

Webassembly 学习手册（全）

原文：annas-

archive.org/md5/d5832e9a9d99a1607969f42f55873dd5

译者：飞龙

PDF 整理：hamburgerdog

协议：CC BY-NC-SA 4.0

[前言](#)

[本书适合对象](#)

[本书涵盖内容](#)

[充分利用本书](#)

[下载示例代码文件](#)

[下载彩色图片](#)

[使用的约定](#)

第一章：什么是 WebAssembly？

[通往 WebAssembly 的道路](#)

[JavaScript 的演变](#)

[谷歌和 Native Client](#)

[Mozilla 和 asm.js](#)

[WebAssembly 的诞生](#)

WebAssembly 到底是什么，我在哪里可以使用它？

[官方定义](#)

[二进制指令格式](#)

[可移植的编译目标](#)

[核心规范](#)

[语言概念](#)

[语义阶段](#)

[JavaScript 和 Web API](#)

那么它会取代 JavaScript 吗？

我可以在哪里使用它？

支持哪些语言？

[C 和 C++](#)

[Rust](#)

[其他语言](#)

[有哪些限制？](#)

[没有垃圾回收](#)

[没有直接的 DOM 访问](#)

[旧版浏览器不支持](#)

[它与 Emscripten 有什么关系？](#)

[Emscripten 的作用](#)

[EMSDK 和 Binaryen](#)

[总结](#)

[问题](#)

[进一步阅读](#)

[第二章：WebAssembly 的元素-Wat、Wasm 和 JavaScript API](#)

[共同结构和抽象语法](#)

[Wat](#)

[定义和 S 表达式](#)

[值、类型和指令](#)

[在开发过程中的作用](#)

[二进制格式和模块文件（Wasm）](#)

[定义和模块概述](#)

[模块部分](#)

[JavaScript 和 Web API](#)

[WebAssembly 存储和对象缓存](#)

[加载模块和 WebAssembly 命名空间方法](#)

[WebAssembly 对象](#)

[WebAssembly.Module](#)

[WebAssembly.Instance](#)

[WebAssembly.Memory](#)

[WebAssembly.Table](#)

[WebAssembly 错误（CompileError、LinkError、RuntimeError）](#)

[使用 WasmFiddle 连接各个部分](#)

[什么是 WasmFiddle？](#)

[C 代码转换为 Wat](#)

[Wasm 到 JavaScript](#)

[总结](#)

[问题](#)

[进一步阅读](#)

[第三章：设置开发环境](#)

[安装开发工具](#)

操作系统和硬件

[macOS](#)

[Ubuntu](#)

[Windows](#)

软件包管理器

[macOS 的 Homebrew](#)

[Ubuntu 的 Apt](#)

[Windows 的 Chocolatey](#)

Git

[在 macOS 上安装 Git](#)

[在 Ubuntu 上安装 Git](#)

[在 Windows 上安装 Git](#)

Node.js

[nvm](#)

[在 macOS 上安装 nvm](#)

[在 Ubuntu 上安装 nvm](#)

[在 Windows 上安装 nvm](#)

[使用 nvm 安装 Node.js](#)

GNU make 和 rimraf

[macOS 和 Ubuntu 上的 GNU Make](#)

[在 macOS 上安装 GNU Make](#)

[在 Ubuntu 上安装 GNU Make](#)

[在 Windows 上安装 GNU make](#)

安装 rimraf

VS Code

[在 macOS 上安装 Visual Studio Code](#)

[在 Ubuntu 上安装 Visual Studio Code](#)

[在 Windows 上安装 VS Code](#)

配置 VS Code

[管理设置和自定义](#)

[扩展概述](#)

[C/C++ 和 WebAssembly 的配置](#)

[为 VS Code 安装 C/C++](#)

[为 VS Code 配置 C/C++](#)

[VSCode 的 WebAssembly 工具包](#)

[其他有用的扩展](#)

[自动重命名标签](#)

[括号对颜色器](#)

[Material Icon 主题和 Atom One Light 主题](#)

[为 Web 设置](#)

[克隆书籍示例存储库](#)

[安装本地服务器](#)

[验证您的浏览器](#)

[验证 Google Chrome](#)

[验证 Mozilla Firefox](#)

[验证其他浏览器](#)

[其他工具](#)

[macOS 的 iTerm2](#)

[Ubuntu 的 Terminator](#)

[Windows 的 cmder](#)

[Zsh 和 Oh-My-Zsh](#)

[摘要](#)

[问题](#)

[进一步阅读](#)

[第四章：安装所需的依赖项](#)

[开发工作流程](#)

[工作流程中的步骤](#)

[将工具集成到工作流程中](#)

[Emscripten 和 EMSDK](#)

[Emscripten 概述](#)

[EMSDK 适用于哪里？](#)

[安装先决条件](#)

[常见的先决条件](#)

[在 macOS 上安装先决条件](#)

[在 Ubuntu 上安装先决条件](#)

[在 Windows 上安装先决条件](#)

[安装和配置 EMSDK](#)

[跨所有平台的安装过程](#)

[在 macOS 和 Ubuntu 上安装](#)

[在 Windows 上安装和配置](#)

[在 VS Code 中配置](#)

[测试编译器](#)

[C 代码](#)

[编译 C 代码](#)

[摘要](#)

[问题](#)

进一步阅读

第五章：创建和加载 WebAssembly 模块

使用 Emscripten 粘合代码编译 C

编写示例 C 代码

编译示例 C 代码

输出带有胶水代码的 HTML

输出没有 HTML 的胶水代码

加载 Emscripten 模块

预生成的加载代码

编写自定义加载代码

编译不带粘合代码的 C 代码

用于 WebAssembly 的 C 代码

在 VS Code 中使用构建任务进行编译

获取和实例化 Wasm 文件

常见的 JavaScript 加载代码

HTML 页面

提供所有服务

总结

问题

进一步阅读

第六章：与 JavaScript 交互和调试

Emscripten 模块与 WebAssembly 对象

什么是 Emscripten 模块？

胶水代码中的默认方法

WebAssembly 对象的差异

从 JavaScript 调用编译后的 C/C++ 函数

从 Module 调用函数

Module.ccall()

Module.cwrap()

C++ 和名称修饰

从 WebAssembly 实例调用函数

从 C/C++ 调用 JavaScript 函数

使用粘合代码与 JavaScript 交互

使用 `emscripten_run_script()` 执行字符串。

使用 `EM_ASM()` 执行内联 JavaScript()

重用内联 JavaScript 与 `EM_JS()`

使用粘合代码的示例

C 代码

[HTML 代码](#)

[编译和提供结果](#)

[无需胶水代码与 JavaScript 交互](#)

[使用导入对象将 JavaScript 传递给 C/C++](#)

[在 C/C++ 中调用导入的函数](#)

[一个没有胶水代码的例子](#)

[C++ 代码](#)

[HTML 代码](#)

[编译和提供结果](#)

[高级 Emscripten 功能](#)

[Embind](#)

[文件系统 API](#)

[Fetch API](#)

[在浏览器中调试](#)

[高级概述](#)

[使用源映射](#)

[总结](#)

[问题](#)

[进一步阅读](#)

[第七章：从头开始创建一个应用程序](#)

[Cook the Books – 使 WebAssembly 负责](#)

[概述和功能](#)

[使用的 JavaScript 库](#)

[Vue](#)

[UIkit](#)

[Lodash](#)

[数据驱动文档](#)

[其他库](#)

[C 和构建过程](#)

[项目设置](#)

[为 Node.js 配置](#)

[添加文件和文件夹](#)

[配置构建步骤](#)

[设置模拟 API](#)

[下载 C stdlib Wasm](#)

[最终结果](#)

[构建 C 部分](#)

[概述](#)

[关于工作流程的说明](#)

[C 文件内容](#)

[声明](#)

[链表操作](#)

[交易操作](#)

[交易计算](#)

[类别计算](#)

[编译为 Wasm](#)

[构建 JavaScript 部分](#)

[概述](#)

[关于浏览器兼容性的说明](#)

[在 initializeWasm.js 中创建一个 Wasm 实例](#)

[在 WasmTransactions.js 中与 Wasm 交互](#)

[在 api.js 中利用 API](#)

[在 store.js 中管理全局状态](#)

[导入和存储声明](#)

[交易操作](#)

[交易模态管理](#)

[余额计算](#)

[存储初始化](#)

[在 main.js 中加载应用程序](#)

[添加 web 资源](#)

[创建 Vue 组件](#)

[Vue 组件的结构](#)

[App 组件](#)

[BalancesBar](#)

[TransactionsTab](#)

[ChartsTab](#)

[运行应用程序](#)

[验证/src 文件夹](#)

[启动它！](#)

[测试一下](#)

[更改初始余额](#)

[创建新交易](#)

[删除现有交易](#)

[编辑现有交易](#)

[测试图表选项卡](#)

[总结](#)

[摘要](#)

[问题](#)

[进一步阅读](#)

[第八章：使用 Emscripten 移植游戏](#)

[游戏概述](#)

[什么是俄罗斯方块？](#)

[源的源](#)

[关于移植的说明](#)

[获取代码](#)

[构建本地项目](#)

[游戏的运行情况](#)

[深入了解代码库](#)

[将代码分解为对象](#)

[常量文件](#)

[方块类](#)

[构造函数和 draw\(\) 函数](#)

[move\(\)、rotate\(\) 和 isBlock\(\) 函数](#)

[getColumn\(\) 和 getRow\(\) 函数](#)

[Board 类](#)

[构造函数和 draw\(\) 函数](#)

[isCollision\(\) 函数](#)

[unite\(\) 函数](#)

[displayScore\(\) 函数](#)

[游戏类](#)

[构造函数和析构函数](#)

[loop\(\) 函数](#)

[主文件](#)

[移植到 Emscripten](#)

[为移植做准备](#)

[有什么改变？](#)

[添加 web 资源](#)

[移植现有代码](#)

[更新常量文件](#)

[构建和运行游戏](#)

[使用 VS Code 任务进行构建](#)

[使用 Makefile 进行构建](#)

[运行游戏](#)

[总结](#)

[问题](#)

[进一步阅读](#)

[第九章：与 Node.js 集成](#)

[为什么选择 Node.js ?](#)

[无缝集成](#)

[互补技术](#)

[使用 npm 进行开发](#)

[使用 Express 进行服务器端 WebAssembly](#)

[项目概述](#)

[Express 配置](#)

[使用 Node.js 实例化 Wasm 模块](#)

[创建模拟数据库](#)

[与 WebAssembly 模块交互](#)

[在 Transaction.js 中包装交互](#)

[在 assign-routes.js 中的交易操作](#)

[构建和运行应用程序](#)

[构建应用程序](#)

[启动和测试应用程序](#)

[使用 Webpack 进行客户端 WebAssembly](#)

[项目概述](#)

[什么是 Webpack ?](#)

[安装和配置 Webpack](#)

[依赖概述](#)

[在 webpack.config.js 中配置加载器和插件](#)

[C 代码](#)

[定义和声明](#)

[start\(\) 函数](#)

[更新 updateRectLocation\(\) 函数](#)

[JavaScript 代码](#)

[导入语句](#)

[组件状态](#)

[Wasm 初始化](#)

[组件挂载](#)

[组件渲染](#)

[构建和运行应用程序](#)

[测试构建](#)

[运行启动脚本](#)

[使用 Jest 测试 WebAssembly 模块](#)

[正在测试的代码](#)

[测试配置](#)

[测试文件审查](#)

[运行测试](#)

[总结](#)

[问题](#)

[进一步阅读](#)

[第十章：高级工具和即将推出的功能](#)

[WABT 和 Binaryen](#)

[WABT-WebAssembly 二进制工具包](#)

[Binaryen](#)

[使用 LLVM 编译](#)

[安装过程](#)

[示例代码](#)

[C++ 文件](#)

[HTML 文件](#)

[编译和运行示例](#)

[在线工具](#)

[WasmFiddle](#)

[WebAssembly Explorer](#)

[WebAssembly Studio](#)

[使用 Web Workers 实现并行 Wasm](#)

[Web Workers 和 WebAssembly](#)

[创建一个工作线程](#)

[WebAssembly 工作流程](#)

[谷歌 Chrome 中的限制](#)

[代码概述](#)

[C 代码](#)

[JavaScript 代码](#)

[在 worker.js 中定义线程执行](#)

[在 WasmWorker.js 中与 Wasm 交互](#)

[在 index.js 中加载应用程序](#)

[Web 资产](#)

[构建和运行应用程序](#)

[编译 C 文件](#)

[与应用程序交互](#)

[调试 Web Workers](#)

[即将推出的功能](#)

[标准化过程](#)
[线程](#)
[主机绑定](#)
[垃圾回收](#)
[引用类型](#)
[摘要](#)
[问题](#)
[进一步阅读](#)

前言

本书介绍了 WebAssembly，这是一项新颖而令人兴奋的技术，能够在浏览器中执行除 JavaScript 以外的其他语言。本书描述了如何从头开始构建一个 C/JavaScript 应用程序，使用 WebAssembly，并将现有的 C++ 代码库移植到浏览器中运行的过程，借助 Emscripten 的帮助。

WebAssembly 代表了 Web 平台的重要转变。作为诸如 C、C++ 和 Rust 等语言的编译目标，它提供了构建新型应用程序的能力。WebAssembly 得到了所有主要浏览器供应商的支持，并代表了一项协作努力。

在本书中，我们将描述构成 WebAssembly 的元素及其起源。我们将介绍安装所需工具、设置开发环境以及与 WebAssembly 交互的过程。我们将通过简单示例并逐渐深入的用例来工作。通过本书结束时，您将能够在 C、C++ 或 JavaScript 项目中充分利用 WebAssembly。

本书适合对象

如果您是希望为 Web 构建应用程序的 C/C++ 程序员，或者是希望改进其 JavaScript 应用程序性能的 Web 开发人员，那么本书适合您。本书面向熟悉 JavaScript 的开发人员，他们不介意学习一些 C 和 C++（反之亦然）。本书通过提供两个示例应用程序，同时考虑到了 C/C++ 程序员和 JavaScript 程序员的需求。

本书涵盖内容

第一章，什么是 WebAssembly？，描述了 WebAssembly 的起源，并提供了对该技术的高级概述。它涵盖了 WebAssembly 的用途，支持哪些编程语言以及当前的限制。

第二章，WebAssembly 的元素- Wat、Wasm 和 JavaScript API，概述了构成 WebAssembly 的元素。它详细解释了文本和二进制格式，以及相应的 JavaScript 和 Web API。

第三章，设置开发环境，介绍了用于开发 WebAssembly 的工具。它提供了每个平台的安装说明，并提供了改进开发体验的建议。

第四章，安装所需的依赖项，提供了每个平台安装工具链要求的说明。通过本章结束时，您将能够将 C 和 C++ 编译为 WebAssembly 模块。

第五章，创建和加载 WebAssembly 模块，解释了如何使用 Emscripten 生成 WebAssembly 模块，以及传递给编译器的标志如何影响生成的输出。它描述了在浏览器中加载 WebAssembly 模块的技术。

第六章，与 JavaScript 交互和调试，详细介绍了 Emscripten 的 Module 对象和浏览器的全局 WebAssembly 对象之间的区别。本章描述了 Emscripten 提供的功能，以及生成源映射的说明。

第七章，从头开始创建应用程序，介绍了创建一个与 WebAssembly 模块交互的 JavaScript 会计应用程序的过程。我们将编写 C 代码来计算会计交易的值，并在 JavaScript 和编译后的 WebAssembly 模块之间传递数据。

第八章，使用 Emscripten 移植游戏，采用逐步方法将现有的 C++ 游戏移植到 WebAssembly 上，使用 Emscripten。在审查现有的 C++ 代码库之后，对适当的文件进行更改，以使游戏能够在浏览器中运行。

第九章，与 Node.js 集成，演示了如何在服务器端和客户端使用 Node.js 和 npm 与 WebAssembly。本章涵盖了在 Express 应用程序中使用 WebAssembly，将 WebAssembly 与 webpack 集成以及使用 Jest 测试 WebAssembly 模块。

第十章，高级工具和即将推出的功能，涵盖了正在标准化过程中的高级工具，用例和新的 WebAssembly 功能。本章描述了 WABT，Binaryen 和在线可用的工具。在本章中，您将学习如何使用 LLVM 编译 WebAssembly 模块，以及如何将 WebAssembly 模块与 Web Workers 一起使用。本章最后描述了标准化过程，并审查了一些正在添加到规范中的令人兴奋的功能。

充分利用本书

您应该具有一些编程经验，并了解变量和函数等概念。如果您从未见过 JavaScript 或 C/C++ 代码，您可能需要在阅读本书的示例之前进行一些初步研究。我选择使用

JavaScript ES6/7 功能，如解构和箭头函数，因此如果您在过去 3-4 年内没有使用 JavaScript，语法可能会有些不同。

下载示例代码文件

您可以从www.packtpub.com的帐户中下载本书的示例代码文件。如果您在其他地方购买了本书，您可以访问www.packtpub.com/support并注册，以便将文件直接发送到您的邮箱。

您可以按照以下步骤下载代码文件：

1. 在www.packtpub.com上登录或注册。
2. 选择“支持”选项卡。
3. 单击“代码下载和勘误”。
4. 在搜索框中输入书名，然后按照屏幕上的说明操作。

下载文件后，请确保使用以下最新版本解压或提取文件夹：

- Windows 上的 WinRAR/7-Zip
- Mac 上的 ZipEg/iZip/UnRarX
- Linux 上的 7-Zip/PeaZip

本书的代码包也托管在 GitHub 上，网址为 github.com/PacktPublishing/Learn-WebAssembly。如果代码有更新，将在现有的 GitHub 存储库上进行更新。

我们还有其他代码包来自我们丰富的书籍和视频目录，可在 github.com/PacktPublishing/ 上找到。去看看吧！

下载彩色图片

我们还提供了一个 PDF 文件，其中包含本书中使用的屏幕截图/图表的彩色图片。您可以在此处下载：www.packtpub.com/sites/default/files/downloads/9781788997379_ColorImages.pdf。

使用的约定

本书中使用了许多文本约定。

CodeInText：表示文本中的代码词，数据库表名，文件夹名，文件名，文件扩展名，路径名，虚拟 URL，用户输入和 Twitter 句柄。例如：" `instantiate()` 是编译和实例化 WebAssembly 代码的主要 API。"

代码块设置如下：

```
int addTwo(int num) {  
    return num + 2;  
}
```

当我们希望引起您对代码块的特定部分的注意时，相关行或项目将以粗体显示：

```
int calculate(int firstVal, int secondVal) {  
    return firstVal - secondVal;  
}
```

任何命令行输入或输出都将按照以下格式编写：

```
npm install -g webassembly
```

粗体：表示新术语，重要单词或屏幕上看到的单词。例如，菜单或对话框中的单词会在文本中出现。例如：“您可以通过按下“开始”菜单按钮，右键单击“命令提示符”应用程序并选择“以管理员身份运行”来执行此操作。”

警告或重要说明会出现在这样的地方。

提示和技巧会出现在这样的地方。

第一章：什么是 WebAssembly？

WebAssembly (Wasm) 代表了 Web 平台的一个重要里程碑。使开发人员能够在 Web 上运行编译后的代码，而无需插件或浏览器锁定，带来了许多新的机会。关于 WebAssembly 是什么以及对其持续能力的一些怀疑，存在一些混淆。

在本章中，我们将讨论 WebAssembly 的产生过程，WebAssembly 在官方定义方面的含义以及它所涵盖的技术。将涵盖潜在的用例、支持的语言和局限性，以及如何找到额外的信息。

我们本章的目标是了解以下内容：

- 为 WebAssembly 铺平道路的技术
- WebAssembly 是什么以及它的一些潜在用例
- 可以与 WebAssembly 一起使用的编程语言
- WebAssembly 的当前局限性
- WebAssembly 与 Emscripten 和 asm.js 的关系

通往 WebAssembly 的道路

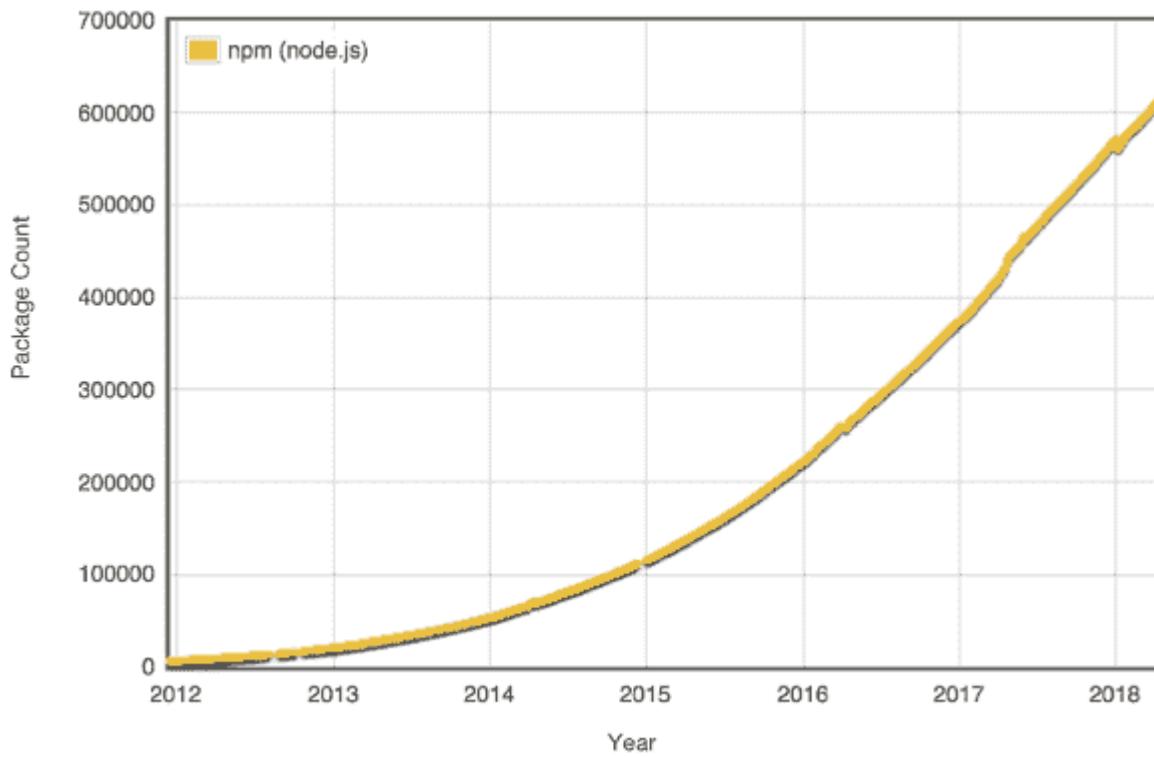
可以说，Web 开发有一个有趣的历史。已经进行了几次（失败的）尝试来扩展平台以支持不同的语言。诸如插件之类的笨拙解决方案未能经受住时间的考验，而将用户限制在单个浏览器上则是一种灾难的预兆。

WebAssembly 作为一个优雅的解决方案，解决了自从浏览器能够执行代码以来一直存在的问题：如果你想为 Web 开发，你必须使用 JavaScript。幸运的是，使用 JavaScript 并没有像在 2000 年代初那样带有负面含义，但它作为一种编程语言仍然有一定的局限性。在本节中，我们将讨论导致 WebAssembly 出现的技术，以更好地理解为什么需要这种新技术。

JavaScript 的演变

JavaScript 是由 Brendan Eich 在 1995 年的短短 10 天内创建的。最初被程序员视为一种玩具语言，主要用于在网页上制作按钮闪烁或横幅出现。过去的十年里，JavaScript 已经从一个玩具演变成了一个具有深远能力和庞大追随者的平台。

2008 年，浏览器市场的激烈竞争导致了即时（JIT）编译器的添加，这提高了 JavaScript 的执行速度 10 倍。Node.js 于 2009 年首次亮相，代表了 Web 开发的范式转变。Ryan Dahl 结合了谷歌的 V8 JavaScript 引擎、事件循环和低级 I/O API，构建了一个平台，允许在服务器和客户端使用 JavaScript。Node.js 导致了 npm，这是一个允许在 Node.js 生态系统内使用的库的包管理器。截至撰写本文时，有超过 60 万个可用的包，每天都有数百个包被添加：



自 2012 年以来 npm 包数量的增长，来自 Modulecounts

不仅是 Node.js 生态系统在增长；JavaScript 本身也在积极发展。ECMA 技术委员会 39 (TC39) 规定了 JavaScript 的标准，并监督新语言特性的添加，每年发布一次 JavaScript 的更新，采用社区驱动的提案流程。凭借其丰富的库和工具、对语言的不断改进以及拥有最庞大的程序员社区之一，JavaScript 已经成为一个不可忽视的力量。

但是这种语言确实有一些缺点：

- 直到最近，JavaScript 只包括 64 位浮点数。这可能会导致非常大或非常小的数字出现问题。`BigInt` 是一种新的数值原语，可以缓解一些这些问题，正在被添加到 ECMAScript 规范中，但可能需要一些时间才能在浏览器中得到完全支持。
- JavaScript 是弱类型的，这增加了它的灵活性，但可能会导致混淆和错误。它基本上给了你足够的绳子来绞死自己。
- 尽管浏览器供应商尽最大努力，但 JavaScript 并不像编译语言那样高效。
- 如果开发人员想要创建 Web 应用程序，他们需要学习 JavaScript——不管他们喜不喜欢。

为了避免编写超过几行 JavaScript，一些开发人员构建了**转译器**，将其他语言转换为 JavaScript。转译器（或源到源编译器）是一种将一种编程语言的源代码转换为另一种编程语言等效源代码的编译器。TypeScript 是前端 JavaScript 开发的流行工具，将 TypeScript 转译为针对浏览器或 Node.js 的有效 JavaScript。选择任何编程语言，都有很大可能有人为其创建了 JavaScript 转译器。例如，如果你喜欢编写 Python，你有大约 15 种不同的工具可以用来生成 JavaScript。但最终，它仍然是 JavaScript，因此你仍然受到该语言的特殊性的影响。

随着 Web 逐渐成为构建和分发应用程序的有效平台，越来越复杂和资源密集型的应用程序被创建。为了满足这些应用程序的需求，浏览器供应商开始研发新技术，将其集成到软件中，而不会干扰 Web 开发的正常进程。谷歌和 Mozilla 分别是 Chrome 和 Firefox 的创建者，他们采取了两种不同的路径来实现这一目标，最终形成了 WebAssembly。

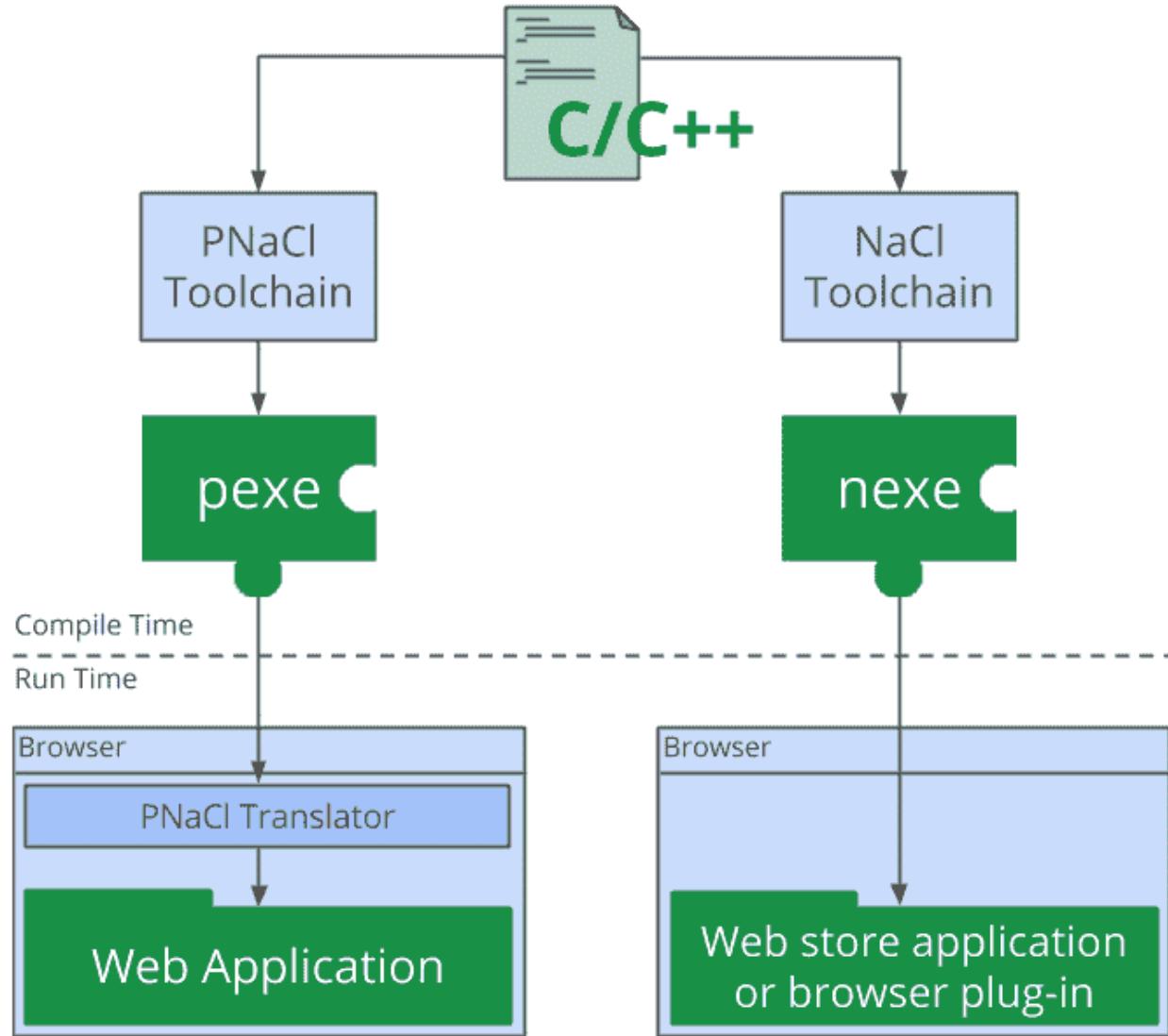
谷歌和 Native Client

谷歌开发了**Native Client (NaCl)**，旨在安全地在 Web 浏览器中运行本机代码。可执行代码将在**沙盒**中运行，并提供本机代码执行的性能优势。

在软件开发的背景下，沙盒是一个环境，防止可执行代码与系统的其他部分进行交互。它旨在防止恶意代码的传播，并对软件的操作进行限制。

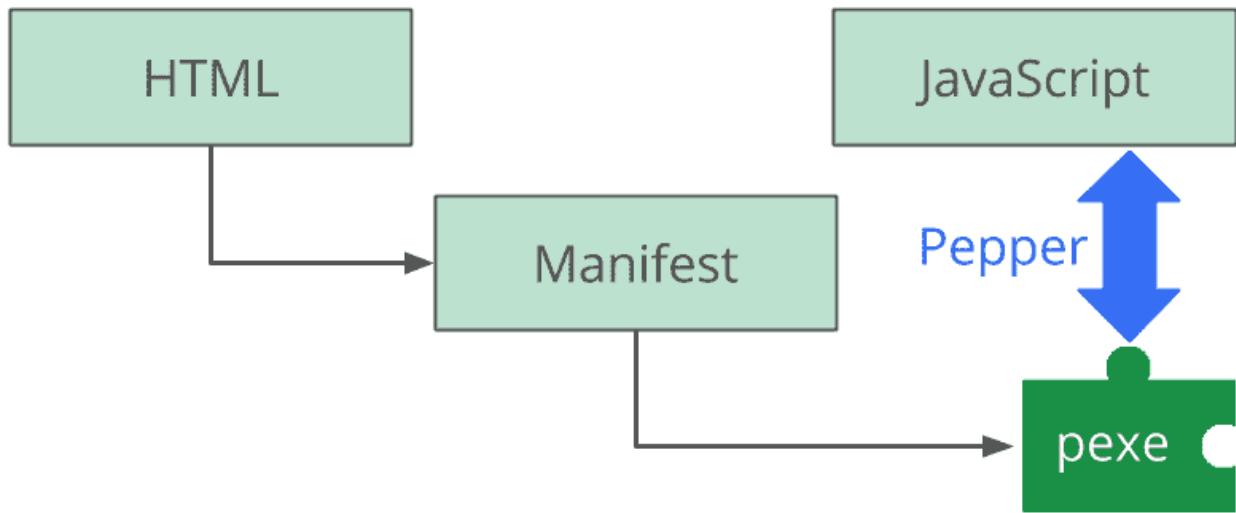
NaCl 与特定架构相关，而**Portable Native Client (PNaCl)** 是 NaCl 的独立于架构的版本，可在任何平台上运行。该技术由两个元素组成：

- 可以将 C/C++ 代码转换为 NaCl 模块的工具链
- 运行时组件是嵌入在浏览器中的组件，允许执行 NaCl 模块：



本机客户端工具链及其输出

NaCl 的特定架构可执行文件（`nexe`）仅限于从谷歌 Chrome Web 商店安装的应用程序和扩展，但 PNaCl 可执行文件（`pexe`）可以在 Web 上自由分发并嵌入 Web 应用程序中。Pepper 使得可移植性成为可能，Pepper 是用于创建 NaCl 模块的开源 API，以及其相应的插件 API（PPAPI）。Pepper 实现了 NaCl 模块与托管浏览器之间的通信，并以安全和可移植的方式访问系统级功能。通过包含清单文件和已编译模块（`pexe`）以及相应的 HTML、CSS 和 JavaScript，应用程序可以轻松分发：



Pepper 在本机客户端应用程序中的作用

NaCl 提供了克服 Web 性能限制的有希望的机会，但也有一些缺点。尽管 Chrome 内置支持 PNaCl 可执行文件和 Pepper，其他主要浏览器却没有。技术的反对者对应用程序的黑盒性质以及潜在的安全风险和复杂性表示了异议。

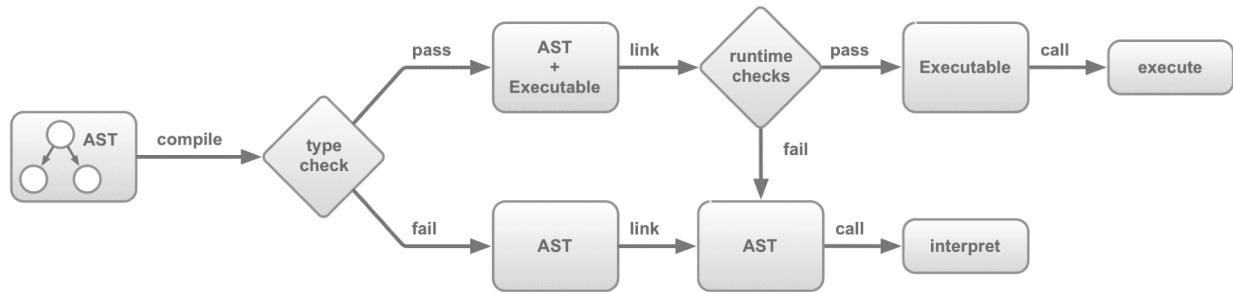
Mozilla 致力于改进 JavaScript 的性能，使用 `asm.js`。由于 API 规范的不完整和文档有限，他们不会为 Firefox 添加对 Pepper 的支持。最终，NaCl 于 2017 年 5 月被弃用，改为支持 WebAssembly。

Mozilla 和 `asm.js`

Mozilla 于 2013 年推出了 `asm.js`，并为开发人员提供了一种将其 C 和 C++ 源代码转换为 JavaScript 的方法。`asm.js` 的官方规范将其定义为 JavaScript 的严格子集，可用作编译器的低级高效目标语言。它仍然是有效的 JavaScript，但语言特性仅限于适合**提前（AOT）**优化的特性。AOT 是浏览器的 JavaScript 引擎用来通过将其编译为本机机器代码来更有效地执行代码的技术。`asm.js` 通过具有 100% 类型一致性和手动内存管理来实现这些性能增益。

使用 Emscripten 等工具，C/C++ 代码可以被转译成 `asm.js`，并且可以使用与普通 JavaScript 相同的方式进行分发。访问 `asm.js` 模块中的函数需要**链接**，这涉及调用其函数以获取具有模块导出的对象。

`asm.js` 非常灵活，但是与模块的某些交互可能会导致性能损失。例如，如果 `asm.js` 模块被赋予访问自定义 JavaScript 函数的权限，而该函数未通过动态或静态验证，代码就无法利用 AOT 并会退回到解释器：



asm.js 的 AOT 编译工作流程

asm.js 不仅仅是一个过渡阶段。它构成了 WebAssembly 的**最小可行产品（MVP）**的基础。官方 WebAssembly 网站在标题为 *WebAssembly 高级目标* 的部分明确提到了 asm.js。

那么为什么要创建 WebAssembly 而不使用 asm.js 呢？除了潜在的性能损失外，asm.js 模块是一个必须在编译之前通过网络传输的文本文件。WebAssembly 模块是以二进制格式，这使得由于其较小的大小而更加高效地传输。

WebAssembly 模块使用基于 promise 的实例化方法，利用现代 JavaScript 并消除了任何这个加载了吗的代码。

WebAssembly 的诞生

万维网联盟（W3C）是一个致力于制定 Web 标准的国际社区，于 2015 年 4 月成立了 WebAssembly 工作组，以标准化 WebAssembly 并监督规范和提案过程。自那时起，核心规范和相应的 JavaScript API 和 Web API 已经发布。浏览器中对 WebAssembly 支持的初始实现是基于 asm.js 的功能集。WebAssembly 的二进制格式和相应的 .wasm 文件结合了 asm.js 输出的特征和 PNaCl 的分布式可执行概念。

那么 WebAssembly 将如何成功，而 NaCl 失败了呢？根据 Axel Rauschmayer 博士的说法，详细原因在 zality.com/2015/06/web-assembly.html#what-is-different-this-time 中有三个原因。

“首先，这是一个协作努力，没有任何一家公司单独进行。目前，涉及的项目有：Firefox，Chromium，Edge 和 WebKit。”

其次，与 Web 平台和 JavaScript 的互操作性非常出色。从 JavaScript 中使用 WebAssembly 代码将像导入模块一样简单。

第三，这不是要取代 JavaScript 引擎，而是要为它们增加一个新功能。这大大减少了实现 WebAssembly 的工作量，并有助于获得 Web 开发社区的支持。”

- Dr. Axel Rauschmayer

WebAssembly 到底是什么，我在哪里可以使用它？

WebAssembly 在官方网站上有一个简明扼要的定义，但这只是一个部分。

WebAssembly 还有其他几个组件。了解每个组件的作用将让您更好地理解整个技术。在本节中，我们将详细解释 WebAssembly 的定义，并描述潜在的用例。

官方定义

官方的 WebAssembly 网站（webassembly.org）提供了这个定义：

Wasm 是一种基于堆栈的虚拟机的二进制指令格式。Wasm 被设计为高级语言（如 C/C++/Rust）的可移植编译目标，从而可以在 Web 上部署客户端和服务器应用程序。

让我们把这个定义分解成几个部分，以便更清楚地解释。

二进制指令格式

WebAssembly 实际上包括几个元素——二进制格式和文本格式，这些都在核心规范中有文档记录，对应的 API（JavaScript 和 Web），以及一个编译目标。二进制和文本格式都映射到一个公共结构，以**抽象语法**的形式存在。为了更好地理解抽象语法，可以在**抽象语法树（AST）**的上下文中解释。AST 是编程语言源代码结构的树形表示。诸如 ESLint 之类的工具使用 JavaScript 的 AST 来查找 linting 错误。以下示例包含 JavaScript 的函数和相应的 AST（来自 astexplorer.net）。

一个简单的 JavaScript 函数如下：

```
function doStuff(thingToDo) {  
    console.log(thingToDo);  
}
```

相应的 AST 如下：

```
{  
  "type": "Program",  
  "start": 0,  
  "end": 57,  
  "body": [  
    {
```

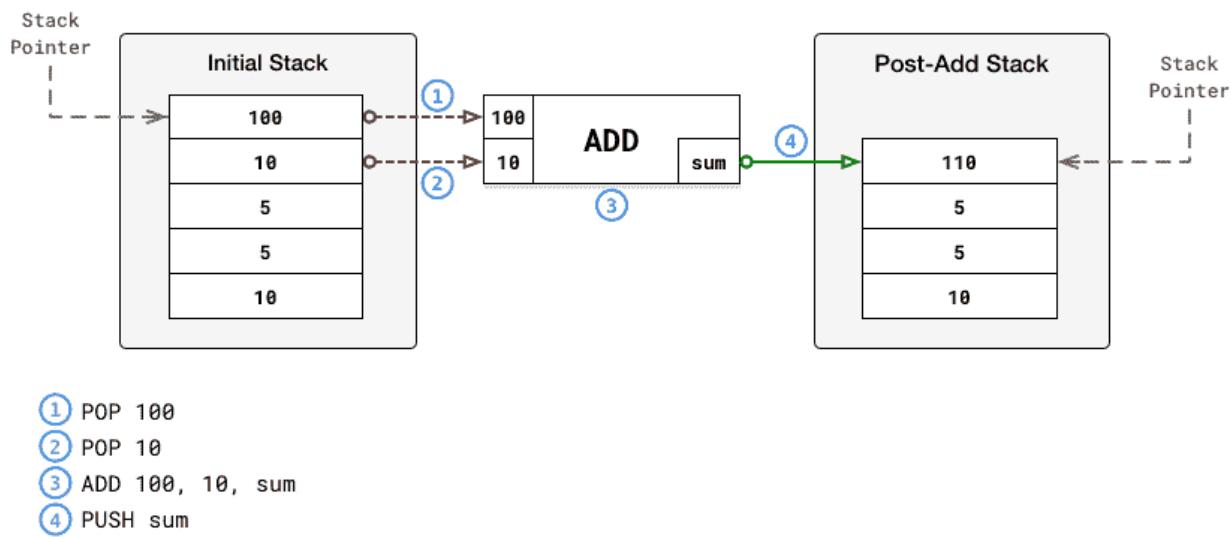
```
"type": "FunctionDeclaration",
"start": 9,
"end": 16,
"id": {
  "type": "Identifier",
  "start": 17,
  "end": 26,
  "name": "doStuff"
},
"generator": false,
"expression": false,
"params": [
  {
    "type": "Identifier",
    "start": 28,
    "end": 57,
    "name": "thingToDo"
  }
],
"body": {
  "type": "BlockStatement",
  "start": 32,
  "end": 55,
  "body": [
    {
      "type": "ExpressionStatement",
      "start": 32,
      "end": 55,
      "expression": {
        "type": "CallExpression",
        "start": 32,
        "end": 54,
        "callee": {
          "type": "MemberExpression",
          "start": 32,
          "end": 43,
```

```
        "object": {
            "type": "Identifier",
            "start": 32,
            "end": 39,
            "name": "console"
        },
        "property": {
            "type": "Identifier",
            "start": 40,
            "end": 43,
            "name": "log"
        },
        "computed": false
    },
    "arguments": [
        {
            "type": "Identifier",
            "start": 44,
            "end": 53,
            "name": "thingToDo"
        }
    ]
}
],
"sourceType": "module"
}
```

AST 可能会很冗长，但它在描述程序的组件方面做得很好。在 AST 中表示源代码使得验证和编译变得简单高效。WebAssembly 文本格式的代码被序列化为 AST，然后编译为二进制格式（作为 `.wasm` 文件），然后被网页获取、加载和利用。模块加载时，浏览器的 JavaScript 引擎利用**解码堆栈**将 `.wasm` 文件解码为 AST，执行类型检查，并解释执行函

数。WebAssembly 最初是用于 AST 的二进制指令格式。由于验证返回 `void` 的 Wasm 表达式的性能影响，二进制指令格式已更新为针对堆栈机。

堆栈机由两个元素组成：堆栈和指令。堆栈是一个具有两个操作的数据结构：`push` 和 `pop`。项目被推送到堆栈上，然后按照后进先出（LIFO）的顺序从堆栈中弹出。堆栈还包括一个指针，指向堆栈顶部的项目。指令表示对堆栈中项目执行的操作。例如，一个 `ADD` 指令可能从堆栈中弹出顶部的两个项目（值为 `100` 和 `10`），并将总和推回到堆栈上（值为 `110`）：



一个简单的堆栈机

WebAssembly 的堆栈机操作方式相同。程序计数器（指针）维护代码中的执行位置，虚拟控制堆栈跟踪 `blocks` 和 `if` 结构的进入（推入）和退出（弹出）。指令执行时不涉及 AST。因此，定义中的**二进制指令格式**部分指的是一种二进制表示的指令，这些指令可以被浏览器中的解码堆栈读取。

可移植的编译目标

WebAssembly 从一开始就考虑了可移植性。在这个上下文中，可移植性意味着 WebAssembly 的二进制格式可以在各种操作系统和指令集架构上高效地执行，无论是在 Web 上还是离线。WebAssembly 的规范定义了执行环境中的可移植性。WebAssembly 被设计为在符合某些特征的环境中高效运行，其中大部分与内存有关。WebAssembly 的可移植性也可以归因于核心技术周围缺乏特定的 API。相反，它定义了一个 `import` 机制，其中可用的导入集由宿主环境定义。

简而言之，这意味着 WebAssembly 不与特定环境绑定，比如 Web 或桌面。

WebAssembly 工作组已经定义了一个 Web API，但这与 核心规范 是分开的。Web API 适用于 WebAssembly，而不是反过来。

定义中的编译方面表明，WebAssembly 从高级语言编写的源代码编译成其二进制格式将会很简单。MVP 关注两种语言，C 和 C++，但由于 Rust 与 C++ 相似，也可以使用。编译将通过使用 Clang/LLVM 后端来实现，尽管在本书中我们将使用 Emscripten 生成我们的 Wasm 模块。计划最终支持其他语言和编译器（比如 GCC），但 MVP 专注于 LLVM。

核心规范

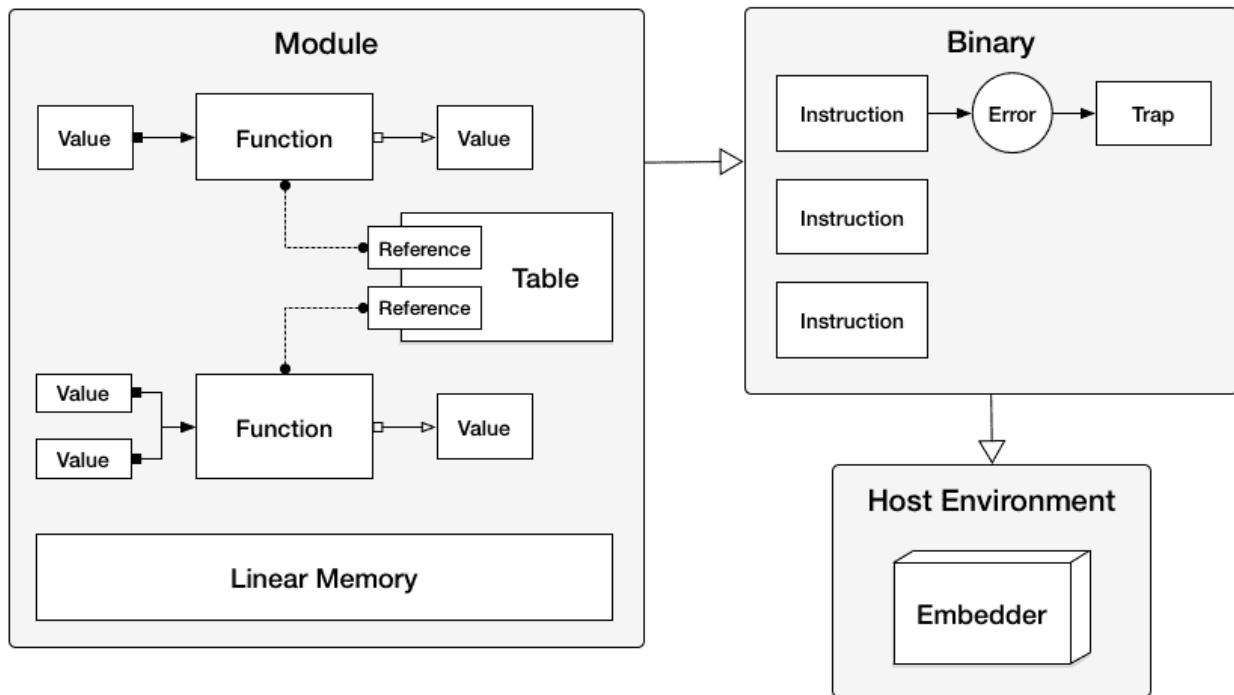
官方定义为我们提供了对整体技术的高层洞察，但为了完整起见，值得深入挖掘一下。WebAssembly 的 核心规范 是官方文档，如果你想深入了解 WebAssembly，可以参考这个文档。如果你对运行时结构的特征感兴趣，可以查看第 4 节：执行。我们在这里不会涉及这一点，但了解 核心规范 的位置将有助于建立对 WebAssembly 的完整定义。

语言概念

核心规范 表明 WebAssembly 编码了一种低级的、类似汇编的编程语言。规范定义了这种语言的结构、执行和验证，以及二进制和文本格式的细节。语言本身围绕以下概念构建：

- 值，或者说 WebAssembly 提供的值类型
- 在堆栈机器内执行的指令
- 在错误条件下产生的陷阱并中止执行
- 函数，代码组织成的函数，每个函数都以一系列值作为参数，并返回一系列值作为结果
- 表，这是特定元素类型（比如函数引用）的值数组，可以被执行程序选择
- 线性内存，这是一个原始字节的数组，可以用来存储和加载值
- 模块，WebAssembly 二进制（.wasm 文件）包含函数、表和线性内存
- 嵌入器，WebAssembly 可以在宿主环境（比如 Web 浏览器）中执行的机制

函数、表、内存和模块与 JavaScript API 直接相关，对此有所了解是很重要的。这些概念描述了语言本身的基本结构以及如何编写或编码 WebAssembly。就使用而言，理解 WebAssembly 对应的语义阶段提供了对该技术的完整定义：



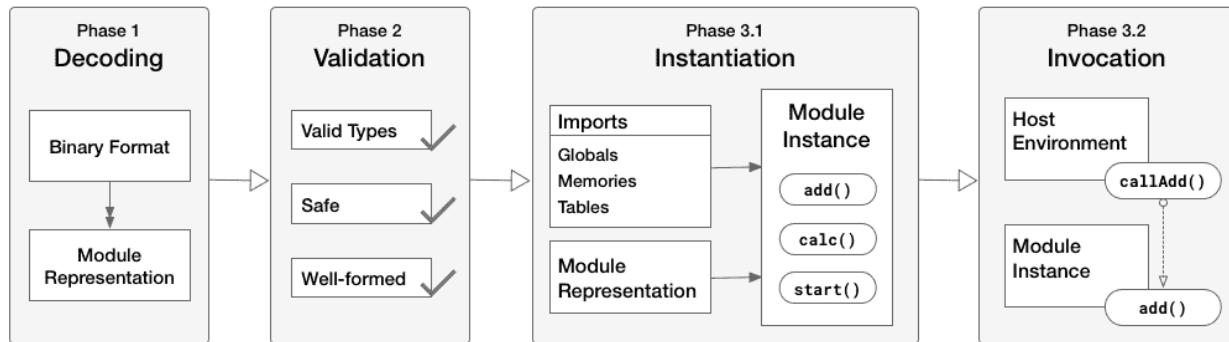
语言概念及其关系

语义阶段

核心规范描述了编码模块（`.wasm` 文件）在宿主环境（比如 Web 浏览器）中被利用时经历的不同阶段。规范的这一方面代表了输出是如何处理和执行的：

- **解码**：将二进制格式转换为模块
- **验证**：解码模块经过验证检查（例如类型检查），以确保模块形式良好且安全
- **执行，第 1 部分：实例化**：通过初始化全局变量、内存和表来实例化模块实例，然后调用模块的 `start()` 函数
- **执行，第 2 部分：调用**：从模块实例调用导出的函数：

以下图表提供了语义阶段的可视化表示：



模块使用的语义阶段

JavaScript 和 Web API

WebAssembly 工作组还发布了与 JavaScript 和 Web 交互的 API 规范，使它们有资格被纳入 WebAssembly 技术领域。JavaScript API 的范围仅限于 JavaScript 语言本身，而不是特定于环境（例如 Web 浏览器或 Node.js）。它定义了用于与 WebAssembly 交互和管理编译和实例化过程的类、方法和对象。Web API 是 JavaScript API 的扩展，定义了特定于 Web 浏览器的功能。Web API 规范目前仅定义了两种方法，`compileStreaming` 和 `instantiateStreaming`，这些是简化在浏览器中使用 Wasm 模块的便利方法。这些将在第二章中更详细地介绍，WebAssembly 的要素 - Wat、Wasm 和 JavaScript API。

那么它会取代 JavaScript 吗？

WebAssembly 的最终目标不是取代 JavaScript，而是补充它。JavaScript 丰富的生态系统和灵活性仍然使其成为 Web 的理想语言。WebAssembly 的 JavaScript API 使得两种技术之间的互操作性相对简单。那么你是否能够只使用 WebAssembly 构建 Web 应用程序？WebAssembly 的一个明确目标是可移植性，复制 JavaScript 的所有功能可能会阻碍该目标。然而，官方网站包括一个目标，即执行并与现有 Web 平台很好地集成，所以只有时间能告诉我们。在一种编译为 WebAssembly 的语言中编写整个代码库可能并不实际，但将一些应用程序逻辑移动到 Wasm 模块可能在性能和加载时间方面有益。

我可以在哪里使用它？

WebAssembly 的官方网站列出了大量潜在的用例。我不打算在这里覆盖它们所有，但有几个代表了对 Web 平台功能的重大增强：

- 图像/视频编辑
- 游戏

- 音乐应用程序（流媒体、缓存）
- 图像识别
- 实时视频增强
- 虚拟现实和增强现实

尽管一些用例在技术上可以使用 JavaScript、HTML 和 CSS 实现，但使用 WebAssembly 可以带来显著的性能提升。提供一个二进制文件（而不是单个 JavaScript 文件）可以大大减少捆绑包大小，并且在页面加载时实例化 Wasm 模块可以加快代码执行速度。

WebAssembly 不仅仅局限于浏览器。在浏览器之外，您可以使用它来构建移动设备上的混合本机应用程序，或者执行不受信任代码的服务器端计算。在手机应用程序中使用 Wasm 模块可能在功耗和性能方面非常有益。

WebAssembly 在使用上也提供了灵活性。您可以在 WebAssembly 中编写整个代码库，尽管在当前形式或 Web 应用程序的上下文中可能不太实际。鉴于 WebAssembly 的强大 JavaScript API，您可以在 JavaScript/HTML 中编写 UI，并使用 Wasm 模块来实现不直接访问 DOM 的功能。一旦支持了其他语言，对象就可以在 Wasm 模块和 JavaScript 代码之间轻松传递，这将大大简化集成并增加开发者的采用率。

支持哪些语言？

WebAssembly 的 MVP 的高级目标是提供与 `asm.js` 大致相同的功能。这两种技术非常相关。C、C++ 和 Rust 是非常受欢迎的支持手动内存分配的语言，这使它们成为最初实现的理想候选。在本节中，我们将简要概述每种编程语言。

C 和 C++

C 和 C++ 是已经存在 30 多年的低级编程语言。C 是过程化的，不本质上支持类和继承等面向对象编程概念，但它快速、可移植且被广泛使用。

C++ 是为了填补 C 的不足而构建的，它添加了诸如运算符重载和改进的类型检查等功能。这两种语言一直稳居前 10 最受欢迎的编程语言之列，这使它们非常适合 MVP：

Programming Language	2018	2013	2008	2003	1998	1993	1988
Java	1	2	1	1	16	-	-
C	2	1	2	2	1	1	1
C++	3	4	3	3	2	2	5
Python	4	7	6	12	24	19	-
C#	5	5	7	9	-	-	-
Visual Basic .NET	6	13	-	-	-	-	-
JavaScript	7	10	8	7	21	-	-
PHP	8	6	4	5	-	-	-
Ruby	9	9	9	19	-	-	-
Perl	10	8	5	4	3	11	-
Objective-C	18	3	44	50	-	-	-
Ada	30	16	17	14	6	6	2
Lisp	31	11	15	13	5	4	3
Pascal	143	14	18	97	11	3	13

TIOBE 长期历史上前 10 种编程语言的排名

C 和 C++的支持也内置在 Emscripten 中，因此除了简化编译过程，它还允许你充分利用 WebAssembly 的功能。还可以使用 LLVM 将 C/C++代码编译成 `.wasm` 文件。LLVM 是一组模块化和可重用的编译器和工具链技术。简而言之，它是一个简化从源代码到机器代码的编译过程配置的框架。如果你想制作自己的编程语言并且想要构建编译器，LLVM 有工具来简化这个过程。我将在第十章中介绍如何使用 LLVM 将 C/C++编译成 `.wasm` 文件，高级工具和即将推出的功能。

以下片段演示了如何使用 C++将“Hello World！”打印到控制台：

```
#include <iostream>int main() {
    std::cout << "Hello, World!\n";
    return 0;
}
```

Rust

C 和 C++原本是 WebAssembly 的主要使用语言，但 Rust 也是一个完全合适的替代品。Rust 是一种系统编程语言，语法与 C++类似。它设计时考虑了内存安全性，但仍保留了 C 和 C++的性能优势。Rust 当前的夜间构建版本的编译器可以从 Rust 源代码生

成 `.wasm` 文件，因此如果你更喜欢 Rust 并且熟悉 C++，你应该能够在本书的大多数示例中使用 Rust。

以下片段演示了如何使用 Rust 将“Hello World！”打印到控制台：

```
fn main() {
    println!("Hello World!");
}
```

其他语言

还存在各种工具，可以使其他流行的编程语言与 WebAssembly 一起使用，尽管它们大多是实验性的：

- 通过 Blazor 的 C#
- 通过 WebIDL 的 Haxe
- 通过 TeaVM 或 Bytecoder 的 Java
- 通过 TeaVM 的 Kotlin
- 通过 AssemblyScript 的 TypeScript

技术上也可以将一种语言转译为 C，然后将其编译为 Wasm 模块，但编译的成功取决于转译器的输出。很可能你需要对代码进行重大更改才能使其正常工作。

有哪些限制？

诚然，WebAssembly 并非没有局限性。新功能正在积极开发，技术不断发展，但 MVP 功能仅代表了 WebAssembly 功能的一部分。在本节中，我们将介绍其中一些限制以及它们对开发过程的影响。

没有垃圾回收

WebAssembly 支持平面线性内存，这本身并不是一个限制，但需要一些了解如何显式分配内存以执行代码。C 和 C++ 是 MVP 的逻辑选择，因为内存管理内置于语言中。一开始没有包括一些更流行的高级语言，比如 Java，原因是**垃圾回收（GC）**。

GC 是一种自动内存管理形式，程序不再使用的对象占用的内存会被自动回收。GC 类似于汽车上的自动变速器。经过熟练工程师的大力优化，它可以尽可能高效地运行，但限制了驾驶员的控制量。手动分配内存就像驾驶手动变速器的汽车。它可以更好地控制速度和

扭矩，但错误使用或缺乏经验可能会导致汽车严重损坏。C 和 C++ 的出色性能和速度部分归功于手动分配内存。

GC 语言允许您编程而无需担心内存可用性或分配。JavaScript 就是一个 GC 语言的例子。浏览器引擎采用一种称为标记-清除算法来收集不可达对象并释放相应的内存。

WebAssembly 目前正在努力支持 GC 语言，但很难准确说出何时会完成。

没有直接的 DOM 访问

WebAssembly 无法访问 DOM，因此任何 DOM 操作都需要间接通过 JavaScript 或使用诸如 Emscripten 之类的工具来完成。有计划添加引用 DOM 和其他 Web API 对象的能力，但目前仍处于提案阶段。DOM 操作可能会与 GC 语言紧密相关，因为它将允许在 WebAssembly 和 JavaScript 代码之间无缝传递对象。

旧版浏览器不支持

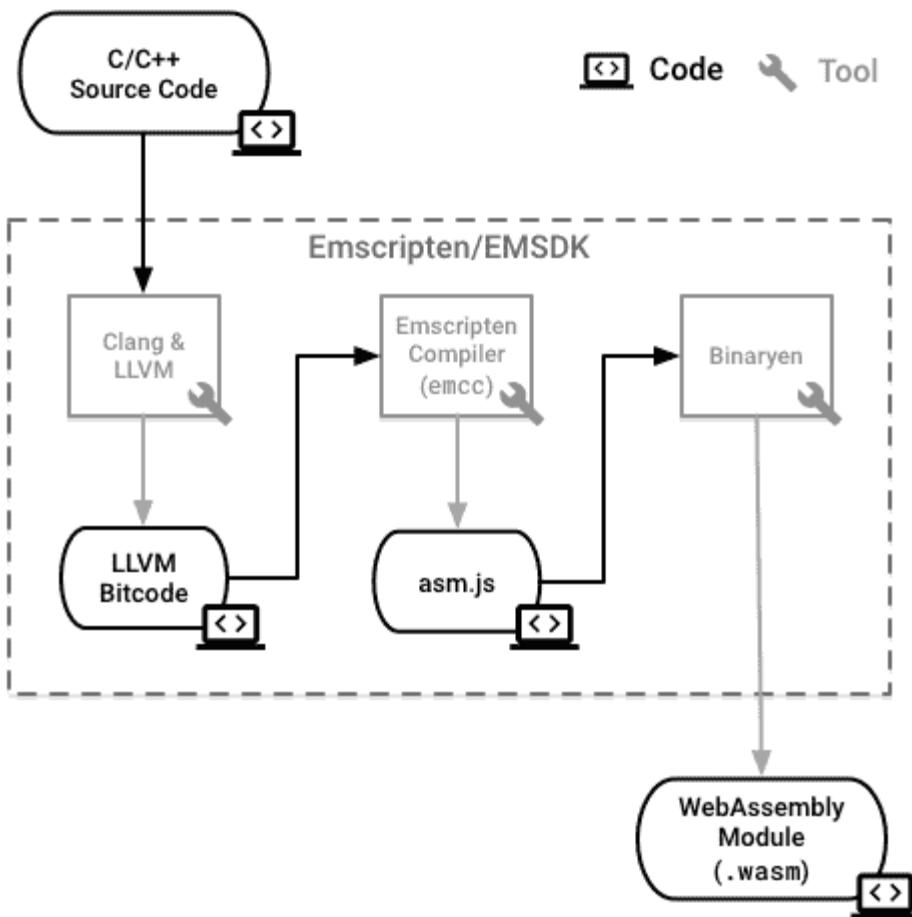
旧版浏览器没有全局的 `WebAssembly` 对象可用来实例化和加载 Wasm 模块。如果找不到该对象，有一些实验性的 polyfills 会使用 `asm.js`，但 WebAssembly 工作组目前没有创建的计划。由于 `asm.js` 和 WebAssembly 密切相关，如果 `WebAssembly` 对象不可用，简单地提供一个 `asm.js` 文件仍然可以提供性能增益，同时适应向后兼容性。您可以在 caniuse.com/#feat=wasm 上查看当前支持 WebAssembly 的浏览器。

它与 Emscripten 有什么关系？

Emscripten 是可以从 C 和 C++ 源代码生成 `asm.js` 的源到源编译器。我们将使用它作为一个构建工具来生成 Wasm 模块。在本节中，我们将快速回顾 Emscripten 与 WebAssembly 的关系。

Emscripten 的作用

Emscripten 是一个 LLVM 到 JavaScript 的编译器，这意味着它接受诸如 Clang（用于 C 和 C++）的编译器的 LLVM 位码输出，并将其转换为 JavaScript。它不是一个特定的技术，而是一组技术的组合，它们一起构建、编译和运行 `asm.js`。为了生成 Wasm 模块，我们将使用 **Emscripten SDK (EMSDK)** 管理器：



使用 EMSDK 生成 Wasm 模块

EMSDK 和 Binaryen

在第四章中，安装所需的依赖项，我们将安装 EMSDK 并使用它来管理编译 C 和 C++ 到 Wasm 模块所需的依赖项。Emscripten 使用 Binaryen 的 `asm2wasm` 工具将 Emscripten 输出的 `asm.js` 编译成 `.wasm` 文件。Binaryen 是一个编译器和工具链基础库，包括将各种格式编译成 WebAssembly 模块以及反之的工具。了解 Binaryen 的内部工作对于使用 WebAssembly 并不是必需的，但重要的是要意识到底层技术以及它们如何协同工作。通过将某些标志传递给 Emscripten 的编译命令 (`emcc`)，我们可以将结果的 `asm.js` 代码传递给 Binaryen 以输出我们的 `.wasm` 文件。

总结

在本章中，我们讨论了与 WebAssembly 的历史相关的技术，以及导致其创建的技术。提供了对 WebAssembly 定义的详细概述，以便更好地理解涉及的底层技术。

核心规范、*JavaScript API* 和 *Web API* 被提出为 WebAssembly 的重要元素，并展示了技术将如何发展。我们还审查了潜在的用例、当前支持的语言以及使非支持语言可用的工具。

WebAssembly 的局限性是缺乏 GC、无法直接与 DOM 通信以及不支持旧版浏览器。这些都是为了传达技术的新颖性并揭示其中一些缺点而进行讨论的。最后，我们讨论了 Emscripten 在开发过程中的作用以及它在 WebAssembly 开发工作流程中的位置。

在第二章中，*WebAssembly 元素 - Wat、Wasm 和 JavaScript API*，我们将更深入地探讨构成 WebAssembly 的元素：**WebAssembly 文本格式（Wat）**、二进制格式（Wasm）、JavaScript 和 Web API。

问题

1. 哪两种技术影响了 WebAssembly 的创建？
2. 什么是堆栈机器，它与 WebAssembly 有什么关系？
3. WebAssembly 如何补充 JavaScript？
4. 哪三种编程语言可以编译成 Wasm 模块？
5. LLVM 在 WebAssembly 方面扮演什么角色？
6. WebAssembly 有哪三个潜在的用例？
7. DOM 访问和 GC 有什么关系？
8. Emscripten 使用什么工具来生成 Wasm 模块？

进一步阅读

- 官方 WebAssembly 网站：webassembly.org
- 原生客户端技术概述：developer.chrome.com/native-client/overview
- LLVM 编译器基础设施项目：llvm.org
- 关于 Emscripten：kripken.github.io/emscripten-site/docs/introducing_emscripten/about_emscripten.html
- asm.js 规范：asmjs.org/spec/latest

第二章：WebAssembly 的元素-Wat、Wasm 和 JavaScript API

第一章《什么是 WebAssembly ?》描述了 WebAssembly 的历史，并提供了技术的高层概述以及潜在的用例和限制。WebAssembly 被描述为由多个元素组成，不仅仅是官方定义中指定的二进制指令格式。

在本章中，我们将深入研究与 WebAssembly 工作组创建的官方规范相对应的元素。我们将更详细地检查 Wat 和二进制格式，以更好地理解它们与模块的关系。我们将审查 *JavaScript API* 和 *Web API*，以确保您能够有效地在浏览器中使用 WebAssembly。

