Micro-app实战

- `micro-app`采用WebComponent的思想实现，实现微前端的组件化渲染。
- `micro-app`不需要像`single-spa`和`qiankun`一样要求子应用修改渲染逻辑并暴露出方法
- `micro-app`不需要修改webpack配置

一.主应用搭建

> 主应用我们采用react作为基座

```
npx create-react-app substrate
npm install react-router-dom @micro-zoe/micro-app
```

1.加载Micro-app

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import microApp from '@micro-zoe/micro-app';
```

```
const root =
ReactDOM.createRoot(document.getElementById('roo
t'));
root.render(<App/>);
microApp.start({
  lifeCycles:{
    created:()=>console.log('created'),
    beforemount:()=>console.log('beforemount'),
    mounted:()=>console.log('mounted'),
    unmount:()=>console.log('unmount'),
    error:()=>console.log('error')
  }
});
```

2.路由注册

App.js

```
import { BrowserRouter, Link, Route, Routes }
from 'react-router-dom'
import Page1 from './page-1.js'; // 放入React应用
import Page2 from './page-2.js'; // 放入Vue应用
function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <Link to="/react">React项目</Link>
```

```
        <Link to="/vue">Vue项目</Link>
        <Routes>
          <Route path="/react/*" element={<Page1
/>}></Route>
          <Route path="/vue/*" element={<Page2
/>}></Route>
        </Routes>
      </BrowserRouter>
    </div>
  );
}
export default App;
```

3.子应用加载

page-1.js

```
export default function Page1() {
    return (
        <div>
            <div>React应用</div>
            <micro-app name="app1"
url="http://localhost:10000" baseroute="/react">
</micro-app>
        </div>
    )
}
```

page-2.js

```
export default function Page2() {
    return (
        <div>
            <div>Vue应用</div>
            <micro-app name="app2"
url="http://localhost:20000" baseroute="/vue">
</micro-app>
        </div>
    )
}
```

二.创建React微应用

```
npx create-react-app m-react
npm install react-router-dom
npm install @rescripts/cli -D
```

1.`.env`环境变量配置

```
PORT=10000
WDS_SOCKET_PORT=10000
```

2.支持子应用跨域

.rescriptsrc.js

```
module.exports = {
    devServer: (config) => {
        config.headers = {
            'Access-Control-Allow-Origin': '*',
        };
        return config;
    },
};
```

package.json

修改启动方式

```
"scripts": {
    "start": "rescripts start",
    "build": "rescripts build",
    "test": "rescripts test",
    "eject": "rescripts eject"
}
```

3.配置publicPath

```
if (window.__MICRO_APP_ENVIRONMENT__) {
  __webpack_public_path__ =
window.__MICRO_APP_PUBLIC_PATH__
}
```

4.路由配置

```
import { BrowserRouter, Route, Routes, Link }
from 'react-router-dom'
function App() {
  return (
    <div className="App">
      <BrowserRouter basename=
{window.__MICRO_APP_BASE_ROUTE__ || '/'}>
        <Link to="/home">主页</Link>
        <Link to="/about">关于页面</Link>
        <Routes >
          <Route path="/home" element=
{<div>react home</div>}></Route>
          <Route path="/about" element=
{<div>react about</div>}></Route>
        </Routes>
      </BrowserRouter>
    </div>
```

```
  );
}
export default App;
```

三.创建Vue应用

```
vue create m-vue
```

1.配置publicPath

```
if (window.__MICRO_APP_ENVIRONMENT__) {
  __webpack_public_path__ =
window.__MICRO_APP_PUBLIC_PATH__
}
```

2.支持子应用跨域

```
const { defineConfig } = require('@vue/cli-service')
module.exports = defineConfig({
  transpileDependencies: true,
  devServer: {
    port: 20000,
    headers: {
      'Access-Control-Allow-Origin': '*',
    },
  },
})
```

四.WebComponent

1.WebComponent组成

- **Custom elements**：一组JavaScript API，允许您定义custom elements及其行为，然后可以在您的用户界面中按照需要使用它们
- **Shadow DOM**：一组JavaScript API，用于将封装的“影子”DOM树附加到元素（与主文档DOM分开呈现）并控制其关联的功能。通过这种方式，您可以保持元素的功能私有，这样它们就可以被脚本化和样式化，而不用担心与文档的其他部分发生冲突。
- **HTML templates**： `<template>` 和 `<slot>` 元素使您可

以编写不在呈现页面中显示的标记模板。然后它们可以作为自定义元素结构的基础被多次重用。

2.WebComponent实战

