

*Electron实战*

史蒂文·金尼 著

杜冬斌 译

Steve Kinney

·金尼

*Electron 实战*

史蒂文·金尼 著

杜冬斌 译

MANNIN G

SHELTER ISLAND

有关本书和其他曼宁书籍的在线信息和订购，请访问www.manning.com。出版商在大量订购时会提供此书的折扣。获取更多资讯，请联系

Special Sales Department Manning Publications Co. 20 Baldwin Road

PO Box 761

Shelter Island, NY 11964 Email: [orders@manning.com](mailto:orders@manning.com)

©2019 by Manning Publications Co.保留所有权利。

未经出版商事先书面许可，不得以任何形式或通过Electron，机械，影印或其他方式复制，存储于检索系统或传播本出版物的任何部分。

制造商和销售商用于区分其产品的许多名称都是商标。如果这些名称出现在书中，并且Manning Publications知道商标声明，则这些名称已经以初始上限或全部大写字母印刷。

认识到保存所写内容的重要性，曼宁的政策是将我们出版的书籍印在无酸纸上，我们尽最大努力实现这一目标。同时认识到我们有责任保护地球资源，曼宁书籍印在纸上，至少15％的回收和加工，不使用元素氯。

 Manning Publications Co. Development editor: Helen Stergius 20 Baldwin Road Technical development editor: Nickie Buckner

PO Box 761 Review editor: Aleksandar Dragosavljevic´

Shelter Island, NY 11964 Project editor: Lori Weidert Copy editor: Pamela Hunt Proofreader: Elizabeth Martin

Technical proofreader: Doug Warren

Typesetter: Dennis Dalinnik Cover designer: Marija Tudor

ISBN: 9781617294143

在美利坚合众国印刷

1 2 3 4 5 6 7 8 9 10 – DP – 23 22 21 20 19 18

*这本书献给我的妻子洛根和我的儿子，韦斯和杰克。你是我生命中的爱。*

# *译序*

我是一名IT软件开发的从业者,对主流的开发语言广泛涉猎:Java/Python/C++/JavaScript/TypeScript/SQL等等.

毕业后从事约3年的运维,后转做软件开发类至今,精通JavaEE后端的开发,做过JavaEE的全栈讲师和技术总监

现在从事的工作主要涉及到Java的微服务,分布式docker集群,JavaEE前后端分离实现,基于Angular和Ionic移动快平台应用开发,微信小程序开发等

代码写的久了,就想找一些能够实现跨平台或者全平台的框架,其实JavaWeb是可以实现基于浏览器的全平台访问,但是很多本机的原生功能是受限制的做不了

在桌面端应用方面寻求过:以下仅是个人观点

A.Java技术

Java技术实现桌面应用的前景堪忧,Oracle官方都把包袱甩出去了,即便不甩出去,Java实现因为还有VM虚拟机的性能损耗,个人觉的不是最佳选择,研究一段时间后放弃了;

B.C++技术

对于C++技术的实现,这是目前主流的做法,在C++这一块还有一个QT开发桌面应用使用的人非常多,但是学习成本比较高,要先学习C++,再学习QT,然而学习C++会让很多初学者望而却步,

即便学会了C++及QT的语法规则,因为经验不足的问题,很可能导致内存泄露的严重问题!

C.Python技术

对于Python的技术实现,主要是个人忍受不了代码缩进的语法;

苦苦找寻,尝试都没有找到让自己满意的,终于在阅读<<NodeJS实战>>时,得知还有Electron可以写跨平台的桌面应用,而且基于最基本三种Web技术(Html,JavaScript,CSS),欣喜若狂,

然而却发现,此框架,目前国内市面上没有一本介绍的书,能参考的也只有Electron官网的文档了,但是总感觉没有人引导,肯定会走很多的弯路,

于是使用各种搜索引擎,国内外各种搜索,终于发现在国外已经有人写了Electron实战的英文版,可是国内却没有对应的中文版,

脑海中突发奇想,不如自己尝试翻译!

翻译过程:

将原书的PDF转换为可编辑的 Word版本;

利用自己的知识范围,尝试将英文翻译为中文版,

--特专业术语自己不是很熟悉的求助谷歌翻译,尽管如此,有些地方可能还是有点生硬

通篇阅读自己翻译的草稿,检查明显不通顺的地方,酌情修改;

--力争体现原作者的意图,在不曲解作者意图的前提下,修改为适合国人阅读的方式,因此有些地方,翻译可能和原文翻译过来有些差异.

执行代码再一次阅读审稿,防止出现代码里的变量名,常量名方法名等被翻译;这种情况下读者可能会感觉明显不顺畅.

标注处理:

原书中很多标注说明都是箭头指示,但是在Word中翻译,很容易走位导致指向位置不对,因此在不引起误会的情况下,我会酌情将一些箭头指示修改为代码注释,以保证读者能看到作者的注释说明. **关键处的代码会粗体显示,以突出视觉效果!**

代码验证:

本书的所有代码,几乎都会经过三大主流PC操作系统平台的验证,

都是截止验证代码时的最新版:

Electron:5.0.2,

Node: 10.15.3 长期支持版

Windows 10 最新更新版

macOS 10.14.5

Deepin Linux 15.10.1

作者几乎是手把手教读者写代码了,译者基本上是沿着作者的一步步代码实现的!

本书优先验证作者的代码在新版本下的运行状态,遇到问题以及如何适配到新版本中,作者的版本作者自己已经验证过了,可以略去;

当然本书,代码也会在最新的Electron框架版本下运行,是否异常,以及如何处理,我都会以译注的方式说明,

虽然本书的Electron的版本不是最新的,但是我将尽可能提供在最新版本下运行遇到的一些问题及如何处理;毕竟是技术书籍,我将尽可能减少读者学习第二遍(新版本不同有差异)的问题.

艰辛之处:

主要是时间,其次是代码验证以及出现的异常问题处理,最后是排版问题.

翻译校对:

这是译者第一次翻译一本英文技术书,经验不足,翻译不准确肯定是有的,所以读者可以通过Email地址(dusuanyun@sina.com)向我提出校对建议,我将不胜感激!因为这不仅对我个人有很大的帮助,也会帮助更多的读者.

关于排版:

本人对Word的排版,不是很擅长,非常欢迎有排版经验的朋友帮助一起完善!

20190717 合肥 杜冬斌

# *前言*

Electron是我现在最兴奋的技术之一。当你阅读Electron实战时，我希望你能够接受这种兴奋。正如我将在本书中解释的那样，Electron允许Web开发人员创建具有浏览器中不可用功能的桌面应用程序。它允许您为我们的命令行工具创建图形用户界面，向可能不熟悉终端的更广泛的受众开放我们的创作。Electron使您能够使用您无法构建的Web技术构建应用程序。

Electron产生了一个在开源中很少见的甜蜜点。它的低级别足以让您快速掌握基础知识，并且功能强大，足以让您构建极其复杂的应用程序。它描绘了构建桌面应用程序时需要做的一些更乏味的事情，同时不会陷入依赖太多黑魔法的陷阱。该平台由热情的社区提供支持，该社区提供的库可帮助您轻松完成各种功能。

您可能听说过通过Atom，Slack，Visual Studio Code或使用它的任何其他大牌应用程序的Electron。但是我为爱好者或独立开发者写了这本书，他想要创造一些原创和新的东西。Electron在大型团队中很受欢迎，但对于想要构建只有他们可能使用的应用程序的单个开发人员或者需要构建内部使用工具的小团队来说，它也很有用。

这是一种工具，当您熟悉它时，它开辟了新的途径，否则是不可能的。当你对Electron感到满意时，它看起来很酷，但最初可能很难想出一个用例。让它沉入其中，你很快就会发现自己走在街上，提出你可以建立的应用程序的想法。

这就是我发生的事情，我已经让Electron教给了足够多的人来强烈怀疑这对你的影响。Electron实战就像我在美国和哥伦比亚旅行一样，教授有关使用Electron构建跨平台桌面应用程序的研讨会。曼宁邀请我写一本关于这个主题的书，我抓住了这个机会。这本书告知了研讨会，并提供了研讨会给了我新的见解，帮助改进了这本书。

如果留给我自己的设备，我会永远精炼这本书。它帮助我阐明了自己构建Electron应用程序的想法，并在我处理项目中的新挑战和实现功能时成为了一份工作日记。这是一个我会想念的良性循环，尽管如此，我很乐意让我的夜晚和周末回来。

# *感谢*

最让我惊讶的一件事就是写一本书是多少工作。其次是有多少人参与实现这一目标。首先，我要感谢我的妻子洛根,我本应该和家人一起度过,她却能容忍我在晚上和周末在这本书上的工作。她的宽容和支持对于制作这本书至关重要。

感谢海伦·斯特吉斯（Helen Stergius），因为我平衡了我的家庭，我的日常工作和这本书，所以他们提出了最后期限。无论我在任何特定时刻有多紧张，海伦都保持着积极的态度。感谢Nickie Buckner在编写本书的过程中完成了代码，提供了鼓励，并修复了我的拼写错误。感谢Doug Warren在本书接近完成时做了最后的技术准证。感谢Brian Sawyer伸出手来邀请我写这本书，以及Marjan Bace，他点亮了这个项目。

感谢Frontend Masters的Marc Grabanski让我在全球观众面前讨论这些内容并提供富有洞察力的反馈。感谢Jeff Casimir为我提供了一个平台，教育无数新兴软件工程师。感谢Meeka Gayhart，Louisa Barrett，Jhun de Andreas，Brenna Martenson和Brittany Storoz对我的蔑视，因为我把球放在他们依赖我的各种事情上，以及总体上忍受我。

感谢以下评论者，他们正在阅读本书，并在论坛中留下反馈：Aiden Mark Humphreys，Alan Bogusiewicz，Alexey Galiulin，Anto Aravinth，Ashwin Raj，Buu Nguyen，Daniel Posey，Frederic Flayol，Harald Kuhn，Hari Khalsa，Iain Shigeoka，Jay Kelkar，Jim McGinn，Jimmy Qiu，Jon Riddle，Matteo Gildone，MladenÐuric'，PhilippeCharrière，Raq Khan和William Wheeler。你帮助我改进内容并在整个过程中发现错误。

感谢Cheng Zhao和所有维护Electron的人。没有你所有令人难以置信的工作，这本书就不存在了。此外，您对细节和用户优先心态的认真关注使我能够轻松解释如何实现在其他平台上可能难以实现的功能。当您阅读Electron实战时，我会多次尝试使用Electron内置的一些API来解决棘手的问题。感谢精彩的社区，它提供了第三方库的生态系统，以帮助Electron应用程序。在极少数情况下，Electron几遍无法开箱即用，也总有一个库可以为您解决问题。平台与周围的社区一样好。

最后，感谢丹佛的Novo Coffee和他们的冷饮给我一个写作的地方，以及要求说出来的咖啡因。

# *关于本书*

Electron实战的主要目标是让您快速开始构建Electron应用程序。当我们将它们用于代码实践时，我们通过学习它们来探索许多基础概念。本书不仅向您介绍Electron的基础知识，还为您提供自己应用的灵感和想法。

### *谁应该读这本书*

本书适用于任何想要构建违反浏览器限制的应用程序的人。对于那些想要抓住自己的痒并构建桌面应用程序而不必学习新的编程语言或框架的人来说，它是一本非常合适的书。这本书适合小型团队超越自身体重，并通过一个代码库提供在多个操作系统上运行的应用程序。本书最贴心的是，本书适用于任何想要使用命令行应用程序并提供GUI或删除用户在其计算机上安装了Node.js以便使用其应用程序或工具的要求的人。

我假设您熟悉JavaScript，但会引导您完成可能对您不熟悉的Web平台或Node.js的任何部分，因为您可能只具有其中一个领域的经验，具体取决于您的背景。

### *路线图*

本书分为十六章。确实很多章节都是从最后一个章节开始的地方继续下去的，但我希望如果您只是想在应用程序中实现特定功能，那么您将直接阅读这些章节。

在第1章中，我们将介绍Electron是什么，以及它不是什么。我们将介绍一些你可以用Electron做的事情，并且是你无法单独使用浏览器或Node.js完成的。

在第2章中，我们从一个非常简单的Electron应用程序开始。这里的目标是弄脏我们的手(译注:作者很幽默)，并证明开始使用Electron很容易。

第3章向您介绍了本书的主要应用之一：Fire Sale，它是一个允许用户在其文件系统上打开Markdown文件并进行编辑的应用程序。

在第4章中，我们使用原生系统对话框和警告，允许用户从文件系统中选择一个文件，以便在Fire Sale中进行编辑。该应用程序将模糊DOM和Node的标准库之间的界限，在两者之间进行协调以实现此功能。

在第5章中，我们将为您的应用程序添加多窗口支持，它引入了一系列您不习惯在单个浏览器选项卡或Node.js中处理的挑战，其中没有任何窗口要说的。

第6章进一步集成到原生操作系统中。我们将Fire Sale中打开的文档附加到操作系统最近打开的文件列表中，设置监听器以查看其他应用程序是否更改了您打开的文件的内容，并根据是否或更新窗口的标题栏,基于 该文件是否没未保存的更改。

第7章探讨了构建原生应用程序的技术，这些应用程序在桌面应用程序的所有窗口和上下文菜单中共享，在应用程序中单击鼠标右键可以使用。

在第8章中，我们将研究如何根据应用程序启用和禁用菜单项的状态来更新应用程序菜单。

在第9章中，我们切换齿轮并创建一种新类型的应用程序，一种位于macOS上的菜单栏或Windows中的系统托盘中。这不是我们用于构建Web应用程序的地方。在本章中，您将构建Clipmaster，它是一个小型剪贴板管理器，可以读取和写入系统剪贴板，响应全局热键并显示通知。

第10章提高了赌注，并使用第三方库创建了一个具有基于DOM的UI的Clipmaster版本,就像Fire Sale一样。据称，Clipmaster 9000能够访问GitHub的Gist API并通过一次击键发布剪辑。

直到第11章，我们一直在使用vanilla JavaScript来实现Electron应用程序中的功能。在本章中，我将向您展示在您的应用程序中使用像Babel，TypeScript和CoffeeScript这样的编译到JavaScript工具以及使用Sass和Less进行样式化是多么容易。在本章中，您将使用React构建一个名为Jetsetter的装箱单应用程序。

在第12章中，我们将讨论除了写入文件系统之外的持久化数据的策略。我将演示如何设置一个SQLite数据库，您可以从客户端代码中读取和写入。然后，我们将使用基于浏览器的IndexedDB对问题进行第二次调整。

第13章介绍了Spectron，它允许您为Electron应用程序编写Selenium测试。我们将为之前创建的Clipmaster 9000应用程序编写一组测试。

在第14章中，我们将介绍一些工具，这些工具可以帮助我们将Electron应用程序打包，以便分发给那些对使用命令行启动应用程序不感兴趣的用户--几乎每个人都不是开发人员，坦率地说，很多开发人员同样如此。

第15章介绍了如何为macOS应用程序签名，为Windows创建安装程序以及设置简单服务器以收集错误日志和崩溃报告。

在第16章中，我将逐步完成将您的应用程序放入Mac应用商店的过程。如果您希望自己发布应用程序，这不是必需的步骤，但如果您没有Apple的流程经验，这肯定是有用的。

### *关于代码*

本书包含编号清单中的源代码的许多示例，并与普通文本一致。在这两种情况下，源代码都以这样的固定宽度字体格式化，以将其与普通文本分开。有时代码也是粗体到高亮代码，这些代码已经从本章前面的步骤发生了变化，例如当新功能添加到现有代码行时。

在许多情况下，原始源代码已经重新格式化; 我们添加了换行符和重写缩进以适应书中可用的页面空间。在极少数情况下，即使这还不够，列表包括行继续标记（➥）。另外，当文本中描述代码时，源代码中的注释通常已从列表中删除。代码注释伴随着许多列表，突出了重要的概念。

本书的所有代码均可从出版商的网站www.manning.com/books/electron-in-action以及Github上获取，网址为https://github.com/electron-in-action。在大多数情况下，每章都有一个分支。对于我们从本书前面提到应用程序的后续章节中的一些章节，我在本章开头提供了一个分支作为起点，也为我们在本章末尾留下代码的地方提供了一个分支章节。如果章节的最终代码很短，我已将其包含在本章末尾。有关较长示例的章节的代码可以在本书的附录中找到。在2018年5月，GitHub宣布了一个网络服务和npm软件包，可以轻松实现使用GitHub发布（https://electronjs.org/blog/autoupdating-electronic-apps）发布的开源Electron应用程序的自动更新。如果您的应用程序符合这些条件，您可以考虑使用update-electron-app。第15章介绍了在您不能或不想使用update-electron-app的情况下如何推送自己的解决方案。

编写一本书最可怕的部分之一是Electron，Node.js或Chromium的新版本 --甚至是次要版本--可能会破坏其中一个例子。当我写这本书时，这种情况不止一次发生。

我承诺保持这些代码是最新的，并将在该章的分支上的README.md中提供任何勘误。如果某些内容在本书中无法正常工作，请务必查看GitHub上的存储库或查看本书的论坛。

### *书籍论坛*

购买Electron实战包括免费访问由Manning出版社运营的私人网络论坛，您可以在该论坛上对该书进行评论，提出技术问题，并从作者和其他用户那里获得帮助。要访问论坛，请访问https://forums.manning.com/forums/electron-in-action。您还可以访问https://forums.manning.com/forums/about，了解有关Manning论坛和行为规则的更多信息。

曼宁对读者的承诺是提供一个平台，让读者之间以及读者与作者之间进行有意义的对话。这并不是对作者任何具体参与的承诺，作者对论坛的贡献仍然是自愿的（而且是无偿的）。我们建议您尝试向作者提出一些具有挑战性的问题，以免他的兴趣流浪！ 只要本书出版，论坛和之前讨论的档案就可以从出版商的网站上获取。

### *关于作者*

STEVE KINNEY是SendGrid的首席工程师，他是国际演讲者，也是DinosaurJS的组织者，他是科罗拉多州丹佛市的一个JavaScript会议。之前，他是图灵软件与设计学院前端工程项目的创始主任。并且是纽约市的一位七年老师。

# *关于封面插图*

Electron实战封面上的数字标题为“吉普赛女人”。这个插图取自1802年1月1日威廉米勒出版的奥斯曼帝国的一系列简单和盛大的服装。伦敦老邦德街。该集合中缺少标题页，到目前为止我们无法追踪它。这本书的目录标识了英语和法语的数字，每个插图还带有两位艺术家的名字，他们无疑会惊讶地发现他们的艺术在200年后的电脑编程书的封面上熠熠生辉。

该系列由Manning编辑在位于曼哈顿西26街的“车库”的一个古老的跳蚤市场购买。这位卖家是位于土耳其安卡拉的美国人，这笔交易发生在他当天收拾摊位的时候。Manning编辑没有向他的人提供购买所需的大量现金，信用卡和支票都被礼貌地拒绝了。当晚卖家飞回安卡拉，情况似乎无望。解决方案是什么？事实证明，这只不过是一个握手的老式口头协议。卖方建议通过电汇将钱转移给他，编辑带着银行信息走出一张纸，手臂上的图像组合。毋庸置疑，我们第二天转移了资金，我们对这位不知名的人对我们中的一个人的信任感到非常感激和印象深刻。它回忆起很久以前可能发生过的事情。

奥斯曼系列中的图片，就像曼宁的封面上出现的其他插图一样，将两个世纪以前的丰富多样的服饰风格赋予生命。他们回忆起那个时期的隔离感和距离感，以及除了我们自己的超运动现象之外的所有其他历史时期。从那时起，着装规范发生了变化，当时如此丰富的地区多样性逐渐消失。现在通常很难将一个大陆的居民与另一个大陆的居民区分开来。可悲的是，我们已经将文化和视觉多样性换成了更多变化的个人生活。或者更多样化和有趣的知识和技术生活。

我们在Manning庆祝计算机业务的创造性，主动性，以及是，计算机业务的乐趣，基于两个世纪以前丰富的区域生活多样性，通过这个系列中的图片重现生机。

# *第一部分*

*Electron入门*

你最近用过Slack吗？ 也许你已经在Atom或Visual Studio Code中编写了一些代码，或者使用WhatsApp桌面应用程序向朋友发送了一条消息。如果是这样，那么你已经使用了Electron应用程序。那么，Electron是什么？ 简短的版本是它是一个用于构建使用Web技术在macOS，Windows和Linux上运行的桌面应用程序的平台。Electron将Node.js与Chromium（Google Chrome的开源基础）相结合。长篇答案是本书的重点，特别是第1章。

如果您是一个负责为多个平台构建桌面应用程序的小团队的一员，那么Electron是一种构建产品的好方法，无需管理两到三个不同的代码库，在两个或三个平台上压缩相关错误， 或实施相同功能两到三次。如果您是Node.js开发人员，希望将您的命令行应用程序放在更广泛的受众面前，Electron可以轻松构建图形用户界面（GUI），而无需学习全新的技能。如果您是一位已经习惯于构建自己的问题解决方案的网络开发人员，Electron可以轻松访问存在于浏览器沙箱之外的计算机部分。

根据我的经验，学习Electron既有短期和长期的影响。当您启动它或使用JavaScript从操作系统触发原生文件对话框时，立即看到停靠栏或任务栏中出现一个图标, 这一点非常令人满意。但是，随着您对Electron越来越熟悉，您将找到无法使用浏览器或Node.js构建的应用程序的想法。您将能够构建一个您可能无法构建的新类别的应用程序。我希望本书中的示例在您开始作为桌面应用程序开发人员的旅程时提供灵感，而不仅仅是指导。

在第一部分中，我们将弄清楚Electron是什么和不是什么。我们将看一些在野外使用它的大型玩家。我将在第1章详细说明它与基于浏览器的应用程序的不同之处。在第2章中，我们将构建一个简单的Electron应用程序，试图说服您使用Electron构建应用程序既简单又有趣。

*介绍Electron*

本章包括

* 了解Electron是什么
* 学习Electron所依赖的技术
* 了解Electron与传统Web应用程序的不同之处
* 构建Electron应用
* 在生产中使用Electron来构建真实世界的应用程序

网络的一大特色就是无处不在。这是一个创建协作应用程序的惊人平台，可以从运行不同操作系统的各种设备访问这些应用程序。也就是说，整个应用程序类都无法在浏览器环境中构建。Web应用程序无法访问文件系统。他们无法执行非JavaScript编写的代码。它们无法连接到桌面应用程序可以使用的许多操作系统API。当没有可靠的互联网连接时，大多数Web应用程序都不可用。

很长一段时间，为桌面构建涉及采用完全不同的技能。我们中的许多人没有足够的空间来学习学习新语言和框架所需的长学习曲线。使用Electron，您可以使用现有技能作为Web开发人员来构建具有原生桌面应用程序许多功能的应用程序。

**3**

### *Electron是什么?*

Electron是一个运行时，允许您使用HTML5，CSS和JavaScript创建桌面应用程序。这是一个开源项目，由GitHub的工程师Cheng Zhao（又名zcbenz）发起。以前称为Atom Shell，Electron是Atom的基础，Atom是GitHub使用Web技术构建的跨平台文本编辑器。

您可能已经听说过或使用过Apache Cordova或Adobe PhoneGap来构建包含在本机shell中的Web应用程序,用于移动操作系统，如iOS，Android和Windows Phone。如果是这样，那么将Electron视为构建桌面应用程序的类似工具可能会有所帮助。

Electron允许您使用您已知的Web技术来构建您不会构建的应用程序。在本书中，您将学习如何构建与Windows，macOS和Linux上的本机操作系统API相连的应用程序。

Electron结合了Chromium Content Module和Node.js运行时。它允许开发人员使用网页构建GUI，并通过与操作系统无关的API访问Windows，macOS和Linux上的本机操作系统功能。

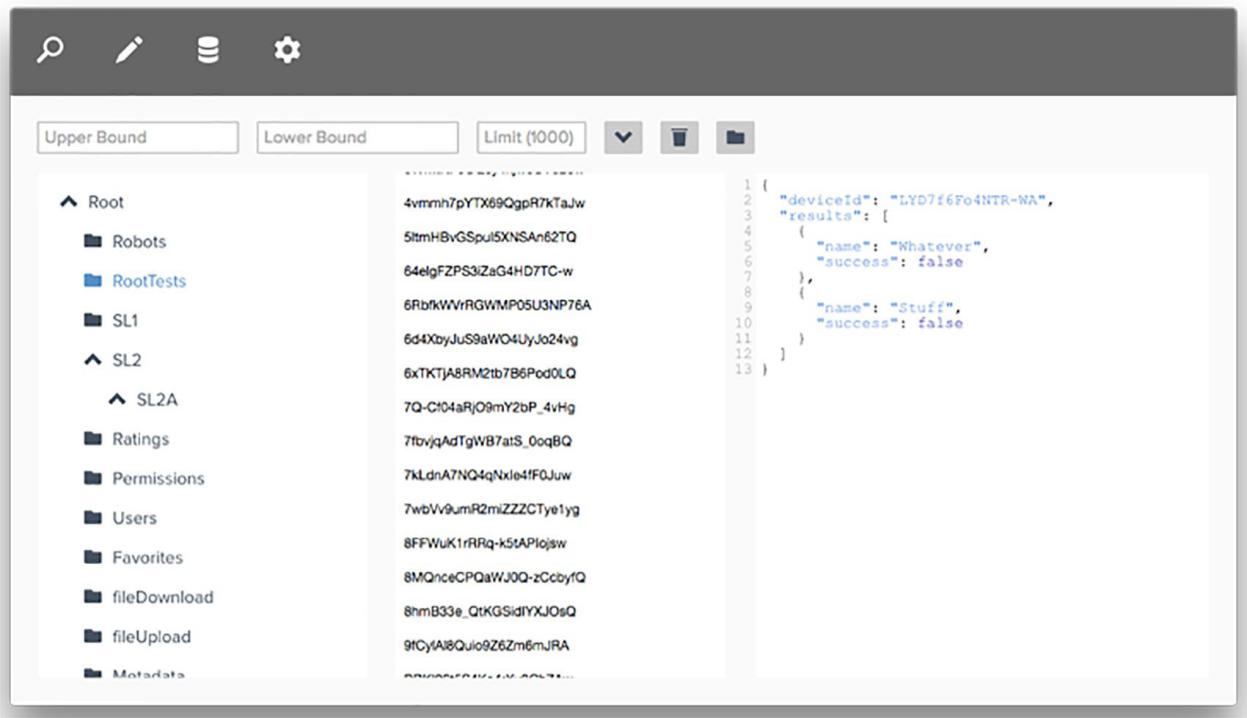
Chromium和Node本身就是广受欢迎的应用程序平台，它们都被独立用于创建雄心勃勃的应用程序。Electron将两个平台结合在一起，允许您使用JavaScript构建一个全新的应用程序类。您可以在浏览器中执行任何操作，您可以使用Electron。您可以使用Node做任何事情，您可以使用Electron。

令人兴奋的部分是您可以将两种技术结合在一起。您可以构建利用这两个平台并构建仅在一个平台上无法实现的应用程序。这就是本书的全部内容。Electron设备不仅是构建与原生桌面应用程序相似的Web应用程序的绝佳选择。它也是围绕Node应用程序构建GUI的绝佳选择，否则这些应用程序将仅限于命令行界面。见图1.1。

假设您要构建一个允许您在计算机上查看和编辑图像文件夹的应用程序。传统的浏览器应用程序无法访问文件系统。他们无法访问照片目录，加载目录中的任何照片，或保存您在应用程序中所做的任何更改。使用Node，您可以实现所有这些功能，但是您无法提供GUI，这会使您的应用程序难以用于普通用户。通过将浏览器环境与Node相结合，您可以使用Electron创建一个应用程序，您可以在其中打开和编辑照片，并提供用户界面。见图1.2。

Electron不是一个复杂的框架,它是一个简单的运行时。与您从命令行使用Node的方式类似，您可以使用electron命令行工具运行Electron应用程序。您无需学习许多约定即可开始使用，并且您可以自由地构建应用程序，但是我会在本书中提供提示和最佳实践。

***Electron是什么* 5**



**图1.1 LevelUI是使用Electron构建的Node的LevelUp数据库的GUI。您无法在传统浏览器中构建此应用程序，因为它无法访问用户计算机上的本地数据库。它也无法使用LevelUI库，因为它是一个已编译的C++模块，只有Node而不是浏览器才能使用。**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Electron** | | | |  |
|  | | |  | | |
| **Node.js** | |  |  | **Chromium Content Module** | |
|  | |
| Filesystem access | | Rendering HTML and CSS | |
| Support for compiled modules | | Document Object Model (DOM) | |
| CommonJS Module System | | Web APIs | |

图1.2 Electron将Chromium的核心Web浏览组件与Node的低级系统访问相结合。

###### *Chromium Content Module是什么?*

Chromium是Google Chrome浏览器的开源版本。它共享许多相同的代码和功能，只有一些细微差别和不同的许可。内容模块是核心代码，允许Chromium在独立进程中呈现网页并使用GPU加速。它包括Blink渲染引擎和V8 JavaScript引擎。内容模块使Web浏览器成为Web浏览器。它处理从Web服务器获取和呈现HTML，加载任何引用的CSS和JavaScript，相应地设置页面样式以及执行JavaScript。

考虑内容模块的最简单方法是考虑它不做的事情。内容模块不包含对Chrome扩展程序的支持。它不会处理您的书签和历史记录与Google的云服务同步。当您访问页面时，它无法安全地存储您保存的密码或自动填写密码。它不会检测某个页面是否是用其他语言编写的，然后会调用Google的翻译服务寻求帮助。内容模块仅包含呈现HTML，CSS和JavaScript所需的核心技术。

###### *Node.js是什么?*

在其存在的前15年中，JavaScript传统上是在Web浏览器中隔离的。在服务器上运行JavaScript的支持方式并不多。项目存在，但他们从来没有任何牵引力。Node.js项目最初于2009年发布，是一个开源的跨平台运行时，用于使用JavaScript开发服务器端应用程序。它使用Google的开源V8引擎来解释JavaScript并添加了API，用于访问文件系统，创建服务器以及从外部模块加载代码。

在过去几年中，Node受到了越来越多的关注和普及，并且用于各种用途，从编写Web服务器到控制机器人，以及您猜测它构建桌面应用程序。Node捆绑了一个名为npm的软件包管理器，可以轻松地依赖其注册表中的250,000多个库。

### *谁在使用Electron?*

Electron被大大小小的公司用于构建桌面应用程序。如前所述，它最初是作为GitHub的Atom文本编辑器的基础而开发的。Atom需要访问文件系统以履行其作为文本编辑器的职责。同样，其他公司已将Electron作为其文本编辑应用程序的基础。Facebook发布了Nuclide作为Atom之上的软件包，它将文本编辑器变成了一个成熟的集成开发环境（IDE），为使用React Native，Hack和Flow项目提供了一流的支持。微软还将Electron用于其跨平台的Visual Studio Code编辑器，该编辑器可在macOS，Windows和Linux上运行。

您可以使用Electron构建多个文本编辑器。Slack是一种流行的消息传递应用程序，它将Windows用于Windows和Linux版本。Nylas使用Electron作为其N1Electron邮件客户端，其设计在所有主要平台上都很漂亮。它还支持JavaScript插件架构，允许第三方开发人员添加功能并扩展UI。

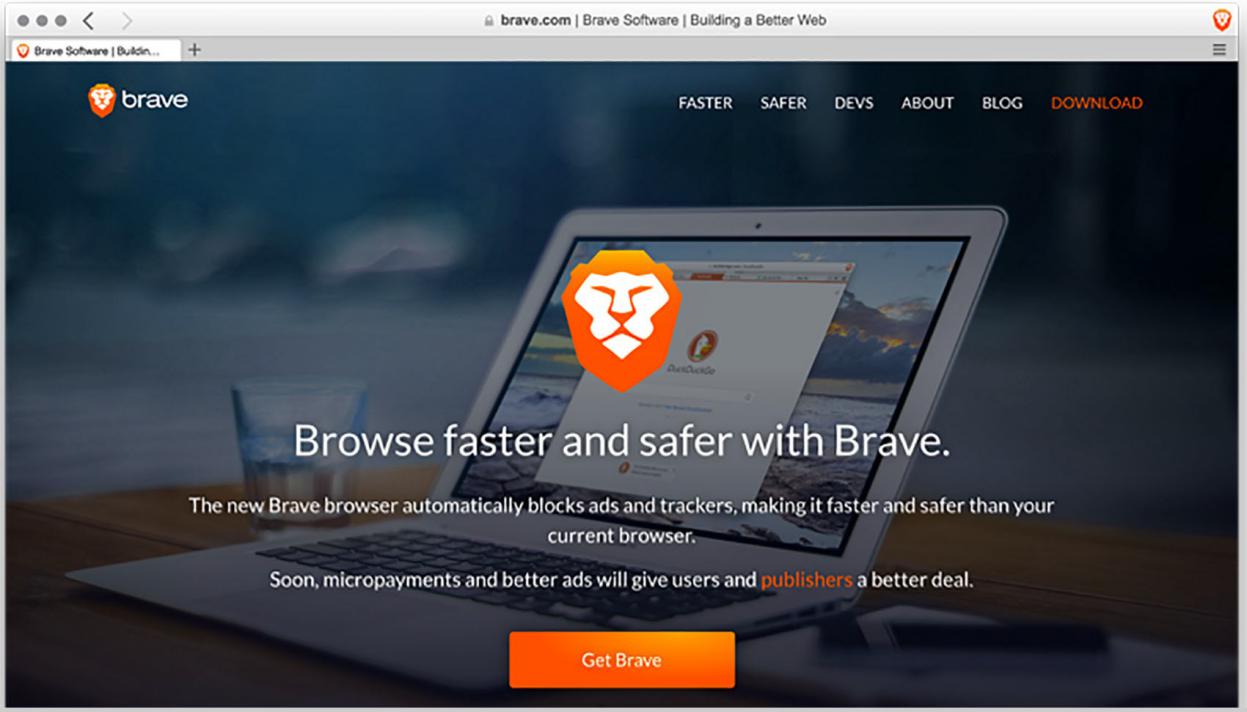
***谁在使用Electron?* 7**

Particle公司生产用于创建自定义硬件的开发套件，它使用Electron作为其IDE，它允许用户通过蜂窝或Wi-Fi网络编写代码并将其部署到硬件设备。使用Mapbox Studio，用户可以导入本地存储的数据并在其计算机上处理它，而无需通过Internet将其发送到Mapbox的服务器。结果是更快更好的体验，使设计师可以轻松创建自定义地图。

Dat是一个用于共享，同步和版本化分散数据的开源工具。由赠款资助的项目由三个Web开发人员组成。尽管是一个相对较小的团队，Dat还是使用Electron为该项目发布了一个桌面应用程序。2015年，Wiredcraft,一家软件咨询公司使用Electron构建了一个离线友好的Windows应用程序，用于收集和纠正缅甸的选民登记信息。该公司需要一个应用程序，可以离线存储收集的数据，然后在设备连接到网络时发布它。该公司选择Electron作为使用C++构建它的替代方案，因为它允许Wiredcraft利用其现有的HTML，CSS和JavaScript实力，而不是为不同的生态系统重新学习这些技能。

Brave是一个专注于速度和安全性的新浏览器，它是JavaScript的创造者Brendan Eich，它本身建立在Electron之上。见图1.3。没错，您甚至可以使用Web技术构建Web浏览器。

建立在Electron之上的新项目每天都在发布，因为公司和开发人员看到了构建产品的价值，这些产品使用了桌面应用程序提供的功能，同时仍然维护着网络的内在平台不可知性。在本书的最后，您将学习现有的Web开发技能并将其应用于创建传统浏览器环境中无法实现的新应用程序。



**图1.3 Brave是一个构建在Electron之上的完整Web浏览器。**

### *我需要知道什么?*

让我们从你不需要知道的东西开始吧。本书面向希望使用现有技能组合创建在传统浏览器环境中无法实现的桌面应用程序的Web开发人员。您不需要任何构建桌面应用程序的经验就能从本书中获取价值。

也就是说，你应该习惯编写JavaScript，HTML和CSS，但并不意味着需要你是一个专家。我不会在本书中介绍变量或条件，但如果您熟悉JavaScript的一般语言特性，那么您可能需要具备必要的技能。如果您熟悉Node.js中的一些约定和模式（例如模块系统的工作方式），这也很有帮助。我们会在遇到这些概念时探索这些概念。

### *我为什么要使用Electron？*

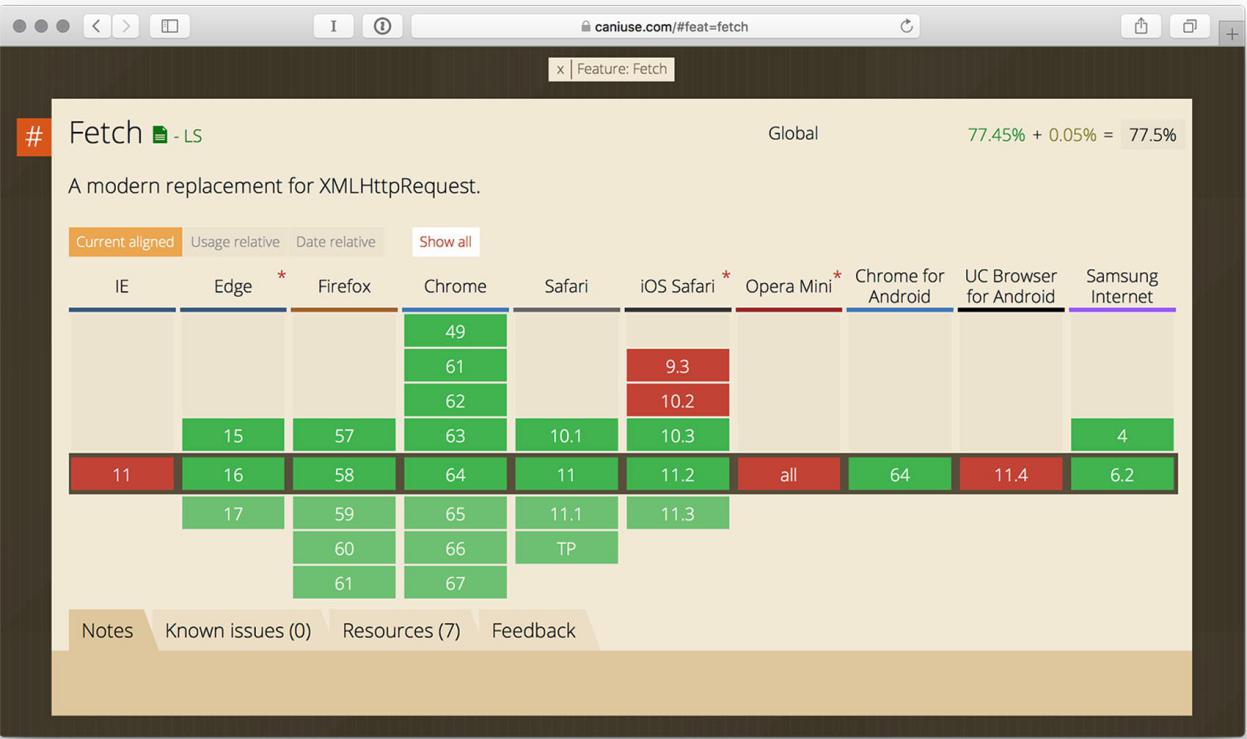
当您为Web浏览器编写应用程序时，您必须保守选择使用哪种技术，并谨慎编写代码。这是因为与许多服务器端情况不同:您编写的代码将在其他人的计算机上执行。

您的用户可能正在使用最新版本的现代浏览器，例如Chrome或Firefox，或者他们可能正在使用过时版本的Internet Explorer。你几乎无法说出代码的渲染和执行位置。你必须为任何事情做好准备。

您通常必须为当前使用的所有浏览器的所有版本提供最广泛支持的功能的最低公分母编写代码。即使问题存在更好，更有效或通常更具吸引力的解决方案，您也可能无法使用该方法。当您决定使用现代浏览器功能时，通常需要实施优雅后备，特征检测和渐进增强的应急计划，这会为您的开发工作流程增加一些重要的摩擦。

当您使用Electron构建应用程序时，您将打包特定版本的Chromium和Node.js，因此您可以依赖这些版本中可用的任何功能。您不必关心其他浏览器及其版本支持的功能。例如，如果您的应用程序附带的Chromium构建支持Service Worker API，那么您可以放心地在应用程序中依赖该API。见图1.4。

Electron允许您使用尖端的Web平台功能，因为它包含相对较新版本的Chromium。一般来说，Electron产品中的Chromium版本比最新的稳定版本落后一到两周,并且每六周就有一个新的稳定版本发布。Electron通常会在发布后大约一个月内包含新版本的Node.js，以确保它包含最新版本的V8。Electron已经包含了Chromium的V8现代版本，可以在升级到最新版本的Node之前等待小错误修复。



**图1.4 在基于浏览器的Web应用程序中，由于支持不一致，依赖Fetch API可能不太实际。但是在您的Electron应用程序中，您正在捆绑当前稳定版本的Chromium，并完全支持Fetch API。**

###### *以您现有的技能为基础*

如果你像我一样，你可能比桌面应用程序更有经验构建Web应用程序。您希望添加为您的工具集创建桌面应用程序的功能，但是您没有足够的带宽来学习新的编程语言，也可能学习新的框架。

学习一门新语言或框架是一项不容忽视的投资。作为Web开发人员，您习惯于编写对所有用户同样有效的应用程序,即使这意味着要与特定浏览器或屏幕大小的特性作斗争。但是当你在考虑构建传统的桌面应用程序时，你不仅仅在谈论学习一种语言和框架。如果您想要定位Windows，macOS和Linux，您还需要学习至少三种不同的语言和框架。

个人和小团队可以使用Electron在其他情况下无法提供桌面应用程序。对于一个小团队，雇用熟练为每个平台构建应用程序的开发人员可能不是一个选择。Electron允许您使用现有技能并将应用程序部署到所有主要平台。使用Electron，您可以支持多个操作系统，而不是通常用于支持多个浏览器的工作。

###### *访问原生操作系统API*

Electron应用程序与任何其他桌面应用程序类似。它们与其他原生应用程序一起存在于文件系统中。它们位于macOS中的dock或Windows和Linux中的任务栏中，其他所有原生应用程序都会挂起。Electron应用程序可以触发原生打开和保存文件对话框。可以将这些对话框配置为允许操作系统仅选择具有特定文件扩展名，整个目录或多个文件的文件。您可以将文件拖到Electron应用程序上并触发不同的操作。

此外，Electron应用程序可以像任何其他应用程序一样设置自定义应用程 见图1.5。当用户从应用程序中右键单击时，他们可以创建自定义上下文菜单。您可以使用Chromium的通知API来触发系统级通知。他们可以从系统剪贴板中读取文本，图像和其他媒体。



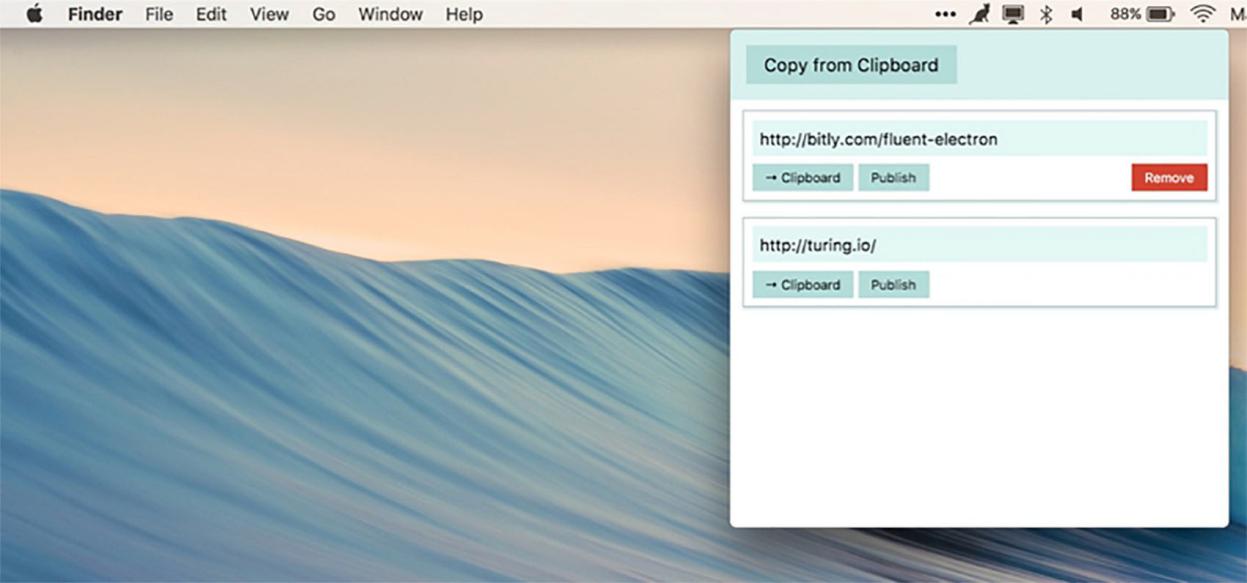
**图1.5 Electron允许您创建自定义应用程序菜单。**

与传统的Web应用程序不同，Electron应用程序不仅限于浏览器。您可以创建位于菜单栏或系统托盘中的应用程序。见图1.6。您甚至可以通过操作系统中的任何位置通过特殊按键来注册全局快捷方式来触发这些应用程序或其任何属性。

Electron应用程序可以访问系统级信息，例如计算机是使用电池供电还是插入墙上(译注:直接供电)。它们还可以使操作系统保持唤醒状态，并在必要时防止其进入省电模式。

###### *增强的权限和更宽松的限制*

Web是历史上最大的分布式应用程序平台。它无处不在，以至于Web开发人员将许多相关的头痛视为理所当然。构建Web应用程序涉及精心编排服务器端应用程序与可能的数千个客户端应用程序实例之间的通信。您的客户端代码在用户的Web浏览器中运行--远离服务器。



**图1.6 您可以创建一个位于操作系统菜单栏或系统托盘中的应用程序。**

除非将更改发送回您的服务器，否则客户端中发生的任何事情对于该浏览器会话都是唯一的。出于同样的原因，如果您的结果发生任何变化，您必须等到客户端发送另一个要求更新的HTTP请求;或者，如果您已在客户端和服务器上实现该功能，则可以通过WebSockets发送更新。

桌面应用程序享有更广泛的功能，并且对他们允许做的事情的限制更少，因为用户明确地竭力下载，安装和打开应用程序。但是，当您浏览网页时，您的代理机构数量不同。您正在执行未选择在计算机上安装的代码。因此，Web应用程序对其允许的操作有很多限制。

当浏览器访问Web上的页面时，它会愉快地下载所加载的文档的HTML代码中引用的所有资产，以及这些第一个资源添加的任何其他依赖项，然后开始执行代码。多年来，浏览器供应商对浏览器可以采取的措施增加了限制，以防止恶意代码损害用户或互联网上的其他网站。

我不是一个坏人，但让我说,为了争论,我是。我们还说我经营一个热门网站，销售手工制作的小工具。有一天，竞争对手突然出现在我的雷达上，以极低的折扣销售同样自命不凡的小部件。我的网站目前仍在增加流量，但这个新的挑战者正在影响我睡美容觉。

作为一个坏人，我决定在我的网站上添加JavaScript，每隔几毫秒就会向我的竞争对手的网站发出一个AJAX请求，希望我网站上的数千名访问者能够下载这些代码并有效地淹没我死敌的服务器并导致它无法处理任何合法的请求。它也会降低我的访问者在我的网站上的体验，但这是我愿意付出的代价，以使我的竞争对手的网站瘫痪。

尽管我的计划具有恶魔般的性质，但它不起作用。现代浏览器限制客户端代码向第三方服务器发出请求，除非该服务器明确声明允许此类请求的策略。

一般来说，大多数网站都不这样做。如果要向第三方服务器发送请求，则必须先向自己的服务器发出请求，让它与第三方联系，然后将结果转发回客户端。在前面的示例中，这会将我的服务器添加为成千上万请求的瓶颈，这将使我无法启动此类攻击，并且我的竞争对手很容易阻止我的单个IP地址而不是我的网站上成千上万的访客IP。浏览器还严格限制客户端代码可以访问的内容以及它可以执行的操作。所有这些都为用户提供了更安全，最终更好的体验。这一切都非常实用，也是使网络成为用户极好且平易近人的平台的一部分。

也就是说，所有这些有用且重要的安全限制严重限制了您可以使用Web技术构建的应用程序类型。用户明确地下载并安装Electron应用程序，就像任何其他原生应用程序一样。您可以像任何原生桌面应用程序或服务器端Node进程一样自由访问文件系统。您还可以自由地向第三方API发出请求，而无需通过Node服务器，因为您可以访问与任何其他Node进程相同的权限。见图1.7。

传统Web应用

在传统的Web应用程序 中，客户端代码无法从第三方API请求数据。请求必须通过服务器端应用程序进行代理。

My server application

My client-side code

A third-party API

Electron 应用

My client-side code

A third-party API

**在Electron应用程序中，客户端代码具有与服务器端代码相同的所有权限，可以直接向第三方API发出请求。**

**图1.7 Electron应用程序可以使用其Node.js运行时向第三方API发出请求。**

***Electron如何工作？* 13**

###### *从浏览器上下文访问Node*

除了授予对文件系统的访问权限以及启动Web服务器的能力之外，Node.js还使用基于CommonJS模块规范的模块系统。从最早的版本开始，Node就支持将代码分解为多个模块并明确包含您在给定文件中所需的代码。

为浏览器打包任何大量的JavaScript代码并不总是那么容易。对于少量代码，您可以将其包含在匹配的一对开始和结束<script>标记之间的标记中。对于较大的代码块，可以使用src属性引用外部JavaScript文件。您可以随心所欲地多次这样做，但是当浏览器触发额外的获取每个外部资产的请求时，您将不得不支付性能损失。

如果您愿意，欢迎使用构建工具，例如webpack或Browserify，但在Electron应用程序中通常不需要，因为浏览器中可以使用Node的所有全局属性（例如，require，module和exports）内容。您可以将Node的模块系统用于您传统上认为的客户端，而无需向应用程序添加构建过程。

您可以从Electron应用程序的浏览器上下文中访问所有Node的API。除了利用Node的模块系统之外，您还可以使用具有本机扩展的编译模块，访问文件系统，以及执行浏览器环境中通常不支持的其他内容。

###### *离线优先*

任何曾经在跨大陆航班上使用计算机的人都可以证明，如果没有与互联网的连接，大多数基于浏览器的Web应用程序都不是很好。即使使用任何流行的客户端框架（如Ember，React或Angular）的高级Web应用程序，通常也需要连接到远程服务器以下载其需要的资源。

Electron应用程序已经下载到用户的计算机上。通常，它们加载本地存储的HTML文件。从那里，如果有连接，他们可以请求远程数据和资源。Electron甚至提供API，允许您检测连接是否可用。使用Electron构建离线应用程序不需要特殊的清单或前沿技术--它是默认状态，除非应用程序明确要求来自Internet的内容。除非有特殊情况,例如，您正在构建聊天客户端 ,Electron应用程序与其他任何应用程序一样可以离线工作。

### *Electron如何工作？*

Electron应用程序包括两种类型的进程：主进程和零个或多个渲染器进程。每个进程在应用程序中扮演不同的角色。Electron运行时包含不同的模块，可帮助您构建应用程序。某些模块（例如从系统剪贴板读取和写入的功能）可用于两种类型的进程。其他方面，例如访问操作系统API的能力，仅限于主进程。见图1.8。

Electron读取package.json中的“main”条目，以确定要作为主进程运行的文件。

主进程

渲染器进程

渲染器进程



主进程可以创建多个渲染器进程

渲染器进程

**如果需要访问操作系统级API，渲染器进程可以与主进程通信。**

**图1.8 Electron多进程架构**

当Electron启动时，它会转到包含在项目中的package.json清单中的开始条目，以确定应用程序的入口点。只要在package.json中正确包含此文件，就可以将此文件命名为您想要的任何名称。Electron将此文件作为主进程运行。

###### *主进程*

主进程有一些重要的责任。它可以响应应用程序生命周期事件，例如启动，退出，准备退出，前往后台，到达前台等等。主进程还负责与原生操作系统API进行通信。如果要显示打开或保存文件的对话框，可以从主进程执行。

###### *渲染器进程*

主进程可以使用Electron的Browser-Window模块创建和销毁渲染器进程。渲染器进程可以加载网页以显示GUI。每个流程都利用Chromium的多进程架构，并在自己的线程上运行。然后，这些页面可以加载其他JavaScript文件并在此过程中执行代码。与普通网页不同，您可以访问渲染器进程中的所有Node API，从而允许您使用原生模块和较低级别的系统交互。

渲染器进程彼此隔离，无法访问操作系统集成API。Electron拥有促进进程之间通信的能力，以允许渲染器进程在需要触发打开或保存文件对话框或访问任何其他OS级集成的情况下与主进程进行通信。

***Electron对比NW.js* 15**

### *Electron对比NW.js*

Electron类似于另一个名为NW.js的项目（以前称为node-webkit）。这两者有很多共同之处。事实上，在开始Electron工作之前，zcbenz是NW.js的重要贡献者。也就是说，它们在几个重要方面有所不同，如表1.1所示。

**表1.1 Electron和NW.js之间的一些主要差异的比较**

|  |  |  |
| --- | --- | --- |
|  | **Electron** | **NW.js** |
| **Platform** | Officially supported Chromium Content Module from recent build | Forked version of Chromium |
| **Process model** | Separate processes | Shared Node process |
| **Crash reporting** | Built in | Not included |
| **Auto-updater** | Built in | Not included |
| **Windows support** | Windows 7 and later | Windows XP and later |

NW.js使用分叉版本的Chromium。Electron使用Chromium和Node.js但不修改它们。这使Electron更容易与最新版本的Chromium和Node保持同步。Electron还包括用于自动下载更新和报告崩溃的模块。NW.js没有。

NW.js应用程序从HTML页面开始。每个浏览器窗口共享一个共同的Node进程。如果打开了多个窗口，它们都共享相同的Node进程。Electron使Node和浏览器进程分开。在Electron中，您从Node启动主进程。这个主进程可以打开浏览器窗口，每个窗口都是自己的进程。Electron提供了API来促进主进程和浏览器窗口之间的通信，我们在本书中称之为渲染器进程。

如果需要考虑向后兼容性，那么NW.js可能是更好的选择，因为它支持Windows XP和Vista。Electron仅支持Windows 7及更高版本。对于以多媒体为重点的应用程序，Electron通常是更好的选择，因为Chromium的FFmpeg库是静态链接的依赖项，因此Electron支持更多开箱即用的编解码器。使用NW.js，您需要手动链接FFmpeg库。

### *本章小结*

* + - Electron是使用Web技术构建桌面应用程序的运行时。
    - 该项目始于GitHub，作为Atom文本编辑器的基础。
    - Electron将Chromium Content Module与Chrome网页浏览器的精简版本与Node结合在一起。
    - 这种组合允许您构建可以访问文件系统和已编译模块的应用程序，以及呈现UI和使用Web API。
    - Electron被大大小小的应用程序使用，例如Atom，Microsoft的Visual Studio Code和Slack。
    - Electron非常适合个人或小团队，他们可能希望定位多个平台，而无需学习三种或更多语言，以及每个平台的框架。
    - Electron允许Web开发人员使用他们现有的技能集来构建在浏览器环境中无法实现的应用程序。
    - Electron附带现代版的Chromium和Node，这意味着您可以使用Web平台的最新和最强大的功能。
    - Electron应用程序可以访问操作系统API，例如应用程序和上下文菜单，文件打开和保存对话框，电池状态和电源设置等。
    - 与基于浏览器的Web应用程序相比，Electron应用程序允许增强权限，对其功能的限制更少。
    - Electron应用程序由一个主进程和一个或多个渲染器进程组成。
    - 主进程处理操作系统集成，管理应用程序的生命周期，并创建渲染器进程。
    - 渲染器进程渲染呈现UI并响应用户事件。
    - Electron与NW.js的不同之处在于它使用Chromium官方支持的内容模块而不是NW.js，后者使用Chromium的自定义分支。

*你的第一个Electron应用*

***本章包括***

* 构建和设置Electron应用程序
* 生成package.json，并将其配置为与Electron一起开发
* 在您的项目中为您的平台包含一个预构建的Electron版本
* 配置package.json以启动主进程
* 从主进程中创建渲染器进程
* 利用Electron轻松的沙盒限制来构建浏览器内部通常无法实现的功能
* 使用Electron的内置模块来解决一些常见问题

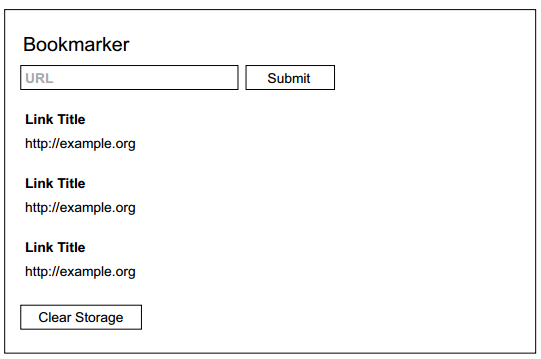
在第1章中，我们讨论了Electron处于高水平的问题。也就是说，这本书叫做Electron实战吧？ 在本章中，我们通过设置和构建一个简单的应用程序来管理书签列表，从而学习Electron的知识。该应用程序将利用仅在最现代的浏览器中可用的功能。

在前一章的高级讨论中，我提到Electron是一个类似Node的运行时。这仍然是正确的，但我想暂时重温这一点。Electron不是一个框架,它不提供任何脚手架，也没有关于如何构建应用程序或命名文件的强有力规则。这些选择由我们和开发人员决定。从好的方面来看，它也没有强制执行任何约定，而且在弄脏手之前(译注:作者风趣)我们需要讨论较少的概念样板信息。

### *构建一个书签列表应用*

让我们从构建一个简单而且有点天真的Electron应用程序开始，以强化我们所涵盖的一切。我们的应用接受URL。当用户提供URL时，我们获取URL引用的页面标题并将其保存在我们的应用程序的localStorage中。最后，我们显示应用程序中的所有链接。您可以在GitHub上找到本章的完整源代码（https://github.com/electron- in-action/bookmarker）。

在此过程中，我们发现了在Electron中构建应用程序的一些优势，例如能够绕过对服务器的需求并使用在所有浏览器中没有广泛支持但是在现代Chromium的版本实现的尖端Web API。图2.1是我们在本章中构建的应用程序的线框图。



**图2.1 我们在本章中构建的应用程序的线框图**

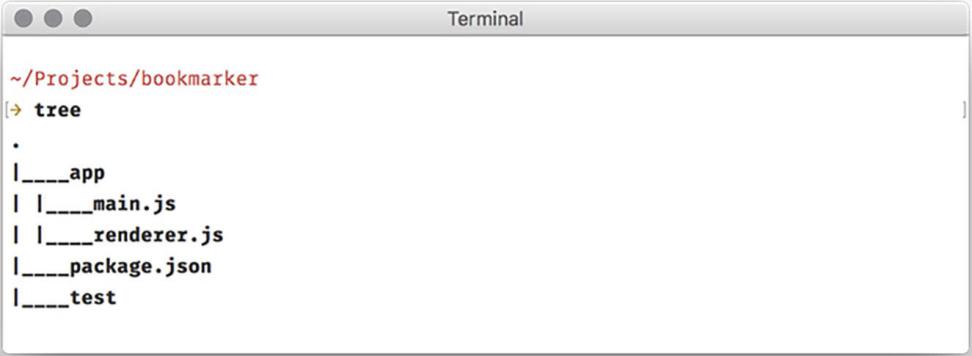
当用户将他们想要保存的网站的URL添加到输入字段下方的列表中时，应用程序向网站发送请求以获取标记。成功收到标记后，应用程序会提取网站标题，并将标题和URL附加到网站列表中，这些网站存储在浏览器的localStorage中。应用程序启动时，它从localStorage读取并恢复列表。我们添加一个带有命令的按钮，以便在出现任何问题时清除localStorage。由于这个简单的应用程序旨在帮助您熟悉Electron，因此我们不会实施高级移动，例如从列表中删除单个网站。

###### *结构化Electron应用程序*

您选择如何构建应用程序取决于您的团队或处理该应用程序的个人。许多开发人员的方法略有不同。看一些更成熟的Electron应用程序，我们可以识别常见的模式，并决定我们在本书中如何处理我们的应用程序。

出于我们的目的，让我们就本书其余部分的文件结构达成一致。我们有一个app目录，用于存储所有应用程序代码。我们还有一个package.json，它将存储依赖项列表，有关我们应用程序的元数据以及脚本，并声明Electron应该在哪里查找我们的主进程。在我们安装了依赖项之后，我们最终得到了Electron代表我们创建的node\_modules目录，但我们不会在初始设置中包含它。

就文件而言，让我们从我们的应用程序中的两个文件开始：main.js和renderer.js。这些是故意简单的文件名，因此我们可以跟踪两种类型的过程。我们在本书中构建的所有应用程序的开始大致遵循图2.2中所示的目录结构。（如果您正在运行macOS，则可以使用brew install tree命令。）



**图2.2 我们第一个Electron应用程序的文件树结构**

创建一个名为“bookmarker”的目录并导航到它。您可以通过从命令行运行以下两个命令来快速创建此结构。稍后您将使用npm init生成package.json文件。

**mkdir app**

**touch app/main.js app/renderer.js app/style.css app/index.html**

Electron不需要这种结构，但它受到其他Electron应用程序建立的一些最佳实践的启发。Atom将所有应用程序代码保存在app目录及其所有样式表和其他资源（如静态目录中的图像）中。LevelUI在顶层有一个index.js和一个client.js，并将所有依赖文件保存在样式目录中的src目录和样式表中。Yoda将所有文件（包括加载应用程序其余部分的文件）保存在src目录中。app，src和lib是包含大部分应用程序代码的文件夹的通用名称，样式，静态和资产是保存应用程序中使用的静态资产的目录的通用名称。

###### *pac**kage.json*

package.json清单用于许多（如果不是大多数）Node项目中。该文件包含有关该项目的重要信息。它列出了元数据，例如作者姓名以及他们的Electron邮件地址，项目发布的许可证，项目的git存储库的位置以及存档问题的位置。它还定义了常见任务的脚本，例如运行测试套件或与构建应用程序的需求相关的脚本。package.json文件还列出了用于运行和开发应用程序的所有依赖项。

理论上，您可能有一个没有package.json的Node项目。但是Electron依赖于这个文件及其主要属性来确定在加载或构建应用程序时从哪里开始。

npm，随Node一起提供的包管理器，附带了一个生成package.json的有用工具。从您之前创建的“bookmarker”目录中，运行npm init。如果将提示留空，则npm将冒号后括号中的任何内容用作默认答案。当然，您的答案应该与图2.3类似，但作者姓名除外。

值得注意的是示例package.json中的主要条目。在这里，您可以看到我根据我们设置应用程序的方式将其设置为指向./app/main.js。您可以指向您想要的任何文件。我们将要使用的主文件恰好被称为main.js，但它可以被命名为任何东西（例如，sandwich.js，index.js，app.js）。



**图2.3 npm init提供了一系列提示并设置了package.json文件**

###### *在我们的应用中下载并安装Electron*

我们已经建立了应用程序的基本结构，但是Electron无处可寻。从源建立Electron需要一段时间，可能很乏味。我们依赖于每个平台（macOS，Windows和Linux）和两种架构（32位和64位）的预构建版Electron。我们使用npm安装Electron。

npm允许我们在全局或本地为每个项目安装二进制文件。全局安装Electron似乎很方便，但如果我们有多个应用程序使用不同版本的Electron，它可能会给您带来麻烦。我们最好为我们工作的每个项目指定和安装一个独特版本的Electron。

下载和安装Electron很容易。从先前运行npm init的项目目录中运行以下命令：

**npm install electron --save**

**(译注:译者验证时Electron的版本是5.0.2,npm初次下载可能会有些慢甚至失败,译者多次失败,使用**npm install -g cnpm

**后,执行cnpm install electron --save 很快就成功了!)**

此命令将在项目的node\_modules目录中下载并安装Electron。（如果你还没有目录，它也会创建目录。） --save标志将它添加到package.json中的依赖项列表中。这意味着如果有人下载项目并运行npm install，它们将默认获得electron。

**关于electron-prebuilt一词**

当你适应Electron的世界时，你可能会看到博客文章，文档，甚至本书的早期版本都是指electron-prebuilt包而不是electron。过去，前者是为您的操作系统安装预编译版Electron的首选方式。后者是新的优先方式。截至2017年初，不再支持electron-prebuilt。

npm还允许您定义在package.json中运行公共脚本的快捷方式。当您运行package.json中定义的脚本时，npm会自动将node\_modules添加到路径中。这意味着默认情况下它将使用本地安装的Electron版本。让我们在package.json中添加一个启动脚本。

**清单2.1 向package.json添加启动脚本**

{

"name": "bookmarker", "version": "1.0.0",

"description": "Our very first Electron application", "main": "./app/main.js",

"scripts": {

**//当我们使用npm start时，npm会运行什么。**

"start": "electron .",

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "Steve Kinney", "license": "ISC", "dependencies": {

"electron": "^2.0.4"

**//此处可以看出此项目基于2.0.4的版本,译者的已经是基于5.0.2版本**

}

}

现在当我们运行npm start时，npm使用我们本地安装的electron版本来启动Electron应用程序。你会发现似乎没有多少事情发生。您应该在终端应用程序中看到以下代码：

> [bookmarker@1.0.0](mailto:bookmarker@1.0.0) start /Users/stevekinney/Projects/bookmarker

> electron .

***使用主进程* 23**

您还将在Dock或任务栏中看到一个新应用程序,我们刚刚设置的Electron应用程序,如图2.4所示。现在，它被称做简单的“Electron”，它使用Electron的默认应用程序图标。在后面的章节中，我们将看到如何自定义这些属性，但默认情况下现在已经足够了。我们所有的实现文件都是完全空白的。因此，此应用程序没有太多进展，但它存在并正确启动。我们认为这是暂时的胜利。在Windows上关闭应用程序的所有窗口或从应用程序菜单中选择退出将终止该过程。或者，您可以在Windows命令提示符或终端中按Control-C以退出应用程序。按Command-Period终止macOS上的进程。(译注:在deepin和Windows10上没有看到任何窗口与图标)



**图2.4 Dock中的应用程序不仅仅是任何Electron应用程序; 这是我们刚刚设置的Electron应用程序。**

### *使用主进程*

既然我们有一个Electron应用程序，那么如果我们真的可以让它做某事就会很酷。如果你回忆起第1章，Electron从一个主进程开始，可以创建一个或多个渲染器进程。我们首先在main.js中编写代码以使我们的应用程序开始。

要使用Electron，我们需要导入electron库。Electron附带了许多我们在本书中使用的有用模块。第一个也可以说是最重要的就是app模块。

**清单2.2 添加基本主进程：./app/main.js**

const {app} = require('electron');

**// 应用程序完全启动后立即调用。**

app.on('ready', () => { console.log('Hello from Electron');

});

app是一个处理应用程序生命周期和配置的模块。我们可以使用它来退出，隐藏和显示应用程序，以及获取和设置应用程序的属性。当应用程序进入不同状态时，app模块还会运行事件 包括退出前，窗口全部关闭，浏览器窗口模糊和浏览器窗口焦点。

在完全启动并准备好之前，我们无法使用我们的应用程序。幸运的是，应用程序启动了一个准备好的事件。这意味着我们需要耐心等待，并在我们做任何事情之前先监听应用程序启动就绪事件。在前面的代码中，我们登录到控制台，这是我们可以轻松完成的，没有electron，但此代码突出显示了如何监听ready事件。

### *创建一个渲染器进程*

我们的主进程与任何其他Node流程非常相似。它可以访问Node的所有内置库以及Electron提供的一组特殊模块，我们将在本书的过程中探讨这些模块。但是，与任何其他Node进程一样，我们的主进程没有DOM（文档对象模型），也无法呈现UI。主进程负责与操作系统交互，管理状态以及协调应用程序中的所有其他进程。它不负责呈现HTML和CSS。这是渲染器进程的工作。我们报名参加整个Electron冒险的主要原因之一是我们想要为Node流程创建一个GUI。

主进程可以使用BrowserWindow模块创建多个渲染器进程。每个BrowserWindow都是一个独立且独特的渲染器进程，包括DOM，访问Chromium Web API和Node内置模块。我们可以像访问app模块一样访问BrowserWindow模块。

**清单2.3 需要BrowserWindow模块：./app/main.js**

const {app, BrowserWindow} = require('electron');

您可能已经注意到BrowserWindow模块以大写字母开头。根据标准JavaScript约定，这通常意味着我们将其称为具有new关键字的构造函数。我们可以使用这个构造函数来创建我们喜欢或我们的计算机可以处理的任意数量的渲染器进程。当应用程序准备就绪时，我们创建一个BrowserWindow实例。让我们按如下方式更新代码。

**清单2.4 创建BrowserWindow：./app/main.js**

const {app, BrowserWindow} = require('electron');

//在我们的应用程序的主窗口的顶级范围中创建一个变量

let mainWindow = null;

app.on('ready', () => {

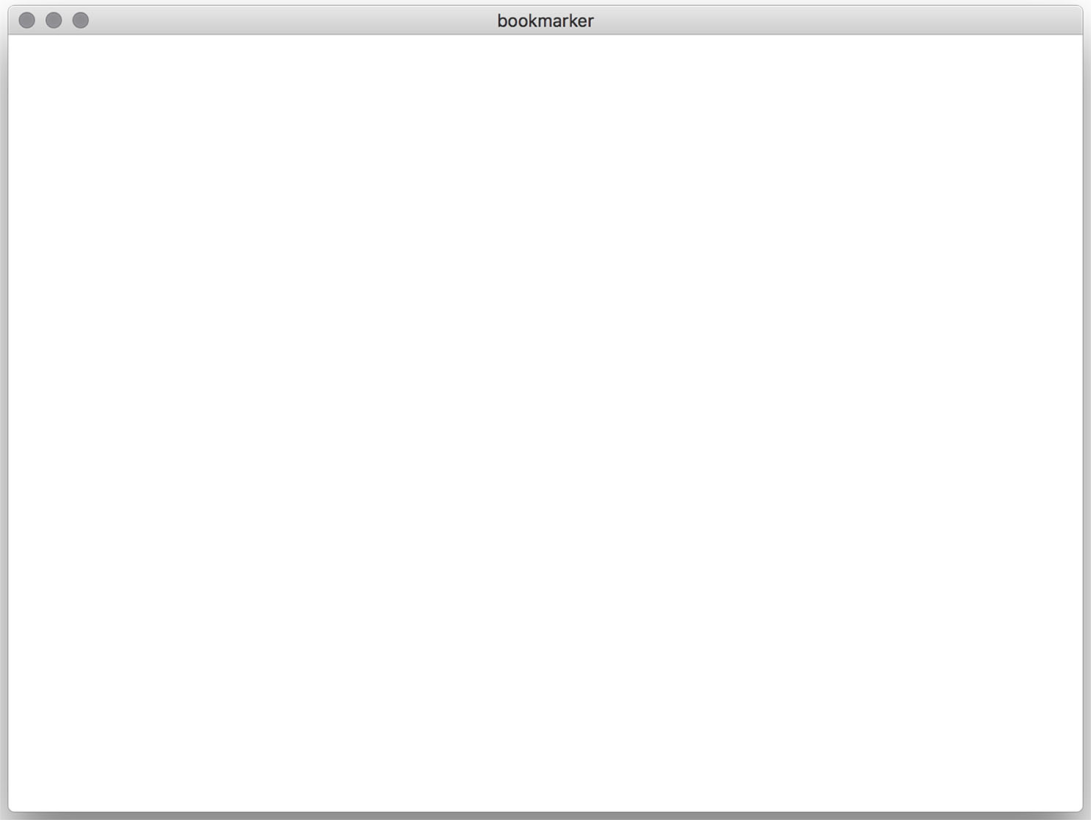
console.log('Hello from Electron.');

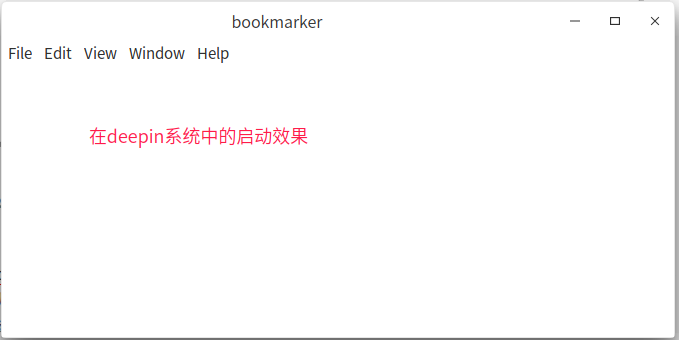
**//应用程序准备就绪后，创建一个浏览器窗口,并将其分配给在顶级作用域中创建的变量**

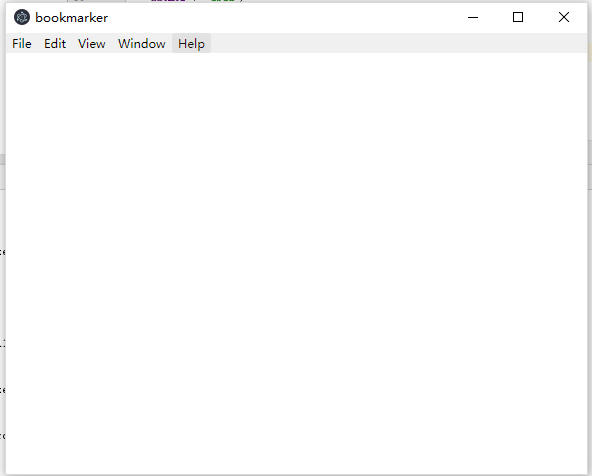
mainWindow = new BrowserWindow();

});

我们在ready事件监听器之外声明了mainWindow。JavaScript使用函数范围。如果我们在事件监听器中声明了mainWindow，则mainWindow将有资格进行垃圾收集，因为分配给ready事件的函数已经运行完成。如果收集垃圾，我们的窗户会神秘地消失。如果我们运行此代码，我们会在屏幕中央看到一个不起眼的小窗口，如图2.5所示。







**图2.5 没有加载HTML文档的空BrowserWindow(译注:依次为macOS,deepin,Windows10中的效果,后两个图后译者追加)**

这是一个窗口，但看起来并不多。下一步是将HTML页面加载到我们创建的BrowserWindow实例中。所有BrowserWindow实例都有一个webContents属性，它具有几个有用的功能，例如将HTML文件加载到渲染器进程窗口，从主进程发送消息到渲染器进程，将页面打印到PDF或打印机等等更多。现在，我们最关心的是将内容加载到我们刚刚创建的无聊窗口中。

我们需要加载一个HTML页面，因此在项目的app目录中创建一个index.html。让我们将以下内容添加到HTML页面，使其成为有效的文档。

**清单2.5 创建index.html：./app/index.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<meta http-equiv="Content-Security-Policy"

content="

default-src 'self';

script-src 'self' 'unsafe-inline'; connect-src \*

"

>

<meta name="viewport" content="width=device-width,initial-scale=1">

<title>Bookmarker</title>

</head>

<body>

<h1>Hello from Electron</h1>

</body>

</html>

它很简单，但它完成了工作，并为构建它提供了良好的基础。我们将以下内容添加到app /main.js，以告知渲染器进程在我们之前创建的窗口中加载此HTML文档。

**清单2.6 将HTML文档加载到主窗口：./app/main.js**

app.on('ready', () => {

console.log('Hello from Electron.');

mainWindow = new BrowserWindow();

**//告诉浏览器窗口加载与主进程位于同一目录中的HTML文件**

//mainWindow.webContents.loadFile('index.html'); //原代码已不能工作

mainWindow.webContents.loadFile('./app/index.html'); **//2.0.4与5.0.2版本下工作**

});

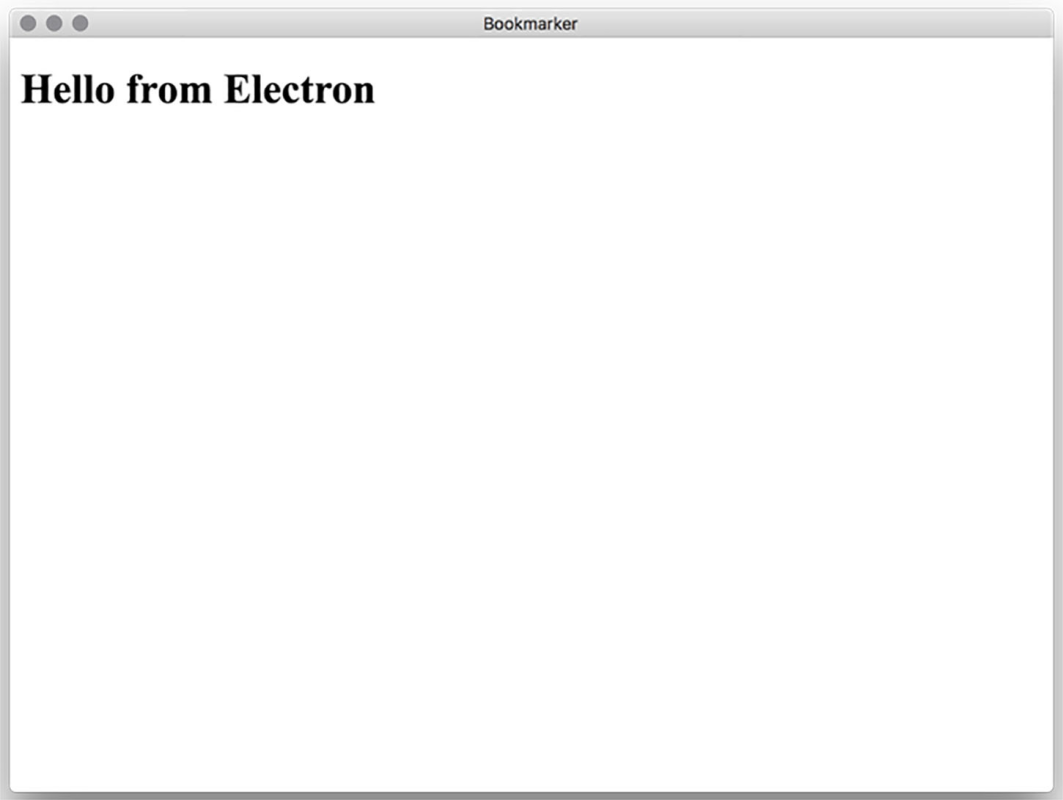
我们使用file://协议和\_dirname变量，它在Node中是全局可用的。\_dirname是执行Node进程的目录的完整路径。就我而言，\_dirname扩展为/Users/stevekinney/ Projects/bookmarker/app。这就像在macOS和Linux中键入pwd或在Windows中键入chdir。

现在，我们可以使用npm start启动我们的应用程序并观察它加载我们的新HTML文件。如果一切顺利，你应该看到类似于图2.6的东西。

###### *从渲染器进程装载代码*

从渲染器进程加载的HTML文件中，我们可以加载我们可能需要的任何其他文件，就像在传统的基于浏览器的Web应用程序中一样,即<script>和<link>标记。

是什么让Electron与我们在浏览器中习惯的不同之处在于我们可以访问所有Node,甚至可以从我们通常认为的“客户端”中获取。这意味着我们可以使用require或甚至是Node-only对象和像\_dirname或进程模块这样的变量。同时，我们也提供所有浏览器API。我们只能在客户端上做什么以及我们只能在服务器上做什么之间的划分开始逐渐消失。



**图2.6 加载了简单HTML文档的BrowserWindow (deepin中没有按预期显示内容)**

让我们来看看这个实际情况。\_dirname在传统浏览器环境中不可用，并且Node中没有document或alert。但在Electron中，我们可以将它们无缝地结合使用。让我们在页面上添加一个按钮。

**清单2.7 向HTML文档添加按钮：./app/index.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<meta http-equiv="Content-Security-Policy" content="

default-src 'self';

script-src 'self' 'unsafe-inline'; connect-src \*

"

>

<meta name="viewport" content="width=device-width,initial-scale=1">

<title>Bookmarker</title>

</head>

<body>

<h1>Hello from Electron</h1>

<p>

**<!-- 这是我们的新按钮。 -->**

<button class="alert">Current Directory</button>

</p>

</body>

</html>

现在我们有了按钮，让我们添加一个事件监听器，它提醒我们运行应用程序的当前目录。

**清单2.8 在浏览器上下文中添加带有Node.js全局的脚本：./app/index.html**

<script>

const button = document.querySelector('.alert');

button.addEventListener('click', () => {

**//单击按钮时，使用浏览器alert显示Node全局变量**

alert( dirname);

});

</script>

alert()仅在浏览器中可用。\_dirname仅在Node中可用。当我们点击按钮时，我们会看到Node和Chromium在甜蜜，甜蜜的和谐中共同工作，如图2.7所示。

(译注: d前面是双下划线,在作者2.0.4的版本下正常,在5.0.2版本下需要修改main.js代码才可以有同样的效果:

mainWindow = new BrowserWindow(**{**

// 在5.0.2下必须加此配置才会得到和作者一样的预期效果

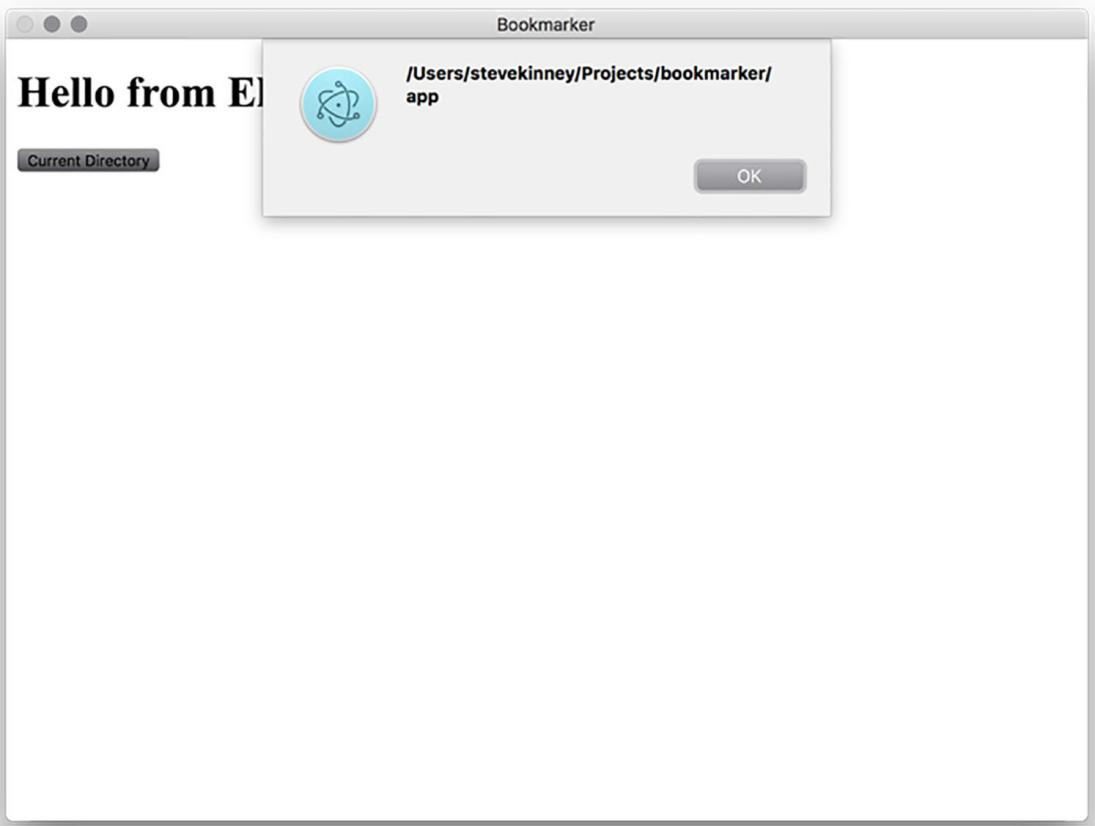
**webPreferences: {**

**nodeIntegration: true**

**}**

**}**);

)



**图2.7 BrowserWindow从渲染器进程的上下文中执行JavaScript。**

###### *渲染器进程中必需的文件*

在我们的HTML文件中编写代码显然有效，但可能不难想象我们的代码可能会增长到这种方法不再可行的情况。我们可以添加带有src属性的脚本标签来引用其他文件，但这也很快变得麻烦。

这是Web开发变得棘手的地方。尽管模块已添加到ECMAScript规范中，但目前没有任何浏览器具有模块系统的工作实现。在客户端上，我们可能会考虑某种构建工具，如Browserify（http://browserify.org）或模块捆绑器，webpack，以及可能像Gulp或Grunt这样的任务运行器。

我们可以使用Node的模块系统而无需额外配置。让我们将这些<script>标记内的所有代码移动到我们当前的empty-app/renderer.js文件中。现在我们只需要引用renderer.js就可以替换<script>标签内的内容。

**清单2.9 从renderer.js加载JavaScript：./ app/index.html**

<script>

//使用Node的require函数将其他JavaScript模块加载到渲染器进程中

require('./renderer');

</script>

如果我们启动我们的应用程序，您将看到它的功能没有改变。每件事情仍然可行。这在软件开发中很少发生。让我们在继续之前简单地品味一下这种感觉。

###### *在渲染器进程中添加样式*

当我们在Electron应用程序中引用样式表时，很少会出现意外情况。之后，我们讨论使用Sass和Less with Electron。在Electron应用程序中添加样式表与使用传统Web应用程序时没有太大区别。也就是说，一些细微差别值得一提。

让我们首先将style.css文件添加到我们的app目录。我们将以下内容添加到style.css中。

**清单2.10 添加基本样式：./app/style.css**

html {

box-sizing: border-box;

}

\*, \*:before, \*:after { box-sizing: inherit;

}

body, input {

/\*使用运行页面的操作系统的默认系统字体 \*/

font: menu;

}

最后一个声明可能看起来有点陌生。它是Chromium独有的，允许我们在CSS中使用系统字体。这种能力对于使我们的应用程序适应其原生兄弟姐妹非常重要。在macOS上，它是使用San Francisco的唯一方式，旧版本是El Capitan 10.11及更高版本附带的系统字体。

在我们的Electron应用程序中使用CSS时，我们应该考虑另一个重要的区别。我们的应用程序仅在我们随应用程序提供的Chromium版本中运行。我们不必担心跨浏览器支持或传统兼容性。如第1章所述，Electron附带了相对较新的Chromium版本。这意味着我们可以自由使用flexbox和CSS变量等技术。

我们像在传统的浏览器环境中一样引用我们的新样式表，然后将以下内容添加到index.html的<head>部分。我将包含用于链接到样式表的HTML标记,因为在我作为Web开发人员的20年中，我仍然记得第一次尝试时如何做到这一点。

**清单2.11引用HTML文档中的样式表：./app/ index.html**

<link rel="stylesheet" href="style.css" type="text/css">

### *实现UI*

我们首先使用UI所需的标签更新index.html。

**清单2.12 为应用程序的UI添加标记：./app/index.html**

<h1>Bookmarker</h1>

<div class="error-message"></div>

<section class="add-new-link">

<form class="new-link-form">

<input type="url" class="new-link-url" placeholder="URL"size="100" required>

<input type="submit" class="new-link-submit" value="Submit" disabled>

</form>

</section>

<section class="links"></section>

<section class="controls">

<button class="clear-storage">Clear Storage</button>

</section>

我们有一个用于添加新链接的部分，一个用于显示所有精彩链接的部分，以及一个用于清除所有链接并重新开始的按钮。应用程序中的<script>标签应该像我们在本章前面提到的那样，但是如果不是这样的话，那么这里应该是这样的：

<script> require('./renderer');

</script>

随着我们的标签到位，我们现在可以将注意力转向功能。让我们清除app/renderer.js中可能存在的任何内容并重新开始。在我们共同的时间里，我们需要使用我们添加到标签中的一些元素，所以让我们首先查询那些选择器并将它们缓存到变量中。将以下内容添加到app/renderer.js。

**清单2.13 缓存DOM元素选择器：./app/renderer.js**

const linksSection = document.querySelector('.links');

const errorMessage = document.querySelector('.error-message'); const newLinkForm = document.querySelector('.new-link-form'); const newLinkUrl = document.querySelector('.new-link-url');

const newLinkSubmit = document.querySelector('.new-link-submit'); const clearStorageButton = document.querySelector('.clear-storage');

如果你回顾一下清单2.12，你会注意到我们在输入元素的类型属性设置为标记中的“url”。如果内容与有效的URL模式不匹配，Chromium会将该字段标记为无效。我们可以设置元素的有效和无效状态，甚至可以使用JavaScript检查其状态。很遗憾，我们无法访问Chrome或Firefox中的内置错误消息弹出窗口。这些弹出窗口不是Chromium内容模块的一部分，因此不是Electron的一部分。现在，我们默认禁用启动按钮，然后每次用户在URL字段中键入字母时检查是否有有效的URL模式。

如果用户提供了有效的URL，那么我们按下该提交按钮上的开关并允许他们提交URL。我们将此代码添加到app/renderer.js。

**清单2.14 添加一个事件监听器来启用提交按钮：./ app/renderer.js**

newLinkUrl.addEventListener('keyup', () => {

**//当用户键入输入字段时，它使用Chromium的ValidityState API来确定输入是否有效。如果是这样，请从提交按钮中删除已禁用的属性。**

newLinkSubmit.disabled = !newLinkUrl.validity.valid;

});

现在也是添加小帮助函数以清除URL字段内容的好时机。在一个完美的世界中，只要我们成功存储了链接，我们就会调用它。

**清单2.15 添加一个帮助函数来清除表单输入：./app/renderer.js**

const clearForm= () => {

**//通过将其值设置为null来清除新链接输入字段的值。**

newLinkUrl.value = null;

};

当用户提交链接时，我们希望浏览器发出对该URL的请求，然后获取响应正文，解析它，找到title元素，从该title元素中获取文本，将书签的标题和URL存储在 localStorage，然后,最后使用书签更新页面。

###### *在Electron中制作跨域请求*

您可能会或可能不会感觉到脖子后面的一些毛发开始引起注意。你甚至可能在想，”这个计划无法奏效。您无法向第三方服务器发出请求。浏览器不允许这样做。“

通常情况下，你是对的。在传统的基于浏览器的应用程序中，不允许您的客户端代码向其他服务器发出请求。通常，您的客户端代码向您的服务器发出请求，而该请求又将请求代理到第三方服务器。当它听到回来时，它会将响应代理回客户端。我们在第1章中讨论了这背后的一些推理。

Electron拥有Node服务器的所有功能以及浏览器的所有功能。这意味着我们可以自由地进行跨域请求，而无需服务器阻碍。

在Electron中编写此应用程序的另一个好处是，我们能够使用即将推出的Fetch API向远程服务器发出请求。Fetch API使我们不必手动设置XMLHttpRequests，并提供了一个很好的，基于promise的接口来处理我们的请求。在撰写本文时，Fetch在主流浏览器中的支持有限。也就是说，它在当前版本的Chromium中得到了全力支持，这意味着我们可以使用它。

我们在表单中添加一个事件监听器，以便在提交表单时立即采取行动。我们没有服务器，因此我们需要确保阻止发出请求的默认操作。我们通过阻止默认操作来完成此操作。我们还会缓存URL输入字段的值以供将来使用。

**清单2.16 向提交按钮添加一个事件监听器：./app/renderer.js**

newLinkForm.addEventListener('submit', (event) => {

**//告诉Chromium不要触发HTTP请求，这是表单提交的默认操作**

event.preventDefault();

**// 在新链接输入字段中抓取URL。我们很快就会需要这个值。**

const url = newLinkUrl.value;

// More code to come…

});

Fetch API可用作全局可用的提取变量。获取URL将返回promise对象，该对象将在浏览器完成获取远程资源时完成。使用此promise对象，我们可以根据是否决定获取网页，图像或其他类型的内容来区别对待响应。在这种情况下，我们正在获取网页，因此我们将响应转换为文本。我们从事件监听器中的以下代码开始。

**清单2.17 使用Fetch API来请求远程资源：./app/renderer.js**

**//使用Fetch API获取提供的URL的内容**

fetch(url)

.then(response => response.text());**//将响应解析为纯文本**

Promises是可以链接的。我们可以获取先前承诺的返回值，然后再调用另一个。另外，response.text()本身返回一个promise。我们的下一步将是获取我们收到的大块标记并解析它以便遍历它并找到<title>元素。

###### *解析响应*

Chromium提供了一个解析器，它将为我们执行此操作，但我们需要实例化它。在app/renderer.js的顶部，我们创建了一个DOMParser实例并将其存储起来供以后使用。

