目　录

致谢

当前文档 《阮一峰 JavaScript 教程》 由 进击的皇虫 使用书栈(BookStack.CN) 进行构建，生成于 2018-02-26。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/javascript-tutorial

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

本教程全面介绍 JavaScript 核心语法，从最简单的开始讲起，循序渐进、由浅入深，力求清晰易懂。所有章节都带有大量的代码实例，便于理解和模仿，可以用到实际项目中，即学即用。

本教程适合初学者当作 JavaScript 语言的入门教程，也适合当作日常使用的参考手册。

前言

前言

我写这本教程，主要原因是自己需要。

编程时，往往需要查阅资料，确定准确用法。理想的 JavaScript 参考书，应该简明易懂，一目了然，告诉我有哪些注意点，提供代码范例。如果涉及重要概念，还应该适当讲解。可是大多数时候，现实都不是如此。找到的资料冗长难懂，抓不住重点，有时还很陈旧，跟不上语言标准和浏览器的快速发展，且大多数是英文资料。

学习过程中，我做了很多 JavaScript 笔记。多年累积，数量相当庞大。遇到问题，我首先查自己的笔记，如果笔记里没有，再到网上查，最后回过头把笔记补全。终于有一天，我意识到可以把笔记做成书，这就是这本书的由来。

正因为脱胎于笔记，这本教程跟其他 JavaScript 书籍有所不同。作为教程，本书对所有重要概念都进行了讲解，努力把复杂的问题讲得简单，希望一两分钟内就能抓住重点。本书还可以作为参考手册，罗列了主要用法和各种 API 接口，并给出可以立即运行的代码。所有章节按照语言的 API 编排，方便以后的查阅。

如果你发现某处出现陌生的新概念，请不要担心，可以继续阅读下去。以后查阅这些章节的时候，你会发现很方便找到某个知识点相关的所有内容。

入门篇

入门篇

导论

导论

什么是 JavaScript 语言？

JavaScript 是一种轻量级的脚本语言。所谓“脚本语言”（script language），指的是它不具备开发操作系统的能力，而是只用来编写控制其他大型应用程序（比如浏览器）的“脚本”。

JavaScript 也是一种嵌入式（embedded）语言。它本身提供的核心语法不算很多，只能用来做一些数学和逻辑运算。JavaScript 本身不提供任何与 I/O（输入/输出）相关的 API，都要靠宿主环境（host）提供，所以 JavaScript 只合适嵌入更大型的应用程序环境，去调用宿主环境提供的底层 API。

目前，已经嵌入 JavaScript 的宿主环境有多种，最常见的环境就是浏览器，另外还有服务器环境，也就是 Node 项目。

从语法角度看，JavaScript 语言是一种“对象模型”语言。各种宿主环境通过这个模型，描述自己的功能和操作接口，从而通过 JavaScript 控制这些功能。但是，JavaScript 并不是纯粹的“面向对象语言”，还支持其他编程范式（比如函数式编程）。这导致几乎任何一个问题，JavaScript 都有多种解决方法。阅读本书的过程中，你会诧异于 JavaScript 语法的灵活性。

JavaScript 的核心语法部分相当精简，只包括两个部分：基本的语法构造（比如操作符、控制结构、语句）和标准库（就是一系列具有各种功能的对象比如 `Array`、`Date`、`Math` 等）。除此之外，各种宿主环境提供额外的 API（即只能在该环境使用的接口），以便 JavaScript 调用。以浏览器为例，它提供的额外 API 可以分成三大类。

- 浏览器控制类：操作浏览器
- DOM 类：操作网页的各种元素
- Web 类：实现互联网的各种功能

如果宿主环境是服务器，则会提供各种操作系统的 API，比如文件操作 API、网络通信 API等等。这些你都可以在 Node 环境中找到。

本书主要介绍 JavaScript 核心语法和浏览器网页开发的基本知识，不涉及 Node。全书可以分成以下四大部分。

- 基本语法
- 标准库
- 浏览器 API
- DOM

JavaScript 语言有多个版本。本书的内容主要基于 ECMAScript 5.1 版本，这是学习 JavaScript 语法的基础。ES6 和更新的语法

请参考我写的《ECMAScript 6入门》。

为什么学习 JavaScript？

JavaScript 语言有一些显著特点，使得它非常值得学习。它既适合作为学习编程的入门语言，也适合当作日常开发的工作语言。它是目前最有希望、前途最光明的计算机语言之一。

操控浏览器的能力

JavaScript 的发明目的，就是作为浏览器的内置脚本语言，为网页开发者提供操控浏览器的能力。它是目前唯一一种通用的浏览器脚本语言，所有浏览器都支持。它可以让网页呈现各种特殊效果，为用户提供良好的互动体验。

目前，全世界几乎所有网页都使用 JavaScript。如果不用，网站的易用性和使用效率将大打折扣，无法成为操作便利、对用户友好的网站。

对于一个互联网开发者来说，如果你想提供漂亮的网页、令用户满意的上网体验、各种基于浏览器的便捷功能、前后端之间紧密高效的联系，JavaScript 是必不可少的工具。

广泛的使用领域

近年来，JavaScript 的使用范围，慢慢超越了浏览器，正在向通用的系统语言发展。

（1）浏览器的平台化

随着 HTML5 的出现，浏览器本身的功能越来越强，不再仅仅能浏览网页，而是越来越像一个平台，JavaScript 因此得以调用许多系统功

能，比如操作本地文件、操作图片、调用摄像头和麦克风等等。这使得 JavaScript 可以完成许多以前无法想象的事情。

（2）Node

Node 项目使得 JavaScript 可以用于开发服务器端的大型项目，网站的前后端都用 JavaScript 开发已经成为了现实。有些嵌入式平台（Raspberry Pi）能够安装 Node，于是 JavaScript 就能为这些平台开发应用程序。

（3）数据库操作

JavaScript 甚至也可以用来操作数据库。NoSQL 数据库这个概念，本身就是在 JSON（JavaScript Object Notation）格式的基础上诞生的，大部分 NoSQL 数据库允许 JavaScript 直接操作。基于 SQL 语言的开源数据库 PostgreSQL 支持 JavaScript 作为操作语言，可以部分取代 SQL 查询语言。

（4）移动平台开发

JavaScript 也正在成为手机应用的开发语言。一般来说，安卓平台使用 Java 语言开发，iOS 平台使用 Objective-C 或 Swift 语言开发。许多人正在努力，让 JavaScript 成为各个平台的通用开发语言。

PhoneGap 项目就是将 JavaScript 和 HTML5 打包在一个容器之中，使得它能同时在 iOS 和安卓上运行。Facebook 公司的 React Native 项目则是将 JavaScript 写的组件，编译成原生组件，从而使它们具备优秀的性能。

Mozilla 基金会的手机操作系统 Firefox OS，更是直接将 JavaScript 作为操作系统的平台语言，但是很可惜这个项目没有成

功。

（5）内嵌脚本语言

越来越多的应用程序，将 JavaScript 作为内嵌的脚本语言，比如 Adobe 公司的著名 PDF 阅读器 Acrobat、Linux 桌面环境 GNOME 3。

（6）跨平台的桌面应用程序

Chromium OS、Windows 8 等操作系统直接支持 JavaScript 编写应用程序。Mozilla 的 Open Web Apps 项目、Google 的 Chrome App 项目、Github 的 Electron 项目、以及 TideSDK 项目，都可以用来编写运行于 Windows、Mac OS 和 Android 等多个桌面平台的程序，不依赖浏览器。

（7）小结

可以预期，JavaScript 最终将能让你只用一种语言，就开发出适应不同平台（包括桌面端、服务器端、手机端）的程序。早在2013年9月的统计之中，JavaScript 就是当年 Github 上使用量排名第一的语言。

著名程序员 Jeff Atwood 甚至提出了一条 “Atwood 定律”：

> *“所有可以用 JavaScript 编写的程序，最终都会出现 JavaScript 的版本。”(Any application that can be written in JavaScript will eventually be written in JavaScript.)*

易学性

相比学习其他语言，学习 JavaScript 有一些有利条件。

（1）学习环境无处不在

只要有浏览器，就能运行 JavaScript 程序；只要有文本编辑器，就能编写 JavaScript 程序。这意味着，几乎所有电脑都原生提供 JavaScript 学习环境，不用另行安装复杂的 IDE（集成开发环境）和编译器。

（2）简单性

相比其他脚本语言（比如 Python 或 Ruby），JavaScript 的语法相对简单一些，本身的语法特性并不是特别多。而且，那些语法中的复杂部分，也不是必需要学会。你完全可以只用简单命令，完成大部分的操作。

（3）与主流语言的相似性

JavaScript 的语法很类似 C/C++ 和 Java，如果学过这些语言（事实上大多数学校都教），JavaScript 的入门会非常容易。

必须说明的是，虽然核心语法不难，但是 JavaScript 的复杂性体现在另外两个方面。

首先，它涉及大量的外部 API。JavaScript 要发挥作用，必须与其他组件配合，这些外部组件五花八门，数量极其庞大，几乎涉及网络应用的各个方面，掌握它们绝非易事。

其次，JavaScript 语言有一些设计缺陷。某些地方相当不合理，另一些地方则会出现怪异的运行结果。学习 JavaScript，很大一部分时间是用来搞清楚哪些地方有陷阱。Douglas Crockford 写过一本有名的书，名字就叫《JavaScript: The Good Parts》，言下之意就是这门语言不好的地方很多，必须写一本书才能讲清楚。另外一些程序员则感到，为了更合理地编写 JavaScript 程序，就不能用 JavaScript 来写，而必须发明新的语言，比如 CoffeeScript、TypeScript、Dart 这些新语言的发明目的，多多少少都有这个因

素。

尽管如此，目前看来，JavaScript 的地位还是无法动摇。加之，语言标准的快速进化，使得 JavaScript 功能日益增强，而语法缺陷和怪异之处得到了弥补。所以，JavaScript 还是值得学习，况且它的入门真的不难。

强大的性能

JavaScript 的性能优势体现在以下方面。

（1）灵活的语法，表达力强。

JavaScript 既支持类似 C 语言清晰的过程式编程，也支持灵活的函数式编程，可以用来写并发处理（concurrent）。这些语法特性已经被证明非常强大，可以用于许多场合，尤其适用异步编程。

JavaScript 的所有值都是对象，这为程序员提供了灵活性和便利性。因为你可以很方便地、按照需要随时创造数据结构，不用进行麻烦的预定义。

JavaScript 的标准还在快速进化中，并不断合理化，添加更适用的语法特性。

（2）支持编译运行。

JavaScript 语言本身，虽然是一种解释型语言，但是在现代浏览器中，JavaScript 都是编译后运行。程序会被高度优化，运行效率接近二进制程序。而且，JavaScript 引擎正在快速发展，性能将越来越好。

此外，还有一种 WebAssembly 格式，它是 JavaScript 引擎的中间码格式，全部都是二进制代码。由于跳过了编译步骤，可以达到接近

原生二进制代码的运行速度。各种语言（主要是 C 和 C++）通过编译成 WebAssembly，就可以在浏览器里面运行。

（3）事件驱动和非阻塞式设计。

JavaScript 程序可以采用事件驱动（event-driven）和非阻塞式（non-blocking）设计，在服务器端适合高并发环境，普通的硬件就可以承受很大的访问量。

开放性

JavaScript 是一种开放的语言。它的标准 ECMA-262 是 ISO 国际标准，写得非常详尽明确；该标准的主要实现（比如 V8 和 SpiderMonkey 引擎）都是开放的，而且质量很高。这保证了这门语言不属于任何公司或个人，不存在版权和专利的问题。

语言标准由 TC39 委员会负责制定，该委员会的运作是透明的，所有讨论都是开放的，会议记录都会对外公布。

不同公司的 JavaScript 运行环境，兼容性很好，程序不做调整或只做很小的调整，就能在所有浏览器上运行。

社区支持和就业机会

全世界程序员都在使用 JavaScript，它有着极大的社区、广泛的文献和图书、丰富的代码资源。绝大部分你需要用到的功能，都有多个开源函数库可供选用。

作为项目负责人，你不难招聘到数量众多的 JavaScript 程序员；作为开发者，你也不难找到一份 JavaScript 的工作。

实验环境

本教程包含大量的示例代码，只要电脑安装了浏览器，就可以用来实验了。读者可以一边读一边运行示例，加深理解。

推荐安装 Chrome 浏览器，它的“开发者工具”（Developer Tools）里面的“控制台”（console），就是运行 JavaScript 代码的理想环境。

进入 Chrome 浏览器的“控制台”，有两种方法。

- 直接进入：按下 `Option + Command + J`（Mac）或者 `Ctrl + Shift + J`（Windows / Linux）
- 开发者工具进入：开发者工具的快捷键是 F12，或者 `Option + Command + I`（Mac）以及 `Ctrl + Shift + I`（Windows / Linux），然后选择 Console 面板

进入控制台以后，就可以在提示符后输入代码，然后按 `Enter` 键，代码就会执行。如果按 `Shift + Enter` 键，就是代码换行，不会触发执行。建议阅读本教程时，将代码复制到控制台进行实验。

作为尝试，你可以将下面的程序复制到“控制台”，按下回车后，就可以看到运行结果。

```
function greetMe(yourName) {
  console.log('Hello ' + yourName);
}

greetMe('World')
// Hello World
```

历史

JavaScript 语言的历史

诞生

JavaScript 因为互联网而生，紧跟着浏览器的出现而问世。回顾它的历史，就要从浏览器的历史讲起。

1990年底，欧洲核能研究组织（CERN）科学家 Tim Berners-Lee，在全世界最大的电脑网络——互联网的基础上，发明了万维网（World Wide Web），从此可以在网上浏览网页文件。最早的网页只能在操作系统的终端里浏览，也就是说只能使用命令行操作，网页都是在字符窗口中显示，这当然非常不方便。

1992年底，美国国家超级电脑应用中心（NCSA）开始开发一个独立的浏览器，叫做 Mosaic。这是人类历史上第一个浏览器，从此网页可以在图形界面的窗口浏览。

1994年10月，NCSA 的一个主要程序员 Marc Andreessen 联合风险投资家 Jim Clark，成立了 Mosaic 通信公司（Mosaic Communications），不久后改名为 Netscape。这家公司的方向，

就是在 Mosaic 的基础上，开发面向普通用户的新一代的浏览器 Netscape Navigator。

1994年12月，Navigator 发布了1.0版，市场份额一举超过90%。

Netscape 公司很快发现，Navigator 浏览器需要一种可以嵌入网页的脚本语言，用来控制浏览器行为。当时，网速很慢而且上网费很贵，有些操作不宜在服务器端完成。比如，如果用户忘记填写“用户名”，就点了“发送”按钮，到服务器再发现这一点就有点太晚了，最好能在用户发出数据之前，就告诉用户“请填写用户名”。这就需要在网页中嵌入小程序，让浏览器检查每一栏是否都填写了。

管理层对这种浏览器脚本语言的设想是：功能不需要太强，语法较为简单，容易学习和部署。那一年，正逢 Sun 公司的 Java 语言问世，市场推广活动非常成功。Netscape 公司决定与 Sun 公司合作，浏览器支持嵌入 Java 小程序（后来称为 Java applet）。但是，浏览器脚本语言是否就选用 Java，则存在争论。后来，还是决定不使用 Java，因为网页小程序不需要 Java 这么“重”的语法。但是，同时也决定脚本语言的语法要接近 Java，并且可以支持 Java 程序。这些设想直接排除了使用现存语言，比如 Perl、Python 和 TCL。

1995年，Netscape 公司雇佣了程序员 Brendan Eich 开发这种网页脚本语言。Brendan Eich 有很强的函数式编程背景，希望以 Scheme 语言（函数式语言鼻祖 LISP 语言的一种方言）为蓝本，实现这种新语言。

1995年5月，Brendan Eich 只用了10天，就设计完成了这种语言的第一版。它是一个大杂烩，语法有多个来源。

- 基本语法：借鉴 C 语言和 Java 语言。
- 数据结构：借鉴 Java 语言，包括将值分成原始值和对象两大

类。

- 函数的用法：借鉴 Scheme 语言和 Awk 语言，将函数当作第一等公民，并引入闭包。
- 原型继承模型：借鉴 Self 语言（Smalltalk 的一种变种）。
- 正则表达式：借鉴 Perl 语言。
- 字符串和数组处理：借鉴 Python 语言。

为了保持简单，这种脚本语言缺少一些关键的功能，比如块级作用域、模块、子类型（subtyping）等等，但是可以利用现有功能找出解决办法。这种功能的不足，直接导致了后来 JavaScript 的一个显著特点：对于其他语言，你需要学习语言的各种功能，而对于 JavaScript，你常常需要学习各种解决问题的模式。而且由于来源多样，从一开始就注定，JavaScript 的编程风格是函数式编程和面向对象编程的一种混合体。

Netscape 公司的这种浏览器脚本语言，最初名字叫做 Mocha，1995年9月改为 LiveScript。12月，Netscape 公司与 Sun 公司（Java 语言的发明者和所有者）达成协议，后者允许将这种语言叫做 JavaScript。这样一来，Netscape 公司可以借助 Java 语言的声势，而 Sun 公司则将自己的影响力扩展到了浏览器。

之所以起这个名字，并不是因为 JavaScript 本身与 Java 语言有多么深的关系（事实上，两者关系并不深，详见下节），而是因为 Netscape 公司已经决定，使用 Java 语言开发网络应用程序，JavaScript 可以像胶水一样，将各个部分连接起来。当然，后来的历史是 Java 语言的浏览器插件失败了，JavaScript 反而发扬光大。

1995年12月4日，Netscape 公司与 Sun 公司联合发布了 JavaScript 语言，对外宣传 JavaScript 是 Java 的补充，属

于轻量级的 Java，专门用来操作网页。

1996年3月，Navigator 2.0 浏览器正式内置了 JavaScript 脚本语言。

JavaScript 与 Java 的关系

这里专门说一下 JavaScript 和 Java 的关系。它们是两种不一样的语言，但是彼此存在联系。

JavaScript 的基本语法和对象体系，是模仿 Java 而设计的。但是，JavaScript 没有采用 Java 的静态类型。正是因为 JavaScript 与 Java 有很大的相似性，所以这门语言才从一开始的 LiveScript 改名为 JavaScript。基本上，JavaScript 这个名字的原意是“很像Java的脚本语言”。

JavaScript 语言的函数是一种独立的数据类型，以及采用基于原型对象（prototype）的继承链。这是它与 Java 语法最大的两点区别。JavaScript 语法要比 Java 自由得多。

另外，Java 语言需要编译，而 JavaScript 语言则是运行时由解释器直接执行。

总之，JavaScript 的原始设计目标是一种小型的、简单的动态语言，与 Java 有足够的相似性，使得使用者（尤其是 Java 程序员）可以快速上手。

JavaScript 与 ECMAScript 的关系

1996年8月，微软模仿 JavaScript 开发了一种相近的语言，取名为 JScript（JavaScript是Netscape的注册商标，微软不能用），首先内置于IE 3.0。Netscape 公司面临丧失浏览器脚本语言的主导权

的局面。

1996年11月，Netscape 公司决定将 JavaScript 提交给国际标准化组织 ECMA（European Computer Manufacturers Association），希望 JavaScript 能够成为国际标准，以此抵抗微软。ECMA 的39号技术委员会（Technical Committee 39）负责制定和审核这个标准，成员由业内的大公司派出的工程师组成，目前共25个人。该委员会定期开会，所有的邮件讨论和会议记录，都是公开的。

1997年7月，ECMA 组织发布262号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为 ECMAScript。这个版本就是 ECMAScript 1.0 版。之所以不叫 JavaScript，一方面是由于商标的关系，Java 是 Sun 公司的商标，根据一份授权协议，只有 Netscape 公司可以合法地使用 JavaScript 这个名字，且 JavaScript 已经被 Netscape 公司注册为商标，另一方面也是想体现这门语言的制定者是 ECMA，不是 Netscape，这样有利于保证这门语言的开放性和中立性。因此，ECMAScript 和 JavaScript 的关系是，前者是后者的规格，后者是前者的一种实现。在日常场合，这两个词是可以互换的。

ECMAScript 只用来标准化 JavaScript 这种语言的基本语法结构，与部署环境相关的标准都由其他标准规定，比如 DOM 的标准就是由 W3C组织（World Wide Web Consortium）制定的。

ECMA-262 标准后来也被另一个国际标准化组织 ISO（International Organization for Standardization）批准，标准号是 ISO-16262。

JavaScript的版本

1997年7月，ECMAScript 1.0发布。

1998年6月，ECMAScript 2.0版发布。

1999年12月，ECMAScript 3.0版发布，成为 JavaScript 的通行标准，得到了广泛支持。

2007年10月，ECMAScript 4.0版草案发布，对3.0版做了大幅升级，预计次年8月发布正式版本。草案发布后，由于4.0版的目标过于激进，各方对于是否通过这个标准，发生了严重分歧。以 Yahoo、Microsoft、Google 为首的大公司，反对 JavaScript 的大幅升级，主张小幅改动；以 JavaScript 创造者 Brendan Eich 为首的 Mozilla 公司，则坚持当前的草案。

2008年7月，由于对于下一个版本应该包括哪些功能，各方分歧太大，争论过于激进，ECMA 开会决定，中止 ECMAScript 4.0 的开发（即废除了这个版本），将其中涉及现有功能改善的一小部分，发布为 ECMAScript 3.1，而将其他激进的设想扩大范围，放入以后的版本，由于会议的气氛，该版本的项目代号起名为 Harmony（和谐）。会后不久，ECMAScript 3.1 就改名为 ECMAScript 5。

2009年12月，ECMAScript 5.0版 正式发布。Harmony 项目则一分为二，一些较为可行的设想定名为 JavaScript.next 继续开发，后来演变成 ECMAScript 6；一些不是很成熟的设想，则被视为 JavaScript.next.next，在更远的将来再考虑推出。TC39 的总体考虑是，ECMAScript 5 与 ECMAScript 3 基本保持兼容，较大的语法修正和新功能加入，将由 JavaScript.next 完成。当时，JavaScript.next 指的是ECMAScript 6。第六版发布以后，将指 ECMAScript 7。TC39 预计，ECMAScript 5 会在2013年的年中成为 JavaScript 开发的主流标准，并在此后五年中一直保持这个位置。

2011年6月，ECMAscript 5.1版发布，并且成为 ISO 国际标准（ISO/IEC 16262:2011）。到了2012年底，所有主要浏览器都支持ECMAScript 5.1版的全部功能。

2013年3月，ECMAScript 6 草案冻结，不再添加新功能。新的功能设想将被放到 ECMAScript 7。

2013年12月，ECMAScript 6 草案发布。然后是12个月的讨论期，听取各方反馈。

2015年6月，ECMAScript 6 正式发布，并且更名为“ECMAScript 2015”。这是因为 TC39 委员会计划，以后每年发布一个 ECMAScript 的版本，下一个版本在2016年发布，称为”ECMAScript 2016”，2017年发布“ECMAScript 2017”，以此类推。

周边大事记

JavaScript 伴随着互联网的发展一起发展。互联网周边技术的快速发展，刺激和推动了 JavaScript 语言的发展。下面，回顾一下 JavaScript 的周边应用发展。

1996年，样式表标准 CSS 第一版发布。

1997年，DHTML（Dynamic HTML，动态 HTML）发布，允许动态改变网页内容。这标志着 DOM 模式（Document Object Model，文档对象模型）正式应用。

1998年，Netscape 公司开源了浏览器，这导致了 Mozilla 项目的诞生。几个月后，美国在线（AOL）宣布并购 Netscape。

1999年，IE 5部署了 XMLHttpRequest 接口，允许 JavaScript

发出 HTTP 请求，为后来大行其道的 Ajax 应用创造了条件。

2000年，KDE 项目重写了浏览器引擎 KHTML，为后来的 WebKit 和 Blink 引擎打下基础。这一年的10月23日，KDE 2.0发布，第一次将 KHTML 浏览器包括其中。

2001年，微软公司时隔5年之后，发布了 IE 浏览器的下一个版本 Internet Explorer 6。这是当时最先进的浏览器，它后来统治了浏览器市场多年。

2001年，Douglas Crockford 提出了 JSON 格式，用于取代 XML 格式，进行服务器和网页之间的数据交换。JavaScript 可以原生支持这种格式，不需要额外部署代码。

2002年，Mozilla 项目发布了它的浏览器的第一版，后来起名为 Firefox。

2003年，苹果公司发布了 Safari 浏览器的第一版。

2004年，Google 公司发布了 Gmail，促成了互联网应用程序（Web Application）这个概念的诞生。由于 Gmail 是在4月1日发布的，很多人起初以为这只是一个玩笑。

2004年，Dojo 框架诞生，为不同浏览器提供了同一接口，并为主要功能提供了便利的调用方法。这标志着 JavaScript 编程框架的时代开始来临。

2004年，WHATWG 组织成立，致力于加速 HTML 语言的标准化进程。

2005年，苹果公司在 KHTML 引擎基础上，建立了 WebKit 引擎。

2005年，Ajax 方法（Asynchronous JavaScript and XML）正式诞生，Jesse James Garrett 发明了这个词汇。它开始流行的标

志是，2月份发布的 Google Maps 项目大量采用该方法。它几乎成了新一代网站的标准做法，促成了 Web 2.0时代的来临。

2005年，Apache 基金会发布了 CouchDB 数据库。这是一个基于 JSON 格式的数据库，可以用 JavaScript 函数定义视图和索引。它在本质上有别于传统的关系型数据库，标识着 NoSQL 类型的数据库诞生。

2006年，jQuery 函数库诞生，作者为John Resig。jQuery 为操作网页 DOM 结构提供了非常强大易用的接口，成为了使用最广泛的函数库，并且让 JavaScript 语言的应用难度大大降低，推动了这种语言的流行。

2006年，微软公司发布 IE 7，标志重新开始启动浏览器的开发。

2006年，Google推出 Google Web Toolkit 项目（缩写为 GWT），提供 Java 编译成 JavaScript 的功能，开创了将其他语言转为 JavaScript 的先河。

2007年，Webkit 引擎在 iPhone 手机中得到部署。它最初基于 KDE 项目，2003年苹果公司首先采用，2005年开源。这标志着 JavaScript 语言开始能在手机中使用了，意味着有可能写出在桌面电脑和手机中都能使用的程序。

2007年，Douglas Crockford 发表了名为《JavaScript: The good parts》的演讲，次年由 O’Reilly 出版社出版。这标志着软件行业开始严肃对待 JavaScript 语言，对它的语法开始重新认识，

2008年，V8 编译器诞生。这是 Google 公司为 Chrome 浏览器而开发的，它的特点是让 JavaScript 的运行变得非常快。它提高了 JavaScript 的性能，推动了语法的改进和标准化，改变外界对 JavaScript 的不佳印象。同时，V8 是开源的，任何人想要一种快