```
<my-button type="primary">按钮</my-button>
    <template id="btn">
        <button class="my-btn">
            <slot>按钮</slot>
        </button>
    </template>
    <script>
      class MyButton extends HTMLElement {
        constructor() {
          super();
        }
        // 2.监控属性变化，触发修改回调
        static get observedAttributes() {
          return ['type']
        }
        attributeChangedCallback(name, oldVal,
newVal) { // 获取属性列表
          if (this.shadow) {
            const btn =
this.shadow.querySelector('.my-btn');
            btn.style.backgroundColor =
this.types[newVal].backgroundColor;
```

```
            btn.style.color =
this.types[newVal].color;
          }
        }
        // 3.挂载后触发事件
        connectedCallback() {
          this.shadow = this.attachShadow({
mode: 'open' });
          let btn =
document.getElementById('btn');
          // 拷贝模板
          let cloneTemplate =
btn.content.cloneNode(true);
          let style =
document.createElement('style');
          this.types = {
            'primary': {
              backgroundColor: 'blue',
              color: '#fff'
            },
            'default': {
              backgroundColor: '#a1a1a1',
              color: '#fff'
            }
          }
          const btnType =
this.getAttribute('type') || 'default';
```

```
            style.innerHTML = `
                  .my-btn {
                      outline:none;
                      border:none;
                      border-radius:4px;
                      display:inline-block;
                      cursor:pointer;
                      padding:6px 20px;
                      background:
${this.types[btnType].backgroundColor};

 color:${this.types[btnType].color};
                  }`
            this.shadow.appendChild(style);

 this.shadow.appendChild(cloneTemplate);
            this.dispatchEvent(new
Event('mounted'))
          }
      }
      const customBtn =
document.querySelector('my-button')

 customBtn.addEventListener('mounted',function()
{
        console.log('mounted')
      })
```

```
        // 4.定义自定义元素
        window.customElements.define('my-button',
MyButton);
        setTimeout(()=>{
          customBtn.setAttribute('type',
'default')
        },1000)
</script>
```

3.WebCompoent生命周期

- `connectedCallback`：当custom element首次被插入文档DOM时，被调用
- `disconnectedCallback`：当 custom element从文档DOM中删除时，被调用
- `adoptedCallback`：当 custom element被移动到新的文档时，被调用 (移动到iframe中)
- `attributeChangedCallback`:当 custom element增加、删除、修改自身属性时，被调用

五.MicroApp源码分析

1.start方法剖析

```
export class MicroApp extends
EventCenterForBaseApp implements
MicroAppBaseType {
  tagName = 'micro-app'
  options: OptionsType = {}
  router: Router = router
  preFetch = preFetch
  unmountApp = unmountApp
  unmountAllApps = unmountAllApps
  getActiveApps = getActiveApps
  getAllApps = getAllApps
  reload = reload
  renderApp = renderApp
  start (options?: OptionsType): void {
    // 不支持自定义元素
    if (!isBrowser || !window.customElements) {
      return logError('micro-app is not
supported in this environment')
    }
    // 允许用户修改tagName
    if (options?.tagName) {
      if (/^micro-app(-
\S+)?/.test(options.tagName)) {
        this.tagName = options.tagName
```

```
      } else {
        return logError(`${options.tagName} is
invalid tagName`)
      }
    }
    // 初始化全局环境，保留原始方法
    initGlobalEnv()
    // 查看window上是否已经定义过这个组件了
    if
(globalEnv.rawWindow.customElements.get(this.tag
Name)) {
      return logWarn(`element ${this.tagName} is
already defined`)
    }
    // 是否禁用scopecss及sandbox
    if (isPlainObject<OptionsType>(options)) {
      this.options = options

      options['disable-scopecss'] =
options['disable-scopecss'] ??
options.disableScopecss
      options['disable-sandbox'] =
options['disable-sandbox'] ??
options.disableSandbox

      // 是否在空闲时预先加载应用
```

```
      options.preFetchApps &&
preFetch(options.preFetchApps)

      // 是否在空闲时预加载资源
      options.globalAssets &&
getGlobalAssets(options.globalAssets)

