作为面试官，为什么我推荐微前端作为前端面试的亮点？

> 前段时间陆续面试了一波候选人，其中提到最多的就是微前端方案，微前端不像前端框架的面试题那样，它更偏重于项目实战，更加考察候选人的技术水平，不像 React,Vue 随便一问，就是各种响应式原理 ,Fiber 架构等等烂大街的。

为什么选择微前端作为项目亮点

> 如果你的简历平平无奇，面试官实在在你的简历上问不出什么，那么只能给你上点“手写题”强度了

作为面试官，我经常听到很多候选人说在公司做的项目很简单，平常就是堆页面，写管理端，写 H5, 没有任何亮点，我以我一次面试候选人的经历分享给大家

面试官：你为什么选择用微前端做管理端升级，你的项目很庞大么？

候选人： 不是的，其实是我们把两个管理端合并，让用户方便使用。

面试官：咦，竟然这样你们还不如用 a 标签链接或者 nginx 转发一下就行了，更加方便，杀鸡焉用牛刀啊

候选人：为了让客户体验到单页面的感觉，体验感更好

面试官：enen....

从这里你会觉得候选人的想法有点奇葩，但是换个角度来想，一定要等到项目庞大拆服务了才用微前端么，我管理端项目一开始就上微前端不行么。其实从这里可以看出来，管理系统使用微前端的成本并不会太大，而且从后面的技术问答中，候选人的微前端还是挺优秀的，各个细节基本都涉略到了。

如果你在公司内部很闲，又是刚好负责无关紧要的运营管理端，那么新的管理端可以一开始接入微前端方案，为未来的技术升级提供一个接口，风险也可控，还能够倒腾技术，简历还能新

增亮点，何乐而不为

另外提到 H5 了，就提多一嘴，H5 面向 C 端用户比较多，这方面更应该关心一些性能指标数据，比如 `FP`，`FCP` 等等，围绕这些指标进行优化，亮点不就来了么，这类例子比比皆是，要学会多挖掘

接下来是我作为面试官，经常考察候选人的问题，因为大部分候选人都是用 qiankun 框架，所以本文以 qiankun 框架为模板，重点剖析项目实战中微前端中遇到的问题和原理

请解释一下微前端的概念以及它的主要优点和挑战？

微前端是一种将不同的前端应用组合到一起的架构模式。这些应用可以独立开发、独立部署、独立运行，然后在一个主应用中进行集成。这种模式的主要目标是解决大型、长期演进的前端

项目的复杂性问题。

主要优点：

1. **解耦：** 微前端架构可以将大型项目分解为多个可以独立开发、测试和部署的小型应用。这种解耦可以提高开发效率，减少团队间的协调成本。
2. **技术栈无关：** 不同的微前端应用可以使用不同的技术栈，这为使用新技术、升级旧技术提供了可能。
3. **并行开发：** 因为微前端应用是独立的，所以多个团队可以并行开发不同的应用，无需担心相互影响。
4. **独立部署：** 每个微前端应用可以独立部署，这意味着可以更快地推出新功能，同时降低了部署失败的风险。

主要挑战：

1. **性能问题：** 如果不同的微前端应用使用了不同的库或框架，可能会导致加载和运行的性能问题。
2. **一致性：** 保持不同的微前端应用在用户体验、设计和行为上的一致性可能会比较困难。
3. **状态共享：** 在微前端应用之间共享状态可能会比较复杂，需要使用特殊的工具或模式。
4. **复杂性：** 尽管微前端可以解决大型项目的复杂性问题，但是它自身也带来了一些复杂性，比如需要管理和协调多个独立的应用。
5. **安全性：** 微前端架构可能会增加跨域等安全问题。

你能详细描述一下 qiankun 微前端框架的工作原理吗？

> qiankun 是一个基于 single-spa 的微前端实现框架。它的工作原理主要涉及到以下几个方面：

1. **应用加载** ： qiankun 通过动态创建 script 标签的方式加载子应用的入口文件。加载完成后，会执行子应用暴露出的生命周期函数。
2. **生命周期管理** ： qiankun 要求每个子应用都需要暴露出 bootstrap、mount 和 unmount 三个生命周期函数。bootstrap 函数在应用加载时被调用，mount 函数在应用启动时被调