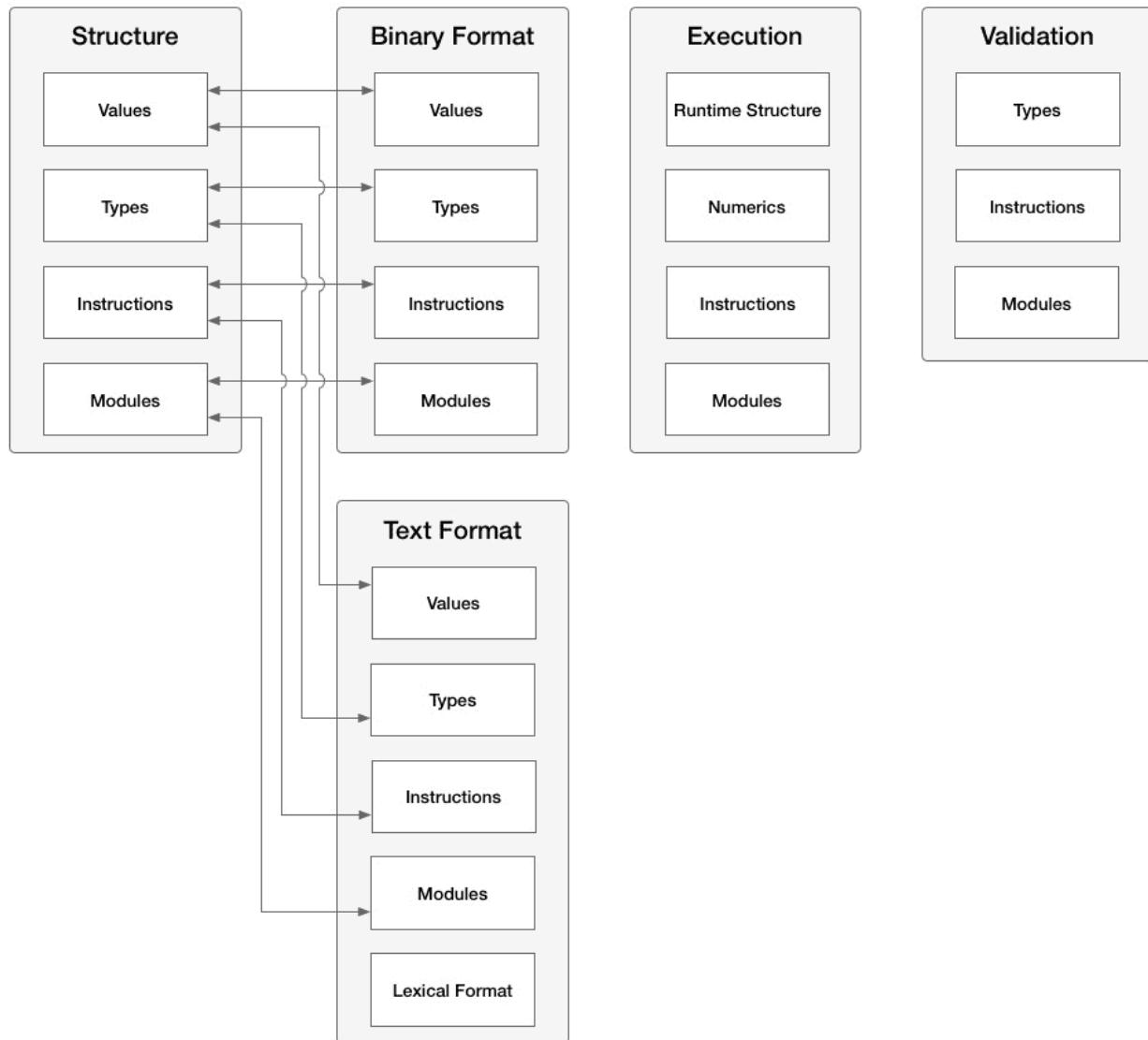
本章的目标是理解以下内容：

- 文本和二进制格式之间的关系
- Wat 是什么以及它在开发过程中的作用
- 二进制格式和模块（Wasm）文件
- JavaScript 和 Web API 的组件以及它们与 Wasm 模块的关系
- 如何利用 WasmFiddle 评估 WebAssembly 的阶段（C/C++ > Wat > Wasm）

共同结构和抽象语法

在第一章中，《什么是 WebAssembly ?》，我们讨论了 WebAssembly 的二进制和文本格式如何映射到抽象语法的共同结构。在深入了解这些格式之前，值得一提的是它们在核心规范中的关系。以下图表是目录的可视化表示（为了清晰起见，排除了一些部分）：

Core Specification



核心规范

目录

正如您所看到的，**文本格式**和**二进制格式**部分包含与**结构**部分相关的**值、类型、指令**和**模块**的子部分。因此，我们在下一节中涵盖的许多内容与二进制格式有直接的对应关系。考虑到这一点，让我们深入了解文本格式。

Wat

文本格式部分提供了对常见语言概念（如值、类型和指令）的技术描述。如果您打算为 WebAssembly 构建工具，这些都是重要的概念，但如果您只打算在应用程序中使用它，则不是必需的。话虽如此，文本格式是 WebAssembly 的重要部分，因此有一些概念您应该了解。在本节中，我们将深入了解文本格式的一些细节，并从核心规范中突出重点。

定义和 S 表达式

要理解 Wat，让我们从直接从 WebAssembly 核心规范中提取的描述的第一句开始：

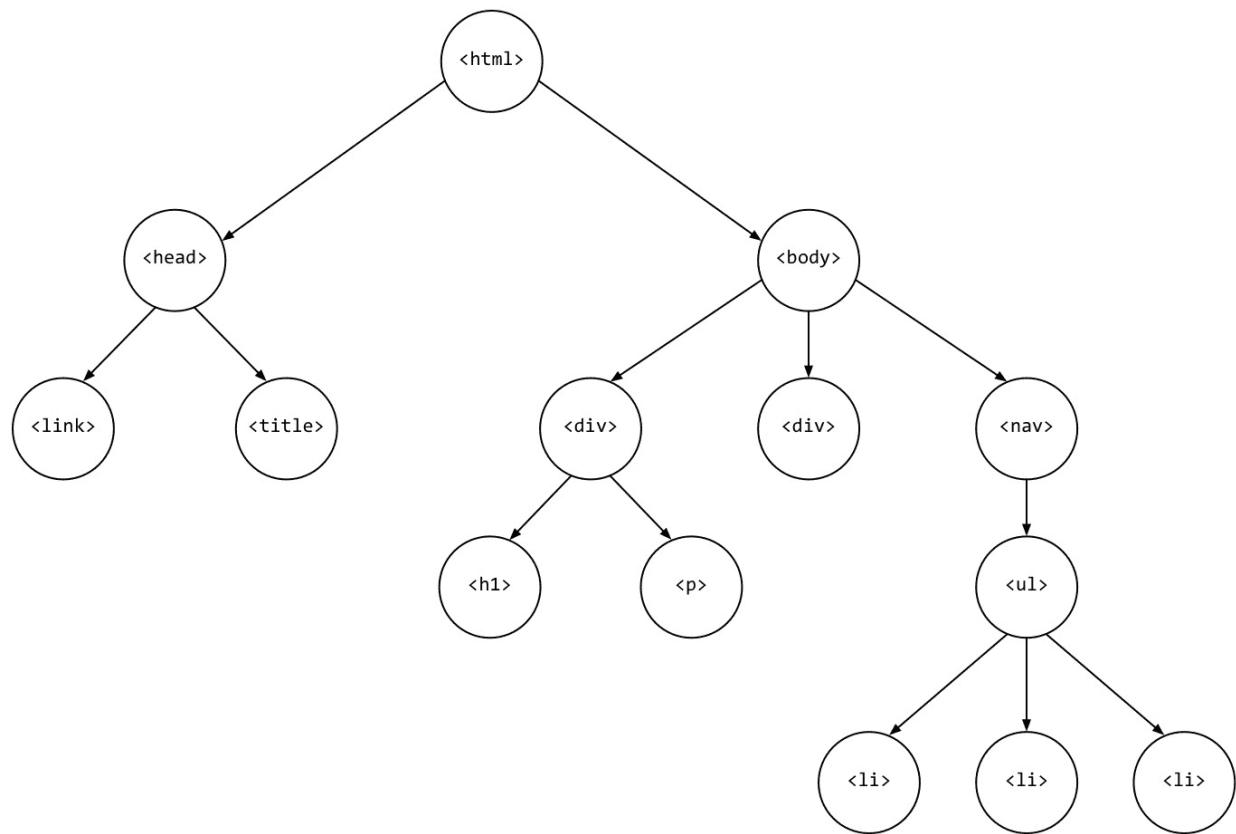
"WebAssembly 模块的文本格式是它们的抽象语法渲染成 S 表达式。"

那么什么是**符号表达式（S 表达式）**？S 表达式是嵌套列表（树形结构）数据的表示。基本上，它们提供了一种在文本形式中表示基于列表的数据的简单而优雅的方式。要理解文本表示的嵌套列表如何映射到树形结构，让我们从 HTML 页面中推断树形结构。以下示例包含一个简单的 HTML 页面和相应的树形结构图。

一个简单的 HTML 页面：

```
<html>
<head>
  <link rel="icon" href="favicon.ico">
  <title>Page Title</title>
</head>
<body>
  <div>
    <h1>Header</h1>
    <p>This is a paragraph.</p>
  </div>
  <div>Some content</div>
  <nav>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
  </nav>
</body>
</html>
```

相应的树形结构是：



HTML 页面的树形结构图

即使你以前从未见过树形结构，也很容易看出 HTML 如何在结构和层次结构方面映射到树形结构。映射 HTML 元素相对简单，因为它是一种具有明确定义标签且没有实际逻辑的标记语言。

Wat 表示可以具有多个具有不同参数的函数的模块。为了演示源代码、Wat 和相应的树结构之间的关系，让我们从一个简单的 C 函数开始，该函数将 2 添加到作为参数传入的数字中：

这是一个将 `2` 添加到传入的 `num` 参数并返回结果的 C 函数：

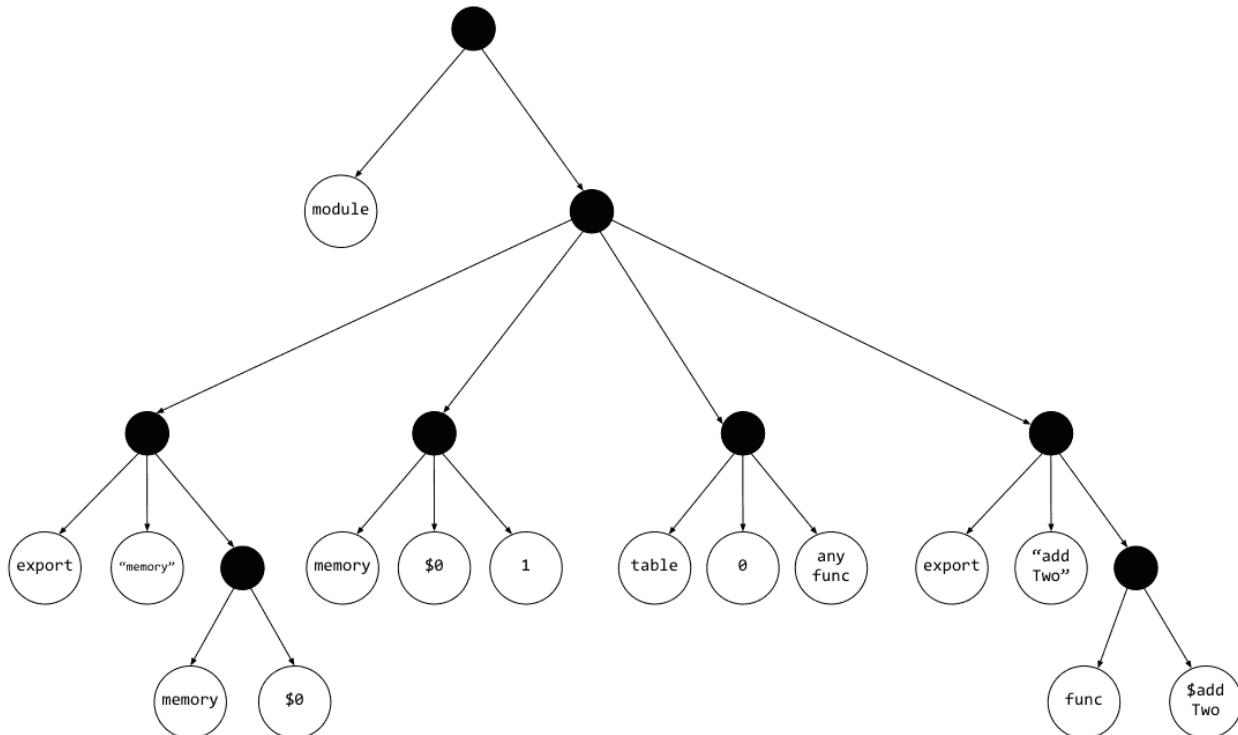
```
int addTwo(int num) {
    return num + 2;
}
```

将 `addTwo` 函数转换为有效的 Wat 会产生以下结果：

```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "addTwo" (func $addTwo))
  (func $addTwo (; 0 ;) (param $0 i32) (result i32)
    (i32.add
      (get_local $0)
      (i32.const 2)
    )
  )
)
```

在第一章中，什么是 WebAssembly？，我们谈到了与核心规范相关的语言概念（函数、线性内存、表等）。在该规范中，结构部分在抽象语法的上下文中定义了每个这些概念。规范的文本格式部分也与这些概念对应，您可以在前面的片段中通过它们的关键字来定义它们（`func`、`memory`、`table`）。

树结构：



Wat 的树结构图

整个树太大，无法放在一页上，因此此图表仅限于 Wat 源文本的前五行。每个填充的点代表一个列表节点（或一组括号的内容）。正如您所看到的，用 s 表达式编写的代码可以以树结构清晰简洁地表达，这就是为什么 s 表达式被选择为 WebAssembly 的文本格式的原因。

值、类型和指令

尽管详细覆盖核心规范的文本格式部分超出了本文的范围，但值得演示一些语言概念如何映射到相应的 Wat。以下图表演示了这些映射在一个样本 Wat 片段中。这是从 C 代码编译而来的，表示一个以单词作为参数并返回字符数的平方根的函数：

Block Comments are surrounded with a <code>(;</code> and <code>;</code>).	<pre>(; ; This is a block comment. ; It allows you to create a multi-line comment without having to ; use a double-semicolon in front of each line. ;)</pre>
table is a Table Type .	<pre>(module (type \$FUNCSIG\$dd (func (param f64) (result f64))) (table \$0 anyfunc) (memory \$0 1))</pre>
16 is an Integer Value .	<pre>(data (i32.const 16) "Test\n00") "Test\n00" is a String Value. (data (i32.const 24) "\10\00\00\00")</pre>
"memory" is an Name Value .	<pre>(export "memory" (memory \$0)) \$0 is an Identifier Value. (export "getCharCountSqrt" (func \$getCharCountSqrt)) (func \$getCharCountSqrt (param \$0 i32) (result f32))</pre>
func , param , and result are Function Types .	<pre>(local \$1 i32) (local \$2 i32) i32 is a Value Type. (local \$3 f64)</pre>
Line Comments start with a <code>;;</code> .	<pre>;; This is a line comment.</pre>
block is a Block Control Instruction .	<pre>(block \$label\$0 \$label\$1 is a Label Identifier.)</pre>
br_if is a Plain Control Instruction .	<pre>(br_if \$label\$1 (i32.eqz (i32.load8_u) load8_u is a Memory Instruction.)</pre>
get_local is a Variable Instruction .	<pre>(get_local \$0))) (set_local \$0 (i32.add (get_local \$0))</pre>
i32.const is a Numeric Instruction .	<pre>(i32.const 1))</pre>

具有语言概念细节的 Wat 示例

如果您打算编写或编辑 Wat，请注意它支持块和行注释。指令被分成块，并包括设置和获取与有效类型相关联的变量的内存。您可以使用 `if` 语句控制逻辑流，并且使用 `loop` 关键字支持循环。

在开发过程中的作用

文本格式允许以文本形式表示二进制 Wasm 模块。这对于开发和调试的便利性有一些深远的影响。拥有 WebAssembly 模块的文本表示允许开发人员在浏览器中查看加载模块的源代码，从而消除了抑制 NaCl 采用的黑匣子问题。它还允许围绕故障排除模块构建工具。官方网站描述了驱动文本格式设计的用例：

- 在 WebAssembly 模块上查看源代码，从而自然地适应 Web（其中可以查看每个源代码）。
- 在没有源映射的情况下，在浏览器开发工具中呈现（这在最小可行产品（MVP）的情况下是必然的）。
- 直接编写 WebAssembly 代码的原因包括教学、实验、调试、优化和测试规范本身。

列表中的最后一项反映了文本格式并不打算在正常开发过程中手动编写，而是从诸如 Emscripten 之类的工具生成。在生成模块时，您可能不会看到或操作任何 `.wat` 文件，但在调试上下文中可能会查看它们。

文本格式不仅在调试方面有价值，而且具有这种中间格式可以减少对单个编译工具的依赖。目前存在多种不同的工具来消耗和发出这种 s 表达式语法，其中一些工具被 Emscripten 用于将您的代码编译成 `.wasm` 文件。

二进制格式和模块文件（Wasm）

二进制格式部分的核心规范提供了与文本格式部分相同级别的语言概念细节。在本节中，我们将简要介绍二进制格式的一些高级细节，并讨论构成 Wasm 模块的各个部分。

定义和模块概述

二进制格式被定义为抽象语法的密集线性编码。不要过于技术化，这基本上意味着它是一种高效的二进制形式，可以快速解码，文件大小小，内存使用减少。二进制格式的文件表示是 `.wasm` 文件，这将是 Emscripten 的编译输出，我们将用于示例。

值、类型和指令子部分在二进制格式的核心规范中与文本格式部分直接相关。每个概念都在编码的上下文中进行了介绍。例如，根据规范，整数类型使用 LEB128 可变长度整数编码进行编码，可以是无符号或有符号变体。如果您希望为 WebAssembly 开发工具，这些都是重要的细节，但如果您只打算在网站上使用它，则不是必需的。

结构、二进制格式和文本格式（wat）部分的核心规范都有一个模块子部分。我们在上一节中没有涵盖模块的方面，因为在二进制的上下文中描述它们更为谨慎。官方的

WebAssembly 网站为模块提供了以下描述：

"WebAssembly 中的可分发、可加载和可执行的代码单元称为**模块**。在运行时，可以使用一组导入值对模块进行**实例化**，以产生一个**实例**，它是一个不可变的元组，引用了运行模块可访问的所有状态。"

我们将在本章后面讨论如何使用 JavaScript 和 Web API 与模块进行交互，因此让我们建立一些上下文，以了解模块元素如何映射到 API 方法。

模块部分

一个模块由几个部分组成，其中一些您将通过 JavaScript API 进行交互：

- 导入（`import`）是可以在模块内访问的元素，可以是以下之一：
 - 函数，可以在模块内使用 `call` 运算符调用
 - 全局变量，可以通过 `global` 运算符在模块内访问
 - 线性内存，可以通过 `memory` 运算符在模块内访问
 - 表，可以通过 `call_indirect` 在模块内访问
- 导出（`export`）是可以由消费 API（即由 JavaScript 函数调用）访问的元素
- 模块启动函数（`start`）在模块实例初始化后调用
- 全局（`global`）包含全局变量的内部定义
- 线性内存（`memory`）包含具有初始内存大小和可选最大大小的线性内存的内部定义
- 数据（`data`）包含数据段数组，指定给定内存的固定范围的初始内容
- 表（`table`）是一个线性内存，其元素是特定表元素类型的不透明值：
- 在 MVP 中，其主要目的是在 C/C++ 中实现间接函数调用
- 元素（`elements`）是一个允许模块使用任何其他模块中的任何导入或内部定义表的元素进行初始化的部分
- 函数和代码：
 - 函数部分声明了模块中定义的每个内部函数的签名
 - 代码部分包含由函数部分声明的每个函数的函数体

一些关键字（`import`，`export` 等）可能看起来很熟悉；它们出现在前一节的 Wat 文件的内容中。WebAssembly 的组件遵循一个直接对应 API 的逻辑映射（例如，您将 `memory`

和 `table` 实例传递给 JavaScript 的 `WebAssembly.instantiate()` 函数）。您与二进制格式的模块的主要交互将通过这些 API 进行。

JavaScript 和 Web API

除了 *WebAssembly* 核心规范之外，还有两个用于与 *WebAssembly* 模块交互的 API 规范：*WebAssembly JavaScript 接口*（JavaScript API）和 *WebAssembly Web API*。在前面的章节中，我们涵盖了核心规范的相关方面，以便熟悉基础技术。如果您从未阅读过核心规范（或者跳过了本章的前几节），这并不会阻碍您在应用程序中使用 *WebAssembly*。但对于 API 来说情况并非如此，因为它们描述了实例化和与编译后的 Wasm 模块交互所需的方法和接口。在本节中，我们将回顾 Web 和 JavaScript API，并描述如何使用 JavaScript 加载和与 Wasm 模块进行通信。

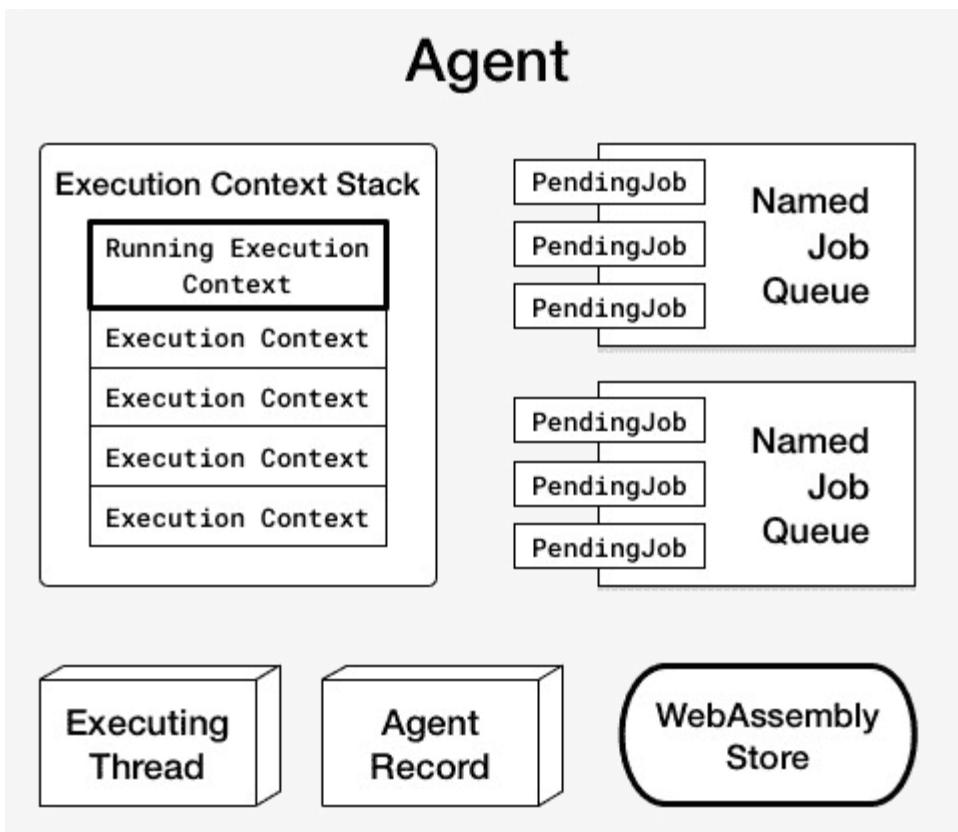
WebAssembly 存储和对象缓存

在深入讨论交互之前，让我们讨论 JavaScript 和 *WebAssembly* 在执行上下文中的关系。核心规范在执行部分包含了以下描述：

"在实例化模块或调用结果模块实例上的导出函数时，将执行 *WebAssembly* 代码。

执行行为是根据模拟程序状态的抽象机器来定义的。它包括一个堆栈，记录操作数值和控制结构，以及包含全局状态的抽象存储。"

在幕后，JavaScript 使用称为**代理**的东西来管理执行。定义中提到的**存储**包含在代理中。以下图表代表了一个 JavaScript 代理：



JavaScript 代理元素

存储表示抽象机器的状态。WebAssembly 操作接受存储并返回更新后的存储。每个代理都与将 JavaScript 对象映射到 WebAssembly 地址的缓存相关联。那么这为什么重要呢？它代表了 WebAssembly 模块与 JavaScript 之间交互的基本方法。JavaScript 对象对应于 *JavaScript API* 中的 WebAssembly 命名空间。考虑到这一点，让我们深入了解接口。

加载模块和 WebAssembly 命名空间方法

JavaScript API 涵盖了浏览器中全局 `WebAssembly` 对象上可用的各种对象。在讨论这些对象之前，我们将从 `WebAssembly` 对象上可用的方法开始，简要概述它们的预期目的：

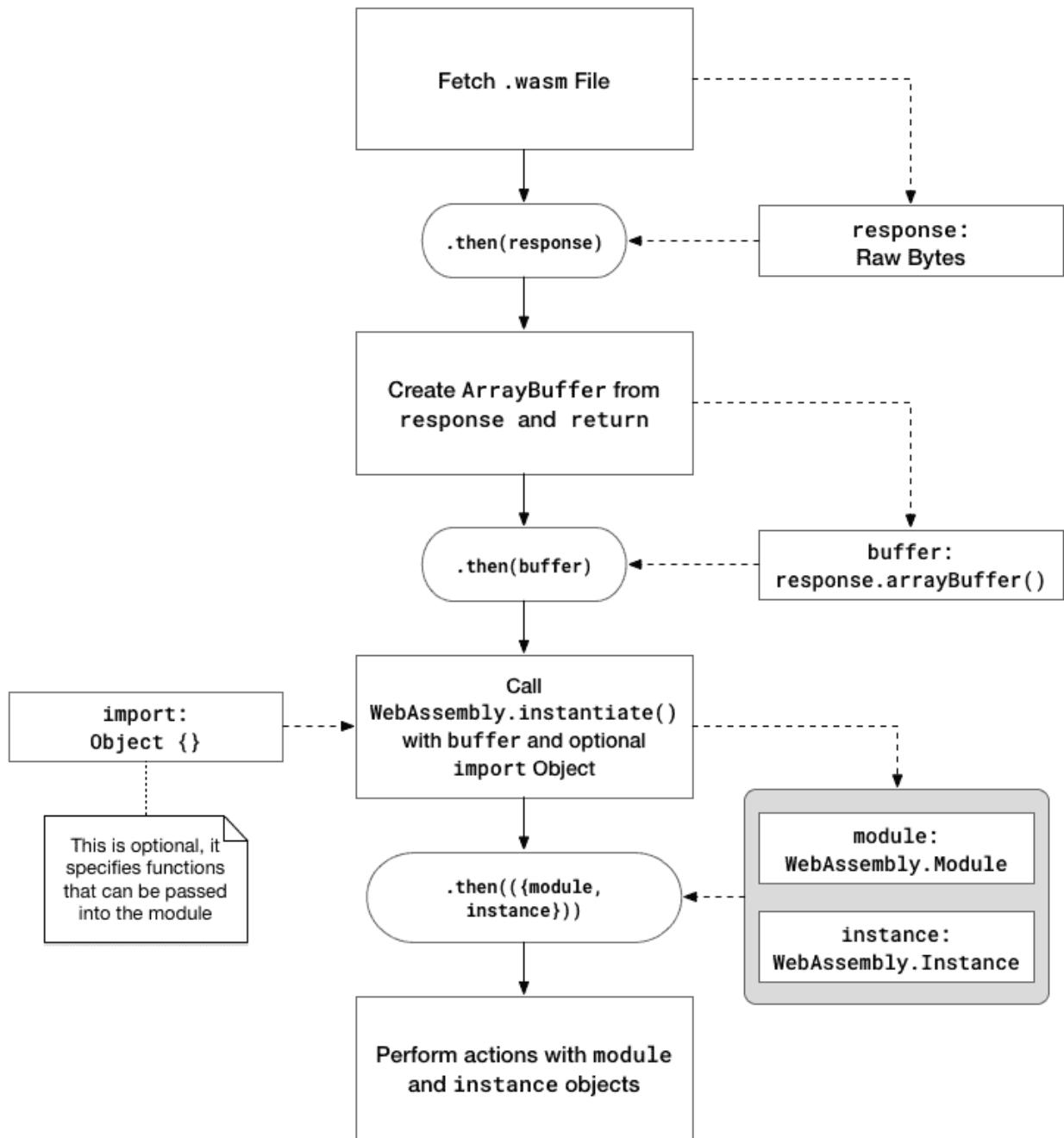
- `instantiate()` 是用于编译和实例化 WebAssembly 代码的主要 API
- `instantiateStreaming()` 执行与 `instantiate()` 相同的功能，但它使用流式处理来编译和实例化模块，从而消除了一个中间步骤
- `compile()` 只编译 WebAssembly 模块，但不实例化它

- `compileStreaming()` 也只编译 WebAssembly 模块，但它使用类似于 `instantiateStreaming()` 的流式处理
- `validate()` 检查 WebAssembly 二进制代码以确保字节有效，并在有效时返回 true，无效时返回 false

`instantiateStreaming()` 和 `compileStreaming()` 方法目前仅存在于 *Web API* 中。事实上，这两种方法构成了整个规范。`WebAssembly` 对象上可用的方法主要用于编译和实例化模块。考虑到这一点，让我们讨论如何获取和实例化一个 Wasm 模块。

当您执行一个 `fetch` 调用来获取一个模块时，它会返回一个 `Promise`，该 `Promise` 解析为该模块的原始字节，这些字节需要加载到一个 `ArrayBuffer` 中并进行实例化。从现在开始，我们将把这个过程称为加载模块。

以下图表展示了这个过程：



获取和加载 WebAssembly 模块

使用 Promises 实际上非常简单。以下代码演示了如何加载一个模块。`importObj` 参数传递任何数据或函数给 Wasm 模块。您现在可以忽略它，因为我们将在第五章中更详细地讨论它，[创建和加载 WebAssembly 模块](#)：

```
fetch('example.wasm')
  .then(response => response.arrayBuffer())
  .then(buffer => WebAssembly.instantiate(buffer, importObj))
  .then(({ module, instance }) => {
    // Do something with module or instance
  });
}
```

上面的示例规定了使用 `instantiate()` 方法加载模块的方法。`instantiateStreaming()` 方法有些不同，并通过一步完成获取、编译和实例化模块来简化这个过程。以下代码使用这种方法实现了相同的目标（加载模块）：

```
WebAssembly.instantiateStreaming(fetch('example.wasm'), importObj)
  .then(({ module, instance }) => {
    // Do something with module or instance
  });
}
```

实例化方法返回一个 Promise，该 Promise 解析为一个包含编译的 `WebAssembly.Module` (`module`) 和 `WebAssembly.Instance` (`instance`) 的对象，这两者将在本节后面进行详细介绍。在大多数情况下，您将使用其中一种方法在您的站点上加载 Wasm 模块。实例包含了所有可以从 JavaScript 代码调用的导出的 WebAssembly 函数。

`compile()` 和 `compileStreaming()` 方法返回一个 Promise，该 Promise 只解析为一个编译的 `WebAssembly.Module`。如果您想要在以后编译一个模块并实例化它，这将非常有用。

Mozilla 开发者网络 (MDN)，由 Mozilla 管理的 Web 文档站点，提供了一个示例，其中编译的模块被传递给了一个 Web Worker。

就 `validate()` 方法而言，它的唯一目的是测试作为参数传入的类型数组或 `ArrayBuffer` 是否有效。这将在响应的原始字节加载到 `ArrayBuffer` 后调用。这个方法没有包含在代码示例中，因为尝试实例化或编译无效的 Wasm 模块将抛出 `TypeError` 或 `WebAssembly` 对象上存在的 `Error` 对象之一。我们将在本节后面介绍这些 `Error` 对象。

WebAssembly 对象

除了在加载模块和 `WebAssembly` 命名空间方法部分介绍的方法之外，全局 `WebAssembly` 对象还有子对象，用于与和排查 `WebAssembly` 交互。这些对象直接对应我们在

WebAssembly 二进制和文本格式部分讨论的概念。以下列表包含了这些对象以及它们的定义，这些定义来自 MDN：

- `WebAssembly.Module` 对象包含了已经被浏览器编译的无状态 WebAssembly 代码，可以有效地与 worker 共享，缓存在 `IndexedDB` 中，并且可以被多次实例化
- `WebAssembly.Instance` 对象是 `WebAssembly.Module` 的一个有状态的可执行实例，其中包含了所有导出的 WebAssembly 函数，允许从 JavaScript 调用 WebAssembly 代码
- `WebAssembly.Memory`，在使用构造函数调用时，创建一个新的 `Memory` 对象，它是一个可调整大小的 `ArrayBuffer`，保存着被 WebAssembly `Instance` 访问的内存的原始字节
- `WebAssembly.Table`，在使用构造函数调用时，创建一个给定大小和元素类型的新 `Table` 对象，表示一个 WebAssembly `Table`（存储函数引用）
- `WebAssembly.CompileError` 在使用构造函数调用时，创建一个错误，指示在 WebAssembly 解码或验证过程中发生了问题
- `WebAssembly.LinkError` 在使用构造函数调用时，创建一个错误，指示在模块实例化过程中发生了问题
- `WebAssembly.RuntimeError` 在调用构造函数时创建一个错误，指示 WebAssembly 指定了一个陷阱（例如，发生了堆栈溢出）。

让我们分别深入研究每一个，从 `WebAssembly.Module` 对象开始。

WebAssembly.Module

`WebAssembly.Module` 对象是 `ArrayBuffer` 和实例化模块之间的中间步骤。`compile()` 和 `instantiate()` 方法（以及它们的流式处理对应方法）返回一个解析为模块的 Promise（小写的 `module` 表示已编译的 `Module`）。一个模块也可以通过直接将类型化数组或 `ArrayBuffer` 传递给构造函数来同步创建，但对于大型模块，这是不鼓励的。

`Module` 对象还有三个静态方法：`exports()`、`imports()` 和 `customSections()`。所有三个方法都以模块作为参数，但 `customSections()` 以表示部分名称的字符串作为其第二个参数。自定义部分在 *Core Specification* 的 *Binary Format* 部分中描述，并且旨在用于调试信息或第三方扩展。在大多数情况下，你不需要定义这些。`exports()` 函数在你使用一个你没有创建的 Wasm 模块时很有用，尽管你只能看到每个导出的名称和种类（例如，`function`）。

对于简单的用例，你不会直接处理 `Module` 对象或已编译的模块。大部分交互将在 `Instance` 中进行。

WebAssembly.Instance

`WebAssembly.Instance` 对象是实例化的 WebAssembly 模块，这意味着你可以从中调用导出的 WebAssembly 函数。调用 `instantiate()` 或 `instantiateStreaming()` 会返回一个解析为包含实例的对象的 Promise。你可以通过引用实例的 `export` 属性上函数的名称来调用 WebAssembly 函数。例如，如果一个模块包含一个名为 `sayHello()` 的导出函数，你可以使用 `instance.exports.sayHello()` 来调用该函数。

WebAssembly.Memory

`WebAssembly.Memory` 对象保存了 WebAssembly `Instance` 访问的内存。这个内存可以从 JavaScript 和 WebAssembly 中访问和改变。要创建一个新的 `Memory` 实例，你需要通过 `WebAssembly.Memory()` 构造函数传递一个带有 `initial` 和（可选的）`maximum` 值的对象。这些值以 WebAssembly 页面为单位，其中一个页面是 64KB。通过调用带有表示要增长的 WebAssembly 页面数量的单个参数的 `grow()` 函数来增加内存实例的大小。你也可以通过其 `buffer` 属性访问内存实例中包含的当前缓冲区。

MDN 描述了获取 `WebAssembly.Memory` 对象的两种方法。第一种方法是从 JavaScript 中构造它（`var memory = new WebAssembly.Memory(...)`），而第二种方法是由 WebAssembly 模块导出它。重要的一点是内存可以在 JavaScript 和 WebAssembly 之间轻松传递。

WebAssembly.Table

`WebAssembly.Table` 对象是一个类似数组的结构，用于存储函数引用。与 `WebAssembly.Memory` 一样，`Table` 可以从 JavaScript 和 WebAssembly 中访问和改变。在撰写时，表只能存储函数引用，但随着技术的发展，很可能还可以存储其他实体。

要创建一个新的 `Table` 实例，你需要传递一个带有 `element`、`initial` 和（可选的）`maximum` 值的对象。`element` 成员是一个表示表中存储的值类型的字符串；目前唯一有效的值是 `"anyfunc"`（用于函数）。`initial` 和 `maximum` 值表示 WebAssembly `Table` 中的元素数量。

您可以使用 `length` 属性访问 `Table` 实例中的元素数量。该实例还包括用于操作和查询表中元素的方法。`get()` 方法允许您访问给定索引处的元素，该索引作为参数传递。`set()` 方法允许您将第一个参数指定的索引处的元素设置为第二个参数指定的值（根据前面的说明，仅支持函数）。最后，`grow()` 允许您增加 `Table` 实例（元素数量）的大小，增加的数量作为参数传递。

WebAssembly 错误（CompileError、LinkError、RuntimeError）

JavaScript API 提供了用于创建特定于 WebAssembly 的 `Error` 对象实例的构造函数，但我们不会花太多时间来介绍这些对象。本节开头的对象定义列表描述了每个错误的性质，如果满足指定条件，则可能引发这些错误。这三个错误都可以使用消息、文件名和行号参数（均为可选）进行构造，并且具有与标准 JavaScript `Error` 对象相同的属性和方法。

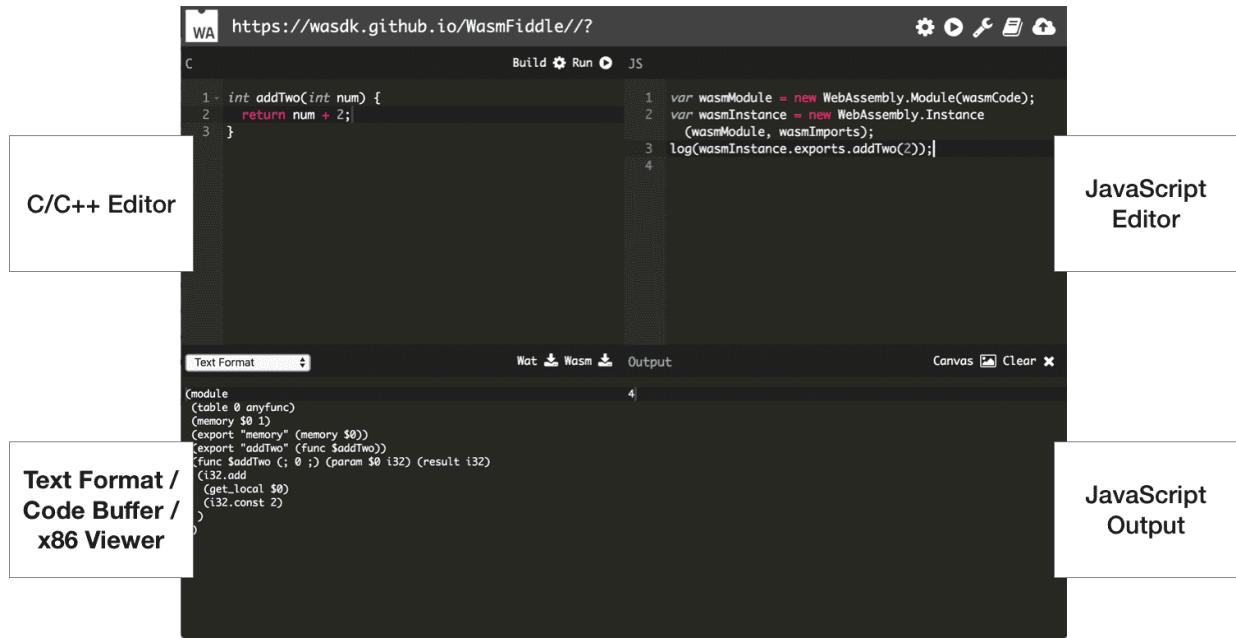
使用 WasmFiddle 连接各个部分

我们在本章中回顾了 WebAssembly 的各个元素以及相应的 JavaScript 和 Web API，但是理解这些元素如何组合在一起仍然可能会令人困惑。随着我们在本书中的示例中的进展，您将能够看到 C/C++、WebAssembly 和 JavaScript 是如何相互交互的，这些概念将变得更加清晰。

话虽如此，演示这种交互可能有助于澄清一些困惑。在本节中，我们将使用一个名为 WasmFiddle 的在线工具来演示这些元素之间的关系，以便您可以看到 WebAssembly 的实际运行情况，并对开发工作流程有一个高层次的概述。

什么是 WasmFiddle？

WasmFiddle 位于 wasdk.github.io/WasmFiddle/，是一个在线代码编辑工具，允许您编写一些 C 或 C++ 代码并将其转换为 Wat，编译为 Wasm，或者直接使用 JavaScript 进行交互。C/C++ 和 JavaScript 编辑器都很简单，不打算用作您的主要开发环境，但它在 Wasm 编译器中提供了有价值的服务。在第三章 设置开发环境中，您将发现从零开始生成 Wasm 文件需要一些工作——能够将您的 C 代码粘贴到浏览器中并点击几个按钮会使事情变得更加方便。以下图表快速概述了界面：



WasmFiddle 用户界面的组件

如您所见，界面相对简单。让我们尝试一些代码！

C 代码转换为 Wat

以下屏幕截图中左上角的窗格包含一个简单的 C 函数，该函数将 2 添加到指定为参数的数字。左下角的窗格包含相应的 Wat：

The screenshot shows the WasmFiddle interface. In the top left, there's a 'C' icon. To the right are 'Build' and 'Run' buttons. Below the code editor, there are 'Text Format' and 'Wat' buttons with download icons, and 'Wasm' buttons with download icons. The code editor contains the following C code:

```
1 int addTwo(int num) {  
2     return num + 2;  
3 }
```

Below the code editor, the generated WebAssembly binary is shown in a monospaced font:

```
(module  
  (table 0 anyfunc)  
  (memory $0 1)  
  (export "memory" (memory $0))  
  (export "addTwo" (func $addTwo))  
  (func $addTwo (; 0 ;) (param $0 i32) (result i32)  
    (i32.add  
      (get_local $0)  
      (i32.const 2)  
    )  
  )  
)
```

C 函数和相应的 Wat

如果这看起来很熟悉，那是因为相同的代码在本章开头对 Wat 的 s 表达式进行了解释时使用过。深入挖掘一下，您可以看到 C 代码如何对应于 Wat 输出。`addTwo()` 函数作为字符串从模块中导出，位于第 5 行。第 5 行还包含 `(func $addTwo)`，它引用了第 6 行上的 `$addTwo` 函数。第 6 行指定可以传入一个 `i32` 类型（整数）的单个参数，并且返回的结果也是 `i32`。在左上角（或 C/C++ 编辑器上方）按下“Build”按钮将把 C 代码编译成 Wasm 文件。一旦构建完成，Wasm 将可以供下载或与 JavaScript 进行交互。

Wasm 到 JavaScript

以下屏幕截图中的右上方窗格包含一些 JavaScript 代码，用于编译在上一步生成的 Wasm。`wasmCode` 是在构建完成时生成的，因此应该自动可用。WasmFiddle 不使用 `instantiate()` 方法，而是创建一个编译后的 `WebAssembly.Module` 实例，并将其传递给新的

`WebAssembly.Instance` 的构造函数。`wasmImports` 对象目前为空，但如果需要，我们可以传入 `WebAssembly.Memory` 和 `WebAssembly.Table` 实例：

The screenshot shows a developer tool interface for WebAssembly. At the top, there are several icons: gear, play, wrench, clipboard, and cloud. Below that, the word "JS" is displayed. A code editor window contains the following JavaScript code:

```
1 var wasmModule = new WebAssembly.Module(wasmCode);
2 var wasmInstance = new WebAssembly.Instance
3     (wasmModule, wasmImports);
4 log(wasmInstance.exports.addTwo(2));|
```

To the right of the code editor, there are three buttons: "Canvas" with a camera icon, "Clear" with a trash bin icon, and a close button. Below the code editor, the word "Output" is displayed. In the output area, the number "4" is shown.

JavaScript 代码调用从编译后的 Wasm 模块中的 C 函数

JavaScript 的最后一行将 `addTwo()` 的结果打印到右下窗格中，当传入数字 2 时。`log()` 方法是一个自定义函数，确保输出打印到右下窗格（数字 4）。请注意 JavaScript 代码如何与 `wasmInstance` 交互。`addTwo()` 函数是从实例的 `exports` 对象中调用的。尽管这是一个人为的例子，但它演示了 C 或 C++ 代码在被 JavaScript 用作 Wasm 模块之前经历的步骤。

总结

在本章中，我们讨论了 WebAssembly 的元素及其关系。核心规范的结构被用来描述文本和二进制格式到一个共同的抽象语法的映射。我们强调了文本格式 (Wat) 在调试和开发环境中的有用性，以及为什么 s 表达式非常适合抽象语法的文本表示。我们还回顾了有关二进制格式和构成模块的各种元素的细节。在 JavaScript 和 Web API 中定义了方

法和对象，并描述了它们在 WebAssembly 交互中的作用。最后，使用 WasmFiddle 工具演示了源代码、Wat 和 JavaScript 之间的关系的简单示例。

在第三章中，设置开发环境，我们将安装开发工具，以便有效地使用 WebAssembly 进行工作。

问题

1. s 表达式擅长表示什么类型的数据？
2. 二进制和文本格式之间共享的四个语言概念是什么？
3. 文本格式的一个用例是什么？
4. 可以存储在 WebAssembly `Table` 中的唯一元素类型是什么？
5. JavaScript 引擎使用什么来管理执行？
6. 哪种方法需要更少的代码来实例化一个模块，`instantiate()` 还是 `instantiateStreaming()`？
7. `WebAssembly` JavaScript 对象上有哪些错误对象，以及是什么事件导致了每一个错误对象？

进一步阅读

- MDN 上的 WebAssembly：developer.mozilla.org/en-US/docs/WebAssembly
- WasmFiddle：wasdk.github.io/WasmFiddle
- 维基百科上的 s 表达式：en.wikipedia.org/wiki/S-expression
- 树的示例：interactivepython.org/runestone/static/pythonds/Trees/ExamplesofTrees.html

第三章：设置开发环境

现在您熟悉了 WebAssembly 的元素，是时候设置一个合适的开发环境了。使用 WebAssembly 进行开发等同于使用 C 或 C++ 进行开发。区别在于构建过程和输出。在本章中，我们将介绍开发工具，并讨论如何在您的系统上安装和配置它们。

本章的目标是了解以下内容：

- 如何安装所需的开发工具（Git、Node.js 和 Visual Studio Code）

- 如何配置 Visual Studio Code 以便使用 C/C++ 和 WebAssembly 扩展
- 如何设置本地 HTTP 服务器来提供 HTML、JavaScript 和 `.wasm` 文件
- 检查浏览器是否支持 WebAssembly
- 有哪些有用的工具可以简化和改进开发过程

安装开发工具

您需要安装一些应用程序和工具来开始开发 WebAssembly。我们将使用文本编辑器 Visual Studio Code 来编写我们的 C/C++、JavaScript、HTML 和 Wat。我们还将使用 Node.js 来提供文件和 Git 来管理我们的代码。我们将使用软件包管理器来安装这些工具，这使得安装过程比手动下载和安装要简单得多。在本节中，我们将涵盖操作系统，以及每个平台的软件包管理器。我们还将简要介绍每个应用程序在开发过程中的作用。

操作系统和硬件

为了确保安装和配置过程顺利进行，重要的是要了解我在本书中使用的操作系统。如果遇到问题，可能是由于您使用的平台与我使用的平台不兼容。在大多数情况下，您不应该遇到问题。为了排除操作系统版本可能导致的问题，我提供了我在下面列表中使用的操作系统的详细信息：

macOS

- High Sierra，版本 10.13.x
- 2.2 GHz 英特尔 i7 处理器
- 16 GB 的 RAM

Ubuntu

- 在 VMware Fusion 中运行的 Ubuntu 16.04 LTS
- 2.2 GHz 英特尔 i7 处理器
- 4 GB 的 RAM

Windows

- Windows 10 Pro 在 VMware Fusion 中运行
- 2.2 GHz 英特尔 i7 处理器

- 8 GB 的 RAM

软件包管理器

软件包管理器是简化软件安装过程的工具。它们允许我们在命令行中升级、配置、卸载和搜索可用软件，而无需访问网站下载和运行安装程序。它们还简化了具有多个依赖项或需要在使用前手动配置的软件的安装过程。在本节中，我将介绍每个平台的软件包管理器。

macOS 的 Homebrew

Homebrew 是 macOS 的一个优秀的软件包管理器，它允许我们直接安装大多数我们将使用的工具。Homebrew 就像在终端中粘贴以下命令并运行它一样简单：

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

您将在终端中看到一些消息，指导您完成安装过程。完成后，您需要安装一个名为 **Homebrew-Cask** 的 Homebrew 扩展，它允许您安装 macOS 应用程序，而无需下载安装程序，挂载它，并将应用程序拖入 `Applications` 文件夹。您可以通过运行以下命令来安装：

```
brew tap caskroom/cask
```

就是这样！现在你可以通过运行以下任一命令来安装应用程序：

```
# For command line tools: brew install <Tool Name>
# For desktop applications:
brew cask install <Application Name>
```

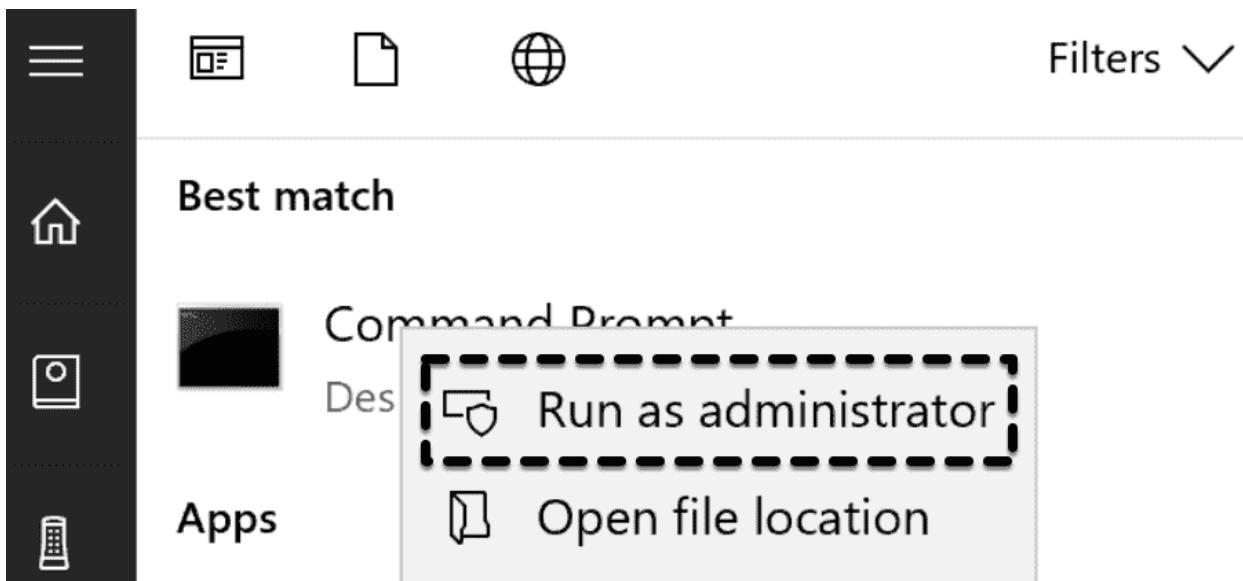
Ubuntu 的 Apt

Apt 是 Ubuntu 提供的软件包管理器；无需安装。它允许您直接安装命令行工具和应用程序。如果 Apt 的存储库中没有某个应用程序，您可以使用以下命令添加存储库：

```
add-apt-repository
```

Windows 的 Chocolatey

Chocolatey 是 Windows 的软件包管理器。它类似于 Apt，可以让您安装命令行工具和应用程序。要安装 Chocolatey，您需要以管理员身份运行命令提示符（cmd.exe）。您可以通过按下开始菜单按钮，输入 cmd，右键单击命令提示符应用程序并选择以管理员身份运行来实现这一点：



以管理员身份运行命令提示符

然后运行以下命令：

```
@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile -InputFormat None -ExecutionPolicy Bypass -Command "iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))" && SET "PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"
```

获取命令文本的最简单方法是通过 Chocolatey 的安装页面 chocolatey.org/install。在使用 cmd.exe 安装部分下有一个按钮可以将文本复制到剪贴板上。您也可以按照安装页面上的步骤使用 PowerShell 来安装应用程序。

Git

Git 是一个版本控制系统（VCS），它允许您跟踪文件的更改并在多个开发人员共同贡献到同一代码库的工作之间进行管理。Git 是 GitHub 和 GitLab 的 VCS 引擎，并且也可在 Bitbucket 上使用（它们还提供 Mercurial，这是另一个 VCS）。Git 将允许我们从

GitHub 克隆存储库，并且是下一章中将要介绍的 EMS DK 的先决条件。在本节中，我们将介绍 Git 的安装过程。

在 macOS 上安装 Git

如果您使用的是 macOS，Git 可能已经可用。macOS 自带了 Apple Git，可能会比最新版本落后几个版本。对于本书的目的，您已经安装的版本应该足够了。如果您希望升级，可以通过在终端中运行以下命令来安装最新版本的 Git：

```
# Install Git to the Homebrew installation folder (/usr/local/bin/git):  
brew install git  
  
# Ensure the default Git is pointing to the Homebrew installation:  
sudo mv /usr/bin/git /usr/bin/git-apple
```

如果运行此命令，您应该会看到 `/usr/local/bin/git`：

```
which git
```

您可以通过运行以下命令来检查安装是否成功：

```
git --version
```

在 Ubuntu 上安装 Git

您可以使用 `apt` 来安装 Git；只需在终端中运行以下命令：

```
sudo apt install git
```

您可以通过运行以下命令来检查安装是否成功：

```
git --version
```

在 Windows 上安装 Git

您可以使用 Chocolatey 来安装 Git。打开命令提示符或 PowerShell 并运行以下命令：

```
choco install git
```

您可以通过运行以下命令来检查安装是否成功：

```
git --version
```

您可以通过在安装命令的末尾添加 `-y` 来绕过确认消息（例如，`choco install git -y`）。您还可以选择始终跳过确认，方法是输入

```
choco feature enable -n allowGlobalConfirmation
```

 命令。

Node.js

Node.js 的官方网站将其描述为一个异步事件驱动的 JavaScript 运行时。Node 旨在构建可扩展的网络应用程序。我们将在本书中使用它来提供我们的文件并在浏览器中处理它们。Node.js 捆绑了 `npm`，这是 JavaScript 的软件包管理器，它将允许我们全局安装软件包并通过命令行访问它们。在本节中，我们将介绍使用**Node 版本管理器 (nvm)** 在每个平台上的安装过程。

nvm

我们将使用 Node.js 的**长期稳定 (LTS)** 版本（版本 8）来确保我们使用平台的最稳定版本。我们将使用 `nvm` 来管理 Node.js 版本。这将防止冲突，如果您已经在计算机上安装了更高（或更低）版本的 Node.js。`nvm` 允许您安装多个 Node.js 版本，并可以快速切换到单个终端窗口的上下文中进行隔离。

在 macOS 上安装 nvm

在终端中运行以下命令：

```
brew install nvm
```

按照 Homebrew 指定的后续安装步骤确保您可以开始使用它（您可能需要重新启动终端会话）。如果在执行步骤之前清除了终端内容，您可以运行此命令再次查看安装步骤：

```
brew info nvm
```

您可以通过运行以下命令来检查安装是否成功：

```
nvm --version
```

在 Ubuntu 上安装 nvm

Ubuntu 捆绑了 `wget`，它可以帮助使用 HTTP/S 和 FTP/S 协议检索文件。`nvm` 的 GitHub 页面 (github.com/creationix/nvm) 包含使用 `wget` 安装它的以下命令：

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/install.sh | bash
```

安装完成后，重新启动终端以完成安装。您可以通过运行以下命令来检查安装是否成功：

```
nvm --version
```

在 Windows 上安装 nvm

`nvm` 目前不支持 Windows，因此您实际上正在安装一个名为 `nvm-windows` 的不同应用程序。`nvm-windows` 的 GitHub 页面位于 github.com/coreybutler/nvm-windows。一些命令略有不同，但我们运行的安装命令将是相同的。要安装 `nvm-windows`，请打开命令提示符或 PowerShell 并运行此命令：

```
choco install nvm
```

您可以通过运行以下命令来检查安装是否成功：

```
nvm --version
```

使用 nvm 安装 Node.js

安装 `nvm` 后，您需要安装本书中将使用的 Node.js 版本：版本 8.11.1。要安装它，请运行以下命令：

```
nvm install 8.11.1
```

如果您之前没有安装 Node.js 或 `nvm`，它将自动将其设置为默认的 Node.js 安装，因此此命令的输出应为 `v8.11.1`：

```
node --version
```

如果您已安装现有的 Node.js 版本，您可以将 `v8.11.1` 作为默认版本，或者确保在使用本书示例时运行此命令以使用 `v8.11.1`：

```
nvm use 8.11.1
```

您可以在代码所在的文件夹中创建一个名为 `.nvmrc` 的文件，并将其填充为 `v8.11.1`。您可以在此目录中运行 `nvm use`，它将设置版本为 `8.11.1`，而无需指定它。

GNU make 和 rimraf

在 `learn-webassembly` 存储库中，代码示例使用 GNU Make 和 VS Code 的任务功能（我们将在第五章中介绍）来执行整本书中定义的构建任务。GNU Make 是一个非常好的跨平台工具，用于自动化构建过程。您可以在 www.gnu.org/software/make 上阅读更多关于 GNU Make 的信息。让我们回顾每个平台的安装步骤。

macOS 和 Ubuntu 上的 GNU Make

如果您使用的是 macOS 或 Linux，则 GNU `make` 应该已经安装。要验证这一点，请在终端中运行以下命令：

```
make -v
```

如果您看到版本信息，您已经准备好了。跳到安装 `rimraf` 部分。否则，请按照您的平台的 GNU Make 安装说明进行操作。

在 macOS 上安装 GNU Make

要在 macOS 上安装 GNU Make，请从终端运行以下命令：

```
brew install make
```

您可以通过运行以下命令来检查安装是否成功：

```
make -v
```

如果您看到版本信息，请跳到安装 *rimraf* 部分。

在 Ubuntu 上安装 GNU Make

要在 Ubuntu 上安装 GNU Make，请从终端运行以下命令：

```
sudo apt-get install make
```

您可以通过运行以下命令来检查安装是否成功：

```
make -v
```

如果您看到版本信息，请跳到安装 *rimraf* 部分。

在 Windows 上安装 GNU make

您可以使用 Chocolatey 在 Windows 上安装 GNU `make`。打开命令提示符或 PowerShell 并运行以下命令：

```
choco install make
```

您可能需要重新启动 CLI 以使用 `make` 命令。重新启动后，运行以下命令以验证安装：

```
make -v
```

如果您看到版本信息，请继续下一节。如果遇到问题，您可能需要下载并安装 gnuwin32.sourceforge.net/packages/make.htm 上的设置包。

安装 rimraf

在 Makefiles 或 VS Code 任务中定义的一些构建步骤会删除文件或目录。根据您的平台和 shell，删除文件或文件夹所需的命令会有所不同。为了解决这个问题，我们将使用 `rimraf npm` 包 (www.npmjs.com/package/rimraf)。全局安装该包会提供一个 `rimraf` 命令，该命令可以执行适合操作系统和 shell 的正确删除操作。

要安装 `rimraf`，请确保已安装 Node.js，并从 CLI 运行以下命令：

```
npm install -g rimraf
```

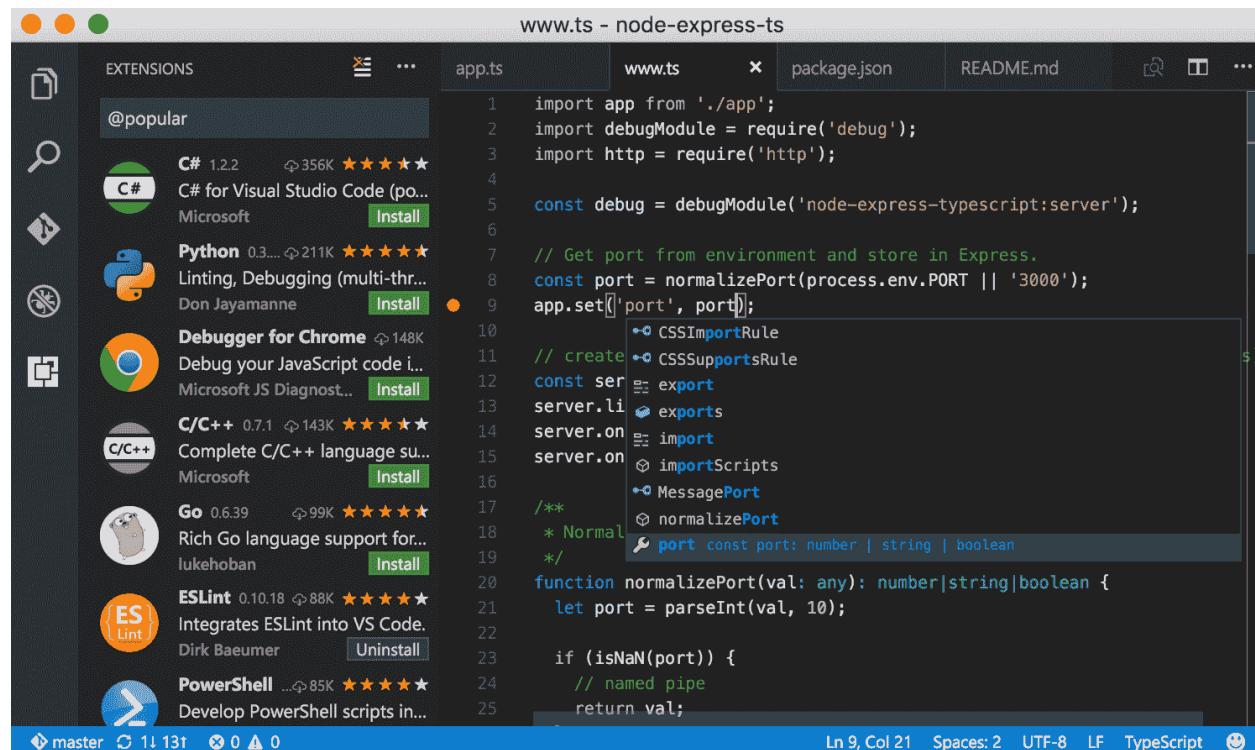
为了确保安装成功，请运行以下命令：

```
rimraf --help
```

您应该看到使用说明和一系列命令行标志。让我们继续进行 VS Code 安装。

VS Code

VS Code 是一个跨平台的文本编辑器，支持多种语言，并拥有丰富的扩展生态系统。集成调试和 Git 支持内置，并且不断添加新功能。我们可以在本书的整个 WebAssembly 开发过程中使用它。在本节中，我们将介绍每个平台的安装步骤：



来自 Visual Studio Code 网站的屏幕截图

在 macOS 上安装 Visual Studio Code

使用 Homebrew-Cask 安装 VS Code。在终端中运行以下命令进行安装：

```
brew cask install visual-studio-code
```

安装完成后，您应该能够从“应用程序”文件夹或 Launchpad 启动它。

在 Ubuntu 上安装 Visual Studio Code

在 Ubuntu 上安装 VS Code 的过程有一些额外的步骤，但仍然相对简单。首先，从 VS Code 的下载页面（code.visualstudio.com/Download）下载 .deb 文件。下载完成后，运行以下命令完成安装：

```
# Change directories to the Downloads folder  
cd ~/Downloads  
  
# Replace <file> with the name of the downloaded file  
sudo dpkg -i <file>.deb  
  
# Complete installation  
sudo apt-get install -f
```

如果出现缺少依赖项错误，您可以在 `sudo dpkg` 之前运行以下命令来解决它：

```
sudo apt-get install libgconf-2-4  
sudo apt --fix-broken install
```

您现在应该能够从启动器中打开 VS Code 了。

在 Windows 上安装 VS Code

您可以使用 Chocolatey 安装 VS Code。从命令提示符或 PowerShell 运行以下命令：

