Module Federation 详解

1.1 理解 Module Federation 的动机

在我们日常的前端开发工作中，“打包（Bundle）”是一个无法绕开的环节。无论是使用 Webpack、Vite 还是 Rollup，我们都习惯于将散落的模块、组件和第三方库，通过构建工具打包成浏览器可直接执行的静态文件。这个过程早已成为前端工程化的标准流程。

然而，随着应用规模的扩大和微前端架构的兴起，这种传统的、边界清晰的打包模式开始暴露出一些固有的痛点。今天，我们就从打包和依赖的视角出发，聊一聊为什么需要 Module Federation，它究竟想要解决什么问题。

回顾传统：Webpack 的打包边界

在理解新问题之前，我们先快速回顾一下传统打包工具（以 Webpack 为例）的核心工作模式。

当我们执行构建命令时，Webpack 会从一个或多个入口文件（Entry）出发，分析代码中的 import 和 require 语句，形成一个项目内所有模块的**依赖关系图**。

随后，它会遍历这个图，将所有涉及到的 JavaScript 模块、CSS 文件、图片资源等，根据配置转换并打包成一个或多个 bundle 文件。

这个过程的本质，是将一个“项目”的所有依赖，在**构建时**就组织好，打包成一个自给自足、边界清晰的整体。发布到服务器上的，就是这个构建完成的产物。在它内部，模块间的引用关系已经确定；对于它外部，则一无所知。

当项目变得复杂：我们遇到了什么问题？

起初，这种模式运行得非常好。但当我们的业务扩展，开始维护多个独立的应用或项目时，一些棘手的问题便浮出水面。

公共组件与代码的共享难题

1. 我们常常遇到这样的场景：公司内部有多个前端项目（例如，商品管理后台、营销活动后台），它们都需要使用同一套 UI 组件库，或者共享一些通用的业务模块（如登录、权限控制等）。
2. 传统的做法通常是将其发布为 npm 包。但这引入了新的管理成本：

- **更新成本高**：一个微小的改动（比如修复一个组件的样式），就需要更新 npm 包版本，所有依赖它的项目都必须重新安装依赖、构建、部署上线。整个流程相当繁琐。
- **版本管理复杂**：不同项目可能依赖了公共组件的不同版本，导致用户体验不一致，也增加了维护的心智负担。

重复打包与资源浪费

这是最直观的问题。如果多个独立项目都依赖了同一个大型库（如 React、Vue、ECharts 等），那么按照传统打包方式，每个项目都会将这个库完整地打包进自己的 bundle 中。

部署解耦困难

因为所有依赖都在构建时绑定，一个大型应用的不同部分（例如，一个后台系统的“报表页面”和“设置页面”）如果由不同团队开发，只要它们属于同一个项目，其中一个团队的发布就可能需要整个项目进行回归测试和重新部署，团队之间产生了不必要的耦合。

一个直观的例子：被重复打包的 React

为了更清晰地理解“重复打包”的问题，我们来看一个具体的场景。

假设我们有两个完全独立部署的 React 项目：

- **应用 A**：公司的官网（app-official-site）
- **应用 B**：用户个人中心（app-user-dashboard）

这两个项目都使用 create-react-app 创建，并各自独立开发、构建和部署。

当 app-official-site 执行构建命令后，会在其 build/static/js/ 目录下生成一个包含业务代码和 **React、ReactDOM** 完整源码的 main.[hash].js 文件。这个文件可能大小为 500KB，其中 React 相关的库就占了相当一部分体积。

同样地，app-user-dashboard 构建后，也会生成一个独立的、同样包含了 **React、ReactDOM** 的 bundle 文件。

现在，设想一个用户的访问路径：他首先访问官网（应用 A），浏览器下载了应用 A 的 bundle，其中包括了 React。然后，他点击登录，跳转到个人中心（应用 B）。此时，浏览器必须**再次下载**应用 B 的 bundle，这里面又包含了一份几乎一模一样的 React。

用户的网络带宽被浪费了，应用的加载性能也受到了影响。

一个自然而然的思考：如何共享依赖？

面对上述场景，我们很自然地会产生一个疑问：

有没有办法让这两个独立部署的应用，在运行时共享同一个 React 副本呢？

也就是说，当用户访问完应用 A 后，浏览器已经缓存了 React。当他再访问应用 B 时，应用 B 能否直接利用这个缓存，而无需重新下载？

这种“在运行时动态共享模块”的想法，正是驱动 Module Federation 诞生的核心动机。

它希望打破传统打包的“项目边界”，让一个应用可以直接引用另一个独立部署应用的模块，就像引用项目内部的本地模块一样自然，并且能智能地处理共享依赖，避免重复加载。

总结：从“构建时”到“运行时”

