异步图书
www.epubit.com

一切可编译为 WebAssembly 的，
终将被编译为 WebAssembly。

WebAssembly 标准入门

柴树杉 丁尔男 著

www.epubit.com

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

WebAssembly标准入门

柴树杉 丁尔男 著

异步社区 www.epubit.com

人民邮电出版社
北京

图书在版编目（CIP）数据

WebAssembly标准入门 / 柴树杉，丁尔男著. -- 北京 : 人民邮电出版社，2019.1
ISBN 978-7-115-50059-5

Ⅰ. ①W… Ⅱ. ①柴… ②丁… Ⅲ. ①编译软件 Ⅳ. ①TP314

中国版本图书馆CIP数据核字(2018)第251367号

内 容 提 要

WebAssembly 是一种新兴的网页虚拟机标准，它的设计目标包括高可移植性、高安全性、高效率（包括载入效率和运行效率）、尽可能小的程序体积。本书详尽介绍了 WebAssembly 程序在 JavaScript 环境下的使用方法、WebAssembly 汇编语言和二进制格式，给出了大量简单易懂的示例，同时以 C/C++ 和 Go 语言开发环境为例，介绍了如何使用其他高级语言开发 WebAssembly 模块。

本书适合从事高性能 Web 前端开发、跨语言 Web 应用开发的技术人员学习参考，也可以作为 WebAssembly 标准参考手册随时查阅。

异步社区 www.epubit.com

◆ 著　　　柴树杉　丁尔男
责任编辑　杨海玲
责任印制　焦志炜
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市君旺印务有限公司印刷
◆ 开本：800×1000　1/16
印张：13.25
字数：246 千字　　　　2019 年 1 月第 1 版
印数：1 – 3 000 册　　　2019 年 1 月河北第 1 次印刷

定价：49.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

序

某一天，有朋友向我推荐了一项新技术——WebAssembly。我认为这是一项值得关注的技术。

说 WebAssembly 是一门编程语言，但它更像一个编译器。实际上它是一个虚拟机，包含了一门低级汇编语言和对应的虚拟机体系结构，而 WebAssembly 这个名字从字面理解就说明了一切——Web 的汇编语言。它的优点是文件小、加载快、执行效率非常高，可以实现更复杂的逻辑。

其实，我觉得出现这样的技术并不令人意外，而只是顺应了潮流，App 的封闭系统必然会被新一代 Web OS 取代。但现有的 Web 开发技术，如 JavaScript，前端执行效率和解决各种复杂问题的能力还不足，而 WebAssembly 的编译执行功能恰恰能弥补这些不足。WebAssembly 标准是在谋智（Mozilla）、谷歌（Google）、微软（Microsoft）、苹果（Apple）等各大厂商的大力推进下诞生的，目前包括 Chrome、Firefox、Safari、Opera、Edge 在内的大部分主流浏览器均已支持 WebAssembly。这使得 WebAssembly 前景非常好。

WebAssembly 是 Web 前端技术，具有很强的可移植性，技术的潜在受益者不局限于传统的前端开发人员，随着技术的推进，越来越多的其他语言的开发者也将从中受益。如果开发者愿意，他们可以使用 C/C++、Go、Rust、Kotlin、C#等开发语言来写代码，然后编译为 WebAssembly，并在 Web 上执行，这是不是很酷？它能让我们很容易将用其他编程语言编写的程序移植到 Web 上，对于企业级应用和工业级应用都是巨大利好。

WebAssembly 的应用场景也相当丰富，如 Google Earth，2017 年 10 月 Google Earth 开始在 Firefox 上运行，其中的关键就是使用了 WebAssembly；再如网页游戏，WebAssembly 能让 HTML5 游戏引擎速度大幅提高，国内一家公司使用 WebAssembly 后引擎效率提高了 300%。

WebAssembly 作为一种新兴的技术，为开发者提供了一种崭新的思路和工作方式，未来是很有可能大放光彩的，不过目前其相关的资料和社区还不够丰富，尽管已经有一些社区开始出现了相关技术文章，CSDN 上也有较多的文章，但像本书这样全面系统地介绍 WebAssembly 技术的还不多，甚至没有。本书的两位作者都是有 10 多年经验的一线开发者，他们从 WebAssembly 概念诞生之初就开始密切关注该技术的发展，其中柴树

杉是 Emscripten（WebAssembly 的技术前身之一）的首批实践者，丁尔男是国内首批工程化使用 WebAssembly 的开发者。

2018 年 7 月，WebAssembly 社区工作组发布了 WebAssembly 1.0 标准。现在，我在第一时间就向国内开发者介绍和推荐本书，是希望开发者能迅速地了解和学习新技术，探索新技术的价值。

蒋涛
CSDN 创始人、总裁，极客帮创始合伙人

异步社区 www.epubit.com

本书结构

第 0 章回顾了 WebAssembly 的诞生背景。

第 1 章针对不熟悉 JavaScript 的读者，介绍本书将使用到的部分 JavaScript 基础知识，包括 console 对象、箭头函数、Promise、ArrayBuffer 等。对 JavaScript 很熟悉的读者可以跳过本章。

第 2 章通过两个简单的小例子展示 WebAssembly 的基本用法。

第 3 章和第 4 章分别从外部和内部两个角度详细介绍 WebAssembly，前者着重于相关的 JavaScript 对象，后者着重于 WebAssembly 虚拟机的内部运行机制。因为 WebAssembly 跨越了两种语言、两套运行时结构，所以读者阅读第 3 章时可能会感到不明就里——为什么多数指令中没有操作数？所谓的“栈式虚拟机”到底是什么？类似的疑问都将在第 4 章中得到解答。在写作本书时，我们期望读者读完第 4 章后复读第 3 章时，能有豁然开朗的感觉。

第 5 章介绍 WebAssembly 汇编的二进制格式。若想尝试自己实现 WebAssembly 虚拟机或者其他语言到 WebAssembly 的编译器，掌握二进制汇编格式是必需的。即使不开展类似的项目，通过阅读本章也可以加深对 WebAssembly 虚拟机架构的整体认识，厘清各种全局索引的相互关系。

第 6 章和第 7 章分别以 C/C++和 Go 语言为例，介绍如何使用高级语言来开发 WebAssembly 应用。WebAssembly 全手工编写.wat 文件实现大型模块的机会并不会很多。在实际工程中，WebAssembly 作为一门类汇编语言，更多的是作为其他语言的编译目标而存在。目前 C/C++、Rust、Go、Lua、Kotlin、C#均已支持 WebAssembly，可以预见这一支持列表将越来越长。

附录列出了现有的 200 多条 WebAssembly 指令及其作用。

异步社区 www.epubit.com

致谢

感谢蒋涛先生为本书作序，感谢所有为 WebAssembly 标准诞生作出努力的朋友。其中特别感谢 Emscripten 和 asm.js 的作者，没有他们的灵感，WebAssembly 标准就不可能诞生。感谢 WebAssembly 工作组的专家，是他们的工作让我们看到了草案 1.0 的成果。感谢各大浏览器厂商为 WebAssembly 提供的支持。最后，感谢人民邮电出版社的杨海玲编辑，没有她，本书就不可能出版。谢谢大家！

异步社区
www.epubit.com

资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

配套资源

本书提供书中的源代码，要获得以上配套资源，请在异步社区本书页面中点击 配套资源，跳转到下载界面，按提示进行操作即可。注意：为保证购书读者的权益，该操作会给出相关提示，要求输入提取码进行验证。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，点击“提交勘误”，输入勘误信息，点击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的 100 积分。积分可用于在异步社区兑换优惠券、样书或奖品。

异步社区 www.epubit.com

与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问 www.epubit.com/selfpublish/submission 即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“**异步社区**”是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于 2015 年 8 月，提供大量精品 IT 技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网 https://www.epubit.com。

“**异步图书**”是由异步社区编辑团队策划出版的精品 IT 专业图书的品牌，依托于人民邮电出版社近 30 年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异

异步社区 www.epubit.com

步图书的 LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。

异步社区

微信服务号

异步社区 www.epubit.com

目　录

第 0 章　WebAssembly 诞生背景 ······ 1

0.1　JavaScript 简史 ······ 1
0.2　asm.js 的尝试 ······ 2
0.3　WebAssembly 的救赎 ······ 5

第 1 章　JavaScript 语言基础 ······ 7

1.1　console 对象 ······ 7
1.2　函数和闭包 ······ 9
1.3　Promise 对象 ······ 12
1.4　二进制数组 ······ 13

第 2 章　WebAssembly 快速入门 ······ 17

2.1　准备工作 ······ 17
2.1.1　WebAssembly 兼容性 ······ 17
2.1.2　WebAssembly 文本格式与 wabt 工具集 ······ 19
2.2　首个例程 ······ 21
2.3　WebAssembly 概览 ······ 22
2.3.1　WebAssembly 中的关键概念 ······ 23
2.3.2　WebAssembly 程序生命周期 ······ 24
2.3.3　WebAssembly 虚拟机体系结构 ······ 25
2.4　你好，WebAssembly ······ 25
2.4.1　WebAssembly 部分 ······ 26
2.4.2　JavaScript 部分 ······ 27
2.5　WebAssembly 调试及代码编辑环境 ······ 28

异步社区 www.epubit.com

第 3 章 JavaScript 中的 WebAssembly 对象 …… 31
3.1 WebAssembly 对象简介 …… 31
3.2 全局方法 …… 32
3.2.1 WebAssembly.compile() …… 32
3.2.2 WebAssembly.instantiate() …… 33
3.2.3 WebAssembly.validate() …… 34
3.2.4 WebAssembly.compileStreaming() …… 35
3.2.5 WebAssembly.instantiateStreaming() …… 35
3.3 WebAssembly.Module 对象 …… 36
3.3.1 WebAssembly.Module() …… 36
3.3.2 WebAssembly.Module.exports() …… 37
3.3.3 WebAssembly.Module.imports() …… 38
3.3.4 WebAssembly.Module.customSections() …… 39
3.3.5 缓存 Module …… 40
3.4 WebAssembly.Instance 对象 …… 41
3.4.1 WebAssembly.Instance() …… 41
3.4.2 WebAssembly.Instance.prototype.exports …… 42
3.4.3 创建 WebAssembly.Instance 的简洁方法 …… 43
3.5 WebAssembly.Memory 对象 …… 44
3.5.1 WebAssembly.Memory() …… 44
3.5.2 WebAssembly.Memory.prototype.buffer …… 44
3.5.3 WebAssembly.Memory.prototype.grow() …… 47
3.6 WebAssembly.Table 对象 …… 50
3.6.1 WebAssembly.Table() …… 51
3.6.2 WebAssembly.Table.prototype.get() …… 52
3.6.3 WebAssembly.Table.prototype.length …… 54
3.6.4 在 WebAssembly 内部使用表格 …… 55
3.6.5 多个实例通过共享表格及内存协同工作 …… 57
3.6.6 WebAssembly.Table.prototype.set() …… 60
3.6.7 WebAssembly.Table.prototype.grow() …… 61
3.7 小结及错误类型 …… 61

异步社区 www.epubit.com

第 4 章 WebAssembly 汇编语言 ······ 65
4.1 S-表达式 ······ 65
4.2 数据类型 ······ 66
4.3 函数定义 ······ 67
4.3.1 函数签名 ······ 67
4.3.2 局部变量表 ······ 68
4.3.3 函数体 ······ 68
4.3.4 函数别名 ······ 68
4.4 变量 ······ 69
4.4.1 参数与局部变量 ······ 69
4.4.2 变量别名 ······ 70
4.4.3 全局变量 ······ 70
4.5 栈式虚拟机 ······ 72
4.5.1 栈 ······ 72
4.5.2 WebAssembly 栈式虚拟机 ······ 72
4.5.3 栈式调用 ······ 73
4.6 函数调用 ······ 75
4.6.1 直接调用 ······ 75
4.6.2 间接调用 ······ 76
4.6.3 递归 ······ 78
4.7 内存读写 ······ 79
4.7.1 内存初始化 ······ 79
4.7.2 读取内存 ······ 80
4.7.3 写入内存 ······ 81
4.7.4 获取内存容量及内存扩容 ······ 82
4.8 控制流 ······ 83
4.8.1 nop 和 unreachable ······ 83
4.8.2 block 指令块 ······ 83
4.8.3 if 指令块 ······ 85
4.8.4 loop 指令块 ······ 86
4.8.5 指令块的 label 索引及嵌套 ······ 86
4.8.6 br ······ 87

