QianKun前端微服务技术栈总结

一、沙箱机制

1.1 原理

1. **全局变量隔离**：
   - 通过代理（Proxy）对象来拦截和管理全局变量（如 `window` 对象）的读写操作，实现全局变量的隔离。
   - 当微应用尝试访问或修改全局变量时，沙箱会捕获这些操作并进行处理，确保不会影响其他微应用。
2. **样式隔离**：
   - 乾坤使用 Shadow DOM 或 scoped CSS 来隔离微应用的样式，防止样式冲突。
   - 对于不支持 Shadow DOM 的浏览器，乾坤会通过 CSS 前缀或其他方式来实现样式隔离。
3. **事件隔离**：
   - 乾坤会拦截和管理全局事件（如 `click`、`resize` 等），确保事件不会跨微应用传播。
   - 通过事件代理和事件委托，实现事件的精确控制和隔离。
4. **生命周期管理**：
   - 乾坤为每个微应用定义了详细的生命周期钩子，包括 `bootstrap`、`mount` 和 `unmount`，确保微应用在不同阶段的行为可控。
   - 在 `unmount` 阶段，乾坤会清理微应用的全局变量、事件监听器等，确保微应用卸载后不会留下残留。

1.2 代码实现实例

这段代码实现了一个简单的沙箱类 `Sandbox`，用于隔离微应用的全局变量操作，避免对原始 `window` 对象产生影响。下面我们将逐步解读这段代码。

5. 类的定义和构造函数

代码块

```
// 沙箱类
class Sandbox {
 constructor() {
 this.originalWindow = window; // 保存原始的 window 对象
 this.proxyWindow = new Proxy(window, {
```

```
  get: (target, key) => {
  // 检查是否已经存在隔离的变量
  if (this[key] !== undefined) {
  return this[key];
  }
  return target[key];
  },
  set: (target, key, value) => {
  // 检查是否已经存在隔离的变量
  if (this[key] !== undefined) {
  this[key] = value;
  return true;
  }
  target[key] = value;
  return true;
  }
  });
  }
 }
```

- **类的定义**：`class Sandbox` 定义了一个名为 `Sandbox` 的类，用于创建沙箱实例。
- **构造函数 `constructor`**：在创建 `Sandbox` 类的实例时，会自动调用构造函数。
  - `this.originalWindow = window;`：将当前的 `window` 对象保存到 `originalWindow` 属性中，以便后续恢复原始的 `window` 对象。
  - `this.proxyWindow = new Proxy(window, {...})`：使用 `Proxy` 对象创建一个代理 `window` 对象，用于拦截对 `window` 对象的属性访问和赋值操作。
  - **`get` 拦截器**：当访问 `proxyWindow` 的属性时，会触发 `get` 拦截器。如果 `this` 对象中已经存在该属性，则返回该属性的值；否则，返回原始 `window` 对象的属性值。
  - **`set` 拦截器**：当给 `proxyWindow` 的属性赋值时，会触发 `set` 拦截器。如果 `this` 对象中已经存在该属性，则将新值赋给该属性；否则，将新值赋给原始 `window` 对象的属性。

2. 激活沙箱方法 `activate`

代码块

```
  activate() {
  // 激活沙箱，将 window 替换为 proxyWindow
  window = this.proxyWindow;
  }
```

- `activate` 方法用于激活沙箱，将全局的 `window` 对象替换为 `proxyWindow`。这样，后续对 `window` 对象的属性访问和赋值操作都会被 `Proxy` 对象拦截。

6. 停用沙箱方法 `deactivate`

代码块

```
deactivate() {
// 恢复原始的 window 对象
window = this.originalWindow;
}
```

- `deactivate` 方法用于停用沙箱，将全局的 `window` 对象恢复为原始的 `window` 对象。

7. 清理沙箱方法 `clear`

代码块

```
clear() {
// 清理沙箱中的所有变量
for (const key in this) {
if (this.hasOwnProperty(key) && key !== 'originalWindow' && key !==
'proxyWindow') {
delete this[key];
}
}
}
```