综上所述，我们可以看到，传统打包方案的核心是将所有依赖关系在**构建时**固化。这种模式在单一应用中非常高效，但在多应用、微前端的复杂场景下，却带来了代码共享、重复打包和部署耦合等一系列问题。

Module Federation 的出现，旨在将一部分依赖关系的管理，从“构建时”推迟到“**运行时**”。它提供了一种优雅的方案，使得不同的应用之间可以动态地、高效地共享模块和依赖库。

理解了“为什么需要它”，我们就能更好地把握 Module Federation 的设计哲学。在后续的文章中，我们将进一步探讨，它是如何通过 exposes 和 remotes 等概念，将这一愿景变为现实的。

1.2 MF的原理： Dynamic Import 与远程加载

在前端开发中，模块化是管理代码复杂度的关键。然而，传统的静态 `import` 语句会将所有模块在应用初始化时一次性加载进来。当项目规模越来越大，这种方式可能会导致首屏加载时间过长，影响用户体验。为了解决这个问题，我们需要一种更灵活的加载方式——动态加载。

本文将探讨两种核心的动态加载技术：Webpack 支持的 `import()` 语法和传统的远程脚本加载。理解它们的工作原理，是深入学习 Module Federation 等高级技术的重要基石。

Webpack 的 `import()`：实现模块的按需加载

Webpack 为我们提供了一个强大的工具—— `import()` 语法。与静态的 `import` 不同，`import()` 函数会返回一个 Promise，允许我们在代码的任何位置异步加载模块。当 Webpack 在编译时遇到 `import()` 语法，它会自动进行代码分割（Code Splitting），将被动态导入的模块打包成一个独立的 chunk 文件（bundle）。

这个独立的 bundle 不会包含在主入口文件中，只有当 `import()` 语句被执行时，浏览器才会发起网络请求去获取它。

这种机制带来了两个显而易见的好处：

1. **懒加载（Lazy Loading）**：我们可以将一些非首屏必需的组件或库设置为懒加载。例如，一个只有在用户点击特定按钮后才会弹出的对话框组件，就没有必要在页面初始化时加载。通过动态 `import()`，我们可以将其加载时机延迟到用户实际触发相应操作时。
2. **按需加载（On-Demand Loading）**：在单页面应用（SPA）中，路由切换是一个典型的按需加载场景。每个页面或功能区可以被视为一个独立的模块。当用户访问特定路由时，我们才去加载对应的组件资源，而不是在应用启动时就加载所有页面的代码。

示例：在 React 中实现路由组件的按需加载

在 React 生态中，结合 `React.lazy` 和 `Suspense`，我们可以非常优雅地实现组件的按需加载。

假设我们有一个 `AboutPage` 组件，希望在用户访问 `/about` 路由时才加载它。我们可以这样实现：

代码块

```
import React, { Suspense } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

// 使用 React.lazy 动态导入 AboutPage 组件
const AboutPage = React.lazy(() => import('./pages/AboutPage'));

const App = () => (
  <Router>
    {/* 使用 Suspense 组件包裹动态加载的组件，并提供 fallback UI */}
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route path="/about" component={AboutPage} />
        {/* 其他路由 */}
      </Switch>
    </Suspense>
  </Router>
);
```

通过这种方式，`AboutPage` 组件及其依赖的代码会被打包成一个独立的 chunk。只有当应用的路由匹配到 `/about` 时，浏览器才会去请求这个 chunk，从而实现了真正的按需加载，优化了应用的首屏性能。

传统远程脚本加载：`<script>` 标签的动态插入

在 `import()` 成为标准之前，开发者长期以来依赖一种更“手动”的方式来加载外部脚本：通过 JavaScript 动态创建一个 `<script>` 标签，并将其插入到 DOM 中。

这种方法的本质很简单：利用浏览器加载并执行 `<script>` 标签 `src` 属性指向的脚本文件的能力。

示例：手动实现一个 `loadScript` 函数

我们可以封装一个简单的 `loadScript` 函数，来模拟这个过程。这个函数接受一个脚本 URL，并在加载成功或失败时提供回调。

代码块

```
function loadScript(url) {
  return new Promise((resolve, reject) => {
    const script = document.createElement('script');
    script.src = url;
    script.async = true;

    script.onload = () => {
      // 脚本加载并执行成功
      resolve();
      // 清理工作，防止内存泄漏
      document.head.removeChild(script);
    };

    script.onerror = () => {
      // 脚本加载失败
      reject(new Error(`Failed to load script at ${url}`));
      document.head.removeChild(script);
    };

    document.head.appendChild(script);
  });
}
```