速的嵌入式脚本语言，都可以采用 V8，这拓展了 JavaScript 的应用领域。

2009年，Node.js 项目诞生，创始人为 Ryan Dahl，它标志着 JavaScript 可以用于服务器端编程，从此网站的前端和后端可以使用同一种语言开发。并且，Node.js 可以承受很大的并发流量，使得开发某些互联网大规模的实时应用变得容易。

2009年，Jeremy Ashkenas 发布了 CoffeeScript 的最初版本。CoffeeScript 可以被转换为 JavaScript 运行，但是语法要比 JavaScript简洁。这开启了其他语言转为 JavaScript 的风潮。

2009年，PhoneGap 项目诞生，它将 HTML5 和 JavaScript 引入移动设备的应用程序开发，主要针对 iOS 和 Android 平台，使得 JavaScript 可以用于跨平台的应用程序开发。

2009，Google 发布 Chrome OS，号称是以浏览器为基础发展成的操作系统，允许直接使用 JavaScript 编写应用程序。类似的项目还有 Mozilla 的 Firefox OS。

2010年，三个重要的项目诞生，分别是 NPM、BackboneJS 和 RequireJS，标志着 JavaScript 进入模块化开发的时代。

2011年，微软公司发布 Windows 8操作系统，将 JavaScript 作为应用程序的开发语言之一，直接提供系统支持。

2011年，Google 发布了 Dart 语言，目的是为了结束 JavaScript 语言在浏览器中的垄断，提供更合理、更强大的语法和功能。Chromium浏览器有内置的 Dart 虚拟机，可以运行 Dart 程序，但 Dart 程序也可以被编译成 JavaScript 程序运行。

2011年，微软工程师Scott Hanselman提出，JavaScript 将是互联网的汇编语言。因为它无所不在，而且正在变得越来越快。其他语言的程序可以被转成 JavaScript 语言，然后在浏览器中运行。

2012年，单页面应用程序框架（single-page app framework）开始崛起，AngularJS 项目和 Ember 项目都发布了1.0版本。

2012年，微软发布 TypeScript 语言。该语言被设计成 JavaScript 的超集，这意味着所有 JavaScript 程序，都可以不经修改地在 TypeScript 中运行。同时，TypeScript 添加了很多新的语法特性，主要目的是为了开发大型程序，然后还可以被编译成 JavaScript 运行。

2012年，Mozilla 基金会提出 asm.js 规格。asm.js 是 JavaScript 的一个子集，所有符合 asm.js 的程序都可以在浏览器中运行，它的特殊之处在于语法有严格限定，可以被快速编译成性能良好的机器码。这样做的目的，是为了给其他语言提供一个编译规范，使其可以被编译成高效的 JavaScript 代码。同时，Mozilla 基金会还发起了 Emscripten 项目，目标就是提供一个跨语言的编译器，能够将 LLVM 的位代码（bitcode）转为 JavaScript 代码，在浏览器中运行。因为大部分 LLVM 位代码都是从 C / C++ 语言生成的，这意味着 C / C++ 将可以在浏览器中运行。此外，Mozilla 旗下还有 LLJS （将 JavaScript 转为 C 代码）项目和 River Trail （一个用于多核心处理器的 ECMAScript 扩展）项目。目前，可以被编译成 JavaScript 的语言列表，共有将近40种语言。

2013年，Mozilla 基金会发布手机操作系统 Firefox OS，该操作系统的整个用户界面都使用 JavaScript。

2013年，ECMA 正式推出 JSON 的国际标准，这意味着 JSON 格式已经变得与 XML 格式一样重要和正式了。

2013年5月，Facebook 发布 UI 框架库 React，引入了新的 JSX 语法，使得 UI 层可以用组件开发，同时引入了网页应用是状态机的概念。

2014年，微软推出 JavaScript 的 Windows 库 WinJS，标志微软公司全面支持 JavaScript 与 Windows 操作系统的融合。

2014年11月，由于对 Joyent 公司垄断 Node 项目、以及该项目进展缓慢的不满，一部分核心开发者离开了 Node.js，创造了 io.js 项目，这是一个更开放、更新更频繁的 Node.js 版本，很短时间内就发布到了2.0版。三个月后，Joyent 公司宣布放弃对 Node 项目的控制，将其转交给新成立的开放性质的 Node 基金会。随后，io.js 项目宣布回归 Node，两个版本将合并。

2015年3月，Facebook 公司发布了 React Native 项目，将 React 框架移植到了手机端，可以用来开发手机 App。它会将 JavaScript 代码转为 iOS 平台的 Objective-C 代码，或者 Android 平台的 Java 代码，从而为 JavaScript 语言开发高性能的原生 App 打开了一条道路。

2015年4月，Angular 框架宣布，2.0 版将基于微软公司的 TypeScript语言开发，这等于为 JavaScript 语言引入了强类型。

2015年5月，Node 模块管理器 NPM 超越 CPAN，标志着 JavaScript 成为世界上软件模块最多的语言。

2015年5月，Google 公司的 Polymer 框架发布1.0版。该项目的目标是生产环境可以使用 WebComponent 组件，如果能够达到目标，Web 开发将进入一个全新的以组件为开发基础的阶段。

2015年6月，ECMA 标准化组织正式批准了 ECMAScript 6 语言标准，定名为《ECMAScript 2015 标准》。JavaScript语言正式进

入了下一个阶段，成为一种企业级的、开发大规模应用的语言。这个标准从提出到批准，历时10年，而 JavaScript 语言从诞生至今也已经20年了。

2015年6月，Mozilla 在 asm.js 的基础上发布 WebAssembly 项目。这是一种 JavaScript 引擎的中间码格式，全部都是二进制，类似于 Java 的字节码，有利于移动设备加载 JavaScript 脚本，执行速度提高了 20+ 倍。这意味着将来的软件，会发布 JavaScript 二进制包。

2016年6月，《ECMAScript 2016 标准》发布。与前一年发布的版本相比，它只增加了两个较小的特性。

2017年6月，《ECMAScript 2017 标准》发布，正式引入了 async 函数，使得异步操作的写法出现了根本的变化。

2017年11月，所有主流浏览器全部支持 WebAssembly，这意味着任何语言都可以编译成 JavaScript，在浏览器运行。

参考链接

- Axel Rauschmayer, The Past, Present, and Future of JavaScript
- John Dalziel, The race for speed part 4: The future for JavaScript
- Axel Rauschmayer, Basic JavaScript for the impatient programmer
- resin.io, Happy 18th Birthday JavaScript! A look at an unlikely past and bright future

基本语法

JavaScript 的基本语法

语句

JavaScript 程序的执行单位为行（line），也就是一行一行地执

行。一般情况下，每一行就是一个语句。

语句（statement）是为了完成某种任务而进行的操作，比如下面就是一行赋值语句。

```
var a = 1 + 3;
```

这条语句先用 `var` 命令，声明了变量 `a`，然后将 `1 + 3` 的运算结果赋值给变量 `a`。

`1 + 3` 叫做表达式（expression），指一个为了得到返回值的计算式。语句和表达式的区别在于，前者主要为了进行某种操作，一般情况下不需要返回值；后者则是为了得到返回值，一定会返回一个值。凡是 JavaScript 语言中预期为值的地方，都可以使用表达式。比如，赋值语句的等号右边，预期是一个值，因此可以放置各种表达式。

语句以分号结尾，一个分号就表示一个语句结束。多个语句可以写在一行内。

```
var a = 1 + 3 ; var b = 'abc';
```

分号前面可以没有任何内容，JavaScript引擎将其视为空语句。

```
;;;
```

上面的代码就表示3个空语句。

表达式不需要分号结尾。一旦在表达式后面添加分号，则 JavaScript 引擎就将表达式视为语句，这样会产生一些没有任何意义的语句。

```
1 + 3;
'abc';
```

上面两行语句只是单纯地产生一个值，并没有任何实际的意义。

变量

概念

变量是对“值”的具名引用。变量就是为“值”起名，然后引用这个名字，就等同于引用这个值。变量的名字就是变量名。

```
1. var a = 1;
```

上面的代码先声明变量 `a` ，然后在变量 `a` 与数值1之间建立引用关系，称为将数值1“赋值”给变量 `a` 。以后，引用变量名 `a` 就会得到数值1。最前面的 `var` ，是变量声明命令。它表示通知解释引擎，要创建一个变量 `a` 。

注意，JavaScript 的变量名区分大小写， `A` 和 `a` 是两个不同的变量。

变量的声明和赋值，是分开的两个步骤，上面的代码将它们合在了一起，实际的步骤是下面这样。

```
1. var a;
2. a = 1;
```

如果只是声明变量而没有赋值，则该变量的值是 `undefined` 。 `undefined` 是一个 JavaScript 关键字，表示“无定义”。

```
1. var a;
2. a // undefined
```

如果变量赋值的时候，忘了写 `var` 命令，这条语句也是有效的。

```
var a = 1;
// 基本等同
a = 1;
```

但是，不写 `var` 的做法，不利于表达意图，而且容易不知不觉地创建全局变量，所以建议总是使用 `var` 命令声明变量。

如果一个变量没有声明就直接使用，JavaScript 会报错，告诉你变量未定义。

```
x
// ReferenceError: x is not defined
```

上面代码直接使用变量 `x` ，系统就报错，告诉你变量 `x` 没有声明。

可以在同一条 `var` 命令中声明多个变量。

```
var a, b;
```

JavaScript 是一种动态类型语言，也就是说，变量的类型没有限制，变量可以随时更改类型。

```
var a = 1;
a = 'hello';
```

上面代码中，变量 `a` 起先被赋值为一个数值，后来又被重新赋值为一个字符串。第二次赋值的时候，因为变量 `a` 已经存在，所以不需要使用 `var` 命令。

如果使用 `var` 重新声明一个已经存在的变量，是无效的。

```
var x = 1;
```

```
var x;
x // 1
```

上面代码中，变量 `x` 声明了两次，第二次声明是无效的。

但是，如果第二次声明的时候还进行了赋值，则会覆盖掉前面的值。

```
var x = 1;
var x = 2;

// 等同于

var x = 1;
var x;
x = 2;
```

变量提升

JavaScript 引擎的工作方式是，先解析代码，获取所有被声明的变量，然后再一行一行地运行。这造成的结果，就是所有的变量的声明语句，都会被提升到代码的头部，这就叫做变量提升（hoisting）。

```
console.log(a);
var a = 1;
```

上面代码首先使用 `console.log` 方法，在控制台（console）显示变量 `a` 的值。这时变量 `a` 还没有声明和赋值，所以这是一种错误的做法，但是实际上不会报错。因为存在变量提升，真正运行的是下面的代码。

```
var a;
console.log(a);
a = 1;
```

最后的结果是显示 `undefined`，表示变量 `a` 已声明，但还未赋值。

标识符

标识符（identifier）指的是用来识别各种值的合法名称。最常见的标识符就是变量名，以及后面要提到的函数名。JavaScript 语言的标识符对大小写敏感，所以 `a` 和 `A` 是两个不同的标识符。

标识符有一套命名规则，不符合规则的就是非法标识符。JavaScript 引擎遇到非法标识符，就会报错。

简单说，标识符命名规则如下。

- 第一个字符，可以是任意 Unicode 字母（包括英文字母和其他语言的字母），以及美元符号（`$`）和下划线（`_`）。
- 第二个字符及后面的字符，除了 Unicode 字母、美元符号和下划线，还可以用数字 `0-9`。

下面这些都是合法的标识符。

```
arg0
_tmp
$elem
π
```

下面这些则是不合法的标识符。

```
1a  // 第一个字符不能是数字
23  // 同上
***  // 标识符不能包含星号
a+b  // 标识符不能包含加号
-d  // 标识符不能包含减号或连词线
```

中文是合法的标识符，可以用作变量名。

```
var 临时变量 = 1;
```

> *JavaScript有一些保留字，不能用作标识符：arguments、break、case、catch、class、const、continue、debugger、default、delete、do、else、enum、eval、export、extends、false、finally、for、function、if、implements、import、in、instanceof、interface、let、new、null、package、private、protected、public、return、static、super、switch、this、throw、true、try、typeof、var、void、while、with、yield。*

注释

源码中被 JavaScript 引擎忽略的部分就叫做注释，它的作用是对代码进行解释。Javascript 提供两种注释的写法：一种是单行注释，用 `//` 起头；另一种是多行注释，放在 `/*` 和 `*/` 之间。

```
// 这是单行注释

/*
 这是
 多行
 注释
*/
```

此外，由于历史上 JavaScript 可以兼容 HTML 代码的注释，所以 `<!--` 和 `-->` 也被视为合法的单行注释。

```
x = 1; <!-- x = 2;
--> x = 3;
```

上面代码中，只有 `x = 1` 会执行，其他的部分都被注释掉了。

需要注意的是， `-->` 只有在行首，才会被当成单行注释，否则会当作正常的运算。

```
function countdown(n) {
```

```
  while (n --> 0) console.log(n);
}
countdown(3)
// 2
// 1
// 0
```

上面代码中，`n --> 0` 实际上会当作 `n-- > 0` ，因此输出2、1、0。

区块

JavaScript 使用大括号，将多个相关的语句组合在一起，称为“区块”（block）。

对于 `var` 命令来说，JavaScript 的区块不构成单独的作用域（scope）。

```
{
  var a = 1;
}

a // 1
```

上面代码在区块内部，使用 `var` 命令声明并赋值了变量 `a` ，然后在区块外部，变量 `a` 依然有效，区块对于 `var` 命令不构成单独的作用域，与不使用区块的情况没有任何区别。在 JavaScript 语言中，单独使用区块并不常见，区块往往用来构成其他更复杂的语法结构，比如 `for` 、 `if` 、 `while` 、 `function` 等。

条件语句

JavaScript 提供 `if` 结构和 `switch` 结构，完成条件判断，即只有满足预设的条件，才会执行相应的语句。

if 结构

`if` 结构先判断一个表达式的布尔值，然后根据布尔值的真伪，执行不同的语句。所谓布尔值，指的是 JavaScript 的两个特殊值，`true` 表示真，`false` 表示 `伪` 。

```
if (布尔值)
  语句;

// 或者
if (布尔值) 语句;
```

上面是 `if` 结构的基本形式。需要注意的是，“布尔值”往往由一个条件表达式产生的，必须放在圆括号中，表示对表达式求值。如果表达式的求值结果为 `true` ，就执行紧跟在后面的语句；如果结果为 `false` ，则跳过紧跟在后面的语句。

```
if (m === 3)
  m = m + 1;
```

上面代码表示，只有在 `m` 等于3时，才会将其值加上1。

这种写法要求条件表达式后面只能有一个语句。如果想执行多个语句，必须在 `if` 的条件判断之后，加上大括号，表示代码块（多个语句合并成一个语句）。

```
if (m === 3) {
  m += 1;
}
```

建议总是在 `if` 语句中使用大括号，因为这样方便插入语句。

注意， `if` 后面的表达式之中，不要混淆赋值表达式（ `=` ）、严格相

等运算符（ `===` ）和相等运算符（ `==` ）。尤其是赋值表达式不具有比较作用。

```
var x = 1;
var y = 2;
if (x = y) {
  console.log(x);
}
// "2"
```

上面代码的原意是，当 `x` 等于 `y` 的时候，才执行相关语句。但是，不小心将严格相等运算符写成赋值表达式，结果变成了将 `y` 赋值给变量 `x` ，再判断变量 `x` 的值（等于2）的布尔值（结果为 `true` ）。

这种错误可以正常生成一个布尔值，因而不会报错。为了避免这种情况，有些开发者习惯将常量写在运算符的左边，这样的话，一旦不小心将相等运算符写成赋值运算符，就会报错，因为常量不能被赋值。

```
if (x = 2) { // 不报错
if (2 = x) { // 报错
```

至于为什么优先采用“严格相等运算符”（ `===` ），而不是“相等运算符”（ `==` ），请参考《运算符》章节。

if…else 结构

`if` 代码块后面，还可以跟一个 `else` 代码块，表示不满足条件时，所要执行的代码。

```
if (m === 3) {
  // 满足条件时，执行的语句
} else {
  // 不满足条件时，执行的语句
}
```

上面代码判断变量 `m` 是否等于3，如果等于就执行 `if` 代码块，否则执行 `else` 代码块。

对同一个变量进行多次判断时，多个 `if...else` 语句可以连写在一起。

```
if (m === 0) {
  // ...
} else if (m === 1) {
  // ...
} else if (m === 2) {
  // ...
} else {
  // ...
}
```

`else` 代码块总是与离自己最近的那个 `if` 语句配对。

```
var m = 1;
var n = 2;

if (m !== 1)
if (n === 2) console.log('hello');
else console.log('world');
```

上面代码不会有任何输出， `else` 代码块不会得到执行，因为它跟着的是最近的那个 `if` 语句，相当于下面这样。

```
if (m !== 1) {
  if (n === 2) {
    console.log('hello');
  } else {
    console.log('world');
  }
}
```

如果想让 `else` 代码块跟随最上面的那个 `if` 语句，就要改变大括号的位置。

```
if (m !== 1) {
  if (n === 2) {
    console.log('hello');
  }
} else {
  console.log('world');
}
// world
```

switch 结构

多个 `if...else` 连在一起使用的时候，可以转为使用更方便的 `switch` 结构。

```
switch (fruit) {
  case "banana":
    // ...
    break;
  case "apple":
    // ...
    break;
  default:
    // ...
}
```

上面代码根据变量 `fruit` 的值，选择执行相应的 `case` 。如果所有 `case` 都不符合，则执行最后的 `default` 部分。需要注意的是，每个 `case` 代码块内部的 `break` 语句不能少，否则会接下去执行下一个 `case` 代码块，而不是跳出 `switch` 结构。

```
var x = 1;

```

```
switch (x) {
  case 1:
    console.log('x 等于1');
  case 2:
    console.log('x 等于2');
  default:
    console.log('x 等于其他值');
}
// x等于1
// x等于2
// x等于其他值
```

上面代码中，`case` 代码块之中没有 `break` 语句，导致不会跳出 `switch` 结构，而会一直执行下去。正确的写法是像下面这样。

```
switch (x) {
  case 1:
    console.log('x 等于1');
    break;
  case 2:
    console.log('x 等于2');
    break;
  default:
    console.log('x 等于其他值');
}
```

`switch` 语句部分和 `case` 语句部分，都可以使用表达式。

```
switch (1 + 3) {
  case 2 + 2:
    f();
    break;
  default:
    neverHappens();
}
```

上面代码的 `default` 部分，是永远不会执行到的。

需要注意的是，`switch`语句后面的表达式，与`case`语句后面的表示式比较运行结果时，采用的是严格相等运算符（`===`），而不是相等运算符（`==`），这意味着比较时不会发生类型转换。

```
var x = 1;

switch (x) {
  case true:
    console.log('x 发生类型转换');
  default:
    console.log('x 没有发生类型转换');
}
// x 没有发生类型转换
```

上面代码中，由于变量`x`没有发生类型转换，所以不会执行`case true`的情况。这表明，`switch`语句内部采用的是“严格相等运算符”，详细解释请参考《运算符》一节。

三元运算符 ?:

JavaScript还有一个三元运算符（即该运算符需要三个运算子）`?:`，也可以用于逻辑判断。

```
(条件) ? 表达式1 : 表达式2
```

上面代码中，如果“条件”为`true`，则返回“表达式1”的值，否则返回“表达式2”的值。

```
var even = (n % 2 === 0) ? true : false;
```

上面代码中，如果`n`可以被2整除，则`even`等于`true`，否则等于`false`。它等同于下面的形式。

```
var even;
if (n % 2 === 0) {
  even = true;
} else {
  even = false;
}
```

这个三元运算符可以被视为 `if...else...` 的简写形式，因此可以用于多种场合。

```
var myVar;
console.log(
  myVar ?
  'myVar has a value' :
  'myVar does not have a value'
)
// myVar does not have a value
```

上面代码利用三元运算符，输出相应的提示。

```
var msg = '数字' + n + '是' + (n % 2 === 0 ? '偶数' : '奇数');
```

上面代码利用三元运算符，在字符串之中插入不同的值。

循环语句

循环语句用于重复执行某个操作，它有多种形式。

while 循环

`While` 语句包括一个循环条件和一段代码块，只要条件为真，就不断循环执行代码块。

```
while (条件)
```

```
  语句;

// 或者
while (条件) 语句;
```

`while` 语句的循环条件是一个表达式，必须放在圆括号中。代码块部分，如果只有一条语句，可以省略大括号，否则就必须加上大括号。

```
while (条件) {
  语句;
}
```

下面是 `while` 语句的一个例子。

```
var i = 0;

while (i < 100) {
  console.log('i 当前为：' + i);
  i = i + 1;
}
```

上面的代码将循环100次，直到 `i` 等于100为止。

下面的例子是一个无限循环，因为循环条件总是为真。

```
while (true) {
  console.log('Hello, world');
}
```

for 循环

`for` 语句是循环命令的另一种形式，可以指定循环的起点、终点和终止条件。它的格式如下。

```
for (初始化表达式; 条件; 递增表达式)
```

```
  语句

// 或者

for (初始化表达式; 条件; 递增表达式) {
  语句
}
```

`for` 语句后面的括号里面，有三个表达式。

- 初始化表达式（initialize）：确定循环变量的初始值，只在循环开始时执行一次。
- 条件表达式（test）：每轮循环开始时，都要执行这个条件表达式，只有值为真，才继续进行循环。
- 递增表达式（increment）：每轮循环的最后一个操作，通常用来递增循环变量。

下面是一个例子。

```
var x = 3;
for (var i = 0; i < x; i++) {
  console.log(i);
}
// 0
// 1
// 2
```

上面代码中，初始化表达式是 `var i = 0` ，即初始化一个变量 `i` ；测试表达式是 `i < x` ，即只要 `i` 小于 `x` ，就会执行循环；递增表达式是 `i++` ，即每次循环结束后， `i` 增大1。

所有 `for` 循环，都可以改写成 `while` 循环。上面的例子改为 `while` 循环，代码如下。

```
var x = 3;
var i = 0;

while (i < x) {
  console.log(i);
  i++;
}
```

`for` 语句的三个部分（initialize、test、increment），可以省略任何一个，也可以全部省略。

```
for ( ; ; ){
  console.log('Hello World');
}
```

上面代码省略了 `for` 语句表达式的三个部分，结果就导致了一个无限循环。

do…while 循环

`do...while` 循环与 `while` 循环类似，唯一的区别就是先运行一次循环体，然后判断循环条件。

```
do
  语句
while (条件);

// 或者
do {
  语句
} while (条件);
```

不管条件是否为真， `do...while` 循环至少运行一次，这是这种结构最大的特点。另外， `while` 语句后面的分号注意不要省略。

下面是一个例子。

```
var x = 3;
var i = 0;

do {
  console.log(i);
  i++;
} while(i < x);
```

break 语句和 continue 语句

`break` 语句和 `continue` 语句都具有跳转作用，可以让代码不按既有的顺序执行。

`break` 语句用于跳出代码块或循环。

```
var i = 0;

while(i < 100) {
  console.log('i 当前为：' + i);
  i++;
  if (i === 10) break;
}
```

上面代码只会执行10次循环，一旦 `i` 等于10，就会跳出循环。

`for` 循环也可以使用 `break` 语句跳出循环。

```
for (var i = 0; i < 5; i++) {
  console.log(i);
  if (i === 3)
    break;
}
// 0
// 1
```

```
// 2
// 3
```

上面代码执行到 `i` 等于3，就会跳出循环。

`continue` 语句用于立即终止本轮循环，返回循环结构的头部，开始下一轮循环。

```
var i = 0;

while (i < 100){
  i++;
  if (i % 2 === 0) continue;
  console.log('i 当前为：' + i);
}
```

上面代码只有在 `i` 为奇数时，才会输出 `i` 的值。如果 `i` 为偶数，则直接进入下一轮循环。

如果存在多重循环，不带参数的 `break` 语句和 `continue` 语句都只针对最内层循环。

标签（label）

JavaScript 语言允许，语句的前面有标签（label），相当于定位符，用于跳转到程序的任意位置，标签的格式如下。

```
label:
  语句
```

标签可以是任意的标识符，但不能是保留字，语句部分可以是任意语句。

标签通常与 `break` 语句和 `continue` 语句配合使用，跳出特定的循环。

```
top:
  for (var i = 0; i < 3; i++){
    for (var j = 0; j < 3; j++){
      if (i === 1 && j === 1) break top;
      console.log('i=' + i + ', j=' + j);
    }
  }
// i=0, j=0
// i=0, j=1
// i=0, j=2
// i=1, j=0
```

上面代码为一个双重循环区块，`break`命令后面加上了`top`标签（注意，`top`不用加引号），满足条件时，直接跳出双层循环。如果`break`语句后面不使用标签，则只能跳出内层循环，进入下一次的外层循环。

标签也可以用于跳出代码块。

```
foo: {
  console.log(1);
  break foo;
  console.log('本行不会输出');
}
console.log(2);
// 1
// 2
```

上面代码执行到`break foo`，就会跳出区块。

`continue`语句也可以与标签配合使用。

```
top:
  for (var i = 0; i < 3; i++){
    for (var j = 0; j < 3; j++){
      if (i === 1 && j === 1) continue top;
```

```
      console.log('i=' + i + ', j=' + j);
    }
  }
// i=0, j=0
// i=0, j=1
// i=0, j=2
// i=1, j=0
// i=2, j=0
// i=2, j=1
// i=2, j=2
```

上面代码中，`continue`命令后面有一个标签名，满足条件时，会跳过当前循环，直接进入下一轮外层循环。如果`continue`语句后面不使用标签，则只能进入下一轮的内层循环。

参考链接

- Axel Rauschmayer, A quick overview of JavaScript

数据类型

数据类型

概述

数据类型概述

简介

JavaScript 语言的每一个值，都属于某一种数据类型。JavaScript 的数据类型，共有六种。（ES6 又新增了第七种 Symbol 类型的值，本教程不涉及。）

- 数值（number）：整数和小数（比如`1`和`3.14`）
- 字符串（string）：文本（比如`Hello World`）。
- 布尔值（boolean）：表示真伪的两个特殊值，即`true`（真）和`false`（假）
- `undefined`：表示“未定义”或不存在，即由于目前没有定义，所以此处暂时没有任何值
- `null`：表示空值，即此处的值为空。
- 对象（object）：各种值组成的集合。

通常，数值、字符串、布尔值这三种类型，合称为原始类型（primitive type）的值，即它们是最基本的数据类型，不能再细分了。对象则称为合成类型（complex type）的值，因为一个对象往往是多个原始类型的值的合成，可以看作是一个存放各种值的容器。至于`undefined`和`null`，一般将它们看成两个特殊值。