用，unmount 函数在应用卸载时被调用。

3. **沙箱隔离**： qiankun 通过 Proxy 对象创建了一个 JavaScript 沙箱，用于隔离子应用的全局变量，防止子应用之间的全局变量污染。
4. **样式隔离**： qiankun 通过动态添加和移除样式标签的方式实现了样式隔离。当子应用启动时，会动态添加子应用的样式标签，当子应用卸载时，会移除子应用的样式标签。
5. **通信机制**： qiankun 提供了一个全局的通信机制，允许子应用之间进行通信。

在使用 qiankun 时，如果子应用是基于 jQuery 的多页应用，你会如何处理静态资源的加载问题？

> 在使用 qiankun 时，如果子应用是基于 jQuery 的多页应用，静态资源的加载问题可能会成为一个挑战。这是因为在微前端环境中，子应用的静态资源路径可能需要进行特殊处理才能正确加载。这里有几种可能的解决方案：

方案一：使用公共路径

在子应用的静态资源路径前添加公共路径前缀。例如，如果子应用的静态资源存放在 `http://localhost:8080/static/`，那么可以在所有的静态资源路径前添加这个前缀。

方案二：劫持标签插入函数

这个方案分为两步：

1. 对于 HTML 中已有的 img/audio/video 等标签，qiankun 支持重写 getTemplate 函数，可以将入口文件 index.html 中的静态资源路径替换掉。
2. 对于动态插入的 img/audio/video 等标签，劫持 appendChild、innerHTML、insertBefore 等事件，将资源的相对路径替换成绝对路径。

例如，我们可以传递一个 getTemplate 函数，将图片的相对路径转为绝对路径，它会在处理模板时使用：

TypeScript

```
start({
  getTemplate(tpl,...rest) {
    // 为了直接看到效果，所以写死了，实际中需要用正则匹配
    return tpl.replace('<img src="./img/jQuery1.png">', '<img src="http://localhost:3333/img/jQuery1.png">');
  }
});
```

对于动态插入的标签，劫持其插入 DOM 的函数，注入前缀。

TypeScript

```
beforeMount: app => {
    if(app.name === 'purehtml'){
        // jQuery 的 html 方法是一个挺复杂的函数，这里只是为了看效果，简写了
        $.prototype.html = function(value){
            const str = value.replace('<img src="/img/jQuery2.png">', '<img src="http://localhost:3333/img/jQuery2.png">')
            this[0].innerHTML = str;
        }
    }
}
```

方案三：给 jQuery 项目加上 webpack 打包

这个方案的可行性不高，都是陈年老项目了，没必要这样折腾。

在使用 `qiankun` 时，如果子应用动态插入了一些标签，你会如何处理？

> 在使用 `qiankun` 时，如果子应用动态插入了一些标签，我们可以通过劫持 DOM 的一些方法来处理。例如，我们可以劫持 `appendChild`、`innerHTML` 和 `insertBefore` 等方法，将资源的相对路径替换为绝对路径。

以下是一个例子，假设我们有一个子应用，它使用 jQuery 动态插入了一张图片：

TypeScript

```
const render = $ => {
  $('#app-container').html('<p>Hello, render with jQuery</p><img src="./img/my-image.png">');
  return Promise.resolve();
};
```

我们可以在主应用中劫持 jQuery 的 `html` 方法，将图片的相对路径替换为绝对路径：

TypeScript

```
beforeMount: app => {
    if(app.name === 'my-app'){
        // jQuery 的 html 方法是一个复杂的函数，这里为了简化，我们只处理 img 标签
        $.prototype.html = function(value){
            const str = value.replace('<img src="./img/my-image.png">', '<img src="http://localhost:8080/img/my-image.png">')
            this[0].innerHTML = str;
        }
    }
}
```

在这个例子中，我们劫持了 jQuery 的 `html` 方法，将图片的相对路径 `./img/my-image.png` 替换为了绝对路径 `http://localhost:8080/img/my-image.png`。这样，无论子应用在哪里运行，图片都可以正确地加载。