**清单2.18 实例化DOMParser：./app/renderer.js**

**//创建DOMParser实例。我们将在获取提供的URL的文本内容后使用它。**

const parser = new DOMParser();

让我们设置一对辅助函数来解析响应并为我们找到标题。

**清单2.19 添加解析响应和查找标题的函数：./app/renderer.js**

const parseResponse = (text) => {

**//从URL获取HTML字符串并将其解析为DOM树。**

return parser.parseFromString(text, 'text/html');

}

**// 遍历DOM树以查找<title>节点。**

const findTitle = (nodes) =>{

return nodes.querySelector('title').innerText;

}

我们现在可以将这两个步骤添加到我们的承诺链中。

**清单2.20 解析响应并在获取页面时查找标题：./app/renderer.js**

fetch(url)

.then(response => response.text())

.then(parseResponse)

.then(findTitle);

此时，app/renderer.js中的代码如下所示。

**清单2.21 app/renderer.js的当前内容**

const parser = new DOMParser();

const linksSection = document.querySelector('.links');

const errorMessage = document.querySelector('.error-message'); const newLinkForm = document.querySelector('.new-link-form'); const newLinkUrl = document.querySelector('.new-link-url');

const newLinkSubmit = document.querySelector('.new-link-submit'); const clearStorageButton = document.querySelector('.clear-storage');

newLinkUrl.addEventListener('keyup', () => { newLinkSubmit.disabled = !newLinkUrl.validity.valid;

});

newLinkForm.addEventListener('submit', (event) => { event.preventDefault();

const url = newLinkUrl.value; fetch(url)

.then(response => response.text())

.then(parseResponse)

.then(findTitle)

});

const clearForm = () => { newLinkUrl.value = null;

}

const parseResponse = (text) => {

return parser.parseFromString(text, 'text/html');

}

const findTitle = (nodes) => {

return nodes.querySelector('title').innerText;

}

###### *使用Web存储API存储响应*

localStorage是一个简单的键/值存储，内置于浏览器中并在会话之间保留。您可以在任意键下存储字符串和数字等简单数据类型。让我们设置另一个辅助函数，它将从标题和URL中创建一个简单的对象，使用内置的JSON库将其转换为字符串，然后使用URL作为键存储它。

**清单2.22 创建一个在本地存储中保留链接的函数：./app/renderer.js**

const storeLink = (title, url) => {

localStorage.setItem(url, JSON.stringify({ title: title, url: url }));

};

我们的新storeLink函数需要标题和URL来完成其工作，但之前的promise只返回标题。我们使用箭头函数将对storeLink的调用包装在一个匿名函数中，该函数可以访问范围内的url变量。如果成功，我们也会清除表格。

**清单2.23 在获取远程资源时存储链接并清除表单：**

fetch(url)

.then(response => response.text())

.then(parseResponse)

.then(findTitle)

.then(title =>

**//将标题和URL存储到localStorage。**

storeLink(title, url))

.then(clearForm);

###### *显示请求结果*

存储链接是不够的。我们还想将它们显示给用户。这意味着我们需要创建功能来遍历我们存储的所有链接，将它们转换为DOM节点，然后将它们添加到页面中。

让我们从localStorage获取所有链接的能力开始。如果你还记得，localStorage是一个键/值存储。我们可以使用Object.keys来获取对象的所有键。我们必须给自己另一个帮助函数来获取localStorage的所有链接。这不是什么大不了的事，因为无论如何我们需要将它们从字符串转换回真实对象。让我们定义一个getLinks函数。

**清单2.24 创建一个从本地存储获取链接的函数：./ app/renderer.js**

const getLinks = () => {

return

**//获取当前存储在localStorage中的所有键的数组**

Object.keys(localStorage)

**//对于每个键，获取其值并将其从JSON解析为JavaScript对象**

.map(key => JSON.parse(localStorage.getItem(key)));

}

接下来，我们将这些简单对象转换为标签，以便稍后将它们添加到DOM中。我们创建了一个简单的convertToElement帮助器，它也可以处理这个问题。重要的是要提到我们的convertToElement函数有点天真并且不会尝试清理用户输入。从理论上讲，您的应用程序很容易受到脚本注入攻击。这有点超出了本章的范围，因此我们只做了最小的工作，将这些链接呈现在页面上。我会把它作为练习留给读者来保证这个功能。

**清单2.25 创建一个从链接数据创建DOM节点的函数：./app/renderer.js**

const convertToElement = (link) => { return `

<div class="link">

<h3>${link.title}</h3>

<p>

<a href="${link.url}">${link.url}</a>

</p>

</div>

`;

};

最后，我们创建一个renderLinks()函数，它调用getLinks，连接它们，使用convertToElement()转换集合，然后替换页面上的linksSection元素。

**清单2.26 创建一个函数来渲染所有链接并将它们添加到DOM：./app/renderer.js**

const renderLinks = () => {

**//将所有链接转换为HTML元素并将它们组合在一起**

const linkElements = getLinks().map(convertToElement).join('');

**//用组合的链接元素替换链接部分的内容**

linksSection.innerHTML = linkElements;

};

我们现在可以添加这个最后一步到我们的promise链。

**清单2.27 获取远程资源后的渲染链接：./app/renderer.js**

fetch(url)

.then(response => response.text())

.then(parseResponse)

.then(findTitle)

.then(title => storeLink(title, url))

.then(clearForm)

.then(renderLinks);

我们还通过在顶级范围内调用renderLinks()来在页面最初加载时呈现所有链接。

**清单2.28 加载和渲染链接：./app/render.js**

**// 一旦页面加载，就调用我们之前创建的renderLinks()函数**

renderLinks();

使用promise与将功能分解为命名辅助函数的优点之一是，通过获取外部网页，解析外部网页，存储结果以及重新呈现链接列表，可以非常清楚我们的代码在做什么。

我们需要完成简单应用的所有功能的最后一件事是连接Clear Storage按钮。我们在localStorage上调用clear方法，然后清空linksSection中的列表。

**清单2.29 连接Clear Storage按钮：./ app/renderer.js**

clearStorageButton.addEventListener('click', () => {

**//清空localStorage中的所有链接**

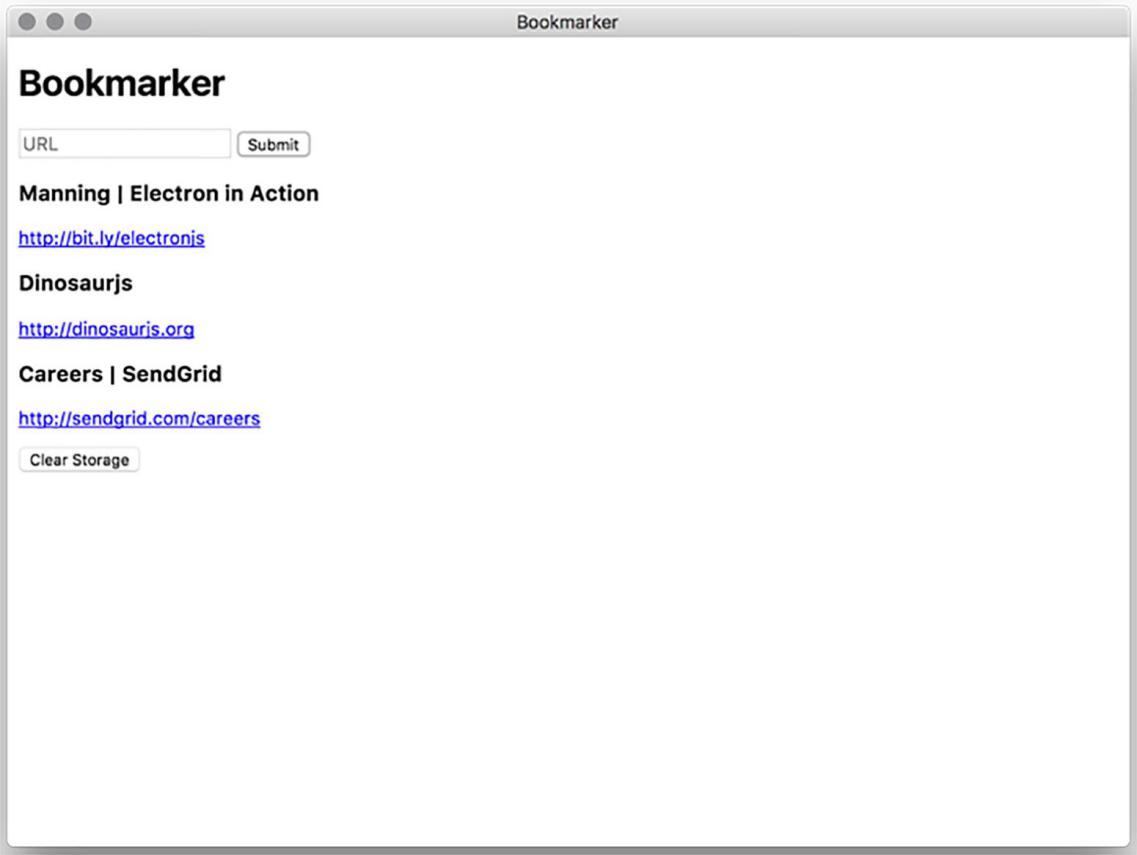
localStorage.clear();

**//从UI中删除链接**

linksSection.innerHTML = '';

});

使用Clear Storage按钮，我们似乎已经拥有了大部分功能。我们的应用程序现在看起来像图2.8。此时，我们的渲染器进程代码应该类似于清单2.30。



**图2.8 完整的Bookmarker应用程序**

**清单2.30 渲染进程抓取,存储和渲染链接：./ app/renderer.js**

const parser = new DOMParser();

const linksSection = document.querySelector('.links');

const errorMessage = document.querySelector('.error-message'); const newLinkForm = document.querySelector('.new-link-form'); const newLinkUrl = document.querySelector('.new-link-url');

const newLinkSubmit = document.querySelector('.new-link-submit'); const clearStorageButton = document.querySelector('.clear-storage');

newLinkUrl.addEventListener('keyup', () => { newLinkSubmit.disabled = !newLinkUrl.validity.valid;

});

newLinkForm.addEventListener('submit', (event) => { event.preventDefault();

const url = newLinkUrl.value; fetch(url)

.then(response => response.text())

.then(parseResponse)

});

.then(findTitle)

.then(title => storeLink(title, url))

.then(clearForm)

.then(renderLinks);

clearStorageButton.addEventListener('click', () => { localStorage.clear();

linksSection.innerHTML = '';

});

const clearForm = () => { newLinkUrl.value = null;

}

const parseResponse = (text) => {

return parser.parseFromString(text, 'text/html');

}

const findTitle = (nodes) => {

return nodes.querySelector('title').innerText;

}

const storeLink = (title, url) => {

localStorage.setItem(url, JSON.stringify({ title: title, url: url }));

}

const getLinks = () => {

return Object.keys(localStorage)

.map(key => JSON.parse(localStorage.getItem(key)));

}

const convertToElement = (link) => {

return `<div class="link"><h3>${link.title}</h3>

<p><a href="${link.url}">${link.url}</a></p></div>`;

}

const renderLinks = () => {

const linkElements = getLinks().map(convertToElement).join(''); linksSection.innerHTML = linkElements;

}

renderLinks();

###### *不开心的Path*

到目前为止，一切似乎都有效。我们的应用程序从外部网页获取标题，在本地存储链接，在页面上呈现链接，并在我们要求时从页面中清除它们。

但如果出现问题会怎样？ 如果我们给它一个无效链接会怎么样？ 如果请求超时，会发生什么？ 我们将处理两种最可能的情况：用户提供的URL在输入字段上通过验证检查但实际上不是有效的，并且当URL有效但服务器返回400或500级错误时。

我们添加的第一件事是处理任何错误的能力。Promise链支持catch方法，该方法在未被捕获的错误的情况下被调用。我们在此事件中定义另一个辅助方法。

**清单2.31 显示错误消息：./app/renderer.js**

const handleError = (error, url) => {

**//如果提取链接失败，则设置错误消息元素的内容**

errorMessage.innerHTML = `

There was an issue adding "${url}": ${error.message}

`.trim();

**//5秒后清除错误消息**

setTimeout(() => errorMessage.innerText = null, 5000);

}

我们可以将其添加到链中。我们使用另一个匿名函数来传递带有错误消息的URL。这主要是为了提供更好的错误消息。如果您不想在错误消息中包含URL，则没有必要。

**清单2.32 获取,解析和渲染链接时捕获错误：./app/renderer.js**

fetch(url)

.then(response => response.text())

.then(parseResponse)

.then(findTitle)

.then(title => storeLink(title, url))

.then(clearForm)

.then(renderLinks)

.catch(error => handleError(error, url));

**如果此链中的任何承诺拒绝或抛出错误，则捕获错误并将其显示在UI中**

我们还在链条的早期添加了一个步骤，检查请求是否成功。如果是这样，它会将请求传递给链中的下一个承诺。如果它不成功，那么我们抛出一个错误，它绕过链中其余的promise并直接跳到handleError()步骤。这里有一个我没有处理的边缘情况：如果无法建立网络连接，则从Fetch API返回的promise会完全拒绝。我将这作为练习留给读者来处理，因为我们在本书中有很多内容和有限数量的页面要做。如果状态代码在400或500中，response.ok将是假的 范围。

**清单2.33 验证来自远程服务器的响应：./app/renderer.js**

const validateResponse = (response) => {

**//如果响应成功，则将其传递给下一个承诺。**

if (response.ok) { return response; }

**//如果请求收到400或500系列响应，则会引发错误。**

throw new Error(`Status code of ${response.status}

${response.statusText}`);

}

如果没有错误，此代码将传递响应对象。但是如果出现错误，它会抛出一个错误，由handleError()捕获并相应处理。

**清单2.34 将validateResponse()添加到链中：./ app/renderer.js**

fetch(url)

**.then(validateResponse)**

.then(response => response.text())

.then(parseResponse)

.then(findTitle)

.then(title => storeLink(title, url))

.then(clearForm)

.then(renderLinks)

.catch(error => handleError(error, url));

###### *意外Bug*

我们还没有走出困境,如果一切顺利，我们也会遇到问题。如果我们单击应用程序中的一个链接会发生什么？ 也许是不可思议的，它涉及到这个环节。我们的Electron应用程序的Chromium部分认为它是一个Web浏览器，因此它可以实现Web浏览器最好的功能,它可以进入页面。除了我们的应用程序不是一个真正的Web浏览器。它缺少重要的东西，如后退按钮或地址栏。如果我们点击我们的应用程序中的任何链接，我们就会陷入困境。我们唯一的选择是杀死应用程序并从头开始。解决方案是在真实的浏览器中打开链接。但这提出了一个问题，哪个浏览器？ 我们如何判断用户将其设置为默认浏览器？ 我们肯定不想采取任何幸运的猜测，因为我们不知道用户安装了什么浏览器，没有人喜欢看到错误的应用程序开始只是因为他们点击了一个链接。

Electron附带shell模块，提供与高级桌面集成相关的一些功能。shell模块可以询问用户的操作系统他们喜欢哪种浏览器，并将URL传递给该浏览器以打开。让我们首先导入Electron并在app/renderer.js顶部存储对其shell模块的引用。

**清单2.35 需要Electron的shell模块：./app/renderer.js**

const {shell} = require('electron');

我们可以使用JavaScript来确定我们要在应用程序中处理哪些URL以及我们要将哪些URL传递给默认浏览器。在我们简单的应用中，区别很简单。我们希望在默认浏览器中打开所有链接。在此应用程序中添加和删除了链接，因此我们在linksSection元素上设置了一个事件监听器，并允许点击事件冒泡。如果目标元素具有href属性，我们会阻止默认操作并将URL传递给默认浏览器。

**清单2.36 在用户默认浏览器中打开链接：/app/renderer.js**

linksSection.addEventListener('click', (event) => {

**//通过查找href属性来检查单击的元素是否为链接**

if (event.target.href) {

**//如果是链接，请不要正常打开它。**

event.preventDefault();

**//使用Electron的shell模块在用户的默认浏览器中打开链接**

shell.openExternal(event.target.href);

}

});

通过相对简单的更改，我们的代码按预期运行。单击链接将在用户的默认浏览器中打开该页面。我们有一个简单但功能齐全的桌面应用程序。

我们完成的代码应该类似于以下代码示例。您可以按不同的顺序使用您的功能。

**清单2.37 已完成的应用程序：./app/renderer.js**

const {shell} = require('electron'); const parser = new DOMParser();

const linksSection = document.querySelector('.links');

const errorMessage = document.querySelector('.error-message'); const newLinkForm = document.querySelector('.new-link-form'); const newLinkUrl = document.querySelector('.new-link-url');

const newLinkSubmit = document.querySelector('.new-link-submit'); const clearStorageButton = document.querySelector('.clear-storage');

newLinkUrl.addEventListener('keyup', () => { newLinkSubmit.disabled = !newLinkUrl.validity.valid;

});

newLinkForm.addEventListener('submit', (event) => { event.preventDefault();

const url = newLinkUrl.value; fetch(url)

.then(response => response.text())

.then(parseResponse)

.then(findTitle)

.then(title => storeLink(title, url))

.then(clearForm)

.then(renderLinks)

.catch(error => handleError(error, url));

});

clearStorageButton.addEventListener('click', () => { localStorage.clear();

linksSection.innerHTML = '';

});

linksSection.addEventListener('click', (event) => { if (event.target.href) {

event.preventDefault(); shell.openExternal(event.target.href);

}

});

const clearForm = () => { newLinkUrl.value = null;

};

const parseResponse = (text) => {

return parser.parseFromString(text, 'text/html');

};

const findTitle = (nodes) => {

return nodes.querySelector('title').innerText;

};

const storeLink = (title, url) => {

localStorage.setItem(url, JSON.stringify({ title: title, url: url }));

};

const getLinks = () => {

return Object.keys(localStorage)

.map(key => JSON.parse(localStorage.getItem(key)));

};

const convertToElement = (link) => {

return `<div class="link"><h3>${link.title}</h3>

<p><a href="${link.url}">${link.url}</a></p></div>`;

};

const renderLinks = () => {

const linkElements = getLinks().map(convertToElement).join(''); linksSection.innerHTML = linkElements;

};

const handleError = (error, url) => { errorMessage.innerHTML = `

There was an issue adding "${url}": ${error.message}

`.trim();

setTimeout(() => errorMessage.innerText = null, 5000);

};

const validateResponse = (response) => { if (response.ok) { return response; }

***本章小结* 43**

throw new Error(`Status code of ${response.status}

${response.statusText}`);

}

renderLinks();

### *本章小结*

* + - * Electron不推荐或强制特定的项目结构。
      * Electron使用npm的package.json清单来确定它应该加载为主进程的文件。
      * 我们可以使用npm init从样板文件中生成package.json。
      * 我们通常在我们工作的每个项目中本地安装Electron。这使我们可以拥有项目特定版本的Electron。
      * 我们可以在Electron应用程序中使用require（'electron'）来访问Electron特定的模块和功能。
      * 应用程序模块管理我们的Electron应用程序的生命周期。
      * 主进程无法渲染UI。
      * 我们可以使用BrowserWindow模块从主进程创建渲染器进程。
      * Electron允许我们直接从浏览器向第三方服务器发出请求而无需中间服务器。传统的Web应用程序不允许这样做。
      * 在我们退出并重新打开应用程序时，localStorage中的数据将允许它持久化。

# *第二部分*

*使用Electron构建跨平台应用程序*

在第一部分中，我谈了一些关于是什么让Electron变得有趣和特别，但我们在构建Electron应用程序时的第一次尝试是故意简单的,我承认有点没有灵感。在第二部分,它占据了本书的最大份额,我们将开始深入研究Electron更具吸引力的能力。在本节中，我们将构建三个应用程序：Markdown文本编辑器，具有实时预览和直接访问文件系统，剪辑管理器位于操作系统的菜单栏或系统托盘中，可以读取和写入系统剪贴板，以及使用React构建的旅行装箱单，可以直接读取和写入本机数据库。

在第二部分，我试着通过让你构建足够简单的应用程序以便你能够在短时间内理解它们，同时又具有足够的复杂性以便我们遇到一些等待我们的大问题 ，更复杂的应用程序。

在第3章中，我们将通过实现UI为Fire Sale（我们的Markdown编辑器）奠定基础。我们还将研究调试Electron应用程序的细节。在第4章中，我们将探讨主Node.js进程与一个或多个浏览器窗口之间的关系,这些窗口称为渲染进程,您可以创建它。我们将读取和写入文件系统，并在进程之间来回发送数据。在第5章中，我将介绍如何管理多个窗口并深入研究Electron的进程间通信模型。

第6章介绍了与原生操作系统集成的一些细节。我们将确定文件是否有未保存的更改，并相应地更新操作系统提供的标题栏。我们会将文档附加到操作系统的最近文档列表中，并监听另一个应用程序是否更改了Fire Sale中当前打开的文件。第7章探讨了如何构建原生应用程序和上下文菜单，这通常不是Web和Node.js开发人员经常遇到的。在第8章中，我们将通过实现与主机操作系统的额外集成以及通过在应用程序状态发生变化时启用和禁用菜单项来动态更新菜单来将完成接触放在Fire Sale上。

在第9章中，我们将着手在以前没有Web开发者的地方构建应用程序：macOS中的菜单栏和Windows中的系统托盘。Clip-master是一个简单的应用程序，允许您从系统剪贴板中读取文本并将其存储在内存中。选择某个项目后，会将其写回剪贴板以便于粘贴。这是我在撰写本书时使用的一个应用程序。在第10章中，我们将烧掉整个应用程序并从头开始，从而在此过程中为其提供更强大的UI。

在第11章之前，我们使用一组非常有限的工具来操作DOM。当您阅读第11章前面的章节时，您可能会对如何集成您最喜爱的前端框架有很大的想法。当我们看到像Babel，TypeScript或Sass这样的实现工具比你想象的更容易时，你的耐心会得到回报。我们将构建一个名为Jetsetter的简单列表管理器，它可以跟踪您在下次冒险之前需要打包的项目。我将仅使用React，因为它具有最温和的学习曲线，但我向你保证，Ember，Angular和Vue也有很棒的库。

您可能使用了像webpack或Browserify这样的工具将npm中的依赖项导入应用程序。如果是这样，您可能会遇到一组模块，这些模块通常使用已编译的C++代码，这些模块是不受限制的。一般来说，浏览器只能使用JavaScript。但Electron应用程序不仅仅是浏览器应用程序。在第12章中，我们将采用Jetsetter并将其直接挂钩到可移植的SQLite数据库。我们将弄清楚如何为Electron附带的Node.js版本正确编译模块。我将在基于浏览器的IndexedDB数据库中实现持久存储，以帮助您确定您可能更喜欢自己的应用程序的方法。

第13章介绍使用Spectron的端到端测试，Spectron基于Selenium。在本章中，我将向您展示如何编写可以启动Electron应用程序并将其用于搭载的测试。到第二部分结束时，你会对Electron提供的大部分内容感到满意，创意果汁肯定会流淌。

*构建一个记事本应用*

***本章包括***

* 介绍我们将在接下来的几章中构建的应用程序
* 将我们的CSS样式表配置为更像原生应用程序
* 回顾Electron中主进程和渲染器进程之间的关系
* 实现主进程和染器进程的基本功能
* 在Electron中的渲染器进程中访问Chrome开发者工具

我们的书签管理器是一个很好的起点，但它只是触及了我们可以用Electron做的表面。在本章中，我们深入挖掘并为与用户操作系统建立更紧密联系的应用程序奠定基础。在接下来的几章中，我们将实现触发操作系统GUI，读取和写入文件系统以及访问剪贴板的功能。

我们正在构建一个简单的记事本编辑器，允许我们创建新的或打开现有的Markdown文件，将它们转换为HTML，并将HTML保存到文件系统和剪贴板。让我们将应用程序Fire Sale称为价格降价的唯一一个稍微聪明的游戏,因为它毕竟是Markdown编辑器。在本章的最后，我们将讨论在出现问题时可用于调试Electron应用程序的技术和工具。

### *定义我们的应用*

让我们从为我们简陋的小应用程序设定目标开始。我们的许多功能对于桌面应用程序来说似乎有点平庸，这就是重点。它们是桌面应用程序的标准票价，但完全超出了传统Web应用程序的能力范围，传统Web应用程序无法访问其独立浏览器选项卡之外的任何内容。我们的应用程序将包含两个窗格：一个用户可以编写或编辑Markdown的左窗格和一个显示用户的Markdown呈现为HTML的右窗格。在顶部，我们有一系列按钮，允许用户从文件系统加载文本文件，并将结果写入剪贴板或文件系统。

在我们的应用程序的第一阶段，我们基于图3.1中的线框构建UI。我们还可以在线框中添加额外的UI元素,随后是我们的应用程序 - 但是这是一个很好的起点。

**Some Very Important Markdown**

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

## Some Very Important Markdown

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Save File

…

Revert

Open File

New File

**图3.1 我们的应用程序的线框显示用户可以在左窗格中输入文本或从用户文件系统的文件加载文本。**

***奠定基础* 49**

在本章中，我们为我们的应用奠定了基础。我们创建项目的结构，安装我们的依赖项，设置我们的主进程和渲染器进程，构建我们的UI，并在用户在左窗格中输入文本时实现Markdown-to-HTML呈现。

我们将在接下来的几章中分阶段构建应用程序的其余部分。在每章中，您将下载我们的应用程序的当前状态。通过这种方式，您可以翻到涵盖您感兴趣的功能的章节，而无需从头开始构建整个应用程序。

在第一阶段，我们的应用程序将能够:

* + - 打开文件并将文件保存到文件系统
    - 从这些文件中获取Markdown内容
    - 将Markdown内容渲染为HTML
    - 将生成的HTML保存到文件系统中
    - 将生成的HTML写入剪贴板

在后面的章节中，我们的应用程序使用原生操作系统API跟踪最近打开的文档。我们可以将Markdown文件从Finder或Windows资源管理器拖到我们的应用程序中，让应用程序立即打开Markdown文件。当我们右键单击应用程序的不同区域时，我们的应用程序将拥有自己的自定义应用程序菜单以及自定义上下文菜单。

我们还利用特定于操作系统的功能，例如更新应用程序的标题栏以显示当前打开的文件以及自上次保存以来是否已更改。我们还实现了其他功能，例如，如果计算机上的某些其他应用程序在我们打开文件时更改了文件，则更新应用程序中的内容。

### *奠定基础*

文件结构如图3.2所示，与我们在前一章中为我们的书签管理器达成一致并用于我们的书签管理器的结构不足为奇。为了简单和清晰起见，我们继续熟悉Electron，我们在app/main.js中保留了主进程的所有代码，并在app/renderer.js中保存了我们单个渲染器进程的所有代码。我们将app文件夹存储在基于UNIX的操作系统上，以便我们可以快速生成它，如下面的清单所示。或者，您可以通过https://github.com/electron-in-action/firesale在GitHub上查看此项目的主分支。

LICENSE

index.html

README.md

**Project**

main.js

app

package.json

renderer.js

style.css

**图3.2 我们项目的结构**

**清单3.1 生成应用程序的文件结构**

mkdir app && touch app/index.html app/main.js app/renderer.js app/style.css

项目的各个部分是:

* + - index.html - 包含为UI提供结构的所有HTML标签
    - main.js -包含主进程的代码
    - renderer.js - 包含UI的交互性的所有代码
    - style.css -包含用于设置UI样式的CSS
    - package.json -包含所有依赖项，并在启动时加载主进程时指向Electron到main.js

为了简单起见，除了Electron之外，我们从两个依赖项开始作为运行时。我们使用一个名为marked的库来处理将Markdown转换为HTML的繁重工作。

要为此项目生成package.json，请运行npm init --yes。--yes标志允许您跳过上一章中的提示。生成package.json文件后，运行以下命令以安装必要的依赖项：

npm install electron marked --save

**译注:如你此时执行上面的指令,electron和marked的默认版本和作者是不一致的,译者为5.0.2/0.6.2,当然你可以带上版本后以求和作者一致1.4.8/0.3.6,例如: cnpm install electron@1.4.8 marked@0.3.6 --save**



**Electron读取package.json中的主条目，以确定哪个文件作为主进程运行。**

使用BrowserWindow加载一个或多个渲染器进程。

主进程

渲染进程

**图3.3 Electron启动时查找我们的单个主进程，该进程负责生成一个或多个负责显示我们UI的渲染器进程。**

### *引导(启动)应用*

我们的package.json中的主条目被配置为将index.js加载为我们的应用程序的主进程，如图3.3所示。我们需要将其调整为app/main.js。我们还需要启动渲染器进程以向用户显示应用程序的界面。在app/main.js中，让我们添加以下代码。

**清单3.2 Bootstrapping主进程：./ app/main.js**

const { app, BrowserWindow } = require('electron');

在顶层声明mainWindow，以便在“ready”事

件完成后不会将其收集为垃圾

let mainWindow = null;

app.on('ready', () => {

mainWindow = new BrowserWindow();

~~mainWindow.loadFile('index.html')~~; mainWindow.on('closed', () => {

使用默认属性创建新的BrowserWindow

mainWindow = null;

});

});

**窗口关闭时将进程设置回null**

**在我们刚刚创建的BrowserWindow实例中加载app/index.html**

这足以启动我们的应用程序。也就是说，没有太多事情发生，因为我们的主进程当前在渲染器进程中加载了一个空文件。

###### *实现UI*

实现必要数量的HTML和CSS以获得图3.1中线框的可行版本在Electron中相当容易，因为我们只需要支持一个浏览器，并且该浏览器支持Web平台提供的最新和最好的功能，如图所示 在图3.4中。

main.js

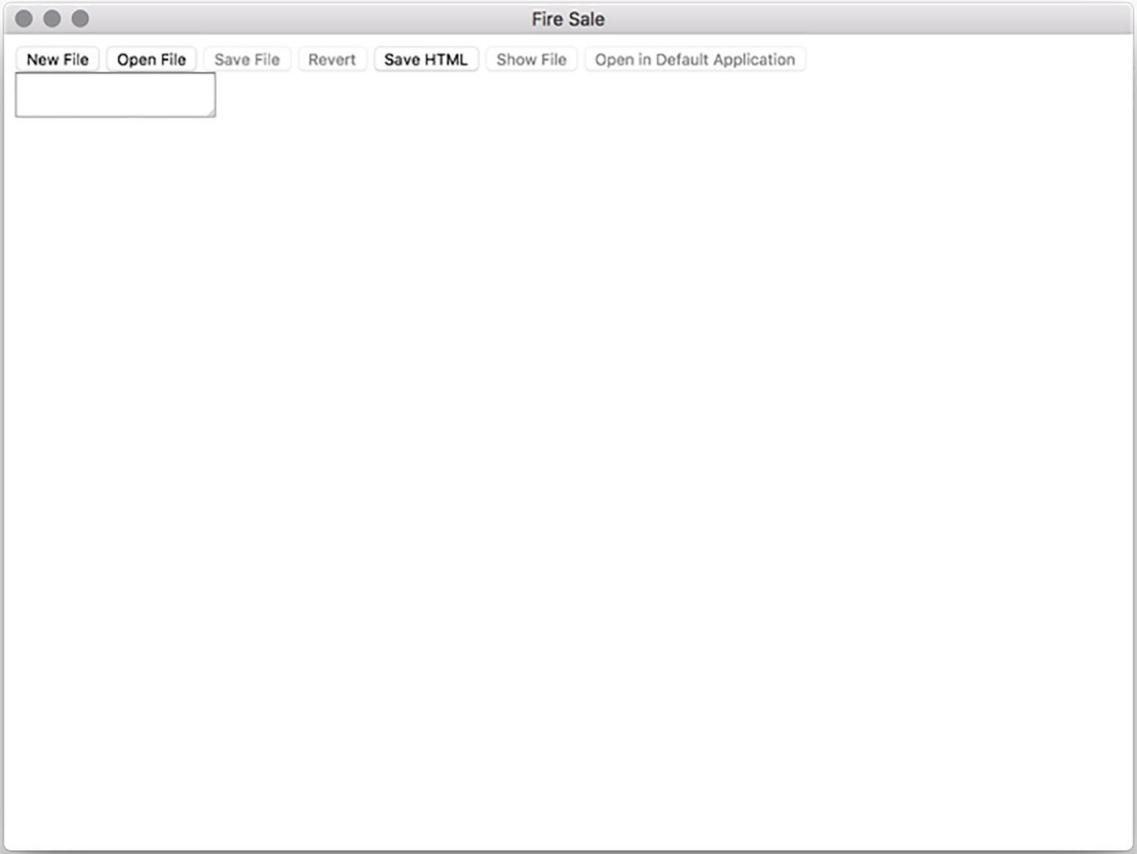
index.html

style.css

renderer.js

**图3.4主进程将创建一个渲染器进程并告诉它加载index.html，然后加载CSS和JavaScript就像在浏览器中一样。**

在index.html中，我们在清单3.3中添加标签以创建图3.5中的浏览器窗口。



**图3.5 无样式地开始我们的第一个Electron应用程序**

**清单3.3我们的应用程序的标签：./app/index.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1">

<title>Fire Sale</title>

<link rel="stylesheet" href="style.css" type="text/css">

</head>

<body>

<section class="controls">

<button id="new-file">New File</button>

<button id="open-file">Open File</button>

**.controls部分在顶部添加了打开和保存文件的按钮。我们稍后会为这些按钮添加功能。**

<button id="save-markdown" disabled>Save File</button>

<button id="revert" disabled>Revert</button>

<button id="save-html">Save HTML</button>

<button id="show-file" disabled>Show File</button>

<button id="open-in-default" disabled>Open in Default

➥ Application</button>

</section>

<section class="content">

<label for="markdown" hidden>Markdown Content</label>

<textarea class="raw-markdown" id="markdown"></textarea>

<div class="rendered-html" id="html"></div>

</section>

</body>

<script> require('./renderer');

</script>

</html>

**在文件末尾的<script>标记中，我们需要渲染器进程的代码，该代码位于同一目录中的renderer.js中。**

**<label>标签是可选的，包含在内以使视觉障碍用户更容易使用该应用程序。**

**我们的应用程序允许我们使用.raw-markdown类在文本区域中编写和编辑内容，并使用.rendered-html类在div元素中呈现该内容。**

我们的应用程序还没有太多关注。如果你像我一样，你对我在线框中引入的双列接口持怀疑态度。在讨论如何使用HTML和CSS实现列时，很少使用easy这个词。幸运的是，我们可以放心地使用添加到CSS3的新布局模式（称为Flexbox）来快速定义应用程序的双列布局。Flexbox可以轻松创建在各种屏幕尺寸上可预测的页面布局，如清单3.4所示。这对CSS来说相对较新，直到最近才被Internet Explorer支持。正如我们在第1章和第2章中讨论的那样，我们的应用程序总是与最新版本的Chrome相结合，因此我们可以放心地使用Flexbox布局模式，而无需担心跨浏览器兼容性。

**清单3.4 使用Flexbox创建页面布局：./app/style.css**

html {

box-sizing: border-box;

}

\*, \*:before, \*:after {

box-sizing: inherit;

**选择更新的CSS框模型，该模型将正确设置元素的宽度和高度**

}

html, body { height: 100%;

width: 100%; overflow: hidden;

**将此设置传递给页面上的每个其他元素和伪元素**

}

body {

margin: 0;

padding: 0; position: absolute;

}

body, input {

font: menu;

**在整个应用程序中使用操作系统的默认字体**

}

textarea, input, div, button {

outline: none;

margin: 0;

}

.controls {

**删除浏览器在活动输入字段周围的默认突出显示**

background-color: rgb(217, 241, 238); padding: 10px 10px 10px 10px;

}

button {

font-size: 14px;

background-color: rgb(181, 220, 216); border: none;

padding: 0.5em 1em;

}

button:hover {

background-color: rgb(156, 198, 192);

}

button:active {

background-color: rgb(144, 182, 177);

}

button:disabled {

background-color: rgb(196, 204, 202);

}

.container { display: flex;

flex-direction: column; min-height: 100vh;

min-width: 100vw; position: relative;

}

.content {

height: 100vh; display: flex;

**使用Flexbox来对齐我们应用程序的两个窗格**

}

.raw-markdown, .rendered-html {

min-height: 100%;

max-width: 50%;

flex-grow: 1; padding: 1em; overflow: scroll; font-size: 16px;

}

**使用Flexbox将两个窗格设置为相等的宽度**

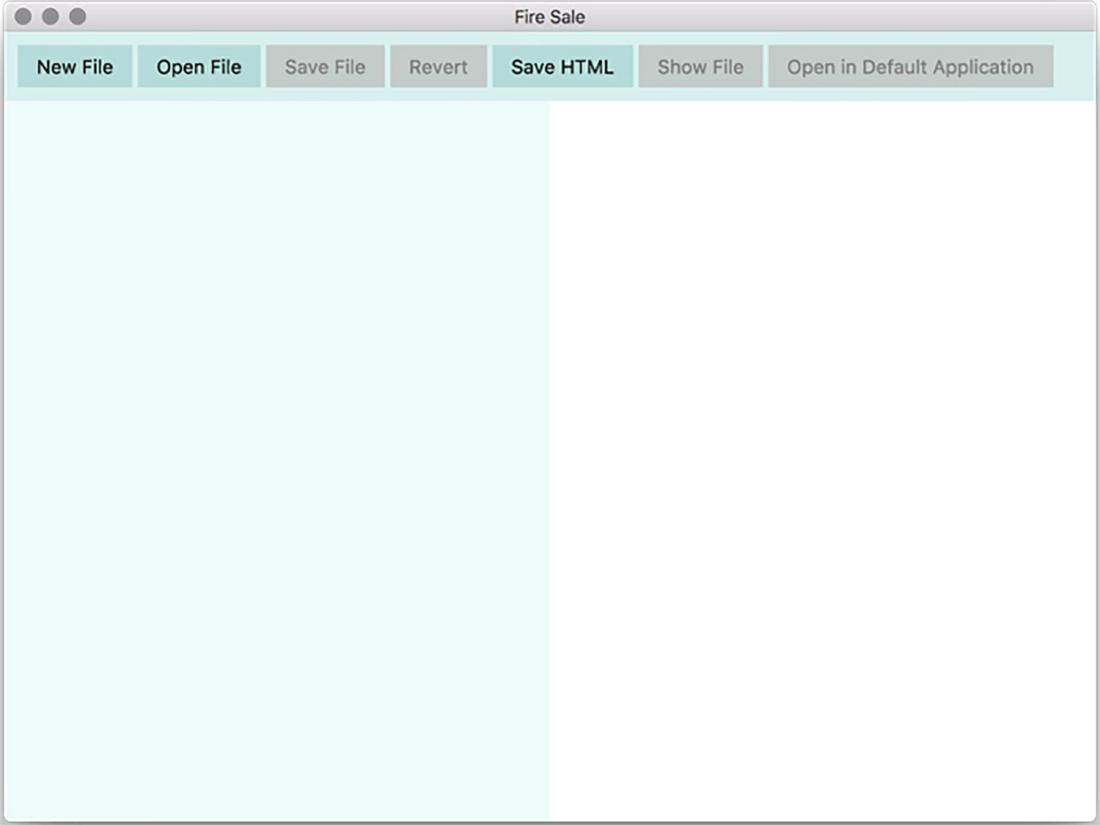
.raw-markdown {

border: 5px solid rgb(238, 252, 250);;

background-color: rgb(238, 252, 250); font-family: monospace;

}

样式表有两个主要目标。首先，我们希望利用Flexbox等现代CSS功能来布局我们的UI。其次，我们希望采取一些小步骤，使我们的应用程序看起来更像一个真正的Web应用程序（见图3.6）。



**图3.6 我们的应用程序已经使用CSS的现代功能给出了一些基本样式。**

box-sizing属性处理CSS中的历史奇怪，其中向宽度为200像素的元素添加50像素的填充将导致其为300像素宽（每侧添加50像素的填充），同样为真对于边界也是如此。当box-sizing设置为border-box时，我们的元素会尊重我们设置它们的高度和宽度。一般来说，这是一件好事。在这个CSS规则中，我们还有其他所有元素和伪元素都通过将box-sizing设置为border-box来尊重我们所做的艰苦工作。

我们希望我们的应用程序适合他们的本地同事。朝这个方向迈出的重要一步是使用所有其他应用程序使用的系统字体。这说起来容易做起来难。例如，尽管macOS使用San Francisco作为整个操作系统的默认字体，但它不能作为常规字体使用。我们将font属性设置为menu，这会使操作系统使用其默认字体,即使我们无法访问它。

浏览器在当前活动的任何UI元素周围放置边框。在macOS中，此边框是蓝色光晕。你可能从来没有想过太多，因为我们已经习惯了它，但是当我们开发桌面应用程序时，它看起来不合适。在我们的应用程序中，它看起来特别糟糕，其中一半的UI实际上是一个大文本输入。通过将轮廓设置为无，我们消除了活动元素周围的不自然的光晕。

在.content，.raw-markdown和.rendered-html规则中，我们实现了一个简单的Flexbox布局，这将使我们的应用程序看起来更像我们在图3.1中介绍的线框。具有内容类的元素将保留我们的两列。我们将display属性设置为flex，以使用我们之前讨论过的Flexbox技术。在下一步中，我们设置flex-grow，它当然指定了flex项的增长因子。将其视为与其兄弟姐妹相关的元素规模可能会有所帮助。在这种情况下，我们使用Flexbox将两列设置为相等的比例。

###### *优雅地显示浏览器窗口*

如果您在应用程序启动时仔细查看，您会注意到在Electron加载index.html并在窗口中呈现DOM之前窗口完全空白的短暂时刻。用户不习惯在原生应用程序中看到这一点，我们可以通过重新思考启动窗口的方式来避免它。

如果你考虑主进程中的代码，应用程序首次启动时的虚无闪现是有意义的：它创建一个窗口，然后在其中加载内容。如果我们隐藏窗口直到加载内容怎么办？ 然后，当UI准备就绪时，我们会显示窗口并避免短暂暴露一个空窗口。

**清单3.5 在DOM准备就绪时优雅地显示窗口：./app/main.js**

app.on('ready', () => {

mainWindow = new BrowserWindow({ show: false }); mainWindow.loadFile('index.html');

mainWindow.once('ready-to-show', () => {

mainWindow.show();

});

**首先在首次创建时隐藏窗口。**

将单个事件侦听器添加到窗口的“ready-to-show”事件中。

mainWindow.on('closed', () => { mainWindow = null;

});

});

**在DOM准备好时显示窗口。**

我们将一个对象传递给BrowserWindow构造函数，默认情况下将其设置为隐藏。当BrowserWindow实例触发其'ready-to-show'事件时，我们将调用它的show()方法，这将在UI完全准备就绪后将其隐藏起来。当应用程序通过网络加载远程资源时，此方法甚至更有用，初始化页面可能需要更长的时间。

译注:新版欲正常运行,第二行需改为:

mainWindow = new BrowserWindow(

{

show: false, // 首先在首次创建时隐藏窗口。

// 5.0开始,此值nodeIntegration默认false,导致html中不可以使用require函数,提示未定义,特添加此配置解决require报错的问题

webPreferences: {

nodeIntegration: true

}

}

);

***实现基本功能* 57**

### *实现基本功能*

让我们通过获得一些基本功能来实现利益。对于初学者，只要左窗格中的Markdown发生变化，我们就会在右窗格中更新呈现的HTML视图（参见图3.7）。这是我们的一个依赖标记 - 在哪里发挥作用。

**更改会触发keyup事件**

**Markdown更改时重新呈现HTML视图**

htmlView

markdownView

…

Open File

New File

markdownView.addEventListener('keyup')

**图3.7 我们将向左窗格添加一个事件监听器，它将Markdown呈现为HTML并将其显示在右窗格中。**

引入我们的依赖项很容易，因为我们可以使用Node的require来标记。我们在app/renderer.js中添加以下内容。

**清单3.6 导入我们的依赖项：./app/renderer.js**

const marked = require('marked');

现在，我们可以使用marked访问Marked。鉴于我们对应用程序功能的讨论以及图3.7中的图表，您可能已经开始怀疑我们将在开发应用程序时使用#markdown文本区域和#html元素。让我们使用一对变量来存储对每个元素的引用，以便它们更容易使用，如清单3.7所示。在我们处理它时，让我们也为UI顶部的每个按钮创建变量。

**清单3.7 缓存DOM选择器：./ app/renderer.js**

const markdownView = document.querySelector('#markdown'); const htmlView = document.querySelector('#html');

const newFileButton = document.querySelector('#new-file'); const openFileButton = document.querySelector('#open-file');

const saveMarkdownButton = document.querySelector('#save-markdown'); const revertButton = document.querySelector('#revert');

const saveHtmlButton = document.querySelector('#save-html'); const showFileButton = document.querySelector('#show-file');

const openInDefaultButton = document.querySelector('#open-in-default');

我们也经常将Markdown渲染到htmlView中，因此我们希望自己能够在将来为我们提供更轻松的功能。

**清单3.8 将Markdown转换为HTML：./ app/renderer.js**

const renderMarkdownToHtml = (markdown) => {

htmlView.innerHTML = marked(markdown, { sanitize: true });

};

marked将我们要呈现的Markdown内容作为第一个参数呈现，将选项对象作为第二个参数。我们希望保护自己免受偶然的脚本注入，因此我们传入一个将sanitize属性设置为true的对象。

最后，我们向markdownView添加一个事件监听器，在keyup上将读取其内容（在textarea元素中，存储在其value属性中），通过marked运行它们，然后将它们加载到htmlView中。结果如图3.8所示。

**清单3.9 Markdown更改时重新呈现HTML：./ app/renderer.js**

markdownView.addEventListener('keyup', (event) => {

const currentContent = event.target.value; renderMarkdownToHtml(currentContent);

});



**图3.8 我们的应用程序在左窗格中获取用户键入的内容，并在右窗格中自动将其呈现为HTML。此内容由用户提供，不属于我们的应用程序。**

基本功能已经到位，我们已经准备好开始处理只能在Electron应用程序中实现的功能 --从文件系统读取和写入文件开始。完成所有操作后，我们的应用程序的渲染器过程应该如下所示。

**清单3.10 渲染器进程：./app/renderer.js**

const marked = require('marked');

const markdownView = document.querySelector('#markdown'); const htmlView = document.querySelector('#html');

const newFileButton = document.querySelector('#new-file'); const openFileButton = document.querySelector('#open-file');

const saveMarkdownButton = document.querySelector('#save-markdown'); const revertButton = document.querySelector('#revert');

const saveHtmlButton = document.querySelector('#save-html'); const showFileButton = document.querySelector('#show-file');

const openInDefaultButton = document.querySelector('#open-in-default');

const renderMarkdownToHtml = (markdown) => {

htmlView.innerHTML = marked(markdown, { sanitize: true });

};

markdownView.addEventListener('keyup', (event) => {

const currentContent = event.target.value; renderMarkdownToHtml(currentContent);

});

### *调试Electron应用*

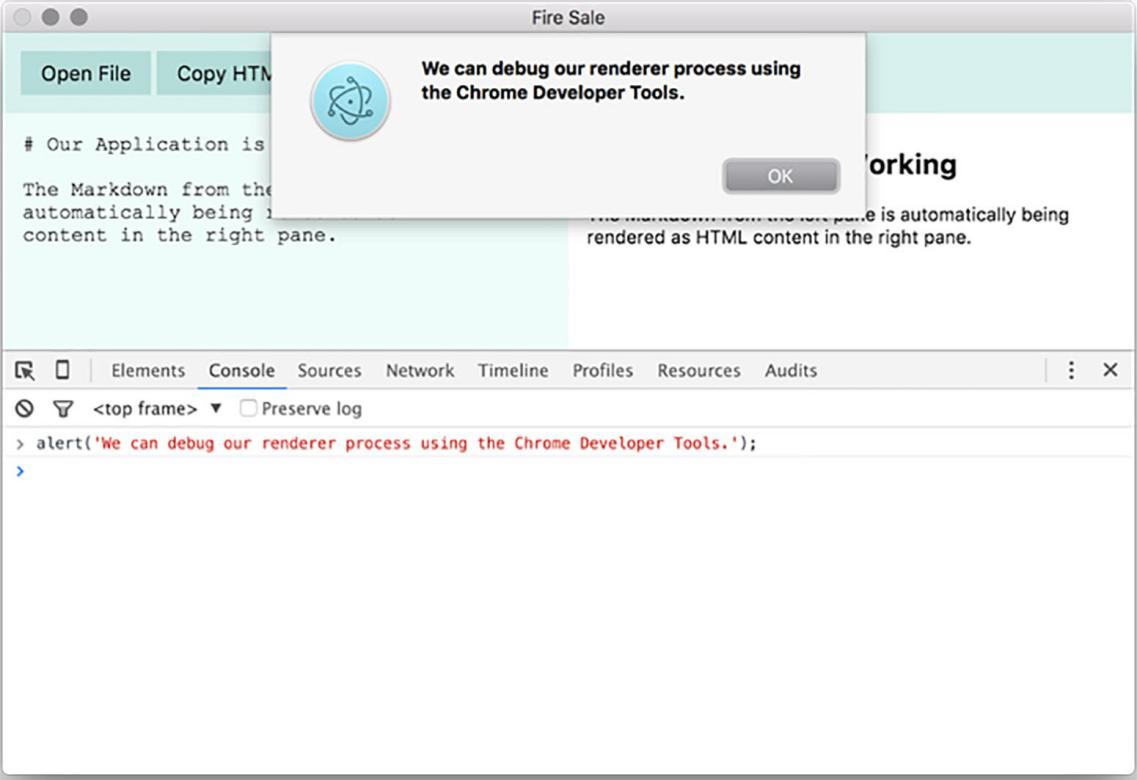
在理想的世界中，我们在编写代码时永远不会犯错误。API和方法永远不会在版本之间发生变化，并且每当本书中的应用程序使用的新版本的依赖项发布时，您的作者就不必屏住呼吸。但是我们不是生活在那个世界里。因此，我们可以使用开发人员工具来帮助我们追踪并希望消除错误。

###### *调试渲染器进程*

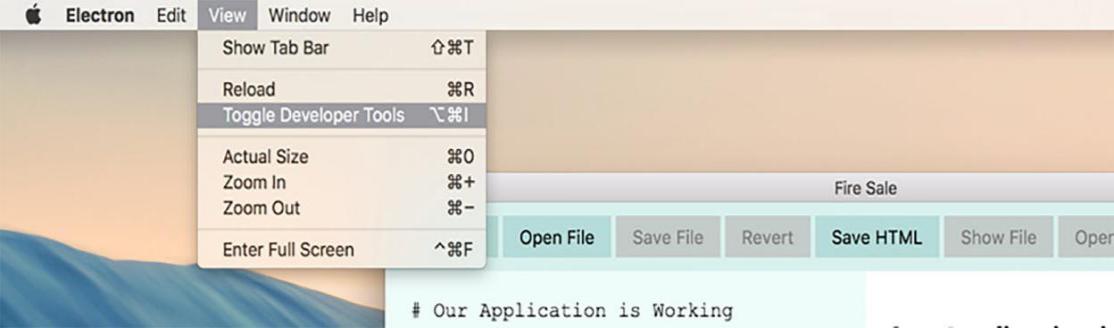
到目前为止，一切都进展顺利，但可能不久我们就不得不调试一些棘手的情况。由于Electron应用程序基于Chrome，因此在构建Electron应用程序时我们可以访问Chrome开发者工具并不奇怪（图3.9）。

调试渲染器过程相对简单。Electron的默认应用程序菜单提供了在我们的应用程序中打开Chrome开发者工具的命令。在第6章中，我们将学习如何创建自己的自定义菜单，并在您不希望向用户公开时删除此功能。

还有另外两种方法可以访问开发人员工具。在任何时候，您都可以在Windows或Linux上的macOS或Control-Shift-I上按Command-Option-I来打开工具（图3.10）。此外，您可以以编程方式触发Developer Tools。BrowserWindow实例上的webContents属性有一个名为openDevTools()的方法。这个方法，如清单3.11所述，将打开它所调用的BrowserWindow中的Developer Tools。



**图3.9 Chrome开发者工具在渲染器过程中可用，就像在基于浏览器的应用程序中一样。**

**图3.10可以在Electron提供的默认菜单中打开和关闭工具。您也可以使用Windows上的Control-Shift-I或macOS上的Command-Option-I切换它们。**

**清单3.11 从主进程打开Developer Tools：./ app/main.js**

app.on('ready', () => {

mainWindow = new BrowserWindow();

mainWindow.loadFile('index.html');

mainWindow.once('ready-to-show', () => {

mainWindow.show(); mainWindow.webContents.openDevTools();

});

mainWindow.on('closed', () => { mainWindow = null;

});

});

**我们可以在加载后立即以编程方式触发主窗口上的开发人员工具的打开。**

###### *调试主进程*

调试主进程并不那么容易。Node Inspector是调试Node.js应用程序的常用工具，目前还不完全支持Electron。您可以使用--debug标志以调试模式启动Electron应用程序，默认情况下将在端口5858上启用远程调试。

官方文档中提供了对使用Node Inspector with Electron的有限支持。由于暂时还有一点变化，如果您不使用Visual Studio Code，请查看最新版本的文档（<http://electron.atom.io/docs/tutorial/debugging-main-process/>）。也就是说，我没有发现这种技术特别稳定，不会推荐它。你的旅费可能会改变。

###### *用Visual Studio Code调试主进程*

Visual Studio Code是一个免费的开源IDE，可用于Windows，Linux和macOS，巧妙地构建在Microsoft的Electron之上。Visual Studio Code附带了一组丰富的工具，用于调试Node应用程序，使调试Electron应用程序比以前提到的更容易。设置构建任务的一种快速方法是让Visual Studio代码在没有构建任务的情况下构建应用程序。在Windows上按Control-Shift-B或在macOS上按Command-Shift-B，系统将提示您创建构建任务，如图3.11所示。

单击“配置构建任务”菜单项将提示您选择是要创建“开始”还是“测试”任务。选择“开始”将生成一个调用npm start的任务。选择“test”将生成npm测试。代码清单3.12是“开始”任务的示例。



**图3.11在没有安装任务的情况下触发构建任务将提示Visual Studio代码代您创建一个。**

**清单3.12 在Visual Studio Code for Windows中设置构建任务：tasks.json**

{

// See https://go.microsoft.com/fwlink/?LinkId=733558

// for the documentation about the tasks.json format "version": "2.0.0",

"tasks": [

{

"type": "npm",

"script": "start", "group": {

"kind": "build", "isDefault": true

}

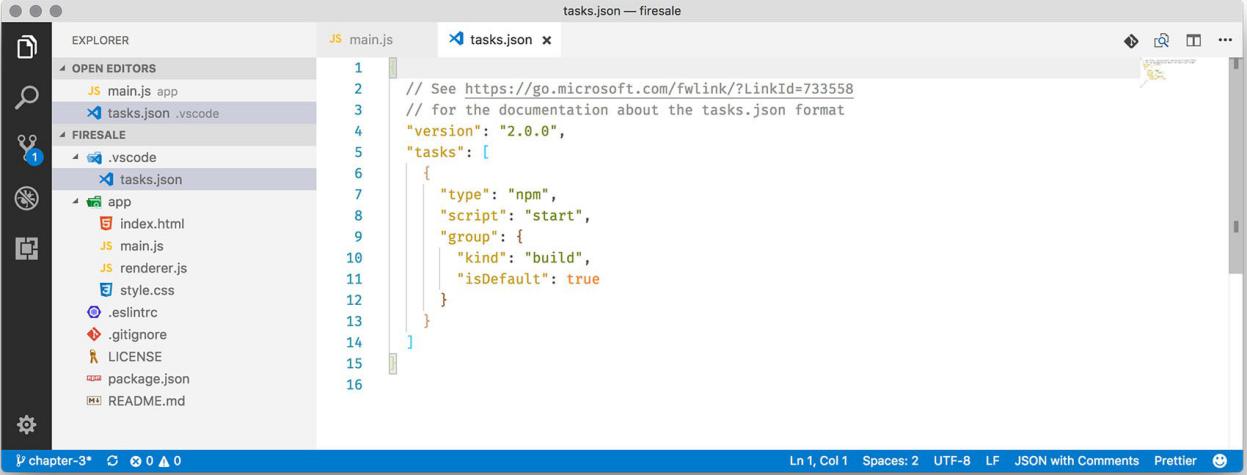
}

]

}

现在，当您在Windows上按Control-Shift-B或在macOS上按下Command-Shift-B时，您的Electron应用程序将启动。这不仅对于在Visual Studio Code中设置调试很重要，它也是一般启动应用程序的便捷方式。下一步是设置Visual Studio Code以启动应用程序并将其连接到其内置调试器（图3.12）。

要创建启动任务，请转到左侧窗格中的“调试”选项卡，然后单击左上角的小齿轮。Visual Studio Code将询问您要创建哪种配置文件。选择Node并用列表3.13替换文件的内容。



**图3.12 在Debug选项卡内，单击齿轮，Visual Studio Code将创建一个配置文件，代您启动调试器。**

**清单3.13 为Visual Studio Code for Windows设置启动任务：launch.json**

{

"version": "0.2.0", "configurations": [

{

"name": "Debug Main Process", "type": "node",

"request": "launch",

"cwd": "${workspaceRoot}", "runtimeExecutable":

"${workspaceRoot}/node\_modules/.bin/electron", "windows": {

"runtimeExecutable": "${workspaceRoot}/node\_modules/.bin/electron.cmd"

},

"args":

"."

]

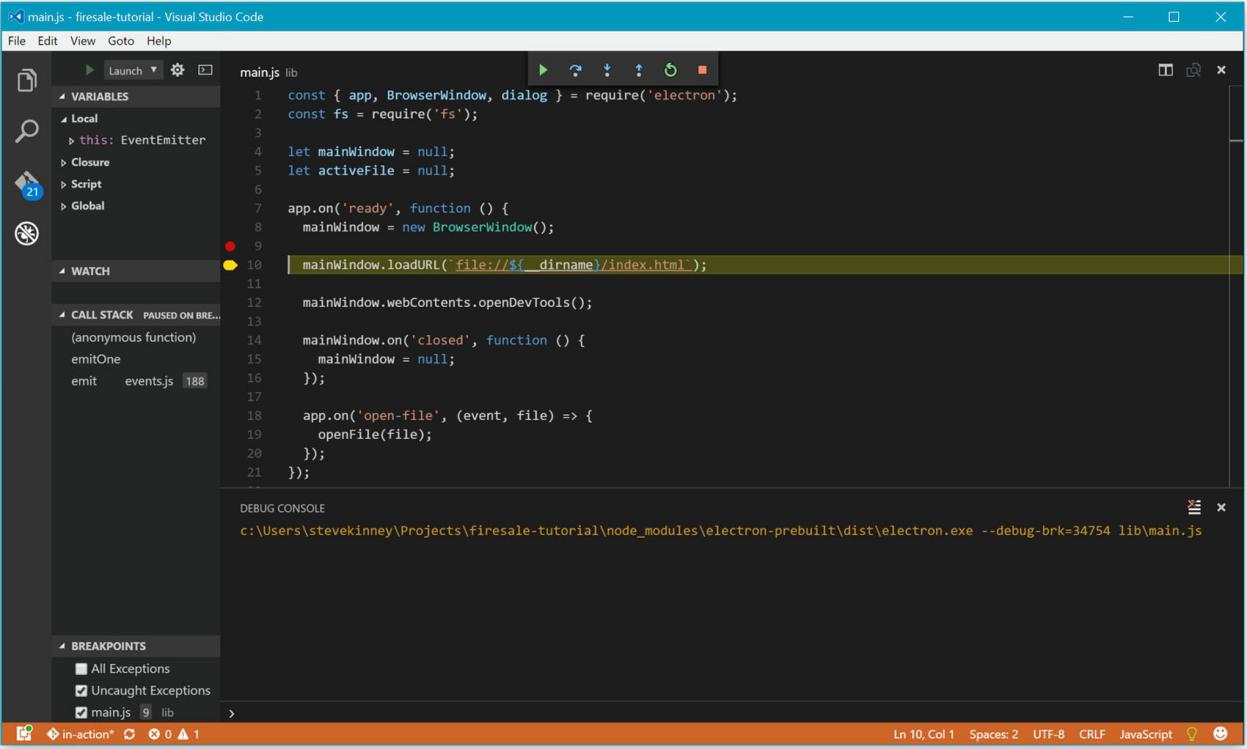
}

]

}

使用此配置文件，您可以单击主进程中任何行的左边距来设置断点，然后按F5运行该应用程序。执行将在断点处暂停，允许您检查调用堆栈，确定范围内的变量，并与实时控制台交互。断点不是调试代码的唯一方法。每当抛出未捕获的异常时，您也可以监视特定的表达式或放入调试器（图3.13）。

您很可能没有使用Visual Studio Code。没关系。这不是本书的先决条件，你几乎可以使用你最熟悉的文本编辑器或IDE。此外，Visual Studio Code并不是唯一支持调试主进程的代码。例如，您可以在此处找到有关配置WebStorm的详细信息：http://mng.bz/Y5T6。使用IDEA的读者可以参考译者的笔记:http://note.youdao.com/noteshare?id=4136dc5f21f9519a1373828b864facff



**图3.13 Visual Studio Code内置的调试器允许您暂停应用程序的执行并插入以调试错误。**

### *本章小结*

* + - * 在接下来的几章中，我们将开发一个Markdown-to-HTML渲染器。
      * Flexbox受现代浏览器支持，允许我们轻松实现双窗格界面，随着用户更改窗口大小，该界面将适应。
      * Chrome开发人员工具可用于所有渲染器进程，可以从Electron的默认应用程序，键盘快捷键或主进程中触发。
      * 此时Electron不完全支持Node Inspector。
      * VisualStudioCode提供了一组丰富的工具，用于调试应用程序主进程中的问题。

# *使用原生文件对话框并促进进程间通讯*

***本章包括***

* 使用Electron的对话框模块实现原生打开文件对话框
* 促进主进程和渲染器进程之间的通信
* 从主进程到渲染器进程的功能
* 使用Electron的远程模块将主进程的功能导入渲染器进程
* 使用webContents模块从主进程向渲染器进程发送信息，并使用ipcRenderer模块为主进程设置监听器

在上一章中，我们为我们的第一个Electron项目奠定了基础，这是一个Notes应用程序，它从左侧窗格中获取Markdown，并在右侧窗格中将其呈现为HTML。我们设置了主进程并将其配置为生成渲染器。我们设置了package.json，安装了必要的依赖项，创建了主进程和渲染器进程，并布局了UI。我们还探索了使应用程序感觉像桌面应用程序的方式，但是我们还没有添加一个远远超出传统Web应用程序可以做的范围的功能。

现在，该应用程序允许用户在Markdown视图中书写。当用户在Markdown视图中按下某个键时，应用程序会自动将Markdown呈现为HTML并将其显示在HTML视图中。

在本章中，我们将添加触发本机文件对话框的功能，并从文件系统的任何位置选择文本文件并将其加载到我们的应用程序中。在本章结束时，渲染器进程的浏览器窗口中的Open File按钮将从主进程触发Open File对话框。在我们能够做到这一点之前，重要的是讨论如何在更深入的进程之间进行通信。我们从第3章分支开始，可以在http://mng.bz/11Kd找到。本章末尾的代码可以在http://mng.bz/0C34找到。或者，您可以pull主分支并checkout这两个分支中的任何一个。

git clone https://github.com/electron-in-action/firesale.git chapter-3 git checkout -f chapter3

### *触发原生的文件对话框*

一个简单的入门方法是在应用程序首次启动时提示用户打开文件并发出ready事件，如图4.1所示。在创建BrowserWindow实例之前，我们的应用程序已经在侦听ready事件。在本章的后面部分，我们将学习如何从UI触发此功能。在下一章中，我们将学习如何从应用程序菜单中触发它。

app.on('ready')

Electron

Load index.html into mainWindow

Create BrowserWindow and store it in mainWindow

**读取package.json以加载主进程**

主进程

Open developer tools

首次准备好应用程序时添加打开文件对话框。

Add event listener for when mainWindow closes

Display file open dialog

**图4.1我们的应用程序将在启动时触发Open File对话框。到本章结束时，该功能将被从UI触发对话框的能力所取代。**

您可以使用Electron的对话框模块创建本机对话框。将清单4.1中的代码添加到app/main.js，就在需要其他Electron模块的位置下方。

***触发原生的文件对话框* 67**

**清单4.1导入对话框模块：./ app/main.js**

const { app, BrowserWindow, dialog } = require('electron');

最终，应用程序应该从多个位置触发我们的文件打开功能。第一步是创建一个稍后引用的函数。首先，在选择文件后将所选文件的名称记录到控制台。

**清单4.2创建getFileFromUser()函数：./ app/main.js**

const getFileFromUser = () => {

const files = dialog.showOpenDialog({ properties: ['openFile']

});

if (!files) { return; } console.log(files);

**触发操作系统的“打开文件”对话框。我们还将一个具有不同配置参数的JavaScript对象传递给该函数。**

**配置对象在“打开文件”对话框中设置不同的属性。**

}; **将文件记录到控制台**

**如果我们没有任何文件，从函数中提前返回。**

我们的getFileFromUser()函数是dialog.showOpenDialog()的包装器，我们可以在应用程序的多个位置使用它，而不必重复自己。它将触发对话框上的showOpenDialog()方法，并将其传递给具有不同设置的JavaScript对象，我们可以根据需要进行调整。在JavaScript中，对象的键称为其属性。传递给dialog.showOpenDialog()的对象的属性配置了我们正在创建的对话框的某些特征。一个这样的设置是对话框本身的属性。配置对象上的properties属性采用我们可以在对话框中设置的不同标志的数组。在这种情况下，我们只激活openFile标志，这表示此对话框用于选择要打开的文件,而不是选择目录或多个文件。可用的其他标志是openDirectory和multiselections。

dialog.showOpenDialog()返回所选文件的名称。用户选择的路径数组存储在名为files的变量中。如果用户按下取消，则dialog.showOpenDialog()返回undefined并在我们尝试在未定义的文件上调用任何方法时中断。如果files变量是假值，则return语句通过尽早离开函数来防止这种情况,并且undefined实际上是假值。

必须在我们的应用程序中的某处调用getFileFromUser()来触发对话框。最终，它将从UI和应用程序菜单中调用。现在这样做的一个方便的地方是应用程序启动时。当app模块触发ready事件时调用getFileFromUser()，如下面的清单所示。当我们的UI配置为从渲染器进程触发getFileFromUser()时，将删除此步骤。

**清单4.3 在应用程序第一次准备好时调用getFileFromUser()**

app.on('ready', () => {

mainWindow = new BrowserWindow({ show: false }); mainWindow.loadFile('index.html');

mainWindow.once('ready-to-show', () => { mainWindow.show();

getFileFromUser();

});

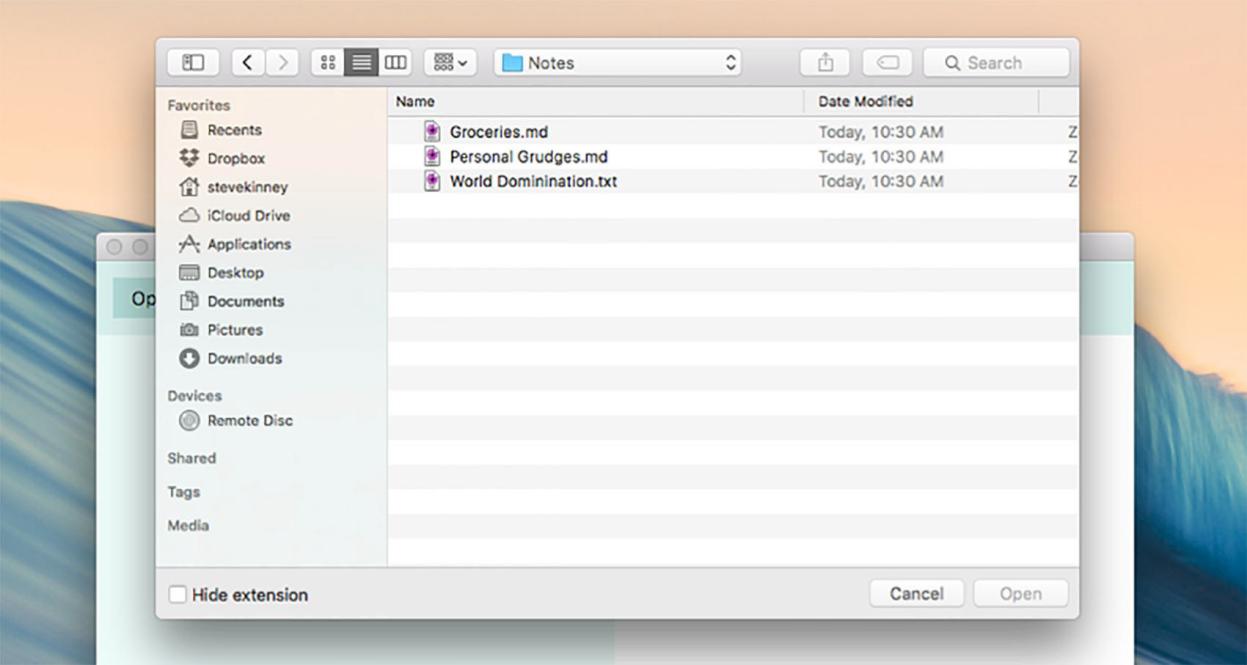
mainWindow.on('closed', () => { mainWindow = null;

});

});

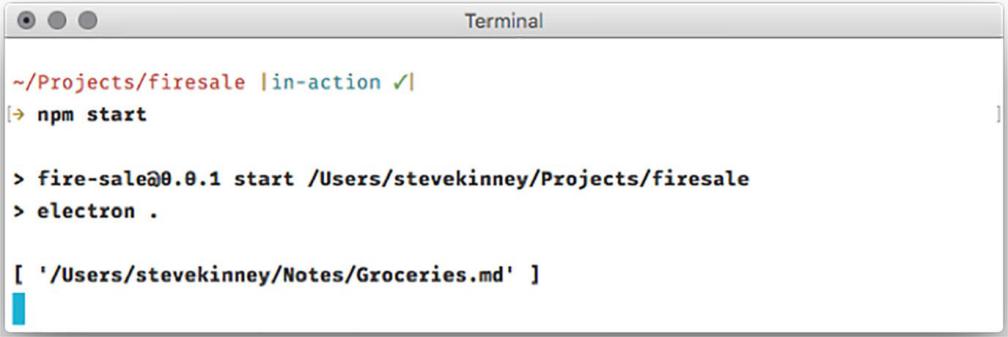
**当窗口准备好显示时，我们将调getFileFromUser()。getFileFromUser()在清单4.2中定义。**

当我们的应用程序启动并且窗口完全加载时，用户会立即看到一个文件对话框，这将允许他们选择一个文件（参见图4.2）。我们最终从启动过程中删除了此函数调用，并将其分配给UI中的“打开文件”按钮。



**图4.2 Electron能够在其支持的每个操作系统中触发本机文件对话框。**

在图4.3中，我们可以在终端中显示的“打开文件”对话框中看到我们选择的结果。请注意，dialog.showOpenDialog()返回一个数组。如果在对话框的属性数组中激活了多项选择，则用户可以选择多个文件。Electron始终返回一个数组以保持一致性。



**图4.3 选择文件后，文件的完整路径将记录在终端窗口的控制台中。**

### *使用Node读取文件*

dialog.showOpenDialog()返回一个数组，该数组由用户选择的文件或文件的路径组成，但不代表我们读取它们。根据我们正在构建的文件类型，我们可能希望以不同方式处理打开文件。在此应用程序中，文件的内容被读取并立即显示在UI中。当用户选择文件时，处理复制图像或将其上载到外部服务的不同应用程序可能采用对比方法。还有一个应用程序可能会将大型电影添加到播放列表中以便稍后观看。在这种情况下，立即开始打开大文件将是浪费。

Node在其标准库中附带了一组用于使用文件的工具。内置的fs库处理常见的文件系统操作，例如读取和写入文件，因此您应该在app/main.js的顶部附近使用它。

**清单4.4 导入Node的fs模块：./app/main.js**

const { app, BrowserWindow, dialog } = require('electron');

const fs = require('fs');

app.on('ready',() => { … }); const getFileFromUser = () => {

为清楚起见省略了代码

**需要Node的fs库。**

const files = dialog.showOpenDialog(mainWindow, { properties: ['openFile']

});

if (!files) { return; }

const file = files[0];

**从数组中取出第一个文件**

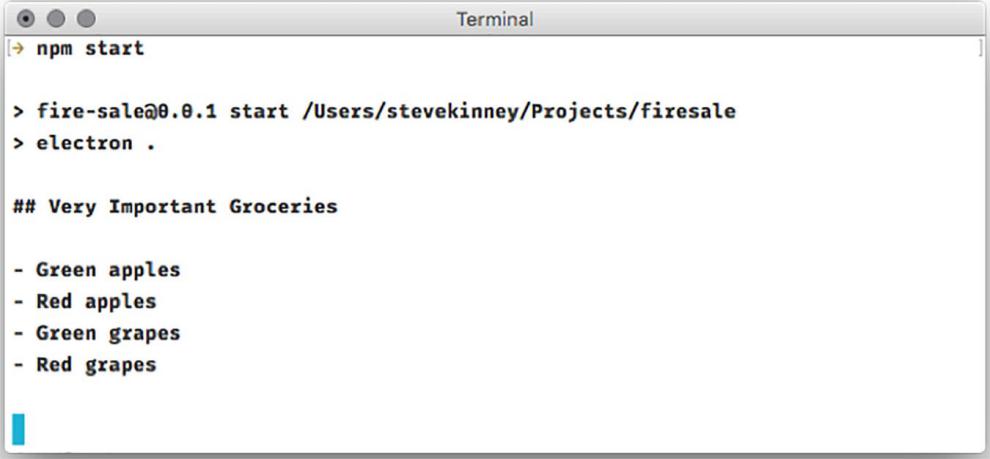
const content = fs.readFileSync(file).toString();

console.log(content);

};

**从文件中读取，并将生成的缓冲区转换为字符串。**

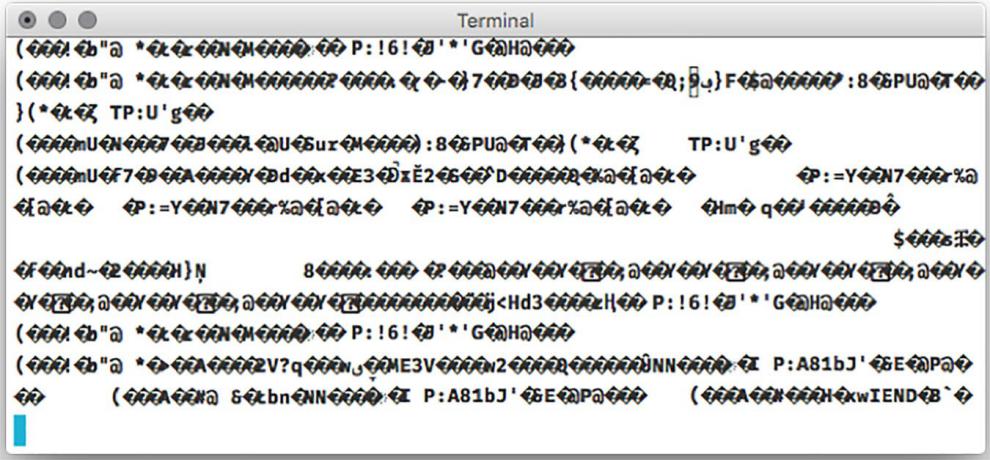
在清单4.4中，应用程序一次只打开一个文件。files [0]从dialog.showOpenDialog()中选择数组中的第一个和唯一文件路径。在fs.readFileSync（file）中，文件路径作为参数传递给fs.readFileSync()。Node不知道打开了哪种文件，因此fs.readFileSync()返回一个缓冲区对象。但是，我们知道，我们通常在此限定应用程序中使用纯文本。我们将其转换为字符串并将文件内容记录到终端，如图4.4所示。



**图4.4文件的内容被记录到用户的终端。**

###### *限定文件对话框范围*

如图4.4所示，getFileFromUser( )成功将文本文件的内容记录到终端。但有一个问题。默认情况下，dialog.showOpenDialog( )允许我们打开计算机上的任何文件，而不考虑我们准备处理的文件类型。图4.5显示了通过对话框打开图像文件而不是文本文件时出现的问题。



**图4.5如果用户选择非文本文件，该函数会记录二进制数据。**

许多桌面应用程序可以限制用户可以打开的文件类型。对于使用Electron构建的应用程序也是如此。我们的应用程序不适合打开音乐文件，因此我们不应该让用户选择MP3。可以将其他选项添加到传递给dialog.showOpenDialog( )的配置对象，以将对话框限制为我们已列入白名单的文件扩展名。

**清单4.5 将特定文件类型列入白名单：./app/main.js**

const getFileFromUser = () => {

const files = dialog.showOpenDialog({ properties: ['openFile'],

filters: [

**filters属性允许我们指定应用程序应该能够打开的文件类型，并禁用任何与我们的条件不匹配的文件。**

]

});

{ name: 'Text Files', extensions: ['txt'] },

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

if (!files) { return; }

const file = files[0];

const content = fs.readFileSync(file).toString();

console.log(content);

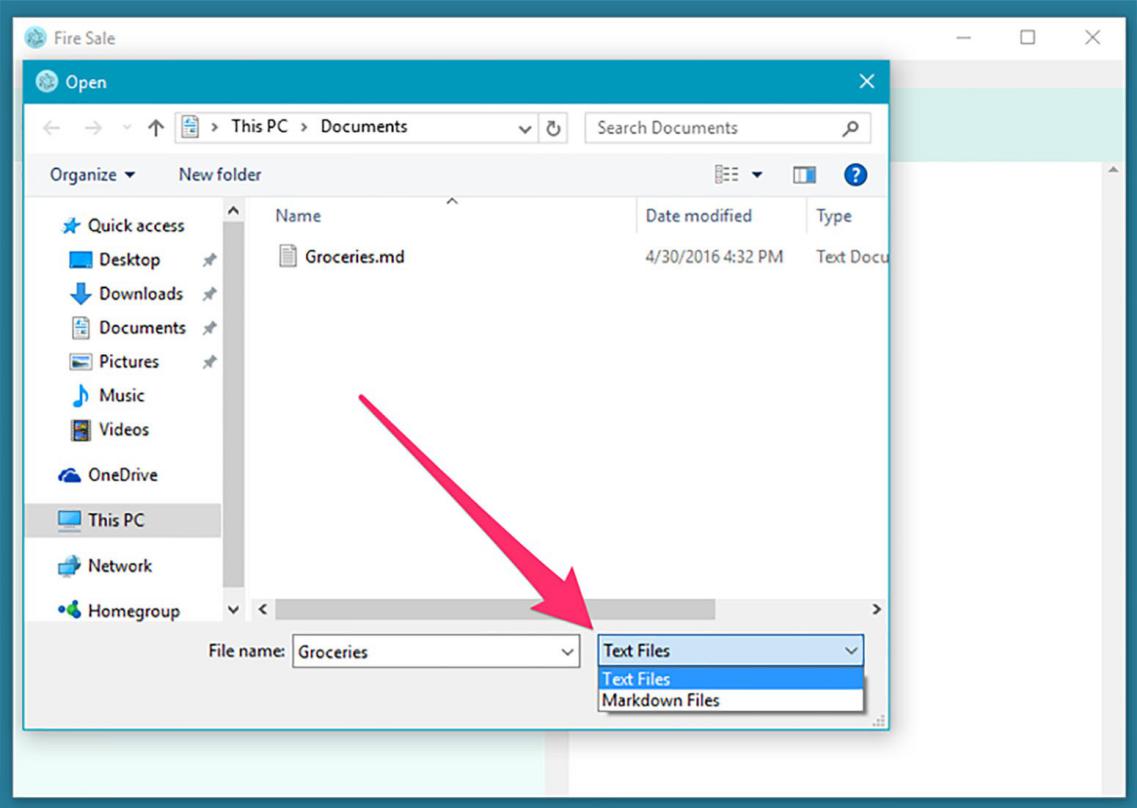
};

在列表中，我们向传递给dialog.showOpenDialog( )的对象添加了第二个属性。在Windows中，对话框在下拉菜单中显示名称Markdown Files，如图4.6所示。在macOS中，没有下拉菜单，但我们无法选择没有其中一个扩展名的图像，如图4.7所示。

###### *在macOS中实现对话框*

Electron应用程序设计为跨平台，这意味着它们可以在macOS，Windows和Linux上运行。Electron为每个支持操作系统中存在的本机功能和API提供接口，但在其他支持操作系统中不存在。我们之前为文件扩展名过滤器提供了一个名称。此名称出现在Windows中，但macOS没有此功能。如果可用，Electron会利用此功能，但它仍然适用于不可用的情况。

在macOS中，我们能够显示从窗口顶部作为工作表下拉而不是显示在其前面的对话框（清单4.6）。我们可以在Electron中轻松创建这个UI，方法是在配置对象之前将对我们存储在mainWindow中的BrowserWindow实例的引用作为dialog.showOpenDialog( )的第一个参数传递给我们。



**图4.6 在Windows中，我们可以在不同类型的文件之间切换。**



**图4.7 macOS不支持在文件类型之间切换，但允许我们选择任何符合筛选器选项定义的文件。**

**清单4.6 在macOS中创建工作表对话框：./app/main.js**

const getFileFromUser = () => {

const files = dialog.showOpenDialog(mainWindow, { properties: ['openFile'],

filters: [

{ name: 'Text Files', extensions: ['txt'] },

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

if (!files) { return; } const file = files[0];

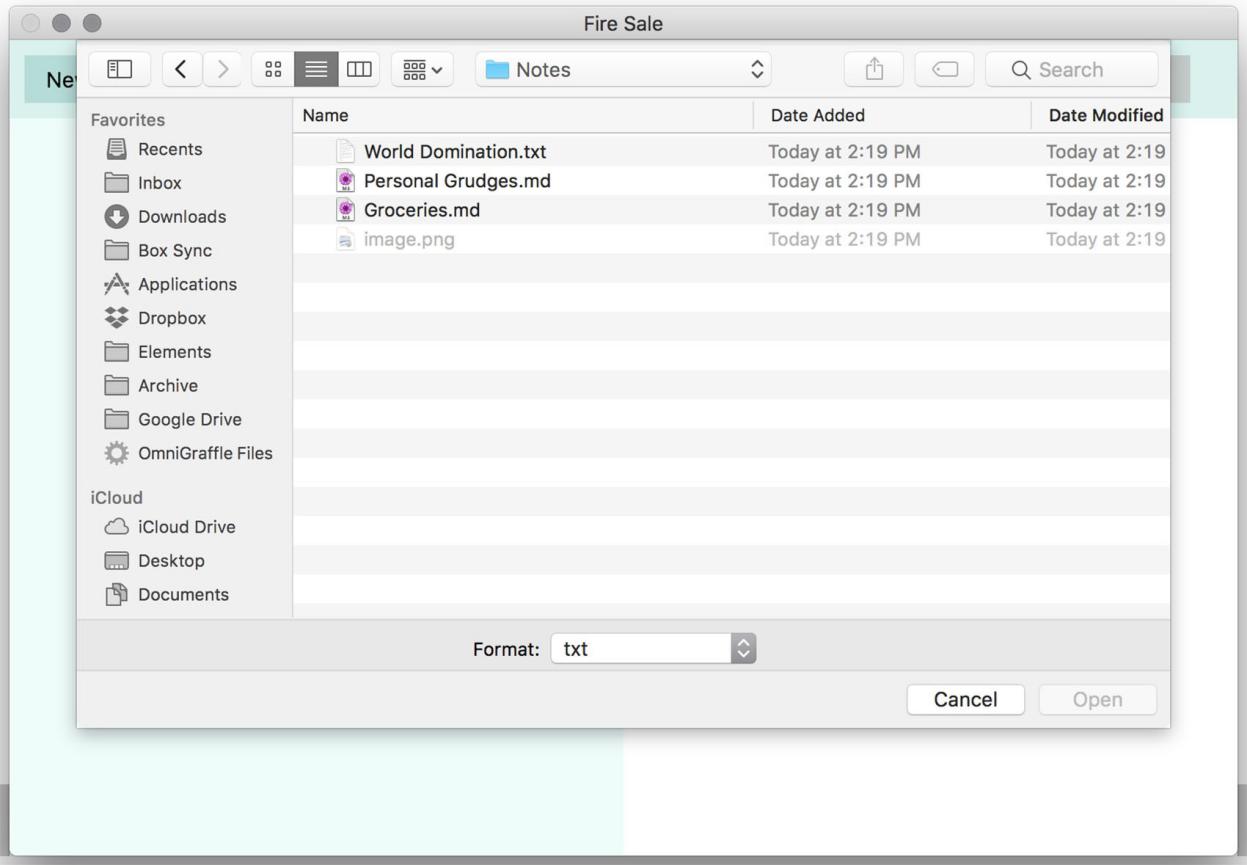
const content = fs.readFileSync(file).toString();

console.log(content);

};

**将对BrowserWindow实例的引用传递给dialog.showOpenDialog将导致macOS将对话框显示为从窗口标题栏下来的工作表。它对Windows和Linux无效。**

通过这个简单的更改，Electron现在将打开文件对话框显示为从传递给方法的窗口下拉的工作表，如图4.8所示。



**图4.8 现在，打开文件对话框不再显示在应用程序窗口前面的附加窗口，而是从macOS中的菜单标题栏下拉。**

### *促进进程间通信*

我们编写了所有用于在主进程中选择文件和读取文件的代码。但是我们如何将文件的内容发送到渲染器进程？ 我们如何从UI中触发主进程中的getFileFromUser( )函数？

在构建传统Web应用程序时，我们必须处理类似的问题。它并不完全相同，因为所有代码都在客户端的计算机上运行，但考虑我们通常如何构建Web应用程序可以作为理解如何构建我们的Electron应用程序的有用隐喻。见图4.9。

你的服务器

客户端的计算机

**渲染器进程**

渲染 UI.

响应用户事件

**主进程**

写入文件系统或数据库。

客户端之间的协调。

与操作系统通信。

**服务器端代码**

可选的。 Electron应用程序可以与外部服务器在多个客户端之间协调工作

**客户端代码**

渲染 UI.