      // 插件配置
      if (isPlainObject(options.plugins)) {
        const modules = options.plugins.modules
        if (isPlainObject(modules)) {
          for (const appName in modules) {
            const formattedAppName =
formatAppName(appName)
            if (formattedAppName && appName !==
formattedAppName) {
              modules[formattedAppName] =
modules[appName]
              delete modules[appName]
            }
          }
        }
      }
    }
    // 初始化后定义元素
    defineElement(this.tagName)
  }
```

```
}
```

2.defineElement

```
export function defineElement (tagName: string): void {
  class MicroAppElement extends HTMLElement implements MicroAppElementType {
    // 检测name和url属性
    static get observedAttributes (): string[] {
      return ['name', 'url']
    }
    // 拦截setAttribute方法，给组件生成this.data属性，并调用原有setAttribute方法设置name以及url属性，触发attributeChangedCallback
    constructor () {
      super()
      patchSetAttribute()
    }
    // 当设置data时会调用setData方法
    set data (value: Record<PropertyKey, unknown> | null) {
      if (this.appName) {
        microApp.setData(this.appName, value as Record<PropertyKey, unknown>)
      } else {
        this.cacheData = value
```

```
      }
    }
    // 当访问data时会调用getData方法
    get data (): Record<PropertyKey, unknown> |
null {
      if (this.appName) {
        return microApp.getData(this.appName,
true)
      } else if (this.cacheData) {
        return this.cacheData
      }
      return null
    }
    // ....
    // 定义WebCompoent组件

 globalEnv.rawWindow.customElements.define(tagNa
me, MicroAppElement)
}
```

1).重写setAttribute方法

```
Element.prototype.setAttribute = function
setAttribute (key: string, value: string): void
{
    if (/^micro-app(-\S+)?/i.test(this.tagName)
&& key === 'data') {
```

```
      if (isPlainObject(value)) {
        const cloneValue: Record<PropertyKey,
unknown> = {}

 Object.getOwnPropertyNames(value).forEach((prop
ertyKey: PropertyKey) => {
          if (!(isString(propertyKey) &&
propertyKey.indexOf('__') === 0)) {
            // @ts-ignore
            cloneValue[propertyKey] =
value[propertyKey]
          }
        })
        // 对data属性的处理，并且处理成getter和
setter
        this.data = cloneValue
      } else if (value !== '[object Object]') {
        logWarn('property data must be an
object', this.getAttribute('name'))
      }
    } else if (
      (
        ((key === 'src' || key === 'srcset') &&
/^(img|script)$/i.test(this.tagName)) ||
        (key === 'href' &&
/^link$/i.test(this.tagName))
      ) &&
```

```
      this.__MICRO_APP_NAME__ &&

 appInstanceMap.has(this.__MICRO_APP_NAME__)
    ) {
      // 对src属性处理
      const app =
appInstanceMap.get(this.__MICRO_APP_NAME__)
      globalEnv.rawSetAttribute.call(this, key,
CompletionPath(value, app!.url))
    } else {
      // 其它属性处理
      globalEnv.rawSetAttribute.call(this, key,
value)
    }
  }
```

2) 属性修改时调用此方法

```
public attributeChangedCallback (attr:
ObservedAttrName, _oldVal: string, newVal:
string): void {
  // 对url和name进行处理, this.appName = 'name',
this.appUrl = 'url'
  if (
    this.legalAttribute(attr, newVal) &&
    this[attr === ObservedAttrName.NAME ?
'appName' : 'appUrl'] !== newVal
```

```
  ) {
    if (attr === ObservedAttrName.URL &&
!this.appUrl) {
      newVal = formatAppURL(newVal,
this.appName)
      if (!newVal) {
        return logError(`Invalid attribute url
${newVal}`, this.appName)
      }
      this.appUrl = newVal
      this.handleInitialNameAndUrl()
    } else if (attr === ObservedAttrName.NAME &&
!this.appName) {
      const formatNewName =
formatAppName(newVal)

      if (!formatNewName) {
        return logError(`Invalid attribute name
${newVal}`, this.appName)
      }

      if (this.cacheData) {
        microApp.setData(formatNewName,
this.cacheData)
        this.cacheData = null
      }
```

```
        this.appName = formatNewName
        if (formatNewName !== newVal) {
          this.setAttribute('name', this.appName)
        }
        this.handleInitialNameAndUrl()
      } else if (!this.isWaiting) {
        this.isWaiting = true // name和url在初始化化后进行修改
        defer(this.handleAttributeUpdate)
      }
    }
}
```

3).DOM挂载后执行逻辑