```
choco install visualstudiocode
```

安装后，您可以从“开始”菜单中访问它。

您可以通过在 CLI 中运行 `code .` 来打开当前工作目录的 VS Code。

配置 VS Code

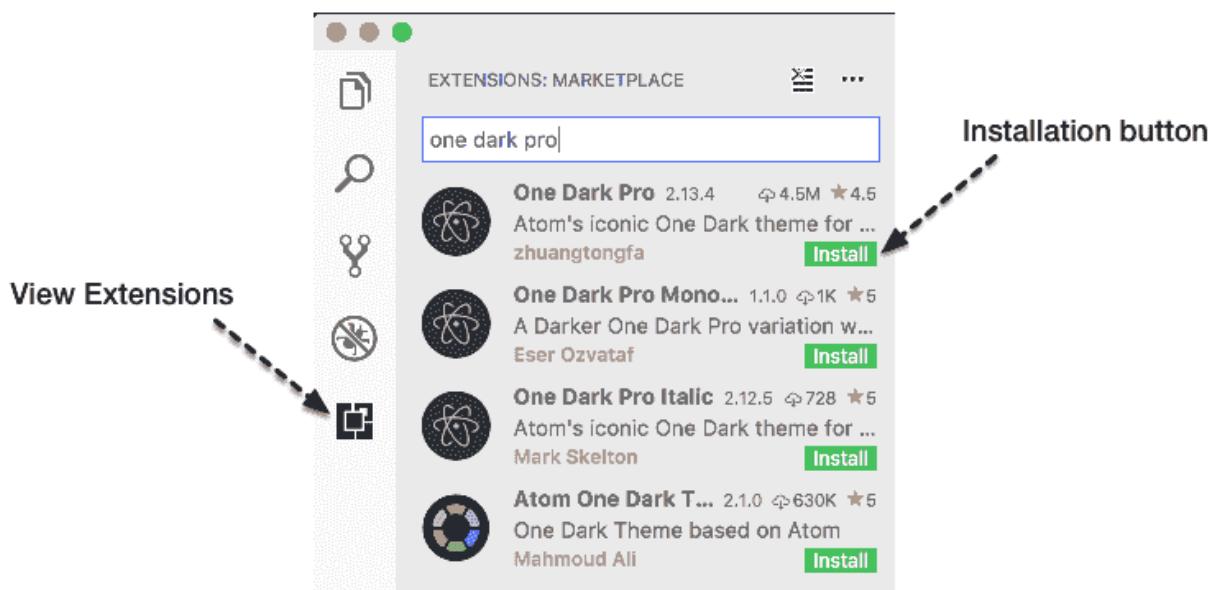
VS Code 是一个功能强大的文本编辑器，具有许多出色的功能。除了高度可配置和可定制之外，它还拥有一个非常丰富的扩展生态系统。我们需要安装其中一些扩展，这样我们就不需要为不同的编程语言使用不同的编辑器。在本节中，我们将介绍如何配置 VS Code 以及安装哪些扩展来简化 WebAssembly 开发过程。

管理设置和自定义

自定义和配置 VS Code 非常简单和直观。您可以通过在 macOS 上选择 Code | Preferences | Settings 或在 Windows 上选择 File | Preferences | Settings 来管理自定义设置，如编辑器字体和选项卡大小。用户和工作区设置分别在 JSON 文件中管理，并且在您无法记住设置的确切名称时提供自动完成。您还可以通过在首选项菜单中选择适当的选项来更改主题或键盘快捷键。设置文件也是您可以为安装的任何扩展设置自定义设置的地方。安装扩展时会默认添加一些设置，因此更改它们就像更新和保存此文件一样简单。

扩展概述

在配置过程中，我们需要安装一些扩展。在 VS Code 中，有多种方式可以查找和安装扩展。我喜欢点击扩展按钮（编辑器左侧活动栏顶部的第四个按钮），在搜索框中输入我要找的内容，然后点击绿色的安装按钮来安装我想要的扩展。你也可以访问 VS Code Marketplace (marketplace.visualstudio.com/vscode)，搜索并选择你想要安装的扩展，然后在扩展页面上点击绿色的安装按钮。你也可以通过命令行来管理扩展。更多信息，请访问 code.visualstudio.com/docs/editor/extension-gallery：



在 VS Code 中安装扩展

C/C++和 WebAssembly 的配置

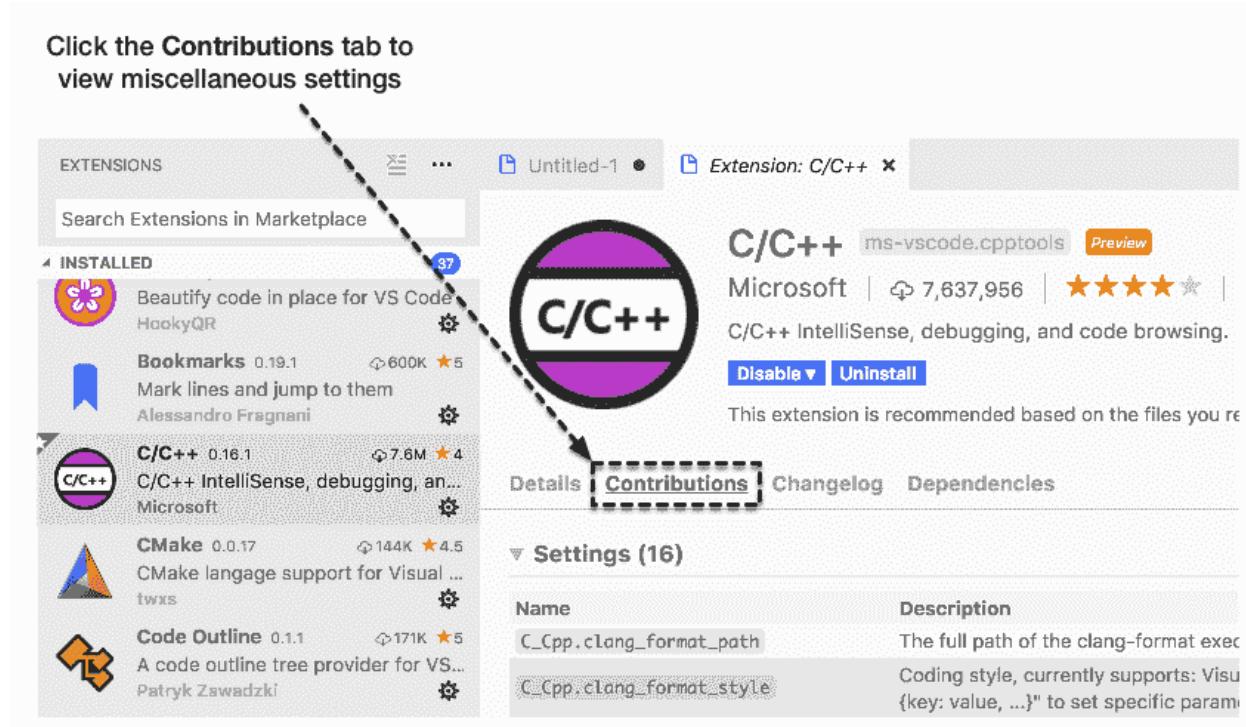
VS Code 默认不支持 C 和 C++，但有一个很好的扩展可以让你使用这些语言。它也不支持 WebAssembly 文本格式的语法高亮，但有一个扩展可以添加这个功能。在本节中，我们将介绍为 VS Code 安装和配置 C/C++和WebAssembly Toolkit for VSCode 扩展。

为 VS Code 安装 C/C++

VS Code 的 C/C++ 扩展包括了一些用于编写和调试 C 和 C++ 代码的功能，比如自动补全、符号搜索、类/方法导航、逐行代码步进等等。要安装这个扩展，可以在扩展中搜索 C/C++ 并安装由微软创建的名为 C/C++ 的扩展，或者访问扩展的官方页面

marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools 并点击绿色的安装按钮。

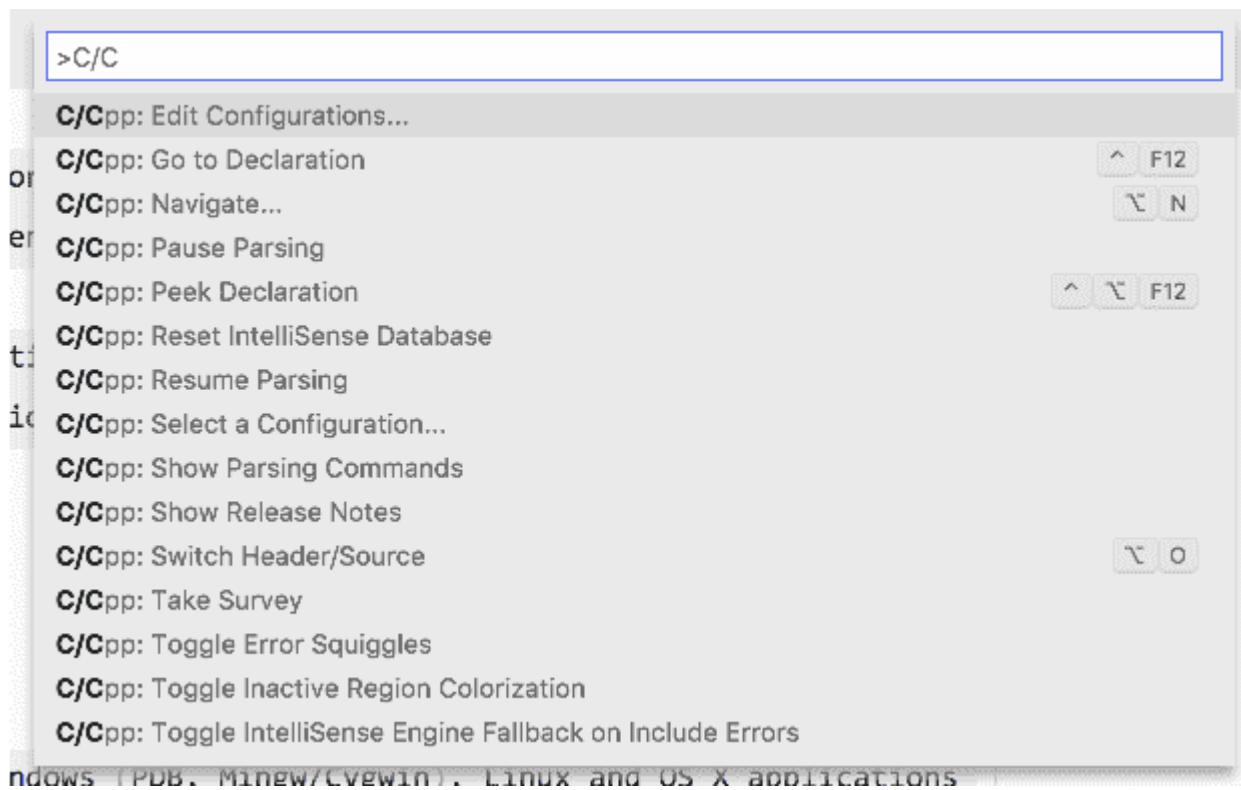
安装完成后，你可以通过在 VS Code 的扩展列表中选择扩展并选择 Contributions 标签来查看扩展的配置细节。这个标签包含了各种设置、命令和调试器的详细信息：



C/C++ 扩展的 Contributions 标签

为 VS Code 配置 C/C++

微软有一个官方页面专门介绍这个扩展，你可以在code.visualstudio.com/docs/languages/cpp上查看。这个页面描述了如何通过使用 JSON 文件进行配置等内容。让我们首先创建一个新的配置文件来管理我们的 C/C++ 环境。你可以通过按下 F1 键，输入 C/C，然后选择 C/Cpp: Edit Configurations... 来生成一个新的配置文件：

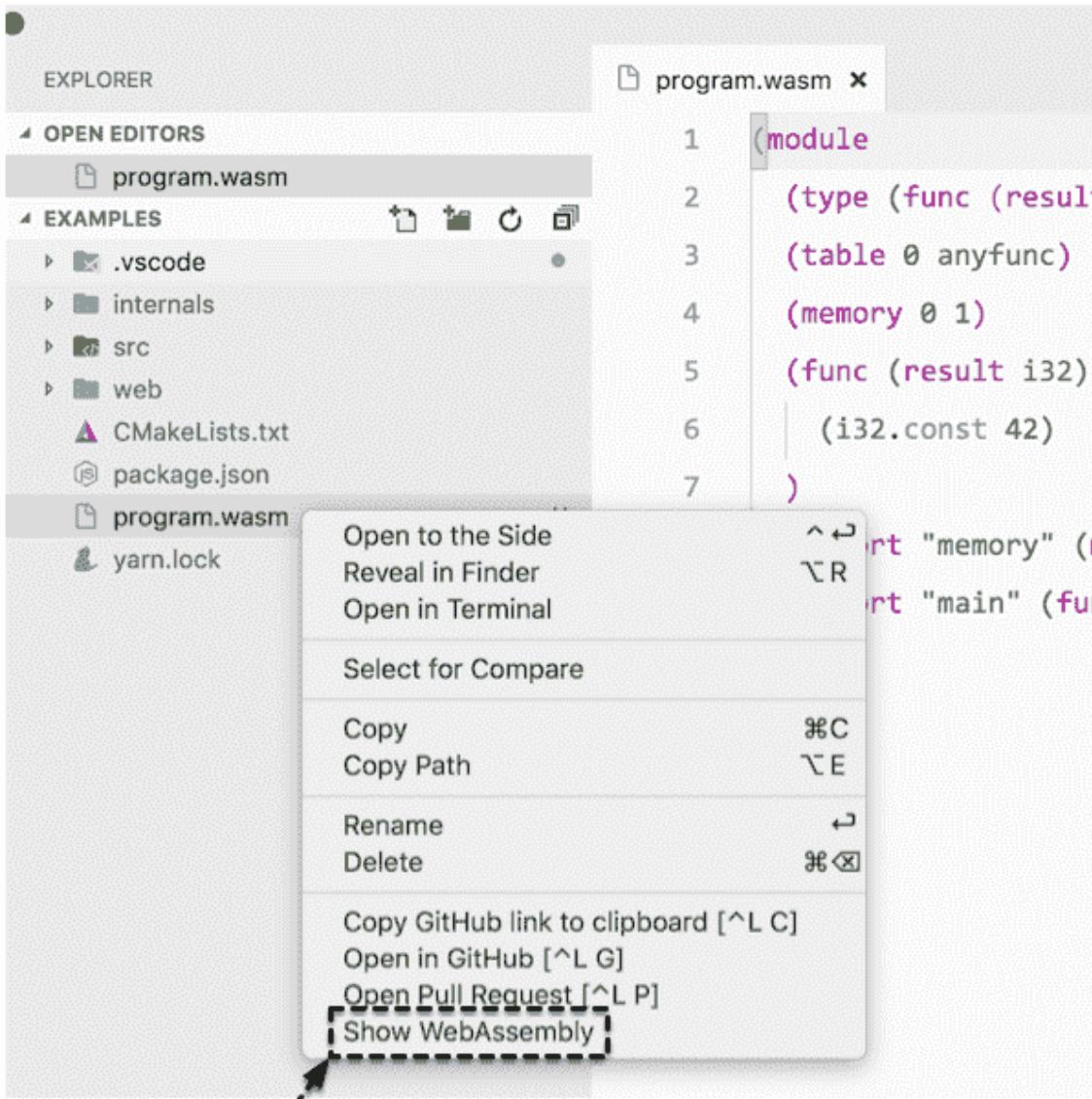


C/C++ 扩展选项的命令面板

这将在当前项目的 `.vscode` 文件夹中生成一个新的 `c_cpp_properties.json` 文件。该文件包含了关于你的 C/C++ 编译器的配置选项，基于你的平台、要使用的 C 和 C++ 标准，以及头文件的包含路径。生成后，你可以关闭这个文件。当我们配置 EMSDK 时，我们会再次访问它。

VSCode 的 WebAssembly 工具包

目前有几种不同的 WebAssembly 扩展可用于 VS Code。我正在使用 VSCode 的 WebAssembly 工具包扩展，因为它允许你右键单击一个 `.wasm` 文件并选择 Show WebAssembly，这样就可以显示文件的 Wat 表示。你可以通过扩展面板（搜索 WebAssembly）或从 VS Code Marketplace 的官方扩展页面（marketplace.visualstudio.com/items?itemName=dtsvet.vscode-wasm）安装这个扩展：



VS Code 有一些很棒的扩展，可以提高效率并自定义界面。在本节中，我将介绍一些我安装的扩展，这些扩展可以简化常见任务以及用户界面/图标主题。您不需要为本书中的示例安装这些扩展，但您可能会发现其中一些有用。

自动重命名标签

在处理 HTML 时，此扩展非常有用。如果更改标记类型，它会自动更改关闭标记的名称。例如，如果您有一个 `<div>` 元素，并且想将其更改为 ``，将打开元素的文本更新为 `span` 将更新关闭元素的文本（`</div>` 更改为 ``）：



自动重命名标签重命名 HTML 标签

括号对颜色器

此扩展为您的代码着色括号，大括号和括号，以便您可以快速识别开放和关闭括号。WebAssembly 的文本格式广泛使用括号，因此能够确定哪些元素包含在哪个列表中，使调试和评估变得更加简单：

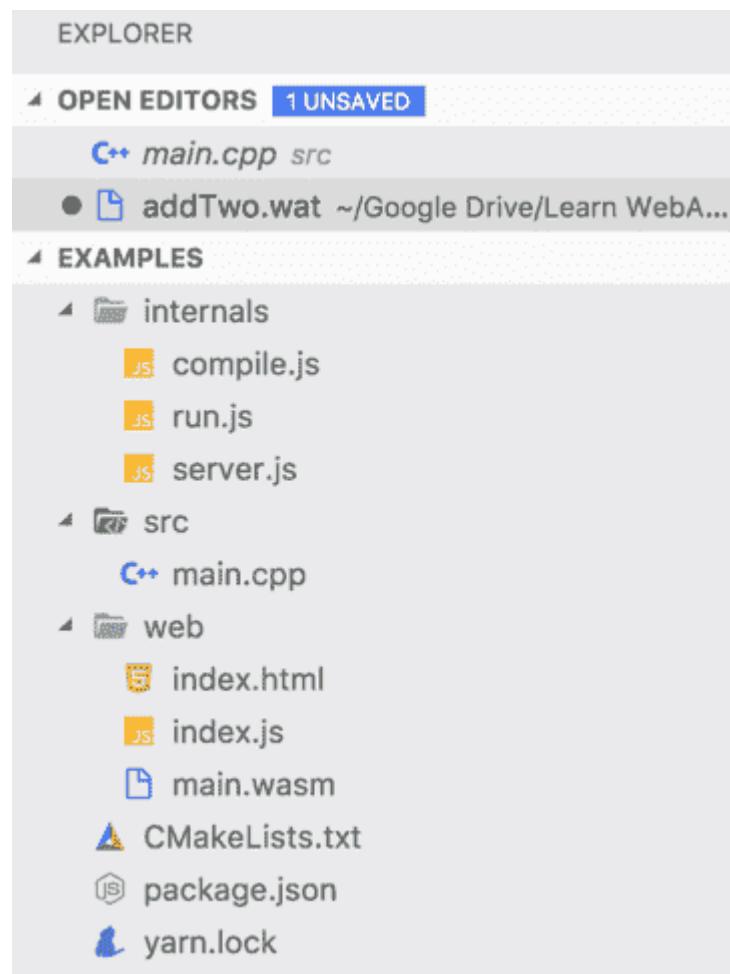
```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "addTwo" (func $addTwo))
  (func $addTwo (; 0 ;) (param $0 i32) (result i32)
    (i32.add
      (get_local $0)
      (i32.const 2))
  )
)
```

在 Wat 文件中匹配括号的括号对颜色器

Material Icon 主题和 Atom One Light 主题

在 VS Code Marketplace 上有超过 1,000 个图标和界面主题可用。我在本节中包括 Material Icon 主题和 Atom One Light 主题，因为它们在本书的截图中被使用。

Material Icon 主题非常受欢迎，已经有超过 200 万次下载，而 Atom One Light 主题已经有超过 70,000 次下载：



Material Icons 主题中的图标

为 Web 设置

与 Wasm 模块交互和调试将在浏览器中进行，这意味着我们需要一种方法来提供包含我们示例文件的文件夹。正如我们在第二章中讨论的那样，*WebAssembly 的元素-Wat，Wasm 和 JavaScript API*，*WebAssembly* 被集成到浏览器的 JavaScript 引擎中，但您需要确保您使用支持它的浏览器。在本节中，我们将提供克隆书籍示例存储库的说明。我们还将回顾如何快速设置本地 Web 服务器以进行测试和评估浏览器选项，以确保您能够在本地开发。

克隆书籍示例存储库

您可能希望现在克隆 GitHub 存储库，其中包含本书中的所有示例。您绝对需要为第七章 *从头开始创建应用程序* 克隆代码，因为应用程序的代码库太大，无法放入单个章节

中。选择硬盘上的一个文件夹，并运行以下命令来克隆存储库：

```
git clone https://github.com/mikerourke/learn-webassembly
```

克隆过程完成后，您会发现示例按章节组织。如果一个章节中有几个示例，它们将按章节文件夹内的子文件夹进行拆分。

如果您使用 Windows，请不要将存储库克隆到 `\Windows` 文件夹或任何其他权限受限的文件夹中。否则，在尝试编译示例时，您将遇到问题。

安装本地服务器

我们将使用一个 `npm` 包 `serve` 来提供文件。要安装，只需运行此命令：

```
npm install -g serve
```

安装完成后，您可以在任何文件夹中提供文件。为了确保它正常工作，让我们尝试提供一个本地文件夹。本节的代码位于 `learn-webassembly` 存储库的 `/chapter-03-dev-env` 文件夹中。按照以下说明验证您的服务器安装：

1. 首先，让我们创建一个包含我们将在本书的其余部分中使用的代码示例的文件夹（示例使用名称 `book-examples`）。
2. 启动 VS Code，并从菜单栏中选择文件 | 打开...（对于 macOS/Linux），以及文件 | 打开文件夹...（对于 Windows）。
3. 接下来，选择文件夹 `book-examples`，然后按打开（或选择文件夹）按钮。
4. 一旦 VS Code 完成加载，右键单击 VS Code 文件资源管理器中的位置，并从菜单中选择新文件夹，命名文件夹为 `chapter-03-dev-env`。
5. 选择 `chapter-03-dev-env` 文件夹，然后按新建文件按钮（或 `Cmd/Ctrl + N`）创建一个新文件。将文件命名为 `index.html`，并填充以下内容：

```
<!doctype html>
<html lang="en-us">
  <title>Test Server</title>
</head>
<body>
  <h1>Test</h1>
```

```
<div>
    This is some text on the main page. Click <a href="stuff.
    html">here</a>
    to check out the stuff page.
</div>
</body>
</html>
```

1. 在 chapter-03-dev-env 文件夹中创建另一个名为 stuff.html 的文件，并填充以下内容：

```
<!doctype html>
<html lang="en-us">
<head>
    <title>Test Server</title>
</head>
<body>
    <h1>Stuff</h1>
    <div>
        This is some text on the stuff page. Click <a href="inde
        x.html">here</a>
        to go back to the index page.
    </div>
</body>
</html>
```

1. 我们将使用 VS Code 的集成终端来提供文件。您可以通过选择 View | Integrated Terminal 来访问此功能，或者使用键盘快捷键 $Ctrl + \text{*}$ ($*$ 是 Esc 键下的反引号键)。加载后，运行此命令来提供工作文件夹：

```
serve -l 8080 chapter-03-dev-env
```

您应该看到以下内容：

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
~/Repositories/learn-webassembly master
λ serve -l 8080 chapter-03-dev-env
```

Serving!

- Local: http://localhost:8080

- On Your Network: http://192.168.1.233:8080

Copied local address to clipboard!

在终端中运行 `serve` 命令的结果

- `-l 8080` 标志告诉 `serve` 在端口 `8080` 上提供文件。第一个链接 (`http://127.0.0.1:8080`) 只能在您的计算机上访问。下面的任何链接都可以用来从本地网络上的另一台计算机访问页面。如果您在浏览器中导航到第一个链接 (`http://127.0.0.1:8080/index.html`)，您应该会看到这个：



Test

This is some text on the main page. Click [here](#) to check out the stuff page.

在 Google Chrome 中提供的测试页面

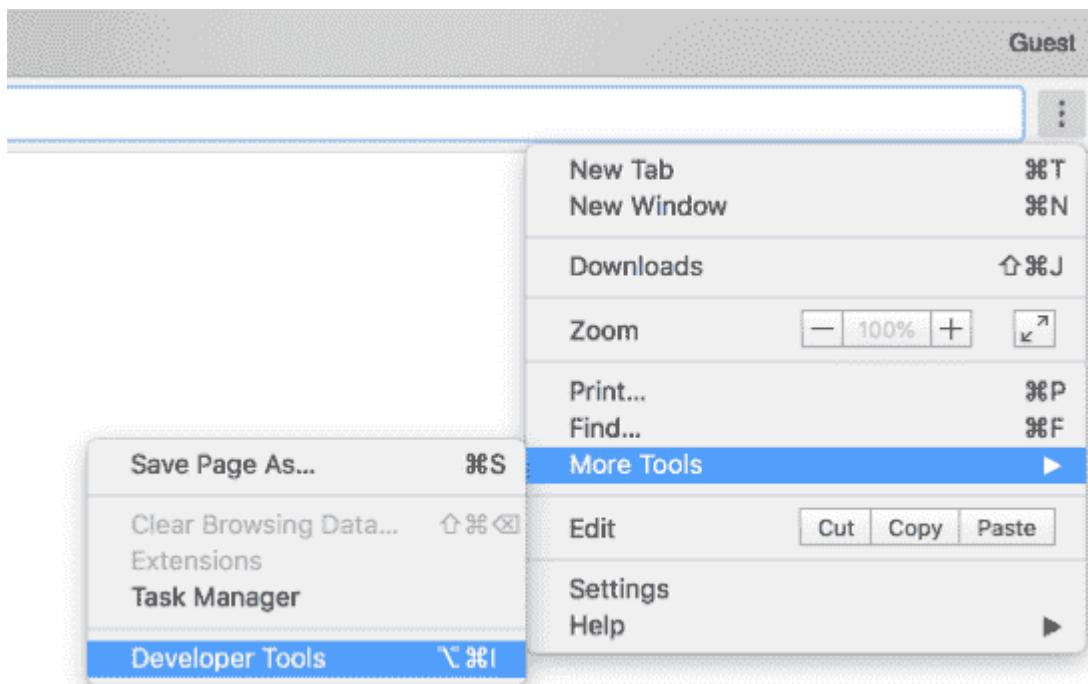
单击此处链接应该将您带到 Stuff 页面（地址栏将显示 `127.0.0.1:8080/stuff.html`）。如果一切正常，现在是验证您的浏览器的时候了。

验证您的浏览器

为了确保您能够在浏览器中测试示例，您需要确保全局存在 `WebAssembly` 对象。为了防止与浏览器兼容性相关的任何问题，我建议您安装 Google Chrome 或 Mozilla Firefox 进行开发。如果您之前安装了这两个浏览器中的任何一个，那么您的浏览器很有可能已经是有效的。为了做到全面，我们仍将介绍验证过程。在本节中，我将回顾您可以采取的步骤，以确保您的浏览器支持 WebAssembly。

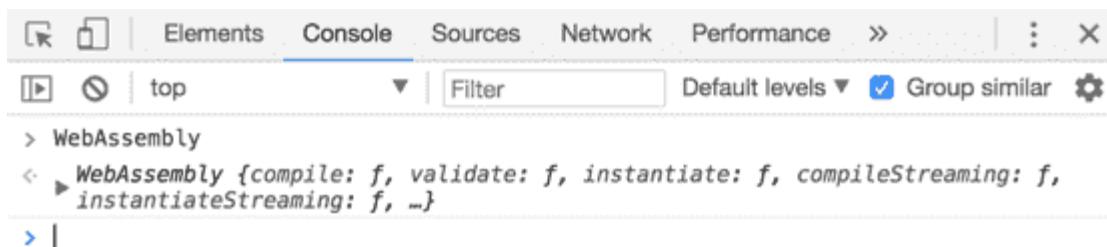
验证 Google Chrome

验证 Chrome 的过程非常简单。选择看起来像三个垂直点的按钮（在地址栏旁边），然后选择**更多工具 | 开发者工具**，或者使用键盘快捷键`Cmd/Ctrl + Shift + I`：



在 Google Chrome 中访问开发者工具

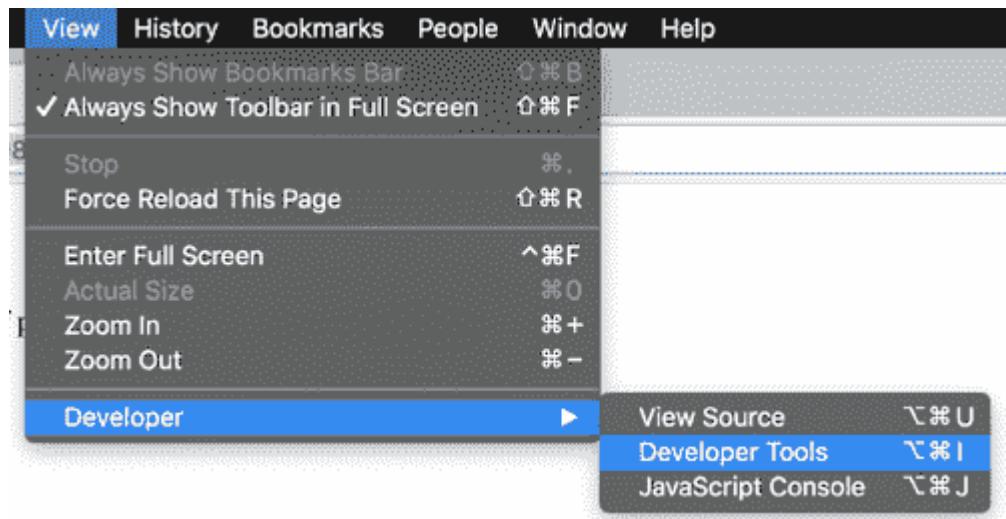
一旦开发者工具窗口出现，选择控制台选项卡，输入 `WebAssembly`，然后按`Enter`。如果您看到这个，您的浏览器是有效的：



在 Google Chrome 的开发者工具控制台中验证 WebAssembly 的结果

验证 Mozilla Firefox

验证 Firefox 的过程与验证 Google Chrome 几乎相同。选择 **工具 | Web 开发者 | 切换工具**，或者使用键盘快捷键 **Cmd/Ctrl + Shift + I**：



在 Mozilla Firefox 中访问开发者工具

选择控制台选项卡，点击命令输入框，输入 `WebAssembly`，然后按 *Enter*。如果您的 Firefox 版本有效，您将看到这个：



在 Mozilla Firefox 中验证 WebAssembly 的结果

验证其他浏览器

其他浏览器的验证过程基本相同；在不同浏览器之间唯一不同的验证方面是如何访问开发者工具。如果 `WebAssembly` 对象可以通过您正在使用的浏览器的控制台访问，您可以使用该浏览器进行 WebAssembly 开发。

其他工具

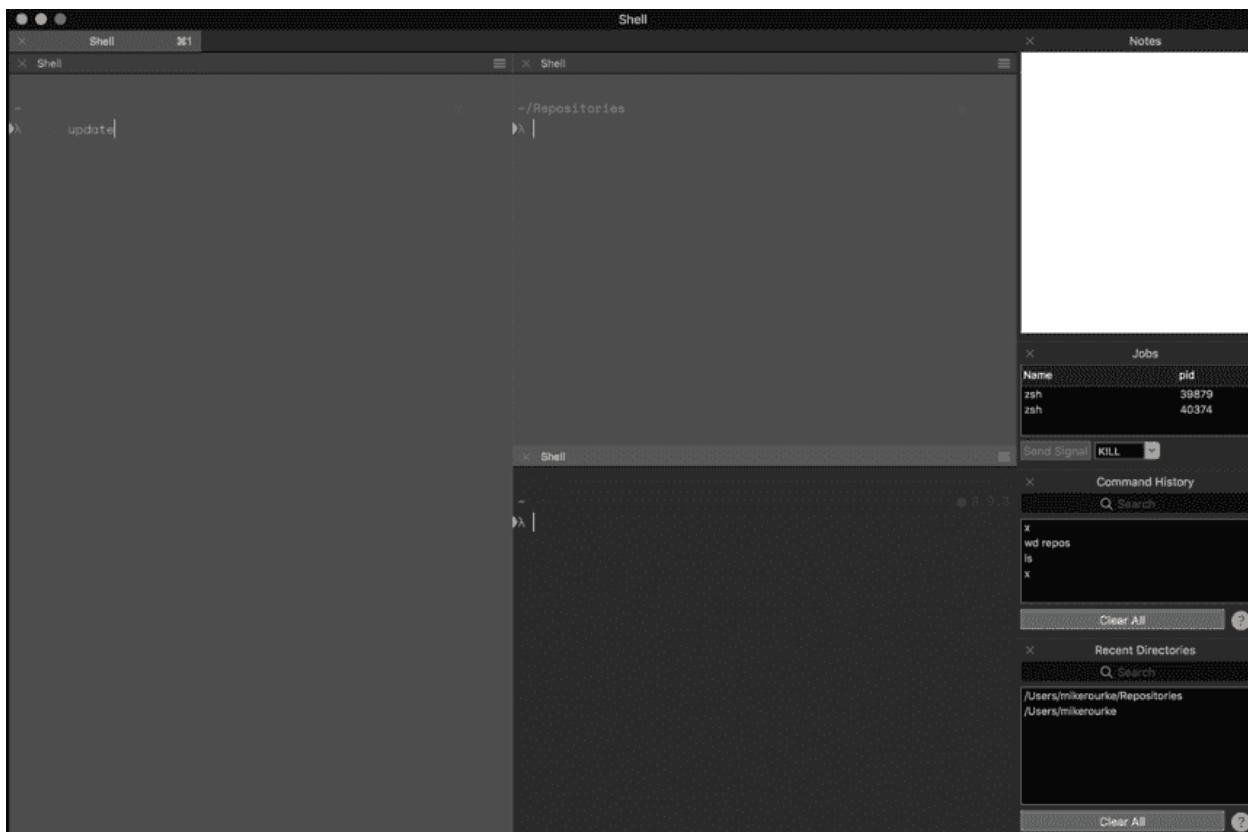
除了我们在前几节中介绍的应用程序和工具之外，还有一些功能丰富且免费的工具可以极大地改善您的开发过程。我没有时间介绍它们所有，但我想强调一下我经常使用的工具。在本节中，我将简要介绍每个平台上可用的一些流行的工具和应用程序。

macOS 的 iTerm2

默认的 macOS 安装包括 Terminal 应用程序，Terminal，这对本书的使用已经足够了。如果您想要一个更全面的终端，iTerm2 是一个很好的选择。它提供诸如分割窗口、广泛的定制、多个配置文件和可以显示笔记、运行作业、命令历史等的工具栏功能。您可以从官方网站(www.iterm2.com/)下载图像文件并手动安装，或者使用 Homebrew-Cask 安装 iTerm，使用以下命令：

```
brew cask install iterm2
```

这是 iTerm2 打开并显示多个编辑器窗口的样子：



具有多个窗格和工具栏的 iTerm 实例

Ubuntu 的 Terminator

Terminator 是 Ubuntu 的 iTerm 和 cmdr，是一个终端仿真器，允许在单个窗口内使用多个选项卡和窗格。Terminator 还提供诸如拖放、查找功能和大量插件和主题等功能。您可以通过 apt 安装 Terminator。为了确保您使用的是最新版本，请在终端中运行以下命令：

```
sudo add-apt-repository ppa:gnome-terminator
sudo apt-get update
sudo apt-get install terminator
```

参考截图：

```

#!/bin/bash 63x20
csatornalistához hozzáadva.
A(z) 703,00 MHz (701,75 - 704,00 MHz) frekvencián csatorn
a található;
csatornalistához hozzáadva.
A(z) 711,25 MHz (710,50 - 712,00 MHz) frekvencián csatorn
a található;
csatornalistához hozzáadva.
A(z) 719,25 MHz (718,50 - 720,00 MHz) frekvencián csatorn
a található;
csatornalistához hozzáadva.
A(z) 735,25 MHz (734,50 - 736,00 MHz) frekvencián csatorn
a található;
csatornalistához hozzáadva.
A(z) 743,25 MHz (742,50 - 744,00 MHz) frekvencián csatorn
a található;
csatornalistához hozzáadva.
A(z) 751,25 MHz (750,25 - 752,00 MHz) frekvencián csatorn
a található;
csatornalistához hozzáadva.
jac@antares ~ $ [redacted]

```

```

#####
#INSTALL
#####

INSTALLATION TUTORIAL ENGLISH:

IS SO EASY:
1:INSTALL ZENITY & MPLAYER & UNRAR & RAR
apt-get install zenity mplayer unrar rar

2:DECOMPRESS a-desk
WITH YOUR FAVORITE DECOMPRESSOR

3:INSTALL XWINWRAP
Enter into zip a-desk and execute xwinwrapcv5.deb or xwinwrap64
.deb (2 clicks or execute by terminal by -> dpkg -i xwinwrapcv5
.deb or xwinwrap64.deb

4:EXECUTE A-DESK.INSTALLER
:|
```

```

mc [jac@antares]:~/gnome2/nautilus-scripts 56x20
top - 18:45:08 up 3:47, 5 users, load average: 0.83,
Tasks: 165 total, 1 running, 163 sleeping, 0 stopped
Cpu(s): 3.7%us, 1.0%sy, 0.0%ni, 94.7%id, 0.0%wa, 0.
Mem: 1026468k total, 970176k used, 56292k free,
Swap: 2096472k total, 1884k used, 2094588k free,
[redacted]

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM
5455	root	20	0	74736	51m	16m	S	3.7	5.2
5737	jac	20	0	64404	4324	3064	S	1.0	0.4
5569	jac	20	0	89592	9656	7488	S	0.3	0.9
7050	jac	20	0	2540	1176	876	R	0.3	0.1
1	root	20	0	2800	1628	1180	S	0.0	0.2
2	root	20	0	0	0	0	S	0.0	0.0
3	root	RT	0	0	0	0	S	0.0	0.0
4	root	20	0	0	0	0	S	0.0	0.0
5	root	RT	0	0	0	0	S	0.0	0.0
6	root	20	0	0	0	0	S	0.0	0.0
7	root	20	0	0	0	0	S	0.0	0.0
8	root	20	0	0	0	0	S	0.0	0.0
9	root	20	0	0	0	0	S	0.0	0.0

从 <http://technicalworldforyou.blogspot.com> 获取的终结者截图

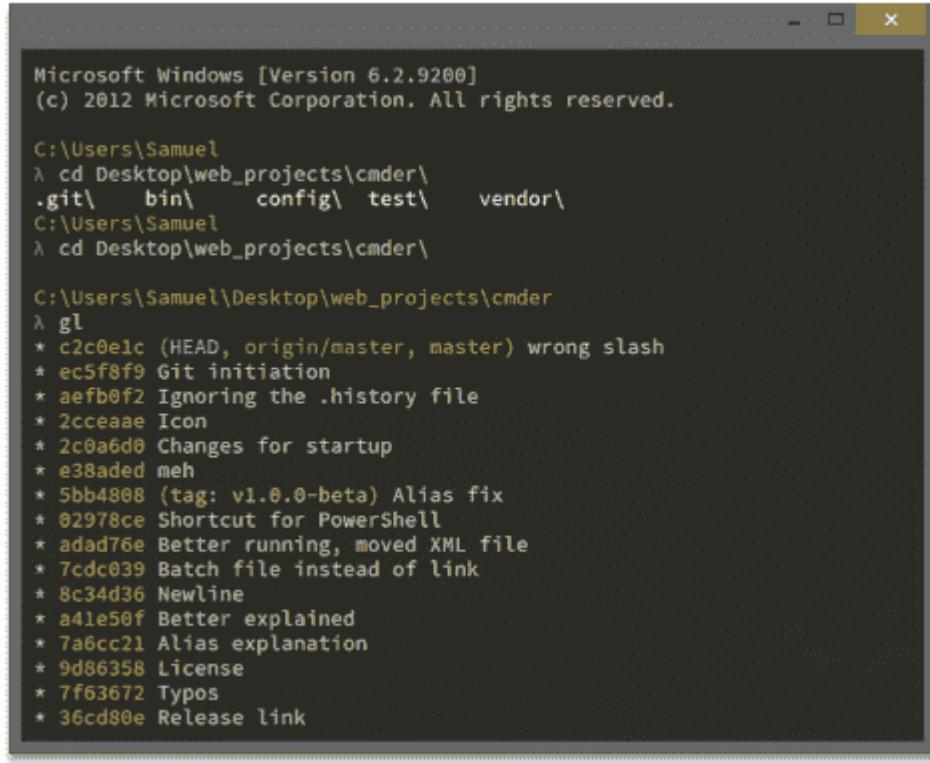
B09984_03_17

Windows 的 cmder

cmder 是 Windows 的控制台仿真器，为标准命令提示符或 PowerShell 添加了许多功能和特性。它提供诸如多个选项卡和可定制性之类的功能。它允许您在同一程序中打开不同外壳的实例。您可以从官方网站(cmder.net)下载并安装它，或者使用以下命令使用 Chocolatey 安装它：

```
choco install cmder
```

这就是它的样子：



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\Samuel
\ cd Desktop\web_projects\cmder\
.\git\ bin\ config\ test\ vendor\
C:\Users\Samuel
\ cd Desktop\web_projects\cmder\

C:\Users\Samuel\Desktop\web_projects\cmder
\ gl
* c2c0e1c (HEAD, origin/master, master) wrong slash
* ec5f8f9 Git initiation
* aefb0f2 Ignoring the .history file
* 2cceaae Icon
* 2c0a6d0 Changes for startup
* e38aded meh
* 5bb4808 (tag: v1.0.0-beta) Alias fix
* 02978ce Shortcut for PowerShell
* adad76e Better running, moved XML file
* 7cdc039 Batch file instead of link
* 8c34d36 Newline
* a41e50f Better explained
* 7a6cc21 Alias explanation
* 9d86358 License
* 7f63672 Typos
* 36cd80e Release link
```

官方网站的 cmder 截图

Zsh 和 Oh-My-Zsh

Zsh 是一个改进了 Bash 的交互式 shell。Oh-My-Zsh 是 Zsh 的配置管理器，具有各种有用的插件。您可以在他们的网站上看到整个列表(github.com/robbyrussell/oh-my-zsh)。例如，如果您想在 CLI 中拥有强大的自动完成和语法高亮功能，可以使用诸如 zsh-autosuggestion 和 zsh-syntax-highlighting 等插件。您可以在 macOS、Linux 和 Windows 上安装和配置 Zsh 和 Oh-My-Zsh。Oh-My-Zsh 页面上有安装说明以及主题和插件列表。

摘要

在本章中，我们介绍了我们将用于开始使用 WebAssembly 进行工作的开发工具的安装和配置过程。我们讨论了如何使用操作系统的软件包管理器（例如 macOS 的 Homebrew）快速轻松地安装 Git、Node.js 和 VS Code。还介绍了配置 VS Code 的步骤以及您可以添加的必需和可选扩展以增强开发体验。我们讨论了如何安装本地 Web 服务器进行测试以及如何验证浏览器以确保支持 WebAssembly。最后，我们简要回顾了一些您可以安装到平台上以帮助开发的其他工具。

在第四章中，安装所需的依赖项，我们将安装所需的依赖项并测试工具链。

问题

1. 你应该使用哪个操作系统的软件包管理器？
2. BitBucket 支持 Git 吗？
3. 为什么我们使用 Node.js 的第 8 个版本而不是最新版本？
4. 你如何在 Visual Studio Code 中更改颜色主题？
5. 你如何访问 Visual Studio Code 中的命令面板？
6. 你如何检查浏览器是否支持 WebAssembly？
7. 其他工具部分中的工具在所有三个操作系统上都受支持吗？

进一步阅读

- Homebrew : [brew.sh](#)
- apt 文档 : [help.ubuntu.com/lts/serverguide/apt.html.en](#)
- Chocolatey : [chocolatey.org](#)
- Git : [git-scm.com](#)
- Node.js : [nodejs.org/en](#)
- GNU Make : [www.gnu.org/software/make](#)
- VS Code : [code.visualstudio.com](#)

第四章：安装所需的依赖项

现在您已经设置好了开发环境，并准备开始编写 C、C++ 和 JavaScript，是时候添加最后一块拼图了。为了从我们的 C/C++ 代码生成 .wasm 文件，我们需要安装和配置 **Emscripten SDK (EMSDK)**。

在本章中，我们将讨论开发工作流程，并谈论 EMSDK 如何融入开发过程。我们将提供详细的说明，说明如何在每个平台上安装和配置 EMSDK，以及任何先决条件。安装和配置过程完成后，您将通过编写和编译一些 C 代码来测试它。

本章的目标是理解以下内容：

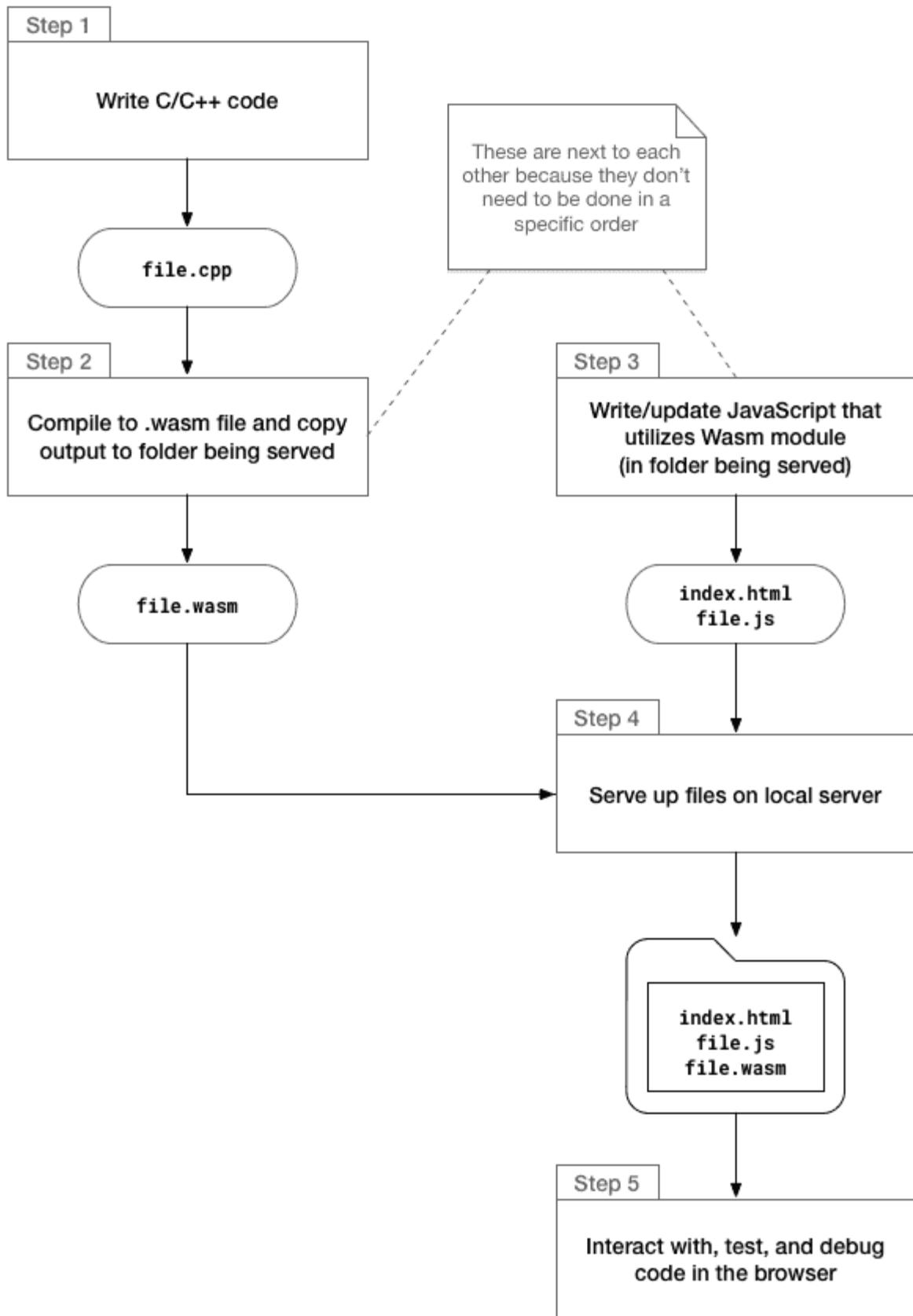
- 与 WebAssembly 一起工作时的整体开发工作流程
- EMSDK 与 Emscripten 和 WebAssembly 的关系以及为什么需要它
- 如何安装 EMSDK 的先决条件
- 如何安装和配置 EMSDK
- 如何测试 EMSDK 以确保它正常工作

开发工作流程

WebAssembly 的开发工作流程与大多数其他需要编译和构建过程的语言类似。在进入工具设置之前，我们将介绍开发周期。在本节中，我们将为本章其余部分将安装和配置的工具建立一些上下文。

工作流程中的步骤

对于本书，我们将编写 C 和 C++ 代码，并将其编译为 Wasm 模块，但这个工作流程适用于任何编译为 `.wasm` 文件的编程语言。以下图表概述了这个过程：



开发工作流程中的步骤

本书中将使用这个过程来进行示例，因此您将了解项目结构如何与工作流程对应。我们将使用一些可用的工具来加快和简化这个过程，但步骤仍将保持不变。

将工具集成到工作流程中

有许多编辑器和工具可用于简化开发过程。幸运的是，C/C++ 和 JavaScript 已经存在了相当长的时间，因此您可以利用最适合您的选项。WebAssembly 的工具列表要短得多，因为这项技术存在的时间较短，但它们确实存在。

我们将使用的主要工具是 VS Code，它提供了一些优秀和有用的功能，可以简化构建和开发过程。除了用它来编写我们的代码外，我们还将利用 VS Code 内置的任务功能从 C/C++ 构建 `.wasm` 文件。通过在项目根文件夹中创建一个 `.vscode/tasks.json` 文件，我们可以指定与构建步骤相关的所有参数，并使用键盘快捷键快速运行它。除了执行构建之外，我们还可以启动和停止运行的 Node.js 进程（即工作流程图中的本地服务器）。我们将在下一章中介绍如何添加和配置这些功能。

Emscripten 和 EMSDK

我们将使用 Emscripten 将我们的 C/C++ 代码编译为 `.wasm` 文件。到目前为止，Emscripten 只是在一般情况下简要提到过。由于我们将在构建过程中使用这个工具和相应的 Emscripten SDK（EMSDK），因此了解每种技术的作用以及它在开发工作流程中的作用是很重要的。在本节中，我们将描述 Emscripten 的目的，并讨论它与 EMSDK 的关系。

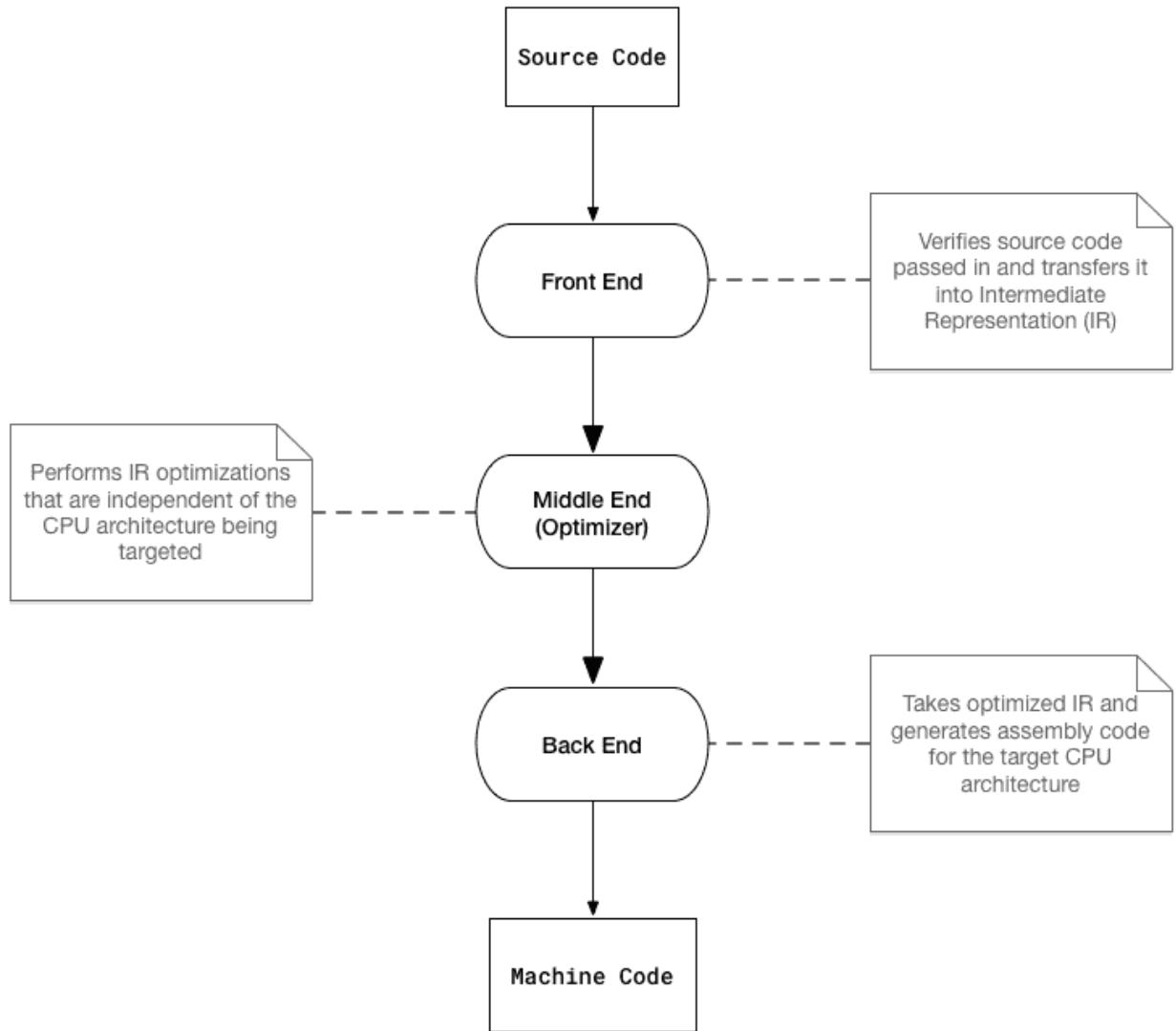
Emscripten 概述

那么 Emscripten 是什么？维基百科提供了以下定义：

“Emscripten 是一个源到源编译器，作为 LLVM 编译器的后端运行，并生成称为 asm.js 的 JavaScript 子集。它也可以生成 WebAssembly。”

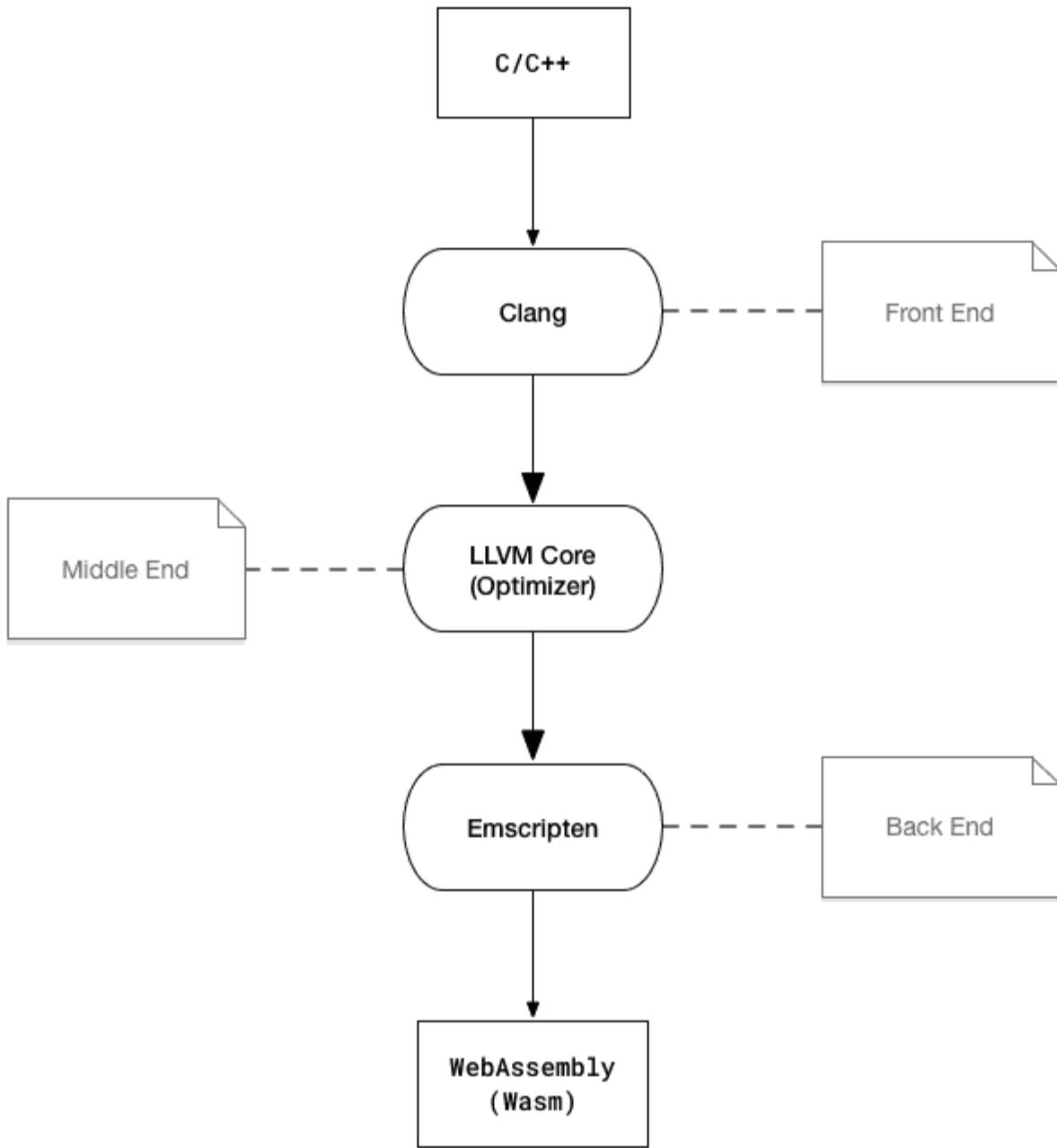
我们在第一章中讨论了源到源编译器（或转换器），并以 TypeScript 为例。转换器将一种编程语言的源代码转换为另一种编程语言的等效源代码。为了详细说明 Emscripten 作为 LLVM 编译器的后端运行，我们需要提供有关 LLVM 的一些额外细节。

LLVM 的官方网站（llvm.org）将 LLVM 定义为一组模块化和可重用的编译器和工具链技术。LLVM 由几个子项目组成，但我们将重点放在 Emscripten 使用的两个项目上：Clang 和 LLVM 核心库。为了了解这些部件如何组合在一起，让我们回顾一下三阶段编译器的设计：



通用三阶段编译器的设计

该过程相对简单：三个独立的阶段或端处理编译过程。这种设计允许不同的前端和后端用于各种编程语言和目标架构，并通过使用中间表示将机器代码与源代码完全解耦。现在让我们将每个编译阶段与我们将用于生成 WebAssembly 的工具链的组件相关联：

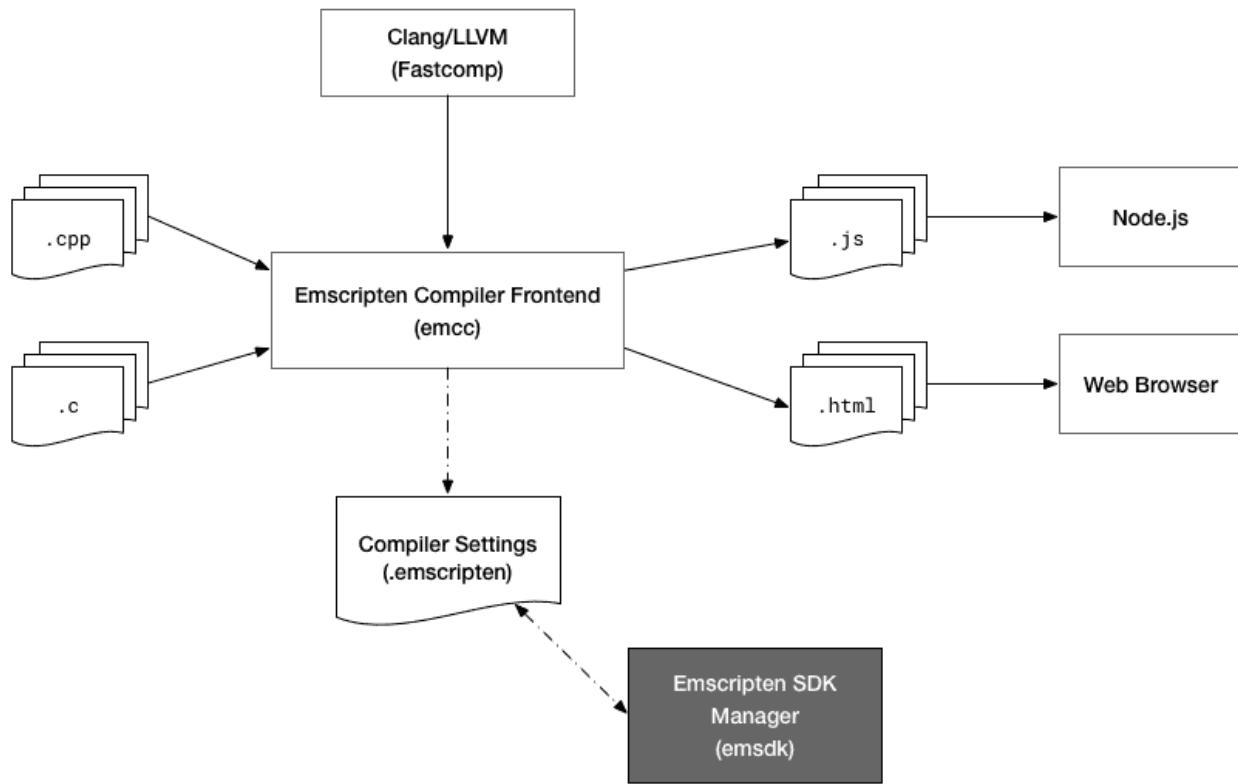


使用 LLVM、Clang 和 Emscripten 的三阶段编译

Clang 用于将 C/C++ 编译为 LLVM 的 **中间表示 (IR)**，Emscripten 将其编译为 Wasm 模块（二进制格式）。这两个图表还展示了 Wasm 和机器代码之间的关系。您可以将 WebAssembly 视为浏览器中的 CPU，Wasm 是其运行的机器代码。

EMSDK 适用于哪里？

Emscripten 是指用于将 C 和 C++ 编译为 `asm.js` 或 WebAssembly 的工具链。EMSDK 用于管理工具链中的工具和相应的配置。这消除了复杂的环境设置需求，并防止了工具版本不兼容的问题。通过安装 EMSDK，我们拥有了使用 Emscripten 编译器所需的所有工具（除了先决条件）。以下图表是 Emscripten 工具链的可视化表示（EMSDK 显示为深灰色）：



Emscripten 工具链（从 emscripten.org 稍作修改）

现在您对 Emscripten 和 EMSDK 有了更好的了解，让我们继续安装先决条件的过程。

安装先决条件

在安装和配置 EMSDK 之前，我们需要安装一些先决条件。您在第三章中安装了两个先决条件：Node.js 和 Git。每个平台都有略有不同的安装过程和工具要求。在本节中，我们将介绍每个平台的先决条件工具的安装过程。

常见的先决条件

您可能已经安装了所有的先决条件。以下是无论平台如何都需要的三个先决条件：

- Git

- Node.js
- Python 2.7

注意 Python 版本；这很重要，因为安装错误的版本可能会导致安装过程失败。如果您在第二章中跟随并安装了 Node.js 和 Git，那么剩下的就是安装 Python 2.7 和为您的平台指定的任何其他先决条件。每个平台的 Python 安装过程将在以下子节中指定。

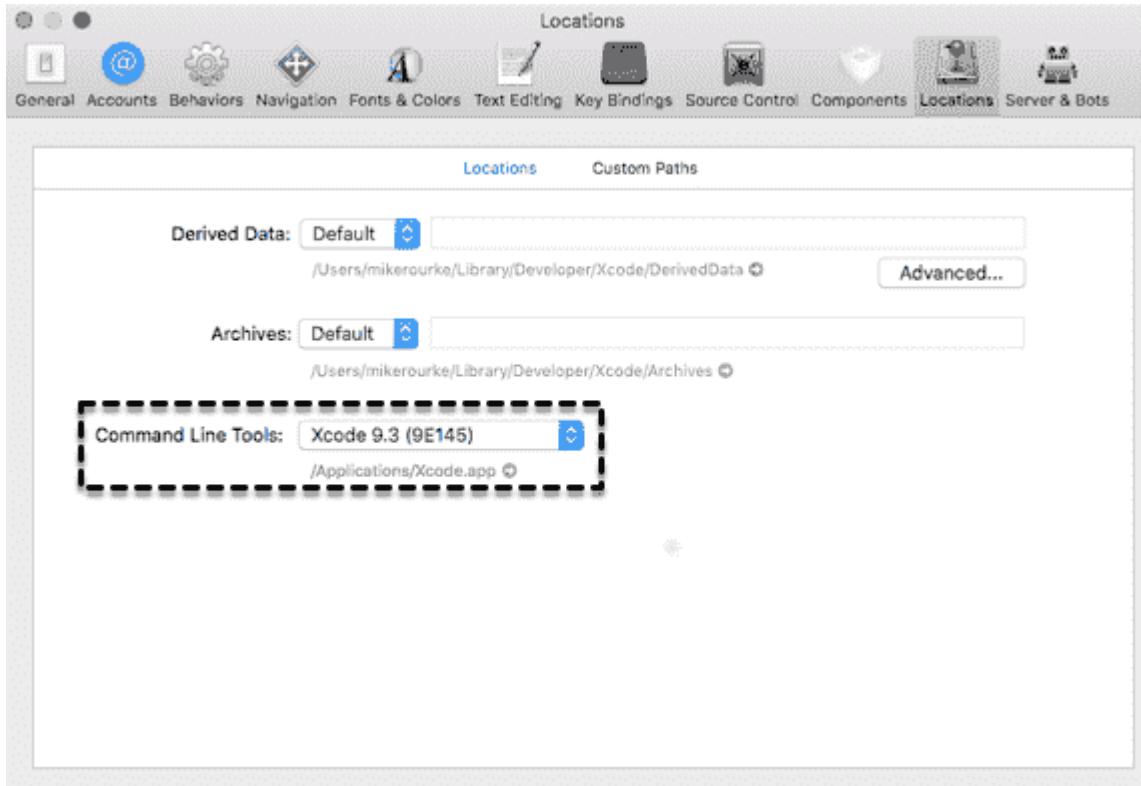
Python 是一种用于通用编程的高级编程语言。如果您想了解更多，请访问官方网站
www.python.org/。

在 macOS 上安装先决条件

在安装 EMSDK 之前，您需要安装另外三个工具：

- Xcode
- Xcode 命令行工具
- CMake

您可以从 macOS 应用商店安装 Xcode。如果您已经安装了 Xcode，可以通过转到 Xcode | 首选项 | 位置并检查命令行工具选项是否有值来检查是否已安装命令行工具。如果您安装了 Homebrew 软件包管理器，则应该已经安装了命令行工具：



检查 Xcode 命令行工具的当前版本

如果没有看到，请打开终端并运行此命令：

```
xcode-select --install
```

完成后，可以通过运行此命令来安装 CMake：

```
brew install cmake
```

在安装 Python 之前，请运行此命令：

```
python --version
```

如果您看到 `Python 2.7.xx`（其中 `xx` 是补丁版本，可以是任何数字），则可以准备安装 EMSDK。如果出现错误，表示找不到 Python 命令，或者看到 `Python 3.x.xx`，我建议您安装 `pyenv`，一个 Python 版本管理器。要安装 `pyenv`，请运行此命令：

```
brew install pyenv
```

您需要执行一些额外的配置步骤才能完成安装。请按照 github.com/pyenv/pyenv#homebrew-on-mac-os-x 上的 Homebrew 安装说明进行操作。安装和配置 `pyenv` 后，运行此命令安装 Python 2.7：

```
pyenv install 2.7.15
```

安装完成后，运行此命令：

```
pyenv global 2.7.15
```

为确保您使用的是正确版本的 Python，请运行此命令：

```
python --version
```

您应该看到 Python `2.7.xx`，其中 `xx` 是补丁版本（我看到的是 `2.7.10`，这将可以正常工作）。

在 Ubuntu 上安装先决条件

Ubuntu 应该已经安装了 Python 2.7。您可以通过运行此命令确认：

```
python --version
```

如果您看到 Python `2.7.xx`（其中 `xx` 是补丁版本，可以是任何数字），则可以准备安装 EMSDK。如果出现错误，表示找不到 `python` 命令，或者看到 `Python 3.x.xx`，我建议您安装 `pyenv`，一个 Python 版本管理器。在安装 `pyenv` 之前，请检查是否已安装 `curl`。您可以通过运行以下命令来执行此操作：

```
curl --version
```

如果您看到版本号和其他信息，则已安装 `curl`。如果没有，您可以通过运行以下命令来安装 `curl`：

```
sudo apt-get install curl
```

`curl` 安装完成后，运行此命令安装 `pyenv`：

```
curl -L https://github.com/pyenv/pyenv-installer/raw/master/bin/pyenv-installer | bash
```

安装和配置 `pyenv` 后，运行此命令安装 Python 2.7：

```
pyenv install 2.7.15
```

如果遇到构建问题，请转到 github.com/pyenv/pyenv/wiki/common-build-problems 上的常见构建问题页面。安装完成后，运行此命令：

```
pyenv global 2.7.15
```

为确保您使用的是正确版本的 Python，请运行此命令：

```
python --version
```

您应该看到 `Python 2.7.xx`，其中 `xx` 是补丁版本（我看到的是 `2.7.10`，这将可以正常工作）。

在 Windows 上安装先决条件

Windows 的唯一额外先决条件是 Python 2.7。在尝试安装之前，运行此命令：

```
python --version
```

如果您看到 `Python 2.7.xx`（其中 `xx` 是补丁版本，可以是任何数字），则可以准备安装 EMSDK。如果出现错误，表示找不到 Python 命令，或者看到 `Python 3.x.xx` 并且系统上没有安装 Python 2.7，请运行此命令安装 Python 2.7：

```
choco install python2 -y
```

如果在安装 Python 2.7 之前看到 `Python 3.x.xx`，您应该能够通过更新路径来更改当前的 Python 版本。在尝试安装 EMSDK 之前，运行此命令将 Python 设置为 2.7：

```
SET PATH=C:\Python27\python.exe
```

安装和配置 EMSDK

如果您已安装了所有先决条件，就可以准备安装 EMSDK 了。获取 EMSDK 并使其运行的过程相对简单。在本节中，我们将介绍 EMSDK 的安装过程，并演示如何更新您的 VS Code C/C++ 配置以适应 Emscripten。

跨所有平台的安装过程

首先，选择一个文件夹来安装 EMSDK。我创建了一个文件夹在 `~/Tooling`（或者在 Windows 上是 `C:\Users\Mike\Tooling`）。在终端中，`cd` 到你刚创建的文件夹，并运行这个命令：

```
git clone https://github.com/juj/emsdk.git
```

一旦克隆过程完成，请按照下面对应你的平台的部分中的说明完成安装。

在 macOS 和 Ubuntu 上安装

一旦克隆过程完成，运行以下代码片段中的每个命令。如果看到一条建议你运行 `git pull` 而不是 `./emsdk update` 的消息，请在运行 `./emsdk install latest` 命令之前使用 `git pull` 命令：

```
# Change directory into the EMSDK installation folder
cd emsdk

# Fetch the latest registry of available tools
./emsdk update

# Download and install the latest SDK tools
./emsdk install latest

# Make the latest SDK active for the current user (writes ~/.
```

```
emscripten file)
./emsdk activate latest

# Activate PATH and other environment variables in the current
# Terminal
source ./emsdk_env.sh
```

`source ./emsdk_env.sh` 命令将在当前终端中激活环境变量，这意味着每次创建新的终端实例时，你都需要重新运行它。为了避免这一步，你可以将以下行添加到你的 Bash 或 Zsh 配置文件中（即 `~/.bash_profile` 或 `~/.zshrc`）：

```
source ~/Tooling/emsdk/emsdk_env.sh > /dev/null
```

如果你将 EMSDK 安装在不同的位置，请确保更新路径以反映这一点。将这行添加到你的配置文件中将自动运行该环境更新命令，这样你就可以立即开始使用 EMSDK。为了确保你可以使用 Emscripten 编译器，请运行这个命令：

```
emcc --version
```

如果你看到一个带有版本信息的消息，设置就成功了。如果你看到一个错误消息，说明找不到该命令，请仔细检查你的配置。你可能在你的 Bash 或 Zsh 配置文件中指定了无效的 `emsdk_env.sh` 路径。