异步社区 www.epubit.com

4.8.7 br_if …… 89
4.8.8 return …… 90
4.8.9 br_table …… 90
4.9 导入和导出 …… 91
4.9.1 导出对象 …… 91
4.9.2 导入对象 …… 93
4.10 start()函数及指令折叠 …… 96
4.10.1 start()函数 …… 96
4.10.2 指令折叠 …… 97
第 5 章 WebAssembly 二进制格式 …… 99
5.1 LEB128 编码 …… 99
5.1.1 LEB128 编码原理 …… 99
5.1.2 无符号数的 LEB128 编码 …… 100
5.1.3 有符号数的 LEB128 编码 …… 101
5.2 头部和段数据 …… 101
5.2.1 头部 …… 101
5.2.2 段类型列表 …… 101
5.2.3 段数据结构 …… 102
5.3 内存段和数据段 …… 104
5.3.1 内存段 …… 104
5.3.2 数据段 …… 105
5.4 表格段和元素段 …… 106
5.4.1 表格段 …… 107
5.4.2 元素段 …… 108
5.5 开始段和函数索引 …… 108
5.5.1 开始段 …… 109
5.5.2 函数索引 …… 110
5.6 全局段 …… 111
5.6.1 全局变量索引 …… 111
5.6.2 全局段编码方式 …… 112
5.7 函数段、代码段和类型段 …… 113
5.7.1 函数段、代码段和类型段之间的关系 …… 113

异步社区 www.epubit.com

5.7.2 更简单的例子 …… 114
5.7.3 函数段 …… 115
5.7.4 类型段 …… 116
5.7.5 代码段 …… 117
5.8 导入段和导出段 …… 118
5.8.1 例子 …… 118
5.8.2 类型段 …… 119
5.8.3 导入段 …… 119
5.8.4 导出段 …… 120
5.9 自定义段 …… 121
5.9.1 自定义段结构 …… 122
5.9.2 模块的名字 …… 122
5.9.3 全局变量的名字 …… 123
5.9.4 函数的名字 …… 124
5.9.5 局部变量的名字 …… 124
5.10 补充说明 …… 126

第 6 章 Emscripten 和 WebAssembly …… 127

6.1 安装环境 …… 127
6.1.1 emsdk 命令安装 …… 127
6.1.2 Docker 环境安装 …… 128
6.1.3 验证 emcc 命令 …… 129
6.2 你好，Emscripten！ …… 129
6.2.1 生成 wasm 文件 …… 130
6.2.2 浏览器环境 …… 130
6.2.3 自动生成 HTML 测试文件 …… 132
6.3 C/C++内联 JavaScript 代码 …… 133
6.3.1 EM_ASM 宏 …… 133
6.3.2 EM_ASM_ 宏 …… 134
6.3.3 EM_ASM_*宏 …… 135
6.3.4 函数参数 …… 137
6.3.5 注意问题 …… 138
6.4 C/C++调用 JavaScript 函数 …… 139

异步社区 www.epubit.com

6.4.1 C 语言版本的 eval() 函数 …… 139
6.4.2 打造带参数的 eval() 函数 …… 141
6.4.3 打造可变参数的 eval() 函数 …… 143
6.4.4 eval() 函数返回字符串 …… 146
6.5 JavaScript 调用 C 导出函数 …… 149
6.5.1 调用导出函数 …… 149
6.5.2 辅助函数 ccall() 和 cwrap() …… 152
6.6 运行时和消息循环 …… 155
6.6.1 Emscripten 运行时 …… 155
6.6.2 消息循环 …… 158
6.7 补充说明 …… 162
第 7 章 Go 语言和 WebAssembly …… 163
7.1 你好，Go 语言 …… 163
7.2 浏览器中的 Go 语言 …… 166
7.3 使用 JavaScript 函数 …… 168
7.4 回调 Go 函数 …… 170
7.5 syscall/js 包 …… 172
7.6 WebAssembly 模块的导入函数 …… 175
7.7 WebAssembly 虚拟机 …… 178
7.8 补充说明 …… 180
附录 指令参考 …… 181

异步社区 www.epubit.com

第 0 章

WebAssembly 诞生背景

一切可编译为 WebAssembly 的，终将被编译为 WebAssembly。

——Ending

WebAssembly 是一种新兴的网页虚拟机标准，它的设计目标包括：高可移植性、高安全性、高效率（包括载入效率和运行效率）、尽可能小的程序体积。

0.1 JavaScript 简史

只有了解了过去才能理解现在，只有理解了现在才可能掌握未来的发展趋势。JavaScript 语言因为互联网而生，紧随着浏览器的出现而问世。JavaScript 语言是 Brendan Eich 为网景（Netscape）公司的浏览器设计的脚本语言，据说前后只花了 10 天的时间就设计成型。为了借当时的明星语言 Java 的东风，这门新语言被命名为 JavaScript。其实 Java 语言和 JavaScript 语言就像是雷锋和雷峰塔一样没有什么关系。

JavaScript 语言从诞生开始就是严肃程序员鄙视的对象：语言设计垃圾、运行比蜗牛还慢、它只是给不懂编程的人用的玩具等。当然出现这些观点也有一定的客观因素：JavaScript 运行确实够慢，语言也没有经过严谨的设计，甚至没有很多高级语言标配的块作用域特性。

但是到了 2005 年，Ajax（Asynchronous JavaScript and XML）方法横空出世，JavaScript 终于开始火爆。据说是 Jesse James Garrett 发明了这个词汇。谷歌当时发布的 Google Maps

项目大量采用该方法标志着 Ajax 开始流行。Ajax 几乎成了新一代网站的标准做法，并且促成了 Web 2.0 时代的来临。作为 Ajax 核心部分的 JavaScript 语言突然变得异常重要。

然后是 2008 年，谷歌公司为 Chrome 浏览器而开发的 V8 即时编译器引擎的诞生彻底改变了 JavaScript 低能儿的形象。V8 引擎下的 JavaScript 语言突然成了地球上最快的脚本语言！在很多场景下的性能已经和 C/C++程序在一个数量级（作为参考，JavaScript 比 Python 要快 10～100 倍或更多）。

JavaScript 终于手握 Ajax 和 V8 两大神器，此后真的是飞速发展。2009 年，Ryan Dahl 创建 Node.js 项目，JavaScript 开始进军服务器领域。2013 年，Facebook 公司发布 React 项目，2015 年发布 React Native 项目。目前 JavaScript 语言已经开始颠覆 iOS 和 Android 等手机应用的开发。

回顾整个互联网技术的发展历程，可以发现在 Web 发展历程中出现过各种各样的技术，例如，号称跨平台的 Java Applet、仅支持 IE 浏览器的 ActiveX 控件、曾经差点称霸浏览器的 Flash 等。但是，在所有的脚本语言中只有 JavaScript 语言顽强地活了下来，而且有席卷整个软件领域的趋势。

JavaScript 语言被历史选中并不完全是偶然的，偶然之中也有着必然的因素。它的优点同样不可替代。

- 简单易用，不用专门学习就可以使用。
- 运行系统极其稳定，想写出一个让 JavaScript 崩溃的程序真是一个挑战。
- 紧抱 HTML 标准，站在 Ajax、WebSocket、WebGL、WebWorker、SIMD.js 等前沿技术的肩膀之上。

因为 Web 是一个开放的生态，所以如果一个技术太严谨注定就不会流行，XHTML 就是一个活生生的反面教材。超强容错是所有 Web 技术流行的一个必备条件。JavaScript 刚好足够简单和稳定、有着超强的容错能力、语言本身也有着极强的表达力。同时，Ajax、WebSocket、WebGL、WebWorker 等标准的诞生也为 JavaScript 提供了更广阔的应用领域。

异步社区 www.epubit.com

0.2 asm.js 的尝试

JavaScript 是弱类型语言，由于其变量类型不固定，使用变量前需要先判断其类型，这样无疑增加了运算的复杂度，降低了执行效能。谋智公司的工程师创建了 Emscripten 项目，尝试通过 LLVM 工具链将 C/C++语言编写的程序转译为 JavaScript 代码，在此过程中创建了 JavaScript 子集 asm.js，asm.js 仅包含可以预判变量类型的数值运算，有效地避免了 JavaScript 弱类型变量语法带来的执行效能低下的问题。根据测试，针对 asm.js

优化的引擎执行速度和 C/C++原生应用在一个数量级。

图 0-1 给出了 ams.js 优化的处理流程，其中上一条分支是经过高度优化的执行分支。

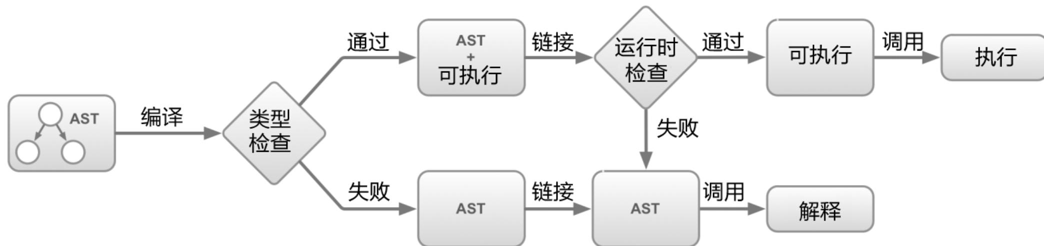

图 0-1

因为增加了类型信息，所以 asm.js 代码采用定制优化的 AOT（Ahead Of Time）编译器，生成机器指令执行。如果中途发现语法错误或违反类型标记的情况出现，则回退到传统的 JavaScript 引擎解析执行。

asm.js 中只有有符号整数、无符号整数和浮点数这几种类型。图 0-2 展示了 asm.js 中几种数值类型之间的关系。

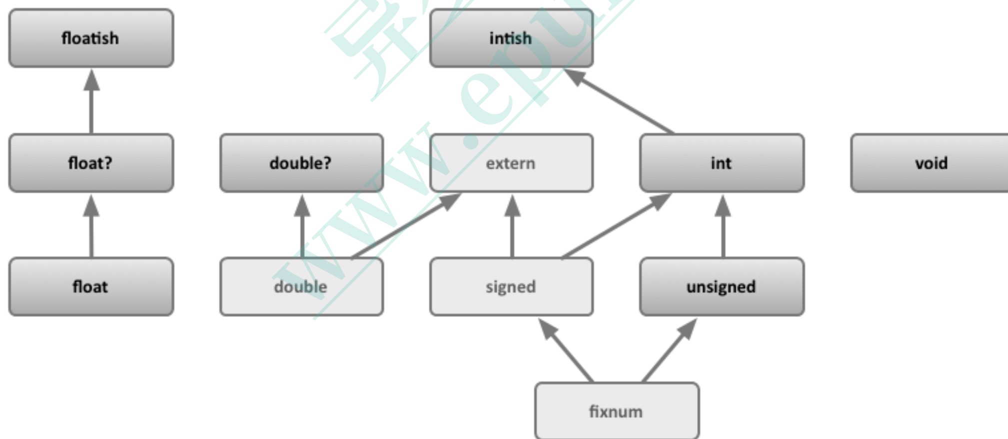

图 0-2

而字符串等其他类型则必须在 `TypedArray` 中提供，同时通过类似指针的技术访问 `TypedArray` 中的字符串数据。

asm.js 的高明之处就是通过 JavaScript 已有的语法和语义为变量增加了类型信息：

```
var x = 9527;

var i = a | 0;    // int32
var u = a >>> 0; // uint32
var f = +a;       // float64
```

以上代码中 a|0 表示一个整数，而 a >>> 0 表示一个无符号整数，+a 表示一个浮点数。即使不支持 asm.js 的引擎也可以产生正确的结果。

然后通过 ArrayBuffer 来模拟真正的内存：

```
var buffer = new ArrayBuffer(1024*1024*8);
var HEAP8 = new Int8Array(buffer);
```

基于 HEAP8 模拟的内存，就可以采用类似 C 语言风格计算的字符串长度。而 C 语言的工作模型和冯・诺伊曼计算机体系结构是高度适配的，这也是 C 语言应用具有较高性能的原因。下面是 asm.js 实现的 strlen 函数：

```
function strlen(s) {
    var p = s|0;
    while(HEAP8[p]|0 != 0) {
        p = (p+1)|0
    }
    return (p-s)|0;
}
```