对象是最复杂的数据类型，又可以分成三个子类型。

- 狭义的对象（object）
- 数组（array）
- 函数（function）

狭义的对象和数组是两种不同的数据组合方式，除非特别声明，本教程的”对象“都特指狭义的对象。函数其实是处理数据的方法，JavaScript 把它当成一种数据类型，可以赋值给变量，这为编程带来了很大的灵活性，也为 JavaScript 的“函数式编程”奠定了基础。

typeof 运算符

JavaScript 有三种方法，可以确定一个值到底是什么类型。

- `typeof`运算符
- `instanceof`运算符
- `Object.prototype.toString`方法

`instanceof`运算符和`Object.prototype.toString`方法，将在后文介绍。这里介绍`typeof`运算符。

`typeof`运算符可以返回一个值的数据类型。

数值、字符串、布尔值分别返回`number`、`string`、`boolean`。

```
typeof 123 // "number"
typeof '123' // "string"
typeof false // "boolean"
```

函数返回`function`。

```
function f() {}
typeof f
// "function"
```

`undefined` 返回 `undefined` 。

```
typeof undefined
// "undefined"
```

利用这一点， `typeof` 可以用来检查一个没有声明的变量，而不报错。

```
v
// ReferenceError: v is not defined

typeof v
// "undefined"
```

上面代码中，变量 `v` 没有用 `var` 命令声明，直接使用就会报错。但是，放在 `typeof` 后面，就不报错了，而是返回 `undefined` 。

实际编程中，这个特点通常用在判断语句。

```
// 错误的写法
if (v) {
  // ...
}
// ReferenceError: v is not defined

// 正确的写法
if (typeof v === "undefined") {
  // ...
}
```

对象返回 `object` 。

```
typeof window // "object"
typeof {} // "object"
typeof [] // "object"
```

上面代码中，空数组（ `[]` ）的类型也是 `object` ，这表示在 JavaScript 内部，数组本质上只是一种特殊的对象。这里顺便提一下， `instanceof` 运算符可以区分数组和对象。 `instanceof` 运算符的详细解释，请见《面向对象编程》一章。

```
var o = {};
var a = [];

o instanceof Array // false
a instanceof Array // true
```

`null` 返回 `object` 。

```
`typeof null // "object"
```

`null` 的类型是 `object` ，这是由于历史原因造成的。1995年的 JavaScript 语言第一版，只设计了五种数据类型（对象、整数、浮点数、字符串和布尔值），没考虑 `null` ，只把它当作 `object` 的一种特殊值。后来 `null` 独立出来，作为一种单独的数据类型，为了兼容以前的代码， `typeof null` 返回 `object` 就没法改变了。

参考链接

- Axel Rauschmayer, Improving the JavaScript typeof operator

null，undefined 和布尔值

null, undefined 和布尔值

null 和 undefined

概述

`null` 与 `undefined` 都可以表示“没有”，含义非常相似。将一个变量赋值为 `undefined` 或 `null` ，老实说，语法效果几乎没区别。

```
var a = undefined;
// 或者
var a = null;
```

上面代码中，变量 `a` 分别被赋值为 `undefined` 和 `null` ，这两种写法的效果几乎等价。

在 `if` 语句中，它们都会被自动转为 `false` ，相等运算符（ `==` ）甚至直接报告两者相等。

```
if (!undefined) {
  console.log('undefined is false');
}
// undefined is false
```

```
5.
6.  if (!null) {
7.    console.log('null is false');
8.  }
9.  // null is false
10.
11. undefined == null
12. // true
```

从上面代码可见，两者的行为是何等相似！谷歌公司开发的 JavaScript 语言的替代品 Dart 语言，就明确规定只有 `null`，没有 `undefined`！

既然含义与用法都差不多，为什么要同时设置两个这样的值，这不是无端增加复杂度，令初学者困扰吗？这与历史原因有关。

1995年 JavaScript 诞生时，最初像 Java 一样，只设置了 `null` 表示"无"。根据 C 语言的传统，`null` 可以自动转为 `0`。

```
1.  Number(null) // 0
2.  5 + null // 5
```

上面代码中，`null` 转为数字时，自动变成0。

但是，JavaScript 的设计者 Brendan Eich，觉得这样做还不够。首先，第一版的 JavaScript 里面，`null` 就像在 Java 里一样，被当成一个对象，Brendan Eich 觉得表示"无"的值最好不是对象。其次，那时的 JavaScript 不包括错误处理机制，Brendan Eich 觉得，如果 `null` 自动转为0，很不容易发现错误。

因此，他又设计了一个 `undefined`。区别是这样的：`null` 是一个表示"空"的对象，转为数值时为 `0`；`undefined` 是一个表示"此处无定义"的原始值，转为数值时为 `NaN`。

```
Number(undefined) // NaN
5 + undefined // NaN
```

用法和含义

对于 `null` 和 `undefined`，大致可以像下面这样理解。

`null` 表示空值，即该处的值现在为空。调用函数时，某个参数未设置任何值，这时就可以传入 `null`，表示该参数为空。比如，某个函数接受引擎抛出的错误作为参数，如果运行过程中未出错，那么这个参数就会传入 `null`，表示未发生错误。

`undefined` 表示“未定义”，下面是返回 `undefined` 的典型场景。

```
// 变量声明了，但没有赋值
var i;
i // undefined

// 调用函数时，应该提供的参数没有提供，该参数等于 undefined
function f(x) {
  return x;
}
f() // undefined

// 对象没有赋值的属性
var  o = new Object();
o.p // undefined

// 函数没有返回值时，默认返回 undefined
function f() {}
f() // undefined
```

布尔值

布尔值代表“真”和“假”两个状态。“真”用关键字 `true` 表示，“假”用关键字 `false` 表示。布尔值只有这两个值。

下列运算符会返回布尔值：

- 两元逻辑运算符： `&&` (And)， `||` (Or)
- 前置逻辑运算符： `!` (Not)
- 相等运算符： `===`， `!==`， `==`， `!=`
- 比较运算符： `>`， `>=`， `<`， `<=`

如果 JavaScript 预期某个位置应该是布尔值，会将该位置上现有的值自动转为布尔值。转换规则是除了下面六个值被转为 `false`，其他值都视为 `true`。

- `undefined`
- `null`
- `false`
- `0`
- `NaN`
- `""` 或 `''`（空字符串）

布尔值往往用于程序流程的控制，请看一个例子。

```
if ('') {
  console.log('true');
}
// 没有任何输出
```

上面代码中， `if` 命令后面的判断条件，预期应该是一个布尔值，所以 JavaScript 自动将空字符串，转为布尔值 `false`，导致程序不会进入代码块，所以没有任何输出。

注意，空数组（ `[]` ）和空对象（ `{}` ）对应的布尔值，都

是 `true` 。

```
if ([]) {
  console.log('true');
}
// true

if ({}) {
  console.log('true');
}
// true
```

更多关于数据类型转换的介绍，参见《数据类型转换》一章。

参考链接

- Axel Rauschmayer, Categorizing values in JavaScript

数值

数值

概述

整数和浮点数

JavaScript 内部，所有数字都是以64位浮点数形式储存，即使整数也是如此。所以， `1` 与 `1.0` 是相同的，是同一个数。

```
1. 1 === 1.0 // true
```

这就是说，JavaScript 语言的底层根本没有整数，所有数字都是小数（64位浮点数）。容易造成混淆的是，某些运算只有整数才能完成，此时 JavaScript 会自动把64位浮点数，转成32位整数，然后再进行运算，参见《运算符》一章的”位运算“部分。

由于浮点数不是精确的值，所以涉及小数的比较和运算要特别小心。

```
1. 0.1 + 0.2 === 0.3
2. // false
3.
4. 0.3 / 0.1
5. // 2.9999999999999996
6.
7. (0.3 - 0.2) === (0.2 - 0.1)
8. // false
```

数值精度

根据国际标准 IEEE 754，JavaScript 浮点数的64个二进制位，从最左边开始，是这样组成的。

- 第1位：符号位，`0`表示正数，`1`表示负数
- 第2位到第12位（共11位）：指数部分
- 第13位到第64位（共52位）：小数部分（即有效数字）

符号位决定了一个数的正负，指数部分决定了数值的大小，小数部分决定了数值的精度。

指数部分一共有11个二进制位，因此大小范围就是0到2047。IEEE 754 规定，如果指数部分的值在0到2047之间（不含两个端点），那么有效数字的第一位默认总是1，不保存在64位浮点数之中。也就是

说，有效数字这时总是 `1.xx...xx` 的形式，其中 `xx..xx` 的部分保存在64位浮点数之中，最长可能为52位。因此，JavaScript 提供的有效数字最长为53个二进制位。

```
(-1)^符号位 * 1.xx...xx * 2^指数部分
```

上面公式是正常情况下（指数部分在0到2047之间），一个数在 JavaScript 内部实际的表示形式。

精度最多只能到53个二进制位，这意味着，绝对值小于等于2的53次方的整数，即-2^{53}到2^{53}，都可以精确表示。

```
Math.pow(2, 53)
// 9007199254740992

Math.pow(2, 53) + 1
// 9007199254740992

Math.pow(2, 53) + 2
// 9007199254740994

Math.pow(2, 53) + 3
// 9007199254740996

Math.pow(2, 53) + 4
// 9007199254740996
```

上面代码中，大于2的53次方以后，整数运算的结果开始出现错误。所以，大于2的53次方的数值，都无法保持精度。由于2的53次方是一个16位的十进制数值，所以简单的法则就是，JavaScript 对15位的十进制数都可以精确处理。

```
Math.pow(2, 53)
// 9007199254740992
```

```
3.
4.  // 多出的三个有效数字，将无法保存
5.  9007199254740992111
6.  // 9007199254740992000
```

上面示例表明，大于2的53次方以后，多出来的有效数字（最后三位的 `111` ）都会无法保存，变成0。

数值范围

根据标准，64位浮点数的指数部分的长度是11个二进制位，意味着指数部分的最大值是2047（2的11次方减1）。也就是说，64位浮点数的指数部分的值最大为2047，分出一半表示负数，则 JavaScript 能够表示的数值范围为2^{1024}到2^{-1023}（开区间），超出这个范围的数无法表示。

如果一个数大于等于2的1024次方，那么就会发生“正向溢出”，即 JavaScript 无法表示这么大的数，这时就会返回 `Infinity` 。

```
1.  Math.pow(2, 1024) // Infinity
```

如果一个数小于等于2的-1075次方（指数部分最小值-1023，再加上小数部分的52位），那么就会发生为“负向溢出”，即 JavaScript 无法表示这么小的数，这时会直接返回0。

```
1.  Math.pow(2, -1075) // 0
```

下面是一个实际的例子。

```
1.  var x = 0.5;
2.
3.  for(var i = 0; i < 25; i++) {
4.    x = x * x;
```

```
}

x // 0
```

上面代码中，对 `0.5` 连续做25次平方，由于最后结果太接近0，超出了可表示的范围，JavaScript 就直接将其转为0。

JavaScript 提供 `Number` 对象的 `MAX_VALUE` 和 `MIN_VALUE` 属性，返回可以表示的具体的最大值和最小值。

```
Number.MAX_VALUE // 1.7976931348623157e+308
Number.MIN_VALUE // 5e-324
```

数值的表示法

JavaScript 的数值有多种表示方法，可以用字面形式直接表示，比如 `35`（十进制）和 `0xFF`（十六进制）。

数值也可以采用科学计数法表示，下面是几个科学计数法的例子。

```
123e3 // 123000
123e-3 // 0.123
-3.1E+12
.1e-23
```

科学计数法允许字母 `e` 或 `E` 的后面，跟着一个整数，表示这个数值的指数部分。

以下两种情况，JavaScript 会自动将数值转为科学计数法表示，其他情况都采用字面形式直接表示。

（1）小数点前的数字多于21位。

```
1234567890123456789012
```

```
// 1.2345678901234568e+21

12345678901234567901
// 12345678901234680000
```

（2）小数点后的零多于5个。

```
// 小数点后紧跟5个以上的零，
// 就自动转为科学计数法
0.0000003 // 3e-7

// 否则，就保持原来的字面形式
0.000003 // 0.000003
```

数值的进制

使用字面量（literal）直接表示一个数值时，JavaScript 对整数提供四种进制的表示方法：十进制、十六进制、八进制、二进制。

- 十进制：没有前导0的数值。
- 八进制：有前缀`0o`或`0O`的数值，或者有前导0、且只用到0-7的八个阿拉伯数字的数值。
- 十六进制：有前缀`0x`或`0X`的数值。
- 二进制：有前缀`0b`或`0B`的数值。

默认情况下，JavaScript 内部会自动将八进制、十六进制、二进制转为十进制。下面是一些例子。

```
0xff // 255
0o377 // 255
0b11 // 3
```

如果八进制、十六进制、二进制的数值里面，出现不属于该进制的数

字，就会报错。

```
0xzz // 报错
0o88 // 报错
0b22 // 报错
```

上面代码中，十六进制出现了字母 `z` 、八进制出现数字 `8` 、二进制出现数字 `2` ，因此报错。

通常来说，有前导0的数值会被视为八进制，但是如果前导0后面有数字 `8` 和 `9` ，则该数值被视为十进制。

```
0888 // 888
0777 // 511
```

前导0表示八进制，处理时很容易造成混乱。ES5 的严格模式和ES6，已经废除了这种表示法，但是浏览器为了兼容以前的代码，目前还继续支持这种表示法。

特殊数值

JavaScript 提供了几个特殊的数值。

正零和负零

前面说过，JavaScript 的64位浮点数之中，有一个二进制位是符号位。这意味着，任何一个数都有一个对应的负值，就连 `0` 也不例外。

JavaScript 内部实际上存在2个 `0` ：一个是 `+0` ，一个是 `-0` ，区别就是64位浮点数表示法的符号位不同。它们是等价的。

```
-0 === +0 // true
0 === -0 // true
```

```
3. 0 === +0 // true
```

几乎所有场合，正零和负零都会被当作正常的 `0` 。

```
1. +0 // 0
2. -0 // 0
3. (-0).toString() // '0'
4. (+0).toString() // '0'
```

唯一有区别的场合是， `+0` 或 `-0` 当作分母，返回的值是不相等的。

```
1. (1 / +0) === (1 / -0) // false
```

上面的代码之所以出现这样结果，是因为除以正零得到 `+Infinity` ，除以负零得到 `-Infinity` ，这两者是不相等的（关于 `Infinity` 详见下文）。

NaN

（1）含义

`NaN` 是 JavaScript 的特殊值，表示“非数字”（Not a Number），主要出现在将字符串解析成数字出错的场合。

```
1. 5 - 'x' // NaN
```

上面代码运行时，会自动将字符串 `x` 转为数值，但是由于 `x` 不是数值，所以最后得到结果为 `NaN` ，表示它是“非数字”（ `NaN` ）。

另外，一些数学函数的运算结果会出现 `NaN` 。

```
1. Math.acos(2) // NaN
2. Math.log(-1) // NaN
3. Math.sqrt(-1) // NaN
```

`0` 除以 `0` 也会得到 `NaN`。

```
0 / 0 // NaN
```

需要注意的是，`NaN` 不是独立的数据类型，而是一个特殊数值，它的数据类型依然属于 `Number`，使用 `typeof` 运算符可以看得很清楚。

```
typeof NaN // 'number'
```

（2）运算规则

`NaN` 不等于任何值，包括它本身。

```
NaN === NaN // false
```

数组的 `indexOf` 方法内部使用的是严格相等运算符，所以该方法对 `NaN` 不成立。

```
[NaN].indexOf(NaN) // -1
```

`NaN` 在布尔运算时被当作 `false`。

```
Boolean(NaN) // false
```

`NaN` 与任何数（包括它自己）的运算，得到的都是 `NaN`。

```
NaN + 32 // NaN
NaN - 32 // NaN
NaN * 32 // NaN
NaN / 32 // NaN
```

Infinity

（1）含义

`Infinity` 表示“无穷”，用来表示两种场景。一种是一个正的数值太大，或一个负的数值太小，无法表示；另一种是非0数值除以0，得到 `Infinity` 。

```
// 场景一
Math.pow(2, 1024)
// Infinity

// 场景二
0 / 0 // NaN
1 / 0 // Infinity
```

上面代码中，第一个场景是一个表达式的计算结果太大，超出了能够表示的范围，因此返回 `Infinity` 。第二个场景是 `0` 除以 `0` 会得到 `NaN` ，而非0数值除以 `0` ，会返回 `Infinity` 。

`Infinity` 有正负之分， `Infinity` 表示正的无穷， `-Infinity` 表示负的无穷。

```
Infinity === -Infinity // false

1 / -0 // -Infinity
-1 / -0 // Infinity
```

上面代码中，非零正数除以 `-0` ，会得到 `-Infinity` ，负数除以 `-0` ，会得到 `Infinity` 。

由于数值正向溢出（overflow）、负向溢出（underflow）和被 `0` 除，JavaScript 都不报错，而是返回 `Infinity` ，所以单纯的数学运算几乎没有可能抛出错误。

`Infinity` 大于一切数值（除了 `NaN` ）， `-Infinity` 小于一切数值（除了 `NaN` ）。

```
1. Infinity > 1000 // true
2. -Infinity < -1000 // true
```

`Infinity` 与 `NaN` 比较，总是返回 `false` 。

```
1. Infinity > NaN // false
2. -Infinity > NaN // false
3.
4. Infinity < NaN // false
5. -Infinity < NaN // false
```

（2）运算规则

`Infinity` 的四则运算，符合无穷的数学计算规则。

```
1. 5 * Infinity // Infinity
2. 5 - Infinity // -Infinity
3. Infinity / 5 // Infinity
4. 5 / Infinity // 0
```

0乘以 `Infinity` ，返回 `NaN` ；0除以 `Infinity` ，返回 `0` ； `Infinity` 除以0，返回 `Infinity` 。

```
1. 0 * Infinity // NaN
2. 0 / Infinity // 0
3. Infinity / 0 // Infinity
```

`Infinity` 加上或乘以 `Infinity` ，返回的还是 `Infinity` 。

```
1. Infinity + Infinity // Infinity
2. Infinity * Infinity // Infinity
```

`Infinity` 减去或除以 `Infinity` ，得到 `NaN` 。

```
1. Infinity - Infinity // NaN
```

```
2. Infinity / Infinity // NaN
```

`Infinity` 与 `null` 计算时， `null` 会转成0，等同于与 `0` 的计算。

```
1. null * Infinity // NaN
2. null / Infinity // 0
3. Infinity / null // Infinity
```

`Infinity` 与 `undefined` 计算，返回的都是 `NaN` 。

```
1. undefined + Infinity // NaN
2. undefined - Infinity // NaN
3. undefined * Infinity // NaN
4. undefined / Infinity // NaN
5. Infinity / undefined // NaN
```

与数值相关的全局方法

parseInt()

（1）基本用法

`parseInt` 方法用于将字符串转为整数。

```
1. parseInt('123') // 123
```

如果字符串头部有空格，空格会被自动去除。

```
1. parseInt('   81') // 81
```

如果 `parseInt` 的参数不是字符串，则会先转为字符串再转换。

```
1. parseInt(1.23) // 1
2. // 等同于
```

```
3. parseInt('1.23') // 1
```

字符串转为整数的时候，是一个个字符依次转换，如果遇到不能转为数字的字符，就不再进行下去，返回已经转好的部分。

```
1. parseInt('8a') // 8
2. parseInt('12**') // 12
3. parseInt('12.34') // 12
4. parseInt('15e2') // 15
5. parseInt('15px') // 15
```

上面代码中，`parseInt`的参数都是字符串，结果只返回字符串头部可以转为数字的部分。

如果字符串的第一个字符不能转化为数字（后面跟着数字的正负号除外），返回`NaN`。

```
1. parseInt('abc') // NaN
2. parseInt('.3') // NaN
3. parseInt('') // NaN
4. parseInt('+') // NaN
5. parseInt('+1') // 1
```

所以，`parseInt`的返回值只有两种可能，要么是一个十进制整数，要么是`NaN`。

如果字符串以`0x`或`0X`开头，`parseInt`会将其按照十六进制数解析。

```
1. parseInt('0x10') // 16
```

如果字符串以`0`开头，将其按照10进制解析。

```
1. parseInt('011') // 11
```

对于那些会自动转为科学计数法的数字，`parseInt`会将科学计数法的表示方法视为字符串，因此导致一些奇怪的结果。

```
parseInt(1000000000000000000000.5) // 1
// 等同于
parseInt('1e+21') // 1

parseInt(0.0000008) // 8
// 等同于
parseInt('8e-7') // 8
```

（2）进制转换

`parseInt`方法还可以接受第二个参数（2到36之间），表示被解析的值的进制，返回该值对应的十进制数。默认情况下，`parseInt`的第二个参数为10，即默认是十进制转十进制。

```
parseInt('1000') // 1000
// 等同于
parseInt('1000', 10) // 1000
```

下面是转换指定进制的数的例子。

```
parseInt('1000', 2) // 8
parseInt('1000', 6) // 216
parseInt('1000', 8) // 512
```

上面代码中，二进制、六进制、八进制的`1000`，分别等于十进制的8、216和512。这意味着，可以用`parseInt`方法进行进制的转换。

如果第二个参数不是数值，会被自动转为一个整数。这个整数只有在2到36之间，才能得到有意义的结果，超出这个范围，则返回`NaN`。如果第二个参数是`0`、`undefined`和`null`，则直接忽略。

```
parseInt('10', 37) // NaN
parseInt('10', 1) // NaN
parseInt('10', 0) // 10
parseInt('10', null) // 10
parseInt('10', undefined) // 10
```

如果字符串包含对于指定进制无意义的字符，则从最高位开始，只返回可以转换的数值。如果最高位无法转换，则直接返回 `NaN` 。

```
parseInt('1546', 2) // 1
parseInt('546', 2) // NaN
```

上面代码中，对于二进制来说， `1` 是有意义的字符， `5` 、 `4` 、 `6` 都是无意义的字符，所以第一行返回1，第二行返回 `NaN` 。

前面说过，如果 `parseInt` 的第一个参数不是字符串，会被先转为字符串。这会导致一些令人意外的结果。

```
parseInt(0x11, 36) // 43
parseInt(0x11, 2) // 1

// 等同于
parseInt(String(0x11), 36)
parseInt(String(0x11), 2)

// 等同于
parseInt('17', 36)
parseInt('17', 2)
```

上面代码中，十六进制的 `0x11` 会被先转为十进制的17，再转为字符串。然后，再用36进制或二进制解读字符串 `17` ，最后返回结果 `43` 和 `1` 。

这种处理方式，对于八进制的前缀0，尤其需要注意。

```
parseInt(011, 2) // NaN

// 等同于
parseInt(String(011), 2)

// 等同于
parseInt(String(9), 2)
```

上面代码中，第一行的 `011` 会被先转为字符串 `9` ，因为 `9` 不是二进制的有效字符，所以返回 `NaN` 。如果直接计算 `parseInt('011', 2)` ， `011` 则是会被当作二进制处理，返回3。

JavaScript 不再允许将带有前缀0的数字视为八进制数，而是要求忽略这个 `0` 。但是，为了保证兼容性，大部分浏览器并没有部署这一条规定。

parseFloat()

`parseFloat` 方法用于将一个字符串转为浮点数。

```
parseFloat('3.14') // 3.14
```

如果字符串符合科学计数法，则会进行相应的转换。

```
parseFloat('314e-2') // 3.14
parseFloat('0.0314E+2') // 3.14
```

如果字符串包含不能转为浮点数的字符，则不再进行往后转换，返回已经转好的部分。

```
parseFloat('3.14more non-digit characters') // 3.14
```

`parseFloat` 方法会自动过滤字符串前导的空格。

```
parseFloat('\t\v\r12.34\n ') // 12.34
```

如果参数不是字符串，或者字符串的第一个字符不能转化为浮点数，则返回 `NaN` 。

```
parseFloat([]) // NaN
parseFloat('FF2') // NaN
parseFloat('') // NaN
```

上面代码中，尤其值得注意， `parseFloat` 会将空字符串转为 `NaN` 。

这些特点使得 `parseFloat` 的转换结果不同于 `Number` 函数。

```
parseFloat(true)  // NaN
Number(true) // 1

parseFloat(null) // NaN
Number(null) // 0

parseFloat('') // NaN
Number('') // 0

parseFloat('123.45#') // 123.45
Number('123.45#') // NaN
```

isNaN()

`isNaN` 方法可以用来判断一个值是否为 `NaN` 。

```
isNaN(NaN) // true
isNaN(123) // false
```

但是， `isNaN` 只对数值有效，如果传入其他值，会被先转成数值。比

如，传入字符串的时候，字符串会被先转成 `NaN` ，所以最后返回 `true` ，这一点要特别引起注意。也就是说， `isNaN` 为 `true` 的值，有可能不是 `NaN` ，而是一个字符串。

```
isNaN('Hello') // true
// 相当于
isNaN(Number('Hello')) // true
```

出于同样的原因，对于对象和数组， `isNaN` 也返回 `true` 。

```
isNaN({}) // true
// 等同于
isNaN(Number({})) // true

isNaN(['xzy']) // true
// 等同于
isNaN(Number(['xzy'])) // true
```

但是，对于空数组和只有一个数值成员的数组， `isNaN` 返回 `false` 。

```
isNaN([]) // false
isNaN([123]) // false
isNaN(['123']) // false
```

上面代码之所以返回 `false` ，原因是这些数组能被 `Number` 函数转成数值，请参见《数据类型转换》一章。

因此，使用 `isNaN` 之前，最好判断一下数据类型。

```
function myIsNaN(value) {
  return typeof value === 'number' && isNaN(value);
}
```

判断 `NaN` 更可靠的方法是，利用 `NaN` 为唯一不等于自身的值的这个特

点，进行判断。

```
function myIsNaN(value) {
  return value !== value;
}
```

isFinite()

`isFinite` 方法返回一个布尔值，表示某个值是否为正常的数值。

```
isFinite(Infinity) // false
isFinite(-Infinity) // false
isFinite(NaN) // false
isFinite(-1) // true
```

除了 `Infinity` 、 `-Infinity` 和 `NaN` 这三个值会返回 `false` ， `isFinite` 对于其他的数值都会返回 `true` 。

参考链接

- Dr. Axel Rauschmayer, How numbers are encoded in JavaScript
- Humphry, JavaScript 中 Number 的一些表示上/下限

字符串

字符串

概述

定义

字符串就是零个或多个排在一起的字符，放在单引号或双引号之中。

```
1. 'abc'
2. "abc"
```

单引号字符串的内部，可以使用双引号。双引号字符串的内部，可以使用单引号。

```
1. 'key = "value"'
2. "It's a long journey"
```

上面两个都是合法的字符串。

如果要在单引号字符串的内部，使用单引号，就必须在内部的单引号前面加上反斜杠，用来转义。双引号字符串内部使用双引号，也是如此。

```
'Did she say \'Hello\'?'
// "Did she say 'Hello'?"

"Did she say \"Hello\"?"
// "Did she say "Hello"?"
```