在 Windows 上安装和配置

在完成安装之前，我建议你以后使用 **PowerShell**。本书中的示例将在 `cmder` 中使用 PowerShell。一旦克隆过程完成，运行以下代码片段中给出的每个命令。如果看到一条建议你运行 `git pull` 而不是 `./emsdk update` 的消息，请在运行 `./emsdk install latest` 命令之前使用 `git pull` 命令：

```
# Change directory into the EMSDK installation folder
cd emsdk

# Fetch the latest registry of available tools
.\emsdk update
```

```
# Download and install the latest SDK tools
.\emsdk install latest

# Make the latest SDK active for the current user (writes ~/.emscripten file)
.\emsdk activate --global latest
```

`.\emsdk activate` 命令中的 `--global` 标志允许你在每个会话中运行 `emcc` 而无需运行脚本来设置环境变量。为了确保你可以使用 Emscripten 编译器，请重新启动你的 CLI 并运行这个命令：

```
emcc --version
```

如果你看到一个带有版本信息的消息，设置就成功了。

在 VS Code 中配置

如果你还没有这样做，创建一个包含我们将要使用的代码示例的文件夹（示例使用名称 `book-examples`）。在 VS Code 中打开这个文件夹，按 *F1* 键，选择 *C/Cpp: Edit Configurations...* 来创建一个 `.vscode/c_cpp_properties.json` 文件在你项目的根目录。它应该会自动打开文件。将以下行添加到 `browse.path` 数组

中：`"${env:EMSCRIPTEN}/system/include"`。这将防止在包含 `emscripten.h` 头文件时抛出错误。如果它没有自动生成，你可能需要手动创建 `browse` 对象并添加一个 `path` 条目。以下代码片段代表了 Ubuntu 上更新后的配置文件：

```
{
  "name": "Linux",
  "includePath": [
    "/usr/include",
    "/usr/local/include",
    "${workspaceFolder}",
    "${env:EMSCRIPTEN}/system/include"
  ],
  "defines": [],
  "intelliSenseMode": "clang-x64",
  "browse": {
```

```
"path": [
    "/usr/include",
    "/usr/local/include",
    "${workspaceFolder}"
],
"limitSymbolsToIncludedHeaders": true,
"databaseFilename": ""

}

}
```

测试编译器

安装和配置 EMSDK 后，你需要测试它以确保你能够从 C/C++ 代码生成 Wasm 模块。测试的最简单方法是使用 `emcc` 命令编译一些代码，并尝试在浏览器中运行它。在这一部分，我们将通过编写和编译一些简单的 C 代码并评估与 `.wasm` 输出相关联的 Wat 来验证 EMSDK 的安装。

C 代码

我们将使用一些非常简单的 C 代码来测试我们的编译器安装。我们不需要导入任何头文件或外部库。我们不会在这个测试中使用 C++，因为我们需要对 C++ 执行额外的步骤，以防止名称混淆，我们将在第六章中更详细地描述。本节的代码位于 `learn-webassembly` 存储库的 `/chapter-04-installing-deps` 文件夹中。按照这里列出的说明来测试 EMSDK。

在你的 `/book-examples` 文件夹中创建一个名为 `/chapter-04-installing-deps` 的子文件夹。接下来，在这个文件夹中创建一个名为 `main.c` 的新文件，并填充以下内容：

```
int addTwoNumbers(int leftValue, int rightValue) {
    return leftValue + rightValue;
}
```

编译 C 代码

为了使用 Emscripten 编译 C/C++ 文件，我们将使用 `emcc` 命令。我们需要向编译器传递一些参数，以确保我们获得一个在浏览器中可以利用的有效输出。为了从 C/C++ 文件生成 Wasm 文件，命令遵循这种格式：

```
emcc <file.c> -Os -s WASM=1 -s SIDE_MODULE=1 -s BINARYEN_ASYNC_COMPILATION=0 -o <file.wasm>
```

以下是 `emcc` 命令的每个参数的详细说明：

参数	描述
<code><file.c></code>	将被编译为 Wasm 模块的 C 或 C++ 输入文件的路径；当我们运行命令时，我们将用实际文件路径替换它。
<code>-Os</code>	编译器优化级别。这个优化标志允许模块实例化，而不需要 Emscripten 的粘合代码。
<code>-s WASM=1</code>	告诉编译器将代码编译为 WebAssembly。
<code>-s SIDE_MODULE=1</code>	确保只输出一个 WebAssembly 模块（没有粘合代码）。
<code>-s BINARYEN_ASYNC_COMPILATION=0</code>	来自官方文档：是否异步编译 wasm，这更有效，不会阻塞主线程。目前，这对于除了最小的模块之外的所有模块在 V8 中运行是必需的。
<code>-o <file.wasm></code>	输出文件 <code>.wasm</code> 文件的路径。当我们运行命令时，我们将用所需的输出路径替换它。

为了测试 Emscripten 是否正常工作，请在 VS Code 中打开集成终端并运行以下命令：

```
# Ensure you're in the /chapter-04-installing-deps folder:  
cd chapter-04-installing-deps  
  
# Compile the main.c file to main.wasm:  
emcc main.c -Os -s WASM=1 -s SIDE_MODULE=1 -s BINARYEN_ASYNC_COMPILATION=0 -o main.wasm
```

第一次编译文件可能需要一分钟，但后续构建将会快得多。如果编译成功，你应该在 `/chapter-04-installing-deps` 文件夹中看到一个 `main.wasm` 文件。如果遇到错误，Emscripten 的错误消息应该足够详细，以帮助你纠正问题。

如果一切顺利完成，你可以通过在 VS Code 的文件资源管理器中右键单击 `main.wasm` 并从上下文菜单中选择显示 WebAssembly 来查看与 `main.wasm` 文件相关的 Wat。输出应该如下所示：

```
(module
  (type $t0 (func (param i32)))
  (type $t1 (func (param i32 i32) (result i32)))
  (type $t2 (func))
  (type $t3 (func (result f64)))
  (import "env" "table" (table $env.table 2 anyfunc))
  (import "env" "memoryBase" (global $env.memoryBase i32))
  (import "env" "tableBase" (global $env.tableBase i32))
  (import "env" "abort" (func $env.abort (type $t0)))
  (func $_addTwoNumbers (type $t1) (param $p0 i32) (param $p1
i32) (result i32)
    get_local $p1
    get_local $p0
    i32.add)
  (func $runPostSets (type $t2)
    nop)
  (func $__post_instantiate (type $t2)
    get_global $env.memoryBase
    set_global $g2
    get_global $g2
    i32.const 5242880
    i32.add
    set_global $g3)
  (func $f4 (type $t3) (result f64)
    i32.const 0
    call $env.abort
    f64.const 0x0p+0 (;=0;))
  (global $g2 (mut i32) (i32.const 0))
  (global $g3 (mut i32) (i32.const 1))
  (global $fp$_addTwoNumbers i32 (i32.const 1))
  (export "__post_instantiate" (func $__post_instantiate))
  (export "_addTwoNumbers" (func $_addTwoNumbers))
  (export "runPostSets" (func $runPostSets)))
```

```
(export "fp$_addTwoNumbers" (global 4))
(elem (get_global $env.tableBase) $f4 $_addTwoNumbers))
```

如果编译器成功运行，你就可以继续下一步，编写 JavaScript 代码与模块进行交互，这将在下一章中介绍。

摘要

在本章中，我们介绍了在使用 WebAssembly 时的整体开发工作流程。为了生成我们的 `.wasm` 文件，我们正在使用 Emscripten，这需要安装 EMSDK。在审查任何安装细节之前，我们讨论了底层技术，并描述了它们如何相互关联以及与 WebAssembly 的关系。我们介绍了在本地计算机上使 EMSDK 工作所需的每个步骤。每个平台上 EMSDK 的安装过程都有所介绍，以及 EMSDK 的安装和配置说明。安装 EMSDK 之后，我们测试了编译器（不是）。那是我们在上一节中运行的 `emcc` 命令。使用 `emcc` 命令对一个简单的 C 代码文件，以确保 Emscripten 工作正常。在下一章中，我们将详细介绍创建和加载你的第一个模块的过程！

问题

1. 开发工作流程中的五个步骤是什么？
2. Emscripten 在编译过程中代表哪个阶段或结束？
3. IR 代表什么（LLVM 的输出）？
4. EMSDK 在 Emscripten 的编译过程中扮演什么角色？
5. 在所有三个平台（macOS、Windows 和 Linux）上需要哪些 EMSDK 先决条件？
6. 为什么需要在使用 Emscripten 编译器之前运行 `emsdk_env` 脚本？
7. 为什么需要将 `"${env:EMSCRIPTEN}/system/include"` 路径添加到 C/Cpp 配置文件中？
8. 用于将 C/C++ 编译为 Wasm 模块的命令是什么？
9. `os` 编译器标志代表什么？

进一步阅读

- Emscripten: emscripten.org
- LLVM 编译器基础设施项目：llvm.org

- 使用 Visual Studio Code 进行 C++ 编程：code.visualstudio.com/docs/languages/cpp

第五章：创建和加载 WebAssembly 模块

我们在第四章 安装所需的依赖项中向 `emcc` 命令传递的标志产生了一个单一的 `.wasm` 文件，可以使用本机的 `WebAssembly` 对象在浏览器中加载和实例化。C 代码是一个非常简单的示例，旨在测试编译器，而无需考虑包含的库或 WebAssembly 的限制。通过利用 Emscripten 的一些功能，我们可以克服 C/C++ 代码的一些 WebAssembly 的限制，而只有最小的性能损失。

在本章中，我们将涵盖与 Emscripten 的粘合代码使用对应的编译和加载步骤。我们还将描述使用浏览器的 `WebAssembly` 对象编译/输出严格的 `.wasm` 文件并加载它们的过程。

本章的目标是理解以下内容：

- 利用 Emscripten 的 JavaScript “glue” 代码编译 C 代码的过程
- 如何在浏览器中加载 Emscripten 模块
- 只输出 `.wasm` 文件的 C 代码编译过程（没有“glue”代码）
- 如何在 VS Code 中配置构建任务
- 如何使用全局的 `WebAssembly` 对象在浏览器中编译和加载 Wasm 模块

使用 Emscripten 粘合代码编译 C

在第四章 安装所需的依赖项中，您编写并编译了一个简单的三行程序，以确保您的 Emscripten 安装有效。我们向 `emcc` 命令传递了几个标志，这些标志要求只输出一个 `.wasm` 文件。通过向 `emcc` 命令传递其他标志，我们可以在 `.wasm` 文件旁边输出 JavaScript 粘合代码以及一个处理加载过程的 HTML 文件。在本节中，我们将编写一个更复杂的 C 程序，并使用 Emscripten 提供的输出选项进行编译。

编写示例 C 代码

我们在第四章中涵盖的示例中没有包含任何头文件或传递任何函数，安装所需的依赖项。由于代码的目的仅是测试编译器安装是否有效，因此并不需要太多。Emscripten 提供了许多额外的功能，使我们能够与 JavaScript 以及反之互动我们的 C 和 C++ 代码。其中一些功能是 Emscripten 特有的，不对应核心规范或其 API。在我们的第一个示例中，我们将利用 Emscripten 的一个移植库和 Emscripten 的 API 提供的一个函数。

以下程序使用**Simple DirectMedia Layer (SDL2)** 在画布上对角移动一个矩形的无限循环。它取自 github.com/timhutton/sdl-canvas-wasm，但我将其从 C++ 转换为 C 并稍微修改了代码。本节的代码位于 `learn-webassembly` 存储库的 `/chapter-05-create-load-module` 文件夹中。按照以下说明使用 Emscripten 编译 C。

在您的 `/book-examples` 文件夹中创建一个名为 `/chapter-05-create-load-module` 的文件夹。在此文件夹中创建一个名为 `with-glue.c` 的新文件，并填充以下内容：

```
/*
 * Converted to C code taken from:
 * https://github.com/timhutton/sdl-canvas-wasm
 * Some of the variable names and comments were also
 * slightly updated.
 */
#include <SDL2/SDL.h>#include <emscripten.h>#include <stdlib.h>// This enables us to have a single point of reference// for the current iteration and renderer, rather than// have to refer to them separately.
typedef struct Context {
    SDL_Renderer *renderer;
    int iteration;
} Context;

/*
 * Looping function that draws a blue square on a red
 * background and moves it across the <canvas>.
*/
void mainloop(void *arg) {
    Context *ctx = (Context *)arg;
    SDL_Renderer *renderer = ctx->renderer;
    int iteration = ctx->iteration;

    // This sets the background color to red:
    SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
    SDL_RenderClear(renderer);
```

```
// This creates the moving blue square, the rect.x
// and rect.y values update with each iteration to move
// 1px at a time, so the square will move down and
// to the right infinitely:
SDL_Rect rect;
rect.x = iteration;
rect.y = iteration;
rect.w = 50;
rect.h = 50;
SDL_SetRenderDrawColor(renderer, 0, 0, 255, 255);
SDL_RenderFillRect(renderer, &rect);

SDL_RenderPresent(renderer);

// This resets the counter to 0 as soon as the iteration
// hits the maximum canvas dimension (otherwise you'd
// never see the blue square after it travelled across
// the canvas once).
if (iteration == 255) {
    ctx->iteration = 0;
} else {
    ctx->iteration++;
}
}

int main() {
    SDL_Init(SDL_INIT_VIDEO);
    SDL_Window *window;
    SDL_Renderer *renderer;

    // The first two 255 values represent the size of the <ca
    nvas>
    // element in pixels.
    SDL_CreateWindowAndRenderer(255, 255, 0, &window, &render
er);
```

```

Context ctx;
ctx.renderer = renderer;
ctx.iteration = 0;

// Call the function repeatedly:
int infinite_loop = 1;

// Call the function as fast as the browser wants to rend
er
// (typically 60fps):
int fps = -1;

// This is a function from emscripten.h, it sets a C func
tion
// as the main event loop for the calling thread:
emscripten_set_main_loop_arg(mainloop, &ctx, fps, infinit
e_loop);

SDL_DestroyRenderer(renderer);
SDL_DestroyWindow(window);
SDL_Quit();

return EXIT_SUCCESS;
}

```

`main()` 函数末尾的 `emscripten_set_main_loop_arg()` 是可用的，因为我们在文件顶部包含了 `emscripten.h`。以 `SDL_` 为前缀的变量和函数是可用的，因为在文件顶部包含了 `#include <SDL2/SDL.h>`。如果您在 `<SDL2/SDL.h>` 语句下看到了红色的波浪线错误，您可以忽略它。这是因为 SDL 的 `include` 路径不在您的 `c_cpp_properties.json` 文件中。

编译示例 C 代码

现在我们已经编写了我们的 C 代码，我们需要编译它。您必须传递给 `emcc` 命令的一个必需标志是 `-o <target>`，其中 `<target>` 是所需输出文件的路径。该文件的扩展名不仅仅是输出该文件；它会影响编译器做出的一些决定。下表摘自 Emscripten 的 `emcc` 文档

kripken.github.io/emscripten-site/docs/tools_reference/emcc.html#emcc-o-target，定义了根据指定的文件扩展名生成的输出类型：

扩展名	输出
<code><name>.js</code>	JavaScript 胶水代码（如果指定了 <code>-s WASM=1</code> 标志，则还有 <code>.wasm</code> ）。
<code><name>.html</code>	HTML 和单独的 JavaScript 文件（ <code><name>.js</code> ）。有单独的 JavaScript 文件可以提高页面加载时间。
<code><name>.bc</code>	LLVM 位码（默认）。
<code><name>.o</code>	LLVM 位码（与 <code>.bc</code> 相同）。
<code><name>.wasm</code>	仅 Wasm 文件（使用第四章中指定的标志）。

您可以忽略 `.bc` 和 `.o` 文件扩展名，我们不需要输出 LLVM 位码。`.wasm` 扩展名不在 `emcc` 工具参考页面上，但如果您传递正确的编译器标志，它是一个有效的选项。这些输出选项影响我们编写的 C/C++ 代码。

输出带有胶水代码的 HTML

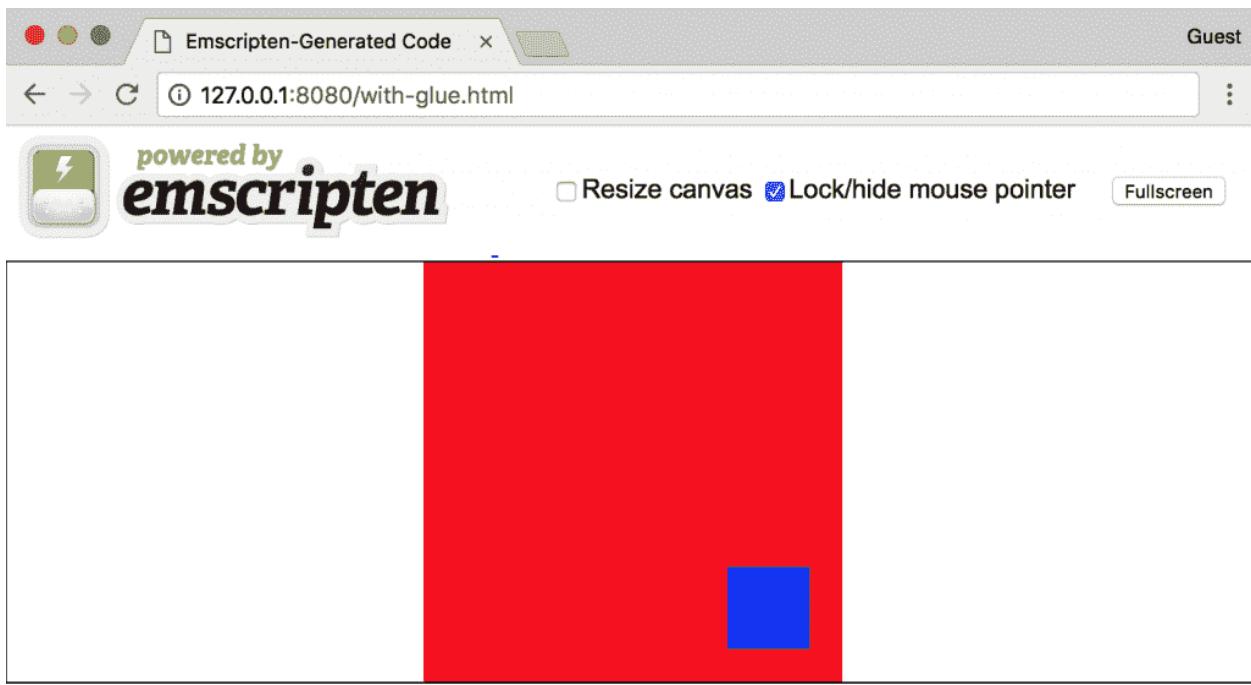
如果您为输出文件指定 HTML 文件扩展名（例如，`-o with-glue.html`），您将得到一个 `with-glue.html`、`with-glue.js` 和 `with-glue.wasm` 文件（假设您还指定了 `-s WASM=1`）。如果您在源 C/C++ 文件中有一个 `main()` 函数，它将在 HTML 加载后立即执行该函数。让我们编译我们的示例 C 代码，看看它是如何运行的。要使用 HTML 文件和 JavaScript 胶水代码进行编译，`cd` 到 `/chapter-05-create-load-module` 文件夹，并运行以下命令：

```
emcc with-glue.c -O3 -s WASM=1 -s USE	SDL2=2 -o with-glue.html
```

第一次运行此命令时，Emscripten 将下载并构建 `SDL2` 库。这可能需要几分钟才能完成，但您只需要等待一次。Emscripten 会缓存该库，因此后续构建速度会快得多。构建完成后，您将在文件夹中看到三个新文件：HTML、JavaScript 和 Wasm 文件。运行以下命令在本地 `serve` 文件：

```
serve -l 8080
```

如果您在浏览器中打开 `http://127.0.0.1:8080/with-glue.html`，您应该会看到以下内容：



在浏览器中运行 Emscripten 加载代码

蓝色矩形应该从红色矩形的左上角对角线移动到右下角。由于您在 C 文件中指定了 `main()` 函数，Emscripten 知道应该立即执行它。如果您在 VS code 中打开 `with-glue.html` 文件并滚动到文件底部，您将看到加载代码。您不会看到任何对 `WebAssembly` 对象的引用；这在 JavaScript 胶水代码文件中处理。

输出没有 HTML 的胶水代码

Emscripten 在 HTML 文件中生成的加载代码包含错误处理和其他有用的功能，以确保模块在执行 `main()` 函数之前加载。如果您为输出文件的扩展名指定 `.js`，则必须自己创建 HTML 文件并编写加载代码。在下一节中，我们将更详细地讨论加载代码。

加载 Emscripten 模块

加载和与使用 Emscripten 的胶水代码的模块进行交互与 WebAssembly 的 JavaScript API 有很大不同。这是因为 Emscripten 为与 JavaScript 代码交互提供了额外的功能。在本节中，我们将讨论 Emscripten 在输出 HTML 文件时提供的加载代码，并审查在浏览器中加载 Emscripten 模块的过程。

预生成的加载代码

如果在运行 `emcc` 命令时指定了 `-o <target>.html`，Emscripten 会生成一个 HTML 文件，并自动添加代码来加载模块到文件的末尾。以下是 HTML 文件中加载代码的样子，其中排除了每个 `Module` 函数的内容：

```
var statusElement = document.getElementById('status');
var progressElement = document.getElementById('progress');
var spinnerElement = document.getElementById('spinner');

var Module = {
    preRun: [],
    postRun: [],
    print: (function() {...})(),
    printErr: function(text) {...},
    canvas: (function() {...})(),
    setStatus: function(text) {...},
    totalDependencies: 0,
    monitorRunDependencies: function(left) {...}
};

Module.setStatus('Downloading...');

window.onerror = function(event) {
    Module.setStatus('Exception thrown, see JavaScript console');
    spinnerElement.style.display = 'none';
    Module.setStatus = function(text) {
        if (text) Module.printErr('[post-exception status] ' + text);
    };
};
```

`Module` 对象内的函数用于检测和解决错误，监视 `Module` 的加载状态，并在对应的粘合代码文件执行 `run()` 方法之前或之后可选择执行一些函数。下面的代码片段中显示的 `canvas` 函数返回了在加载代码之前在 HTML 文件中指定的 DOM 中的 `<canvas>` 元素：

```
canvas: (function() {
    var canvas = document.getElementById('canvas');
    canvas.addEventListener(
        'webglcontextlost',
        function(e) {
            alert('WebGL context lost. You will need to reload the
page.');
            e.preventDefault();
        },
        false
    );

    return canvas;
})(),
```

这段代码方便检测错误并确保 `Module` 已加载，但对于我们的目的，我们不需要那么冗长。

编写自定义加载代码

Emscripten 生成的加载代码提供了有用的错误处理。如果你在生产中使用 Emscripten 的输出，我建议你包含它以确保你正确处理错误。然而，我们实际上不需要所有的代码来使用我们的 `Module`。让我们编写一些更简单的代码并测试一下。首先，让我们将我们的 C 文件编译成没有 HTML 输出的粘合代码。为此，运行以下命令：

```
emcc with-glue.c -O3 -s WASM=1 -s USE_SDL=2 -s MODULARIZE=1 -
o custom-loading.js
```

- `s MODULARIZE=1` 编译器标志允许我们使用类似 Promise 的 API 来加载我们的 `Module`。编译完成后，在 `/chapter-05-create-load-module` 文件夹中创建一个名为 `custom-loading.html` 的文件，并填充以下内容：

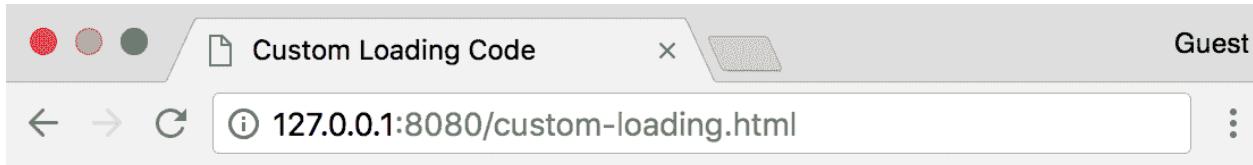
```
<!doctype html>
<html lang="en-us">
<head>
    <title>Custom Loading Code</title>
</head>
```

```
<body>
  <h1>Using Custom Loading Code</h1>
  <canvas id="canvas"></canvas>
  <script type="application/javascript" src="img/custom-loading.js"></script>
  <script type="application/javascript">
    Module({
      canvas: () => document.getElementById('canvas'))(),
    })
    .then(() => {
      console.log('Loaded!');
    });
  </script>
</body>
</html>
```

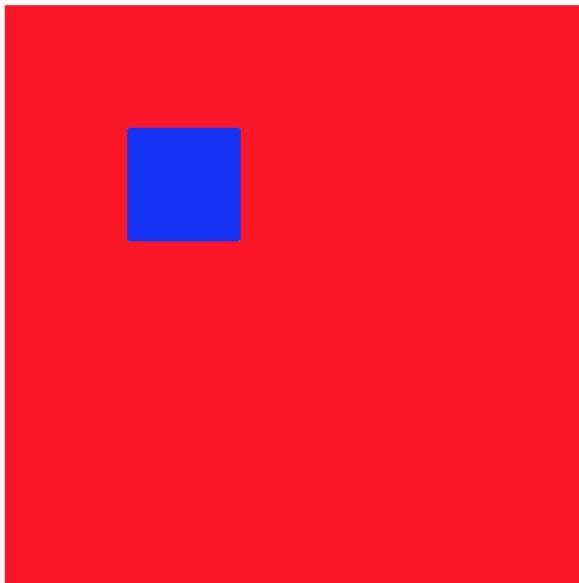
现在加载代码使用了 ES6 的箭头函数语法来加载画布函数，这减少了所需的代码行数。
通过在 `/chapter-05-create-load-module` 文件夹中运行 `serve` 命令来启动本地服务器：

```
serve -l 8080
```

当你在浏览器中导航到 `http://127.0.0.1:8080/custom-loading.html` 时，你应该看到这个：



Using Custom Loading Code



在浏览器中运行自定义加载代码

当然，我们运行的函数并不是非常复杂，但它演示了加载 Emscripten 的 `Module` 所需的基本要求。我们将在第六章中更详细地研究 `Module` 对象，与 JavaScript 交互和调试，但现在只需知道加载过程与 WebAssembly 不同，我们将在下一节中介绍。

编译不带粘合代码的 C 代码

如果我们想要按照官方规范使用 WebAssembly，而不使用 Emscripten 提供的额外功能，我们需要向 `emcc` 命令传递一些标志，并确保编写的代码可以相对轻松地被 WebAssembly 使用。在编写示例 C 代码部分，我们编写了一个程序，它在红色画布上对角移动的蓝色矩形。它利用了 Emscripten 的一个移植库 SDL2。在本节中，我们将编写和编译一些不依赖于 Emscripten 辅助方法和移植库的 C 代码。

用于 WebAssembly 的 C 代码

在我们开始编写用于 WebAssembly 模块的 C 代码之前，让我们进行一个实验。

在 `/chapter-05-create-load-module` 文件夹中打开 CLI，并尝试运行以下命令：

```
emcc with-glue.c -Os -s WASM=1 -s USE SDL=2 -s SIDE_MODULE=1  
-s BINARYEN_ASYNC_COMPILATION=0 -o try-with-glue.wasm
```

在编译完成后，你应该在 VS Code 的文件资源管理器面板中看到一个 `try-with-glue.wasm` 文件。右键单击该文件，选择显示 WebAssembly。相应的 Wat 表示的开头应该类似于以下代码：

```
(module  
  (type $t0 (func (param i32)))  
  (type $t1 (func (param i32 i32 i32 i32) (result i32)))  
  (type $t2 (func (param i32) (result i32)))  
  (type $t3 (func))  
  (type $t4 (func (param i32 i32) (result i32)))  
  (type $t5 (func (param i32 i32 i32 i32)))  
  (type $t6 (func (result i32)))  
  (type $t7 (func (result f64)))  
  (import "env" "memory" (memory $env.memory 256))  
  (import "env" "table" (table $env.table 4 anyfunc))  
  (import "env" "memoryBase" (global $env.memoryBase i32))  
  (import "env" "tableBase" (global $env.tableBase i32))  
  (import "env" "abort" (func $env.abort (type $t0)))  
  (import "env" "_SDL_CreateWindowAndRenderer" (func $env._SD  
L_CreateWindowAndRenderer (type $t1)))  
  (import "env" "_SDL_DestroyRenderer" (func $env._SDL_Destro  
yRenderer (type $t0)))  
  (import "env" "_SDL_DestroyWindow" (func $env._SDL_DestroyW  
indow (type $t0)))  
  (import "env" "_SDL_Init" (func $env._SDL_Init (type $t2)))  
  (import "env" "_SDL_Quit" (func $env._SDL_Quit (type $t3)))  
  (import "env" "_SDL_RenderClear" (func $env._SDL_RenderClea  
r (type $t2)))  
  (import "env" "_SDL_RenderFillRect" (func $env._SDL_RenderF  
illRect (type $t4)))
```

```
(import "env" "_SDL_RenderPresent" (func $env._SDL_RenderPresent (type $t0)))
(import "env" "_SDL_SetRenderDrawColor" (func $env._SDL_SetRenderDrawColor (type $t1)))
(import "env" "_emscripten_set_main_loop_arg" (func $env._emscripten_set_main_loop_arg (type $t5)))
...
...
```

如果你想在浏览器中加载并执行它，你需要向 WebAssembly 的 `instantiate()` 或 `compile()` 函数传递一个 `importObj` 对象，其中包含每个 `import "env"` 函数的 `env` 对象。Emscripten 在幕后处理所有这些工作，使用粘合代码使其成为一个非常有价值的工具。然而，我们可以通过使用 DOM 替换 SDL2 功能，同时仍然在 C 中跟踪矩形的位置。

我们将以不同的方式编写 C 代码，以确保我们只需要将一些函数传递到 `importObj.env` 对象中来执行代码。在 `/chapter-05-create-load-module` 文件夹中创建一个名为 `without-glue.c` 的文件，并填充以下内容：

```
/*
 * This file interacts with the canvas through imported functions.
 * It moves a blue rectangle diagonally across the canvas
 * (mimics the SDL example).
*/
#include <stdbool.h>#define BOUNDS 255
#define RECT_SIDE 50
#define BOUNCE_POINT (BOUNDS - RECT_SIDE)

// These functions are passed in through the importObj.env object
// and update the rectangle on the <canvas>:
extern int jsClearRect();
extern int jsFillRect(int x, int y, int width, int height);

bool isRunning = true;

typedef struct Rect {
```

```
int x;
int y;
char direction;
} Rect;

struct Rect rect;

/*
 * Updates the rectangle location by 1px in the x and y in a
 * direction based on its current position.
*/
void updateRectLocation() {
    // Since we want the rectangle to "bump" into the edge of
    // the
    // canvas, we need to determine when the right edge of th
    e
    // rectangle encounters the bounds of the canvas, which i
    s why
    // we're using the canvas width - rectangle width:
    if (rect.x == BOUNCE_POINT) rect.direction = 'L';

    // As soon as the rectangle "bumps" into the left side of
    // the
    // canvas, it should change direction again.
    if (rect.x == 0) rect.direction = 'R';

    // If the direction has changed based on the x and y
    // coordinates, ensure the x and y points update
    // accordingly:
    int incrementer = 1;
    if (rect.direction == 'L') incrementer = -1;
    rect.x = rect.x + incrementer;
    rect.y = rect.y + incrementer;
}

/*

```

```

        * Clear the existing rectangle element from the canvas and draw a
        * new one in the updated location.
    */
void moveRect() {
    jsClearRect();
    updateRectLocation();
    jsFillRect(rect.x, rect.y, RECT_SIDE, RECT_SIDE);
}

bool getIsRunning() {
    return isRunning;
}

void setIsRunning(bool newIsRunning) {
    isRunning = newIsRunning;
}

void init() {
    rect.x = 0;
    rect.y = 0;
    rect.direction = 'R';
    setIsRunning(true);
}

```

我们将从 C 代码中调用这些函数来确定x和y坐标。`setIsRunning()` 函数可用于暂停矩形的移动。现在我们的 C 代码已经准备好了，让我们来编译它。在 VS Code 终端中，`cd` 进入 `/chapter-05-create-load-module` 文件夹，并运行以下命令：

```
emcc without-glue.c -Os -s WASM=1 -s SIDE_MODULE=1 -s BINARYEN_ASYNC_COMPILATION=0 -o without-glue.wasm
```

编译完成后，你可以右键单击生成的 `without-glue.wasm` 文件，选择 Show WebAssembly 来查看 Wat 表示。你应该在文件顶部看到 `import "env"` 项的以下内容：

```
(module
  (type $t0 (func (param i32)))
  (type $t1 (func (result i32)))
  (type $t2 (func (param i32 i32 i32 i32) (result i32)))
  (type $t3 (func))
  (type $t4 (func (result f64)))
  (import "env" "memory" (memory $env.memory 256))
  (import "env" "table" (table $env.table 8 anyfunc))
  (import "env" "memoryBase" (global $env.memoryBase i32))
  (import "env" "tableBase" (global $env.tableBase i32))
  (import "env" "abort" (func $env.abort (type $t0)))
  (import "env" "_jsClearRect" (func $env._jsClearRect (type
$t1)))
  (import "env" "_jsFillRect" (func $env._jsFillRect (type $t
2)))
  ...
)
```

我们需要在 `importObj` 对象中传入 `_jsClearRect` 和 `_jsFillRect` 函数。我们将在 HTML 文件与 JavaScript 交互代码的部分介绍如何做到这一点。

在 VS Code 中使用构建任务进行编译

`emcc` 命令有点冗长，手动为不同文件在命令行上运行这个命令可能会变得麻烦。为了加快编译过程，我们可以使用 VS Code 的 Tasks 功能为我们将要使用的文件创建一个构建任务。要创建一个构建任务，选择 `Tasks | Configure Default Build Task...`，选择 `Create tasks.json from template` 选项，并选择 `Others` 来在 `.vscode` 文件夹中生成一个简单的 `tasks.json` 文件。更新文件的内容以包含以下内容：

```
{
// See https://go.microsoft.com/fwlink/?LinkId=733558
// for the documentation about the tasks.json format
"version": "2.0.0",
"tasks": [
{
  "label": "Build",
  "type": "shell",
```

```

    "command": "emcc",
    "args": [
        "${file}",
        "-Os",
        "-s", "WASM=1",
        "-s", "SIDE_MODULE=1",
        "-s", "BINARYEN_ASYNC_COMPILATION=0",
        "-o", "${fileDirname}/${fileBasenameNoExtension}.wasm"
    ],
    "group": {
        "kind": "build",
        "isDefault": true
    },
    "presentation": {
        "panel": "new"
    }
}
]
}

```

`label` 值只是一个运行任务时的名称。`type` 和 `command` 值表示它应该在 shell（终端）中运行 `emcc` 命令。`args` 值是要传递给 `emcc` 命令的参数数组（基于空格分隔）。`"${file}"` 参数告诉 VS Code 编译当前打开的文件。`"${fileDirname}/${fileBasenameNoExtension}.wasm"` 参数表示 `.wasm` 输出将与当前打开的文件具有相同的名称（带有 `.wasm` 扩展名），并且应放在当前打开文件的活动文件夹中。如果不指定 `fileDirname`，输出文件将放在根文件夹中（而不是在本例中的 `/chapter-05-create-load-module` 中）。

`group` 对象表示这个任务是默认的构建步骤，所以如果你使用键盘快捷键 `Cmd/Ctrl + Shift + B`，这就是将要运行的任务。`presentation.panel` 值为 `"new"` 告诉 VS Code 在运行构建步骤时打开一个新的 CLI 实例。这是个人偏好，可以省略。

一旦 `tasks.json` 文件完全填充，你可以保存并关闭它。要测试它，首先删除在上一节中使用 `emcc` 命令生成的 `without-glue.wasm` 文件。接下来，确保你打开了 `without-glue.c` 文件，并且光标在文件中，然后通过选择 **Tasks** | Run Build Task... 或使用键盘快捷键 `Cmd/Ctrl + Shift + B` 来运行构建任务。集成终端中的一个新面板将执行编译，一两秒后会出现一个 `without-glue.wasm` 文件。

获取和实例化 Wasm 文件

现在我们有了一个 Wasm 文件，我们需要一些 JavaScript 代码来编译和执行它。有一些步骤我们需要遵循，以确保代码可以成功地在浏览器中使用。在本节中，我们将编写一些常见的 JavaScript 加载代码，以便在其他示例中重用，创建一个演示 Wasm 模块使用的 HTML 文件，并在浏览器中测试结果。

常见的 JavaScript 加载代码

我们将在几个示例中获取和实例化一个 `.wasm` 文件，因此将 JavaScript 加载代码移到一个公共文件是有意义的。实际的获取和实例化代码只有几行，但是反复重新定义 Emscripten 期望的 `importObj` 对象是一种浪费时间。我们将使这段代码在一个通常可访问的文件中，以加快编写代码的过程。在 `/book-examples` 文件夹中创建一个名为 `/common` 的新文件夹，并添加一个名为 `load-wasm.js` 的文件，其中包含以下内容：

```
/***
 * Returns a valid importObj.env object with default values to
 * pass
 * into the WebAssembly.Instance constructor for Emscripten's
 * Wasm module.
 */
const getDefaultEnv = () => ({
  memoryBase: 0,
  tableBase: 0,
  memory: new WebAssembly.Memory({ initial: 256 }),
  table: new WebAssembly.Table({ initial: 2, element: 'anyfunc' }),
  abort: console.log
});

/***
 * Returns a WebAssembly.Instance instance compiled from the
 * specified
 * .wasm file.
 */
function loadWasm(fileName, importObj = { env: {} }) {
  // Override any default env values with the passed in importObj
}
```

```

tObj.env
  // values:
  const allEnv = Object.assign({}, getDefaultEnv(), importObj.env);

  // Ensure the importObj object includes the valid env value:
  const allImports = Object.assign({}, importObj, { env: allEnv });

  // Return the result of instantiating the module (instance and module):
  return fetch(fileName)
    .then(response => {
      if (response.ok) return response.arrayBuffer();
      throw new Error(`Unable to fetch WebAssembly file ${fileName}`);
    })
    .then(bytes => WebAssembly.instantiate(bytes, allImports));
}

```

`getDefaultEnv()` 函数为 Emscripten 的 Wasm 模块提供所需的 `importObj.env` 内容。我们希望能够传入任何其他的导入，这就是为什么使用 `Object.assign()` 语句的原因。除了 Wasm 模块期望的任何其他导入之外，Emscripten 的 Wasm 输出将始终需要这五个 `"env"` 对象的导入语句：

```

(import "env" "memory" (memory $env.memory 256))
(import "env" "table" (table $env.table 8 anyfunc))
(import "env" "memoryBase" (global $env.memoryBase i32))
(import "env" "tableBase" (global $env.tableBase i32))
(import "env" "abort" (func $env.abort (type $t0)))

```

我们需要将这些传递给 `instantiate()` 函数，以确保 Wasm 模块成功加载，否则浏览器将抛出错误。现在我们的加载代码准备好了，让我们继续进行 HTML 和矩形渲染代码。

HTML 页面

我们需要一个包含 `<canvas>` 元素和与 Wasm 模块交互的 JavaScript 代码的 HTML 页
面。在 `/chapter-05-create-load-module` 文件夹中创建一个名为 `without-glue.html` 的文件，并
填充以下内容：

```
<!doctype html>
<html lang="en-us">
<head>
  <title>No Glue Code</title>
  <script type="application/javascript" src="img/load-wasm.js"></script>
</head>
<body>
  <h1>No Glue Code</h1>
  <canvas id="myCanvas" width="255" height="255"></canvas>
  <div style="margin-top: 16px;">
    <button id="actionButton" style="width: 100px; height: 24px;">
      Pause
    </button>
  </div>
  <script type="application/javascript">
    const canvas = document.querySelector('#myCanvas');
    const ctx = canvas.getContext('2d');

    const env = {
      table: new WebAssembly.Table({ initial: 8, element: 'anyfunc' }),
      _jsFillRect: function (x, y, w, h) {
        ctx.fillStyle = '#0000ff';
        ctx.fillRect(x, y, w, h);
      },
      _jsClearRect: function() {
        ctx.fillStyle = '#ff0000';
        ctx.fillRect(0, 0, 255, 255);
      }
    };
  </script>
</body>

```

```

        },
    };

    loadWasm('without-glue.wasm', { env }).then(({ instance }) => {
        const m = instance.exports;
        m._init();

        // Move the rectangle by 1px in the x and y every 20 milliseconds:
        const loopRectMotion = () => {
            setTimeout(() => {
                m._moveRect();
                if (m._getIsRunning()) loopRectMotion();
            }, 20)
        };
    });

    // Enable you to pause and resume the rectangle movement:
    document.querySelector('#actionButton')
        .addEventListener('click', event => {
            const newIsRunning = !m._getIsRunning();
            m._setIsRunning(newIsRunning);
            event.target.innerHTML = newIsRunning ? 'Pause' :
            'Start';
            if (newIsRunning) loopRectMotion();
        });

    loopRectMotion();
});
</script>
</body>
</html>

```

这段代码将复制我们在前几节中创建的 SDL 示例，并添加一些功能。当矩形撞到右下角时，它会改变方向。您还可以使用 `<canvas>` 元素下的按钮暂停和恢复矩形的移动。您可以

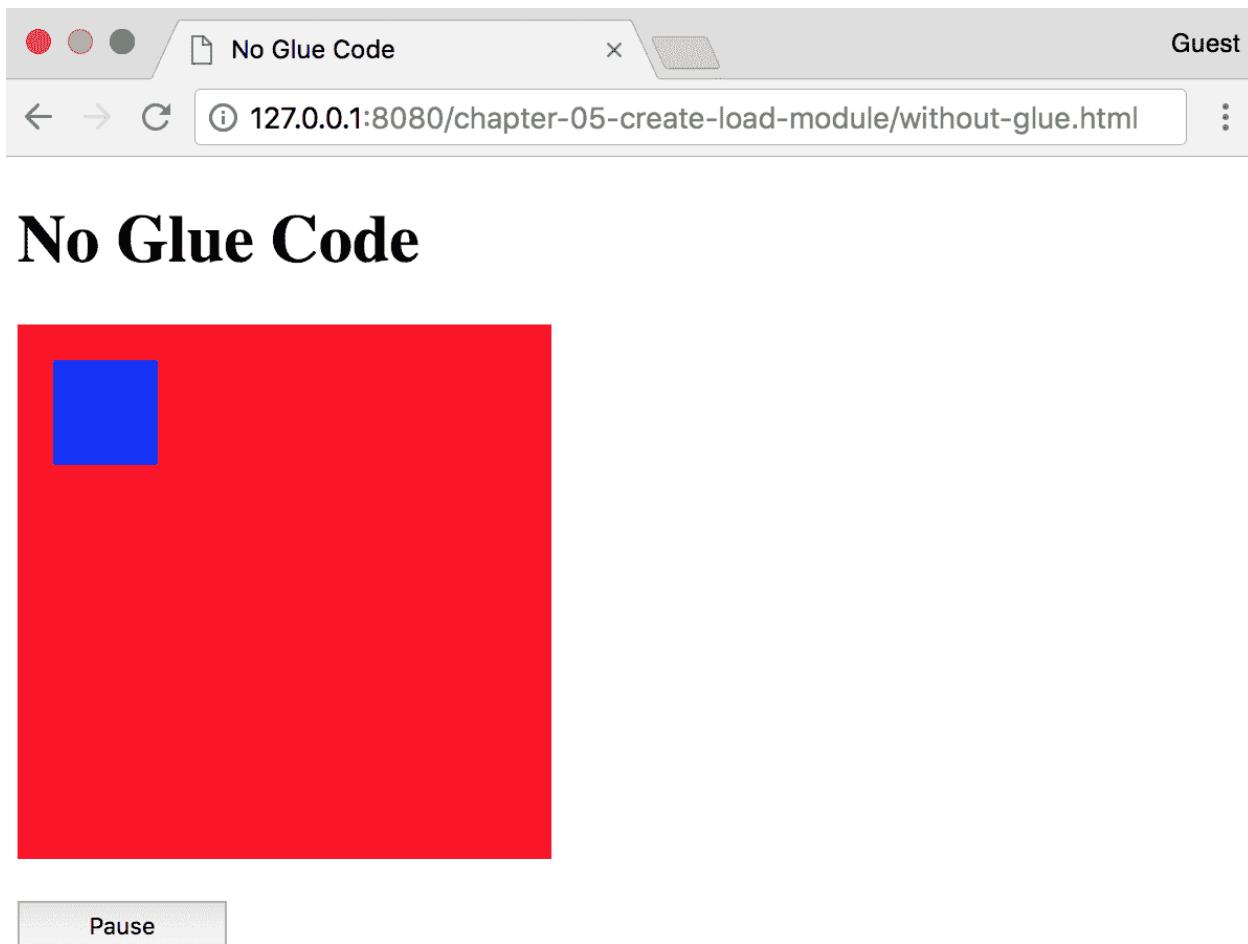
看到我们如何将 `_jsFillRect` 和 `_jsClearRect` 函数传递给 `importObj.env` 对象，以便 Wasm 模块可以引用它们。

提供所有服务

让我们在浏览器中测试我们的代码。从 VS Code 终端，确保您在 `/book-examples` 文件夹中，并运行命令启动本地服务器：

```
serve -l 8080
```

重要的是您要在 `/book-examples` 文件夹中。如果您只尝试在 `/chapter-05-create-load-module` 文件夹中提供代码，您将无法使用 `loadWasm()` 函数。如果您在浏览器中打开 `http://127.0.0.1:8080/chapter-05-create-load-module/without-glue.html`，您应该会看到这个：



在浏览器中运行的无粘合代码示例

尝试按下暂停按钮；标题应该更改为开始，矩形应该停止移动。再次点击它应该导致矩形重新开始移动。

总结

在本章中，我们介绍了使用 Emscripten 粘合代码和 Wasm 模块的编译和加载过程。通过利用 Emscripten 的一些内置功能，如移植库和辅助方法，我们能够展示 Emscripten 提供的优势。我们讨论了一些可以传递给 `emcc` 命令的编译器标志，以及这将如何影响您的输出。通过利用 VS Code 的任务功能，我们能够设置一个构建命令，以加快未来的构建过程。我们还回顾了在没有粘合代码的情况下编译和加载 Wasm 模块的过程。我们编写了一些可重用的 JavaScript 代码来加载模块，以及与我们编译的 Wasm 模块交互的代码。

在第六章，*与 JavaScript 交互和调试*中，我们将介绍在浏览器中与 JavaScript 交互和调试技术。

问题

1. SDL 代表什么？
2. 除了 JavaScript、HTML 和 Wasm，您还可以使用 `emcc` 命令的 `o` 标志生成什么其他输出类型？
3. 使用 Emscripten 的预生成加载代码有哪些优势？
4. 在 C/C++文件中，您必须如何命名您的函数，以确保它会自动在浏览器中执行编译后的输出？
5. 为什么在使用移植库时不能只使用 Wasm 文件输出而不使用“粘合”代码？
6. 在 VS Code 中运行默认构建任务的键盘快捷键是什么？
7. 在 Wasm 加载代码中，为什么我们需要 `getDefaultValue()` 方法？
8. 对于使用 Emscripten 创建的 Wasm 模块，传递给 Wasm 实例化代码的 `importObj.env` 对象需要哪五个项目？

进一步阅读

- 关于 SDL：www.libsdl.org/index.php

- Emscripten 编译器前端（emcc）：kripken.github.io/emscripten-site/docs/tools_reference/emcc.html
- 通过任务与外部工具集成：code.visualstudio.com/docs/editor/tasks
- 加载和运行 WebAssembly 代码：developer.mozilla.org/en-US/docs/WebAssembly>Loading_and_running

第六章：与 JavaScript 交互和调试

WebAssembly 中有许多令人兴奋的功能和提案。然而，在撰写本书时，功能集相当有限。就目前而言，您可以从 Emscripten 提供的一些功能中获益良多。从 JavaScript 与 C/C++ 交互（反之亦然）的过程将取决于您是否决定使用 Emscripten。

在本章中，我们将介绍如何使用 JavaScript 函数与 C/C++ 代码以及如何与 JavaScript 中编译输出的 C/C++ 代码进行交互。我们还将描述 Emscripten 的 *glue* 代码如何影响 Wasm 实例的使用方式以及如何在浏览器中调试编译代码。

本章的目标是理解以下内容：

- Emscripten 的 `Module` 与浏览器的 `WebAssembly` 对象之间的差异
- 如何从您的 JavaScript 代码中调用编译后的 C/C++ 函数
- 如何从您的 C/C++ 代码中调用 JavaScript 函数
- 在使用 C++ 时需要注意的特殊考虑事项
- 在浏览器中调试编译输出的技术

Emscripten 模块与 WebAssembly 对象

在上一章中，我们简要介绍了 Emscripten 的 `Module` 对象以及如何在浏览器中加载它。`Module` 对象提供了几种方便的方法，并且与浏览器的 `WebAssembly` 对象有很大的不同。在本节中，我们将更详细地回顾 Emscripten 的 `Module` 对象。我们还将讨论 Emscripten 的 `Module` 与 WebAssembly 的 JavaScript API 中描述的对象之间的差异。

什么是 Emscripten 模块？

Emscripten 的官方网站为 `Module` 对象提供了以下定义：

“`Module` 是一个全局 JavaScript 对象，Emscripten 生成的代码在其执行的各个点上调用它的属性。”

`Module` 不仅在加载过程中与 WebAssembly 的 `compile` 和 `instantiate` 函数不同，而且 `Module` 在全局范围内提供了一些有用的功能，否则在 WebAssembly 中需要自定义实现。在获取和加载 Emscripten 的 JavaScript *glue* 代码后，`Module` 在全局范围内 (`window.Module`) 可用。

胶水代码中的默认方法

Emscripten 的 `Module` 对象提供了一些默认方法和属性，以帮助调试和确保编译代码的成功执行。您可以利用 `preRun` 和 `postRun` 属性在 `run()` 函数调用之前或之后执行 JavaScript 代码，或将 `print()` 和 `printErr()` 函数的输出导入页面上的 HTML 元素。我们将在本书的后面使用其中一些方法。您可以在 kripken.github.io/emscripten-site/docs/api_reference/module.html 了解更多信息。

WebAssembly 对象的差异

我们在第五章中介绍了浏览器的 WebAssembly 对象和相应的加载过程，创建和加载 WebAssembly 模块。WebAssembly 的 JavaScript 和 Web API 定义了浏览器的 `window.WebAssembly` 对象中可用的对象和方法。Emscripten 的 `Module` 可以看作是 WebAssembly 的 `Module` 和 `Instance` 对象的组合，这些对象存在于 WebAssembly 的实例化函数返回的 `result` 对象中。通过将 `-s MODULARIZE=1` 标志传递给 `emcc` 命令，我们能够复制 WebAssembly 的实例化方法（在一定程度上）。随着我们评估在即将到来的章节中集成 JavaScript 和 C/C++ 的方法，我们将更详细地检查 Emscripten 的 `Module` 与浏览器的 `WebAssembly` 对象之间的差异。

从 JavaScript 调用编译后的 C/C++ 函数

从 Wasm 实例调用函数是一个相对简单的过程，无论是否使用 Emscripten 的粘合代码。利用 Emscripten 的 API 可以提供更广泛的功能和集成，但需要将粘合代码与 `.wasm` 文件一起包含。在本节中，我们将回顾通过 JavaScript 与编译后的 Wasm 实例进行交互的方法以及 Emscripten 提供的附加工具。

从 Module 调用函数

Emscripten 提供了两个函数来从 JavaScript 调用编译后的 C/C++ 函数：`ccall()` 和 `cwrap()`。这两个函数都存在于 `Module` 对象中。决定使用哪一个取决于函数是否会被多次调用。以下内容摘自 Emscripten 的 API 参考文档 `preamble.js`，可以在 kripken.github.io/emscripten-site/docs/api_reference/preamble.js.html 上查看。

在使用 `ccall()` 或 `cwrap()` 时，不需要在函数调用前加上 `_` 前缀，只需使用 C/C++ 文件中指定的名称。

Module.ccall()

`Module.ccall()` 从 JavaScript 调用编译后的 C 函数，并返回该函数的结果。

`Module.ccall()` 的函数签名如下：

```
ccall(ident, returnType, argTypes, args, opts)
```

在 `returnType` 和 `argTypes` 参数中必须指定类型名称。可能的类型

有 `"number"`、`"string"`、`"array"` 和 `"boolean"`，分别对应适当的 JavaScript 类型。不能在 `returnType` 参数中指定 `"array"`，因为无法知道数组的长度。如果函数不返回任何内容，可以为 `returnType` 指定 `null`（注意没有引号）。

`opts` 参数是一个可选的选项对象，可以包含一个名为 `async` 的布尔属性。为此属性指定值 `true` 意味着调用将执行异步操作。我们不会在任何示例中使用此参数，但如果想了解更多信息，可以在文档 kripken.github.io/emscripten-site/docs/api_reference/preamble.js.html#calling-compiled-c-functions-from-javascript 中找到。

让我们看一个 `ccall()` 的例子。以下代码取自 Emscripten 网站，演示了如何从 C 文件的编译输出中调用名为 `c_add()` 的函数：

```
// Call C from JavaScript
var result = Module.ccall(
    'c_add', // name of C function
    'number', // return type
    ['number', 'number'], // argument types
    [10, 20] // arguments
);

// result is 30
```

Module.cwrap()

`Module.cwrap()` 类似于 `ccall()`，它调用一个编译后的 C 函数。然而，它不是返回一个值，而是返回一个 JavaScript 函数，可以根据需要重复使用。`Module.cwrap()` 的函数签名

如下：

```
cwrap(ident, returnType, argTypes)
```

与 `ccall()` 一样，您可以指定代表 `returnType` 和 `argTypes` 参数的字符串值。在调用函数时，不能在 `argTypes` 中使用 `"array"` 类型，因为无法知道数组的长度。对于不返回值的函数，可以在 `returnType` 参数中使用 `null`（不带引号）。

以下代码取自 Emscripten 网站，演示了如何使用 `cwrap()` 创建可重用的函数：

```
// Call C from JavaScript
var c_javascript_add = Module.cwrap(
    'c_add', // name of C function
    'number', // return type
    ['number', 'number'] // argument types
);

// Call c_javascript_add normally
console.log(c_javascript_add(10, 20)); // 30
console.log(c_javascript_add(20, 30)); // 50
```

C++和名称修饰

您可能已经注意到，`ccall()` 和 `cwrap()` 的描述指出两者都用于调用编译后的 C 函数。故意省略了 C++，因为需要额外的步骤才能从 C++ 文件中调用函数。C++ 支持函数重载，这意味着可以多次使用相同的函数名称，但对每个函数传递不同的参数以获得不同的结果。以下是使用函数重载的一些代码示例：

```
int addNumbers(int num1, int num2) {
    return num1 + num2;
}

int addNumbers(int num1, int num2, int num3) {
    return num1 + num2 + num3;
}
```

```
int addNumbers(int num1, int num2, int num3, int num4) {
    return num1 + num2 + num3 + num4;
}

// The function will return a value based on how many
// arguments you pass it:
int getSumOfTwoNumbers = addNumbers(1, 2);
// returns 3

int getSumOfThreeNumbers = addNumbers(1, 2, 3);
// returns 6

int getSumOfFourNumbers = addNumbers(1, 2, 3, 4);
// returns 10
```

编译器需要区分这些函数。如果它使用了名称 `addNumbers`，并且您尝试在一个地方用两个参数调用该函数，在另一个地方用三个参数调用该函数，那么它将失败。要在编译后的 Wasm 中按名称调用函数，您需要将函数包装在 `extern` 块中。包装函数的一个影响是您必须明确为每个条件定义函数。以下代码片段演示了如何实现之前的函数而不进行名称混淆：

```
extern "C" {
int addTwoNumbers(int num1, int num2) {
    return num1 + num2;
}

int addThreeNumbers(int num1, int num2, int num3) {
    return num1 + num2 + num3;
}

int addFourNumbers(int num1, int num2, int num3, int num4) {
    return num1 + num2 + num3 + num4;
}
```

从 WebAssembly 实例调用函数

我们在上一章中演示了如何从 JavaScript 中调用 Wasm 实例中的函数，但那是假设您在浏览器中实例化了一个模块而没有粘合代码。Emscripten 还提供了从 Wasm 实例调用函数的能力。在模块实例化后，您可以通过从已解析的 `Promise` 的结果中访问的 `instance.exports` 对象来调用函数。MDN 的文档为 `WebAssembly.instantiateStreaming` 提供了以下函数签名：

```
Promise<ResultObject> WebAssembly.instantiateStreaming(source, importObject);
```

根据您的浏览器，您可能需要使用 `WebAssembly.instantiate()` 方法。Chrome 目前支持 `WebAssembly.instantiateStreaming()`，但如果在尝试加载模块时遇到错误，请改用 `WebAssembly.instantiate()` 方法。

`ResultObject` 包含我们需要引用的 `instance` 对象，以便从模块中调用导出的函数。以下是调用编译后的 Wasm 实例中名为 `_addTwoNumbers` 的函数的一些代码：

```
// Assume the importObj is already defined.  
WebAssembly.instantiateStreaming(  
    fetch('simple.wasm'),  
    importObj  
)  
.then(result => {  
    const addedNumbers = result.instance.exports._addTwoNumbers(1, 2);  
    // result is 3  
});
```

Emscripten 提供了一种以类似的方式执行函数调用的方法，尽管实现略有不同。如果使用类似 `Promise` 的 API，您可以从 `Module()` 解析出的 `asm` 对象中访问函数。以下示例演示了如何利用这个功能：

```
// Using Emscripten's Module  
Module()  
.then(result => {  
    // "asm" is essentially "instance"
```

```
const exports = result.asm;
const addedNumbers = exports._addTwoNumbers(1, 2);
// result is 3
});
```

使用 Emscripten 复制 WebAssembly 的 Web API 语法可以简化任何未来的重构。如果决定使用 WebAssembly 的 Web API，您可以轻松地将 `Module()` 替换为 WebAssembly 的 `instantiateStreaming()` 方法，并将 `result.asm` 替换为 `result.instance`。

从 C/C++ 调用 JavaScript 函数

从 C/C++ 代码访问 JavaScript 的功能可以在使用 WebAssembly 时增加灵活性。在 Emscripten 的粘合代码和仅使用 Wasm 的实现之间，利用 JavaScript 的方法和手段有很大的不同。在本节中，我们将介绍您可以在 C/C++ 代码中集成 JavaScript 的各种方式，无论是否使用 Emscripten。

使用粘合代码与 JavaScript 交互

Emscripten 提供了几种将 JavaScript 与 C/C++ 代码集成的技术。可用的技术在实现和复杂性上有所不同，有些只适用于特定的执行环境（例如浏览器）。决定使用哪种技术取决于您的具体用例。我们将重点介绍 `emscripten_run_script()` 函数和使用 `EM_*` 包装器内联 JavaScript 的内容。以下部分的内容取自 Emscripten 网站的与代码交互部分，网址为 kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#interacting-with-code。

使用 `emscripten_run_script()` 执行字符串。

Emscripten 网站将 `emscripten_run_script()` 函数描述为调用 JavaScript 进行 C/C++ 的最直接但略慢的方法。这是一种非常适合单行 JavaScript 代码的技术，并且对于调试非常有用。文档说明它有效地使用 `eval()` 运行代码，`eval()` 是一个执行字符串作为代码的 JavaScript 函数。以下代码取自 Emscripten 网站，演示了使用 `emscripten_run_script()` 调用浏览器的 `alert()` 函数并显示文本 'hi' 的方法：

```
emscripten_run_script("alert('hi'));
```

对于性能是一个因素的更复杂的用例，使用内联 JavaScript 提供了更好的解决方案。

使用 `EM_ASM()` 执行内联 JavaScript()

您可以在 C/C++ 文件中使用 `EM_ASM()` 包装 JavaScript 代码，并在浏览器中运行编译后的代码时执行它。以下代码演示了基本用法：

```
#include <emscripten.h>
int main() {
    EM_ASM(
        console.log('This is some JS code.');
    );
    return 0;
}
```

JavaScript 代码会立即执行，并且无法在包含它的 C/C++ 文件中重复使用。参数可以传递到 JavaScript 代码块中，其中它们作为变量 `$0`、`$1` 等到达。这些参数可以是 `int32_t` 或 `double` 类型。以下代码片段取自 Emscripten 网站，演示了如何在 `EM_ASM()` 块中使用参数：

```
EM_ASM({
    console.log('I received: ' + [ $0, $1 ]);
}, 100, 35.5);
```

重用内联 JavaScript 与 `EM_JS()`

如果您需要在 C/C++ 文件中使用可重用的函数，可以将 JavaScript 代码包装在 `EM_JS()` 块中，并像普通的 C/C++ 函数一样执行它。`EM_JS()` 的定义如下代码片段所示：

```
EM_JS(return_type, function_name, arguments, code)
```

`return_type` 参数表示与 JavaScript 代码输出对应的 C 类型（例如 `int` 或 `float`）。如果从 JavaScript 代码中没有返回任何内容，请为 `return_type` 指定 `void`。下一个参数 `function_name` 表示在从 C/C++ 文件的其他位置调用 JavaScript 代码时要使用的名称。`arguments` 参数用于定义可以从 C 调用函数传递到 JavaScript 代码中的参数。`code` 参数是用大括号括起来的 JavaScript 代码。以下代码片段取自 Emscripten 网站，演示了在 C 文件中使用 `EM_JS()` 的方法：

```
#include <emscripten.h>
EM_JS(void, take_args, (int x, float y), {
    console.log(`I received ${x} and ${y}`);
}
```

```
});

int main() {
    take_args(100, 35.5);
    return 0;
}
```

使用粘合代码的示例

让我们编写一些代码来利用所有这些功能。在本节中，我们将修改我们在第五章中使用的代码，即编译 C 而不使用粘合代码和获取和实例化 Wasm 文件部分，创建和加载 WebAssembly 模块。这是显示在红色画布上移动的蓝色矩形的代码，并且可以通过单击按钮暂停和重新启动。本节的代码位于 `learn-webassembly` 存储库中的 `/chapter-06-interact-with-js` 文件夹中。让我们首先更新 C 代码。

C 代码

在您的 `/book-examples` 文件夹中创建一个名为 `/chapter-06-interact-with-js` 的新文件夹。在 `/chapter-06-interact-with-js` 文件夹中创建一个名为 `js-with-glue.c` 的新文件，并填充以下内容：

```
/*
 * This file interacts with the canvas through imported functions.
 * It moves a blue rectangle diagonally across the canvas
 * (mimics the SDL example).
 */

#include <emsdk.h>#include <stdbool.h>#define BOUNDS 255
#define RECT_SIDE 50
#define BOUNCE_POINT (BOUNDS - RECT_SIDE)

bool isRunning = true;

typedef struct Rect {
    int x;
    int y;
```

```
char direction;
} Rect;

struct Rect rect;

/*
 * Updates the rectangle location by 1px in the x and y in a
 * direction based on its current position.
*/
void updateRectLocation() {
    // Since we want the rectangle to "bump" into the edge of
    // the
    // canvas, we need to determine when the right edge of th
    e
    // rectangle encounters the bounds of the canvas, which i
    s why
    // we're using the canvas width - rectangle width:
    if (rect.x == BOUNCE_POINT) rect.direction = 'L';

    // As soon as the rectangle "bumps" into the left side of
    // the
    // canvas, it should change direction again.
    if (rect.x == 0) rect.direction = 'R';

    // If the direction has changed based on the x and y
    // coordinates, ensure the x and y points update
    // accordingly:
    int incrementer = 1;
    if (rect.direction == 'L') incrementer = -1;
    rect.x = rect.x + incrementer;
    rect.y = rect.y + incrementer;
}

EM_JS(void, js_clear_rect, (), {
    // Clear the rectangle to ensure there's no color where i
    t
```

```

    // was before:
    var canvas = document.querySelector('#myCanvas');
    var ctx = canvas.getContext('2d');
    ctx.fillStyle = '#ff0000';
    ctx.fillRect(0, 0, 255, 255);
});

EM_JS(void, js_fill_rect, (int x, int y, int width, int height), {
    // Fill the rectangle with blue in the specified coordinates:
    var canvas = document.querySelector('#myCanvas');
    var ctx = canvas.getContext('2d');
    ctx.fillStyle = '#0000ff';
    ctx.fillRect(x, y, width, height);
});

/*
 * Clear the existing rectangle element from the canvas and draw a
 * new one in the updated location.
*/
EMSCRIPTEN_KEEPALIVE
void moveRect() {
    // Event though the js_clear_rect doesn't have any
    // parameters, we pass 0 in to prevent a compiler warning:
    js_clear_rect(0);
    updateRectLocation();
    js_fill_rect(rect.x, rect.y, RECT_SIDE, RECT_SIDE);
}

EMSCRIPTEN_KEEPALIVE
bool getIsRunning() {
    return isRunning;
}

```

```

EMSCRIPTEN_KEEPALIVE
void setIsRunning(bool newIsRunning) {
    isRunning = newIsRunning;
    EM_ASM({
        // isRunning is either 0 or 1, but in JavaScript, 0
        // is "falsy", so we can set the status text based
        // without explicitly checking if the value is 0 or
1:
        var newStatus = $0 ? 'Running' : 'Paused';
        document.querySelector('#runStatus').innerHTML = newS
tatus;
    }, isRunning);
}

EMSCRIPTEN_KEEPALIVE
void init() {
    emscripten_run_script("console.log('Initializing rectangl
e...')");
    rect.x = 0;
    rect.y = 0;
    rect.direction = 'R';
    setIsRunning(true);
    emscripten_run_script("console.log('Rectangle should be m
oving!')");
}

```

您可以看到我们使用了 Emscripten 提供的所有三种 JavaScript 集成。有两个函数 `js_clear_rect()` 和 `js_fill_rect()`，它们在 `EM_JS()` 块中定义，代替了原始示例中导入的函数。`setIsRunning()` 函数中的 `EM_ASM()` 块更新了我们将添加到 HTML 代码中的新状态元素的文本。`emscripten_run_script()` 函数只是简单地记录一些状态消息。我们需要在我们计划在模块外部使用的函数上方指定 `EMSCRIPTEN_KEEPALIVE`。如果不指定这一点，编译器将把这些函数视为死代码并将其删除。

HTML 代码

让我们在 `/chapter-06-interact-with-js` 文件夹中创建一个名为 `js-with-glue.html` 的文件，并填充以下内容：

```
<!doctype html>
<html lang="en-us">
<head>
    <title>Interact with JS using Glue Code</title>
</head>
<body>
    <h1>Interact with JS using Glue Code</h1>
    <canvas id="myCanvas" width="255" height="255"></canvas>
    <div style="margin-top: 16px;">
        <button id="actionButton" style="width: 100px; height: 24px;">Pause</button>
        <span style="width: 100px; margin-left: 8px;">Status:</span>
        <span id="runStatus" style="width: 100px;"></span>
    </div>
    <script type="application/javascript" src="img/js-with-glue.js"></script>
    <script type="application/javascript">
        Module()
        .then(result => {
            const m = result.asm;
            m._init();

            // Move the rectangle by 1px in the x and y every 20 milliseconds:
            const loopRectMotion = () => {
                setTimeout(() => {
                    m._moveRect();
                    if (m._getIsRunning()) loopRectMotion();
                }, 20)
            };
        });

        // Enable you to pause and resume the rectangle motion
    </script>
</body>

```

```
ent:  
    document.querySelector('#ActionButton')  
        .addEventListener('click', event => {  
            const newIsRunning = !m._getIsRunning();  
            m._setIsRunning(newIsRunning);  
            event.target.innerHTML = newIsRunning ? 'Pause' :  
                'Start';  
            if (newIsRunning) loopRectMotion();  
        });  
  
    loopRectMotion();  
});  
</script>  
</body>  
</html>
```

我们添加了两个 `` 元素来显示矩形移动的状态，以及相应的标签。我们使用 Emscripten 的类似 Promise 的 API 来加载模块并引用编译代码中的函数。我们不再将 `_jsFillRect` 和 `_jsClearRect` 函数传递给模块，因为我们在 `js-with-glue.c` 文件中处理了这个问题。

编译和提供结果

要编译代码，请确保你在 `/chapter-06-interact-with-js` 文件夹中，并运行以下命令：

```
emcc js-with-glue.c -O3 -s WASM=1 -s MODULARIZE=1 -o js-with-glue.js
```