我们可以像下面这样使用它来加载一个外部模块，例如从 CDN 加载 `lodash`：

代码块

```
loadScript('https://cdn.jsdelivr.net/npm/lodash@4.17.21/lodash.min.js')
  .then(() => {
    // 此时 lodash 已经加载到全局作用域（通常是 window._）
    console.log(window._.defaults({ 'a': 1 }, { 'b': 2 }));
    // -> { 'a': 1, 'b': 2 }
```

```
  })
  .catch(error => {
    console.error(error);
  });
```

这种方式虽然看起来有些“原始”，但它揭示了远程模块加载的核心原理：**通过网络请求获取代码文本，然后通过某种方式（如 `eval` 或 script 标签）执行它，从而将其内容注入到当前应用的运行时环境中。**

小结：动态加载是走向联邦模块的基石

无论是 Webpack 提供的 `import()`，还是手动的 `<script>` 标签注入，它们都实现了一个共同的目标：将模块的加载时机从“编译时静态链接”转变为“运行时动态获取”。

这种从静态到动态的转变，是现代前端架构演进的一个重要方向。它不仅是实现代码分割和性能优化的基础，更为 Module Federation（模块联邦）这样的高级架构铺平了道路。Module Federation 正是建立在强大的运行时动态加载能力之上，才得以实现跨应用共享模块的愿景。

理解了动态加载的本质，我们就能更好地把握 Module Federation 的核心思想，并为探索更复杂的分布式前端系统打下坚实的基础。

1.3 MF的基础配置

Host 与 Remote：联邦的两种角色

在 Module Federation 的世界里，有两个核心角色：

- **Host（宿主）**：消费（加载）其他应用模块的那个应用。
- **Remote（远程）**：暴露（提供）模块给其他应用消费的那个应用。

一个应用可以同时是 Host 和 Remote，即所谓的“双向宿主”（Bi-directional Host）。这意味着它既可以消费来自其他应用的模块，也可以将自己的模块暴露出去。这种设计让应用间的协作变得极其

灵活。

最小化配置：ModuleFederationPlugin

要启用模块联邦，我们需要在 Webpack 配置中加入 `ModuleFederationPlugin`。下面我们以两个 React 项目 `app1` (Host) 和 `app2` (Remote) 为例，看看最精简的配置是怎样的。

app2 (Remote) 的配置

🌰 作为 Remote，`app2` 的任务是暴露一个组件（例如 `Button`）给 `app1` 使用。

代码块

```
// app2/webpack.config.js
const { ModuleFederationPlugin } = require("webpack").container;

module.exports = {
  // ... 其他配置
  plugins: [
    new ModuleFederationPlugin({
      name: "app2",
      filename: "remoteEntry.js",
      exposes: {
        "./Button": "./src/Button",
      },
      shared: {
        react: { singleton: true, requiredVersion: "^18.2.0" },
        "react-dom": { singleton: true, requiredVersion: "^18.2.0" },
      },
    }),
  ],
};
```

- **`name`**：`app2`，这是该 Remote 的唯一标识，在 Host 中会用到。
- **`filename`**：`remoteEntry.js`，这是 `app2` 打包后生成的入口文件，它包含了所有暴露模块的元信息和加载逻辑。Host 将通过访问这个文件来了解 `app2` 提供了哪些模块。

- **`exposes`**：`{"./Button": "./src/Button"}`，这里定义了要暴露的模块。`./Button` 是模块的别名，Host 将通过 `app2/Button` 的路径来引用它。`./src/Button` 是该模块在 `app2` 项目中的实际路径。
- **`shared`**：定义了共享的依赖库。我们稍后会详细解释。

app1 (Host) 的配置

🎨 作为 Host，`app1` 需要知道从哪里加载 `app2` 的模块。

代码块

```
// app1/webpack.config.js
const { ModuleFederationPlugin } = require("webpack").container;

module.exports = {
  // ... 其他配置
  plugins: [
    new ModuleFederationPlugin({
      name: "app1",
      remotes: {
        app2: "app2@http://localhost:3002/remoteEntry.js",
      },
      shared: {
        react: { singleton: true, requiredVersion: "^18.2.0" },
        "react-dom": { singleton: true, requiredVersion: "^18.2.0" },
      },
    }),
  ],
};
```