由于 HTML 语言的属性值使用双引号，所以很多项目约定 JavaScript 语言的字符串只使用单引号，本教程遵守这个约定。当然，只使用双引号也完全可以。重要的是坚持使用一种风格，不要一会使用单引号表示字符串，一会又使用双引号表示。

字符串默认只能写在一行内，分成多行将会报错。

```
'a
b
c'
// SyntaxError: Unexpected token ILLEGAL
```

上面代码将一个字符串分成三行，JavaScript 就会报错。

如果长字符串必须分成多行，可以在每一行的尾部使用反斜杠。

```
var longString = 'Long \
long \
long \
string';

longString
// "Long long long string"
```

上面代码表示，加了反斜杠以后，原来写在一行的字符串，可以分成多行书写。但是，输出的时候还是单行，效果与写在同一行完全一样。注

意，反斜杠的后面必须是换行符，而不能有其他字符（比如空格），否则会报错。

连接运算符（`+`）可以连接多个单行字符串，将长字符串拆成多行书写，输出的时候也是单行。

```
var longString = 'Long '
  + 'long '
  + 'long '
  + 'string';
```

如果想输出多行字符串，有一种利用多行注释的变通方法。

```
(function () { /*
line 1
line 2
line 3
*/}).toString().split('\n').slice(1, -1).join('\n')
// "line 1
// line 2
// line 3"
```

上面的例子中，输出的字符串就是多行。

转义

反斜杠（\）在字符串内有特殊含义，用来表示一些特殊字符，所以又称为转义符。

需要用反斜杠转义的特殊字符，主要有下面这些。

- `\0` ：null（`\u0000`）
- `\b` ：后退键（`\u0008`）
- `\f` ：换页符（`\u000C`）

- `\n` ：换行符（`\u000A`）
- `\r` ：回车键（`\u000D`）
- `\t` ：制表符（`\u0009`）
- `\v` ：垂直制表符（`\u000B`）
- `\'` ：单引号（`\u0027`）
- `\"` ：双引号（`\u0022`）
- `\\` ：反斜杠（`\u005C`）

上面这些字符前面加上反斜杠，都表示特殊含义。

```
1. console.log('1\n2')
2. // 1
3. // 2
```

上面代码中，`\n` 表示换行，输出的时候就分成了两行。

反斜杠还有三种特殊用法。

（1）`\HHH`

反斜杠后面紧跟三个八进制数（`000` 到 `377`），代表一个字符。`HHH` 对应该字符的 Unicode 码点，比如 `\251` 表示版权符号。显然，这种方法只能输出256种字符。

（2）`\xHH`

`\x` 后面紧跟两个十六进制数（`00` 到 `FF`），代表一个字符。`HH` 对应该字符的 Unicode 码点，比如 `\xA9` 表示版权符号。这种方法也只能输出256种字符。

（3）`\uXXXX`

`\u` 后面紧跟四个十六进制数（`0000` 到 `FFFF`），代表一个字符。`HHHH` 对应该字符的 Unicode 码点，比如 `\u00A9` 表示版权符

号。

下面是这三种字符特殊写法的例子。

```
'\251' // "©"
'\xA9' // "©"
'\u00A9' // "©"

'\172' === 'z' // true
'\x7A' === 'z' // true
'\u007A' === 'z' // true
```

如果在非特殊字符前面使用反斜杠，则反斜杠会被省略。

```
'\a'
// "a"
```

上面代码中，`a`是一个正常字符，前面加反斜杠没有特殊含义，反斜杠会被自动省略。

如果字符串的正常内容之中，需要包含反斜杠，则反斜杠前面需要再加一个反斜杠，用来对自身转义。

```
"Prev \\ Next"
// "Prev \ Next"
```

字符串与数组

字符串可以被视为字符数组，因此可以使用数组的方括号运算符，用来返回某个位置的字符（位置编号从0开始）。

```
var s = 'hello';
s[0] // "h"
s[1] // "e"
s[4] // "o"
```

```
5.
6.  // 直接对字符串使用方括号运算符
7.  'hello'[1] // "e"
```

如果方括号中的数字超过字符串的长度，或者方括号中根本不是数字，则返回 `undefined` 。

```
1.  'abc'[3] // undefined
2.  'abc'[-1] // undefined
3.  'abc'['x'] // undefined
```

但是，字符串与数组的相似性仅此而已。实际上，无法改变字符串之中的单个字符。

```
1.  var s = 'hello';
2.
3.  delete s[0];
4.  s // "hello"
5.
6.  s[1] = 'a';
7.  s // "hello"
8.
9.  s[5] = '!';
10.  s // "hello"
```

上面代码表示，字符串内部的单个字符无法改变和增删，这些操作会默默地失败。

length 属性

`length` 属性返回字符串的长度，该属性也是无法改变的。

```
1.  var s = 'hello';
2.  s.length // 5
3.
```

```
s.length = 3;
s.length // 5

s.length = 7;
s.length // 5
```

上面代码表示字符串的 `length` 属性无法改变，但是不会报错。

字符集

JavaScript 使用 Unicode 字符集。JavaScript 引擎内部，所有字符都用 Unicode 表示。

JavaScript 不仅以 Unicode 储存字符，还允许直接在程序中使用 Unicode 码点表示字符，即将字符写成 `\uxxxx` 的形式，其中 `xxxx` 代表该字符的 Unicode 码点。比如， `\u00A9` 代表版权符号。

```
var s = '\u00A9';
s // "©"
```

解析代码的时候，JavaScript 会自动识别一个字符是字面形式表示，还是 Unicode 形式表示。输出给用户的时候，所有字符都会转成字面形式。

```
var f\u006F\u006F = 'abc';
foo // "abc"
```

上面代码中，第一行的变量名 `foo` 是 Unicode 形式表示，第二行是字面形式表示。JavaScript 会自动识别。

我们还需要知道，每个字符在 JavaScript 内部都是以16位（即2个字节）的 UTF-16 格式储存。也就是说，JavaScript 的单位字符

长度固定为16位长度，即2个字节。

但是，UTF-16 有两种长度：对于码点在 `U+0000` 到 `U+FFFF` 之间的字符，长度为16位（即2个字节）；对于码点在 `U+10000` 到 `U+10FFFF` 之间的字符，长度为32位（即4个字节），而且前两个字节在 `0xD800` 到 `0xDBFF` 之间，后两个字节在 `0xDC00` 到 `0xDFFF` 之间。举例来说，码点 `U+1D306` 对应的字符为 `?`，它写成 UTF-16 就是 `0xD834 0xDF06`。

JavaScript 对 UTF-16 的支持是不完整的，由于历史原因，只支持两字节的字符，不支持四字节的字符。这是因为 JavaScript 第一版发布的时候，Unicode 的码点只编到 `U+FFFF`，因此两字节足够表示了。后来，Unicode 纳入的字符越来越多，出现了四字节的编码。但是，JavaScript 的标准此时已经定型了，统一将字符长度限制在两字节，导致无法识别四字节的字符。上一节的那个四字节字符 `?`，浏览器会正确识别这是一个字符，但是 JavaScript 无法识别，会认为这是两个字符。

```
1.  '?'.length // 2
```

上面代码中，JavaScript 认为 `?` 的长度为2，而不是1。

总结一下，对于码点在 `U+10000` 到 `U+10FFFF` 之间的字符，JavaScript 总是认为它们是两个字符（`length` 属性为2）。所以处理的时候，必须把这一点考虑在内，也就是说，JavaScript 返回的字符串长度可能是不正确的。

Base64 转码

有时，文本里面包含一些不可打印的符号，比如 ASCII 码0到31的符号都无法打印出来，这时可以使用 Base64 编码，将它们转成可以打

印的字符。另一个场景是，有时需要以文本格式传递二进制数据，那么也可以使用 Base64 编码。

所谓 Base64 就是一种编码方法，可以将任意值转成 0～9、A～Z、a-z、`+`和`/`这64个字符组成的可打印字符。使用它的主要目的，不是为了加密，而是为了不出现特殊字符，简化程序的处理。

JavaScript 原生提供两个 Base64 相关的方法。

- `btoa()`：任意值转为 Base64 编码
- `atob()`：Base64 编码转为原来的值

```
var string = 'Hello World!';
btoa(string) // "SGVsbG8gV29ybGQh"
atob('SGVsbG8gV29ybGQh') // "Hello World!"
```

注意，这两个方法不适合非 ASCII 码的字符，会报错。

```
btoa('你好') // 报错
```

要将非 ASCII 码字符转为 Base64 编码，必须中间插入一个转码环节，再使用这两个方法。

```
function b64Encode(str) {
  return btoa(encodeURIComponent(str));
}

function b64Decode(str) {
  return decodeURIComponent(atob(str));
}

b64Encode('你好') // "JUU0JUJEJUEwJUU1JUE1JUJE"
b64Decode('JUU0JUJEJUEwJUU1JUE1JUJE') // "你好"
```

参考链接

- Mathias Bynens, JavaScript’s internal character encoding: UCS-2 or UTF-16?
- Mathias Bynens, JavaScript has a Unicode problem
- Mozilla Developer Network, Window.btoa

对象

对象

概述

生成方法

对象（object）是 JavaScript 语言的核心概念，也是最重要的数据类型。

什么是对象？简单说，对象就是一组“键值对”（key-value）的集合，是一种无序的复合数据集合。

```
var obj = {
  foo: 'Hello',
  bar: 'World'
};
```

上面代码中，大括号就定义了一个对象，它被赋值给变量 `obj` ，所以变量 `obj` 就指向一个对象。该对象内部包含两个键值对（又称为两个“成员”），第一个键值对是 `foo: 'Hello'` ，其中 `foo` 是“键名”（成员的名称），字符串 `Hello` 是“键值”（成员的值）。键名与键值之间用冒号分隔。第二个键值对是 `bar: 'World'` ， `bar` 是键名， `World` 是键值。两个键值对之间用逗号分隔。

键名

对象的所有键名都是字符串（ES6 又引入了 Symbol 值也可以作为键值），所以加不加引号都可以。上面的代码也可以写成下面这样。

```
var obj = {
  'foo': 'Hello',
  'bar': 'World'
};
```

如果键名是数值，会被自动转为字符串。

```
var obj = {
  1: 'a',
  3.2: 'b',
  1e2: true,
  1e-2: true,
  .234: true,
  0xFF: true
};

obj
```

```
// Object {
//   1: "a",
//   3.2: "b",
//   100: true,
//   0.01: true,
//   0.234: true,
//   255: true
// }

obj['100'] // true
```

上面代码中，对象 `obj` 的所有键名虽然看上去像数值，实际上都被自动转成了字符串。

如果键名不符合标识名的条件（比如第一个字符为数字，或者含有空格或运算符），且也不是数字，则必须加上引号，否则会报错。

```
// 报错
var obj = {
  1p: 'Hello World'
};

// 不报错
var obj = {
  '1p': 'Hello World',
  'h w': 'Hello World',
  'p+q': 'Hello World'
};
```

上面对象的三个键名，都不符合标识名的条件，所以必须加上引号。

对象的每一个键名又称为“属性”（property），它的“键值”可以是任何数据类型。如果一个属性的值为函数，通常把这个属性称为“方法”，它可以像函数那样调用。

```
var obj = {
```

```
  p: function (x) {
    return 2 * x;
  }
};

obj.p(1) // 2
```

上面代码中，对象 `obj` 的属性 `p` ，就指向一个函数。

如果属性的值还是一个对象，就形成了链式引用。

```
var o1 = {};
var o2 = { bar: 'hello' };

o1.foo = o2;
o1.foo.bar // "hello"
```

上面代码中，对象 `o1` 的属性 `foo` 指向对象 `o2` ，就可以链式引用 `o2` 的属性。

对象的属性之间用逗号分隔，最后一个属性后面可以加逗号（trailing comma），也可以不加。

```
var obj = {
  p: 123,
  m: function () { ... },
}
```

上面的代码中， `m` 属性后面的那个逗号，有没有都可以。

属性可以动态创建，不必在对象声明时就指定。

```
var obj = {};
obj.foo = 123;
obj.foo // 123
```

上面代码中，直接对 `obj` 对象的 `foo` 属性赋值，结果就在运行时创建了 `foo` 属性。

对象的引用

如果不同的变量名指向同一个对象，那么它们都是这个对象的引用，也就是说指向同一个内存地址。修改其中一个变量，会影响到其他所有变量。

```
var o1 = {};
var o2 = o1;

o1.a = 1;
o2.a // 1

o2.b = 2;
o1.b // 2
```

上面代码中， `o1` 和 `o2` 指向同一个对象，因此为其中任何一个变量添加属性，另一个变量都可以读写该属性。

此时，如果取消某一个变量对于原对象的引用，不会影响到另一个变量。

```
var o1 = {};
var o2 = o1;

o1 = 1;
o2 // {}
```

上面代码中， `o1` 和 `o2` 指向同一个对象，然后 `o1` 的值变为1，这时不会对 `o2` 产生影响， `o2` 还是指向原来的那个对象。

但是，这种引用只局限于对象，如果两个变量指向同一个原始类型的

值。那么，变量这时都是值的拷贝。

```
var x = 1;
var y = x;

x = 2;
y // 1
```

上面的代码中，当 `x` 的值发生变化后， `y` 的值并不变，这就表示 `y` 和 `x` 并不是指向同一个内存地址。

表达式还是语句？

对象采用大括号表示，这导致了一个问题：如果行首是一个大括号，它到底是表达式还是语句？

```
{ foo: 123 }
```

JavaScript 引擎读到上面这行代码，会发现可能有两种含义。第一种可能是，这是一个表达式，表示一个包含 `foo` 属性的对象；第二种可能是，这是一个语句，表示一个代码区块，里面有一个标签 `foo` ，指向表达式 `123` 。

为了避免这种歧义，JavaScript 规定，如果行首是大括号，一律解释为语句（即代码块）。如果要解释为表达式（即对象），必须在大括号前加上圆括号。

```
({ foo: 123})
```

这种差异在 `eval` 语句（作用是对字符串求值）中反映得最明显。

```
eval('{foo: 123}') // 123
eval('({foo: 123})') // {foo: 123}
```

上面代码中，如果没有圆括号，`eval`将其理解为一个代码块；加上圆括号以后，就理解成一个对象。

属性的操作

属性的读取

读取对象的属性，有两种方法，一种是使用点运算符，还有一种是使用方括号运算符。

```
var obj = {
  p: 'Hello World'
};

obj.p // "Hello World"
obj['p'] // "Hello World"
```

上面代码分别采用点运算符和方括号运算符，读取属性`p`。

请注意，如果使用方括号运算符，键名必须放在引号里面，否则会被当作变量处理。

```
var foo = 'bar';

var obj = {
  foo: 1,
  bar: 2
};

obj.foo  // 1
obj[foo]  // 2
```

上面代码中，引用对象`obj`的`foo`属性时，如果使用点运算符，`foo`就是字符串；如果使用方括号运算符，但是不使用引号，那

么 `foo` 就是一个变量，指向字符串 `bar` 。

方括号运算符内部还可以使用表达式。

```
obj['hello' + ' world']
obj[3 + 3]
```

数字键可以不加引号，因为会自动转成字符串。

```
var obj = {
  0.7: 'Hello World'
};

obj['0.7'] // "Hello World"
obj[0.7] // "Hello World"
```

上面代码中，对象 `obj` 的数字键 `0.7` ，加不加引号都可以，因为会被自动转为字符串。

注意，数值键名不能使用点运算符（因为会被当成小数点），只能使用方括号运算符。

```
var obj = {
  123: 'hello world'
};

obj.123 // 报错
obj[123] // "hello world"
```

上面代码的第一个表达式，对数值键名 `123` 使用点运算符，结果报错。第二个表达式使用方括号运算符，结果就是正确的。

属性的赋值

点运算符和方括号运算符，不仅可以用来读取值，还可以用来赋值。

```
var obj = {};

obj.foo = 'Hello';
obj['bar'] = 'World';
```

上面代码中，分别使用点运算符和方括号运算符，对属性赋值。

JavaScript 允许属性的“后绑定”，也就是说，你可以在任意时刻新增属性，没必要在定义对象的时候，就定义好属性。

```
var obj = { p: 1 };

// 等价于

var obj = {};
obj.p = 1;
```

属性的查看

查看一个对象本身的所有属性，可以使用 `Object.keys` 方法。

```
var obj = {
  key1: 1,
  key2: 2
};

Object.keys(obj);
// ['key1', 'key2']
```

属性的删除：delete 命令

`delete` 命令用于删除对象的属性，删除成功后返回 `true` 。

```
var obj = { p: 1 };
Object.keys(obj) // ["p"]
```

```
3.
4.  delete obj.p // true
5.  obj.p // undefined
6.  Object.keys(obj) // []
```

上面代码中，`delete`命令删除对象`obj`的`p`属性。删除后，再读取`p`属性就会返回`undefined`，而且`Object.keys`方法的返回值也不再包括该属性。

注意，删除一个不存在的属性，`delete`不报错，而且返回`true`。

```
1.  var obj = {};
2.  delete obj.p // true
```

上面代码中，对象`obj`并没有`p`属性，但是`delete`命令照样返回`true`。因此，不能根据`delete`命令的结果，认定某个属性是存在的。

只有一种情况，`delete`命令会返回`false`，那就是该属性存在，且不得删除。

```
1.  var obj = Object.defineProperty({}, 'p', {
2.    value: 123,
3.    configurable: false
4.  });
5.
6.  obj.p // 123
7.  delete obj.p // false
```

上面代码之中，对象`obj`的`p`属性是不能删除的，所以`delete`命令返回`false`（关于`Object.defineProperty`方法的介绍，请看《标准库》的 Object 对象一章）。

另外，需要注意的是，`delete`命令只能删除对象本身的属性，无法删

除继承的属性（关于继承参见《面向对象编程》章节）。

```
var obj = {};
delete obj.toString // true
obj.toString // function toString() { [native code] }
```

上面代码中，`toString` 是对象 `obj` 继承的属性，虽然 `delete` 命令返回 `true`，但该属性并没有被删除，依然存在。这个例子还说明，即使 `delete` 返回 `true`，该属性依然可能读取到值。

属性是否存在：in 运算符

`in` 运算符用于检查对象是否包含某个属性（注意，检查的是键名，不是键值），如果包含就返回 `true`，否则返回 `false`。它的左边是一个字符串，表示属性名，右边是一个对象。

```
var obj = { p: 1 };
'p' in obj // true
'toString' in obj // true
```

`in` 运算符的一个问题是，它不能识别哪些属性是对象自身的，哪些属性是继承的。就像上面代码中，对象 `obj` 本身并没有 `toString` 属性，但是 `in` 运算符会返回 `true`，因为这个属性是继承的。

这时，可以使用对象的 `hasOwnProperty` 方法判断一下，是否为对象自身的属性。

```
var obj = {};
if ('toString' in obj) {
  console.log(obj.hasOwnProperty('toString')) // false
}
```

属性的遍历：for…in 循环

`for...in` 循环用来遍历一个对象的全部属性。

```
var obj = {a: 1, b: 2, c: 3};

for (var i in obj) {
  console.log('键名：', i);
  console.log('键值：', obj[i]);
}
// 键名： a
// 键值： 1
// 键名： b
// 键值： 2
// 键名： c
// 键值： 3
```

`for...in` 循环有两个使用注意点。

- 它遍历的是对象所有可遍历（enumerable）的属性，会跳过不可遍历的属性。
- 它不仅遍历对象自身的属性，还遍历继承的属性。

举例来说，对象都继承了 `toString` 属性，但是 `for...in` 循环不会遍历到这个属性。

```
var obj = {};

// toString 属性是存在的
obj.toString // toString() { [native code] }

for (var p in obj) {
  console.log(p);
} // 没有任何输出
```

上面代码中，对象 `obj` 继承了 `toString` 属性，该属性不会被 `for...in` 循环遍历到，因为它默认是“不可遍历”的。关于对象属性

的可遍历性，参见《标准库》章节中 Object 一章的介绍。

如果继承的属性是可遍历的，那么就会被 `for...in` 循环遍历到。但是，一般情况下，都是只想遍历对象自身的属性，所以使用 `for...in` 的时候，应该结合使用 `hasOwnProperty` 方法，在循环内部判断一下，某个属性是否为对象自身的属性。

```
var person = { name: '老张' };

for (var key in person) {
  if (person.hasOwnProperty(key)) {
    console.log(key);
  }
}
// name
```

with 语句

`with` 语句的格式如下：

```
with (对象) {
  语句;
}
```

它的作用是操作同一个对象的多个属性时，提供一些书写的方便。

```
// 例一
var obj = {
  p1: 1,
  p2: 2,
};
with (obj) {
  p1 = 4;
  p2 = 5;
}
```

```
// 等同于
obj.p1 = 4;
obj.p2 = 5;

// 例二
with (document.links[0]){
  console.log(href);
  console.log(title);
  console.log(style);
}
// 等同于
console.log(document.links[0].href);
console.log(document.links[0].title);
console.log(document.links[0].style);
```

注意，如果 `with` 区块内部有变量的赋值操作，必须是当前对象已经存在的属性，否则会创造一个当前作用域的全局变量。

```
var obj = {};
with (obj) {
  p1 = 4;
  p2 = 5;
}

obj.p1 // undefined
p1 // 4
```

上面代码中，对象 `obj` 并没有 `p1` 属性，对 `p1` 赋值等于创造了一个全局变量 `p1` 。正确的写法应该是，先定义对象 `obj` 的属性 `p1` ，然后在 `with` 区块内操作它。

这是因为 `with` 区块没有改变作用域，它的内部依然是当前作用域。这造成了 `with` 语句的一个很大的弊病，就是绑定对象不明确。

```
with (obj) {
  console.log(x);
```

```
3. }
```

单纯从上面的代码块，根本无法判断 `x` 到底是全局变量，还是对象 `obj` 的一个属性。这非常不利于代码的除错和模块化，编译器也无法对这段代码进行优化，只能留到运行时判断，这就拖慢了运行速度。因此，建议不要使用 `with` 语句，可以考虑用一个临时变量代替 `with` 。

```
1. with(obj1.obj2.obj3) {
2.   console.log(p1 + p2);
3. }
4.
5. // 可以写成
6. var temp = obj1.obj2.obj3;
7. console.log(temp.p1 + temp.p2);
```

参考链接

- Dr. Axel Rauschmayer, Object properties in JavaScript
- Lakshan Perera, Revisiting JavaScript Objects
- Angus Croll, The Secret Life of JavaScript Primitivesi
- Dr. Axel Rauschmayer, JavaScript’s with statement and why it’s deprecated

函数

函数

函数是一段可以反复调用的代码块。函数还能接受输入的参数，不同的参数会返回不同的值。

概述

函数的声明

JavaScript 有三种声明函数的方法。

（1）function 命令

`function` 命令声明的代码区块，就是一个函数。 `function` 命令后面是函数名，函数名后面是一对圆括号，里面是传入函数的参数。函数体放在大括号里面。

```
function print(s) {
  console.log(s);
}
```

上面的代码命名了一个 `print` 函数，以后使用 `print()` 这种形式，就可以调用相应的代码。这叫做函数的声明（Function Declaration）。

（2）函数表达式

除了用 `function` 命令声明函数，还可以采用变量赋值的写法。

```
var print = function(s) {
```

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

这种写法将一个匿名函数赋值给变量。这时，这个匿名函数又称函数表达式（Function Expression），因为赋值语句的等号右侧只能放表达式。

采用函数表达式声明函数时，`function` 命令后面不带有函数名。如果加上函数名，该函数名只在函数体内部有效，在函数体外部无效。

```
var print = function x(){
  console.log(typeof x);
};

x
// ReferenceError: x is not defined

print()
// function
```

上面代码在函数表达式中，加入了函数名 `x`。这个 `x` 只在函数体内部可用，指代函数表达式本身，其他地方都不可用。这种写法的用处有两个，一是可以在函数体内部调用自身，二是方便除错（除错工具显示函数调用栈时，将显示函数名，而不再显示这里是一个匿名函数）。因此，下面的形式声明函数也非常常见。

```
var f = function f() {};
```

需要注意的是，函数的表达式需要在语句的结尾加上分号，表示语句结束。而函数的声明在结尾的大括号后面不用加分号。总的来说，这两种声明函数的方式，差别很细微，可以近似认为是等价的。

（3）Function 构造函数

第三种声明函数的方式是 `Function` 构造函数。

```
var add = new Function(
  'x',
  'y',
  'return x + y'
);

// 等同于
function add(x, y) {
  return x + y;
}
```

上面代码中， `Function` 构造函数接受三个参数，除了最后一个参数是 `add` 函数的“函数体”，其他参数都是 `add` 函数的参数。

你可以传递任意数量的参数给 `Function` 构造函数，只有最后一个参数会被当做函数体，如果只有一个参数，该参数就是函数体。

```
var foo = new Function(
  'return "hello world"'
);

// 等同于
function foo() {
  return 'hello world';
}
```

`Function` 构造函数可以不使用 `new` 命令，返回结果完全一样。

总的来说，这种声明函数的方式非常不直观，几乎无人使用。

函数的重复声明

如果同一个函数被多次声明，后面的声明就会覆盖前面的声明。

```
function f() {
  console.log(1);
}
f() // 2

function f() {
  console.log(2);
}
f() // 2
```

上面代码中，后一次的函数声明覆盖了前面一次。而且，由于函数名的提升（参见下文），前一次声明在任何时候都是无效的，这一点要特别注意。

圆括号运算符，return 语句和递归

调用函数时，要使用圆括号运算符。圆括号之中，可以加入函数的参数。

```
function add(x, y) {
  return x + y;
}

add(1, 1) // 2
```

上面代码中，函数名后面紧跟一对圆括号，就会调用这个函数。

函数体内部的 `return` 语句，表示返回。JavaScript 引擎遇到 `return` 语句，就直接返回 `return` 后面的那个表达式的值，后面即使还有语句，也不会得到执行。也就是说， `return` 语句所带的那个表达式，就是函数的返回值。 `return` 语句不是必需的，如果没有的话，该函数就不返回任何值，或者说返回 `undefined` 。

函数可以调用自身，这就是递归（recursion）。下面就是通过递

归，计算斐波那契数列的代码。

```
function fib(num) {
  if (num === 0) return 0;
  if (num === 1) return 1;
  return fib(num - 2) + fib(num - 1);
}

fib(6) // 8
```

上面代码中，`fib`函数内部又调用了`fib`，计算得到斐波那契数列的第6个元素是8。

第一等公民

JavaScript 语言将函数看作一种值，与其它值（数值、字符串、布尔值等等）地位相同。凡是可以使用值的地方，就能使用函数。比如，可以把函数赋值给变量和对象的属性，也可以当作参数传入其他函数，或者作为函数的结果返回。函数只是一个可以执行的值，此外并无特殊之处。

