声　明

此书电子版免费供大家下载阅读，如果您已为此副本付费，请立即申请退款并联系作者举报此行为。请注意，虽然此书电子版免费供大家阅读，但这并不代表作者放弃了版权，您在未经授权的情况下依然不得以任何方式复制或抄袭本书内容。此书的电子版目前仅授权图灵社区和百度阅读两个平台发布，如果您通过其他渠道获取到了此副本，则是侵权行为，请到上述两个平台下载合法授权的副本。获取合法授权副本的好处是可以及时得到此书的最新版本，早期版本中的错误会被及时纠正。感谢您对版权保护工作所做出的贡献。

前　言

一个电子专业的在校学生，每天学习的是电子在晶格中如何游走，研究的是半导体器件的电学特性，无论如何都不会与这本书的作者联系在一起。

说起来写这本书非常偶然，在某一天我突然就想写点什么了，想写点很多人都会看的东西，作为人生中的一个成就。虽然我的专业是电子，但编程一直都是我最大的爱好，前端更是我最熟悉的领域。作为 Google 的追随者，我是第一批使用 Chrome 浏览器的用户，并在 Chrome 推出扩展功能后较早投入到其中的开发者之一，所以 Chrome 开发自然就成为了我写作的选题。

这本书诞生于图灵社区，图灵社区的写作氛围很好，而且在线编辑器支持 Markdown 语法。在开始我并不好意思直接说写的是书，直到定稿前夕我才把“文集”二字改成了“书”。回顾几个月之前，一拍脑门夹着笔记本就去图书馆开写了，在写作的过程中遇到一个又一个坑，有时为了让一个实例跑通要调上一整个下午，如果当初我知道会遇到这么多困难想来是不会动笔的。但既然动笔了，半途而废着实没有颜面，所以就一直坚持到了最后。

写作对个人能力的提升是非常大的，由于每一个知识点都必须咬死，不可含糊其辞，所以我在写作的过程中不得不一遍遍仔细翻阅 Chrome 官方开发文档和 W3C 标准，同时还要编写实例进行验证。

值得庆幸的是，这本书还没有完成就得到了很多读者的关注，他们给了我很大的鼓励，有的读者还表示可以无私帮我校审书稿。

在此我要重点感谢方觉，大家可能对这个名字并不熟悉，但他创建和维护的 crxdoc-zh.appspot.com 相信开发 Chrome 扩展和应用的开发者没有几个不知道，这本教程也参考和引用了上面的部分内容。他不仅仅对本书的语言表述进行了仔细认真的推敲，而且还纠正了一些知识点中的错误，包括官方文档中同样出现的错误，这让我感到十分惊讶，后来才发现他还是 Chromium 项目的贡献者，这更是令我敬佩不已。

吕鹏和李典是我很早就在互联网上结识的朋友。吕鹏同学和我一直在一起鼓捣些小程序，都说能找到志同道合的小伙伴不易，我十分庆幸能在广阔的互联网中与他结识。最初我认识吕鹏时他还是大二的在校生，转眼已是微软的大牛，我也为能有如此优秀的朋友感到自豪。在我刚刚开始写独立博客时，李典同学就成为了我的读者，虽然他一直躲在 Google Reader 后面。李典同学是第一个为此书提交勘误表的读者，在早先他为我的一个项目贡献代码时，

我就发现他是一个做事十分认真的人，这次也不例外，以至于后来我将他的勘误表作为样板发给了参与校审的每一位读者。

参与校审的还有赵余和韩骏，在此一并表示感谢！

另外不得不提的是，我在设计此书的封面时，使用了 Chan Cheong Pin 的摄影作品，海龟。他在得知我要将这幅作品用于此书的封面设计后，慷慨地授权我免费使用，在此也向他表达诚挚的感谢！

由于作者水平有限，书中不免出现错误，欢迎读者朋友指正。您可以通过 lizhe@lizhe.org 与作者联系，也可以通过图灵社区在线提交勘误信息，在此先行感谢。

第 1 章　初步接触 Chrome 扩展应用开发

Chrome 是 Google 公司基于 WebKit 开发的一款浏览器 1，但从某种角度上来说它已经超越了浏览器成为了一个平台甚至是一个操作系统。Chrome 继承了 WebKit 内核对 HTML 的高速渲染，同时 Google 自行开发的 V8 引擎使得 JavaScript 在 Chrome 中的执行效率大幅提升，这使得更加高级复杂的 JavaScript 程序在 Chrome 中运行成为可能。

1 Chrome 28 之后使用的 Blink 渲染引擎是 WebKit 中 WebCore 组件的一个分支。

Chrome 浏览器除了页面渲染速度快，JavaScript 执行速度快以外，另一大特点就是支持开发者为其编写各种各样的扩展来扩充其功能，用 HTML5 编写桌面程序，这使得 Chrome 变得更加强大。编写这样的程序就是本书所要讲解的内容。

本章首先对 Chrome 扩展应用进行简单概述，之后带着读者编写一个简单的扩展，使读者对扩展的认识更加深入。在讲解扩展 Manifest 文件格式时，也会简单讲解一下 JSON 数据格式 2，避免没有接触过 JSON 的读者阅读后续的内容产生困难。另外本章也用一小节简单讲解了一下 DOM，这对从未接触过网页编程的读者会非常有帮助。

2 JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。

1.1　认识 Chrome 扩展及应用

Chrome 扩展是用于扩充 Chrome 浏览器功能的程序，Chrome 应用是以 Chrome 为平台运行的程序，两者似乎并没有太明确的区别，甚至有些程序既可以设计成 Chrome 扩展也可以设计成 Chrome 应用。但既然 Google 将基于 Chrome 平台的程序分为了两类，说明两者还是有区别的。

Chrome 扩展主要用于对浏览器功能的增强，它更强调与浏览器相结合。比如 Chrome 扩展可以在浏览器的工具栏和地址栏中显示图标，它可以更改用户当前浏览的网页中的内容，也可以更改浏览器代理服务器的设置等等。

Chrome 应用更强调是独立的程序，你可以不打开 Chrome 浏览器而运行这些程序。同时这些程序可以调用更加底层的系统接口，比如串口、USB、本地文件读写等等。同时 Chrome 应用可以拥有样式更加自由的独立窗口，而 Chrome 扩展的界面只能限定在浏览器窗口中。

由于 Chrome 扩展和 Chrome 应用有很多相似之处，为了叙述方便本章会统一说成 Chrome 扩展，但应该清楚同样适用于 Chrome 应用。

Chrome 扩展是一系列文件的集合，这些文件包括 HTML 文件、CSS 样式文件、JavaScript 脚本文件、图片等静态文件以及 manifest.json。个别扩展还会包含二进制文件，如 DLL 动态库和 so 动态库等，但这需要调用 NPAPI，而 Google 出于安全性考虑已经决定逐渐淘汰 NPAPI，所以我不准备在本书中向大家介绍有关 NPAPI 的内容。

扩展被安装后，Chrome 就会读取扩展中的 manifest.json 文件。这个文件的文件名固定为 manifest.json，内容是按照一定格式描述的扩展相关信息，如扩展名称、版本、更新地址、请求的权限、扩展的 UI 界面入口等等。这样 Chrome 就可以知道在浏览器中如何呈现这个扩展，以及这个扩展如何同用户进行交互。

由于 Chrome 扩展是基于 Chrome 平台的，说得直白些，是基于 WebKit 浏览器的——当然有些更加高级的接口不仅仅依赖于 WebKit 浏览器——所以 Chrome 扩展在处理逻辑运算和实现程序功能时所采用的编程语言必然只能是 JavaScript。

可能你会感到惊讶，毕竟 JavaScript 最开始是为提升网站与用户交互体验而设计出的一种轻量级脚本语言，怎么会脱离网站而成为一种程序的逻辑语言呢？随着 Chrome 浏览器 V8 引擎的出现，JavaScript 的执行效率得到了大幅提升，甚至出现了将其作为后端语言的项目——Node.js。所以将 JavaScript 作为一种客户端程序语言就更是绰绰有余了——只要提供更加丰富的功能函数——而 Chrome 平台正提供了这样的环境。

总的来说，Chrome 扩展更像是一个运行于本地的网站，只是它可以利用 Chrome 平台提供的丰富的接口，获得更加全面的信息，进行更加复杂的操作。而它的界面则使用 HTML 和 CSS 进行描述，这样的好处是可以用很短的时间构建出赏心悦目的 UI。界面的渲染完全不需要开发者操心，而是交给 Chrome 去做。同时由于 JavaScript 是一门解释语言 1，无需对其配置编译器，调试代码时你只要刷新一下浏览器就可以看到修改后的结果，这使得开发周期大大缩短。

1 现代浏览器使用的 JavaScript 引擎会对 JavaScript 编译，V8。

同时 Chrome 浏览器相比于 Java 虚拟机、Python 解释器（Linux 和 OS X 中默认安装了 Python，而 Windows 中默认没有安装）等其他语言环境更加普及——我甚至可以在我们学校的图书馆计算机中找到 Chrome 浏览器——所以你所开发的 Chrome 扩展就可以在更多的计算机中运行。当你眼前遇到一个问题需要利用计算机去处理，而这台计算机恰好安装了 Chrome 浏览器，那么你就可以欢快地打开记事本开始编写程序了，之后加载到 Chrome 浏览器中就可以查看运行结果，这是一件多么酷的事啊！

别急，后面的内容就会让你得到这项新技能！

1.2 我的第一个 Chrome 扩展

我发现很多讲解编程的书籍，在前面都会详细地讲解相关的预备知识，而大多数读者却更希望马上进行实践。没错，人们总是对基础知识很排斥，这也就是为什么在教育行业开始推崇自顶向下的教材设计方案了——先让读者看到一个最接近表面的东西，之后再慢慢深入地讲解内在的原理和基础。所以我决定在还没有讲什么的时候，先带大家写一个 Demo 程序。这样不仅可以让大家在实践中对基础知识掌握得更加牢靠，同时也调动了大家的积极性。

Chrome 扩展的启动入口可以在浏览器的工具栏和地址栏中，用户单击后激活扩展进行下一步的操作，也可以干脆没有图标，在后台静默地运行。比如微博的扩展，可以设计成将图标显示在工具栏中，用户点击后打开一个显示用户微博时间轴的界面；RSS 订阅器扩展可以设计成将图标显示在地址栏中，当用户点击后，订阅地址栏中当前显示的 URL；自动使用

Google SSL 链接的扩展可以不显示图标，只是在后台默默地监视，当用户访问了非 SSL 的 Google 链接后，自动跳转到 SSL 链接即可。

Chrome 扩展图标在浏览器中的位置

我们准备编写一款显示用户计算机当前时间的扩展，这应该比 Hello World 有趣得多。设计思路是在浏览器的工具栏中显示一个时钟的图标，当用户点击这个图标时显示一个实时显示计算机时间的界面。

首先新建一个名为 my_clock 的文件夹，在此文件夹中新建一个名为 manifest.json 的文件，内容如下：

```
{
    "manifest_version": 2,
    "name": "我的时钟",
    "version": "1.0",
    "description": "我的第一个 Chrome 扩展",
    "icons": {
        "16": "images/icon16.png",
        "48": "images/icon48.png",
        "128": "images/icon128.png"
    },
    "browser_action": {
        "default_icon": {
            "19": "images/icon19.png",
            "38": "images/icon38.png"
        },
        "default_title": "我的时钟",
        "default_popup": "popup.html"
    }
}
```

上面的字段有些我们可以一眼看出在定义什么，比如 name 定义了扩展的名称，version 定义了扩展的版本，description 定义了扩展的描述，icons 定义了扩展相关图标文件的位置。version 的值最多可以是由三个圆点分为四段的版本号，每段只能是数字，每段数字不能大于 65535 且不能以 0 开头（可以是 0，但不可以是 0123），版本号段左侧为高位，比如 1.0.2.0 版本比 1.0.0.1 版本更高。每次更新扩展时，新的版本号必须比之前的版本号高。

browser_action 指定扩展的图标放在 Chrome 的工具栏中，browser_action 中的 default_icon 属性定义了相应图标文件的位置，default_title 定义了当用户鼠标悬停于扩展图标上所显示的文字，default_popup 则定义了当用户单击扩展图标时所显示页面的文件位置。

接下来我们开始编写 popup.html。

```
<html>
<head>
<style>
* {
    margin: 0;
    padding: 0;
```

```
}

body {
    width: 200px;
    height: 100px;
}

div {
    line-height: 100px;
    font-size: 42px;
    text-align: center;
}
</style>
</head>
<body>
<div id="clock_div"></div>
<script src="js/my_clock.js"></script>
</body>
</html>
```

如果你曾经编写过网页，会发现上面这个页面省略了很多内容，比如<title>标签。因为对于 Chrome 扩展来说，很多对网页有意义的内容是无意义的，所以我们可以只挑需要的写，当然你全写出来也不会有什么问题。

上面的这个页面首先定义了全局元素的 margin 和 padding 为 0，这样我们可以更加自由地控制元素的外观。在编写网页时，body 的尺寸往往不会专门给定，但对于 Chrome 扩展有时这是必要的，比如此例中我们需要告诉 Chrome 当用户单击扩展图标后展示一个多大的界面。

之后我们在 body 标签中定义了一个 id 为 clock_div 的 div 容器，用这个容器来显示当前的时间，这样我们就把 HTML 的布局写好了。接下来我们就需要引入 JavaScript 处理数据并动态显示了。值得注意的是 Chrome 不允许将 JavaScript 代码段直接内嵌入 HTML 文档，所以我们需要通过外部引入的方式引用 JS 文件。当然 inline-script 也是被禁止的，所以所有元素的事件都需要使用 JavaScript 代码进行绑定，如果你没有使用一个拥有强大选择器的库（如 jQuery），最好给需要绑定事件的元素分配一个 id 以便进行操作。

下面来编写 my_clock.js 文件。

```
function my_clock(el){
    var today=new Date();
    var h=today.getHours();
    var m=today.getMinutes();
    var s=today.getSeconds();
    m=m>=10?m:('0'+m);
    s=s>=10?s:('0'+s);
    el.innerHTML = h+":"+m+":"+s;
    setTimeout(function(){my_clock(el)}, 1000);
}
```

```
var clock_div = document.getElementById('clock_div');
my_clock(clock_div);
```

在 my_clock.js 文件中我们定义了一个 my_clock 函数用于显示时间，这个函数包含了一个 el 参数，这个参数为显示时间的容器，由于在 HTML 文档中我们设计在 id 为 clock_div 的 div 容器中显示时间，所以调用 my_clock 函数时我们传入了这个容器，在 js 文件中用变量 clock_div 表示。之后 my_clock 函数 1000 毫秒之后又会再次调用自身，这样 clock_div 中显示的时间就会被更新。

至此这个扩展就编写完毕了，当然别忘了将图标文件也放入相应的文件夹中。

扩展的文件结构

下面我们就需要将这个扩展载入 Chrome 中运行了。依次点击“”-“工具”-“扩展程序”打开扩展程序页面（也可以直接在地址栏中输入 chrome://extensions 进入），勾选右上角的“开发者模式”，点击“加载正在开发的扩展程序”，选择扩展所在的文件夹，就可以在浏览器工具栏中看到我们的扩展了。

将扩展载入到 Chrome 中

当鼠标点击扩展图标后，一个显示时钟的界面就出现了。

时钟扩展的运行界面

这个扩展的源代码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/my_clock 下载。

1.3 Manifest 文件格式

Chrome 扩展都包含一个 Manifest 文件——manifest.json，这个文件可以告诉 Chrome 关于这个扩展的相关信息，它是整个扩展的入口，也是 Chrome 扩展必不可少的部分。

Manifest 文件使用 JSON 格式保存数据，为了避免有的读者对 JSON 不了解而无法继续阅读，下面我将简单介绍一下 JSON。JSON 是 JavaScript Object Notation 的缩写，这是一种基于 JavaScript 语言的轻量级数据交换格式。由于 JSON 储存的数据冗余度比 XML 更低，而且便于读取，所以也被很多其他语言所支持，现在 JSON 已经成为一种跨平台跨语言的通用数据交换格式。

JSON 包含两种结构：一种是 key:value 对的形式，名称和值之间用冒号（:）连接，多个 key:value 对之间用逗号（,）连接，最后在整个对象两侧加上“{”和“}”；另一种是值的有序集合，值与值之间用逗号（,）连接，最后在整个数组两侧加上“[”和“]”。

对象形式的结构，图片来源于 www.json.org

数组形式的结构，图片来源于 www.json.org

其中无论是对象形式还是数组形式，它们的值均可以是字符串、数字、对象、数组、布

尔和 null 中的一种，也就是说 JSON 有嵌套的性质，值也可以是 JSON 格式的数据。

下面给出了一个 JSON 的例子：

```
{
    "name" : "Harry Potter",
    "author" : {
        "name" : "J.K.Rowling",
        "birth" : 1964
    },
    "books" : [
        "Harry Potter and the Philosopher's Stone",
        "Harry Potter and the Chamber of Secrets",
        "Harry Potter and the Prisoner of Azkaban",
        "Harry Potter and the Goblet of Fire",
        "Harry Potter and the Order of the Phoenix",
        "Harry Potter and the Half-Blood Prince",
        "Harry Potter and the Deathly Hallows"
    ]
}
```

上述例子中的 JSON 整体是一个对象的形式，这个对象包含三个属性，分别是 name、author 和 books。其中 name 的值是字符串，为"Harry Potter"；author 的值是一个对象，这个对象有两个属性，分别是 name 和 birth，name 的值是字符串，为"J.K.Rowling"，birth 的值是数字，为 1964，可以说 author 的值也是一个 JSON 格式的数据；books 的值是数组，这个数组包含七个元素，每个元素都是一个字符串。

接下来我们看看 Chrome 扩展中 Manifest 的内容。Google 的官方文档中对于扩展和应用给出了两个不同的 Manifest 介绍界面，这是因为有些属性只能由扩展使用，而有些属性只能由应用使用。如果这两者同时出现在同一个 Manifest 文件中，就会使 Chrome 困惑，不知是按照扩展对待这个程序还是按照应用来对待这个程序。但无论是扩展还是应用，它们的 Manifest 又有很多共有的属性，所以我决定还是放到一起讲。

Chrome 扩展的 Manifest 必须包含 name、version 和 manifest_version 属性，目前来说 manifest_version 属性值只能为数字 2，对于应用来说，还必须包含 app 属性。

其他常用的可选属性还有 browser_action、page_action、background、permissions、options_page、content_scripts，所以我们可以保留一份 manifest.json 模板，当编写新的扩展时直接填入相应的属性值。如果我们需要的属性不在这个模板中，可以再去查阅官方文档，但我想这样的一份模板可以应对大部分的扩展了。

```
{
    "app": {
        "background": {
            "scripts": ["background.js"]
        }
    },
    "manifest_version": 2,
    "name": "My Extension",
    "version": "versionString",
```

```
    "default_locale": "en",
    "description": "A plain text description",
    "icons": {
        "16": "images/icon16.png",
        "48": "images/icon48.png",
        "128": "images/icon128.png"
    },
    "browser_action": {
        "default_icon": {
            "19": "images/icon19.png",
            "38": "images/icon38.png"
        },
        "default_title": "Extension Title",
        "default_popup": "popup.html"
    },
    "page_action": {
        "default_icon": {
            "19": "images/icon19.png",
            "38": "images/icon38.png"
        },
        "default_title": "Extension Title",
        "default_popup": "popup.html"
    },
    "background": {
        "scripts": ["background.js"]
    },
    "content_scripts": [
        {
            "matches": ["http://www.google.com/*"],
            "css": ["mystyles.css"],
            "js": ["jquery.js", "myscript.js"]
        }
    ],
    "options_page": "options.html",
    "permissions": [
        "*://www.google.com/*"
    ],
    "web_accessible_resources": [
        "images/*.png"
    ]
}
```

在官方文档中可以找到完整的 Manifest 属性列表，扩展在 https://developer.chrome.com/extensions/manifest ，应用在

https://developer.chrome.com/apps/manifest。由于 Google 更新得非常频繁，上述页面内容可能会经常变动，但那些比较基本的属性变动的几率不会很大。

1.4 DOM 简述

DOM 是 Document Object Model 的缩写，翻译过来叫文档对象模型，但我觉得这个听起来很生疏，不如还是直接叫 DOM，所以本节的标题就定为了 DOM 简述。由于 Chrome 扩展应用使用 HTML 渲染界面，所以不可避免地要接触 DOM。考虑到并非所有读者都编写过 HTML，我决定单独拿出一小节来讲解 DOM，帮助这些读者快速入门。当然，用短短的一节是无法讲透的——毕竟 DOM 可以写另外一本书了——这里只是要给大家引出一个方向，浅浅地打下一点基础，深入的学习还需要读者去阅读更加详细的资料。

HTML DOM 树，图片来源于 www.w3school.com.cn

DOM 分为 3 个不同的部分，分别是核心 DOM、XML DOM 和 HTML DOM，我们主要关心的是 HTML DOM，所以我也只讲解 HTML DOM。

上图给出了 HTML DOM 的树状结构图，可以看到 HTML 文档都有一个<html>根元素。<html>根元素又有两个子元素，分别是<head>和<body>，所以已经最简单而完整的 HTML 文档如下所示：

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

这个文档没有任何内容，但拥有 HTML 完整的结构。在 DOM 中，每个元素通常是以<tag_name>的形式开始，并以</tag_name>的形式结束。在 HTML 中，有一些特定的 tag_name，如 div、p、a、form 等等。

这些元素可以包含一些属性，还可以包含子节点，子节点可以是元素也可以是文本。如：

```
<img src="images/dog.png" />
<div>Hello World!</div>
```

上面的例子中 img 元素不是以成对的标签形式出现的，而是在标签内部末尾使用“/”闭合标签，这样的元素在 HTML 文档中没有子节点，所以称为自闭标签。类似的元素还有 input。

除了自闭标签，其他的标签必须成对出现，并且嵌套规则必须明确，这有点像我们小学时学习数学所使用的括号“()”和中括号“[]”。比如下面的嵌套方式是正确的：

```
<div><p>Hello World!</p></div>
```

但下面的例子是错误的：

```
<div><p></div></p>
<div><p></div>
```

第一个是嵌套错误，第二个是 p 标签没有成对出现，标签没有闭合。

有时元素还会拥有属性，比如下面的例子：

```
<input type="text" id="stu_name" value="Billy" />
```

上面这个 input 有三个属性，分别是 type、id 和 value，type="text"表明这个输入框的类型是文本输入框，id="stu_name"表明给这个元素分配了一个名为 stu_name 的 id，这样可以更加方便地被 JavaScript 和 CSS 选择器定位到，value="Billy"表明将这个输入框的默认值设定为 Billy。

不同的元素往往拥有不同的属性名，比如对于 img 元素，通常会包含 src 属性以指定所显示图片的地址，而 input 元素往往会包含 type 属性来描述输入框的类型。

在 JavaScript 中有多种获取 DOM 元素的方法，常见的有 getElementById、getElementsByName、getElementsByTagName、getElementsByClassName，分别是通过 id、name、标签名和类名获取元素。

请注意，上面提到的四种方法中，第一个方法名中是 Element，而后面的都是 Elements。这是因为 HTML 中元素的 id 必须是唯一的，但是不同的元素可以拥有同样的 name、标签名和类名，所以通过第一种方式获取的是一个元素，而后几种方法获取的是一个包含多个元素的数组。值得强调的是，即使 HTML 中只有一个元素的 name 为"my_element"，那么通过 getElementsByName('my_element')获取到的也是数组型的数据——虽然这个数组只包含一个元素。

JavaScript 可以通过 getAttribute 方法读取元素的属性，通过 setAttribute 方法添加或更改元素的属性，通过 removeAttribute 方法删除元素的属性。对于非自定义的属性，JavaScript 可以直接像读取对象属性那样读取或更改它们，比如：

```
var imgurl = document.getElementById('my_image').src;
document.getElementById('my_another_image').src = imgurl;
// var imgurl = document.getElementById('my_image').getAttribute('src');
// document.getElementById('my_another_image').setAttribute('src', imgurl);
```

CSS 的选择器基本分为三种，分别是 tagName、.className 和#id。如下面的例子：

```
p {
      width: 200px;
}

.postlist {
      width: 150px;
}

#footer {
    width: 100px;
}
```

分别定义了 p 标签元素宽度为 200 像素，类名为 postlist 的元素宽度为 150 像素，id 为 footer 的元素宽度为 100 像素。这个样式表分别作用于以下元素：

```
<p></p>
<div class="postlist"></div>
<div id="footer"></div>
```

CSS 选择器还可以通过元素属性进行定位，比如下面的例子可以作用于所有文本输入框：

```
input[type="text"] {
    font-size: 16px;
}
```

更多关于 DOM 的知识可以参阅 http://www.w3school.com.cn/htmldom/。

第 2 章　Chrome 扩展基础

本章会讲解 Chrome 扩展的一些基础功能，这些基础的功能在后续的扩展编写中可能会被频繁用到，所以有必要提前进行详细的讲解。本章会配有多个实例，一步步带着读者完成一个个有趣的例子。

2.1　操作用户正在浏览的页面

通过 Chrome 扩展我们可以对用户当前浏览的页面进行操作，实际上就是对用户当前浏览页面的 DOM 进行操作。通过 Manifest 中的 content_scripts 属性可以指定将哪些脚本何时注入到哪些页面中，当用户访问这些页面后，相应脚本即可自动运行，从而对页面 DOM 进行操作。

Manifest 的 content_scripts 属性值为数组类型，数组的每个元素可以包含 matches、exclude_matches、css、js、run_at、all_frames、include_globs 和 exclude_globs 等属性。其中 matches 属性定义了哪些页面会被注入脚本，exclude_matches 则定义了哪些页面不会被注入脚本，css 和 js 对应要注入的样式表和 JavaScript，run_at 定义了何时进行注入，all_frames 定义脚本是否会注入到嵌入式框架中，include_globs 和 exclude_globs 则是全局 URL 匹配，最终脚本是否会被注入由 matches、exclude_matches、include_globs 和 exclude_globs 的值共同决定。简单的说，如果 URL 匹配 mathces 值的同时也匹配 include_globs 的值，会被注入；如果 URL 匹配 exclude_matches 的值或者匹配 exclude_globs 的值，则不会被注入。

content_scripts 中的脚本只是共享页面的 DOM1，而并不共享页面内嵌 JavaScript 的命名空间。也就是说，如果当前页面中的 JavaScript 有一个全局变量 a，content_scripts 中注入的脚本也可以有一个全局变量 a，两者不会相互干扰。当然你也无法通过 content_scripts 访问到页面本身内嵌 JavaScript 的变量和函数。

1 DOM 中的自定义属性不会被共享。

下面我们来写一个恶作剧的小扩展，名字就叫做永远点不到的搜索按钮吧 :)

首先创建 Manifest 文件，内容如下：

```
{
    "manifest_version": 2,
    "name": "永远点不到的搜索按钮",
    "version": "1.0",
    "description": "让你永远也点击不到 Google 的搜索按钮",
```

```
    "content_scripts": [
        {
            "matches": ["*://www.google.com/"],
            "js": ["js/cannot_touch.js"]
        }
    ]
}
```

在 content_scripts 属性中我们定义了一个匹配规则，当 URL 符合*://www.google.com/规则的时候，就将 js/cannot_touch.js 注入到页面中。其中*代表任意字符，这样当用户访问 http://www.google.com/和 https://www.google.com/时就会触发脚本。

右键单击搜索按钮，选择“审查元素”，我们发现 Google 搜索按钮的 id 为'gbqfba'。

通过 Chrome 浏览器的开发者工具可以看到 Google 搜索按钮的 id

接下来我们开始编写 cannot_touch.js。

```
function btn_move(el, mouseLeft, mouseTop){
    var leftRnd = (Math.random()-0.5)*20;
    var topRnd = (Math.random()-0.5)*20;
    var btnLeft = mouseLeft+(leftRnd>0?100:-100)+leftRnd;
    var btnTop = mouseTop+(topRnd>0?30:-30)+topRnd;
    btnLeft = btnLeft<100?(btnLeft+window.innerWidth-200):(btnLeft>window.innerWidth-100?btnLeft-window.innerWidth+200:btnLeft);
    btnTop = btnTop<100?(  btnTop+window.innerHeight-200):(btnTop>window.innerHeight-100?btnTop-window.innerHeight+200:btnTop);
    el.style.position = 'fixed';
    el.style.left = btnLeft+'px';
    el.style.top = btnTop+'px';
}

function over_btn(e){
    if(!e){
        e = window.event;
    }
    btn_move(this, e.clientX, e.clientY);
}

document.getElementById('gbqfba').onmouseover = over_btn;
```

由于 Manifest 将此脚本的位置指定到了 js/cannot_touch.js，所以要记得将这个脚本保存到扩展文件夹中的 js 文件夹下，否则会出现错误。

“永远点不到的搜索按钮”扩展运行的结果

可以看出，content_scripts 很像 Userscript，它就是将指定的脚本文件插入到符合规则的特定页面中，从而使插入的脚本可以对页面的 DOM 进行操作。

这个扩展的源码可以在 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/cannot_touch 下载到。

2.2 跨域请求

跨域指的是 JavaScript 通过 XMLHttpRequest 请求数据时，调用 JavaScript 的页面所在的域和被请求页面的域不一致。对于网站来说，浏览器出于安全考虑是不允许跨域。另外，对于域相同，但端口或协议不同时，浏览器也是禁止的。下表给出了进一步的说明：

URL
说明
是否允许请求

http://a.example.com/http://a.example.com/a.txt
同域下
允许

http://a.example.com/http://a.example.com/b/a.txt
同域下不同目录
允许

http://a.example.com/http://a.example.com:8080/a.txt
同域下不同端口
不允许

http://a.example.com/https://a.example.com/a.txt
同域下不同协议
不允许

http://a.example.com/http://b.example.com/a.txt
不同域下
不允许

http://a.example.com/http://a.foo.com/a.txt
不同域下
不允许

但这个规则如果同样限制 Chrome 扩展应用，就会使其能力大打折扣，所以 Google 允许 Chrome 扩展应用不必受限于跨域限制。但出于安全考虑，需要在 Manifest 的 permissions 属性中声明需要跨域的权限。

比如，如果我们想设计一款获取维基百科数据并显示在其他网页中的扩展，就要在 Manifest 中进行如下声明：

```
{
    ...
    "permissions": [
        "*://*.wikipedia.org/*"
    ]
}
```

这样 Chrome 就会允许你的扩展在任意页面请求维基百科上的内容了。

我们可以利用如下的代码发起异步请求：

```
function httpRequest(url, callback){
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            callback(xhr.responseText);
        }
    }
    xhr.send();
}
```

这样每次发起请求时，只要调用 httpRequest 函数，并传入要请求的 URL 和接收返回结果的函数就可以了。为什么要使用 callback 函数接收请求结果，而不直接用 return 将结果作为函数值返回呢？因为 XMLHttpRequest 不会阻塞下面代码的运行。

为了更加明确地说清上述问题，让我们来举两个例子。

```
function count(n){
    var sum = 0;
    for(var i=1; i<=n; i++){
        sum += i;
    }
    return sum;
}

var c = count(5)+1;
```

```
console.log(c);
```

上面这个例子会在控制台显示 16，因为 count(5)=1+2+3+4+5=15，c=15+1=16。我们再看下面的例子：

```
function httpRequest(url){
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            return xhr.responseText;
        }
    }
    xhr.send();
}

var html = httpRequest('test.txt');
console.log(html);
```

上例运行结果

通过上图可以发现，虽然请求的资源内容为 Hello World!，但却并没有正确地显示出来。

对于第一个例子，count 函数是一个阻塞函数，在它没有运行完之前它会阻塞下面的代码运行，所以直到 count 计算结束后才将结果返回后再加 1 赋给变量 c，最后将变量 c 的值打印出来。而第二个例子中的 httpRequest 函数不是一个阻塞函数，在它没运行完之前后面的代码就已经开始运行，这样 html 变量在 httpRequest 函数没返回值之前就被赋值，所以最终的结果必然就是 undefined 了。

既然这样，如何将非阻塞函数的最终结果传递下去呢？方法就是使用回调函数。在 Chrome 扩展应用的 API 中，大部分函数都是非阻塞函数，所以使用回调函数传递结果的方法以后会经常用到。

让我们来用回调函数的形式重写第二个例子：

```
function httpRequest(url, callback){
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            callback(xhr.responseText);
        }
    }
    xhr.send();
}

var html;
httpRequest('test.txt', function(result){
    html = result;
```

```
    console.log(html);
});
```

改进后第二个例子的结果

可以看到 httpRequest 函数运行的结果已经被正确地打印出来了。

下面来实战编写一款显示用户 IP 的扩展。

```
{
    "manifest_version": 2,
    "name": "查看我的 IP",
    "version": "1.0",
    "description": "查看我的电脑当前的公网 IP",
    "icons": {
        "16": "images/icon16.png",
        "48": "images/icon48.png",
        "128": "images/icon128.png"
    },
    "browser_action": {
        "default_icon": {
            "19": "images/icon19.png",
            "38": "images/icon38.png"
        },
        "default_title": "查看我的 IP",
        "default_popup": "popup.html"
    },
    "permissions": [
        "http://sneezryworks.sinaapp.com/ip.php"
    ]
}
```

上面的 Manifest 定义了这个扩展允许对 http://sneezryworks.sinaapp.com/ip.php 发起跨域请求，其他的属性在 1.2 节中都有介绍，在此就不再赘述了。

popup.html 的结构也完全可以按照时钟的扩展照抄下来，只是个别元素的 id 和脚本的路径根据当前扩展的名称稍加更改，同样不再赘述。

```
<html>
<head>
<style>
* {
    margin: 0;
    padding: 0;
}

body {
    width: 400px;
```

```
        height: 100px;
    }

    div {
        line-height: 100px;
        font-size: 42px;
        text-align: center;
    }
    </style>
    </head>
    <body>
    <div id="ip_div">正在查询……</div>
    <script src="js/my_ip.js"></script>
    </body>
    </html>
```

下面编写 my_ip.js。

```
function httpRequest(url, callback){
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            callback(xhr.responseText);
        }
    }
    xhr.send();
}

httpRequest('http://sneezryworks.sinaapp.com/ip.php', function(ip){
    document.getElementById('ip_div').innerText = ip;
});
```

“查看我的 IP”扩展运行结果

作为一个开发者，安全问题永远都不应被轻视。在你从外域获取到数据后，不要轻易作为当前页面元素的 innerHTML 直接插入，更不要用 eval 函数去执行它，否则很可能将用户置于危险的境地。如果要将请求到的数据写入页面，可以使用 innerText，就像我们这个查看 IP 的扩展那样。如果是 JSON 格式是数据就使用 JSON.parse 函数去解析。为了避免请求数据返回的格式错误，结合 try-catch 一起使用也是不错的选择。

本节中扩展的源码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/what_is_my_ip 下载到。

2.3 常驻后台

有时我们希望扩展不仅在用户主动发起时（如开启特定页面或点击扩展图标等）才运行，而是希望扩展自动运行并常驻后台来实现一些特定的功能，比如实时提示未读邮件数量、后台播放音乐等等。

Chrome 允许扩展应用在后台常驻一个页面以实现这样的功能。在一些典型的扩展中，UI 页面，如 popup 页面或者 options 页面，在需要更新一些状态时，会向后台页面请求数据，而当后台页面检测到状态发生改变时，也会通知 UI 界面刷新。

后台页面与 UI 页面可以相互通信，这将在后续的章节中做进一步的讲解，本节将主要讲解后台页面是如何工作的。

在 Manifest 中指定 background 域可以使扩展常驻后台。background 可以包含三种属性，分别是 scripts、page 和 persistent。如果指定了 scripts 属性，则 Chrome 会在扩展启动时自动创建一个包含所有指定脚本的页面；如果指定了 page 属性，则 Chrome 会将指定的 HTML 文件作为后台页面运行。通常我们只需要使用 scripts 属性即可，除非在后台页面中需要构建特殊的 HTML——但一般情况下后台页面的 HTML 我们是看不到的。persistent 属性定义了常驻后台的方式——当其值为 true 时，表示扩展将一直在后台运行，无论其是否正在工作；当其值为 false 时，表示扩展在后台按需运行，这就是 Chrome 后来提出的 Event Page。Event Page 可以有效减小扩展对内存的消耗，如非必要，请将 persistent 设置为 false。persistent 的默认值为 true。

由于编写一个只有后台页面的扩展，很难看到扩展运行的结果，所以我决定在本节中破例使用一个尚未讲到但是很简单的扩展功能，动态改变扩展图标，这在后面的例子中会进行说明。

下面我们来编写一款实时监视网站在线状态的扩展。思路很简单，每隔 5 秒就发起一次连接请求，如果请求成功就代表网站在线，将扩展图标显示为绿色，如果请求失败就代表网站不在线，将扩展图标显示为红色。

下面是这个扩展的 Manifest 文件，此例中以检测 www.google.cn 为例，你可以根据自己的意愿更改为其他的网站。

```
{
    "manifest_version": 2,
    "name": "Google 在线状态",
    "version": "1.0",
    "description": "监视 Google 是否在线",
    "icons": {
        "16": "images/icon16.png",
        "48": "images/icon48.png",
        "128": "images/icon128.png"
    },
    "browser_action": {
        "default_icon": {
            "19": "images/icon19.png",
            "38": "images/icon38.png"
        }
```

```
    },
    "background": {
        "scripts": [
            "js/status.js"
        ]
    },
    "permissions": [
        "http://www.google.cn/"
    ]
}
```