- `clear` 方法用于清理沙箱中的所有变量。通过遍历 `this` 对象的所有属性，删除除 `originalWindow` 和 `proxyWindow` 之外的所有属性。

8. 使用沙箱

代码块

```
// 使用沙箱
const sandbox = new Sandbox();
// 激活沙箱
sandbox.activate();
// 模拟微应用的全局变量操作
window.myVar = 'Hello, Qiankun!';
// 检查沙箱中的全局变量
console.log(sandbox.myVar); // 输出: Hello, Qiankun!
// 恢复原始的 window 对象
sandbox.deactivate();
// 清理沙箱
sandbox.clear();
```

```
// 检查原始的 window 对象
console.log(window.myVar); // 输出: undefined
```

- `const sandbox = new Sandbox();`：创建一个 `Sandbox` 类的实例。
- `sandbox.activate();`：激活沙箱，将全局的 `window` 对象替换为 `proxyWindow`。
- `window.myVar = 'Hello, Qiankun!';`：模拟微应用的全局变量操作，给 `window` 对象的 `myVar` 属性赋值。
- `console.log(sandbox.myVar);`：检查沙箱中的全局变量，由于 `myVar` 是在沙箱激活期间赋值的，所以会输出 `Hello, Qiankun!`。
- `sandbox.deactivate();`：停用沙箱，将全局的 `window` 对象恢复为原始的 `window` 对象。
- `sandbox.clear();`：清理沙箱中的所有变量。
- `console.log(window.myVar);`：检查原始的 `window` 对象，由于沙箱已经停用并清理，所以 `window.myVar` 为 `undefined`。 通过以上步骤，我们可以看到，使用 `Sandbox` 类可以实现对微应用的全局变量操作的隔离，避免对原始 `window` 对象产生影响。

二、样式隔离原理

Shadow DOM 、Scoped CSS

2.1 Shadow DOM 隔离

利用浏览器原生支持的Shadow DOM API，为子应用创建独立的DOM子树，其内部样式与外部完全隔离。

代码块

```
// 主应用配置子应用
registerMicroApps([
  {
    name: 'subApp',
    entry: '//localhost:7100',
    container: '#subapp-container',
    activeRule: '/sub-app',
    props: {
      sandbox: {
        strictStyleIsolation: true // 启用Shadow DOM
      }
    }
  },
]);
```

优点：- **强隔离性**：Shadow DOM内部的CSS选择器不会影响外部，反之亦然。

- **原生支持**：无需额外处理CSS，由浏览器保证隔离性。

缺点：

- **兼容性限制**：部分旧浏览器不支持（如IE11）。
- **UI组件穿透问题**：如Ant Design的Modal组件可能因挂载到 `body` 导致样式失效。
- **事件穿透复杂**：需手动处理Shadow DOM内外的事件通信。

2.2 Scoped CSS

乾坤通过动态重写子应用的CSS规则，为每个选择器添加唯一前缀（如 `div` → `[qiankun-subapp] div`），将样式限制在子应用容器内。

代码块

```
// 主应用配置
start({
  sandbox: {
    experimentalStyleIsolation: true // 启用动态作用域
  }
});
```

优点：

- **无侵入性**：无需修改子应用代码，乾坤自动处理样式作用域。
- **兼容性佳**：支持所有浏览器。
- **灵活可控**：支持动态加载/卸载子应用样式表。

缺点：

- **性能损耗**：动态重写CSS可能影响大型应用的加载性能。
- **动态样式失效**：通过JavaScript插入的样式需额外处理（如监听DOM变化）。

技术对比与选型建议

方案	适用场景	注意事项
Shadow DOM	高隔离需求、现代浏览器环境	处理全局组件（如弹窗）、事件通信
动态样式表作用域	兼容旧浏览器、快速迁移现有项目	监控动态插入样式、性能优化

乾坤样式隔离的实现细节

1. **CSS重写机制**：
2. 乾坤劫持 `document.createElement` 等方法，在子应用加载时解析其CSS文本，通过正则匹配重写选择器，例如：

```
代码块
1  /* 原始CSS */ .button { color: red; }
2  /* 重写后 */ [data-qiankun-subapp] .button { color: red; }
```