在使用 `qiankun` 时，你如何处理老项目的资源加载问题？你能给出一些具体的解决方案吗？

> 在使用 `qiankun` 时，处理老项目的资源加载问题可以有多种方案，具体的选择取决于项目的具体情况。以下是一些可能的解决方案：

1. **使用 `qiankun` 的 `getTemplate` 函数重写静态资源路径**：对于 HTML 中已有的 `img/audio/video` 等标签，`qiankun` 支持重写 `getTemplate` 函数，可以将入口文件 `index.html` 中的静态资源路径替换掉。例如：

TypeScript

```
start({
  getTemplate(tpl,...rest) {
    // 为了直接看到效果，所以写死了，实际中需要用正则匹配
    return tpl.replace('<img src="./img/my-image.png">', '<img src="http://localhost:8080/img/my-image.png">');
  }
});
```

1. **劫持标签插入函数**：对于动态插入的 `img/audio/video` 等标签，我们可以劫持 `appendChild` 、 `innerHTML` 、 `insertBefore` 等事件，将资源的相对路径替换成绝对路径。例如，我们可以劫持 jQuery 的 `html` 方法，将图片的相对路径替换为绝对路径：

TypeScript

```
beforeMount: app => {
    if(app.name === 'my-app'){
        $.prototype.html = function(value){
            const str = value.replace('<img src="./img/my-image.png">', '<img src="http://localhost:8080/img/my-image.png">')
            this[0].innerHTML = str;
        }
    }
}
```

1. **给老项目加上 webpack 打包**：这个方案的可行性不高，都是陈年老项目了，没必要这样折腾。
2. **使用 iframe 嵌入老项目**：虽然 `qiankun` 支持 jQuery 老项目，但是似乎对多页应用没有很好的解决办法。每个页面都去修改，成本很大也很麻烦，但是使用 iframe 嵌入这些老项目就比较方便。

你能解释一下 `qiankun` 的 `start` 函数的作用和参数吗？如果只有一个子项目，你会如何启用预加载？

> `qiankun` 的 `start` 函数是用来启动微前端应用的。在注册完所有的子应用之后，我们需要调用 `start` 函数来启动微前端应用。

`start` 函数接收一个可选的配置对象作为参数，这个对象可以包含以下属性：

- `prefetch`：预加载模式，可选值有 `true`、`false`、`'all'`、`'popstate'`。默认值为 `true`，即在主应用 `start` 之后即刻开始预加载所有子应用的静态资源。如果设置为 `'all'`，则主应用 `start` 之后会预加载所有子应用静态资源，无论子应用是否激活。如果设置为 `'popstate'`，则只有在路由切换的时候才会去预加载对应子应用的静态资源。

- `sandbox` ：沙箱模式，可选值有 `true` 、 `false` 、 `{ strictStyleIsolation: true }` 。默认值为 `true` ，即为每个子应用创建一个新的沙箱环境。如果设置为 `false` ，则子应用运行在当前环境下，没有任何的隔离。如果设置为 `{ strictStyleIsolation: true }` ，则会启用严格的样式隔离模式，即子应用的样式会被完全隔离，不会影响到其他子应用和主应用。
- `singular` ：是否为单例模式，可选值有 `true` 、 `false` 。默认值为 `true` ，即一次只能有一个子应用处于激活状态。如果设置为 `false` ，则可以同时激活多个子应用。
- `fetch` ：自定义的 `fetch` 方法，用于加载子应用的静态资源。

如果只有一个子项目，要想启用预加载，可以这样使用 `start` 函数：

TypeScript

```
start({ prefetch: 'all' });
```

这样，主应用 `start` 之后会预加载子应用的所有静态资源，无论子应用是否激活。

在使用 `qiankun` 时，你如何处理 `js` 沙箱不能解决的 `js` 污染问题？

> `qiankun` 的 `js` 沙箱机制主要是通过代理 `window` 对象来实现的，它可以有效地隔离子应用的全局变量，防止子应用之间的全局变量污染。然而，这种机制并不能解决所有的 `js` 污染问题。例如，如果我们使用 `onclick` 或 `addEventListener` 给 `<body>` 添加了一个点击事件， `js` 沙箱并不能消除它的影响。