- **`name`**：`app1`，Host 的名称。
- **`remotes`**：`{"app2": "app2@http://localhost:3002/remoteEntry.js"}`，这是配置的核心。它告诉 `app1`：
  - 有一个名为 `app2` 的 Remote。
  - 这个 Remote 的 `name` 是 `app2`（`@` 符号前）。

- 它的入口文件地址是 `http://localhost:3002/remoteEntry.js`（`@` 符号后）。

动态加载 Remote 组件

配置完成后，`app1` 就可以像加载本地模块一样，通过 `React.lazy` 动态引入 `app2` 的 `Button` 组件。

代码块

```
// app1/src/App.js
import React from "react";

const RemoteButton = React.lazy(() => import("app2/Button"));

const App = () => (
  <div>
    <h1>Basic Host-Remote</h1>
    <h2>App 1 (Host)</h2>
    <React.Suspense fallback="Loading Button...">
      <RemoteButton />
    </React.Suspense>
  </div>
);

export default App;
```

当 `app1` 渲染到 `RemoteButton` 时，Webpack 会在运行时自动处理：

1. 根据 `remotes` 配置找到 `app2` 的 `remoteEntry.js` 地址。
2. 加载并执行 `remoteEntry.js` 文件。
3. 通过 `remoteEntry.js` 提供的接口，找到并加载 `Button` 组件的代码。

整个过程对开发者是透明的，实现了跨应用的无缝组件复用。

1.4 shared 配置与依赖共享

在微前端架构中，如何优雅地管理各个独立应用之间的公共依赖，是一个无法回避的核心问题。如果我们处理不当，每个应用都打包一份自己的 react、react-dom 或 vue，不仅会造成网络资源的极大浪费，更可能因为多实例共存而引发难以预料的运行时错误。

Module Federation 提供的 shared 配置，正是为了解决这一挑战而设计的。它不仅仅是一个简单的打包配置，更是一套精巧的运行时依赖协商与共享机制。理解它的工作原理，是真正掌握 Module Federation 精髓的关键。

接下来，我们将深入探讨 shared 的声明方式、关键参数、运行时版本匹配逻辑，以及它相比传统方案的优势所在。

shared 的作用与声明方式

shared 的核心目标，是让多个独立的应用在运行时尽可能地复用同一个依赖实例，同时又保留各自在开发时的版本灵活性。

最简单的声明方式是使用一个数组：

代码块

```
// webpack.config.js
plugins: [
  new ModuleFederationPlugin({
    // ...
    shared: ['react', 'react-dom'],
  }),
],
```

这种写法是一个便捷的缩写，它告诉 Webpack：“请尝试共享 react 和 react-dom 这两个库”。Webpack 会自动从 package.json 中读取它们的版本信息，并应用一套默认的共享策略。

然而，在真实项目中，我们往往需要更精细的控制：

代码块

```
// webpack.config.js
plugins: [
  new ModuleFederationPlugin({
    // ...
    shared: {
      'react': {
        singleton: true,
        requiredVersion: '^18.2.0',
        strictVersion: false,
        eager: false,
      },
      'react-dom': {
```

```
          singleton: true,
          requiredVersion: '^18.2.0',
        },
      },
    }),
  ],
```

我们可以为每个共享的依赖包配置几个关键参数：

`singleton`：这是 shared 配置中最重要的参数之一。当设置为 true 时，它**强制要求共享的这个依赖在整个应用生态中只能存在唯一一个实例**。这对于像 react 这样含有内部状态、无法容忍多实例共存的库来说，是必须开启的。

`requiredVersion`：用于指定该依赖可接受的版本范围，其语法遵循 `semver` 规范。例如，`^18.2.0` 表示所有 18.x.x 系列的兼容版本都可以接受。Module Federation 会利用这个范围来进行运行时的版本协商。如果不指定，则默认使用当前项目 package.json 中的版本。

`strictVersion`：默认是 false。如果设置为 true，则会严格匹配 requiredVersion 中定义的版本号，不再遵循 semver 的兼容规则。这通常用于需要强制统一版本的场景，但会牺牲一定的灵活性。

`eager`：默认是 false。在默认情况下，共享的依赖是异步加载的（lazy-loaded）。当设置为 true 时，这个共享依赖会被打包到初始的 entry chunk 中，与应用代码一同被立即加载。这会增加初始包体积，但可以避免后续的异步加载请求。通常，只有那些应用启动时就必须用到的核心库，我们才考虑开启 eager。