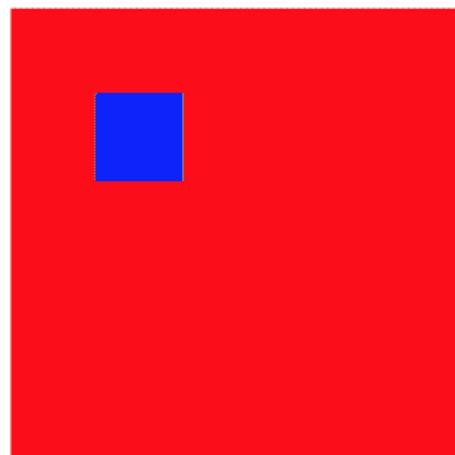
完成后，运行以下命令启动本地服务器：

```
serve -l 8080
```

打开浏览器，转到 `http://127.0.0.1:8080/js-with-glue.html`。你应该会看到类似这样的东西：



Interact with JS using Glue Code



Pause

Status: Running

在浏览器中运行胶水代码

如果你按下暂停按钮，按钮上的标题应该会变成开始，状态旁边的文本应该会变成暂停，矩形应该会停止移动。

无需胶水代码与 JavaScript 交互

在 C/C++ 文件中利用 JavaScript 代码遵循与 Emscripten 使用的技术不同的范例。你不是在 C/C++ 文件中编写 JavaScript，而是将函数传递到你的 WebAssembly 实例化代码中。在本节中，我们将更详细地描述这个过程。

使用导入对象将 JavaScript 传递给 C/C++

为了在你的 C/C++ 代码中利用 JavaScript 的功能，你需要向传递到 WebAssembly 实例化函数的 `importObj.env` 参数中添加一个函数定义。你可以在 `importObj.env` 之外或内联定义函数。以下代码片段演示了每个选项：

```
// You can define the function inside of the env object:  
const env = {  
    // Make sure you prefix the function name with "_"!
```

```

    _logValueToConsole: value => {
      console.log(`The value is ${value}`);
    }
  };

// Or define it outside of env and reference it within env:
const logValueToConsole = value => {
  console.log(`The value is ${value}`);
};

const env = {
  _logValueToConsole: logValueToConsole
};

```

考虑到 C、C++ 和 Rust 的手动内存管理和严格类型要求，你在 Wasm 模块中可以传递和利用的内容是有限的。JavaScript 允许你在代码执行过程中轻松地添加、删除和更改对象的属性值。你甚至可以通过向内置语言特性的 `prototype` 添加函数来扩展语言。C、C++ 和 Rust 更加严格，如果你不熟悉这些语言，要充分利用 WebAssembly 可能会很困难。

在 C/C++ 中调用导入的函数

你需要在使用 `importObj.env` 的 C/C++ 代码中定义你传递的 JavaScript 函数。函数签名必须与你传递的相匹配。以下示例更详细地演示了这一点。以下是与编译的 C 文件 (`index.html`) 交互的 JavaScript 代码：

```

// index.html <script> contents
const env = {
  _logAndMultiplyTwoNums: (num1, num2) => {
    const result = num1 * num2;
    console.log(result);
    return result;
  },
};

loadWasm('main.wasm', { env })

```

```
.then(({ instance }) => {
  const result = instance.exports._callMultiply(5.5, 10);
  console.log(result);
  // 55 is logged to the console twice
});
```

这是 `main.c` 的内容，它被编译为 `main.wasm` 并在 `index.html` 中使用：

```
// main.c (compiled to main.wasm)
extern float logAndMultiplyTwoNums(float num1, float num2);

float callMultiply(float num1, float num2) {
    return logAndMultiplyTwoNums(num1, num2);
}
```

你调用 C/C++ 中的 JavaScript 函数的方式与调用普通的 C/C++ 函数相同。虽然当你将它传递到 `importObj.env` 时，你需要在你的函数前加上 `_`，但在 C/C++ 文件中定义时，你不需要包括前缀。

一个没有胶水代码的例子

来自第五章的 编译不使用胶水代码的 C 和 获取和实例化 Wasm 文件 部分的示例代码演示了如何在我们的 C 文件中集成 JavaScript 而不使用 Emscripten 的胶水代码。在本节中，我们将稍微修改示例代码，并将文件类型更改为 C++。

C++ 代码

在你的 `/chapter-06-interact-with-js` 文件夹中创建一个名为 `js-without-glue.cpp` 的文件，并填充以下内容：

```
/*
 * This file interacts with the canvas through imported functions.
 * It moves a circle diagonally across the canvas.
*/
#define BOUNDS 255
#define CIRCLE_RADIUS 50
```

```
#define BOUNCE_POINT (BOUNDS - CIRCLE_RADIUS)

bool isRunning = true;

typedef struct Circle {
    int x;
    int y;
    char direction;
} Circle;

struct Circle circle;

/*
 * Updates the circle location by 1px in the x and y in a
 * direction based on its current position.
 */
void updateCircleLocation() {
    // Since we want the circle to "bump" into the edge of the
    // canvas,
    // we need to determine when the right edge of the circle
    // encounters the bounds of the canvas, which is why we're
    // using
    // the canvas width - circle width:
    if (circle.x == BOUNCE_POINT) circle.direction = 'L';

    // As soon as the circle "bumps" into the left side of the
    // canvas, it should change direction again.
    if (circle.x == CIRCLE_RADIUS) circle.direction = 'R';

    // If the direction has changed based on the x and y
    // coordinates, ensure the x and y points update accordingly:
    int incrementer = 1;
    if (circle.direction == 'L') incrementer = -1;
    circle.x = circle.x + incrementer;
```

```
    circle.y = circle.y - incrementer;
}

// We need to wrap any imported or exported functions in an
// extern block, otherwise the function names will be mangle
d.

extern "C" {
// These functions are passed in through the importObj.env ob
ject
// and update the circle on the <canvas>:
extern int jsClearCircle();
extern int jsFillCircle(int x, int y, int radius);

/*
 * Clear the existing circle element from the canvas and draw
a
 * new one in the updated location.
*/
void moveCircle() {
    jsClearCircle();
    updateCircleLocation();
    jsFillCircle(circle.x, circle.y, CIRCLE_RADIUS);
}

bool getIsRunning() {
    return isRunning;
}

void setIsRunning(bool newIsRunning) {
    isRunning = newIsRunning;
}

void init() {
    circle.x = 0;
    circle.y = 255;
    circle.direction = 'R';
```

```
    setIsRunning(true);
}
}
```

这段代码与之前的例子类似，但画布上元素的形状和方向已经改变。现在，元素是一个圆，从画布的左下角开始，沿对角线向右上移动。

HTML 代码

接下来，在你的 `/chapter-06-interact-with-js` 文件夹中创建一个名为 `js-without-glue.html` 的文件，并填充以下内容：

```
<!doctype html>
<html lang="en-us">
<head>
    <title>Interact with JS without Glue Code</title>
    <script
        type="application/javascript"
        src="img/load-wasm.js">
    </script>
    <style>
        #myCanvas {
            border: 2px solid black;
        }
        #actionButtonWrapper {
            margin-top: 16px;
        }
        #actionButton {
            width: 100px;
            height: 24px;
        }
    </style>
</head>
<body>
    <h1>Interact with JS without Glue Code</h1>
    <canvas id="myCanvas" width="255" height="255"></canvas>
```

```
<div id="actionButtonWrapper">
  <button id="actionButton">Pause</button>
</div>
<script type="application/javascript">
  const canvas = document.querySelector('#myCanvas');
  const ctx = canvas.getContext('2d');

  const fillCircle = (x, y, radius) => {
    ctx.fillStyle = '#fed530';
    // Face outline:
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, 2 * Math.PI);
    ctx.fill();
    ctx.stroke();
    ctx.closePath();

    // Eyes:
    ctx.fillStyle = '#000000';
    ctx.beginPath();
    ctx.arc(x - 15, y - 15, 6, 0, 2 * Math.PI);
    ctx.arc(x + 15, y - 15, 6, 0, 2 * Math.PI);
    ctx.fill();
    ctx.closePath();

    // Mouth:
    ctx.beginPath();
    ctx.moveTo(x - 20, y + 10);
    ctx.quadraticCurveTo(x, y + 30, x + 20, y + 10);
    ctx.lineWidth = 4;
    ctx.stroke();
    ctx.closePath();
  };

  const env = {
    table: new WebAssembly.Table({ initial: 8, element: 'anyfunc' })
  };

```

```
_jsFillCircle: fillCircle,
_jsClearCircle: function() {
    ctx.fillStyle = '#fff';
    ctx.fillRect(0, 0, 255, 255);
},
};

loadWasm('js-without-glue.wasm', { env }).then(({ instance }) => {
    const m = instance.exports;
    m._init();

    // Move the circle by 1px in the x and y every 20 milliseconds:
    const loopCircleMotion = () => {
        setTimeout(() => {
            m._moveCircle();
            if (m._getIsRunning()) loopCircleMotion();
        }, 20)
    };
}

// Enable you to pause and resume the circle movement:
document.querySelector('#ActionButton')
    .addEventListener('click', event => {
        const newIsRunning = !m._getIsRunning();
        m._setIsRunning(newIsRunning);
        event.target.innerHTML = newIsRunning ? 'Pause' :
        'Start';
        if (newIsRunning) loopCircleMotion();
    });

    loopCircleMotion();
});
</script>
```

```
</body>  
</html>
```

我们可以使用 canvas 元素的 2D 上下文上可用的函数手动绘制路径，而不是使用 `rect()` 元素。

编译和提供结果

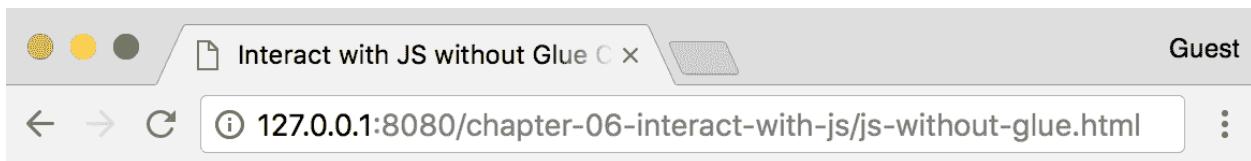
我们只生成了一个 Wasm 模块，因此可以使用我们在上一章中设置的构建任务来编译我们的代码。选择任务 | 运行构建任务...或使用键盘快捷键 *Ctrl/Cmd + Shift + B* 来编译代码。如果您不使用 VS Code，请在 `/chapter-06-interact-with-js` 文件夹中打开 CLI 实例并运行以下命令：

```
emcc js-without-glue.cpp -Os -s WASM=1 -s SIDE_MODULE=1 -s BIGNARYEN_ASYNC_COMPILATION=0 -o js-without-glue.wasm
```

完成后，在 `/book-examples` 文件夹中打开终端，并运行以下命令启动本地服务器：

```
serve -l 8080
```

打开浏览器并导航到 `http://127.0.0.1:8080/chapter-06-interact-with-js/js-without-glue.html`。您应该会看到类似以下的内容：



Interact with JS without Glue Code



Pause

在浏览器中运行的 Wasm 模块，无需粘合代码

与之前的示例一样，如果按下暂停按钮，则按钮上的标题应更改为开始，并且圆圈应停止移动。

高级 Emscripten 功能

我们在前面的部分中介绍了我们将在 JavaScript 和 C/C++ 之间频繁使用的 Emscripten 功能，但这并不是 Emscripten 提供的唯一功能。还有一些高级功能和额外的 API，您需要了解，特别是如果您计划向应用程序添加更复杂的功能。在本节中，我们将简要介绍一些这些高级功能，并提供有关您可以了解更多信息的详细信息。

Embind

Embind 是 Emscripten 提供的用于连接 JavaScript 和 C++ 的附加功能。Emscripten 的网站提供了以下描述：

"Embind 用于将 C++ 函数和类绑定到 JavaScript，以便编译后的代码可以被'普通'JavaScript 以自然的方式使用。Embind 还支持从 C++ 调用 JavaScript 类。"

Embind 是一个强大的功能，允许 JavaScript 和 C++ 之间进行紧密集成。您可以将一些 C++ 代码包装在 `EMSCRIPTEN_BINDINGS()` 块中，并通过浏览器中的 `Module` 对象引用它。让我们看一个来自 Emscripten 网站的例子。以下文件 `example.cpp` 使用 `emcc` 的 `--bind` 标志编译：

```
// example.cpp
#include <emsdk/include/bind.h>using namespace emscripten;

float lerp(float a, float b, float t) {
    return (1 - t) * a + t * b;
}

EMSCRIPTEN_BINDINGS(my_module) {
    function("lerp", &lerp);
}
```

生成的模块在 `example.html` 中加载，并调用 `lerp()` 函数：

```
<!-- example.html -->
<!doctype html>
<html>
<script src="img/example.js"></script>
<script>
    // example.js was generated by running this command:
    // emcc --bind -o example.js example.cpp
    console.log('lerp result: ' + Module.lerp(1, 2, 0.5));
</script>
</html>
```

上述示例仅代表 EmBind 功能的一小部分。您可以在 kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/embind.html 了解更多关于 EmBind 的信息。

文件系统 API

Emscripten 通过使用 FS 库提供对文件操作的支持，并公开了一个用于处理文件系统的 API。但是，默认情况下在编译项目时不会包含它，因为它可能会显著增加文件的大小。如果您的 C/C++ 代码使用文件，该库将自动添加。文件系统类型根据执行环境而异。例如，如果在 worker 内运行代码，则可以使用 `WORKERFS` 文件系统。默认情况下使用 `MEMFS`，它将数据存储在内存中，当页面重新加载时，内存中的任何数据都将丢失。您可以在 kripken.github.io/emscripten-site/docs/api_reference/Filesystem-API.html#filesystem-api 阅读有关文件系统 API 的更多信息。

Fetch API

Emscripten 还提供了 Fetch API。以下内容摘自文档：

"Emscripten Fetch API 允许本机代码通过 XHR (HTTP GET、PUT、POST) 从远程服务器传输文件，并将下载的文件持久存储在浏览器的 IndexedDB 存储中，以便可以在随后的页面访问中本地重新访问。Fetch API 可以从多个线程调用，并且可以根据需要同步或异步运行网络请求。"

Fetch API 可用于与 Emscripten 的其他功能集成。如果您需要获取 Emscripten 未使用的数据，应使用浏览器的 Fetch API (developer.mozilla.org/en-US/docs/Web/API/Fetch_API)。您可以在 kripken.github.io/emscripten-site/docs/api_reference/fetch.html 上了解有关 Fetch API 的更多信息。

在浏览器中调试

在浏览器中有效地调试 JavaScript 代码并不总是容易的。然而，浏览器和具有内置调试功能的编辑器/IDE 的开发工具已经显著改进。不幸的是，将 WebAssembly 添加到 Web 应用程序会给调试过程增加额外的复杂性。在本节中，我们将回顾一些调试 JavaScript 并利用 Wasm 的技术，以及 Emscripten 提供的一些额外功能。

高级概述

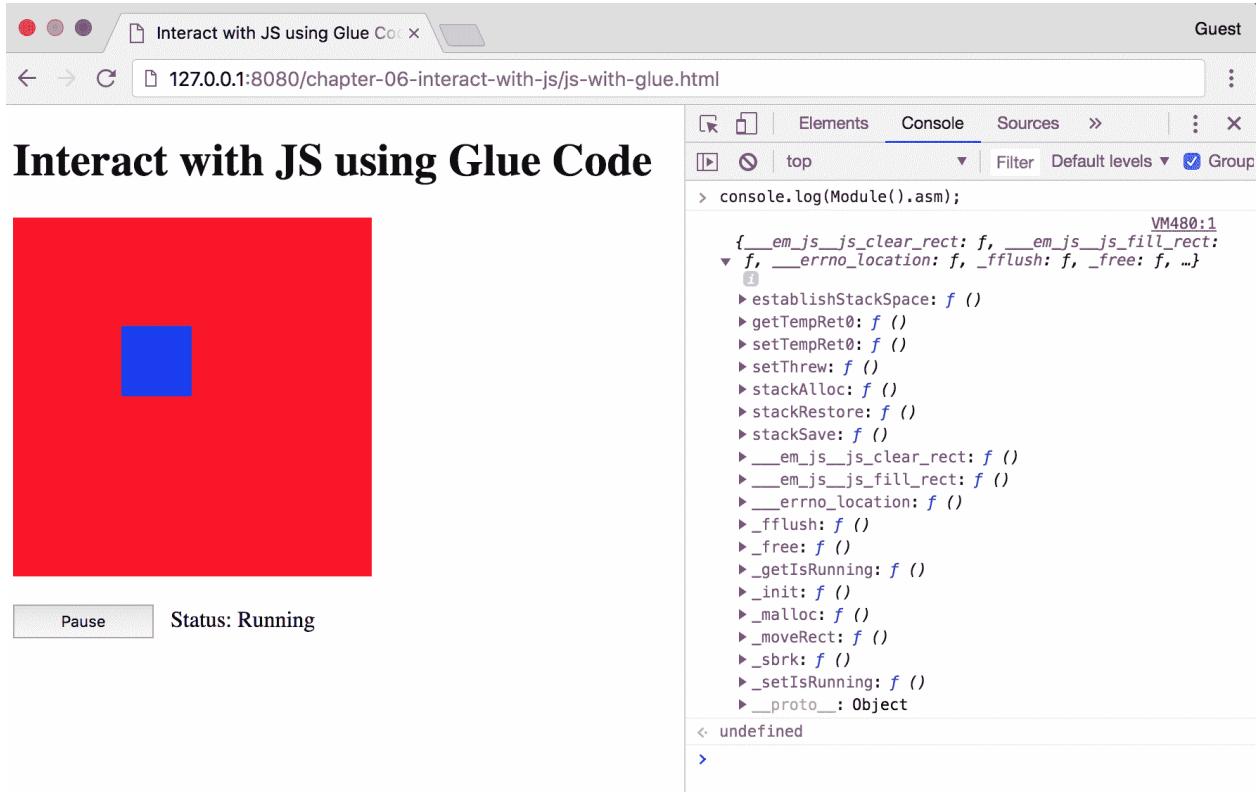
调试 Emscripten 的 `Module` 相对比较简单。Emscripten 的错误消息形式良好且描述清晰，因此通常您会立即发现问题的原因。您可以在浏览器的开发工具控制台中查看这些消息。

如果在运行 `emcc` 命令时指定了 `.html` 输出，一些调试代码将已经内置（`Module.print` 和 `Module.printErr`）。在 HTML 文件中，加载代码设置了 `window.onerror` 事件来调用 `Module.printErr` 事件，因此您可以查看加载时发生的错误的详细信息。

您可能会遇到的一个常见错误是调用错误的函数名称。如果您正在使用 Emscripten 的类似 Promise 的 API，可以通过在浏览器控制台中运行以下代码来打印出可用的函数：

```
console.log(Module().asm);
```

以下屏幕截图显示了我们在本章的从 C/C++ 调用 JavaScript 函数部分中使用的 `js-with-glue.js` 示例的输出：



手的错误，记录到控制台可能会变得繁琐和难以管理。幸运的是，您可以使用源映射来提高调试能力。

使用源映射

Emscripten 有能力通过向编译器传递一些额外的标志来生成源映射。源映射允许浏览器将文件的源映射到应用程序中使用的文件。例如，您可以使用 JavaScript 构建工具（如 Webpack）在构建过程中对代码进行缩小。但是，如果您试图查找错误，导航和调试缩小的代码将变得非常困难。通过生成源映射，您可以在浏览器的开发工具中查看原始形式的代码，并设置断点进行调试。让我们为我们的 `/chapter-06-interact-with-js/js-without-glue.cpp` 文件生成一个源映射。在 `/book-examples` 文件夹中，在终端中运行以下命令：

```
emcc chapter-06-interact-with-js/js-without-glue.cpp -O1 -g4  
-s WASM=1 -s SIDE_MODULE=1 -s BINARYEN_ASYNC_COMPILATION=0 -o  
chapter-06-interact-with-js/js-without-glue.wasm --source-map  
-base http://localhost:8080/chapter-06-interact-with-js/
```

- `-g4` 参数启用源映射，而 `-source-map-base` 参数告诉浏览器在哪里找到源映射文件。编译后，通过运行以下命令从 `/book-examples` 文件夹启动本地服务器：

```
serve -l 8080
```

转到 `http://127.0.0.1:8080/chapter-06-interact-with-js/js-without-glue.html`，打开开发者工具，并选择源标签（在 Chrome 中）或调试器标签（在 Firefox 中）。如果您使用 Chrome，您应该会看到以下内容：

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. The left sidebar lists files under '127.0.0.1:8080' and 'wasm'. The 'wasm-c17a6e66-5' file is selected, displaying its assembly code. The code defines a function `func $moveCircle` with local variables `i32 i32`, performing operations like `call`, `drop`, `get_global`, `i32.const`, `i32.add`, `i32.load`, `set_local`, and `end`. The right panel contains sections for 'Watch', 'Call Stack' (labeled 'Not paused'), 'Scope' (labeled 'Not paused'), 'Breakpoints' (labeled 'No breakpoints'), 'XHR/fetch Breakpoints', 'DOM Breakpoints', 'Global Listeners', and 'Event Listener Breakpoints'.

```
1 func $moveCircle
2 (local i32 i32)
3 call 1
4 drop
5 call 3
6 get_global 0
7 i32.const 5242896
8 i32.add
9 i32.load offset=0 align=4
10 set_local 0
11 get_global 0
12 i32.const 5242900
13 i32.add
14 i32.load offset=0 align=4
15 set_local 1
16 get_local 0
17 get_local 1
18 i32.const 50
19 call 2
20 drop
21 end
22
```

Chrome 开发者工具中的 Wasm 源映射

正如您所看到的，文件名并不是很有帮助。每个文件应该在顶部包含函数名称，尽管其中一些名称可能已经被搅乱。如果遇到错误，您可以设置断点，Chrome 的调试功能允许您导航调用堆栈。Firefox 以不同的方式处理它们的源映射。以下截图显示了 Firefox 的开发者工具中的调试器视图：

```

(module
  (type $type0 (func (param i32)))
  (type $type1 (func (result i32)))
  (type $type2 (func (param i32 i32 i32) (result i32)))
  (type $type3 (func))
  (type $type4 (func (result f64)))
  (import "env" "memory" (memory (;0;) 256))
  (import "env" "table" (table $table0 8 anyfunc))
  (import "env" "memoryBase" (global $global0 i32))
  (import "env" "tableBase" (global $global1 i32))
  (import "env" "abort" (func $abort (;0;) (param i32)))
  (import "env" "_jsClearCircle" (func $_jsClearCircle (;1;))
  (import "env" "_jsFillCircle" (func $_jsFillCircle (;2;)
  (global $global2 (mut i32) (i32.const 0))
  (global $global3 (mut i32) (i32.const 0))
  (global $global4 (mut i32) (i32.const 5242896))
  (global $global5 (mut i32) (i32.const 0))
  (global $global6 (mut i32) (i32.const 1))
  (global $global7 (mut i32) (i32.const 3))
  (global $global8 (mut i32) (i32.const 5))
  (global $global9 (mut i32) (i32.const 2))
  (global $global10 (mut i32) (i32.const 4))
  (export "__Z20updateCircleLocationv" (func $__Z20updateCircleLocation))
  (export "__post_instantiate" (func $__post_instantiate))
  (export "_getIsRunning" (func $__getIsRunning))
  (export "_init" (func $__init))
  (export "_moveCircle" (func $__moveCircle))
  (export "_setIsRunning" (func $__setIsRunning))
  (export "runPostSets" (func $runPostSets))
  (export "_circle" (global $global4))
  (export "_isRunning" (global $global5))
  (export "fp$__Z20updateCircleLocationv" (global $global6))
  (export "fp$_getIsRunning" (global $global7))
  (export "fp$__init" (global $global8))
  (export "fp$__moveCircle" (global $global9))
  (export "fp$_setIsRunning" (global $global10))
  (elem (get_global $global1) $b0 $__Z20updateCircleLocation
  (func $__Z20updateCircleLocationv (;3;
  (local $var0 i32) (local $var1 i32)
  (block $label0
    block $label1 (result i32)
    get_global $global0
    i32.const 5242896
    i32.add

```

Firefox 开发者工具中的 Wasm 源映射

源映射是一个包含 Wasm 文件的 Wat 表示的单个文件。您也可以在这里设置断点和调试代码。随着 WebAssembly 的发展，将会有更多（和更好）的工具可用。与此同时，记录到控制台和利用源映射是您可以使用的当前调试方法。

总结

在本章中，我们专注于 JavaScript 和 C/C++之间的互联，Emscripten 提供的一些功能，以及如何有效地调试在浏览器中使用 Wasm 的 Web 应用程序。我们回顾了从 JavaScript 调用编译后的 C/C++函数的各种方法，以及如何将 JavaScript 与您的 C/C++代码集成。Emscripten 的 API 被提出作为一种理解如何通过在编译后的 Wasm 文件中包含粘合代码来克服 WebAssembly 当前限制的方法。即使 Emscripten 提供的功能不在官方的 WebAssembly Core Specification 中（也许永远不会），这也不应该阻止您利用它们。最后，我们简要介绍了如何在浏览器中调试 Wasm 文件，以及 Emscripten 模块或 WebAssembly 实例的上下文。

在下一章中，我们将从头开始构建一个真实的 WebAssembly 应用程序。

问题

1. 您用于与浏览器中的编译代码交互的 `Module` 对象上的两个函数的名称是什么？
2. 您需要用什么来包装您的 C++ 代码，以确保函数名称不会被搅乱？
3. `EM_ASM()` 和 `EM_JS()` 之间有什么区别？
4. `emscripten_run_script()` 和 `EM_ASM()` / `EM_JS()` 中哪个更有效？
5. 如果您想在 C/C++ 代码之外使用它，您需要在函数上面的行中包含什么（提示：它以 `EMSCRIPTEN` 开头）？
6. 在哪里可以定义需要传递到 `importObj.env` 对象中的函数，当实例化模块时？
7. Emscripten 提供了哪些额外的 API？
8. 源映射的目的是什么？

进一步阅读

- Emscripten API 参考：kripken.github.io/emscripten-site/docs/api_reference/index.html
- 源映射简介：blog.teamtreehouse.com/introduction-source-maps
- 使用浏览器调试 WebAssembly：webassemblycode.com/using-browsers-debug-webassembly

第七章：从头开始创建一个应用程序

现在是应用你的知识的时候了！由于 WebAssembly 的主要设计目标之一是在现有的 Web 平台内执行并与之很好地集成，因此构建一个 Web 应用程序来测试它是有意义的。即使 WebAssembly 的当前功能集相当有限，我们仍然可以在基本水平上利用这项技术。在本章中，我们将从头开始构建一个单页应用程序，该应用程序在核心规范的上下文中利用 Wasm 模块。

在本章结束时，您将知道如何：

- 编写使用 C 执行简单计算的函数
- 使用 Vue 构建一个基本的 JavaScript 应用程序
- 将 Wasm 集成到您的 JavaScript 应用程序中
- 确定 WebAssembly 在当前形式下的能力和限制

- 使用 `browser-sync` 运行和测试 JavaScript 应用程序

Cook the Books – 使 WebAssembly 负责

如前所述，WebAssembly 的当前功能集相当有限。我们可以使用 Emscripten 大大扩展 Web 应用程序的功能，但这会带来与官方规范的不兼容以及添加粘合代码的成本。我们仍然可以有效地使用 WebAssembly，这就是我们将在本章中构建的应用程序。在本节中，我们将回顾构建应用程序所使用的库和工具，以及其功能的简要概述。

概述和功能

在 WebAssembly 的当前形式中，我们可以相对容易地在 Wasm 模块和 JavaScript 代码之间传递数字。在现实世界中，会计应用程序似乎是一个合乎逻辑的选择。我对会计软件唯一的争议是它有点无聊（无意冒犯）。我们将通过一些不道德的会计实践来调味一下。该应用程序被命名为 *Cook the Books*，这是与会计欺诈相关的术语。Investopedia 提供了对 Cook the Books 的以下定义：

"Cook the Books 是一个成语，用来描述公司为了伪造其财务报表而进行的欺诈活动。通常，Cook the Books 涉及增加财务数据以产生以前不存在的收益。用于 Cook the Books 的技术示例包括加速收入，延迟支出，操纵养老金计划以及实施合成租赁。"

Investopedia 页面 www.investopedia.com/terms/c/cookthebooks.asp 提供了构成 Cook the Books 的详细示例。我们将为我们的应用程序采取简单的方法。我们将允许用户输入一个交易，包括原始金额和虚假金额。原始金额代表实际存入或取出的金额，而虚假金额是其他人看到的金额。该应用程序将生成显示原始或虚假交易的按类别显示支出和收入的饼图。用户可以轻松地在两种视图之间切换。该应用程序包括以下组件：

- 用于在交易和图表之间切换的选项卡
- 显示交易的表格
- 允许用户添加、编辑或删除交易的按钮
- 用于添加/更新交易的模态对话框
- 显示按类别的收入/支出的饼图

使用的 JavaScript 库

应用程序的 JavaScript 部分将使用从 CDN 提供的几个库。它还将使用一个本地安装的库来监视代码的更改。以下各节将描述每个库及其在应用程序中的目的。

Vue

Vue 是一个 JavaScript 框架，允许您将应用程序拆分为单独的组件，以便于开发和调试。我们使用它来避免一个包含所有应用程序逻辑的单片 JavaScript 文件和另一个包含整个 UI 的单片 HTML 文件。选择 Vue 是因为它不需要构建系统的额外复杂性，并且允许我们在不进行任何转换的情况下使用 HTML、CSS 和 JavaScript。官方网站是 vuejs.org。

UIkit

UIkit 是我们将用来为应用程序添加样式和布局的前端框架。有数十种替代方案，如 Bootstrap 或 Bulma，它们提供了类似的组件和功能。但我选择了 UIkit，因为它具有有用的实用类和附加的 JavaScript 功能。您可以在 getuikit.com 上查看文档。

Lodash

Lodash 是一个出色的实用程序库，提供了在 JavaScript 中执行常见操作的方法，这些方法在语言中尚未内置。我们将使用它来执行计算和操作交易数据。文档和安装说明可以在 lodash.com 找到。

数据驱动文档

数据驱动文档（D3）是一个多功能库，允许您将数据转化为令人印象深刻的可视化效果。D3 的 API 由几个模块组成，从数组操作到图表和过渡。我们将主要使用 D3 来创建饼图，但我们将利用它提供的一些实用方法。您可以在 d3js.org 找到更多信息。

其他库

为了以正确的格式显示货币值并确保用户输入有效的美元金额，我们将利用 **accounting.js** (openexchangerates.github.io/accounting.js) 和 **vue-numeric** (kevinongko.github.io/vue-numeric) 库。为了简化开发，我们将设置一个基本的 **npm** 项目，并使用 **browser-sync** (www.browsersync.io) 来立即看到运行应用程序中的代码更改。

C 和构建过程

该应用程序使用 C，因为我们正在进行基本代数的简单计算。在这种情况下使用 C++ 是没有意义的。这将引入一个额外的步骤，确保我们需要从 JavaScript 调用的函数被包装在 **extern** 块中。我们将在一个单独的 C 文件中编写计算函数，并将其编译成一个单独的

Wasm 模块。我们可以继续使用 VS Code 的任务功能来执行构建，但是参数将需要更新，因为我们只编译一个文件。让我们继续进行项目配置。

项目设置

WebAssembly 还没有存在足够长的时间来建立关于文件夹结构、文件命名约定等方面的最佳实践。如果您搜索 C/C++ 或 JavaScript 项目的最佳实践，您会遇到大量相互矛盾的建议和坚定的观点。考虑到这一点，让我们在本节中花时间设置我们的项目所需的配置文件。

该项目的代码位于 `learn-webassembly` 存储库中的 `/chapter-07-cook-the-books` 文件夹中。当我们进行应用程序的 JavaScript 部分时，您必须拥有此代码。我不会提供书中所有 Vue 组件的源代码，因此您需要从存储库中复制它们。

为 Node.js 配置

为了尽可能保持应用程序的简单性，我们将避免使用 Webpack 或 Rollup.js 等构建/捆绑工具。这样可以减少所需的依赖项数量，并确保您遇到的任何问题都不是由构建依赖项的重大更改引起的。

我们将创建一个 Node.js 项目，因为它允许我们运行脚本并为开发目的本地安装依赖项。到目前为止，我们使用了 `/book-examples` 文件夹，但我们将 `/book-examples` 之外创建一个新的项目文件夹，以配置 VS Code 中不同的默认构建任务。打开终端，`cd` 到所需的文件夹，并输入以下命令：

```
// Create a new directory and cd into it:  
mkdir cook-the-books  
cd cook-the-books  
  
// Create a package.json file with default values  
npm init -y
```

- `y` 命令跳过提示，并使用合理的默认值填充 `package.json` 文件。完成后，运行以下命令安装 `browser-sync`：

```
npm install -D browser-sync@^2.24.4
```

- `D` 是可选的，表示该库是开发依赖项。如果您正在构建和分发应用程序，您将使用 `D` 标志，因此我包含它以遵循常见做法。我建议安装特定版本以确保 `start` 脚本可以正常运行。安装完 `browser-sync` 后，将以下条目添加到 `package.json` 文件中的 `scripts` 条目中：

```
...
"scripts": {
  ...
  "start": "browser-sync start --server \"src\" --files \"src/
**\" --single --no-open --port 4000"
},
...
...
```

如果您使用 `-y` 标志运行 `npm init`，应该会有一个名为 `test` 的现有脚本，为了清晰起见，我省略了它。如果您没有使用 `-y` 标志运行它，您可能需要创建 `scripts` 条目。

如果需要，您可以填写 `"description"` 和 `"author"` 键。文件最终应该看起来类似于这样：

```
{
  "name": "cook-the-books",
  "version": "1.0.0",
  "description": "Example application for Learn WebAssembly",
  "main": "src/index.js",
  "scripts": {
    "start": "browser-sync start --server \"src\" --files \"src/
**\" --single --no-open --port 4000",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "Mike Rourke",
  "license": "MIT",
  "devDependencies": {
    "browser-sync": "2.24.4"
  }
}
```

如果您从 `start` 脚本中省略了 `--no-open` 标志，浏览器将自动打开。该标志被包含在其中以防止用户在无头环境中运行时出现问题。

添加文件和文件夹

在根文件夹中创建两个新文件夹：`/lib` 和 `/src`。JavaScript、HTML、CSS 和 Wasm 文件将位于 `/src` 文件夹中，而 C 文件将位于 `/lib` 文件夹中。我只想在 `/src` 中包含 Web 应用程序使用的文件。我们永远不会直接从应用程序中使用 C 文件，只会使用编译后的输出。

将 `/book-examples` 项目中的 `/.vscode` 文件夹复制到根文件夹中。这将确保您使用现有的 C/C++ 设置，并为构建任务提供一个良好的起点。

如果您使用的是 macOS 或 Linux，您将需要使用终端来复制文件夹；您可以通过运行 `cp -r` 命令来实现这一点。

配置构建步骤

我们需要修改 `/.vscode/tasks.json` 文件中的默认构建步骤，以适应我们更新后的工作流。我们在 `/book-examples` 项目中使用的构建步骤的参数允许我们编译当前在编辑器中活动的任何文件。它还将 `.wasm` 文件输出到与源 C 文件相同的文件夹中。然而，这个配置对于这个项目来说是没有意义的。我们将始终编译相同的 C 文件，并将输出到特定文件夹中的编译后的 `.wasm` 文件。为了实现这一点，在 `/.vscode/tasks.json` 中的 `Build` 任务的 `args` 数组中更新为以下内容：

```
"args": [
    "${workspaceFolder}/lib/main.c",
    "-Os",
    "-s", "WASM=1",
    "-s", "SIDE_MODULE=1",
    "-s", "BINARYEN_ASYNC_COMPILATION=0",
    "-o", "${workspaceFolder}/src/assets/main.wasm"
],
```

我们更改了输入和输出路径，它们是 `args` 数组中的第一个和最后一个元素。现在两者都是静态路径，无论打开的是哪个文件，都会编译和输出相同的文件。

设置模拟 API

我们需要一些模拟数据和一种持久化任何更新的方法。如果您将数据存储在本地的 JSON 文件中，那么您对交易所做的任何更改都将在刷新页面后丢失。我们可以使用 Express 这样的库来设置一个本地服务器，模拟一个数据库，编写路由等等。但是，相反地，我们将利用在线可用的优秀开发工具。在线工具 jsonstore.io 允许您为小型项目存储 JSON 数据，并提供开箱即用的端点。按照以下步骤来启动和运行您的模拟 API：

1. 转到 www.jsonstore.io/ 并点击复制按钮将端点复制到剪贴板；这是您将发出 HTTP 请求的端点。
2. 转到 JSFiddle 网站 jsfiddle.net/mikerourke/cta0km6d/，将您的 jsonstore.io 端点粘贴到输入中，然后按“填充数据”按钮。
3. 打开一个新标签，并在地址栏中粘贴您的 jsonstore.io 端点，然后在 URL 的末尾添加 `/transactions`，然后按Enter。如果您在浏览器中看到 JSON 文件的内容，则 API 设置成功。

将 jsonstore.io 端点保持方便——在构建应用程序的 JavaScript 部分时会用到它。

下载 C stdlib Wasm

我们需要 C 标准库中的 `malloc()` 和 `free()` 函数来实现我们 C 代码中的功能。

WebAssembly 没有内置这些函数，因此我们需要提供自己的实现。

幸运的是，有人已经为我们构建了这个；我们只需要下载模块并将其包含在实例化步骤中。该模块可以从 Guy Bedford 的 [wasm-stdlib-hack](https://github.com/guybedford/wasm-stdlib-hack) GitHub 存储库

github.com/guybedford/wasm-stdlib-hack 中下载。您需要从 `/dist` 文件夹中下载 `memory.wasm` 文件。下载文件后，在项目的 `/src` 文件夹中创建一个名为 `/assets` 的文件夹，并将 `memory.wasm` 文件复制到其中。

您可以从 [learn-webassembly](https://github.com/learn-webassembly/learn-webassembly) 存储库的 `/chapter-07-cook-the-books/src/assets` 文件夹中复制 `memory.wasm` 文件，而不是从 GitHub 上下载它。

最终结果

执行这些步骤后，您的项目应如下所示：

```
└── .vscode
    ├── tasks.json
    └── c_cpp_properties.json
└── /lib
└── /src
```

```
|   └── /assets  
|       └── memory.wasm  
└── package.json  
└── package-lock.json
```

构建 C 部分

应用程序的 C 部分将聚合交易和类别金额。我们在 C 中执行的计算可以很容易地在 JavaScript 中完成，但 WebAssembly 非常适合计算。我们将在第八章《使用 Emscripten 移植游戏》中深入探讨 C/C++ 的更复杂用法，但现在我们试图限制我们的范围，以符合“核心规范”的限制。在本节中，我们将编写一些 C 代码，以演示如何在不使用 Emscripten 的情况下将 WebAssembly 与 Web 应用程序集成。

概述

我们将编写一些 C 函数，用于计算原始和烹饪交易的总额以及结余。除了计算总额外，我们还需要计算每个类别的总额，以在饼图中显示。所有这些计算将在单个 C 文件中执行，并编译为单个 Wasm 文件，该文件将在应用程序加载时实例化。对于未经培训的人来说，C 可能有点令人生畏，因此为了清晰起见，我们的代码将牺牲一些效率。我想抽出一点时间向阅读本书的 C/C++ 程序员道歉；你们可能不会喜欢你们所看到的 C 代码。

为了动态执行计算，我们需要在添加和删除交易时分配和释放内存。为此，我们将使用**双向链表**。双向链表是一种数据结构，允许我们在列表内部删除项目或节点，并根据需要添加和编辑节点。节点使用 `malloc()` 添加，使用 `free()` 删除，这两者都是在上一节中下载的 `memory.wasm` 模块提供的。

关于工作流程的说明

开发操作的顺序并不反映通常构建使用 WebAssembly 的应用程序的方式。工作流程将包括在 C/C++ 和 JavaScript 之间跳转，以实现所需的结果。在这种情况下，我们从 JavaScript 中转移到 WebAssembly 的功能已经知道，因此我们将首先编写 C 代码。

C 文件内容

让我们逐个讨论 C 文件的每个部分。在 `/lib` 文件夹中创建一个名为 `main.c` 的文件，并在每个部分中填充以下内容。如果我们将其分成较小的块，那么更容易理解 C 文件中发生的事情。让我们从**声明部分**开始。

声明

第一部分包含我们将用于创建和遍历双向链表的声明，如下所示：

```
#include <stdlib.h>struct Node {  
    int id;  
    int categoryId;  
    float rawAmount;  
    float cookedAmount;  
    struct Node *next;  
    struct Node *prev;  
};  
  
typedef enum {  
    RAW = 1,  
    COOKED = 2  
} AmountType;  
  
struct Node *transactionsHead = NULL;  
struct Node *categoriesHead = NULL;
```

`Node` 结构用于表示交易或类别。`transactionsHead` 和 `categoriesHead` 节点实例表示我们将使用的每个链表中的第一个节点（一个用于交易，一个用于类别）。`AmountType` 枚举不是必需的，但当我们到达使用它的代码部分时，我们将讨论它的用途。

链表操作

第二部分包含用于向链表中添加和删除节点的两个函数：

```
void deleteNode(struct Node **headNode, struct Node *delNode)  
{  
    // Base case:  
    if (*headNode == NULL || delNode == NULL) return;  
  
    // If node to be deleted is head node:  
    if (*headNode == delNode) *headNode = delNode->next;  
  
    // Change next only if node to be deleted is NOT the last
```

```

node:
    if (delNode->next != NULL) delNode->next->prev = delNode-
>prev;

        // Change prev only if node to be deleted is NOT the firs
t node:
        if (delNode->prev != NULL) delNode->prev->next = delNode-
>next;

        // Finally, free the memory occupied by delNode:
        free(delNode);
}

void appendNode(struct Node **headNode, int id, int categoryId,
                float rawAmount, float cookedAmount) {
    // 1\. Allocate node:
    struct Node *newNode = (struct Node *) malloc(sizeof(stru
ct Node));
    struct Node *last = *headNode; // Used in Step 5

    // 2\. Populate with data:
    newNode->id = id;
    newNode->categoryId = categoryId;
    newNode->rawAmount = rawAmount;
    newNode->cookedAmount = cookedAmount;

    // 3\. This new node is going to be the last node, so mak
e next NULL:
    newNode->next = NULL;

    // 4\. If the linked list is empty, then make the new nod
e as head:
    if (*headNode == NULL) {
        newNode->prev = NULL;
        *headNode = newNode;
}

```

```

        return;
    }

    // 5\. Otherwise, traverse till the last node:
    while (last->next != NULL) {
        last = last->next;
    }

    // 6\. Change the next of last node:
    last->next = newNode;

    // 7\. Make last node as previous of new node:
    newNode->prev = last;
}

```

代码中的注释描述了每个步骤发生的情况。当我们需要向列表中添加一个节点时，我们必须使用 `malloc()` 分配 `struct Node` 占用的内存，并将其附加到链表中的最后一个节点。如果我们需要删除一个节点，我们必须从链表中删除它，并通过调用 `free()` 函数释放节点使用的内存。

交易操作

第三部分包含用于向 `transactions` 链表中添加、编辑和删除交易的函数，如下所示：

```

struct Node *findNodeById(int id, struct Node *withinNode) {
    struct Node *node = withinNode;
    while (node != NULL) {
        if (node->id == id) return node;
        node = node->next;
    }
    return NULL;
}

void addTransaction(int id, int categoryId, float rawAmount,
                    float cookedAmount) {
    appendNode(&transactionsHead, id, categoryId, rawAmount,

```

```

cookedAmount);
}

void editTransaction(int id, int categoryId, float rawAmount,
                     float cookedAmount) {
    struct Node *foundNode = findNodeById(id, transactionsHead);
    if (foundNode != NULL) {
        foundNode->categoryId = categoryId;
        foundNode->rawAmount = rawAmount;
        foundNode->cookedAmount = cookedAmount;
    }
}

void removeTransaction(int id) {
    struct Node *foundNode = findNodeById(id, transactionsHead);
    if (foundNode != NULL) deleteNode(&transactionsHead, foundNode);
}

```

我们在上一部分中审查的 `appendNode()` 和 `deleteNode()` 函数并不打算从 JavaScript 代码中调用。相反，调用 `addTransaction()`、`editTransaction()` 和 `removeTransaction()` 用于更新本地链表。`addTransaction()` 函数调用 `appendNode()` 函数将传递的数据添加到本地链表中的新节点中。`removeTransaction()` 调用 `deleteNode()` 函数删除相应的交易节点。`findNodeById()` 函数用于根据指定的 ID 确定需要在链表中更新或删除的节点。

交易计算

第四部分包含用于计算原始和处理后 `transactions` 的总额和最终余额的函数，如下所示：

```

void calculateGrandTotals(float *totalRaw, float *totalCooked) {
    struct Node *node = transactionsHead;
    while (node != NULL) {
        *totalRaw += node->rawAmount;

```

```

        *totalCooked += node->cookedAmount;
        node = node->next;
    }
}

float getGrandTotalForType(AmountType type) {
    float totalRaw = 0;
    float totalCooked = 0;
    calculateGrandTotals(&totalRaw, &totalCooked);

    if (type == RAW) return totalRaw;
    if (type == COOKED) return totalCooked;
    return 0;
}

float getFinalBalanceForType(AmountType type, float initialBalance) {
    float totalForType = getGrandTotalForType(type);
    return initialBalance + totalForType;
}

```

我们在声明部分中声明的 `AmountType enum` 在这里用于避免魔术数字。这使得很容易记住 1 代表原始交易，2 代表处理后的交易。原始和处理后的交易的总额都是在 `calculateGrandTotals()` 函数中计算的，即使在 `getGrandTotalForType()` 中只请求一个类型。由于我们只能从 Wasm 函数中返回一个值，当我们为原始和处理后的交易都调用 `getGrandTotalForType()` 时，我们最终会循环遍历所有交易两次。对于相对较少的交易量和计算的简单性，这并不会产生任何问题。`getFinalBalanceForType()` 返回指定 `initialBalance` 加上总额。当我们在 Web 应用程序中添加更改初始余额的功能时，您将看到这一点。

类别计算

第五和最后一部分包含用于按类别计算总额的函数，我们将在饼图中使用，如下所示：

```

void upsertCategoryNode(int categoryId, float transactionRaw,
                      float transactionCooked) {
    struct Node *foundNode = findNodeById(categoryId, categor

```

```
iesHead);
    if (foundNode != NULL) {
        foundNode->rawAmount += transactionRaw;
        foundNode->cookedAmount += transactionCooked;
    } else {
        appendNode(&categoriesHead, categoryId, categoryId, t
ransactionRaw,
                    transactionCooked);
    }
}

void buildValuesByCategoryList() {
    struct Node *node = transactionsHead;
    while (node != NULL) {
        upsertCategoryNode(node->categoryId, node->rawAmount,
                            node->cookedAmount);
        node = node->next;
    }
}

void recalculateForCategories() {
    categoriesHead = NULL;
    buildValuesByCategoryList();
}

float getCategoryTotal(AmountType type, int categoryId) {
    // Ensure the category totals have been calculated:
    if (categoriesHead == NULL) buildValuesByCategoryList();

    struct Node *categoryNode = findNodeById(categoryId, cate
goriesHead);
    if (categoryNode == NULL) return 0;

    if (type == RAW) return categoryNode->rawAmount;
    if (type == COOKED) return categoryNode->cookedAmount;
```

```
    return 0;  
}
```

每当调用 `recalculateForCategories()` 或 `getCategoryTotal()` 函数时，都会调用 `buildValuesByCategoryList()` 函数。该函数循环遍历 `transactions` 链表中的所有交易，并为每个对应的类别创建一个节点，其中包含聚合的原始和总金额。`upsertCategoryNode()` 函数在 `categories` 链表中查找与 `categoryId` 对应的节点。如果找到，则将原始和处理后的交易金额添加到该节点上的现有金额中，否则为该类别创建一个新节点。调用 `recalculateForCategories()` 函数以确保类别总额与任何交易更改保持最新。

编译为 Wasm

填充文件后，我们需要将其编译为 Wasm，以便在应用程序的 JavaScript 部分中使用。通过从菜单中选择任务 | 运行构建任务... 或使用键盘快捷键 *Cmd/Ctrl + Shift + B* 来运行构建任务。如果构建成功，您将在 `/src/assets` 文件夹中看到一个名为 `main.wasm` 的文件。如果出现错误，终端应提供有关如何解决错误的详细信息。

如果您没有使用 VS Code，请在 `/cook-the-books` 文件夹中打开终端实例，并运行以下命令：

```
emcc lib/main.c -Os -s WASM=1 -s SIDE_MODULE=1 -s BINARYEN_ASYNC_COMPILATION=0 -o src/assets/main.wasm
```

C 代码就是这样。让我们继续进行 JavaScript 部分。

构建 JavaScript 部分

应用程序的 JavaScript 部分向用户呈现交易数据，并允许他们轻松添加、编辑和删除交易。该应用程序分为几个文件，以简化开发过程，并使用本章节中描述的库。在本节中，我们将逐步构建应用程序，从 API 和全局状态交互层开始。我们将编写函数来实例化和与我们的 Wasm 模块交互，并审查构建用户界面所需的 Vue 组件。

概述

该应用程序被分解为上下文，以简化开发过程。我们将从底层开始构建应用程序，以确保在编写代码时不必在不同的上下文之间来回跳转。我们将从 Wasm 交互代码开始，然后转向全局存储和 API 交互。我将描述每个 Vue 组件的目的，但只会为少数几个提供源代码。如果您正在跟随并希望在本地运行应用程序，则需要将 `learn-webassembly` 存储库

中 `/chapter-07-cook-the-books` 文件夹中的 `/src/components` 文件夹复制到您的项目的 `/src` 文件夹中。

关于浏览器兼容性的说明

在我们开始编写任何代码之前，您必须确保您的浏览器支持我们将在应用程序中使用的较新的 JavaScript 功能。您的浏览器必须支持 ES 模块（`import` 和 `export`）、Fetch API 和 `async / await`。您至少需要 Google Chrome 的版本 61 或 Firefox 的版本 60。您可以通过从菜单栏中选择关于 Chrome 或关于 Firefox 来检查您当前使用的版本。我目前正在使用 Chrome 版本 67 和 Firefox 版本 61 运行应用程序，没有任何问题。

在 `initializeWasm.js` 中创建一个 Wasm 实例

您的项目的 `/src/assets` 文件夹中应该有两个编译好的 Wasm 文件：`main.wasm` 和 `memory.wasm`。由于我们需要在 `main.wasm` 代码中使用从 `memory.wasm` 导出的 `malloc()` 和 `free()` 函数，我们的加载代码将与之前的示例有所不同。在 `/src/store` 文件夹中创建一个名为 `initializeWasm.js` 的文件，并填充以下内容：

```
/***
 * Returns an array of compiled (not instantiated!) Wasm modules.
 * We need the main.wasm file we created, as well as the memory.wasm file
 * that allows us to use C functions like malloc() and free().
 */
const fetchAndCompileModules = () =>
  Promise.all(
    ['./assets/main.wasm', './assets/memory.wasm'].map(fileName =>
      fetch(fileName)
        .then(response => {
          if (response.ok) return response.arrayBuffer();
          throw new Error(`Unable to fetch WebAssembly file: ${fileName}`);
        })
        .then(bytes => WebAssembly.compile(bytes))
    )
  )
}

// Create a function to load the Wasm modules
// and instantiate them
const loadModules = () =>
```

```
        )
    );

/***
 * Returns an instance of the compiled "main.wasm" file.
 */
const instantiateMain = (compiledMain, memoryInstance, wasmMemory) => {
    const memoryMethods = memoryInstance.exports;
    return WebAssembly.instantiate(compiledMain, {
        env: {
            memoryBase: 0,
            tableBase: 0,
            memory: wasmMemory,
            table: new WebAssembly.Table({ initial: 16, element: 'anyfunc' }),
            abort: console.log,
            _consoleLog: value => console.log(value),
            _malloc: memoryMethods.malloc,
            _free: memoryMethods.free
        }
    });
};

/***
 * Compiles and instantiates the "memory.wasm" and "main.wasm" files and
 * returns the `exports` property from main's `instance`.
 */
export default async function initializeWasm() {
    const wasmMemory = new WebAssembly.Memory({ initial: 1024 });
    const [compiledMain, compiledMemory] = await fetchAndCompileModules();

    const memoryInstance = await WebAssembly.instantiate(compil
```

```

    edMemory, {
      env: {
        memory: wasmMemory
      }
    });

    const mainInstance = await instantiateMain(
      compiledMain,
      memoryInstance,
      wasmMemory
    );

    return mainInstance.exports;
}

```

文件的默认 `export` 函数 `initializeWasm()` 执行以下步骤：

1. 创建一个新的 `WebAssembly.Memory` 实例（`wasmMemory`）。
2. 调用 `fetchAndCompileModules()` 函数以获取 `memory.wasm`（`compiledMemory`）和 `main.wasm`（`compiledMain`）的 `WebAssembly.Module` 实例。
3. 实例化 `compiledMemory`（`memoryInstance`）并将 `wasmMemory` 传递给 `importObj`。
4. 将 `compiledMain`、`memoryInstance` 和 `wasmMemory` 传递给 `instantiateMain()` 函数。
5. 实例化 `compiledMain` 并将从 `memoryInstance` 导出的 `malloc()` 和 `free()` 函数以及 `wasmMemory` 传递给 `importObj`。
6. 返回从 `instantiateMain` 返回的 `Instance` 的 `exports` 属性。

如您所见，当 Wasm 模块内部存在依赖关系时，该过程更加复杂。

您可能已经注意到 `memoryInstance` 的 `exports` 属性上的 `malloc` 和 `free` 方法没有用下划线前缀。这是因为 `memory.wasm` 文件是使用 LLVM 而不是 Emscripten 编译的，后者不会添加下划线。

在 `WasmTransactions.js` 中与 Wasm 交互

我们将使用 JavaScript 的 `class` 语法来创建一个封装 Wasm 交互函数的包装器。这使我们能够快速更改 C 代码，而无需搜索整个应用程序以找到调用 Wasm 函数的位置。如果

您在 C 文件中重命名一个方法，您只需要在一个地方重命名它。在 `/src/store` 文件夹中创建一个名为 `WasmTransactions.js` 的新文件，并填充以下内容：

```
import initializeWasm from './initializeWasm.js';

/**
 * Class used to wrap the functionality from the Wasm module
(rather
 * than access it directly from the Vue components or store).
 * @class
 */
export default class WasmTransactions {
  constructor() {
    this.instance = null;
    this.categories = [];
  }

  async initialize() {
    this.instance = await initializeWasm();
    return this;
  }

  getCategoryById(category) {
    return this.categories.indexOf(category);
  }

  // Ensures the raw and cooked amounts have the proper sign
(withdrawals
  // are negative and deposits are positive).
  getValidAmounts(transaction) {
    const { rawAmount, cookedAmount, type } = transaction;
    const getAmount = amount =>
      type === 'Withdrawal' ? -Math.abs(amount) : amount;
    return {
      validRaw: getAmount(rawAmount),
      validCooked: getAmount(cookedAmount)
    };
  }
}
```

```
    };
}

// Adds the specified transaction to the linked list in the
// Wasm module.
addToWasm(transaction) {
    const { id, category } = transaction;
    const { validRaw, validCooked } = this.getValidAmounts(tr
ansaction);
    const categoryId = this.getCategoryId(category);
    this.instance._addTransaction(id, categoryId, validRaw, v
alidCooked);
}

// Updates the transaction node in the Wasm module:
editInWasm(transaction) {
    const { id, category } = transaction;
    const { validRaw, validCooked } = this.getValidAmounts(tr
ansaction);
    const categoryId = this.getCategoryId(category);
    this.instance._editTransaction(id, categoryId, validRaw,
validCooked);
}

// Removes the transaction node from the linked list in the
// Wasm module:
removeFromWasm(transactionId) {
    this.instance._removeTransaction(transactionId);
}

// Populates the linked list in the Wasm module. The catego
ries are
// needed to set the categoryId in the Wasm module.
populateInWasm(transactions, categories) {
    this.categories = categories;
    transactions.forEach(transaction => this.addToWasm(transa
```

```
ction));
}

// Returns the balance for raw and cooked transactions base
d on the
// specified initial balances.
getCurrentBalances(initialRaw, initialCooked) {
    const currentRaw = this.instance._getFinalBalanceForType(
        AMOUNT_TYPE.raw,
        initialRaw
    );
    const currentCooked = this.instance._getFinalBalanceForTy
pe(
        AMOUNT_TYPE.cooked,
        initialCooked
    );
    return { currentRaw, currentCooked };
}

// Returns an object that has category totals for all incom
e (deposit)
// and expense (withdrawal) transactions.
getCategoryTotals() {
    // This is done to ensure the totals reflect the most rec
ent
    // transactions:
    this.instance._recalculateForCategories();
    const categoryTotals = this.categories.map((category, id
x) => ({
        category,
        id: idx,
        rawTotal: this.instance._getCategoryTotal(AMOUNT_TYPE.r
aw, idx),
        cookedTotal: this.instance._getCategoryTotal(AMOUNT_TYP
E.cooked, idx)
    }));
}
```

```

        const totalsByGroup = { income: [], expenses: [] };
        categoryTotals.forEach(categoryTotal => {
            if (categoryTotal.rawTotal < 0) {
                totalsByGroup.expenses.push(categoryTotal);
            } else {
                totalsByGroup.income.push(categoryTotal);
            }
        });
        return totalsByGroup;
    }
}

```

当对类的实例调用 `initialize()` 函数时，`initializeWasm()` 函数的返回值被分配给类的 `instance` 属性。`class` 方法调用 `this.instance` 中的函数，并在适用的情况下返回所需的结果。请注意 `getCurrentBalances()` 和 `getCategoryTotals()` 函数中引用的 `AMOUNT_TYPE` 对象。这对应于我们 C 文件中的 `AmountType enum`。`AMOUNT_TYPE` 对象在加载应用程序的 `/src/main.js` 文件中全局声明。现在我们已经编写了 Wasm 交互代码，让我们继续编写 API 交互代码。

在 `api.js` 中利用 API

API 提供了在 `fetch` 调用上定义的 HTTP 方法的方式来添加、编辑、删除和查询交易。为了简化执行这些操作的过程，我们将编写一些 API“包装”函数。在 `/src/store` 文件夹中创建一个名为 `api.js` 的文件，并填充以下内容：

```

// Paste your jsonstore.io endpoint here (no ending slash):
const API_URL = '[JSONSTORE.IO ENDPOINT]';

/**
 * Wrapper for performing API calls. We don't want to call re
 * sponse.json()
 * each time we make a fetch call.
 * @param {string} endpoint Endpoint (e.g. "/transactions" to
 * make API call to
 * @param {Object} init Fetch options object containing any c

```

```

ustom settings
 * @returns {Promise<*>}
 * @see https://developer.mozilla.org/en-US/docs/Web/API/Wind
owOrWorkerGlobalScope/fetch
 */
const performApiFetch = (endpoint = '', init = {}) =>
  fetch(` ${API_URL}${endpoint}`, {
    headers: {
      'Content-type': 'application/json'
    },
    ...init
  }).then(response => response.json());

export const apiFetchTransactions = () =>
  performApiFetch('/transactions').then(({ result }) =>
  /*
   * The response object looks like this:
   * {
   *   "result": {
   *     "1": {
   *       "category": "Sales Revenue",
   *       ...
   *     },
   *     "2": {
   *       "category": "Hotels",
   *       ...
   *     },
   *     ...
   *   }
   * }
   * We need the "1" and "2" values for deleting or editing
existing
   * records, so we store that in the transaction record as
"apiId".
 */
  Object.keys(result).map(apiId => ({

```

```

    ...result[apiId],
    apiId
  }))  

);  
  

export const apiEditTransaction = transaction =>  

  performApiFetch(`/transactions/${transaction.apiId}`, {  

    method: 'POST',  

    body: JSON.stringify(transaction)  

});  
  

export const apiRemoveTransaction = transaction =>  

  performApiFetch(`/transactions/${transaction.apiId}`, {  

    method: 'DELETE'  

});  
  

export const apiAddTransaction = transaction =>  

  performApiFetch(`/transactions/${transaction.apiId}`, {  

    method: 'POST',  

    body: JSON.stringify(transaction)  

});

```

您需要在设置项目部分创建的 jsonstore.io 端点才能与 API 交互。将 `[JSONSTORE.IO ENDPOINT]` 替换为您的 jsonstore.io 端点。确保端点不以斜杠或单词 transactions 结尾。

在 `store.js` 中管理全局状态

在应用程序中管理全局状态的文件有很多组成部分。因此，我们将代码分解成较小的块，并逐个部分地进行讲解。在 `/src/store` 文件夹中创建一个名为 `store.js` 的文件，并填充以下各部分的内容。

导入和存储声明

第一部分包含 `import` 语句和导出的 `store` 对象上的 `wasm` 和 `state` 属性，如下所示：

```

import {  

  apiFetchTransactions,  

}

```

```
apiAddTransaction,
apiEditTransaction,
apiRemoveTransaction
} from './api.js';
import WasmTransactions from './WasmTransactions.js';

export const store = {
  wasm: null,
  state: {
    transactions: [],
    activeTransactionId: 0,
    balances: {
      initialRaw: 0,
      currentRaw: 0,
      initialCooked: 0,
      currentCooked: 0
    }
  },
  ...
}
```

所有 API 交互都限于 `store.js` 文件。由于我们需要操作、添加和搜索交易，所以从 `api.js` 导出的所有函数都被导入。`store` 对象在 `wasm` 属性中保存了 `WasmTransactions` 实例，并在 `state` 属性中保存了初始状态。`state` 中的值在应用程序的多个位置引用。当应用程序加载时，`store` 对象将被添加到全局 `window` 对象中，因此所有组件都可以访问全局状态。

交易操作

第二部分包含管理 Wasm 实例（通过 `WasmTransactions` 实例）和 API 中的交易的函数，如下所示：

```
...
getCategories() {
  const categories = this.state.transactions.map(
    ({ category }) => category
);
```

```

        // Remove duplicate categories and sort the names in ascending order:
        return _.uniq(categories).sort();
    },

    // Populate global state with the transactions from the API response:
    populateTransactions(transactions) {
        const sortedTransactions = _.sortBy(transactions, [
            'transactionDate',
            'id'
        ]);
        this.state.transactions = sortedTransactions;
        store.wasm.populateInWasm(sortedTransactions, this.getCategories());
        this.recalculateBalances();
    },

    addTransaction(newTransaction) {
        // We need to assign a new ID to the transaction, so this just adds
        // 1 to the current maximum transaction ID:
        newTransaction.id = _.maxBy(this.state.transactions, 'id').id + 1;
        store.wasm.addTowasm(newTransaction);
        apiAddTransaction(newTransaction).then(() => {
            this.state.transactions.push(newTransaction);
            this.hideTransactionModal();
        });
    },

    editTransaction(editedTransaction) {
        store.wasm.editInWasm(editedTransaction);
        apiEditTransaction(editedTransaction).then(() => {
            this.state.transactions = this.state.transactions.map(
                transaction => {

```

```

        if (transaction.id === editedTransaction.id) {
            return editedTransaction;
        }
        return transaction;
    }
);
this.hideTransactionModal();
});
},
removeTransaction(transaction) {
    const transactionId = transaction.id;
    store.wasm.removeFromWasm(transactionId);

    // We're passing the whole transaction record into the API call
    // for the sake of consistency:
    apiRemoveTransaction(transaction).then(() => {
        this.state.transactions = this.state.transactions.filter(
            ({ id }) => id !== transactionId
        );
        this.hideTransactionModal();
    });
},
...