由于函数与其他数据类型地位平等，所以在 JavaScript 语言中又称函数为第一等公民。

```
function add(x, y) {
  return x + y;
}

// 将函数赋值给一个变量
var operator = add;

// 将函数作为参数和返回值
function a(op){
  return op;
}
```

```
a(add)(1, 1)
// 2
```

函数名的提升

JavaScript 引擎将函数名视同变量名，所以采用 `function` 命令声明函数时，整个函数会像变量声明一样，被提升到代码头部。所以，下面的代码不会报错。

```
f();

function f() {}
```

表面上，上面代码好像在声明之前就调用了函数 `f` 。但是实际上，由于“变量提升”，函数 `f` 被提升到了代码头部，也就是在调用之前已经声明了。但是，如果采用赋值语句定义函数，JavaScript 就会报错。

```
f();
var f = function (){};
// TypeError: undefined is not a function
```

上面的代码等同于下面的形式。

```
var f;
f();
f = function () {};
```

上面代码第二行，调用 `f` 的时候， `f` 只是被声明了，还没有被赋值，等于 `undefined` ，所以会报错。因此，如果同时采用 `function` 命令和赋值语句声明同一个函数，最后总是采用赋值语句的定义。

```
var f = function () {
```

```
  console.log('1');
}

function f() {
  console.log('2');
}

f() // 1
```

函数的属性和方法

name 属性

函数的 `name` 属性返回函数的名字。

```
function f1() {}
f1.name // "f1"
```

如果是通过变量赋值定义的函数，那么 `name` 属性返回变量名。

```
var f2 = function () {};
f2.name // "f2"
```

但是，上面这种情况，只有在变量的值是一个匿名函数时才是如此。如果变量的值是一个具名函数，那么 `name` 属性返回 `function` 关键字之后的那个函数名。

```
var f3 = function myName() {};
f3.name // 'myName'
```

上面代码中， `f3.name` 返回函数表达式的名字。注意，真正的函数名还是 `f3` ，而 `myName` 这个名字只在函数体内部可用。

`name` 属性的一个用处，就是获取参数函数的名字。

```
var myFunc = function () {};

function test(f) {
  console.log(f.name);
}

test(myFunc) // myFunc
```

上面代码中，函数 `test` 内部通过 `name` 属性，就可以知道传入的参数是什么函数。

length 属性

函数的 `length` 属性返回函数预期传入的参数个数，即函数定义之中的参数个数。

```
function f(a, b) {}
f.length // 2
```

上面代码定义了空函数 `f` ，它的 `length` 属性就是定义时的参数个数。不管调用时输入了多少个参数， `length` 属性始终等于2。

`length` 属性提供了一种机制，判断定义时和调用时参数的差异，以便实现面向对象编程的”方法重载“（overload）。

toString()

函数的 `toString` 方法返回一个字符串，内容是函数的源码。

```
function f() {
  a();
  b();
  c();
}
```

```
6.
7.  f.toString()
8.  // function f() {
9.  //  a();
10. //  b();
11. //  c();
12. // }
```

函数内部的注释也可以返回。

```
1.  function f() {/*
2.    这是一个
3.    多行注释
4.  */}
5.
6.  f.toString()
7.  // "function f(){/*
8.  //   这是一个
9.  //   多行注释
10. // */}"
```

利用这一点，可以变相实现多行字符串。

```
1.  var multiline = function (fn) {
2.    var arr = fn.toString().split('\n');
3.    return arr.slice(1, arr.length - 1).join('\n');
4.  };
5.
6.  function f() {/*
7.    这是一个
8.    多行注释
9.  */}
10.
11. multiline(f);
12. // " 这是一个
13. //   多行注释"
```

函数作用域

定义

作用域（scope）指的是变量存在的范围。在 ES5 的规范中，Javascript 只有两种作用域：一种是全局作用域，变量在整个程序中一直存在，所有地方都可以读取；另一种是函数作用域，变量只在函数内部存在。ES6 又新增了块级作用域，本教程不涉及。

函数外部声明的变量就是全局变量（global variable），它可以在函数内部读取。

```
var v = 1;

function f() {
  console.log(v);
}

f()
// 1
```

上面的代码表明，函数 `f` 内部可以读取全局变量 `v` 。

在函数内部定义的变量，外部无法读取，称为“局部变量”（local variable）。

```
function f(){
  var v = 1;
}

v // ReferenceError: v is not defined
```

上面代码中，变量 `v` 在函数内部定义，所以是一个局部变量，函数之外就无法读取。

函数内部定义的变量，会在该作用域内覆盖同名全局变量。

```
var v = 1;

function f(){
  var v = 2;
  console.log(v);
}

f() // 2
v // 1
```

上面代码中，变量 `v` 同时在函数的外部和内部有定义。结果，在函数内部定义，局部变量 `v` 覆盖了全局变量 `v` 。

注意，对于 `var` 命令来说，局部变量只能在函数内部声明，在其他区块中声明，一律都是全局变量。

```
if (true) {
  var x = 5;
}
console.log(x);  // 5
```

上面代码中，变量 `x` 在条件判断区块之中声明，结果就是一个全局变量，可以在区块之外读取。

函数内部的变量提升

与全局作用域一样，函数作用域内部也会产生“变量提升”现象。 `var` 命令声明的变量，不管在什么位置，变量声明都会被提升到函数体的头部。

```
function foo(x) {
  if (x > 100) {
    var tmp = x - 100;
```

```
  }
}

// 等同于
function foo(x) {
  var tmp;
  if (x > 100) {
    tmp = x - 100;
  };
}
```

函数本身的作用域

函数本身也是一个值，也有自己的作用域。它的作用域与变量一样，就是其声明时所在的作用域，与其运行时所在的作用域无关。

```
var a = 1;
var x = function () {
  console.log(a);
};

function f() {
  var a = 2;
  x();
}

f() // 1
```

上面代码中，函数 `x` 是在函数 `f` 的外部声明的，所以它的作用域绑定外层，内部变量 `a` 不会到函数 `f` 体内取值，所以输出 `1`，而不是 `2`。

总之，函数执行时所在的作用域，是定义时的作用域，而不是调用时所在的作用域。

很容易犯错的一点是，如果函数 `A` 调用函数 `B` ，却没考虑到函数 `B` 不会引用函数 `A` 的内部变量。

```
var x = function () {
  console.log(a);
};

function y(f) {
  var a = 2;
  f();
}

y(x)
// ReferenceError: a is not defined
```

上面代码将函数 `x` 作为参数，传入函数 `y` 。但是，函数 `x` 是在函数 `y` 体外声明的，作用域绑定外层，因此找不到函数 `y` 的内部变量 `a` ，导致报错。

同样的，函数体内部声明的函数，作用域绑定函数体内部。

```
function foo() {
  var x = 1;
  function bar() {
    console.log(x);
  }
  return bar;
}

var x = 2;
var f = foo();
f() // 1
```

上面代码中，函数 `foo` 内部声明了一个函数 `bar` ， `bar` 的作用域绑定 `foo` 。当我们在 `foo` 外部取出 `bar` 执行时，变量 `x` 指向的

是 `foo` 内部的 `x`，而不是 `foo` 外部的 `x`。正是这种机制，构成了下文要讲解的“闭包”现象。

参数

概述

函数运行的时候，有时需要提供外部数据，不同的外部数据会得到不同的结果，这种外部数据就叫参数。

```
function square(x) {
  return x * x;
}

square(2) // 4
square(3) // 9
```

上式的 `x` 就是 `square` 函数的参数。每次运行的时候，需要提供这个值，否则得不到结果。

参数的省略

函数参数不是必需的，Javascript 允许省略参数。

```
function f(a, b) {
  return a;
}

f(1, 2, 3) // 1
f(1) // 1
f() // undefined

f.length // 2
```

上面代码的函数 `f` 定义了两个参数，但是运行时无论提供多少个参数（或者不提供参数），JavaScript 都不会报错。省略的参数的值就变为 `undefined`。需要注意的是，函数的 `length` 属性与实际传入的参数个数无关，只反映函数预期传入的参数个数。

但是，没有办法只省略靠前的参数，而保留靠后的参数。如果一定要省略靠前的参数，只有显式传入 `undefined`。

```
function f(a, b) {
  return a;
}

f( , 1) // SyntaxError: Unexpected token ,(…)
f(undefined, 1) // undefined
```

上面代码中，如果省略第一个参数，就会报错。

传递方式

函数参数如果是原始类型的值（数值、字符串、布尔值），传递方式是传值传递（passes by value）。这意味着，在函数体内修改参数值，不会影响到函数外部。

```
var p = 2;

function f(p) {
  p = 3;
}
f(p);

p // 2
```

上面代码中，变量 `p` 是一个原始类型的值，传入函数 `f` 的方式是传值传递。因此，在函数内部，`p` 的值是原始值的拷贝，无论怎么修

改，都不会影响到原始值。

但是，如果函数参数是复合类型的值（数组、对象、其他函数），传递方式是传址传递（pass by reference）。也就是说，传入函数的原始值的地址，因此在函数内部修改参数，将会影响到原始值。

```
var obj = { p: 1 };

function f(o) {
  o.p = 2;
}
f(obj);

obj.p // 2
```

上面代码中，传入函数 `f` 的是参数对象 `obj` 的地址。因此，在函数内部修改 `obj` 的属性 `p`，会影响到原始值。

注意，如果函数内部修改的，不是参数对象的某个属性，而是替换掉整个参数，这时不会影响到原始值。

```
var obj = [1, 2, 3];

function f(o) {
  o = [2, 3, 4];
}
f(obj);

obj // [1, 2, 3]
```

上面代码中，在函数 `f` 内部，参数对象 `obj` 被整个替换成另一个值。这时不会影响到原始值。这是因为，形式参数（`o`）的值实际是参数 `obj` 的地址，重新对 `o` 赋值导致 `o` 指向另一个地址，保存在原地址上的值当然不受影响。

同名参数

如果有同名的参数，则取最后出现的那个值。

```
function f(a, a) {
  console.log(a);
}

f(1, 2) // 2
```

上面代码中，函数 `f` 有两个参数，且参数名都是 `a` 。取值的时候，以后面的 `a` 为准，即使后面的 `a` 没有值或被省略，也是以其为准。

```
function f(a, a) {
  console.log(a);
}

f(1) // undefined
```

调用函数 `f` 的时候，没有提供第二个参数， `a` 的取值就变成了 `undefined` 。这时，如果要获得第一个 `a` 的值，可以使用 `arguments` 对象。

```
function f(a, a) {
  console.log(arguments[0]);
}

f(1) // 1
```

arguments 对象

（1）定义

由于 JavaScript 允许函数有不定数目的参数，所以需要一种机制，

可以在函数体内部读取所有参数。这就是 `arguments` 对象的由来。

`arguments` 对象包含了函数运行时的所有参数， `arguments[0]` 就是第一个参数， `arguments[1]` 就是第二个参数，以此类推。这个对象只有在函数体内部，才可以使用。

```
var f = function (one) {
  console.log(arguments[0]);
  console.log(arguments[1]);
  console.log(arguments[2]);
}

f(1, 2, 3)
// 1
// 2
// 3
```

正常模式下， `arguments` 对象可以在运行时修改。

```
var f = function(a, b) {
  arguments[0] = 3;
  arguments[1] = 2;
  return a + b;
}

f(1, 1) // 5
```

上面代码中，函数 `f` 调用时传入的参数，在函数内部被修改成 `3` 和 `2` 。

严格模式下， `arguments` 对象是一个只读对象，修改它是无效的，但不会报错。

```
var f = function(a, b) {
  'use strict'; // 开启严格模式
```

```
  arguments[0] = 3; // 无效
  arguments[1] = 2; // 无效
  return a + b;
}

f(1, 1) // 2
```

上面代码中，函数体内是严格模式，这时修改 `arguments` 对象就是无效的。

通过 `arguments` 对象的 `length` 属性，可以判断函数调用时到底带几个参数。

```
function f() {
  return arguments.length;
}

f(1, 2, 3) // 3
f(1) // 1
f() // 0
```

（2）与数组的关系

需要注意的是，虽然 `arguments` 很像数组，但它是一个对象。数组专有的方法（比如 `slice` 和 `forEach` ），不能在 `arguments` 对象上直接使用。

如果要让 `arguments` 对象使用数组方法，真正的解决方法是将 `arguments` 转为真正的数组。下面是两种常用的转换方法： `slice` 方法和逐一填入新数组。

```
var args = Array.prototype.slice.call(arguments);

// 或者
var args = [];
```

```
for (var i = 0; i < arguments.length; i++) {
  args.push(arguments[i]);
}
```

（3）callee 属性

`arguments` 对象带有一个 `callee` 属性，返回它所对应的原函数。

```
var f = function () {
  console.log(arguments.callee === f);
}

f() // true
```

可以通过 `arguments.callee` ，达到调用函数自身的目的。这个属性在严格模式里面是禁用的，因此不建议使用。

函数的其他知识点

闭包

闭包（closure）是 Javascript 语言的一个难点，也是它的特色，很多高级应用都要依靠闭包实现。

理解闭包，首先必须理解变量作用域。前面提到，JavaScript 有两种作用域：全局作用域和函数作用域。函数内部可以直接读取全局变量。

```
var n = 999;

function f1() {
  console.log(n);
}
f1() // 999
```

上面代码中，函数 `f1` 可以读取全局变量 `n` 。

但是，函数外部无法读取函数内部声明的变量。

```
function f1() {
  var n = 999;
}

console.log(n)
// Uncaught ReferenceError: n is not defined(
```

上面代码中，函数 `f1` 内部声明的变量 `n` ，函数外是无法读取的。

如果出于种种原因，需要得到函数内的局部变量。正常情况下，这是办不到的，只有通过变通方法才能实现。那就是在函数的内部，再定义一个函数。

```
function f1() {
  var n = 999;
  function f2() {
    console.log(n); // 999
  }
}
```

上面代码中，函数 `f2` 就在函数 `f1` 内部，这时 `f1` 内部的所有局部变量，对 `f2` 都是可见的。但是反过来就不行， `f2` 内部的局部变量，对 `f1` 就是不可见的。这就是 JavaScript 语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然 `f2` 可以读取 `f1` 的局部变量，那么只要把 `f2` 作为返回值，我们不就可以在 `f1` 外部读取它的内部变量了吗！

```
function f1() {
  var n = 999;
  function f2() {
    console.log(n);
  }
  return f2;
}

var result = f1();
result(); // 999
```

上面代码中，函数 `f1` 的返回值就是函数 `f2` ，由于 `f2` 可以读取 `f1` 的内部变量，所以就可以在外部获得 `f1` 的内部变量了。

闭包就是函数 `f2` ，即能够读取其他函数内部变量的函数。由于在 JavaScript 语言中，只有函数内部的子函数才能读取内部变量，因此可以把闭包简单理解成“定义在一个函数内部的函数”。闭包最大的特点，就是它可以“记住”诞生的环境，比如 `f2` 记住了它诞生的环境 `f1` ，所以从 `f2` 可以得到 `f1` 的内部变量。在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

闭包的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量始终保持在内存中，即闭包可以使得它诞生环境一直存在。请看下面的例子，闭包使得内部变量记住上一次调用时的运算结果。

```
function createIncrementor(start) {
  return function () {
    return start++;
  };
}

var inc = createIncrementor(5);

inc() // 5
inc() // 6
```

```
inc() // 7
```

上面代码中，`start`是函数`createIncrementor`的内部变量。通过闭包，`start`的状态被保留了，每一次调用都是在上一次调用的基础上进行计算。从中可以看到，闭包`inc`使得函数`createIncrementor`的内部环境，一直存在。所以，闭包可以看作是函数内部作用域的一个接口。

为什么会这样呢？原因就在于`inc`始终在内存中，而`inc`的存在依赖于`createIncrementor`，因此也始终在内存中，不会在调用结束后，被垃圾回收机制回收。

闭包的另一个用处，是封装对象的私有属性和私有方法。

```
function Person(name) {
  var _age;
  function setAge(n) {
    _age = n;
  }
  function getAge() {
    return _age;
  }

  return {
    name: name,
    getAge: getAge,
    setAge: setAge
  };
}

var p1 = Person('张三');
p1.setAge(25);
p1.getAge() // 25
```

上面代码中，函数`Person`的内部变量`_age`，通过闭

包 `getAge` 和 `setAge` ，变成了返回对象 `p1` 的私有变量。

注意，外层函数每次运行，都会生成一个新的闭包，而这个闭包又会保留外层函数的内部变量，所以内存消耗很大。因此不能滥用闭包，否则会造成网页的性能问题。

立即调用的函数表达式（IIFE）

在 Javascript 中，圆括号 `()` 是一种运算符，跟在函数名之后，表示调用该函数。比如， `print()` 就表示调用 `print` 函数。

有时，我们需要在定义函数之后，立即调用该函数。这时，你不能在函数的定义之后加上圆括号，这会产生语法错误。

```
function(){ /* code */ }();
// SyntaxError: Unexpected token (
```

产生这个错误的原因是， `function` 这个关键字即可以当作语句，也可以当作表达式。

```
// 语句
function f() {}

// 表达式
var f = function f() {}
```

为了避免解析上的歧义，JavaScript 引擎规定，如果 `function` 关键字出现在行首，一律解释成语句。因此，JavaScript引擎看到行首是 `function` 关键字之后，认为这一段都是函数的定义，不应该以圆括号结尾，所以就报错了。

解决方法就是不要让 `function` 出现在行首，让引擎将其理解成一个表达式。最简单的处理，就是将其放在一个圆括号里面。

```
(function(){ /* code */ }());
// 或者
(function(){ /* code */ })();
```

上面两种写法都是以圆括号开头，引擎就会认为后面跟的是一个表示式，而不是函数定义语句，所以就避免了错误。这就叫做“立即调用的函数表达式”（Immediately-Invoked Function Expression），简称 IIFE。

注意，上面两种写法最后的分号都是必须的。如果省略分号，遇到连着两个 IIFE，可能就会报错。

```
// 报错
(function(){ /* code */ }())
(function(){ /* code */ }())
```

上面代码的两行之间没有分号，JavaScript 会将它们连在一起解释，将第二行解释为第一行的参数。

推而广之，任何让解释器以表达式来处理函数定义的方法，都能产生同样的效果，比如下面三种写法。

```
var i = function(){ return 10; }();
true && function(){ /* code */ }();
0, function(){ /* code */ }();
```

甚至像下面这样写，也是可以的。

```
!function () { /* code */ }();
~function () { /* code */ }();
-function () { /* code */ }();
+function () { /* code */ }();
```

通常情况下，只对匿名函数使用这种“立即执行的函数表达式”。它的目

的有两个：一是不必为函数命名，避免了污染全局变量；二是 IIFE 内部形成了一个单独的作用域，可以封装一些外部无法读取的私有变量。

```
// 写法一
var tmp = newData;
processData(tmp);
storeData(tmp);

// 写法二
(function () {
  var tmp = newData;
  processData(tmp);
  storeData(tmp);
}());
```

上面代码中，写法二比写法一更好，因为完全避免了污染全局变量。

eval 命令

基本用法

`eval` 命令接受一个字符串作为参数，并将这个字符串当作语句执行。

```
eval('var a = 1;');
a // 1
```

上面代码将字符串当作语句运行，生成了变量 `a` 。

如果参数字符串无法当作语句运行，那么就会报错。

```
eval('3x') // Uncaught SyntaxError: Invalid or unexpected token
```

放在 `eval` 中的字符串，应该有独自存在的意义，不能用来与 `eval` 以

外的命令配合使用。举例来说，下面的代码将会报错。

```
eval('return;'); // Uncaught SyntaxError: Illegal return statement
```

上面代码会报错，因为 `return` 不能单独使用，必须在函数中使用。

如果 `eval` 的参数不是字符串，那么会原样返回。

```
eval(123) // 123
```

`eval` 没有自己的作用域，都在当前作用域内执行，因此可能会修改当前作用域的变量的值，造成安全问题。

```
var a = 1;
eval('a = 2');

a // 2
```

上面代码中， `eval` 命令修改了外部变量 `a` 的值。由于这个原因， `eval` 有安全风险。

为了防止这种风险，JavaScript 规定，如果使用严格模式， `eval` 内部声明的变量，不会影响到外部作用域。

```
(function f() {
  'use strict';
  eval('var foo = 123');
  console.log(foo);  // ReferenceError: foo is not defined
})()
```

上面代码中，函数 `f` 内部是严格模式，这时 `eval` 内部声明的 `foo` 变量，就不会影响到外部。

不过，即使在严格模式下， `eval` 依然可以读写当前作用域的变量。

```
(function f() {
  'use strict';
  var foo = 1;
  eval('foo = 2');
  console.log(foo);  // 2
})()
```

上面代码中，严格模式下，`eval` 内部还是改写了外部变量，可见安全风险依然存在。

总之，`eval` 的本质是在当前作用域之中，注入代码。由于安全风险和不利于 JavaScript 引擎优化执行速度，所以一般不推荐使用。通常情况下，`eval` 最常见的场合是解析 JSON 数据的字符串，不过正确的做法应该是使用原生的 `JSON.parse` 方法。

eval 的别名调用

前面说过 `eval` 不利于引擎优化执行速度。更麻烦的是，还有下面这种情况，引擎在静态代码分析的阶段，根本无法分辨执行的是 `eval`。

```
var m = eval;
m('var x = 1');
x // 1
```

上面代码中，变量 `m` 是 `eval` 的别名。静态代码分析阶段，引擎分辨不出 `m('var x = 1')` 执行的是 `eval` 命令。

为了保证 `eval` 的别名不影响代码优化，JavaScript 的标准规定，凡是使用别名执行 `eval`，`eval` 内部一律是全局作用域。

```
var a = 1;

function f() {
  var a = 2;
```

```
    var e = eval;
    e('console.log(a)');
}

f() // 1
```

上面代码中，`eval`是别名调用，所以即使它是在函数中，它的作用域还是全局作用域，因此输出的`a`为全局变量。这样的话，引擎就能确认`e()`不会对当前的函数作用域产生影响，优化的时候就可以把这一行排除掉。

`eval`的别名调用的形式五花八门，只要不是直接调用，都属于别名调用，因为引擎只能分辨`eval()`这一种形式是直接调用。

```
eval.call(null, '...')
window.eval('...')
(1, eval)('...')
(eval, eval)('...')
```

上面这些形式都是`eval`的别名调用，作用域都是全局作用域。

参考链接

- Ben Alman, Immediately-Invoked Function Expression (IIFE)
- Mark Daggett, Functions Explained
- Juriy Zaytsev, Named function expressions demystified
- Marco Rogers polotek, What is the arguments object?
- Juriy Zaytsev, Global eval. What are the options?

- Axel Rauschmayer, Evaluating JavaScript code via eval() and new Function()

数组

数组

定义

数组（array）是按次序排列的一组值。每个值的位置都有编号（从0开始），整个数组用方括号表示。

```
var arr = ['a', 'b', 'c'];
```

上面代码中的 `a`、 `b`、 `c` 就构成一个数组，两端的方括号是数组的标志。 `a` 是0号位置， `b` 是1号位置， `c` 是2号位置。

除了在定义时赋值，数组也可以先定义后赋值。

```
var arr = [];

arr[0] = 'a';
arr[1] = 'b';
arr[2] = 'c';
```

任何类型的数据，都可以放入数组。

```
var arr = [
  {a: 1},
  [1, 2, 3],
  function() {return true;}
];

arr[0] // Object {a: 1}
arr[1] // [1, 2, 3]
arr[2] // function (){return true;}
```

上面数组 `arr` 的3个成员依次是对象、数组、函数。

如果数组的元素还是数组，就形成了多维数组。

```
var a = [[1, 2], [3, 4]];
a[0][1] // 2
a[1][1] // 4
```

数组的本质

本质上，数组属于一种特殊的对象。 `typeof` 运算符会返回数组的类型是 `object` 。

```
typeof [1, 2, 3] // "object"
```

上面代码表明， `typeof` 运算符认为数组的类型就是对象。

数组的特殊性体现在，它的键名是按次序排列的一组整数（0，1，2…）。

```
var arr = ['a', 'b', 'c'];

Object.keys(arr)
```

```
// ["0", "1", "2"]
```

上面代码中，`Object.keys` 方法返回数组的所有键名。可以看到数组的键名就是整数0、1、2。

由于数组成员的键名是固定的（默认总是0、1、2…），因此数组不用为每个元素指定键名，而对象的每个成员都必须指定键名。JavaScript 语言规定，对象的键名一律为字符串，所以，数组的键名其实也是字符串。之所以可以用数值读取，是因为非字符串的键名会被转为字符串。

```
var arr = ['a', 'b', 'c'];

arr['0'] // 'a'
arr[0] // 'a'
```

上面代码分别用数值和字符串作为键名，结果都能读取数组。原因是数值键名被自动转为了字符串。

注意，这点在赋值时也成立。如果一个值总是先转成字符串，再进行赋值。

```
var a = [];

a[1.00] = 6;
a[1] // 6
```

上面代码中，由于 `1.00` 转成字符串是 `1` ，所以通过数字键 `1` 可以读取值。

上一章说过，对象有两种读取成员的方法：点结构（ `object.key` ）和方括号结构（ `object[key]` ）。但是，对于数值的键名，不能使用点结构。

```
var arr = [1, 2, 3];
arr.0 // SyntaxError
```

上面代码中，`arr.0` 的写法不合法，因为单独的数值不能作为标识符（identifier）。所以，数组成员只能用方括号 `arr[0]` 表示（方括号是运算符，可以接受数值）。

length 属性

数组的 `length` 属性，返回数组的成员数量。

```
['a', 'b', 'c'].length // 3
```

JavaScript 使用一个32位整数，保存数组的元素个数。这意味着，数组成员最多只有 4294967295 个（2^{32} - 1）个，也就是说 `length` 属性的最大值就是 4294967295。

只要是数组，就一定有 `length` 属性。该属性是一个动态的值，等于键名中的最大整数加上 `1`。

```
var arr = ['a', 'b'];
arr.length // 2

arr[2] = 'c';
arr.length // 3

arr[9] = 'd';
arr.length // 10

arr[1000] = 'e';
arr.length // 1001
```

上面代码表示，数组的数字键不需要连续，`length` 属性的值总是比最

大的那个整数键大 `1` 。另外，这也表明数组是一种动态的数据结构，可以随时增减数组的成员。

`length` 属性是可写的。如果人为设置一个小于当前成员个数的值，该数组的成员会自动减少到 `length` 设置的值。

```
var arr = [ 'a', 'b', 'c' ];
arr.length // 3

arr.length = 2;
arr // ["a", "b"]
```

上面代码表示，当数组的 `length` 属性设为2（即最大的整数键只能是1）那么整数键2（值为 `c` ）就已经不在数组中了，被自动删除了。

清空数组的一个有效方法，就是将 `length` 属性设为0。

```
var arr = [ 'a', 'b', 'c' ];

arr.length = 0;
arr // []
```