运行时版本匹配与冲突处理

shared 机制的精髓在于其**运行时动态协商**的能力。当 `Host` 应用（宿主）加载一个 `Remote` 应用（远程模块）时，它们之间会就共享依赖进行一次“对话”，决策过程大致如下：

我们可以将这个过程想象成一个决策时序图：

1. **Host 初始化**：Host 应用启动时，会初始化一个“共享作用域 `(Shared Scope)`”。它会把自己 shared 配置中声明的依赖及其版本注册到这个作用域中，准备好随时分享给其他应用。
2. **Remote 加载**：当 Host 代码 `import('remote/Button')` 时，Remote 应用被加载。
3. **版本协商**：Remote 在真正执行自己的代码前，会先检查自己的 shared 依赖。对于每一个依赖（例如 react），它会：
    - 查看 Host 的共享作用域中是否已经存在一个名为 react 的依赖。
    - 如果存在，它会用自己 requiredVersion 定义的版本范围，去匹配 Host 提供的 react 版本。
4. **决策结果**：
    - **匹配成功 (Ideal Case)**：如果 Host 提供的版本在 Remote 的可接受范围内，Remote 会放弃加载自己的 react 副本，直接使用 Host 提供的实例。这是最理想的共享状态，实现了真正的单例。

- **匹配失败 (Fallback)**：如果 Host 没有提供该依赖，或者提供的版本不符合 Remote 的要求，Remote 会加载自己打包的 react 副本作为兜底 (fallback)。
- **冲突与警告 (Conflict)**：当 `singleton: true` 被开启，但版本又不兼容时，情况会变得特殊。为了避免应用因多实例而崩溃，Module Federation 通常会选择使用其中一个版本（往往是 Host 的），但同时会在浏览器控制台打印一条醒目的警告信息。这条警告是重要的开发信号，提示我们需要统一相关依赖的版本。

通过几个具体的场景，我们可以更直观地理解这个过程：

场景一：未配置 shared

- 如果 Host 和 Remote 都没有配置 shared: ['react']，那么它们会各自为政，在自己的包里都打入一份完整的 React。结果就是应用被重复加载，体积臃肿，并可能因双实例导致 Invalid hook call 等错误。

场景二：正确配置 shared

- Host 和 Remote 都正确配置了 shared，且依赖的 React 版本（例如 ^18.2.0）互相兼容。当 Remote 加载时，它发现 Host 已经提供了一个符合要求的 React 实例，于是愉快地复用了它。应用中自始至终只有一个 React 实例在工作。

场景三：版本不一致

- Host 使用 React 18.2.0，而 Remote 配置了 requiredVersion: '17.0.2'。当 Remote 加载时，版本协商失败。由于 singleton: true 的存在，Module Federation 会强制使用其中一个版本（**以 host 的版本为准**。Host 是运行环境的控制方），并抛出版本不匹配的警告，提醒开发者潜在的风险。

shared 为何比 CDN 或 externals 更灵活?

在 `Module Federation` 出现之前，我们通常用 `externals` 或 CDN 来处理公共依赖。这些方案虽然能解决问题，但灵活性远远不足。

externals + UMD 方案：这种方式将依赖的控制权完全交给了外部环境。所有微应用都必须依赖于一个通过 `<script>` 标签全局注入的库，版本被死死地钉在 index.html 中。任何一个应用想升级依赖版本，都可能需要协调所有团队同步修改，牵一发而动全身。

CDN 方案：本质上与 externals 类似，只是依赖的托管方变成了 CDN 服务。它同样面临版本僵化和“全体升级或全体不动”的困境。

相比之下，shared 机制的优势在于其**去中心化的、基于** semver **的自动化协商能力**。

它允许每个应用在自己的 package.json 中独立管理依赖版本，只要版本在兼容范围内，运行时就能自动寻找到最佳的共享方案。即使版本不完全兼容，它也提供了清晰的 fallback 机制和警告信息，而不是直接导致应用崩溃。这种设计，在保证了应用性能的同时，最大程度地保留了各个开发团队的自主权和灵活性。

shared 与 bundle splitting 的关系

最后，有必要厘清 shared 和传统 bundle splitting（如 Webpack 的 SplitChunksPlugin）的区别。

bundle splitting 是**单个应用内部**的优化策略。它的目标是将一个庞大的应用代码库拆分成多个小的 chunk，以便按需加载，优化首屏性能。它的作用域是应用之内。

shared 是**多个独立应用之间**的依赖共享策略。它的目标是让这些本无关联的应用在组合到一起时，能够复用公共资源。它的作用域是跨应用的。

两者虽然都涉及代码拆分和复用，但解决的是不同维度的问题。在微前端架构中，我们通常会将两者结合使用：在每个独立应用内部使用 bundle splitting 进行优化，同时通过 Module Federation 的 shared 机制处理跨应用间的依赖共享。

