

第一部分

JavaScript 编程语言

JS

Ilya Kantor

建于 2020年6月2日

最新版教程详见 <https://zh.javascript.info.>

我们在不断努力改进教程。如果你发现任何错误, 请在[我们的 GitHub](#) 上告诉我们。.

- 简介
 - JavaScript 简介
 - 手册与规范
 - 代码编辑器
 - 开发者控制台
- JavaScript 基础知识
 - Hello, world!
 - 代码结构
 - 现代模式, "use strict"
 - 变量
 - 数据类型
 - 类型转换
 - 运算符
 - 值的比较
 - 交互: alert、prompt 和 confirm
 - 条件运算符: if 和 ?
 - 逻辑运算符
 - 循环: while 和 for
 - "switch" 语句
 - 函数
 - 函数表达式
 - 箭头函数, 基础知识
 - JavaScript 特性
- 代码质量
 - 在 Chrome 中调试
 - 代码风格
 - 注释
 - 忍者代码
 - 使用 Mocha 进行自动化测试
 - Polyfill
- Object (对象) : 基础知识
 - 对象
 - 垃圾回收
 - Symbol 类型
 - 对象方法, "this"
 - 对象 — 原始值转换
 - 构造器和操作符 "new"
- 数据类型

- 原始类型的方法
- 数字类型
- 字符串
- 数组
- 数组方法
- `Iterable object` (可迭代对象)
- `Map and Set` (映射和集合)
- `WeakMap and WeakSet` (弱映射和弱集合)
- `Object.keys, values, entries`
- 解构赋值
- 日期和时间
- `JSON` 方法, `toJSON`
- 函数进阶内容
 - 递归和堆栈
 - `Rest` 参数与 `Spread` 语法
 - 闭包
 - 旧时的 "var"
 - 全局对象
 - 函数对象, `NFE`
 - "`new Function`" 语法
 - 调度: `setTimeout` 和 `setInterval`
 - 装饰者模式和转发, `call/apply`
 - 函数绑定
 - 深入理解箭头函数
- 对象属性配置
 - 属性标志和属性描述符
 - 属性的 `getter` 和 `setter`
- 原型, 继承
 - 原型继承
 - `F.prototype`
 - 原生的原型
 - 原型方法, 没有 `__proto__` 的对象
- 类
 - `Class` 基本语法
 - 类继承
 - 静态属性和静态方法
 - 私有的和受保护的属性和方法
 - 扩展内建类
 - 类检查: "`instanceof`"
 - `Mixin` 模式
- 错误处理

- 错误处理, "try..catch"
- 自定义 Error, 扩展 Error
- Promise, async/await
 - 简介: 回调
 - Promise
 - Promise 链
 - 使用 promise 进行错误处理
 - Promise API
 - Promisification
 - 微任务 (Microtask)
 - Async/await
- Generator, 高级 iteration
 - Generator
 - Async iterator 和 generator
- 模块
 - 模块 (Module) 简介
 - 导出和导入
 - 动态导入
- 杂项
 - Proxy 和 Reflect
 - Eval: 执行代码字符串
 - 柯里化 (Currying)
 - BigInt

在这儿我们将从头开始学习 JavaScript，也会学习 OOP 等相关高级概念。

本教程专注于语言本身，我们默认使用最小环境。

简介

介绍 JavaScript 语言及其开发环境。

JavaScript 简介

让我们来看看 JavaScript 有什么特别之处，我们可以用它实现什么，以及哪些其他技术可以很好地使用它。

什么是 JavaScript?

JavaScript 最初的目的是为了“赋予网页生命”。

这种编程语言我们称之为 **脚本**。它们可以写在 HTML 中，在页面加载的时候会自动执行。

脚本作为纯文本存在和执行。它们不需要特殊的准备或编译即可运行。

这方面，JavaScript 和 Java ↗ 有很大的区别。

① 为什么叫 JavaScript?

JavaScript 在刚诞生的时候，它的名字叫 “LiveScript”。但是因为当时 Java 很流行，所以决定将一种新语言定位为 Java 的“弟弟”会有助于它的流行。

随着 JavaScript 的发展，它已经变成了一门独立的语言，同时也有了自己的语言规范 [ECMAScript](#) ↗。现在，它和 Java 之间没有任何关系。

现在，JavaScript 不仅仅是在浏览器内执行，也可以在服务端执行，甚至还能在任意搭载了 [JavaScript 引擎](#) ↗ 的设备中都可以执行。

浏览器中嵌入了 JavaScript 引擎，有时也称作 JavaScript 虚拟机。

不同的引擎有不同的“代号”，例如：

- [V8](#) ↗ —— Chrome 和 Opera 中的 JavaScript 引擎。
- [SpiderMonkey](#) ↗ —— Firefox 中的 JavaScript 引擎。
-还有其他一些代号，像 “Trident”，“Chakra” 用于不同版本的 IE，“ChakraCore” 用于 Microsoft Edge，“Nitro” 和 “SquirrelFish” 用于 Safari，等等。

上面这些名称很容易记忆，因为经常出现在网上开发者的文章中。我们也会用到这些名称。例如：某个新的功能，如果“[JavaScript 引擎 V8 是支持的](#)”，那么我们可以认为这个功能大概能在 Chrome 和 Opera 中正常运行。

① 引擎是如何工作的？

引擎很复杂，但是基本原理很简单。

1. 引擎（通常嵌入在浏览器中）读取（“解析”）脚本。
2. 然后将脚本转化（“编译”）为机器语言。
3. 然后这机器语言代码快速地运行。

引擎会对流程中的每个阶段都进行优化。它甚至可以在运行时监视编译的脚本，分析数据流并根据这些进一步优化机器代码。

浏览器中的 JavaScript 能做什么？

现代的 **JavaScript** 是一种“安全”语言。它不提供对内存或 CPU 的底层访问，因为它最初是为浏览器创建的，不需要这些功能。

JavaScript 的能力很大程度上依赖于它执行的环境。例如：[Node.js](#) 允许 **JavaScript** 读写任意文件、执行网络请求等。

浏览器中的 **JavaScript** 可以做与网页操作、用户交互和 Web 服务器相关的所有事情。

例如，浏览器中的 **JavaScript** 可以完成下面这些事：

- 在网页中插入新的 **HTML**，修改现有的网页内容和网页的样式。
- 响应用户的行为，响应鼠标的点击或移动、键盘的敲击。
- 向远程服务器发送网络请求，下载或上传文件（所谓 [AJAX](#) 和 [COMET](#) 技术）。
- 获取或修改 **cookie**，向访问者提出问题、发送消息。
- 记住客户端的数据（本地存储）。

浏览器中的 JavaScript 不能做什么？

为了用户的（信息）安全，在浏览器中的 **JavaScript** 的能力是有限的。这样主要是为了阻止邪恶的网站获得或修改用户的私人数据。

这些限制的例子有：

- 网页中的 **JavaScript** 不能读、写、复制及执行用户磁盘上的文件或程序。它没有直接访问操作系统的功能。

现代浏览器允许 **JavaScript** 做一些文件相关操作，但是这个操作是受到限制的。仅当用户做出特定的行为，**JavaScript** 才能操作这个文件。例如，把文件“拖”到浏览器中，或者通过 **<input>** 标签选择文件。

JavaScript 有很多方式和照相机/麦克风或者其他设备进行交互，但是这些都需要提前获得用户的授权许可。所以，启用了 **JavaScript** 的网页应该不会偷偷地启动网络摄像头观察你，并把你信息发送到 [美国国家安全局](#)。

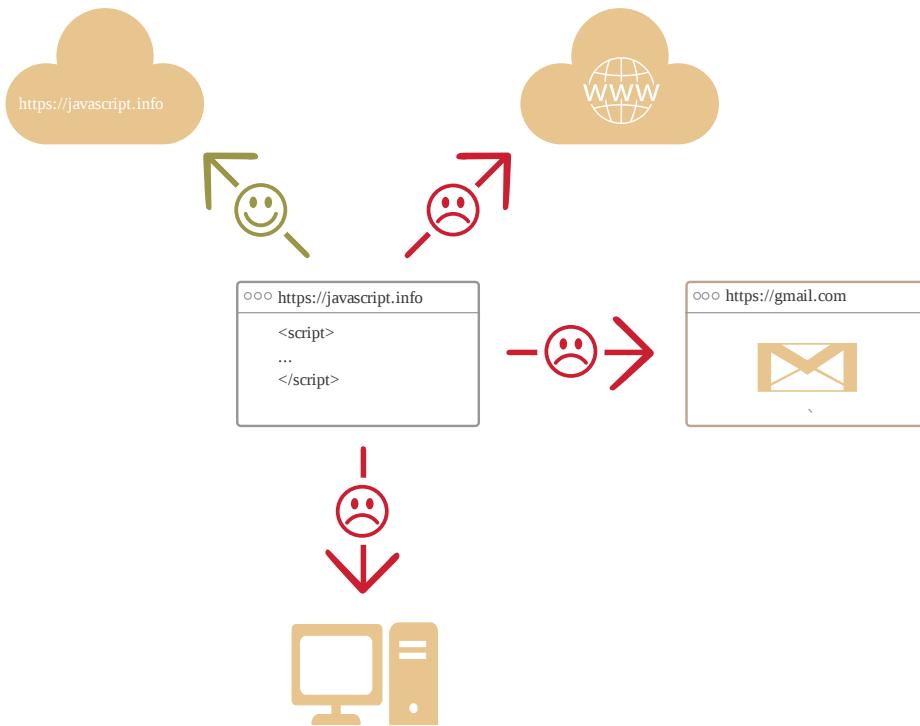
- 不同的浏览器标签页之间基本彼此不相关。有时候，也会有一些关系。例如，一个标签页通过 **JavaScript** 打开另外一个新的标签页。但即使在这种情况下，如果两个标签页打开的不是同一个网站（域名、协议或者端口任一不相同的网站），他们都不能够相互通信。

这就是“同源策略”。为了解决“同源策略”问题，两个标签页必须 **都** 包含一些处理这个问题的特殊的 **JavaScript** 代码，并均允许数据交换，这样才能够实现两个同源标签页的数据交换。本教程

会讲到这部分相关的知识。

这个限制也是为了用户的信息安全。例如，用户打开的 `http://anysite.com` 网页的 JavaScript 肯定不能访问 `http://gmail.com`（另外一个标签页打开的网页）也不能从那里窃取信息。

- JavaScript 通过互联网可以轻松地和当前网页域名的服务器进行通讯。但是从其他网站/域名的服务器中获取数据的能力是受限的。尽管这可以实现，但是需要来自远程服务器的明确协议（在 HTTP header 中）。这也是为了用户的数据安全。



浏览器环境外的 JavaScript 一般没有这些限制。例如服务端的 JavaScript 就没有这些限制。现代浏览器还允许安装可能会要求扩展权限的插件或扩展。

是什么使得 JavaScript 与众不同？

至少有 **3** 件事值得一提：

- 和 HTML/CSS 完全的集成。
- 使用简单的工具完成简单的任务。
- 被所有的主流浏览器支持，并且默认开启。

满足这三条的浏览器技术也只有 JavaScript 了。

这就是为什么 JavaScript 与众不同！这也是为什么它是创建浏览器界面的最普遍的工具。

此外，JavaScript 还支持创建服务器，移动端应用程序等。

比 JavaScript “更好”的语言

不同的人喜欢不同的功能，**JavaScript** 的语法也不能够满足所有人的需求。

这是正常的，因为每个人的项目和需求都不一样。

所以，最近出现了很多不同的语言，这些语言在浏览器中执行之前，都会被**编译**（转化）成 **JavaScript**。

现代化的工具使得编译速度非常快速且透明，实际上允许开发人员使用另一种语言编写代码并将其自动转换为 **JavaScript**。

这些编程语言的例子有：

- [CoffeeScript ↗](#) 是 **JavaScript** 的语法糖，它语法简短，明确简洁。通常使用 **Ruby** 的人喜欢用。
- [TypeScript ↗](#) 将注意力集中在增加严格的数据类型。这样就能简化开发，也能用于开发复杂的系统。**TypeScript** 是微软开发的。
- [Flow ↗](#) 也添加了数据类型，但是以一种不同的方式。由 **Facebook** 开发。
- [Dart ↗](#) 是一门独立的语言。它拥有自己的引擎用于在非浏览器环境中运行（如：手机应用），它也能被编译成 **JavaScript**。由 **Google** 开发。

还有很多其他的语言。当然，即使我们在使用这些语言，我们也需要知道 **JavaScript**。因为学习 **JavaScript** 可以让我们真正明白我们自己在做什么。

总结

- **JavaScript** 最开始是为浏览器设计的一门语言，但是现在也被用于很多其他的环境。
- 现在，**JavaScript** 是一门在浏览器中使用最广、并且能够很好集成 **HTML/CSS** 的语言。
- 有很多其他的语言可以被编译成 **JavaScript**，这些语言还提供了更多的功能。最好还是了解一下这些语言，至少在掌握了 **JavaScript** 之后简单地看一下。

手册与规范

这本书是一个**教程**。它旨在帮助你逐渐掌握 **JavaScript** 这门语言。但是一旦你已经熟悉了这门语言的基础知识，你就会需要其他资料。

规范

ECMA-262 规范 包含了大部分深入的、详细的、规范化的关于 **JavaScript** 的信息。这份规范明确地定义了这门语言。

但正因其规范化，对于新手来说难以理解。所以如果你需要知道关于这门语言细节最权威的信息来源，这份规范就很适合你（去阅读）。但是它并不适合日常使用。

最新的规范草案在此 <https://tc39.es/ecma262/>。

想要知道最新最前沿且将要“标准化”的功能，请看这里的提案
<https://github.com/tc39/proposals>。

当然，如果你正在做浏览器相关的开发工作，那么本教程的 [第二节](#) 涵盖了其他规范。

手册

- **MDN (Mozilla) JavaScript 索引**是一本带有用例和其他信息的手册。它是一个获取关于个别语言函数、方法等深入信息的很好的来源。

你可以在 [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference ↗](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference) 找到这本手册。

虽然，利用互联网搜索通常是最好的选择。只需在查询时输入“MDN [关键字]”，例如 [https://google.com/search?q=MDN+parseInt ↗](https://google.com/search?q=MDN+parseInt) 搜索 `parseInt` 函数。

- **MSDN** ——一本微软的手册，它包含大量的信息，包括 `JavaScript`（在里面经常被写成 `JScript`）。如果有人需要关于 `Internet Explorer` 的规范细节，最好去看：
[http://msdn.microsoft.com/ ↗](http://msdn.microsoft.com/)。

我们还可以在使用互联网搜索时使用如“`RegExp MSDN`”或“`RegExp MSDN jscript`”这样的词条。

兼容性表

`JavaScript` 还是一门还在发展中的语言，经常会添加一些新的功能。

如果想要获得一些关于浏览器和其他引擎的兼容性信息，请看：

- [http://caniuse.com ↗](http://caniuse.com) ——每个功能都列有一个支持信息表格，例如想看哪个引擎支持现代加密（cryptography）函数：[http://caniuse.com/#feat=cryptography ↗](http://caniuse.com/#feat=cryptography)。
- [https://kangax.github.io/compat-table ↗](https://kangax.github.io/compat-table) ——一份列有语言功能以及引擎是否支持这些功能的表格。

所有这些资源在实际开发中都有用武之地，因为他们包含了语言细节以及它们被支持的程度等非常有价值的信息。

为了不要让你在真正需要深入了解特定功能的时候捉襟见肘，请记住它们（或者这一页）。

代码编辑器

程序员接触时间最长的就是代码编辑器。

代码编辑器主要分两种：`IDE`（集成开发环境）和轻量编辑器。很多人喜欢这两种各选一个。

IDE

[IDE ↗](#)（集成开发环境）是用于管理整个项目具有强大功能的编辑器。顾名思义，它不仅仅是一个编辑器，而且还是个完整的开发环境。

`IDE` 加载项目（通常包含很多文件），并且允许在不同文件之间导航（navigation）。`IDE` 还提供基于整个项目（不仅仅是打开的文件）的自动补全功能，集成版本控制（如 [git ↗](#)）、集成测试环境等一些其他“项目层面”的东西。

如果你还没考虑好选哪一款 `IDE`，可以考虑下面两个：

- [Visual Studio Code ↗](#)（跨平台，免费）
- [WebStorm ↗](#)（跨平台，收费）

对于 Windows 系统来说，也有个叫“`Visual Studio`”的 `IDE`，请不要跟“`Visual Studio Code`”混淆。“`Visual Studio`”是一个收费的、强大的 Windows 专用编辑器，它十分适合于 .NET 开发。用

它进行 JavaScript 开发也不错。“Visual Studio”有个免费的版本 [Visual Studio Community](#)。

大多数 IDE 是收费的，但是他们都可以试用。购买 IDE 的费用对于一名合格的程序员的薪水来说，肯定算不了什么，所以去选一个对你来说最好的吧。

轻量编辑器

“轻量编辑器”没有 IDE 功能那么强大，但是他们一般很快、优雅而且简单。

“轻量编辑器”主要用于立即打开编辑一个文件。

“轻量编辑器”和 IDE 最大的区别是，IDE 一般在项目中使用，这也就意味着在开启的时候要加载很多数据，如果需要的话，在使用的过程中还会分析项目的结构等。如果我们只需要编辑一个文件，那么“轻量编辑器”会更快。

实际上，“轻量编辑器”一般都有各种各样的插件，这些插件可以做目录级（directory-level）的语法分析和补全代码。所以“轻量编辑器”和 IDE 也没有严格的界限。

下面是一些值得你关注的“轻量编辑器”：

- [Atom](#)（跨平台，免费）。
- [Visual Studio Code](#)（跨平台，免费）。
- [Sublime Text](#)（跨平台，共享软件）。
- [Notepad++](#)（Windows，免费）。
- [Vim](#) 和 [Emacs](#) 很棒，前提是你知道怎么用。

不要争吵

上面列表中的编辑器都是我和我的朋友（他们都是我认为很优秀的开发人员）已经使用了很长时间并且很满意。

世上还有很多其他很好的编辑器，你可以选择一个你最喜欢的。

选择编辑器就像选择其他工具一样。要看你的项目，以及个人的习惯和喜好。

开发者控制台

代码是很容易出现错误的。你也很可能犯错误……哦，我在说什么？只要你是人，你一定会犯错误（在写代码的时候），除非你是[机器人](#)。

但在浏览器中，默认情况下用户是看不到错误的。所以，如果脚本中有错误，我们看不到是什么错误，更不能够修复它。

为了发现错误并获得一些与脚本相关且有用的信息，浏览器内置了“开发者工具”。

通常，开发者倾向于使用 Chrome 或 Firefox 进行开发，因为它们有最好的开发者工具。一些其他的浏览器也提供开发者工具，有时还具有一些特殊的功能，通常它们都是在追赶 Chrome 或 Firefox。所以大多数人都有“最喜欢”的浏览器，当遇到某个浏览器独有的问题的时候，人们就会切换到其他的浏览器。

开发者工具很强大，功能丰富。首先，我们将学习如何打开它们，查找错误和运行 JavaScript 命令。

Google Chrome

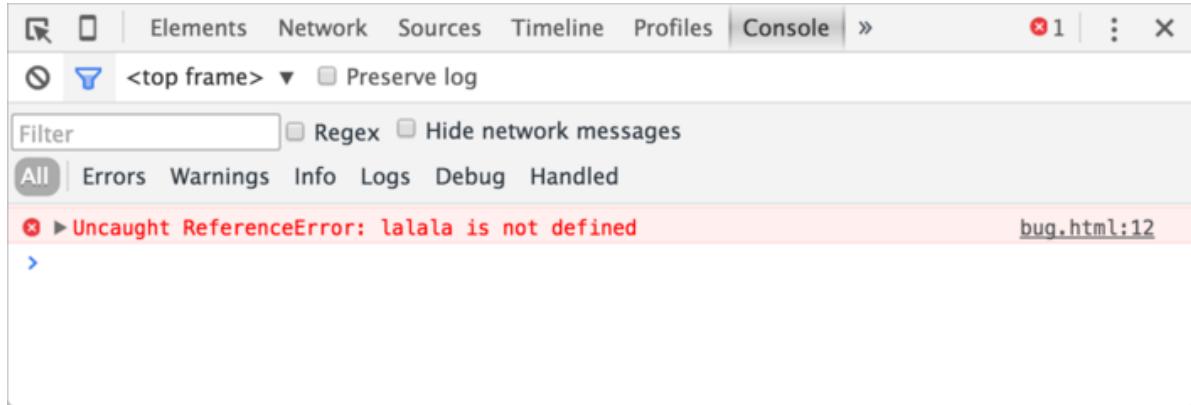
打开网页 [bug.html](#)。

在 JavaScript 代码中有一个错误。一般的访问者看不到这个错误，所以让我们打开开发者工具看看吧。

按下 **F12** 键，如果你使用 Mac，试试 **Cmd+Opt+J**。

开发者工具会被打开，Console 标签页是默认的标签页。

就像这样：



具体什么样，要看你的 Chrome 版本。它随着时间一直在变，但是都很类似。

- 在这我们能看到红色的错误提示信息。这个场景中，脚本里有一个未知的“lalala”命令。
- 在右边，有个可点击的链接 `bug.html:12`。这个链接会链接到错误发生的行号。

在错误信息的下方，有个 `>` 标志。它代表“命令行”，在“命令行”中，我们可以输入 JavaScript 命令，按下 **Enter** 来执行。

现在，我们能看到错误就够了。稍后，在 [在 Chrome 中调试](#) 章节中，我们会重新更加深入地讨论开发者工具。

➊ 多行输入

通常，当我们向控制台输入一行代码后，按 **Enter**，这行代码就会立即执行。

如果想要插入多行代码，请按 **Shift+Enter** 来进行换行。这样就可以输入长片段的 JavaScript 代码了。

Firefox、Edge 和其他浏览器

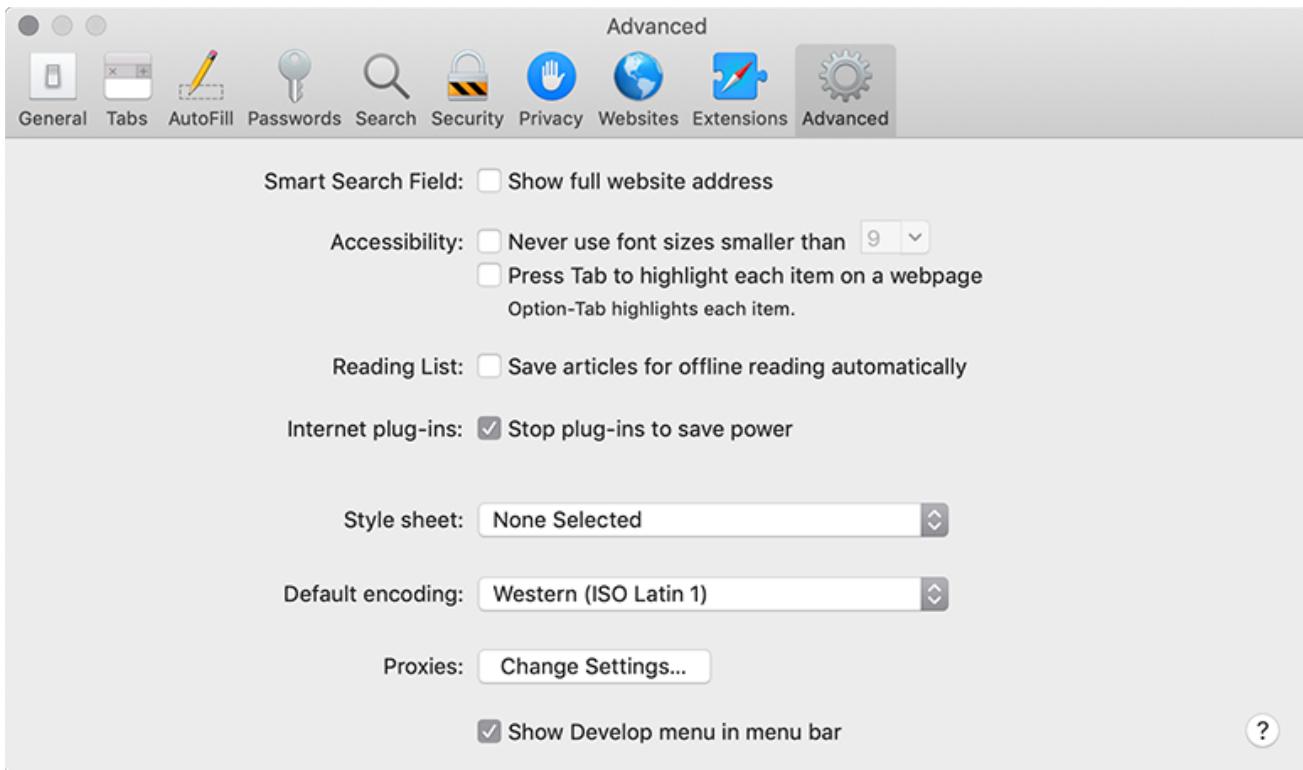
大多数其他的浏览器都是通过 **F12** 来打开开发者工具。

他们的外观和感觉都非常相似，一旦你学会了他们中的一个（可以先尝试 Chrome），其他的也就很快了。

Safari

Safari（Mac 系统中的浏览器，Windows 和 Linux 系统不支持）有一点点不同。我们需要先开启“开发菜单”。

打开“偏好设置”，选择“高级”选项。选中最下方的那个选择框。



现在，我们通过 **Cmd+Opt+C** 就能打开或关闭控制台了。另外注意，有一个名字为“开发”的顶部菜单出现了。它有很多命令和选项。

总结

- 开发者工具允许我们查看错误、执行命令、检查变量等等。
- 在 Windows 系统中，可以通过 **F12** 开启开发者工具。Mac 系统下，Chrome 需要使用 **Cmd+Opt+J**，Safari 使用 **Cmd+Opt+C**（需要提前开启）。

现在我们的环境准备好了。下一章，我们将正式开始学习 JavaScript。

JavaScript 基础知识

让我们来一起学习 JavaScript 脚本构建的基础知识。

Hello, world!

本教程的这一部分内容是关于 JavaScript 语言本身的。

但是，我们需要一个工作环境来运行我们的脚本，由于本教程是在线的，所以浏览器是一个不错的选择。我们会尽可能少地使用浏览器特定的命令（比如 `alert`），所以如果你打算专注于另一个环境（比如 `Node.js`），你就不必多花时间来关心这些特定指令了。我们将在本教程的 [下一部分](#) 中专注于浏览器中的 JavaScript。

首先，让我们看看如何将脚本添加到网页上。对于服务器端环境（如 `Node.js`），你只需要使用诸如 "`node my.js`" 的命令行来执行它。

“`script`” 标签

JavaScript 程序可以在 `<script>` 标签的帮助下插入到 HTML 文档的任何地方。

比如：

```
<!DOCTYPE HTML>
<html>

<body>

<p>script 标签之前...</p>

<script>
  alert('Hello, world!');
</script>

<p>...script 标签之后</p>

</body>

</html>
```

`<script>` 标签中包裹了 JavaScript 代码，当浏览器遇到 `<script>` 标签，代码会自动运行。

现代的标记（markup）

`<script>` 标签有一些现在很少用到的特性（attribute），但是我们可以在老代码中找到它们：

`type` 特性: `<script type=...>`

在老的 HTML4 标准中，要求 script 标签有 `type` 特性。通常是 `type="text/javascript"`。这样的特性声明现在已经不再需要。而且，现代 HTML 标准已经完全改变了此特性的含义。现在，它可以用作 JavaScript 模块。但这是一个高级话题，我们将在本教程的另一部分中探讨 JavaScript 模块。

`language` 特性: `<script language=...>`

这个特性是为了显示脚本使用的语言。这个特性现在已经没有任何意义，因为语言默认就是 JavaScript。不再需要使用它了。

脚本前后的注释。

在非常古老的书籍和指南中，你可能会在 `<script>` 标签里面找到注释，就像这样：

```
<script type="text/javascript"><!--
...
--></script>
```

现代 JavaScript 中已经不这样使用了。这些注释是用于不支持 `<script>` 标签的古老的浏览器隐藏 JavaScript 代码的。由于最近 15 年内发布的浏览器都没有这样的问题，因此这种注释能帮你辨认出一些老掉牙的代码。

外部脚本

如果你有大量的 JavaScript 代码，我们可以将它放入一个单独的文件。

脚本文件可以通过 `src` 特性 (attribute) 添加到 HTML 文件中。

```
<script src="/path/to/script.js"></script>
```

这里, `/path/to/script.js` 是脚本文件从网站根目录开始的绝对路径。当然也可以提供当前页面的相对路径。例如, `src ="script.js"` 表示当前文件夹中的 `"script.js"` 文件。

我们也可以提供一个完整的 URL 地址, 例如:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"></script>
```

要附加多个脚本, 请使用多个标签:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
...

```

i 请注意:

一般来说, 只有最简单的脚本才嵌入到 HTML 中。更复杂的脚本存放在单独的文件中。

使用独立文件的好处是浏览器会下载它, 然后将它保存到浏览器的 [缓存](#) 中。

之后, 其他页面想要相同的脚本就会从缓存中获取, 而不是下载它。所以文件实际上只会下载一次。

这可以节省流量, 并使得页面 (加载) 更快。

⚠ 如果设置了 `src` 特性, `script` 标签内容将会被忽略。

一个单独的 `<script>` 标签不能同时有 `src` 特性和内部包裹的代码。

这将不会工作:

```
<script src="file.js">
  alert(1); // 此内容会被忽略, 因为设定了 src
</script>
```

我们必须进行选择, 要么使用外部的 `<script src="...">`, 要么使用正常包裹代码的 `<script>`。

为了让上面的例子工作, 我们可以将它分成两个 `<script>` 标签。

```
<script src="file.js"></script>
<script>
  alert(1);
</script>
```

总结

- 我们可以使用一个 `<script>` 标签将 JavaScript 代码添加到页面中。
- `type` 和 `language` 特性（attribute）不是必需的。
- 外部的脚本可以通过 `<script src="path/to/script.js"></script>` 的方式插入。

有关浏览器脚本以及它们和网页的关系，还有很多可学的。但是请记住，教程的这部分主要是针对 JavaScript 语言本身的，所以我们不该被浏览器特定的实现分散自己的注意力。我们将使用浏览器作为运行 JavaScript 的一种方式，这种方式非常便于我们在线阅读，但这只是很多种方式中的一种。

代码结构

我们将要学习的第一个内容就是构建代码块。

语句

语句是执行行为（action）的语法结构和命令。

我们已经见过了 `alert('Hello, world!')` 这样可以用来显示消息的语句。

我们可以在代码中编写任意数量的语句。语句之间可以使用分号进行分割。

例如，我们将“Hello World”这条信息一分为二：

```
alert('Hello'); alert('World');
```

通常，每条语句独占一行，以提高代码的可读性：

```
alert('Hello');
alert('World');
```

分号

当存在分行符（line break）时，在大多数情况下可以省略分号。

下面的代码也是可以运行的：

```
alert('Hello')
alert('World')
```

在这，JavaScript 将分行符理解成“隐式”的分号。这也被称为 [自动分号插入 ↗](#)。

在大多数情况下，换行意味着一个分号。但是“大多数情况”并不意味着“总是”！

有很多换行并不是分号的例子，例如：

```
alert(3 +
1)
```

```
+ 2);
```

代码输出 `6`，因为 JavaScript 并没有在这里插入分号。显而易见的是，如果一行以加号 `"+"` 结尾，那么这是一个“不完整的表达式”，不需要分号。所以，这个例子得到了预期的结果。

但存在 JavaScript 无法确定是否真的需要自动插入分号的情况。

这种情况下发生的错误是很难被找到和解决的。

❶ 一个错误的例子

如果你好奇地想知道一个这种错误的具体例子，那你可以看看下面这段代码：

```
[1, 2].forEach(alert)
```

你不需要考虑方括号 `[]` 和 `forEach` 的含义，现在它们并不重要，之后我们会学习它们。让我们先记住这段代码的运行结果：先显示 `1`，然后显示 `2`。

现在我们在代码前面插入一个 `alert` 语句，并且不加分号：

```
alert("There will be an error")
[1, 2].forEach(alert)
```

现在，如果我们运行代码，只有第一个 `alert` 语句的内容被显示了出来，随后我们收到了一个错误！

但是，如果我们在第一个 `alert` 语句末尾加上一个分号，就工作正常了：

```
alert("All fine now");
[1, 2].forEach(alert)
```

现在，我们能得到 “All fine now”，然后是 `1` 和 `2`。

出现无分号变量（variant）的错误，是因为 JavaScript 并不会在方括号 `[...]` 前添加一个隐式的分号。

所以，因为没有自动插入分号，第一个例子中的代码被视为了一条简单的语句，我们从引擎看到的是这样的：

```
alert("There will be an error") [1, 2].forEach(alert)
```

但它应该是两条语句，而不是一条。这种情况下的合并是不对的，所以才会造成错误。诸如此类，还有很多。

即使语句被换行符分隔了，我们依然建议在它们之间加分号。这个规则被社区广泛采用。我们再次强调一下——大部分时候可以省略分号，但是最好不要省略分号，尤其对新手来说。

注释

随着时间推移，程序变得越来越复杂。为代码添加 **注释** 来描述它做了什么和为什么要这样做，变得非常有必要了。

你可以在脚本的任何地方添加注释，它们并不会影响代码的执行，因为引擎会直接忽略它们。

单行注释以两个正斜杠字符 `//` 开始。

这一行的剩余部分是注释。它可能独占一行或者跟随在一条语句的后面。

就像这样：

```
// 这行注释独占一行
alert('Hello');

alert('World'); // 这行注释跟随在语句后面
```

多行注释以一个正斜杠和星号开始 `“/*”` 并以一个星号和正斜杆结束 `“*/”`。

就像这样：

```
/* 两个消息的例子。
这是一个多行注释。
*/
alert('Hello');
alert('World');
```

注释的内容被忽略了，所以如果我们在 `/* ... */` 中放入代码，并不会执行。

有时候，可以很方便地临时禁用代码：

```
/* 注释代码
alert('Hello');
*/
alert('World');
```

❶ 使用热键！

在大多数的编辑器中，一行代码可以使用 `Ctrl+ /` 热键进行单行注释，诸如 `Ctrl+Shift+ /` 的热键可以进行多行注释（选择代码，然后按下热键）。对于 Mac 电脑，应使用 `Cmd` 而不是 `Ctrl`，使用 `Option` 而不是 `Shift`。

⚠ 不支持注释嵌套！

不要在 `/* ... */` 内嵌套另一个 `/* ... */`。

下面这段代码报错而无法执行：

```
/*
 * 嵌套注释 ?!?
 */
alert( 'World' );
```

对你的代码进行注释，这还有什么可犹豫的！

注释会增加代码总量，但这一点也不是什么问题。有很多工具可以帮你在把代码部署到服务器之前缩减代码。这些工具会移除注释，这样注释就不会出现在发布的脚本中。所以，注释对我们的生产没有任何负面影响。

在后面的教程中，会有一章 [代码质量](#) 的内容解释如何更好地写注释。

现代模式，“use strict”

长久以来，JavaScript 不断向前发展且并未带来任何兼容性问题。新的特性被加入，旧的功能也没有改变。

这么做有利于兼容旧代码，但缺点是 JavaScript 创造者的任何错误或不完善的决定也将永远被保留在 JavaScript 语言中。

这种情况一直持续到 2009 年 ECMAScript 5 (ES5) 的出现。ES5 规范增加了新的语言特性并且修改了一些已经存在的特性。为了保证旧的功能能够使用，大部分的修改是默认不生效的。你需要一个特殊的指令——`"use strict"` 来明确地激活这些特性。

“use strict”

这个指令看上去像一个字符串 `"use strict"` 或者 `'use strict'`。当它处于脚本文件的顶部时，则整个脚本文件都将以“现代”模式进行工作。

比如：

```
"use strict";

// 代码以现代模式工作
...
```

稍后我们才会学习函数（一种组合命令的方式）。

但我们可以提前了解一下，`"use strict"` 可以被放在函数主体的开头，而不是整个脚本的开头。这样则可以只在该函数中启用严格模式。但通常人们会在整个脚本中启用严格模式。

⚠ 确保 "use strict" 出现在最顶部

请确保 `"use strict"` 出现在脚本的最顶部，否则严格模式可能无法启用。

这里的严格模式就没有被启用：

```
alert("some code");
// 下面的 "use strict" 会被忽略，必须在最顶部。

"use strict";

// 严格模式没有被激活
```

只有注释可以出现在 `"use strict"` 的上面。

⚠ 没办法取消 `use strict`

没有类似于 `"no use strict"` 这样的指令可以使程序返回默认模式。

一旦进入了严格模式，就没有回头路了。

浏览器控制台

以后，当你使用浏览器控制台去测试功能时，请注意 `use strict` 默认不会被启动。

有时，使用 `use strict` 会产生一些不一样的影响，你会得到错误的结果。

你可以试试按下 `Shift+Enter` 去输入多行代码，然后将 `use strict` 置顶，就像这样：

```
'use strict'; <Shift+Enter 换行>
// ...你的代码
<按下 Enter 以运行>
```

它在大部分浏览器中都有效，像 Firefox 和 Chrome。

如果依然不行，那确保 `use strict` 被开启的最可靠的方法是，像这样将代码输入到控制台：

```
(function() {
  'use strict';

  // ...你的代码...
})()
```

总是使用 "use strict"

我们还没说到使用 `"use strict"` 与“默认”模式的区别。

在接下来的章节中，当我们学习语言功能时，我们会标注严格模式与默认模式的差异。幸运的是，差异其实没有那么多。并且这些差异可以让我们更好地编程。

当前，一般来说了解这些就够了：

1. `"use strict"` 指令将浏览器引擎转换为“现代”模式，改变一些内建特性的行为。我们会在之后的学习中了解这些细节。
2. 严格模式通过将 `"use strict"` 放置在整个脚本或函数的顶部来启用。一些新语言特性诸如 `"classes"` 和 `"modules"` 也会自动开启严格模式。
3. 所有的现代浏览器都支持严格模式。
4. 我们建议始终使用 `"use strict"` 启动脚本。本教程的所有例子都默认采用严格模式，除非特别指定（非常少）。

变量

大多数情况下，JavaScript 应用需要处理信息。这有两个例子：

1. 一个网上商店 —— 这里的信息可能包含正在售卖的商品和购物车。
2. 一个聊天应用 —— 这里的信息可能包括用户和消息等等。

变量就是用来储存这些信息的。

变量

变量 ↪ 是数据的“命名存储”。我们可以使用变量来保存商品、访客和其他信息。

在 JavaScript 中创建一个变量，我们需要用到 `let` 关键字。

下面的语句创建（也可以称为 **声明** 或者 **定义**）了一个名称为 “`message`” 的变量：

```
let message;
```

现在，我们可以通过赋值运算符 `=` 为变量添加一些数据：

```
let message;  
  
message = 'Hello'; // 保存字符串
```

现在这个字符串已经保存到与该变量相关联的内存区域了，我们可以通过使用该变量名称访问它：

```
let message;  
message = 'Hello!';  
  
alert(message); // 显示变量内容
```

简洁一点，我们可以将变量定义和赋值合并成一行：

```
let message = 'Hello!'; // 定义变量，并且赋值  
  
alert(message); // Hello!
```

也可以在一行中声明多个变量：

```
let user = 'John', age = 25, message = 'Hello';
```

看上去代码长度更短，但并不推荐这样。为了更好的可读性，请一行只声明一个变量。

多行变量声明有点长，但更容易阅读：

```
let user = 'John';
let age = 25;
let message = 'Hello';
```

一些程序员采用下面的形式书写多个变量：

```
let user = 'John',
  age = 25,
  message = 'Hello';
```

.....甚至使用“逗号在前”的形式：

```
let user = 'John'
, age = 25
, message = 'Hello';
```

技术上讲，这些变体都有一样的效果。所以，这是个个人品味和审美方面的问题。

① var 而不是 let

在较旧的脚本中，你也可能发现另一个关键字 `var`，而不是 `let`：

```
var message = 'Hello';
```

`var` 关键字与 `let` 大体相同，也用来声明变量，但稍微有些不同，也有点“老派”。

`let` 和 `var` 之间有些微妙的差别，但目前对于我们来说并不重要。我们将会在 [旧时的 "var"](#) 章节中介绍它们。

一个现实生活的类比

如果将变量想象成一个“数据”的盒子，盒子上有一个唯一的标注盒子名字的贴纸。这样我们能更容易地掌握“变量”的概念。

例如，变量 `message` 可以被想象成一个标有 `"message"` 的盒子，盒子里面的值为 `"Hello!"`：



我们可以在盒子内放入任何值。

并且，这个盒子的值，我们想改变多少次，就可以改变多少次：

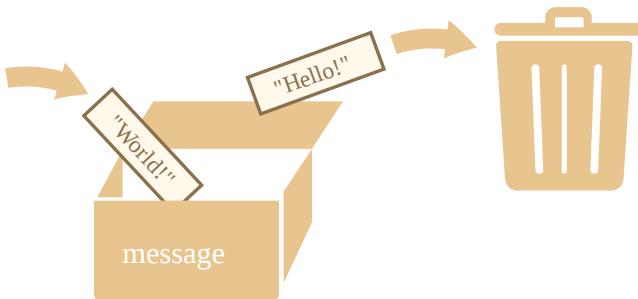
```
let message;

message = 'Hello!';

message = 'World!'; // 值改变了

alert(message);
```

当值改变的时候，之前的数据就被从变量中删除了：



我们还可以声明两个变量，然后将其中一个变量的数据拷贝到另一个变量。

```
let hello = 'Hello world!';

let message;

// 将字符串 'Hello world' 从变量 hello 拷贝到 message
message = hello;

// 现在两个变量保存着相同的数据
alert(hello); // Hello world!
alert(message); // Hello world!
```

① 函数式语言

有趣的是，也存在禁止更改变量值的 [函数式](#) 编程语言。比如 [Scala](#) 或 [Erlang](#)。

在这种类型的语言中，一旦值保存在盒子中，就永远存在。如果你试图保存其他值，它会强制创建一个新盒子（声明一个新变量），无法重用之前的变量。

虽然第一次看上去有点奇怪，但是这些语言有很大的发展潜力。不仅如此，在某些领域，比如并行计算，这个限制有一定的好处。研究这样的一门语言（即使不打算很快就用上它）有助于开阔视野。

变量命名

JavaScript 的变量命名有两个限制：

1. 变量名称必须仅包含字母、数字、符号 `$` 和 `_`。
2. 首字符必须非数字。

有效的命名，例如：

```
let userName;  
let test123;
```

如果命名包括多个单词，通常采用驼峰式命名法（[camelCase](#)）。也就是，单词一个接一个，除了第一个单词，其他的每个单词都以大写字母开头：`myVeryLongName`。

有趣的是，美元符号 `'$'` 和下划线 `'_'` 也可以用于变量命名。它们是正常的符号，就跟字母一样，没有任何特殊的含义。

下面的命名是有效的：

```
let $ = 1; // 使用 "$" 声明一个变量  
let _ = 2; // 现在用 "_" 声明一个变量  
  
alert($ + _); // 3
```

下面的变量命名不正确：

```
let 1a; // 不能以数字开始  
  
let my-name; // 连字符 '-' 不允许用于变量命名
```

① 区分大小写

命名为 `apple` 和 `AppLE` 的变量是不同的两个变量。

① 允许非英文字母，但不推荐

可以使用任何语言，包括西里尔字母（cyrillic letters）甚至是象形文字，就像这样：

```
let имя = '...';
let 我 = '...';
```

技术上讲，完全没有错误，这样的命名是完全允许的，但是用英文进行变量命名是国际传统。哪怕我们正在写一个很小的脚本，它也有可能有很长的生命周期。某个时候，来自其他国家的人可能会阅读它。

⚠ 保留学

有一张 [保留学列表 ↗](#)，这张表中的保留学无法用作变量命名，因为它们被用于编程语言本身了。

比如，`let`、`class`、`return`、`function` 都被保留了。

下面的代码将会抛出一个语法错误：

```
let let = 5; // 不能用 "let" 来命名一个变量，错误!
let return = 5; // 同样，不能使用 "return"，错误!
```

⚠ 未采用 `use strict` 下的赋值

一般，我们需要在使用一个变量前定义它。但是在早期，我们可以不使用 `let` 进行变量声明，而可以简单地通过赋值来创建一个变量。现在如果我们不在脚本中使用 `use strict` 声明启用严格模式，这仍然可以正常工作，这是为了保持对旧脚本的兼容。

```
// 注意：这个例子中没有 "use strict"

num = 5; // 如果变量 "num" 不存在，就会被创建

alert(num); // 5
```

上面这是个糟糕的做法，严格模式下会报错。

```
"use strict";

num = 5; // 错误：num 未定义
```

常量

声明一个常数（不变）变量，可以使用 `const` 而非 `let`：

```
const myBirthday = '18.04.1982';
```

使用 `const` 声明的变量称为“常量”。它们不能被修改，如果你尝试修改就会发现报错：

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // 错误，不能对常量重新赋值
```

当程序员能确定这个变量永远不会改变的时候，就可以使用 `const` 来确保这种行为，并且清楚地向别人传递这一事实。

大写形式的常数

一个普遍的做法是将常量用作别名，以便记住那些在执行之前就已知的难以记住的值。

使用大写字母和下划线来命名这些常量。

例如，让我们以所谓的“web”（十六进制）格式为颜色声明常量：

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// .....当我们需要选择一个颜色
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

好处：

- `COLOR_ORANGE` 比 `"#FF7F00"` 更容易记忆。
- 比起 `COLOR_ORANGE` 而言，`"#FF7F00"` 更容易输错。
- 阅读代码时，`COLOR_ORANGE` 比 `#FF7F00` 更易懂。

什么时候该为常量使用大写命名，什么时候进行常规命名？让我们弄清楚一点。

作为一个“常数”，意味着值永远不变。但是有些常量在执行之前就已知了（比如红色的十六进制值），还有些在执行期间被“计算”出来，但初始赋值之后就不会改变。

例如：

```
const pageLoadTime = /* 网页加载所需的时间 */;
```

`pageLoadTime` 的值在页面加载之前是未知的，所以采用常规命名。但是它仍然是个常量，因为赋值之后不会改变。

换句话说，大写命名的常量仅用作“硬编码（hard-coded）”值的别名。

正确命名变量

谈到变量，还有一件非常重要的事。

一个变量名应该有一个清晰、明显的含义，对其存储的数据进行描述。

变量命名是编程过程中最重要且最复杂的技能之一。快速地浏览变量的命名就知道代码是一个初学者还是有经验的开发者写的。

在一个实际项目中，大多数的时间都被用来修改和扩展现有的代码库，而不是从头开始写一些完全独立的代码。当一段时间后，我们做完其他事情，重新回到我们的代码，找到命名良好的信息要容易得多。换句话说，变量要有个好名字。

声明变量之前，多花点时间思考它的更好的命名。你会受益良多。

一些可以遵循的规则：

- 使用易读的命名，比如 `userName` 或者 `shoppingCart`。
- 离诸如 `a`、`b`、`c` 这种缩写和短名称远一点，除非你真的知道你在干什么。
- 变量名在能够准确描述变量的同时要足够简洁。不好的例子就是 `data` 和 `value`，这样的名称等于什么都没说。如果能够非常明显地从上下文知道数据和值所表达的含义，这样使用它们也是可以的。
- 脑海中的术语要和团队保持一致。如果网站的访客称为“用户”，则我们采用相关的变量命名，比如 `currentUser` 或者 `newUser`，而不要使用 `currentVisitor` 或者一个 `newManInTown`。

听上去很简单？确实如此，但是在实践中选择一个一目了然的变量名称并非如此简单。大胆试试吧。

i 重用还是新建？

最后一点，有一些懒惰的程序员，倾向于重用现有的变量，而不是声明一个新的变量。

结果是，这个变量就像是被扔进不同东西盒子，但没有改变它的贴纸。现在里面是什么？谁知道呢。我们需要靠近一点，仔细检查才能知道。

这样的程序员节省了一点变量声明的时间，但却在调试代码的时候损失数十倍时间。

额外声明一个变量绝对是利大于弊的。

现代的 JavaScript 压缩器和浏览器都很够很好地对代码进行优化，所以不会产生性能问题。为不同的值使用不同的变量可以帮助引擎对代码进行优化。

总结

我们可以使用 `var`、`let` 或 `const` 声明变量来存储数据。

- `let` — 现代的变量声明方式。
- `var` — 老旧的变量声明方式。一般情况下，我们不会再使用它。但是，我们会在[旧时的 "var"](#)章节介绍 `var` 和 `let` 的微妙差别，以防你需要它们。
- `const` — 类似于 `let`，但是变量的值无法被修改。

变量应当以一种容易理解变量内部是什么的方式进行命名。

数据类型

JavaScript 中的变量可以保存任何数据。变量在前一刻可以是个字符串，下一刻就可以变成 `number` 类型：

```
// 没有错误
let message = "hello";
message = 123456;
```

允许这种操作的编程语言称为“动态类型”（*dynamically typed*）的编程语言，意思是虽然编程语言中有不同的数据类型，但是你定义的变量并不会在定义后，被限制为某一数据类型。

在 JavaScript 中有八种基本的数据类型。这一章我们会学习数据类型的基本知识，在下一章我们会对他们一一进行详细讲解。

Number 类型

```
let n = 123;
n = 12.345;
```

`number` 类型代表整数和浮点数。

数字可以有很多操作，比如，乘法 `*`、除法 `/`、加法 `+`、减法 `-` 等等。

除了常规的数字，还包括所谓的“特殊数值”（“special numeric values”）也属于这种类型：`Infinity`、`-Infinity` 和 `NaN`。

- `Infinity` 代表数学概念中的 无穷大 ∞ 。是一个比任何数字都大的特殊值。

我们可以通过除以 0 来得到它：

```
alert( 1 / 0 ); // Infinity
```

或者在代码中直接使用它：

```
alert( Infinity ); // Infinity
```

- `NaN` 代表一个计算错误。它是一个不正确的或者一个未定义的数学操作所得到的结果，比如：

```
alert( "not a number" / 2 ); // NaN, 这样的除法是错误的
```

`NaN` 是粘性的。任何对 `NaN` 的进一步操作都会返回 `NaN`：

```
alert( "not a number" / 2 + 5 ); // NaN
```

所以，如果在数学表达式中有一个 `NaN`，会被传播到最终结果。

i 数学运算是安全的

在 JavaScript 中做数学运算是安全的。我们可以做任何事：除以 0，将非数字字符串视为数字，等等。

脚本永远不会因为一个致命的错误（“死亡”）而停止。最坏的情况下，我们会得到 `Nan` 的结果。

特殊的数值属于 “`number`” 类型。当然，对“特殊的数值”这个词的一般认识是，它们并不是数字。我们将在 [数字类型](#) 一节中学习数字的更多细节。

BigInt 类型

在 JavaScript 中，“`number`” 类型无法代表大于 2^{53} （或小于 -2^{53} ）的整数，这是其内部表示形式导致的技术限制。这大约是 16 位的十进制数字，因此在大多数情况下，这个限制不是问题，但有时我们需要很大的数字，例如用于加密或微秒精度的时间戳。

`BigInt` 类型是最近被添加到 JavaScript 语言中的，用于表示任意长度的整数。

通过将 `n` 附加到整数字段的末尾来创建 `BigInt`。

```
// 尾部的 "n" 表示这是一个 BigInt 类型
const bigInt = 1234567890123456789012345678901234567890n;
```

由于很少需要 `BigInt` 类型的数字，因此我们在单独的章节 [BigInt](#) 中专门对其进行介绍。

i 兼容性问题

目前 Firefox 和 Chrome 已经支持 `BigInt` 了，但 Safari/IE/Edge 还没有。

String 类型

JavaScript 中的字符串必须被括在引号里。

```
let str = "Hello";
let str2 = 'Single quotes are ok too';
let phrase = `can embed another ${str}`;
```

在 JavaScript 中，有三种包含字符串的方式。

1. 双引号: `"Hello"`.
2. 单引号: `'Hello'`.
3. 反引号: ``Hello``.

双引号和单引号都是“简单”引用，在 JavaScript 中两者几乎没有什么差别。

反引号是 **功能扩展** 引号。它们允许我们通过将变量和表达式包装在 `${...}` 中，来将它们嵌入到字符串中。例如：

```
let name = "John";

// 嵌入一个变量
alert(`Hello, ${name}!`); // Hello, John!

// 嵌入一个表达式
alert(`the result is ${1 + 2}`); // the result is 3
```

``${...}`` 内的表达式会被计算，计算结果会成为字符串的一部分。可以在 ``${...}`` 内放置任何东西：诸如名为 `name` 的变量，或者诸如 `1 + 2` 的算数表达式，或者其他一些更复杂的。

需要注意的是，这仅仅在反引号内有效，其他引号不允许这种嵌入。

```
alert("the result is ${1 + 2}"); // the result is ${1 + 2} (使用双引号则不会计算 `${...}` 中的内容)
```

我们会在 [字符串](#) 一节中学习字符串的更多细节。

① JavaScript 中没有 `character` 类型。

在一些语言中，单个字符有一个特殊的“`character`”类型，在 C 语言和 Java 语言中被称为“`char`”。

在 JavaScript 中没有这种类型。只有一种 `string` 类型，一个字符串可以包含一个或多个字符。

Boolean 类型（逻辑类型）

`boolean` 类型仅包含两个值：`true` 和 `false`。

这种类型通常用于存储表示 yes 或 no 的值：`true` 意味着“yes，正确”，`false` 意味着“no，不正确”。

比如：

```
let nameFieldChecked = true; // yes, name field is checked
let ageFieldChecked = false; // no, age field is not checked
```

布尔值也可作为比较的结果：

```
let isGreater = 4 > 1;

alert(isGreater); // true (比较的结果是 "yes")
```

更详细的内容将会在 [逻辑运算符](#) 一节中介绍。

“null” 值

特殊的 `null` 值不属于上述任何一种类型。

它构成了一个独立的类型，只包含 `null` 值：

```
let age = null;
```

相比较于其他编程语言，JavaScript 中的 `null` 不是一个“对不存在的 `object` 的引用”或者“`null` 指针”。

JavaScript 中的 `null` 仅仅是一个代表“无”、“空”或“值未知”的特殊值。

上面的代码表示，由于某些原因，`age` 是未知或空的。

“`undefined`”值

特殊值 `undefined` 和 `null` 一样自成类型。

`undefined` 的含义是 未被赋值。

如果一个变量已被声明，但未被赋值，那么它的值就是 `undefined`：

```
let x;  
  
alert(x); // 弹出 "undefined"
```

原理上来说，可以为任何变量赋值为 `undefined`：

```
let x = 123;  
  
x = undefined;  
  
alert(x); // "undefined"
```

.....但是不建议这样做。通常，使用使用 `null` 将一个“空”或者“未知”的值写入变量中，`undefined` 仅仅用于检验，例如查看变量是否被赋值或者其他类似的操作。

`object` 类型和 `symbol` 类型

`object` 类型是一个特殊的类型。

其他所有的数据类型都被称为“原生类型”，因为它们的值只包含一个单独的内容（字符串、数字或者其他）。相反，`object` 则用于储存数据集合和更复杂的实体。在充分了解原生类型之后，我们将会在[对象](#)一节中介绍 `object`。

`symbol` 类型用于创建对象的唯一标识符。我们在这里提到 `symbol` 类型是为了学习的完整性，但我们在学完 `object` 类型后再学习它。

`typeof` 运算符

`typeof` 运算符返回参数的类型。当我们想要分别处理不同类型值的时候，或者想快速进行数据类型检验时，非常有用。

它支持两种语法形式：

1. 作为运算符: `typeof x`。
2. 函数形式: `typeof(x)`。

换言之，有括号和没有括号，得到的结果是一样的。

对 `typeof x` 的调用会以字符串的形式返回数据类型：

```
typeof undefined // "undefined"  
typeof 0 // "number"  
typeof 10n // "bigint"  
typeof true // "boolean"  
typeof "foo" // "string"  
typeof Symbol("id") // "symbol"  
typeof Math // "object" (1)  
typeof null // "object" (2)  
typeof alert // "function" (3)
```

最后三行可能需要额外的说明：

1. `Math` 是一个提供数学运算的内建 `object`。我们会在 [数字类型](#) 一节中学习它。此处仅作为一个 `object` 的示例。
2. `typeof null` 的结果是 `"object"`。这其实是不对的。官方也承认了这是 `typeof` 运算符的问题，现在只是为了兼容性而保留了下来。当然，`null` 不是一个 `object`。`null` 有自己的类型，它是一个特殊值。再次强调，这是 JavaScript 语言的一个错误。
3. `typeof alert` 的结果是 `"function"`，因为 `alert` 在 JavaScript 语言中是一个函数。我们会在下一章学习函数，那时我们会了解到，在 JavaScript 语言中没有一个特别的 `"function"` 类型。函数隶属于 `object` 类型。但是 `typeof` 会对函数区分对待。这不是很正确的做法，但在实际编程中非常方便。

总结

JavaScript 中有八种基本的数据类型（译注：前七种为基本数据类型，也称为原始类型，而 `object` 为复杂数据类型）。

- `number` 用于任何类型的数字：整数或浮点数，在 $\pm 2^{53}$ 范围内的整数。
- `bigint` 用于任意长度的整数。
- `string` 用于字符串：一个字符串可以包含一个或多个字符，所以没有单独的单字符类型。
- `boolean` 用于 `true` 和 `false`。
- `null` 用于未知的值——只有一个 `null` 值的独立类型。
- `undefined` 用于未定义的值——只有一个 `undefined` 值的独立类型。
- `symbol` 用于唯一的标识符。

- `object` 用于更复杂的数据结构。

我们可以通过 `typeof` 运算符查看存储在变量中的数据类型。

- 两种形式: `typeof x` 或者 `typeof(x)`。
- 以字符串的形式返回类型名称, 例如 `"string"`。
- `typeof null` 会返回 `"object"` —— 这是 JavaScript 编程语言的一个错误, 实际上它并不是一个 `object`。

在接下来的章节中, 我们将重点介绍原生类型值, 当你掌握了原生数据类型后, 我们将继续学习 `object`。

类型转换

大多数情况下, 运算符和函数会自动将赋予他们的值转换为正确的类型。

比如, `alert` 会自动将任何值都转换为字符串以进行显示。算术运算符会将值转换为数字。

在某些情况下, 我们需要将值显式地转换为我们期望的类型。

① 对象还未纳入讨论中

本章不会讨论 `object` 类型。先学习原始类型, 之后我们会学习 `object` 类型。我们会在 [对象 — 原始值转换](#) 一节中学习对象的类型转换。

字符串转换

当我们需要一个字符串形式的值时, 就会进行字符串转换。

比如, `alert(value)` 将 `value` 转换为字符串类型, 然后显示这个值。

我们也可以显式地调用 `String(value)` 来将 `value` 转换为字符串类型:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // 现在, 值是一个字符串形式的 "true"
alert(typeof value); // string
```

字符串转换最明显。`false` 变成 `"false"`, `null` 变成 `"null"` 等。

数字型转换

在算术函数和表达式中, 会自动进行 `number` 类型转换。

比如, 当把除法 `/` 用于非 `number` 类型:

```
alert("6" / "2"); // 3, string 类型的值被自动转换成 number 类型后进行计算
```

我们也可以使用 `Number(value)` 显式地将这个 `value` 转换为 `number` 类型。

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // 变成 number 类型 123

alert(typeof num); // number
```

当我们从 `string` 类型源（如文本表单）中读取一个值，但期望输入一个数字时，通常需要进行显式转换。

如果该字符串不是一个有效的数字，转换的结果会是 `Nan`。例如：

```
let age = Number("an arbitrary string instead of a number");

alert(age); // NaN, 转换失败
```

`number` 类型转换规则：

值	变成.....
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>true</code> 和 <code>false</code>	<code>1</code> 和 <code>0</code>
<code>string</code>	去掉首尾空格后的纯数字字符串中含有的数字。如果剩余字符串为空，则转换结果为 <code>0</code> 。否则，将会从剩余字符串中“读取”数字。当类型转换出现 <code>error</code> 时返回 <code>NaN</code> 。

例子：

```
alert( Number(" 123  ") ); // 123
alert( Number("123z") );      // NaN (从字符串“读取”数字，读到 "z" 时出现错误)
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

请注意 `null` 和 `undefined` 在这有点不同：`null` 变成数字 `0`，`undefined` 变成 `NaN`。

（译注：此外，字符串转换为 `number` 类型时，除了 `undefined`、`null` 和 `boolean` 三种特殊情况，只有字符串是由空格和数字组成时，才能转换成功，否则会出现 `error` 返回 `NaN`。）

大多数数学运算符也执行这种转换，我们将在下一节中进行介绍。

布尔型转换

布尔 (`boolean`) 类型转换是最简单的一个。

它发生在逻辑运算中（稍后我们将进行条件判断和其他类似的东西），但是也可以通过调用 `Boolean(value)` 显式地进行转换。

转换规则如下：

- 直观上为“空”的值（如 `0`、空字符串、`null`、`undefined` 和 `NaN`）将变为 `false`。

- 其他值变成 `true`。

比如：

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false

alert( Boolean("hello") ); // true
alert( Boolean("") ); // false
```

⚠️ 请注意：包含 0 的字符串 "0" 是 true

一些编程语言（比如 PHP）视 `"0"` 为 `false`。但在 JavaScript 中，非空的字符串总是 `true`。

```
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // 空白，也是 true (任何非空字符串是 true)
```

总结

有三种常用的类型转换：转换为 `string` 类型、转换为 `number` 类型和转换为 `boolean` 类型。

字符串转换 —— 转换发生在输出内容的时候，也可以通过 `String(value)` 进行显式转换。原始类型值的 `string` 类型转换通常是很明显的。

数字型转换 —— 转换发生在进行算术操作时，也可以通过 `Number(value)` 进行显式转换。

数字型转换遵循以下规则：

值	变成.....
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>true / false</code>	<code>1 / 0</code>
<code>string</code>	“按原样读取”字符串，两端的空白会被忽略。空字符串变成 <code>0</code> 。转换出错则输出 <code>NaN</code> 。

布尔型转换 —— 转换发生在进行逻辑操作时，也可以通过 `Boolean(value)` 进行显式转换。

布尔型转换遵循以下规则：

值	变成.....
<code>0, null, undefined, NaN, ""</code>	<code>false</code>
其他值	<code>true</code>

上述的大多数规则都容易理解和记忆。人们通常会犯错误的值得注意的例子有以下几个：

- 对 `undefined` 进行数字型转换时，输出结果为 `NaN`，而非 `0`。

- 对 "`0`" 和只有空格的字符串（比如: " "）进行布尔型转换时，输出结果为 `true`。

我们在本小结没有讲 `object` 类型的转换。在我们学习完更多关于 JavaScript 的基本知识后，我们会在专门介绍 `object` 的章节 [对象 — 原始值转换](#) 中详细讲解 `object` 类型转换。

运算符

我们从学校里了解到过很多运算符，比如说加号 `+`、乘号 `*`、减号 `-` 等。

在这个章节，我们将关注一些在学校数学课程中没有涵盖的运算符。

术语：“一元运算符”，“二元运算符”，“运算元”

在正式开始前，我们先简单浏览一下常用术语。

- **运算元** —— 运算符应用的对象。比如说乘法运算 `5 * 2`，有两个运算元：左运算元 `5` 和右运算元 `2`。有时候人们也称其为“参数”而不是“运算元”。
- 如果一个运算符对应的只有一个运算元，那么它是 **一元运算符**。比如说一元负号运算符（`unary negation`）`-`，它的作用是对数字进行正负转换：

```
let x = 1;  
  
x = -x;  
alert( x ); // -1, 一元负号运算符生效
```

- 如果一个运算符拥有两个运算元，那么它是 **二元运算符**。减号还存在二元运算符形式：

```
let x = 1, y = 3;  
alert( y - x ); // 2, 二元运算符减号做减运算
```

严格地说，在上面的示例中，我们使用一个相同的符号表征了两个不同的运算符：负号运算符，即反转符号的一元运算符，减法运算符，是从另一个数减去一个数的二进制运算符。

字符串连接，二元运算符 `+`

下面，让我们看一下在学校数学课程范围外的 JavaScript 运算符特性。

通常，加号 `+` 用于求和。

但是如果加号 `+` 被应用于字符串，它将合并（连接）各个字符串：

```
let s = "my" + "string";  
alert(s); // mystring
```

注意：只要其中一个运算元是字符串，那么另一个运算元也将被转化为字符串。

举个例子：

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

可以看出，字符串在前和在后并不影响这个规则。简单来说：如果任一运算元是字符串，那么其它运算元也将被转化为字符串。

但是，请注意：运算符的运算方向是由左至右。如果是两个数字，后面再跟一个字符串，那么两个数字会先相加，再转化为字符串：

```
alert(2 + 2 + '1' ); // "41" 而不是 "221"
```

字符串连接和转化是二元运算符加号 `+` 的一个特性。其它的数学运算符都只对数字有效。通常，他们会把运算元转化为数字。

举个例子，减法和除法：

```
alert( 2 - '1' ); // 1
alert( '6' / '2' ); // 3
```

数字转化，一元运算符 `+`

加号 `+` 有两种形式。一种是上面我们刚刚讨论的二元运算符，还有一种是一元运算符。

一元运算符加号，或者说，加号 `+` 应用于单个值，对数字没有任何作用。但是如果运算元不是数字，加号 `+` 则会将其转化为数字。

例如：

```
// 对数字无效
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// 转化非数字
alert( +true ); // 1
alert( +"0" ); // 0
```

它的效果和 `Number(...)` 相同，但是更加简短。

我们经常会有将字符串转化为数字的需求。比如，如果我们正在从 `HTML` 表单中取值，通常得到的都是字符串。如果我们想对他们求和，该怎么办？

二元运算符加号会把他们合并成字符串：

```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // "23", 二元运算符加号合并字符串
```

如果我们想把它们当做数字对待，我们需要转化它们，然后再求和：

```
let apples = "2";
let oranges = "3";

// 在二元运算符加号起作用之前，所有的值都被转化为了数字
alert( +apples + +oranges ); // 5

// 更长的写法
// alert( Number(apples) + Number(oranges) ); // 5
```

从一个数学家的视角来看，大量的加号可能很奇怪。但是从一个程序员的视角，没什么好奇怪的：一元运算符加号首先起作用，他们将字符串转为数字，然后二元运算符加号对它们进行求和。

为什么一元运算符先于二元运算符作用于运算元？接下去我们将讨论到，这是由于它们拥有 **更高的优先级**。

运算符优先级

如果一个表达式拥有超过一个运算符，执行的顺序则由 **优先级** 决定。换句话说，所有的运算符中都隐含着优先级顺序。

从小学开始，我们就知道在表达式 `1 + 2 * 2` 中，乘法先于加法计算。这就是一个优先级问题。乘法比加法拥有 **更高的优先级**。

圆括号拥有最高优先级，所以如果我们将现有的运算顺序不满意，我们可以使用圆括号来修改运算顺序，就像这样：`(1 + 2) * 2`。

在 JavaScript 中有众多运算符。每个运算符都有对应的优先级数字。数字越大，越先执行。如果优先级相同，则按照由左至右的顺序执行。

这是一个摘抄自 Mozilla 的 [优先级表](#)（你没有必要把这全记住，但要记住一元运算符优先级高于二元运算符）：

优先级	名称	符号
...
17	一元加号	+
17	一元负号	-
15	乘号	*
15	除号	/
13	加号	+
13	减号	-
...
3	赋值符	=
...

我们可以看到，“一元加号运算符”的优先级是 `17`，高于“二元加号运算符”的优先级 `13`。这也是为什么表达式 `"+apples + +oranges"` 中的一元加号先生效，然后才是二元加法。

赋值运算符

我们知道赋值符号 `=` 也是一个运算符。从优先级表中可以看到它的优先级非常低，只有 `3`。

这也是为什么，当我们赋值时，比如 `x = 2 * 2 + 1`，所有的计算先执行，然后 `=` 才执行，将计算结果存储到 `x`。

```
let x = 2 * 2 + 1;  
alert( x ); // 5
```

链式赋值也是可以的：

```
let a, b, c;  
  
a = b = c = 2 + 2;  
  
alert( a ); // 4  
alert( b ); // 4  
alert( c ); // 4
```

链式赋值由右向左执行。首先执行最右侧表达式 `2 + 2`，然后将结果赋值给左侧：`c`、`b`、`a`。最后，所有的变量都共享一个值。

① 赋值运算符 "`=`" 会返回一个值

每个运算符都有一个返回值。对于以加号 `+` 或者乘号 `*` 为例的大部分运算符而言，这一点很显然。对于赋值运算符而言，这一点同样适用。

语句 `x = value` 把 `value` 的值写入 `x` 然后返回 `x`。

下面是一个在复杂语句中使用赋值的例子：

```
let a = 1;  
let b = 2;  
  
let c = 3 - (a = b + 1);  
  
alert( a ); // 3  
alert( c ); // 0
```

上面这个例子，`(a = b + 1)` 的结果是赋给 `a` 的值（也就是 `3`）。然后该值被用于进一步的运算。

这段代码是不是很好玩儿？我们应该理解它的原理，因为我们有时会在第三方库中见到这样的写法，但我们自己不应该这样写。这样的小技巧让代码变得整洁度和可读性都很差。

求余运算符 `%`

求余运算符 `%` 尽管看上去是个百分号，但它和百分数没有什么关系。

`a % b` 的结果是 `a` 除以 `b` 的余数。

举个例子：

```
alert( 5 % 2 ); // 1 是 5 / 2 的余数
alert( 8 % 3 ); // 2 是 8 / 3 的余数
alert( 6 % 3 ); // 0 是 6 / 3 的余数
```

幂运算符 `**`

幂运算符 `**` 是最近被加入到 JavaScript 中的。

对于自然数 `b`, `a ** b` 的结果是 `a` 与自己相乘 `b` 次。

举个例子：

```
alert( 2 ** 2 ); // 4 (2 * 2)
alert( 2 ** 3 ); // 8 (2 * 2 * 2)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2)
```

这个运算符对于 `a` 和 `b` 是非整数的情况依然适用。

例如：

```
alert( 4 ** (1/2) ); // 2 (1/2 幂相当于开方, 这是数学常识)
alert( 8 ** (1/3) ); // 2 (1/3 幂相当于开三次方)
```

自增/自减

对一个数进行加一、减一是最常见的数学运算符之一。

所以，对此有一些专门的运算符：

- **自增 `++`** 将变量与 `1` 相加：

```
let counter = 2;
counter++; // 和 counter = counter + 1 效果一样, 但是更简洁
alert( counter ); // 3
```

- **自减 `--`** 将变量与 `1` 相减：

```
let counter = 2;
counter--; // 和 counter = counter - 1 效果一样, 但是更简洁
alert( counter ); // 1
```



重要：

自增/自减只能应用于变量。试一下，将其应用于数值（比如 `5++`）则会报错。

运算符 `++` 和 `--` 可以置于变量前，也可以置于变量后。

- 当运算符置于变量后，被称为“后置形式”：`counter++`。
- 当运算符置于变量前，被称为“前置形式”：`++counter`。

两者都做同一件事：将变量 `counter` 与 `1` 相加。

那么他们有区别吗？有，但只有当我们使用 `++/-` 的返回值时才能看到区别。

详细点说。我们知道，所有的运算符都有返回值。自增/自减也不例外。前置形式返回一个新的值，但后置返回原来的值（做加法/减法之前的值）。

为了直观看到区别，看下面的例子：

```
let counter = 1;
let a = ++counter; // (*)  
  
alert(a); // 2
```

`(*)` 所在的行是前置形式 `++counter`，对 `counter` 做自增运算，返回的是新的值 `2`。因此 `alert` 显示的是 `2`。

下面让我们看看后置形式：

```
let counter = 1;
let a = counter++; // (*) 将 ++counter 改为 counter++  
  
alert(a); // 1
```

`(*)` 所在的行是后置形式 `counter++`，它同样对 `counter` 做加法，但是返回的是 **旧值**（做加法之前的值）。因此 `alert` 显示的是 `1`。

总结：

- 如果自增/自减的值不会被使用，那么两者形式没有区别：

```
let counter = 0;
counter++;
++counter;
alert(counter); // 2, 以上两行作用相同
```

- 如果我们想要对变量进行自增操作，**并且** 需要立刻使用自增后的值，那么我们需要使用前置形式：

```
let counter = 0;
alert(++counter); // 1
```

- 如果我们想要将一个数加一，但是我们想使用其自增之前的值，那么我们需要使用后置形式：

```
let counter = 0;  
alert( counter++ ); // 0
```

① 自增/自减和其它运算符的对比

`++/-` 运算符同样可以在表达式内部使用。它们的优先级比绝大部分的算数运算符要高。

举个例子：

```
let counter = 1;  
alert( 2 * ++counter ); // 4
```

与下方例子对比：

```
let counter = 1;  
alert( 2 * counter++ ); // 2, 因为 counter++ 返回的是“旧值”
```

尽管从技术层面上来说可行，但是这样的写法会降低代码的可阅读性。在一行上做多个操作——这样并不好。

当阅读代码时，快速的视觉“纵向”扫描会很容易漏掉 `counter++`，这样的自增操作并不明显。

我们建议用“一行一个行为”的模式：

```
let counter = 1;  
alert( 2 * counter );  
counter++;
```

位运算符

位运算符把运算元当做 32 位整数，并在它们的二进制表现形式上操作。

这些运算符不是 JavaScript 特有的。大部分的编程语言都支持这些运算符。

下面是位运算符：

- 按位与 (`&`)
- 按位或 (`|`)
- 按位异或 (`^`)
- 按位非 (`~`)
- 左移 (`<<`)
- 右移 (`>>`)
- 无符号右移 (`>>>`)

这些操作使用得非常少。为了理解它们，我们需要探讨底层的数字表达形式，现在不是做这个的最好时机。尤其是我们现在不会立刻使用它。如果你感兴趣，可以阅读 MDN 中的 [位运算符](#) 相关文章。当有相关实际需求的时候再去阅读是更明智的选择。

修改并替换

我们经常需要对一个变量进行操作，并把计算得到的新结果存储在这个变量中。

举个例子：

```
let n = 2;  
n = n + 5;  
n = n * 2;
```

这个操作可以通过使用运算符 `+=` 和 `*=` 进行简化：

```
let n = 2;  
n += 5; // now n = 7 (同 n = n + 5)  
n *= 2; // now n = 14 (同n = n * 2)  
  
alert( n ); // 14
```

简短的“修改并替换”运算符对所有的运算符包括位运算符都有效：`/=`、`-=` 等等。

这些运算符和正常的赋值运算符拥有相同的优先级，因此它们会在其它大部分运算完成之后运行：

```
let n = 2;  
  
n *= 3 + 5;  
  
alert( n ); // 16 (右侧计算首先进行，和 n *= 8 相同)
```

逗号运算符

逗号运算符 `,` 是最少见最不常使用的运算符之一。有时候它会被用来写更简短的代码，因此为了能够理解代码，我们需要了解它。

逗号运算符能让我们处理多个语句，使用 `,` 将它们分开。每个语句都运行了，但是只有最后的语句的结果会被返回。

举个例子：

```
let a = (1 + 2, 3 + 4);  
  
alert( a ); // 7 (3 + 4 的结果)
```

这里，第一个语句 `1 + 2` 运行了，但是它的结果被丢弃了。随后计算 `3 + 4`，并且该计算结果被返回。

① 逗号运算符的优先级非常低

请注意逗号运算符的优先级非常低，比 `=` 还要低，因此上面你的例子中圆括号非常重要。

如果没有圆括号：`a = 1 + 2, 3 + 4` 会先执行 `+`，将数值相加得到 `a = 3, 7`，然后赋值运算符 `=` 执行，'a = 3'，然后逗号之后的数值 `7` 不会再执行，它被忽略掉了。相当于 `(a = 1 + 2), 3 + 4`。

为什么我们需要这样一个运算符，它只返回最后一个值呢？

有时候，人们会使用它把几个行为放在一行上来进行复杂的运算。

举个例子：

```
// 一行上有三个运算符
for (a = 1, b = 3, c = a * b; a < 10; a++) {
  ...
}
```

这样的技巧在许多 JavaScript 框架中都有使用，这也是为什么我们提到它。但是通常它并不能提升代码的可读性，使用它之前，我们要想清楚。

值的比较

我们知道，在数学中有很多用于比较大小的运算符：

- 大于 / 小于：`a > b`, `a < b`。
- 大于等于 / 小于等于：`a >= b`, `a <= b`。
- 检查两个值的相等：`a == b`（注意表达式中是两个等号 `=`，若写为单个等号 `a = b` 则表示赋值）。
- 检查两个值不相等，在数学中使用 `≠` 符号，而在 JavaScript 中则通过在赋值符号前加感叹号表示：`a != b`。

比较结果为 Boolean 类型

和其他运算符一样，比较运算符也会有返回值，返回值为布尔值（Boolean）。

- `true` —— 表示“yes（是）”，“correct（正确）”或“the truth（真相）”。
- `false` —— 表示“no（否）”，“wrong（错误）”或“not the truth（非真相）”。

示例：

```
alert( 2 > 1 ); // true (正确)
alert( 2 == 1 ); // false (错误)
alert( 2 != 1 ); // true (正确)
```

和其他类型的值一样，比较的结果可以被赋值给任意变量：

```
let result = 5 > 4; // 把比较的结果赋值给 result
alert( result ); // true
```

字符串比较

在比较字符串的大小时，JavaScript 会使用“字典（dictionary）”或“词典（lexicographical）”顺序进行判定。

换言之，字符串是按字符（母）逐个进行比较的。

例如：

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```

字符串的比较算法非常简单：

1. 首先比较两个字符串的首位字符大小。
2. 如果一方字符较大（或较小），则该字符串大于（或小于）另一个字符串。算法结束。
3. 否则，如果两个字符串的首位字符相等，则继续取出两个字符串各自的后一位字符进行比较。
4. 重复上述步骤进行比较，直到比较完成某字符串的所有字符为止。
5. 如果两个字符串的字符同时用完，那么则判定它们相等，否则未结束（还有未比较的字符）的字符串更大。

在上面的例子中，`'Z' > 'A'` 在算法的第 1 步就得到了返回结果，而字符串 `Glow` 与 `Glee` 则继续逐个字符比较：

1. `G` 和 `G` 相等。
2. `l` 和 `l` 相等。
3. `o` 比 `e` 大，算法停止，第一个字符串大于第二个。

➊ 非真正的字典顺序，而是 Unicode 编码顺序

在上面的算法中，比较大小的逻辑与字典或电话簿中的排序很像，但也不完全相同。

比如说，字符串比较对字母大小写是敏感的。大写的 `"A"` 并不等于小写的 `"a"`。哪一个更大呢？实际上小写的 `"a"` 更大。这是因为在 JavaScript 使用的内部编码表中（Unicode），小写字母的字符索引值更大。我们会在 [字符串](#) 这章讨论更多关于字符串的细节。

不同类型间的比较

当对不同类型的值进行比较时，JavaScript 会首先将其转化为数字（number）再判定大小。

例如：

```
alert( '2' > 1 ); // true, 字符串 '2' 会被转化为数字 2
alert( '01' == 1 ); // true, 字符串 '01' 会被转化为数字 1
```

对于布尔类型值，`true` 会被转化为 `1`、`false` 转化为 `0`。

例如：

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

① 一个有趣的现象

有时候，以下两种情况会同时发生：

- 若直接比较两个值，其结果是相等的。
- 若把两个值转为布尔值，它们可能得出完全相反的结果，即一个是 `true`，一个是 `false`。

例如：

```
let a = 0;
alert( Boolean(a) ); // false

let b = "0";
alert( Boolean(b) ); // true

alert(a == b); // true!
```

对于 JavaScript 而言，这种现象其实挺正常的。因为 JavaScript 会把待比较的值转化为数字后再做比较（因此 `"0"` 变成了 `0`）。若只是将一个变量转化为 `Boolean` 值，则会使用其他的类型转换规则。

严格相等

普通的相等性检查 `==` 存在一个问题，它不能区分出 `0` 和 `false`：

```
alert( 0 == false ); // true
```

也同样无法区分空字符串和 `false`：

```
alert( '' == false ); // true
```

这是因为在比较不同类型的值时，处于相等判断符号 `==` 两侧的值会先被转化为数字。空字符串和 `false` 也是如此，转化后它们都为数字 `0`。

如果我们需要区分 `0` 和 `false`，该怎么办？

严格相等运算符 `===` 在进行比较时不会做任何的类型转换。

换句话说，如果 `a` 和 `b` 属于不同的数据类型，那么 `a === b` 不会做任何的类型转换而立刻返回 `false`。

让我们试试：

```
alert( 0 === false ); // false, 因为被比较值的数据类型不同
```

同样的，与“不相等”符号 `!=` 类似，“严格不相等”表示为 `!==`。

严格相等的运算符虽然写起来稍微长一些，但是它能够很清楚地显示代码意图，降低你犯错的可能性。

对 `null` 和 `undefined` 进行比较

当使用 `null` 或 `undefined` 与其他值进行比较时，其返回结果常常出乎你的意料。

当使用严格相等 `==` 比较二者时

它们不相等，因为它们属于不同的类型。

```
alert( null === undefined ); // false
```

当使用非严格相等 `==` 比较二者时

JavaScript 存在一个特殊的规则，会判定它们相等。他们俩就像“一对恋人”，仅仅等于对方而不等于其他任何的值（只在非严格相等下成立）。

```
alert( null == undefined ); // true
```

当使用数学式或其他比较方法 `<` `>` `<=` `>=` 时：

`null/undefined` 会被转化为数字：`null` 被转化为 `0`，`undefined` 被转化为 `NaN`。

下面让我们看看，这些规则会带来什么有趣的现象。同时更重要的是，我们需要从中学会如何远离这些特性带来的“陷阱”。

奇怪的结果：`null vs 0`

通过比较 `null` 和 `0` 可得：

```
alert( null > 0 ); // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
```

是的，上面的结果完全打破了你对数学的认识。在最后一行代码显示“`null` 大于等于 `0`”的情况下，前两行代码中一定会有一个是正确的，然而事实表明它们的结果都是 `false`。

为什么会出现这种反常结果，这是因为相等性检查 `==` 和普通比较符 `>` `<` `>=` `<=` 的代码逻辑是相互独立的。进行值的比较时，`null` 会被转化为数字，因此它被转化为了 `0`。这就是为什么（3）中 `null >= 0` 返回值是 `true`，（1）中 `null > 0` 返回值是 `false`。

另一方面，`undefined` 和 `null` 在相等性检查 `==` 中不会进行任何的类型转换，它们有自己独立的比较规则，所以除了它们之间互等外，不会等于任何其他的值。这就解释了为什么（2）中

`null == 0` 会返回 `false`。

特立独行的 `undefined`

`undefined` 不应该被与其他值进行比较：

```
alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

为何它看起来如此厌恶 `0`? 返回值都是 `false`!

原因如下：

- (1) 和 (2) 都返回 `false` 是因为 `undefined` 在比较中被转换为了 `NaN`，而 `NaN` 是一个特殊的数值型值，它与任何值进行比较都会返回 `false`。
- (3) 返回 `false` 是因为这是一个相等性检查，而 `undefined` 只与 `null` 相等，不会与其他值相等。

规避错误

我们为何要研究上述示例？我们需要时刻记得这些古怪的规则吗？不，其实不需要。虽然随着代码写得越来越多，我们对这些规则也都会烂熟于胸，但是我们需要更为可靠的方法来避免潜在的问题：

除了严格相等 `==` 外，其他凡是有 `undefined/null` 参与的比较，我们都需要额外小心。

除非你非常清楚自己在做什么，否则永远不要使用 `>= > < <=` 去比较一个可能为 `null/undefined` 的变量。对于取值可能是 `null/undefined` 的变量，请按需要分别检查它的取值情况。

总结

- 比较运算符始终返回布尔值。
- 字符串的比较，会按照“词典”顺序逐字符地比较大小。
- 当对不同类型的值进行比较时，它们会先被转化为数字（不包括严格相等检查）再进行比较。
- 在非严格相等 `==` 下，`null` 和 `undefined` 相等且各自不等于任何其他的值。
- 在使用 `>` 或 `<` 进行比较时，需要注意变量可能为 `null/undefined` 的情况。比较好的方法是单独检查变量是否等于 `null/undefined`。

交互：`alert`、`prompt` 和 `confirm`

本教程的这部分内容主要使用原生 JavaScript，你无需针对特定环境进行调整。

但我们仍然会使用浏览器作为演示环境。所以我们至少应该知道一些用户界面函数。在这一节，我们一起来熟悉一下浏览器中 `alert`、`prompt` 和 `confirm` 函数的用法。

`alert`

语法：

```
alert(message);
```

运行这行代码，浏览器会弹出一个信息弹窗并暂停脚本，直到用户点击了“确定”。

举个例子：

```
alert("Hello");
```

弹出的这个带有信息的小窗口被称为 **模态窗**。“modal”意味着用户不能与页面的其他部分（例如点击其他按钮等）进行交互，直到他们处理完窗口。在上面示例这种情况下——直到用户点击“确定”按钮。

prompt

prompt 函数接收两个参数：

```
result = prompt(title, [default]);
```

浏览器会显示一个带有文本消息的模态窗口，还有 **input** 框和确定/取消按钮。

title

显示给用户的文本

default

可选的第二个参数，指定 **input** 框的初始值。

用户可以在 **prompt** 对话框的 **input** 框内输入一些内容，然后点击确定。或者他们可以通过按“取消”按钮或按下键盘的 **Esc** 键，以取消输入。

prompt 将返回用户在 **input** 框内输入的文本，如果用户取消了输入，则返回 **null**。

举个例子：

```
let age = prompt('How old are you?', 100);

alert(`You are ${age} years old!`); // You are 100 years old!
```

IE 浏览器会提供默认值

第二个参数是可选的。但是如果不提供的话，Internet Explorer 会把 "undefined" 插入到 prompt。

我们可以在 Internet Explorer 中运行下面这行代码来看看效果：

```
let test = prompt("Test");
```

所以，为了 prompt 在 IE 中有好的效果，我们建议始终提供第二个参数：

```
let test = prompt("Test", ''); // <-- for IE
```

confirm

语法：

```
result = confirm(question);
```

confirm 函数显示一个带有 question 以及确定和取消两个按钮的模态窗口。

点击确定返回 true，点击取消返回 false。

例如：

```
let isBoss = confirm("Are you the boss?");

alert( isBoss ); // 如果“确定”按钮被按下，则显示 true
```

总结

我们学习了与用户交互的 3 个浏览器的特定函数：

alert

显示信息。

prompt

显示信息要求用户输入文本。点击确定返回文本，点击取消或按下 Esc 键返回 null。

confirm

显示信息等待用户点击确定或取消。点击确定返回 true，点击取消或按下 Esc 键返回 false。

这些方法都是模态的：它们暂停脚本的执行，并且不允许用户与该页面的其余部分进行交互，直到窗口被解除。

上述所有方法共有两个限制：

1. 模态窗口的确切位置由浏览器决定。通常在页面中心。
2. 窗口的确切外观也取决于浏览器。我们不能修改它。

这就是简单的代价。还有其他一些方法可以显示更漂亮的窗口，并与用户进行更丰富的交互，但如果“花里胡哨”不是非常重要，那使用本节讲的这些方法也挺好。

条件运算符：if 和 '?'

有时我们需要根据不同条件执行不同的操作。

我们可以使用 `if` 语句和条件运算符 `?`（也称为“问号”运算符）来实现。

“if”语句

`if(...)` 语句计算括号里的条件表达式，如果计算结果是 `true`，就会执行对应的代码块。

例如：

```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');
if (year == 2015) alert( 'You are right!' );
```

在上面这个例子中，条件是一个简单的相等性检查 (`year == 2015`)，但它还可以更复杂。

如果有多个语句要执行，我们必须将要执行的代码块封装在大括号内：

```
if (year == 2015) {
  alert( "That's correct!" );
  alert( "You're so smart!" );
}
```

建议每次使用 `if` 语句都用大括号 `{}` 来包装代码块，即使只有一条语句。这样可以提高代码可读性。

布尔转换

`if (...)` 语句会计算圆括号内的表达式，并将计算结果转换为布尔型。

让我们回顾一下 [类型转换](#) 一章中的转换规则：

- 数字 `0`、空字符串 `""`、`null`、`undefined` 和 `Nan` 都会被转换成 `false`。因为他们被称为“falsy”值。
- 其他值被转换为 `true`，所以它们被称为“truthy”。

所以，下面这个条件下的代码永远不会执行：

```
if (0) { // 0 是 falsy
  ...
}
```

```
}
```

.....但下面的条件——始终有效:

```
if (1) { // 1 是 truthy
  ...
}
```

我们也可以将未计算的布尔值传入 `if` 语句, 像这样:

```
let cond = (year == 2015); // 相等运算符的结果是 true 或 false

if (cond) {
  ...
}
```

“else”语句

`if` 语句有时会包含一个可选的 “`else`” 块。如果判断条件不成立, 就会执行它内部的代码。

例如:

```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');

if (year == 2015) {
  alert('You guessed it right!');
} else {
  alert('How can you be so wrong?'); // 2015 以外的任何值
}
```

多个条件: “`else if`”

有时我们需要测试一个条件的几个变体。我们可以通过使用 `else if` 子句实现。

例如:

```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');

if (year < 2015) {
  alert('Too early...');

} else if (year > 2015) {
  alert('Too late');

} else {
  alert('Exactly!');
}
```

在上面的代码中, JavaScript 先先检查 `year < 2015`。如果条件不符合, 就会转到下一个条件 `year > 2015`。如果这个条件也不符合, 则会显示最后一个 `alert`。

可以有更多的 `else if` 块。结尾的 `else` 是可选的。

条件运算符 ‘?’

有时我们需要根据一个条件去赋值一个变量。

如下所示：

```
let accessAllowed;
let age = prompt('How old are you?', '');

if (age > 18) {
    accessAllowed = true;
} else {
    accessAllowed = false;
}

alert(accessAllowed);
```

所谓的“条件”或“问号”运算符让我们可以更简短地达到目的。

这个运算符通过问号 `?` 表示。有时它被称为三元运算符，被称为“三元”是因为该运算符中有三个操作数。实际上它是 JavaScript 中唯一一个有这么多操作数的运算符。

语法：

```
let result = condition ? value1 : value2;
```

计算条件结果，如果结果为真，则返回 `value1`，否则返回 `value2`。

例如：

```
let accessAllowed = (age > 18) ? true : false;
```

技术上讲，我们可以省略 `age > 18` 外面的括号。问号运算符的优先级较低，所以它会在比较运算符 `>` 后执行。

下面这个示例会执行和前面那个示例相同的操作：

```
// 比较运算符 “age > 18” 首先执行
// (不需要将其包含在括号中)
let accessAllowed = age > 18 ? true : false;
```

但括号可以使代码可读性更强，所以我们建议使用它们。

ⓘ 请注意:

在上面的例子中，你可以不使用问号运算符，因为比较本身就返回 `true/false`：

```
// 下面代码同样可以实现
let accessAllowed = age > 18;
```

多个‘?’

使用一系列问号 `?` 运算符可以返回一个取决于多个条件的值。

例如：

```
let age = prompt('age?', 18);

let message = (age < 3) ? 'Hi, baby!' :
  (age < 18) ? 'Hello!' :
  (age < 100) ? 'Greetings!' :
  'What an unusual age!';

alert(message);
```

可能很难一下子看出发生了什么。但经过仔细观察，我们可以看到它只是一个普通的检查序列。

1. 第一个问号检查 `age < 3`。
2. 如果为真 — 返回 `'Hi, baby!'`。否则，会继续执行冒号 :" 后的表达式，检查 `age < 18`。
3. 如果为真 — 返回 `'Hello!'`。否则，会继续执行下一个冒号 :" 后的表达式，检查 `age < 100`。
4. 如果为真 — 返回 `'Greetings!'`。否则，会继续执行最后一个冒号 :" 后面的表达式，返回 `'What an unusual age!'`。

这是使用 `if..else` 实现上面的逻辑的写法：

```
if (age < 3) {
  message = 'Hi, baby!';
} else if (age < 18) {
  message = 'Hello!';
} else if (age < 100) {
  message = 'Greetings!';
} else {
  message = 'What an unusual age!';
}
```

‘?’ 的非常规使用

有时可以使用问号 `?` 来代替 `if` 语句：

```
let company = prompt('Which company created JavaScript?', '');
(company == 'Netscape') ?
  alert('Right!') : alert('Wrong.');
```

根据条件 `company == 'Netscape'`，要么执行 `?` 后面的第一个表达式并显示对应内容，要么执行第二个表达式并显示对应内容。

在这里我们不是把结果赋值给变量。而是根据条件执行不同的代码。

不建议这样使用问号运算符。

这种写法比 `if` 语句更短，对一些程序员很有吸引力。但它的可读性差。

下面是使用 `if` 语句实现相同功能的代码，进行下比较：

```
let company = prompt('Which company created JavaScript?', '');
if (company == 'Netscape') {
  alert('Right!');
} else {
  alert('Wrong.');
}
```

因为我们的眼睛垂直扫描代码。所以，跨越几行的代码块比长而水平的代码更易于理解。

问号 `?` 的作用是根据条件返回一个或另一个值。请正确使用它。当需要执行不同的代码分支时，请使用 `if`。

逻辑运算符

JavaScript 里有三个逻辑运算符：`||`（或），`&&`（与），`!`（非）。

虽然他们被称为“逻辑”运算符，但这些运算符却可以被应用于任意类型的值，而不仅仅是布尔值。他们的结果也同样可以是任意类型。

让我们来详细看一下。

`||`（或）

两个竖线符号表示了“或”运算：

```
result = a || b;
```

在传统的编程中，逻辑或仅能够操作布尔值。如果参与运算的任意一个参数为 `true`，返回的结果就为 `true`，否则返回 `false`。

在 JavaScript 中，逻辑运算符更加灵活强大。但是首先我们看一下操作数是布尔值的时候发生了什么。

下面是四种可能的逻辑组合：

```
alert( true || true ); // true
alert( false || true ); // true
alert( true || false ); // true
alert( false || false ); // false
```

正如我们所见，除了两个操作数都是 `false` 的情况，结果都是 `true`。

如果操作数不是布尔值，那么它将会被转化为布尔值来参与运算。

例如，数字 `1` 将会被作为 `true`，数字 `0` 则作为 `false`：

```
if (1 || 0) { // 工作原理相当于 if( true || false )
  alert('truthy!');
}
```

大多数情况，逻辑或 `||` 会被用在 `if` 语句中，用来测试是否有 **任何** 给定的条件为 `true`。

例如：

```
let hour = 9;

if (hour < 10 || hour > 18) {
  alert('The office is closed.');
}
```

我们可以传入更多的条件：

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert('The office is closed.'); // 是周末
}
```

或运算寻找第一个真值

上文提到的逻辑处理多少有些传统了。下面让我们看看 JavaScript 的“附加”特性。

拓展的算法如下所示。

给定多个参与或运算的值：

```
result = value1 || value2 || value3;
```

或运算符 `||` 做了如下的事情：

- 从左到右依次计算操作数。
- 处理每一个操作数时，都将其转化为布尔值。如果结果是 `true`，就停止计算，返回这个操作数的初始值。

- 如果所有的操作数都被计算过（也就是，转换结果都是 `false`），则返回最后一个操作数。

返回的值是操作数的初始形式，不会做布尔转换。

也就是说，一个或 `"||"` 运算的链，将返回第一个真值，如果不存在真值，就返回该链的最后一个值。

例如：

```
alert( 1 || 0 ); // 1 (1 是真值)
alert( true || 'no matter what' ); // (true 是真值)

alert( null || 1 ); // 1 (1 是第一个真值)
alert( null || 0 || 1 ); // 1 (第一个真值)
alert( undefined || null || 0 ); // 0 (所有的转化结果都是 false, 返回最后一个值)
```

与“纯粹的、传统的、仅仅处理布尔值的或运算”相比，这个规则就引起了一些很有趣的用法。

1. 获取变量列表或者表达式的第一真值。

假设我们有几个变量，它们可能包含某些数据或者是 `null/undefined`。我们需要选出第一个包含数据的变量。

我们可以这样应用或运算 `||`：

```
let currentUser = null;
let defaultUser = "John";

let name = currentUser || defaultUser || "unnamed";

alert( name ); // 选出了 "John" – 第一个真值
```

如果 `currentUser` 和 `defaultUser` 都是假值，那么结果就是 `"unnamed"`。

2. 短路取值。

操作数不仅仅可以是值，还可以是任意表达式。或运算会从左到右计算并测试每个操作数。当找到第一个真值，计算就会停止，并返回这个值。这个过程就叫做“短路取值”，因为它尽可能地减少从左到右计算的次数。

当表达式作为第二个参数并且有一定的副作用（`side effects`），比如变量赋值的时候，短路取值的情况就清楚可见。

如果我们运行下面的例子，`x` 将不会被赋值：

```
let x;

true || (x = 1);

alert(x); // undefined, 因为 (x = 1) 没有被执行
```

如果第一个参数是 `false`，或运算将会继续，并计算第二个参数，也就会运行赋值操作。

```
let x;  
  
false || (x = 1);  
  
alert(x); // 1
```

赋值操作只是一个很简单的情况。可能有副作用，如果计算没有到达，副作用就不会发生。

正如我们所见，这种用法是“`if` 语句的简短方式”。第一个操作数被转化为布尔值，如果是假，那么第二个参数就会被执行。

大多数情况下，最好使用“常规的”`if` 语句，这样代码可读性更高，但是有时候这种方式会很简洁。

&& (与)

两个 `&` 符号表示 `&&` 与操作：

```
result = a && b;
```

传统的编程中，当两个操作数都是真值，与操作返回 `true`，否则返回 `false`：

```
alert( true && true ); // true  
alert( false && true ); // false  
alert( true && false ); // false  
alert( false && false ); // false
```

使用 `if` 语句的例子：

```
let hour = 12;  
let minute = 30;  
  
if (hour == 12 && minute == 30) {  
  alert('Time is 12:30');  
}
```

就像或运算一样，与运算的操作数可以是任意类型的值：

```
if (1 && 0) { // 作为 true && false 来执行  
  alert("won't work, because the result is falsy");  
}
```

与操作寻找第一个假值

给出多个参加与运算的值：

```
result = value1 && value2 && value3;
```

与运算 `&&` 做了如下的事：

- 从左到右依次计算操作数。
- 将处理每一个操作数时，都将其转化为布尔值。如果结果是 `false`，就停止计算，并返回这个操作数的初始值。
- 如果所有的操作数都被计算过（也就是，转换结果都是 `true`），则返回最后一个操作数。

换句话说，与运算符返回第一个假值，如果没有假值就返回最后一个值。

上面的规则和或运算很像。区别就是与运算返回第一个假值而或操作返回第一个真值。

例如：

```
// 如果第一个运算符是真值,  
// 与操作返回第二个操作数:  
alert( 1 && 0 ); // 0  
alert( 1 && 5 ); // 5  
  
// 如果第一个运算符是假值,  
// 与操作直接返回它。第二个操作数被忽略  
alert( null && 5 ); // null  
alert( 0 && "no matter what" ); // 0
```

我们也可以在一行代码上串联多个值。查看第一个假值是否被返回：

```
alert( 1 && 2 && null && 3 ); // null
```

如果所有的值都是真值，最后一个值将会被返回：

```
alert( 1 && 2 && 3 ); // 3, 最后一个值
```

① 与运算 `&&` 在或运算符 `||` 之前执行

与运算 `&&` 的优先级比或运算 `||` 要高。

所以代码 `a && b || c && d` 完全跟 `&&` 表达式加了括号一样：`(a && b) || (c && d)`。

就像或运算一样，与运算 `&&` 有时候能够代替 `if`。

例如：

```
let x = 1;  
  
(x > 0) && alert( 'Greater than zero!' );
```

`&&` 右边的代码只有运算抵达到那里才能被执行。也就是，当且仅当 `(x > 0)` 返回了真值。

所以我们基本可以类似地得到：

```
let x = 1;

if (x > 0) {
  alert('Greater than zero!');
}
```

带 `&&` 的代码变体看上去更短。但是 `if` 的含义更明显，可读性也更高。

所以建议是根据目的选择代码的结构。需要条件判断就用 `if`，需要与运算就用 `&&`。

! (非)

感叹符号 `!` 表示布尔非运算。

语法相当简单：

```
result = !value;
```

运算符接受一个参数，并按如下运作：

1. 将操作数转化为布尔类型：`true/false`。
2. 返回相反的值。

例如：

```
alert(!true); // false
alert(!0); // true
```

两个非运算 `!!` 有时候用来将某个值转化为布尔类型：

```
alert (!!"non-empty string"); // true
alert (!!null); // false
```

也就是，第一个非运算将该值转化为布尔类型并取反，第二个非运算再次取反。最后我们就得到了一个任意值到布尔值的转化。

有更多详细的方法可以完成同样的事——一个内置的 `Boolean` 函数：

```
alert(Boolean("non-empty string")); // true
alert(Boolean(null)); // false
```

非运算符 `!` 的优先级在所有逻辑运算符里面最高，所以它总是在 `&&` 和 `||` 前执行。

循环：while 和 for

我们经常需要重复执行一些操作。

例如，我们需要将列表中的商品逐个输出，或者运行相同的代码将数字 1 到 10 逐个输出。

循环 是一种重复运行同一代码的方法。

“while” 循环

`while` 循环的语法如下：

```
while (condition) {  
    // 代码  
    // 所谓的“循环体”  
}
```

当 `condition` 为 `true` 时，执行循环体的 `code`。

例如，以下将循环输出当 `i < 3` 时的 `i` 值：

```
let i = 0;  
while (i < 3) { // 依次显示 0、1 和 2  
    alert(i);  
    i++;  
}
```

循环体的单次执行叫作 **一次迭代**。上面示例中的循环进行了三次迭代。

如果上述示例中没有 `i++`，那么循环（理论上）会永远重复执行下去。实际上，浏览器提供了阻止这种循环的方法，我们可以通过终止进程，来停掉服务器端的 JavaScript。

任何表达式或变量都可以是循环条件，而不仅仅是比较。在 `while` 中的循环条件会被计算，计算结果会被转化为布尔值。

例如，`while (i != 0)` 可简写为 `while (i)`：

```
let i = 3;  
while (i) { // 当 i 变成 0 时，条件为 false，循环终止  
    alert(i);  
    i--;  
}
```

① 单行循环体不需要大括号

如果循环体只有一条语句，则可以省略大括号 `{...}`：

```
let i = 3;  
while (i) alert(i--);
```

“do...while” 循环

使用 `do..while` 语法可以将条件检查移至循环体 **下面**:

```
do {  
    // 循环体  
} while (condition);
```

循环首先执行循环体，然后检查条件，当条件为真时，重复执行循环体。

例如:

```
let i = 0;  
do {  
    alert( i );  
    i++;  
} while (i < 3);
```

这种形式的语法很少使用，除非你希望不管条件是否为真，循环体 **至少执行一次**。通常我们更倾向于使用另一个形式: `while(...){...}`。

“for” 循环

`for` 循环更加复杂，但它是最常使用的循环形式。

`for` 循环看起来就像这样:

```
for (begin; condition; step) {  
    // .....循环体.....  
}
```

我们通过示例来了解一下这三个部分的含义。下述循环从 `i` 等于 `0` 到 `3`（但不包括 `3`）运行 `alert(i)`:

```
for (let i = 0; i < 3; i++) { // 结果为 0、1、2  
    alert(i);  
}
```

我们逐个部分分析 `for` 循环:

语句段

begin	<code>i = 0</code>	进入循环时执行一次。
condition	<code>i < 3</code>	在每次循环迭代之前检查，如果为 <code>false</code> ，停止循环。
body (循环体)	<code>alert(i)</code>	条件为真时，重复运行。
step	<code>i++</code>	在每次循环体迭代后执行。

一般循环算法的工作原理如下:

```
开始运行
→ (如果 condition 成立 → 运行 body 然后运行 step)
→ (如果 condition 成立 → 运行 body 然后运行 step)
→ (如果 condition 成立 → 运行 body 然后运行 step)
→ ...
```

所以，`begin` 执行一次，然后进行迭代：每次检查 `condition` 后，执行 `body` 和 `step`。

如果你这是第一次接触循环，那么回到这个例子，在一张纸上重现它逐步运行的过程，可能会对你有所帮助。

以下是在这个示例中发生的事：

```
// for (let i = 0; i < 3; i++) alert(i)

// 开始
let i = 0
// 如果条件为真，运行下一步
if (i < 3) { alert(i); i++ }
// 如果条件为真，运行下一步
if (i < 3) { alert(i); i++ }
// 如果条件为真，运行下一步
if (i < 3) { alert(i); i++ }
// .....结束，因为现在 i == 3
```

❶ 内联变量声明

这里“计数”变量 `i` 是在循环中声明的。这叫做“内联”变量声明。这样的变量只在循环中可见。

```
for (let i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // 错误，没有这个变量。
```

除了定义一个变量，我们也可以使用现有的变量：

```
let i = 0;

for (i = 0; i < 3; i++) { // 使用现有的变量
  alert(i); // 0, 1, 2
}

alert(i); // 3, 可见，因为是在循环之外声明的
```

省略语句段

`for` 循环的任何语句段都可以被省略。

例如，如果我们在循环开始时不需要做任何事，我们就可以省略 `begin` 语句段。

就像这样：

```
let i = 0; // 我们已经声明了 i 并对它进行了赋值

for (; i < 3; i++) { // 不再需要 "begin" 语句段
    alert( i ); // 0, 1, 2
}
```

我们也可以移除 `step` 语句段:

```
let i = 0;

for (; i < 3;) {
    alert( i++ );
}
```

该循环与 `while (i < 3)` 等价。

实际上我们可以删除所有内容，从而创建一个无限循环:

```
for (;;) {
    // 无限循环
}
```

请注意 `for` 的两个 `;` 必须存在，否则会出现语法错误。

跳出循环

通常条件为假时，循环会终止。

但我们随时都可以使用 `break` 指令强制退出。

例如，下面这个循环要求用户输入一系列数字，在输入的内容不是数字时“终止”循环。

```
let sum = 0;

while (true) {

    let value = +prompt("Enter a number", '');

    if (!value) break; // (*)

    sum += value;

}

alert( 'Sum: ' + sum );
```

如果用户输入空行或取消输入，在 `(*)` 行的 `break` 指令会被激活。它立刻终止循环，将控制权传递给循环后的第一行，即，`alert`。

根据需要，“无限循环 + `break`”的组合非常适用于不必在循环开始/结束时检查条件，但需要在中间甚至是主体的多个位置进行条件检查的情况。

继续下一次迭代

`continue` 指令是 `break` 的“轻量版”。它不会停掉整个循环。而是停止当前这一次迭代，并强制启动新一轮循环（如果条件允许的话）。

如果我们完成了当前的迭代，并且希望继续执行下一次迭代，我们就可以使用它。

下面这个循环使用 `continue` 来只输出奇数：

```
for (let i = 0; i < 10; i++) {  
    //如果为真，跳过循环体的剩余部分。  
    if (i % 2 == 0) continue;  
  
    alert(i); // 1, 然后 3, 5, 7, 9  
}
```

对于偶数的 `i` 值，`continue` 指令会停止本次循环的继续执行，将控制权传递给下一次 `for` 循环的迭代（使用下一个数字）。因此 `alert` 仅被奇数值调用。

💡 `continue` 指令利于减少嵌套

显示奇数的循环可以像下面这样：

```
for (let i = 0; i < 10; i++) {  
  
    if (i % 2) {  
        alert( i );  
    }  
  
}
```

从技术角度看，它与上一个示例完全相同。当然，我们可以将代码包装在 `if` 块而不使用 `continue`。

但在副作用方面，它多创建了一层嵌套（大括号内的 `alert` 调用）。如果 `if` 中代码有多行，则可能会降低代码整体的可读性。

⚠ 禁止 `break/continue` 在 '?' 的右边

请注意非表达式的语法规则不能与三元运算符 `?` 一起使用。特别是 `break/continue` 这样的指令是不允许这样使用的。

例如，我们使用如下代码：

```
if (i > 5) {  
    alert(i);  
} else {  
    continue;  
}
```

.....用问号重写：

```
(i > 5) ? alert(i) : continue; // continue 不允许在这个位置
```

.....代码会停止运行，并显示有语法错误。

这是不（建议）使用问号 `?` 运算符替代 `if` 语句的另一个原因。

`break/continue` 标签

有时候我们需要从一次从多层嵌套的循环中跳出来。

例如，下述代码中我们的循环使用了 `i` 和 `j`，从 `(0,0)` 到 `(3,3)` 提示坐标 `(i, j)`：

```
for (let i = 0; i < 3; i++) {  
  
    for (let j = 0; j < 3; j++) {  
  
        let input = prompt(`Value at coords (${i},${j})`);  
  
        // 如果我想从这里退出并直接执行 alert('Done!')  
    }  
}  
  
alert('Done!');
```

我们需要提供一种方法，以在用户取消输入时来停止这个过程。

在 `input` 之后的普通 `break` 只会打破内部循环。这还不够——标签可以实现这一功能！

标签 是在循环之前带有冒号的标识符：

```
labelName: for (...) {  
    ...  
}
```

`break <labelName>` 语句跳出循环至标签处：

```
outer: for (let i = 0; i < 3; i++) {  
  
    for (let j = 0; j < 3; j++) {  
  
        let input = prompt(`Value at coords (${i},${j})`);  
  
        // 如果是空字符串或被取消，则中断并跳出这两个循环。  
        if (!input) break outer; // (*)  
  
