同编辑可能是少数大公司在研究的技术，Quill 编辑之后协同编辑就逐渐应用更多的中小公司产品中，比如国内的石墨文档整个核心技术包括协同编辑可能就是基于 Quill 和 Delta 实现的。

2013 年 ShareDB 开源，代表着基于 OT 的一套完整解决方案的落地。

2015 年 Yjs 开源代表着基于 CRDT 的协同方案正式得到发展。2019 年 Slate 框架基于 TypeScript 完全重构，它的数据模型得到进一步优化，目前已经极其简洁优雅，我觉得这也代表着一种变化。

2020 年 slate-yjs 开源，它是 Yjs 和 Slate 的一个结合，有了这个结合其实就有了一个基于 Slate 的完整协同方案。

2021 年我觉得我们在这个时间选择 Yjs 也很合理，不同的时期技术的选择一定是不同的。

这里想延伸一点就是 OT 算法其实是在现有的编辑器数据模型的基础上实现的协同编辑，它的思想也很好理解，其实反过来想，现在协同编辑所遇到的数据一致性的问题也有一部分原因是由于数据模型中「数据修改操作」的表达所引起的，比如数据修改操作中基于索引的方式去定位要修改的数据所产生的脏路径问题，总之 OT 可以理解现有技术思路下的解决方案。然后 CRDT 其实是一种独立于现有编辑器架构的解决方案，是一种技术上的创新，它为实现协同编辑提供了一种新的思路，并且它有很多优秀的特性，比如支持点到点的数据同步，并且基于数据结构的冲突处理其实是更稳健的，虽然基于 CRDT 的数据结构在实现起来复杂度比较高，但是这个复杂度可以完全由框架层去完成，使用者其实对这块可以是无感的。

## 收尾

这篇文章其实是我们公司今年举办的「PingCode 开发者大会 2021」而准备的主题内容，然后我本身其实也想对协同编辑这块的内容做一个整理，趁这个机会就一起做了，主要是阐述了我对这块技术的一个认识，包括协同编辑是什么，协同编辑所遇到的一些问题或者说挑战，然后主流协同编辑冲突处理算法是怎么工作的，再到后面的基于冲突处理算法的开源解决方案等等，这里面提到的大部分技术其实都是开源的，内心其实是非常佩服这些开源作品的贡献者的，也在督促自己努力的去做更多的开源输出。

开源项目地址：



[github.com/quilljs/quil...](#)  
[github.com/ottypes](#)  
[github.com/pubuzhixing8...](#)  
[github.com/qqwee/slate-...](#)  
[github.com/share/shared...](#)  
[github.com/yjs/yjs](#)

参考文章

OT

[SharedPen 之 Operational Transformation](#)  
[This Is How to Build a Collaborative Text Editor Using Rails](#)

[协同编辑原理与实践 - 沙洲](#)

Yjs

[Yjs——一个基于CRDT的数据协同框架](#)

[Yjs deep dive: How Yjs makes real-time collaboration easier and more efficient](#)  
[blog.kevinjahns.de/are-...](#)

这个仓储记录了我们在做协同编辑时整理的一些资料[github.com/pubuzhixing8...](#)

编辑于 2021-10-25 20:22

[协同编辑](#)   [富文本编辑器](#)   [在线文档](#)

31 条评论

切换为时间排序

写下你的评论...