1.5 Qiankun/Single-SPA 与 Webpack Module Federation

在构建大型前端应用时，微前端架构已经从一个前沿概念，演变为许多团队技术选型中的重要一环。它将庞大、单一的应用拆分为多个更小、更独立的部分，使得不同团队可以并行开发、独立部署，从而提高了整体的灵活性和可维护性。

在实现微前端的众多方案中，基于 `single-spa` 的 `qiankun` 和 Webpack 5 之后推出的 `Module Federation` (MF) 是两种主流路径。它们虽然都致力于解决同样的问题，但其背后的设计哲学和实现方式却大相径庭。

本篇将深入探讨这两种方案的核心差异，并分析为什么 Module Federation 常被认为是更“原生”的微前端解决方案。

容器式微前端：Qiankun 与 Single-SPA

Qiankun 是在 single-spa 基础之上，提供了更完善开箱即用能力的微前端框架。我们可以将这类方案理解为“应用容器”模型。

其核心思路是：存在一个主应用（基座），它作为整个系统的底盘，负责承载和调度其他独立的微应用。

工作机制

在这种模型下，主应用和微应用的关系更像是“房东”与“租客”：

1. **注册与加载**：主应用会注册一系列微应用，并定义它们的激活规则（通常与路由绑定）。当用户访问特定路径时，主应用会匹配规则，然后动态加载对应微应用的资源（JS 和 CSS 文件）。
2. **沙箱隔离**：为了避免不同微应用之间的全局变量污染、事件冲突和样式覆盖，qiankun 引入了沙箱机制。
    - **JS 沙箱**：通过 Proxy 等技术，为每个微应用创建一个隔离的 JavaScript 运行环境。当微应用操作 window 对象时，实际上是在自己的“私有领域”内活动，不会影响到其他应用。
    - **CSS 隔离**：通过 Shadow DOM 或为样式表动态添加作用域的方式，确保微应用的样式不会“泄露”到全局，反之亦然。

3. **生命周期管理**：主应用会接管微应用的整个生命周期，包括初始化 (bootstrap)、挂载 (mount) 和卸载 (unmount)，确保它们在合适的时机被正确地渲染到页面指定容器中。

简单来说，容器式方案的核心在于“隔离”。它将每个微应用视为一个完整的、有边界的独立个体，主应用负责管理这些个体，但通常不深入其内部实现。这种方式很像是在一个页面里运行了多个“没有 iframe 边框的 iframe”。

编译时联邦：Webpack Module Federation

Module Federation (MF) 则提供了一种完全不同的视角。它并非一个应用框架，而是 Webpack 内置的一个功能，旨在让不同的 Webpack 构建（即不同的应用）可以在运行时共享模块（Module）。

我们可以将其理解为“模块联邦”模型。

工作机制

如果说 qiankun 是在应用层面进行集成，那么 MF 则下沉到了更细的模块层面：

1. **模块暴露与消费**：在 MF 的世界里，每个应用既可以是模块的提供方 (Remote)，也可以是消费方 (Host)。
    - **Remote**：通过 Webpack 配置，将应用内部的特定模块（如组件、函数）暴露出去。Webpack 会为此生成一个清单文件（通常是 remoteEntry.js），其中记录了所有可供外部使用的模块及其访问路径。
    - **Host**：在需要时，通过 import() 动态地从 Remote 应用中引入模块。这个过程对于开发者来说，与普通的异步代码分割非常相似。
2. **依赖共享**：MF 最具吸引力的特性之一是其原生的依赖共享机制。我们可以在配置中明确指定哪些依赖库（如 react, vue）是共享的。当 Host 和 Remote 都依赖同一个库的兼容版本时，Webpack 可以确保这个库在整个应用生态中只被加载一次。
3. **动态加载**：当 Host 应用尝试加载一个来自 Remote 的模块时，它会先请求 remoteEntry.js 清单，然后根据清单的指引，按需加载真正包含该模块代码的代码块 (Chunk)。