由于这个扩展没有 UI，所以我们不必编写 HTML 文件，下面直接编写 status.js。

```
function httpRequest(url, callback){
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            callback(true);
        }
    }
    xhr.onerror = function(){
        callback(false);
    }
    xhr.send();
}

setInterval(function(){
    httpRequest('http://www.google.cn/', function(status){
        chrome.browserAction.setIcon({path:
'images/'+(status?'online.png':'offline.png')});
    });
},5000);
```

status.js 调用了我们之前没有介绍过的方法，chrome.browserAction.setIcon。Chrome 为扩展应用提供了很多类似的方法可以使得扩展应用做更多的事情，并且与浏览器结合得更加紧密。这个方法的作用就是更换扩展在浏览器工具栏中的图标。

本节示例扩展中的 httpRequest 函数，与上节所讲述跨域请求中所使用的函数非常类似，但请注意本节在 httpRequest 函数中加入了 onerror 事件，正是因为加入了这个事件才能捕捉到请求过程中是否发生了错误，从而得知所监视的网站是否在线。将本例载入 Chrome 后，在联网的情况下可以看到扩展图标为绿色，断开网络连接后扩展图标变为了红色。

本节示例扩展的运行结果

小提示：如果想在用户打开浏览器之前就让扩展运行，可以在 Manifest 的 permissions 属性中加入"background"，但除非必要，否则尽量不要这么做，因为大部分用户不喜欢这样。

本例中所编写的扩展源码可以通过https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/website_status 下载得到。

2.4 带选项页面的扩展

有一些扩展允许用户进行个性化设置，这样就需要向用户提供一个选项页面。Chrome通过 Manifest 文件的 options_page 属性为开发者提供了这样的接口，可以为扩展指定一个选项页面。当用户在扩展图标上点击右键，选择菜单中的“选项”后，就会打开这个页面 1。

1 对于没有图标的扩展，可以在 chrome://extensions 页面中单击“选项”。

指定 options_page 属性后，扩展图标上的右键菜单会包含“选项”链接

对于网站来说，用户的设置通常保存在 Cookies 中，或者保存在网站服务器的数据库中。对于 JavaScript 来说，一些数据可以保存在变量中，但如果用户重新启动浏览器，这些数据就会消失。那么如何在扩展中保存用户的设置呢？我们可以使用 HTML5 新增的 localStorage 接口。除了 localStorage 接口以外，还可以使用其他的储存方法。后面将专门拿出一节来讲解数据存储，本节中我们先使用最简单的 localStorage 方法储存数据。

localStorage 是 HTML5 新增的方法，它允许 JavaScript 在用户计算机硬盘上永久储存数据（除非用户主动删除）。但 localStorage 也有一些限制，首先是 localStorage 和 Cookies 类似，都有域的限制，运行在不同域的 JavaScript 无法调用其他域 localStorage 的数据；其次是单个域在 localStorage 中存储数据的大小通常有限制（虽然 W3C 没有给出限制），对于 Chrome 这个限制是 5MB2；最后 localStorage 只能储存字符串型的数据，无法保存数组和对象，但可以通过 join、toString 和 JSON.stringify 等方法先转换成字符串再储存。

2 通过声明 unlimitedStorage 权限，Chrome 扩展和应用可以突破这一限制。

下面我们将编写一个天气预报的扩展，这个扩展将提供一个选项页面供用户填写所关注的城市。

有很多网站提供天气预报的 API，比如 OpenWeatherMap 的 API。可以通过 http://openweathermap.org/API 了解更多相关内容。

```
{
    "manifest_version": 2,
    "name": "天气预报",
    "version": "1.0",
    "description": "查看未来两周的天气情况",
    "icons": {
        "16": "images/icon16.png",
        "48": "images/icon48.png",
        "128": "images/icon128.png"
    },
    "browser_action": {
        "default_icon": {
            "19": "images/icon19.png",
            "38": "images/icon38.png"
```

```
        },
        "default_title": "天气预报",
        "default_popup": "popup.html"
    },
    "options_page": "options.html",
    "permissions": [
        "http://api.openweathermap.org/data/2.5/forecast?q=*"
    ]
}
```

上面是这个扩展的 Manifest 文件，options.html 为设定选项的页面。下面开始编写 options.html 文件。

```
<html>
    <head>
        <title>设定城市</title>
    </head>
    <body>
        <input type="text" id="city" />
        <input type="button" id="save" value="保存" />
        <script src="js/options.js"></script>
    </body>
</html>
```

这个页面提供了一个 id 为 city 的文本框和一个 id 为 save 的按钮。由于 Chrome 不允许将 JavaScript 内嵌在 HTML 文件中，所以我们单独编写一个 options.js 脚本文件，并在 HTML 文件中引用它。下面来编写 options.js 文件。

```
var city = localStorage.city || 'beijing';
document.getElementById('city').value = city;
document.getElementById('save').onclick = function(){
    localStorage.city = document.getElementById('city').value;
    alert('保存成功。');
}
```

从 options.js 的代码中可以看到，localStorage 的读取和写入方法很简单，和 JavaScript 中的变量读写方法类似。localStorage 除了使用 localStorage.namespace 的方法引用和写入数据外，还可以使用 localStorage['namespace']的形式。请注意第二种方法 namespace 要用引号包围，单引号和双引号都可以。如果想彻底删除一个数据，可以使用 localStorage.removeItem('namespace')方法。

为了显示天气预报的结果，我们为扩展指定了一个 popup 页面，popup.html。下面来编写这个 UI 页面。

```
<html>
<head>
<style>
* {
```

```
        margin: 0;
        padding: 0;
}

body {
        width: 520px;
        height: 270px;
}

table {
        font-family: "Lucida Sans Unicode", "Lucida Grande", Sans-Serif;
        font-size: 12px;
        width: 480px;
        text-align: left;
        border-collapse: collapse;
        border: 1px solid #69c;
        margin: 20px;
        cursor: default;

}

table th {
        font-weight: normal;
        font-size: 14px;
        color: #039;
        border-bottom: 1px dashed #69c;
        padding: 12px 17px;
        white-space: nowrap;
}

table td {
        color: #669;
        padding: 7px 17px;
        white-space: nowrap;
}

table tbody tr:hover td {
        color: #339;
        background: #d0dafd;
}

</style>
</head>
<body>
```

```
<div id="weather"></div>
<script src="js/weather.js"></script>
</body>
</html>
```

其中 id 为 weather 的 div 元素将用于显示天气预报的结果。下面来编写 weather.js 文件。

```
function httpRequest(url, callback){
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            callback(xhr.responseText);
        }
    }
    xhr.send();
}

function showWeather(result){
    result = JSON.parse(result);
    var list = result.list;
    var table = '<table><tr><th>日期</th><th>天气</th><th>最低温度</th><th>最高温度</th></tr>';
    for(var i in list){
        var d = new Date(list[i].dt*1000);
        table += '<tr>';
        table += '<td>'+d.getFullYear()+'-'+(d.getMonth()+1)+'-'+d.getDate()+'</td>';
        table += '<td>'+list[i].weather[0].description+'</td>';
        table += '<td>'+Math.round(list[i].temp.min-273.15)+'°C</td>';
        table += '<td>'+Math.round(list[i].temp.max-273.15)+'°C</td>';
        table += '</tr>';
    }
    table += '</table>';
    document.getElementById('weather').innerHTML = table;
}

var city = localStorage.city;
city = city?city:'beijing';
var url = 'http://api.openweathermap.org/data/2.5/forecast/daily?q='+city+',china&lang=zh_cn';
httpRequest(url, showWeather);
```

小提示：无论是 options.js 还是 weather.js 中都有如下语句：

```
var city = localStorage.city;
city = city?city:'beijing';
```

也就是说，当选项没有值时，应设定一个默认值，以避免程序出错。此处如果用户未设置城市，扩展将显示北京的天气预报。

weather 扩展的选项页面，点击保存按钮后会提示保存成功

weather 扩展的运行界面

本节示例扩展的源代码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/weather 下载得到。

2.5 扩展页面间的通信

有时需要让扩展中的多个页面之间，或者不同扩展的多个页面之间相互传输数据，以获得彼此的状态。比如音乐播放器扩展，当用户鼠标点击 popup 页面中的音乐列表时，popup 页面应该将用户这个指令告知后台页面，之后后台页面开始播放相应的音乐。

Chrome 提供了 4 个有关扩展页面间相互通信的接口，分别是 runtime.sendMessage、runtime.onMessage、runtime.connect 和 runtime.onConnect。做为一部入门级教程，此节将只讲解 runtime.sendMessage 和 runtime.onMessage 接口，runtime.connect 和 runtime.onConnect 做为更高级的接口请读者依据自己的兴趣自行学习，你可以在 http://developer.chrome.com/extensions/extension 得到有关这两个接口的完整官方文档。

请注意，Chrome 提供的大部分 API 是不支持在 content_scripts 中运行的，但 runtime.sendMessage 和 runtime.onMessage 可以在 content_scripts 中运行，所以扩展的其他页面也可以同 content_scripts 相互通信。

runtime.sendMessage 完整的方法为：

chrome.runtime.sendMessage(extensionId, message, options, callback)

其中 extensionId 为所发送消息的目标扩展，如果不指定这个值，则默认为发起此消息的扩展本身；message 为要发送的内容，类型随意，内容随意，比如可以是'Hello'，也可以是{action: 'play'}、2013 和['Jim', 'Tom', 'Kate']等等；options 为对象类型，包含一个值为布尔型的 includeTlsChannelId 属性，此属性的值决定扩展发起此消息时是否要将 TLS 通道 ID 发送给监听此消息的外部扩展 1，有关 TLS 的相关内容可以参考 http://www.google.com/intl/zh-CN/chrome/browser/privacy/whitepaper.html#tls，这是有关加强用户连接安全性的技术，如果这个参数你捉摸不透，不必理睬它，options 是一个可选参数；callback 是回调函数，用于接收返回结果，同样是一个可选参数。

1 此属性仅在扩展和网页间通信时才会用到。

runtime.onMessage 完整的方法为：

chrome.runtime.onMessage.addListener(callback)

此处的 callback 为必选参数，为回调函数。callback 接收到的参数有三个，分别是 message、sender 和 sendResponse，即消息内容、消息发送者相关信息和相应函数。其中 sender 对象包含 4 个属性，分别是 tab、id、url 和 tlsChannelId，tab 是发起消息的标签，有关标签的内

容可以参看 4.5 节的内容。

为了进一步说明，下面举一个例子。

在 popup.html 中执行如下代码：

```
chrome.runtime.sendMessage('Hello', function(response){
    document.write(response);
});
```

在 background 中执行如下代码：

```
chrome.runtime.onMessage.addListener(function(message, sender, sendResponse){
    if(message == 'Hello'){
        sendResponse('Hello from background.');
    }
});
```

查看 popup.html 页面会发现有输出“Hello from background.”。

扩展内部通信 Demo 的运行画面

上面这个小例子的源代码可以从 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/runtime.sendMessage_runtime.onMessage_demo 下载到。

2.6 储存数据

一个程序免不了要储存数据，对于 Chrome 扩展也是这样。通常 Chrome 扩展使用以下三种方法中的一种来储存数据：第一种是使用 HTML5 的 localStorage，这种方法在上一节的内容中已经涉及；第二种是使用 Chrome 提供的存储 API；第三种是使用 Web SQL Database。

对于一般的扩展，“设置”这种简单的数据可以优先选择第一种，因为这种方法使用简单，可以看成是特殊的 JavaScript 变量；对于结构稍微复杂一些的数据可以优先选择第二种，这种方法可以保存任意类型的数据，但需要异步调用 Chrome 的 API，结果需要使用回调函数接收，不如第一种操作简单；第三种目前使用的不算太多，因为需要使用 SQL 语句对数据库进行读写操作，较前两者更加复杂，但是对于数据量庞大的应用来说是个不错的选择。开发者应根据实际的情况选择上述三种方法中的一种或几种来存储扩展中的数据。

由于上节已经讲解了 localStorage 的使用方法，下面将详细讲解后两种储存数据的方法。

Chrome 存储 API

Chrome 为扩展应用提供了存储 API，以便将扩展中需要保存的数据写入本地磁盘。Chrome 提供的存储 API 可以说是对 localStorage 的改进，它与 localStorage 相比有以下区别：

如果储存区域指定为 sync，数据可以自动同步；

content_scripts 可以直接读取数据，而不必通过 background 页面；

在隐身模式下仍然可以读出之前存储的数据；

读写速度更快；

用户数据可以以对象的类型保存。

对于第二点要进一步说明一下。首先 localStorage 是基于域名的，这在前面的小节中已经提到过了。而 content_scripts 是注入到用户当前浏览页面中的，如果 content_scripts 直接读取 localStorage，所读取到的数据是用户当前浏览页面所在域中的。所以通常的解决办法是 content_scripts 通过 runtime.sendMessage 和 background 通信，由 background 读写扩展所在域（通常是 chrome-extension://extension-id/）的 localStorage，然后再传递给 content_scripts。

使用 Chrome 存储 API 必须要在 Manifest 的 permissions 中声明"storage"，之后才有权限调用。Chrome 存储 API 提供了 2 种储存区域，分别是 sync 和 local。两种储存区域的区别在于，sync 储存的区域会根据用户当前在 Chrome 上登陆的 Google 账户自动同步数据，当无可用网络连接可用时，sync 区域对数据的读写和 local 区域对数据的读写行为一致。

对于每种储存区域，Chrome 又提供了 5 个方法，分别是 get、getBytesInUse、set、remove 和 clear。

get 方法即为读取数据，完整的方法为：

```
chrome.storage.StorageArea.get(keys, function(result){
    console.log(result);
});
```

keys 可以是字符串、包含多个字符串的数组或对象。如果 keys 是字符串，则和 localStorage 的用法类似；如果是数组，则相当于一次读取了多个数据；如果 keys 是对象，则会先读取以这个对象属性名为键值的数据，如果这个数据不存在则返回 keys 对象的属性值（比如 keys 为{'name':'Billy'}，如果 name 这个值存在，就返回 name 原有的值，如果不存在就返回 Billy）。如果 keys 为一个空数组（[]）或空对象（{}），则返回一个空列表，如果 keys 为 null，则返回所有存储的数据。

getBytesInUse 方法为获取一个数据或多个数据所占用的总空间，返回结果的单位是字节，完整方法为：

```
chrome.storage.StorageArea.getBytesInUse(keys, function(bytes){
    console.log(bytes);
});
```

此处的 keys 只能为 null、字符串或包含多个字符串的数组。

set 方法为写入数据，完整方法为：

```
chrome.storage.StorageArea.set(items, function(){
    //do something
});
```

items 为对象类型，形式为键/值对。items 的属性值如果是字符型、数字型和数组型，则储存的格式不会改变，但如果是对象型和函数型的，会被储存为“{}”，如果是日期型和正则型的，会被储存为它们的字符串形式。

remove 方法为删除数据，完整方法为：

```
chrome.storage.StorageArea.remove(keys, function(){
    //do something
});
```

其中 keys 可以是字符串，也可以是包含多个字符串的数组。

clear 方法为删除所有数据，完整方法为：

```
chrome.storage.StorageArea.clear(function(){
    //do something
});
```

请注意，上述五种完整方法中，StorageArea 必须指定为 local 或 sync 中的一个。

Chrome 同时还为存储 API 提供了一个 onChanged 事件，当存储区的数据发生改变时，这个事件会被激发。

onChanged 的完整方法为：

```
chrome.storage.onChanged.addListener(function(changes, areaName){
    console.log('Value in '+areaName+' has been changed:');
    console.log(changes);
});
```

callback 会接收到两个参数，第一个为 changes，第二个是 StorageArea。changes 是词典对象，键为更改的属性名称，值包含两个属性，分别为 oldValue 和 newValue；StorageArea 为 local 或 sync。

Web SQL Database

Web SQL Database 的三个核心方法为 openDatabase、transaction 和 executeSql。openDatabase 方法的作用是与数据库建立连接，transaction 方法的作用是执行查询，executeSql 方法的作用是执行 SQL 语句。

下面举一个简单的例子：

```
db = openDatabase("db_name", "0.1", "This is a test db.", 1024*1024);
if(!db){
    alert('数据库连接失败。');
}
else {
    db.transaction( function(tx) {
        tx.executeSql(
            "SELECT COUNT(*) FROM db_name",
            [],
            function(tx, result){
                console.log(result);
            },
            function(tx, error){
                alert('查询失败：'+error.message);
            }
        );
    }
}
```

更多关于 Web SQL Database 的资料可以参考 http://www.w3.org/TR/webdatabase/。由于原生的 Web SQL Database 并不算好用，也有一些开源的二次封装的库来简化 Web SQL Database 的使用，如 https://github.com/KenCorbettJr/html5sql。

以上几种数据的存储方式都不会对数据加密，如果储存的是敏感的数据，应该先进行加密处理。比如不要将用户密码的明码直接储存，而应先进行 MD5 加密。

第 3 章　Chrome 扩展的 UI 界面

前两章我们所设计的扩展，使用的 UI 设计都非常简单。对于一个面向用户的产品，这样显然是不合适的。用户对一个程序的第一印象就是 UI 的设计，拙劣的 UI 设计完全可能将 90%的用户挡在门外——即使功能设计得非常完美。

本章将专门讲解 Chrome 扩展的 UI 界面，通过 Chrome 提供丰富的界面 API，我们可以设计出交互出色的扩展。

3.1　CSS 简述

CSS 是 Cascading Style Sheets 的缩写，翻译过来叫做层叠样式表，一般简称为样式表，但通常大家还是习惯叫 CSS。

最初的 HTML 很单一，甚至无法显示图片，随着使用范围越来越广泛，HTML 支持的标签开始多了起来，所支持的样式也开始增多。但是把样式完全交给 HTML 去做不是一个好想法，因为 HTML 更侧重于页面的结构，于是在 1994 年 CSS 被提出。CSS 旨在对 HTML 元素的外观加以描述，来提供更多更加复杂丰富的样式。

现在多数浏览器会默认使用一些样式，比如 div 元素默认会占据整行——两个 div 元素不会出现在同一行，而 span 元素则不是这样，这就是因为浏览器默认将 div 元素样式的 display 属性值设为了 block。

浏览器这种为 HTML 元素附加默认样式的做法，大部分情况下是好的，但有时为了个性化设计，我们需要另外编写 CSS 来自定义 HTML 元素的外观。

Chrome 会自动为 HTML 元素附加 margin 和 padding 样式

我们来看看上图所示的页面在 Chrome 浏览器中的渲染结果。可以看到 HTML 元素并没有被指定样式，因为我们没有编写 CSS。但是 Chrome 已经自动为文本框添加了 margin 和 padding 样式，这在外观表现上，会在文本框周围有一圈间隙，这样其他 HTML 元素不会与它挨得太紧。这种设计显然是出于好意，但有时我们需要更加灵活个性化的样式，这就是为什么在前面的例子中都会出现下面的代码。

```
* {
    margin: 0;
    padding: 0;
}
```

CSS 的选择器在第 1 章第 4 节已经介绍过，在此就不再赘述。下面讲一讲 CSS 的基本语法。

CSS 是一种描述型语言，它更像是一种陈述，而不是逻辑运算。CSS 的语法形式如下所

示：

```
选择器 {
    属性名: 属性值;
}
```

CSS 的选择器非常灵活，更加高级的使用方法大家可以参考相关的书籍。CSS 的属性名也非常丰富，涉及到尺寸、边框、边距、位置、层叠顺序、文字（包括颜色、字体、粗细等等）和背景等等。

CSS 使用 box 模型处理元素的尺寸、边框和边距，下图展示了它们之间的关系。

CSS 的 box 模型

那么 margin 和 padding 有什么区别呢？padding 区域算元素的一部分，所以元素的背景样式同样也会适用于 padding 区域。比如元素背景颜色设定为灰色，padding 区域的背景颜色也会是灰色的，就如上图所示的那样。

需要注意的地方是，虽然 padding 是元素的内边距，也算元素的一部分，但元素的高度和宽度却并不包含 padding 区域。

元素的 margin、padding、height 和 width 的单位一般为 px，即像素，也可以使用百分百的形式，如 50%。如果使用的是百分百的形式，所相对的是此元素指定了绝对尺寸的父系元素。比如下面的例子：

```
<div id="outer" style="width: 500px">
    <div id="inner">
        <div id="content" style="width: 80%">Hello</div>
    </div>
</div>
```

其中 id 为 content 的 div 元素的宽度为 80%，这 80%是相对 id 为 outer 的元素而言的。虽然 content 的直接父系元素为 inner，但是由于 inner 并没有指定宽度，所以会继续向上寻找父系元素，直到找到定义了 width 的元素为止。如果所有的父系元素都没有指定，则这个值是相对于 body 的。

对于元素的位置，默认情况就像报纸排版一样，将一个板块设计好之后，下一个板块会接着排列。这种像瀑布一样的排列方式我们形象地称为 HTML 流（flow）。默认情况下元素的 position 属性值为 static，元素排列在正常的流中。position 属性还有另外的三个值，分别是 absolute、relative 和 fixed。如果元素的位置属性为 absolute，则它的位置是相对于除 static 定位以外的父系元素的，如果没有这样的父系元素，则相对于 body；如果元素的位置属性为 relative，则它的位置是相对于它默认在 HTML 流中位置的；如果元素的位置属性为 fixed，则它的位置是相对于浏览器窗口的。

不同位置属性的元素的定位效果

上图中浅灰色的元素是所有元素的父系元素，它的 position 属性为 relative。深灰色和黑色边框的元素 position 都是默认的 static，所以它们按照 HTML 流的方式依次布局。黑色的元素拥有 absolute 的位置属性，并指定 left 为 10px，top 为 10px，它的定位是相对于浅灰色元素的。

对于 relative 定位，更像是对原有定位的偏移。对于 left 来说，负值向元素的左侧偏移，正值向右侧偏移，right 与其相反；对于 top 来说，负值向元素的上侧偏移，正值向下侧偏移，

bottom 与其相反。需要注意的是，relative 定位所定义的偏移不会影响元素原本在 HTML 流中的位置，下图给出了说明。

relative 定位所定义的偏移不会影响元素原本在 HTML 流中的位置

虽然中间深灰色的元素相对于 HTML 流中的位置产生了偏移，但它原本在 HTML 流中的位置却没有改变，所以并没有影响下面浅灰色元素的位置。

默认情况下，如果元素和元素有重叠的部分，在 HTML 文档中靠后的元素会被显示在上面。但是可以通过 CSS 的 z-index 属性改变层叠顺序，z-index 的值大的元素会显示在 z-index 值小的元素上面。如果一个元素没有被指定 z-index 的值，则在 Chrome 中默认为 0（注意，并非所有浏览器都是这样，比如 IE 默认为负无穷大）。position 属性为 static 的元素（即没有指定 position 属性的元素）z-index 的值会被浏览器忽略。

CSS 还可以定义元素中文字的大小、字体和颜色等，高级的属性还可以定义文字之间的距离、段首缩进、文字阴影等特殊的效果，下面我们主要讲讲对文字大小、字体和颜色的控制。

CSS 使用 font-size 属性控制文字的大小，font-size 的值可以是固定值也可以是百分比。如果是百分比，则相对的是父系的文字尺寸。如果是固定值，常见的单位有 px、pt 和 em，另外还有一些其他的单位，如 in、cm、mm、ex 和 pc。px 最好理解，就是像素，和其他属性一个道理；pt 是印刷界的单位，这个单位与物理尺寸相对应，如果使用 pt 作为单位，则在任何设备上，显示出来的大小都是一样的；em 是个相对的单位，它是相对于元素当前文字尺寸的，比如元素当前文字尺寸为 16px，则 font-size 为 2em，显示出来的文字大小为 32px。另外也可以使用特定的常量来设定文字大小，如 xx-small、medium 和 large 等，smaller 和 larger 则把 font-size 设置为比父元素更小和更大的尺寸。

文字的字体使用 font-family 属性控制，这个属性可以有多个值。浏览器优先使用靠前的值，但如果用户的系统中没有安装指定的字体，则浏览器就会考虑使用后面的值，如果所有指定的字体用户的系统中都没有，则浏览器使用默认字体。对于 Windows 操作系统，中文的默认字体一般是宋体。需要注意的是，如果字体的名称中包含空格，需要用引号将字体名包含，多个值之间用逗号隔开。

文字的颜色使用 color 属性控制，color 的值常见的有三种方式，分别是颜色名、十六进制颜色值和 rgba。除此之外还可以使用 HSL 和 HSLA 格式。颜色名有 black、red 等，网上可以找到一份比较全的颜色名列表。但是能用名称表示的颜色十分有限，多数情况还是需要用颜色值表示。用十六进制的颜色值表示颜色的方法，是一个#符号后面接着 6 位十六进制数值，这 6 位数值每两位为一组，从左至右分别代表红色、绿色和蓝色的强度，#000000 代表黑色，#FFFFFF 代表白色。有时我们会遇到用三位十六进制数值表示颜色的情况，这是颜色值的缩略表示方式，表示每组颜色的十六进制码两位相同，如#ABC 和#AABBCC 表示的颜色相同。rgba 表示方式除了包含红绿蓝三种颜色强度外还包含不透明度。其中前三个数字表示色值，第四个数字表示透明度。表示色值的数字有效值为 0-255 的整数或百分比（百分比也可以表示成小数，如 50%也可以用 0.5 表示），表示透明度的数字有效值为 0-1 的小数。比如 rgba(255, 0, 0, 0.5)表示透明度为 0.5 的红色。

CSS 可以通过 font 属性将多种和文字相关的属性连在一起作为值，这种方式对初学者来说不直观，但对于熟练的人是个节省时间的好办法。

另外不得不再提一下 line-height 这个属性。对于文字来说，它每行所占据的高度是它的大小决定的，默认情况下两行相邻的文字不会重叠，也不会离得太远。下图展示了通过调整 line-height 属性使得两行相邻的文字重叠。

调整 line-height 属性使得两个相邻行的文字重叠

当想让文字在元素中垂直居中时，就可以通过指定 line-height 与 height 相同而达到目的1。

1 也可以使用 vertical-align 属性控制垂直位置。

CSS 还可以控制元素的背景颜色和背景图片。背景颜色通过 background-color 进行控制，值的形式与 color 属性相同。背景图片通过 background-image 进行控制，值为 url(图片位置)。对于背景图片，往往还要结合 background-repeat 和 background-position 使用。前者是控制图片重复的方式，默认是平铺，还可以指定为 repeat-x（横向重复）、repeat-y（纵向重复）和 no-repeat（不重复）。background-position 是控制背景图片的位置，值的形式可以是 top、bottom、left、right 和 center 的结合，比如 top left 为左上角，center left 为左侧中间，如果只指定了一个值，则另一个值默认为 center。也可以是 x% y%的形式，同样是相对于父系元素尺寸的。也可以以像素为单位，如 10px 20px 为距左侧 10 像素，上侧 20 像素。

背景样式还有很多更加丰富的属性，如 background-size，这些更加高级的属性留给感兴趣的读者自行研究吧 :)

最后再强调一次，本节只是对 CSS 的简述，如果想学好 CSS 还应参考相关更加专业的书籍和资料，本节的作用只是避免没有任何基础的读者阅读后面的内容有障碍。

3.2 Browser Actions

Browser Actions 将扩展图标置于 Chrome 浏览器工具栏中，地址栏的右侧。如果声明了 popup 页面，当用户点击图标时，在图标的下侧会打开这个页面 1。同时图标上面还可以附带 badge——一个带有显示有限字符空间的区域——用以显示一些有用的信息，如未读邮件数、当前音乐播放时间等。

1 如果没有足够的空间，会在图标的上侧打开。

下面将对 Browser Actions 中的图标、popup 页面、标题和 badge 做详细介绍。

3.2.1 图标

Browser Actions 可以在 Manifest 中设定一个默认的图标，比如：

```
"browser_action": {
    "default_icon": {
        "19": "images/icon19.png",
        "38": "images/icon38.png"
    }
}
```

一般情况下，Chrome 会选择使用 19 像素的图片显示在工具栏中，但如果用户正在使用视网膜屏幕的计算机，则会选择 38 像素的图片显示。两种尺寸的图片并不是必须都指定的，如果只指定一种尺寸的图片，在另外一种环境下，Chrome 会试图拉伸图片去适应，这样可能会导致图标看上去很难看。

另外，default_icon 也不是必须指定的，如果没有指定，Chrome 将使用一个默认图标。

通过 setIcon 方法可以动态更改扩展的图标，setIcon 的完整方法如下：

chrome.browserAction.setIcon(details, callback)

其中 details 的类型为对象，可以包含三个属性，分别是 imageData、path 和 tabId。

imageData 的值可以是 imageData，也可以是对象。如果是对象，其结构为{size: imageData}，比如{'19': imageData}，这样可以单独更换指定尺寸的图片。imageData 是图片的像素数据，可以通过 HTML 的 canvas 标签获取到。

path 的值可以是字符串，也可以是对象。如果是对象，结构为{size: imagePath}。imagePath 为图片在扩展根目录下的相对位置。

不必同时指定 imageData 和 path，这两个属性都是指定图标所要更换的图片的。

tabId 的值限定了浏览哪个标签页时，图标将被更改。

callback 为回调函数，当 chrome.browserAction.setIcon 方法执行成功后，callback 指定的函数将被运行。此函数没有可接收的回调结果。

下面来编写一个图标不停旋转的扩展。

首先在 Manifest 中定义如下 browser_action：

```
"browser_action": {
    "default_icon": {
        "19": "images/icon19_0.png",
        "38": "images/icon38_0.png"
    },
    "default_title": "Turtle"
}
```

为了让图标动起来，需要一个 background 脚本在后台不停地换图标，这个脚本如下：

```
function chgIcon(index){
    if(index === undefined){
        index = 0;
    }
    else{
        index = index%20;
    }
    chrome.browserAction.setIcon({path: {'19': 'images/icon19_'+index+'.png'}});
    chrome.browserAction.setIcon({path: {'38': 'images/icon38_'+index+'.png'}});
    setTimeout(function(){chgIcon(index+1)},50);
}

chgIcon();
```

为了达到动态旋转的效果，我们需要制作多张图片连续替换，这和 gif 的工作原理是一样的。

如果你不想用这种费力的方法，可以只制作两幅图片，分别对应于 19 像素和 38 像素两个尺寸，在 background 中通过 canvas 绘图来动态改变图片角度，然后输出 imageData。感兴趣的读者可以搜索 HTML5 canvas 了解更多，本文在此不做详细介绍。

本小节编写的扩展源码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/browser_

actions_icon 下载到。

3.2.2 Popup 页面

Popup 页面是当用户点击扩展图标时，展示在图标下面的页面。下图是“网易云音乐（Unofficial）”扩展的 popup 页面。

“网易云音乐（Unofficial）”扩展的 popup 页面

Popup 页面提供了一个简单便捷的 UI 接口。由于有时新窗口会使用户反感，而 popup 页面的设计更像是浏览器的一部分，看上去更加友好，但 popup 页面并不适用于所有情况。由于其在关闭后，就相当于用户关闭了相应的标签页，这个页面不会继续运行。当用户再次打开这个页面时，所有的 DOM 和 js 空间变量都将被重新创建。

所以，popup 页面更多地是用来作为结果的展示，而不是数据的处理。通常情况下，如果需要扩展实时处理数据，而不是只在用户打开时才运行，我们需要创建一个在后端一直运行的页面或者脚本，这可以通过 manifest.json 的 background 域来声明，具体请参考 2.3 节所讲述的内容。而 popup 页面获取后端运行的结果，可以通过扩展内部的通信接口来完成，具体请参考 2.5 节所讲述的内容。

下面来重点讲述一下 popup 页面的设计和需要注意的地方。

Popup 页面是一个扩展与用户交互的窗口，这个窗口设计的好坏直接会影响到用户的使用体验，所以更应加以重视。

首先，popup 页面会根据内容自动显示合适的大小，但还是建议给出页面中 body 元素的尺寸，主要原因有两点：第一，如果不指定页面尺寸，在用户点击图标的瞬间会打开一个很小的窗口，DOM 渲染完毕后页面尺寸才会正常，这个小的细节可能会给用户带来不好的体验；第二，我们也应该设计尺寸可控的页面，这样才能更好地优化布局，同时又不会出现小屏幕设备无法完全显示的情况。考虑到部分小尺寸的上网本，建议 popup 页面的高度最好不要超过 500 像素。当然也可以先给出一个默认的 500 像素高度，之后再通过 js 获取当前设备屏幕的尺寸后，再决定是否需要更改这个高度。但请注意，一个默认的页面尺寸是必要的。

不要尝试模仿 Chrome 的原生 UI。有的开发者为了使自己的扩展看上去更像 Chrome 的一部分，而去刻意模仿 Chrome 的原生 UI，这样做并不值得鼓励。首先我们应该让用户一眼就能看出，哪些部分是 Chrome 自带的功能，哪些部分是来自第三方提供的扩展功能，混淆用户的判断不是一个好主意。另外如果用户使用的不是 Chrome 的默认主题，这种设计看上去将很不协调。

使用带有滚动条的 DIV 容器。指定 body 元素的尺寸后，如果页面内容过长，就会被撑开，导致高度不可控，这可不是我们想要的结果。一个可行的解决办法是通过带有滚动条的 DIV 容器来防止页面被撑开。通常需要设定 DIV 容器的高度为 100%，overflow-y 属性为 auto，这样当不需要滚动条时，DIV 容器将不会显示滚动条。

设定 body 高度为 200 像素后不使用 DIV 容器和使用 DIV 容器的对比

设计一个更好的滚动条样式。每款浏览器都有自己默认的滚动条样式，一般情况下不需要去更改它们——有些浏览器也不允许我们去更改，但对于扩展的 popup 页面则是另外一回事了。首先我们并不是在设计一个网站，而是在设计一款程序，滚动条作为程序的一部分，应尽量和程序的整体风格保持一致；再者，Chrome 默认的滚动条，对尺寸有限的 popup 页面来说显得过于臃肿。“网易云音乐（Unofficial）”扩展就有一个自定义的滚动条样式，这要

比直接使用默认的滚动条强上百倍。关于如何通过 CSS 来自定义 webkit 内核浏览器的滚动条样式，可以通过 http://css-tricks.com/custom-scrollbars-in-webkit/了解更多。

考虑屏蔽右键菜单。如果你是一个追求尽善尽美的开发者，也许不希望用户在你的扩展页面上点击右键时，会出现 Chrome 的默认菜单，取而代之的应该是你自己设计的菜单。你要做的就是屏蔽掉右键，同时通过 DIV 浮动层模拟出一个自己的菜单。但需要注意的是，由于这个模拟的菜单还是在 popup 窗口之内的 DOM 元素，所以它不会像系统菜单那样可以超越页面边界。在设计这个菜单时要考虑到会不会有部分被遮挡，在用户点击鼠标唤出菜单前，请先让一段代码决定这个菜单适当的显示位置。

使用外部引用的脚本。这并不只是 popup 页面需要注意的地方，实际上 Google 在之前较早的某个版本开始，就不再允许 HTML 和 JavaScript 写在一个文件里了。所以要通过<script src="script-path/script-name.js"></script>来引用外部的脚本，而不是将 JavaScript 代码直接写在<script>标签内。

不要在 popup 页面的 js 空间变量中保存数据。由于 popup 页面只在用户点击图标时才会开启，当用户关闭这个页面时就会停止，并没有一个从始至终的实例分配给 popup 页面。所以每当用户打开 popup 页面时，它都是崭新的，之前保存在变量中的数据都会消失。如果需要通过 popup 页面保存用户的数据，可以通过通信将数据交给后台页面处理，或者通过 localStorage 和 chrome.storage 将数据保存在用户的硬盘上。

3.2.3 标题和 badge

将鼠标移至扩展图标上，片刻后所显示的文字就是扩展的标题。

在 Manifest 中，browser_action 的 default_title 属性可以设置扩展的默认标题，比如如下的例子：

```
"browser_action": {
    "default_title": "Extension Title"
}
```

在这个扩展中，默认标题就是“Extension Title”。还可以用 JavaScript 来动态更改扩展的标题，方法如下：

```
chrome.browserAction.setTitle({title: 'This is a new title'});
```

标题不仅仅只是给出扩展的名称，有时它能为用户提供更多的信息。比如一款聊天客户端的标题，可以动态地显示当前登录的帐户信息，如号码和登录状态等。所以如果能合理使用好扩展的标题，会给用户带来更好的体验。

标题我们已经清楚了，那么什么是 badge 呢？我们来看一幅图：

扩展的标题和 badge

上图中，标有“Extension Title”的地方就是扩展标题，而标有“Badg”的地方就是 badge。Badge 是扩展为用户提供有限信息的另外一种方法，这种方法较标题优越的地方是它可以一直显示，其缺点是只能显示大约 4 字节长度的信息，这就是为什么上例中显示的是“Badg”而不是“Badge”。

目前来看使用 badge 比较典型的应用是音乐播放器，它们使用 badge 显示当前音乐播放的时间；另一些内容类的应用，如邮件、微博、RSS 阅读器等，则显示未读条目。无论你打算用 badge 显示何种信息，请记住它只能显示 4 字节长度的内容。对于内容类的扩展，当用

户未读条目足够多时，一般采用的解决方法是显示“999+”。

Badge 目前只能够通过 JavaScript 设定显示的内容，同时 Chrome 还提供了更改 badge 背景的方法。如果不定义 badge 的背景颜色，默认将使用红色，就是上图显示的那样。