```

`populateTransactions()` 函数从 API 中获取所有交易，并将它们加载到全局状态和 Wasm 实例中。类别名称是从 `getCategories()` 函数中的 `transactions` 数组中推断出来的。当调用 `store.wasm.populateInWasm()` 时，结果将传递给 `WasmTransactions` 实例。

`addTransaction()`、`editTransaction()` 和 `removeTransaction()` 函数执行与它们的名称相对应的操作。所有三个函数都操作 Wasm 实例，并通过 `fetch` 调用更新 API 上的数据。每个函数都调用 `this.hideTransactionModal()`，因为只能通过 `TransactionModal` 组件对交易进行更改。一旦更改成功，模态应该关闭。接下来让我们看一下 `TransactionModal` 管理代码。

交易模态管理

第三部分包含管理 `TransactionModal` 组件（位于 `/src/components/TransactionsTab/TransactionModal.js`）的可见性和内容的函数，如下所示：

```
...
  showTransactionModal(transactionId) {
    this.state.activeTransactionId = transactionId || 0;
    const transactModal = document.querySelector('#transactionModal');
    UIkit.modal(transactModal).show();
  },

  hideTransactionModal() {
    this.state.activeTransactionId = 0;
    const transactModal = document.querySelector('#transactionModal');
    UIkit.modal(transactModal).hide();
  },

  getActiveTransaction() {
    const { transactions, activeTransactionId } = this.state;
    const foundTransaction = transactions.find(transaction =>
      transaction.id === activeTransactionId);
    return foundTransaction || { id: 0 };
  },
...
}
```

`showTransactionModal()` 和 `hideTransactionModal()` 函数应该是不言自明的。在代表 `TransactionModal` 的 DOM 元素上调用 `UIkit.modal()` 的 `hide()` 或 `show()` 方法。
`getActiveTransaction()` 函数返回与全局状态中的 `activeTransactionId` 值相关联的交易记录。

余额计算

第四部分包含计算和更新全局状态中 `balances` 对象的函数：

```
...
    updateInitialBalance(amount, fieldName) {
        this.state.balances[fieldName] = amount;
    },

    // Update the "balances" object in global state based on the current
    // initial balances:
    recalculateBalances() {
        const { initialRaw, initialCooked } = this.state.balances;
        const { currentRaw, currentCooked } = this.wasm.getCurrentBalances(
            initialRaw,
            initialCooked
        );
        this.state.balances = {
            initialRaw,
            currentRaw,
            initialCooked,
            currentCooked
        };
    }
};
```

`updateInitialBalance()` 函数根据 `amount` 和 `fieldName` 参数设置全局状态中 `balances` 对象的属性值。`recalculateBalances()` 函数更新 `balances` 对象上的所有字段，以反映对初始余额或交易所做的任何更改。

存储初始化

文件中的最后一部分代码初始化了存储：

```
/**
 * This function instantiates the Wasm module, fetches the transactions
```

```
* from the API endpoint, and loads them into state and the W
asm
* instance.
*/
export const initializeStore = async () => {
  const wasmTransactions = new WasmTransactions();
  store.wasm = await wasmTransactions.initialize();
  const transactions = await apiFetchTransactions();
  store.populateTransactions(transactions);
};
```

`initializeStore()` 函数实例化 Wasm 模块，从 API 获取所有交易，并填充状态的内容。这个函数是从 `/src/main.js` 中的应用程序加载代码中调用的，我们将在下一节中介绍。

在 main.js 中加载应用程序

我们需要一个入口点来加载我们的应用程序。在 `/src` 文件夹中创建一个名为 `main.js` 的文件，并填充以下内容：

```
import App from './components/App.js';
import { store, initializeStore } from './store/store.js';

// This allows us to use the <vue-numeric> component globally:
Vue.use(VueNumeric.default);

// Create a globally accessible store (without having to pass it down
// as props):
window.$store = store;

// Since we can only pass numbers into a Wasm function, these flags
// represent the amount type we're trying to calculate:
window.AMOUNT_TYPE = {
  raw: 1,
```

```
        cooked: 2
    };

    // After fetching the transactions and initializing the Wasm
    // module,
    // render the app.
    initializeStore()
        .then(() => {
            new Vue({ render: h => h(App), el: '#app' });
        })
        .catch(err => {
            console.error(err);
        });
}
```

这个文件是在从 `/src/index.html` 中的 CDN 中获取和加载库之后加载的。我们使用全局的 `Vue` 对象来指定我们要使用 `VueNumeric` 组件。我们将从 `/store/store.js` 导出的 `store` 对象添加到 `window` 中作为 `$store`。这不是最健壮的解决方案，但在应用程序的范围内将足够。如果你正在创建一个生产应用程序，你会使用像 **Vuex** 或 **Redux** 这样的库来进行全局状态管理。出于简化的目的，我们将放弃这种方法。

我们还将 `AMOUNT_TYPE` 添加到 `window` 对象中。这样做是为了确保整个应用程序可以引用 `AMOUNT_TYPE` 值，而不是指定一个魔术数字。在将值分配给 `window` 之后，将调用 `initializeStore()` 函数。如果 `initializeStore()` 函数成功触发，将创建一个新的 `Vue` 实例来渲染应用程序。接下来让我们添加 web 资源，然后转向 `Vue` 组件。

添加 web 资源

在我们开始向应用程序添加 `Vue` 组件之前，让我们创建包含我们标记和样式的 HTML 和 CSS 文件。在 `/src` 文件夹中创建一个名为 `index.html` 的文件，并填充以下内容：

```
<!doctype html>
<html lang="en-us">
<head>
    <title>Cook the Books</title>
    <link
        rel="stylesheet"
```

```
    type="text/css"
    href="https://cdnjs.cloudflare.com/ajax/libs/uikit/3.0.0-
rc.6/css/uikit.min.css"
/>
<link rel="stylesheet" type="text/css" href="styles.css" />
<script src="img/uikit.min.js"></script>
<script src="img/uikit-icons.min.js"></script>
<script src="img/accounting.umd.js"></script>
<script src="img/lodash.min.js"></script>
<script src="img/d3.min.js"></script>
<script src="img/vue.min.js"></script>
<script src="img/vue-numeric.min.js"></script>
<script src="img/main.js" type="module"></script>
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

我们只使用 HTML 文件从 CDN 中获取库，指定 Vue 可以渲染的 `<div>`，并加载 `main.js` 来启动应用程序。请注意最后一个 `<script>` 元素上的 `type="module"` 属性。这允许我们在整个应用程序中使用 ES 模块。现在让我们添加 CSS 文件。在 `/src` 文件夹中创建一个名为 `styles.css` 的文件，并填充以下内容：

```
@import url("https://fonts.googleapis.com/css?family=Quicksan
d");
:root {
  --blue: #2889ed;
}

* {
  font-family: "Quicksand", Helvetica, Arial, sans-serif !impo
rtant;
}
```

```
#app {
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

.addTransactionButton {
  color: white;
  height: 64px;
  width: 64px;
  background: var(--blue);
  position: fixed;
  bottom: 24px;
  right: 24px;
}

.addTransactionButton:hover {
  color: white;
  background-color: var(--blue);
  opacity: .6;
}

.errorText {
  color: white;
  font-size: 36px;
}

.appHeader {
  height: 80px;
  margin: 0;
}

.balanceEntry {
  font-size: 2rem;
}

.tableAmount {
```

```
white-space: pre;  
}
```

这个文件只有几个类，因为大部分的样式将在组件级别处理。在下一节中，我们将回顾构成我们应用程序的 Vue 组件。

创建 Vue 组件

使用 Vue，我们可以创建单独的组件，封装其自身的功能，然后组合这些组件来构建应用程序。这比将应用程序存储在单个庞大文件中更容易进行调试、扩展和变更管理。

该应用程序使用单文件组件开发方法。在开始审查组件文件之前，让我们看看最终产品。以下屏幕截图显示了选择了 TRANSACTIONS 选项卡的应用程序：

The screenshot shows a web browser window for 'Cook the Books' application at 'localhost:4000'. The title bar says 'Cook the Books' and 'Guest'. The main content area has a blue header 'Cook the Books'. Below it, there are two tabs: 'TRANSACTIONS' (which is underlined and highlighted) and 'CHARTS'. The main content area displays financial data in four boxes:

INITIAL RAW BALANCE	CURRENT RAW BALANCE	INITIAL COOKED BALANCE	CURRENT COOKED BALANCE
\$ 0.00	\$185,515.00	\$ 0.00	\$247,531.00

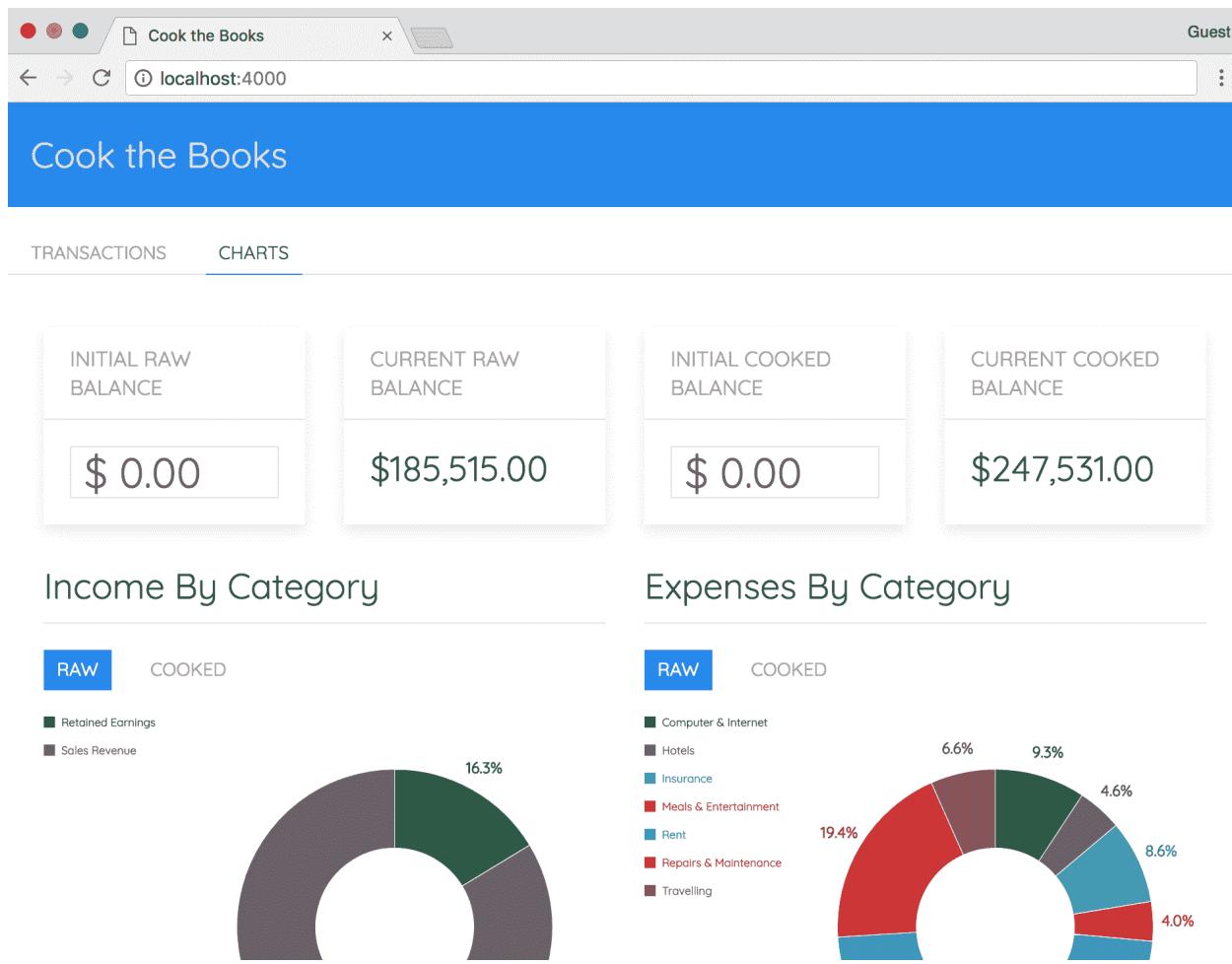
Below this, a table lists transaction history:

DATE	TYPE	TO/FROM	CATEGORY	RAW AMOUNT	COOKED AMOUNT
08/11/2017	Deposit	Limin Zhu	Sales Revenue	\$ 5,904.00	\$ 7,250.00
08/14/2017	Deposit	Benjamin Bouvier	Sales Revenue	\$ 5,390.00	\$ 5,805.00
08/16/2017	Withdrawal	Uncle Buck's Computer Warehouse	Computer & Internet	\$ (350.00)	\$ (250.00)
09/01/2017	Withdrawal	The Food Restaurant	Meals &	\$ (127.00)	\$ (102.00)

A large blue circular button with a white plus sign is located in the bottom right corner of the transaction table.

使用 TRANSACTIONS 选项卡运行应用程序

以下是应用程序的屏幕截图，选择了 CHARTS 选项卡：



使用 CHARTS 选项卡运行应用程序

Vue 组件的结构

Vue 组件只是一个包含属性的导出对象文件，定义了该组件的外观和行为。这些属性必须具有符合 Vue API 的名称。您可以在 vuejs.org/v2/api 上阅读有关这些属性和 Vue API 的其他方面。以下代码代表包含此应用程序中使用的 Vue API 元素的示例组件：

```
import SomeComponent from './SomeComponent.js';

export default {
  name: 'dummy-component',
```

```
// Props passed from other components:  
props: {  
    label: String,  
},  
  
// Other Vue components to render within the template:  
components: {  
    SomeComponent  
,  
  
    // Used to store local data/state:  
    data() {  
        return {  
            amount: 0  
        }  
    },  
  
    // Used to store complex logic that outside of the `template`:  
    computed: {  
        negativeClass() {  
            return {  
                'negative': this.amount < 0  
            };  
        }  
    },  
  
    // Methods that can be performed within the component:  
    methods: {  
        addOne() {  
            this.amount += 1;  
        }  
    },  
  
    // Perform actions if the local data changes:  
    watch: {
```

```

        amount(val, oldVal) {
            console.log(`New: ${val} | Old: ${oldVal}`);
        }
    },

// Contains the HTML to render the component:
template: `
<div>
    <some-component></some-component>
    <label for="someAmount">{{ label }}</label>
    <input
        id="someAmount"
        :class="negativeClass"
        v-model="amount"
        type="number"
    />
    <button @click="addOne">Add One</button>
</div>
`,

};


```

上面每个属性的注释描述了其目的，尽管在非常高的层次上。让我们通过审查 `App` 组件来看看 Vue 的实际运行情况。

App 组件

`App` 组件是渲染应用程序中所有子组件的基本组件。我们将简要审查 `App` 组件的代码，以更好地理解 Vue。接下来，我们将描述每个剩余组件的作用，但只审查相应代码的部分。`App` 组件文件的内容，位于 `/src/components/App.js`，如下所示：

```

import BalancesBar from './BalancesBar/BalancesBar.js';
import ChartsTab from './ChartsTab/ChartsTab.js';
import TransactionsTab from './TransactionsTab/TransactionsTab.js';

/***

```

```
* This component is the entry point for the application. It
contains the
* header, tabs, and content.
*/
export default {
  name: 'app',
  components: {
    BalancesBar,
    ChartsTab,
    TransactionsTab
  },
  data() {
    return {
      balances: $store.state.balances,
      activeTab: 0
    };
  },
  methods: {
    // Any time a transaction is added, edited, or removed, we
    // need to
    // ensure the balance is updated:
    onTransactionChange() {
      $store.recalculateBalances();
      this.balances = $store.state.balances;
    },
    // When the "Charts" tab is activated, this ensures that
    // the charts
    // get automatically updated:
    onTabClick(event) {
      this.activeTab = +event.target.dataset.tab;
    }
  },
  template: `
<div>
  <div class="appHeader uk-background-primary uk-flex uk-
```

```
flex-middle">
    <h2 class="uk-light uk-margin-remove-bottom uk-margin-left">
        Cook the Books
    </h2>
</div>
<div class="uk-position-relative">
    <ul uk-tab class="uk-margin-small-bottom uk-margin-top">
        <li class="uk-margin-small-left">
            <a href="#" data-tab="0" @click="onTabClick">Transactions</a>
        </li>
        <li>
            <a href="#" data-tab="1" @click="onTabClick">Charts</a>
        </li>
    </ul>
    <balances-bar
        :balances="balances"
        :onTransactionChange="onTransactionChange">
    </balances-bar>
    <ul class="uk-switcher">
        <li>
            <transactions-tab :onTransactionChange="onTransactionChange">
                </transactions-tab>
            </li>
            <li>
                <charts-tab :isActive="this.activeTab === 1"></charts-tab>
            </li>
    </ul>
</div>
</div>
```

```
};
```

我们使用 `components` 属性指定在 `App` 组件的 `template` 中渲染的其他 Vue 组件。`data()` 函数返回本地状态，用于跟踪余额和活动的选项卡（TRANSACTIONS 或 CHARTS）。

`methods` 属性包含两个函数：`onTransactionChange()` 和 `onTabClick()`。`onTransactionChange()` 函数调用 `$store.recalculateBalances()`，如果对交易记录进行更改，则更新本地状态中的 `balances`。`onTabClick()` 函数将本地状态中的 `activeTab` 值更改为所点击选项卡的 `data-tab` 属性。最后，`template` 属性包含用于渲染组件的标记。

如果您在 Vue 中不使用单文件组件（.vue 扩展名），则需要将模板属性中的组件名称转换为 kebab case。例如，在前面显示的 `App` 组件中，`BalancesBar` 被更改为 `<balances-bar>`。

BalancesBar

`/components/BalancesBar` 文件夹包含两个组件文件：`BalanceCard.js` 和 `BalancesBar.js`。`BalancesBar` 组件跨越 TRANSACTIONS 和 CHARTS 选项卡，并直接位于选项卡控制下方。它包含四个 `BalanceCard` 组件，分别对应四种余额类型：初始原始、当前原始、初始熟练和当前熟练。代表初始余额的第一和第三张卡包含输入，因此余额可以更改。代表当前余额的第二和第四张卡在 Wasm 模块中动态计算（使用 `getFinalBalanceForType()` 函数）。以下代码片段来自 `BalancesBar` 组件，演示了 Vue 的绑定语法：

```
<balance-card
  title="Initial Raw Balance"
  :value="balances.initialRaw"
  :onChange="amount => onBalanceChange(amount, 'initialRaw')"
>
</balance-card>
```

`value` 和 `onChange` 属性之前的 `:` 表示这些属性绑定到了 Vue 组件。如果 `balances.initialRaw` 的值发生变化，`BalanceCard` 中显示的值也会更新。此卡的 `onBalanceChange()` 函数会更新全局状态中 `balances.initialRaw` 的值。

TransactionsTab

`/components/TransactionsTab` 文件夹包含以下四个组件文件：

- `ConfirmationModal.js`
- `TransactionModal.js`
- `TransactionsTab.js`
- `TransactionsTable.js`

`TransactionsTab` 组件包含 `TransactionsTable` 和 `TransactionsModal` 组件，以及用于添加新交易的按钮。更改和添加是通过 `TransactionModal` 组件完成的。`TransactionsTable` 包含所有当前的交易，每行都有按钮，可以编辑或删除交易。如果用户按下删除按钮，`ConfirmationModal` 组件将出现并提示用户继续。如果用户按下“是”，则删除交易。以下摘录来自 `TransactionsTable` 组件的 `methods` 属性，演示了如何格式化显示值：

```
getFormattedTransactions() {
  const getDisplayAmount = (type, amount) => {
    if (amount === 0) return accounting.formatMoney(amount);
    return accounting.formatMoney(amount, {
      format: { pos: '%s %v', neg: '%s (%v)' }
    });
  };

  const getDisplayDate = transactionDate => {
    if (!transactionDate) return '';
    const parsedTime = d3.timeParse('%Y-%m-%d')(transactionDate);
    return d3.timeFormat('%m/%d/%Y')(parsedTime);
  };

  return $store.state.transactions.map(
    ({
      type,
      rawAmount,
      cookedAmount,
      transactionDate,
      ...transaction
    }) => ({
      ...transaction,
```

```

        type,
        rawAmount: getDisplayAmount(type, rawAmount),
        cookedAmount: getDisplayAmount(type, cookedAmount),
        transactionDate: getDisplayDate(transactionDate)
    })
);
}

```

上述 `getFormattedTransactions()` 函数应用格式化到每个 `transaction` 记录中的 `rawAmount`、`cookedAmount` 和 `transactionDate` 字段。这样做是为了确保显示的值包括美元符号（对于金额）并以用户友好的格式呈现。

ChartsTab

`/components/ChartsTab` 文件夹包含两个组件文件：`ChartsTab.js` 和 `PieChart.js`。`ChartsTab` 组件包含两个 `PieChart` 组件的实例，一个用于收入，一个用于支出。每个 `PieChart` 组件显示按类别的原始或烹饪百分比。用户可以通过图表上方的按钮在原始或烹饪视图之间切换。`PieChart.js` 中的 `drawChart()` 方法使用 D3 来渲染饼图和图例。它使用 D3 的内置动画在加载时对饼图的每个部分进行动画处理：

```

arc
.append('path')
.attr('fill', d => colorScale(d.data.category))
.transition()
.delay((d, i) => i * 100)
.duration(500)
.attrTween('d', d => {
  const i = d3.interpolate(d.startAngle + 0.1, d.endAngle);
  return t => {
    d.endAngle = i(t);
    return arcPath(d);
  };
});

```

<https://bl.ocks.org>. That's it for the components review; le

```
t's try running the application.
```

运行应用程序

您已经编写并编译了 C 代码，并添加了前端逻辑。现在是时候启动应用程序并与之交互了。在本节中，我们将验证应用程序的 `/src` 文件夹，运行应用程序，并测试功能，以确保一切都正常工作。

验证/`src` 文件夹

在启动应用程序之前，请参考以下结构，确保您的`/src` 文件夹结构正确，并包含以下内容：

```
├── /assets
│   ├── main.wasm
│   └── memory.wasm
├── /components
│   ├── /BalancesBar
│   │   ├── BalanceCard.js
│   │   └── BalancesBar.js
│   ├── /ChartsTab
│   │   ├── ChartsTab.js
│   │   └── PieChart.js
│   ├── /TransactionsTab
│   │   ├── ConfirmationModal.js
│   │   ├── TransactionModal.js
│   │   ├── TransactionsTab.js
│   │   └── TransactionsTable.js
│   └── App.js
└── /store
    ├── api.js
    ├── initializeWasm.js
    ├── store.js
    └── WasmTransactions.js
└── index.html
```

```
└── main.js  
└── styles.css
```

如果一切匹配，您就可以继续了。

启动它！

要启动应用程序，请在 `/cook-the-books` 文件夹中打开终端并运行以下命令：

```
npm start
```

`browser-sync` 是我们在本章第一节安装的开发依赖项，它充当本地服务器（类似于 `serve` 库）。它使应用程序可以从 `package.json` 文件中指定的端口（在本例中为 `4000`）在浏览器中访问。如果您在浏览器中导航到 `http://localhost:4000/index.html`，您应该会看到这个：

The screenshot shows a web application titled "Cook the Books". The top navigation bar includes icons for refresh, back, forward, and search, along with the title "Cook the Books" and a "Guest" link. Below the header, there are two tabs: "TRANSACTIONS" (which is selected) and "CHARTS". The main content area displays four large boxes showing financial balances: "INITIAL RAW BALANCE" (\$0.00), "CURRENT RAW BALANCE" (\$185,515.00), "INITIAL COOKED BALANCE" (\$0.00), and "CURRENT COOKED BALANCE" (\$247,531.00). Below these boxes is a table of transaction history:

DATE	TYPE	TO/FROM	CATEGORY	RAW AMOUNT	COOKED AMOUNT
08/11/2017	Deposit	Limin Zhu	Sales Revenue	\$ 5,904.00	\$ 7,250.00
08/14/2017	Deposit	Benjamin Bouvier	Sales Revenue	\$ 5,390.00	\$ 5,805.00
08/16/2017	Withdrawal	Uncle Buck's Computer Warehouse	Computer & Internet	\$ (350.00)	\$ (250.00)
09/01/2017	Withdrawal	The Food Restaurant	Meals &	\$ (127.00)	\$ (102.00)

A blue circular button with a white plus sign is located in the bottom right corner of the transaction table.

初始加载的应用程序

我们使用 `browser-sync` 而不是 `serve`，因为它会监视文件的更改，并在您进行更改时自动重新加载应用程序。要看到它的效果，请尝试将 `App.js` 中标题栏的内容从 `Cook the Books` 更改为 `Broil the Books`。浏览器将刷新，您将在标题栏中看到更新后的文本。

测试一下

为了确保一切都正常工作，请测试一下应用程序。以下各节描述了应用程序特定功能的操作和预期行为。跟着操作，看看是否得到了预期结果。如果遇到问题，您可以随时参考 `learn-webassembly` 存储库中 `/chapter-07-cook-the-books` 文件夹。

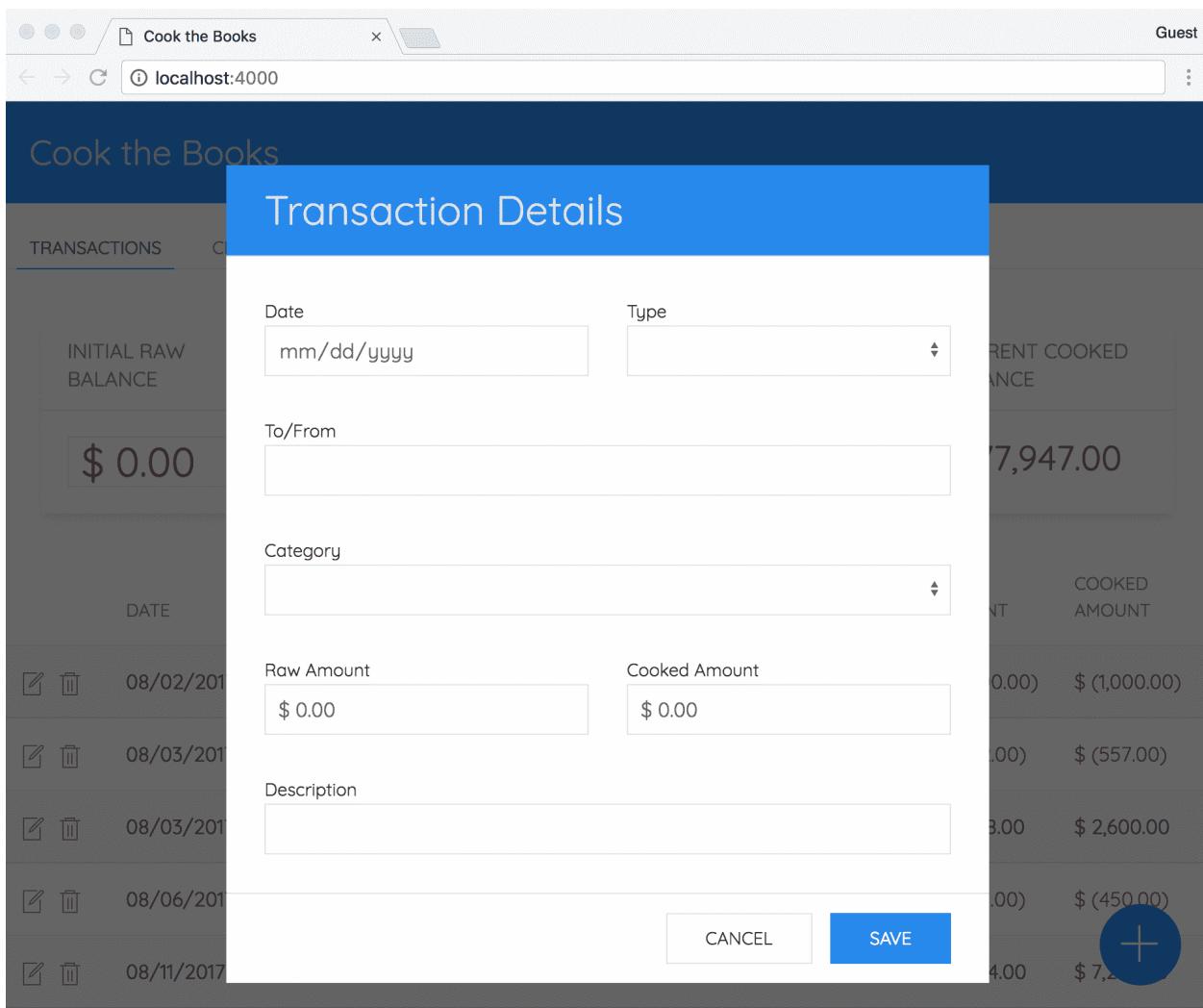
更改初始余额

尝试更改“INITIAL RAW BALANCE”和“INITIAL COOKED BALANCE”`BalanceCard` 组件上的输入值。当前的“CURRENT RAW BALANCE”和“CURRENT COOKED BALANCE”卡片数值应该更新以反映您的更改。

创建新交易

记下当前的原始和处理后的余额，然后按下窗口右下角的蓝色添加按钮。它应该加载 `TransactionModal` 组件。填写输入，记下**类型**，**原始金额**和**处理后的金额**，然后按保存按钮。

余额应该已经更新以反映新的金额。如果您选择了“提款”作为**类型**，则余额应该减少，否则，它们会增加（存款）如下截图所示：

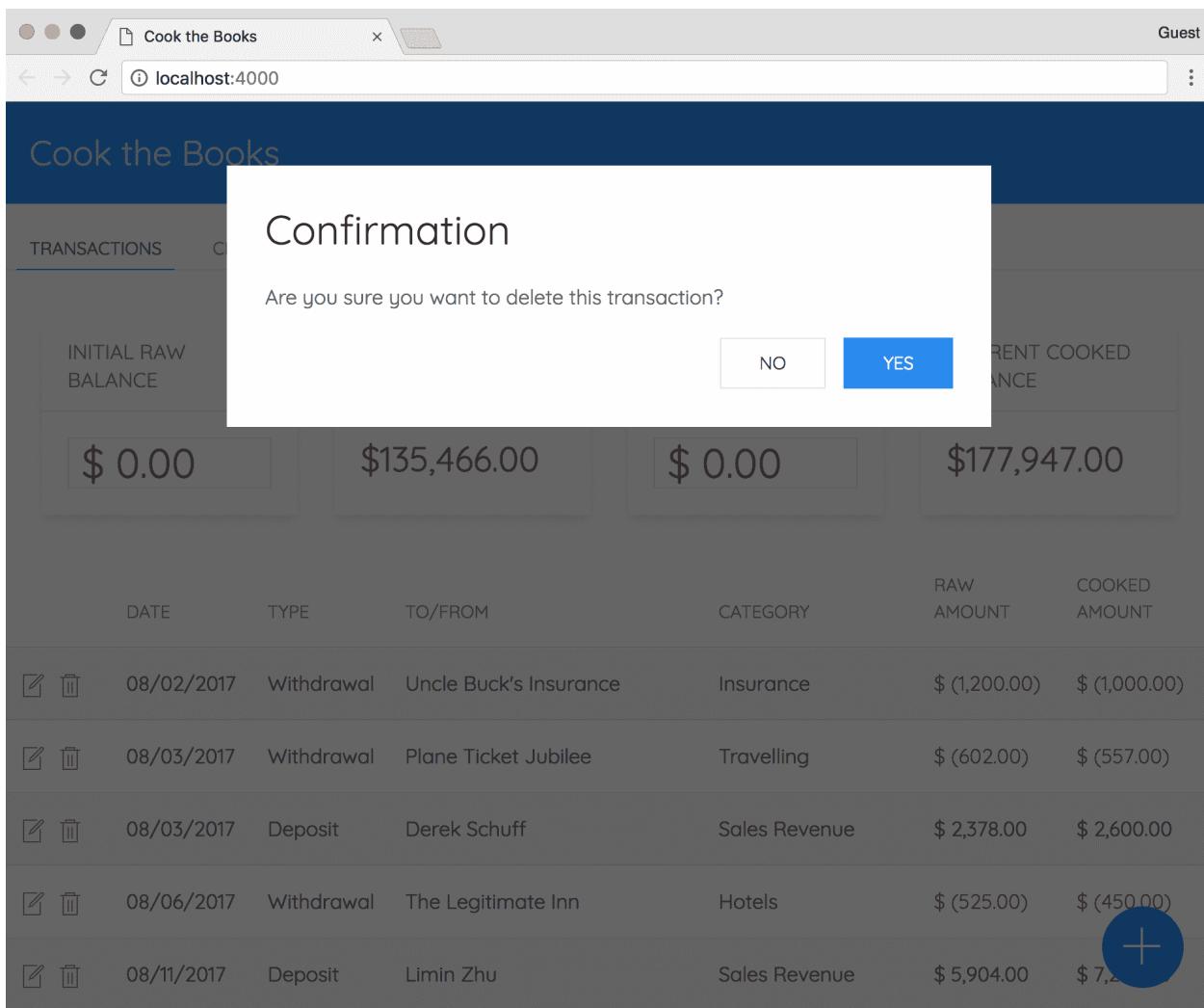


添加新交易时的 TransactionModal

删除现有交易

在 `TransactionsTable` 组件中选择一行，注意金额，然后按下该记录的垃圾桶按钮。

`ConfirmationModal` 组件应该出现。当您按下是按钮时，交易记录应该不再出现在表中，并且当前余额应该更新以反映与已删除交易相关的金额，如下截图所示：



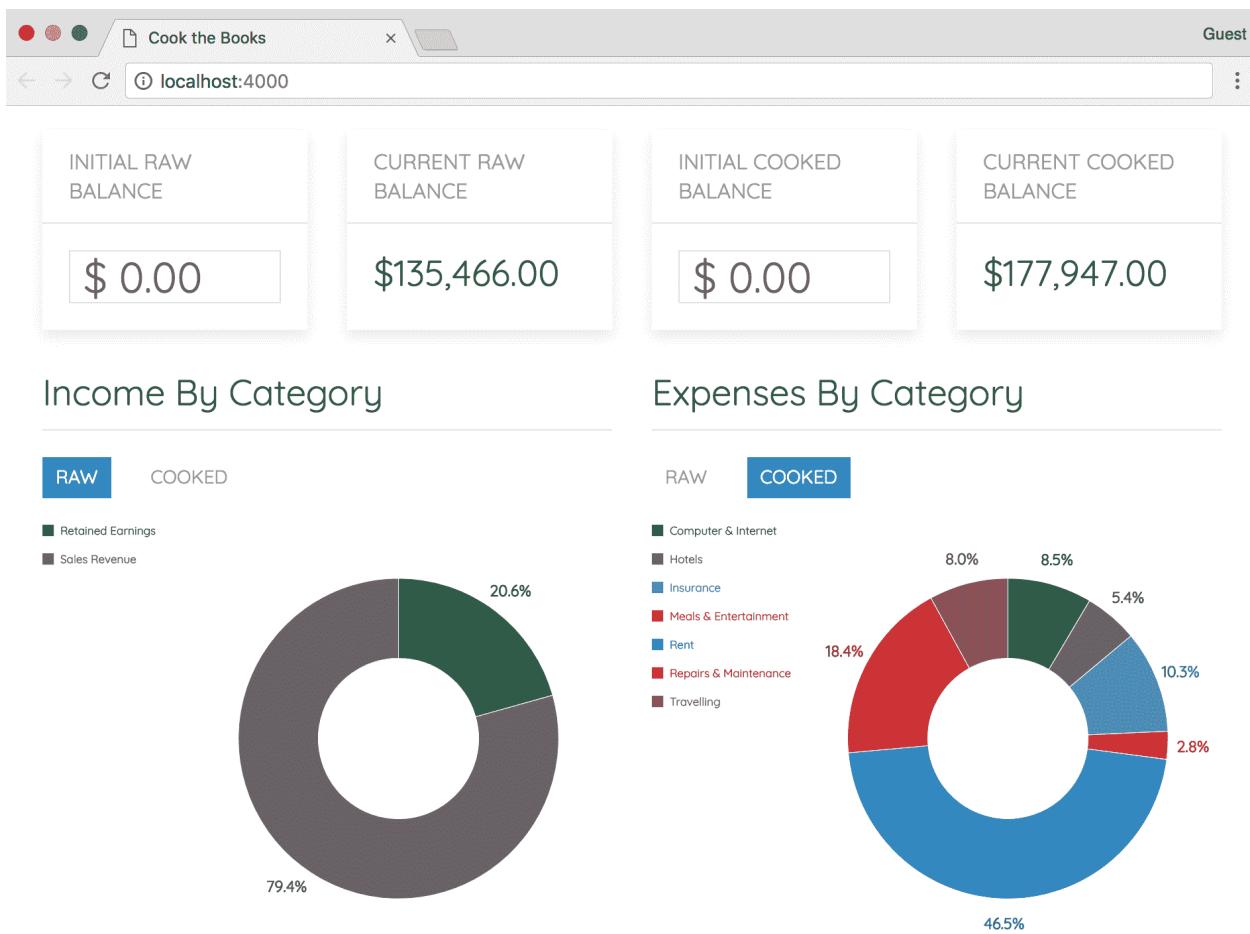
在按下删除按钮后显示确认模态

编辑现有交易

按照创建新交易的相同步骤，除了更改现有金额。检查当前余额以确保它们反映了更新后的交易金额。

测试图表选项卡

选择“图表”选项卡以加载 `chartsTab` 组件。按下每个 `PieChart` 组件中的按钮以在原始视图和处理后的视图之间切换。饼图应该重新渲染以显示更新后的值：



选择 CHARTS 选项卡的内容，选择不同的金额类型

总结

恭喜，您刚刚构建了一个使用 WebAssembly 的应用程序！告诉您的朋友！现在您了解了 WebAssembly 的能力和限制，是时候扩展我们的视野，并使用 Emscripten 提供的一些出色功能了。

摘要

在本章中，我们从头开始构建了一个会计应用程序，该应用程序使用 WebAssembly 而没有 Emscripten 提供的任何额外功能。通过遵守核心规范，我们展示了 WebAssembly 在其当前形式下的限制。然而，我们能够通过使用 Wasm 模块快速执行计算，这非常适合会计。我们使用 Vue 将应用程序拆分为组件，使用 UIKit 进行设计和布局，并使用 D3 从我们的交易数据创建饼图。在第八章中，使用 Emscripten 移植游戏，我们将充分利用 Emscripten 将现有的 C++ 代码库移植到 WebAssembly。

问题

1. 为什么我们在这个应用程序中使用 Vue (而不是 React 或 Angular) ?
2. 为什么我们在这个项目中使用 C 而不是 C++ ?
3. 为什么我们需要使用 jsonstore.io 设置一个模拟 API, 而不是在本地的 JSON 文件中存储数据 ?
4. 我们在 C 文件中使用的数据结构的名称是什么 ?
5. 我们从 `memory.wasm` 文件中需要哪些函数, 它们用于什么 ?
6. 为什么我们要在 Wasm 模块周围创建一个包装类 ?
7. 为什么我们将 `$store` 对象设为全局 ?
8. 在生产应用程序中, 您可以使用哪些库来管理全局状态 ?
9. 我们为什么使用 `browser-sync` 而不是 `serve` 来运行应用程序 ?

进一步阅读

- Vue: vuejs.org

第八章：使用 Emscripten 移植游戏

如 第七章 所示, 从头开始创建应用程序, WebAssembly 在当前形式下仍然相对有限。Emscripten 提供了强大的 API, 用于扩展 WebAssembly 的功能, 以添加功能到您的应用程序。在某些情况下, 编译为 WebAssembly 模块和 JavaScript 粘合代码 (而不是可执行文件) 可能只需要对现有的 C 或 C++ 源代码进行轻微更改。

在本章中, 我们将接受一个用 C++ 编写的代码库, 将其编译为传统可执行文件, 然后更新代码, 以便将其编译为 Wasm/JavaScript。我们还将添加一些额外功能, 以更紧密地集成到浏览器中。

通过本章结束时, 您将知道如何执行以下操作 :

- 更新 C++ 代码库以编译为 Wasm 模块/JavaScript 粘合代码 (而不是本机可执行文件) 是很重要的
- 使用 Emscripten 的 API 将浏览器集成到 C++ 应用程序中
- 使用正确的 `emcc` 标志构建一个多文件的 C++ 项目

- 使用 `emrun` 在浏览器中运行和测试 C++ 应用程序

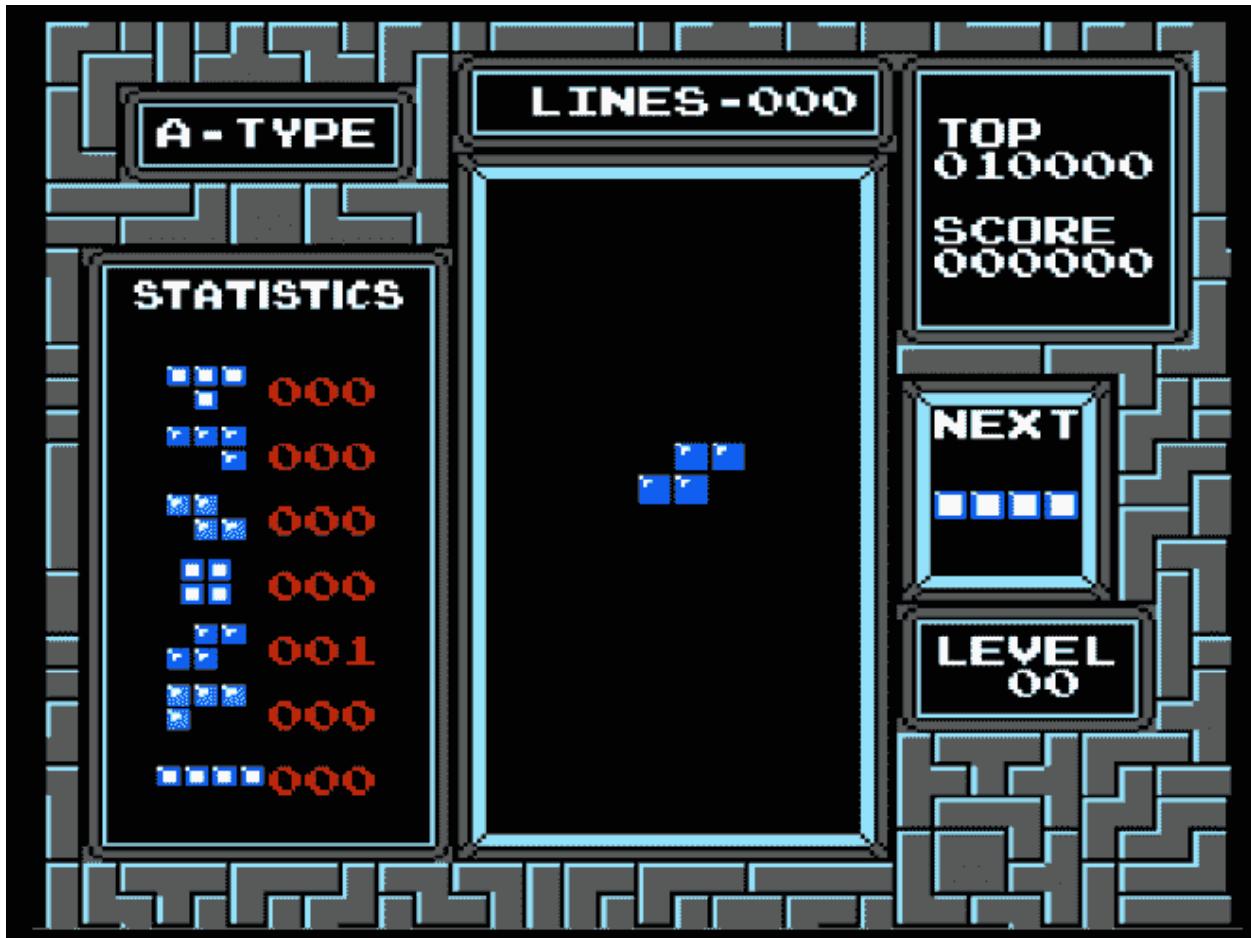
游戏概述

在本章中，我们将接受一个用 C++ 编写的俄罗斯方块克隆，并更新代码以集成 Emscripten 并编译为 Wasm/JS。原始形式的代码库利用 SDL2 编译为可执行文件，并可以从命令行加载。在本节中，我们将简要回顾一下俄罗斯方块是什么，如何获取代码（而无需从头开始编写），以及如何运行它。

什么是俄罗斯方块？

俄罗斯方块的主要目标是在游戏区域内旋转和移动各种形状的方块 (*Tetriminos*)，以创建没有间隙的一行方块。当创建了一整行时，它将从游戏区域中删除，并且您的得分将增加一分。在我们的游戏版本中，不会有获胜条件（尽管很容易添加）。

重要的是要了解游戏的规则和机制，因为代码使用算法来实现诸如碰撞检测和记分等概念。了解函数的目标有助于理解其中的代码。如果需要提高俄罗斯方块技能，我建议您在线尝试一下。您可以在 emulatoronline.com/nes-games/classic-tetris/ 上玩，无需安装 Adobe Flash。它看起来就像原始的任天堂版本：



在 EmulatorOnline.com 上玩经典的俄罗斯方块

我们将要处理的版本不包含方块计数器、级别或分数（我们只关注行数），但其操作方式将相同。

源的源

事实证明，搜索 Tetris C++ 会提供大量的教程和示例存储库供选择。为了保持到目前为止使用的格式和命名约定，我将这些资源结合起来创建了自己的游戏版本。本章结束时的进一步阅读部分中有这些资源的链接，如果您有兴趣了解更多。无论来源如何，移植代码库的概念和过程都是适用的。在这一点上，让我们简要讨论一下移植的一般情况。

关于移植的说明

将现有代码库移植到 Emscripten 并不总是一项简单的任务。在评估 C、C++ 或 Rust 应用程序是否适合转换时，需要考虑几个变量。例如，使用多个第三方库的游戏，甚至使用几个复杂的第三方库可能需要大量的工作。Emscripten 提供了以下常用库：

- `asio` : 一个网络和低级 I/O 编程库
- `Bullet` : 一个实时碰撞检测和多物理模拟库
- `Cocos2d` : 一套开源的跨平台游戏开发工具
- `FreeType` : 用于呈现字体的库
- `HarfBuzz` : 一个 OpenType 文本整形引擎
- `libpng` : 官方 PNG 参考库
- `Ogg` : 一个多媒体容器格式
- `SDL2` : 设计用于提供对音频、键盘、鼠标、操纵杆和图形硬件的低级访问的库
- `SDL2_image` : 一个图像文件加载库
- `SDL2_mixer` : 一个示例多通道音频混音库
- `SDL2_net` : 一个小型的跨平台网络库
- `SDL2_ttf` : 一个示例库，允许您在 SDL 应用程序中使用 TrueType 字体
- `Vorbis` : 通用音频和音乐编码格式
- `zlib` : 无损数据压缩库

如果库尚未移植，您将需要自行移植。这将有利于社区，但需要大量的时间和资源投入。我们的俄罗斯方块示例只使用了 SDL2，这使得移植过程相对简单。

获取代码

本章的代码位于 `learn-webassembly` 存储库的 `/chapter-08-tetris` 文件夹中。`/chapter-08-tetris` 中有两个目录：`/output-native` 文件夹，其中包含原始（未移植）代码，以及 `/output-wasm` 文件夹，其中包含移植后的代码。

如果您想要使用 VS Code 的任务功能进行本地构建步骤，您需要在 VS Code 中打开 `/chapter-08-tetris/output-native` 文件夹，而不是顶层的 `/learn-webassembly` 文件夹。

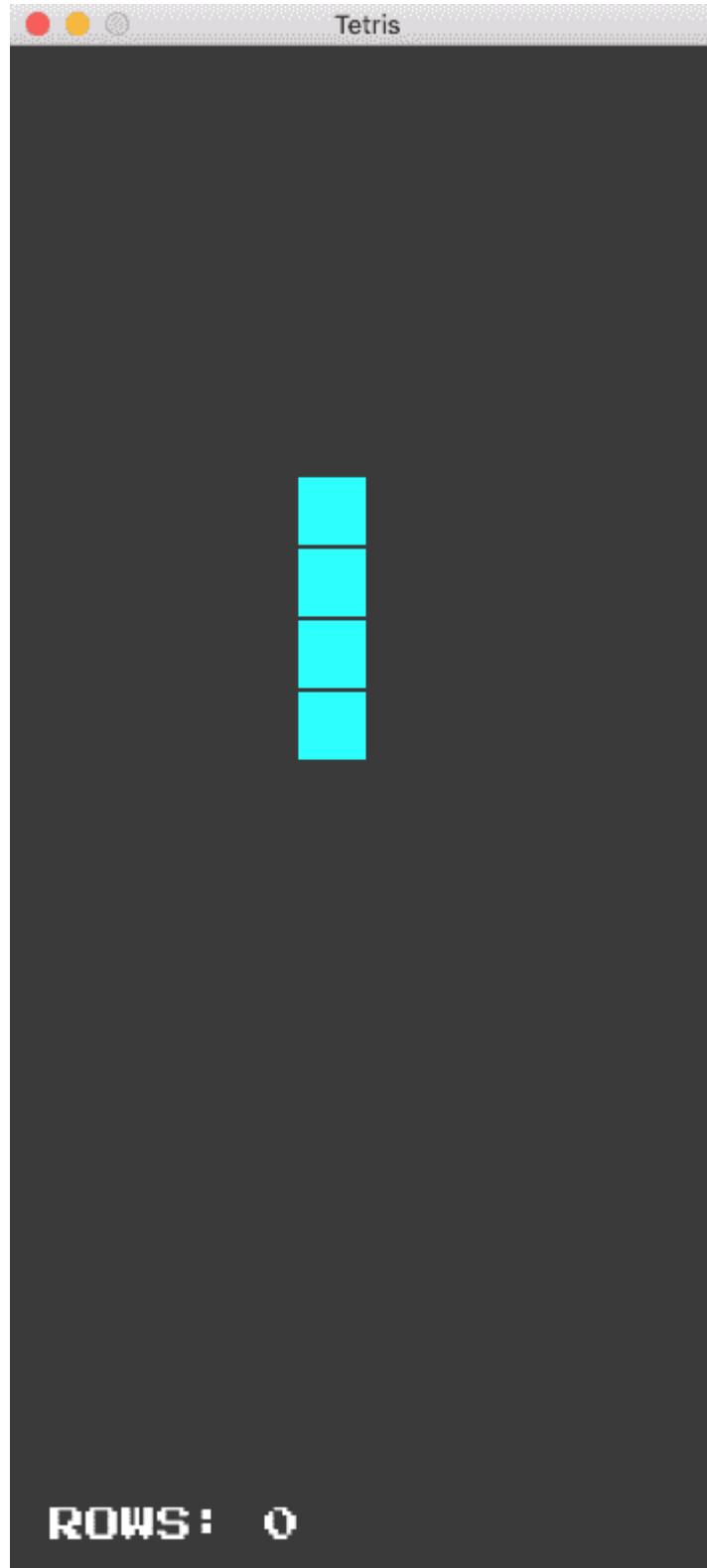
构建本地项目

`/output-native` 文件夹中的 `/cmake` 文件夹和 `CMakeLists.txt` 文件是构建项目所必需的。`README.md` 文件包含了在每个平台上启动代码的说明。构建项目并不是必须要通过移植过程。在您的平台上安装所需的依赖项并成功构建项目的过程可能会耗费大量时间和精力。

如果您仍然希望继续，您可以按照 [README.md](#) 文件中的说明，在选择任务 | 运行任务... 后从列表中选择构建可执行文件来通过 VS Code 的任务功能构建可执行文件。

游戏的运行情况

如果您成功构建了项目，您应该能够通过从 VS Code 菜单中选择任务 | 运行任务... 并从列表中选择启动可执行任务来运行它。如果一切顺利，您应该会看到类似以下的内容：



编译后的游戏可以本地运行

我们的游戏版本没有失败条件；它只是每清除一行就将行数增加一。如果俄罗斯方块中的一个方块触及到了板的顶部，游戏就结束了，板重新开始。这是游戏的一个基本实现，但是额外的功能会增加复杂性和所需的代码量。让我们更详细地审查代码库。

深入了解代码库

现在您已经可以使用代码了，您需要熟悉代码库。如果您不了解要移植的代码，那么您将更难成功地进行移植。在本章中，我们将逐个讨论每个 C++ 类和头文件，并描述它们在应用程序中的作用。

将代码分解为对象

C++ 是围绕面向对象的范式设计的，这正是俄罗斯方块代码库用来简化应用程序管理的方式。代码库由 C++ 类文件组成

(`.cpp`) 和头文件 (`.h`) 代表游戏上下文中的对象。我使用了什么是俄罗斯方块？部分的游戏概述来推断我需要哪些对象。

游戏方块 (Tetriminos) 和游戏区（称为井或矩阵）是类的良好候选对象。也许不那么直观，但同样有效的是游戏本身。类不一定需要像实际对象那样具体——它们非常适合存储共享代码。我很喜欢少打字，所以我选择使用 `Piece` 来表示一个 Tetrimino，`Board` 来表示游戏区（尽管井这个词更短，但并不太合适）。我创建了一个头文件来存储全局变量 (`constants.h`)，一个 `Game` 类来管理游戏过程，以及一个 `main.cpp` 文件，它作为游戏的入口点。以下是 `/src` 文件夹的内容：

```
|── board.cpp  
|── board.h  
|── constants.h  
|── game.cpp  
|── game.h  
|── main.cpp  
|── piece.cpp  
└── piece.h
```

每个文件（除了 `main.cpp` 和 `constants.h`）都有一个类 (`.cpp`) 和头文件 (`.h`)。头文件允许您在多个文件中重用代码并防止代码重复。进一步阅读部分包含了一些资源，供您了解更多关于头文件的知识。`constants.h` 文件几乎在应用程序的所有其他文件中都被使用，所以让我们首先来回顾一下它。

常量文件

我选择使用一个包含我们将要使用的常量的头文件，而不是在代码库中到处使用令人困惑的魔术数字。这个文件的内容如下：

```
#ifndef TETRIS_CONSTANTS_H
#define TETRIS_CONSTANTS_H

namespace Constants {
    const int BoardColumns = 10;
    const int BoardHeight = 720;
    const int BoardRows = 20;
    const int BoardWidth = 360;
    const int Offset = BoardWidth / BoardColumns;
    const int PieceSize = 4;
    const int ScreenHeight = BoardHeight + 50;
}

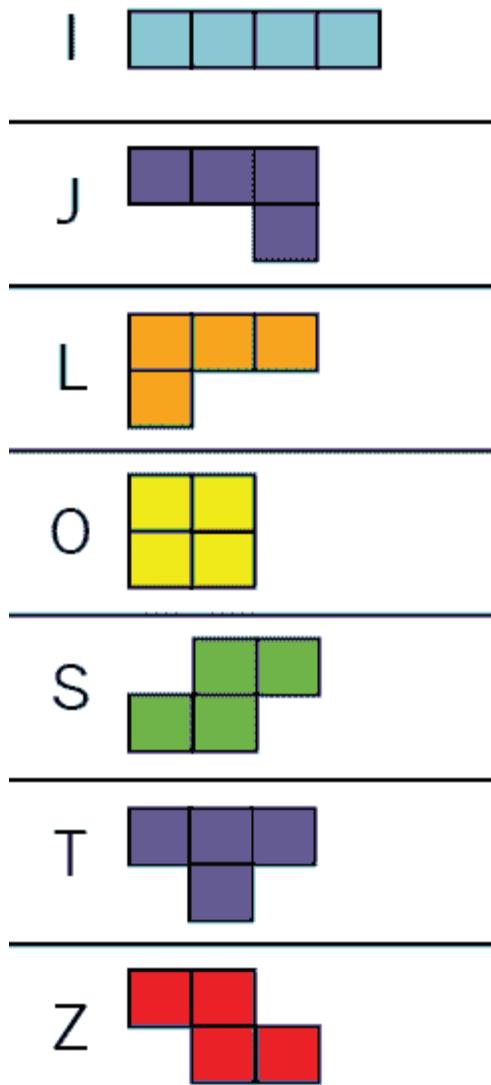
#endif // TETRIS_CONSTANTS_H
```

文件第一行的 `#ifndef` 语句是一个 `#include` 保护，它可以防止在编译过程中多次包含头文件。这些保护在应用程序的所有头文件中都被使用。每个常量的目的将在我们逐个讨论每个类时变得清晰。我首先包含它是为了提供各种元素大小及其相互关系的上下文。

让我们继续看一下代表游戏各个方面的各种类。`Piece` 类代表最低级别的对象，所以我们从这里开始，逐步向上到 `Board` 和 `Game` 类。

方块类

方块，或 *Tetrimino*，是可以在棋盘上移动和旋转的元素。有七种不同的 Tetriminos — 每种都用一个字母表示，并有对应的颜色：



Tetrimino 颜色，取自维基百科

我们需要一种方式来定义每个方块的形状、颜色和当前方向。每个方块有四种不同的方向（每次旋转 90 度），这导致了所有方块的 28 种总变化。颜色不会改变，所以只需要分配一次。有了这个想法，让我们首先看一下头文件 (`piece.h`)：

```
#ifndef TETRIS_PIECE_H
#define TETRIS_PIECE_H

#include <SDL2/SDL.h>
#include "constants.h"

class Piece {
public:
    enum Kind { I = 0, J, L, O, S, T, Z };
```

```

explicit Piece(Kind kind);

void draw(SDL_Renderer *renderer);
void move(int columnDelta, int rowDelta);
void rotate();
bool isBlock(int column, int row) const;
int getColumn() const;
int getRow() const;

private:
    Kind kind_;
    int column_;
    int row_;
    int angle_;
};

#endif // TETRIS_PIECE_H

```

游戏使用 SDL2 来渲染各种图形元素并处理键盘输入，这就是为什么我们将 `SDL_Renderer` 传递给 `draw()` 函数。您将看到 SDL2 是如何在 `Game` 类中使用的，但现在只需知道它被包含在内即可。头文件定义了 `Piece` 类的接口；让我们来看一下 `piece.cpp` 中的实现。我们将逐段代码进行讨论并描述功能。

构造函数和 `draw()` 函数

代码的第一部分定义了 `Piece` 类的构造函数和 `draw()` 函数：

```

#include "piece.h"using namespace Constants;

Piece::Piece(Piece::Kind kind) :
    kind_(kind),
    column_(BoardColumns / 2 - PieceSize / 2),
    row_(0),
    angle_(0) {
}

```

```
void Piece::draw(SDL_Renderer *renderer) {
    switch (kind_) {
        case I:
            SDL_SetRenderDrawColor(renderer,
                /* Cyan: */ 45, 254, 254, 255);
            break;
        case J:
            SDL_SetRenderDrawColor(renderer,
                /* Blue: */ 11, 36, 251, 255);
            break;
        case L:
            SDL_SetRenderDrawColor(renderer,
                /* Orange: */ 253, 164, 41, 255);
            break;
        case O:
            SDL_SetRenderDrawColor(renderer,
                /* Yellow: */ 255, 253, 56, 255);
            break;
        case S:
            SDL_SetRenderDrawColor(renderer,
                /* Green: */ 41, 253, 47, 255);
            break;
        case T:
            SDL_SetRenderDrawColor(renderer,
                /* Purple: */ 126, 15, 126, 255);
            break;
        case Z:
            SDL_SetRenderDrawColor(renderer,
                /* Red: */ 252, 13, 28, 255);
            break;
    }

    for (int column = 0; column < PieceSize; ++column) {
        for (int row = 0; row < PieceSize; ++row) {
            if (isBlock(column, row)) {
```

```

        SDL_Rect rect{
            (column + column_) * Offset + 1,
            (row + row_) * Offset + 1,
            Offset - 2,
            Offset - 2
        };
        SDL_RenderFillRect(renderer, &rect);
    }
}
}

```

构造函数用默认值初始化类。`BoardColumns` 和 `PieceSize` 的值是来自 `constants.h` 文件的常量。`BoardColumns` 表示棋盘上可以放置的列数，在这种情况下是 `10`。`PieceSize` 常量表示方块在列中占据的区域或块，为 `4`。分配给私有 `columns_` 变量的初始值表示棋盘的中心。

`draw()` 函数循环遍历棋盘上所有可能的行和列，并填充任何由棋子占据的单元格与其对应的颜色。判断单元格是否被棋子占据是在 `isBlock()` 函数中执行的，接下来我们将讨论这个函数。

move()、rotate()和isBlock()函数

第二部分包含移动或旋转方块并确定其当前位置的逻辑：

```

void Piece::move(int columnDelta, int rowDelta) {
    column_ += columnDelta;
    row_ += rowDelta;
}

void Piece::rotate() {
    angle_ += 3;
    angle_ %= 4;
}

bool Piece::isBlock(int column, int row) const {
    static const char *Shapes[][][4] = {
        // I

```

```
{  
    " * "  
    " * "  
    " * "  
    " * ",  
    "  
    " * * * "  
    "  
    " * ",  
    " * "  
    " * "  
    " * ",  
    "  
    " * * * "  
    "  
    " * ",  
},  
// J  
{  
    " * "  
    " * "  
    " * * "  
    "  
    "  
    " * "  
    " * * "  
    "  
    " * ",  
    " * * "  
    " * "  
    " * "  
    "  
    "  
    " * "  
    " * "  
    "  
    "  
    " * * "  
    " * ",
```

```

        },
        ...
    };
    return Shapes[kind_][angle_][column + row * PieceSize] ==
'*';
}

int Piece::getColumn() const {
    return column_;
}
int Piece::getRow() const {
    return row_;
}

```

`move()` 函数更新了私有 `column_` 和 `row_` 变量的值，从而决定了方块在棋盘上的位置。

`rotate()` 函数将私有 `angle_` 变量的值设置为 0、1、2 或 3（这就是为什么使用 `%= 4`）。

确定显示哪种类型的方块，它的位置和旋转是在 `isBlock()` 函数中执行的。我省略了 `Shapes` 多维数组的除了前两个元素之外的所有内容，以避免文件混乱，但是剩下的五种方块类型在实际代码中是存在的。我承认这不是最优雅的实现，但它完全适合我们的目的。

私有的 `kind_` 和 `angle_` 值被指定为 `Shapes` 数组中的维度，以选择四个相应的 `char*` 元素。这四个元素代表方块的四种可能的方向。如果字符串中的 `column + row * PieceSize` 索引是一个星号，那么方块就存在于指定的行和列。如果你决定通过网络上的一个俄罗斯方块教程（或者查看 GitHub 上的许多俄罗斯方块存储库之一）来学习，你会发现有几种不同的方法来计算一个单元格是否被方块占据。我选择了这种方法，因为它更容易可视化方块。

getColumn() 和 getRow() 函数

代码的最后一部分包含了获取方块的行和列的函数：

```

int Piece::getColumn() const {
    return column_;
}

int Piece::getRow() const {

```

```
    return row_;  
}
```

这些函数只是简单地返回私有 `column_` 或 `row_` 变量的值。现在你对 `Piece` 类有了更好的理解，让我们继续学习 `Board`。

Board 类

`Board` 包含 `Piece` 类的实例，并且需要检测方块之间的碰撞，行是否已满，以及游戏是否结束。让我们从头文件（`board.h`）的内容开始：

```
#ifndef TETRIS_BOARD_H  
#define TETRIS_BOARD_H  
  
#include <SDL2/SDL.h>#include <SDL2/SDL2_ttf.h>#include "constants.h"#include "piece.h"using namespace Constants;  
  
class Board {  
public:  
    Board();  
    void draw(SDL_Renderer *renderer, TTF_Font *font);  
    bool isCollision(const Piece &piece) const;  
    void unite(const Piece &piece);  
  
private:  
    bool isRowFull(int row);  
    bool areFullRowsPresent();  
    void updateOffsetRow(int fullRow);  
    void displayScore(SDL_Renderer *renderer, TTF_Font *font);  
  
    bool cells_[BoardColumns][BoardRows];  
    int currentScore_;  
};  
  
#endif // TETRIS_BOARD_H
```

`Board` 有一个 `draw()` 函数，类似于 `Piece` 类，还有一些其他函数用于管理行和跟踪棋盘上哪些单元格被占据。`SDL2_ttf` 库用于在窗口底部渲染带有当前分数（清除的行数）的“ROWS:”文本。现在，让我们来看看实现文件（`board.cpp`）的每个部分。

构造函数和 `draw()` 函数

代码的第一部分定义了 `Board` 类的构造函数和 `draw()` 函数：

```
#include <iostream>#include "board.h"using namespace Constant  
s;  
  
Board::Board() : cells_{{ false }}, currentScore_(0) {}  
  
void Board::draw(SDL_Renderer *renderer, TTF_Font *font) {  
    displayScore(renderer, font);  
    SDL_SetRenderDrawColor(  
        renderer,  
        /* Light Gray: */ 140, 140, 140, 255);  
    for (int column = 0; column < BoardColumns; ++column) {  
        for (int row = 0; row < BoardRows; ++row) {  
            if (cells_[column][row]) {  
                SDL_Rect rect{  
                    column * Offset + 1,  
                    row * Offset + 1,  
                    Offset - 2,  
                    Offset - 2  
                };  
                SDL_RenderFillRect(renderer, &rect);  
            }  
        }  
    }  
}
```

`Board` 构造函数将私有 `cells_` 和 `currentScore_` 变量的值初始化为默认值。`cells_` 变量是一个布尔值的二维数组，第一维表示列，第二维表示行。如果一个方块占据特定的列和行，数组中相应的值为 `true`。`draw()` 函数的行为类似于 `Piece` 中的 `draw()` 函数，它用颜色填充

包含方块的单元格。然而，这个函数只填充被已经到达底部的方块占据的单元格，颜色为浅灰色，不管是什么类型的方块。

isCollision()函数

代码的第二部分包含了检测碰撞的逻辑：

```
bool Board::isCollision(const Piece &piece) const {
    for (int column = 0; column < PieceSize; ++column) {
        for (int row = 0; row < PieceSize; ++row) {
            if (piece.isBlock(column, row)) {
                int columnTarget = piece.getColumn() + column;
                int rowTarget = piece.getRow() + row;
                if (
                    columnTarget < 0
                    || columnTarget >= BoardColumns
                    || rowTarget < 0
                    || rowTarget >= BoardRows
                ) {
                    return true;
                }
                if (cells_[columnTarget][rowTarget]) return true;
            }
        }
    }
    return false;
}
```

`isCollision()` 函数循环遍历棋盘上的每个单元格，直到找到由作为参数传递的 `&piece` 占据的单元格。如果方块即将与棋盘的任一侧碰撞，或者已经到达底部，函数返回 `true`，否则返回 `false`。

unite()函数

代码的第三部分包含了将方块与顶行合并的逻辑，当方块停止时。

```

void Board::unite(const Piece &piece) {
    for (int column = 0; column < PieceSize; ++column) {
        for (int row = 0; row < PieceSize; ++row) {
            if (piece.isBlock(column, row)) {
                int columnTarget = piece.getColumn() + column;
                int rowTarget = piece.getRow() + row;
                cells_[columnTarget][rowTarget] = true;
            }
        }
    }

    // Continuously loops through each of the rows until no full rows are
    // detected and ensures the full rows are collapsed and non-full rows
    // are shifted accordingly:
    while (areFullRowsPresent()) {
        for (int row = BoardRows - 1; row >= 0; --row) {
            if (isRowFull(row)) {
                updateOffsetRow(row);
                currentScore_ += 1;
                for (int column = 0; column < BoardColumns; ++column) {
                    cells_[column][0] = false;
                }
            }
        }
    }
}

bool Board::isRowFull(int row) {
    for (int column = 0; column < BoardColumns; ++column) {
        if (!cells_[column][row]) return false;
    }
}

```

```

        return true;
    }

bool Board::areFullRowsPresent() {
    for (int row = BoardRows - 1; row >= 0; --row) {
        if (isRowFull(row)) return true;
    }
    return false;
}

void Board::updateOffsetRow(int fullRow) {
    for (int column = 0; column < BoardColumns; ++column) {
        for (int rowOffset = fullRow - 1; rowOffset >= 0; --rowOffset) {
            cells_[column][rowOffset + 1] =
            cells_[column][rowOffset];
        }
    }
}

```

`unite()` 函数和相应的 `isRowFull()`、`areFullRowsPresent()` 和 `updateOffsetRow()` 函数执行多个操作。它通过将适当的数组位置设置为 `true`，使用指定的 `&piece` 参数更新了私有的 `cells_` 变量，该参数占据了行和列。它还通过将相应的 `cells_` 数组位置设置为 `false` 来清除棋盘上的任何完整行（所有列都填满），并增加了 `currentScore_`。清除行后，`cells_` 数组被更新，将清除的行上面的行向下移动 1。

displayScore()函数

代码的最后部分在游戏窗口底部显示分数：

```

void Board::displayScore(SDL_Renderer *renderer, TTF_Font *font) {
    std::stringstream message;
    message << "ROWS: " << currentScore_;
    SDL_Color white = { 255, 255, 255 };
    SDL_Surface *surface = TTF_RenderText_Blended(

```

```

        font,
        message.str().c_str(),
        white);
    SDL_Texture *texture = SDL_CreateTextureFromSurface(
        renderer,
        surface);
    SDL_Rect messageRect{ 20, BoardHeight + 15, surface->w, s
urface->h };
    SDL_FreeSurface(surface);
    SDL_RenderCopy(renderer, texture, nullptr, &messageRect);
    SDL_DestroyTexture(texture);
}

```

`displayScore()` 函数使用 `SDL2_ttf` 库在窗口底部（在棋盘下方）显示当前分数。`TTF_Font` `*font` 参数从 `Game` 类传递进来，以避免在更新分数时每次初始化字体。`stringstream` `message` 变量用于创建文本值，并将其设置为 `TTF_RenderText_Blended()` 函数内的 C `char*`。其余代码绘制文本在 `SDL_Rect` 上，以确保正确显示。

这就是 `Board` 类的全部内容；让我们继续看看 `Game` 类是如何组合在一起的。

游戏类

`Game` 类包含循环函数，使您可以通过按键在棋盘上移动方块。以下是头文件 (`game.h`) 的内容：

```

#ifndef TETRIS_GAME_H
#define TETRIS_GAME_H

#include <SDL2/SDL.h>#include <SDL2/SDL2_ttf.h>#include "cons
tants.h"#include "board.h"#include "piece.h"class Game {
public:
    Game();
    ~Game();
    bool loop();

private:
    Game(const Game &);
}

```

```

Game &operator=(const Game &);

void checkForCollision(const Piece &newPiece);
void handleKeyEvents(SDL_Event &event);

SDL_Window *window_;
SDL_Renderer *renderer_;
TTF_Font *font_;
Board board_;
Piece piece_;
uint32_t moveTime_;

};

#endif // TETRIS_GAME_H

```

`loop()` 函数包含游戏逻辑，并根据事件管理状态。在 `private:` 标头下的前两行防止创建多个游戏实例，这可能会导致内存泄漏。私有方法减少了 `loop()` 函数中的代码行数，简化了维护和调试。让我们继续看 `game.cpp` 中的实现。

构造函数和析构函数

代码的第一部分定义了在加载类实例（构造函数）和卸载类实例（析构函数）时执行的操作：

```

#include <cstdlib>#include <iostream>#include <stdexcept>#include "game.h"using namespace std;
using namespace Constants;

Game::Game() :
    // Create a new random piece:
    piece_{ static_cast<Piece::Kind>(rand() % 7) },
    moveTime_(SDL_GetTicks())
{
    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        throw runtime_error(
            "SDL_Init(SDL_INIT_VIDEO): " + string(SDL_GetError())
    }
}

```

```

        r()));
    }

    SDL_CreateWindowAndRenderer(
        BoardWidth,
        ScreenHeight,
        SDL_WINDOW_OPENGL,
        &window_,
        &renderer_);
    SDL_SetWindowPosition(
        window_,
        SDL_WINDOWPOS_CENTERED,
        SDL_WINDOWPOS_CENTERED);
    SDL_SetWindowTitle(window_, "Tetris");

    if (TTF_Init() != 0) {
        throw runtime_error("TTF_Init(): " + string(TTF_GetError()));
    }
    font_ = TTF_OpenFont("PressStart2P.ttf", 18);
    if (font_ == nullptr) {
        throw runtime_error("TTF_OpenFont: " + string(TTF_GetError()));
    }
}

Game::~Game() {
    TTF_CloseFont(font_);
    TTF_Quit();
    SDL_DestroyRenderer(renderer_);
    SDL_DestroyWindow(window_);
    SDL_Quit();
}

```

构造函数代表应用程序的入口点，因此所有必需的资源都在其中分配和初始化。

`TTF_OpenFont()` 函数引用了从 Google Fonts 下载的 TrueType 字体文件，名为 Press Start 2P。您可以在 fonts.google.com/specimen/Press+Start+2P 上查看该字体。它存在于存储

库的 `/resources` 文件夹中，并在构建项目时复制到可执行文件所在的相同文件夹中。如果在初始化 SDL2 资源时发生错误，将抛出 `runtime_error` 并提供错误的详细信息。析构函数 `(~Game())` 在应用程序退出之前释放我们为 SDL2 和 `SDL2_ttf` 分配的资源，以避免内存泄漏。

loop()函数

代码的最后部分代表了 `Game::loop`：