响应用户事件

**服务器端代码**

写入文件系统或数据库。

客户端之间的协调。

与操作系统通信。

传统Web 应用

图4.9 Electron应用程序与传统Web应用程序的职责分工。

Electron 应用

在Web上，我们通常编写在以下两个位置之一运行的代码：在我们的服务器上或在用户浏览器中运行的客户端代码。客户端代码是呈现UI的内容。它监听并处理用户操作并更新UI以显示应用程序的当前状态。但是，我们使用客户端代码有什么限制?正如我们在第1章中讨论的那样，我们无法读取或写入数据库或文件系统。服务器端代码在我们的计算机上运行。它可以访问数据库。它可以写入我们系统上的日志文件。

在传统的Web应用程序中，我们通常使用HTTP等协议来促进客户端和服务器端进程之间的通信。使用HTTP，客户端可以发送包含信息的请求。服务器接收此请求，适当地处理它，并向客户端发送响应。

在Electron应用中，情况有所不同。正如我们在前面的章节中所讨论的那样，Electron应用程序由多个进程组成：一个主进程和一个或多个渲染器进程。一切都在我们的计算机上运行，但是角色与客户端,服务器模型有类似的分离。我们不使用HTTP在进程之间进行通信。相反，Electron提供了几个模块来协调主进程和渲染器进程之间的通信。

我们的主进程负责与本机操作系统API进行交互。它负责创建渲染器进程，定义应用程序菜单，显示打开和保存对话框，注册全局快捷方式，从操作系统请求电源信息等。Electron通过使执行这些任务所需的许多模块仅在主进程中可用来强制执行此操作，如图4.10所示。

**主进程可用的模块**

app autoUpdater BrowserWindow contentTracing dialog globalShortcut ipcMain

Menu MenuItem powerMonitor

powerSaveBlocker protocol session

webContents tray

**两个进程都可用的模块**

**渲染器进程可用的模块**

clipboard

crashReporter nativeImage screen

shell

desktopCapturer

ipcRenderer remote webFrame

**图4.10 Electron为主进程和渲染器进程提供了不同的模块。这些模块代表了Electron的代码功能。此列表可能会增长，并且在您阅读本文时可能不完整。我鼓励您访问文档以查看最新和最好的功能。**

Electron仅为每个进程提供其模块的子集，并且不会阻止我们访问与Electron模块分离的Node API。如果需要，我们可以从渲染器进程访问数据库或文件系统，但有充分的理由在主进程中保留这种功能。我们可能有很多渲染器进程，但我们总是只有一个主进程。从我们的渲染器进程中读取和写入文件系统可能会成为问题; 我们最终可能会遇到一个或多个进程尝试同时写入同一文件或从文件读取而另一个渲染器进程覆盖它的情况。

JavaScript中的给定进程在单个线程上执行我们的代码，并且一次只能执行一项操作。通过将这些任务委托给主进程，我们可以确信一次只有一个进程正在执行对给定文件或数据库的读取或写入。其他任务遵循正常的JavaScript协议，在事件队列中耐心等待，直到主进程完成其当前任务。

主进程处理调用本机操作系统API或提供文件系统访问的任务是有道理的，但是在渲染器进程中调用可能触发这些操作的UI。即使所有代码都在同一台计算机上运行，我们仍然需要协调流程之间的通信，就像我们必须协调客户端和服务器之间的通信一样。

最近，出现了诸如WebSockets和WebRTC之类的协议，允许客户端和服务器之间的双向通信，甚至客户端之间的通信，而不需要中央服务器来促进通信。当我们构建桌面应用程序时，我们通常不会使用HTTP或WebSockets，但Electron有几种方法来协调进程间通信，我们将在本章开始探讨这些方法，如图4.11所示。

**渲染器进程在主进程中触发Open File**

**渲染器进程**

**主进程将文件内容发送回渲染器进程**

**主进程**

渲染器进程将Markdown转换为HTML并更新HTML视图

渲染器进程使用文件内容更新Markdown视图

主进程使用Node的fs模块读取文件的内容

用户使用对话框从文件系统中选择一个文件

主进程创建本机打开文件对话框

用户单击Open File按钮

**图4.11实现Open File按钮涉及协调渲染器进程和主进程之间的通信。**

我们的UI包含一个标签为Open File的按钮。当用户单击此按钮时，我们的应用程序应提供一个对话框，允许用户选择要打开的文件。在用户选择文件后，我们的应用程序应该读取文件的内容，在我们的应用程序的左侧窗格中显示它们，并在右侧窗格中呈现相应的HTML。

您可能已经猜到了，这需要我们在单击按钮的渲染器进程和主进程之间进行协调，主进程负责显示对话框并从文件系统中读取所选文件。在读取文件后，主进程需要将文件的内容重新发送到渲染器进程（下一清单），以分别在左窗格和右窗格中显示和渲染。

**清单4.7 在渲染器进程中添加事件监听器：./app/renderer.js**

const marked = require('marked');

const markdownView = document.querySelector('#markdown'); const htmlView = document.querySelector('#html');

const newFileButton = document.querySelector('#new-file'); const openFileButton = document.querySelector('#open-file');

const saveMarkdownButton = document.querySelector('#save-markdown'); const revertButton = document.querySelector('#revert');

const saveHtmlButton = document.querySelector('#save-html'); const showFileButton = document.querySelector('#show-file');

const openInDefaultButton = document.querySelector('#open-in-default');

const renderMarkdownToHtml = (markdown) => { htmlView.innerHTML = marked(markdown, { sanitize: true });

};

markdownView.addEventListener('keyup', (event) => { const currentContent = event.target.value; renderMarkdownToHtml(currentContent);

});

openFileButton.addEventListener('click', () => { alert('You clicked the "Open File" button.');

});

**选择更新的CSS框模型，该模型将正确设置元素的宽度和高度**

首先在我们的渲染器过程中向Open File按钮添加一个事件侦听器。有了我们的事件监听器，就可以与主进程协调以触发我们之前创建的Open File对话框。

###### *介绍远程模块*

Electron提供了许多方法来促进进程间通信。第一个是远程(remote)模块 - 从渲染器进程到主进程执行进程间通信的简单方法。远程模块仅在渲染器进程中可用，通过镜像主进程中可访问的模块，作为主进程的代理。当我们访问任何这些属性时，远程模块还负责与主进程之间的通信。

如图4.12所示，远程模块具有几个与仅可用于主进程的模块重叠的属性。在我们的渲染器过程中，我们可以要求远程模块，它提供对主进程中对象和属性的访问，如图4.13所示。

当我们在远程对象上调用方法或属性时，它会向主进程发送同步消息，在主进程中执行，并将结果发送回渲染器进程。远程模块允许我们在主进程中定义功能，并使其易于用于我们的渲染器进程。

**可用于远程模块的模块**

**主进程可用的模块**

app autoUpdater BrowserWindow contentTracing dialog globalShortcut ipcMain

Menu MenuItem powerMonitor

powerSaveBlocker protocol session

webContents tray

clipboard

crashReporter nativeImage screen

shell

图4.12远程模块与主进程中的Electron模块共享许多相同的属性。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Main process**  const electron = require('electron');  const app = electron.app;  const BrowserWindow = electron.BrowserWindow;  const globalShortcut = electron.globalShortcut; |  | **Renderer process**  const electron = require('electron');  const remote = electron.remote; | | |
|  | const app = remote.app; |  |
|  |  |
|  | | |
|  | const BrowserWindow = remote.BrowserWindow; |  |
|  |  |
|  | | |
|  | const globalShortcut = remote.globalShortcut; |  |
|  |  |
|  | | |

图4.13远程模块提供对通常仅适用于主进程的模块的访问。

### *使用进程间通信触发打开文件功能*

应用程序现在可以触发 Open File 对话框并读取用户在主进程中选择的文件的内容。我们还在渲染器进程中的 Open File 按钮中添加了一个事件监听器。现在只需使用我们之前探索过的进程间通信技术来连接它们。

###### *了解CommonJS的require系统*

要使用远程模块使用主进程的功能，我们需要利用Node的CommonJS模块系统将该功能公开给我们应用程序中的其他文件。我们在本书中使用了require来从Electron，Node标准库和第三方库中获取功能，但这是我们第一次将它与我们自己的代码一起使用。让我们花几分钟时间来回顾它是如何工作的。

Node的模块系统由两个主要机制组成：能够从其他源获取功能，以及导出其他源供消费的功能。当我们需要来自其他来源的代码时，另一个来源可能是我们编写的文件，第三方模块，Node中的模块或Electron提供的模块。我们在主进程和渲染器进程的顶部使用了Node的内置require函数。

当我们需要一个模块时，我们究竟要导入什么？在Node中，我们明确声明应该从模块中导出哪些功能，如清单4.8所示。此功能在清单4.9中导入。Node中的每个模块都有一个名为exports的内置对象，它以一个空对象开头。当我们从另一个文件中需要时，我们添加到exports对象的任何内容都可用。

**清单4.8 在Node中导出函数：basic-math.js**

exports.addTwo = n => n + 2;

**清单4.9 在Node中导入函数**

const basicMath = require('./basic-math'); basicMath.addTwo(4); // returns 6

###### *使用(require)来自其他进程的功能*

内置的require函数不能跨进程工作。当我们在渲染器进程中工作时，我们从内置的require函数中使用的任何功能都将成为渲染器进程的一部分。当我们在主进程中工作时，我们需要的任何功能都将成为主进程的一部分。但是当我们进入渲染器进程并且我们想要从主进程中获取功能时会发生什么？

Electron的远程模块有自己的require方法，允许它在我们的渲染器过程中需要主进程的功能。使用remote.require返回代理对象,就像远程对象上的其他属性一样。Electron代表我们处理所有进程间通信。

为了实现我们在本章开头提出的功能，主进程必须导出其getFileFromUser()函数，以便我们可以将其导入到渲染器代码中。此列表更新app/main.js中的单行。

**清单4.10 导出从渲染器进程打开文件对话框的功能：./app/main.js**

**除了在此文件中创建常量之外，我们还将其指定为exports对象的属性，该属性可以从其他文件（特别是渲染器进程）访问。**

const getFileFromUser = exports.getFileFromUser = () => { const files = dialog.showOpenDialog(mainWindow, {

properties: ['openFile'], filters: [

{ name: 'Text Files', extensions: ['txt'] },

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

if (!files) { return; } const file = files[0];

const content = fs.readFileSync(file).toString();

console.log(content);

};

代码采用我们创建的getFileFromUser( )函数，并将其作为导出对象上具有相同名称的属性导出。渲染过程需要引入Electron的远程模块，然后使用remote.require函数在我们的渲染器过程中从主进程获取对getFileFromUser( )函数的引用。这与清单4.11中显示的内置require函数不同，因为导入的代码是根据主进程计算的，而不是需要它的渲染器进程。这可以通过四个步骤完成：

1.在我们的渲染器过程中需要Electron。

2.存储对远程模块的引用。

3.使用remote.require来引入主进程。

4.存储对从主进程导出的getFileFromUser( )函数的引用。

**清单4.11 在渲染器进程中需要主进程的函数：./app/renderer.js**

const { remote } = require('electron');

const mainProcess = remote.require('./main.js');

我们现在可以调用我们在渲染器进程中从主进程导出的getFileFromUser()函数。让我们替换事件侦听器中的功能来触发 Open File对话框，而不是触发alert。

**清单4.12从UI触发主进程中的getFileFromUser()：./app/renderer.js**

openFileButton.addEventListener('click', () => { mainProcess.getFileFromUser();

});

如果我们启动Electron应用程序并单击 Open File 按钮，它会正确触发 Open File 对话框。有了这个，我们仍然只将文件记录到主进程中的控制台。要完成我们的功能，主进程必须将文件的内容发送回渲染器进程以显示在我们的应用程序中。

### *从主进程发送content到渲染器进程*

远程模块有助于从我们的渲染器进程中的访问主进程功能，但它不允许反向。要将用户选择的文件的内容发送回渲染器进程以在UI中呈现，我们需要学习另一种在进程之间进行通信的技术。

每个BrowserWindow实例都有一个名为webContents的属性，它存储一个对象，该对象负责我们在调用new BrowserWindow( )时创建的Web浏览器窗口。webContents类似于app，因为它根据渲染器进程中网页的生命周期发出事件。

以下是您可以在webContentsobject上监听的一些事件的不完整列表：

* did-start-loading
* did-stop-loading
* dom-ready
* blur
* focus
* resize
* enter-full-screen
* leave-full-screen

webContents还有许多方法可以从主进程触发渲染器进程中的不同功能。在上一章中，我们使用mainWindow.webContents.openDevTools()从主进程在渲染器进程中打开了Chrome开发人员工具。mainWindow.loadURL（'file：// $ {\_dirname}/index.html'）是mainWindow.webContents.loadURL()的别名，在应用程序首次启动时将我们的HTML文件加载到渲染器进程中。图4.14显示了更多别名。

is an alias for

mainWindow.loadURL()

mainWindow.webContents.loadURL()

is an alias for

mainWindow.reload()

mainWindow.webContents.reload()

is an alias for

mainWindow.print()

mainWindow.webContents.print()

is an alias for

mainWindow.printToPDF()

mainWindow.webContents.printToPDF()

**图4.14 BrowserWindow实例的方法是Electron的webContents API的别名。**

webContents有一个名为send()的方法，它将信息从主进程发送到渲染器进程。webContents.send()接受可变数量的参数。第一个参数是一个任意字符串，是发送消息的通道的名称。渲染器进程中的事件监听器侦听同一通道。当我们看到它在运行时，这种流程将变得更加清晰。第一个之后的所有后续参数都传递给渲染器进程。

###### *发送文件content到渲染器进程*

我们当前的实现读取用户选择的文件并将其记录到终端。mainWindow.webContents.send()将文件的内容发送到渲染器进程。下一章将介绍打开不需要提示用户选择特定文件的对话框的文件的其他方法，因为我们遇到的情况是我们要在不触发对话框的情况下打开文件。

**清单4.13 将内容从主进程发送到渲染器进程：./ app/main.js**

const getFileFromUser = exports.getFileFromUser = () => { const files = dialog.showOpenDialog(mainWindow, {

properties: ['openFile'], filters: [

{ name: 'Text Files', extensions: ['txt'] },

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

if (files) { openFile(files[0]); }

};

const openFile = (file) => {

const content = fs.readFileSync(file).toString(); mainWindow.webContents.send('file-opened', file, content);

};

**以前，如果文件未定义，我们使用return语句中断函数。在这个例子中，当dialog.showOpenFile( )成功返回文件路径数组时，我们将翻转该逻辑并将第一个文件传递给Open File。**

**我们将通过file-opened通道将文件名及其内容发送到渲染器进程。**

主进程现在通过file-opened的通道广播文件的名称及其内容。下一步是使用ipcRenderer模块在渲染器进程中的file-opened通道上设置监听器。Electron附带两个基本模块，用于在进程之间来回发送消息：ipcRenderer和ipcMain。每个模块仅在与其共享名称的进程类型中可用。

ipcRenderer可以向主进程发送消息。对我们的直接需求更重要的是，它还可以监听使用webContents.send()从主进程发送的消息。它需要渲染器进程中的ipcRenderer模块。

**清单4.14 导入ipcRenderer模块：./app/renderer.js**

**const { remote, ipcRenderer } = require('electron');**

const mainProcess = remote.require('./main.js');

**我们将在渲染器进程中导入ipcRenderer模块。**

有了这个，我们现在可以建立一个监听器。ipcRenderer监听file-opened的通道，将内容添加到页面，并将Markdown呈现为HTML。

**清单4.15 在file-opened的通道上收听消息：./ app/renderer.js**

ipcRenderer.on('file-opened', (event, file, content) => { markdownView.value = content; renderMarkdownToHtml(content);

});

ipcRenderer.on()有两个参数：要监听的通道和一个回调函数，该函数定义了渲染器进程在您正在设置监听器的通道上接收消息时要采取的操作。调用回调函数时会提供一些参数。第一个是事件对象，就像浏览器中的普通事件监听器一样。它包含有关我们为其设置监听器的事件的信息。附加参数是在主进程中使用webContents.send( )时提供的参数。在清单4.13中，我们发送了文件名及其内容。这些将是传递给我们的监听器的附加参数。

通过这些新增功能，用户现在可以单击Open File按钮，使用本机文件对话框选择文件，然后在UI中呈现内容。我们已经成功实现了我们在本章开头要实现的功能。我们的主进程和渲染器进程的代码应该类似于下面的两个清单。

**清单4.16 在主进程中实现的Open File功能：./app/main.js**

const { app, BrowserWindow, dialog } = require('electron'); const fs = require('fs');

let mainWindow = null; app.on('ready', () => {

mainWindow = new BrowserWindow({ show: false });

mainWindow.loadFile('index.html'); mainWindow.once('ready-to-show', () => {

mainWindow.show();

});

mainWindow.on('closed', () => { mainWindow = null;

});

});

const getFileFromUser = exports.getFileFromUser = () => {

const files = dialog.showOpenDialog(mainWindow, {

properties: ['openFile'], filters: [

{ name: 'Text Files', extensions: ['txt'] },

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

if (files) { openFile(files[0]) }

};

const openFile = (file) => {

const content = fs.readFileSync(file).toString(); mainWindow.webContents.send('file-opened', file, content);

};

**清单4.17 实现的Open File功能：./app/renderer.js**

const { remote, ipcRenderer } = require('electron'); const mainProcess = remote.require('./main.js');

const marked = require('marked');

const markdownView = document.querySelector('#markdown'); const htmlView = document.querySelector('#html');

const newFileButton = document.querySelector('#new-file'); const openFileButton = document.querySelector('#open-file');

const saveMarkdownButton = document.querySelector('#save-markdown'); const revertButton = document.querySelector('#revert');

const saveHtmlButton = document.querySelector('#save-html'); const showFileButton = document.querySelector('#show-file');

const openInDefaultButton = document.querySelector('#open-in-default');

const renderMarkdownToHtml = (markdown) => {

htmlView.innerHTML = marked(markdown, { sanitize: true });

};

markdownView.addEventListener('keyup', (event) => {

const currentContent = event.target.value; renderMarkdownToHtml(currentContent);

});

openFileButton.addEventListener('click', () => {

mainProcess.getFileFromUser();

});

ipcRenderer.on('file-opened', (event, file, content) => {

markdownView.value = content; renderMarkdownToHtml(content);

});

### *本章小结*

* + - * Electron提供了用于创建各种本机操作系统对话框的对话模块。
      * 打开对话框可以配置为允许单个文件或目录以及多个文件或目录。
      * 可以配置打开对话框以允许用户仅选择某些文件类型。
      * 打开对话框返回一个数组，该数组由用户选择的一个或多个文件或目录组成。
      * Electron不具备读取文件的能力。相反，我们使用Node的fs模块来读取和写入文件系统。
      * 每个操作系统都提供不同的功能。如果给定操作系统中不存在该功能，Electron将使用可用功能，同时提供优雅的回退。
      * 在macOS中，我们可以通过提供对该窗口的引用作为dialog.showOpenDialog( )中的第一个参数，从一个窗口下拉一个对话框作为工作表。
      * 原生操作系统API和文件系统访问应由主进程处理，而渲染UI和响应用户输入应由渲染器进程处理。
      * Electron为主进程和渲染器进程提供了一组不同的模块。

*第5章 使用多窗口*

本章包括  
使用JavaScript Set数据结构跟踪多个窗口  
促进主进程和多个渲染器进程之间的通信  
使用Node API来检测运行应用程序的平台

现在，当Fire Sale启动时，它会为UI创建一个窗口。该窗口关闭后，应用程序退出。虽然这种行为是完全可以接受的，但我们通常希望能够打开多个独立的窗口。在本章中，我们将Fire Sale从单窗口应用程序转换为支持多个窗口的应用程序。在此过程中，我们将探索新的Electron API以及一些JavaScript最新增加的内容。我们还探讨了在采用配置为与一个渲染器进程通信的主进程（参见图5.1）时发生的问题的解决方案，并重构它以管理可变数量的进程（参见图5.2）。本章末尾的完整代码可以在http://mng.bz/V145找到。然而，我们从第4章分支开始。

我们首先实例化一个Set数据结构，该结构在2015年被添加到JavaScript,我们用它来跟踪所有用户的窗口。接下来，我们创建一个管理单个窗口生命周期的函数。在这之后，我们修改了我们在第4章中创建的功能，以提示用户选择文件并打开它以定位正确的窗口。此外，我们还会处理一些常见的边缘情况和沿途出现的其他怪现象，例如相互遮挡的窗口。

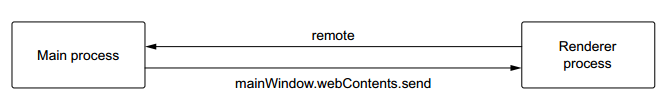


图5.1 在第4章中，我们建立了主进程和一个渲染器进程之间的通信。



图5.2 在本章中，我们更新Fire Sale以支持多个窗口并促进它们之间的通信。

### *创建和管理多个窗口*

集合(Set)是JavaScript的新数据结构，并在ES2015规范中添加。集合是不重复元素的集合; 数组中可以包含重复值。我选择使用集合而不是数组，因为它更容易删除元素。此清单显示如何在JavaScript中创建Set。

**清单5.1 创建一个跟踪新窗口的Set：./app/main.js**

const windows = new Set();

使用数组，我们必须找到窗口的索引并将其删除，或者创建一个没有该窗口的数组。这两种方法都不像在集合上调用delete方法那样简单，并且将引用传递给我们要删除的窗口。

使用数据结构来跟踪所有应用程序的窗口，下一步是将创建BrowserWindow（清单5.2）的过程从应用程序的“ready”事件侦听器移到其自己的函数中。

**清单5.2 实现创建新窗口的函数：./app/main.js**

const createWindow = exports.createWindow = () => { let newWindow = new BrowserWindow({ show: false });

~~newWindow.loadFile('index.html');~~ newWindow.once('ready-to-show', () => {

newWindow.show();

});

newWindow.on('closed', () => { windows.delete(newWindow); newWindow = null;

});

windows.add(newWindow); return newWindow;

};

**关闭后，从窗口集中删除引用**

将窗口添加到打开时设置的窗口

createWindow()函数创建一个BrowserWindow实例，并将其添加到我们在清单5.1中创建的一组窗口中。接下来，我们重复前面章节中创建新窗口的步骤。关闭窗口会将其从集合中删除。最后，我们返回对刚刚创建的窗口的引用。我们在下一章中需要这个引用。

应用程序准备就绪后，调用新的createWindow()函数，如下面的清单所示。应用程序应该以与实现此更改之前相同的方式启动，但它也设置了在其他上下文中创建其他窗口的阶段。

**清单5.3 应用程序就绪时创建窗口：./app/main.js**

app.on('ready', () => { createWindow();

});

应用程序像以前一样启动，但如果您尝试单击Open File按钮，您会发现它已损坏。这是因为我们仍然在几个地方引用mainWindow。它在dialog.showOpenDialog()中引用，以在macOS中将对话框显示为工作表。更重要的是，在从文件系统读取文件内容并将其发送到窗口后，在openFile()中引用它。

###### *在主进程和多个窗口之间进行通信*

多个窗口产生一个问题：我们向哪个窗口发送文件路径和内容？ 要支持多个窗口，这两个函数必须引用应显示对话框的窗口和发送的内容，如图5.3所示。

**使用远程模块传递对自身的引用**

**将文件路径和内容发送到请求窗口**

**传递窗口引用和文件路径**

getFileFromUser openFile

主进程

渲染器进程

**图5.3要确定发送文件内容的窗口，渲染器进程在与主进程通信以调用getFileFromUser()时必须向自身发送引用。**

在清单5.4中，让我们重构getFileFromUser( )函数以接受给定窗口作为参数，而不是总是假设范围中有mainWindow实例。

**清单5.4 重构getFileFromUser()以使用特定窗口：./app/main.js**

**dialog.showOpenDialog()接受对浏览器窗口对象的引用**

**引用浏览器窗口以确定哪个窗口应显示文件对话框，然后加载用户选择的文件。**

const getFileFromUser = exports.getFileFromUser = (targetWindow) => { const files = dialog.showOpenDialog(targetWindow, {

properties: ['openFile'],

filters: [

{ name: 'Text Files', extensions: ['txt'] },

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

if (files) { openFile(targetWindow, files[0]); }

};

**openFile()函数接受对浏览器窗口对象的引用，以确定哪个窗口应该接收用户打开的文件的内容。**

在代码摘要中，我们修改了getFileFromUser( )以将对窗口的引用作为参数。我避免命名参数窗口，因为它可能与浏览器中的全局对象混淆。用户选择文件后，除了文件路径外，我们还将targetWindow传递给openFile( )，如下所示。

**清单5.5 重构openFile()以使用特定窗口：./app/main.js**

将文件的内容发送到提供的浏览器窗口

**接受对浏览器窗口对象的引用**

const openFile = exports.openFile = (targetWindow, file) => { const content = fs.readFileSync(file).toString(); targetWindow.webContents.send('file-opened', file, content);

};

###### *将对当前窗口的引用传递给主进程*

从文件系统读取文件内容后，我们将文件的路径和内容作为第一个参数传递给传入的窗口。然而，这提出了一个问题：我们如何获得对窗口的引用？

使用远程模块从渲染器进程调用getFileFromUser( )以促进与主进程的通信。正如我们在前一章中看到的那样，远程模块包含对主进程独有的所有模块的引用。事实证明，remote还有一些其他方法,特别是remote.getCurrentWindow( )，它返回对调用它的BrowserWindow实例的引用，如下所示。

**清单5.6 获取对渲染器进程中当前窗口的引用：./app/renderer.js**

const currentWindow = remote.getCurrentWindow();

现在我们有了对窗口的引用，完成该功能所需的最后一步是将它传递给getFileFromUser()。这让主进程中的功能知道我们即将成为他们正在使用的许多浏览器窗口。

**Listing 5.7 将对当前窗口的引用传递给主进程:./app/renderer.js**

openFileButton.addEventListener('click', () => {

mainProcess.getFileFromUser(currentWindow);

});

当我们在第3章中为UI实现Markup时，我们添加了一个New File按钮。我们现在在主进程中实现并导出了createWindow()函数。我们也可以快速连接该按钮。

**清单5.8 向newFileButton添加监听器：./app/renderer.js**

newFileButton.addEventListener('click', () => { mainProcess.createWindow();

});

我们可以在主进程中对多个窗口的实现进行一些增强，但是我们已经完成了本章的渲染器进程。app/renderer.js中代码的当前状态如下。

**清单5.9 在渲染器进程中实现的newFileButton：./app/renderer.js**

const { remote, ipcRenderer } = require('electron'); const mainProcess = remote.require('./main.js'); const currentWindow = remote.getCurrentWindow();

const marked = require('marked');

const markdownView = document.querySelector('#markdown'); const htmlView = document.querySelector('#html');

const newFileButton = document.querySelector('#new-file'); const openFileButton = document.querySelector('#open-file');

const saveMarkdownButton = document.querySelector('#save-markdown'); const revertButton = document.querySelector('#revert');

const saveHtmlButton = document.querySelector('#save-html'); const showFileButton = document.querySelector('#show-file');

const openInDefaultButton = document.querySelector('#open-in-default');

const renderMarkdownToHtml = (markdown) => { htmlView.innerHTML = marked(markdown, { sanitize: true });

};

markdownView.addEventListener('keyup', (event) => { const currentContent = event.target.value; renderMarkdownToHtml(currentContent);

});

newFileButton.addEventListener('click', () => { mainProcess.createWindow();

});

openFileButton.addEventListener('click', () => { mainProcess.getFileFromUser(currentWindow);

});

ipcRenderer.on('file-opened', (event, file, content) => { markdownView.value = content; renderMarkdownToHtml(content);

});

***改善创建新窗口的用户体验* 93**

### *改善创建新窗口的用户体验*

在上一章中实现事件监听器后单击New File按钮时，您可能会对它是否正常工作感到困惑。您可能已经注意到窗口周围的阴影变暗，或者您可能已经单击并拖动新窗口并显示下面的前一个窗口。

我们现在遇到的一个小问题是，每个新窗口都出现在与第一个窗口相同的默认位置，并完全遮盖它。如果新窗口稍微偏离前一个窗口，则可能更明显地创建新窗口，如图5.4所示。此列表显示如何偏移窗口。

**清单5.10 根据当前关注的窗口偏移新窗口：./app/main.js**

如果上一步中存在当前活动的窗口，则将下一个窗口的坐标设置为当前活动窗口的下方和右侧。

const createWindow = exports.createWindow = () => {

let x, y;

const currentWindow = BrowserWindow.getFocusedWindow();

**获取当前处于活动状态的浏览器窗口。**

if (currentWindow) {

const [ currentWindowX, currentWindowY ] = currentWindow.getPosition(); x = currentWindowX + 10;

y = currentWindowY + 10;

}

let newWindow = new BrowserWindow({ x, y, show: false }); newWindow.loadFile('index.html');

newWindow.once('ready-to-show', () => { newWindow.show();

});

newWindow.on('closed', () => { windows.delete(newWindow); newWindow = null;

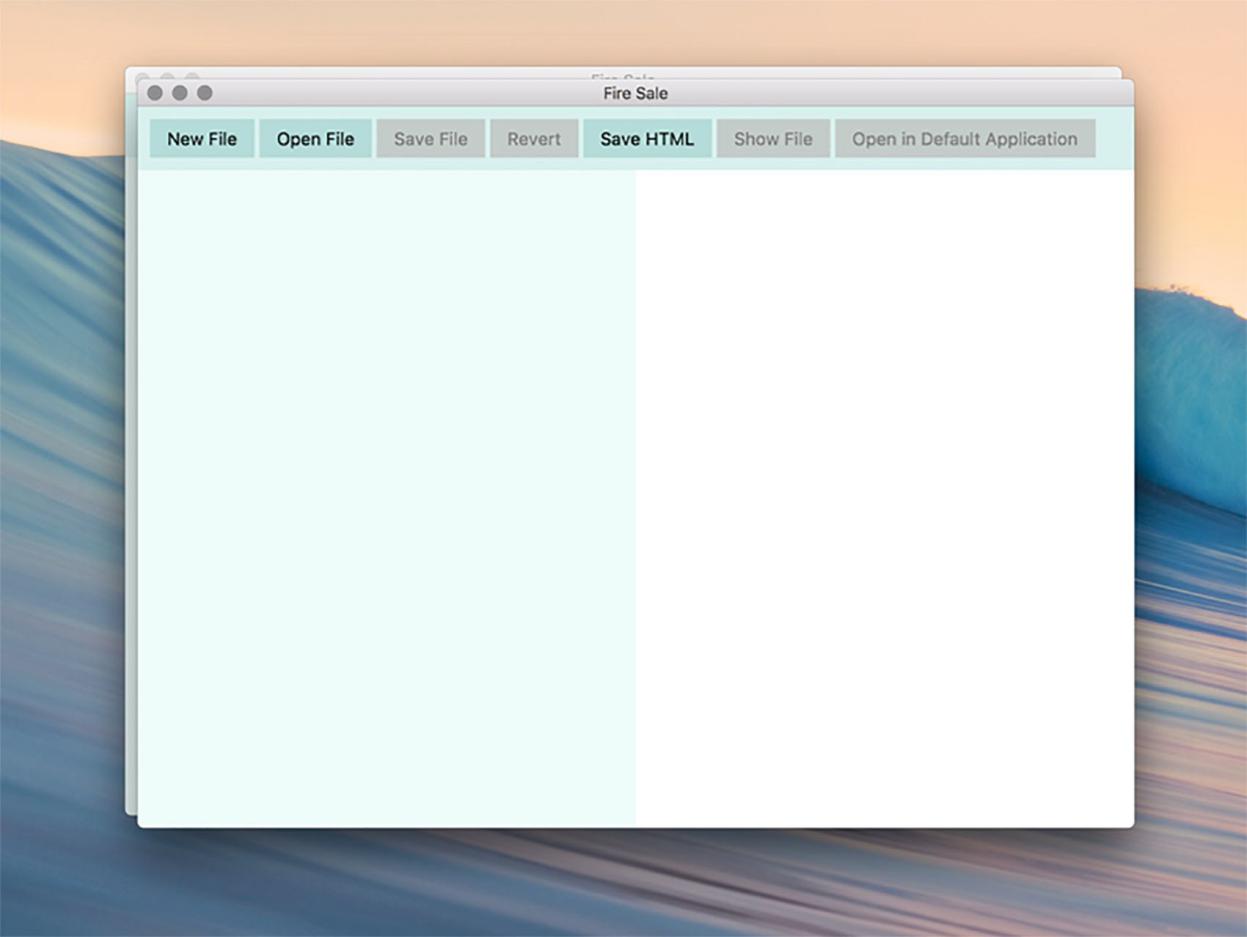
});

windows.add(newWindow); return newWindow;

};

**创建新窗口，首先使用x坐标和y坐标隐藏它。如果上一步中的代码已运行，则会设置这些设置，如果不执行，则设置为未定义，在这种情况下，窗口将在默认位置创建。**

除了使用new关键字实例化实例之外，BrowserWindow模块还具有自己的方法。我们可以使用BrowserWindow.getFocusedWindow()来获取用户当前正在使用的窗口的引用。当应用程序第一次就绪并且我们调用createWindow()时，没有一个聚焦窗口，而BrowserWindow.getFocusedWindow()返回undefined。如果有一个窗口，我们调用它的getWindow()方法，该方法返回一个带有窗口x和y坐标的数组。我们将这些值存储在条件块之外的两个变量中，并将它们传递给BrowserWindow构造函数。如果它们仍然未定义（例如，没有聚焦窗口），那么Electron使用默认值，就像我们实现此功能之前一样。图5.4显示了与第一个窗口偏移的第二个窗口。



**图5.4 新窗口偏离当前窗口。**

这不是实现此功能的唯一方法。或者，您可以跟踪初始的x和y位置，并在每个新窗口上递增这些值。或者，您可以为默认的x和y值添加一点点随机性，以便每个窗口稍微偏移。我将这些方法作为练习留给读者。

***与macOS集成* 95**

### *与macOS集成*

在macOS中，许多（但不是全部）应用程序保持打开状态，即使它们的所有窗口都已关闭。例如，如果您关闭了Chrome中的所有窗口，则应用程序将在Dock中保持活动状态，并仍显示在应用程序切换器中。Fire Sale还做不到这样。

在前面的章节中，这可能是可以接受的。我们有一个窗口，无法创建额外的窗口。在本节中，我们使应用程序仅在macOS中保持打开状态。默认情况下，Electron在触发window-all-closed的事件时退出应用程序。如果我们在macOS上运行, 如果我们想要阻止这种行为，我们必须监听这个事件并有条件地阻止它关闭。

**清单5.11 关闭所有窗口时保持应用程序活动：./ app/main.js**

app.on('window-all-closed', () => {

if (process.platform === 'darwin') {

return false;

**检查应用程序是否在macOS上运行**

}

app.quit();

});



**如果不是，则退出应用程序**

**如果是，则返回false以阻止默认操作**

进程(process)对象由Node提供，全局可用，无需任何操作。process.platform返回应用程序当前正在执行的平台的名称。在撰写本文时，process.platform返回五个字符串之一：darwin，freebsd，linux，sunos或win32。Darwin是构建macOS的UNIX操作系统。在清单5.11中，我们检查了process.platform是否等于darwin。如果是，则应用程序在macOS上运行，我们希望返回false以阻止发生默认操作。

让应用程序保持活动状态是成功的一半。如果用户在Dock中单击应用程序并且没有打开窗口会发生什么？ 在这种情况下，Fire Sale应该打开一个新窗口并将其显示给用户，如下所示。

**清单5.12 在打开应用程序并且没有窗口时创建一个窗口：./app/main.js**

// Electron提供了hasVisibleWindows参数，它是一个布尔值。

app.on('activate', (event, hasVisibleWindows) => {

// 如果用户激活应用程序时没有可见窗口，则创建一个。

if (!hasVisibleWindows) { createWindow(); }

});

activate事件将两个参数传递给提供的回调函数。第一个是事件对象。第二个是布尔值，如果任何窗口可见则返回true，如果所有窗口都关闭则返回false。在后者的情况下，我们调用我们在本章前面编写的createWindow( )函数。

activate事件仅在macOS上触发，但有很多原因导致您可能选择让您的应用程序在Windows或Linux上保持打开状态，特别是如果应用程序正在运行您希望继续的后台进程，即使窗口被关闭也是如此。另一种可能性是您有一个可以隐藏或使用全局快捷方式显示的应用程序，或者从托盘或菜单栏中显示。我们将在后面的章节中实现其中的每一个。

通过这两个额外的事件，我们将Fire Sale从单窗口应用程序转换为支持多个窗口的应用程序。此清单以当前形式显示主进程的代码。

**清单5.13 在主进程中实现的多个窗口：./app/main.js**

const { app, BrowserWindow, dialog } = require('electron'); const fs = require('fs');

const windows = new Set();

app.on('ready', () => {

createWindow();

});

app.on('window-all-closed', () => {

if (process.platform === 'darwin') {

return false;

}

});

app.on('activate', (event, hasVisibleWindows) => {

if (!hasVisibleWindows) { createWindow(); }

});

const createWindow = exports.createWindow = () => {

let x, y;

const currentWindow = BrowserWindow.getFocusedWindow(); if (currentWindow) {

const [ currentWindowX, currentWindowY ] = currentWindow.getPosition();

x = currentWindowX + 10; y = currentWindowY + 10;

}

let newWindow = new BrowserWindow({ x, y, show: false }); newWindow.loadFile('index.html');

newWindow.once('ready-to-show', () => { newWindow.show();

});

newWindow.on('closed', () => { windows.delete(newWindow);

***本章小结* 97**

newWindow = null;

});

windows.add(newWindow); return newWindow;

};

const getFileFromUser = exports.getFileFromUser = (targetWindow) => {

const files = dialog.showOpenDialog(targetWindow, {

properties: ['openFile'], filters: [

{ name: 'Text Files', extensions: ['txt'] },

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

if (files) { openFile(targetWindow, files[0]); }

};

const openFile = exports.openFile = (targetWindow, file) => {

const content = fs.readFileSync(file).toString(); targetWindow.webContents.send('file-opened', file, content);

};

### *本章小结*

* 当创建具有多个窗口的Electron应用程序时，我们无须再对主进程发送数据的窗口进行硬编码。
* 我们可以使用Electron的远程(remote)模块向渲染器进程中的窗口询问对自身的引用，并在与主进程通信时发送该引用。
* 所有窗口关闭时，macOS上的应用程序并不总是退出。我们可以使用Node的进程(process)对象来确定应用程序在哪个平台上运行。
* 如果process.platform是darwin，则应用程序在macOS上运行。
* 在监听app的windows-all-closed事件的函数中返回false会阻止应用程序退出。
* 在macOS上，app会在用户单击停靠栏图标时触发激活事件。
* activate事件包含一个名为hasVisibleWindows的布尔值作为传递给回调函数的第二个参数。如果当前打开任何窗口，则为true;如果没有，则为false。我们可以使用它来确定是否应该打开一个新窗口。

*使用文件*

***本章包括***

* 确定内容是否已编辑且未保存
* 根据当前活动文档的状态修改窗口的标题
* 在macOS上运行的应用程序中使用Windows可用的自定义交互
* 实现将文档附加到操作系统的最近文档列表中
* 从操作系统中查看当前文件的更改

在前两章中，我们实现了从文件系统读取文件并将其显示在应用程序的浏览器窗口中的功能。本练习演示了Electron中的进程间通信如何工作，以及Electron弥合传统基于浏览器的应用程序和Node.js应用程序之间差距的能力。在上一章中，我们还添加了对多个浏览器窗口的支持。

为了清晰起见，我将我们最初的实现保持得非常简单。事实证明，用户以惊人的方式与文件进行交互,即使是像Fire Sale这样的简单笔记应用程序。用户可能会从应用程序最初启动时生成的空窗口开始编写新笔记，或者他们可能会选择从文件系统打开现有文件。用户可能会单击我们之前实现的Open File按钮，或者他们可能会从最近打开的文档列表中选择该文件。用户选择的路径会影响应用程序在保存时的行为方式。如果它是新文件，那么我们必须提示用户提供写入新文件的位置。如果它是现有文件，则应用程序应该只覆盖用户最初从Open File对话框中选择的文件。

用户已经开始期待现代应用程序的几个功能。我们的应用程序应提供视觉指示，表明文件自原始打开或上次保存后已更改。它应该与操作系统的最近文档列表集成，并遵循操作系统级别约定，了解窗口标题栏的外观，具体取决于文件的状态。

在本章中，我们重新思考管理文件的方法，因为我们实现了保存应用程序的Markdown文本，导出呈现的HTML以及将更改还原为未保存文件的功能。我们还探索了打开文件的其他方法，例如使用HTML5 File API在应用程序的左窗格中实现拖放功能。最后，我们不希望我们的用户最终在某些其他应用程序中编辑Markdown文件，并在Fire Sale中保存文件时意外覆盖更改。因此，我们监听其他应用程序对当前文件的外部更改。在本章中，我们将实现图6.1中列出的所有功能。

我将从存储库的第5章分支开始作为起点。您还可以在第6章分支中找到已完成的示例。

### *跟踪当前文件*

当我们开始在我们的应用程序中使用不同的文件时，跟踪我们当前正在使用的文件是有帮助的。这样，如果用户要求保存，我们就知道用户正在处理哪个文件。假设Mildred有一个重要的灵感，并希望将她的天才想法记录到Fire Sale中。她打开了她的重要思想清单，记下了笔记。目前，Fire Sale不知道Mildred是否正在编辑新文件或现有文件。当应用程序选择要打开的文件时，该应用程序也无法跟踪Mildred正在处理的文件。

要解决此问题，我们需要实现跟踪Fire Sale当前使用的文件的能力。在第4章中，每当打开一个新文件时，我们都会将文件的内容以及刚从主进程打开的文件的路径发送到请求该文件的渲染器进程。我们使用内容填充了应用程序的左右窗格。我们的下一步是使渲染器进程能够跟踪当前显示给用户的文件的路径。这样做可以让我们保存更改，而不会提示用户输入文件位置。

用户编辑磁盘上文件

User saves an existing file

用户保存新文件

Prompt user for

new file name

用户编辑未保存的文件

将内容写入文件系统

opens with an already active file

在UI中渲染内容

更新窗口标题

更新已编辑的状态和窗口标题

读取文件的内容

User opens a file

User closes a file or the window

Stop watching the previous file for changes

图6.1用户希望以多种方式与文件交互。在本章中，我们将实现所有必要的功能，以满足用户的期望。

开始监视新文件更改

我们在UI中也有一个Revert按钮。如果用户单击此按钮，则应回滚所有未保存的更改并将内容返回到上次保存的状态。处理此操作的一种简单方法是在打开文件时存储内容的副本。如果用户单击“还原”，则Fire Sale会使用上次打开文件时的缓存内容替换UI中的内容。

让我们从一些合理的默认值开始，用户打开一个尚未绑定到给定文件的新窗口。我们在渲染器进程的顶级声明了两个变量，以跟踪当前文件的原始内容及其文件路径。

**清单6.1 为跟踪当前文件声明全局变量：./app/renderer.js**

let filePath = null;

let originalContent = '';

我选择为originalContent使用空字符串而不是null，因为这是浏览器中空输入或textarea的值。在本章后面，此设置使我们更容易查看是否已编辑新文档。

每当打开文件并将其发送到渲染器进程时，我们都需要更新这些值。我们在第4章中设置的file-opened通道上的IPC事件监听器中处理这个问题。

**清单6.2 打开新文件时更新全局变量：./app/renderer.js**

ipcRenderer.on('file-opened', (event, file, content) => {

**filePath = file; originalContent = content;**

markdownView.value = content; renderMarkdownToHtml(content);

**更新存储在顶级作用域中的当前打开文件的路径**

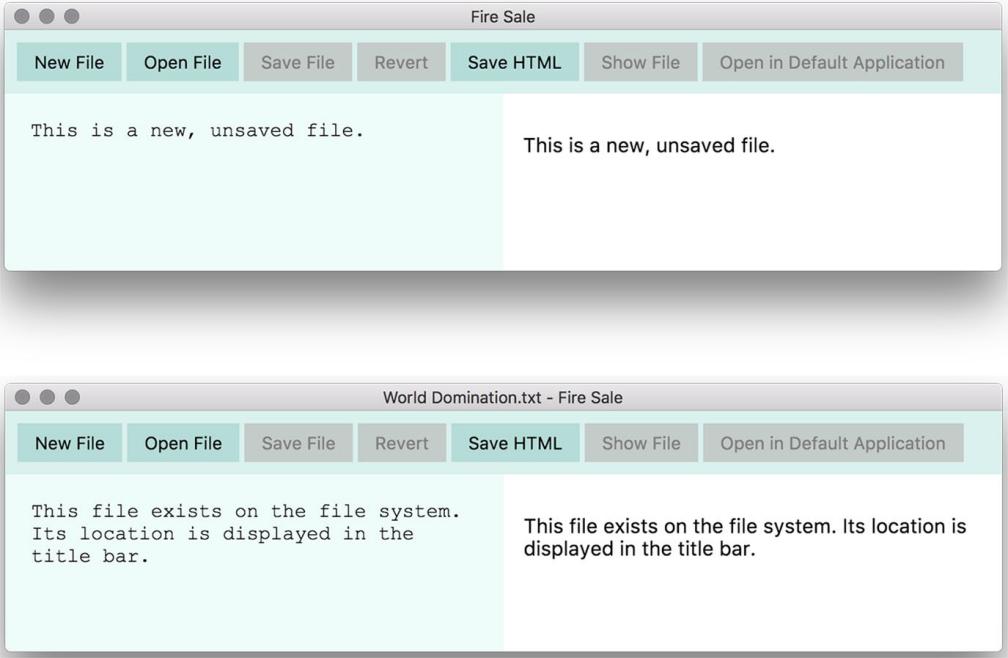
});

**更新UI中的HTML内容更新UI中的Markdown内容**

**更新原始内容以确定文件是否有未保存的更改**

###### *根据当前文件更新窗口标题*

Mildred可以在她的计算机上打开文件，编辑它并保存更改，但她目前无法知道她正在使用哪个文件。这些是本周的会议纪要，还是她在格陵兰度假时错过的会议？ 她有多个窗口打开，无法分辨哪个是哪个。在上一节中，我们实现了一项功能，允许Fire Sale中的每个窗口保持对当前文件的引用，但我们没有向UI添加任何内容以与用户共享该信息。桌面应用程序的常见模式是在窗口的标题栏中显示当前处于活动状态的文件的名称，如图6.2所示。在本节中，我们将遵循最佳实践并在Fire Sale中实施此模式。



**图6.2 文件系统中文件的名称显示在窗口标题中。**

默认情况下，应用程序的窗口显示HTML页面的标题，该页面在app/index.html中定义。这是合理的默认设置，但许多本机桌面应用程序显示当前文件的名称或路径。一种方法是在用户打开新文件时更新窗口的标题。除了显示当前打开文件的名称（如清单6.3所示）之外，我们可能还希望在窗口标题中显示其他信息，例如当前文件自保存以来是否已被编辑。我们还需要为几个不同的上下文更新窗口的标题，例如编辑和保存文件。

所有BrowserWindow实例都有一个名为setTitle( )的方法，它允许我们以编程方式操作窗口的标题。在本章的后面部分，我们将显示自上次保存文件或打开文件以来文件是否已修改的信息。我们创建了一个名为updateUserInterface( )的方法，该方法最终封装了所有这些逻辑以及其他一些功能，如果文件包含未保存的更改，则启用Save File和 Revert 按钮。

**Listing 6.3 根据当前文件更新窗口标题: ./app/renderer.js**

const path = require('path');

const updateUserInterface = () => {

let title = 'Fire Sale';

**如果文件已打开，将该文件的名称添加到标题中**

if (filePath) { title = `${path.basename(filePath)} - ${title}`; }

currentWindow.setTitle(title);

}; **更新窗口的标题**

我们从默认标题开始。如果文件当前处于打开状态，我们会修改标题以包含文件路径。路径可能很长，大多数信息（例如文件系统的根目录或用户文件夹的存储位置）对我们的用户来说并不重要。我们使用path.basename( )从完整文件路径中提取文件本身的名称。最后，我们引用我们在第5章中定义的当前窗口并设置其标题。每当打开一个新文件时，我们都会将此函数称为最后一步。

**清单6.4 打开新文件时调用updateUserInterface( )：./app/renderer.js**

ipcRenderer.on('file-opened', (event, file, content) => {

filePath = file;

originalContent = content;

markdownView.value = content; renderMarkdownToHtml(content);

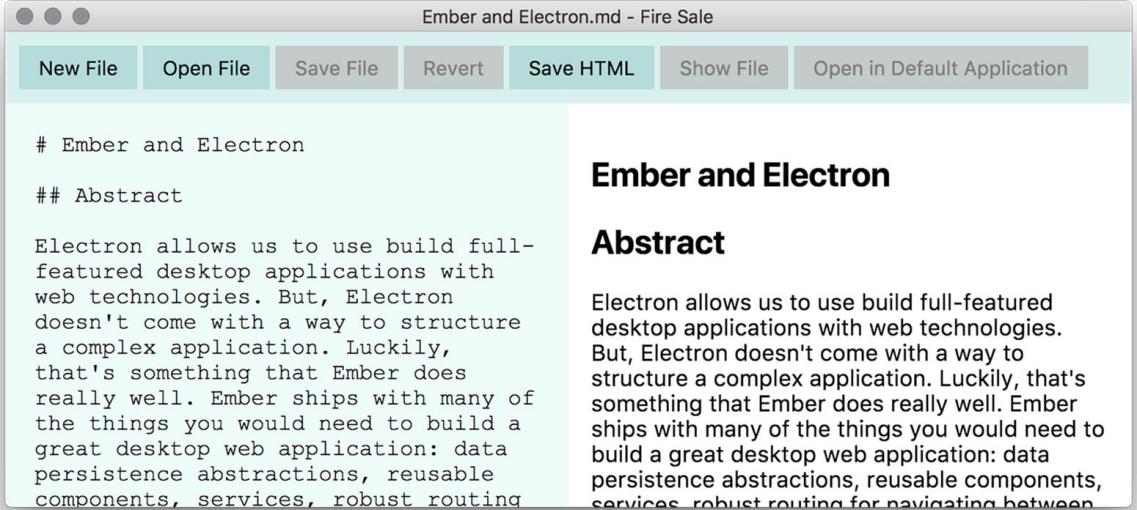
updateUserInterface();

});

**每当打开新文件时，调用更新窗口标题栏的方法。**

###### *确定当前文件是否已更改*

在Fire Sale中做重要笔记的过程中，Mildred意识到她迟到了20分钟。她想在赶到五楼之前关闭Fire Sale，但她不确定她是否已将她最近的更改保存到她正在处理的文件中。我们有很多很好的理由来跟踪用户是否在打开文件后编辑了文件。我们可能想要提示用户是否尝试关闭窗口并且他们有未保存的更改。或者，如果文件已被修改，我们可能只想显示某些UI元素（参见图6.3）。



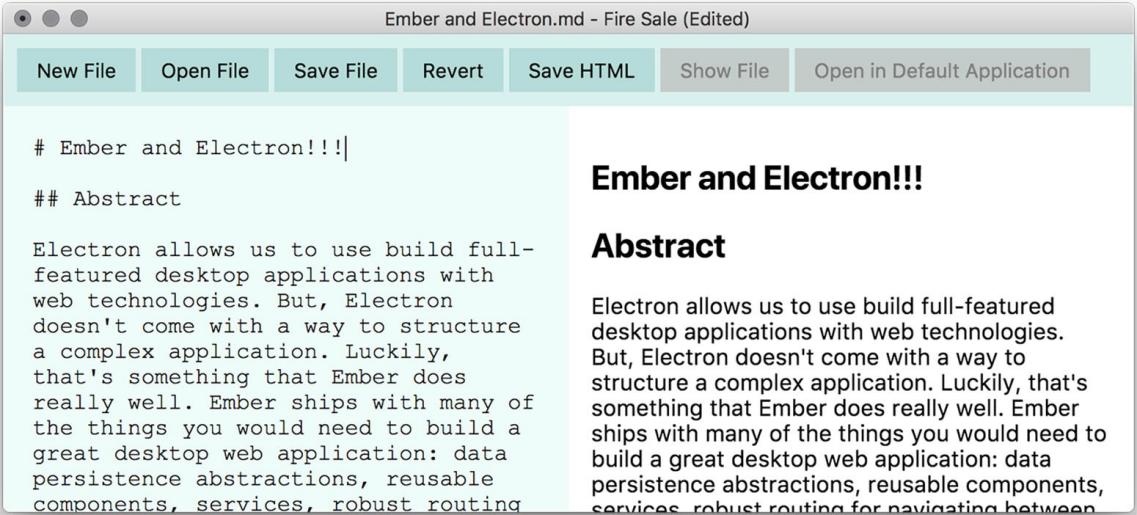
**图6.3 未修改文件时的Fire Sale UI。请注意，已禁用Save File和Revert按钮。**

在本节中，我们在UI中添加了一个视觉提示。仅当文件已被修改时，才会启用Save File按钮。另外，我们将（Edited）追加到标题栏（见图6.4）。要添加此功能，我们将利用本章前面开始构建的抽象，并添加功能以检测文件是否已被修改。我们可以采取一些方法。

检查文件是否已更改的天真且有缺陷的方法是在UI中侦听keyup或change事件。如果用户添加了一个字符然后将其删除，则此方法仍会将文件视为已修改，这与其他本机桌面应用程序的行为方式不一致。

要确定文件是否已被修改，我们需要两条信息：文件的原始内容和当前内容。我们非常狡猾地将原始内容存储在清单6.2中。如果这两条信息相同，则文件没有改变。但如果它们有所不同，即使是略有不同，那么我们知道我们手上有一个修改过的文件。

要实现此功能，请修改updateUserInterface( )以获取名为isEdited的参数。在keyup上，我们将textarea的当前值与originalContent进行比较，并使用结果调用updateUserInterface( )。BrowserWindow实例有一个setDocumentEdited( )方法，它接受一个布尔值。这将巧妙地修改macOS上的窗口; 对于Windows和Linux用户，我们将（Edited）追加到窗口标题。



**图6.4 修改文件时的Fire Sale UI。请注意，标题栏内容已附加（Edited），并且不再禁用Save File和 Revert按钮。**

**清单6.5 如果文档有未保存的更改，则更新UI：./app/renderer.js**

传递一个布尔值，表示文档是否有

未保存的更改

const updateUserInterface = (**isEdited**) => {

let title = 'Fire Sale';

if (filePath) { title = `${path.basename(filePath)} - ${title}`; } **if (isEdited) { title = `${title} (Edited)`; }**

currentWindow.setTitle(title); **currentWindow.setDocumentEdited(isEdited);**

};

**如果isEdited为true，则相应地更新窗口**

最后一步是让每次用户在键入时从键中抬起手指时，渲染器进程都会调用updateUserInterface( )方法。

**清单6.6 每当用户输入时检查更改：./app/renderer.js**

markdownView.addEventListener('keyup', (event) => {

const currentContent = event.target.value; renderMarkdownToHtml(currentContent);

**updateUserInterface(currentContent !== originalContent);**

});

**每当用户在Markdown视图中输入击键时，检查当前内容是否与我们存储在变量中的内容相匹配，并相应地更新UI。**

###### *在UI中启用Save和Revert按钮*

有了这些步骤，您的应用程序就可以判断它是否处于已编辑和未保存的状态。但是我们遇到了问题。Save File和Revert按钮仍处于禁用状态。仅当存在未保存的更改时，才应启用这些按钮。当我们更新窗口本身时，很容易处理这个问题。

**清单6.7 当存在未保存的更改时启用Save和Revert按钮：./app/renderer.js**

const updateUserInterface = (isEdited) => {

let title = 'Fire Sale';

if (filePath) { title = `${path.basename(filePath)} - ${title}`; } if (isEdited) { title = `${title} (Edited)`; }

currentWindow.setTitle(title); currentWindow.setDocumentEdited(isEdited);

**saveMarkdownButton.disabled = !isEdited; revertButton.disabled = !isEdited;**

};

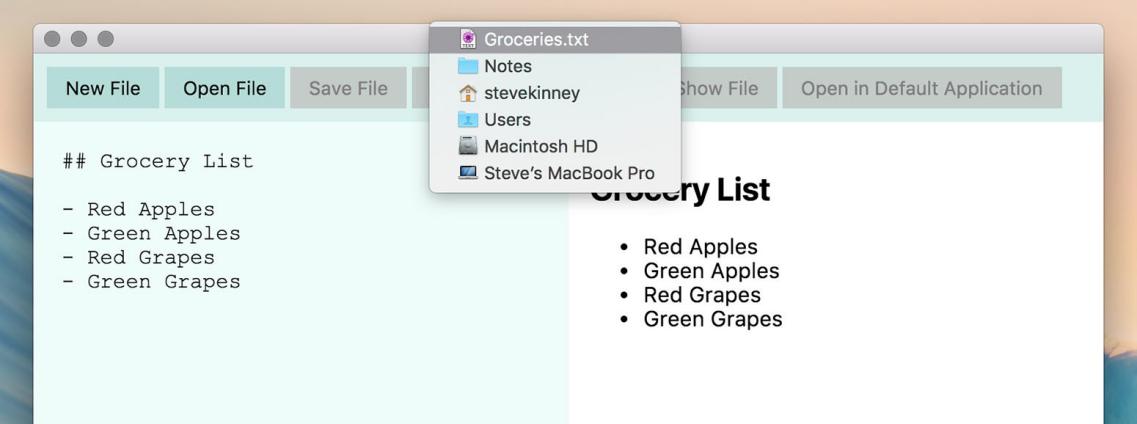
**如果文档未经编辑，则禁用Save按钮**

如果文档没有未保存的更改，请禁用还原未保存更改的按钮

我们将在本章后面实现这些按钮的功能。

###### *在macOS上更新代表的文件*

macOS窗口支持窗口菜单栏中当前文件的小表示。按住并按下Command键，同时单击文件图标以触发下拉菜单，显示文件系统层次结构中文件的位置，如图6.5所示。您也可以单击并拖动图标,就像从Finder中拖动文件一样。所有BrowserWindow实例都有一个名为setRepresentedFilename( )的方法，它接受一个有效的文件路径作为参数。此方法在Windows中无效。让我们将这个特性添加到我们的updateWindowTitle( )方法中。然后我们检查是否有一个有效的路径属性,如果是这样设置它为macOS窗口的代表文件。



**图6.5 单击标题栏中的文件图标时按住Command可以让我们看到它在文件系统上的位置。我们也可以拖放图标，就好像它是文件本身一样。**

这与使用附加信息更新窗口标题不同，例如文件是否已被编辑。在用户在同一窗口中打开另一个文件之前，所表示的文件保持不变。我们不需要在keyup上更新此值。我们有两个选择：我们可以在将路径和内容发送到渲染器进程之前在主进程中设置代表文件，或者我们可以在收到文件后在渲染器进程中使用当前窗口引用。我会选择前者，但两种方法都可以接受。

**清单6.8 在macOS中设置代表文件：./ app /main.js**

const openFile = exports.openFile = (targetWindow, file) => {

const content = fs.readFileSync(file).toString(); **targetWindow.setRepresentedFilename(file);** targetWindow.webContents.send('file-opened', file, content);

};

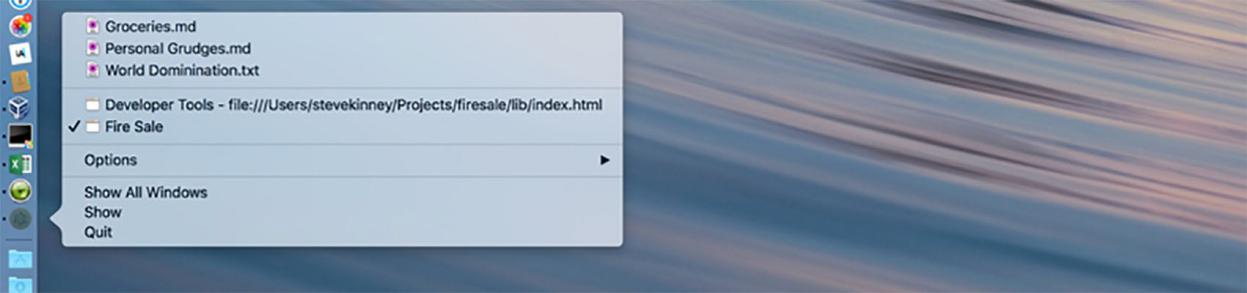
**BrowserWindow实例有一个方法，允许您设置代表的文件。**

### *跟踪最近打开的文件*

现在Mildred已经回到她的会议中了，她想回去工作。这是一次很长的会议，她不太记得她最近在做什么笔记。我们目前的实施也没有办法帮助她。但是，操作系统始终执行此操作。每当用户打开文件时，让Electron通知操作系统它应该将文件添加到最近打开的文件列表中。

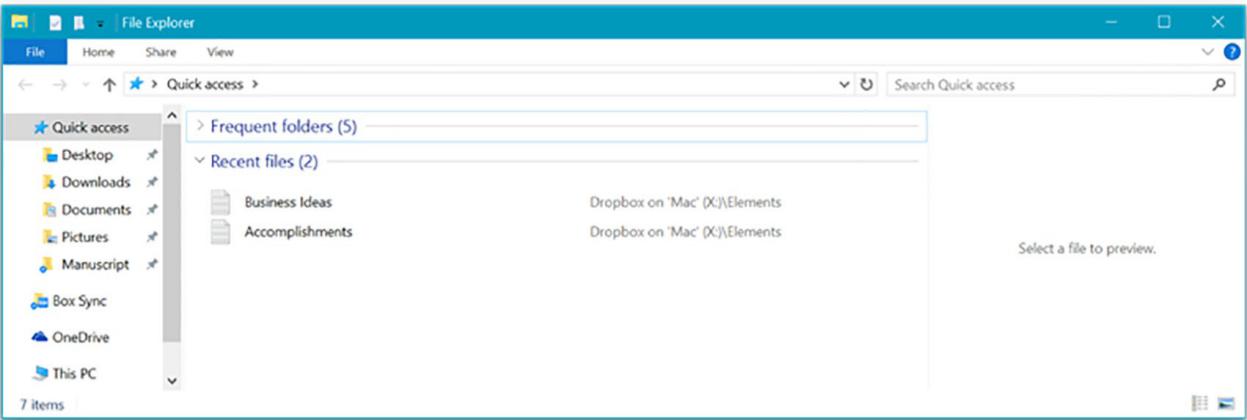
在macOS或Windows中打开文件时，Electron可以将文件路径添加到操作系统最近打开的文档列表中。通过右键单击macOS中的Dock图标（图6.6）或Windows中的任务栏图标（图6.7），可以使用此列表。

操作系统跟踪每个应用程序打开的文件。它还提供了最近打开的文件的主列表。在Windows中，此列表位于文件资源管理器中。在macOS中，您可以在Apple菜单中找到最近项目列表（图6.8）。从最近文档的全局列表中选择文档时，将在该类型文档的默认应用程序中打开该文档。在本章中，我们主要关注特定于我们应用程序的最新文档列表。

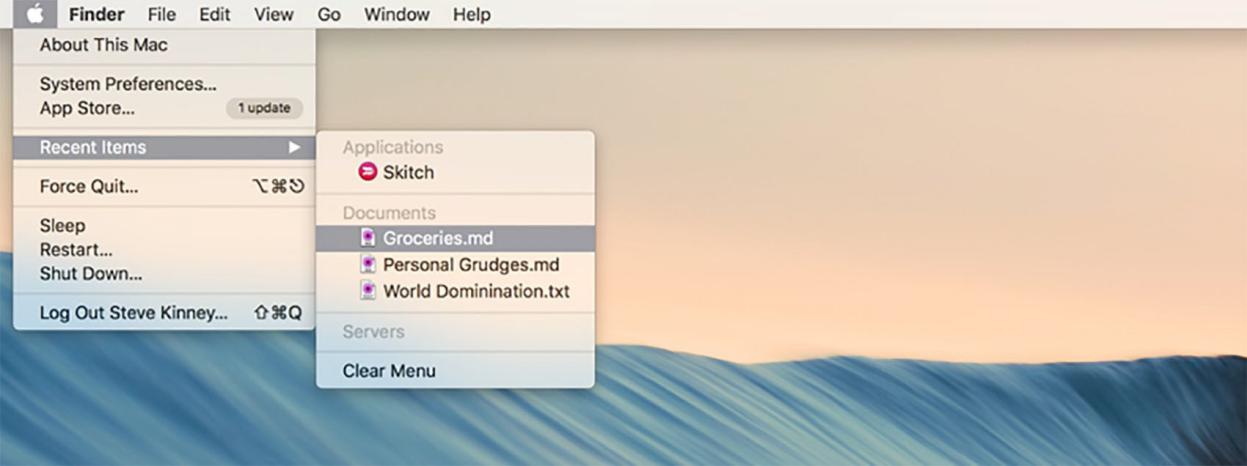


**图6.6 macOS中的最新文件**

***跟踪最近打开的文件* 107**



**图6.7 Windows 10中的最近文件**



**图6.8 macOS中系统范围的最近文档**

您可以使用app.addRecentDocument( )将文件路径添加到Electron中的最近文档列表，并将文件路径作为参数提供。在下一个清单中，我们将使用openFile( )打开的文件路径添加到最近的文档列表中。

**清单6.9 附加到最近文档列表：./app/main.js**

const openFile = exports.openFile = (targetWindow, file) => {

const content = fs.readFileSync(file).toString(); **app.addRecentDocument(file);** targetWindow.setRepresentedFilename(file); targetWindow.webContents.send('file-opened', file, content);

};

**Electron的app模块提供了一种附加到操作系统最近打开的文档列表的方法。**

如果您在Fire Sale中打开几个文件，您会看到它们被附加到最近文件列表中。但是，如果您尝试选择此列表中的任何条目，则不会发生任何事情。操作系统要求Fire Sale打开文件，但我们的应用程序还不知道如何操作。我们必须自己实现这个功能。

在第4章中，我们添加了从UI触发 Open File对话框的功能。当用户使用对话框选择文件时，文件路径将传递给打开的文件，该文件读取文件的内容并将这些内容发送到渲染器进程，以便在左窗格中显示，最后触发使用生成的HTML更新右窗格。当时，我们简要地讨论了用户可能以其他方式打开文件，这就是为什么将触发对话框的过程与打开文件的过程分开是有意义的。

从Recent Documents菜单中选择一个项目是用户想要打开文件而不通过“打开文件”对话框提示的一个实例。每当从应用程序外部打开文件时，Electron的应用程序模块都会触发一个打开文件的事件。在我们的代码中，我们可以监听open-file事件并相应地处理它。但是，我们应该等到应用程序完全启动并运行，因此一旦应用程序在app/main.js中触发它的will-finish-launch事件，我们就会设置我们的监听器。

**清单6.10 响应打开文件的外部请求：./app/main.js**

app.on('will-finish-launching', () => {

app.on('open-file', (event, file) => {

const win = createWindow(); win.once('ready-to-show', () => {

openFile(win, file);

});

});

});

**监听open-file事件，该事件提供外部打开文件的路径，然后将该文件路径传递给openFile()函数。**

现在，当用户在Fire Sale的最近文档列表中选择一个文件时，应用程序会创建一个新窗口并在新窗口中打开文件路径,就像您在第4章使用我们实现的UI中的Open File 按钮选择它一样。

### *保存文件*

保存文件与Electron类似于打开文件，但有一点不同：用户可能希望保存对Markdown文件的更改，或者他们可能希望导出由其应用程序生成的HTML。

如果用户第一次保存新文件，则应用程序应询问用户想要保存文件的位置以及他们想要提供的名称。之后，它应该跟踪该名称并更新窗口标题，如果它最初是从文件系统打开文件。如果用户正在保存对现有标记文件的更改，则应用程序无需提示指定文件的位置和文件名。实现保存文件的能力不仅仅是将内容写入文件系统。我们还必须更新UI以显示当前文件的保存位置以及自上次将文件保存到文件系统后是否已被修改。

在Fire Sale案例中，保存HTML输出更简单，因为应用程序不允许用户在保存后编辑HTML输出。导出生成的HTML与第一次保存文件非常相似，但我们不需要跟踪保存文件的位置或在UI中反映其新位置。这是三个中最容易实现的，所以让我们首先处理这个功能。

###### *导出呈现的HTML输出*

为了允许导出生成的HTML，我们向主进程添加了一个saveHtml( )函数，询问用户他们想要保存HTML文件的位置，从HTML视图中获取内容，然后将文件写入文件系统。

正如您所料，触发保存文件的本机对话框类似于触发打开文件。最大的区别是，我们不会提示用户选择要打开的特定文件，而是要求用户输入要写入文件系统的文件名和位置。该文件的内容作为参数传递给showSaveFileDialog( )函数。

在app/main.js中，添加清单6.11中显示的函数。运行此功能时，Electron会出现一个对话框，要求用户选择应写入内容的文件路径。一旦用户选择了文件路径，我们就使用Node的内置fs模块将文件内容写入文件系统。

**清单6.11 保存生成的输出：./app/main.js**

const saveHtml = exports.saveHtml = (targetWindow, content) => {

const file = dialog.showSaveDialog(targetWindow, { title: 'Save HTML',

defaultPath: app.getPath('documents'), filters: [

{ name: 'HTML Files', extensions: ['html', 'htm'] }

**默认为操作系统定义的用户“**documents**”目录**

]

});

if (!file) return;

**如果用户在文件对话框中选择“取消”，则中止该功能。**

fs.writeFileSync(file, content);

};

在示例中，dialog.showSaveDialog( )接受两个参数。第一个是对BrowserWindow的引用，它用于仅在macOS中将对话框显示为工作表。第二个参数是一个选项对象，允许您传递键和值以配置对话框本身。提供给dialog.showSaveDialog( )的对象使用以下选项：

* title：设置对话框的标题。这不会出现在macOS中。
* defaultPath：设置Save对话框的默认目录。
* buttonLabel：允许您为Save按钮设置自定义文本。
* filters：设置允许选择覆盖的文件。如果用户未提供默认文件扩展名，Electron也会使用此选项设置默认文件扩展名。

###### *常用路径*

我们正在实现保存文件的功能，但是我们应该在哪里提示用户保存这些文件以及它们如何根据他们使用的操作系统而有所不同？ Windows，macOS和Linux以不同方式组织文件。理想情况下，跨平台的Electron应用程序应默认在每个平台上显示正确的目录。Electron提供了app.getPath( )，它可以根据用户的平台自动返回正确的文件路径，从而使开发人员不必编写容易出错的条件逻辑。在清单6.11中，我们设置了app.getPath（'documents'）的默认路径，它将是Windows上的My Documents和macOS上用户主目录中的Documents文件夹。Electron提供以下附加路径：

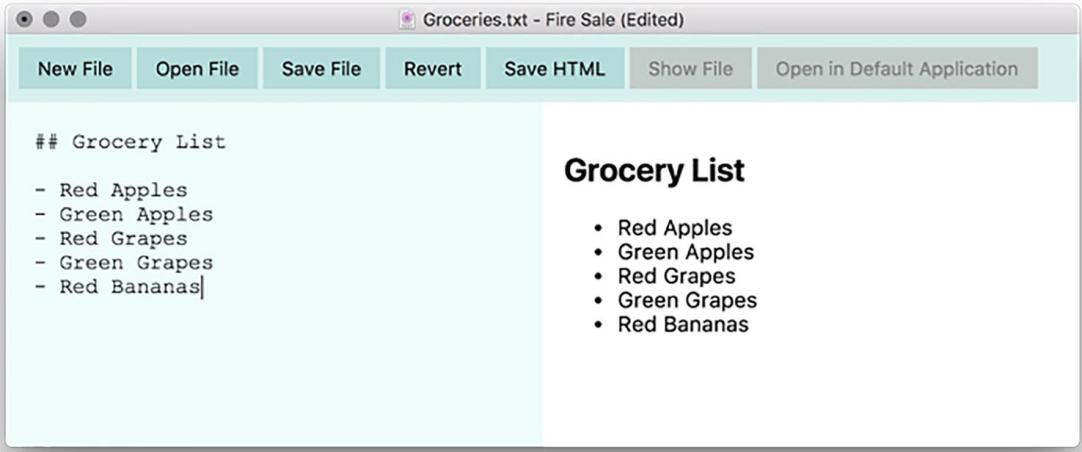
* + - * home 解析到用户的主目录。
      * desktop, documents, downloads, pictures, music,和videos均解析为用户主目录中的相应路径。
      * temp 解析为操作系统的临时文件目录。
      * exe 解析为当前可执行文件的位置。
      * appData 解析为用户的应用程序数据目录。这将是Windows上的％APPDATA％，macOS上的〜/Library/Application/Support，以及Linux上的$XDG\_CONFIG\_HOME或〜/.config。
      * userData 使用附加的应用程序名称解析为appData。例如，在macOS上，userData将在本章中解析为应用程序的〜/Library/Application/Support/fire sale。此名称来自package.json中的名称条目。

您可能希望应用程序覆盖app.getPath( )提供的默认值之一。您可以使用app.setPath( )执行此操作，它带有两个参数：上一个列表中的名称以及您希望它解析的新路径。请务必注意，您只能覆盖上一个列表中的路径。如果您要覆盖其中一个路径，则必须在应用程序触发其“ready”事件之前执行此操作。

我们没有在前一章的showOpenFile( )函数中实现默认路径，但这也是使用这种方法的一个很好的候选者。音乐播放器默认为用户通常存储其音乐的目录，或者照片管理应用程序默认为app.getPath（'pictures'）是有意义的。

###### *从渲染器进程保存文件*

现在我们启用Fire Sale应用程序来保存呈现的HTML内容，如图6.9所示。为了专注于学习Electron的基础知识，我们只需获取右侧窗格的HTML内容并将其传递给showSaveFileDialog( )。更强大的方法是添加doctype以及<html>，<head>和<body>标记，以使其成为有效的HTML文档。此外，我们可以添加有关文档的元数据和默认样式表，但这超出了本书的范围。在下一章中，我们还添加了从应用程序菜单中触发此功能的功能。



**图6.9 Fire Sale允许用户在左窗格中保存其Markdown内容以及右窗格中显示的呈现HTML输出。**

**清单6.12 从渲染器进程触发Save File对话框：. /app/renderer.js**

saveHtmlButton.addEventListener('click', () => {

mainProcess.saveHtml(currentWindow, htmlView.innerHTML);

});

###### *保存当前文件*

保存当前文件就像保存HTML输出一样，只有一个小区别：如果文件是从文件系统打开的，那么应用程序不需要提示用户输入文件路径。相反，应用程序只使用filePath作为应写入文件的位置。

**Listing 6.13 保存当前文件: ./app/main.js**

const saveMarkdown = exports.saveMarkdown = (targetWindow, file, content) => {

if (!file) {

file = dialog.showSaveDialog(targetWindow, { title: 'Save Markdown',

defaultPath: app.getPath('documents'), filters: [

**如果这是没有文件路径的新文件，则提示用户选择带有对话框的文件路径**

]

});

}

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

if (!file) return;

fs.writeFileSync(file, content); openFile(targetWindow, file);

};

**如果用户在“文件”对话框中选择“取消”，则中止该功能**

将缓冲区的内容写入文件系统

此代码足够灵活，可以处理我们保存新文件的情况，以及我们更新现有文件的情况。如果您回忆一下本章前面的内容，则在用户打开文件之前，filePath属性的值默认为null。如果filePath属性为false，则Electron会提示用户使用本机“保存文件”对话框选择文件路径。然后，它将用户选择的位置保存到filePath属性。

相反，如果它已经知道文件的位置，那么它会向前移动并直接跳过将内容写入文件系统。为了简洁起见，我在刚刚保存的文件上调用了openFile( )。如果它是我们第一次保存的新文件，那么我们希望将其添加到操作系统的最近文档列表中，并将其设置为代表文件。将这个问题抽象出来，我们可以调用它而不必再从磁盘读取文件，这是我留给读者的一个练习。

如果无法触发，则无法完成任何功能。应用程序需要在Save File按钮上触发我们的新功能的事件侦听器。

**清单6.14 在Save File按钮中添加一个事件监听器：./app/renderer.js**

saveMarkdownButton.addEventListener('click', () => {

mainProcess.saveMarkdown(currentWindow, filePath, markdownView.value);

});

###### *还原文件*

鉴于我们到目前为止构建应用程序的方式，添加功能很容易。当用户单击Revert按钮时，我们将Markdown视图的值替换为上次打开或保存时缓存的文件的原始内容，然后触发HTML视图以使用缓存的内容进行重新渲染。

**清单6.15 将UI中的内容恢复为上次保存的内容：./app/renderer.js**

revertButton.addEventListener('click', () => {

markdownView.value = originalContent; renderMarkdownToHtml(originalContent);

});

在本章后面，我们将提示用户确保他们想要在继续前进之前将所有更改都吹走。

### *使用拖放操作打开文件*

Electron应用程序支持HTML5文件API，它允许我们为应用程序创建一个功能，用户可以将文件拖到DOM中的某些元素上并删除它们。您已经看到此API用于在Twitter的Web应用程序中上传照片或将文件附加到Gmail中的Electron邮件中。在Fire Sale中，我们利用此API允许用户通过将文件拖到UI的Markdown视图来打开文件。

###### *忽略其他地方放开的文件*

将文件放入浏览器窗口时，Web浏览器的默认操作是在浏览器中打开该文件。我们可以将文件拖到我们的应用程序上，并在文件内容完全取代UI时恐怖地观看。如果您认为我们没有依赖的Back按钮，则会出现更多问题。

为我们的应用程序创建拖放功能的第一步是通过向文档本身添加一个阻止默认操作的事件监听器来禁用默认行为。稍后，我们将重新选择并在我们的UI中自定义Markdown窗格的行为。

**清单6.16 为拖放事件设置基础：./app/renderer.js**

document.addEventListener('dragstart', event => event.preventDefault()); document.addEventListener('dragover', event => event.preventDefault()); document.addEventListener('dragleave', event => event.preventDefault()); document.addEventListener('drop', event => event.preventDefault());

###### *提供视觉反馈*

虽然没有必要自己实现该功能，但为用户提供可视指示以便将文件拖到应用程序中的某个区域通常很有帮助。我们定义了两个可以使用JavaScript添加和删除的CSS类，具体取决于被拖动的项是否有效。

**清单6.17 为拖放功能添加样式：./app/style.css**

.raw-markdown.drag-over {

background-color: rgb(181, 220, 216);

border-color: rgb(75, 160, 151);

}

.raw-markdown.drag-error {

background-color: rgba(170, 57, 57,1); border-color: rgba(255,170,170,1);

**这种深蓝绿色表示用户这是一个有效的放松目标.**

此红色表示用户将要放松的文件存在问题。

}

在我们开始实现此功能之前，如清单6.18所示，拥有一些帮助函数会很不错。我将创建两个可疑的类似函数：getDraggedFile( )和getDroppedFile( )。两者之间的一个重要区别：当用户拖动文件时，我们只能访问其元数据。只有在用户正式放开文件后，我们才能访问File对象。getDraggedFile( )将以事件对象的形式从DataTransferItem对象中选择文件的元数据，该对象具有大量其他属性，例如事件被触发时鼠标所在的位置等等。getDroppedFile( )从files数组中提取第一个元素，当用户只是拖动文件时，该元素为空。

这个过程可能看起来很艰巨，但都是以安全为名。您可能会通过窗口，这些窗口不应该知道您尝试在前往预期应用程序的路上拖放的文件，因为该文件很可能包含敏感信息。但是，一旦你拖放了文件，浏览器就会假定这个操作是有意的，并允许应用程序读取文件。fileTypeIsSupported( )检查要拖动的文件类型是否为Fire Sale支持的两种类型之一，并根据结果返回布尔值。

**清单6.18 Helper方法：./app/renderer.js**

这与getDraggedFile( )类似，但在用户放开文件后，我们可以访问文件本身，而不仅仅是元数据。

如果用户选择多个项目，这将始终是一个数组。该应用程序一次只支持一个文件。我们取数组中的第一项。

const getDraggedFile = (event) => event.dataTransfer.items[0]; const getDroppedFile = (event) => event.dataTransfer.files[0];

const fileTypeIsSupported = (file) => {

return ['text/plain', 'text/markdown'].includes(file.type);

}; 如果文件的类型在受支持的文件类型数组中，则此帮助程序函数返回true或false。

当用户通过浏览器窗口拖动文件时，它会快速触发dragover事件，直到用户离开目标区域,在这种情况下，一个dragleave事件,或者用户从鼠标或触控板上抬起手指并将文件放到 目标区域，触发drop事件。

在dragover阶段，我们可以给用户一个关于拖放是否成功的直观线索，如图6.10所示。如果用户正在拖动我们不支持的文件类型，我们可以将.drag-error类添加到元素中（参见图6.11）。否则，我们将添加.drag-over类以指示用户可以在此处放开文件。当用户从目标区域中删除文件时，我们将清理所添加的任何类并将UI恢复到其默认状态。

**清单6.19 在dragover和dragleave上添加和删除类：./app/renderer.js**

markdownView.addEventListener('dragover', (event) => { const file = getDraggedFile(event);

if (fileTypeIsSupported(file)) {

markdownView.classList.add('drag-over');

} else {

**如果支持文件类型，则添加CSS类以指示这是放置文件的有效位置。**

markdownView.classList.add('drag-error');

}

});

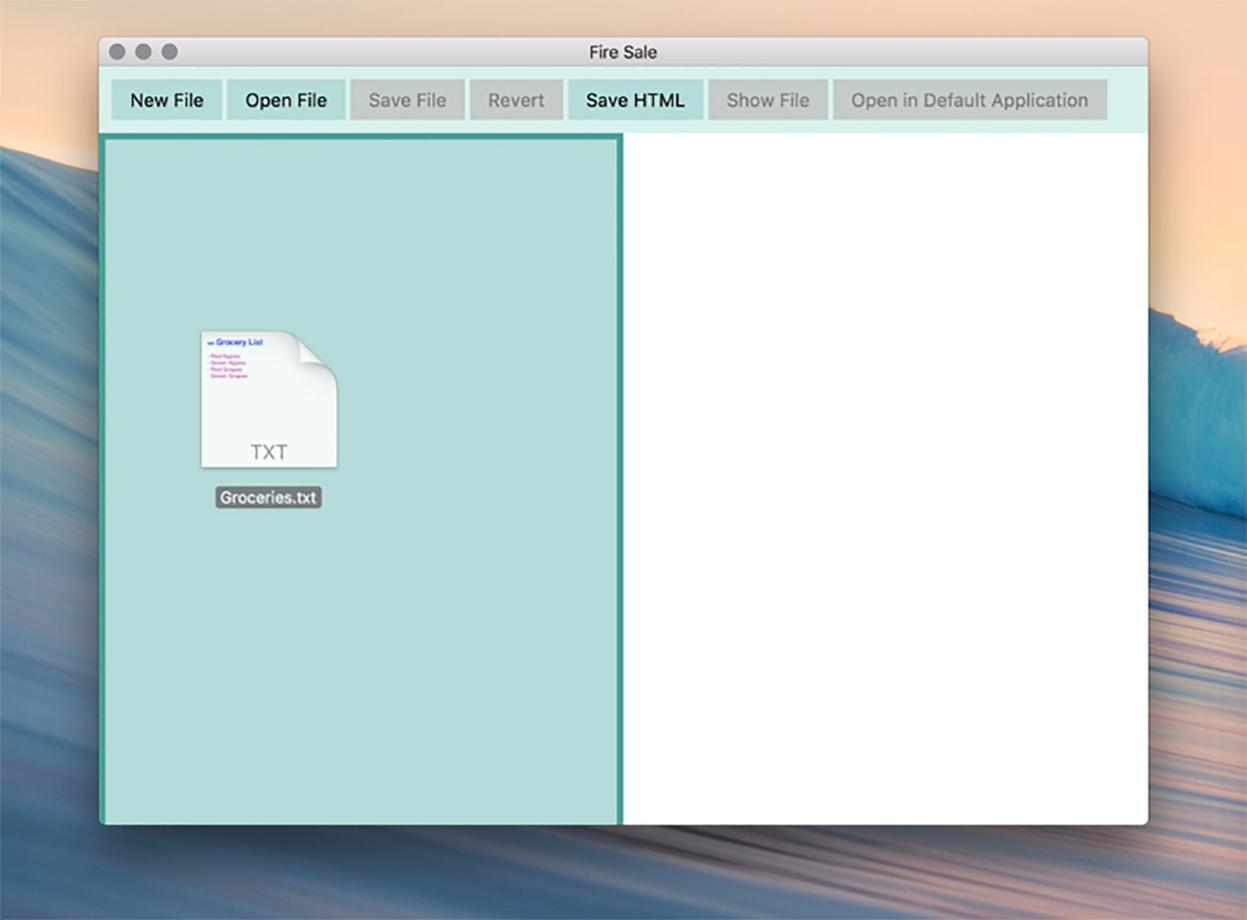
markdownView.addEventListener('dragleave', () => {

markdownView.classList.remove('drag-over'); markdownView.classList.remove('drag-error');

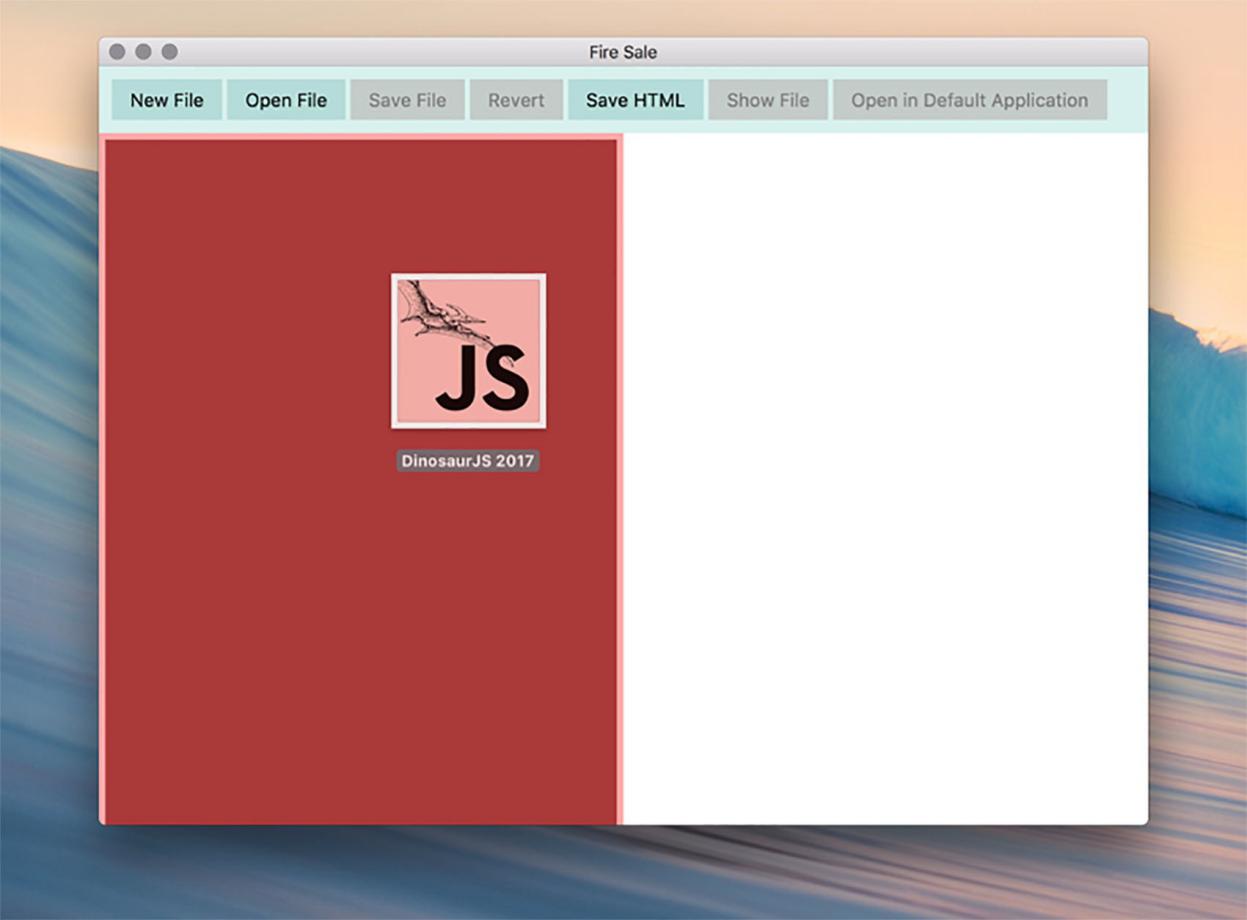
});

**如果不支持文件类型，则添加CSS类以指示尽管这是放置文件的有效位置，但不接受此文件。**

**如果用户从Markdown视图中获取文件，请取消我们之前添加的类。**



**图6.10 向Markdown视图添加CSS类为用户提供了一个可视提示，即这是放置此文件的有效位置。**



**图6.11 同样，Fire Sale不支持图像。我们即将编写的代码将拒绝任何不属于我们支持的类型的文件，但在视觉上向用户显示该文件将不被接受允许他们提前取消其操作。**

###### *打开拖放的文件*

当用户成功将文件拖放到左窗格时，清单6.20中的代码再次确认Fire Sale支持此类文件。如果是，我们将它传递给我们之前创建的主进程中的activeFile对象，就像我们从对话框或最近的文档列表中选择它一样。

**清单6.20 拖放功能：./app/renderer.js**

markdownView.addEventListener('drop', (event) => {

const file = getDroppedFile(event);

if (fileTypeIsSupported(file)) {

mainProcess.openFile(currentWindow, file.path);

} else {

alert('That file type is not supported');

}

**如果支持文件类型，则渲染器进程与主进程通信。**

如果不支持文件类型，则应用程序会警告用户。

***监视文件更改* 117**

markdownView.classList.remove('drag-over'); markdownView.classList.remove('drag-error');

});

由于新代码非常少，我们已经成功实现了拖放功能，让人联想到原生桌面应用程序。

### *监视文件更改*

我们的应用程序现在存在一个潜在危险的边缘情况。如果我们在其他编辑器中打开Markdown文件并对其进行更改，那么Fire Sale就完全没有意识到这一点。这意味着如果我们在其他编辑器中进行更改后将文件保存在Fire Sale中，它将破坏性地覆盖这些更改。

我们有几种解决方法。例如，当Fire Sale重新成为焦点时，我们可以重新读取文件。我们可以定期检查文件并查看其内容是否已更改。我们现在采用的方法是利用Node的fs.watchFile功能，该功能使用特定于操作系统的库来监视文件或目录，并在文件发生更改时发出事件。

这种方法的一个警告是，当我们打开新文件时，我们必须小心停止观看文件。否则，我们不断添加观察者，这是一个内存泄漏。由于我们有多个窗口这使事情变得更加复杂。如果文件发生变化，我们必须确保将其发送到正确的窗口。我们的功能将如下工作：

**1.设置数据结构以跟踪文件监视器及其关联的窗口。**

**2.打开文件时开始观察文件更改。**

**3.打开后续文件时，在创建新观察者之前关闭现有观察者。**

**4.窗口关闭时关闭观察者。**

我们的第一个任务是找出一种方法来管理窗口，当前显示在窗口中的文件和/或文件的观察者之间的关系。在第5章中，我们使用Set来跟踪应用程序中当前打开的所有窗口。在本节中，我们将使用JavaScript语言的另一个新增功能--Map。

Map是键值存储，与JavaScript中的常规对象非常相似，具有重要的区别。对象只能包含字符串和数字作为键。Map可以使用任何类型的对象或值作为键。为了实现此功能，我们实例化一个Map，它使用BrowserWindow实例作为键，文件观察者使用值作为值。当用户关闭窗口时，我们找到与该窗口关联的观察者并将其停止。

**清单6.21 设置Map来监视文件：./app/main.js**

const openFiles = new Map();

正如我之前提到的，我们希望在打开时开始观察文件路径，并在窗口关闭或用户在窗口中打开不同文件时停止观看。让我们设置两个函数：startWatchingFile( )和stopWatchingFile( )。

**清单6.22 设置一个监听器：./app/main.js**

const startWatchingFile = (targetWindow, file) => {

stopWatchingFile(targetWindow);

const watcher = fs.watchFile(file, (~~event~~) => {

~~if (event === 'change') {~~

const content = fs.readFileSync(file);

**如果有，则关闭现有的观察者.**

如果观察者触发更改事件，则重新读取该文件**.**

~~}~~

});

targetWindow.webContents.send('file-opened', file, content);

使用文件内容向渲染器进程发送消息**.**

openFiles.set(targetWindow, watcher);

}; 

const stopWatchingFile = (targetWindow) => {

if (openFiles.has(targetWindow)) {

openFiles.get(targetWindow).stop(); openFiles.delete(targetWindow);

**跟踪观察者，以便我们以后可以停止它.**

**检查我们是否有为此窗口运行的观察程序.**

}

}; **从打开的窗口的Map中删除观察者。**

**停止观察者。**

译注:

const openFile = exports.openFile = (targetWindow, file) => {

const content = fs.readFileSync(file).toString();

app.addRecentDocument(file);

targetWindow.setRepresentedFilename(file);

targetWindow.webContents.send('file-opened', file, content);

**startWatchingFile(targetWindow, file);**

};

在清单中，我们开始观察给定的文件路径。如果观察者发出change事件，我们会向窗口发送一条消息，提醒它该文件已更改。最后，将窗口和相关的观察者添加到openFiles Map，这样我们可以在关闭观察者时找到它。作为一项额外的预防措施，让我们在创建一个新窗口之前关闭该窗口的任何现有观察者。如果用户打开已在观看文件的文件窗口，这样做会很有帮助。

**清单6.23 关闭浏览器窗口时关闭观察者：./app/main.js**

const createWindow = exports.createWindow = () => {

let x, y;

const currentWindow = BrowserWindow.getFocusedWindow(); if (currentWindow) {

const [ currentWindowX, currentWindowY ] = currentWindow.getPosition();

x = currentWindowX + 10; y = currentWindowY + 10;

}

let newWindow = new BrowserWindow({ x, y, show: false }); newWindow.loadFile('index.html');

newWindow.once('ready-to-show', () => { newWindow.show();

});

newWindow.on('close', (event) => {

if (newWindow.isDocumentEdited()) {

// …

}

});

newWindow.on('closed', () => {

windows.delete(newWindow); **stopWatchingFile(newWindow);** newWindow = null;

});

**关闭窗口时，停止观察者查找与该窗口关联的文件。**

windows.add(newWindow); return newWindow;

};

### *在放弃更改之前提示用户*

现在，窗口的标题反映了用户是否对当前文档进行了尚未保存的更改。但是如果用户试图关闭窗口怎么办？ 变化消失了，无法恢复。在上一节中，我们添加了查看当前活动文件路径的功能。如果文件被其他应用程序更改，Fire Sale会在不发出警告的情况下覆盖更改。在这两种情况下，这对于桌面应用程序来说都是不可接受的行为。如果用户即将失去工作，则会提示他们。在本节中，我们将实施这些保护措施。

在第5章中，我们为closed事件设置了一个监听器，当窗口成功关闭时触发该事件。Electron还支持close事件，当用户尝试关闭窗口时会触发该事件。如果用户有未保存的更改，我们可以进行干预并提示用户确认他们实际上是要关闭窗口并丢失更改。

**清单6.24 如果用户尝试关闭未保存更改的窗口，则提示用户：./app/main.js**

检查文档是否已被编辑。我们通过将当前内容与原始内容进行比较，在Markdown视图中的每个键盘上的渲染器过程中设置此项。

newWindow.on('close', (event) => {

if (newWindow.isDocumentEdited()) {

event.preventDefault();

**如果窗口有未保存的更改，则阻止其关闭。**

提示用户使用自定义消息框询问他们是否确定要关闭窗口并丢失其更改。将他们的选择保存到result中。

const result = dialog.showMessageBox(newWindow, {

type: 'warning',

title: 'Quit with Unsaved Changes?',

message: 'Your changes will be lost if you do not save.', buttons: [

'Quit Anyway',

**提供一个按钮标签列表.**

'Cancel',

],

**如果用户点击Return键，则将第一个选项设置为默认选项。**

defaultId: 0,

cancelId: 1

});

如果用户关闭消息框，则将第二个按钮设置为所选按钮。

}

});

if (result === 0) newWindow.destroy();

如果用户选择“Quit Anyway”，则强制窗口关闭。

译注:

newWindow.isDocumentEdited() 这行代码经测试仅在macOS下有效,所以本节只会在macOS下起作用.更多信息参见:[https://electronjs.org/docs/api/browser-window#winisdocumentedited-macos](https://electronjs.org/docs/api/browser-window" \l "winisdocumentedited-macos)

在前面的章节中，我们使用dialog.showOpenDialog( )和dialog.showSaveDialog( )来提示用户选择文件。dialog.showMessageBox( )是一个通用的，可自定义的对话框。您可以为按钮数组提供按钮标签列表。dialog.showMessageBox( )返回用户选择的按钮的索引。如果用户选择第一个按钮，则dialog.showMessageBox( )返回0.我们可以使用返回值根据用户的偏好确定如何在我们的应用程序中继续。dialog.showMessageBox( )还采用了其他选项，允许我们指定用户单击Return按钮时应该执行的默认操作，以及如果用户取消对话框应返回的选项。

在前面的示例中，我们监听close事件。如果窗口有未经编辑的更改，我们会提示用户。根据用户的响应，我们要么阻止窗口关闭，要么故意破坏窗口。

我们还需要防范用户可能丢失更改的其他两种情况。第一种是当用户尝试在未保存的更改时在同一窗口中打开另一个文件时;第二个是如果另一个应用程序更改文件。这两个功能之间的主要区别在于向用户显示的消息。无论哪种方式，如果用户决定继续前进，那么我们加载文件并替换UI中的内容。为了避免重复我们自己，我们将把这个过程转移到它自己的函数中，以便在两个地方使用它。

**清单6.25 重构显示新文件的进程：./ app/renderer.js**

const renderFile = (file, content) => {

filePath = file;

originalContent = content;

markdownView.value = content; renderMarkdownToHtml(content);

updateUserInterface(false);

};

使用我们新的renderFile( )函数，我们可以设置两个IPC监听器。当用户打开一个新文件（列出6.26）时，我们继续使用file-opened的通道。但是，如果该文件已被其他应用程序修改（列出6.27），我们将通过 file-changed的通道发送消息。根据我们收到消息的通道，我们向用户显示不同的消息。

**清单6.26 如果有未保存的更改，则在打开新文件时提示用户：./app/renderer.js**

ipcRenderer.on('file-opened', (event, file, content) => { if (currentWindow.isDocumentEdited()) {

const result = remote.dialog.showMessageBox(currentWindow, { type: 'warning',

title: 'Overwrite Current Unsaved Changes?',

message: 'Opening a new file in this window will overwrite your unsaved

changes. Open this file anyway?',

buttons: [ 'Yes',

'Cancel',

],

defaultId: 0,

cancelId: 1

});

if (result === 1) { return; }

}

**使用远程模块从主进程触发对话框。**

如果用户取消，则提前从该功能返回。

renderFile(file, content);

});

**将窗口设置为未编辑状态，因为用户刚刚打开了一个新文件。**

**清单6.27 文件更改时提示用户：./ app/renderer.js**

ipcRenderer.on('file-changed', (event, file, content) => { const result = remote.dialog.showMessageBox(currentWindow, {

type: 'warning',

title: 'Overwrite Current Unsaved Changes?',

message: 'Another application has changed this file. Load changes?', buttons: [

'Yes',

'Cancel',

],

defaultId: 0,

cancelId: 1

});

renderFile(file, content);

});

**在这种情况下，我们不关心文档是否已被编辑。我们想要提示用户。**

最后一步是修改startWatchingFile( )函数，通过 file-changed的通道而不是file-opened的通道发送消息，以触发正确的消息框。

**清单6.28 通过file-changed的通道发送消息：./ app/main.js**

const startWatchingFile = (targetWindow, file) => {

stopWatchingFile(targetWindow);

const watcher = fs.watch(file, (event) => {

if (event === 'change') {

}

});

const content = fs.readFileSync(file).toString(); **targetWindow.webContents.send('file-changed', file, content);**

如果当前文件发生更改，则触发不同的事件

openFiles.set(targetWindow, watcher);

};

有了它，我们的应用程序现在支持拖放功能，监视文件系统的更改，将文件添加到操作系统的最近打开的文件列表，更新窗口的标题栏，在macOS上设置代表文件，并提醒用户在丢弃未保存的更改之前。可以在https://github.com/electron-in-action/firesale/tree/chapter-6或附录中找到该应用程序的代码。

### *本章小结*

* + - 在实现保存文件的能力时，我们必须考虑这是一个新文件还是现有文件，并以不同方式处理每个场景。
    - 保存新文件时，我们可以使用dialog.showSaveFileDialog( )来提示用户选择要写入文件的位置。
    - 保存现有文件时，Fire Sale会写入现有文件的当前位置。
    - 默认情况下，Electron窗口显示HTML文档的<title>标记的内容。所有BrowserWindow实例上的setTitle( )方法都允许用户根据应用程序的状态更新和自定义窗口的标题。
    - Electron提供了在macOS中进一步自定义窗口的功能。

- 我们可以将“代表文件”设置为给定路径，并将其添加到菜单栏中。macOS允许用户拖动文件，就好像它们是从Finder中拖动它一样。

- 我们可以在BrowserWindow实例上使用setDocumentedEdited( )方法在窗口的关闭按钮中显示一个小点，表示用户他们有未保存的更改。

* + - Electron提供了app.addRecentDocument( )方法，该方法将给定的文件路径附加到操作系统最近打开的文档列表中。这适用于所有受支持的平台。
    - 当用户从操作系统的最近打开的文档列表中选择一个文件时，Electron默认不知道如何处理这个文件。我们必须在处理文件打开事件的app对象上提供自定义监听器。
    - Electron为用户通常想要保存文件的常见位置提供了几个快捷方式。这是在引擎盖下完成的，使我们免于为每个支持的操作系统定制默认位置的责任。
    - 除了Electron提供的文件选择对话框，我们还可以使用HTML File API来支持用户的拖放操作。
    - Node提供了fs.watch( )方法，它允许我们监视当前打开的文件，并在其他应用程序更改后提醒我们。

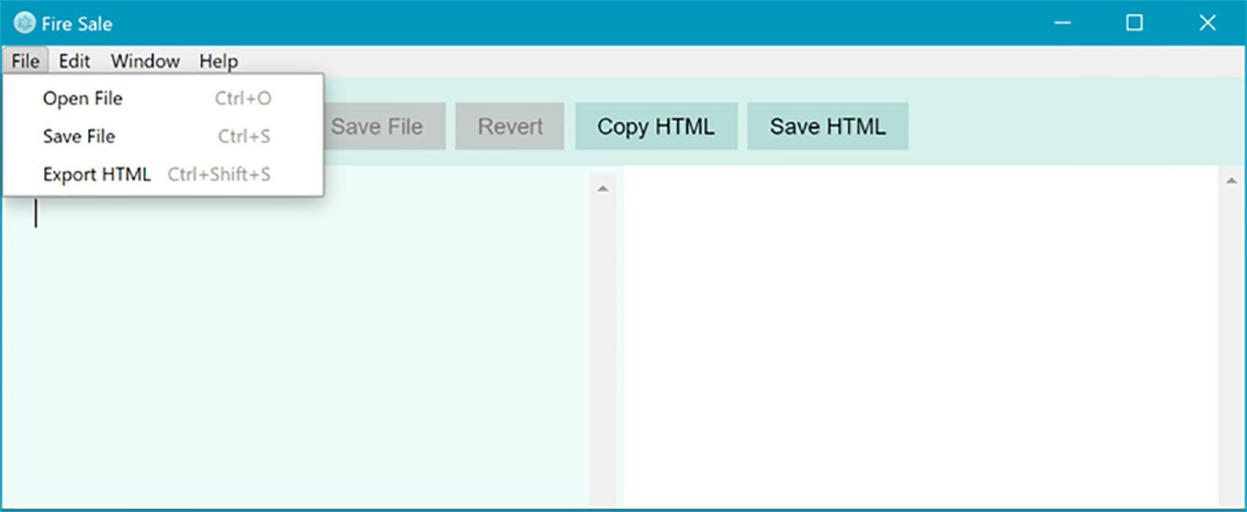
*构建应用程序和上下文菜单*

***本章包括***

* 使用Electron的Menu和MenuItem模块创建菜单
* 从模板构建菜单
* 为目标操作系统定义自定义菜单
* 将常见的操作系统角色分配给我们的菜单项
* 使用自定义的应用程序特定功能制作菜单项
* 为UI的不同部分创建自定义上下文菜单

在基于浏览器的应用程序中，开发人员只能访问应用程序窗口的可见区域。他们无法将控件添加到浏览器的工具栏或菜单栏中。应用程序功能的整个UI必须位于窗口内。开发人员也在窗口内面临限制。他们无法修改用户右键单击其UI时显示的上下文菜单。为每个选项和命令找到一个位置可能是一个挑战。另一方面，Electron使开发人员能够在浏览器窗口之外添加功能，例如当用户右键单击UI组件时出现的自定义应用程序和上下文菜单。

在本章中，我们将探讨如何在Fire Sale中创建和配置这些菜单。我们将用我们自己的默认菜单替换Electron提供的默认菜单，并在菜单中展示常见的操作系统功能。我们为菜单项分配键盘快捷键，以便在应用程序的任何位置轻松触发。通过实现基本菜单功能，我们可以添加我们自己的特定于应用程序的菜单项,特别是能够从文件系统打开Markdown文件，将其显示在UI的左侧窗格中，并将其内容呈现为HTML 在右侧窗格中。最后，只要用户右键单击左窗格，我们就会创建一个自定义上下文菜单，其中包含常见的文本操作任务（剪切，复制和粘贴，如图7.1所示）。



**图7.1 在本章中，我们构建了自定义菜单项，触发了UI中的一些功能。**

在前几章中，我们在Fire Sale中有一个菜单。那为什么现在要建立一个定制的呢？ 开发人员可以覆盖Electron的默认菜单，但是他们负责从头开始构建菜单。在本章的过程中，我们将恢复大多数桌面应用程序常见的一些基本功能。奠定基础后，我们使用自己的自定义功能扩展它。从我们的菜单中，用户可以保存当前活动的文件，也可以将HTML导出到自己的文件中。除了能够从应用程序菜单访问此功能外，用户还可以使用键盘快捷键来触发菜单项。在本章中，我们为Fire Sale构建了一个菜单，其结构如图7.2所示。

Quit

Show All

Hide Others

Hide

Services

About

Application (macOS only)

File

Save File

Export HTML

Open File

Select All

Paste

Copy

Cut

Redo

Undo

Edit

Close

Minimize

Window

Visit Website

Search (macOS only)

Help

**图7.2 Fire Sale的应用程序菜单的结构**

### *7.1 替换并复制默认菜单*

要开始，请创建一个名为./app/application-menu.js的新文件。这个文件将在本章结尾处变大，所以我们现在通过将其分解为自己的文件来解决这个问题。让我们首先将复制并粘贴回应用程序菜单。

**清单7.1 创建一个带有复制和粘贴的Edit菜单：./ app/application-menu.js**

const { app, BrowserWindow, Menu, shell } = require('electron');

const mainProcess = require('./main');

const template = [

{

label: 'Edit', submenu: [

**创建一个模板数组，该数组将用作菜单的蓝图**

**在本章中我们构建应用程序菜单时需要的模块**

{

label: 'Copy',

accelerator: 'CommandOrControl+C', role: 'copy',

},

{

label: 'Paste',

accelerator: 'CommandOrControl+V', role: 'paste',

},

]

}

];

**创建一个模板数组，该数组将用作菜单的蓝图--重复?**

菜单项可以被赋予称为加速器的键盘快捷键。

从模板构建菜单，并将其导出，以便可以在主进程中使用。

module.exports = Menu.buildFromTemplate(template);

接下来，当app触发ready事件时，将菜单设置为应用程序的菜单。

**清单7.2 在主应用程序文件中加载菜单：./app /main.js**

const { app, BrowserWindow, dialog, **Menu** } = require('electron'); **const applicationMenu = require('./application-menu');**

const fs = require('fs');

**需要上一个列表中内置的菜单**

**需要Electron的Menu模块**

const windows = new Set();

const openFiles = new Map();

app.on('ready', () => {

**Menu.setApplicationMenu(applicationMenu);** createWindow();

});

// … Additional methods below …

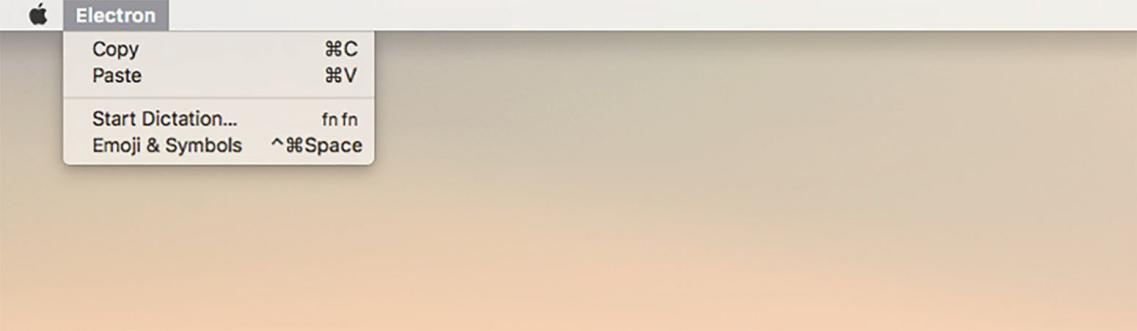
成功启动后将其设置为应用程序菜单

Electron包含用于构建菜单的Menu和MenuItem模块。从理论上讲，我们可以从单个MenuItems构建菜单，但这种方法可能很乏味且容易出错。为方便起见，Menu提供了接受常规JavaScript对象数组的buildFromTemplate( )方法。在内部，Electron根据您提供的数组创建MenuItems。

###### *macOS和缺少编辑菜单的情况*

如果在Windows中启动应用程序，则应看到包含两个菜单项的“编辑”菜单：Copy和Paste。这是可以预料的。但是如果你在macOS上测试应用程序，你会看到一些不同的东西，如图7.3所示。

在macOS中，菜单称为Electron而不是Edit，因为macOS上的第一个菜单始终是Application菜单。要在Electron中解决这个问题，我们需要将Edit菜单和所有后续菜单项移到未来的一个位置，如清单7.3和图7.4所示，为Application菜单腾出空间，我们稍后会在本章实现。



**图7.3 macOS获取第一个菜单项并将其用作应用程序菜单，这并不总是预期或预期的行为。**



**图7.4 通过将所有菜单向下移动一个位置，Edit菜单可以正确呈现。很快，我们实现了一个应用程序菜单，其行为类似于本机macOS应用程序。**

**清单7.3 在macOS中的菜单项列表之前：./app /application-menu.js**

const { app, BrowserWindow, Menu, shell } = require('electron'); const mainProcess = require('./main');

…

const template = [

// … Menu template from the last section.