下面的代码显示了一个背景颜色为蓝色，内容为“Dog”的 badge：

```
chrome.browserAction.setBadgeBackgroundColor({color: '#0000FF'});
chrome.browserAction.setBadgeText({text: 'Dog'});
```

对于背景颜色的设定，设定值可以是十六进制的字符串颜色码，如#FF0000 代表颜色；也可以是 rgba 格式的数组，但需要注意的是其中的 alpha 变量的取值范围同样为 0-255，这与 CSS 有所区别。

下面的例子使用 rgba 的定义方式，将背景设置为 50%透明度的绿色：

```
chrome.browserAction.setBadgeBackgroundColor({color: [0, 255, 0, 128]});
```

最后需要注意的一点，就是 badge 目前还不支持更改文字的颜色——始终是白色，所以应避免使用浅颜色作为背景。

3.3 右键菜单

当用户在网页中点击鼠标右键后，会唤出一个菜单，在上面有复制、粘贴和翻译等选项，为用户提供快捷便利的功能。Chrome 也将这里开放给了开发者，也就是说我们可以把自己所编写的扩展功能放到右键菜单中。

要将扩展加入到右键菜单中，首先要在 Manifest 的 permissions 域中声明 contextMenus 权限。

```
"permissions": [
    "contextMenus"
]
```

同时还要在 icons 域声明 16 像素尺寸的图标，这样在右键菜单中才会显示出扩展的图标。

```
"icons": {
    "16": "icon16.png"
}
```

Chrome 提供了三种方法操作右键菜单，分别是 create、update 和 remove，对应于创建、更新和移除操作。

通常 create 方法由后台页面来调用，即通过后台页面创建自定义菜单。如果后台页面是 Event Page，通常在 onInstalled 事件中调用 create 方法。

右键菜单提供了 4 种类型，分别是普通菜单、复选菜单、单选菜单和分割线，其中普通菜单还可以有下级菜单。连续相邻的单选菜单会被自动认为是对同一设置的选项，同时单选菜单会自动在两端生成分割线。下面的代码生成了一系列的菜单：

```
chrome.contextMenus.create({
    type: 'normal',
```

```
    title: 'Menu A',
    id: 'a'
});

chrome.contextMenus.create({
    type: 'radio',
    title: 'Menu B',
    id: 'b',
    checked: true
});

chrome.contextMenus.create({
    type: 'radio',
    title: 'Menu C',
    id: 'c'
});

chrome.contextMenus.create({
    type: 'checkbox',
    title: 'Menu D',
    id: 'd',
    checked: true
});

chrome.contextMenus.create({
    type: 'separator'
});

chrome.contextMenus.create({
    type: 'checkbox',
    title: 'Menu E',
    id: 'e'
});

chrome.contextMenus.create({
    type: 'normal',
    title: 'Menu F',
    id: 'f',
    parentId: 'a'
});

chrome.contextMenus.create({
    type: 'normal',
    title: 'Menu G',
```

```
    id: 'g',
    parentId: 'a'
});
```

上面的代码生成的菜单如下图所示。

自定义右键菜单

我们还可以定义自定义的右键菜单在何时显示，比如当用户选择文本时，或者在超级链接上单击右键时。下面的代码定义当用户在超级链接上点击右键时，在菜单中显示“My Menu”菜单：

```
chrome.contextMenus.create({
    type: 'normal',
    title: 'My Menu',
    contexts: ['link']
});
```

contexts 域的值是数组型的，也就是说我们可以定义多种情况下显示自定义菜单，完整的选项包括 all、page、frame、selection、link、editable、image、video、audio 和 launcher，默认情况下为 page，即在所有的页面唤出右键菜单时都显示自定义菜单。其中 launcher 只对 Chrome 应用有效，如果包含 launcher 选项，则当用户在 chrome://apps/或者其他地方的应用图标点击右键，将显示相应的自定义菜单。需要注意的是，all 选项不包括 launcher。

有时我们不仅想在特定的情况下显示自定义菜单，还希望限定 URL，chrome 同样提供了匹配 URL 的选项。documentUrlPatterns 允许限定页面的 URL，比如我们可以限定只在 Google 的网站上显示自定义菜单；targetUrlPatterns 和 documentUrlPatterns 差不多，但它所限定的不是标签的 URL，而是诸如图片、视频和音频等资源的 URL。

如果在创建菜单时，定义了 onclick 域，则菜单被点击后就会调用 onclick 指定的函数。调用的函数会接收到两个参数，分别是点击后的相关信息和当前标签信息。点击后的相关信息包括菜单 id、上级菜单 id、媒体类型（image、video 或 audio）、超级链接目标、媒体 URL、页面 URL、框架 URL、选择的文字、是否可编辑（只针对 text input 和 textarea 等控件）、用户点击前是否被选中和当前是否被选中（只针对 checkbox 或 radio）。完整的信息结构可以通过 http://developer.chrome.com/extensions/contextMenus#type-OnClickData 查看。

update 方法可以动态更改菜单属性，指定需要更改菜单的 id 和所需要更改的属性即可。remove 方法可以删除指定的菜单，removeAll 方法可以删除所有的菜单。

下面我们来创建一个通过右键菜单使用 Google 翻译当前用户所选文本的扩展。我们希望只有当用户选择了文本才显示这个菜单，所以要将 contexts 的值设为 selection。

```
chrome.contextMenus.create({
    type: 'normal',
    title: '使用 Google 翻译……',
    id: 'cn',
    contexts: ['selection']
});
```

下面来编写调用的函数。Google 翻译可以通过 http://translate.google.com.hk/#auto/zh-CN/{翻译文本}调用，所以只需要获取用户所选择的

文本，同时打开这个 URL 就可以了。

```
function translate(info, tab){
    var url = 'http://translate.google.com.hk/#auto/zh-CN/'+info.selectionText ;
    window.open(url, '_blank');
}
```

现在我们把 create 函数补充完整，把调用函数添加进去：

```
chrome.contextMenus.create({
    type: 'normal',
    title: '使用 Google 翻译……',
    contexts: ['selection'],
    id: 'cn',
    onclick: translate
});
```

最后把这段代码写进 background.js 中，让扩展在浏览器启动后自动执行就可以了。

Google 翻译扩展

但我们发现这样无法在菜单中动态显示用户所选择的内容，那么如何动态显示诸如用 Google 翻译“XXX”这样的菜单呢？首先要获取用户所选择的文本，可以通过下面的代码来实现：

```
window.onmouseup = function(){
    var selection = window.getSelection();
    if(selection.anchorOffset != selection.extentOffset){
        //do something
    }
}
```

那么这段代码在 background 中执行会成功吗？显然不能，因为 background 和当前页面并不在一个空间中，所以我们需要用 content_script 来注入脚本，对 content_script 不了解的读者可以参考 2.1 节的内容。content_script 获取到用户所选文字后，就可以通过 2.5 节所讲述的内容，传递给后台页面。

改进后的 Google 翻译扩展

由于改进的部分不是本章的重点，所以就不详细讲解了，大家可以参考前面的章节 1。完整的代码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/google_translate 下载得到。

1 创建菜单时也可以直接使用%s 表示选定的文字。

Chrome 还提供了 onClicked 事件，虽然在 create 方法中可以指定点击时调用的函数，但对于 Event Page 只能通过 onClicked 事件调用函数。Event Page 与一般的 background 类似，但它只按需加载，并不像 background 那样一直驻守后台。

3.4 桌面提醒

之前的章节提到过利用标题和 badge 向用户提供有限的信息，那么如果需要向用户提供更加丰富的信息怎么办呢？Chrome 提供了桌面提醒功能，这个功能可以为用户提供更加丰富的信息。

桌面提醒，图片来自 http://developer.chrome.com

要使用桌面提醒功能，需要在 Manifest 中声明 notifications 权限。

```
"permissions": [
    "notifications"
]
```

创建桌面提醒非常容易，只需指定标题、内容和图片即可。下面的代码生成了标题为“Notification Demo”，内容为“Merry Christmas”，图片为“icon48.png”的桌面提醒窗口。

```
var notification = webkitNotifications.createNotification(
    'icon48.png',
    'Notification Demo',
    'Merry Christmas'
);
```

桌面系统窗口创建之后是不会立刻显示出来的，为了让其显示，还要调用 show 方法：

```
notification.show();
```

需要注意的是，对于要在桌面窗口中显示的图片，必须在 Manifest 的 web_accessible_resources 域中进行声明，否则会出现图片无法打开的情况：

```
"web_accessible_resources": [
    "icon48.png"
]
```

如果希望 images 文件夹下的所有 png 图片都可被显示，可以通过如下声明实现：

```
"web_accessible_resources": [
    "images/*.png"
]
```

桌面提醒窗口提供了四种事件：ondisplay、onerror、onclose 和 onclick。

除了用户主动关闭桌面提醒窗口外，还可以通过 cancel 方法自动关闭。下面的代码可以实现 5 秒后自动关闭窗口的效果。

```
setTimeout(function(){
    notification.cancel();
},5000);
```

由于桌面提醒界面可能将不再支持引入 JS 脚本，桌面提醒窗口与其他界面的通信本节

不进行讲解。

桌面提醒已经被纳入了 W3C 草案，相关信息可以访问 http://dev.chromium.org/developers/design-documents/desktop-notifications/api-specification 查看。

除此之外，也可以通过 Chrome 提供的 chrome.notifications 方法来创建功能更加丰富的提醒框。

3.5 Omnibox

Chrome 和其他浏览器相比一个最大的区别就是地址栏——其实不仅仅是地址栏，而是一个多功能的输入框，Google 将其称为 omnibox（中文为“多功能框”）。我们熟悉的一个功能就是用户可以直接在 omnibox 搜索关键字，Chrome 也将 omnibox 开放给开发者，这使得 omnibox 更加强大。

要使用 omnibox 需要在 Manifest 的 omnibox 域指定 keyword：

```
"omnibox": { "keyword" : "hamster" }
```

同时最好指定一个 16 像素的图标，当用户键入关键字后，这个图标会显示在地址栏的前端。

```
"icons": {
    "16": "icon16.png"
}
```

Chrome 会自动将这个图标渲染成灰度图标，而无需开发者指定一个灰度的图标，由于右键菜单等其他地方也会用到 16 像素的图标，所以应该指定一个彩色的图标。

Omnibox 只提供了一个方法，就是 setDefaultSuggestion，这个方法用来定义默认建议。对于这个默认建议用文字怎么讲解恐怕都不容易讲清楚，那么不妨来看一看设置了默认建议和不设置默认建议的对比：

未设置默认建议和设置了默认建议的对比

上图中左侧为未设置默认建议，显示为“运行 XXX 命令：XXX”，这样显然看起来不够友好。右侧则用更加友好的方式显示查询当前美元价格。

默认建议会在用户输入 keyword 之后一直显示在地址栏下方并且紧挨着地址栏，所以设定一个默认建议是必要的，否则简单地显示“运行 XXX 命令：XXX”会让用户摸不到头脑。

Omnibox 有四种事件：onInputStarted、onInputChanged、onInputEntered 和 onInputCancelled，分别用于监听用户开始输入、输入变化、执行指令和取消输入行为。其中执行指令是指用户敲击回车键或用鼠标点击建议结果。

```
onInputStarted(function(){console.log('Input started.')});
onInputCancelled(function(){console.log('Input cancelled.')});
```

上面的代码执行后，用户开始输入和取消输入时，都会在控制台记录相应日志。下面我们重点来讲一讲另外两个事件。

onInputChanged 事件所承接的只有一个 function 类型的参数，这个 function 参数又有两

个承接参数，第一个参数是字符串型，值为用户当前的输入值，第二个参数还是 function 型，用于返回建议结果，建议的结果为数组型数据，数组中的元素是建议结果对象。

```
chrome.omnibox.onInputChanged.addListener(function(text, suggest){
    suggest([{
        content: text,
        description: 'Search '+text+' in Wikipedia'
    }]);
});
```

onInputEntered 事件同样只有一个 function 类型的承接参数，这个 function 有两个承接参数，第一个是用户输入的值，字符串型，第二个是对结果的建议打开方式，字符串型，但取值范围固定。

```
chrome.omnibox.onInputEntered.addListener(function(text, disposition){
    switch(disposition){
        case 'currentTab': //do something in the current tab
                break;
        case 'newForegroundTab': //do something in a new tab and active it
                break;
        case 'newBackgroundTab': //do something in a new tab
                break;
    }
});
```

下面来制作一款实时查询美元价格的扩展。首先通过异步请求获取 Yahoo 上美元的价格，对这部分不熟悉的读者可以参考前面 2.2 节的内容。获取到数据后我们就要开始编写提供建议的函数了。

```
function updateAmount(amount, exchange){
    amount = Number(amount);
    if(isNaN(amount) || !amount){
        exchange([{
            'content': '$1 = ¥'+price,
            'description': '$1 = ¥'+price
        },{
            'content': '¥1 = $'+(1/price).toFixed(6),
            'description': '¥1 = $'+(1/price).toFixed(6)
        }]);
    }
    else{
        exchange([{
            'content': '$'+amount+' = ¥'+(amount*price).toFixed(2),
            'description': '$'+amount+' = ¥'+(amount*price).toFixed(2)
        },{
            'content': '¥'+amount+' = $'+(amount/price).toFixed(6),
            'description': '¥'+amount+' = $'+(amount/price).toFixed(6)
```

```
            }]);
        }
    }

    var url = 'http://query.yahooapis.com/v1/public/yql?'+
                'q=select%20Rate%20from%20'+
                'yahoo.finance.xchange%20'+
                'where%20pair%20in%20(%22USDCNY%22)&'+
                'env=store://datatables.org/alltableswithkeys&'+
                'format=json';
    var price;

    httpRequest(url, function(r){
        price = JSON.parse(r);
        price = price.query.results.rate.Rate;
        price = Number(price);
    });

    chrome.omnibox.onInputChanged.addListener(updateAmount);
```

大家可以对照前面所讲解的部分来看这段代码，代码中的每个部分都与前面的讲解有所对应。接下来编写用户执行指令时所运行的函数。

```
    function gotoYahoo(text, disposition){
        window.open('http://finance.yahoo.com/q?s=USDCNY=X');
    }

    chrome.omnibox.onInputEntered.addListener(gotoYahoo);
```

此例中并没有理会 disposition 的取值，Chrome 官方也指出 disposition 只是给出结果呈现的建议方式，而非必须遵循的方式，所以是否理会这个值由你自己说了算。

最后就像前面所说的那样，记得设定一个默认的建议，这样会使你的扩展看起来更加友好。

前面讲解默认建议的截图就是这个例子运行的结果，所以在此就不重复贴图了。本例的完整代码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/usd_price 下载，载入扩展后在浏览器地址栏中输入“usd”后按空格键或 Tab 键就可以使用。

3.6 Page Actions

Page Actions 与 Browser Actions 非常类似，除了 Page Actions 没有 badge 外，其他 Browser Actions 所有的方法 Page Actions 都有。另外的区别就是，Page Actions 并不像 Browser Actions 那样一直显示图标，而是可以在特定标签特定情况下显示或隐藏，所以它还具有独有的 show

和 hide 方法。

```
chrome.pageAction.show(integer tabId);
chrome.pageAction.hide(integer tabId);
```

tabId 为标签 id，可以通过 tabs 接口获取，有关 tab 相关的内容将在后面进行讲解。

由于 Page Actions 和 Browser Actions 有大量相似之处，在此就不详细介绍了，还请读者参照前面 3.2 节的内容。

第 4 章　管理你的浏览器

前面介绍了 Chrome 扩展基础和 UI 界面，接下来我们来讲一讲有关管理浏览器的相关内容。本章将涉及到书签、Cookies、历史记录、扩展管理和标签有关的内容，通过本章的内容，你将能够创建功能更加强大的扩展。

4.1　书签

书签这个功能在早期的浏览器就是标配了，浏览器在几十年的更新中，很多功能都已经被新的技术和方法替代，但书签这个功能一直保留至今，可见它对用户的重要程度。

在搜索引擎如此强大的今天，传统的书签已经不再拥有往日的优势，那么我们为什么现在还要保留和讨论这个功能呢？既然互联网索引从早期的人工编排（雅虎早期就是人工编排互联网黄页的）进化到了机器自动抓取并排序，那么书签这个古老的功能也没有理由止步不前。

说到了书签功能的进步，我们不妨来想一想哪些功能是现在书签所具有而曾经没有的。首先是同步，这个一定要放在第一位。当初多少人重新安装系统后望着浏览器空空如也的收藏夹（书签在原来的部分浏览器中也叫收藏夹）捶胸顿足，甚至当时把导出浏览器收藏夹都写入了重装系统的标配步骤中。现在我们再也不担心这个问题了，各大浏览器基本都支持了同步书签的功能，当然前提是你绑定了一个支持同步的账户。其次就是搜索功能，大家发现本来将页面放入书签是方便以后继续查看，但当书签数量变得庞大之后，这种方便也就无从谈起了，所以书签的搜索功能也就出现了。

在前面提到书签发展的进步并非与本节内容无关，这种进步会激发你的创造力，来想一想怎么通过下面将要讲解的浏览器书签管理接口打造更加智能的书签。

Chrome 为开发者提供了添加、分类（书签文件夹）和排序等方法来操作书签，同时也提供了读取书签的方法。

要在扩展中操作书签，需要在 Manifest 中声明 bookmarks 权限：

```
"permissions": [
    "bookmarks"
]
```

在具体讲解操作书签的方法前，先让我们来了解一下书签对象的数据结构。书签对象有

8 个属性，分别是 id、parentId、index、url、title、dateAdded、dateGroupModified 和 children。这 8 个属性并不是每个书签对象都具有的，比如书签分类，即一个文件夹，它就不具有 url 属性。index 属性是这个书签在其父节点中的位置，它的值是从 0 开始的。children 属性值是一个包含若干书签对象的数组。dateAdded 和 dateGroupModified 的值是自 1970 年 1 月 1 日至修改时间所经过的毫秒数。只有 id 和 title 是书签对象必有的属性，其他的属性都是可选的。id 不需要人为干预，它是由 Chrome 管理的。根的 id 为'0'。

创建书签。可以通过 create 方法来创建书签，下面的代码创建了一个标题为“Google”，URL 为“http://www.google.com/”的书签：

```
chrome.bookmarks.create({
    parentId: '1',
    index: 0,
    title: 'Google',
    url: 'http://www.google.com/'
}, function(bookmark){
    console.log(bookmark);
});
```

请注意上面代码的 parentId 属性，'0'为根节点 id，根节点下是不允许创建书签和书签分组的，它的下面默认只有三个书签分组：书签栏、其他书签和移动设备书签，如果创建时不指定 parentId，则所创建的书签会默认加入到其他书签中。create 方法成功后会调用指定的回调函数，回调结果是书签对象。create 方法支持指定的书签属性只有上述代码中所列出的 4 个：parentId、index、title 和 url，其他属性均不支持指定。如果不指定 index，这个书签就将自动添加到相应父节点的尾部。

创建书签分类。创建书签分类的方法和创建书签的方法大致相同，如果创建的书签不包含 url 属性，则 Chrome 自动将其视作为书签分类。

调整书签位置。通过 move 方法可以调整书签的位置，这种调整可以是跨越父节点的，下面的代码将 id 为'16'的书签移动到了 id 为'7'的父节点第 5 个位置：

```
chrome.bookmarks.move('16', {
    parentId:'7',
    index:4
}, function(bookmark){
    console.log(bookmark);
});
```

更新书签。通过 update 方法可以更改书签属性，包括标题和 URL，更新时未指定的属性值将不会更改。下面的代码将将 id 为'16'的书签标题改为'Gmail'，URL 改为'https://mail.google.com/'：

```
chrome.bookmarks.update('16', {
    title: 'Gmail',
    url: 'https://mail.google.com/'
}, function(bookmark){
    console.log(bookmark);
});
```

移除书签。通过 remove 和 removeTree 可以删除书签，remove 方法可以删除书签和空的书签分组，removeTree 可以删除包含书签的书签分组。下面的代码移除了 id 为'16'的书签和 id 为'6'的书签分组。请注意，下面的代码实际上并不能看出删除的是书签还是分组，这要结合用户的实际情况。

```
chrome.bookmarks.remove('16', function(){
    console.log('Bookmark 16 has been removed.');
});

chrome.bookmarks.removeTree('6', function(){
    console.log('Bookmark group 6 has been removed.');
});
```

下面我们来了解一下如何获取用户的书签内容。通过 getTree 方法可以获得用户完整的书签树，但请注意，如果用户的书签树结构过于复杂或内容过多，getTree 方法的效率会很低，而且也会消耗较多的资源，所以请考虑使用后面的方法按需获取部分书签树。下面的代码获取了用户的整个书签树：

```
chrome.bookmarks.getTree(function(bookmarkArray){
    console.log(bookmarkArray);
});
```

需要指出，上面的代码的返回结果依然是一个数组，虽然这个数组永远都只包含一个元素，书签树的根节点。

getChildren 方法可以返回以指定节点为父节点的下一级书签节点，但不包括再下一级的节点，也就是说返回的书签对象不包括 children 属性，无论它是否具有子节点。通过这个方法我们可以一层一层地按需获取用户的书签结构。下面的方法获取了根节点的所有子节点。

```
chrome.bookmarks.getChildren('0', function(bookmarkArray){
    console.log(bookmarkArray);
});
```

getSubTree 方法可以返回自指定节点开始包括当前节点及向下的所有节点，这个方法与 getChildren 的区别是返回值会包含父节点，且没有层级限制，即包含书签对象的 children 属性。下面的代码返回的结果与 getTree 方法返回的结果相同：

```
chrome.bookmarks.getSubTree('0', function(bookmarkArray){
    console.log(bookmarkArray);
});
```

get 方法可以返回指定节点不包含 children 属性的书签对象数组，指定的节点可以是一个或多个。比如下面的代码获取了 id 为'16'和'17'的书签对象：

```
chrome.bookmarks.get(['16', '17'], function(bookmarkArray){
    console.log(bookmarkArray);
});
```

getRecent 方法提供了获取最近添加的多个书签，下面的代码获取了最近添加的 5 个书签：

```
chrome.bookmarks.getRecent(5, function(bookmarkArray){
    console.log(bookmarkArray);
});
```

search 方法可以返回匹配指定条件的书签对象，匹配的条件只能字符串，比如下面的代码会返回所有标题或 URL 中包含 google 的书签：

```
chrome.bookmarks.search('google', function(bookmarkArray){
    console.log(bookmarkArray);
});
```

最后我们来看一看书签的事件，Chrome 提供了多个事件来监控书签操作行为。

onCreated 事件用以监控书签的创建行为：

```
chrome.bookmarks.onCreated.addListener(function(bookmark){
    console.log(bookmark);
});
```

onRemoved 事件用以监控书签的移除行为：

```
chrome.bookmarks.onRemoved.addListener(function(id, removeInfo){
    console.log('Bookmark '+id+' has been removed:');
    console.log(removeInfo);
});
```

removeInfo 包含 parentId 和 index 属性，与所删除书签对象之前的属性相对应。

onChanged 事件用以监控书签的更新行为：

```
chrome.bookmarks.onChanged.addListener(function(id, changeInfo){
    console.log('Bookmark '+id+' has been changed:');
    console.log(changeInfo);
});
```

changeInfo 包含 title 和 url 属性，与所更改书签对象更新后的属性相对应。

onMoved 事件用以监控书签的移动行为：

```
chrome.bookmarks.onMoved.addListener(function(id, moveInfo){
    console.log('Bookmark '+id+' has been moved:');
    console.log(moveInfo);
});
```

moveInfo 包含 parentId、index、oldParentId 和 oldIndex 属性，与所移动书签对象移动前后的属性相对应。

onChildrenReordered 事件用以监控一个书签分组下的更改子节点顺序的行为：

```
chrome.bookmarks.onChildrenReordered.addListener(function(id, reorderInfo){
    console.log('Bookmark '+id+' has a new children order:');
    console.log(reorderInfo);
});
```

reorderInfo 是包含顺序更改后子节点 id 的数组。

onImportBegan 和 onImportEnded 事件分别用以监控导入书签开始和结束的行为：

```
onImportBegan(function(){
    console.log('Bookmark import began.');
});

onImportEnded(function(){
    console.log('Bookmark import ended.');
});
```

请注意，如果检测到浏览器正在导入书签（onImportBegan 事件被触发但 onImportEnded 事件还未被触发），应当忽略 onCreated 事件，但其他的操作可以被立即执行。

以上就是书签相关的全部内容，读者可以结合之前的内容创建更加智能方便的书签管理扩展。比如可以直接通过地址栏搜索书签，或者当用户使用 Google 搜索时将匹配到的书签结果添加到 Google 搜索结果的前端，类似 Google 广告推广那样。这些新奇的点子就交给读者们自行实现吧，在此就不给出实例了。

4.2 Cookies

Cookies 是浏览器记录在本地的用户数据，如用户的登录信息。Chrome 为扩展提供了 Cookies API 用以管理 Cookies。

要管理 Cookies，需要在 Manifest 中声明 cookies 权限，同时也要声明所需管理 Cookies 所在的域：

```
"permissions": [
    "cookies",
    "*://*.google.com"
]
```

如果想要管理所有的 Cookies 可以声明如下权限：

```
"permissions": [
    "cookies",
    "<all_urls>"
]
```

请注意，除非必要，否则请不要如此声明权限，这会提示此扩展可以访问所有网络资源，给用户带来不安。

Chrome 定义的 Cookie 对象包含如下属性：name（名称）、value（值）、domain（域）、hostOnly（是否只允许完全匹配 domain 的请求访问）、path（路径）、secure（是否只允许安全连接调用）、httpOnly（是否禁止客户端调用）、session（是否是 session cookie）、expirationDate（过期时间）和 storeId（包含此 cookie 的 cookie store 的 id）。

读 Cookies。Chrome 提供了 get 和 getAll 两个方法读取 Cookies，get 方法可以读取指定 name、url 和 storeId 的 Cookie，其中 storeId 可以不指定，但是 name 和 url 必须指定。如果

在同一 URL 中包含多个 name 相同的 Cookies，则会返回 path 最长的那个，如果有多个 Cookies 的 path 长度相同，则返回创建最早的那个。

```
chrome.cookies.get({
    url: 'https://github.com',
    name: 'dotcom_user'
}, function(cookie){
    console.log(cookie.value);
});
```

这里需要注意一点，如果 cookie 的 secure 属性值为 true，那么通过 get 获取时 url 应该是 https 协议。

getAll 方法与 get 方法不同，它可以获取所有符合条件的 Cookies，支持的匹配条件包括 url、name、domain、path、secure、session 和 storeId 中的任意一个或多个，如果一个都不指定，则返回所有此扩展有权访问到的 Cookies。比如下面的代码就可以获取到所有可以读取的 Cookies：

```
chrome.cookies.getAll({}, function(cookies){
    console.log(cookies);
});
```

设置 Cookie。set 方法可以设置 Cookie：

```
chrome.cookies.set({
    'url':'http://github.com/test_cookie',
    'name':'TEST',
    'value':'foo',
    'secure':false,
    'httpOnly':false
}, function(cookie){
    console.log(cookie);
});
```

如果创建成功，则回调函数会获取到创建后的 cookie 对象，否则会得到 null。url 是必须指定的，其他的属性可选。另外扩展对 URL 必须有访问权限，否则会设置失败。如果不指定 expirationDate 属性，则所创建的 Cookie 将在浏览器关闭后被删除。

设置 Cookies

删除 Cookie。remove 方法可以删除指定 url、name 和 storeId 的 Cookie。

```
chrome.cookies.remove({
    url: 'http://www.google.com',
    name: '_ga'
}, function(result){
    console.log(result);
});
```

同样，扩展首先要具有对 URL 的访问权限，否则删除操作会失败。

除非你清楚在做什么，不要轻易删除用户的 Cookies，否则你可能会收到大量用户抱怨气愤的邮件。

getAllCookieStores 方法用来获取全部的 cookie store，cookie store 包含一个 id 属性和一个 tabIds 属性，id 的属性值为这个 cookie store 的 id，tabIds 为包含共享这个 cookie store 所有 tab 的 id 的数组。有关 tab 的内容将在后面的章节讲解。

onChanged 事件用来监控 cookie 的设置和删除行为：

```
chrome.cookies.onChanged.addListener(function(changeInfo){
    console.log(changeInfo);
});
```

changeInfo 包含三个属性：removed，是否是删除行为；cookie，被设置或删除的 cookie 对象；cause，Cookie 变化的原因，可能的值包括 evicted、expired、explicit、expired_overwrite 和 overwrite。

再次提醒，Cookies 是用户的敏感数据，在进行操作时一定倍加小心，并要让用户有知情权，必要时一定要先得到用户的确认。

4.3 历史

历史用于记录用户访问过页面的信息。与书签一样，历史也是浏览器很早就具有的功能，对用户来说也是一个很重要的功能。Chrome 提供了 history 接口，允许扩展对用户的历史进行管理。

要使用 history 接口，需要在 Manifest 中声明 history 权限：

```
"permissions": [
    "history"
]
```

管理历史的方法包括 search、getVisits、addUrl、deleteUrl、deleteRange 和 deleteAll。其中 search 和 getVisits 用于读取历史，addUrl 用于添加历史，deleteUrl、deleteRange 和 deleteAll 用于删除历史。

读取历史。Chrome 提供了 search 和 getVisits 两种方法读取历史。通过 search 方法可以读取匹配指定文字，指定时间区间，指定条目的历史结果。

```
chrome.history.search({
    text: 'Google',
    startTime: new Date().getTime()-24*3600*1000,
    endTime: new Date().getTime(),
    maxResults: 20
}, function(historyItemArray){
    console.log(historyItemArray);
});
```

上述代码会返回最近 24 小时内匹配“Google”的 20 条历史结果。startTime 和 endTime 都是距 1970 年 1 月 1 日的毫秒数。返回结果是包含多个 historyItem 对象的数组，historyItem

对象包含6个属性，分别是id、url、title、lastVisitTime、visitCount和typedCount，其中typedCount是用户通过在地址栏键入访问此历史的次数。若不指定 text 属性，则返回全部历史结果。

getVisits 方法可以获取指定 URL 的访问结果。必须指定完整的 URL，返回的结果会绝对匹配指定的 URL，也就是说，如果指定'http://www.google.com/'，返回的结果不会包含'http://www.google.com/a/'的内容。不要忘记 http://，这也是不可省略的。

```
chrome.history.getVisits(
    url: 'http://www.google.com/'
}, function(visitItemArray){
    console.log(visitItemArray);
});
```

返回的结果是包含多个 visitItem 对象的数组，visitItem 对象包含 5 个属性，分别是 id、visitId、visitTime、referringVisitId 和 transition。其中 id 为与指定 URL 匹配的对象的 id，对于匹配同一 URL 的对象拥有相同的 id，visitId 是这个访问结果的 id，visitId 是唯一的。visitTime 同样是毫秒数。transition 是此访问记录打开的方式，具体解释如下。

Chrome 对每一个访问记录都详细地归类了打开方式，用 transition 属性记录。打开方式一共分为 11 种，这看起来确实会让人有一些头疼。比较常见的有四种，分别为 link、typed、reload 和 form_submit。link 是用户通过超级链接打开的方式，typed 是用户通过在地址栏中输入网址打开的方式，reload 是用户通过刷新（包括恢复关闭的标签）打开的方式，form_submit 是通过提交表单打开的方式（通过脚本提交表单的情况不算此方式）。

与浏览器 UI 和设置相关的有两种，分别为 auto_bookmark 和 auto_toplevel。auto_bookmark 是通过浏览器 UI 中的建议打开的方式——比如通过菜单等。auto_toplevel 为浏览器设置中默认打开的方式，比如浏览器的主页，或者是通过命令行启动时附带的参数。

嵌入式框架相关的有两个，auto_subframe 和 manual_subframe，其中 auto_subframe 为自动加载的嵌入式框架打开的方式，很多广告都是这样的打开方式——很多用户并不知道其实那些广告是在一个独立的页面中。manual_subframe 则是用户手动加载的嵌入式框架打开的方式，比如用户操作商品菜单查看不同款式商品页面，就是手动加载嵌入式框架。

最后还有三种是和 omnibox 搜索建议相关的，分别为 generated、keyword 和 keyword_generated。generated 为通过 omnibox 给出搜索建议打开的方式，所打开的页面通常为搜索引擎的结果界面。keyword 和 keyword_generated 都是通过用户在地址栏中输入的关键字生成的 URL 访问的方式，但其 URL 并不是默认搜索引擎生成的（否则就是 generated 了）。

添加历史。addUrl 方法可以将特定的 url 以当前时间为访问时间，添加至历史中。

```
chrome.history.addUrl({
    url: 'http://twitter.com'
}, function(){
    console.log('Twitter has been added to history.');
});
```

删除历史。deleteUrl 可以删除指定 URL 的历史，deleteRange 可以删除指定时间段的历史，deleteAll 可以删除全部历史。

```
chrome.history.deleteUrl({
    url: 'http://www.google.com'
}, function(){
```

```
        console.log('Google has been deleted from history.');
});

chrome.history.deleteRange({
        startTime: new Date().getTime()-24*3600*1000,
        endTime: new Date().getTime()
}, function(){
        console.log('History in past 24 hours has been deleted.');
});

chrome.history.deleteAll(function(){
        console.log('All history has been deleted.');
});
```

Chrome 提供两个事件，onVisited 和 onVisitRemoved，分别监听用户访问历史和历史被删除的事件。

```
chrome.history.onVisited.addListener(function(historyItem){
        console.log(historyItem);
});

chrome.history.onVisitRemoved.addListener(function(removedObject){
        console.log(removedObject);
});
```

对于 onVisitRemoved 事件，返回的 removedObject 结果包含两个属性，allHistory 和 urls。其中 urls 属性包含所有被删除历史的 URL。allHistory 为布尔型，如果所有历史均被删除，allHistory 的值为 ture，同时 urls 的值会为一个空数组。

历史和 cookies 一样都是用户的敏感数据，进行操作时应让用户有知情权，尤其是要将用户历史数据与第三方共享时（包括开发者自己的服务器），一定要先得到用户的同意，并且要让用户得知哪些数据会被使用。

4.4 管理扩展与应用

除了通过 chrome://extensions/管理 Chrome 扩展和应用外，也可以通过 Chrome 的 management 接口管理。management 接口可以获取用户已安装的扩展和应用信息，同时还可以卸载和禁用它们。通过 management 接口可以编写出智能管理扩展和应用的程序。

要使用 management 接口，需要在 Manifest 中声明 management 权限：

```
"permissions": [
        "management"
]
```

读取用户已安装扩展和应用的信息。Management 提供了两个方法获取用户已安装扩展

应用的信息，分别是 getAll 和 get。

```
chrome.management.getAll(function(exInfoArray){
    console.log(exInfoArray);
});

chrome.management.get(exId, function(exInfo){
    console.log(exInfo);
});
```

exInfo 是扩展信息对象，其结构如下：

```
{
    id: 扩展 id,
    name: 扩展名称,
    shortName: 扩展短名称,
    description: 扩展描述,
    version: 扩展版本,
    mayDisable: 是否可被用户卸载或禁用,
    enabled: 是否已启用,
    disabledReason: 扩展被禁用原因,
    type: 类型,
    appLaunchUrl: 启动 url,
    homepageUrl: 主页 url,
    updateUrl: 更新 url,
    offlineEnabled: 离线是否可用,
    optionsUrl: 选项页面 url,
    icons: [{
        size: 图片尺寸,
        url: 图片 URL
    }],
    permissions: 扩展权限,
    hostPermissions: 扩展有权限访问的 host,
    installType: 扩展被安装的方式
}
```

其中 type 属性的可能值为 extension、hosted_app、packaged_app、legacy_packaged_app 或 theme。installType 可能的值为 admin（管理员安装）、development（载入未打包的扩展）、normal（通过 crx 正常安装）、sideload（第三方程序安装）或 other（其他）。

获取权限警告。getPermissionWarningsById 和 getPermissionWarningsByManifest 方法可以获取权限警告，这些警告与用户安装扩展时网上应用商店弹出的警告类似。

```
chrome.management.getPermissionWarningsById(exId, function(permissionWarningArray){
    console.log(permissionWarningArray);
});

getPermissionWarningsByManifest(exManifest, function(permissionWarningArray){
```

```
    console.log(permissionWarningArray);
});
```

上述代码中，exManifest 是字符串型的，不是对象型的。

启用、禁用、卸载扩展和启动应用。setEnabled 方法可以启用或禁用扩展应用，如果一个扩展或应用被禁用，它的后台页面不会运行。

```
chrome.management.setEnabled(exId, enabled, function(){
    if(enabled){
        console.log('Extension '+exId+' has been enabled.');
    }
    else{
        console.log('Extension '+exId+' has been disabled.');
    }
});
```

卸载扩展有两种方法，uninstall 可以卸载指定 id 的扩展，uninstallSelf 可以卸载扩展自身且无需请求 management 权限。

```
uninstall(exId, {
    showConfirmDialog: true
}, function(){
    console.log('Extension '+exId+' has been uninstalled.');
});

uninstallSelf({
    showConfirmDialog: true
}, function(){
    console.log('This extension has been uninstalled.');
});
```