所有的 asm.js 代码将被组织到一个模块中：

```
function MyasmModule(stdlib, foreign, heap) {
    "use asm";

    var HEAP8 = new Int8Array(heap);

function strlen(s) {
      // ...
    }

    return {
strlen: strlen,
    };
}
```

asm.js 模块通过 stdlib 提供了基本的标准库，通过 foreign 可以传入外部定义

的函数，通过 heap 为模块配置堆内存。最后，模块可以将 asm.js 实现的函数或变量导出。模块开头通过"use asm"标注内部是 asm.js 规格的实现，即使是旧的引擎也可以正确运行。

asm.js 优越的性能让浏览器能够运行很多 C/C++开发的 3D 游戏。同时，Lua、SQLite 等 C/C++开发的软件被大量编译为纯 JavaScript 代码，极大地丰富了 JavaScript 社区的生态。asm.js 诸多技术细节我们就不详细展开了，感兴趣的读者可以参考它的规范文档。

0.3 WebAssembly 的救赎

在整个 Web 技术变革的过程之中，不断有技术人员尝试在浏览器中直接运行 C/C++程序。自 1995 年起包括 Netscape Plugin API（NPAPI）在内的许多知名项目相继开发。微软公司的 IE 浏览器甚至可以直接嵌入运行本地代码的 ActiveX 控件。同样，谷歌公司在 2010 年也开发了一项 Native Clients 技术（简称 NaCL）。然而这些技术都太过复杂、容错性也不够强大，它们最终都未能成为行业标准。

除尝试直接运行本地代码这条路之外，也有技术人员开始另辟蹊径，将其他语言直接转译为 JavaScript 后运行，2006 年，谷歌公司推出 Google Web Toolkit（GWT）项目，提供将 Java 转译成 JavaScript 的功能，开创了将其他语言转为 JavaScript 的先河。之后的 CoffeeScript、Dart、TypeScript 等语言都是以输出 JavaScript 程序为最终目标。

在众多为 JavaScript 提速的技术中，Emscripten 是与众不同的一个。它利用 LLVM 编译器前端编译 C/C++代码，生成 LLVM 特有的跨平台中间语言代码，最终再将 LLVM 跨平台中间语言代码转译为 JavaScript 的 asm.js 子集。这带来的直接结果就是，C/C++程序经过编译后不仅可在旧的 JavaScript 引擎上正确运行，同时也可以被优化为机器码之后高速运行。

2015 年 6 月谋智公司在 asm.js 的基础上发布了 WebAssembly 项目，随后谷歌、微软、苹果等各大主流的浏览器厂商均大力支持。WebAssembly 不仅拥有比 asm.js 更高的执行效能，而且由于使用了二进制编码等一系列技术，WebAssembly 编写的模块体积更小且解析速度更快。目前不仅 C/C++语言编写的程序可以编译为 WebAssembly 模块，而且 Go、Kotlin、Rust 等新兴的编程语言都开始对 WebAssembly 提供支持。2018 年 7 月，WebAssembly 1.0 标准正式发布，这必将开辟 Web 开发的新纪元！

异步社区 www.epubit.com

异步社区
www.epubit.com

第 2 章

WebAssembly 快速入门

WebAssembly，一次编写到处运行。

——yjhmelody

本章将快速展示几个小例子，借此对 WebAssembly 形成一个大致的印象，掌握一些基本的用法。在后续的章节将会系统、深入地学习各个技术细节。

2.1 准备工作

“工欲善其事，必先利其器。”在正式开始之前，需要先准备好兼容 WebAssembly 的运行环境以及 WebAssembly 文本格式转换工具集。

2.1.1 WebAssembly 兼容性

常见桌面版浏览器及 Node.js 对 WebAssembly 特性的支持情况如表 2-1 所示，表内的数字表示浏览器版本。

异步社区 www.epubit.com

表 2-1

WebAssembly 特性	Chrome	Edge	Firefox	IE	Opera	Safari	Node.js
基本支持	57	16	52	不支持	44	11	8.0.0
CompileError	57	16	52	不支持	44	11	8.0.0
Global	不支持	不支持	62	不支持	不支持	不支持	不支持
Instance	57	16	52	不支持	44	11	8.0.0
LinkError	57	16	52	不支持	44	11	8.0.0
Memory	57	16	52	不支持	44	11	8.0.0
Module	57	16	52	不支持	44	11	8.0.0
RuntimeError	57	16	52	不支持	44	11	8.0.0
Table	57	16	52	不支持	44	11	8.0.0
compile	57	16	52	不支持	44	11	8.0.0
compileStreaming	61	16	58	不支持	47	不支持	不支持
instantiate	57	16	52	不支持	44	11	8.0.0
instantiateStreaming	61	16	58	不支持	47	不支持	不支持
validate	57	16	52	不支持	44	11	8.0.0

常见移动版（Android 及 iOS）浏览器对 WebAssembly 特性的支持情况如表 2-2 所示，表内的数字表示浏览器版本。

表 2-2

WebAssembly 特性	Webview	Chrome	Edge	Firefox	Opera	Safari	Samsung Internet
基本支持	57	57	支持	52	未知	11	7.0
CompileError	57	57	支持	52	未知	11	7.0
Global	不支持	不支持	不支持	62	未知	不支持	不支持
Instance	57	57	支持	52	未知	11	7.0
LinkError	57	57	支持	52	未知	11	7.0
Memory	57	57	支持	52	未知	11	7.0
Module	57	57	支持	52	未知	11	7.0
RuntimeError	57	57	支持	52	未知	11	7.0
Table	57	57	支持	52	未知	11	7.0
compile	57	57	支持	52	未知	11	7.0

续表

WebAssembly 特性	Webview	Chrome	Edge	Firefox	Opera	Safari	Samsung Internet
compileStreaming	61	61	不支持	58	未知	不支持	不支持
instantiate	57	57	支持	52	未知	11	7.0
instantiateStreaming	61	61	不支持	58	未知	不支持	不支持
validate	57	57	支持	52	未知	11	7.0

可见大多数现代浏览器都已支持 WebAssembly，可以在列表中任选一种支持 WebAssembly 的浏览器来运行测试本书的例程。

当浏览器启用 WebAssembly 模块时，会强行启用同源及沙盒等安全策略。因此本书的 WebAssembly 例程需通过 http 网页发布后方可运行。本书的例程目录中有一个名为“py_simple_server.bat”的批处理文件，该文件用于在 Windows 操作系统下使用 python 将当前目录发布为 http 服务；当然也可以使用 Nginx、IIS、Apache 或任意一种惯用的工具来完成该操作。

2.1.2 WebAssembly 文本格式与 wabt 工具集

一般来说，浏览器加载并运行的 WebAssembly 程序是二进制格式的 WebAssembly 汇编代码，文件扩展名通常为.wasm。由于二进制文件难以阅读编辑，WebAssembly 提供了一种基于 S-表达式的文本格式，文件扩展名通常为.wat。下面是一个 WebAssembly 文本格式的例子：

```
(module
   (import "console" "log" (func $log (param i32)))
   (func $add (param i32 i32)
      get_local 0
      get_local 1
      i32.add
      call $log
   )
   (export "add" (func $add))
)
```

上述程序定义了一个名为$add 的函数，该函数将两个 i32 类型的输入参数相加，并使用由外部 JavaScript 导入的 log 函数将结果输出。最后该 add 函数被导出，以供外

部 JavaScript 环境调用。

WebAssembly 文本格式（.wat）与 WebAssembly 汇编格式（.wasm）的关系类似于宏汇编代码与机器码的关系。如同.asm 文件向机器码转换需要使用 nasm 这样的编译工具一样，.wat 文件向.wasm 文件的转换需要用到 wabt 工具集，该工具集提供了.wat 与.wasm 相互转换的编译器等功能。

wabt 工具集可从 GitHub 上下载获取。按照页面说明下载编译后将获得 `wat2wasm` 程序。在命令行下执行

```
wat2wasm input.wat -o output.wasm
```

即可将 WebAssembly 文本格式文件 input.wat 编译为 WebAssembly 汇编格式文件 output.wasm。

使用`-v` 选项调用 `wat2wasm` 可以查看汇编输出，例如将前述的代码保存为 test.wat 并执行：

```
wat2wasm test.wat -v
```

终端输出如图 2-1 所示。

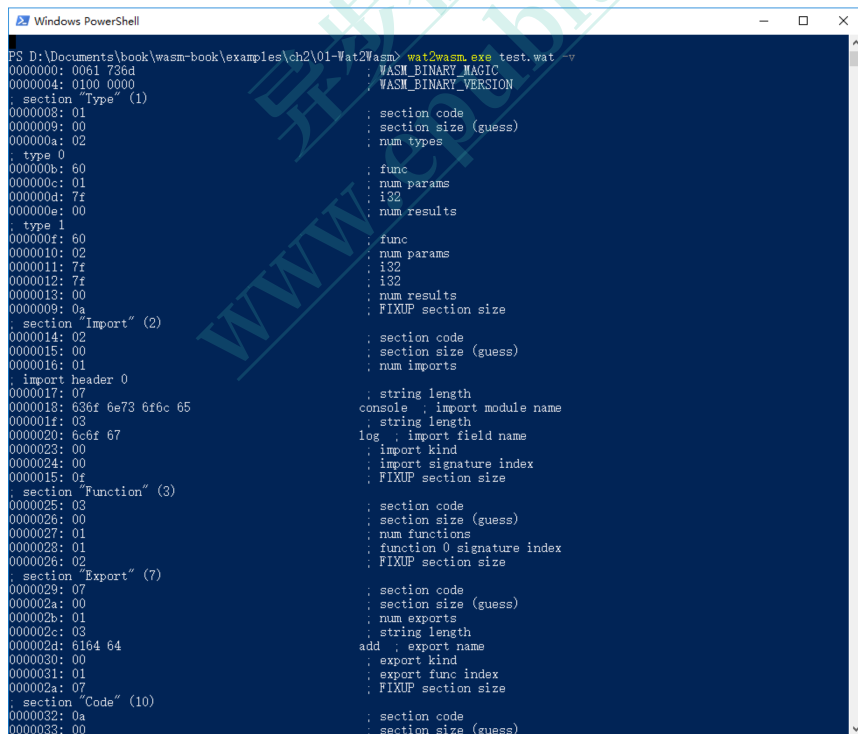

图 2-1

异步社区 www.epubit.com

第 5 章介绍 WebAssembly 二进制格式时给出的示例代码中含有二进制的部分，除此之外其他章节的 WebAssembly 代码均以文本格式（.wat）提供以便于阅读。

2.2 首个例程

很多语言的入门教程都始于“Hello, World!”例程，但是对于 WebAssembly 来说，一个完整的“Hello, World!”程序仍然过于复杂，因此，我们将从一个更简单的例子开始。本节的例程名为 ShowMeTheAnswer，其中 WebAssembly 代码位于 show_me_the_answer.wat 中，如下：

```
(module
    (func (export "showMeTheAnswer") (result i32)
        i32.const 42
    )
)
```

