富文本编辑器的技术演进

通常大家把编辑器技术的发展分为三个阶段。

Level 0 阶段

编辑器的起始阶段，代表完全基于浏览器原生技术实现。

L0的实现依赖于浏览器提供的两个原生特性：

- contenteditable
  contenteditable 特性，可以指定某一容器变成可编辑器区域，即用户可以在容器内直接输入内容，或者删减内容。
- document.execCommand()(当前已经被废弃)
  execCommand API，可以对选中的某一段结构体，执行一个命令，譬如赋予黑体格式。

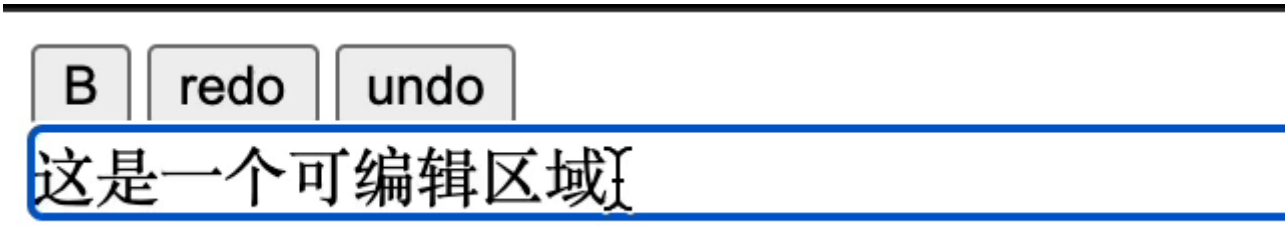
基于这两个特性我们来实现一个支持redo、undo以及字体加粗的编辑器：

- 支持编辑
  非常简单，只需要一句代码：

```
<div contenteditable="true">这是一个可编辑区域</div>
```

- 支持redo、undo以及字体加粗
  也非常简单，只需要通过execCommand执行相应命令：

```
function bold(){
  document.execCommand('bold');
}
function redo(){
  document.execCommand('redo');
}
function undo(){
  document.execCommand('undo');
}
```



看上去非常简单，L0的优点在于：

- 底层完全使用浏览器原生能力，不用考虑数据模型、排版，输入等问题。
- 直接使用 HTML 文件作为存储模型，不需要任何 UI 框架。

同样，L0的缺点也很突出：

- 兼容性问题
  完全依赖execCommand去操作dom，各家浏览器可能有不同的实现，导致相同的数据呈现出不同的效果。
- 能力有限
  只接受有限的 commands。

L0阶段的编辑器(CKEditor(1-4)、TinyMCE、UEditor等)针对上述缺点采取的主要思路是：对不同浏览器作兼容性处理，并规避一些bug；对有限的命令集进行补充。但仍存在很多问题：

- 兼容不同浏览器成本巨大且不够稳定
- 对命令集只是进行补充，自行封装实现效果并不提供通用的可扩展的接口
- 不可预测的交互容易出现数据混乱(拖拽、复制粘贴、删除)
- 特定结构的富文本内容(图片+Caption)实现复杂
- 协同编辑困难(CKEditor 5重头开始做的根本原因)

Level 1 阶段

这个阶段，弃用浏览器自带的 execCommand，完全自己实现富文本样式操作。
根据数据模型的不同又可以分为：

- 传统模式
  dom树就是数据模型，各种操作都是调用dom api来实现。典型的产品有：CKEditor4、TinyMCE、UEditor...

- MVC模式
  这里的MVC模式强调的是自建数据模型，通过更新数据来触发渲染并提供通用扩展接口。典型的产品有：CKEditor 5、Slate.js、Quill.js、Draft.js、ProseMirror...