```
bool Game::loop() {
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        switch (event.type) {
            case SDL_KEYDOWN:
                handleKeyEvents(event);
                break;
            case SDL_QUIT:
                return false;
            default:
                return true;
        }
    }

    SDL_SetRenderDrawColor(renderer_, /* Dark Gray: */ 58, 5
8, 58, 255);
    SDL_RenderClear(renderer_);
    board_.draw(renderer_, font_);
    piece_.draw(renderer_);

    if (SDL_GetTicks() > moveTime_) {
        moveTime_ += 1000;
        Piece newPiece = piece_;
        newPiece.move(0, 1);
        checkForCollision(newPiece);
    }
    SDL_RenderPresent(renderer_);
```

```
        return true;
    }

void Game::checkForCollision(const Piece &newPiece) {
    if (board_.isCollision(newPiece)) {
        board_.unite(piece_);
        piece_ = Piece{ static_cast<Piece::Kind>(rand() % 7)
    };
        if (board_.isCollision(piece_)) board_ = Board();
    } else {
        piece_ = newPiece;
    }
}

void Game::handleKeyEvents(SDL_Event &event) {
    Piece newPiece = piece_;
    switch (event.key.keysym.sym) {
        case SDLK_DOWN:
            newPiece.move(0, 1);
            break;
        case SDLK_RIGHT:
            newPiece.move(1, 0);
            break;
        case SDLK_LEFT:
            newPiece.move(-1, 0);
            break;
        case SDLK_UP:
            newPiece.rotate();
            break;
        default:
            break;
    }
    if (!board_.isCollision(newPiece)) piece_ = newPiece;
}
```

`loop()` 函数返回一个布尔值，只要 `SDL_QUIT` 事件尚未触发。每隔 1 秒，执行 `Piece` 和 `Board` 实例的 `draw()` 函数，并相应地更新棋盘上的方块位置。左、右和下箭头键控制方块的移动，而上箭头键将方块旋转 90 度。对按键的适当响应在 `handleKeyEvents()` 函数中处理。`checkForCollision()` 函数确定活动方块的新实例是否与棋盘的任一侧发生碰撞，或者停在其他方块的顶部。如果是，就创建一个新方块。清除行的逻辑（通过 `Board` 的 `unite()` 函数）也在这个函数中处理。我们快要完成了！让我们继续看 `main.cpp` 文件。

主文件

`main.cpp` 没有关联的头文件，因为它的唯一目的是作为应用程序的入口点。实际上，该文件只有七行：

```
#include "game.h"
int main() {
    Game game;
    while (game.loop());
    return 0;
}
```

`while` 语句在 `loop()` 函数返回 `false` 时退出，这发生在 `SDL_QUIT` 事件触发时。这个文件所做的就是创建一个新的 `Game` 实例并启动循环。这就是代码库的全部内容；让我们开始移植！

移植到 Emscripten

你对代码库有很好的理解，现在是时候开始用 Emscripten 移植了。幸运的是，我们能够利用一些浏览器的特性来简化代码，并完全移除第三方库。在这一部分，我们将更新代码以编译为 Wasm 模块和 JavaScript *glue* 文件，并更新一些功能以利用浏览器。

为移植做准备

`/output-wasm` 文件夹包含最终结果，但我建议你创建一个 `/output-native` 文件夹的副本，这样你就可以跟随移植过程。为本地编译和 Emscripten 编译设置了 VS Code 任务。如果你遇到困难，你可以随时参考 `/output-wasm` 的内容。确保你在 VS Code 中打开你复制的文件夹（文件 | 打开并选择你复制的文件夹），否则你将无法使用任务功能。

有什么改变？

这个游戏是移植的理想候选，因为它使用了 SDL2，这是一个广泛使用的库，已经有了 Emscripten 移植。在编译步骤中包含 SDL2 只需要传递一个额外的参数给 `emcc` 命令。

`SDL2_ttf` 库的 Emscripten 移植也存在，但保留它在代码库中并没有太多意义。它的唯一目的是以文本形式呈现得分（清除的行数）。我们需要将 TTF 文件与应用程序一起包含，并复杂化构建过程。Emscripten 提供了在我们的 C++ 中使用 JavaScript 代码的方法，所以我们将采取一个更简单的方法：在 DOM 中显示得分。

除了改变现有的代码，我们还需要创建一个 HTML 和 CSS 文件来在浏览器中显示和样式化游戏。我们编写的 JavaScript 代码将是最小的——我们只需要加载 Emscripten 模块，所有功能都在 C++ 代码库中处理。我们还需要添加一些 `<div>` 元素，并相应地布局以显示得分。让我们开始移植！

添加 web 资源

在你的项目文件夹中创建一个名为 `/public` 的文件夹。在 `/public` 文件夹中添加一个名为 `index.html` 的新文件，并填充以下内容：

```
<!doctype html>
<html lang="en-us">
<head>
  <title>Tetris</title>
  <link rel="stylesheet" type="text/css" href="styles.css" />
</head>
<body>
  <div class="wrapper">
    <h1>Tetris</h1>
    <div>
      <canvas id="canvas"></canvas>
      <div class="scoreWrapper">
        <span>ROWS:</span><span id="score"></span>
      </div>
    </div>
  </div>
  <script type="application/javascript" src="img/index.js"></script>
  <script type="application/javascript">
    Module({ canvas: () => document.getElementById('canva
```

```
s'))() })  
    </script>  
</body>  
</html>
```

在第一个 `<script>` 标签中加载的 `index.js` 文件尚不存在；它将在编译步骤中生成。让我们为元素添加一些样式。在 `/public` 文件夹中创建一个 `styles.css` 文件，并填充以下内容：

```
@import url("https://fonts.googleapis.com/css?family=Press+Start+2P");  
  
* {  
    font-family: "Press Start 2P", sans-serif;  
}  
  
body {  
    margin: 24px;  
}  
  
h1 {  
    font-size: 36px;  
}  
  
span {  
    color: white;  
    font-size: 24px;  
}  
  
.wrapper {  
    display: flex;  
    align-items: center;  
    flex-direction: column;  
}  
  
.titleWrapper {  
    display: flex;
```

```
    align-items: center;
    justify-content: center;
}

.header {
    font-size: 24px;
    margin-left: 16px;
}

.scorewrapper {
    background-color: #3A3A3A;
    border-top: 1px solid white;
    padding: 16px 0;
    width: 360px;
}

span:first-child {
    margin-left: 16px;
    margin-right: 8px;
}
```

由于我们使用的 Press Start 2P 字体托管在 Google Fonts 上，我们可以导入它以在网站上使用。这个文件中的 CSS 规则处理简单的布局和样式。这就是我们需要创建的与 web 相关的文件。现在，是时候更新 C++ 代码了。

移植现有代码

我们只需要编辑一些文件才能正确使用 Emscripten。为了简单和紧凑起见，只包含受影响的代码部分（而不是整个文件）。让我们按照上一节的顺序逐个文件进行，并从 `constants.h` 开始。

更新常量文件

我们将在 DOM 上显示清除的行数，而不是在游戏窗口本身上显示，所以你可以从文件中删除 `ScreenHeight` 常量。我们不再需要额外的空间来容纳得分文本：

```
namespace Constants {
    const int BoardColumns = 10;
    const int BoardHeight = 720;
    const int BoardRows = 20;
    const int BoardWidth = 360;
    const int Offset = BoardWidth / BoardColumns;
    const int PieceSize = 4;
    // const int ScreenHeight = BoardHeight + 50; ----- Delete this line
}
```

不需要对 `Piece` 类文件（`piece.cpp` / `piece.h`）进行任何更改。但是，我们需要更新 `Board` 类。让我们从头文件（`board.h`）开始。从底部开始，逐步更新 `displayScore()` 函数。在 `index.html` 文件的 `<body>` 部分，有一个 `id="score"` 的 `` 元素。我们将使用 `emscripten_run_script` 命令来更新此元素以显示当前分数。因此，`displayScore()` 函数变得更短了。变化前后如下所示。

这是 `Board` 类的 `displayScore()` 函数的原始版本：

```
void Board::displayScore(SDL_Renderer *renderer, TTF_Font *font) {
    std::stringstream message;
    message << "ROWS: " << currentScore_;
    SDL_Color white = { 255, 255, 255 };
    SDL_Surface *surface = TTF_RenderText_Blended(
        font,
        message.str().c_str(),
        white);
    SDL_Texture *texture = SDL_CreateTextureFromSurface(
        renderer,
        surface);
    SDL_Rect messageRect{ 20, BoardHeight + 15, surface->w, surface->h };
    SDL_FreeSurface(surface);
    SDL_RenderCopy(renderer, texture, nullptr, &messageRect);
```

```
    SDL_DestroyTexture(texture);
}
```

这是 `displayScore()` 函数的移植版本：

```
void Board::displayScore(int newScore) {
    std::stringstream action;
    action << "document.getElementById('score').innerHTML ="
<< newScore;
    emscripten_run_script(action.str().c_str());
}
```

`emscripten_run_script` 操作只是在 DOM 上找到 `` 元素，并将 `innerHTML` 设置为当前分数。我们无法在这里使用 `EM_ASM()` 函数，因为 Emscripten 不识别 `document` 对象。由于我们可以访问类中的私有 `currentScore` 变量，我们将把 `draw()` 函数中的 `displayScore()` 调用移动到 `unite()` 函数中。这限制了对 `displayScore()` 的调用次数，以确保只有在分数实际改变时才调用该函数。我们只需要添加一行代码来实现这一点。现在 `unite()` 函数的样子如下：

```
void Board::unite(const Piece &piece) {
    for (int column = 0; column < PieceSize; ++column) {
        for (int row = 0; row < PieceSize; ++row) {
            if (piece.isBlock(column, row)) {
                int columnTarget = piece.getColumn() + column;
                int rowTarget = piece.getRow() + row;
                cells_[columnTarget][rowTarget] = true;
            }
        }
    }

    // Continuously loops through each of the rows until no full rows are
    // detected and ensures the full rows are collapsed and non-full rows
    // are shifted accordingly:
```

```

        while (areFullRowsPresent()) {
            for (int row = BoardRows - 1; row >= 0; --row) {
                if (isRowFull(row)) {
                    updateOffsetRow(row);
                    currentScore_ += 1;
                    for (int column = 0; column < BoardColumns; +
+column) {
                        cells_[column][0] = false;
                    }
                }
            }
            displayScore(currentScore_); // ----- Add this line
        }
    }
}

```

由于我们不再使用 `SDL2_ttf` 库，我们可以更新 `draw()` 函数的签名并删除 `displayScore()` 函数调用。更新后的 `draw()` 函数如下：

```

void Board::draw(SDL_Renderer *renderer/*, TTF_Font *font */)
{
    // ^^^^^^^^^^^^^^ <-- Remove this argument
    // displayScore(renderer, font); <----- Delete this line
    SDL_SetRenderDrawColor(
        renderer,
        /* Light Gray: */ 140, 140, 140, 255);
    for (int column = 0; column < BoardColumns; ++column) {
        for (int row = 0; row < BoardRows; ++row) {
            if (cells_[column][row]) {
                SDL_Rect rect{
                    column * Offset + 1,
                    row * Offset + 1,
                    Offset - 2,
                    Offset - 2
                };
                SDL_RenderFillRect(renderer, &rect);
            }
        }
    }
}

```

```
}
```

`displayScore()` 函数调用已从函数的第一行中删除，并且 `TTF_Font *font` 参数也被删除了。让我们在构造函数中添加一个对 `displayScore()` 的调用，以确保当游戏结束并开始新游戏时，初始值设置为 `0`。

```
Board::Board() : cells_{ { false } }, currentScore_(0) {
    displayScore(0); // ----- Add this line
}
```

课堂文件就到这里。由于我们更改了 `displayScore()` 和 `draw()` 函数的签名，并移除了对 `SDL2_ttf` 的依赖，我们需要更新头文件。从 `board.h` 中删除以下行：

```
#ifndef TETRIS_BOARD_H
#define TETRIS_BOARD_H

#include <SDL2/SDL.h> // #include <SDL2/SDL2_ttf.h> <----- Delete this line
#include "constants.h" #include "piece.h" using namespace Constants;

class Board {
public:
    Board();
    void draw(SDL_Renderer *renderer /*, TTF_Font *font */);
                                            // ^^^^^^^^^^ <-- Remove this
    bool isCollision(const Piece &piece) const;
    void unite(const Piece &piece);

private:
    bool isRowFull(int row);
    bool areFullRowsPresent();
```

```

void updateOffsetRow(int fullRow);
void displayScore(SDL_Renderer *renderer, TTF_Font *font);
// ^^^^^^<-
- Remove this
bool cells_[BoardColumns][BoardRows];
int currentScore_;
};

#endif // TETRIS_BOARD_H

```

我们正在顺利进行！我们需要做的最后一个更改也是最大的一个。现有的代码库有一个 `Game` 类来管理应用程序逻辑，以及一个 `main.cpp` 文件来在 `main()` 函数中调用 `Game.loop()` 函数。循环机制是一个 `while` 循环，只要 `SDL_QUIT` 事件没有触发就会继续运行。我们需要改变我们的方法以适应 Emscripten。

Emscripten 提供了一个 `emscripten_set_main_loop` 函数，接受一个 `em_callback_func` 循环函数、`fps` 和一个 `simulate_infinite_loop` 标志。我们不能包含 `Game` 类并将 `Game.loop()` 作为 `em_callback_func` 参数，因为构建会失败。相反，我们将完全消除 `Game` 类，并将逻辑移到 `main.cpp` 文件中。将 `game.cpp` 的内容复制到 `main.cpp`（覆盖现有内容）并删除 `Game` 类文件（`game.cpp` / `game.h`）。由于我们不再声明 `Game` 类，因此从函数中删除 `Game::` 前缀。构造函数和析构函数不再有效（它们不再是类的一部分），因此我们需要将该逻辑移动到不同的位置。我们还需要重新排列文件以确保我们调用的函数出现在调用函数之前。最终结果如下：

```

#include <emscripten/emscripten.h>
#include <SDL2/SDL.h>
#include <stdexcept>
#include "constants.h"
#include "board.h"
#include "piece.h"
using namespace std;
using namespace Constants;

static SDL_Window *window = nullptr;
static SDL_Renderer *renderer = nullptr;
static Piece currentPiece{ static_cast<Piece::Kind>(rand() % 7) };
static Board board;
static int moveTime;

```

```
void checkForCollision(const Piece &newPiece) {
    if (board.isCollision(newPiece)) {
        board.unite(currentPiece);
        currentPiece = Piece{ static_cast<Piece::Kind>(rand() % 7) };
        if (board.isCollision(currentPiece)) board = Board();
    } else {
        currentPiece = newPiece;
    }
}

void handleKeyEvents(SDL_Event &event) {
    Piece newPiece = currentPiece;
    switch (event.key.keysym.sym) {
        case SDLK_DOWN:
            newPiece.move(0, 1);
            break;
        case SDLK_RIGHT:
            newPiece.move(1, 0);
            break;
        case SDLK_LEFT:
            newPiece.move(-1, 0);
            break;
        case SDLK_UP:
            newPiece.rotate();
            break;
        default:
            break;
    }
    if (!board.isCollision(newPiece)) currentPiece = newPiece;
}

void loop() {
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
```

```
        switch (event.type) {
            case SDL_KEYDOWN:
                handleKeyEvents(event);
                break;
            case SDL_QUIT:
                break;
            default:
                break;
        }
    }

    SDL_SetRenderDrawColor(renderer, /* Dark Gray: */ 58, 58,
58, 255);
    SDL_RenderClear(renderer);
    board.draw(renderer);
    currentPiece.draw(renderer);

    if (SDL_GetTicks() > moveTime) {
        moveTime += 1000;
        Piece newPiece = currentPiece;
        newPiece.move(0, 1);
        checkForCollision(newPiece);
    }
    SDL_RenderPresent(renderer);
}

int main() {
    moveTime = SDL_GetTicks();
    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        throw std::runtime_error("SDL_Init(SDL_INIT_VIDEO)");
    }
    SDL_CreateWindowAndRenderer(
        BoardWidth,
        BoardHeight,
        SDL_WINDOW_OPENGL,
        &window,
```

```

    &renderer);

emscripten_set_main_loop(loop, 0, 1);

SDL_DestroyRenderer(renderer);
renderer = nullptr;
SDL_DestroyWindow(window);
window = nullptr;
SDL_Quit();
return 0;
}

```

`handleKeyEvents()` 和 `checkForCollision()` 函数没有改变；我们只是将它们移到了文件的顶部。`loop()` 函数的返回类型从 `bool` 改为 `void`，这是 `emscripten_set_main_loop` 所需的。最后，构造函数和析构函数中的代码被移动到了 `main()` 函数中，并且移除了对 `SDL2_ttf` 的任何引用。我们不再使用调用 `Game` 的 `loop()` 函数的 `while` 语句，而是使用 `emscripten_set_main_loop(loop, 0, 1)`。我们修改了文件顶部的 `#include` 语句以适应 Emscripten、SDL2 和我们的 `Board` 和 `Piece` 类。这就是所有的更改——现在是时候配置构建并测试游戏了。

构建和运行游戏

随着代码的更新和所需的 Web 资产的准备，现在是构建和测试游戏的时候了。编译步骤与本书中之前的示例类似，但我们将使用不同的技术来运行游戏。在本节中，我们将配置构建任务以适应 C++ 文件，并使用 Emscripten 提供的功能来运行应用程序。

使用 VS Code 任务进行构建

我们将以两种方式配置构建：使用 VS Code 任务和 Makefile。如果您喜欢使用 VS Code 以外的编辑器，Makefile 是一个不错的选择。`./vscode/tasks.json` 文件已经包含了构建项目所需的任务。Emscripten 构建步骤是默认的（还有一组本地构建任务）。让我们逐个检查 `tasks` 数组中的每个任务，看看发生了什么。第一个任务在构建之前删除任何现有的编译输出文件：

```
{
  "label": "Remove Existing Web Files",
  "type": "shell",

```

```
"command": "rimraf",
"options": {
  "cwd": "${workspaceRoot}/public"
},
"args": [
  "index.js",
  "index.wasm"
]
}
```

第二个任务使用 `emcc` 命令进行构建：

```
{
  "label": "Build WebAssembly",
  "type": "shell",
  "command": "emcc",
  "args": [
    "--bind", "src/board.cpp", "src/piece.cpp", "src/main.cpp",
    "-std=c++14",
    "-O3",
    "-s", "WASM=1",
    "-s", "USE	SDL=2",
    "-s", "MODULARIZE=1",
    "-o", "public/index.js"
  ],
  "group": {
    "kind": "build",
    "isDefault": true
  },
  "problemMatcher": [],
  "dependsOn": ["Remove Existing Web Files"]
}
```

相关的参数都放在同一行上。`args` 数组中唯一的新的和陌生的添加是 `--bind` 参数和相应的 `.cpp` 文件。这告诉 Emscripten 所有在 `--bind` 之后的文件都是构建项目所需的。通过

从菜单中选择任务|运行构建任务...或使用键盘快捷键`Cmd/Ctrl + Shift + B`来测试构建。构建需要几秒钟，但终端会在编译过程完成时通知您。如果成功，您应该在`/public`文件夹中看到一个`index.js`和一个`index.wasm`文件。

使用 Makefile 进行构建

如果您不想使用 VS Code，您可以使用 Makefile 来实现与 VS Code 任务相同的目标。在项目文件夹中创建一个名为`Makefile`的文件，并填充以下内容（确保文件使用制表符而不是空格）：

```
# This allows you to just run the "make" command without specifying
# arguments:
.DEFAULT_GOAL := build

# Specifies which files to compile as part of the project:
CPP_FILES = $(wildcard src/*.cpp)

# Flags to use for Emscripten emcc compile command:
FLAGS = -std=c++14 -O3 -s WASM=1 -s USE	SDL=2 -s MODULARIZE=1 \
        --bind $(CPP_FILES)

# Name of output (the .wasm file is created automatically):
OUTPUT_FILE = public/index.js

# This is the target that compiles our executable
compile: $(CPP_FILES)
    emcc $(FLAGS) -o $(OUTPUT_FILE)

# Removes the existing index.js and index.wasm files:
clean:
    rimraf $(OUTPUT_FILE)
    rimraf public/index.wasm

# Removes the existing files and builds the project:
```

```
build: clean compile  
  @echo "Build Complete!"
```

所执行的操作与 VS Code 任务中执行的操作相同，只是使用更通用的工具格式。默认的构建步骤已在文件中设置，因此您可以在项目文件夹中运行以下命令来编译项目：

```
make
```

现在您已经有了一个编译好的 Wasm 文件和 JavaScript 粘合代码，让我们尝试运行游戏。

运行游戏

我们将使用 Emscripten 工具链的内置功能 `emrun`，而不是使用 `serve` 或 `browser-sync`。它提供了一个额外的好处，即捕获 `stdout` 和 `stderr`（如果您将 `--emrun` 链接标志传递给 `emcc` 命令），并在需要时将它们打印到终端。我们不会使用 `--emrun` 标志，但是在不必安装任何额外的依赖项的情况下拥有一个本地 Web 服务器是一个很好的附加功能。在项目文件夹中打开一个终端实例，并运行以下命令来启动游戏：

```
emrun --browser chrome --no_emrun_detect public/index.html
```

如果您正在开发中使用 `firefox`，可以为浏览器指定 `firefox`。`--no_emrun_detect` 标志会隐藏终端中的一条消息，指出 HTML 页面不支持 `emrun`。如果您导航到 <http://localhost:6931/index.html>，您应该会看到以下内容：



在浏览器中运行的俄罗斯方块

尝试旋转和移动方块，以确保一切都正常工作。当成功清除一行时，行数应该增加一。您还可能注意到，如果您离棋盘边缘太近，您将无法旋转一些方块。恭喜，您已成功将一个 C++ 游戏移植到 Emscripten！

总结

在本章中，我们将一个使用 SDL2 编写的 C++ Tetris 克隆移植到 Emscripten，以便可以在浏览器中使用 WebAssembly 运行。我们介绍了 Tetris 的规则以及它们如何映射到现有代码库中的逻辑。我们还逐个审查了现有代码库中的每个文件以及必须进行的更改，以成功编译为 Wasm 文件和 JavaScript 粘合代码。更新现有代码后，我们创建了所需的 HTML 和 CSS 文件，然后使用适当的 `emcc` 标志配置了构建步骤。构建完成后，使用 Emscripten 的 `emrun` 命令运行游戏。

在第九章中，与 `Node.js` 集成，我们将讨论如何将 WebAssembly 集成到 `Node.js` 中，以及这种集成提供的好处。

问题

1. Tetris 中的方块叫什么？
2. 选择不将现有的 C++ 代码库移植到 Emscripten 的一个原因是什 么？
3. 我们用什么工具来将游戏编译成本机代码（例如，可执行文件）？
4. `constants.h` 文件的目的是什 么？
5. 为什么我们能够消除 `SDL2_ttf` 库？
6. 我们使用了哪个 Emscripten 函数来开始运行游戏？
7. 我们在 `emcc` 命令中添加了哪个参数来构建游戏，它有什么作用？
8. `emrun` 相对于 `serve` 和 `Browsersync` 这样的工具有什么优势？

进一步阅读

- C++ 中的头文件：www.sitesbay.com/cpp/cpp-header-files
- GitHub 上的 `SDL2 Tetris`：github.com/andwn/sdl2-tetris
- GitHub 上的 `Tetris`：github.com/abesary/tetris

- Tetris - Linux on GitHub: github.com/abesary/tetris-linux

第九章：与 Node.js 集成

现代 Web 在开发和服务器端管理方面严重依赖 Node.js。随着越来越复杂的浏览器应用程序执行计算密集型操作，性能的提升将非常有益。在本章中，我们将描述通过各种示例集成 WebAssembly 与 Node.js 的各种方式。

本章的目标是理解以下内容：

- 将 WebAssembly 与 Node.js 集成的优势
- 如何与 Node.js 的 WebAssembly API 交互
- 如何在使用 Webpack 的项目中利用 Wasm 模块
- 如何使用 `npm` 库为 WebAssembly 模块编写单元测试

为什么选择 Node.js？

在第三章中，描述了 Node.js 作为异步事件驱动的 JavaScript 运行时，这是从官方网站上获取的定义。然而，Node.js 代表的是我们构建和管理 Web 应用程序方式的深刻转变。在本节中，我们将讨论 WebAssembly 和 Node.js 之间的关系，以及为什么这两种技术如此互补。

无缝集成

Node.js 在 Google 的 V8 JavaScript 引擎上运行，该引擎驱动着 Google Chrome。由于 V8 的 WebAssembly 实现遵循核心规范，因此您可以使用与浏览器相同的 API 与 WebAssembly 模块进行交互。您可以使用 Node.js 的 `fs` 模块将 `.wasm` 文件的内容读入缓冲区，然后对结果调用 `instantiate()`，而不是执行 `.wasm` 文件的 `fetch` 调用。

互补技术

JavaScript 在服务器端也存在一些限制。使用 WebAssembly 的卓越性能可以优化昂贵的计算或处理大量数据。作为一种脚本语言，JavaScript 擅长自动化简单的任务。您可以编写一个脚本来将 C/C++ 编译为 Wasm 文件，将其复制到 `build` 文件夹中，并在浏览器中查看变化（如果使用类似 `Browsersync` 的工具）。

使用 npm 进行开发

Node.js 拥有一个庞大的工具和库生态系统，以 `npm` 的形式存在。Sven Sauleau 和其他开源社区成员创建了 `webassemblyjs`，这是一个使用 Node.js 构建的 WebAssembly 工具套件。`webassemblyjs` 网站 webassembly.js.org 包括标语 *WebAssembly* 的工具链。目前有超过 20 个 `npm` 包可执行各种任务并辅助开发，例如 ESLint 插件、AST 验证器和格式化程序。AssemblyScript 是一种 TypeScript 到 WebAssembly 的编译器，允许您编写高性能的代码，无需学习 C 或 C++ 即可编译为 Wasm 模块。Node.js 社区显然对 WebAssembly 的成功充满信心。

使用 Express 进行服务器端 WebAssembly

Node.js 可以以多种方式用于增加 WebAssembly 项目的价值。在本节中，我们将通过一个示例 Node.js 应用程序来介绍集成 WebAssembly 的方法。该应用程序使用 Express 和一些简单的路由来调用编译后的 Wasm 模块中的函数。

项目概述

该项目重用了我们在第七章中构建的应用程序（从头开始创建应用程序）的一些代码，以演示如何将 Node.js 与 WebAssembly 一起使用。本节的代码位于 `learn-webassembly` 存储库中的 `/chapter-09-node/server-example` 文件夹中。我们将审查与 Node.js 直接相关的应用程序部分。以下结构代表项目的文件结构：

```
└── /lib
    └── main.c
└── /src
    ├── Transaction.js
    ├── /assets
    │   ├── db.json
    │   ├── main.wasm
    │   └── memory.wasm
    ├── assign-routes.js
    ├── index.js
    └── load-assets.js
└── package.json
└── package-lock.json
└── requests.js
```

关于依赖项，该应用程序使用 `express` 和 `body-parser` 库来设置路由并解析来自请求主体的 JSON。对于数据管理，它使用 `lowdb`，这是一个提供读取和更新 JSON 文件方法的库。JSON 文件位于 `/src/assets/db.json` 中，其中包含了从 Cook the Books 数据集中略微修改的数据。我们使用 `nodemon` 来监视 `/src` 文件夹中的更改并自动重新加载应用程序。我们使用 `rimraf` 来管理文件删除。该库作为依赖项包含在事件中，以防您没有在第三章中全局安装它，设置开发环境。最后，`node-fetch` 库允许我们在测试应用程序时使用 fetch API 进行 HTTP 请求。

为了简化 JavaScript 和 C 文件中的功能，`rawAmount` 和 `cookedAmount` 字段被替换为单个 `amount` 字段，`category` 字段现在是 `categoryId`，它映射到 `db.json` 中的 `categories` 数组。

Express 配置

应用程序在 `/src/index.js` 中加载。该文件的内容如下所示：

```
const express = require('express');
const bodyParser = require('body-parser');
const loadAssets = require('./load-assets');
const assignRoutes = require('./assign-routes');

// If you preface the npm start command with PORT=[Your Port]
on
// macOS/Ubuntu or set PORT=[Your Port] on Windows, it will c
hange the port
// that the server is running on, so PORT=3001 will run the a
pp on
// port 3001:
const PORT = process.env.PORT || 3000;

const startApp = async () => {
  const app = express();

  // Use body-parser for parsing JSON in the body of a reques
t:
  app.use(bodyParser.urlencoded({ extended: true }));
  app.use(bodyParser.json());
```

```

// Instantiate the Wasm module and local database:
const assets = await loadAssets();

// Setup routes that can interact with Wasm and the database:
assignRoutes(app, assets);

// Start the server with the specified port:
app.listen(PORT, (err) => {
  if (err) return Promise.reject(err);
  return Promise.resolve();
});

};

startApp()
  .then(() => console.log(`Server is running on port ${PORT}`)
  )
  .catch(err => console.error(`An error occurred: ${err}`));

```

该文件设置了一个新的 Express 应用程序，添加了 `body-parser` 中间件，加载了模拟数据库和 Wasm 实例，并分配了路由。让我们继续讨论在浏览器和 Node.js 中实例化 Wasm 模块的区别。

使用 Node.js 实例化 Wasm 模块

Wasm 文件在 `/src/load-assets.js` 中实例化。我们使用了来自 Cook the Books 的 `memory.wasm` 文件，但 `/assets/main.wasm` 文件是从位于 `/lib` 文件夹中的稍微不同版本的 `main.c` 编译而来。`loadWasm()` 函数执行的操作与 Cook the Books 中的 Wasm 初始化代码相同，但是将 `bufferSource` 传递给 `WebAssembly.instantiate()` 的方法不同。让我们通过查看 `load-assets.js` 文件中 `loadWasm()` 函数的部分代码来进一步了解这一点：

```

const fs = require('fs');
const path = require('path');

const assetsPath = path.resolve(__dirname, 'assets');

```

```

const getBufferSource = fileName => {
  const filePath = path.resolve(assetsPath, fileName);
  return fs.readFileSync(filePath); // <- Replaces the fetch()
  () and .arrayBuffer()
};

// We're using async/await because it simplifies the Promise
syntax
const loadWasm = async () => {
  const wasmMemory = new WebAssembly.Memory({ initial: 1024
});

  const memoryBuffer = getBufferSource('memory.wasm');
  const memoryInstance = await WebAssembly.instantiate(memory
Buffer, {
  env: {
    memory: wasmMemory
  }
});
...

```

为了详细说明区别，以下是使用 `fetch` 实例化模块的一些代码：

```

fetch('main.wasm')
  .then(response => {
    if (response.ok) return response.arrayBuffer();
    throw new Error('Unable to fetch WebAssembly file');
  })
  .then(bytes => WebAssembly.instantiate(bytes, importObj));

```

在使用 Node.js 时，`fetch` 调用被 `fs.readFileSync()` 函数替换，不再需要 `arrayBuffer()` 函数，因为 `fs.readFileSync()` 返回一个可以直接传递给 `instantiate()` 函数的缓冲区。一旦 Wasm 模块被实例化，我们就可以开始与实例交互。

创建模拟数据库

`load-assets.js` 文件还包含了创建模拟数据库实例的方法：

```
const loadDb = () => {
  const dbPath = path.resolve(assetsPath, 'db.json');
  const adapter = new FileSync(dbPath);
  return low(adapter);
};
```

`loadDb()` 函数将 `/assets/db.json` 的内容加载到 `lowdb` 的实例中。从 `load-assets.js` 中默认导出的函数调用了 `loadWasm()` 和 `loadDb()` 函数，并返回一个包含模拟数据库和 Wasm 实例的对象：

```
module.exports = async function loadAssets() {
  const db = loadDb();
  const wasmInstance = await loadWasm();
  return {
    db,
    wasmInstance
  };
};
```

接下来，我将使用术语数据库来指代访问 `db.json` 文件的 `lowdb` 实例。现在资产已加载，让我们回顾一下应用程序如何与它们交互。

与 WebAssembly 模块交互

与数据库和 Wasm 实例的交互发生在 `/src` 文件夹中的两个文件中：`Transaction.js` 和 `assign-routes.js`。在我们的示例应用程序中，所有与 API 的通信都是通过 HTTP 请求完成的。向特定端点发送请求将触发服务器上与数据库/Wasm 实例的一些交互。让我们从直接与数据库和 Wasm 实例交互的 `Transaction.js` 开始回顾。

在 `Transaction.js` 中包装交互

就像 Cook the Books 一样，有一个类包装了 Wasm 交互代码并提供了一个清晰的接口。`Transaction.js` 的内容与 Cook the Books 中的 `/src/store/WasmTransactions.js` 的内容非常相似。大部分更改是为了适应交易记录中存在 `categoryId` 和单个 `amount` 字段（不再有原始和烹饪金额）。还添加了与数据库交互的附加功能。例如，这是一个编辑现有交易的函数，既在数据库中，又在 Wasm 实例的链接列表中：

```

getValidAmount(transaction) {
  const { amount, type } = transaction;
  return type === 'Withdrawal' ? -Math.abs(amount) : amount;
}

edit(transactionId, contents) {
  const updatedTransaction = this.db.get('transactions')
    .find({ id: transactionId })
    .assign(contents)
    .write();

  const { categoryId, ...transaction } = updatedTransaction;
  const amount = this.getValidAmount(transaction);
  this.wasmInstance._editTransaction(transactionId, categoryId, amount);

  return updatedTransaction;
}

```

`edit()` 函数使用 `contents` 参数中的值更新与 `transactionId` 参数对应的数据库记录。
`this.db` 是在 `load-assets.js` 文件中创建的数据库实例。由于 `updatedTransaction` 记录上可用 `categoryId` 字段，我们可以直接将其传递给 `this.wasmInstance._editTransaction()`。当创建 `Transaction` 的新实例时，它会被传递到构造函数中。

在 `assign-routes.js` 中的交易操作

`assign-routes.js` 文件定义了路由并将它们添加到 `index.js` 中创建的 `express` 实例 (`app`) 中。在 Express 中，路由可以直接在 `app` 上定义（例如 `app.get()`），也可以通过使用 `Router` 来定义。在这种情况下，使用了 `Router` 来将多个方法添加到相同的路由路径上。以下代码取自 `assign-routes.js` 文件，创建了一个 `Router` 实例并添加了两个路由：一个 `GET` 路由返回所有交易，一个 `POST` 路由创建一个新的交易。

```

module.exports = function assignRoutes(app, assets) {
  const { db, wasmInstance } = assets;
  const transaction = new Transaction(db, wasmInstance);
  const transactionsRouter = express.Router();

```

```

transactionsRouter
  .route('/')
  .get((req, res) => {
    const transactions = transaction.findAll();
    res.status(200).send(transactions);
  })
  .post((req, res) => {
    const { body } = req;
    if (!body) {
      return res.status(400).send('Body of request is empty');
    }
    const newRecord = transaction.add(body);
    res.status(200).send(newRecord);
  });
}

...
// Set the base path for all routes on transactionsRouter:
app.use('/api/transactions', transactionsRouter);
}

```

片段末尾的 `app.use()` 函数指定了在 `transactionsRouter` 实例上定义的所有路由都以 `/api/transactions` 为前缀。如果您在本地端口 `3000` 上运行应用程序，可以在浏览器中导航到 `http://localhost:3000/api/transactions`，并以 JSON 格式查看所有交易的数组。

从 `get()` 和 `post()` 函数的主体中可以看出，与任何交易记录的交互都被委托给了第 3 行创建的 `Transaction` 实例。这完成了我们对代码库相关部分的审查。每个文件都包含描述文件功能和目的的注释，因此在继续下一部分之前，您可能需要审查这些内容。在下一部分中，我们将构建、运行并与应用程序交互。

构建和运行应用程序

在构建和测试项目之前，您需要安装 `npm` 依赖项。在 `/server-example` 文件夹中打开终端并运行以下命令：

```
npm install
```

完成后，您可以继续进行构建步骤。

构建应用程序

在这个应用程序中，构建是指使用 `emcc` 命令将 `lib/main.c` 编译为 `.wasm` 文件。由于这是一个 Node.js 项目，我们可以使用 `package.json` 文件中的 `scripts` 键来定义任务。您仍然可以使用 VS Code 的任务功能，因为它会自动检测 `package.json` 文件中的脚本，并在选择任务时将它们呈现在任务列表中。以下代码包含了该项目 `package.json` 文件中 `scripts` 部分的内容：

```
"scripts": {  
  "prebuild": "rimraf src/assets/main.wasm",  
  "build": "emcc lib/main.c -Os -s WASM=1 -s SIDE_MODULE=1  
           -s BINARYEN_ASYNC_COMPILATION=0 -s ALLOW_MEMORY_GROWTH=1  
           -o src/assets/main.wasm",  
  "start": "node src/index.js",  
  "watch": "nodemon src/* --exec 'npm start'"  
},
```

`build` 脚本被拆分成多行以便显示，因此您需要将这些行组合成有效的 JSON。`prebuild` 脚本会删除现有的 Wasm 文件，而 `build` 脚本会使用所需的标志运行 `emcc` 命令，将 `lib/main.c` 编译并将结果输出到 `src/assets/main.wasm`。要运行该脚本，请在 `/server-example` 文件夹中打开终端并运行以下命令：

```
npm run build
```

如果 `/src/assets` 文件夹中包含名为 `main.wasm` 的文件，则构建已成功完成。如果发生错误，终端应提供错误的描述以及堆栈跟踪。

你可以创建 `npm` 脚本，在特定脚本之前或之后运行，方法是创建一个与相同名称的条目，并在前面加上 `pre` 或 `post`。例如，如果你想在 `build` 脚本完成后运行一个脚本，你可以创建一个名为 `"postbuild"` 的脚本，并指定你想要运行的命令。

启动和测试应用程序

如果你正在对应用程序进行更改或尝试修复错误，你可以使用 `watch` 脚本来监视 `/src` 文件夹中内容的任何更改，并在有更改时自动重新启动应用程序。由于我们只是运行和测试应用程序，所以可以使用 `start` 命令。在终端中，确保你在 `/server-example` 文件夹中，并运行以下命令：

```
npm start
```

你应该看到一个消息，上面写着 `服务器正在 3000 端口上运行`。现在你可以向服务器发送 HTTP 请求了。要测试应用程序，在 `server-example` 目录中打开一个新的终端实例，并运行以下命令：

```
node ./requests.js 1
```

这应该记录下对 `/api/transactions` 端点的 `GET` 调用的响应主体。`requests.js` 文件包含了允许你对所有可用路由进行请求的功能。`getFetchActionForId()` 函数返回一个带有端点和选值的对象，对应于 `assign-routes.js` 文件中的一个路由。`actionId` 是一个任意的数字，用于简化测试并减少运行命令时的输入量。例如，你可以运行以下命令：

```
node ./requests.js 5
```

它将记录下计算机与互联网类别的所有交易的总和。如果你想要其他类别的总和，可以向 `node` 命令传递额外的参数。要获取保险类别的所有交易总和，运行以下命令：

```
node ./requests.js 5 3
```

尝试通过每个请求（总共有八个）进行。如果你发出了一个添加、删除或编辑交易的请求，你应该在 `/src/assets/db.json` 文件中看到变化。这就是 Node.js 示例项目的全部内容。在下一节中，我们将利用 Webpack 来加载和与 Wasm 模块交互。

使用 Webpack 进行客户端 WebAssembly

Web 应用程序在复杂性和规模上继续增长。简单地提供一些手写的 HTML、CSS 和 JavaScript 文件对于大型应用程序来说是不可行的。为了管理这种复杂性，Web 开发人员使用捆绑器来实现模块化，确保浏览器兼容性，并减少 JavaScript 文件的大小。在本节中，我们将使用一种流行的捆绑器 Webpack 来利用 Wasm，而不使用 `emcc`。

项目概述

示例 Webpack 应用程序扩展了我们在第五章的编译 C 而不使用粘合代码部分中编写的 C 代码的功能，创建和加载 *WebAssembly* 模块。我们不再展示一个蓝色矩形在红色背景上弹跳，而是展示一个飞船在马头星云中弹跳。碰撞检测功能已经修改，以适应在矩形内弹跳，所以飞船的移动将是随机的。本节的代码位于 [learn-webassembly](#) 存储库中的 `/chapter-09-node/webpack-example` 文件夹中。项目的文件结构如下所示：

```
└── /src
    ├── /assets
    │   ├── background.jpg
    │   └── spaceship.svg
    ├── App.js
    ├── index.html
    ├── index.js
    ├── main.c
    └── styles.css
    ├── package.json
    ├── package-lock.json
    └── webpack.config.js
```

我们将在后面的章节中审查 Webpack 配置文件。现在，让我们花一点时间更详细地讨论 Webpack。

什么是 Webpack？

在过去的几年里，JavaScript 生态系统一直在迅速发展，导致不断涌现新的框架和库。捆绑器的出现使开发人员能够将 JavaScript 应用程序分成多个文件，而不必担心管理全局命名空间、脚本加载顺序或 HTML 文件中的一长串 `<script>` 标签。捆绑器将所有文件合并为一个文件，并解决任何命名冲突。

截至撰写本文时，Webpack 是前端开发中最流行的打包工具之一。然而，它的功能远不止于合并 JavaScript 文件。它还执行复杂的任务，如代码拆分和摇树（死代码消除）。Webpack 采用了插件架构，这导致了大量由社区开发的插件。在 [npm](#) 上搜索 Webpack 目前返回超过 12,000 个包！这个详尽的插件列表，加上其强大的内置功能集，使 Webpack 成为一个功能齐全的构建工具。

安装和配置 Webpack

在开始应用程序演示之前，在 `/webpack-example` 文件夹中打开终端并运行以下命令：

```
npm install
```

依赖概述

应用程序使用 Webpack 的版本 4（在撰写本文时为最新版本）来构建我们的应用程序。我们需要使用 Webpack 插件来加载应用程序中使用的各种文件类型，并使用 Babel 来利用较新的 JavaScript 功能。以下片段列出了我们在项目中使用的 `devDependencies`（来自 `package.json`）：

```
...
"devDependencies": {
  "@babel/core": "7.0.0-rc.1",
  "@babel/preset-env": "7.0.0-rc.1",
  "babel-loader": "8.0.0-beta.4",
  "cpp-wasm-loader": "0.7.7",
  "css-loader": "1.0.0",
  "file-loader": "1.1.11",
  "html-loader": "0.5.5",
  "html-webpack-plugin": "3.2.0",
  "mini-css-extract-plugin": "0.4.1",
  "rimraf": "2.6.2",
  "webpack": "4.16.5",
  "webpack-cli": "3.1.0",
  "webpack-dev-server": "3.1.5"
},
...
```

我为一些库指定了确切的版本，以确保应用程序能够成功构建和运行。任何以 `-loader` 或 `-plugin` 结尾的库都与 Webpack 一起使用。`cpp-wasm-loader` 库允许我们直接导入 C 或 C++ 文件，而无需先将其编译为 Wasm。Webpack 4 内置支持导入 `.wasm` 文件，但无法指定 `importObj` 参数，这是使用 Emscripten 生成的模块所必需的。

在 `webpack.config.js` 中配置加载器和插件

除了 JavaScript 之外，我们还在应用程序中使用了几种不同的文件类型：CSS、SVG、HTML 等。安装 `-loader` 依赖项只是问题的一部分——您还需要告诉 Webpack 如何加载它们。您还需要为已安装的任何插件指定配置详细信息。您可以在项目的根文件夹中的 `webpack.config.js` 文件中指定加载和配置详细信息。以下片段包含了 `/webpack-example/webpack.config.js` 的内容：

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            // We need this to use async/await:
            presets: [
              [
                '@babel/preset-env', {
                  targets: { node: '10' }
                }
              ]
            ]
          }
        },
      },
      {
        test: /\.html$/,
        use: {
          loader: 'html-loader',
          options: { minimize: true }
        }
      }
    ]
  }
}
```

```
  },
  {
    test: /\.css$/,
    use: [MiniCssExtractPlugin.loader, 'css-loader']
  },
  {
    test: /\.(c|cpp)$/,
    use: {
      loader: 'cpp-wasm-loader',
      options: {
        emitWasm: true
      }
    }
  },
  {
    test: /\.(png|jpg|gif|svg)$/,
    use: {
      loader: 'file-loader',
      options: {
        name: 'assets/[name].[ext]'
      }
    }
  }
],
},
plugins: [
  new HtmlWebpackPlugin({
    template: './src/index.html',
    filename: './index.html'
  }),
  // This is used for bundling (building for production):
  new MiniCssExtractPlugin({
    filename: '[name].css',
    chunkFilename: '[id].css'
})
]
```

```
];
};
```

`rules` 部分告诉 Webpack 使用哪个加载器来处理文件扩展名。数组中的第四项处理 C/C++ 文件（注意 `test` 字段值包含 `c|cpp`）。`HtmlWebpackPlugin` 获取 `/src/index.html` 的内容，添加任何所需的 `<script>` 标签，对其进行最小化，并在 `build` 文件夹中创建一个 `index.html`，默认为 `/dist`。`MiniCssExtractPlugin` 将任何导入的 CSS 复制到 `/dist` 文件夹中的单个 CSS 文件中。我们将在后面的部分中讨论如何构建项目，所以让我们继续进行应用程序代码的讲解，从 C 文件开始。

C 代码

由于我们可以直接导入 C 和 C++ 文件，因此 C 文件位于 `/src` 文件夹中。这个文件，`main.c`，包含了管理碰撞检测和移动飞船的逻辑。这段代码基于我们在第五章中创建的 `without-glue.c` 文件，创建和加载 WebAssembly 模块。我们不打算审查整个文件，只审查已更改并值得解释的部分。让我们从定义和声明部分开始，其中包括一个新的 `struct`：`Bounds`。

定义和声明

包含定义和声明部分的代码如下所示：

```
typedef struct Bounds {
    int width;
    int height;
} Bounds;

// We're using the term "Rect" to represent the rectangle the
// image occupies:
typedef struct Rect {
    int x;
    int y;
    int width;
    int height;
    // Horizontal direction of travel (L/R):
    char horizDir;
    // Vertical direction of travel (U/D):
```

```
    char vertDir;  
} Rect;  
  
struct Bounds bounds;  
struct Rect rect;
```

对现有的 `Rect` 定义添加了新属性，以适应灵活的大小和在 `x` 和 `y` 方向上的移动跟踪。我们定义了一个新的 `struct`，`Bounds`，并删除了现有的 `#define` 语句，因为 `<canvas>` 元素不再是具有静态尺寸的正方形。模块加载时声明了这两个元素的新实例。这些实例的尺寸属性在 `start()` 函数中赋值，接下来我们将介绍这个函数。

start() 函数

更新的 `start()` 函数，作为模块的入口点，如下所示：

```
EMSCRIPTEN_KEEPALIVE  
void start(int boundsWidth, int boundsHeight, int rectWidth,  
           int rectHeight) {  
    rect.x = 0;  
    rect.y = 0;  
    rect.horizDir = 'R';  
    rect.vertDir = 'D';  
    rect.width = rectWidth;  
    rect.height = rectHeight;  
    bounds.width = boundsWidth;  
    bounds.height = boundsHeight;  
    setIsRunning(true);  
}
```

从 JavaScript 调用的任何函数都以 `EMSCRIPTEN_KEEPALIVE` 语句为前缀。现在，我们将 `Bounds` 和 `Rect` 元素的宽度和高度作为参数传递给 `start()` 函数，然后将其分配给本地的 `bounds` 和 `rect` 变量。这使我们可以轻松地更改任一元素的尺寸，而无需对碰撞检测逻辑进行任何更改。在这个应用程序的上下文中，`rect` 表示飞船图像所在的矩形。我们设置了 `rect` 的默认水平和垂直方向，使图像最初向右和向下移动。让我们继续进行 `rect` 移动/碰撞检测代码。

更新 `updateRectLocation()` 函数

与碰撞检测和 `Rect` 移动相关的代码在 `updateRectLocation()` 函数中处理，如下所示：

```
/*
 * Updates the rectangle location by +/- 1px in the x or y ba
sed on
 * the current location.
*/
void updateRectLocation() {
    // Determine if the bounding rectangle has "bumped" into
either
    // the left/right side or top/bottom side. Depending on w
hich side,
    // flip the direction:
    int xBouncePoint = bounds.width - rect.width;
    if (rect.x == xBouncePoint) rect.horizDir = 'L';
    if (rect.x == 0) rect.horizDir = 'R';

    int yBouncePoint = bounds.height - rect.height;
    if (rect.y == yBouncePoint) rect.vertDir = 'U';
    if (rect.y == 0) rect.vertDir = 'D';

    // If the direction has changed based on the x and y
    // coordinates, ensure the x and y points update
    // accordingly:
    int horizIncrement = 1;
    if (rect.horizDir == 'L') horizIncrement = -1;
    rect.x = rect.x + horizIncrement;

    int vertIncrement = 1;
    if (rect.vertDir == 'U') vertIncrement = -1;
    rect.y = rect.y + vertIncrement;
}
```

这段代码与我们在第五章中编写的代码的主要区别是碰撞检测逻辑。现在，函数不仅仅是水平跟踪 `rect` 实例的位置，并在其击中右边界时改变方向，而是现在函数同时跟踪水平和垂直方向，并独立管理每个方向。虽然这不是最高效的算法，但它确实实现了确保飞船在遇到 `<canvas>` 边缘时改变方向的目标。

JavaScript 代码

我们应用程序唯一的生产依赖是 Vue。虽然应用程序只包含一个组件，但 Vue 使得管理数据、函数和组件生命周期比手动操作简单得多。`index.js` 文件包含了 Vue 初始化代码，而渲染和应用程序逻辑在 `/src/App.js` 中。这个文件有很多部分，所以我们将像在一节一样分块审查代码。让我们从 `import` 语句开始。

导入语句

以下代码演示了 Webpack 加载器的工作原理：

```
// This is loaded using the css-loader dependency:  
import './styles.css';  
  
// This is loaded using the cpp-wasm-loader dependency:  
import wasm from './main.c';  
  
// These are loaded using the file-loader dependency:  
import backgroundImage from './assets/background.jpg';  
import spaceshipImage from './assets/spaceship.svg';
```

我们在 `webpack.config.js` 文件中配置的加载器知道如何处理 CSS、C 和图像文件。现在我们有了所需的资源，我们可以开始定义我们的组件状态。

组件状态

以下代码在 `data()` 函数中初始化了组件的本地状态：

```
export default {  
  data() {  
    return {  
      instance: null,  
      bounds: { width: 800, height: 592 },
```

```
    rect: { width: 200, height: 120 },
    speed: 5
  };
},
...

```

虽然 `bounds` 和 `rect` 属性永远不会改变，但我们在本地状态中定义它们，以便将组件使用的所有数据保存在一个位置。`speed` 属性决定了飞船在 `<canvas>` 上移动的速度，并且范围为 `1` 到 `10`。`instance` 属性初始化为 `null`，但将用于访问编译后的 Wasm 模块的导出函数。让我们继续进行编译 Wasm 文件并填充 `<canvas>` 的 Wasm 初始化代码。

Wasm 初始化

编译 Wasm 文件并填充 `<canvas>` 元素的代码如下所示：

```
methods: {
  // Create a new Image instance to pass into the drawImage function
  // for the <canvas> element's context:
  loadImage(imageSrc) {
    const loadedImage = new Image();
    loadedImage.src = imageSrc;
    return new Promise((resolve, reject) => {
      loadedImage.onload = () => resolve(loadedImage);
      loadedImage.onerror = () => reject();
    });
  },
  // Compile/load the contents of main.c and assign the resulting
  // Wasm module instance to the components this.instance property:
  async initializeWasm() {
    const ctx = this.$refs.canvas.getContext('2d');

    // Create Image instances of the background and spaceshi
```

```

p.
    // These are required to pass into the ctx.drawImage() function:
    const [bouncer, background] = await Promise.all([
        this.loadImage(spaceshipImage),
        this.loadImage(backgroundImage)
    ]);

    // Compile the C code to Wasm and assign the resulting
    // module.exports to this.instance:
    const { width, height } = this.bounds;
    return wasm
        .init(imports => ({
            ...imports,
            _jsFillRect(x, y, w, h) {
                ctx.drawImage(bouncer, x, y, w, h);
            },
            _jsClearRect() {
                ctx.drawImage(background, 0, 0, width, height);
            }
        }))
        .then(module => {
            this.instance = module.exports;
            return Promise.resolve();
        });
},
...

```

在组件的 `methods` 键中定义了其他函数，但现在我们将专注于将导入的 C 文件编译为 Wasm 的代码。在为飞船和背景图像创建 `Image` 实例之后，将 `main.c` 文件（导入为 `.wasm`）编译为 Wasm 模块，并将结果的 `exports` 分配给 `this.instance`。完成这些操作后，可以从导出的 Wasm 模块中调用 `start()` 函数。由于 `initializeWasm()` 函数调用了 `<canvas>` 元素的 `getContext()` 函数，因此在调用此函数之前，组件需要被挂载。让我们审查 `methods` 定义的其余部分和 `mounted()` 事件处理程序。

组件挂载

其余的 `methods` 定义和 `mounted()` 事件处理程序函数如下所示：

```
...
// Looping function to move the spaceship across the canvas.
loopRectMotion() {
  setTimeout(() => {
    this.instance.moveRect();
    if (this.instance.getIsRunning()) this.loopRectMotion();
  }, 15 - this.speed);
},
// Pauses/resumes the spaceship's movement when the button is
// clicked:
onActionClick(event) {
  const newIsRunning = !this.instance.getIsRunning();
  this.instance.setIsRunning(newIsRunning);
  event.target.innerHTML = newIsRunning ? 'Pause' : 'Resume';
  if (newIsRunning) this.loopRectMotion();
}
},
mounted() {
  this.initializeWasm().then(() => {
    this.instance.start(
      this.bounds.width,
      this.bounds.height,
      this.rect.width,
      this.rect.height
    );
    this.loopRectMotion();
  });
},
```

一旦 Wasm 模块被编译，`start()` 函数就可以在 `this.instance` 上访问。`bounds` 和 `rect` 尺寸被传递到 `start()` 函数中，然后调用 `loopRectFunction()` 来开始移动飞船。

`onActionClick()` 事件处理程序函数根据飞船当前是否在运动来暂停或恢复飞船的移动。

`loopRectMotion()` 函数的工作方式与第五章中的示例代码相同，[创建和加载 WebAssembly 模块](#)，只是现在速度是可调节的。`15 - this.speed` 的计算可能看起来有点奇怪。由于图像的移动速度是基于函数调用之间经过的时间，增加这个数字实际上会减慢飞船的速度。因此，`this.speed` 从 `15` 中减去，选择 `15` 是因为它略大于 `10`，但不会在将 `this.speed` 增加到最大值时使飞船变得模糊。这就是组件逻辑；让我们继续到代码的渲染部分，其中定义了 `template`。

组件渲染

`template` 属性的内容，决定了要渲染的内容，如下所示：

```
template: `<div class="flex column">
  <h1>SPACE WASM!</h1>
  <canvas
    ref="canvas"
    :height="bounds.height"
    :width="bounds.width">
  </canvas>
  <div class="flex controls">
    <div>
      <button class="defaultText" @click="onActionClick">
        Pause
      </button>
    </div>
    <div class="flex column">
      <label class="defaultText" for="speed">Speed: {{speed}}</label>
      <input
        v-model="speed"
        id="speed"
        type="range"
        min="1">
    </div>
  </div>
</div>
```

```
    max="10"
    step="1">
</div>
</div>
</div>
```

由于我们使用了 Vue，我们可以将 HTML 元素的属性和事件处理程序绑定到组件中定义的属性和方法。除了一个暂停/恢复按钮，还有一个范围 `<input>`，允许您改变速度。通过将其向左或向右滑动，您可以减慢或加快飞船的速度，并立即看到变化。这就结束了我们的回顾；让我们看看 Webpack 如何用来构建或运行应用程序。

构建和运行应用程序

使用 `cpp-wasm-loader` 库可以消除构建步骤生成 Wasm 模块的需要，但我们仍然需要将应用程序捆绑起来进行分发。在 `package.json` 的 `scripts` 部分，有一个 `build` 和 `start` 脚本。运行 `build` 脚本会执行生成捆绑包的 `webpack` 命令。为了确保这一切都正常工作，打开 `/webpack-example` 文件夹中的终端实例，并运行以下命令：

```
npm run build
```

第一次运行项目构建可能需要一分钟。这可能归因于 Wasm 编译步骤。但是，后续的构建应该会快得多。如果构建成功，您应该会看到一个新创建的 `/dist` 文件夹，其中包含以下内容：

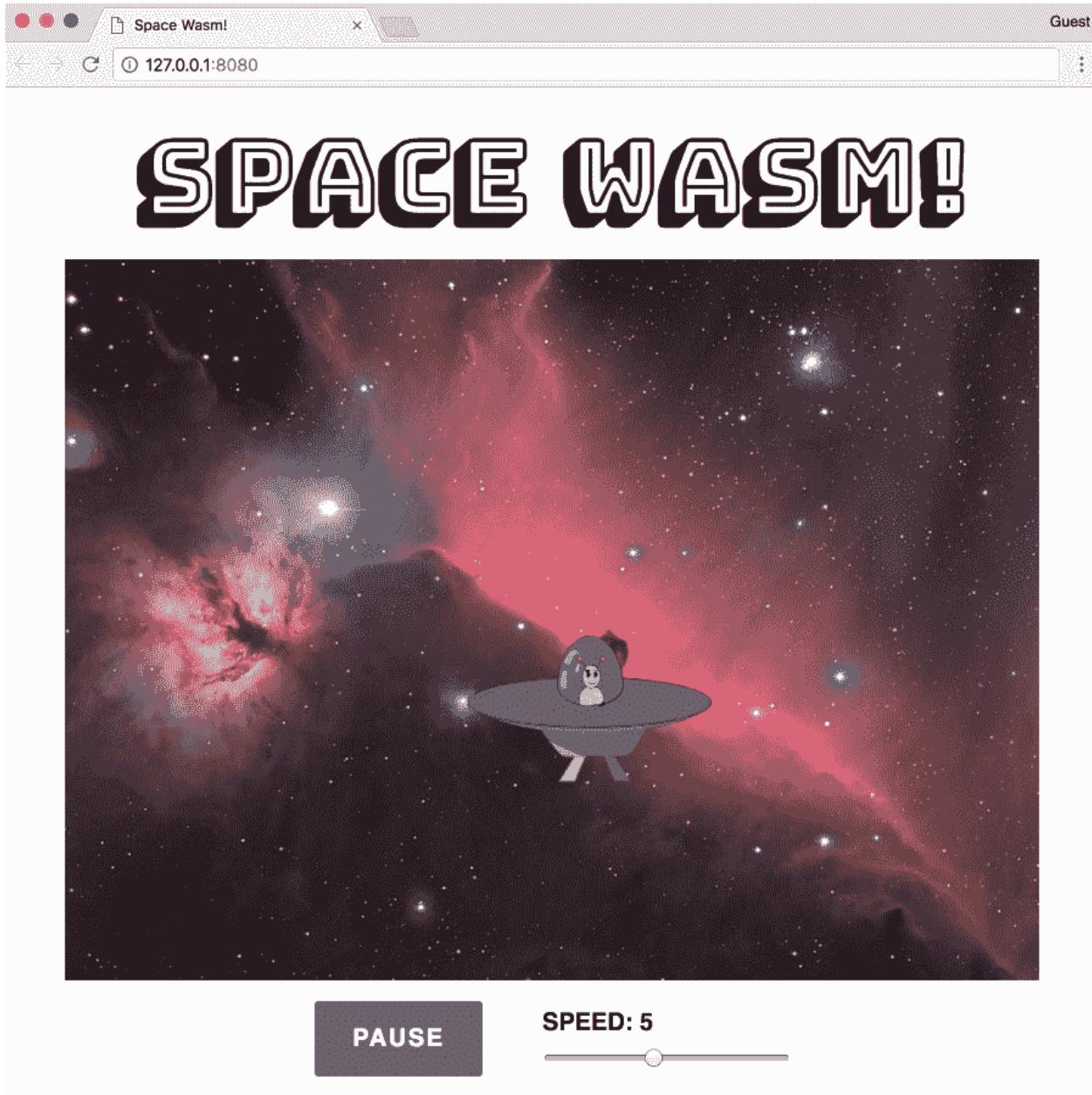
```
└── /assets
    ├── background.jpg
    └── spaceship.svg
└── index.html
└── main.css
└── main.js
└── main.wasm
```

测试构建

让我们尝试构建以确保一切都正常工作。在终端实例中运行以下命令来启动应用程序：

```
serve -l 8080 dist
```

如果在浏览器中导航到 <http://127.0.0.1:8080/index.html>，您应该会看到这个：



Webpack 应用程序在浏览器中运行

飞船图像（取自 commons.wikimedia.org/wiki/File:Alien_Spaceship_-_SVG_Vector.svg）在 Horsehead 星云背景图像（取自

commons.wikimedia.org/wiki/File:Horsehead_Nebula_Christmas_2017_Deography.jpg) 的范围内弹来弹去。当按下暂停按钮时，按钮的标题会更改为恢复，飞船停止移动。再次按下按钮将会将标题更改回暂停，并且飞船将再次开始移动。调整速度滑块会增加或减少飞船的速度。

运行启动脚本

应用程序已安装 `webpack-dev-server` 库，它的操作方式类似于 `Browsersync`。该库使用 LiveReloading，在您对 `/src` 中的文件进行任何更改时会自动更新应用程序。由于我们使用了 C 和 C++ 文件的 Webpack 加载器，因此如果您更改了 C 文件，自动更新事件也会触发。运行以下命令来启动应用程序并监视更改：

```
npm start
```

当构建完成时，浏览器窗口应该会自动打开，然后将您引导到运行的应用程序。要查看实时重新加载功能的操作，请尝试将 `main.c` 中的 `setIsRunning()` 函数中的 `isRunning` 变量的值设置为 `false`，而不是 `newIsRunning`：

```
EMSCRIPTEN_KEEPALIVE
void setIsRunning(bool newIsRunning) {
    // isRunning = newIsRunning;

    // Set the value to always false:
    isRunning = false;
}
```

飞船应该被卡在左上角。如果您将其改回，飞船将重新开始移动。在下一节中，我们将编写 JavaScript 单元测试来测试 WebAssembly 模块。

使用 Jest 测试 WebAssembly 模块

经过充分测试的代码可以防止回归错误，简化重构，并减轻添加新功能时的一些挫折感。一旦您编译了一个 Wasm 模块，您应该编写测试来确保它的功能符合预期，即使您已经为您从中编译出来的 C、C++ 或 Rust 代码编写了测试。在本节中，我们将使用 **Jest**，一个 JavaScript 测试框架，来测试编译后的 Wasm 模块中的函数。

正在测试的代码

此示例中使用的所有代码都位于 `/chapter-09-node/testing-example` 文件夹中。代码和相应的测试非常简单，不代表真实应用程序，但旨在演示如何使用 Jest 进行测试。以下代码表示 `/testing-example` 文件夹的文件结构：

```
|── /src
|   ├── /__tests__
|   |   └── main.test.js
|   └── main.c
└── package.json
└── package-lock.json
```

我们将要测试的 C 文件的内容，`/src/main.c`，如下所示：

```
int addTwoNumbers(int leftValue, int rightValue) {
    return leftValue + rightValue;
}

float divideTwoNumbers(float leftValue, float rightValue) {
    return leftValue / rightValue;
}

double findFactorial(float value) {
    int i;
    double factorial = 1;

    for (i = 1; i <= value; i++) {
        factorial = factorial * i;
    }
    return factorial;
}
```

文件中的所有三个函数都执行简单的数学运算。`package.json` 文件包含一个脚本，用于将 C 文件编译为 Wasm 文件进行测试。运行以下命令来编译 C 文件：

```
npm run build
```

`/src` 目录中应该有一个名为 `main.wasm` 的文件。让我们继续描述测试配置步骤。

测试配置

在这个示例中，我们将使用 Jest 作为唯一的依赖项，Jest 是 Facebook 开发的 JavaScript 测试框架。Jest 是测试的绝佳选择，因为它包含大多数您需要的功能，如覆盖率、断言和模拟等。在大多数情况下，您可以在零配置的情况下使用它，具体取决于您的应用程序的复杂性。如果您想了解更多，请访问 Jest 的网站 jestjs.io。在 `/chapter-09-node/testing-example` 文件夹中打开一个终端实例，并运行以下命令来安装 Jest：

```
npm install
```

在 `package.json` 文件中，`scripts` 部分有三个条目：`build`、`pretest` 和 `test`。`build` 脚本使用所需的标志执行 `emcc` 命令，将 `/src/main.c` 编译为 `/src/main.wasm`。`test` 脚本使用 `--verbose` 标志执行 `jest` 命令，为每个测试套件提供额外的细节。`pretest` 脚本只是运行 `build` 脚本，以确保在运行任何测试之前存在 `/src/main.wasm`。

测试文件审查

让我们来看一下位于 `/src/_tests_/main.test.js` 的测试文件，并审查代码的每个部分的目的。测试文件的第一部分实例化 `main.wasm` 文件，并将结果分配给本地的 `wasmInstance` 变量：

```
const fs = require('fs');
const path = require('path');

describe('main.wasm Tests', () => {
  let wasmInstance;

  beforeAll(async () => {
    const wasmPath = path.resolve(__dirname, '.', 'main.wasm');
    const buffer = fs.readFileSync(wasmPath);
    const results = await WebAssembly.instantiate(buffer, {
      env: {
        memoryBase: 0,
        tableBase: 0,
```

```
        memory: new WebAssembly.Memory({ initial: 1024 }),  
        table: new WebAssembly.Table({ initial: 16, element:  
          'anyfunc' }),  
        abort: console.log  
      }  
    );  
    wasmInstance = results.instance.exports;  
  });  
  ...
```

Jest 提供了生命周期方法来执行任何设置或拆卸操作以便在运行测试之前进行。您可以指定在所有测试之前或之后运行的函数（`beforeAll()` / `afterAll()`），或者在每个测试之前或之后运行的函数（`beforeEach()` / `afterEach()`）。我们需要一个编译后的 Wasm 模块实例，从中我们可以调用导出的函数，因此我们将实例化代码放在 `beforeAll()` 函数中。

我们将整个测试套件包装在文件的 `describe()` 块中。Jest 使用 `describe()` 函数来封装相关测试套件，使用 `test()` 或 `it()` 来表示单个测试。以下是这个概念的一个简单示例：

```
const add = (a, b) => a + b;  
  
describe('the add function', () => {  
  test('returns 6 when 4 and 2 are passed in', () => {  
    const result = add(4, 2);  
    expect(result).toEqual(6);  
  });  
  
  test('returns 20 when 12 and 8 are passed in', () => {  
    const result = add(12, 8);  
    expect(result).toEqual(20);  
  });  
});
```

下一节代码包含了所有的测试套件和每个导出函数的测试：

```
...  
describe('the _addTwoNumbers function', () => {
```

```
    test('returns 300 when 100 and 200 are passed in', () =>
{
    const result = wasmInstance._addTwoNumbers(100, 200);
    expect(result).toEqual(300);
});

    test('returns -20 when -10 and -10 are passed in', () =>
{
    const result = wasmInstance._addTwoNumbers(-10, -10);
    expect(result).toEqual(-20);
});
});

describe('the _divideTwoNumbers function', () => {
    test.each([
        [10, 100, 10],
        [-2, -10, 5],
    ])('returns %f when %f and %f are passed in', (expected,
a, b) => {
        const result = wasmInstance._divideTwoNumbers(a, b);
        expect(result).toEqual(expected);
    });
}

    test('returns ~3.77 when 20.75 and 5.5 are passed in', () => {
        const result = wasmInstance._divideTwoNumbers(20.75, 5.
5);
        expect(result).toBeCloseTo(3.77, 2);
    });
});

describe('the _findFactorial function', () => {
    test.each([
        [120, 5],
        [362880, 9.2],
    ])('returns %p when %p is passed in', (expected, input) =
```

```
> {
    const result = wasmInstance._findFactorial(input);
    expect(result).toEqual(expected);
  });
});
});
});
```

