使用 WebAssembly 在浏览器中运行 R 语言

马浩琨

Aug 23, 2025

1 副标题：无需服务器，无需安装，点击即得的 R 语言数据分析体验是如何实现的?

作为 R 语言用户，我们常常面临环境配置的困扰。传统的工作流程需要安装 IDE 如 RStudio，配置复杂的依赖环境，并管理各种包。这不仅耗时，而且在协作和分享时带来巨大挑战。如何让没有安装 R 的同事或客户复现分析结果？这似乎是一个不可能的任务。但 WebAssembly（Wasm）的出现带来了转机。WebAssembly 是一种可以在现代 Web 浏览器中运行的高性能、低级别字节码格式。想象一下，将 C++ 或 R 代码编译成一种浏览器都能理解的「世界语」，从而打破环境壁垒。是的，通过 WebAssembly，我们可以将完整的 R 语言引擎移植到浏览器中，实现点击即得的体验。

2 核心技术解构：这一切是如何实现的?

Emscripten 是一个关键工具链，它允许将 C 和 C++ 代码编译为 WebAssembly。由于 R 语言的底层大量使用 C 和 Fortran 编写，Emscripten 能够处理这些代码，将其转换为 Wasm 模块。例如，R 的统计函数和线性代数运算都依赖于这些底层库，Emscripten 将它们「翻译」成浏览器可执行的格式。更宏大的工程是将整个 R 解释器、基础库和必要扩展包编译成 WebAssembly。社区项目如 `r-wasm` 或 `WebR` 推动了这一进程。以 `WebR` 为例，它积极维护，旨在提供完整的 R 环境。流程上，R 源代码通过 Emscripten 编译成 `R.wasm` 文件，并生成 JavaScript 胶水代码来处理交互。当用户访问网页时，浏览器下载 `R.wasm` 文件并通过 JavaScript WebAssembly API 实例化它。Emscripten 模拟了一个虚拟文件系统在浏览器内存中，用于存放 R 库、用户数据和安装的包。交互方式多样：可以通过 XTerm.js 终端模拟命令行，或通过 JavaScript 调用 R 函数。例如，用 JavaScript 将数据传入 R，执行模型，再获取结果。

3 实战演示：构建一个浏览器内的 R 应用

我们构建一个简单的线性回归分析与可视化应用。技术栈包括 `WebR` 提供 R 能力，HTML/CSS 用于布局，JavaScript 处理逻辑，以及 Chart.js 用于可视化。首先，在 HTML 中引入 `webR.js` 库。代码示例：

```
<script src="https://webr.r-wasm.org/latest/webr.js"></script>
```

这段代码加载了 `WebR` 的 JavaScript 库，它为浏览器中的 R 提供接口。库的 URL 指向最新版本，确保功能更新。然后，初始化 WebR：

```
const webR = new WebR();
await webR.init();
```

这里，我们创建了一个 WebR 实例并初始化它。await 关键字表示异步操作，等待初始化完成，这是因为 Wasm 模块的加载和实例化是异步过程，避免阻塞主线程。接下来，创建 UI 元素，如文件上传、代码输入、运行按钮和输出区域。在 JavaScript 中，我们可以添加事件监听器。例如，处理文件上传：

```
document.getElementById('uploadButton').addEventListener('click', async () => {
  const file = document.getElementById('fileInput').files[0];
  const text = await file.text();
  await webR.writeFile('data.csv', text);
});
```

这段代码监听按钮点击事件，读取用户上传的 CSV 文件，并使用 webR.writeFile 方法将其写入 WebR 的虚拟文件系统作为 'data.csv'。这模拟了 R 中的文件操作，但所有数据存储在浏览器内存中。然后，执行 R 代码进行回归分析：

```
const code = `
  data <- read.csv('data.csv')
  model <- lm(y ~ x, data=data)
  summary(model)
`;
const output = await webR.evalR(code);
console.log(output);
```

webR.evalR 方法执行 R 代码字符串，并返回输出。这里，我们读取数据，拟合线性模型 $y = \beta_0 + \beta_1 x + \epsilon$，并打印摘要。输出可以是文本或结构化数据，通过 JavaScript 处理。对于可视化，我们可以使用 R 的 plot 函数或集成 Chart.js。由于 R 图形需要额外处理，我们可以选择用 JavaScript 库直接渲染。例如，从 R 获取拟合值并用 Chart.js 创建图表：

```
const fittedValues = await webR.evalR('model$fitted.values');
// 假设 fittedValues 是数组，然后使用 Chart.js 渲染
```

这段代码通过 webR.evalR 获取模型拟合值，然后在 JavaScript 中传递给 Chart.js 进行可视化。整个应用在浏览器中运行，无需服务器支持。

4 优势与挑战：理性看待这项技术

WebAssembly 为 R 语言带来巨大优势。可移植性极高，真正实现「一次编写，随处运行」，因为所有内容在浏览器中执行，无需安装或部署。安全方面，代码在沙盒中运行，无法访问本地系统，保护用户隐私。分享和嵌入变得简单，可以轻松添加到博客或教程中。此外，客户端计算减轻服务器负担，所有处理在用户端完成。然而，挑战也存在。初始化性能可能较慢，因为需要下载几 MB 的 Wasm 文件。计算性能虽优于 JavaScript，但相

比原生 R 有损耗，尤其对于计算密集型任务。包兼容性不是完美的；一些依赖系统库的包可能无法使用。虚拟文件系统是易失的，刷新页面后数据丢失，需要重新加载。

5 未来展望

未来，我们可能看到与 Shiny 的深度融合，实现完全客户端的交互式应用，无需后端服务器。随着 Wasm 标准如垃圾回收（GC）的发展，加载速度和模块体积将优化。生态将更丰富，更多 R 包被移植，社区提供预编译包仓库。这将在隐私敏感领域如医疗或金融中发挥重要作用， enabling offline data analysis.
WebAssembly 正在改变 R 语言的范式，从依赖特定环境的桌面软件转向开放、共享的 Web 平台。鼓励读者尝试 `WebR`，亲身体验浏览器中运行 R 的魅力。更多资源可参考官方文档和 GitHub 仓库。

6 互动环节

讨论问题：您能想到哪些场景最适合使用浏览器内的 R？又有哪些场景目前还不适合？邀请行动：您是否会尝试在项目中使用 WebR？在评论区分享您的想法！