```
connectedCallback (): void {
  this.hasConnected = true
  // 异步触发created生命周期
  defer(() => dispatchLifecyclesEvent(
    this,
    this.appName,
    lifeCycles.CREATED,
  ))
  // 初始化应用挂载
  this.initialMount()
}
```

```
private initialMount (): void {
      if (!this.appName || !this.appUrl) return
      // 是否有shadowDOM属性,如果有则添加shadowDOM
      if (this.getDisposeResult('shadowDOM') &&
!this.shadowRoot &&
isFunction(this.attachShadow)) {
        this.attachShadow({ mode: 'open' })
      }
      // 处理ssr路径
      if (this.getDisposeResult('ssr')) {
        this.ssrUrl =
CompletionPath(globalEnv.rawWindow.location.path
name, this.appUrl)
      } else if (this.ssrUrl) {
        this.ssrUrl = ''
      }
      // 已经加载过
      if (appInstanceMap.has(this.appName)) {
        const app =
appInstanceMap.get(this.appName)!
        const existAppUrl = app.ssrUrl ||
app.url
        const activeAppUrl = this.ssrUrl ||
this.appUrl
        // ....
      } else {
        // 处理创建应用逻辑
```

```
        this.handleCreateApp()
      }
    }
}
```

4).创建应用实例

```
private handleCreateApp (): void {
  // 创建实例，核心就是加载url路径、css隔离、js沙箱
  const instance: AppInterface = new CreateApp({
    name: this.appName,
    url: this.appUrl,
    ssrUrl: this.ssrUrl,
    container: this.shadowRoot ?? this,
    inline: this.getDisposeResult('inline'),
    scopecss: !
(this.getDisposeResult('disableScopecss') ||
this.getDisposeResult('shadowDOM')),
    useSandbox:
!this.getDisposeResult('disableSandbox'),
    baseroute: this.getBaseRouteCompatible(),
  })
  // 缓存应用实例
  appInstanceMap.set(this.appName, instance)
}
```

```
export default class CreateApp implements
AppInterface {
  private state: string = appStates.NOT_LOADED
  private keepAliveState: string | null = null
  private keepAliveContainer: HTMLElement | null
= null
  private loadSourceLevel: -1|0|1|2 = 0
  private umdHookMount: Func | null = null
  private umdHookUnmount: Func | null = null
  private libraryName: string | null = null
  umdMode = false
  isPrefetch = false
  prefetchResolve: (() => void) | null = null
  name: string
  url: string
  ssrUrl: string
  container: HTMLElement | ShadowRoot | null =
null
  inline: boolean
  scopecss: boolean
  useSandbox: boolean
  baseroute = ''
  source: sourceType
  sandBox: SandBoxInterface | null = null

  constructor ({
    name,
```

```
    url,
    ssrUrl,
    container,
    inline,
    scopecss,
    useSandbox,
    baseroute,
  }: CreateAppParam) {
    this.container = container ?? null // shadowRoot
    this.inline = inline ?? false // 内嵌js
    this.baseroute = baseroute ?? '' // 子应用路由路径
    this.ssrUrl = ssrUrl ?? ''
    // optional during init👆
    this.name = name // 名字
    this.url = url // 路径
    this.useSandbox = useSandbox // 沙箱
    this.scopecss = this.useSandbox && scopecss // 作用域css
    this.source = { // 暂存资源  link、scripts
      links: new Map<string, sourceLinkInfo>(),
      scripts: new Map<string, sourceScriptInfo>(),
    }
    this.loadSourceCode() // 加载资源代码
```

```
    this.useSandbox && (this.sandBox = new
SandBox(name, url)) // 创建沙箱
  }
  // Load resources
  loadSourceCode (): void {
    // 加载资源并且调用run方法
    this.state = appStates.LOADING_SOURCE_CODE
    HTMLLoader.getInstance().run(this,
extractSourceDom)
  }
}
```

3.资源加载逻辑

```
public run (app: AppInterface, successCb:
CallableFunction): void {
  const appName = app.name
  const htmlUrl = app.ssrUrl || app.url
  // 通过fetch加载应用
  fetchSource(htmlUrl, appName, { cache: 'no-
cache' }).then((htmlStr: string) => {
    if (!htmlStr) {
      const msg = 'html is empty, please check
in detail'
      app.onerror(new Error(msg))
      return logError(msg, appName)
    }
```