如果人为设置 `length` 大于当前元素个数，则数组的成员数量会增加到这个值，新增的位置都是空位。

```
var a = ['a'];

a.length = 3;
a[1] // undefined
```

上面代码表示，当 `length` 属性设为大于数组个数时，读取新增的位置都会返回 `undefined` 。

如果人为设置 `length` 为不合法的值，JavaScript 会报错。

```
// 设置负值
[].length = -1
// RangeError: Invalid array length

// 数组元素个数大于等于2的32次方
[].length = Math.pow(2, 32)
// RangeError: Invalid array length

// 设置字符串
[].length = 'abc'
// RangeError: Invalid array length
```

值得注意的是，由于数组本质上是一种对象，所以可以为数组添加属性，但是这不影响 `length` 属性的值。

```
var a = [];

a['p'] = 'abc';
a.length // 0

a[2.1] = 'abc';
a.length // 0
```

上面代码将数组的键分别设为字符串和小数，结果都不影响 `length` 属性。因为， `length` 属性的值就是等于最大的数字键加1，而这个数组没有整数键，所以 `length` 属性保持为 `0` 。

如果数组的键名是添加超出范围的数值，该键名会自动转为字符串。

```
var arr = [];
arr[-1] = 'a';
arr[Math.pow(2, 32)] = 'b';

arr.length // 0
arr[-1] // "a"
arr[4294967296] // "b"
```

上面代码中，我们为数组 `arr` 添加了两个不合法的数字键，结果 `length` 属性没有发生变化。这些数字键都变成了字符串键名。最后两行之所以会取到值，是因为取键值时，数字键名会默认转为字符串。

in 运算符

检查某个键名是否存在的运算符 `in` ，适用于对象，也适用于数组。

```
var arr = [ 'a', 'b', 'c' ];
2 in arr  // true
'2' in arr // true
4 in arr // false
```

上面代码表明，数组存在键名为 `2` 的键。由于键名都是字符串，所以数值 `2` 会自动转成字符串。

注意，如果数组的某个位置是空位， `in` 运算符返回 `false` 。

```
var arr = [];
arr[100] = 'a';

100 in arr // true
1 in arr // false
```

上面代码中，数组 `arr` 只有一个成员 `arr[100]` ，其他位置的键名都会返回 `false` 。

for…in 循环和数组的遍历

`for...in` 循环不仅可以遍历对象，也可以遍历数组，毕竟数组只是一种特殊对象。

```
var a = [1, 2, 3];
```

```
2.
3.  for (var i in a) {
4.    console.log(a[i]);
5.  }
6.  // 1
7.  // 2
8.  // 3
```

但是，`for...in` 不仅会遍历数组所有的数字键，还会遍历非数字键。

```
1.  var a = [1, 2, 3];
2.  a.foo = true;
3.
4.  for (var key in a) {
5.    console.log(key);
6.  }
7.  // 0
8.  // 1
9.  // 2
10. // foo
```

上面代码在遍历数组时，也遍历到了非整数键 `foo` 。所以，不推荐使用 `for...in` 遍历数组。

数组的遍历可以考虑使用 `for` 循环或 `while` 循环。

```
1.  var a = [1, 2, 3];
2.
3.  // for循环
4.  for(var i = 0; i < a.length; i++) {
5.    console.log(a[i]);
6.  }
7.
8.  // while循环
9.  var i = 0;
10. while (i < a.length) {
11.   console.log(a[i]);
```

```
  i++;
}

var l = a.length;
while (l--) {
  console.log(a[l]);
}
```

上面代码是三种遍历数组的写法。最后一种写法是逆向遍历，即从最后一个元素向第一个元素遍历。

数组的 `forEach` 方法，也可以用来遍历数组，详见《标准库》的 Array 对象一章。

```
var colors = ['red', 'green', 'blue'];
colors.forEach(function (color) {
  console.log(color);
});
// red
// green
// blue
```

数组的空位

当数组的某个位置是空元素，即两个逗号之间没有任何值，我们称该数组存在空位（hole）。

```
var a = [1, , 1];
a.length // 3
```

上面代码表明，数组的空位不影响 `length` 属性。

需要注意的是，如果最后一个元素后面有逗号，并不会产生空位。也就是说，有没有这个逗号，结果都是一样的。

```
var a = [1, 2, 3,];

a.length // 3
a // [1, 2, 3]
```

上面代码中，数组最后一个成员后面有一个逗号，这不影响 `length` 属性的值，与没有这个逗号时效果一样。

数组的空位是可以读取的，返回 `undefined` 。

```
var a = [, , ,];
a[1] // undefined
```

使用 `delete` 命令删除一个数组成员，会形成空位，并且不会影响 `length` 属性。

```
var a = [1, 2, 3];
delete a[1];

a[1] // undefined
a.length // 3
```

上面代码用 `delete` 命令删除了数组的第二个元素，这个位置就形成了空位，但是对 `length` 属性没有影响。也就是说， `length` 属性不过滤空位。所以，使用 `length` 属性进行数组遍历，一定要非常小心。

数组的某个位置是空位，与某个位置是 `undefined` ，是不一样的。如果是空位，使用数组的 `forEach` 方法、 `for...in` 结构、以及 `Object.keys` 方法进行遍历，空位都会被跳过。

```
var a = [, , ,];

a.forEach(function (x, i) {
  console.log(i + '. ' + x);
})
```

```
// 不产生任何输出

for (var i in a) {
  console.log(i);
}
// 不产生任何输出

Object.keys(a)
// []
```

如果某个位置是 `undefined`，遍历的时候就不会被跳过。

```
var a = [undefined, undefined, undefined];

a.forEach(function (x, i) {
  console.log(i + '. ' + x);
});
// 0. undefined
// 1. undefined
// 2. undefined

for (var i in a) {
  console.log(i);
}
// 0
// 1
// 2

Object.keys(a)
// ['0', '1', '2']
```

这就是说，空位就是数组没有这个元素，所以不会被遍历到，而 `undefined` 则表示数组有这个元素，值是 `undefined`，所以遍历不会跳过。

类似数组的对象

如果一个对象的所有键名都是正整数或零，并且有 `length` 属性，那么这个对象就很像数组，语法上称为“类似数组的对象”（array-like object）。

```
var obj = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};

obj[0] // 'a'
obj[1] // 'b'
obj.length // 3
obj.push('d') // TypeError: obj.push is not a function
```

上面代码中，对象 `obj` 就是一个类似数组的对象。但是，“类似数组的对象”并不是数组，因为它们不具备数组特有的方法。对象 `obj` 没有数组的 `push` 方法，使用该方法就会报错。

“类似数组的对象”的根本特征，就是具有 `length` 属性。只要有 `length` 属性，就可以认为这个对象类似于数组。但是有一个问题，这种 `length` 属性不是动态值，不会随着成员的变化而变化。

```
var obj = {
  length: 0
};
obj[3] = 'd';
obj.length // 0
```

上面代码为对象 `obj` 添加了一个数字键，但是 `length` 属性没变。这就说明了 `obj` 不是数组。

典型的“类似数组的对象”是函数的 `arguments` 对象，以及大多数 DOM

元素集，还有字符串。

```
// arguments对象
function args() { return arguments }
var arrayLike = args('a', 'b');

arrayLike[0] // 'a'
arrayLike.length // 2
arrayLike instanceof Array // false

// DOM元素集
var elts = document.getElementsByTagName('h3');
elts.length // 3
elts instanceof Array // false

// 字符串
'abc'[1] // 'b'
'abc'.length // 3
'abc' instanceof Array // false
```

上面代码包含三个例子，它们都不是数组（`instanceof`运算符返回`false`），但是看上去都非常像数组。

数组的`slice`方法可以将“类似数组的对象”变成真正的数组。

```
var arr = Array.prototype.slice.call(arrayLike);
```

除了转为真正的数组，“类似数组的对象”还有一个办法可以使用数组的方法，就是通过`call()`把数组的方法放到对象上面。

```
function print(value, index) {
  console.log(index + ' : ' + value);
}

Array.prototype.forEach.call(arrayLike, print);
```

上面代码中，`arrayLike`代表一个类似数组的对象，本来是不可以使用数组的`forEach()`方法的，但是通过`call()`，可以把`forEach()`嫁接到`arrayLike`上面调用。

下面的例子就是通过这种方法，在`arguments`对象上面调用`forEach`方法。

```
// forEach 方法
function logArgs() {
  Array.prototype.forEach.call(arguments, function (elem, i) {
    console.log(i + '. ' + elem);
  });
}

// 等同于 for 循环
function logArgs() {
  for (var i = 0; i < arguments.length; i++) {
    console.log(i + '. ' + arguments[i]);
  }
}
```

字符串也是类似数组的对象，所以也可以用`Array.prototype.forEach.call`遍历。

```
Array.prototype.forEach.call('abc', function (chr) {
  console.log(chr);
});
// a
// b
// c
```

注意，这种方法比直接使用数组原生的`forEach`要慢，所以最好还是先将“类似数组的对象”转为真正的数组，然后再直接调用数组的`forEach`方法。

```
var arr = Array.prototype.slice.call('abc');
arr.forEach(function (chr) {
  console.log(chr);
});
// a
// b
// c
```

参考链接

- Axel Rauschmayer, Arrays in JavaScript
- Axel Rauschmayer, JavaScript: sparse arrays vs. dense arrays
- Felix Bohm, What They Didn’t Tell You About ES5's Array Extras
- Juriy Zaytsev, How ECMAScript 5 still does not allow to subclass an array

运算符

运算符

算术运算符

算术运算符

运算符是处理数据的基本方法，用来从现有的值得到新的值。

JavaScript 提供了多种运算符，覆盖了所有主要的运算。

概述

JavaScript 共提供10个算术运算符，用来完成基本的算术运算。

- 加法运算符：`x + y`
- 减法运算符：`x - y`
- 乘法运算符：`x * y`
- 除法运算符：`x / y`
- 指数运算符：`x ** y`
- 余数运算符：`x % y`
- 自增运算符：`++x` 或者 `x++`

- 自减运算符：`--x` 或者 `x--`
- 数值运算符：`+x`
- 负数值运算符：`-x`

减法、乘法、除法运算法比较单纯，就是执行相应的数学运算。下面介绍其他几个算术运算符，重点是加法运算符。

加法运算符

基本规则

加法运算符（`+`）是最常见的运算符，用来求两个数值的和。

```
1. 1 + 1 // 2
```

JavaScript 允许非数值的相加。

```
1. true + true // 2
2. 1 + true // 2
```

上面代码中，第一行是两个布尔值相加，第二行是数值与布尔值相加。这两种情况，布尔值都会自动转成数值，然后再相加。

比较特殊的是，如果是两个字符串相加，这时加法运算符会变成连接运算符，返回一个新的字符串，将两个原字符串连接在一起。

```
1. 'a' + 'bc' // "abc"
```

如果一个运算子是字符串，另一个运算子是非字符串，这时非字符串会转成字符串，再连接在一起。

```
1. 1 + 'a' // "1a"
2. false + 'a' // "falsea"
```

加法运算符是在运行时决定，到底是执行相加，还是执行连接。也就是说，运算子的不同，导致了不同的语法行为，这种现象称为“重载”（overload）。由于加法运算符存在重载，可能执行两种运算，使用的时候必须很小心。

```
'3' + 4 + 5 // "345"
3 + 4 + '5' // "75"
```

上面代码中，由于从左到右的运算次序，字符串的位置不同会导致不同的结果。

除了加法运算符，其他算术运算符（比如减法、除法和乘法）都不会发生重载。它们的规则是：所有运算子一律转为数值，再进行相应的数学运算。

```
1 - '2' // -1
1 * '2' // 2
1 / '2' // 0.5
```

上面代码中，减法、除法和乘法运算符，都是将字符串自动转为数值，然后再运算。

对象的相加

如果运算子是对象，必须先转成原始类型的值，然后再相加。

```
var obj = { p: 1 };
obj + 2 // "[object Object]2"
```

上面代码中，对象 `obj` 转成原始类型的值是 `[object Object]` ，再加 `2` 就得到了上面的结果。

对象转成原始类型的值，规则如下。

首先，自动调用对象的 `valueOf` 方法。

```
var obj = { p: 1 };
obj.valueOf() // { p: 1 }
```

一般来说，对象的 `valueOf` 方法总是返回对象自身，这时再自动调用对象的 `toString` 方法，将其转为字符串。

```
var obj = { p: 1 };
obj.valueOf().toString() // "[object Object]"
```

对象的 `toString` 方法默认返回 `[object Object]` ，所以就得到了最前面那个例子的结果。

知道了这个规则以后，就可以自己定义 `valueOf` 方法或 `toString` 方法，得到想要的结果。

```
var obj = {
  valueOf: function () {
    return 1;
  }
};

obj + 2 // 3
```

上面代码中，我们定义 `obj` 对象的 `valueOf` 方法返回 `1` ，于是 `obj + 2` 就得到了 `3` 。这个例子中，由于 `valueOf` 方法直接返回一个原始类型的值，所以不再调用 `toString` 方法。

下面是自定义 `toString` 方法的例子。

```
var obj = {
  toString: function () {
```

```
    return 'hello';
  }
};

obj + 2 // "hello2"
```

上面代码中，对象 `obj` 的 `toString` 方法返回字符串 `hello` 。前面说过，只要有一个运算子是字符串，加法运算符就变成连接运算符，返回连接后的字符串。

这里有一个特例，如果运算子是一个 `Date` 对象的实例，那么会优先执行 `toString` 方法。

```
var obj = new Date();
obj.valueOf = function () { return 1 };
obj.toString = function () { return 'hello' };

obj + 2 // "hello2"
```

上面代码中，对象 `obj` 是一个 `Date` 对象的实例，并且自定义了 `valueOf` 方法和 `toString` 方法，结果 `toString` 方法优先执行。

余数运算符

余数运算符（ `%` ）返回前一个运算子被后一个运算子除，所得的余数。

```
12 % 5 // 2
```

需要注意的是，运算结果的正负号由第一个运算子的正负号决定。

```
-1 % 2 // -1
1 % -2 // 1
```

所以，为了得到负数的正确余数值，可以先使用绝对值函数。

```
// 错误的写法
function isOdd(n) {
  return n % 2 === 1;
}
isOdd(-5) // false
isOdd(-4) // false

// 正确的写法
function isOdd(n) {
  return Math.abs(n % 2) === 1;
}
isOdd(-5) // true
isOdd(-4) // false
```

余数运算符还可以用于浮点数的运算。但是，由于浮点数不是精确的值，无法得到完全准确的结果。

```
6.5 % 2.1
// 0.19999999999999973
```

自增和自减运算符

自增和自减运算符，是一元运算符，只需要一个运算子。它们的作用是将运算子首先转为数值，然后加上1或者减去1。它们会修改原始变量。

```
var x = 1;
++x // 2
x // 2

--x // 1
x // 1
```

上面代码的变量 `x` 自增后，返回 `2` ，再进行自减，返回 `1` 。这两

种情况都会使得，原始变量 `x` 的值发生改变。

运算之后，变量的值发生变化，这种效应叫做运算的副作用（side effect）。自增和自减运算符是仅有的两个具有副作用的运算符，其他运算符都不会改变变量的值。

自增和自减运算符有一个需要注意的地方，就是放在变量之后，会先返回变量操作前的值，再进行自增/自减操作；放在变量之前，会先进行自增/自减操作，再返回变量操作后的值。

```
1. var x = 1;
2. var y = 1;
3.
4. x++ // 1
5. ++y // 2
```

上面代码中，`x` 是先返回当前值，然后自增，所以得到 `1`；`y` 是先自增，然后返回新的值，所以得到 `2`。

数值运算符，负数值运算符

数值运算符（`+`）同样使用加号，但它是一元运算符（只需要一个操作数），而加法运算符是二元运算符（需要两个操作数）。

数值运算符的作用在于可以将任何值转为数值（与 `Number` 函数的作用相同）。

```
1. +true // 1
2. +[] // 0
3. +{} // NaN
```

上面代码表示，非数值经过数值运算符以后，都变成了数值（最后一行 `NaN` 也是数值）。具体的类型转换规则，参见《数据类型转换》一

章。

负数值运算符（ `-` ），也同样具有将一个值转为数值的功能，只不过得到的值正负相反。连用两个负数值运算符，等同于数值运算符。

```
var x = 1;
-x // -1
-(-x) // 1
```

上面代码最后一行的圆括号不可少，否则会变成自减运算符。

数值运算符号和负数值运算符，都会返回一个新的值，而不会改变原始变量的值。

指数运算符

指数运算符（ `**` ）完成指数运算，前一个运算子是底数，后一个运算子是指数。

```
2 ** 4 // 16
```

赋值运算符

赋值运算符（Assignment Operators）用于给变量赋值。

最常见的赋值运算符，当然就是等号（ `=` ）。

```
// 将 1 赋值给变量 x
var x = 1;

// 将变量 y 的值赋值给变量 x
var x = y;
```

赋值运算符还可以与其他运算符结合，形成变体。下面是与算术运算符

的结合。

```
// 等同于 x = x + y
x += y

// 等同于 x = x - y
x -= y

// 等同于 x = x * y
x *= y

// 等同于 x = x / y
x /= y

// 等同于 x = x % y
x %= y

// 等同于 x = x ** y
x **= y
```

下面是与位运算符的结合（关于位运算符，请见后文的介绍）。

```
// 等同于 x = x >> y
x >>= y

// 等同于 x = x << y
x <<= y

// 等同于 x = x >>> y
x >>>= y

// 等同于 x = x & y
x &= y

// 等同于 x = x | y
x |= y

```

```
// 等同于 x = x ^ y
x ^= y
```

这些复合的赋值运算符，都是先进行指定运算，然后将得到值返回给左边的变量。

比较运算符

比较运算符

概述

比较运算符用于比较两个值的大小，然后返回一个布尔值，表示是否满足指定的条件。

```
1.  2 > 1 // true
```

上面代码比较 `2` 是否大于 `1` ，返回 `true` 。

> 注意，比较运算符可以比较各种类型的值，不仅仅是数值。

JavaScript 一共提供了8个比较运算符。

- `>` 大于运算符
- `<` 小于运算符
- `<=` 小于或等于运算符
- `>=` 大于或等于运算符
- `==` 相等运算符

- `===` 严格相等运算符
- `!=` 不相等运算符
- `!==` 严格不相等运算符

这八个比较运算符分成两类：相等比较和非相等比较。两者的规则是不一样的，对于非相等的比较，算法是先看两个运算子是否都是字符串，如果是的，就按照字典顺序比较（实际上是比较 Unicode 码点）；否则，将两个运算子都转成数值，再比较数值的大小。

非相等运算符：字符串的比较

字符串按照字典顺序进行比较。

```
1. 'cat' > 'dog' // false
2. 'cat' > 'catalog' // false
```

JavaScript 引擎内部首先比较首字符的 Unicode 码点。如果相等，再比较第二个字符的 Unicode 码点，以此类推。

```
1. 'cat' > 'Cat' // true'
```

上面代码中，小写的 `c` 的 Unicode 码点（ `99` ）大于大写的 `C` 的 Unicode 码点（ `67` ），所以返回 `true` 。

由于所有字符都有 Unicode 码点，因此汉字也可以比较。

```
1. '大' > '小' // false
```

上面代码中，“大”的 Unicode 码点是22823，“小”是23567，因此返回 `false` 。

非相等运算符：非字符串的比较

如果两个运算子都不是字符串，分成以下三种情况。

（1）原始类型值

如果两个运算子都是原始类型的值，则是先转成数值再比较。

```
5 > '4' // true
// 等同于 5 > Number('4')
// 即 5 > 4

true > false // true
// 等同于 Number(true) > Number(false)
// 即 1 > 0

2 > true // true
// 等同于 2 > Number(true)
// 即 2 > 1
```

上面代码中，字符串和布尔值都会先转成数值，再进行比较。

任何值（包括 `NaN` 本身）与 `NaN` 比较，返回的都是 `false` 。

```
1 > NaN // false
1 <= NaN // false
'1' > NaN // false
'1' <= NaN // false
NaN > NaN // false
NaN <= NaN // false
```

（2）对象

如果运算子是对象，会转为原始类型的值，再进行比较。

对象转换成原始类型的值，算法是先调用 `valueOf` 方法；如果返回的还是对象，再接着调用 `toString` 方法，详细解释参见《数据类型的转换》一章。

```
var x = [2];
x > '11' // true
// 等同于 [2].valueOf().toString() > '11'
// 即 '2' > '11'

x.valueOf = function () { return '1' };
x > '11' // false
// 等同于 [2].valueOf() > '11'
// 即 '1' > '11'
```

两个对象之间的比较也是如此。

```
[2] > [1] // true
// 等同于 [2].valueOf().toString() > [1].valueOf().toString()
// 即 '2' > '1'

[2] > [11] // true
// 等同于 [2].valueOf().toString() > [11].valueOf().toString()
// 即 '2' > '11'

{ x: 2 } >= { x: 1 } // true
// 等同于 { x: 2 }.valueOf().toString() >= { x: 1 }.valueOf().toString()
// 即 '[object Object]' >= '[object Object]'
```

严格相等运算符

JavaScript 提供两种相等运算符：`==` 和 `===`。

简单说，它们的区别是相等运算符（`==`）比较两个值是否相等，严格相等运算符（`===`）比较它们是否为“同一个值”。如果两个值不是同一类型，严格相等运算符（`===`）直接返回 `false`，而相等运算符（`==`）会将它们转换成同一个类型，再用严格相等运算符进行比较。

本节介绍严格相等运算符的算法。

（1）不同类型的值

如果两个值的类型不同，直接返回 `false` 。

```
1. 1 === "1" // false
2. true === "true" // false
```

上面代码比较数值的 `1` 与字符串的“1”、布尔值的 `true` 与字符串 `"true"` ，因为类型不同，结果都是 `false` 。

（2）同一类的原始类型值

同一类型的原始类型的值（数值、字符串、布尔值）比较时，值相同就返回 `true` ，值不同就返回 `false` 。

```
1. 1 === 0x1 // true
```

上面代码比较十进制的 `1` 与十六进制的 `1` ，因为类型和值都相同，返回 `true` 。

需要注意的是， `NaN` 与任何值都不相等（包括自身）。另外，正 `0` 等于负 `0` 。

```
1. NaN === NaN  // false
2. +0 === -0 // true
```

（3）复合类型值

两个复合类型（对象、数组、函数）的数据比较时，不是比较它们的值是否相等，而是比较它们是否指向同一个地址。

```
1. {} === {} // false
2. [] === [] // false
3. (function () {} === function () {}) // false
```

上面代码分别比较两个空对象、两个空数组、两个空函数，结果都是不相等。原因是对于复合类型的值，严格相等运算比较的是，它们是否引用同一个内存地址，而运算符两边的空对象、空数组、空函数的值，都存放在不同的内存地址，结果当然是 `false` 。

如果两个变量引用同一个对象，则它们相等。

```
var v1 = {};
var v2 = v1;
v1 === v2 // true
```

注意，对于两个对象的比较，严格相等运算符比较的是地址，而大于或小于运算符比较的是值。

```
var obj1 = {};
var obj2 = {};

obj1 > obj2 // false
obj1 < obj2 // false
obj1 === obj2 // false
```

上面的三个比较，前两个比较的是值，最后一个比较的是地址，所以都返回 `false` 。

（4）undefined 和 null

`undefined` 和 `null` 与自身严格相等。

```
undefined === undefined // true
null === null // true
```

由于变量声明后默认值是 `undefined` ，因此两个只声明未赋值的变量是相等的。

```
var v1;
var v2;
v1 === v2 // true
```

严格不相等运算符

严格相等运算符有一个对应的“严格不相等运算符”（ `!==` ），它的算法就是先求严格相等运算符的结果，然后返回相反值。

```
1 !== '1' // true
// 等同于
!(1 === '1')
```

上面代码中，感叹号 `!` 是求出后面表达式的相反值。

相等运算符

相等运算符用来比较相同类型的数据时，与严格相等运算符完全一样。

```
1 == 1.0
// 等同于
1 === 1.0
```

比较不同类型的数据时，相等运算符会先将数据进行类型转换，然后再用严格相等运算符比较。类型转换规则如下。

（1）原始类型值

原始类型的值会转换成数值再进行比较。

```
1 == true // true
// 等同于 1 === Number(true)

0 == false // true
```

```
// 等同于 0 === Number(false)

2 == true // false
// 等同于 2 === Number(true)

2 == false // false
// 等同于 2 === Number(false)

'true' == true // false
// 等同于 Number('true') === Number(true)
// 等同于 NaN === 1

'' == 0 // true
// 等同于 Number('') === 0
// 等同于 0 === 0

'' == false  // true
// 等同于 Number('') === Number(false)
// 等同于 0 === 0

'1' == true  // true
// 等同于 Number('1') === Number(true)
// 等同于 1 === 1

'\n  123  \t' == 123 // true
// 因为字符串转为数字时，省略前置和后置的空格
```

上面代码将字符串和布尔值都转为数值，然后再进行比较。具体的字符串与布尔值的类型转换规则，参见《数据类型转换》一章。

（2）对象与原始类型值比较

对象（这里指广义的对象，包括数组和函数）与原始类型的值比较时，对象转换成原始类型的值，再进行比较。

```
[1] == 1 // true
// 等同于 Number([1]) == 1
```

```
3.
4. [1] == '1' // true
5. // 等同于 String([1]) == Number('1')
6.
7. [1] == true // true
8. // 等同于 Number([1]) == Number(true)
```

上面代码中，数组 `[1]` 与数值进行比较，会先转成数值，再进行比较；与字符串进行比较，会先转成字符串，再进行比较；与布尔值进行比较，两个运算子都会先转成数值，然后再进行比较。

（3）undefined 和 null

`undefined` 和 `null` 与其他类型的值比较时，结果都为 `false`，它们互相比较时结果为 `true`。

```
1. false == null // false
2. false == undefined // false
3.
4. 0 == null // false
5. 0 == undefined // false
6.
7. undefined == null // true
```

（4）相等运算符的缺点

相等运算符隐藏的类型转换，会带来一些违反直觉的结果。

```
1. 0 == ''             // true
2. 0 == '0'            // true
3.
4. 2 == true           // false
5. 2 == false          // false
6.
7. false == 'false'    // false
8. false == '0'        // true
```

```
9.
10. false == undefined  // false
11. false == null       // false
12. null == undefined   // true
13.
14. ' \t\r\n ' == 0     // true
```

上面这些表达式都不同于直觉，很容易出错。因此建议不要使用相等运算符（ `==` ），最好只使用严格相等运算符（ `===` ）。

不相等运算符

相等运算符有一个对应的“不相等运算符”（ `!=` ），它的算法就是先求相等运算符的结果，然后返回相反值。

```
1. 1 != '1' // false
2.
3. // 等同于
4. !(1 == '1')
```