上述代码定义了一个返回值为 42（32 位整型数）的函数，并将该函数以 showMeTheAnswer 为名字导出，供 JavaScript 调用。

JavaScript 代码位于 show_me_the_answer.html 中，如下：

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Show me the answer</title>
    </head>
    <body>
        <script>
            fetch('show_me_the_answer.wasm').then(response =>
                response.arrayBuffer()
            ).then(bytes =>
                WebAssembly.instantiate(bytes)
            ).then(result =>
                console.log(result.instance.exports.showMeTheAnswer()) //42
            );
        </script>
```

异步社区 www.epubit.com

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

上述代码首先使用 fetch()函数获取 WebAssembly 汇编代码文件，将其转为 ArrayBuffer 后使用 WebAssembly.instantiate()函数对其进行编译及初始化实例，最后调用该实例导出的函数 showMeTheAnswer()并打印结果。

将例程目录发布后，通过浏览器访问 show_me_the_answer.html，浏览器控制台应输出结果 42，如图 2-2 所示。

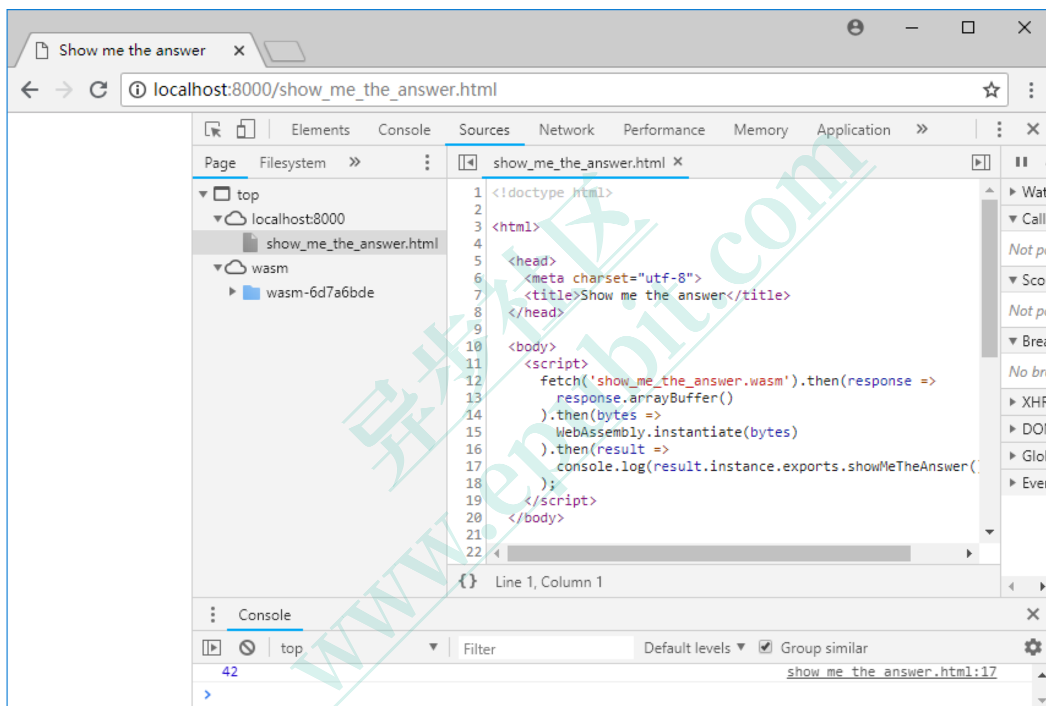

图 2-2

2.3　WebAssembly 概览

JavaScript 代码运行在 JavaScript 虚拟机上，相对地，WebAssembly 代码也运行在其特有的虚拟机上。参照 2.1 节内容，大部分最新的浏览器均提供了 WebAssembly 虚拟机，然而 WebAssembly 代码并非只能在浏览器中运行，Node.js 8.0 之后的版本也能运行 WebAssembly；更进一步来说，WebAssembly 虚拟机甚至可以脱离 JavaScript 环境的支持。

不过正如其名，WebAssembly 的设计初衷是运行于网页之中，因此本书绝大部分内容均是围绕网页应用展开。

2.3.1 WebAssembly 中的关键概念

在深入 WebAssembly 内部运行机制之前，我们暂时将其看作一个黑盒，这个黑盒需要通过一些手段来与外部环境——调用 WebAssembly 的 JavaScript 网页程序等进行交互，这些手段可以抽象为以下几个关键概念。

1. 模块

模块是已被编译为可执行机器码的二进制对象，模块可以简单地与操作系统中的本地可执行程序进行类比，是无状态的（正如 Windows 的.exe 文件是无状态的）。模块是由 WebAssembly 二进制汇编代码（.wasm）编译而来的。

2. 内存

在 WebAssembly 代码看来，内存是一段连续的空间，可以使用 `load`、`store` 等低级指令按照地址读写其中的数据（时刻记住 WebAssembly 汇编语言是一种低级语言），在网页环境下，WebAssembly 内存是由 JavaScript 中的 `ArrayBuffer` 对象实现的，这意味着，整个 WebAssembly 的内存空间对 JavaScript 来说是完全可见的，JavaScript 和 WebAssembly 可以通过内存交换数据。

3. 表格

C/C++等语言强烈依赖于函数指针，WebAssembly 汇编代码作为它们的编译目标，必须提供相应的支持。受限于 WebAssembly 虚拟机结构和安全性的考虑，WebAssembly 引入了表格对象用于存储函数引用，后续章节将对表格对象进行详细介绍。

4. 实例

用可执行程序与进程的关系进行类比，在 WebAssembly 中，实例用于指代一个模块及其运行时的所有状态，包括内存、表格以及导入对象等，配置这些对象并基于模块创建一个可被调用的实例的过程称为实例化。模块只有在实例化之后才能被调用——这与 C++/Java 中类与其实例的关系是类似的。

异步社区 www.epubit.com

导入/导出对象是模块实例很重要的组成部分，模块内部的 WebAssembly 代码可以通过导入对象中的导入函数调用外部 JavaScript 环境的方法，导出对象中的导出函数是模块提供给外部 JavaScript 环境使用的接口。

按照目前的规范，一个实例只能拥有一个内存对象以及一个表格对象。内存对象和表格对象都可以通过导入/导出对象被多个实例共有，这意味着多个 WebAssembly 模块可以以.dll 动态链接库的模式协同工作。

2.3.2 WebAssembly 程序生命周期

WebAssembly 程序从开发到运行于网页中大致可以分为以下几个阶段。

（1）使用 WebAssembly 文本格式或其他语言（C++、Go、Rust 等）编写程序，通过各自的工具链编译为 WebAssembly 汇编格式.wasm 文件。

（2）在网页中使用 `fetch`、`XMLHttpRequest` 等获取.wasm 文件（二进制流）。

（3）将.wasm 编译为模块，编译过程中进行合法性检查。

（4）实例化。初始化导入对象，创建模块的实例。

（5）执行实例的导出函数，完成所需操作。

流程图如图 2-3 所示。

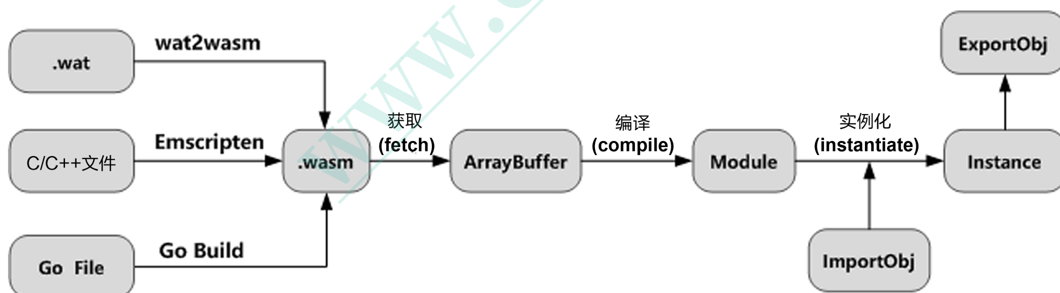

图 2-3

第 2 步到第 5 步为 WebAssembly 程序的运行阶段，该阶段与 JavaScript 环境密切相关，第 3 章将系统地介绍相关知识。

2.3.3 WebAssembly 虚拟机体系结构

WebAssembly 模块在运行时由以下几部分组成，如图 2-4 所示。

（1）一个全局类型数组。与很多语言不同，在 WebAssembly 中“类型”指的并非数据类型，而是函数签名，函数签名定义了函数的参数个数/参数类型/返回值类型；某个函数签名在类型数组中的下标（或者说位置）称为*类型索引*。

（2）一个全局函数数组，其中容纳了所有的函数，包括导入的函数以及模块内部定义的函数，某个函数在函数数组中的下标称为*函数索引*。

（3）一个全局变量数组，其中容纳了所有的全局变量——包括导入的全局变量以及模块内部定义的全局变量，某个全局变量在全局变量数组中的下标称为*全局变量索引*。

（4）一个全局表格对象，表格也是一个数组，其中存储了元素（目前元素类型只能为函数）的引用，某个元素在表格中的下标称为*元素索引*。

（5）一个全局内存对象。

（6）一个运行时栈。

（7）函数执行时可以访问一个局部变量数组，其中容纳了函数所有的局部变量，某个局部变量在局部变量数组中的下标称为*局部变量索引*。

图 2-4

在 WebAssembly 中，操作某个具体的对象（如读写某个全局变量/局部变量、调用某个函数等）都是通过其索引完成的。在当前版本中，所有的“索引”都是 32 位整型数。

2.4 你好，WebAssembly

本节将介绍经典的 HelloWorld 例程。开始之前让我们先梳理一下需要完成哪些功能。

（1）函数导入。WebAssembly 虚拟机本身没有提供打印函数，因此需要将 JavaScript 中的字符串输出功能通过函数导入的方法导入 WebAssembly 中供其使用。

（2）初始化内存，并在内存中存储将要打印的字符串。

（3）函数导出，提供外部调用入口。

2.4.1　WebAssembly 部分

在 WebAssembly 部分，首先将来自 JavaScript 的 `js.print` 对象导入为函数，并命名为 `js_print`：

```
;;hello.wat
(module
    ;;import js:print as js_print():
    (import "js" "print" (func $js_print (param i32 i32)))
```

该函数有两个参数，类型均为 32 位整型数，第一个参数为将要打印的字符串在内存中的开始地址，第二个参数为字符串的长度（字节数）。

> **提示**　WebAssembly 文本格式中，双分号“;;”表示该行后续为注释，作用类似于 JavaScript 中的双斜杠“//”。

接下来将来自 JavaScript 的 `js.mem` 对象导入为内存：

```
(import "js" "mem" (memory 1)) ;;import js:mem as memory
```

`memory` 后的 1 表示内存的起始长度至少为 1 页（在 WebAssembly 中，1 页=64 KB=65 536 字节）。

接下来的 `data` 段将字符串"你好，WASM"写入了内存，起始地址为 0：

```
(data (i32.const 0) "你好，WASM")
```

最后定义了导出函数 `hello()`：

```
    (func (export "hello")
        i32.const 0    ;;pass offset 0 to js_print
        i32.const 13   ;;pass length 13 to js_print
        call $js_print
    )
)
```

异步社区 www.epubit.com

函数 hello()先后在栈上压入了字符串的起始地址 0 以及字符串的字节长度 13（每个汉字及全角标点的 UTF-8 编码占 3 字节），然后调用导入的 js_print()函数打印输出。

2.4.2　JavaScript 部分

在 JavaScript 部分，首先创建内存对象 wasmMem，初始长度为 1 页：

```
//hello.html
      var wasmMem = new WebAssembly.Memory({initial:1});
```

接下来定义用于打印字符串的方法 printStr()：

```
function printStr(offset, length) {
    var bytes = new Uint8Array(wasmMem.buffer, offset, length);
    var string = new TextDecoder('utf8').decode(bytes);
    console.log(string);
}
```

对应于.wat 部分的定义，该方法的两个参数分别为字符串在内存中的起始地址及字节长度。从内存中获取字节流后，使用 TextDecoder 将其解码为字符串并输出。

然后将上述 Memory 对象 wasmMem、printStr()方法组合成对象 importObj，导入并实例化：

```
var importObj = { js: { print: printStr, mem: wasmMem } };