1. **样式卸载策略**：
   子应用卸载时，乾坤自动移除其动态注入的 `<style>` 标签，避免残留样式影响。
2. **第三方库适配**：
   针对UI库（如Ant Design）的全局样式，可通过配置 `excludeAssetFilter` 排除特定资源，或在子应用内使用定制前缀。

常见问题

1. **弹窗组件样式失效**
   - **方案**：将弹窗挂载到子应用容器内，而非 `document.body` 。
   - **代码示例**：

```
代码块
1  // 子应用修改Modal挂载点
2  Modal.config({ rootSelector: '#subapp-container' });
```

2. **动态加载样式丢失**
   - **方案**：监听 `DOMNodeInserted` 事件，对新增 `<style>` 标签自动重写。
3. **字体图标跨域问题**
   - **方案**：在主应用配置跨域头，或通过乾坤的 `fetch` 劫持重定向资源路径。

三、如何同时启动子应用

3.1开发环境

3.1.1 主应用配置

安装

```
npm install qiankun
```

然后在主应用代码中进行如下配置（以 React 为例）：

代码块

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import { registerMicroApps, start } from 'qiankun';
// 注册子应用
registerMicroApps([
  {
    name: 'sub-app1', // 子应用名称
    entry: '//localhost:8081', // 子应用入口地址
    container: '#sub-app1-container', // 子应用挂载节点
    activeRule: '/sub-app1', // 子应用激活规则
  },
  {
    name: 'sub-app2',
    entry: '//localhost:8082',
    container: '#sub-app2-container',
    activeRule: '/sub-app2',
  },
  // 可以根据需要添加更多子应用配置
]);
// 启动 qiankun
start();
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

3.1.2 子应用配置

子应用的 `package.json` 中添加启动脚本：

代码块

```
{
  "scripts": {
    "start": "webpack-dev-server --config webpack.config.js"
  }
}
```

3.1.3 同时启动多个应用

可以使用 `concurrently` 工具来同时启动主应用和多个子应用。先安装 `concurrently`：

代码块

```
npm install concurrently --save-dev
```

然后在主应用的 `package.json` 中修改 `scripts`：

代码块

```
{
  "scripts": {
    "start:main": "webpack-dev-server --config webpack.config.js",
    "start:sub-app1": "cd ../sub-app1 && npm start",
    "start:sub-app2": "cd ../sub-app2 && npm start",
    "start": "concurrently \"npm run start:main\" \"npm run start:sub-app1\" \"npm run start:sub-app2\""
  }
}
```

3.2 生产环境

在生产环境中，通常会先将主应用和子应用分别构建打包，然后部署到服务器上。

将构建好的主应用和子应用部署到服务器上，确保主应用能够正确访问子应用的入口地址。主应用的代码和开发环境基本一致，通过 `registerMicroApps` 注册子应用并启动 `qiankun`。当用户访问主应用时，`qiankun` 会根据激活规则加载相应的子应用。

四、子任务之间通信

三种实现方式

4.1 通过主应用作为中间媒介通信

主应用可以作为不同子应用之间通信的桥梁，利用 `qiankun` 的 props 机制传递数据和事件

主应用配置

在主应用中注册子应用时，可以通过 `props` 传递一个通信对象，该对象包含数据和方法。

代码块

```
import { registerMicroApps, start } from 'qiankun';
// 定义通信对象
const communication = {
    data: {},
```

```
    setData: (key, value) => {
        communication.data[key] = value;
    },
    getData: (key) => {
        return communication.data[key];
    }
};
registerMicroApps([
    {
        name: 'subApp1',
        entry: '//localhost:8081',
        container: '#sub-app1',
        activeRule: '/sub-app1',
        props: { communication }
    },
    {
        name: 'subApp2',
        entry: '//localhost:8082',
        container: '#sub-app2',
        activeRule: '/sub-app2',
        props: { communication }
    }
]);
start();
```