];

**if (process.platform === 'darwin')** {

const name = 'Fire Sale'; template.unshift({ label: name });

}

**询问Node的进程全局运行应用程序的平台。macOS报告它是darwin。如果是这种情况，请将新菜单项移动到模板数组的开头。**

**获取应用程序的名称。这不会出现在菜单中，但后面会很有用。**

module.exports = Menu.buildFromTemplate(template);

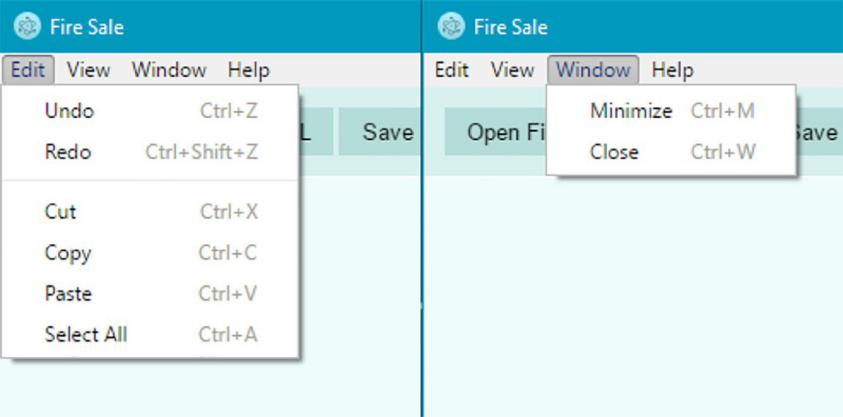
使用Electron构建应用程序的一个好处是，开发人员可以使用一个代码库来定位macOS，Windows和Linux。需要注意的是，开发人员在编写代码时应考虑每个支持的操作系统的特性。幸运的是，Node提供了进程(process)对象，该进程对象具有多个属性，方法和事件，可以为运行应用程序的环境提供默认。

process.platform返回应用程序当前正在执行的平台的名称。在撰写本文时，process.platform返回五个字符串之一：darwin，freebsd，linux，sunos或win32。Darwin是构建macOS的UNIX操作系统。我们可以通过检查process.platform是否等于darwin来在运行时调整我们的菜单。如果是，则应用程序在macOS上运行，所有菜单项应该向右移动一个位置。

对于以正确的顺序获取菜单所需的所有额外工作，您可能已经在图7.4中注意到我们获得了听写和表情符号支持，而无需通过编辑菜单实现它。

###### *替换Electron默认菜单的隐藏成本*

Electron提供了一个默认菜单，但这是一个全有或全无的事情。当我们更换菜单时，我们失去了所有原始功能。我们不仅丢失了一些菜单项，还丢失了键盘快捷键。尝试使用macOS上的Command-X键盘快捷键或Windows和Linux上的Control-X来从左侧窗格中剪切文本。那么分别在macOS或Windows上使用Command-A或Control-A来选择所有文本？ 如何撤消Command-Z或Control-Z？ 什么都没发生。如果您使用的是macOS，请尝试按Command-Q退出该应用程序。再一次，没有任何反应 我们也失去了在macOS中隐藏此应用程序和其他应用程序的功能。在所有操作系统上，我们都无法撤消和重做更改，最小化和关闭窗口，并选择每个字段中的所有文本。剩下的就是复制和粘贴的能力，如图7.5所示,这只是因为它被添加回我们的自定义菜单中。



**图7.5 在Electron的内置菜单中实现的编辑和窗口菜单**

由开发人员将这些功能添加回应用程序。如果我们想从我们的应用程序中省略任何这些功能，我们可以。您的第一个想法可能是重新实现此功能有点像重新发明轮子。幸运的是，Electron可以轻松创建执行常见操作系统任务的菜单项。创建新菜单项时，可以在其上设置许多选项。到目前为止，我们已经接触到label选项和type选项，我们在之前的每个列表中的第三个菜单项上设置为分隔符。

###### *实现编辑和窗口菜单*

为了在Electron中练习构建菜单，让我们从实现编辑和窗口菜单开始，类似于在Electron的默认菜单中定义它们的方式，如图7.5所示。

**清单7.4 编辑菜单模板：./app /application-menu.js**

const template = [

{

label: 'Edit', submenu: [

{

label: 'Undo',

accelerator: 'CommandOrControl+Z', role: 'undo',

},

{

label: 'Redo',

accelerator: 'Shift+CommandOrControl+Z', role: 'redo',

},

**{ type: 'separator' },**

{

label: 'Cut',

accelerator: 'CommandOrControl+X', role: 'cut',

},

{

label: 'Copy',

accelerator: 'CommandOrControl+C', role: 'copy',

},

{

label: 'Paste',

accelerator: 'CommandOrControl+V', role: 'paste',

},

{

label: 'Select All',

accelerator: 'CommandOrControl+A', role: 'selectall',

},

],

},

{

label: 'Window', submenu: [

{

label: 'Minimize',

accelerator: 'CommandOrControl+M', role: 'minimize',

},

{

label: 'Close',

accelerator: 'CommandOrControl+W', role: 'close',

},

],

},

];

if (process.platform === 'darwin') {

**const name = app.getName();** template.unshift({ label: name });

}

module.exports = Menu.buildFromTemplate(template);

###### *定义菜单项角色和键盘快捷键*

您可能注意到的一件事是，到目前为止添加的所有菜单项都有一个特殊的角色(role)属性。此设置很重要，因为复制和粘贴等功能很难手工实现。菜单项可以具有角色，该角色与操作系统向所有应用程序提供的内置功能相关。在Windows，Linux和macOS上，菜单项的角色可以设置为以下任何一种：

* + - * undo
      * redo
      * cut
      * copy
      * paste
      * selectall
      * minimize
      * close

当我们用自己的默认菜单替换默认菜单时，这些角色与我们丢失的大部分功能重叠。添加具有这些角色的菜单项会恢复菜单的功能，但不会恢复许多用户习惯使用的键盘快捷键。

Electron提供了一个称为加速器(accelerator)的附加属性，用于定义键盘快捷键以触发菜单项的操作。创建菜单项时，可以将accelerator属性设置为遵循一组Electron特定约定的字符串。清单7.5编写了一个添加复制功能的菜单项。

**清单7.5 使用角色和加速器：./app/application-menu.js**

const { app, BrowserWindow, Menu, MenuItem, shell } = require('electron');

const copyMenuItem = new MenuItem({ label: 'Copy',

accelerator: 'CommandOrControl+C', role: 'copy'

});

在Windows和Linux上，通常使用Control键为键盘快捷键添加前缀。在macOS上，将Command键用于类似目的是很常见的。除了非常规之外，Linux和Windows上也没有Command键。Electron不需要依赖process.platform和菜单项中的条件逻辑，而是提供CommandOrControl的简写。在macOS上，这会将键盘快捷键绑定到Command键。在Windows和Linux上，Electron使用Control键代替。作为附加速记，Electron提供了Cmd,Ctrl和CmdOrCtrl，它们分别别名为Command,Control和CommandOrControl。

###### *在macOS上恢复应用程序菜*

当Electron运行时，它会将模板编译为MenuItems的集合，并相应地设置应用程序的菜单。恢复常见操作（如复制和粘贴）的键盘快捷键，应用程序在Windows和Linux中的行为与预期的一样。但是，在macOS中，应用程序仍然缺少重要的功能，尤其是退出应用程序的功能。macOS中的标准应用程序菜单具有图7.6所示的结构。

Quit

Show All

Hide Others

Hide

Services

About

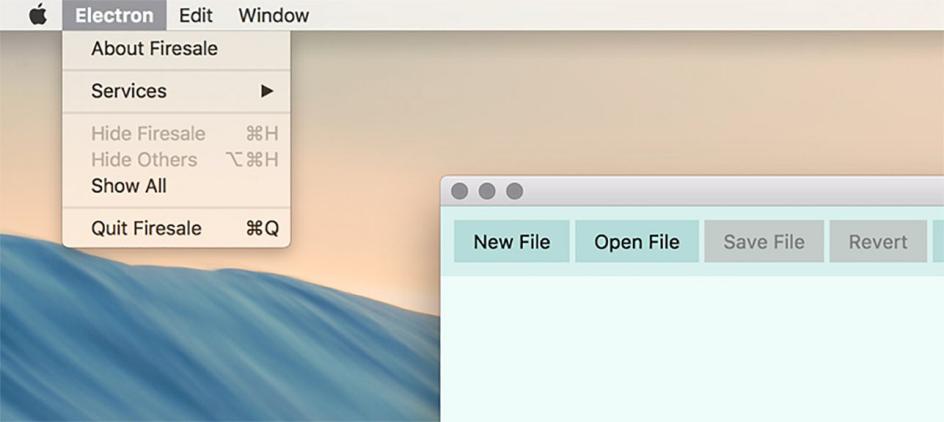
Application (macOS only)

**图7.6 macOS应用程序中应用程序菜单的结构**

在macOS上运行时，Electron提供了一组额外的角色，可以轻松恢复大多数Mac应用程序常用的应用程序菜单。这些额外的角色是

* + - * about
      * hide
      * hideothers
      * unhide
      * front
* window
* help
* services

Electron提供的默认应用程序菜单包含用于显示应用程序的About面板，公开macOS提供的服务，隐藏应用程序，隐藏所有其他应用程序以及退出应用程序的菜单项，如图7.7所示。



**图7.7应用程序菜单中的菜单项使用Electron中的特殊角色，允许您触发操作系统功能而无需重新发明轮子。**

实现应用程序菜单类似于实现编辑和窗口菜单。命令优于CommandOrControl以定义加速器，因为此菜单仅出现在macOS上。此外，我们使用模板字符串来获取About，Hide和Quit菜单的应用程序名称，因为习惯上在这些菜单项中包含应用程序的名称。

**清单7.6 macOS的应用程序菜单：./app/application-menu.js**

if (process.platform === 'darwin') {

const name = ~~'Fire Sale';~~ template.unshift({

label: name, submenu: [

{

label: `About ${name}`, role: 'about',

},

{ type: 'separator' },

{

label: 'Services', role: 'services', submenu: [],

},

{ type: 'separator' },

{

label: `Hide ${name}`, accelerator: 'Command+H', role: 'hide',

},

{

label: 'Hide Others', accelerator: 'Command+Alt+H', role: 'hideothers',

},

{

label: 'Show All', role: 'unhide',

},

{ type: 'separator' },

{

label: `Quit ${name}`, accelerator: 'Command+Q', **click() { app.quit();** },

},

],

});

}

**退出应用程序没有内置角色。相反，我们添加一个click方法，只要单击菜单项或激活键盘快捷键，就会调用该方法。**

我们的应用程序现在几乎具有macOS上本机应用程序的所有功能，但我们仍然需要解决一些细微差别。在macOS上，Window菜单有一些额外的菜单项,最值得注意的是Bring All to Front，它将应用程序的所有窗口移动到堆栈的前面。此外，macOS独占窗口角色还增加了从窗口菜单关闭和最小化当前窗口的功能，以及所有应用程序窗口的列表，以及将它们全部放在前面的能力。在不支持此角色的平台上会忽略此角色。

**清单7.7 组合应用程序，编辑和窗口菜单：./app /application-menu.js**

const template = [

{

label: 'Edit', submenu: [

// "Edit" menu shown in Listing 7.4

],

},

{

label: 'Window', **role: 'window',** submenu: [

**窗口菜单上的窗口角色使Electron在macOS中运行时添加菜单末尾所有打开窗口的列表。**

// "Window" menu shown in Listing 7.4

],

},

];

if (process.platform === 'darwin') { const name = app.getName(); template.unshift({

label: name, submenu: [

// #Application menu shown in Listing 7.6

],

});

const windowMenu = template.find(item => item.label === 'Window'); windowMenu.role = 'window';

windowMenu.submenu.push(

{ type: 'separator' },

{

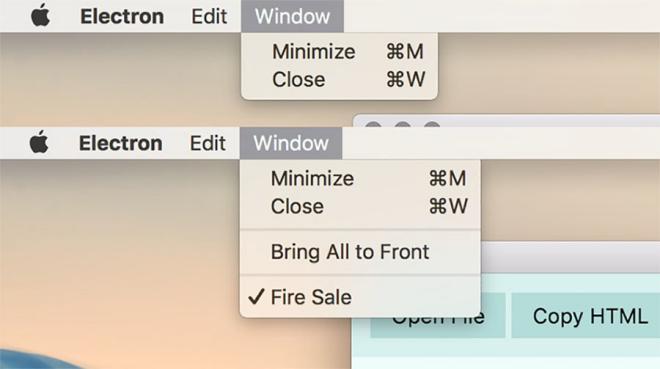
label: 'Bring All to Front', role: 'front',

}

**Array.prototype.find( )方法遍历我们的菜单模板并查找标签为Window的菜单。如果项目的顺序发生变化，这种方法可以适应变化。**

);

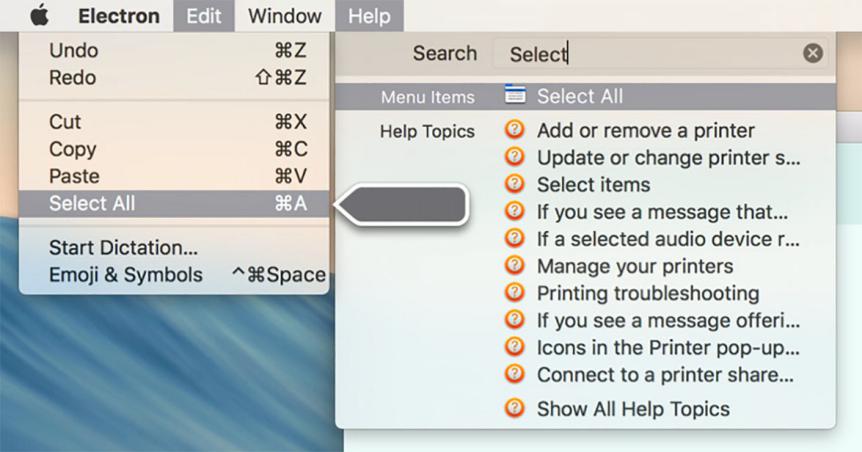
} **将菜单的角色设置为“window”。这样可以显示当前打开的窗口列表，如图7.8所示。**

图7.8 macOS中的Window菜单允许您查看应用程序中当前打开的所有窗口。

###### *添加Help菜单*

添加帮助菜单是一个很好的做法，无论平台如何，但在macOS上这样做有额外的好处。即使您的应用程序还没有任何文档或支持，内置的Help菜单允许用户搜索应用程序以查找菜单项，如图7.9所示。这适用于大多数macOS应用程序，可用于快速搜索深层嵌套菜单。您可以按Command-Shift-来访问菜单搜索？随时。

要向应用程序添加“Help”菜单，例如图7.10中所示的结构，请添加一个附加菜单，其中包含帮助角色和其他菜单项的子菜单。您必须提供一个数组作为子菜单，如清单7.8所示，即使它是空的。目前，我们还可以添加触发开发人员工具的功能。根据应用程序的不同，您可能希望在发布应用程序之前删除此功能。也就是说，Atom，Nylas Mail和Visual Studio Code等流行的应用程序已选择将其保留。



**图7.9 在macOS上，Help菜单允许您搜索菜单中的项目。**

Help

Visit Website **图7.10 我们在清单7.8中构建的Help菜单的结构。**

Search (macOS only)

**清单7.8 创建Help菜单：./ app/application-menu.js**

const template = [

// "Edit" and "Window" menus defined in Listing 7.7

{

label: 'Help',

role: 'help', submenu: [

{

label: 'Visit Website',

click() { /\* To be implemented \*/ }

},

{

label: 'Toggle Developer Tools', click(item, focusedWindow) {

**Click方法可以选择将菜单项本身和当前聚焦的窗口作为参数。**

if (focusedWindow) focusedWindow.webContents.toggleDevTools();

}

}

],

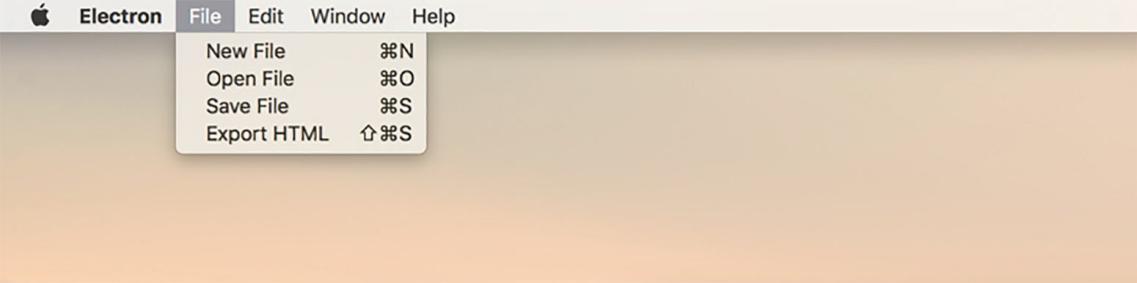
}

];

click( )方法最多可以选择三个参数：菜单项本身，当前聚焦的BrowserWindow实例和事件对象。在清单7.8中，我们使用第二个参数,当前关注的窗口,来确定我们应该告诉哪个窗口切换开发人员工具。

### *添加特定于应用程序的菜单功能*

通过所有这些工作来恢复我们最初免费获得的许多功能，只有当我们将其用作模板来添加自定义功能时才值得。用户通常希望能够从File菜单中打开和保存文件。Fire Sale目前缺乏此功能。现在，我们可以使用UI中的Open File按钮从文件系统中选择并打开Markdown文件。我们的下一步（如图7.11所示）是使用New File，Open File，Save File和“Export HTML”菜单项以及键盘快捷键修改File菜单以触发每个操作。



**图7.11 在本节中，我们将添加一个具有特定于应用程序功能的File菜单。**

当用户单击Open File菜单项或按下键盘快捷键时，菜单项会从主进程触发与UI中的按钮触发相同的openFile( )函数。单击New File从主进程调用createWindow( )函数。让我们首先在我们的模板中添加一个File菜单，其中每个特性如图7.11所示，作为子菜单数组的菜单项。

但是，在保存或导出文件的情况下，我们分别需要Markdown窗格或HTML窗格的当前内容。如果存在，我们还需要当前打开文件的名称，因为主进程无法访问此信息。相反，我们向当前关注的窗口发送消息，它应该为我们收集此信息，然后触发与用户单击UI中的按钮时相同的功能。

**清单7.9 自定义菜单功能：./app/application-menu.js**

const template = [

{

label: 'File', submenu: [

{

label: 'New File',

accelerator: 'CommandOrControl+N', click() {

mainProcess.createWindow();

**当选择New File时，告诉主进程创建一个新窗口**

}

},

{

label: 'Open File',

accelerator: 'CommandOrControl+O', click(item, focusedWindow) {

mainProcess.getFileFromUser(focusedWindow);

},

},

{

label: 'Save File',

accelerator: 'CommandOrControl+S', click(item, focusedWindow) {

focusedWindow.webContents.send('save-markdown');

},

},

{

label: 'Export HTML',

accelerator: 'Shift+CommandOrControl+S', click(item, focusedWindow) {

focusedWindow.webContents.send('save-html');

**提示用户选择要在当前窗口中打开的新文件**

向当前聚焦的窗口发送消息以保存其Markdown

**将消息发送到当前聚焦的窗口以导出其HTML**

},

},

],

},

// "Edit", "Window", and "Help" menus are defined here as well.

];

向聚焦窗口发送消息是成功的一半。我们仍然需要配置渲染器进程来侦听这些消息并采取相应的措施。让我们设置一个IPC监听器来接收这些消息，并在收到消息时调用我们现有的保存和导出功能。

**清单7.10 将IPC监听器添加到渲染器进程：./app/renderer.js**

当在save-markdown通道上收到消息时，将消息发送回主进程，其中包含当前打开文件的名称（如果有）以及来自DOM的文本内容。

ipcRenderer.on('save-markdown', () => {

mainProcess.saveMarkdown(currentWindow, filePath, markdownView.value);

});

ipcRenderer.on('save-html', () => {

mainProcess.saveHtml(currentWindow, filePath, markdownView.value);

});

**在save-html通道上收到消息时，将消息发送回主进程，其中包含当前打开文件的名称和呈现的HTML。**

###### *处理没有聚焦窗口的情况*

在Windows和Linux中，应用程序在所有窗口关闭时退出。在macOS上，即使所有窗口都已关闭，应用程序仍会继续运行。单击图标时会打开一个新窗口，但在某些情况下，用户可能会选择我们刚刚实现的三个菜单项之一，并且未定义焦点窗口。在第9章中，我们将介绍如何启用和禁用菜单项。现在，我们采用一种更简单的方法：如果用户选择Open File，则打开一个新窗口，如果没有要保存或导出的内容，则显示错误消息。

要在用户尝试保存或导出不存在的文件时显示错误消息，我们使用dialog.showErrorBox( )，它类似于dialog.showMessageBox( )，但专门用于显示错误消息，并且没有用于配置的任何选项。

**清单7.11 在尝试保存或导出不存在的文件时显示错误：./app /application-menu.js**

const { app, **dialog**, Menu, MenuItem shell } = require('electron'); const mainProcess = require('./main');

const template = [

{

label: 'File', submenu: [

{

**需要Electron的对话框模块**

label: 'New File',

accelerator: 'CommandOrControl+N', click() {

mainProcess.createWindow();

}

},

{

label: 'Open File',

accelerator: 'CommandOrControl+O', click(item, focusedWindow) {

mainProcess.getFileFromUser(focusedWindow);

},

},

{

label: 'Save File',

accelerator: 'CommandOrControl+S', click(item, focusedWindow) {

**if (!focusedWindow) {**

return dialog.showErrorBox( 'Cannot Save or Export',

**使用dialog.showErrorBox( )显示警报，从中提前返回**

**。**

'There is currently no active document to save or export.'

);

}

focusedWindow.webContents.send('save-markdown');

},

},

{

label: 'Export HTML',

accelerator: 'Shift+CommandOrControl+S', click(item, focusedWindow) {

**if (!focusedWindow) {**

**return** dialog.showErrorBox( 'Cannot Save or Export',

**如果用户尝试导出不存在的文件，则提供相同的功能**

'There is currently no active document to save or export.'

);

}

focusedWindow.webContents.send('save-html');

},

},

],

},

如果用户选择Open File并且没有可用于接收命令的窗口，则事情几乎没有希望。我们只需创建一个新窗口，等待它显示，然后触发“文件选择”对话框，就像窗口一直存在一样。

**清单7.12 当用户打开一个新文件时创建一个窗口：./app/application-menu.js**

const template = [

{

label: 'File', submenu: [

**当显示新窗口时，提示用户选择一个文件，就好像窗口一直在那里一样。**

{

label: 'Open File',

accelerator: 'CommandOrControl+O', click(item, focusedWindow) {

if (focusedWindow) {

**如果定义了focusedWindow，则使用我们在本章前面定义的功能。**

return mainProcess.getFileFromUser(focusedWindow);

}

const newWindow = mainProcess.createWindow();

newWindow.on('show', () => {

mainProcess.getFileFromUser(newWindow);

});

},

**如果没有focusedWindow，则使用我们在第5章的主进程中创建的createWindow( )函数创建一个。**

}, // "Save File" and "Export HTML" menus are defined here.

],

}, // "Edit", "Window", and "Help" menus are defined here.

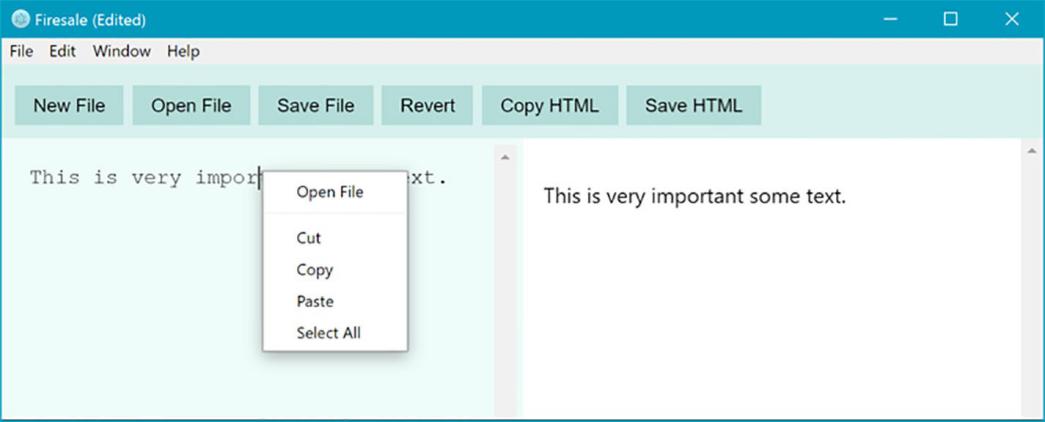
];

首先，我们检查是否有一个focusWindow。如果有，我们希望触发我们之前实现的功能并尽早从函数返回。如果没有焦点窗口，我们需要创建一个。幸运的是，我们在第5章中创建了一个函数来帮助我们完成这个过程。当新窗口完成初始化时，我们使用它，因为我们将使用任何现有窗口。我们的代码现在能够适应这种情况，我们已准备好继续前进。

译注:如果读者想验证在macOS下作者本节代码的功能,需要这样做,运行File Sale程序后,关闭File Sale的所有可见窗口,此时会见到top栏上还是有FileSale的菜单,这样点击File菜单里的Save File 或者Export Html功能就会触发这些代码.

### *构建上下文菜单*

在上一节中，我们定义了一个菜单，并在app模块触发其“ready”事件时将其设置为主进程中的应用程序菜单。我们的应用程序一次只能有一个应用程序菜单。但是，我们可以在渲染器进程中定义其他菜单，如图7.12所示，当用户右键单击（或在某些计算机上双指单击）UI的一部分时，该菜单会生效。



**图7.12 Electron允许开发人员在用户右键单击DOM的特定部分时定义自定义上下文菜单。**

接下来，我们在左侧标记窗格中监听contextmenu事件。

**清单7.13 监听contextmenu事件：./app/renderer.js**

markdownView.addEventListener('contextmenu', (event) => {

event.preventDefault();

alert('One day, a context menu will go here.');

});

请注意，除非用户单击左窗格，否则不会触发alert。如果需要从应用程序内的任何位置触发的上下文菜单，请监听窗口对象而不是DOM节点。菜单模块在渲染器进程中不可用，但可以使用远程模块从主进程的上下文访问它，如下面的清单所示。导入后，我们可以使用Menu.buildFromTemplate( )来构建菜单，如清单7.15所示。

**清单7.14 创建上下文菜单：./ app/renderer.js**

const { **remote**, ipcRenderer } = require('electron'); const { **Menu** } = remote;

const path = require('path');

const mainProcess = remote.require('./main.js'); const currentWindow = remote.getCurrentWindow();

**导入通过远程模块从主进程的上下文中的Menu模块。**

***本章小结* 141**

// Our existing renderer code…

const markdownContextMenu = Menu.buildFromTemplate([

{ label: 'Open File',

click() {

// 作者的原代码,可能作者没有验证此功能,但是因为没有传递窗口对象,此处并没有实现预期的功能

~~mainProcess.getFileFromUser();~~

**mainProcess.getFileFromUser(currentWindow); // 译注:这才是正确的代码**

} },

{ type: 'separator' },

{ label: 'Cut', role: 'cut' },

{ label: 'Copy', role: 'copy' },

{ label: 'Paste', role: 'paste' },

{ label: 'Select All', role: 'selectall' },

]);

要触发此菜单，请将contextmenu事件监听器替换为将在新创建的菜单上调用popup( )方法的函数，如此处所示。

**清单7.15 触发上下文菜单：./app/renderer.js**

markdownView.addEventListener('contextmenu', (event) => {

event.preventDefault();

**markdownContextMenu.popup();**

});

popup( )方法有四个参数：BrowserWindow，x，y和一个positioningItem。所有这些参数都是可选的，如果它们被省略，那么上下文将直接显示在鼠标光标下的当前浏览器窗口中，这是我们在此上下文中所期望的行为。有了这些代码，我们现在可以在Markdown窗格中触发上下文菜单。我们在上下文菜单中添加了功能以及其他上下文菜单，因为我们为应用程序添加了更多功能。本章的完整代码可以在https://github.com/electron-in-action/firesale/ tree/chapter-7或附录中找到。或者，您可以从https://github.com/electron-in-action/firesale.git从GitHub存储库中克隆，查看第7章分支，然后运行npm install以查看它的运行情况。

### *本章小结*

* Electron允许开发人员构建自定义应用程序和上下文菜单。
* Electron提供了用于构建菜单的Menu和MenuItem模块。
* Menu.buildFromTemplate( )允许开发人员从JavaScript对象数组中构建菜单，而不必使用MenuItem构造函数。
* Electron带有内置的应用程序菜单，其中包含合理的默认值。覆盖此菜单意味着我们必须替换内置功能。
* process.platform允许开发人员检测其应用程序运行的操作系统。
* macOS需要一个特殊的应用程序菜单作为其第一个菜单项。
* Electron为MenuItem提供角色，允许开发人员轻松实现通用的操作系统级功能。
* MenuItems有一个click( )方法，用于定义用户单击时的行为。
* MenuItems支持加速器属性，允许开发人员定义键盘快捷键以触发其操作。
* Electron支持渲染器进程中的contextmenu事件，当用户右键单击DOM时，该事件将触发。

# *进一步的操作系统集成和动态启用菜单项*

***本章包括***

* 在本机操作系统的文件系统管理器中显示文件
* 在操作系统的默认应用程序中打开该文件类型的文件
* 动态修改应用程序和上下文菜单
* 在默认浏览器中打开URL而不是应用程序本身

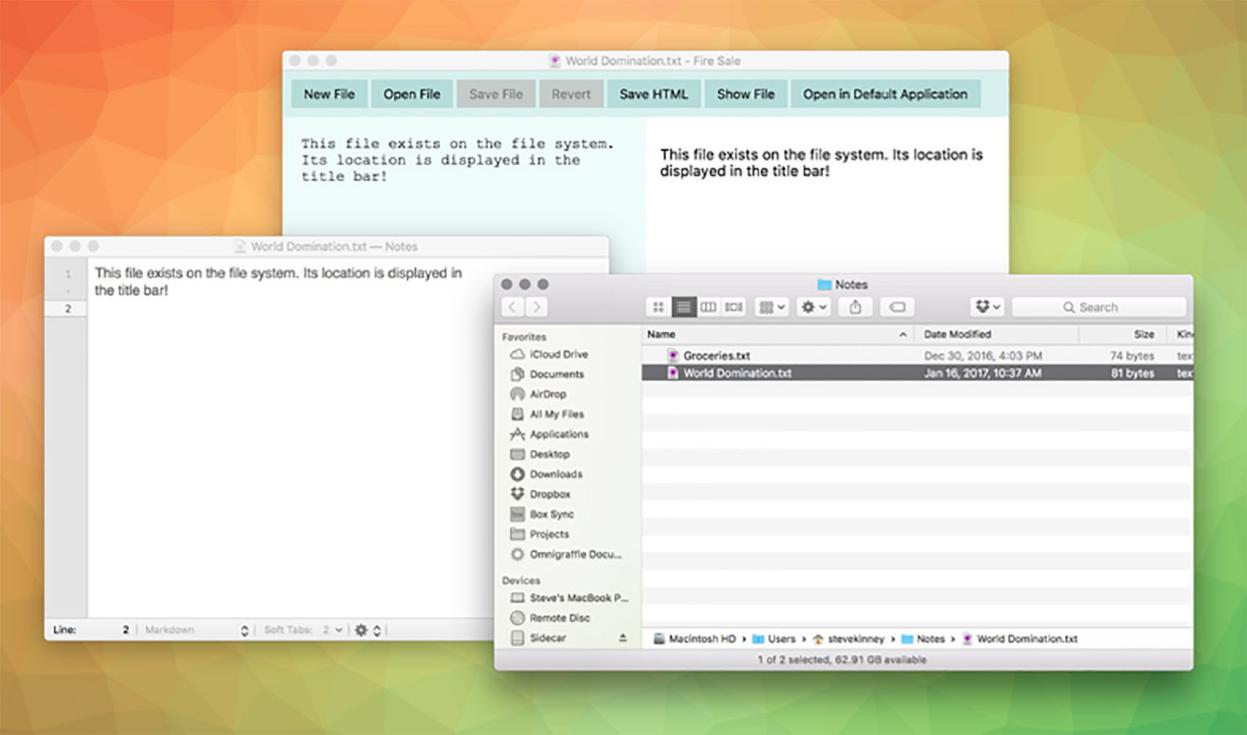
除了创建自定义应用程序模块，打开隔离的浏览器窗口进程，以及访问Electron中的本机文件对话框，我们还可以使用shell模块与操作系统进行交互。

您的应用程序如何知道系统打开网站的默认浏览器？ 如果要在其默认应用程序中打开图像文件，应将其发送到哪个应用程序？

我们的应用程序的用户通常在操作系统本身中设置这些首选项。我们最好只是向操作系统询问答案，而不是试图找出我们用户的设置。Electron提供了shell模块，使我们可以轻松实现这一目标。

与Electron提供的一些其他模块相比，shell模块相对较小。它允许我们在操作系统的文件浏览器中打开给定的文件路径，将其发送到默认应用程序，将其移动到废纸篓，触发系统蜂鸣声，并在Windows中创建快捷方式链接。shell模块在第2章中做了简短的介绍，当时我们使用它来确保在用户选择的Web浏览器中打开超链接，而不是在应用程序本身中打开，虽然这也恰好是Web浏览器。

在本章中，我们将这些功能的一部分,在文件浏览器中打开文件的位置并将文件发送到操作系统的默认应用程序,并在Fire Sale中的多个位置实现它们，如图8.1所示。我们将此功能添加到Fire Sale的UI，用户右键单击编辑器窗格时打开的上下文菜单以及应用程序的菜单。根据您在应用程序中实现的功能，这些功能中的全部或部分可能是适合您的目的的正确方法。当我们在本章的三个地方中的每个地方实施它们时，我将讨论每种方法的优缺点。本章中的所有代码都可以在GitHub的第8章分支中找到（https://github.com/electron-in-action/firesale/tree/chapter-8）。

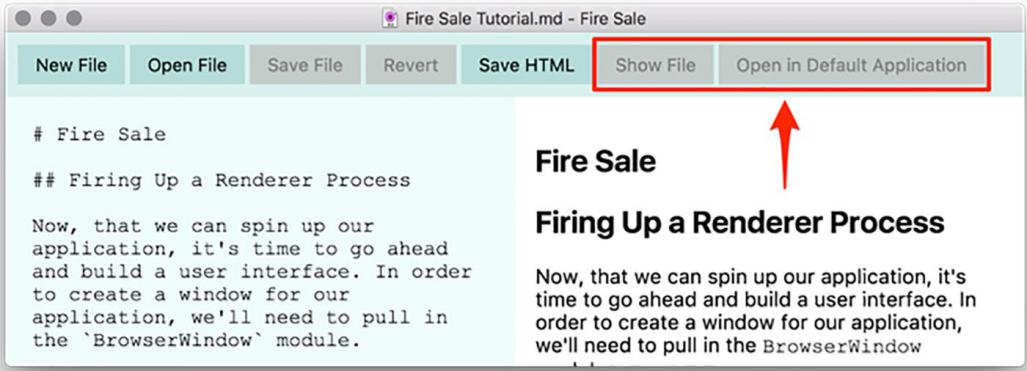


**图8.1 Electron允许我们与操作系统集成，以触发文件浏览器导航到给定文件的位置或在该文件类型的默认应用程序中打开文件路径。**

***在渲染器进程中使用UI中的shell模块* 145**

### *在渲染器进程中使用UI中的shell模块*

在第7章中，在学习如何实现自定义菜单时，我们像在传统浏览器环境中一样为Fire Sale添加了功能：通过向DOM添加按钮。可以从UI中的按钮，应用程序和上下文菜单项，通过击键等触发Electron shell模块，该模块可用于主进程以及渲染器进程。但是，让我们从我们擅长的方面开始：从UI触发功能。Show File和Open in Default Application的按钮，如图8.2所示，自第3章开始就在UI中。现在轮到他们了。打开文件或首次将新文件保存到文件系统时，将启用按钮。当用户单击Show File按钮时，将打开文件浏览器，以便用户可以导航到包含当前打开文件的目录。选择Open in Default Application按钮后，操作系统将在应用程序中打开文件，该文件通常会在用户计算机上打开Markdown和文本文件。



**图8.2自第3章以来，Show File和Open in Default Application按钮已经存在。在本章中，我们将使用Electron shell模块来实现它们的功能。**

在前面的章节中，默认情况下已禁用Show File和Open in Default Application按钮。与前一章中实现的Save和Revert按钮类似，有些情况下应启用按钮。使用Save和Revert按钮，只要文件的原始内容,或者在新的未保存文件的情况下为空字符串,与左编辑器窗格中当前显示的内容不同。因此，我们在编辑器中侦听了键盘事件并比较了内容以评估是否应该启用按钮。对于Show File和Open in Default Application，我们并不关心它们是否已被修改。我们想要的是分别显示或打开的文件路径。

**清单8.1 当有要显示的文件时启用按钮：./app/renderer.js**

const { remote, ipcRenderer, **shell** } = require('electron'); //译注:引入shell模块

const renderFile = (file, content) => {

filePath = file;

originalContent = content;

markdownView.value = content; renderMarkdownToHtml(content);

**showFileButton.disabled = false; openInDefaultButton.disabled = false;**

**当文件路径已发送到要显示的渲染器进程时，我们激活Show File和Open in Default按钮。**

updateUserInterface(false);

};

每当我们处理存储在文件系统中的文件而不是新的未保存文件时，都应该启用Show File和Open in Default Editor按钮。每当更改活动文件时，我们都会更新filePath变量。设置此变量后，启用按钮。

**清单8.2 向按钮添加事件监听器：./app/renderer.js**

用户似乎不太可能点击禁用按钮，但我们通过显示更有用的按钮并从函数返回来防止他们看到一个神秘的错误。

const showFile = () => {

if (!filePath) {

return alert('This file has not been saved to the filesystem.');

}

shell.showItemInFolder(filePath);

};

const openInDefaultApplication = () => {

if (!filePath) {

**触发操作系统的本机文件浏览器以打开一个新窗口，突出显示提供的文件路径。**

return alert('This file has not been saved to the filesystem.');

}

shell.openItem(filePath);

};

**请求由用户指定的默认应用程序打开提供的操作系统。**

showFileButton.addEventListener('click', showFile); openInDefaultButton.addEventListener('click', openInDefaultApplication);

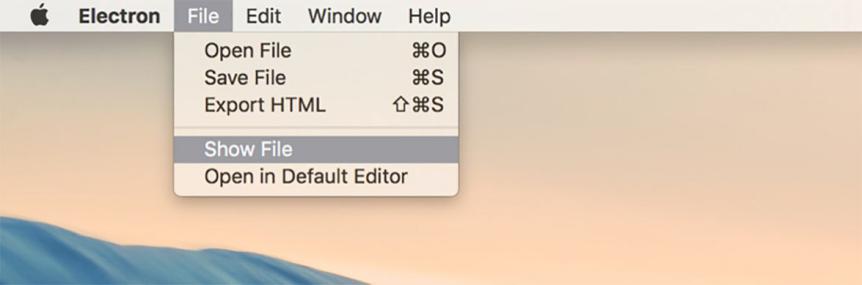
**单击任一按钮时，我们会触发前面声明的两个函数。通过省略括号，我们提供了对单击按钮时要执行的函数的引用，而不是立即调用它。**

启用按钮很有帮助，但如果按钮执行某些操作会更好。只要单击一个按钮，它就会从shell模块触发相应的方法。您可能会注意到我决定在常量中定义函数，而不是仅将匿名函数传递给addEventListener。在本章后面，我们从Fire Sale的应用程序菜单以及上下文菜单中调用这些函数。这些名称允许我稍后引用这两个函数。

### *在应用程序菜单中使用shell模块*

我们可能会考虑添加此功能的第一个地方是应用程序菜单，位于允许用户保存和导出文件的菜单项下方。在本节中，我们实现了两个新的菜单项:Show File和Open in Default Editor，在图8.3中，它将文件暴露在其包含的文件夹中，并在应用程序中打开该文件(设置为操作系统中标记文件的默认值)。

***在应用程序菜单中使用shell模块* 147**



**图8.3 与Electron shell模块配合使用以与本机操作系统通信的菜单项**

您必须将这两个新菜单项添加到应用程序菜单模板，以便在应用程序启动时显示。如果没有窗口打开怎么办？ 在完美的世界中，除非至少打开一个窗口，否则我们将禁用菜单项。但这超出了本章的范围，因此我们将做下一个最好的事情并显示有用的错误消息，以帮助用户查看其方式的错误。

**清单8.3 添加其他菜单项：./ app/application-menu.js**

const { app, BrowserWindow, dialog, Menu, **shell** } = require('electron'); const mainProcess = require('./main');

const template = [

{

label: 'File', submenu: [

//Additional submenu items above.

{ **type: 'separator'** },

{

**将Electron的shell模块添加到electron库中所需的模块列表中。**

**Electron的菜单模板允许我们在其他更多功能菜单项之间轻松定义分隔符。**

label: 'Show File',

accelerator: 'Shift+CommandOrControl+S', click(item, focusedWindow) {

if **(!focusedWindow)** {

return dialog.showErrorBox( 'Cannot Show File\'s Location',

**如果没有聚焦窗口，那么我们向用户显示错误并从函数返回。**

'There is currently no active document show.'

);

}

focusedWindow.webContents.send('**show-file'**);

},

},

{

**如果用户选择Show File菜单项，我们会通过IPC通过show-file通道向最前面的窗口发送消息。**

label: 'Open in Default Editor',

accelerator: 'Shift+CommandOrControl+S', click(item, focusedWindow) {

if (!focusedWindow) {

return dialog.showErrorBox(

'Cannot Open File in Default Editor',

'There is currently no active document to open.'

);

}

focusedWindow.webContents.send('**open-in-default**');

},

},

],

},

**如果用户选择Open in Default Editor菜单项，我们会通过IPC通过open-in-default通道向最前面的窗口发送消息。**

// Edit, Window, and Help menus defined here.

];

在清单8.3中，我们在上一章中添加的菜单项和本章中添加的菜单项之间添加了一个分隔符。此分隔符仅出于美观原因，并且菜单无法正常工作。在分隔符之后，我们添加了两个新菜单项：Show File和Open in Default Editor。每个窗口都引用当前在其中呈现的文件的路径。那么，我们如何知道要显示或打开的文件？ 我们可以设计一个复杂的系统来确定在哪些窗口中打开了哪些文件，或者我们可以通过进程间通信（IPC）向窗口发送消息，然后让窗口为自己找出它。

为了实现这一点，我们通过show-file和open-in-default通道监听IPC消息。幸运的是，我们之前足够聪明，可以指出负责处理这些请求的功能。这使得设置听众变得容易。

**清单8.4 为Show File实现IPC监听器并在默认应用程序中打开：./app/renderer.js**

ipcRenderer.on('show-file', showFile);

**当ipcRenderer从主进程接收到show-file事件时，触发showFile( )函数。**

ipcRenderer.on('open-in-default', openInDefaultApplication);

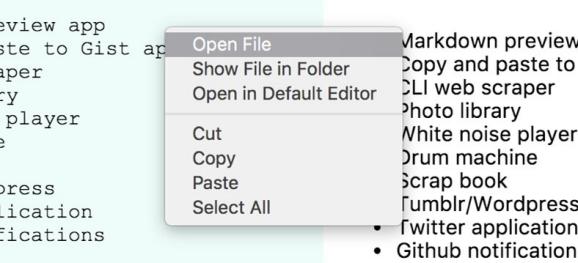
**当ipcRenderer从主进程接收到open-in-default事件时，触发openInDefaultApplication( )函数。**

###### *8.2.1 Electron shell模块的其他功能*

Electron shell模块还允许开发人员将给定文件移动到macOS Trash或Windows回收站，而不必关注每个平台上如何工作的细节。它还可以允许开发人员触发系统蜂鸣声。在Windows上，开发人员也可以创建和读取快捷方式链接。

### *从上下文菜单访问shell模块*

您还可以将这些功能添加到我们最初添加到第7章中的应用程序的上下文菜单中。当用户右键单击左侧编辑器窗格时，会出现此菜单，如图8.4所示。要实现此功能，我们需要结合本章前面的两种方法：我们将菜单项添加到模板中，但由于我们已经在渲染器进程中，我们可以直接调用函数,就像我们使用 UI中的按钮,而不是必须通过IPC发送消息。

图8.4从上下文菜单访问shell模块。在处理新的未保存文件时，将禁用Show File in Folder和 Open in Default Editor，因为未保存的文件没有shell模块方法的有效文件路径。

**清单8.5 添加对上下文菜单模板的shell模块访问：./app/renderer.js**

const { remote, ipcRenderer, shell } = require('electron'); const { Menu } = remote;

const path = require('path');

const mainProcess = remote.require('./main.js'); const currentWindow = remote.getCurrentWindow();

**在渲染器进程中添加Electron所需的shell模块和其他模块。**

// Other renderer code from the previous chapters will remain here, but has been ommited for brevity…

const markdownContextMenu = Menu.buildFromTemplate([

{ label: 'Open File', click() { mainProcess.getFileFromUser(); } },

**{**

**label: 'Show File in Folder', click: showFile**

**},**

**{**

**label: 'Open in Default Editor', click: openInDefaultApplication**

**},**

{ type: 'separator' },

{ label: 'Cut', role: 'cut' },

{ label: 'Copy', role: 'copy' },

{ label: 'Paste', role: 'paste' },

**这里的语法略有不同，因为我们只是将click方法指向showFile( )函数的引用。当用户单击菜单项时，它将被打开。**

我们采用类似的方法使用Open in Default Editor按钮。

{ label: 'Select All', role: 'selectall' },

]);

###### *决定在菜单中还是在浏览器中放置功能*

何时向UI添加按钮比向应用程序或上下文菜单中的菜单项添加相同功能更好？ 它取决于功能在您的应用程序中扮演的角色。上下文菜单项不明确; 用户可能永远不会想到右键单击应用程序的给定部分。如果您要实现的功能对于应用程序至关重要，则最好将其放入UI中。也就是说，只有有限的空间可供使用，而且使用较少但功能较好的功能非常适合上下文菜单项。

###### *决定在应用程序或上下文菜单中放置功能*

什么时候使用上下文菜单比在上一节中我们在应用程序菜单中包含这些菜单项更好？ 为了简化Fire Sale，它被设计为单窗口应用程序，用户一次只编辑一个文件。当用户选择Open File in Default Editor时，应用程序会隐式知道哪个文件，因为只有一个文件。

但是，如果我们要构建一个照片管理应用程序，我们想要添加一个功能，用户可以右键单击每个照片以在Finder或Windows资源管理器中显示它？ 此功能很难添加到应用程序菜单中，因为很难准确地解读用户想要在其包含文件夹中看到的照片。通过使用上下文菜单，我们可以轻松确定用户右键单击哪张照片。

### *适当时禁用菜单项*

此时，如果用户选择了不可用的菜单项，Fire Sale会显示错误消息。它比崩溃或抛出难以理解的错误消息更好，但它不会因为成为一个出色的用户体验而赢得任何奖项。如果我们禁用上下文和应用程序菜单项，如果单击会导致错误，该怎么办？这样，用户可以清楚地知道这不是有效的操作。

菜单设置完成后可以改变菜单，但这很棘手。菜单项存储为数组，查找给定的菜单项涉及遍历所有顶级菜单项及其子菜单。大多数情况下，在将窗口显示给用户之前基于窗口的当前状态生成新的上下文菜单或者在应用程序的状态发生变化时重新生成新的应用程序菜单更容易。

我们如何解决这个问题取决于我们正在使用哪个菜单。窗口中的上下文菜单更容易，因为我们只使用一个窗口，并且我们将filePath变量存储在全局范围内。如果有filePath，那么我们启用在文件夹中显示文件并在默认情况下打开。如果filePath为false，则禁用这些菜单项。

应用程序菜单有点棘手。除了在默认应用程序中显示文件和打开之外，我们还有保存文件和导出HTML，如果文件尚未保存到文件系统，则可以保持启用，但只有在至少打开一个窗口时才应启用。见图8.5。

###### *在上下文菜单中动态启用和禁用菜单项*

在本章的前面，我们从模板创建了一个上下文菜单，然后在触发contextmenu DOM事件时调用它的popup( )方法。要动态启用或禁用 Show File和Open in Default菜单项，如图8.6所示，每次用户右键单击“Markdown”视图时，都要创建一个上下文菜单。我们根据是否有filePath启用这些项目。为了实现这一点，我们将菜单的创建移动到一个函数中，这允许我们在每次调用菜单时重新创建菜单。

|  |  |  |  |
| --- | --- | --- | --- |
|  | Enable Save File and Export HTML | |  |
|  | |  | |
|  | |  | |
|  | Is there a file loaded in the currently focused window? | | Context Menu |
| Yes No  Enable Show File and Disable Show File  Open in Default and Open in  Application Default Application | | | |

图8.5根据应用程序的状态，启用或禁用某些菜单项。在这个应用程序中，我们有两个菜单：应用程序和上下文。应用程序菜单必须跟踪所有打开的窗口，而上下文菜单应仅处理相同问题的子集，因为它可以假设至少有一个窗口是打开的。

Application Menu

Ye

**清单8.6 创建一个动态创建上下文菜单的函数：./app/renderer.js**

在第7章中，我们制作了一个单独的上下文菜单，每次用户右键单击Markdown视图时都会显示该菜单。在本章中，我们将该菜单替换为每次都创建新上下文菜单的函数。

const createContextMenu = () => { return Menu.buildFromTemplate([

**此函数的返回值是使用Electron的Menu.buildFromTemplate( )函数创建的Menu对象。**

{ label: 'Open File',

|  |  |
| --- | --- |
| Is there at least one window open? | |
| s | No |
| Disable Save File, Export HTML, Show File, and  Open in Default Application | |

click() {

//~~mainProcess.getFileFromUser();~~ 作者的原代码,可能作者没有验证此功能,但是因为没有传递窗口对象,此处并没有实现预期的功能

**mainProcess.getFileFromUser(currentWindow); // 译注:这才是正确的代码**

} },

{

label: 'Show File in Folder', click: showFile,

enabled: !!filePath

},

{

label: 'Open in Default',

click: openInDefaultApplication, enabled: !!filePath

},

{ type: 'separator' },

{ label: 'Cut', role: 'cut' },

{ label: 'Copy', role: 'copy' },

{ label: 'Paste', role: 'paste' },

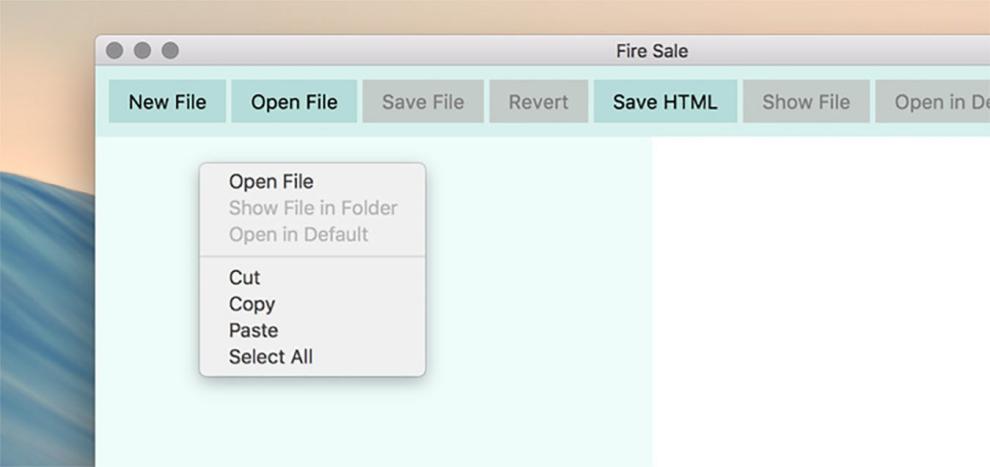
**创建菜单后，我们检查filePath是否有值。我们将filePath强制转换为布尔值以相应地启用或禁用菜单。**

{ label: 'Select All', role: 'selectall' },

]);

};

在前面的代码示例中，我们创建了一个createContextMenu( )函数，每次调用它时都会返回一个新菜单。当它创建菜单项Show File和Open in Default时，它会检查是否可以使用!!强制将filePath强制为true。enabled属性需要布尔值:true或false。如果你不熟悉这种技术，那么分别采用真实或虚假的值并将其强制转换为真或假是一种技巧。！操作符对于任何truthy值返回false，对于任何falsey值都返回true。使用它第二次将其翻转，但这次是布尔基元，如清单8.7所示。



**图8.6 如果没有打开文件，Show File in Folder和Open in Default是禁用的。**

**清单8.7 将值转换为布尔值**

null // Null falsey in JavaScript.

!null // Returns true.

!!null // Returns false.

'/Users/stevekinney/Notes/Groceries.txt' // A file path

!/Users/stevekinney/Notes/Groceries.txt' // Returns false.

!!/Users/stevekinney/Notes/Groceries.txt' // Returns true.

下一步是修改我们的事件监听器以使用我们刚刚创建的函数。每次生成新菜单时，该函数都会检查filePath是否真实或虚假，并相应地启用Show File和Open in Default Application。下一步是每次用户右键单击应用程序的Markdown视图时，使用此功能即时生成新菜单。

**清单8.8 每次触发contextmenu事件时创建上下文菜单：./app/renderer.js**

markdownView.addEventListener('contextmenu', (event) => {

event.preventDefault(); **createContextMenu().popup();**

});

**我们不再使用预先存在的菜单，而是每次都调用createContextMenu( )创建一个菜单，然后立即调用它的popup( )方法。**

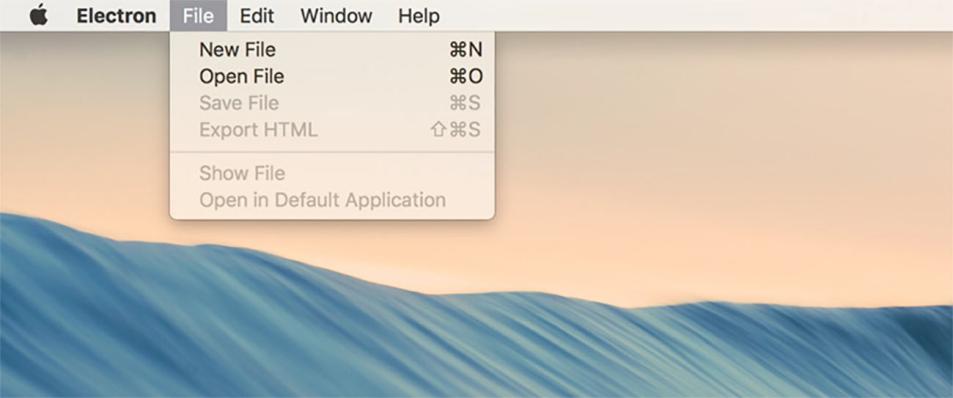
通过此更改，用户每次请求时都会获得一个新菜单。如果要在文件系统中显示或在另一个应用程序中打开文件，则会启用相应的菜单项。如果没有，他们被禁用。现在是时候将注意力转向稍微棘手的应用程序菜单了。

###### *在应用程序菜单中动态启用和禁用菜单项*

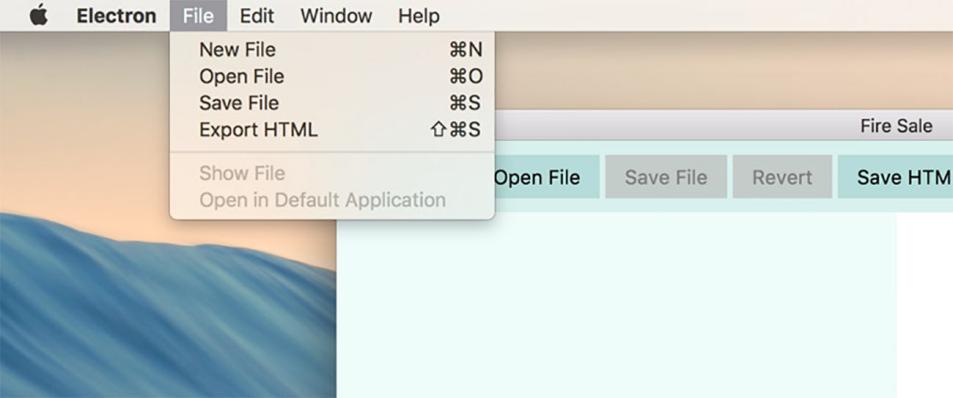
在应用程序菜单中动态启用菜单项的过程与在上下文菜单中这样做有一些相似之处。我们使用相同的方法生成新菜单并替换现有的应用程序菜单，但只有几个捕获：应用程序菜单在所有窗口共享。第一个窗口可能有一个我们可以在另一个应用程序中显示或打开的文件，但第二个窗口可能没有。在macOS上，我们可能遇到没有打开窗口的场景。在这种情况下，应该禁用Save File和Export HTML菜单项以及Show File和Open in Default Application，如图8.7所示。此外，我们的上下文菜单可以利用filePath在范围内的事实。应用程序菜单位于主进程中，无权访问此变量。见图8.8和8.9。

要实现此功能，我们采用以下方法。首先，我们将创建一个函数，该函数根据我们在前一章中实现的菜单返回一个新的应用程序菜单。每次我们创建一个新菜单时，我们都会检查是否有任何窗口以及当前关注的窗口是否代表文件系统上的文件。有了这个，当有下面的场景时,我们将修改主进程以生成新的应用程序菜单:

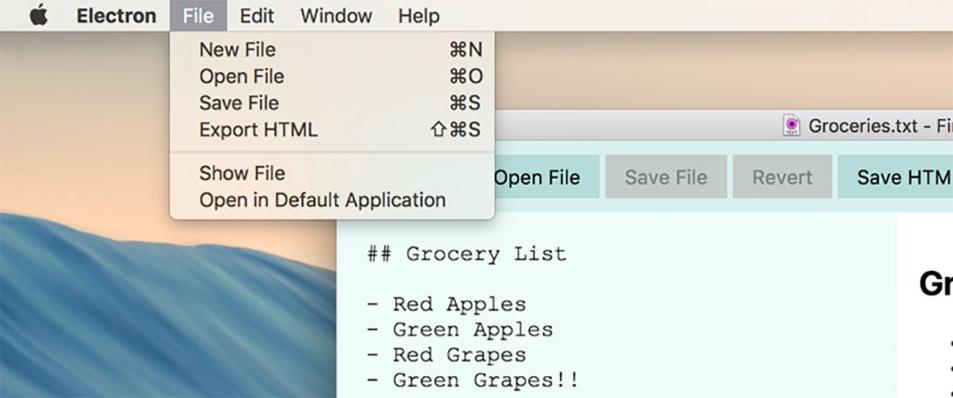
* + - * 该应用程序触发其ready事件
      * 窗口关闭。
      * 窗口获得焦点（从而成为新的焦点窗口）。
      * 打开一个文件。



**图8.7 如果没有打开窗口，则应禁用Save File, Export HTML, Show File, 以及Open in Default Application。**



**图8.8 如果存在焦点窗口，但用户正在处理尚未保存到文件系统的文件，则应启用Save File和Export HTML项。但是，不应显示Show File和Open in Default Application菜单项，因为没有要显示或打开的文件位置。**



**图8.9 如果窗口显示文件系统上的文件，则应启用所有菜单项。**

让我们首先将application-menu.js中的模板重构为一个函数，该函数将生成一个菜单并将其设置为新的应用程序菜单。

**清单8.9 根据应用程序菜单生成一个新的应用程序菜单：./app/application-menu.js**

const { app,

**BrowserWindow**, dialog,

Menu, shell

} = require('electron');

**我们导入了BrowserWindow模块，因为它有一些方法可用于推理我们应用程序中的窗口。**

查看是否有任何窗口打开的简单方法是使用BrowserWindow.getAllWindows( )。如果没有打开窗口，则此数组将为空，长度为0，这在JavaScript中是假的。

const mainProcess = require('./main');

const createApplicationMenu = () => {

**const hasOneOrMoreWindows = !!BrowserWindow.getAllWindows().length;**

const focusedWindow = BrowserWindow.getFocusedWindow();

**const hasFilePath = !!(focusedWindow && focusedWindow.getRepresentedFilename());**

const template = [

{

获取当前聚焦的窗口（如果没有聚焦窗口，则返回null）

label: 'File', submenu: [

{

label: 'New File',

accelerator: 'CommandOrControl+N', click() {

mainProcess.createWindow();

**如果有一个焦点窗口，我们使用它的getRepresentedFile( )**

**获取对当前文件路径的引用的方法。如果存在，我们将设置hasFilePath布尔值。**

}

},

{

label: 'Open File',

accelerator: 'CommandOrControl+O', click(item, focusedWindow) {

// Omitted for brevity…

},

},

{

label: 'Save File',

accelerator: 'CommandOrControl+S', **enabled: hasOneOrMoreWindows,** click(item, focusedWindow) {

// Omitted for brevity…

},

},

{

label: 'Export HTML',

accelerator: 'Shift+CommandOrControl+S', **enabled: hasOneOrMoreWindows,** click(item, focusedWindow) {

// Omitted for brevity…

**Save File和Export HTML菜单项基于hasOneOrMoreWindows布尔值设置。**

},

},

{ type: 'separator' },

{

label: 'Show File', **enabled: hasFilePath,**

click(item, focusedWindow) {

// Omitted for brevity…

},

},

{

label: 'Open in Default Application', **enabled: hasFilePath,**

click(item, focusedWindow) {

// Omitted for brevity…

**Show File和Open in Default Application菜单项被设置基于hasFilePath布尔值设置。**

},

},

],

},

// Additional Menus…

];

// Additional Functionality…

**从模板构建菜单，并将其设置为应用程序菜单。**

return **Menu.setApplicationMenu(Menu.buildFromTemplate(template));**

};

module.exports = createApplicationMenu;

**译注:**

**在Linux和Windows10下:const focusedWindow = BrowserWindow.getFocusedWindow(); 似乎是异步的返回的太迟,导致菜单模板都已经初始化好了---原因待进一步研究;在macOS下没有功能问题.**

在第7章中，我们在应用程序启动时构建了一次菜单，并在应用程序菜单准备就绪后立即将其设置为应用程序菜单。此方法仍然有效，现在该模块导出一个创建菜单并将其设置为应用程序菜单的函数。一路上，它检查是否有任何浏览器窗口，如果有一个焦点窗口，它会检查该窗口是否有一个代表的文件集，如清单8.10所示。如果您还记得，我们在打开文件时在第4章中设置了代表文件。当时，我们这样做是为了在macOS的窗口标题栏中获取小文件图标。（调用此方法在Windows或Linux上没有明显的视觉效果。）

**清单8.10 设置代表文件：./app/main.js**

const openFile = exports.openFile = (targetWindow, file) => {

const content = fs.readFileSync(file).toString(); startWatchingFile(targetWindow, file); app.addRecentDocument(file); **targetWindow.setRepresentedFilename(file);** targetWindow.webContents.send('file-opened', file, content);

**我们在打开文件时设置代表的文件路径。**

};

通过这两条信息,打开窗口的数量以及聚焦窗口是否表示文件路径,我们可以适当地切换菜单项。下一步是在这两者中的任何一个发生变化时触发此过程。触发事件包括应用程序首次启动时，新窗口聚焦时，窗口关闭时（可能是最后一个窗口）以及用户打开文件时。每当发生其中一个事件时，我们就会重新创建应用程序菜单。

**清单8.11 当状态可能已更改时生成新的应用程序菜单：./app/main.js**

const { app, BrowserWindow, dialog, Menu } = require('electron');

const **createApplicationMenu** = require('./application-menu'); const fs = require('fs');

const windows = new Set(); const openFiles = new Map();

app.on('ready', () => {

**createApplicationMenu();** createWindow();

});

app.on('window-all-closed', () => {

// Omitted for brevity…

});

**更新require语句以反映我们正在导入的新函数createApplicationMenu**

首次启动应用程序并准备就绪时创建应用程序菜单

app.on('activate', (event, hasVisibleWindows) => { if (!hasVisibleWindows) { createWindow(); }

});

const createWindow = exports.createWindow = () => { let x, y;

const currentWindow = BrowserWindow.getFocusedWindow(); if (currentWindow) {

const [ currentWindowX, currentWindowY ] = currentWindow.getPosition();

x = currentWindowX + 10; y = currentWindowY + 10;

}

let newWindow = new BrowserWindow({ x, y, show: false }); newWindow.loadFile('index.html');

newWindow.once('ready-to-show', () => { newWindow.show();

});

**newWindow.on('focus', createApplicationMenu);**

**每当新窗口获得焦点时，都会创建一个新的应用程序菜单**

newWindow.on('close', (event) => {

// Omitted for brevity…

});

newWindow.on('closed', () => {

windows.delete(newWindow); **createApplicationMenu();**

**窗口关闭时创建一个新的应用程序**

newWindow = null;

});

windows.add(newWindow); return newWindow;

};

const getFileFromUser = exports.getFileFromUser = (targetWindow) => {

// Omitted for brevity…

};

const openFile = exports.openFile = (targetWindow, file) => {

const content = fs.readFileSync(file).toString(); startWatchingFile(targetWindow, file); app.addRecentDocument(file); targetWindow.setRepresentedFilename(file); targetWindow.webContents.send('file-opened', file, content); **createApplicationMenu();**

};

// Additional functions below…

**只要打开文件并设置了表示的文件，就会创建一个新的应用程序菜单。**

由于能够创建应用程序菜单并替换现有菜单，因此在main.js中实现此功能只需在适当的时间调用此功能。有了这个，我们已经完成了我们在本节开头实现的功能。本章末尾的完整代码可以在附录书或GitHub上找到，网址为https://github.com/electron-in-action/firesale。

### *本章小结*

* shell模块使应用程序能够与操作系统通信，以在文件浏览器中显示文件，在默认应用程序中打开文件，将文件移动到废纸篓或回收站（取决于操作系统），触发系统蜂鸣声，以及 在Windows中创建或读取快捷方式链接。
* shell模块可用于主进程以及渲染器进程。因此，它可以在UI元素的应用程序菜单，上下文菜单或事件监听器中使用。

*介绍托盘模块*

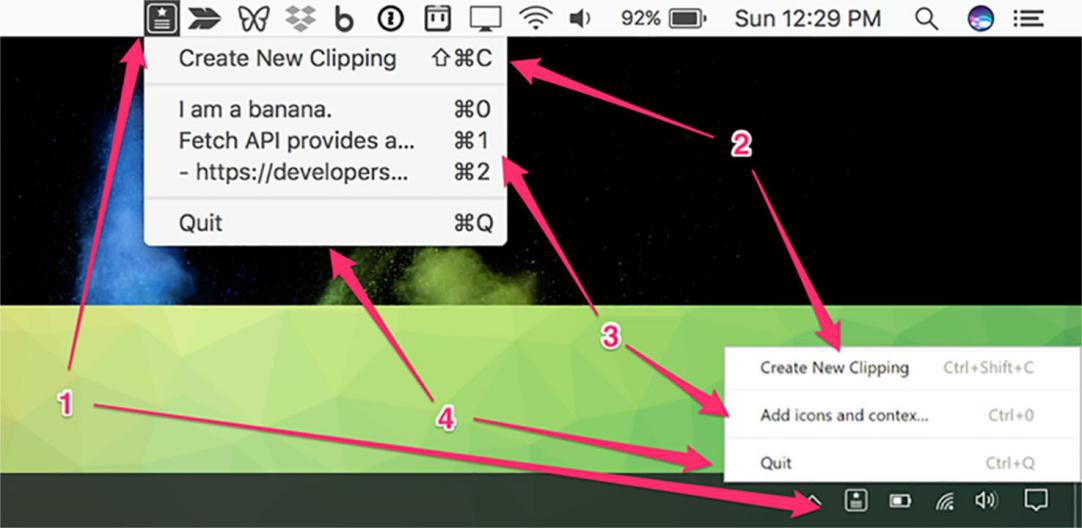
***本章包括***

* 构建一个位于macOS菜单栏或Windows系统托盘中的简单应用程序。
* 使用Electron的托盘模块创建位于操作系统菜单栏或系统托盘中的应用程序。
* 读取和写入系统剪贴板。
* 注册全局快捷方式，即使在应用程序未使用时也会监听特定的击键。
* 在macOS和Windows 10中触发本机通知。

在本书上半部分的大部分时间里，我们都参与了Fire Sale。尽管第3章中的初始功能集可以与传统的基于浏览器的Web应用程序相匹配，但我们在后续章节中添加的功能超出了大多数浏览器允许的范围。在本章中，我们从我们期望找到Web技术的领域开始：macOS菜单栏和Windows系统托盘。一开始，我们不会有渲染器进程或DOM。在本章的最后，我们将探讨如何创建隐藏的渲染器进程以访问主进程不可用的功能。在下一章中，我们将创建一个BrowserWindow作为托盘应用程序的UI。

**159**

随着 Fire Sale的落幕，我们将着手构建一个新的应用程序：Clipmaster，如图9.1所示。到本章结束时，我们将启动该应用程序。它将没有自己的停靠栏，任务栏图标或窗口。我们通过按下整个操作系统全局可用的键盘快捷键来激活它。触发快捷方式时，Clipmaster将读取并记录用户剪贴板的内容。如果用户从菜单中选择以前保存的剪贴板项，则应用程序将其放回系统剪贴板以便粘贴到另一个应用程序中。



**图9.1 这是应用程序完成后的样子。macOS位于顶部，Windows位于底部。**

如图所示，我们需要实现以下内容：

* 可从相应操作系统中的任何位置获得的菜单栏或托盘图标。
* 从剪贴板读取并将其内容放入菜单项。
* 菜单项列表。当用户单击此部分中的一个菜单项时，Clipmaster会将文本字符串写回剪贴板。
* 退出Clipmaster的最终菜单项。

如果仔细观察，您会发现每个项目都有键盘快捷键。除了这些基于菜单的加速器之外，我们还在操作系统中注册了全局快捷方式，允许用户激活此菜单或从任何地方创建新的剪辑。

Clipmaster的源代码可在https://github.com/electron-in-action/clipmaster获得。我使用master分支作为样板和本章代码的起点。还有一个名为“completed-example”的分支，其中包含本章末尾所示的代码。我随着时间展示代码，最后将提供完整的版本。

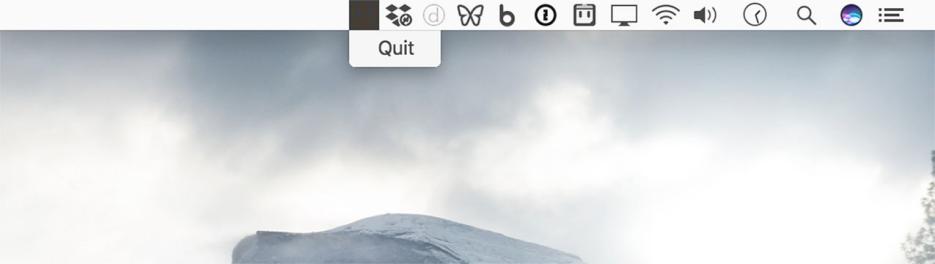
### *Clipmaster入门*

Clipmaster的文件夹结构与我们在Fire Sale中看到的大致相同，其中包含./app目录中的大部分代码。您可能会注意到没有renderer.js或index.html。这些是在本章末尾添加的，但在开头不需要。您可能还会注意到一些小图像文件。我们的应用程序需要一个图标，如果它将存在于托盘或菜单栏中。macOS希望此图标为PNG文件，如果在带有视网膜屏幕的设备上运行，Electron会自动检查是否有带后缀“@2px”的图像版本。Windows接受PNG，但最适合使用ICO文件。与基于显示器密度的高分辨率和低分辨率版本之间的内置开关不同，我们需要手动检查运行应用程序的操作系统以选择最佳图像。

要开始，请克隆master分支，然后运行npm install或yarn install来下载依赖项。安装完所有内容后，我们就可以开始构建Clipmaster了。

### *使用托盘模块创建应用程序*

为了实现滚动，我们可以将我们的应用程序添加到系统托盘或菜单栏中，只需一个功能：单击退出并关闭应用程序的功能（参见图9.2）。要完成这项英雄般的任务，我们需要Electron的托盘模块的帮助。您可以将托盘模块视为BrowserWindow的对等方。它是一个构造函数,在实例化时创建一个系统托盘或菜单栏项，其方式与BrowserWindow创建浏览器窗口的方式大致相同。



**图9.2 在第一次迭代中，Clipmaster只不过是一个允许用户立即退出它的小应用程序。不要担心：还有更多功能，到本章末尾您将拥有一个功能齐全的应用程序。**

为此，正如您在清单9.1中看到的那样，我们需要等到应用程序准备就绪，创建一个托盘实例，并为其提供一个图标和一个加载了Quit命令的菜单。当应用程序准备就绪时，我们创建菜单并将其设置为托盘实例的上下文菜单。为了实现这一目标，我们还设置了一个工具提示，当用户将鼠标悬停在我们自豪的新托盘图标上时，将显示该工具提示。见图9.3。



**图9.3 当用户将鼠标悬停在图标上时，他们会看到工具提示。这可以根据应用程序的状态进行自定义和更改。**

**清单9.1 创建托盘实例：./app/main.js**

const path = require('path'); const {

app, Menu, Tray,

} = require('electron');

let tray = null; app.on('ready', () => {

**在全局范围内声明一个变量，该变量最终存储对托盘实例的引用**

**通过使用图像路径调用构造函数来创建托盘实例**

tray = new Tray(path.join(\_ dirname, '/Icon.png'));

if (process.platform === 'win32') {

tray.on('click', tray.popUpContextMenu);

}

**在Windows上，我们注册了一个click事件监听器来打开菜单。**

const menu = Menu.buildFromTemplate([

{

label: 'Quit',

click() { app.quit(); }

**以与我们在前面章节中构建应用程序和上下文菜单相同的方式构建菜单**

}

]);

tray.setToolTip('Clipmaster'); tray.setContextMenu(menu);

});

**可选,定义用户将鼠标悬停在托盘图标上时显示的工具提示。**

创建菜单并将其设置为当用户分别在macOS和Windows中单击菜单或系统托盘中的图标时显示的菜单。

译注:tray.setToolTip('Clipmaster');在仅Windows10中有效!