如果我们要自己实现一个L1的编辑该怎么办呢？首先，我们要对选中的节点进行操作(比如加粗)，就需要知道哪些节点当前被选中了，这里需要了解两个非常重要的概念Selection和Range，知道这两个概念后，我们来实现一个具有加粗功能的编辑器：

```
<!DOCTYPE html>
<html lang="en">
  <script>
    function bold(){
      const selection = getSelection();
      const range = selection.getRangeAt(0);
      const boldNode = document.createElement('b');
      //为简单起见直接用range的api在选中的范围外用tag b包裹
      range.surroundContents(boldNode);
    }
    </script>
  <body>
    <button onclick="bold()">B</button>
    <div contenteditable="true">这是一个可编辑区域</div>
  </body>
</html>
```

通过调用bold函数就能够加粗选中的区域，但实现加粗的方式就是直接去操纵dom。如果要以MVC的方式去实现，首先需要定义数据模型：

```
interface Block {
  id: string;
  type: string;
  data: any
}
interface Model {
  version: string;
  time: number;
  blocks: Block[]
}
```

给定一个简单的模型：

```
{
  "version": "0.0.1",
  "time": 1634199202533,
  "blocks": [
    {
      "id": "1",
      "type": "paragraph",
      "data": {
        "text": "这是一个可编辑区域"
      }
    }
  ]
}
```


通过MVC的方式实现加粗的功能：

```
<!DOCTYPE html>
<html lang="en">
  <script>
    var model = {
      version: "0.0.1",
      time: 1634199202533,
      blocks: [
        {
          id: "1",
          type: "paragraph",
          data: {
            text: "这是一个可编辑区域",
          },
        },
      ],
    };

    function renderBlock(block) {
      switch (block.type) {
        case "paragraph":
          return `<p id=${block.id}>${block.data.text}</p>`;
        default:
          break;
      }
    }

    function render(data) {
      let html = "";
      data.blocks.forEach((block) => {
        html += renderBlock(block);
      });
      const root = document.getElementById("editor");
      root.innerHTML = html;
    }

    function getSelectedNodeInfo() {
      const selection = getSelection();
      const range = selection.getRangeAt(0);

      const id = range.commonAncestorContainer.parentNode.getAttribute("id");
      const fragment = range.cloneContents();
      const serializer = new XMLSerializer();
      const docFragmentStr = serializer.serializeToString(fragment);

      return { id, text: docFragmentStr };
    }

    function updateModel(selectedNodeInfo) {
      const foundIndex = model.blocks.findIndex(
        ({ id }) => selectedNodeInfo.id === id
      );
```

```
      if (foundIndex >= 0) {
        const newBlocks = [...model.blocks];
        const originBlock = newBlocks[foundIndex];
        const textNeedBold = selectedNodeInfo.text;
        const newText = originBlock.data.text.replace(
          textNeedBold,
          `<b>${textNeedBold}</b>`
        );
        newBlocks[foundIndex] = {
          ...originBlock,
          data: {
            text: newText,
          },
        };
        const newModel = {
          ...model,
          time: Date.now(),
          blocks: newBlocks,
        };
        return newModel;
      }
      return model;
    }

    function bold() {
      const selectedNodeInfo = getSelectedNodeInfo();
      const newModel = updateModel(selectedNodeInfo);
      render(newModel);
    }

    window.onload = () => {
      render(model);
    };
  </script>
  <body>
    <button onclick="bold()">B</button>
    <div id="editor" contenteditable="true"></div>
  </body>
</html>
```

这是一个可编辑区域

以上，粗陋地实现了一个L1的编辑器，其主要的思路就是定义数据模型，通过数据驱动来修改文档。现代L1的编辑器总体也是这个思路，只是他们对数据模型的设计、虚拟dom、协同编辑、扩展性等有更深更广的考虑。

我们来看一下主流的L1开源编辑器的实现方式：

- Quill.js(2012)
  quilljs 是一款非常优秀的编辑器，github数量已达31.1k，足见其受欢迎程度。
  quilljs底层虽然依赖浏览器dom contentEditable特性，但对dom tree以及数据模型的修改进行了抽象，实现编辑功能不再是直接修改dom而是通过操作模型api来实现。这里需要知道几个重要的概念：Delta、Parchment & Blots。

  Delta

  > Deltas are a simple, yet expressive format that can be used to describe Quill's contents and changes. The format is a strict subset of JSON, is human readable, and easily parsible by machines. Deltas can describe any Quill document, includes all text and formatting information, without the ambiguity and complexity of HTML.