子应用接收和使用通信对象

在子应用中可以通过 `props` 接收通信对象，并使用其中的方法进行数据的设置和获取。

代码块

```
// 子应用 1
export async function mount(props) {
    const { communication } = props;
    // 设置数据
    communication.setData('message', 'Hello from subApp1');
}
// 子应用 2
export async function mount(props) {
    const { communication } = props;
    // 获取数据
    const message = communication.getData('message');
    console.log(message);
}
```

4.2 使用全局事件总线

可以创建一个全局的事件总线，让不同的子应用都可以监听和触发事件。

创建全局事件总线

可以在主应用中创建一个事件总线对象，并通过 `props` 传递给子应用。

代码块

```
import { EventEmitter } from 'events';
// 创建事件总线
const eventBus = new EventEmitter();
registerMicroApps([
    {
        name: 'subApp1',
        entry: '//localhost:8081',
        container: '#sub-app1',
        activeRule: '/sub-app1',
        props: { eventBus }
    },
    {
        name: 'subApp2',
        entry: '//localhost:8082',
        container: '#sub-app2',
        activeRule: '/sub-app2',
        props: { eventBus }
    }
]);
start();
```

子应用监听和触发事件

子应用可以通过 `eventBus` 监听和触发事件。

代码块

```
// 子应用 1
export async function mount(props) {
    const { eventBus } = props;
    // 触发事件
    eventBus.emit('message', 'Hello from subApp1');
}
// 子应用 2
export async function mount(props) {
    const { eventBus } = props;
    // 监听事件
```

```
    eventBus.on('message', (message) => {
        console.log(message);
    });
}
```

4.3 使用浏览器的本地存储或会话存储

可以利用浏览器的 `localStorage` 或 `sessionStorage` 在不同子应用之间共享数据。

子应用存储数据

代码块

```
// 子应用 1
function sendData() {
    localStorage.setItem('message', 'Hello from subApp1');
}
```

子应用获取数据

代码块

```
// 子应用 2
function receiveData() {
    const message = localStorage.getItem('message');
    console.log(message);
}
```

注意事项

- **数据同步问题**：使用本地存储或会话存储时，需要注意数据的同步问题，因为不同子应用可能会在不同时间点读取和写入数据。
- **事件清理**：使用事件总线时，要在子应用卸载时清理事件监听，避免内存泄漏。例如：

代码块

```
export async function unmount(props) {
    const { eventBus } = props;
    eventBus.removeAllListeners('message');
}
```

五、实战

5.1 开发环境

在 `qiankun` 微前端架构里，子应用以 `mount` 函数形式暴露其挂载逻辑，虽代码里没直接调用 `mount` 函数，但 `qiankun` 主应用会在合适时机自动调用它来启动子应用，下面详细解释启动机制和相关要点。

1. `qiankun` 自动调用 `mount` 函数机制

`qiankun` 主应用借助 `registerMicroApps` 函数注册子应用，当用户访问的路由匹配子应用的 `activeRule` 时，`qiankun` 会按如下步骤操作：

- **加载子应用资源**：根据子应用的 `entry` 地址，加载子应用打包后的 HTML、JavaScript 和 CSS 等资源。
- **调用 `mount` 函数**：加载完子应用资源后，`qiankun` 会自动调用子应用导出的 `mount` 函数，并传递 `props` 参数，进而启动子应用的渲染流程。

4. 代码示例说明

主应用代码

代码块

```
import { registerMicroApps, start } from 'qiankun';
// 注册子应用
registerMicroApps([
  {
    name: 'vue-sub-app',
    entry: '//localhost:8082', // 子应用入口地址
    container: '#vue-sub-app-container', // 子应用挂载容器
    activeRule: '/vue-sub-app', // 子应用激活规则
    props: {
      // 可以传递一些数据或事件总线等
      someData: '这是主应用传递给子应用的数据'
    }
  }
]);
// 启动 qiankun
start();
```

在上述代码中，主应用通过 `registerMicroApps` 注册了一个 Vue 子应用。当用户访问的 URL 匹配 `/vue-sub-app` 时，`qiankun` 会加载 `//localhost:8082` 地址的子应用资源，并调用子应用的 `mount` 函数。