与Fire Sale中的窗口一样，我们在全局范围内声明了托盘变量，以防止在应用程序就绪事件的事件监听器运行完成后的某个时间被抛出。在事件监听器内部，我们为变量分配一个新的托盘实例，并引用我们想要用作图标的图像。

如果您从第7章回忆起来，Menu.buildFromTemplate( )是一个抽象，它允许您使用对象和数组创建复杂的菜单结构。这个菜单并不复杂，但它比通过一次实例化一个MenuItem实例来构建菜单更容易。此时，菜单中有一个项目，其中Quit作为其标签。当它被点击或稍后通过按键激活时，它会告诉Electron退出应用程序。通过构建菜单，剩下的就是将其设置为托盘的上下文菜单。

Windows将托盘实例菜单视为上下文菜单。这意味着只有右键单击图标才会显示菜单。在清单9.1中，如果应用程序在Windows上运行，我们还会注册一个触发菜单的单击事件。

该应用程序可以工作，但是如果你在macOS上启动它，你会发现它也出现在Dock中。单击它不会执行任何操作，因为应用程序没有任何要显示的浏览器窗口。我们可以添加功能，例如在单击Dock栏图标时从菜单栏触发菜单的功能，但我投票表示我们完全隐藏了dock栏图标。

**清单9.2 在macOS上隐藏dock图标：./app/main.js**

app.on('ready', () => {

if (app.dock) app.dock.hide();

// …

**如果在macOS上运行，则隐藏停靠栏图标。**

}

代码行检查应用程序是否具有dock对象，如果它在macOS上运行则会执行。如果是这样，那么Electron告诉Dock图标隐藏自己。这种方法很有意思，因为它会让开发人员随意隐藏或显示停靠图标，具体取决于他们的应用程序所处的模式或用户的偏好。在我们的例子中，我们在应用程序启动时隐藏它，而不再显示它。

###### *使用macOS和Windows的正确图标*

macOS和Windows更喜欢图标的不同文件类型。他们的UI每个都有不同的颜色。默认情况下，macOS上的菜单栏为白色，黑色图标效果更好，而Windows 10有黑色任务栏，白色图标效果更好。Windows更喜欢ICO文件，而macOS使用PNG文件。为解决此问题，Node会检查其运行的平台，并根据平台获取相应的图标。Electron在提供一致的跨平台体验方面表现非常出色，这是本书中为数不多的几次，我们发现自己正在做这样的事情。

**清单9.3 有条件地选择基于平台的图标：./app/main.js**

const getIcon = () => {

if (process.platform === 'win32') return 'icon-light@2x.ico'; return 'icon-dark.png';

};

app.on('ready', () => {

if (app.dock) app.dock.hide();

**getIcon()函数检查运行应用程序的平台并返回相应的文件名。**

tray = new Tray(path.join(\_\_dirname, **getIcon**()));

// … More code below …

});

**创建新托盘实例时，使用getIcon()获取正确的文件名。**

如果我们在Windows上，它会为我们提供light ICO图标的文件名。否则，它会为我们提供黑暗PNG图标的文件名。当应用程序准备就绪时，我们使用新的getIcon()函数而不是我们最初硬编码的字符串。

###### *支持macOS中的暗模式*

早些时候，我说默认情况下macOS菜单栏是白色的。在macOS El Capitan及更高版本中，用户可以打开暗模式，它会反转菜单栏和停靠栏的颜色。在这种情况下，我们希望像通常用于macOS一样使用PNG文件，但我们也希望像Windows 10中的透明黑色系统托盘一样使用亮白版本，如清单9.4和图9.4所示。



**图9.4 如果macOS菜单栏处于黑暗模式，我们将使用相反的图标。**

**清单9.4 支持macOS上的暗模式：./app/main.js**

const path = require('path'); const {

app, Menu, Tray,

systemPreferences,

} = require('electron'); let tray = null;

const getIcon = () => {

**从Electron导入systemPreferences模块**

**使用systemPreferences.isDarkMode( )来检测macOS是否在**

**黑暗模式**

if (process.platform === 'win32') return 'icon-light.ico'; if (systemPreferences.isDarkMode()) return 'icon-light.png'; return 'icon-dark.png';

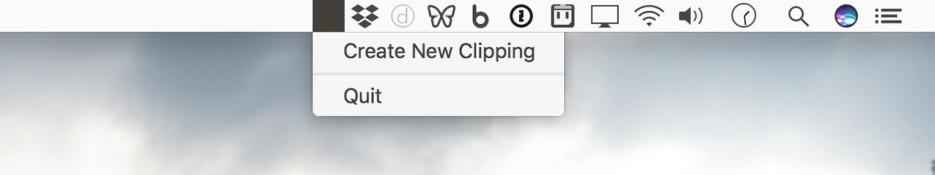
};

Electron使用systemPreferences模块可以轻松实现这一点，该模块方便地使用一个名为isDarkMode( )的方法返回一个布尔值。我将它作为练习留给读者，以确定它返回真或假的条件。

###### *从剪贴板中读取并存储剪辑*

这个应用程序应该存储剪辑，我们有很多方法可以做到这一点。在第11章中，我们将使用SQLite数据库来存储剪切。但是现在，让我们从最简单的解决方案开始：将它们存储在内存中。这种方法很简单，因为它允许我们使用内置的JavaScript数据结构，但它的缺点是每当用户退出应用程序时都会被清除。

要发布此功能，我们需要创建一个数组来保存剪辑。我们还需要创建一个从剪贴板读取并将内容添加到数组的函数，以及用户触发此函数的方法。最后，我们想要使用我们存储的剪辑更新菜单，并允许用户选择一个添加回剪贴板。见图9.5和9.6。



**图9.5 应用程序及其在macOS上的两个基本命令**



**图9.6 除了能够退出应用程序之外，用户还需要一种向应用程序添加剪辑的方法。这里的应用程序显示在Windows托盘中。**

每当用户保存新剪辑时，我们都会更新菜单。正如我们在第7章中讨论的那样，在菜单创建后可以遍历和改变菜单，但是完全替换菜单通常更容易，更有效。为了实现这一点，我们移动代码将上下文菜单创建为自己的函数，我们可以在需要更新菜单时调用它。

**清单9.5 使用数组在内存中存储剪辑：./app/main.js**

const path = require('path'); const {

app, Menu, Tray,

systemPreferences,

} = require('electron');

**const clippings = [];**  let tray = null;

**声明一个空数组来存储剪辑。**

const getIcon = () => {

if (process.platform === 'win32') return 'icon-light.ico'; if (systemPreferences.isDarkMode()) return 'icon-light.png'; return 'icon-dark.png';

};

app.on('ready', () => {

if (app.dock) app.dock.hide();

tray = new Tray(path.join( dirname, getIcon())); if (process.platform === 'win32') {

tray.on('click', tray.popUpContextMenu);

**当应用程序第一次开始构建时立即更新菜单。**

}

**updateMenu();** tray.setToolTip('Clipmaster');

});

const updateMenu = () => {

const menu = Menu.buildFromTemplate([

{

label: 'Create New Clipping', click() { null; }

**最终，我们实现了向数组添加剪辑的功能。**

},

{ type: 'separator' },

...clippings.map((clipping, index) => ({ label: clipping })),

{ type: 'separator' },

{

}

]);

label: 'Quit',

click() { app.quit(); },

**每次调用updateMenu( )时，我们都会映射剪辑数组并将它们渲染为简单的菜单项。**

tray.setContextMenu(menu);

};

**每当剪辑列表被修改后，tray.setContextMenu( )已被移动到updateMenu( )中以替换菜单。**

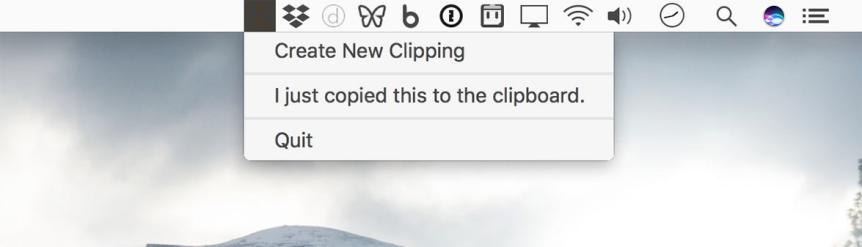
我们首先创建一个名为剪辑的简单数组，该数组具有存储剪辑的全局范围，并且可以在整个应用程序中访问。接下来，我们创建一个名为updateMenu( )的函数来生成一个新菜单，并将其设置为托盘实例的上下文菜单。updateMenu( )获取剪辑数组的内容，将其映射到可以转换为MenuItem实例的对象，然后使用ECMAScript 2015扩展运算符将它们包含为Create New Clipping和Quit菜单项的兄弟。

我们添加了一个带有标签Create New Clipping的菜单项（见图9.7），但如果你仔细观察，你会注意到我们还没有添加任何功能。为了使它工作，我们需要找到一种方法来访问操作系统的本机剪贴板。一旦我们能够做到这一点，我们可以更新此函数以从剪贴板中读取并将其内容推送到剪辑数组中，然后调用updateMenu( )来重新呈现它。

### *9.3 读写剪贴板*

在这一点上，Electron提供的模块可以让您轻松访问Linux，macOS和Windows上的操作系统剪贴板。这个模块很方便地称为剪贴板(clipboard)，这应该不足为奇。

剪贴板(clipboard)模块在主进程和渲染器进程中都可用，它有几种有用的方法来读取图像，富文本，HTML，书签和其他格式。现在，让我们保持简单并坚持使用纯文本字符串。图9.7显示了具有单个剪辑的应用程序。



**图9.7 当剪辑添加到数组中时，菜单将使用新剪辑的内容作为标签的新菜单项进行更新。**

要使Create New Clipping菜单项起作用，我们需要创建一个函数，我们可以调用它从剪贴板中读取并将其添加到剪辑数组中。本着简洁明了的精神，我们将此函数称为addClipping( )。

**清单9.6 单击菜单项时向数组添加剪辑：./app/main.js**

const path = require('path'); const {

app,

**clipboard,**

Menu,  Tray,

systemPreferences

} = require('electron');

const clippings = []; let tray = null;

app.on('ready', () => {

// …

});

**从Electron拉出剪贴板模块**

const updateMenu = () => {

const menu = Menu.buildFromTemplate([

{

label: 'Create New Clipping', click() { **addClipping();** },

**当用户单击Create New Clipping菜单项时，调用addClipping( )函数**

为Create New Clipping菜单项添加加速器。菜单处于活动状态时可用。

accelerator: '**CommandOrControl+Shift+C**'

},

{ type: 'separator' },

...clippings.map((clipping, index) => ({ label: clipping })),

{ type: 'separator' },

{

}

]);

label: 'Quit',

click() { app.quit(); }, accelerator: '**CommandOrControl+Q**'

**为Quit菜单项添加加速器。菜单处于活动状态时可用。**

tray.setContextMenu(menu);

};

const addClipping = () => {

const clipping = clipboard.readText(); clippings.push(clipping);

updateMenu(); 

return clipping;

**使用Electron的剪贴板模块从系统剪贴板中读取文本**

将从剪贴板读取的文本推送到剪辑数组中

}; **重新生成菜单以将新剪辑显示为菜单项**

要对Electron的剪贴板模块进行任何操作，我们需要像Electron的其他模块一样包含它。通过包含该模块，addClipping( )可以从剪贴板中读取文本并将其推送到数组中。使用数组中的新剪辑，下一个逻辑步骤是更新菜单并向用户显示新内容，如图9.7所示。

###### *写入剪贴板*

有了这个功能，我们可以从剪贴板中读取并将文本片段保存在我们的应用程序中，但是我们还没有编写功能来获取其中一个已保存的剪辑并将其写回剪贴板。就目前而言，我们的应用程序是我们过去保存的剪辑的剪贴簿。

写入剪贴板与阅读它并没有多大区别。因此，让我们提高赌注，并为与剪辑相关的菜单项分配键盘快捷键，如图9.8和清单9.7所示。当用户按下击键时，相应的剪辑被写入剪贴板。



**图9.8 应用程序现在有键盘快捷键。**

**清单9.7 将剪辑列为菜单项：./app/main.js**

const path = require('path'); const {

app, clipboard, Menu, Tray,

systemPreferences

} = require('electron');

const clippings = []; let tray = null;

app.on('ready', () => {

// …

});

const updateMenu = () => {

const menu = Menu.buildFromTemplate([

{

label: 'Create New Clipping', click() { addClipping(); },

accelerator: 'CommandOrControl+Shift+C'

},

{ type: 'separator' },

...clippings.map(**createClippingMenuItem**),

{ type: 'separator' },

{

label: 'Quit',

click() { app.quit(); }, accelerator: 'CommandOrControl+Q'

**使用createClippingMenu( )代替我们之前在剪切数组上映射时使用的匿名函数。**

}

]);

tray.setContextMenu(menu);

};

const addClipping = () => {

// …

};

const createClippingMenuItem = (clipping, index) => { return {

label: clipping,

click() { clipboard.writeText(clipping); }, accelerator: `CommandOrControl+${index}`

**创建一个名为createClippingMenuItem( )的函数**



};

}; **根据剪辑数组中的索引为菜单项指定加速器**

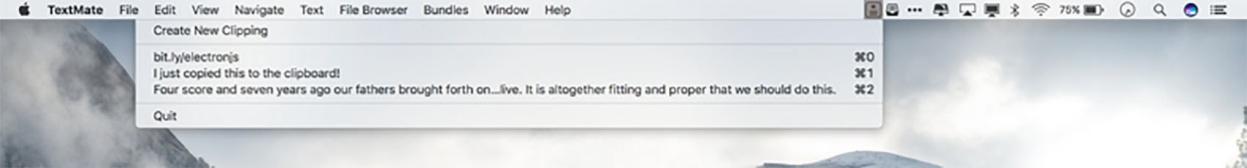
**当用户单击给定剪辑时，将其写入剪贴板。正确的剪辑包裹在一个闭包内。**

使用匿名函数将字符串映射到数组内部的菜单项是最好的短期解决方案。既然我们正在为每个菜单项添加点击方法和加速器，那么将这个过程分解为自己的功能是有意义的。

createClippingMenuItem( )将Array.prototype.map( )传递的前两个参数传递给它的回调函数：当前正在迭代的项及其索引。我们使用此索引来确定要为其分配的加速器。

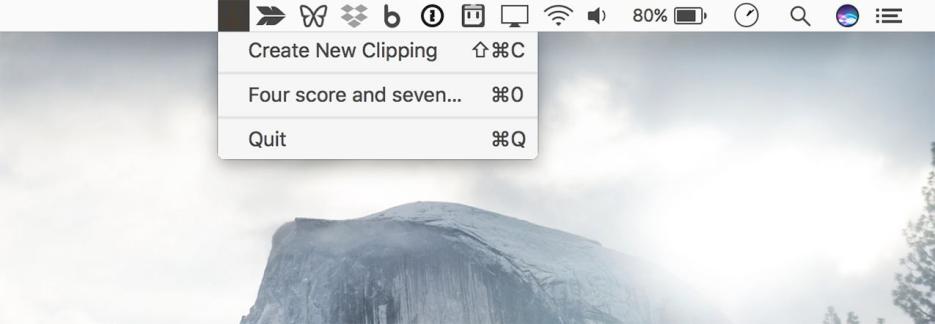
###### *处理边缘情况*

有了这个，用户现在可以通过击键将剪辑写回剪贴板。这很好，但如果用户复制了大量文本会发生什么？ 最终操作系统将削减内容，如图9.9所示，但我们需要介入并做得更好。



**图9.9 操作系统最终将截断长菜单项标签，但即使这样也有点笨拙。**

当我们遍历剪辑以创建菜单项时，我们检查它是否超过20个字符。如果是，则切掉前20个字符，添加省略号，并将其用作标签。图9.10显示了缩短的菜单项名称的示例，列表9.8给出了截断标签的代码。此截断对剪辑本身没有影响。如果用户选择剪辑，则将其全文写回剪贴板。如果裁剪小于20个字符，请将其用作标签而不进行修改。



**图9.10 剪切菜单项标签现在上限为20个字符。您可以根据自己的喜好进行调整，或创建一个允许用户控制长度的设置。**

**清单9.8 截断菜单项标签：./ main/app.js**

const createClippingMenuItem = (clipping, index) => {

return {

label: **clipping.length > 20**

**? clipping.slice(0, 20) + '…'**

: clipping,

click() { clipboard.writeText(clipping); }, accelerator: `CommandOrControl+${index}`

};

};

**如果剪辑的长度超过20个字符，则切掉前20个字符并添加省略号。**

我选择20作为任意数字。如果您愿意，可以选择其他数字。在更大的应用程序中，允许用户表达他们的偏好并使用该数字可能是有意义的。

如果用户意外添加了已存储在我们阵列中的剪辑怎么办？ 为了解决这个问题，我们检查数组是否包含当前剪辑。如果是，则提前返回以使功能短路。另一种选择是使用Set而不是数组，这是我们在Fire Sale中管理独特窗口时采用的方法。此方法适用于防止重复，但Set没有map( )方法，因此我们需要提供另一种方法将剪辑转换为菜单项。

**清单9.9 防止重复剪辑：./ app/main.js**

const addClipping = () => {

const clipping = clipboard.readText();

**if (clippings.includes(clipping)) return;** clippings.push(clipping);

updateMenu(); return clipping;

};

**检查剪辑数组是否已包含当前剪辑。如果是这样，请从函数中提前返回。**

Array.prototype.push( )将项添加到数组的末尾，因此添加的第一个剪辑始终被指定为Command-0或Control-0作为加速器。Array.prototype.unshift( )将新项添加到数组的开头。这意味着可以使用Command-0或Control-0访问最近保存的剪辑，可以使用Command-1或Control-1访问最近保存的第二个剪辑，依此类推。

**清单9.10 将剪辑添加到数组的开头：./ main.js**

const addClipping = () => {

const clipping = clipboard.readText(); if (clippings.includes(clipping)) return; **clippings.unshift(clipping);** updateMenu();

return clipping;

};

**Unshift将一个元素添加到数组的开头。**

如果用户爱上我们的应用程序，它可能会相当长。因此，限制菜单中的剪辑数是有意义的。我选择将其限制为10个项目，因为我们根据数组索引分配键盘快捷键，并且大多数键盘上没有“11”键。

**清单9.11 只显示前10个剪辑：./ app/main.js**

const menu = Menu.buildFromTemplate([

{

label: 'Create New Clipping', click() { addClipping(); },

accelerator: 'CommandOrControl+Shift+C'

},

{ type: 'separator' },

...clippings**.slice(0, 10)**.map(createClippingMenuItem),

{ type: 'separator' },

{

**使用Array.prototype.slice( )仅显示数组的前10项**

}

]);

label: 'Quit',

click() { app.quit(); }, accelerator: 'CommandOrControl+Q'

这些快捷方式有一个限制,它们只有在我们积极使用菜单时才有效。这意味着用户必须将鼠标移到菜单栏或系统托盘图标上并单击它。在该操作之后，他们可以使用提供的键盘快捷键之一。这稍微有用，但没有像全局可用的快捷方式那样有用。

译注:

1.先读取:先复制一段文本,然后点击Create New Clipping按钮即可;

2.再使用:点击应用里的剪辑项目后,粘贴即可!

3.作者在代码中设置的快捷键,仅在macOS下有效,并且点击应用在应用激活状态下!

### *注册全局快捷键*

到目前为止，我们使用的所有加速器都只在应用程序处于活动状态时才有效。通过使用globalShortcut模块，Electron还允许我们使用操作系统注册全局快捷键，即使应用程序在后台也可以激活。我们将为Clipmaster注册两个全局快捷键：一个用于触发菜单显示，另一个用于将剪贴板的内容保存到Clipmaster，而无需触发菜单。

**清单9.12 注册全局快捷方式：./app/main.js**

const path = require('path'); const {

app, clipboard,

**globalShortcut**, Menu,

Tray, systemPreferences

} = require('electron');

const clippings = []; let tray = null;

**需要Electron的globalShortcut模块**

app.on('ready', () => {

if (app.dock) app.dock.hide();

tray = new Tray(path.join( dirname, '/Icon.png'));

if (process.platform === 'win32') { tray.on('click', tray.popUpContextMenu);

}

const activationShortcut = **globalShortcut.register( 'CommandOrControl+Option+C',**

**() => { tray.popUpContextMenu(); }**

**);**

**173**

**传递定义加速器的字符串和按下加速器时应调用的匿名函数**

if (!activationShortcut) {

console.error('Global activation shortcut failed to register');

**我们注册了第二个快捷键，为数组添加剪辑。**

}

const newClippingShortcut = globalShortcut.register( 'CommandOrControl+Shift+Option+C',

() => { addClipping(); }

);

if (!newClippingShortcut) {

**如果注册失败，Electron不会抛出错误。相反，它返回undefined。在这一行中，我们检查是否定义了activationShortcut。**

console.error('Global new clipping shortcut failed to register');

}

updateMenu(); tray.setToolTip('Clipmaster');

});

const updateMenu = () => {

// …

};

const addClipping = () => {

// …

};

**如果任一快捷键失败，我们使用console.error记录问题。在更强大的应用程序中，您可能会向用户显示存在问题或实施回退。**

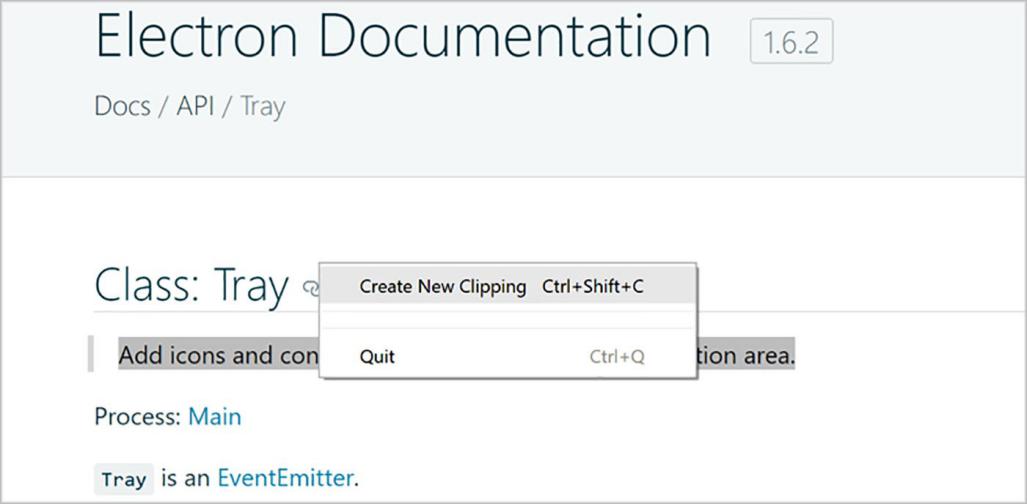
const createClippingMenuItem = (clipping, index) => {

// …

};

我们需要做的第一件事是需要Electron的globalShortcut模块。在我们有了模块后，我们可以使用它的register( )方法，该方法有两个参数：一个加速器，以及一个被激活时调用的函数。如果已经采用全局快捷键，Electron将返回null而不是抛出错误，如果未捕获，则可能会删除整个应用程序。在这种情况下，我们将一条消息记录到控制台，通知我们注册失败。在更强大的应用程序中，您可以提示用户选择新的快捷键或回退到第二个选项。

此功能在macOS和Windows上的行为略有不同。在macOS上，菜单的显示方式与用户单击时相同。但是，在Windows上，菜单显示在光标下方，如图9.11所示，而不是您所期望的系统托盘图标上方。



**图9.11 在Windows上，使用全局快捷键触发时，菜单显示在光标下方。**

译注:

Option键等效键盘上的Alt键

在Deepin上,弹出菜单的快捷键注册成功,但是不能触发,第二个可以正常触发;菜单项上不显示快捷键

在macOS上,快捷键都可以正常触发,但是菜单项上Create New Clipping的显示的快捷不对,它显示为Shift+Supper+Option+C.

在Windows 10上,功能可以正常触发,但是菜单项上Create New Clipping的显示的快捷不对,它显示为Ctrl+Shift+C,应为Ctrl+Shift+Alt+C

###### *检查注册并取消注册全局快捷键*

Electron的globalShortcut模块还提供了其他有用的方法来处理快捷键。globalShortcut.isRegistered( )返回一个布尔值，如果应用程序已经注册了快捷键，则为true; 否则，它返回false。但是，如果另一个应用程序已注册该快捷键，它也会返回false。

在Clipmaster中，您选择硬编码我们的全局快捷键，但您可以选择创建一个允许用户设置自己的键盘快捷键的UI。如果用户决定切换给定命令的快捷键，我们希望在成功注册新命令后使用globalShortcut.unregister( )取消注册旧快捷键。我们还可以使用方便命名的globalShortcut.unregisterAll( )方法取消注册所有全局快捷键。

### *显示通知*

借助我们的全局快捷方式，用户可以通过几个按钮从任何地方保存新的剪辑。但是用户如何知道他们何时成功保存剪辑？ 我们的应用程序不仅没有很多用户界面，而且在使用全局快捷方式激活它时也没有显示。一种解决方案是显示本机系统通知，如图9.12和9.13所示。

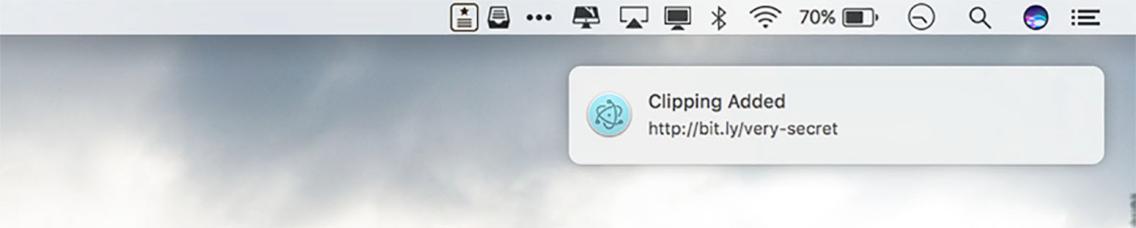
**支持通知**

**通知在macOS和Windows 10上开箱即用。它们也适用于最常见的Linux桌面环境。使用旧版本的Windows时，事情变得有点棘手。这超出了本书的范围，但官方文档中对此进行了介绍：http://mng.bz/nJR0。**

***显示通知* 175**



**图9.12 Windows 10中的通知**



**图9.13 macOS中的通知**

这是我们要采用的解决方案，但并非没有它的复杂性。Electron应用程序可以使用Chromium的Notification API创建通知。作为Web API，通知仅在渲染器进程中可用，而在主进程中不可用。

要提供此功能，我们需要创建一个隐藏的BrowserWindow实例。当用户使用全局快捷键保存新剪辑时，我们通过IPC向渲染器进程发送消息。当渲染器进程收到此消息时，它会触发通知。

让我们首先创建一个不可见的进程，然后向它发送消息并让它触发通知。如果您从Fire Sale中回忆起，BrowserWindow实例可以加载HTML，而HTML又可以像传统网页一样加载JavaScript。我们需要的第一件事是最低限度的HTML页面。

**清单9.13 设置一个简单的HTML文档：./app/index.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1">

<title>Clipmaster</title>

</head>

<body></body>

<script> require('./renderer');

</script>

</html>

**此HTML文件的目的是为渲染器进程加载JavaScript。**

此页面有一些基本标签，使其成为有效的HTML页面。重要的是它包含一个脚本标签，用于加载renderer.js，其中包含监听来自主进程的消息并触发通知的代码。接下来，让我们设置该监听器并编写代码以显示通知。

**清单9.14 监听消息和显示通知：./app/renderer.js**

const { ipcRenderer } = require('electron');

ipcRenderer.on('show-notification', (event, title, body) => {

const myNotification = new Notification(title, { body });

});

**通知是Chromium内置API的一部分，并非特定于Electron。它有两个参数：标题的字符串和附加参数的对象。在这个例子中，我们使用ES2015的增强对象文字语法提供了一个主体。这相当于{body：body}。**

**译注:**

**browserWindow.webContents.send('show-notification', 'Clipping Added', clipping,); //在Windows10中5.0.x下没有实现预期的功能**

我们引入ipcRenderer模块并开始在适当命名的show-notification通道上监听通知。除了默认包含的事件对象之外，此监听器还需要三个参数：通知的标题和正文，以及详细说明用户单击通知时要执行的操作的函数。如果没有提供任何功能，默认情况下onClick是一个空函数。

使用我们的渲染器进程的代码，我们需要在应用程序启动时运行它。在Fire Sale中，我们创建了隐藏的BrowserWindow实例，然后在加载内容时显示。在Clipmaster中，没有任何内容可以显示，因此我们从不显示窗口本身。在app.on ready块中包含此代码段。

**清单9.15 启动隐藏的浏览器窗口：./app /main.js**

const {

// . . . **BrowserWindow,**

// . . .

} = requre('electron');

browserWindow = new BrowserWindow({ **show: false**

});

browserWindow.load.URL(`file://{$\_\_dirname}/index.html`); // 译注:有两处错误

译注:正确的代码:

browserWindow = new BrowserWindow({

show: false,

**webPreferences: { //适配5.0.x版本**

**nodeIntegration: true**

**}**

});

browserWindow.**loadURL**(`file://**${\_\_dirname}**/index.html`);

即使没有显示窗口，我们也可以加载我们之前创建的HTML页面来执行代码并设置我们的监听器。现在，当我们添加剪辑时，我们会向渲染器进程发送一条消息，让它显示通知。

**清单9.16 向渲染器进程发送消息：./app/main.js**

constnewClippingShortcut=globalShortcut.register( 'CommandOrControl+Shift+Option+C',

***在macOS中按下时切换菜单栏图标* 177**

() => {

const clipping=addClipping(); **if (clipping) {**

**browserWindow.webContents.send( 'show-notification',**

**addClipping( )返回添加到数组的剪辑的字符串。**

**'Clipping Added', clipping,**

**);**

}

},

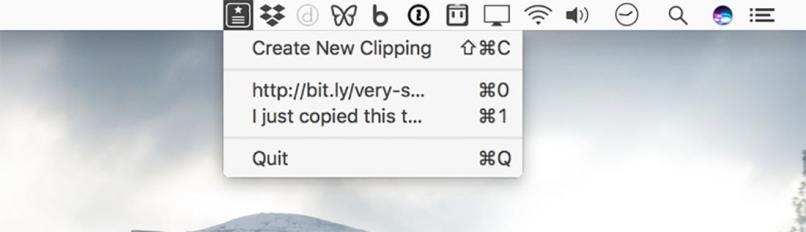
);

**如果保存了剪辑，我们会向渲染器进程发送通知，从而触发通知。**

添加剪辑后，会向渲染器进程发送一条消息。在发送消息之前，我们检查addClipping( )是否返回了值。如果你还记得，如果剪辑列表已经包含新剪辑，那么函数会提前返回值undefined。如果没有保存新剪辑，此条件会阻止发送消息。我把它作为练习留给读者，以显示一个有用的通知，通知用户剪辑已经存在。

### *在macOS中按下时切换菜单栏图标*

在macOS上，我们的菜单栏图标的行为方式与其对等方式不同。预期的行为是激活菜单时图标的颜色会反转。幸运的是，Electron可以轻松实现此功能。见图9.14。



**图9.14 单击菜单栏应用程序时，macOS可以使用备用图标。在此示例中，我们使用图标的反转版本来匹配其余的菜单栏图标。**

托盘模块有一个名为setPressedImage( )的方法。在Windows上，此方法将被忽略。在macOS上，它允许我们提供第二个图像文件的路径。单击菜单栏图标时，Electron会交换第二张图像的主图像。此清单中的代码紧跟在设置tray的语句之后，该语句也调用getIcon( )。

**清单9.17 设置按下图标时的备用图标：./ app/main.js**

tray.setPressedImage(path.join( \_dirname, 'icon-light.png'));

译注:icon-light.png的收字母i需要大写,否则找不到文件,因为文件名就是大写的.

您可能会问，“暗模式怎么样？”事实证明，macOS中的默认行为是在黑暗模式下按下图标时不反转图标。因此，您无需实现任何其他逻辑来处理这种情况。

### *完成的代码*

Clipmaster的代码与本章中的功能一致，实现如下：代码清单9.18显示了主进程的代码，清单9.19显示了渲染器进程的代码。您还可以在本章开头克隆的存储库的完整示例分支上找到此代码（http://mng.bz/xJ98）。

**清单9.18 Clipmaster完成的主要进程：./ app/main.js**

const path = require('path'); const {

app, BrowserWindow, clipboard, globalShortcut, Menu,

Tray, systemPreferences,

} = require('electron');

const clippings = []; let tray = null;

let browserWindow = null;

const getIcon = () => {

if (process.platform === 'win32') return 'icon-light@2x.ico'; if (systemPreferences.isDarkMode()) return 'icon-light.png'; return 'icon-dark.png';

};

app.on('ready', () => {

if (app.dock) app.dock.hide();

tray = new Tray(path.join( dirname, getIcon())); **tray.setPressedImage(path.join(\_\_dirname, 'icon-light.png'));**

if (process.platform === 'win32') { tray.on('click', tray.popUpContextMenu);

}

**browserWindow = new BrowserWindow({ show: false,**

**});**

**browserWindow.loadURL(`file://${\_\_dirname}/index.html`);** const activationShortcut = globalShortcut.register(

'CommandOrControl+Option+C', () => {

tray.popUpContextMenu();

},

);

***完成的代码* 179**

if (!activationShortcut)

console.error('Global activation shortcut failed to regiester');

const newClippingShortcut = globalShortcut.register( 'CommandOrControl+Shift+Option+C',

() => {

const clipping = addClipping(); if (clipping) {

browserWindow.webContents.send( 'show-notification', 'Clipping Added',

clipping,

);

}

},

);

if (!newClippingShortcut)

console.error('Global new clipping shortcut failed to regiester'); updateMenu();

tray.setToolTip('Clipmaster');

});

const updateMenu = () => {

const menu = Menu.buildFromTemplate([

{

label: 'Create New Clipping', click() {

addClipping();

},

accelerator: 'CommandOrControl+Shift+C',

},

{ type: 'separator' },

...clippings.slice(0, 10).map(createClippingMenuItem),

{ type: 'separator' },

{

label: 'Quit',

click() {

app.quit();

},

accelerator: 'CommandOrControl+Q',

},

]);

tray.setContextMenu(menu);

};

const addClipping = () => {

const clipping = clipboard.readText(); if (clippings.includes(clipping)) return; clippings.unshift(clipping); updateMenu();

return clipping;

};

const createClippingMenuItem = (clipping, index) => { return {

label: clipping.length > 20 ? clipping.slice(0, 20) + '…' : clipping, click() {

clipboard.writeText(clipping);

},

accelerator: `CommandOrControl+${index}`,

};

};

**清单9.19 Clipmaster完成的渲染器进程：./ app/renderer.js**

const { ipcRenderer } = require('electron');

ipcRenderer.on('show-notification', (event, title, body, onClick = () => { })

=> {

const myNotification = new Notification(title, { body });

myNotification.onclick = onClick;

});

### *本章小结*

* 剪贴板模块提供了几种从剪贴板读取和写入内容的方法。
* globalShortcut模块允许Electron应用程序为键盘快捷键注册侦听器。
* 渲染器进程可以用作后台线程，并不总是需要显示。
* Chromium的Notification API允许我们在macOS和Windows 10上触发本机通知。

*使用菜单栏库构建应用程序*

***本章包括***

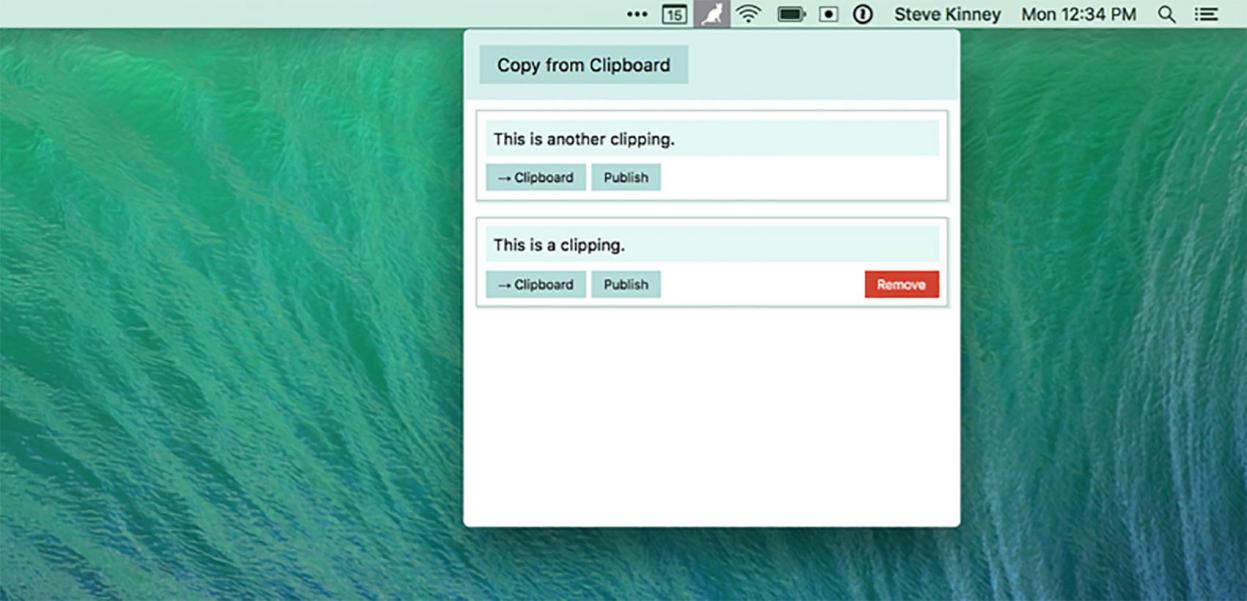
* 使用菜单栏库创建应用程序
* 使用请求库发送HTTP请求
* 创建可点击的通知
* 将辅助菜单添加到托盘实例

在上一章中，我们创建了一个应用程序，它位于macOS上的菜单栏或Windows上的系统托盘中。开箱即用，Electron的托盘模块允许您设置菜单，以便在用户单击托盘图标时显示。这与我们在Fire Sale中构建的应用程序和上下文菜单中使用的菜单类型相同。此菜单也具有相同的限制：它仅限于文本，难以修改，并提供有限的功能。

能够构建系统托盘或菜单栏中的应用程序使我们能够构建我们无法在浏览器中构建的整个应用程序类。不幸的是，我们有这些限制，但幸运的是，我们可以解决它们。在本章中，我们将探索一种巧妙的方法，使用第三方库menubar方便地绕过tray模块的限制。

menubar是一个基于我们之前在本书中使用过的一组核心Electron模块的抽象。一个高级别的解释是它创建了一个空的托盘模块。当用户单击该图标时，菜单栏会在图标下方显示无框架，正确定位的BrowserWindow实例，从而产生附加到图标的错觉。菜单栏还默认提供可爱的猫咪图标。

在本章中，我们将从一开始就将Clipmaster重建为一个全新的应用程序。这次它有一个更舒适的UI，如图10.1所示，与其前身相比。我们添加了从列表中删除剪辑或使用示例API将其发布到Web的功能，以便可以公开共享。最后，我们为通知和全局快捷键添加了一些交互性，以使用预定的按键触发应用程序的功能。



**图10.1 这是应用程序在本章末尾的样子。**

### *启动带菜单栏应用程序*

menubar库提供了一个允许您创建新菜单栏应用程序的功能。它使用app模块来控制应用程序的生命周期，使用托盘模块中的实例在操作系统的菜单栏或系统托盘中创建图标，使用BrowserWindow实例来显示UI（如图10.2所示）。菜单栏使用另一个名为electron-positioner的第三方库来正确定位BrowserWindow实例在图标下。它还提供了以编程方式隐藏和显示窗口的方法。

Github上提供了Clipmaster 9000的样板（https://github.com/electron-in-action/clipmaster-9000）。您可以从主分支开始跟进代码，或者检出completed-example分支，以查看本章末尾的状态代码。

**清单10.1 menubar入门：./app/main.js**

const Menubar = require('menubar'); const menubar = Menubar();

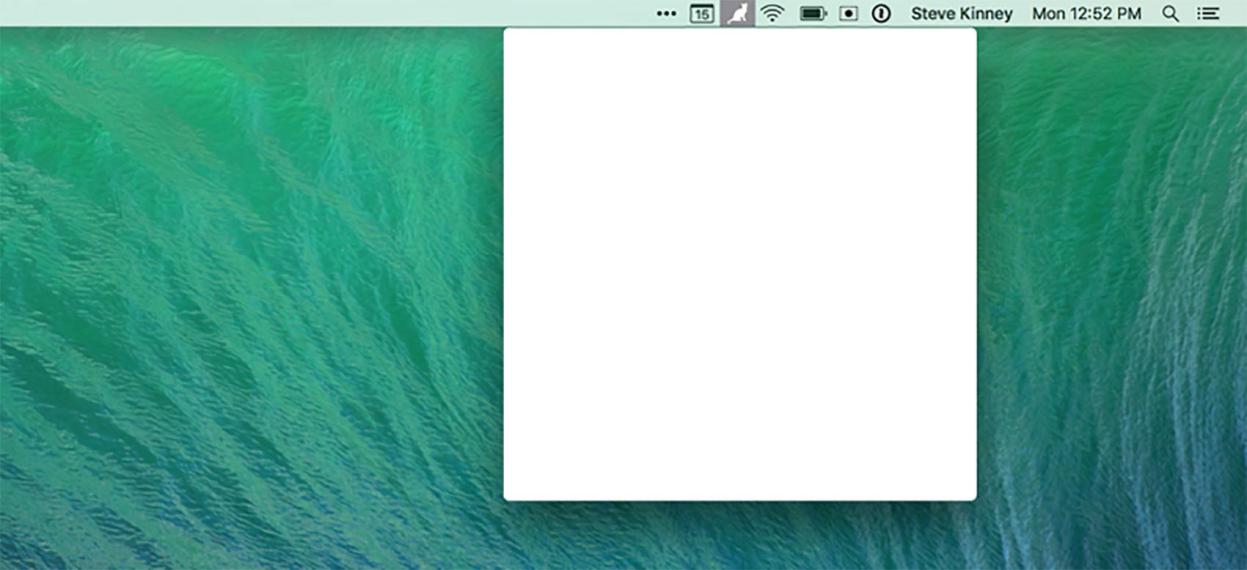
menubar.on('ready', () => { console.log('Application is ready.');

});

**我们创建了一个菜单栏实例，而不是要求Electron提供app模块。**

菜单栏包含几个常见的Electron模块。当app模块触发ready事件时，它会触发ready事件。

在清单10.1中，我们不需要Electron的app模块。当我们调用函数并创建实例时，menubar会为我们做这件事。你还可以看到我们在菜单上而不是应用程序上监听就绪事件。除了监听应用程序的ready事件之外，menubar的活动正在等待其他所有设置。



**图10.2 默认情况下，菜单栏创建一个浏览器窗口，但不会将HTML文档加载到其中。**

您可以使用npm start命令启动简单应用程序。如果一切顺利，您应该在终端中看到一条消息以及菜单栏或系统托盘中的小猫，具体取决于运行应用程序的平台。如果单击该图标，则会看到一个空的浏览器窗口。menubar代表我们创建了一个BrowserWindow实例，但它没有将HTML文档加载到窗口中。

要在该窗口中获取UI，我们需要执行清单10.2中显示的一些操作。首先，我们需要创建一个带有一些基本标记的HTML文档。其次，我们需要定义一个CSS来设置UI的样式。第三，我们需要HTML文档从renderer.js加载我们的UI代码。

**清单10.2 UI的标记：./app/index.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1">

<meta http-equiv="Content-Security-Policy" content="

default-src 'self';

script-src 'self' 'unsafe-inline'; connect-src https://cliphub.glitch.com/\*

"

>

<!--

Change the URL in the line above if you fork the back end server.

-->

<title>Clipmaster 9000</title>

<link rel="stylesheet" href="style.css" type="text/css">

</head>

<body>

<div class="container">

<section class="controls">

**此元素包含Copy from Clipboard按钮，如图10.1所示。**

<button id="copy-from-clipboard">Copy from Clipboard</button>

</section>

<section class="content">

<div id="clippings-list"></div>

</section>

</div>

<script>

require('./renderer');

**此元素包含用户使用应用程序保存的所有剪辑。**

</script>

</body>

</html>

**需要JavaScript用于渲染器进程。**

我在样本库中包含了样式表，但接下来让我们重点介绍一些有趣的内容。我在CSS中使用一种或两种特定于Electron的技术，使应用程序更具原生感; renderer.js开始时是完全空的，但是随着章节的继续，我们会添加它。

**清单10.3 用户界面的样式：./app/style.css**

// …Omitted for brevity…

body > div { height: 100%; overflow: scroll;

-webkit-overflow-scrolling: touch;

}

**为了让Clipmaster 9000感受到macOS上的本机应用程序，我们打开动量滚动。您可以在CSS Tricks（http://mng.bz/i82A）上阅读更多相关信息。**

.container { position: absolute; top: 0;

bottom: 0;

left: 0;

right: 0; overflow: auto;

}

textarea, input, div, button { outline: none; }

// …Omitted for brevity…

.clipping-text::-webkit-scrollbar { display: none;

}

.clipping-controls { margin-top: 0.5em;

**我们不希望显示可能出现在剪辑阴影顶部的滚动条。**

}

// …Omitted for brevity…

有了这些文件，我们现在有了构建应用程序的基础。菜单栏不会立即创建窗口。相反，它会在用户第一次单击图标并触发窗口时创建BrowserWindow实例。当我们尝试在加载DOM之前操作DOM时，这个行为又回到了咬我们，但是现在让我们通过监听在窗口创建之后触发的事件并随后加载 新创建的窗口中的HTML。

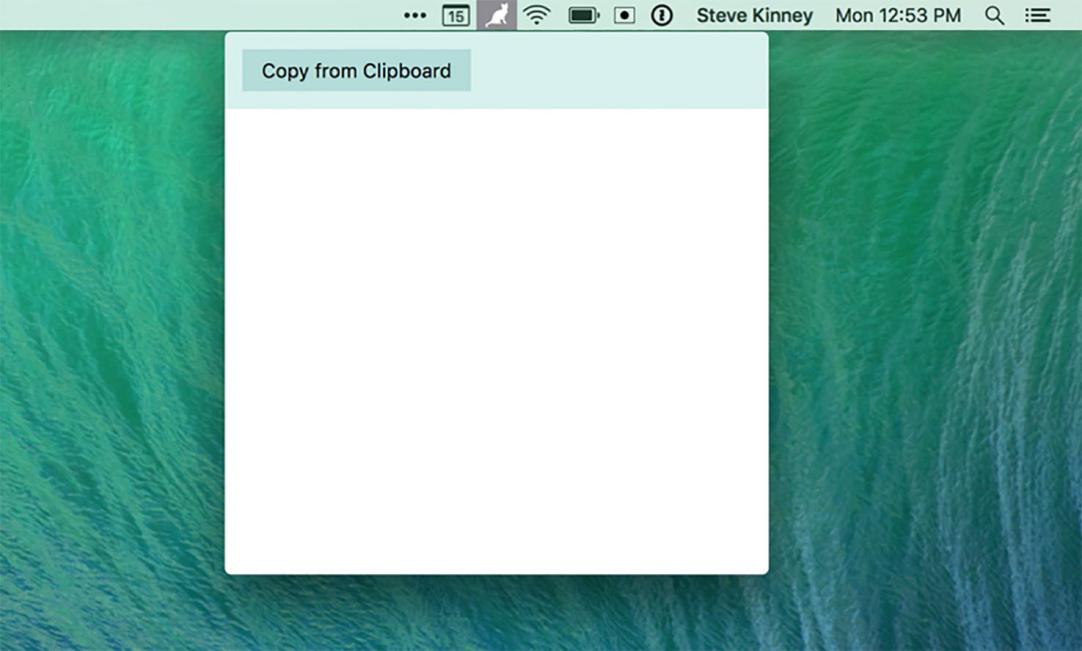
**清单10.4 在创建窗口后加载HTML页面：./app/main.js**

menubar.on('after-create-window', () => { menubar.window.loadURL(`file://${ dirname}/index.html`);

});

**创建窗口后会触发after-create-window事件。默认情况下，菜单栏在首次加载时不会创建窗口。它会等到菜单栏或托盘图标被点击并需要显示一个窗口。**

您在此阶段的应用应如图10.3所示。此时，该按钮不起作用，因为我们没有为前端编写任何JavaScript。



**图10.3 Clipmaster 9000的基本UI已就绪。**

### *将剪辑添加到UI*

我们的应用程序有一个UI，但它还没有做任何事情。就像在第9章中一样，我们首先要允许用户向应用程序添加剪辑。这比前一章稍微复杂一些，因为我们必须为剪辑生成DOM节点。我们需要将这个新节点添加到我们的标签中包含剪辑列表的节点。最后，我们需要向按钮添加一个事件监听器来调用我们刚才描述的两个方法。让我们从查看和缓存我们经常使用的两个选择器开始。

**清单10.5 查询和缓存常用的选择器：./app/renderer.js**

const clippingsList = document.getElementById('clippings-list'); const copyFromClipboardButton = document.getElementById(

'copy-from-clipboard'

);

创建元素很简单，只有一个重要的元素。在清单10.6中，我们创建了一个article元素并将.clippings-list-item类添加到它，以便它的样式适当。接下来，我们设置其内部内容。我们查询负责显示剪辑文本的节点并相应地设置其内部文本。最后，我们返回新元素，以便将其添加到DOM中

***将剪切添加到UI* 187**

**清单10.6 根据剪辑文本为UI创建元素：./app/renderer.js**

const createClippingElement = (clippingText) => {

const clippingElement = document.createElement('article'); clippingElement.classList.add('clippings-list-item');

clippingElement.innerHTML = `

<div class="clipping-text" disabled="true"></div>

<div class="clipping-controls">

<button class="copy-clipping">&rarr; Clipboard</button>

<button class="publish-clipping">Publish</button>

<button class="remove-clipping">Remove</button>

</div>

`;

**为剪辑创建一个新元素**

设置新元素的内部HTML

clippingElement.querySelector('.clipping-text').innerText = clippingText;

return clippingElement;

};

**返回新元素**

**查找剪切文本应该到达的节点，并将其内容设置为剪辑的文本**

您可能想知道为什么在设置元素的内部HTML时我没有使用模板文字插值。我们没有使用innerHTML设置新元素的内容，因为它不会转义输入并将其呈现为HTML。如果用户将HTML复制到剪贴板，它将呈现标记，这不是我们想要的。相反，我们使用innerText来设置该节点的内容，该节点将转义任何HTML并按照用户的预期呈现它。

我们现在可以获取任意文本字符串并返回UI所需的元素。下一步是让文本提供此功能，获取结果，并将其添加到页面。

**清单10.7 从剪贴板中读取并向列表添加剪辑：./app/renderer.js**



const { clipboard } = require('electron');

const addClippingToList = () => {

const clippingText = clipboard.readText();

需要 **Electron的clipboard模块**

使用Electron的剪贴板模块从剪贴板中读取文本

const clippingElement = createClippingElement(clippingText);

创建一个DOM节点在UI中显示剪辑

clippingsList.prepend(clippingElement);

};

**将其添加到UI中剪辑列表的顶部**

copyFromClipboardButton.addEventListener('click', addClippingToList);

每当用户单击UI中的Copy from Clipboard按钮时，都会触发addClippingToList( )。

从系统的剪贴板中读取就像要求Electron的剪贴板模块并调用其readText( )方法一样简单。使用剪辑文本，我们创建元素，然后将其添加到剪辑列表的顶部。最后，我们向Copy from Clipboard 按钮添加一个事件监听器，以触发整个过程。

### *在应用程序中处理剪辑*

用户现在可以将剪辑保存到Clipmaster 9000，但这只是成功的一半。如果他们想将剪辑的内容写回剪贴板怎么办？ 从技术上讲，因为这个新版本的应用程序有一个UI，他们可以选择文本并再次复制，但我们可以做得更好。此外，Clipmaster 9000将改进旧版本，允许用户删除他们不再想要存储的剪辑，并将其发布到网络上，以便在需要时公开分享。

###### *使用事件代理防止内存泄漏*

实现上一节中描述的功能的最明显的方法是向每个调用适当函数的按钮添加一个事件监听器。使用createClippingElement( )为每个剪辑创建新元素时，我们可以添加这些事件监听器。

这里的问题是，如果用户删除剪辑，我们还需要删除这些事件监听器。如果不这样做会导致内存泄漏，因为事件监听器和DOM元素仍然会相互引用。利用事件冒泡DOM的事实要容易得多。

如果单击某个按钮，浏览器将检查该元素是否包含click事件的任何事件监听器。接下来，它会检查该元素的父元素。它将继续这个过程，直到它到达DOM树的顶部。通过向clippingsList添加事件监听器，我们可以捕获源自特定剪辑的事件。因为列表本身永远不会从DOM中删除，所以无论何时从UI中删除剪辑，我们都不必担心删除事件监听器。

DOM中的所有事件对象都带有一个target属性，该属性包含对触发事件的元素的引用。我们看看这个，找出哪个按钮被点击了哪个剪辑，然后采取适当的行动。

**清单10.8 为每个剪辑的按钮设置一个事件监听器: ./app/renderer.js**

将事件侦听器添加到剪辑列表中。单个剪辑中的单击事件会在列表中冒泡。

clippingsList.addEventListener('click', (event) => {

const hasClass = className => event.target.classList.contains(className);

**创建一个辅助函数，用于确定目标元素是否具有给定的类。**

if (hasClass('remove-clipping')) console.log('Remove clipping'); if (hasClass('copy-clipping')) console.log('Copy clipping');

if (hasClass('publish-clipping')) console.log('Publish clipping');

});

**现在，我们将确定单击了哪种按钮并将适当的消息记录到控制台。随着章节的进展，将添加该功能。**

***在应用程序中处理剪切* 189**

我们检查三个不同类的元素，因此创建一个名为hasClass( )的小辅助方法来帮助这个过程是有意义的。基于类，我们需要调用不同的函数。我们还没有编写这些函数，因此我们现在只需登录控制台即可确认一切正常。

###### *删除剪辑*

要删除剪辑，我们向上导航到其祖父母，这是整个剪辑的元素。

**Listing 10.9 从DOM中删除剪辑: ./app/renderer.js**

const removeClipping = (target) => {

target.parentNode.parentNode.remove();

};

**从DOM中删除整个剪辑。这也有效地将其从内存中删除。**

当我们引用剪辑元素时，从页面中删除它就像调用元素的remove( )方法一样简单。

**清单10.10 配置事件监听器以删除剪辑：./app/renderer.js**

clippingsList.addEventListener('click', (event) => {

const hasClass = className => event.target.classList.contains(className);

if (hasClass('remove-clipping')) **removeClipping(event.target);** if (hasClass('copy-clipping')) console.log('Copy clipping');

if (hasClass('publish-clipping')) console.log('Publish clipping');

});

**使用新函数替换日志以删除剪辑。**

每当用户单击剪辑的Remove按钮时，我们都会调用removeClipping( )并向其传递对该按钮的引用，而不是记录到控制台。这有效，但我们可以做得更好一点。展望未来，我们可以安全地假设我们需要获得对clip元素或其文本的引用。将它们引入自己的功能是有意义的。

**清单10.11 设置使用剪辑的辅助方法: ./app/renderer.js**

const getButtonParent = ({ target }) => {

return target.parentNode.parentNode;

};

const getClippingText = (clippingListItem) => {

**创建抽象以导航到包含整个剪辑的DOM节点。**

return clippingListItem.querySelector('.clipping-text').innerText;

}; **创建一个抽象，用于遍历剪辑并查找最初由用户保存的文本。**

getButtonParent( )可以从三个按钮中的任何一个导航到父级。虽然简单，但在我们更新标记的情况下，此功能非常有用。您不希望必须更改用于将DOM从按钮遍历到剪切元素三次的代码。从父级，我们需要获取剪辑的文本，因为我们有效地使用DOM作为我们的数据存储。幸运的是，遍历DOM比遍历更容易，getClippingText( )可以利用querySelector( )方法。我们现在可以在事件监听器中使用此方法。

**清单10.12 配置事件监听器以显示剪辑的文本：./app/renderer.js**

clippingsList.addEventListener('click', (event) => {

const hasClass = className =>

event.target.classList.contains(className); **const clippingListItem = getButtonParent(event);**

**获取包含DOM节点的开头而不是每个函数**

if (hasClass('remove-clipping')) removeClipping(**clippingListItem**); if (hasClass('copy-clipping')) console.log('Copy clipping',

➥ **getClippingText(clippingListItem)**);

if (hasClass('publish-clipping')) console.log('Publish clipping',

➥ **getClippingText(clippingListItem)**);;

});

**将clip元素传递给removeClipping( )而不是事件**

在上一个清单中，我们立即获得了对clip元素的引用，因为我们在每种情况下都需要它。现在获取剪辑文本很简单，让我们更新我们的控制台日志以包含剪辑文本以验证我们是否正确实现了这个功能。removeClipping( )现在也可以变得更简单。可以说我们根本不需要这个功能，但可以想象我们可能希望以后添加更多功能。因此，暂时离开它是有道理的。

**清单10.13 重构removeClipping( )函数：./app/renderer.js**

const removeClipping = (target) => {

target.remove();

};

###### *写入剪贴板*

**重构removeClipping( )函数以使用元素而不是事件的目标节点。**

我们不仅在前一章中实现了写入剪贴板的功能，我们已经在本章中奠定了很多基础，以便轻松实现此功能。我们需要的第一件事是一个处理将剪辑文本写入剪贴板的函数。

**清单10.14 创建一个向剪贴板写剪辑的函数：./app/renderer.js**

const writeToClipboard = (clippingText) => {

clipboard.writeText(clippingText);

};

**创建一个负责将文本写入剪贴板的函数。现在这很简单，但我们将在本章后面添加更多内容。**

我们将在本章后面添加更多内容，因此将它保留为函数是有意义的，而不是仅仅在事件监听器中添加内联。下一步是使用此新函数替换事件监听器中的控制台日志，并将函数传递给用户选择的剪辑文本。

***发布剪辑* 191**

**清单10.15 将writeToClipboard( )添加到事件监听器：./app/renderer.js**

clippingsList.addEventListener('click', event => {

const hasClass = className => event.target.classList.contains(className);

const clippingListItem = getButtonParent(event);

if (hasClass('remove-clipping')) removeClipping(clippingListItem); if (hasClass('copy-clipping'))

**writeToClipboard(getClippingText(clippingListItem)**); if (hasClass('publish-clipping'))

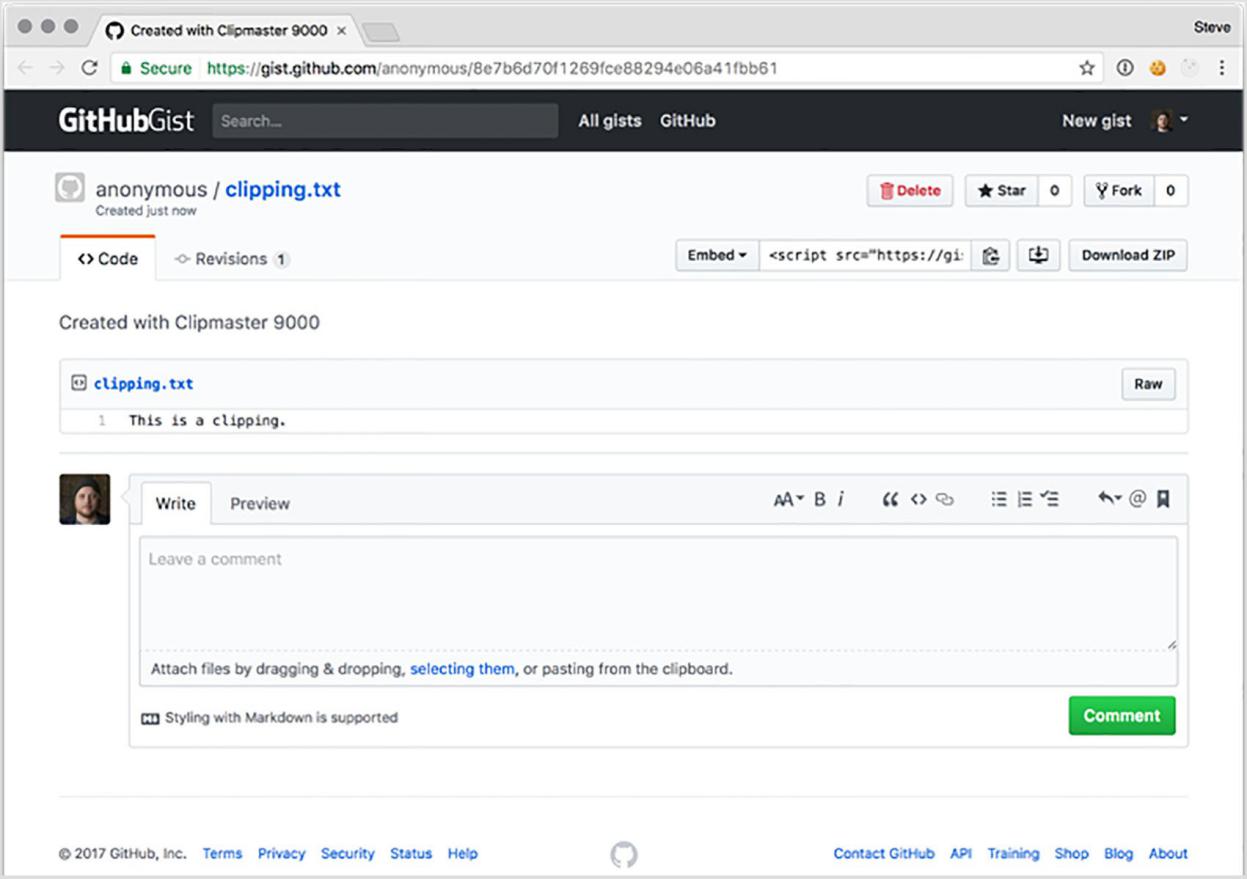
console.log('Publish Clipping', getClippingText(clippingListItem));

**设置Write to Clipboard按钮以调用我们的新功能**

});

### *10.4 发布剪辑*

我们现在拥有核心功能，任何自称为Clipmaster 9000的自尊应用程序都需要完成它的工作。是时候开始申请额外学分了。让我们实现将剪辑发布到示例API的功能，如图10.4所示。



**图10.4 ClipHub服务上发布的剪辑**

出于我们在第2章中讨论的相同安全原因，我们无法在浏览器中执行此操作。此外，我们使用通常无法在浏览器中工作的Node库将我们的请求发送到API。

在焦点的名义，我们将采取一些快捷方式。这是一个审慎的简单API,它将剪辑存储在内存中。所有剪辑都会定期清除。可以在https://glitch.com/~cliphub找到服务器的源代码。

###### *10.4.1 设置请求*

request是另一个流行的，名称很好的库，可以轻松地对远程服务器执行HTTP请求。请求允许我们为它所做的每个请求设置默认值。对于此应用程序，我们将所有请求发送到同一API端点，因此将其设置为默认值是有意义的。我们还将创建一个自定义用户代理字符串，以区分Clipmaster和普通浏览器。

在本节中，我们设置了对ClipHub API发出请求，格式化我们的剪辑，以便ClipHub API接受这些请求，然后设置我们的UI来发出请求。

**清单10.16 要求和配置请求的默认参数：./app/renderer.js**

const request = require('request').defaults({ url: 'https://cliphub.glitch.me/clippings', headers: { 'User-Agent': 'Clipmaster 9000' }, json: true,

**request允许您为每个HTTP请求设置默认参数**

通常，浏览器设置用户代理字符串。因为我们使用的是库，所以我们可以设置自己的用户代理。

});

**设置为true时，request将代表我们自动解析和串化JSON。**

现在我们可以格式化API剪辑的文本并向API发送HTTP请求。下一步是将这两个函数绑定在一起，向用户显示请求是否成功，以及ClipHub上剪辑的URL是否成功。

**清单10.17 创建一个向ClipHub的API发布剪辑的函数：./ app/renderer.js**

格式化剪辑文本并通过POST请求将其发送到ClipHub API。

const publishClipping = (clipping) => {

**// 译注：原代码的下面这一行错误，在｝｝后面多了一个）右括号，导致功能失败！**

request.post({ json: { clipping } }**~~)~~**, (error, response, body) => {

if (error) { return alert(JSON.parse(error).message); }

const url = body.url;

alert(url); clipboard.writeText(url);

**查找已发布剪辑的URL。**

通过alert向用户显示url



**如果由于某种原因出现错误，我们会向用户显示警告。**

});

};

**将其写入剪贴板。**

request.post( )向URL发送POST请求。我们之前设置了默认URL，因此无需立即指定。request.post( )有两个参数：我们要发送的数据和当我们从服务器收到回复时调用的回调。请求传递回调的三个参数：请求不成功时的错误对象，完整的HTTP响应以及响应的主体。如果请求成功，则error为null。

如果请求成功，我们将获取url属性，该属性包含新发布剪辑的URL。目前，我们使用alert来显示网址。我们还将其写入剪贴板，以便用户可以将其粘贴到他们喜欢的Web浏览器的地址栏中。使用此功能的代码，最后要做的是在用户单击剪辑上的Publish按钮时调用它。

**清单10.18 将publishClipping( )添加到事件监听器：./ app/renderer.js**

clippingsList.addEventListener('click', (event) => {

const hasClass = className => event.target.classList.contains(className); const clippingListItem = getButtonParent(event);

if (hasClass('remove-clipping')) removeClipping(clippingListItem);

if (hasClass('copy-clipping'))

writeToClipboard(getClippingText(clippingListItem)); if (hasClass('publish-clipping'))

**publishClipping(getClippingText(clippingListItem)**);

**正确设置Publish按钮。**

});

在这个清单中，我们使用了一种类似于将剪辑的文本写入剪贴板的技术。唯一的区别是我们换掉writeToClipboard( )而转而使用publishClipping( )。

### *10.5 显示通知和注册全局快捷键*

我们让用户知道Clipmaster以各种方式发生了一些事情。我们默默地写入剪贴板，而不通知用户该操作是否成功结束。发布时，我们使用一个alert来锁定应用程序，直到用户解除它。

在本章的后面部分，我们实现了全局快捷键，允许用户在不打开应用程序的情况下触发应用程序的功能。在这种情况下，拥有有用的通知更为重要。显示这些通知很简单，因为我们在渲染器进程中完成了所有繁重工作，并且不必担心IPC。考虑到这一点，我们将研究如何在通知中添加事件处理程序，以添加上一章中未提供的功能。

让我们从处理用户发布剪辑时弹出的警报开始。取而代之的是，我们显示了两个通知中的一个，如清单10.19所示：从服务器收到的消息发生错误，或者请求成功使用ClipHub上新发布的剪辑的URL。如果请求成功，我们会在用户单击通知时在其默认浏览器中打开ClipHub的通知中添加事件处理程序。

**清单10.19 发布剪辑时设置通知：./ app/renderer.js**

const { clipboard, **shell** } = require('electron');

// Code omitted for clarity…

const publishClipping = (clippingText) => {

**// 译注：原代码的下面这一行错误，在｝｝后面多了一个）右括号，导致功能失败！**

**如果出现错误，则创建通知**

request.post({ json: { clipping **} }**, (error, response, body) => {

if (**error**) {

return new Notification('Error Publishing Your Clipping', { body: JSON.parse(error).message

});

}

const url = body.url;

**const notification = new Notification( 'Your Clipping Has Been Published',**

**{ body: `Click to open ${url} in your browser.` }**

**);**

**成功发布剪辑时创建通知**

**notification.onclick = () => { shell.openExternal(url); };**

clipboard.writeText(url);

};

};

**如果用户单击通知，则将其带到已发布的剪辑。**

**译注：译者在Windows10上验证，通知功能是无效的！**

要在用户的默认浏览器中打开URL，我们需要从Electron中引入shell模块。我们将通知的onclick方法设置为在用户单击通知时触发的匿名函数。

###### *注册全局快捷键*

要求用户将手从键盘上移开并导航到菜单栏或系统托盘中的小图标并不总是最佳的。他们可能在想要创建剪辑时键入内容，并且他们希望有一个键组合的便利性，他们可以从操作系统的任何位置按下以触发Clipmaster 9000内的命令。

如果您还记得上一章，我们必须在主进程中注册全局快捷键。在第9章中，我们在主进程中实现了大部分应用程序的功能，我们可以直接调用函数。在本章中，情况恰恰相反，我们需要建立一些IPC。在下面的清单中，我们首先注册一个快捷键来创建一个新剪辑。我们现在开始登录控制台。在下一步中，我们实现其功能。

**清单10.20 设置一个简单的全局快捷键：./ app/main.js**

const { **globalShortcut** } = require('electron'); const Menubar = require('menubar');

const menubar = Menubar();

menubar.on('ready', function() {

console.log('Application is ready.');

**需要Electron的globalShortcut模块**

**const createClipping = globalShortcut.register('CommandOrControl+!', ()**

**➥ => {**

**console.log('This will eventually trigger creating a new clipping.');**

**});**

**if (!createClipping) {**

**console.error('Registration failed', 'createClipping');**

**}**

});

menubar.on('after-create-window', () => { menubar.window.loadURL(`file://${ dirname}/index.html`);

});

**最终创建新函数的全局快捷键**

**如果注册全局快捷键失败，将错误记录到控制台**

这与我们在前一章中所做的类似。重要的区别是我们在渲染器进程中将所有剪辑存储在DOM中。因此，我们需要与渲染器进程通信以创建剪辑。在Fire Sale应用中，我们保留了对我们创建的每个窗口的引用。menubar代表我们创建了一个浏览器窗口并将其存储在其window属性中。让我们更新我们的全局快捷键，只要用户按下击键创建新的快捷键，就会向渲染器进程发送消息。我们还可以注册用于将剪辑写回剪贴板的快捷键，并将它们发布到ClipHub。

**清单10.21 使用IPC设置全局快捷键：./ app/main.js**

const createClipping = globalShortcut.register('CommandOrControl+!', () => {

menubar.window.webContents.send('create-new-clipping');

});

const writeClipping = globalShortcut.register('CmdOrCtrl+Alt+@', () => {

menubar.window.webContents.send('write-to-clipboard');

});

const publishClipping = globalShortcut.register('CmdOrCtrl+Alt+#', () => {

menubar.window.webContents.send('publish-clipping');

});

if (!createClipping) {

console.error('Registration failed', 'createClipping');

}

if (!writeClipping) {

console.error('Registration failed', 'writeClipping');

}

if (!publishClipping) {

console.error('Registration failed', 'publishClipping');

}

每个快捷键都是在不同的通道上发送消息。在这种情况下，我们不向渲染器进程发送任何其他信息，因为该进程从剪贴板中读取。下一步是配置渲染器进程以接收从主进程发送的消息。用户按下按键时可能没有打开应用程序，因此我们会在适当的位置添加通知。

**清单10.22 在渲染器进程中设置IPC监听器: ./app/renderer.js**

添加剪辑后，显示通知，因为用户没有打开UI，可能不知道它已成功添加。

从Electron中提取ipcRenderer模块

const { clipboard, **ipcRenderer**, shell } = require('electron');

ipcRenderer.on('create-new-clipping', () => {

addClippingToList();

new Notification('Clipping Added', { body: `${clipboard.readText()}`

});

});

ipcRenderer.on('write-to-clipboard', () => {

const clipping = clippingsList.firstChild; writeToClipboard(getClippingText(clipping));

});

ipcRenderer.on('publish-clipping', () => {

const clipping = clippingsList.firstChild; publishClipping(getClippingText(clipping));

});

**如果在创建新剪辑通道上遇到IPC消息，则调用用户单击Copy to Clipboard按钮时调用的函数。**

以前，我们根据用户点击的按钮知道用户想要的剪辑。在这种情况下，我们不知道，所以我们取列表中的第一个。

**将剪辑写入剪贴板。**

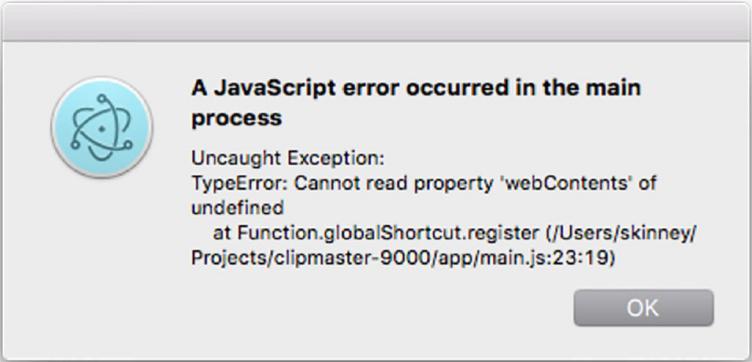
发布剪辑的过程类似于将剪辑写入剪贴板：查找第一个剪辑并将其传递给函数。

在第9章中，我们将所有剪辑存储在主进程中的数组中。在本章中，我们将DOM用作临时数据存储。如果用户单击按钮以写入剪贴板或发布剪辑，我们将根据单击的按钮知道哪个剪辑。但是，当用户激活全局快捷键时，我们如何找到合适的剪辑？ 在上一个清单中，我们遍历剪切列表的第一个子节点，并调用我们在本章前面用该元素编写的函数之一。

###### *解决从未显示过窗口时出现的边缘情况*

实现最后一个功能时，您可能已经注意到了一个错误。如果您启动应用程序并立即按下其中一个新的全局快捷键，则会出现错误，如图10.5所示，其中显示“TypeError: Cannot read property 'webContents' of undefined.”如果您查看上一个列表，那么' 请注意我们正在尝试访问menubar.window上的webContents属性，这显然是未定义的。单击猫图标以打开窗口并再次尝试快捷方式。它这次应该工作。

***添加二级菜单* 197**



**图10.5如果您在第一次打开窗口之前尝试使用其中一个全局快捷键，则会看到此错误。译注：在 linux下也可以见到**

那它为什么突然起作用呢？ 菜单栏在第一次需要时懒洋洋地加载窗口。为防止出现此错误，我们必须告诉菜单栏在启动时立即加载窗口。我们还希望它加载我们的HTML页面，它反过来加载renderer.js，它设置我们的IPC监听器。

**清单10.23 启动时预加载窗口和内容：./ app/main.js**

const menubar = Menubar({ **preloadWindow**: true,

index: `file://${ dirname}/index.html`,

});

**将preloadWindow选项设置为true以加载UI，即使从未请求过UI也是如此。**

指定应预加载的HTML文档。

以前，我们在没有参数的情况下调用Menubar( )函数。在清单10.23中，我们修改了函数调用并传入了一个配置对象。我们在用户第一次点击菜单栏或托盘图标之前预加载窗口，我们会在菜单栏加载窗口时加载到HTML页面。

### *添加二级菜单*

在Fire Sale中，我吹嘘说在Electron中构建应用程序的一个好处是我们可以使用应用程序和上下文菜单来提供功能，而无需在UI中找到它的位置。在前一章中，如有必要，我们可以创建其他子菜单。但是Clipmaster 9000怎么样？ 事实证明，托盘模块允许我们在用户右键单击图标时显示辅助菜单。让我们创建一个简单的菜单，允许用户退出应用程序。我们分三步完成：使用Menu.buildFromTemplate( )创建菜单，为menubar.tray右键单击添加事件处理程序，并在用户右键单击图标时弹出菜单。

**清单10.24 创建二级菜单：./ app/main.js**

const { globalShortcut, **Menu** } = require('electron');

const secondaryMenu = Menu.buildFromTemplate([

{

**需要Electron的菜单模块。**

label: 'Quit',

click() { menubar.app.quit(); }, accelerator: 'CommandOrControl+Q'

},

]);

**从JavaScript对象构建菜单。**

获取对app模块的引用，并告诉它退出**.**

menubar.on('ready', function () {

console.log('Application is ready.');

menubar.tray.on('right-click', () => { menubar.tray.popUpContextMenu(secondaryMenu);

});

// Omitted for brevity…

});

**在菜单栏或系统托盘图标上监听右键单击事件。**

触发弹出菜单。

**译注：在Deepin中右键退出功能是无效的，因为根本看不见菜单！**

通过这一小改动，我们为应用程序创建了一个完整的辅助接口。这将是用户首选项和其他高级选项的好地方，这些选项在应用程序的主UI中没有好的位置。

Clipmaster 9000的第一阶段现已完成。您可以在附录中或此存储库的completed-example分支（http://mng.bz/UE9）中找到完整的代码。

译注1:

menubar库在github上的地址:<https://github.com/maxogden/menubar>

译注2:

关于5.0.x版本下,需要注意的地方:

**A.升级到最新版:**

electron: 1.8.2 ==> ^5.0.4";

menubar:5.2.3 ==> ^6.0.5;

request:2.83 ==> ^2.88.0";

eslint:4.17.0 ==> ^5.16.0;

eslint-plugin-import: 2.8.0 ==> ^2.17.3

**B.menubar引入和实例化的变化为:**

const menubar = require('menubar');

const mb = menubar.menubar({

browserWindow: {

webPreferences: {

nodeIntegration: true // 一定要配置,否则报require未定义错误

}

}

});// 这一点github官网上的menubar也没有及时更新!

**C.新版的menubar中，index.html文件必须在项目的根目录中存在，否则运行报错提示找不到文件！**

**D.说明：main.js中的**

mb.on('after-create-window', () => {

// mb.window.loadURL(`file://${\_\_dirname}/index2.html`); 这一行不是必须的了，如果存在反而会替换index.html的内容

});

**E.Deepin下,功能可以工作,但是点击图标会产生空标签的二级菜单,点击二级菜单才会显示index.html的内容.**

**F.Windows10下要想显示通知需要app.setAppUsesrModelId(),详细参见:https://electronjs.org/docs/tutorial/notifications**

### *本章小结*

* + - menubar是第三方库，是围绕核心Electron模块和另一个称为electron-positioner的第三方库的高级抽象。
    - menubar创建一个浏览器窗口并将其直接放置在菜单栏图标（在macOS上）或托盘图标上方（在Windows上）。
    - 事件委派是一种技术，允许您向父节点添加事件侦听器并等待事件冒泡，而不是手动添加和删除子节点中的侦听器以防止内存泄漏。
    - request是Node.js的一个库，允许您向外部URL发出HTTP请求。
    - 我们可以定义Notifications的onclick方法，以便在用户单击通知时允许自定义行为。
    - 应用程序启动时，可以预加载menubar的浏览器窗口实例。
    - 可以将辅助菜单添加到托盘实例以提供其他功能。

*使用转发器和框架*

***本章包括***

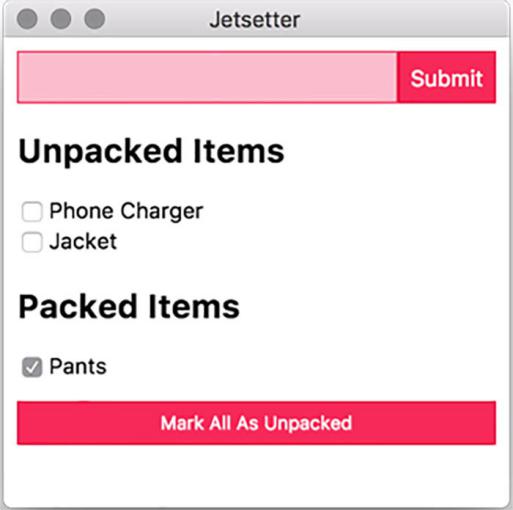
* 设置electron-compile以转换CoffeeScript，TypeScript和即将推出的JavaScript功能（使用Babel），而无需在Electron应用程序中构建步骤。
* 使用electron-compile将Less，Sass和Stylus转换为CSS，将Jade转换为HTML。
* 使用React，Sass和Jade构建一个简单的Electron应用程序。
* 设置实时重载，只要更改了渲染器进程中的代码，就会重新加载浏览器窗口。

在这篇文章的章节中，我们一直使用vanilla JavaScript和CSS编写所有UI。这是故意的,毕竟这是一本关于Electron的书。它不是一本关于客户端框架的书。也就是说，不可忽视的是，现代Web开发人员使用各种语言，框架和工具来构建用户期望从Web获得的越来越复杂的UI。

在本章中，我将向您展示如何设置Electron以使用各种Web语言。有了这个，我们使用三种语言构建一个小应用程序：React（使用JSX和即将支持的Node或Chromium尚未支持的JavaScript功能），Sass和Jade。您并不要求使用React或任何其他语言。事实上，如果你是其中任何一个的专家，你可能会注意到我们故意简单地保留它们并省略PropTypes之类的东西。我之所以选择React，Sass和Jade，是因为它们为不熟悉它们的开发人员带来了受欢迎和相对平易近人的好处。虽然这本书不是关于React，Sass或Jade，但它的主题是Electron实战，所以我们将通过实践来学习。当我们将真实的持久数据存储连接到应用程序时，我们将在下一章中使用此应用程序。如果您已经熟悉React或者没有强烈的学习愿望，可以使用此应用程序的一个分支。也就是说，不要错过第11.5节，其中我将介绍如何使用electron-compile启用实时重载。

在本章中，我们构建了一个名为Jetsetter的应用程序，如图11.1所示。这个应用程序解决了我一直遇到的问题。我最近一直在旅行，在JavaScript会议上发言，唯一比在没有合适的适配器的情况下出现在会场的情况更糟糕的是出现在酒店一周却没有足够的袜子。在过去，我尝试使用待办事项列表来跟踪我需要打包的所有内容。但是待办事项列表应用程序并不是真正为此目的而构建的，而且当我打包下一次旅行时，我发现检查和取消选中所有任务是很繁琐的。因此，Jetsetter诞生了。

您可以在https://github.com/electron-in-action/jetsetter找到Jetsetter的源代码。在我们完成本章的过程中，我们从master分支开始作为应用程序的基础，但您也可以在名为completed-example的分支上找到已完成的版本。首先，克隆代码库，然后安装依赖项。



**图11.1在本章中，我们构建了一个应用程序来跟踪即将到来的行程需要打包的内容。**

我们构建了一个应用程序，允许用户添加项目并将其标记为打包或解压缩。UI底部的按钮允许用户在准备好下次旅行时将所有内容标记为再次解压缩。我们在本章中构建的版本有一个致命的缺陷：它不会在任何地方保存项目列表。这意味着每当您关闭应用程序时，所有新项目都将丢失。第12章通过展示如何以几种不同的方式保存数据来解决这个问题，从浏览器内存储到直接从UI访问数据库。

***介绍electron-compile* 201**

### *介绍electron-compile*

尽管开发人员目前使用所有语言来构建Web应用程序，但浏览器只能理解其中的三种语言：HTML，CSS和JavaScript。多年来一直存在一些小的例外--谷歌的Dartium，它是一个与Dart一起运行的Chroium，并且可以想到Netscape Navigator对JavaScript Stylesheets的短暂支持,但是在大多数情况下，这种情况已经出现了只要网络出现，就一直都是如此。

除了将其他语言编译成JavaScript之外，将JavaScript编译成JavaScript也变得时髦。诸如Babel之类的工具允许开发人员使用最新且最好的,在某些情况下，即将到来的语言功能，然后将它们编译成更旧的浏览器支持的更详细的版本。在构建Electron应用程序时，我们并不关心旧浏览器，因为我们正在发布我们自己的现代版本的Chromium和Node，但Babel插件可用于即将推出的语言功能和替代语法，如JSX或Flow，前者我们在本章中使用。

通常，开发人员使用他们选择的语言编写，然后使用构建工具将其代码编译为浏览器的JavaScript，CSS或HTML。这个过程通常称为转换，因为我们将一种语言编译成另一种语言而不是字节代码或汇编。我们当然可以在本章中这样做，但它需要运行一个单独的过程来编译我们的代码。在刷新Electron应用程序的浏览器窗口以查看更改之前，我们必须等待该过程完成。

更重要的是，Electron应用程序的过程与传统的Web应用程序没有太大差别，当然也不保证本书的整个章节(都是如此)。因此，我非常高兴地向您介绍electron-compile。除了Chromium和Node支持的语言外，electron-compile还使Electron能够使用其他语言，如图11.2所示。它们可以即时编译，无需额外的构建过程。

JavaScript

CSS

HTML

JSON

JavaScript

(Babel)

CoffeeScript TypeScript

GraphQL

Sass

Less

Stylus

Jade

Vue.js 2.0

(single-file components)

CSON

(CoffeeScript)

electron-compile

**图11.2 electron-compile支持多种语言。**

如图11.2所示，electron-compile支持大量(原文意:健康)数量的语言。开始使用electron-compile非常容易。在应用程序的package.json文件中，您将找到重要的依赖electron以及electron-prebuilt-compile，这是一个已经配置了electron-compile的Electron版本。

electron-compile通过查看文件扩展名来确定语言。如果您加载带有.coffee扩展名的文件，它会使用CoffeeScript自动转换它。具有.ts扩展名的TypeScript也是如此，依此类推。所有扩展名为.js的文件都将使用Babel进行转换。默认情况下，Babel不执行任何操作，因此它对您的JavaScript文件没有任何影响，除非您有意将其配置为，我将在下一节中介绍。

现在，如果你运行npm start，应用程序就会按照你的预期启动，但它现在完全支持electron-compile。如果您的需求很简单，那么这就是您必须做的一切。在下一节中，我们将配置electron-compile以支持JSX和其他即将推出的JavaScript功能，并支持开发中的源映射。

### *奠定应用程序的基础*

在本章中，我们使用JavaScript以及一些附加功能并支持JSX-类似HTML的语法，这样可以更轻松地定义React组件。因此，我们需要配置Babel以支持这些功能。当应用程序启动时，electron-compile会查找名为.compilerc的文件，以获取超出默认值的任何其他配置。我已经在本章前面克隆的存储库中为您提供了这个文件，但让我们在这个清单中查看它。

**清单11.1 配置用于electron-compile的Babel：./.compilerc**

{

**我们希望使用的Babel预设列表。预设是一组插件。**

**electron-compile允许您为开发指定不同的配置而不是像生产那样。这是开发的配置.**

"env": { "development": {

"application/javascript": { "presets": [

[

"env",

{

"targets": { "electron": "1.6"

}

}

],

**electron-compile支持各种文件类型。这是适用于JavaScript文件的配置。**

babel-env允许我们根据目标动态选择插件。

Jetsetter使用Electron 1.6。此设置告诉Babel不要转换Electron 1.6已经支持的功能。译注:原书1.8

尽管支持ES2015的大多数，但Node不支持ES模块。 这个插件增加了支持.译注:作者提供的源码中没有这一行

**为Babel加载预设的React插件集合**

"react"

],

"plugins": [

"transform-es2015-classes", "transform-es2015-modules-commonjs", "transform-object-rest-spread"

**Object Rest Spread是React开发人员常用于制作对象副本的第3阶段提案.**

],

"sourceMaps": "inline"

},

**源码映射允许开发人员在他们的开发人员工具中查看未转换的源码。**

"text/jade": {

"pretty": true

}

},

"production": { "application/javascript": {

"presets": [ [

"env",

{

"targets": { "electron": "1.6" 译注:同样为1.6

默认情况下，Jade格式化其输出，没有任何空格或新行。此设置可生成更易读的输出。

**在生产环境中，我们指示electron-compile不包括源maps或格式化Jade的输出。**

}

}

],

"react"

],

"plugins": [

"transform-es2015-classes", "transform-es2015-modules-commonjs", "transform-object-rest-spread"

],

"sourceMaps": "none"

}

}

}

}

乍一看，这个配置文件可能很庞大，但是它有一个优点。首先，electron-compile允许我们维护两个独立的配置集：一个用于开发，一个用于生产。我们选择省略源maps，而不是格式化Jade生成的HTML。源maps对开发人员来说很方便。

当我们使用转换时，我们编写的代码不是浏览器正在执行的代码。这本身就有意义，因为这是使用转换的全部要点，但问题是，调试已编译的代码很难。输出中代码的第36行上的错误可能位于您编写的代码的第12行。源maps允许Chromium将输出映射到输入，并显示您编写的代码中的错误。生成源maps可能需要一些时间，因此通常会从应用程序的生产版本中省略它们。

electron-compile不会配置转换过程本身。它只是将这些选项传递给工具进行实际的转换。在开发环境下，我们有两个额外的键：text/jade和application/javascript。这些选项分别传递给Jade和Babel。application/coffeescript下列出的任何选项都将传递给CoffeeScript编译器。

在传递给Babel的选项中，我们看到我们正在使用插件和预设。预设只是一组插件。react预设包括与React相关的所有预设，例如解析和转换JSX的能力。npm上还有许多其他的预置，但是babel-preset-env有点特殊。它维护了一个常用插件列表和浏览器--Electron包含在这些插件中,已经内置支持并且不需要插件。这允许Babel智能地跳过Electron本身支持的转换功能。

除了预设，我们还包括一些其他插件。Node.js目前不支持ES模块，因此我添加了一个插件，它将ES模块语法转换为我们在本书中使用的CommonJS模块语法。我们还添加了对即将推出的语言功能的支持，该功能称为Object Rest Spread，它在React开发人员中很受欢迎，用于克隆对象以避免变异。

**对ECMAScript类支持的是什么？**

您可能已经注意到我已经为ECMAScript类添加了一个插件，Node和Chromium都支持这个插件，所以这个插件不是必需的。在撰写本文时，React中的热模块重载的错误要求包含此插件。我之所以提到这一点，是因为如果你将来尝试应用本章的概念，你可能会遇到这个错误（https://github.com/gaearon/react-hot- loader/issues/313）。这个错误已经打开了一年半，所以你可能会遇到这个错误并且像我一样失去你下午的大部分时间。

配置electron-compile后，让我们看一下应用程序的基础。我们使用Jade作为标记语言而不是HTML文件。这个文件相对简单，因为React处理UI的大部分繁重工作。将index.html替换为index.Jade，并添加以下内容。

**清单11.2 用Jade编写的渲染器进程的HTML文件：./ app/index.jade**

doctype html html(lang='en')

head

title Jetsetter meta(charset='UTF-8')

meta(name='viewport', content='width=device-width, initial-scale=1') meta(

http-equiv="Content-Security-Policy",

content="default-src 'self'; script-src 'self' 'unsafe-inline'"

)

link(rel='stylesheet', href='style.scss') body

#application

.loading Loading… script.

require('./renderer');

由于商标争议，Jade在一段时间后正式更名为Pug，但它仍然被electron-compile称为Jade，所以我在本章的其余部分也这样做。我们还在Sass中编写样式表，这是CSS的一个功能丰富的替代品。将style.css重命名为style.scss，并按如下方式更新内容。

 **清单11.3 使用Sass的应用程序样式：./ app/style.scss**

**$accent-color: rgb(243,46,91);**

html {

box-sizing: border-box;

**Sass支持变量。**

}

body, input { font: caption;

}

input {

padding: 0.5em;

border: 1px solid $accent-color;

background-color: lighten(**$accent-color**, 30);

**Sass包含许多用于处理颜色的辅助函数。第一个参数是颜色。第二个参数是您希望调整颜色的数量。**

}

button, .button, input[type="submit"] { background-color: $accent-color;

border: 1px solid darken($accent-color, 10); color: white;

padding: 0.5em; &:hover {

background-color: lighten($accent-color, 10);

}

&:active {

background-color: lighten($accent-color, 5);

}

&.full-width { width: 100%;

margin: 1em 0;

**Sass允许您嵌套选择器。这相当于通过添加：hover伪选择器来编写每个选择器。**

}

}

.NewItem { display: flex;

}

.NewItem-input { width: 100%;

}

Sass支持广泛的功能,我将在本章中讨论其中的许多功能。在前面的示例中，我使用变量来设置按钮和输入字段的主要颜色，并使用内置辅助函数如lighten( )和darken( )来调整UI的不同部分的颜色。

通过标记和样式，我们可以设置主进程和渲染器进程。主进程故意比前几章更简单，因为它不是重点。

**清单11.4 Jetsetter的主进程：./ app/main.js**

import { app, BrowserWindow } from 'electron'; let mainWindow;

app.on('ready', () => {

mainWindow = new BrowserWindow({ width: 300,

height: 600,

minWidth: 300,

minHeight: 300, show: false

});

**mainWindow.loadFile('index.Jade');** mainWindow.once('ready-to-show', () => {

mainWindow.show();

});

});

**electron-compile允许我们使用非标准文件扩展名.**

译注:原后缀为jade

请注意，我们可以使用浏览器通常不支持的扩展名的文件。electron-compile将其转换为动态的HTML文件，而Chromium则不是更明智的。如果再看一下清单11.2，您会注意到我们请求了一个名为style.scss而不是style.css的样式表。Sass使用.scss扩展名，在将CSS交给Chromium之前，electron-compile使用Sass进行转换。最后一个难题是设置渲染器进程。

**清单11.5 使用JSX和React的简单组件：./ app/renderer.js**

import React from 'react';



import { render } from 'react-dom';

const Application = () => { return (

<div>

<h1>Hello world!</h1>

<button className="full-width"> This button does not do anything.

</button>

</div>

**需要React库。**

**从ReactDOM库中提取render( )方法。**

**使用一些占位符内容创建一个Application组件。**

);

}; **将应用程序组件渲染到ID为“application”的DOM节点中。**

render(<Application />, document.getElementById('application'));

如果这是您第一次看到React，您可能会惊讶于在JavaScript中看到HTML。此语法称为JSX（带XML的JavaScript）。这是描述React组件应该创建的标记的便捷方式。Babel将此代码转换为传统的JavaScript以供浏览器读取。实际的JavaScript看起来像这个清单。

**清单11.6 在转换后应用程序组件**

const Application = () => { return React.createElement(

"div", null,

React.createElement( "h1",

null,

"Hello world!"

),

React.createElement( "button",

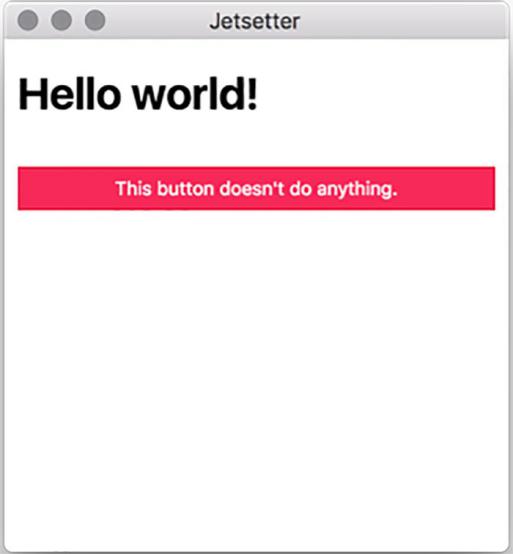
{ className: "full-width" },

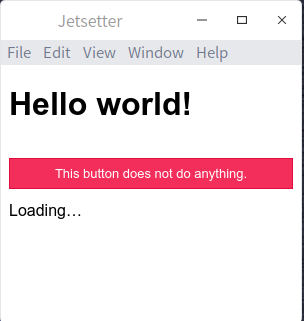
"This button does not do anything."