MF 的关注点在于“共享”与“连接”。它打破了应用之间的墙壁，让模块可以跨越构建边界自由流通，形成一个更紧密、更一体化的联邦。

为什么 Module Federation 更“原生”？

当我们说 MF 更“原生”时，我们指的是它更贴近现代 JavaScript 的模块化本质和构建生态。

1. **粒度：从“应用级”到“模块级”**

这是两者最根本的区别。Qiankun 操作的最小单位是“应用”，它加载的是一个完整的应用包，并通过沙箱进行隔离。而 MF 操作的最小单位是“模块”，它可以是任何可被 import 的 JavaScript 单元，比如一个组件、一个工具函数。

现代前端开发本身就是围绕模块构建的。我们通过 import 和 export 组织代码，构建工具则将这些模块打包成高效的静态资源。MF 恰好是这一思想的自然延伸，它让 import 能够跨越应用的边界，直接

将远程模块无缝集成到当前应用的依赖图中。这种体验对于开发者而言，无疑是更自然、更符合直觉的。

2. **依赖管理：从“外部协调”到“内建机制”**

在 qiankun 方案中，处理公共依赖是一个需要审慎规划的问题。我们通常需要借助 Webpack Externals、import-maps 或其他约定来确保多个微应用共用同一份依赖库，以避免冗余加载和版本冲突。这增加了配置的复杂性和维护成本。

相比之下，**依赖共享是 MF 的核心能力之一**。通过简单的配置，Webpack 就能在构建时分析出共享依赖，并在运行时智能地决策是否需要加载它。这种内建的、自动化的管理方式，显然比外部协调来得更优雅、更可靠。

3. **性能：从“宏观加载”到“精准加载”**

Qiankun 加载的是整个微应用的入口 JS。即使我们只需要其中的一个小组件，也必须将整个应用包下载并执行。

MF 则实现了真正的按需加载。由于它在模块层面工作，当 Host 需要 Remote 的某个模块时，只会去加载包含该模块的特定代码块，而不是整个 Remote 应用。这种精准打击式的加载方式，带来了更优的性能表现和更快的页面响应速度。

我们应该如何选择？

- **选择** Qiankun / Single-SPA：当你的目标是整合多个技术栈差异巨大、甚至包含陈旧遗留系统的应用时，qiankun 强大的隔离能力是巨大优势。它提供了一种非侵入式的整合方式，让你可以用较低的成本将“老死不相往来”的应用装进同一个屋檐下。
- **选择** Module Federation：如果你的团队正在构建一个全新的、基于 Webpack 生态（如 React、Vue 3）的复杂系统，MF 无疑是更现代、更高效的选择。它提供了无与伦比的开发体验和性能优势，让微前端的感觉不再是“拼凑”，而是真正的“联邦”。

总而言之，qiankun 和 Module Federation 并非简单的替代关系，而是针对不同场景、不同目标的两种解法。qiankun 胜在“隔离”与“兼容”，而 MF 则强在“共享”与“原生”。理解它们背后的设计哲学，将帮助我们在复杂的技术世界中，做出更明智的抉择。

1.6 Vite、Rspack 与 Next.js 中的联邦实现

自 Webpack 5 问世以来，模块联邦（Module Federation, MF）就一直是微前端领域备受瞩目的技术。它提供了一种在多个独立构建的应用之间动态共享代码的能力，优雅地解决了传统微前端方案中常见的依赖共享和版本控制难题。

然而，前端工具链的演进从未停歇。以原生 ESM 为基础的 Vite 迅速崛起，Rust 编写的 Rspack 带来了极致的性能想象，而 Next.js 则在全栈框架的道路上不断探索。

本篇我们将一起梳理模块联邦在不同工具链中的实现现状，并探讨 Webpack MF 本身的未来走向。

Webpack Module Federation

作为模块联邦的开创者，Webpack MF 的设计无疑是成功且影响深远的。它通过一种巧妙的运行时机制，让不同的 Webpack 构建产物可以像一个单体应用一样共享模块，同时保持各自的独立开发与部署。

在很多复杂的业务场景中，我们都能看到 Webpack MF 的身影：

- **设计系统（Design System）分发**：主应用消费由专门团队维护的组件库，组件库可独立更新。
- **中后台“巨石应用”拆分**：将庞大的后台系统按业务线拆分为多个子应用，共享通用的平台能力（如登录、权限）。
- **跨团队协作**：不同团队负责的应用可以相互消费对方的功能模块，而无需通过发布 npm 包的冗长流程。