我不想加班背绩效

2021-11-24

建议看看2018年专门针对保留操作意图的协同编辑算法的论文 是一个综述 知网能搜到。另外还有一种协同编辑算法叫ast，他思想跟其他两个又不一样。

1



知乎用户

2021-10-25

为什么不用网络游戏中的协同操作，要求数据实时更新，低延迟高同步才是完美的协同解决方案把。有什么难点么？

2



pubuzhixing (作者) 回复 知乎用户

2021-10-25

可能是场景不一样，网络游戏中的协同操作是使用类似 OT 的方法吗？OT 可能是最简单高效的，但是很多复杂的情况需要自己控制。

赞



我不想加班背绩效 回复 知乎用户

2021-11-24

协同编辑主要是解决冲突问题 游戏那种场景里面不存在冲突 只是实时传输最新数据

赞

[展开其他 1 条回复](#)



monkey Tao

2021-12-06

啥时候讲讲微软的 Fluid

赞



Shawn旭

2021-12-03

请问大佬您们的产品支持表格的协同吗？

👍 赞



pubuzhixing (作者) 回复 Shawn旭

2021-12-03

我们产品是没有 Excel 类型的文档，富文本内的表格是支持协同的

👍 赞



sunson

2021-11-06

非常赞的文章！不过有个小疑问，没有中心化服务器的话，这里的实时保存是怎么实现的呢，需要前端不停向socket发全量文档数据么。用OT+中心服务器就可以在服务器里“重装”最新文档了。

👍 赞



pubuzhixing (作者) 回复 sunson

2021-11-06

通常无中心化的服务器也是需要服务器参与的，只不过服务器只是作为终端之一，当有数据更改时服务器端也可以收到增量的同步数据，与所有协同者一致。这个时候可以在服务器端做数据持久化或者版本保存。

👍 4



yichen

2021-11-03

后悔没有听到大佬的分享，看了一遍受益匪浅

👍 赞



pubuzhixing (作者) 回复 yichen

2021-11-03

哈哈，太能捧了

👍 赞



袁凡迪

2021-10-27

列表元素的 move 是怎么解决的？原始内容123用户 a 把 3 移动到 2 之前。用户 b 把 a 移动到 1 之前。

👍 赞



pubuzhixing (作者) 回复 袁凡迪

2021-10-27

描述是不是有错误，最后说的是用户 b 把 2 移动到 1 之前吧？😂

目前 yjs 数据结构其实是没提供移动操作，slate 的移动操作对应到 yjs 中是先删除再插入逻辑，按照这个推论，两个用户操作后的结果 a: 132, b: 213，最终数据同步之后我猜测的结果是：213，删除操作可以忽略，因为删除操作是对应的位置标记删除，所以就是看插入操作的处理，a 在 1 和 2 之间插入 3，b 在 1 之前插入 2

👍 赞



袁凡迪 回复 pubuzhixing (作者)

2021-10-27

说错了，应该是 3 移到 1 之前。

这样交换之后结果应该会变成 3132

👍 赞

[查看全部 8 条回复](#)



寒翅

2021-10-27

虽然看不懂，但是感觉好厉害的样子😂先赞为敬

👍 赞



Power

2021-10-27

大佬牛逼

👍 赞



- **Anomalous**

2021-10-26

大佬牛逼

 赞
- **英俊的胖子**

2021-10-25

大佬牛逼

 赞
- **覃茜**

2021-10-25

大佬牛逼

 赞
- **路路**

2021-10-25



 赞
- **小猪折射的阳光**

2021-10-25



 赞
- **微澜**

2021-10-25

不愧是大佬 

 赞
- **狼与香辛料** 

2021-10-25

强 


 赞
- **徐海峰**

2021-10-25

 大佬威武

 赞

文章被以下专栏收录



slate-angular

Angular + Slate富文本编辑器技术

推荐阅读



INNFO... 发表于机器人硬件

INNFO SCA：基于QDD技术的一体化低成本机器人运动控



ROKAE珞石

协作机械臂的设计解析及应用介绍 | 珞石机器人研发中心系统



阿里云云栖号

数字化时代，阿里云云效如何构建下一代研发协作工具平

简介： 本次分享主要由四部分组成： 1、企业在成长过程中遇到的研发效能困境； 2、研发管理从信息化走向数字化的路径，以及背后的逻辑； 3、云原生和 AI 两项新技术在研发平台上的落地； 4...



量子位

清华教授沈向洋到极致，用开源