```
    // 使用插件处理html内容 , 并且将head -> micro-app-head / body -> micro-app-body
    htmlStr = this.formatHTML(htmlUrl, htmlStr, appName)
    // 将处理好的字符串传递到cb中
    successCb(htmlStr, app)
  }).catch((e) => {
    logError(`Failed to fetch data from ${app.url}, micro-app stop rendering`, appName, e)
    app.onLoadError(e)
  })
}
```

```
export function extractSourceDom (htmlStr: string, app: AppInterface) {
  const wrapElement = getWrapElement(htmlStr) // 将字符串放入到div中
  const microAppHead = wrapElement.querySelector('micro-app-head') // 找到头部
  const microAppBody = wrapElement.querySelector('micro-app-body') // 找到身体

  if (!microAppHead || !microAppBody) {
```

```
    const msg = `element ${microAppHead ? 'body' : 'head'} is missing`
    app.onerror(new Error(msg))
    return logError(msg, app.name)
  }
  // 处理子元素，将css和js进行抽离，style标签增加scopecss
  flatChildren(wrapElement, app, microAppHead)
  // fetch link标签，增加到style标签中同时采用scopecss来处理
  if (app.source.links.size) {
    fetchLinksFromHtml(wrapElement, app, microAppHead)
  } else {
    app.onLoad(wrapElement)
  }
  // 将script脚本进行加载
  if (app.source.scripts.size) {
    fetchScriptsFromHtml(wrapElement, app)
  } else {
    app.onLoad(wrapElement)
  }
}
```

4.沙箱加载

```
// 1)通过proxy创建代理window
this.proxyWindow =
this.createProxyWindow(appName)
// 2)给代理window进行初始化
this.initMicroAppWindow(this.microAppWindow,
appName, url)
// 3)重写事件和定时器,可用于卸载应用
Object.assign(this, effect(this.microAppWindow))
```

5.应用挂载逻辑

```
mount (
  container?: HTMLElement | ShadowRoot,
  inline?: boolean,
  baseroute?: string,
): void {
  if (isBoolean(inline) && inline !==
this.inline) {
    this.inline = inline
  }

  this.container = this.container ?? container!
    this.baseroute = baseroute ?? this.baseroute
```

```
  if (this.loadSourceLevel !== 2) {
    this.state = appStates.LOADING_SOURCE_CODE
    return
  }
  // 挂载前
  dispatchLifecyclesEvent(
    this.container,
    this.name,
    lifeCycles.BEFOREMOUNT,
  )
  // 更改状态为挂载中
  this.state = appStates.MOUNTING
  // 将内容移动到组件内部
  cloneContainer(this.source.html as Element,
this.container as Element, !this.umdMode)
  // 开启沙箱
  this.sandBox?.start(this.baseroute)

  let umdHookMountResult: any // result of mount
function

  if (!this.umdMode) {
    let hasDispatchMountedEvent = false
    // js全部执行，并且bindScope为代理proxy
    execScripts(this.source.scripts, this,
(isFinished: boolean) => {
      if (!this.umdMode) {
```

```
        // 看是否是umd格式，如果是umd格式通过类库的名字获取mount和unmount方法
        const { mount, unmount } = this.getUmdLibraryHooks()
        if (isFunction(mount) && isFunction(unmount)) {
          this.umdHookMount = mount as Func
          this.umdHookUnmount = unmount as Func
          // umd模式
          this.umdMode = true
          // 记录umd快照
          this.sandBox?.recordUmdSnapshot()
          try {
            umdHookMountResult = this.umdHookMount()
          } catch (e) {
            logError('an error occurred in the mount function \n', this.name, e)
          }
        }
      }
      if (!hasDispatchMountedEvent && (isFinished === true || this.umdMode)) {
        hasDispatchMountedEvent = true
        //  触发挂载完成
        this.handleMounted(umdHookMountResult)
      }
```

```
    })
  } else {
    this.sandBox?.rebuildUmdSnapshot() // 重构umd
快照
    try {
      umdHookMountResult = this.umdHookMount!()
    } catch (e) {
      logError('an error occurred in the mount
function \n', this.name, e)
    }
    // 触发挂载完成
    this.handleMounted(umdHookMountResult)
  }
}
```