如果不希望在卸载前显示确认窗口，可以将 showConfirmDialog 的值设为 false。

通过 launchApp 方法启动应用：

```
chrome.management.launchApp(exId, function(){
    console.log('App '+exId+' has been launched.');
});
```

management 接口提供了四种事件，onInstalled、onUninstalled、onEnabled 和 onDisabled，分别用于监听安装、卸载、启用和禁用扩展应用。

```
chrome.management.onInstalled.addListener(function(exInfo){
    console.log('Extension '+exInfo.id+' has been installed.')
});

chrome.management.onUninstalled.addListener(function(exId){
    console.log('Extension '+exId+' has been uninstalled.');
});
```

```
chrome.management.onEnabled.addListener(function(exInfo){
    console.log('Extension '+exInfo.id+' has been enabled.');
});

chrome.management.onDisabled.addListener(function(exInfo){
    console.log('Extension '+exInfo.id+' has been disabled.');
});
```

本节讲解了管理扩展和应用的接口内容，看起来有些枯燥，但如果使用恰当设计合理，可以编写出让用户很 feel 的扩展。

4.5 标签

前面的章节中，多次提到了标签，本节将详细讲解对标签信息获取和操作的内容。在开始介绍之前，先让我们来看一下标签对象的结构：

```
{
    id: 标签 id,
    index: 标签在窗口中的位置，以 0 开始,
    windowId: 标签所在窗口的 id,
    openerTabId: 打开此标签的标签 id,
    highlighted: 是否被高亮显示,
    active: 是否是活动的,
    pinned: 是否被固定,
    url: 标签的 URL,
    title: 标签的标题,
    favIconUrl: 标签 favicon 的 URL,
    status :标签状态，loading 或 complete,
    incognito: 是否在隐身窗口中,
    width: 宽度,
    height: 高度,
    sessionId: 用于 sessions API 的唯一 id
}
```

Chrome 通过 tabs 方法提供了管理标签的方法与监听标签行为的事件，大多数方法与事件是无需声明特殊权限的，但有关标签的 url、title 和 favIconUrl 的操作（包括读取），都需要声明 tabs 权限。

```
"permissions": [
    "tabs"
]
```

获取标签信息。Chrome 提供了三种获取标签信息的方法，分别是 get、getCurrent 和 query。

get 方法可以获取到指定 id 的标签，getCurrent 则获取运行的脚本本身所在的标签，query 可以获取所有符合指定条件的标签。

```
chrome.tabs.get(tabId, function(tab){
    console.log(tab);
});

chrome.tabs.getCurrent(function(tab){
    console.log(tab);
});
```

query 方法可以指定的匹配条件如下：

```
{
    active: 是否是活动的,
    pinned: 是否被固定,
    highlighted: 是否正被高亮显示,
    currentWindow: 是否在当前窗口,
    lastFocusedWindow: 是否是上一次选中的窗口,
    status: 状态，loading 或 complete,
    title: 标题,
    url: 所打开的 url,
    windowId: 所在窗口的 id,
    windowType: 窗口类型，normal、popup、panel 或 app,
    index: 窗口中的位置
}
```

下面的代码获取了所有在窗口中活动的标签：

```
chrome.tabs.query({
    active: true
}, function(tabArray){
    console.log(tabArray);
});
```

创建标签。创建标签与在浏览器中打开新的标签行为类似，但可以指定更加丰富的信息，如 URL、窗口中的位置和活动状态等。

```
chrome.tabs.create({
    windowId: wId,
    index: 0,
    url: 'http://www.google.com',
    active: true,
    pinned: false,
    openerTabId: tId
}, function(tab){
    console.log(tab);
});
```

其中 wId 是创建标签所在窗口的 id，如果不指定，则默认在当前窗口中打开。tId 是打开此标签的标签 id，可以不指定，但如果指定，那么所创建的标签必须与这个标签在同一窗口中。

除了用 create 方法，还可以使用 duplicate 方法“复制”指定标签：

```
chrome.tabs.duplicate(tabId, function(tab){
    console.log(tab);
});
```

更新标签。通过 update 方法可以更新标签的属性：

```
chrome.tabs.update(tabId, {
    url: 'http://www.google.com'
}, function(tab){
   console.log(tab);
});
```

更新标签时也可以不指定 tabId，如果不指定，默认会更改当前窗口的活动标签。需要指出，直到 31.0.1650.63 m，更新 highlighted 属性为 true 后，标签 active 属性也会被指定为 true，所以如果只是想将某个标签高亮以引起用户的注意，需要先记录当前的标签 id，更新后再将这个标签的 active 属性改回 true。这个 bug 在之后的版本也许会被修正。

移动标签。move 方法可以将指定的一个或多个标签移动到指定位置：

```
chrome.tabs.move(tabIds, {
    'windowId':wId,
    'index':0
}, function(tabs){
    console.log(tabs);
});
```

其中 tabIds 可以是一个数字型的标签 id，也可以是一个包含多个标签 id 的数组。返回的 tabs 可能是标签对象也可能是包含多个标签对象的数组。如果指定的 index 为-1，会将标签移动到指定窗口的最后面。

重载标签。reload 方法可以重载指定标签，同时还可以指定是否跳过缓存（强制刷新）：

```
chrome.tabs.reload(tabId, {
    bypassCache: true
}, function(){
    console.log('The tab has been reloaded.');
});
```

浏览器通常会对一些静态资源进行缓存，JavaScript 中的 location.reload()方法通常无法实现强制刷新，此时上面的方法就会很好地解决这个问题。

移除标签。通过 remove 方法可以关闭一个或多个标签：

```
chrome.tabs.remove(tabIds, function(){
    console.log('The tabs has been closed.');
});
```

其中 tabIds 可以是一个数字型的标签 id，也可以是一个包含多个标签 id 的数组。

获取当前标签页面的显示语言。有时可能需要针对用户浏览内容语言的不同，采用不同的处理方法。比如翻译扩展就要根据不同的语言决定是否提示用户进行翻译。

```
chrome.tabs.detectLanguage(tabId, function(lang){
    console.log('The primary language of the tab is '+lang);
});
```

如果不指定 tabId，则返回当前窗口当前标签的语言。

获取指定窗口活动标签可见部分的截图。Chrome 提供了截取指定窗口活动标签页面为图片的接口：

```
chrome.tabs.captureVisibleTab(windowId, {
    format: 'jpeg',
    quality: 50
}, function(dataUrl){
    window.open(dataUrl, 'tabCapture');
});
```

其中 format 还支持 png，如果指定为 png，则 quality 属性会被忽略。如果指定 jpeg 格式，quality 的取值范围为 0-100，数值越高，图片质量越好，体积也越大。扩展只有声明 activeTab 或<all_url>权限能获取到活动标签的截图：

```
"permissions": [
    "activeTab"
]
```

注入 JS 和 CSS。之前我们接触过 content_scripts，它可以向匹配条件的页面注入 JS 和 CSS，但是却无法向用户指定的标签注入。通过 executeScript 和 insertCSS 可以做到向指定的标签注入脚本。

```
chrome.tabs.executeScript(tabId, {
    file: 'js/ex.js',
    allFrames: true,
    runAt: 'document_start'
}, function(resultArray){
    console.log(resultArray);
});
```

也可以直接注入代码：

```
chrome.tabs.executeScript(tabId, {
    code: 'document.body.style.backgroundColor="red"',
    allFrames: true,
    runAt: 'document_start'
}, function(resultArray){
    console.log(resultArray);
});
```

向指定的标签注入 CSS：

```
chrome.tabs.insertCSS(tabId, {
    file: 'css/insert.css',
    allFrames: false,
    runAt: 'document_start'
}, function(){
    console.log('The css has been inserted.');
});
```

插入 CSS 也可以指定具体代码。

executeScript 和 insertCSS 方法中 runAt 的值可以是'document_start'、'document_end'或'document_idle'。

与指定标签中的内容脚本（content script）通信。前面章节介绍过扩展页面间的通信，我们也可以与指定的标签通信，方法如下：

```
chrome.tabs.sendMessage(tabId, message, function(response){
    console.log(response);
});
```

请注意，后台页面主动与 content_scripts 通信需要使用 chrome.tabs.sendMessage 方法 1。

1 chrome.tabs.executeScript 方法也可以实现后台页面与内容脚本的通信，但更强调是后台页面向标签页注入脚本。

由于标签的操作行为比较多，所以相应的监视事件也很多。监控标签行为的事件包含 onCreated、onUpdated、onMoved、onActivated、onHighlighted、onDetached、onAttached、onRemoved 和 onReplaced。

大部分事件都比较好理解，下面重点讲一讲不易理解的事件。onHighlighted 是当标签被高亮显示时所触发的事件，active 和 highlight 是有区别的，active 是指标签在当前窗口中正被显示，highlight 只是标签的颜色被显示成了白色——如果此标签没有被选中正常情况下是浅灰色。onDetached 是当标签脱离窗口时所触发的事件，导致此事件触发的原因是用户在两个不同的窗口直接拖拽标签。onAttached 是标签附着到窗口上时所触发的事件，同样是用户在两个不同的窗口直接拖拽标签导致的。onReplaced 是当标签被其他标签替换时触发的事件 2。

2 要解释清楚 onReplaced 就不得不提一下即搜即得和预呈现（Instant search, Prerendering）。例如默认搜索引擎为 Google，启用了即搜即得，网络条件也足够好，在打开的另一个网页地址栏中开始输入关键字并且即时出现结果时，此时按下回车键，当前标签页就会被 Google 搜索结果替换，产生 onReplaced 事件。如果扩展程序通过 tabId 追踪标签页的话就必须处理该事件。

```
chrome.tabs.onCreated.addListener(function(tab){
    console.log(tab);
});

chrome.tabs.onUpdated.addListener(function(tabId, changeInfo, tab){
    console.log('Tab '+tabId+' has been changed with these options:');
    console.log(changeInfo);
```

```
});

chrome.tabs.onMoved.addListener(function(tabId, moveInfo){
    console.log('Tab '+tabId+' has been moved:');
    console.log(moveInfo);
});

chrome.tabs.onActivated.addListener(function(activeInfo){
    console.log('Tab '+activeInfo.tabId+' in window '+activeInfo.windowId+' is active now.');
});

chrome.tabs.onHighlighted.addListener(function(highlightInfo){
    console.log('Tab '+activeInfo.tabId+' in window '+activeInfo.windowId+' is highlighted now.');
});

chrome.tabs.onDetached.addListener(function(tabId, detachInfo){
    console.log('Tab '+tabId+' in window '+detachInfo.oldWindowId+' at position '+detachInfo.oldPosition+' has been detached.');
});

chrome.tabs.onAttached.addListener(function(tabId, attachInfo){
    console.log('Tab '+tabId+' has been attached to window '+detachInfo.newWindowId+' at position '+detachInfo.newPosition+' .');
});

chrome.tabs.onRemoved.addListener(function(tabId, removeInfo){
    console.log('Tab '+tabId+' in window '+removeInfo.windowId+', and the window is '+(removeInfo.isWindowClosing?'closed.':'open.'));
});

chrome.tabs.onReplaced.addListener(function(addedTabId, removedTabId){
    console.log('Tab '+removedTabId+' has been replaced by tab '+addedTabId+'.');
);
```

通过标签接口，扩展可以更灵活地处理不同标签。虽然标签涉及到的内容很多，但常用的部分很有限，读者在阅读此节时，不妨先把精力重点放在那些常用易懂的方法事件上，对于剩下的部分随用随查即可。

4.6 Override Pages

Chrome 不仅提供了管理书签、历史和标签的接口，还支持用自定义的页面替换 Chrome

相应默认的页面，这就是 override pages。目前支持替换的页面包含 Chrome 的书签页面、历史记录和新标签页面。

使用 override pages 很简单，只需在 Manifest 中进行声明即可（一个扩展只能替换一个页面）：

```
"chrome_url_overrides" : {
    "bookmarks": "bookmarks.html"
}

"chrome_url_overrides" : {
    "history": "history.html"
}

"chrome_url_overrides" : {
    "newtab": "newtab.html"
}
```

把上面页面的地址替换成你自己的就可以了。

Google 官方对 override pages 给出了几点建议（以下内容翻译来自 https://crxdoc-zh.appspot.com/extensions/override）：

使您的页面又快又小。

用户期望内置的浏览器页面能够立即打开。请避免做任何可能花较长时间的事情，例如，避免同步地获取网络或数据库资源。

在您的页面中包含标题。

否则用户可能会看到页面的 URL，会令人感到疑惑。这是一个指定标题的例子：新标签页

不要假定页面具有键盘焦点。

当用户创建新标签页时总是地址栏先获得焦点。

不要试着模仿默认的“打开新的标签页”页面。

用于创建与默认的“打开新的标签页”页面类似（具有最常访问的网站、最近关闭的标签页、提示、主题背景图像等等）的修改版本所需的 API 还不存在。在出现那些 API 之前您还是最好还是考虑一些完全不同的新想法。

第 5 章　部分高级 API

在前面的章节，我们已经接触到了 Chrome 扩展中常用的大多数 API，本章将挑选部分较为常用的高级 API 进行讲解，以便有更高要求的读者阅读。

5.1 下载

Chrome 提供了 downloads API，扩展可以通过此 API 管理浏览器的下载功能，包括暂停、搜索和取消等。

相对于管理下载，更令人关注的是创建下载的功能。Chrome 应用市场中之前包括很多下载页面所有图片等类似功能的扩展，大多数是将图片包含在一个网页中让用户另存为，或者是列出所有 URL 让用户自行下载。这样做明显不友好，Chrome 处于早期版本时，开发者对开放下载功能的呼声也越来越高。所以本节将重点讲解如何让扩展通过 downloads 接口创建下载，有关进一步管理下载行为的内容请感兴趣的读者自行阅读。完整有关 downloads 接口的官方文档可以通过 http://developer.chrome.com/extensions/downloads 阅读。

扩展使用 downloads 接口需要在 Manifest 文件中声明 downloads 权限：

```
"permissions": [
    "downloads"
]
```

创建下载可以通过 downloads 中的 download 方法实现。download 方法包含两个参数，第一个是有关下载的属性对象，包括 URL、保存位置、文件名等信息，第二个是创建成功后的回调函数。

```
chrome.downloads.download(options, callback);
```

其中 options 的完整结构如下：

```
{
    url: 下载文件的 url,
    filename: 保存的文件名,
    conflictAction: 重名文件的处理方式,
    saveAs: 是否弹出另存为窗口,
    method: 请求方式（POST 或 GET）,
    headers: 自定义 header 数组,
    body: POST 的数据
}
```

其中 conflictAction 的取值只能是 uniquify（在文件名后添加带括号的序号保证文件名唯一）、overwrite（覆盖）和 prompt（给出提示让用户自行决定重命名或者覆盖）。

filename 可以是单纯的文件名，如'foo.txt'；也可以带有相对路径，如'mypath/foo.txt'。但不可以是绝对路径，或是一个目录，也不可以在路径中包含上级路径'..'。如这三种情况都是非法的：'/mypath/foo.txt'、'mypath/'、'../mypath/foo.txt'。

如果给定了 filename，同时 saveAs 属性为 true，则弹出的另存为对话框中，文件名一栏的默认值会被设为 filename 指定的值。

下面让我们来一起编写一个下载当前页面所有图片的扩展。

这个扩展我准备设计成用户在页面点击右键时，菜单中包含一个下载所有图片的选项。这个过程首先要在右键菜单中创建一个选项，我们需要一个 background 脚本。因为要获取当前页面的图片元素，所以我们要向当前标签页注入脚本。这几点分析清楚之后，我们就可

以开始了。

首先创建 manifest.json。

```
{
    "manifest_version": 2,
    "name": "Save all images",
    "version": "1.0",
    "description": "Save all images in current tab",
    "background": {
        "scripts": ["background.js"],
        "persistent": false
    },
    "permissions": [
        "activeTab",
        "contextMenus",
        "downloads"
    ]
}
```

下面编写 background.js 文件，这个文件用来创建右键菜单，并在用户点击菜单后向当前标签页注入脚本，最后还要完成下载的行为。

```
chrome.runtime.onInstalled.addListener(function(){
    chrome.contextMenus.create({
        'id':'saveall',
        'type':'normal',
        'title':'保存所有图片',
    });
});

chrome.contextMenus.onClicked.addListener(function(info, tab){
    if(info.menuItemId == 'saveall'){
        chrome.tabs.executeScript(tab.id, {file: 'main.js'}, function(results){
            if (results && results[0] && results[0].length){
                results[0].forEach(function(url) {
                    chrome.downloads.download({
                        url: url,
                        conflictAction: 'uniquify',
                        saveAs: false
                    });
                });
            }
        });
    }
});
```

最后来编写注入脚本，main.js。

```
[].map.call(document.getElementsByTagName('img'), function(img){
    return img.src;
});
```

至此，通过右键菜单下载所有图片的扩展就编写完成了。

本例中没有指定扩展的图标，但在成熟的产品中，自定义右键菜单时，应当指定一个16像素的图标。

本节所涉及到的代码可以通过https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/save_all_images下载得到。

5.2 网络请求

Chrome提供了较为完整的方法供扩展程序分析、阻断及更改网络请求，同时也提供了一系列较为全面的监听事件以监听整个网络请求生命周期的各个阶段。网络请求的整个生命周期所触发事件的时间顺序如下图所示。

网络请求的生命周期，图片来自developer.chrome.com

要对网络请求进行操作，需要在Manifest中声明webRequest权限以及相关被操作的URL。如需要阻止网络请求，需要声明webRequestBlocking权限。

```
"permissions": [
    "webRequest",
    "webRequestBlocking",
    "*://*.google.com/"
]
```

上面的权限声明表示此扩展可以对浏览器向Google发起的网络请求进行更改。webRequest接口无法在Event Page中使用。

目前对于网络请求，比较实用的功能包括阻断连接、更改header和重定向。

下面的代码阻断了所有向bad.example.com的连接：

```
chrome.webRequest.onBeforeRequest.addListener(
    function(details){
        return {cancel: true};
    },
    {
        urls: [
            "*://bad.example.com/*"
        ]
    },
    [
        "blocking"
```

```
        ]
    );
```

而下面的代码则将所有连接中的 User-Agent 信息都删除了：

```
    chrome.webRequest.onBeforeSendHeaders.addListener(
        function(details){
            for(var i=0, headerLen=details.requestHeaders.length; i<headerLen; ++i){
                if(details.requestHeaders[i].name == 'User-Agent'){
                    details.requestHeaders.splice(i, 1);
                    break;
                }
            }
            return {requestHeaders: details.requestHeaders};
        },
        {
            urls: [
                "<all_urls>"
            ]
        },
        [
            "blocking",
            "requestHeaders"
        ]
    );
```

需要注意的是，header 中的如下属性是不支持更改的：Authorization、Cache-Control、Connection、Content-Length、Host、If-Modified-Since、If-None-Match、If-Range、Partial-Data、Pragma、Proxy-Authorization、Proxy-Connection 和 Transfer-Encoding。

下面的代码将所有访问 www.google.com.hk 的请求重定向到了 www.google.com：

```
    chrome.webRequest.onBeforeRequest.addListener(
        function(details){
            return {redirectUrl: details.url.replace( "www.google.com.hk",
"www.google.com")};
        },
        {
            urls: [
                "*://www.google.com.hk/*"
            ]
        },
        [
            "blocking"
        ]
    );
```

想想我们是不是可以做一个本地的 JS Library CDN 呢？

所有事件中，回调函数所接收到的信息对象均包括如下属性：requestId、url、method、frameId、parentFrameId、tabId、type 和 timeStamp。其中 type 可能的值包括"main_frame"、"sub_frame"、"stylesheet"、"script"、"image"、"object"、"xmlhttprequest"和"other"。

除了 onBeforeRequest 和 onErrorOccurred 事件外，其他所有事件返回的信息对象均包含 HttpHeaders 属性；onHeadersReceived、onAuthRequired、onResponseStarted、onBeforeRedirect 和 onCompleted 事件均包括 statusLine 属性以显示请求状态，如'HTTP/0.9 200 OK'。其他的属性还包括 scheme、realm、challenger、isProxy、ip、fromCache、statusCode、redirectUrl 和 error 等，由于使用范围较小，在此不详细介绍，读者可自行到 http://developer.chrome.com/extensions/webRequest 阅读完整内容。

5.3 代理

代理可以让用户通过代理服务器浏览网络资源以达到匿名访问等目的。代理的类型有多种，常用的包括 http 代理和 socks 代理等。有时我们不希望所有的网络资源都通过代理浏览，这种情况下通常会使用 pac 脚本来告诉浏览器使用代理访问的规则。

Chrome 浏览器提供了代理设置管理接口，这样可以让扩展来做到更加智能的代理设置。要让扩展使用代理接口，需要声明 proxy 权限：

```
"permissions": [
    "proxy"
]
```

通过 chrome.proxy.settings.set 方法可以设置代理服务器，该方法需要两个参数，一个是代理设置对象，另一个是回调函数。

代理设置对象包括 mode 属性、rules 属性和 pacScript 属性。其中 mode 属性为代理模式，可选的值有'direct'（直接连接，即不通过代理）、'auto_detect'（通过 WPAD 协议自动获取 pac 脚本）、'pac_script'（使用指定的 pac 脚本）、'fixed_servers'（固定的代理服务器）和'system'（使用系统的设置）。

rules 属性和 pacScript 属性都是可选的，rules 指定了不同的协议通过不同的代理，比如：

```
var config = {
    mode: "fixed_servers",
    rules: {
        proxyForHttp: {
            scheme: "socks5",
            host: "1.2.3.4",
            port: 1080
        },
        proxyForHttps: {
            scheme: "socks5",
            host: "1.2.3.5",
            port: 1080
        },
```

```
            proxyForFtp: {
                scheme: "http",
                host: "1.2.3.6",
                port: 80
            }
            bypassList: ["foobar.com"]
        }
    };
    chrome.proxy.settings.set(
        {value: config},
        function() {
    });
```

上面的代码定义了所有 http 协议的流量都使用 1.2.3.4:1080 这个 socks5 代理服务器代理浏览，所有 https 协议的流量都使用 1.2.3.5:1080 这个 socks5 代理服务器浏览，所有 ftp 协议的流量都使用 1.2.3.6:80 这个 http 代理服务器浏览，而 foobar.com 的流量不使用任何代理服务器，直接进行访问。rules 还提供了 singleProxy 属性（任何协议都使用此代理）和 fallbackProxy 属性（未匹配到的协议使用此代理）。

pacScript 指定了使用的 pac 脚本，可以通过 url 属性指定脚本位置，也可以直接通过 data 属性指定脚本内容。pacScript 还提供了 mandatory 属性以让浏览器决定当 pac 无效时是否阻止自动切换成直接访问，此属性默认为 false，即当 pac 无效时浏览器直接访问。

通过 chrome.proxy.settings.get 方法可以获取到浏览器当前的代理设置：

```
chrome.proxy.settings.get(
    {},
    function(config) {
        console.log(config.value);
    }
);
```

本节将不为大家提供 demo，而是直接带大家分析目前比较流行的 Chrome 代理管理扩展，SwitchySharp 有关代理设置的核心代码。

SwitchySharp 的完整代码可以通过 https://code.google.com/p/switchysharp 获取到，其中代理设置核心的代码为 assets/scripts/plugin.js，可以通过 https://code.google.com/p/switchysharp/source/browse/assets/scripts/plugin.js 在线查看此文件。

```
var ProxyPlugin = {};
ProxyPlugin.memoryPath = memoryPath;
ProxyPlugin.proxyMode = Settings.getValue('proxyMode', 'direct');
ProxyPlugin.proxyServer = Settings.getValue('proxyServer', '');
ProxyPlugin.proxyExceptions = Settings.getValue('proxyExceptions', '');
ProxyPlugin.proxyConfigUrl = Settings.getValue('proxyConfigUrl', '');
ProxyPlugin.autoPacScriptPath = Settings.getValue('autoPacScriptPath', '');
ProxyPlugin.mute = false;
```

SwitchySharp 首先声明了一个 ProxyPlugin 对象，此对象用来储存代理设置和代理设置方法。其中 proxyMode 属性为代理模式，和上文中讲到的代理模式相对应，但 fixed_server 模式在 proxyMode 中对应的值为 manual；proxyServer 属性为代理服务器地址；proxyExceptions 属性为不使用代理设置的例外，与上文提到的 bypassList 相对应；proxyConfigUrl 属性为 pac 脚本的 URL；autoPacScriptPath 为 SwitchySharp 中自动切换模式下使用的 pac 脚本路径。

mute 属性用来记录代理是否正在设置当中，如果不是，则此属性值为 false，如果代理设置正在被更改，则此值为 ture，用来避免设置冲突。最后_proxy 属性用来获取 Chrome 中代理设置的方法，为了做到最大限度兼容，SwitchySharp 对代理接口依然处于实验性阶段版本的 Chrome 进行了优化：

```
if (chrome.experimental !== undefined && chrome.experimental.proxy !== undefined)
    ProxyPlugin._proxy = chrome.experimental.proxy;
else if (chrome.proxy !== undefined)
    ProxyPlugin._proxy = chrome.proxy;
else
    alert('Need proxy api support, please update your Chrome');
```

ProxyPlugin 的 updateProxy 方法用来更新代理设置选项，这个方法在开始就先判断 mute 的值是否为真，也就是判断此时代理设置是否正在被更改，如果是则退出避免设置冲突。

```
ProxyPlugin._parseProxy = function (str) {
    if (str) {
        var proxy = {scheme:'http', host:'', port:80};
        var t1 = null;
        var t = str.indexOf(']') + 1;
        if (t > 0) {
            t1 = new Array();
            t1.push(proxy.host = str.substr(0, t));
            if (t < str.length - 1)
                t1.push(str.substr(t + 1));
        }
        else {
            t1 = str.split(':');
            proxy.host = t1[0];
        }
        var t2 = proxy.host.split('=');
        if (t2.length > 1) {
            proxy.scheme = t2[0] == 'socks' ? 'socks4' : t2[0];
            proxy.host = t2[1];
        }
        if (t1.length > 1)
            proxy.port = parseInt(t1[1]);
        return proxy;
    }
    else
        return {}
```

```
};
```

_parseProxy 方法用来解析声明多种代理的规则字符串，此方法将字符串转化为用于 fixed_servers 模式下的 rules 对象。

```
ProxyPlugin.setProxy = function (proxyMode, proxyString, proxyExceptions, proxyConfigUrl) {
    ...
    switch (proxyMode) {
        case 'system':
            config = {mode:"system"};
            break;
        ...
    }
    ProxyPlugin.mute = true;
    ProxyPlugin._proxy.settings.set({'value':config}, function () {
        ProxyPlugin.mute = false;
        if (ProxyPlugin.setProxyCallback != undefined) {
            ProxyPlugin.setProxyCallback();
            ProxyPlugin.setProxyCallback = undefined;
        }
    });
    profile = null;
    config = null;
    return 0;
};
```

最后 setProxy 方法将 ProxyPlugin 中与设置相关的属性重新整合成一个适用于 chrome.proxy.settings.set 方法的 config 对象，并调用 ProxyPlugin._proxy.settings.set 方法使之生效。

5.4 系统信息

Chrome 提供了获取系统 CPU、内存和存储设备的信息，要获取这些信息，需要在 Manifest 中分别声明如下权限：

```
"permissions": [
    "system.cpu",
    "system.memory",
    "system.storage"
]
```

三个接口都提供了 getInfo 方法以获取信息：

```
chrome.system.cpu.getInfo(function(info){
    console.log(info);
```

```
});

chrome.system.memory.getInfo(function(info){
    console.log(info);
});

chrome.system.storage.getInfo(function(info){
    console.log(info);
});
```

CPU 的信息包括 numOfProcessors、archName、modelName、features 和 processors，其中 processors 为一个记录所有逻辑处理器信息的数组。

内存信息包括 capacity 和 availableCapacity，即总容量和可用容量。

存储空间信息为一个包含多个存储设备信息的数组，每个存储设备的信息包括 id、name、type 和 capacity，其中 type 的可能值包括 fixed（本地磁盘）、removable（可移动磁盘）和 unknown（未知设备）。

system.storage 还提供了获取指定设备剩余空间和移除移动磁盘的方法 1：

```
chrome.system.storage.getAvailableCapacity(deviceId, function(info){
    console.log(info.availableCapacity);
});

chrome.system.storage.ejectDevice(deviceId, function(result){
    console.log(result);
});
```

1 目前 getAvailableCapacity 在稳定版 Chrome 中不可用。

chome.system.storage.onAttached 和 chome.system.storage.onDetached 事件分别用于监听可移动设备的插入和移除。

```
chrome.system.storage.onAttached.addListener(function(info){
    console.log(info);
});

chrome.system.storage.onDetached.addListener(function(deviceId){
    console.log(deviceId);
});
```

以上三个接口目前来说还比较新，这意味着 Google 可能会添加新的方法或者更改现有的方法，也可能移除这些方法，建议开发者在使用这些接口时谨慎选择。

第 6 章　Chrome 应用基础

从本章开始将为大家讲解应用（App）的部分。很多人难以区分 Chrome 中扩展和应用

的区别，后面的内容将向大家介绍何时使用扩展而何时使用应用，以及创建 Chrome 应用需要注意的地方。

6.1 应用与扩展的区别

Chrome 将其平台上的程序分为扩展与应用，并且使用了同样的文件结构，那么两者的区别是什么呢？在早期的 Chrome 版本中两者的区别非常模糊，而且有些扩展也可以用应用实现，反之亦然。但今天看来，Google 正在努力使两者的界限变得清晰。

总的来说，扩展与浏览器结合得更紧密些，更加强调扩展浏览器功能。而应用无法像扩展一样轻易获取用户在浏览器中浏览的内容并进行更改，实际上应用有更加严格的权限限制。所以应用更强调是一个独立的与 Chrome 浏览器关联不大的程序，此时你可以把 Chrome 看成是一个开发环境，而不是一个浏览器。

不过到目前为止，Google 还没有强制规定只能用扩展做什么，只能用应用做什么，所以对于那些扩展和应用都可以实现的功能，到底用何种方式实现，那是你自己的选择。不过我建议大家遵照上述的原则选择实现方式。

除此之外，Chrome 应用还分为 Hosted App（托管应用）和 Packaged App（打包应用），这两者也是有明显区别的。相对而言，Hosted App 更像是一个高级的书签，这种应用只提供一个图标和 Manifest 文件，在 Manifest 中声明了此应用的启动页面 URL，以及包含的其他页面 URL 和这些页面请求的高级权限。比如下面的例子创建了一个启动页面为 http://mail.google.com/mail/，包含 mail.google.com/mail/ 和 www.google.com/mail/ 且请求 unlimitedStorage 和 notifications 权限的应用。

```
{
    "name": "Google Mail",
    "description": "Read your gmail",
    "version": "1",
    "app": {
        "urls": [
            "*://mail.google.com/mail/",
            "*://www.google.com/mail/"
        ],
        "launch": {
            "web_url": "http://mail.google.com/mail/"
        }
    },
    "icons": {
        "128": "icon_128.png"
    },
    "permissions": [
        "unlimitedStorage",
        "notifications"
    ]
}
```

Packaged App，顾名思义，就是将所有文件打包在一起的应用，这类应用通常可以在离线时使用，因为它运行所需的全部文件都在本地。

由于 Hosted App 结构和功能都相对简单，所以后面的内容都将重点讲解 Packaged App。

6.2 更加严格的内容安全策略

在讲解 Chrome 扩展的安全策略时，提到过其不允许 inline-script，默认也不允许引用外部的 JavaScript 文件，而 Chrome 应用使用了更加严格的限制。

Chrome 扩展和应用都使用了 CSP（Content Security Policy）声明可以引用哪些资源，虽然之前我们并没有涉及到 CSP 的内容，但是 Chrome 扩展和应用会在我们创建时提供一个默认的值，对于 Chrome 扩展来说是 script-src 'self'; object-src 'self'。上面的 CSP 规则表示只能引用自身（同域下）的 JavaScript 文件和自身的 object 元素（如 Flash 等），其他资源未做限定。

Chrome 扩展允许开发者放宽一点点 CSP 的限制，即可以在声明权限的情况下引用 https 协议的外部 JavaScript 文件，如 script-src 'self' https://ajax.googleapis.com; object-src 'self'。但是 Chrome 应用不允许更改默认的 CSP 设置。

那么 CSP 到底是什么呢？它是如何定义安全内容引用范围的？

CSP 通常是在 header 或者 HTML 的 meta 标签中定义的，它声明了一系列可以被当前网页合法引用的资源，如果不在 CSP 声明的合法范围内，浏览器会拒绝引用这些资源，CSP 的主要目的是防止跨站脚本攻击（XSS）。

CSP 还是 W3C 草案，最新的 1.1 版文档还在撰写之中，所以在未来可能还会增加更多特性。目前 CSP 定义了 9 种属性，分别是 connect-src、font-src、frame-src、img-src、media-src、object-src、style-src、script-src 和 default-src。connect-src 声明了通过 XHR 和 WebSocket 等方式的合法引用范围，font-src 声明了在线字体的合法引用范围，frame-src 声明了嵌入式框架的合法引用范围，img-src 声明了图片的合法引用范围，media-src 声明了声音和视频媒体的合法引用范围，object-src 声明了 Flash 等对象的合法引用范围，style-src 声明了 CSS 的合法引用范围，script-src 声明了 JavaScript 的合法引用范围，最后 default-src 声明了未指定的其他引用方式的合法引用范围。

CSP 的可选属性值有'self'、'unsafe-inline'、'unsafe-eval'、'none'，这四个属性值都必须带引号代表特殊含义的值，分别表示允许引用同域资源、允许执行 inline-script、允许执行字符串转换的代码（如在 eval 函数和 setTimeout 中的字符串代码）、不允许引用任何资源。

同时还支持 host，如 www.google.com 表示可以引用 www.google.com 的资源，或者 *.google.com 允许引用 google.com 所有子域的资源（但不允许 google.com 根域的资源）。也可以声明只允许引用 https 下的资源，属性值声明为 https:即可。或者声明只允许引用特定协议特定 host 的资源，如 https://github.com。*则代表任何资源，即不受限制。

Chrome 应用默认的 CSP 规则为：

```
default-src 'self';
connect-src *;
style-src 'self' data: chrome-extension-resource: 'unsafe-inline';
img-src 'self' data: chrome-extension-resource:;
frame-src 'self' data: chrome-extension-resource:;
```

```
font-src 'self' data: chrome-extension-resource:;
media-src *;
```

也就是说在 Chrome 应用中，我们可以使用 XHR 请求任何资源；但是只能引用应用自身的 CSS 文件或者是 dataURL 转换的 CSS 文件和 chrome-extension-resource 协议下的 CSS 文件，同时我们可以在 HTML 直接写 style 代码块和在 DOM 中指定 style 属性；图片、嵌入式框架和字体只能引用自身或者 dataURL 转换的文件和 chrome-extension-resource 协议下的文件；可以引用任何音频和视频媒体资源；其他未指定的引用方式（object-src 和 script-src）只能引用自身资源。

这么做显然会大大提高 Chrome 应用的安全性，防止被黑客利用盗取用户的数据，但也显然带来了新的问题。从 Chrome 应用的 CSP 规则中我们发现其不允许通过嵌入式框架引用外部资源，那么如果我们真的需要将一个外部页面展示在 Chrome 应用中怎么办呢？Google 提供了 webview 标签代替 iframe 标签，使用 webview 标签必须指定大小和引用 URL。

```
<webview src="http://news.google.com/" width="640" height="480"></webview>
```

同样 Chrome 应用也不允许引用外部的图片，但是我们可以通过 XHR 请求外部图片资源（XHR 是可以请求到任何资源的，只要在 Manifest 中声明权限），然后通过转换成 blob URL 再添加到应用中。

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://supersweetdomainbutnotcspfriendly.com/image.png', true);
xhr.responseType = 'blob';
xhr.onload = function(e) {
    var img = document.createElement('img');
    img.src = window.URL.createObjectURL(this.response);
    document.body.appendChild(img);
};

xhr.send();
```

最后如果无法避免使用 inline-script 和 eval 等方式执行 JavaScript 代码，我们可以将“违规”的页面放入沙箱中执行，方法是在 Manifest 的 sandbox 中列出需要在沙箱中执行的页面。