)

);

};

清单11.5比清单11.6更容易,特别是当应用程序开始增长时。我们不需要使用webpack或Gulp建立构建链。electron-compile负责根据我们在.compilerc中指定的选项配置Babel。如果你使用npm start启动应用程序，你应该看到图11.3的稍高版本。

**图11.3 使用React，Sass和Jade的应用程序的基础。**

### *在React中构建UI*

我们的应用程序组件是Jetsetter UI的起点。Jetsetter有四个主要部分：添加新项目的表单，仍需要打包的所有项目的列表，已打包的所有项目的列表，以及用于返回标记为打包的所有项目的按钮 解包列表。

这两个列表几乎相同，因此它们使用相同的组件，但每个列表也有一个子组件用于列表中的每个项目。除了修改Application更有用之外，我们还需要创建另外三个组件：Item，Items和NewItem。Mark All as Unpacked按钮可以使用标准的<button>元素。组件的层次结构如图11.4所示。

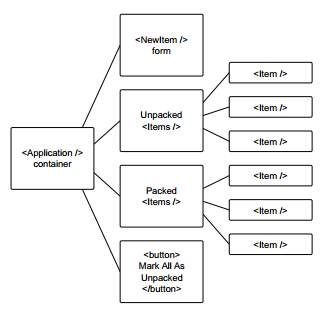


图11.4 Jetsetter分为如下图所示的小组件。

###### *应用程序组件*

我们对世界的愉快问候和无用的按钮作为electron-compile处理我们的JavaScript的证据令人钦佩，但是我们需要将它更换为更具功能性的东西。我们还没有构建我们的NewItem表单，项目列表或单个Item组件，但我们现在为管理应用程序的状态和获取骨骼奠定了基础。到本节结束时，我们的应用将如图11.5所示。

我们的Application组件的第一个版本就是所谓的无状态功能组件。这是一个函数，并在调用时返回一个值。它无法维持任何UI状态。在本章中，应用程序组件维护Jetsetter中所有被跟踪项目的主列表。与Clipmaster一样，我们在内存中使用一个简单的数组来跟踪列表。在第12章中，我们将使用持久数据存储替换此数组，以便每次退出应用程序时都不会丢失列表。

图11.5应用程序组件没有项目。

应用程序组件跟踪Jetsetter中的项目，向该列表添加新项目，并将它们标记为打包或解压缩。我们根据需要添加这些功能。让我们首先用一个项目列表初始化组件，然后添加占位符方法来添加和标记项目。我们还为UI建立了基础，如图11.1所示。您将在./app/components目录中找到应用程序组件的文件。

**清单11.7 应用程序组件基础：./app/component/Application.js**

应用程序组件是React组件类的子类。

除了导入React之外，我们还导入它的Component类。

**首次初始化组件时，将调用constructor( )方法。**

import React, { Component } from 'react';

class Application extends Component {

constructor(props) {

super(props); this.state = {

**初始化组件时，我们将其状态设置为包含一个项目数组，其中包含一个项目.**

items: [{ value: 'Pants', id: Date.now(), packed: false }]

};

this.addItem = this.addItem.bind(this); this.markAsPacked = this.markAsPacked.bind(this);

this.markAllAsUnpacked = this.markAllAsUnpacked.bind(this);

}

addItem(item) {} // To be implemented… markAsPacked(item) {} // To be implemented… markAllAsUnpacked() {} // To be implemented…

**每个动作都发生在事件循环上，这意味着它们会丢失此组件的上下文。将方法绑定到应用程序组件的当前实例是React中的常见模式。**

**过滤列表以查找未标记为已打包的所有项目。**

**过滤列表以查找标记为已打包的所有项目。**

render() {

const { items } = this.state;

const unpackedItems = items.filter(item => !item.packed); const packedItems = items.filter(item => item.packed);

return (

<div className="Application">

{/\* To be implented: <NewItem /> \*/}

{/\* To be implented: <Items title=”Unpacked Items” /> \*/}

{/\* To be implented: <Items title=”Packed Items” /> \*/}

<button

className="full-width" onClick={this.markAllAsUnpacked}

>

**从组件的状态中拉出项目列表。或者，您可以在任何地方使用this.state.items，您可以在后面的代码中看到项目。**

Mark All As Unpacked

</button>

</div>

);

}

}

**export default与CommonJS模块中的module.exports类似，这是我们在前面章节中使用的。**

**React组件使用className而不是class，因为class是JavaScript中的保留字。**

export default Application;

从React的Component类继承的组件可以拥有方法并保持状态。在上一个清单中，我们将状态初始化为具有一系列项目。现在，我们在那里放了一件重要的衣服。在第12章中，我们将在应用程序启动时从持久存储中获取此项。我们还遵循一个常见的React模式，在初始化时将方法绑定到新实例。这可以确保方法在以后调用时绑定到正确的方法。我们现在为addItem( )，markAsPacked( )和markAllAsUnpacked( )添加空占位符。我们在本章中逐一介绍这些内容。

在将组件呈现给DOM时，我们将完整的项目列表分成较小的打包和解包项目列表。我们还没有构建渲染UI所需的其余组件，所以我们现在包含占位符。但是，我们可以在页面底部放置Mark All as Unpacked按钮。我们需要做的最后一件事是将./app/renderer.js中的占位符Application组件替换为real-if incomplete-thing。

**清单11.8 将新的Application组件安装到DOM上：./app/renderer.js**

import React from 'react';

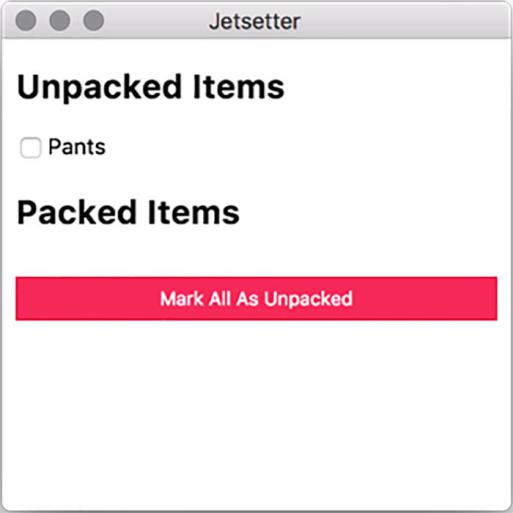
import { render } from 'react-dom';

**import Application from './components/Application';**

**导入新的应用程序组件。**

render(<Application />, document.getElementById('application'));

###### *显示项目列表*

如果无法向您显示这些项目，则跟踪打包和解包项目的应用程序并不是很好。在本节中，我们实现了两个列表，以及在打包和解包状态之间切换项目的功能。此功能需要为单个项目和项目列表创建组件。由于能够将物品标记为已包装或未包装，我们还可以加倍努力，将Mark All as Unpacked按钮连接到物品清单。在本节末尾，该应用程序如图11.6所示。

**图11.6 应用程序现在包含打包和解包项目列表。**

**INDIVIDUAL ITEMS**

应用程序组件正在努力跟踪加载到Jetsetter中的所有项目。只需要告知单个项目应显示的内容，并在用户单击其复选框时调用函数。

**清单11.9 单个项目组件：./ app/components/Item.js**

import React from 'react';

const Item = (({ packed, id, value, onCheckOff }) => { return (

<article className="Item">

<label>

**此功能组件需要四个属性，并将它们存储为函数内的局部变量。**

<input type="checkbox" checked={packed} onChange={onCheckOff} />

{value}

);

});

</label>

</article>

**只要单击复选框，就会在调用onChange属性时传递给输入的函数。**

export default Item;

单个Item组件相对简单。这是一个带有复选框和标签的容器。围绕<label>标签中的输入允许用户除了复选框之外单击标签以选中或取消选中该框。只要选中或取消选中复选框，React就会触发传递给onChange( )的函数。此功能最终成为应用程序组件的markAsPacked( )方法。

单个Item是好的，但他们需要一个容器。在下面的清单中，我们创建了一个简单的组件，它接受一个项目列表以及如果单击其中一个项目该怎么做。它返回一个<section>，每个项目都是一个子组件。它还需要title属性来帮助我们区分打包和解包列表。

**Listing 11.10 Item list component: ./app/components/Items.js**

import React from 'react'; import Item from './Item';

const Items = ({ title, items, onCheckOff }) => { return (

<section className="Items">

<h2>{ title }</h2>

{items.map(item => (

<Item

key={item.id}

onCheckOff={() => onCheckOff(item)}

**使用一组组件时，React要求每个组件都有一个唯一的key。**

将应检查的特定项目传递给onCheckOff( )

{...item}

/>

))}

</section>

);

};

export default Items;

**spread运算符将所有项的属性传递给组件。**

Items组件为自己获取title属性，然后根据它提供的项目数组创建一组Item组件，并通过对特定项目的引用传递给每个组件的onCheckoff( )函数。我们现在拥有了我们需要的UI元素。是时候将它们连接到应用程序组件了。

Items组件知道如何使用各个Item组件，因此我们只需要将Items组件放入应用程序组件中。我们确实需要实现markAsPacked( )和markAllAsUnpacked( )。

**清单 11.11 Item和Items在应用程序组件:./app/components/Application.js**

import React, { Component } from 'react'; import **Items** from './Items';

class Application extends Component { constructor(props) {

// Omitted for brevity…

**导入Items组件。**

}

addItem(item) {} // To be implemented…

**创建一个不是我们正在寻找的所有项目的新数组。**

markAsPacked(item) {

const otherItems = this.state.items.filter( other => other.id !== item.id

);

const updatedItem = { ...item, packed: !item.packed }; this.setState({ items: [updatedItem, ...otherItems] });

}

**用项目和所有其他项目替换组件状态中的项目。**

**使用object rest spread运算符来克隆对象，用其相反的方式替换打包key。**

markAllAsUnpacked() {

const items = this.state.items.map(item => ({ ...item, packed: false }));

this.setState({ items });

}

**使用现有项的副本创建一个新的项目数组，并将其packed属性设置为false。**

**用我们刚制作的新项目数组替换状态中的项目。**

render() {

const { items } = this.state;

const unpackedItems = items.filter(item => !item.packed); const packedItems = items.filter(item => item.packed);

return (

<div className="Application">

<Items

title="Unpacked Items" items={unpackedItems} onCheckOff={this.markAsPacked}

**使用新的Items组件显示解压缩的项目。**

/>

<Items

title="Packed Items" items={packedItems} onCheckOff={this.markAsPacked}

/>

<button

className="button full-width" onClick={this.markAllAsUnpacked}

>

Mark All As Unpacked

</button>

</div>

);

}

}

**使用新的Items组件显示打包的项目。**

单击Mark All as Unpacked按钮时，调用markAllAsUnpacked( )方法。

export default Application;

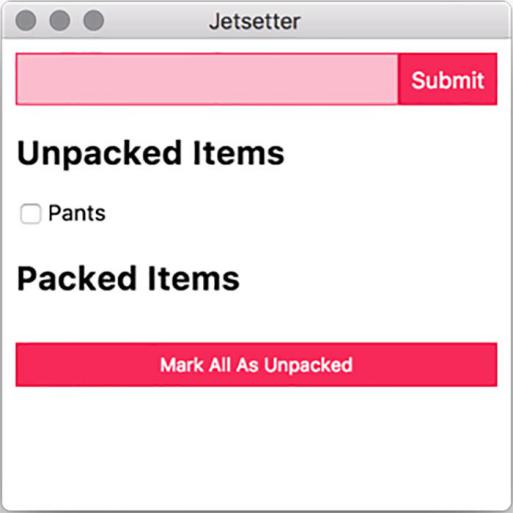
我们首先引入我们在上一节中创建的Items组件。我们在render( )方法中使用它两次：一次用于打包的项目，一次用于解压缩的项目。我们将每个实例的各个项目列表以及标题和markAsPacked( )方法传递给每个实例，该方法最终将传递给各个组件。

更棘手的部分在于markAsPacked( )和markAllAsPacked( )方法。两者都在修改和更新传递给两个列表的项目数组。构建React动作的核心租户是我们尽量不改变对象和数组;相反，我们创建新的动作。React的虚拟DOM差异算法更容易看到它正在使用一个全新的对象，而不是现有对象的新版本已经巧妙地改变了。

在markAsPacked( )中，我们从一个我们没有使用的所有项目的列表开始，并将它们放在一边。然后，我们制作传递给该方法的项目的副本。我们保持一切相同，但覆盖相反的打包密钥。如果这是真的，现在它是假的; 如果它是假的，现在是真的。然后，我们使用包含新项目的新数组更新组件的状态，其中包含我们之前放置的所有项目。

在markAllAsPacked( )中，我们通过映射现有项并使用值false替换其打包密钥来创建项数组。无论是Mark All as Unpacked按钮都应该将项目设置为unpacked。使用这些方法后，您应该能够将“Pants”标记为已打包或未打包。这个应用程序受益于我们将项目硬编码到应用程序的初始状态这一事实，但现在是时候我们让用户能够将他们自己的项目添加到Jetsetter。

### *添加新项目*

我们的下一步是创建NewItem组件并将其连接到应用程序的状态。与Item和Items不同，NewItem需要保持自己的状态：输入字段的内容。在本节结束时，应用程序将如图11.7所示。

**图11.7 除了“Pants”之外，应用程序现在还可以添加一个新项目。**

NewItem是另一个有状态组件。它需要跟踪输入字段的值，以便在用户提交表单时正确命名它创建的新项目。每当用户键入输入字段时，它都会更新其内部状态以反映该值。此更改本身在输入字段中表示，从而创建良性循环。

***添加新项目* 215**

**清单11.12 添加NewItem：./ app /components/NewItem.js**

import React, { Component } from 'react'; class NewItem extends Component {

**绑定handleChange( )方法，以便在从事件队列调用时具有对组件的引用。**

constructor(props) {

super(props); this.state = {

value: ''

};

**将输入字段的初始状态设置为空字符串。**

绑定handleSubmit( )方法，以便在从事件队列调用时具有对组件的引用。

this.handleChange = this.handleChange.bind(this); this.handleSubmit = this.handleSubmit.bind(this);

}

**根据输入字段的值更新组件的状态。**

**从组件的状态中拉出输入字段的当前值。**

handleChange(event) {

const { value } = event.target; this.setState({ value });

}

handleSubmit(event) {

const { onSubmit } = this.props; const { value } = this.state;

event.preventDefault();

**event.target是触发事件的DOM节点。在这种情况下，这是输入字段。我们从输入字段中提取值。**

我们传入一个函数，指示当用户单击提交（或按回车键）从Application作为此组件的prop时要执行的操作。

使用当前UNIX时间戳将表示新项目的对象传递为合理的唯一标识符。

onSubmit({ value, packed: false, id: Date.now() });

this.setState({ value: '' });

}

**将输入值重置为空字符串，以便用户轻松输入另一个项目。**

render() {

const { value } = this.state;

return (

<form className="NewItem" onSubmit={this.handleSubmit}>

**当用户提交表单时触发handleSubmit( )方法。**

<input

className="NewItem-input" type="text"

value={value} onChange={this.handleChange}

/>

**当用户提交表单时，使用当前状态值。**

<input className="NewItem-submit button" type="submit" />

</form>

);

} **每当用户对输入字段进行更改时，都会更新状态值。**

}

NewItem.defaultProps = { onSubmit: () => {}

};

export default NewItem;

**将onSubmit( )的默认值设置为空函数，以便我们不会意外触发“undefined is not a function”作为错误，如果省略它。**

键入输入字段的用户触发作为onChange prop传入的函数。为此，我们使用了恰当命名的handleChange( )方法。onChange将DOM事件的表示传递给handleChange( )。event.target是输入字段的DOM节点。正如我们对Fire Sale中的文本区域所做的那样，我们要要它的值。使用此值，我们更新组件的状态，该状态还更新UI中显示的输入值。

在React中，组件通常不了解其父母。NewItem无法直接与应用程序通信。相反，Application将函数作为NewItem的prop传递。当用户提交表单时，NewItem使用此函数。我们还没有编写这个函数，但是,当我们这样做时,它只是将新项目添加到Application的项目数组中，而现在它只包含一条孤独的裤子。

当用户提交表单时，将调用作为onSubmit prop传入的函数。同样，我们擅长命名并选择将此方法命名为handleSubmit( )。在表单提交时，创建一个项目并将其传递给作为NewItem的onSubmit prop传入的方法。应用程序组件将其一个方法传递给NewItem作为其onSubmit prop。反过来，NewItem将它作为onSubmit prop传递给它创建的表单。提交表单后，我们将状态中的值替换为空字符串,允许用户输入另一个项目。

NewItem目前生活在真空中。它理论上是有效的，但在我们能够有效地使用之前，需要将它添加到应用程序组件中。我们的下一步是将其添加到UI并向其传递一个函数，将其添加到应用程序的项列表中。我们在这里这样做。

**清单11.13 应用程序组件中的NewItem：./app/components/application.js**

import React, { Component } from 'react'; **import NewItem from './NewItem';**

import Items from './Items';

class Application extends Component { constructor(props) {

// Omitted for brevity…

}

addItem(item) {

this.setState({ items: [item, ...this.state.items] });

}

markAsPacked(item) {

// Omitted for brevity…

}

markAllAsUnpacked() {

// Omitted for brevity…

**addItem( )将状态设置为一个新数组，该数组包含所有现有项以及作为参数传入的项。**

}

render() {

const { items } = this.state;

const unpackedItems = items.filter(item => !item.packed); const packedItems = items.filter(item => item.packed);

return (

<div className="Application">

**<NewItem onSubmit={this.addItem} />**

<Items

title="Unpacked Items" items={unpackedItems} onCheckOff={this.markAsPacked}

**我们将NewItem组件添加到应用程序组件中，并将addItem( )方法作为其onSubmit prop传递给它。**

/>

<Items

title="Packed Items" items={packedItems} onCheckOff={this.markAsPacked}

/>

<button

className="button full-width" onClick={this.markAllAsUnpacked}

>

Mark All As Unpacked

</button>

</div>

);

}

}

export default Application;

将NewItem组件添加到应用程序组件后，您现在应该能够创建其他项目，在解压缩列表和打包列表之间切换它们，并将所有项目移动到解压缩列表。

### *实时加载和热模块重载*

在本书中，我们通过按Command-R或Control-R来开发肌肉记忆，分别在macOS和Windows/Linux上进行更改后刷新呈现器进程。但是，每次进行相对较小的更改时，在文本编辑器和应用程序之间来回切换可能会有点乏味。

electron-compile可以在磁盘上更改文件时触发重新加载。这意味着当您在文本编辑器中保存文件时，它会立即重新加载打开的渲染器进程,允许您立即查看更改。electron-compile还支持React中的热模块重新加载，通过交换已经动态更新的模块而无需重新加载页面本身，可以进一步实时重新加载。在本节中，我们将介绍如何实现这些方法。

###### *启用实时加载*

在此之前，electron-compile工作正常。我们在package.json中使用了electron-compile-prebuilt代替electron，做了一些最小和可选配置，然后我们就出发了。electron-compile也可以像任何其他模块一样需要。该模块提供了一个enableLiveReload( )方法，可以执行它在tin上所说的内容。

**Listing 11.14 启用实时加载: ./app/main.js**

import { app, BrowserWindow } from 'electron'; import { enableLiveReload } from 'electron-compile';

从electron-compile模块中提取enableLiveReload( )函数

**enableLiveReload();**

let mainWindow; app.on('ready', () => {

// Omitted for brevity…

});

**在应用程序准备好启用实时重新加载之前调用该函数。**

当我们在应用程序启动之前调用enableLiveReload( )函数时，electron-compile会建立一个IPC连接，用于侦听文件系统更改并向渲染器进程发送消息，请求它在检测到更改时重新加载。

###### *设置热模块重载*

enableLiveReload( )与框架无关，甚至可以在Fire Sale和Clipmaster 9000等简单应用程序中运行.enableLiveReload( )还支持使用React应用程序重新加载热模块。第一步是让enableLiveReload( )知道我们更喜欢使用热模块重新加载作为刷新整个页面的替代方法。

**清单11.15 设置热模块重载: ./app/main.js**

import { app, BrowserWindow } from 'electron'; import { enableLiveReload } from 'electron-compile';

enableLiveReload(**{ strategy: 'react-hmr' }**); let mainWindow;

app.on('ready', () => {

// Omitted for brevity…

});

**通过enableLiveReload( )传递一个对象，告诉它使用热模块重新加载。**

将enableLiveReload()配置为使用热模块重新加载，当您更改文件时，它不再刷新页面。但是热模块重载也不起作用。~~游手好闲的人~~。我们实际上回到了起点。我们还有一些工作要做。

如果您使用的是TypeScript，则可以跳过下一步。但是我们在本章中没有使用TypeScript，所以我们让Babel知道我们正在使用热模块重新加载。我们将它添加到.compilerc中的插件列表中。我们不会将其作为插件添加到生产中，因为我们可能不会经常在用户的计算机上更改文件。

**清单11.16 为热模块重新加载添加Babel支持：./.compilerc**

{

"env": { "development": {

"application/javascript": {

// Omitted for brevity… "plugins": [

**"react-hot-loader/babel",**

"syntax-dynamic-import",

"dynamic-import-node",

"transform-es2015-classes",

// Additional plugins here…

**将react-hot-loader/babel添加到Babel插件列表中**

],

"sourceMaps": "inline"

},

"text/jade": { "pretty": true

}

},

"production": {

// Omitted for brevity…

}

}

译注：上面的３行红色代码插件是根据作者的最终代码补加的，如果不加界面按钮就失效了！

我们还没有走出困境,我们需要做出另外一个改变：我们的应用程序需要包装在一个容器组件中，该组件监听更改并将这些更改传递给我们的Application组件。好消息是我们不必自己编写这个组件; 我们只需要从react-hot-loader模块中获取它。

**清单11.17 使用AppContainer订阅更新：./app/renderer.js**

import React from 'react';

import { render } from 'react-dom';

import { **AppContainer** } from 'react-hot-loader';

const renderApplication = () => {

**在我们可以重复调用的函数中包装我们的初始渲染方法。**

const { default: Application } = require('./components/Application'); render(

<AppContainer>

<Application />

</AppContainer>, document.getElementById('application')

**需要一个新版本的Application，它在渲染时需要我们应用程序中的所有其他模块。**

);

};

renderApplication();

**像应用程序第一次启动时通常一样呈现应用程序。**

if (module.hot) { module.hot.accept(renderApplication); }

**如果启用了热模块重新加载，则只要我们收到组件已更改的消息，就会再次呈现应用程序。**

我们有一些活动件。在我们最初的实现中，我们引入应用程序组件并在渲染器进程启动时呈现它。这个方法是构建React应用程序时的标准做法，而不是特定于我们在Electron内部完成所有这些操作的事实。但是如果我们要利用热模块重新加载，我们需要做一些调整。我们需要将呈现应用程序的行为转移到我们可以重复调用的函数中。我们将此函数命名为renderApplication( )，并在首次计算渲染器进程时调用它。在我们实现前面描述的更改之前，此函数模仿现有功能。

在渲染应用程序时，我们需要一个新版本的应用程序组件，因为它可能已经改变了。在撰写本文时，ECMAScript的模块导入只能在顶层使用，因此我们必须依靠CommonJS的require( )函数来引入应用程序组件。我们将应用程序包装在react-hot-loader的AppContainer组件中，该组件让它知道文件系统的更改，并将其安装到我们在本章开头所做的同一个DOM节点上。

完成所有这些后，应启用热模块重新加载，您可以实时查看模块的更改。如果CommonJS require( )和ECMAScript import语句的混合让你觉得内心很奇怪，你确实有一些选择。即将进行的规范更改允许使用基于承诺的API异步获取模块的第二版import( )。清单11.19显示了如何实现它。我们在./components/Application.js中使用了export default，因此我们需要构造传递给promise的对象的默认属性并将其命名为Application。

**清单11.18使用import( )异步加载依赖项：./ app/renderer.js**

const renderApplication = () => {

import('./components/Application').then(

({ default: Application }) => { render(

<AppContainer>

<Application />

</AppContainer>, document.getElementById('application'),

**import( )返回带有模块内容的promise。**

);

}

);

};

Node和Chromium都支持即将发布的async/await语法，当使用基于承诺(Promise)的API时，它允许更传统的同步语法。您可以在没有Babel和electron-compile的情况下使用它，因为Node和浏览器本身支持它。

***本章小结* 221**

**清单11.19 使用async/await异步加载依赖项：./app/renderer.js**

const renderApplication = async () => {

const { default: Application } = await import('./components/Application'); render(

<AppContainer>

<Application />

</AppContainer>, document.getElementById('application')

);

};

这两个重构都是完全可选的，主要用于说明Electron支持开箱即用的尖端语法，并且electron-compile使得Babel可以轻松地添加对JavaScript语言未来功能的支持。

译注:

关于5.0.x版本

热更新经测试虽然警告,但是依然可用;

Scss后缀文件在React中不认,可以改为css后缀,但是部分计算样式丢失!

### *本章小结*

* + - * electron-compile允许开发人员使用各种语言编译成HTML，JavaScript和CSS，如Jade，TypeScript，CoffeeScript，Sass和Less。
      * electron-prebuilt-compile是安装了electron-compile的Electron二进制版本。
      * 可以在没有配置的情况下使用多种语言，但electron-compile还支持.compilerc文件，允许开发人员配置每种语言。
      * electron-compile可以配置为使用Babel编译JSX。
      * electron-compile可以根据文件扩展名自动检测使用哪个转换器。
      * 使用electron-compile的electron应用程序可以使用实时重新加载来自动刷新所有打开的浏览器窗口。
      * electron-compile的实时重载功能还支持React应用程序中的热模块重载。

*持久化用户数据与使用原生Node.js模块*

***本章包括***

* 在主进程和渲染器进程中使用用C ++构建的Node.js模块
* 为Electron的Node版本获取正确版本的依赖项
* 使用SQLite和IndexedDB数据库来保存数据
* 在操作系统的指定位置监控用户特定的应用程序数据

在第11章中，我们构建了一个小应用程序来跟踪我们在旅行之间需要打包的项目。到前一章结束时，我们的UI工作正常，但应用程序仍有一个致命的缺陷：每当页面刷新时，它都会丢失所有数据。对于据称应该帮助你记住事情的应用程序来说，这是致命的。Fire Sale是使用文件存储在磁盘上，所以这不是问题，但是Clipmaster和Clipmaster 9000也有这个问题。每当用户退出应用程序时，所有剪辑都会丢失，或者我们在开发过程中刷新了页面。

在本章中，我们将一劳永逸地解决这个问题。数据在页面加载之间保持不变，即使用户退出应用程序并重新启动计算机，也仍然可用。为了更好地解决这个问题，我们使用两种方法:我们创建了一个本地SQLite数据库和一个基于浏览器的IndexedDB存储。在此过程中，我们还介绍了一些有趣的实现细节：我们在哪里以每个用户为基础存储数据？如果它与我们计算机上安装的版本不同，我们如何为Electron的Node版本构建编译模块？如何使用像SQLite这样的SQL数据库与NoSQL数据库（如IndexedDB）有何不同？

在第11章中删除项目并不是特别重要，因为我们无法保留数据，但现在肯定是这样。到本章结束时，我们的应用程序看起来与我们在开始时的情况类似，但有一些主要区别：数据持久存储到磁盘，用户可以从列表中删除项目并删除所有项目还没有打包。

**222**

### *12.1 将数据存储在SQLite数据库中*

我们采用的第一种方法是将数据存储在SQLite数据库中。传统的本机应用程序通常使用此方法，尤其是在macOS和iOS上。SQLite是一个很好的选择，因为数据库存储在一个文件中，不需要用户在他们的系统上安装MySQL或PostgreSQL。

如果你是一名正在恢复的数据库管理员，我会给你一个公平的警告：我不打算优化每个查询，有些技术可能有点浪费。我正在优化代码清晰度而不是性能。我们知道在这种情况下,我们正在处理一个非常小的数据集。

如果您来自前端Web开发背景，您可能没有想过在您的应用程序中使用SQLite数据库。您通常将HTTP请求发送到服务器或使用基于浏览器的解决方案，如IndexedDB，WebSQL或LocalStorage。SQLite数据库在传统Web应用程序中不经常使用的原因是因为它们不能--我们无法从浏览器访问文件系统。此外，在本章期间，我们不能使用我将要称为原生模块的内容。

什么是原生模块？许多类似Lodash或Moment.js的库都是用JavaScript编写的。浏览器和Node都可以执行JavaScript，因此这些模块可以在任何一个上下文中使用。一些类似jQuery的库与DOM紧密耦合，因此只能在浏览器上下文中使用。原生模块通常用JavaScript包装C或C ++库。必须为将使用库的操作系统编译库的C或C ++组件。SQLite和大多数其他数据库驱动程序都有用C或C ++编写的库。npm上的sqlite3模块将此库包装在JavaScript绑定中，以便我们可以在Node应用程序中使用它。浏览器不仅无法访问文件系统，还无法运行特定于平台的C和C ++代码。像Emscripten这样的项目编译C代码以在JavaScript虚拟机上运行，但这远远超出了本书的范围。

正如我们在本书中讨论的那样，Electron应用程序将Chromium浏览器运行时与Node运行时结合在一起，因此我们可以在应用程序中使用原生模块。我们首先设置一个SQLite数据库，为我们的项创建一个表，然后连接UI以直接从渲染器进程读取和写入数据库。

###### *使用正确的electron-rebuild版本*

当我们在Node中安装原生模块时，它将根据Node使用的当前版本的V8引擎进行编译。升级版本的Node通常会导致必须重新编译应用程序使用的所有原生模块。Electron附带自己的Node运行时，当您运行npm install或yarn install时，它可能与您计算机上运行的Node版本相同或不同。当您尝试在Electron应用程序中使用原生模块时，这种不匹配可能会导致问题。

对我们来说幸运的是，社区已经非常友好地为我们提供了一个称为electron-rebuild的解决方案，该解决方案针对Electron使用的Node版本而不是文件系统上安装的版本编译原生模块。您可以使用**npm install electron-rebuild --save-dev**通过npm安装electron-rebuild。然后可以通过在macOS上使用$（npm bin)/ electron-rebuild来触发它或者Windows上的.\ node\_modules\.bin\electron-rebuild.cmd。

我不希望每次安装依赖项时都记得调用它。相反，我建议在package.json中使用postinstall挂钩，它将在每次安装后运行。

**清单12.1 添加一个postinstall钩子：/package.json**

{

"name": "jetsetter",

"version": "1.0.0",

"description": "An application for keeping track of the things you need

➥ to pack.",

"main": "app/main.js", "scripts": {

"start": "electron .",

"test": "echo \"Error: no test specified\" && exit 1", **"postinstall": "electron-rebuild"**

},

// Additional configuration omitted for brevity.

**每次运行npm install后都会调用postinstall脚本。**

}

默认情况下，npm在运行任何脚本之前检查前前置或后置脚本。当你运行npm test时，它首先尝试运行pretest，然后测试，然后进行posttest。多年来，我发现这个技巧在各种各样的项目中都非常有用。

###### *设置SQLite和Knex.js*

在本章中，我们使用一个名为Knex.js的有用库来使Node中的SQL更容易一些。它充当我们的应用程序代码和底层SQLite查询之间的接口。这是在Jetsetter存储库的第12章开始分支中为您设置的。如果要从第11章开始构建实现，可以使用**npm install sqlite3 knex**安装这些依赖项。

在我们将SQLite集成到我们的应用程序之前，我们必须做一些初始设置。正如我之前提到的，**SQLite将数据存储在一个文件中**，因此我们需要弄清楚我们将要存储此文件的位置。我们现在从一个简单但有缺陷的解决方案开始。在本章后面，我将重新审视存储用户数据的最佳位置，但现在让我们专注于让我们的应用程序正常工作。

**清单12.2 设置SQLite数据库：./app/database.js**

import 'sqlite3';

import knex from 'knex';

const database = knex({ client: 'sqlite3', connection: {

filename: './db.sqlite'

},

useNullAsDefault: true

});

export default database;

**导入SQLite库**

告诉Knex.js我们打算将它与SQLite数据库一起使用

指定应创建SQLite数据库的位置。

将Knex.js配置为在未提供特定列的值时使用NULL。

导出已配置的数据库。

我们需要包含sqlite3库，以便在Knex寻找它时将其加载到应用程序中，并让Knex知道我们将使用SQLite作为此应用程序的数据库。

调用此属性连接在这里有点用词不当。如果我们连接到MySQL服务器，这个名称就可以了，但正如我所提到的，SQLite使用存储在磁盘上的文件，所以我们在这里放置了我们想要存储文件的位置的名称。我现在称它为db.sqlite。我们稍后再回顾一下。

那个useNullAsDefault选项有什么用？ Knex.js不是专门用于SQLite的库。相反，它适用于PostgreSQL，MySQL，MSSQL和Oracle数据库。其中许多支持数据库中列的默认值。SQLite没有，如果我们不打开此选项，Knex会显示警告，这会导致Knex.js选择NULL而不是尝试使用默认值。

这还不足以让我们一路走来。我们创建了数据库，但是我们仍然没有使用表来配置它来存储我们的项目。当应用程序启动时，我们检查是否有用于存储项目的表。如果没有，那么我们创建表。

**清单12.3 创建一个用于存储项目的表：./app/database.js**

创建项目表

import 'sqlite3';

**检查数据库是否已有项目表**

仅当项目表不存在时才创建

const database = require('knex')(//…);

**将value列设置为宽度为100个字符的字符串。**

database.schema.hasTable('items').then(exists => {

if (!exists) {

return database.schema.createTable('items', t => {

t.increments('id').primary();

t.string('value', 100);

**创建一个id列作为主键并自动递增**

}

});

t.boolean('packed');

});

**设置packed列以存储布尔类型**

export default database;

Knex.js使用基于承诺的API。对表的检查将返回一个承诺，该承诺使用布尔值来实现，具体取决于它是否确实存在。我们在下一节中的所有查询都基于承诺。Jetsetter中的数据非常简单，我们的架构反映了这一点。SQLite创建一个唯一的ID，并在添加到数据库的每个新项目上自动递增ID。我们还将项目的名称作为字符串存储在值列中，并使用布尔值表示项目是否已打包。

###### *Hook数据库到React*

在传统的Web应用程序中，如果我们想将某些内容放入SQLite数据库中，我们可能不得不将AJAX请求发送到服务器，该服务器将与数据库进行交互。这意味着我们可能还会实施某种身份验证以及授权，以确保用户无法读取或编辑其他用户的数据。在Electron应用程序中，我们可以从客户端代码直接与数据库通信。

在顶层，我们引入我们刚刚创建的已配置数据库，并将其作为prop传递给应用程序组件,React代表“property”，它可以访问其方法内的数据库。

**清单12.4 将数据库传递给应用程序组件：./app/renderer.js**

import React from 'react';

import { render } from 'react-dom';

import { AppContainer } from 'react-hot-loader'; **import database from './database';**

**导入我们在./app/database.js中创建的数据库。**

const renderApplication = async () => {

const { default: Application } = await import('./components/Application'); render(

<AppContainer>

<Application **database={database}** />

</AppContainer>, document.getElementById('application')

**将其作为属性传递给Application。**

);

};

renderApplication();

if (module.hot) { module.hot.accept(renderApplication); }

有趣的来了。以前，我们将应用程序的状态存储在应用程序组件中。每次重新加载应用程序时，都会替换该状态。每次用户退出应用程序时，状态都会消失。在接下来的几个示例中，我们通过读取和写入数据库来替换此行为。

这个过程包含几个部分。当应用程序组件第一次启动时，它会从数据库中读取所有项目并将其加载到其内部状态。只要它有理由相信数据已经改变，它也可以做到这一点，这使我们有一个事实来源：数据库。根据您的需要，您可能决定以不同方式处理数据存储策略，但此方法适用于Jetsetter。

当用户创建新项目时，我们将其添加到数据库中。当他们检查一个项目时，我们通过将“packed”布尔值翻到相反的位置来更新它。当用户选择Mark All as Unpacked时，我们 毫不奇怪地选择数据库的所有项目并将其“packed”属性设置为false。

正如我在本章开头所提到的，我们正在添加从数据库中删除项目的功能。当我们在每次重新加载时丢失所有内容时，此功能并不重要，但现在有必要。此外，我们添加了一个按钮来从数据库中删除所有解压缩的项目。

###### *从数据库中获取所有项目*

我们继续使用this.state来保存最近从数据库中提取的项目列表。但是，我们不是将一条pants硬编码到这个列表中，而是添加一个从数据库中获取所有项目然后更新此列表的方法。每当组件第一次启动时，我们也会调用此方法。

**清单12.5 从数据库中获取项目：./app/component/Application.js**

class Application extends Component { constructor(props) {

super(props);

this.state = {

**items: []**

};

**将组件的初始状态设置为空数组**

**绑定提取功能，以便它可以访问正确的上下文**

**this.fetchItems = this.fetchItems.bind(this);** this.addItem = this.addItem.bind(this); this.markAsPacked = this.markAsPacked.bind(this);

this.markAllAsUnpacked = this.markAllAsUnpacked.bind(this);

}

**componentDidMount()** {

this.fetchItems();

}

**fetchItems() {**  this.props

.database('items')

.select()

**组件启动后立即从数据库中获取项目**

查询数据库以获取项目列表

.then(items => this.setState({ items }))

.catch(console.error);

}

**更新状态中存储的项目数组**

addItem(item) { … } markAsPacked(item) { … } markAllAsUnpacked() { … }

render() { … }

}

export default Application;

我们不希望在尝试映射或迭代未定义的值时出现任何错误，因此我们将组件的初始状态设置为空数组。在前面的示例中，我们将作为属性传递给应用程序组件的数据库实例作为属性，并询问它们之前设置的项目表。.select( )方法选择该表中的所有行,这是我们案例中的所有项并返回一个promise。如果一切都成功，我们可以使用.then( )从.select( )方法中获取结果并使用这些行。

我们从作为空数组的项列表开始。this.fetchItems( )查询数据库并返回一个promise。当这个承诺解决后，我们将存储在组件状态中的项目与我们从数据库中收到的项目进行交换。React非常聪明，能够找出代表我们改变UI的含义。我们不需要担心这一点。

因为this.fetchItems( )以异步方式工作，所以我们需要将它绑定到组件的上下文，就像我们使用this.addItem( )，this.markAsPacked( )和this.markAllAsPacked( )一样。最后，我们需要调用this.fetchItems( )来加载初始状态。我们在组件启动后执行此操作，这会立即更新组件的状态。

###### *将项添加到数据库*

我们实施的功能令人印象深刻。我们从UI直接连接到数据库。这不是我们在传统Web应用程序中能够做到的事情。但它似乎并没有令人印象深刻，因为数据库中没有任何东西可以显示。目前这只是一个空列表。我们可以在其中放入模拟数据，但是让我们切入正题，实现向数据库添加新项的功能。

如果您从第11章回忆起，我们将一个函数从Application组件传递给NewItem组件。当用户单击Submit按钮时，NewItem组件获取其输入字段的内容并将其传递给应用程序组件提供的功能，该功能又将其推送(push)到状态存储的项目数组的末尾。

我们将保留大部分此功能，但有一个值得注意的例外：我们将项目插入到存储在Application组件状态中的数组末尾，而不是将其插入数据库中，然后触发this.fetchItems( )现在重新加载数据库中的所有项目。我们首先在应用程序组件中重写this.addItem( )以使用数据库而不是内存数组。

**清单12.6 实现向数据库添加项目的能力: ./app/components/Application.js**

class Application extends Component { constructor(props) { … } componentDidMount() { … } fetchItems() { … }

addItem(item) {

this.props

.database('items')

.**insert(item)**

**.then(this.fetchItems);**

}

markAsPacked(item) { … } markAllAsUnpacked() { … }

render() { … }

}

**将项目插入数据库。**

将项目插入数据库完成后，重新获取所有项目。

export default Application;

访问数据库中的items表与获取所有项目的项目表相同。主要区别在于我们使用insert( )方法将NewItem组件中的新项添加到数据库中。当成功完成后，我们调用this.fetchItems( )，它获取最新的项列表，然后更新UI的状态。

我们可以在数据库中添加项目并在UI中查看它们，但是我们有一个微妙的问题。React特别关注每个项目都有唯一的键。我们通过将当前日期用作唯一键的整数来进行不太理想的黑客攻击。这很有效，但我当时感觉并不好，而且我现在感觉不太好，因为我们有一个跟踪和自动增加唯一ID的数据库。让我们删除手动创建唯一ID的技巧。

**清单12.7 让数据库处理递增ID：./app/components/NewItem.js**

class NewItem extends Component { constructor(props) { … } handleChange(event) { … }

handleSubmit(event) {

const { onSubmit } = this.props; const { value } = this.state;

event.preventDefault();

**onSubmit({ value, packed: false });** this.setState({ value: '' });

**删除在我们的新项目上设置ID的属性**

}

render() { … }

}

一开始可能很难注意到这种变化，但我们只是作为一名软件工程师参与了我最喜欢的活动之一：删除代码。以前，我们传递了三个属性：value，packed和id。通过省略我们自己的ID, SQLite为我们创建了一个ID，从而消除了我们对涉及时空连续体的奇怪技巧的依赖。

###### *更新数据库中的项目*

大多数使用数据的应用程序都实现了四种基本的CRUD操作：创建，读取，更新和删除。我们能够从数据库中读取所有项目，我们只是实现了创建新项目的能力。我们在那里进行一半了。我们的下一步是能够更新现有项目。

我们的应用程序的UI提供了两种更新数据库中项目状态的方法：用户可以选中或取消选中与项目关联的复选框输入。用户可以单击Mark All as Unpacked按钮来操作数据库中的所有项目。对于每种情况，我们需要两种略有不同的方法。

让我们从我们想要更新数据库中的单个项目的情况开始。为此，我们需要执行两个操作：找到我们想要更新的特定项目，然后更新它。

**清单12.8 将项目标记为已打包：./app/component/Application.js**

class Application extends Component { constructor(props) { // … } componentDidMount() { // … } fetchItems() { // … }

addItem(item) { // … }

markAsPacked(item) {

this.props

.database('items')

**.where('id', '=', item.id)**

**.update({**

**packed: !item.packed**

**})**

.then(this.fetchItems)

.catch(console.error);

****  查找具有正确ID的项目

将项目的packed列更新为其当前相反的状态

}

markAllAsUnpacked() { // … }

render() { // … }

}

export default Application;

我们找到id列中的值与刚刚单击的UI中的项的ID匹配的所有项。提示：只存在其中一个。然后，我们使用update( )方法将打包列更新为UI中项目的相反方向。如果此操作成功，我们将获取所有项目,包括我们新更新的项目并刷新UI。如果出现问题，我们会将该错误记录到控制台以进行调试。更强大的应用程序可以在此处实现回退，或者至少显示某种通知，提醒用户他们的更改无法成功完成。

我们现在可以改变一个项目的状态，但是所有这些项目呢？答案介于this.fetchItems( )和this.markAsPacked( )的实现之间。使用this.markAsPacked( )，我们发现该项目的ID与我们正在寻找的ID匹配，然后更新它。使用this.fetchItems( )，我们使用select( )来获取所有项目。要实现this.markAllAsUnpacked( )(译注:原this.markAllAsPacked( ) 不存在,应为this.markAllAsUnpacked( ))，我们将获得整个集合中的所有项目。

**清单12.9 将所有项标记为unpacked：./app/component/Application.js**

class Application extends Component {

constructor(props) { // … } componentDidMount() { // … } fetchItems() { // … } addItem(item) { // … } markAsPacked(item) { // … }

markAllAsUnpacked() {

this.props

.database('items')

**.select()**

**.update({ packed: false**

**})**

.then(this.fetchItems)

.catch(console.error);

**选择数据库中的所有项目**

**通过将其打包列设置为false来更新所有项目。**

}

render() { // … }

}

export default Application;

正如我之前提到的，这个实现大致结合了我们之前的两种方法。从数据库中选择所有项目允许您使用SQL批量进行更改。对于我们在更新单个项目时使用的.where( )方法，情况也是如此。它恰好是当您基于唯一标识符进行查询时，您希望最终只有一条记录。this.markAllAsPacked( )(译注:this.markAllAsPacked( ) 不存在,应为this.markAllAsUnpacked( ))可以更改为使用.where( )来查找打包设置为true的所有项目。我将此作为练习留给读者，因为我将在下一节讨论它。

###### *删除项目*

当我们的应用程序在每次重新加载时丢失了所有数据时，我们没有很多用户要求删除项目。但现在我们已经在我们实施持久存储的地方了。也许你已经决定不再需要使用自拍杆了，而且你想将它从列表中删除。

我们现在实现两个功能：删除单个项目的能力，以及删除所有未打包的项目的能力，其中可能包括我们的自拍杆，Furby和丑陋的毛衣。我们也需要UI元素，但让我们从这个数据库片开始。

**清单12.10 删除项目：./app/components/Application.js**

class Application extends Component {

constructor(props) {

super(props);

// Omitted for brevity…

**this.deleteItem = this.deleteItem.bind(this);**

**绑定this.deleteItem( )方法的上下文**

**this.deleteUnpackedItems = this.deleteUnpackedItems.bind(this);**

}

componentDidMount() { … } fetchItems() { … } addItem(item) { … } markAsPacked(item) { … } markAllAsUnpacked() { … }

**deleteItem(item) {**

**this.props**

**.database('items')**

**.where('id', item.id)**

**.delete()**

**.then(this.fetchItems)**

**.catch(console.error);**

**}**

**绑定this.deleteUnpackedItems( )方法的上下文**

查找与从UI中选择的项目的ID匹配的项目

使用delete( )方法从数据库中删除该项。

**deleteUnpackedItems() {**

**this.props**

**.database('items')**

**.where('packed', false)**

**.delete()**

**.then(this.fetchItems)**

**.catch(console.error);**

**查找packed属性设置为false的所有项目**

**}**

render() { … }

}

export default Application;

细心的读者可以看到我在这里使用的语法略有不同与this.markAsPacked( )。当您尝试根据相等性进行匹配时，可以使用此缩写来提供列的名称和您尝试匹配的值。在前面的示例中，我们提供了一个运算符。这种方法很强大，因为我们可以使用更复杂的逻辑进行过滤在this.deleteUnpackedItems( )中，我们使用where( )方法查找packed属性设置为false的所有项。

这一切都很好，但要知道它们是否有效有点困难，因为我们没有任何UI来触发这些方法。这与我们设置复选框以在打包状态和解压缩状态之间切换项目的方式非常相似，但那是整整一章。我们来看看这个过程。

我们将this.deleteItem( )方法传递给每个列表。反过来，每个列表都将此方法传递给列表中的项目。我们为每个项目添加一个复选框X按钮。当用户单击X按钮时，我们将对特定项的引用传递给从应用程序组件传递下来的this.deleteItem( )。这些新增功能为我们提供了触发我们之前实施的方法所需的一切。我们还在Mark All as Unpacked按钮下方添加了Remove Unpacked Items按钮。这个按钮要简单得多，因为它不需要传递该方法。

**清单12.11 传入删除方法：./app/component /Application.js**

class Application extends Component { constructor(props) { … }

componentDidMount() { … } fetchItems() { … } addItem(item) { … } markAsPacked(item) { … } markAllAsUnpacked() { … } deleteItem(item) { … } deleteUnpackedItems() { … }

render() {

const { items } = this.state;

const unpackedItems = items.filter(item => !item.packed); const packedItems = items.filter(item => item.packed);

return (

<div className="Application">

<NewItem onSubmit={this.addItem} />

<Items

title="Unpacked Items" items={unpackedItems} onCheckOff={this.markAsPacked} **onDelete={this.deleteItem}**

**将this.deleteItem( )传递给Unpacked Items**

/>

<Items

title="Packed Items" items={packedItems}

onCheckOff={this.markAsPacked} **onDelete={this.deleteItem}**

/>

**将this.deleteItem( )传递给Packed Items**

<button

className="button full-width" onClick={this.markAllAsUnpacked}> Mark All As Unpacked

</button>

<button

className="button full-width secondary" onClick={this.deleteUnpackedItems}>

**Remove Unpacked Items** 

</button>

</div>

);

}

}

**添加一个按钮来触发this.deleteUnpackedItems( )。**

export default Application;

此时，Remove Unpacked Items按钮应该完全正常。但是我们仍然需要继续传递this.deleteItem( )。下一步是将它作为Items组件中的onDelete属性接收，该组件为Packed Items和Unpacked Items提供支持，然后将其传递给每个单独的项目并引用特定项目。

**清单12.12 Items组件：./app/components/Items.js**

const Items = ({ title, items, onCheckOff, **onDelete** }) => { return (

<section className="Items">

<h2>{ title }</h2>

{items.map(item => (

<Item

key={item.id}

onCheckOff={() => onCheckOff(item)} **onDelete={() => onDelete(item)}**

{...item}

/>

))}

</section>

);

};

export default Items;

这里没有很多新东西。为了完整起见，我把它包括在内。我们几乎就在那里，下一步是将按钮添加到单个项目中。此按钮应该有一个单击事件，该事件触发从应用程序组件作为onDelete( )命令传入的this.deleteItem( )方法。

**清单12.13 Item组件：./app/component/Item.js**

使用对象解构来拉入onDelete( )属性

const Item = (({ packed, id, value, onCheckOff, **onDelete** }) => { return (

<article className="Item">

<label>

<input type="checkbox" checked={packed} onChange={onCheckOff} />

{value}

</label>

**<button className="delete" onClick={onDelete}>?</button>**

</article>

);

});

**添加按钮并将其click事件处理程序设置为从父项传入的onDelete( )函数**

最后一部分，删除功能现已到位。我们已经实现了从数据库中创建，读取，更新和删除项目的功能。如果用户退出Jetsetter，重新启动他们的计算机，然后重新启动应用程序，他们的项目将处于与他们离开时相同的状态。也许如此。

###### *将数据库存储在正确的位置*

为了方便起见，我让您将应用程序放在项目目录的根目录中。但这通常不是用户特定数据的来源。目前，该数据库由给定计算机上的所有用户共享，这当然令人困惑。

一方面，操作系统已经为我们解决了这个问题。用户指定了应存储特定数据的位置。但是，每个操作系统都以稍微不同的方式解决了这个问题。幸运的是，我们并不关心这一点，因为Electron保护我们。Electron app模块公开了一个名为getPath( )的方法，该方法计算出公共位置的特定于操作系统的路径。我们将查找userData路径，但您可以在（https://electron.atom.io/docs/api/app/#appgetpathname）中查看可用路径的完整列表。

**清单12.14 将数据库存储在适当的位置：./app /database.js**

**import \* as path from 'path'; ~~import { app } from 'electron';~~**

import {remote} from 'electron';

const {app} = remote;

const database = knex({ client: 'sqlite3', connection: {

filename: **path.join( app.getPath('userData'), 'jetsetter-items.sqlite'**

**)**

},

useNullAsDefault: true

});

**使用Electron的内置API，根据操作系统查找用户数据的正确路径**

在开发过程中为数据库提供唯一的名称

译注:删除线下面的那两行才是作者的最终代码

我倾向于提供存储在项目外部的文件唯一名称，以避免与我的机器上的另一个项目发生冲突。在品牌推广之前，所有Electron应用程序都被称为“Electron”。如果我给出一个独特的名称，如果它们恰好最终出现在引擎盖下的同一个文件夹中，则无关紧要。

### *I**ndexedDB*

您可以通过多种方式将数据存储在Electron应用程序中。我们之前讨论过SQLite，但您可以轻松使用NoSQL数据库，例如LevelDB。您甚至可以使用您写入和读取的JSON文件。

如果管理文件和重新编译依赖项似乎对您的应用程序有点多，您可以选择基于浏览器的存储。在第2章中，我们使用localStorage来跟踪我们的书签。localStorage很棒，但它有一些限制：一切都必须存储为字符串的大型JSON对象。每当我们想要读取或写入localStorage时，我们必须解析并重新保存对象。

在我们开始使用IndexedDB之前，我们应该涵盖一些术语。SQL数据库包含行和列的表。它们很像非常强大，互联的Electron表格。NoSQL数据库通常是键值存储,就像一个巨大的JavaScript对象。使用IndexedDB时，您将创建包含一系列键的数据存储。每个键指向一个对象。IndexedDB与SQLite的不同之处在于，键和值可以是任何有效的JavaScript类型,包括对象，数组，映射和集合。

与IndexedDB的所有交互都是异步的，这是有道理的。但是，在本机(原生)方面，IndexedDB使用事件来处理异步，这可能会使阅读和理解变得混乱。幸运的是，谷歌的开发者倡导者杰克阿奇博尔德已经做了自己的工作，用基于承诺的API（称为idb）包装基于事件的API。抽象是轻量级的，并不隐藏IndexedDB的任何内部工作。这是我个人推荐的，在您对基于承诺的API感到满意之后，它比基于事件的API更容易理解。您通常不需要库来使用IndexedDB，但如果您想使用基于promise的API，则需要导入idb。

###### *使用IndexedDB创建存储*

在这个例子中，我们首先打开jetsetter数据库。该数据库的名称完全是任意的。你可以将它命名为eggplant-parmigiana，如果这对你更有吸引力的话。我们还传递了数字1，它代表了我们数据库的版本。版本1是一个很好的起点。（没有类似SQLite的架构！）

**清单12.15 在IndexedDB中设置存储：./app/database.js**

打开jetsetter数据库的第1版

import idb from 'idb';

const database = idb.open('jetsetter', 1, upgradeDb => {

upgradeDb.createObjectStore('items', {

**为应用程序中的项创建一个存储**

**我们将使用对象的id属性作为键。**

keyPath: 'id', autoIncrement: true

});

});

**告诉IndexedDB代表我们自增长id的key**

版本控制非常重要，因为在Web应用程序中，每个用户都将其IndexedDB本地存储在浏览器中。如果更改代码的工作方式，如果用户使用旧版本的数据库，则可能会损坏用户的数据。

拥有版本号可以让您在更改数据库工作方式时提高该数字。在这种情况下，您可以迁移数据，以便在任何其他应用程序代码访问数据库之前与您所做的更改兼容。我们在本章中坚持使用一个版本的数据库，因为我们的数据非常简单。

通常，您有一个数据库。这类似于大多数中小型服务器端Web应用程序只有一个数据库，它们在给定环境（例如，生产）中使用的事实。但是你可能有很多存储。想想像SQL数据库中的表这样的存储。在此应用程序中，我们只有一个存储，但您可以在应用程序中为人员，位置或任何其他模型存储。

我们创建了一个名为items的存储，这是我们要存储所有商品的合理名称。我们还传递了两个选项：我们将keyPath设置为id，将autoIncrement设置为true。这是什么意思？这意味着如果我们传入一个带有id属性的项，它就会使用该属性作为键。如果我们不这样做，那么它会向对象添加一个并将其设置为键，每次递增数字以使键唯一。

###### *从IndexedDB获取数据*

当我们通过使用Knex.js实现SQLite来消除大量的乏味时，我们做了一些欺骗。idb是对IndexedDB的抽象，但它比Knex.js更轻量级。这意味着我们必须自己做更多的手工劳动。像localForage（https://github.com/localForage/localForage）这样的库提供了更高级别的IndexedDB抽象，但让我们坚持使用idb并创建我们自己的抽象。

以下是我们采用的方法：在./app/database.js中，我们创建一个对象，其中包含从IndexedDB获取所有项目，创建新项目，更新现有项目和删除项目的方法。然后我们在应用程序组件中使用这些方法，在我们去的时候替换对SQLite的调用。让我们从编写一个方法开始，从数据库中获取所有项目。

**清单12.16 获取IndexedDB中的所有项目：./app/database.js**

export default {

**getAll() {**

**return database.then(db => { return db.transaction('items')**

**.objectStore('items')**

**.getAll();**

**});**

**},**

**访问数据库**

获取存储中的所有项目

**启动一个事务并声明您将使用items存储**

**访问项目存储**

add(item) { },

update(item) { }, markAllAsUnpacked() { }, delete(item) { }, deleteUnpackedItems() { }

};

我首先导出一个对象，其中包含我们需要匹配SQLite功能的所有方法。大多数这些方法现在都是空的，但我们在完成本章时会填写它们。但是，我已实现了getAll( )方法。

我们首先访问数据库，类似于我们对Knex.js和SQLite的处理。接下来，我们开始一个事务。事务阻止数据库的多个更改同时发生。如果事务中出现某些事情，则会恢复事务中所做的所有更改。这种做法可以防止数据损坏。与IndexedDB的所有交互必须包含在事务中。进入我们的事务后，我们访问items存储并从中获取所有条目。

###### *将数据写入IndexedDB*

如果存储中没有任何项目，那么从IndexedDB获取所有项目并不是特别有用。让我们实现将项添加到数据库的功能。此过程类似于从数据库中读取一些重要的区别。从数据库读取时，当我们完成读取时，事务完成。修改数据库有点复杂。因此，我们需要更明确地确定事务何时完成。我们还需要让IndexedDB知道我们打算写入数据库。之后，将项目添加到数据库然后调用事务完成是一个问题。

**清单12.17向IndexedDB添加项目：./app/database.js**

export default {

getAll() { … }, **add(item) {**

return database.then(db => {

**使用items存储创建新的读/写事务**

**访问项目存储，并将项目添加到数据库**

const tx = db.transaction('items', 'readwrite');

tx.objectStore('items').add(item); return tx.complete;

});

},

update(item) { }, markAllAsUnpacked() { }, delete(item) { }, deleteUnpackedItems() { }

**返回已完成的事务**

};

当我们取出所有项目时，我们整齐地将这些承诺捆绑在一起。如果这段代码看起来有点复杂，请不要担心,它们很类似。我们需要返回我们在第一行创建的事务承诺，因此我们将其存储在变量中。在事务内部，我们将作为参数传递的项添加到items存储。如果我们还有其他工作要做，我们可以在这里做。最后，我们返回一个在事务完成时解析的promise。

**清单12.18 更新IndexedDB中的项目：./app /database.js**

export default {

getAll() { … },

add(item) { … }, update(item) {

**return database.then(db => {**

**const tx = db.transaction('items', 'readwrite'); tx.objectStore('items').put(item);**

**return tx.complete;**

**});**

},

markAllAsUnpacked() { }, delete(item) { }, deleteUnpackedItems() { }

};

更新数据库中的项类似于向数据库添加新项，我们使用put( )方法代替add( )方法。然而，更新所有这些更为复杂。首先，我们需要从数据库中获取所有项目。然后我们将每个的打包状态更改为false。最后，我们创建一个事务，我们更新数据库中的每个项目。

**清单12.19 将所有项标记为packed：./app/database.js**

export default {

getAll() { … },

add(item) { … },

update(item) { … }, markAllAsUnpacked() {

return this.getAll()

**从数据库获取所有的项目**

**将每个项目的打包状态设置为false**

.then(items => items.map(item => ({ ...item, packed: false })))

.then(items => {

**在数据库中创建一个更新items的事务**

return database.then(db => {

const tx = db.transaction('items', 'readwrite'); for (const item of items) {

tx.objectStore('items').put(item);

**迭代所有项目**

}

return tx.complete;

});

});

},

**完成事务**

**更新数据库中的项目**

delete(item) { }, deleteUnpackedItems() { }

};

如果单个更新失败，则整个事务将失败，并且所有项目都将返回其原始状态。使用事务的好处是它确保您不会最终处于一半数据已更新且一半保持不变的状态。事务是一种全有或全无的事情。

从数据库中删除项目与更新它们几乎相同。我们使用IndexedDB delete( )方法而不是update( )。如果你想要更聪明，你可能会提供这些实现的共享部分的抽象，但我作为你的作者的工作是更清楚明白而不是聪明。

**清单12.20 从IndexedDB中删除项目：./app/database.js**

export default {

getAll() { … },

add(item) { … },

update(item) { … }, markAllAsUnpacked() { … }, delete(item) {

**return database.then(db => {**

**const tx = db.transaction('items', 'readwrite'); tx.objectStore('items').delete(item.id);**

**return tx.complete;**

**});**

},

deleteUnpackedItems() { }

};

**使用delete方法从数据库中删除项目**

当我们实现markAllAsUnpacked( )时，我们映射了所有项目，无论其状态如何。在实现deleteUnpackedItems( )时，我们需要更加小心。我们的工作是只删除那些打包属性设置为false的那些。

**清单12.21 从IndexedDB中删除所有解压缩的项目：./app/database.js**

export default {

getAll() { … },

add(item) { … },

update(item) { … }, markAllAsUnpacked() { … }, delete(item) { … }, deleteUnpackedItems() {

**return this.getAll()**

**.then(items => items.filter(item => !item.packed))**

**.then(items => {**

**return database.then(db => {**

**const tx = db.transaction('items', 'readwrite'); for (const item of items) {**

**tx.objectStore('items').delete(item.id);**

**} **

**return tx.complete;**

**});**

**});**

**过滤掉所有已打包的项目，并返回仍未解压缩的项目数组。**

**删除每个项目**

}

};

与SQLite不同，IndexedDB不提供对查询数据库的支持，考虑到它是NoSQL数据库，这有一定的意义。使用NoSQL数据库时，您可以采取一些有趣的方法，例如将数据存储在具有不同索引的多个位置以便快速检索，但这有点超出了本书的范围。

我们选择了最简单的方法,并且非常适合我们的数据集，即从数据库中获取所有项目并仅查找符合我们标准的项目。我们再次在事务中这样做。如果删除任何这些项失败，数据库将返回到我们触发此方法之前所处的状态。

###### *将数据库连接到UI*

使用最后一个方法，我们需要将所有新数据库方法连接到UI。通过在上一节中构建数据库方法，我们已经从SQLite实现转换为基于IndexedDB的实现，相对简单。主要区别在于我们不查询数据库中的项目。

**清单12.22 更新应用程序组件以使用IndexedDB：./app/components/Application.js**

fetchItems() { this.props

.database

.getAll()

.then(items => this.setState({ items }))

.catch(console.error);

}

addItem(item) { this.props.database.add(item).then(this.fetchItems);

}

deleteItem(item) { this.props

.database

.delete(item)

.then(this.fetchItems)

.catch(console.error);

}

markAsPacked(item) {

const updatedItem = { ...item, packed: !item.packed }; this.props

.database

.update(updatedItem)

.then(this.fetchItems)

.catch(console.error);

}

markAllAsUnpacked() { this.props

.database

.markAllAsUnpacked()

.then(this.fetchItems)

.catch(console.error);

}

deleteUnpackedItems() { this.props

.database

.deleteUnpackedItems()

.then(this.fetchItems)

.catch(console.error);

}

译注:

1. sqlite实现的项目,仅在deepin平台下正常工作,在Windows及MacOS平台,似乎和node10版本不兼容,不能正常工作;
2. 在5.0.x下sqlite项目也不能正常工作,另外热更新配置不可用!
3. 5.0.x下项目仅支持css后缀的样式表文件!

### *本章小结*

* + - * 编译的依赖项仅适用于Node编译的V8引擎版本。
      * 系统上的Node版本可能与Electron捆绑的Node版本不同。
      * 通常，当您使用npm install或yarn install安装依赖项时，依赖项是针对Node的系统版本而不是Electron的系统版本构建的。
      * electron-rebuild会检查已安装的依赖项，并针对与Electron一起打包的Node版本重建它们。
      * 用户数据应存储在文件系统的适当位置。这在文件系统之间有所不同，但Electron提供了一个名为app.getPath( )的有用抽象，可以代表您确定正确的路径。
      * SQLite是本机(原生)应用程序的常见选择，因为它将数据存储在文件中，而不是要求安装和运行数据库服务器。
      * IndexedDB是一种流行的基于浏览器的选项，用于在客户端上存储用户数据。它是Chromium提供的NoSQL数据库。

*使用Spectron测试应用*

***本章包括***

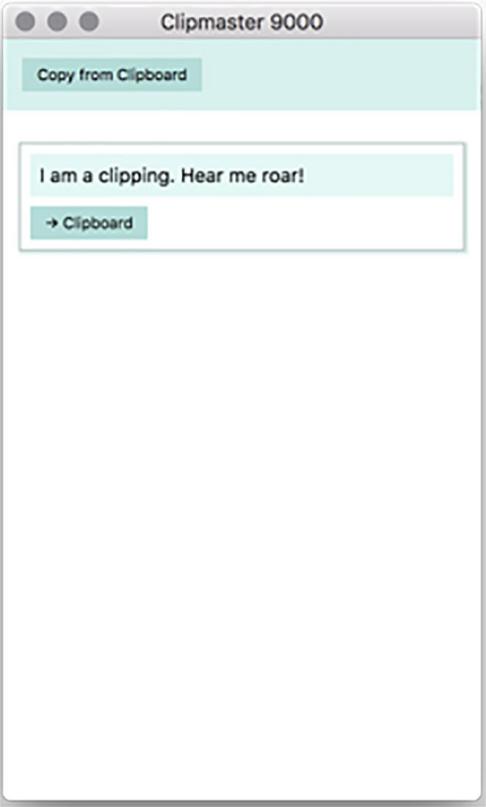
* 使用Spectron测试Electron应用
* 了解Spectron，WebdriverIO和Selenium WebDriver之间的关系
* 在集成测试中控制Electron API

在本书的这个时候，我们已经构建了许多Electron应用程序。测试这些应用程序以确保它们按预期工作是一个手动过程。进行更改，启动应用程序，然后单击以确保一切正常。这适用于小型应用程序，但这种方法在较大的应用程序中可能会变得乏味。如果应用程序的某个部分中的代码更改破坏了其他地方的某些功能，该怎么办？

在这些情况下，自动化测试很有帮助。让计算机完成无聊，重复的工作，同时专注于为客户带来价值的功能。听起来不错，但我们怎么做呢？ 在传统的Web应用程序中，您可能会将Selenium WebDriver之类的内容指向您的网站，对网站的行为方式做出一些断言，然后让它点击并确认一切都按预期工作。

这很不错，但是我们遇到了一些问题：我们的Electron应用程序本身没有Web地址，但它们支持API，这些API为我们提供了前所未有的对底层操作系统的访问，这些操作系统通常不受Web浏览器的支持。在这一点上，这个棘手的问题有一个简单的解决方案，你不应该感到惊讶。Electron团队支持一个名为Spectron的项目，该项目允许开发人员为我们的Electron应用程序编写集成测试。

在本章中，我们编写了一个特殊版本的Clipmaster 9000的集成测试，它不在菜单栏中，如图13.1所示。我们测试所有核心功能，并在此过程中学习一些技巧。首先，您需要将Spectron和测试运行器添加到package.json中。我选择使用Mocha试图与官方文档保持一致，但您可以使用Jest，Karma，Jasmine或任何其他让您开心的测试运行器。我已将其包含在示例存储库（http://mng.bz/UxHY）中，但如果你是一个人在工作，你可以运行npm install——save-dev spectron mocha。



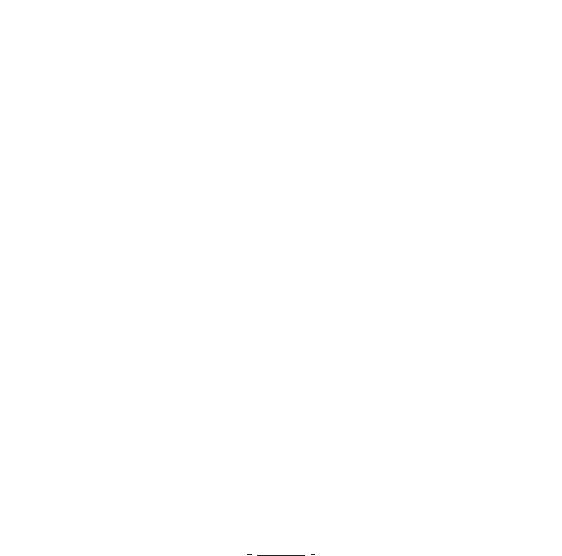
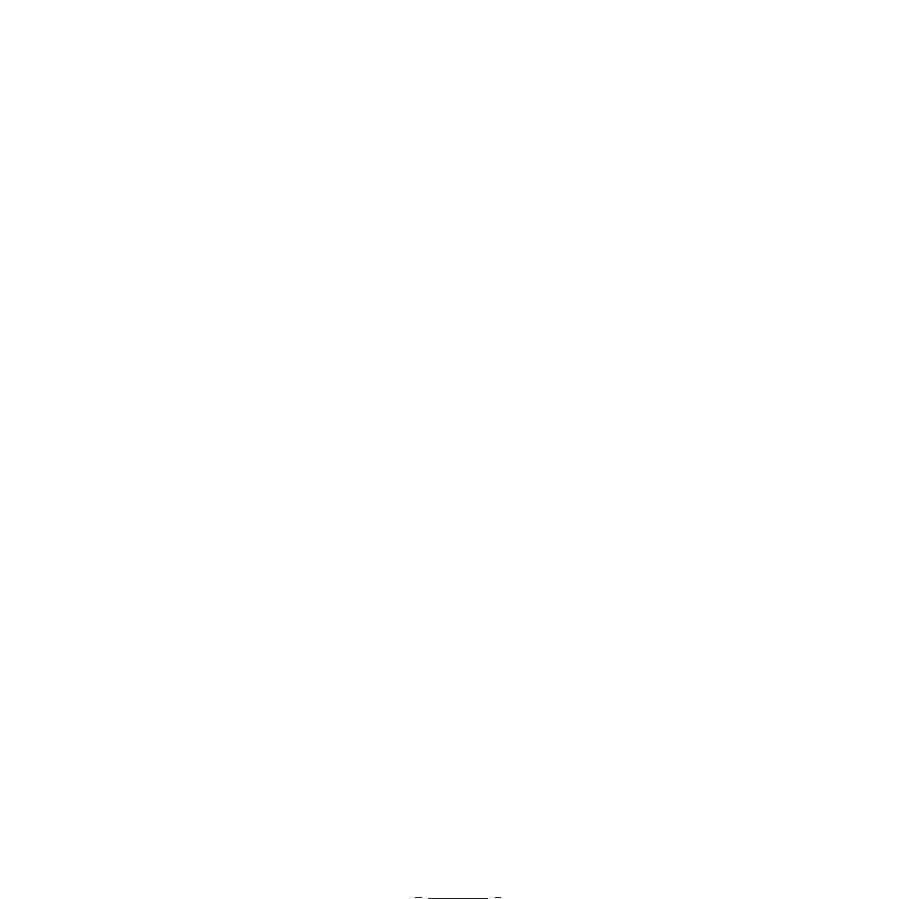
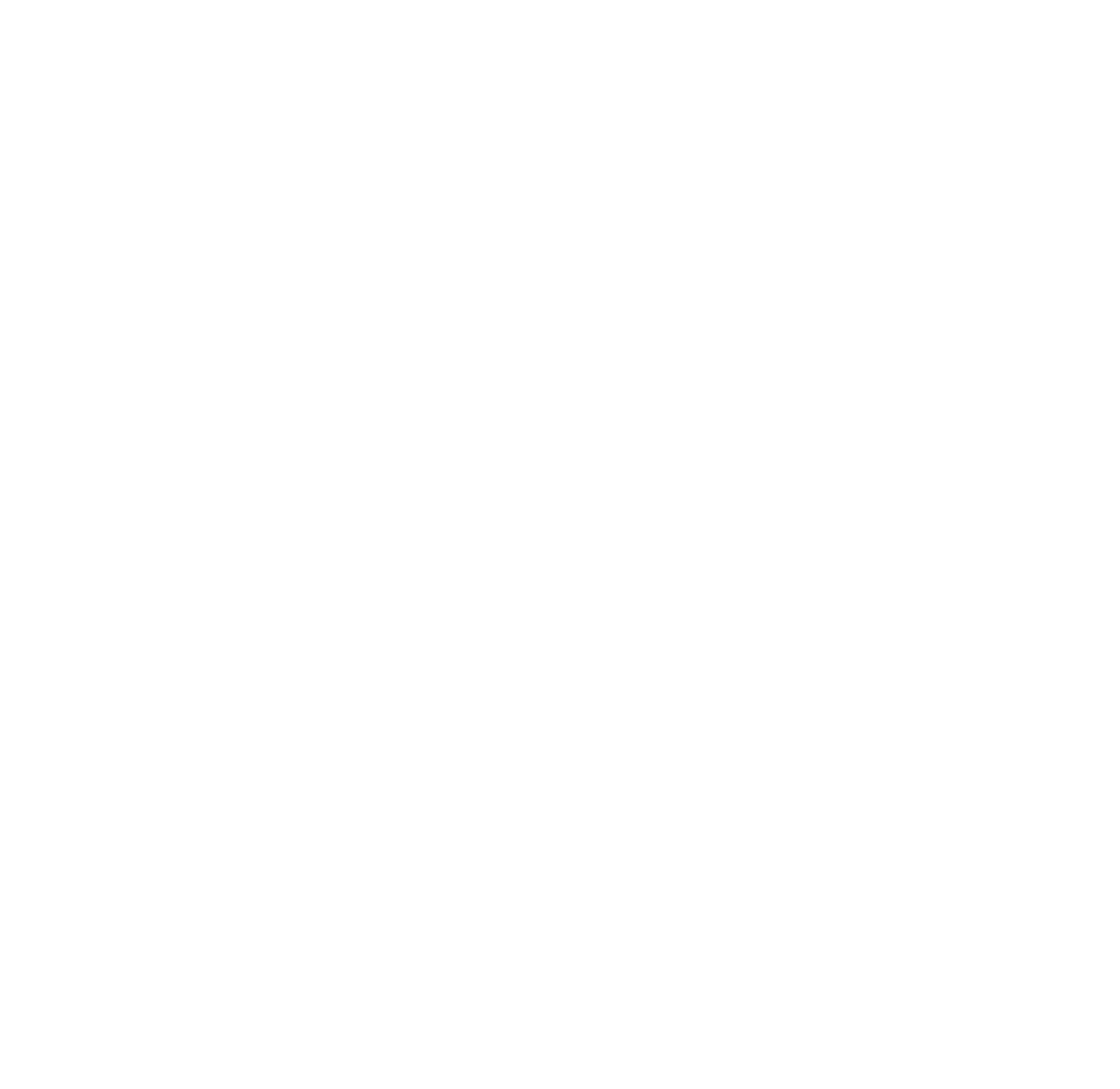
**图13.1 Clipmaster 9000的简化版本，它不在菜单栏中。**

首先，了解Spectron，WebdriverIO和Selenium WebDriver之间的关系非常重要，因为我们在给定的测试中使用来自它们间的多个方法。让我们从底部开始，继续前进。Selenium WebDriver允许开发人员编写控制Web浏览器的测试，以便他们可以从用户的角度测试他们的应用程序,而不是单独测试，它单独执行一段代码。Selenium WebDriver是一个与语言无关的库，通常由一个库包装，该库为给定的编程语言提供本机API绑定。

***介绍Spectron* 245**

进入WebdriverIO，它包含了一个令人愉快的JavaScript API的Selenium WebDriver，并且在我们的案例中可以很容易地在Node.js或Electron应用程序中使用。但正如我在本章开头指出的那样，Electron应用程序与传统的Web应用程序有很大的不同，以及许多额外的功能和功能。Spectron包装WebdriverIO，如图13.2所示，并允许我们在为Electron应用程序定制的环境中访问此功能。

允许您将WebdriverIO与Electron应用程序一起使用的附加功能。



Spectron

**绑定Selenium，允许您使用JavaScript控制它。**

Webdriver.io

Selenium

驱动像Chrome这样的网络浏览器。这允许你自动化测试。

**图13.2 Spectron是WebdriverIO的包装器，WebDriver是Selenium的包装器。**

我们不需要在本章中触及Selenium WebDriver。WebdriverIO代表我们负责控制Selenium WebDriver。我们从Spectron和WebdriverIO中获取方法。实际上，您并不需要考虑两者之间的界限，但是Spectron的一些方法会委托给WebdriverIO。最重要的是，当需要查找给定方法的文档时。我将确保在我们委托给WebdriverIO时指出，而不是使用属于Spectron本身的方法。

### *介绍Spectron*

Spectron可以轻松启动我们的应用程序并从我们的测试套件控制其UI。使用Spectron的主要方法是创建一个Application实例。该对象包含许多子对象，允许我们访问应用程序的不同部分，如图13.2所示

rendererProcess: 渲染器进程中的Node进程对象。

mainProcess:主进程中Node的进程对象。

webContents:Electron的WebContents API。

new Application()

browserWindow: 当前聚焦(活动)的浏览器窗口。

electron: Electron的渲染器进程API。

client: 底层的WebdriverIO实例。

**图13.3 Spectron API**

Spectron分为以下几个部分：

1 client是底层的WebdriverIO实例并公开其所有方法。（这不完全正确，正如我在下一节中解释的那样。）当您想要搜索特定节点的DOM或触发点击事件时，可以使用此选项。

2 electron是Electron的渲染器进程API。这里提供了在渲染器进程中使用require（'electron'）时可用的任何内容。因此，您可以使用electron.remote以与在渲染器进程中相同的方式访问主进程。

3 browserWindow是一个方便的别名，可以访问应用程序中当前聚焦(激活)的浏览器窗口。它相当于electron.remote.getCurrentWindow( )。此外，当您想要截取当前活动的浏览器窗口并将其保存到桌面时，browserWindow.capturePage( )非常有用。当您尝试诊断测试失败的原因时，这可能很有用。

4 webContents是Electron的webContents API的别名，它可用于获取有关您的应用程序的信息或控制正在执行的浏览器。它是electron.remote.getCurrentWebContents( )的别名。在本书中，我们使用此API将HTML页面加载到新创建的BrowserWindow实例中或切换开发人员工具。API提供对前进和后退按钮，打印页面，设置缩放级别和重新加载页面等浏览器功能的访问。它会在浏览器加载时发出事件，这对测试很有用。它还提供webContents.savePage( )方法，该方法允许您将当前加载的页面保存为文件系统上的HTML文件，以便在测试未按预期运行时进一步检查。

5 mainProcess是主进程中Node的进程全局的别名。它相当于electron.remote.process。重要的是要注意mainProcess不是Electron的主进程API的别名，但可以使用electron.remote访问它。如果您需要访问环境变量或启动时传递给Electron的参数，mainProcess非常有用。

6 rendererProcess类似于mainProcess，并提供对渲染器进程的global.process对象的访问。同样，这对于读取环境变量很有用。如果您需要Electron的渲染器处理API，请改用electron。

Application实例的前两个属性提供对浏览器API以及Electron自己的API的深入访问。通过这种组合，我们可以随时深入地了解和控制应用程序的几乎所有内部工作。这种访问使得Electron应用程序的测试变得简单易用。browserWindow，webContents，mainProcess和rendererProcess属性为前两个属性提供了方便的别名。

### *熟悉Spectron和WebdriverIO*

在上一节中，我提到当我说client属性委托给WebdriverIO时，我有点不知所措。这种说法在技术上是正确的,但这不是全部真相。client还有额外的Electron专用方法，如图13.4所示，使您的测试体验更加愉快。

(All of the Webdriver.io API)

windowByIndex(index)

waitUntilWindowLoaded()

waitUntilTextExists(selector, text)

app.client

getWindowCount()

getSelectedText()

getRenderProcessLogs()

getMainProcessLogs()

**图13.4 Spectron为WebdriverIO API添加了许多实用工具方法。这是图13.3中客户端的分解图。**

如果您以前编写过集成测试，有些方法看起来很熟悉。例如，像getSelectedText( )这样的方法考虑到您的应用程序可能有多个窗口,这在传统的Web应用程序中是不可能的。其他的是测试Electron应用的独特之处。我现在向您介绍这些方法作为参考。我们在后面的章节中使用了很多。

getMainProcessLogs( )和getRenderProcessLogs( )返回一个promise，该promise解析为已记录到相应控制台的消息数组。只要调用任一方法，它就会清除来自控制台的消息。这意味着后续调用仅包含自上次调用方法以来已记录到控制台的消息。

getWindowCount( )返回一个解析为整数的promise，毫无疑问,表示当前在应用程序中打开的窗口数。如果您正在为Fire Sale等应用程序编写测试，则此值可能很有用，该应用程序具有允许用户打开另一个窗口的按钮。app.browserWindow引用当前活动的窗口。app.client.windowByIndex( )允许您专注于备用窗口。

waitUntilWindowLoaded( )接受给定的毫秒数作为参数并返回一个promise(译注:原文大写的U前面还有小写u,删去!)。如果窗口在指定的时间内加载，则promise将解析。否则承诺被拒绝。此方法允许您延迟测试的执行，直到您确认窗口已实际加载为止。

waitUntilTextExists( )接受三个参数：一个选择器，一个文本字符串，以及在放弃之前要等待的可选毫秒数。它返回一个在选择器包含文本时解析的promise。或者，如果5秒或提供的毫秒数通过且文本没有出现，它将拒绝。此设置对于UI组件非常有用，其中内容可能在加载窗口后不久异步进入。正常测试会失败，因为它们会在异步内容加载之前立即运行。

### *设置Spectron和测试运行器*

Spectron提供启动Electron应用程序并从测试中控制它的能力，但它不提供运行测试或验证应用程序按预期工作的框架。如前所述，我们使用了Clipmaster 9000的简化版本。代码的起点可以在https://github.com/electron-in-action/clipmaster-9000-spectron上找到。

这是一件非常好的事情，因为它允许您使用您喜欢的任何测试运行器和断言库。只需要在您选择的测试运行器中使用Spectron，我们将在清单13.2中看到它。在本章中，我使用Mocha和Node的内置断言库，但如果您愿意，您当然可以使用Chai和断言库或Jest，Karma，Jasmine或任何其他框架。Spectron对此并不反对。相反，它允许您从您选择的框架中驱动您的Electron应用程序。

我已经将Mocha作为示例项目的开发依赖项之一。正如我之前提到的，如果你正在开发自己的应用程序并继续使用，你可以从命令行运行npm install --save-dev spectron mocha来安装Spectron和Mocha。这会将mocha命令行工具安装到您的node\_modules目录中，该目录可以通过键入来运行

./node\_modules/.bin/mocha。这可能很乏味。将它指定为使用npm test运行要容易得多。所以，让我们将它添加到我们的package.json中的项目脚本中。（如果您已经从GitHub克隆了存储库，那么这应该已经为您设置了。）

**清单13.1 将Mocha设置为测试脚本：./ package.json**

"scripts": {

"start": "electron .",

"test": "mocha"

}

**设置npm test以运行本地安装的Mocha版本**

现在每当我们运行npm test时，Jest都会通过查找带有\* .test.js后缀的所有文件来运行我们的测试套件。您也可以将此后缀用于您要编写的任何单元测试--Mocha也会运行这些测试。在较大的应用程序中，您可以为所有测试创建一个文件夹，或将它们与实现文件一起包含在内。为了简单起见，我将使用一个名为test.js的文件来保存我们应用程序的所有测试。

让我们首先检查存储库中提供的样板，以便在每次测试之前启动我们的应用程序，并在单个测试运行后停止应用程序。这为我们提供了每个测试的应用程序的新实例，从而无需清除以前测试中的任何副作用和剩余状态。

**清单13.2 用测试运行器编写Spectron：./test /spec.js**

**拉入Spectron的应用程序驱动程序**

**导入Node的内置断言库**

const assert = require('assert'); const path = require('path');

**带来Node的帮助工具，用于处理文件路径**

需要Electron。这将使我们能够访问本地安装的开发版本的Electron。

const Application = require('spectron').Application; const electronPath = require('electron');

const app = new Application({

**指向应用程序本身的根目录作为应用程序的起点**

path: electronPath,

args: [path.join( dirname, '..')],

});

describe('Clipmaster 9000', function () { this.timeout(10000);

beforeEach(() => {

**告诉Spectron的应用程序使用本地安装的Electron开发版本**

**增加Mocha的默认超时，因为启动应用程序可能需要一段时间**

return app.start();

});

**在每次测试之前启动应用程序**

afterEach(() => {

if (app && app.isRunning()) { return app.stop();

}

});

});

**每次测试后停止应用程序**

我们需要的第一件事是我们想要测试的Electron应用程序的位置。从理论上讲，您可以将Spectron指向计算机上的任何基于Electron的应用程序并以编程方式控制它们，但我将其视为读者的狡猾行为。我将成为一个无聊的规则追随者，并获得我们在使用npm install安装我们的依赖项时从npm下拉的本地安装的Electron版本的路径。

我们希望在每个ourtests中访问保存应用程序实例的变量，因此我们在describe( )块中声明它。在每次测试之前，我们创建一个应用程序实例并将其分配给app变量。新的Application( )构造函数获取Electron二进制文件的路径，以及我们通常键入npm start的目录的位置。这应该是包含package.json的目录，该目录引用包含主进程代码的JavaScript文件。创建实例后，我们启动应用程序。我们现在已准备好进行测试,尽管我们还没有编写测试用例。但想象一下，我们有一个测试，它已经运行。我们检查是否为app变量分配了值，并且应用程序仍在运行。（它可能已经崩溃。）如果这两种情况都是真的，那么我们停止应用程序并转到下一个测试用例，我们再次经历这个过程。

### *使用Spectron编写异步测试*

我们已经讨论过我们的测试工具。我们已经设置好了。现在让我们坐下来写一些测试。在本章的过程中，我们将检查Clipmaster

* + - 启动时显示初始窗口
    - 显示“Clipmaster 9000”作为标题
    - 启动时没有打开Developer Tools
    - 有一个带有文本Copy from Clipboard的按钮
    - 应用程序启动时，UI中没有显示任何剪辑
    - 一次按下Copy from Clipboard按钮时，包含一个剪辑
    - 单击剪辑的Remove按钮后，成功删除剪辑
    - 创建新剪辑时具有正确的文本单击Copy按钮时，剪辑文本将写入剪贴板

我们将从一个简单的测试开始：启动应用程序，并确认此特殊版本的Clipmaster 9000创建一个浏览器窗口。Spectron和WebdriverIO使用异步的，基于承诺的API来控制Electron和Web应用程序。大多数方法返回promises，可以链接。正如我在本书中反复提到的那样，Electron为我们提供了现代版本的Node和Chromium。我们经常拥有最新，最好的浏览器和语言功能。这包括用于处理异步API的async/await语法，其方式与同步代码一样容易。使用此语法，您不必担心传递回调或链接承诺。

如清单13.3所示，我们在箭头函数之前使用async关键字来表示我们将在此函数中使用async/await语法。await关键字暂停，直到app.client.getWindowCount( )已解析，然后将结果分配给count变量。完成此操作后，我们希望计数等于1。Mocha希望异步测试在完成后返回一个承诺，因此您需要确保使用return关键字为您的期望添加前缀。在第一杯咖啡之前写这一章时，你的作者多次没有这样做。他以他的时间和理智为这种疏忽付出了沉重的代价。

**清单13.3 编写测试来计算窗口数：./ test/spec.js**

it('shows an initial window', async () => {

const count = await app.client.getWindowCount(); return assert.equal(count, 1);

});

**应用启动后; 得到所有窗口的数量。**

验证此版本的Clipmaster仅创建一个窗口。

这个测试很简单，使用promises的传统实现看起来并没有太大的不同，但是更多涉及的测试需要更长的承诺链，这些承诺链可能会变得冗长乏味，或者混淆不清。

###### *等待窗口加载*

在许多测试中，您必须等待窗口加载HTML，CSS和JavaScript，然后才能继续测试应用程序。Electron快速加载HTML，但它不是即时的。我们经常使用前面描述的app.client.waitUntilWindowLoaded( )方法等待应用程序完全加载后才能继续前进。让我们编写一个测试，确认窗口的标题是应用程序的名称。

**清单13.4 编写测试以验证窗口标题：./test/spec.js**

在窗口加载后**,**

it('has the correct title', async () => {

**获取窗口的标题**

const title = await app.client.waitUntilWindowLoaded().getTitle(); return assert.equal(title, 'Clipmaster 9000');

});

**验证窗口的标题是否符合我们的预期**

Spectron的许多方法都是可链接的，链中的每个后续方法在从前一个方法返回的promise得到解决后被调用。在前面的例子中，我们等待窗口完成加载，然后获取窗口的标题并将其存储在title变量中; 在异步操作完成之后，我们可以验证窗口的标题是我们期望的。

###### *测试Electron BrowserWindow API*

传统上，Selenium和WebdriverIO等集成测试工具将浏览器指向网页并与之交互。他们通常没有像开发人员那样更多地访问浏览器本身的内部。Electron允许开发人员在窗口首次加载时以编程方式打开开发人员工具。也就是说，如果我们不小心忘记删除此代码并将其发送给用户，那将会很糟糕。让我们编写一个测试，验证加载应用程序时是否打开了开发人员工具。

**清单13.5 测试开发人员工具是否未打开：./test/specpec .js**

检查开发者工具是否打开

it('does not have the developer tools open', async () => {

const devToolsAreOpen = await app.client

**.waitUntilWindowLoaded()**

**获取对浏览器窗口实例的引用**

**.browserWindow.isDevToolsOpened();**

return **assert.equal(devToolsAreOpen, false);**

});

**验证开发人员工具是否未打开**

**i**

isDevToolsOpened( )方法适用于所有BrowserWindow实例。加载应用程序后，我们将获取当前浏览器窗口并询问它是否打开了开发人员工具。我们希望这是false的。如果true，则测试将失败。

###### *使用Spectron遍历和测试DOM*

除了测试Electron API之外，我们几乎肯定希望像传统应用程序一样测试应用程序的UI。让我们从一个简单的例子开始，我们在页面中搜索特定元素并验证它是否具有我们期望的副本。

**清单13.6 测试Copy from Clipboard按钮的内容：./test/spec.js**

it('has a button with the text "Copy from Clipboard"', async () => {

const buttonText = await app.client

.getText('#copy-from-clipboard');

return assert.equal(buttonText, 'Copy from Clipboard');

});

**使用WebdriverIO API从页面上的DOM节点获取文本**

getText( )方法由WebdriverIO提供，而不是Spectron。它接受一个选择器作为参数，找到与该选择器匹配的第一个节点，并返回一个解析为该节点的文本内容的promise。最后，我们确认该文本是我们期望的。

这有效，但更复杂的遍历呢？ 当应用程序启动时，剪辑列表是空的,记住这是在我们知道如何在Electron应用程序中保存数据之前。每次用户单击Copy from Clipboard按钮时，都应该向页面添加新剪辑。我们怎样才能测试这些场景？让我们从第一种情况开始：当应用程序启动时，页面上不应该有任何剪辑。

**清单13.7 测试UI的初始状态：./test/spec.js**

**等待窗口已经加载其内容**

it('should not have clippings when it starts up', async () => {

await app.client.waitUntilWindowLoaded();

const clippings = await app.client.$$('.clippings-list-item'); return assert.equal(clippings.length, 0);

使用WebdriverIO的API查找页面上的所有剪贴列表项

});

**验证默认情况下没有剪切列表项**

Webdriver公开$和$$方法，它们分别别名为document.querySelector和document.querySelectorAll。我们使用$$来选择页面上的所有剪切列表项，它返回一个NodeList对象。理想情况下，页面上没有剪切列表项，并且该NodeList的长度为0。

到目前为止，我们已经检查并验证了应用程序启动时的状态,这是一项重要的任务，但通常只是我们为给定应用程序编写的一小部分测试。我们已确认应用程序启动时页面上没有剪辑。按理说，当用户添加剪辑时，应将其添加到UI中。我们已经知道如何检查与页面上给定选择器匹配的元素数量。但是你如何添加剪辑呢？ 单击Copy from Clipboard按钮。我们需要弄清楚如何编程WebdriverIO来点击页面上的特定元素。

**清单13.8 测试点击交互：./test/specs.js**

it('should have one clipping when the "Copy from Clipboard" button has been

pressed', async () => {

在Copy from Clipboard按钮上触发单击事件

await app.client.waitUntilWindowLoaded();

await app.client.click('#copy-from-clipboard');

// **查找页面上的所有剪辑**

const clippings = await app.client.$$('.clippings-list-item');

return assert.equal(clippings.length, 1);

});

**验证现在有一个剪辑**

幸运的是，app.client提供了一个click( )方法，它完全符合您的预期：它会触发与提供的选择器匹配的节点上的单击事件。当我们单击Copy from Clipboard按钮时，我们希望在页面中添加剪辑。先前的测试验证在单击“从剪贴板复制”按钮后页面上有一个剪辑。

反面怎么样？如果您还记得，也可以从UI中删除剪辑。但是有一个问题,只有当用户的光标悬停在裁剪上时，才会出现Remove按钮。我们不能简单地查询Remove按钮，因为它还没有在页面上。相反，我们需要控制光标并将其移到剪辑上以显示Remove按钮。

**清单13.9 在测试中移动光标：./test/specpec .js**

it('should successfully remove a clipping', async () => {

**单击Remove按钮**await app.client.waitUntilWindowLoaded(); await app.client

.click('#copy-from-clipboard')

.moveToObject('.clippings-list-item')

.click('.remove-clipping');

**将光标移动到剪辑以触发Remove按钮**

const clippings = await app.client.$$('.clippings-list-item'); return assert.equal(clippings.length, 0);

});

**验证剪辑不再在页面上**

单击Copy from Clipboard按钮后，我们使用moveToObject( )方法将光标移动到新剪辑上，这会导致Remove按钮出现。我们单击Remove，然后验证剪辑不再在页面上。

**但史蒂夫，每当我进行这项测试时，我都会看到一个弃用警告！**

不幸的是，看到警告是正常的。当您在清单13.9中运行测试时，您会看到与moveTo相关的弃用警告，因为它与最新版本的Firefox不兼容,这个问题与我们无关。问题是尽管moveTo已被弃用，但在撰写本文时尚无替代品。关于这一点，GitHub上存在许多未解决的问题。可以在https://github.com/webdriverio/webdriverio/issues/2076找到一个很好的例子。你不是唯一一个看到这些警告而感到沮丧的人。

要禁止这些警告，您可以在创建Spectron应用程序时配置WebdriverIO的选项，如下所示：

const app = new Application({ path: electronPath,

args: [path.join( dirname, '..')], **webdriverOptions: {**

**deprecationWarnings: false**

**}**

});

###### *用Spectron控制Electron的API*

在这一点上，我们已经测试过我们可以在页面中添加和删除剪辑，但我们仍然有一个主要的盲点：我们不知道剪辑是否显示正确的文本。如果您能够在之前的测试期间眯着眼睛看应用程序，您可能已经注意到添加的剪辑包含当时剪贴板上的内容。这将是棘手的，因为我们的测试应该与外界隔离。

我们需要做的是使用Electron的剪贴板模块将文本写入剪贴板，然后验证提供的文本实际上是UI中显示的内容。幸运的是，Spectron让我们可以访问Electron的所有API。这意味着我们可以操作剪贴板上当前的内容作为测试的一部分，并确认在剪辑添加到页面时它是在UI中找到的内容。

**清单13.10 在测试中访问Electron API：./test/spec.js**

访问Electron的API以将文本写入系统的剪贴板

it('should have the correct text in a new clipping', async () => {

await app.client.waitUntilWindowLoaded();

await app.electron.clipboard.writeText('Vegan Ham'); await app.client.click('#copy-from-clipboard');

const clippingText = await app.client.getText('.clipping-text');

return assert.equal(clippingText, 'Vegan Ham');

});

**验证新剪辑是否包含剪贴板的内容获取已创建的新剪辑的文本**

剪贴板模块在主进程和渲染器进程中都可用，因此不需要使用electron.remote。在单击Copy from Clipboard按钮之前，我们将文本写入剪贴板。然后我们单击按钮并验证剪辑的文本正是我们写入剪贴板的内容。

这涵盖了Clipmaster 9000功能的一半。硬币的另一面是，当点击剪辑的剪贴板按钮时，用户应该能够将剪辑复制回剪贴板。为了测试这一点，我们需要在我们之前测试中所做的工作的基础上进行。单击Copy from Clipboard按钮并将其添加到页面后，我们将剪贴板的内容更改为其他内容，然后单击剪贴板按钮，从剪贴板中读取，并验证它是否已恢复为我们添加到Clipmaster 9000的原始文本。

**清单13.11 应用程序写入剪贴板的测试：./test/spec.js**

使用Electron的API将文本写入剪贴板

**单击Copy from Clipboard按钮**

单击应据称将文本复制回剪贴板的按钮

it('should write the clipping text to the clipboard', async () => {

await app.client.waitUntilWindowLoaded();

await app.electron.clipboard.writeText('Vegan Ham'); await app.client.click('#copy-from-clipboard');

await app.electron.clipboard.writeText('Something different'); await app.client.click('.copy-clipping');

const clipboardText = await app.electron.clipboard.readText(); return assert.equal(clipboardText, 'Vegan Ham');

**将其他一些文本写入剪贴板**

});

**验证文本现在是剪辑的内容，而不是我们在测试中手动写入剪贴板的文本**

**从剪贴板中读取文本**

通过这个最终测试，我们可以验证所有UI元素是否按预期工作和运行。我们能够结合使用WebdriverIO和Electron API从多个角度测试应用程序。本章末尾的代码可以在存储库的已完成示例分支中找到。

译注:

1. 关于升级到5.0.x,一切正常完美运行:

A.先删除现有的依赖,然后安装最新的依赖:

npm remove electron menubar request --save

npm remove eslint eslint-plugin-import mocha sinon spectron --save-dev

npm install electron menubar request --save

npm install eslint eslint-plugin-import mocha sinon spectron --save-dev

cnpm install

B.修改main.js与node集成:

mainWindow = new BrowserWindow({

height: 500,

width: 300,

show: false,

**webPreferences: {**

**nodeIntegration: true**

**}**

});

2.测试中有不合理之处是,有的测试依赖于前面的测试,而每次都初始化销毁,比较浪费时间!