布尔运算符

布尔运算符

概述

布尔运算符用于将表达式转为布尔值，一共包含四个运算符。

- 取反运算符：`!`
- 且运算符：`&&`
- 或运算符：`||`
- 三元运算符：`?:`

取反运算符（！）

取反运算符是一个感叹号，用于将布尔值变为相反值，即 `true` 变成 `false`，`false` 变成 `true`。

```
!true // false
!false // true
```

对于非布尔值，取反运算符会将其转为布尔值。可以这样记忆，以下六个值取反后为 `true`，其他值都为 `false`。

- `undefined`
- `null`
- `false`
- `0`
- `NaN`
- 空字符串（`''`）

```
!undefined // true
!null // true
!0 // true
!NaN // true
!"" // true

!54 // false
!'hello' // false
![] // false
!{} // false
```

上面代码中，不管什么类型的值，经过取反运算后，都变成了布尔值。

如果对一个值连续做两次取反运算，等于将其转为对应的布尔值，与 `Boolean` 函数的作用相同。这是一种常用的类型转换的写法。

```
!!x
// 等同于
Boolean(x)
```

上面代码中，不管 `x` 是什么类型的值，经过两次取反运算后，变成了与 `Boolean` 函数结果相同的布尔值。所以，两次取反就是将一个值转为布尔值的简便写法。

且运算符（&&）

且运算符（ `&&` ）往往用于多个表达式的求值。

它的运算规则是：如果第一个运算子的布尔值为 `true` ，则返回第二个运算子的值（注意是值，不是布尔值）；如果第一个运算子的布尔值为 `false` ，则直接返回第一个运算子的值，且不再对第二个运算子求值。

```
't' && '' // ""
't' && 'f' // "f"
't' && (1 + 2) // 3
'' && 'f' // ""
'' && '' // ""

var x = 1;
(1 - 1) && ( x += 1) // 0
x // 1
```

上面代码的最后一个例子，由于且运算符的第一个运算子的布尔值为 `false` ，则直接返回它的值 `0` ，而不再对第二个运算子求值，所以变量 `x` 的值没变。

这种跳过第二个运算子的机制，被称为“短路”。有些程序员喜欢用它取代 `if` 结构，比如下面是一段 `if` 结构的代码，就可以用且运算符改写。

```
if (i) {
  doSomething();
}

// 等价于

i && doSomething();
```

上面代码的两种写法是等价的，但是后一种不容易看出目的，也不容易

除错，建议谨慎使用。

且运算符可以多个连用，这时返回第一个布尔值为 `false` 的表达式的值。

```
true && 'foo' && '' && 4 && 'foo' && true
// ''
```

上面代码中，第一个布尔值为 `false` 的表达式为第三个表达式，所以得到一个空字符串。

或运算符（||）

或运算符（ `||` ）也用于多个表达式的求值。它的运算规则是：如果第一个运算子的布尔值为 `true` ，则返回第一个运算子的值，且不再对第二个运算子求值；如果第一个运算子的布尔值为 `false` ，则返回第二个运算子的值。

```
't' || '' // "t"
't' || 'f' // "t"
'' || 'f' // "f"
'' || '' // ""
```

短路规则对这个运算符也适用。

```
var x = 1;
true || (x = 2) // true
x // 1
```

上面代码中，且运算符的第一个运算子为 `true` ，所以直接返回 `true` ，不再运行第二个运算子。所以， `x` 的值没有改变。这种只通过第一个表达式的值，控制是否运行第二个表达式的机制，就称为“短路”（short-cut）。

或运算符可以多个连用，这时返回第一个布尔值为 `true` 的表达式的值。

```
false || 0 || '' || 4 || 'foo' || true
// 4
```

上面代码中第一个布尔值为 `true` 的表达式是第四个表达式，所以得到数值4。

或运算符常用于为一个变量设置默认值。

```
function saveText(text) {
  text = text || '';
  // ...
}

// 或者写成
saveText(this.text || '')
```

上面代码表示，如果函数调用时，没有提供参数，则该参数默认设置为空字符串。

三元条件运算符（?:）

三元条件运算符由问号（?）和冒号（:）组成，分隔三个表达式。它是 JavaScript 语言唯一一个需要三个运算子的运算符。如果第一个表达式的布尔值为 `true`，则返回第二个表达式的值，否则返回第三个表达式的值。

```
't' ? 'hello' : 'world' // "hello"
0 ? 'hello' : 'world' // "world"
```

上面代码的 `t` 和 `0` 的布尔值分别为 `true` 和 `false`，所以分别返回

第二个和第三个表达式的值。

通常来说，三元条件表达式与 `if...else` 语句具有同样表达效果，前者可以表达的，后者也能表达。但是两者具有一个重大差别， `if...else` 是语句，没有返回值；三元条件表达式是表达式，具有返回值。所以，在需要返回值的场合，只能使用三元条件表达式，而不能使用 `if..else` 。

```
1.  console.log(true ? 'T' : 'F');
```

上面代码中， `console.log` 方法的参数必须是一个表达式，这时就只能使用三元条件表达式。如果要用 `if...else` 语句，就必须改变整个代码写法了。

二进制位运算符

二进制位运算符

概述

二进制位运算符用于直接对二进制位进行计算，一共有7个。

- 二进制或运算符（or）：符号为`|`，表示若两个二进制位都为`0`，则结果为`0`，否则为`1`。
- 二进制与运算符（and）：符号为`&`，表示若两个二进制位都为1，则结果为1，否则为0。
- 二进制否运算符（not）：符号为`~`，表示对一个二进制位取反。
- 异或运算符（xor）：符号为`^`，表示若两个二进制位不相同，则结果为1，否则为0。
- 左移运算符（left shift）：符号为`<<`，详见下文解释。
- 右移运算符（right shift）：符号为`>>`，详见下文解释。

- 带符号位的右移运算符（zero filled right shift）：符号为`>>>`，详见下文解释。

这些位运算符直接处理每一个比特位（bit），所以是非常底层的运算，好处是速度极快，缺点是很不直观，许多场合不能使用它们，否则会使代码难以理解和查错。

有一点需要特别注意，位运算符只对整数起作用，如果一个运算子不是整数，会自动转为整数后再执行。另外，虽然在 JavaScript 内部，数值都是以64位浮点数的形式储存，但是做位运算的时候，是以32位带符号的整数进行运算的，并且返回值也是一个32位带符号的整数。

```
i = i | 0;
```

上面这行代码的意思，就是将`i`（不管是整数或小数）转为32位整数。

利用这个特性，可以写出一个函数，将任意数值转为32位整数。

```
function toInt32(x) {
  return x | 0;
}
```

上面这个函数将任意值与`0`进行一次或运算，这个位运算会自动将一个值转为32位整数。下面是这个函数的用法。

```
toInt32(1.001) // 1
toInt32(1.999) // 1
toInt32(1) // 1
toInt32(-1) // -1
toInt32(Math.pow(2, 32) + 1) // 1
toInt32(Math.pow(2, 32) - 1) // -1
```

上面代码中，`toInt32`可以将小数转为整数。对于一般的整数，返回值

不会有任何变化。对于大于2的32次方的整数，大于32位的数位都会被舍去。

二进制或运算符

二进制或运算符（ `|` ）逐位比较两个运算子，两个二进制位之中只要有一个为 `1` ，就返回 `1` ，否则返回 `0` 。

```
0 | 3 // 3
```

上面代码中， `0` 和 `3` 的二进制形式分别是 `00` 和 `11` ，所以进行二进制或运算会得到 `11` （即 `3` ）。

位运算只对整数有效，遇到小数时，会将小数部分舍去，只保留整数部分。所以，将一个小数与 `0` 进行二进制或运算，等同于对该数去除小数部分，即取整数位。

```
2.9 | 0 // 2
-2.9 | 0 // -2
```

需要注意的是，这种取整方法不适用超过32位整数最大值 `2147483647` 的数。

```
2147483649.4 | 0;
// -2147483647
```

二进制与运算符

二进制与运算符（ `&` ）的规则是逐位比较两个运算子，两个二进制位之中只要有一个位为 `0` ，就返回 `0` ，否则返回 `1` 。

```
0 & 3 // 0
```

上面代码中，0（二进制 `00` ）和3（二进制 `11` ）进行二进制与运算会得到 `00` （即 `0` ）。

二进制否运算符

二进制否运算符（ `~` ）将每个二进制位都变为相反值（ `0` 变为 `1` ， `1` 变为 `0` ）。它的返回结果有时比较难理解，因为涉及到计算机内部的数值表示机制。

```
1. ~ 3 // -4
```

上面表达式对 `3` 进行二进制否运算，得到 `-4` 。之所以会有这样的结果，是因为位运算时，JavaScirpt 内部将所有的运算子都转为32位的二进制整数再进行运算。

`3` 的32位整数形式是 `00000000000000000000000000000011` ，二进制否运算以后得到 `11111111111111111111111111111100` 。由于第一位（符号位）是1，所以这个数是一个负数。JavaScript 内部采用补码形式表示负数，即需要将这个数减去1，再取一次反，然后加上负号，才能得到这个负数对应的10进制值。这个数减去1等于 `11111111111111111111111111111011` ，再取一次反得到 `00000000000000000000000000000100` ，再加上负号就是 `-4` 。考虑到这样的过程比较麻烦，可以简单记忆成，一个数与自身的取反值相加，等于-1。

```
1. ~ -3 // 2
```

上面表达式可以这样算， `-3` 的取反值等于 `-1` 减去 `-3` ，结果为 `2` 。

对一个整数连续两次二进制否运算，得到它自身。

```
~~3 // 3
```

所有的位运算都只对整数有效。二进制否运算遇到小数时，也会将小数部分舍去，只保留整数部分。所以，对一个小数连续进行两次二进制否运算，能达到取整效果。

```
~~2.9 // 2
~~47.11 // 47
~~1.9999 // 1
~~3 // 3
```

使用二进制否运算取整，是所有取整方法中最快的一种。

对字符串进行二进制否运算，JavaScript 引擎会先调用 `Number` 函数，将字符串转为数值。

```
// 相当于~Number('011')
~'011'  // -12

// 相当于~Number('42 cats')
~'42 cats' // -1

// 相当于~Number('0xcafebabe')
~'0xcafebabe' // 889275713

// 相当于~Number('deadbeef')
~'deadbeef' // -1
```

`Number` 函数将字符串转为数值的规则，参见《数据的类型转换》一章。

对于其他类型的值，二进制否运算也是先用 `Number` 转为数值，然后再进行处理。

```
// 相当于 ~Number([])
```

```
~[] // -1

// 相当于 ~Number(NaN)
~NaN // -1

// 相当于 ~Number(null)
~null // -1
```

异或运算符

异或运算（ `^` ）在两个二进制位不同时返回 `1` ，相同时返回 `0` 。

```
0 ^ 3 // 3
```

上面表达式中， `0` （二进制 `00` ）与 `3` （二进制 `11` ）进行异或运算，它们每一个二进制位都不同，所以得到 `11` （即 `3` ）。

“异或运算”有一个特殊运用，连续对两个数 `a` 和 `b` 进行三次异或运算， `a^=b; b^=a; a^=b;` ，可以互换它们的值。这意味着，使用“异或运算”可以在不引入临时变量的前提下，互换两个变量的值。

```
var a = 10;
var b = 99;

a ^= b, b ^= a, a ^= b;

a // 99
b // 10
```

这是互换两个变量的值的最快方法。

异或运算也可以用来取整。

```
12.9 ^ 0 // 12
```

左移运算符

左移运算符（ `<<` ）表示将一个数的二进制值向左移动指定的位数，尾部补 `0` ，即乘以 `2` 的指定次方（最高位即符号位不参与移动）。

```
// 4 的二进制形式为100,
// 左移一位为1000（即十进制的8）
// 相当于乘以2的1次方
4 << 1
// 8

-4 << 1
// -8
```

上面代码中， `-4` 左移一位得到 `-8` ，是因为 `-4` 的二进制形式是 `11111111111111111111111111111100` ，左移一位后得到 `11111111111111111111111111111000` ，该数转为十进制（减去1后取反，再加上负号）即为 `-8` 。

如果左移0位，就相当于将该数值转为32位整数，等同于取整，对于正数和负数都有效。

```
13.5 << 0
// 13

-13.5 << 0
// -13
```

左移运算符用于二进制数值非常方便。

```
var color = {r: 186, g: 218, b: 85};

// RGB to HEX
// (1 << 24)的作用为保证结果是6位数
var rgb2hex = function(r, g, b) {
```

```
  return '#' + ((1 << 24) + (r << 16) + (g << 8) + b)
    .toString(16) // 先转成十六进制，然后返回字符串
    .substr(1);   // 去除字符串的最高位，返回后面六个字符串
}

rgb2hex(color.r, color.g, color.b)
// "#bada55"
```

上面代码使用左移运算符，将颜色的 RGB 值转为 HEX 值。

右移运算符

右移运算符（ `>>` ）表示将一个数的二进制值向右移动指定的位数，头部补 `0` ，即除以 `2` 的指定次方（最高位即符号位不参与移动）。

```
4 >> 1
// 2
/*
// 因为4的二进制形式为 00000000000000000000000000000100,
// 右移一位得到 00000000000000000000000000000010,
// 即为十进制的2
*/

-4 >> 1
// -2
/*
// 因为-4的二进制形式为 11111111111111111111111111111100,
// 右移一位，头部补1，得到 11111111111111111111111111111110,
// 即为十进制的-2
*/
```

右移运算可以模拟 2 的整除运算。

```
5 >> 1
// 2
// 相当于 5 / 2 = 2
```

```
4.
5.  21 >> 2
6.  // 5
7.  // 相当于 21 / 4 = 5
8.
9.  21 >> 3
10. // 2
11. // 相当于 21 / 8 = 2
12.
13. 21 >> 4
14. // 1
15. // 相当于 21 / 16 = 1
```

带符号位的右移运算符

带符号位的右移运算符（ `>>>` ）表示将一个数的二进制形式向右移动，包括符号位也参与移动，头部补 `0` 。所以，该运算总是得到正值。对于正数，该运算的结果与右移运算符（ `>>` ）完全一致，区别主要在于负数。

```
1.  4 >>> 1
2.  // 2
3.
4.  -4 >>> 1
5.  // 2147483646
6.  /*
7.  // 因为-4的二进制形式为11111111111111111111111111111100,
8.  // 带符号位的右移一位，得到01111111111111111111111111111110,
9.  // 即为十进制的2147483646。
10. */
```

这个运算实际上将一个值转为32位无符号整数。

查看一个负整数在计算机内部的储存形式，最快的方法就是使用这个运算符。

```
-1 >>> 0 // 4294967295
```

上面代码表示，`-1` 作为32位整数时，内部的储存形式使用无符号整数格式解读，值为 4294967295（即 `(2^32)-1`，等于 `11111111111111111111111111111111`）。

开关作用

位运算符可以用作设置对象属性的开关。

假定某个对象有四个开关，每个开关都是一个变量。那么，可以设置一个四位的二进制数，它的每个位对应一个开关。

```
var FLAG_A = 1; // 0001
var FLAG_B = 2; // 0010
var FLAG_C = 4; // 0100
var FLAG_D = 8; // 1000
```

上面代码设置 A、B、C、D 四个开关，每个开关分别占有一个二进制位。

然后，就可以用二进制与运算检验，当前设置是否打开了指定开关。

```
var flags = 5; // 二进制的0101

if (flags & FLAG_C) {
  // ...
}
// 0101 & 0100 => 0100 => true
```

上面代码检验是否打开了开关 `C`。如果打开，会返回 `true`，否则返回 `false`。

现在假设需要打开 `A`、`B`、`D` 三个开关，我们可以构造一个掩码

变量。

```
var mask = FLAG_A | FLAG_B | FLAG_D;
// 0001 | 0010 | 1000 => 1011
```

上面代码对 `A` 、 `B` 、 `D` 三个变量进行二进制或运算，得到掩码值为二进制的 `1011` 。

有了掩码，二进制或运算可以确保打开指定的开关。

```
flags = flags | mask;
```

二进制与运算可以将当前设置中凡是与开关设置不一样的项，全部关闭。

```
flags = flags & mask;
```

异或运算可以切换（toggle）当前设置，即第一次执行可以得到当前设置的相反值，再执行一次又得到原来的值。

```
flags = flags ^ mask;
```

二进制否运算可以翻转当前设置，即原设置为 `0` ，运算后变为 `1` ；原设置为 `1` ，运算后变为 `0` 。

```
flags = ~flags;
```

参考链接

- Michal Budzynski, JavaScript: The less known parts. Bitwise Operators
- Axel Rauschmayer, Basic JavaScript for the

impatient programmer

- Mozilla Developer Network, Bitwise Operators

其他运算符，运算顺序

其他运算符，运算顺序

void 运算符

`void` 运算符的作用是执行一个表达式，然后不返回任何值，或者说返回 `undefined` 。

```
1. void 0 // undefined
2. void(0) // undefined
```

上面是 `void` 运算符的两种写法，都正确。建议采用后一种形式，即总是使用圆括号。因为 `void` 运算符的优先性很高，如果不使用括号，容易造成错误的结果。比如， `void 4 + 7` 实际上等同于 `(void 4) + 7` 。

下面是 `void` 运算符的一个例子。

```
1. var x = 3;
2. void (x = 5) //undefined
3. x // 5
```

这个运算符的主要用途是浏览器的书签工具（bookmarklet），以及

在超级链接中插入代码防止网页跳转。

请看下面的代码。

```
<script>
function f() {
  console.log('Hello World');
}
</script>
<a href="http://example.com" onclick="f(); return false;">点击</a>
```

上面代码中，点击链接后，会先执行 `onclick` 的代码，由于 `onclick` 返回 `false` ，所以浏览器不会跳转到 example.com。

`void` 运算符可以取代上面的写法。

```
<a href="javascript: void(f())">文字</a>
```

下面是一个更实际的例子，用户点击链接提交表单，但是不产生页面跳转。

```
<a href="javascript: void(document.form.submit())">
  提交
</a>
```

逗号运算符

逗号运算符用于对两个表达式求值，并返回后一个表达式的值。

```
'a', 'b' // "b"

var x = 0;
var y = (x++, 10);
x // 1
y // 10
```

上面代码中，逗号运算符返回后一个表达式的值。

逗号运算符的一个用途是，在返回一个值之前，进行一些辅助操作。

```
var value = (console.log('Hi!'), true);
// Hi!

value // true
```

上面代码中，先执行逗号之前的操作，然后返回逗号后面的值。

运算顺序

优先级

JavaScript 各种运算符的优先级别（Operator Precedence）是不一样的。优先级高的运算符先执行，优先级低的运算符后执行。

```
4 + 5 * 6 // 34
```

上面的代码中，乘法运算符（`*`）的优先性高于加法运算符（`+`），所以先执行乘法，再执行加法，相当于下面这样。

```
4 + (5 * 6) // 34
```

如果多个运算符混写在一起，常常会导致令人困惑的代码。

```
var x = 1;
var arr = [];

var y = arr.length <= 0 || arr[0] === undefined ? x : arr[0];
```

上面代码中，变量`y`的值就很难看出来，因为这个表达式涉及5个运

算符，到底谁的优先级最高，实在不容易记住。

根据语言规格，这五个运算符的优先级从高到低依次为：小于等于（ `<=` ）、严格相等（ `===` ）、或（ `||` ）、三元（ `?:` ）、等号（ `=` ）。因此上面的表达式，实际的运算顺序如下。

```
1.  var y = ((arr.length <= 0) || (arr[0] === undefined)) ? x : arr[0];
```

记住所有运算符的优先级，是非常难的，也是没有必要的。

圆括号的作用

圆括号（ `()` ）可以用来提高运算的优先级，因为它的优先级是最高的，即圆括号中的表达式会第一个运算。

```
1.  (4 + 5) * 6 // 54
```

上面代码中，由于使用了圆括号，加法会先于乘法执行。

运算符的优先级别十分繁杂，且都是硬性规定，因此建议总是使用圆括号，保证运算顺序清晰可读，这对代码的维护和除错至关重要。

顺便说一下，圆括号不是运算符，而是一种语法结构。它一共有两种用法：一种是把表达式放在圆括号之中，提升运算的优先级；另一种是跟在函数的后面，作用是调用函数。

注意，因为圆括号不是运算符，所以不具有求值作用，只改变运算的优先级。

```
1.  var x = 1;
2.  (x) = 2;
```

上面代码的第二行，如果圆括号具有求值作用，那么就会变成 `1 =`

`2`，这是会报错了。但是，上面的代码可以运行，这验证了圆括号只改变优先级，不会求值。

这也意味着，如果整个表达式都放在圆括号之中，那么不会有任何效果。

```
(exprssion)
// 等同于
expression
```

函数放在圆括号中，会返回函数本身。如果圆括号紧跟在函数的后面，就表示调用函数。

```
function f() {
  return 1;
}

(f) // function f(){return 1;}
f() // 1
```

上面代码中，函数放在圆括号之中会返回函数本身，圆括号跟在函数后面则是调用函数。

圆括号之中，只能放置表达式，如果将语句放在圆括号之中，就会报错。

```
(var a = 1)
// SyntaxError: Unexpected token var
```

左结合与右结合

对于优先级别相同的运算符，大多数情况，计算顺序总是从左到右，这叫做运算符的“左结合”（left-to-right associativity），即

从左边开始计算。

```
x + y + z
```

上面代码先计算最左边的 `x` 与 `y` 的和，然后再计算与 `z` 的和。

但是少数运算符的计算顺序是从右到左，即从右边开始计算，这叫做运算符的“右结合”（right-to-left associativity）。其中，最主要的是赋值运算符（ `=` ）和三元条件运算符（ `?:` ）。

```
w = x = y = z;
q = a ? b : c ? d : e ? f : g;
```

上面代码的运算结果，相当于下面的样子。

```
w = (x = (y = z));
q = a ? b : (c ? d : (e ? f : g));
```

上面的两行代码，各有三个等号运算符和三个三元运算符，都是先计算最右边的那个运算符。

语法专题

语法专题

数据类型的转换

数据类型的转换

概述

JavaScript 是一种动态类型语言，变量没有类型限制，可以随时赋予任意值。

```
var x = y ? 1 : 'a';
```

上面代码中，变量 `x` 到底是数值还是字符串，取决于另一个变量 `y` 的值。 `y` 为 `true` 时， `x` 是一个数值； `y` 为 `false` 时， `x` 是一个字符串。这意味着， `x` 的类型没法在编译阶段就知道，必须等到运行时才能知道。

虽然变量的数据类型是不确定的，但是各种运算符对数据类型是有要求的。如果运算符发现，运算子的类型与预期不符，就会自动转换类型。

比如，减法运算符预期左右两侧的运算子应该是数值，如果不是，就会自动将它们转为数值。

```
'4' - '3' // 1
```

上面代码中，虽然是两个字符串相减，但是依然会得到结果数值 `1`，原因就在于 JavaScript 将运算子自动转为了数值。

本章讲解数据类型自动转换的规则。在此之前，先讲解如何手动强制转换数据类型。

强制转换

强制转换主要指使用 `Number`、 `String` 和 `Boolean` 三个函数，手动将各种类型的值，分布转换成数字、字符串或者布尔值。

Number()

使用 `Number` 函数，可以将任意类型的值转化成数值。

下面分成两种情况讨论，一种是参数是原始类型的值，另一种是参数是对象。

（1）原始类型值

原始类型值的转换规则如下。

```
// 数值：转换后还是原来的值
Number(324) // 324

// 字符串：如果可以被解析为数值，则转换为相应的数值
Number('324') // 324

// 字符串：如果不可以被解析为数值，返回 NaN
```

```
Number('324abc') // NaN

// 空字符串转为0
Number('') // 0

// 布尔值：true 转成 1，false 转成 0
Number(true) // 1
Number(false) // 0

// undefined：转成 NaN
Number(undefined) // NaN

// null：转成0
Number(null) // 0
```

`Number` 函数将字符串转为数值，要比 `parseInt` 函数严格很多。基本上，只要有一个字符无法转成数值，整个字符串就会被转为 `NaN` 。

```
parseInt('42 cats') // 42
Number('42 cats') // NaN
```

上面代码中， `parseInt` 逐个解析字符，而 `Number` 函数整体转换字符串的类型。

另外， `parseInt` 和 `Number` 函数都会自动过滤一个字符串前导和后缀的空格。

```
parseInt('\t\v\r12.34\n') // 12
Number('\t\v\r12.34\n') // 12.34
```

（2）对象

简单的规则是， `Number` 方法的参数是对象时，将返回 `NaN` ，除非是包含单个数值的数组。

```
Number({a: 1}) // NaN
Number([1, 2, 3]) // NaN
Number([5]) // 5
```

之所以会这样，是因为 `Number` 背后的转换规则比较复杂。

第一步，调用对象自身的 `valueOf` 方法。如果返回原始类型的值，则直接对该值使用 `Number` 函数，不再进行后续步骤。

第二步，如果 `valueOf` 方法返回的还是对象，则改为调用对象自身的 `toString` 方法。如果 `toString` 方法返回原始类型的值，则对该值使用 `Number` 函数，不再进行后续步骤。

第三步，如果 `toString` 方法返回的是对象，就报错。

请看下面的例子。

```
var obj = {x: 1};
Number(obj) // NaN

// 等同于
if (typeof obj.valueOf() === 'object') {
  Number(obj.toString());
} else {
  Number(obj.valueOf());
}
```

上面代码中， `Number` 函数将 `obj` 对象转为数值。背后发生了一连串的操作，首先调用 `obj.valueOf` 方法， 结果返回对象本身；于是，继续调用 `obj.toString` 方法，这时返回字符串 `[object Object]` ，对这个字符串使用 `Number` 函数，得到 `NaN` 。

默认情况下，对象的 `valueOf` 方法返回对象本身，所以一般总是会调用 `toString` 方法，而 `toString` 方法返回对象的类型字符串（比

如 `[object Object]` ）。所以，会有下面的结果。

```
Number({}) // NaN
```

如果 `toString` 方法返回的不是原始类型的值，结果就会报错。

```
var obj = {
  valueOf: function () {
    return {};
  },
  toString: function () {
    return {};
  }
};

Number(obj)
// TypeError: Cannot convert object to primitive value
```

上面代码的 `valueOf` 和 `toString` 方法，返回的都是对象，所以转成数值时会报错。

从上例还可以看到， `valueOf` 和 `toString` 方法，都是可以自定义的。

```
Number({
  valueOf: function () {
    return 2;
  }
})
// 2

Number({
  toString: function () {
    return 3;
  }
})
// 3

```

```
Number({
  valueOf: function () {
    return 2;
  },
  toString: function () {
    return 3;
  }
})
// 2
```

上面代码对三个对象使用 `Number` 函数。第一个对象返回 `valueOf` 方法的值，第二个对象返回 `toString` 方法的值，第三个对象表示 `valueOf` 方法先于 `toString` 方法执行。