fetch('hello.wasm').then(response =>
    response.arrayBuffer()
).then(bytes =>
    WebAssembly.instantiate(bytes, importObj)
).then(result =>
    result.instance.exports.hello()
);
```

最后调用实例的导出函数 hello()，在控制台输出“你好，WASM”，如图 2-5 所示。

异步社区 www.epubit.com

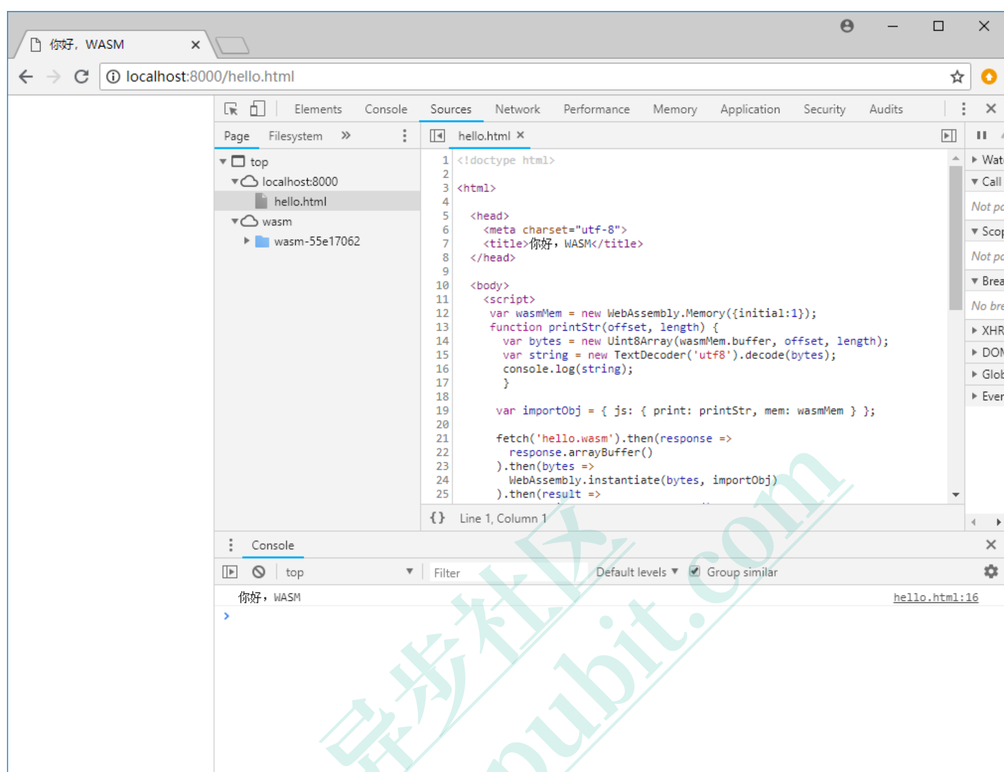

图 2-5

异步社区 www.epubit.com

2.5　WebAssembly 调试及代码编辑环境

作为最主要的WebAssembly运行平台，浏览器普遍提供了WebAssembly的调试环境，当页面上包含 WebAssembly 模块时，可以使用开发面板对其运行进行调试。

图 2-6 是在 Chrome 中使用 F12 调出开发面板调试程序的截图，我们可以在 WebAssembly 的函数体中下断点，查看局部变量/全局变量的值，查看调用栈和当前函数栈等。

WebAssembly 文本格式（.wat）文件可以使用任何文本编辑器编辑，对于 Windows 用户我们推荐使用 VSCode，并安装 WebAssembly 插件，如图 2-7 所示。

该插件除.wat 文件编辑器语法高亮之外，甚至还支持直接打开.wasm 文件反汇编，图 2-8 是安装插件后打开 2.4 节例子的 hello.wasm 文件的截图。

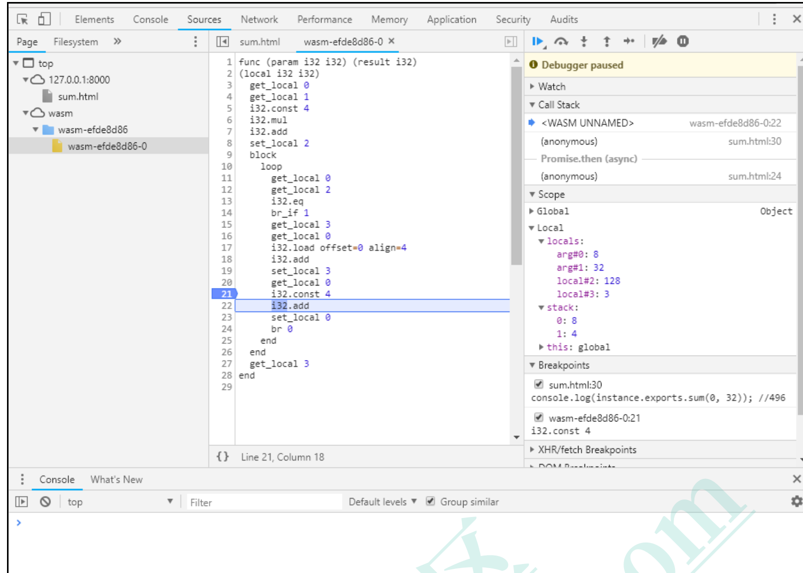

图 2-6

图 2-7

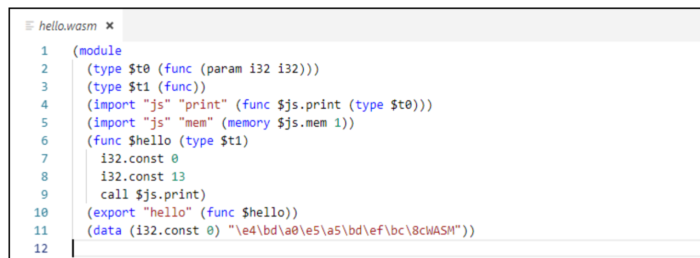

图 2-8

异步社区 www.epubit.com

异步社区
www.epubit.com

第 3 章

JavaScript 中的 WebAssembly 对象

异步社区 www.epubit.com

WebAssembly 的到来，使自制虚拟机不再遥远。

——Ending

在浏览器环境中，WebAssembly 程序运行在 WebAssembly 虚拟机之上，页面可以通过一组 JavaScript 对象进行 WebAssembly 模块的编译、载入、配置、调用等操作。本章将介绍这些 JavaScript 对象的使用方法。

3.1 WebAssembly 对象简介

在 2.3 节中，我们介绍了 WebAssembly 中的几个关键概念：模块、内存、表格以及实例。事实上，每个概念在 JavaScript 中都有对象与之一一对应，分别为 `WebAssembly.Module`、`WebAssembly.Memory`、`WebAssembly.Table` 以及 `WebAssembly.Instance`。

所有与 WebAssembly 相关的功能，都属于全局对象 `WebAssembly`。除刚才提到的对象之外，`WebAssembly` 对象中还包含了一些全局方法，如之前章节中曾出现的用于执行实例化的 `WebAssembly.instantiate()`。

值得注意的是，与很多全局对象（如 `Date`）不同，`WebAssembly` 不是一个构造函数，而是一个命名空间，这与 `Math` 对象相似——当我们使用 WebAssembly 相关功能时，直接调用 `WebAssembly.XXX()`，无须（也不能）使用类似于 `Date()` 的构造函数。

后续各节在具体介绍 WebAssembly 相关对象时使用的例子中，包含了一些.wasm 汇编模块，本章将暂不深入汇编模块的内部。WebAssembly 汇编语言将在第 4 章中介绍。

3.2　全局方法

本节将介绍全局对象 WebAssembly 的全局方法，这些方法主要用于 WebAssembly 二进制模块的编译及合法性检查。

3.2.1　**WebAssembly.compile()**

该方法用于将 WebAssembly 二进制代码（.wasm）编译为 `WebAssembly.Module`。语法为：

```
Promise<WebAssembly.Module>WebAssembly.compile(bufferSource);
```

参数

- `bufferSource`：包含 WebAssembly 二进制代码（.wasm）的 `TypedArray` 或 `ArrayBuffer`。

返回值

- `Promise` 对象，编译好的模块对象，类型为 `WebAssembly.Module`。

异常

- 如果传入的 `bufferSource` 不是 `TypedArray` 或 `ArrayBuffer`，将抛出 `TypeError`。
- 如果编译失败，将抛出 `WebAssembly.CompileError`。

示例

```
fetch('show_me_the_answer.wasm').then(response =>
    response.arrayBuffer()
```

异步社区 www.epubit.com

```
).then(bytes =>
    WebAssembly.compile(bytes)
).then(module =>
    console.log(module.toString()) //"object WebAssembly.Module"
);
```

3.2.2 WebAssembly.instantiate()

该方法有两种重载形式。第一种用于将 WebAssembly 二进制代码编译为模块，并创建其第一个实例，语法为：

```
Promise<ResultObject>WebAssembly.instantiate(bufferSource, importObject);
```

参数

- `bufferSource`：包含 WebAssembly 二进制代码（.wasm）的 `TypedArray` 或 `ArrayBuffer`。
- `importObject`：可选，将被导入新创建的实例中的对象，它可以包含 JavaScript 方法、`WebAssembly.Memory` 和 `WebAssembly.Table`。

返回值

- `Promise` 对象，该对象包含两个属性。
  - `module`：编译好的模块对象，类型为 `WebAssembly.Module`。
  - `instance`：上述 `module` 的第一个实例，类型为 `WebAssembly.Instance`。

异常

- 如果传入的 `bufferSource` 不是 `TypedArray` 或 `ArrayBuffer`，将抛出 `TypeError`。
- 如果操作失败，根据失败的不同原因，`Promise` 会抛出下面 3 种异常之一：`WebAssembly.CompileError`、`WebAssembly.LinkError` 和 `WebAssembly.RuntimeError`。

2.2.节中我们就曾经使用过这种重载形式：

```
fetch('show_me_the_answer.wasm').then(response =>
    response.arrayBuffer()
).then(bytes =>
    WebAssembly.instantiate(bytes)
```

异步社区 www.epubit.com

```
).then(result =>
    console.log(result.instance.exports.showMeTheAnswer()) //42
);
```

WebAssembly.instantiate()的另一种重载形式用于基于已编译好的模块创建实例，语法为：

```
Promise<WebAssembly.Instance>WebAssembly.instantiate(module, importObject);
```

参数

- module：已编译好的模块对象，类型为 WebAssembly.Module。
- importObject：可选，将被导入新创建的实例中的对象。

返回值

- Promise 对象，新建的实例，类型为 WebAssembly.Instance。

异常

- 如果参数类型或结构不正确，将抛出 TypeError。
- 如果操作失败，根据失败的不同原因，抛出下述 3 种异常之一：WebAssembly.CompileError、WebAssembly.LinkError 和 WebAssembly.RuntimeError。

在需要为一个模块创建多个实例时，使用这种重载形式可以省去多次编译模块的开销。例如：

```
fetch('show_me_the_answer.wasm').then(response =>
    response.arrayBuffer()
).then(bytes =>
    WebAssembly.compile(bytes)
).then(mod => {
        WebAssembly.instantiate(mod).then(result =>
            console.log('Instance0:', result.exports.showMeTheAnswer()));
        WebAssembly.instantiate(mod).then(result =>
            console.log('Instance1:', result.exports.showMeTheAnswer()));
    }
);
```

3.2.3 WebAssembly.validate()

该方法用于校验 WebAssembly 二进制代码是否合法，语法为：

异步社区 www.epubit.com

```
var valid = WebAssembly.validate(bufferSource);
```

参数

- bufferSource：包含 WebAssembly 二进制代码（.wasm）的 TypedArray 或 ArrayBuffer。

返回值

- 布尔型，合法返回 true，否则返回 false。

异常

- 如果传入的 bufferSource 不是 TypedArray 或 ArrayBuffer，将抛出 TypeError。

3.2.4 **WebAssembly.compileStreaming()**

该方法与 WebAssembly.compile()类似，用于 WebAssembly 二进制代码的编译，区别在于本方法使用流式底层源作为输入：

```
Promise<WebAssembly.Module>WebAssembly.compileStreaming(source);
```

参数 source 通常是 fetch()方法返回的 Response 对象，例如：

```
WebAssembly.compileStreaming(fetch('show_me_the_answer.wasm')).
    then(module => console.log(module.toString()) //"object WebAssembly.Module"
);
```

本方法的返回值和异常与 WebAssembly.compile()相同。

3.2.5 **WebAssembly.instantiateStreaming()**

该方法与 WebAssembly.instantiate()的第一种重载形式类似，用于将 WebAssembly 二进制代码编译为模块，并创建其第一个实例，区别在于本方法使用流式底层源作为输入：

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

参数 source 通常是 fetch()方法返回的 Response 对象，例如：

异步社区 www.epubit.com

```
WebAssembly.instantiateStreaming(fetch('show_me_the_answer.wasm')).
    then(result =>
    console.log(result.instance.exports.showMeTheAnswer()) //42
);
```

本方法的返回值和异常与 WebAssembly.instantiate() 的第一种重载形式相同。

> **提示**　WebAssembly.compileStreaming()/WebAssembly.instantiate-Streaming() 与其非流式版的孪生函数相比，虽然书写较为简单，但是对浏览器的要求较高（目前 Safari 全系不支持），因此，若无特殊情况，不建议使用这一对函数。

3.3　WebAssembly.Module 对象

与模块对应的 JavaScript 对象为 WebAssembly.Module，它是无状态的，可以被多次实例化。

将 WebAssembly 二进制代码（.wasm）编译为模块需要消耗相当大的计算资源，因此获取模块的主要方法是上一节中讲到的异步方法 WebAssembly.compile() 和 WebAssembly.instantiate()。

异步社区 www.epubit.com

3.3.1　WebAssembly.Module()

WebAssembly.Module 的构造器方法用于同步地编译.wasm 为模块：

```
var module = new WebAssembly.Module(bufferSource);
```

参数

- bufferSource：包含 WebAssembly 二进制代码（.wasm）的 TypedArray 或 ArrayBuffer。

返回值

- 编译好的模块对象。

异常

- 如果传入的 bufferSource 不是 TypedArray 或 ArrayBuffer，将抛出

TypeError。

- 如果编译失败，将抛出 WebAssembly.CompileError。

示例

```
//constructor.html
    fetch('hello.wasm').then(response =>
    response.arrayBuffer()
    ).then(bytes => {
            var module = new WebAssembly.Module(bytes);
            console.log(module.toString()); //"object WebAssembly.Module"
        }
    );
```

3.3.2 WebAssembly.Module.exports()

该方法用于获取模块的导出信息，语法为：