        // 用得到的值做些事.....  
    }  
}  
alert('Done!');
```

上述代码中，`break outer` 向上寻找名为 `outer` 的标签并跳出当前循环。

因此，控制权直接从 `(*)` 转至 `alert('Done!')`。

我们还可以将标签移至单独一行：

```
outer:  
for (let i = 0; i < 3; i++) { ... }
```

`continue` 指令也可以与标签一起使用。在这种情况下，执行跳转到标记循环的下一次迭代。

⚠ 标签并不允许“跳到”所有位置

标签不允许我们跳到代码的任意位置。

例如，这样做是不可能的：

```
break label; // 无法跳转到这个标签  
  
label: for (...)
```

只有在循环内部才能调用 `break/continue`，并且标签必须位于指令上方的某个位置。

总结

我们学习了三种循环：

- `while` —— 每次迭代之前都要检查条件。
- `do..while` —— 每次迭代后都要检查条件。
- `for (;;)` —— 每次迭代之前都要检查条件，可以使用其他设置。

通常使用 `while(true)` 来构造“无限”循环。这样的循环和其他循环一样，都可以通过 `break` 指令来终止。

如果我们不想在当前迭代中做任何事，并且想要转移至下一次迭代，那么可以使用 `continue` 指令。

`break/continue` 支持循环前的标签。标签是 `break/continue` 跳出嵌套循环以转到外部的唯一方法。

"switch" 语句

`switch` 语句可以替代多个 `if` 判断。

`switch` 语句为多分支选择的情况提供了一个更具描述性的方式。

语法

`switch` 语句有至少一个 `case` 代码块和一个可选的 `default` 代码块。

就像这样：

```
switch(x) {  
    case 'value1': // if (x === 'value1')  
        ...  
        [break]  
  
    case 'value2': // if (x === 'value2')  
        ...  
        [break]  
  
    default:  
        ...  
        [break]  
}
```

- 比较 `x` 值与第一个 `case`（也就是 `value1`）是否严格相等，然后比较第二个 `case`（`value2`）以此类推。
- 如果相等，`switch` 语句就执行相应 `case` 下的代码块，直到遇到最靠近的 `break` 语句（或者直到 `switch` 语句末尾）。
- 如果没有符合的 `case`，则执行 `default` 代码块（如果 `default` 存在）。

举个例子

`switch` 的例子（高亮的部分是执行的 `case` 部分）：

```
let a = 2 + 2;  
  
switch (a) {  
    case 3:  
        alert( 'Too small' );  
        break;  
    case 4:  
        alert( 'Exactly!' );  
        break;  
    case 5:  
        alert( 'Too large' );  
        break;  
    default:}
```

```
    alert( "I don't know such values" );
}
```

这里的 `switch` 从第一个 `case` 分支开始将 `a` 的值与 `case` 后的值进行比较，第一个 `case` 后的值为 `3` 匹配失败。

然后比较 `4`。匹配，所以从 `case 4` 开始执行直到遇到最近的 `break`。

如果没有 `break`，程序将不经过任何检查就会继续执行下一个 `case`。

无 `break` 的例子：

```
let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Too small' );
  case 4:
    alert( 'Exactly!' );
  case 5:
    alert( 'Too big' );
  default:
    alert( "I don't know such values" );
}
```

在上面的例子中我们会看到连续执行的三个 `alert`：

```
alert( 'Exactly!' );
alert( 'Too big' );
alert( "I don't know such values" );
```

① 任何表达式都可以成为 `switch/case` 的参数

`switch` 和 `case` 都允许任意表达式。

比如：

```
let a = "1";
let b = 0;

switch (+a) {
  case b + 1:
    alert("this runs, because +a is 1, exactly equals b+1");
    break;

  default:
    alert("this doesn't run");
}
```

这里 `+a` 返回 `1`，这个值跟 `case` 中 `b + 1` 相比较，然后执行对应的代码。

“case” 分组

共享同一段代码的几个 `case` 分支可以被分为一组：

比如，如果我们想让 `case 3` 和 `case 5` 执行同样的代码：

```
let a = 3;

switch (a) {
  case 4:
    alert('Right!');
    break;

  case 3: // (*) 下面这两个 case 被分在一组
  case 5:
    alert('Wrong!');
    alert("Why don't you take a math class?");
    break;

  default:
    alert('The result is strange. Really.');
}
```

现在 `3` 和 `5` 都显示相同的信息。

`switch/case` 有通过 `case` 进行“分组”的能力，其实是 `switch` 语句没有 `break` 时的副作用。因为没有 `break`，`case 3` 会从 `(*)` 行执行到 `case 5`。

类型很关键

强调一下，这里的相等是严格相等。被比较的值必须是相同的类型才能进行匹配。

比如，我们来看下面的代码：

```
let arg = prompt("Enter a value?")
switch (arg) {
  case '0':
  case '1':
    alert( 'One or zero' );
    break;

  case '2':
    alert( 'Two' );
    break;

  case 3:
    alert( 'Never executes!' );
    break;
  default:
    alert( 'An unknown value' )
}
```

1. 在 `prompt` 对话框输入 `0`、`1`，第一个 `alert` 弹出。
2. 输入 `2`，第二个 `alert` 弹出。

3. 但是输入 `3`，因为 `prompt` 的结果是字符串类型的 `"3"`，不严格相等 `==` 于数字类型的 `3`，所以 `case 3` 不会执行！因此 `case 3` 部分是一段无效代码。所以会执行 `default` 分支。

函数

我们经常需要在脚本的许多地方执行很相似的操作。

例如，当访问者登录、注销或者其他地方时，我们需要显示一条好看的信息。

函数是程序的主要“构建模块”。函数使该段代码可以被调用很多次，而不需要写重复的代码。

我们已经看到了内置函数的示例，如 `alert(message)`、`prompt(message, default)` 和 `confirm(question)`。但我们也创建自己的函数。

函数声明

使用 **函数声明** 创建函数。

看起来就像这样：

```
function showMessage() {  
    alert('Hello everyone!');  
}
```

`function` 关键字首先出现，然后是 **函数名**，然后是括号之间的 **参数** 列表（用逗号分隔，在上述示例中为空），最后是花括号之间的代码（即“**函数体**”）。

```
function name(parameters) {  
    ...body...  
}
```

我们的新函数可以通过名称调用：`showMessage()`。

例如：

```
function showMessage() {  
    alert('Hello everyone!');  
}
```

```
showMessage();  
showMessage();
```

调用 `showMessage()` 执行函数的代码。这里我们会看到显示两次消息。

这个例子清楚地演示了函数的主要目的之一：避免代码重复。

如果我们需要更改消息或其显示方式，只需在一个地方修改代码：输出它的函数。

局部变量

在函数中声明的变量只在该函数内部可见。

例如：

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // 局部变量

  alert( message );
}

showMessage(); // Hello, I'm JavaScript!

alert( message ); // <-- 错误！变量是函数的局部变量
```

外部变量

函数也可以访问外部变量，例如：

```
let userName = 'John';

function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}

showMessage(); // Hello, John
```

函数对外部变量拥有全部的访问权限。函数也可以修改外部变量。

例如：

```
let userName = 'John';

function showMessage() {
  userName = "Bob"; // (1) 改变外部变量

  let message = 'Hello, ' + userName;
  alert(message);
}

alert( userName ); // John 在函数调用之前

showMessage();

alert( userName ); // Bob, 值被函数修改了
```

只有在没有局部变量的情况下才会使用外部变量。

如果在函数内部声明了同名变量，那么函数会 **遮蔽** 外部变量。例如，在下面的代码中，函数使用局部的 `userName`，而外部变量被忽略：

```
let userName = 'John';

function showMessage() {
  let userName = "Bob"; // 声明一个局部变量

  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

// 函数会创建并使用它自己的 userName
showMessage();

alert( userName ); // John, 未被更改, 函数没有访问外部变量。
```

① 全局变量

任何函数之外声明的变量，例如上述代码中的外部变量 `userName`，都被称为 **全局** 变量。

全局变量在任意函数中都是可见的（除非被局部变量遮蔽）。

减少全局变量的使用是一种很好的做法。现代的代码有很少甚至没有全局变量。大多数变量存在于它们的函数中。但是有时候，全局变量能够用于存储项目级别的数据。

参数

我们可以使用参数（也称“函数参数”）来将任意数据传递给函数。

在如下示例中，函数有两个参数：`from` 和 `text`。

```
function showMessage(from, text) { // 参数: from 和 text
  alert(from + ': ' + text);
}

showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

当函数在 `(*)` 和 `(**)` 行中被调用时，给定值被复制到了局部变量 `from` 和 `text`。然后函数使用它们进行计算。

这里还有一个例子：我们有一个变量 `from`，并将它传递给函数。请注意：函数会修改 `from`，但在函数外部看不到更改，因为函数修改的是复制的变量值副本：

```
function showMessage(from, text) {

  from = '*' + from + '*'; // 让 "from" 看起来更优雅

  alert( from + ': ' + text );
}

let from = "Ann";

showMessage(from, "Hello"); // *Ann*: Hello
```

```
// "from" 值相同，函数修改了一个局部的副本。  
alert( from ); // Ann
```

默认值

如果未提供参数，那么其默认值则是 `undefined`。

例如，之前提到的函数 `showMessage(from, text)` 可以只使用一个参数调用：

```
showMessage("Ann");
```

那不是错误，这样调用将输出 `"Ann: undefined"`。这里没有参数 `text`，所以程序假定 `text === undefined`。

如果我们想在本示例中设定“默认”的 `text`，那么我们可以在 `=` 之后指定它：

```
function showMessage(from, text = "no text given") {  
  alert( from + ": " + text );  
}  
  
showMessage("Ann"); // Ann: no text given
```

现在如果 `text` 参数未被传递，它将会得到值 `"no text given"`。

这里 `"no text given"` 是一个字符串，但它可以是更复杂的表达式，并且只会在缺少参数时才会被计算和分配。所以，这也是可能的：

```
function showMessage(from, text = anotherFunction()) {  
  // anotherFunction() 仅在没有给定 text 时执行  
  // 其运行结果将成为 text 的值  
}
```

i 默认参数的计算

在 JavaScript 中，每次函数在没带个别参数的情况下被调用，默认参数会被计算出来。

在上面的例子中，每次 `showMessage()` 不带 `text` 参数被调用时，`anotherFunction()` 就会被调用。

i 旧式默认参数

旧版本的 JavaScript 不支持默认参数。所以在大多数旧版本的脚本中，你能找到其他设置默认参数的方法。

例如，用于 `undefined` 的显式检查：

```
function showMessage(from, text) {
    if (text === undefined) {
        text = 'no text given';
    }

    alert( from + ": " + text );
}
```

.....或使用 `||` 运算符：

```
function showMessage(from, text) {
    // 如果 text 能转为 false, 那么 text 会得到“默认”值
    text = text || 'no text given';
    ...
}
```

返回值

函数可以将一个值返回到调用代码中作为结果。

最简单的例子是将两个值相加的函数：

```
function sum(a, b) {
    return a + b;
}

let result = sum(1, 2);
alert( result ); // 3
```

指令 `return` 可以在函数的任意位置。当执行到达时，函数停止，并将值返回给调用代码（分配给上述代码中的 `result`）。

在一个函数中可能会出现很多次 `return`。例如：

```
function checkAge(age) {
    if (age >= 18) {
        return true;
    } else {
        return confirm('Got a permission from the parents?');
    }
}

let age = prompt('How old are you?', 18);
```

```
if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
  alert( 'Access denied' );
}
```

只使用 `return` 但没有返回值也是可行的。但这会导致函数立即退出。

例如：

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Showing you the movie" ); // (*)
  // ...
}
```

在上述代码中，如果 `checkAge(age)` 返回 `false`，那么 `showMovie` 将不会运行到 `alert`。

❶ 空值的 `return` 或没有 `return` 的函数返回值为 `undefined`

如果函数无返回值，它就会像返回 `undefined` 一样：

```
function doNothing() { /* 没有代码 */ }

alert( doNothing() === undefined ); // true
```

空值的 `return` 和 `return undefined` 等效：

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

⚠ 不要在 `return` 与返回值之间添加新行

对于 `return` 的长表达式，可能你会很想将其放在单独一行，如下所示：

```
return  
(some + long + expression + or + whatever * f(a) + f(b))
```

但这不行，因为 JavaScript 默认会在 `return` 之后加上分号。上面这段代码和下面这段代码运行流程相同：

```
return;  
(some + long + expression + or + whatever * f(a) + f(b))
```

因此，实际上它的返回值变成了空值。

如果我们想要将返回的表达式写成跨多行的形式，那么应该在 `return` 的同一行开始写此表达式。或者至少按照如下的方式放上左括号：

```
return (  
  some + long + expression  
  + or +  
  whatever * f(a) + f(b)  
)
```

然后它就能像我们预想的那样正常运行了。

函数命名

函数就是行为（action）。所以它们的名字通常是动词。它应该简短且尽可能准确地描述函数的作用。这样读代码的人就能清楚地知道这个函数的功能。

一种普遍的做法是用动词前缀来开始一个函数，这个前缀模糊地描述了这个行为。团队内部必须就前缀的含义达成一致。

例如，以 "show" 开头的函数通常会显示某些内容。

函数以 XX 开始……

- "get..." —— 返回一个值，
- "calc..." —— 计算某些内容，
- "create..." —— 创建某些内容，
- "check..." —— 检查某些内容并返回 boolean 值，等。

这类名字的示例：

```
showMessage(..)      // 显示信息  
getAge(..)          // 返回 age (gets it somehow)  
calcSum(..)         // 计算求和并返回结果
```

```
createForm(..)      // 创建表格（通常会返回它）
checkPermission(..) // 检查权限并返回 true/false
```

有了前缀，只需瞥一眼函数名，就可以了解它的功能是什么，返回什么样的值。

① 一个函数 —— 一个行为

一个函数应该只包含函数名所指定的功能，而不是做更多与函数名无关的功能。

两个独立的行为通常需要两个函数，即使它们通常被一起调用（在这种情况下，我们可以创建第三个函数来调用这两个函数）。

有几个违反这一规则的例子：

- `getAge` —— 如果它通过 `alert` 将 `age` 显示出来，那就有问题了（只应该是获取）。
- `createForm` —— 如果它包含修改文档的操作，例如向文档添加一个表单，那就有问题了（只应该创建表单并返回）。
- `checkPermission` —— 如果它显示 `access granted/denied` 消息，那就有问题了（只应执行检查并返回结果）。

这些例子假设函数名前缀具有通用的含义。你和你的团队可以自定义这些函数名前缀的含义，但是通常都没有太大的不同。无论怎样，你都应该对函数名前缀的含义、带特定前缀的函数可以做什么以及不可以做什么有深刻的理解。所有相同前缀的函数都应该遵守相同的规则。并且，团队成员应该形成共识。

① 非常短的函数命名

常用的函数有时会有**非常短**的名字。

例如，`jQuery` 框架用 `$` 定义一个函数。`LoDash` 库的核心函数用 `_` 命名。

这些都是例外，一般而言，函数名应简明扼要且具有描述性。

函数 == 注释

函数应该简短且只有一个功能。如果这个函数功能复杂，那么把该函数拆分成几个小的函数是值得的。有时候遵循这个规则并不是那么容易，但这绝对是件好事。

一个单独的函数不仅更容易测试和调试 —— 它的存在本身就是一个很好的注释！

例如，比较如下两个函数 `showPrimes(n)`。他们的功能都是输出到 `n` 的 素数。

第一个变体使用了一个标签：

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert( i ); // 一个素数
  }
}
```

第二个变体使用附加函数 `isPrime(n)` 来检验素数：

```
function showPrimes(n) {  
  for (let i = 2; i < n; i++) {  
    if (!isPrime(i)) continue;  
  
    alert(i); // 一个素数  
  }  
}  
  
function isPrime(n) {  
  for (let i = 2; i < n; i++) {  
    if (n % i == 0) return false;  
  }  
  return true;  
}
```

第二个变体更容易理解，不是吗？我们通过函数名（`isPrime`）就可以看出函数的行为，而不需要通过代码。人们通常把这样的代码称为 **自描述**。

因此，即使我们不打算重用它们，也可以创建函数。函数可以让代码结构更清晰，可读性更强。

总结

函数声明方式如下所示：

```
function name(parameters, delimited, by, comma) {  
  /* code */  
}
```

- 作为参数传递给函数的值，会被复制到函数的局部变量。
- 函数可以访问外部变量。但它只能从内到外起作用。函数外部的代码看不到函数内的局部变量。
- 函数可以返回值。如果没有返回值，则其返回的结果是 `undefined`。

为了使代码简洁易懂，建议在函数中主要使用局部变量和参数，而不是外部变量。

与不获取参数但将修改外部变量作为副作用的函数相比，获取参数、使用参数并返回结果的函数更容易理解。

函数命名：

- 函数名应该清楚地描述函数的功能。当我们在代码中看到一个函数调用时，一个好的函数名能够让我们马上知道这个函数的功能是什么，会返回什么。
- 一个函数是一个行为，所以函数名通常是动词。
- 目前有许多优秀的函数名前缀，如 `create...`、`show...`、`get...`、`check...` 等等。使用它们来提示函数的作用吧。

函数是脚本的主要构建块。现在我们已经介绍了基本知识，现在我们就可以开始创建和使用函数了。但这只是学习和使用函数的开始。我们将继续学习更多函数的相关知识，更深入地研究它们的先进特征。

函数表达式

在 JavaScript 中，函数不是“神奇的语言结构”，而是一种特殊的值。

我们在前面章节使用的语法称为 **函数声明**：

```
function sayHi() {  
  alert( "Hello" );  
}
```

另一种创建函数的语法称为 **函数表达式**。

通常会写成这样：

```
let sayHi = function() {  
  alert( "Hello" );  
};
```

在这里，函数被创建并像其他赋值一样，被明确地分配给了一个变量。不管函数是被怎样定义的，都只是一个存储在变量 `sayHi` 中的值。

上面这两段示例代码的意思是一样的：“创建一个函数，并把它存进变量 `sayHi`”。

我们还可以用 `alert` 打印这个变量值：

```
function sayHi() {  
  alert( "Hello" );  
}  
  
alert( sayHi ); // 显示函数代码
```

注意，最后一行代码并不会运行函数，因为 `sayHi` 后没有括号。在其他编程语言中，只要提到函数的名称都会导致函数的调用执行，但 JavaScript 可不是这样。

在 JavaScript 中，函数是一个值，所以我们可以把它当成值对待。上面代码显示了一段字符串值，即函数的源码。

的确，在某种意义上说一个函数是一个特殊值，我们可以像 `sayHi()` 这样调用它。

但它依然是一个值，所以我们可以像使用其他类型的值一样使用它。

我们可以复制函数到其他变量：

```
function sayHi() { // (1) 创建  
  alert( "Hello" );  
}  
  
let func = sayHi; // (2) 复制
```

```
func(); // Hello      // (3) 运行复制的值（正常运行）！  
sayHi(); // Hello     //       这里也能运行（为什么不行呢）
```

解释一下上段代码发生的细节：

1. (1) 行声明创建了函数，并把它放入到变量 `sayHi`。
2. (2) 行将 `sayHi` 复制到了变量 `func`。请注意：`sayHi` 后面没有括号。如果有括号，`func = sayHi()` 会把 `sayHi()` 的调用结果写进 `func`，而不是 `sayHi` 函数本身。
3. 现在函数可以通过 `sayHi()` 和 `func()` 两种方式进行调用。

注意，我们也可以在第一行中使用函数表达式来声明 `sayHi`：

```
let sayHi = function() {  
  alert("Hello");  
};  
  
let func = sayHi;  
// ...
```

这两种声明的函数是一样的。

💡 为什么这里末尾会有个分号？

你可能想知道，为什么函数表达式结尾有一个分号 `;`，而函数声明没有：

```
function sayHi() {  
  // ...  
}  
  
let sayHi = function() {  
  // ...  
};
```

答案很简单：

- 在代码块的结尾不需要加分号 `;`，像 `if { ... }`, `for { }`, `function f { }` 等语法结构后面都不用加。
- 函数表达式是在语句内部的：`let sayHi = ...;`，作为一个值。它不是代码块而是一个赋值语句。不管值是什么，都建议在语句末尾添加分号 `;`。所以这里的分号与函数表达式本身没有任何关系，它只是用于终止语句。

回调函数

让我们多举几个例子，看看如何将函数作为值来传递以及如何使用函数表达式。

我们写一个包含三个参数的函数 `ask(question, yes, no)`：

`question`

关于问题的文本

yes

当回答为 “Yes” 时，要运行的脚本

no

当回答为 “No” 时，要运行的脚本

函数需要提出 `question` (问题)，并根据用户的回答，调用 `yes()` 或 `no()`：

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert("You agreed.");
}

function showCancel() {
  alert("You canceled the execution.");
}

// 用法：函数 showOk 和 showCancel 被作为参数传入到 ask
ask("Do you agree?", showOk, showCancel);
```

在实际开发中，这样的的函数是非常有用的。实际开发与上述示例最大的区别是，实际开发中的函数会通过更加复杂的方式与用户进行交互，而不是通过简单的 `confirm`。在浏览器中，这样的函数通常会绘制一个漂亮的提问窗口。但这是另外一件事了。

`ask` 的两个参数值 `showOk` 和 `showCancel` 可以被称为 **回调函数** 或简称 **回调**。

主要思想是我们传递一个函数，并期望在稍后必要时将其“回调”。在我们的例子中，`showOk` 是回答 “yes” 的回调，`showCancel` 是回答 “no” 的回调。

我们可以用函数表达式对同样的函数进行大幅简写：

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

这里直接在 `ask(...)` 调用内进行函数声明。这两个函数没有名字，所以叫 **匿名函数**。这样的函数在 `ask` 外无法访问（因为没有对它们分配变量），不过这正是我们想要的。

这样的代码在我们的脚本中非常常见，这正符合 JavaScript 语言的思想。

① 一个函数是表示一个“行为”的值

字符串或数字等常规值代表 **数据**。

函数可以被视为一个 **行为 (action)**。

我们可以在变量之间传递它们，并在需要时运行。

函数表达式 vs 函数声明

让我们来总结一下函数声明和函数表达式之间的主要区别。

首先是语法：如何通过代码对它们进行区分。

- **函数声明：**在主代码流中声明为单独的语句的函数。

```
// 函数声明
function sum(a, b) {
  return a + b;
}
```

- **函数表达式：**在一个表达式中或另一个语法结构中创建的函数。下面这个函数是在赋值表达式 = 右侧创建的：

```
// 函数表达式
let sum = function(a, b) {
  return a + b;
};
```

更细微的差别是，JavaScript 引擎会在 **什么时候** 创建函数。

函数表达式是在代码执行到达时被创建，并且仅从那一刻起可用。

一旦代码执行到赋值表达式 `let sum = function...` 的右侧，此时就会开始创建该函数，并且可以从现在开始使用（分配，调用等）。

函数声明则不同。

在函数声明被定义之前，它就可以被调用。

例如，一个全局函数声明对整个脚本来说都是可见的，无论它被写在这个脚本的哪个位置。

这是内部算法的原故。当 **JavaScript 准备** 运行脚本时，首先会在脚本中寻找全局函数声明，并创建这些函数。我们可以将其视为“**初始化阶段**”。

在处理完所有函数声明后，代码才被执行。所以运行时能够使用这些函数。

例如下面的代码会正常工作：

```
sayHi("John"); // Hello, John

function sayHi(name) {
  alert(`Hello, ${name}`);
}
```

函数声明 `sayHi` 是在 JavaScript 准备运行脚本时被创建的，在这个脚本的任何位置都可见。

.....如果它是一个函数表达式，它就不会工作：

```
sayHi("John"); // error!

let sayHi = function(name) { // (*) no magic any more
  alert(`Hello, ${name}`);
};
```

函数表达式在代码执行到它时才会被创建。只会发生在 `(*)` 行。为时已晚。

函数声明的另外一个特殊的功能是它们的块级作用域。

严格模式下，当一个函数声明在一个代码块内时，它在该代码块内的任何位置都是可见的。但在代码块外不可见。

例如，想象一下我们需要依赖于在代码运行过程中获得的变量 `age` 声明一个函数 `welcome()`。并且我们计划在之后的某个时间使用它。

如果我们使用函数声明，以下则代码不能如愿工作：

```
let age = prompt("what is your age?", 18);

// 有条件地声明一个函数
if (age < 18) {

  function welcome() {
    alert("Hello!");
  }

} else {

  function welcome() {
    alert("Greetings!");
  }

}

// .....稍后使用
welcome(); // Error: welcome is not defined
```

这是因为函数声明只在它所在的代码块中可见。

下面是另一个例子：

```
let age = 16; // 拿 16 作为例子

if (age < 18) {
  welcome(); // \ (运行)
  // |
  function welcome() { // |
    alert("Hello!"); // | 函数声明在声明它的代码块内任意位置都可用
  } // |
  // |
```

```
welcome(); // / (运行)

} else {

    function welcome() {
        alert("Greetings!");
    }
}

// 在这里，我们在花括号外部调用函数，我们看不到它们内部的函数声明。
```

```
welcome(); // Error: welcome is not defined
```

我们怎么才能让 `welcome` 在 `if` 外可见呢？

正确的做法是使用函数表达式，并将 `welcome` 赋值给在 `if` 外声明的变量，并具有正确的可见性。

下面的代码可以如愿运行：

```
let age = prompt("What is your age?", 18);

let welcome;

if (age < 18) {

    welcome = function() {
        alert("Hello!");
    };
}

} else {

    welcome = function() {
        alert("Greetings!");
    };
}

welcome(); // 现在可以了
```

或者我们可以使用问号运算符 `?` 来进一步对代码进行简化：

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
    function() { alert("Hello!"); } :
    function() { alert("Greetings!"); };

welcome(); // 现在可以了
```

① 什么时候选择函数声明与函数表达式?

根据经验，当我们需要声明一个函数时，首先考虑函数声明语法。它能够为组织代码提供更多灵活性。因为我们在声明这些函数之前调用这些函数。

这对代码可读性也更好，因为在代码中查找 `function f(...){...}` 比 `let f = function(...){...}` 更容易。函数声明更“醒目”。

……但是，如果由于某种原因而导致函数声明不适合我们（我们刚刚看过上面的例子），那么应该使用函数表达式。

总结

- 函数是值。它们可以在代码的任何地方被分配，复制或声明。
- 如果函数在主代码流中被声明为单独的语句，则称为“函数声明”。
- 如果该函数是作为表达式的一部分创建的，则称其“函数表达式”。
- 在执行代码块之前，内部算法会先处理函数声明。所以函数声明在其被声明的代码块内的任何位置都是可见的。
- 函数表达式在执行流程到达时创建。

在大多数情况下，当我们需要声明一个函数时，最好使用函数声明，因为函数在被声明之前也是可见的。这使我们在代码组织方面更具灵活性，通常也会使得代码可读性更高。

所以，仅当函数声明不适合对应的任务时，才应使用函数表达式。在本章中，我们已经看到了几个例子，以后还会看到更多的例子。

箭头函数，基础知识

创建函数还有另外一种非常简单的语法，并且这种方法通常比函数表达式更好。

它被称为“箭头函数”，因为它看起来像这样：

```
let func = (arg1, arg2, ...argN) => expression
```

……这里创建了一个函数 `func`，它接受参数 `arg1..argN`，然后使用参数对右侧的 `expression` 求值并返回其结果。

换句话说，它是下面这段代码的更短的版本：

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

让我们来看一个具体的例子：

```
let sum = (a, b) => a + b;

/* 这个箭头函数是下面这个函数的更短的版本：
```

```
let sum = function(a, b) {
  return a + b;
};

alert( sum(1, 2) ); // 3
```

可以看到 `(a, b) => a + b` 表示一个函数接受两个名为 `a` 和 `b` 的参数。在执行时，它将对表达式 `a + b` 求值，并返回计算结果。

- 如果我们只有一个参数，还可以省略掉参数外的圆括号，使代码更短。

例如：

```
let double = n => n * 2;
// 差不多等同于: let double = function(n) { return n * 2 }

alert( double(3) ); // 6
```

- 如果没有参数，括号将是空的（但括号应该保留）：

```
let sayHi = () => alert("Hello!");

sayHi();
```

箭头函数可以像函数表达式一样使用。

例如，动态创建一个函数：

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  () => alert('Hello') :
  () => alert("Greetings!");

welcome();
```

一开始，箭头函数可能看起来并不熟悉，也不容易读懂，但一旦我们看习惯了之后，这种情况很快就会改变。

箭头函数对于简单的单行行为（action）来说非常方便，尤其是当我们懒得打太多字的时候。

多行的箭头函数

上面的例子从 `=>` 的左侧获取参数，然后使用参数计算右侧表达式的值。

但有时我们需要更复杂一点的东西，比如多行的表达式或语句。这也是可以做到的，但是我们应该用花括号括起来。然后使用一个普通的 `return` 将需要返回的值进行返回。

就像这样：

```
let sum = (a, b) => { // 花括号表示开始一个多行函数
  let result = a + b;
  return result; // 如果我们使用了花括号，那么我们需要一个显式的“return”
};

alert( sum(1, 2) ); // 3
```

1 更多内容

在这里，我们赞扬了箭头功能的简洁性。但还不止这些！

箭头函数还有其他有趣的特性。

为了更深入地学习它们，我们首先需要了解一些 JavaScript 其他方面的知识，因此我们将在后面的 [深入理解箭头函数](#) 一章中再继续研究箭头函数。

现在，我们已经可以用箭头函数进行单行行为和回调了。

总结

对于一行代码的函数来说，箭头函数是相当方便的。它具体有两种：

1. 不带花括号：`(...args) => expression` — 右侧是一个表达式：函数计算表达式并返回其结果。
2. 带花括号：`(...args) => { body }` — 花括号允许我们在函数中编写多个语句，但是我们需要显式地 `return` 来返回一些内容。

JavaScript 特性

本章简要回顾我们到现在为止学到的 JavaScript 特性，并特别注意了一些细节。

代码结构

语句用分号分隔：

```
alert('Hello'); alert('World');
```

通常，换行符也被视为分隔符，因此下面的例子也能正常运行：

```
alert('Hello')
alert('World')
```

这就是所谓的「自动分号插入」。但有时它不起作用，例如：

```
alert("There will be an error after this message")

[1, 2].forEach(alert)
```

大多数代码风格指南都认为我们应该在每个语句后面都加上分号。

在代码块 `{...}` 后以及有代码块的语法结构（例如循环）后不需要加分号：

```
function f() {  
    // 函数声明后不需要加分号  
}  
  
for(;;) {  
    // 循环语句后不需要加分号  
}
```

.....但即使我们在某处添加了「额外的」分号，这也不是错误。分号会被忽略的。

更多内容：[代码结构](#)。

严格模式

为了完全启用现代 JavaScript 的所有特性，我们应该在脚本顶部写上 `"use strict"` 指令。

```
'use strict';  
  
...
```

该指令必须位于 JavaScript 脚本的顶部或函数体的开头。

如果没有 `"use strict"`，所有东西仍可以正常工作，但是某些特性的表现方式与旧式「兼容」方式相同。我们通常更喜欢现代的方式。

语言的一些现代特征（比如我们将来要学习的类）会隐式地启用严格模式。

更多内容：[现代模式, "use strict"](#)。

变量

可以使用以下方式声明变量：

- `let`
- `const`（不变的，不能被改变）
- `var`（旧式的，稍后会看到）

一个变量名可以由以下组成：

- 字母和数字，但是第一个字符不能是数字。
- 字符 `$` 和 `_` 是允许的，用法同字母。
- 非拉丁字母和象形文字也是允许的，但通常不会使用。

变量是动态类型的，它们可以存储任何值：

```
let x = 5;  
x = "John";
```

有 8 种数据类型：

- `number` — 可以是浮点数，也可以是整数，
- `bigint` — 用于任意长度的整数，
- `string` — 字符串类型，
- `boolean` — 逻辑值：`true/false`，
- `null` — 具有单个值 `null` 的类型，表示“空”或“不存在”，
- `undefined` — 具有单个值 `undefined` 的类型，表示“未分配（未定义）”，
- `object` 和 `symbol` — 对于复杂的数据结构和唯一标识符，我们目前还没学习这个类型。

`typeof` 运算符返回值的类型，但有两个例外：

```
typeof null == "object" // JavaScript 编程语言的设计错误
typeof function(){} == "function" // 函数被特殊对待
```

更多内容：[变量](#) 和 [数据类型](#)。

交互

我们使用浏览器作为工作环境，所以基本的 UI 功能将是：

[**prompt\(question\[, default\]\)**](#)

提出一个问题，并返回访问者输入的内容，如果他按下「取消」则返回 `null`。

[**confirm\(question\)**](#)

提出一个问题，并建议用户在“确定”和“取消”之间进行选择。选择结果以 `true/false` 形式返回。

[**alert\(message\)**](#)

输出一个 `消息`。

这些函数都会产生 **模态框**，它们会暂停代码执行并阻止访问者与页面的其他部分进行交互，直到用户做出回答为止。

举个例子：

```
let userName = prompt("Your name?", "Alice");
let isTeaWanted = confirm("Do you want some tea?");

alert( "Visitor: " + userName ); // Alice
alert( "Tea wanted: " + isTeaWanted ); // true
```

更多内容：[交互：alert、prompt 和 confirm](#)。

运算符

JavaScript 支持以下运算符:

算数运算符

常规的: `+ - * /` (加减乘除) , 取余运算符 `%` 和幂运算符 `**`。

二进制加号 `+` 可以连接字符串。如果任何一个操作数是一个字符串, 那么另一个操作数也将被转换为字符串:

```
alert( '1' + 2 ); // '12', 字符串
alert( 1 + '2' ); // '12', 字符串
```

赋值

简单的赋值: `a = b` 和合并了其他操作的赋值: `a *= 2`。

按位运算符

按位运算符在最低位级上操作 32 位的整数: 详见 [文档](#)。

三元运算符

唯一具有三个参数的操作: `cond ? resultA : resultB`。如果 `cond` 是真的, 则返回 `resultA`, 否则返回 `resultB`。

逻辑运算符

逻辑与 `&&` 和或 `||` 执行短路运算, 然后返回运算停止处的值 (`true/false` 不是必须的)。逻辑非 `!` 将操作数转换为布尔值并返回其相反的值。

比较运算符

对不同类型的值进行相等检查时, 运算符 `==` 会将不同类型的值转换为数字 (除了 `null` 和 `undefined`, 它们彼此相等而没有其他情况), 所以下面的例子是相等的:

```
alert( 0 == false ); // true
alert( 0 == '' ); // true
```

其他比较也将转换为数字。

严格相等运算符 `===` 不会进行转换: 不同的类型总是指不同的值。

值 `null` 和 `undefined` 是特殊的: 它们只在 `==` 下相等, 且不相等于其他任何值。

大于/小于比较, 在比较字符串时, 会按照字符顺序逐个字符地进行比较。其他类型则被转换为数字。

其他运算符

还有很少一部分其他运算符, 如逗号运算符。

更多内容: [运算符](#), [值的比较](#), [逻辑运算符](#)。

循环

- 我们涵盖了 3 种类型的循环：

```
// 1
while (condition) {
  ...
}

// 2
do {
  ...
} while (condition);

// 3
for(let i = 0; i < 10; i++) {
  ...
}
```

- 在 `for(let...)` 循环内部声明的变量，只在该循环内可见。但我们也省略 `let` 并重用已有的变量。

- 指令 `break/continue` 允许退出整个循环/当前迭代。使用标签来打破嵌套循环。

更多内容：[循环: while 和 for](#)。

稍后我们将学习更多类型的循环来处理对象。

“switch” 结构

“switch” 结构可以替代多个 `if` 检查。它内部使用 `==` (严格相等) 进行比较。

例如：

```
let age = prompt('Your age?', 18);

switch (age) {
  case 18:
    alert("Won't work"); // prompt 的结果是一个字符串，而不是数字

  case "18":
    alert("This works!");
    break;

  default:
    alert("Any value not equal to one above");
}
```

详情请见：["switch" 语句](#)。

函数

我们介绍了三种在 JavaScript 中创建函数的方式：

1. 函数声明：主代码流中的函数

```
function sum(a, b) {
  let result = a + b;

  return result;
}
```

2. 函数表达式: 表达式上下文中的函数

```
let sum = function(a, b) {
  let result = a + b;

  return result;
}
```

3. 箭头函数:

```
// 表达式在右侧
let sum = (a, b) => a + b;

// 或带 {...} 的多行语法, 此处需要 return:
let sum = (a, b) => {
  // ...
  return a + b;
}

// 没有参数
let sayHi = () => alert("Hello");

// 有一个参数
let double = n => n * 2;
```

- 函数可能具有局部变量: 在函数内部声明的变量。这类变量只在函数内部可见。
- 参数可以有默认值: `function sum(a = 1, b = 2) {...}`。
- 函数总是返回一些东西。如果没有 `return` 语句, 那么返回的结果是 `undefined`。

详细内容: 请见 [函数, 箭头函数, 基础知识](#)。

更多内容

这些是 JavaScript 特性的简要概述。截至目前, 我们仅仅学习了基础知识。随着教程的深入, 你会发现 JavaScript 的更多特性和高级特性。

代码质量

本章介绍了我们将在开发中进一步使用的编码实践。

在 Chrome 中调试

在编写更复杂的代码前, 让我们先来聊聊调试吧。

调试 是指在一个脚本中找出并修复错误的过程。所有的现代浏览器和大多数其他环境都支持调试工具——开发者工具中的一个令调试更加容易的特殊用户界面。它也可以让我们一步步地跟踪代码以查看当前实际运行情况。

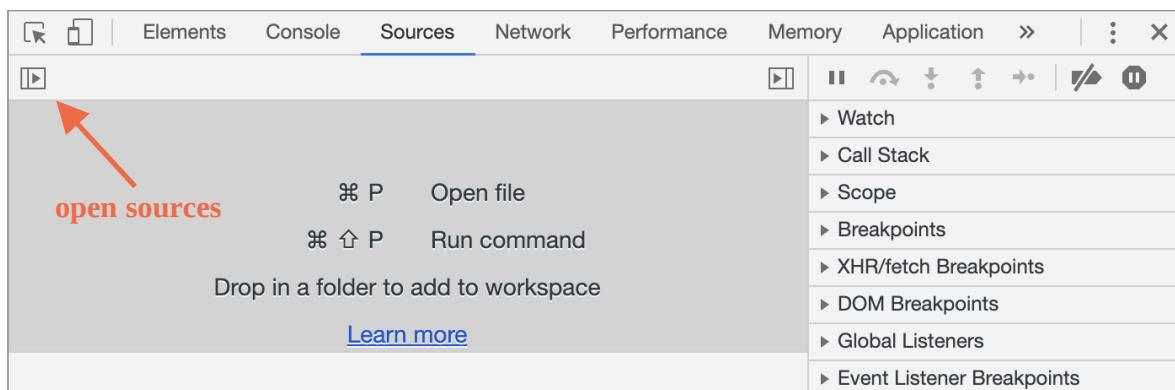
在这里我们将会使用 **Chrome**（谷歌浏览器），因为它拥有足够多的功能，其他大部分浏览器的功能也与之类似。

“资源 (Sources) ”面板

你的 Chrome 版本可能看起来有一点不同，但是它应该还是处于很明显的位置。

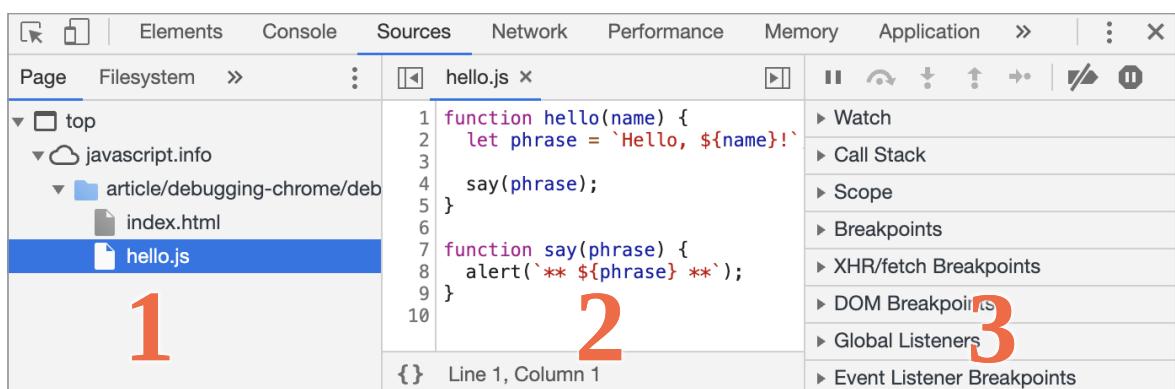
- 在 Chrome 中打开 [示例页面](#)。
- 使用快捷键 **F12** (Mac: **Cmd+Opt+I**) 打开开发者工具。
- 选择 **Sources** (资源) 面板。

如果你是第一次这么做，那你应该会看到下面这个样子：



切换按钮 会打开文件列表的选项卡。

让我们在预览树中点击和选择 `hello.js`。这里应该会如下图所示：



资源 (Sources) 面板包含三个部分：

- 文件导航 (File Navigator)** 区域列出了 HTML、JavaScript、CSS 和包括图片在内的其他依附于此页面的文件。Chrome 扩展程序也会显示在这。
- 代码编辑 (Code Editor)** 区域展示源码。
- JavaScript 调试 (JavaScript Debugging)** 区域是用于调试的，我们很快就会来探索它。

现在你可以再次点击切换按钮 隐藏资源列表来给代码腾出一些空间。

控制台 (Console)

如果我们按下 **Esc**，下面会出现一个控制台，我们可以输入一些命令然后按下 **Enter** 来执行。

语句执行完毕之后，其执行结果会显示在下面。

例如，`1+2` 将会返回 `3`，`hello("debugger")` 函数什么也不返回，因此结果是 `undefined`：



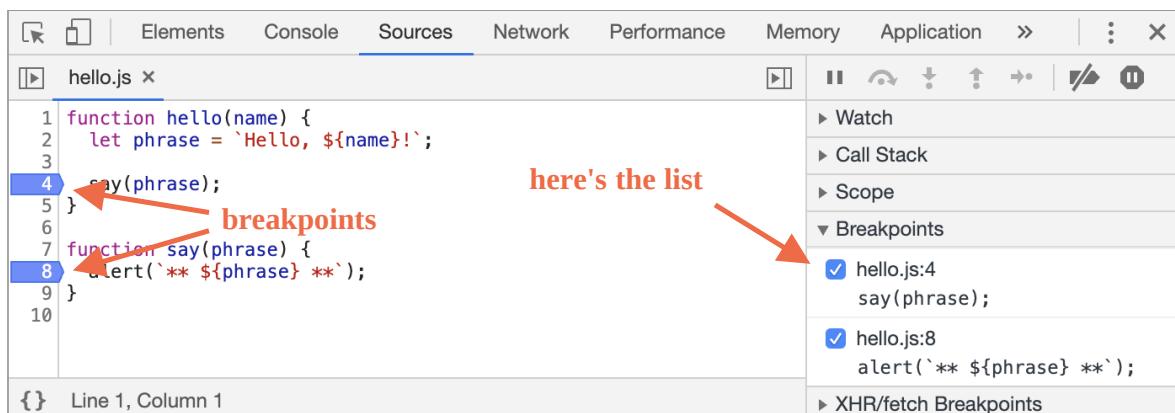
```
Console
Default levels ▾
▶ 1 + 2
< 3
▶ hello("debugger")
< undefined
> |
```

断点 (Breakpoints)

我们来看看 [示例页面](#) 发生了什么。在 `hello.js` 中，点击第 **4** 行。是的，就点击数字 `"4"` 上，不是点击代码。

恭喜你！你已经设置了一个断点。现在，请在第 **8** 行的数字上也点击一下。

看起来应该是这样的（蓝色是你应该点击的地方）：



```
hello.js
1 function hello(name) {
2   let phrase = `Hello, ${name}!`;
3
4   say(phrase);
5 }
6
7 function say(phrase) {
8   alert(`** ${phrase} **`);
9 }
10

{} Line 1, Column 1
```

here's the list

- ▶ Watch
- ▶ Call Stack
- ▶ Scope
- ▼ Breakpoints
 - hello.js:4
say(phrase);
 - hello.js:8
alert(`** \${phrase} **`);
 - ▶ XHR/fetch Breakpoints

断点 是调试器会自动暂停 JavaScript 执行的地方。

当代码被暂停时，我们可以检查当前的变量，在控制台执行命令等等。换句话说，我们可以调试它。

我们总是可以在右侧的面板中找到断点的列表。当我们在数个文件中有许多断点时，这是非常有用的。它允许我们：

- 快速跳转至代码中的断点（通过点击右侧面板中的对应的断点）。
- 通过取消选中断点来临时禁用对应的断点。
- 通过右键单击并选择移除来删除一个断点。
-等等。

ⓘ 条件断点

在行号上 右键单击 允许你创建一个 条件 断点。只有当给定的表达式为真（即满足条件）时才会被触发。

当我们需要在特定的变量值或参数的情况下暂停程序执行时，这种调试方法就很有用了。

Debugger 命令

我们也可以使用 `debugger` 命令来暂停代码，像这样：

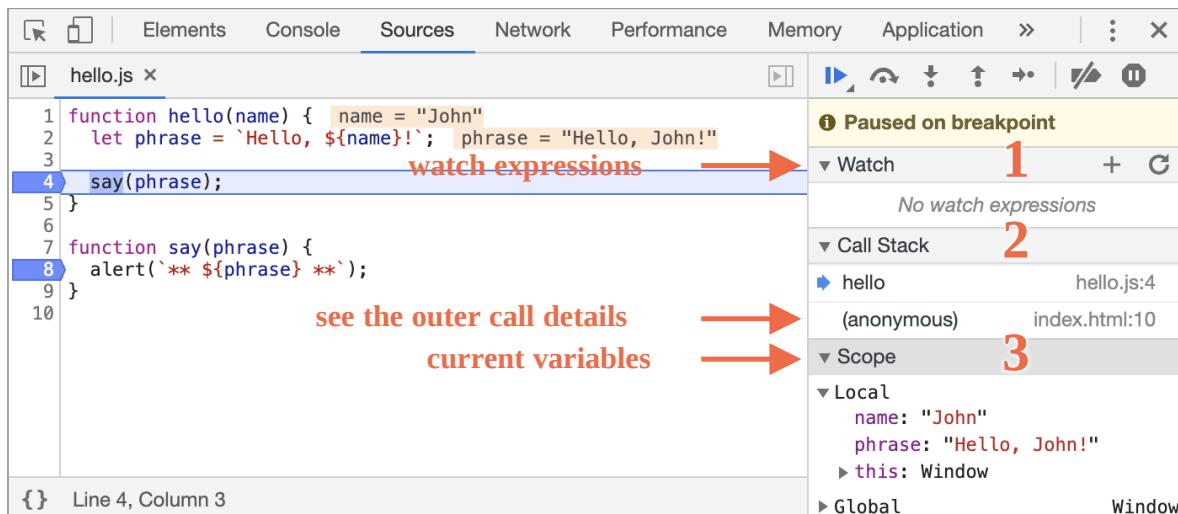
```
function hello(name) {  
  let phrase = `Hello, ${name}!`;  
  
  debugger; // <-- 调试器会在这停止  
  
  say(phrase);  
}
```

当我们在一个代码编辑器中并且不想切换到浏览器在开发者工具中查找脚本来设置断点时，这真的非常方便。

暂停并查看

在我们的例子中，`hello()` 函数在页面加载期间被调用，因此激活调试器的最简单的方法（在我们已经设置了断点后）就是——重新加载页面。因此让我们按下 `F5` (Windows, Linux) 或 `Cmd+R` (Mac) 吧。

设置断点之后，程序会在第 4 行暂停执行：



请打开右侧的信息下拉列表（箭头指示出的地方）。这里允许你查看当前的代码状态：

1. 察看 (Watch) —— 显示任意表达式的当前值。

你可以点击加号 `+` 然后输入一个表达式。调试器将随时显示它的值，并在执行过程中自动重新计算该表达式。

2. 调用栈 (Call Stack) —— 显示嵌套的调用链。

此时，调试器正在 `hello()` 的调用链中，被 `index.html` 中的一个脚本调用（这里没有函数，因此显示“anonymous”）

如果你点击了一个堆栈项，调试器将跳到对应的代码处，并且还可以查看其所有变量。

3. 作用域 (Scope) —— 显示当前的变量。

`Local` 显示当前函数中的变量，你还可以在源代码中看到它们的值高亮显示了出来。

Global 显示全局变量（不在任何函数中）。

这里还有一个 **this** 关键字，目前我们还没有学到它，不过我们很快就会学习它了。

跟踪执行

现在是 **跟踪** 脚本的时候了。

在右侧面板的顶部是一些关于跟踪脚本的按钮。让我们来使用它们吧。

► —— “恢复（Resume）”：继续执行，快捷键 **F8**。

继续执行。如果没有其他的断点，那么程序就会继续执行，并且调试器不会再控制程序。

我们点击它一下之后，我们会看到这样的情况：

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. On the left, the code editor displays `hello.js` with the following content:

```
function hello(name) {
  let phrase = `Hello, ${name}!`;
  say(phrase);
}

function say(phrase) {
  phrase = "Hello, John!";
  alert(`** ${phrase} **`);
}
```

A red arrow points from the text "nested calls" in the main content area to the "Call Stack" section of the right-hand sidebar. The sidebar shows the current state of the application:

- Paused on breakpoint**
- Watch**: No watch expressions
- Call Stack**:
 - say (hello.js:8)
 - hello (hello.js:4)
 - (anonymous) (index.html:10)
- Scope**
- Local**:
 - phrase: "Hello, John!"
 - this: Window
- Global**: Window

执行恢复了，执行到 `say()` 函数中的另外一个断点后暂停在了那里。看一下右边的“Call stack”。它已经增加了一个调用信息。我们现在在 `say()` 里面。

→ —— “下一步（Step）”：运行下一条指令，快捷键 **F9**。

运行下一条语句。如果我们现在点击它，`alert` 会被显示出来。

一次接一次地点击此按钮，整个脚本的所有语句会被逐个执行。

↷ —— “跨步（Step over）”：运行下一条指令，但 不会进入到一个函数中，快捷键 **F10**。

跟上一条命令“下一步（Step）”类似，但如果下一条语句是函数调用则表现不同。这里的函数指的是：不是内置的如 `alert` 函数等，而是我们自己写的函数。

“下一步（Step）”命令进入函数内部并在第一行暂停执行，而“跨步（Step over）”在无形中执行函数调用，跳过了函数的内部。

执行会在该函数执行后立即暂停。

如果我们对该函数的内部执行不感兴趣，这命令会很有用。

✿ —— “步入（Step into）”，快捷键 **F11**。

和“下一步（Step）”类似，但在异步函数调用情况下表现不同。如果你刚刚才开始学 JavaScript，那么你可以先忽略此差异，因为我们还没有用到异步调用。

至于之后，只需要记住“下一步（Step）”命令会忽略异步行为，例如 `setTimeout`（计划的函数调用），它会过一段时间再执行。而“步入（Step into）”会进入到代码中并等待（如果需要）。详见 [DevTools 手册](#)。

↑ —— “步出（Step out）”：继续执行到当前函数的末尾，快捷键 `Shift+F11`。

继续执行代码并停止在当前函数的最后一行。当我们使用 ↗ 偶然地进入到一个嵌套调用，但是我们又对这个函数不感兴趣时，我们想要尽可能的继续执行到最后的时候是非常方便的。

▶ —— 启用/禁用所有的断点。

这个按钮不会影响程序的执行。只是一个批量操作断点的开/关。

▶ —— 启用/禁用出现错误时自动暂停脚本执行。

当启动此功能并且开发者工具是打开着的时候，任何一个脚本的错误都会导致该脚本执行自动暂停。然后我们可以分析变量来看一下什么出错了。因此如果我们的脚本因为错误挂掉的时候，我们可以打开调试器，启用这个选项然后重载页面，查看一下哪里导致它挂掉了和当时的上下文是什么。

Continue to here

在代码中的某一行上右键，在显示的关联菜单（context menu）中点击一个非常有用的名字“Continue to here”的选项。

当你想要向前移动很多步到某一行为止，但是又懒得设置一个断点时非常的方便。

日志记录

想要输出一些东西到控制台上？`console.log` 函数可以满足你。

例如：将从 0 到 4 的值输出到控制台上：

```
// 打开控制台来查看
for (let i = 0; i < 5; i++) {
  console.log("value", i);
}
```

普通用户看不到这个输出，它是在控制台里面的。要想看到它——要么打开开发者工具中的 **Console**（控制台）选项卡，要么在一个其他的选项卡中按下 `Esc`：这会在下方打开一个控制台。

如果我们在代码中有足够的日志记录，那么我们可以从记录中看到刚刚发生了什么，而不需要借助调试器。

总结

我们可以看到，这里有 3 种方式来暂停一个脚本：

1. 一个断点。
2. `debugger` 语句。
3. 一个错误（如果开发者工具是打开状态，并且按钮 ① 是开启的状态）。

当脚本执行暂停时，我们就可以进行调试——检查变量，跟踪代码来查看执行出错的位置。

开发人员工具中的选项比本文介绍的多得多。完整的手册请点击这个链接查看：

<https://developers.google.com/web/tools/chrome-devtools>。

本章节的内容足够让你上手代码调试了，但是之后，尤其是你做了大量关于浏览器的东西后，推荐你查看上面那个链接中讲的开发者工具更高级的功能。

对了，你也可以点击开发者工具中的其他地方来看一下会显示什么。这可能是你学习开发者工具最快的方式了。不要忘了还有右键单击和关联菜单哟。

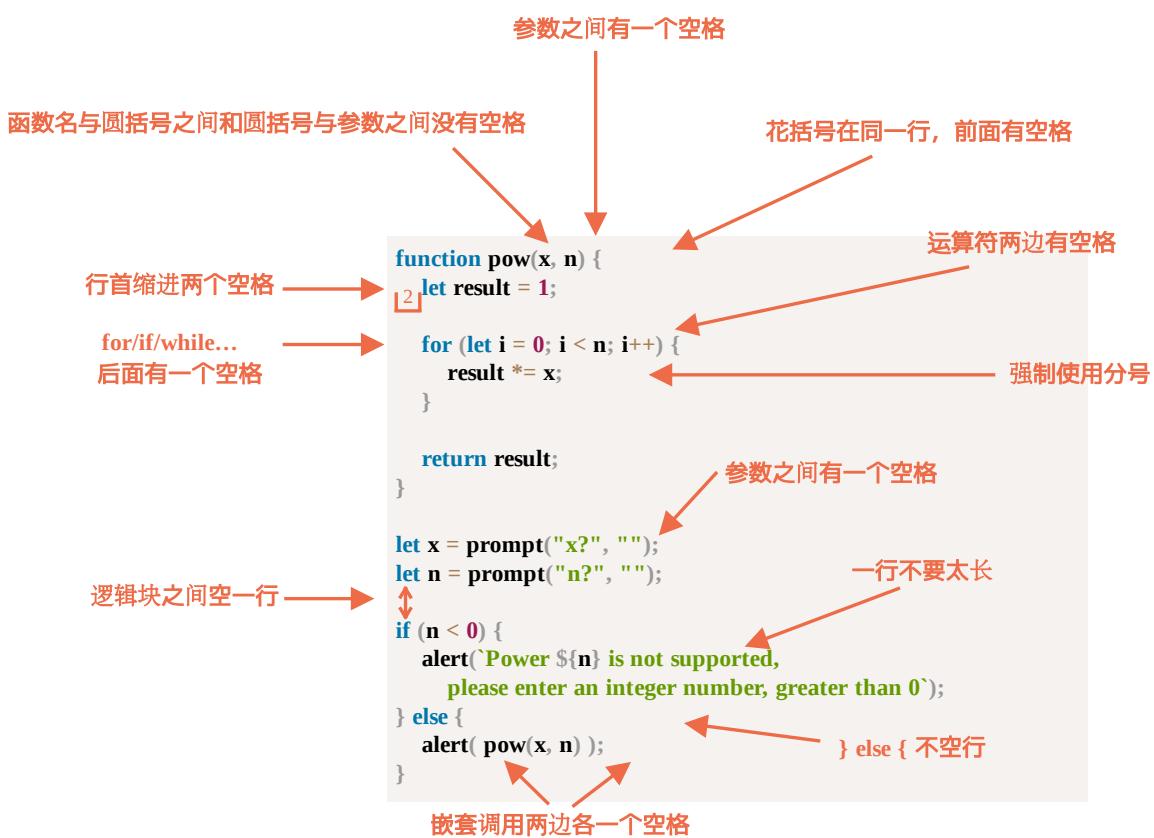
代码风格

我们的代码必须尽可能的清晰和易读。

这实际上是一种编程艺术——以一种正确并且人们易读的方式编码来完成一个复杂的任务。一个良好的代码风格大大有助于实现这一点。

语法

下面是一个备忘单，其中列出了一些建议的规则（详情请参阅下文）：



现在，让我们详细讨论一下这些规则和它们的原因吧。

⚠ 没有什么规则是“必须”的

没有什么规则是“刻在石头上”的。这些是风格偏好，而不是宗教教条。

花括号

在大多数的 JavaScript 项目中，花括号以“Egyptian”风格（译注：“egyptian”风格又称 K&R 风格 — 代码段的开括号位于一行的末尾，而不是另起一行的风格）书写，左花括号与相应的关键词在同一行上 — 而不是新起一行。左括号前还应该有一个空格，如下所示：

```
if (condition) {  
    // do this  
    // ...and that  
    // ...and that  
}
```

单行构造（如 `if (condition) doSomething()`）也是一个重要的用例。我们是否应该使用花括号？如果是，那么在哪里？

下面是这几种情况的注释，你可以自己判断一下它们的可读性：

1. ⓘ 初学者常这样写。非常不好！这里不需要花括号：

```
if (n < 0) {alert(`Power ${n} is not supported`);}
```

2. ⓘ 拆分为单独的行，不带花括号。永远不要这样做，添加新行很容易出错：

```
if (n < 0)  
    alert(`Power ${n} is not supported`);
```

3. ⓘ 写成一行，不带花括号 — 如果短的话，也是可以的：

```
if (n < 0) alert(`Power ${n} is not supported`);
```

4. ☺ 最好的方式：

```
if (n < 0) {  
    alert(`Power ${n} is not supported`);  
}
```

对于很短的代码，写成一行是可以接受的：例如 `if (cond) return null`。但是代码块（最后一个示例）通常更具可读性。

行的长度

没有人喜欢读一长串代码，最好将代码分割一下。

例如：

```
// 回勾引号 ` 允许将字符串拆分为多行  
let str = `  
    ECMA International's TC39 is a group of JavaScript developers,  
    implementers, academics, and more, collaborating with the community  
    to maintain and evolve the definition of JavaScript.  
`;
```

对于 `if` 语句:

```
if (
  id === 123 &&
  moonPhase === 'Waning Gibbous' &&
  zodiacSign === 'Libra'
) {
  letTheSorceryBegin();
}
```

一行代码的最大长度应该在团队层面上达成一致。通常是 80 或 120 个字符。

缩进

有两种类型的缩进:

- **水平方向上的缩进: 2 或 4 个空格。**

一个水平缩进通常由 2 或 4 个空格或者“Tab”制表符 (key `Tab`) 构成。选择哪一个方式是一场古老的圣战。如今空格更普遍一点。

选择空格而不是 `tabs` 的优点之一是，这允许你做出比“Tab”制表符更加灵活的缩进配置。

例如，我们可以将参数与左括号对齐，像下面这样:

```
show(parameters,
  aligned, // 左边有 5 个空格
  one,
  after,
  another
) {
// ...
}
```

- **垂直方向上的缩进: 用于将代码拆分成逻辑块的空行。**

即使是单个函数通常也被分割为数个逻辑块。在下面的示例中，初始化的变量、主循环结构和返回值都被垂直分割了:

```
function pow(x, n) {
  let result = 1;
  //           <-
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  //           <-
  return result;
}
```

插入一个额外的空行有助于使代码更具可读性。写代码时，不应该出现连续超过 9 行都没有被垂直分割的代码。

分号

每一个语句后面都应该有一个分号。即使它可能被跳过。

有一些编程语言的分号确实是可选的，那些语言中也很少使用分号。但是在 JavaScript 中，极少数情况下，换行符有时不会被解释为分号，这时代码就容易出错。更多内容请参阅[代码结构](#)一章的内容。

如果你是一个有经验的 JavaScript 程序员，你可以选择像[StandardJS ↗](#)这样的无分号的代码风格。否则，最好使用分号以避免可能出现的陷阱。大多数开发人员都应该使用分号。

嵌套的层级

尽量避免代码嵌套层级过深。

例如，在循环中，有时候使用 `continue` 指令以避免额外的嵌套是一个好主意。

例如，不应该像下面这样添加嵌套的 `if` 条件：

```
for (let i = 0; i < 10; i++) {
  if (cond) {
    ... // <- 又一层嵌套
  }
}
```

我们可以这样写：

```
for (let i = 0; i < 10; i++) {
  if (!cond) continue;
  ... // <- 没有额外的嵌套
}
```

使用 `if/else` 和 `return` 也可以做类似的事情。

例如，下面的两个结构是相同的。

第一个：

```
function pow(x, n) {
  if (n < 0) {
    alert("Negative 'n' not supported");
  } else {
    let result = 1;

    for (let i = 0; i < n; i++) {
      result *= x;
    }

    return result;
  }
}
```

第二个：

```
function pow(x, n) {
  if (n < 0) {
```

```

    alert("Negative 'n' not supported");
    return;
}

let result = 1;

for (let i = 0; i < n; i++) {
    result *= x;
}

return result;
}

```

但是第二个更具可读性，因为 `n < 0` 这个“特殊情况”在一开始就被处理了。一旦条件通过检查，代码执行就可以进入到“主”代码流，而不需要额外的嵌套。

函数位置

如果你正在写几个“辅助”函数和一些使用它们的代码，那么有三种方式来组织这些函数。

1. 在调用这些函数的代码的 上方 声明这些函数：

```

// 函数声明
function createElement() {
    ...
}

function setHandler(elem) {
    ...
}

function walkAround() {
    ...
}

// 调用函数的代码
let elem = createElement();
setHandler(elem);
walkAround();

```

2. 先写调用代码，再写函数

```

// 调用函数的代码
let elem = createElement();
setHandler(elem);
walkAround();

// --- 辅助函数 ---
function createElement() {
    ...
}

function setHandler(elem) {
    ...
}

```

```
function walkAround() {  
    ...  
}
```

3. 混合：在第一次使用一个函数时，对该函数进行声明。

大多数情况下，第二种方式更好。

这是因为阅读代码时，我们首先想要知道的是“它做了什么”。如果代码先行，那么在整个程序的最开始就展示出了这些信息。之后，可能我们就不再需要阅读这些函数了，尤其是他们的名字清晰地展示了他们的功能的时候。

风格指南

风格指南包含了“如何编写”代码的通用规则，例如：使用哪个引号、用多少空格来缩进、一行代码最大长度等非常多的细节。

当团队中的所有成员都使用相同的风格指南时，代码看起来将是统一的。无论是团队中谁写的，都是一样的风格。

当然，一个团队可以制定他们自己的风格指南，但是没必要这样做。现在已经有了很多制定好的代码风格指南可供选择。

一些受欢迎的选择：

- [Google JavaScript 风格指南 ↗](#)
- [Airbnb JavaScript 风格指南 ↗](#)
- [Idiomatic.JS ↗](#)
- [StandardJS ↗](#)
- 还有很多.....

如果你是一个初学者，你可以从本章中上面的内容开始。然后你可以浏览其他风格指南，并选择一个你最喜欢的。

自动检查器

检查器（Linters）是可以自动检查代码样式，并提出改进建议的工具。

他们的妙处在于进行代码风格检查时，还可以发现一些代码错误，例如变量或函数名中的错别字。因此，即使你不想坚持某一种特定的代码风格，我也建议你安装一个检查器。

下面是一些最出名的代码检查工具：

- [JSLint ↗](#) — 第一批检查器之一。
- [JSHint ↗](#) — 比 JSLint 多了更多设置。
- [ESLint ↗](#) — 应该是最新的一个。

它们都能够做好代码检查。我使用的是 [ESLint ↗](#)。

大多数检查器都可以与编辑器集成在一起：只需在编辑器中启用插件并配置代码风格即可。

例如，要使用 ESLint 你应该这样做：

1. 安装 [Node.js](#)。
2. 使用 `npm install -g eslint` 命令（`npm` 是一个 JavaScript 包安装工具）安装 ESLint。
3. 在你的 JavaScript 项目的根目录（包含该项目的所有文件的那个文件夹）创建一个名为 `.eslintrc` 的配置文件。
4. 在集成了 ESLint 的编辑器中安装/启用插件。大多数编辑器都有这个选项。

下面是一个 `.eslintrc` 文件的例子：

```
{  
  "extends": "eslint:recommended",  
  "env": {  
    "browser": true,  
    "node": true,  
    "es6": true  
  },  
  "rules": {  
    "no-console": 0,  
    "indent": ["warning", 2]  
  }  
}
```

这里的 `"extends"` 指令表示我们是基于 `"eslint:recommended"` 的设置项而进行设置的。之后，我们制定我们自己的规则。

你也可以从网上下载风格规则集并进行扩展。有关安装的更多详细信息，请参见 <http://eslint.org/docs/user-guide/getting-started>。

此外，某些 IDE 有内置的检查器，这非常方便，但是不像 ESLint 那样可自定义。

总结

本章描述的（和提到的代码风格指南中的）所有语法规则，都旨在帮助你提高代码可读性。他们都是值得商榷的。

当我们思考如何写“更好”的代码的时候，我们应该问自己的问题是：“什么可以让代码可读性更高，更容易被理解？”和“什么可以帮助我们避免错误？”这些是我们讨论和选择代码风格时要牢记的主要原则。

阅读流行的代码风格指南，可以帮助你了解有关代码风格的变化趋势和最佳实践的最新想法。

注释

正如我们在 [代码结构](#) 一章所了解到的那样，注释可以是以 `//` 开始的单行注释，也可以是 `/* ... */` 结构的多行注释。

我们通常通过注释来描述代码怎样工作和为什么这样工作。

乍一看，写注释可能很简单，但初学者在编程的时候，经常错误地使用注释。

糟糕的注释

新手倾向于使用注释来解释“代码中发生了什么”。就像这样：

```
// 这里的代码会先做这件事（.....）然后做那件事（.....）
// .....谁知道还有什么.....
very;
complex;
code;
```

但在好的代码中，这种“解释性”注释的数量应该是最少的。严格地说，就算没有它们，代码也应该很容易理解。

关于这一点有一个很棒的原则：“如果代码不够清晰以至于需要一个注释，那么或许它应该被重写。”

配方：分解函数

有时候，用一个函数来代替一个代码片段是更好的，就像这样：

```
function showPrimes(n) {
  nextPrime:
  for (let i = 2; i < n; i++) {

    // 检测 i 是否是一个质数（素数）
    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert(i);
  }
}
```

更好的变体，使用一个分解出来的函数 `isPrime`：

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

    alert(i);
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if (n % i == 0) return false;
  }

  return true;
}
```

现在我们可以很容易地理解代码了。函数自己就变成了一个注释。这种代码被称为 **自描述型** 代码。

配方：创建函数

如果我们有一个像下面这样很长的代码块：

```

// 在这里我们添加威士忌（译注：国外的一种酒）
for(let i = 0; i < 10; i++) {
  let drop = getWhiskey();
  smell(drop);
  add(drop, glass);
}

// 在这里我们添加果汁
for(let t = 0; t < 3; t++) {
  let tomato = getTomato();
  examine(tomato);
  let juice = press(tomato);
  add(juice, glass);
}

// ...

```

我们像下面这样，将上面的代码重构为函数，可能会是一个更好的变体：

```

addWhiskey(glass);
addJuice(glass);

function addWhiskey(container) {
  for(let i = 0; i < 10; i++) {
    let drop = getWhiskey();
    //...
  }
}

function addJuice(container) {
  for(let t = 0; t < 3; t++) {
    let tomato = getTomato();
    //...
  }
}

```

同样，函数本身就可以告诉我们发生了什么。没有什么地方需要注释。并且分割之后代码的结构也更好了。每一个函数做什么、需要什么和返回什么都非常地清晰。

实际上，我们不能完全避免“解释型”注释。例如在一些复杂的算法中，会有一些出于优化的目的而做的一些巧妙的“调整”。但是通常情况下，我们应该尽可能地保持代码的简单和“自我描述”性。

好的注释

所以，解释性注释通常来说都是不好的。那么哪一种注释才是好的呢？

描述架构

对组件进行高层次的整体概括，它们如何相互作用、各种情况下的控制流程是什么样的……简而言之——代码的鸟瞰图。有一个专门用于构建代码的高层次架构图，以对代码进行解释的特殊编程语言 [UML](#)。绝对值得学习。

记录函数的参数和用法

有一个专门用于记录函数的语法 [JSDoc](#)：用法、参数和返回值。

例如：

```
/**  
 * 返回 x 的 n 次幂的值。  
 *  
 * @param {number} x 要改变的值。  
 * @param {number} n 幂数，必须是一个自然数。  
 * @return {number} x 的 n 次幂的值。  
 */  
function pow(x, n) {  
    ...  
}
```

这种注释可以帮助我们理解函数的目的，并且不需要研究其内部的实现代码，就可以直接正确地使用它。

顺便说一句，很多诸如 [WebStorm](#) 这样的编辑器，都可以很好地理解和使用这些注释，来提供自动补全和一些自动化代码检查工作。

当然，也有一些像 [JSDoc 3](#) 这样的工具，可以通过注释直接生成 HTML 文档。你可以在 <http://usejsdoc.org/> 阅读更多关于 JSDoc 的信息。

为什么任务以这种方式解决？

写了什么代码很重要。但是为什么 不 那样写可能对于理解正在发生什么更重要。为什么任务是通过这种方式解决的？代码并没有给出答案。

如果有很多种方法都可以解决这个问题，为什么偏偏是这一种？尤其当它不是最显而易见的那一种的时候。

没有这样的注释的话，就可能会发生下面的情况：

1. 你（或者你的同事）打开了前一段时间写的代码，看到它不是最理想的实现方式。
2. 你会想：“我当时是有多蠢啊，现在我真是太聪明了”，然后用“更显而易见且正确的”方式重写了一遍。
3.重写的这股冲动劲是好的。但是在重写的过程中你发现“更显而易见”的解决方案实际上是有缺陷的。你甚至依稀地想起了为什么会这样，因为你很久之前就已经尝试过这样做了。于是你又还原了那个正确的实现，但是时间已经浪费了。

解决方案的注释非常的重要。它们可以帮助你以正确的方式继续开发。

代码有哪些巧妙的特性？它们被用在了什么地方？

如果代码存在任何巧妙和不显而易见的方法，那绝对需要注释。

总结

一个好的开发者的标志之一就是他的注释：它们的存在甚至它们的缺席（译注：在该注释的地方注释，在不需要注释的地方则不注释，甚至写得好的自描述函数本身就是一种注释）。

好的注释可以使我们更好地维护代码，一段时间之后依然可以更高效地回到代码高效开发。

注释这些内容：

- 整体架构，高层次的观点。

- 函数的用法。
- 重要的解决方案，特别是在不是很明显时。

避免注释：

- 描述“代码如何工作”和“代码做了什么”。
- 避免在代码已经足够简单或代码有很好的自描述性而不需要注释的情况下，还写些没必要的注释。

注释也被用于一些如 **JSDoc3** 等文档自动生成工具：他们读取注释然后生成 **HTML** 文档（或者其他格式的文档）。

忍者代码

学而不思则罔，思而不学则殆。

“ 孔子

过去的程序员忍者使用这些技巧，来使代码维护者的头脑更加敏锐。

代码审查大师在测试任务中寻找它们。

一些新入门的开发者有时候甚至比忍者程序员能够更好地使用它们。

仔细阅读本文，找出你是谁——一个忍者、一个新手、或者一个代码审查者？

⚠ 检测到讽刺意味

许多人试图追随忍者的脚步。只有极少数成功了。

简洁是智慧的灵魂

把代码尽可能写得短。展示出你是多么的聪明啊。

在编程中，多使用一些巧妙的编程语言特性。

例如，看一下这个三元运算符 '`? :`'：

```
// 从一个著名的 JavaScript 库中截取的代码
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

很酷，对吗？如果你这样写了，那些看到这一行代码并尝试去理解 `i` 的值是什么的开发者们，就会有一个“快活的”的时光了。然后会来找你寻求答案。

告诉他短一点总是更好的。引导他进入忍者之路。

一个字母的变量

道隐无名。夫唯道善贷且成。

“ 老子（道德经）

另一个提高编程速度的方法是，到处使用单字母的变量名。例如 `a`、`b` 或 `c`。

短变量就像森林中真正的忍者一样，一下就找不到了。没有人能够通过编辑器的“搜索”功能找到它。即使有人做到了，他也不能“破译”出变量名 `a` 或 `b` 到底是什么意思。

.....但是有一个例外情况。一个真正的忍者绝不会在 `"for"` 循环中使用 `i` 作为计数器。在任何地方都可以，但是这里不会用。你随便一找，就能找到很多不寻常的字母。例如 `x` 或 `y`。

使用一个不寻常的变量多酷啊，尤其是在长达 1-2 页（如果可以的话，你可以写得更长）的循环体中使用的时候。如果某人要研究循环内部实现的时候，他就很难很快地找出变量 `x` 其实是循环计数器啦。

使用缩写

如果团队规则中禁止使用一个字母和模糊的命名 — 那就缩短命名，使用缩写吧。

像这样：

- `list` → `lst`
- `userAgent` → `ua`
- `browser` → `brsr`
-等

只有具有真正良好直觉的人，才能够理解这样的命名。尽可能缩短一切。只有真正有价值的人，才能够维护这种代码的开发。

Soar high，抽象化。

大方无隅，
大器晚成，
大音希声，
大象无形。

“老子（道德经）

当选择一个名字时，尽可能尝试使用最抽象的词语。例如 `obj`、`data`、`value`、`item` 和 `elem` 等。

- 一个变量的理想名称是 `data`。在任何能用的地方都使用它。的确，每个变量都持有 数据 (`data`)，对吧？

.....但是 `data` 已经用过了怎么办？可以尝试一下 `value`，它也很普遍。毕竟，一个变量总会有一个 值 (`value`)，对吧？

- 根据变量的类型为变量命名：`str`、`num`

尝试一下吧。新手可能会诧异 — 这些名字对于忍者来说真的有用吗？事实上，有用的！

一方面，变量名仍然有着一些含义。它说明了变量内是什么：一个字符串、一个数字或是其他的东西。但是当一个局外人试图理解代码时，他会惊讶地发现实际上没有任何有效信息！最终就无法修改你精心思考过的代码。

我们可以通过代码调试，很容易地看出值的类型。但是变量名的含义呢？它存了哪一个字符串或数字？

如果思考的深度不够，是没有办法搞明白的。

-但是如果找不到更多这样的名字呢？可以加一个数字： `data1, item2, elem5`

注意测试

只有一个真正细心的程序员才能理解你的代码。但是怎么检验呢？

方式之一——使用相似的变量名，像 `date` 和 `data`。

尽你所能地将它们混合在一起。

想快速阅读这种代码是不可能的。并且如果有一个错别字时.....额.....我们卡在这儿好长时间了，到饭点了(⊙v⊙)。

智能同义词

最难的事情是在黑暗的房间里找到一只黑猫，尤其是如果没有猫。

“ 孔子

对同一个东西使用类似命名，可以使生活更有趣，并且能够展现你的创造力。

例如，函数前缀。如果一个函数的功能是在屏幕上展示一个消息 — 名称可以以 `display...` 开头，例如 `displayMessage`。如果另一个函数展示别的东西，比如一个用户名，名称可以以 `show...` 开始（例如 `showName`）。

暗示这些函数之间有微妙的差异，实际上并没有。

与团队中的其他忍者们达成一个协议：如果张三在他的代码中以 `display...` 来开始一个“显示”函数，那么李四可以用 `render...`，王二可以使用 `paint...`。你可以发现代码变得多么地有趣多样呀。

.....现在是帽子戏法！

对于有非常重要的差异的两个函数 — 使用相同的前缀。

例如，`printPage(page)` 函数会使用一个打印机 (`printer`)。`printText(text)` 函数会将文字显示到屏幕上。让一个不熟悉的读者来思考一下：“名字为 `printMessage(message)` 的函数会将消息放到哪里呢？打印机还是屏幕上？”为了让代码真正耀眼，`printMessage(message)` 应该将消息输出到新窗口中！

重用名字

始制有名，
名亦既有名，
夫亦将知止，
知止可以不殆。

“ 老子（道德经）

仅在绝对必要时才添加新变量。

否则，重用已经存在的名字。直接把新值写进变量即可。

在一个函数中，尝试仅使用作为参数传递的变量。

这样就很难确定这个变量的值现在是什么了。也不知道它是从哪里来的。目的是提高阅读代码人的直觉和记忆力。一个直觉较弱的人必须逐行分析代码，跟踪每个代码分支中的更改。

这个方法的一个进阶方案是，在循环或函数中偷偷地替换掉它的值。

例如：

```
function ninjaFunction(elem) {  
    // 基于变量 elem 进行工作的 20 行代码  
  
    elem = clone(elem);  
  
    // 又 20 行代码，现在使用的是 clone 后的 elem 变量。  
}
```

想要在后半部分中使用 `elem` 的程序员会感到很诧异……只有在调试期间，检查代码之后，他才会发现他正在使用克隆过的变量！

经常看到这样的代码，即使对经验丰富的忍者来说也是致命的。

下划线的乐趣

在变量名前加上下划线 `_` 和 `__`。例如 `_name` 和 `__value`。如果只有你知道他们的含义，那就非常棒了。或者，加这些下划线只是为了好玩儿，没有任何含义，那就更棒了！

加下划线可谓是一箭双雕。首先，代码变得更长，可读性更低；并且，你的开发者小伙伴可能会花费很长时间，来弄清楚下划线是什么意思。

聪明的忍者会在代码的一个地方使用下划线，然后在其他地方刻意避免使用它们。这会使代码变得更加脆弱，并提高了代码未来出现错误的可能性。

展示你的爱

向大家展现一下你那丰富的情感！像 `superElement`、`megaFrame` 和 `niceItem` 这样的名字一定会启发读者。

事实上，从一方面来说，看似写了一些东西：`super..`、`mega..`、`nice..`。但从另一方面来说——并没有提供任何细节。阅读代码的人可能需要耗费一到两个小时的带薪工作时间，冥思苦想来寻找一个隐藏的含义。

重叠外部变量

处明者不见暗中一物，
处暗者能见明中区事。

“ 关尹子

对函数内部和外部的变量，使用相同的名称。很简单，不用费劲想新的名称。

```
let user = authenticateUser();
```

```
function render() {  
  let user = anotherValue();  
  ...  
  ...许多行代码...  
  ...  
  ... // <-- 某个程序员想要在这里使用 user 变量.....  
  ...  
}
```

在研究 `render` 内部代码的程序员可能不会注意到，有一个内部变量 `user` 屏蔽了外部的 `user` 变量。

然后他会假设 `user` 仍然是外部的变量然后使用它，`authenticateUser()` 的结果.....陷阱出来啦！你好呀，调试器.....

无处不在的副作用！

有些函数看起来它们不会改变任何东西。例如 `isReady()`, `checkPermission()`, `findTags()`它们被假定用于执行计算、查找和返回数据，而不会更改任何他们自身之外的数据。这被称为“无副作用”。

一个非常惊喜的技巧就是，除了主要任务之外，给它们添加一个“有用的”行为。

当你的同事看到被命名为 `is..`、`check..` 或 `find...` 的函数改变了某些东西的时候，他脸上肯定是一脸懵逼的表情 — 这会扩大你的理性界限。

另一个惊喜的方式是，返回非标准的结果。

展示你原来的想法！让调用 `checkPermission` 时的返回值不是 `true/false`，而是一个包含检查结果的复杂对象。

那些尝试写 `if (checkPermission(...))` 的开发者，会很疑惑为什么它不能工作。告诉他们：“去读文档吧”。然后给出这篇文章。

强大的函数！

大道泛兮，
其左可右。

“ 老子（道德经）

不要让函数受限于名字中写的内容。拓宽一些。

例如，函数 `validateEmail(email)` 可以（除了检查邮件的正确性之外）显示一个错误消息并要求重新输入邮件。

额外的行为在函数名称中不应该很明显。一个真正的忍者会使它们在代码中也不明显。

将多个行为合并到一起，可以保护你的代码不被重用。

想象一下，另一个开发者只想检查邮箱而不想输出任何信息。你的函数 `validateEmail(email)` 对他而言就不合适啦。所以他不会找你问关于这些函数的任何事而打断你的思考。

总结

上面的所有“建议”都是从真实的代码中提炼而来的……有时候，这些代码是由有经验的开发者写的。也许比你更有经验 ;)

- 遵从其中的一丢丢，你的代码就会变得充满惊喜。
- 遵从其中的一大部分，你的代码将真正成为你的代码，没有人会想改变它。
- 遵从所有，你的代码将成为寻求启发的年轻开发者的宝贵案例。

使用 Mocha 进行自动化测试

自动化测试将被用于进一步的任务中，并且还将被广泛应用在实际项目中。

为什么我们需要测试？

当我们在写一个函数时，我们通常可以想象出它应该做什么：哪些参数会给出哪些结果。

在开发期间，我们可以通过运行程序来检查它并将结果与预期进行比较。例如，我们可以在控制台中这么做。

如果出了问题——那么我们会修复代码，然后再一次运行并检查结果——直到它工作为止。

但这样的手动“重新运行”是不完美的。

当通过手动重新运行来测试代码时，很容易漏掉一些东西。

例如，我们要创建一个函数 `f`。写一些代码，然后测试：`f(1)` 可以执行，但是 `f(2)` 不执行。我们修复了一下代码，现在 `f(2)` 可以执行了。看起来已经搞定了？但是我们忘了重新测试 `f(1)`。这样有可能会导致出现错误。

这是非常典型的。当我们在开发一些东西时，我们会保留很多可能需要的用例。但是不要想着程序员在每一次代码修改后都去检查所有的案例。所以这就很容易造成修复了一个问题却造成另一个问题的情况。

自动化测试意味着测试是独立于代码的。它们以各种方式运行我们的函数，并将结果与预期结果进行比较。

行为驱动开发（BDD）

我们来使用一种名为 [行为驱动开发](#) 或简言为 BDD 的技术。

BDD 包含了三部分内容：测试、文档和示例。

为了理解 BDD，我们将研究一个实际的开发案例。

开发“pow”：规范

我们想要创建一个函数 `pow(x, n)` 来计算 `x` 的 `n` 次幂 (`n` 为整数)。我们假设 `n ≥ 0`。

这个任务只是一个例子：JavaScript 中有一个 `**` 运算符可以用于幂运算。但是在这里我们专注于可以应用于更复杂任务的开发流程上。

在创建函数 `pow` 的代码之前，我们可以想象函数应该做什么并且描述出来。

这样的描述被称作 **规范** (**specification, spec**)，包含用例的描述以及针对它们的测试，如下所示：

```
describe("pow", function() {  
  it("raises to n-th power", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
});
```

正如你所看到的，一个规范包含三个主要的模块：

describe("title", function() { ... })

表示我们正在描述的功能是什么。在我们的例子中我们正在描述函数 `pow`。用于组织“工人（workers）”—— `it` 代码块。

it("use case description", function() { ... })

`it` 里面的描述部分，我们以一种 **易于理解** 的方式描述特定的用例，第二个参数是用于对其进行测试的函数。

assert.equal(value1, value2)

`it` 块中的代码，如果实现是正确的，它应该在执行的时候不产生任何错误。

`assert.*` 函数用于检查 `pow` 函数是否按照预期工作。在这里我们使用了其中之一—— `assert.equal`，它会对参数进行比较，如果它们不相等则会抛出一个错误。这里它检查了 `pow(2, 3)` 的值是否等于 `8`。还有其他类型的比较和检查，我们将在后面介绍到。

规范可以被执行，它将运行在 `it` 块中指定的测试。我们稍后会看到。

开发流程

开发流程通常看起来像这样：

1. 编写初始规范，测试最基本的功能。
2. 创建一个最初始的实现。
3. 检查它是否工作，我们运行测试框架 [Mocha ↗](#)（很快会有更多细节）来运行测试。当功能未完成时，将显示错误。我们持续修正直到一切都能工作。
4. 现在我们有一个带有测试的能工作的初步实现。
5. 我们增加更多的用例到规范中，或许目前的程序实现还不支持。无法通过测试。
6. 回到第 3 步，更新程序直到测试不会抛出错误。
7. 重复第 3 步到第 6 步，直到功能完善。

如此来看，开发就是不断地 **迭代**。我们写规范，实现它，确保测试通过，然后写更多的测试，确保它们工作等等。最后，我们有了一个能工作的实现和针对它的测试。

让我们在我们的开发案例中看看这个开发流程吧。

在我们的案例中，第一步已经完成了：我们有一个针对 `pow` 的初始规范。因此让我们来实现它吧。但在此之前，让我们用一些 `JavaScript` 库来运行测试，就是看看测试是通过了还是失败了。

行为规范

在本教程中，我们将使用以下 **JavaScript** 库进行测试：

- **Mocha** —— 核心框架：提供了包括通用型测试函数 `describe` 和 `it`，以及用于运行测试的主函数。
- **Chai** —— 提供很多断言（`assertion`）支持的库。它提供了很多不同的断言，现在我们只需要用 `assert.equal`。
- **Sinon** —— 用于监视函数、模拟内置函数和其他函数的库，我们在后面才会用到它。

这些库都既适用于浏览器端，也适用于服务器端。这里我们将使用浏览器端的变体。

包含这些框架和 `pow` 规范的完整的 **HTML** 页面：

```
<!DOCTYPE html>
<html>
<head>
    <!-- add mocha css, to show results -->
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.css">
    <!-- add mocha framework code -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.js"></script>
    <script>
        mocha.setup('bdd'); // minimal setup
    </script>
    <!-- add chai -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js"></script>
    <script>
        // chai has a lot of stuff, let's make assert global
        let assert = chai.assert;
    </script>
</head>

<body>

<script>
    function pow(x, n) {
        /* function code is to be written, empty now */
    }
</script>

<!-- the script with tests (describe, it...) -->
<script src="test.js"></script>

<!-- the element with id="mocha" will contain test results -->
<div id="mocha"></div>

<!-- run tests! -->
<script>
    mocha.run();
</script>
</body>

</html>
```

该页面可分为五个部分：

1. `<head>` —— 添加用于测试的第三方库和样式文件。
2. `<script>` 包含测试函数，在我们的例子中——和 `pow` 相关的代码。

3. 测试代码——在我们的案例中是名为 `test.js` 的脚本，它包含上面 `describe("pow", ...)` 的那些代码。
4. HTML 元素 `<div id="mocha">` 将被 Mocha 用来输出结果。
5. 可以使用 `mocha.run()` 命令来开始测试。

结果：

The screenshot shows the Mocha test runner interface. At the top right, it displays "passes: 0 failures: 1 duration: 0.23s 100%". Below this, there's a section titled "pow" with a red error message: "✗ raises to n-th power". A detailed error box shows an "AssertionError: expected undefined to equal 8 at Context.<anonymous> (test.js:4:12)".

到目前为止，测试失败了，出现了一个错误。这是合乎逻辑的：我们的 `pow` 是一个空函数，因此 `pow(2, 3)` 返回了 `undefined` 而不是 `8`。

未来，我们会注意到有更高级的测试工具，像是 [karma](#) 或其他的，使自动运行许多不同的测试变得更容易。

初始实现

为了可以通过测试，让我们写一个 `pow` 的简单实现：

```
function pow() {
  return 8; // :) 我们作弊啦!
}
```

哇哦，现在它可以工作了。

The screenshot shows the Mocha test runner interface. At the top right, it displays "passes: 1 failures: 0 duration: 0.21s 100%". Below this, there's a section titled "pow" with a green checkmark indicating success: "✓ raises to n-th power".

改进规范

我们所做的这些绝对是作弊。函数是不起作用的：尝试计算 `pow(3, 4)` 的话就会得到一个不正确的结果，但是测试却通过了。

.....但是这种情况却是在实际中相当典型例子。测试通过了，但是函数却是错误的。我们的规范是不完善的。我们需要给它添加更多的测试用例。

这里我们又添加了一个测试来检查 `pow(3, 4) = 81`。

我们可以选择两种方式中的任意一种来组织测试代码：

1. 第一种——在同一个 `it` 中再添加一个 `assert`：

```
describe("pow", function() {
  it("raises to n-th power", function() {
    assert.equal(pow(2, 3), 8);
    assert.equal(pow(3, 4), 81);
  });
});
```

2. 第二种——写两个测试：

```
describe("pow", function() {
  it("2 raised to power 3 is 8", function() {
    assert.equal(pow(2, 3), 8);
  });

  it("3 raised to power 4 is 81", function() {
    assert.equal(pow(3, 4), 81);
  });
});
```

主要的区别是，当 `assert` 触发一个错误时，`it` 代码块会立即终止。因此，在第一种方式中，如果第一个 `assert` 失败了，我们将永远不会看到第二个 `assert` 的结果。

保持测试之间独立，有助于我们获知代码中正在发生什么，因此第二种方式更好一点。

除此之外，还有一个规范值得遵循。

一个测试检查一个东西。

如果我们在看测试代码的时候，发现在其中有两个相互独立的检查——最好将它拆分成两个更简单的检查。

因此让我们继续使用第二种方式。

结果：

passes: 1 failures: 1 duration: 0.22s 100%

pow

- ✓ 2 raised to power 3 is 8
- ✗ 3 raised to power 4 is 81

```
AssertionError: expected 8 to equal 81
  at Context.<anonymous> (test.js:8:12)
```

正如我们可以想到的，第二条测试失败了。当然啦，我们的函数总会返回 8，而 assert 期望的是 81。

改进实现

让我们写一些更加实际的代码来通过测试吧：

```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

为了确保函数可以很好地工作，我们来使用更多值测试它吧。除了手动地编写 it 代码块，我们可以使用 for 循环来生成它们：

```
describe("pow", function() {

  function makeTest(x) {
    let expected = x * x * x;
    it(`${x} in the power 3 is ${expected}`, function() {
      assert.equal(pow(x, 3), expected);
    });
  }

  for (let x = 1; x <= 5; x++) {
    makeTest(x);
  }
});
```

结果：

passes: 5 failures: 0 duration: 0.20s 100%

pow

- ✓ 1 in the power 3 is 1
- ✓ 2 in the power 3 is 8
- ✓ 3 in the power 3 is 27
- ✓ 4 in the power 3 is 64
- ✓ 5 in the power 3 is 125



嵌套描述

我们继续添加更多的测试。但在此之前，我们需要注意到辅助函数 `makeTest` 和 `for` 应该被组合到一起。我们在其他测试中不需要 `makeTest`，只有在 `for` 循环中需要它：它们共同的任务就是检查 `pow` 是如何自乘至给定的幂次方。

使用嵌套的 `describe` 来进行分组：

```
describe("pow", function() {  
  
  describe("raises x to power 3", function() {  
  
    function makeTest(x) {  
      let expected = x * x * x;  
      it(`$x in the power 3 is ${expected}`, function() {  
        assert.equal(pow(x, 3), expected);  
      });  
    }  
  
    for (let x = 1; x <= 5; x++) {  
      makeTest(x);  
    }  
  
  });  
  
  // .....可以在这里写更多的测试代码, describe 和 it 都可以添加在这。  
});
```

嵌套的 `describe` 定义了一个新的“subgroup”测试。在输出中我们可以看到带有标题的缩进：

passes: 5 failures: 0 duration: 0.20s 100%

pow

- ### raises x to power 3
- ✓ 1 in the power 3 is 1
 - ✓ 2 in the power 3 is 8
 - ✓ 3 in the power 3 is 27
 - ✓ 4 in the power 3 is 64
 - ✓ 5 in the power 3 is 125



将来，我们可以在顶级域中使用 `it` 和 `describe` 的辅助函数添加更多的 `it` 和 `describe`，它们不会看到 `makeTest`。

i `before/after` 和 `beforeEach/afterEach`

我们可以设置 `before/after` 函数来在运行测试之前/之后执行。也可以使用 `beforeEach/afterEach` 函数来设置在执行 每一个 `it` 之前/之后执行。

例如：

```
describe("test", function() {  
  
  before(() => alert("Testing started - before all tests"));  
  after(() => alert("Testing finished - after all tests"));  
  
  beforeEach(() => alert("Before a test - enter a test"));  
  afterEach(() => alert("After a test - exit a test"));  
  
  it('test 1', () => alert(1));  
  it('test 2', () => alert(2));  
  
});
```

运行顺序将为：

```
Testing started - before all tests (before)  
Before a test - enter a test (beforeEach)  
1  
After a test - exit a test (afterEach)  
Before a test - enter a test (beforeEach)  
2  
After a test - exit a test (afterEach)  
Testing finished - after all tests (after)
```

[Open the example in the sandbox.](#)

通常，`beforeEach/afterEach` 和 `before/after` 被用于执行初始化，清零计数器或做一些介于每个测试（或测试组）之间的事情。

延伸规范

`pow` 的基础功能已经完成了。第一次迭代开发完成啦。当我们庆祝和喝完香槟之后，让我们继续改进它吧。

正如前面所说，函数 `pow(x, n)` 适用于正整数 `n`。

`JavaScript` 函数通常会返回 `Nan` 以表示一个数学错误。接下来我们对无效的 `n` 值执行相同的操作。

让我们首先将这个行为加到规范中(!)：

```
describe("pow", function() {
```

```
// ...

it("for negative n the result is NaN", function() {
  assert.isNaN(pow(2, -1));
});

it("for non-integer n the result is NaN", function() {
  assert.isNaN(pow(2, 1.5));
});

});
```

新测试的结果：

```
passes: 5 failures: 2 duration: 0.20s 100%
```

pow

✖ if n is negative, the result is NaN

```
AssertionError: expected 1 to be NaN
  at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:1:1)
  at Context.<anonymous> (test.js:19:12)
```

✖ if n is not integer, the result is NaN

```
AssertionError: expected 4 to be NaN
  at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:1:1)
  at Context.<anonymous> (test.js:23:12)
```

raises x to power 3

✓ 1 in the power 3 is 1
✓ 2 in the power 3 is 8
✓ 3 in the power 3 is 27
✓ 4 in the power 3 is 64
✓ 5 in the power 3 is 125

新加的测试失败了，因为我们的实现方式是不支持它们的。这就是 BDD 的做法：我们首先写一些暂时无法通过的测试，然后去实现它们。

i Other assertions

请注意断言语句 `assert.isNaN`: 它用来检查 `Nan`。

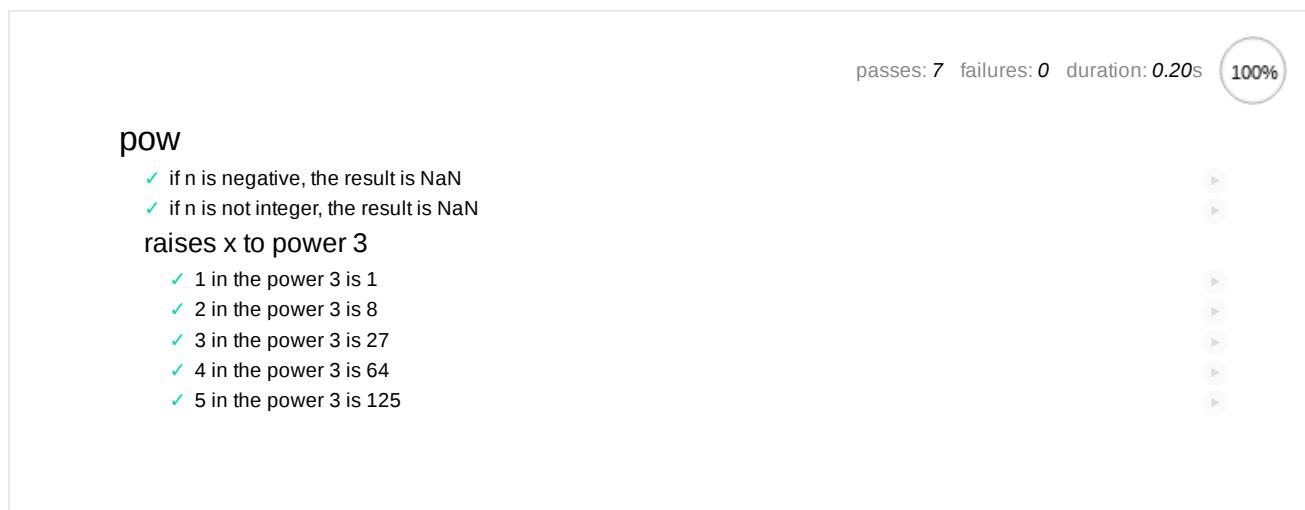
在 [Chai ↗](#) 中也有其他的断言，例如：

- `assert.equal(value1, value2)` —— 检查相等 `value1 == value2`。
- `assert.strictEqual(value1, value2)` —— 检查严格相等 `value1 === value2`。
- `assert.notEqual`, `assert.notStrictEqual` —— 执行和上面相反的检查。
- `assert.isTrue(value)` —— 检查 `value === true`。
- `assert.isFalse(value)` —— 检查 `value === false`。
-完整的列表请见 [docs ↗](#)

因此我们应该给 `pow` 再加几行：

```
function pow(x, n) {  
    if (n < 0) return NaN;  
    if (Math.round(n) != n) return NaN;  
  
    let result = 1;  
  
    for (let i = 0; i < n; i++) {  
        result *= x;  
    }  
  
    return result;  
}
```

现在它可以工作了，所有的测试也都通过了：



passes: 7 failures: 0 duration: 0.20s 100%

pow

- ✓ if n is negative, the result is NaN
- ✓ if n is not integer, the result is NaN
- raises x to power 3
 - ✓ 1 in the power 3 is 1
 - ✓ 2 in the power 3 is 8
 - ✓ 3 in the power 3 is 27
 - ✓ 4 in the power 3 is 64
 - ✓ 5 in the power 3 is 125

[Open the full final example in the sandbox. ↗](#)

总结

在 BDD 中，规范先行，实现在后。最后我们同时拥有了规范和代码。

规范有三种使用方式：

1. 作为 **测试** —— 保证代码正确工作。
2. 作为 **文档** —— `describe` 和 `it` 的标题告诉我们函数做了什么。
3. 作为 **案例** —— 测试实际工作的例子展示了一个函数可以被怎样使用。

有了规范，我们可以安全地改进、修改甚至重写函数，并确保它仍然正确地工作。

这在一个函数会被用在多个地方的大型项目中尤其重要。当我们改变这样一个函数时，没有办法手动检查每个使用它们的地方是否仍旧正确。

如果没有测试，一般有两个办法：

1. 展示修改，无论修改了什么。然后我们的用户遇到了 **bug**，这应该是我们没有手动完成某些检查。
2. 如果对出错的惩罚比较严重，并且没有测试，那么大家会很害怕修改这样的函数，然后这些代码就会越来越陈旧，没有人会想接触它。这很不利于发展。

自动化测试则有助于避免这样的问题！

如果这个项目被测试代码覆盖了，就不会出现这种问题。在任何修改之后，我们都可以运行测试，并在几秒钟内看到大量的检查。

另外，一个经过良好测试的代码通常都有更好的架构。

当然，这是因为覆盖了自动化测试的代码更容易修改和改进。但还有另一个原因。

要编写测试，代码的组织方式应确保每个函数都有一个清晰描述的任务、定义良好的输入和输出。这意味着从一开始就有一个好的架构。

在实际开发中有时候可能不容易，有时很难在写实际代码之前编写规范，因为还不清楚它应该如何表现。但一般来说，编写测试使得开发更快更稳定。

在本教程的后面部分，你将遇到许多包含了测试的任务。所以你会看到更多的实际例子。

编写测试需要良好的 **JavaScript** 知识。但我们刚刚开始学习它。因此，为了解决所有问题，到目前为止，你不需要编写测试，但是你应该已经能够阅读测试了，即使它们比本章中的内容稍微复杂一些。

Polyfill

JavaScript 语言在稳步发展。也会定期出现一些对语言的新提议，它们会被分析讨论，如果认为有价值，就会被加入到 <https://tc39.github.io/ecma262/> 的列表中，然后被加到 [规范](#) 中。

JavaScript 引擎背后的团队关于首先要实现什么有着他们自己想法。他们可能会决定执行草案中的建议，并推迟已经在规范中的内容，因为它们不太有趣或者难以实现。

因此，一个 **JavaScript** 引擎只能实现标准中的一部分是很常见的情况。

查看语言特性的当前支持状态的一个很好的页面是 <https://kangax.github.io/compat-table/es6/>（它很大，我们现在还有很多东西要学）。

Babel

当我们使用语言的一些现代特性时，一些引擎可能无法支持这样的代码。正如上所述，并不是所有功能在任何地方都有实现。

这就是 Babel 来拯救的东西。

Babel [↗](#) 是一个 [transpiler](#) [↗](#)。它将现代的 JavaScript 代码转化为以前的标准形式。

实际上，Babel 包含了两部分：

1. 第一，用于重写代码的 [transpiler](#) 程序。开发者在自己的电脑上运行它。它以之前的语言标准对代码进行重写。然后将代码传到面向用户的网站。像 [webpack](#) [↗](#) 这样的现代项目构建系统，提供了在每次代码改变时自动运行 [transpiler](#) 的方法，因此很容易集成在开发过程中。
2. 第二，[polyfill](#)。

新的语言特性可能包括新的内建函数和语法结构。[transpiler](#) 会重写代码，将语法结构转换为旧的结构。但是对于新的内建函数，需要我们去实现。JavaScript 是一个高度动态化的语言。脚本可以添加/修改任何函数，从而使它们的行为符合现代标准。

更新/添加新函数的脚本称为“[polyfill](#)”。它“填补”了缺口，并添加了缺少的实现。

两个有意思的 [polyfills](#) 是：

- [core js](#) [↗](#) 支持很多，允许只包含需要的功能。
- [polyfill.io](#) [↗](#) 根据功能和用户的浏览器，为脚本提供 [polyfill](#) 的服务。

所以，如果我们要使用现代语言功能，[transpiler](#) 和 [polyfill](#) 是必要的。

教程中的案例

当你正在阅读离线版本时，在 PDF 中，示例是不可运行的。在 EPUB 格式中，部分例子可以运行。

Google Chrome 通常是对新语言特性支持更新最快的，在没有任何 [transpiler](#) 的情况下，也能很好地运行前沿的演示，当然其他的现代浏览器也挺好。

Object（对象）：基础知识

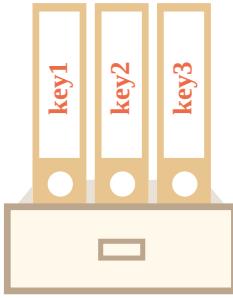
对象

正如我们在 [数据类型](#) 一章学到的，JavaScript 中有八种数据类型。有七种原始类型，因为它们的值只包含一种东西（字符串、数字或者其他）。

相反，对象则用来存储键值对和更复杂的实体。在 JavaScript 中，对象几乎渗透到了这门编程语言的方方面面。所以，在我们深入理解这门语言之前，必须先理解对象。

我们可以通过使用带有可选 [属性列表](#) 的花括号 `{...}` 来创建对象。一个属性就是一个键值对（“`key: value`”），其中键（`key`）是一个字符串（也叫做属性名），值（`value`）可以是任何值。

我们可以把对象想象成一个带有签名文件的文件柜。每一条数据都基于键（`key`）存储在文件中。这样我们就可以很容易根据文件名（也就是“键”）查找文件或添加/删除文件了。



我们可以用下面两种语法中的任一种来创建一个空的对象（“空柜子”）：

```
let user = new Object(); // "构造函数" 的语法  
let user = {}; // "字面量" 的语法
```



通常，我们用花括号。这种方式我们叫做**字面量**。

文本和属性

我们可以在创建对象的时候，立即将一些属性以键值对的形式放到 `{...}` 中。

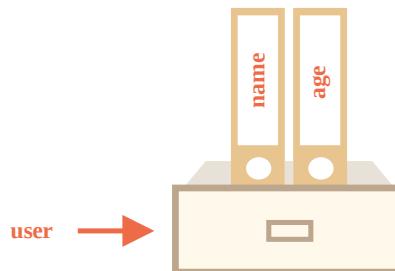
```
let user = { // 一个对象  
  name: "John", // 键 "name", 值 "John"  
  age: 30       // 键 "age", 值 30  
};
```

属性有键（或者也可以叫做“名字”或“标识符”），位于冒号 `:` 的前面，值在冒号的右边。

在 `user` 对象中，有两个属性：

1. 第一个的键是 `"name"`，值是 `"John"`。
2. 第二个的键是 `"age"`，值是 `30`。

生成的 `user` 对象可以被想象为一个放置着两个标记有 `"name"` 和 `"age"` 的文件的柜子。



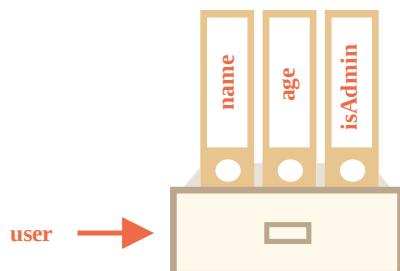
我们可以随时添加、删除和读取文件。

可以使用点符号访问属性值：

```
// 读取文件的属性:  
alert( user.name ); // John  
alert( user.age ); // 30
```

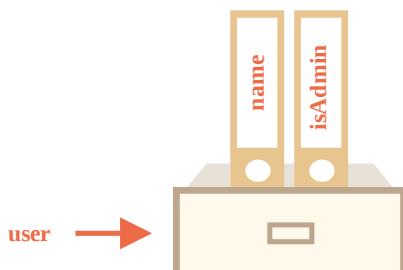
属性的值可以是任意类型，让我们加个布尔类型：

```
user.isAdmin = true;
```



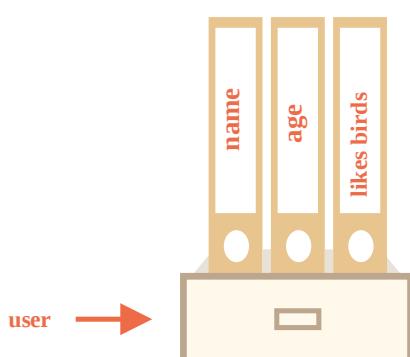
我们可以用 `delete` 操作符移除属性：

```
delete user.age;
```



我们也可以用多字词语来作为属性名，但必须给它们加上引号：

```
let user = {  
  name: "John",  
  age: 30,  
  "likes birds": true // 多词属性名必须加引号  
};
```



列表中的最后一个属性应以逗号结尾:

```
let user = {  
    name: "John",  
    age: 30,  
}
```

这叫做尾随 (trailing) 或悬挂 (hanging) 逗号。这样便于我们添加、删除和移动属性，因为所有的行都是相似的。

方括号

对于多词属性，点操作就不能用了:

```
// 这将提示有语法错误  
user.likes birds = true
```

JavaScript 理解不了。它认为我们在处理 `user.likes`，然后在遇到意外的 `birds` 时给出了语法错误。

点符号要求 `key` 是有效的变量标识符。这意味着：不包含空格，不以数字开头，也不包含特殊字符（允许使用 `$` 和 `_`）。

有另一种方法，就是使用方括号，可用于任何字符串:

```
let user = {};  
  
// 设置  
user["likes birds"] = true;  
  
// 读取  
alert(user["likes birds"]); // true  
  
// 删除  
delete user["likes birds"];
```

现在一切都可行了。请注意方括号中的字符串要放在引号中，单引号或双引号都可以。

方括号同样提供了一种可以通过任意表达式来获取属性名的方法——跟语义上的字符串不同——比如像类似于下面的变量:

```
let key = "likes birds";  
  
// 跟 user["likes birds"] = true; 一样  
user[key] = true;
```

在这里，变量 `key` 可以是程序运行时计算得到的，也可以是根据用户的输入得到的。然后我们可以用它来访问属性。这给了我们很大的灵活性。

例如:

```
let user = {
  name: "John",
  age: 30
};

let key = prompt("What do you want to know about the user?", "name");

// 访问变量
alert( user[key] ); // John (如果输入 "name")
```

点符号不能以类似的方式使用:

```
let user = {
  name: "John",
  age: 30
};

let key = "name";
alert( user.key ) // undefined
```

计算属性

我们可以在对象字面量中使用方括号。这叫做 **计算属性**。

例如:

```
let fruit = prompt("Which fruit to buy?", "apple");

let bag = {
  [fruit]: 5, // 属性名是从 fruit 变量中得到的
};

alert( bag.apple ); // 5 如果 fruit="apple"
```

计算属性的含义很简单: `[fruit]` 含义是属性名应该从 `fruit` 变量中获取。

所以, 如果一个用户输入 `"apple"`, `bag` 将变为 `{apple: 5}`。

本质上, 这跟下面的语法效果相同:

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {};

// 从 fruit 变量中获取值
bag[fruit] = 5;
```

.....但是看起来更好。

我们可以在方括号中使用更复杂的表达式:

```
let fruit = 'apple';
let bag = {
```

```
[fruit + 'Computers']: 5 // bag.appleComputers = 5  
};
```

方括号比点符号更强大。它允许任何属性名和变量，但写起来也更加麻烦。

所以大部分时间里，当属性名是已知且简单的时候，就是用点符号。如果我们需要一些更复杂的内容，那么就用方括号。

属性值简写

在实际开发中，我们通常用已存在的变量当做属性名。

例如：

```
function makeUser(name, age) {  
  return {  
    name: name,  
    age: age,  
    // .....其他的属性  
  };  
}  
  
let user = makeUser("John", 30);  
alert(user.name); // John
```