对于这种情况，我们需要依赖于良好的代码规范和开发者的自觉。在开发子应用时，我们需要避免直接操作全局对象，如 `window` 和 `document` 。如果必须要操作，我们应该在子应用卸载时，清理掉这些全局事件和全局变量，以防止对其他子应用或主应用造成影响。

例如，如果我们在子应用中添加了一个全局的点击事件，我们可以在子应用的`unmount`生命周期函数中移除这个事件：

TypeScript

```typescript
export async function mount(props) {
  // 添加全局点击事件
  window.addEventListener('click', handleClick);
}

export async function unmount() {
  // 移除全局点击事件
  window.removeEventListener('click', handleClick);
}

function handleClick() {
  // 处理点击事件
}
```

这样，当子应用卸载时，全局的点击事件也会被移除，不会影响到其他的子应用。

你能解释一下`qiankun`如何实现`keep-alive`的需求吗？

> 在`qiankun`中，实现`keep-alive`的需求有一定的挑战性。这是因为`qiankun`的设计理念是在子应用卸载时，将环境还原到子应用加载前的状态，以防止子应用对全局环境造成污染。这种设计理念与`keep-alive`的需求是相悖的，因为`keep-alive`需要保留子应用的状态，而不是在子应用卸载时将其状态清除。

然而，我们可以通过一些技巧来实现`keep-alive`的效果。一种可能的方法是在子应用的生命周期函数中保存和恢复子应用的状态。例如，我们可以在子应用的`unmount`函数中保存子应用的状态，然后在`mount`函数中恢复这个状态：

TypeScript

```typescript
// 伪代码
let savedState;

export async function mount(props) {
  // 恢复子应用的状态
  if (savedState) {
    restoreState(savedState);
  }
}

export async function unmount() {
  // 保存子应用的状态
  savedState = saveState();
}

function saveState() {
  // 保存子应用的状态
  // 这个函数的实现取决于你的应用
}

function restoreState(state) {
  // 恢复子应用的状态
  // 这个函数的实现取决于你的应用
}
```

这种方法的缺点是需要手动保存和恢复子应用的状态，这可能会增加开发的复杂性。此外，这种方法也不能保留子应用的 DOM 状态，只能保留 JavaScript 的状态。

还有一种就是手动 `*loadMicroApp*` + `display:none`，直接隐藏 Dom

另一种可能的方法是使用 `single-spa` 的 `Parcel` 功能。`Parcel` 是 `single-spa` 的一个功能，它允许你在一个应用中挂载另一个应用，并且可以控制这个应用的生命周期。通过 `Parcel`，我们可以将子应用挂载到一个隐藏的 DOM 元素上，从而实现 `keep-alive` 的效

果。然而，这种方法需要对 `qiankun` 的源码进行修改，因为 `qiankun` 目前并不支持 `Parcel` 。

你能解释一下 `qiankun` 和 `iframe` 在微前端实现方式上的区别和优劣吗？在什么情况下，你会选择使用 `iframe` 而不是 `qiankun` ？

`qiankun` 和 `iframe` 都是微前端的实现方式，但它们在实现原理和使用场景上有一些区别。

`qiankun` 是基于 `single-spa` 的微前端解决方案，它通过 JavaScript 的 `import` 功能动态加载子应用，然后在主应用的 DOM 中挂载子应用的 DOM。 `qiankun` 提供了一种 JavaScript 沙箱机制，可以隔离子应用的全局变量，防止子应用之间的全局变量污染。此外， `qiankun` 还提供了一种样式隔离机制，可以防止子应用的 CSS 影响其他应用。这些特性使得 `qiankun` 在处理复杂的微前端场景时具有很高的灵活性。

`iframe` 是一种较为传统的前端技术，它可以在一个独立的窗口中加载一个 HTML 页面。 `iframe` 本身就是一种天然的沙箱，它可以完全隔离子应用的 JavaScript 和 CSS，防止子应用之间的相互影响。然而， `iframe` 的这种隔离性也是它的缺点，因为它使得主应用和子应用之间的通信变得困难。此外， `iframe` 还有一些其他的问题，比如性能问题、SEO 问题等。