```
var exports = WebAssembly.Module.exports(module);
```

参数

- module：WebAssembly.Module 对象。

返回值

- module 的导出对象信息的数组。

异常

- 如果 module 不是 WebAssembly.Module，抛出 TypeError。

示例

```
//exports.html
    fetch('hello.wasm').then(response =>
        response.arrayBuffer()
    ).then(bytes =>
        WebAssembly.compile(bytes)
    ).then(module =>{
            var exports = WebAssembly.Module.exports(module);
```

异步社区 www.epubit.com

```
            for (var e in exports) {
                console.log(exports[e]);
            }
        }
    );
```

执行后，控制台将输出：

```
{name: "hello", kind: "function"}
```

3.3.3 WebAssembly.Module.imports()

该方法用于获取模块的导入信息，语法为：

```
var imports = WebAssembly.Module.imports(module);
```

参数

- module：WebAssembly.Module 对象。

返回值

- module 的导入对象信息的数组。

异常

- 如果 module 不是 WebAssembly.Module，抛出 TypeError。

示例

```
//imports.html
    fetch('hello.wasm').then(response =>
        response.arrayBuffer()
    ).then(bytes =>
        WebAssembly.compile(bytes)
    ).then(module =>{
            var imports = WebAssembly.Module.imports(module);
            for (var i in imports) {
                console.log(imports[i]);
            }
        }
    );
```

异步社区 www.epubit.com

执行后，控制台将输出：

```
{module: "js", name: "print", kind: "function"}
{module: "js", name: "mem", kind: "memory"}
```

3.3.4　WebAssembly.Module.customSections()

该方法用于获取模块中的自定义段（section）的数据。WebAssembly 代码是由一系列以 S-表达式描述的段嵌套而成，在 WebAssembly 的二进制规范中，允许包含带名字的自定义段。编译器可以利用这一特性，在生成 WebAssembly 二进制格式（.wasm）的过程中插入符号/调试信息等数据以利于运行时调试，遗憾的是目前 WebAssembly 文本格式（.wat）并不支持自定义段。

```
var sections = WebAssembly.Module.customSections(module, secName);
```

参数

- module ：WebAssembly.Module 对象。
- secName：欲获取的自定义段的名字。

返回值

- 一个数组，其中包含所有名字与 secName 相同的自定义段，每个段均为一个 ArrayBuffer。

异常

- 如果 module 不是 WebAssembly.Module，抛出 TypeError。

示例

```
//custom_sections.html
    fetch('hello.wasm').then(response =>
        response.arrayBuffer()
    ).then(bytes =>
        WebAssembly.compile(bytes)
    ).then(module =>{
            var sections = WebAssembly.Module.customSections(module, "name");
            for (var i in sections) {
```

异步社区 www.epubit.com

```
                        console.log(sections[i]);
                    }
                }
            );
```

执行后，控制台输出如图 3-1 所示。

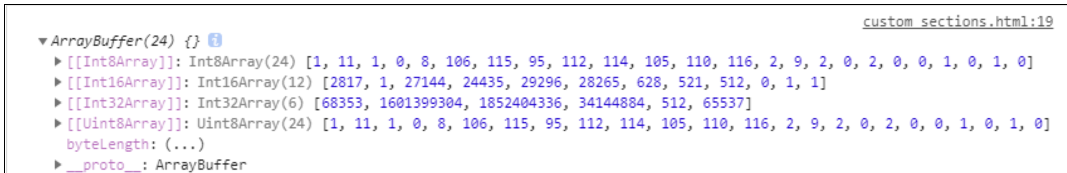

图 3-1

3.3.5　缓存 Module

在部分浏览器（如 Firefox）中，Module 可以像 Blob 一样被装入 IndexedDB 缓存，也可以在多个 Worker 间传递，下面的例子展示了这种用法（Chrome/Safari 不支持）：

```
//worker.html
    var sub_worker = new Worker("worker.js");
    sub_worker.onmessage = function (event) {
        console.log(event.data);
    }

    fetch('show_me_the_answer.wasm').then(response =>
        response.arrayBuffer()
    ).then(bytes =>
        WebAssembly.compile(bytes)
    ).then(module =>
        sub_worker.postMessage(module)
    );

//worker.js
onmessage = function (event){
    WebAssembly.instantiate(event.data).then(instance =>
        postMessage('' + instance.exports.showMeTheAnswer())
    );
}
```

异步社区 www.epubit.com

3.4 WebAssembly.Instance 对象

与实例对应的 JavaScript 对象为 WebAssembly.Instance，获取实例的主要方法是 3.2 节中介绍过的 WebAssembly.instantiate()方法。

3.4.1 WebAssembly.Instance()

实例的构造器方法，该方法用于同步地创建模块的实例，语法为：

```
var instance = new WebAssembly.Instance(module, importObject);
```

参数

- module：用于创建实例的模块。
- importObject：可选，新建实例的导入对象，它可以包含 JavaScript 方法、WebAssembly.Memory、WebAssembly.Table 和 WebAssembly 全局变量对象。

返回值

- 新创建的实例。

异常

- 如果传入参数的类型不正确，将抛出 TypeError。
- 如果链接失败，将抛出 WebAssembly.LinkError。

值得注意的是，如果 Module 中声明了导入对象，无论用哪种方法进行实例化[WebAssembly.Instance()、WebAssembly.instantiate()和 WebAssembly.instantiateStreaming()]，都必须提供完整的导入对象。例如，我们在“你好，WASM”例子的 WebAssembly 代码中声明了导入内存及 print()函数，若在实例化时不提供导入对象，则实例化将失败：

```
//linkerror.html
    fetch('hello.wasm').then(response =>
        response.arrayBuffer()
    ).then(bytes =>
        WebAssembly.compile(bytes)
    ).then(module =>
```

异步社区 www.epubit.com

```
        WebAssembly.instantiate(module);
        //TypeError: Imports argument must be present and must be an object
    );
```

即使在实例化时提供了导入对象，若不完整，例如，只导入内存，而不导入 js.print 方法，实例化仍然会失败：

```
//linkerror.html
    var wasmMem = new WebAssembly.Memory({initial:1});
    var importObj = { js: { /*print: printStr, */mem: wasmMem } };

    fetch('hello.wasm').then(response =>
        response.arrayBuffer()
    ).then(bytes =>
        WebAssembly.compile(bytes)
    ).then(module =>
        WebAssembly.instantiate(module, importObj)
    );
```

控制台输出如下：

```
Uncaught (in promise) LinkError: Import #0 module="js" function="print" error:
function import requires a callable
```

异步社区 www.epubit.com

3.4.2　WebAssembly.Instance.prototype.exports

WebAssembly.Instance 的只读属性 exports 包含了实例的所有导出函数，即实例供外部 JavaScript 程序调用的接口。例如，我们定义一个 WebAssembly 模块如下：

```
;;test.wat
(module
    (func (export "add") (param $i1 i32) (param $i2 i32) (result i32)
        get_local $i1
        get_local $i2
        i32.add
    )
    (func (export "inc") (param $i1 i32) (result i32)
        get_local $i1
        i32.const 1
        i32.add
    )
)
```

它导出了两个函数 add()以及 inc()，分别执行加法以及加 1 操作。执行 JavaScript 程序：

```
//exports.html
fetch('test.wasm').then(response =>
        response.arrayBuffer()
    ).then(bytes =>
        WebAssembly.compile(bytes)
    ).then(module =>
        WebAssembly.instantiate(module)
    ).then(instance =>{
            console.log(instance.exports);
            console.log(instance.exports.add(21, 21));    //42
            console.log(instance.exports.inc(12));        //13
        }
    );
```

控制台将输出：

```
{add: ƒ, inc: ƒ}
42
13
```

3.4.3　创建 **WebAssembly.Instance** 的简洁方法

鉴于在创建实例的过程中，fetch()、compile()等操作会重复出现，我们定义一个 fetchAndInstantiate()方法如下：

```
function fetchAndInstantiate(url, importObject) {
    return fetch(url).then(response =>
        response.arrayBuffer()
    ).then(bytes =>
        WebAssembly.instantiate(bytes, importObject)
    ).then(results =>
        results.instance
    );
}
```

这样只需一行代码即可完成实例的创建，例如：

```
fetchAndInstantiate('test.wasm', importObject).then(function(instance) {
    //do sth. with instance...
})
```

异步社区 www.epubit.com

在后续各节中，为了简化代码，方便阅读，我们将使用这一方法。

3.5 WebAssembly.Memory 对象

与 WebAssembly 内存对应的 JavaScript 对象为 `WebAssembly.Memory`，它用于在 WebAssembly 程序中存储运行时数据。无论从 JavaScript 的角度，还是 WebAssembly 的角度，内存对象本质上都是一个一维数组，JavaScript 和 WebAssembly 可以通过内存相互传递数据。较为常见的用法是：在 JavaScript 中创建内存对象（该对象包含了一个 `ArrayBuffer`，用于存储上述一维数组），模块实例化时将其通过导入对象导入 WebAssembly 中。一个内存对象可以导入多个实例，这使得多个实例可以通过共享一个内存对象的方式交换数据。

3.5.1 WebAssembly.Memory()

WebAssembly 内存对象的构造器方法，语法为：

```
var memory = new WebAssembly.Memory(memDesc);
```

参数

- `memDesc`：新建内存的参数，包含下述属性。
    - ♦ `initial`：内存的初始容量，以页为单位（1 页=64 KB=65 536 字节）。
    - ♦ `maximum`：可选，内存的最大容量，以页为单位。

异常

- 如果传入参数的类型不正确，将抛出 `TypeError`。
- 如果传入的参数包含 `maximum` 属性，但是其值小于 `initial` 属性，将抛出 `RangeError`。

3.5.2 WebAssembly.Memory.prototype.buffer

`buffer` 属性用于访问内存对象的 `ArrayBuffer`。

异步社区 www.epubit.com

例如：

```
//sum.html
    var memory = new WebAssembly.Memory({initial:1, maximum:10});
    fetchAndInstantiate('sum.wasm', {js:{mem:memory}}).then(
        function(instance) {
            var i32 = new Uint32Array(memory.buffer);
            for (var i = 0; i < 32; i++) {
                i32[i] = i;
            }
            console.log(instance.exports.sum(0, 32));    //496
        }
    );
```

上述代码创建了初始容量为 1 页的内存对象并导入实例，从内存的起始处开始依次填入了 32 个 32 位整型数，然后调用了实例导出的 sum() 方法：

```
;;sum.wat
(module
    (import "js" "mem" (memory 1))
    (func (export "sum") (param $offset i32) (param $count i32) (result i32)
        (local $end i32) (local $re i32)
        get_local $offset
        get_local $count
        i32.const 4
        i32.mul
        i32.add
        set_local $end
        block
            loop
                get_local $offset
                get_local $end
                i32.eq
                br_if 1
                get_local $re
                get_local $offset
                i32.load ;;Load i32 from memory:offset
                i32.add
                set_local $re
                get_local $offset
                i32.const 4
                i32.add
```

异步社区 www.epubit.com

```
                    set_local $offset
                    br 0
                end
            end
            get_local $re
        )
    )
```

sum()按照输入参数，从内存中依次取出整型数，计算其总和并返回。

上述例子展示了在 JavaScript 中创建内存导入 WebAssembly 的方法，然而，反向操作，也就是说，在 WebAssembly 中创建内存，导出到 JavaScript，也是可行的。例如：

```
;;export_mem.wat
(module
    (memory $mem 1)                     ;;define $mem, initSize = 1page
    (export "memory" (memory $mem))     ;;$export $mem as "memory"
    (func (export "fibonacci") (param $count i32)
        (local $i i32) (local $a i32) (local $b i32)
        i32.const 0
        i32.const 1
    ...
```

我们在 WebAssembly 中定义了初始容量为 1 页的内存，并将其导出；同时导出了名为 fibonacci()的方法，用于根据输入的数列长度来生成斐波拉契数列。

```
//export_mem.htm
    fetchAndInstantiate('export_mem.wasm').then(
        function(instance) {
            console.log(instance.exports); //{memory: Memory, fibonacci: ƒ}
            console.log(instance.exports.memory); //Memory {}
            console.log(instance.exports.memory.buffer.byteLength); //65536
            instance.exports.fibonacci(10);
            var i32 = new Uint32Array(instance.exports.memory.buffer);
            var s = "";
            for (var i = 0; i < 10; i++) {
                s += i32[i] + ' '
            }
            console.log(s);    //1 1 2 3 5 8 13 21 34 55
        }
    );
```

上述 JavaScript 代码运行后，控制台输出如下：

异步社区 www.epubit.com