  Delta为JSON的一个子集，用来描述编辑器的**内容**及**变化**，只包含一个 ops 属性，它的值是一个对象数组，每个数组项代表对编辑器的一个操作。
  下面是一段富文本内容描述：
  hello **Quill**
  用Delta进行描述：

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

  Delta只有3种动作和1种属性，却足以描述任何富文本内容和变化。
  3种动作：
  - insert：插入
  - retain：保留
  - delete：删除

  1种属性：
  - attributes：格式属性

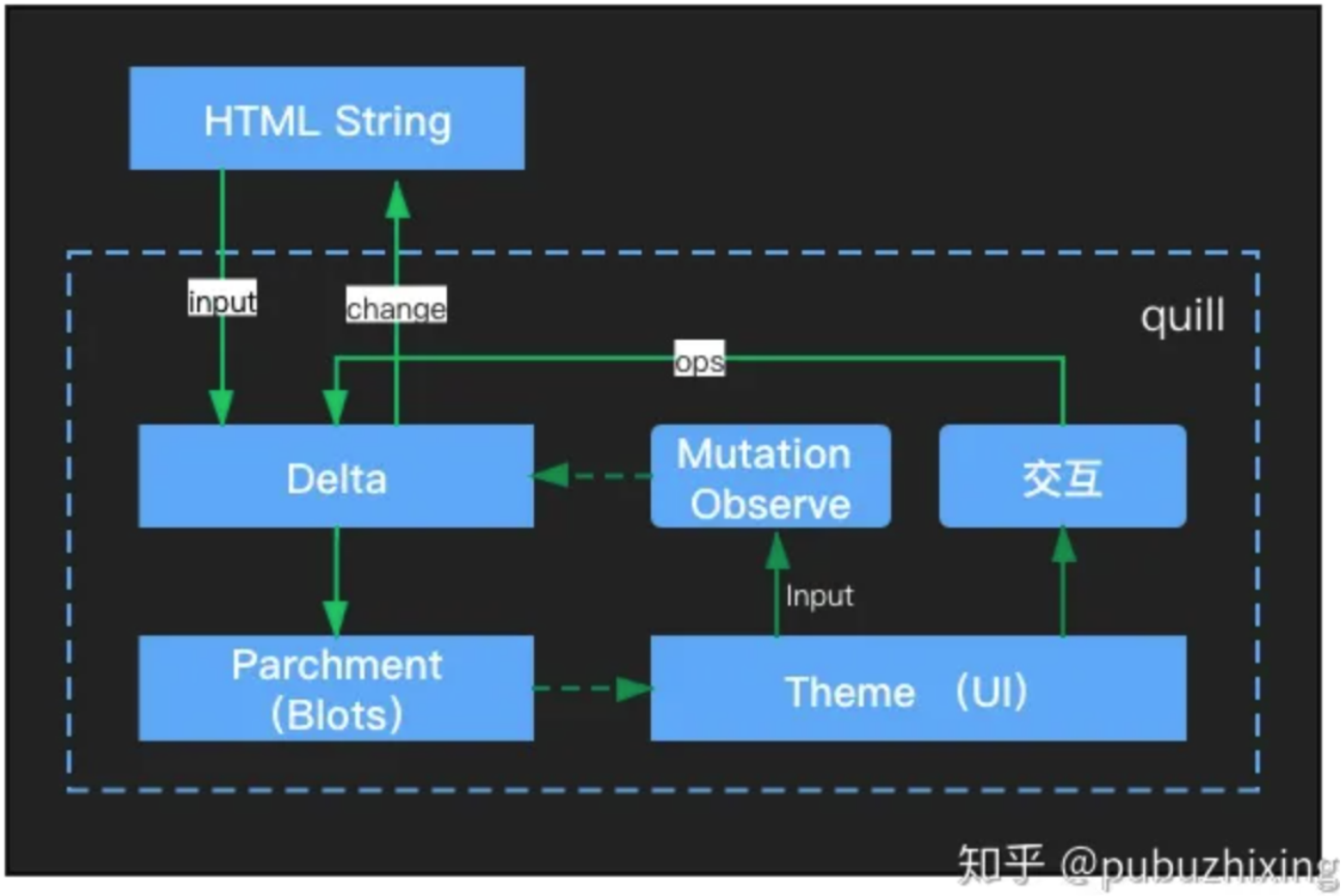
Delta的描述方式其实OT(Operation Transformation)模型的一种实现，而协同编辑通常都是基于OT操作之上实现的，因此Quill天然就支持协同编辑。