```
"sandbox": {
    "pages": ["sandboxed.html"]
}
```

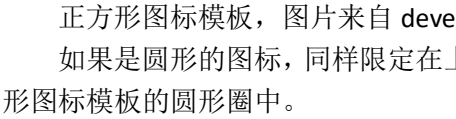
6.3 图标设计规范

虽然 Google 没有对应用图标的设计做出强制规定，但给出了一份建议文档，完整文档可以参见 https://developer.chrome.com/webstore/images，本节将根据原始文档内容，对图标设计规范相关的部分进行转述。

在应用展示页面（chrome://apps/），Chrome 默认会以 128 像素的尺寸展示应用图标，

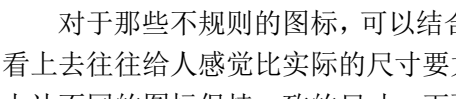
但根据窗口实际尺寸会自动进行缩放，最小展示 64 像素的图标。

Chrome 应用的图标只支持 png 格式，而且 Google 建议将图标的可视部分定在 96 像素之内，在可视部分周围留出边距。即如下图所示，将正方形的图标限定在方形框中。



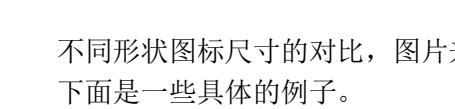
正方形图标模板，图片来自 developers.google.com

如果是圆形的图标，同样限定在上述模板的方形框中会显得过小，可以控制在下图的圆形图标模板的圆形圈中。



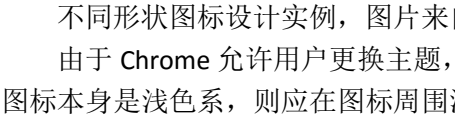
圆形图标模板，图片来自 developers.google.com

对于那些不规则的图标，可以结合正方形和圆形的模板进行设计。正方形和圆形的图标看上去往往给人感觉比实际的尺寸要大一些，所以在设计图标时要注意这一点，尽量在视觉上让不同的图标保持一致的尺寸。下面是不同形状的图标在一起的对比。



不同形状图标尺寸的对比，图片来自 developers.google.com

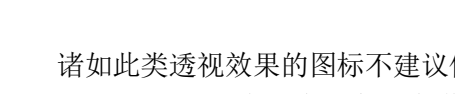
下面是一些具体的例子。



不同形状图标设计实例，图片来自 developers.google.com

由于 Chrome 允许用户更换主题，所以应考虑图标在不同明暗背景下的显示效果。如果图标本身是浅色系，则应在图标周围添加深色边界，反之亦然。

最后 Google 还建议，如果设计的图标有一定浮雕效果，凸起高度最好限制在 4 像素。图标最好是正对用户的，而不是侧面 45 度的透视效果，如下图所示。



诸如此类透视效果的图标不建议使用，图片来自 developers.google.com

虽然 Google 不会因为开发者未遵循上述规范而驳回或撤销应用，但是图标是一个应用给用户的第一印象，与周围应用图标风格明显有别的应用会流失一部分用户。

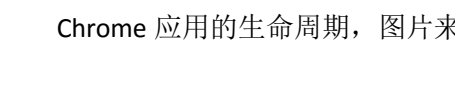
6.4 应用的生命周期

本章第一节的内容简单讲解了应用和扩展的区别，本节将为大家讲解应用和扩展的另一大区别，生命周期。

对于扩展来说，如果定义了后台脚本，同时指定 persistent 属性为 true，那么这个扩展只要浏览器运行就会一直运行，除非用户手动去关闭它。如果声明了 background 权限，则这个扩展会一开机就运行，并且一直运行下去。但是对于应用来说情况会有所不同。

Chrome 应用目前不允许使用永久运行的后台脚本（仅指 Packaged App），无论你多么想让它一直运行下去，比如用于监听来自 WebSocket 的消息等，Chrome 在认为应该关闭它时就会关闭它。这种做法确实有利于减少不必一直运行的应用消耗有限的内存资源，但就像上面举的例子那样，也同样会在某些方面带来局限。

既然关闭应用的行为并非开发者可以直接控制，那么我们就有必要了解应用何时开始运行，又会在何时被关闭——也就是应用的生命周期。下图给出了应用生命周期的简单图示。



Chrome 应用的生命周期，图片来自 developer.chrome.com

Event Page 就是 Chrome 应用的后台脚本，它用于监听各种事件。当用户运行应用，Event Page 加载完成后，onLaunched 事件就会被触发。如果这个应用运行后要向用户提供一个窗口，就是在 onLaunched 事件触发后后台脚本创建的。当这个窗口被关闭后，并且 Event Page 也没有需要处理的任务，Chrome 就会彻底关闭这个应用——连同 Event Page 一起关闭。在关闭应用前会触发 onSuspend 事件，这个事件可以提醒应用的后台脚本应用即将被关闭，以给应用一个准备退出的机会。

每一个应用都会有一个 Event Page，可以通过 Event Page 监听 onLaunched 事件，然后创建一个窗口。在 Manifest 的 app 属性中，通过 background 域定义 Event Page。

```
"app": {
    "background": {
        "scripts": ["background.js"]
    }
}
```

然后在 background.js 中指定应用启动时创建窗口 1。

1 从 Chrome 36 开始，create 方法的选项对象不再支持 bounds、minWidth、maxWidth、minHeight 和 maxHeight 属性，请使用 innerBounds 和 outerBounds 属性代替。innerBounds 和 outerBounds 的属性值包括 width、height、left、top、minWidth、maxWidth、minHeight 和 maxHeight。

```
chrome.app.runtime.onLaunched.addListener(function() {
    chrome.app.window.create('main.html', {
        id: 'MyWindowID',
        bounds: {
            width: 800,
            height: 600,
            left: 100,
            top: 100
        },
        minWidth: 800,
        minHeight: 600
    });
});
```

有关在应用中创建窗口的具体内容将在下一节给出。

在应用首次被安装或者更新到新版本时，会触发 onInstalled 事件，Event Page 可以在此事件触发时做一些初始化任务，如向本地写入默认设置、向用户展示欢迎窗口等等。

Chrome 应用可以使用 chrome.storage 存储数据，如：

```
chrome.runtime.onInstalled.addListener(function() {
    chrome.storage.local.set(object items, function callback);
});
```

数据保存在用户本地时，可能会面临数据永远丢失的风险——当用户卸载应用或者重新安装操作系统后，应用保存在本地的数据都会永久丢失。为防止这种风险，可以选择使用一种在线的存储方式，最简单的方法就是使用 Chrome storage API 中的 sync 域储存数据：

```
chrome.runtime.onInstalled.addListener(function() {
    chrome.storage.sync.set(items, function(){...});
});
```

这样这些数据会随 Chrome 在线同步保存在 Google 的服务器中。这对诸如应用设置等数据非常重要，因为用户也许会有一天重新安装卸载掉的应用。

最后当 Chrome 认为一个应用处于空闲状态时就会清理掉这个应用的进程，在清理之前会触发 onSuspend 事件，以让 Event Page 在退出前有机会做一些清理工作，从而不会导致意外退出。

有一种办法可以让应用一直保持运行，就是永远都不关闭前端窗口——当用户关闭窗口时隐藏它而不是真的关闭，或者让后台直接创建一个隐藏的窗口，这样前端窗口就可以一直运行且不显示出来。当用户再次启动应用时，再显示出来就可以了。

虽然也可以使用 setInterval 定期做一些简单的任务，如每 10 秒就让 Event Page 做点什么，但是 Chrome 清理应用进程的时间间隔是可以通过指定参数更改的，并且将来很有可能会发生变化，所以这种方法并不值得推荐。

对于个别应用确实需要长时间在后台运行，Google 也清楚这一点，但是为什么 Chrome 应用却不提供 background 权限呢？按照 Chromium 开发者的说法，由于要专门为应用提供 system tray 特性，应用可以通过这一特性常驻后台，在没有提供这一特性前，为了防止应用开发者依赖 background 权限，所以禁用了它。

那么现在的情况就比较尴尬了，system tray 尚未实现，background 权限又已被禁用，所以如果想让应用随系统启动并常驻后台，目前一种可行的方法——虽然笨拙但却有效，安装一个拥有 background 权限的扩展，这样可以让 Chrome 一直运行；在 Event Page 中添加一个 setInterval 任务以防止被 Chrome 认为应用处于空闲状态；创建一个 Chrome 应用的快捷方式（在 chrome://apps/中应用图标的右键菜单中，选择“创建快捷方式...”），并将这个快捷方式放到系统的启动文件夹下。

6.5 应用窗口

Chrome 应用中创建的窗口与 Chrome 浏览器中的窗口没有任何关系，这一点与 Chrome 扩展不同。 本节将详细讲解应用窗口的创建风格以及窗口相关的其他方法和事件。请记住，我们并不是在创建一个网页，而是在创建一个桌面程序，不要把应用的窗口风格搞得和网页一样。

6.5.1 创建窗口

通过 chrome.app.window.create 方法可以创建应用窗口，应用窗口与扩展中新建的窗口并不相同，应用窗口的默认样式与操作系统并没有太大关系，所以不同平台下 Chrome 应用的窗口能够保持较高的一致性。

如果不给出任何有关窗口外观样式的参数，应用窗口会是下面的样子：

应用窗口默认样式

创建这个窗口的代码为：

```
chrome.app.window.create('blank.html', {
    id: 'default'
});
```

其中 blank.html 为新建窗口嵌入的页面。

在上面的代码中只指定了窗口的 id，在以后创建新窗口时，也建议总是指定一个窗口 id，id 相同的窗口只会创建一个。

这个窗口没有指定大小，Chrome 给出的默认大小一般是 512×384 像素（不算标题栏），标题栏一般的默认高度是 22 像素，具体与系统设置有关。

窗口的控制按钮 Chrome 根据系统不同给出了相应的样式和位置，以照顾不同平台用户的使用习惯。比如 Windows 下控制按钮是方形并放置在右上角的，而对于 OS X 则是圆形的并放置在左上角。

Chrome 默认的应用窗口非常简洁，但在实际使用时需要注意在某些系统下默认窗口是没有阴影效果的，在白色背景的衬托下，用户将很难找到应用窗口的边缘。比如在早期版本的 Windows 下，如果没有定义一个窗口边框，应用窗口又恰好在一张白色网页前打开，用户就会看到下面的情景（窗口两侧是未被遮挡的 Google 搜索框）：

未定义窗口边框时在个别系统下用户会很难找到窗口边缘

最简单的方法是用 CSS 为 body 添加一个边框：

```
body {
    border: black 1px solid;
}
```

或者为 body 指定一个背景颜色，而不使用窗口默认的白色：

```
body {
    background: #EEE;
}
```

在创建窗口时也可以指定窗口的大小，如：

```
chrome.app.window.create('main.html', {
    id: 'main',
    bounds: {
        width: 800,
        height: 600
    }
});
```

如果窗口定义了 id，且用户对窗口进行了尺寸调整，下次再创建此窗口时 Chrome 会使用用户上次调整后的尺寸取代代码中的尺寸，这也是指定窗口 id 的好处之一。

通过 bounds 指定的尺寸是不包含窗口外框的，如标题栏等，只是窗口内嵌入页面的显示尺寸。如果不希望用户调整窗口尺寸可以指定窗口的 resizable 属性值为 false：

```
chrome.app.window.create('main.html', {
    id: 'main',
    bounds: {
```

```
        width: 800,
        height: 600
    },
    resizable: false
});
```

也可以指定窗口可调节尺寸的范围，比如：

```
chrome.app.window.create('main.html', {
    id: 'main',
    bounds: {
        width: 800,
        height: 600,
        minWidth: 400,
        minHeight: 300,
        maxWidth: 1600,
        maxHeight: 1200
    }
});
```

除了指定窗口大小，还可以指定窗口位置，如果不指定，则默认显示在屏幕中心。

```
chrome.app.window.create('main.html', {
    id: 'main',
    bounds: {
        top: 0,
        left: 0
    }
});
```

上面的代码创建了一个在屏幕左上角的窗口。如果指定了 id，Chrome 同样会记住用户上次将窗口放置的位置，并在下次创建窗口时使用记录的值。

其他的特性还包括新建窗口状态（最大化、最小化、正常或者全屏）和窗口是否总是在最前面，在声明 app.window.alwaysOnTop 权限的情况下，下面的代码创建了一个总是在最前面的全屏窗口：

```
chrome.app.window.create('main.html', {
    id: 'main',
    state: 'fullscreen',
    alwaysOnTop: true
});
```

其中 state 的值还可以是 normal、maximized 和 minimized。

最后窗口的 hidden 属性是非常重要的，它可以让窗口在后台静默运行，类似于后台脚本，但在需要时可以使用 show 方法重新显示出来，具体有关隐藏窗口的内容将在后面的内容中详细讲解。下面的代码创建了一个隐藏的窗口：

```
chrome.app.window.create('main.html', {
```

```
    id: 'main',
    hidden: true
});
```

窗口创建完成后我们也可以使用回调函数获取刚刚创建窗口的属性：

```
chrome.app.window.create('main.html', {'id': 'main'}, function(appWindow){
    console.log(appWindow);
});
```

6.5.2 样式更加自由的窗口

在上一节中讲解了创建应用窗口的方法，同时也介绍了部分窗口属性。虽然默认的窗口样式已经非常简洁，我们可以在窗口内部自由地进行设计，但是自带的标题栏却无法更改样式，本节将进一步讲解如何创建样式更加自由的窗口。

将窗口的 frame 属性值定为'none'，新建的窗口将不显示标题栏，如：

```
chrome.app.window.create('blank.html', {
    id: 'blank',
    frame: 'none'
});
```

上面的代码会生成下面所示的窗口：

没有标题栏的窗口

由于这个窗口没有标题栏，也没有控制按钮，所以无法拖拽，也无法通过在窗口上点击鼠标来关闭它，或者改变它的显示状态，比如最大化最小化等。

我们可以在 HTML 中指定可以拖拽的元素，这样当鼠标在这些元素上的时候就可以拖拽整个窗口了，下面对 blank.html 进行改进一下：

```
<html>
<head>
<title>A more free style window</title>
<style>
body {
    margin: 0;
    padding: 0;
    border: #EEE 1px solid;
}

#title_bar {
    -webkit-app-region: drag;
    height: 22px;
    line-height: 22px;
    font-size: 16px;
    background: #EEE;
```

```
        padding: 0 10px;
        box-sizing: border-box;
    }
    </style>
    </head>
    <body>
    <div id="title_bar">A more free style window</div>
    </body>
    </html>
```

可以拖拽的窗口

现在这个窗口可以拖拽了，重点就在于上面代码中的-webkit-app-region: drag。

在介绍自定义窗口控制按钮之前需要先了解获取当前窗口的方法，因为所有控制函数都是当前窗口对象的子元素。

```
var current_window = chrome.app.window.current();
```

上面的代码可以获取到当前代码所在窗口的窗口对象。

窗口对象的 close 方法可以关闭当前窗口，如：

```
current_window.close();
```

同样还可以最大化窗口、最小化窗口、还原窗口或全屏窗口：

```
current_window.maximize();
current_window.minimize();
current_window.restore();
current_window.fullscreen();
```

也可以获取当前窗口是否处于某种状态：

```
var is_maximize = current_window.isMaximized();
var is_minimize = current_window.isMinimized();
var is_fullscreen = current_window.isFullscreen();
```

以上函数均返回布尔型结果。

下面给 blank.html 页面添加上控制按钮。首先在 title_bar 的右侧添加三个圆形的按钮，分别对应最小化、最大化（还原）和关闭。

```
<div id="title_bar">A more free style window
    <a id="close" href="#"></a>
    <a id="maximize" href="#"></a>
    <a id="minimize" href="#"></a>
</div>
```

然后在样式表中添加这三个按钮的显示样式：

```
#title_bar a {
    display: inline-block;
```

```
    float: right;
    height: 12px;
    width: 12px;
    margin: 5px;
    border: black 1px solid;
    box-sizing: border-box;
    border-radius: 6px;
}
```

在添加按钮元素时，之所以将关闭按钮放在最前面，是因为样式表中定义了 float: right，这将使最先出现的元素放置在最右侧。

下面是添加按钮后的窗口：

添加按钮后的窗口

现在看起来虽然感觉好多了，但是当鼠标悬浮在按钮上时并没有反馈交互，所以我们还应该更加细化一下设计。继续在样式表中添加交互特性：

```
#title_bar a:hover {
    background: black;
}
```

当鼠标放在按钮上，按钮就会变成黑色的实心圆：

鼠标悬浮在按钮上的反馈交互

下面来为这三个按钮绑定事件。最小化和关闭按钮都很容易：

```
var current_window = chrome.app.window.current();

document.getElementById('minimize').onclick = function(){
    current_window.minimize();
}

document.getElementById('close').onclick = function(){
    current_window.close();
}
```

对于最大化的按钮，因为同时也是还原窗口的按钮，所以当用户点击时要进行判断：

```
document.getElementById('maximize').onclick = function(){
    current_window.isMaximized() ?
        current_window.restore() :
        current_window.maximize();
}
```

最后将写好的 JavaScript 紧贴在</body>标签之前引用。

但当我们进行测试时却发现点击按钮并没有反应，这是怎么回事呢？因为我们将控制按钮放在了 title_bar 之内，而 title_bar 之前定义了-webkit-app-region: drag 样式用于拖拽，这

会使 Chrome 阻止鼠标点击事件，解决的方法是专门为控制按钮定义-webkit-app-region: no-drag 样式。这一点没有在创建按钮时直接提出是为了强调其重要性，尤其是对于整个窗口都可以拖动的应用，应该为所有的控制按钮都专门指定-webkit-app-region: no-drag 样式。

下面是改进后完整的 HTML 代码：

```
<html>
<head>
<title>A more free style window</title>
<style>
body {
    margin: 0;
    padding: 0;
    border: #EEE 1px solid;
}

#title_bar {
    -webkit-app-region: drag;
    height: 22px;
    line-height: 22px;
    font-size: 16px;
    background: #EEE;
    padding: 0 10px;
    box-sizing: border-box;
}

#title_bar a {
    -webkit-app-region: no-drag;
    display: inline-block;
    float: right;
    height: 12px;
    width: 12px;
    margin: 5px;
    border: black 1px solid;
    box-sizing: border-box;
    border-radius: 6px;
}

#title_bar a:hover {
    background: black;
}
</style>
</head>
<body>
<div id="title_bar">A more free style window
    <a id="close" href="#"></a>
```

```
        <a id="maximize" href="#"></a>
        <a id="minimize" href="#"></a>
    </div>
    <script src="control.js"></script>
    </body>
    </html>
```

下面是完整的 JavaScript 代码：

```
var current_window = chrome.app.window.current();

document.getElementById('minimize').onclick = function(){
    current_window.minimize();
}

document.getElementById('close').onclick = function(){
    current_window.close();
}

document.getElementById('maximize').onclick = function(){
    current_window.isMaximized() ?
        current_window.restore() :
        current_window.maximize();
}
```

6.5.3 获取窗口

之前我们接触了 chrome.app.window.current 方法获取代码所在的窗口，除此之外还可以通过 chrome.app.window.getAll 方法获取全部窗口，以及 chrome.app.window.get 方法获取指定窗口。

chrome.app.window.current 和 chrome.app.window.get 方法均返回窗口对象，chrome.app.window.getAll 方法则返回包含若干窗口对象的数组。

调用 chrome.app.window.get 方法时需要指定窗口 id：

```
var main_window = chrome.app.window.get('main');
```

以下是窗口对象的完整结构，其中除 id 为字符串、contentWindow 为 JavaScript window object，其他均为函数。

```
{
    focus: 将焦点放在窗口上,
    fullscreen: 将窗口全屏,
    isFullscreen: 判断窗口是否处于全屏状态,
    minimize: 将窗口最小化,
    isMinimized: 判断窗口是否处于最小化状态,
    maximize: 将窗口最大化,
```

```
    isMaximized: 判断窗口是否处于最大化状态,
    restore: 还原窗口,
    moveTo: 将窗口移动到指定位置，调用方法为 moveTo(left, top),
    resizeTo: 将窗口尺寸设定为指定大小，调用方法为 resizeTo(width, height),
    drawAttention: 将窗口高亮显示,
    clearAttention: 清除窗口高亮显示,
    close: 关闭窗口,
    show: 显示隐藏窗口,
    hide: 隐藏窗口,
    getBounds: 获取窗口内容区域尺寸和位置,
    setBounds: 设置窗口内容区域尺寸和位置,
    isAlwaysOnTop: 判断窗口是否一直显示在最前端,
    setAlwaysOnTop: 将窗口设为总是最前端显示,
    contentWindow: JavaScript window object,
    id: 窗口 id，此 id 为创建时所指定
}
```

获取窗口时，如果指定的窗口不存在则返回 null。使用 getAll 方法时，如果不存在任何窗口则返回一个空数组。

在 6.4 节中提到了用隐藏窗口的方法防止应用被 Chrome 关闭，下面对之前的代码进行更改。首先将关闭按钮绑定的事件改为隐藏：

```
var current_window = chrome.app.window.current();

document.getElementById('close').onclick = current_window.hide();
```

其次将 Event Page 中启动事件改写成先判断窗口是否存在，如果存在则调用 show 方法显示，否则创建：

```
chrome.app.runtime.onLaunched.addListener(function() {
    var main_window = chrome.app.window.get('main');
    if(main_window){
        main_window.show();
    }
    else{
        chrome.app.window.create('main.html', {
            id: 'main',
            bounds: {
                width: 800,
                height: 600,
                left: 100,
                top: 100
            },
            frame: 'none'
        });
    }
```

```
});
```

当窗口关闭后，可以看到扩展程序管理器里显示 main.html 依然在运行。

使用 hide 方法阻止应用被关闭

6.5.4 窗口事件

应用窗口有 6 种事件，其中有 4 种用于监听窗口状态，分别是 onFullscreened、onMaximized、onMinimized 和 onRestored：

```
chrome.app.window.onFullscreened.addListener(function(){
    //do something when the window is set to fullscreen.
});

chrome.app.window.onMaximized.addListener(function(){
    //do something when the window is set to maximized.
});

chrome.app.window.onMinimized.addListener(function(){
    //do something when the window is set to minimized.
});

chrome.app.window.onRestored.addListener(function(){
    //do something when the window is set to restored.
});
```

另外两种事件一个用于监听窗口尺寸变化，另一个用于监听窗口被关闭：

```
chrome.app.window.onBoundsChanged.addListener(function(){
    //do something when the window is resized.
});

chrome.app.window.onClosed.addListener(function(){
    //do something when the window is closed.
});
```

6.6 编写第一个 Chrome 应用

在编写 Chrome 应用时请时刻记住，这已经不是单纯地开发浏览器扩展了，现在要编写的是一款真正的桌面程序，而 Chrome 只是类似 CLR 和 Java 的环境而已。

下面我们来一起编写一个计算机性能监视器。

首先来创建 Manifest 文件：

```
{
    "app": {
        "background": {
            "scripts": ["background.js"]
        }
    },
    "manifest_version": 2,
    "name": "Performance Monitor",
    "version": "1.0",
    "description": "A performance monitor to show cpu and memory status.",
    "icons": {
        "16": "images/icon16.png",
        "48": "images/icon48.png",
        "128": "images/icon128.png"
    },
    "permissions": [
        "system.cpu",
        "system.memory"
    ]
}
```

下面编写 background.js 脚本，根据 6.5 的内容可以直接写出如下代码：

```
chrome.app.runtime.onLaunched.addListener(function() {
    chrome.app.window.create('main.html', {
        'id': 'main',
        'bounds': {
            'width': 542,
            'height': 360
        },
        'resizable': false,
        'frame': 'none'
    });
});
```

同理，main.html 中的自定义菜单我们也用上一节提到的代码，但取消最大化按钮。为绘制曲线和饼图，本例使用了一个 JS 图表库，Chart.js，有关 Chart.js 的详细内容可以通过 http://www.bootcss.com/p/chart.js/查看。

下面是 main.html 的代码：

```
<html>
<head>
<title>Performance Monitor</title>
<style>
body {
    margin: 0;
```

```
    padding: 0;
    border: #EEE 1px solid;
}

#title_bar {
    -webkit-app-region: drag;
    height: 22px;
    line-height: 22px;
    font-size: 16px;
    background: #EEE;
    padding: 0 10px;
    box-sizing: border-box;
}

#title_bar a {
    -webkit-app-region: no-drag;
    display: inline-block;
    float: right;
    height: 12px;
    width: 12px;
    margin: 5px;
    border: gray 1px solid;
    box-sizing: border-box;
    border-radius: 6px;
}

#title_bar a:hover {
    background: gray;
}

#box_body {
    padding: 20px;
}

.chart {
    margin-bottom: 20px;
    font-size: 0;
}

.usage_item {
    color: gray;
    padding: 2px 0;
    font-size: 14px;
    border-bottom: #EEE 1px solid;
```

```
        margin-bottom: 4px;
}
</style>
</head>
<body>
<div id="title_bar">Performance Monitor
        <a id="close" href="#"></a>
        <a id="minimize" href="#"></a>
</div>
<div id="box_body">
<div class="usage_item">CPU Usage</div>
<div class="chart">
<canvas id="cpu_total" width="100" height="100"></canvas>
<canvas id="cpu_history" width="400" height="100"></canvas>
</div>
<div class="usage_item">Memory Usage</div>
<div class="chart">
<canvas id="mem_total" width="100" height="100"></canvas>
<canvas id="mem_history" width="400" height="100"></canvas>
</div>
</div>
<script src="control.js"></script>
<script src="Chart.js"></script>
<script src="main.js"></script>
</body>
</html>
```

其中的 canvas 用来展示数据。

control.js 的代码：

```
var current_window = chrome.app.window.current();

document.getElementById('minimize').onclick = function(){
        current_window.minimize();
}

document.getElementById('close').onclick = function(){
        current_window.close();
}
```

下面来编写 main.js，这个脚本用来定时获取数据并进行展示。

```
function getCpuUsage(callback){
        chrome.system.cpu.getInfo(function(info){
                var total = 0;
                var user = 0;
```

```
            var kernel = 0;
            for(var i=0; i<info.processors.length; i++){
                total += info.processors[i].usage.total - cpu_history.last_total[i];
                cpu_history.last_total[i] = info.processors[i].usage.total;
                user += info.processors[i].usage.user - cpu_history.last_user[i];
                cpu_history.last_user[i] = info.processors[i].usage.user;
                kernel += info.processors[i].usage.kernel - cpu_history.last_kernel[i];
                cpu_history.last_kernel[i] = info.processors[i].usage.kernel;
            }
            user = Math.round(user/total*100);
            kernel = Math.round(kernel/total*100);
            callback({user:user,kernel:kernel,total:user+kernel});
        });
    }

    function getMemUsage(callback){
        chrome.system.memory.getInfo(function(info){
            callback(info);
        });
    }

    function updateCpuHistory(){
        getCpuUsage(function(usage){
            cpu_history.user.shift();
            cpu_history.user.push(usage.user);
            cpu_history.kernel.shift();
            cpu_history.kernel.push(usage.kernel);
            cpu_history.total.shift();
            cpu_history.total.push(usage.total);
            showCpu();
        });
    }

    function updateMemHistory(){
        getMemUsage(function(usage){
            mem_history.used.shift();

mem_history.used.push(Math.round((usage.capacity-usage.availableCapacity)/usage.capacity*10
0));
            showMem();
        });
    }

    function updateData(){
```

```
        updateCpuHistory();
        updateMemHistory();
    }

    function showCpu(){
        var history = {
            labels  :  (function(){for(var  i=0,labels=[];i<ponits_num;labels.push(''),i++);return
labels;})(),
            datasets : [
                {
                    fillColor : "rgba(220,220,220,0.5)",
                    data : cpu_history.total
                },
                {
                    fillColor : "rgba(90,140,255,0.5)",
                    data : cpu_history.kernel
                },
                {
                    fillColor : "rgba(255,90,90,0.5)",
                    data : cpu_history.user
                }
            ]
        };

        var now = [
            {
                value: cpu_history.total[ponits_num-1],
                color:"rgba(220,220,220,0.7)"
            },
            {
                value : 100-cpu_history.total[ponits_num-1],
                color : "rgba(220,220,220,0.3)"
            }
        ];
        var his_ctx = document.getElementById('cpu_history').getContext("2d");
        var now_ctx = document.getElementById("cpu_total").getContext("2d");
        new Chart(his_ctx).Line(history, {scaleFontSize:4,pointDot:false,animation:false});
        new Chart(now_ctx).Pie(now, {segmentShowStroke:false,animation:false});
    }

    function showMem(){
        var history = {
            labels  :  (function(){for(var  i=0,labels=[];i<ponits_num;labels.push(''),i++);return
labels;})(),
```

```
            datasets : [
                {
                    fillColor : "rgba(220,220,220,0.5)",
                    data : mem_history.used
                }
            ]
        };

        var now = [
            {
                value: mem_history.used[ponits_num-1],
                color:"rgba(220,220,220,0.7)"
            },
            {
                value : 100-mem_history.used[ponits_num-1],
                color : "rgba(220,220,220,0.3)"
            }
        ];
        var his_ctx = document.getElementById('mem_history').getContext("2d");
        var now_ctx = document.getElementById("mem_total").getContext("2d");
        new Chart(his_ctx).Line(history, {scaleFontSize:4,pointDot:false,animation:false});
        new Chart(now_ctx).Pie(now, {segmentShowStroke:false,animation:false});
}

function init(){
        cpu_history = {
            user: [],
            kernel: [],
            total: [],
            last_user: [],
            last_kernel: [],
            last_total: []
        };
        mem_history = {
            used: []
        };
        init_cpu_history();
}

function init_cpu_history(){
        for(var i=0; i<ponits_num; i++){
            cpu_history.user.push(0);
            cpu_history.kernel.push(0);
            cpu_history.total.push(0);
```

```
        }
        chrome.system.cpu.getInfo(function(info){
            for(var i=0; i<info.processors.length; i++){
                cpu_history.last_total.push(info.processors[i].usage.total);
                cpu_history.last_user.push(info.processors[i].usage.user);
                cpu_history.last_kernel.push(info.processors[i].usage.kernel);
            }
            init_mem_history();
        });
    }

    function init_mem_history(){
        for(var i=0; i<ponits_num; i++){
            mem_history.used.push(0);
        }
        updateData();
        setInterval(updateData, 1000);
    }

    var cpu_history, mem_history, ponits_num=20;

    init();
```

其中 getCpuUsage 和 getMemUsage 函数分别用于获取 CPU 和内存的使用量。值得注意的是，Chrome 返回的 CPU 使用量是累计使用时间，并不是获取瞬间的 CPU 占用量，所以需要对前后两个时间点获得的结果做差。showCpu 和 showMem 方法将获取的数据显示成图表，两个函数都是参照 Chart.js 文档编写的，感兴趣的读者可以自行查阅。

另外 Chart.js 本身有 new Function 声明函数的部分，由于之前介绍的 CSP 规则，这在 Chrome 应用中是不被允许的，所以本例中的 Chart.js 是编者改写后的，具体差异读者可以下载后自行对照。在以后编写 Chrome 应用引用现成的库时，可能会经常遇到由于 CSP 的限制而无法直接运行的情况，这需要读者拥有自行更改库代码的能力。

本例应用运行的截图如下所示：

Performance Monitor 运行截图

本节中讲解的实例的源代码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/performance%20monitor 下载到。

第 7 章　文件系统

读写本地文件是一个程序最基本的功能，而对于 Web 技术来说，出于安全因素考虑，浏览器一直没有完全将这一功能开放给 JavaScript，直到 HTML5 提出了 FileSystem API。

Chrome 为应用提供了权限更加开放，功能更加强大的一系列文件系统接口，以满足 Chrome 应用作为桌面程序对磁盘读写的需求。在本章将详细为大家讲解选择目录、读取文件和写文件的方法。

要使用 FileSystem API 需要在 Manifest 中声明 fileSystem 权限：

```
permissions: {
    "fileSystem"
}
```

但如果只声明了上述权限，并不能写入文件及获取目录。如果还需要写入文件和获取目录需要进行如下声明：

```
permissions: {
    {"fileSystem": ["write", "directory"]}
}
```

值得注意的是，上面的权限声明中请求的权限值为对象型，即{"fileSystem": ["write", "directory"]}，而多数情况下是字符串型，如"storage"。

7.1 目录及文件操作对象

在 C 语言中操作文件时，实际操作的是文件指针，在 Chrome 应用中是通过目录及文件操作对象进行的。我们称目录操作对象为 DirectoryEntry，文件操作对象为 FileEntry，两者均继承自 Entry 对象。

Entry 有五个属性，分别是 filesystem、fullPath、isDirectory、isFile 和 name。其中 filesystem 是当前 Entry 所在的文件系统，filesystem 还有两个属性，分别是 name 和 root，name 是此文件系统的名称，root 是此文件系统的根目录 Entry。fullPath 是当前目录的绝对地址，字符串型。isDirectory 和 isFile 分别用于标识当前操作对象的类型，对于 DirectoryEntry 来说，两者的值分别为 true 和 false。name 是当前目录的名称，字符串型。

另外 DirectoryEntry 还有四种方法，分别是 createReader、getDirectory、getFile 和 removeRecursively。createReader 用于创建新的 DirectoryReader 对象来读取当前目录中的子目录和文件。getDirectory 用于读取或创建当前目录下的子目录。getFile 用于读取或创建当前目录下的文件。removeRecursively 用于删除当前目录下的所有文件和子目录，以及当前目录本身。

DirectoryEntry 的结构

FileEntry 与 DirectoryEntry 有很多类似的地方，如 FileEntry 具有和 DirectoryEntry 一样的五个属性，只不过对于 FileEntry 来说 isDirectory 和 isFile 的值分别为 false 和 true。

但是 FileEntry 所具有的方法与 DirectoryEntry 不同，FileEntry 只有两种方法，分别是 createWriter 和 file。其中 createWriter 用于创建一个新的 FileWriter 对象以用来向当前文件写入数据。file 方法会返回 File 对象，继承自 Blob 对象（包含文件内容、大小、MIME 类型），包含文件名和最后修改时间。

FileEntry 的结构

Chrome 应用中的 fileSystem 接口是对 HTML5 已有的文件系统接口的扩充，它允许 Chrome 应用读写硬盘中用户选择的任意位置，而 HTML5 本身提供的文件系统接口则只能在沙箱中读写文件，并不能获取用户磁盘中真正的目录。

有关 HTML5 文件系统更加详细的说明可以参见 W3C 文档 http://www.w3.org/TR/file-system-api/

7.2 获取目录及文件操作对象

无论是操作文件还是操作目录，都是对相应的操作对象进行操作，所以第一步都需要获取到目录及文件操作对象。Chrome 应用无法像 C 语言那样通过路径直接操作文件，目录及文件操作对象总是需要通过 Chrome 自带的文件选择窗口获取的。

通过 chooseEntry 方法可以获取到目录及文件操作对象。当 chooseEntry 被执行时，一个文件选择窗口会马上弹出，所以应该让一些事件来触发其运行，比如点击按钮等，否则可能会让用户感到困惑。

```
document.getElementById('openfile').onclick = function(){
    chrome.fileSystem.chooseEntry({}, function(fileEntry){
        console.log(fileEntry);
        //do something with fileEntry
    });
}
```

上面这段代码会让用户选择一个已存在的文件，并返回此文件对应的操作对象。如果在 Manifest 中声明了写权限，fileEntry 是可写的，否则是只读的。

在调用 chooseEntry 方法时，我们在上例中传递了一个空对象，这个对象用来定义 chooseEntry 打开的参数，默认情况下会以打开文件的方式获取操作对象。这个定义打开参数的对象完整结构如下：

```
{
    type: 打开类型，包括 openFile、openWritableFile、saveFile 和 openDirectory,
    suggestedName: 建议的文件名，会自动显示在保存窗口的文件名输入框中,
    accepts: [
        {
            description: 此选项的文字描述,
            mimeTypes: [接受的 mime 类型，如"image/jpeg"或"audio/*"],
            extensions: [接受的文件后缀，如"jpg"或"gif"]
        }
    ],
    acceptsAllTypes: 如果设定了接受的指定类型文件，是否接受所有的类型文件,
    acceptsMultiple: 是否接受多个文件，只支持 openFile 和 openWritableFile 的打开方式
}
```

在参数对象中如果未指定 type 属性，则默认为 openFile。由于在声明 write 权限后

openFile 方法获取的 FileEntry 可写，所以请考虑避免使用 openWritableFile，因为在以后 openWritableFile 很可能被 openFile 替代。

但是 saveFile 却无法被 openFile 替代，因为 saveFile 可以创建新的文件，openFile 则不可以。

将 type 指定为 openDirectory 则可以获取到目录操作对象：

```
document.getElementById('opendirectory').onclick = function(){
    chrome.fileSystem.chooseEntry({type: 'openDirectory'}, function(Entry){
        console.log(Entry);
        //do something with Entry
    });
}
```

7.3 读取文件

在 7.1 节中提到过 FileEntry 的 file 方法可以获取到文件的相关信息，实际上 file 方法返回的是 HTML5 中的 File 类型对象，所以有必要先介绍一下 HTML5 中的 FILE 对象。

HTML5 可以在文件未上传之前在浏览器端获取到文件的相关信息，就是通过 File API。当用户通过文件选择控件选择文件后，JavaScript 就可以通过控件 DOM 的 files 属性获取到对应的 File 对象：

```
document.getElementById('myFile').onchange = function(){
    var file = this.files[0];
    console.log(file);
}
```

对应的 HTML 为：

```
<input type="file" id="myFile" />
```

获取到的 FILE 对象包括文件最后更改日期、文件名、文件大小和文件类型等。

获取到的 File 对象

HTML5 还提供了 FileReader 对象，通过 FileReader 可以读取 File 对象对应文件的内容。

```
var reader = new FileReader();
reader.onload = function(){
    console.log(this.result);
}
reader.readAsText(File);
```

上述代码中 readAsText 是以文本方式读取文件内容，还可以通过 readAsDataURL 方式将文件内容读取成 dataURL，或者通过 readAsBinaryString 方式将文件内容读取成二进制字符串，以及 readAsArrayBuffer 方法读取二进制原始缓存区。

下面我们回到 Chrome 应用中。首先通过 chooseEntry 方法以 openFile 的方式获取 fileEntry：

```
chrome.fileSystem.chooseEntry({type: 'openFile'}, function(fileEntry){
    //We'll do something with fileEntry later
});
```

之后通过 FileEntry 的 file 方法获取到 File 对象：

```
fileEntry.file(function(file){
    //We'll do something with file later
});
```

最后用 FileReader 读取 file 中的内容：

```
var reader = new FileReader();
reader.onload = function(){
    var text = this.result;
    console.log(text);
    //do something with text
}
reader.readAsText(file);
```

将上面这三个过程连起来就可以得到如下代码：