### *本章小结*

* + - * Spectron是一个官方支持的Electron应用测试库。
      * Spectron包装WebdriverIO，它为Selenium提供Node.js绑定。
      * 所有Electron的API均可在Spectron中获得。
      * Spectron不提供自己的测试运行或断言库。相反，它允许您选择要使用的那个。
      * Electron支持async/await语法，极大地简化了编写异步代码的过程。

# *第三部分*

*部署Electron应用程序*

在第二部分，我们构建了许多应用程序，但每个应用程序都有一个共同的缺陷：我们的用户需要使用命令行导航到相应的目录并安装Node.js来启动应用程序。这不完全理想。我相信你已经猜到了，事情并非如此。这不是我今天早上开始Slack的方式。第三部分涵盖了打包和分发我们精彩应用的复杂性。

第14章直接介绍了我在上一段中说明的问题，向我们介绍了两种不同的方法来打包我们的应用程序以进行分发。我们将在此过程中创建Fire Sale和Jetsetter的独立版本。在第15章中，我将解决一个可能会或可能没有出现在你心中的恐惧：一旦我们将它们发布到野外，我们怎么知道我们的应用程序是否按预期运行？我们将与Electron的内置崩溃记录器联系Fire Sale。我们还将设置一个简单的服务器来接收和跟踪崩溃报告。崩溃完全不是我们的用户可能遇到的唯一问题。我们还将构建自己的捕获未捕获异常的能力，并将它们报告给我们的服务器。这超出了Electron开箱即用的范围，但最终实现起来相当简单。

第16章讨论如何为Mac App Store准备好应用程序并做好准备。您绝不必沿着这条路走下去，但如果您决定要从Electron应用程序中获利，它确实可以解决您可能遇到的一些问题。

*构建部署应用程序*

***本章包括***

* 包装您的分发应用
* 为打包的应用程序设置自定义名称和图标
* 使用Electron混合物轻松包装含有转化代码的应用程序
* 在Electron档案中构建源代码（asar）

在本书中，我们已经构建了多个应用程序。我们以相同的方式启动它们：打开终端，导航到目录，然后使用npm启动应用程序。如果你已经下载了我已完成的任何应用程序，那么很可能是你去了GitHub上的项目，克隆了存储库，并安装了它的依赖项。这对我们的用户来说有点单调乏味,坦率地说,这是一种令人无法接受的体验。我们的Electron应用程序应该像任何其他桌面应用程序一样。用户应该能够导航到网站，下载应用程序，然后双击其图标以启动应用程序。没有终端。没有安装依赖项。没有克隆存储库。

在本章中，我们将使用我们在本书前面构建的应用程序Fire Sale,并使用名为Electron Packager的第三方库将其构建为桌面应用程序。您可以在GitHub上找到存储库，网址为https://github.com/electron-in-action/firesale。我们首先检查一个方便地称为第14章开头的分支。这大致是我们在第8章中停下的地方。

我们从设置Electron Packager开始。我们使用命令行标志来定制诸如输出目录之类的东西，并设置允许我们同时为多个平台构建的npm脚本,尽管我将解释为什么这可能是不明智的。接下来，我们为Fire Sale添加自定义图标。最后，我们讨论Electron的asar归档格式并将其添加到我们的构建过程中。

在使用Electron Packager建立Fire Sale之后，我们使用一种称为Electron Forge物的替代品来构建Jetsetter。Electron Packager简单，低水平，是了解发生了什么而不会深入杂草的好方法，Forge包含更多的铃声和口哨声。我为Jetsetter选择了它，因为我们使用electron-compile将JSX转换为vanilla JavaScript。对于新的Electron应用，Electron Forge对于奠定基础也是很好的。在学习和弄脏手的名义，我选择不使用Electron Forge，但你可能想在开始下一个Electron应用程序时考虑它。本章介绍如何转换最初未使用Electron Forge创建的Electron应用程序。

在这个简短的章节结束时，我们将有一个可执行的Electron应用程序，它不需要从命令行启动，并且可以由未在其计算机上安装Node的人使用。第15章和第16章深入讨论如何正确打包macOS和Windows应用程序。

### *介绍Electron Packager*

Electron Packager是围绕开发应用程序并将其打包以进行分发所需的众多步骤的抽象。在开发过程中，我们为我们的平台（macOS，Windows或Linux）和处理器架构（32位或64位）下载了预装版本的Electron。我们不一定知道用户使用的平台和架构，也不会自己构建。我们有责任为我们计划支持的每个平台和架构组合构建应用程序版本。Electron Packager在这里帮助我们完成这项工作。

###### *设置Electron Packager*

您可以使用npm install electron-packager -g或yarn global add electron-packager全局安装Electron Packager。我更喜欢在个别项目上本地安装它，因为每个项目都指定了自己的Electron Packager版本。我将它添加到项目的package.json中的devDependencies中。如果您克隆了存储库并安装了依赖项，那么您应该很高兴。

我们首先向package.json添加构建脚本。在本章中，我们从故意简单的配置开始，并在我们继续的基础上构建它，沿途改进此构建脚本。

**清单14.1添加构建脚本：./package.json**

{

// …Omitted… "scripts": {

"start": "electron .",

**"build": "electron-packager ."**,

**向package.json添加一个简单的构建脚本**

"test": "echo \"Error: no test specified\" && exit 1"

},

// …Omitted…

}

使用我们的构建脚本，是时候启动Electron Packager了。您可以使用npm run build运行构建脚本，该脚本运行安装在node\_modules目录中的本地版本的electron-packager。构建完成后，您应该看到一个附加目录，其中包含应用程序的名称以及您当前使用的平台和体系结构，如图14.1所示。在我写这篇文章时，我使用的是64位Mac，目录名为firesale-darwin-x64。



**图14.1 默认情况下，Electron Packager将构建放在项目的根目录中。这可能会有问题，因为我们通常不希望将构建的应用程序提交到源代码管理中。**

firesale-darwin-x64目录包含名为firesale的桌面应用程序以及许可证和包含该版本的小文本文件。双击该应用程序以像任何其他应用程序一样打开它。如果您需要自己证明，请随意将应用程序拖到另一个目录并从那里启动它。

Windows上Electron Packager的输出更加冗长。在macOS上，应用程序是有效的文件夹，可以隐藏我们的大部分复杂性。在第15章中，我们将介绍如何为Windows创建一个隐藏用户复杂性的安装程序。

这是快速和简单的，但仍有许多需要改进的地方。文件名没有大写，我们仍然使用股票Electron图标，我们已经为当前的架构和平台以及其他问题构建了一个版本。

**快速了解依赖关系**

默认情况下，Electron Packager包含node\_modules目录中package.json中列出的所有依赖项。它确实省略了package.json中devDependencies下列出的依赖项。这不是我们在本书中的任何应用程序遇到的问题，但它是您在冒险中可能遇到的问题。您可以通过两种方式处理此问题：将依赖项从devDependencies移至依赖项或将--no-prune标志传递给Electron Packager。

###### *配置输出目录*

我刚刚列出的问题都是有效的，我们确实解决了这些问题，但我们首先要解决一个更紧迫的问题。我们可能不想将我们构建的应用程序检查到版本控制中。将应用程序构建到项目的根目录使这很困难。当我们开始构建其他平台和架构时，它会增加越来越多的视觉噪音和混乱。

如果从命令行键入./node\_modules/.bin/electron-packager --help，Electron Packager会显示它接受的所有选项和标志。要指定我们希望Electron Packager放置应用程序的目录，我们可以使用--out标志。将构建脚本更新为electron-packager . --out=build。您可以将build目录添加到.gitignore文件中，以使编译的应用程序不受版本控制。我已经在这个项目中为你完成了这个。

###### *配置应用程序的名称和版本*

Electron Packager使用项目的package.json中的信息在构建过程中设置某些参数。例如，它读取版本字段以确定这是什么版本的应用程序。它还使用name字段来设置应用程序的名称。但是，这有一些局限性。您不能在package.json的名称字段中使用空格或大小写。幸运的是，package.json只是一个文本文件，而npm忽略了它不理解的任何属性。这是存储可能对Electron Packger有意义的其他信息的好地方。我们知道我们想要更改应用程序的名称，如图14.2所示。

**清单14.2 自定义应用程序名称和版本：./package.json**

in

{

"name": "firesale",

"version": "1.0.0",

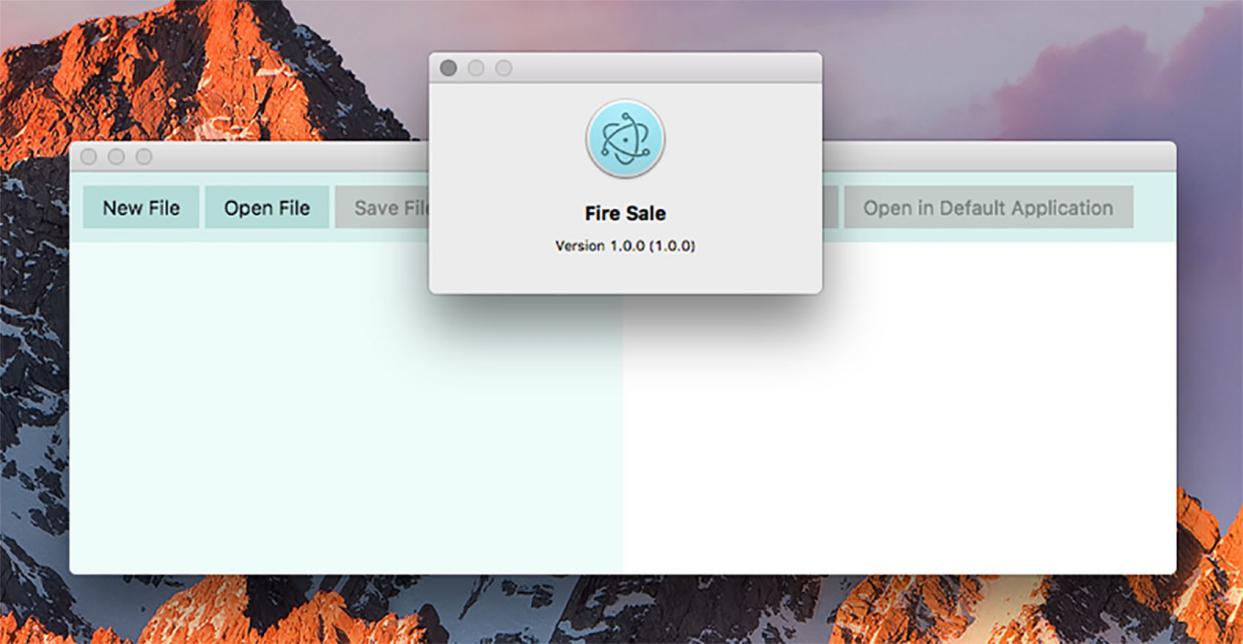
"description": "完成的代码 from [Electron [Action](http://bit.ly/electronjs).",](http://bit.ly/electronjs))

**"productName": "Fire Sale",**

// …Additional properties omitted…

}

**Electron Packager使用productName字段（如果存在）作为应用程序的名称而不是name，name不能包含空格或大小写。**



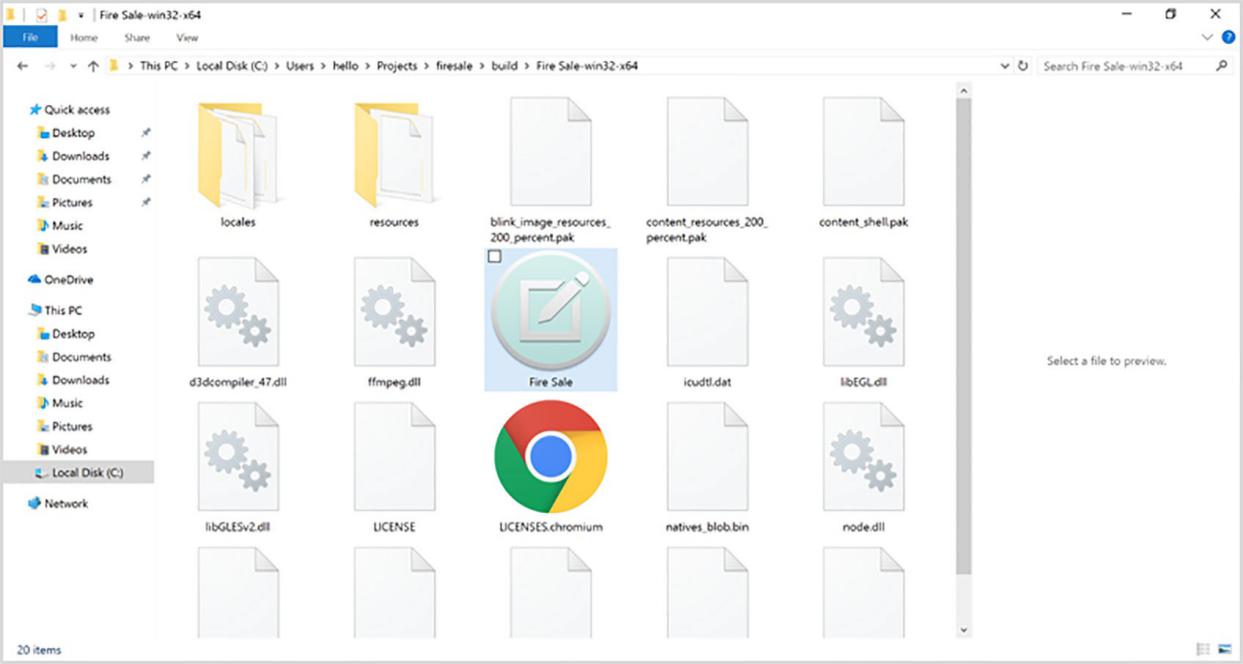
**图14.2 设置productName后，我们的应用程序在构建后具有正确的名称。**

###### *更新应用程序图标*

我喜欢Electron的标志。在本书中我们做得很好。但是，如果构建在Electron之上的每个应用程序都使用相同的徽标，那么很难将一个应用程序区分另一个应用程序。让我们用自己的图标替换图标。

正如我们在第9章中看到的，不同的操作系统对其图标的文件格式有不同的偏好。此时，我们正在构建自己的操作系统，因此我们将根据我们使用的操作系统做出选择，如图14.3所示。在本章后面，我们添加了一些细微差别来指导Electron Packager为给定平台选择正确的图标。

macOS支持.icns文件。Windows支持.ico文件。Linux支持.png文件。现在，根据您当前的操作系统将相应的文件传递给--icon标志，如清单14.3所示。



**图14.3 嘿，看！ 我们现在有一个自定义图标！**

**清单14.3 使用自定义图标构建应用程序：./package.json**

在Windows上使用此选项

**在macOS上使用此选项**

electron-packager . --overwrite --out=build --icon=icons/Icon.icns electron-packager . --overwrite --out=build --icon=icons/Icon.ico electron-packager . --overwrite --out=build --icon=icons/Icon.png

**在Linux上使用此选项**

###### *为多个操作系统构建*

Electron Packager支持--all标志，它创建了五个应用程序版本：64位macOS，32位和64位Windows，以及32位和64位Linux。在本章的后面部分，我们将讨论为什么这并不总是可行的。虽然--all标志提供了为每个平台创建构建的简单方法，但在上一节中，我们为每个平台创建了不同的图标。因此，我们希望在平台上向Electron Packager传递不同的选项。

**清单14.4 为多个平台构建：./package.json**

{

// …Omitted… "scripts": {

"start": "electron .",

"build": "npm run build-mac && npm run build-win

➥ && npm run build-linux",

"build-mac": "electron-packager . --platform=darwin

➥ --out=build --icons=icons/Icon.icns --overwrite",

"build-win": "electron-packager . --platform=win32

➥ --out=build --icons=icons/Icon.ico --overwrite",  "build-linux": "electron-packager . --platform=linux

➥ --out=build --icons=icons/Icon.png --overwrite",

"test": "echo \"Error: no test specified\" && exit 1" 

},

**构建macOS应用程序**

**构建Windows应用程序**

**构建Linux应用程序**

// …Omitted…

}

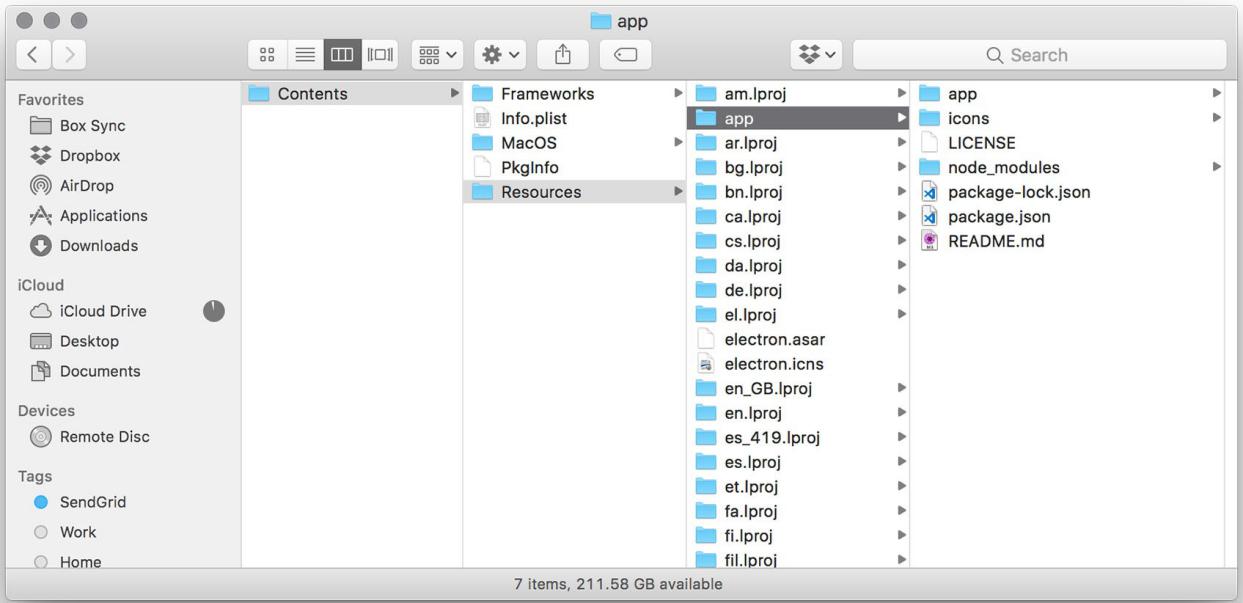
**构建脚本调用三个后续构建脚本中的每一个。**

如果您正在使用macOS，则可以构建Windows二进制文件（如果已安装Wine）以及Linux的二进制文件。Wine是一个用于在Linux和macOS上运行Windows应用程序的实用程序，可以从www.winehq.org下载。在Windows上，您无法为macOS创建符号链接，因此无法构建应用程序。在第15章和第16章中，我们讨论了macOS上的代码签名，这只能在Mac上完成。

### *使用asar*

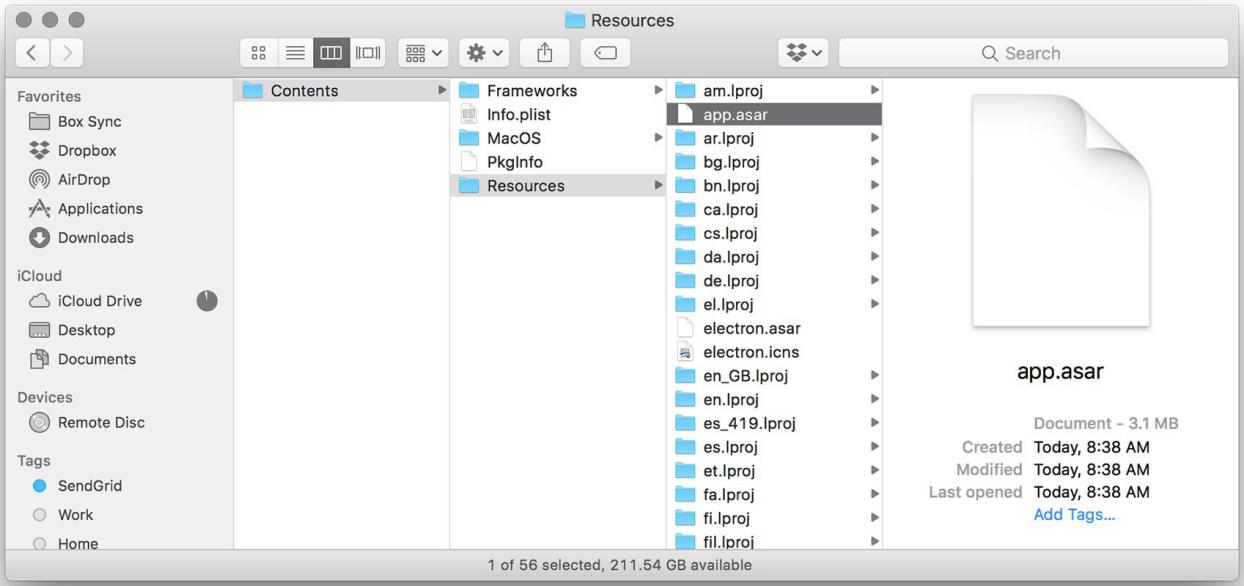
Electron应用程序就像传统的Web应用程序一样，我们所有的代码都隐藏在视线之内。在Web应用程序中，用户可以右键单击并选择“查看源”或打开“开发人员工具”。在我们的Electron应用程序中，一个好奇的用户可以打开应用程序包并导航到我们的源代码，如图14.4所示。

如果您使用的是Windows(译注:在Deepin Linux上也一样)，则可以在与应用程序相同的目录中看到名为resources的文件夹。如果您使用的是macOS，请右键单击“Fire Sale”并选择 Show Package Contents。接下来，您将看到一个名为Contents的文件夹。导航到Contents目录，然后导入Resources。这个文件夹里面有很多文件夹。我们感兴趣的是app。打开此文件夹，您可以找到Fire Sale的所有源代码。



**图14.4 应用程序的源代码位于应用程序本身内。**

您的下一个冲动可能是在流行的Electron应用程序（如Slack，Atom或Visual Studio Code）上运行相同的实验。但是您找不到包含应用程序源代码的文件夹。相反，您将看到一个文件，其中包含一个名为app.asar的不熟悉的文件扩展名，如图14.5所示。



**图14.5 包含在asar存档中的应用程序。**

这是什么？ Asar是Electron应用程序使用的归档格式。它有效地连接文件并在JSON对象前面添加每个文件的起始位置及其长度，从而允许随机访问存档中的特定文件。

为什么要使用asar？ Asar加速了Node中需要文件的速度，因为它们都位于同一个文件中，并在应用程序启动时加载到内存中。它还可以帮助我们避免因长文件名导致的Windows中的错误。错误是相当不言自明的：指定的路径，文件名或两者都太长。完全限定的文件名必须少于260个字符，目录名称必须少于248个字符。虽然您可以练习避免令人难以置信的长文件名的规则，但您无法控制依赖关系,尤其是开发人员使用运行macOS或Linux的计算机编写的依赖关系。Asar通过创建名为app.asar的单个文件来解决此问题。

使用Electron Packager创建一个asar存档非常容易。只需在命令中添加--asar标志并构建应用程序即可。该文件夹将消失并替换为存档。

**清单14.5 使用asar构建多个平台：./package.json**

{

// …Omitted… "scripts": {

"start": "electron .",

"build": "npm run build-mac && npm run build-win && npm run build-linux", "build-mac": "electron-packager . --platform=darwin

➥ --out=build --icons=icons/Icon.icns --**asar** --overwrite",

"build-win": "electron-packager . --platform=win32

➥ --out=build --icons=icons/Icon.ico --**asar** --overwrite", "build-linux": "electron-packager . --platform=linux

➥ --out=build --icons=icons/Icon.png --**asar** --overwrite",

"test": "echo \"Error: no test specified\" && exit 1"

},

// …Omitted…

**将--asar标志添加到macOS的构建脚本中**

将--asar标志添加到Linux的构建脚本中

}

**将--asar标志添加到Windows的构建脚本中**

Asar不是保护或隐藏源代码的方法。npm上的asar模块允许任何人从存档中提取文件夹结构。或者，在文本编辑器中打开存档会显示asar不会以混淆的方式执行任何操作。如果您想查看Electron应用程序的源代码，请全局安装asar二进制文件，导航到应用程序的app.asar，然后解压缩文件夹结构。

译注:

全局安装asar命令:

npm install -g asar

查看asar命令的用法:

asar --help

解压的例子:

dbdu@dbdu:/media/dbdu/Data/repository/GitLab/electron-in-action/ch14/filesale/build/Fire Sale-linux-x64/resources$ asar e app.asar .

注意最后的点不能丢,表示解压目标目录为当前目录.

Electron不提供隐藏或加密HTML，CSS和JavaScript代码的方法。如果这是您的业务需求，您可能需要考虑使用编译语言编写应用程序的敏感部分，例如带有Node绑定的C ++。同样，没有什么可以阻止你打开你喜欢的Electron应用程序Slack，例如:并查看创建Electron应用程序时使用的一些方法。

**在Electron应用中使用asar**

Electron修补Node API以读取和写入文件系统以自动支持asar。这意味着您可以将asar存档视为原始文件夹结构。如果您需要使用Node的内置fs模块的未修补版本，您可以使用Electron的内置original-fs模块。

我们已成功为Fire Sale设置包装。您可以在第14章结尾分支（http://mng.bz/vI4g）中找到Fire Sale的代码。此设置适用于像Fire Sale这样的简单应用程序，但是具有需要编译的代码的应用程序呢？ 是的，您可以在打包之前设置一些Gulp任务或其他类型的构建步骤，但是在本书的这一点上，Electron社区已经为您解决了这个问题，您不应该感到惊讶。在下一节中，我们将介绍使用Electron Forge来构建和打包Jetsetter，这需要在打包之前进行编译步骤。

### *Electron Forge*

Electron Forge（https://github.com/electron-userland/electron-forge）是 electron-compile, electron-rebuild, Electron Packager和其他流行的第三方Electron库的抽象。Electron Forge使用几种不同的蓝图之一为全新的Electron应用奠定了基础。它还可以导入现有的应用程序。我们使用Electron Bleconf与第11章和第12章的Jetsetter应用程序，因为它依赖于 electron-compile和electron-rebuild。使用像Electron Packager这样的工具肯定不是不可能，但与Electron Forge程序的简单程序相比，这是一件非常繁琐的事情，它具有 electron-compile和electron-rebuild的内置支持。

您将从第14章开始分支开始，该分支可以在GitHub（http://mng.bz/MdLb）上找到。您还可以在第14章结束分支（http://mng.bz/mzhH）上引用已完成的示例。

**关于第三方依赖关系的快速说明**

出版一本书的麻烦在于，有时，依赖库会发生变化。在本节中，我们正在几个不同的库的交叉点工作：Electron, electron-precompile，Electron Forge，React等。所有这些库的交叉兼容性使您的作者充满了存在感的恐惧感。如果您对本节中打印的代码有任何问题，我建议您访问GitHub上的存储库，该存储库应该与所涉及的库的最新版本一起使用。

开箱即用，Electron Forge还处理了分发您的应用程序的许多问题，我们将在第15章和第16章介绍。它可以负责将您的应用程序发布到Amazon S3，GitHub版本或您自己的自定义服务器。正如我之前所说，我们选择以了解Electron如何工作的名义手工完成大量工作，但我强烈建议您考虑使用Electron Forge物来完成下一个应用。在本节中，我们将介绍如何将最初未使用Electron Forge创建的应用程序转换为已准备好并能够利用Electron Forge的应用程序。

我们首先需要完成几个步骤：安装electronic-forge二进制文件，将Jetsetter转换为Electron Forge应用程序，然后运行build命令。Electron Bleconf使用Yarn作为npm的替代品来安装和管理包装。如果您还没有安装它，则需要安装它。请访问官方文档（**https://yarnpkg.com/en/docs/install**）以获取最新说明，并了解如何在操作系统上安装它。

在系统上安装Yarn后，运行yarn global add electron-forge。这将安装命令行工具，允许您创建Electron Forge应用程序，以及转换现有的Electron应用程序以与Electron Forge使用。

译注:

请注意,在系统上安装yarn命令后,请务必配置环境变量,否则提示electron-forge命令找不到,配置方法参照作者提供的链接即可.

***Electron Forge* 269**

###### *将Electron应用导入Electron Forge中*

当我们在第11章中构建Jetsetter时，我们还没有讨论过Electron Forge。因此，Jetsetter不是Electron Forge应用。幸运的是，很容易将其转换为一个。从jetsetter目录中，运行**electron-forge import**。此时，系统会询问您有关如何继续执行脚本的一系列问题。回答如下问题：

**1** WARNING: We will now attempt to import "/Users/<username>/Projects/ jetsetter". This will involve modifying some files, are you sure you want to continue? *Yes*.

**2** Do you want us to change the "main" attribute of your package.json? If you are currently using Babel and pointing to a "build" directory say yes. *No.*

**3** Do you want us to update the "start" script to instead call the electron-forge task "electron-forge start"? *Yes*.

**4** Do you want us to update the "package" script to instead call the electron-forge task "electron-forge package"? *Yes*.

**5** Do you want us to update the "make" script to instead call the electron-forge task "electron-forge make"? *Yes.*

第一个问题是请求您的许可修改应用程序的package.json文件。这是必要的。回答不会中止这个过程。第二个问题不适用于我们。Jetsetter使用Electron Compile来动态转换JavaScript，Sass和其他语言。如果我们使用Gulp，Grunt或webpack之类的任务运行程序将我们的源代码转换到另一个目录中，我们将对此问题回答Yes。

最后三个问题是询问我们是否想要使用Electron Forge来开发我们的开发应用程序，将我们在本章前面所做的应用程序打包到Fire Sale，并构建一个可分发的,我们将在下一章中介绍。因为这就是我们开始导入应用程序的原因，我们将对每个问题回答Yes。

###### *使用Electron Forge物构建应用程序*

正如我之前提到的，Electron Forge是Electron生态系统中几个有用的库的抽象。存在使共同任务更容易。构建应用程序就像运行纱线运行包一样简单。这使得Electron Packager完成了本章前面介绍的Fire Sale中的许多步骤。

Electron Forge将参数传递给Electron Packager。它调用输出目录而不是构建。Electron Packager从node\_modules目录中提取模块并将它们安装到您的应用程序中。在撰写本文时，存在模块未正确移动的问题。此问题与当前版本的npm一起出现，并且可能会在您阅读这段文字时得到缓解。与此同时，我发现使用Yarn而不是npm很容易解决这个问题。Electron Forge为您的package.json添加了一些字段，因此找到标题为electronPackagerConfig的字段，并修改它以匹配此列表。

**清单14.6 使用Yarn作为包管理器：./package.json**

"electronPackagerConfig": {

"packageManager": "yarn"

}

使用此代码，运行纱线运行包。您的应用程序应成功构建。见图14.6。您可能已经注意到Electron Forge为package.json添加了一些其他配置字段。正如我之前提到的，Electron Forge还可以帮助创建安装程序并将您的应用程序发布到Web上。我们将在接下来的章节中介绍这些功能。



**图14.6 使用Electron Forge物打包应用程序**

使用Electron Forge物，一切都是代表您设置的。在这一点上，你可能属于两个类别中的一个：你要么惊讶于Forge与Packager相比多么简单，要么你对所有的黑魔法感到困惑。这些反应都是公平的，并强调了在惯例和配置之间的一些权衡取舍。我希望通过尝试两者，你会对自己的应用程序有所了解。

译注:

1.编译输出的目录在项目的根目录下的out目录里.

2.可能是electron-forge安装的时候是最新版(基于electron也可能是最新),发现样式丢失.

### *本章小结*

* + - * ElectronPackager自动化手动和容易出错的包装Electron应用程序以进行分发的过程。
      * 应用程序应打包在您希望分发的每个目标平台上。
      * 在package.json中设置productName键指示Electron Packager使用支持空格和大小写的自定义名称。
      * ElectronPackager接受--icon标志，允许您为应用程序定义自定义图标。
      * Asar是Electron应用程序使用的归档格式，它连接文件并允许随机访问归档中的特定文件。
        + Electron Forge物可用作新Electron项目的样板。它还可以用于促进依赖electron-compile的构建应用程序。

*发布和更新应用程序*

***本章包括***

* 从Electron发送崩溃报告
* 从Electron发送未捕获异常的报告
* 创建服务器以收集崩溃和异常报告
* 自动将应用程序更新推送给用户

您已经构建了应用程序，并且已准备好将其分发给全世界。你已经准备好了你的公告，而你的光标突然停留在发布按钮上,突然之间你的思想开始与一些非常重要的问题竞争：如果它不能按预期在所有平台上运行怎么办？ 如果您还没有遇到错误怎么办？ 如果您需要向用户推出新版本的应用程序，该怎么办？

这些担忧都是有效的。好消息是Electron让你在每个方面都有所涉及。在本章中，我们将介绍如何收集崩溃报告，以及如果您收到意外大量的崩溃报告，如何将更新推送到当前在其系统上安装了应用程序的所有用户。

### *收集崩溃报告*

无论您的开发人员技能有多好，但您的应用程序仍会出现错误。一些用户报告他们，但许多其他人是沉默。即使在报告错误的用户中，也很难确定整个用户群中给定问题的常见程度。崩溃报告对于发现发生了哪些类型的问题以及用户体验它们的频率至关重要。

尽管Electron对崩溃报告提供支持肯定是一种解脱，但令人惊讶的是。正如我们在本书中所述，Electron建立在Chromium之上，而崩溃报告肯定是Chromium团队在其项目中处理的事情。然而，站在巨人的肩膀上，我们需要在我们的Electron应用程序中设置崩溃报告器，以及部署一个简单的服务器来收集这些崩溃报告以供以后分析。

在本章中，我们将故障报告和自动更新实施到Fire Sale中，这是我们在第14章开始为部署做好准备的应用程序。我们开始研究第15章开始的分支，可在https:// github.com/electron-in-action/firesale找到。

我们解决以下问题：

* + - 配置Electron的内置功能，以便在应用程序意外崩溃时发送崩溃报告。应用程序完全关闭时会触发崩溃报告器。
    - 设置一个天真简单的服务器，以接收和收集从我们的应用程序版本发送的崩溃报告。
    - 配置我们的应用程序以侦听在运行时发生的未捕获异常。这段代码是我们自己的手工而不是Electron开箱即用的东西。当发生较小的错误时，将触发此功能，这必然会导致应用程序本身完全失败。
    - 更新我们的服务器以接收这些错误报告。

###### *设置崩溃记录器*

在幕后，Electron使用两个崩溃报告系统之一：在macOS上，Chrome团队的Crashpad报告引擎; 在Windows和Linux上，一种称为Breakpad的旧引擎。这些选项会影响我们在Electron中实施崩溃报告的方式。主要的区别是Breakpad崩溃报告只需要在主进程中设置，而Crashpad必须在main和render染器进程中启动。

是的，您可以将主进程中的代码复制并粘贴到每个渲染器进程中。（我们很幸运，在Fire Sale中只有一个。）但这意味着你需要记住要在需要更新配置的情况下在多个地方进行更改。就个人而言，我宁愿花一些额外的时间来做一辈子的懒惰。让我们设置一个我们可以在两个地方使用的功能。

**清单15.1 创建一个文件来配置和启动Crash Reporter：./app/crash-reporter.js**

const { crashReporter } = require('electron'); const host = 'http://localhost:3000/'; 

**通过HTTP POST请求发送崩溃报告的URL。**

const config = {

productName: 'Fire Sale',

companyName: 'Electron实战', submitURL: host + 'crashreports', uploadToServer: true,

};

crashReporter.start(config);

**表示您希望将崩溃结果发送到服务器**

使用传入的配置选项启动Crashpad或Breakpad崩溃报告器

console.log('[INFO] Crash reporting started.', crashReporter); module.exports = crashReporter;

我们设置了一个可重用的模块来启动崩溃报告器，但您可以更改任何这些选项的值。特别是，我们需要更改submitURL，因为我们的用户不太可能在该端口上运行服务器，如果他们在本地收集崩溃报告并且从未将它们发送给我们，那么最终对我们没有帮助。

除非将代码拉入main和render染器进程，否则代码本身不会启动崩溃报告引擎。我们首先将其拉入主进程。

**清单15.2 在主进程中启动崩溃报告器：./app/main.js**

const { app, BrowserWindow, dialog, Menu } = require('electron'); const createApplicationMenu = require('./application-menu'); const fs = require('fs');

**require('./crash-reporter');**

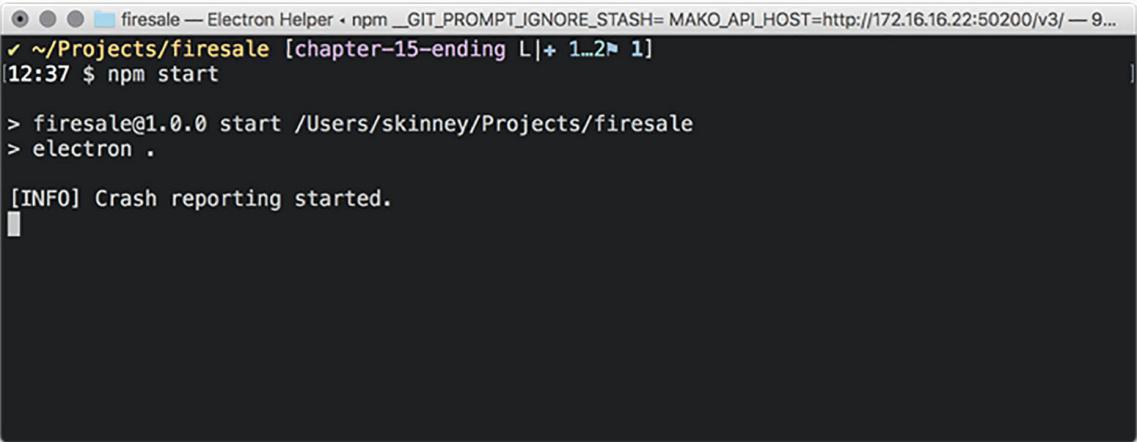
const windows = new Set(); const openFiles = new Map();

**我们导入crash-reporter.js，它将启动它。**

// 剩下的主进程代码…

这种方法足以实现在Windows和Linux上发送崩溃报告的能力。我们可以通过检查终端是否在应用程序启动时检查列表15.1中我们登录到控制台的消息来验证崩溃报告器中的代码是否已执行，如图15.1所示。

正如我之前提到的，我们需要做一个,可以说是简单的步骤来使崩溃报告在macOS上工作：在渲染器进程中执行相同的代码。



**图15.1 控制台消息验证crash-reporter.js中的代码是否已执行。**

**清单15.3 在渲染器进程中启动崩溃报告器：./app/renderer.js**

const { remote, ipcRenderer, shell } = require('electron'); const { Menu } = remote;

const path = require('path');

const mainProcess = remote.require('./main.js');

const currentWindow = remote.getCurrentWindow(); **require('./crash-reporter');**

**在渲染器进程中启动崩溃报告器**

我们可以通过打开Fire Sale UI中的开发人员工具并确认控制台消息已正确记录来验证崩溃报告器是否已启动，如图15.2所示。

译注:

1.崩溃报告终端打印,Windows下打印信息最多,Linux和Mac下都很简短;

2.因为在上一章我们在Mac上我们安装了yarn命令,和node有点冲突,运行npm start会报:

[Cannot find module '../lib/utils/unsupported.js'](https://www.cnblogs.com/wanghui-garcia/p/9947347.html)

译者,没有去卸载重装node,而是直接用yarn install及yarn start运行,OK!

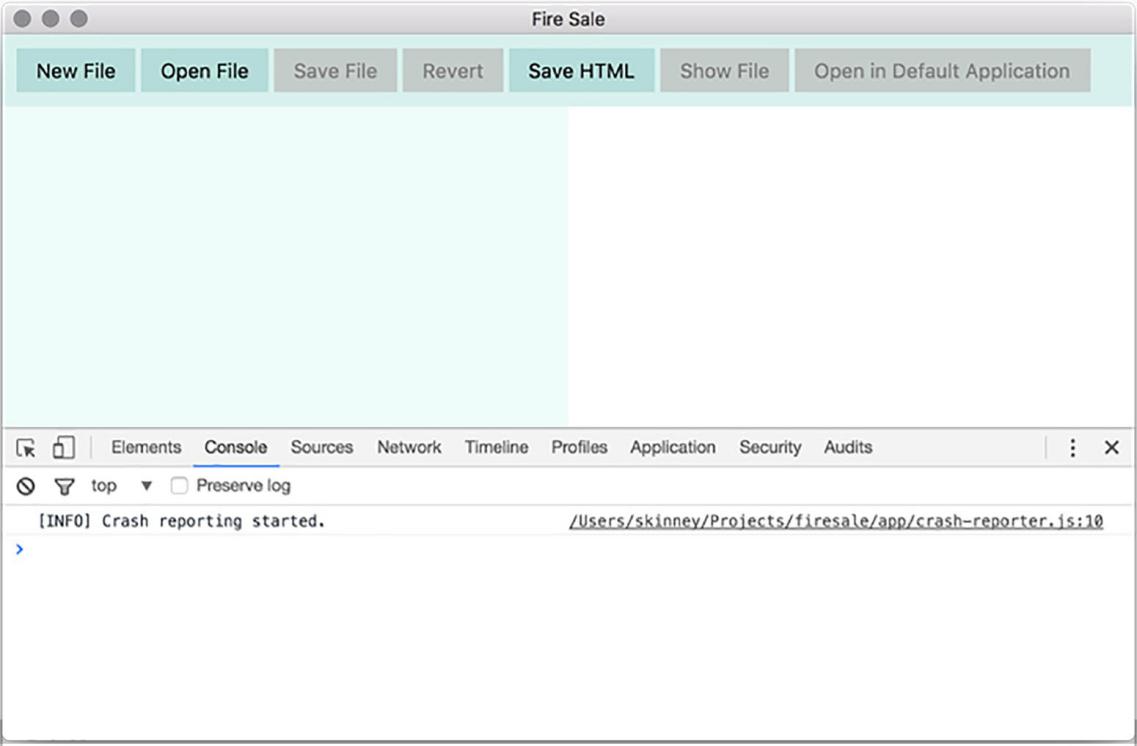
###### *设置一个服务器以接收崩溃报告*

我们可以发送崩溃报告,但在哪里？我们可以通过在渲染器进程中的Developer Tools中键入process.crash( )来触发崩溃。如图15.3和15.4所示，这样做会导致以下结果：页面崩溃，Developer Tools锁定，主进程记录错误。

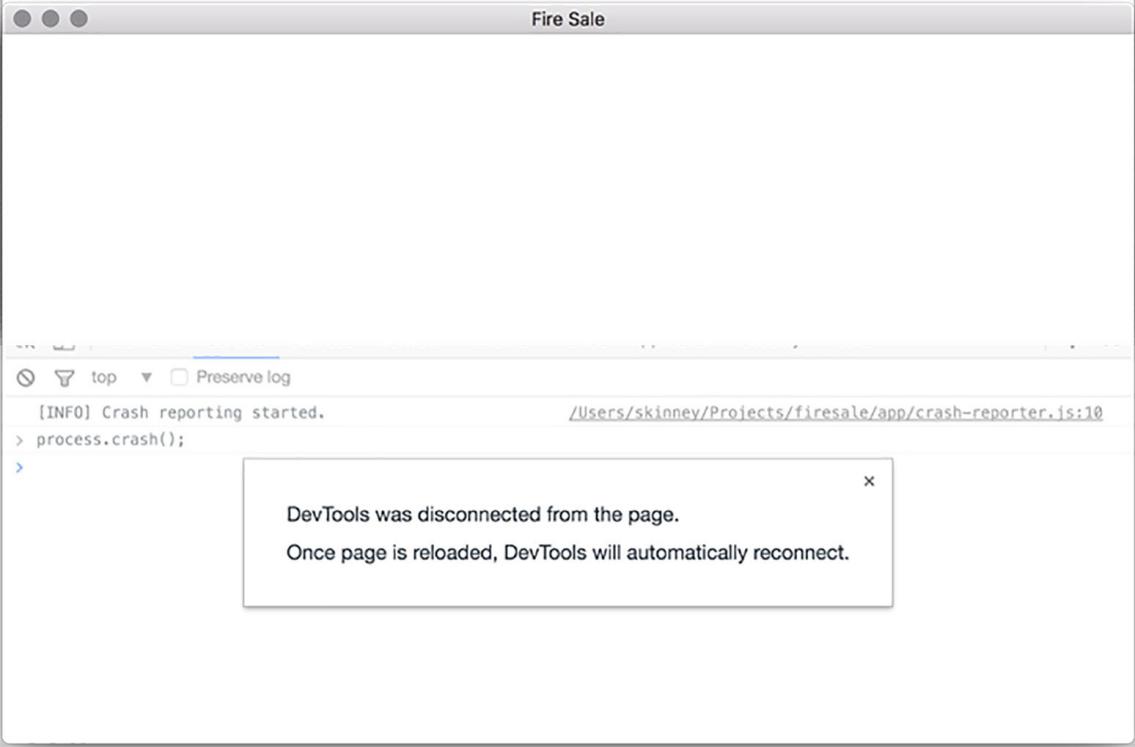
我们需要解决的直接问题是终端中的错误。错误的确切内容因操作系统而异，但要点是崩溃报告者无法连接到服务器。这是有道理的，因为我们还没有设置服务器。

您可能已经在使用API服务器，具体取决于您的应用程序。如果是这样，我建议添加一个端点来接收来自您的应用程序的POST请求。也就是说，Fire Sale不需要API服务器，我将假设我们没有API服务器。对于那些你这样做的人，我怀疑后面的内容仍然有用，因为它说明了报告崩溃时可以预期的有效负载类型。

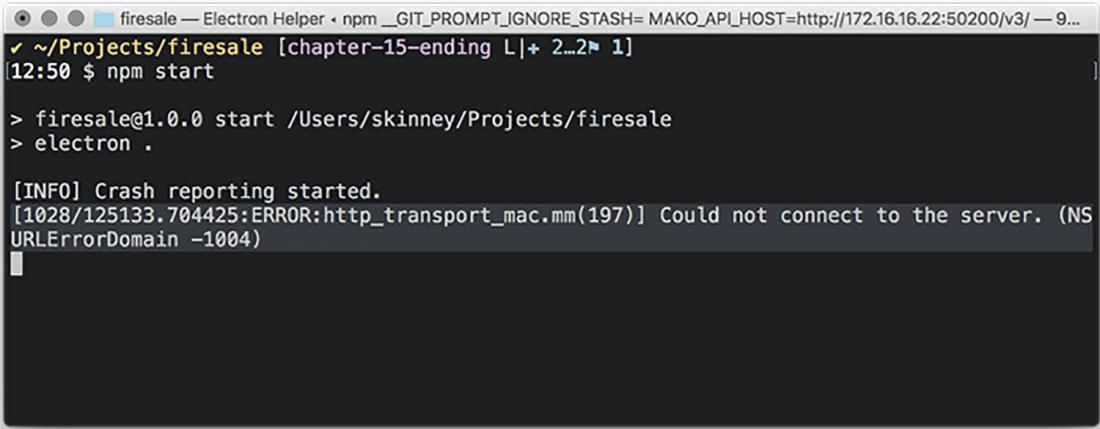
如清单15.4所述，让我们创建一个简单的服务器来捕获和记录崩溃报告。该代码可以在GitHub上找到，网址是http://mng.bz/k6eT。



**图15.2 通过检查控制台验证是否已执行崩溃报告代码。**



**图15.3 在渲染进程中触发崩溃需要重新加载UI。**



**图15.4 Electron尝试发送崩溃报告但无法找到服务器。**

**清单15.4 创建一个简单的崩溃报告服务器**

const express = require('express'); const multer = require('multer');

const bodyParser = require('body-parser'); const uuid = require('uuid');

const writeFile = require('write-file'); const path = require('path');

const http = require('http');

// Express是Node.js的简单Web服务器库。

const app = express();

const server = http.createServer(app);

// bodyParser库是Express的中间件，可以处理HTTP请求的主体。

app.use(bodyParser.urlencoded({ extended: false })); const crashesPath = path.join( dirname, 'crashes');

const upload = multer({ dest: crashesPath,

}).single('upload\_file\_minidump');

**Multer允许我们接收多部分文件（例如Electron生成的崩溃报告小型转储）。**

app.post('/crashreports', upload, (request, response) => {

const body = {

...request.body,

filename: request.file.filename, date: new Date(),

};

const filePath = `${request.file.path}.json`; const report = JSON.stringify(body);

writeFile(filePath, report, error => {

**将迷你转储的文件名和当前时间添加到请求正文中的JSON**

将文件写入文件系统

if (error) return console.error('Error Saving', report); console.log('Crash Saved', filePath, report);

});

response.end();

});

server.listen(3000, () => {

console.log('Crash report server running on Port 3000.');

});

如果从GitHub克隆库，请安装依赖项，然后运行npm start。您应该准备好服务器，并且能够收集在http://localhost:3000上运行的崩溃报告，这正是我们在15.1清单中指出Electron崩溃报告器的地方。

每个崩溃报告在./crashes目录中创建两个文件：来自Chromium的小型转储和一个带有有关崩溃的其他元数据的JSON文件。JSON文件包含有关崩溃的应用程序版本及其运行的平台的有用数据。

**清单15.5 崩溃报告中的JSON元数据示例**

{

"\_companyName": "Electron实战", "\_productName": "Fire Sale", "\_version": "1.0.0",

"guid": "46cecb8f-f2de-4159-a235-dc8713f8393f", "platform": "darwin",

"process\_type": "renderer", "prod": "Electron",

"ver": "2.0.4",

"filename": "a51d2ca2e13cad25cea6c2bd15ae4d4d"

}

**发生崩溃的平台。**

发生崩溃的进程类型。

目前正在使用的Electron版本。

mini-dump所在文件的名称。

报告崩溃时，服务器将两个文件保存到/crashes目录：a51d2ca2e13cad25cea6c2bd15ae4d4d和a51d2ca2e13cad25cea6c2bd15ae4d4d.json。前者是Chromium在应用程序崩溃时发生的所有事情的转储。此文件采用二进制格式，无法在文本编辑器中打开。

解析此文件的最简单方法是使用Electron团队的minidump库。您可以使用npm install -g minidump全局安装它。安装后，您可以使用minidump\_stackwalk命令行工具，并可以使用minidump\_stackwalk <mini-dump file>的名称来读取转储的内容。或者，您可以按照工具文档（www.npmjs.com/package/minidump）中的说明以编程方式在崩溃报告服务器上使用此工具。

译注:

1.Linux下运行,控制台在发生崩溃时,控制台显示:

Failed to get crash dump id.

Report Id:

2.macOS下是yarn install 安装依赖,使用yarn start运行这两个项目.

###### *报告未捕获的异常*

Electron的内置崩溃记录器很擅长它所宣称的关于箱子报告崩溃的事情。但根据我的经验，一般的错误和小错误比直接崩溃更常见。如果我们能够收集这些错误，更好地了解应用程序的粗糙边缘以及如何修复它们，那将会很棒。

不幸的是，这不是Electron已经内置的工具。好消息是，鉴于我们在本书的整个过程中已经获得的知识，这个功能相对容易实现。

要实现此功能，我们必须侦听任何未捕获的异常。主进程和渲染器进程之间的语法略有不同，但实现方式相同，如下面的清单所示。当发生未捕获的异常时，向我们的崩溃服务器发送HTTP POST请求，然后将该错误记录到JSON文件中。

**清单15.6 更新崩溃报告器以报告未捕获的异常：./app/crash-reporter.js**

const { crashReporter } = require('electron'); **const request = require('request');**

**const manifest = require('../package.json');** const host = 'http://localhost:3000/';

const config = { productName: 'Fire Sale',

companyName: 'Electron实战', submitURL: host + 'crashreports', uploadToServer: true,

};

**crashReporter.start(config);**

// 设置报告未捕获异常的函数

**const sendUncaughtException** = error => {

const { productName, companyName } = config;

// 向我们之前创建的崩溃服务器发送HTTP POST请求

request.post(host + 'uncaughtexceptions', {

form: {

\_productName: productName,

\_companyName: companyName,

\_version: manifest.version, platform: process.platform,

process\_type: process.type, ver: process.versions.electron,

// 发送有关已触发错误的信息

error: {

name: error.name, message: error.message,

stack: error.stack,

},

},

});

};

// 检查我们是在主进程还是渲染器进程中运行

if (process.type === 'browser') {

//如果在主进程中发生错误，则使用Node的uncaughtException事件

process.on('uncaughtException', sendUncaughtException);

} else {

//如果渲染器进程中发生错误，则向全局对象添加事件监听器

window.addEventListener('error', sendUncaughtException);

}

console.log('[INFO] Crash reporting started.', crashReporter);

module.exports = crashReporter;

每当一个错误被触发并且它在没有被处理的情况下冒泡到窗口对象时，就会向我们的服务器发送一个报告。但就像上次一样，我们的服务器还没有设置好处理这个报告。我们需要添加另一条路径来记录此错误。

**清单15.7 设置服务器路由以接收未捕获异常的报告**

const express = require('express'); const multer = require('multer');

const bodyParser = require('body-parser'); const uuid = require('uuid');

const writeFile = require('write-file'); const path = require('path');

const http = require('http');

const app = express();

const server = http.createServer(app); app.use(bodyParser.urlencoded({ extended: false }));

const crashesPath = path.join( dirname, 'crashes');

//在文件系统上添加路径以存储未捕获的异常报告

const exceptionsPath = path.join( dirname, 'uncaughtexceptions');

const upload = multer({ dest: crashesPath,

}).single('upload\_file\_minidump');

app.post('/crashreports', upload, (request, response) => {

// …

});

app.post('/uncaughtexceptions', (request, response) => {

//使用UUID模块为崩溃报告创建唯一标识符

const filePath = path.join(exceptionsPath, `${uuid()}.json`); const report = JSON.stringify({

...request.body,

//将日期添加到崩溃报告中

date: new Date()

});

writeFile(filePath, report, error => {

if (error) return console.error('Error Saving', report); console.log('Exception Saved', filePath, report);

});

response.end();

});

**Writes the report to the filesystem**

server.listen(3000, () => {

console.log('Crash report server running on Port 3000.');

});

译注：

作者没有提到如何验证代码是否工作，译者实现了一个简单的在开发者工具的控制台测试例子：

＞var ev = new CustomEvent('error', {

bubbles: 'true',

cancelable: 'true',

detail: 'tcstory'

});

undefined

＞window.dispatchEvent(ev);

true

　　　　　　　　　此时观察服务器控制台收到异常报告！

### *签名您的应用*

在第14章中，我们打包了我们的应用程序，以便用户不需要安装Node或熟悉命令行来启动它。在上一节中，我们添加了崩溃和异常报告，以确保在我们的应用程序无法按预期工作的情况下通知我们。我们的下一步是签署我们的应用，以便我们的用户可以确定他们是真正的东西，而不是廉价的替代品。

这个过程在macOS和Windows上有所不同，所以让我们分别浏览每个平台。虽然我们在第16章中为Mac App Store打包Fire Sale时，macOS的某些步骤似乎重复，但存在无数微妙的差异。

在我们深入了解代码签名我们的应用程序的细节之前，我将花一点时间来解释代码签名是什么。互联网是一个疯狂的地方。您可以将代码签名视为围绕您的应用程序的防篡改密封。对我们的应用程序进行签名后，用户可以确信它们都来自您，并且没有人修改它或者在此过程中篡改它。除了一般的好主意外，macOS和Windows更喜欢用户使用已签名的应用程序，并在尝试打开未签名的应用程序时提出一系列警告。根据用户的设置，用户可能根本无法打开未签名的应用程序。

###### *为macOS签名应用程序*

您需要一些先决条件来签署macOS应用程序。首先，您必须是Apple Developer Program（https://developer.apple.com/programs/）的注册会员。其次，您必须安装Xcode。这可以从Mac App Store或Apple的开发者网站（https://developer.apple.com/xcode/download/）下载。除了安装Xcode之外，还需要安装Xcode命令行工具。要设置它们，请从命令行键入xcode-select -install，然后按照所有后续提示进行操作。

**创建证书**

要签署您的应用程序，您需要通过iTunes Connect或Xcode内部创建证书。我们在第16章的iTunes Connect中度过了一段不愉快的时间，所以这次我们将在Xcode中进行。

我们生成两个证书：“Developer ID Application”和“Developer ID Installer”证书。要创建这些证书:

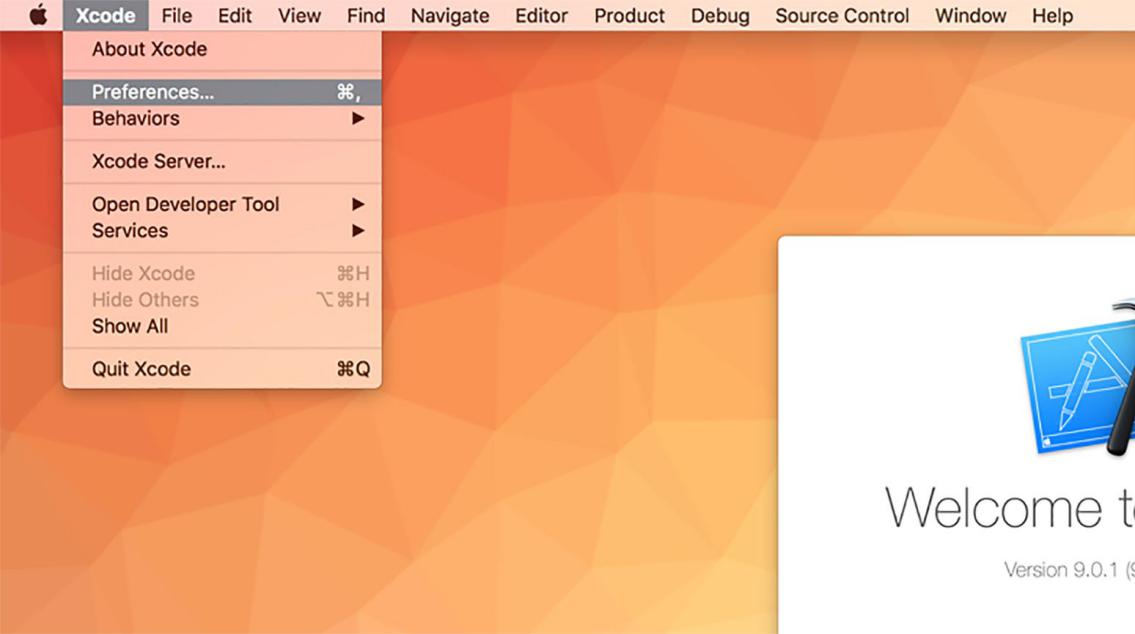
1.打开Xcode

2.从Xcode菜单中选择Preferences（如图15.5所示）

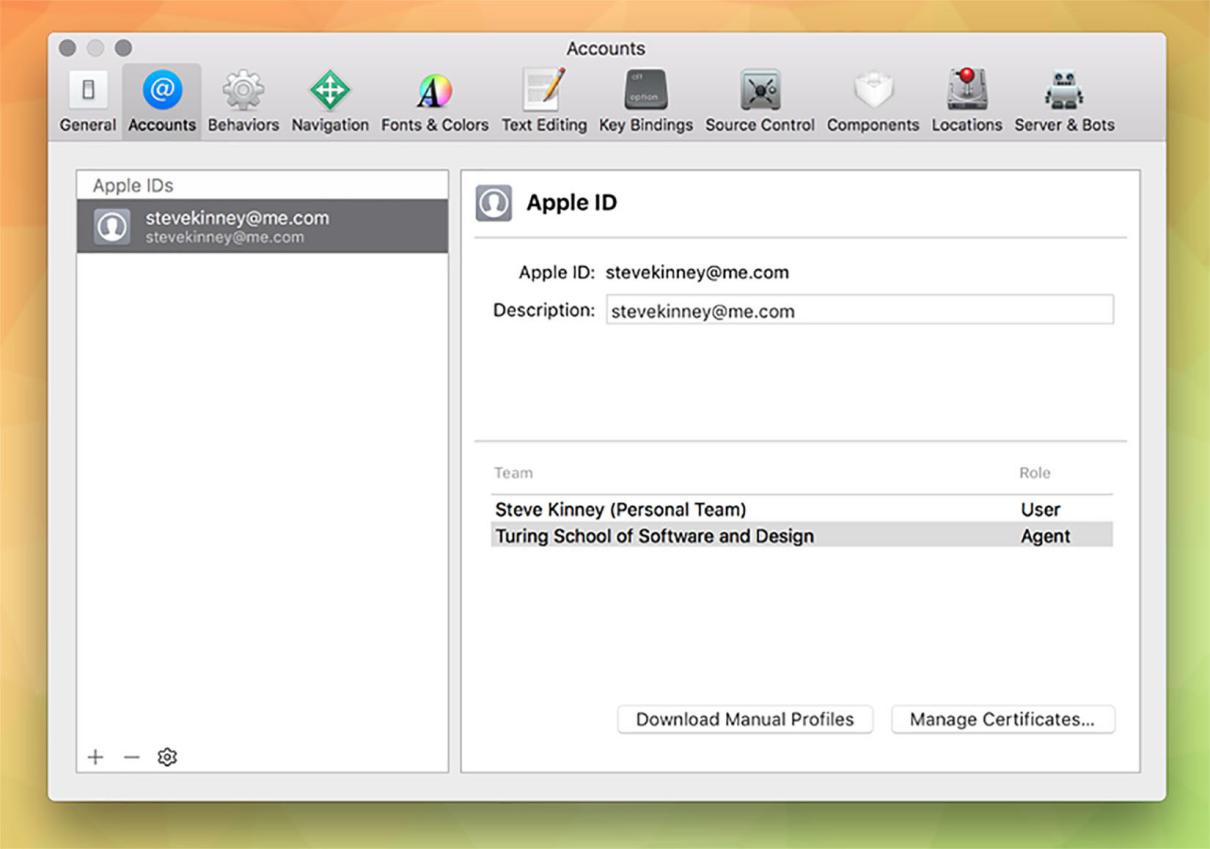
3.选择Accounts选项卡

4.选择您的团队

5.单击Manage Certificates按钮（如图15.6所示）



**图15.5 从应用程序菜单中选择首选项。**

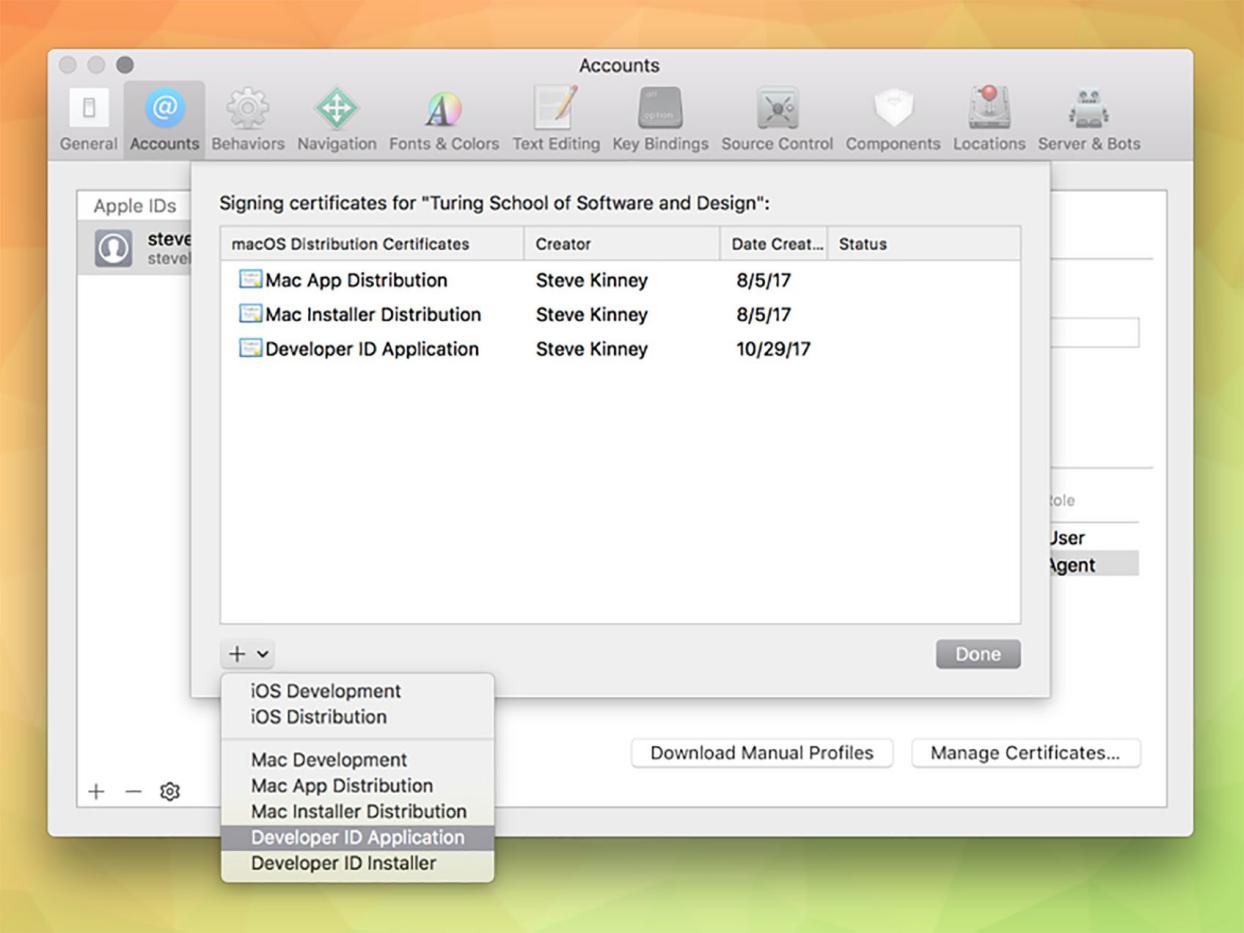


**图15.6 从Accounts选项卡中选择您的团队，然后单击Manage Certificates。**

6.单击左下角的+按钮

7.从下拉菜单中选择Developer ID Application（如图15.7所示）

8.使用Developer ID Installer重复此过程



**图15.7 从下拉菜单中选择Developer ID Application和Developer ID Installer。**

Xcode会自动将这些证书添加到您的Keychain中。当打包应用程序并签名时，Electron Packager会查看这些内容。首先，我们需要修改package.json中的打包脚本。

**清单15.8 更新的Mac打包脚本：./package.json**

"build-mac": "electron-packager . --platform=darwin

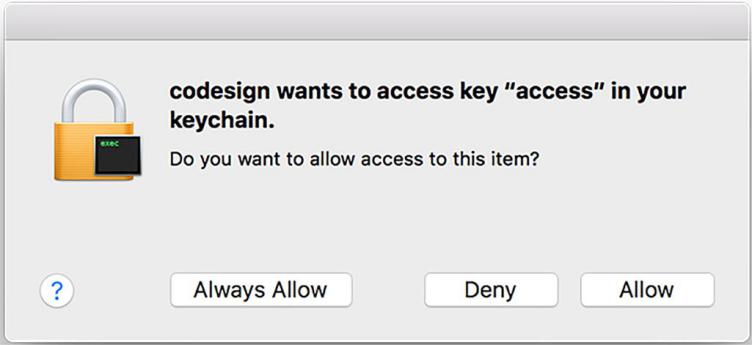
➥--out=build --icon=./icons/Icon.icns --asar –overwrite

➥--app-bundle-id=\"net.stevekinney.firesale\"

➥--app-version=\"1.0.0\" --build-version=\"1.0.100\" **--osx-sign**",

这一行有点长，我们在第16章开始讨论它，但是现在，我们在现有脚本中添加了一些重要的新参数：--app-bundle-id,--app-version,--build-version和--osx-sign。

运行npm run build-mac时，会多次提示您输入密码，如图15.8所示，因为Electron Packager访问您的钥匙串以使用证书对您的应用程序进行签名。



**图15.8 macOS的codesign实用程序访问您的钥匙串，以使用您生成的证书对应用程序进行签名。**

###### *在Windows上构建安装程序和代码签名*

在Windows上，您需要代码签名证书。Microsoft建议从其文档中列出的证书颁发机构列表中购买证书（http://mng.bz/3VEH）。

您仍然可以在没有证书的情况下构建安装程序，但是您可能无法将应用程序分发给没有证书的用户。Microsoft的SmartScreen筛选器可能会阻止您的应用程序被下载，并且许多防病毒程序可能会将您的应用程序错误地标记为恶意软件。出于本章的目的，当我们完成构建安装程序的过程时，我会向您展示插入证书的位置，如果您继续前进并决定购买一个。

Electron团队维护着一个有用的库，用于在Windows上构建安装程序。您可以使用npm install --save-dev electron-winstaller安装此工具。我们创建一个名为scripts的新文件夹，并创建一个名为scripts/windows.js的文件来存储Windows安装程序的配置。将清单15.9中的内容添加到我们刚创建的文件中。

**清单15.9 Windows安装程序配置: ./scripts/window.js**

//导入该功能以创建Windows安装程序

const { createWindowsInstaller } = require('electron-winstaller'); const path = require('path');

//找到应用程序图标的路径

const iconPath = path.resolve( dirname, '../icons/Icon.ico');

const result =

//创建安装程序。这会返回一个承诺。

createWindowsInstaller({ title: 'Fire Sale',

authors: 'Steve Kinney',

//找到您在第14章中首次构建的打包应用程序。

appDirectory: path.resolve(

dirname,

'../build/Fire Sale-win32-x64'

),

outputDirectory: path.resolve(

dirname,

'../build/Fire Sale-win32-x64-installer'

),

icon: iconPath,

**指定要生成安装程序的目录**

setupIcon: iconPath, name: 'FireSale',

setupExe: 'FireSaleSetup.exe', setupMsi: 'FireSaleSetup.msi',

});

result

.then(() => console.log('Success'))

**设置应用程序本身的图标**

**设置安装程序包的图标。 我选择使用相同的图标。**

如果安装程序创建成功，承诺将被解析。

.catch(error => console.error('Failed', error));

**如果发生错误并且承诺失败，将错误记录到控制台。**

如果您有方便的证书，可以将它们添加到配置对象。certificateFile应指向证书所在的路径。此过程类似于我们找到iconPath的方式。certificatePassword是证书的密码。不要将密码存储在版本控制中,特别是如果您的应用程序是或者将是开源的。

**设置SQUIRREL事件**

如果您真的很兴奋并双击FireSaleSetup可执行文件，您可能会注意到事情有点奇怪。您会看到加载GIF，然后应用程序立即打开。它没有添加桌面快捷方式，也没有任何其他方式返回应用程序。

卸载应用程序会增加一些奇怪的东西。单击Add and Remove Programs设置面板中的按钮时，您会看到应用程序在最终卸载之前再次打开。我们需要解决这个问题。

Windows安装程序使用Squirrel.Windows框架进行设置。当您第一次启动应用程序，或者正在卸载应用程序，更新或查找可用更新时，Squirrel会将参数传递给您的应用程序。幸运的是，在Windows上使用Squirrel可以从非常简单到非常简单。让我们以非常简单的方式开始，然后我将向您展示引擎盖下的内容。

最简单的入门方法是使用npm install electron-squirrel-startup安装electron-squirrel-startup，将工作委托给其他人。安装此软件包后，将此列表中的代码添加到./app/main.js。.

**清单15.10 在主进程中设置Squirrel事件：./ app/main.js**

const { app, BrowserWindow, dialog, Menu } = require('electron'); const createApplicationMenu = require('./application-menu'); const fs = require('fs');

require('./crash-reporter'); **if(require('electron-squirrel-startup')) return;**

// …

**如果electron-squirrel-startup中的函数返回true，则退出主进程。**

通过这个简单的更改，我们在使用FireSaleSetup.exe时遇到的所有奇怪之处都被平方。那么，这个模块中发生了什么？ 该模块是开源的，可以在GitHub上找到（https://github.com/mongodb-js/electron-squirrel-startup），但它是一个小文件，为了完整起见，我也在这里列出。

**清单15.11 electron-squirrel-startup**

var path = require('path');

var spawn = require('child\_process').spawn;

var debug = require('debug')('electron-squirrel-startup'); var app = require('electron').app;

var run = function(args, done) {

var updateExe = path.resolve(

path.dirname(process.execPath), '..', 'Update.exe'

);

debug('Spawning `%s` with args `%s`', updateExe, args); spawn(updateExe, args, {

**获取Squirrel更新程序的路径**

detached: true

}).on('close', done);

};

var check = function() {

**仅当应用程序在Windows上运行时，才执行以下任何操作**

**获取从命令行传递的第一个参数**

返回true，它满足主进程中的条件

if (process.platform === 'win32') { var cmd = process.argv[1];

debug('processing squirrel command `%s`', cmd); var target = path.basename(process.execPath);

**检查应用程序是作为安装程序还是更新程序运行**

if (cmd === '--squirrel-install' || cmd === '--squirrel-updated') { run(['--createShortcut=' + target + ''], app.quit);

return true;

}

**在桌面上创建应用程序快捷方式，然后退出**

**检查程序是否将被卸载**

if (cmd === '--squirrel-uninstall') {

run(['--removeShortcut=' + target + ''], app.quit);

return true;

}

从桌面删除快捷方式，然后退出

应用程序

if (cmd === '--squirrel-obsolete') {

app.quit(); return true;

}

}

return false;

};

module.exports = check();

**如果没有一个是true，则返回false。**

如果Squirrel做了什么，它会返回true。回想一下./app/main.js我们使用了条件，如果require（'electron-squirrel-startup'）返回true，我们就提前结束了主进程。这允许我们在必要时使用Squirrel框架，而不是在我们安装或卸载它时启动应用程序。您还可以看到，在安装和卸载时，Squirrel会根据需要创建和删除桌面快捷方式。

### *自动更新应用程序*

无论您是否已经完成了希望每个人都拥有的热门新功能，或者您正在尝试纠正在实施崩溃和错误报告后发现的严重错误，向用户推送更新的能力非常重要。

我们倾向于将此功能视为Web开发人员。每当我们将新应用程序部署到Web时，我们都可以合理地期望用户获得最新和最好的版本。但是，当我们构建桌面应用程序时，这并不一定正确。我们可以告诉用户，但不能保证他们会花时间下载它。

浏览器曾经遭受过这个问题。将发布新的JavaScript语言和Web平台功能，但您无法使用它们，因为用户通常不会定期更新其浏览器。如今，Chrome和Firefox等现代浏览器每六周左右就会向用户推出新版本。更新会自动下载，下次用户启动浏览器时，新的浏览器将被交换，而无需考虑繁琐的升级过程。

在这一点上听到Electron为我们的应用程序提供了一种机制，除了一些例外情况，这应该不足为奇。在撰写本文时，此功能仅限于macOS和Windows。目前在Linux上不支持自动更新Electron应用程序。

###### *在Electron中设置自动更新*

与崩溃报告一样，实现自动更新有两个方面：您的应用程序和托管应用程序版本的服务器。我们暂时深入了解细节，但让我们首先谈谈它如何在高水平上运作。配置autoUpdater模块后，每次应用程序启动时都会对发布服务器进行ping操作。如果服务器响应HTTP 204响应，则Electron知道它正在运行最新版本的应用程序。如果有新版本，则服务器将返回带有新版本URL的HTTP 200 JSON格式的响应。

您可以拥有自己的服务器来托管版本。那很好。如果不这样做，我们会创建一个简单的服务器，让您立即开始并演示自动更新。但这不是规定性的。只要你有一个服务器响应正确的HTTP状态代码和有效负载（如果有新版本），Electron对实现细节没有强烈的意见，我们暂时也不会。

为了实现这一点，我们检查是否正在使用该应用程序的生产版本。如果是这样，我们向服务器发送请求，要求提供最新版本的应用程序。如果存在新版本，请询问用户是否有兴趣进行更新。当他们同意时，告诉autoUpdater模块退出应用程序并代表我们安装新版本。

**清单15.12 实现Electron的autoUpdater：./app/auto-updater.js**

获取正在运行应用程序的当前OS

**检查我们是否在开发中运行此Electron应用程序**

const { app, autoUpdater, dialog, BrowserWindow } = require('electron');

const isDevelopment = app.getPath('exe').indexOf('electron') !== -1;

**获取当前应用程序的版本**

const baseUrl = 'https://firesale-releases.glitch.me';

const platform = process.platform;

const currentVersion = app.getVersion();

const releaseFeed =

**存储托管发行版的服务器的基本URL**

根据操作系统和应用程序版本创建从中请求更新的路径

`${baseUrl}/releases/${platform}?currentVersion=${currentVersion}`;

如果应用程序处于开发模式，则不检查更新...

if (isDevelopment) {

**...否则，将autoUpdater的feed设置为刚刚创建的URL。**

console.info('[AutoUpdater]', 'In Developement Mode. Skipping…');

} else {

console.info('[AutoUpdater]', `Setting release feed to ${releaseFeed}.`); autoUpdater.setFeedURL(releaseFeed);

}

autoUpdater.addListener('update-available', () => {

dialog.showMessageBox({

type: 'question',

buttons: ['Install & Relaunch', 'Not Now'], defaultId: 0,

},

message: `${app.getName()} has been updated!`,

**如果有更新，请执行提供的操作**

**删除windows-all-closed事件的事件侦听器**

**关闭所有窗口**detail: 'An update has been downloaded and can be installed now.' response => {

if (response === 0) {

setTimeout(() => {

app.removeAllListeners('window-all-closed'); BrowserWindow.getAllWindows().forEach(win => win.close()); autoUpdater.quitAndInstall();

**退出应用程序，然后安装更新**

}, 0);

}

});

});

module.exports = autoUpdater;

译注:

需要在main.js中引入:

const autoUpdater = require('./auto-updater');

在macOS上，如果应用程序尚未进行代码签名，则autoUpdater将失败。当我们开发应用程序时，我们使用的是尚未签名的版本。这两个事实是直接对立的。如果我们正在开发中，最好的做法是不获取更新。这个结果也很有意义，因为我们可能会在下一个版本的应用程序上工作，而不是最近发布的版本。

您可以通过几种方式实现此目的。例如，您可以尝试设置提要URL，然后在应用程序未进行代码签名时捕获错误。纯粹基于美学原因，这不是我喜欢做的事情。Electron没有API让我们知道我们是否处于开发阶段，但是当我们处于开发阶段时，我们正在使用埋在node\_modules目录中的Electron版本。就我而言，它位于./node\_modules/electron/dist/Electron.app/Contents/MacOS/Electron。在生产中，它位于应用程序包本身中。我利用这个有趣的事实来确定应用程序是否正在开发中。

如果它不在开发中，那么我们创建一个URL以根据运行应用程序的平台查找更新。从技术上讲，此URL可以是您想要的任何内容。如果你的设置与我们接下来一起编写的设置不同，你不必按照我的例子。

我们告诉autoUpdater模块向提供的URL发送请求，并询问是否有任何更新。如果有，则运行可更新的事件。我们为该事件设置了一个监听器。如果有更新，我们会询问用户是否要更新到最新版本的Fire Sale。如果他们同意，那么我们采取一系列步骤将他们优雅地转换到新版本。

首先，我删除了我们在./app/main.js中设置的事件监听器，如果所有窗口都在macOS上关闭，则会阻止应用程序退出。接下来，我们遍历窗口并关闭它们。我们这样做是为了提示用户保存对文件所做的任何更改。最后，我们调用autoUpdater.quitAndInstall( )，它无关紧要地关闭应用程序并安装新下载的更新。Squirrel代表我们处理所有这些事情。

###### *设置服务器以进行自动更新*

与崩溃记录器一样，我们设置了一个故意简单的服务器来通知您的用户更新。更强大的示例可能使用数据库来存储最新版本以及发行说明等。您可以将应用程序包存储在S3上，但是完成创建功能齐全的Web服务器的过程超出了本书的范围。

正如我前面提到的，我们的服务器必须完成一个简单的契约，才能很好地使用electronic的autoUpdater模块。如果应用程序与最新版本匹配，则服务器应返回具有204状态代码的响应。如果有可用更新，则服务器应返回一个JSON对象，该对象具有url属性，并将新应用程序的URL作为其值。

我在Glitch上托管了服务器（https://firesale-releases.glitch.me/）及其代码（https://glitch.com/edit/#!/firesale-releases），您可以在其中重新混音 你自己的目的。Glitch还托管应用程序，因此我们可以在Fire Sale中使用它来下载更新。我建议您访问最新版本的先前代码的链接，但我也在此处添加了带注释的版本。

**清单15.13 设置发布服务器**

const express = require('express'); const fs = require('fs');

const path = require('path'); const app = express();

app.use(express.static('public'));

const latestRelease = '1.2.0';  app.get("/", (request, response) => {

**这是Fire Sale的最新版本.**

**从URL参数中提取OS平台**

response.sendFile( dirname + '/views/index.html');

});

app.get('/releases/:platform', (request, response) => {

const { platform } = request.params;

**设置一个路由，该路由在传入可选版本的特定平台上侦听GET请求**

**如果匹配，则返回HTTP 204状态代码**

const { currentVersion } = request.query;

if (currentVersion === latestRelease) {

response.status(204);

return response.end();

}

if (platform === 'darwin') {

return response.json({

**从查询参数中提取当前版本**

检查当前版本是否等于之前引用的最新版本

url: …

});

}

**如果平台是macOS，则返回带有URL的有效负载到macOS的最新捆绑包**

if (platform === 'win32') {

return response.json({

url: …

});

**如果平台是Windows，则返回带有URL的有效内容，以获取最新的Windows捆绑包**

}

if (platform === 'linux') {

return response.json({

url: …

});

**如果平台是Linux，则返回带有URL的有效负载到最新的Linux捆绑包**

}

response.status(404).end();

});

const listener = app.listen(process.env.PORT, () => {

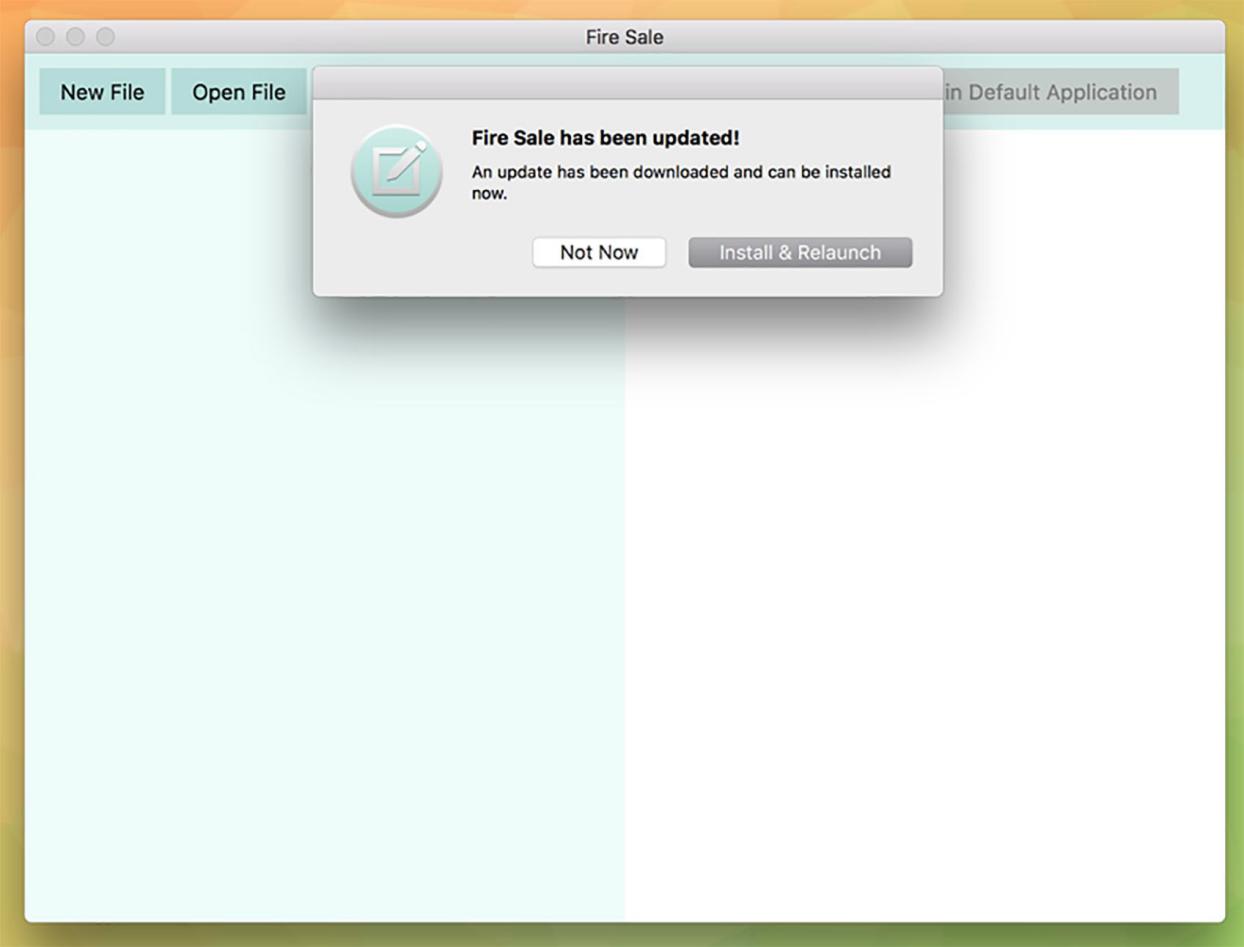
console.log('Your app is listening on port ' + listener.address().port);

});

我开始在变量中存储对Fire Sale的最新版本的引用。接下来，我建立了一个动态路线。如果您还记得，Fire Sale会根据其运行的平台请求更新。在macOS，Windows和Linux上运行的版本分别从/ releases/darwin，/ releases/win32和/ releases/linux请求更新。我们还检查版本号是否包含在查询参数中。

如果当前版本和最新版本匹配，那么我们将使用204状态代码进行响应，以便让应用程序知道它当前正在运行最新版本。如果它们不匹配，我们假设有更新。然后，我们检查用户请求的平台，并向他们发送最新版Fire Sale的适当风格的URL，如图15.9所示。

如果您使用了我为autoUpdater提供的代码并将其指向https://firesale-releases.glitch.me/，那么您应该收到有关Fire Sale非常重要的更新的通知。



**图15.9 Fire Sale会在服务器上有新版本时自动通知用户。**

### *本章小结*

* + - * Electron包含用于处理崩溃报告和自动更新的内置模块。
      * 当应用程序彻底崩溃时，崩溃报告以迷你转储格式发送报告。
      * 还可以监视未捕获的异常并将其报告给服务器以获得更多信息。
      * 应用程序应进行代码签名，以验证它们是否未被篡改。

- 在macOS上，可以从Xcode生成代码签名证书。

- 在Windows上，开发人员可以从证书颁发机构购买证书。

* + - * 可以使用npm上的electron-winstaller和electron-squirrel-startup软件包轻松创建Windows安装程序。
      * Electron的autoUpdater模块在启动时检查更新。
      * autoUpdater使用开源Squirrel框架来管理安装和更新。

*通过Mac应用商店发布您的应用程序*

***本章包括***

* 包装Mac App Store的应用程序
* 创建证书并签署您的应用
* 将您的应用程序上传到iTunes Connect

在上一章中，我们从本书的前面部分构建了两个项目。这消除了我们的用户安装Node.js并使用命令行启动我们的应用程序的需要。在本章中，我们将讨论分发使用Electron构建的应用程序的策略。

### *16.1 将您的应用程序提交到Mac应用商店*

通过Mac应用商店发布您的应用程序有一些明显的优势。首先，如果您有兴趣沿着这条路走下去，您可以轻松地为您的应用收取费用。其次，Mac App Store代表您处理应用程序更新。缺点是注册Apple的开发者计划不是免费的 - 如果您决定为您的应用程序收费 - Apple将您的应用程序的销售价格降低30％。

将您的应用程序提交到Mac应用商店除了简单地打包您的应用程序之外还需要很多步骤,就像我们在第14章中所做的那样。您还必须创建安全证书并签署您的应用程序以确保用户正在收到它的正式版本。签署您的应用程序是一般的好习惯，即使您不打算将应用程序提交到Mac App Store，本章也值得一读。签署和上传应用程序的过程主要通过Apple提供的本机和基于Web的UI执行。因此，本章具有高度的视觉效果。

在将应用程序提交到Mac App Store之前，必须在Mac上安装Xcode，并且需要注册Apple Developer Program。Mac App Store上提供的Xcode是一个大型应用程序，因此在注册开发人员计划之前首先开始下载。要执行此操作，请访问https:// developer.apple.com/programs/，并以个人或组织的身份注册该计划。

###### *签名应用*

代码签名是一种技术，允许您证明您的应用程序实际上是由您创建的，而不是由某些冒名顶替者创建的。当用户首次安装您的应用程序时，操作系统会跟踪您包含的证书。只有您可以生成此证书，这意味着如果它发生更改，则新版本的应用程序可能来自恶意攻击者。

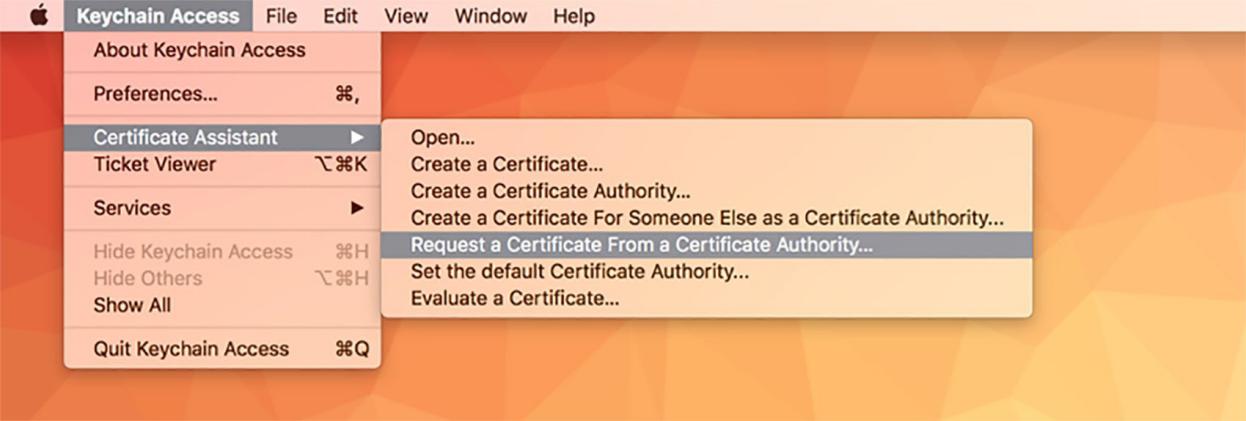
我们需要两个证书：一个用于应用程序本身，另一个用于我们上传到Mac App Store的安装程序,我们将在本章后面创建。要创建这些证书，请打开Keychain Access应用程序，如图16.1所示，位于/Applications/Utilities目录。此应用程序包含在macOS中，即使您尚未安装Xcode也应该可用。

**Keychain Access的其他用途**

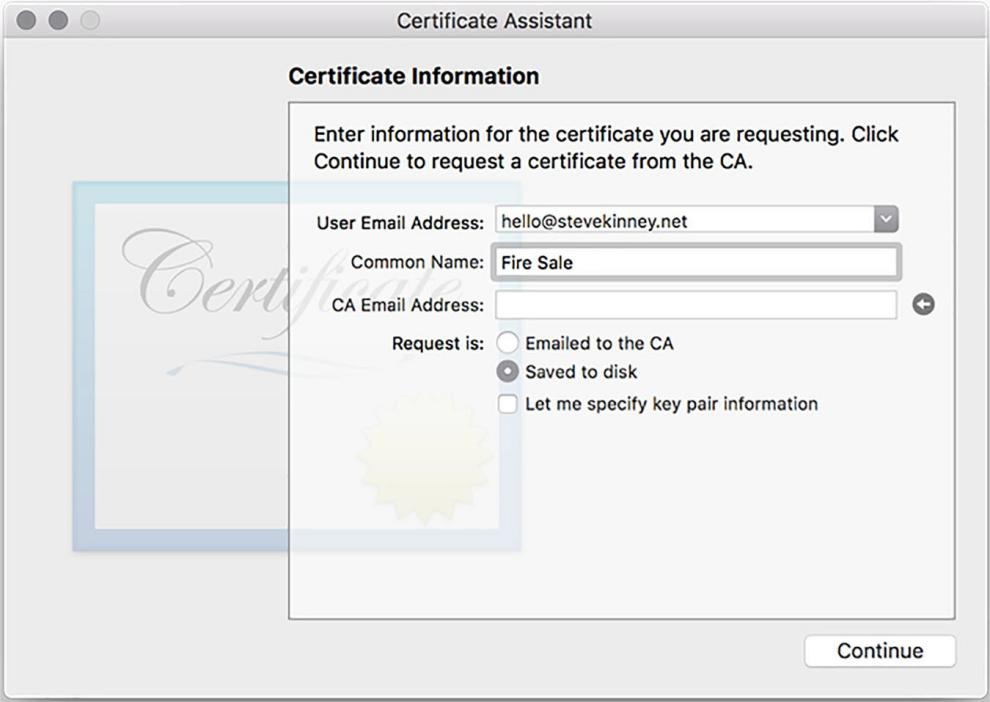
正如您可能从名称中猜到的那样，Keychain Access不仅仅为签名应用程序生成证书。这是macOS存储所有已保存密码和证书的地方。它还存储您要求它记住的无线网络的所有密码。如果您需要查找密码，可以使用Keychain Access进行查找。