在选择 `qiankun` 和 `iframe` 时，需要根据具体的使用场景来决定。如果你的子应用是基于现代前端框架（如 React、Vue、Angular 等）开发的单页应用，那么 `qiankun` 可能是一个更好的选择，因为它可以提供更好的用户体验和更高的开发效率。如果你的子应用是基于 jQuery 或者其他传统技术开发的多页应用，或者你需要在子应用中加载一些第三方的页面，那么 `iframe` 可能是一个更好的选择，因为它可以提供更强的隔离性。

在使用 `qiankun` 时，你如何处理多个子项目的调试问题？

在使用 `qiankun` 处理多个子项目的调试问题时，通常的方式是将每个子项目作为一个独立的应用进行开发和调试。每个子项目都可以在本地启动，并通过修改主应用的配置，让主应用去加载本地正在运行的子应用，这样就可以对子应用进行调试了。这种方式的好处是，子应用与主应用解耦，可以独立进行开发和调试，不会相互影响。

对于如何同时启动多个子应用，你可以使用 `npm-run-all` 这个工具。`npm-run-all` 是一个 CLI 工具，可以并行或者串行执行多个 npm 脚本。这个工具对于同时启动多个子应用非常有用。使用方式如下：

1. 首先，你需要在你的项目中安装 `npm-run-all` ，可以通过下面的命令进行安装：

```
TypeScript

1 npm install --save-dev npm-run-all
```

1. 然后，在你的 `package.json` 文件中定义你需要并行运行的脚本。比如，你有两个子应用，分别为 `app1` 和 `app2` ，你可以定义如下的脚本：

```
TypeScript

1 "scripts": {
2     "start:app1": "npm start --prefix ./app1",
3     "start:app2": "npm start --prefix ./app2",
4     "start:all": "npm-run-all start:app1 start:app2"
5 }
```

在这个例子中， `start:app1` 和 `start:app2` 脚本分别用于启动 `app1` 和 `app2` 应用，`start:all` 脚本则用于同时启动这两个应用。

1. 最后，通过执行 `npm run start:all` 命令，就可以同时启动 `app1` 和 `app2` 这两个应用了。

`npm-run-all` 不仅可以并行运行多个脚本，还可以串行运行多个脚本。在某些情况下，你可能需要按照一定的顺序启动你的应用，这时你可以使用 `npm-run-all` 的 `-s` 选项来串行执行脚本，例如： `npm-run-all -s script1 script2` ，这将会先执行 `script1` ，然后再执行 `script2` 。

qiankun 是如何实现 CSS 隔离的，该方案有什么缺点，还有其它方案么

`qiankun` 主要通过使用 `Shadow DOM` 来实现 CSS 隔离。

1. `Shadow DOM` ： `Shadow DOM` 是一种浏览器内置的 Web 标准技术，它可以创建一个封闭的 DOM 结构，这个 DOM 结构对外部是隔离的，包括其 CSS 样式。 `qiankun` 在挂载子应用时，会将子应用的 HTML 元素挂载到 `Shadow DOM` 上，从而实现 CSS 的隔离。

TypeScript

```
// qiankun使用Shadow DOM挂载子应用
const container = document.getElementById('container');
const shadowRoot = container.attachShadow({mode: 'open'});
shadowRoot.innerHTML = '<div id="subapp-container"></div>';
```

对于 `qiankun` 的隔离方案，一个潜在的缺点是它需要浏览器支持 `Shadow DOM` ，这在一些旧的浏览器或者不兼容 `Shadow DOM` 的浏览器中可能会出现问题。

另一种可能的方案是使用 CSS 模块（CSS Modules）。CSS 模块是一种将 CSS 类名局部化的方式，可以避免全局样式冲突。在使用 CSS 模块时，每个模块的类名都会被转换成一个唯一的名字，从而实现样式的隔离。

例如，假设你有一个名为 `Button` 的 CSS 模块：

TypeScript

```
/* Button.module.css */
.button {
    background-color: blue;
}
```

在你的 JavaScript 文件中，你可以这样引入并使用这个模块：

```typescript
import styles from './Button.module.css';

function Button() {
    return <button className={styles.button}>Click me</button>;
}
```

