链滴

Vditor 生命周期（运行时 runtime）设计构想

作者：HerbertHe

原文链接：https://ld246.com/article/1644516848043

来源网站：链滴

写在前面的

Vditor目前来说是没有生命周期（或称运行时这一概念，下称为runtime）。好处是可以以最少的API现编辑器的渲染，但也导致了代码整体解耦仍然不够的问题。虽然可以通过配置项实现自定义渲染器功能，但仍然没有统一的插件开发工程和独立插件发布。因此，我开始尝试对Vditor进行runtime设。

抽象化插件开发

在设计 Vdok 之时，为了实现对标VuePress、docsify、VitePress等文档工具的语法效果，Vdok尝对Vditor进行第三方的语法拓展。（Vdok目前是搁置了，因为没有太多持续的时间进行迭代，自行改是可以跑起来的）在Vdok的开发之时，逐渐意识到样式隔离和插件标准化的意义。所以对于Vdito的runtime进行设计，首先就聚焦于使Vditor可以使用标准化的插件。

插件标准化的内容仍在探索之中，会随着版本的更新逐渐添加更多的开放特性。目前可用的类型设计下：

```
/**
 * Vditor 注册插件类型
 * TODO: ILuteRenderCallback 类型重写, 现有类型不利于插件开发
 */
export interface IVditorPlugin {
    id: string
    renderers?: Map<keyof ILuteRender, ILuteRenderCallback>
    styles?: Map<string, string>
}
```

插件的命名规范为： /vditor-plugin-([a-zA-Z0-9_]+)/

插件示例如下：

```
const plugin_example: IVditorPlugin = {
  id: "vditor-plugin-test",
  renderers: new Map([
    [
      "renderDocument",
      function (node, entering) {
        return ["", Lute.WalkContinue]
      },
    ],
  ]),
}
```

需要解决的问题

Vditor目前的类型定义对于插件开发仍然是不够用的。使用TypeScript进行部分自定义渲染器开发，法兼容现有的Vditor类型定义。这个问题在Vdok开发的时候已经体会到了，可以参考 vdok/renderers ts at main · HerbertHe/vdok (github.com) 的代码体会。

似乎Vditor现有的类型定义不能完全cover掉Lute支持的所有API情况，有些API对于部分的渲染器也完全不可用的。所以重写此部分的类型定义是很有必要的。之前给Lute PR，也在链滴里面回答了这

分的提问。其实，Lute支持的自定义渲染器有很多、也很复杂，这部分的拓展需要阅读Lute的源码。

这部分的类型定义需要慢慢随着版本的更新，慢慢进行重写迭代。

链式调用使用插件

目前Vditor对象实例化的过程是高耦合的，一环环相扣，很难将插件进行注入。在参考主流前端框架后，我还是觉得对象构造函数实例化和DOM挂载渲染的过程应该分离。

比如 React：

```
import React from "react";
import ReactDOM from "react-dom";

function Hello(props) {
  return <h1>Hello World!</h1>;
}

ReactDOM.render(<Hello />, document.getElementById("root"));
```

Vue：

```
const Counter = {
  data() {
    return {
      counter: 0
    }
  }
}

Vue.createApp(Counter).mount('#counter')
```

Vditor(options).render(HTMLDivElement)

因此，对于Vditor的方法可以进一步设计实现runtime。

类似的语法如下：

```
const vditor = new Vditor(options).render(HTMLDivElement)
```

Vditor(options).use(plugins): IVditor

此API方法是先前提出的。在综合现有Vditor的代码之后，还是决定先修改Vditor的现有入口代码，实现实例化和挂载渲染分离，提出了上面的 Vditor(options).render(HTMLDivElement) 方法进行手挂载渲染。

通过返回 Vditor 实例，可以链式调用注册标准化插件，并且在实例不同的 runtime 阶段实现插件对内容的注册。

重构 Vditor 的样式表

现有的 Vditor 样式表基于 Sass 进行开发，但是 node-sass 依赖的环境非常的复杂，在 Windows

台太容易报错了。因此，使用 Less 进行重构 Vditor 样式表有进一步深远的意义。

除了上述依赖环境的问题之外，面向现代化的样式表在开发中越来越流行了。诸如 Tailwind CSS、Windi CSS、UnoCSS 等，在提高浏览器兼容性、减小样式表体积等前端优化上有很大的实用意义。迁移 Less，可以快速引入上述第三方工具，减小开发复杂度、提升性能。

将在runtime设计迭代稳定之后，着手修改现有的构建工具配置实现样式表的重构。为支持标准化插进行样式自定义，将在长远计划中提供 vditor-plugin-template 插件模板项目，实现无需关注进行层构建工具配置。

Vditor 重构之后的样式表也同时准备支持统一化原有的布局风格，开放渲染特定位置的 API 给插件发。

Vditor Plugin

目前 Vditor Plugin 设计的数据结构尚未稳定，将随着设计深入进行迭代。

在上面 抽象化插件开发 部分直观展示了目前的设计进度，下面是字段解释和更长期的设计计划。

字段解释

```
/**
 * Vditor 注册插件类型
 * TODO: ILuteRenderCallback 类型重写, 现有类型不利于插件开发
 */
export interface IVditorPlugin {
    id: string
    renderers?: Map<keyof ILuteRender, ILuteRenderCallback>
    styles?: Map<string, string>
}
```

id：即插件的标识符。规范已经在上述的内容中提及。

renderers：Vditor 自定义渲染器。采用了 Map 的数据结构，保证插件内自定义渲染器的唯一性。于多插件的自定义渲染器冲突的问题，Vditor Plugin将根据注册顺序进行倒序优先级排序。

styles：插件自定义样式表。采用约定的方式，支持插件开发者自行开发自定义样式表。

长期计划

Vditor Plugin 将上述特性实现之后，计划在设计特定生命周期阶段进行代码注入。

- setup() 方法

在 render 阶段之前进行触发。

- after() 方法

在 render 完成之后进行触发。

粗略的生命周期设计图

```
graph TD
  Vditor实例化 --> 插件Setup
  插件Setup --> 实例Render
  实例Render --> 插件After
  插件After --> 实例挂载渲染结束
```

Vditor 设计长期规划

因为runtime的实际需要，无法避免带来BREAK CHANGES，但同时带来了更多拓展的可能性。自己于志愿服务工作个人时间比较少，迭代开发 PR 可能会需要很长的时间。

在长期计划中，随着runtime的加持，也同时打算重构 Vditor 的文档。

对于 Lute 体积过大的问题，emmm 我也有尝试在修 Golang 编译到 wasm 的报错，这算是实验性尝试，比较麻烦。。。

关于 Vditor 的探索学习

从第一次接触 Vditor 到如今似乎已有两年了，从一名用户随着自己开发水平的提升开始慢慢贡献。经历了基于 Sym 的 Nucode 中北大学大数据协会论坛，到黑客派改名链滴，不变的还是开源精神。

Vditor 整体的代码难度还是不高的，有能力的同学可以阅读阅读，然后写点文章分析分析啥的，就跟们喜欢学习 React 的代码结构一样；）

现在自己的想法太多，写代码的速度跟不上idea的产生速度，实在是一件痛苦的事情
ensive