打开应用程序，导航到Keychain Access应用程序菜单，然后选择Certificate Assistant，然后从证书颁发机构申请证书，如图16.1所示。此操作会触发一个对话框，如图16.2所示，您可以在其中添加有关要生成的证书的其他信息。

确保在创建证书时选择“保存到磁盘”。单击“继续”后，系统将提示您保存名为CertificateSigningRequest.certSigningRequest的文件。您选择保存文件的位置取决于您,只要您以后可以再次找到它，它就没有任何区别。

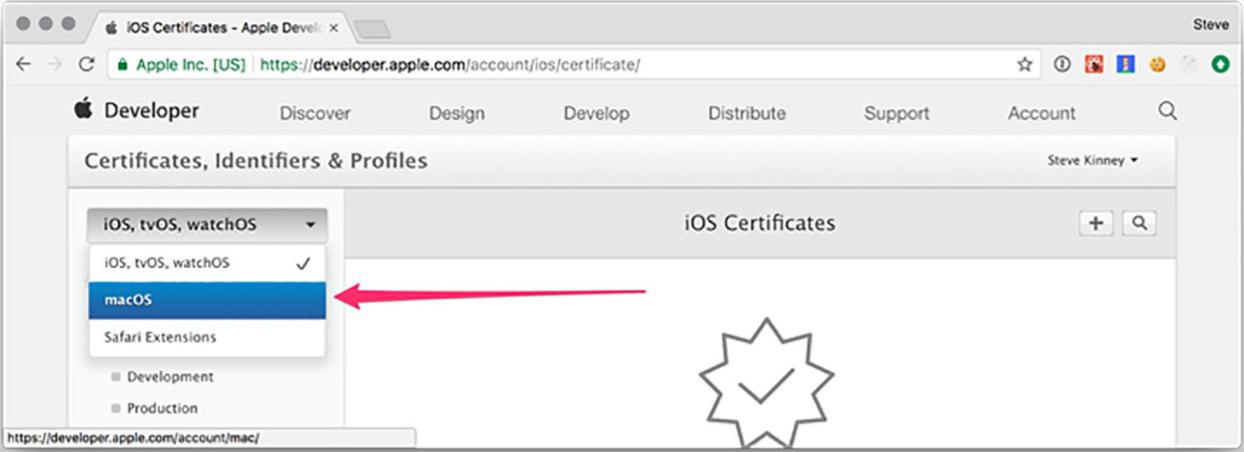


**图16.1 使用Keychain Access请求证书。**



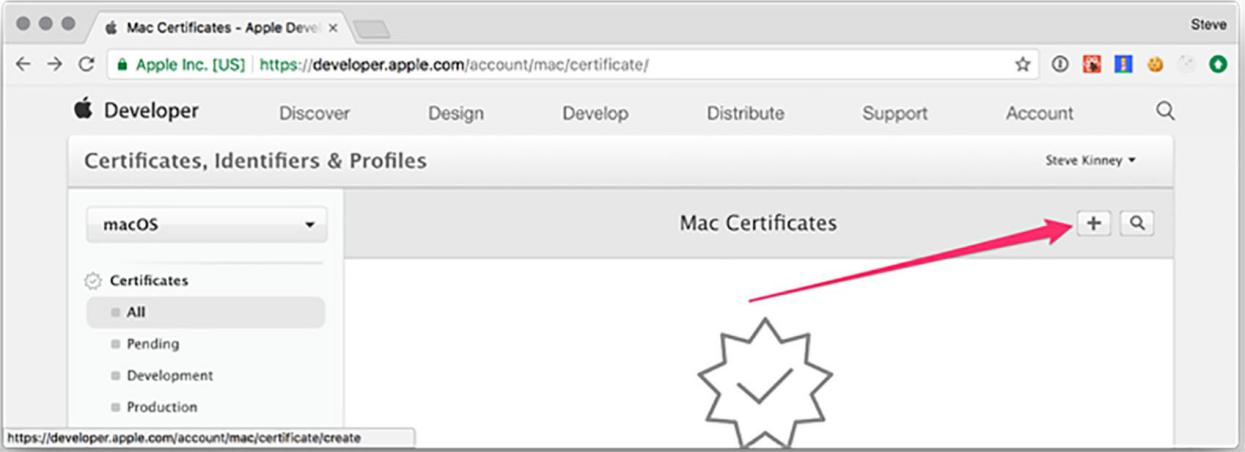
**图16.2 输入应用程序签名证书请求的信息。**

获得证书后，您需要让Apple了解它。访问Apple Developer Program Member Center（https://developer.apple.com/account），然后从左侧边栏中选择证书，ID和配置文件。默认情况下，将打开用于管理iOS证书的页面。选择macOS作为目标平台，如图16.3所示。



**图16.3选择macOS作为目标平台。**

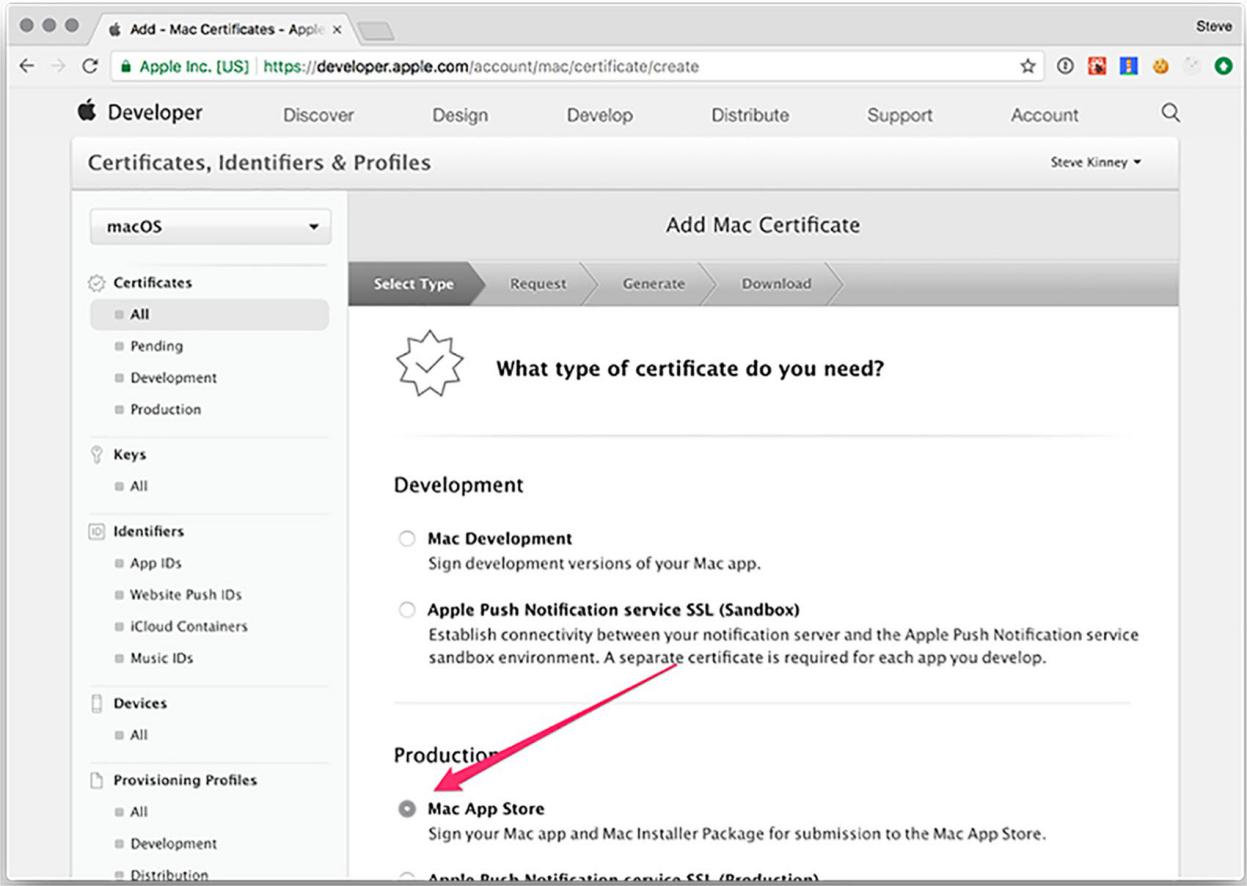
从iOS切换到macOS后，您会看到过去上传的证书列表或空状态。无论哪种方式，因为我们正在上传新证书，请单击屏幕右上角的+按钮，如图16.4所示。



**图16.4 添加先前生成的证书请求。**

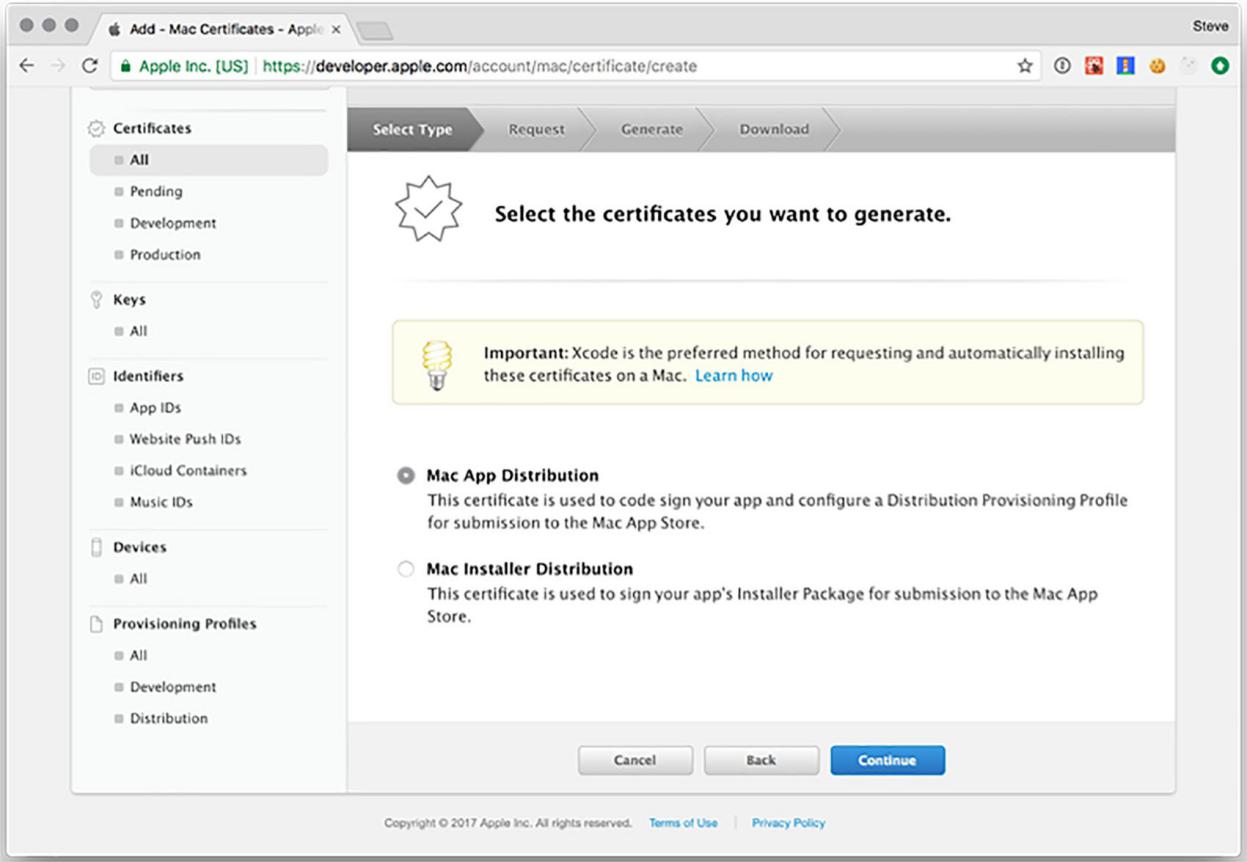
证书可用于各种目的。除了对应用程序进行签名之外，还可以使用证书对服务器生成的推送消息进行签名，您可能已在移动平台上看到这些消息，以通知用户当前未打开的应用程序中发生了某些事情。

这超出了本书的范围，也是另一天的主题。正如您可能已经猜到的那样，我们希望创建一个生产就绪的Mac App Store版本的Fire Sale，并且需要一个Mac App Store证书来实现这一点。滚动浏览顶部的开发选项，然后选择Mac App Store，如图16.5所示。



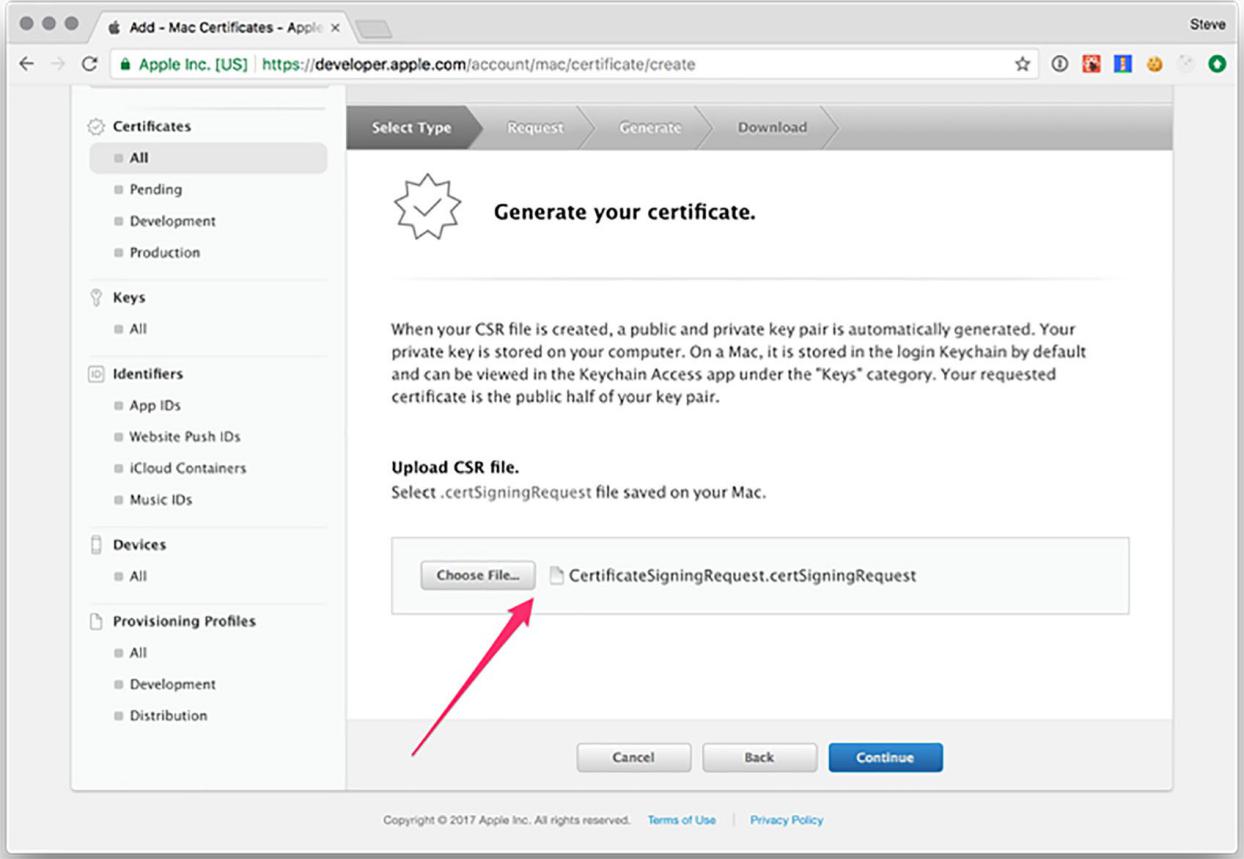
**图16.5 选择Mac App Store作为您要求的证书类型。**

图16.6到16.8显示我正在创建Mac App Distribution证书。但实际上，您需要执行两次这些步骤,选择Mac App Distribution一次，选择Mac Installer Distribution一次。这两个过程完全相同，最后您会收到两个证书。唯一的区别是您在流程的最开始选择哪个单选按钮。



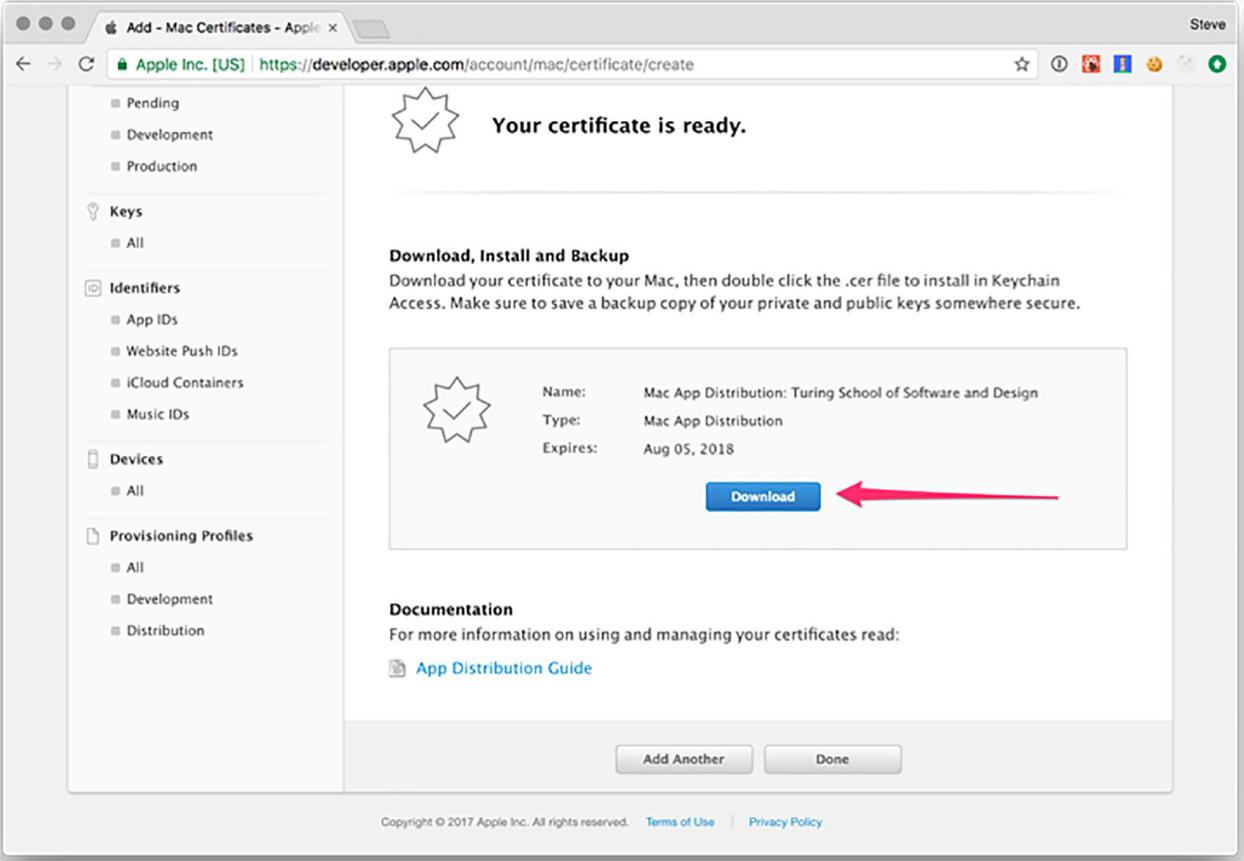
**图16.6 为应用程序和安装程序生成证书。**

在本章的前面，我们创建了一个扩展名为certSigningRequest的文件。我们使用此文件向Apple申请证书，该证书由Apple Worldwide Developer Relations Certification Authority签署。签署应用程序时，使用的密钥只能使用计算机本地存储的私钥生成。您必须小心使用此私钥。这是签署与公钥匹配的应用程序的唯一方法，任何有权访问您私钥的人都可以代表您签署应用程序。



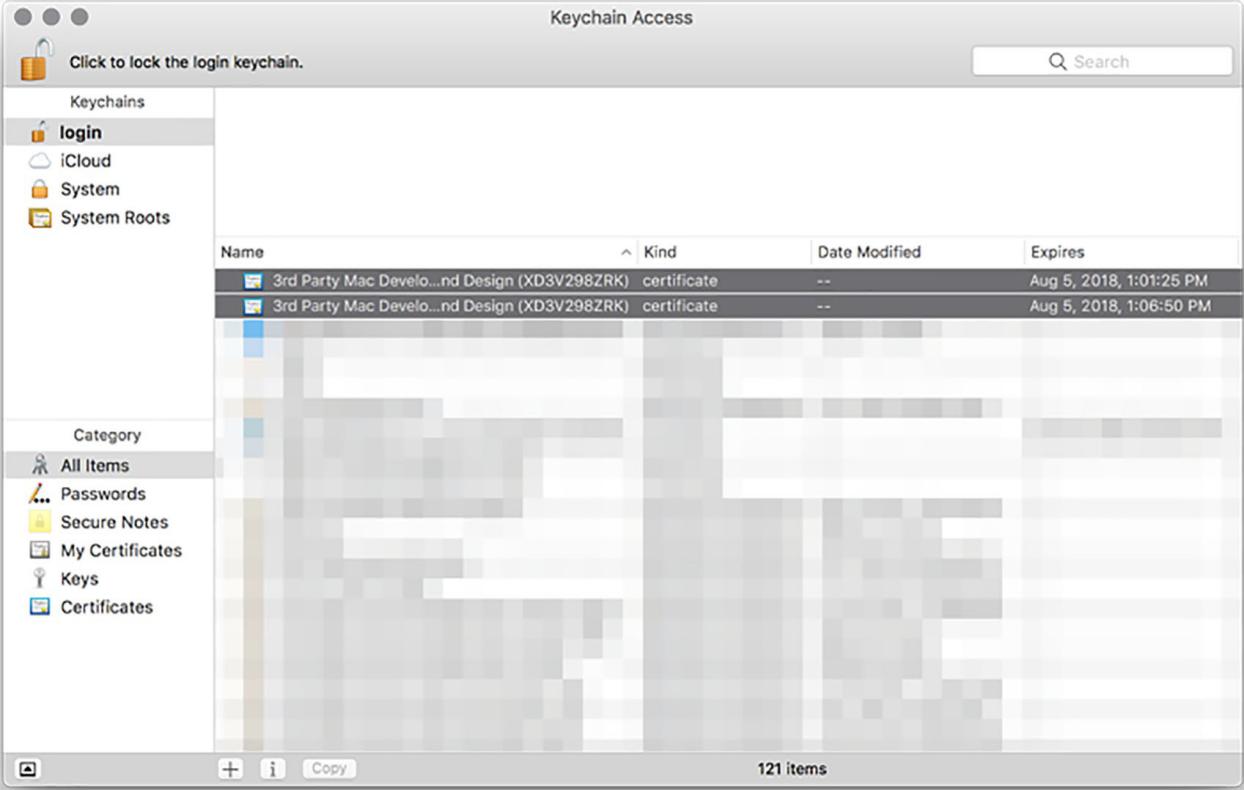
**图16.7 上传您之前创建的证书签名请求文件。**

出现提示时，上传您在Keychain Access中生成的证书签名请求。Apple签署证书并提示您下载证书，如图16.8所示。下载证书后，您可以双击它以自动添加到Keychain Access。



**图16.8 下载已完成的证书。**

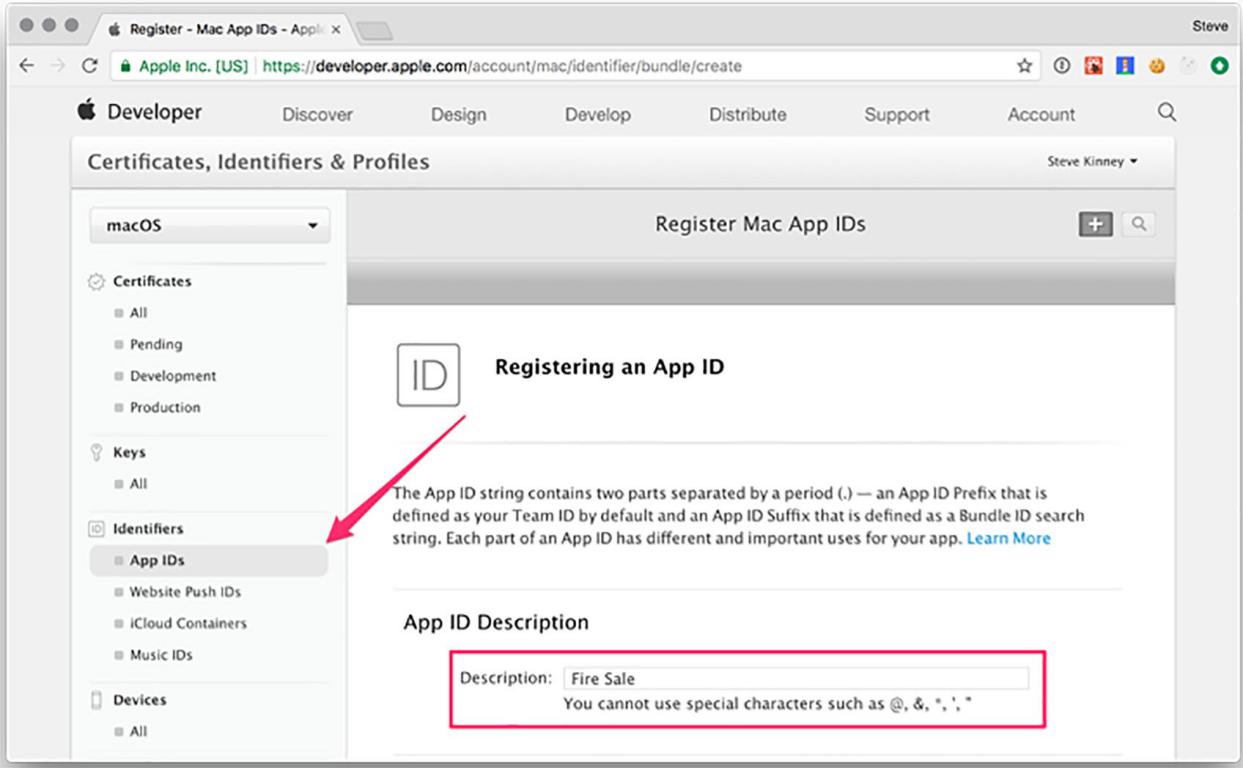
完成Mac App Distribution和Mac Installer Distribution证书的此过程后，您应该在Keychain Access中看到两个证书，如图16.9所示。恭喜,您现在已准备好签署您的应用程序并开始准备提交到Mac App Store。在下一节中，我们将开始向Apple注册应用程序并提供要在App Store中列出的元数据的过程。



**图16.9 Keychain Access中列出的应用程序和安装程序证书。**

###### *使用Mac应用商店注册您的应用程序*

在您的证书到位后，您必须先将应用程序本身注册到Apple，然后才能上传应用程序二进制文件。即使您无意通过Mac App Store分发应用程序，签署您的应用程序并向Apple注册也是一种很好的做法。当用户试图在macOS的现代版本中打开一个未签名的应用程序时，苹果会显示那些令人生畏的看门人错误。此外，根据用户的设置，可能根本不允许他们打开未签名的应用程序。在Certificates, Identifiers, 和Profiles部分中，从左侧边栏中选择“App ID”，如图16.10所示。



**图16.10 注册您的App ID。**

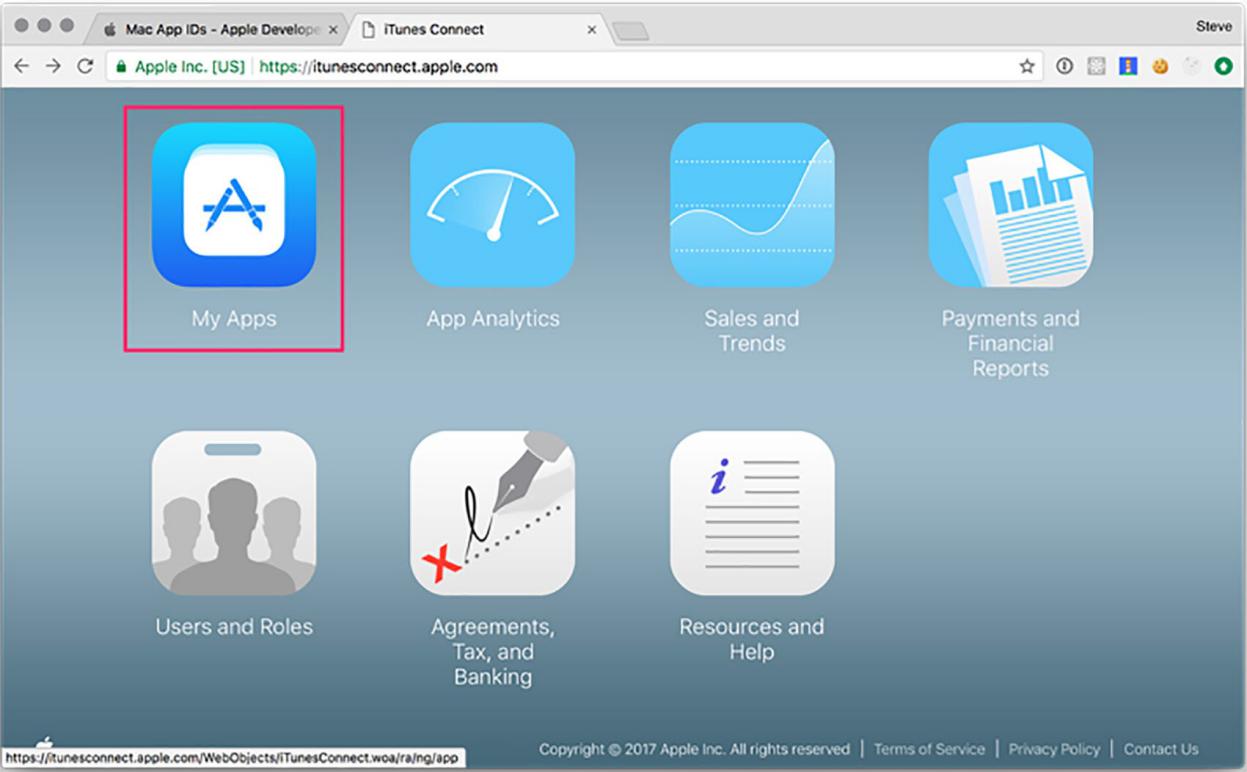
应用程序ID是应用程序的唯一标识符。创建应用程序ID包含三个步骤。第一个是给它一个App ID描述。这是应用程序的通俗名称，通常是用户在App Store中查看时所看到的应用程序标题。

第二个是应用程序请求访问的服务列表，例如发送推送通知，显示地图或从iCloud保存或检索文件。Fire Sale不使用任何这些，因此我们将坚持使用默认值。第三，您创建了一个App ID后缀，它是您应用程序的真正唯一标识符。这很像一个倒置的网络域名。在这种情况下，我使用net.stevekinney.firesale而不是firesale.stevekinney.net，这是你可能期望在网上找到的东西。

###### *将应用程序添加到iTunes Connect*

你还没有走出困境。下一步是将应用程序添加到iTunes Connect（https://itunesconnect.apple.com/），它代表应用程序在Mac App Store中的列表。在这里，您可以添加应用程序的说明，屏幕截图等。需要注意的是：我不会涵盖上传屏幕截图和各种内容的各个方面，但我会告诉您，Apple会认真对待并拒绝您的应用程序，如果您未能包含所有必要信息,包括链接 到有效的网页，用户可以在其中找到对您的应用程序的支持。

进入iTunes Connect后，单击左上角的My Apps图标，如图16.11所示。在本节中，您将看到现有应用程序的列表，如果这是您的第一个应用程序，则为空状态。单击窗口左上角的+按钮，然后选择New Mac App，如图16.12所示。

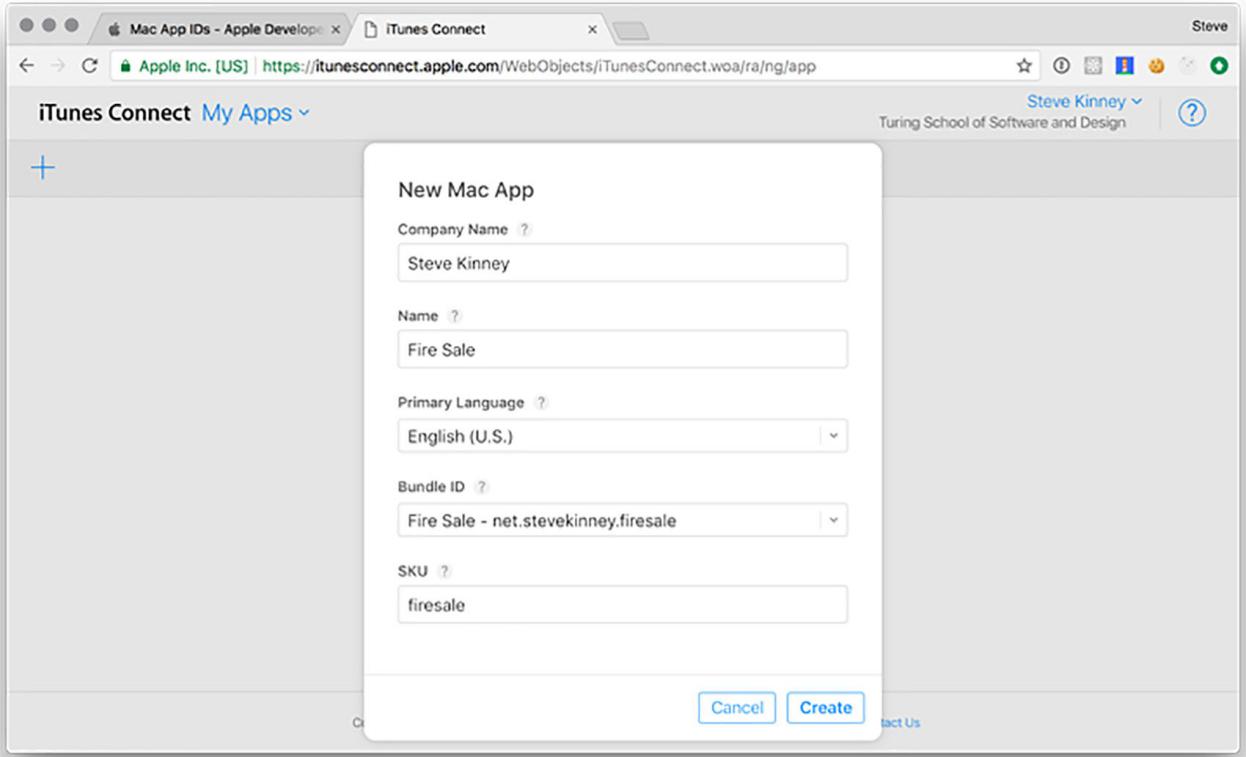


**图16.11 在iTunes Connect中选择我的应用程序。**



**图16.12 在iTunes Connect中添加新的Mac应用程序。**

在此阶段，如图16.13所示，系统会要求您提供有关应用程序的一些粗略信息：名称，主要语言，您在上一节中创建的应用程序ID描述和后缀等。SKU是可选的，仅用于识别应用程序销售以进行会计核算。



**图16.13 提供有关iTunes Connect中应用程序的详细信息。**

###### *为Mac应用商店打包应用程序*

在第14章中，我们在package.json文件中执行了所有内联配置。要打包我们的Mac App Store应用程序，我们需要对其进行签名以及提供一些其他元数据以帮助进行操作系统集成。macOS应用程序使用类似于XML的属性列表（.plist）文件来存储重要的元数据，例如名称以及应用程序从操作系统请求的权限。

一个简单的应用程序可能能够使用单个属性列表文件。但Electron应用并不是简单的应用。如果你从第1章回忆起来，每个进程:主进程和渲染器进程都是分开的。这意味着我们需要为子进程以及主应用程序进程设置权限。此外，Electron捆绑了自己的框架以及用于媒体播放的库。这些也必须签名。

为此，我们设置了三个属性列表文件：info.plist存储应用程序的通用元数据，parent.plist存储父进程的沙箱权限，child.plist存储子进程的沙箱权限。在此示例中，子进程扩展了父进程的权限。最终使用构建脚本安装它们。因此，我决定将其中的每一个放在名为./scripts/mas的文件夹中。我的命名约定并不重要。如果您想将这些文件存储在其他地方，那也很好。Electron API Demos应用程序将属性列表存储在其assets文件夹中。为了简单和清晰起见，我选择将列表16.1-16.3中的所有文件保存在与我们将很快创建的构建脚本相同的文件夹中。

**清单16.1 设置应用程序：./scripts/mas/info.plist**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ["http://www.apple.com/DTDs/PropertyList-1.0.dtd">](http://www.apple.com/DTDs/PropertyList-1.0.dtd)

<plist version="1.0">

<dict>

<key>CFBundleURLTypes</key>

<array>

<dict>

<key>CFBundleURLSchemes</key>

<array>

<string>firesale</string>

</array>

<key>CFBundleURLName</key>

<string>Fire Sale</string>

</dict>

</array>

<key>ElectronTeamID</key>

<string>XD3V298ZRK</string>

</dict>

</plist>

**这与package.json中的productName字段相同.**

您的Apple团队ID。

**清单16.2 设置父权限：./scripts/mas/parent.plist**

**使用团队ID和捆绑标识符标识您的团队**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ["http://www.apple.com/DTDs/PropertyList-1.0.dtd">](http://www.apple.com/DTDs/PropertyList-1.0.dtd)

<plist version="1.0">

<dict>

<key>com.apple.security.app-sandbox</key>

<true/>

<key>com.apple.security.application-groups</key>

<string>XD3V298ZRK.net.stevekinney.firesale</string>

**Mac App Store应用程序必须使用沙盒。**

允许您的团队访问其他应用程序的沙盒容器

<key>com.apple.security.files.user-selected.read-write</key>

<true/>

</dict>

</plist>

**请求使用Open和Save对话框读取和写入文件的权限**

Mac App Store上的所有应用程序都必须启用沙盒，沙盒将每个应用程序封装到一个隔离的容器中，并限制其访问操作系统其余部分的能力。Apple以安全的名义这样做。也就是说，如果隔离到沙箱，很多应用程序都无法完成工作。Apple了解这一点并允许应用程序清楚地说明他们需要哪些权利。我们知道Fire Sale使用“打开”和“保存”对话框来从文件系统读取和写入，因此我们明确指出我们需要该访问权限。

沙盒的一个例外是允许应用程序进入由同一开发人员创建的其他应用程序的容器中。我们从未在Fire Sale和Clipmaster 900或Jetsetter之间建立紧密集成，但我们可能会下线。说明应用程序组让macOS知道这个应用程序是一个家庭的一部分 - 即使它现在是一个人的家庭。

**清单16.3 扩展权限：./scripts/mas/child.plist**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ["http://www.apple.com/DTDs/PropertyList-1.0.dtd">](http://www.apple.com/DTDs/PropertyList-1.0.dtd)

<plist version="1.0">

<dict>

<key>com.apple.security.app-sandbox</key>

<true/>

<key>com.apple.security.inherit</key>

<true/>

</dict>

</plist>

**告诉Apple，子进程将继承与主进程相同的权限**

有了三个属性列表，我们使用electronic-packager来构建应用程序。这种方法类似于我们在第14章中使用的技术，有一些重要的变化。首先，我们告诉electron-packager我们的目标是mas（Mac App Store）平台，而不是darwin用于常规macOS应用程序。我们还编写了相当繁琐的代码签名过程，这些代码签署了我们的应用程序使用的所有进程，框架和库。该脚本使用Xcode codesign命令行界面工具。该脚本的灵感来自Electron Core Team用于构建Electron API演示应用程序的方法（https://github.com/electron/electron-api-demos）。在运行以下列表中的脚本之前，请确保已安装Xcode。您还需要确保使用本章前面生成的证书的名称。

**清单16.4 构建脚本**

#!/bin/bash

set -ex APP="Fire Sale"

electron-packager . \ "$APP" \

--asar \

--overwrite \

--platform=mas \

**此脚本的运行类似于第14章中package.json中使用的构建脚本.**

一个小的改变是我们正在为Mac App Store设置平台mas。

**选项采用属性列表并将其与Electron附带的列表合并.**

--app-bundle-id=net.stevekinney.firesale \

--app-version="$npm\_package\_version" \

--build-version="1.0.0" \

--arch=x64 \

--icon=./icons/Icon.icns \

--out=build \

--extend-info=scripts/mas/info.plist

APP\_PATH="./build/$APP-mas-x64/$APP.app" RESULT\_PATH="./build/$APP.pkg"

APP\_KEY="3rd Party Mac Developer Application:

**应用程序在electron-packager构建之后的位置。**

由Xcode为Mac应用程序商店准备好安装程序后，安装程序的目的地。

**安装程序的签名证书.**

➥ Turing School of Software and Design (XD3V298ZRK)"

INSTALLER\_KEY="3rd Party Mac Developer Installer:

➥ Turing School of Software and Design (XD3V298ZRK)"

FRAMEWORKS\_PATH="$APP\_PATH/Contents/Frameworks" CHILD\_PLIST="./scripts/mas/child.plist" PARENT\_PLIST="./scripts/mas/parent.plist"

**我们之前为应用程序本身签署的证书。**

Electron的框架和依赖关系的路径，也需要签名。

**用于主进程和父进程的属性列表。**

**用于子进程和依赖项的属性列表。**

codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/Electron Framework.framework/Versions/A/Electron

➥ Framework"

codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/Electron

➥ Framework.framework/Versions/A/Libraries/libffmpeg.dylib"

codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/Electron

➥ Framework.framework/Versions/A/Libraries/libnode.dylib"

codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/Electron Framework.framework"

codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/$APP Helper.app/Contents/MacOS/$APP Helper" codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/$APP Helper.app/"

codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/$APP Helper EH.app/Contents/MacOS/$APP Helper EH"

codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/$APP Helper EH.app/"

codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/$APP Helper NP.app/Contents/MacOS/$APP Helper NP" codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$FRAMEWORKS\_PATH/$APP Helper NP.app/"

codesign -s "$APP\_KEY" -f --entitlements "$CHILD\_PLIST"

➥ "$APP\_PATH/Contents/MacOS/$APP"

codesign -s "$APP\_KEY" -f --entitlements "$PARENT\_PLIST" "$APP\_PATH" productbuild --component "$APP\_PATH" /Applications --sign "$INSTALLER\_KEY"

"$RESULT\_PATH"

**使用Xcode构建安装程序包。**

**使用parent.plist中包含的沙盒权利对父应用程序进行代码签名。**

###### *配置应用程序类别*

我们可以在我们的属性列表中包含额外的元数据，这些元数据完全属于“nice to have.”的类别.Apple包含其官方文档中所有选项的详尽列表（http://mng.bz/2TDO），所以我 这里不会涵盖所有这些内容。

**清单16.5 向特定类别添加应用程序：./scripts/info.plist**

<key>NSHumanReadableCopyright</key>

<string>2017 Steve Kinney</string>

<key>LSApplicationCategoryType</key>

<string> public.app-category.developer-tools</string>

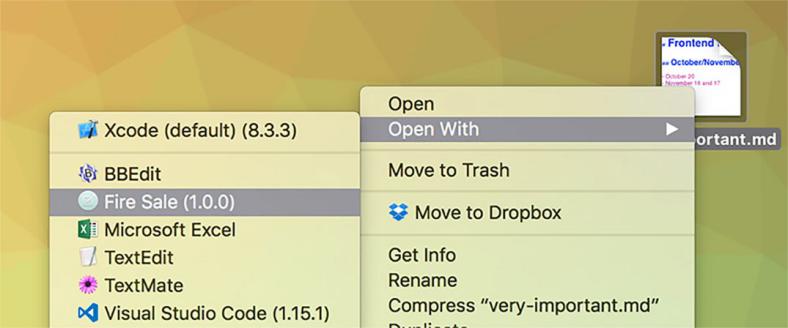
<key>LSApplicationSecondaryCategoryType</key>

<string public.app-category.productivity</string>

在上一个清单中，我们包括应用程序应列出的Mac App Store类别的元数据以及一些通用版权信息。您可以在Apple的官方文档（http://mng.bz/65Eg）中找到可用类别的完整列表。

###### *注册应用程序可以打开文件类型*

Fire Sale是一个适用于Markdown应用程序的应用程序。在开发过程中，它未被列为可以处理打开Markdown文件的应用程序。此外，我们无法将Markdown文件拖到Dock中的应用程序图标上。这种限制是因为尽管我们知道Fire Sale是为了使用Markdown文件而构建的，但macOS却没有。要添加对给定文件类型的支持,在这种情况下，Markdown文件,我们将此代码添加到./scripts/mas/info.plist。结果如图16.14所示。



**图16.14 Fire Sale现在已在macOS中注册为可以打开Markdown文件的应用程序。**

**清单16.6 设置Fire Sale可用于打开Markdown文件：/script/info.plist**

<key>CFBundleDocumentTypes</key>

<array>

<dict>

<key>CFBundleTypeExtensions</key>

**包含应用程序应识别的文件扩展名字符串数组**

***验证并上传您的应用* 309**

<array>

<string>markdown</string>

<string>mdown</string>

<string>md</string>

</array>

<key>CFBundleTypeRole</key>

<string>Editor</string>

<key>LSHandlerRank</key>

<string>Alternate</string>

</dict>

</array>

**告诉macOS该应用程序与文件类型之间的关系**

定义应为相应文件类型列出此应用程序的优先级

CFBundleDocumentTypes包含一组项目，用于定义此应用程序应该能够打开的文件类型。Fire Sale仅适用于Markdown文件（以及技术上的文本文件），但像Atom这样适用于各种文件类型的应用程序具有更长的列表，可以在其开源存储库中看到（http ://mng.bz/zQ4B）。

CFBundleDocumentTypes字典中的每个项都包含一个CFBundleTypeExtensions数组，该数组列出了应用程序应支持的文件类型。此外，CFBundleTypeRole键定义应用程序与文件类型的关系。Fire Sale编辑Markdown文件，因此我们将其列为编辑器是有道理的。可用选项包括Editor，Viewer，Shell和none。

许多应用程序可以打开像Markdown这样的文件类型。通过LSHandlerRank属性，您可以确定当用户双击文件时操作系统应为应用程序提供的优先级。如果应用程序创建给定的文件类型，则应选择所有者。对于只能由特定应用程序打开的专有文件类型，这是一个不错的选择。如果应用程序是作为默认应用程序的良好候选者，则应选择默认值。Fire Sale是一个很棒的应用程序，但用户可能更喜欢,通常用户希望在双击时打开Atom或Visual Studio Code。这就是我选择Alternate的原因。

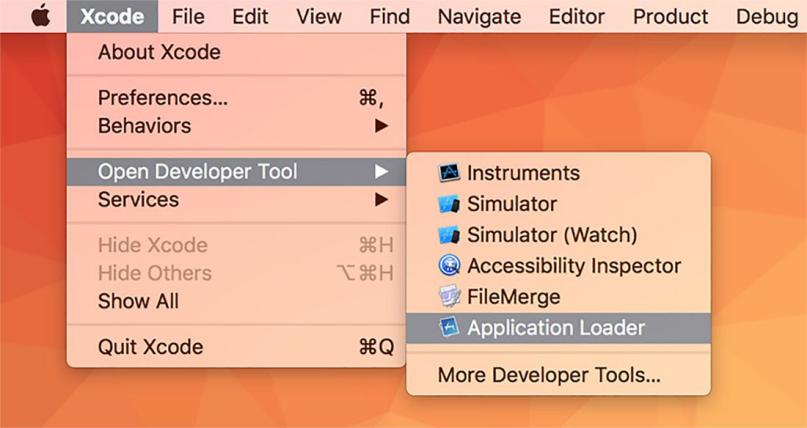
您也可以定义许多其他键。例如，如果您希望所有Markdown文件都具有将其与Fire Sale关联的特殊图标，则可以定义CFBundleTypeIconFile。可以在Apple的官方文档（http://mng.bz/4DTL）中找到可用选项的完整列表。

### *验证并上传您的应用*

您的应用程序已打包并签名。它的权利已经确定。它已准备好上传到Mac App Store。当然，下一步是将包上传到商店本身。您可以使用Xcode执行此操作。

要开始使用，请打开Xcode并导航到Xcode应用程序菜单。在Open Developer Tool下，您可以找到Application Loader，如图16.15所示。这是上传您的Electron应用程序的帮助应用程序。

打开Application Loader后，单击Deliver Your App。如图16.16所示，系统会提示您选择要上传的软件包 - 在本例中为Fire Sale.pkg，位于./build中。



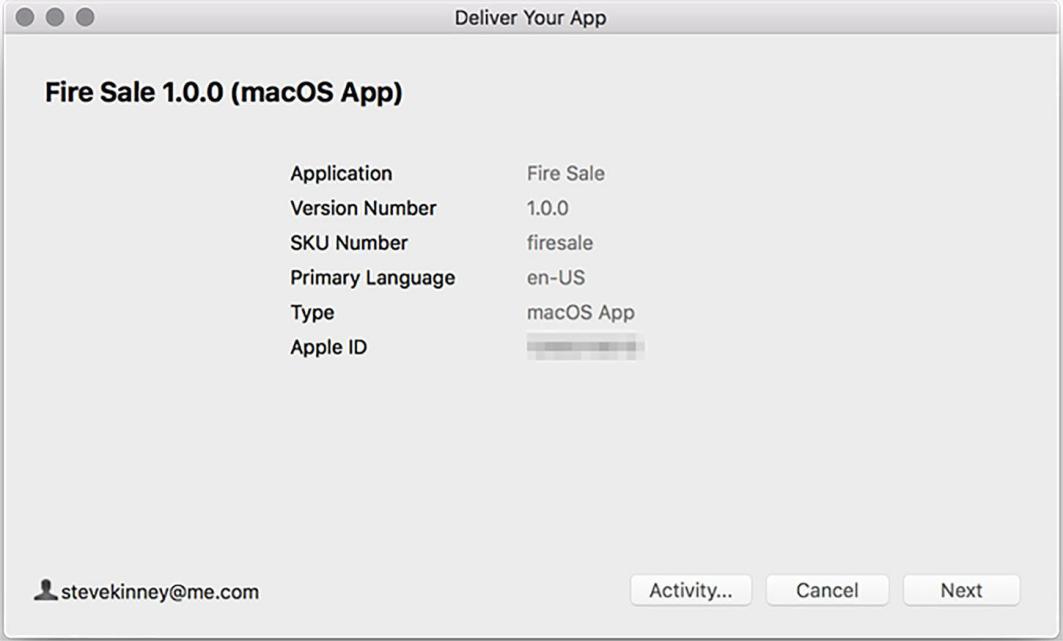
**图16.15 使用Application Loader将Electron应用程序上传到Mac App Store。**



**图16.16 在这种情况下，我们将一个完整的应用程序上传到Mac App Store。**

选择要上载的包后，可以查看其元数据，如图16.17所示。如果一切正常，请单击“下一步”开始上载过程。它不应该花很长时间，但这是基于您的连接速度和应用程序包的大小。

***本章小结* 311**



**图16.17 验证应用程序的元数据是否正确。**

### *完成接触*

Apple希望确保为用户提供最佳体验。因此，您必须提供屏幕截图，说明，指向用户可以获得帮助的页面的链接等。如果您未能做到这些，Apple很可能会拒绝该申请，并再次要求您提供此信息。我不打算在本书中深入介绍添加屏幕截图，主页和文档，因为它既相对简单，也可能在数字墨水在此页面上干涸时更改。

### *本章小结*

* + - 所有Mac App Store应用程序及其安装程序必须经过代码签名才能验证其真实性。
    - 您可以使用包含在macOS中的Keychain Access创建证书请求。
    - 在收到证书请求后，您可以将其上传到Apple的开发人员平台以接收签名证书。
    - 应用程序必须通过其开发人员平台向Apple注册。
    - 应用程序的Bundle ID是其唯一标识符，类似于Web域，但是倒置的（例如，net.stevekinney.firesale）。
    - Mac App Store应用程序必须使用沙盒。必须在属性列表中定义所需的任何权利。
    - 必须为所有子进程以及主应用程序本身定义权利。
    - 开发人员可以通过向其属性列表添加条目来定义Electron应用程序应该能够打开的文件类型。
    - 可以使用Xcode附带的Application Loader将应用程序上传到iTunes Connect。
    - Apple要求网络上的屏幕截图，描述和帮助页面被批准用于Mac App Store。

# 附录

# 来自Fire Sale和Clipmaster 9000的代码示例

Fire Sale是价格降价上唯一一个稍微聪明的玩法 - 因为它毕竟是Markdown编辑器。Clipmaster 9000是Clipmaster应用程序的简单UI。

### *第6章末尾的代码*

**清单1 Fire Sale的主进程：./app/main.js**

const { app, BrowserWindow, dialog } = require('electron'); const fs = require('fs');

const windows = new Set(); const openFiles = new Map();

app.on('ready', () => { createWindow();

});

app.on('window-all-closed', () => { if (process.platform === 'darwin') {

return false;

}

});

app.on('activate', (event, hasVisibleWindows) => { if (!hasVisibleWindows) { createWindow(); }

});

const createWindow = exports.createWindow = () => { let x, y;

const currentWindow = BrowserWindow.getFocusedWindow();

**313**

if (currentWindow) {

const [ currentWindowX, currentWindowY ] = currentWindow.getPosition(); x = currentWindowX + 10;

y = currentWindowY + 10;

}

let newWindow = new BrowserWindow({ x, y, show: false }); newWindow.loadURL(`file://${ dirname}/index.html`); newWindow.once('ready-to-show', () => {

newWindow.show();

});

newWindow.on('close', (event) => {

if (newWindow.isDocumentEdited()) { event.preventDefault();

const result = dialog.showMessageBox(newWindow, { type: 'warning',

title: 'Quit with Unsaved Changes?',

message: 'Your changes will be lost permanently if you do not save.', buttons: [

'Quit Anyway', 'Cancel',

],

cancelId: 1,

defaultId: 0

});

if (result === 0) newWindow.destroy();

}

});

newWindow.on('closed', () => { windows.delete(newWindow); stopWatchingFile(newWindow); newWindow = null;

});

windows.add(newWindow); return newWindow;

};

const getFileFromUser = exports.getFileFromUser = (targetWindow) => { const files = dialog.showOpenDialog(targetWindow, {

properties: ['openFile'], filters: [

{ name: 'Text Files', extensions: ['txt'] },

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

if (files) { openFile(targetWindow, files[0]); }

};

const openFile = exports.openFile = (targetWindow, file) => { const content = fs.readFileSync(file).toString(); app.addRecentDocument(file); targetWindow.setRepresentedFilename(file); targetWindow.webContents.send('file-opened', file, content); startWatchingFile(targetWindow, file);

};

const saveMarkdown = exports.saveMarkdown = (targetWindow, file, content) =>

{

if (!file) {

file = dialog.showSaveDialog(targetWindow, { title: 'Save Markdown',

defaultPath: app.getPath('documents'), filters: [

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

}

if (!file) return;

fs.writeFileSync(file, content); openFile(targetWindow, file);

};

const saveHtml = exports.saveHtml = (targetWindow, content) => { const file = dialog.showSaveDialog(targetWindow, {

title: 'Save HTML',

defaultPath: app.getPath('documents'), filters: [

{ name: 'HTML Files', extensions: ['html', 'htm'] }

]

});

if (!file) return;

fs.writeFileSync(file, content);

};

const startWatchingFile = (targetWindow, file) => { stopWatchingFile(targetWindow);

const watcher = fs.watchFile(file, () => { const content = fs.readFileSync(file);

targetWindow.webContents.send('file-changed', file, content);

});

openFiles.set(targetWindow, watcher);

};

const stopWatchingFile = (targetWindow) => { if (openFiles.has(targetWindow)) {

openFiles.get(targetWindow).stop();

openFiles.delete(targetWindow);

}

};

**清单2 Fire Sale的渲染器进程：./app /renderer.js**

const { remote, ipcRenderer } = require('electron'); const path = require('path');

const mainProcess = remote.require('./main.js'); const currentWindow = remote.getCurrentWindow();

const marked = require('marked');

const markdownView = document.querySelector('#markdown'); const htmlView = document.querySelector('#html');

const newFileButton = document.querySelector('#new-file'); const openFileButton = document.querySelector('#open-file');

const saveMarkdownButton = document.querySelector('#save-markdown'); const revertButton = document.querySelector('#revert');

const saveHtmlButton = document.querySelector('#save-html'); const showFileButton = document.querySelector('#show-file');

const openInDefaultButton = document.querySelector('#open-in-default');

let filePath = null;

let originalContent = '';

const isDifferentContent = (content) => content !== markdownView.value; const renderMarkdownToHtml = (markdown) => {

htmlView.innerHTML = marked(markdown, { sanitize: true });

};

const renderFile = (file, content) => { filePath = file;

originalContent = content;

markdownView.value = content; renderMarkdownToHtml(content);

updateUserInterface(false);

};

const updateUserInterface = (isEdited) => { let title = 'Fire Sale';

if (filePath) { title = `${path.basename(filePath)} - ${title}`; } if (isEdited) { title = `${title} (Edited)`; }

currentWindow.setTitle(title); currentWindow.setDocumentEdited(isEdited);

saveMarkdownButton.disabled = !isEdited; revertButton.disabled = !isEdited;

};

markdownView.addEventListener('keyup', (event) => { const currentContent = event.target.value; renderMarkdownToHtml(currentContent);

updateUserInterface(currentContent !== originalContent);

});

newFileButton.addEventListener('click', () => { mainProcess.createWindow();

});

openFileButton.addEventListener('click', () => { mainProcess.getFileFromUser(currentWindow);

});

saveMarkdownButton.addEventListener('click', () => { mainProcess.saveMarkdown(currentWindow, filePath, markdownView.value);

});

revertButton.addEventListener('click', () => { markdownView.value = originalContent; renderMarkdownToHtml(originalContent);

});

saveHtmlButton.addEventListener('click', () => { mainProcess.saveHtml(currentWindow, htmlView.innerHTML);

});

ipcRenderer.on('file-opened', (event, file, content) => {

if (currentWindow.isDocumentEdited() && isDifferentContent(content)) { const result = remote.dialog.showMessageBox(currentWindow, {

type: 'warning',

title: 'Overwrite Current Unsaved Changes?',

message: 'Opening a new file in this window will overwrite your unsaved changes. Open this file anyway?',

buttons: [ 'Yes',

'Cancel',

],

defaultId: 0,

cancelId: 1,

});

if (result === 1) { return; }

}

renderFile(file, content);

});

ipcRenderer.on('file-changed', (event, file, content) => { if (isDifferentContent(content)) return;

const result = remote.dialog.showMessageBox(currentWindow, { type: 'warning',

title: 'Overwrite Current Unsaved Changes?',

message: 'Another application has changed this file. Load changes?', buttons: [

'Yes',

'Cancel',

],

defaultId: 0,

cancelId: 1

});

renderFile(file, content);

});

/\* Implement Drag and Drop \*/

document.addEventListener('dragstart', event => event.preventDefault()); document.addEventListener('dragover', event => event.preventDefault()); document.addEventListener('dragleave', event => event.preventDefault()); document.addEventListener('drop', event => event.preventDefault());

const getDraggedFile = (event) => event.dataTransfer.items[0]; const getDroppedFile = (event) => event.dataTransfer.files[0];

const fileTypeIsSupported = (file) => {

return ['text/plain', 'text/markdown'].includes(file.type);

};

markdownView.addEventListener('dragover', (event) => { const file = getDraggedFile(event);

if (fileTypeIsSupported(file)) { markdownView.classList.add('drag-over');

} else {

markdownView.classList.add('drag-error');

}

});

markdownView.addEventListener('dragleave', () => { markdownView.classList.remove('drag-over'); markdownView.classList.remove('drag-error');

});

markdownView.addEventListener('drop', (event) => { const file = getDroppedFile(event);

if (fileTypeIsSupported(file)) { mainProcess.openFile(currentWindow, file.path);

} else {

alert('That file type is not supported');

}

markdownView.classList.remove('drag-over'); markdownView.classList.remove('drag-error');

});

### *第7章末尾的代码*

**清单3 Fire Sale的应用程序菜单：./app/application-menu.js**

const { app, dialog, Menu } = require('electron'); const mainProcess = require('./main');

const template = [

{

label: 'File', submenu: [

{

label: 'New File',

accelerator: 'CommandOrControl+N', click() {

mainProcess.createWindow();

}

},

{

label: 'Open File',

accelerator: 'CommandOrControl+O', click(item, focusedWindow) {

if (focusedWindow) {

return mainProcess.getFileFromUser(focusedWindow);

}

const newWindow = mainProcess.createWindow();

newWindow.on('show', () => { mainProcess.getFileFromUser(newWindow);

});

},

},

{

label: 'Save File',

accelerator: 'CommandOrControl+S', click(item, focusedWindow) {

if (!focusedWindow) {

return dialog.showErrorBox( 'Cannot Save or Export',

'There is currently no active document to save or export.'

);

}

mainProcess.saveMarkdown(focusedWindow);

},

},

{

label: 'Export HTML',

accelerator: 'Shift+CommandOrControl+S', click(item, focusedWindow) {

if (!focusedWindow) {

return dialog.showErrorBox( 'Cannot Save or Export',

'There is currently no active document to save or export.'

);

}

mainProcess.saveHtml(focusedWindow);

},

},

],

},

{

label: 'Edit', submenu: [

{

label: 'Undo',

accelerator: 'CommandOrControl+Z', role: 'undo',

},

{

label: 'Redo',

accelerator: 'Shift+CommandOrControl+Z', role: 'redo',

},

{ type: 'separator' },

{

label: 'Cut',

accelerator: 'CommandOrControl+X', role: 'cut',

},

{

label: 'Copy',

accelerator: 'CommandOrControl+C', role: 'copy',

},

{

label: 'Paste',

accelerator: 'CommandOrControl+V', role: 'paste',

},

{

label: 'Select All',

accelerator: 'CommandOrControl+A', role: 'selectall',

},

],

},

{

label: 'Window', submenu: [

{

label: 'Minimize',

accelerator: 'CommandOrControl+M', role: 'minimize',

},

{

label: 'Close',

accelerator: 'CommandOrControl+W', role: 'close',

},

],

},

{

label: 'Help',

role: 'help', submenu: [

{

label: 'Visit Website',

click() { /\* To be implemented \*/ }

},

{

label: 'Toggle Developer Tools', click(item, focusedWindow) {

if (focusedWindow) focusedWindow.webContents.toggleDevTools();

}

}

],

}

];

if (process.platform === 'darwin') { const name = 'Fire Sale'; template.unshift({

label: name, submenu: [

{

label: `About ${name}`, role: 'about',

},

{ type: 'separator' },

{

label: 'Services', role: 'services', submenu: [],

},

{ type: 'separator' },

{

label: `Hide ${name}`, accelerator: 'Command+H', role: 'hide',

},

{

label: 'Hide Others', accelerator: 'Command+Alt+H', role: 'hideothers',

},

{

label: 'Show All', role: 'unhide',

},

{ type: 'separator' },

{

label: `Quit ${name}`, accelerator: 'Command+Q', click() { app.quit(); },

},

],

});

const windowMenu = template.find(item => item.label === 'Window'); windowMenu.submenu.push(

{ type: 'separator' },

{

label: 'Bring All to Front', role: 'front',

}

);

}

module.exports = Menu.buildFromTemplate(template);

**清单4 Fire Sale的渲染进程：./app/renderer.js**

const { remote, ipcRenderer } = require('electron'); const { Menu } = remote;

const path = require('path');

const mainProcess = remote.require('./main.js'); const currentWindow = remote.getCurrentWindow();

const marked = require('marked');

const markdownView = document.querySelector('#markdown'); const htmlView = document.querySelector('#html');

const newFileButton = document.querySelector('#new-file'); const openFileButton = document.querySelector('#open-file');

const saveMarkdownButton = document.querySelector('#save-markdown'); const revertButton = document.querySelector('#revert');

const saveHtmlButton = document.querySelector('#save-html'); const showFileButton = document.querySelector('#show-file');

const openInDefaultButton = document.querySelector('#open-in-default');

let filePath = null;

let originalContent = '';

const isDifferentContent = (content) => content !== markdownView.value; const renderMarkdownToHtml = (markdown) => {

htmlView.innerHTML = marked(markdown, { sanitize: true });

};

const renderFile = (file, content) => { filePath = file;

originalContent = content;

markdownView.value = content; renderMarkdownToHtml(content);

updateUserInterface(false);

};

const updateUserInterface = (isEdited) => { let title = 'Fire Sale';

if (filePath) { title = `${path.basename(filePath)} - ${title}`; } if (isEdited) { title = `${title} (Edited)`; }

currentWindow.setTitle(title); currentWindow.setDocumentEdited(isEdited);

saveMarkdownButton.disabled = !isEdited; revertButton.disabled = !isEdited;

};

markdownView.addEventListener('keyup', (event) => { const currentContent = event.target.value; renderMarkdownToHtml(currentContent);

updateUserInterface(currentContent !== originalContent);

});

newFileButton.addEventListener('click', () => { mainProcess.createWindow();

});

openFileButton.addEventListener('click', () => { mainProcess.getFileFromUser(currentWindow);

});

saveMarkdownButton.addEventListener('click', () => { mainProcess.saveMarkdown(currentWindow, filePath, markdownView.value);

});

revertButton.addEventListener('click', () => { markdownView.value = originalContent; renderMarkdownToHtml(originalContent);

});

saveHtmlButton.addEventListener('click', () => { mainProcess.saveHtml(currentWindow, htmlView.innerHTML);

});

ipcRenderer.on('file-opened', (event, file, content) => {

if (currentWindow.isDocumentEdited() && isDifferentContent(content)) { const result = remote.dialog.showMessageBox(currentWindow, {

type: 'warning',

title: 'Overwrite Current Unsaved Changes?',

message: 'Opening a new file in this window will overwrite your unsaved changes. Open this file anyway?',

buttons: [ 'Yes',

'Cancel',

],

defaultId: 0,

cancelId: 1,

});

if (result === 1) { return; }

}

renderFile(file, content);

});

ipcRenderer.on('file-changed', (event, file, content) => { if (isDifferentContent(content)) return;

const result = remote.dialog.showMessageBox(currentWindow, { type: 'warning',

title: 'Overwrite Current Unsaved Changes?',

message: 'Another application has changed this file. Load changes?', buttons: [

'Yes',

'Cancel',

],

defaultId: 0,

cancelId: 1

});

renderFile(file, content);

});

/\* Implement Drag and Drop \*/

document.addEventListener('dragstart', event => event.preventDefault()); document.addEventListener('dragover', event => event.preventDefault()); document.addEventListener('dragleave', event => event.preventDefault()); document.addEventListener('drop', event => event.preventDefault());

const getDraggedFile = (event) => event.dataTransfer.items[0]; const getDroppedFile = (event) => event.dataTransfer.files[0];

const fileTypeIsSupported = (file) => {

return ['text/plain', 'text/markdown'].includes(file.type);

};

markdownView.addEventListener('dragover', (event) => { const file = getDraggedFile(event);

if (fileTypeIsSupported(file)) { markdownView.classList.add('drag-over');

} else {

markdownView.classList.add('drag-error');

}

});

markdownView.addEventListener('dragleave', () => { markdownView.classList.remove('drag-over'); markdownView.classList.remove('drag-error');

});

markdownView.addEventListener('drop', (event) => { const file = getDroppedFile(event);

if (fileTypeIsSupported(file)) { mainProcess.openFile(currentWindow, file.path);

} else {

alert('That file type is not supported');

}

markdownView.classList.remove('drag-over'); markdownView.classList.remove('drag-error');

});

ipcRenderer.on('save-markdown', () => { mainProcess.saveMarkdown(currentWindow, filePath, markdownView.value);

});

ipcRenderer.on('save-html', () => { mainProcess.saveHtml(currentWindow, htmlView.innerHTML);

});

const markdownContextMenu = Menu.buildFromTemplate([

{ label: 'Open File', click() { mainProcess.getFileFromUser(); } },

{ type: 'separator' },

{ label: 'Cut', role: 'cut' },

{ label: 'Copy', role: 'copy' },

{ label: 'Paste', role: 'paste' },

{ label: 'Select All', role: 'selectall' },

]);

markdownView.addEventListener('contextmenu', (event) => { event.preventDefault();

markdownContextMenu.popup();

});

### *第8章末尾的代码*

**清单5 Fire Sale的渲染器进程：./app/renderer.js**

const { remote, ipcRenderer, shell } = require('electron'); const { Menu } = remote;

const path = require('path');

const mainProcess = remote.require('./main.js'); const currentWindow = remote.getCurrentWindow();

const marked = require('marked');

const markdownView = document.querySelector('#markdown'); const htmlView = document.querySelector('#html');

const newFileButton = document.querySelector('#new-file'); const openFileButton = document.querySelector('#open-file');

const saveMarkdownButton = document.querySelector('#save-markdown'); const revertButton = document.querySelector('#revert');

const saveHtmlButton = document.querySelector('#save-html'); const showFileButton = document.querySelector('#show-file');

const openInDefaultButton = document.querySelector('#open-in-default');

let filePath = null;

let originalContent = '';

const isDifferentContent = (content) => content !== markdownView.value; const renderMarkdownToHtml = (markdown) => {

htmlView.innerHTML = marked(markdown, { sanitize: true });

};

const renderFile = (file, content) => { filePath = file;

originalContent = content;

markdownView.value = content; renderMarkdownToHtml(content);

showFileButton.disabled = false; openInDefaultButton.disabled = false;

updateUserInterface(false);

};

const updateUserInterface = (isEdited) => { let title = 'Fire Sale';

if (filePath) { title = `${path.basename(filePath)} - ${title}`; } if (isEdited) { title = `${title} (Edited)`; }

currentWindow.setTitle(title); currentWindow.setDocumentEdited(isEdited);

saveMarkdownButton.disabled = !isEdited; revertButton.disabled = !isEdited;

};

markdownView.addEventListener('keyup', (event) => { const currentContent = event.target.value; renderMarkdownToHtml(currentContent);

updateUserInterface(currentContent !== originalContent);

});

newFileButton.addEventListener('click', () => { mainProcess.createWindow();

});

openFileButton.addEventListener('click', () => { mainProcess.getFileFromUser(currentWindow);

});

saveMarkdownButton.addEventListener('click', () => { mainProcess.saveMarkdown(currentWindow, filePath, markdownView.value);

});

revertButton.addEventListener('click', () => { markdownView.value = originalContent;

renderMarkdownToHtml(originalContent);

});

saveHtmlButton.addEventListener('click', () => { mainProcess.saveHtml(currentWindow, htmlView.innerHTML);

});

const showFile = () => {

if (!filePath) { return alert('This file has not been saved to the file system.'); }

shell.showItemInFolder(filePath);

};

const openInDefaultApplication = () => {

if (!filePath) { return alert('This file has not been saved to the file system.'); }

shell.openItem(filePath);

};

showFileButton.addEventListener('click', showFile); openInDefaultButton.addEventListener('click', openInDefaultApplication); ipcRenderer.on('show-file', showFile);

ipcRenderer.on('open-in-default', openInDefaultApplication);

ipcRenderer.on('file-opened', (event, file, content) => {

if (currentWindow.isDocumentEdited() && isDifferentContent(content)) { const result = remote.dialog.showMessageBox(currentWindow, {

type: 'warning',

title: 'Overwrite Current Unsaved Changes?',

message: 'Opening a new file in this window will overwrite your unsaved changes. Open this file anyway?',

buttons: [ 'Yes',

'Cancel',

],

defaultId: 0,

cancelId: 1,

});

if (result === 1) { return; }

}

renderFile(file, content);

});

ipcRenderer.on('file-changed', (event, file, content) => { if (isDifferentContent(content)) return;

const result = remote.dialog.showMessageBox(currentWindow, { type: 'warning',

title: 'Overwrite Current Unsaved Changes?',

message: 'Another application has changed this file. Load changes?', buttons: [

'Yes',

'Cancel',

],

defaultId: 0,

cancelId: 1

});

renderFile(file, content);

});

/\* Implement Drag and Drop \*/

document.addEventListener('dragstart', event => event.preventDefault()); document.addEventListener('dragover', event => event.preventDefault()); document.addEventListener('dragleave', event => event.preventDefault()); document.addEventListener('drop', event => event.preventDefault());

const getDraggedFile = (event) => event.dataTransfer.items[0]; const getDroppedFile = (event) => event.dataTransfer.files[0];

const fileTypeIsSupported = (file) => {

return ['text/plain', 'text/markdown'].includes(file.type);

};

markdownView.addEventListener('dragover', (event) => { const file = getDraggedFile(event);

if (fileTypeIsSupported(file)) { markdownView.classList.add('drag-over');

} else {

markdownView.classList.add('drag-error');

}

});

markdownView.addEventListener('dragleave', () => { markdownView.classList.remove('drag-over'); markdownView.classList.remove('drag-error');

});

markdownView.addEventListener('drop', (event) => { const file = getDroppedFile(event);

if (fileTypeIsSupported(file)) { mainProcess.openFile(currentWindow, file.path);

} else {

alert('That file type is not supported');

}

markdownView.classList.remove('drag-over'); markdownView.classList.remove('drag-error');

});

const createContextMenu = () => { return Menu.buildFromTemplate([

{ label: 'Open File', click() { mainProcess.getFileFromUser(); } },

{

label: 'Show File in Folder', click: showFile,

enabled: !!filePath

},

{

label: 'Open in Default',

click: openInDefaultApplication, enabled: !!filePath

},

{ type: 'separator' },

{ label: 'Cut', role: 'cut' },

{ label: 'Copy', role: 'copy' },

{ label: 'Paste', role: 'paste' },

{ label: 'Select All', role: 'selectall' },

]);

};

markdownView.addEventListener('contextmenu', (event) => { event.preventDefault();

createContextMenu().popup();

});

ipcRenderer.on('save-markdown', () => { mainProcess.saveMarkdown(currentWindow, filePath, markdownView.value);

});

ipcRenderer.on('save-html', () => { mainProcess.saveHtml(currentWindow, filePath, markdownView.value);

});

**清单6 Fire Sale的应用程序菜单：./app/application-menu.js**

const { app, BrowserWindow, dialog, Menu, shell } = require('electron'); const mainProcess = require('./main');

const createApplicationMenu = () => {

const hasOneOrMoreWindows = !!BrowserWindow.getAllWindows().length; const focusedWindow = BrowserWindow.getFocusedWindow();

const hasFilePath = !!(focusedWindow && focusedWindow.getRepresentedFilename());

const template = [

{

label: 'File', submenu: [

{

label: 'New File',

accelerator: 'CommandOrControl+N', click() {

mainProcess.createWindow();

}

},

{

label: 'Open File',

accelerator: 'CommandOrControl+O', click(item, focusedWindow) {

if (focusedWindow) {

return mainProcess.getFileFromUser(focusedWindow);

}

const newWindow = mainProcess.createWindow();

newWindow.on('show', () => { mainProcess.getFileFromUser(newWindow);

});

},

},

{

label: 'Save File',

accelerator: 'CommandOrControl+S', enabled: hasOneOrMoreWindows, click(item, focusedWindow) {

if (!focusedWindow) {

return dialog.showErrorBox( 'Cannot Save or Export',

'There is currently no active document to save or export.'

);

}

mainProcess.saveMarkdown(focusedWindow);

},

},

{

label: 'Export HTML',

accelerator: 'Shift+CommandOrControl+S', enabled: hasOneOrMoreWindows, click(item, focusedWindow) {

if (!focusedWindow) {

return dialog.showErrorBox( 'Cannot Save or Export',

'There is currently no active document to save or export.'

);

}

mainProcess.saveHtml(focusedWindow);

},

},

{ type: 'separator' },

{

label: 'Show File', enabled: hasFilePath,

click(item, focusedWindow) { if (!focusedWindow) {

return dialog.showErrorBox( 'Cannot Show File\'s Location',

'There is currently no active document show.'

);

}

focusedWindow.webContents.send('show-file');

},

},

{

label: 'Open in Default Application', enabled: hasFilePath,

click(item, focusedWindow) { if (!focusedWindow) {

return dialog.showErrorBox(

'Cannot Open File in Default Application', 'There is currently no active document to open.'

);

}

focusedWindow.webContents.send('open-in-default');

},

},

],

},

{

label: 'Edit', submenu: [

{

label: 'Undo',

accelerator: 'CommandOrControl+Z', role: 'undo',

},

{

label: 'Redo',

accelerator: 'Shift+CommandOrControl+Z', role: 'redo',

},

{ type: 'separator' },

{

label: 'Cut',

accelerator: 'CommandOrControl+X', role: 'cut',

},

{

label: 'Copy',

accelerator: 'CommandOrControl+C', role: 'copy',

},

{

label: 'Paste',

accelerator: 'CommandOrControl+V', role: 'paste',

},

{

label: 'Select All',

accelerator: 'CommandOrControl+A', role: 'selectall',

},

],

},

{

label: 'Window', submenu: [

{

label: 'Minimize',

accelerator: 'CommandOrControl+M', role: 'minimize',

},

{

label: 'Close',

accelerator: 'CommandOrControl+W', role: 'close',

},

],

},

{

label: 'Help',

role: 'help', submenu: [

{

label: 'Visit Website',

click() { /\* To be implemented \*/ }

},

{

label: 'Toggle Developer Tools', click(item, focusedWindow) {

if (focusedWindow) focusedWindow.webContents.toggleDevTools();

}

}

],

}

];

if (process.platform === 'darwin') { const name = 'Fire Sale'; template.unshift({

label: name, submenu: [

{

label: `About ${name}`, role: 'about',

},

{ type: 'separator' },

{

label: 'Services', role: 'services', submenu: [],

},

{ type: 'separator' },

{

label: `Hide ${name}`, accelerator: 'Command+H', role: 'hide',

},

{

label: 'Hide Others', accelerator: 'Command+Alt+H', role: 'hideothers',

},

{

label: 'Show All', role: 'unhide',

},

{ type: 'separator' },

{

],

});

label: `Quit ${name}`, accelerator: 'Command+Q', click() { app.quit(); },

},

const windowMenu = template.find(item => item.label === 'Window'); windowMenu.submenu.push(

{ type: 'separator' },

{

label: 'Bring All to Front', role: 'front',

}

);

}

return Menu.setApplicationMenu(Menu.buildFromTemplate(template));

};

module.exports = createApplicationMenu;

**清单7 Fire Sale的主进程：./app/main.js**

const { app, BrowserWindow, dialog, Menu } = require('electron'); const createApplicationMenu = require('./application-menu'); const fs = require('fs');

const windows = new Set(); const openFiles = new Map();

app.on('ready', () => { createApplicationMenu(); createWindow();

});

app.on('window-all-closed', () => { if (process.platform === 'darwin') {

return false;

}

});

app.on('activate', (event, hasVisibleWindows) => { if (!hasVisibleWindows) { createWindow(); }

});

const createWindow = exports.createWindow = () => { let x, y;

const currentWindow = BrowserWindow.getFocusedWindow();

if (currentWindow) {

const [ currentWindowX, currentWindowY ] = currentWindow.getPosition(); x = currentWindowX + 10;

y = currentWindowY + 10;

}

let newWindow = new BrowserWindow({ x, y, show: false }); newWindow.loadURL(`file://${ dirname}/index.html`); newWindow.once('ready-to-show', () => {

newWindow.show();

});

newWindow.on('focus', createApplicationMenu); newWindow.on('close', (event) => {

if (newWindow.isDocumentEdited()) {

event.preventDefault();

const result = dialog.showMessageBox(newWindow, { type: 'warning',

title: 'Quit with Unsaved Changes?',

message: 'Your changes will be lost permanently if you do not save.', buttons: [

'Quit Anyway', 'Cancel',

],

cancelId: 1,

defaultId: 0

});

if (result === 0) newWindow.destroy();

}

});

newWindow.on('closed', () => { windows.delete(newWindow); createApplicationMenu(); newWindow = null;

});

windows.add(newWindow); return newWindow;

};

const getFileFromUser = exports.getFileFromUser = (targetWindow) => { const files = dialog.showOpenDialog(targetWindow, {

properties: ['openFile'], filters: [

{ name: 'Text Files', extensions: ['txt'] },

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

if (files) { openFile(targetWindow, files[0]); }

};

const openFile = exports.openFile = (targetWindow, file) => { const content = fs.readFileSync(file).toString(); startWatchingFile(targetWindow, file); app.addRecentDocument(file); targetWindow.setRepresentedFilename(file); targetWindow.webContents.send('file-opened', file, content); createApplicationMenu();

};

const saveMarkdown = exports.saveMarkdown = (targetWindow, file, content) =>

{

if (!file) {

file = dialog.showSaveDialog(targetWindow, { title: 'Save Markdown',

defaultPath: app.getPath('documents'), filters: [

{ name: 'Markdown Files', extensions: ['md', 'markdown'] }

]

});

}

if (!file) return;

fs.writeFileSync(file, content); openFile(targetWindow, file);

};

const saveHtml = exports.saveHtml = (targetWindow, content) => { const file = dialog.showSaveDialog(targetWindow, {

title: 'Save HTML',

defaultPath: app.getPath('documents'), filters: [

{ name: 'HTML Files', extensions: ['html', 'htm'] }

]

});

if (!file) return;

fs.writeFileSync(file, content);

};

const startWatchingFile = (targetWindow, file) => { stopWatchingFile(targetWindow);

const watcher = fs.watchFile(file, () => { const content = fs.readFileSync(file);

targetWindow.webContents.send('file-changed', file, content);

});

openFiles.set(targetWindow, watcher);

};

const stopWatchingFile = (targetWindow) => { if (openFiles.has(targetWindow)) {

openFiles.get(targetWindow).stop(); openFiles.delete(targetWindow);

}

};

### *从第10章末尾完成的代码*

**清单8 Clipmaster 9000的主进程：./app/main.js**

const Menubar = require('menubar');

const { globalShortcut, Menu } = require('electron');

const menubar = Menubar({ preloadWindow: true,

index: `file://${ dirname}/index.html`,

});

menubar.on('ready', () => {

const secondaryMenu = Menu.buildFromTemplate([

{

label: 'Quit',

click() { menubar.app.quit(); }, accelerator: 'CommandOrControl+Q'

},

]);

menubar.tray.on('right-click', () => { menubar.tray.popUpContextMenu(secondaryMenu);

});

const createClipping = globalShortcut.register('CommandOrControl+!', () =>

{

menubar.window.webContents.send('create-new-clipping');

});

const writeClipping = globalShortcut.register('CmdOrCtrl+Alt+@', () => { menubar.window.webContents.send('write-to-clipboard');

});

const publishClipping = globalShortcut.register('CmdOrCtrl+Alt+#', () => { menubar.window.webContents.send('publish-clipping');

});

if (!createClipping) { console.error('Registration failed', 'createClipping'); }

if (!writeClipping) { console.error('Registration failed', 'writeClipping'); }

if (!publishClipping) { console.error('Registration failed', 'publishClipping'); }

});

**清单9 Clipmaster 9000的渲染器进程：./app/renderer.js**

const { clipboard, ipcRenderer, shell } = require('electron');

const request = require('request').defaults({ url: 'https://cliphub.glitch.me/clippings', headers: { 'User-Agent': 'Clipmaster 9000' }, json: true,

});

const clippingsList = document.getElementById('clippings-list'); const copyFromClipboardButton = document.getElementById('copy-from-

clipboard');

ipcRenderer.on('create-new-clipping', () => { addClippingToList();

new Notification('Clipping Added', { body: `${clipboard.readText()}`

});

});

ipcRenderer.on('write-to-clipboard', () => { const clipping = clippingsList.firstChild; writeToClipboard(getClippingText(clipping)); new Notification('Clipping Copied', {

body: `${clipboard.readText()}`

});

});

ipcRenderer.on('publish-clipping', () => { const clipping = clippingsList.firstChild; publishClipping(getClippingText(clipping));

});

const createClippingElement = (clippingText) => {

const clippingElement = document.createElement('article');

clippingElement.classList.add('clippings-list-item'); clippingElement.innerHTML = `

<div class="clipping-text" disabled="true"></div>

<div class="clipping-controls">

<button class="copy-clipping">&rarr; Clipboard</button>

<button class="publish-clipping">Publish</button>

<button class="remove-clipping">Remove</button>

</div>

`;

clippingElement.querySelector('.clipping-text').innerText = clippingText; return clippingElement;

};

const addClippingToList = () => {

const clippingText = clipboard.readText();

const clippingElement = createClippingElement(clippingText); clippingsList.prepend(clippingElement);

};

copyFromClipboardButton.addEventListener('click', addClippingToList); clippingsList.addEventListener('click', (event) => {

const hasClass = className => event.target.classList.contains(className); const clippingListItem = getButtonParent(event);

if (hasClass('remove-clipping')) removeClipping(clippingListItem); if (hasClass('copy-clipping'))

writeToClipboard(getClippingText(clippingListItem)); if (hasClass('publish-clipping'))

publishClipping(getClippingText(clippingListItem));

});

const removeClipping = (target) => { target.remove();

};

const writeToClipboard = (clippingText) => { clipboard.writeText(clippingText);

};

const publishClipping = (clippingText) => {

request.post({ json: { clipping: clippingText } }, (err, response, body) =>

{

if (err) {

return new Notification('Error Publishing Your Clipping', { body: JSON.parse(err).message

});

}

const gistUrl = body.url;

const notification = new Notification('Your Clipping Has Been Published',

{

body: `Click to open ${gistUrl} in your browser.`

});

notification.onclick = () => { shell.openExternal(gistUrl); }; clipboard.writeText(gistUrl);

});

};

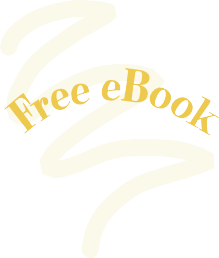
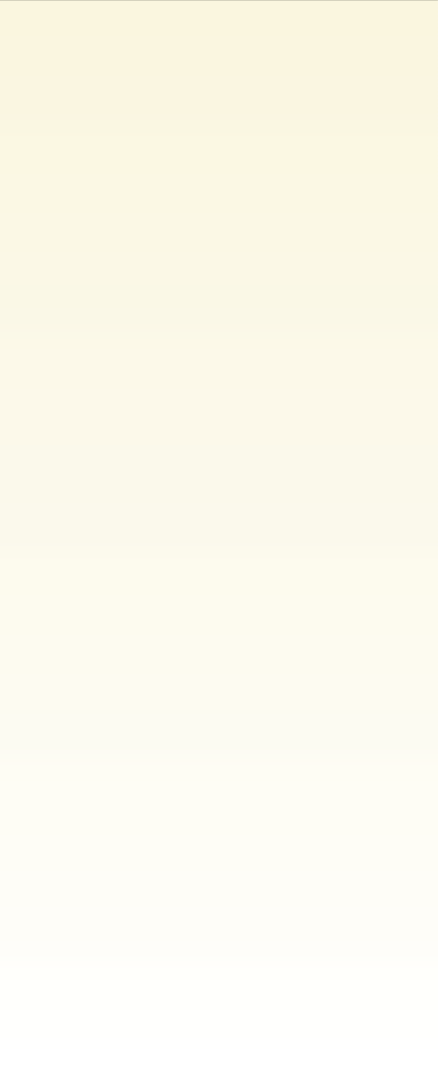
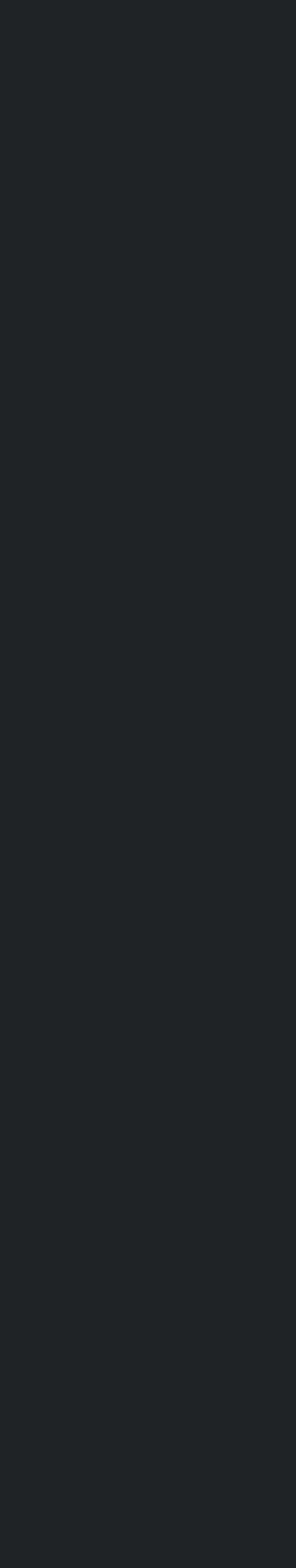
const getButtonParent = ({ target }) => { return target.parentNode.parentNode;

};

const getClippingText = (clippingListItem) => {

return clippingListItem.querySelector('.clipping-text').innerText;

};



PROGRAMMING/WEB DEVELOPMENT

Electron实战

Steve Kinney

使用web开发技能构建桌面应用程序不是很好吗？ Electron是一个专为此而设计的框架！ 完全跨平台，Electron允许您使用JavaScript和Node创建简单，快捷的桌面应用程序。使用Electron实现工具，游戏和实用工具的快速，实用和有趣！

Electron实战 教您使用JavaScript，Node和Electron框架构建跨平台应用程序。当您构建一个读取和呈现Markdown的文本工具时，您将学习如何像桌面开发人员一样思考。您将添加特定于操作系统的功能，如文件系统，菜单和剪贴板，并使用Chromium的工具来分发成品。您甚至可以通过数据存储，性能优化和测试来完善您的学习。

涵盖什么

* 构建macOS，Windows和Linux
* 本机操作系统API
* 使用React等第三方框架
* 部署到Mac App Store

需要中级JavaScript和Node技能。无需构建桌面应用程序的经验。

Steve Kinney 是SendGrid的首席工程师，Frontend Masters的讲师，以及科罗拉多州丹佛市DinosaurJS会议的组织者。

要下载PDF，ePub和Kindle格式的免费Electron书，本书的所有者应访问manning.com/books/electron-in-action

See first page

“一个代码驱动叙述的跨平台桌面应用程序开发的权威源代码。”

## —Ashwin K. Raj, Innocepts

“让您从简单了解Electron的含义，到实际编写复杂的Electron应用程序。”

—Alexey Galiullin, Voiceworks

“允许我快速构建自己的日常工具。”

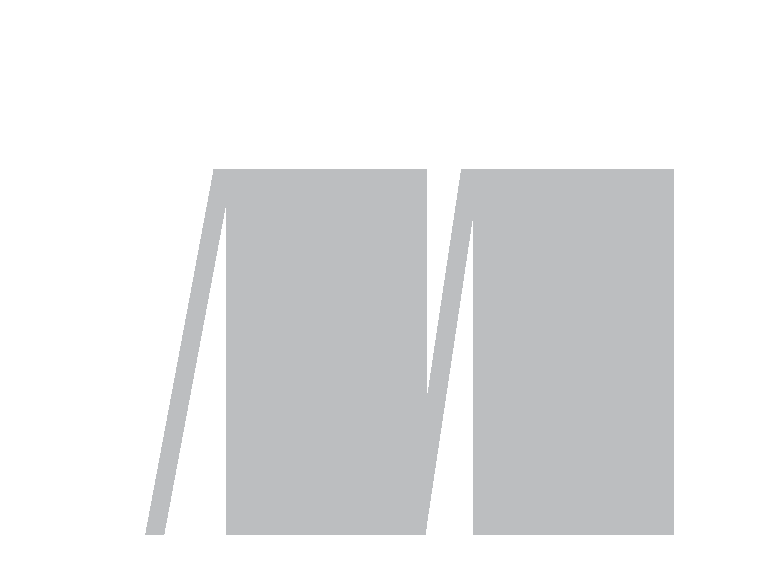
—Philippe Charrière, GitLab

“快速阅读和易于理解。”

—Jay Kelkar, Kelkar Systems

“最后，JavaScript无处不在！”

—William E. Wheeler, consultant

**M A N N I N G**

$44.99/Can $59.99 [包含eBOOK]