Parchment & Blots

> Parchment is Quill's document model. It is a parallel tree structure to the DOM tree, and provides functionality useful for content editors, like Quill. A Parchment tree is made up of Blots, which mirror a DOM node counterpart. Blots can provide structure, formatting, and/or content. Attributors can also provide lightweight formatting information.

Parchment是Quill.js中对于DOM的抽象，Parchment其实是与DOM树对应的结构，Parchment由Blots组成，Blot即与DOM的Node对应，Quill.js文档怎么渲染完全由Blot决定，那么这层模型其实就是Delta数据与最终UI之间的一个中间层。这与我最初的想象是不一样的，我最初认为dom树直接由数据模型来映射，并不存在像Parchment这样的中间层。这个中间层带来的好处是能够更容易的控制dom的修改。

quiijs的架构图(引自)如下，



Delta模型的改变有两种方式：

- 文本输入
  通过MutationObserver来监听dom的变化，然后同步变化到Delta。
- 复杂的样式或者格式操作等非浏览器默认行为
  交互行为改变Delta，Delta驱动Parchment的改变进而改变dom。

特点

- 依赖浏览器原生的编辑能力（Level 1）
- 数据更新主体是Delta，DOM的更新由单独的Parchment & Blots描述
- 输出数据可以是HTML的字符串也可以由Delta描述的一系列操作（也就是JSON）
- Quill.js主体、Parchment、Delta都是独立的仓储，架构良好

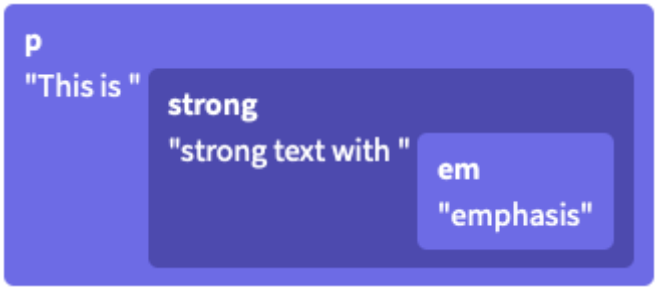
quilljs引入了数据模型、抽象出了数据变化的操作，后面出来的编辑器多少都有借鉴Quill.js的实现思路。

- ProseMirror(2015)
  ProseMirror也是依赖contentEditable，不过ProseMirror将主流的前端架构理念应用到了编辑器的开发中，比如彻底使用纯JSON数据描述富文本内容，引入不可变数据以及Virtual DOM的概念，还有插件机制、分层、Schemas(范式)等等，所以ProseMirror是一款理念先进且体系相对比较完善的一款编辑器(或者说框架)。
  它的核心库有:
  - prosemirror-model
    定义编辑器的文档模型，用来描述编辑器内容的数据结构。
  - prosemirror-state
    提供描述编辑器整个状态的数据结构，包括selection(选择)，以及从一个状态到下一个状态的transaction(事务)。
  - prosemirror-view
    实现一个在浏览器中将给定编辑器状态显示为可编辑元素，并且处理用户交互的用户界面组件。
  - prosemirror-transform
    包括以记录和重放的方式修改文档的功能，这是state模块中transaction(事务)的基础，并且它使得撤销和协作编辑成为可能。