```
chrome.fileSystem.chooseEntry({type: 'openFile'}, function(fileEntry){
    fileEntry.file(function(file){
        var reader = new FileReader();
        reader.onload = function(){
            var text = this.result;
            console.log(text);
            //do something with text
        }
        reader.readAsText(File);
    });
});
```

当然如果读取的文件中，并非是文本类型的数据，可以使用 readAsBinaryString 方式直接读取文件的二进制数据。

读取文件内容

7.4 遍历目录

通过 Entry 的 createReader 方法可以创建 DirectoryReader 对象，而 DirectoryReader 对象的 readEntries 方法又可以读取出当前目录下的一级子目录和文件，依次类推就可以遍历整个目录。

下面我们来实践写一个遍历目录的函数。

首先通过 chooseEntry 方法获取 Entry：

```
chrome.fileSystem.chooseEntry({type: 'openDirectory'}, function(Entry) {
    //We'll do something with Entry later
});
```

接下来我们来获取 Entry 下的子目录和文件：

```
var dirReader = Entry.createReader();
dirReader.readEntries (function(Entries) {
    //We'll do something with Entries later
}, errorHandler);
```

获取到 Entries 之后要对其中的每个元素进行判断是目录还是文件，如果是文件直接输出文件名，如果还是目录，则继续遍历：

```
for(var i=0; i<Entries.length; i++){
    //We'll print name of this Entry
    if(Entries[i].isDirectory){
        //We'll get sub Entries for this Entry
    }
}
```

基本的过程已经搞清楚了，现在开始编写打印 Entry 名的函数。我们希望设计成以下输出格式：

```
The full path of the selected Entry
|-Entry1
| |-sub Entry1
| | |-File1 in sub Entry1
| |-File1 in Entry1
| |-File2 in Entry1
|-File1
|-File2
```

所以显示 Entry 需要指定当前的目录深度以输出相应的层次格式：

```
function echoEntry(depth, Entry){
    var tree = '|';
    for(var i=0; i<depth-1; i++){
        tree += ' |';
    }
    console.log(tree+'-'+Entry.name);
}
```

然后我们将获取子目录和文件的代码也封装成一个函数以便复用：

```
function getSubEntries(depth, Entry){
    var dirReader = Entry.createReader();
    dirReader.readEntries (function(Entries) {
```

```
            for(var i=0; i<Entries.length; i++){
                echoEntry(depth+1, Entries[i]);
                if(Entries[i].isDirectory){
                    getSubEntries(depth+1, Entries[i]);
                }
            }
        }, errorHandler);
    }
```

最后在 chooseEntry 获取到 Entry 之后调用 getSubEntries 函数：

```
    chrome.fileSystem.chooseEntry({type: 'openDirectory'}, function(Entry) {
        console.log(Entry.fullPath);
        getSubEntries(0, Entry);
    });
```

别忘了定义 errorHandler 函数用于抓取错误：

```
    function errorHandler(e){
        console.log(e.message);
    }
```

但是细心的读者会发现按照上面的写法会先显示一级目录，而后显示二级目录以此类推，并不是像我们所设计的那样展示实际的目录结构。这是因为 getSubEntries 函数得到的结果是以回调的形式传递的，也就是说 getSubEntries 函数未执行结束并不会阻塞循环体。这个问题只是在显示结果时会造成一点小麻烦，在实际遍历目录时我们并不在意哪些先得到哪些后得到。但为了使本小节的例子更加完善，现将代码修改如下：

```
    var loopEntriesButton = document.getElementById('le');

    loopEntriesButton.addEventListener('click', function(e) {
        chrome.fileSystem.chooseEntry({type: 'openDirectory'}, function(Entry) {
            document.getElementById('loopEntry').innerText = Entry.fullPath;
            getSubEntries(0, Entry, document.getElementById('loopEntry'));
        });
    });

    function getSubEntries(depth, Entry, parent){
        var dirReader = Entry.createReader();
        dirReader.readEntries(function(Entries) {
            for(var i=0; i<Entries.length; i++){
                var newParent = document.createElement('div');
                newParent.id = Date.now();
                newParent.innerText = echoEntry(depth+1, Entries[i]);
                parent.appendChild(newParent);
                if(Entries[i].isDirectory){
                    getSubEntries(depth+1, Entries[i], newParent);
```

```
                }
            }
        }, errorHandler);
    }

    function echoEntry(depth, Entry){
        var tree = '|';
        for(var i=0; i<depth-1; i++){
            tree += ' |';
        }
        return (tree+'-'+Entry.name);
    }
```

对应的 HTML 为：

```
<input type="button" id="le" value="Loop Entries" />
<div id="loopEntry"></div>
```

最终运行的结果如下图所示：

遍历目录所得到的结果

7.5 创建及删除目录和文件

在 7.1 节中介绍过，Entry 的 getDirectory 和 getFile 方法可以获取和创建子目录和文件，在本节将主要讲解创建目录和文件。同时也会介绍删除目录和文件的方法。

在调用 getDirectory 方法时，如果在参数对象中指定 create 属性为 true，则会创建相应的子目录，如：

```
chrome.fileSystem.chooseEntry({type: 'openDirectory'}, function(Entry) {
    Entry.getDirectory('new_folder', {create: true}, function(subEntry) {
        //We'll do something with subEntry later
    }, errorHandler);
});
```

这将在用户所选择的目录下创建一个名为 new_folder 的子文件夹。同时也可以指定参数对象 exclusive 属性为 true，这将避免创建同名子目录——如果一旦创建的目录名与一已存在的子目录相同，会返回错误，而不会自动使用其他目录名。

当指定 exclusive 为 true，且创建同名目录时会抛出错误

同样在调用 getFile 方法时，参数对象中指定 create 属性为 true 会创建文件：

```
chrome.fileSystem.chooseEntry({type: 'openDirectory'}, function(Entry) {
    Entry.getFile('log.txt', {create: true}, function(fileEntry) {
        //We'll do something with fileEntry later
```

```
    }, errorHandler);
});
```

创建文件与创建目录基本相同，指定 exclusive 属性为 true 时，创建同名文件也会引起错误，所得到的错误信息与目录相同。

除了在用户选择的目录下创建文件外，也可以指定 chooseEntry 方法的打开类型为 saveFile，这样用户看到的将不是一个目录选择窗口，而是一个另存为窗口：

```
chrome.fileSystem.chooseEntry({
    type: 'saveFile',
    suggestedName: 'log.txt'
}, function(fileEntry) {
    //We'll do something with fileEntry later
});
```

通过指定 suggestedName 的值可以在另存为窗口中给出默认文件名，但用户可以自行更改这个文件名。

带有默认文件名的另存为窗口

Entry 和 FileEntry 的 remove 方法可以删除自身：

```
chrome.fileSystem.chooseEntry({type: 'openDirectory'}, function(Entry) {
    Entry.getDirectory('new_folder', {}, function(subEntry) {
        subEntry.remove(function(){
            console.log('Directory has been removed.');
        }, errorHandler);
    }, errorHandler);

    Entry.getFile('log.txt', {}, function(fileEntry) {
        fileEntry.remove(function(){
            console.log('File has been removed.');
        }, errorHandler);
    }, errorHandler);
});
```

对于目录来说，只有当目录不包含任何文件和子目录的时候 remove 方法才会调用成功，否则会报错。如果想删除包含内容的目录，需要使用 removeRecursively 方法：

```
chrome.fileSystem.chooseEntry({type: 'openDirectory'}, function(Entry) {
    Entry.getDirectory('new_folder', {}, function(subEntry) {
        subEntry.removeRecursively(function(){
            console.log('Directory has been removed.');
        }, errorHandler);
    }, errorHandler);
});
```

7.6 写入文件

通过 FileEntry 的 createWriter 方法可以获取 FileWriter 对象，通过 FileWriter 可以对文件进行写操作：

```
fileEntry.createWriter(function(fileWriter) {
    //We'll do something with fileWriter later
}, errorHandler);
```

对于 FileEntry，可以通过 Entry 的 getFile 方法获取，也可以直接通过指定 saveFile 类型的 chooseEntry 获得：

```
chrome.fileSystem.chooseEntry({type: 'openDirectory'}, function(Entry) {
    Entry.getFile('log.txt', {}, function(fileEntry) {
        fileEntry.createWriter(function(fileWriter) {
            //We'll do something with fileWriter later
        }, errorHandler);
    }, errorHandler);
});
```

或

```
chrome.fileSystem.chooseEntry({
    type: 'saveFile',
    suggestedName: 'log.txt'
}, function(fileEntry) {
    fileEntry.createWriter(function(fileWriter) {
        //We'll do something with fileWriter later
    }, errorHandler);
});
```

由于之后的操作都是针对 FileWriter 的，下面将只讲解与 FileWriter 相关的内容。

7.6.1 Typed Array

Typed Array（类型数组）为 JavaScript 直接处理原始二进制数据提供了接口。随着 HTML5 功能的增加，JavaScript 处理的数据已不仅仅局限于数字和字符串等基本类型，也会处理图像、声音、视频等更加复杂的数据，所以 JavaScript 需要一个直接操作原始二进制数据的接口。有关 Typed Array 草案和 WebGL 的内容可以通过 http://www.khronos.org/registry/typedarray/specs/latest/查看。

Typed Array 接口定义了一类固定长度的，可以直接获取缓存区数据的数组类型，ArrayBuffer 类型。可以通过 new ArrayBuffer(length)来创建一个长度为 length 字节的二进制缓存区，如：

```
var buf = new ArrayBuffer(8);
```

创建了一个长度为 8 字节（64 位）的 ArrayBuffer。

ArrayBuffer 类型的数据不可以直接读写，需要再构建 ArrayBufferView 类型数据才可以进行操作。那么 ArrayBuffer 和 ArrayBufferView 是什么样的关系呢？ArrayBuffer 是最原始的二进制数据，它没有附加任何信息，如数据是如何构造的。而 ArrayBufferView 则指定了原始二进制数据应该被如何看待——多少位被看做一个基本处理单元。为更加直观阐述这一关系，现举例如下：

```
var buf = new ArrayBuff(8);
```

此时对应于 buf 的数据是 8 字节（64 位），数据结构为：

```
+----+-+-+-+-+-+-+-+-+
|byte|0|1|2|3|4|5|6|7|
+----+-+-+-+-+-+-+-+-+
```

如果通过 Uint32Array 这一 ArrayBufferView 来格式化 buf 数据：

```
var uintBuf = new Uint32Array(buf);
```

则 uintBuf 的数据结构为：

```
+----+-+-+-+-+-+-+-+-+
|byte|0|1|2|3|4|5|6|7|
+----+-+-+-+-+-+-+-+-+
|uint|     0     |     1     |
+----+-+-+-+-+-+-+-+-+
```

在处理 uintBuf 数据时，JavaScript 会自动以一个 uint 为单位读写数组。

需要说明的是，创建 ArrayBufferView 并不会改变 ArrayBuffer 数据，它只是定义了 ArrayBuffer 的操作方式，实际上多个 ArrayBufferView 可以指向同一个 ArrayBuffer，但最终对 ArrayBufferView 的操作都是对 ArrayBuffer 数据的操作。

ArrayBufferView 一共有 8 种，分别是 Int8Array、Uint8Array、Int16Array、Uint16Array、Int32Array、Uint32Array、Float32Array 和 Float64Array，名称中间的数字代表格式化后数据基本单元位（bit）的长度。

ArrayBufferView 也可以指定 ArrayBuffer 中数据的起止位置，如：

```
var partUintBuf = new Uint8Array(buf, 3, 4);
```

这样 partUintBuf[0]将指向 buf 的第 4 个字节，partUintBuf[1]将指向 buf 的第 5 个字节……而 partUintBuf 这个数组只有 4 个元素。

下面我们来将一个 ArrayBuffer 数据按照字符串的方式读取出来。首先在 JavaScript 中字符类型（String）是占 16 位的，所以应该使用 Uint16Array 这个 ArrayBufferView 指定读取格式：

```
var stringBuf = new Uint16Array(buf);
```

这样 stringBuf 中的每个元素保存的就都是字符的 Unicode 码了，再使用 fromCharCode 方法转换成字符就可以了。但是 fromCharCode 方法需要传递多个参数：

```
String.fromCharCode(num0, num1, ..., numX);
```

而不是一个数组：

```
String.fromCharCode([num0, num1, ..., numX]);
```

可是我们获得的 stringBuf 是一个数组，所以不能直接传给 fromCharCode。当然可以使用一个循环将每个 Unicode 码进行转换，之后再拼接起来，但有简单的方法，apply 方法。apply 方法可以将一个对象的方法应用到另一个对象上，同时改变原方法中的 this 替换为指定的值。虽然看着有点乱，但这不是我们关心的，重要的是它可以自动将一个数组中的元素转化为函数的参数列表，即 foo.apply(null, [a, b, c])等同于 foo(a, b, c)，这正是我们所需要的。所以将 stringBuf 转换为字符串的方法就是：

```
String.fromCharCode.apply(null, stringBuf);
```

将 stringBuf 变量省略，就可以得到如下 ArrayBuffer 转换为 String 的函数：

```
function ab2str(buf){
    return String.fromCharCode.apply(null, new Uint16Array(buf));
}
```

7.6.2 Blob 对象

Blob 对象是对二进制数据的封装，它介于 ArrayBuffer 和应用层面数据之间。创建 Blob 对象非常简单，只需指定数据内容和数据类型即可：

```
var str = 'Internet Explorer is a good tool to download Chrome.';
var oneBlob = new Blob([str], {type: 'text/plain'});
```

值得注意的是创建 Blob 对象时的第一个参数永远都是一个数组，即使只有一个元素。第二个参数是创建 Blob 对象的可选参数，目前只包含 type 属性，指定 Blob 对象数据的类型，值为 MIME。如果不指定 type 的值，则 type 默认为一个空字符串。

创建 Blob 对象时可以通过字符串指定数据，如上例代码；也可以通过 ArrayBuffer、ArrayBufferView 和 Blob 类型数据，还可以是它们的组合，如：

```
var str = 'Internet Explorer is a good tool to download Chrome.';
var ab = new ArrayBuffer(8);
var abv = new Unit16Array(ab, 2, 2);
var oneBlob = new Blob([str], {type: 'text/plain'});
var anotherBlob = new Blob([ab, abv, oneBlob]);
```

当通过一个 Blob 被作为另一个 Blob 的数据时，它的类型会被忽略，即使数据数组中只有它一个元素时，如：

```
var oneBlob = new Blob(['Hello World.'], {type: 'text/plain'});
var anotherBlob = new Blob([oneBlob]);
```

anotherBlob 的类型不是'text/plain'，而是一个空字符串（因为创建时没有指定）。

Blob 对象有两个属性，分别是 size 和 type，其中 size 为 Blob 数据的字节长度，type 为指定的数据类型，两者均只读。

Blob 对象还有两种方法，分别是 slice 和 close。slice 方法与 String 中的分割非常像，只

不过在 Blob 中分割的是二进制数据。如：

```
var oneBlob = new Blob(['Hello World.'], {type: 'text/plain'});
var anotherBlob = oneBlob.slice(2, 4, 'text/plain');
```

slice 方法不会将原始 Blob 对象的类型传递给新的 Blob 对象，如果不指定新的 Blob 对象类型，其类型是一个空字符串。

close 方法用于永久删除 Blob 对象释放空间，一旦 Blob 被关闭，它将永远无法被再次调用。

7.6.3 FileWriter 对象

在本节开始介绍过，通过 FileWriter 可以对文件进行写操作，下面来详细介绍 FileWriter 相关的内容。

FileWriter 有两个属性，分别是 length 和 position。其中 length 为文件的长度，position 为指针的当前位置，即在文件中写入下一个数据的位置。两者均只读。

另外 FileWriter 还有三种方法，分别是 write、seek 和 truncate。其中 write 方法用来写入数据，数据类型为 Blob。如：

```
fileWriter.write(new Blob(['Hello World'], {type: 'text/plain'}));
```

可以通过 onwrite 和 onwriteend 监听数据开始写入和写入完毕事件：

```
fileWriter.onwrite = function(){
    console.log('Write begin.');
}

fileWriter.onwriteend = function(){
    console.log('Write complete.');
}
```

seek 方法用于移动指针到文件指定位置，之后的写操作将从指针指向的位置开始。如果 seek 给出的偏移量为负数，则将指针移动到距文件末端 n 个字节的位置。如果 seek 给出的偏移量为负数且绝对值比文件长度大，则将指针指向 0。如果偏移量比文件长度大，则指向文件末端。

```
//set position to beginning of the file
fileWriter.seek(0);

//set position to subtracting 5 from file length
fileWriter.seek(-5);

//set position to end of the file
fileWriter.seek(fileWriter.length);
```

truncate 方法用于更改文件长度，如果文件之前的长度比给定的值小，则以 0 填补，如果比给定的大，则舍弃超出部分的数据。

注意，如果 truncate 将文件长度缩小，而文件的指针又处于更改长度后文件范围之外（如

将文件长度更改为 10，而文件指针的位置在 20），那么新写入的数据将不会出现在文件中！在调用 truncate 方法后，一定记得检查指针位置是否依然在文件内部 1。

1 W3C 标准中指明 truncate 的值必须大于当前指针位置，但实际发现突破这一限制时，Chrome 依然可以成功执行。

结合前面的内容，我们就可以得到完整的写入文件的代码了：

```
chrome.fileSystem.chooseEntry({
    type: 'saveFile',
    suggestedName: 'log.txt'
}, function(fileEntry) {
    fileEntry.createWriter(function(fileWriter) {
        fileWriter.write(new Blob(['Hello World'], {type: 'text/plain'}));
    }, errorHandler);
});
```

7.7 复制及移动目录和文件

Entry 和 FileEntry 均有 copyTo 和 moveTo 方法用来复制和移动目录和文件。

```
Entry.copyTo(newEntry, 'new_Entry_name', function(copiedEntry){
    console.log('Entry moved.');
}, errorHandler);

Entry.moveTo(newEntry, 'new_Entry_name', function(movedEntry){
    console.log('Entry copied.');
}, errorHandler);

fileEntry.copyTo(newEntry, 'new_fileEntry_name', function(copiedFileEntry){
    console.log('fileEntry copied.');
}, errorHandler);

fileEntry.moveTo(newEntry, 'new_fileEntry_name', function(movedFileEntry){
    console.log('fileEntry moved.');
}, errorHandler);
```

如果不指定新的名称，则使用目录和文件原来的名称。

对于 moveTo 方法，不可以：

将目录移动到自身路径或其子目录路径下；
在其父系目录下移动且不指定新的名称；
将文件移动到已被其他目录占用的路径；
将目录移动到已被其他文件占用的路径；
将目录移动到一个非空目录占用的路径。

对于 copyTo 方法，不可以：

将一个目录复制到自身路径或其子目录路径下；
在其父系目录下复制且不指定新的名称；
将文件复制到已被其他目录占用的路径；
将目录复制到已被其他文件占用的路径；
将目录复制到一个非空目录占用的路径。

第 8 章　媒体库

通过 mediaGalleries 接口 Chrome 应用可以操作计算机中的媒体库，如音乐文件夹、图片文件夹、iPod 设备和 iTunes 等。

Chrome 应用操作媒体库与操作文件系统类似——其实媒体库也是文件系统的一部分，但是 mediaGalleries 接口与 fileSystem 有些区别。

首先 mediaGalleries 能自动找到计算机中的媒体库而不必像 fileSystem 那样需要用户手动指定目录或文件位置，其次 mediaGalleries 只会获取到支持的媒体文件，其他文件会被自动过滤掉。

要使用 mediaGalleries 接口需要在 Manifest 中声明 mediaGalleries 权限：

```
"permissions": {
    {"mediaGalleries": ["read", "allAutoDetected"]}
}
```

mediaGalleries 权限的声明与 fileSystem 类似，需要指定更加详细的权限。"read"表示有读取文件内容的权限，"allAutoDetected"表示有自动获取媒体库位置的权限。其他的权限还包括"delete"和"copyTo"，分别代表删除文件和复制文件。

需要注意的是 mediaGalleries 接口不提供"write"权限——直接在媒体库中创建或更改文件是禁止的，但可以在临时文件夹中创建文件后复制到媒体库中。

8.1　获取媒体库

如果在 Manifest 中声明了了"allAutoDetected"权限，则 Chrome 应用可以无需用户手动指定，自动获取到媒体库的位置。

通过 getMediaFileSystems 方法可以获取到媒体库对应的 fileSystem：

```
chrome.mediaGalleries.getMediaFileSystems({
    interactive: 'if_needed'
}, function(fileSystemArray){
    //We'll do something with fileSystemArray later
```

```
});
```

得到的是一个包含多个 fileSystem 对象的数组 fileSystemArray。fileSystem 对象我们在第 7 章虽有提及，但却接触不多，更多地是对 Entry 对象的操作。不过我们可以回忆一下 7.1 节在介绍 fileSystem 时提到过其有两个属性，分别是 name 和 root，其中 root 是此文件系统的根目录 DirectoryEntry。所以 filesystem.root 就是我们熟悉的 Entry 对象。

虽然通过 filesystem.root 可以像操作文件系统一样操作媒体库，但是除了文件系统中提供的属性外（如 isDirectory 和 isFile 等），对于媒体库我们还希望获得其他的信息——是否是媒体设备（如音乐播放器）、是否是可以移动设备（让我们来决定是否应进行同步操作）、目前设备是否可用等等。

为了得到这些信息，通过文件系统的接口是不够的，为此 Chrome 提供了获取此类信息的方法，getMediaFileSystemMetadata：

```
mediaInfo = chrome.mediaGalleries.getMediaFileSystemMetadata(mediaFileSystem);
```

它的传入参数是 fileSystem 对象，而不是 Entry 对象。

也可以通过 getAllMediaFileSystemMetadata 方法获取到全部的媒体库信息，但是 getAllMediaFileSystemMetadata 方法与 getMediaFileSystemMetadata 方法不同的是它使用回调函数的方式传回结果：

```
chrome.mediaGalleries.getAllMediaFileSystemMetadata(function(mediaInfoArray){
    //do something with mediaInfoArray
});
```

结果是一个数组，描述每个媒体库信息，这些对象的结构与通过 getMediaFileSystemMetadata 方法获取到的对象结构相同。

mediaInfo 对象包含 6 个属性，分别是 name、galleryId、deviceId、isRemovable、isMediaDevice 和 isAvailable。其中 isRemovable 表明此媒体库是否是一个可移动设备，isMediaDevice 表明此媒体库是否是一个媒体设备，isAvailable 表明此媒体库现在是否可用。

下面我们来一起制作一款媒体库管理应用，并在之后的小节中逐步完善它。

首先创建 Manifest 文件：

```
{
    "app": {
        "background": {
            "scripts": ["background.js"]
        }
    },
    "manifest_version": 2,
    "name": "Media Manager",
    "version": "1.0",
    "description": "A media manage tool.",
    "icons": {
        "128": "logo.png"
    },
    "permissions": [
        {"mediaGalleries": ["read", "delete", "copyTo", "allAutoDetected"]}
```

```
    ]
}
```

background.js 用来监控应用启动事件，当用户启动应用后创建一个窗口：

```
chrome.app.runtime.onLaunched.addListener(function() {
    chrome.app.window.create('main.html', {
        id: 'main',
        bounds: {
            width: 800,
            height: 600
        }
    });
});
```

在 main.html 用于展示检测到的媒体库：

```
<html>
<head>
<style>
@font-face {
    font-family: 'iconfont';
    src: url('iconfont.woff') format('woff');
}

* {
    padding: 0;
    margin: 0;
}

body {
    background: #2D2D2D;
}

#appTitle {
    height: 60px;
    line-height: 60px;
    padding: 0 20px;
    font-size: 24px;
    color: #CCC;
    background: #222;
}

#path {
    height: 40px;
    line-height: 40px;
```

```
        padding: 0 20px;
        color: #888;
        font-size: 16px;
        background: #222;
        border-bottom: black 1px solid;
        box-sizing: border-box;
}

#path span {
        display: block;
        float: left;
}

#path span.name {
        max-width: 100px;
        white-space: nowrap;
        text-overflow: ellipsis;
        overflow: hidden;
}

#path span.pointer {
        padding: 0 10px 0 5px;
}

#container .item {
        display: block;
        height: 100px;
        width: 100px;
        float: left;
        color: white;
}

#container .item .icon {
        display: block;
        text-align: center;
        height: 80px;
        line-height: 80px;
        font-size: 60px;
        font-family: 'iconfont';
}

#container .item .text {
        display: block;
        text-align: center;
```

```
        padding: 0 5px;
        height: 20px;
        line-height: 20px;
        font-size: 14px;
        text-overflow: ellipsis;
        overflow: hidden;
    }
    </style>
    </head>
    <body>
    <div id="appTitle">Media Manager</div>
    <div id="path"><span class="name">媒体库</span><span class="pointer">»</span></div>
    <div id="container"></div>
    <script src="main.js"></script>
    </body>
    </html>
```

其中用到了图标字体以显示矢量图标，字体来自 http://www.iconfont.cn/。

下面来编写 main.js：

```
function getMedia(){
    chrome.mediaGalleries.getMediaFileSystems({
        interactive: 'if_needed'
    }, listMediaGalleries);
}

function listMediaGalleries(fileSystemArray){
    document.getElementById('container').innerHTML = '';
    for(var i=0; i<fileSystemArray.length; i++){
        var info = chrome.mediaGalleries.getMediaFileSystemMetadata(fileSystemArray[i]);
        var item = document.createElement('span');
        item.className = 'item';
        item.title = info.name;
        document.getElementById('container').appendChild(item);
        var icon = document.createElement('span');
        icon.className = 'icon';
        icon.innerHTML = '&#xf00c5;';
        item.appendChild(icon);
        var text = document.createElement('span');
        text.className = 'text';
        text.innerHTML = info.name;
        item.appendChild(text);
    }
}
```

getMedia();

载入到 Chrome 后可以看到目前运行的截图。

获取媒体库

8.2 添加及移除媒体库

除了通过"allAutoDetected"权限让 Chrome 应用自动查找媒体库外，也可以让用户手动添加或者移除媒体库。

在上一节中我们调用 getMediaFileSystems 方法时，将其参数中的 interactive 指定为了 if_needed，如果将其指定为 yes 则会出现一个弹出让用户选择保留的媒体库或者添加其他媒体库：

媒体库选择弹窗

如果只想单纯提供添加其他位置的功能，可以使用 addUserSelectedFolder 方法，当调用 addUserSelectedFolder 方法时，会弹出一个目录选择窗口让用户选择新媒体库的位置：

添加新媒体库位置

addUserSelectedFolder 方法使用回调函数传递用户选择结果：

```
chrome.mediaGalleries.addUserSelectedFolder(function(mediaFileSystems,
selectedFileSystemName){
    //We'll do something with mediaFileSystems later
});
```

其中 mediaFileSystems 是一个包含多个 FileSystem 的数组，其包含的是应用有权限访问的所有媒体库 FileSystem，而非只是用户刚刚选择的。selectedFileSystemName 是一个字符串，如果用户在添加媒体库完成前点击了取消按钮，则 selectedFileSystemName 返回一个空值。

使用 dropPermissionForMediaFileSystem 方法可以取消对指定媒体库的访问权 1：

```
chrome.mediaGalleries.dropPermissionForMediaFileSystem(galleryId, function(){
    //do something after give up access a media gallery
});
```

1 从 Chrome 36 开始支持。

下面为 Media Manager 加上添加移除媒体库按钮：

```
<div id="appTitle">Media Manager<span id="edit">&#x3466;</span></div>
```

在 CSS 中添加按钮样式：

```
#edit {
    display: inline-block;
    font-size: 12px;
```

```
        font-family: 'iconfont';
        height: 20px;
        line-height: 20px;
        padding: 5px;
        cursor: pointer;
    }
```

在 JS 中添加按钮事件：

```
document.getElementById('edit').onclick = function(){
    document.getElementById('container').innerHTML = '';
    chrome.mediaGalleries.getMediaFileSystems({
        interactive: 'yes'
    }, listMediaGalleries);
}
```

改进后的窗口如下所示。

带有添加移除媒体库的窗口

8.3 更新媒体库

有时我们需要更新媒体库以让应用自动发现最新的媒体库。

通过 startMediaScan 方法开始更新媒体库 1：

```
chrome.mediaGalleries.startMediaScan();
```

1 从 Chrome 35 开始支持。

startMediaScan 没有然后返回值，也不会调用任何回调函数，因为更新的过程所花费的时间可能非常长，所以要使用 onScanProgress 来监听更新过程：

```
chrome.mediaGalleries.onScanProgress.addListener(function(details){
    //do something with details
});
```

其中 details 是一个对象，包含 5 个属性，分别是 type、galleryCount、audioCount、imageCount 和 videoCount。type 的可能值有 start、cancel、finish 和 error，分别对应于开始更新、取消更新、完成更新和遇到错误。

在更新媒体库的过程中，通过 cancelMediaScan 方法可以随时取消更新：

```
chrome.mediaGalleries.cancelMediaScan();
```

同样 cancelMediaScan 方法也没有提供回调函数，而应通过 onScanProgress 监测更新过程中的取消事件。

当 onScanProgress 监测到更新完成事件之后，可以通过 addScanResults 方法向用户展示一个选择添加最新检测到媒体库的窗口：

```
chrome.mediaGalleries.addScanResults(function(mediaFileSystems){
    //do something with mediaFileSystems
});
```

mediaFileSystems 是一个包含多个 FileSystem 的数组，其包含的是应用有权限访问的所有媒体库 FileSystem，而非只是用户刚刚选择的。

下面我们来将更新媒体库的功能加入到 Media Manager。首先在 HTML 中添加更新按钮、loading 元素和出错提示框：

```
<div id="error">更新失败</div>
<div id="appTitle">Media Manager<span id="edit">&#x3466;</span><span id="scan">&#xf015c;</span>
<div id="loading">
<div class="loading" index="0"></div>
<div class="loading" index="1"></div>
<div class="loading" index="2"></div>
<div class="loading" index="3"></div>
<div class="loading" index="4"></div>
</div>
</div>
```

之后在 CSS 中添加相应的样式：

```
#appTitle {
    height: 60px;
    line-height: 60px;
    padding: 0 20px;
    font-size: 24px;
    color: #CCC;
    background: #222;
    position: relative;
}

#edit, #scan {
    display: inline-block;
    font-size: 12px;
    font-family: 'iconfont';
    height: 20px;
    line-height: 20px;
    padding: 5px;
    cursor: pointer;
}

@-webkit-keyframes loading {
    0% {
        left: 0;
```

```
            opacity: 0;
        }
        5% {
            opacity: 1;
        }
        95% {
            opacity: 1;
        }
        100% {
            left: 100%;
            opacity: 0;
        }
}

.loading {
        width: 5px;
        height: 5px;
        background: #CCC;
        position: absolute;
        opacity: 0;
        -webkit-animation: loading 5s;
        -webkit-animation-timing-function: cubic-bezier(0.1, 0.48, 0.9, 0.52);
        -webkit-animation-iteration-count: infinite;
}

.loading[index="0"] {
        -webkit-animation-delay: 0.3s;
}

.loading[index="1"] {
        -webkit-animation-delay: 0.6s;
}

.loading[index="2"] {
        -webkit-animation-delay: 0.9s;
}

.loading[index="3"] {
        -webkit-animation-delay: 1.2s;
}

.loading[index="4"] {
        -webkit-animation-delay: 1.5s;
}
```

```
#loading {
    position: absolute;
    bottom: 0;
    left: 0;
    width: 100%;
    height: 5px;
    display: none;
}

#error {
    background: rgba(255, 255, 255, 0.5);
    height: 20px;
    line-height: 20px;
    font-size: 14px;
    color: #222;
    text-align: center;
    display: none;
}
```

其中为了让 loading 元素位置相对于 appTitle，将 appTitle 的 position 属性更改为了 relative。

最后在 JS 中加入相应事件：

```
var scanning = false;

document.getElementById('scan').onclick = function(){
    scanning?

chrome.mediaGalleries.startMediaScan&&chrome.mediaGalleries.startMediaScan():

chrome.mediaGalleries.cancelMediaScan&&chrome.mediaGalleries.cancelMediaScan();
}

document.getElementById('error').onclick = function(){
    this.style.display = 'none';
}

chrome.mediaGalleries.onScanProgress&&chrome.mediaGalleries.onScanProgress.addListe
ner(function(details){
    switch(details.type){
        case 'start':
            scanning = true;
            document.getElementById('loading').style.display = 'block';
            break;
        case 'cancel':
```

```
                scanning = false;
                document.getElementById('loading').style.display = 'none';
                break;
            case 'finish':
                scanning = false;
                document.getElementById('loading').style.display = 'none';
                chrome.mediaGalleries.addScanResults(listMediaGalleries);
                break;
            case 'error':
                scanning = false;
                document.getElementById('loading').style.display = 'none';
                document.getElementById('error').style.display = 'block';
                break;
        }
    });
```

其中 scanning 变量用来记录应用是否正在更新媒体库，如果正在更新，当用户点击更新按钮后会停止更新，否则开始更新。

更新媒体库相关方法和事件在部分版本中尚未生效，所以在调用时均先做以判断，防止出现方法未定义的情况发生。

8.4 获取媒体文件信息

在 8.1 节中提到过，通过 getMediaFileSystems 方法获取到的 fileSystem 中的 root 属性值就是 Entry 对象，结合第 7 章的内容就可以对媒体库中的文件进行操作。

通过 getMetadata 方法可以读取出媒体文件相关信息 1：

```
chrome.mediaGalleries.getMetadata(mediaFile, {metadataType: 'all'}, function(metadata){
    //do something with metadata
});
```

1 目前处于 Beta 分支。

其中 mediaFile 为 Blob 类型数据。metadataType 为获取信息的类型，如果不指定默认为 all，即全部信息，还可以指定为 mimeTypeOnly 来只获取 MIME。

metadata 为一个包含媒体信息的对象，完整结构如下：

```
{
    mimeType: MIME 类型,
    height: 视频或图片的高度，单位为像素,
    width: 视频或图片的宽度，单位为像素,
    xResolution: 照片的水平分辨率，用电视线表示,
    yResolution: 照片的垂直分辨率，用电视线表示,
    duration: 视频或音乐的长度，以秒为单位,
    rotation: 视频或图片的旋转角度，以度为单位,
```

```
        cameraMake: 图片中的相机制造商,
        cameraModel: 相机模式,
        exposureTimeSeconds: 曝光时间,
        flashFired: 是否开启闪光灯,
        fNumber: 光圈大小,
        focalLengthMm: 焦距，单位为毫米,
        isoEquivalent: 等效 ISO,
        album: 视频或音乐专辑,
        artist: 艺术家,
        comment: 评分,
        copyright: 版权信息,
        disc: 盘片编号,
        genre: 流派,
        language: 语言,
        title: 标题,
        track: 音轨数,
        rawTags: [
            {
                type: 类型，如 mp3、h264,
                tags: 标签
            }
        ]
    }
```

最后值得说明的是，由于已经在 Manifest 中声明了媒体库的读取权限，而不必另外声明 fileSystem 权限用于读取媒体文件。

由于遍历目录和读取、复制、删除文件与文章讲解的内容无关，读者可自行参考第 7 章内容实现相关功能。Media Manager 实例的源码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/media_manager 下载到。

第 9 章　网络通信

Chrome 应用通过 sockets 接口支持 TCP 和 UDP 协议，使网络通信成为可能。使用 sockets 接口时，声明权限比较特殊，并不在 permissions 中声明，而是直接在 Manifest 的 sockets 中声明：

```
    "sockets": {
        "udp": {
            "send": ["host-pattern1", ...],
            "bind": ["host-pattern2", ...],
            ...
        },
```

```
        "tcp" : {
            "connect": ["host-pattern1", ...],
            ...
        },
        "tcpServer" : {
            "listen": ["host-pattern1", ...],
            ...
        }
    }
```

但在早期的 Chrome 版本中 socket 权限依然在 permissions 中声明。

sockets 接口传输的数据类型为 ArrayBuffer，有关 ArrayBuffer 的内容可以参阅 7.6.1 节的内容。

最后本章还会介绍有关 WebSocket 的内容，这是 HTML5 原生支持的方法。

9.1 UDP 协议

UDP 协议是一个简单的面向数据报的传输层协议，它是一种不可靠数据报协议。由于缺乏可靠性且属于非连接导向协定，UDP 应用一般必须允许一定量的丢包和出错。

Chrome 提供 sockets.udp 接口使 Chrome 应用可以进行 UDP 通信。要使用 sockets.udp 接口需要在 sockets 域中声明 udp 权限：

```
"sockets": {
    "udp": {
        "send": ["192.168.1.106:8000", ":8001"],
        "bind": ":8000"
    }
}
```

上面的代码表示应用可以通过 UDP 与 192.168.1.106 的 8000 端口通信，也可以与任意主机的 8001 端口通信。应用可以绑定本地的 8000 端口用来接收 UDP 消息。

如果想连接任意主机的任意端口可以声明为"send": "*"。主机和端口可以是一个特定的字符串，也可以是一个数组表示多个规则，如"bind": [":8000", ":8001"]。如果想连接 192.168.1.106 的任意端口可以声明为"send": "192.168.1.106"。

9.1.1 建立与关闭连接

创建 socket

```
var socketOption = {
    persistent: true,
    name: 'udpSocket',
    bufferSize: 4096
};
```

```
chrome.sockets.udp.create(socketOption, function(socketInfo){
    //We'll do the next step later
});
```

socketOption 是一个可选参数，其中 persistent 代表 Chrome 应用的生命周期结束后（参见 6.4 节）这个 socket 是否还依然处于开启状态。bufferSize 定义 socket 接收数据的缓存区大小，如果这个值过小，会导致数据丢失，默认值是 4096。

当 socket 创建完毕就返回一个对象，这个对象包含代表这个 socket 的唯一 id，之后对 socket 的操作都将根据这个 id。

更新 socket 属性

此处说的属性是指创建 socket 时提到的 socketOptions。通过 update 方法可以更新 socket 的属性：