然而，随着时间的推移和项目规模的扩张，Webpack MF 的一些固有局限也逐渐显现。

1. **构建性能**：这是 Webpack 生态的老问题。虽然有各种缓存和优化策略，但在一个由数十个联邦模块组成的大型系统中，本地开发启动和生产环境构建的耗时依然是绕不开的痛点。
2. **配置复杂性**：MF 的配置 (ModuleFederationPlugin) 虽然强大，但其参数如 shared, remotes, exposes 的精细化配置需要深入的理解和大量的样板代码。尤其是 shared 依赖的处理，为了追求最优的共享策略，往往需要复杂的配置，这构成了不低的上手门槛。
3. **生态绑定**：它的实现与 Webpack 的构建管线和运行时深度绑定。如果想在一个非 Webpack 的项目（例如 Vite）中消费一个 Webpack MF 模块，就需要引入额外的适配层，过程并不顺畅。

这些局限，为新一代构建工具的入局提供了契机。

Vite、Rspack 与 Next.js 中的联邦实现

模块联邦作为一种架构思想，其生命力正在 Webpack 之外的生态中延续。我们来看看它在几个主流工具中的现状。

1. Vite / Rollup 体系

Vite 在开发环境以其基于原生 ESM 的秒级冷启动速度著称，这与 Webpack MF 基于打包（bundle）的理念存在根本差异。因此，Vite 社区通过插件化的方式来拥抱联邦。

其中，@originjs/vite-plugin-federation 是目前最主流的方案。它的实现思路很巧妙：

开发环境：插件会启动一个服务器，将 exposes 的模块转换为符合 ESM 规范的导入，并处理好 shared 依赖。remote 应用在请求模块时，会被代理到对应的开发服务器上，实现了动态加载。

生产环境：在构建阶段，它会借助 Rollup 将模块打包成符合 MF 运行时规范的格式，生成 remoteEntry.js 文件，从而与 Webpack MF 的产物保持兼容。

现状：通过社区插件，Vite 已经具备了完整的模块联邦能力，且能与现有的 Webpack MF 系统进行混用。但它并非“一等公民”，其稳定性和对复杂场景的处理能力，仍需更多社区实践的检验。对于追

求极致开发体验的团队，这是一个值得尝试的方向。

2. Rspack

Rspack 的定位非常明确：一个由 Rust 编写、性能超群且与 Webpack 生态兼容的打包工具。对于模块联邦，Rspack 选择了**原生兼容**的路线。

这意味着，你几乎可以将 Webpack MF 的配置原封不动地迁移到 Rspack 的配置文件 (rspack.config.js) 中。Rspack 在底层用 Rust 重新实现了 Webpack MF 的核心逻辑。

现状：Rspack 的 MF 实现是目前最具吸引力的“平替”方案。它解决了 Webpack MF 最大的痛点——**性能**。对于那些深受大型联邦系统构建速度困扰的团队，迁移到 Rspack 几乎成了一个无需犹豫的选择。它继承了 Webpack MF 的稳定性和生态，同时带来了数倍乃至十数倍的性能提升。可以说，Rspack 让模块联邦这个优秀的架构思想，得以在性能上获得新生。

3. Next.js

在 Next.js 这样的全栈框架中实现模块联邦，复杂度远高于纯客户端应用。因为共享的模块可能同时需要在服务端（SSR/SSG）和客户端运行，这涉及到一系列复杂问题：

- **双端执行**：同一个联邦模块如何在 Node.js 环境和浏览器环境中正确加载和执行？
- **异步加载与 Hydration**：服务端渲染出的 HTML 如何与客户端动态加载的联邦模块正确地进行水合（Hydration），避免内容闪烁或交互失效？
- **运行时隔离**：服务端的多个请求之间，联邦模块的运行时状态如何隔离，避免相互污染？

针对这些挑战，`@module-federation/nextjs-mf` 插件提供了专门的解决方案。它通过更高阶的封装，处理了服务端模块的同步/异步加载、客户端脚本的自动注入等一系列脏活累活。

现状：Next.js 中的联邦方案功能强大，但也是配置和理解成本最高的。它不仅仅是前端模块的共享，更是“分布式 Next.js 应用”的雏形。官方（Vercel）目前并未内置该能力，而是依赖社区和核心贡献者的推动。对于希望在多个 Next.js 应用之间深度复用页面、组件乃至数据获取逻辑的场景，这是一个强有力的工具，但使用前需要团队具备对 Next.js 和 MF 都有深入的理解。

归根结底，模块联邦的核心价值在于“**在独立与集成之间取得平衡**”。无论底层工具如何变迁，这个核心思想都将持续演化，并以更多元、更高效的方式融入我们的日常开发。未来的前端架构，无疑会因为这种演进而变得更加灵活和富有弹性。