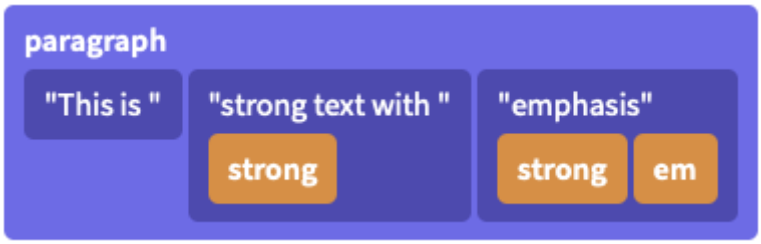
ProseMirror模型为一个扁平化的结构：

在 HTML 中, 一个 paragraph 及其中包含的标记, 表现形式就像一个树, 比如有以下 HTML 结构:

```
<p>This is <strong>strong text with <em>emphasis</em></strong></p
```



然而在 Prosemirror 中, 内联元素被表示成一个扁平的模型, 他们的节点标记被作为 metadata 信息附加到相应 node 上:



对应的JSON结构如下：

```
{
  "type": "doc",
  "content": [
    { "type": "paragraph" },
    { "type": "text", "text": "This is" },
    {
      "type": "text",
      "text": "strong text with",
      "marks": [{ "type": "strong" }]
    },
    { "type": "text", "text": " " },
    {
      "type": "text",
      "text": "emphasis",
      "marks": [{ "type": "strong" }, { "type": "em" }]
    },
    { "type": "paragraph" }
  ]
}
```

这种数据结构能够使用字符的偏移量而不是一个树节点的路径来表示其所处段落中的位置，并且使一些诸如 splitting 内容或者改变内容 style 的操作变得很容易，而不是以一种笨拙的树的操作来修改内容。

schema

上诉定义了扁平的数据结构来表示文档内容，在数据结构与HTML的Dom结构之间，需要一次解析与转化，这两者间相互转化的桥梁，就是schema。

例子：

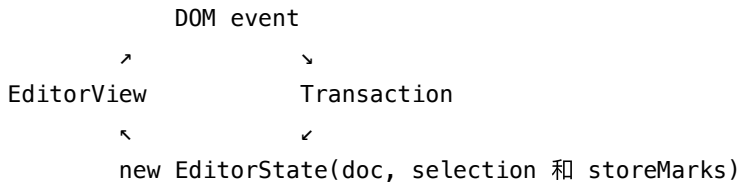
```
const dinoNodeSpec = {
  attrs: {type: {default: "brontosaurus"}},
  inline: true,
  group: "inline",
  draggable: true,
  toDOM: node => ["img", {"dino-type": node.attrs.type,
                          src: "/img/dino/" + node.attrs.type + ".png",
                          title: node.attrs.type,
                          class: "dinosaur"}],
  parseDOM: [{
    tag: "img[dino-type]",
    getAttrs: dom => {
      let type = dom.getAttribute("dino-type")
      return dinos.indexOf(type) > -1 ? {type} : false
    }
  }]
}
```

有了数据以及数据类型对应的范式的定义，从JSON数据到DOM的更改是可以完全由ProseMirror接管，ProseMirror是在中间做了一层虚拟DOM来完成数据到DOM的驱动更新。

Transform

ProseMirror有一个单独的模块来定义和实现文档的修改，这样内容的修改被统一起来，并且最终都会转化为底层的原子操作（为协同编辑提供可能），而且可以在任何插件中做拦截处理，比如实现：记录数据更改操作来实现撤销和重做等。