在上面的例子中，属性名跟变量名一样。这种通过变量生成属性的应用场景很常见，在这有一种特殊的 **属性值缩写** 方法，使属性名变得更短。

可以用 `name` 来代替 `name:name` 像下面那样：

```
function makeUser(name, age) {  
  return {  
    name, // 与 name: name 相同  
    age, // 与 age: age 相同  
    // ...  
  };  
}
```

我们可以把属性名简写方式和正常方式混用：

```
let user = {  
  name, // 与 name:name 相同  
  age: 30  
};
```

属性名称限制

属性名（key）必须是字符串或 **Symbol**（标识符的一种特殊类型，稍后将介绍）。

其它类型将被自动地转化为字符串。

例如当我们使用数字 `0` 作为属性 `key` 时，它将被转化为字符串 `"0"`：

```
let obj = {
  0: "test" // 和 "0": "test" 相同
};

// 两个 alert 访问的是同一个属性（数字 `0` 被转化为了字符串 "0"）
alert( obj["0"] ); // test
alert( obj[0] ); // test (同一个属性)
```

保留字段可以被用作属性名。

正如我们所知道的，像 `“for”`、`“let”` 和 `“return”` 等保留字段不能用作变量名。

但是对于对象的属性，没有这些限制。任何名字都可以：

```
let obj = {
  for: 1,
  let: 2,
  return: 3
}

alert( obj.for + obj.let + obj.return ); // 6
```

我们可以将任意字符串作为属性键（key），只有一个特殊的：`__proto__` 因为历史原因要特别对待。

比如，我们不能把它设置为非对象的值：

```
let obj = {};
obj.__proto__ = 5; // 分配一个数字
alert(obj.__proto__); // [object Object] – 值为对象，与预期结果不同
```

我们从代码中可以看出来，把它赋值为 `5` 的操作被忽略了。

关于 `__proto__` 的详细信息将在稍后的 [原型继承](#) 一章中详细介绍。

到目前为止，重要的是要知道，如果我们打算将用户提供的密钥存储在对象中，则 `__proto__` 的这种行为可能会成为错误甚至漏洞的来源。

因为用户可能会选择 `__proto__` 作为键，这个赋值的逻辑就失败了（像上面那样）。

有两个解决该问题的方法：

1. 修改对象的行为，使其将 `__proto__` 作为常规属性对待。我们将在 [原型方法，没有 __proto__ 的对象](#) 一章中学习如何进行修改。
2. 使用支持任意键的数据结构 `Map`。我们将在 [Map and Set \(映射和集合\)](#) 章节学习它。

属性存在性测试，“in” 操作符

对象的一个显著的特点就是其所有的属性都是可访问的。如果某个属性不存在也不会报错！访问一个不存在的属性只是会返回 `undefined`。这提供了一种普遍的用于检查属性是否存在方法

—— 获取值来与 `undefined` 比较:

```
let user = {};  
  
alert( user.noSuchProperty === undefined ); // true 意思是没有这个属性
```

这里同样也有一个特别的操作符 `"in"` 来检查属性是否存在。

语法是:

```
"key" in object
```

例如:

```
let user = { name: "John", age: 30 };  
  
alert( "age" in user ); // true, user.age 存在  
alert( "blabla" in user ); // false, user.blabla 不存在。
```

请注意，`in` 的左边必须是 **属性名**。通常是一个带引号的字符串。

如果我们省略引号，则意味着将测试包含实际名称的变量。例如:

```
let user = { age: 30 };  
  
let key = "age";  
alert( key in user ); // true, 从 key 获得属性名并检查这个属性
```

i 对存储值为 `undefined` 的属性使用 “in”

通常，检查属性是否存在时，使用严格比较 `"== undefined"` 就够了。但在一种特殊情况下，这种方式会失败，而 `"in"` 却可以正常工作。

那就是属性存在，但是存储值为 `undefined`:

```
let obj = {  
  test: undefined  
};  
  
alert( obj.test ); // 显示 undefined, 所以属性不存在?  
  
alert( "test" in obj ); // true, 属性存在!
```

在上面的代码中，属性 `obj.test` 事实上是存在的，所以 `in` 操作符检查通过。

这种情况很少发生，因为通常情况下是不会给对象赋值 `undefined` 的，我们经常会用 `null` 来表示未知的或者空的值。

“for...in” 循环

为了遍历一个对象的所有键（key），可以使用一个特殊形式的循环：`for..in`。这跟我们在前面学到的 `for(;;)` 循环是完全不一样的东西。

语法：

```
for (key in object) {  
    // 对此对象属性中的每个键执行的代码  
}
```

例如，让我们列出 `user` 所有的属性：

```
let user = {  
    name: "John",  
    age: 30,  
    isAdmin: true  
};  
  
for (let key in user) {  
    // keys  
    alert(key); // name, age, isAdmin  
    // 属性键的值  
    alert(user[key]); // John, 30, true  
}
```

注意，所有的“for”结构体都允许我们在循环中定义变量，像这里的 `let key`。

同样，我们可以用其他属性名来替代 `key`。例如 `"for(let prop in obj)"` 也很常用。

像对象一样排序

对象有顺序吗？换句话说，如果我们遍历一个对象，我们获取属性的顺序是和属性添加时的顺序相同吗？这靠谱吗？

简短的回答是：“有特别的顺序”：整数属性会被进行排序，其他属性则按照创建的顺序显示。详情如下：

例如，让我们考虑一个带有电话号码的对象：

```
let codes = {  
    "49": "Germany",  
    "41": "Switzerland",  
    "44": "Great Britain",  
    // ...  
    "1": "USA"  
};  
  
for(let code in codes) {  
    alert(code); // 1, 41, 44, 49  
}
```

对象可用于面向用户的建议选项列表。如果我们的网站主要面向德国观众，那么我们可能希望 `49` 排在第一。

但如果我们执行代码，会看到完全不同的景象：

- USA (1) 排在了最前面
- 然后是 Switzerland (41) 及其它。

因为这些电话号码是整数，所以它们以升序排列。所以我们看到的是 **1, 41, 44, 49**。

i 整数属性？那是什么？

这里的“整数属性”指的是一个可以在不作任何更改的情况下转换为整数的字符串（包括整数到整数）。

所以，“49”是一个整数属性名，因为我们把它转换成整数，再转换回来，它还是一样。但是“+49”和“1.2”就不行了：

```
// Math.trunc 是内置的去除小数部分的方法。
alert( String(Math.trunc(Number("49")))); // "49", 相同, 整数属性
alert( String(Math.trunc(Number("+49")))); // "49", 不同于 "+49" => 不是整数属性
alert( String(Math.trunc(Number("1.2")))); // "1", 不同于 "1.2" => 不是整数属性
```

.....此外，如果属性名不是整数，那它们就按照创建时候的顺序来排序，例如：

```
let user = {
  name: "John",
  surname: "Smith"
};
user.age = 25; // 增加一个

// 非整数属性是按照创建的顺序来排列的
for (let prop in user) {
  alert( prop ); // name, surname, age
}
```

所以，为了解决电话号码的问题，我们可以使用非整数属性名来 **欺骗** 程序。只需要给每个键名加一个加号 **“+”** 前缀就行了。

像这样：

```
let codes = {
  "+49": "Germany",
  "+41": "Switzerland",
  "+44": "Great Britain",
  // ...
  "+1": "USA"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

现在跟预想的一样了。

引用复制

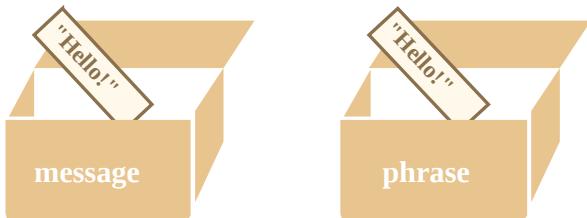
对象和其他原始类型的一个根本的区别是，对象都是“通过引用”存储和复制的。

原始类型：字符串，数字，布尔类型 — 作为整体值被赋值或复制。

例如：

```
let message = "Hello!";
let phrase = message;
```

结果是我们得到了两个独立变量，每个变量存的都是 "Hello!"。

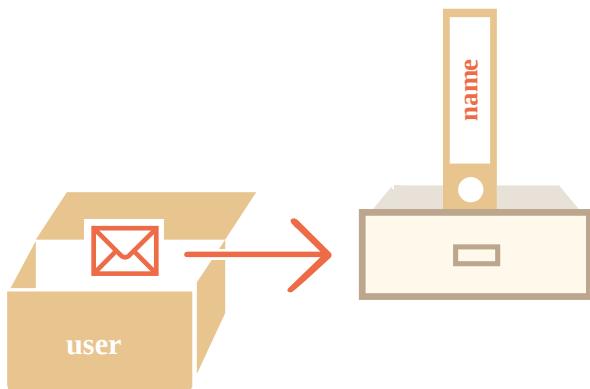


对象跟这个不一样。

变量存储的不是对象本身，而是“内存中的地址”，换句话说就是对象的“引用”。

下面是这个对象的存储结构图：

```
let user = {
  name: "John"
};
```



在这里，对象被存储在内存中的某个位置。变量 `user` 有一个对它的引用。

当对象被复制的时候 — 引用被复制了一份，对象并没有被复制。

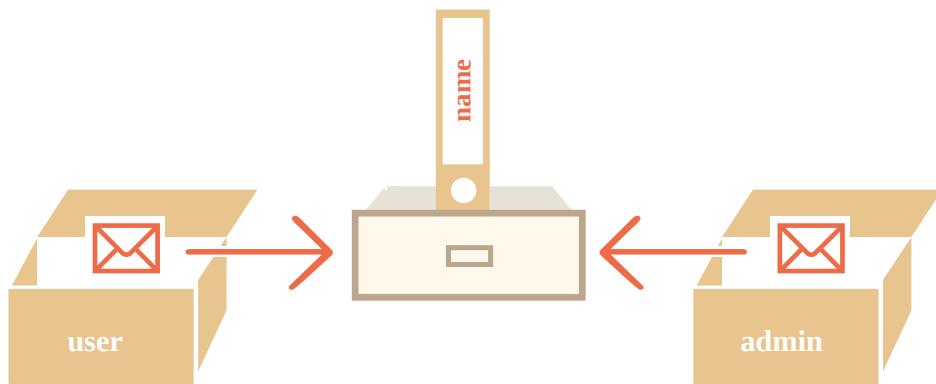
如果我们将对象想象成是一个抽屉，那么变量就是一把钥匙。拷贝对象是复制了钥匙，但是并没有复制抽屉本身。

例如：

```
let user = { name: "John" };

let admin = user; // 复制引用
```

现在我们有了两个变量，但是都指向同一个对象：



我们可以通过其中任意一个变量访问抽屉并改变其中的内容：

```
let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // 被通过名为 "admin" 的引用修改了

alert(user.name); // 'Pete', 通过名为 "user" 的引用查看修改
```

上面的例子证实了只存在一个对象。就像我们的一个抽屉带有两把钥匙，如果使用其中一把钥匙（`admin`）打开抽屉并改变抽屉里放的东西，稍后使用另外一把钥匙（`user`）打开抽屉的时候，就会看到变化。

比较引用

等号 `==` 和严格相等 `===` 运算符对于对象来说没差别。

两个对象只有在它们其实是一个对象时才会相等。

例如，如果两个变量引用指向同一个对象，那么它们相等：

```
let a = {};
let b = a; // 复制引用

alert( a == b ); // true, 两个变量指向同一个对象
alert( a === b ); // true
```

以下两个独立的对象不相等，即使都是空对象。

```
let a = {};
let b = {};// 两个独立的对象

alert( a == b ); // false
```

对于像 `obj1 > obj2` 这样两个对象的比较，或对象与原始值的比较 `obj == 5`，对象会被转换成原始值。我们很快就会学习到对象的转化是如何实现的，但是事实上，这种比较真的极少用到，这种比较的出现经常是代码的 **BUG** 导致的。

常量对象

一个被 `const` 修饰的对象是 **可以** 被修改。

例如：

```
const user = {  
    name: "John"  
};  
  
user.age = 25; // (*)  
  
alert(user.age); // 25
```

看起来好像 `(*)` 这行代码会导致错误，但并没有，这里完全没问题。这是因为 `const` 修饰的只是 `user` 本身存储的值。在这里 `user` 始终存储的都是对同一个对象的引用。`(*)` 这行代码修改的是对象内部的内容，并没有改变 `user` 存储的对象的引用。

如果你想把其他内容赋值给 `user`，那就会报错了，例如：

```
const user = {  
    name: "John"  
};  
  
// 错误（不能再给 user 赋值）  
user = {  
    name: "Pete"  
};
```

.....那么如果我们想要创建不可变的对象属性，应该怎么做呢？想让 `user.age = 25` 这样的赋值报错，这也是可以的。我们会在 [属性标志和属性描述符](#) 这章学习这部分内容。

复制和合并，`Object.assign`

复制一个对象变量会创建指向此对象的另一个引用。

那如果我们需要复制一个对象呢？创建一份独立的拷贝，一份克隆？

这也是可行的，但是有一点麻烦，因为 JavaScript 中没有支持这种操作的内置函数。实际上，我们很少这么做。在大多数时候，复制引用都很好用。

但如果我们真想这么做，就需要创建一个新的对象，然后遍历现有对象的属性，在原始级别的状态下复制给新的对象。

像这样：

```
let user = {  
    name: "John",
```

```
age: 30
};

let clone = {} // 新的空对象

// 复制所有的属性值
for (let key in user) {
  clone[key] = user[key];
}

// 现在的复制是独立的了
clone.name = "Pete"; // 改变它的值

alert( user.name ); // 原对象属性值不变
```

我们也可以用 `Object.assign ↗` 来实现。

语法是:

```
Object.assign(dest, [ src1, src2, src3... ])
```

- 参数 `dest` 和 `src1, ..., srcN` (你需要多少就可以设置多少, 没有限制) 是对象。
- 这个方法将 `src1, ..., srcN` 这些所有的对象复制到 `dest`。换句话说, 从第二个参数开始, 所有对象的属性都复制给了第一个参数对象, 然后返回 `dest`。

例如, 我们可以用这个方法来把几个对象合并成一个:

```
let user = { name: "John" };

let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

// 把 permissions1 和 permissions2 的所有属性都拷贝给 user
Object.assign(user, permissions1, permissions2);

// 现在 user = { name: "John", canView: true, canEdit: true }
```

如果用于接收的对象 (`user`) 已经有了同样属性名的属性, 已有的则会被覆盖:

```
let user = { name: "John" };

// 覆盖 name, 增加 isAdmin
Object.assign(user, { name: "Pete", isAdmin: true });

// 现在 user = { name: "Pete", isAdmin: true }
```

我们可以用 `Object.assign` 来替代循环赋值进行简单的克隆操作:

```
let user = {
  name: "John",
```

```
    age: 30
};

let clone = Object.assign({}, user);
```

它将对象 `user` 的所有的属性复制给了一个空对象并返回。实际上和循环赋值没什么区别，只是更短了。

直到现在，我们都是假设 `user` 的所有属性都是原始值。但是属性也可以是其他对象的引用。这种我们应该怎么操作呢？

例如：

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

alert( user.sizes.height ); // 182
```

现在，仅仅进行 `clone.sizes = user.sizes` 复制是不够的，因为 `user.sizes` 是一个对象，这个操作只能复制这个对象的引用。所以 `clone` 和 `user` 共享了一个对象。

像这样：

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes ); // true, 同一个对象

// user 和 clone 共享 sizes 对象
user.sizes.width++;           // 在这里改变一个属性的值
alert(clone.sizes.width); // 51, 在这里查看属性的值
```

为了解决这个问题，我们在复制的时候应该检查 `user[key]` 的每一个值，如果它是一个对象，那么把它也复制一遍，这叫做深拷贝（deep cloning）。

有一个标准的深拷贝算法，用于解决上面这种和一些更复杂的情况，叫做 [结构化克隆算法（Structured cloning algorithm）](#)。为了不重复造轮子，我们可以使用它的一个 JavaScript 实现的库 `lodash`，方法名叫做 `_cloneDeep(obj)`。

总结

对象是具有一些特殊特性的关联数组。

它们存储属性（键值对），其中：

- 属性的键必须是字符串或者 `symbol`（通常是字符串）。
- 值可以是任何类型。

我们可以用下面的方法访问属性：

- 点符号：`obj.property`。
- 方括号 `obj["property"]`，方括号允许从变量中获取键，例如 `obj[varWithKey]`。

其他操作：

- 删除属性：`delete obj.prop`。
- 检查是否存在给定键的属性：`"key" in obj`。
- 遍历对象：`for(let key in obj)` 循环。

对象是通过引用被赋值或复制的。换句话说，变量存储的不是“对象的值”，而是值的“引用”（内存地址）。所以复制这样的变量或者将其作为函数参数进行传递时，复制的是引用，而不是对象。基于复制的引用（例如添加/删除属性）执行的所有操作，都是在同一个对象上执行的。

我们可以使用 `Object.assign` 或者 `_.cloneDeep(obj)` 进行“真正的复制”（一个克隆）。

我们在这一章学习的叫做“基本对象”，或者就叫对象。

JavaScript 中还有很多其他类型的对象：

- `Array` 用于存储有序数据集合，
- `Date` 用于存储时间日期，
- `Error` 用于存储错误信息。
-等等。

它们有着各自特别的特性，我们将在后面学习到。有时候大家会说“数组类型”或“日期类型”，但其实它们并不是自身所属的类型，而是属于一个对象类型即“`object`”。它们以不同的方式对“`object`”做了一些扩展。

JavaScript 中的对象非常强大。这里我们只接触了冰山一角。在后面的章节中，我们将频繁使用对象进行编程，并学习更多关于对象的知识。

垃圾回收

对于开发者来说，JavaScript 的内存管理是自动的、无形的。我们创建的原始值、对象、函数……这一切都会占用内存。

当我们不再需要某个东西时会发生什么？JavaScript 引擎如何发现它并清理它？

可达性（Reachability）

JavaScript 中主要的内存管理概念是 可达性。

简而言之，“可达”值是那些以某种方式可访问或可用的值。它们一定是存储在内存中的。

1. 这里列出固有的可达值的基本集合，这些值明显不能被释放。

比方说：

- 当前函数的局部变量和参数。
- 嵌套调用时，当前调用链上所有函数的变量与参数。
- 全局变量。
- （还有一些内部的）

这些值被称作 **根（roots）**。

2. 如果一个值可以通过引用或引用链从根访问任何其他值，则认为该值是可达的。

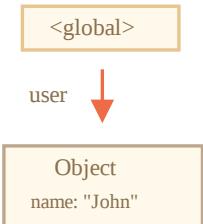
比方说，如果局部变量中有一个对象，并且该对象有一个属性引用了另一个对象，则该对象被认为是可达的。而且它引用的内容也是可达的。下面是详细的例子。

在 JavaScript 引擎中有一个被称作 [垃圾回收器](#) 的东西在后台执行。它监控着所有对象的状态，并删除掉那些已经不可达的。

一个简单的例子

这里是一个最简单的例子：

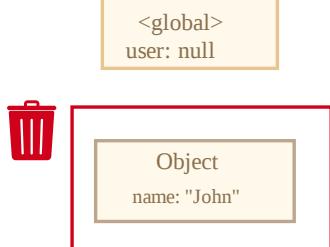
```
// user 具有对这个对象的引用
let user = {
  name: "John"
};
```



这里的箭头描述了一个对象引用。全局变量 `"user"` 引用了对象 `{name: "John"}`（为简洁起见，我们称它为 **John**）。**John** 的 `"name"` 属性存储一个原始值，所以它被写在对象内部。

如果 `user` 的值被重写了，这个引用就没了：

```
user = null;
```



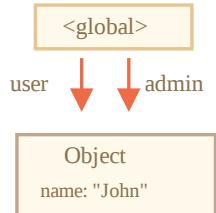
现在 **John** 变成不可达的了。因为没有引用了，就不能访问到它了。垃圾回收器会认为它是垃圾数据并进行回收，然后释放内存。

两个引用

现在让我们想象下，我们把 `user` 的引用复制给 `admin`：

```
// user 具有对这个对象的引用
let user = {
  name: "John"
};

let admin = user;
```



现在如果执行刚刚的那个操作：

```
user = null;
```

.....然后对象仍然可以被通过 `admin` 这个全局变量访问到，所以对象还在内存中。如果我们又重写了 `admin`，对象就会被删除。

相互关联的对象

现在来看一个更复杂的例子。这是个家庭：

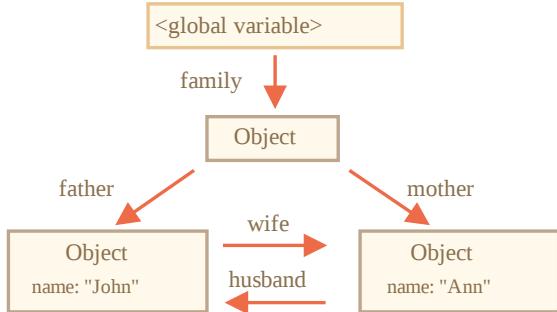
```
function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;

  return {
    father: man,
    mother: woman
  }
}

let family = marry({
  name: "John"
}, {
  name: "Ann"
});
```

`marry` 函数通过让两个对象相互引用使它们“结婚”了，并返回了一个包含这两个对象的新对象。

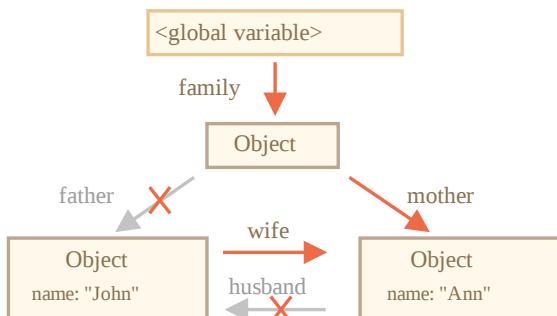
由此产生的内存结构：



到目前为止，所有对象都是可达的。

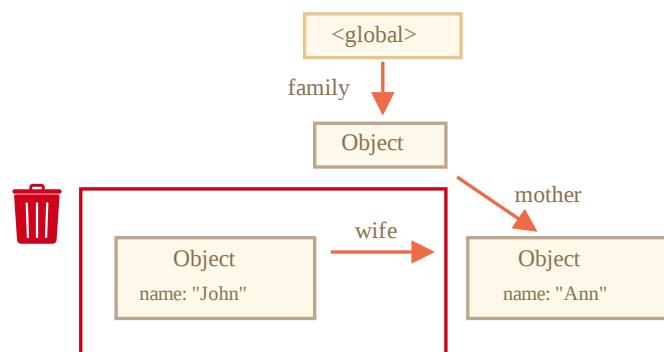
现在让我们移除两个引用：

```
delete family.father;
delete family.mother.husband;
```



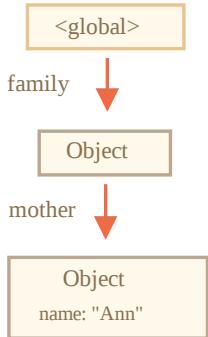
仅删除这两个引用中的一个时是不够的，因为所有的对象仍然都是可达的。

但是，如果我们把这两个都删除，那么我们可以看到再也没有对 John 的引用了：



对外引用不重要，只有传入引用才可以使对象可达。所以，John 现在是不可达的，并且将被从内存中删除，同时 John 的所有数据也将变得不可达。

经过垃圾回收：



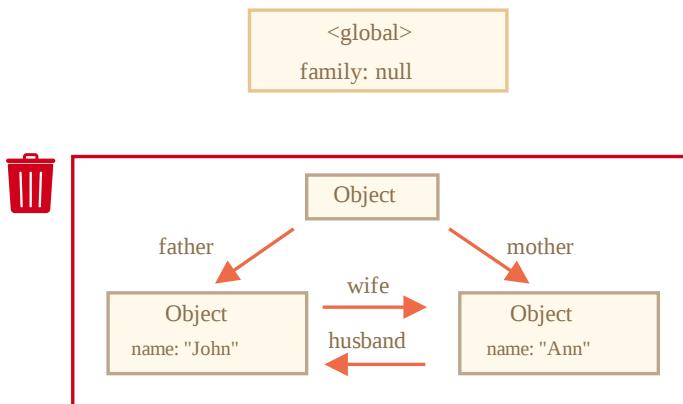
无法到达的岛屿

几个对象相互引用，但外部没有对其任意对象的引用，这些对象也可能是不可达的，并被从内存中删除。

源对象与上面相同。然后：

```
family = null;
```

内存内部状态将变成：



这个例子展示了可达性概念的重要性。

显而易见，John 和 Ann 仍然连着，都有传入的引用。但是，这样还不够。

前面说的 "family" 对象已经不再与根相连，没有了外部对它的引用，所以它变成了一座“孤岛”，并且将被从内存中删除。

内部算法

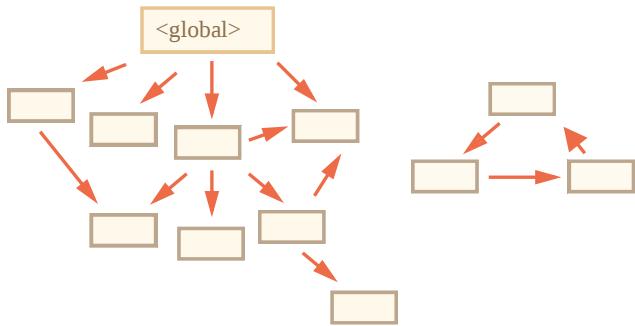
垃圾回收的基本算法被称为“mark-and-sweep”。

定期执行以下“垃圾回收”步骤：

- 垃圾收集器找到所有的根，并“标记”（记住）它们。
- 然后它遍历并“标记”来自它们的所有引用。
- 然后它遍历标记的对象并标记 **他们的** 引用。所有被遍历到的对象都会被记住，以免将来再次遍历到同一个对象。

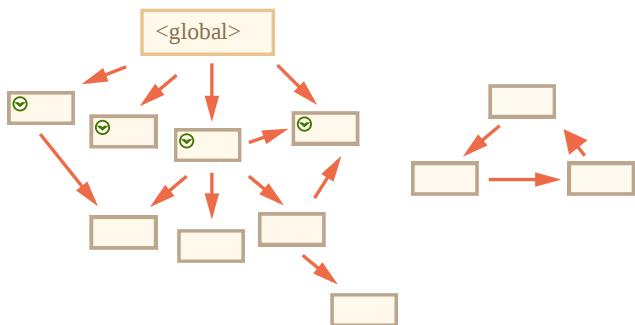
-如此操作，直到所有可达的（从根部）引用都被访问到。
- 没有被标记的对象都会被删除。

例如，使我们的对象有如下的结构:

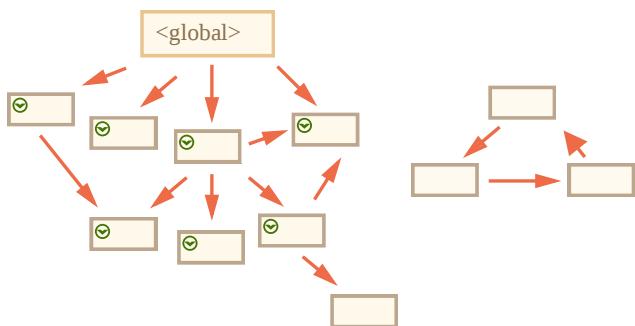


我们可以清楚地看到右侧有一个“无法到达的岛屿”。现在我们来看看“标记和清除”垃圾收集器如何处理它。

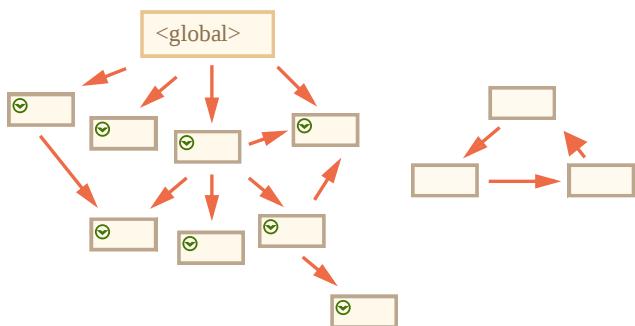
第一步标记所有的根:



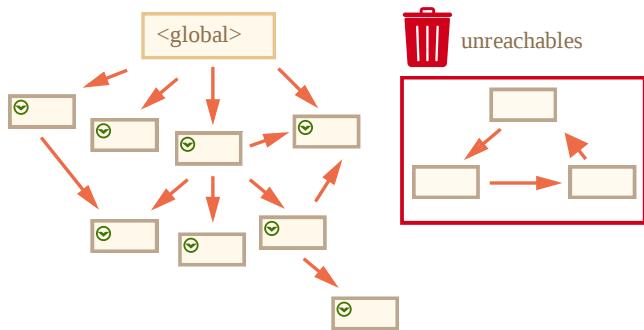
然后他们的引用被标记了:



.....如果还有引用的话，继续标记:



现在，无法通过这个过程访问到的对象被认为是不可达的，并且会被删除。



我们还可以将这个过程想象成从根溢出一个巨大的油漆桶，它流经所有引用并标记所有可到达的对象。然后移除未标记的。

这是垃圾收集工作的概念。JavaScript 引擎做了许多优化，使垃圾回收运行速度更快，并且不影响正常代码运行。

一些优化建议：

- **分代收集（Generational collection）** —— 对象被分成两组：“新的”和“旧的”。许多对象出现，完成他们的工作并很快死去，他们可以很快被清理。那些长期存活的对象会变得“老旧”，而且被检查的频次也会减少。
- **增量收集（Incremental collection）** —— 如果有许多对象，并且我们试图一次遍历并标记整个对象集，则可能需要一些时间，并在执行过程中带来明显的延迟。所以引擎试图将垃圾收集工作分成几部分来做。然后将这几部分会逐一进行处理。这需要他们之间有额外的标记来追踪变化，但是这样会有许多微小的延迟而不是一个大的延迟。
- **闲时收集（Idle-time collection）** —— 垃圾收集器只会在 CPU 空闲时尝试运行，以减少可能对代码执行的影响。

还有其他垃圾回收算法的优化和风格。尽管我想在这里描述它们，但我必须打住了，因为不同的引擎会有不同的调整和技巧。而且，更重要的是，随着引擎的发展，情况会发生变化，所以在没有真实需求的时候，“提前”学习这些内容是不值得的。当然，除非这是一个纯粹的利益关系。我在下面给你提供了一些相关链接。

总结

主要需要掌握的内容：

- 垃圾回收是自动完成的，我们不能强制执行或是阻止执行。
- 当对象是可达状态时，它一定是存在于内存中的。
- 被引用与可访问（从一个根）不同：一组相互连接的对象可能整体都不可达。

现代引擎实现了垃圾回收的高级算法。

《The Garbage Collection Handbook: The Art of Automatic Memory Management》 (R. Jones 等人著) 这本书涵盖了其中一些内容。

如果你熟悉底层（low-level）编程，关于 V8 引擎垃圾回收器的更详细信息请参阅文章 [V8 之旅：垃圾回收](#)。

[V8 博客](#) 还不时发布关于内存管理变化的文章。当然，为了学习垃圾收集，你最好通过学习 V8 引擎内部知识来进行准备，并阅读一个名为 [Vyacheslav Egorov](#) 的 V8 引擎工程师的博客。我

之所以说“V8”，因为网上关于它的文章最丰富的。对于其他引擎，许多方法是相似的，但在垃圾收集上许多方面有所不同。

当你需要底层的优化时，对引擎有深入了解将很有帮助。在熟悉了这门编程语言之后，把熟悉引擎作为下一步计划是明智之选。

Symbol 类型

根据规范，对象的属性键只能是字符串类型或者 **Symbol** 类型。不是 **Number**，也不是 **Boolean**，只有字符串或 **Symbol** 这两种类型。

到目前为止，我们只见过字符串。现在我们来看看 **Symbol** 能给我们带来什么好处。

Symbol

“**Symbol**” 值表示唯一的标识符。

可以使用 `Symbol()` 来创建这种类型的值：

```
// id 是 symbol 的一个实例化对象
let id = Symbol();
```

创建时，我们可以给 **Symbol** 一个描述（也称为 **Symbol** 名），这在代码调试时非常有用：

```
// id 是描述为 "id" 的 Symbol
let id = Symbol("id");
```

Symbol 保证是唯一的。即使我们创建了许多具有相同描述的 **Symbol**，它们的值也是不同。描述只是一个标签，不影响任何东西。

例如，这里有两个描述相同的 **Symbol** —— 它们不相等：

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

如果你熟悉 **Ruby** 或者其他有“**Symbol**”的语言 —— 别被误导。**JavaScript** 的 **Symbol** 是不同的。

⚠️ Symbol 不会被自动转换为字符串

JavaScript 中的大多数值都支持字符串的隐式转换。例如，我们可以 `alert` 任何值，都可以生效。`Symbol` 比较特殊，它不会被自动转换。

例如，这个 `alert` 将会提示出错：

```
let id = Symbol("id");
alert(id); // 类型错误：无法将 Symbol 值转换为字符串。
```

这是一种防止混乱的“语言保护”，因为字符串和 `Symbol` 有本质上的不同，不应该意外地将它们转换成另一个。

如果我们真的想显示一个 `Symbol`，我们需要在它上面调用 `.toString()`，如下所示：

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id)，现在它有效了
```

或者获取 `symbol.description` 属性，只显示描述（`description`）：

```
let id = Symbol("id");
alert(id.description); // id
```

“隐藏”属性

`Symbol` 允许我们创建对象的“隐藏”属性，代码的任何其他部分都不能意外访问或重写这些属性。

例如，如果我们使用的是属于第三方代码的 `user` 对象，我们想要给它们添加一些标识符。

我们可以给它们使用 `Symbol` 键：

```
let user = { // 属于另一个代码
  name: "John"
};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // 我们可以使用 Symbol 作为键来访问数据
```

在字符串 `"id"` 上使用 `Symbol("id")` 有什么好处？

因为 `user` 属于另一个代码，另一个代码也使用它执行一些操作，所以我们不应该在它上面加任何字段，这样很不安全。但是 `Symbol` 不会被意外访问到，所以第三方代码看不到它，所以使用 `Symbol` 也许不会有什么问题。

另外，假设另一个脚本希望在 `user` 中有自己的标识符，以实现自己的目的。这可能是另一个 JavaScript 库，因此脚本之间完全不了解彼此。

然后该脚本可以创建自己的 `Symbol("id")`，像这样：

```
// ...
let id = Symbol("id");

user[id] = "Their id value";
```

我们的标识符和他们的标识符之间不会有冲突，因为 `Symbol` 总是不同的，即使它们有相同的名字。

.....但如果我们处于同样的目的，使用字符串 `"id"` 而不是用 `symbol`，那么 **就会** 出现冲突：

```
let user = { name: "John" };

// 我们的脚本使用了 "id" 属性。
user.id = "Our id value";

// .....另一个脚本也想将 "id" 用于它的目的......

user.id = "Their id value"
// 碰！无意中被另一个脚本重写了 id!
```

字面量中的 `Symbol`

如果我们要在对象字面量 `{...}` 中使用 `Symbol`，则需要使用方括号把它括起来。

就像这样：

```
let id = Symbol("id");

let user = {
  name: "John",
  [id]: 123 // 而不是 "id: 123"
};
```

这是因为我们需要变量 `id` 的值作为键，而不是字符串 `"id"`。

`Symbol` 在 `for...in` 中会被跳过

`Symbol` 属性不参与 `for..in` 循环。

例如：

```
let id = Symbol("id");
let user = {
  name: "John",
  age: 30,
  [id]: 123
};

for (let key in user) alert(key); // name, age (no symbols)

// 使用 Symbol 任务直接访问
alert( "Direct: " + user[id] );
```

`Object.keys(user)` 也会忽略它们。这是一般“隐藏符号属性”原则的一部分。如果另一个脚本或库遍历我们的对象，它不会意外地访问到符号属性。

相反，`Object.assign` 会同时复制字符串和 `symbol` 属性：

```
let id = Symbol("id");
let user = {
  [id]: 123
};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123
```

这里并不矛盾，就是这样设计的。这里的看法是当我们克隆或者合并一个 `object` 时，通常希望 **所有** 属性被复制（包括像 `id` 这样的 `Symbol`）。

全局 `symbol`

正如我们所看到的，通常所有的 `Symbol` 都是不同的，即使它们有相同的名字。但有时我们想要名字相同的 `Symbol` 具有相同的实体。例如，应用程序的不同部分想要访问的 `Symbol "id"` 指的是完全相同的属性。

为了实现这一点，这里有一个 **全局 `Symbol` 注册表**。我们可以在其中创建 `Symbol` 并在稍后访问它们，它可以确保每次访问相同名字的 `Symbol` 时，返回的都是相同的 `Symbol`。

要从注册表中读取（不存在则创建）`Symbol`，请使用 `Symbol.for(key)`。

该调用会检查全局注册表，如果有一个描述为 `key` 的 `Symbol`，则返回该 `Symbol`，否则将创建一个新 `Symbol` (`Symbol(key)`)，并通过给定的 `key` 将其存储在注册表中。

例如：

```
// 从全局注册表中读取
let id = Symbol.for("id"); // 如果该 Symbol 不存在，则创建它

// 再次读取（可能是在代码中的另一个位置）
let idAgain = Symbol.for("id");

// 相同的 Symbol
alert( id === idAgain ); // true
```

注册表内的 `Symbol` 被称为 **全局 `Symbol`**。如果我们想要一个应用程序范围内的 `Symbol`，可以在代码中随处访问——这就是它们的用途。

ⓘ 这听起来像 Ruby

在一些编程语言中，例如 `Ruby`，每个名字都有一个 `Symbol`。

正如我们所看到的，在 `JavaScript` 中，全局 `Symbol` 也是这样的。

`Symbol.keyFor`

对于全局 `Symbol`, 不仅有 `Symbol.for(key)` 按名字返回一个 `Symbol`, 还有一个反向调用: `Symbol.keyFor(sym)`, 它的作用完全反过来: 通过全局 `Symbol` 返回一个名字。

例如:

```
// 通过 name 获取 Symbol
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// 通过 Symbol 获取 name
alert( Symbol.keyFor(sym) ); // name
alert( Symbol.keyFor(sym2) ); // id
```

`Symbol.keyFor` 内部使用全局 `Symbol` 注册表来查找 `Symbol` 的键。所以它不适用于非全局 `Symbol`。如果 `Symbol` 不是全局的, 它将无法找到它并返回 `undefined`。

也就是说, 任何 `Symbol` 都具有 `description` 属性。

例如:

```
let globalSymbol = Symbol.for("name");
let localSymbol = Symbol("name");

alert( Symbol.keyFor(globalSymbol) ); // name, 全局 Symbol
alert( Symbol.keyFor(localSymbol) ); // undefined, 非全局

alert( localSymbol.description ); // name
```

系统 `Symbol`

JavaScript 内部有很多“系统” `Symbol`, 我们可以使用它们来微调对象的各个方面。

它们都被列在了 [众所周知的 `Symbol`](#) 表的规范中:

- `Symbol.hasInstance`
- `Symbol.isConcatSpreadable`
- `Symbol.iterator`
- `Symbol.toPrimitive`
-等等。

例如, `Symbol.toPrimitive` 允许我们将对象描述为原始值转换。我们很快就会看到它的使用。

当我们研究相应的语言特征时, 我们对其他的 `Symbol` 也会慢慢熟悉起来。

总结

`Symbol` 是唯一标识符的基本类型

`Symbol` 是使用带有可选描述 (`name`) 的 `Symbol()` 调用创建的。

`Symbol` 总是不同的值，即使它们有相同的名字。如果我们希望同名的 `Symbol` 相等，那么我们应该使用全局注册表：`Symbol.for(key)` 返回（如果需要的话则创建）一个以 `key` 作为名字的全局 `Symbol`。使用 `Symbol.for` 多次调用 `key` 相同的 `Symbol` 时，返回的就是同一个 `Symbol`。

`Symbol` 有两个主要的使用场景：

1. “隐藏” 对象属性。如果我们想要向“属于”另一个脚本或者库的对象添加一个属性，我们可以创建一个 `Symbol` 并使用它作为属性的键。`Symbol` 属性不会出现在 `for..in` 中，因此它不会意外地被与其他属性一起处理。并且，它不会被直接访问，因为另一个脚本没有我们的 `symbol`。因此，该属性将受到保护，防止被意外使用或重写。

因此我们可以使用 `Symbol` 属性“秘密地”将一些东西隐藏到我们需要的对象中，但其他地方看不到它。

2. JavaScript 使用了许多系统 `Symbol`，这些 `Symbol` 可以作为 `Symbol.*` 访问。我们可以使用它们来改变一些内置行为。例如，在本教程的后面部分，我们将使用 `Symbol.iterator` 来进行 `迭代` 操作，使用 `Symbol.toPrimitive` 来设置 `对象原始值的转换` 等等。

从技术上说，`Symbol` 不是 100% 隐藏的。有一个内置方法

`Object.getOwnPropertySymbols(obj)` 允许我们获取所有的 `Symbol`。还有一个名为 `Reflect.ownKeys(obj)` 的方法可以返回一个对象的 `所有` 键，包括 `Symbol`。所以它们并不是真正的隐藏。但是大多数库、内置方法和语法结构都没有使用这些方法。

对象方法，“this”

通常创建对象来表示真实世界中的实体，如用户和订单等：

```
let user = {
  name: "John",
  age: 30
};
```

并且，在现实世界中，用户可以进行 **操作**：从购物车中挑选某物、登录和注销等。

在 JavaScript 中，行为（action）由属性中的函数来表示。

方法示例

刚开始，我们来教 `user` 说 hello：

```
let user = {
  name: "John",
  age: 30
};

user.sayHi = function() {
  alert("Hello!");
};

user.sayHi(); // Hello!
```

这里我们使用函数表达式创建了一个函数，并将其指定给对象的 `user.sayHi` 属性。

随后我们调用它。用户现在可以说话了！

作为对象属性的函数被称为 **方法**。

所以，在这我们得到了 `user` 对象的 `sayHi` 方法。

当然，我们也可以使用预先声明的函数作为方法，就像这样：

```
let user = {  
  // ...  
};  
  
// 首先，声明函数  
function sayHi() {  
  alert("Hello!");  
}  
  
// 然后将其作为一个方法添加  
user.sayHi = sayHi;  
  
user.sayHi(); // Hello!
```

① 面向对象编程

当我们在代码中用对象表示实体时，就是所谓的 [面向对象编程](#)，简称为“OOP”。

OOP 是一门大学问，本身就是一门有趣的科学。怎样选择合适的实体？如何组织它们之间的交互？这就是架构，有很多关于这方面的书，例如 E. Gamma、R. Helm、R. Johnson 和 J. Vissides 所著的《设计模式：可复用面向对象软件的基础》，G. Booch 所著的《面向对象分析与设计》等。

方法简写

在对象字面量中，有一种更短的（声明）方法的语法：

```
// 这些对象作用一样  
  
user = {  
  sayHi: function() {  
    alert("Hello");  
  }  
};  
  
// 方法简写看起来更好，对吧?  
let user = {  
  sayHi() { // 与 "sayHi: function()" 一样  
    alert("Hello");  
  }  
};
```

如上所示，我们可以省略 `"function"`，只写 `sayHi()`。

说实话，这种表示法还是有些不同。在对象继承方面有一些细微的差别（稍后将会介绍），但目前它们并不重要。在几乎所有的情况下，较短的语法是首选的。

方法中的“this”

通常，对象方法需要访问对象中存储的信息才能完成其工作。

例如，`user.sayHi()` 中的代码可能需要用到 `user` 的 `name` 属性。

为了访问该对象，方法中可以使用 `this` 关键字。

`this` 的值就是在点之前的这个对象，即调用该方法的对象。

举个例子：

```
let user = {
  name: "John",
  age: 30,
  sayHi() {
    // "this" 指的是“当前的对象”
    alert(this.name);
  }
};

user.sayHi(); // John
```

在这里 `user.sayHi()` 执行过程中，`this` 的值是 `user`。

技术上讲，也可以在不使用 `this` 的情况下，通过外部变量名来引用它：

```
let user = {
  name: "John",
  age: 30,
  sayHi() {
    alert(user.name); // "user" 替代 "this"
  }
};
```

.....但这样的代码是不可靠的。如果我们决定将 `user` 复制给另一个变量，例如 `admin = user`，并赋另外的值给 `user`，那么它将访问到错误的对象。

下面这个示例证实了这一点：

```
let user = {
  name: "John",
  age: 30,
  sayHi() {
    alert( user.name ); // 导致错误
  }
};
```

```
let admin = user;
user = null; // 重写让其更明显

admin.sayHi(); // 噢哟！在 sayHi() 使用了旧的 name 属性！报错！
```

如果我们在 `alert` 中以 `this.name` 替换 `user.name`，那么代码就会正常运行。

“this”不受限制

在 JavaScript 中，`this` 关键字与其他大多数编程语言中的不同。JavaScript 中的 `this` 可以用于任何函数。

下面这样的代码没有语法错误：

```
function sayHi() {
  alert( this.name );
}
```

`this` 的值是在代码运行时计算出来的，它取决于代码上下文。

例如，这里相同的函数被分配给两个不同的对象，在调用中有着不同的“this”值：

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
  alert( this.name );
}

// 在两个对象中使用相同的函数
user.f = sayHi;
admin.f = sayHi;

// 这两个调用有不同的 this 值
// 函数内部的 "this" 是“点符号前面”的那个对象
user.f(); // John (this == user)
admin.f(); // Admin (this == admin)

admin['f'](); // Admin (使用点符号或方括号语法来访问这个方法，都没有关系。)
```

这个规则很简单：如果 `obj.f()` 被调用了，则 `this` 在 `f` 函数调用期间是 `obj`。所以在上面的例子中 `this` 先是 `user`，之后是 `admin`。

① 在没有对象的情况下调用: `this == undefined`

我们甚至可以在没有对象的情况下调用函数:

```
function sayHi() {  
  alert(this);  
}  
  
sayHi(); // undefined
```

在这种情况下，严格模式下的 `this` 值为 `undefined`。如果我们尝试访问 `this.name`，将会报错。

在非严格模式的情况下，`this` 将会是 **全局对象**（浏览器中的 `window`，我们稍后会在 [全局对象](#) 一章中学习它）。这是一个历史行为，`"use strict"` 已经将其修复了。

通常这种调用是程序出错了。如果在一个函数内部有 `this`，那么通常意味着它是在对象上下文环境中被调用的。

② 解除 `this` 绑定的后果

如果你经常使用其他的编程语言，那么你可能已经习惯了“绑定 `this`”的概念，即在对象中定义的方法总是有指向该对象的 `this`。

在 JavaScript 中，`this` 是“自由”的，它的值是在调用时计算出来的，它的值并不取决于方法声明的位置，而是取决于在“点符号前”的是什么对象。

在运行时对 `this` 求值的这个概念既有优点也有缺点。一方面，函数可以被重用于不同的对象。另一方面，更大的灵活性造成了更大的出错的可能。

这里我们的立场并不是要评判编程语言的这个设计是好是坏。而是要了解怎样使用它，如何趋利避害。

内部: 引用类型

⚠ 高阶语言特性

这一小节介绍了一个进阶主题，来更好地理解一些特殊情况。

如果你想学得更快，这部分你可以跳过或者过后来读。

“复杂”的方法调用可能会失去 `this`，例如:

```
let user = {  
  name: "John",  
  hi() { alert(this.name); },  
  bye() { alert("Bye"); }  
};  
  
user.hi(); // John (简单的调用工作正常)
```

```
// 现在我们要根据 name 来决定调用 user.hi 还是 user.bye。  
(user.name == "John" ? user.hi : user.bye)(); // 报错!
```

最后一行中有一个三元运算符，用来决定调用 `user.hi` 还是 `user.bye`。在这种情况下，结果会是 `user.hi`。

该方法立即被括号 `()` 调用。但它不能正常工作！

你可以看到该调用导致了错误，因为调用中的 `"this"` 为 `undefined`。

这样能正常工作（对象点方法）：

```
user.hi();
```

这样不行（对方法求值）：

```
(user.name == "John" ? user.hi : user.bye)(); // 报错!
```

为什么？如果我们想了解为什么会这样，那么我们要深入理解 `obj.method()` 的调用原理。

仔细看，我们可能注意到 `obj.method()` 语句中有两个操作符。

1. 首先，点符号 `'.'` 取得这个 `obj.method` 属性。
2. 其后的括号 `()` 调用它。

那么，`this` 是怎样被从第一部分传递到第二部分的呢？

如果把这些操作拆分开，那么 `this` 肯定会丢失：

```
let user = {  
  name: "John",  
  hi() { alert(this.name); }  
}
```

```
// 将赋值与方法调用拆分为两行  
let hi = user.hi;  
hi(); // 错误，因为 this 未定义
```

这里 `hi = user.hi` 把函数赋值给变量，其后的最后一行代码是完全独立的，所以它没有 `this`。

为了让 `user.hi()` 有效，JavaScript 用了一个技巧——这个 `'.'` 点符号返回的不是一个函数，而是一种特殊的引用类型 ↗ 的值。

引用类型是一种“规范中有的类型”。我们不能明确地指定它，但它被用在编程语言的内部。

引用类型的值是三部分的结合 `(base, name, strict)`，如下：

- `base` 是对象。
- `name` 是属性名。
- 在严格模式 `use strict` 下，`strict` 为真。

属性访问 `user.hi` 的结果不是函数，而是引用类型。在严格模式下的 `user.hi` 是：

```
// 引用类型值  
(user, "hi", true)
```

括号 `()` 调用引用类型时，将接收关于该对象及其方法的所有信息，并且可以设定正确的 `this` 值（这里等于 `user`）。

引用类型是一种特殊的“中间”内部类型，用于将信息从点符号 `.` 传递到调用括号 `()`。

像赋值 `hi = user.hi` 等其他的操作，将引用类型作为一个整体丢弃，只获取 `user.hi`（一个函数）的值进行传递。因此，任何进一步的操作都会“失去” `this`。

因此，结果是，只有使用点符号 `obj.method()` 或方括号语法 `obj[method]()`（它们在这里作用相同）调用函数时，`this` 的值才被正确传递（这里的例子也一样）。在本教程的后面，我们将学习解决此问题的各种方法，例如 `func.bind()`。

箭头函数没有自己的“this”

箭头函数有些特别：它们没有自己的 `this`。如果我们在这样的函数中引用 `this`，`this` 值取决于外部“正常的”函数。

举个例子，这里的 `arrow()` 使用的 `this` 来自于外部的 `user.sayHi()` 方法：

```
let user = {  
  firstName: "Ilya",  
  sayHi() {  
    let arrow = () => alert(this.firstName);  
    arrow();  
  }  
};  
  
user.sayHi(); // Ilya
```

这是箭头函数的一个特性，当我们并不想要一个独立的 `this`，反而想从外部上下文中获取时，它很有用。在后面的 [深入理解箭头函数](#) 一章中，我们将深入介绍箭头函数。

总结

- 存储在对象属性中的函数被称为“方法”。
- 方法允许对象进行像 `object.doSomething()` 这样的“操作”。
- 方法可以将对象引用为 `this`。

`this` 的值是在程序运行时得到的。

- 一个函数在声明时，可能就使用了 `this`，但是这个 `this` 只有在函数被调用时才会有值。
- 可以在对象之间复制函数。
- 以“方法”的语法调用函数时：`object.method()`，调用过程中的 `this` 值是 `object`。

请注意箭头函数有些特别：它们没有 `this`。在箭头函数内部访问到的 `this` 都是从外部获取的。

对象 — 原始值转换

当对象相加 `obj1 + obj2`，相减 `obj1 - obj2`，或者使用 `alert(obj)` 打印时会发生什么？

在这种情况下，对象会被自动转换为原始值，然后执行操作。

在 [类型转换](#) 一章中，我们已经看到了数值，字符串和布尔转换的规则。但是我们没有讲对象的转换规则。现在我们已经掌握了方法（method）和 symbol 的相关知识，可以开始学习对象原始值转换了。

1. 所有的对象在布尔上下文（context）中均为 `true`。所以对于对象，不存在 `to-boolean` 转换，只有字符串和数值转换。
2. 数值转换发生在对象相减或应用数学函数时。例如，`Date` 对象（将在 [日期和时间](#) 一章中介绍）可以相减，`date1 - date2` 的结果是两个日期之间的差值。
3. 至于字符串转换——通常发生在我们像 `alert(obj)` 这样输出一个对象和类似的上下文中。

ToPrimitive

我们可以使用特殊的对象方法，对字符串和数值转换进行微调。

下面是三个类型转换的变体，被称为“hint”，在 [规范](#) 中有详细介绍（译注：当一个对象被用在需要原始值的上下文中时，例如，在 `alert` 或数学运算中，对象会被转换为原始值）：

"string"

对象到字符串的转换，当我们对期望一个字符串的对象执行操作时，如“`alert`”：

```
// 输出
alert(obj);

// 将对象作为属性键
anotherObj[obj] = 123;
```

"number"

对象到数字的转换，例如当我们进行数学运算时：

```
// 显式转换
let num = Number(obj);

// 数学运算（除了二进制加法）
let n = +obj; // 一元加法
let delta = date1 - date2;

// 小于/大于的比较
let greater = user1 > user2;
```

"default"

在少数情况下发生，当运算符“不确定”期望值的类型时。

例如，二进制加法 `+` 可用于字符串（连接），也可以用于数字（相加），所以字符串和数字这两种类型都可以。因此，当二元加法得到对象类型的参数时，它将依据 `"default"` hint 来对其进行转换。

此外，如果对象被用于与字符串、数字或 `symbol` 进行 `==` 比较，这时到底应该进行哪种转换也不是很明确，因此使用 `"default"` hint。

```
// 二元加法使用默认 hint
let total = obj1 + obj2;

// obj == number 使用默认 hint
if (user == 1) { ... };
```

像 `<` 和 `>` 这样的小于/大于比较运算符，也可以同时用于字符串和数字。不过，它们使用 `"number"` hint，而不是 `"default"`。这是历史原因。

实际上，我们没有必要记住这些奇特的细节，除了一种情况（`Date` 对象，我们稍后会学到它）之外，所有内建对象都以和 `"number"` 相同的方式实现 `"default"` 转换。我们也可以这样做。

① 没有 `"boolean"` hint

请注意——只有三种 hint。就这么简单。

没有 `"boolean"` hint（在布尔上下文中所有对象都是 `true`）或其他任何东西。如果我们将 `"default"` 和 `"number"` 视为相同，就像大多数内建函数一样，那么就只有两种转换了。

为了进行转换，`JavaScript` 尝试查找并调用三个对象方法：

1. 调用 `obj[Symbol.toPrimitive](hint)` — 带有 `symbol` 键 `Symbol.toPrimitive`（系统 `symbol`）的方法，如果这个方法存在的话。
2. 否则，如果 `hint` 是 `"string"` — 尝试 `obj.toString()` 和 `obj.valueOf()`，无论哪个存在。
3. 否则，如果 `hint` 是 `"number"` 或 `"default"` — 尝试 `obj.valueOf()` 和 `obj.toString()`，无论哪个存在。

Symbol.toPrimitive

我们从第一个方法开始。有一个名为 `Symbol.toPrimitive` 的内建 `symbol`，它被用来给转换方法命名，像这样：

```
obj[Symbol.toPrimitive] = function(hint) {
  // 返回一个原始值
  // hint = "string"、"number" 和 "default" 中的一个
}
```

例如，这里 `user` 对象实现了它：

```

let user = {
  name: "John",
  money: 1000,

  [Symbol.toPrimitive](hint) {
    alert(`hint: ${hint}`);
    return hint == "string" ? `{"name": "${this.name}"}` : this.money;
  }
};

// 转换演示:
alert(user); // hint: string -> {"name": "John"}
alert(+user); // hint: number -> 1000
alert(user + 500); // hint: default -> 1500

```

从代码中我们可以看到，根据转换的不同，`user` 变成一个自描述字符串或者一个金额。单个方法 `user[Symbol.toPrimitive]` 处理了所有的转换情况。

toString/valueOf

方法 `toString` 和 `valueOf` 来自上古时代。它们不是 `symbol`（那时候还没有 `symbol` 这个概念），而是“常规的”字符串命名的方法。它们提供了一种可选的“老派”的实现转换的方法。

如果没有 `Symbol.toPrimitive`，那么 JavaScript 将尝试找到它们，并且按照下面的顺序进行尝试：

- 对于 “string” hint, `toString -> valueOf`。
- 其他情况, `valueOf -> toString`。

这些方法必须返回一个原始值。如果 `toString` 或 `valueOf` 返回了一个对象，那么返回值会被忽略（和这里没有方法的时候相同）。

默认情况下，普通对象具有 `toString` 和 `valueOf` 方法：

- `toString` 方法返回一个字符串 `"[object Object]"`。
- `valueOf` 方法返回对象自身。

下面是一个示例：

```

let user = {name: "John"};

alert(user); // [object Object]
alert(user.valueOf() === user); // true

```

所以，如果我们尝试将一个对象当做字符串来使用，例如在 `alert` 中，那么在默认情况下我们会看到 `[object Object]`。

这里提到默认值 `valueOf` 只是为了完整起见，以避免混淆。正如你看到的，它返回对象本身，因此被忽略。别问我为什么，那是历史原因。所以我们可以假设它根本就不存在。

让我们实现一下这些方法。

例如，这里的 `user` 执行和前面提到的那个 `user` 一样的操作，使用 `toString` 和 `valueOf` 的组合（而不是 `Symbol.toPrimitive`）：

```
let user = {
  name: "John",
  money: 1000,

  // 对于 hint="string"
  toString() {
    return `name: ${this.name}`;
  },

  // 对于 hint="number" 或 "default"
  valueOf() {
    return this.money;
  }
};

alert(user); // toString -> {name: "John"}
alert(+user); // valueOf -> 1000
alert(user + 500); // valueOf -> 1500
```

我们可以看到，执行的动作和前面使用 `Symbol.toPrimitive` 的那个例子相同。

通常我们希望有一个“全能”的地方来处理所有原始转换。在这种情况下，我们可以只实现 `toString`，就像这样：

```
let user = {
  name: "John",

  toString() {
    return this.name;
  }
};

alert(user); // toString -> John
alert(user + 500); // toString -> John500
```

如果没有 `Symbol.toPrimitive` 和 `valueOf`，`toString` 将处理所有原始转换。

返回类型

关于所有原始转换方法，有一个重要的点需要知道，就是它们不一定会返回 “`hint`” 的原始值。

没有限制 `toString()` 是否返回字符串，或 `Symbol.toPrimitive` 方法是否为 `hint` “`number`” 返回数字。

唯一强制性的事情是：这些方法必须返回一个原始值，而不是对象。

历史原因

由于历史原因，如果 `toString` 或 `valueOf` 返回一个对象，则不会出现 `error`，但是这种值会被忽略（就像这种方法根本不存在）。这是因为在 JavaScript 语言发展初期，没有很好的“`error`”的概念。

相反，`Symbol.toPrimitive` 必须返回一个原始值，否则就会出现 `error`。

进一步的转换

我们已经知道，许多运算符和函数执行类型转换，例如乘法 `*` 将操作数转换为数字。

如果我们将对象作为参数传递，则会出现两个阶段：

1. 对象被转换为原始值（通过前面我们描述的规则）。
2. 如果生成的原始值的类型不正确，则继续进行转换。

例如：

```
let obj = {
  // toString 在没有其他方法的情况下处理所有转换
  toString() {
    return "2";
  }
};

alert(obj * 2); // 4, 对象被转换为原始值字符串 "2"，之后它被乘法转换为数字 2。
```

1. 乘法 `obj * 2` 首先将对象转换为原始值（字符串 “2”）。
2. 之后 `"2" * 2` 变为 `2 * 2`（字符串被转换为数字）。

二元加法在同样的情况下会将其连接成字符串，因为它更愿意接受字符串：

```
let obj = {
  toString() {
    return "2";
  }
};

alert(obj + 2); // 22 ("2" + 2) 被转换为原始值字符串 => 级联
```

总结

对象到原始值的转换，是由许多期望以原始值作为值的内建函数和运算符自动调用的。

这里有三种类型（`hint`）：

- `"string"`（对于 `alert` 和其他需要字符串的操作）
- `"number"`（对于数学运算）
- `"default"`（少数运算符）

规范明确描述了哪个运算符使用哪个 hint。很少有运算符“不知道期望什么”并使用 "default" hint。通常对于内建对象，"default" hint 的处理方式与 "number" 相同，因此在实践中，最后两个 hint 常常合并在一起。

转换算法是：

1. 调用 `obj[Symbol.toPrimitive](hint)` 如果这个方法存在，
2. 否则，如果 hint 是 "string"
 - 尝试 `obj.toString()` 和 `obj.valueOf()`，无论哪个存在。
3. 否则，如果 hint 是 "number" 或者 "default"
 - 尝试 `obj.valueOf()` 和 `obj.toString()`，无论哪个存在。

在实践中，为了便于进行日志记录或调试，对于所有能够返回一种“可读性好”的对象的表达形式的转换，只实现以 `obj.toString()` 作为全能转换的方法就够了。

构造器和操作符 "new"

常规的 `{...}` 语法允许创建一个对象。但是我们经常需要创建许多类似的对象，例如多个用户或菜单项等。

这可以使用构造函数和 "new" 操作符来实现。

构造函数

构造函数在技术上是常规函数。不过有两个约定：

1. 它们的命名以大写字母开头。
2. 它们只能由 "new" 操作符来执行。

例如：

```
function User(name) {  
  this.name = name;  
  this.isAdmin = false;  
}  
  
let user = new User("Jack");  
  
alert(user.name); // Jack  
alert(user.isAdmin); // false
```

当一个函数被使用 `new` 操作符执行时，它按照以下步骤：

1. 一个新的空对象被创建并分配给 `this`。
2. 函数体执行。通常它会修改 `this`，为其添加新的属性。
3. 返回 `this` 的值。

换句话说，`new User(...)` 做的就是类似的事情：

```
function User(name) {
  // this = {}; (隐式创建)

  // 添加属性到 this
  this.name = name;
  this.isAdmin = false;

  // return this; (隐式返回)
}
```

所以 `new User("Jack")` 的结果是相同的对象：

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

现在，如果我们想创建其他用户，我们可以调用 `new User("Ann")`, `new User("Alice")` 等。比每次都使用字面量创建要短得多，而且更易于阅读。

这是构造器的主要目的 — 实现可重用的对象创建代码。

让我们再强调一遍 — 从技术上讲，任何函数都可以用作构造器。即：任何函数都可以通过 `new` 来运行，它会执行上面的算法。“首字母大写”是一个共同的约定，以明确表示一个函数将被使用 `new` 来运行。

① new function() { ... }

如果我们有许多行用于创建单个复杂对象的代码，我们可以将它们封装在构造函数中，像这样：

```
let user = new function() {
  this.name = "John";
  this.isAdmin = false;

  // .....用于用户创建的其他代码
  // 也许是复杂的逻辑和语句
  // 局部变量等
};
```

构造器不能被再次调用，因为它不保存在任何地方，只是被创建和调用。因此，这个技巧旨在封装构建单个对象的代码，而无需将来重用。

构造器模式测试: `new.target`

① 进阶内容

本节涉及的语法内容很少使用，除非你想了解所有内容，否则你可以直接跳过该语法。

在一个函数内部，我们可以使用 `new.target` 属性来检查它是否被使用 `new` 进行调用了。

对于常规调用，它为空，对于使用 `new` 的调用，则等于该函数：

```
function User() {
  alert(new.target);
}

// 不带 "new":
User(); // undefined

// 带 "new":
new User(); // function User { ... }
```

它可以被用在函数内部，来判断该函数是被通过 `new` 调用的“构造器模式”，还是没被通过 `new` 调用的“常规模式”。

我们也可以让 `new` 调用和常规调用做相同的工作，像这样：

```
function User(name) {
  if (!new.target) { // 如果你没有通过 new 运行我
    return new User(name); // .....我会给你添加 new
  }

  this.name = name;
}

let john = User("John"); // 将调用重定向到新用户
alert(john.name); // John
```

这种方法有时被用在库中以使语法更加灵活。这样人们在调用函数时，无论是否使用了 `new`，程序都能工作。

不过，到处都使用它并不是一件好事，因为省略了 `new` 使得很难观察到代码中正在发生什么。而通过 `new` 我们都可以知道这创建了一个新对象。

构造器的 `return`

通常，构造器没有 `return` 语句。它们的任务是将所有必要的东西写入 `this`，并自动转换为结果。

但是，如果这有一个 `return` 语句，那么规则就简单了：

- 如果 `return` 返回的是一个对象，则返回这个对象，而不是 `this`。
- 如果 `return` 返回的是一个原始类型，则忽略。

换句话说，带有对象的 `return` 返回该对象，在所有其他情况下返回 `this`。

例如，这里 `return` 通过返回一个对象覆盖 `this`：

```
function BigUser() {

  this.name = "John";

  return { name: "Godzilla" }; // <-- 返回这个对象
```

```
}

alert( new BigUser().name ); // Godzilla, 得到了那个对象
```

这里有一个 `return` 为空的例子（或者我们可以在它之后放置一个原始类型，没有什么影响）：

```
function SmallUser() {

    this.name = "John";

    return; // <-- 返回 this
}

alert( new SmallUser().name ); // John
```

通常构造器没有 `return` 语句。这里我们主要为了完整性而提及返回对象的特殊行为。

i 省略括号

顺便说一下，如果没有参数，我们可以省略 `new` 后的括号：

```
let user = new User; // <-- 没有参数
// 等同于
let user = new User();
```

这里省略括号不被认为是一种“好风格”，但是规范允许使用该语法。

构造器中的方法

使用构造函数来创建对象会带来很大的灵活性。构造函数可能有一些参数，这些参数定义了如何构造对象以及要放入什么。

当然，我们不仅可以将属性添加到 `this` 中，还可以添加方法。

例如，下面的 `new User(name)` 用给定的 `name` 和方法 `sayHi` 创建了一个对象：

```
function User(name) {
    this.name = name;

    this.sayHi = function() {
        alert( "My name is: " + this.name );
    };
}

let john = new User("John");

john.sayHi(); // My name is: John

/*
john = {
    name: "John",
    sayHi: function() { ... }
}
```

```
}
```

```
*/
```

[类](#) 是用于创建复杂对象的一个更高级的语法，我们稍后会讲到。

总结

- 构造函数，或简称构造器，就是常规函数，但大家对于构造器有个共同的约定，就是其命名首字母要大写。
- 构造函数只能使用 `new` 来调用。这样的调用意味着在开始时创建了空的 `this`，并在最后返回填充了值的 `this`。

我们可以使用构造函数来创建多个类似的对象。

JavaScript 为许多内置的对象提供了构造函数：比如日期 `Date`、集合 `Set` 以及其他我们计划学习的内容。

i 对象，我们还会回来哒！

在本章中，我们只介绍了关于对象和构造器的基础知识。它们对于我们将在下一章中，学习更多关于数据类型和函数的相关知识非常重要。

在我们学习了那些之后，我们将回到对象，在 [原型](#)，[继承](#) 和 [类](#) 章节中深入介绍它们。

数据类型

更多的数据结构和更深入的类型研究。

原始类型的方法

JavaScript 允许我们像使用对象一样使用原始类型（字符串，数字等）。JavaScript 还提供了这样的调用方法。我们很快就会学习它们，但是首先我们将了解它的工作原理，毕竟原始类型不是对象（在这里我们会分析地更加清楚）。

我们来看看原始类型和对象之间的关键区别。

一个原始值：

- 是原始类型中的一种值。
- 在 JavaScript 中有 7 种原始类型：`string`，`number`，`bigint`，`boolean`，`symbol`，`null` 和 `undefined`。

一个对象：

- 能够存储多个值作为属性。
- 可以使用大括号 `{}` 创建对象，例如：`{name: "John", age: 30}`。JavaScript 中还有其他种类的对象，例如函数就是对象。

关于对象的最好的事儿之一是，我们可以把一个函数作为对象的属性存储到对象中。

```
let john = {  
  name: "John",
```

```
sayHi: function() {
  alert("Hi buddy!");
}
};

john.sayHi(); // Hi buddy!
```

所以我们在那里创建了一个包含 `sayHi` 方法的对象 `john`。

许多内建对象已经存在，例如那些处理日期、错误、HTML 元素等的内建对象。它们具有不同的属性和方法。

但是，这些特性（**feature**）都是有成本的！

对象比原始类型“更重”。它们需要额外的资源来支持运作。

当作对象的原始类型

以下是 JavaScript 创建者面临的悖论：

- 人们可能想对诸如字符串或数字之类的原始类型执行很多操作。最好将它们作为方法来访问。
- 原始类型必须尽可能的简单轻量。

而解决方案看起来多少有点尴尬，如下：

- 原始类型仍然是原始的。与预期相同，提供单个值
- JavaScript 允许访问字符串，数字，布尔值和 `symbol` 的方法和属性。
- 为了使它们起作用，创建了提供额外功能的特殊“对象包装器”，使用后即被销毁。

“对象包装器”对于每种原始类型都是不同的，它们被称为 `String`、`Number`、`Boolean` 和 `Symbol`。因此，它们提供了不同的方法。

例如，字符串方法 `str.toUpperCase()` 返回一个大写化处理的字符串。

用法演示如下：

```
let str = "Hello";

alert( str.toUpperCase() ); // HELLO
```

很简单，对吧？以下是 `str.toUpperCase()` 中实际发生的情况：

- 字符串 `str` 是一个原始值。因此，在访问其属性时，会创建一个包含字符串字面值的特殊对象，并且具有有用的方法，例如 `toUpperCase()`。
- 该方法运行并返回一个新的字符串（由 `alert` 显示）。
- 特殊对象被销毁，只留下原始值 `str`。

所以原始类型可以提供方法，但它们依然是轻量级的。

JavaScript 引擎高度优化了这个过程。它甚至可能跳过创建额外的对象。但是它仍然必须遵守规范，并且表现得好像它创建了一样。

数字有其自己的方法，例如，`toFixed(n)` 将数字舍入到给定的精度：

```
let n = 1.23456;  
  
alert( n.toFixed(2) ); // 1.23
```

我们将在后面 [数字类型](#) 和 [字符串](#) 章节中看到更多具体的方法。

⚠ 构造器 `String/Number/Boolean` 仅供内部使用

像 Java 这样的语言允许我们使用 `new Number(1)` 或 `new Boolean(false)` 等语法，明确地为原始类型创建“对象包装器”。

在 JavaScript 中，由于历史原因，这也是可以的，但极其 **不推荐**。因为这样会出问题。

例如：

```
alert( typeof 0 ); // "number"  
  
alert( typeof new Number(0) ); // "object"!
```

对象在 `if` 中始终是 `true`，因此此处的 `alert` 将显示：

```
let zero = new Number(0);  
  
if (zero) { // zero 为 true, 因为它是一个对象  
  alert( "zero is truthy!?" );  
}
```

另一方面，调用不带 `new`（关键字）的 `String/Number/Boolean` 函数是完全理智和有用的。它们将一个值转换为相应的类型：转成字符串、数字或布尔值（原始类型）。

例如，下面完全是有效的：

```
let num = Number("123"); // 将字符串转成数字
```

⚠ `null/undefined` 没有任何方法

特殊的原始类型 `null` 和 `undefined` 是例外。它们没有对应的“对象包装器”，也没有提供任何方法。从某种意义上说，它们是“最原始的”。

尝试访问这种值的属性会导致错误：

```
alert(null.test); // error
```

总结

- 除 `null` 和 `undefined` 以外的原始类型都提供了许多有用的方法。我们后面的章节中学习这些内容。
- 从形式上讲，这些方法通过临时对象工作，但 JavaScript 引擎可以很好地调整，以在内部对其进行优化，因此调用它们并不需要太高的成本。

数字类型

在现代 JavaScript 中，数字（`number`）有两种类型：

1. JavaScript 中的常规数字以 64 位的格式 IEEE-754 存储，也被称为“双精度浮点数”。这是我们大多数时候所使用的数字，我们将在本章中学习它们。
2. `BigInt` 数字，用于表示任意长度的整数。有时会需要它们，因为常规数字不能超过 2^{53} 或小于 -2^{53} 。由于仅在少数特殊领域才会用到 `BigInt`，因此我们在特殊的章节 `BigInt` 中对其进行了介绍。

所以，在这里我们将讨论常规数字类型。现在让我们开始学习吧。

编写数字的更多方法

想象一下，我们需要写 10 亿。显然的方法是：

```
let billion = 1000000000;
```

但在现实生活中，我们通常避免写一长串零，因为它很容易打错。另外，我们很懒。我们通常会将 10 亿写成 `"1bn"`，或将 72 亿写成 `"7.3bn"`。对于大多数大的数字来说都是如此。

在 JavaScript 中，我们通过在数字后附加字母 “e”，并指定零的数量来缩短数字：

```
let billion = 1e9; // 10 亿，字面意思：数字 1 后面跟 9 个 0  
alert( 7.3e9 ); // 73 亿 (7,300,000,000)
```

换句话说，`"e"` 把数字乘以 1 后面跟着给定数量的 0 的数字。

```
1e3 = 1 * 1000  
1.23e6 = 1.23 * 1000000
```

现在让我们写一些非常小的数字。例如，1 微秒（百万分之一秒）：

```
let ms = 0.000001;
```

就像以前一样，可以使用 `"e"` 来完成。如果我们想避免显式地写零，我们可以这样写：

```
let ms = 1e-6; // 1 的左边有 6 个 0
```

如果我们数一下 `0.000001` 中的 0 的个数，是 6 个。所以自然是 `1e-6`。

换句话说，`e` 后面的负数表示除以 1 后面跟着给定数量的 0 的数字：

```
// -3 除以 1 后面跟着 3 个 0 的数字  
1e-3 = 1 / 1000 (=0.001)  
  
// -6 除以 1 后面跟着 6 个 0 的数字  
1.23e-6 = 1.23 / 1000000 (=0.00000123)
```

十六进制，二进制和八进制数字

[十六进制](#) 数字在 JavaScript 中被广泛用于表示颜色，编码字符以及其他许多东西。所以自然地，有一种较短的写方法：`0x`，然后是数字。

例如：

```
alert( 0xff ); // 255  
alert( 0xFF ); // 255 (一样，大小写没影响)
```

二进制和八进制数字系统很少使用，但也支持使用 `0b` 和 `0o` 前缀：

```
let a = 0b11111111; // 二进制形式的 255  
let b = 0o377; // 八进制形式的 255  
  
alert( a == b ); // true, 两边是相同的数字，都是 255
```

只有这三种进制支持这种写法。对于其他进制，我们应该使用函数 `parseInt`（我们将在本章后面看到）。

`toString(base)`

方法 `num.toString(base)` 返回在给定 `base` 进制数字系统中 `num` 的字符串表示形式。

举个例子：

```
let num = 255;  
  
alert( num.toString(16) ); // ff  
alert( num.toString(2) ); // 11111111
```

`base` 的范围可以从 `2` 到 `36`。默认情况下是 `10`。

常见的用例如下：

- **base=16** 用于十六进制颜色，字符编码等，数字可以是 `0..9` 或 `A..F`。
- **base=2** 主要用于调试按位操作，数字可以是 `0` 或 `1`。
- **base=36** 是最大进制，数字可以是 `0..9` 或 `A..Z`。所有拉丁字母都被用于了表示数字。对于 `36` 进制来说，一个有趣且有用的例子是，当我们需要将一个较长的数字标识符转换成较短的时

候，例如做一个短的 URL。可以简单地使用基数为 36 的数字系统表示：

```
alert( 123456..toString(36) ); // 2n9c
```

⚠ 使用两个点来调用一个方法

请注意 `123456..toString(36)` 中的两个点不是打错了。如果我们想直接在一个数字上调用一个方法，比如上面例子中的 `toString`，那么我们需要在它后面放置两个点 `..`。

如果我们放置一个点：`123456.toString(36)`，那么就会出现一个 error，因为 JavaScript 语法隐含了第一个点之后的部分为小数部分。如果我们再放一个点，那么 JavaScript 就知道小数部分为空，现在使用该方法。

也可以写成 `(123456).toString(36)`。

舍入

舍入 (rounding) 是使用数字时最常用的操作之一。

这里有几个对数字进行舍入的内建函数：

`Math.floor`

向下舍入：`3.1` 变成 `3`，`-1.1` 变成 `-2`。

`Math.ceil`

向上舍入：`3.1` 变成 `4`，`-1.1` 变成 `-1`。

`Math.round`

向最近的整数舍入：`3.1` 变成 `3`，`3.6` 变成 `4`，`-1.1` 变成 `-1`。

`Math.trunc` (IE 浏览器不支持这个方法)

移除小数点后的所有内容而没有舍入：`3.1` 变成 `3`，`-1.1` 变成 `-1`。

这是总结它们之间差异的表格：

	<code>Math.floor</code>	<code>Math.ceil</code>	<code>Math.round</code>	<code>Math.trunc</code>
<code>3.1</code>	<code>3</code>	<code>4</code>	<code>3</code>	<code>3</code>
<code>3.6</code>	<code>3</code>	<code>4</code>	<code>4</code>	<code>3</code>
<code>-1.1</code>	<code>-2</code>	<code>-1</code>	<code>-1</code>	<code>-1</code>
<code>-1.6</code>	<code>-2</code>	<code>-1</code>	<code>-2</code>	<code>-1</code>

这些函数涵盖了处理数字小数部分的所有可能方法。但是，如果我们想将数字舍入到小数点后 `n` 位，该怎么办？

例如，我们有 `1.2345`，并且想把它舍入到小数点后两位，仅得到 `1.23`。

有两种方式可以实现这个需求：

1. 乘除法

例如，要将数字舍入到小数点后两位，我们可以将数字乘以 `100`，调用舍入函数，然后再将其除回。

```
let num = 1.23456;

alert( Math.floor(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

2. 函数 `toFixed(n)` ↗ 将数字舍入到小数点后 `n` 位，并以字符串形式返回结果。

```
let num = 12.34;
alert( num.toFixed(1) ); // "12.3"
```

这会向上或向下舍入到最接近的值，类似于 `Math.round`：

```
let num = 12.36;
alert( num.toFixed(1) ); // "12.4"
```

请注意 `toFixed` 的结果是一个字符串。如果小数部分比所需要的短，则在结尾添加零：

```
let num = 12.34;
alert( num.toFixed(5) ); // "12.34000", 在结尾添加了 0, 以达到小数点后五位
```

我们可以使用一元加号或 `Number()` 调用，将其转换为数字：`+ num.toFixed(5)`。

不精确的计算

在内部，数字是以 64 位格式 [IEEE-754](#) 表示的，所以正好有 64 位可以存储一个数字：其中 52 位被用于存储这些数字，其中 11 位用于存储小数点的位置（对于整数，它们为零），而 1 位用于符号。

如果一个数字太大，则会溢出 64 位存储，并可能会导致无穷大：

```
alert( 1e500 ); // Infinity
```

这可能不那么明显，但经常会发生的是，精度的损失。

考虑下这个 (`falsy!`) 测试：

```
alert( 0.1 + 0.2 == 0.3 ); // false
```

没错，如果我们检查 `0.1` 和 `0.2` 的总和是否为 `0.3`，我们会得到 `false`。

奇了怪了！如果不是 `0.3`，那能是啥？

```
alert( 0.1 + 0.2 ); // 0.3000000000000004
```

哎哟！这个错误比不正确的比较的后果更严重。想象一下，你创建了一个电子购物网站，如果访问者将价格为 **¥ 0.10** 和 **¥ 0.20** 的商品放入了他的购物车。订单总额将是 **¥ 0.3000000000000004**。这会让任何人感到惊讶。

但为什么会这样呢？

一个数字以其二进制的形式存储在内存中，一个 1 和 0 的序列。但是在十进制数字系统中看起来很简单的 **0.1**, **0.2** 这样的小数，实际上在二进制形式中是无限循环小数。

换句话说，什么是 **0.1**? **0.1** 就是 **1** 除以 **10**, **1/10**，即十分之一。在十进制数字系统中，这样的数字表示起来很容易。将其与三分之一进行比较：**1/3**。三分之一变成了无限循环小数 **0.33333(3)**。

在十进制数字系统中，可以保证以 **10** 的整数次幂作为除数能够正常工作，但是以 **3** 作为除数则不能。也是同样的原因，在二进制数字系统中，可以保证以 **2** 的整数次幂作为除数时能够正常工作，但 **1/10** 就变成了一个无限循环的二进制小数。

使用二进制数字系统无法 **精确** 存储 **0.1** 或 **0.2**，就像没有办法将三分之一存储为十进制小数一样。

IEEE-754 数字格式通过将数字舍入到最接近的可能数字来解决此问题。这些舍入规则通常不允许我们看到“极小的精度损失”，但是它确实存在。

我们可以看到：

```
alert( 0.1.toFixed(20) ); // 0.1000000000000000555
```

当我们对两个数字进行求和时，它们的“精度损失”会叠加起来。

这就是为什么 **0.1 + 0.2** 不等于 **0.3**。

i 不仅仅是 JavaScript

许多其他编程语言也存在同样的问题。

PHP, **Java**, **C**, **Perl**, **Ruby** 给出的也是完全相同的结果，因为它们基于的是相同的数字格式。

我们能解决这个问题吗？当然，最可靠的方法是借助方法 **toFixed(n)** 对结果进行舍入：

```
let sum = 0.1 + 0.2;
alert( sum.toFixed(2) ); // 0.30
```

请注意，**toFixed** 总是返回一个字符串。它确保小数点后有 **2** 位数字。如果我们有一个电子购物网站，并需要显示 **¥ 0.30**，这实际上很方便。对于其他情况，我们可以使用一元加号将其强制转换为一个数字：

```
let sum = 0.1 + 0.2;
alert( +sum.toFixed(2) ); // 0.3
```

我们可以将数字临时乘以 100（或更大的数字），将其转换为整数，进行数学运算，然后再除回。当我们使用整数进行数学运算时，误差会有所减少，但仍然可以在除法中得到：

```
alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
alert( (0.28 * 100 + 0.14 * 100) / 100); // 0.4200000000000001
```

因此，乘/除法可以减少误差，但不能完全消除误差。

有时候我们可以尝试完全避免小数。例如，我们正在创建一个电子购物网站，那么我们可以用角而不是元来存储价格。但是，如果我们要打 30% 的折扣呢？实际上，完全避免小数处理几乎是不可能的。只需要在必要时剪掉其“尾巴”来对其进行舍入即可。

i 有趣的事儿

尝试运行下面这段代码：

```
// Hello! 我是一个会自我增加的数字!
alert( 9999999999999999 ); // 显示 10000000000000000
```

出现了同样的问题：精度损失。有 64 位来表示该数字，其中 52 位可用于存储数字，但这还不够。所以最不重要的数字就消失了。

JavaScript 不会在此类事件中触发 `error`。它会尽最大努力使数字符合所需的格式，但不幸的是，这种格式不够大到满足需求。

i 两个零

数字内部表示的另一个有趣结果是存在两个零：`0` 和 `-0`。

这是因为在存储时，使用一位来存储符号，因此对于包括零在内的任何数字，可以设置这一位或者不设置。

在大多数情况下，这种区别并不明显，因为运算符将它们视为相同的值。

测试：`isFinite` 和 `isNaN`

还记得这两个特殊的数值吗？

- `Infinity`（和 `-Infinity`）是一个特殊的数值，比任何数值都大（小）。
- `Nan` 代表一个 `error`。

它们属于 `number` 类型，但不是“普通”数字，因此，这里有用于检查它们的特殊函数：

- `isNaN(value)` 将其参数转换为数字，然后测试它是否为 `Nan`：

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
```

但是我们需要这个函数吗？我们不能只使用 `==` `NaN` 比较吗？不好意思，这不行。值 “`NaN`” 是独一无二的，它不等于任何东西，包括它自身：

```
alert( NaN === NaN ); // false
```

- `isFinite(value)` 将其参数转换为数字，如果是常规数字，则返回 `true`，而不是 `NaN/Infinity/-Infinity`：

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, 因为是一个特殊的值: NaN
alert( isFinite(Infinity) ); // false, 因为是一个特殊的值: Infinity
```

有时 `isFinite` 被用于验证字符串值是否为常规数字：

```
let num = +prompt("Enter a number", '');
// 结果会是 true, 除非你输入的是 Infinity、-Infinity 或不是数字
alert( isFinite(num) );
```

请注意，在所有数字函数中，包括 `isFinite`，空字符串或仅有空格的字符串均被视为 `0`。

i 与 `Object.is` 进行比较

有一个特殊的内建方法 `Object.is`，它类似于 `==` 一样对值进行比较，但它对于两种边缘情况更可靠：

1. 它适用于 `NaN`: `Object.is(NaN, NaN) === true`，这是件好事。
2. 值 `0` 和 `-0` 是不同的: `Object.is(0, -0) === false`，从技术上讲这是对的，因为在内部，数字的符号位可能会不同，即使其他所有位均为零。

在所有其他情况下，`Object.is(a, b)` 与 `a === b` 相同。

这种比较方式经常被用在 JavaScript 规范中。当内部算法需要比较两个值是否完全相同时，它使用 `Object.is`（内部称为 `SameValue`）。

parseInt 和 parseFloat

使用加号 `+` 或 `Number()` 的数字转换是严格的。如果一个值不完全是一个数字，就会失败：

```
alert( +"100px" ); // NaN
```

唯一的例外是字符串开头或结尾的空格，因为它们会被忽略。

但在现实生活中，我们经常会有带有单位的值，例如 CSS 中的 `"100px"` 或 `"12pt"`。并且，在很多国家，货币符号是紧随金额之后的，所以我们有 `"19€"`，并希望从中提取出一个数值。

这就是 `parseInt` 和 `parseFloat` 的作用。

它们可以从字符串中“读取”数字，直到无法读取为止。如果发生 `error`，则返回收集到的数字。函数 `parseInt` 返回一个整数，而 `parseFloat` 返回一个浮点数：

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12, 只有整数部分被返回了
alert( parseFloat('12.3.4') ); // 12.3, 在第二个点出停止了读取
```

某些情况下，`parseInt/parseFloat` 会返回 `NaN`。当没有数字可读时会发生这种情况：

```
alert( parseInt('a123') ); // NaN, 第一个符号停止了读取
```

i `parseInt(str, radix)` 的第二个参数

`parseInt()` 函数具有可选的第二个参数。它指定了数字系统的基数，因此 `parseInt` 还可以解析十六进制数字、二进制数字等的字符串：

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, 没有 0x 仍然有效

alert( parseInt('2n9c', 36) ); // 123456
```

其他数学函数

JavaScript 有一个内建的 `Math ↗` 对象，它包含了一个小型的数学函数和常量库。

几个例子：

`Math.random()`

返回一个从 `0` 到 `1` 的随机数（不包括 `1`）

```
alert( Math.random() ); // 0.1234567894322
alert( Math.random() ); // 0.5435252343232
alert( Math.random() ); // ... (任何随机数)
```

`Math.max(a, b, c...)` / `Math.min(a, b, c...)`

从任意数量的参数中返回最大/最小值。

```
alert( Math.max(3, 5, -10, 0, 1) ); // 5
alert( Math.min(1, 2) ); // 1
```

`Math.pow(n, power)`

返回 `n` 的给定 (`power`) 次幂

```
alert( Math.pow(2, 10) ); // 2 的 10 次幂 = 1024
```

`Math` 对象中还有更多函数和常量，包括三角函数，你可以在 [Math 函数文档](#) 中找到这些内容。

总结

要写有很多零的数字：

- 将 `"e"` 和 0 的数量附加到数字后。就像：`123e6` 与 `123` 后面接 6 个 0 相同。
- `"e"` 后面的负数将使数字除以 1 后面接着给定数量的零的数字。例如 `123e-6` 表示 `0.000123` (`123` 的百万分之一)。

对于不同的数字系统：

- 可以直接在十六进制 (`0x`)，八进制 (`0o`) 和二进制 (`0b`) 系统中写入数字。
- `parseInt(str, base)` 将字符串 `str` 解析为在给定的 `base` 数字系统中的整数，`2 ≤ base ≤ 36`。
- `num.toString(base)` 将数字转换为在给定的 `base` 数字系统中的字符串。

要将 `12pt` 和 `100px` 之类的值转换为数字：

- 使用 `parseInt/parseFloat` 进行“软”转换，它从字符串中读取数字，然后返回在发生 `error` 前可以读取到的值。

小数：

- 使用 `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` 或 `num.toFixed(precision)` 进行舍入。
- 请确保记住使用小数时会损失精度。

更多数学函数：

- 需要时请查看 [Math](#) 对象。这个库很小，但是可以满足基本的需求。

字符串

在 JavaScript 中，文本数据被以字符串形式存储，单个字符没有单独的类型。

字符串的内部格式始终是 [UTF-16](#)，它不依赖于页面编码。

引号 (Quotes)

让我们回忆一下引号的种类。

字符串可以包含在单引号、双引号或反引号中：

```
let single = 'single-quoted';
let double = "double-quoted";

let backticks = `backticks`;
```

单引号和双引号基本相同。但是，反引号允许我们通过 `${...}` 将任何表达式嵌入到字符串中：

```
function sum(a, b) {
  return a + b;
}

alert(`1 + 2 = ${sum(1, 2)} `); // 1 + 2 = 3.
```

使用反引号的另一个优点是它们允许字符串跨行：

```
let guestList = `Guests:
  * John
  * Pete
  * Mary
`;

alert(guestList); // 客人清单, 多行
```

看起来很自然，不是吗？但是单引号和双引号可不能这样做。

如果我们使用单引号或双引号来实现字符串跨行的话，则会出现错误：

```
let guestList = "Guests: // Error: Unexpected token ILLEGAL
  * John";
```

当不考虑多行字符串的需要时，单引号和双引号来自语言创建的古时代。反引号出现较晚，因此更通用。

反引号还允许我们在第一个反引号之前指定一个“模版函数”。语法是：`func`string``。函数 `func` 被自动调用，接收字符串和嵌入式表达式，并处理它们。你可以在 [docs ↗](#) 中阅读更多关于它们的信息。这叫做“**tagged templates**”。此功能可以更轻松地将字符串包装到自定义模版或其他函数中，但这很少使用。

特殊字符

我们仍然可以通过使用“换行符（newline character）”，以支持使用单引号和双引号来创建跨行字符串。换行符写作 `\n`，用来表示换行：

```
let guestList = "Guests:\n  * John\n  * Pete\n  * Mary";

alert(guestList); // 一个多行的客人列表
```

例如，这两行描述的是一样的，只是书写方式不同：

```
let str1 = "Hello\nWorld"; // 使用“换行符”创建的两行字符串

// 使用反引号和普通的换行创建的两行字符串
```

```
let str2 = `Hello  
World`;  
  
alert(str1 == str2); // true
```

还有其他不常见的“特殊”字符。

这是完整列表：

字符	描述
\n	换行
\r	回车：不单独使用。Windows 文本文件使用两个字符 \r\n 的组合来表示换行。
\' , \"	引号
\\"	反斜线
\t	制表符
\b , \f , \v	退格，换页，垂直标签 —— 为了兼容性，现在已经不使用了。
\xXX	具有给定十六进制 Unicode XX 的 Unicode 字符，例如：'\x7A' 和 'z' 相同。
\uXXXX	以 UTF-16 编码的十六进制代码 XXXX 的 unicode 字符，例如 \u00A9 —— 是版权符号 © 的 unicode。它必须正好是 4 个十六进制数字。
\u{X...XXXXXX} (1 到 6 个十六进制字符)	具有给定 UTF-32 编码的 unicode 符号。一些罕见的字符用两个 unicode 符号编码，占用 4 个字节。这样我们就可以插入长代码了。

unicode 示例：

```
alert( "\u00A9" ); // ©  
alert( "\u{20331}" ); // 品，罕见的中国象形文字（长 unicode）  
alert( "\u{1F60D}" ); // 😊, 笑脸符号（另一个长 unicode）
```

所有的特殊字符都以反斜杠字符 \ 开始。它也被称为“转义字符”。

如果我们想要在字符串中插入一个引号，我们也会使用它。

例如：

```
alert( 'I\'m the Walrus!' ); // I'm the Walrus!
```

正如你所看到的，我们必须在内部引号前加上反斜杠 \'，否则它将表示字符串结束。

当然，只有与外部闭合引号相同的引号才需要转义。因此，作为一个更优雅的解决方案，我们可以改用双引号或者反引号：

```
alert( `I'm the Walrus!` ); // I'm the Walrus!
```

注意反斜杠 \ 在 JavaScript 中用于正确读取字符串，然后消失。内存中的字符串没有 \。你从上述示例中的 alert 可以清楚地看到这一点。

但是如果我们在字符串中显示一个实际的反斜杠 \ 应该怎么做？

我们可以这样做，只需要将其书写两次 \\：

```
alert(`The backslash: \\`); // The backslash: \
```

字符串长度

`length` 属性表示字符串长度：

```
alert(`My\n`.length); // 3
```

注意 \n 是一个单独的“特殊”字符，所以长度确实是 3。

⚠ `length` 是一个属性

掌握其他编程语言的人，有时会错误地调用 `str.length()` 而不是 `str.length`。这是行不通的。

请注意 `str.length` 是一个数字属性，而不是函数。后面不需要添加括号。

访问字符

要获取在 `pos` 位置的一个字符，可以使用方括号 `[pos]` 或者调用 `str.charAt(pos)` 方法。第一个字符从零位置开始：

```
let str = `Hello`;

// 第一个字符
alert(str[0]); // H
alert(str.charAt(0)); // H

// 最后一个字符
alert(str[str.length - 1]); // o
```

方括号是获取字符的一种现代化方法，而 `charAt` 是历史原因才存在的。

它们之间的唯一区别是，如果没有找到字符，`[]` 返回 `undefined`，而 `charAt` 返回一个空字符串：

```
let str = `Hello`;

alert(str[1000]); // undefined
alert(str.charAt(1000)); // '' (空字符串)
```

我们也可以使用 `for..of` 遍历字符：

```
for (let char of "Hello") {
  alert(char); // H,e,l,l,o (char 变为 "H", 然后是 "e", 然后是 "l" 等)
}
```

字符串是不可变的

在 JavaScript 中，字符串不可更改。改变字符是不可能的。

我们证明一下为什么不可能：

```
let str = 'Hi';

str[0] = 'h'; // error
alert( str[0] ); // 无法运行
```

通常的解决方法是创建一个新的字符串，并将其分配给 `str` 而不是以前的字符串。

例如：

```
let str = 'Hi';

str = 'h' + str[1]; // 替换字符串

alert( str ); // hi
```

在接下来的章节，我们将看到更多相关示例。

改变大小写

`toLowerCase()` 和 `toUpperCase()` 方法可以改变大小写：

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

或者我们想要使一个字符变成小写：

```
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

查找子字符串

在字符串中查找子字符串有很多种方法。

`str.indexOf`

第一个方法是 `str.indexOf(substr, pos)`。

它从给定位置 `pos` 开始，在 `str` 中查找 `substr`，如果没有找到，则返回 `-1`，否则返回匹配成功的位置。

例如：

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, 因为 'Widget' 一开始就被找到
alert( str.indexOf('widget') ); // -1, 没有找到, 检索是大小写敏感的

alert( str.indexOf("id") ); // 1, "id" 在位置 1 处 (.....idget 和 id)
```

可选的第二个参数允许我们从给定的起始位置开始检索。

例如，"id" 第一次出现的位置是 1。查询下一个存在位置时，我们从 2 开始检索：

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

如果我们对所有存在位置都感兴趣，可以在一个循环中使用 `indexOf`。每一次新的调用都发生在上一匹配位置之后：

```
let str = 'As sly as a fox, as strong as an ox';

let target = 'as'; // 这是我们要查找的目标

let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert(`Found at ${foundPos}`);
  pos = foundPos + 1; // 继续从下一个位置查找
}
```

相同的算法可以简写：

```
let str = "As sly as a fox, as strong as an ox";
let target = "as";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}
```

① `str.lastIndexOf(substr, pos)`

还有一个类似的方法 `str.lastIndexOf(substr, position)` ↗，它从字符串的末尾开始搜索到开头。它会以相反的顺序列出这些事件。

在 `if` 测试中 `indexOf` 有一点不方便。我们不能像这样把它放在 `if` 中：

```
let str = "Widget with id";

if (str.indexOf("Widget")) {
    alert("We found it"); // 不工作!
}
```

上述示例中的 `alert` 不会显示，因为 `str.indexOf("Widget")` 返回 `0`（意思是它在起始位置就查找到了匹配项）。是的，但是 `if` 认为 `0` 表示 `false`。

因此我们应该检查 `-1`，像这样：

```
let str = "Widget with id";

if (str.indexOf("Widget") != -1) {
    alert("We found it"); // 现在工作了!
}
```

按位 (bitwise) NOT 技巧

这里使用的一个老技巧是 `bitwise NOT ↩ ~` 运算符。它将数字转换为 32-bit 整数（如果存在小数部分，则删除小数部分），然后对其二进制表示形式中的所有位均取反。

实际上，这意味着一件很简单的事儿：对于 32-bit 整数，`~n` 等于 `-(n+1)`。

例如：

```
alert(~2); // -3, 和 -(2+1) 相同
alert(~1); // -2, 和 -(1+1) 相同
alert(~0); // -1, 和 -(0+1) 相同
alert(~-1); // 0, 和 -(-1+1) 相同
```

正如我们看到这样，只有当 `n == -1` 时，`~n` 才为零（适用于任何 32-bit 带符号的整数 `n`）。

因此，仅当 `indexOf` 的结果不是 `-1` 时，检查 `if (~str.indexOf(...))` 才为真。换句话说，当有匹配时。

人们用它来简写 `indexOf` 检查：

```
let str = "Widget";

if (~str.indexOf("Widget")) {
    alert('Found it!'); // 正常运行
}
```

通常不建议以非显而易见的方式使用语言特性，但这种特殊技巧在旧代码中仍被广泛使用，所以我们应该理解它。

只要记住：`if (~str.indexOf(...))` 读作“if found”。

确切地说，由于 `~` 运算符将大数字截断为 32 位，因此存在给出 `0` 的其他数字，最小的数字是 `-4294967295=0`。这使得这种检查只有在字符串没有那么长的情况下才是正确的。

现在我们只会在旧的代码中看到这个技巧，因为现代 JavaScript 提供了 `.includes` 方法（见下文）。

includes, startsWith, endsWith

更现代的方法 `str.includes(substr, pos)` ↗ 根据 `str` 中是否包含 `substr` 来返回 `true/false`。

如果我们需要检测匹配，但不需要它的位置，那么这是正确的选择：

```
alert( "Widget with id".includes("Widget") ); // true  
alert( "Hello".includes("Bye") ); // false
```

`str.includes` 的第二个可选参数是开始搜索的起始位置：

```
alert( "Midget".includes("id") ); // true  
alert( "Midget".includes("id", 3) ); // false, 从位置 3 开始没有 "id"
```

方法 `str.startsWith` ↗ 和 `str.endsWith` ↗ 的功能与其名称所表示的意思相同：

```
alert( "Widget".startsWith("Wid") ); // true, "Widget" 以 "Wid" 开始  
alert( "Widget".endsWith("get") ); // true, "Widget" 以 "get" 结束
```

获取子字符串

JavaScript 中有三种获取字符串的方法：`substring`、`substr` 和 `slice`。

str.slice(start [, end])

返回字符串从 `start` 到（但不包括）`end` 的部分。

例如：

```
let str = "stringify";  
alert( str.slice(0, 5) ); // 'strin', 从 0 到 5 的子字符串（不包括 5）  
alert( str.slice(0, 1) ); // 's', 从 0 到 1, 但不包括 1, 所以只有在 0 处的字符
```

如果没有第二个参数，`slice` 会一直运行到字符串末尾：

```
let str = "stringify";  
alert( str.slice(2) ); // 从第二个位置直到结束
```

`start/end` 也有可能是负值。它们的意思是起始位置从字符串结尾计算：

```
let str = "stringify";
```

```
// 从右边的第四个位置开始，在右边的第一个位置结束
alert( str.slice(-4, -1) ); // 'gif'
```

str.substring(start [, end])

返回字符串在 `start` 和 `end` 之间的部分。

这与 `slice` 几乎相同，但它允许 `start` 大于 `end`。

例如：

```
let str = "stringify";

// 这些对于 substring 是相同的
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// .....但对 slice 是不同的：
alert( str.slice(2, 6) ); // "ring" (一样)
alert( str.slice(6, 2) ); // "" (空字符串)
```

不支持负参数（不像 `slice`），它们被视为 `0`。

str.substr(start [, length])

返回字符串从 `start` 开始的给定 `length` 的部分。

与以前的方法相比，这个允许我们指定 `length` 而不是结束位置：

```
let str = "stringify";
alert( str.substr(2, 4) ); // 'ring'，从位置 2 开始，获取 4 个字符
```

第一个参数可能是负数，从结尾算起：

```
let str = "stringify";
alert( str.substr(-4, 2) ); // 'gi'，从第 4 位获取 2 个字符
```

我们回顾一下这些方法，以免混淆：

方法	选择方式.....	负值参数
<code>slice(start, end)</code>	从 <code>start</code> 到 <code>end</code> (不含 <code>end</code>)	允许
<code>substring(start, end)</code>	<code>start</code> 与 <code>end</code> 之间 (包括 <code>start</code> ，但不包括 <code>end</code>)	负值代表 <code>0</code>
<code>substr(start, length)</code>	从 <code>start</code> 开始获取长为 <code>length</code> 的字符串	允许 <code>start</code> 为负数

i 使用哪一个?