```
{memory: Memory, fibonacci: ƒ}
Memory {}
65536
1 1 2 3 5 8 13 21 34 55
```

可见，在 WebAssembly 内部创建的内存被成功导出，调用 `fibonacci()`后在内存中生成的斐波拉契数列亦可正常访问。

值得注意的是，如果 WebAssembly 内部创建了内存，但是在实例化时又导入了内存对象，那么 WebAssembly 会使用内部创建的内存，而不是外部导入的内存。例如：

```
var memory = new WebAssembly.Memory({initial:1, maximum:10});
fetchAndInstantiate('export_mem.wasm', {js:{mem:memory}}).then(
    function(instance) {
        instance.exports.fibonacci(10);
        var i32 = new Uint32Array(instance.exports.memory.buffer);
        var s = "Internal mem:";
        for (var i = 0; i < 10; i++) {
            s += i32[i] + ' '
        }
        console.log(s);    //1 1 2 3 5 8 13 21 34 55

        var i32 = new Uint32Array(memory.buffer);
        s = "Imported mem:";
        for (var i = 0; i < 10; i++) {
            s += i32[i] + ' '
        }
        console.log(s);    //0 0 0 0 0 0 0 0 0 0
    }
);
```

上述代码执行后，控制台将输出：

```
Internal mem:1 1 2 3 5 8 13 21 34 55
Imported mem:0 0 0 0 0 0 0 0 0 0
```

由此可见，WebAssembly 优先使用内部创建的内存。

3.5.3　`WebAssembly.Memory.prototype.grow()`

该方法用于扩大内存对象的容量。语法为：

```
var pre_size = memory.grow(number);
```

参数

- number：内存对象扩大的量，以页为单位。

返回值

- 内存对象扩大前的容量，以页为单位。

异常

- 如果内存对象构造时指定了最大容量，且要扩至的容量（即扩大前的容量加 number）已超过指定的最大容量，则抛出 RangeError。

内存对象扩大后，其扩大前的数据将被复制到扩大后的 buffer 中，例如：

```
//grow.html
    fetchAndInstantiate('export_mem.wasm').then(
        function(instance) {
            instance.exports.fibonacci(10);
            var i32 = new Uint32Array(instance.exports.memory.buffer);
            console.log("mem size before grow():",
                instance.exports.memory.buffer.byteLength); //65536
            var s = "mem content before grow():";
            for (var i = 0; i < 10; i++) {
                s += i32[i] + ' '
            }
            console.log(s); //1 1 2 3 5 8 13 21 34 55

            instance.exports.memory.grow(99);
            i32 = new Uint32Array(instance.exports.memory.buffer);
            console.log("mem size after grow():",
                instance.exports.memory.buffer.byteLength); //6553600
            var s = "mem content after grow():";
            for (var i = 0; i < 10; i++) {
                s += i32[i] + ' '
            }
            console.log(s); //still 1 1 2 3 5 8 13 21 34 55
        }
    );
```

上述程序执行后，控制台将输出：

异步社区 www.epubit.com

```
mem size before grow(): 65536
mem content before grow():1 1 2 3 5 8 13 21 34 55
mem size after grow(): 6553600
mem content after grow():1 1 2 3 5 8 13 21 34 55
```

可见在 JavaScript 看来，数据并未丢失；在 WebAssembly 看来也一样：

```
//grow2.html
    var memory = new WebAssembly.Memory({initial:1, maximum:10});
    fetchAndInstantiate('sum.wasm', {js:{mem:memory}}).then(
        function(instance) {
            var i32 = new Uint32Array(memory.buffer);
            for (var i = 0; i < 32; i++) {
                i32[i] = i;
            }
            console.log("mem size before grow():",
                memory.buffer.byteLength); //65536
            console.log("sum:", instance.exports.sum(0, 32)); //496

            memory.grow(9);
            console.log("mem size before grow():",
                memory.buffer.byteLength); //655360
            console.log("sum:", instance.exports.sum(0, 32)); //still 496
        }
    );
```

上述代码运行后，控制台输出：

```
mem size before grow(): 65536
sum: 496
mem size before grow(): 655360
sum: 496
```

可见在 WebAssembly 看来，`grow()`也不会丢失数据。但值得注意的是，`grow()`有可能引发内存对象的 `ArrayBuffer` 重分配，从而导致引用它的 `TypedArray` 失效。例如：

```
//grow3.html
    fetchAndInstantiate('export_mem.wasm').then(
        function(instance) {
            instance.exports.fibonacci(10);
            var i32 = new Uint32Array(instance.exports.memory.buffer);
```

```
                var s = "i32 content before grow():";
                for (var i = 0; i < 10; i++) {
                    s += i32[i] + ' '
                }
                console.log(s); //1 1 2 3 5 8 13 21 34 55

                instance.exports.memory.grow(99);
                //i32 = new Uint32Array(instance.exports.memory.buffer);
                var s = "i32 content after grow():";
                for (var i = 0; i < 10; i++) {
                    s += i32[i] + ' '
                }
                console.log(s); //undefined...
            }
        );
```

上述代码注释掉了第二条 i32 = new Uint32Array(instance.exports.memory.buffer);，由于 grow()后 Memory.buffer 重分配，导致之前创建的 i32 失效，控制台输出如下：

```
    i32 content before grow():1 1 2 3 5 8 13 21 34 55
    i32 content after grow():undefined undefined undefined undefined undefined
undefined undefined undefined undefined undefined
```

这意味着通过 TypedArray 读写 Memory.buffer 时，必须随用随创建。

如果内存对象在创建时指定了最大容量，则使用 grow()扩容时不能超过最大容量值。例如，下列代码第二次 grow()时将抛出 RangeError：

```
var memory = new WebAssembly.Memory({initial:1, maximum:10});
//...
memory.grow(9);  //ok, current size = 10 pages
memory.grow(1);  //RangeError
```

3.6 WebAssembly.Table 对象

通过前面的介绍我们不难发现，在 WebAssembly 的设计思想中，与执行过程相关的代码段/栈等元素与内存是完全分离的，这与通常体系结构中代码段/数据段/堆/栈全都处于统一编址内存空间的情况完全不一样，函数地址对 WebAssembly 程序来说不可见，更遑论将其当作变量一样传递、修改以及调用了。然而函数指针对很多高级语言来说是必

异步社区 www.epubit.com

不可少的特性，如回调、C++的虚函数都依赖于它。解决这一问题的关键，就是本节将要介绍的表格对象。

表格是保存了对象引用的一维数组。目前可以保存在表格中的元素只有函数引用一种类型，随着 WebAssembly 的发展，将来或许有更多类型的元素（如 DOM 对象）能被存入其中，但到目前为止，可以说表格是专为函数指针而生。目前每个实例只能包含一个表格，因此相关的 WebAssembly 指令隐含的操作对象均为当前实例拥有的唯一表格。表格不占用内存地址空间，二者是相互独立的。

使用函数指针的本质行为是：通过变量（即函数地址）找到并执行函数。在 WebAssembly 中，当一个函数被存入表格中后，即可通过它在表格中的索引（该函数在表格中的位置，或者说数组下标）来调用它，这就间接地实现了函数指针的功能，只不过用来寻找函数的变量不是函数地址，而是它在表格中的索引。

WebAssembly 为何使用这种拐弯抹角的方式来实现函数指针？最重要的原因是为了安全。倘若能通过函数的真实地址来调用它，那么 WebAssembly 代码的执行范围将不可控，例如，调用非法地址导致浏览器崩溃，甚至下载恶意程序后导入运行等，而在 WebAssembly 当前的设计框架下，保存在表格中的函数地址对 WebAssembly 代码不可见、无法修改，只能通过表格索引来调用，并且运行时的栈数据并不保存在内存对象中，由此彻底断绝了 WebAssembly 代码越界执行的可能，最糟糕的情况不过是在内存对象中产生一堆错误数据而已。

与 WebAssembly 表格对应的 JavaScript 对象为 `WebAssembly.Table`。

异步社区 www.epubit.com

3.6.1 WebAssembly.Table()

表格的构造器方法，语法为：

```
var table = new WebAssembly.Table(tableDesc);
```

参数

- `tableDesc`：新建表格的参数，包含下述属性。
  - `element`：存入表格中的元素的类型，当前只能为 `anyfunc`，即函数引用。
  - `initial`：表格的初始容量。
  - `maximum`：可选，表格的最大容量。

“表格的最大容量”指表格能容纳的函数索引的个数（即数组长度），这与内存对象的容量以页为单位不同，注意区分。

异常

- 如果传入参数的类型不正确，将抛出 TypeError。
- 如果传入的参数包含 maximum 属性，但是其值小于 initial 属性，将抛出 RangeError。

3.6.2 WebAssembly.Table.prototype.get()

该方法用于获取表格中指定索引位置的函数引用，语法为：

```
var funcRef = table.get(index);
```

参数

- index：欲获取的函数引用的索引。

返回值

- WebAssembly 函数的引用。

异常

- 如果 index 大于等于表格当前的容量，抛出 RangeError。

示例

```
//import_table.html
    var table = new WebAssembly.Table({element:'anyfunc', initial:2});
    console.log(table);
    console.log(table.get(0));
    console.log(table.get(1));
    fetchAndInstantiate('import_table.wasm', {js:{table:table}}).then(
        function(instance) {
            console.log(table.get(0));
            console.log(table.get(1));
        }
    );

;;import_table.wat
(module
```

异步社区 www.epubit.com

```
        (import "js" "table" (table 2 anyfunc))
        (elem (i32.const 0) $func1 $func0) ;;set $func0,$func1 to table
        (func $func0 (result i32)
            i32.const 13
        )
        (func $func1 (result i32)
            i32.const 42
        )
    )
```

我们在 JavaScript 中创建了初始容量为 2 的表格并为其导入实例，在 WebAssembly 的 elem 段将 func1、func0 存入表格中（刻意颠倒了顺序）。上述程序执行后控制台输出如下：

```
Table {}
null
null
ƒ 1() { [native code] }
ƒ 0() { [native code] }
```

table.get()的返回值类型与 Instance 导出的函数是一样的，这意味着我们可以调用它。将上述例子略为修改：

```
//import_table2.html
    var table = new WebAssembly.Table({element:'anyfunc', initial:2});
    fetchAndInstantiate('import_table.wasm', {js:{table:table}}).then(
        function(instance) {
            var f0 = table.get(0);
            console.log(f0());
            console.log(table.get(1)());
        }
    );
```

运行后控制台输出如下：

```
42
13
```

> **提示** 在上一个例子中，func0 和 func1 并未导出，而将其存入表格后，JavaScript 依然可以通过表格对其进行调用。

异步社区 www.epubit.com

3.6.3 **WebAssembly.Table.prototype.length**

length 属性用于获取表格的当前容量。

与内存对象类似，表格既可以在 JavaScript 中创建后被导入 WebAssembly，亦可在 WebAssembly 中创建后导出到 JavaScript，并且优先使用模块内部创建的表格。例如：

```
;;export_table.wat
(module
    (table $tab 2 anyfunc)                  ;;define $tab, initSize = 2
    (export "table" (table $tab))          ;;export $tab as "table
    (elem (i32.const 0) $func1 $func0)      ;;set $func0,$func1 to table
    (func $func0 (result i32)
        i32.const 13
    )
    (func $func1 (result i32)
        i32.const 42
    )
)
//export_table.html
    var table = new WebAssembly.Table({element:'anyfunc', initial:1});
    fetchAndInstantiate('export_table.wasm', {js:{table:table}}).then(
        function(instance) {
            console.log(table.get(0)); //null

            console.log(instance.exports);
            console.log('instance.exports.table.length:'
                + instance.exports.table.length);
            for (var i = 0; i < instance.exports.table.length; i++){
                console.log(instance.exports.table.get(i));
                console.log(instance.exports.table.get(i)());
            }
        }
    );
```

程序执行后控制台将输出：

```
null
{table: Table}
instance.exports.table.length:2
```

```
ƒ 1() { [native code] }
42
ƒ 0() { [native code] }
13
```

3.6.4 在 WebAssembly 内部使用表格

前面的例子展示的是表格中的函数被 JavaScript 调用，然而表格更主要的作用还是被 WebAssembly 调用，例如：