在这个例子中，`button`类名会被转换成一个唯一的名字，如`Button_button__xxx`，这样就可以避免全局样式冲突了。

3.BEM 命名规范隔离

qiankun 中如何实现父子项目间的通信？如果让你实现一套通信机制，你该如何实现？

- `Actions` 通信： `qiankun` 官方提供的通信方式，适合业务划分清晰，较简单的微前端应用。这种通信方式主要通过 `setGlobalState` 设置 `globalState` ，并通过 `onGlobalStateChange` 和 `offGlobalStateChange` 来注册和取消 `观察者` 函数，从而实现通信。
- 自己实现一套通信机制（可以思考一下如何追踪 State 状态，类似 Redux 模式）

1. **全局变量**：在全局（window）对象上定义共享的属性或方法。这种方式简单明了，但有可能导致全局污染，需要注意变量命名以避免冲突。
2. **自定义事件**：使用原生的 CustomEvent 或类似的第三方库来派发和监听自定义事件。这种方式避免了全局污染，更加符合模块化的原则，但可能需要更复杂的事件管理。
    - 2.1. **定义一个全局的通信对象**，例如 window.globalEvent，这个对象提供两个方法，emit 和 on。
    - 2.2. **emit 方法** 用于派发事件，接收事件名称和可选的事件数据作为参数。

- 2.3. **on 方法** 用于监听事件，接收事件名称和回调函数作为参数。当相应的事件被派发时，回调函数将被执行。

TypeScript

```typescript
window.globalEvent = {
  events: {},
  emit(event, data) {
    if (!this.events[event]) {
      return;
    }
    this.events[event].forEach(callback => callback(data));
  },
  on(event, callback) {
    if (!this.events[event]) {
      this.events[event] = [];
    }
    this.events[event].push(callback);
  },
};
```

1. 在主项目中使用 qiankun 注册子项目时，如何解决子项目路由的 hash 与 history 模式之争？

如果主项目使用 `history` 模式，并且子项目可以使用 `history` 或 `hash` 模式，这是 `qiankun` 推荐的一种形式。在这种情况下，子项目可以选择适合自己的路由模式，而且对于已有的子项目不需要做太多修改。但是子项目之间的跳转需要通过父项目的 `router` 对象或原生的 `history` 对象进行。

2. 如果主项目和所有子项目都采用 `hash` 模式，可以有两种做法：

- 使用 `path` 来区分子项目：这种方式不需要对子项目进行修改，但所有项目之间的跳转需要借助原生的 `history` 对象。

- 使用 `hash` 来区分子项目：这种方式可以通过自定义 `activeRule` 来实现，但需要对子项目进行一定的修改，将子项目的路由加上前缀。这样的话，项目之间的跳转可以直接使用各自的 `router` 对象或 `<router-link>` 。

3. 如果主项目采用`hash`模式，而子项目中有些采用`history`模式，这种情况下，子项目间的跳转只能借助原生的`history`对象，而不使用子项目自己的`router`对象。对于子项目，可以选择使用`path`或`hash`来区分不同的子项目。

在 qiankun 中，如果实现组件在不同项目间的共享，有哪些解决方案？

> 在项目间共享���件时，可以考虑以下几种方式：

1. **父子项目间的组件共享**：主项目加载时，将组件挂载到全局对象（如 `window` ）上，在子项目中直接注册使用该组件。
2. **子项目间的组件共享（弱依赖）**：通过主项目提供的全局变量，子项目挂载到全局对象上。子项目中的共享组件可以使用异步组件来实现，在加载组件前先检查全局对象中是否存在，存在则复用，否则加载组件。
3. **子项目间的组件共享（强依赖）**：在主项目中通过 `loadMicroApp` 手动加载提供组件的子项目，确保先加载该子项目。在加载时，将组件挂载到全局对象上，并将 `loadMicroApp` 函数传递给子项目。子项目在需要使用共享组件的地方，手动加载提供组件的子项目，等待加载完成后即可获取组件。