下图是prosemirror简单的循环数据流：



和redux很像，Transaction可以看成是action，action触发state的改变进而更新UI。

特点

- 依赖浏览器原生的编辑能力（Level 1）
- 嵌套的文档模型（区别于Delta的OT模型，它的文档模型是通常意义上的JS对象模型，对应的模型数据可以作为结果直接存储）
- Schemas（范式）约定模型嵌套以及渲染规则
- 统一数据更新流，采用单向数据流、不可变数据及虚拟DOM避免直接操作DOM（这一点确实融合了主流的函数式编程的思想）
- 输出的数据是纯JSON

- Draft.js(2015)
- Slate(2016)

- Slate Core(2018)
- Slate Migration(2019)

尽管这些主流的L1编辑器都取得了长足的进步，但终究是脱离不了浏览器的contenteditable特性，基于model-base的编辑器为了能够达到model-drive-render 的效果就需要监听用户的输入，然后修改模型驱动UI改变。但是在一些场景下输入是不受控的或者无法识别，一旦无法识别就会导致事件无法正确截取，最终流向了原生的行为，dom上的内容发生了变化，但是model的数据没有变化，这种不一致就会直接摧毁编辑器的model-base机制。当然，L1级别的编辑器可以满足绝大多数场景的需求了。

Level 2 第三阶段，完全不依赖浏览器的编辑能力，独立的实现光标和排版

既然 contentEditable 作为编辑区域他的行为不可控，基于L2的编辑器就直接抛弃浏览器原生的contenteditable以及execCommand特性，自己实现光标和排版，从而可以不受浏览器差异的影响并能做到对编辑器的完全控制。

为了实现可控的编辑区域，L2阶段的编辑器发展有了大致两个分支

- Google Docs，腾讯文档
  完整实现一套排版编辑系统，从元素排版到光标选区，输入事件等等，没有使用任何 HTML Form 元素或者是 contentEditable 特性，实现成本和难度都是十分巨大的，尤其是腾讯文档，甚至抛弃了 DOM，直接使用 Canvas 作为排版文档的技术，主要是为了解决高级排版的问题。
- 有道云笔记这类
  依旧使用 DOM 的排版能力，但是自绘了选区和光标，通过 Hack 的方式实现原生输入事件，成本相对于 Google Docs 是一个折中方案，效果足够好，表现也十分稳定，实现难度和成本也属于可接受的范围。

当然抛弃浏览器原生特性也意味着难度地陡然提升，没有巨人的肩膀只能从头做起，这也许是目前的开源编辑器都是采用L1方案的原因吧。

总结

- 编辑器的发展历史，就是对浏览器原生特性的抛弃史。
- 尽管目前开源编辑器基本都处于L1阶段，但是其优秀的数据模型设计、插件化架构、协同编辑算法等都值得深入学习。

参考文档

- 钉钉文档编辑器的前世今生
- 有道云笔记跨平台富文本编辑器的技术演进
- 我做编辑器这些年：钉钉文档编辑器的前世今生

- 2021年富文本编辑器架构之道
- 开源富文本编辑器技术的演进（2020 1024）
- 富文本编辑器 L1 能力调研记录
- 深入浅出contenteditable富文本编辑器
- 开源富文本编辑器技术的演进
- 富文本编辑器初探
- 富文本原理了解一下?
- 富文本编辑器初探
- 富文本编辑器之游戏角色升级ing
- 从流行的编辑器架构聊聊富文本编辑器的困境
- 富文本编辑器的技术演进
- 富文本编辑器的技术演进之路
- 现代编辑器技术原理
- 开源富文本编辑器技术的演进
- 从零写一个富文本编辑器（二）——文档模型
- Why ContentEditable is Terrible
- 揭开在线协作的神秘面纱 – OT 算法
- 现代 Web 富文本编辑器 Quill.js
- 深入浅出Quill-现代富文本编辑器Quill的模块化机制
- quill-better-table：赋予quill富文本编辑器强大的表格编辑功能！
- quilljs
- Getting to know QuillJS - Part 1
- The State of Quill and 2.0
- mutation-observer
- google-docs
- prosemirror
- prosemirror中文
- ProseMirror - 模块化的富文本编辑框架