```
chrome.sockets.udp.update(socketId, newSocketOption, function(){
    //do something when update complete
});
```

阻止和解除阻止 socket 接收数据

当一个 socket 被阻止后，将不会触发消息接收事件，解除阻止后将恢复正常。

```
//Blocking socket receiving data
var isPaused = true;

chrome.sockets.udp.setPaused(socketId, isPaused, function(){
    //do something after pause a socket
});
```

如果想解除阻止，将上述代码中的 isPaused 设为 false 即可。

绑定端口

绑定固定端口用以接收 UDP 消息：

```
var localAddress = '10.60.37.105';
var localPort = 6259;

chrome.sockets.udp.bind(socketId, localAddress, localPort, function(code){
    //if a negative value is returned, an error occurred
    //otherwise do something after bind a port
});
```

一台计算机往往有多个 IP，如本地回路 IP、局域网 IP 和公网 IP 等，那么指定 localAddress 为"0.0.0.0"系统会自动绑定一个合适的 IP 地址，而不必让开发者获取所有 IP 地址后再进行选择。同样可以指定 localPort 为 0，这样系统会自动分配一个空闲的端口给应用使用。

关闭 socket

当一个 socket 不再被使用了我们应该关闭它。

```
chrome.socket.udp.close(socketId, function(){
    //do something after close a socket
```

```
});
```

下面我们来试着封装一个 udp 类，并在之后的内容逐步扩充它：

```
function udp(){
    var _udp = chrome.sockets.udp;
    this.option = {},
    this.socketId = 0,
    this.localAddress = '0.0.0.0',
    this.localPort = 0,

    this.create = function(callback){
        _udp.create(this.option, function(socketInfo){
            this.socketId = socketInfo.socketId;
            callback();
        }.bind(this));
    }.bind(this),

    this.update = function(){
        _udp.update(this.socketId, newSocketOption, callback);
    }.bind(this),

    this.pause = function(isPaused, callback){
        _udp.setPaused(this.socketId, isPaused, callback);
    }.bind(this),

    this.bind = function(callback){
        _udp.bind(this.socketId, this.localAddress, this.localPort, callback);
    }.bind(this),

    this.close = function(callback){
        _udp.close(this.socketId, callback);
    }.bind(this),

    this.init = function(callback){
        this.create(function(){
            this.bind(callback);
        }.bind(this));
    }.bind(this)
}
```

调用时指定 socket 属性、绑定 IP 和端口就可以进行初始化了：

```
var udpSocket = new udp();
udpSocket.option = {
    persistent: true
```

```
};
udpSocket.localPort = 8000;
udpSocket.init(function(code){
    if(code<0){
        console.log('UDP Socket bind failed, error code: '+code);
        return false;
    }
    else{
        //We'll do something after udp socket init later
    }
});
```

9.1.2 发送与接收数据

发送数据

Socket 发送的数据类型为 ArrayBuffer，对 ArrayBuffer 不熟悉的读者请参阅 7.6.1 节的内容。

```
chrome.sockets.udp.send(socketId, data, address, port, function(){
    //do something after send some data
});
```

其中 data 为 ArrayBuffer 类型的数据，address 为接收方的 ip，port 为接收方的端口。

接收数据

当 socket 接收到数据时，就会触发 onReceive 事件：

```
chrome.socket.udp.onReceive.addListener(function(info){
    //We'll do something with info later
});
```

返回的 info 是一个对象，包括 4 个属性：socketId、data、remoteAddress 和 remotePort。其中 data 为 ArrayBuffer 类型数据，remoteAddress 和 remotePort 分别为发送方 ip 地址和端口。

处理异常

当网络出现问题时，会触发 onReceiveError 事件，同时 socket 会被阻断：

```
chrome.sockets.udp.onReceiveError.addListener(function(info){
    //We'll do something with info later
});
```

与接收数据相似，info 也是一个对象，但只包含 socketId 和 resultCode 两个属性。

现在我们来把 8.1.1 中的 udp 类完善一下，由于篇幅限制，将只写出添加或改动的部分：

```
function udp(){
    this.send = function(address, port, data, callback){
        _udp.send(this.socketId, data, address, port, callback);
    }.bind(this),
```

```
    this.receive = function(info){
        console.log('Received data from '+info.removeAddress+':'+info.removePort);
    },

    this.error = function(code){
        console.log('An error occurred with code '+code);
    },

    this.init = function(callback){
        this.create(function(){
            this.bind(function(code){
                if(code<0){
                    this.error(code);
                    return false;
                }
                else{
                    callback();
                }
            }.bind(this));
            _udp.onReceive.addListener(function(info){
                if(info.socketId==this.socketId){
                    this.receive(info);
                }
            }.bind(this));
            _udp.onReceiveError.addListener(function(info){
                if(info.socketId==this.socketId){
                    this.error(info.resultCode);
                }
            });
        }.bind(this));
    }.bind(this)
}
```

9.1.3 多播

多播用于一个有限的局部网络中的 UDP 一对多通信。之所以说是一个有限的局部网络是因为这个范围是无法确定的，一个是因为一个数据能传多远由 TTL 决定，多播中 TTL 一般被设为 15，最多不会超过 30，也有设为 0 的（数据不会流出本地）。再一个就是有的路由是不会转发多播数据的，即使 TTL 在此节点并没有减为 0。

多播使用一段保留的 ip 地址，224.0.0.0 到 239.255.255.255。其中 224.0.0.0 到 224.0.0.255 为局部连接多播地址，224.0.1.0 到 238.255.255.255 为预留多播地址，239.0.0.0 到 239.255.255.255 为管理权限多播地址。局部连接多播地址和管理权限多播地址均为保留多

播地址，可被自由使用的只有预留多播地址，即 224.0.1.0 到 238.255.255.255。

多播地址是特殊的 ip 地址，它不对应任何物理设备。但 UDP 进行多播时，只将其看做普通的 ip 地址就可以。

要使用多播，需要在 sockets 的 udp 中声明 multicastMembership 权限：

```
"sockets": {
    "udp": {
        "multicastMembership": "*";
    }
}
```

如果提示 Invalid host:port pattern，这是 Chrome 的 Bug，解决方法是 multicastMembership 的值设定成空字符串：

```
"sockets": {
    "udp": {
        "multicastMembership": "";
    }
}
```

加入组

```
chrome.sockets.udp.joinGroup(socketId, address, function(code){
    //if a negative value is returned, an error occurred
    //otherwise do something after join a group
});
```

离开组

```
chrome.sockets.udp.leaveGroup(socketId, address, function(code){
    //if a negative value is returned, an error occurred
    //otherwise do something after leave a group
});
```

设置多播的 TTL

```
chrome.sockets.udp.setMulticastTimeToLive(socketId, ttl, function(code){
    //if a negative value is returned, an error occurred
    //otherwise do something after set multicast TTL
});
```

设置多播回环模式

这个模式定义了当主机本身处于多播的目标组中时，是否接收来自自身的数据。如果一台主机中多个程序加入了同一个多播组，但回环模式设置矛盾时，Windows 会不接收来自本机自身的数据，而基于 Unix 的系统则不接收来自程序自身的数据。

```
chrome.sockets.udp.setMulticastLoopbackMode(sockedId, enabled, function(code){
    //if a negative value is returned, an error occurred
    //otherwise do something after set multicast loopback mode
});
```

让我们把多播的功能加入到 udp 类中：

```
function udp(){
    this.joinGroup = function(address, callback){
        _udp.joinGroup(this.socketId, address, function(code){
            if(code<0){
                this.error(code);
                return false;
            }
            else{
                callback();
            }
        }.bind(this));
    }.bind(this),

    this.leaveGroup = function(address, callback){
        _udp.leaveGroup(this.socketId, address, function(code){
            if(code<0){
                this.error(code);
                return false;
            }
            else{
                callback();
            }
        }.bind(this));
    }.bind(this),

    this.setMilticastTTL = function(ttl, callback){
        _udp.setMulticastTimeToLive(this.socketId, ttl, function(code){
            if(code<0){
                this.error(code);
                return false;
            }
            else{
                callback();
            }
        }.bind(this));
    }.bind(this),

    this.setMilticastLoopback = function(enabled, callback){
        _udp.setMulticastLoopbackMode(this.sockedId, enabled, function(code){
            if(code<0){
                this.error(code);
                return false;
```

```
                }
                else{
                    callback();
                }
            }.bind(this));
        }.bind(this)
    }
```

9.1.4 获取 socket 和组

获取指定 socket

```
chrome.sockets.udp.getInfo(socketId, function(socketInfo){
    //do something with socketInfo
});
```

socketInfo 是一个描述 socket 信息的对象，包括 socketId、persistent、name、bufferSize、paused、localAddress 和 localPort。

获取全部活动的 socket

```
chrome.sockets.udp.getSockets(function(socketInfoArray){
    //do something with socketInfoArray
});
```

socketInfoArray 是一个包含一个或多个 socketInfo 对象的数组。

获取指定 socket 加入的组

```
chrome.sockets.udp.getJoinedGroups(socketId, function(groupArray){
    //do something with groupArray
});
```

groupArray 是一个包含多个组地址字符串的数组。

将以上方法加入到 udp 类中：

```
function udp(){
    this.getInfo = function(callback){
        _udp.getInfo(this.socketId, callback);
    }.bind(this),

    this.getSockets = function(callback){
        _udp.getSockets (callback);
    }.bind(this),

    this.getGroups = function(callback){
        _udp.getJoinedGroups(this.socketId, callback);
    }.bind(this)
}
```

9.1.5 局域网聊天应用

经过前面的介绍，我们对 UDP 在 Chrome 应用中的使用有了一定的了解，本节来和大家一起编写一款局域网聊天的应用。

这个应用利用 UDP 的多播功能，进行一对多通信，来实现局域网聊天。首先需要在 Manifest 的 sockets 中声明 udp 的权限：

```
{
    "app": {
        "background": {
            "scripts": ["udp.js", "background.js"]
        }
    },
    "manifest_version": 2,
    "name": "Local Messager",
    "version": "1.0",
    "description": "A local network chating application.",
    "icons": {
        "128": "lm.png"
    },
    "sockets": {
        "udp": {
            "send": "224.0.1.100",
            "bind": ":*",
            "multicastMembership": "224.0.1.100"
        }
    }
}
```

注意，由于 8.1.3 节中提到的 Bug，multicastMembership 的值在实际操作中可能需要指定为空字符串""，即"multicastMembership": ""。

Event Page 中指定的 udp.js 就是我们在之前写好的 udp 类。下面我们来编写 background.js。

首先当应用运行时开始创建 UDP 连接并加入到多播组：

```
var udpSocket = new udp();
udpSocket.localPort = 8943;
udpSocket.receive = receiveMsg;
udpSocket.init(function(){
    udpSocket.joinGroup('224.0.1.100', function(){
        //Joined group 224.0.1.100
    });
});
```

下面需要 background 来监听来自前端页面发来的指令：

```
chrome.runtime.onMessage.addListener(function(message, sender, callback){
    if(message.action == 'send'){
        var buf = str2ab(message.msg);
        udpSocket.send('224.0.1.100', udpSocket.localPort, buf, function(){
            //message is sent
        });
    }
});
```

下面我们来编写接收消息的函数：

```
function receiveMsg(info){
    var msg = ab2str(info.data);
    chrome.runtime.sendMessage({action:'receive', msg:msg});
}
```

最后来编写 ArrayBuffer 和 String 类型数据互换的两个函数：

```
function str2ab(str){
    var buf = new ArrayBuffer(str.length*2);
    bufView = new Uint16Array(buf);
    for(var i=0; i<str.length; i++){
        bufView[i] = str.charCodeAt(i);
    }
    return buf;
}

function ab2str(buf){
    return String.fromCharCode.apply(null, new Uint16Array(buf));
}
```

UDP 通信相关的内容写好了，接下来创建前端窗口：

```
chrome.app.runtime.onLaunched.addListener(function(){
    chrome.app.window.create('main.html', {
        'id': 'main',
        'bounds': {
            'width': 400,
            'height': 600
        }
    });
});
```

前端窗口 main.html 的 HTML 代码：

```
<html>
<head>
<title>Local Messager</title>
```

```
<style>
* {
	margin: 0;
	padding: 0;
}

body {
	border: #EEE 1px solid;
}

#history {
	margin: 10px;
	border: gray 1px solid;
	height: 480px;
	overflow-y: auto;
	overflow-x: hidden;
}

#history div {
	padding: 10px;
	border-bottom: #EEE 1px solid;
}

#msg {
	margin: 10px;
	width: 480px;
	height: 80px;
	box-sizing: border-box;
	border: gray 1px solid;
	font-size: 24px;
}
</style>
</head>
<body>
<div id="history"></div>
<input type="text" id="msg" />
<script src="main.js"></script>
</body>
</html>
```

main.js 的代码：

```
document.getElementById('msg').onkeyup = function(e){
	if(e.keyCode==13){
		chrome.runtime.sendMessage({
```

```
                action:'send',
                msg:encodeURIComponent(this.value)
            });
            this.value = '';
        }
    }

    chrome.runtime.onMessage.addListener(function(message, sender, callback){
        if(message.action == 'receive'){
            var el = document.createElement('div');
            el.innerText = decodeURIComponent(message.msg);
            document.getElementById('history').appendChild(el);
        }
    });
```

下面是 Local Messager 的运行截图。

Local Messager

本节讲解的应用源码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/local_messager 下载得到。

9.2 TCP 协议

TCP 是一种面向连接的、可靠的、基于字节流的传输层通信协议。

Chrome 提供 sockets.tcp 接口使 Chrome 应用可以进行 TCP 通信。要使用 sockets.tcp 接口需要在 sockets 域中声明 tcp 权限：

```
"sockets": {
    "tcp": {
        "connect": ["192.168.1.100:80", ":8080"]
    }
}
```

上面的代码表示应用可以通过 TCP 与 192.168.1.100 的 80 端口和任意主机的 8080 端口通信。

9.2.1 建立与关闭连接

创建 socket

```
var socketOption = {
    persistent: true,
    name: 'tcpSocket',
```

```
    bufferSize: 4096
};

chrome.sockets.tcp.create(socketOption, function(socketInfo){
    //We'll do the next step later
});
```

socketOption 是一个可选参数，其中 persistent 代表 Chrome 应用的生命周期结束后（参见 6.4 节）这个 socket 是否还依然处于开启状态。bufferSize 定义 socket 接收数据的缓存区大小，默认值是 4096。

当 socket 创建完毕就返回一个对象，这个对象包含代表这个 socket 的唯一 id，之后对 socket 的操作都将根据这个 id。

更新 socket 属性

此处说的属性是指创建 socket 时提到的 socketOptions。通过 update 方法可以更新 socket 的属性：

```
chrome.sockets.tcp.update(socketId, newSocketOption, function(){
    //do something when update complete
});
```

阻止和解除阻止 socket 接收数据

当一个 socket 被阻止后，将不会触发消息接收事件，解除阻止后将恢复正常。

```
//Blocking socket receiving data
var isPaused = true;

chrome.sockets.tcp.setPaused(socketId, isPaused, function(){
    //do something after pause a socket
});
```

如果想解除阻止，将上述代码中的 isPaused 设为 false 即可。

长连接

```
chrome.sockets.tcp.setKeepAlive(socketId, enable, delay, function(code){
    //if a negative value is returned, an error occurred
    //otherwise do something after set keep-alive
});
```

当 enable 设为 true 时启用保持连接功能，delay 定义了最后接收的数据包与第一次探测之间的时间，以秒为单位。

禁用和启用纳格算法

纳格算法以减少封包传送量来增进 TCP/IP 网络的效能。可以通过 setNoDelay 方法禁用或启用。

```
chrome.sockets.tcp.setNoDelay(socketId, noDelay, function(code){
    //if a negative value is returned, an error occurred
    //otherwise do something after set no delay
});
```

当 noDelay 设为 true 时设置 TCP 连接的 TCP_NODELAY 标志，禁用纳格算法。
断开连接

```
chrome.sockets.tcp.disconnect(socketId, function(){
    //do something after disconnect a connection
});
```

关闭 socket
当一个 socket 不再被使用了我们应该关闭它。

```
chrome.socket.tcp.close(socketId, function(){
    //do something after close a socket
});
```

下面我们来试着封装一个 tcp 类，并在之后的内容逐步扩充它：

```
function tcp(){
    var _tcp = chrome.sockets.tcp;
    this.option = {},
    this.socketId = 0,

    this.create = function(callback){
        _tcp.create(this.option, function(socketInfo){
            this.socketId = socketInfo.socketId;
            callback();
        }.bind(this));
    }.bind(this),

    this.update = function(){
        _tcp.update(this.socketId, newSocketOption, callback);
    }.bind(this),

    this.pause = function(isPaused, callback){
        _tcp.setPaused(this.socketId, isPaused, callback);
    }.bind(this),

    this.keepAlive = function(enable, delay, callback){
        _tcp.setKeepAlive(this.socketId, enable, delay, function(code){
            if(code<0){
                this.error(code);
            }
            else{
                callback();
            }
        }.bind(this));
    }.bind(this),
```

```
    this.noDelay = function(noDelay, callback){
        _tcp.setNoDelay(this.socketId, noDelay, function(code){
            if(code<0){
                this.error(code);
            }
            else{
                callback();
            }
        }.bind(this));
    }.bind(this),

    this.disconnect = function(callback){
        _tcp.disconnect(this.socketId, callback);
    }.bind(this),

    this.close = function(callback){
        _tcp.close(this.socketId, callback);
    }.bind(this),

    this.error = function(code){
        console.log('An error occurred with code '+code);
    },

    this.init = function(callback){
        this.create(callback);
    }.bind(this)
}
```

调用时指定 socket 属性就可以进行初始化了：

```
var tcpSocket = new tcp();
tcpSocket.option = {
    persistent: true
};
tcpSocket.init(function(){
    //We'll do something after tcp socket init later
});
```

9.2.2 发送与接收数据

连接

```
chrome.sockets.tcp.connect(socketId, peerAddress, peerPort, function(code){
    //if a negative value is returned, an error occurred
```

```
    //otherwise do something after bind a port
});
```

发送数据

Socket 发送的数据类型为 ArrayBuffer，对 ArrayBuffer 不熟悉的读者请参阅 7.6.1 节的内容。

```
chrome.sockets.tcp.send(socketId, data, function(info){
    //if info.resultCode is a negative value, an error occurred
    //otherwise do something after send some data
});
```

其中 data 为 ArrayBuffer 类型的数据。

接收数据

当 socket 接收到数据时，就会触发 onReceive 事件：

```
chrome.socket.tcp.onReceive.addListener(function(info){
    //We'll do something with info later
});
```

返回的 info 是一个对象，包括两个属性：socketId 和 data。其中 data 为 ArrayBuffer 类型数据。

处理异常

当网络出现问题时，会触发 onReceiveError 事件，同时 socket 会被阻断：

```
chrome.sockets.tcp.onReceiveError.addListener(function(info){
    //We'll do something with info later
});
```

与接收数据相似，info 也是一个对象，但包含 socketId 和 resultCode 两个属性。

现在我们来把 8.2.1 中的 tcp 类完善一下，由于篇幅限制，将只写出添加或改动的部分：

```
function tcp(){
    this.connect = function(address, port, callback){
        _tcp.connect(this.socketId, address, port, function(){
            _tcp.onReceive.addListener(function(info){
                if(info.socketId==this.socketId){
                    this.receive(info);
                }
            }.bind(this));
            _tcp.onReceiveError.addListener(function(info){
                if(info.socketId==this.socketId){
                    this.error(info.resultCode);
                }
            }.bind(this));
        }.bind(this));
    }.bind(this),
```

```
    this.send = function(data, callback){
        _tcp.send(this.socketId, data, callback);
    }.bind(this),

    this.receive = function(info){
        console.log('Received data.');
    }
}
```

9.2.3 获取 socket

获取指定 socket

```
chrome.sockets.tcp.getInfo(socketId, function(socketInfo){
    //do something with socketInfo
});
```

socketInfo 是一个描述 socket 信息的对象，包括 socketId、name、bufferSize、paused、connected、localAddress、localPort、peerAddress 和 peerPort。

获取全部活动的 socket

```
chrome.sockets.tcp.getSockets(function(socketInfoArray){
    //do something with socketInfoArray
});
```

socketInfoArray 是一个包含一个或多个 socketInfo 对象的数组。

将以上方法加入到 tcp 类中：

```
function tcp(){
    this.getInfo = function(callback){
        _tcp.getInfo(this.socketId, callback);
    }.bind(this),

    this.getSockets = function(callback){
        _tcp.getSockets (callback);
    }.bind(this)
}
```

9.3 TCP Server

TCP Server 可以绑定指定端口并被动地接收信息。

Chrome 提供 sockets.tcpServer 接口使 Chrome 应用可以作为 TCP 服务器。要使用 sockets.tcpServer 接口需要在 sockets 域中声明 tcpServer 权限：

```
"sockets": {
    "tcpServer": {
        "listen": ":80"
    }
}
```

上面的代码表示应用可以通过监听本地 80 端口接收 TCP 消息。

9.3.1 建立与关闭连接

创建 socket

```
var socketOption = {
    persistent: true,
    name: 'tcpSocket'
};

chrome.sockets.tcpServer.create(socketOption, function(socketInfo){
    //We'll do the next step later
});
```

socketOption 是一个可选参数，其中 persistent 代表 Chrome 应用的生命周期结束后（参见 6.4 节）这个 socket 是否还依然处于开启状态。

当 socket 创建完毕就返回一个对象，这个对象包含代表这个 socket 的唯一 id，之后对 socket 的操作都将根据这个 id。

更新 socket 属性

此处说的属性是指创建 socket 时提到的 socketOptions。通过 update 方法可以更新 socket 的属性：

```
chrome.sockets.tcpServer.update(socketId, newSocketOption, function(){
    //do something when update complete
});
```

阻止和解除阻止 socket 接收数据

当一个 socket 被阻止后，将不会触发消息接收事件，解除阻止后将恢复正常。

```
//Blocking socket receiving data
var isPaused = true;

chrome.sockets.tcpServer.setPaused(socketId, isPaused, function(){
    //do something after pause a socket
});
```

如果想解除阻止，将上述代码中的 isPaused 设为 false 即可。

断开连接

```
chrome.sockets.tcpServer.disconnect(socketId, function(){
    //do something after disconnect a connection
```

```
});
```

关闭 socket
当一个 socket 不再被使用了我们应该关闭它。

```
chrome.socket.tcpServer.close(socketId, function(){
    //do something after close a socket
});
```

下面我们来试着封装一个 tcpServer 类，并在之后的内容逐步扩充它：

```
function tcpServer(){
    var _tcpServer = chrome.sockets.tcpServer;
    this.option = {},
    this.socketId = 0,

    this.create = function(callback){
        _tcpServer.create(this.option, function(socketInfo){
            this.socketId = socketInfo.socketId;
            callback();
        }.bind(this));
    }.bind(this),

    this.update = function(){
        _tcpServer.update(this.socketId, newSocketOption, callback);
    }.bind(this),

    this.pause = function(isPaused, callback){
        _tcpServer.setPaused(this.socketId, isPaused, callback);
    }.bind(this),

    this.disconnect = function(callback){
        _tcpServer.disconnect(this.socketId, callback);
    }.bind(this),

    this.close = function(callback){
        _tcpServer.close(this.socketId, callback);
    }.bind(this),

    this.init = function(callback){
        this.create(callback);
    }.bind(this)
}
```

调用时指定 socket 属性就可以进行初始化了：

```
var tcpSocket = new tcpServer();
```

```
tcpSocket.option = {
    persistent: true
};
tcpSocket.init(function(){
    //We'll do something after tcpSocket socket init later
});
```

9.3.2 监听数据

监听端口

```
chrome.sockets.tcpServer.listen(socketId, address, port, backlog, function(code){
    //if a negative value is returned, an error occurred
    //otherwise do something after listen complete
});
```

address 和 port 分别是监听本地的地址和端口，此处的端口不要使用 0，因为这样并不知道具体监听的端口。

接受连接

```
chrome.sockets.tcpServer.onAccept.addListener(function(info){
    //do something when a connection has been made to the server socket
});
```

其中 info 包含两个属性，socketId 和 clientSocketId。clientSocketId 是服务器主动与客户端建立的新 socket 连接，通过操作此 socket 可以向客户端发送数据。clientSocketId 只能使用 chrome.sockets.tcp 接口操作，而不能使用 chrome.sockets.tcpServer 接口。另外 clientSocketId 指向的 socket 处于阻止状态，它并不能接收客户端发回的消息，除非手动执行 setPaused 方法解除阻止。

处理异常

当网络出现问题时，会触发 onAcceptError 事件，同时 socket 会被阻断：

```
chrome.sockets.tcpServer.onAcceptError.addListener(function(info){
    //do something with info
});
```

与监测连接相似，info 也是一个对象，但包含 socketId 和 resultCode 两个属性。

现在我们来把 8.3.1 中的 tcpServer 类完善一下，由于篇幅限制，将只写出添加或改动的部分：

```
function tcpServer(){
    this.listen = function(address, port, callback){
        _tcpServer.listen(this.socketId, address, port, function(code){
            if(code<0){
                this.error(code);
                return false;
            }
```

```
                else{
                    _tcpServer.onAccept.addListener(function(info){
                        if(info.socketId==this.socketId){
                            this.accept(info);
                        }
                    }.bind(this));
                    _tcpServer.onAcceptError.addListener(function(info){
                        if(info.socketId==this.socketId){
                            this.error(info.resultCode);
                        }
                    }.bind(this));
                    callback();
                }
            }.bind(this));
        }.bind(this),

        this.error = function(code){
            console.log('An error occurred with code '+code);
        },

        this.accept = function(info){
            console.log('New connection.');
        }
    }
```

注意，上面的代码中_tcpServer.listen 后的参数没有 backlog，这不是笔误。因为 backlog 是可选参数，如果不指定会由系统自动管理。系统会设定一个保证大多数程序正常运行的值，所以不需要手动设定。

9.3.3 获取 socket

获取指定 socket

```
chrome.sockets.tcpServer.getInfo(socketId, function(socketInfo){
    //do something with socketInfo
});
```

socketInfo 是一个描述 socket 信息的对象，包括 socketId、name、paused、persistent、localAddress 和 localPort。

获取全部活动的 socket

```
chrome.sockets.tcpServer.getSockets(function(socketInfoArray){
    //do something with socketInfoArray
});
```

socketInfoArray 是一个包含一个或多个 socketInfo 对象的数组。

将以上方法加入到 tcpServer 类中：

```
function tcpServer(){
    this.getInfo = function(callback){
        _tcpServer.getInfo(this.socketId, callback);
    }.bind(this),

    this.getSockets = function(callback){
        _tcpServer.getSockets (callback);
    }.bind(this)
}
```

9.3.4 HTTP Server

通过 8.3.2 和 8.3.3 两小节的内容我们了解的 TCP 在 Chrome 应用中的使用，本小节将实战编写一个 HTTP 服务器。

HTTP 服务器需要监听 TCP 连接同时使用 TCP 与客户端进行通信，所以需要 tcp 和 tcpServer 权限：

```
{
    "app": {
        "background": {
            "scripts": ["tcp.js", "tcpServer.js", "background.js"]
        }
    },
    "manifest_version": 2,
    "name": "HTTP Server",
    "version": "1.0",
    "description": "An HTTP server.",
    "icons": {
        "128": "http_server.png"
    },
    "sockets": {
        "tcp": {
            "connect": "*"
        },
        "tcpServer": {
            "listen": ":80"
        }
    }
}
```

其中 tcp.js 和 tcpServer.js 就是前两节中我们写的两个类。下面来写 background.js。

首先需要创建 tcpServerSocket：

```
var tcpServerSocket = new tcpServer();
```

```
tcpServerSocket.option = {
    persistent: true
};
tcpServerSocket.accept = handleAccept.bind(tcpServerSocket);
tcpServerSocket.init(function(){
    tcpServerSocket.listen('127.0.0.1', 80, function(){
        console.log('Listening 127.0.0.1:80...');
    });
});
```

创建完成后来编写监听连接的函数 handleAccept：

```
function handleAccept(info){
    if(info.socketId==this.socketId){
        var _tcp = chrome.sockets.tcp;
        var tcpSocket = new tcp();
        tcpSocket.socketId = info.clientSocketId;
        tcpSocket.keepAlive(true, 5, function(){
            _tcp.onReceive.addListener(function(info){
                if(info.socketId==tcpSocket.socketId){
                    tcpSocket.receive(info);
                }
            });
            _tcp.onReceiveError.addListener(function(info){
                if(info.socketId==tcpSocket.socketId){
                    tcpSocket.error(info.resultCode);
                }
            });
            tcpSocket.receive = handleRequest.bind(tcpSocket);
            tcpSocket.pause(false, function(){
                console.log('Receiving data...');
            });
        });
    }
}
```

在 handleAccept 函数中，我们获取到 clientSocketId 后创建了一个新的 tcp 对象 tcpSocket，并将 clientSocketId 的值赋给了 tcpSocket 的 socketId，这样对 tcpSocket 的操作就是对 clientSocketId 指向的 TCP socket 了。

之后设定了 keep-alive 属性，并解除了此 TCP socket 的阻止状态开始接收数据。接收到的数据通过 handleRequest 函数来处理。下面来编写 handleRequest 函数：

```
function handleRequest(info){
    var header = ab2str(info.data);
    header = header.split("\r\n").join('<br />');
    var body = "<h1>It Works!</h1>"+
```

```
                    "<hr />"+
                    "Request Header:<br />"+header;
    var respondse = "HTTP/1.1 200 OK\r\n"+
                        "Connection: Keep-Alive\r\n"+
                        "Content-Length: "+body.length+"\r\n"+
                        "Content-Type: text/html\r\n"+
                        "Connection: close\r\n\r\n"+body;
    respondse = str2ab(respondse);
    this.send(respondse, function(){
        console.log('Sent.');
        this.close(function(){
            console.log('Closed.');
        })
    }.bind(this));
}
```

handleRequest 函数将得到的用户请求的 HTTP 头展示给用户，同时在最前端显示“It Works!”。

最后来编写 ArrayBuffer 和字符串直接转换的函数，本例中使用到的转换函数与 8.1.5 节中所使用的略有不同：

```
function str2ab(str){
    var buf = new ArrayBuffer(str.length);
    bufView = new Uint8Array(buf);
    for(var i=0; i<str.length; i++){
        bufView[i] = str.charCodeAt(i);
    }
    return buf;
}

function ab2str(buf){
    return String.fromCharCode.apply(null, new Uint8Array(buf));
}
```

在 8.1.5 节中我们使用的是 Uint16Array 这个 ArrayBufferView，因为在 JavaScript 中一个字符占 16 位，所以应用两个字节来储存，但在 HTTP 协议中一个字符占 8 位，所以要用 1 个字节来储存，这点请读者注意。

HTTP Server 运行截图

本节讲到的应用源码可以通过 https://github.com/sneezry/chrome_extensions_and_apps_programming/tree/master/http_server 下载。

9.4 WebSocket

WebSocket 与本章前三节介绍的内容不同，它是 HTML5 原生支持的功能。使用 WebSocket 需要外部服务器的支持。

在使用 WebSocket 通信前需要先连接到外部的 WebSocket 服务器：

```
var connection = new WebSocket('ws://127.0.0.1');
```

当连接打开后会触发 onopen 事件：

```
connection.onopen = function(){
    //do something when the connection is open
}
```

向服务器发送数据使用 send 方法：

```
connection.send(data);
```

其中 data 可以是 ArrayBuffer、Blob 或字符串。

当接收到来自服务器的数据时会触发 onmessage 事件：

```
connection.onmessage = function(result){
    //do something with result
}
```

其中 result 是一个对象，可以通过 result.data 访问数据。如果传送的数据是二进制数据，还可以通过 result.data.byteLength 和 result.data.binaryType 获取二进制数据的长度和类型（ArrayBuffer 或 Blob）。

当 WebSocket 连接发生异常时会触发 onerror 事件：

```
connection.onerror = function(error){
    console.log(error);
};
```

更多有关 WebSocket 的内容可以参见 http://www.html5rocks.com/zh/tutorials/websockets/basics/。

第 10 章　其他接口

除上述接口外 Chrome 应用还有其他各类丰富的接口，在本章将对其他的接口做以介绍。

10.1 操作 USB 设备

通过 usb 接口可以与 USB 设备进行交互，这能让 Chrome 应用作为 USB 设备的驱动程序。要使用 usb 接口需要在 Manifest 中声明 usb 权限：

```
"permissions": [
    "usb"
]
```

本章内容参考自 https://crxdoc-zh.appspot.com/apps/usb。

10.1.1 发现设备

列出指定的 USB 设备：

```
var options = {
    vendorId: 0x05ac,    //Apple, Inc.
    productId: 0x12a0    //iPhone 4s
};

chrome.usb.getDevices(options, function(deviceArray){
    //do something with deviceArray;
});
```

请求访问操作系统占用的设备（仅限 Chrome OS）：

```
chrome.usb.requestAccess(device, interfaceId, function(sucess){
    //sucess is boolean
});
```

其中 interfaceId 为接口标识符。

打开设备：

```
chrome.usb.openDevice(device, function(ConnectionHandle){
    //do something with handle
});
```

ConnectionHandle 为连接句柄，包括三个属性，handle、vendorId 和 productId，其中 handle 为连接句柄的标识符。

寻找指定的 USB 设备（如果权限允许的话同时打开设备以便使用）：

```
var options = {
    vendorId: 0x05ac,    //Apple, Inc.
    productId: 0x12a0    //iPhone 4s
    interfaceId: null    //only for Chrome OS
};
```

```
chrome.usb.findDevices(options, function(ConnectionHandleArray){
    //do something with ConnectionHandleArray
});
```

关闭连接句柄：

```
chrome.usb.closeDevice(ConnectionHandle, function(){
    //do something after close a device
});
```

重置 USB 设备：

```
chrome.usb.resetDevice(ConnectionHandle, function(result){
    //result is boolean
});
```

10.1.2 操作接口

列举 USB 设备上的所有接口：

```
chrome.usb.listInterfaces(ConnectionHandle, function(descriptorsArray){
    //do something with descriptorsArray
});
```

descriptorsArray 是一个包含多个 descriptors 对象的数组，descriptors 包含的属性有 interfaceNumber、 alternateSetting、 interfaceClass、 interfaceSubclass、 interfaceProtocol、 description 和 endpoints。endpoints 也是一个数组，每个对象包含的属性有 address、type、direction、maximumPacketSize、synchronization、usage 和 pollingInterval。

在指定 USB 设备上获取接口：

```
chrome.usb.claimInterface(ConnectionHandle, interfaceNumber, function(){
    //do something after the interface is claimed
});
```

释放在提供的设备上获取的接口：

```
chrome.usb.releaseInterface(ConnectionHandle, interfaceNumber, function(){
    //do something after the interface is released
});
```

在之前获取的设备接口上选择替代的设置：

```
chrome.usb.setInterfaceAlternateSetting(ConnectionHandle, interfaceNumber, alternateSetting, function(){
    //do something after the interface setting is set
});
```

alternateSetting 为要设置的替代设置。

10.1.3 操作传输

在指定设备上进行控制传输：

```
chrome.usb.controlTransfer(ConnectionHandle, transferInfo, function(info){
    //do something with info
});
```

其中 transferInfo 为对象，包含的属性有 direction、recipient、requestType、request、value、index、length 和 data。如果输出数据，transferInfo.data 必须指定，值的类型为 ArrayBuffer。

info 为对象，包含的属性有 resultCode 和 data，resultCode 为 0 时表示成功，data 为传入数据，类型为 ArrayBuffer。

在指定设备上进行大块传输：

```
chrome.usb.bulkTransfer(ConnectionHandle, transferInfo, function(info){
    //do something with info
});
```

transferInfo 为对象，包含的属性有 direction、endpoint、length 和 data。此方法返回的 info 类型与 controlTransfer 相同。

在指定设备上进行中断传输：

```
chrome.usb.interruptTransfer(ConnectionHandle, transferInfo, function(info){
    //do something with info
});
```

此方法各参数类型与 bulkTransfer 相同。

在指定设备上进行同步传输：

```
chrome.usb.isochronousTransfer(ConnectionHandle, transferInfo, function(info){
    //do something with info
});
```

此方法各参数类型与 bulkTransfer 相同。

10.2 串口通信

通过 serial 接口可以使 Chrome 应用进行串口通信。使用 serial 接口需要在 Manifest 中声明 serial 权限：

```
"permissions": [
    "serial"
]
```

本章内容参考自 https://crxdoc-zh.appspot.com/apps/serial。

10.2.1 建立连接

获取系统中可用串行设备的有关信息：

```
chrome.serial.getDevices(function(portsArray){
    //do something with portsArray
});
```

portsArray 为包含多个 ports 的数组。每个 ports 是一个包含多个属性的对象，其属性有 path、vendorId、productId 和 displayName。

连接到指定的串行端口：

```
chrome.serial.connect(path, options, function(connectionInfo){
    //do something with connectionInfo
});
```

其中 options 为端口配置选项，完整的结构如下：