它们可以完成这项工作。形式上, `substr` 有一个小缺点: 它不是在 JavaScript 核心规范中描述的, 而是在附录 B 中, 它涵盖了主要由于历史原因而存在的仅浏览器特性。因此, 非浏览器环境可能无法支持它。但实际上它在任何地方都有效。

相较于其他两个变体, `slice` 稍微灵活一些, 它允许以负值作为参数并且写法更简短。因此仅仅记住这三种方法中的 `slice` 就足够了。

比较字符串

正如我们从 [值的比较](#) 一章中了解到的, 字符串按字母顺序逐字比较。

不过, 也有一些奇怪的地方。

1. 小写字母总是大于大写字母:

```
alert( 'a' > 'Z' ); // true
```

2. 带变音符号的字母存在“乱序”的情况:

```
alert( 'Österreich' > 'Zealand' ); // true
```

如果我们对这些国家名进行排序, 可能会导致奇怪的结果。通常, 人们会期望 `Zealand` 在名单中的 `Österreich` 之后出现。

为了明白发生了什么, 我们回顾一下在 JavaScript 中字符串的内部表示。

所有的字符串都使用 [UTF-16 ↗](#) 编码。即: 每个字符都有对应的数字代码。有特殊的方法可以获取代码表示的字符, 以及字符对应的代码。

`str.codePointAt(pos)`

返回在 `pos` 位置的字符代码:

```
// 不同的字母有不同的代码
alert( "z".codePointAt(0) ); // 122
alert( "Z".codePointAt(0) ); // 90
```

`String.fromCodePoint(code)`

通过数字 `code` 创建字符

```
alert( String.fromCodePoint(90) ); // Z
```

我们还可以用 `\u` 后跟十六进制代码, 通过这些代码添加 `unicode` 字符:

```
// 在十六进制系统中 90 为 5a  
alert( '\u005a' ); // Z
```

现在我们看一下代码为 `65..220` 的字符（拉丁字母和一些额外的字符），方法是创建一个字符串：

看到没？先是大写字符，然后是一些特殊字符，然后是小写字符，而 Ö 几乎是最后输出。

现在很明显为什么 $a > z$ 。

字符通过数字代码进行比较。越大的代码意味着字符越大。`a` (97) 的代码大于 `z` (90) 的代码。

- 所有小写字母追随在大写字母之后，因为它们的代码更大。
 - 一些像 Ö 的字母与主要字母表不同。这里，它的代码比任何从 a 到 z 的代码都要大。

正确的比较

执行字符串比较的“正确”算法比看起来更复杂，因为不同语言的字母都不相同。

因此浏览器需要知道要比较的语言。

幸运的是，所有现代浏览器（IE10+ 需要额外的库 Intl.js [↗](#)）都支持国际化标准 ECMA-402 [↗](#)。

它提供了一种特殊的方法来比较不同语言的字符串，遵循它们的规则。

调用 `str.localeCompare(str2)` 会根据语言规则返回一个整数，这个整数能表明 `str` 是否在 `str2` 前、后或者等于它：

- 如果 `str` 小于 `str2` 则返回负数。
 - 如果 `str` 大于 `str2` 则返回正数。
 - 如果它们相等则返回 `0`。

例如：

```
alert( 'Österreich'.localeCompare('Zealand') ); // -1
```

这个方法实际上在 [文档](#) 中指定了两个额外的参数，这两个参数允许它指定语言（默认语言从环境中获取，字符顺序视语言不同而不同）并设置诸如区别大小之类的附加规则，或应该将 "a" 和 "á" 看作相等情况等。

内部: Unicode

⚠ 进阶内容

这部分会深入字符串内部。如果你计划处理 emoji、罕见的数学或象形文字或其他罕见的符号，这些知识会对你有用。

如果你不打算支持它们，你可以跳过这一部分。

代理对

所有常用的字符都是一个 2 字节的代码。大多数欧洲语言，数字甚至大多数象形文字中的字母都有 2 字节的表示形式。

但 2 字节只允许 65536 个组合，这对于表示每个可能的符号是不够的。所以稀有的符号被称为“代理对”的一对 2 字节的符号编码。

这些符号的长度是 2：

```
alert('ꝝ'.length); // 2, 大写数学符号 X  
alert('ꝑ'.length); // 2, 笑哭表情  
alert('𩫓'.length); // 2, 罕见的中国象形文字
```

注意，代理对在 JavaScript 被创建时并不存在，因此无法被编程语言正确处理！

我们实际上在上面的每个字符串中都有一个符号，但 `length` 显示长度为 2。

`String.fromCodePoint` 和 `str.codePointAt` 是几种处理代理对的少数方法。它们最近才出现在编程语言中。在它们之前，只有 `String.fromCharCode ↗` 和 `str.charCodeAt ↗`。这些方法实际上与 `fromCodePoint/codePointAt` 相同，但是不适用于代理对。

获取符号可能会非常麻烦，因为代理对被认为是两个字符：

```
alert('ꝝ'[0]); // 奇怪的符号.....  
alert('ꝝ'[1]); // .....代理对的一块
```

请注意，代理对的各部分没有任何意义。因此，上述示例中的 `alert` 显示的实际上是垃圾信息。

技术角度来说，代理对也是可以通过它们的代码检测到的：如果一个字符的代码在 `0xd800..0xdbff` 范围内，那么它是代理对的第一部分。下一个字符（第二部分）必须在 `0xdc00..0xffff` 范围中。这些范围是按照标准专门为代理对保留的。

在上述示例中：

```
// charCodeAt 不理解代理对，所以它给出了代理对的代码  
  
alert('ꝝ'.charCodeAt(0).toString(16)); // d835, 在 0xd800 和 0xdbff 之间  
alert('ꝝ'.charCodeAt(1).toString(16)); // dc03, 在 0xdc00 和 0xffff 之间
```

本章节后面的 [Iterable object \(可迭代对象\)](#) 章节中，你可以找到更多处理代理对的方法。可能也专门的库，这里没有什么足够好的建议了。

变音符号与规范化

在许多语言中，都有一些由基本字符组成的符号，在其上方/下方有一个标记。

例如，字母 `a` 可以是 `àáâäååä` 的基本字符。最常见的“复合”字符在 **UTF-16** 表中都有自己的代码。但不是全部，因为可能的组合太多。

为了支持任意组合，**UTF-16** 允许我们使用多个 `unicode` 字符：基本字符紧跟“装饰”它的一个或多个“标记”字符。

例如，如果我们 `S` 后跟有特殊的“dot above”字符（代码 `\u0307`），则显示 `Ś`。

```
alert( 'S\u0307' ); // Ś
```

如果我们需要在字母上方（或下方）添加额外的标记——没问题，只需要添加必要的标记字符即可。

例如，如果我们追加一个字符“dot below”（代码 `\u0323`），那么我们将得到“`S` 上面和下面都有点”的字符：`Ś`。

例如：

```
alert( 'S\u0307\u0323' ); // Ś
```

这在提供良好灵活性的同时，也存在一个有趣的问题：两个视觉上看起来相同的字符，可以用不同的 `unicode` 组合表示。

例如：

```
let s1 = 'S\u0307\u0323'; // Ś, S + 上点 + 下点
let s2 = 'S\u0323\u0307'; // Ś, S + 下点 + 上点

alert( `s1: ${s1}, s2: ${s2}` );

alert( s1 == s2 ); // false, 尽管字符看起来相同 (?!)
```

为了解决这个问题，有一个“`unicode` 规范化”算法，它将每个字符串都转化成单个“通用”格式。

它由 `str.normalize()` 实现。

```
alert( "S\u0307\u0323".normalize() == "S\u0323\u0307".normalize() ); // true
```

有趣的是，在实际情况下，`normalize()` 实际上将一个由 3 个字符组成的序列合并为一个：`\u1e68` (`S` 有两个点)。

```
alert( "S\u0307\u0323".normalize().length ); // 1

alert( "S\u0307\u0323".normalize() == "\u1e68" ); // true
```

事实上，情况并非总是如此，因为符号 `Ś` 是“常用”的，所以 **UTF-16** 创建者把它包含在主表中并给它了对应的代码。

如果你想了解更多关于规范化规则和变体的信息——它们在 [Unicode](#) 标准附录中有详细描述：[Unicode 规范化形式](#)，但对于大多数实际目的来说，本文的内容就已经足够了。

总结

- 有 3 种类型的引号。反引号允许字符串跨越多行并可以使用 ``${...}`` 在字符串中嵌入表达式。
- JavaScript 中的字符串使用的是 [UTF-16](#) 编码。
- 我们可以使用像 `\n` 这样的特殊字符或通过使用 `\u...` 来操作它们的 `unicode` 进行字符插入。
- 获取字符时，使用 `[]`。
- 获取子字符串，使用 `slice` 或 `substring`。
- 字符串的大/小写转换，使用： `toLowerCase/toUpperCase`。
- 查找子字符串时，使用 `indexOf` 或 `includesstartsWith/endsWith` 进行简单检查。
- 根据语言比较字符串时使用 `localeCompare`，否则将按字符代码进行比较。

还有其他几种有用的字符串方法：

- `str.trim()` —— 删除字符串前后的空格 (“trims”)。
- `str.repeat(n)` —— 重复字符串 `n` 次。
-更多内容细节请参见 [手册](#)。

字符串还具有使用正则表达式进行搜索/替换的方法。但这个话题很大，因此我们将在本教程中单独的 [正则表达式](#) 章节中进行讨论。

数组

对象允许存储键值集合，这很好。

但很多时候我们发现还需要 **有序集合**，里面的元素都是按顺序排列的。例如，我们可能需要存储一些列表，比如用户、商品以及 HTML 元素等。

这里使用对象就不是很方便了，因为对象不能提供能够管理元素顺序的方法。我们不能在已有的元素“之间”插入一个新的属性。这种场景下对象就不太适用了。

这时一个特殊的数据结构数组（`Array`）就派上用场了，它能存储有序的集合。

声明

创建一个空数组有两种语法：

```
let arr = new Array();
let arr = [];
```

绝大多数情况下使用的都是第二种语法。我们可以在方括号中添加初始元素：

```
let fruits = ["Apple", "Orange", "Plum"];
```

数组元素从 0 开始编号。

我们可以通过方括号中的数字获取元素：

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

可以替换元素：

```
fruits[2] = 'Pear'; // 现在变成了 ["Apple", "Orange", "Pear"]
```

.....或者向数组新加一个元素：

```
fruits[3] = 'Lemon'; // 现在变成 ["Apple", "Orange", "Pear", "Lemon"]
```

`length` 属性的值是数组中元素的总个数：

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits.length ); // 3
```

也可以用 `alert` 来显示整个数组。

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits ); // Apple,Orange,Plum
```

数组可以存储任何类型的元素。

例如：

```
// 混合值
let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];

// 获取索引为 1 的对象然后显示它的 name
alert( arr[1].name ); // John

// 获取索引为 3 的函数并执行
arr[3](); // hello
```

① 以逗号结尾

数组就像对象一样，可以以逗号结尾：

```
let fruits = [  
    "Apple",  
    "Orange",  
    "Plum",  
];
```

因为每一行都是相似的，所以这种以“逗号结尾”的方式使得插入/移除项变得更加简单。

pop/push, shift/unshift 方法

队列（queue）[🔗](#)是最常见的使用数组的方法之一。在计算机科学中，这表示支持两个操作的一个有序元素的集合：

- `push` 在末端添加一个元素。
- `shift` 取出队列前端的一个元素，整个队列往前移，这样原先排第二的元素现在排在了第一。



这两种操作数组都支持。

队列的应用在实践中经常会碰到。例如需要在屏幕上显示消息队列。

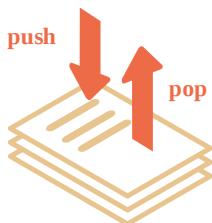
数组还有另一个用例，就是数据结构栈[🔗](#)。

它支持两种操作：

- `push` 在末端添加一个元素。
- `pop` 从末端取出一个元素。

所以新元素的添加和取出都是从“末端”开始的。

栈通常被形容成一叠卡片：要么在最上面添加卡片，要么从最上面拿走卡片：



对于栈来说，最后放进去的内容是最先接收的，也叫做 LIFO（Last-In-First-Out），即后进先出法则。而与队列相对应的叫做 FIFO（First-In-First-Out），即先进先出。

JavaScript 中的数组既可以用作队列，也可以用作栈。它们允许你从首端/末端来添加/删除元素。

这在计算机科学中，允许这样的操作的数据结构被称为 双端队列（deque）[↗](#)。

作用于数组末端的方法：

pop

取出并返回数组的最后一个元素：

```
let fruits = ["Apple", "Orange", "Pear"];  
  
alert( fruits.pop() ); // 移除 "Pear" 然后 alert 显示出来  
  
alert( fruits ); // Apple, Orange
```

push

在数组末端添加元素：

```
let fruits = ["Apple", "Orange"];  
  
fruits.push("Pear");  
  
alert( fruits ); // Apple, Orange, Pear
```

调用 `fruits.push(...)` 与 `fruits[fruits.length] = ...` 是一样的。

作用于数组首端的方法：

shift

取出数组的第一个元素并返回它：

```
let fruits = ["Apple", "Orange", "Pear"];  
  
alert( fruits.shift() ); // 移除 Apple 然后 alert 显示出来  
  
alert( fruits ); // Orange, Pear
```

unshift

在数组的首端添加元素：

```
let fruits = ["Orange", "Pear"];  
  
fruits.unshift('Apple');  
  
alert( fruits ); // Apple, Orange, Pear
```

`push` 和 `unshift` 方法都可以一次添加多个元素：

```
let fruits = ["Apple"];
fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");
// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert( fruits );
```

内部

数组是一种特殊的对象。使用方括号来访问属性 `arr[0]` 实际上是来自于对象的语法。它其实与 `obj[key]` 相同，其中 `arr` 是对象，而数字用作键 (`key`)。

它们扩展了对象，提供了特殊的方法来处理有序的数据集合以及 `length` 属性。但从本质上讲，它仍然是一个对象。

记住，在 JavaScript 中只有 7 种基本类型。数组是一个对象，因此其行为也像一个对象。

例如，它是通过引用来复制的：

```
let fruits = ["Banana"]

let arr = fruits; // 通过引用复制（两个变量引用的是相同的数组）

alert( arr === fruits ); // true

arr.push("Pear"); // 通过引用修改数组

alert( fruits ); // Banana, Pear – 现在有 2 项了
```

.....但是数组真正特殊的是它们的内部实现。JavaScript 引擎尝试把这些元素一个接一个地存储在连续的内存区域，就像本章的插图显示的一样，而且还有一些其它的优化，以使数组运行得非常快。

但是，如果我们不像“有序集合”那样使用数组，而是像常规对象那样使用数组，这些就都不生效了。

例如，从技术上讲，我们可以这样做：

```
let fruits = []; // 创建一个数组

fruits[99999] = 5; // 分配索引远大于数组长度的属性

fruits.age = 25; // 创建一个具有任意名称的属性
```

这是可以的，因为数组是基于对象的。我们可以给它们添加任何属性。

但是 Javascript 引擎会发现，我们在像使用常规对象一样使用数组，那么针对数组的优化就不再适用了，然后对应的优化就会被关闭，这些优化所带来的优势也就荡然无存了。

数组误用的几种方式：

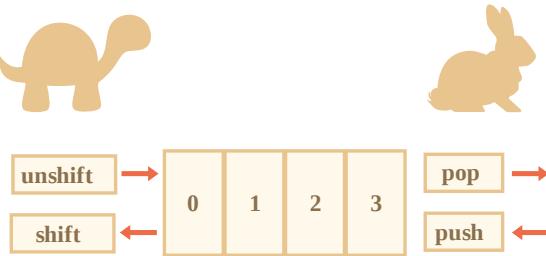
- 添加一个非数字的属性，比如 `arr.test = 5`。

- 制造空洞，比如：添加 `arr[0]`，然后添加 `arr[1000]`（它们中间什么都没有）。
- 以倒序填充数组，比如 `arr[1000]`, `arr[999]` 等等。

请将数组视为作用于 **有序数据** 的特殊结构。它们为此提供了特殊的方法。数组在 JavaScript 引擎内部是经过特殊调整的，使得更好地作用于连续的有序数据，所以请以正确的方式使用数组。如果你需要任意键值，那很有可能实际上你需要的是常规对象 `{}`。

性能

`push/pop` 方法运行的比较快，而 `shift/unshift` 比较慢。



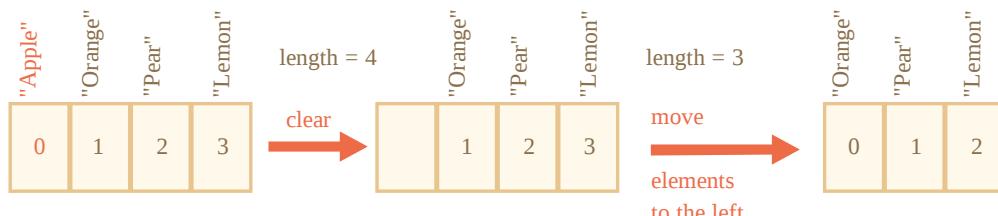
为什么作用于数组的末端会比首端快呢？让我们看看在执行期间都发生了什么：

```
fruits.shift(); // 从首端取出一个元素
```

只获取并移除数字 `0` 对应的元素是不够的。其它元素也需要被重新编号。

`shift` 操作必须做三件事：

1. 移除索引为 `0` 的元素。
2. 把所有的元素向左移动，把索引 `1` 改成 `0`, `2` 改成 `1` 以此类推，对其重新编号。
3. 更新 `length` 属性。



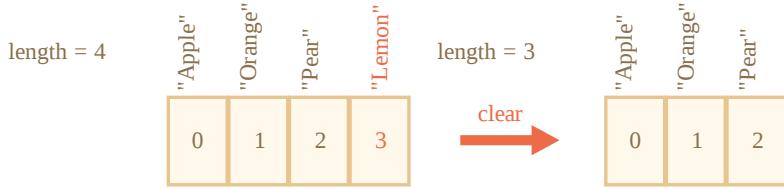
数组里的元素越多，移动它们就要花越多的时间，也就意味着越多的内存操作。

`unshift` 也是一样：为了在数组的首端添加元素，我们首先需要将现有的元素向右移动，增加它们的索引值。

那 `push/pop` 是什么样的呢？它们不需要移动任何东西。如果从末端移除一个元素，`pop` 方法只需要清理索引值并缩短 `length` 就可以了。

`pop` 操作的行为：

```
fruits.pop(); // 从末端取走一个元素
```



`pop` 方法不需要移动任何东西，因为其它元素都保留了各自的索引。这就是为什么 `pop` 会特别快。

`push` 方法也是一样的。

循环

遍历数组最古老的方式就是 `for` 循环：

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

但对于数组来说还有另一种循环方式，`for..of`：

```
let fruits = ["Apple", "Orange", "Plum"];

// 遍历数组元素
for (let fruit of fruits) {
  alert( fruit );
}
```

`for..of` 不能获取当前元素的索引，只是获取元素值，但大多数情况是够用的。而且这样写更短。

技术上来讲，因为数组也是对象，所以使用 `for..in` 也是可以的：

```
let arr = ["Apple", "Orange", "Pear"];

for (let key in arr) {
  alert( arr[key] ); // Apple, Orange, Pear
}
```

但这其实是一个很不好的想法。会有一些潜在问题存在：

1. `for..in` 循环会遍历 **所有属性**，不仅仅是这些数字属性。

在浏览器和其它环境中有一种称为“类数组”的对象，它们 **看似是数组**。也就是说，它们有 `length` 和索引属性，但是也可能有其它的非数字的属性和方法，这通常是我们不需要的。

`for..in` 循环会把它们都列出来。所以如果我们要处理类数组对象，这些“额外”的属性就会存在问题。

2. `for..in` 循环适用于普通对象，并且做了对应的优化。但是不适用于数组，因此速度要慢 10-100 倍。当然即使是这样也依然非常快。只有在遇到瓶颈时可能会有问题。但是我们仍然应该了解这其中的不同。

通常来说，我们不应该用 `for..in` 来处理数组。

关于 “`length`”

当我们修改数组的时候，`length` 属性会自动更新。准确来说，它实际上不是数组里元素的个数，而是最大的数字索引值加一。

例如，一个数组只有一个元素，但是这个元素的索引值很大，那么这个数组的 `length` 也会很大：

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

要知道的是我们通常不会这样使用数组。

`length` 属性的另一个有意思的点是它是可写的。

如果我们手动增加它，则不会发生任何有趣的事儿。但是如果我们减少它，数组就会被截断。该过程是不可逆的，下面是例子：

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // 截断到只剩 2 个元素
alert( arr ); // [1, 2]

arr.length = 5; // 又把 length 加回来
alert( arr[3] ); // undefined: 被截断的那些数值并没有回来
```

所以，清空数组最简单的方法就是：`arr.length = 0;`。

`new Array()`

这是创建数组的另一种语法：

```
let arr = new Array("Apple", "Pear", "etc");
```

它很少被使用，因为方括号 `[]` 更短更简洁。而且这种语法还存在一些诡异的特性。

如果使用单个参数（即数字）调用 `new Array`，那么它会创建一个 指定了长度，却没有任何项目的数组。

让我们看看如何搬起石头砸自己的脚：

```
let arr = new Array(2); // 会创建一个 [2] 的数组吗?  
alert( arr[0] ); // undefined! 没有元素。  
alert( arr.length ); // length 2
```

在上面的代码中，`new Array(number)` 创建的数组的所有元素都是 `undefined`。

为了避免这种乌龙事件，我们通常都是使用方括号的，除非我们清楚地知道自己正在做什么。

多维数组

数组里的项也可以是数组。我们可以将其用于多维数组，例如存储矩阵：

```
let matrix = [  
  [1, 2, 3],  
  [4, 5, 6],  
  [7, 8, 9]  
];  
  
alert( matrix[1][1] ); // 最中间的那个数
```

toString

数组有自己的 `toString` 方法的实现，会返回以逗号隔开的元素列表。

例如：

```
let arr = [1, 2, 3];  
  
alert( arr ); // 1,2,3  
alert( String(arr) === '1,2,3' ); // true
```

此外，我们试试运行一下这个：

```
alert( [] + 1 ); // "1"  
alert( [1] + 1 ); // "11"  
alert( [1,2] + 1 ); // "1,21"
```

数组没有 `Symbol.toPrimitive`，也没有 `valueOf`，它们只能执行 `toString` 进行转换，所以这里 `[]` 就变成了一个空字符串，`[1]` 变成了 `"1"`，`[1,2]` 变成了 `"1,2"`。

当 `+"` 运算符把一些项加到字符串后面时，加号后面的项也会被转换成字符串，所以下一步就会是这样：

```
alert( "" + 1 ); // "1"  
alert( "1" + 1 ); // "11"  
alert( "1,2" + 1 ); // "1,21"
```

总结

数组是一种特殊的对象，适用于存储和管理有序的数据项。

- 声明：

```
// 方括号 (常见用法)
let arr = [item1, item2...];

// new Array (极其少见)
let arr = new Array(item1, item2...);
```

调用 `new Array(number)` 会创建一个给定长度的数组，但不含有任何项。

- `length` 属性是数组的长度，准确地说，它是数组最后一个数字索引值加一。它由数组方法自动调整。
- 如果我们手动缩短 `length`，那么数组就会被截断。

我们可以通过下列操作以双端队列的方式使用数组：

- `push(...items)` 在末端添加 `items` 项。
- `pop()` 从末端移除并返回该元素。
- `shift()` 从首端移除并返回该元素。
- `unshift(...items)` 从首端添加 `items` 项。

遍历数组的元素：

- `for (let i=0; i<arr.length; i++)` — 运行得最快，可兼容旧版本浏览器。
- `for (let item of arr)` — 现代语法，只能访问 `items`。
- `for (let i in arr)` — 永远不要用这个。

在下一章节 [数组方法](#) 中，我们会继续学习数组，学习更多添加、移除、提取元素和数组排序的方法。

数组方法

数组提供的方法有很多。为了方便起见，在本章中，我们将按组讲解。

添加/移除数组元素

我们已经学了从数组的首端或尾端添加和删除元素的方法：

- `arr.push(...items)` — 从尾端添加元素，
- `arr.pop()` — 从尾端提取元素，
- `arr.shift()` — 从首端提取元素，
- `arr.unshift(...items)` — 从首端添加元素。

这里还有其他几种方法。

splice

如何从数组中删除元素?

数组是对象, 所以我们可以尝试使用 `delete`:

```
let arr = ["I", "go", "home"];  
  
delete arr[1]; // remove "go"  
  
alert( arr[1] ); // undefined  
  
// now arr = ["I", , "home"];  
alert( arr.length ); // 3
```

元素被删除了, 但数组仍然有 3 个元素, 我们可以看到 `arr.length == 3`。

这很正常, 因为 `delete obj.key` 是通过 `key` 来移除对应的值。对于对象来说是可以的。但是对于数组来说, 我们通常希望剩下的元素能够移动并占据被释放的位置。我们希望得到一个更短的数组。

所以应该使用特殊的方法。

`arr.splice(str)` 方法可以说是处理数组的瑞士军刀。它可以做所有事情: 添加, 删除和插入元素。

语法是:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

从 `index` 开始: 删除 `deleteCount` 个元素并在当前位置插入 `elem1, ..., elemN`。最后返回已删除元素的数组。

通过例子我们可以很容易地掌握这个方法。

让我们从删除开始:

```
let arr = ["I", "study", "JavaScript"];  
  
arr.splice(1, 1); // 从索引 1 开始删除 1 个元素  
  
alert( arr ); // ["I", "JavaScript"]
```

简单, 对吧? 从索引 `1` 开始删除 `1` 个元素。

在下一个例子中, 我们删除了 3 个元素, 并用另外两个元素替换它们:

```
let arr = ["I", "study", "JavaScript", "right", "now"];  
  
// remove 3 first elements and replace them with another  
arr.splice(0, 3, "Let's", "dance");  
  
alert( arr ) // now ["Let's", "dance", "right", "now"]
```

在这里我们可以看到 `splice` 返回了已删除元素的数组:

```
let arr = ["I", "study", "JavaScript", "right", "now"];  
  
// 删除前两个元素  
let removed = arr.splice(0, 2);  
  
alert( removed ); // "I", "study" <-- 被从数组中删除了的元素
```

我们可以将 `deleteCount` 设置为 `0`, `splice` 方法就能够插入元素而不用删除任何元素:

```
let arr = ["I", "study", "JavaScript"];  
  
// 从索引 2 开始  
// 删除 0 个元素  
// 然后插入 "complex" 和 "language"  
arr.splice(2, 0, "complex", "language");  
  
alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

i 允许负向索引

在这里和其他数组方法中, 负向索引都是被允许的。它们从数组末尾计算位置, 如下所示:

```
let arr = [1, 2, 5];  
  
// 从索引 -1 (尾端前一位)  
// 删除 0 个元素,  
// 然后插入 3 和 4  
arr.splice(-1, 0, 3, 4);  
  
alert( arr ); // 1,2,3,4,5
```

slice

`arr.slice ↗` 方法比 `arr.splice` 简单得多。

语法是:

```
arr.slice([start], [end])
```

它会返回一个新数组, 将所有从索引 `start` 到 `end` (不包括 `end`) 的数组项复制到一个新的数组。`start` 和 `end` 都可以是负数, 在这种情况下, 从末尾计算索引。

它和字符串的 `str.slice` 方法有点像, 就是把子字符串替换成子数组。

例如:

```
let arr = ["t", "e", "s", "t"];
```

```
alert( arr.slice(1, 3) ); // e,s (复制从位置 1 到位置 3 的元素)
```

```
alert( arr.slice(-2) ); // s,t (复制从位置 -2 到尾端的元素)
```

我们也可以不带参数地调用它: `arr.slice()` 会创建一个 `arr` 的副本。其通常用于获取副本, 以进行不影响原始数组的进一步转换。

concat

`arr.concat ↗` 创建一个新数组, 其中包含来自于其他数组和其他项的值。

语法:

```
arr.concat(arg1, arg2...)
```

它接受任意数量的参数 — 数组或值都可以。

结果是一个包含来自于 `arr`, 然后是 `arg1`, `arg2` 的元素的新数组。

如果参数 `argN` 是一个数组, 那么其中的所有元素都会被复制。否则, 将复制参数本身。

例如:

```
let arr = [1, 2];

// create an array from: arr and [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4

// create an array from: arr and [3,4] and [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

通常, 它只复制数组中的元素。其他对象, 即使它们看起来像数组一样, 但仍然会被作为一个整体添加:

```
let arr = [1, 2];

let arrayLike = {
  0: "something",
  length: 1
};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

.....但是, 如果类似数组的对象具有 `Symbol.isConcatSpreadable` 属性, 那么它就会被 `concat` 当作一个数组来处理: 此对象中的元素将被添加:

```
let arr = [1, 2];

let arrayLike = {
```

```
0: "something",
1: "else",
[Symbol.isConcatSpreadable]: true,
length: 2
};

alert( arr.concat(arrayLike) ); // 1,2,something,else
```

遍历: `forEach`

`arr.forEach ↗` 方法允许为数组的每个元素都运行一个函数。

语法:

```
arr.forEach(function(item, index, array) {
  // ... do something with item
});
```

例如，下面这个程序显示了数组的每个元素:

```
// 对每个元素调用 alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

而这段代码更详细地介绍了它们在目标数组中的位置:

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`#${item} is at index ${index} in ${array}`);
});
```

该函数的结果（如果它有返回）会被抛弃和忽略。

在数组中搜索

现在，让我们介绍在数组中进行搜索的方法。

`indexOf/lastIndexOf` 和 `includes`

`arr.indexOf ↗`、`arr.lastIndexOf ↗` 和 `arr.includes ↗` 方法与字符串操作具有相同的语法，并且作用基本上也与字符串的方法相同，只不过这里是对数组元素而不是字符进行操作:

- `arr.indexOf(item, from)` 从索引 `from` 开始搜索 `item`，如果找到则返回索引，否则返回 `-1`。
- `arr.lastIndexOf(item, from)` — 和上面相同，只是从右向左搜索。
- `arr.includes(item, from)` — 从索引 `from` 开始搜索 `item`，如果找到则返回 `true`（译注：如果没找到，则返回 `false`）。

例如:

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true
```

请注意，这些方法使用的是严格相等 `==` 比较。所以如果我们搜索 `false`，会精确到的确是 `false` 而不是数字 `0`。

如果我们想检查是否包含某个元素，并且不想知道确切的索引，那么 `arr.includes` 是首选。

此外，`includes` 的一个非常小的差别是它能正确处理 `Nan`，而不像 `indexOf/lastIndexOf`：

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (应该为 0, 但是严格相等 === equality 对 NaN 无效)
alert( arr.includes(NaN) );// true (这个结果是对的)
```

find 和 `findIndex`

想象一下，我们有一个对象数组。我们如何找到具有特定条件的对象？

这时可以用 `arr.find` 方法。

语法如下：

```
let result = arr.find(function(item, index, array) {
  // 如果返回 true，则返回 item 并停止迭代
  // 对于 falsy 则返回 undefined
});
```

依次对数组中的每个元素调用该函数：

- `item` 是元素。
- `index` 是它的索引。
- `array` 是数组本身。

如果它返回 `true`，则搜索停止，并返回 `item`。如果没有搜索到，则返回 `undefined`。

例如，我们有一个存储用户的数组，每个用户都有 `id` 和 `name` 字段。让我们找到 `id == 1` 的那个用户：

```
let users = [
  {id: 1, name: "John"}, 
  {id: 2, name: "Pete"}, 
  {id: 3, name: "Mary"}
];

let user = users.find(item => item.id == 1);

alert(user.name); // John
```

在现实生活中，对象数组是很常见的，所以 `find` 方法非常有用。

注意在这个例子中，我们传给了 `find` 一个单参数函数 `item => item.id == 1`。这很典型，并且 `find` 方法的其他参数很少使用。

`arr.findIndex ↗` 方法（与 `arr.find` 方法）基本上是一样的，但它返回找到元素的索引，而不是元素本身。并且在未找到任何内容时返回 `-1`。

filter

`find` 方法搜索的是使函数返回 `true` 的第一个（单个）元素。

如果需要匹配的有很多，我们可以使用 `arr.filter(fn) ↗`。

语法与 `find` 大致相同，但是 `filter` 返回的是所有匹配元素组成的数组：

```
let results = arr.filter(function(item, index, array) {
  // 如果 true item 被 push 到 results, 迭代继续
  // 如果什么都没找到，则返回空数组
});
```

例如：

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

// 返回前两个用户的数组
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

转换数组

让我们继续学习进行数组转换和重新排序的方法。

map

`arr.map ↗` 方法是最有用和经常使用的方法之一。

它对数组的每个元素都调用函数，并返回结果数组。

语法：

```
let result = arr.map(function(item, index, array) {
  // 返回新值而不是当前元素
})
```

例如，在这里我们将每个元素转换为它的字符串长度：

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

sort(fn)

`arr.sort` 方法对数组进行 **原位 (in-place)** 排序，更改元素的顺序。(译注：原位是指在此数组内，而非生成一个新数组。)

它还返回排序后的数组，但是返回值通常会被忽略，因为修改了 `arr` 本身。

语法：

```
let arr = [ 1, 2, 15 ];

// 该方法重新排列 arr 的内容
arr.sort();

alert( arr ); // 1, 15, 2
```

你有没有注意到结果有什么奇怪的地方？

顺序变成了 `1, 15, 2`。不对，但为什么呢？

这些元素默认情况下被按字符串进行排序。

从字面上看，所有元素都被转换为字符串，然后进行比较。对于字符串，按照词典顺序进行排序，实际上应该是 `"2" > "15"`。

要使用我们自己的排序顺序，我们需要提供一个函数作为 `arr.sort()` 的参数。

该函数应该比较两个任意值并返回：

```
function compare(a, b) {
  if (a > b) return 1; // 如果第一个值比第二个值大
  if (a == b) return 0; // 如果两个值相等
  if (a < b) return -1; // 如果第一个值比第二个值小
}
```

例如，按数字进行排序：

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr); // 1, 2, 15
```

现在结果符合预期了。

我们思考一下这儿发生了什么。`arr` 可以是由任何内容组成的数组，对吗？它可能包含数字、字符串、对象或其他任何内容。我们有一组 **一些元素**。要对其进行排序，我们需要一个 **排序函数** 来确认如何比较这些元素。默认是按字符串进行排序的。

`arr.sort(fn)` 方法实现了通用的排序算法。我们不需要关心它的内部工作原理（大多数情况下都是经过 [快速排序](#) 算法优化的）。它将遍历数组，使用提供的函数比较其元素并对其重新排序，我们所需要的就是提供执行比较的函数 `fn`。

顺便说一句，如果我们想知道要比较哪些元素 — 那么什么都不会阻止 `alert` 它们：

```
[1, -2, 15, 2, 0, 8].sort(function(a, b) {
  alert( a + " <> " + b );
});
```

该算法可以在此过程中，将一个元素与多个其他元素进行比较，但是它会尝试进行尽可能少的比较。

① 比较函数可以返回任何数字

实际上，比较函数只需要返回一个正数表示“大于”，一个负数表示“小于”。

通过这个原理我们可以编写更短的函数：

```
let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });

alert(arr); // 1, 2, 15
```

① 箭头函数最好

你还记得 [箭头函数](#) 吗？这里使用箭头函数会更加简洁：

```
arr.sort( (a, b) => a - b );
```

这与上面更长的版本完全相同。

1 使用 `localeCompare` for strings

你记得 [字符串比较](#) 算法吗？默认情况下，它通过字母的代码比较字母。

对于许多字母，最好使用 `str.localeCompare` 方法正确地对字母进行排序，例如 `Ö`。

例如，让我们用德语对几个国家/地区进行排序：

```
let countries = ['Österreich', 'Andorra', 'Vietnam'];

alert( countries.sort( (a, b) => a > b ? 1 : -1) ); // Andorra, Vietnam, Österreich (错的)

alert( countries.sort( (a, b) => a.localeCompare(b) ) ); // Andorra, Österreich, Vietnam (对的!)
```

reverse

`arr.reverse` ↗ 方法用于颠倒 `arr` 中元素的顺序。

例如：

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();

alert( arr ); // 5,4,3,2,1
```

它也会返回颠倒后的数组 `arr`。

split 和 join

举一个现实生活场景的例子。我们正在编写一个消息应用程序，并且该人员输入以逗号分隔的接收者列表：`John, Pete, Mary`。但对我们来说，名字数组比单个字符串舒适得多。怎么做才能获得这样的数组呢？

`str.split(delim)` ↗ 方法可以做到。它通过给定的分隔符 `delim` 将字符串分割成一个数组。

在下面的例子中，我们用“逗号后跟着一个空格”作为分隔符：

```
let names = 'Bilbo, Gandalf, Nazgul';

let arr = names.split(' ', ' ');

for (let name of arr) {
  alert(`A message to ${name}.`); // A message to Bilbo (和其他名字)
}
```

`split` 方法有一个可选的第二个数字参数 — 对数组长度的限制。如果提供了，那么额外的元素会被忽略。但实际上它很少使用：

```
let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(' ', ' ', 2);

alert(arr); // Bilbo, Gandalf
```

① 拆分为字母

调用带有空参数 `s` 的 `split(s)`，会将字符串拆分为字母数组：

```
let str = "test";
alert( str.split('') ); // t,e,s,t
```

`arr.join(glue)` 与 `split` 相反。它会在它们之间创建一串由 `glue` 粘合的 `arr` 项。

例如：

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];
let str = arr.join(';'); // 使用分号；将数组粘合成字符串
alert( str ); // Bilbo;Gandalf;Nazgul
```

reduce/reduceRight

当我们需要遍历一个数组时 — 我们可以使用 `forEach`, `for` 或 `for..of`。

当我们需要遍历并返回每个元素的数据时 — 我们可以使用 `map`。

`arr.reduce` 方法和 `arr.reduceRight` 方法和上面的种类差不多，但稍微复杂一点。它们用于根据数组计算单个值。

语法是：

```
let value = arr.reduce(function(accumulator, item, index, array) {
  // ...
}, [initial]);
```

该函数一个接一个地应用于所有数组元素，并将其结果“搬运（carry on）”到下一个调用。

参数：

- `accumulator` — 是上一个函数调用的结果，第一次等于 `initial`（如果提供了 `initial` 的话）。
- `item` — 当前的数组元素。
- `index` — 当前索引。
- `arr` — 数组本身。

应用函数时，上一个函数调用的结果将作为第一个参数传递给下一个函数。

因此，第一个参数本质上是累加器，用于存储所有先前执行的组合结果。最后，它成为 `reduce` 的结果。

听起来复杂吗？

掌握这个知识点的最简单的方法就是通过示例。

在这里，我们通过一行代码得到一个数组的总和：

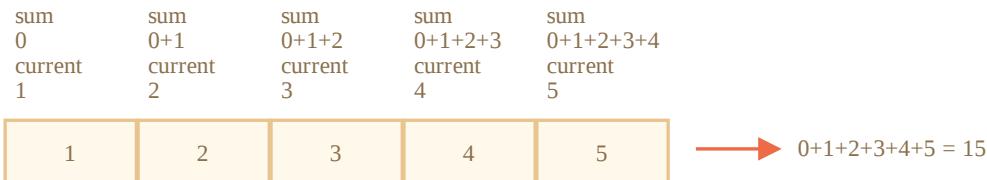
```
let arr = [1, 2, 3, 4, 5];  
  
let result = arr.reduce((sum, current) => sum + current, 0);  
  
alert(result); // 15
```

传递给 `reduce` 的函数仅使用了 2 个参数，通常这就足够了。

让我们看看细节，到底发生了什么。

1. 在第一次运行时，`sum` 的值为初始值 `initial` (`reduce` 的最后一个参数)，等于 0，`current` 是第一个数组元素，等于 1。所以函数运行的结果是 1。
2. 在第二次运行时，`sum = 1`，我们将第二个数组元素 (2) 与其相加并返回。
3. 在第三次运行中，`sum = 3`，我们继续把下一个元素与其相加，以此类推……。

计算流程：



或者以表格的形式表示，每一行代表的是对下一个数组元素的函数调用：

	sum	current	result
第 1 次调用	0	1	1
第 2 次调用	1	2	3
第 3 次调用	3	3	6
第 4 次调用	6	4	10
第 5 次调用	10	5	15

在这里，我们可以清楚地看到上一个调用的结果如何成为下一个调用的第一个参数。

我们也可以省略初始值：

```
let arr = [1, 2, 3, 4, 5];  
  
// 删除 reduce 的初始值 (没有 0)  
let result = arr.reduce((sum, current) => sum + current);  
  
alert(result); // 15
```

结果是一样的。这是因为如果没有初始值，那么 `reduce` 会将数组的第一个元素作为初始值，并从第二个元素开始迭代。

计算表与上面相同，只是去掉第一行。

但是这种使用需要非常小心。如果数组为空，那么在没有初始值的情况下调用 `reduce` 会导致错误。

例如：

```
let arr = [];

// Error: Reduce of empty array with no initial value
// 如果初始值存在，则 reduce 将为空 arr 返回它（即这个初始值）。
arr.reduce((sum, current) => sum + current);
```

所以建议始终指定初始值。

`arr.reduceRight` 和 `arr.reduce` 方法的功能一样，只是遍历为从右到左。

Array.isArray

数组是基于对象的，不构成单独的语言类型。

所以 `typeof` 不能帮助从数组中区分出普通对象：

```
alert(typeof {}); // object
alert(typeof []); // same
```

.....但是数组经常被使用，因此有一种特殊的方法用于判断：`Array.isArray(value)`。如果 `value` 是一个数组，则返回 `true`；否则返回 `false`。

```
alert(Array.isArray({})); // false
alert(Array.isArray([])); // true
```

大多数方法都支持“thisArg”

几乎所有调用函数的数组方法 – 比如 `find`, `filter`, `map`，除了 `sort` 是一个特例，都接受一个可选的附加参数 `thisArg`。

上面的部分中没有解释该参数，因为该参数很少使用。但是为了完整性，我们需要讲讲它。

以下是这些方法的完整语法：

```
arr.find(func, thisArg);
arr.filter(func, thisArg);
arr.map(func, thisArg);
// ...
// thisArg 是可选的最后一个参数
```

`thisArg` 参数的值在 `func` 中变为 `this`。

例如，在这里我们使用 `army` 对象方法作为过滤器，`thisArg` 用于传递上下文（passes the context）：

```

let army = {
  minAge: 18,
  maxAge: 27,
  canJoin(user) {
    return user.age >= this.minAge && user.age < this.maxAge;
  }
};

let users = [
  {age: 16},
  {age: 20},
  {age: 23},
  {age: 30}
];

// 找到 army.canJoin 返回 true 的 user
let soldiers = users.filter(army.canJoin, army);

alert(soldiers.length); // 2
alert(soldiers[0].age); // 20
alert(soldiers[1].age); // 23

```

如果在上面的示例中我们使用了 `users.filter(army.canJoin)`，那么 `army.canJoin` 将被作为独立函数调用，并且这时 `this=undefined`，从而会导致即时错误。

可以用 `users.filter(user => army.canJoin(user))` 替换对 `users.filter(army.canJoin, army)` 的调用。前者的使用频率更高，因为对于大多数人来说，它更容易理解。

总结

数组方法备忘单：

- 添加/删除元素：
 - `push(...items)` — 向尾端添加元素，
 - `pop()` — 从尾端提取一个元素，
 - `shift()` — 从首端提取一个元素，
 - `unshift(...items)` — 向首端添加元素，
 - `splice(pos, deleteCount, ...items)` — 从 `index` 开始删除 `deleteCount` 个元素，并在当前位置插入 `items`。
 - `slice(start, end)` — 创建一个新数组，将从位置 `start` 到位置 `end`（但不包括 `end`）的元素复制进去。
 - `concat(...items)` — 返回一个新数组：复制当前数组的所有元素，并向其中添加 `items`。如果 `items` 中的任意一项是一个数组，那么就取其元素。
- 搜索元素：
 - `indexOf/lastIndexOf(item, pos)` — 从位置 `pos` 开始搜索 `item`，搜索到则返回该项的索引，否则返回 `-1`。
 - `includes(value)` — 如果数组有 `value`，则返回 `true`，否则返回 `false`。
 - `find/filter(func)` — 通过 `func` 过滤元素，返回使 `func` 返回 `true` 的第一个值/所有值。

- `findIndex` 和 `find` 类似，但返回索引而不是值。
- 遍历元素：
 - `forEach(func)` — 对每个元素都调用 `func`，不返回任何内容。
- 转换数组：
 - `map(func)` — 根据对每个元素调用 `func` 的结果创建一个新数组。
 - `sort(func)` — 对数组进行原位（in-place）排序，然后返回它。
 - `reverse()` — 原位（in-place）反转数组，然后返回它。
 - `split/join` — 将字符串转换为数组并返回。
 - `reduce(func, initial)` — 通过对每个元素调用 `func` 计算数组上的单个值，并在调用之间传递中间结果。
- 其他：— `Array.isArray(arr)` 检查 `arr` 是否是一个数组。

请注意，`sort`，`reverse` 和 `splice` 方法修改的是数组本身。

这些是最常用的方法，它们覆盖 99% 的用例。但是还有其他几个：

- `arr.some(fn) ↗ /arr.every(fn) ↗` 检查数组。
与 `map` 类似，对数组的每个元素调用函数 `fn`。如果任何/所有结果为 `true`，则返回 `true`，否则返回 `false`。
- `arr.fill(value, start, end) ↗` — 从索引 `start` 到 `end`，用重复的 `value` 填充数组。
- `arr.copyWithin(target, start, end) ↗` — 将从位置 `start` 到 `end` 的所有元素复制到 `自身` 的 `target` 位置（覆盖现有元素）。

有关完整列表，请参阅 [手册 ↗](#)。

乍看起来，似乎有很多方法，很难记住。但实际上这比看起来要容易得多。

浏览这个备忘单，以了解这些方法。然后解决本章中的习题来进行练习，以便让你有数组方法的使用经验。

然后，每当你需要对数组进行某些操作，而又不知道怎么做的时候，请回到这儿，查看这个备忘单，然后找到正确的方法。示例将帮助你正确编写它。用不了多久，你就自然而然地记住这些方法了，根本不需要你死记硬背。

Iterable object（可迭代对象）

可迭代（Iterable） 对象是数组的泛化。这个概念是说任何对象都可以被定制为可在 `for..of` 循环中使用的对象。

数组是可迭代的。但不仅仅是数组。很多其他内建对象也都是可迭代的。例如字符串也是可迭代的。

如果从技术上讲，对象不是数组，而是表示某物的集合（列表，集合），`for..of` 是一个能够遍历它的很好的语法，因此，让我们来看看如何使其发挥作用。

Symbol.iterator

通过自己创建一个对象，我们就可以轻松地掌握可迭代的概念。

例如，我们有一个对象，它并不是数组，但是看上去很适合使用 `for..of` 循环。

比如一个 `range` 对象，它代表了一个数字区间：

```
let range = {
  from: 1,
  to: 5
};

// 我们希望 for..of 这样运行:
// for(let num of range) ... num=1,2,3,4,5
```

为了让 `range` 对象可迭代（也就让 `for..of` 可以运行）我们需要为对象添加一个名为 `Symbol.iterator` 的方法（一个专门用于使对象可迭代的内置 `symbol`）。

1. 当 `for..of` 循环启动时，它会调用这个方法（如果没找到，就会报错）。这个方法必须返回一个 **迭代器 (iterator)** —— 一个有 `next` 方法的对象。
2. 从此开始，`for..of` 仅适用于这个被返回的对象。
3. 当 `for..of` 循环希望取得下一个数值，它就调用这个对象的 `next()` 方法。
4. `next()` 方法返回的结果的格式必须是 `{done: Boolean, value: any}`，当 `done=true` 时，表示迭代结束，否则 `value` 是下一个值。

这是带有注释的 `range` 的完整实现：

```
let range = {
  from: 1,
  to: 5
};

// 1. for..of 调用首先会调用这个:
range[Symbol.iterator] = function() {

  // .....它返回迭代器对象 (iterator object) :
  // 2. 接下来，for..of 仅与此迭代器一起工作，要求它提供下一个值
  return {
    current: this.from,
    last: this.to,

    // 3. next() 在 for..of 的每一轮循环迭代中被调用
    next() {
      // 4. 它将会返回 {done:..., value :...} 格式的对象
      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
};

// 现在它可以运行了!
for (let num of range) {
  alert(num); // 1, 然后是 2, 3, 4, 5
}
```

请注意可迭代对象的核心功能：关注点分离。

- `range` 自身没有 `next()` 方法。
- 相反，是通过调用 `range[Symbol.iterator]()` 创建了另一个对象，即所谓的“迭代器”对象，并且它的 `next` 会为迭代生成值。

因此，迭代器对象和与其进行迭代的对象是分开的。

从技术上说，我们可以将它们合并，并使用 `range` 自身作为迭代器来简化代码。

就像这样：

```
let range = {
  from: 1,
  to: 5,

  [Symbol.iterator]() {
    this.current = this.from;
    return this;
  },

  next() {
    if (this.current <= this.to) {
      return { done: false, value: this.current++ };
    } else {
      return { done: true };
    }
  }
};

for (let num of range) {
  alert(num); // 1, 然后是 2, 3, 4, 5
}
```

现在 `range[Symbol.iterator]()` 返回的是 `range` 对象自身：它包括了必需的 `next()` 方法，并通过 `this.current` 记忆了当前的迭代进程。这样更短，对吗？是的。有时这样也可以。

但缺点是，现在不可能同时在对象上运行两个 `for..of` 循环了：它们将共享迭代状态，因为只有一个迭代器，即对象本身。但是两个并行的 `for..of` 是很罕见的，即使在异步情况下。

① 无穷迭代器（iterator）

无穷迭代器也是可能的。例如，将 `range` 设置为 `range.to = Infinity`，这时 `range` 则成为了无穷迭代器。或者我们可以创建一个可迭代对象，它生成一个无穷伪随机数序列。也是可能的。

`next` 没有什么限制，它可以返回越来越多的值，这是正常的。

当然，迭代这种对象的 `for..of` 循环将不会停止。但是我们可以通过使用 `break` 来停止它。

字符串是可迭代的

数组和字符串是使用最广泛的内建可迭代对象。

对于一个字符串，`for..of` 遍历它的每个字符：

```
for (let char of "test") {
  // 触发 4 次，每个字符一次
  alert( char ); // t, then e, then s, then t
}
```

对于代理对（surrogate pairs），它也能正常工作！（译注：这里的代理对也就指的是 UTF-16 的扩展字符）

```
let str = '𠮷';
for (let char of str) {
  alert( char ); // 𠮷, 然后是 𠮷
}
```

显式调用迭代器

为了更深层地了解底层知识，让我们来看看如何显式地使用迭代器。

我们将会采用与 `for..of` 完全相同的方式遍历字符串，但使用的是直接调用。这段代码创建了一个字符串迭代器，并“手动”从中获取值。

```
let str = "Hello";

// 和 for..of 做相同的事
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {
  let result = iterator.next();
  if (result.done) break;
  alert(result.value); // 一个接一个地输出字符
}
```

很少需要我们这样做，但是比 `for..of` 给了我们更多的控制权。例如，我们可以拆分迭代过程：迭代一部分，然后停止，做一些其他处理，然后再恢复迭代。

可迭代（iterable）和类数组（array-like）

有两个看起来很相似，但又有很大不同的正式术语。请你确保正确地掌握它们，以免造成混淆。

- **Iterable** 如上所述，是实现 `Symbol.iterator` 方法的对象。
- **Array-like** 是有索引和 `length` 属性的对象，所以它们看起来很像数组。

当我们使用 JavaScript 用于编写在浏览器或其他环境中的实际任务时，我们可能会遇到可迭代对象或类数组对象，或两者兼有。

例如，字符串即是可迭代的（`for..of` 对它们有效），又是类数组的（它们有数值索引和 `length` 属性）。

但是一个可迭代对象也许不是类数组对象。反之亦然，类数组对象可能不可迭代。

例如，上面例子中的 `range` 是可迭代的，但并非类数组对象，因为它没有索引属性，也没有 `length` 属性。

下面这个对象则是类数组的，但是不可迭代：

```
let arrayLike = { // 有索引和 length 属性 => 类数组对象
  0: "Hello",
  1: "World",
  length: 2
};

// Error (no Symbol.iterator)
for (let item of arrayLike) {}
```

可迭代对象和类数组对象通常都 **不是数组**，它们没有 `push` 和 `pop` 等方法。如果我们有一个这样的对象，并想像数组那样操作它，那就非常不方便。例如，我们想使用数组方法操作 `range`，应该如何实现呢？

Array.from

有一个全局方法 `Array.from` 可以接受一个可迭代或类数组的值，并从中获取一个“真正的”数组。然后我们就可以对其调用数组方法了。

例如：

```
let arrayLike = {
  0: "Hello",
  1: "World",
  length: 2
};

let arr = Array.from(arrayLike); // (*)
alert(arr.pop()); // World (pop 方法有效)
```

在 (*) 行的 `Array.from` 方法接受对象，检查它是一个可迭代对象或类数组对象，然后创建一个新数组，并将该对象的所有元素复制到这个新数组。

如果是可迭代对象，也是同样：

```
// 假设 range 来自上文的例子中
let arr = Array.from(range);
alert(arr); // 1,2,3,4,5 (数组的 toString 转化方法生效)
```

`Array.from` 的完整语法允许我们提供一个可选的“映射（mapping）”函数：

```
Array.from(obj[, mapFn, thisArg])
```

可选的第二个参数 `mapFn` 可以是一个函数，该函数会在对象中的元素被添加到数组前，被应用于每个元素，此外 `thisArg` 允许我们为该函数设置 `this`。

例如：

```
// 假设 range 来自上文例子中

// 求每个数的平方
let arr = Array.from(range, num => num * num);

alert(arr); // 1,4,9,16,25
```

现在我们用 `Array.from` 将一个字符串转换为单个字符的数组：

```
let str = '𠮷𠮷';

// 将 str 拆分为字符数组
let chars = Array.from(str);

alert(chars[0]); // 𠮷
alert(chars[1]); // 𠮷
alert(chars.length); // 2
```

与 `str.split` 方法不同，它依赖于字符串的可迭代特性。因此，就像 `for..of` 一样，可以正确地处理代理对（surrogate pair）。（译注：代理对也就是 UTF-16 扩展字符。）

技术上来说，它和下文做了同样的事：

```
let str = '𠮷𠮷';

let chars = []; // Array.from 内部执行相同的循环
for (let char of str) {
  chars.push(char);
}

alert(chars);
```

.....但 `Array.from` 精简很多。

我们甚至可以基于 `Array.from` 创建代理感知（surrogate-aware）的 `slice` 方法（译注：也就是能够处理 UTF-16 扩展字符的 `slice` 方法）：

```
function slice(str, start, end) {
  return Array.from(str).slice(start, end).join('');
}

let str = '𠮷𠮷𩿵';

alert( slice(str, 1, 3) ); // 𩿵

// 原生方法不支持识别代理对（译注：UTF-16 扩展字符）
alert( str.slice(1, 3) ); // 亂碼（两个不同 UTF-16 扩展字符碎片拼接的结果）
```

总结

可以应用 `for..of` 的对象被称为 可迭代的。

- 技术上来说，可迭代对象必须实现 `Symbol.iterator` 方法。
- `obj[Symbol.iterator]` 的结果被称为 迭代器 (`iterator`) 。由它处理进一步的迭代过程。
- 一个迭代器必须有 `next()` 方法，它返回一个 `{done: Boolean, value: any}` 对象，这里 `done:true` 表明迭代结束，否则 `value` 就是下一个值。
- `Symbol.iterator` 方法会被 `for..of` 自动调用，但我们也可以直接调用它。
- 内置的可迭代对象例如字符串和数组，都实现了 `Symbol.iterator` 。
- 字符串迭代器能够识别代理对 (`surrogate pair`) 。（译注：代理对也就是 UTF-16 扩展字符。）

有索引属性和 `length` 属性的对象被称为 **类数组对象**。这种对象可能还具有其他属性和方法，但是没有数组的内建方法。

如果我们仔细研究一下规范 —— 就会发现大多数内建方法都假设它们需要处理的是可迭代对象或者类数组对象，而不是“真正的”数组，因为这样抽象度更高。

`Array.from(obj[, mapFn, thisArg])` 将可迭代对象或类数组对象 `obj` 转化为真正的数组 `Array`，然后我们就可以对它应用数组的方法。可选参数 `mapFn` 和 `thisArg` 允许我们将函数应用到每个元素。

Map and Set (映射和集合)

我们已经了解了以下复杂的数据结构：

- 存储带键的数据 (`keyed`) 集合的对象。
- 存储有序集合的数组。

但这还不足以应对现实情况。这就是为什么存在 `Map` 和 `Set` 。

Map

`Map ↗` 是一个带键的数据项的集合，就像一个 `Object` 一样。但是它们最大的差别是 `Map` 允许任何类型的键 (`key`) 。

它的方法和属性如下：

- `new Map()` —— 创建 `map` 。
- `map.set(key, value)` —— 根据键存储值。
- `map.get(key)` —— 根据键来返回值，如果 `map` 中不存在对应的 `key`，则返回 `undefined` 。
- `map.has(key)` —— 如果 `key` 存在则返回 `true`，否则返回 `false` 。
- `map.delete(key)` —— 删除指定键的值。
- `map.clear()` —— 清空 `map` 。
- `map.size` —— 返回当前元素个数。

举个例子：

```
let map = new Map();

map.set('1', 'str1'); // 字符串键
map.set(1, 'num1'); // 数字键
map.set(true, 'bool1'); // 布尔值键

// 还记得普通的 Object 吗？它会将键转化为字符串
// Map 则会保留键的类型，所以下面这两个结果不同：
alert( map.get(1) ); // 'num1'
alert( map.get('1') ); // 'str1'

alert( map.size ); // 3
```

如我们所见，与对象不同，键不会被转换成字符串。键可以是任何类型。

i `map[key]` 不是使用 `Map` 的正确方式

虽然 `map[key]` 也有效，例如我们可以设置 `map[key] = 2`，这样会将 `map` 视为 JavaScript 的 plain object，因此它暗含了所有相应的限制（没有对象键等）。

所以我们应该使用 `map` 方法：`set` 和 `get` 等。

`Map` 还可以使用对象作为键。

例如：

```
let john = { name: "John" };

// 存储每个用户的来访次数
let visitsCountMap = new Map();

// john 是 Map 中的键
visitsCountMap.set(john, 123);

alert( visitsCountMap.get(john) ); // 123
```

使用对象作为键是 `Map` 最值得注意和重要的功能之一。对于字符串键，`Object`（普通对象）也能正常使用，但对于对象键则不行。

我们来尝试一下：

```
let john = { name: "John" };

let visitsCountObj = {}; // 尝试使用对象

visitsCountObj[john] = 123; // 尝试将 john 对象作为键

// 是写成了这样！
alert( visitsCountObj["[object Object]"] ); // 123
```

因为 `visitsCountObj` 是一个对象，它会将所有的键如 `john` 转换为字符串，所以我们得到字符串键 `"[object Object]"`。这显然不是我们想要的结果。

i Map 是怎么比较键的？

`Map` 使用 `SameValueZero ↗` 算法来比较键是否相等。它和严格等于 `==` 差不多，但区别是 `Nan` 被看成是等于 `Nan`。所以 `Nan` 也可以被用作键。

这个算法不能被改变或者自定义。

i 链式调用

每一次 `map.set` 调用都会返回 `map` 本身，所以我们可以进行“链式”调用：

```
map.set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

Map 迭代

如果要在 `map` 里使用循环，可以使用以下三个方法：

- `map.keys()` —— 遍历并返回所有的键（returns an iterable for keys）,
- `map.values()` —— 遍历并返回所有的值（returns an iterable for values）,
- `map.entries()` —— 遍历并返回所有的实体（returns an iterable for entries）`[key, value]`, `for..of` 在默认情况下使用的就是这个。

例如：

```
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion', 50]
]);

// 遍历所有的键 (vegetables)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // cucumber, tomatoes, onion
}

// 遍历所有的值 (amounts)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}

// 遍历所有的实体 [key, value]
for (let entry of recipeMap) { // 与 recipeMap.entries() 相同
  alert(entry); // cucumber, 500 (and so on)
}
```

① 使用插入顺序

迭代的顺序与插入值的顺序相同。与普通的 `Object` 不同，`Map` 保留了此顺序。

除此之外，`Map` 有内置的 `forEach` 方法，与 `Array` 类似：

```
// 对每个键值对 (key, value) 运行 forEach 函数
recipeMap.forEach( (value, key, map) => {
  alert(` ${key}: ${value}`); // cucumber: 500 etc
});
```

Object.entries：从对象创建 Map

当创建一个 `Map` 后，我们可以传入一个带有键值对的数组（或其它可迭代对象）来进行初始化，如下所示：

```
// 键值对 [key, value] 数组
let map = new Map([
  ['1', 'str1'],
  [1, 'num1'],
  [true, 'bool1']
]);

alert( map.get('1') ); // str1
```

如果我们想从一个已有的普通对象（plain object）来创建一个 `Map`，那么我们可以使用内建方法 `Object.entries(obj)` ↗，该返回对象的键/值对数组，该数组格式完全按照 `Map` 所需的格式。

所以可以像下面这样从一个对象创建一个 `Map`：

```
let obj = {
  name: "John",
  age: 30
};

let map = new Map(Object.entries(obj));

alert( map.get('name') ); // John
```

这里，`Object.entries` 返回键/值对数组：`[["name", "John"], ["age", 30]]`。这就是 `Map` 所需要的格式。

Object.fromEntries：从 Map 创建对象

我们刚刚已经学习了如何使用 `Object.entries(obj)` 从普通对象（plain object）创建 `Map`。

`Object.fromEntries` 方法的作用是相反的：给定一个具有 `[key, value]` 键值对的数组，它会根据给定数组创建一个对象：

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// 现在 prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

我们可以使用 `Object.fromEntries` 从 `Map` 得到一个普通对象（plain object）。

例如，我们在 `Map` 中存储了一些数据，但是我们需要把这些数据传给需要普通对象（plain object）的第三方代码。

我们来开始：

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // 创建一个普通对象 (plain object) (*)

// 完成了!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

调用 `map.entries()` 将返回键/值对数组，这刚好是 `Object.fromEntries` 所需要的格式。

我们可以把带 `(*)` 这一行写得更短：

```
let obj = Object.fromEntries(map); // 省掉 .entries()
```

上面的代码作用也是一样的，因为 `Object.fromEntries` 期望得到一个可迭代对象作为参数，而不一定是数组。并且 `map` 的标准迭代会返回跟 `map.entries()` 一样的键/值对。因此，我们可以获得一个普通对象（plain object），其键/值对与 `map` 相同。

Set

`Set` 是一个特殊的类型集合 — “值的集合”（没有键），它的每一个值只能出现一次。

它的主要方法如下：

- `new Set(iterable)` —— 创建一个 `set`，如果提供了一个 `iterable` 对象（通常是数组），将会从数组里面复制值到 `set` 中。
- `set.add(value)` —— 添加一个值，返回 `set` 本身
- `set.delete(value)` —— 删除值，如果 `value` 在这个方法调用的时候存在则返回 `true`，否则返回 `false`。

- `set.has(value)` —— 如果 `value` 在 `set` 中，返回 `true`，否则返回 `false`。
- `set.clear()` —— 清空 `set`。
- `set.size` —— 返回元素个数。

它的主要特点是，重复使用同一个值调用 `set.add(value)` 并不会发生什么改变。这就是 `Set` 里面的每一个值只出现一次的原因。

例如，我们有客人来访，我们想记住他们每一个人。但是已经来访过的客人再次来访，不应造成重复记录。每个访客必须只被“计数”一次。

`Set` 可以帮助我们解决这个问题：

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// visits, 一些访客来访好几次
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set 只保留不重复的值
alert( set.size ); // 3

for (let user of set) {
  alert(user.name); // John (然后 Pete 和 Mary)
}
```

`Set` 的替代方法可以是一个用户数组，用 `arr.find ↗` 在每次插入值时检查是否重复。但是这样性能会很差，因为这个方法会遍历整个数组来检查每个元素。`Set` 内部对唯一性检查进行了更好的优化。

Set 迭代（iteration）

我们可以使用 `for..of` 或 `forEach` 来遍历 `Set`:

```
let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);

// 与 forEach 相同:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

注意一件有趣的事儿。`forEach` 的回调函数有三个参数：一个 `value`，然后是 **同一个值** `valueAgain`，最后是目标对象。没错，同一个值在参数里出现了两次。

`forEach` 的回调函数有三个参数，是为了与 `Map` 兼容。当然，这看起来确实有些奇怪。但是这对在特定情况下轻松地用 `Set` 代替 `Map` 很有帮助，反之亦然。

`Map` 中用于迭代的方法在 `Set` 中也同样支持：

- `set.keys()` —— 遍历并返回所有的值（returns an iterable object for values）,
- `set.values()` —— 与 `set.keys()` 作用相同，这是为了兼容 `Map`,
- `set.entries()` —— 遍历并返回所有的实体（returns an iterable object for entries）`[value, value]`，它的存在也是为了兼容 `Map`。

总结

`Map` — 是一个带键的数据项的集合。

方法和属性如下：

- `new Map([iterable])` —— 创建 map，可选择带有 `[key, value]` 对的 `iterable`（例如数组）来进行初始化。
- `map.set(key, value)` —— 根据键存储值。
- `map.get(key)` —— 根据键来返回值，如果 `map` 中不存在对应的 `key`，则返回 `undefined`。
- `map.has(key)` —— 如果 `key` 存在则返回 `true`，否则返回 `false`。
- `map.delete(key)` —— 删除指定键的值。
- `map.clear()` —— 清空 map。
- `map.size` —— 返回当前元素个数。

与普通对象 `Object` 的不同点：

- 任何键、对象都可以作为键。
- 有其他的便捷方法，如 `size` 属性。

`Set` —— 是一组唯一值的集合。

方法和属性：

- `new Set([iterable])` —— 创建 set，可选择带有 `iterable`（例如数组）来进行初始化。
- `set.add(value)` —— 添加一个值（如果 `value` 存在则不做任何修改），返回 `set` 本身。
- `set.delete(value)` —— 删减值，如果 `value` 在这个方法调用的时候存在则返回 `true`，否则返回 `false`。
- `set.has(value)` —— 如果 `value` 在 `set` 中，返回 `true`，否则返回 `false`。
- `set.clear()` —— 清空 `set`。
- `set.size` —— 元素的个数。

在 `Map` 和 `Set` 中迭代总是按照值插入的顺序进行的，所以我们不能说这些集合是无序的，但是我们不能对元素进行重新排序，也不能直接按其编号来获取元素。

WeakMap and WeakSet（弱映射和弱集合）

我们从前面的 [垃圾回收](#) 章节中知道，JavaScript 引擎在值可访问（并可能被使用）时将其存储在内存中。

例如：

```
let john = { name: "John" };

// 该对象能被访问, john 是它的引用

// 覆盖引用
john = null;

// 该对象将被从内存中清除
```

通常，当对象、数组这类数据结构在内存中时，它们的子元素，如对象的属性、数组的元素都是可以访问的。

例如，如果把一个对象放入到数组中，那么只要这个数组存在，那么这个对象也就存在，即使没有其他对该对象的引用。

就像这样：

```
let john = { name: "John" };

let array = [ john ];

john = null; // 覆盖引用

// john 被存储在数组里, 所以它不会被垃圾回收机制回收
// 我们可以通过 array[0] 来获取它
```

类似的，如果我们使用对象作为常规 `Map` 的键，那么当 `Map` 存在时，该对象也将存在。它会占用内存，并且应该不会被（垃圾回收机制）回收。

例如：

```
let john = { name: "John" };

let map = new Map();
map.set(john, "...");

john = null; // 覆盖引用

// john 被存储在 map 中,
// 我们可以使用 map.keys() 来获取它
```

`WeakMap` 在这方面有着根本上的不同。它不会阻止垃圾回收机制对作为键的对象（key object）的回收。

让我们通过例子来看看这指的到底是什么。

WeakMap

`WeakMap` 和 `Map` 的第一个不同点就是，`WeakMap` 的键必须是对象，不能是原始值：

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // 正常工作（以对象作为键）

// 不能使用字符串作为键
weakMap.set("test", "Whoops"); // Error, 因为 "test" 不是一个对象
```

现在，如果我们在 `weakMap` 中使用一个对象作为键，并且没有其他对这个对象的引用 — 该对象将会被从内存（和`map`）中自动清除。

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // 覆盖引用

// john 被从内存中删除了！
```

与上面常规的 `Map` 的例子相比，现在如果 `john` 仅仅是作为 `WeakMap` 的键而存在 — 它将会被从 `map`（和内存）中自动删除。

`WeakMap` 不支持迭代以及 `keys()`, `values()` 和 `entries()` 方法。所以没有办法获取 `WeakMap` 的所有键或值。

`WeakMap` 只有以下的方法：

- `weakMap.get(key)`
- `weakMap.set(key, value)`
- `weakMap.delete(key)`
- `weakMap.has(key)`

为什么会有这种限制呢？这是技术的原因。如果一个对象丢失了其它所有引用（就像上面示例中的 `john`），那么它就会被垃圾回收机制自动回收。但是在从技术的角度并不能准确知道 **何时会被回收**。

这些都是由 JavaScript 引擎决定的。JavaScript 引擎可能会选择立即执行内存清理，如果现在正在发生很多删除操作，那么 JavaScript 引擎可能就会选择等一等，稍后再进行内存清理。因此，从技术上讲，`WeakMap` 的当前元素的数量是未知的。JavaScript 引擎可能清理了其中的垃圾，可能没清理，也可能清理了一部分。因此，暂不支持访问 `WeakMap` 的所有键/值的方法。

那么，在哪里我们会需要这样的数据结构呢？

使用案例：额外的数据

`WeakMap` 的主要应用场景是 **额外数据的存储**。

假如我们正在处理一个“属于”另一个代码的一个对象，也可能是第三方库，并想存储一些与之相关的数据，那么这些数据就应该与这个对象共存亡 — 这时候 `WeakMap` 正是我们所需要的利器。

我们将这些数据放到 `WeakMap` 中，并使用该对象作为这些数据的键，那么当该对象被垃圾回收机制回收后，这些数据也会被自动清除。

```
weakMap.set(john, "secret documents");
// 如果 john 消失, secret documents 将会被自动清除
```

让我们来看一个例子。

例如，我们有用于处理用户访问计数的代码。收集到的信息被存储在 `map` 中：一个用户对象作为键，其访问次数为值。当一个用户离开时（该用户对象将被垃圾回收机制回收），这时我们就不再需要他的访问次数了。

下面是一个使用 `Map` 的计数函数的例子：

```
// visitsCount.js
let visitsCountMap = new Map(); // map: user => visits count

// 递增用户来访次数
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

下面是其他部分的代码，可能是使用它的其它代码：

```
// main.js
let john = { name: "John" };

countUser(john); // count his visits

// 不久之后, john 离开了
john = null;
```

现在 `john` 这个对象应该被垃圾回收，但他仍在内存中，因为它是 `visitsCountMap` 中的一个键。

当我们移除用户时，我们需要清理 `visitsCountMap`，否则它将在内存中无限增大。在复杂的架构中，这种清理会成为一项繁重的任务。

我们可以通过使用 `WeakMap` 来避免这样的问题：

```
// visitsCount.js
let visitsCountMap = new WeakMap(); // weakmap: user => visits count

// 递增用户来访次数
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

现在我们不需要去清理 `visitsCountMap` 了。当 `john` 对象变成不可访问时，即便它是 `WeakMap` 里的一个键，它也会连同它作为 `WeakMap` 里的键所对应的信息一同被从内存中删除。

使用案例：缓存

另外一个普遍的例子是缓存：当一个函数的结果需要被记住（“缓存”），这样在后续的对同一个对象的调用时，就可以重用这个被缓存的结果。

我们可以使用 `Map` 来存储结果，就像这样：

```
// □ cache.js
let cache = new Map();

// 计算并记住结果
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calculations of the result for */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// 现在我们在其它文件中使用 process()

// □ main.js
let obj = {/* 假设我们有个对象 */};

let result1 = process(obj); // 计算完成

// .....稍后，来自代码的另外一个地方.....
let result2 = process(obj); // 取自缓存的被记忆的结果

// .....稍后，我们不再需要这个对象时：
obj = null;

alert(cache.size); // 1 (啊！该对象依然在 cache 中，并占据着内存！)
```

对于多次调用同一个对象，它只需在第一次调用时计算出结果，之后的调用可以直接从 `cache` 中获取。这样做的缺点是，当我们不再需要这个对象的时候需要清理 `cache`。

如果我们用 `WeakMap` 替代 `Map`，这个问题便会消失：当对象被垃圾回收时，对应的缓存的结果也会被自动地从内存中清除。

```
// □ cache.js
let cache = new WeakMap();

// 计算并记结果
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* calculate the result for */ obj;
```

```

        cache.set(obj, result);
    }

    return cache.get(obj);
}

// □ main.js
let obj = /* some object */;

let result1 = process(obj);
let result2 = process(obj);

// .....稍后，我们不再需要这个对象时：
obj = null;

// 无法获取 cache.size，因为它是一个 WeakMap,
// 要么是 0，或即将变为 0
// 当 obj 被垃圾回收，缓存的数据也会被清除

```

WeakSet

`WeakSet` 的表现类似：

- 与 `Set` 类似，但是我们只能向 `WeakSet` 添加对象（而不能是原始值）。
- 对象只有在其它某个（些）地方能被访问的时候，才能留在 `set` 中。
- 跟 `Set` 一样，`WeakSet` 支持 `add`，`has` 和 `delete` 方法，但不支持 `size` 和 `keys()`，并且不可迭代。

变“弱（weak）”的同时，它也可以作为额外的存储空间。但并非针对任意数据，而是针对“是/否”的事实。`WeakSet` 的元素可能代表着有关该对象的某些信息。

例如，我们可以将用户添加到 `WeakSet` 中，以追踪访问过我们网站的用户：

```

let visitedSet = new WeakSet();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

visitedSet.add(john); // John 访问了我们
visitedSet.add(pete); // 然后是 Pete
visitedSet.add(john); // John 再次访问

// visitedSet 现在有两个用户了

// 检查 John 是否来访过？
alert(visitedSet.has(john)); // true

// 检查 Mary 是否来访过？
alert(visitedSet.has(mary)); // false

john = null;

// visitedSet 将被自动清理

```

`WeakMap` 和 `WeakSet` 最明显的局限性就是不能迭代，并且无法获取所有当前内容。那样可能会造成不便，但是并不会阻止 `WeakMap/WeakSet` 完成其主要工作 — 成为在其它地方管理/存储“额外”的对象数据。

总结

`WeakMap` 是类似于 `Map` 的集合，它仅允许对象作为键，并且一旦通过其他方式无法访问它们，便会将它们与其关联值一同删除。

`WeakSet` 是类似于 `Set` 的集合，它仅存储对象，并且一旦通过其他方式无法访问它们，便会将其删除。

它们都不支持引用所有键或其计数的方法和属性。仅允许单个操作。

`WeakMap` 和 `WeakSet` 被用作“主要”对象存储之外的“辅助”数据结构。一旦将对象从主存储器中删除，如果该对象仅被用作 `WeakMap` 或 `WeakSet` 的键，那么它将被自动清除。

Object.keys, values, entries

对各个数据结构的学习至此告一段落，下面让我们讨论一下如何迭代它们。

在前面的章节中，我们认识了 `map.keys()`, `map.values()` 和 `map.entries()` 方法。

这些方法是通用的，有一个共同的约定来将它们用于各种数据结构。如果我们创建一个我们自己的数据结构，我们也应该实现这些方法。

它们支持：

- `Map`
- `Set`
- `Array`

普通对象也支持类似的方法，但是语法上有一些不同。

Object.keys, values, entries

对于普通对象，下列这些方法是可用的：

- `Object.keys(obj)` ↗ —— 返回一个包含该对象所有的键的数组。
- `Object.values(obj)` ↗ —— 返回一个包含该对象所有的值的数组。
- `Object.entries(obj)` ↗ —— 返回一个包含该对象所有 `[key, value]` 键值对的数组。

.....但是请注意区别（比如说跟 `map` 的区别）：

	<code>Map</code>	<code>Object</code>
调用语法	<code>map.keys()</code>	<code>Object.keys(obj)</code> ，而不是 <code>obj.keys()</code>
返回值	可迭代项	“真正的”数组

第一个区别是，对于对象我们使用的调用语法是 `Object.keys(obj)`，而不是 `obj.keys()`。

为什么会这样？主要原因是灵活性。请记住，在 JavaScript 中，对象是所有复杂结构的基础。因此，我们可能有一个自己创建的对象，比如 `data`，并实现了它自己的 `data.values()` 方法。同时，我们依然可以对它调用 `Object.values(data)` 方法。

第二个区别是 `Object.*` 方法返回的是“真正的”数组对象，而不只是一个可迭代项。这主要是历史原因。

举个例子：

```
let user = {  
    name: "John",  
    age: 30  
};
```

- `Object.keys(user) = ["name", "age"]`
- `Object.values(user) = ["John", 30]`
- `Object.entries(user) = [["name", "John"], ["age", 30]]`

这里有一个使用 `Object.values` 来遍历属性值的例子：

```
let user = {  
    name: "John",  
    age: 30  
};  
  
// 遍历所有的值  
for (let value of Object.values(user)) {  
    alert(value); // John, then 30  
}
```

⚠️ `Object.keys/values/entries` 会忽略 `symbol` 属性

就像 `for..in` 循环一样，这些方法会忽略使用 `Symbol(...)` 作为键的属性。

通常这很方便。但是，如果我们也想要 `Symbol` 类型的键，那么这儿有一个单独的方法 `Object.getOwnPropertySymbols` ↗，它会返回一个只包含 `Symbol` 类型的键的数组。另外，还有一种方法 `Reflect.ownKeys(obj)` ↗，它会返回 所有 键。

转换对象

对象缺少数组存在的许多方法，例如 `map` 和 `filter` 等。

如果我们想应用它们，那么我们可以在 `Object.entries` 之后使用 `Object.fromEntries`：

1. 使用 `Object.entries(obj)` 从 `obj` 获取由键/值对组成的数组。
2. 对该数组使用数组方法，例如 `map`。
3. 对结果数组使用 `Object.fromEntries(array)` 方法，将结果转回成对象。

例如，我们有一个带有价格的对象，并想将它们加倍：

```
let prices = {  
    banana: 1,  
    orange: 2,  
    meat: 4,  
};  
  
let doublePrices = Object.fromEntries(  
    // 转换为数组，之后使用 map 方法，然后通过 fromEntries 再转回到对象  
    Object.entries(prices).map(([key, value]) => [key, value * 2])  
);  
  
alert(doublePrices.meat); // 8
```

乍一看，可能看起来很困难，但是使用一次或两次后，就很容易理解了。我们可以通过这种方式建立强大的转换链。

解构赋值

JavaScript 中最常用的两种数据结构是 `Object` 和 `Array`。

对象让我们能够创建通过键来存储数据项的单个实体，数组则让我们能够将数据收集到一个有序的集合中。

但是，当我们把它们传递给函数时，它可能不需要一个整体的对象/数组，而是需要单个块。

解构赋值 是一种特殊的语法，它使我们可以将数组或对象“拆包”为到一系列变量中，因为有时候使用变量更加方便。解构操作对那些具有很多参数和默认值等的函数也很奏效。

数组解构

下面是一个将数组解构到变量中的例子：

```
// 我们有一个存放了名字和姓氏的数组  
let arr = ["Ilya", "Kantor"]  
  
// 解构赋值  
// sets firstName = arr[0]  
// and surname = arr[1]  
let [firstName, surname] = arr;  
  
alert(firstName); // Ilya  
alert(surname); // Kantor
```

现在我们就可以针对这些变量进行操作，而不是针对原来的数组元素。

当与 `split` 函数（或其他返回值是数组的函数）结合使用时，看起来就更优雅了：

```
let [firstName, surname] = "Ilya Kantor".split(' ');
```

i “解构”并不意味着“破坏”

这种语法叫做“解构赋值”，因为它通过将结构中的各元素复制到变量中来达到“解构”的目的。但数组本身是没有被修改的。

这只是下面这些代码的更精简的写法而已：

```
// let [firstName, surname] = arr;  
let firstName = arr[0];  
let surname = arr[1];
```

i 忽略使用逗号的元素

数组中不想要的元素也可以通过添加额外的逗号来把它丢弃：

```
// 不需要第二个元素  
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];  
  
alert( title ); // Consul
```

在上面的代码中，数组的第二个元素被跳过了，第三个元素被赋值给了 `title` 变量，数组中剩下的元素也都被跳过了（因为在这没有对应给它们的变量）。

i 等号右侧可以是任何可迭代对象

.....实际上，我们可以将其与任何可迭代的数组一起使用，而不仅限于数组：

```
let [a, b, c] = "abc"; // ["a", "b", "c"]  
let [one, two, three] = new Set([1, 2, 3]);
```

i 赋值给等号左侧的任何内容

我们可以在等号左侧使用任何“可以被赋值的”东西。

例如，一个对象的属性：

```
let user = {};  
[user.name, user.surname] = "Ilya Kantor".split(' ');\n  
alert(user.name); // Ilya
```

① 与 .entries() 方法进行循环操作

在前面的章节中我们已经见过了 `Object.entries(obj)` ↗ 方法。

我们可以将 `.entries()` 方法与解构语法一同使用，来遍历一个对象的“键—值”对：

```
let user = {  
    name: "John",  
    age: 30  
};  
  
// 循环遍历键-值对  
for (let [key, value] of Object.entries(user)) {  
    alert(` ${key}: ${value}`); // name: John, then age: 30  
}
```

.....对于 `map` 对象也类似：

```
let user = new Map();  
user.set("name", "John");  
user.set("age", "30");  
  
for (let [key, value] of user) {  
    alert(` ${key}: ${value}`); // name: John, then age: 30  
}
```

剩余的 ‘...’

如果我们不只是要获得第一个值，还要将后续的所有元素都收集起来 — 我们可以使用三个点 “...” 来再加一个参数来接收“剩余的”元素：

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];  
  
alert(name1); // Julius  
alert(name2); // Caesar  
  
// 请注意，`rest` 的类型是数组  
alert(rest[0]); // Consul  
alert(rest[1]); // of the Roman Republic  
alert(rest.length); // 2
```

`rest` 的值就是数组中剩下的元素组成的数组。不一定要使用变量名 `rest`，我们也可以使用其他的变量名，只要确保它前面有三个点，并且在解构赋值的最后一个参数位置上就行了。

默认值

如果赋值语句中，变量的数量多于数组中实际元素的数量，赋值不会报错。未赋值的变量被认为是 `undefined`：

```
let [firstName, surname] = [];
```

```
alert(firstName); // undefined
alert(surname); // undefined
```

如果我们想要一个“默认”值给未赋值的变量，我们可以使用 `=` 来提供：

```
// 默认值
let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name);    // Julius (来自数组的值)
alert(surname); // Anonymous (默认值被使用了)
```

默认值可以是更加复杂的表达式甚至可以是函数调用，这些表达式或函数只会在这个变量未被赋值的时候才会被计算。

举个例子，我们使用了 `prompt` 函数来提供两个默认值，但它只会在未被赋值的那个变量上进行调用：

```
// 只会提示输入姓氏
let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];

alert(name);    // Julius (来自数组)
alert(surname); // 你输入的值
```

对象解构

解构赋值同样适用于对象。

基本语法是：

```
let {var1, var2} = {var1:..., var2:...}
```

在等号右侧有一个已经存在的对象，我们想把它拆开到变量中。等号左侧包含了对象相应属性的一个“模式（pattern）”。在简单的情况下，等号左侧的就是 `{...}` 中的变量名列表。

举个例子：

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

alert(title); // Menu
alert(width); // 100
alert(height); // 200
```

属性 `options.title`、`options.width` 和 `options.height` 值被赋给了对应的变量。变量的顺序并不重要，下面这个代码也奏效：

```
// 改变 let {...} 中元素的顺序
let {height, width, title} = { title: "Menu", height: 200, width: 100 }
```

等号左侧的模式（pattern）可以更加复杂，并且指定了属性和变量之间的映射关系。

如果我们想把一个属性赋值给另一个名字的变量，比如把 `options.width` 属性赋值给变量 `w`，那么我们可以使用冒号来指定：

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;

// width -> w
// height -> h
// title -> title

alert(title); // Menu
alert(w); // 100
alert(h); // 200
```

冒号表示“什么值：赋值给谁”。上面的例子中，属性 `width` 被赋值给了 `w`，属性 `height` 被赋值给了 `h`，属性 `title` 被赋值给了同名变量。

对于可能缺失的属性，我们可以使用 `"=` 设置默认值，如下所示：

```
let options = {
  title: "Menu"
};

let {width = 100, height = 200, title} = options;

alert(title); // Menu
alert(width); // 100
alert(height); // 200
```

就像数组或函数参数一样，默认值可以是任意表达式甚至可以是函数调用。它们只会在未提供对应的值时才会被计算/调用。

在下面的代码中，`prompt` 提示输入 `width` 值，但不会提示输入 `title` 值：

```
let options = {
  title: "Menu"
};
```

```
let {width = prompt("width?"), title = prompt("title?")} = options;

alert(title); // Menu
alert(width); // (无论 prompt 的结果是什么)
```

我们还可以将冒号和等号结合起来:

```
let options = {
  title: "Menu"
};

let {width: w = 100, height: h = 200, title} = options;

alert(title); // Menu
alert(w); // 100
alert(h); // 200
```

如果我们有一个具有很多属性的复杂对象，那么我们可以只提取所需的内容:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// 仅提取 title 作为变量
let { title } = options;

alert(title); // Menu
```

剩余模式 (pattern) "..."

如果对象拥有的属性数量比我们提供的变量数量还多，该怎么办？我们可以只取其中的某一些属性，然后把“剩余的”赋值到其他地方吗？

我们可以使用剩余模式 (pattern)，就像我们对数组那样。一些较旧的浏览器不支持此功能（例如，使用 **Babel** 对其进行填充），但可以在现代浏览器中使用。

看起来就像这样：

```
let options = {
  title: "Menu",
  height: 200,
  width: 100
};

// title = 名为 title 的属性
// rest = 存有剩余属性的对象
let {title, ...rest} = options;

// 现在 title="Menu", rest={height: 200, width: 100}
alert(rest.height); // 200
alert(rest.width); // 100
```

① 不使用 `let` 时的陷阱

在上面的示例中，变量都是在赋值中通过正确方式声明的：`let ... = ...`。当然，我们也可以使用已有的变量，而不用 `let`，但这里有一个陷阱。

以下代码无法正常运行：

```
let title, width, height;

// 这一行发生了错误
{title, width, height} = {title: "Menu", width: 200, height: 100};
```

问题在于 JavaScript 把主代码流（即不在其他表达式中）的 `{...}` 当做一个代码块。这样的代码块可以用于对语句分组，如下所示：

```
{
  // 一个代码块
  let message = "Hello";
  // ...
  alert( message );
}
```

因此，这里 JavaScript 假定我们有一个代码块，这就是报错的原因。我们需要解构它。

为了告诉 JavaScript 这不是一个代码块，我们可以把整个赋值表达式用括号 `(...)` 包起来：

```
let title, width, height;

// 现在就可以了
({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu
```

嵌套解构

如果一个对象或数组嵌套了其他的对象和数组，我们可以在等号左侧使用更复杂的模式（pattern）来提取更深层的数据。

在下面的代码中，`options` 的属性 `size` 是另一个对象，属性 `items` 是另一个数组。赋值语句中等号左侧的模式（pattern）具有相同的结构以从中提取值：

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};
```

```
// 为了清晰起见，解构赋值语句被写成多行的形式
let {
  size: { // 把 size 赋值到这里
    width,
    height
  },
  items: [item1, item2], // 把 items 赋值到这里
  title = "Menu" // 在对象中不存在（使用默认值）
} = options;

alert(title); // Menu
alert(width); // 100
alert(height); // 200
alert(item1); // Cake
alert(item2); // Donut
```

对象 `options` 的所有属性，除了 `extra` 属性在等号左侧不存在，都被赋值给了对应的变量：

```
let {
  size: {
    width,
    height
  },
  items: [item1, item2],
  title = "Menu"
}

let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
}
```

最终，我们得到了 `width`、`height`、`item1`、`item2` 和具有默认值的 `title` 变量。

注意，`size` 和 `items` 没有对应的变量，因为我们取的是它们的内容。

智能函数参数

有时，一个函数有很多参数，其中大部分的参数都是可选的。对用户界面来说更是如此。想象一个创建菜单的函数。它可能具有宽度参数，高度参数，标题参数和项目列表等。

下面是实现这种函数的一个很不好的写法：

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) {
  // ...
}
```

在实际开发中存在一个问题就是你怎么记得住这么多参数的顺序。通常集成开发环境工具（IDE）会尽力帮助我们，特别是当代码有良好的文档注释的时候，但是……另一个问题就是，当大部分的参数采用默认值就好了的情况下，怎么调用这个函数。

难道像这样？

```
// 在采用默认值就可以的位置设置 undefined
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

这太难看了。而且，当我们处理更多参数的时候可读性会变得很差。

解构赋值语法前来救援！

我们可以把所有参数当作一个对象来传递，然后函数马上把这个对象解构成多个变量：

```
// 我们传递一个对象给函数
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

// .....然后函数马上把对象展开成变量
function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {
  // title, items - 提取于 options,
  // width, height - 使用默认值
  alert(` ${title} ${width} ${height}`); // My Menu 200 100
  alert( items ); // Item1, Item2
}

showMenu(options);
```

我们同样可以使用带有嵌套对象和冒号映射的更加复杂的解构：

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

function showMenu({
  title = "Untitled",
  width: w = 100, // width goes to w
  height: h = 200, // height goes to h
  items: [item1, item2] // items first element goes to item1, second to item2
}) {
  alert(` ${title} ${w} ${h}`); // My Menu 100 200
  alert( item1 ); // Item1
  alert( item2 ); // Item2
}

showMenu(options);
```

完整语法和解构赋值是一样的：

```
function({
  incomingProperty: varName = defaultValue
  ...
})
```

对于参数对象，属性 `incomingProperty` 对应的变量是 `varName`，默认值是 `defaultValue`。

请注意，这种解构假定了 `showMenu()` 函数确实存在参数。如果我们想让所有的参数都使用默认值，那我们应该传递一个空对象：

```
showMenu({}); // 不错，所有值都取默认值  
showMenu(); // 这样会导致错误
```

我们可以通过指定空对象 `{}` 为整个参数对象的默认值来解决这个问题：

```
function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {  
  alert(` ${title} ${width} ${height}`);  
}  
  
showMenu(); // Menu 100 200
```

在上面的代码中，整个参数对象的默认是 `{}`，因此总会有内容可以用来解构。

总结

- 解构赋值可以立即将一个对象或数组映射到多个变量上。
- 解构对象的完整语法：

```
let {prop : varName = default, ...rest} = object
```

这表示属性 `prop` 会被赋值给变量 `varName`，如果没有这个属性的话，就会使用默认值 `default`。

没有对应映射的属性被复制到 `rest` 对象。

- 解构数组的完整语法：

```
let [item1 = default, item2, ...rest] = array
```

数组的第一个元素被赋值给 `item1`，第二个元素被赋值给 `item2`，剩下的所有元素被复制到另一个数组 `rest`。

- 从嵌套数组/对象中提取数据也是可以的，此时等号左侧必须和等号右侧有相同的结构。

日期和时间

让我一起学习一个新的内建对象：[日期 \(Date\)](#)。该对象存储日期和时间，并提供了日期/时间的管理方法。

例如，我们可以使用它来存储创建/修改时间，或者用来测量时间，再或者仅用来打印当前时间。

创建

创建一个新的 `Date` 对象，只需要调用 `new Date()`，在调用时可以带有下面这些参数之一：

```
new Date()
```

不带参数 —— 创建一个表示当前日期和时间的 `Date` 对象：

```
let now = new Date();
alert( now ); // 显示当前的日期/时间
```

`new Date(milliseconds)`

创建一个 `Date` 对象，其时间等于 `1970-01-01 00:00:00 UTC+0` 再过一毫秒（`1/1000` 秒）。

```
// 0 表示 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// 现在增加 24 小时，得到 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

传入的整数参数代表的是自 `1970-01-01 00:00:00` 以来经过的毫秒数，该整数被称为 **时间戳**。

这是一种日期的轻量级数字表示形式。我们通常使用 `new Date(timestamp)` 通过时间戳来创建日期，并可以使用 `date.getTime()` 将现有的 `Date` 对象转化为时间戳（下文会讲到）。

在 `01.01.1970` 之前的日期带有负的时间戳，例如：

```
// 31 Dec 1969
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert( Dec31_1969 );
```

`new Date(datestring)`

如果只有一个参数，并且是字符串，那么它会被自动解析。该算法与 `Date.parse` 所使用的算法相同，我们将在下文中进行介绍。

```
let date = new Date("2017-01-26");
alert(date); // 时间未设置，因此假定为格林尼治标准时间午夜 50 // 根据运行代码的时区进行调整
// 该时间未被设定，因此被假定为格林尼治标准时间 (GMT) 的午夜 (midnight)
// 并会根据你运行代码时的时区进行调整
// 因此，结果可能是
// Thu Jan 26 2017 11:00:00 GMT+1100 (Australian Eastern Daylight Time)
// 或
// Wed Jan 25 2017 16:00:00 GMT-0800 (Pacific Standard Time)
```

`new Date(year, month, date, hours, minutes, seconds, ms)`

使用当前时区中的给定组件创建日期。只有前两个参数是必须的。

- `year` 必须是四位数：`2013` 是合法的，`98` 是不合法的。
- `month` 计数从 `0`（一月）开始，到 `11`（十二月）结束。
- `date` 是当月的具体某一天，如果缺失，则为默认值 `1`。
- 如果 `hours/minutes/seconds/ms` 缺失，则均为默认值 `0`。

例如：

```
new Date(2011, 0, 1, 0, 0, 0); // 1 Jan 2011, 00:00:00  
new Date(2011, 0, 1); // 同样，时分秒等均为默认值 0
```

时间度量最小精确到 1 毫秒（1/1000 秒）：

```
let date = new Date(2011, 0, 1, 2, 3, 4, 567);  
alert(date); // 1.01.2011, 02:03:04.567
```

访问日期组件

从 `Date` 对象中访问年、月等信息有多种方式：

`getFullYear()` ↗

获取年份（4 位数）

`getMonth()` ↗

获取月份，从 **0** 到 **11**。

`getDate()` ↗

获取当月的具体日期，从 **1** 到 **31**，这个方法名称可能看起来有些令人疑惑。

`getHours()` ↗ , `getMinutes()` ↗ , `getSeconds()` ↗ , `getMilliseconds()` ↗

获取相应的时间组件。

⚠ 不是 `getYear()`，而是 `getFullYear()`

很多 JavaScript 引擎都实现了一个非标准化的方法 `getYear()`。不推荐使用这个方法。它有时候可能会返回 2 位的年份信息。永远都不要使用它。要获取年份就使用 `getFullYear()`。

另外，我们还可以获取一周中的第几天：

`getDay()` ↗

获取一周中的第几天，从 **0**（星期日）到 **6**（星期六）。第一天始终是星期日，在某些国家可能不是这样的习惯，但是这不能被改变。

以上的所有方法返回的组件都是基于当地时区的。

当然，也有与当地时区的 UTC 对应项，它们会返回基于 UTC+0 时区的日、月、年等：

`getUTCFullYear()` ↗ , `getUTCMonth()` ↗ , `getUTCDay()` ↗ 。只需要在 "get" 之后插入 "UTC" 即可。

如果你当地时区相对于 UTC 有偏移，那么下面代码会显示不同的小时数：

```
// 当前日期
let date = new Date();

// 当地时区的小时数
alert( date.getHours() );

// 在 UTC+0 时区的小时数（非夏令时的伦敦时间）
alert( date.getUTCHours() );
```

除了上述给定的方法，还有两个没有 **UTC** 变体的特殊方法：

getTime() ↗

返回日期的时间戳——从 **1970-1-1 00:00:00 UTC+0** 开始到现在所经过的毫秒数。

getTimezoneOffset() ↗

返回 **UTC** 与本地时区之间的时差，以分钟为单位：

```
// 如果你在时区 UTC-1，输出 60
// 如果你在时区 UTC+3，输出 -180
alert( new Date().getTimezoneOffset() );
```

设置日期组件

下列方法可以设置日期/时间组件：

- **setFullYear(year, [month], [date])** ↗
- **setMonth(month, [date])** ↗
- **setDate(date)** ↗
- **setHours(hour, [min], [sec], [ms])** ↗
- **setMinutes(min, [sec], [ms])** ↗
- **setSeconds(sec, [ms])** ↗
- **setMilliseconds(ms)** ↗
- **setTime(milliseconds)** ↗ （使用自 **1970-01-01 00:00:00 UTC+0** 以来的毫秒数来设置整个日期）

以上方法除了 **setTime()** 都有 **UTC** 变体，例如： **setUTCHours()**。

我们可以看到，有些方法可以一次性设置多个组件，比如 **setHours**。未提及的组件不会被修改。

举个例子：

```
let today = new Date();

today.setHours(0);
alert(today); // 日期依然是今天，但是小时数被改为了 0

today.setHours(0, 0, 0, 0);
alert(today); // 日期依然是今天，时间为 00:00:00。
```

自动校准 (Autocorrection)

自动校准 是 `Date` 对象的一个非常方便的特性。我们可以设置超范围的数值，它会自动校准。举个例子：

```
let date = new Date(2013, 0, 32); // 32 Jan 2013 ?!
alert(date); // .....是 1st Feb 2013!
```

超出范围的日期组件将会被自动分配。

假设我们要在日期 “28 Feb 2016” 上加 2 天。结果可能是 “2 Mar” 或 “1 Mar”，因为存在闰年。但是我们不需要去考虑这些，只需要直接加 2 天，剩下的 `Date` 对象会帮我们处理：

```
let date = new Date(2016, 1, 28);
date.setDate(date.getDate() + 2);

alert( date ); // 1 Mar 2016
```

这个特性经常被用来获取给定时间段后的日期。例如，我们想获取“现在 70 秒后”的日期：

```
let date = new Date();
date.setSeconds(date.getSeconds() + 70);

alert( date ); // 显示正确的日期信息
```

我们还可以设置 0 甚至可以设置负值。例如：

```
let date = new Date(2016, 0, 2); // 2016 年 1 月 2 日

date.setDate(1); // 设置为当月的第一天
alert( date );

date.setDate(0); // 天数最小可以设置为 1，所以这里设置的是上一月的最后一天
alert( date ); // 31 Dec 2015
```

日期转化为数字，日期差值

当 `Date` 对象被转化为数字时，得到的是对应的时间戳，与使用 `date.getTime()` 的结果相同：

```
let date = new Date();
alert(+date); // 以毫秒为单位的数值，与使用 date.getTime() 的结果相同
```

有一个重要的副作用：日期可以相减，相减的结果是以毫秒为单位时间差。

这个作用可以用于时间测量：

```
let start = new Date(); // 开始测量时间

// do the job
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = new Date(); // 结束测量时间

alert(`The loop took ${end - start} ms`);
```

Date.now()

如果我们仅仅想要测量时间间隔，我们不需要 `Date` 对象。

有一个特殊的方法 `Date.now()`，它会返回当前的时间戳。

它相当于 `new Date().getTime()`，但它不会创建中间的 `Date` 对象。因此它更快，而且不会对垃圾处理造成额外的压力。

这种方法很多时候因为方便，又或是因性能方面的考虑而被采用，例如使用 JavaScript 编写游戏或其他的特殊应用场景。

因此这样做可能会更好：

```
let start = Date.now(); // 从 1 Jan 1970 至今的时间戳

// do the job
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = Date.now(); // 完成

alert(`The loop took ${end - start} ms`); // 相减的是时间戳，而不是日期
```

度量 (Benchmarking)

如果我们想要为一个很耗 CPU 性能的函数提供一个可靠的度量（benchmark），我们应该小心一点。

例如，我们想判断两个计算日期差值的函数：哪个更快？

这种性能测量通常称为“度量（benchmark）”。

```
// 我们有 date1 和 date2，哪个函数会更快地返回两者的时间差？
function diffSubtract(date1, date2) {
  return date2 - date1;
}

// or
function diffGetTime(date1, date2) {
```

```
    return date2.getTime() - date1.getTime();
}
```

这两个函数做的事情完全相同，但是其中一个函数使用显性的 `date.getTime()` 来获取毫秒形式的日期，另一个则依赖于“日期 — 数字”的转换。它们的结果是一样的。

那么，哪个更快呢？

首先想到的方法可能是连续运行它们很多次，并计算时间差。就我们的例子而言，函数非常简单，所以我们必须执行至少 100000 次。

让我们开始测量：

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

alert('Time of diffSubtract: ' + bench(diffSubtract) + 'ms');
alert('Time of diffGetTime: ' + bench(diffGetTime) + 'ms');
```

哇！使用 `getTime()` 这种方式快得多！原因是它没有类型转化，这样对引擎优化来说更加简单。

好，我们得到了结论，但是这并不是一个很好的度量的例子。

想象一下当运行 `bench(diffSubtract)` 的同时，CPU 还在并行处理其他事务，并且这也会占用资源。然而，运行 `bench(diffGetTime)` 的时候，并行处理的事务完成了。

这是对于现代多进程操作系统来说的一个非常真实的场景。

结果就是，第一个函数相比于第二个函数，缺少 CPU 资源。这可能导致错误的结论。

为了得到更加可靠的度量，整个度量测试包应该重新运行多次。

例如，像下面的代码这样：

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}
```

```

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

let time1 = 0;
let time2 = 0;

// 交替运行 bench(upperSlice) 和 bench(upperLoop) 各 10 次
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
  time2 += bench(diffGetTime);
}

alert('Total time for diffSubtract: ' + time1);
alert('Total time for diffGetTime: ' + time2);

```

现代的 JavaScript 引擎的先进优化策略只对执行很多次的“hot code”有效（对于执行很少次数的代码没有必要优化）。因此，在上面的例子中，第一次执行的优化程度不高。我们可能需要增加一个升温步骤：

```

// 在主循环中增加“升温”环节
bench(diffSubtract);
bench(diffGetTime);

// 开始度量
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
  time2 += bench(diffGetTime);
}

```

进行微度量测试时要小心

现代的 JavaScript 引擎执行了很多优化。与“正常使用”相比，它们可能会改变“人为测试”的结果，特别是在我们对很细微的东西进行度量测试时，例如 **operator** 的工作方式或内建函数。因此，如果你想好好了解一下性能，请学习 JavaScript 引擎的工作原理。在那之后，你可能再也不需要微度量了。

关于 V8 引擎的大量文章，可以在 <http://mrale.ph> 找到。

对一个字符串使用 **Date.parse**

Date.parse(str) 方法可以从一个字符串中读取日期。

字符串的格式应该为： YYYY-MM-DDTHH:mm:ss.sssZ，其中：

- YYYY-MM-DD —— 日期：年-月-日。
- 字符 "T" 是一个分隔符。
- HH:mm:ss.sss —— 时间：小时，分钟，秒，毫秒。

- 可选字符 'Z' 为 `+hh:mm` 格式的时区。单个字符 Z 代表 UTC+0 时区。

简短形式也是可以的，比如 `YYYY-MM-DD` 或 `YYYY-MM`，甚至可以是 `YYYY`。

`Date.parse(str)` 调用会解析给定格式的字符串，并返回时间戳（自 1970-01-01 00:00:00 起所经过的毫秒数）。如果给定字符串的格式不正确，则返回 `Nan`。

举个例子：

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417 (时间戳)
```

我们可以通过时间戳来立即创建一个 `new Date` 对象：

```
let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );

alert(date);
```

总结

- 在 JavaScript 中，日期和时间使用 `Date` 对象来表示。我们不能只创建日期，或者只创建时间，`Date` 对象总是同时创建两者。
- 月份从 0 开始计数（对，一月是 0）。
- 一周中的某一天 `getDay()` 同样从 0 开始计算（0 代表星期日）。
- 当设置了超出范围的组件时，`Date` 会进行自我校准。这一点对于日/月/小时的加减很有用。
- 日期可以相减，得到的是以毫秒表示的两者的差值。因为当 `Date` 被转换为数字时，`Date` 对象会被转换为时间戳。
- 使用 `Date.now()` 可以更快地获取当前时间的时间戳。

和其他系统不同，JavaScript 中时间戳以毫秒为单位，而不是秒。

有时我们需要更加精准的时间度量。JavaScript 自身并没有测量微秒的方法（百万分之一秒），但大多数运行环境会提供。例如：浏览器有 `performance.now()` 方法来给出从页面加载开始的以毫秒为单位的微秒数（精确到毫秒的小数点后三位）：

```
alert(`Loading started ${performance.now()}ms ago`);
// 类似于 "Loading started 34731.2600000001ms ago"
// .26 表示的是微秒 (260 微秒)
// 小数点后超过 3 位的数字是精度错误，只有前三位数字是正确的
```

Node.js 有 `microtime` 模块以及其他方法。从技术上讲，几乎所有的设备和环境都允许获取更高精度的数值，只是不是通过 `Date` 对象。

JSON 方法, `toJSON`

假设我们有一个复杂的对象，我们希望将其转换为字符串，以通过网络发送，或者只是为了在日志中输出它。

当然，这样的字符串应该包含所有重要的属性。

我们可以像这样实现转换：

```
let user = {
  name: "John",
  age: 30,
  toString() {
    return `${name}: ${this.name}, age: ${this.age}`;
  }
};

alert(user); // {name: "John", age: 30}
```

.....但在开发过程中，会新增一些属性，旧的属性会被重命名和删除。每次更新这种 `toString` 都会非常痛苦。我们可以尝试遍历其中的属性，但是如果对象很复杂，并且在属性中嵌套了对象呢？我们也需要对它们进行转换。

幸运的是，不需要编写代码来处理所有这些问题。这项任务已经解决了。

JSON.stringify

`JSON ↗` (JavaScript Object Notation) 是表示值和对象的通用格式。在 `RFC 4627 ↗` 标准中有对其的描述。最初它是为 `JavaScript` 而创建的，但许多其他编程语言也有用于处理它的库。因此，当客户端使用 `JavaScript` 而服务器端是使用 `Ruby/PHP/Java` 等语言编写的时，使用 `JSON` 可以很容易地进行数据交换。

`JavaScript` 提供了如下方法：

- `JSON.stringify` 将对象转换为 `JSON`。
- `JSON.parse` 将 `JSON` 转换回对象。

例如，在这里我们 `JSON.stringify` 一个 `student` 对象：

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  wife: null
};

let json = JSON.stringify(student);

alert(typeof json); // we've got a string!

alert(json);
/* JSON 编码的对象:
{
  "name": "John",
  "age": 30,
```

```
"isAdmin": false,  
"courses": ["html", "css", "js"],  
"wife": null  
}  
*/
```

方法 `JSON.stringify(student)` 接收对象并将其转换为字符串。

得到的 `json` 字符串是一个被称为 **JSON 编码 (JSON-encoded)** 或 **序列化 (serialized)** 或 **字符串化 (stringified)** 或 **编组化 (marshalled)** 的对象。我们现在已经准备好通过有线发送它或将其放入普通数据存储。

请注意，**JSON** 编码的对象与对象字面量有几个重要的区别：

- 字符串使用双引号。**JSON** 中没有单引号或反引号。所以 `'John'` 被转换为 `"John"`。
- 对象属性名称也是双引号的。这是强制性的。所以 `age:30` 被转换成 `"age":30`。

`JSON.stringify` 也可以应用于原始 (**primitive**) 数据类型。

JSON 支持以下数据类型：

- Objects `{ ... }`
- Arrays `[...]`
- Primitives:
 - strings,
 - numbers,
 - boolean values `true/false`,
 - `null`。

例如：

```
// 数字在 JSON 还是数字  
alert( JSON.stringify(1) ) // 1  
  
// 字符串在 JSON 中还是字符串，只是被双引号扩起来  
alert( JSON.stringify('test') ) // "test"  
  
alert( JSON.stringify(true) ); // true  
  
alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

JSON 是语言无关的纯数据规范，因此一些特定于 **JavaScript** 的对象属性会被 `JSON.stringify` 跳过。

即：

- 函数属性（方法）。
- **Symbol** 类型的属性。
- 存储 `undefined` 的属性。

```
let user = {
  sayHi() { // 被忽略
    alert("Hello");
  },
  [Symbol("id")]: 123, // 被忽略
  something: undefined // 被忽略
};

alert( JSON.stringify(user) ); // {} (空对象)
```

通常这很好。如果这不是我们想要的方式，那么我们很快就会看到如何自定义转换方式。

最棒的是支持嵌套对象转换，并且可以自动对其进行转换。

例如：

```
let meetup = {
  title: "Conference",
  room: {
    number: 23,
    participants: ["john", "ann"]
  }
};

alert( JSON.stringify(meetup) );
/* 整个解构都被字符串化了
{
  "title": "Conference",
  "room": {"number": 23, "participants": ["john", "ann"]},
}
*/
```

重要的限制：不得有循环引用。

例如：

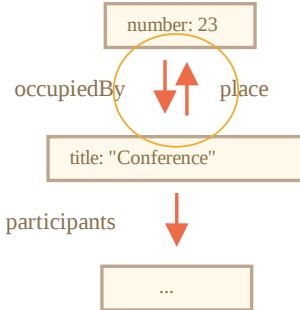
```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: ["john", "ann"]
};

meetup.place = room;           // meetup 引用了 room
room.occupiedBy = meetup; // room 引用了 meetup

JSON.stringify(meetup); // Error: Converting circular structure to JSON
```

在这里，转换失败了，因为循环引用：`room.occupiedBy` 引用了 `meetup`，`meetup.place` 引用了 `room`：



排除和转换: `replacer`

`JSON.stringify` 的完整语法是:

```
let json = JSON.stringify(value[, replacer, space])
```

value

要编码的值。

replacer

要编码的属性数组或映射函数 `function(key, value)`。

space

用于格式化的空格数量

大部分情况, `JSON.stringify` 仅与第一个参数一起使用。但是, 如果我们需要微调替换过程, 比如过滤掉循环引用, 我们可以使用 `JSON.stringify` 的第二个参数。

如果我们传递一个属性数组给它, 那么只有这些属性会被编码。

例如:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup 引用了 room
};

room.occupiedBy = meetup; // room 引用了 meetup

alert( JSON.stringify(meetup, ['title', 'participants']) );
// {"title":"Conference","participants":[]}
```

这里我们可能过于严格了。属性列表应用于了整个对象结构。所以 `participants` 是空的, 因为 `name` 不在列表中。

让我们包含除了会导致循环引用的 `room.occupiedBy` 之外的所有属性:

```

let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup 引用了 room
};

room.occupiedBy = meetup; // room 引用了 meetup

alert( JSON.stringify(meetup, ['title', 'participants', 'place', 'name', 'number']) );
/*
{
  "title": "Conference",
  "participants": [{"name": "John"}, {"name": "Alice"}],
  "place": {"number": 23}
}
*/

```

现在，除 `occupiedBy` 以外的所有内容都被序列化了。但是属性的列表太长了。

幸运的是，我们可以使用一个函数代替数组作为 `replacer`。

该函数会为每个 `(key, value)` 对调用并返回“已替换”的值，该值将替换原有的值。如果值被跳过了，则为 `undefined`。

在我们的例子中，我们可以为 `occupiedBy` 以外的所有内容按原样返回 `value`。为了 `occupiedBy`，下面的代码返回 `undefined`：

```

let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup 引用了 room
};

room.occupiedBy = meetup; // room 引用了 meetup

alert( JSON.stringify(meetup, function replacer(key, value) {
  alert(` ${key}: ${value}`);
  return (key == 'occupiedBy') ? undefined : value;
}));

/* key:value pairs that come to replacer:
: [object Object]
title: Conference
participants: [object Object], [object Object]
0: [object Object]
name: John
1: [object Object]
name: Alice
place: [object Object]
*/

```

```
number:      23
*/
```

请注意 `replacer` 函数会获取每个键/值对，包括嵌套对象和数组项。它被递归地应用。
`replacer` 中的 `this` 的值是包含当前属性的对象。

第一个调用很特别。它是使用特殊的“包装对象”制作的: `{"": meetup}`。换句话说，第一个 `(key, value)` 对的键是空的，并且该值是整个目标对象。这就是上面的示例中第一行是 `"": [object Object]"` 的原因。

这个理念是为了给 `replacer` 提供尽可能多的功能：如果有必要，它有机会分析并替换/跳过整个对象。

格式化: `space`

`JSON.stringify(value, replacer, spaces)` 的第三个参数是用于优化格式的空格数量。

以前，所有字符串化的对象都没有缩进和额外的空格。如果我们想通过网络发送一个对象，那就没什么问题。`space` 参数专门用于调整出更美观的输出。

这里的 `space = 2` 告诉 JavaScript 在多行中显示嵌套的对象，对象内部缩紧 2 个空格：

```
let user = {
  name: "John",
  age: 25,
  roles: {
    isAdmin: false,
    isEditor: true
  }
};

alert(JSON.stringify(user, null, 2));
/* 两个空格的缩进:
{
  "name": "John",
  "age": 25,
  "roles": {
    "isAdmin": false,
    "isEditor": true
  }
}
*/
/* 对于 JSON.stringify(user, null, 4) 的结果会有更多缩进:
{
  "name": "John",
  "age": 25,
  "roles": {
    "isAdmin": false,
    "isEditor": true
  }
}
*/
```

`spaces` 参数仅用于日志记录和美化输出。

自定义“`toJSON`”

像 `toString` 进行字符串转换，对象也可以提供 `toJSON` 方法来进行 JSON 转换。如果可用，`JSON.stringify` 会自动调用它。

例如：

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  date: new Date(Date.UTC(2017, 0, 1)),
  room
};

alert( JSON.stringify(meetup) );
/*
{
  "title": "Conference",
  "date": "2017-01-01T00:00:00.000Z", // (1)
  "room": {"number": 23}           // (2)
}
*/
```

在这儿我们可以看到 `date` (1) 变成了一个字符串。这是因为所有日期都有一个内置的 `toJSON` 方法来返回这种类型的字符串。

现在让我们为对象 `room` 添加一个自定义的 `toJSON`：

```
let room = {
  number: 23,
  toJSON() {
    return this.number;
  }
};

let meetup = {
  title: "Conference",
  room
};

alert( JSON.stringify(room) ); // 23

alert( JSON.stringify(meetup) );
/*
{
  "title": "Conference",
  "room": 23
}
*/
```

正如我们所看到的，`toJSON` 既可以用于直接调用 `JSON.stringify(room)` 也可以用于当 `room` 嵌套在另一个编码对象中时。

JSON.parse

要解码 JSON 字符串，我们需要另一个方法 `JSON.parse` 。

语法:

```
let value = JSON.parse(str, [reviver]);
```

str

要解析的 JSON 字符串。

reviver

可选的函数 `function(key,value)`，该函数将为每个 `(key, value)` 对调用，并可以对值进行转换。

例如:

```
// 字符串化数组
let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1
```

对于嵌套对象:

```
let userData = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

let user = JSON.parse(userData);

alert( user.friends[1] ); // 1
```

JSON 可能会非常复杂，对象和数组可以包含其他对象和数组。但是他们必须遵循相同的 JSON 格式。

以下是手写 JSON 时的典型错误（有时我们必须出于调试目的编写它）：

```
let json = `{
  name: "John",           // 错误: 属性名没有双引号
  "surname": 'Smith',     // 错误: 值使用的是单引号（必须使用双引号）
  'isAdmin': false        // 错误: 键使用的是单引号（必须使用双引号）
  "birthday": new Date(2000, 2, 3), // 错误: 不允许使用 "new"，只能是裸值
  "friends": [0,1,2,3]      // 这个没问题
};`;
```

此外，JSON 不支持注释。向 JSON 添加注释无效。

还有另一种名为 [JSON5 ↗](#) 的格式，它允许未加引号的键，也允许注释等。但这是一个独立的库，不在语言的规范中。

常规的 JSON 格式严格，并不是因为它的开发者很懒，而是为了实现简单，可靠且快速地实现解析算法。

使用 reviver

想象一下，我们从服务器上获得了一个字符串化的 `meetup` 对象。

它看起来像这样：

```
// title: (meetup title), date: (meetup date)
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

.....现在我们需要对它进行 **反序列（deserialize）**，把它转换回 JavaScript 对象。

让我们通过调用 `JSON.parse` 来完成：

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!
```

啊！报错了！

`meetup.date` 的值是一个字符串，而不是 `Date` 对象。`JSON.parse` 怎么知道应该将字符串转换为 `Date` 呢？

让我们将 `reviver` 函数传递给 `JSON.parse` 作为第二个参数，该函数按照“原样”返回所有值，但是 `date` 会变成 `Date`：

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( meetup.date.getDate() ); // 现在正常运行了!
```

顺便说一下，这也适用于嵌套对象：

```
let schedule = `{
  "meetups": [
    {"title":"Conference", "date":"2017-11-30T12:00:00.000Z"},
    {"title":"Birthday", "date":"2017-04-18T12:00:00.000Z"}
]
```