需要注意的是，在使用异步组件或手动加载子项目时，可能会遇到样式加载的问题，可以尝试解决该问题。另外，如果共享的组件依赖全局插件（如 `store` 和 `i18n` ），需要进行特殊处理以确保插件的正确初始化。

在 qiankun 中，应用之间如何复用依赖，除了 npm 包方案外？

1. 在使用 `webpack` 构建的子项目中，要实现复用公共依赖，需要配置 `webpack` 的 `externals` ，将公共依赖指定为外部依赖，不打包进子项目的代码中。
2. 子项目之间的依赖复用可以通过保证依赖的 URL 一致来实现。如果多个子项目都使用同一份 CDN 文件，加载时会先从缓存读取，避免重复加载。
3. 子项目复用主项目的依赖可以通过给子项目的 `index.html` 中的公共依赖的 `script` 和 `link` 标签添加自定义属性 `ignore` 来实现。在 `qiankun` 运行子项目时， `qiankun` 会忽略这些带有 `ignore` 属性的依赖，子项目独立运行时仍然可以加载这些依赖。
4. 在使用 `qiankun` 微前端框架时，可能会出现子项目之间和主项目之间的全局变量冲突的问题。这是因为子项目不配置 `externals` 时，子项目的全局 `Vue` 变量不属于 `window` 对象，而 `qiankun` 在运行子项目时会先找子项目的 `window` ，再找父项目的 `window` ，导致全局变量冲突。
5. 解决全局变量冲突的方案有三种：
    - 方案一是在注册子项目时，在 `beforeLoad` 钩子函数中处理全局变量，将子项目的全局 `Vue` 变量进行替换，以解决子项目独立运行时的全局变量冲突问题。
    - 方案二是通过主项目将依赖通过 `props` 传递给子项目，子项目在独立运行时使用传递过来的依赖，避免与主项目的全局变量冲突。
    - 方案三是修改主项目和子项目的依赖名称，使它们不会相互冲突，从而避免全局变量冲突的问题。

说说 webpack5 联邦模块在微前端的应用

> Webpack 5 的联邦模块（Federation Module）是一个功能强大的特性，可以在微前端应用中实现模块共享和动态加载，从而提供更好的代码复用和可扩展性

1. 模块共享

Webpack 5 的联邦模块允许不同的微前端应用之间共享模块，避免重复加载和代码冗余。通过联邦模块，我们可以将一些公共的模块抽离成一个独立的模块，并在各个微前端应用中进行引用。这样可以节省资源，并提高应用的加载速度。

TypeScript

```
// main-app webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin');
const { ModuleFederationPlugin } = require('webpack').container;

module.exports = {
  // ...其他配置

  plugins: [
    new HtmlWebpackPlugin(),
    new ModuleFederationPlugin({
      name: 'main_app',
      remotes: {
        shared_module: 'shared_module@http://localhost:8081/remoteEntry.js',
      },
    }),
  ],
};

// shared-module webpack.config.js
const { ModuleFederationPlugin } = require('webpack').container;

module.exports = {
  // ...其他配置

  plugins: [
    new ModuleFederationPlugin({
      name: 'shared_module',
      filename: 'remoteEntry.js',
      exposes: {
        './Button': './src/components/Button',
      },
    }),
  ],
};
```

在上述示例中，`main-app`和`shared-module`分别是两个微前端应用的 webpack 配置文件。通过`ModuleFederationPlugin`插件，`shared-module`将`Button`组件暴露给其他应用使用，而`main-app`则通过`remotes`配置引入了`shared-module`。

2. 动态加载

Webpack 5 联邦模块还支持动态加载模块，这对于微前端应用的按需加载和性能优化非常有用。通过动态加载，可以在需要时动态地加载远程模块，而不是在应用初始化时一次性加载所有模块。

TypeScript

```
// main-app
const remoteModule = () => import('shared_module/Button');

// ...其他代码

// 在需要的时候动态加载模块
remoteModule().then((module) => {
  // 使用加载的模块
  const Button = module.default;
  // ...
});
```

在上述示例中，`main-app`使用`import()`函数动态加载`shared_module`中的`Button`组件。通过动态加载，可以在需要时异步地加载远程模块，并在加载完成后使用模块。