第一个 `describe()` 块，用于 `_addTwoNumbers()` 函数，有两个 `test()` 实例，以确保函数返回作为参数传入的两个数字的总和。接下来的两个 `describe()` 块，用于 `_divideTwoNumbers()` 和 `_findFactorial()` 函数，使用了 Jest 的 `.each` 功能，允许您使用不同的数据运行相同的测试。`expect()` 函数允许您对作为参数传入的值进行断言。最后一个 `_divideTwoNumbers()` 测试中的 `.toBeCloseTo()` 断言检查结果是否在 `3.77` 的两个小数位内。其余使用 `.toEqual()` 断言来检查相等性。

使用 Jest 编写测试相对简单，运行测试甚至更容易！让我们尝试运行我们的测试，并查看 Jest 提供的一些 CLI 标志。

运行测试

要运行测试，请在 `/chapter-09-node/testing-example` 文件夹中打开终端实例，并运行以下命令：

```
npm test
```

您应该在终端中看到以下输出：

```
main.wasm Tests
  the _addTwoNumbers function
    ✓ returns 300 when 100 and 200 are passed in (4ms)
    ✓ returns -20 when -10 and -10 are passed in
  the _divideTwoNumbers function
    ✓ returns 10 when 100 and 10 are passed in
    ✓ returns -2 when -10 and 5 are passed in (1ms)
    ✓ returns ~3.77 when 20.75 and 5.5 are passed in
  the _findFactorial function
    ✓ returns 120 when 5 is passed in (1ms)
    ✓ returns 362880 when 9.2 is passed in
```

```
Test Suites: 1 passed, 1 total
Tests: 7 passed, 7 total
Snapshots: 0 total
Time: 1.008s
Ran all test suites.
```

如果您有大量的测试，可以从 `package.json` 中的 `test` 脚本中删除 `--verbose` 标志，并仅在需要时将标志传递给 `npm test` 命令。您可以将其他几个 CLI 标志传递给 `jest` 命令。以下列表包含一些常用的标志：

- `-bail`：在第一个失败的测试套件后立即退出测试套件
- `-coverage`：收集测试覆盖率，并在测试运行后在终端中显示
- `-watch`：监视文件更改并重新运行与更改文件相关的测试

您可以通过在 `--` 之后添加这些标志来将这些标志传递给 `npm` 测试命令。例如，如果您想使用 `--bail` 标志，您可以运行以下命令：

```
npm test -- --bail
```

您可以在官方网站上查看所有 CLI 选项的完整列表：jestjs.io/docs/en/cli。

总结

在本章中，我们讨论了将 WebAssembly 与 Node.js 集成的优势，并演示了 Node.js 如何在服务器端和客户端使用。我们评估了一个使用 Wasm 模块执行会计交易计算的 Express 应用程序。然后，我们审查了一个基于浏览器的应用程序，该应用程序利用 Webpack 从 C 文件中导入和调用函数，而无需编写任何 Wasm 实例化代码。最后，我们看到了如何利用 Jest 测试框架来测试编译模块并确保其正常运行。在第十章中，高级工具和即将推出的功能，我们将介绍高级工具，并讨论 WebAssembly 即将推出的功能。

问题

1. 将 WebAssembly 与 Node.js 集成的优势之一是什么？
2. Express 应用程序使用哪个库来读取和写入数据到 JSON 文件？

3. 在浏览器和 Node.js 中加载模块有什么区别？
4. 您可以使用什么技术在现有的 `npm` 脚本之前或之后运行一个 `npm` 脚本？
5. Webpack 执行的任务名称是什么，以消除死代码？
6. Webpack 中加载程序的目的是什么？
7. Jest 中 `describe()` 和 `test()` 函数之间的区别是什么？
8. 如何将额外的 CLI 标志传递给 `npm test` 命令？

进一步阅读

- Express: expressjs.com
- Webpack: webpack.js.org
- Jest API: jestjs.io/docs/en/api

第十章：高级工具和即将推出的功能

WebAssembly 的生态系统不断增长和发展。开发人员已经看到了 WebAssembly 的潜力。他们构建工具来改善开发体验或从他们选择的语言输出 Wasm 模块（尽管有一些限制）。

在本章中，我们将评估使 WebAssembly 运行的基础技术。我们还将审查您可以在浏览器中使用的工具，并介绍一种利用 Web Workers 的高级用例。最后，我们将快速审查即将推出的功能和 WebAssembly 路线图上的提案。

本章的目标是理解以下内容：

- WABT 和 Binaryen 如何适应构建过程以及它们可以用于什么
- 如何使用 LLVM 编译 WebAssembly 模块（而不是 Emscripten）
- WasmFiddle 等在线工具和其他有用的在线工具
- 如何利用 Web Workers 并行运行 WebAssembly
- 未来将集成到 WebAssembly 中的功能（提议和进行中）

WABT 和 Binaryen

WABT 和 Binaryen 允许开发人员使用源文件并开发 WebAssembly 工具。如果您有兴趣以更低的级别使用 WebAssembly，这些工具提供了实现这一目标的手段。在本节中，我们将更详细地评估这些工具，并审查每个工具的目的和功能。

WABT-WebAssembly 二进制工具包

WABT 的重点是对 WebAssembly 二进制 (.wasm) 文件和文本 (.wat) 文件进行操作，以及在这两种格式之间进行转换。WABT 提供了将 Wat 转换为 Wasm (wat2wasm) 和反之 (wasm2wat) 的工具，以及将 Wasm 文件转换为 C 源文件和头文件 (wasm2c) 的工具。您可以在 WABT GitHub 存储库的 README 文件中查看所有工具的完整列表 github.com/WebAssembly/wabt。

WABT 的一个示例用例是我们在第三章中安装的*VS Code WebAssembly Toolkit*扩展，设置开发环境。该扩展依赖于 WABT 来查看与.wasm 文件相关联的文本格式。存储库提供了 wat2wasm 和 wasm2wat 演示的链接，您可以使用这些演示来测试 Wat 程序的有效性或使用 JavaScript 与编译后的二进制文件进行交互。以下屏幕截图包含 wat2wasm 演示中的 Wat 和 JavaScript 实例化代码：

WAT

example: simple

Download

```
1 (module
2   (func $addTwo (param i32 i32) (result i32)
3     get_local 0
4     get_local 1
5     i32.add)
6   (export "addTwo" (func $addTwo)))
```

JS

```
1 const wasmInstance =
2   new WebAssembly.Instance(wasmModule, {});
3 const { addTwo } = wasmInstance.exports;
4 for (let i = 0; i < 10; i++) {
5   console.log(addTwo(i, i));
6 }
7 }
```

wat2wasm 的 Wat 和 JavaScript 加载代码的“simple”示例

在 JS 面板的第 3 行中，您可能已经注意到 `wasmInstance.exports` 中的 `addTwo()` 函数没有前缀 `_`。Emscripten 在编译过程中会自动添加 `_`。您可以通过将.wasm 文件转换为.wat 文件，更新函数名称，然后使用 WABT 将其转换回.wasm 文件来省略 `_`，尽管这不太实用。WABT 简化了将文本格式转换为二进制格式和反之的过程。如果您想构建 WebAssembly 的编译工具，您将使用 Binaryen，我们将在下一节中介绍。

Binaryen

Binaryen 的 GitHub 页面 github.com/WebAssembly/binaryen 将 Binaryen 描述为用 C++ 编写的 WebAssembly 编译器和工具链基础库。它旨在使编译到 WebAssembly 变得简单、快速和有效。它通过提供简单的 C API、内部 IR 和优化器来实现这些目标。与 WABT 一样，Binaryen 提供了一套广泛的工具，用于开发 WebAssembly 工具。以下列表描述了 Binaryen 提供的一部分工具：

- **wasm-shell**：能够加载和解释 WebAssembly
- **asm2wasm**：将 asm.js 代码编译为 Wasm 模块
- **wasm2js**：将 Wasm 模块编译为 JavaScript
- **wasm-merge**：将多个 Wasm 文件合并为一个
- **wasm.js**：包括 Binaryen 解释器、asm2wasm、Wat 解析器和其他 Binaryen 工具的 JavaScript 库
- **binaryen.js**：提供 Binaryen 工具链的 JavaScript 接口的 JavaScript 库

wasm.js 和 binaryen.js 工具对于对构建 WebAssembly 工具感兴趣的 JavaScript 开发人员特别有吸引力。`binaryen.js` 库可作为 `npm` 包使用（www.npmjs.com/package/binaryen）。

`binaryen.js` 的一个很好的示例是 AssemblyScript

（github.com/AssemblyScript/assemblyscript）。AssemblyScript 是 TypeScript 的严格类型子集，可生成 WebAssembly 模块。该库附带了一个 CLI，可以快速搭建新项目并管理构建步骤。在使用 LLVM 编译部分，我们将介绍如何使用 LLVM 编译 Wasm 模块。

使用 LLVM 编译

在第一章中，我们讨论了 Emscripten 的 EMSDK 和 LLVM 之间的关系。Emscripten 使用 LLVM 和 Clang 将 C/C++ 编译为 LLVM 位码。Emscripten 编译器（`emcc`）将该位码编译为 asm.js，然后传递给 Binaryen 生成 Wasm 文件。如果您有兴趣使用 LLVM，可以在不安装 EMSDK 的情况下将 C/C++ 编译为 Wasm。在本节中，我们将回顾使用 LLVM 启用 Wasm 编译的过程。在将一些示例 C++ 代码编译为 Wasm 文件后，我们将在浏览器中尝试它。

安装过程

如果要使用 LLVM 编译 WebAssembly 模块，需要安装和配置多个工具。使这些工具正确配合工作可能是一个费时费力的过程。幸运的是，有人经历了这个麻烦，使这个过程变得简单得多。Daniel Wirtz 创建了一个名为 `webassembly` 的 `npm` 包

（www.npmjs.com/package/webassembly），可以执行以下操作（带有相应的 CLI 命令）：

- 将 C/C++ 代码编译为 WebAssembly 模块 (`wa compile`)
- 将多个 WebAssembly 模块链接到一个模块 (`wa link`)
- 将 WebAssembly 模块反汇编为文本格式 (`wa disassemble`)
- 将 WebAssembly 文本格式组装为模块 (`wa assemble`)

该库在后台使用 Binaryen、Clang、LLVM 和其他 LLVM 工具。我们将全局安装此包，以确保我们可以访问 `wa` 命令。要安装，请打开终端实例并运行以下命令：

```
npm install -g webassembly
```

安装所需的依赖可能需要几分钟。完成后，运行以下命令验证安装：

```
wa
```

您应该在终端中看到以下内容：

The screenshot shows a terminal window with the following content:

```
λ wa
[
| webassembly v0.11.0 CLI
]

Compiles, links, assembles and disassembles WebAssembly modules.

For command specific usage instructions, type:

wa compile      or    wa-compile
wa link         or    wa-link
wa assemble     or    wa-assemble
wa disassemble  or    wa-disassemble

usage: wa <compile|link|assemble|disassemble> [options] file
```

wa 命令的输出

您应该准备好开始编译 Wasm 模块了。让我们继续进行示例代码。

示例代码

为了测试编译器，我们将使用第五章中无需胶水代码与 JavaScript 交互部分的 `without-glue.c` 文件的稍作修改的版本。此部分的代码位于 `learn-webassembly` 存储库的 `/chapter-10-advanced-tools/compile-with-llvm` 目录中。按照以下说明创建编译器测试所需的文件。让我们从 C++ 文件开始。

C++文件

在您的 `/book-examples` 目录中创建一个名为 `/compile-with-llvm` 的新目录。在 `/compile-with-llvm` 目录中创建一个名为 `main.cpp` 的新文件，并填充以下内容：

```
#include <stdbool.h>
#define BOUNDS 255
#define RECT_SIDE 50
#define BOUNCE_POINT (BOUNDS - RECT_SIDE)

bool isRunning = true;

typedef struct Rect {
    int x;
    int y;
    char direction;
} Rect;

struct Rect rect;

void updateRectLocation() {
    if (rect.x == BOUNCE_POINT) rect.direction = 'L';
    if (rect.x == 0) rect.direction = 'R';
    int incrementer = 1;
    if (rect.direction == 'L') incrementer = -1;
    rect.x = rect.x + incrementer;
    rect.y = rect.y + incrementer;
}

extern "C" {
extern int jsClearRect();
extern int jsFillRect(int x, int y, int width, int height);

__attribute__((visibility("default")))
void moveRect() {
    jsClearRect();
    updateRectLocation();
    jsFillRect(rect.x, rect.y, RECT_SIDE, RECT_SIDE);
}

__attribute__((visibility("default")))
}
```

```

bool getIsRunning() {
    return isRunning;
}

__attribute__((visibility("default")))
void setIsRunning(bool newIsRunning) {
    isRunning = newIsRunning;
}

__attribute__((visibility("default")))
void init() {
    rect.x = 0;
    rect.y = 0;
    rect.direction = 'R';
    setIsRunning(true);
}
}

```

该文件中的代码与第五章中的 `without-glue.c` 的内容几乎相同，创建和加载 WebAssembly 模块。文件中的注释已被删除，并且导入/导出的函数被包装在 `extern "C"` 块中。`__attribute__((visibility("default")))` 行是宏语句（类似于 `EMSCRIPTEN_KEEPALIVE`），它们确保在死代码消除步骤期间不会从编译输出中删除这些函数。与之前的示例一样，我们将通过 HTML 文件与编译后的 Wasm 模块进行交互。让我们接下来创建这个文件。

HTML 文件

在 `/compile-with-llvm` 目录中创建一个名为 `index.html` 的文件，并填充以下内容：

```

<!doctype html>
<html lang="en-us">
<head>
    <title>LLVM Test</title>
</head>
<body>
    <h1>LLVM Test</h1>

```

```
<canvas id="myCanvas" width="255" height="255"></canvas>
<div style="margin-top: 16px;">
    <button id="actionButton" style="width: 100px; height: 24
px;">
        Pause
    </button>
</div>
<script type="application/javascript">
    const canvas = document.querySelector('#myCanvas');
    const ctx = canvas.getContext('2d');

    const importObj = {
        env: {
            memoryBase: 0,
            tableBase: 0,
            memory: new WebAssembly.Memory({ initial: 256 }),
            table: new WebAssembly.Table({ initial: 8, element:
'anyfunc' }),
            abort: console.log,
            jsFillRect: function(x, y, w, h) {
                ctx.fillStyle = '#0000ff';
                ctx.fillRect(x, y, w, h);
            },
            jsClearRect: function() {
                ctx.fillStyle = '#ff0000';
                ctx.fillRect(0, 0, 255, 255);
            }
        }
    };

    WebAssembly.instantiateStreaming(fetch('main.wasm'), impo
rtObj)
        .then(({ instance }) => {
            const m = instance.exports;
            m.init();
        })
    </script>
```

```

const loopRectMotion = () => {
    setTimeout(() => {
        m.moveRect();
        if (!m.getIsRunning()) loopRectMotion();
    }, 20)
};

document.querySelector('#actionButton')
    .addEventListener('click', event => {
        const newIsRunning = !m.getIsRunning();
        m.setIsRunning(newIsRunning);
        event.target.innerHTML = newIsRunning ? 'Pause' : 'Start';
        if (newIsRunning) loopRectMotion();
    });
}

loopRectMotion();
});
</script>
</body>
</html>

```

该文件的内容与第五章中的 `without-glue.html` 文件非常相似，创建和加载 WebAssembly 模块。我们不再使用 `/common/load-wasm.js` 文件中的 `loadWasm()` 函数，而是使用 `WebAssembly.instantiateStreaming()` 函数。这使我们可以省略一个额外的 `<script>` 元素，并直接从 `/compile-with-llvm` 目录中提供文件。

在传递给 `importObj` 的 `jsFillRect` 和 `jsClearRect` 函数中省略了 `_`。我们也可以在 `instance.exports` 对象中省略这些函数的 `_`。LLVM 不会在模块中传入或传出的数据/函数前缀中使用 `_`。在下一节中，我们将编译 `main.cpp` 并在浏览器中与生成的 Wasm 文件交互。

编译和运行示例

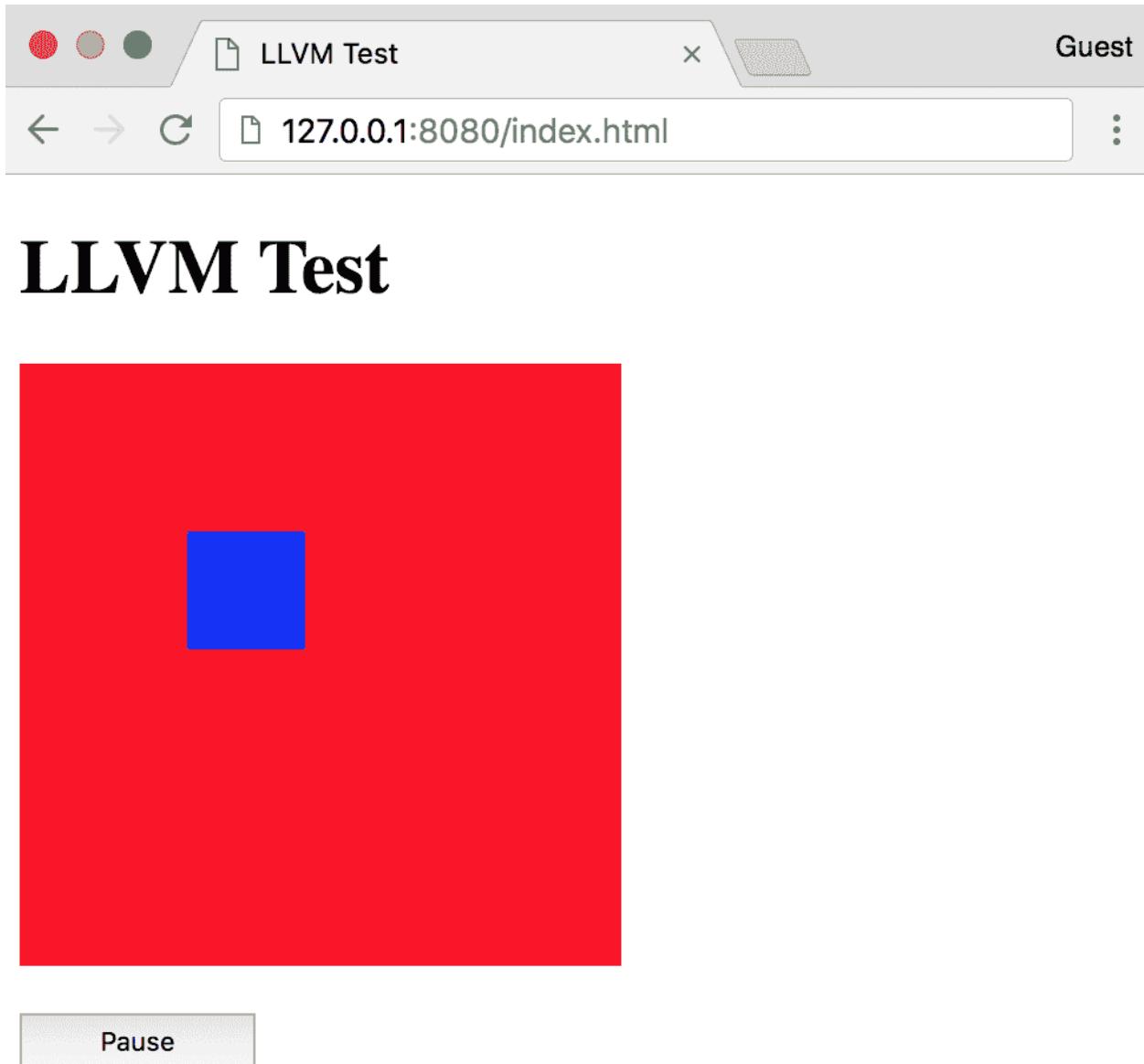
我们使用了 `webassembly npm` 包，并带有 `-g` 标志，因此终端中应该可以使用 `wa` 命令。在 `/compile-with-llvm` 目录中打开一个终端实例，并运行以下命令：

```
wa compile main.cpp -o main.wasm
```

您应该会在 VS Code 的文件资源管理器中的 `compile-with-llvm` 文件夹中看到一个名为 `main.wasm` 的文件。为了确保 Wasm 模块编译正确，运行以下命令：

```
serve -l 8080
```

如果您在浏览器中导航到 `http://127.0.0.1:8080/index.html`，您应该会看到以下内容：



在浏览器中运行的 LLVM 编译模块

在线工具

本地编译 WebAssembly 模块的安装和配置过程，诚然有点繁琐。幸运的是，有几种在线工具可供您在浏览器中开发和交互使用 WebAssembly。在本节中，我们将回顾这些工具，并讨论它们各自提供的功能。

WasmFiddle

在第二章的用 WasmFiddle 连接各部分部分中，WebAssembly 的要素 - Wat，Wasm 和 JavaScript API，我们使用 WasmFiddle 将一个简单的 C 函数编译为 Wasm，并使用 JavaScript 进行交互。WasmFiddle 提供了 C/C++ 编辑器，JavaScript 编辑器，Wat/x86 查看器和 JavaScript 输出面板。如果需要，还可以与 `<canvas>` 进行交互。WasmFiddle 使用 LLVM 生成 Wasm 模块，这就是为什么导入和导出不会以 `_` 为前缀。您可以在 wasdk.github.io/WasmFiddle 上使用 WasmFiddle。

WebAssembly Explorer

WebAssembly Explorer 位于 mbebenita.github.io/WasmExplorer，提供了与 WasmFiddle 类似功能。它允许您将 C 或 C++ 编译为 Wasm 模块，并查看相应的 Wat。但是，WebAssembly Explorer 提供了 WasmFiddle 中没有的额外功能。例如，您可以将 C 或 C++ 编译为 Wasm，并查看相应的 Firefox x86 和 LLVM x86 代码。您可以从代码示例列表中进行选择，并指定优化级别（`emcc` 中的 `-O` 标志）。它还提供了一个按钮，允许您将代码导入到 WasmFiddle 中：

The screenshot shows the WebAssembly Explorer interface at <https://mbebenita.github.io/WasmExplorer/>. The main area displays a C++11 code snippet for calculating factorials:

```

1 double fact(int i) {
2     long long n = 1;
3     for (; i > 0; i--) {
4         n *= i;
5     }
6     return (double)n;
7 }

```

The interface includes tabs for Options, C++11 -Os, COMPILER, Wat, ASSEMBLE, DOWNLOAD, and Firefox x86 Assembly. The ASSEMBLE tab is selected, showing the generated WebAssembly (Wasm) code:

```

1 (module
2   (table $0 anyfunc)
3   (memory $0 1)
4   (export "memory" (memory $0))
5   (export "_Z4facti" (func $._Z4facti))
6   (func $._Z4facti (; 0 ;) (param $0 i32)
7     (result f64)
8     (local $1 i64)
9     (local $2 i64)
10    (block $label$0
11      (br_if $label$0
12        (i32.lt_s
13          (get_local $0)
14          (i32.const 1)
15        )
16        (set_local $1
17          (i64.add
18            (i64.extend_s/i32
19              (get_local $0)
20            )
21            (i64.const 1)
22          )
23        )
24        (set_local $2
25          (i64.const 1)
26        )
27      (loop $label$1
28        (set_local $2
29          (i64.mul
30            (get_local $2)
31            (tee_local $1
32              (i64.add
33                (get_local $1)
34                (i64.const -1)
35              )
36            )
37          )
38        )
39      )
40    )
41  )
42)

```

The Firefox x86 Assembly tab shows the corresponding x86 assembly code:

```

- wasm-function[0]:
  sub rsp, 8
  cmp edi, 1
  jl 0x35
  0x00000d:
  movsd rax, edi
  add rax, 1
  mov ecx, 1
  0x000019:
  add rax, -1
  imul rax, rax
  cmp rax, 1
  jg 0x19
  0x000027:
  xorpd xmm0, xmm0
  cvtsi2sd xmm0, rcx
  jmp 0x3d
  0x000035:
  movsd xmm0, qword ptr [rip + 0x803]
  0x00003d:
  nop
  add rsp, 8
  ret

```

The bottom section shows a Console log with various compilation messages:

```

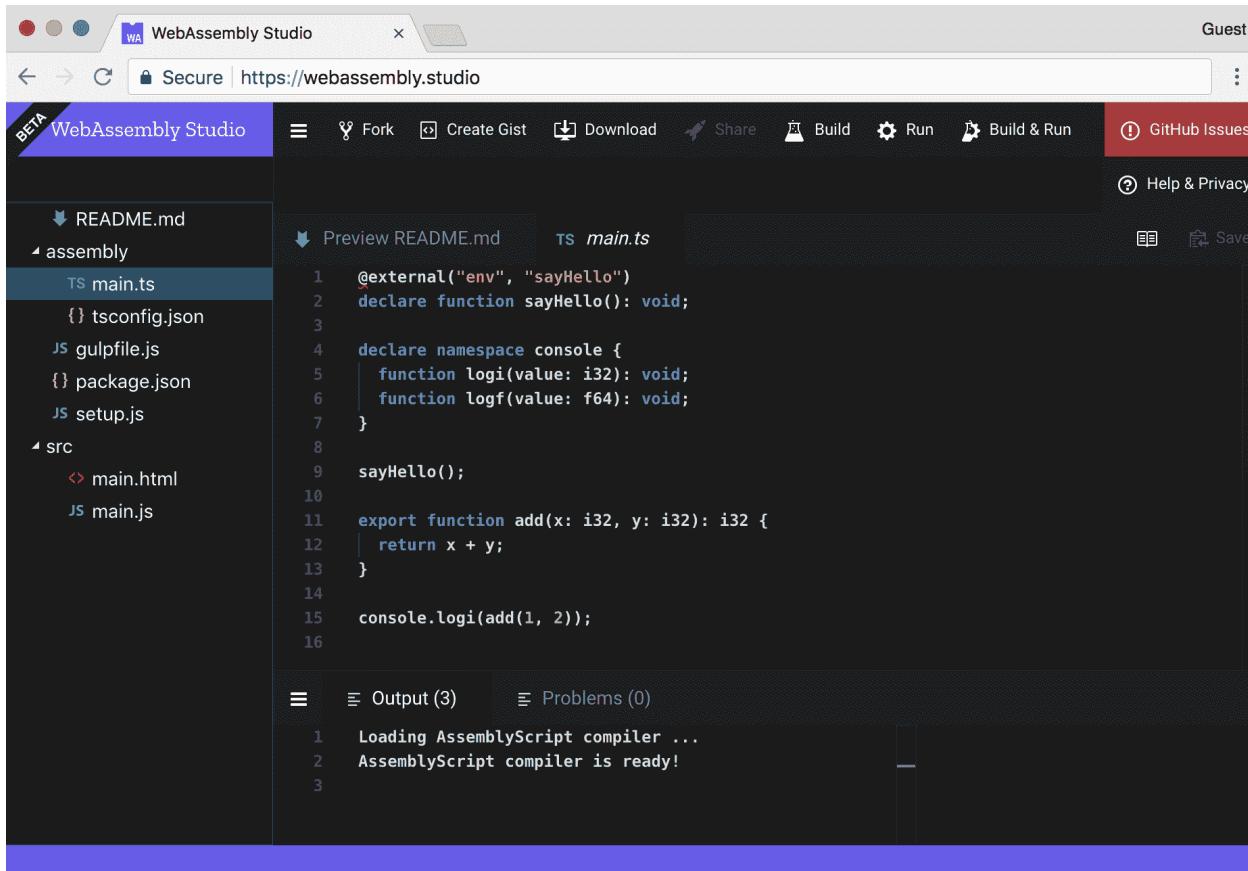
18 Compiling .wat to .wasm
19 Compiling C/C++ to Wat
20 Compiling .wat to x86
21 Compiling .wat to .wasm
22 Compiling C/C++ to Wat
23 Compiling .wat to x86
24 Compiling .wat to .wasm
25 Compiling C/C++ to Wat
26 Compiling .wat to x86
27 Compiling .wat to .wasm
28

```

WebAssembly Explorer 的屏幕截图

WebAssembly Studio

位于 webassembly.studio 的 WebAssembly Studio 是一个功能丰富的编辑器和开发环境。您可以创建 C、Rust 和 AssemblyScript 项目。它提供了在浏览器中构建和运行代码的能力，并与 GitHub 很好地集成。WebAssembly Studio 使您能够构建 Web 应用程序，而无需在本地安装和配置所需的 WebAssembly 工具：



WebAssembly Studio 的屏幕截图

在下一节中，我们将演示如何使用 Web Workers 为您的 WebAssembly 应用程序添加并行性。

使用 Web Workers 实现并行 Wasm

构建执行大量计算或其他资源密集型工作的复杂应用程序的过程可以从使用线程中受益。线程允许您通过将功能分配给独立运行的任务来并行执行操作。在撰写本文时，WebAssembly 中的线程支持处于功能提案阶段。在此阶段，规范尚未编写，功能尚未实现。幸运的是，JavaScript 提供了 Web Workers 形式的线程功能。在本节中，我们将演示如何使用 JavaScript 的 Web Workers API 与单独的线程中的 Wasm 模块进行交互。

Web Workers 和 WebAssembly

Web Workers 允许您在浏览器中利用线程，可以通过将一些逻辑从主（UI）线程中卸载来提高应用程序的性能。工作线程还能够使用 XMLHttpRequest 执行 I/O 操作。工作线程通

过将消息发布到事件处理程序与主线程通信。

Web Workers 允许我们将 Wasm 模块加载到单独的线程中，并执行不会影响 UI 性能的操作。Web Workers 确实有一些限制。它们无法直接操作 DOM 或访问 `window` 对象上的某些方法和属性。线程之间传递的消息必须是序列化对象，这意味着你不能传递函数。现在你知道了 worker 是什么，让我们讨论如何创建一个。

创建一个工作线程

在创建工作线程之前，您需要一个包含在工作线程中运行的代码的 JavaScript 文件。您可以在 github.com/mdn/simple-web-worker/blob/gh-pages/worker.js 上看到一个简单的工作定义文件的示例。该文件应包含一个 `message` 事件监听器，当从其他线程接收到消息时执行操作并做出相应响应。

创建了该文件后，您就可以使用它来创建工作线程。通过将 URL 参数传递给 `Worker()` 构造函数来创建工作线程。URL 可以是表示包含工作定义代码的文件名称的字符串，也可以使用 `Blob` 构造。如果您从服务器获取工作定义代码，则 `Blob` 技术可能很有用。示例应用程序演示了如何同时使用这两种方法。让我们继续介绍如何将 WebAssembly 与 Web Workers 集成的过程。

WebAssembly 工作流程

为了在单独的线程中利用 Wasm 模块，必须在主线程中编译 Wasm 文件，并在 Web Worker 中实例化。让我们更详细地审查这个过程：

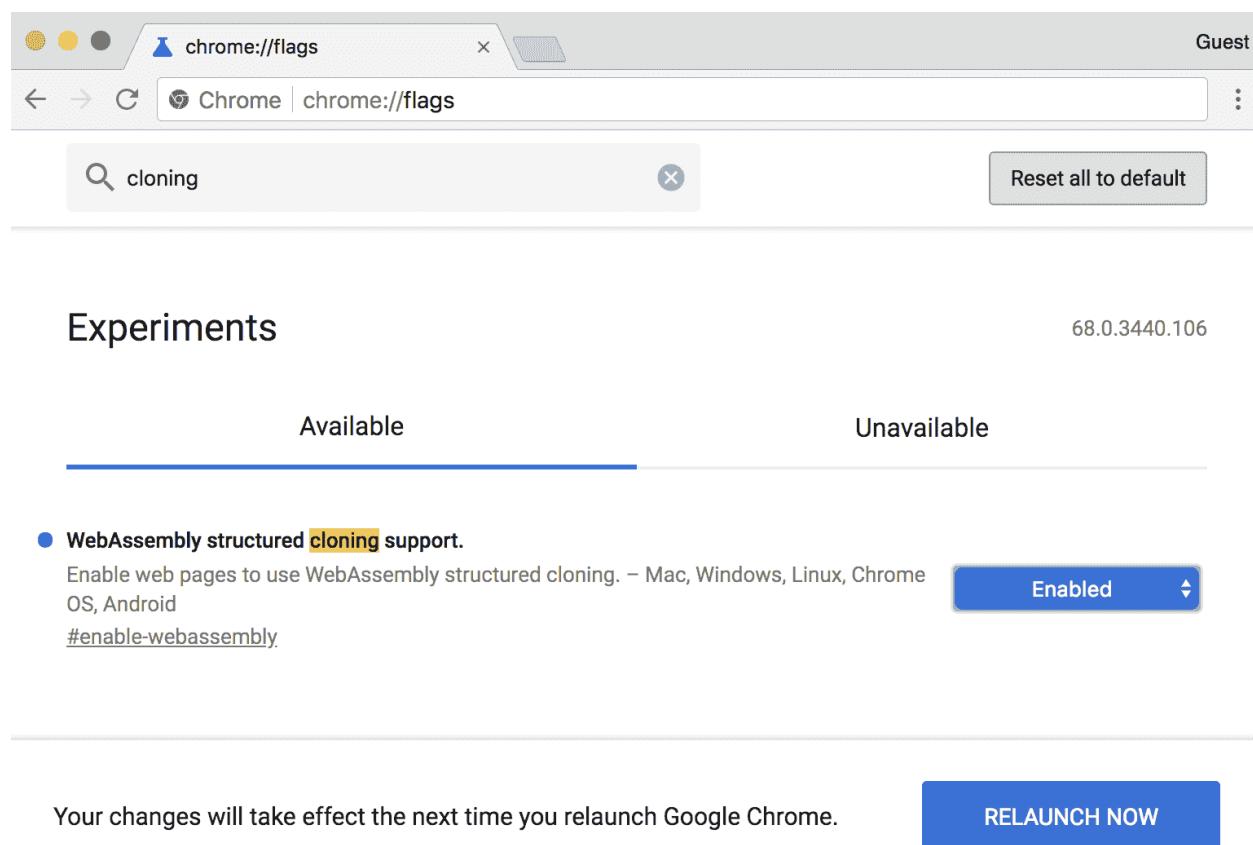
1. 使用 `Worker()` 构造函数创建了一个新的 Web Worker（我们将其称为 `wasmWorker`）。
2. 发出 `fetch` 调用以检索 `.wasm` 文件，并在响应上调用 `arrayBuffer()` 函数。
3. `arrayBuffer()` 函数的解析值传递给 `WebAssembly.compile()` 函数。
4. `WebAssembly.compile()` 函数解析为 `WebAssembly.Module` 实例，并使用 `postMessage()` 函数将其包含在发送到 `wasmWorker` 的消息体中。
5. 在 `wasmWorker` 中，从消息体中传递给 `WebAssembly.instantiate()` 函数的 `WebAssembly.Module` 实例，将解析为一个 `WebAssembly.Instance`。
6. `WebAssembly.Instance` 导出对象被分配给 `wasmWorker` 中的一个本地变量，并用于调用 Wasm 函数。

要从 `wasmWorker` 的 Wasm 实例中调用函数，你需要向工作线程发送一个消息，其中包含要传递给 Wasm 函数的任何参数。然后，`wasmWorker` 执行该函数，并将结果传递回主线程。这就是在 Web Workers 的上下文中如何利用线程的关键。在我们继续进行示例应用

程序之前，你可能需要解决谷歌 Chrome 施加的一个限制。请按照“谷歌 Chrome 中的限制”部分中的说明，确保示例应用程序能够成功运行。

谷歌 Chrome 中的限制

谷歌 Chrome 对 Web Worker 的 `postMessage()` 函数的主体中可以包含什么施加了限制。如果你尝试将编译后的 `WebAssembly.Module` 发送到工作线程，你将收到错误消息，并且操作将不成功。你可以通过设置一个标志来覆盖这一限制。要启用此功能，请打开谷歌 Chrome，并在地址栏中输入 `chrome://flags`。在页面顶部的搜索框中输入 `cloning`。你应该看到一个名为 WebAssembly structured cloning support 的列表项。从下拉菜单中选择 Enabled 选项，并在提示时点击 RELAUNCH NOW 按钮：



在谷歌 Chrome 中更新 WebAssembly 标志

Chrome 重新启动后，你可以无问题地运行示例应用程序。如果你使用 Mozilla Firefox，则无需任何操作。它默认支持此功能。让我们继续进行示例应用程序，演示在线程中使用 WebAssembly 的用法。

代码概述

示例应用程序并不是一个真正的应用程序。它是一个简单的表单，接受两个输入值，并返回这两个值的和或差。加法和减法操作分别从它们自己的 Wasm 模块中导出，并在工作线程中实例化。这个示例可能是刻意构造的，但它有效地演示了如何将 WebAssembly 集成到 Web Workers 中。

这一部分的代码位于 `learn-webassembly` 存储库的 `/chapter-10-advanced-tools/parallel-wasm` 目录中。接下来的部分将逐步介绍代码库的每个部分，并描述如何从头开始构建应用程序。如果你想跟着做，可以在你的 `/book-examples` 目录中创建一个名为 `/parallel-wasm` 的文件夹。

C 代码

该示例使用了两个工作线程：一个用于加法，另一个用于减法。因此，我们需要两个单独的 Wasm 模块。在你的 `/parallel-wasm` 目录中创建一个名为 `/lib` 的文件夹。在 `/lib` 目录中，创建一个名为 `add.c` 的文件，并填充以下内容：

```
int calculate(int firstVal, int secondVal) {
    return firstVal + secondVal;
}
```

在 `/lib` 中创建另一个名为 `subtract.c` 的文件，并填充以下内容：

```
int calculate(int firstVal, int secondVal) {
    return firstVal - secondVal;
}
```

请注意，两个文件中的函数名都是 `calculate`。这样做是为了避免在工作代码中编写任何条件逻辑来确定要调用的 Wasm 函数。代数运算与工作线程相关联，因此当我们需要加两个数字时，将在 `addWorker` 中调用 `_calculate()` 函数。当我们查看代码的 JavaScript 部分时，这将变得更清晰。

JavaScript 代码

在我们深入研究 JavaScript 代码之前，在你的 `/parallel-wasm` 目录中创建一个名为 `/src` 的文件夹。让我们从包含在工作线程中运行的代码的文件开始。

在 worker.js 中定义线程执行

在 `/src` 目录中创建一个名为 `worker.js` 的新文件，并填充以下内容：

```
var wasmInstance = null;

self.addEventListener('message', event => {
  /**
   * Once the WebAssembly compilation is complete, this posts
   * a message
   * back with whether or not the instantiation was successful.
   * If the
   * payload is null, the compilation succeeded.
  */

  const sendCompilationMessage = (error = null) => {
    self.postMessage({
      type: 'COMPILE_WASM_RESPONSE',
      payload: error
    });
  };

  const { type, payload } = event.data;
  switch (type) {
    // Instantiates the compiled Wasm module and posts a message back to
    // the main thread indicating if the instantiation was successful:
    case 'COMPILE_WASM_REQUEST':
      const importObj = {
        env: {
          memoryBase: 0,
          tableBase: 0,
          memory: new WebAssembly.Memory({ initial: 256 }),
          table: new WebAssembly.Table({ initial: 2, element: 'anyfunc' }),
          abort: console.log
        }
      };
      // Create a new WebAssembly instance
      wasmInstance = new WebAssembly.Instance(importObj);
      // Post a message back to the main thread
      sendCompilationMessage(null);
  }
});
```

```

        }

    };

    WebAssembly.instantiate(payload, importObj)
        .then(instance => {
            wasmInstance = instance.exports;
            sendCompilationMessage();
        })
        .catch(error => {
            sendCompilationMessage(error);
        });
    break;
}

// Calls the `calculate` method associated with the instance (add or
// subtract, and posts the result back to the main thread:
case 'CALC_REQUEST':
    const { firstVal, secondVal } = payload;
    const result = wasmInstance._calculate(firstVal, secondVal);

    self.postMessage({
        type: 'CALC_RESPONSE',
        payload: result
    });
    break;

default:
    break;
}
}, false);

```

代码封装在 `message` 事件的事件监听器中（`self.addEventListener(...)`），当对应的 worker 上调用 `postMessage()` 函数时会触发该事件。事件监听器的回调函数中的 `event` 参数包含一个 `data` 属性，其中包含消息的内容。应用程序中线程之间传递的所有消息都遵

循Flux Standard Action (FSA) 约定。遵循此约定的对象具有 `type` 和 `payload` 属性，其中 `type` 是一个字符串，`payload` 可以是任何类型。您可以在 github.com/redux-utilities/flux-standard-action 上了解更多关于 FSA 的信息。

您可以使用 `postMessage()` 函数传递的数据的任何格式或结构，只要数据是可序列化的。

`switch` 语句根据消息的 `type` 值执行操作，该值是一个字符串。如果 `type` 是 `'COMPILE_WASM_REQUEST'`，则调用 `WebAssembly.instantiate()` 函数，传入消息的 `payload` 和 `importObj`。结果的 `exports` 对象分配给本地的 `wasmInstance` 变量以供以后使用。如果 `type` 是 `'CALC_REQUEST'`，则使用 `payload` 对象中的 `firstVal` 和 `secondVal` 值调用 `wasmInstance._calculate()` 函数。计算代码应该解释为什么函数被命名为 `_calculate()` 而不是 `_add()` 或 `_subtract()`。通过使用一个通用名称，工作线程不关心它执行的是什么操作，它只是调用函数以获得结果。

在这两种情况下，工作线程都使用 `postMessage()` 函数向主线程发送消息。我使用了 `REQUEST / RESPONSE` 约定来表示 `type` 属性的值。这使您可以快速识别消息的来源线程。从主线程发送的消息以 `_REQUEST` 结尾，而来自工作线程的响应以 `_RESPONSE` 结尾。让我们继续进行 WebAssembly 交互代码。

在 WasmWorker.js 中与 Wasm 交互

在 `/src` 目录中创建一个名为 `WasmWorker.js` 的新文件，并填充以下内容：

```
/***
 * Web Worker associated with an instantiated Wasm module.
 * @class
 */
export default class WasmWorker {
  constructor(workerUrl) {
    this.worker = new Worker(workerUrl);
    this.listenersByType = {};
    this.addListeners();
  }

  // Add a listener associated with the `type` value from the
  // Worker message:
  addListenerForType(type, listener) {
    this.listenersByType[type] = listener;
  }
}
```

```
}

// Add event listeners for error and message handling.
addListeners() {
    this.worker.addEventListener('error', event => {
        console.error(`%cError: ${event.message}`, 'color: red;');
    }, false);

    // If a handler was specified using the `addListener` method,
    // fire that method if the `type` matches:
    this.worker.addEventListener('message', event => {
        if (
            event.data instanceof Object &&
            event.data.hasOwnProperty('type') &&
            event.data.hasOwnProperty('payload')
        ) {
            const { type, payload } = event.data;
            if (this.listenersByType[type]) {
                this.listenersByType[type];
            }
        } else {
            console.log(event.data);
        }
    }, false);
}

// Fetches the Wasm file, compiles it, and passes the compiled result
// to the corresponding worker. The compiled module is instantiated
// in the worker.
initialize(name) {
    return fetch(`calc-${name}.wasm`)
        .then(response => response.arrayBuffer())
```

```

        .then(bytes => WebAssembly.compile(bytes))
        .then(wasmModule => {
            this.worker.postMessage({
                type: 'COMPILE_WASM_REQUEST',
                payload: wasmModule
            });
            return Promise.resolve();
        });
    }

    // Posts a message to the worker thread to call the `calculate` method from the Wasm instance:
    calculate(firstVal, secondVal) {
        this.worker.postMessage({
            type: 'CALC_REQUEST',
            payload: {
                firstVal,
                secondVal
            }
        });
    }
}

```

`WasmWorker` 类管理与 Wasm 文件关联的工作线程。在 `WasmWorker` 构造函数中，创建一个新的 `Worker`，并为 `error` 和 `message` 事件添加了默认的事件监听器。`initialize()` 函数获取与 `name` 参数关联的 `.wasm` 文件，编译它，并将结果的 `WebAssembly.Module` 实例发送到工作线程以进行实例化。

`addListenerForType()` 函数用于指定一个 `callback` 函数 (`listener`)，当消息响应中的 `type` 字段与传递给函数的 `type` 参数匹配时执行。这是为了捕获来自工作线程的 `_calculate()` 函数的结果。

最后，在 `WasmWorker` 中的 `calculate()` 函数向工作线程发送一条消息，消息中包括从 `<form>` 中的 `<input>` 元素传入的 `firstVal` 和 `secondVal` 参数。让我们继续看应用加载代码，以了解 `WasmWorker` 如何与 UI 交互。

在 index.js 中加载应用程序

在 `/src` 目录中创建一个名为 `index.js` 的新文件，并填充以下内容：

```
import WasmWorker from './WasmWorker.js';

/**
 * If you add ?blob=true to the end of the URL (e.g.
 * http://localhost:8080/index.html?blob=true), the worker will be
 * created from a Blob rather than a URL. This returns the
 * URL to use for the Worker either as a string or created from a Blob.
 */
const getWorkerUrl = async () => {
    const url = new URL(window.location);
    const isBlob = url.searchParams.get('blob');
    var workerUrl = 'worker.js';
    document.title = 'Wasm Worker (String URL)';

    // Create a Blob instance from the text contents of `worker.js`:
    if (isBlob === 'true') {
        const response = await fetch('worker.js');
        const results = await response.text();
        const workerBlob = new Blob([results]);
        workerUrl = window.URL.createObjectURL(workerBlob);
        document.title = 'Wasm Worker (Blob URL)';
    }

    return Promise.resolve(workerUrl);
};

/**
 * Instantiates the Wasm module associated with the specified worker

```

```
* and adds event listeners to the "Add" and "Subtract" buttons.  
*/  
const initializeWorker = async (wasmWorker, name) => {  
    await wasmWorker.initialize(name);  
    wasmWorker.addListenerForType('CALC_RESPONSE', payload => {  
        document.querySelector('#result').value = payload;  
    });  
  
    document.querySelector(`#${name}`).addEventListener('click', () => {  
        const inputs = document.querySelectorAll('input');  
        var [firstInput, secondInput] = inputs.values();  
        wasmWorker.calculate(+firstInput.value, +secondInput.value);  
    });  
};  
  
/**  
 * Spawns (2) workers: one associated with calc-add.wasm and another  
 * with calc-subtract.wasm. Adds an event listener to the "Reset"  
 * button to clear all the input values.  
 */  
const loadPage = async () => {  
    document.querySelector('#reset').addEventListener('click', () => {  
        const inputs = document.querySelectorAll('input');  
        inputs.forEach(input => (input.value = 0));  
    });  
  
    const workerUrl = await getWorkerUrl();  
    const addWorker = new WasmWorker(workerUrl);  
    await initializeWorker(addWorker, 'add');
```

```

    const subtractWorker = new WasmWorker(workerUrl);
    await initializeWorker(subtractWorker, 'subtract');

};

loadPage()
  .then(() => console.log('%cPage loaded!', 'color: green;'))
  .catch(error => console.error(error));

```

应用程序的入口点是 `loadPage()` 函数。在我们深入讨论工作线程初始化代码之前，让我们讨论一下 `getWorkerUrl()` 函数。在本节的前面，我们了解到可以将表示文件名的字符串或从 `Blob` 创建的 URL 传递给 `Worker()` 构造函数。以下示例代码演示了第一种技术：

```
var worker = new Worker('worker.js');
```

第二种技术在 `getWorkerUrl()` 函数的 `if (isBlob === 'true')` 块中进行演示。如果当前的 `window.location` 值以 `?blob=true` 结尾，那么传递给 `Worker()` 构造函数的 URL 将从 `Blob` 创建。唯一显着的区别是 `document.title` 的值，它会更新以反映 URL 类型。让我们回到 `loadPage()` 函数，讨论初始化代码。

在 `loadPage()` 函数中为重置按钮添加事件侦听器后，创建了两个 `WasmWorker` 实例：
`addWorker` 和 `subtractWorker`。每个 `worker` 都作为 `wasmWorker` 参数传递给 `initializeWorker()` 函数。在 `initializeWorker()` 中，调用 `wasmWorker.initialize()` 函数来实例化 Wasm 模块。调用 `wasmWorker.addListenerForType()` 函数来将 `Result <input>` 的值设置为对应 `worker` 中 `_calculate()` 函数返回的值。最后，为 `<button>` 的 `click` 事件添加了一个事件侦听器，该事件要么将 `firstVal` 和 `secondVal` `<input>` 值相加，要么相减（基于 `name` 参数）。这就是 JavaScript 代码的全部内容。让我们创建一个 HTML 和 CSS 文件，然后继续进行构建步骤。

Web 资产

我们需要一个 HTML 文件作为应用程序的入口点。在 `/src` 目录中创建一个名为 `index.html` 的文件，并填充以下内容：

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">

```

```

<title>Wasm Workers</title>
<link rel="stylesheet" type="text/css" href="styles.css" />
</head>
<body>
<form class="valueForm">
  <div class="valueForm">
    <label for="firstVal">First Value:</label>
    <input id="firstVal" type="number" value="0" />
  </div>
  <div class="valueForm">
    <label for="secondVal">Second Value:</label>
    <input id="secondVal" type="number" value="0" />
  </div>
  <div class="valueForm">
    <label for="result">Result:</label>
    <input id="result" type="number" value="0" readonly />
  </div>
</form>
<div>
  <button id="add">Add</button>
  <button id="subtract">Subtract</button>
  <button id="reset">Reset</button>
</div>
<script type="module" src="img/index.js"></script>
</body>
</html>

```

该应用程序由一个带有三个 `<input>` 元素和一个包含三个 `<button>` 元素的块组成。前两个 `<input>` 元素对应于 `payload` 中包含的 `firstVal` 和 `secondVal` 属性，这些属性发送到任一工作线程。最后一个 `<input>` 是只读的，并显示任一操作的结果。

在 `<form>` 下面的一组 `<button>` 元素对 `<input>` 值执行操作。前两个 `<button>` 元素将 `<input>` 值发送到 `addWorker` 或 `subtractWorker` 线程（取决于按下哪个按钮）。最后一个 `<button>` 将所有 `<input>` 值设置为 `0`。

应用程序在 `</body>` 结束标记之前的最后一行的 `<script>` 标记中初始化。与 Cook the Books 一样，`type="module"` 属性允许我们在较新的浏览器中使用 `import` / `export` 语法。最

后，我们需要为应用程序添加一些样式。在 `/src` 目录中创建一个名为 `styles.css` 的文件，并填充以下内容：

```
* {
    font-family: sans-serif;
    font-size: 14px;
}

body {
    margin: 16px;
}

form.valueForm {
    display: table;
}

div.valueForm {
    display: table-row;
}

label, input {
    display: table-cell;
    margin-bottom: 16px;
}

label {
    font-weight: bold;
    padding-right: 16px;
}

button {
    border: 1px solid black;
    border-radius: 4px;
    cursor: pointer;
    font-weight: bold;
    height: 24px;
```

```
margin-right: 4px;  
width: 80px;  
}  
  
button:hover {  
    background: lightgray;  
}
```

这是我们需要创建的最后一个文件，但不是运行应用程序所需的最后一个文件。我们仍然需要从 `/lib` 目录中的 C 文件生成 Wasm 文件。让我们继续进行构建步骤。

构建和运行应用程序

代码编写完成后，现在是时候构建和测试应用程序了。完成构建步骤后，我们将与正在运行的应用程序进行交互，并回顾如何使用浏览器的开发工具来排除 Web Workers 的故障。

编译 C 文件

我们需要将每个 C 文件编译为单独的 `.wasm` 文件，这意味着执行编译步骤所需的命令是冗长的。要执行构建，请在 `/parallel-wasm` 目录中打开一个终端实例，并运行以下命令：

```
# First, compile the add.c file: emcc -Os -s WASM=1 -s SIDE_MODULE=1 -s BINARYEN_ASYNC_COMPILATION=0 lib/add.c -o src/calc-add.wasm # Next, compile the subtract.c fileemcc -Os -s WASM=1 -s SIDE_MODULE=1 -s BINARYEN_ASYNC_COMPILATION=0 lib/subtract.c -o src/calc-subtract.wasm
```

您应该在 `/src` 目录中看到两个新文件：`calc-add.wasm` 和 `calc-subtract.wasm`。有了所需的文件，现在是时候测试应用程序了。

与应用程序交互

在 `/parallel-wasm` 目录中打开一个终端实例，并运行以下命令：

```
serve -l 8080 src
```

如果在浏览器中导航到 `http://127.0.0.1:8080/index.html`，您应该会看到这个：

First Value:

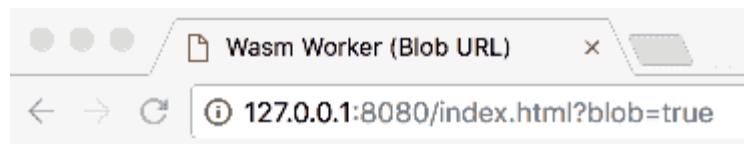
Second Value:

Result:

Add Subtract Reset

在浏览器中运行的 Wasm Workers 应用程序

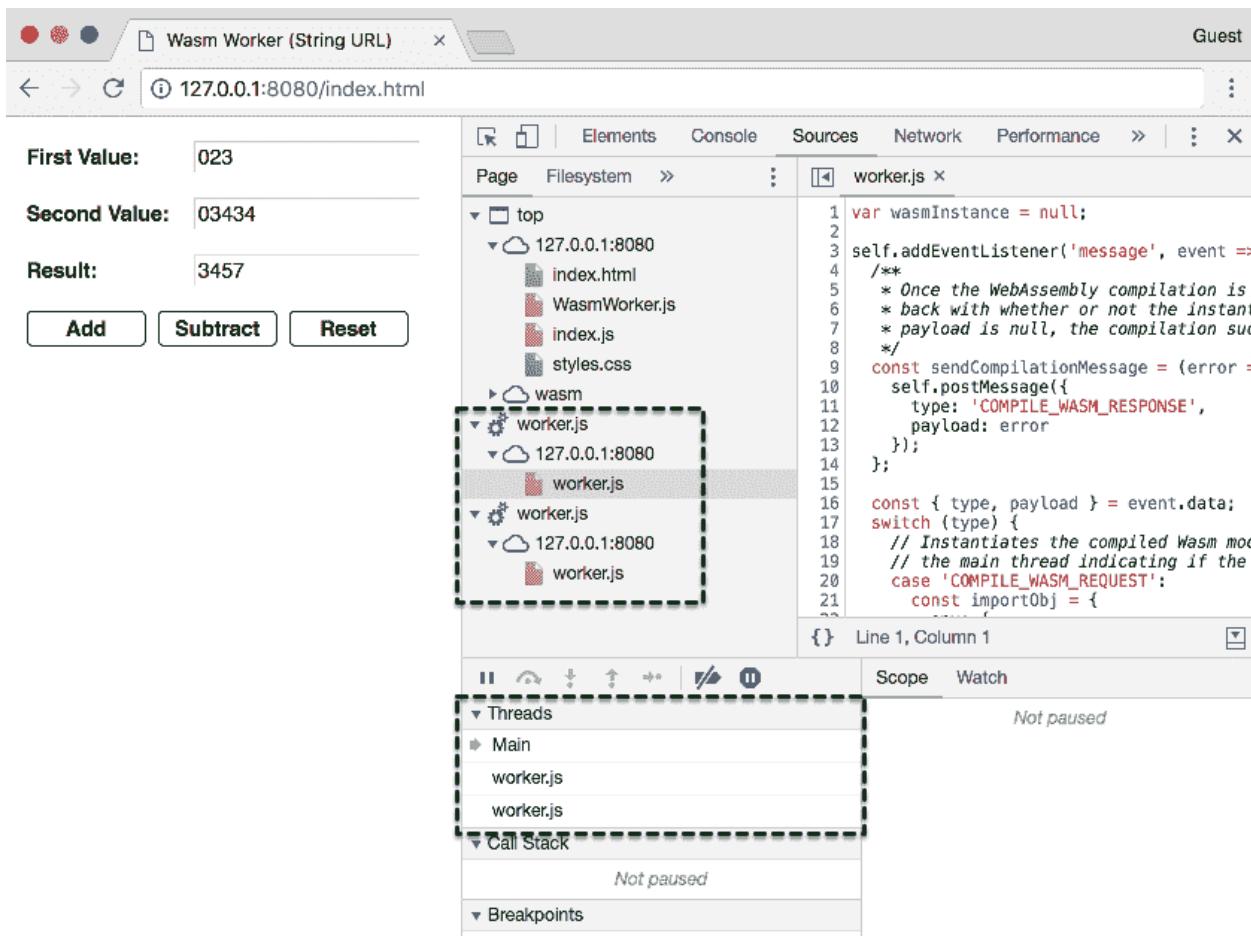
尝试更改第一个值和第二个值输入中的值，然后按“添加”和“减去”按钮。结果输入应该更新为计算结果。如果您导航到 `http://127.0.0.1:8080/index.html?blob=true`，传递给 `Worker()` 构造函数的 URL 参数将使用 `Blob` 而不是文件名。选项卡应更改以反映使用 `Blob` 技术来构造 URL：



选项卡标题已更新以反映 Blob URL 技术

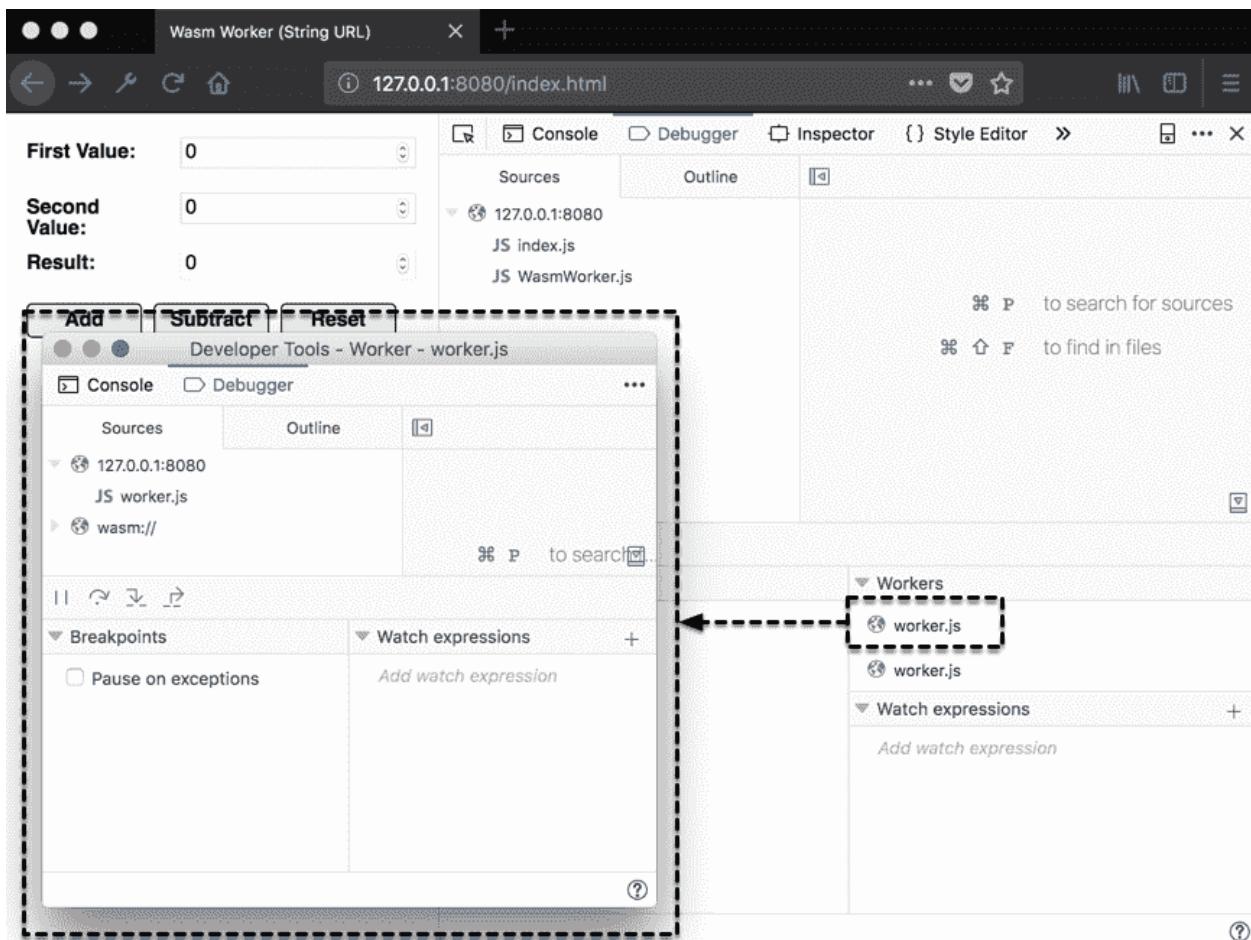
调试 Web Workers

您可以使用浏览器的开发工具设置断点并与工作线程进行交互。在 Google Chrome 中，打开开发者工具并选择“源”选项卡。文件列表面板应包含两个 `worker.js` 实例。调试器面板包含一个“线程”部分，其中包含“主”线程和两个 `worker.js` 线程。以下屏幕截图显示了 Chrome 开发者工具面板中用于运行应用程序的线程调试元素：



Chrome 开发者工具面板中的线程调试工具

在 Firefox 中，worker 调试是在单独的开发者工具窗口中完成的。要查看此操作，请在 Firefox 中打开开发者工具，并选择调试器面板。单击 Workers 面板中的 `worker.js` 列表项之一。应该会出现一个新的开发者工具窗口，对应于所选的 worker。以下屏幕截图显示了从 Workers 面板中选择的 `worker.js` 实例的单独开发者工具窗口：



Firefox 开发者工具面板中的线程调试工具

在下一节中，我们将讨论一些即将推出的 WebAssembly 功能。

即将推出的功能

有几个即将推出的 WebAssembly 功能处于标准化过程的各个阶段。其中一些比其他功能更具影响力，但它们都是有价值的改进。在本节中，我们将描述标准化过程，并审查代表 WebAssembly 能力显著转变的一些功能的子集。本节大部分内容引用自 Colin Eberhardt 的博客文章，标题为 *WebAssembly 的未来-即将推出的功能和提案*。该文章可以在 blog.scottlogic.com/2018/07/20/wasm-future.html 找到。

标准化过程

WebAssembly W3C 流程文档位于

github.com/WebAssembly/meetings/blob/master/process/phases.md，描述了标准化过程的六个阶段（从 0 到 5）。以下列表提供了对每个阶段的简要描述：

- **第 0 阶段。预提案**：WebAssembly 社区组（CG）成员有一个想法，并且 CG 投票决定是否将其移至第 1 阶段。
- **第 1 阶段。功能提案**：预提案过程已成功，并在 GitHub 的 WebAssembly 组织中创建了一个存储库以记录该功能。
- **第 2 阶段。提议的规范文本可用**：完整的提议规范文本可用，可能的实现已经原型化，并添加了测试套件。
- **第 3 阶段。实施阶段**：嵌入器实施该功能，存储库已更新以包括对规范的修订，并且规范已更新以包括对参考解释器中功能的实施。
- **第 4 阶段。标准化功能**：如果两个或两个以上的 Web VM 和至少一个工具链实现了该功能，则该功能将完全移交给 WebAssembly 工作组（WG）。
- **第 5 阶段。功能已标准化**：WG 成员已达成共识，该功能已完成。

现在您已经熟悉了与标准化过程相关的阶段，让我们继续讨论线程提案。

线程

在前一节中，我们使用 Web Workers 将 Wasm 模块移动到工作线程中，这样我们就可以调用 Wasm 函数而不会阻塞主线程。然而，在工作线程之间传递消息存在性能限制。为了解决这个问题，提出了一个 WebAssembly 的线程功能。

该提案目前处于第一阶段，详细描述在

github.com/WebAssembly/threads/blob/master/proposals/threads/Overview.md。根据提案文件，线程功能添加了一个新的共享线性内存类型和一些新的原子内存访问操作。这个提案在范围上相对有限。Eberhardt 在他的博客文章中提供了以下阐述：

“值得注意的是，这个提案并没有引入创建线程的机制（这引起了很多争论），而是由主机提供这个功能。在浏览器中执行的 wasm 的上下文中，这将是熟悉的 WebWorkers。”

虽然这个功能不允许创建线程，但它提供了一种更简单的方法来在 JavaScript 中创建的工作线程之间共享数据。

主机绑定

主机绑定提案，也处于第一阶段，将解决 WebAssembly 在浏览器中使用时的一个重要限制：DOM 操作。该提案文件在 github.com/WebAssembly/host-bindings/blob/master/proposals/host-bindings/Overview.md 中提供了该功能的以下目标列表：

- **人机工程学**：允许 WebAssembly 模块创建、传递、调用和操作 JavaScript + DOM 对象
- **速度**：允许 JS/DOM 或其他主机调用进行优化
- **平台一致性**：允许使用 WebIDL 来注释 Wasm 的导入/导出（通过工具）
- **渐进式**：提供一种可填充的策略

改进 WebAssembly 与 JavaScript 和 Web API 的互操作性将大大简化开发过程。它还将消除诸如 Emscripten 等工具提供的“胶水”代码的需求。

垃圾回收

垃圾回收 (GC) 提案目前处于第一阶段。我们在第一章的 *WebAssembly 是什么？* 部分讨论了垃圾回收。提案文件在

github.com/WebAssembly/gc/blob/master/proposals/gc/Overview.md 中提供了该功能的广泛概述，并描述了需要添加到规范中的元素。Eberhardt 在他的博客文章中对提案进行了以下描述：

“这个提案为 WebAssembly 添加了 GC 功能。有趣的是，它不会有自己 GC，而是将与主机环境提供的 GC 集成。这是有道理的，因为这个提案和其他各种提案（主机绑定，引用类型）旨在改进与主机的互操作性，使共享状态和调用 API 变得更容易。拥有一个单一的 GC 来管理内存会使这一切变得更容易。”

这个功能将需要大量的工作来实现，但将其添加到 WebAssembly 中将是值得的。让我们用一个当前正在实施阶段的功能来结束这一节：引用类型。

引用类型

引用类型，目前处于第三阶段，构成了主机绑定和 GC 功能的基础。提案文档位于

github.com/WebAssembly/reference-types/blob/master/proposals/reference-types/Overview.md，描述了添加一个新类型 `anyref`，它可以作为值类型和表元素类型使用。`anyref` 类型允许您将 JavaScript 对象传递给 Wasm 模块。Eberhardt 在他的博客文章中描述了这一功能的影响：

“通过 `anyref` 类型，wasm 模块实际上无法对对象执行太多操作。更重要的是，模块持有对 JS 堆中的垃圾收集对象的引用，这意味着它们在 wasm 执行期间需要被跟踪。这个提案被视为通往更重要的垃圾收集提案的一个垫脚石。”

WebAssembly 还有其他一些令人兴奋的功能正在开发中。WebAssembly CG 和 WG 正在投入他们的时间和资源，使这些功能成为现实。您可以在 GitHub 上的 WebAssembly

组织页面上查看所有提案，网址为 github.com/WebAssembly。

摘要

在这一章中，我们回顾了 WebAssembly 的高级工具和另一种编译方法。我们了解了 WABT 和 Binaryen 在 WebAssembly 开发过程中的作用以及它们提供的功能。我们通过使用 WebAssembly 的 `npm` 包，使用 LLVM 编译了一个 Wasm 模块，并在浏览器中与结果进行了交互。我们回顾了一些在线可用的 WebAssembly 工具，并创建了一个简单的应用程序，使用 Web Workers 将 Wasm 模块存储在单独的线程中。最后，我们讨论了 WebAssembly 的即将推出的功能和标准化过程。现在你对 WebAssembly 有了更深入的了解，可以开始构建一些东西了！

问题

- WABT 代表什么？
- Binaryen 提供了哪三个元素，使编译到 WebAssembly 变得简单、快速和有效？
- 使用 Emscripten 和 LLVM 编译的模块在 `importObj` / `exports` 方面的主要区别是什么？
- 哪个在线工具允许您使用 AssemblyScript？
- 您可以传递给 `Worker()` 构造函数的两种参数类型是什么？
- 主线程和工作线程之间传递消息使用了什么约定？
- WebAssembly 标准化过程中有多少个阶段？
- 在引用类型功能中定义的新类型的名称是什么？

进一步阅读

- 内存管理速成课：hacks.mozilla.org/2017/06/a-crash-course-in-memory-management
- MDN Web Workers API：developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API
- WebAssembly - Web Workers：medium.com/@c.gerard.gallant/webassembly-web-workers-f2ba637c3e4a