```
;;call_by_index.wat
(module
    (import "js" "table" (table 2 anyfunc))
    (elem (i32.const 0) $plus13 $plus42)       ;;set $plus13,$plus42 to table
    (type $type_0 (func (param i32) (result i32)))  ;;define func Signatures
    (func $plus13 (param $i i32) (result i32)
        i32.const 13
        get_local $i
        i32.add
    )
    (func $plus42 (param $i i32) (result i32)
        i32.const 42
        get_local $i
        i32.add
    )
    (func (export "call_by_index")(param $id i32)(param $input i32)(result i32)
        get_local $input             ;;push param into stack
        get_local $id                ;;push id into stack
        call_indirect (type $type_0) ;;call table:id
    )
)
```

WebAssembly 代码定义了两个函数，即 `plus13()`以及 `plus42()`，并将其分别存入表格的 0 和 1 处。

`(type $type_0 (func (param i32) (result i32)))`定义了将要被调用的函数的签名(即函数的参数列表及返回值类型)——根据栈式虚拟机的特性，WebAssembly 在执行函数调用时，调用方与被调用方需要严格匹配签名；若签名不匹配，会抛出 `WebAssembly.RuntimeError`。

> **提示**　WebAssembly 汇编语言的相关细节将在第 4 章详细介绍。

导出函数 `call_by_index()` 调用表格中的第`$id` 个函数并返回。

按下列方法在 JavaScript 中调用后：

```
//call_by_index.html
    var table = new WebAssembly.Table({element:'anyfunc', initial:2});
    fetchAndInstantiate('call_by_index.wasm', {js:{table:table}}).then(
        function(instance) {
            console.log(instance.exports.call_by_index(0, 10));
            console.log(instance.exports.call_by_index(1, 10));
        }
    );
```

控制台将输出：

```
23
52
```

在 WebAssembly 中使用 `call_indirect` 时，如果试图调用索引范围超过表格容量的函数，将抛出 `WebAssembly.RuntimeError`，如在上述例子中，表格的容量为 2，如果我们执行下列操作：

```
var table = new WebAssembly.Table({element:'anyfunc', initial:2});
fetchAndInstantiate('call_by_index.wasm', {js:{table:table}}).then(
    function(instance) {
        instance.exports.call_by_index(2, 10); //WebAssembly.RuntimeError
    }
);
```

控制台输出如图 3-2 所示。

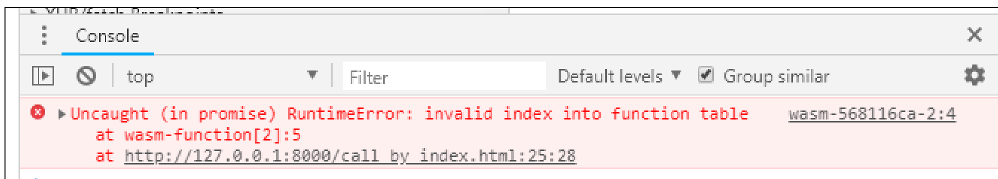

图 3-2

异步社区 www.epubit.com

3.6.5 多个实例通过共享表格及内存协同工作

与内存对象类似，一个表格也可以被导入多个实例从而被多个实例共享。接下来，我们以一个相对复杂的例子来展示多个实例共享内存和表格协同工作。在这个例子中，我们将载入两个模块并各自创建一个实例，其中一个将在内存中生成斐波拉契数列的函数，另一个将调用前者，计算数列的和并输出。

计算斐波拉契数列的模块如下：

```
;;fibonacii.wat
(module
    (import "js" "mem" (memory 1))
    (import "js" "table" (table 1 anyfunc))
    (elem (i32.const 0) $fibonacci)    ;;set $fibonacci to table:0
    (func $fibonacci (param $count i32)
        (local $i i32) (local $a i32) (local $b i32)
        i32.const 0
        i32.const 1
        i32.store
        i32.const 4
        i32.const 1
        i32.store
        i32.const 1
        set_local $a
        i32.const 1
        set_local $b
        i32.const 8
        set_local $i
        get_local $count
        i32.const 4
        i32.mul
        set_local $count
        block
            loop
                get_local $i
                get_local $count
                i32.ge_s
                br_if 1
                get_local $a
```

异步社区 www.epubit.com

```
                get_local $b
                i32.add
                set_local $b
                get_local $b
                get_local $a
                i32.sub
                set_local $a
                get_local $i
                get_local $b
                i32.store
                get_local $i
                i32.const 4
                i32.add
                set_local $i
                br 0
            end
        end
    )
)
```

上述代码定义了计算斐波拉契数列的函数 fibonacci()，并将其引用存入表格的索引 0 处。

求和模块如下：

```
;;sumfib.wat
(module
    (import "js" "mem" (memory 1))
    (import "js" "table" (table 1 anyfunc))
    (type $type_0 (func (param i32))) ;;define func Signatures
    (func (export "sumfib") (param $count i32) (result i32)
     (local $offset i32)(local $end i32)(local $re i32)
        ;;call table:element 0:
        (call_indirect (type $type_0) (get_local $count) (i32.const 0))
        i32.const 0
        set_local $offset
        get_local $count
        i32.const 4
        i32.mul
        set_local $end
        block
            loop
                get_local $offset
```

异步社区 www.epubit.com

```
                get_local $end
                i32.eq
                br_if 1
                get_local $re
                get_local $offset
                i32.load ;;Load i32 from memory:offset
                i32.add
                set_local $re
                get_local $offset
                i32.const 4
                i32.add
                set_local $offset
                br 0
            end
        end
        get_local $re
    )
)
```

(call_indirect (type $type_0) (get_local $count) (i32.const 0))按照函数签名将参数$count 压入了栈，然后通过 call_indirect 调用了表格中索引 0 处的函数，后续的代码计算并返回了数列的和。

```
//sum_fibonacci.html
    var memory = new WebAssembly.Memory({initial:1, maximum:10});
    var table = new WebAssembly.Table({element:'anyfunc', initial:2});
    Promise.all([
        fetchAndInstantiate('sumfib.wasm', {js:{table:table, mem:memory}}),
        fetchAndInstantiate('fibonacci.wasm', {js:{table:table, mem:memory}})
    ]).then(function(results) {
        console.log(results[0].exports.sumfib(10));
    });
```

不出意外，上述程序按照我们设计的路径（JavaScript->sumfib.wasm.sumfib->table:0->fibonacci.wasm.fibonacci->sumfib.wasm.sumfib->return）正确执行，并输出了斐波拉契数列前 10 项的和：

```
143
```

刚才的例子中都是在 WebAssembly 中修改表格，事实上，运行时我们还可以在 JavaScript 中修改它。

异步社区 www.epubit.com

3.6.6 WebAssembly.Table.prototype.set()

该方法用于将一个 WebAssembly 函数的引用存入表格的指定索引处，语法为：

```
table.set(index, value);
```

参数

- index：表格索引。
- value：函数引用，函数指的是实例导出的函数或保存在表格中的函数。

异常

- 如果 index 大于等于表格当前的容量，抛出 RangeError。
- 如果 value 为 null，或者不是合法的函数引用，抛出 TypeError。

把刚才的斐波拉契数列求和例子略为修改：

```
;;fibonacci2.wat
(module
    (import "js" "mem" (memory 1))
    (import "js" "table" (table 1 anyfunc))
    (func (export "fibonacci") (param $count i32)
    ...

//sum_fibonacci2.html
    var memory = new WebAssembly.Memory({initial:1, maximum:10});
    var table = new WebAssembly.Table({element:'anyfunc', initial:2});
    Promise.all([
        fetchAndInstantiate('sumfib.wasm', {js:{table:table, mem:memory}}),
        fetchAndInstantiate('fibonacci2.wasm', {js:{table:table, mem:memory}})
    ]).then(function(results) {
        console.log(results[1].exports);
        table.set(0, results[1].exports.fibonacci);
        console.log(results[0].exports.sumfib(10));
    });
```

也就是说，fibonacci()函数是在 JavaScript 中动态导入表格的。程序运行后，结果依然：

异步社区 www.epubit.com

```
{fibonacci: ƒ}
143
```

表格和内存的跨实例共享，加上表格在运行时可变，使 WebAssembly 可以实现非常复杂的动态链接。

3.6.7 WebAssembly.Table.prototype.grow()

该方法用于扩大表格的容量，语法为：

```
preSize = table.grow(number);
```

参数

- number：表格扩大的量。

返回值

- 表格扩大前的容量。

异常

- 如果表格构造时指定了最大容量，且要扩至的容量（即扩大前的容量加 number）已超过指定的最大容量，则抛出 RangeError。

与内存类似，表格的 grow() 方法不会丢失扩容前的数据。鉴于相关的验证方法和例子与内存高度相似，在此不再重复。

3.7 小结及错误类型

除了常规的 TypeError 和 RangeError，与 WebAssembly 相关的异常还有另外 3 种，即 WebAssembly.CompileError、WebAssembly.LinkError 和 WebAssembly.RuntimeError。这 3 种异常正好对应 WebAssembly 程序生命周期的 3 个阶段，即编译、链接（实例化）和运行。

编译阶段的主要任务是将 WebAssembly 二进制代码编译为模块（如何转码取决于虚拟机实现）。在此过程中，若 WebAssembly 二进制代码的合法性无法通过检查，则抛出

异步社区 www.epubit.com

WebAssembly.CompileError。

在链接阶段，将创建实例，并链接导入/导出对象。导致该阶段抛出 WebAssembly.LinkError 异常的典型情况是导入对象不完整，导入对象中缺失了模块需要导入的函数、内存或表格。另外，在实例初始化时，可能会执行内存/表格的初始化，如 2.4 节的例子中使用 data 段初始化内存：

```
(data (i32.const 0) "你好，WASM")
```

以及 3.6 节的例子中使用 elem 段初始化表格：

```
;;import_table.wat
(module
    (import "js" "table" (table 2 anyfunc))
    (elem (i32.const 0) $func1 $func0) ;;set $func0,$func1 to table
    ...
)
```

在此过程中，若内存/表格的容量不足以容纳装入的数据/函数引用，也会导致 WebAssembly.LinkError。

运行时抛出 WebAssembly.RuntimeError 的情况较多，常见的主要有以下几个。

（1）内存访问越界。试图读写超过内存当前容量的地址空间。

（2）调用函数时，调用方与被调用方签名不匹配。

（3）表格访问越界。试图调用/修改大于等于表格容量的索引处的函数。

在运行时产生的异常中有一种情况需要特别注意：目前 JavaScript 的 Number 类型无法无损地表达 64 位整型数。这意味着虽然 WebAssembly 支持 i64 类型的运算，但是与 JavaScript 对接的导出函数不能使用 i64 类型作为参数或返回值，一旦在 JavaScript 中调用参数或返回值类型为 i64 的 WebAssembly 函数，将抛出 TypeError。

下面给出了一些异常的例子。WebAssembly 代码如下：

```
;;exceptions.wat
(module
    (import "js" "mem" (memory 1))
    (import "js" "table" (table 2 anyfunc))
    (elem (i32.const 0) $f0 $f1)
    (type $type_0 (func (result i32)))
    (func $f0 (param i32)(result i32)
        i32.const 13
    )
```

异步社区 www.epubit.com

```
    (func $f1 (result i32)
        i32.const 65540
        i32.load
    )
    (func (export "call_by_index") (param $index i32)(result i32)
        get_local $index
        call_indirect (type $type_0)
    )
    (func (export "return_i64") (result i64)
        i64.const 0
    )
    (func (export "param_i64") (param i64))
)
```

JavaScript 代码如下：

```
//exceptions.html
    var table = new WebAssembly.Table({element:'anyfunc', initial:3});
    var memory = new WebAssembly.Memory({initial:1});
    fetchAndInstantiate('exceptions.wasm', {js:{table:table, mem:memory}}).then(
        function(instance) {
            try{
                instance.exports.call_by_index(0)
            }
            catch(err){
                console.log(err);
            }
            try{
                instance.exports.call_by_index(1)
            }
            catch(err){
                console.log(err);
            }
            try{
                instance.exports.call_by_index(2)
            }
            catch(err){
                console.log(err);
            }
            try{
                instance.exports.call_by_index(3)
            }
```

异步社区 www.epubit.com

```
            catch(err){
                console.log(err);
            }
            try{
                instance.exports.return_i64()
            }
            catch(err){
                console.log(err);
            }
            try{
                instance.exports.param_i64(0)
            }
            catch(err){
                console.log(err);
            }
        }
    );
```

程序运行后，捕获异常如图 3-3 所示。

图 3-3

异步社区 www.epubit.com