在微前端应用中可以实现模块共享和动态加载，提供了更好的代码复用和可扩展性。通过模块共享，可以避免重复加载和代码冗余，而动态加载则可以按需加载模块，提高应用的性能和用户体验。

说说 qiankun 的资源加载机制（import-html-entry）

> `qiankun import-html-entry`是 qiankun 框架中用于加载子应用的 HTML 入口文件的工具函数。它提供了一种方便的方式来动态加载和解析子应用的 HTML 入口文件，并返回一个可以加载子应用的 JavaScript 模块。

具体而言，`import-html-entry`实现了以下功能：

1. 加载 HTML 入口文件：`import-html-entry` 会通过创建一个 `<link>` 标签来加载子应用的 HTML 入口文件。这样可以确保子应用的资源得到正确加载，并在加载完成后进行处理。
2. 解析 HTML 入口文件：一旦 HTML 入口文件加载完成，`import-html-entry` 将解析该文件的内容，提取出子应用的 JavaScript 和 CSS 资源的 URL。
3. 动态加载 JavaScript 和 CSS 资源：`import-html-entry` 使用动态创建 `<script>` 和 `<link>` 标签的方式，按照正确的顺序加载子应用的 JavaScript 和 CSS 资源。
4. 创建沙箱环境：在加载子应用的 JavaScript 资源时，`import-html-entry` 会创建一个沙箱环境（sandbox），用于隔离子应用的全局变量和运行环境，防止子应用之间的冲突和污染。
5. 返回子应用的入口模块：最后，`import-html-entry` 返回一个可以加载子应用的 JavaScript 模块。这个模块通常是一个包含子应用初始化代码的函数，可以在主应用中调用以加载和启动子应用。

通过使用`qiankun import-html-entry`，开发者可以方便地将子应用的 HTML 入口文件作为模块加载，并获得一个可以加载和启动子应用的函数，简化了子应用的加载和集成过程。

说说现有的几种微前端框架，它们的优缺点？

以下是对各个微前端框架优缺点的总结：

1. **qiankun 方案 优点**
   - 降低了应用改造的成本，通过 html entry 的方式引入子应用；
   - 提供了完备的沙箱方案，包括 js 沙箱和 css 沙箱；
   - 支持静态资源预加载能力。
2. **缺点**

- 适配成本较高，包括工程化、生命周期、静态资源路径、路由等方面的适配；
- css 沙箱的严格隔离可能引发问题，js 沙箱在某些场景下执行性能下降；
- 无法同时激活多个子应用，不支持子应用保活；
- 不支持 vite 等 esmodule 脚本运行。

3. **micro-app 方案 优点**
   - 使用 webcomponent 加载子应用，更优雅；
   - 复用经过大量项目验证过的 qiankun 沙箱机制，提高了框架的可靠性；
   - 支持子应用保活；
   - 降低了子应用改造的成本，提供了静态资源预加载能力。
4. **缺点**
   - 接入成本虽然降低，但路由依然存在依赖；
   - 多应用激活后无法保持各子应用的路由状态，刷新后全部丢失；
   - css 沙箱无法完全隔离，js 沙箱做全局变量查找缓存，性能有所优化；
   - 支持 vite 运行，但必须使用 plugin 改造子应用，且 js 代码没办法做沙箱隔离；
   - 对于不支持 webcomponent 的浏览器没有做降级处理。
5. **EMP 方案 优点**
   - webpack 联邦编译可以保证所有子应用依赖解耦；
   - 支持应用间去中心化的调用、共享模块；
   - 支持模块远程 ts 支持。
6. **缺点**
   - 对 webpack 强依赖，对于老旧项目不友好；
   - 没有有效的 css 沙箱和 js 沙箱，需要靠用户自觉；
   - 子应用保活、多应用激活无法实现；
   - 主、子应用的路由可能发生冲突。
7. **无界方案 优点**
   - 基于 webcomponent 容器和 iframe 沙箱，充分解决了适配成本、样式隔离、运行性能、页面白屏、子应用通信、子应用保活、多应用激活、vite 框架支持、应用共享等问题。
8. **缺点**
   - 在继承了 iframe 优点的同时，缺点依旧还是存在