```
};

schedule = JSON.parse(schedule, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( schedule.meetups[1].date.getDate() ); // 正常运行了!
```

总结

- JSON 是一种数据格式，具有自己的独立标准和大多数编程语言的库。
- JSON 支持 object, array, string, number, boolean 和 null。
- JavaScript 提供序列化（serialize）成 JSON 的方法 `JSON.stringify` 和解析 JSON 的方法 `JSON.parse`。
- 这两种方法都支持用于智能读/写的转换函数。
- 如果一个对象具有 `toJSON`，那么它会被 `JSON.stringify` 调用。

函数进阶内容 递归和堆栈

让我们回到函数，进行更深入的研究。

我们的第一个主题是 **递归（recursion）**。

如果你不是刚接触编程，那么你可能已经很熟悉它了，那么你可以跳过这一章。

递归是一种编程模式，在一个任务可以自然地拆分成多个相同类型但更简单的任务的情况下非常有用。或者，在一个任务可以简化为一个简单的行为加上该任务的一个更简单的变体的时候可以使。或者，就像我们很快会看到的那样，处理某些数据结构。

当一个函数解决一个任务时，在解决的过程中它可以调用很多其它函数。在部分情况下，函数会调用 **自身**。这就是所谓的 **递归**。

两种思考方式

简单起见，让我们写一个函数 `pow(x, n)`，它可以计算 `x` 的 `n` 次方。换句话说就是，`x` 乘以自身 `n` 次。

```
pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16
```

有两种实现方式。

1. 迭代思路：使用 `for` 循环：

```
function pow(x, n) {
  let result = 1;

  // 再循环中，用 x 乘以 result n 次
```

```

for (let i = 0; i < n; i++) {
    result *= x;
}

return result;
}

alert( pow(2, 3) ); // 8

```

2. 递归思路：简化任务，调用自身：

```

function pow(x, n) {
    if (n == 1) {
        return x;
    } else {
        return x * pow(x, n - 1);
    }
}

alert( pow(2, 3) ); // 8

```

请注意，递归变体在本质上是不同的。

当 `pow(x, n)` 被调用时，执行分为两个分支：

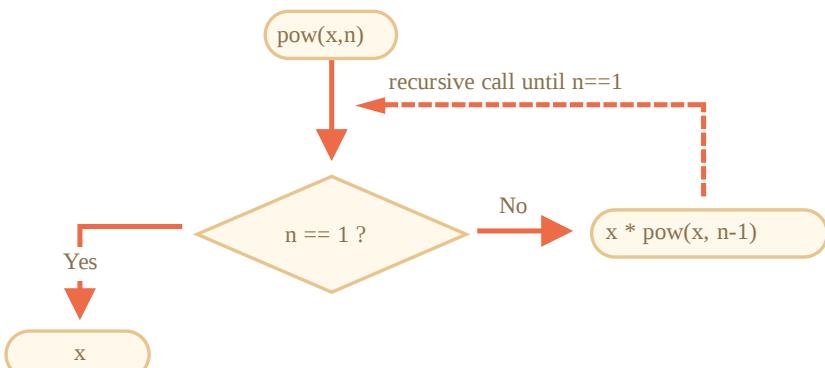
```

if n==1 = x
/
pow(x, n) =
 \
else      = x * pow(x, n - 1)

```

1. 如果 `n == 1`，所有事情都会很简单，这叫做 **基础** 的递归，因为它会立即产生明显的结果：`pow(x, 1)` 等于 `x`。
2. 否则，我们可以用 `x * pow(x, n - 1)` 表示 `pow(x, n)`。在数学里，可能会写为 $x^n = x \cdot x^{n-1}$ 。这叫做 **一个递归步骤**：我们将任务转化为更简单的行为（`x` 的乘法）和更简单的同类任务的调用（带有更小的 `n` 的 `pow` 运算）。接下来的步骤将其进一步简化，直到 `n` 达到 `1`。

我们也可以将 `pow` 递归地调用自身直到 `n == 1`。



比如，为了计算 `pow(2, 4)`，递归变体经过了下面几个步骤：

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

因此，递归将函数调用简化为一个更简单的函数调用，然后再将其简化为一个更简单的函数，以此类推，直到结果变得显而易见。

❶ 递归通常更短

递归解通常比迭代解更短。

在这儿，我们可以使用条件运算符 `?` 而不是 `if` 语句，从而使 `pow(x, n)` 更简洁并且可读性依然很高：

```
function pow(x, n) {  
    return (n == 1) ? x : (x * pow(x, n - 1));  
}
```

最大的嵌套调用次数（包括首次）被称为 **递归深度**。在我们的例子中，它正好等于 `n`。

最大递归深度受限于 JavaScript 引擎。对我们来说，引擎在最大迭代深度为 10000 及以下时是可靠的，有些引擎可能允许更大的最大深度，但是对于大多数引擎来说，100000 可能就超出限制了。有一些自动优化能够帮助减轻这种情况（尾部调用优化），但目前它们还没有被完全支持，只能用于简单场景。

这就限制了递归的应用，但是递归仍然被广泛使用。有很多任务中，递归思维方式会使代码更简单，更容易维护。

执行上下文和堆栈

现在我们来研究一下递归调用是如何工作的。为此，我们会先看看函数底层的工作原理。

有关正在运行的函数的执行过程的相关信息被存储在其 **执行上下文** 中。

执行上下文 ↗ 是一个内部数据结构，它包含有关函数执行时的详细细节：当前控制流所在的位置，当前的变量，`this` 的值（此处我们不使用它），以及其它的一些内部细节。

一个函数调用仅具有一个与其相关联的执行上下文。

当一个函数进行嵌套调用时，将发生以下的事儿：

- 当前函数被暂停；
- 与它关联的执行上下文被一个叫做 **执行上下文堆栈** 的特殊数据结构保存；
- 执行嵌套调用；
- 嵌套调用结束后，从堆栈中恢复之前的执行上下文，并从停止的位置恢复外部函数。

让我们看看 `pow(2, 3)` 调用期间都发生了什么。

pow(2, 3)

在调用 `pow(2, 3)` 的开始，执行上下文（context）会存储变量：`x = 2, n = 3`，执行流程在函数的第 1 行。

我们将其描绘如下：

- **Context: { x: 2, n: 3, at line 1 }** call: `pow(2, 3)`

这是函数开始执行的时候。条件 `n == 1` 结果为 `false`，所以执行流程进入 `if` 的第二分支。

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3));
```

变量相同，但是行改变了，因此现在的上下文是：

- **Context: { x: 2, n: 3, at line 5 }** call: `pow(2, 3)`

为了计算 `x * pow(x, n - 1)`，我们需要使用带有新参数的新的 `pow` 子调用 `pow(2, 2)`。

pow (2, 2)

为了执行嵌套调用，JavaScript 会在 **执行上下文堆栈** 中记住当前的执行上下文。

这里我们调用相同的函数 `pow`，但这绝对没问题。所有函数的处理都是一样的：

1. 当前上下文被“记录”在堆栈的顶部。
2. 为子调用创建新的上下文。
3. 当子调用结束后——前一个上下文被从堆栈中弹出，并继续执行。

下面是进入子调用 `pow(2, 2)` 时的上下文堆栈：

- **Context: { x: 2, n: 2, at line 1 }** call: `pow(2, 2)`
- **Context: { x: 2, n: 3, at line 5 }** call: `pow(2, 3)`

新的当前执行上下文位于顶部（粗体显示），之前记住的上下文位于下方。

当我们完成子调用后——很容易恢复上一个上下文，因为它既保留了变量，也保留了当时所在代码的确切位置。

i 请注意：

在上面的图中，我们使用“行”一词作为示例，每一行只有一个子调用，但通常一行代码可能会包含多个子调用，像 `pow(...) + pow(...) + somethingElse(...)`。

因此，更准确地说，执行是“在子调用之后立即恢复”的。

pow(2, 1)

重复该过程：在第 5 行生成新的子调用，现在的参数是 `x=2, n=1`。

新的执行上下文被创建，前一个被压入堆栈顶部：

- **Context: { x: 2, n: 1, at line 1 }** call: pow(2, 1)
- **Context: { x: 2, n: 2, at line 5 }** call: pow(2, 2)
- **Context: { x: 2, n: 3, at line 5 }** call: pow(2, 3)

此时，有 2 个旧的上下文和 1 个当前正在运行的 `pow(2, 1)` 的上下文。

出口

在执行 `pow(2, 1)` 时，与之前的不同，条件 `n == 1` 为 true，因此 if 的第一个分支生效：

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
```

此时不再有更多的嵌套调用，所以函数结束，返回 2。

函数完成后，就不再需要其执行上下文了，因此它被从内存中移除。前一个上下文恢复到堆栈的顶部：

- **Context: { x: 2, n: 2, at line 5 }** call: pow(2, 2)
- **Context: { x: 2, n: 3, at line 5 }** call: pow(2, 3)

恢复执行 `pow(2, 2)`。它拥有子调用 `pow(2, 1)` 的结果，因此也可以完成 `x * pow(x, n - 1)` 的执行，并返回 4。

然后，前一个上下文被恢复：

- **Context: { x: 2, n: 3, at line 5 }** call: pow(2, 3)

当它结束后，我们得到了结果 `pow(2, 3) = 8`。

本示例中的递归深度为：3。

从上面的插图我们可以看出，递归深度等于堆栈中上下文的最大数量。

请注意内存要求。上下文占用内存，在我们的示例中，求 `n` 次方需要存储 `n` 个上下文，以供更小的 `n` 值进行计算使用。

而循环算法更节省内存：

```
function pow(x, n) {
  let result = 1;
```

```
for (let i = 0; i < n; i++) {
    result *= x;
}

return result;
}
```

迭代 `pow` 的过程中仅使用了一个上下文用于修改 `i` 和 `result`。它的内存要求小，并且是固定了，不依赖于 `n`。

任何递归都可以用循环来重写。通常循环变体更有效。

.....但有时重写很难，尤其是函数根据条件使用不同的子调用，然后合并它们的结果，或者分支比较复杂时。而且有些优化可能没有必要，完全不值得。

递归可以使代码更短，更易于理解和维护。并不是每个地方都需要优化，大多数时候我们需要一个好代码，这就是为什么要使用它。

递归遍历

递归的另一个重要应用就是递归遍历。

假设我们有一家公司。人员结构可以表示为一个对象：

```
let company = {
  sales: [
    {
      name: 'John',
      salary: 1000
    },
    {
      name: 'Alice',
      salary: 1600
    }
  ],

  development: [
    {
      sites: [
        {
          name: 'Peter',
          salary: 2000
        },
        {
          name: 'Alex',
          salary: 1800
        }
      ]
    }
};
```

换句话说，一家公司有很多部门。

- 一个部门可能有一 **数组** 的员工，比如，`sales` 部门有 2 名员工：John 和 Alice。
- 或者，一个部门可能会划分为几个子部门，比如 `development` 有两个分支：`sites` 和 `internals`，它们都有自己的员工。

- 当一个子部门增长时，它也有可能被拆分成几个子部门（或团队）。

例如，`sites` 部门在未来可能会分为 `siteA` 和 `siteB`。并且，它们可能会被再继续拆分。
没有图示，脑补一下吧。

现在，如果我们需要一个函数来获取所有薪资的总数。我们该怎么做？

迭代方式不容易，因为结构比较复杂。首先想到的可能是在 `company` 上使用 `for` 循环，并在第一层部分上嵌套子循环。但是，之后我们需要更多的子循环来遍历像 `sites` 这样的二级部门的员工……然后，将来可能会出现在三级部门上的另一个子循环？如果我们在代码中写 3-4 级嵌套的子循环来遍历单个对象，那代码得多丑啊。

我们试试递归吧。

我们可以看到，当我们的函数对一个部门求和时，有两种可能的情况：

- 要么是由一 **数组** 的人组成的“简单”的部门——这样我们就可以通过一个简单的循环来计算薪资的总和。
- 或者它是一个有 **N** 个子部门的 **对象**——那么我们可以通过 **N** 层递归调用来求每一个子部门的薪资，然后将它们合并起来。

第一种情况是由一数组的人组成的部门，这种情况很简单，是最基础的递归。

第二种情况是我们得到的是对象。那么可将这个复杂的任务拆分成适用于更小部门的子任务。它们可能会被继续拆分，但很快或者不久就会拆分到第一种情况那样。

这个算法从代码来看可能会更简单：

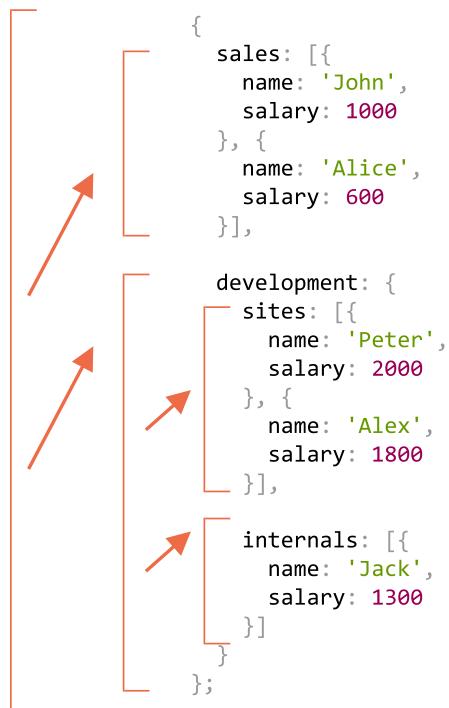
```
let company = { // 是同一个对象，简介起见被压缩了
  sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 1600}],
  development: {
    sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800}],
    internals: [{name: 'Jack', salary: 1300}]
  }
};

// 用来完成任务的函数
function sumSalaries(department) {
  if (Array.isArray(department)) { // 情况 (1)
    return department.reduce((prev, current) => prev + current.salary, 0); // 求数组的和
  } else { // 情况 (2)
    let sum = 0;
    for (let subdep of Object.values(department)) {
      sum += sumSalaries(subdep); // 递归调用所有子部门，对结果求和
    }
    return sum;
  }
}

alert(sumSalaries(company)); // 7700
```

代码很短也容易理解（希望是这样？）。这就是递归的能力。它适用于任何层次的子部门嵌套。

下面是调用图：



我们可以很容易地看到其原理：对于对象 `{ ... }` 会生成子调用，而数组 `[...]` 是递归树的“叶子”，它们会立即给出结果。

请注意，该代码使用了我们之前讲过的智能特性（smart features）：

- 在 `数组方法` 中我们介绍过的数组求和方法 `arr.reduce`。
- 使用循环 `for(val of Object.values(obj))` 遍历对象的（属性）值：
`Object.values` 返回它们组成的数组。

递归结构

递归（递归定义的）数据结构是一种部分复制自身的结构。

我们刚刚在上面的公司结构的示例中看过了它。

一个公司的 **部门** 是：

- 一数组的人。
- 或一个 **部门** 对象。

对于 Web 开发者而言，有更熟知的例子：HTML 和 XML 文档。

在 HTML 文档中，一个 **HTML 标签** 可能包括以下内容：

- 文本片段。
- HTML 注释。
- 其它 **HTML 标签**（它有可能又包括文本片段、注释或其它标签等）。

这又是一个递归定义。

为了更好地理解递归，我们再讲一个递归结构的例子“链表”，在某些情况下，它可能是优于数组的选择。

链表

想象一下，我们要存储一个有序的对象列表。

正常的选择会是一个数组:

```
let arr = [obj1, obj2, obj3];
```

.....但是用数组有个问题。“删除元素”和“插入元素”的操作代价非常大。例如，`arr.unshift(obj)` 操作必须对所有元素重新编号以便为新的元素 `obj` 腾出空间，而且如果数组很大，会很耗时。`arr.shift()` 同理。

唯一对数组结构做修改而不需要大量重排的操作就是对数组末端的操作：`arr.push/pop`。因此，对于大队列来说，当我们必须对数组首端的元素进行操作时，数组会很慢。（译注：此处的首端操作其实指的是在尾端以外的数组内的元素进行插入/删除操作。）

如果我们确实需要快速插入/删除，则可以选择另一种叫做 [链表 ↗](#) 的数据结构。

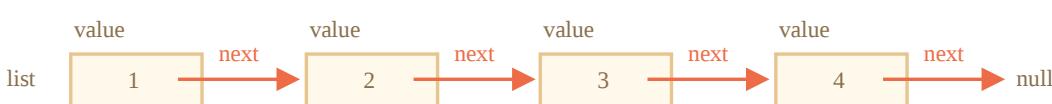
链表元素 是一个使用以下元素通过递归定义的对象：

- `value`。
- `next` 属性引用下一个 **链表元素** 或者代表末尾的 `null`。

例如：

```
let list = {  
  value: 1,  
  next: {  
    value: 2,  
    next: {  
      value: 3,  
      next: {  
        value: 4,  
        next: null  
      }  
    }  
  }  
};
```

链表的图形表示：



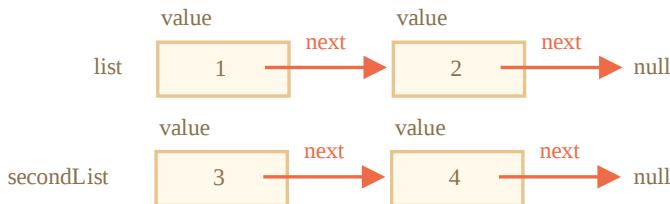
一段用来创建链表的代码：

```
let list = { value: 1 };  
list.next = { value: 2 };  
list.next.next = { value: 3 };  
list.next.next.next = { value: 4 };  
list.next.next.next.next = null;
```

在这儿我们可以清楚地看到，这里有很多个对象，每一个都有 `value` 和指向邻居的 `next`。变量 `list` 是链条中的第一个对象，因此顺着 `next` 指针，我们可以抵达任何元素。

该链表很容易被拆分为多个部分，然后再重新组装回去：

```
let secondList = list.next.next;
list.next.next = null;
```



合并：

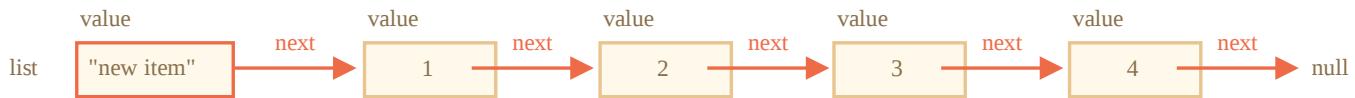
```
list.next.next = secondList;
```

当然，我们可以在任何位置插入或移除元素。

比如，要添加一个新值，我们需要更新链表的头：

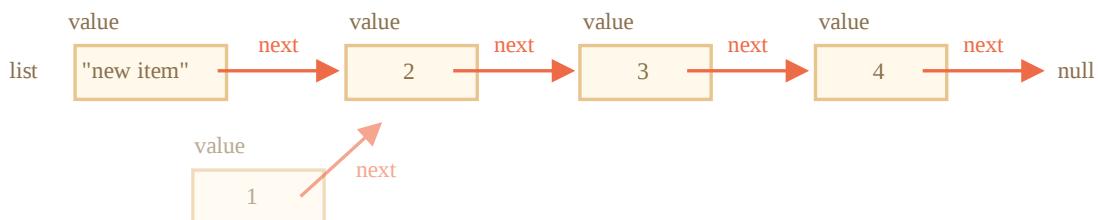
```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };

// 将新值添加到链表头部
list = { value: "new item", next: list };
```



要从中间删除一个值，可以修改前一个元素的 `next`：

```
list.next = list.next.next;
```



我们让 `list.next` 从 1 跳到值 2。现在值 1 就被从链表中移除了。如果它没有被存储在其它任何地方，那么它会被自动从内存中删除。

与数组不同，链表没有大规模重排，我们可以很容易地重新排列元素。

当然，链表也不总是优于数组的。不然大家就都去使用链表了。

链表主要的缺点就是我们无法很容易地通过元素的编号获取元素。但在数组中却很容易：`arr[n]` 是一个直接引用。而在链表中，我们需要从起点元素开始，顺着 `next` 找 `N` 次才能获取到第 `N` 个元素。

.....但是我们也并不是总需要这样的操作。比如，当我们需要一个队列甚至一个 [双向队列](#) —— 有序结构必须可以快速地从两端添加/移除元素，但是不需要访问的元素。

链表可以得到增强：

- 我们可以在 `next` 之外，再添加 `prev` 属性来引用前一个元素，以便轻松地往回移动。
- 我们还可以添加一个名为 `tail` 的变量，该变量引用链表的最后一个元素（并在从末尾添加/删除元素时对该引用进行更新）。
-数据结构可能会根据我们的需求而变化。

总结

术语：

- **递归** 是编程的一个术语，表示从自身调用函数（译注：也就是自调用）。递归函数可用于以更优雅的方式解决问题。

当一个函数调用自身时，我们称其为 **递归步骤**。递归的 **基础** 是函数参数使任务简单到该函数不再需要进行进一步调用。

- [递归定义](#) 的数据结构是指可以使用自身来定义的数据结构。

例如，链表可以被定义为由对象引用一个列表（或 `null`）而组成的数据结构。

```
list = { value, next -> list }
```

像 `HTML` 元素树或者本章中的 `department` 树等，本质上也是递归：它们有分支，而且分支又可以有其他分支。

就像我们在示例 `sumSalary` 中看到的那样，可以使用递归函数来遍历它们。

任何递归函数都可以被重写为迭代（译注：也就是循环）形式。有时这是在优化代码时需要做的。但对于大多数任务来说，递归方法足够快，并且容易编写和维护。

Rest 参数与 Spread 语法

在 `JavaScript` 中，很多内建函数都支持传入任意数量的参数。

例如：

- `Math.max(arg1, arg2, ..., argN)` —— 返回入参中的最大值。
- `Object.assign(dest, src1, ..., srcN)` —— 依次将属性从 `src1..N` 复制到 `dest`。
-等。

在本章中，我们将学习如何编程实现支持函数可传入任意数量的参数。以及，如何将数组作为参数传递给这类函数。

Rest 参数 ...

在 JavaScript 中，无论函数是如何定义的，你都可以使用任意数量的参数调用函数。

例如：

```
function sum(a, b) {
  return a + b;
}

alert( sum(1, 2, 3, 4, 5) );
```

虽然这里不会因为传入“过多”的参数而报错。但是当然，在结果中只有前两个参数被计算进去了。

Rest 参数可以通过使用三个点 `...` 并在后面跟着包含剩余参数的数组名称，来将它们包含在函数定义中。这些点的字面意思是“将剩余参数收集到一个数组中”。

例如，我们需要把所有的参数都放到数组 `args` 中：

```
function sumAll(...args) { // 数字名为 args
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

我们也可以选择获取第一个参数作为变量，并将剩余的参数收集起来。

下面的例子把前两个参数定义为变量，并把剩余的参数收集到 `titles` 数组中：

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julius Caesar

  // 剩余的参数被放入 titles 数组中
  // i.e. titles = ["Consul", "Imperator"]
  alert( titles[0] ); // Consul
  alert( titles[1] ); // Imperator
  alert( titles.length ); // 2
}

showName("Julius", "Caesar", "Consul", "Imperator");
```

⚠ Rest 参数必须放到参数列表的末尾

Rest 参数会收集剩余的所有参数，因此下面这种用法没有意义，并且会导致错误：

```
function f(arg1, ...rest, arg2) { // arg2 在 ...rest 后面? !
  // error
}
```

`...rest` 必须处在最后。

“arguments” 变量

有一个名为 `arguments` 的特殊的类数组对象，该对象按参数索引包含所有参数。

例如：

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );

  // 它是可遍历的
  // for(let arg of arguments) alert(arg);
}

// 依次显示: 2, Julius, Caesar
showName("Julius", "Caesar");

// 依次显示: 1, Ilya, undefined (没有第二个参数)
showName("Ilya");
```

在过去，JavaScript 中没有 `rest` 参数，而使用 `arguments` 是获取函数所有参数的唯一方法。现在它仍然有效，我们可以在一些老代码里找到它。

但缺点是，尽管 `arguments` 是一个类数组且可遍历的变量，但它终究不是数组。它不支持数组方法，因此我们不能调用 `arguments.map(...)` 等方法。

此外，它始终包含所有参数，我们不能像使用 `rest` 参数那样只截取入参的一部分。

因此，当我们需要这些功能时，最好使用 `rest` 参数。

① 箭头函数是没有 "arguments"

如果我们在箭头函数中访问 `arguments`，访问到的 `arguments` 并不属于箭头函数，而是属于箭头函数外部的“普通”函数。

举个例子：

```
function f() {
  let showArg = () => alert(arguments[0]);
  showArg();
}

f(1); // 1
```

我们已经知道，箭头函数没有自身的 `this`。现在我们知道了它们也没有特殊的 `arguments` 对象。

Spread 语法

我们刚刚看到了如何从参数列表中获取数组。

不过有时候我们也需要做与之相反的事儿。

例如，内建函数 `Math.max ↗` 会返回参数中最大的值：

```
alert( Math.max(3, 5, 1) ); // 5
```

假如我们有一个数组 `[3, 5, 1]`，我们该如何用它调用 `Math.max` 呢？

直接把数组“原样”传入是不会奏效的，因为 `Math.max` 希望你传入一个列表形式的数值型参数，而不是一个数组：

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

毫无疑问，我们不可能手动地去一一设置参数 `Math.max(arg[0], arg[1], arg[2])`，因为我们不确定这儿有多少个。在脚本执行时，可能参数数组中有很多个元素，也可能一个都没有。并且这样设置的代码也很丑。

Spread 语法 来帮助你了！它看起来和 `rest` 参数很像，也使用 `...`，但是二者的用途完全相反。

当在函数调用中使用 `...arr` 时，它会把可迭代对象 `arr` “展开”到参数列表中。

以 `Math.max` 为例：

```
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread 语法把数组转换为参数列表)
```

我们还可以通过这种方式传递多个可迭代对象:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // 8
```

我们甚至还可以将 **spread** 语法与常规值结合使用:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

并且，我们还可以使用 **spread** 语法来合并数组:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, 然后是 arr, 然后是 2, 然后是 arr2)
```

在上面的示例中，我们使用数组展示了 **spread** 语法，其实任何可迭代对象都可以。

例如，在这儿我们使用 **spread** 语法将字符串转换为字符数组:

```
let str = "Hello";

alert( [...str] ); // H,e,l,l,o
```

Spread 语法内部使用了迭代器来收集元素，与 **for..of** 的方式相同。

因此，对于一个字符串，**for..of** 会逐个返回该字符串中的字符，**...str** 也同理会得到 "**H**", "**e**", "**l**", "**l**", "**o**" 这样的结果。随后，字符列表会被传递给数组初始化器 **[...str]**。

对于这个特定任务，我们还可以使用 **Array.from** 来实现，因为该方法会将一个可迭代对象（如字符串）转换为数组:

```
let str = "Hello";

// Array.from 将可迭代对象转换为数组
alert( Array.from(str) ); // H,e,l,l,o
```

运行结果与 **[...str]** 相同。

不过 **Array.from(obj)** 和 **[...obj]** 存在一个细微的差别:

- **Array.from** 适用于类数组对象也适用于可迭代对象。

- Spread 语法只适用于可迭代对象。

因此，对于将一些“东西”转换为数组的任务，`Array.from` 往往更通用。

获取一个 `object/array` 的副本

还记得我们 [之前讲过的](#) `Object.assign()` 吗？

使用 `spread` 操作符也可以做同样的事情。

```
let arr = [1, 2, 3];
let arrCopy = [...arr]; // 将数组 spread 到参数列表中
                        // 然后将结果放到一个新数组

// 两个数组中的内容相同吗?
alert(JSON.stringify(arr) === JSON.stringify(arrCopy)); // true

// 两个数组相等吗?
alert(arr === arrCopy); // false (它们的引用是不同的)

// 修改我们初始的数组不会修改副本:
arr.push(4);
alert(arr); // 1, 2, 3, 4
alert(arrCopy); // 1, 2, 3
```

并且，也可以通过相同的方式来复制一个对象：

```
let obj = { a: 1, b: 2, c: 3 };
let objCopy = { ...obj }; // 将对象 spread 到参数列表中
                        // 然后将结果返回到一个新对象

// 两个对象中的内容相同吗?
alert(JSON.stringify(obj) === JSON.stringify(objCopy)); // true

// 两个对象相等吗?
alert(obj === objCopy); // false (not same reference)

// 修改我们初始的对象不会修改副本:
obj.d = 4;
alert(JSON.stringify(obj)); // {"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}
```

这种方式比使用 `let arrCopy = Object.assign([], arr);` 来复制数组，或使用 `let objCopy = Object.assign({}, obj);` 来复制对象写起来要短得多。因此，只要情况允许，我们更喜欢使用它。

总结

当我们在代码中看到 `"..."` 时，它要么是 `rest` 参数，要么就是 `spread` 语法。

有一个简单的方法可以区分它们：

- 若 `...` 出现在函数参数列表的最后，那么它就是 `rest` 参数，它会把参数列表中剩余的参数收集到一个数组中。
- 若 `...` 出现在函数调用或类似的表达式中，那它就是 `spread` 语法，它会把一个数组展开为列表。

使用场景：

- `Rest` 参数用于创建可接受任意数量参数的函数。
- `Spread` 语法用于将数组传递给通常需要含有许多参数的列表的函数。

它们俩的出现帮助我们轻松地在列表和参数数组之间来回转换。

“旧式”的 `arguments`（类数组对象）也依然能够帮助我们获取函数调用中的所有参数。

闭包

JavaScript 是一种非常面向函数的语言。它给了我们很大的自由度。在 JavaScript 中，我们可以动态创建函数，可以将函数作为参数传递给另一个函数，并在完全不同的代码位置进行调用。

我们已经知道函数可以访问其外部的变量。

现在，让我们扩展知识，来看看更复杂的场景。

① 我们将在这探讨一下 `let/const`

在 JavaScript 中，有三种声明变量的方式：`let`，`const`（现代方式），`var`（过去留下来的方式）。

- 在本文的示例中，我们将使用 `let` 声明变量。
- 用 `const` 声明的变量的行为也相同（译注：与 `let` 在作用域等特性上是相同的），因此，本文也涉及用 `const` 进行变量声明。
- 旧的 `var` 与上面两个有着明显的区别，我们将在 [旧时的 "var"](#) 中详细介绍。

代码块

如果在代码块 `{...}` 内声明了一个变量，那么这个变量只在该代码块内可见。

例如：

```
{
  // 使用在代码块外不可见的局部变量做一些工作

  let message = "Hello"; // 只在此代码块内可见

  alert(message); // Hello
}

alert(message); // Error: message is not defined
```

我们可以使用它来隔离一段代码，该段代码执行自己的任务，并使用仅属于自己的变量：

```
{  
  // 显示 message  
  let message = "Hello";  
  alert(message);  
}  
  
{  
  // 显示另一个 message  
  let message = "Goodbye";  
  alert(message);  
}
```

❶ 这里如果没有代码块则会报错

请注意，如果我们使用 `let` 对已存在的变量进行重复声明，如果对应的变量没有单独的代码块，则会出现错误：

```
// 显示 message  
let message = "Hello";  
alert(message);  
  
// 显示另一个 message  
let message = "Goodbye"; // Error: variable already declared  
alert(message);
```

对于 `if`, `for` 和 `while` 等，在 `{...}` 中声明的变量也仅在内部可见：

```
if (true) {  
  let phrase = "Hello!";  
  
  alert(phrase); // Hello!  
}  
  
alert(phrase); // Error, no such variable!
```

在这儿，当 `if` 执行完毕，则下面的 `alert` 将看不到 `phrase`，因此会出现错误。（译注：就算下面的 `alert` 想在 `if` 没执行完成时去取 `phrase`（虽然这种情况不可能发生）也是取不到的，因为 `let` 声明的变量在代码块外不可见。）

太好了，因为这就允许我们创建特定于 `if` 分支的块级局部变量。

对于 `for` 和 `while` 循环也是如此：

```
for (let i = 0; i < 3; i++) {  
  // 变量 i 仅在这个 for 循环的内部可见  
  alert(i); // 0, 然后是 1, 然后是 2  
}  
  
alert(i); // Error, no such variable
```

从视觉上看，`let i` 位于 `{...}` 之外。但是 `for` 构造在这里很特殊：在其中声明的变量被视为块的一部分。

嵌套函数

当一个函数是在另一个函数中创建的时，那么该函数就被称为“嵌套”的。

在 JavaScript 中很容易实现这一点。

我们可以使用嵌套来组织代码，比如这样：

```
function sayHiBye(firstName, lastName) {  
  
    // 辅助嵌套函数使用如下  
    function getFullName() {  
        return firstName + " " + lastName;  
    }  
  
    alert( "Hello, " + getFullName() );  
    alert( "Bye, " + getFullName() );  
  
}
```

这里创建的 **嵌套** 函数 `getFullName()` 是为了更加方便。它可以访问外部变量，因此可以返回全名。嵌套函数在 JavaScript 中很常见。

更有意思的是，可以返回一个嵌套函数：作为一个新对象的属性或作为结果返回。之后可以在其他地方使用。不论在哪里调用，它仍然可以访问相同的外部变量。

下面的 `makeCounter` 创建来一个“counter”函数，该函数在每次调用时返回下一个数字：

```
function makeCounter() {  
    let count = 0;  
  
    return function() {  
        return count++;  
    };  
}  
  
let counter = makeCounter();  
  
alert( counter() ); // 0  
alert( counter() ); // 1  
alert( counter() ); // 2
```

尽管很简单，但稍加变型就具有很强的实际用途，比如，用作 [随机数生成器](#) 以生成用于自动化测试的随机数值。

这是如何运作的呢？如果我们创建多个计数器，它们会是独立的吗？这里的变量是怎么回事？

理解这些内容对于掌握 JavaScript 的整体知识很有帮助，并且对于应对更复杂的场景也很有益处。因此，让我们继续深入探究。

词法环境

⚠ 这是龙！

深入的技术讲解就在下面。

尽管我很想避免编程语言的一些底层细节，但是如果没有这些细节，它们就不完整，所以请准备开始学习吧！

为了使内容更清晰，这里将分步骤进行讲解。

Step 1. 变量

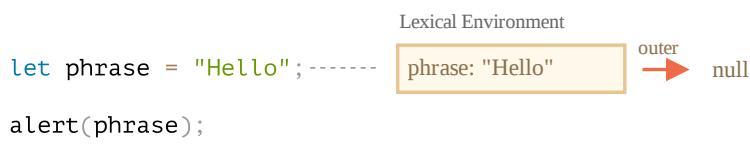
在 JavaScript 中，每个运行的函数，代码块 `{ ... }` 以及整个脚本，都有一个被称为 **词法环境** (**Lexical Environment**) 的内部（隐藏）的关联对象。

词法环境对象由两部分组成：

1. **环境记录 (Environment Record)** —— 一个存储所有局部变量作为其属性（包括一些其他信息，例如 `this` 的值）的对象。
2. 对 **外部词法环境** 的引用，与外部代码相关联。

一个“变量”只是 **环境记录** 这个特殊的内部对象的一个属性。“获取或修改变量”意味着“获取或修改词法环境的一个属性”。

举个例子，这段没有函数的简单的代码中只有一个词法环境：

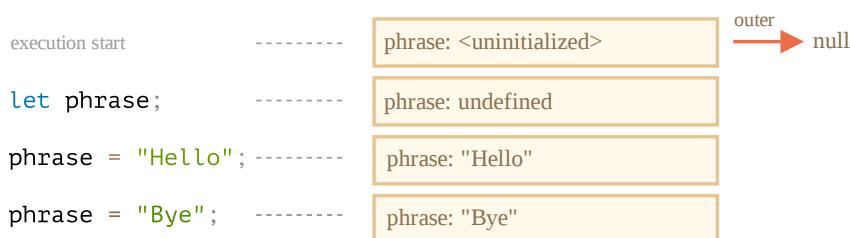


这就是所谓的与整个脚本相关联的 **全局** 词法环境。

在上面的图片中，矩形表示环境记录（变量存储），箭头表示外部引用。全局词法环境没有外部引用，所以箭头指向了 `null`。

随着代码开始并继续运行，词法环境发生了变化。

这是更长的代码：



右侧的矩形演示了执行过程中全局词法环境的变化：

1. 当脚本开始运行，词法环境预先填充了所有声明的变量。
 - 最初，它们处于“未初始化（Uninitialized）”状态。这是一种特殊的内部状态，这意味着引擎知道变量，但是在用 `let` 声明前，不能引用它。几乎就像变量不存在一样。
2. 然后 `let phrase` 定义出现了。它尚未被赋值，因此它的值为 `undefined`。从这一刻起，我们就可以使用变量了。
3. `phrase` 被赋予了一个值。

4. `phrase` 的值被修改。

现在看起来都挺简单的，是吧？

- 变量是特殊内部对象的属性，与当前正在执行的（代码）块/函数/脚本有关。
- 操作变量实际上是操作该对象的属性。

① 词法环境是一个规范对象

“词法环境”是一个规范对象（specification object）：它仅仅是存在于[编程语言规范](#)中的“理论上”存在的，用于描述事物如何运作的对象。我们无法在代码中获取该对象并直接对其进行操作。

但 JavaScript 引擎同样可以优化它，比如清除未被使用的变量以节省内存和执行其他内部技巧等，但显性行为应该是和上述的无差。

Step 2. 函数声明

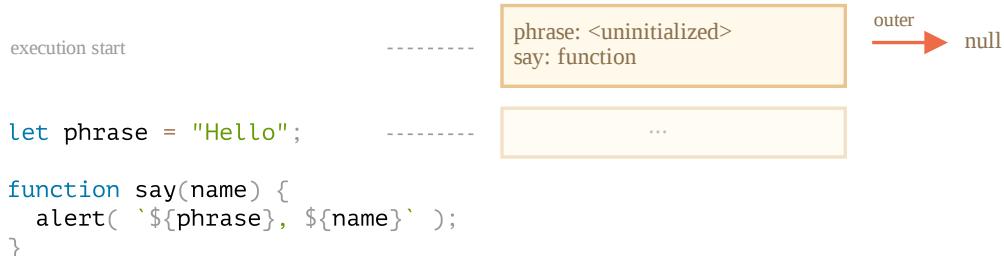
一个函数其实也是一个值，就像变量一样。

不同之处在于函数声明的初始化会被立即完成。

当创建了一个词法环境（Lexical Environment）时，函数声明会立即变为即用型函数（不像 `let` 那样直到声明处才可用）。

这就是为什么我们可以在（函数声明）的定义之前调用函数声明。

例如，这是添加一个函数时全局词法环境的初始状态：

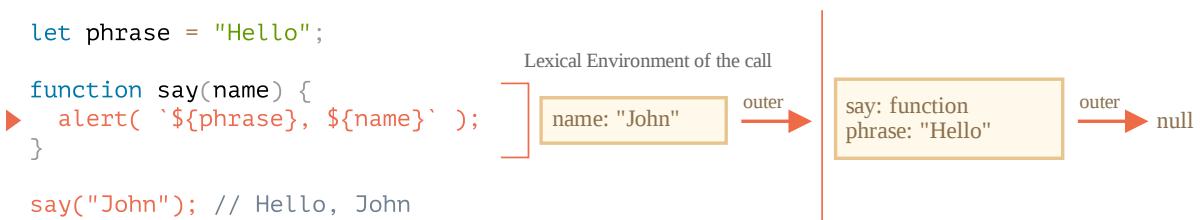


正常来说，这种行为仅适用于函数声明，而不适用于我们将函数分配给变量的函数表达式，例如 `let say = function(name)...`。

Step 3. 内部和外部的词法环境

在一个函数运行时，在调用刚开始时，会自动创建一个新的词法环境以存储这个调用的局部变量和参数。

例如，对于 `say("John")`，它看起来像这样（当前执行位置在箭头标记的那一行上）：



在这个函数调用期间，我们有两个词法环境：内部一个（用于函数调用）和外部一个（全局）：

- 内部词法环境与 `say` 的当前执行相对应。它具有一个单独的属性: `name`, 函数的参数。我们调用的是 `say("John")`, 所以 `name` 的值为 `"John"`。
- 外部词法环境是全局词法环境。它具有 `phrase` 变量和函数本身。

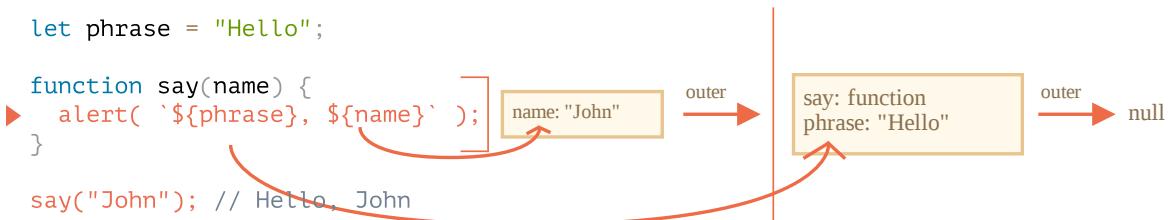
内部词法环境引用了 `outer`。

当代码要访问一个变量时 —— 首先会搜索内部词法环境，然后搜索外部环境，然后搜索更外部的环境，以此类推，直到全局词法环境。

如果在任何地方都找不到这个变量，那么在严格模式下就会报错（在非严格模式下，为了向下兼容，给未定义的变量赋值会创建一个全局变量）。

在这个示例中，搜索过程如下：

- 对于 `name` 变量，当 `say` 中的 `alert` 试图访问 `name` 时，会立即在内部词法环境中找到它。
- 当它试图访问 `phrase` 时，然而内部没有 `phrase`，所以它顺着对外部词法环境的引用找到了它。



Step 4. 返回函数

让我们回到 `makeCounter` 这个例子。

```

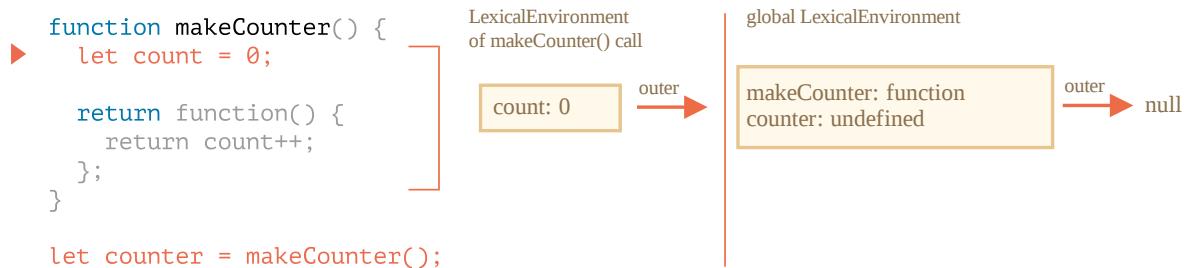
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
  
```

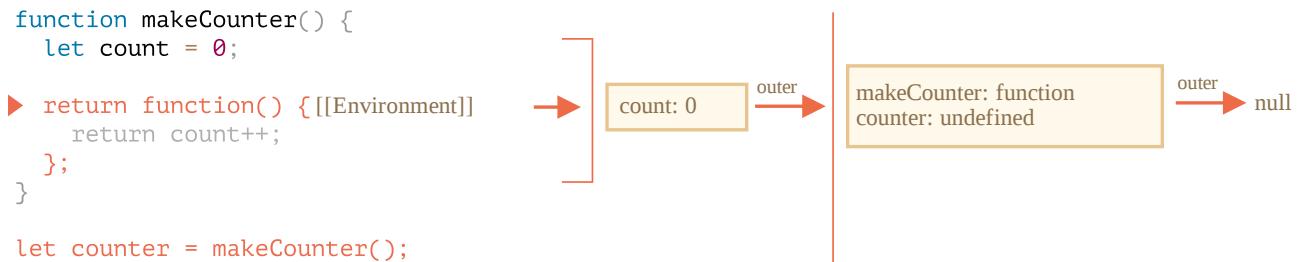
在每次 `makeCounter()` 调用的开始，都会创建一个新的词法环境对象，以存储该 `makeCounter` 运行时的变量。

因此，我们有两层嵌套的词法环境，就像上面的示例一样：



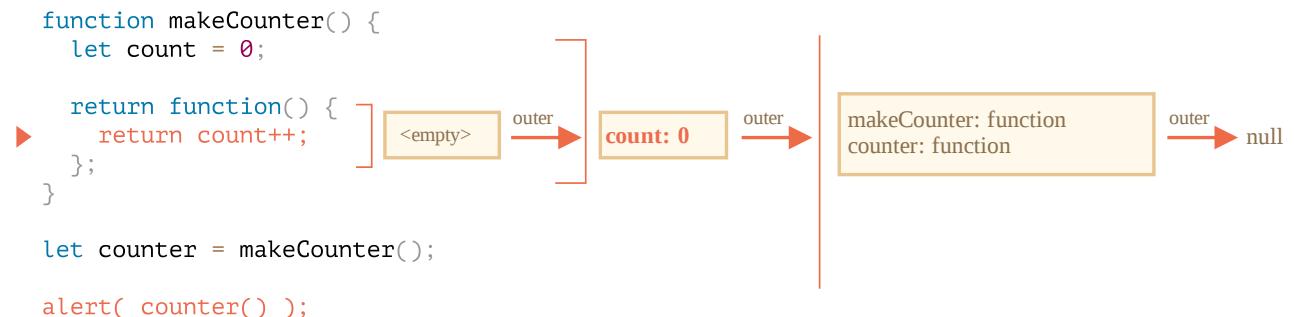
不同的是，在执行 `makeCounter()` 的过程中创建了一个仅占一行的嵌套函数：`return count++`。我们尚未运行它，仅创建了它。

所有的函数在“诞生”时都会记住创建它们的词法环境。从技术上讲，这里没有什么魔法：所有函数都有名为 `[[Environment]]` 的隐藏属性，该属性保存了对创建该函数的词法环境的引用。



因此，`counter.[[Environment]]` 有对 `{count: 0}` 词法环境的引用。这就是函数记住它创建于何处的方式，与函数被在哪儿调用无关。`[[Environment]]` 引用在函数创建时被设置并永久保存。

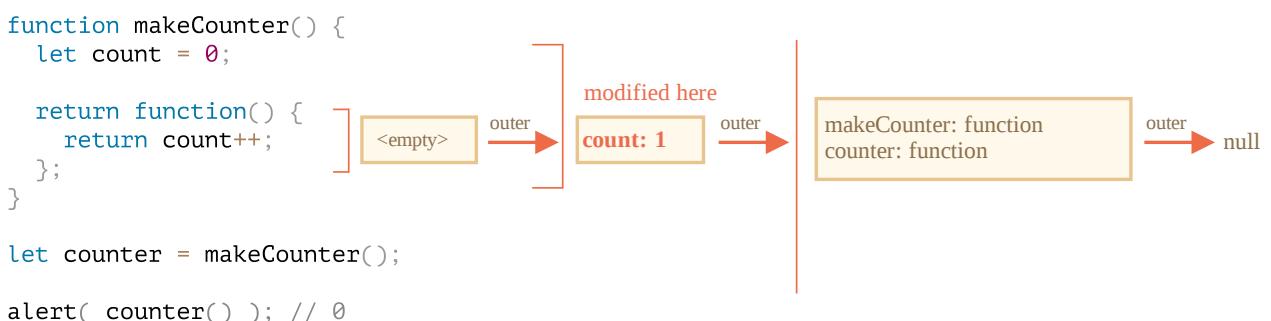
稍后，当调用 `counter()` 时，会为该调用创建一个新的词法环境，并且其外部词法环境引用获取于 `counter.[[Environment]]`：



现在，当 `counter()` 中的代码查找 `count` 变量时，它首先搜索自己的词法环境（为空，因为那里没有局部变量），然后是外部 `makeCounter()` 的词法环境，并且在那里找到就在哪里修改。

在变量所在的词法环境中更新变量。

这是执行后的状态：



如果我们调用 `counter()` 多次，`count` 变量将在同一位置增加到 `2`, `3` 等。

① 闭包

开发者通常应该都知道“闭包”这个通用的编程术语。

闭包 [🔗](#) 是指内部函数总是可以访问其所在的外部函数中声明的变量和参数，即使在其外部函数被返回（寿命终结）了之后。在某些编程语言中，这是不可能的，或者应该以特殊的方式编写函数来实现。但是如上所述，在 **JavaScript** 中，所有函数都是天生闭包的（只有一个例外，将在 “[new Function](#)” 语法 中讲到）。

也就是说： **JavaScript** 中的函数会自动通过隐藏的 `[[Environment]]` 属性记住创建它们的位置，所以它们都可以访问外部变量。

在面试时，前端开发者通常会被问到“什么是闭包？”，正确的回答应该是闭包的定义，并解释清楚为什么 **JavaScript** 中的所有函数都是闭包的，以及可能的关于 `[[Environment]]` 属性和词法环境原理的技术细节。

垃圾收集

通常，函数调用完成后，会将词法环境和其中的所有变量从内存中删除。因为现在没有任何对它们的引用了。与 **JavaScript** 中的任何其他对象一样，词法环境仅在可达时才会被保留在内存中。

.....但是，如果有一个嵌套函数在函数结束后仍可达，则它具有引用词法环境的 `[[Environment]]` 属性。

在下面这个例子中，即使在函数执行完成后，词法环境仍然可达。因此，此嵌套函数仍然有效。

例如：

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // g.[[Environment]] 存储了对相应 f() 调用的词法环境的引用
```

请注意，如果多次调用 `f()`，并且返回的函数被保存，那么所有相应的词法环境对象也会保留在内存中。下面代码中有三个这样的函数：

```
function f() {
  let value = Math.random();

  return function() { alert(value); };
}

// 数组中的 3 个函数，每个都与来自对应的 f() 的词法环境相关联
let arr = [f(), f(), f()];
```

当词汇环境对象变得不可达时，它就会死去（就像其他任何对象一样）。换句话说，它仅在至少有一个嵌套函数引用它时才存在。

在下面的代码中，嵌套函数被删除后，其封闭的词法环境（以及其中的 `value`）也会被从内存中删除：

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // 当 g 函数存在时，该值会被保留在内存中

g = null; // .....现在内存被清理了
```

实际开发中的优化

正如我们所看到的，理论上当函数可达时，它外部的所有变量也都将存在。

但在实际中，JavaScript 引擎会试图优化它。它们会分析变量的使用情况，如果从代码中可以明显看出有未使用的外部变量，那么就会将其删除。

在 V8（Chrome, Opera）中的一个重要的副作用是，此类变量在调试中将不可用。

打开 Chrome 浏览器的开发者工具，并尝试运行下面的代码。

当代码执行暂停时，在控制台中输入 `alert(value)`。

```
function f() {
  let value = Math.random();

  function g() {
    debugger; // 在 Console 中：输入 alert(value); No such variable!
  }

  return g;
}

let g = f();
g();
```

正如你所见的——`No such variable!` 理论上，它应该是可以访问的，但引擎把它优化掉了。

这可能会导致有趣的（如果不是那么耗时的）调试问题。其中之一——我们可以看到的是一个同名的外部变量，而不是预期的变量：

```
let value = "Surprise!";

function f() {
  let value = "the closest value";

  function g() {
    debugger; // 在 console 中：输入 alert(value); Surprise!
  }

  return g;
}
```

```
}
```

```
let g = f();
```

```
g();
```

V8 引擎的这个特性你真的应该知道。如果你要使用 Chrome/Opera 进行代码调试，迟早会遇到这样的问题。

这不是调试器的 bug，而是 V8 的一个特别的特性。也许以后会被修改。你始终可以通过运行本文中的示例来进行检查。

补充内容

① 说明

为了更清晰地讲解闭包，本文经过大幅重写，以下内容是重写时部分被优化掉的内容，译者认为还是很有学习价值的，遂保留下来供大家学习。

代码块

我们可以使用“空”的代码块将变量隔离到“局部作用域”中。

比如，在 Web 浏览器中，所有脚本都共享同一个全局环境。如果我们在一个脚本中创建一个全局变量，对于其他脚本来说它也是可用的。但是如果两个脚本有使用同一个变量并且相互覆盖，那么这会成为冲突的根源。

如果变量名是一个被广泛使用的词，并且不同脚本的作者可能彼此也不知道。

如果我们要避免这个，我们可以使用代码块来隔离整个脚本或其中一部分：

```
{  
  // 用局部变量完成一些不应该被外面访问的工作  
  
  let message = "Hello";  
  
  alert(message); // Hello  
}  
  
alert(message); // Error: message is not defined
```

这是因为代码块有其自身的词法环境，块之外（或另一个脚本内）的代码访问不到代码块内的变量。

旧时的 "var"

① 本文用于帮助理解旧脚本

本文所讲的内容对于帮助理解旧脚本很有用。

但这不是我们编写新代码的方式。

在本教程最开始那部分的 [变量](#) 这章中，我们提到了变量声明的三种方式：

1. `let`

2. `const`

3. `var`

`let` 和 `const` 在词法环境中的行为完全一样。

但 `var` 却是一头完全不同的，源自非常古老的时代的怪兽。在现代脚本中一般不再使用它，但它仍然潜伏在旧脚本中。

如果你不打算接触这样的脚本，你甚至可以跳过本章或推迟阅读本章，但是之后你很可能会踩到它的坑。

乍一看，`var` 和 `let` 的行为相似，不就是声明变量嘛：

```
function sayHi() {  
  var phrase = "Hello"; // 局部变量，使用 "var"，而不是 "let"  
  
  alert(phrase); // Hello  
}  
  
sayHi();  
  
alert(phrase); // Error, phrase is not defined
```

.....但两者存在区别。

“var”没有块级作用域

用 `var` 声明的变量，不是函数作用域就是全局作用域。它们在代码块外也是可见的（译注：也就是说，`var` 声明的变量只有函数作用域和全局作用域，没有块级作用域）。

举个例子：

```
if (true) {  
  var test = true; // 使用 "var" 而不是 "let"  
}  
  
alert(test); // true, 变量在 if 结束后仍存在
```

由于 `var` 会忽略代码块，因此我们有了一个全局变量 `test`。

如果我们在第二行使用 `let test` 而不是 `var test`，那么该变量将仅在 `if` 内部可见：

```
if (true) {  
  let test = true; // 使用 "let"  
}  
  
alert(test); // Error: test is not defined
```

对于循环也是这样的，`var` 声明的变量没有块级作用域也没有循环局部作用域：

```
for (var i = 0; i < 10; i++) {  
    // ...  
}  
  
alert(i); // 10, "i" 在循环结束后仍可见, 它是一个全局变量
```

如果一个代码块位于函数内部, 那么 `var` 声明的变量的作用域将为函数作用域:

```
function sayHi() {  
    if (true) {  
        var phrase = "Hello";  
    }  
  
    alert(phrase); // works  
}  
  
sayHi();  
alert(phrase); // Error: phrase is not defined (Check the Developer Console)
```

可以看到, `var` 穿透了 `if`, `for` 和其它代码块。这是因为在早期的 JavaScript 中, 块没有词法环境。而 `var` 就是这个时期的代表之一。

“var” 声明在函数开头就会被处理

当函数开始的时候, 就会处理 `var` 声明 (脚本启动对应全局变量) 。

换言之, `var` 声明的变量会在函数开头被定义, 与它在代码中定义的位置无关 (这里不考虑定义在嵌套函数中的情况) 。

那么看一下这段代码:

```
function sayHi() {  
    phrase = "Hello";  
  
    alert(phrase);  
  
    var phrase;  
}  
sayHi();
```

.....从技术上讲, 它与下面这种情况是一样的 (`var phrase` 被上移至函数开头) :

```
function sayHi() {  
    var phrase;  
  
    phrase = "Hello";  
  
    alert(phrase);  
}  
sayHi();
```

.....甚至与这种情况也一样（记住，代码块是会被忽略的）：

```
function sayHi() {  
    phrase = "Hello"; // (*)  
  
    if (false) {  
        var phrase;  
    }  
  
    alert(phrase);  
}  
sayHi();
```

人们将这种行为称为“提升”（英文为“hoisting”或“raising”），因为所有的 `var` 都被“提升”到了函数的顶部。

所以，在上面的例子中，`if (false)` 分支永远都不会执行，但没关系，它里面的 `var` 在函数刚开始时就被处理了，所以在执行 `(*)` 那行代码时，变量是存在的。

声明会被提升，但是赋值不会。

我们最好用例子来说明：

```
function sayHi() {  
    alert(phrase);  
  
    var phrase = "Hello";  
}  
  
sayHi();
```

`var phrase = "Hello"` 这行代码包含两个行为：

1. 使用 `var` 声明变量
2. 使用 `=` 给变量赋值。

声明在函数刚开始执行的时候（“提升”）就被处理了，但是赋值操作始终是在它出现的地方才起作用。所以这段代码实际上是这样工作的：

```
function sayHi() {  
    var phrase; // 在函数刚开始时进行变量声明  
  
    alert(phrase); // undefined  
  
    phrase = "Hello"; // .....赋值 – 当程序执行到这一行时。  
}  
  
sayHi();
```

因为所有的 `var` 声明都是在函数开头处理的，我们可以在任何地方引用它们。但是在它们被赋值之前都是 `undefined`。

上面两个例子中 `alert` 运行都不会报错，因为变量 `phrase` 是存在的。但是它还没有被赋值，所以显示 `undefined`。

IIFE

在之前，JavaScript 中只有 `var` 这一种声明变量的方式，并且这种方式声明的变量没有块级作用域，程序员们就发明了一种模仿块级作用域的方法。这种方法被称为“立即调用函数表达式”（immediately-invoked function expressions, IIFE）。

如今，我们不应该再使用 IIFE 了，但是你可以在旧脚本中找到它们。

IIFE 看起来像这样：

```
(function() {  
  let message = "Hello";  
  alert(message); // Hello  
})();
```

这里创建了一个函数表达式并立即调用。因此，代码立即执行并拥有了自己的私有变量。

函数表达式被括号 `(function {...})` 包裹起来，因为在 JavaScript 中，当主代码流遇到 `"function"` 时，它会把它当成一个函数声明的开始。但函数声明必须有一个函数名，所以这样的代码会导致错误：

```
// 尝试声明并立即调用一个函数  
function() { // <-- Error: Function statements require a function name  
  
  let message = "Hello";  
  
  alert(message); // Hello  
  
}();
```

即使我们说：“好吧，那我们加一个名称吧”，但它仍然不工作，因为 JavaScript 不允许立即调用函数声明：

```
// 下面的括号会导致语法错误  
function go() {  
  
}(); // <-- 不能立即调用函数声明
```

因此，需要使用圆括号把告函数表达式包起来，以告诉 JavaScript，这个函数是在另一个表达式的上下文中创建的，因此它是一个函数表达式：它不需要函数名，可以立即调用。

除了使用括号，还有其他方式可以告诉 JavaScript 在这我们指的是函数表达式：

```
// 创建 IIFE 的方法  
  
(function() {  
  alert("Parentheses around the function");
```

```
})();
(function() {
  alert("Parentheses around the whole thing");
})();

(function() {
  alert("Bitwise NOT operator starts the expression");
})();

(function() {
  alert("Unary plus starts the expression");
})();
```

在上面的所有情况中，我们都声明了一个函数表达式并立即运行它。请再注意一下：如今我们没有理由来编写这样的代码。

总结

`var` 与 `let/const` 有两个主要的区别：

1. `var` 声明的变量没有块级作用域，它们的最小作用域就是函数级作用域。
2. `var` 变量声明在函数开头就会被处理（脚本启动对应全局变量）。

涉及全局对象时，还有一个非常小的差异，我们将在下一章中介绍。

这些差异使 `var` 在大多数情况下都比 `let` 更糟糕。块级作用域是这么好的一个东西。这就是 `let` 在几年前就被写入到标准中的原因，并且现在（与 `const` 一起）已经成为了声明变量的主要方式。

全局对象

全局对象提供可在任何地方使用的变量和函数。默认情况下，这些全局变量内置于语言或环境中。在浏览器中，它的名字是“`window`”，对 `Node.js` 而言，它的名字是“`global`”，其它环境可能用的是别的名字。

最近，`globalThis` 被作为全局对象的标准名称加入到了 `JavaScript` 中，所有环境都应该支持该名称。在有些浏览器中，即 non-Chromium Edge，尚不支持 `globalThis`，但可以很容易地对其进行填充（`polyfilled`）。

假设我们的环境是浏览器，我们将在这儿使用“`window`”。如果你的脚本可能会用来在其他环境中运行，则最好使用 `globalThis`。

全局对象的所有属性都可以被直接访问：

```
alert("Hello");
// 等同于
window.alert("Hello");
```

在浏览器中，使用 `var`（而不是 `let/const`！）声明的全局函数和变量会成为全局对象的属性。

```
var gVar = 5;

alert(window.gVar); // 5 (成为了全局对象的属性)
```

请不要依赖它！这种行为是出于兼容性而存在的。现代脚本通过使用 [JavaScript modules](#) 来避免这种情况的发生。

如果我们使用 `let`，就不会发生这种情况：

```
let gLet = 5;

alert(window.gLet); // undefined (不会成为全局对象的属性)
```

如果一个值非常重要，以至于你想使它在全局范围内可用，那么可以直接将其作为属性写入：

```
// 将当前用户信息全局化，以允许所有脚本访问它
window.currentUser = {
  name: "John"
};

// 代码中的另一个位置
alert(currentUser.name); // John

// 或者，如果我们有一个名为 "currentUser" 的局部变量
// 从 window 显示地获取它（这是安全的！）
alert(window.currentUser.name); // John
```

也就是说，一般不建议使用全局变量。全局变量应尽可能的少。与使用外部变量或全局变量相比，函数获取“输入”变量并产生特定“输出”的代码设计更加清晰，不易出错且更易于测试。

使用 polyfills

我们使用全局对象来测试对现代语言功能的支持。

例如，测试是否存在内建的 `Promise` 对象（在版本特别旧的浏览器中不存在）：

```
if (!window.Promise) {
  alert("Your browser is really old!");
}
```

如果没有（例如，我们使用的是旧版浏览器），那么我们可以创建“polyfills”：添加环境不支持但在现代标准中存在的功能。

```
if (!window.Promise) {
  window.Promise = ... // 定制实现现代语言功能
}
```

总结

- 全局对象包含应该在任何位置都可见的变量。

其中包括 JavaScript 的内建方法，例如“Array”和环境特定（environment-specific）的值，例如 `window.innerHeight` — 浏览器中的窗口高度。

- 全局对象有一个通用名称 `globalThis`。

.....但是更常见的是使用“老式”的环境特定（environment-specific）的名字，例如 `window`（浏览器）和 `global`（Node.js）。由于 `globalThis` 是最近的提议，因此在 non-Chromium Edge 中不受支持（但可以进行 polyfills）。

- 仅当值对于我们的项目而言确实是全局的时，才应将其存储在全局对象中。并保持其数量最少。
- 在浏览器中，除非我们使用 `modules`，否则使用 `var` 声明的全局函数和变量会成为全局对象的属性。
- 为了使我们的代码面向未来并更易于理解，我们应该使用直接的方式访问全局对象的属性，如 `window.x`。

函数对象，NFE

我们已经知道，在 JavaScript 中，函数就是值。

JavaScript 中的每个值都有一种类型，那么函数是什么类型呢？

在 JavaScript 中，函数就是对象。

一个容易理解的方式是把函数想象成可被调用的“行为对象（action object）”。我们不仅可以调用它们，还能把它们当作对象来处理：增/删属性，按引用传递等。

属性“name”

函数对象包含一些便于使用的属性。

比如，一个函数的名字可以通过属性“`name`”来访问：

```
function sayHi() {
  alert("Hi");
}

alert(sayHi.name); // sayHi
```

更有趣的是，名称赋值的逻辑很智能。即使函数被创建时没有名字，名称赋值的逻辑也能给它赋予一个正确的名字，然后进行赋值：

```
let sayHi = function() {
  alert("Hi");
};

alert(sayHi.name); // sayHi (有名字! )
```

当以默认值的方式完成了赋值时，它也有效：

```
function f(sayHi = function() {}) {
  alert(sayHi.name); // sayHi (生效了!)
}

f();
```

规范中把这种特性叫做「上下文命名」。如果函数自己没有提供，那么在赋值中，会根据上下文来推测一个。

对象方法也有名字：

```
let user = {

  sayHi() {
    // ...
  },

  sayBye: function() {
    // ...
  }
}

alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

这没有什么神奇的。有时会出现无法推测名字的情况。此时，属性 `name` 会是空，像这样：

```
// 函数是在数组中创建的
let arr = [function() {}];

alert( arr[0].name ); // <空字符串>
// 引擎无法设置正确的名字，所以没有值
```

而实际上，大多数函数都是有名字的。

属性 “`length`”

还有另一个内置属性 “`length`”，它返回函数入参的个数，比如：

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

可以看到，余参不参与计数。

属性 `length` 有时用于对其他函数进行操作的函数的 [自我检查 \(introspection\)](#) 。在函数中操作其它函数的内省。

比如，下面的代码中函数 `ask` 接受一个询问答案的参数 `question` 和可能包含任意数量 `handler` 的参数 `...handlers`。

当用户提供了自己的答案后，函数会调用那些 `handlers`。我们可以传入两种 `handlers`：

- 一种是无参函数，它仅在用户回答给出积极的答案时被调用。
- 一种是有参函数，它在两种情况都会被调用，并且返回一个答案。

为了正确地调用 `handler`，我们需要检查 `handler.length` 属性。

我们的想法是，我们用一个简单的无参数的 `handler` 语法来处理积极的回答（最常见的变体），但也要能够提供通用的 `handler`:

```
function ask(question, ...handlers) {
  let isYes = confirm(question);

  for(let handler of handlers) {
    if (handler.length == 0) {
      if (isYes) handler();
    } else {
      handler(isYes);
    }
  }
}

// 对于积极的回答，两个 handler 都会被调用
// 对于负面的回答，只有第二个 handler 被调用
ask("Question?", () => alert('You said yes'), result => alert(result));
```

这种特别的情况就是所谓的 [多态性](#) —— 根据参数的类型，或者根据在我们的具体情景下的 `length` 来做不同的处理。这种思想在 JavaScript 的库里有应用。

自定义属性

我们也可以添加我们自己的属性。

这里我们添加了 `counter` 属性，用来跟踪总的调用次数：

```
function sayHi() {
  alert("Hi");

  // 计算调用次数
  sayHi.counter++;
}

sayHi.counter = 0; // 初始值

sayHi(); // Hi
sayHi(); // Hi

alert(`Called ${sayHi.counter} times`); // Called 2 times
```

⚠ 属性不是变量

被赋值给函数的属性，比如 `sayHi.counter = 0`，不会在函数内定义一个局部变量 `counter`。换句话说，属性 `counter` 和变量 `let counter` 是毫不相关的两个东西。

我们可以把函数当作对象，在它里面存储属性，但是这对它的执行没有任何影响。变量不是函数属性，反之亦然。它们之间是平行的。

函数属性有时会用来替代闭包。例如，我们可以使用函数属性将 [闭包](#) 章节中 `counter` 函数的例子进行重写：

```
function makeCounter() {
  // 不需要这个了
  // let count = 0

  function counter() {
    return counter.count++;
  }

  counter.count = 0;

  return counter;
}

let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
```

现在 `count` 被直接存储在函数里，而不是它外部的词法环境。

那么它和闭包谁好谁赖？

两者最大的不同就是如果 `count` 的值位于外层（函数）变量中，那么外部的代码无法访问到它，只有嵌套的函数可以修改它。而如果它是绑定到函数的，那么就很容易：

```
function makeCounter() {

  function counter() {
    return counter.count++;
  }

  counter.count = 0;

  return counter;
}

let counter = makeCounter();

counter.count = 10;
alert( counter() ); // 10
```

所以，选择哪种实现方式取决于我们的需求是什么。

命名函数表达式

命名函数表达式（NFE, Named Function Expression），指带有名字的函数表达式的术语。

例如，让我们写一个普通的函数表达式：

```
let sayHi = function(who) {
  alert(`Hello, ${who}`);
};
```

然后给它加一个名字：

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
```

我们这里得到了什么吗？为它添加一个 "func" 名字的目的是什么？

首先请注意，它仍然是一个函数表达式。在 `function` 后面加一个名字 "func" 没有使它成为一个函数声明，因为它仍然是作为赋值表达式中的一部分被创建的。

添加这个名字当然也没有打破任何东西。

函数依然可以通过 `sayHi()` 来调用：

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};

sayHi("John"); // Hello, John
```

关于名字 `func` 有两个特殊的地方，这就是添加它的原因：

1. 它允许函数在内部引用自己。
2. 它在函数外是不可见的。

例如，下面的函数 `sayHi` 会在没有入参 `who` 时，以 "Guest" 为入参调用自己：

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // 使用 func 再次调用函数自身
  }
};

sayHi(); // Hello, Guest

// 但这不工作：
func(); // Error, func is not defined (在函数外不可见)
```

我们为什么使用 `func` 呢？为什么不直接使用 `sayHi` 进行嵌套调用？

当然，在大多数情况下我们可以这样做：

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest");
  }
};
```

上面这段代码的问题在于 `sayHi` 的值可能会被函数外部的代码改变。如果该函数被赋值给另外一个变量（译注：也就是原变量被修改），那么函数就会开始报错：

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Error: sayHi is not a function
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Error, 嵌套调用 sayHi 不再有效!
```

发生这种情况是因为该函数从它的外部词法环境获取 `sayHi`。没有局部的 `sayHi` 了，所以使用外部变量。而当调用时，外部的 `sayHi` 是 `null`。

我们给函数表达式添加的可选的名字，正是用来解决这类问题的。

让我们使用它来修复我们的代码：

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // 现在一切正常
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (嵌套调用有效)
```

现在它可以正常运行了，因为名字 `func` 是函数局部域的。它不是从外部获取的（而且它对外部也是不可见的）。规范确保它只会引用当前函数。

外部代码仍然有该函数的 `sayHi` 或 `welcome` 变量。而且 `func` 是一个“内部函数名”，可用于函数在自身内部进行自调用。

① 函数声明没有这个东西

这里所讲的“内部名”特性只针对函数表达式，而不是函数声明。对于函数声明，没有用来添加“内部”名的语法。

有时，当我们需要一个可靠的内部名时，这就成为了你把函数声明重写成函数表达式的原因了。

总结

函数就是对象。

我们介绍了它们的一些属性：

- `name` — 函数的名字。通常取自函数定义，但如果函数定义时没设定函数名，JavaScript 会尝试通过函数的上下文猜一个函数名（例如把赋值的变量名取为函数名）。
- `length` — 函数定义时的入参的个数。Rest 参数不参与计数。

如果函数是通过函数表达式的形式被声明的（不是在主代码流里），并且附带了名字，那么它被称为命名函数表达式（**Named Function Expression**）。这个名字可以用于在该函数内部进行自调用，例如递归调用等。

此外，函数可以带有额外的属性。很多知名的 JavaScript 库都充分利用了这个功能。

它们创建一个“主”函数，然后给它附加很多其它“辅助”函数。例如，[jQuery ↗](#) 库创建了一个名为 `$` 的函数。[lodash ↗](#) 库创建一个 `_` 函数，然后为其添加了 `_.add`、`_.keyBy` 以及其它属性（欲了解详情，参见 [docs ↗](#)）。实际上，它们这么做是为了减少对全局空间的污染，这样一个库就只会有一个全局变量。这样就降低了命名冲突的可能性。

所以，一个函数本身可以完成一项有用的工作，还可以在自身的属性中附带许多其他功能。

"new Function" 语法

还有一种创建函数的方法。它很少被使用，但有些时候只能选择它。

语法

创建函数的语法：

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

该函数是通过使用参数 `arg1...argN` 和给定的 `functionBody` 创建的。

下面这个例子可以帮助你理解创建语法。这是一个带有两个参数的函数：

```
let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3
```

这里有一个没有参数的函数，只有函数体：

```
let sayHi = new Function('alert("Hello")');

sayHi(); // Hello
```

与我们已知的其他方法相比，这种方法最大的不同在于，它实际上是通过运行时通过参数传递过来的字符串创建的。

以前的所有声明方法都需要我们——程序员，在脚本中编写函数的代码。

但是 `new Function` 允许我们将任意字符串变为函数。例如，我们可以从服务器接收一个新的函数并执行它：

```
let str = ... 动态地接收来自服务器的代码 ...

let func = new Function(str);
func();
```

使用 `new Function` 创建函数的应用场景非常特殊，比如在复杂的 Web 应用程序中，我们需要从服务器获取代码或者动态地从模板编译函数时才会使用。

闭包

通常，闭包是指使用一个特殊的属性 `[[Environment]]` 来记录函数自身的创建时的环境的函数。它具体指向了函数创建时的词法环境。（我们在 [闭包](#) 一章中对此进行了详细的讲解）。

但是如果我们使用 `new Function` 创建一个函数，那么该函数的 `[[Environment]]` 并不指向当前的词法环境，而是指向全局环境。

因此，此类函数无法访问外部（`outer`）变量，只能访问全局变量。

```
function getFunc() {
  let value = "test";

  let func = new Function('alert(value)');

  return func;
}

getFunc()(); // error: value is not defined
```

将其与常规行为进行比较：

```
function getFunc() {
  let value = "test";

  let func = function() { alert(value); };

  return func;
}

getFunc()(); // "test", 从 getFunc 的词法环境中获取的
```

`new Function` 的这种特性看起来有点奇怪，不过在实际中却非常实用。

想象以下我们必须通过一个字符串来创建一个函数。在编写脚本时我们不会知道该函数的代码（这也就是为什么我们不用常规方法创建函数），但在执行过程中会知道了。我们可能会从服务器或其他来源获取它。

我们的新函数需要和主脚本进行交互。

如果这个函数能够访问外部（`outer`）变量会怎么样？

问题在于，在将 JavaScript 发布到生产环境之前，需要使用 **压缩程序（minifier）** 对其进行压缩——一个特殊的程序，通过删除多余的注释和空格等压缩代码——更重要的是，将局部变量命名为较短的变量。

例如，如果一个函数有 `let userName`，压缩程序会把它替换为 `let a`（如果 `a` 已被占用了，那就使用其他字符），剩余的局部变量也会被进行类似的替换。一般来说这样的替换是安全的，毕竟这些变量是函数内的局部变量，函数外的任何东西都无法访问它。在函数内部，压缩程序会替换所有使用了这些变量的代码。压缩程序很聪明，它会分析代码的结构，而不是呆板地查找然后替换，因此它不会“破坏”你的程序。

但是在这种情况下，如果使 `new Function` 可以访问自身函数以外的变量，它也很有可能无法找到重命名的 `userName`，这是因为新函数的创建发生在代码压缩以后，变量名已经被替换了。

即使我们可以在 `new Function` 中访问外部词法环境，我们也会受挫于压缩程序。

此外，这样的代码在架构上很差并且容易出错。

当我们需要向 `new Function` 创建出的新函数传递数据时，我们必须显式地通过参数进行传递。

总结

语法：

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

由于历史原因，参数也可以按逗号分隔符的形式给出。

以下三种声明的含义相同：

```
new Function('a', 'b', 'return a + b'); // 基础语法
new Function('a,b', 'return a + b'); // 逗号分隔
new Function('a , b', 'return a + b'); // 逗号和空格分隔
```

使用 `new Function` 创建的函数，它的 `[[Environment]]` 指向全局词法环境，而不是函数所在的外部词法环境。因此，我们不能在 `new Function` 中直接使用外部变量。不过这样是好事，这有助于降低我们代码出错的可能。并且，从代码架构上讲，显式地使用参数传值是一种更好的方法，并且避免了与使用压缩程序而产生冲突的问题。

调度：`setTimeout` 和 `setInterval`

有时我们并不想立即执行一个函数，而是等待特定一段时间之后再执行。这就是所谓的“计划调用（scheduling a call）”。

目前有两种方式可以实现：

- `setTimeout` 允许我们将函数推迟到一段时间间隔之后再执行。
- `setInterval` 允许我们重复运行一个函数，从一段时间间隔之后开始运行，之后以该时间间隔连续重复运行该函数。

这两个方法并不在 **JavaScript** 的规范中。但是大多数运行环境都有内建的调度程序，并且提供了这些方法。目前来讲，所有浏览器以及 **Node.js** 都支持这两个方法。

setTimeout

语法：

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

参数说明：

func | code

想要执行的函数或代码字符串。一般传入的都是函数。由于某些历史原因，支持传入代码字符串，但是不建议这样做。

delay

执行前的延时，以毫秒为单位（1000 毫秒 = 1 秒），默认值是 0；

arg1 , arg2 ...

要传入被执行函数（或代码字符串）的参数列表（IE9 以下不支持）

例如，在下面这个示例中，`sayHi()` 方法会在 1 秒后执行：

```
function sayHi() {  
  alert('Hello');  
}  
  
setTimeout(sayHi, 1000);
```

带参数的情况：

```
function sayHi(phrase, who) {  
  alert( phrase + ', ' + who );  
}  
  
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

如果第一个参数位传入的是字符串，**JavaScript** 会自动为其创建一个函数。

所以这么写也是可以的：

```
setTimeout("alert('Hello')", 1000);
```

但是，不建议使用字符串，我们可以使用箭头函数代替它们，如下所示：

```
setTimeout(() => alert('Hello'), 1000);
```

❶ 传入一个函数，但不要执行它

新手开发者有时候会误将一对括号 `()` 加在函数后面：

```
// 错的！  
setTimeout(sayHi(), 1000);
```

这样不行，因为 `setTimeout` 期望得到一个对函数的引用。而这里的 `sayHi()` 很明显是在执行函数，所以实际上传入 `setTimeout` 的是 **函数的执行结果**。在这个例子中，`sayHi()` 的执行结果是 `undefined`（也就是说函数没有返回任何结果），所以实际上什么也没有调度。

用 `clearTimeout` 来取消调度

`setTimeout` 在调用时会返回一个“定时器标识符（timer identifier）”，在我们的例子中是 `timerId`，我们可以使用它来取消执行。

取消调度的语法：

```
let timerId = setTimeout(...);  
clearTimeout(timerId);
```

在下面的代码中，我们对一个函数进行了调度，紧接着取消了这次调度（中途反悔了）。所以最后什么也没发生：

```
let timerId = setTimeout(() => alert("never happens"), 1000);  
alert(timerId); // 定时器标识符  
  
clearTimeout(timerId);  
alert(timerId); // 还是这个标识符（并没有因为调度被取消了而变成 null）
```

从 `alert` 的输出来看，在浏览器中，定时器标识符是一个数字。在其他环境中，可能是其他的东西。例如 `Node.js` 返回的是一个定时器对象，这个对象包含一系列方法。

我再重申一遍，这些方法没有统一的规范定义，所以这没什么问题。

针对浏览器环境，定时器在 `HTML5` 的标准中有详细描述，详见 [timers section ↗](#)。

setInterval

`setInterval` 方法和 `setTimeout` 的语法相同：

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

所有参数的意义也是相同的。不过与 `setTimeout` 只执行一次不同，`setInterval` 是每间隔给定的时间周期性执行。

想要阻止后续调用，我们需要调用 `clearInterval(timerId)`。

下面的例子将每间隔 2 秒就会输出一条消息。5 秒之后，输出停止：

```
// 每 2 秒重复一次
let timerId = setInterval(() => alert('tick'), 2000);

// 5 秒之后停止
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

① `alert` 弹窗显示的时候计时器依然在进行计时

在大多数浏览器中，包括 Chrome 和 Firefox，在显示 `alert/confirm/prompt` 弹窗时，内部的定时器仍旧会继续“滴答”。

所以，在运行上面的代码时，如果在一定时间内没有关掉 `alert` 弹窗，那么在你关闭弹窗后，下一个 `alert` 会立即显示。两次 `alert` 之间的时间间隔将小于 2 秒。

嵌套的 `setTimeout`

周期性调度有两种方式。

一种是使用 `setInterval`，另外一种就是嵌套的 `setTimeout`，就像这样：

```
/** instead of:
let timerId = setInterval(() => alert('tick'), 2000);
 */

let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

上面这个 `setTimeout` 在当前这一次函数执行完时 (*) 立即调度下一次调用。

嵌套的 `setTimeout` 要比 `setInterval` 灵活得多。采用这种方式可以根据当前执行结果来调度下一次调用，因此下一次调用可以与当前这一次不同。

例如，我们要实现一个服务（server），每间隔 5 秒向服务器发送一个数据请求，但如果服务器过载了，那么就要降低请求频率，比如将间隔增加到 10、20、40 秒等。

以下是伪代码：

```

let delay = 5000;

let timerId = setTimeout(function request() {
  ...发送请求...

  if (request failed due to server overload) {
    // 下一次执行的间隔是当前的 2 倍
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);

```

并且，如果我们调度的函数占用大量的 CPU，那么我们可以测量执行所需要花费的时间，并安排下次调用是应该提前还是推迟。

嵌套的 `setTimeout` 能够精确地设置两次执行之间的延时，而 `setInterval` 却不能。

下面来比较这两个代码片段。第一个使用的是 `setInterval`：

```

let i = 1;
setInterval(function() {
  func(i++);
}, 100);

```

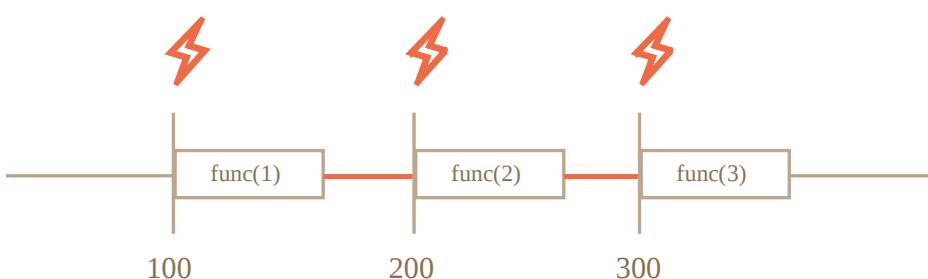
第二个使用的是嵌套的 `setTimeout`：

```

let i = 1;
setTimeout(function run() {
  func(i++);
  setTimeout(run, 100);
}, 100);

```

对 `setInterval` 而言，内部的调度程序会每间隔 100 毫秒执行一次 `func(i++)`：



注意到了吗？

使用 `setInterval` 时，`func` 函数的实际调用间隔要比代码中设定的时间间隔要短！

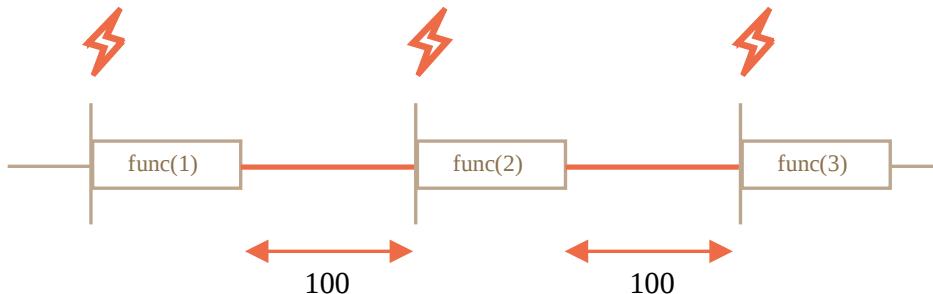
这也是正常的，因为 `func` 的执行所花费的时间“消耗”了一部分间隔时间。

也可能出现这种情况，就是 `func` 的执行所花费的时间比我们预期的时间更长，并且超出了 100 毫秒。

在这种情况下，JavaScript 引擎会等待 `func` 执行完成，然后检查调度程序，如果时间到了，则立即再次执行它。

极端情况下，如果函数每次执行时间都超过 `delay` 设置的时间，那么每次调用之间将完全没有停顿。

这是嵌套的 `setTimeout` 的示意图：



嵌套的 `setTimeout` 就能确保延时的固定（这里是 100 毫秒）。

这是因为下一次调用是在前一次调用完成时再调度的。

① 垃圾回收和 `setInterval/setTimeout` 回调（callback）

当一个函数传入 `setInterval/setTimeout` 时，将为其创建一个内部引用，并保存在调度程序中。这样，即使这个函数没有其他引用，也能防止垃圾回收器（GC）将其回收。

```
// 在调度程序调用这个函数之前，这个函数将一直存在于内存中
setTimeout(function() {...}, 100);
```

对于 `setInterval`，传入的函数也是一直存在于内存中，直到 `clearInterval` 被调用。

这里还要提到一个副作用。如果函数引用了外部变量（译注：闭包），那么只要这个函数还存在，外部变量也会随之存在。它们可能比函数本身占用更多的内存。因此，当我们不再需要调度函数时，最好取消它，即使这是个（占用内存）很小的函数。

零延时的 `setTimeout`

这儿有一种特殊的用法：`setTimeout(func, 0)`，或者仅仅是 `setTimeout(func)`。

这样调度可以让 `func` 尽快执行。但是只有在当前正在执行的脚本执行完成后，调度程序才会调用它。

也就是说，该函数被调度在当前脚本执行完成“之后”立即执行。

例如，下面这段代码会先输出“Hello”，然后立即输出“World”：

```
setTimeout(() => alert("World"));

alert("Hello");
```

第一行代码“将调用安排到日程（calendar）0 毫秒处”。但是调度程序只有在当前脚本执行完毕时才会去“检查日程”，所以先输出 “Hello”，然后才输出 “World”。

此外，还有与浏览器相关的 0 延时 `timeout` 的高级用例，我们将在 [事件循环：微任务和宏任务](#) 一章中详细讲解。

① 零延时实际上不为零（在浏览器中）

在浏览器环境下，嵌套定时器的运行频率是受限制的。根据 [HTML5 标准](#) 所讲：“经过 5 重嵌套定时器之后，时间间隔被强制设定为至少 4 毫秒”。

让我们用下面的示例来看看这到底是什么意思。其中 `setTimeout` 调用会以零延时重新调度自身的调用。每次调用都会在 `times` 数组中记录上一次调用的实际时间。那么真正的延迟是什么样的？让我们来看看：

```
let start = Date.now();
let times = [];

setTimeout(function run() {
    times.push(Date.now() - start); // 保存前一个调用的延时

    if (start + 100 < Date.now()) alert(times); // 100 毫秒之后，显示延时信息
    else setTimeout(run); // 否则重新调度
});

// 输出示例：
// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

第一次，定时器是立即执行的（正如规范里所描述的那样），接下来我们可以看到 `9, 15, 20, 24...`。两次调用之间必须经过 4 毫秒以上的强制延时。（译注：这里作者没说清楚，`timer` 数组里存放的是每次定时器运行的时刻与 `start` 的差值，所以数字只会越来越大，实际上前后调用的延时是数组值的差值。示例中前几次都是 1，所以延时为 0）

如果我们使用 `setInterval` 而不是 `setTimeout`，也会发生类似的情况：

`setInterval(f)` 会以零延时运行几次 `f`，然后以 4 毫秒以上的强制延时运行。

这个限制来自“远古时代”，并且许多脚本都依赖于此，所以这个机制也就存在至今。

对于服务端的 `JavaScript`，就没有这个限制，并且还有其他调度即时异步任务的方式。例如 `Node.js` 的 [setImmediate](#)。因此，这个提醒只是针对浏览器环境的。

总结

- `setTimeout(func, delay, ...args)` 和 `setInterval(func, delay, ...args)` 方法允许我们在 `delay` 毫秒之后运行 `func` 一次或以 `delay` 毫秒为时间间隔周期性运行 `func`。
- 要取消函数的执行，我们应该调用 `clearInterval/clearTimeout`，并将 `setInterval/setTimeout` 返回的值作为入参传入。
- 嵌套的 `setTimeout` 比 `setInterval` 用起来更加灵活，允许我们更精确地设置两次执行之间的时间。
- 零延时调度 `setTimeout(func, 0)`（与 `setTimeout(func)` 相同）用来调度需要尽快执行的调用，但是会在当前脚本执行完成后进行调用。
- 浏览器会将 `setTimeout` 或 `setInterval` 的五层或更多层嵌套调用（调用五次之后）的最小延时限制在 4ms。这是历史遗留问题。

请注意，所有的调度方法都不能 **保证** 确切的延时。

例如，浏览器内的计时器可能由于许多原因而变慢：

- CPU 过载。
- 浏览器页签处于后台模式。
- 笔记本电脑用的是电池供电（译注：使用电池供电会以降低性能为代价提升续航）。

所有这些因素，可能会将定时器的最小计时器分辨率（最小延迟）增加到 300ms 甚至 1000ms，具体以浏览器及其设置为准。

装饰者模式和转发，call/apply

JavaScript 在处理函数时提供了非凡的灵活性。它们可以被传递，用作对象，现在我们将看到如何在它们之间 **转发（forward）** 调用并 **装饰（decorate）** 它们。

透明缓存

假设我们有一个 CPU 重负载的函数 `slow(x)`，但它的结果是稳定的。换句话说，对于相同的 `x`，它总是返回相同的结果。

如果经常调用该函数，我们可能希望将结果缓存（记住）下来，以避免在重新计算上花费额外的时间。

但是我们不是将这个功能添加到 `slow()` 中，而是创建一个包装器（wrapper）函数，该函数增加了缓存功能。正如我们将要看到的，这样做有很多好处。

下面是代码和解释：

```
function slow(x) {
  // 这里可能会有重负载的 CPU 密集型工作
  alert(`Called with ${x}`);
  return x;
}

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) { // 如果缓存中有对应的结果
      return cache.get(x); // 从缓存中读取结果
    }

    let result = func(x); // 否则就调用 func

    cache.set(x, result); // 然后将结果缓存（记住）下来
    return result;
  };
}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) 被缓存下来了
alert( "Again: " + slow(1) ); // 一样的
```

```
alert( slow(2) ); // slow(2) 被缓存下来了
alert( "Again: " + slow(2) ); // 和前面一行结果相同
```

在上面的代码中，`cachingDecorator` 是一个 **装饰者（decorator）**：一个特殊的函数，它接受另一个函数并改变它的行为。

其思想是，我们可以为任何函数调用 `cachingDecorator`，它将返回缓存包装器。这很棒啊，因为我们有很多函数可以使用这样的特性，而我们需要做的就是将 `cachingDecorator` 应用于它们。

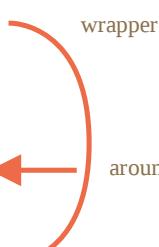
通过将缓存与主函数代码分开，我们还可以使主函数代码变得更容易。

`cachingDecorator(func)` 的结果是一个“包装器”：`function(x)` 将 `func(x)` 的调用“包装”到缓存逻辑中：

```
function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }

    let result = func(x); ← wrapper
    cache.set(x, result);
    return result;
  };
}
```



从外部代码来看，包装的 `slow` 函数执行的仍然是与之前相同的操作。它只是在其行为上添加了缓存功能。

总而言之，使用分离的 `cachingDecorator` 而不是改变 `slow` 本身的代码有几个好处：

- `cachingDecorator` 是可重用的。我们可以将它应用于另一个函数。
- 缓存逻辑是独立的，它没有增加 `slow` 本身的复杂性（如果有的话）。
- 如果需要，我们可以组合多个装饰者（其他装饰者将遵循同样的逻辑）。

使用“`func.call`”设定上下文

上面提到的缓存装饰者不适用于对象方法。

例如，在下面的代码中，`worker.slow()` 在装饰后停止工作：

```
// 我们将对 worker.slow 的结果进行缓存
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    // 可怕的 CPU 过载任务
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};
```

```
// 和之前例子中的代码相同
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x); // (**)
    cache.set(x, result);
    return result;
  };
}

alert( worker.slow(1) ); // 原始方法有效

worker.slow = cachingDecorator(worker.slow); // 现在对其进行缓存

alert( worker.slow(2) ); // 啥! Error: Cannot read property 'someMethod' of undefined
```

错误发生在试图访问 `this.someMethod` 并失败了的 (*) 行中。你能看出来为什么吗？

原因是包装器将原始函数调用为 (***) 行中的 `func(x)`。并且，当这样调用时，函数将得到 `this = undefined`。

如果尝试运行下面这段代码，我们会观察到类似的问题：

```
let func = worker.slow;
func(2);
```

因此，包装器将调用传递给原始方法，但没有上下文 `this`。因此，发生了错误。

让我们来解决这个问题。

有一个特殊的内置函数方法 `func.call(context, ...args)` ↗，它允许调用一个显式设置 `this` 的函数。

语法如下：

```
func.call(context, arg1, arg2, ...)
```

它运行 `func`，提供的第一个参数作为 `this`，后面作为参数（`arguments`）。

简单地说，这两个调用几乎相同：

```
func(1, 2, 3);
func.call(obj, 1, 2, 3)
```

它们调用的都是 `func`，参数是 `1`、`2` 和 `3`。唯一的区别是 `func.call` 还会将 `this` 设置为 `obj`。

例如，在下面的代码中，我们在不同对象的上下文中调用 `sayHi`： `sayHi.call(user)` 运行 `sayHi` 并提供了 `this=user`，然后下一行设置 `this=admin`：

```

function sayHi() {
  alert(this.name);
}

let user = { name: "John" };
let admin = { name: "Admin" };

// 使用 call 将不同的对象传递为 "this"
sayHi.call( user ); // John
sayHi.call( admin ); // Admin

```

在这里我们用带有给定上下文和 `phrase` 的 `call` 调用 `say`：

```

function say(phrase) {
  alert(this.name + ': ' + phrase);
}

let user = { name: "John" };

// user 成为 this, "Hello" 成为第一个参数
say.call( user, "Hello" ); // John: Hello

```

在我们的例子中，我们可以在包装器中使用 `call` 将上下文传递给原始函数：

```

let worker = {
  someMethod() {
    return 1;
  },
  slow(x) {
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func.call(this, x); // 现在 "this" 被正确地传递了
    cache.set(x, result);
    return result;
  };
}

worker.slow = cachingDecorator(worker.slow); // 现在对其进行缓存

alert( worker.slow(2) ); // 工作正常
alert( worker.slow(2) ); // 工作正常，没有调用原始函数（使用的缓存）

```

现在一切都正常工作了。

为了让大家理解地更清晰一些，让我们更深入地看看 `this` 是如何被传递的：

1. 在经过装饰之后，`worker.slow` 现在是包装器 `function (x) { ... }`。
2. 因此，当 `worker.slow(2)` 执行时，包装器将 `2` 作为参数，并且 `this=worker`（它是点符号 `.` 之前的对象）。
3. 在包装器内部，假设结果尚未缓存，`func.call(this, x)` 将当前的 `this` (`=worker`) 和当前的参数 (`=2`) 传递给原始方法。

使用“`func.apply`”来传递多参数

现在让我们把 `cachingDecorator` 写得更加通用。到现在为止，它只能用于单参数函数。

现在如何缓存多参数 `worker.slow` 方法呢？

```
let worker = {
  slow(min, max) {
    return min + max; // scary CPU-hogger is assumed
  }
};

// 应该记住相同参数的调用
worker.slow = cachingDecorator(worker.slow);
```

之前，对于单个参数 `x`，我们可以只使用 `cache.set(x, result)` 来保存结果，并使用 `cache.get(x)` 来检索并获取结果。但是现在，我们需要记住 `参数组合 (min, max)` 的结果。原生的 `Map` 仅将单个值作为键 (`key`)。

这儿有许多解决方案可以实现：

1. 实现一个新的（或使用第三方的）类似 `map` 的更通用并且允许多个键的数据结构。
2. 使用嵌套 `map`: `cache.set(min)` 将是一个存储（键值）对 `(max, result)` 的 `Map`。所以我们可以使用 `cache.get(min).get(max)` 来获取 `result`。
3. 将两个值合并为一个。为了灵活性，我们可以允许为装饰者提供一个“哈希函数”，该函数知道如何将多个值合并为一个值。

对于许多实际应用，第三种方式就足够了，所以我们就用这个吧。

当然，我们需要将 `func.call(this, x)` 替换成 `func.call(this, ...arguments)`，以将所有参数传递给包装的函数调用，而不仅仅是只传递第一个参数。

这是一个更强大的 `cachingDecorator`：

```
let worker = {
  slow(min, max) {
    alert(`Called with ${min}, ${max}`);
    return min + max;
  }
};

function cachingDecorator(func, hash) {
  let cache = new Map();
  return function() {
    let key = hash(arguments); // (*)
```

```

if (cache.has(key)) {
    return cache.get(key);
}

let result = func.call(this, ...arguments); // (**)

cache.set(key, result);
return result;
};

}

function hash(args) {
    return args[0] + ',' + args[1];
}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // works
alert( "Again " + worker.slow(3, 5) ); // same (cached)

```

现在这个包装器可以处理任意数量的参数了（尽管哈希函数还需要被进行调整以允许任意数量的参数。一种有趣的处理方法将在下面讲到）。

这里有两个变化：

- 在 (*) 行中它调用 `hash` 来从 `arguments` 创建一个单独的键。这里我们使用一个简单的“连接”函数，将参数 `(3, 5)` 转换为键 `"3,5"`。更复杂的情况可能需要其他哈希函数。
- 然后 (***) 行使用 `func.call(this, ...arguments)` 将包装器获得的上下文和所有参数（不仅仅是第一个参数）传递给原始函数。

我们可以使用 `func.apply(this, arguments)` 代替 `func.call(this, ...arguments)`。

内建方法 `func.apply ↗` 的语法是：

```
func.apply(context, args)
```

它运行 `func` 设置 `this=context`，并使用类数组对象 `args` 作为参数列表 (`arguments`)。

`call` 和 `apply` 之间唯一的语法区别是，`call` 期望一个参数列表，而 `apply` 期望一个包含这些参数的类数组对象。

因此，这两个调用几乎是等效的：

```

func.call(context, ...args); // 使用 spread 语法将数组作为列表传递
func.apply(context, args); // 与使用 call 相同

```

这里只有很小的区别：

- `Spread` 语法 `...` 允许将 可迭代对象 `args` 作为列表传递给 `call`。
- `apply` 仅接受 类数组对象 `args`。

因此，这些调用可以相互补充。当我们期望可迭代对象时，使用 `call`，当我们期望类数组对象时，使用 `apply`。

对于即可迭代又是类数组的对象，例如一个真正的数组，从技术上讲我们使用 `call` 或 `apply` 都行，但是 `apply` 可能会更快，因为大多数 JavaScript 引擎在内部对其进行了优化。

将所有参数连同上下文一起传递给另一个函数被称为“呼叫转移（call forwarding）”。

这是它的最简形式：

```
let wrapper = function() {
  return func.apply(this, arguments);
};
```

当外部代码调用这种包装器 `wrapper` 时，它与原始函数 `func` 的调用是无法区分的。

借用一种方法

现在，让我们对哈希函数再做一个较小的改进：

```
function hash(args) {
  return args[0] + ',' + args[1];
}
```

截至目前，它仅适用于两个参数。如果它可以适用于任何数量的 `args` 就更好了。

自然的解决方案是使用 `arr.join ↗` 方法：

```
function hash(args) {
  return args.join();
}
```

.....不幸的是，这不行。因为我们正在调用 `hash(arguments)`，`arguments` 对象既是可迭代对象又是类数组对象，但它并不是真正的数组。

所以在它上面调用 `join` 会失败，我们可以在下面看到：

```
function hash() {
  alert( arguments.join() ); // Error: arguments.join is not a function
}

hash(1, 2);
```

不过，有一种简单的方法可以使用数组的 `join` 方法：

```
function hash() {
  alert( [].join.call(arguments) ); // 1,2
}
```

```
hash(1, 2);
```

这个技巧被称为 **方法借用（method borrowing）**。

我们从常规数组 `[] .join` 中获取（借用）`join` 方法，并使用 `[] .join .call` 在 `arguments` 的上下文中运行它。

它为什么有效？

那是因为原生方法 `arr .join(glue)` 的内部算法非常简单。

从规范中几乎“按原样”解释如下：

1. 让 `glue` 成为第一个参数，如果没有参数，则使用逗号 `", "`。
2. 让 `result` 为空字符串。
3. 将 `this[0]` 附加到 `result`。
4. 附加 `glue` 和 `this[1]`。
5. 附加 `glue` 和 `this[2]`。
6.以此类推，直到 `this.length` 项目被粘在一起。
7. 返回 `result`。

因此，从技术上讲，它需要 `this` 并将 `this[0]`, `this[1]`等 `join` 在一起。它的编写方式是故意允许任何类数组的 `this` 的（不是巧合，很多方法都遵循这种做法）。这就是为什么它也可以和 `this=arguments` 一起使用。

装饰者和函数属性

通常，用装饰的函数替换一个函数或一个方法是安全的，除了一件小东西。如果原始函数有属性，例如 `func .calledCount` 或其他，则装饰后的函数将不再提供这些属性。因为这是装饰者。因此，如果有人使用它们，那么就需要小心。

例如，在上面的示例中，如果 `slow` 函数具有任何属性，而 `cachingDecorator(slow)` 则是一个没有这些属性的包装器。

一些包装器可能会提供自己的属性。例如，装饰者会计算一个函数被调用了多少次以及花费了多少时间，并通过包装器属性公开（`expose`）这些信息。

存在一种创建装饰者的方法，该装饰者可保留对函数属性的访问权限，但这需要使用特殊的 `Proxy` 对象来包装函数。我们将在后面的 `Proxy` 和 `Reflect` 中学习它。

总结

装饰者 是一个围绕改变函数行为的包装器。主要工作仍由该函数来完成。

装饰者可以被看作是可以添加到函数的“features”或“aspects”。我们可以添加一个或添加多个。而这一切都无需更改其代码！

为了实现 `cachingDecorator`，我们研究了以下方法：

- `func .call(context, arg1, arg2...)` —— 用给定的上下文和参数调用 `func`。
- `func .apply(context, args)` —— 调用 `func` 将 `context` 作为 `this` 和类数组的 `args` 传递给参数列表。

通用的 **呼叫转移** (**call forwarding**) 通常是使用 `apply` 完成的:

```
let wrapper = function() {
  return original.apply(this, arguments);
};
```

我们也可以看到一个 **方法借用** (**method borrowing**) 的例子，就是我们从一个对象中获取一个方法，并在另一个对象的上下文中“调用”它。采用数组方法并将它们应用于参数 `arguments` 是很常见的。另一种方法是使用 `Rest` 参数对象，该对象是一个真正的数组。

在 JavaScript 领域里有很多装饰者 (decorators)。通过解决本章的任务，来检查你掌握它们的程度吧。

函数绑定

当将对象方法作为回调进行传递，例如传递给 `setTimeout`，这儿会存在一个常见的问题：“丢失 `this`”。

在本章中，我们会学习如何去解决这个问题。

丢失 “`this`”

我们已经看到了丢失 `this` 的例子。一旦方法被传递到与对象分开的某个地方——`this` 就丢失。

下面是使用 `setTimeout` 时 `this` 是如何丢失的:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

正如我们所看到的，输出没有像 `this.firstName` 那样显示 “John”，而显示了 `undefined`！

这是因为 `setTimeout` 获取到了函数 `user.sayHi`，但它和对象分开了。最后一行可以被重写为:

```
let f = user.sayHi;
setTimeout(f, 1000); // 丢失了 user 上下文
```

浏览器中的 `setTimeout` 方法有些特殊：它为函数调用设定了 `this=window`（对于 Node.js，`this` 则会变为计时器 (timer) 对象，但在这儿并不重要）。所以对于 `this.firstName`，它其实试图获取的是 `window.firstName`，这个变量并不存在。在其他类似的情况下，通常 `this` 会变为 `undefined`。

这个需求很典型——我们想将一个对象方法传递到别的地方（这里——传递到调度程序），然后在该位置调用它。如何确保在正确的上下文中调用它？

解决方案 1：包装器

最简单的解决方案是使用一个包装函数：

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Hello, John!
}, 1000);
```

现在它可以正常工作了，因为它从外部词法环境中获取到了 `user`，就可以正常地调用方法了。相同的功能，但是更简短：

```
setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

看起来不错，但是我们的代码结构中出现了一个小漏洞。

如果在 `setTimeout` 触发之前（有一秒的延迟！）`user` 的值改变了怎么办？那么，突然间，它将调用错误的对象！

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(() => user.sayHi(), 1000);

// .....user 的值在不到 1 秒的时间内发生了改变
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};

// Another user in setTimeout!
```

下一个解决方案保证了这样的事情不会发生。

解决方案 2：bind

函数提供了一个内建方法 `bind ↗`，它可以绑定 `this`。

基本的语法是：

```
// 稍后将会有更复杂的语法
let boundFunc = func.bind(context);
```

`func.bind(context)` 的结果是一个特殊的类似于函数的“外来对象（exotic object）”，它可以像函数一样被调用，并且透明地（transparently）将调用传递给 `func` 并设定 `this=context`。

换句话说，`boundFunc` 调用就像绑定了 `this` 的 `func`。

举个例子，这里的 `funcUser` 将调用传递给了 `func` 同时 `this=user`：

```
let user = {
  firstName: "John"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // John
```

这里的 `func.bind(user)` 作为 `func` 的“绑定的（bound）变体”，绑定了 `this=user`。

所有的参数（arguments）都被“原样”传递给了初始的 `func`，例如：

```
let user = {
  firstName: "John"
};

function func(phrase) {
  alert(phrase + ', ' + this.firstName);
}

// 将 this 绑定到 user
let funcUser = func.bind(user);

funcUser("Hello"); // Hello, John (参数 "Hello" 被传递，并且 this=user)
```

现在我们来尝试一个对象方法：

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

let sayHi = user.sayHi.bind(user); // (*)
```

```
// 可以在没有对象（译注：与对象分离）的情况下运行它
sayHi(); // Hello, John!

setTimeout(sayHi, 1000); // Hello, John!