```
{
    persistent: 应用关闭时连接是否保持打开状态,
    name: 与连接相关联的字符串,
    bufferSize: 用于接收数据的缓冲区大小,
    bitrate: 打开连接时请求的比特率,
    dataBits: 默认为"eight",
    parityBit: 默认为"no",
    stopBits: 默认为"one",
    ctsFlowControl: 是否启用 RTS/CTS 硬件流控制,
    receiveTimeout: 等待新数据的最长时间，以毫秒为单位,
    sendTimeout: 等待 send 操作完成的最长时间，以毫秒为单位
}
```

更新打开的串行端口连接的选项设置：

```
chrome.serial.update(connectionId, options, function(result){
    //result is boolean
});
```

断开串行端口连接：

```
chrome.serial.disconnect(connectionId, function(result){
    //result is boolean
});
```

暂停或恢复打开的连接：

```
chrome.serial.setPaused(connectionId, paused, function(){
    //do something after pause a connection
});
```

10.2.2 发送和接收数据

向指定连接写入数据：

```
chrome.serial.send(connectionId, data, function(sendInfo){
    //do something with sendInfo
});
```

sendInfo 为一个包含两个属性的对象，分别为 bytesSent 和 error。
清除指定连接输入输出缓存中的所有内容：

```
chrome.serial.flush(connectionId, function(result){
    //result is boolean
});
```

当接收到数据时，会触发 onReceive 事件：

```
chrome.serial.onReceive.addListener(function(info){
    //do something with info
});
```

info 是一个包含两个属性的对象，分别是 connectionId 和 data。
当发生错误时，会触发 onReceiveError 事件：

```
chrome.serial.onReceiveError.addListener(function(info){
    //do something with info
});
```

info 为一个包含两个属性的对象，分别为 connectionId 和 error。

10.2.3 获取连接及状态

获取指定连接的状态：

```
chrome.serial.getInfo(connectionId, function(connectionInfo){
    //do something with connectionInfo
});
```

获取当前应用拥有并打开的串行端口连接列表：

```
chrome.serial.getConnections(function(connectionInfoArray){
    //do something with connectionInfoArray
});
```

获取指定连接上控制信号的状态：

```
chrome.serial.getControlSignals(connectionId, function(signals){
    //do something with signals
});
```

signals 是一个包含 4 个属性的对象，分别是 dcd、cts、ri 和 dsr。
设置指定连接上控制信号的状态：

```
chrome.serial.setControlSignals(connectionId, signals, function(result){
    //result is boolean
});
```

signals 参数为一个包含两个属性的对象，分别是 dtr 和 rts。

10.3 文字转语音

使用 tts 接口可以将文字转换为语音，tts 接口可以使用不同语速、音调阅读文字。文字转语音对视力不佳的用户来说非常重要。

要在应用中使用 tts 接口，需要在 Manifest 的 permissions 中声明 tts 权限：

```
"permissions": [
    "tts"
]
```

10.3.1 朗读文字

使用 speak 方法来朗读文字：

```
chrome.tts.speak('Hello, world.');
```

speak 方法还可以指定朗读参数和回调函数：

```
chrome.tts.speak(utterance, options, callback);
```

回调函数 callback 会在 speak 方法调用成功后立刻执行，这意味着不会等到朗读结束后才调用 callback。options 指定了朗读时所采用的语调、语速、音量等等，options 完整的结构如下所示：

```
{
    enqueue: 是否将朗读任务放入队列，如果为 true，此朗读任务将在之前的任务结束后才开始,
    voiceName: 朗读所使用的声音名称,
    extensionId: 为朗读提供声音引擎扩展的 id,
    lang: 所朗读文字的语言,
    gender: 朗读声音所使用的性别，male 或 female,
    rate: 朗读语速，默认值为 1.0，允许的值为 0.1 到 10.0，但具体范围还要结合具体使用的声音，值越大速度越快,
    pitch: 朗读语调，默认值为 1.0，允许的值为 0 到 2.0,
    volume: 朗读音量，默认值为 1.0，允许的值为 0 到 1.0,
    requiredEventTypes: 声音必须支持的事件,
    desiredEventTypes: 需要监听的事件，如未指定则监听全部事件,
    onEvent: 用于监听事件的函数
}
```

如下面的例子，用美式英文以正常语速的 2 倍阅读“Hello, world.”：

```
chrome.tts.speak('Hello, world.', {lang: 'en-US', rate: 2.0});
```

而下一个例子会在“Speak this first.”阅读完毕后才阅读“Speak this next, when the first sentence is done.”：

```
chrome.tts.speak('Speak this first.');
chrome.tts.speak('Speak this next, when the first sentence is done.', {enqueue: true});
```

使用 chrome.runtime.lastError 来抓取 tts 接口使用中可能发生的错误：

```
chrome.tts.speak(utterance, options, function() {
    if (chrome.runtime.lastError) {
        console.log('Error: ' + chrome.runtime.lastError.message);
    }
});
```

使用 stop 方法可以随时停止正在进行的朗读任务：

```
chrome.tts.stop();
```

使用 pause 方法可以随时暂停正在进行的朗读任务：

```
chrome.tts.pause();
```

使用 resume 方法可以随时恢复被暂停的朗读任务：

```
chrome.tts.resume();
```

10.3.2 获取声音

通过 getVoices 方法可以获取到目前计算机中提供的声音：

```
chrome.tts.getVoices(function(voices){
    //do something with voices
});
```

返回的结果 voices 是一个包含多个声音对象的数组。声音对象包含 6 个属性，分别是 voiceName、lang、gender、remote、extensionId 和 eventTypes，其中 remote 属性表示此声音是否为网络资源，eventTypes 为此声音支持的全部事件。

如下面为一个声音对象的实例：

```
{
    "eventTypes": [
        "start",
        "end",
        "interrupted",
        "cancelled",
        "error"
```

```
        ],
        "extensionId": "neajdppkdcdipfabeoofebfddakdcjhd",
        "gender": "female",
        "lang": "en-GB",
        "remote": true,
        "voiceName": "Google UK English Female"
    }
```

获取到声音对象后，通过指定 speak 方法中的相应参数来应用声音，如：

```
chrome.tts.speak(utterance, {
    voiceName: 'Google UK English Female',
    lang: 'en-GB'
}, callback);
```

10.3.3 获取朗读状态及监听事件

如果当前应用正在朗读文本，执行一个新的朗读任务会立即停止之前的朗读任务。为了避免打断正在进行的朗读任务，可以通过 isSpeaking 方法获取当前的朗读状态：

```
chrome.tts.isSpeaking(function(isSpeaking){
    //isSpeaking is boolean
});
```

在 10.3.1 中提到 speak 参数的 onEvent 属性用来监听朗读事件：

```
chrome.tts.speak(utterance,{
    onEvent: function(event) {
        console.log('Event ' + event.type ' at position ' + event.charIndex);
        if (event.type == 'error') {
            console.log('Error: ' + event.errorMessage);
        }
    }
}, callback);
```

其中 event 对象包含三个属性，分别是 type、charIndex 和 errorMessage。type 为事件类型，可能的值包括 start、end、word、sentence、marker、interrupted、cancelled、error、pause 和 resume。

朗读任务一开始就会监听到 start 类型事件，当朗读到一个新的词语时会监听到 word 类型事件，朗读完一个句子时会监听到 sentence 类型事件，当朗读任务被中断会监听到 interrupted 类型事件，而如果朗读任务尚未开始即被移除会监听到 cancelled 类型事件，error、pause 和 resume 类型事件分别会在朗读过程中遇到错误、被暂停和被恢复时接收到。

对于 marker 类型事件，它是在朗读任务到达 SSML 标记时触发的，有关 SSML 的详细介绍请读者自行参考 http://www.w3.org/TR/speech-synthesis/，此处不做详细介绍。

不过实际能接收到的类型事件需要根据具体选择的声音的支持情况，在 10.3.2 中获取到的声音对象 eventTypes 属性列出了相应声音支持的全部事件类型。

10.4 系统信息

在讲解 Chrome 扩展时我们提到过获取 CPU、内存和存储设备信息的方法，具体可以参见 5.4 节。Chrome 应用也可以获取到系统信息，并且与 Chrome 扩展类似。

Chrome 应用可以获取到的系统信息包括 CPU、内存、存储设备、显示器和网卡。要获取信息，需要在 Manifest 中声明相应权限：

```
"permissions": [
    "system.cpu",
    "system.memory",
    "system.storage",
    "system.display",
    "system.network"
]
```

由于 CPU、内存和存储设备相关的内容已经在 5.4 节讲过了，所以本节只讲解显示器和网卡的内容。

通过 chrome.system.display.getInfo 方法可以获取到显示器相关信息：

```
chrome.system.display.getInfo(function(displayInfoArray){
    //do something with displayInfoArray
});
```

displayInfoArray 是一个包含多个 displayInfo 对象的数组。displayInfo 对象的完整结构如下：

```
{
    id: 显示器的唯一 id,
    name: 显示器名称，如 HP LCD monitor,
    mirroringSourceId: 镜像的显示器 id，目前只支持 Chrome OS 平台,
    isPrimary: 是否为主显示器,
    isInternal: 是否为笔记本的自带显示器,
    isEnabled: 是否被启用,
    dpiX: 显示器水平 DPI,
    dpiY: 显示器垂直 DPI,
    rotation: 显示器旋转角度，目前只支持 Chrome OS 平台,
    bounds: {
        left: 显示器逻辑范围左上角横坐标,
        top: 显示器逻辑范围左上角纵坐标,
        width: 显示器逻辑范围像素宽度,
        height: 显示器逻辑范围像素高度
    },
    overscan: {
        left: 显示范围距左边框的距离,
```

```
            top: 显示范围距上边框的距离,
            right: 显示范围距右边框的距离,
            bottom: 显示范围距下边框的距离
        },
        workArea: {
            left: 工作区范围左上角横坐标,
            top: 工作区范围左上角纵坐标,
            width: 工作区范围像素宽度,
            height: 工作区范围像素高度
        }
    }
```

其中 overscan 属性目前只在 Chrome OS 平台有效，workArea 的范围不包括系统占用部分，如任务栏等。

通过 chrome.system.display.setDisplayProperties 方法可以更改显示器设置，支持更改的属性包括 mirroringSourceId、isPrimary、overscan、rotation、boundsOriginX 和 boundsOriginY。boundsOriginX 和 boundsOriginY 对应于 bounds.left 和 bounds.top，即显示器逻辑范围的原点坐标。

```
chrome.system.display.setDisplayProperties(id, info, function(){
    //do something after set display properties
});
```

通过 chrome.system.display.onDisplayChanged 监听显示器设置更改事件：

```
chrome.system.display.onDisplayChanged.addListener(function(){
    //do something after display properties are changed
});
```

通过 chrome.system.network.getNetworkInterfaces 方法可以获取到网卡信息：

```
chrome.system.network.getNetworkInterfaces(function(networkInterfaces){
    //do something with networkInterfaces
    console.log(networkInterfaces);
});
```

networkInterfaces 是一个包含多个 networkInterface 对象的数组，networkInterface 对象包含 3 个属性，分别是 name、address 和 prefixLength。name 为网卡的名称，在*nix 系统上通常是 eth0 或 wlan0 等。address 为网卡可用的 IPv4 或 IPv6 地址。prefixLength 为前缀长度。

附录 A　制作 Chrome 主题

Chrome 主题与扩展和应用的结构类似，包含一个 Manifest 文件和一些图片资源。主题的 Manifest 结构如下：

```
{
```

```
    "version": "2.6",
    "name": "camo theme",
    "theme": {
        "images" : {
            "theme_frame" : "images/theme_frame_camo.png",
            "theme_frame_overlay" : "images/theme_frame_stripe.png",
            "theme_toolbar" : "images/theme_toolbar_camo.png",
            "theme_ntp_background" : "images/theme_ntp_background_norepeat.png",
            "theme_ntp_attribution" : "images/attribution.png"
        },
        "colors" : {
            "frame" : [71, 105, 91],
            "toolbar" : [207, 221, 192],
            "ntp_text" : [20, 40, 0],
            "ntp_link" : [36, 70, 0],
            "ntp_section" : [207, 221, 192],
            "button_background" : [255, 255, 255]
        },
        "tints" : {
            "buttons" : [0.33, 0.5, 0.47]
        },
        "properties" : {
            "ntp_background_alignment" : "bottom"
        }
    }
}
```

颜色使用 RGB 格式，即[r, g, b]。图片路径使用基于主题包根路径的相对路径。properties 定义了图片的位置和 repeat 属性。tints 定义了按钮、框架和后台标签页等某些部分的色调，使用 HSL 格式，取值范围为 0 到 1 的浮点型数据。

更详细的内容可以参见 https://code.google.com/p/chromium/wiki/ThemeCreationGuide。

附录 B　i18n

使用 i18n 接口实现扩展应用程序的国际化。本节内容部分参考 https://crxdoc-zh.appspot.com/apps/i18n。

在扩展应用程序的根目录下创建_locales 文件夹，在_locales 下以每个支持的语言的代码为名称创建子文件夹，然后在其中放入 message.json 指定对应语言的字符串。支持的语言和对应的语言代码参加 https://developer.chrome.com/webstore/i18n#localeTable。

```
root directiory
|- manifest.json
|- *.html, *.js
```

```
|- _locales
    |- en
        |- message.json
    |- zh-CN
        |- message.json
```

在 message.json 指定字符串：

```
{
    "extName": {
        "message": "一个国际化扩展",
        "description": "Extension Name"
    },
    ...
}
```

在 Manifest 中调用国际化字符串：

```
{
    "name": "__MSG_extName__",
    "default_locale": "en",
    ...
}
```

在 JavaScript 中获取国际化字符串：

```
title = chrome.i18n.getMessage('extName');
```

有一些预定的国际化字符串：

@@extension_id：扩展应用 id
@@ui_locale：当前语言
@@bidi_dir：当前语言的文字方向，ltr 或 rtl
@@bidi_reversed_dir：如果@@bidi_dir 是 ltr 则该消息为 rtl，否则为 ltr
@@bidi_start_edge：如果@@bidi_dir 是 ltr 则该消息为 left，否则为 right
@@bidi_end_edge：如果@@bidi_dir 是 ltr 则该消息为 right，否则为 left

上面这些预定义字符串可以直接在 JavaScript 和 CSS 中引用，如：

```
body {
    direction: __MSG_@@bidi_dir__;
}

div#header {
    margin-bottom: 1.05em;
    overflow: hidden;
    padding-bottom: 1.5em;
    padding-__MSG_@@bidi_start_edge__: 0;
    padding-__MSG_@@bidi_end_edge__: 1.5em;
```

```
    position: relative;
}
```

获取浏览器可接受的语言：

```
chome.i18n.getAcceptLanguages(function(languageArray){
    //do something with languageArray
});
```

获得指定消息的本地化字符串。如果消息不存在，该方法返回空字符串""：

```
var msg = chrome.i18n.getMessage(messageName, substitutions);
```

获取浏览器用户界面的语言 1：

```
var currentLanguage = chrome.i18n.getUILanguage();
```

1 从 Chrome 35 开始支持。

附录 C　初识 AngularJS

AngularJS 是一套符合 MVC 架构的 JavaScript 框架，由 Google 提出与维护。AngularJS 旨在简化 Web App 的开发与测试，它以 HTML 作为视图，并将使用 JavaScript 编写的模型与其绑定在一起。AngularJS 的官方网站为 https://angularjs.org/，可以在上面找到最新版的代码和完整的文档。

附录 C 的结构和内容参考了 http://www.youtube.com/watch?v=i9MHigUZKEM。

C.1　视图

AngularJS 使用 HTML 作为视图，HTML 就相当于 MVC 中的 V。在<html>标签中表明 ng-app 属性来声明此 HTML 为 AngularJS 视图：

```
<html ng-app>
...
</html>
```

但 ng-app 属性有时会让 HTML5 解释器报错，因为它不是一个标准的属性，如果你希望遵循更加严格的 HTML5 标准，可以将 ng-app 改写成 data-ng-app，两者的效果是相同的：

```
<html data-ng-app>
...
</html>
```

本书为了遵循标准，将全部使用 data-ng-作为前缀，如果读者在其他地方发现使用了 ng-

前缀不要感到困惑。

在视图中使用两个大括号来绑定变量：

```
{{ username }}
```

使用 data-ng-model 属性为元素快捷绑定模型：

```
<input type="text" data-ng-model="username" placeholder="Username" />
```

则一个最简单的 AngularJS 视图就完成了：

```
<html data-ng-app>
<head>
<script src="angular.min.js"></script>
</head>
<body>
<input type="text" data-ng-model="username" placeholder="Username" /> {{ username }}
</body>
</html>
```

这个视图呈现一个输入框，在这个输入框中输入的任何字符都会立即显示在输入框的后面。

一个简单的 AngularJS 视图

除了直接对数据进行引用外，也可以使用循环来输出一个列表，如：

```
<div data-ng-init="pets=['Dog', 'Cat', 'Rabbit', 'Parrot']">
<ul>
<li data-ng-repeat="name in pets">{{ name }}</li>
</ul>
</div>
```

在视图中循环输出数据

下面我们来介绍过滤器。过滤器可以进一步限定展示的数据范围及形式，如下面的例子：

```
<input type="text" data-ng-model="choosenName" />
<div data-ng-init="pets=['Dog', 'Cat', 'Rabbit', 'Parrot']">
<ul>
<li data-ng-repeat="name in pets | filter:choosenName">{{ name }}</li>
</ul>
</div>
```

这样当在文本框中进行输入时，将只列出包含输入字符的名称：

过滤器

还可以通过 orderBy 来指定排列依据，如：

```
<div data-ng-init="pets=[{name: 'Dog'}, {name: 'Cat'}, {name: 'Rabbit'}, {name: 'Parrot'}]">
<ul>
```

```
<li data-ng-repeat="pet in pets | orderBy:'name'">{{ pet.name }}</li>
</ul>
</div>
```

这样最终的显示顺序将与原始数据的顺序无关，会根据给定的属性重新排列：

排列顺序
使用 uppercase 和 lowercase 来限定显示文本大小写格式，如：

```
<div data-ng-init="pets=['Dog', 'Cat', 'Rabbit', 'Parrot']">
<ul>
<li data-ng-repeat="name in pets">{{ name | uppercase }}</li>
</ul>
</div>
```

使用大写格式显示文本

C.2 $scope

$scope 是连接视图与控制器的枢纽。在上一节中我们通过 data-ng-init 指定数据，但如何动态指定数据呢？这就需要$scope 的帮助。

```
function petController($scope){
    $scope.pets = [
        {name: 'Dog'},
        {name: 'Cat'},
        {name: 'Rabbit'},
        {name: 'Parrot'}
    ];
}
```

上面是一个简单的控制器，通过 data-ng-controller 将其与视图绑定：

```
<div data-ng-controller="petController">
...
</div>
```

这样在上述 标签内部就都可以通过 pets 来访问数据了，如：

```
<div data-ng-controller="petController">
<ul>
<li data-ng-repeat="pet in pets | orderBy:'name'">{{ pet.name }}</li>
</ul>
</div>
```

值得注意的是，pets 的作用域仅限于上述 标签内部，在外部是无法访问到的。
$scope 可以有任意多的属性，只要你喜欢， 同时相应的变量也可以在相应的区间使用。

C.3 module 与路由

module 用来初始化应用，通过 angular.module 方法来创建 module，如：

```
angular.module('myApp', []);
```

对应的视图为：

```
<html data-ng-app="myApp">
...
</html>
```

在创建 module 时我们传入了一个空数组[]，这个数组用来指定依赖的其他 module，如：

```
angular.module('myApp', ['helperModule']);
```

如果无需依赖其他 module 则只需给出一个空数组[]即可。可以看出 module 也方便将功能模块化，提高了复用和维护效率。

通过 controller 方法可以动态添加控制器，如：

```
var myApp = angular.module('myApp', []);
myApp.controller('petController', function($scope){
    $scope.pets = [
        {name: 'Dog'},
        {name: 'Cat'},
        {name: 'Rabbit'},
        {name: 'Parrot'}
    ];
});
```

请对比上述代码与 C.2 中的不同。
为使代码更有可读性，我们可以将上述代码改写成如下形式：

```
var myApp = angular.module('myApp', []);
controllers = {};
controllers.petController = function($scope){
    $scope.pets = [
        {name: 'Dog'},
        {name: 'Cat'},
        {name: 'Rabbit'},
        {name: 'Parrot'}
    ];
}
myApp.controller(controllers);
```

通过 config 方法来配置路由，如：

```
var myApp = angular.module('myApp', []);
myApp.config(function($routeProvider){
    $routeProvider
        .when('/', {
            controller: 'petController',
            templateUrl: 'view1.html'
        })
        .when('/price', {
            controller: 'petController',
            templateUrl: 'view2.html'
        })
        .otherwise({redirectTo: '/'});
});
```

在需要动态更改内容的容器中标明 data-ng-view 属性：

```
<div data-ng-view></div>
```

下面是完整的视图：

```
<html data-ng-app="myApp">
<head>
<script src="angular.min.js"></script>
</head>
<div data-ng-view></div>
<a href="#/">Home</a> | <a href="#/price">Price</a>
<script src="module.js"></script>
</html>
```

需要注意的是，AngularJS 在 1.2 版本以后移除了原生对 ngRoute 的支持，这意味着上面的代码在最新的 AngularJS 中并不工作，解决办法是首先到 https://code.angularjs.org/1.2.0rc1/angular-route.js 下载 angular-route.js，并在视图中引用，然后创建 module 时指定依赖于 ngRoute 模块：

```
var myApp = angular.module('myApp', [ngRoute]);
```

下面来编写 module.js 脚本：

```
var myApp = angular.module('myApp', ['ngRoute']);
controllers = {};
controllers.petController = function($scope){
    $scope.pets = [
        {name: 'Dog', price: 200},
        {name: 'Cat', price: 220},
        {name: 'Rabbit', price: 180},
        {name: 'Parrot', price: 240}
```

```
        ];
}
myApp.controller(controllers);
myApp.config(function($routeProvider){
        $routeProvider
                .when('/', {
                        controller: 'petController',
                        templateUrl: 'view1.html'
                })
                .when('/price', {
                        controller: 'petController',
                        templateUrl: 'view2.html'
                })
                .otherwise({redirectTo: '/'});
});
```

最后编写 view1.html 和 view2.html 两个视图，view1.html 视图：

```
<ul>
<li data-ng-repeat="pet in pets">{{ pet.name | uppercase }}</li>
</ul>
```

view2.html 视图：

```
<ul>
<li data-ng-repeat="pet in pets">{{ pet.name }} - ${{ pet.price }}</li>
</ul>
```

最后运行的结果如下，默认加载 view1.html 视图：

默认加载 view1.html 视图
当点击 Price 链接后加载 view2.html 视图：

点击 Price 链接后加载 view2.html 视图
由于动态加载视图是通过 XMLHttpRequest 实现的，所以测试时无法在本地运行 1。
1 使用命令行参数--allow-file-access-from-files 启动 Chrome 浏览器可以解除这一限制。

附录 D Chrome 扩展及应用完整 API 列表

以下内容来自 https://crxdoc-zh.appspot.com。请注意，这些列表中的内容可能会频繁地变化，尤其是 Beta 和 Dev 部分接口甚至可能会在未来的版本中消失，使用时应更为谨慎。最新的列表可以通过官方文档查看，扩展列表参见 https://developer.chrome.com/extensions/api_index ，应用列表参见 https://developer.chrome.com/apps/api_index。

D.1 Chrome 扩展全部 API

稳定 API

名称描述最低版本

alarms
使用 chrome.alarms API 安排代码周期性地或者在将来的指定时间运行。
22

bookmarks
使用 chrome.bookmarks API 创建、组织以及通过其他方式操纵书签。您也可以参见替代页面，通过它您可以创建一个自定义的书签管理器页面。
5

browserAction
使用浏览器按钮可以在 Google Chrome 浏览器主窗口中地址栏右侧的工具栏中添加图标。除了弹出内容。
5

browsingData
使用 chrome.browsingData API 从用户的本地配置文件删除浏览数据。
19

commands
使用命令 API 添加快捷键，触发您的扩展程序中的操作，例如打开浏览器按钮或向扩展程序发送命令。
25

contentSettings
使用 chrome.contentSettings API 更改设置，控制网站能否使用 Cookie、JavaScript 和插件之类的特性。大体上说，内容设置允许您针对不同的站点（而不是全局地）自定义 Chrome 浏览器的行为。
16

contextMenus

使用 chrome.contextMenus API 向 Google Chrome 浏览器的右键菜单添加项目。您可以选择您在右键菜单中添加的项目应用于哪些类型的对象，例如图片、超链接和页面。

6

cookies

使用 chrome.cookies API 查询和修改 Cookie，并在 Cookie 更改时得到通知。

6

debugger

chrome.debugger API 是 Chrome 远程调试协议（英文）的另一种消息传输方式。使用 chrome.debugger 可以附加到一个或多个标签页，以便查看网络交互、调试 JavaScript、改变 DOM 和 CSS 等等。使用调试对象的标签页标识符来指定 sendCommand 的目标标签页，并在 onEvent 的回调函数中通过标签页标识符分发事件。

18

declarativeContent

使用 chrome.declarativeContent API 根据网页内容采取行动，而不需要读取页面内容的权限。

33

desktopCapture

桌面捕获 API 可以用于捕获屏幕、单个窗口或标签页的内容。

34

devtools.inspectedWindow

使用 chrome.devtools.inspectedWindow API 与审查的窗口交互：获得审查页面的标签页标识符，在审查窗口的上下文中执行代码，重新加载页面，或者获取页面中所有资源的列表。

18

devtools.network

使用 chrome.devtools.network API 获取开发者工具的网络面板中显示的与网络请求相关的信息。

18

devtools.panels

使用 chrome.devtools.panels API 将您的扩展程序整合到开发者工具窗口用户界面中：

创建您自己的面板、访问现有的面板以及添加侧边栏。

18

downloads

使用 chrome.downloads API 以编程方式开始下载，监视、操纵、搜索下载的文件。

31

events

chrome.events 命名空间包含 API 分发事件使用的通用类型，以便在某些有意义的事情发生时通知您。

21

extension

chrome.extension API 包含任何扩展程序页面都能使用的实用方法。它包括在扩展程序和内容脚本之间或者两个扩展程序之间交换消息的支持，这一部分内容在消息传递中详细描述。

5

fileBrowserHandler

使用 chrome.fileBrowserHandler API 扩展 Chrome OS 的文件浏览器。例如，您可以使用这一 API 让用户向您的网站上传文件。

12

fontSettings

使用 chrome.fontSettings API 管理 Chrome 浏览器的字体设置。

22

history

使用 chrome.history API 与浏览器的历史记录交互，您可以添加、删除、通过 URL 查询浏览器的历史记录。如果您想要使用您自己的版本替换默认的历史记录页面，请参见替代页面。

5

i18n

使用 chrome.i18n 架构为您的整个应用或扩展程序实现国际化支持。

5

identity

使用 chrome.identity API 获取 OAuth2 访问令牌。

29

idle

使用 chrome.idle API 检测计算机空闲状态的更改。

6

input.ime

使用 chrome.input.ime API 为 Chrome OS 实现自定义的输入法，它允许您的扩展程序处理键盘输入、设置候选内容及管理候选窗口。

21

management

chrome.management API 可以用来管理已经安装并且正在运行的扩展程序或应用，它对于替代内建的“打开新的标签页”页面的扩展程序特别有用。

8

notifications

使用 chrome.notifications API 通过模板创建丰富通知，并在系统托盘中向用户显示这些通知。

28

omnibox

多功能框 API 允许您在 Google Chrome 浏览器的地址栏（又叫多功能框）中注册一个关键字。

9

pageAction

使用 chrome.pageAction API 在地址栏中添加图标。页面按钮代表用于当前页面的操作，但是不适用于所有页面。

5

pageCapture

使用 chrome.pageCapture API 将一个标签页保存为 MHTML。

18

permissions

使用 chrome.permissions API 在运行时而不是安装时请求声明的可选权限，这样用户可以理解为什么需要这些权限，并且仅在必要时授予这些权限。

16

power

使用 chrome.power API 修改系统的电源管理特性。

27

privacy

使用 chrome.privacy API 控制 Chrome 浏览器中可能会影响用户隐私的特性。这一模块依赖于类型 API 中的 ChromeSettings 原型，用于获取和设置 Chrome 浏览器的配置。

18

proxy

使用 chrome.proxy API 管理 Chrome 浏览器的代理服务器设置。该模块依赖于类型 API 中的 ChromeSetting 原型，用于获取和设置代理服务器配置。

13

pushMessaging

使用 chrome.pushMessaging 使应用或扩展程序能够接收通过 Google 云消息服务发送的消息数据。

24

runtime

使用 chrome.runtime API 获取后台页面、返回清单文件的详情、监听并响应应用或扩展程序生命周期内的事件，您还可以使用该 API 将相对路径的 URL 转换为完全限定的 URL。

22

storage

使用 chrome.storage API 存储、获取用户数据，追踪用户数据的更改。

20

system.cpu

使用 systemInfo.cpu API 查询 CPU 元数据。

32

system.memory

chrome.system.memory API。

32

system.storage

使用 chrome.system.storage API 查询存储设备信息，并在连接或移除可移动存储设备时得到通知。

30

tabCapture

使用 chrome.tabCapture API 与标签页的媒体流交互。

31

tabs

使用 chrome.tabs API 与浏览器的标签页系统交互。您可以使用该 API 创建、修改和重新排列浏览器中的标签页。

5

topSites

使用 chrome.topSites API 访问“打开新的标签页”页面中的显示的“常去网站”。

19

tts

使用 chrome.tts API 播放合成的文字语音转换（TTS），同时请您参见相关的 ttsEngine API，允许扩展程序实现语音引擎。

14

ttsEngine

使用 chrome.ttsEngine API 用扩展程序实现文字语音转换（TTS）引擎。如果您的扩展程序注册了这一 API，当任何扩展程序或 Chrome 应用使用 tts 模块朗读时，它会收到事件，包含要朗读的内容以及其他参数。您的扩展程序可以使用任何可用的网络技术合成并输出语音，并向调用方发送事件报告状态。

14

types

chrome.types API 包含用于 Chrome 浏览器的类型声明。

13

webNavigation

使用 chrome.webNavigation API 实时地接收有关导航请求状态的通知。

16

webRequest

使用 chrome.webRequest API 监控与分析流量，还可以实时地拦截、阻止或者修改请求。

17

webstore

使用 chrome.webstore API 从您的网站上“内嵌”安装应用与扩展程序。

15

windows

使用 chrome.windows API 与浏览器窗口交互。您可以使用该模块创建、修改和重新排列浏览器中的窗口。

5

Beta API

名称描述

accessibilityFeatures

使用 chrome.accessibilityFeatures API 管理 Chrome 浏览器的辅助功能。该 API 使用类型 API 的 ChromeSetting 原型获取和设置辅助功能的各种特性。如果要获取特性的状态，扩展程序必须请求 accessibilityFeatures.read 权限。如果要修改特性状态，扩展程序需要 accessibilityFeatures.modify 权限。注意，accessibilityFeatures.modify 权限并不包含 accessibilityFeatures.read 权限。

declarativeWebRequest

使用 chrome.declarativeWebRequest API 实时地拦截、阻止或者修改请求，它比 API 要快得多，因为您注册的规则在浏览器而不是 JavaScript 引擎中求值，这样就减少了来回延迟并且可以获得极高的效率。

gcm
使用 chrome.gcm 通过 Google Cloud Messaging 在应用和扩展程序中发送和接收消息。

Dev API

名称描述

infobars
使用 chrome.infobars API 在标签页内容的正上方添加一个水平面板，如以下屏幕截图所示。

location
使用 chrome.location API 获取计算机的地理位置。该 API 是 HTML 地理定位 API 的另一种版本，与事件页面兼容。

processes
使用 chrome.processes API 与浏览器进程交互。

sessions
使用 chrome.sessions API 查询和恢复浏览器会话中的标签页和窗口。

signedInDevices
使用 chrome.signedInDevices API 获取以当前配置文件所对应的账户登录的设备列表。

D.2 Chrome 应用全部 API

稳定 API

名称描述最低版本

alarms
使用 chrome.alarms API 安排代码周期性地或者在将来的指定时间运行。
22

app.runtime

使用 chrome.app.runtime API 管理应用的生命周期。应用运行时环境管理应用的安装，控制事件页面，并且可以在任何时候关闭应用。

23

app.window

使用 chrome.app.window API 创建窗口。窗口可以有框架，包含标题栏和大小控件，它们不和任何 Chrome 浏览器窗口关联。

23

contextMenus

使用 chrome.contextMenus API 向 Google Chrome 浏览器的右键菜单添加项目。您可以选择您在右键菜单中添加的项目应用于哪些类型的对象，例如图片、超链接和页面。

6

events

chrome.events 命名空间包含 API 分发事件使用的通用类型，以便在某些有意义的事情发生时通知您。

21

fileSystem

使用 chrome.fileSystem API 创建、读取、浏览与写入用户本地文件系统中经过沙盒屏蔽的一个区域。使用该 API，Chrome 应用可以读取和写入用户选定的位置，例如文本编辑应用可以使用该 API 读取和写入本地文档。所有失败信息都通过 runtime.lastError 通知。

23

i18n

使用 chrome.i18n 架构为您的整个应用或扩展程序实现国际化支持。

5

identity

使用 chrome.identity API 获取 OAuth2 访问令牌。

29

idle

使用 chrome.idle API 检测计算机空闲状态的更改。

6

mediaGalleries

使用 chrome.mediaGalleries API 从用户的本地磁盘(包含用户内容)中访问媒体文件(音频、图片、视频)。

23

notifications

使用 chrome.notifications API 通过模板创建丰富通知，并在系统托盘中向用户显示这些通知。

28

permissions

使用 chrome.permissions API 在运行时而不是安装时请求声明的可选权限，这样用户可以理解为什么需要这些权限，并且仅在必要时授予这些权限。

16

power

使用 chrome.power API 修改系统的电源管理特性。

27

pushMessaging

使用 chrome.pushMessaging 使应用或扩展程序能够接收通过 Google 云消息服务发送的消息数据。

24

runtime

使用 chrome.runtime API 获取后台页面、返回清单文件的详情、监听并响应应用或扩展程序生命周期内的事件，您还可以使用该 API 将相对路径的 URL 转换为完全限定的 URL。

22

serial

使用 chrome.serial API 读取和写入连接到串行端口的设备。

23

socket

使用 chrome.socket API 使用 TCP 和 UDP 连接通过网络发送和接收数据。注意：从 Chrome 33 开始该 API 弃用，您应该改用 sockets.udp、sockets.tcp 和 sockets.tcpServer API。

24

sockets.tcp

使用 chrome.sockets.tcp API 使用 TCP 连接通过网络发送和接收数据。该 API 是对原先 chrome.socket API 中 TCP 功能的增强。

33

sockets.tcpServer

使用 chrome.sockets.tcpServer API 创建使用 TCP 连接的服务器应用。该 API 是对原先 chrome.socket API 中 TCP 功能的增强。

33

sockets.udp

使用 chrome.sockets.udp API 使用 UDP 连接通过网络发送和接收数据。该 API 是对原先套接字 API 中 UDP 功能的增强。

33

storage

使用 chrome.storage API 存储、获取用户数据，追踪用户数据的更改。

20

syncFileSystem

使用 chrome.syncFileSystem API 在 Google 云端硬盘上保存和同步数据。该 API 并不是用来访问存储在 Google 云端硬盘上的任何用户文档的，它提供了应用专用的可同步存储，用于离线和缓存用途，这样同样的数据就可以在不同的客户端间使用。有关使用该 API 的更多信息，请阅读管理数据。

27

system.cpu

使用 systemInfo.cpu API 查询 CPU 元数据。

32

system.display

使用 system.display API 查询显示器的元数据。

30

system.memory
chrome.system.memory API。
32

system.network
使用 chrome.system.network API 获取网络接口信息。
33

system.storage
使用 chrome.system.storage API 查询存储设备信息，并在连接或移除可移动存储设备时得到通知。
30

tts
使用 chrome.tts API 播放合成的文字语音转换（TTS），同时请您参见相关的 ttsEngine API，允许扩展程序实现语音引擎。
14

types
chrome.types API 包含用于 Chrome 浏览器的类型声明。
13

usb
使用 chrome.usb API 与已连接的 USB 设备交互。该 API 提供了在应用的环境中进行 USB 操作的能力，通过该 API 应用可以作为硬件设备的驱动程序使用。
26

webstore
使用 chrome.webstore API 从您的网站上“内嵌”安装应用与扩展程序。
15

Beta API

名称描述

accessibilityFeatures

使用 chrome.accessibilityFeatures API 管理 Chrome 浏览器的辅助功能。该 API 使用类型 API 的 ChromeSetting 原型获取和设置辅助功能的各种特性。如果要获取特性的状态，扩展程序必须请求 accessibilityFeatures.read 权限。如果要修改特性状态，扩展程序需要 accessibilityFeatures.modify 权限。注意，accessibilityFeatures.modify 权限并不包含 accessibilityFeatures.read 权限。

gcm

使用 chrome.gcm 通过 Google Cloud Messaging 在应用和扩展程序中发送和接收消息。

Dev API

名称描述

audio

chrome.audio API 允许用户获取连接到系统的音频设备信息，并控制它们。目前该 API 仅在 Chrome OS 上实现。

bluetooth

使用 chrome.bluetooth API 连接到蓝牙设备。所有函数都通过 chrome.runtime.lastError 报告错误。

location

使用 chrome.location API 获取计算机的地理位置。该 API 是 HTML 地理定位 API 的另一种版本，与事件页面兼容。

wallpaper

使用 chrome.wallpaper API 更改 ChromeOS 壁纸。