子应用代码（Vue 子应用）

```
import Vue from 'vue';
import App from './App.vue';
export async function mount(props) {
  const { someData } = props;
  // 监听事件示例
  const handleMessage = (message) => {
    console.log('Received message in Vue sub-app:', message);
  };
  // 假设这里有一个事件总线 eventBus 来监听事件
  // eventBus.on('message', handleMessage);
  // 触发事件示例
  const sendMessage = () => {
    // eventBus.emit('message', 'Hello from Vue sub-app');
  };
  new Vue({
    render: (h) =>
      h(App, {
        props: {
          sendMessage,
          someData
        }
      })
  }).$mount('#app');
  return () => {
    // 子应用卸载时移除事件监听
    // eventBus.removeListener('message', handleMessage);
  };
}
```

在这个子应用代码中，`mount` 函数接收主应用传递的 `props`，并在其中创建 Vue 实例并挂载到 `#app` 元素上。

5. 开发与测试流程

开发开发

- **启动子应用**：在子应用项目目录下，运行启动命令（如 `npm run serve`），启动子应用的开发服务器，使其能在指定端口（如 `localhost:8082`）访问。
- **启动主应用**：在主应用项目目录下，运行启动命令（如 `npm start`），启动主应用的开发服务器。
- **访问子应用**：在浏览器中访问主应用，当访问的 URL 匹配子应用的 `activeRule` 时，`qiankun` 会自动加载并启动子应用。

生产环境

- **打包子应用**：在子应用项目目录下，运行打包命令（如 `npm run build`），将子应用打包成静态资源。
- **部署子应用**：将打包后的子应用静态资源部署到服务器上。
- **打包并部署主应用**：对主应用进行打包并部署到服务器，确保主应用能正确访问子应用的资源地址。

5.2 打包后 `mount` 函数名问题

在使用 Webpack 等打包工具时，默认情况下，`mount` 函数名不会被修改。因为在 JavaScript 里，函数作为导出的模块成员，打包工具会保留其原始名称以保证模块间的正确引用。

不过，若使用了混淆压缩工具且配置不当，可能会导致函数名被修改。为避免这种情况，可在打包配置里排除对导出函数名的混淆。例如，在 Webpack 中使用 `terser-webpack-plugin` 时，可如下配置：

代码块

```
const TerserPlugin = require('terser-webpack-plugin');
module.exports = {
    optimization: {
        minimizer: [
            new TerserPlugin({
                terserOptions: {
                    keep_classnames: true,
                    keep_fnames: true // 保留函数名
                }
            })
        ]
    }
};
```

5.3 应用判断qiankun环境

Vue 子应用入口文件 `main.js`

代码块

```
import Vue from 'vue';
import App from './App.vue';
// 判断是否处于 qiankun 环境
if (!window.
__POWERED_BY_QIANKUN__
) {
    // 独立运行时，直接渲染应用
    new Vue({
```

```
        render: h => h(App)
    }).$mount('#app');
} else {
    // 作为 qiankun 子应用运行时，导出生命周期函数
    let instance;
    export async function mount(props) {
        const { container } = props;
        instance = new Vue({
            render: h => h(App)
        }).$mount(container ? container.querySelector('#app') : '#app');
    }
    export async function unmount() {
        instance.
$destroy();
        instance.$
el.innerHTML = '';
        instance = null;
    }
}
```

详细解释

- **独立运行逻辑**：
  - `if (!window.__POWERED_BY_QIANKUN__)` 条件判断当前环境是否为独立运行环境。若为 `true`，则表明子应用是独立运行的，此时按照传统 Vue 应用的方式创建并挂载 Vue 实例。
- **`qiankun` 子应用逻辑**：
  - 若 `window.__POWERED_BY_QIANKUN__` 为 `true`，则表明子应用处于 `qiankun` 沙箱环境中。此时导出 `mount` 和 `unmount` 生命周期函数。
  - `mount` 函数：接收 `props` 参数，其中包含 `container`，用于指定子应用的挂载节点。创建 Vue 实例并将其挂载到该节点上。
  - `unmount` 函数：在子应用卸载时被调用，负责销毁 Vue 实例并清空挂载节点的内容。