// 即使 user 的值在不到 1 秒内发生了改变
// sayHi 还是会使用预先绑定（pre-bound）的值
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};
```

在 (*) 行，我们取了方法 `user.sayHi` 并将其绑定到 `user`。`sayHi` 是一个“绑定后（bound）”的方法，它可以被单独调用，也可以被传递给 `setTimeout` —— 都没关系，函数上下文都会是正确的。

这里我们能够看到参数（`arguments`）都被“原样”传递了，只是 `this` 被 `bind` 绑定了：

```
let user = {
  firstName: "John",
  say(phrase) {
    alert(`#${phrase}, ${this.firstName}!`);
  }
};

let say = user.say.bind(user);

say("Hello"); // Hello, John (参数 "Hello" 被传递给了 say)
say("Bye"); // Bye, John (参数 "Bye" 被传递给了 say)
```

❶ 便捷方法：`bindAll`

如果一个对象有很多方法，并且我们都打算将它们都传递出去，那么我们可以在一个循环中完成所有方法的绑定：

```
for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}
```

JavaScript 库还提供了方便批量绑定的函数，例如 `lodash` 中的 `_.bindAll(obj)` 。

偏函数（Partial functions）

到现在位置，我们只在谈论绑定 `this`。让我们再深入一步。

我们不仅可以绑定 `this`，还可以绑定参数（`arguments`）。虽然很少这么做，但有时它可以派上用场。

`bind` 的完整语法如下：

```
let bound = func.bind(context, [arg1], [arg2], ...);
```

它允许将上下文绑定为 `this`，以及绑定函数的起始参数。

例如，我们有一个乘法函数 `mul(a, b)`：

```
function mul(a, b) {
  return a * b;
}
```

让我们使用 `bind` 在该函数基础上创建一个 `double` 函数：

```
function mul(a, b) {
  return a * b;
}

let double = mul.bind(null, 2);

alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

对 `mul.bind(null, 2)` 的调用创建了一个新函数 `double`，它将调用传递到 `mul`，将 `null` 绑定为上下文，并将 `2` 绑定为第一个参数。并且，参数（arguments）均被“原样”传递。

它被称为 [偏函数应用程序（partial function application）](#) —— 我们通过绑定先有函数的一些参数来创建一个新函数。

请注意，这里我们实际上没有用到 `this`。但是 `bind` 需要它，所以我们必须传入 `null` 之类的东西。

下面这段代码中的 `triple` 函数将值乘了三倍：

```
function mul(a, b) {
  return a * b;
}

let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
alert( triple(5) ); // = mul(3, 5) = 15
```

为什么我们通常会创建一个偏函数？

好处是我们可以创建一个具有可读性高的名字（`double`, `triple`）的独立函数。我们可以使用它，并且不必每次都提供一个参数，因为参数是被绑定了的。

另一方面，当我们有一个非常通用的函数，并希望有一个通用型更低的该函数的变体时，偏函数会非常有用。

例如，我们有一个函数 `send(from, to, text)`。然后，在一个 `user` 对象的内部，我们可能希望对它使用 `send` 的偏函数变体：从当前 `user` 发送 `sendTo(to, text)`。

在没有上下文情况下的 `partial`

当我们想绑定一些参数（`arguments`），但是这里没有上下文 `this`，应该怎么办？例如，对于一个对象方法。

原生的 `bind` 不允许这种情况。我们不可以省略上下文直接跳到参数（`arguments`）。

幸运的是，仅绑定参数（`arguments`）的函数 `partial` 比较容易实现。

像这样：

```
function partial(func, ...argsBound) {
  return function(...args) { // (*)
    return func.call(this, ...argsBound, ...args);
  }
}

// 用法:
let user = {
  firstName: "John",
  say(time, phrase) {
    alert(`[${time}] ${this.firstName}: ${phrase}!`);
  }
};

// 添加一个带有绑定时间的 partial 方法
user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

user.sayNow("Hello");
// 类似于这样的一些内容:
// [10:00] John: Hello!
```

`partial(func[, arg1, arg2...])` 调用的结果是一个包装器（`*`），它调用 `func` 并具有以下内容：

- 与它获得的函数具有相同的 `this`（对于 `user.sayNow` 调用来说，它是 `user`）
- 然后给它 `...argsBound` —— 来自于 `partial` 调用的参数（`"10:00"`）
- 然后给它 `...args` —— 给包装器的参数（`"Hello"`）

使用 `spread` 可以很容易实现这些操作，对吧？

此外，还有来自 `lodash` 库的现成的 `_partial ↗` 实现。

总结

方法 `func.bind(context, ...args)` 返回函数 `func` 的“绑定的（bound）变体”，它绑定了上下文 `this` 和第一个参数（如果给定了）。

通常我们应用 `bind` 来绑定对象方法的 `this`，这样我们就可以把它们传递到其他地方使用。例如，传递给 `setTimeout`。

当我们绑定一个现有的函数的某些参数时，绑定后的（不太通用的）函数被称为 **partially applied** 或 **partial**。

当我们不想一遍又一遍地重复相同的参数时，`partial` 非常有用。就像我们有一个 `send(from, to)` 函数，并且对于我们的任务来说，`from` 应该总是一样的，那么我们就可以搞一个 `partial` 并使用它。

深入理解箭头函数

让我们深入研究一下箭头函数。

箭头函数不仅仅是编写简洁代码的“捷径”。它还具有非常特殊且有用的特性。

JavaScript 充满了我们需要编写在其他地方执行的小函数的情况。

例如：

- `arr.forEach(func)` —— `forEach` 对每个数组元素都执行 `func`。
- `setTimeout(func)` —— `func` 由内建调度器执行。
-还有更多。

JavaScript 的精髓在于创建一个函数并将其传递到某个地方。

在这样的函数中，我们通常不想离开当前上下文。这就是箭头函数的主战场啦。

箭头函数没有 “this”

正如我们在 [对象方法, "this"](#) 一章中所学到的，箭头函数没有 `this`。如果访问 `this`，则会从外部获取。

例如，我们可以使用它在对象方法内部进行迭代：

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};

group.showList();
```

这里 `forEach` 中使用了箭头函数，所以其中的 `this.title` 其实和外部方法 `showList` 的完全一样。那就是： `group.title`。

如果我们使用正常的函数，则会出现错误：

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],
```

```
showList() {
  this.students.forEach(function(student) {
    // Error: Cannot read property 'title' of undefined
    alert(this.title + ': ' + student)
  });
}

group.showList();
```

报错是因为 `forEach` 运行它里面的这个函数，但是这个函数的 `this` 为默认值 `this=undefined`，因此就出现了尝试访问 `undefined.title` 的情况。

但箭头函数就没事，因为它们没有 `this`。

⚠ 不能对箭头函数进行 `new` 操作

不具有 `this` 自然也就意味着另一个限制：箭头函数不能用作构造器（constructor）。不能用 `new` 调用它们。

ℹ 箭头函数 VS bind

箭头函数 `=>` 和使用 `.bind(this)` 调用的常规函数之间有细微的差别：

- `.bind(this)` 创建了一个该函数的“绑定版本”。
- 箭头函数 `=>` 没有创建任何绑定。箭头函数只是没有 `this`。`this` 的查找与常规变量的搜索方式完全相同：在外部词法环境中查找。

箭头函数没有“arguments”

箭头函数也没有 `arguments` 变量。

当我们需要使用当前的 `this` 和 `arguments` 转发一个调用时，这对装饰者（decorators）来说非常有用。

例如，`defer(f, ms)` 获得了一个函数，并返回一个包装器，该包装器将调用延迟 `ms` 毫秒：

```
function defer(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms)
  };
}

function sayHi(who) {
  alert('Hello, ' + who);
}

let sayHiDeferred = defer(sayHi, 2000);
sayHiDeferred("John"); // 2 秒后显示: Hello, John
```

不用箭头函数的话，可以这么写：

```
function defer(f, ms) {
  return function(...args) {
    let ctx = this;
    setTimeout(function() {
      return f.apply(ctx, args);
    }, ms);
  };
}
```

在这里，我们必须创建额外的变量 `args` 和 `ctx`，以便 `setTimeout` 内部的函数可以获取它们。

总结

箭头函数：

- 没有 `this`
- 没有 `arguments`
- 不能使用 `new` 进行调用
- 它们也没有 `super`，但目前我们还没有学到它。我们将在 [类继承](#) 一章中学习它。

这是因为，箭头函数是针对那些没有自己的“上下文”，但在当前上下文中起作用的短代码的。并且箭头函数确实在这种使用场景中大放异彩。

对象属性配置

在本节中，我们将回到对象，并更深入地研究其属性。

属性标志和属性描述符

我们知道，对象可以存储属性。

到目前为止，属性对我们来说只是一个简单的“键值”对。但对象属性实际上是更灵活且更强大的东西。

在本章中，我们将学习其他配置选项，在下一章中，我们将学习如何将它们无形地转换为 `getter/setter` 函数。

属性标志

对象属性（`properties`），除 `value` 外，还有三个特殊的特性（`attributes`），也就是所谓的“标志”：

- `writable` — 如果为 `true`，则值可以被修改，否则它是只可读的。
- `enumerable` — 如果为 `true`，则会被在循环中列出，否则不会被列出。
- `configurable` — 如果为 `true`，则此特性可以被删除，这些属性也可以被修改，否则不可以。

我们到现在还没看到它们，是因为它们通常不会出现。当我们用“常用的方式”创建一个属性时，它们都为 `true`。但我们也可以随时更改它们。

首先，让我们来看看如何获得这些标志。

`Object.getOwnPropertyDescriptor` 方法允许查询有关属性的 **完整** 信息。

语法是：

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

obj

需要从中获取信息的对象。

propertyName

属性的名称。

返回值是一个所谓的“属性描述符”对象：它包含值和所有的标志。

例如：

```
let user = {
  name: "John"
};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2) );
/* 属性描述符:
{
  "value": "John",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
*/
```

为了修改标志，我们可以使用 `Object.defineProperty`。

语法是：

```
Object.defineProperty(obj, propertyName, descriptor)
```

obj , propertyName

要应用描述符的对象及其属性。

descriptor

要应用的属性描述符对象。

如果该属性存在，`defineProperty` 会更新其标志。否则，它会使用给定的值和标志创建属性；在这种情况下，如果没有提供标志，则会假定它是 `false`。

例如，这里创建了一个属性 `name`，该属性的所有标志都为 `false`：

```
let user = {};  
  
Object.defineProperty(user, "name", {  
    value: "John"  
});  
  
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');  
  
alert( JSON.stringify(descriptor, null, 2) );  
/*  
{  
    "value": "John",  
    "writable": false,  
    "enumerable": false,  
    "configurable": false  
}  
*/
```

将它与上面的“以常用方式创建的” `user.name` 进行比较：现在所有标志都为 `false`。如果这不是我们想要的，那么我们最好在 `descriptor` 中将它们设置为 `true`。

现在让我们通过示例来看看标志的影响。

只读

让我们通过更改 `writable` 标志来把 `user.name` 设置为只读（`user.name` 不能被重新赋值）：

```
let user = {  
    name: "John"  
};  
  
Object.defineProperty(user, "name", {  
    writable: false  
});  
  
user.name = "Pete"; // Error: Cannot assign to read only property 'name'
```

现在没有人可以改变我们 `user` 的 `name`，除非它们应用自己的 `defineProperty` 来覆盖我们的 `user` 的 `name`。

❶ 只在严格模式下会出现 Errors

在非严格模式下，在对不可写的属性等进行写入操作时，不会出现错误。但是操作仍然不会成功。在非严格模式下，违反标志的行为（flag-violating action）只会被默默地忽略掉。

这是相同的示例，但针对的是属性不存在的情况：

```
let user = {};  
  
Object.defineProperty(user, "name", {  
    value: "John",  
});
```

```
// 对于新属性，我们需要明确地列出哪些是 true
enumerable: true,
configurable: true
});

alert(user.name); // John
user.name = "Pete"; // Error
```

不可枚举

现在让我们向 `user` 添加一个自定义的 `toString`。

通常，对象的内置 `toString` 是不可枚举的，它不会显示在 `for..in` 中。但是如果我们添加我们自己的 `toString`，那么默认情况下它将显示在 `for..in` 中，如下所示：

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

// 默认情况下，我们的两个属性都会被列出：
for (let key in user) alert(key); // name, toString
```

如果我们不喜欢它，那么我们可以设置 `enumerable:false`。之后它就不会出现在 `for..in` 循环中了，就像内建的 `toString` 一样：

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

Object.defineProperty(user, "toString", {
  enumerable: false
});

// 现在我们的 toString 消失了：
for (let key in user) alert(key); // name
```

不可枚举的属性也会被 `Object.keys` 排除：

```
alert(Object.keys(user)); // name
```

不可配置

不可配置标志（`configurable:false`）有时会预设在内建对象和属性中。

不可配置的属性不能被删除。

例如，`Math.PI` 是只读的、不可枚举和不可配置的：

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/
```

因此，开发人员无法修改 `Math.PI` 的值或覆盖它。

```
Math.PI = 3; // Error

// 删除 Math.PI 也不会起作用
```

使属性变成不可配置是一条单行道。我们无法使用 `defineProperty` 把它改回去。

确切地说，不可配置性对 `defineProperty` 施加了一些限制：

1. 不能修改 `configurable` 标志。
2. 不能修改 `enumerable` 标志。
3. 不能将 `writable: false` 修改为 `true`（反之亦然）。
4. 不能修改访问者属性的 `get/set`（但是如果没有可以分配它们）。

在这里，我们将 `user.name` 设置为“永久密封”的常量：

```
let user = {};

Object.defineProperty(user, "name", {
  value: "John",
  writable: false,
  configurable: false
});

// 不能修改 user.name 或它的标志
// 下面的所有操作都不起作用：
//   user.name = "Pete"
//   delete user.name
//   defineProperty(user, "name", { value: "Pete" })
Object.defineProperty(user, "name", {writable: true}); // Error
```

ⓘ “Non-configurable” 并不意味着 “non-writable”

值得注意的例外情况：不可配置但可写的属性的值是可以被更改的。

`configurable: false` 的思想是防止更改属性标志或删除属性标志，而不是更改它的值。

Object.defineProperties

有一个方法 `Object.defineProperties(obj, descriptors)` ↗， 允许一次定义多个属性。

语法是：

```
Object.defineProperties(obj, {  
  prop1: descriptor1,  
  prop2: descriptor2  
  // ...  
});
```

例如：

```
Object.defineProperties(user, {  
  name: { value: "John", writable: false },  
  surname: { value: "Smith", writable: false },  
  // ...  
});
```

所以，我们可以一次性设置多个属性。

Object.getOwnPropertyDescriptors

要一次获取所有属性描述符，我们可以使用 `Object.getOwnPropertyDescriptors(obj)` ↗ 方法。

它与 `Object.defineProperties` 一起可以用作克隆对象的“标志感知”方式：

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

通常，当我们克隆一个对象时，我们使用赋值的方式来复制属性，像这样：

```
for (let key in user) {  
  clone[key] = user[key]  
}
```

.....但是，这并不能复制标志。所以如果我们想要一个“更好”的克隆，那么 `Object.defineProperties` 是首选。

另一个区别是 `for..in` 会忽略 `symbol` 类型的属性，但是 `Object.getOwnPropertyDescriptors` 返回包含 `symbol` 类型的属性。在内的 所有 属性描述符。

设定一个全局的密封对象

属性描述符在单个属性的级别上工作。

还有一些限制访问 整个 对象的方法：

[Object.preventExtensions\(obj\)](#)

禁止向对象添加新属性。

[Object.seal\(obj\)](#)

禁止添加/删除/修改属性。为所有现有的属性设置 `configurable: false`。

[Object.freeze\(obj\)](#)

禁止添加/删除/更改属性。为所有现有的属性设置 `configurable: false, writable: false`。

还有针对它们的测试：

[Object.isExtensible\(obj\)](#)

如果添加属性被禁止，则返回 `false`，否则返回 `true`。

[Object.isSealed\(obj\)](#)

如果添加/删除属性被禁止，并且所有现有的属性都具有 `configurable: false` 则返回 `true`。

[Object.isFrozen\(obj\)](#)

如果添加/删除/更改属性被禁止，并且所有当前属性都是 `configurable: false, writable: false`，则返回 `true`。

这些方法在实际中很少使用。

属性的 **getter** 和 **setter**

有两种类型的属性。

第一种是 **数据属性**。我们已经知道如何使用它们了。到目前为止，我们使用过的所有属性都是数据属性。

第二种类型的属性是新东西。它是 **访问器属性（accessor properties）**。它们本质上是用于获取和设置值的函数，但从外部代码来看就像常规属性。

Getter 和 setter

访问器属性由“`getter`”和“`setter`”方法表示。在对象字面量中，它们用 `get` 和 `set` 表示：

```
let obj = {
  get propName() {
    // 当读取 obj.propName 时，getter 起作用
  },
  set propName(value) {
    // 当执行 obj.propName = value 操作时，setter 起作用
  }
};
```

当读取 `obj.propName` 时，`getter` 起作用，当 `obj.propName` 被赋值时，`setter` 起作用。

例如，我们有一个具有 `name` 和 `surname` 属性的对象 `user`：

```
let user = {  
    name: "John",  
    surname: "Smith"  
};
```

现在我们想添加一个 `fullName` 属性，该属性值应该为 `"John Smith"`。当然，我们不想复制粘贴已有的信息，因此我们可以使用访问器来实现：

```
let user = {  
    name: "John",  
    surname: "Smith",  
  
    get fullName() {  
        return `${this.name} ${this.surname}`;  
    }  
};  
  
alert(user.fullName); // John Smith
```

从外表看，访问器属性看起来就像一个普通属性。这就是访问器属性的设计思想。我们不以函数的方式调用 `user.fullName`，我们正常读取它：`getter` 在幕后运行。

截至目前，`fullName` 只有一个 `getter`。如果我们尝试赋值操作 `user.fullName=`，将会出现错误：

```
let user = {  
    get fullName() {  
        return `...`;  
    }  
};  
  
user.fullName = "Test"; // Error (属性只有一个 getter)
```

让我们通过为 `user.fullName` 添加一个 `setter` 来修复它：

```
let user = {  
    name: "John",  
    surname: "Smith",  
  
    get fullName() {  
        return `${this.name} ${this.surname}`;  
    },  
  
    set fullName(value) {  
        [this.name, this.surname] = value.split(" ");  
    }  
};
```

```
// set fullName 将以给定值执行
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper
```

现在，我们就要一个“虚拟”属性。它是可读且可写的。

访问器描述符

访问器属性的描述符与数据属性的不同。

对于访问器属性，没有 `value` 和 `writable`，但是有 `get` 和 `set` 函数。

所以访问器描述符可能有：

- `get` —— 一个没有参数的函数，在读取属性时工作，
- `set` —— 带有一个参数的函数，当属性被设置时调用，
- `enumerable` —— 与数据属性的相同，
- `configurable` —— 与数据属性的相同。

例如，要使用 `defineProperty` 创建一个 `fullName` 访问器，我们可以使用 `get` 和 `set` 来传递描述符：

```
let user = {
  name: "John",
  surname: "Smith"
};

Object.defineProperty(user, 'fullName', {
  get() {
    return `${this.name} ${this.surname}`;
  },
  set(value) {
    [this.name, this.surname] = value.split(" ");
  }
});

alert(user.fullName); // John Smith

for(let key in user) alert(key); // name, surname
```

请注意，一个属性要么是访问器（具有 `get/set` 方法），要么是数据属性（具有 `value`），但不能两者都是。

如果我们试图在同一个描述符中同时提供 `get` 和 `value`，则会出现错误：

```
// Error: Invalid property descriptor.
Object.defineProperty({}, 'prop', {
  get() {
    return 1
  },
  value: 2
});
```

```
    value: 2
});
```

更聪明的 getter/setter

Getter/setter 可以用作“真实”属性值的包装器，以便对它们进行更多的控制。

例如，如果我们想禁止太短的 `user` 的 `name`，我们可以创建一个 `setter name`，并将值存储在一个单独的属性 `_name` 中：

```
let user = {
  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length < 4) {
      alert("Name is too short, need at least 4 characters");
      return;
    }
    this._name = value;
  }
};

user.name = "Pete";
alert(user.name); // Pete

user.name = ""; // Name 太短了.....
```

所以，`name` 被存储在 `_name` 属性中，并通过 `getter` 和 `setter` 进行访问。

从技术上讲，外部代码可以使用 `user._name` 直接访问 `name`。但是，这儿有一个众所周知的约定，即以下划线 "`_`" 开头的属性是内部属性，不应该从对象外部进行访问。

兼容性

访问器的一大用途是，它们允许随时通过使用 `getter` 和 `setter` 替换“正常的”数据属性，来控制和调整这些属性的行为。

想象一下，我们开始使用数据属性 `name` 和 `age` 来实现 `user` 对象：

```
function User(name, age) {
  this.name = name;
  this.age = age;
}

let john = new User("John", 25);

alert(john.age); // 25
```

.....但迟早，情况可能会发生变化。我们可能会决定存储 `birthday`，而不是 `age`，因为它更精确，更方便：

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}

let john = new User("John", new Date(1992, 6, 1));
```

现在应该如何处理仍使用 `age` 属性的旧代码呢？

我们可以尝试找到所有这些地方并修改它们，但这会花费很多时间，而且如果其他很多人都在使用该代码，那么可能很难完成所有修改。而且，`user` 中有 `age` 是一件好事，对吧？

那我们就把它保留下来吧。

为 `age` 添加一个 `getter` 来解决这个问题：

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;

  // 年龄是根据当前日期和生日计算得出的
  Object.defineProperty(this, "age", {
    get() {
      let todayYear = new Date().getFullYear();
      return todayYear - this.birthday.getFullYear();
    }
  });
}

let john = new User("John", new Date(1992, 6, 1));

alert(john.birthday); // birthday 是可访问的
alert(john.age); // .....age 也是可访问的
```

现在旧的代码也可以工作，而且我们还拥有了一个不错的附加属性。

原型，继承 原型继承

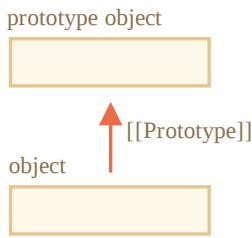
在编程中，我们经常会想获取并扩展一些东西。

例如，我们有一个 `user` 对象及其属性和方法，并希望将 `admin` 和 `guest` 作为基于 `user` 稍加修改的变体。我们想重用 `user` 中的内容，而不是复制/重新实现它的方法，而只是在其之上构建一个新的对象。

原型继承（Prototypal inheritance） 这个语言特性能够帮助我们实现这一需求。

[[Prototype]]

在 JavaScript 中，对象有一个特殊的隐藏属性 `[[Prototype]]`（如规范中所命名的），它要么为 `null`，要么就是对另一个对象的引用。该对象被称为“原型”：



原型有点“神奇”。当我们想要从 `object` 中读取一个缺失的属性时，JavaScript 会自动从原型中获取该属性。在编程中，这种行为被称为“原型继承”。许多炫酷的语言特性和编程技巧都基于此。属性 `[[Prototype]]` 是内部的而且是隐藏的，但是这儿有很多设置它的方式。

其中之一就是使用特殊的名字 `__proto__`，就像这样：

```

let animal = {
  eats: true
};

let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;

```

i `__proto__` 是 `[[Prototype]]` 的因历史原因而留下来的 `getter/setter`

请注意，`__proto__` 与 `[[Prototype]]` 不一样。`__proto__` 是 `[[Prototype]]` 的 `getter/setter`。

`__proto__` 的存在是历史的原因。在现代编程语言中，将其替换为函数

`Object.getPrototypeOf/Object.setPrototypeOf` 也能 `get/set` 原型。我们稍后将学习造成这种情况的原因以及这些函数。

根据规范，`__proto__` 必须仅在浏览器环境下才能得到支持，但实际上，包括服务端在内的所有环境都支持它。目前，由于 `__proto__` 标记在观感上更加明显，所以我们在后面的示例中将使用它。

如果我们在 `rabbit` 中查找一个缺失的属性，JavaScript 会自动从 `animal` 中获取它。

例如：

```

let animal = {
  eats: true
};

let rabbit = {
  jumps: true
};

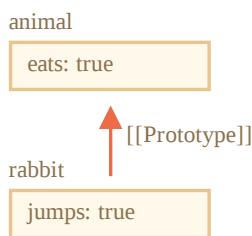
rabbit.__proto__ = animal; // (*)

// 现在这两个属性我们都能在 rabbit 中找到:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true

```

这里的 (*) 行将 `animal` 设置为 `rabbit` 的原型。

当 `alert` 试图读取 `rabbit.eats` (**) 时，因为它不存在于 `rabbit` 中，所以 JavaScript 会顺着 `[[Prototype]]` 引用，在 `animal` 中查找（自下而上）：



在这儿我们可以说 "`animal` 是 `rabbit` 的原型"，或者说 "`rabbit` 的原型是从 `animal` 继承而来的"。

因此，如果 `animal` 有许多有用的属性和方法，那么它们将自动地变为在 `rabbit` 中可用。这种属性被称为“继承”。

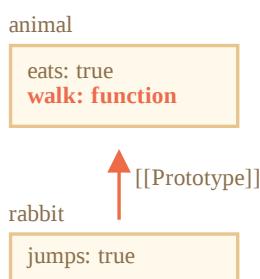
如果我们在 `animal` 中有一个方法，它可以在 `rabbit` 中被调用：

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// walk 方法是从原型中获得的
rabbit.walk(); // Animal walk
```

该方法是自动地从原型中获得的，像这样：



原型链可以很长：

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};
```

```

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk 是通过原型链获得的
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (从 rabbit)

```

animal

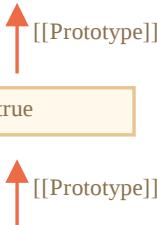
eats: true
walk: function

rabbit

jumps: true

longEar

earLength: 10



这里只有两个限制:

1. 引用不能形成闭环。如果我们试图在一个闭环中分配 `__proto__`, JavaScript 会抛出错误。
2. `__proto__` 的值可以是对象, 也可以是 `null`。而其他的类型都会被忽略。

当然, 这可能很显而易见, 但是仍然要强调: 只能有一个 `[[Prototype]]`。一个对象不能从其他两个对象获得继承。

写入不使用原型

原型仅用于读取属性。

对于写入/删除操作可以直接在对象上进行。

在下面的示例中, 我们将为 `rabbit` 分配自己的 `walk`:

```

let animal = {
  eats: true,
  walk() {
    /* rabbit 不会使用此方法 */
  }
};

let rabbit = {
  __proto__: animal
};

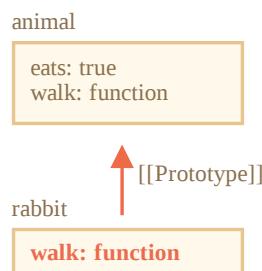
rabbit.walk = function() {

```

```
    alert("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```

从现在开始，`rabbit.walk()` 将立即在对象中找到该方法并执行，而无需使用原型：



访问器（accessor）属性是一个例外，因为分配（assignment）操作是由 `setter` 函数处理的。因此，写入此类属性实际上与调用函数相同。

也就是这个原因，所以下面这段代码中的 `admin.fullName` 能够正常运行：

```
let user = {
  name: "John",
  surname: "Smith",

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  },

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

let admin = {
  __proto__: user,
  isAdmin: true
};

alert(admin.fullName); // John Smith (*)

// setter triggers!
admin.fullName = "Alice Cooper"; // (**)
```

在 (*) 行中，属性 `admin.fullName` 在原型 `user` 中有一个 `getter`，因此它会被调用。在 (**) 行中，属性在原型中有一个 `setter`，因此它会被调用。

“this”的值

在上面的例子中可能会出现一个有趣的问题：在 `set fullName(value)` 中 `this` 的值是什么？属性 `this.name` 和 `this.surname` 被写在哪里：在 `user` 还是 `admin`？

答案很简单：`this` 根本不受原型的影响。

无论在哪里找到方法：在一个对象还是在原型中。在一个方法调用中，`this` 始终是点符号 `.` 前面的对象。

因此，setter 调用 `admin.fullName=` 使用 `admin` 作为 `this`，而不是 `user`。

这是一件非常重要的事儿，因为我们可能有一个带有很多方法的大对象，并且还有从其继承的对象。当继承的对象运行继承的方法时，它们将仅修改自己的状态，而不会修改大对象的状态。

例如，这里的 `animal` 代表“方法存储”，`rabbit` 在使用其中的方法。

调用 `rabbit.sleep()` 会在 `rabbit` 对象上设置 `this.isSleeping`：

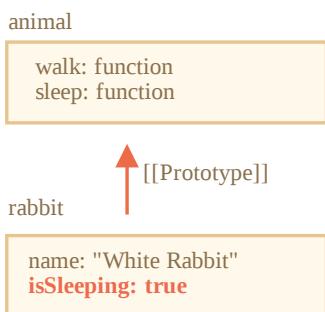
```
// animal 有一些方法
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

// 修改 rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (原型中没有此属性)
```

结果示意图：



如果我们还有从 `animal` 继承的其他对象，像 `bird` 和 `snake` 等，它们也将可以访问 `animal` 的方法。但是，每个方法调用中的 `this` 都是在调用时（点符号前）评估的对应的对象，而不是 `animal`。因此，当我们把数据写入 `this` 时，会将其存储到这些对象中。

所以，方法是共享的，但对象状态不是。

for...in 循环

`for...in` 循环也会迭代继承的属性。

例如：

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// Object.keys 只返回自己的 key
alert(Object.keys(rabbit)); // jumps

// for...in 会遍历自己以及继承的键
for(let prop in rabbit) alert(prop); // jumps, 然后是 eats
```

如果这不是我们想要的，并且我们想排除继承的属性，那么这儿有一个内建方法 `obj.hasOwnProperty(key)` ↗：如果 `obj` 具有自己的（非继承的）名为 `key` 的属性，则返回 `true`。

因此，我们可以过滤掉继承的属性（或对它们进行其他操作）：

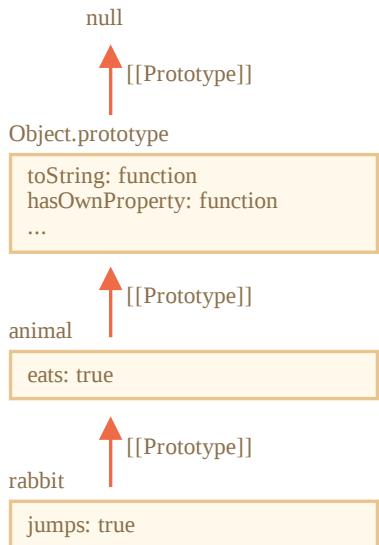
```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);

  if (isOwn) {
    alert(`Our: ${prop}`); // Our: jumps
  } else {
    alert(`Inherited: ${prop}`); // Inherited: eats
  }
}
```

这里我们有以下继承链：`rabbit` 从 `animal` 中继承，`animal` 从 `Object.prototype` 中继承（因为 `animal` 是对象字面量 `{...}`，所以这是默认的继承），然后再向上是 `null`：



注意，这有一件很有趣的事儿。方法 `rabbit.hasOwnProperty` 来自哪儿？我们并没有定义它。从上图中的原型链我们可以看到，该方法是 `Object.prototype.hasOwnProperty` 提供的。换句话说，它是继承的。

.....如果 `for..in` 循环会列出继承的属性，那为什么 `hasOwnProperty` 没有像 `eats` 和 `jumps` 那样出现在 `for..in` 循环中？

答案很简单：它是不可枚举的。就像 `Object.prototype` 的其他属性，`hasOwnProperty` 有 `enumerable:false` 标志。并且 `for..in` 只会列出可枚举的属性。这就是为什么它和其余的 `Object.prototype` 属性都未被列出。

① 几乎所有其他键/值获取方法都忽略继承的属性

几乎所有其他键/值获取方法，例如 `Object.keys` 和 `Object.values` 等，都会忽略继承的属性。

它们只会对对象自身进行操作。**不考虑** 继承自原型的属性。

总结

- 在 JavaScript 中，所有的对象都有一个隐藏的 `[[Prototype]]` 属性，它要么是另一个对象，要么就是 `null`。
- 我们可以使用 `obj.__proto__` 访问它（历史遗留下来的 `getter/setter`，这儿还有其他方法，很快我们就会讲到）。
- 通过 `[[Prototype]]` 引用的对象被称为“原型”。
- 如果我们想要读取 `obj` 的一个属性或者调用一个方法，并且它不存在，那么 JavaScript 就会尝试在原型中查找它。
- 写/删除操作直接在对象上进行，它们不使用原型（假设它是数据属性，不是 `setter`）。
- 如果我们调用 `obj.method()`，而且 `method` 是从原型中获取的，`this` 仍然会引用 `obj`。因此，方法始终与当前对象一起使用，即使方法是继承的。
- `for..in` 循环在其自身和继承的属性上进行迭代。所有其他的键/值获取方法仅对对象本身起作用。

F.prototype

我们还记得，可以使用诸如 `new F()` 这样的构造函数来创建一个新对象。

如果 `F.prototype` 是一个对象，那么 `new` 操作符会使用它为新对象设置 `[[Prototype]]`。

i **请注意：**

JavaScript 从一开始就有了原型继承。这是 JavaScript 编程语言的核心特性之一。

但是在过去，没有直接对其进行访问的方式。唯一可靠的方法是本章中会介绍的构造函数的 `"prototype"` 属性。目前仍有许多脚本仍在使用它。

请注意，这里的 `F.prototype` 指的是 `F` 的一个名为 `"prototype"` 的常规属性。这听起来与“原型”这个术语很类似，但这里我们实际上指的是具有该名字的常规属性。

下面是一个例子：

```
let animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
}

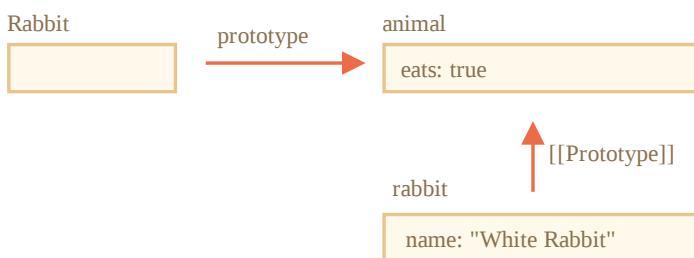
Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); // rabbit.__proto__ == animal

alert( rabbit.eats ); // true
```

设置 `Rabbit.prototype = animal` 的字面意思是：“当创建了一个 `new Rabbit` 时，把它的 `[[Prototype]]` 赋值为 `animal`”。

这是结果示意图：



在上图中，`"prototype"` 是一个水平箭头，表示一个常规属性，`[[Prototype]]` 是垂直的，表示 `rabbit` 继承自 `animal`。

i **`F.prototype` 仅用在 `new F` 时**

`F.prototype` 属性仅在 `new F` 被调用时使用，它为新对象的 `[[Prototype]]` 赋值。

如果在创建之后，`F.prototype` 属性有了变化 (`F.prototype = <another object>`)，那么通过 `new F` 创建的新对象也将随之拥有新的对象作为 `[[Prototype]]`，但已经存在的对象将保持旧有的值。

默认的 `F.prototype`, 构造器属性

每个函数都有 `"prototype"` 属性, 即使我们没有提供它。

默认的 `"prototype"` 是一个只有属性 `constructor` 的对象, 属性 `constructor` 指向函数自身。

像这样:

```
function Rabbit() {}

/* default prototype
Rabbit.prototype = { constructor: Rabbit };
*/
```



我们可以检查一下:

```
function Rabbit() {}
// by default:
// Rabbit.prototype = { constructor: Rabbit }

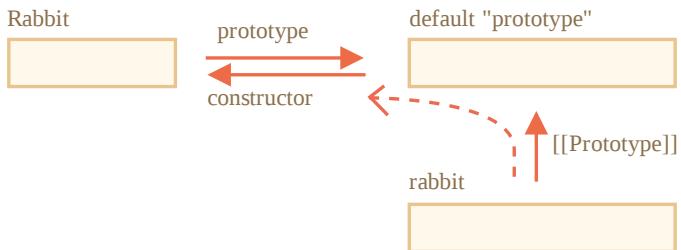
alert( Rabbit.prototype.constructor === Rabbit ); // true
```

通常, 如果我们什么都不做, `constructor` 属性可以通过 `[[Prototype]]` 给所有 `rabbits` 使用:

```
function Rabbit() {}
// by default:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // inherits from {constructor: Rabbit}

alert(rabbit.constructor === Rabbit); // true (from prototype)
```



我们可以使用 `constructor` 属性来创建一个新对象, 该对象使用与现有对象相同的构造器。像这样:

```
function Rabbit(name) {
  this.name = name;
  alert(name);
}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");
```

当我们有一个对象，但不知道它使用了哪个构造器（例如它来自第三方库），并且我们需要创建另一个类似的对象时，用这种方法就很方便。

但是，关于 "constructor" 最重要的是.....

.....JavaScript 自身并不能确保正确的 "constructor" 函数值。

是的，它存在于函数的默认 "prototype" 中，但仅此而已。之后会发生什么——完全取决于我们。

特别是，如果我们将整个默认 prototype 替换掉，那么其中就不会有 "constructor" 了。

例如：

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false
```

因此，为了确保正确的 "constructor"，我们可以选择添加/删除属性到默认 "prototype"，而不是将其整个覆盖：

```
function Rabbit() {}

// 不要将 Rabbit.prototype 整个覆盖
// 可以向其中添加内容
Rabbit.prototype.jumps = true
// 默认的 Rabbit.prototype.constructor 被保留了下来
```

或者，也可以手动重新创建 constructor 属性：

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};

// 这样的 constructor 也是正确的，因为我们手动添加了它
```

总结

在本章中，我们简要介绍了为通过构造函数创建的对象设置 `[[Prototype]]` 的方法。稍后我们将看到更多依赖于此的高级编程模式。

一切都很简单，只需要记住几条重点就可以清晰地掌握了：

- `F.prototype` 属性（不要把它与 `[[Prototype]]` 弄混了）在 `new F` 被调用时为新对象的 `[[Prototype]]` 赋值。
- `F.prototype` 的值要么是一个对象，要么就是 `null`：其他值都不起作用。
- `"prototype"` 属性仅在设置了一个构造函数（constructor function），并通过 `new` 调用时，才具有这种特殊的影响。

在常规对象上，`prototype` 没什么特别的：

```
let user = {
  name: "John",
  prototype: "Bla-bla" // 这里没有魔法了
};
```

默认情况下，所有函数都有 `F.prototype = {constructor: F}`，所以我们可以通过访问它的 `"constructor"` 属性来获取一个对象的构造器。

原生的原型

`"prototype"` 属性在 JavaScript 自身的核心部分中被广泛地应用。所有的内置构造函数都用到了它。

首先，我们将看看原生原型的详细信息，然后学习如何使用它为内建对象添加新功能。

Object.prototype

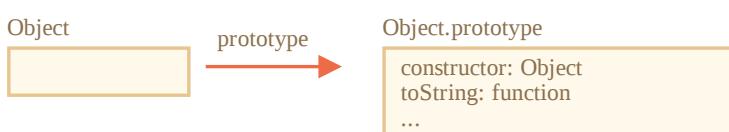
假如我们输出一个空对象：

```
let obj = {};
alert( obj ); // "[object Object]" ?
```

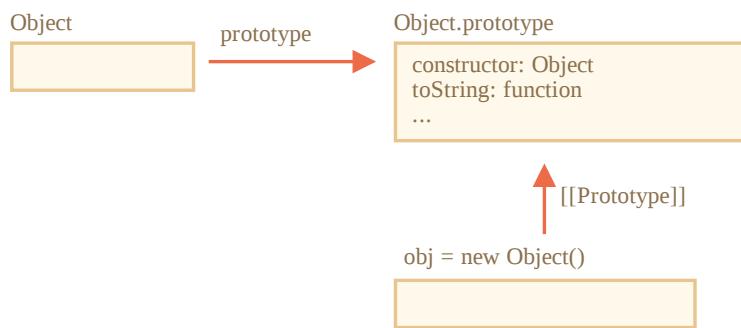
生成字符串 `"[object Object]"` 的代码在哪里？那就是一个内建的 `toString` 方法，但是它在哪里呢？`obj` 是空的！

.....然而简短的表达式 `obj = {}` 和 `obj = new Object()` 是一个意思，其中 `Object` 就是一个内建的对象构造函数，其自身的 `prototype` 指向一个带有 `toString` 和其他方法的一个巨大的对象。

就像这样：



当 `new Object()` 被调用（或一个字面量对象 `{...}` 被创建），按照前面章节中我们学习过的规则，这个对象的 `[[Prototype]]` 属性被设置为 `Object.prototype`：



所以，之后当 `obj.toString()` 被调用时，这个方法是从 `Object.prototype` 中获取的。我们可以这样验证它：

```
let obj = {};  
  
alert(obj.__proto__ === Object.prototype); // true  
// obj.toString === obj.__proto__.toString == Object.prototype.toString
```

请注意在 `Object.prototype` 上方的链中没有更多的 `[[Prototype]]`：

```
alert(Object.prototype.__proto__); // null
```

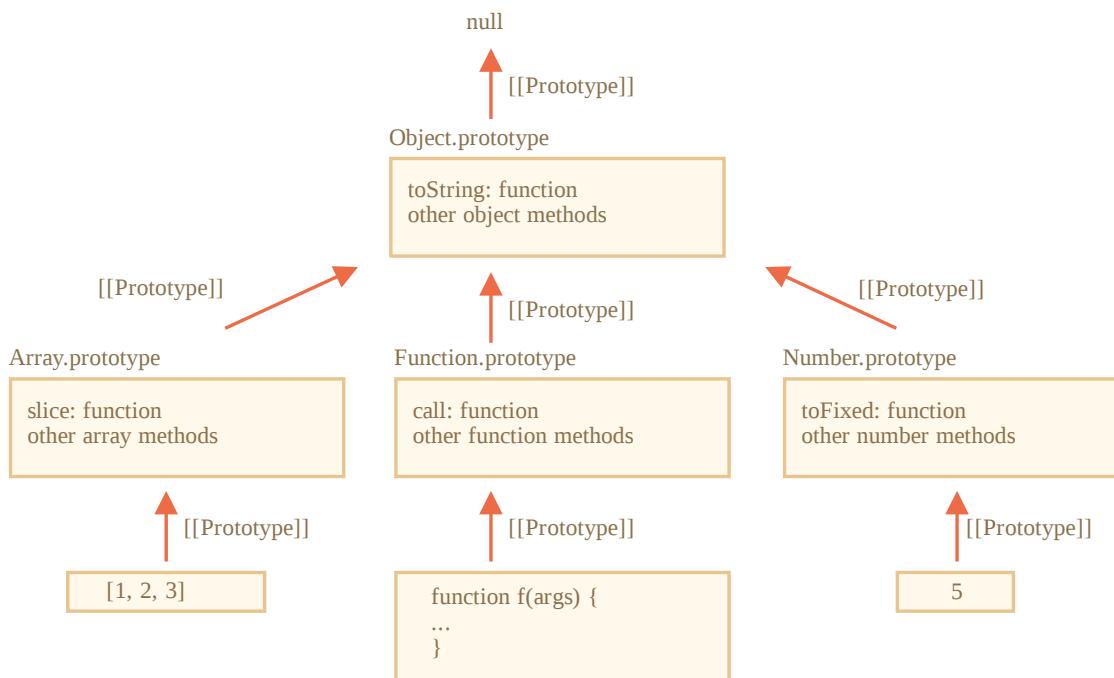
其他内建原型

其他内建对象，像 `Array`、`Date`、`Function` 及其他，都在 `prototype` 上挂载了方法。

例如，当我们创建一个数组 `[1, 2, 3]`，在内部会默认使用 `new Array()` 构造器。因此 `Array.prototype` 变成了这个数组的 `prototype`，并为这个数组提供数组的操作方法。这样内存的存储效率是很高的。

按照规范，所有的内建原型顶端都是 `Object.prototype`。这就是为什么有人说“一切都从对象继承而来”。

下面是完整的示意图（3个内建对象）：



让我们手动验证原型:

```

let arr = [1, 2, 3];

// 它继承自 Array.prototype?
alert( arr.__proto__ === Array.prototype ); // true

// 接下来继承自 Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

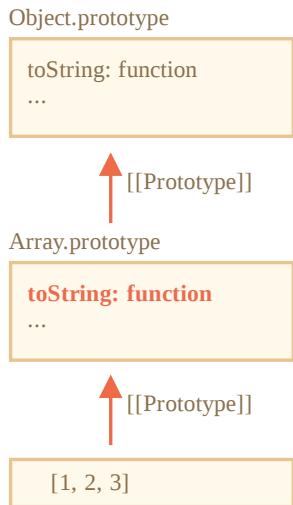
// 原型链的顶端为 null。
alert( arr.__proto__.__proto__.__proto__ ); // null
    
```

一些方法在原型上可能会发生重叠，例如，`Array.prototype` 有自己的 `toString` 方法来列举出来数组的所有元素并用逗号分隔每一个元素。

```

let arr = [1, 2, 3]
alert(arr); // 1,2,3 <-- Array.prototype.toString 的结果
    
```

正如我们之前看到的那样，`Object.prototype` 也有 `toString` 方法，但是 `Array.prototype` 在原型链上更近，所以数组对象原型上的方法会被使用。



浏览器内的工具，像 Chrome 开发者控制台也会显示继承性（可能需要对内置对象使用 `console.dir`）：

```

> console.dir([1,2,3])
▼ Array[3] ⓘ
  0: 1
  1: 2
  2: 3
  length: 3
  ▼ __proto__:=Array.prototype
    ► concat: function concat() { [native code] }
    ► ...
    ► unshift: function unshift() { [native code] }
    ▼ __proto__:=Object.prototype
      ► ...
      ► constructor: function Object() { [native code] }
      ► hasOwnProperty: function hasOwnProperty() { [native code] }
      ► isPrototypeOf: function isPrototypeOf() { [native code] }
      ► ...
    
```

其他内建对象也以同样的方式运行。即使是函数——它们是内建构造器 `Function` 的对象，并且它们的方法（`call/apply` 及其他）都取自 `Function.prototype`。函数也有自己的 `toString` 方法。

```

function f() {}

alert(f.__proto__ == Function.prototype); // true
alert(f.__proto__.__proto__ == Object.prototype); // true, inherit from objects
  
```

基本数据类型

最复杂的事情发生在字符串、数字和布尔值上。

正如我们记忆中的那样，它们并不是对象。但是如果我们试图访问它们的属性，那么临时包装器对象将会通过内建的构造器 `String`、`Number` 和 `Boolean` 被创建。它们提供给我们操作字符串、数字和布尔值的方法然后消失。

这些对象对我们来说是无形地创建出来的。大多数引擎都会对其进行优化，但是规范中描述的就是通过这种方式。这些对象的方法也驻留在它们的 `prototype` 中，可以通过 `String.prototype`、`Number.prototype` 和 `Boolean.prototype` 进行获取。

⚠ 值 `null` 和 `undefined` 没有对象包装器

特殊值 `null` 和 `undefined` 比较特殊。它们没有对象包装器，所以它们没有方法和属性。并且它们也没有相应的原型。

更改原生原型

原生的原型是可以被修改的。例如，我们向 `String.prototype` 中添加一个方法，这个方法将对所有的字符串都是可用的：

```
String.prototype.show = function() {
  alert(this);
};

"BOOM!".show(); // BOOM!
```

在开发的过程中，我们可能会想要一些新的内建方法，并且想把它们添加到原生原型中。但这通常是一个很不好的想法。

⚠ 重要:

原型是全局的，所以很容易造成冲突。如果有两个库都添加了 `String.prototype.show` 方法，那么其中的一个方法将被另一个覆盖。

所以，通常来说，修改原生原型被认为是一个很不好的想法。

在现代编程中，只有一种情况下允许修改原生原型。那就是 **polyfilling**。

Polyfilling 是一个术语，表示某个方法在 JavaScript 规范中已存在，但是特定的 JavaScript 引擎尚不支持该方法，那么我们可以通过手动实现它，并用以填充内建原型。

例如：

```
if (!String.prototype.repeat) { // 如果这儿没有这个方法
  // 那就在 prototype 中添加它

  String.prototype.repeat = function(n) {
    // 重复传入的字符串 n 次

    // 实际上，实现代码比这个要复杂一些（完整的方法可以在规范中找到）
    // 但即使是不够完美的 polyfill 也常常被认为是足够的
    return new Array(n + 1).join(this);
  };
}

alert( "La".repeat(3) ); // LaLaLa
```

从原型中借用

在 [装饰者模式和转发，call/apply](#) 一章中，我们讨论了方法借用。

那是指我们从一个对象获取一个方法，并将其复制到另一个对象。

一些原生原型的方法通常会被借用。

例如，如果我们要创建类数组对象，则可能需要向其中复制一些 `Array` 方法。

例如：

```
let obj = {  
  0: "Hello",  
  1: "world!",  
  length: 2,  
};  
  
obj.join = Array.prototype.join;  
  
alert( obj.join(' ') ); // Hello,world!
```

上面这段代码有效，是因为内建的方法 `join` 的内部算法只关心正确的索引和 `length` 属性。它不会检查这个对象是否是真正的数组。许多内建方法就是这样。

另一种方式是通过将 `obj.__proto__` 设置为 `Array.prototype`，这样 `Array` 中的所有方法都自动地可以在 `obj` 中使用了。

但是如果 `obj` 已经从另一个对象进行了继承，那么这种方法就不可行了（译注：因为这样会覆盖掉已有的继承。此处 `obj` 其实已经从 `Object` 进行了继承，但是 `Array` 也继承自 `Object`，所以此处的方法借用不会影响 `obj` 对原有继承的继承，因为 `obj` 通过原型链依旧继承了 `Object`）。请记住，我们一次只能继承一个对象。

方法借用很灵活，它允许在需要时混合来自不同对象的方法。

总结

- 所有的内建对象都遵循相同的模式（pattern）：
 - 方法都存储在 `prototype` 中（`Array.prototype`、`Object.prototype`、`Date.prototype` 等）。
 - 对象本身只存储数据（数组元素、对象属性、日期）。
- 原始数据类型也将方法存储在包装器对象的 `prototype` 中：`Number.prototype`、`String.prototype` 和 `Boolean.prototype`。只有 `undefined` 和 `null` 没有包装器对象。
- 内建原型可以被修改或被用新的方法填充。但是不建议更改它们。唯一允许的情况可能是，当我们添加一个还没有被 JavaScript 引擎支持，但已经被加入 JavaScript 规范的新标准时，才可能允许这样做。

原型方法，没有 `__proto__` 的对象

在这部分内容的第一章中，我们提到了设置原型的现代方法。

`__proto__` 被认为是过时且不推荐使用的（deprecated），这里的不推荐使用是指 JavaScript 规范中规定，`proto` 必须仅在浏览器环境下才能得到支持。

现代的方法有：

- `Object.create(proto[, descriptors])` —— 利用给定的 `proto` 作为 `[[Prototype]]` 和可选的属性描述来创建一个空对象。

- `Object.getPrototypeOf(obj)` —— 返回对象 `obj` 的 `[[Prototype]]`。
- `Object.setPrototypeOf(obj, proto)` —— 将对象 `obj` 的 `[[Prototype]]` 设置为 `proto`。

应该使用这些方法来代替 `__proto__`。

例如：

```
let animal = {
  eats: true
};

// 创建一个以 animal 为原型的新对象
let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal); // true

Object.setPrototypeOf(rabbit, {}); // 将 rabbit 的原型修改为 {}
```

`Object.create` 有一个可选的第二参数：属性描述器。我们可以在此处为新对象提供额外的属性，就像这样：

```
let animal = {
  eats: true
};

let rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});

alert(rabbit.jumps); // true
```

描述器的格式与 [属性标志和属性描述符](#) 一章中所讲的一样。

我们可以使用 `Object.create` 来实现比复制 `for..in` 循环中的属性更强大的对象克隆方式：

```
// 完全相同的对象 obj 的浅拷贝
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

此调用可以对 `obj` 进行真正准确地拷贝，包括所有的属性：可枚举和不可枚举的，数据属性和 `setters/getters` —— 包括所有内容，并带有正确的 `[[Prototype]]`。

原型简史

如果我们数一下有多少种处理 `[[Prototype]]` 的方式，答案是有很多！很多种方法做的都是同一件事儿！

为什么会出现这种情况？

这是历史原因。

- 构造函数的 `"prototype"` 属性自古以来就起作用。
- 之后，在 2012 年，`Object.create` 出现在标准中。它提供了使用给定原型创建对象的能力。但没有提供 `get/set` 它的能力。因此，许多浏览器厂商实现了非标准的 `__proto__` 访问器，该访问器允许用户随时 `get/set` 原型。
- 之后，在 2015 年，`Object.setPrototypeOf` 和 `Object.getPrototypeOf` 被加入到标准中，执行与 `__proto__` 相同的功能。由于 `__proto__` 实际上已经在所有地方都得到了实现，但它已过时，所以被加入到该标准的附件 B 中，即：在非浏览器环境下，它的支持是可选的。

目前为止，我们拥有了所有这些方式。

为什么将 `__proto__` 替换成函数 `getPrototypeOf/setPrototypeOf`？这是一个有趣的问题，需要我们理解为什么 `__proto__` 不好。继续阅读，你就会知道答案。

⚠ 如果速度很重要，就请不要修改已存在的对象的 `[[Prototype]]`

从技术上来讲，我们可以在任何时候 `get/set [[Prototype]]`。但是通常我们只在创建对象的时候设置它一次，自那之后不再修改：`rabbit` 继承自 `animal`，之后不再更改。

并且，JavaScript 引擎对此进行了高度优化。用 `Object.setPrototypeOf` 或 `obj.__proto__ =` “即时”更改原型是一个非常缓慢的操作，因为它破坏了对象属性访问操作的内部优化。因此，除非你知道自己在做什么，或者 JavaScript 的执行速度对你来说完全不重要，否则请避免使用它。

"Very plain" objects

我们知道，对象可以用作关联数组（associative arrays）来存储键/值对。

.....但是如果我们尝试在其中存储 **用户提供的** 键（例如：一个用户输入的字典），我们可以发现一个有趣的小故障：所有的键都正常工作，除了 `"__proto__"`。

看一下这个例子：

```
let obj = {};  
  
let key = prompt("What's the key?", "__proto__");  
obj[key] = "some value";  
  
alert(obj[key]); // [object Object], 并不是 "some value"!
```

这里如果用户输入 `__proto__`，那么赋值会被忽略！

我们不应该对此感到惊讶。`__proto__` 属性很特别：它必须是对象或者 `null`。字符串不能成为 `prototype`。

但是我们不是 **打算** 实现这种行为，对吗？我们想要存储键值对，然而键名为 `"__proto__"` 的键值对没有被正确存储。所以这是一个 bug。

在这里，后果并没有很严重。但是在其他情况下，我们可能会对对象进行赋值操作，然后原型可能就被更改了。结果，可能会导致完全意想不到的结果。

最可怕的是——通常开发者完全不会考虑到这一点。这让此类 bug 很难被发现，甚至变成漏洞，尤其是在 JavaScript 被用在服务端的时候。

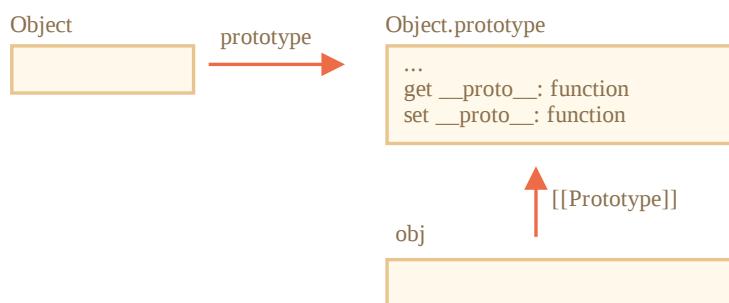
为默认情况下为函数的 `toString` 以及其他内建方法执行赋值操作，也会出现意想不到的结果。

我们怎么避免这样的问题呢？

首先，我们可以改用 `Map`，那么一切都迎刃而解。

但是 `Object` 在这里同样可以运行得很好，因为 JavaScript 语言的制造者很早就注意到了这个问题。

`__proto__` 不是一个对象的属性，只是 `Object.prototype` 的访问器属性：



因此，如果 `obj.__proto__` 被读取或者赋值，那么对应的 `getter/setter` 会被从它的原型中调用，它会 `set/get [[Prototype]]`。

就像在本部分教程的开头所说的那样：`__proto__` 是一种访问 `[[Prototype]]` 的方式，而不是 `[[prototype]]` 本身。

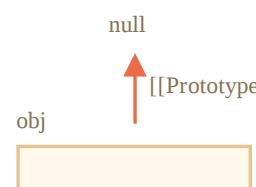
现在，我们想要将一个对象用作关联数组，我们可以使用一些小技巧：

```
let obj = Object.create(null);

let key = prompt("what's the key?", "__proto__");
obj[key] = "some value";

alert(obj[key]); // "some value"
```

`Object.create(null)` 创建了一个空对象，这个对象没有原型（`[[Prototype]]` 是 `null`）：



因此，它没有继承 `__proto__` 的 `getter/setter` 方法。现在，它被作为正常的数据属性进行处理，因此上面的这个示例能够正常工作。

我们可以把这样的对象称为“`very plain`”或“`pure dictionary`”对象，因为它们甚至比通常的对象（`plain object`）`{...}` 还要简单。

缺点是这样的对象没有任何内建的对象的方法，例如 `toString`：

```
let obj = Object.create(null);  
  
alert(obj); // Error (no toString)
```

.....但是它们通常对关联数组而言还是很友好。

请注意，大多数与对象相关的方法都是 `Object.something(...)`，例如 `Object.keys(obj)` —— 它们不在 `prototype` 中，因此在“very plain”对象中它们还是可以继续使用：

```
let chineseDictionary = Object.create(null);  
chineseDictionary.hello = "你好";  
chineseDictionary.bye = "再见";  
  
alert(Object.keys(chineseDictionary)); // hello, bye
```

总结

设置和直接访问原型的现代方法有：

- `Object.create(proto[, descriptors])` —— 利用给定的 `proto` 作为 `[[Prototype]]`（可以是 `null`）和可选的属性描述来创建一个空对象。
- `Object.getPrototypeOf(obj)` —— 返回对象 `obj` 的 `[[Prototype]]`（与 `__proto__` 的 getter 相同）。
- `Object.setPrototypeOf(obj, proto)` —— 将对象 `obj` 的 `[[Prototype]]` 设置为 `proto`（与 `__proto__` 的 setter 相同）。

如果要将一个用户生成的键放入一个对象，那么内建的 `__proto__` getter/setter 是不安全的。因为用户可能会输入 `"__proto__"` 作为键，这会导致一个 `error`，虽然我们希望这个问题不会造成什么大影响，但通常会造成不可预料的后果。

因此，我们可以使用 `Object.create(null)` 创建一个没有 `__proto__` 的“very plain”对象，或者对此类场景坚持使用 `Map` 对象就可以了。

此外，`Object.create` 提供了一种简单的方式来浅拷贝一个对象的所有描述符：

```
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

此外，我们还明确了 `__proto__` 是 `[[Prototype]]` 的 getter/setter，就像其他方法一样，它位于 `Object.prototype`。

我们可以通过 `Object.create(null)` 来创建没有原型的对象。这样的对象被用作“pure dictionaries”，对于它们而言，使用 `"__proto__"` 作为键是没有问题的。

其他方法：

- `Object.keys(obj)` / `Object.values(obj)` / `Object.entries(obj)` —— 返回一个可枚举的由自身的字符串属性名/值/键值对组成的数组。

- `Object.getOwnPropertySymbols(obj)` —— 返回一个由自身所有的 `symbol` 类型的键组成的数组。
- `Object.getOwnPropertyNames(obj)` —— 返回一个由自身所有的字符串键组成的数组。
- `Reflect.ownKeys(obj)` —— 返回一个由自身所有键组成的数组。
- `obj.hasOwnProperty(key)`: 如果 `obj` 拥有名为 `key` 的自身的属性（非继承而来的），则返回 `true`。

所有返回对象属性的方法（如 `Object.keys` 及其他）——都返回“自身”的属性。如果我们想继承它们，我们可以使用 `for...in`。

类

Class 基本语法

在面向对象的编程中，`class` 是用于创建对象的可扩展的程序代码模版，它为对象提供了状态（成员变量）的初始值和行为（成员函数或方法）的实现。

“ Wikipedia”

在日常开发中，我们经常需要创建许多相同类型的对象，例如用户（`users`）、商品（`goods`）或者其他任何其他东西。

正如我们在 [构造器和操作符 "new"](#) 一章中已经学到的，`new function` 可以帮助我们实现这种需求。

但在现代 JavaScript 中，还有一个更高级的“类（`class`）”构造方式，它引入许多非常棒的新功能，这些功能对于面向对象编程很有用。

“class” 语法

基本语法是：

```
class MyClass {
  // class 方法
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

然后使用 `new MyClass()` 来创建具有上述列出的所有方法的新对象。

`new` 会自动调用 `constructor()` 方法，因此我们可以在 `constructor()` 中初始化对象。

例如：

```
class User {
  constructor(name) {
    this.name = name;
  }
}
```

```
}

sayHi() {
  alert(this.name);
}

}

// 用法:
let user = new User("John");
user.sayHi();
```

当 `new User("John")` 被调用:

1. 一个新对象被创建。
2. `constructor` 使用给定的参数运行，并为其分配 `this.name`。

.....然后我们就可以调用对象方法了，例如 `user.sayHi`。

⚠ 类的方法之间没有逗号

对于新手开发人员来说，常见的陷阱是在类的方法之间放置逗号，这会导致语法错误。

不要把这里的符号与对象字面量相混淆。在类中，不需要逗号。

什么是 `class`?

所以，`class` 到底是什么？正如人们可能认为的那样，这不是一个全新的语言级实体。

让我们揭开其神秘面纱，看看类究竟是什么。这将有助于我们理解许多复杂的方面。

在 JavaScript 中，类是一种函数。

看看下面这段代码:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

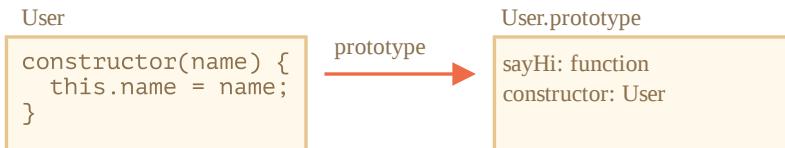
// 佐证: User 是一个函数
alert(typeof User); // function
```

`class User { ... }` 构造实际上做了如下的事儿:

1. 创建一个名为 `User` 的函数，该函数成为类声明的结果。该函数的代码来自于 `constructor` 方法（如果我们不编写这种方法，那么它就被假定为空）。
2. 存储类中的方法，例如 `User.prototype` 中的 `sayHi`。

当 `new User` 对象被创建后，当我们调用其方法时，它会从原型中获取对应的方法，正如我们在 `F.prototype` 一章中所讲的那样。因此，对象 `new User` 可以访问类中的方法。

我们可以将 `class User` 声明的结果解释为:



下面这些代码很好地解释了它们：

```

class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// class 是函数 function
alert(typeof User); // function

// ...或者, 更确切地说, 是构造器方法
alert(User === User.prototype.constructor); // true

// 方法在 User.prototype 中, 例如:
alert(User.prototype.sayHi); // alert(this.name);

// 在原型中实际上有两个方法
alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi

```

不仅仅是语法糖

人们常说 `class` 是一个语法糖（旨在使内容更易阅读，但不引入任何新内容的语法），因为我们实际上可以在没有 `class` 的情况下声明相同的内容：

```

// 用纯函数重写 class User

// 1. 创建构造器函数
function User(name) {
  this.name = name;
}

// 任何函数原型默认都具有构造器属性,
// 所以, 我们不需要创建它

// 2. 将方法添加到原型
User.prototype.sayHi = function() {
  alert(this.name);
};

// 用法:
let user = new User("John");
user.sayHi();

```

这个定义的结果与使用类得到的结果基本相同。因此，这确实是将 `class` 视为一种定义构造器及其原型方法的语法糖的理由。

尽管，它们之间存在着重大差异：

- 首先，通过 `class` 创建的函数具有特殊的内部属性标记 `[[FunctionKind]]: "classConstructor"`。因此，它与手动创建并不完全相同。

不像普通函数，调用类构造器时必须要用 `new` 关键词：

```
class User {  
    constructor() {}  
}  
  
alert(typeof User); // function  
User(); // Error: Class constructor User cannot be invoked without 'new'
```

此外，大多数 JavaScript 引擎中的类构造器的字符串表示形式都以 “`class...`” 开头

```
class User {  
    constructor() {}  
}  
  
alert(User); // class User { ... }
```

2. 类方法不可枚举。类定义将 `"prototype"` 中的所有方法的 `enumerable` 标志设置为 `false`。

这很好，因为如果我们对一个对象调用 `for..in` 方法，我们通常不希望 `class` 方法出现。

3. 类总是使用 `use strict`。在类构造中的所有代码都将自动进入严格模式。

此外，`class` 语法还带来了许多其他功能，我们稍后将会探索它们。

类表达式

就像函数一样，类可以在另外一个表达式中被定义，被传递，被返回，被赋值等。

这是一个类表达式的例子：

```
let User = class {  
    sayHi() {  
        alert("Hello");  
    }  
};
```

类似于命名函数表达式（`Named Function Expressions`），类表达式可能也应该有一个名字。

如果类表达式有名字，那么该名字仅在类内部可见：

```
// "命名类表达式 (Named Class Expression)"  
// (规范中没有这样的术语，但是它和命名函数表达式类似)  
let User = class MyClass {  
    sayHi() {  
        alert(MyClass); // MyClass 这个名字仅在类内部可见  
    }  
};  
  
new User().sayHi(); // 正常运行，显示 MyClass 中定义的内容
```

```
alert(MyClass); // error, MyClass 在外部不可见
```

我们甚至可以动态地“按需”创建类，就像这样：

```
function makeClass(phrase) {
  // 声明一个类并返回它
  return class {
    sayHi() {
      alert(phrase);
    }
  }
}

// 创建一个新的类
let User = makeClass("Hello");

new User().sayHi(); // Hello
```

Getters/setters 及其他速记

就像对象字面量，类可能包括 getters/setters，计算属性（computed properties）等。

这是一个使用 `get/set` 实现 `user.name` 的示例：

```
class User {

  constructor(name) {
    // 调用 setter
    this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    if (value.length < 4) {
      alert("Name is too short.");
      return;
    }
    this._name = value;
  }
}

let user = new User("John");
alert(user.name); // John

user = new User(""); // Name is too short.
```

类声明在 `User.prototype` 中创建 getters 和 setters，就像这样：

```
Object.defineProperties(User.prototype, {
  name: {
    get() {
      return this._name
    },
    set(name) {
      // ...
    }
  }
});
```

这是一个 `[...]` 中有计算属性名称（computed property name）的例子：

```
class User {

  ['say' + 'Hi']() {
    alert("Hello");
  }

}

new User().sayHi();
```

Class 字段



旧的浏览器可能需要 polyfill

类字段（field）是最近才添加到语言中的。

之前，类仅具有方法。

“类字段”是一种允许添加任何属性的语法。

例如，让我们在 `class User` 中添加一个 `name` 属性：

```
class User {
  name = "Anonymous";

  sayHi() {
    alert(`Hello, ${this.name}!`);
  }
}

new User().sayHi();

alert(User.prototype.sayHi); // 被放在 User.prototype 中
alert(User.prototype.name); // undefined, 没有被放在 User.prototype 中
```

关于类字段的重要一点是，它们设置在单个对象上的，而不是设置在 `User.prototype` 上的。从技术上讲，它们是在 `constructor` 完成工作后被处理的。

使用类字段制作绑定方法

正如 [函数绑定](#) 一章中所讲的，JavaScript 中的函数具有动态的 `this`。它取决于调用上下文。

因此，如果一个对象方法被传递到某处，或者在另一个上下文中被调用，则 `this` 将不再是对其对象的引用。

例如，此代码将显示 `undefined`：

```
class Button {
  constructor(value) {
    this.value = value;
  }

  click() {
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // undefined
```

这个问题被称为“丢失 `this`”。

我们在 [函数绑定](#) 一章中讲过，有两种可以修复它的方式：

1. 传递一个包装函数，例如 `setTimeout(() => button.click(), 1000)`。
2. 将方法绑定到对象，例如在 `constructor` 中：

```
class Button {
  constructor(value) {
    this.value = value;
    this.click = this.click.bind(this);
  }

  click() {
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // hello
```

类字段为后一种解决方案提供了更优雅的语法：

```
class Button {
  constructor(value) {
    this.value = value;
  }

  click = () => {
    alert(this.value);
  }
}

let button = new Button("hello");
```

```
setTimeout(button.click, 1000); // hello
```

类字段 `click = () => {...}` 在每个 `Button` 对象上创建一个独立的函数，并将 `this` 绑定到该对象上。然后，我们可以将 `button.click` 传递到任何地方，并且它会被以正确的 `this` 进行调用。

这在浏览器环境中，当我们需要将一个方法设置为事件监听器时尤其有用。

总结

基本的类语法看起来像这样：

```
class MyClass {  
  prop = value; // 属性  
  
  constructor(...) { // 构造器  
    // ...  
  }  
  
  method(...) {} // method  
  
  get something(...) {} // getter 方法  
  set something(...) {} // setter 方法  
  
  [Symbol.iterator]() {} // 有计算名称 (computed name) 的方法 (此处为 symbol)  
  // ...  
}
```

技术上来说，`MyClass` 是一个函数（我们提供作为 `constructor` 的那个），而 `methods`、`getters` 和 `setters` 都被写入了 `MyClass.prototype`。

在下一章，我们将会进一步学习类的相关知识，包括继承和其他功能。

类继承

类继承是一个类扩展另一个类的一种方式。

因此，我们可以在现有功能之上创建新功能。

“`extends`”关键字

假设我们有 `class Animal`：

```
class Animal {  
  constructor(name) {  
    this.speed = 0;  
    this.name = name;  
  }  
  run(speed) {  
    this.speed = speed;  
    alert(`${this.name} runs with speed ${this.speed}.`);  
  }  
}
```

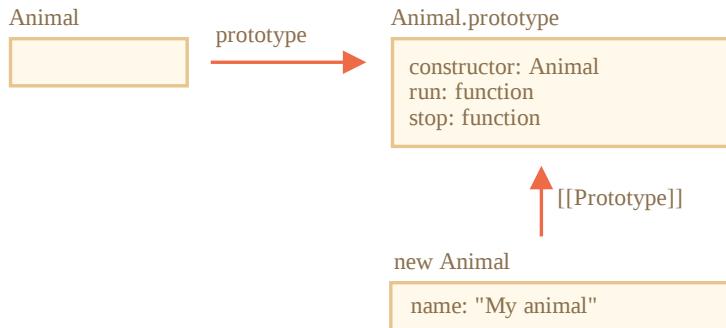
```

stop() {
  this.speed = 0;
  alert(`#${this.name} stands still.`);
}

let animal = new Animal("My animal");

```

这是我们对对象 `animal` 和 class `Animal` 的图形化表示:



.....然后我们想创建另一个 class `Rabbit`:

因为 `rabbits` 是 `animals`, 所以 class `Rabbit` 应该是基于 class `Animal` 的, 可以访问 `animal` 的方法, 以便 `rabbits` 可以做“一般”动物可以做的事情。

扩展另一个类的语法是: `class Child extends Parent`。

让我们创建一个继承自 `Animal` 的 `class Rabbit`:

```

class Rabbit extends Animal {
  hide() {
    alert(`#${this.name} hides!`);
  }
}

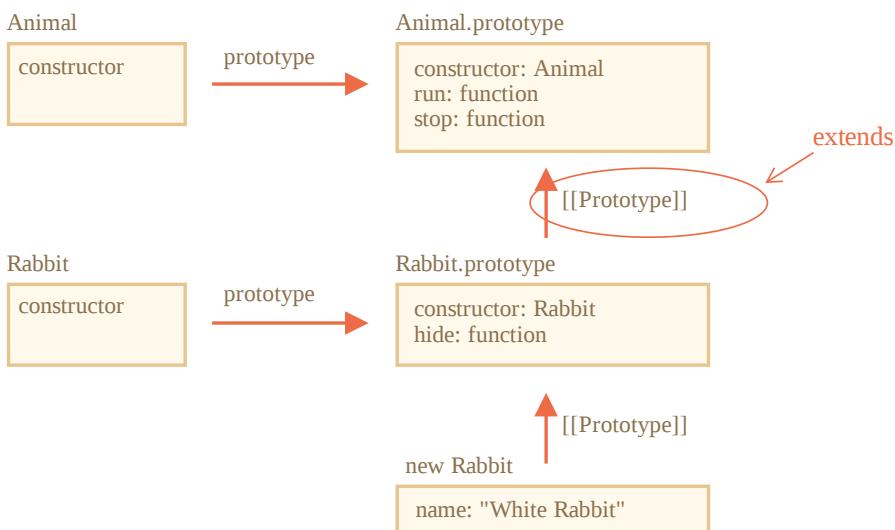
let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!

```

Class `Rabbit` 的对象可以访问例如 `rabbit.hide()` 等 `Rabbit` 的方法, 还可以访问例如 `rabbit.run()` 等 `Animal` 的方法。

在内部, 关键字 `extends` 使用了很好的旧的原型机制进行工作。它将 `Rabbit.prototype.[[Prototype]]` 设置为 `Animal.prototype`。所以, 如果在 `Rabbit.prototype` 中找不到一个方法, JavaScript 就会从 `Animal.prototype` 中获取该方法。



例如，要查找 `rabbit.run` 方法，JavaScript 引擎会进行如下检查（如图所示从下到上）：

1. 查找对象 `rabbit`（没有 `run`）。
2. 查找它的原型，即 `Rabbit.prototype`（有 `hide`，但没有 `run`）。
3. 查找它的原型，即（由于 `extends`） `Animal.prototype`，在这儿找到了 `run` 方法。

我们可以回忆一下 [原生的原型](#) 这一章的内容，JavaScript 内建对象同样也使用原型继承。例如，`Date.prototype.[[Prototype]]` 是 `Object.prototype`。这就是为什么日期可以访问通用对象的方法。

① 在 `extends` 后允许任意表达式

类语法不仅允许指定一个类，在 `extends` 后可以指定任意表达式。

例如，一个生成父类的函数调用：

```

function f(phrase) {
  return class {
    sayHi() { alert(phrase) }
  }
}

class User extends f("Hello") {}

new User().sayHi(); // Hello

```

这里 `class User` 继承自 `f("Hello")` 的结果。

这对于高级编程模式，例如当我们根据许多条件使用函数生成类，并继承它们时来说可能很有用。

重写方法

现在，让我们继续前行并尝试重写一个方法。默认情况下，所有未在 `class Rabbit` 中指定的方法均从 `class Animal` 中直接获取。

但是如果我们在 `Rabbit` 中指定了我们自己的方法，例如 `stop()`，那么将会使用它：

```
class Rabbit extends Animal {  
    stop() {  
        // .....现在这个将被用作 rabbit.stop()  
        // 而不是来自于 class Animal 的 stop()  
    }  
}
```

但是通常来说，我们不希望完全替换父类的方法，而是希望在父类方法的基础上进行调整或扩展其功能。我们在我们的方法中做一些事儿，但是在它之前或之后或在过程中会调用父类方法。

Class 为此提供了 "super" 关键字。

- 执行 `super.method(...)` 来调用一个父类方法。
- 执行 `super(...)` 来调用一个父类 `constructor`（只能在我们的 `constructor` 中）。

例如，让我们的 `rabbit` 在停下来的时候自动 `hide`:

```
class Animal {  
  
    constructor(name) {  
        this.speed = 0;  
        this.name = name;  
    }  
  
    run(speed) {  
        this.speed = speed;  
        alert(`${this.name} runs with speed ${this.speed}.`);  
    }  
  
    stop() {  
        this.speed = 0;  
        alert(`${this.name} stands still.`);  
    }  
}  
  
class Rabbit extends Animal {  
    hide() {  
        alert(`${this.name} hides!`);  
    }  
  
    stop() {  
        super.stop(); // 调用父类的 stop  
        this.hide(); // 然后 hide  
    }  
}  
  
let rabbit = new Rabbit("White Rabbit");  
  
rabbit.run(5); // White Rabbit 以速度 5 奔跑  
rabbit.stop(); // White Rabbit 停止了。White rabbit hide 了!
```

现在，`Rabbit` 在执行过程中调用父类的 `super.stop()` 方法，所以 `Rabbit` 也具有了 `stop` 方法。

① 箭头函数没有 `super`

正如我们在 [深入理解箭头函数](#) 一章中所提到的，箭头函数没有 `super`。

如果被访问，它会从外部函数获取。例如：

```
class Rabbit extends Animal {  
    stop() {  
        setTimeout(() => super.stop(), 1000); // 1 秒后调用父类的 stop  
    }  
}
```

箭头函数中的 `super` 与 `stop()` 中的是一样的，所以它能按预期工作。如果我们在这里指定一个“普通”函数，那么将会抛出错误：

```
// 意料之外的 super  
setTimeout(function() { super.stop() }, 1000);
```

重写 `constructor`

对于重写 `constructor` 来说，则有点棘手。

到目前为止，`Rabbit` 还没有自己的 `constructor`。

根据 [规范 ↗](#)，如果一个类扩展了另一个类并且没有 `constructor`，那么将生成下面这样的“空”`constructor`：

```
class Rabbit extends Animal {  
    // 为没有自己的 constructor 的扩展类生成的  
    constructor(...args) {  
        super(...args);  
    }  
}
```

正如我们所看到的，它调用了父类的 `constructor`，并传递了所有的参数。如果我们没有写自己的 `constructor`，就会出现这种情况。

现在，我们给 `Rabbit` 添加一个自定义的 `constructor`。除了 `name` 之外，它还会指定 `earLength`。

```
class Animal {  
    constructor(name) {  
        this.speed = 0;  
        this.name = name;  
    }  
    // ...  
}  
  
class Rabbit extends Animal {
```

```
constructor(name, earLength) {
  this.speed = 0;
  this.name = name;
  this.earLength = earLength;
}

// ...
}

// 不工作!
let rabbit = new Rabbit("White Rabbit", 10); // Error: this is not defined.
```

哎呦！我们得到了一个报错。现在我们没法新建 `rabbit`。是什么地方出错了？

简短的解释是：继承类的 `constructor` 必须调用 `super(...)`，并且 (!) 一定要在使用 `this` 之前调用。

.....但这是为什么呢？这里发生了什么？确实，这个要求看起来很奇怪。

当然，本文会给出一个解释。让我们深入细节，这样你就可以真正地理解发生了什么。

在 JavaScript 中，继承类（所谓的“派生构造器”，英文为“derived constructor”）的构造函数与其他函数之间是有区别的。派生构造器具有特殊的内部属性

`[[ConstructorKind]]:"derived"`。这是一个特殊的内部标签。

该标签会影响它的 `new` 行为：

- 当通过 `new` 执行一个常规函数时，它将创建一个空对象，并将这个空对象赋值给 `this`。
- 但是当继承的 `constructor` 执行时，它不会执行此操作。它期望父类的 `constructor` 来完成这项工作。

因此，派生的 `constructor` 必须调用 `super` 才能执行其父类（非派生的）的 `constructor`，否则 `this` 指向的那个对象将不会被创建。并且我们会收到一个报错。

为了让 `Rabbit` 的 `constructor` 可以工作，它需要在使用 `this` 之前调用 `super()`，就像下面这样：

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }

  // ...
}
```

```
// 现在可以了
let rabbit = new Rabbit("White Rabbit", 10);
alert(rabbit.name); // White Rabbit
alert(rabbit.earLength); // 10
```

深入：内部探究和 [[HomeObject]]

⚠ 进阶内容

如果你是第一次阅读本教程，那么则可以跳过本节。

这是关于继承和 `super` 背后的内部机制。

让我们更深入地研究 `super`。我们将在这个过程中发现一些有趣的事儿。

首先要说的是，从我们迄今为止学到的知识来看，`super` 是不可能运行的。

的确是这样，让我们问问自己，以技术的角度它是如何工作的？当一个对象方法执行时，它会将当前对象作为 `this`。随后如果我们调用 `super.method()`，那么引擎需要从当前对象的原型中获取 `method`。但这是怎么做到的？

这个任务看起来是挺容易的，但其实并不简单。引擎知道当前对象的 `this`，所以它可以获取父 `method` 作为 `this.__proto__.method`。不幸的是，这个“天真”的解决方法是行不通的。

让我们演示一下这个问题。简单起见，我们使用普通对象而不使用类。

如果你不想知道更多的细节知识，你可以跳过此部分，并转到下面的 `[[HomeObject]]` 小节。这没关系的。但如果你感兴趣，想学习更深入的知识，那就继续阅读吧。

在下面的例子中，`rabbit.__proto__ = animal`。现在让我们尝试一下：在 `rabbit.eat()` 我们将会使用 `this.__proto__` 调用 `animal.eat()`：

```
let animal = {
  name: "Animal",
  eat() {
    alert(` ${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Rabbit",
  eat() {
    // 这就是 super.eat() 可以大概工作的方式
    this.__proto__.eat.call(this); // (*)
  }
};

rabbit.eat(); // Rabbit eats.
```

在 `(*)` 这一行，我们从原型（`animal`）中获取 `eat`，并在当前对象的上下文中调用它。请注意，`.call(this)` 在这里非常重要，因为简单的调用 `this.__proto__.eat()` 将在原型的上下文中执行 `eat`，而非当前对象。

在上面的代码中，它确实按照了期望运行：我们获得了正确的 `alert`。

现在，让我们在原型链上再添加一个对象。我们将看到这件事是如何被打破的：

```
let animal = {
  name: "Animal",
  eat() {
    alert(` ${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  eat() {
    // ...bounce around rabbit-style and call parent (animal) method
    this.__proto__.eat.call(this); // (*)
  }
};

let longEar = {
  __proto__: rabbit,
  eat() {
    // ...do something with long ears and call parent (rabbit) method
    this.__proto__.eat.call(this); // (**)
  }
};

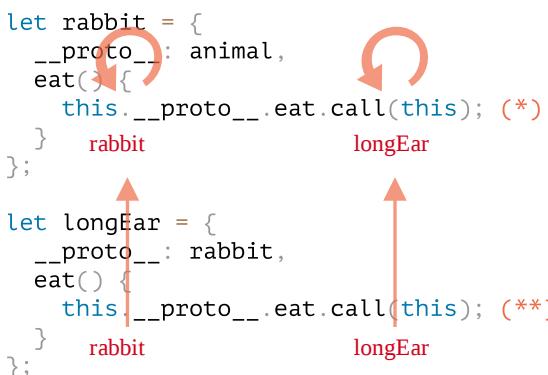
longEar.eat(); // Error: Maximum call stack size exceeded
```

代码无法再运行了！我们可以看到，在试图调用 `longEar.eat()` 时抛出了错误。

原因可能不那么明显，但是如果我们跟踪 `longEar.eat()` 调用，就可以发现原因。在 `(*)` 和 `(**)` 这两行中，`this` 的值都是当前对象（`longEar`）。这是至关重要的一点：所有的对象方法都将当前对象作为 `this`，而非原型或其他什么东西。

因此，在 `(*)` 和 `(**)` 这两行中，`this.__proto__` 的值是完全相同的：都是 `rabbit`。它们俩都调用的是 `rabbit.eat`，它们在不停地循环调用自己，而不是在原型链上向上寻找方法。

这张图介绍了发生的情况：



1. 在 `longEar.eat()` 中，`(**)` 这一行调用 `rabbit.eat` 并为其提供 `this=longEar`。

```
// 在 longEar.eat() 中我们有 this = longEar
this.__proto__.eat.call(this) // (**)
// 变成了
```

```
longEar.__proto__.eat.call(this)
// 也就是
rabbit.eat.call(this);
```

2. 之后在 `rabbit.eat` 的 (*) 行中，我们希望将函数调用在原型链上向更高层传递，但是 `this=longEar`，所以 `this.__proto__.eat` 又是 `rabbit.eat`！

```
// 在 rabbit.eat() 中我们依然有 this = longEar
this.__proto__.eat.call(this) // (*)
// 变成了
longEar.__proto__.eat.call(this)
// 或 (再一次)
rabbit.eat.call(this);
```

3.所以 `rabbit.eat` 在不停地循环调用自己，因此它无法进一步地提升。

这个问题没法仅仅通过使用 `this` 来解决。

[[HomeObject]]

为了提供解决方法，JavaScript 为函数添加了一个特殊的内部属性：`[[HomeObject]]`。

当一个函数被定义为类或者对象方法时，它的 `[[HomeObject]]` 属性就成为了该对象。

然后 `super` 使用它来解析（`resolve`）父原型和它自己的方法。

让我们看看它是怎么工作的，首先，对于普通对象：

```
let animal = {
  name: "Animal",
  eat() {           // animal.eat.[[HomeObject]] == animal
    alert(`#${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Rabbit",
  eat() {           // rabbit.eat.[[HomeObject]] == rabbit
    super.eat();
  }
};

let longEar = {
  __proto__: rabbit,
  name: "Long Ear",
  eat() {           // longEar.eat.[[HomeObject]] == longEar
    super.eat();
  }
};

// 正确执行
longEar.eat(); // Long Ear eats.
```

它基于 `[[HomeObject]]` 运行机制按照预期执行。一个方法，例如 `longEar.eat`，知道其 `[[HomeObject]]` 并且从其原型中获取父方法。并没有使用 `this`。

方法并不是“自由”的

正如我们之前所知道的，函数通常都是“自由”的，并没有绑定到 JavaScript 中的对象。正因如此，它们可以在对象之间复制，并用另外一个 `this` 调用它。

`[[HomeObject]]` 的存在违反了这个原则，因为方法记住了它们的对象。`[[HomeObject]]` 不能被更改，所以这个绑定是永久的。

在 JavaScript 语言中 `[[HomeObject]]` 仅被用于 `super`。所以，如果一个方法不使用 `super`，那么我们仍然可以视它为自由的并且可在对象之间复制。但是用了 `super` 再这样做可能就会出错。

下面是复制后错误的 `super` 结果的示例：

```
let animal = {
  sayHi() {
    console.log(`I'm an animal`);
  }
};

// rabbit 继承自 animal
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
};

let plant = {
  sayHi() {
    console.log("I'm a plant");
  }
};

// tree 继承自 plant
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi // (*)
};

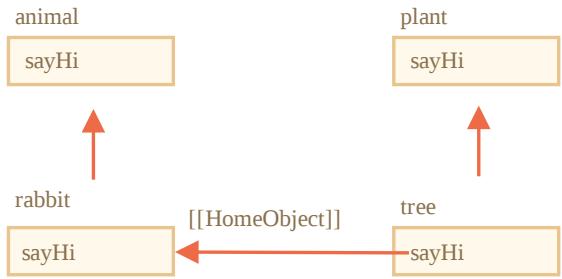
tree.sayHi(); // I'm an animal (!?)
```

调用 `tree.sayHi()` 显示 “I'm an animal”。这绝对是错误的。

原因很简单：

- 在 (*) 行，`tree.sayHi` 方法是从 `rabbit` 复制而来。也许我们只是想避免重复代码？
- 它的 `[[HomeObject]]` 是 `rabbit`，因为它是在 `rabbit` 中创建的。没有办法修改 `[[HomeObject]]`。
- `tree.sayHi()` 内具有 `super.sayHi()`。它从 `rabbit` 中上溯，然后从 `animal` 中获取方法。

这是发生的情况示意图：



方法，不是函数属性

`[[HomeObject]]` 是为类和普通对象中的方法定义的。但是对于对象而言，方法必须确切指定为 `method()`，而不是 `"method: function()"`。

这个差别对我们来说可能不重要，但是对 JavaScript 来说却非常重要。

在下面的例子中，使用非方法（non-method）语法进行了比较。未设置 `[[HomeObject]]` 属性，并且继承无效：

```

let animal = {
  eat: function() { // 这里是故意这样写的，而不是 eat() {...}
    // ...
  }
};

let rabbit = {
  __proto__: animal,
  eat: function() {
    super.eat();
  }
};

rabbit.eat(); // 错误调用 super (因为这里没有 [[HomeObject]])

```

总结

1. 想要扩展一个类： `class Child extends Parent`：
 - 这意味着 `Child.prototype.__proto__` 将是 `Parent.prototype`，所以方法会被继承。
2. 重写一个 `constructor`：
 - 在使用 `this` 之前，我们必须在 `Child` 的 `constructor` 中将父 `constructor` 调用为 `super()`。
3. 重写一个方法：
 - 我们可以在一个 `Child` 方法中使用 `super.method()` 来调用 `Parent` 方法。
4. 内部：
 - 方法在内部的 `[[HomeObject]]` 属性中记住了它们的类/对象。这就是 `super` 如何解析父方法的。
 - 因此，将一个带有 `super` 的方法从一个对象复制到另一个对象是不安全的。

补充：

- 箭头函数没有自己的 `this` 或 `super`，所以它们能融入到就近的上下文中，像透明似的。

静态属性和静态方法

我们可以把一个方法赋值给类的函数本身，而不是赋给它的 "prototype"。这样的方法被称为 **静态的 (static)**。

在一个类中，它们以 `static` 关键字开头，如下所示：

```
class User {  
    static staticMethod() {  
        alert(this === User);  
    }  
}  
  
User.staticMethod(); // true
```

这实际上跟直接将其作为属性赋值的作用相同：

```
class User {}  
  
User.staticMethod = function() {  
    alert(this === User);  
};  
  
User.staticMethod(); // true
```

在 `User.staticMethod()` 调用中的 `this` 的值是类构造器 `User` 自身（“点符号前面的对象”规则）。

通常，静态方法用于实现属于该类但不属于该类任何特定对象的函数。

例如，我们有对象 `Article`，并且需要一个方法来比较它们。一个自然的解决方案就是添加 `Article.compare` 方法，像下面这样：

```
class Article {  
    constructor(title, date) {  
        this.title = title;  
        this.date = date;  
    }  
  
    static compare(articleA, articleB) {  
        return articleA.date - articleB.date;  
    }  
}  
  
// 用法  
let articles = [  
    new Article("HTML", new Date(2019, 1, 1)),  
    new Article("CSS", new Date(2019, 0, 1)),  
    new Article("JavaScript", new Date(2019, 11, 1))  
];  
  
articles.sort(Article.compare);  
  
alert(articles[0].title); // CSS
```

这里 `Article.compare` 代表“上面的”文章，意思是比較它们。它不是文章的方法，而是整个 `class` 的方法。

另一个例子是所谓的“工厂”方法。想象一下，我们需要通过几种方法来创建一个文章：

1. 通过用给定的参数来创建（`title`, `date` 等）。
2. 使用今天的日期来创建一个空的文章。
3.其它方法。

第一种方法我们可以通过 `constructor` 来实现。对于第二种方式，我们可以创建类的一个静态方法来实现。

就像这里的 `Article.createTodays()`：

```
class Article {  
  constructor(title, date) {  
    this.title = title;  
    this.date = date;  
  }  
  
  static createTodays() {  
    // 记住 this = Article  
    return new this("Today's digest", new Date());  
  }  
}  
  
let article = Article.createTodays();  
  
alert( article.title ); // Today's digest
```

现在，每当我们需要创建一个今天的文章时，我们就可以调用 `Article.createTodays()`。再说明一次，它不是一个文章的方法，而是整个 `class` 的方法。

静态方法也被用于与数据库相关的公共类，可以用于搜索/保存/删除数据库中的条目，就像这样：

```
// 假定 Article 是一个用来管理文章的特殊类  
// 静态方法用于移除文章：  
Article.remove({id: 12345});
```

静态属性

⚠ A recent addition

This is a recent addition to the language. Examples work in the recent Chrome.

静态的属性也是可能的，它们看起来就像常规的类属性，但前面加有 `static`：

```
class Article {  
  static publisher = "Levi Ding";  
}
```

```
alert( Article.publisher ); // Levi Ding
```

这等同于直接给 `Article` 赋值：

```
Article.publisher = "Levi Ding";
```

继承静态属性和方法

静态属性和方法是可被继承的。

例如，下面这段代码中的 `Animal.compare` 和 `Animal.planet` 是可被继承的，可以通过 `Rabbit.compare` 和 `Rabbit.planet` 来访问：

```
class Animal {
  static planet = "Earth";

  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }

  run(speed = 0) {
    this.speed += speed;
    alert(` ${this.name} runs with speed ${this.speed}. `);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }
}

// 继承于 Animal
class Rabbit extends Animal {
  hide() {
    alert(` ${this.name} hides! `);
  }
}

let rabbits = [
  new Rabbit("White Rabbit", 10),
  new Rabbit("Black Rabbit", 5)
];

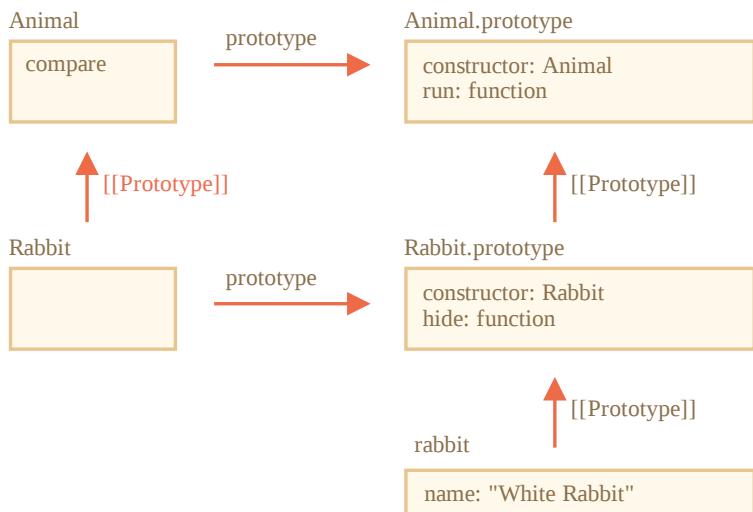
rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Black Rabbit runs with speed 5.

alert(Rabbit.planet); // Earth
```

现在我们调用 `Rabbit.compare` 时，继承的 `Animal.compare` 将会被调用。

它是如何工作的？再次，使用原型。你可能已经猜到了，`extends` 让 `Rabbit` 的 `[[Prototype]]` 指向了 `Animal`。



所以，`Rabbit extends Animal` 创建了两个 `[[Prototype]]` 引用：

1. `Rabbit` 函数原型继承自 `Animal` 函数。
2. `Rabbit.prototype` 原型继承自 `Animal.prototype`。

结果就是，继承对常规方法和静态方法都有效。

这里，让我们通过代码来检验一下：

```
class Animal {}  
class Rabbit extends Animal {}  
  
// 对于静态的  
alert(Rabbit.__proto__ === Animal); // true  
  
// 对于常规方法  
alert(Rabbit.prototype.__proto__ === Animal.prototype); // true
```

总结

静态方法被用于实现属于整个类的功能。它与具体的类实例无关。

举个例子，一个用于进行比较的方法 `Article.compare(article1, article2)` 或一个工厂（factory）方法 `Article.createTodays()`。

在类生命中，它们都被用关键字 `static` 进行了标记。

静态属性被用于当我们想要存储类级别的数据时，而不是绑定到实例。

语法如下所示：

```
class MyClass {  
    static property = ...;  
  
    static method() {  
        ...  
    }  
}
```

```
    }  
}
```

从技术上讲，静态声明与直接给类本身赋值相同：

```
MyClass.property = ...  
MyClass.method = ...
```

静态属性和方法是可被继承的。

对于 `class B extends A`，类 `B` 的 `prototype` 指向了 `A: B.[[Prototype]] = A`。因此，如果一个字段在 `B` 中没有找到，会继续在 `A` 中查找。

私有的和受保护的属性和方法

面向对象编程最重要的原则之一——将内部接口与外部接口分隔开来。

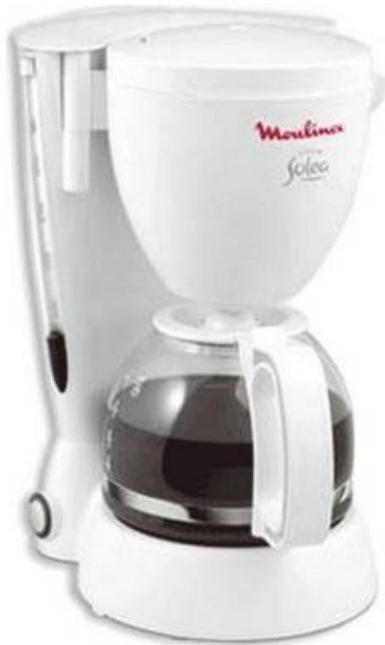
在开发比“hello world”应用程序更复杂的东西时，这是“必须”遵守的做法。

为了理解这一点，让我们脱离开发过程，把目光转向现实世界。

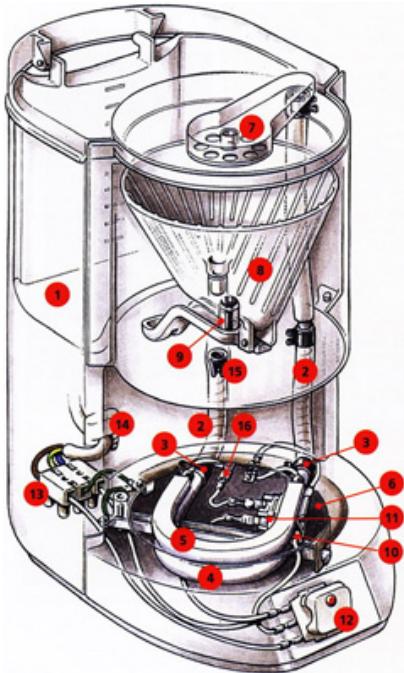
通常，我们使用的设备都非常复杂。但是，将内部接口与外部接口分隔开来可以让我们使用它们且没有任何问题。

一个现实生活中的例子

例如，一个咖啡机。从外面看很简单：一个按钮，一个显示器，几个洞……当然，结果就是——很棒的咖啡！:)



但是在内部……（一张摘自维修手册的图片）



有非常多的细节。但我们可以完全不了解这些内部细节的情况下使用它。

咖啡机非常可靠，不是吗？一台咖啡机我们可以使用好几年，只有在出现问题时——把它送去维修。

咖啡机的可靠性和简洁性的秘诀——所有细节都经过精心校并 **隐藏** 在内部。

如果我们从咖啡机上取下保护罩，那么使用它将变得复杂得多（要按哪里？），并且很危险（会触电）。

正如我们所看到的，在编程中，对象就像咖啡机。

但是为了隐藏内部细节，我们不会使用保护罩，而是使用语言和约定中的特殊语法。

内部接口和外部接口

在面向对象的编程中，属性和方法分为两组：

- **内部接口**——可以通过该类的其他方法访问，但不能从外部访问的方法和属性。
- **外部接口**——也可以从类的外部访问的方法和属性。

如果我们继续用咖啡机进行类比——内部隐藏的内容：锅炉管，加热元件等——是咖啡机的内部接口。

内部接口用于对象工作，它的细节相互使用。例如，锅炉管连接到加热元件。

但是从外面看，一台咖啡机被保护壳罩住了，所以没有人可以接触到其内部接口。细节信息被隐藏起来并且无法访问。我们可以通过外部接口使用它的功能。

所以，我们需要使用一个对象时只需知道它的外部接口。我们可能完全不知道它的内部是如何工作的，这太好了。

这是个概括性的介绍。

在 JavaScript 中，有两种类型的对象字段（属性和方法）：

- 公共的：可从任何地方访问。它们构成了外部接口。到目前为止，我们只使用了公共的属性和方法。

- 私有的：只能从类的内部访问。这些用于内部接口。

在许多其他编程语言中，还存在“受保护”的字段：只能从类的内部和基于其扩展的类的内部访问（例如私有的，但可以从继承的类进行访问）。它们对于内部接口也很有用。从某种意义上讲，它们比私有的属性和方法更为广泛，因为我们通常希望继承类来访问它们。

受保护的字段不是在语言级别的 **Javascript** 中实现的，但实际上它们非常方便，因为它们是在 **Javascript** 中模拟的类定义语法。

现在，我们将使用所有这些类型的属性在 **Javascript** 中制作咖啡机。咖啡机有很多细节，我们不会对它们进行全面模拟以保持简洁（尽管我们可以）。

受保护的“waterAmount”

首先，让我们做一个简单的咖啡机类：

```
class CoffeeMachine {
  waterAmount = 0; // 内部的水量

  constructor(power) {
    this.power = power;
    alert(`Created a coffee-machine, power: ${power}`);
  }
}

// 创建咖啡机
let coffeeMachine = new CoffeeMachine(100);

// 加水
coffeeMachine.waterAmount = 200;
```

现在，属性 `waterAmount` 和 `power` 是公共的。我们可以轻松地从外部将它们 `get/set` 成任何值。

让我们将 `waterAmount` 属性更改为受保护的属性，以对其进行更多控制。例如，我们不希望任何人将它的值设置为小于零的数。

受保护的属性通常以下划线 `_` 作为前缀。

这不是在语言级别强制实施的，但是程序员之间有一个众所周知的约定，即不应该从外部访问此类型的属性和方法。

所以我们的属性将被命名为 `_waterAmount`：

```
class CoffeeMachine {
  _waterAmount = 0;

  set waterAmount(value) {
    if (value < 0) throw new Error("Negative water");
    this._waterAmount = value;
  }

  get waterAmount() {
    return this._waterAmount;
```

```
}

constructor(power) {
  this._power = power;
}

}

// 创建咖啡机
let coffeeMachine = new CoffeeMachine(100);

// 加水
coffeeMachine.waterAmount = -10; // Error: Negative water
```

现在访问已受到控制，因此将水量的值设置为小于零的数将会失败。

只读的“power”

对于 `power` 属性，让我们将它设为只读。有时候一个属性必须只能被在创建时进行设置，之后不再被修改。

咖啡机就是这种情况：功率永远不会改变。

要做到这一点，我们只需要设置 `getter`，而不设置 `setter`：

```
class CoffeeMachine {
  // ...

  constructor(power) {
    this._power = power;
  }

  get power() {
    return this._power;
  }
}

// 创建咖啡机
let coffeeMachine = new CoffeeMachine(100);

alert(`Power is: ${coffeeMachine.power}W`); // 功率是: 100W

coffeeMachine.power = 25; // Error (没有 setter)
```

① Getter/setter 函数

这里我们使用了 `getter/setter` 语法。

但大多数时候首选 `get.../set...` 函数，像这样：

```
class CoffeeMachine {
    _waterAmount = 0;

    setWaterAmount(value) {
        if (value < 0) throw new Error("Negative water");
        this._waterAmount = value;
    }

    getWaterAmount() {
        return this._waterAmount;
    }
}

new CoffeeMachine().setWaterAmount(100);
```

这看起来有点长，但函数更灵活。它们可以接受多个参数（即使我们现在还不需要）。

另一方面，`get/set` 语法更短，所以最终没有严格的规定，而是由你自己来决定。

① 受保护的字段是可以被继承的

如果我们继承 `class MegaMachine extends CoffeeMachine`，那么什么都无法阻止我们从新的类中的方法访问 `this._waterAmount` 或 `this._power`。

所以受保护的字段是自然可被继承的。与我们接下来将看到的私有字段不同。

私有的 “#waterLimit”

⚠ A recent addition

This is a recent addition to the language. Not supported in JavaScript engines, or supported partially yet, requires polyfilling.

这儿有一个马上就会被加到规范中的已完成的 `Javascript` 提案，它为私有属性和方法提供语言级支持。

私有属性和方法应该以 `#` 开头。它们只在类的内部可被访问。

例如，这儿有一个私有属性 `#waterLimit` 和检查水量的私有方法 `#checkWater`：

```
class CoffeeMachine {
    #waterLimit = 200;

    #checkWater(value) {
        if (value < 0) throw new Error("Negative water");
        if (value > this.#waterLimit) throw new Error("Too much water");
    }
}
```

```
}
```

```
let coffeeMachine = new CoffeeMachine();
```

```
// 不能从类的外部访问类的私有属性和方法
```

```
coffeeMachine.#checkWater(); // Error
```

```
coffeeMachine.#waterLimit = 1000; // Error
```

在语言级别，`#` 是该字段为私有的特殊标志。我们无法从外部或从继承的类中访问它。

私有字段与公共字段不会发生冲突。我们可以同时拥有私有的 `#waterAmount` 和公共的 `waterAmount` 字段。

例如，让我们使 `waterAmount` 成为 `#waterAmount` 的一个访问器：

```
class CoffeeMachine {
```

```
#waterAmount = 0;
```

```
get waterAmount() {
```

```
    return this.#waterAmount;
```

```
}
```

```
set waterAmount(value) {
```

```
    if (value < 0) throw new Error("Negative water");
```

```
    this.#waterAmount = value;
```

```
}
```

```
}
```

```
let machine = new CoffeeMachine();
```

```
machine.waterAmount = 100;
```

```
alert(machine.#waterAmount); // Error
```

与受保护的字段不同，私有字段由语言本身强制执行。这是好事儿。

但是如果我们继承自 `CoffeeMachine`，那么我们将无法直接访问 `#waterAmount`。我们需要依靠 `waterAmount` getter/setter：

```
class MegaCoffeeMachine extends CoffeeMachine {
```

```
method() {
```

```
    alert( this.#waterAmount ); // Error: can only access from CoffeeMachine
```

```
}
```

```
}
```

在许多情况下，这种限制太严重了。如果我们扩展 `CoffeeMachine`，则可能有正当理由访问其内部。这就是为什么大多数时候都会使用受保护字段，即使它们不受语言语法的支持。

⚠ 私有字段不能通过 `this[name]` 访问

私有字段很特别。

正如我们所知道的，通常我们可以使用 `this[name]` 访问字段：

```
class User {  
  ...  
  sayHi() {  
    let fieldName = "name";  
    alert(`Hello, ${this[fieldName]}`);  
  }  
}
```

对于私有字段来说，这是不可能的：`this['#name']` 不起作用。这是确保私有性的语法限制。

总结

就面向对象编程（OOP）而言，内部接口与外部接口的划分被称为 [封装](#)。

它具有以下优点：

保护用户，使他们不会误伤自己

想象一下，有一群开发人员在使用一个咖啡机。这个咖啡机是由“最好的咖啡机”公司制造的，工作正常，但是保护罩被拿掉了。因此内部接口暴露了出来。

所有的开发人员都是文明的——他们按照预期使用咖啡机。但其中的一个人，约翰，他认为自己是最聪明的人，并对咖啡机的内部做了一些调整。然而，咖啡机两天后就坏了。

这肯定不是约翰的错，而是那个取下保护罩并让约翰进行操作的人的错。

编程也一样。如果一个 `class` 的使用者想要改变那些本不打算被从外部更改的东西——后果是不可预测的。

可支持性

编程的情况比现实生活中的咖啡机要复杂得多，因为我们不只是购买一次。我们还需要不断开发和改进代码。

如果我们严格界定内部接口，那么这个 `class` 的开发人员可以自由地更改其内部属性和方法，甚至无需通知用户。

如果你是这样的 `class` 的开发者，那么你会很高兴知道可以安全地重命名私有变量，可以更改甚至删除其参数，因为没有外部代码依赖于它们。

对于用户来说，当新版本问世时，应用的内部可能被进行了全面检修，但如果外部接口相同，则仍然很容易升级。

隐藏复杂性

人们喜欢使用简单的东西。至少从外部来看是这样。内部的东西则是另外一回事了。

程序员也不例外。

当实施细节被隐藏，并提供了简单且有据可查的外部接口时，总是很方便的。

为了隐藏内部接口，我们使用受保护的或私有的属性：

- 受保护的字段以 `_` 开头。这是一个众所周知的约定，不是在语言级别强制执行的。程序员应该只通过它的类和从它继承的类中访问以 `_` 开头的字段。
- 私有字段以 `#` 开头。`JavaScript` 确保我们只能从类的内部访问它们。

目前，各个浏览器对私有字段的支持不是很好，但可以用 `polyfill` 解决。

扩展内建类

内建的类，例如 `Array`，`Map` 等也都是可以扩展的（extendable）。

例如，这里有一个继承自原生 `Array` 的类 `PowerArray`：

```
// 给 PowerArray 新增了一个方法（可以增加更多）
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

请注意一个非常有趣的事儿。内建的方法例如 `filter`，`map` 等 — 返回的正是子类 `PowerArray` 的新对象。它们内部使用了对象的 `constructor` 属性来实现这一功能。

在上面的例子中，

```
arr.constructor === PowerArray
```

当 `arr.filter()` 被调用时，它的内部使用的是 `arr.constructor` 来创建新的结果数组，而不是使用原生的 `Array`。这真的很酷，因为我们可以继续在结果数组上使用 `PowerArray` 的方法。

甚至，我们可以定制这种行为。

我们可以给这个类添加一个特殊的静态 getter `Symbol.species`。如果存在，则应返回 `JavaScript` 在内部用来在 `map` 和 `filter` 等方法中创建新实体的 `constructor`。

如果我们希望像 `map` 或 `filter` 这样的内建方法返回常规数组，我们可以在 `Symbol.species` 中返回 `Array`，就像这样：

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}
```

```

// 内建方法将使用这个作为 constructor
static get [Symbol.species]() {
  return Array;
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

// filter 使用 arr.constructor[Symbol.species] 作为 constructor 创建新数组
let filteredArr = arr.filter(item => item >= 10);

// filteredArr 不是 PowerArray, 而是 Array
alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function

```

正如你所看到的，现在 `.filter` 返回 `Array`。所以扩展的功能不再传递。

i 其他集合的工作方式类似

其他集合，例如 `Map` 和 `Set` 的工作方式类似。它们也使用 `Symbol.species`。

内建类没有静态方法继承

内建对象有它们自己的静态方法，例如 `Object.keys`, `Array.isArray` 等。

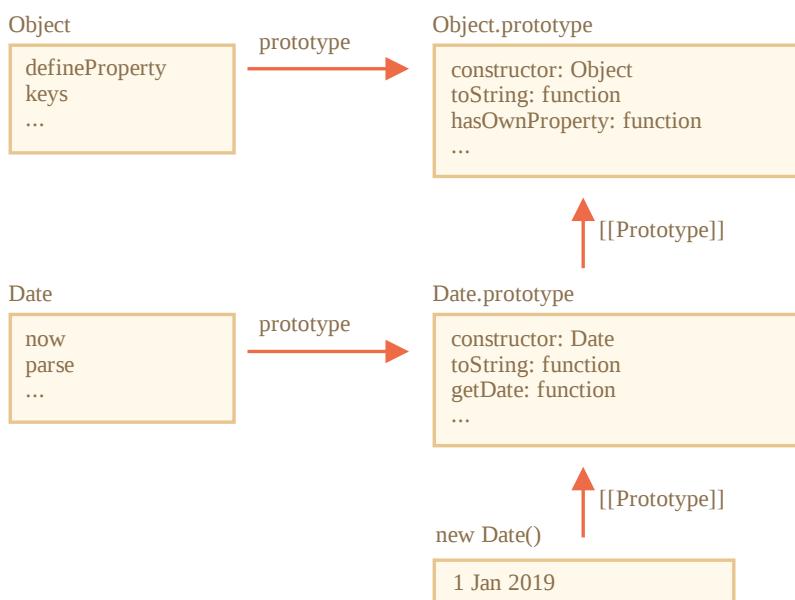
如我们所知道的，原生的类互相扩展。例如，`Array` 扩展自 `Object`。

通常，当一个类扩展另一个类时，静态方法和非静态方法都会被继承。这已经在 [静态属性和静态方法](#) 中详细地解释过了。

但内建类却是一个例外。它们相互间不继承静态方法。

例如，`Array` 和 `Data` 都继承自 `Object`，所以它们的实例都有来自 `Object.prototype` 的方法。但 `Array.[[Prototype]]` 并不指向 `Object`，所以它们没有例如 `Array.keys()` (或 `Data.keys()`) 这些静态方法。

这里有一张 `Date` 和 `Object` 的结构关系图：



正如你所看到的，`Date` 和 `Object` 之间没有连结。它们是独立的，只有 `Date.prototype` 继承自 `Object.prototype`，仅此而已。

与我们所了解的通过 `extends` 获得的继承相比，这是内建对象之间继承的一个重要区别。

类检查: "instanceof"

`instanceof` 操作符用于检查一个对象是否属于某个特定的 `class`。同时，它还考虑了继承。

在许多情况下，可能都需要进行此类检查。在这儿，我们将使用它来构建一个 **多态性 (polymorphic)** 的函数，该函数根据参数的类型对参数进行不同的处理。

instanceof 操作符

语法:

```
obj instanceof Class
```

如果 `obj` 隶属于 `Class` 类（或 `Class` 类的衍生类），则返回 `true`。

例如:

```
class Rabbit {}  
let rabbit = new Rabbit();  
  
// rabbit 是 Rabbit class 的对象吗?  
alert( rabbit instanceof Rabbit ); // true
```

它还可以与构造函数一起使用:

```
// 这里是构造函数，而不是 class  
function Rabbit() {}  
  
alert( new Rabbit() instanceof Rabbit ); // true
```

.....与诸如 `Array` 之类的内建 `class` 一起使用:

```
let arr = [1, 2, 3];  
alert( arr instanceof Array ); // true  
alert( arr instanceof Object ); // true
```

有一点需要留意，`arr` 同时还隶属于 `Object` 类。因为从原型上来说，`Array` 是继承自 `Object` 的。

通常，`instanceof` 在检查中会将原型链考虑在内。此外，我们还可以在静态方法 `Symbol.hasInstance` 中设置自定义逻辑。

`obj instanceof Class` 算法的执行过程大致如下:

1. 如果这儿有静态方法 `Symbol.hasInstance`，那就直接调用这个方法：

例如：

```
// 设置 instanceof 检查
// 并假设具有 canEat 属性的都是 animal
class Animal {
  static [Symbol.hasInstance](obj) {
    if (obj.canEat) return true;
  }
}

let obj = { canEat: true };

alert(obj instanceof Animal); // true: Animal[Symbol.hasInstance](obj) 被调用
```

2. 大多数 class 没有 `Symbol.hasInstance`。在这种情况下，标准的逻辑是：使用 `obj instanceof Class` 检查 `Class.prototype` 是否等于 `obj` 的原型链中的原型之一。

换句话说就是，一个接一个地比较：

```
obj.__proto__ === Class.prototype?
obj.__proto__.__proto__ === Class.prototype?
obj.__proto__.__proto__.__proto__ === Class.prototype?
...
// 如果任意一个的答案为 true，则返回 true
// 否则，如果我们已经检查到了原型链的尾端，则返回 false
```

在上面那个例子中，`rabbit.__proto__ === Rabbit.prototype`，所以立即就给出了结果。

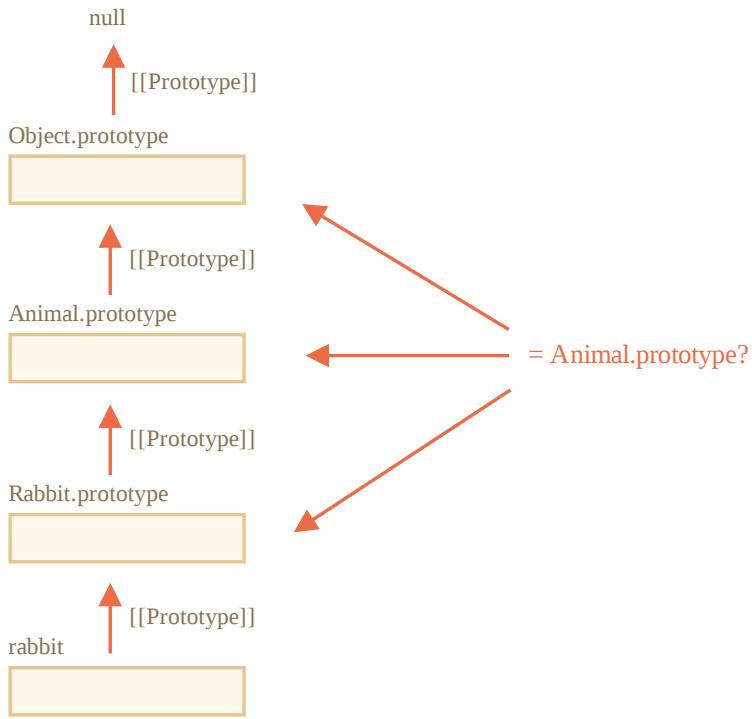
而在继承的例子中，匹配将在第二步进行：

```
class Animal {}
class Rabbit extends Animal {}

let rabbit = new Rabbit();
alert(rabbit instanceof Animal); // true

// rabbit.__proto__ === Rabbit.prototype
// rabbit.__proto__.__proto__ === Animal.prototype (匹配! )
```

下图展示了 `rabbit instanceof Animal` 的执行过程中，`Animal.prototype` 是如何参与比较的：



这里还要提到一个方法 `objA.isPrototypeOf(objB)`，如果 `objA` 处在 `objB` 的原型链中，则返回 `true`。所以，可以将 `obj instanceof Class` 检查改为 `Class.prototype.isPrototypeOf(obj)`。

这很有趣，但是 `Class` 的 `constructor` 自身是不参与检查的！检查过程只和原型链以及 `Class.prototype` 有关。

创建对象后，如果更改 `prototype` 属性，可能会导致有趣的结果。

就像这样：

```

function Rabbit() {}
let rabbit = new Rabbit();

// 修改了 prototype
Rabbit.prototype = {};

// ...再也不是 rabbit 了!
alert( rabbit instanceof Rabbit ); // false
  
```

福利：使用 `Object.prototype.toString` 方法来揭示类型

大家都知道，一个普通对象被转化为字符串时为 `[object Object]`：

```

let obj = {};
alert(obj); // [object Object]
alert(obj.toString()); // 同上
  
```

这是通过 `toString` 方法实现的。但是这儿有一个隐藏的功能，该功能可以使 `toString` 实际上比这更强大。我们可以将其作为 `typeof` 的增强版或者 `instanceof` 的替代方法来使用。

听起来挺不可思议？那是自然，精彩马上揭晓。

按照[规范](#)所讲，内建的 `toString` 方法可以被从对象中提取出来，并在任何其他值的上下文中执行。其结果取决于该值。

- 对于 `number` 类型，结果是 `[object Number]`
- 对于 `boolean` 类型，结果是 `[object Boolean]`
- 对于 `null`: `[object Null]`
- 对于 `undefined`: `[object Undefined]`
- 对于数组: `[object Array]`
-等（可自定义）

让我们演示一下：

```
// 方便起见，将 toString 方法复制到一个变量中
let objectToString = Object.prototype.toString;

// 它是什么类型的？
let arr = [];

alert( objectToString.call(arr) ); // [object Array]
```

这里我们用到了在[装饰者模式和转发，call/apply](#)一章中讲过的 `call` 方法来在上下文 `this=arr` 中执行函数 `objectToString`。

在内部，`toString` 的算法会检查 `this`，并返回相应地结果。再举几个例子：

```
let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]
alert( s.call(null) ); // [object Null]
alert( s.call(alert) ); // [object Function]
```

Symbol.toStringTag

可以使用特殊的对象属性 `Symbol.toStringTag` 自定义对象的 `toString` 方法的行为。

例如：

```
let user = {
  [Symbol.toStringTag]: "User"
};

alert( {}.toString.call(user) ); // [object User]
```

对于大多数特定于环境的对象，都有一个这样的属性。下面是一些特定于浏览器的示例：

```
// 特定于环境的对象和类的 toStringTag:
alert( window[Symbol.toStringTag] ); // window
alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest
```

```
alert( {}.toString.call(window) ); // [object Window]
alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]
```

正如我们所看到的，输出结果恰好是 `Symbol.toStringTag`（如果存在），只不过被包裹进了 `[object ...]` 里。

这样一来，我们手头上就有了个“磕了药似的 `typeof`”，不仅能检查原始数据类型，而且适用于内建对象，更可贵的是还支持自定义。

所以，如果我们想要获取内建对象的类型，并希望把该信息以字符串的形式返回，而不只是检查类型的话，我们可以用 `{}.toString.call` 替代 `instanceof`。

总结

让我们总结一下我们知道的类型检查方法：

	用于	返回值
<code>typeof</code>	原始数据类型	<code>string</code>
<code>{}.toString</code>	原始数据类型，内建对象，包含 <code>Symbol.toStringTag</code> 属性的对象	<code>string</code>
<code>instanceof</code>	对象	<code>true/false</code>

正如我们所看到的，从技术上讲，`{}.toString` 是一种“更高级的” `typeof`。

当我们使用类的层次结构（`hierarchy`），并想要对该类进行检查，同时还要考虑继承时，这种场景下 `instanceof` 操作符确实很出色。

Mixin 模式

在 JavaScript 中，我们只能继承单个对象。每个对象只能有一个 `[[Prototype]]`。并且每个类只可以扩展另外一个类。

但是有些时候这种设定（译注：单继承）会让人感到受限制。例如，我有一个 `StreetSweeper` 类和一个 `Bicycle` 类，现在想要一个它们的 mixin： `StreetSweepingBicycle` 类。

或者，我们有一个 `User` 类和一个 `EventEmitter` 类来实现事件生成（`event generation`），并且我们想将 `EventEmitter` 的功能添加到 `User` 中，以便我们的用户可以触发事件（`emit event`）。

有一个概念可以帮助我们，叫做“mixins”。

根据维基百科的定义，[mixin ↗](#) 是一个包含可被其他类使用而无需继承的方法的类。

换句话说，`mixin` 提供了实现特定行为的方法，但是我们不单独使用它，而是使用它来将这些行为添加到其他类中。

一个 Mixin 实例

在 JavaScript 中构造一个 mixin 最简单的方式就是构造一个拥有实用方法的对象，以便我们可以轻松地将这些实用的方法合并到任何类的原型中。

例如，这个名为 `sayHiMixin` 的 mixin 用于给 `User` 添加一些“语言功能”：

```

// mixin
let sayHiMixin = {
  sayHi() {
    alert(`Hello ${this.name}`);
  },
  sayBye() {
    alert(`Bye ${this.name}`);
  }
};

// 用法:
class User {
  constructor(name) {
    this.name = name;
  }
}

// 拷贝方法
Object.assign(User.prototype, sayHiMixin);

// 现在 User 可以打招呼了
new User("Dude").sayHi(); // Hello Dude!

```

这里没有继承，只有一个简单的方法拷贝。所以 `User` 可以从另一个类继承，还可以包括 mixin 来 "mix-in" 其它方法，就像这样：

```

class User extends Person {
  // ...
}

Object.assign(User.prototype, sayHiMixin);

```

`Mixin` 可以在自己内部使用继承。

例如，这里的 `sayHiMixin` 继承自 `sayMixin`：

```

let sayMixin = {
  say(phrase) {
    alert(phrase);
  }
};

let sayHiMixin = {
  __proto__: sayMixin, // (或者，我们可以在这儿使用 Object.create 来设置原型)

  sayHi() {
    // 调用父类方法
    super.say(`Hello ${this.name}`); // (*)
  },
  sayBye() {
    super.say(`Bye ${this.name}`); // (*)
  }
};

class User {

```

```

constructor(name) {
  this.name = name;
}

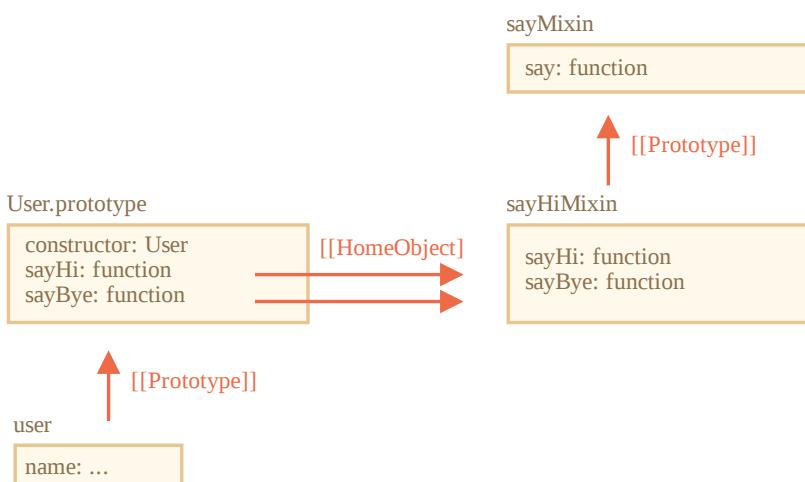
// 拷贝方法
Object.assign(User.prototype, sayHiMixin);

// 现在 User 可以打招呼了
new User("Dude").sayHi(); // Hello Dude!

```

请注意，在 `sayHiMixin` 内部对父类方法 `super.say()` 的调用（在标有 (*) 的行）会在 `mixin` 的原型中查找方法，而不是在 `class` 中查找。

这是示意图（请参见图中右侧部分）：



这是因为方法 `sayHi` 和 `sayBye` 最初是在 `sayHiMixin` 中创建的。因此，即使复制了它们，但是它们的 `[[HomeObject]]` 内部属性仍引用的是 `sayHiMixin`，如上图所示。

当 `super` 在 `[[HomeObject]].[[Prototype]]` 中寻找父方法时，意味着它搜索的是 `sayHiMixin.[[Prototype]]`，而不是 `User.[[Prototype]]`。

EventMixin

现在让我们为实际运用构造一个 `mixin`。

例如，许多浏览器对象的一个重要功能是它们可以生成事件。事件是向任何有需要的人“广播信息”的好方法。因此，让我们构造一个 `mixin`，使我们能够轻松地将与事件相关的函数添加到任意 `class/object` 中。

- `Mixin` 将提供 `.trigger(name, [...data])` 方法，以在发生重要的事情时“生成一个事件”。`name` 参数 (`arguments`) 是事件的名称，`[...data]` 是可选的带有事件数据的其他参数 (`arguments`)。
- 此外还有 `.on(name, handler)` 方法，它为具有给定名称的事件添加了 `handler` 函数作为监听器 (`listener`)。当具有给定 `name` 的事件触发时将调用该方法，并从 `.trigger` 调用中获取参数 (`arguments`)。
-还有 `.off(name, handler)` 方法，它会删除 `handler` 监听器 (`listener`)。

添加完 mixin 后，对象 `user` 将能够在访客登录时生成事件 `"login"`。另一个对象，例如 `calendar` 可能希望监听此类事件以便为登录的人加载日历。

或者，当一个菜单项被选中时，`menu` 可以生成 `"select"` 事件，其他对象可以分配处理程序以对该事件作出反应。诸如此类。

下面是代码：

```
let eventMixin = {
  /**
   * 订阅事件，用法：
   *   menu.on('select', function(item) { ... })
   */
  on(eventName, handler) {
    if (!this._eventHandlers) this._eventHandlers = {};
    if (!this._eventHandlers[eventName]) {
      this._eventHandlers[eventName] = [];
    }
    this._eventHandlers[eventName].push(handler);
  },

  /**
   * 取消订阅，用法：
   *   menu.off('select', handler)
   */
  off(eventName, handler) {
    let handlers = this._eventHandlers && this._eventHandlers[eventName];
    if (!handlers) return;
    for (let i = 0; i < handlers.length; i++) {
      if (handlers[i] === handler) {
        handlers.splice(i--, 1);
      }
    }
  },
}

/**
 * 生成具有给定名称和数据的事件
 *   this.trigger('select', data1, data2);
 */
trigger(eventName, ...args) {
  if (!this._eventHandlers || !this._eventHandlers[eventName]) {
    return; // 该事件名称没有对应的事件处理程序 (handler)
  }

  // 调用事件处理程序 (handler)
  this._eventHandlers[eventName].forEach(handler => handler.apply(this, args));
}
};
```

1. `.on(eventName, handler)` — 指定函数 `handler` 以在具有对应名称的事件发生时运行。从技术上讲，这儿有一个用于存储每个事件名称对应的处理程序（`handler`）的 `_eventHandlers` 属性，在这儿该属性就会将刚刚指定的这个 `handler` 添加到列表中。

2. `.off(eventName, handler)` — 从处理程序列表中删除指定的函数。

3. `.trigger(eventName, ...args)` — 生成事件：所有 `_eventHandlers[eventName]` 中的事件处理程序（`handler`）都被调用，并且 `...args` 会被作为参数传递给它们。

用法:

```
// 创建一个 class
class Menu {
  choose(value) {
    this.trigger("select", value);
  }
}
// 添加带有事件相关方法的 mixin
Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// 添加一个事件处理器 (handler)，在被选择时被调用:
menu.on("select", value => alert(`Value selected: ${value}`));

// 触发事件 => 运行上述的事件处理器 (handler) 并显示:
// 被选中的值: 123
menu.choose("123");
```

现在，如果我们希望任何代码对菜单选择作出反应，我们可以使用 `menu.on(...)` 进行监听。使用 `eventMixin` 可以轻松地将此类行为添加到我们想要的多个类中，并且不会影响继承链。

总结

Mixin — 是一个通用的面向对象编程术语：一个包含其他类的方法的类。

一些其它编程语言允许多重继承。**JavaScript** 不支持多重继承，但是可以通过将方法拷贝到原型中来实现 `mixin`。

我们可以使用 `mixin` 作为一种通过添加多种行为（例如上文中所提到的事件处理）来扩充类的方法。

如果 `Mixins` 意外覆盖了现有类的方法，那么它们可能会成为一个冲突点。因此，通常应该仔细考虑 `mixin` 的命名方法，以最大程度地降低发生这种冲突的可能性。

错误处理

错误处理，“try..catch”

不管你多么精通编程，有时我们的脚本总还是会出现错误。可能是因为我们的编写出错，或是与预期不同的用户输入，或是错误的服务端响应以及其他数千种原因。

通常，如果发生错误，脚本就会“死亡”（立即停止），并在控制台将错误打印出来。

但是有一种语法结构 `try..catch`，它使我们可以“捕获（catch）”错误，因此脚本可以执行更合理地操作，而不是死掉。

“try...catch” 语法

`try..catch` 结构由两部分组成：`try` 和 `catch`：

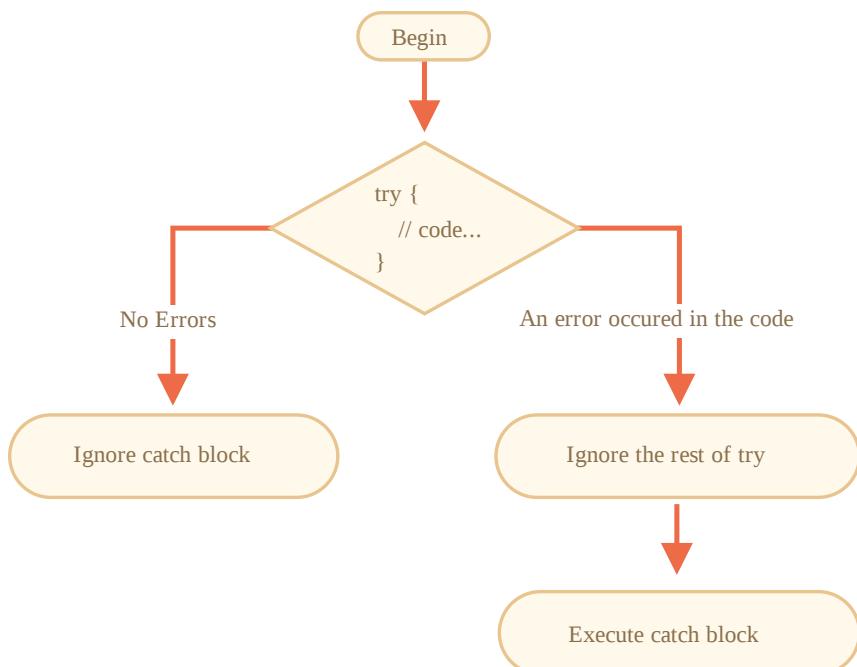
```

try {
    // 代码...
} catch (err) {
    // 错误捕获
}

```

它按照以下步骤执行:

- 首先，执行 `try { ... }` 中的代码。
- 如果这里没有错误，则忽略 `catch(err)`：执行到 `try` 的末尾并跳过 `catch` 继续执行。
- 如果这里出现错误，则 `try` 执行停止，控制流转向 `catch(err)` 的开头。变量 `err`（我们可以使用任何名称）将包含一个 `error` 对象，该对象包含了所发生事件的详细信息。



所以，`try {...}` 块内的错误不会杀死脚本 — 我们有机会在 `catch` 中处理它。

让我们来看一些例子。

- 没有 `error` 的例子：显示 `alert (1)` 和 `(2)`：

```

try {
    alert('Start of try runs'); // (1) <--
    // ...这里没有 error

    alert('End of try runs'); // (2) <--

} catch(err) {
    alert('Catch is ignored, because there are no errors'); // (3)
}

```

```
}
```

- 包含 `error` 的例子：显示 (1) 和 (3) 行的 `alert` 中的内容：

```
try {  
  
    alert('Start of try runs'); // (1) <--  
  
    lalala; // Error, 变量未定义!  
  
    alert('End of try (never reached)'); // (2)  
  
} catch(err) {  
  
    alert(`Error has occurred!`); // (3) <--  
  
}
```

⚠ `try..catch` 仅对运行时的 `error` 有效

要使得 `try..catch` 能工作，代码必须是可执行的。换句话说，它必须是有效的 JavaScript 代码。

如果代码包含语法错误，那么 `try..catch` 将无法正常工作，例如含有不匹配的花括号：

```
try {  
    {{{{{{{{{{{  
} } catch(e) {  
    alert("The engine can't understand this code, it's invalid");  
}
```

JavaScript 引擎首先会读取代码，然后运行它。在读取阶段发生的错误被称为“解析时间（parse-time）”错误，并且无法恢复（从该代码内部）。这是因为引擎无法理解该代码。

所以，`try..catch` 只能处理有效代码之出现的错误。这类错误被称为“运行时的错误（runtime errors）”，有时被称为“异常（exceptions）”。

⚠️ try..catch 同步工作

如果在“计划的（scheduled）”代码中发生异常，例如在 `setTimeout` 中，则 `try..catch` 不会捕获到异常：

```
try {
  setTimeout(function() {
    noSuchVariable; // 脚本将在这里停止运行
  }, 1000);
} catch (e) {
  alert( "won't work" );
}
```

因为 `try..catch` 包裹了计划要执行的函数，该函数本身要稍后才执行，这时引擎已经离开了 `try..catch` 结构。

为了捕获到计划的（scheduled）函数中的异常，那么 `try..catch` 必须在这个函数内：

```
setTimeout(function() {
  try {
    noSuchVariable; // try..catch 处理 error 了!
  } catch {
    alert( "error is caught here!" );
  }
}, 1000);
```

Error 对象

发生错误时，JavaScript 生成一个包含有关其详细信息的对象。然后将该对象作为参数传递给 `catch`：

```
try {
  // ...
} catch(err) { // <-- “error 对象”，也可以用其他参数名代替 err
  // ...
}
```

对于所有内建的 `error`，`error` 对象具有两个主要属性：

`name`

`Error` 名称。例如，对于一个未定义的变量，名称是 `"ReferenceError"`。

`message`

关于 `error` 的详细文字描述。

还有其他非标准的属性在大多数环境中可用。其中被最广泛使用和支持的是：

`stack`

当前的调用栈：用于调试目的的一个字符串，其中包含有关导致 `error` 的嵌套调用序列的信息。

例如：

```
try {
  lalala; // error, variable is not defined!
} catch(err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala is not defined
  alert(err.stack); // ReferenceError: lalala is not defined at (...call stack)

  // 也可以将一个 error 作为整体显示出来 as a whole
  // Error 信息被转换为像 "name: message" 这样的字符串
  alert(err); // ReferenceError: lalala is not defined
}
```

可选的“`catch`”绑定



A recent addition

This is a recent addition to the language. Old browsers may need polyfills.

如果我们不需要 `error` 的详细信息，`catch` 也可以忽略它：

```
try {
  // ...
} catch { // <-- 没有 (err)
  // ...
}
```

使用“`try...catch`”

让我们一起探究一下真实场景中 `try..catch` 的用例。

正如我们所知道的，JavaScript 支持 `JSON.parse(str)` 方法来解析 JSON 编码的值。

通常，它被用来解析从网络，从服务器或是从其他来源接收到的数据。

我们收到数据后，然后像下面这样调用 `JSON.parse`：

```
let json = '{"name":"John", "age": 30}'; // 来自服务器的数据

let user = JSON.parse(json); // 将文本表示转换成 JS 对象

// 现在 user 是一个解析自 json 字符串的有自己属性的对象
alert( user.name ); // John
alert( user.age ); // 30
```

你可以在 [JSON 方法, toJSON](#) 一章中找到更多关于 JSON 的详细内容。

如果 `json` 格式错误，`JSON.parse` 就会生成一个 `error`，因此脚本就会“死亡”。

我们对此满意吗？当然不！

如果这样做，当拿到的数据出了问题，那么访问者永远都不会知道原因（除非他们打开开发者控制台）。代码执行失败却没有提示信息，这真的是很糟糕的用户体验。

让我们用 `try..catch` 来处理这个 `error`:

```
let json = "{ bad json }";

try {

  let user = JSON.parse(json); // <-- 当出现一个 error 时...
  alert( user.name ); // 不工作

} catch (e) {
  // ...执行会跳转到这里并继续执行
  alert( "Our apologies, the data has errors, we'll try to request it one more time." );
  alert( e.name );
  alert( e.message );
}
```

在这儿，我们将 `catch` 块仅仅用于显示信息，但是我们可以做更多的事儿：发送一个新的网络请求，向访问者建议一个替代方案，将有关错误的信息发送给记录日志的设备，……。所有这些都比代码“死掉”好得多。

抛出我们自定义的 `error`

如果这个 `json` 在语法上是正确的，但是没有所必须的 `name` 属性该怎么办？

像这样：

```
let json = '{ "age": 30 }'; // 不完整的数据

try {

  let user = JSON.parse(json); // <-- 没有 error
  alert( user.name ); // 没有 name!

} catch (e) {
  alert( "doesn't execute" );
}
```

这里 `JSON.parse` 正常执行，但是缺少 `name` 属性对我们来说确实是个 `error`。

为了统一进行 `error` 处理，我们将使用 `throw` 操作符。

“Throw” 操作符

`throw` 操作符会生成一个 `error` 对象。

语法如下：

```
throw <error object>
```

技术上讲，我们可以将任何东西用作 `error` 对象。甚至可以时一个原始类型数据，例如数字或字符串，但最好使用对象，最好使用具有具有 `name` 和 `message` 属性的对象（某种程度上保持与内建 `error` 的兼容性）。

JavaScript 中有很多内建的标准 `error` 的构造器：`Error`，`SyntaxError`，`ReferenceError`，`TypeError` 等。我们也可以使用它们来创建 `error` 对象。

它们的语法是：

```
let error = new Error(message);
// 或
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

对于内建的 `error`（不是对于其他任何对象，仅仅是对于 `error`），`name` 属性刚好就是构造器的名字。`message` 则来自于参数（argument）。

例如：

```
let error = new Error("Things happen o_O");

alert(error.name); // Error
alert(error.message); // Things happen o_O
```

让我们来看看 `JSON.parse` 会生成什么样的 `error`：

```
try {
  JSON.parse("{ bad json o_O }");
} catch(e) {
  alert(e.name); // SyntaxError
  alert(e.message); // Unexpected token b in JSON at position 2
}
```

正如我们所看到的，那是一个 `SyntaxError`。

在我们的示例中，缺少 `name` 属性就是一个 `error`，因为用户必须有一个 `name`。

所以，让我们抛出这个 `error`。

```
let json = '{ "age": 30 }'; // 不完整的数据

try {
  let user = JSON.parse(json); // <-- 没有 error

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // (*)
  }

  alert( user.name );
}
```

```
    } catch(e) {
      alert( "JSON Error: " + e.message ); // JSON Error: Incomplete data: no name
    }
}
```

在 (*) 标记的这一行，`throw` 操作符生成了包含着我们所给定的 `message` 的 `SyntaxError`，与 JavaScript 自己生成的方式相同。`try` 的执行立即停止，控制流转向 `catch` 块。

现在，`catch` 成为了所有 `error` 处理的唯一场所：对于 `JSON.parse` 和其他情况都适用。

再次抛出（Rethrowing）

在上面的例子中，我们使用 `try..catch` 来处理不正确的数据。但是在 `try {...}` 块中是否可能发生 另一个预料之外的 `error`？例如编程错误（未定义变量）或其他错误，而不仅仅是这种“不正确的数据”。

例如：

```
let json = '{ "age": 30 }'; // 不完整的数据

try {
  user = JSON.parse(json); // <-- 忘记在 user 前放置 "let"

  // ...
} catch(err) {
  alert("JSON Error: " + err); // JSON Error: ReferenceError: user is not defined
  // (实际上并没有 JSON Error)
}
```

当然，一切皆有可能！程序员也会犯错。即使是被数百万人使用了几十年的开源项目中 — 也可能突然被发现了一个漏洞，并导致可怕的黑客入侵。

在我们的例子中，`try..catch` 旨在捕获“数据不正确”的 `error`。但是实际上，`catch` 会捕获到 所有 来自于 `try` 的 `error`。在这儿，它捕获到了一个预料之外的 `error`，但是仍然抛出的是同样的 “`JSON Error`” 信息。这是不正确的，并且也会使代码变得更难以调试。

幸运的是，我们可以通过其他方式找出我们捕获的是什么 `error`，例如通过它的 `name` 属性：

```
try {
  user = { /*...*/ };
} catch(e) {
  alert(e.name); // "ReferenceError" for accessing an undefined variable
}
```

规则很简单：

`catch` 应该只处理它知道的 `error`，并“抛出”所有其他 `error`。

“再次抛出（rethrowing）”技术可以被更详细地解释为：

1. `Catch` 捕获所有 `error`。
2. 在 `catch(err) {...}` 块中，我们对 `error` 对象 `err` 进行分析。

3. 如果我们不知道如何处理它，那我们就 `throw err`。

在下面的代码中，我们使用“再次抛出”，以达到在 `catch` 中只处理 `SyntaxError` 的目的：

```
let json = '{ "age": 30 }'; // 不完整的数据
try {
  let user = JSON.parse(json);
  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name");
  }
  blabla(); // 预料之外的 error
  alert( user.name );
} catch(e) {
  if (e.name == "SyntaxError") {
    alert( "JSON Error: " + e.message );
  } else {
    throw e; // 再次抛出 (*)
  }
}
```

如果 (*) 标记的这行 `catch` 块中的 `error` 从 `try..catch` 中“掉了出来”，那么它也可以被外部的 `try..catch` 结构（如果存在）捕获到，如果外部不存在这种结构，那么脚本就会被杀死。

所以，`catch` 块实际上只处理它知道该如何处理的 `error`，并“跳过”所有其他的 `error`。

下面这个示例演示了这种类型的 `error` 是如何被另外一级 `try..catch` 捕获的：

```
function readData() {
  let json = '{ "age": 30 }';

  try {
    // ...
    blabla(); // error!
  } catch (e) {
    // ...
    if (e.name != 'SyntaxError') {
      throw e; // 再次抛出 (不知道如何处理它)
    }
  }
}

try {
  readData();
} catch (e) {
  alert( "External catch got: " + e ); // 捕获了它!
}
```

上面这个例子中的 `readData` 只知道如何处理 `SyntaxError`，而外部的 `try..catch` 知道如何处理任意的 `error`。

try...catch...finally

等一下，以上并不是所有内容。

`try..catch` 结构可能还有一个代码子句（clause）：`finally`。

如果它存在，它在所有情况下都会被执行：

- `try` 之后，如果没有 `error`，
- `catch` 之后，如果没有 `error`。

该扩展语法如下所示：

```
try {  
    ... 尝试执行的代码 ...  
} catch(e) {  
    ... 处理 error ...  
} finally {  
    ... 总是会执行的代码 ...  
}
```

试试运行这段代码：

```
try {  
    alert( 'try' );  
    if (confirm('Make an error?')) BAD_CODE();  
} catch (e) {  
    alert( 'catch' );  
} finally {  
    alert( 'finally' );  
}
```

这段代码有两种执行方式：

1. 如果你对于“Make an error?”的回答是“Yes”，那么执行 `try -> catch -> finally`。
2. 如果你的回答是“No”，那么执行 `try -> finally`。

`finally` 子句（clause）通常用在：当我们开始做某事的时候，希望无论出现什么情况都要完成完成某个任务。

例如，我们想要测量一个斐波那契数字函数 `fib(n)` 执行所需要花费的时间。通常，我们可以在运行它之前开始测量，并在运行完成时结束测量。但是，如果在该函数调用期间出现 `error` 该怎么办？特别是，下面这段 `fib(n)` 的实现代码在遇到负数或非整数数字时会返回一个 `error`。

无论如何，`finally` 子句都是一个结束测量的好地方。

在这儿，`finally` 能够保证在两种情况下都能正确地测量时间 — 成功执行 `fib` 以及 `fib` 中出现 `error` 时：

```
let num = +prompt("Enter a positive integer number?", 35)

let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Must not be negative, and also an integer.");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

let start = Date.now();

try {
  result = fib(num);
} catch (e) {
  result = 0;
} finally {
  diff = Date.now() - start;
}

alert(result || "error occurred");

alert(`execution took ${diff}ms`);
```

你可以通过运行上面这段代码并在 `prompt` 弹窗中输入 `35` 来进行检查 — 代码运行正常，先执行 `try` 然后是 `finally`。如果你输入的是 `-1` — 将立即出现 `error`，执行将只花费 `0ms`。以上两种情况下的时间测量都正确地完成了。

换句话说，函数 `fib` 以 `return` 还是 `throw` 完成都无关紧要。在这两种情况下都会执行 `finally` 子句。

i 变量和 `try..catch..finally` 中的局部变量

请注意，上面代码中的 `result` 和 `diff` 变量都是在 `try..catch` 之前声明的。

否则，如果我们使用 `let` 在 `try` 块中声明变量，那么该变量将只在 `try` 块中可见。

i `finally` 和 `return`

`finally` 子句适用于 `try..catch` 的 **任何** 出口。这包括显式的 `return`。

在下面这个例子中，在 `try` 中有一个 `return`。在这种情况下，`finally` 会在控制转向外部代码前被执行。

```
function func() {  
  try {  
    return 1;  
  
  } catch (e) {  
    /* ... */  
  } finally {  
    alert('finally');  
  }  
}  
  
alert(func()); // 先执行 finally 中的 alert, 然后执行这个 alert
```

i `try..finally`

没有 `catch` 子句的 `try..finally` 结构也很有用。当我们不想在这儿处理 `error` (让它们 *fall through*)，但是需要确保我们启动的处理需要被完成。

```
function func() {  
  // 开始执行需要被完成的操作 (比如测量)  
  try {  
    // ...  
  } finally {  
    // 完成前面我们需要完成的那件事儿, 即使 try 中的执行失败了  
  }  
}
```

上面的代码中，由于没有 `catch`，所以 `try` 中的 `error` 总是会使代码执行跳转至函数 `func()` 外。但是，在跳出之前需要执行 `finally` 中的代码。

全局 `catch`

⚠ 环境特定

这个部分的内容并不是 JavaScript 核心的一部分。

设想一下，在 `try..catch` 结构外有一个致命的 `error`，然后脚本死亡了。这个 `error` 就像编程错误或其他可怕的事儿那样。

有什么办法可以用来应对这种情况吗？我们可能想要记录这个 `error`，并向用户显示某些内容（通常用户看不到错误信息）等。

规范中没有相关内容，但是代码的执行环境一般会提供这种机制，因为它确实很有用。例如，Node.js 有 `process.on("uncaughtException")` [。](#) 在浏览器中，我们可以将一个函数赋值给特殊的 `window.onerror` [属性](#)，该函数将在发生未捕获的 `error` 时执行。

语法如下：

```
window.onerror = function(message, url, line, col, error) {  
    // ...  
};
```

message

Error 信息。

url

发生 `error` 的脚本的 URL。

line , col

发生 `error` 处的代码的行号和列号。

error

Error 对象。

例如：

```
<script>  
    window.onerror = function(message, url, line, col, error) {  
        alert(`#${message}\n At ${line}:${col} of ${url}`);  
    };  
  
    function readData() {  
        badFunc(); // 啊，出问题了！  
    }  
  
    readData();  
</script>
```

全局错误处理程序 `window.onerror` 的作用通常不是恢复脚本的执行 — 如果发生编程错误，那几乎是不可能的，它的作用是将错误信息发送给开发者。

也有针对这种情况提供错误日志的 Web 服务，例如 <https://errorception.com> [或](#) <http://www.muscula.com> [。](#)

它们会像这样运行：

1. 我们注册该服务，并拿到一段 JS 代码（或脚本的 URL），然后插入到页面中。
2. 该 JS 脚本设置了自定义的 `window.onerror` 函数。
3. 当发生 `error` 时，它会发送一个此 `error` 相关的网络请求到服务提供方。
4. 我们可以登录到服务方的 Web 界面来查看这些 `error`。

总结

`try..catch` 结构允许我们处理执行过程中出现的 `error`。从字面上看，它允许“尝试”运行代码并“捕获”其中可能发生的错误。

语法如下：

```
try {
  // 执行此处代码
} catch(err) {
  // 如果发生错误，跳转至此处
  // err 是一个 error 对象
} finally {
  // 无论怎样都会在 try/catch 之后执行
}
```

这儿可能会没有 `catch` 部分或者没有 `finally`，所以 `try..catch` 或 `try..finally` 都是可用的。

`Error` 对象包含下列属性：

- `message` — 人类可读的 `error` 信息。
- `name` — 具有 `error` 名称的字符串（`Error` 构造器的名称）。
- `stack`（没有标准，但得到了很好的支持）— `Error` 发生时的调用栈。

如果我们不需要 `error` 对象，我们可以通过使用 `catch {` 而不是 `catch(err) {` 来省略它。

我们也可以使用 `throw` 操作符来生成自定义的 `error`。从技术上讲，`throw` 的参数可以是任何东西，但通常是继承自内建的 `Error` 类的 `error` 对象。下一章我们会详细介绍扩展 `error`。

再次抛出（`rethrowing`）是一种错误处理的重要模式：`catch` 块通常期望并知道如何处理特定的 `error` 类型，因此它应该再次抛出它不知道的 `error`。

即使我们没有 `try..catch`，大多数执行环境也允许我们设置“全局”错误处理程序来捕获“掉出（`fall out`）”的 `error`。在浏览器中，就是 `window.onerror`。

自定义 `Error`, 扩展 `Error`

当我们在开发某些东西时，经常会需要我们自己的 `error` 类来反映在我们的任务中可能出错的特定任务。对于网络操作中的 `error`，我们需要 `HttpError`，对于数据库操作中的 `error`，我们需要 `DbError`，对于搜索操作中的 `error`，我们需要 `NotFoundError`，等等。

我们自定义的 `error` 应该支持基本的 `error` 的属性，例如 `message`，`name`，并且最好还有 `stack`。但是它们也可能会有其他属于它们自己的属性，例如，`HttpError` 对象可能会有一个 `statusCode` 属性，属性值可能为 `404`、`403` 或 `500` 等。

JavaScript 允许将 `throw` 与任何参数一起使用，所以从技术上讲，我们自定义的 `error` 不需要从 `Error` 中继承。但是，如果我们继承，那么就可以使用 `obj instanceof Error` 来识别 `error` 对象。因此，最好继承它。

随着虽开发的应用程序的增长，我们自己的 `error` 自然会形成形成一个层次结构（`hierarchy`）。例如，`HttpTimeoutError` 可能继承自 `HttpError`，等等。

扩展 Error

例如，让我们考虑一个函数 `readUser(json)`，该函数应该读取带有用户数据的 JSON。

这里是一个可用的 `json` 的例子：

```
let json = `{"name": "John", "age": 30}`;
```

在函数内部，我们将使用 `JSON.parse`。如果它接收到格式不正确的 `json`，就会抛出 `SyntaxError`。但是，即使 `json` 在语法上是正确的，也不意味着该数据是有效的用户数据，对吧？因为它可能丢失了某些必要的数据。例如，对用户来说，必不可少的是 `name` 和 `age` 属性。

我们的函数 `readUser(json)` 不仅会读取 JSON，还会检查（“验证”）数据。如果没有所必须的字段，或者（字段的）格式错误，那么就会出现一个 `error`。并且这些并不是 `SyntaxError`，因为这些数据在语法上是正确的，这些是另一种错误。我们称之为 `ValidationException`，并为之创建一个类。这种类型的错误也应该包含有关违规字段的信息。

我们的 `ValidationException` 类应该继承自内建的 `Error` 类。

`Error` 类是内建的，但这是其近似代码，所以我们可以了解我们要扩展的内容：

```
// JavaScript 自身定义的内建的 Error 类的“伪代码”
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (不同的内建 error 类有不同的名字)
    this.stack = <call stack>; // 非标准的，但大多数环境都支持它
  }
}
```

现在让我们从其中继承 `ValidationException`，并尝试进行运行：

```
class ValidationException extends Error {
  constructor(message) {
    super(message); // (1)
    this.name = "ValidationException"; // (2)
  }
}

function test() {
  throw new ValidationException("Whoops!");
}

try {
  test();
} catch(err) {
  alert(err.message); // Whoops!
  alert(err.name); // ValidationException
  alert(err.stack); // 一个嵌套调用的列表，每个调用都有对应的行号
}
```

请注意：在 (1) 行中我们调用了父类的 `constructor`。JavaScript 要求我们在子类的 `constructor` 中调用 `super`，所以这是必须的。父类的 `constructor` 设置了 `message` 属性。

父类的 `constructor` 还将 `name` 属性的值设置为了 `"Error"`，所以在 (2) 行中，我们将其重置为了右边的值。

让我们尝试在 `readUser(json)` 中使用它吧：

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

// 用法
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("No field: age");
  }
  if (!user.name) {
    throw new ValidationError("No field: name");
  }

  return user;
}

// try..catch 的工作示例

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No field: name
  } else if (err instanceof SyntaxError) { // (*)
    alert("JSON Syntax Error: " + err.message);
  } else {
    throw err; // 未知的 error，再次抛出 (**)
  }
}
```

上面代码中的 `try..catch` 块既处理我们的 `ValidationError` 又处理来自 `JSON.parse` 的内建 `SyntaxError`。

请看一下我们是如何使用 `instanceof` 来检查 (*) 行中的特定错误类型的。

我们也可以看看 `err.name`，像这样：

```
// ...
// instead of (err instanceof SyntaxError)
} else if (err.name == "SyntaxError") { // (*)
// ...
```

使用 `instanceof` 的版本要好得多，因为将来我们会对 `ValidationError` 进行扩展，创建它的子类型，例如 `PropertyRequiredError`。而 `instanceof` 检查对于新的继承类也适用。所以这是面向未来的做法。

还有一点很重要，在 `catch` 遇到了未知的错误，它会在 `(**)` 行将该错误再次抛出。`catch` 块只知道如何处理 `validation` 错误和语法错误，而其他错误（由于代码中的错字或其他未知的错误）应该被扔出（`fall through`）。

深入继承

`ValidationError` 类是非常通用的。很多东西都可能出错。对象的属性可能缺失或者属性可能有格式错误（例如 `age` 属性的值为一个字符串）。让我们针对缺少属性的错误来制作一个更具体的 `PropertyRequiredError` 类。它将携带有关缺少的属性的相关信息。

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.name = "PropertyRequiredError";
    this.property = property;
  }
}

// 用法
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError("age");
  }
  if (!user.name) {
    throw new PropertyRequiredError("name");
  }

  return user;
}

// try..catch 的工作示例

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No property: name
    alert(err.name); // PropertyRequiredError
    alert(err.property); // name
  } else if (err instanceof SyntaxError) {
    alert("JSON Syntax Error: " + err.message);
  } else {
    throw err; // 为止 error，将其再次抛出
  }
}
```

```
}
```

这个新的类 `PropertyRequiredError` 使用起来很简单：我们只需要传递属性名：`new PropertyRequiredError(property)`。人类可读的 `message` 是由 `constructor` 生成的。

请注意，在 `PropertyRequiredError` `constructor` 中的 `this.name` 是通过手动重新赋值的。这可能会变得有些乏味 — 在每个自定义 `error` 类中都要进行 `this.name = <class name>` 赋值操作。我们可以通过创建自己的“基础错误（basic error）”类来避免这种情况，该类进行了 `this.name = this.constructor.name` 赋值。然后让所有我们自定义的 `error` 都从这个“基础错误”类进行继承。

让我们称之为 `MyError`。

这是带有 `MyError` 以及其他自定义的 `error` 类的代码，已进行简化：

```
class MyError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name;
  }
}

class ValidationError extends MyError {}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.property = property;
  }
}

// name 是对的
alert( new PropertyRequiredError("field").name ); // PropertyRequiredError
```

现在自定义的 `error` 短了很多，特别是 `ValidationError`，因为我们摆脱了 `constructor` 中的 `"this.name = ..."` 这一行。

包装异常

在上面代码中的函数 `readUser` 的目的就是“读取用户数据”。在这个过程中可能会出现不同类型的 `error`。目前我们有了 `SyntaxError` 和 `ValidationError`，但是将来，函数 `readUser` 可能会不断壮大，并可能会产生其他类型的 `error`。

调用 `readUser` 的代码应该处理这些 `error`。现在它在 `catch` 块中使用了多个 `if` 语句来检查 `error` 类，处理已知的 `error`，并再次抛出未知的 `error`。

该方案是这样的：

```
try {
  ...
  readUser() // 潜在的 error 源
  ...
} catch (err) {
```

```
if (err instanceof ValidationError) {
    // 处理 validation error
} else if (err instanceof SyntaxError) {
    // 处理 syntax error
} else {
    throw err; // 未知 error, 再次抛出它
}
```

在上面的代码中，我们可以看到两种类型的 `error`，但是可以有更多。

如果 `readUser` 函数会产生多种 `error`，那么我们应该问问自己：我们是否真的想每次都一一检查所有的 `error` 类型？

通常答案是“*No*”：我们希望能够“比它高一个级别”。我们只想知道这里是否是“数据读取异常”——为什么发生了这样的 `error` 通常是无关紧要的（`error` 信息描述了它）。或者，如果我们有一种方法能够获取 `error` 的详细信息那就更好了，但前提是我们需要。

我们所描述的这项技术被称为“包装异常”。

1. 我们将创建一个新的类 `ReadError` 来表示一般的“数据读取” `error`。
2. 函数 `readUser` 将捕获内部发生的数据读取 `error`，例如 `ValidationError` 和 `SyntaxError`，并生成一个 `ReadError` 来进行替代。
3. 对象 `ReadError` 会把对原始 `error` 的引用保存在其 `cause` 属性中。

之后，调用 `readUser` 的代码只需要检查 `ReadError`，而不必检查每种数据读取 `error`。并且，如果需要更多 `error` 细节，那么可以检查 `readUser` 的 `cause` 属性。

下面的代码定义了 `ReadError`，并在 `readUser` 和 `try..catch` 中演示了其用法：

```
class ReadError extends Error {
    constructor(message, cause) {
        super(message);
        this.cause = cause;
        this.name = 'ReadError';
    }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ... */ }

function validateUser(user) {
    if (!user.age) {
        throw new PropertyRequiredError("age");
    }

    if (!user.name) {
        throw new PropertyRequiredError("name");
    }
}

function readUser(json) {
    let user;

    try {
        user = JSON.parse(json);
    } catch (err) {
```

```

    if (err instanceof SyntaxError) {
      throw new ReadError("Syntax Error", err);
    } else {
      throw err;
    }
  }

  try {
    validateUser(user);
  } catch (err) {
    if (err instanceof ValidationError) {
      throw new ReadError("Validation Error", err);
    } else {
      throw err;
    }
  }
}

try {
  readUser('{bad json}');
} catch (e) {
  if (e instanceof ReadError) {
    alert(e);
    // Original error: SyntaxError: Unexpected token b in JSON at position 1
    alert("Original error: " + e.cause);
  } else {
    throw e;
  }
}

```

在上面的代码中，`readUser` 正如所描述的那样正常工作 — 捕获语法和验证（validation）错误，并抛出 `ReadError`（对于未知错误将照常再次抛出）。

所以外部代码检查 `instanceof ReadError`，并且它的确是。不必列出所有可能的 `error` 类型。

这种方法被称为“包装异常（wrapping exceptions）”，因为我们将“低级别”的异常“包装”到了更抽象的 `ReadError` 中。它被广泛应用于面向对象的编程中。

总结

- 我们可以正常地从 `Error` 和其他内建的 `error` 类中进行继承，。我们只需要注意 `name` 属性以及不要忘了调用 `super`。
- 我们可以使用 `instanceof` 来检查特定的 `error`。但有时我们有来自第三方库的 `error` 对象，并且在这儿没有简单的方法来获取它的类。那么可以将 `name` 属性用于这一类的检查。
- 包装异常是一项广泛应用的技术：用于处理低级别异常并创建高级别 `error` 而不是各种低级别 `error` 的函数。在上面的示例中，低级别异常有时会成为该对象的属性，例如 `err.cause`，但这不是严格要求的。

Promise, `async/await`

简介：回调

⚠ 我们在这里的示例中使用了浏览器方法

为了演示回调、`promise` 和其他抽象概念的使用，我们将使用一些浏览器方法：具体地说，是加载脚本和执行简单的文档操作的方法。

如果你不熟悉这些方法，并且对它们在这些示例中的用法感到疑惑，那么你可能需要阅读本教程 [下一部分](#) 中的几章。

但是，我们会尽全力使讲解变得更加清晰。在这儿不会有浏览器方面的真正复杂的东西。

JavaScript 主机（host）环境提供了许多函数，这些函数允许我们计划 异步 行为（action）。换句话说，我们现在开始执行的行为，但它们会在稍后完成。

例如，`setTimeout` 函数就是一个这样的函数。

这儿有一些实际中的异步行为的示例，例如加载脚本和模块（我们将在后面的章节中介绍）。

让我们看一下函数 `loadScript(src)`，该函数使用给定的 `src` 加载脚本：

```
function loadScript(src) {
    // 创建一个 <script> 标签，并将其附加到页面
    // 这将使得具有给定 src 的脚本开始加载，并在加载完成后运行
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
}
```

它将带有给定 `src` 的新动态创建的标签 `<script src="...">` 附加到文档中。浏览器将自动开始加载它，并在加载完成后执行。

我们可以像这样使用这个函数：

```
// 在给定路径下加载并执行脚本
loadScript('/my/script.js');
```

脚本是“异步”调用的，因为它从现在开始加载，但是在加载函数执行完成后才运行。

如果在 `loadScript(...)` 下面有任何其他代码，它们不会等到脚本加载完成才执行。

```
loadScript('/my/script.js');
// loadScript 下面的代码
// 不会等到脚本加载完成才执行
// ...
```

假设我们需要在新脚本加载后立即使用它。它声明了新函数，我们想运行它们。

但如果我们在 `loadScript(...)` 调用后立即执行此操作，这将不会有效。

```
loadScript('/my/script.js'); // 这个脚本有 "function newFunction() {...}"
newFunction(); // 没有这个函数!
```

自然情况下，浏览器可能没有时间加载脚本。到目前为止，`loadScript` 函数并没有提供跟踪加载完成的方法。脚本加载并最终运行，仅此而已。但我们希望了解脚本何时加载完成，以使用其中的新函数和变量。

让我们添加一个 `callback` 函数作为 `loadScript` 的第二个参数，该函数应在脚本加载完成时执行：

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(script);

  document.head.append(script);
}
```

现在，如果我们想调用该脚本中的新函数，我们应该将其写在回调函数中：

```
loadScript('/my/script.js', function() {
  // 在脚本加载完成后，回调函数才会执行
  newFunction(); // 现在它工作了
  ...
});
```

这是我们的想法：第二个参数是一个函数（通常是匿名函数），该函数会在行为（action）完成时运行。

这是一个带有真实脚本的可运行的示例：

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
  alert(`Cool, the script ${script.src} is loaded`);
  alert(_); // 所加载的脚本中声明的函数
});
```

这被称为“基于回调”的异步编程风格。异步执行某项功能的函数应该提供一个 `callback` 参数用于在相应事件完成时调用。（译注：上面这个例子中的相应事件是指脚本加载）

这里我们在 `loadScript` 中就是这么做的，但当然这是一种通用方法。

在回调中回调

我们如何依次加载两个脚本：第一个，然后是第二个？

自然的解决方案是将第二个 `loadScript` 调用放入回调中，如下所示：

```
loadScript('/my/script.js', function(script) {
    alert(`Cool, the ${script.src} is loaded, let's load one more`);

    loadScript('/my/script2.js', function(script) {
        alert(`Cool, the second script is loaded`);
    });
});
```

在外部 `loadScript` 执行完成时，内部回调就会被回调。

如果我们还想要一个脚本呢？

```
loadScript('/my/script.js', function(script) {
    loadScript('/my/script2.js', function(script) {
        loadScript('/my/script3.js', function(script) {
            // ...加载完所有脚本后继续
        });
    });
});
```

因此，每一个新行为（action）都在回调内部。这对于几个行为来说还好，但对于许多行为来说就不好了，所以我们很快就会看到其他变体。

处理 Error

在上述示例中，我们并没有考虑出现 `error` 的情况。如果脚本加载失败怎么办？我们的回调应该能够对此作出反应。

这是 `loadScript` 的改进版本，可以跟踪加载错误：

```
function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => callback(null, script);
    script.onerror = () => callback(new Error(`Script load error for ${src}`));

    document.head.append(script);
}
```

加载成功时，它会调用 `callback(null, script)`，否则调用 `callback(error)`。

用法：

```
loadScript('/my/script.js', function(error, script) {
    if (error) {
```

```
// 处理 error
} else {
    // 脚本加载成功
}
});
```

再次强调，我们在 `loadScript` 中所使用的方案其实很普遍。它被称为“Error 优先回调（error-first callback）”风格。

约定是：

1. `callback` 的第一个参数是为 `error` 而保留的。一旦出现 `error`, `callback(err)` 就会被调用。
2. 第二个参数（和下一个参数，如果需要的话）用于成功的结果。此时 `callback(null, result1, result2...)` 就会被调用。

因此，单一的 `callback` 函数可以同时具有报告 `error` 和传递返回结果的作用。

厄运金字塔

乍一看，这是一种可行的异步编程方式。的确如此，对于一个或两个嵌套的调用看起来还不错。但对于一个接一个的多个异步行为，代码将会变成这样：

```
loadScript('1.js', function(error, script) {

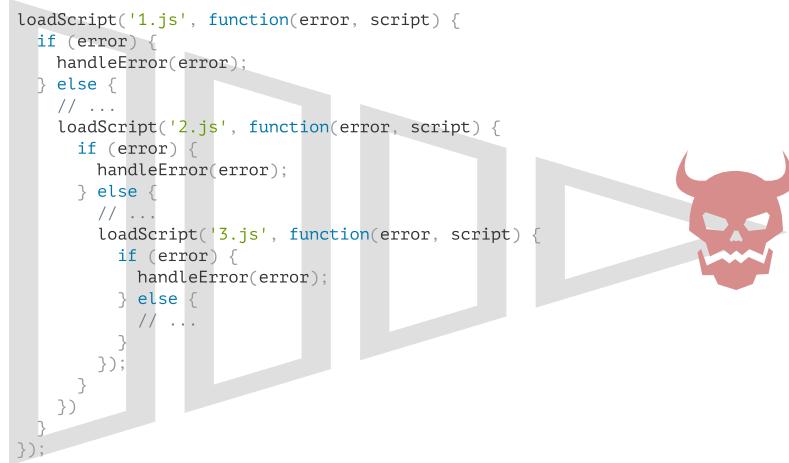
    if (error) {
        handleError(error);
    } else {
        // ...
        loadScript('2.js', function(error, script) {
            if (error) {
                handleError(error);
            } else {
                // ...
                loadScript('3.js', function(error, script) {
                    if (error) {
                        handleError(error);
                    } else {
                        // ...加载完所有脚本后继续 (*)
                    }
                });
            }
        });
    }
});
```

在上面这段代码中：

1. 我们加载 `1.js`，如果没有发生错误。
2. 我们加载 `2.js`，如果没有发生错误。
3. 我们加载 `3.js`，如果没有发生错误 — 做其他操作 (*)。

如果调用嵌套的增加，代码层次变得更深，维护难度也随之增加，尤其是我们使用的是可能包含了很多循环和条件语句的真实代码，而不是例子中的`...`。

有时这些被称为“回调地狱”或“厄运金字塔”。



嵌套调用的“金字塔”随着每个异步行为会向右增长。很快它就失控了。

所以这种编码方式不是很好。

我们可以通过使每个行为都成为一个独立的函数来尝试减轻这种问题，如下所示：

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...加载完所有脚本后继续 (*)
  }
};
```

看到了吗？它的作用相同，但是没有深层的嵌套了，因为我们将每个行为都编写成了一个独立的顶层函数。

它可以工作，但是代码看起来就像是一个被撕裂的表格。你可能已经注意到了，它的可读性很差，在阅读时你需要在各个代码块之间跳转。这很不方便，特别是如果读者对代码不熟悉，他们甚至不知道应该跳转到什么地方。

此外，名为 `step*` 的函数都是一次性使用的，创建它们就是为了避免“厄运金字塔”。没有人会在行为链之外重用它们。因此，这里的命名空间有点混乱。

我们希望还有更好的方法。

幸运的是，有其他方法可以避免此类金字塔。最好的方法之一就是“promise”，我们将在下一章中介绍它。

Promise

想象一下，你是一位顶尖歌手，粉丝没日没夜地询问你下个单曲什么时候发。

为了从中解放，你承诺（`promise`）会在单曲发布的第一时间发给他们。你给了粉丝们一个列表。他们可以在上面填写他们的电子邮件地址，以便当歌曲发布后，让所有订阅了的人能够立即收到。即便遇到不测，例如录音室发生了火灾，以致你无法发布新歌，他们也能及时收到相关通知。

每个人都很开心：你不会被任何人催促，粉丝们也不用担心错过单曲发行。

这是我们在编程中经常遇到的事儿与真实生活的类比：

1. “生产者代码（`producing code`）”会做一些事儿，并且会需要一些时间。例如，通过网络加载数据的代码。它就像一位“歌手”。
2. “消费者代码（`consuming code`）”想要在“生产者代码”完成工作的第一时间就能获得其工作成果。许多函数可能都需要这个结果。这些就是“粉丝”。
3. **Promise** 是将“生产者代码”和“消费者代码”连接在一起的一个特殊的 `JavaScript` 对象。用我们的类比来说：这就是就像是“订阅列表”。“生产者代码”花费它所需的任意长度时间来产出所承诺的结果，而“`promise`”将在它（译注：指的是“生产者代码”，也就是下文所说的 `executor`）准备好了，将结果向所有订阅了的代码开放。

这种类比并不十分准确，因为 `JavaScript` 的 `promise` 比简单的订阅列表更加复杂：它们还拥有其他的功能和局限性。但以此开始挺好的。

`Promise` 对象的构造器（`constructor`）语法如下：

```
let promise = new Promise(function(resolve, reject) {
  // executor (生产者代码, "歌手")
});
```

传递给 `new Promise` 的函数被称为 `executor`。当 `new Promise` 被创建，`executor` 会自动运行。它包含最终应产出结果的生产者代码。按照上面的类比：`executor` 就是“歌手”。

它的参数 `resolve` 和 `reject` 是由 `JavaScript` 自身提供的回调。我们的代码仅在 `executor` 的内部。

当 `executor` 获得了结果，无论是早还是晚都没关系，它应该调用以下回调之一：

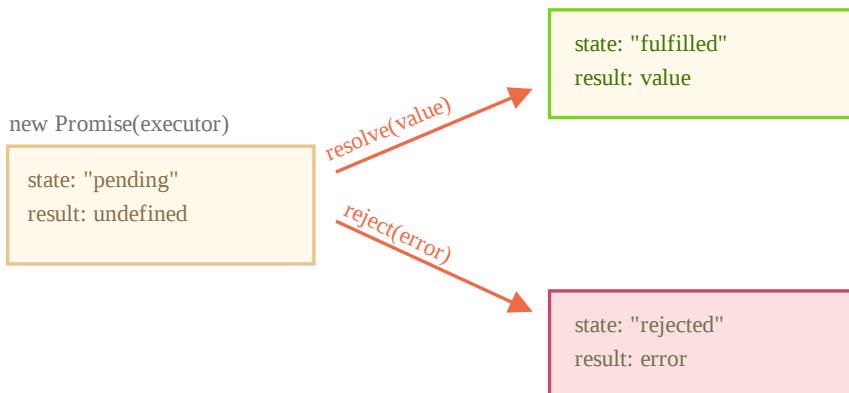
- `resolve(value)` — 如果任务成功完成并带有结果 `value`。
- `reject(error)` — 如果出现了 `error`，`error` 即为 `error` 对象。

所以总结一下就是：`executor` 会自动运行并尝试执行一项工作。尝试结束后，如果成功则调用 `resolve`，如果出现 `error` 则调用 `reject`。

由 `new Promise` 构造器返回的 `promise` 对象具有以下内部属性：

- `state` — 最初是 `"pending"`，然后在 `resolve` 被调用时变为 `"fulfilled"`，或者在 `reject` 被调用时变为 `"rejected"`。
- `result` — 最初是 `undefined`，然后在 `resolve(value)` 被调用时变为 `value`，或者在 `reject(error)` 被调用时变为 `error`。

所以，`executor` 最终将 `promise` 移至以下状态之一：



稍后我们将看到“粉丝”如何订阅这些更改。

下面是一个 `promise` 构造器和一个简单的 `executor` 函数，该 `executor` 函数具有包含时间（即 `setTimeout`）的“生产者代码”：

```
let promise = new Promise(function(resolve, reject) {
  // 当 promise 被构造完成时，自动执行此函数

  // 1 秒后发出工作已经被完成的信号，并带有结果 "done"
  setTimeout(() => resolve("done"), 1000);
});
```

通过运行上面的代码，我们可以看到两件事儿：

1. `executor` 被自动且立即调用（通过 `new Promise`）。
2. `executor` 接受两个参数：`resolve` 和 `reject`。这些函数由 JavaScript 引擎预先定义，因此我们不需要创建它们。我们只需要在准备好（译注：指的是 `executor` 准备好）时调用其中之一即可。

经过 1 秒的“处理”后，`executor` 调用 `resolve("done")` 来产生结果。这将改变 `promise` 对象的状态：



这是一个成功完成任务的例子，一个“成功实现了的诺言”。

现在的则是一个 executor 以 error 拒绝 promise 的示例:

```
let promise = new Promise(function(resolve, reject) {
  // 1 秒后发出工作已经被完成的信号，并带有 error
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

对 `reject(...)` 的调用将 promise 对象的状态移至 "rejected" :

```
new Promise(executor)
```

state: "pending"
result: undefined

`reject(error)`

state: "rejected"
result: error

总而言之，executor 应该执行一项工作（通常是需要花费一些时间的事儿），然后调用 `resolve` 或 `reject` 来改变对应的 promise 对象的状态。

与最初的“pending”promise 相反，一个 resolved 或 rejected 的 promise 都会被称为“settled”。

i 这儿只能有一个结果或一个 error

executor 只能调用一个 `resolve` 或一个 `reject`。任何状态的更改都是最终的。

所有其他的再对 `resolve` 和 `reject` 的调用都会被忽略:

```
let promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error("...")); // 被忽略
  setTimeout(() => resolve("...")); // 被忽略
});
```

这儿的宗旨是，一个被 executor 完成的工作只能有一个结果或一个 error。

并且，`resolve/reject` 只需要一个参数（或不包含任何参数），并且将忽略额外的参数。

i 以 Error 对象 reject

如果什么东西出了问题，executor 应该调用 `reject`。这可以使用任何类型的参数来完成（就像 `resolve` 一样）。但是建议使用 `Error` 对象（或继承自 `Error` 的对象）。这样做的理由很快就会显而易见。

① `Resolve/reject` 可以立即进行

实际上，`executor` 通常是异步执行某些操作，并在一段时间后调用 `resolve/reject`，但这不是必须的。我们还可以立即调用 `resolve` 或 `reject`，就像这样：

```
let promise = new Promise(function(resolve, reject) {
  // 不花时间去做这项工作
  resolve(123); // 立即给出结果: 123
});
```

例如，当我们开始做一个任务时，但随后看到一切都已经完成并已被缓存时，可能就会发生这种情况。

这挺好。我们立即就有了一个 `resolved` 的 `promise`。

① `state` 和 `result` 都是内部的

`Promise` 对象的 `state` 和 `result` 属性都是内部的。我们无法直接访问它们。但我们可以对它们使用 `.then/.catch/.finally` 方法。我们在下面对这些方法进行了描述。

消费者：`then`, `catch`, `finally`

`Promise` 对象充当的是 `executor`（“生产者代码”或“歌手”）和消费函数（“粉丝”）之间的连接，后者将接收结果或 `error`。可以通过使用 `.then`、`.catch` 和 `.finally` 方法为消费函数进行注册。

`then`

最重要最基础的一个就是 `.then`。

语法如下：

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

`.then` 的第一个参数是一个函数，该函数将在 `promise resolved` 后运行并接收结果。

`.then` 的第二个参数也是一个函数，该函数将在 `promise rejected` 后运行并接收 `error`。

例如，以下是对成功 `resolved` 的 `promise` 做出的反应：

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve 运行 .then 中的第一个函数
promise.then(
  result => alert(result), // 1 秒后显示 "done!"
  error => alert(error) // 不运行
);
```

第一个函数被运行了。

在 `reject` 的情况下，运行第二个：

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject 运行 .then 中的第二个函数
promise.then(
  result => alert(result), // 不运行
  error => alert(error) // 1 秒后显示 "Error: Whoops!"
);
```

如果我们只对成功完成的情况感兴趣，那么我们可以只为 `.then` 提供一个函数参数：

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
};

promise.then(alert); // 1 秒后显示 "done!"
```

catch

如果我们只对 `error` 感兴趣，那么我们可以使用 `null` 作为第一个参数：`.then(null, errorHandlingFunction)`。或者我们也可以使用 `.catch(errorHandlingFunction)`，其实是一样的：

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
};

// .catch(f) 与 promise.then(null, f) 一样
promise.catch(alert); // 1 秒后显示 "Error: Whoops!"
```

`.catch(f)` 调用是 `.then(null, f)` 的完全的模拟，它只是一个简写形式。

finally

就像常规 `try {...} catch {...}` 中的 `finally` 子句一样，`promise` 中也有 `finally`。

`.finally(f)` 调用与 `.then(f, f)` 类似，在某种意义上，`f` 总是在 `promise` 被 `settled` 时运行：即 `promise` 被 `resolve` 或 `reject`。

`finally` 是执行清理（cleanup）的很好的处理程序（handler），例如无论结果如何，都停止使用不再需要的加载指示符（indicator）。

像这样：

```
new Promise((resolve, reject) => {
  /* 做一些需要时间的事儿，然后调用 resolve/reject */
})
  // 在 promise 被 settled 时运行，无论成功与否
```

```
.finally(() => stop loading indicator)
.then(result => show result, err => show error)
```

不过，它并不是 `then(f, f)` 的别名。它们之间有几个重要的区别：

1. `finally` 处理程序（handler）没有参数。在 `finally` 中，我们不知道 promise 是否成功。没关系，因为我们的任务通常是执行“常规”的定稿程序（finalizing procedures）。
2. `finally` 处理程序将结果和 error 传递给下一个处理程序。

例如，在这儿结果被从 `finally` 传递给了 `then`：

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
  .finally(() => alert("Promise ready"))
  .then(result => alert(result)); // <-- .then 对结果进行处理
```

在这儿，promise 中有一个 error，这个 error 被从 `finally` 传递给了 `catch`：

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Promise ready"))
  .catch(err => alert(err)); // <-- .catch 对 error 对象进行处理
```

这非常方便，因为 `finally` 并不是意味着要处理 promise 的结果。所以它将结果传递了下去。

在下一章中，我们将详细讨论 promise 链以及处理程序（handler）之间的结果传递。

3. 最后，但并非最不重要的一点是，`.finally(f)` 是比 `.then(f, f)` 更为方便的语法：无需重复函数 `f`。

① 在 `settled` 的 promise 上，`then` 会立即运行

如果 promise 为 `pending` 状态，`.then/catch/finally` 处理程序（handler）将等待它。否则，如果 promise 已经是 `settled` 状态，它们就会立即执行：

```
// the promise becomes resolved immediately upon creation
let promise = new Promise(resolve => resolve("done!"));

promise.then(alert); // done! (现在显示)
```

请注意，这和现实生活中的类比是不同的，并且比现实生活中的“订阅列表”方案强大得多。如果歌手已经发布了他们的单曲，然后某个人在订阅列表上进行了注册，则他们很可能不会收到该单曲。实际生活中的订阅必须在活动开始之前进行。

Promise 则更加灵活。我们可以随时添加处理程序（handler）：如果结果已经在了，我们的处理程序便会立即获得这个结果。

接下来，让我们看一下关于 promise 如何帮助我们编写异步代码的更多实际示例。

示例：loadScript

我们从上一章获得了用于加载脚本的 `loadScript` 函数。

这是基于回调函数的变体，记住它：

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${src}`));

  document.head.append(script);
}
```

让我们用 promise 重写它。

新函数 `loadScript` 将不需要回调。取而代之的是，它将创建并返回一个在加载完成时解析 (`resolve`) 的 promise 对象。外部代码可以使用 `.then` 向其添加处理程序（订阅函数）：

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Script load error for ${src}`));

    document.head.append(script);
  });
}
```

用法：

```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`#${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Another handler...'));
```

我们立刻就能发现 promise 相较于基于回调的模式的一些好处：

Promises

Promises 允许我们按照自然顺序进行编码。首先，我们运行 `loadScript` 和 `.then` 来处理结果。

我们可以根据需要，在 `promise` 上多次调用 `.then`。每次调用，我们都会在“订阅列表”中添加一个新的“分析”，一个新的订阅函数。在下一章将对此内容进行详细介绍：[Promise 链](#)。

Callbacks

在调用 `loadScript(script, callback)` 时，在我们处理的地方（`disposal`）必须有一个 `callback` 函数。换句话说，在调用 `loadScript` 之前，我们必须知道如何处理结果。

只能有一个回调。

因此，`promise` 为我们提供了更好的代码流和灵活性。但其实还有更多相关内容。我们将在下一章看到。

Promise 链

我们回顾一下 [简介：回调](#) 一章中提到的问题：我们有一系列的异步任务要一个接一个地执行 — 例如，加载脚本。我们如何写出更好的代码呢？

`Promise` 提供了一些方案来做到这一点。

在本章中，我们将一起学习 `promise` 链。

它看起来就像这样：

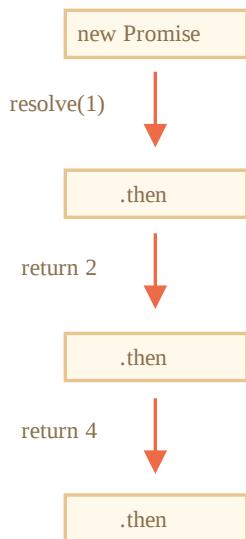
```
new Promise(function(resolve, reject) {  
  setTimeout(() => resolve(1), 1000); // (*)  
}).then(function(result) { // (**)  
  alert(result); // 1  
  return result * 2;  
}).then(function(result) { // (***)  
  alert(result); // 2  
  return result * 2;  
}).then(function(result) {  
  alert(result); // 4  
  return result * 2;  
});
```

它的理念是将 `result` 通过 `.then` 处理程序（`handler`）链进行传递。

运行流程如下：

1. 初始 `promise` 在 1 秒后进行 `resolve` (*)，
2. 然后 `.then` 处理程序（`handler`）被调用 (**）。
3. 它返回的值被传入下一个 `.then` 处理程序（`handler`） (***)
4.依此类推。

随着 `result` 在处理程序（`handler`）链中传递，我们可以看到一系列的 `alert` 调用：`1` → `2` → `4`。



之所以这么运行，是因为对 `promise.then` 的调用会返回了一个 `promise`，所以我们可以在其之上调用下一个 `.then`。

当处理程序（`handler`）返回一个值时，它将成为该 `promise` 的 `result`，所以将使用它调用下一个 `.then`。

新手常犯的一个经典错误：从技术上讲，我们也可以将多个 `.then` 添加到一个 `promise` 上。但这并不是 `promise` 链（`chaining`）。

例如：

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

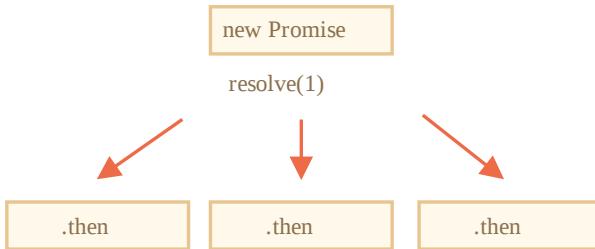
promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

我们在这里所做的只是一个 `promise` 的几个处理程序（`handler`）。他们不会相互传递 `result`；相反，它们之间彼此独立运行处理任务。

这里它的一张示意图（你可以将其与上面的链式调用做一下比较）：



在同一个 `promise` 上的所有 `.then` 获得的结果都相同 — 该 `promise` 的结果。所以，在上面的代码中，所有 `alert` 都显示相同的内容： `1`。

实际上我们极少遇到一个 `promise` 需要多处理程序（`handler`）的情况。使用链式调用的频率更高。

返回 `promise`

`.then(handler)` 中所使用的处理程序（`handler`）可以创建并返回一个 `promise`。

在这种情况下，其他的处理程序（`handler`）将等待它 `settled` 后再获得其结果（`result`）。

例如：

```

new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});

```

这里第一个 `.then` 显示 `1` 并在 `(*)` 行返回 `new Promise(...)`。1 秒后它会进行 `resolve`，然后 `result` (`resolve` 的参数，在这里它是 `result*2`) 被传递给第二个 `.then` 的处理程序（`handler`）。这个处理程序（`handler`）位于 `(**)` 行，它显示 `2`，并执行相同动作（`action`）。

所以输出与前面的示例相同：`1 → 2 → 4`，但是现在在每次 `alert` 调用之间会有 1 秒钟的延迟。

返回 `promise` 使我们能够构建异步行为链。

示例: loadScript

让我们将本章所讲的这个特性与在 [上一章](#) 中定义的 promise 化的 `loadScript` 结合使用，按顺序依次加载脚本：

```
loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // 使用在脚本中声明的函数
    // 以证明脚本确实被加载完成了
    one();
    two();
    three();
  });
});
```

我们可以用箭头函数来重写代码，让其变得简短一些：

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // 脚本加载完成，我们可以在这儿使用脚本中声明的函数
    one();
    two();
    three();
  });
});
```

在这儿，每个 `loadScript` 调用都返回一个 promise，并且在它 `resolve` 时下一个 `.then` 开始运行。然后，它启动下一个脚本的加载。所以，脚本是一个接一个地加载的。

我们可以向链中添加更多的异步行为（action）。请注意，代码仍然是“扁平”的 — 它向下增长，而不是向右。这里没有“厄运金字塔”的迹象。

从技术上讲，我们可以向每个 `loadScript` 直接添加 `.then`，就像这样：

```
loadScript("/article/promise-chaining/one.js").then(script1 => {
  loadScript("/article/promise-chaining/two.js").then(script2 => {
    loadScript("/article/promise-chaining/three.js").then(script3 => {
      // 此函数可以访问变量 script1, script2 和 script3
      one();
      two();
      three();
    });
  });
});
```

这段代码做了相同的事儿：按顺序加载 3 个脚本。但它是“向右增长”的。所以会有和使用回调函数一样的问题。

刚开始使用 `promise` 的人可能不知道 `promise` 链，所以他们就这样写了。通常，链式是首选。

有时候直接写 `.then` 也是可以的，因为嵌套的函数可以访问外部作用域。在上面的例子中，嵌套在最深层的那个回调（callback）可以访问所有变量 `script1`, `script2` 和 `script3`。但这是一个例外，而不是一条规则。

➊ Thenables

确切地说，处理程序（handler）返回的不完全是一个 `promise`，而是返回的被称为“`thenable`”对象——一个具有方法 `.then` 的任意对象。它会被当做一个 `promise` 来对待。

这个想法是，第三方库可以实现自己的“`promise` 兼容（`promise-compatible`）”对象。它们可以具有扩展的方法集，但也与原生的 `promise` 兼容，因为它们实现了 `.then` 方法。

这是一个 `thenable` 对象的示例：

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // 1 秒后使用 this.num*2 进行 resolve
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
  }
}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // 1000ms 后显示 2
```

JavaScript 检查在 `(*)` 行中由 `.then` 处理程序（handler）返回的对象：如果它具有名为 `then` 的可调用方法，那么它将调用该方法并提供原生的函数 `resolve` 和 `reject` 作为参数（类似于 `executor`），并等待直到其中一个函数被调用。在上面的示例中，`resolve(2)` 在 1 秒后被调用 `(**)`。然后，`result` 会被进一步沿着链向下传递。

这个特性允许我们将自定义的对象与 `promise` 链集成在一起，而不必继承自 `Promise`。

更复杂的示例：fetch

在前端编程中，`promise` 通常被用于网络请求。那么，让我们一起来看一个相关的扩展示例吧。

我们将使用 `fetch` 方法从远程服务器加载用户信息。它有很多可选的参数，我们在 [单独的一章](#) 中对其进行了详细介绍，但是基本语法很简单：

```
let promise = fetch(url);
```

执行这条语句，向 `url` 发出网络请求并返回一个 `promise`。当远程服务器返回 `header`（是在 [全部响应加载完成前](#)）时，该 `promise` 用使用一个 `response` 对象来进行 `resolve`。

为了读取完整的响应，我们应该调用 `response.text()` 方法：当全部文字（full text）内容从远程服务器下载完成后，它会返回一个 promise，该 promise 以刚刚下载完成的这个文本作为 `result` 进行 `resolve`。

下面这段代码向 `user.json` 发送请求，并从服务器加载该文本：

```
fetch('/article/promise-chaining/user.json')
  // 当远程服务器响应时，下面的 .then 开始执行
  .then(function(response) {
    // 当 user.json 加载完成时，response.text() 会返回一个新的 promise
    // 该 promise 以加载的 user.json 为 result 进行 resolve
    return response.text();
})
.then(function(text) {
  // ...这是远程文件的内容
  alert(text); // {"name": "iliakan", "isAdmin": true}
});
```

从 `fetch` 返回的 `response` 对象还包括 `response.json()` 方法，该方法读取远程数据并将其解析为 JSON。在我们的例子中，这更加方便，所以让我们切换到这个方法。

为了简洁，我们还将使用箭头函数：

```
// 同上，但是使用 response.json() 将远程内容解析为 JSON
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name)); // iliakan, got user name
```

现在，让我们用加载好的用户信息搞点事情。

例如，我们可以多发一个到 GitHub 的请求，加载用户个人资料并显示头像：

```
// 发送一个对 user.json 的请求
fetch('/article/promise-chaining/user.json')
  // 将其加载为 JSON
  .then(response => response.json())
  // 发送一个到 GitHub 的请求
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // 将响应加载为 JSON
  .then(response => response.json())
  // 显示头像图片 (githubUser.avatar_url) 3 秒（也可以加上动画效果）
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
});
```

这段代码可以工作，具体细节请看注释。但是，这儿有一个潜在的问题，一个新手使用 promise 的典型问题。

请看 (*) 行：我们如何能在头像显示结束并被移除 **之后** 做点什么？例如，我们想显示一个用于编辑该用户或者其他内容的表单。就目前而言，是做不到的。

为了使链可扩展，我们需要返回一个在头像显示结束时进行 `resolve` 的 promise。

就像这样：

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise(function(resolve, reject) { // (*)
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser); // (**)
    }, 3000);
  }))
  // 3 秒后触发
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));

```

也就是说，第 (*) 行的 `.then` 处理程序（handler）现在返回一个 `new Promise`，只有在 `setTimeout` 中的 `resolve(githubUser)` (**) 被调用后才会变为 settled。链中的下一个 `.then` 将一直等待这一时刻的到来。

作为一个好的做法，异步行为应该始终返回一个 promise。这样就可以使得之后我们计划后续的行为成为可能。即使我们现在不打算对链进行扩展，但我们之后可能会需要。

最后，我们可以将代码拆分为可重用的函数：

```
function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

function loadGithubUser(name) {
  return fetch(`https://api.github.com/users/${name}`)
    .then(response => response.json());
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

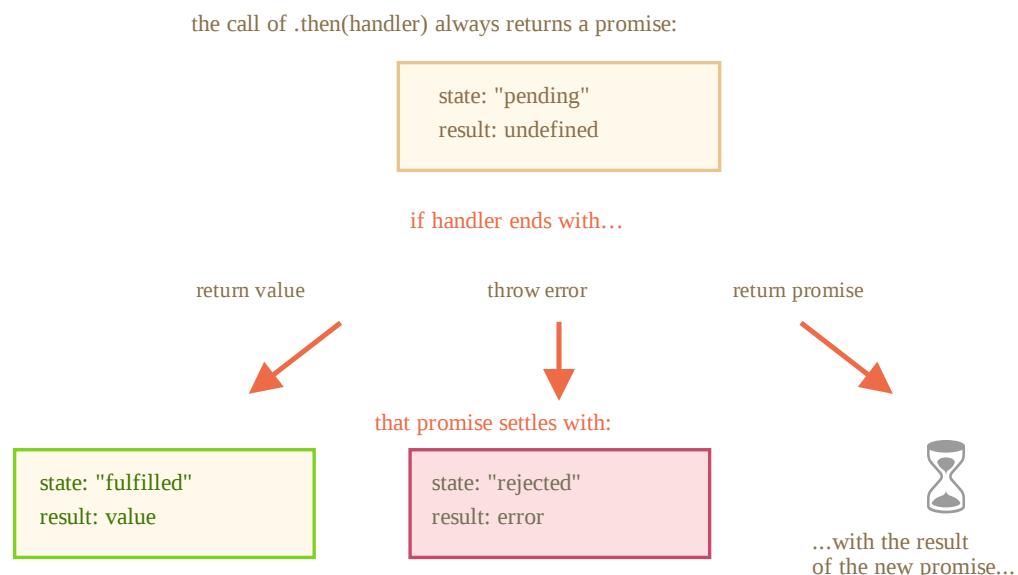
    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });
}
```

```
// 使用它们:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
// ...
```

总结

如果 `.then`（或 `catch/finally` 都可以）处理程序（handler）返回一个 promise，那么链的其余部分将会等待，直到它状态变为 `settled`。当它被 `settled` 后，其 `result`（或 `error`）将被进一步传递下去。

这是一个完整的流程图：



使用 promise 进行错误处理

Promise 链在错误（error）处理中十分强大。当一个 promise 被 `reject` 时，控制权将移交至最近的 `rejection` 处理程序（handler）。这在实际开发中非常方便。

例如，下面代码中所 `fetch` 的 URL 是错的（没有这个网站），`.catch` 对这个 `error` 进行了处理：

```
fetch('https://no-such-server.blabla') // rejects
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch (这里的文字可能有所不同)
```

正如你所看到的，`.catch` 不必是立即的。它可能在一个或多个 `.then` 之后出现。

或者，可能该网站一切正常，但响应不是有效的 JSON。捕获所有 `error` 的最简单的方法是，将 `.catch` 附加到链的末尾：

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
```

```
.then(user => fetch(`https://api.github.com/users/${user.name}`))
.then(response => response.json())
.then(githubUser => new Promise((resolve, reject) => {
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  setTimeout(() => {
    img.remove();
    resolve(githubUser);
  }, 3000);
}))
.catch(error => alert(error.message));
```

通常情况下，这样的 `.catch` 根本不会被触发。但是如果上述任意一个 promise 被 `reject`（网络问题或者无效的 `json` 或其他），`.catch` 就会捕获它。

隐式 try...catch

Promise 的执行者（executor）和 promise 的处理程序（handler）周围有一个“隐式的 `try..catch`”。如果发生异常，它（译注：指异常）就会被捕获，并被视为 `rejection` 进行处理。

例如，下面这段代码：

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!
```

.....与下面这段代码工作上完全相同：

```
new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

在 `executor` 周围的“隐式 `try..catch`”自动捕获了 `error`，并将其变为 `rejected promise`。

这不仅仅发生在 `executor` 函数中，同样也发生在其 `handler` 中。如果我们在 `.then` 处理程序（`handler`）中 `throw`，这意味着 `promise` 被 `rejected`，因此控制权移交至最近的 `error` 处理程序（`handler`）。

这是一个例子：

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("Whoops!"); // reject 这个 promise
}).catch(alert); // Error: Whoops!
```

对于所有的 error 都会发生这种情况，而不仅仅是由于 `throw` 语句导致的这些 error。例如，一个编程错误：

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // 没有这个函数
}).catch(alert); // ReferenceError: blabla is not defined
```

最后的 `.catch` 不仅会捕获显式的 rejection，还会捕获它上面的处理程序（handler）中意外出现的 error。

再次抛出（Rethrowing）

正如我们已经注意到的，链尾端的 `.catch` 的表现有点像 `try..catch`。我们可能有许多个 `.then` 处理程序（handler），然后在尾端使用一个 `.catch` 处理上面的所有 error。

在常规的 `try..catch` 中，我们可以分析错误（error），如果我们无法处理它，可以将其再次抛出。对于 promise 来说，这也是可以的。

如果我们在 `.catch` 中 `throw`，那么控制权就会被移交到下一个最近的 error 处理程序（handler）。如果我们处理该 error 并正常完成，那么它将继续到最近的成功 `.then` 处理程序（handler）。

在下面这个例子中，`.catch` 成功处理了 error：

```
// 执行流: catch -> then
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) {

  alert("The error is handled, continue normally");

}).then(() => alert("Next successful handler runs"));
```

这里 `.catch` 块正常完成。所以下一个成功的 `.then` 处理程序（handler）就会被调用。

在下面的例子中，我们可以看到 `.catch` 的另一种情况。（*）行的处理程序（handler）捕获了 error，但无法处理它（例如，它只知道如何处理 `URIError`），所以它将其再次抛出：

```
// 执行流: catch -> catch -> then
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) { // (*) }

  if (error instanceof URIError) {
    // 处理它
  } else {
    alert("Can't handle such error");
});
```

```
        throw error; // 再次抛出此 error 或另外一个 error, 执行将跳转至下一个 catch
    }

}).then(function() {
    /* 不在这里运行 */
}).catch(error => { // (**)

    alert(`The unknown error has occurred: ${error}`);
    // 不会返回任何内容 => 执行正常进行

});

```

执行从第一个 `.catch (*)` 沿着链跳转至下一个 `(**)`。

未处理的 rejection

当一个 `error` 没有被处理会发生什么？例如，我们忘了在链的尾端附加 `.catch`，像这样：

```
new Promise(function() {
    noSuchFunction(); // 这里出现 error (没有这个函数)
})
.then(() => {
    // 一个或多个成功的 promise 处理程序 (handler)
}); // 尾端没有 .catch!
```

如果出现 `error`, `promise` 的状态将变为“`rejected`”，然后执行应该跳转至最近的 `rejection` 处理程序 (`handler`)。但是上面这个例子中并没有这样的处理程序 (`handler`)。因此 `error` 会“卡住 (`stuck`)”。没有代码来处理它。

在实际开发中，就像代码中常规的未处理的 `error` 一样，这意味着某些东西出了问题。

当发生一个常规的错误 (`error`) 并且未被 `try..catch` 捕获时会发生什么？脚本死了，并在控制台 (`console`) 中留下了一个信息。对于在 `promise` 中未被处理的 `rejection`，也会发生类似的事儿。

JavaScript 引擎会跟踪此类 `rejection`，在这种情况下会生成一个全局的 `error`。如果你运行上面这个代码，你可以在控制台 (`console`) 中看到。

在浏览器中，我们可以使用 `unhandledrejection` 事件来捕获这类 `error`:

```
window.addEventListener('unhandledrejection', function(event) {
    // 这个事件对象有两个特殊的属性:
    alert(event.promise); // [object Promise] - 生成该全局 error 的 promise
    alert(event.reason); // Error: Whoops! - 未处理的 error 对象
});

new Promise(function() {
    throw new Error("Whoops!");
}); // 没有用来处理 error 的 catch
```

这个事件是 [HTML 标准](#) 的一部分。

如果出现了一个 `error`, 并且在这儿没有 `.catch`, 那么 `unhandledrejection` 处理程序 (`handler`) 就会被触发, 并获取具有 `error` 相关信息的 `event` 对象, 所以我们就能做一些后续处理了。

通常此类 `error` 是无法恢复的, 所以我们最好的解决方案是将问题告知用户, 并且可以将事件报告给服务器。

在 Node.js 等非浏览器环境中, 有其他用于跟踪未处理的 `error` 的方法。

总结

- `.catch` 处理 `promise` 中的各种 `error`: 在 `reject()` 调用中的, 或者在处理程序 (`handler`) 中抛出的 (`thrown`) `error`。
- 我们应该将 `.catch` 准确地放到我们想要处理 `error`, 并知道如何处理这些 `error` 的地方。处理程序应该分析 `error` (可以自定义 `error` 类来帮助分析) 并再次抛出未知的 `error` (可能它们是编程错误)。
- 如果没有办法从 `error` 中恢复的话, 不使用 `.catch` 也可以。
- 在任何情况下我们都应该有 `unhandledrejection` 事件处理程序 (用于浏览器, 以及其他环境的模拟), 以跟踪未处理的 `error` 并告知用户 (可能还有我们的服务器) 有关信息, 使我们的应用程序永远不会“死掉”。

补充内容

① 说明

为了更清晰地讲解 `promise`, 本文经过大幅重写, 以下内容是重写时被优化掉的内容, 译者认为还是很有学习价值的, 遂保留下来供大家学习。

Fetch 错误处理示例

让我们改进用户加载 (user-loading) 示例的错误处理。

当请求无法发出时, `fetch ↗` `reject` 会返回 `promise`。例如, 远程服务器无法访问, 或者 `URL` 异常。但是如果远程服务器返回响应错误 404, 甚至是错误 500, 这些都被认为是合法的响应。

如果在 (*) 行, 服务器返回一个错误 500 的非 JSON (non-JSON) 页面该怎么办? 如果没有这个用户, GitHub 返回错误 404 的页面又该怎么办呢?

```
fetch('no-such-user.json') // (*)
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`)) // (**)
  .then(response => response.json())
  .catch(alert); // SyntaxError: Unexpected token < in JSON at position 0
  // ...
```

到目前为止, 代码试图以 `JSON` 格式加载响应数据, 但无论如何都会因为语法错误而失败。你可以通过执行上述例子来查看相关信息, 因为文件 `no-such-user.json` 不存在。

这有点糟糕, 因为错误只是落在链上, 并没有相关细节信息: 什么失败了, 在哪里失败的。

因此我们多添加一步: 我们应该检查具有 `HTTP` 状态的 `response.status` 属性, 如果不是 200 就抛出错误。

```

class HttpError extends Error { // (1)
  constructor(response) {
    super(` ${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) { // (2)
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    })
}

loadJson('no-such-user.json') // (3)
  .catch(alert); // HttpError: 404 for .../no-such-user.json

```

1. 我们为 **HTTP** 错误创建一个自定义类用于区分 **HTTP** 错误和其他类型错误。此外，新的类有一个 `constructor`，它接受 `response` 对象，并将其保存到 `error` 中。因此，错误处理（`error-handling`）代码就能够获得响应数据了。
2. 然后我们将请求（`requesting`）和错误处理代码包装进一个函数，它能够 `fetch url` 并将所有状态码不是 `200` 视为错误。这很方便，因为我们通常需要这样的逻辑。
3. 现在 `alert` 显示更多有用的描述信息。

拥有我们自己的错误处理类的好处是我们可以使用 `instanceof` 很容易地在错误处理代码中检查错误。

例如，我们可以创建请求，如果我们得到 `404` 就可以告知用户修改信息。

下面的代码从 `GitHub` 加载给定名称的用户。如果没有这个用户，它将告知用户填写正确的名称：

```

function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Full name: ${user.name}.`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("No such user, please reenter.");
        return demoGithubUser();
      } else {
        throw err; // (*)
      }
    });
}

demoGithubUser();

```

请注意：这里的 `.catch` 会捕获所有错误，但是它仅仅“知道如何处理” `HttpError 404`。在这种特殊情况下，它意味着没有这样的用户，而 `.catch` 仅仅在这种情况下重试。

对于其他错误，它不知道会出现什么问题。可能是编程错误或者其他错误。所以它仅仅是在 (*) 行再次抛出。

其他

如果我们有加载指示（load-indication），`.finally` 是一个很好的处理程序（handler），在 `fetch` 完成时停止它：

```
function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  document.body.style.opacity = 0.3; // (1) 开始指示 (indication)

  return loadJson(`https://api.github.com/users/${name}`)
    .finally(() => { // (2) 停止指示 (indication)
      document.body.style.opacity = '';
      return new Promise(resolve => setTimeout(resolve)); // (*)
    })
    .then(user => {
      alert(`Full name: ${user.name}`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("No such user, please reenter.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

此处的 (1) 行，我们通过调暗文档来指示加载。指示方法没有什么问题，可以使用任何类型的指示来代替。

当 `promise` 得以解决，`fetch` 可以是成功或者错误，`finally` 在 (2) 行触发并终止加载指示。

有一个浏览器技巧，(*) 是从 `finally` 返回零延时（zero-timeout）的 `promise`。这是因为一些浏览器（比如 Chrome）需要“一点时间”外的 `promise` 处理程序来绘制文档的更改。因此它确保在进入链下一步之前，指示在视觉上是停止的。

Promise API

在 `Promise` 类中，有 5 种静态方法。我们在这里简单介绍下它们的使用场景。

Promise.all

假设我们希望并行执行多个 `promise`，并等待所有 `promise` 都准备就绪。

例如，并行下载几个 `URL`，并等到所有内容都下载完毕后再对它们进行处理。

这就是 `Promise.all` 的用途。

语法:

```
let promise = Promise.all([...promises...]);
```

`Promise.all` 接受一个 `promise` 数组作为参数（从技术上讲，它可以是任何可迭代的，但通常是一个数组）并返回一个新的 `promise`。

当所有给定的 `promise` 都被 `settled` 时，新的 `promise` 才会 `resolve`，并且其结果数组将成为新的 `promise` 的结果。

例如，下面的 `Promise.all` 在 3 秒之后被 `settled`，然后它的结果就是一个 `[1, 2, 3]` 数组：

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
]).then(alert); // 1,2,3 当上面这些 promise 准备好时：每个 promise 都贡献了数组中的一个元素
```

请注意，结果数组中元素的顺序与其在源 `promise` 中的顺序相同。即使第一个 `promise` 花费了最长的时间才 `resolve`，但它仍是结果数组中的第一个。

一个常见的技巧是，将一个任务数据数组映射 (`map`) 到一个 `promise` 数组，然后将其包装到 `Promise.all`。

例如，如果我们有一个存储 `URL` 的数组，我们可以像这样 `fetch` 它们：

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// 将每个 url 映射 (map) 到 fetch 的 promise 中
let requests = urls.map(url => fetch(url));

// Promise.all 等待所有任务都 resolved
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`#${response.url}: ${response.status}`)
  ));
```

一个更真实的示例，通过 `GitHub` 用户名来获取一个 `GitHub` 用户数组中用户的信息（我们也可以通过商品 `id` 来获取商品数组中的商品信息，逻辑都是一样的）：

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
```

```
.then(responses => {
  // 所有响应都被成功 resolved
  for(let response of responses) {
    alert(`#${response.url}: ${response.status}`); // 对应每个 url 都显示 200
  }

  return responses;
})
// 将响应数组映射 (map) 到 response.json() 数组中以读取它们的内容
.then(responses => Promise.all(responses.map(r => r.json())))
// 所有 JSON 结果都被解析: "users" 是它们的数组
.then(users => users.forEach(user => alert(user.name))));
```

如果任意一个 promise 被 reject，由 `Promise.all` 返回的 promise 就会立即 reject，并且带有的就是这个 error。

例如：

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

这里的第二个 promise 在两秒后 reject。这立即导致了 `Promise.all` 的 reject，因此 `.catch` 执行了：被 reject 的 error 成为了整个 `Promise.all` 的结果。

⚠ 如果出现 error，其他 promise 将被忽略

如果其中一个 promise 被 reject，`Promise.all` 就会立即被 reject，完全忽略列表中其他的 promise。它们的结果也被忽略。

例如，像上面那个例子，如果有多个同时进行的 `fetch` 调用，其中一个失败，其他的 `fetch` 操作仍然会继续执行，但是 `Promise.all` 将不会再关心（watch）它们。它们可能会 settle，但是它们的结果将被忽略。

`Promise.all` 没有采取任何措施来取消它们，因为 promise 中没有“取消”的概念。在 [另一个章节](#) 中，我们将介绍可以帮助我们解决这个问题（译注：指的是“取消” promise）的 `AbortController`，但它不是 `Promise API` 的一部分。

i `Promise.all(iterable)` 允许在 `iterable` 中使用 `non-promise` 的“常规”值

通常, `Promise.all(...)` 接受可迭代对象 (`iterable`) 的 promise (大多数情况下是数组)。但是, 如果这些对象中的任意一个都不是 `promise`, 那么它将被“按原样”传递给结果数组。

例如, 这里的结果是 `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2,
  3
]).then(alert); // 1, 2, 3
```

所以我们可以很方便的地方将准备好的值传递给 `Promise.all`。

Promise.allSettled

⚠ A recent addition

This is a recent addition to the language. Old browsers may need polyfills.

如果任意的 `promise` `reject`, 则 `Promise.all` 整个将会 `reject`。当我们需要 **所有** 结果都成功时, 它对这种“全有或全无”的情况很有用:

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
]).then(render); // render 方法需要所有 fetch 的数据
```

`Promise.allSettled` 等待所有的 `promise` 都被 `settle`, 无论结果如何。结果数组具有:

- `{status:"fulfilled", value:result}` 对于成功的响应,
- `{status:"rejected", reason:error}` 对于 `error`。

例如, 我们想要获取 (`fetch`) 多个用户的信息。即使其中一个请求失败, 我们仍然对其他的感兴趣。

让我们使用 `Promise.allSettled`:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
```

```
.then(results => { // (*)  
  results.forEach((result, num) => {  
    if (result.status == "fulfilled") {  
      alert(`#${urls[num]}: ${result.value.status}`);  
    }  
    if (result.status == "rejected") {  
      alert(`#${urls[num]}: ${result.reason}`);  
    }  
  });
});
```

上面的 (*) 行中的 `results` 将会是:

```
[  
  {status: 'fulfilled', value: ...response...},  
  {status: 'fulfilled', value: ...response...},  
  {status: 'rejected', reason: ...error object...}  
]
```

所以，对于每个 `promise`，我们都得到了其状态（`status`）和 `value/reason`。

Polyfill

如果浏览器不支持 `Promise.allSettled`，很容易进行 polyfill:

```
if(!Promise.allSettled) {  
  Promise.allSettled = function(promises) {  
    return Promise.all(promises.map(p => Promise.resolve(p).then(value => ({  
      state: 'fulfilled',  
      value  
    }), reason => ({  
      state: 'rejected',  
      reason  
    }))));  
  };  
}
```

在这段代码中，`promises.map` 获取输入值，并通过 `p => Promise.resolve(p)` 将输入值转换为 `promise`（以防传递了 non-promise），然后向每一个 `promise` 都添加 `.then` 处理程序（`handler`）。

这个处理程序（`handler`）将成功的结果 `value` 转换为 `{state:'fulfilled', value}`，将 `error reason` 转换为 `{state:'rejected', reason}`。这正是 `Promise.allSettled` 的格式。

然后我们就可以使用 `Promise.allSettled` 来获取 所有 给定的 `promise` 的结果，即使其中一些被 `reject`。

Promise.race

与 `Promise.all` 类似，但只等待第一个 `settled` 的 `promise` 并获取其结果（或 `error`）。

语法:

```
let promise = Promise.race(iterable);
```

例如，这里的结果将是 1：

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

这里第一个 promise 最快，所以它变成了结果。第一个 settled 的 promise “赢得了比赛”之后，所有进一步的 result/error 都会被忽略。

Promise.resolve/reject

在现代的代码中，很少需要使用 `Promise.resolve` 和 `Promise.reject` 方法，因为 `async/await` 语法（我们会在 [稍后](#) 讲到）使它们变得有些过时了。

完整起见，以及考虑到那些出于某些原因而无法使用 `async/await` 的人，我们在这里对它们进行介绍。

Promise.resolve

`Promise.resolve(value)` 用结果 `value` 创建一个 resolved 的 promise。

如同：

```
let promise = new Promise(resolve => resolve(value));
```

当一个函数被期望返回一个 `promise` 时，这个方法用于兼容性。（译注：这里的兼容性是指，我们直接从缓存中获取了当前操作的结果 `value`，但是期望返回的是一个 `promise`，所以可以使用 `Promise.resolve(value)` 将 `value` “封装”进 `promise`，以满足期望返回一个 `promise` 的这个需求。）

例如，下面的 `loadCached` 函数获取（`fetch`）一个 `URL` 并记住其内容。以便将来对使用相同 `URL` 的调用，它能立即从缓存中获取先前的内容，但使用 `Promise.resolve` 创建了一个该内容的 `promise`，所以返回的值始终是一个 `promise`。

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url, text);
      return text;
    });
}
```

我们可以使用 `loadCached(url).then(...)`，因为该函数保证了会返回一个 promise。我们就可以放心地在 `loadCached` 后面使用 `.then`。这就是 (*) 行中 `Promise.resolve` 的目的。

Promise.reject

`Promise.reject(error)` 用 `error` 创建一个 `rejected` 的 promise。

如同：

```
let promise = new Promise((resolve, reject) => reject(error));
```

实际上，这个方法几乎从未被使用过。

总结

`Promise` 类有 5 种静态方法：

1. `Promise.all(promises)` —— 等待所有 promise 都 `resolve` 时，返回存放它们结果的数组。如果给定的任意一个 promise 为 `reject`，那么它就会变成 `Promise.all` 的 `error`，所有其他 promise 的结果都会被忽略。
2. `Promise.allSettled(promises)` (ES2020 新增方法) —— 等待所有 promise 都 `settle` 时，并以包含以下内容的对象数组的形式返回它们的结果：
 - `state: "fulfilled"` 或 `"rejected"`
 - `value` (如果 `fulfilled`) 或 `reason` (如果 `rejected`)。
3. `Promise.race(promises)` —— 等待第一个 `settle` 的 promise，并将其 `result/error` 作为结果。
4. `Promise.resolve(value)` —— 使用给定 `value` 创建一个 `resolved` 的 promise。
5. `Promise.reject(error)` —— 使用给定 `error` 创建一个 `rejected` 的 promise。

这五个方法中，`Promise.all` 可能是在实战中使用最多的。

Promisification

“Promisification”是用于一个简单转换的一个长单词。它指将一个接受回调的函数转换为一个返回 `promise` 的函数。

由于许多函数和库都是基于回调的，因此，在实际开发中经常会需要进行这种转换。因为使用 `promise` 更加方便，所以将基于回调的函数和库 `promisify` 是有意义的。（译注：`promisify` 即指 `promise` 化）

例如，在 [简介：回调](#) 一章中我们有 `loadScript(src, callback)`。

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${src}`));
```

```

    document.head.append(script);
}

// 用法:
// loadScript('path/script.js', (err, script) => {...})

```

让我们将其 `promisify` 吧。新的 `loadScriptPromise(src)` 将会达到同样的结果，但它只接受 `src`（没有回调）并返回 `promise`。

```

let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err)
      else resolve(script);
    });
  })
}

// 用法:
// loadScriptPromise('path/script.js').then(...)

```

现在，`loadScriptPromise` 非常适合基于 `promise` 的代码。

正如我们所看到的，它将所有工作都委托给原始的 `loadScript`，并提供了转换成 `promise` `resolve/reject` 的自己的回调。

在实际开发中，我们可能需要 `promisify` 很多函数，所以使用一个 `helper` 很有意义。我们将其称为 `promisify(f)`：它接受一个需要被 `promisify` 的函数 `f`，并返回一个包装（`wrapper`）函数。

该包装（`wrapper`）函数的功能和上面的代码相同：返回一个 `promise`，将调用传递给原始的函数 `f`，并在自定义的回调中跟踪结果：

```

function promisify(f) {
  return function (...args) { // 返回一个包装函数 (wrapper-function)
    return new Promise((resolve, reject) => {
      function callback(err, result) { // 我们对 f 的自定义的回调
        if (err) {
          reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // 将我们的自定义的回调附加到 f 参数 (arguments) 的末尾
      f.call(this, ...args); // 调用原始的函数
    });
  };
}

// 用法:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);

```

这里我们假设，原始的函数期望一个带有两个参数 `(err, result)` 的回调。这就是我们最常遇到的形式。那么我们的自定义回调的格式完全正确，并且 `promisify` 在这种情况下非常有用。

但是如果原始的 `f` 期望一个带有更多参数的回调 `callback(err, res1, res2, ...)`，该怎么办呢？

下面是 `promisify` 的更高级的版本：如果像这样进行调用 `promisify(f, true)`，那么 `promise` 的结果将是回调结果的数组 `[res1, res2, ...]`：

```
// promisify(f, true) 来获取结果数组
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // 我们自定义的 f 的回调
        if (err) {
          reject(err);
        } else {
          // 如果 manyArgs 被指定，则使用所有回调的结果 resolve
          resolve(manyArgs ? results : results[0]);
        }
      }
      args.push(callback);

      f.call(this, ...args);
    });
  };
}

// 用法:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...)
```

对于一些更奇特的回调格式，例如根本没有 `err` 的格式：`callback(result)`，我们可以手动 `promisify` 这样的函数，而不使用 `helper`。

也有一些具有更灵活一点的 `promisification` 函数的模块（module），例如 [es6-promisify ↗](#)。在 Node.js 中，有一个内建的 `promisify` 函数 `util.promisify`。

❶ 请注意：

`Promisification` 是一种很好的方法，特别是在你使用 `async/await` 的时候（请看下一章），但不是回调的完全替代。

请记住，一个 `promise` 可能只有一个结果，但从技术上讲，一个回调可能被调用很多次。

因此，`promisification` 仅适用于调用一次回调的函数。进一步的调用将被忽略。

微任务（Microtask）

`Promise` 的处理程序（handlers）`.then`、`.catch` 和 `.finally` 都是异步的。

即便一个 `promise` 立即被 `resolve`，`.then`、`.catch` 和 `.finally` 下面的代码也会在这些处理程序（handler）之前被执行。

示例代码如下：

```
let promise = Promise.resolve();

promise.then(() => alert("promise done!"));

alert("code finished"); // 这个 alert 先显示
```

如果你运行它，你会首先看到 `code finished`，然后才是 `promise done`。

这很奇怪，因为这个 `promise` 肯定是一开始就完成的。

为什么 `.then` 会在之后才被触发？这是怎么回事？

微任务队列（Microtask queue）

异步任务需要适当的管理。为此，ECMA 标准规定了一个内部队列 `PromiseJobs`，通常被称为“微任务队列（microtask queue）”（ES8 术语）。

如 规范 ↗ 中所述：

- 队列（queue）是先进先出的：首先进入队列的任务会首先运行。
- 只有在 JavaScript 引擎中没有其它任务在运行时，才开始执行任务队列中的任务。

或者，简单地说，当一个 `promise` 准备就绪时，它的 `.then/catch/finally` 处理程序（handler）就会被放入队列中：但是它们不会立即被执行。当 JavaScript 引擎执行完当前的代码，它会从队列中获取任务并执行它。

这就是为什么在上面那个示例中 “`code finished`” 会先显示。



`Promise` 的处理程序（handler）总是会经过这个内部队列。

如果有一个包含多个 `.then/catch/finally` 的链，那么它们中的每一个都是异步执行的。也就是说，它会首先进入队列，然后在当前代码执行完成并且先前排队的处理程序（handler）都完成时才会被执行。

如果执行顺序对我们很重要该怎么办？我们怎么才能让 `code finished` 在 `promise done` 之后运行呢？

很简单，只需要像下面这样使用 `.then` 将其放入队列：

```
Promise.resolve()
  .then(() => alert("promise done!"))
  .then(() => alert("code finished"));
```

现在代码就是按照预期执行的。

未处理的 rejection

还记得 [使用 promise 进行错误处理](#) 一章中的 `unhandledrejection` 事件吗？

现在，我们可以确切地看到 JavaScript 是如何发现未处理的 `rejection` 的。

如果一个 `promise` 的 `error` 未被在微任务队列的末尾进行处理，则会出现“未处理的 `rejection`”。

正常来说，如果我们预期可能会发生错误，我们会在 `promise` 链上添加 `.catch` 来处理 `error`:

```
let promise = Promise.reject(new Error("Promise Failed!"));
promise.catch(err => alert('caught'));

// 不会运行: error 已经被处理
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

但是如果我们忘记添加 `.catch`，那么，微任务队列清空后，JavaScript 引擎会触发下面这件事：

```
let promise = Promise.reject(new Error("Promise Failed!"));

// Promise Failed!
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

如果我们迟一点再处理这个 `error` 会怎样？例如：

```
let promise = Promise.reject(new Error("Promise Failed!"));
setTimeout(() => promise.catch(err => alert('caught')), 1000);

// Error: Promise Failed!
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

现在，如果我们运行上面这段代码，我们会先看到 `Promise Failed!`，然后才是 `caught`。

如果我们并不了解微任务队列，我们可能会想：“为什么 `unhandledrejection` 处理程序（`handler`）会运行？我们已经捕获（`catch`）并处理了 `error`！”

但是现在我们知道了，当微任务队列中的任务都完成时，才会生成 `unhandledrejection`：引擎会检查 `promise`，如果 `promise` 中的任意一个出现“`rejected`”状态，`unhandledrejection` 事件就会被触发。

在上面这个例子中，被添加到 `setTimeout` 中的 `.catch` 也会被触发。只是会在 `unhandledrejection` 事件出现之后才会被触发，所以它并没有改变什么（没有发挥作用）。

总结

`Promise` 处理始终是异步的，因为所有 `promise` 行为都会通过内部的“`promise jobs`”队列，也被称为“微任务队列”（ES8 术语）。

因此，`.then/catch/finally` 处理程序（handler）总是在当前代码完成后才会被调用。**

如果我们要确保一段代码在 `.then/catch/finally` 之后被执行，我们可以将它添加到链式调用的 `.then` 中。

在大多数 JavaScript 引擎中（包括浏览器和 Node.js），微任务（microtask）的概念与“事件循环（event loop）”和“宏任务（macrotasks）”紧密相关。由于这些概念跟 promise 没有直接关系，所以我们将本教程另外一部分的 [事件循环：微任务和宏任务](#) 一章中对它们进行介绍。

Async/await

Async/await 是以更舒适的方式使用 promise 的一种特殊语法，同时它也非常易于理解和使用。

Async function

让我们以 `async` 这个关键字开始。它可以被放置在一个函数前面，如下所示：

```
async function f() {  
  return 1;  
}
```

在函数前面的“`async`”这个单词表达了一个简单的事情：即这个函数总是返回一个 promise。其他值将自动被包装在一个 `resolved` 的 promise 中。

例如，下面这个函数返回一个结果为 `1` 的 `resolved` promise，让我们测试一下：

```
async function f() {  
  return 1;  
}  
  
f().then(alert); // 1
```

.....我们也可以显式地返回一个 promise，结果是一样的：

```
async function f() {  
  return Promise.resolve(1);  
}  
  
f().then(alert); // 1
```

所以说，`async` 确保了函数返回一个 promise，也会将非 promise 的值包装进去。很简单，对吧？但不仅仅这些。还有另外一个叫 `await` 的关键词，它只在 `async` 函数内工作，也非常酷。

Await

语法如下：

```
// 只在 async 函数内工作  
let value = await promise;
```

关键字 `await` 让 JavaScript 引擎等待直到 `promise` 完成（`settle`）并返回结果。

这里的例子就是一个 1 秒后 `resolve` 的 `promise`:

```
async function f() {  
  
  let promise = new Promise((resolve, reject) => {  
    setTimeout(() => resolve("done!"), 1000)  
  });  
  
  let result = await promise; // 等待, 直到 promise resolve (*)  
  
  alert(result); // "done!"  
}  
  
f();
```

这个函数在执行的时候，“暂停”在了 `(*)` 那一行，并在 `promise settle` 时，拿到 `result` 作为结果继续往下执行。所以上面这段代码在一秒后显示“done!”。

让我们强调一下：`await` 字面的意思就是让 JavaScript 引擎等待直到 `promise settle`，然后以 `promise` 的结果继续执行。这个行为不会耗费任何 CPU 资源，因为引擎可以同时处理其他任务：执行其他脚本，处理事件等。

相比于 `promise.then`，它只是获取 `promise` 的结果的一个更优雅的语法，同时也更易于读写。

⚠ 不能在普通函数中使用 `await`

如果我们尝试在非 `async` 函数中使用 `await` 的话，就会报语法错误：

```
function f() {  
  let promise = Promise.resolve(1);  
  let result = await promise; // Syntax error  
}
```

如果函数前面没有 `async` 关键字，我们就会得到一个语法错误。就像前面说的，`await` 只在 `async` 函数 中有效。

让我们拿 [Promise 链](#) 那一章的 `showAvatar()` 例子，并将其改写成 `async/await` 的形式：

1. 我们需要用 `await` 替换掉 `.then` 的调用。
2. 另外，我们需要在函数前面加上 `async` 关键字，以使它们能工作。

```
async function showAvatar() {  
  
  // 读取我们的 JSON  
  let response = await fetch('/article/promise-chaining/user.json');  
  let user = await response.json();  
  
  // 读取 github 用户信息
```

```
let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
let githubUser = await githubResponse.json();

// 显示头像
let img = document.createElement('img');
img.src = githubUser.avatar_url;
img.className = "promise-avatar-example";
document.body.append(img);

// 等待 3 秒
await new Promise((resolve, reject) => setTimeout(resolve, 3000));

img.remove();

return githubUser;
}

showAvatar();
```

简洁明了，是吧？比之前可强多了。

① `await` 不能在顶层代码运行

刚开始使用 `await` 的人常常会忘记 `await` 不能用在顶层代码中。例如，下面这样就不行：

```
// 用在顶层代码中会报语法错误
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();
```

但我们可以将其包裹在一个匿名 `async` 函数中，如下所示：

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

i `await` 接受 “thenables”

像 `promise.then` 那样, `await` 允许我们使用 `thenable` 对象 (那些具有可调用的 `then` 方法的对象)。这里的看法是, 第三方对象可能不是一个 `promise`, 但却是 `promise` 兼容的: 如果这些对象支持 `.then`, 那么就可以对它们使用 `await`。

这有一个用于演示的 `Thenable` 类, 下面的 `await` 接受了该类的实例:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve);
    // 1000ms 后使用 this.num*2 进行 resolve
    setTimeout(() => resolve(this.num * 2), 1000); // (*)
  }
};

async function f() {
  // 等待 1 秒, 之后 result 变为 2
  let result = await new Thenable(1);
  alert(result);
}

f();
```

如果 `await` 接收了一个非 `promise` 的但是提供了 `.then` 方法的对象, 它就会调用这个 `.then` 方法, 并将内建的函数 `resolve` 和 `reject` 作为参数传入 (就像它对待一个常规的 `Promise executor` 时一样)。然后 `await` 等待直到这两个函数中的某个被调用 (在上面这个例子中发生在 `(*)` 行), 然后使用得到的结果继续执行后续任务。

i Class 中的 `async` 方法

要声明一个 `class` 中的 `async` 方法, 只需在对应方法前面加上 `async` 即可:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1
```

这里的含义是一样的: 它确保了方法的返回值是一个 `promise` 并且可以在方法中使用 `await`。

Error 处理

如果一个 promise 正常 resolve, `await promise` 返回的就是其结果。但是如果 promise 被 reject, 它将 throw 这个 error, 就像在这一行有一个 `throw` 语句那样。

这个代码:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

.....和下面是一样的:

```
async function f() {
  throw new Error("Whoops!");
}
```

在真实开发中, promise 可能需要一点时间后才 reject。在这种情况下, 在 `await` 抛出 (`throw`) 一个 `error` 之前会有一个延时。

我们可以用 `try..catch` 来捕获上面提到的那个 `error`, 与常规的 `throw` 使用的是一样的方式:

```
async function f() {
  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

如果有 `error` 发生, 执行控制权马上就会被移交至 `catch` 块。我们也可以用 `try` 包装多行 `await` 代码:

```
async function f() {
  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // 捕获到 fetch 和 response.json 中的错误
    alert(err);
  }
}

f();
```

如果我们没有 `try..catch`, 那么由异步函数 `f()` 的调用生成的 promise 将变为 rejected。我们可以在函数调用后面添加 `.catch` 来处理这个 `error`:

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() 变成了一个 rejected 的 promise
f().catch(alert); // TypeError: failed to fetch // (*)
```

如果我们忘了在这添加 `.catch`，那么我们就会得到一个未处理的 promise error（可以在控制台中查看）。我们可以使用在 [使用 promise 进行错误处理](#) 一章中所讲的全局事件处理程序 `unhandledrejection` 来捕获这类 error。

i `async/await` 和 `promise.then/catch`

当我们使用 `async/await` 时，几乎就不会用到 `.then` 了，因为 `await` 为我们处理了等待。并且我们使用常规的 `try..catch` 而不是 `.catch`。这通常（但不总是）更加方便。

但是当我们在代码的顶层时，也就是在所有 `async` 函数之外，我们在语法上就不能使用 `await` 了，所以这时候通常的做法是添加 `.then/catch` 来处理最终的结果（result）或掉出来的（falling-through）error，例如像上面那个例子中的 (*) 行那样。

i `async/await` 可以和 `Promise.all` 一起使用

当我们需要同时等待多个 promise 时，我们可以用 `Promise.all` 把它们包装起来，然后使用 `await`：

```
// 等待结果数组
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

如果出现 error，也会正常传递，从失败了的 promise 传到 `Promise.all`，然后变成我们能通过使用 `try..catch` 在调用周围捕获到的异常（exception）。

总结

函数前面的关键字 `async` 有两个作用：

1. 让这个函数总是返回一个 promise。
2. 允许在该函数内使用 `await`。

`Promise` 前的关键字 `await` 使 JavaScript 引擎等待该 promise settle，然后：

1. 如果有 error，就会抛出异常 — 就像那里调用了 `throw error` 一样。
2. 否则，就返回结果。

这两个关键字一起提供了一个很好的用来编写异步代码的框架，这种代码易于阅读也易于编写。

有了 `async/await` 之后，我们就几乎不需要使用 `promise.then/catch`，但是不要忘了它们是基于 `promise` 的，因为有些时候（例如在最外层作用域）我们不得不使用这些方法。并且，当我们需要同时等待需要任务时，`Promise.all` 是很好用的。

Generator, 高级 iteration

Generator

常规函数只会返回一个单一值（或者不返回任何值）。

而 Generator 可以按需一个接一个地返回（“yield”）多个值。它们可与 `iterable` 完美配合使用，从而可以轻松地创建数据流。

Generator 函数

要创建一个 generator，我们需要一个特殊的语法结构：`function*`，即所谓的“generator function”。

它看起来像这样：

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

Generator 函数与常规函数的行为不同。在此类函数被调用时，它不会运行其代码。而是返回一个被称为“generator object”的特殊对象，来管理执行流程。

我们来看一个例子：

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

// "generator function" 创建了一个 "generator object"
let generator = generateSequence();
alert(generator); // [object Generator]
```

到目前为止，上面这段代码中的 **函数体** 代码还没有开始执行：

```
function* generateSequence() { ←
  yield 1;
  yield 2;
  return 3;
}
```

一个 generator 的主要方法就是 `next()`。当被调用时（译注：指 `next()` 方法），它会恢复上图所示的运行，执行直到最近的 `yield <value>` 语句（`value` 可以被省略，默认为

`undefined`)。然后函数执行暂停，并将产出的（`yielded`）值返回到外部代码。

`next()` 的结果始终是一个具有两个属性的对象：

- `value`：产出的（`yielded`）的值。
- `done`：如果 `generator` 函数已执行完成则为 `true`，否则为 `false`。

例如，我们可以创建一个 `generator` 并获取其第一个产出的（`yielded`）值：

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  return 3;  
  
}  
  
let generator = generateSequence();  
  
let one = generator.next();  
  
alert(JSON.stringify(one)); // {value: 1, done: false}
```

截至目前，我们只获得了第一个值，现在函数执行处在第二行：

```
function* generateSequence() {  
  yield 1; ← {value: 1, done: false}  
  yield 2;  
  return 3;  
}
```

让我们再次调用 `generator.next()`。代码恢复执行并返回下一个 `yield` 的值：

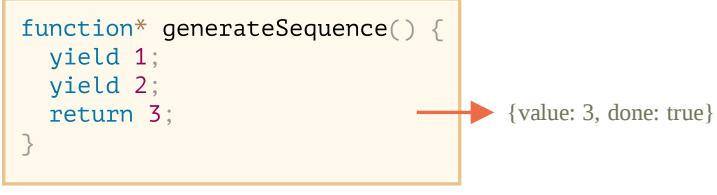
```
let two = generator.next();  
  
alert(JSON.stringify(two)); // {value: 2, done: false}
```

```
function* generateSequence() {  
  yield 1; ← {value: 2, done: false}  
  yield 2;  
  return 3;  
}
```

如果我们第三次调用 `generator.next()`，代码将会执行到 `return` 语句，此时就完成这个函数的执行：

```
let three = generator.next();  
  
alert(JSON.stringify(three)); // {value: 3, done: true}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```



现在 generator 执行完成。我们通过 `done: true` 可以看出来这一点，并且将 `value: 3` 处理为最终结果。

再对 `generator.next()` 进行新的调用不再有任何意义。如果我们这样做，它将返回相同的对象: `{done: true}`。

① `function* f(...)` 或 `function *f(...)` ?

这两种语法都是对的。

但是通常更倾向于第一种语法，因为星号 `*` 表示它是一个 generator 函数，它描述的是函数种类而不是名称，因此 `*` 应该和 `function` 关键字紧贴一起。

Generator 是可迭代的

当你看到 `next()` 方法，或许你已经猜到了 generator 是 可迭代 (iterable) 的。（译注：`next()` 是 iterator 的必要方法）

我们可以使用 `for..of` 循环遍历它所有的值:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, 然后是 2
}
```

`for..of` 写法是不是看起来比 `.next().value` 优雅多了？

.....但是请注意：上面这个例子会先显示 `1`，然后是 `2`，然后就没了。它不会显示 `3`！

这是因为当 `done: true` 时，`for..of` 循环会忽略最后一个 `value`。因此，如果我们想要通过 `for..of` 循环显示所有的结果，我们必须使用 `yield` 返回它们：

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let generator = generateSequence();
```

```
for(let value of generator) {
  alert(value); // 1, 然后是 2, 然后是 3
}
```

因为 **generator** 是可迭代的，我们可以使用 **iterator** 的所有相关功能，例如： **spread** 语法 `...`：

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3
```

在上面这段代码中，`...generateSequence()` 将可迭代的 **generator** 对象转换为了一个数组（关于 **spread** 语法的更多细节请见 [Rest 参数与 Spread 语法](#)）。

使用 **generator** 进行迭代

在前面的 [Iterable object \(可迭代对象\)](#) 一章中，我们创建了一个可迭代的 **range** 对象，它返回 `from..to` 的值。

现在，我们回忆一下代码：

```
let range = {
  from: 1,
  to: 5,

  // for..of range 在一开始就调用一次这个方法
  [Symbol.iterator]() {
    // ...它返回 iterator object:
    // 后续的操作中，for..of 将只针对这个对象，并使用 next() 向它请求下一个值
    return {
      current: this.from,
      last: this.to,

      // for..of 循环在每次迭代时都会调用 next()
      next() {
        // 它应该以对象 {done:..., value :...} 的形式返回值
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }

  // 迭代整个 range 对象，返回从 `range.from` 到 `range.to` 范围的所有数字
  alert([...range]); // 1,2,3,4,5
}
```

我们可以通过提供一个 generator 函数作为 `Symbol.iterator`，来使用 generator 进行迭代：

下面是一个相同的 `range`，但紧凑得多：

```
let range = {
  from: 1,
  to: 5,
  *[Symbol.iterator]() { // [Symbol.iterator]: function*() 的简写形式
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

alert( [...range] ); // 1,2,3,4,5
```

之所以代码正常工作，是因为 `range[Symbol.iterator]()` 现在返回一个 generator，而 generator 方法正是 `for..of` 所期望的：

- 它具有 `.next()` 方法
- 它以 `{value: ..., done: true/false}` 的形式返回值

当然，这不是巧合。Generator 被添加到 JavaScript 语言中是有对 iterator 的考量的，以便更容易地实现 iterator。

带有 generator 的变体比原来的 `range` 迭代代码简洁得多，并且保持了相同的功能。

① Generator 可以永远产出 (yield) 值

在上面的示例中，我们生成了有限序列，但是我们也可以创建一个生成无限序列的 generator，它可以一直产出 (yield) 值。例如，无序的伪随机数序列。

这种情况下肯定需要在 generator 的 `for..of` 循环中添加一个 `break`（或者 `return`）。否则循环将永远重复下去并挂起。

Generator 组合

Generator 组合（composition）是 generator 的一个特殊功能，它允许透明地（transparently）将 generator 彼此“嵌入（embed）”到一起。

例如，我们有一个生成数字序列的函数：

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
```

现在，我们想重用它来生成一个更复杂的序列：

- 首先是数字 `0..9`（字符代码为 48...57），
- 接下来是大写字母 `A..Z`（字符代码为 65...90）
- 接下来是小写字母 `a...z`（字符代码为 97...122）

我们可以对这个序列进行应用，例如，我们可以从这个序列中选择字符来创建密码（也可以添加语法字符），但让我们先生成它。

在常规函数中，要合并其他多个函数的结果，我们需要调用它们，存储它们的结果，最后再将它们合并到一起。

对于 generator 而言，我们可以使用 `yield*` 这个特殊的语法来将一个 generator “嵌入”（组合）到另一个 generator 中：

组合的 generator 的例子：

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generatePasswordCodes() {

  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);

  // a..z
  yield* generateSequence(97, 122);

}

let str = '';

for(let code of generatePasswordCodes()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

`yield*` 指令将执行 **委托** 给另一个 generator。这个术语意味着 `yield* gen` 在 generator `gen` 上进行迭代，并将其产出（`yield`）的值透明地（transparently）转发到外部。就好像这些值就是由外部的 generator `yield` 的一样。

执行结果与我们内联嵌套 generator 中的代码获得的结果相同：

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generateAlphaNum() {

  // yield* generateSequence(48, 57);
  for (let i = 48; i <= 57; i++) yield i;

  // yield* generateSequence(65, 90);
  for (let i = 65; i <= 90; i++) yield i;

  // yield* generateSequence(97, 122);
  for (let i = 97; i <= 122; i++) yield i;
}
```

```

}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z

```

Generator 组合（composition）是将一个 generator 流插入到另一个 generator 流的自然的方式。它不需要使用额外的内存来存储中间结果。

“yield” 是一条双向路

目前看来，generator 和可迭代对象类似，都具有用来生成值的特殊语法。但实际上，generator 更加强大且灵活。

这是因为 `yield` 是一条双向路（two-way street）：它不仅可以向外返回结果，而且还可以将外部的值传递到 generator 内。

调用 `generator.next(arg)`，我们就能将参数 `arg` 传递到 generator 内部。这个 `arg` 参数会变成 `yield` 的结果。

我们来看一个例子：

```

function* gen() {
  // 向外部代码传递一个问题并等待答案
  let result = yield "2 + 2 = ?"; // (*)

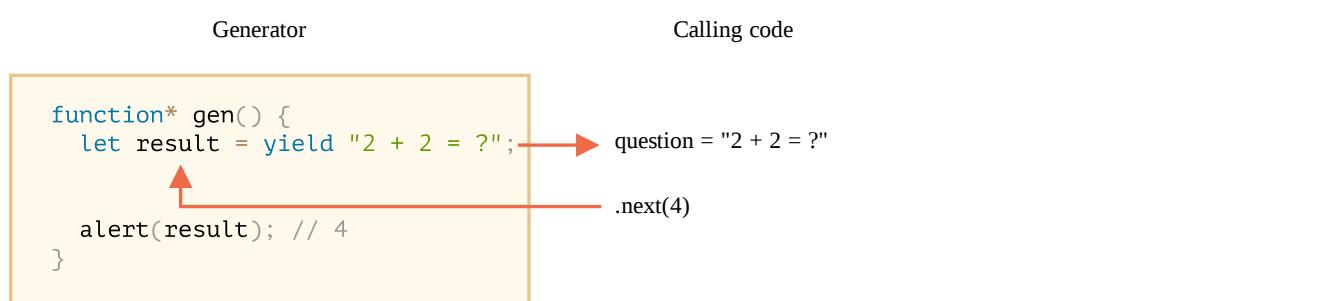
  alert(result);
}

let generator = gen();

let question = generator.next().value; // <-- yield 返回的 value

generator.next(4); // --> 将结果传递到 generator 中

```



- 第一次调用 `generator.next()` 应该是不带参数的（如果带参数，那么该参数会被忽略）。它开始执行并返回第一个 `yield "2 + 2 = ?"` 的结果。此时，generator 执行暂停，而停留在 `(*)` 行上。

- 然后，正如上面图片中显示的那样，`yield` 的结果进入调用代码中的 `question` 变量。
- 在 `generator.next(4)`，`generator` 恢复执行，并获得了 `4` 作为结果：`let result = generator.next(4)`。

请注意，外部代码不必立即调用 `next(4)`。外部代码可能需要一些时间。这没问题：`generator` 将等待它。

例如：

```
// 一段时间后恢复 generator
setTimeout(() => generator.next(4), 1000);
```

我们可以看到，与常规函数不同，`generator` 和调用 `generator` 的代码可以通过在 `next/yield` 中传递值来交换结果。

为了讲得更浅显易懂，我们来看另一个例子，其中包含了许多调用：

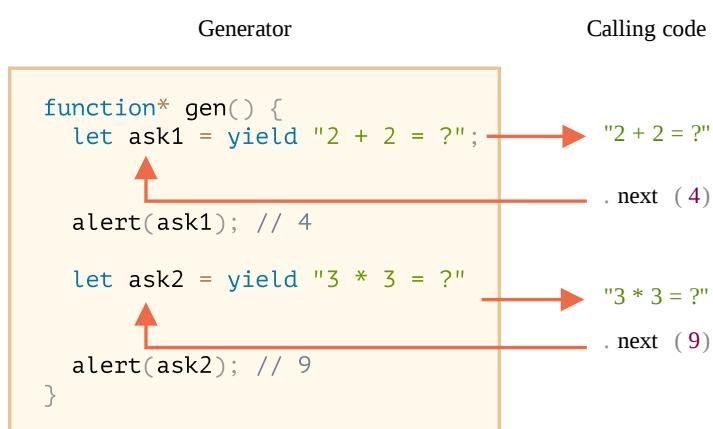
```
function* gen() {
  let ask1 = yield "2 + 2 = ?";
  alert(ask1); // 4

  let ask2 = yield "3 * 3 = ?";
  alert(ask2); // 9
}

let generator = gen();

alert(generator.next().value); // "2 + 2 = ?"
alert(generator.next(4).value); // "3 * 3 = ?"
alert(generator.next(9).done); // true
```

执行图：



- 第一个 `.next()` 启动了 `generator` 的执行……执行到达第一个 `yield`。
- 结果被返回到外部代码中。

3. 第二个 `.next(4)` 将 `4` 作为第一个 `yield` 的结果传递回 `generator` 并恢复 `generator` 的执行。
4.执行到达第二个 `yield`，它变成了 `generator` 调用的结果。
5. 第三个 `next(9)` 将 `9` 作为第二个 `yield` 的结果传入 `generator` 并恢复 `generator` 的执行，执行现在到达了函数的最底部，所以返回 `done: true`。

这个过程就像“乒乓球”游戏。每个 `next(value)`（除了第一个）传递一个值到 `generator` 中，该值变成了当前 `yield` 的结果，然后获取下一个 `yield` 的结果。

generator.throw

正如我们在上面的例子中观察到的那样，外部代码可能会将一个值传递到 `generator`，作为 `yield` 的结果。

.....但是它也可以在那里发起（抛出）一个 `error`。这很自然，因为 `error` 本身也是一种结果。

要向 `yield` 传递一个 `error`，我们应该调用 `generator.throw(err)`。在这种情况下，`err` 将被抛到对应的 `yield` 所在的那一行。

例如，`"2 + 2?"` 的 `yield` 导致了一个 `error`:

```
function* gen() {
  try {
    let result = yield "2 + 2 = ?"; // (1)

    alert("The execution does not reach here, because the exception is thrown above");
  } catch(e) {
    alert(e); // 显示这个 error
  }
}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("The answer is not found in my database")); // (2)
```

在 (2) 行引入到 `generator` 的 `error` 导致了在 (1) 行中的 `yield` 出现了一个异常。在上面这个例子中，`try..catch` 捕获并显示了这个 `error`。

如果我们没有捕获它，那么就会像其他的异常一样，它将从 `generator` “掉出”到调用代码中。

调用代码的当前行是 `generator.throw` 所在的那一行，标记为 (2)。所以我们可以在这里捕获它，就像这样:

```
function* generate() {
  let result = yield "2 + 2 = ?"; // 这行出现 error
}

let generator = generate();

let question = generator.next().value;

try {
```

```
generator.throw(new Error("The answer is not found in my database"));
} catch(e) {
  alert(e); // 显示这个 error
}
```

如果我们没有在那里捕获这个 `error`, 那么, 通常, 它会掉入外部调用代码 (如果有), 如果在外部也没有被捕获, 则会杀死脚本。

总结

- Generator 是通过 `generator` 函数 `function* f(...){...}` 创建的。
- 在 `generator` (仅在) 内部, 存在 `yield` 操作。
- 外部代码和 `generator` 可能会通过 `next/yield` 调用交换结果。

在现代 JavaScript 中, `generator` 很少被使用。但有时它们会派上用场, 因为函数在执行过程中与调用代码交换数据的能力是非常独特的。而且, 当然, 它们非常适合创建可迭代对象。

并且, 在下一章我们将会学习 `async generator`, 它们被用于在 `for await ... of` 循环中读取异步生成的数据流 (例如, 通过网络分页提取 (paginated fetches over a network))。

在 Web 编程中, 我们经常使用数据流, 因此这是另一个非常重要的使用场景。

Async iterator 和 generator

异步迭代器 (iterator) 允许我们对按需通过异步请求而得到的数据进行迭代。例如, 我们通过网络分段 (chunk-by-chunk) 下载数据时。异步生成器 (generator) 使这一步骤更加方便。

首先, 让我们来看一个简单的示例以掌握语法, 然后再看一个实际用例。

Async iterator

异步迭代器 (async iterator) 与常规的迭代器类似, 不过语法上有一点区别。

一个“常规的”可迭代对象, 即我们在 [Iterable object \(可迭代对象\)](#) 一章中提到的, 看起来像这样:

```
let range = {
  from: 1,
  to: 5,

  // 在刚使用 for..of 循环时, for..of 就会调用一次这个方法
  [Symbol.iterator]() {
    // ...它返回 iterator object:
    // 后续的操作中, for..of 将只针对这个对象
    // 并使用 next() 向它请求下一个值
    return {
      current: this.from,
      last: this.to,

      // for..of 循环在每次迭代时都会调用 next()
      next() { // (2)
        // 它应该以对象 {done:..., value :....} 的形式返回值
        if (this.current <= this.last) {
```

```

        return { done: false, value: this.current++ };
    } else {
        return { done: true };
    }
}
};

for(let value of range) {
    alert(value); // 1, 然后 2, 然后 3, 然后 4, 然后 5
}

```

有需要的话，你可以返回 **Iterable object**（可迭代对象）一章学习关于常规迭代器（**iterator**）的详细内容。

为了使对象可以异步迭代：

1. 我们需要使用 **Symbol.asyncIterator** 取代 **Symbol.iterator**。
2. **next()** 方法应该返回一个 **promise**。
3. 我们应该使用 **for await (let item of iterable)** 循环来迭代这样的对象

接下来，让我们创建一个类似于之前的，可迭代的 **range** 对象，不过现在它会按照每秒一个的速度，异步地返回值：

```

let range = {
    from: 1,
    to: 5,

    // 在刚使用 for await..of 循环时，for await..of 就会调用一次这个方法
    [Symbol.asyncIterator]() { // (1)
        // ...它返回 iterator object:
        // 后续的操作中，for await..of 将只针对这个对象
        // 并使用 next() 向它请求下一个值
        return {
            current: this.from,
            last: this.to,

            // for await..of 循环在每次迭代时都会调用 next()
            async next() { // (2)
                // 它应该以对象 {done:..., value :...} 的形式返回值
                // (会被 async 自动包装成一个 promise)

                // 可以在内部使用 await，执行异步任务:
                await new Promise(resolve => setTimeout(resolve, 1000)); // (3)

                if (this.current <= this.last) {
                    return { done: false, value: this.current++ };
                } else {
                    return { done: true };
                }
            }
        };
    };

    (async () => {

```

```
for await (let value of range) { // (4)
  alert(value); // 1,2,3,4,5
}
})()
```

正如我们所看到的，其结构与常规的 `iterator` 类似：

1. 为了使一个对象可以异步迭代，它必须具有方法 `Symbol.asyncIterator` (1)。
2. 这个方法必须返回一个带有 `next()` 方法的对象，`next()` 方法会返回一个 `promise` (2)。
3. 这个 `next()` 方法可以不是 `async` 的，它可以是一个返回值是一个 `promise` 的常规的方法，但是使用 `async` 关键字可以允许我们在方法内部使用 `await`，所以会更加方便。这里我们只是用于延迟 1 秒的操作 (3)。
4. 我们使用 `for await(let value of range)` (4) 来进行迭代，也就是在 `for` 后面添加 `await`。它会调用一次 `range[Symbol.asyncIterator]()` 方法一次，然后调用它的 `next()` 方法获取值。

这是一个小备忘单：

	Iterator	Async iterator
提供 <code>iterator</code> 的对象方法	<code>Symbol.iterator</code>	<code>Symbol.asyncIterator</code>
<code>next()</code> 返回的值是	任意值	<code>Promise</code>
要进行循环，使用	<code>for..of</code>	<code>for await..of</code>

⚠ Spread 语法 ... 无法异步工作

需要常规的同步 `iterator` 的功能，无法与异步 `iterator` 一起使用。

例如，`spread` 语法无法工作：

```
alert( [...range] ); // Error, no Symbol.iterator
```

这很正常，因为它期望找到 `Symbol.iterator`，跟 `for..of` 没有 `await` 一样。并非 `Symbol.asyncIterator`。

Async generator

正如我们所知，JavaScript 也支持生成器（generator），并且它们也是可迭代的。

让我们回顾一下 [Generator](#) 一章的序列生成器（generator）。它生成从 `start` 到 `end` 的一系列值：

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    yield i;
  }
}
```

```
for(let value of generateSequence(1, 5)) {
  alert(value); // 1, 然后 2, 然后 3, 然后 4, 然后 5
}
```

在常规的 `generator` 中，我们无法使用 `await`。所有的值都必须同步获得：`for..of` 中没有延时的地方，它是一个同步结构。

但是，如果我们需要在 `generator` 内使用 `await` 该怎么办呢？我们以执行网络请求为例子。没问题，只需要在它前面加上 `async` 即可，就像这样：

```
async function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    // 耶，可以使用 await 了！
    await new Promise(resolve => setTimeout(resolve, 1000));

    yield i;
  }

  (async () => {
    let generator = generateSequence(1, 5);
    for await (let value of generator) {
      alert(value); // 1, 然后 2, 然后 3, 然后 4, 然后 5
    }
  })();
}
```

现在，我们有了 `async generator`，可以使用 `for await...of` 进行迭代。

这确实非常简单。我们加了 `async` 关键字，然后我们就能在 `generator` 内部使用 `await` 了，依赖于 `promise` 和其他异步函数。

从技术上来讲，`async generator` 的另一个不同之处在于，它的 `generatr.next()` 方法现在也是异步的，它返回 `promise`。

在一个常规的 `generator` 中，我们使用 `result = generator.next()` 来获得值。但在一个 `async generator` 中，我们应该添加 `await` 关键字，像这样：

```
result = await generator.next(); // result = {value: ..., done: true/false}
```

Async iterable

正如我们所知道的，要使一个对象可迭代，我们需要给它添加 `Symbol.iterator`。

```
let range = {
  from: 1,
  to: 5,
```

```
[Symbol.iterator]() {
    return <object with next to make range iterable>
}
}
```

对于 `Symbol.iterator` 来说，一个通常的做法是返回一个 `generator`，而不是像前面的例子中那样返回一个带有 `next()` 方法的普通对象。

让我们回顾一下来自之前 [Generator](#) 一章中的一个示例：

```
let range = {
  from: 1,
  to: 5,

  *[Symbol.iterator]() { // [Symbol.iterator]: function*() 的简写形式
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

for(let value of range) {
  alert(value); // 1, 然后 2, 然后 3, 然后 4, 然后 5
}
```

这有一个自定义的对象 `range`，它是可迭代的，并且它的 generator `*[Symbol.iterator]` 实现了列出值的逻辑。

如果们想要给 `generator` 加上异步行为，那么我们应该将 `Symbol.iterator` 替换成异步的 `Symbol.asyncIterator`：

```
let range = {
  from: 1,
  to: 5,

  async *[Symbol.asyncIterator]() { // 等价于 [Symbol.asyncIterator]: async function*()
    for(let value = this.from; value <= this.to; value++) {

      // 在 value 之间暂停一会儿，等待一些东西
      await new Promise(resolve => setTimeout(resolve, 1000));

      yield value;
    }
  }
};

(async () => {

  for await (let value of range) {
    alert(value); // 1, 然后 2, 然后 3, 然后 4, 然后 5
  }
})();
```

现在，`value` 之间的延迟为 1 秒。

实际的例子

到目前为止，我们为了获得基础的了解，看到的都是简单的例子。接下来，我们来看一个实际的用例。

目前，有很多在线服务都是发送的分页数据（**paginated data**）。例如，当我们需要一个用户列表时，一个请求只返回一个预定义数量的用户（例如 100 个用户）—“一页”，并提供了指向下一页的 **URL**。

这种模式非常常见。不仅可用于获取用户列表，这种模式还可以用于任意东西。例如，GitHub 允许使用相同的分页提交（**paginated fashion**）的方式找回 commit:

- 我们应该提交一个请求到这种格式的 URL:
`https://api.github.com/repos/<repo>/commits`。
- 它返回一个包含 30 条 commit 的 JSON，并在返回的 `Link header` 中提供了指向下一页的链接。
- 然后我们可以将该链接用于下一个请求，以获取更多 commit，以此类推。

但是我们希望有一个更简单的 API：具有 `commit` 的可迭代对象，然后我们就可以像这样来遍历它们：

```
let repo = 'javascript-tutorial/en.javascript.info'; // 用于获取 commit 的 GitHub 仓库

for await (let commit of fetchCommits(repo)) {
  // 处理 commit
}
```

我们想创建一个函数 `fetchCommits(repo)`，用来在任何我们需要的时候发出请求，来为我们获取 `commit`。并且让它关注于所有分页的数据。对于我们来说，它就是一个简单的 `for await..of`。

通过使用 `async generator`，我们可以很容易地实现它：

```
async function* fetchCommits(repo) {
  let url = `https://api.github.com/repos/${repo}/commits`;

  while (url) {
    const response = await fetch(url, { // (1)
      headers: {'User-Agent': 'Our script'}, // github 要求 user-agent header
    });

    const body = await response.json(); // (2) 响应的是 JSON (array of commits)

    // (3) 前往下一页的 URL 在 header 中，提取它
    let nextPage = response.headers.get('Link').match(/<(.+?)>; rel="next"/);
    nextPage = nextPage && nextPage[1];

    url = nextPage;

    for(let commit of body) { // (4) 一个接一个地 yield commit，直到最后一页
      yield commit;
    }
  }
}
```

```
    }  
}
```

1. 我们使用浏览器的 `fetch` 方法从远程 URL 下载数据。它允许我们提供授权和其他 `header`, 如果需要 — 这里 GitHub 需要的是 `User-Agent`。
2. `fetch` 的结果被解析为 `JSON`。这又是 `fetch` 特定的方法。
3. 我们应该从响应 (`response`) 的 `Link header` 中获取前往下一页的 `URL`。它有一个特殊的格式, 所以我们对它使用正则表达式 (`regexp`)。前往下一页的 `URL` 看起来就像这样 `https://api.github.com/repositories/93253246/commits?page=2`。这是由 GitHub 自己生成的。
4. 然后我们将接收到的所有 `commit` 都 `yield` 出来, 当它 `yield` 完成时, 将触发下一个 `while(url)` 迭代, 并发出下一个请求。

这是一个使用示例 (在控制台中显示 `commit` 的作者)

```
(async () => {  
  
  let count = 0;  
  
  for await (const commit of fetchCommits('javascript-tutorial/en.javascript.info')) {  
  
    console.log(commit.author.login);  
  
    if (++count == 100) { // 让我们在获取了 100 个 commit 时停止  
      break;  
    }  
  }  
})();
```

这就是我们想要的。从外部看不到分页请求 (paginated requests) 的内部机制。对我们来说, 它只是一个返回 `commit` 的 `async generator`。

总结

常规的 `iterator` 和 `generator` 可以很好地处理那些不需要花费时间来生成的数据。

当我们期望异步地, 有延迟地获取数据时, 可以使用它们的 `async counterpart`, 并且使用 `for await..of` 替代 `for..of`。

`Async iterator` 与常规 `iterator` 在语法上的区别:

	Iterable	Async Iterable
提供 <code>iterator</code> 的对象方法	<code>Symbol.iterator</code>	<code>Symbol.asyncIterator</code>
<code>next()</code> 返回的值是	{ <code>value:...</code> , <code>done: true/false</code> }	resolve 成 { <code>value:...</code> , <code>done: true/false</code> } 的 <code>Promise</code>

`Async generator` 与常规 `generator` 在语法上的区别:

Generator	Async generator
声明方式	<code>function*</code>
<code>next()</code> 返回的值是	<code>{value:..., done: true/false}</code>
	<code>resolve 成 {value:..., done: true/false} 的 Promise</code>

在 Web 开发中，我们经常会遇到数据流，它们分段流动（flows chunk-by-chunk）。例如，下载或上传大文件。

我们可以使用 `async generator` 来处理此类数据。值得注意的是，在一些环境，例如浏览器环境下，还有另一个被称为 Streams 的 API，它提供了特殊的接口来处理此类数据流，转换数据并将数据从一个数据流传递到另一个数据流（例如，从一个地方下载并立即发送到其他地方）。

模块

模块 (Module) 简介

随着我们的应用越来越大，我们想要将其拆分成多个文件，即所谓的“模块（module）”。一个模块通常包含一个类或一个函数库。

很长一段时间，JavaScript 都没有语言级（language-level）的模块语法。这不是一个问题，因为最初的脚本又小又简单，所以没必要将其模块化。

但是最终脚本变得越来越复杂，因此社区发明了许多种方法来将代码组织到模块中，使用特殊的库按需加载模块。

例如：

- [AMD](#) — 最古老的模块系统之一，最初由 [require.js](#) 库实现。
- [CommonJS](#) — 为 Node.js 服务器创建的模块系统。
- [UMD](#) — 另外一个模块系统，建议作为通用的模块系统，它与 AMD 和 CommonJS 都兼容。

现在，所有他们都在慢慢成为历史的一部分，但我们仍然可以在旧脚本中找到它们。

语言级的模块系统在 2015 年的时候出现在了标准（ES6）中，此后逐渐发展，现在已经得到了所有主流浏览器和 Node.js 的支持。因此，我们将从现在开始学习它们。

什么是模块？

一个模块（module）就是一个文件。一个脚本就是一个模块。

模块可以相互加载，并可以使用特殊的指令 `export` 和 `import` 来交换功能，从另一个模块调用一个模块的函数：

- `export` 关键字标记了可以从当前模块外部访问的变量和函数。
- `import` 关键字允许从其他模块导入功能。

例如，我们有一个 `sayHi.js` 文件导出了一个函数：

```
// sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

.....然后另一个文件可能导入并使用了这个函数：

```
// main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

`import` 指令通过相对于当前文件的路径 `./sayHi.js` 加载模块，并将导入的函数 `sayHi` 分配（assign）给相应的变量。

让我们在浏览器中运行一下这个示例。

由于模块支持特殊的关键字和功能，因此我们必须通过使用 `<script type="module">` 特性（attribute）来告诉浏览器，此脚本应该被当作模块（module）来对待。

像这样：

[https://plnkr.co/edit/ssBCRkCGgMS6Ah6x?p=preview ↗](https://plnkr.co/edit/ssBCRkCGgMS6Ah6x?p=preview)

浏览器会自动获取并解析（evaluate）导入的模块（如果需要，还可以分析该模块的导入），然后运行该脚本。

模块核心功能

与“常规”脚本相比，模块有什么不同呢？

下面是一些核心的功能，对浏览器和服务端的 JavaScript 来说都有效。

始终使用 “use strict”

模块始终默认使用 `use strict`，例如，对一个未声明的变量赋值将产生错误（译注：在浏览器控制台可以看到 `error` 信息）。

```
<script type="module">
  a = 5; // error
</script>
```

模块级作用域

每个模块都有自己的顶级作用域（top-level scope）。换句话说，一个模块中的顶级作用域变量和函数在其他脚本中是不可见的。

在下面这个例子中，我们导入了两个脚本，`hello.js` 尝试使用在 `user.js` 中声明的变量 `user`，失败了：

[https://plnkr.co/edit/bU9ucoDlk5TOu9CX?p=preview ↗](https://plnkr.co/edit/bU9ucoDlk5TOu9CX?p=preview)

模块期望 `export` 它们想要被外部访问的内容，并 `import` 它们所需要的内容。

所以，我们应该将 `user.js` 导入到 `hello.js` 中，并从中获取所需的功能，而不要依赖于全局变量。

这是正确的变体：

[https://plnkr.co/edit/iaAyMLtcXcX7ofrR?p=preview ↗](https://plnkr.co/edit/iaAyMLtcXcX7ofrR?p=preview)

在浏览器中，每个 `<script type="module">` 也存在独立的顶级作用域（译注：在浏览器控制台可以看到 `error` 信息）。

```
<script type="module">
  // 变量仅在这个 module script 内可见
  let user = "John";
</script>

<script type="module">
  alert(user); // Error: user is not defined
</script>
```

如果我们真的需要创建一个 `window-level` 的全局变量，我们可以将其明确地赋值给 `window`，并以 `window.user` 来访问它。但是这需要你有足够充分的理由，否则就不要这样做。

模块代码仅在第一次导入时被解析

如果同一个模块被导入到多个其他位置，那么它的代码仅会在第一次导入时执行，然后将导出（`export`）的内容提供给所有的导入（`importer`）。

这有很重要的影响。让我们通过示例来看一下：

首先，如果执行一个模块中的代码会带来副作用（`side-effect`），例如显示一条消息，那么多次导入它只会触发一次显示 — 即第一次：

```
// ┣ alert.js
alert("Module is evaluated!");

// 在不同的文件中导入相同的模块

// ┣ 1.js
import `./alert.js`; // Module is evaluated!

// ┣ 2.js
import `./alert.js`; // (什么都不显示)
```

在实际开发中，顶级模块代码主要用于初始化，内部数据结构的创建，并且如果我们希望某些东西可以重用 — 请导出它。

下面是一个高级点的例子。

我们假设一个模块导出了一个对象：

```
// ┣ admin.js
export let admin = {
  name: "John"
};
```

如果这个模块被导入到多个文件中，模块仅在第一次被导入时被解析，并创建 `admin` 对象，然后将其传入到所有的导入。

所有的导入都只获得了一个唯一的 `admin` 对象：

```
// 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// 1.js 和 2.js 导入的是同一个对象
// 在 1.js 中对对象做的更改，在 2.js 中也是可见的
```

所以，让我们重申一下 — 模块只被执行一次。生成导出，然后它被分享给所有对其的导入，所以如果某个地方修改了 `admin` 对象，其他的模块也能看到这个修改。

这种行为让我们可以在首次导入时 **设置** 模块。我们只需要设置其属性一次，然后在进一步的导入中就都可以直接使用了。

例如，下面的 `admin.js` 模块可能提供了特定的功能，但是希望凭证（`credential`）从外部进入 `admin` 对象：

```
// admin.js
export let admin = {};

export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}
```

在 `init.js` 中 — 我们 APP 的第一个脚本，设置了 `admin.name`。现在每个位置都能看到它，包括在 `admin.js` 内部的调用。

```
// init.js
import {admin} from './admin.js';
admin.name = "Pete";
```

另一个模块也可以看到 `admin.name`：

```
// other.js
import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete

sayHi(); // Ready to serve, Pete!
```

import.meta

`import.meta` 对象包含关于当前模块的信息。

它的内容取决于其所在的环境。在浏览器环境中，它包含当前脚本的 `URL`，或者如果它是在 `HTML` 中的话，则包含当前页面的 `URL`。

```
<script type="module">
  alert(import.meta.url); // 脚本的 URL (对于内嵌脚本来说, 则是当前 HTML 页面的 URL)
</script>
```

在一个模块中, “this” 是 undefined

这是一个小功能, 但为了完整性, 我们应该提到它。

在一个模块中, 顶级 `this` 是 `undefined`。

将其与非模块脚本进行比较会发现, 非模块脚本的顶级 `this` 是全局对象:

```
<script>
  alert(this); // window
</script>

<script type="module">
  alert(this); // undefined
</script>
```

浏览器特定功能

与常规脚本相比, 拥有 `type="module"` 标识的脚本有一些特定于浏览器的差异。

如果你是第一次阅读或者你不打算在浏览器中使用 JavaScript, 那么你可以跳过本节内容。

模块脚本是延迟的

模块脚本 **总是** 被延迟的, 与 `defer` 特性 (在 [脚本: async, defer](#) 一章中描述的) 对外部脚本和内联脚本 (`inline script`) 的影响相同。

也就是说:

- 下载外部模块脚本 `<script type="module" src="...>` 不会阻塞 HTML 的处理, 它们会与其他资源并行加载。
- 模块脚本会等到 HTML 文档完全准备就绪 (即使它们很小并且比 HTML 加载速度更快), 然后才会运行。
- 保持脚本的相对顺序: 在文档中排在前面的脚本先执行。

它的一个副作用是, 模块脚本总是会“看到”已完全加载的 HTML 页面, 包括在它们下方的 HTML 元素。

例如:

```
<script type="module">
  alert(typeof button); // object: 脚本可以“看见”下面的 button
  // 因为模块是被延迟的 (deferred), 所以模块脚本会在整个页面加载完成后才运行
</script>
```

相较于下面这个常规脚本:

```
<script>
  alert(typeof button); // Error: button is undefined, 脚本看不到下面的元素
  // 常规脚本会立即运行, 常规脚本的运行是在处理页面的其余部分之前进行的
</script>
```

```
<button id="button">Button</button>
```

请注意：上面的第二个脚本实际上要先于前一个脚本运行！所以我们会先看到 `undefined`，然后才是 `object`。

这是因为模块脚本是被延迟的，所以要等到 **HTML** 文档被处理完成才会执行它。而常规脚本则会立即运行，所以我们会先看到常规脚本的输出。

当使用模块脚本时，我们应该知道 **HTML** 页面在加载时就会显示出来，在 **HTML** 页面加载完成后才会执行 **JavaScript** 模块，因此用户可能会在 **JavaScript** 应用程序准备好之前看到该页面。某些功能那时可能还无法正常使用。我们应该放置“加载指示器（loading indicator）”，否则，请确保不会使用户感到困惑。

Async 适用于内联脚本（inline script）

对于非模块脚本，`async` 特性（attribute）仅适用于外部脚本。异步脚本会在准备好后立即运行，独立于其他脚本或 **HTML** 文档。

对于模块脚本，它也适用于内联脚本。

例如，下面的内联脚本具有 `async` 特性，因此它不会等待任何东西。

它执行导入（`fetch './analytics.js'`），并在准备导入完成时运行，即使 **HTML** 文档还未完成，或者其他脚本仍在等待处理中。

这对于不依赖任何其他东西的功能来说是非常棒的，例如计数器，广告，文档级事件监听器。

```
<!-- 所有依赖都获取完成 (analytics.js) 然后脚本开始运行 -->
<!-- 不会等待 HTML 文档或者其他 <script> 标签 -->
<script async type="module">
  import {counter} from './analytics.js';

  counter.count();
</script>
```

外部脚本

具有 `type="module"` 的外部脚本（external script）在两个方面有所不同：

1. 具有相同 `src` 的外部脚本仅运行一次：

```
<!-- 脚本 my.js 被加载完成 (fetched) 并只被运行一次 -->
<script type="module" src="my.js"></script>
<script type="module" src="my.js"></script>
```

2. 从另一个源（例如另一个网站）获取的外部脚本需要 **CORS ↗ header**，如我们在 **Fetch: 跨源请求** 一章中所讲的那样。换句话说，如果一个模块脚本是从另一个源获取的，则远程服务器必须提供表示允许获取的 `header Access-Control-Allow-Origin`。

```
<!-- another-site.com 必须提供 Access-Control-Allow-Origin -->
<!-- 否则，脚本将无法执行 -->
<script type="module" src="http://another-site.com/their.js"></script>
```

默认这样做可以确保更好的安全性。

不允许裸模块 (“bare” module)

在浏览器中，`import` 必须给出相对或绝对的 URL 路径。没有任何路径的模块被称为“裸 (bare) ”模块。在 `import` 中不允许这种模块。

例如，下面这个 `import` 是无效的：

```
import {sayHi} from 'sayHi'; // Error, “裸”模块  
// 模块必须有一个路径，例如 './sayHi.js' 或者其他任何路径
```

某些环境，像 Node.js 或者打包工具（bundle tool）允许没有任何路径的裸模块，因为它们有自己的查找模块的方法和钩子（hook）来对它们进行微调。但是浏览器尚不支持裸模块。

兼容性，“nomodule”

旧时的浏览器不理解 `type="module"`。未知类型的脚本会被忽略。对此，我们可以使用 `nomodule` 特性来提供一个后备：

```
<script type="module">  
  alert("Runs in modern browsers");  
</script>  
  
<script nomodule>  
  alert("Modern browsers know both type=module and nomodule, so skip this")  
  alert("Old browsers ignore script with unknown type=module, but execute this.");  
</script>
```

构建工具

在实际开发中，浏览器模块很少被以“原始”形式进行使用。通常，我们会使用一些特殊工具，例如 [Webpack ↗](#)，将它们打包在一起，然后部署到生产环境的服务器。

使用打包工具的一个好处是 — 它们可以更好地控制模块的解析方式，允许我们使用裸模块和更多的功能，例如 CSS/HTML 模块等。

构建工具做以下这些事儿：

1. 从一个打算放在 HTML 中的 `<script type="module">` “主”模块开始。
2. 分析它的依赖：它的导入，以及它的导入的导入等。
3. 使用所有模块构建一个文件（或者多个文件，这是可调的），并用打包函数（bundler function）替代原生的 `import` 调用，以使其正常工作。还支持像 HTML/CSS 模块等“特殊”的模块类型。
4. 在处理过程中，可能会应用其他转换和优化：
 - 删除无法访问的代码。
 - 删除未使用的导出（“tree-shaking”）。
 - 删除特定于开发的像 `console` 和 `debugger` 这样的语句。
 - 可以使用 [Babel ↗](#) 将前沿的现代的 JavaScript 语法转换为具有类似功能的旧的 JavaScript 语法。
 - 压缩生成的文件（删除空格，用短的名字替换变量等）。

如果我们使用打包工具，那么脚本会被打包进一个单一文件（或者几个文件），在这些脚本中的 `import/export` 语句会被替换成特殊的打包函数（bundler function）。因此，最终打包好的脚本中不包含任何 `import/export`，它也不需要 `type="module"`，我们可以将其放入常规的 `<script>`：

```
<!-- 假设我们从诸如 Webpack 这类的打包工具中获得了 "bundle.js" 脚本 -->
<script src="bundle.js"></script>
```

也就是说，原生模块也是可以使用的。所以，我们在这儿将不会使用 Webpack：你可以稍后再配置它。

总结

下面总结一下模块的核心概念：

1. 一个模块就是一个文件。浏览器需要使用 `<script type="module">` 以使 `import/export` 可以工作。模块（译注：相较于常规脚本）有几点差别：
 - 默认是延迟解析的（deferred）。
 - `Async` 可用于内联脚本。
 - 要从另一个源（域/协议/端口）加载外部脚本，需要 CORS header。
 - 重复的外部脚本会被忽略
2. 模块具有自己的本地顶级作用域，并可以通过 `import/export` 交换功能。
3. 模块始终使用 `use strict`。
4. 模块代码只执行一次。导出仅创建一次，然后会在导入之间共享。

当我们使用模块时，每个模块都会实现特定功能并将其导出。然后我们使用 `import` 将其直接导入到需要的地方即可。浏览器会自动加载并解析脚本。

在生产环境中，出于性能和其他原因，开发者经常使用诸如 [Webpack ↗](#) 之类的打包工具将模块打包到一起。

在下一章里，我们将会看到更多关于模块的例子，以及如何进行导入/导出。

导出和导入

导出（`export`）和导入（`import`）指令有几种语法变体。

在上一章，我们看到了一个简单的用法，现在让我们来探索更多示例吧。

在声明前导出

我们可以通过在声明之前放置 `export` 来标记任意声明为导出，无论声明的是变量，函数还是类都可以。

例如，这里的所有导出均有效：

```
// 导出数组
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

```
// 导出 const 声明的变量
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// 导出类
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

❶ 导出 class/function 后没有分号

注意，在类或者函数前的 `export` 不会让它们变成 [函数表达式](#)。尽管被导出了，但它仍然是一个函数声明。

大部分 JavaScript 样式指南都不建议在函数和类声明后使用分号。

这就是为什么在 `export class` 和 `export function` 的末尾不需要加分号：

```
export function sayHi(user) {
  alert(`Hello, ${user}!`);
} // 在这里没有分号；
```

导出与声明分开

另外，我们还可以将 `export` 分开放置。

下面的例子中，我们先声明函数，然后再导出它们：

```
// □ say.js
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export {sayHi, sayBye}; // 导出变量列表
```

.....从技术上讲，我们也可以把 `export` 放在函数上面。

Import *

通常，我们把要导入的东西列在花括号 `import {...}` 中，就像这样：

```
// □ main.js
import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
```

但是如果有很多要导入的内容，我们可以使用 `import * as <obj>` 将所有内容导入为一个对象，例如：

```
// □ main.js
import * as say from './say.js';

say.sayHi('John');
say.sayBye('John');
```

乍一看，“通通导入”看起来很酷，写起来也很短，但是我们通常为什么要明确列出我们需要导入的内容？

这里有几个原因。

1. 现代的构建工具（[webpack](#) 和其他工具）将模块打包到一起并对其进行优化，以加快加载速度并删除未使用的代码。

比如说，我们向我们的项目里添加一个第三方库 `say.js`，它具有许多函数：

```
// □ say.js
export function sayHi() { ... }
export function sayBye() { ... }
export function becomeSilent() { ... }
```

现在，如果我们只在我们的项目里使用了 `say.js` 中的一个函数：

```
// □ main.js
import {sayHi} from './say.js';
```

.....那么，优化器（optimizer）就会检测到它，并从打包好的代码中删除那些未被使用的函数，从而使构建更小。这就是所谓的“摇树（tree-shaking）”。

2. 明确列出要导入的内容会使得名称较短：`sayHi()` 而不是 `say.sayHi()`。
3. 导入的显式列表可以更好地概述代码结构：使用的内容和位置。它使得代码支持重构，并且重构起来更容易。

Import “as”

我们也可以使用 `as` 让导入具有不同的名字。

例如，简洁起见，我们将 `sayHi` 导入到局部变量 `hi`，将 `sayBye` 导入到 `bye`：

```
// □ main.js
import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!
bye('John'); // Bye, John!
```

Export “as”

导出也具有类似的语法。

我们将函数导出为 `hi` 和 `bye`：

```
// □ say.js
...
export {sayHi as hi, sayBye as bye};
```

现在 `hi` 和 `bye` 是在外面使用时的正式名称：

```
// □ main.js
import * as say from './say.js';

say.hi('John'); // Hello, John!
say.bye('John'); // Bye, John!
```

Export default

在实际中，主要有两种模块。

- 包含库或函数包的模块，像上面的 `say.js`。
- 声名单个实体的模块，例如模块 `user.js` 仅导出 `class User`。

大部分情况下，开发者倾向于使用第二种方式，以便每个“东西”都存在于它自己的模块中。

当然，这需要大量文件，因为每个东西都需要自己的模块，但这根本不是问题。实际上，如果文件具有良好的命名，并且文件夹结构得当，那么代码导航（navigation）会变得更容易。

模块提供了特殊的默认导出 `export default` 语法，以使“一个模块只做一件事”的方式看起来更好。

将 `export default` 放在要导出的实体前：

```
// □ user.js
export default class User { // 只需要添加 "default" 即可
  constructor(name) {
    this.name = name;
  }
}
```

每个文件可能只有一个 `export default`：

.....然后将其导入而不需要花括号：

```
// □ main.js
import User from './user.js'; // 不需要花括号 {User}，只需要写成 User 即可

new User('John');
```

不用花括号的导入看起来很酷。刚开始使用模块时，一个常见的错误就是忘记写花括号。所以，请记住，`import` 命名的导出时需要花括号，而 `import` 默认的导出时不需要花括号。

命名的导出

```
export class User {...}
```

```
import {User} from ...
```

默认的导出

```
export default class User {...}
```

```
import User from ...
```

从技术上讲，我们可以在一个模块中同时有默认的导出和命名的导出，但是实际上人们通常不会混合使用它们。模块要么是命名的导出要么是默认的导出。

由于每个文件最多只能有一个默认的导出，因此导出的实体可能没有名称。

例如，下面这些都是完全有效的默认的导出：

```
export default class { // 没有类名
  constructor() { ... }
}
```

```
export default function(user) { // 没有函数名
  alert(`Hello, ${user}!`);
}
```

```
// 导出单个值，而不使用变量
export default ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

不指定名称是可以的，因为每个文件只有一个 `export default`，因此不带花括号的 `import` 知道要导入的内容是什么。

如果没有 `default`，这样的导出将会出错：

```
export class { // Error! (非默认的导出需要名称)
  constructor() {}
}
```

“`default`” 名称

在某些情况下，`default` 关键词被用于引用默认的导出。

例如，要将函数与其定义分开导出：

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// 就像我们在函数之前添加了 "export default" 一样
export {sayHi as default};
```

或者，另一种情况，假设模块 `user.js` 导出了一个主要的默认的导出和一些命名的导出（虽然很少出现，但是会发生）：

```
// ┣ user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

这是导入默认的导出以及命名的导出的方法：

```
// ┣ main.js
import {default as User, sayHi} from './user.js';

new User('John');
```

如果我们将所有东西 `*` 作为一个对象导入，那么 `default` 属性正是默认的导出：

```
// ┣ main.js
import * as user from './user.js';

let User = user.default; // 默认的导出
new User('John');
```

我应该使用默认的导出吗？

命名的导出是明确的。它们确切地命名了它们要导出的内容，因此我们能从它们获得这些信息，这是一件好事。

命名的导出会强制我们使用正确的名称进行导入：

```
import {User} from './user.js';
// 导入 {MyUser} 不起作用，导入名字必须为 {User}
```

.....对于默认的导出，我们总是在导入时选择名称：

```
import User from './user.js'; // 有效
import MyUser from './user.js'; // 也有效
// 使用任何名称导入都没有问题
```

因此，团队成员可能会使用不同的名称来导入相同的内容，这不好。

通常，为了避免这种情况并使代码保持一致，可以遵从这条规则，即导入的变量应与文件名相对应，例如：

```
import User from './user.js';
import LoginForm from './loginForm.js';
import func from '/path/to/func.js';
...
```

但是，一些团队仍然认为这是默认的导出的严重缺陷。因此，他们更倾向于始终使用命名的导出。即使只导出一个东西，也仍然使用命名的导出，而不是默认的导出。

这也使得重新导出（见下文）更容易。

重新导出

“重新导入（Re-export）”语法 `export ... from ...` 允许导入内容，并立即将其导出（可能是用的是其他的名字），就像这样：

```
export {sayHi} from './say.js'; // 重新导出 sayHi
export {default as User} from './user.js'; // 重新导出 default
```

为什么要这样做？我们看一个实际开发中的用例。

想象一下，我们正在编写一个“package”：一个包含大量模块的文件夹，其中一些功能是导出到外部的（像 NPM 这样的工具允许发布和分发这样的 package），并且其中一些模块仅仅是供其他 package 中的模块内部使用的“helpers”。

文件结构可能是这样的：

```
auth/
  index.js
  user.js
  helpers.js
  tests/
    login.js
  providers/
    github.js
    facebook.js
  ...
...
```

我们想通过单个入口，即“主文件” `auth/index.js` 来公开 package 的功能，进而可以像下面这样使用我们的 package：

```
import {login, logout} from 'auth/index.js'
```

我们的想法是，使用我们 package 的开发者，不应该干预其内部结构，不应该搜索我们 package 的文件夹中的文件。我们只在 `auth/index.js` 中导出必须的内容，并保持其他内容“不可见”。

由于实际导出的功能分散在 package 中，所以我们可以将它们导入到 `auth/index.js`，然后再从中导出它们：

```
// auth/index.js

// 导入 login/logout 然后立即导出它们
import {login, logout} from './helpers.js';
export {login, logout};

// 将默认导出导入为 User, 然后导出它
import User from './user.js';
export {User};

...
```

现在使用我们 package 的人可以 `import {login} from "auth/index.js"`。

语法 `export ... from ...` 只是下面这种导入-导出的简写:

```
// auth/index.js

// 导入 login/logout 然后立即导出它们
export {login, logout} from './helpers.js';

// 将默认导出导入为 User, 然后导出它
export {default as User} from './user.js';
...
```

重新导出默认导出

重新导出时， 默认导出需要单独处理。

假设我们有 `user.js`， 我们想从中重新导出类 `User`:

```
// user.js
export default class User {
  // ...
}
```

1. `export User from './user.js'` 无效。什么出了问题? 这实际上是一个语法错误。

要重新导出默认导出，我们必须明确写出 `export {default as User}`，就像上面的例子中那样。

2. `export * from './user.js'` 重新导出只导出了命名的导出，但是忽略了默认的导出。

如果我们想将命名的导出和默认的导出都重新导出，那么需要两条语句:

```
export * from './user.js'; // 重新导出命名的导出
export {default} from './user.js'; // 重新导出默认的导出
```

重新导出默认的导出的这种奇怪现象是某些开发者不喜欢它们的原因之一。

总结

这是我们在本章和前面章节中介绍的所有 `export` 类型:

你可以阅读并回忆它们的含义来进行自查：

- 在声明一个 class/function/... 之前：
 - `export [default] class/function/variable ...`
- 独立的导出：
 - `export {x [as y], ...}.`
- 重新导出：
 - `export {x [as y], ...} from "module"`
 - `export * from "module"` (不会重新导出默认的导出)。
 - `export {default [as y]} from "module"` (重新导出默认的导出)。

导入：

- 模块中命名的导出：
 - `import {x [as y], ...} from "module"`
- 默认的导出：
 - `import x from "module"`
 - `import {default as x} from "module"`
- 所有：
 - `import * as obj from "module"`
- 导入模块（它的代码，并运行），但不要将其赋值给变量：
 - `import "module"`

我们把 `import/export` 语句放在脚本的顶部或底部，都没关系。

因此，从技术上讲，下面这样的代码没有问题：

```
sayHi();  
// ...  
import {sayHi} from './say.js'; // 在文件底部导入
```

在实际开发中，导入通常位于文件的开头，但是这只是为了更加方便。

请注意在 `{...}` 中的 `import/export` 语句无效。

像这样的有条件的导入是无效的：

```
if (something) {  
  import {sayHi} from './say.js'; // Error: import must be at top level  
}
```

.....但是，如果我们真的需要根据某些条件来进行导入呢？或者在某些合适的时间？例如，根据请求 (`request`) 加载模块，什么时候才是真正需要呢？

我们将在下一章中学习动态导入。

动态导入

我们在前面章节中介绍的导出和导入语句称为“静态”导入。语法非常简单且严格。

首先，我们不能动态生成 `import` 的任何参数。

模块路径必须是原始类型字符串，不能是函数调用，下面这样的 `import` 行不通：

```
import ... from getModuleName(); // Error, only from "string" is allowed
```

其次，我们无法根据条件或者在运行时导入：

```
if(...) {
  import ...; // Error, not allowed!
}

{
  import ...; // Error, we can't put import in any block
}
```

这是因为 `import/export` 旨在提供代码结构的主干。这是非常好的事儿，因为这样便于分析代码结构，可以收集模块，可以使用特殊工具将收集的模块打包到一个文件中，可以删除未使用的导出（“tree-shaken”）。这些只有在 `import/export` 结构简单且固定的情况下才能够实现。

但是，我们如何才能动态地按需导入模块呢？

`import()` 表达式

`import(module)` 表达式加载模块并返回一个 `promise`，该 `promise` `resolve` 为一个包含其所有导出的模块对象。我们可以在代码中的任意位置调用这个表达式。

我们可以在代码中的任意位置动态地使用它。例如：

```
let modulePath = prompt("Which module to load?");

import(modulePath)
  .then(obj => <module object>)
  .catch(err => <loading error, e.g. if no such module>)
```

或者，如果在异步函数中，我们可以使用 `let module = await import(modulePath)`。

例如，如果我们有以下模块 `say.js`：

```
// ┣ say.js
export function hi() {
  alert(`Hello`);
}

export function bye() {
  alert(`Bye`);
}
```

.....那么，可以想像下面这样进行动态导入:

```
let {hi, bye} = await import('./say.js');

hi();
bye();
```

或者，如果 `say.js` 有默认的导出:

```
// ┣ say.js
export default function() {
  alert("Module loaded (export default)!");
}
```

.....那么，为了访问它，我们可以使用模块对象的 `default` 属性:

```
let obj = await import('./say.js');
let say = obj.default;
// or, in one line: let {default: say} = await import('./say.js');

say();
```

这是一个完整的示例:

[https://plnkr.co/edit/gP63g9Nd6NMRkSwR?p=preview ↗](https://plnkr.co/edit/gP63g9Nd6NMRkSwR?p=preview)

i **请注意:**

动态导入在常规脚本中工作时，它们不需要 `script type="module"` .

i **请注意:**

尽管 `import()` 看起来像一个函数调用，但它只是一种特殊语法，只是恰好使用了括号（类似于 `super()`）。

因此，我们不能将 `import` 复制到一个变量中，或者对其使用 `call/apply` 。因为它不是一个函数。

杂项

Proxy 和 Reflect

一个 `Proxy` 对象包装另一个对象并拦截诸如读取/写入属性和其他操作，可以选择自行处理它们，或者透明地允许该对象处理它们。

`Proxy` 被用于了许多库和某些浏览器框架。在本文中，我们将看到许多实际应用。

Proxy

语法:

```
let proxy = new Proxy(target, handler)
```

- `target` — 是要包装的对象，可以是任何东西，包括函数。
- `handler` — 代理配置：带有“陷阱”（“traps”，即拦截操作的方法）的对象。比如 `get` 陷阱用于读取 `target` 的属性，`set` 陷阱用于写入 `target` 的属性，等等。

对 `proxy` 进行操作，如果在 `handler` 中存在相应的陷阱，则它将运行，并且 `Proxy` 有机会对其进行处理，否则将直接对 `target` 进行处理。

首先，让我们创建一个没有任何陷阱的代理（`proxy`）：

```
let target = {};
let proxy = new Proxy(target, {}); // 空的 handler 对象

proxy.test = 5; // 写入 proxy 对象 (1)
alert(target.test); // 5, test 属性出现在了 target 中!

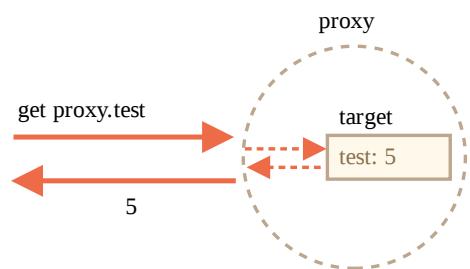
alert(proxy.test); // 5, 我们也可以从 proxy 对象读取它 (2)

for(let key in proxy) alert(key); // test, 迭代也正常工作 (3)
```

由于没有陷阱，所有对 `proxy` 的操作都直接转发给了 `target`。

1. 写入操作 `proxy.test=` 会将值写入 `target`。
2. 读取操作 `proxy.test` 会从 `target` 返回对应的值。
3. 迭代 `proxy` 会从 `target` 返回对应的值。

我们可以看到，没有任何陷阱，`proxy` 是一个 `target` 的透明包装器（wrapper）。



`Proxy` 是一种特殊的“奇异对象（exotic object）”。它没有自己的属性。如果 `handler` 为空，则透明地将操作转发给 `target`。

要激活更多功能，让我们添加陷阱。

我们可以用它们拦截什么？

对于对象的大多数操作，JavaScript 规范中有一个所谓的“内部方法”，它描述了最底层的工作方式。例如 `[[Get]]`，用于读取属性的内部方法，`[[Set]]`，用于写入属性的内部方法，等等。这些方法仅在规范中使用，我们不能直接通过方法名调用它们。

`Proxy` 陷阱会拦截这些方法的调用。它们在 [proxy 规范 ↗](#) 和下表中被列出。

对于每个内部方法，此表中都有一个陷阱：可用于添加到 `new Proxy` 的 `handler` 参数中以拦截操作的方法名称：

内部方法	Handler 方法	何时触发
<code>[[Get]]</code>	<code>get</code>	读取属性
<code>[[Set]]</code>	<code>set</code>	写入属性
<code>[[HasProperty]]</code>	<code>has</code>	<code>in</code> 操作符
<code>[[Delete]]</code>	<code>deleteProperty</code>	<code>delete</code> 操作符
<code>[[Call]]</code>	<code>apply</code>	函数调用
<code>[[Construct]]</code>	<code>construct</code>	<code>new</code> 操作符
<code>[[GetPrototypeOf]]</code>	<code>getPrototypeOf</code>	Object.getPrototypeOf ↗
<code>[[SetPrototypeOf]]</code>	<code>setPrototypeOf</code>	Object.setPrototypeOf ↗
<code>[[IsExtensible]]</code>	<code>isExtensible</code>	Object.isExtensible ↗
<code>[[PreventExtensions]]</code>	<code>preventExtensions</code>	Object.preventExtensions ↗
<code>[[DefineOwnProperty]]</code>	<code>defineProperty</code>	Object.defineProperty ↗ , Object.defineProperties ↗
<code>[[GetOwnProperty]]</code>	<code>getOwnPropertyDescriptor</code>	Object.getOwnPropertyDescriptor ↗ , <code>for..in</code> , <code>Object.keys/values/entries</code>
<code>[[OwnPropertyKeys]]</code>	<code>ownKeys</code>	Object.getOwnPropertyNames ↗ , Object.getOwnPropertySymbols ↗ , <code>for..in</code> , <code>Object/keys/values/entries</code>

⚠ 不变量 (Invariant)

JavaScript 强制执行某些不变量 — 内部方法和陷阱必须满足的条件。。

其中大多数用于返回值：

- `[[Set]]` 如果值已成功写入，则必须返回 `true`，否则返回 `false`。
- `[[Delete]]` 如果已成功删除该值，则必须返回 `true`，否则返回 `false`。
-依此类推，我们将在下面的示例中看到更多内容。

还有其他一些不变量，例如：

- 应用于代理 (proxy) 对象的 `[[GetPrototypeOf]]`，必须返回与应用于被代理对象的 `[[GetPrototypeOf]]` 相同的值。换句话说，读取代理对象的原型必须始终返回被代理对象的原型。

陷阱可以拦截这些操作，但是必须遵循下面这些规则。

不变量确保语言功能的正确和一致的行为。完整的不变量列表在 [规范 ↗](#) 中。如果你不做奇怪的事情，你可能就不会违反它们。

让我们来看看它们是如何在实际示例中工作的。

带有“get”陷阱的默认值

最常见的陷阱是用于读取/写入的属性。

要拦截读取操作，`handler` 应该有 `get(target, property, receiver)` 方法。

读取属性时触发该方法，参数如下：

- `target` — 是目标对象，该对象被作为第一个参数传递给 `new Proxy`，
- `property` — 目标属性名，
- `receiver` — 如果目标属性是一个 getter 访问器属性，则 `receiver` 就是本次读取属性所在的 `this` 对象。通常，这就是 `proxy` 对象本身（或者，如果我们从 `proxy` 继承，则是从该 `proxy` 继承的对象）。现在我们不需要此参数，因此稍后我们将对其进行详细介绍。

让我们用 `get` 来实现一个对象的默认值。

我们将创建一个对不存在的数组项返回 `0` 的数组。

通常，当人们尝试获取不存在的数组项时，他们会得到 `undefined`，但是我们在这将常规数组包装到代理（`proxy`）中，以捕获读取操作，并在没有要读取的属性时返回 `0`：

```
let numbers = [0, 1, 2];

numbers = new Proxy(numbers, {
  get(target, prop) {
    if (prop in target) {
      return target[prop];
    } else {
      return 0; // 默认值
    }
  }
});

alert( numbers[1] ); // 1
alert( numbers[123] ); // 0 (没有这个数组项)
```

正如我们所看到的，使用 `get` 陷阱很容易实现。

我们可以用 `Proxy` 来实现“默认”值的任何逻辑。

想象一下，我们有一本词典，上面有短语及其翻译：

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome'] ); // undefined
```

现在，如果没有我们要读取的短语，那么从 `dictionary` 读取它将返回 `undefined`。但实际上，返回一个未翻译的短语通常比 `undefined` 要好。因此，让我们在这种情况下返回一个未翻译的短语来替代 `undefined`。

为此，我们将把 `dictionary` 包装进一个拦截读取操作的代理：

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

dictionary = new Proxy(dictionary, {
  get(target, phrase) { // 拦截读取属性操作
    if (phrase in target) { // 如果词典中有该短语
      return target[phrase]; // 返回其翻译
    } else {
      // 否则返回未翻译的短语
      return phrase;
    }
  }
});

// 在词典中查找任意短语！
// 最坏的情况也只是它们没有被翻译。
alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome to Proxy'] ); // Welcome to Proxy (没有被翻译)
```

➊ 请注意:

请注意代理如何覆盖变量:

```
dictionary = new Proxy(dictionary, ...);
```

代理应该在所有地方都完全替代目标对象。目标对象被代理后，任何人都不应该再引用目标对象。否则很容易搞砸。

使用“`set`”陷阱进行验证

假设我们想要一个专门用于数字的数组。如果添加了其他类型的值，则应该抛出一个错误。

当写入属性时 `set` 陷阱被触发。

```
set(target, property, value, receiver):
```

- `target` — 是目标对象，该对象被作为第一个参数传递给 `new Proxy`，
- `property` — 目标属性名称，
- `value` — 目标属性的值，
- `receiver` — 与 `get` 陷阱类似，仅与 `setter` 访问器属性相关。

如果写入操作（`setting`）成功，`set` 陷阱应该返回 `true`，否则返回 `false`（触发 `TypeError`）。

让我们用它来验证新值:

```
let numbers = [];

numbers = new Proxy(numbers, { /* */
  set(target, prop, val) { // 拦截写入属性操作
```

```

if (typeof val == 'number') {
  target[prop] = val;
  return true;
} else {
  return false;
}
});

numbers.push(1); // 添加成功
numbers.push(2); // 添加成功
alert("Length is: " + numbers.length); // 2

numbers.push("test"); // TypeError (proxy 的 'set' 返回 false)

alert("This line is never reached (error in the line above)");

```

请注意：数组的内建方法依然有效！值被使用 `push` 方法添加到数组。当值被添加到数组后，数组的 `length` 属性会自动增加。我们的代理对象 `proxy` 不会破坏任何东西。

我们不必重写诸如 `push` 和 `unshift` 等添加元素的数组方法，就可以在其中添加检查，因为在内部它们使用代理所拦截的 `[[Set]]` 操作。

因此，代码简洁明了。

⚠ 别忘了返回 `true`

如上所述，要保持不变量。

对于 `set` 操作，它必须在成功写入时返回 `true`。

如果我们忘记这样做，或返回任何假（`falsy`）值，则该操作将触发 `TypeError`。

使用“`ownKeys`”和“`getOwnPropertyDescriptor`”进行迭代

`Object.keys`，`for..in` 循环和大多数其他遍历对象属性的方法都使用内部方法 `[[OwnPropertyKeys]]`（由 `ownKeys` 陷阱拦截）来获取属性列表。

这些方法在细节上有所不同：

- `Object.getOwnPropertyNames(obj)` 返回非 `Symbol` 键。
- `Object.getOwnPropertySymbols(obj)` 返回 `symbol` 键。
- `Object.keys/values()` 返回带有 `enumerable` 标志的非 `Symbol` 键/值（属性标志在 [属性标志和属性描述符](#) 一章有详细讲解）。
- `for..in` 循环遍历所有带有 `enumerable` 标志的非 `Symbol` 键，以及原型对象的键。

.....但是所有这些都从该列表开始。

在下面这个示例中，我们使用 `ownKeys` 陷阱拦截 `for..in` 对 `user` 的遍历，并使用 `Object.keys` 和 `Object.values` 来跳过以下划线 `_` 开头的属性：

```

let user = {
  name: "John",
  age: 30,

```

```

    _password: "****"
};

user = new Proxy(user, {
  ownKeys(target) {
    return Object.keys(target).filter(key => !key.startsWith('_'));
  }
});

// "ownKeys" 过滤掉了 _password
for(let key in user) alert(key); // name, 然后是 age

// 对这些方法的效果相同:
alert( Object.keys(user) ); // name, age
alert( Object.values(user) ); // John, 30

```

到目前为止，它仍然有效。

尽管如此，但如果我们返回对象中不存在的键，`Object.keys` 并不会列出这些键：

```

let user = { };

user = new Proxy(user, {
  ownKeys(target) {
    return ['a', 'b', 'c'];
  }
});

alert( Object.keys(user) ); // <empty>

```

为什么？原因很简单：`Object.keys` 仅返回带有 `enumerable` 标志的属性。为了检查它，该方法会对每个属性调用内部方法 `[[GetOwnProperty]]` 来获取 [它的描述符（descriptor）](#)。在这里，由于没有属性，其描述符为空，没有 `enumerable` 标志，因此它被略过。

为了让 `Object.keys` 返回一个属性，我们要么需要它要么存在于带有 `enumerable` 标志的对象，要么我们可以拦截对 `[[GetOwnProperty]]` 的调用（陷阱 `getOwnPropertyDescriptor` 可以做到这一点），并返回带有 `enumerable: true` 的描述符。

这是关于此的一个例子：

```

let user = { };

user = new Proxy(user, {
  ownKeys(target) { // 一旦要获取属性列表就会被调用
    return ['a', 'b', 'c'];
  },

  getOwnPropertyDescriptor(target, prop) { // 被每个属性调用
    return {
      enumerable: true,
      configurable: true
      /* ...其他标志，可能是 "value:..." */
    };
  }
}

```

```
});

alert( Object.keys(user) ); // a, b, c
```

让我们再次注意：如果该属性在对象中不存在，那么我们只需要拦截 `[[GetOwnProperty]]`。

具有“`deleteProperty`”和其他陷阱的受保护属性

有一个普遍的约定，即以下划线 `_` 开头的属性和方法是内部的。不应从对象外部访问它们。

从技术上讲，我们也是能访问到这样的属性的：

```
let user = {
  name: "John",
  _password: "secret"
};

alert(user._password); // secret
```

让我们使用代理来防止对以 `_` 开头的属性的任何访问。

我们将需要以下陷阱：

- `get` 读取此类属性时抛出错误，
- `set` 写入属性时抛出错误，
- `deleteProperty` 删除属性时抛出错误，
- `ownKeys` 在使用 `for..in` 和像 `Object.keys` 这样的的方法时排除以 `_` 开头的属性。

代码如下：

```
let user = {
  name: "John",
  _password: "****"
};

user = new Proxy(user, {
  get(target, prop) {
    if (prop.startsWith('_')) {
      throw new Error("Access denied");
    }
    let value = target[prop];
    return (typeof value === 'function') ? value.bind(target) : value; // (*)
  },
  set(target, prop, val) { // 拦截属性写入
    if (prop.startsWith('_')) {
      throw new Error("Access denied");
    } else {
      target[prop] = val;
      return true;
    }
  },
  deleteProperty(target, prop) { // 拦截属性删除
    if (prop.startsWith('_')) {
```

```

        throw new Error("Access denied");
    } else {
        delete target[prop];
        return true;
    }
},
ownKeys(target) { // 拦截读取属性列表
    return Object.keys(target).filter(key => !key.startsWith('__'));
}
});

// "get" 不允许读取 _password
try {
    alert(user._password); // Error: Access denied
} catch(e) { alert(e.message); }

// "set" 不允许写入 _password
try {
    user._password = "test"; // Error: Access denied
} catch(e) { alert(e.message); }

// "deleteProperty" 不允许删除 _password
try {
    delete user._password; // Error: Access denied
} catch(e) { alert(e.message); }

// "ownKeys" 将 _password 过滤出去
for(let key in user) alert(key); // name

```

请注意在 (*) 行中 `get` 陷阱的重要细节：

```

get(target, prop) {
    // ...
    let value = target[prop];
    return (typeof value === 'function') ? value.bind(target) : value; // (*)
}

```

为什么我们需要一个函数去调用 `value.bind(target)`？

原因是对象方法（例如 `user.checkPassword()`）必须能够访问 `_password`：

```

user = {
    // ...
    checkPassword(value) {
        // 对象方法必须能读取 _password
        return value === this._password;
    }
}

```

对 `user.checkPassword()` 的调用会调用被代理的对象 `user` 作为 `this`（点符号之前的对象会成为 `this`），因此，当它尝试访问 `this._password` 时，`get` 陷阱将激活（在任何属性读取时，它都会被触发）并抛出错误。

因此，我们在 (*) 行中将对象方法的上下文绑定到原始对象 `target`。然后，它们将来的调用将使用 `target` 作为 `this`，不会触发任何陷阱。

该解决方案通常可行，但并不理想，因为一个方法可能会将未被代理的对象传递到其他地方，然后我们就会陷入困境：原始对象在哪里，被代理的对象在哪里？

此外，一个对象可能会被代理多次（多个代理可能会对该对象添加不同的“调整”），并且如果我们未包装的对象传递给方法，则可能会产生意想不到的后果。

因此，在任何地方都不应使用这种代理。

① 类的私有属性

现代 Javascript 引擎原生支持 `class` 中的私有属性，这些私有属性以 `#` 为前缀。它们在 [私有的和受保护的属性和方法](#) 一章中有详细描述。无需代理（proxy）。

但是，此类属性有其自身的问题。特别是，它们是不可继承的。

带有“has”陷阱的“in range”

让我们来看更多示例。

我们有一个 `range` 对象：

```
let range = {  
  start: 1,  
  end: 10  
};
```

我们想使用 `in` 操作符来检查一个数字是否在 `range` 范围内。

`has` 陷阱会拦截 `in` 调用。

`has(target, property)`

- `target` — 是目标对象，被作为第一个参数传递给 `new Proxy`，
- `property` — 属性名称

示例如下

```
let range = {  
  start: 1,  
  end: 10  
};  
  
range = new Proxy(range, {  
  has(target, prop) {  
    return prop >= target.start && prop <= target.end;  
  }  
});  
  
alert(5 in range); // true  
alert(50 in range); // false
```

漂亮的语法糖，不是吗？而且实现起来非常简单。

包装函数: "apply"

我们也可以将代理（proxy）包装在函数周围。

`apply(target, thisArg, args)` 陷阱能使代理以函数的方式被调用：

- `target` 是目标对象（在 JavaScript 中，函数就是一个对象），
- `thisArg` 是 `this` 的值。
- `args` 是参数列表。

例如，让我们回忆一下我们在 [装饰者模式和转发，call/apply](#) 一章中所讲的 `delay(f, ms)` 装饰器。

在该章中，我们没有用 `proxy` 来实现它。调用 `delay(f, ms)` 会返回一个函数，该函数会在 `ms` 毫秒后把所有调用转发给 `f`。

这是以前的基于函数的实现：

```
function delay(f, ms) {
  // 返回一个包装器 (wrapper)，该包装器将在时间到了的时候将调用转发给函数 f
  return function() { // (*)
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// 在进行这个包装后，sayHi 函数会被延迟 3 秒后被调用
sayHi = delay(sayHi, 3000);

sayHi("John"); // Hello, John! (after 3 seconds)
```

正如我们所看到的那样，大多数情况下它都是可行的。包装函数 `(*)` 在到达延迟的时间后后执行调用。

但是包装函数不会转发属性读取/写入操作或者任何其他操作。进行包装后，就失去了对原始函数属性的访问，例如 `name`, `length` 和其他属性：

```
function delay(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(user) {
  alert(`Hello, ${user}!`);
}

alert(sayHi.length); // 1 (函数的 length 是函数声明中的参数个数)

sayHi = delay(sayHi, 3000);
```

```
alert(sayHi.length); // 0 (在包装器声明中，参数个数为 0)
```

Proxy 的功能要强大得多，因为它可以将所有东西转发到目标对象。

让我们使用 **Proxy** 来替换掉包装函数：

```
function delay(f, ms) {
  return new Proxy(f, {
    apply(target, thisArg, args) {
      setTimeout(() => target.apply(thisArg, args), ms);
    }
  });
}

function sayHi(user) {
  alert(`Hello, ${user}!`);
}

sayHi = delay(sayHi, 3000);

alert(sayHi.length); // 1 (*) proxy 将“获取 length”的操作转发给目标对象

sayHi("John"); // Hello, John! (3 秒后)
```

结果是相同的，但现在不仅仅调用，而且代理上的所有操作都能被转发到原始函数。所以在 (*) 行包装后的 **sayHi.length** 会返回正确的结果。

我们得到了一个“更丰富”的包装器。

还存在其他陷阱：完整列表在本文的开头。它们的使用模式与上述类似。

Reflect

Reflect 是一个内建对象，可简化 **Proxy** 的创建。

前面所讲过的内部方法，例如 **[[Get]]** 和 **[[Set]]** 等，都只是规范性的，不能直接调用。

Reflect 对象使调用这些内部方法成为了可能。它的方法是内部方法的最小包装。

以下是执行相同操作和 **Reflect** 调用的示例：

操作	Reflect 调用	内部方法
obj[prop]	Reflect.get(obj, prop)	[[Get]]
obj[prop] = value	Reflect.set(obj, prop, value)	[[Set]]
delete obj[prop]	Reflect.deleteProperty(obj, prop)	[[Delete]]
new F(value)	Reflect.construct(F, value)	[[Construct]]
...

例如：

```
let user = {};  
  
Reflect.set(user, 'name', 'John');  
  
alert(user.name); // John
```

尤其是，`Reflect` 允许我们将操作符（`new`, `delete`,）作为函数（`Reflect.construct`, `Reflect.deleteProperty`,）执行调用。这是一个有趣的功能，但是这里还有一点很重要。

对于每个可被 `Proxy` 捕获的内部方法，在 `Reflect` 中都有一个对应的方法，其名称和参数与 `Proxy` 陷阱相同。

所以，我们可以使用 `Reflect` 来将操作转发给原始对象。

在下面这个示例中，陷阱 `get` 和 `set` 均透明地（好像它们都不存在一样）将读取/写入操作转发到对象，并显示一条消息：

```
let user = {  
  name: "John",  
};  
  
user = new Proxy(user, {  
  get(target, prop, receiver) {  
    alert(`GET ${prop}`);  
    return Reflect.get(target, prop, receiver); // (1)  
  },  
  set(target, prop, val, receiver) {  
    alert(`SET ${prop}=${val}`);  
    return Reflect.set(target, prop, val, receiver); // (2)  
  }  
});  
  
let name = user.name; // 显示 "GET name"  
user.name = "Pete"; // 显示 "SET name=Pete"
```

这里：

- `Reflect.get` 读取一个对象属性。
- `Reflect.set` 写入一个对象属性，如果写入成功则返回 `true`，否则返回 `false`。

这样，一切都很简单：如果一个陷阱想要将调用转发给对象，则只需使用相同的参数调用 `Reflect.<method>` 就足够了。

在大多数情况下，我们可以不使用 `Reflect` 完成相同的事情，例如，用于读取属性的 `Reflect.get(target, prop, receiver)` 可以被替换为 `target[prop]`。尽管有一些细微的差别。

代理一个 getter

让我们看一个示例，来说明为什么 `Reflect.get` 更好。此外，我们还将看到为什么 `get/set` 有第四个参数 `receiver`，而且我们之前从来没有使用过它。

我们有一个带有 `_name` 属性和 `getter` 的对象 `user`。

这是对 `user` 对象对一个代理（proxy）：

```

let user = {
  _name: "Guest",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop];
  }
});

alert(userProxy.name); // Guest

```

其 `get` 陷阱在这里是“透明的”，它返回原来的属性，不会做任何其他的事。这对于我们的示例而言就足够了。

一切似乎都很好。但是让我们将示例变得稍微复杂一点。

另一个对象 `admin` 从 `user` 继承后，我们可以观察到错误的行为：

```

let user = {
  _name: "Guest",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop]; // (*) target = user
  }
});

let admin = {
  __proto__: userProxy,
  _name: "Admin"
};

// 期望输出: Admin
alert(admin.name); // 输出: Guest (?!?)

```

读取 `admin.name` 应该返回 `"Admin"`，而不是 `"Guest"`！

发生了什么？或许我们在继承方面做错了什么？

但是，如果我们移除代理，那么一切都会按预期进行。

问题实际上出在代理中，在 `(*)` 行。

1. 当我们读取 `admin.name` 时，由于 `admin` 对象自身没有对应的属性，搜索将转到其原型。
2. 原型是 `userProxy`。

3. 从代理读取 `name` 属性时，`get` 陷阱会被触发，并从原始对象返回 `target[prop]` 属性，在 (*) 行。

当调用 `target[prop]` 时，若 `prop` 是一个 `getter`，它将在 `this=target` 上下文中运行其代码。因此，结果是来自原始对象 `target` 的 `this._name`，即来自 `user`。

为了解决这种情况，我们需要 `get` 陷阱的第三个参数 `receiver`。它保证将正确的 `this` 传递给 `getter`。在我们的例子中是 `admin`。

如何把上下文传递给 `getter`？对于一个常规函数，我们可以使用 `call/apply`，但这是一个 `getter`，它不能“被调用”，只能被访问。

`Reflect.get` 可以做到。如果我们使用它，一切都会正常运行。

这是更正后的变体：

```
let user = {
  _name: "Guest",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) { // receiver = admin
    return Reflect.get(target, prop, receiver); // (*)
  }
});

let admin = {
  __proto__: userProxy,
  _name: "Admin"
};

alert(admin.name); // Admin
```

现在 `receiver` 保留了对正确 `this` 的引用（即 `admin`），该引用是在 (*) 行中被通过 `Reflect.get` 传递给 `getter` 的。

我们可以把陷阱重写得更短：

```
get(target, prop, receiver) {
  return Reflect.get(...arguments);
}
```

`Reflect` 调用的命名与陷阱的命名完全相同，并且接受相同的参数。它们是以这种方式专门设计的。

因此，`return Reflect...` 提供了一个安全的方式，可以轻松地转发操作，并确保我们不会忘记与此相关的任何内容。

Proxy 的局限性

代理提供了一种独特的方法，可以在最底层更改或调整现有对象的行为。但是，它并不完美。有局限性。

内建对象：内部插槽（Internal slot）

许多内建对象，例如 `Map`，`Set`，`Date`，`Promise` 等，都使用了所谓的“内部插槽”。

它们类似于属性，但仅限于内部使用，仅用于规范目的。例如，`Map` 将项目（item）存储在 `[[MapData]]` 中。内建方法可以直接访问它们，而不通过 `[[Get]]/[[Set]]` 内部方法。所以 `Proxy` 无法拦截它们。

为什么要在意这些呢？毕竟它们是内部的！

好吧，问题在这儿。在类似这样的内建对象被代理后，代理对象没有这些内部插槽，因此内建方法将会失败。

例如：

```
let map = new Map();

let proxy = new Proxy(map, {});

proxy.set('test', 1); // Error
```

在内部，一个 `Map` 将所有数据存储在其 `[[MapData]]` 内部插槽中。代理对象没有这样的插槽。`内建方法 Map.prototype.set` 方法试图访问内部属性 `this.[[MapData]]`，但由于 `this=proxy`，在 `proxy` 中无法找到它，只能失败。

幸运的是，这儿有一种解决方法：

```
let map = new Map();

let proxy = new Proxy(map, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
    return typeof value === 'function' ? value.bind(target) : value;
  }
});

proxy.set('test', 1);
alert(proxy.get('test')) // 1 (工作了!)
```

现在它正常工作了，因为 `get` 陷阱将函数属性（例如 `map.set`）绑定到了目标对象（`map`）本身。

与前面的示例不同，`proxy.set(...)` 内部 `this` 的值并不是 `proxy`，而是原始的 `map`。因此，当 `set` 陷阱的内部实现尝试访问 `this.[[MapData]]` 内部插槽时，它会成功。

① `Array` 没有内部插槽

一个值得注意的例外：内建 `Array` 没有使用内部插槽。那是出于历史原因，因为它出现于很久以前。

所以，代理数组时没有这种问题。

私有字段

类的私有字段也会发生类似的情况。

例如，`getName()` 方法访问私有的 `#name` 属性，并在代理后中断（`break`）：

```
class User {
  #name = "Guest";

  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {});

alert(user.getName()); // Error
```

原因是私有字段是通过内部插槽实现的。JavaScript 在访问它们时不使用 `[[Get]]/[[Set]]`。

在调用 `getName()` 时，`this` 的值是代理后的 `user`，它没有带有私有字段的插槽。

再次，带有 `bind` 方法的解决方案使它恢复正常：

```
class User {
  #name = "Guest";

  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
    return typeof value == 'function' ? value.bind(target) : value;
  }
});

alert(user.getName()); // Guest
```

如前所述，该解决方案也有缺点：它将原始对象暴露给该方法，可能使其进一步传递并破坏其他代理功能。

Proxy != target

代理和原始对象是不同的对象。这很自然，对吧？

所以，如果我们使用原始对象作为键，然后对其进行代理，之后却无法找到代理了：

```
let allUsers = new Set();

class User {
```

```
constructor(name) {
  this.name = name;
  allUsers.add(this);
}

let user = new User("John");

alert(allUsers.has(user)); // true

user = new Proxy(user, {});

alert(allUsers.has(user)); // false
```

如我们所见，进行代理后，我们在 `allUsers` 中找不到 `user`，因为代理是一个不同的对象。

⚠️ Proxy 无法拦截严格相等性检查 ===

`Proxy` 可以拦截许多操作符，例如 `new`（使用 `construct`），`in`（使用 `has`），`delete`（使用 `deleteProperty`）等。

但是没有办法拦截对于对象的严格相等性检查。一个对象只严格等于其自身，没有其他值。

因此，比较对象是否相等的所有操作和内建类都会区分对象和代理。这里没有透明的替代品。

可撤销 Proxy

一个 **可撤销** 的代理是可以被禁用的代理。

假设我们有一个资源，并且想随时关闭对该资源的访问。

我们可以做的是将它包装成一个撤销的代理，没有任何陷阱。这样的代理会将操作转发给对象，并且我们可以随时将其禁用。

语法为：

```
let {proxy, revoke} = Proxy.revocable(target, handler)
```

该调用返回一个带有 `proxy` 和 `revoke` 函数的对象以将其禁用。

这是一个例子：

```
let object = {
  data: "Valuable data"
};

let {proxy, revoke} = Proxy.revocable(object, {});

// 将 proxy 传递到其他某处，而不是对象...
alert(proxy.data); // Valuable data

// 稍后，在我们的代码中
revoke();
```

```
// proxy 不再工作 (revoked)
alert(proxy.data); // Error
```

调用 `revoke()` 会从代理中删除对目标对象的所有内部引用，因此它们之间再无连接。之后可以对目标对象进行垃圾回收。

我们还可以将 `revoke` 存储在 `WeakMap` 中，以更便于通过代理对象轻松找到它：

```
let revokes = new WeakMap();

let object = {
  data: "Valuable data"
};

let {proxy, revoke} = Proxy.revocable(object, {});

revokes.set(proxy, revoke);

// ...稍后，在我们的代码中...
revoke = revokes.get(proxy);
revoke();

alert(proxy.data); // Error (revoked)
```

这种方法的好处是，我们不必再随身携带 `revoke`。我们可以在有需要时通过 `proxy` 从 `map` 上获取它。

此处我们使用 `WeakMap` 而不是 `Map`，因为它不会阻止垃圾回收。如果一个代理对象变得“不可访问”（例如，没有变量再引用它），则 `WeakMap` 允许将其与它的 `revoke` 一起从内存中清除，因为我们不再需要它了。

参考资料

- 规范: [Proxy ↗](#)。
- MDN: [Proxy ↗](#)。

总结

`Proxy` 是对象的包装器，将代理上的操作转发到对象，并可以选择捕获其中一些操作。

它可以包装任何类型的对象，包括类和函数。

语法为：

```
let proxy = new Proxy(target, {
  /* trap */
});
```

.....然后，我们应该在所有地方使用 `proxy` 而不是 `target`。代理没有自己的属性或方法。如果提供了陷阱（trap），它将捕获操作，否则会将其转发给 `target` 对象。

我们可以捕获：

- 读取 (`get`)，写入 (`set`)，删除 (`deleteProperty`) 属性（甚至是不存在的属性）。
- 函数调用 (`apply` 陷阱)。
- `new` 操作 (`construct` 陷阱)。
- 许多其他操作（完整列表请见本文开头部分和 [docs ↗](#)）。

这使我们能够创建“虚拟”属性和方法，实现默认值，可观察对象，函数装饰器等。

我们还可以将对象多次包装在不同的代理中，并用多个各个方面功能对其进行装饰。

`Reflect` API 旨在补充 `Proxy`。对于任意 `Proxy` 陷阱，都有一个带有相同参数的 `Reflect` 调用。我们应该使用它们将调用转发给目标对象。

`Proxy` 有一些局限性：

- 内建对象具有“内部插槽”，对这些对象的访问无法被代理。请参阅上文中的解决方法。
- 私有类字段也是如此，因为它们也是在内部使用插槽实现的。因此，代理方法的调用必须具有目标对象作为 `this` 才能访问它们。
- 对象的严格相等性检查 `==` 无法被拦截。
- 性能：基准测试（benchmark）取决于引擎，但通常使用最简单的代理访问属性所需的时间也要长几倍。实际上，这仅对某些“瓶颈”对象来说才重要。

Eval: 执行代码字符串

内建函数 `eval` 函数允许执行一个代码字符串。

语法如下：

```
let result = eval(code);
```

例如：

```
let code = 'alert("Hello")';
eval(code); // Hello
```

代码字符串可能会比较长，包含换行符、函数声明和变量等。

`eval` 的结果是最后一条语句的结果。

例如：

```
let value = eval('1+1');
alert(value); // 2
```

```
let value = eval('let i = 0; ++i');
alert(value); // 1
```

`eval` 内的代码在当前词法环境（lexical environment）中执行，因此它能访问外部变量：

```
let a = 1;

function f() {
    let a = 2;

    eval('alert(a)'); // 2
}

f();
```

它也可以更改外部变量：

```
let x = 5;
eval("x = 10");
alert(x); // 10, 值被更改了
```

严格模式下，`eval` 有属于自己的词法环境。因此我们不能从外部访问在 `eval` 中声明的函数和变量：

```
// 提示：本教程所有可运行的示例都默认启用了严格模式 'use strict'

eval("let x = 5; function f() {}");

alert(typeof x); // undefined (没有这个变量)
// 函数 f 也不可从外部进行访问
```

如果不启用严格模式，`eval` 没有属于自己的词法环境，因此我们可以从外部访问变量 `x` 和函数 `f`。

使用“eval”

现代编程中，已经很少使用 `eval` 了。人们经常说“`eval` 是魔鬼”。

原因很简单：很久很久以前，JavaScript 是一种非常弱的语言，很多东西只能通过 `eval` 来完成。不过那已经是十年前的事了。

如今几乎找不到使用 `eval` 的理由了。如果有人在使用它，那这是一个很好的使用现代语言结构或 [JavaScript Module](#) 来替换它们的机会。

请注意，`eval` 访问外部变量的能力会产生副作用。

代码压缩工具（在把 JS 投入生产环境前对其进行压缩的工具）将局部变量重命名为更短的变量（例如 `a` 和 `b` 等），以使代码体积更小。这通常是安全的，但在使用了 `eval` 的情况下就不一样了，因为局部变量可能会被 `eval` 中的代码访问到。因此压缩工具不会对所有可能被从 `eval` 中访问的变量进行重命名。这样会导致代码压缩率降低。

在 `eval` 中使用外部局部变量也被认为是一个坏的编程习惯，因为这会使代码维护变得更加困难。

有两种方法可以完全避免此类问题。

如果 `eval` 中的代码没有使用外部变量，请以 `window.eval(...)` 的形式调用 `eval`：

通过这种方式，该代码便会在全局作用域内执行：

```
let x = 1;
{
  let x = 5;
  window.eval('alert(x)'); // 1 (全局变量)
}
```

如果 `eval` 中的代码需要访问局部变量，我们可以使用 `new Function` 替代 `eval`，并将它们作为参数传递：

```
let f = new Function('a', 'alert(a)');
f(5); // 5
```

我们在 "new Function" 语法 一章中对 `new Function` 构造器进行了详细说明。`new Function` 从字符串创建一个函数，并且也是在全局作用域中的。所以它无法访问局部变量。但是，正如上面的示例一样，将它们作为参数进行显式传递要清晰得多。

总结

调用 `eval(code)` 会运行代码字符串，并返回最后一条语句的结果。

- 在现代 JavaScript 编程中，很少使用它，通常也不需要使用它。
- 可以访问外部局部变量。这被认为是一个不好的编程习惯。
- 要在全局作用域中 `eval` 代码，可以使用 `window.eval(code)` 进行替代。
- 此外，如果你的代码需要从外部作用域获取数据，请使用 `new Function`，并将数据作为参数传递给函数。

柯里化 (Currying)

柯里化 (Currying) ↪ 是一种关于函数的高阶技术。它不仅被用于 JavaScript，还被用于其他编程语言。

柯里化是一种函数的转换，它是指将一个函数从可调用的 `f(a, b, c)` 转换为可调用的 `f(a)(b)(c)`。

柯里化不会调用函数。它只是对函数进行转换。

让我们先来看一个例子，以更好地理解我们正在讲的内容，然后再进行一个实际应用。

我们将创建一个辅助函数 `curry(f)`，该函数将对两个参数的函数 `f` 执行柯里化。换句话说，对于两个参数的函数 `f(a, b)` 执行 `curry(f)` 会将其转换为以 `f(a)(b)` 形式运行的函数：

```
function curry(f) { // curry(f) 执行柯里化转换
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}
```

```
}

// 用法
function sum(a, b) {
  return a + b;
}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3
```

正如你所看到的，实现非常简单：只有两个包装器（wrapper）。

- `curry(func)` 的结果就是一个包装器 `function(a)`。
- 当它被像 `curriedSum(1)` 这样调用时，它的参数会被保存在词法环境中，然后返回一个新的包装器 `function(b)`。
- 然后这个包装器被以 `2` 为参数调用，并且，它将该调用传递给原始的 `sum` 函数。

柯里化更高级的实现，例如 `lodash` 库的 `_.curry` ↗，会返回一个包装器，该包装器允许函数被正常调用或者以偏函数（partial）的方式调用：

```
function sum(a, b) {
  return a + b;
}

let curriedSum = _.curry(sum); // 使用来自 lodash 库的 _.curry

alert( curriedSum(1, 2) ); // 3, 仍可正常调用
alert( curriedSum(1)(2) ); // 3, 以偏函数的方式调用
```

柯里化？目的是什么？

要了解它的好处，我们需要一个实际中的例子。

例如，我们有一个用于格式化和输出信息的日志（logging）函数 `log(date, importance, message)`。在实际项目中，此类函数具有很多有用的功能，例如通过网络发送日志（log），在这儿我们仅使用 `alert`：

```
function log(date, importance, message) {
  alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}] ${message}`);
}
```

让我们将它柯里化！

```
log = _.curry(log);
```

柯里化之后，`log` 仍正常运行：

```
log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

.....但是也可以以柯里化形式运行:

```
log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

现在，我们可以轻松地为当前日志创建便捷函数:

```
// logNow 会是带有固定第一个参数的日志的偏函数
let logNow = log(new Date());

// 使用它
logNow("INFO", "message"); // [HH:mm] INFO message
```

现在，`logNow` 是具有固定第一个参数的 `log`，换句话说，就是更简短的“偏应用函数（partially applied function）”或“偏函数（partial）”。

我们可以更进一步，为当前的调试日志（debug log）提供便捷函数:

```
let debugNow = logNow("DEBUG");

debugNow("message"); // [HH:mm] DEBUG message
```

所以:

1. 柯里化之后，我们没有丢失任何东西：`log` 依然可以被正常调用。
2. 我们可以轻松地生成偏函数，例如用于生成今天的日志的偏函数。

高级柯里化实现

如果你想了解更多细节，下面是用于多参数函数的“高级”柯里化实现，我们也可以把它用于上面的示例。

它非常短:

```
function curry(func) {

    return function curried(...args) {
        if (args.length >= func.length) {
            return func.apply(this, args);
        } else {
            return function(...args2) {
                return curried.apply(this, args.concat(args2));
            }
        }
    };
}
```

用例:

```

function sum(a, b, c) {
  return a + b + c;
}

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, 仍然可以被正常调用
alert( curriedSum(1)(2,3) ); // 6, 对第一个参数的柯里化
alert( curriedSum(1)(2)(3) ); // 6, 全柯里化

```

新的 `curry` 可能看上去有点复杂，但是它很容易理解。

`curry(func)` 调用的结果是如下所示的包装器 `curried`：

```

// func 是要转换的函数
function curried(...args) {
  if (args.length >= func.length) { // (1)
    return func.apply(this, args);
  } else {
    return function pass(...args2) { // (2)
      return curried.apply(this, args.concat(args2));
    }
  }
};

```

当我们运行它时，这里有两个 `if` 执行分支：

1. 现在调用：如果传入的 `args` 长度与原始函数所定义的 (`func.length`) 相同或者更长，那么只需要将调用传递给它即可。
2. 获取一个偏函数：否则，`func` 还没有被调用。取而代之的是，返回另一个包装器 `pass`，它将重新应用 `curried`，将之前传入的参数与新的参数一起传入。然后，在一个新的调用中，再次，我们将获得一个新的偏函数（如果参数不足的话），或者最终的结果。

例如，让我们看看 `sum(a, b, c)` 这个例子。它有三个参数，所以 `sum.length = 3`。

对于调用 `curried(1)(2)(3)`：

1. 第一个调用 `curried(1)` 将 `1` 保存在词法环境中，然后返回一个包装器 `pass`。
2. 包装器 `pass` 被调用，参数为 `(2)`：它会获取之前的参数 `(1)`，将它与得到的 `(2)` 连在一起，并一起调用 `curried(1, 2)`。由于参数数量仍小于 3，`curry` 函数依然会返回 `pass`。
3. 包装器 `pass` 再次被调用，参数为 `(3)`，在接下来的调用中，`pass(3)` 会获取之前的参数 `(1, 2)` 并将 `3` 与之合并，执行调用 `curried(1, 2, 3)` — 最终有 3 个参数，它们被传入最原始的函数中。

如果这还不够清楚，那你可以把函数调用顺序在你的脑海中或者在纸上过一遍。

i 只允许确定参数长度的函数

柯里化要求函数具有固定数量的参数。

使用 `rest` 参数的函数，例如 `f(...args)`，不能以这种方式进行柯里化。

① 比柯里化多一点

根据定义，柯里化应该将 `sum(a, b, c)` 转换为 `sum(a)(b)(c)`。

但是，如前所述，JavaScript 中大多数的柯里化实现都是高级版的：它们使得函数可以被多参数变体调用。

总结

柯里化 是一种转换，将 `f(a, b, c)` 转换为可以被以 `f(a)(b)(c)` 的形式进行调用。JavaScript 实现通常都保持该函数可以被正常调用，并且如果参数数量不足，则返回偏函数。

柯里化让我们能够更容易地获取偏函数。就像我们在日志记录示例中看到的那样，普通函数 `log(date, importance, message)` 在被柯里化之后，当我们调用它的时候传入一个参数（如 `log(date)`）或两个参数（`log(date, importance)`）时，它会返回偏函数。

BigInt

⚠ A recent addition

This is a recent addition to the language. You can find the current state of support at <https://caniuse.com/#feat=bigint>.

BigInt 是一种特殊的数字类型，它提供了对任意长度整数的支持。

创建 `bigint` 的方式有两种：在一个整数字面量后面加 `n` 或者调用 `BigInt` 函数，该函数从字符串、数字等中生成 `bigint`。

```
const bigint = 1234567890123456789012345678901234567890n;  
  
const sameBigInt = BigInt("1234567890123456789012345678901234567890");  
  
const bigintFromNumber = BigInt(10); // 与 10n 相同
```

数学运算符

BigInt 大多数情况下可以像常规数字类型一样使用，例如：

```
alert(1n + 2n); // 3  
  
alert(5n / 2n); // 2
```

请注意：除法 `5/2` 的结果向零进行舍入，舍入后得到的结果没有了小数部分。对 `bigint` 的所有操作，返回的结果也是 `bigint`。

我们不可以把 `bigint` 和常规数字类型混合使用：

```
alert(1n + 2); // Error: Cannot mix BigInt and other types
```

如果有需要，我们应该显式地转换它们：使用 `BigInt()` 或者 `Number()`，像这样：

```
let bigint = 1n;
let number = 2;

// 将 number 转换为 bigint
alert(bigint + BigInt(number)); // 3

// 将 bigint 转换为 number
alert(Number(bigint) + number); // 3
```

转换操作始终是静默的，绝不会报错，但是如果 `bigint` 太大而数字类型无法容纳，则会截断多余的位，因此我们应该谨慎进行此类转换。

➊ `BigInt` 不支持一元加法

一元加法运算符 `+value`，是大家熟知的将 `value` 转换成数字类型的方法。

为了避免混淆，在 `bigint` 中不支持一元加法：

```
let bigint = 1n;

alert( +bigint ); // error
```

所以我们应该用 `Number()` 来将一个 `bigint` 转换成一个数字类型。

比较运算符

比较运算符，例如 `<` 和 `>`，使用它们来对 `bigint` 和 `number` 类型的数字进行比较没有问题：

```
alert( 2n > 1n ); // true

alert( 2n > 1 ); // true
```

但是请注意，由于 `number` 和 `bigint` 属于不同类型，它们可能在进行 `==` 比较时相等，但在进行 `===`（严格相等）比较时不相等：

```
alert( 1 == 1n ); // true

alert( 1 === 1n ); // false
```

布尔运算

当在 `if` 或其他布尔运算中时，`bigint` 的行为类似于 `number`。

例如，在 `if` 中，`bigint 0n` 为 `false`，其他值为 `true`：

```
if (0n) {  
    // 永远不会执行  
}
```

布尔运算符，例如 `||`、`&&` 和其他运算符，处理 `bigint` 的方式也类似于 `number`:

```
alert( 1n || 2 ); // 1 (1n 被认为是 true)  
alert( 0n || 2 ); // 2 (0n 被认为是 false)
```

Polyfill

Polyfilling `bigint` 比较棘手。原因是许多 JavaScript 运算符，比如 `+` 和 `-` 等，在对待 `bigint` 的行为上与常规 `number` 相比有所不同。

例如，`bigint` 的除法总是返回 `bigint`（如果需要，会进行舍入）。

想要模拟这种行为，`polyfill` 需要分析代码，并用其函数替换所有此类运算符。但是这样做很麻烦，并且会耗费很多性能。

所以，目前并没有一个众所周知的好用的 `polyfill`。

不过，[JSBI](#) 库的开发者提出了另一种解决方案。

该库使用自己的方法实现了大的数字。我们可以使用它们替代原生的 `bigint`:

运算	原生 <code>BigInt</code>	<code>JSBI</code>
从 <code>Number</code> 创建	<code>a = BigInt(789)</code>	<code>a = JSBI.BigInt(789)</code>
加法	<code>c = a + b</code>	<code>c = JSBI.add(a, b)</code>
减法	<code>c = a - b</code>	<code>c = JSBI.subtract(a, b)</code>
...

.....然后，对于那些支持 `bigint` 的浏览器，可以使用 `polyfill`（[Babel 插件](#)）将 `JSBI` 调用转换为原生的 `bigint`。

换句话说，这个方法建议我们在写代码时使用 `JSBI` 替代原生的 `bigint`。但是 `JSBI` 在内部像使用 `bigint` 一样使用 `number`，并最大程度按照规范进行模拟，所以代码已经是准备好转换成 `bigint` 的了（`bigint-ready`）。

对于不支持 `bigint` 的引擎，我们可以“按原样”使用此类 `JSBI` 代码，对于那些支持 `bigint` 的引擎 — `polyfill` 会将调用转换为原生的 `bigint`。

参考

- [MDN 文档对 `BigInt` 的介绍](#)。
- [ECMA262 规范](#)。