String()

`String` 函数可以将任意类型的值转化成字符串，转换规则如下。

（1）原始类型值

- 数值：转为相应的字符串。
- 字符串：转换后还是原来的值。
- 布尔值：`true` 转为字符串 `"true"`，`false` 转为字符串 `"false"`。
- **undefined**：转为字符串 `"undefined"`。
- **null**：转为字符串 `"null"`。

```
String(123) // "123"
String('abc') // "abc"
String(true) // "true"
String(undefined) // "undefined"
String(null) // "null"
```

（2）对象

`String` 方法的参数如果是对象，返回一个类型字符串；如果是数组，

返回该数组的字符串形式。

```
String({a: 1}) // "[object Object]"
String([1, 2, 3]) // "1,2,3"
```

`String` 方法背后的转换规则，与 `Number` 方法基本相同，只是互换了 `valueOf` 方法和 `toString` 方法的执行顺序。

1. 先调用对象自身的 `toString` 方法。如果返回原始类型的值，则对该值使用 `String` 函数，不再进行以下步骤。
2. 如果 `toString` 方法返回的是对象，再调用原对象的 `valueOf` 方法。如果 `valueOf` 方法返回原始类型的值，则对该值使用 `String` 函数，不再进行以下步骤。
3. 如果 `valueOf` 方法返回的是对象，就报错。

下面是一个例子。

```
String({a: 1})
// "[object Object]"

// 等同于
String({a: 1}.toString())
// "[object Object]"
```

上面代码先调用对象的 `toString` 方法，发现返回的是字符串 `[object Object]` ，就不再调用 `valueOf` 方法了。

如果 `toString` 法和 `valueOf` 方法，返回的都是对象，就会报错。

```
var obj = {
  valueOf: function () {
    return {};
  },
```

```
  toString: function () {
    return {};
  }
};

String(obj)
// TypeError: Cannot convert object to primitive value
```

下面是通过自定义 `toString` 方法，改变返回值的例子。

```
String({
  toString: function () {
    return 3;
  }
})
// "3"

String({
  valueOf: function () {
    return 2;
  }
})
// "[object Object]"

String({
  valueOf: function () {
    return 2;
  },
  toString: function () {
    return 3;
  }
})
// "3"
```

上面代码对三个对象使用 `String` 函数。第一个对象返回 `toString` 方法的值（数值3），第二个对象返回的还是 `toString` 方法的值（ `[object Object]` ），第三个对象表示 `toString` 方法先

于 `valueOf` 方法执行。

Boolean()

`Boolean` 函数可以将任意类型的值转为布尔值。

它的转换规则相对简单：除了以下五个值的转换结果为 `false`，其他的值全部为 `true`。

- `undefined`
- `null`
- `-0` 或 `+0`
- `NaN`
- `''`（空字符串）

```
Boolean(undefined) // false
Boolean(null) // false
Boolean(0) // false
Boolean(NaN) // false
Boolean('') // false
```

注意，所有对象（包括空对象）的转换结果都是 `true`，甚至连 `false` 对应的布尔对象 `new Boolean(false)` 也是 `true`（详见《原始类型值的包装对象》一章）。

```
Boolean({}) // true
Boolean([]) // true
Boolean(new Boolean(false)) // true
```

所有对象的布尔值都是 `true`，这是因为 JavaScript 语言设计的时候，出于性能的考虑，如果对象需要计算才能得到布尔值，对于 `obj1 && obj2` 这样的场景，可能会需要较多的计算。为了保证性能，就统一规定，对象的布尔值为 `true`。

自动转换

下面介绍自动转换，它是以强制转换为基础的。

遇到以下三种情况时，JavaScript 会自动转换数据类型，即转换是自动完成的，用户不可见。

第一种情况，不同类型的数据互相运算。

```
123 + 'abc' // "123abc"
```

第二种情况，对非布尔值类型的数据求布尔值。

```
if ('abc') {
  console.log('hello')
}  // "hello"
```

第三种情况，对非数值类型的值使用一元运算符（即 `+` 和 `-` ）。

```
+ {foo: 'bar'} // NaN
- [1, 2, 3] // NaN
```

自动转换的规则是这样的：预期什么类型的值，就调用该类型的转换函数。比如，某个位置预期为字符串，就调用 `String` 函数进行转换。如果该位置即可以是字符串，也可能是数值，那么默认转为数值。

由于自动转换具有不确定性，而且不易除错，建议在预期为布尔值、数值、字符串的地方，全部使用 `Boolean` 、 `Number` 和 `String` 函数进行显式转换。

自动转换为布尔值

JavaScript 遇到预期为布尔值的地方（比如 `if` 语句的条件部

分），就会将非布尔值的参数自动转换为布尔值。系统内部会自动调用 `Boolean` 函数。

因此除了以下五个值，其他都是自动转为 `true` 。

- `undefined`
- `null`
- `+0` 或 `-0`
- `NaN`
- `''` （空字符串）

下面这个例子中，条件部分的每个值都相当于 `false` ，使用否定运算符后，就变成了 `true` 。

```
if ( !undefined
  && !null
  && !0
  && !NaN
  && !''
) {
  console.log('true');
} // true
```

下面两种写法，有时也用于将一个表达式转为布尔值。它们内部调用的也是 `Boolean` 函数。

```
// 写法一
expression ? true : false

// 写法二
!! expression
```

自动转换为字符串

JavaScript 遇到预期为字符串的地方，就会将非字符串的值自动转为字符串。具体规则是，先将复合类型的值转为原始类型的值，再将原始类型的值转为字符串。

字符串的自动转换，主要发生在字符串的加法运算时。当一个值为字符串，另一个值为非字符串，则后者转为字符串。

```
'5' + 1 // '51'
'5' + true // "5true"
'5' + false // "5false"
'5' + {} // "5[object Object]"
'5' + [] // "5"
'5' + function (){} // "5function (){}"
'5' + undefined // "5undefined"
'5' + null // "5null"
```

这种自动转换很容易出错。

```
var obj = {
  width: '100'
};

obj.width + 20 // "10020"
```

上面代码中，开发者可能期望返回 `120` ，但是由于自动转换，实际上返回了一个字符 `10020` 。

自动转换为数值

JavaScript 遇到预期为数值的地方，就会将参数值自动转换为数值。系统内部会自动调用 `Number` 函数。

除了加法运算符（ `+` ）有可能把运算子转为字符串，其他运算符都会把运算子自动转成数值。

```
'5' - '2' // 3
'5' * '2' // 10
true - 1  // 0
false - 1 // -1
'1' - 1   // 0
'5' * []    // 0
false / '5' // 0
'abc' - 1   // NaN
null + 1 // 1
undefined + 1 // NaN
```

上面代码中，运算符两侧的运算子，都被转成了数值。

> 注意：`null` 转为数值时为 `0`，而 `undefined` 转为数值时为 `NaN`。

一元运算符也会把运算子转成数值。

```
+'abc' // NaN
-'abc' // NaN
+true // 1
-false // 0
```

参考链接

- Axel Rauschmayer, What is {} + {} in JavaScript?
- Axel Rauschmayer, JavaScript quirk 1: implicit conversion of values
- Benjie Gillam, Quantum JavaScript?

错误处理机制

错误处理机制

Error 实例对象

JavaScript 解析或运行时，一旦发生错误，引擎就会抛出一个错误对象。JavaScript 原生提供 `Error` 构造函数，所有抛出的错误都是这个构造函数的实例。

```
var err = new Error('出错了');
err.message // "出错了"
```

上面代码中，我们调用 `Error` 构造函数，生成一个实例对象 `err` 。 `Error` 构造函数接受一个参数，表示错误提示，可以从实例的 `message` 属性读到这个参数。抛出 `Error` 实例对象以后，整个程序就中断在发生错误的地方，不再往下执行。

JavaScript 语言标准只提到， `Error` 实例对象必须有 `message` 属性，表示出错时的提示信息，没有提到其他属性。大多数 JavaScript 引擎，对 `Error` 实例还提供 `name` 和 `stack` 属性，分别表示错误的名称和错误的堆栈，但它们是非标准的，不是每种实现都有。

- **message**：错误提示信息
- **name**：错误名称（非标准属性）
- **stack**：错误的堆栈（非标准属性）

使用 `name` 和 `message` 这两个属性，可以对发生什么错误有一个大概的了解。

```
if (error.name) {
  console.log(error.name + ': ' + error.message);
}
```

`stack` 属性用来查看错误发生时的堆栈。

```
function throwit() {
  throw new Error('');
}

function catchit() {
  try {
    throwit();
  } catch(e) {
    console.log(e.stack); // print stack trace
  }
```

```
}

catchit()
// Error
//    at throwit (~/examples/throwcatch.js:9:11)
//    at catchit (~/examples/throwcatch.js:3:9)
//    at repl:1:5
```

上面代码中，错误堆栈的最内层是 `throwit` 函数，然后是 `catchit` 函数，最后是函数的运行环境。

原生错误类型

`Error` 实例对象是最一般的错误类型，在它的基础上，JavaScript 还定义了其他6种错误对象。也就是说，存在 `Error` 的6个派生对象。

SyntaxError 对象

`SyntaxError` 对象是解析代码时发生的语法错误。

```
// 变量名错误
var 1a;
// Uncaught SyntaxError: Invalid or unexpected token

// 缺少括号
console.log 'hello');
// Uncaught SyntaxError: Unexpected string
```

上面代码的错误，都是在语法解析阶段就可以发现，所以会抛出 `SyntaxError` 。第一个错误提示是“token 非法”，第二个错误提示是“字符串不符合要求”。

ReferenceError 对象

`ReferenceError` 对象是引用一个不存在的变量时发生的错误。

```
// 使用一个不存在的变量
unknownVariable
// Uncaught ReferenceError: unknownVariable is not defined
```

另一种触发场景是，将一个值分配给无法分配的对象，比如对函数的运行结果或者 `this` 赋值。

```
// 等号左侧不是变量
console.log() = 1
// Uncaught ReferenceError: Invalid left-hand side in assignment

// this 对象不能手动赋值
this = 1
// ReferenceError: Invalid left-hand side in assignment
```

上面代码对函数 `console.log` 的运行结果和 `this` 赋值，结果都引发了 `ReferenceError` 错误。

RangeError 对象

`RangeError` 对象是一个值超出有效范围时发生的错误。主要有几种情况，一是数组长度为负数，二是 `Number` 对象的方法参数超出范围，以及函数堆栈超过最大值。

```
// 数组长度不得为负数
new Array(-1)
// Uncaught RangeError: Invalid array length
```

TypeError 对象

`TypeError` 对象是变量或参数不是预期类型时发生的错误。比如，对字符串、布尔值、数值等原始类型的值使用 `new` 命令，就会抛出这种错

误，因为 `new` 命令的参数应该是一个构造函数。

```
new 123
// Uncaught TypeError: number is not a func

var obj = {};
obj.unknownMethod()
// Uncaught TypeError: obj.unknownMethod is not a function
```

上面代码的第二种情况，调用对象不存在的方法，也会抛出 `TypeError` 错误，因为 `obj.unknownMethod` 的值是 `undefined` ，而不是一个函数。

URIError 对象

`URIError` 对象是 URI 相关函数的参数不正确时抛出的错误，主要涉及 `encodeURI()` 、 `decodeURI()` 、 `encodeURIComponent()` 、 `decodeURIComponent()` 、 `escape()` 和 `unescape()` 这六个函数。

```
decodeURI('%2')
// URIError: URI malformed
```

EvalError 对象

`eval` 函数没有被正确执行时，会抛出 `EvalError` 错误。该错误类型已经不再使用了，只是为了保证与以前代码兼容，才继续保留。

总结

以上这6种派生错误，连同原始的 `Error` 对象，都是构造函数。开发者可以使用它们，手动生成错误对象的实例。这些构造函数都接受一个函数，代表错误提示信息（message）。

```
var err1 = new Error('出错了！');
var err2 = new RangeError('出错了，变量超出有效范围！');
var err3 = new TypeError('出错了，变量类型无效！');

err1.message // "出错了！"
err2.message // "出错了，变量超出有效范围！"
err3.message // "出错了，变量类型无效！"
```

自定义错误

除了 JavaScript 原生提供的七种错误对象，还可以定义自己的错误对象。

```
function UserError(message) {
  this.message = message || '默认信息';
  this.name = 'UserError';
}

UserError.prototype = new Error();
UserError.prototype.constructor = UserError;
```

上面代码自定义一个错误对象 `UserError`，让它继承 `Error` 对象。然后，就可以生成这种自定义类型的错误了。

```
new UserError('这是自定义的错误！');
```

throw 语句

`throw` 语句的作用是手动中断程序执行，抛出一个错误。

```
if (x < 0) {
  throw new Error('x 必须为正数');
}
// Uncaught ReferenceError: x is not defined
```

上面代码中，如果变量 `x` 小于 `0`，就手动抛出一个错误，告诉用户 `x` 的值不正确，整个程序就会在这里中断执行。可以看到，`throw` 抛出的错误就是它的参数，这里是一个 `Error` 实例。

`throw` 也可以抛出自定义错误。

```
function UserError(message) {
  this.message = message || '默认信息';
  this.name = 'UserError';
}

throw new UserError('出错了！');
// Uncaught UserError {message: "出错了！", name: "UserError"}
```

上面代码中，`throw` 抛出的是一个 `UserError` 实例。

实际上，`throw` 可以抛出任何类型的值。也就是说，它的参数可以是任何值。

```
// 抛出一个字符串
throw 'Error！';
// Uncaught Error！

// 抛出一个数值
throw 42;
// Uncaught 42

// 抛出一个布尔值
throw true;
// Uncaught true

// 抛出一个对象
throw {
  toString: function () {
    return 'Error!';
  }
```

```
};
// Uncaught {toString: ƒ}
```

对于 JavaScript 引擎来说，遇到 `throw` 语句，程序就中止了。引擎会接收到 `throw` 抛出的信息，可能是一个错误实例，也可能是其他类型的值。

try…catch 结构

一旦发生错误，程序就中止执行了。JavaScript 提供了 `try...catch` 结构，允许对错误进行处理，选择是否往下执行。

```
try {
  throw new Error('出错了!');
} catch (e) {
  console.log(e.name + ": " + e.message);
  console.log(e.stack);
}
// Error: 出错了!
//   at <anonymous>:3:9
//   ...
```

上面代码中，`try` 代码块抛出错误（上例用的是 `throw` 语句），JavaScript 引擎就立即把代码的执行，转到 `catch` 代码块，或者说错误被 `catch` 代码块捕获了。`catch` 接受一个参数，表示 `try` 代码块抛出的值。

如果你不确定某些代码是否会报错，就可以把它们放在 `try...catch` 代码块之中，便于进一步对错误进行处理。

```
try {
  f();
} catch(e) {
  // 处理错误
```

```
5. }
```

上面代码中，如果函数 `f` 执行报错，就会进行 `catch` 代码块，接着对错误进行处理。

`catch` 代码块捕获错误之后，程序不会中断，会按照正常流程继续执行下去。

```
1. try {
2.   throw "出错了";
3. } catch (e) {
4.   console.log(111);
5. }
6. console.log(222);
7. // 111
8. // 222
```

上面代码中， `try` 代码块抛出的错误，被 `catch` 代码块捕获后，程序会继续向下执行。

`catch` 代码块之中，还可以再抛出错误，甚至使用嵌套的 `try...catch` 结构。

```
1. var n = 100;
2.
3. try {
4.   throw n;
5. } catch (e) {
6.   if (e <= 50) {
7.     // ...
8.   } else {
9.     throw e;
10.  }
11. }
12. // Uncaught 100
```

上面代码中，`catch` 代码之中又抛出了一个错误。

为了捕捉不同类型的错误，`catch` 代码块之中可以加入判断语句。

```
try {
  foo.bar();
} catch (e) {
  if (e instanceof EvalError) {
    console.log(e.name + ": " + e.message);
  } else if (e instanceof RangeError) {
    console.log(e.name + ": " + e.message);
  }
  // ...
}
```

上面代码中，`catch` 捕获错误之后，会判断错误类型（`EvalError` 还是 `RangeError`），进行不同的处理。

finally 代码块

`try...catch` 结构允许在最后添加一个 `finally` 代码块，表示不管是否出现错误，都必需在最后运行的语句。

```
function cleansUp() {
  try {
    throw new Error('出错了……');
    console.log('此行不会执行');
  } finally {
    console.log('完成清理工作');
  }
}

cleansUp()
// 完成清理工作
// Error: 出错了……
```

上面代码中，由于没有 `catch` 语句块，所以错误没有捕获。执行 `finally` 代码块以后，程序就中断在错误抛出的地方。

```
function idle(x) {
  try {
    console.log(x);
    return 'result';
  } finally {
    console.log("FINALLY");
  }
}

idle('hello')
// hello
// FINALLY
// "result"
```

上面代码说明， `try` 代码块没有发生错误，而且里面还包括 `return` 语句，但是 `finally` 代码块依然会执行。注意，只有在其执行完毕后，才会显示 `return` 语句的值。

下面的例子说明， `return` 语句的执行是排在 `finally` 代码之前，只是等 `finally` 代码执行完毕后才返回。

```
var count = 0;
function countUp() {
  try {
    return count;
  } finally {
    count++;
  }
}

countUp()
// 0
count
```

```
// 1
```

上面代码说明，`return` 语句的 `count` 的值，是在 `finally` 代码块运行之前就获取了。

下面是 `finally` 代码块用法的典型场景。

```
openFile();

try {
  writeFile(Data);
} catch(e) {
  handleError(e);
} finally {
  closeFile();
}
```

上面代码首先打开一个文件，然后在 `try` 代码块中写入文件，如果没有发生错误，则运行 `finally` 代码块关闭文件；一旦发生错误，则先使用 `catch` 代码块处理错误，再使用 `finally` 代码块关闭文件。

下面的例子充分反映了 `try...catch...finally` 这三者之间的执行顺序。

```
function f() {
  try {
    console.log(0);
    throw 'bug';
  } catch(e) {
    console.log(1);
    return true; // 这句原本会延迟到 finally 代码块结束再执行
    console.log(2); // 不会运行
  } finally {
    console.log(3);
    return false; // 这句会覆盖掉前面那句 return
    console.log(4); // 不会运行
  }
```

```
14.
15.   console.log(5); // 不会运行
16. }
17.
18. var result = f();
19. // 0
20. // 1
21. // 3
22.
23. result
24. // false
```

上面代码中，`catch` 代码块结束执行之前，会先执行 `finally` 代码块。

`catch` 代码块之中，触发转入 `finally` 代码快的标志，不仅有 `return` 语句，还有 `throw` 语句。

```
1.  function f() {
2.    try {
3.      throw '出错了！';
4.    } catch(e) {
5.      console.log('捕捉到内部错误');
6.      throw e; // 这句原本会等到finally结束再执行
7.    } finally {
8.      return false; // 直接返回
9.    }
10. }
11.
12. try {
13.   f();
14. } catch(e) {
15.   // 此处不会执行
16.   console.log('caught outer "bogus"');
17. }
18.
19. //  捕捉到内部错误
```

上面代码中，进入 `catch` 代码块之后，一遇到 `throw` 语句，就会去执行 `finally` 代码块，其中有 `return false` 语句，因此就直接返回了，不再会回去执行 `catch` 代码块剩下的部分了。

参考连接

- Jani Hartikainen, JavaScript Errors and How to Fix Them

编程风格

编程风格

概述

”编程风格“（programming style）指的是编写代码的样式规则。不同的程序员，往往有不同的编程风格。

有人说，编译器的规范叫做”语法规则“（grammar），这是程序员必须遵守的；而编译器忽略的部分，就叫”编程风格“（programming style），这是程序员可以自由选择的。这种说法不完全正确，程序员

固然可以自由选择编程风格，但是好的编程风格有助于写出质量更高、错误更少、更易于维护的程序。

所以，编程风格的选择不应该基于个人爱好、熟悉程度、打字量等因素，而要考虑如何尽量使代码清晰易读、减少出错。你选择的，不是你喜欢的风格，而是一种能够清晰表达你的意图的风格。这一点，对于 JavaScript 这种语法自由度很高的语言尤其重要。

必须牢记的一点是，如果你选定了一种“编程风格”，就应该坚持遵守，切忌多种风格混用。如果你加入他人的项目，就应该遵守现有的风格。

缩进

行首的空格和 Tab 键，都可以产生代码缩进效果（indent）。

Tab 键可以节省击键次数，但不同的文本编辑器对 Tab 的显示不尽相同，有的显示四个空格，有的显示两个空格，所以有人觉得，空格键可以使得显示效果更统一。

无论你选择哪一种方法，都是可以接受的，要做的就是始终坚持这一种选择。不要一会使用 Tab 键，一会使用空格键。

区块

如果循环和判断的代码体只有一行，JavaScript 允许该区块（block）省略大括号。

```
1. if (a)
2.   b();
3.   c();
```

上面代码的原意可能是下面这样。

```
if (a) {
  b();
  c();
}
```

但是，实际效果却是下面这样。

```
if (a) {
  b();
}
  c();
```

因此，建议总是使用大括号表示区块。

另外，区块起首的大括号的位置，有许多不同的写法。最流行的有两种，一种是起首的大括号另起一行。

```
block
{
  // ...
}
```

另一种是起首的大括号跟在关键字的后面。

```
block {
  // ...
}
```

一般来说，这两种写法都可以接受。但是，JavaScript 要使用后一种，因为 JavaScript 会自动添加句末的分号，导致一些难以察觉的错误。

```
return
{
  key: value
```

```
};

// 相当于
return;
{
  key: value
};
```

上面的代码的原意，是要返回一个对象，但实际上返回的是 `undefined`，因为 JavaScript 自动在 `return` 语句后面添加了分号。为了避免这一类错误，需要写成下面这样。

```
return {
  key : value
};
```

因此，表示区块起首的大括号，不要另起一行。

圆括号

圆括号（parentheses）在 JavaScript 中有两种作用，一种表示函数的调用，另一种表示表达式的组合（grouping）。

```
// 圆括号表示函数的调用
console.log('abc');

// 圆括号表示表达式的组合
(1 + 2) * 3
```

建议可以用空格，区分这两种不同的括号。

1. 表示函数调用时，函数名与左括号之间没有空格。
2. 表示函数定义时，函数名与左括号之间没有空格。
3. 其他情况时，前面位置的语法元素与左括号之间，都有一个空格。

按照上面的规则，下面的写法都是不规范的。

```
1. foo (bar)
2. return(a+b);
3. if(a === 0) {...}
4. function foo (b) {...}
5. function(x) {...}
```

上面代码的最后一行是一个匿名函数，`function`是语法关键字，不是函数名，所以与左括号之间应该要有一个空格。

行尾的分号

分号表示一条语句的结束。JavaScript 允许省略行尾的分号。事实上，确实有一些开发者行尾从来不写分号。但是，由于下面要讨论的原因，建议还是不要省略这个分号。

不使用分号的情况

首先，以下三种情况，语法规定本来就不需要在结尾添加分号。

（1）for 和 while 循环

```
1. for ( ; ; ) {
2. } // 没有分号
3.
4. while (true) {
5. } // 没有分号
```

注意，`do...while`循环是有分号的。

```
1. do {
2.   a--;
3. } while(a > 0); // 分号不能省略
```

（2）分支语句：if，switch，try

```
if (true) {
} // 没有分号

switch () {
} // 没有分号

try {
} catch {
} // 没有分号
```

（3）函数的声明语句

```
function f() {
} // 没有分号
```

注意，函数表达式仍然要使用分号。

```
var f = function f() {
};
```

以上三种情况，如果使用了分号，并不会出错。因为，解释引擎会把这个分号解释为空语句。

分号的自动添加

除了上一节的三种情况，所有语句都应该使用分号。但是，如果没有使用分号，大多数情况下，JavaScript 会自动添加。

```
var a = 1
// 等同于
var a = 1;
```

这种语法特性被称为“分号的自动添加”（Automatic Semicolon

Insertion，简称 ASI）。

因此，有人提倡省略句尾的分号。麻烦的是，如果下一行的开始可以与本行的结尾连在一起解释，JavaScript 就不会自动添加分号。

```
// 等同于 var a = 3
var
a
=
3

// 等同于 'abc'.length
'abc'
.length

// 等同于 return a + b;
return a +
b;

// 等同于 obj.foo(arg1, arg2);
obj.foo(arg1,
arg2);

// 等同于 3 * 2 + 10 * (27 / 6)
3 * 2
+
10 * (27 / 6)
```

上面代码都会多行放在一起解释，不会每一行自动添加分号。这些例子还是比较容易看出来的，但是下面这个例子就不那么容易看出来了。

```
x = y
(function () {
  // ...
})();

// 等同于
```

```
x = y(function () {...})();
```

下面是更多不会自动添加分号的例子。

```
// 引擎解释为 c(d+e)
var a = b + c
(d+e).toString();

// 引擎解释为 a = b/hi/g.exec(c).map(d)
// 正则表达式的斜杠，会当作除法运算符
a = b
/hi/g.exec(c).map(d);

// 解释为'b'['red', 'green'],
// 即把字符串当作一个数组，按索引取值
var a = 'b'
['red', 'green'].forEach(function (c) {
  console.log(c);
})

// 解释为 function (x) { return x }(a++)
// 即调用匿名函数，结果f等于0
var a = 0;
var f = function (x) { return x }
(a++)
```

只有下一行的开始与本行的结尾，无法放在一起解释，JavaScript 引擎才会自动添加分号。

```
if (a < 0) a = 0
console.log(a)

// 等同于下面的代码，
// 因为 0console 没有意义
if (a < 0) a = 0;
console.log(a)
```

另外，如果一行的起首是“自增”（ `++` ）或“自减”（ `--` ）运算符，则它们的前面会自动添加分号。

```
a = b = c = 1

a
++
b
--
c

console.log(a, b, c)
// 1 2 0
```

上面代码之所以会得到 `1 2 0` 的结果，原因是自增和自减运算符前，自动加上了分号。上面的代码实际上等同于下面的形式。

```
a = b = c = 1;
a;
++b;
--c;
```

如果 `continue` 、 `break` 、 `return` 和 `throw` 这四个语句后面，直接跟换行符，则会自动添加分号。这意味着，如果 `return` 语句返回的是一个对象的字面量，起首的大括号一定要写在同一行，否则得不到预期结果。

```
return
{ first: 'Jane' };

// 解释成
return;
{ first: 'Jane' };
```

由于解释引擎自动添加分号的行为难以预测，因此编写代码的时候不应

该省略行尾的分号。

不应该省略结尾的分号，还有一个原因。有些 JavaScript 代码压缩器（uglifier）不会自动添加分号，因此遇到没有分号的结尾，就会让代码保持原状，而不是压缩成一行，使得压缩无法得到最优的结果。

另外，不写结尾的分号，可能会导致脚本合并出错。所以，有的代码库在第一行语句开始前，会加上一个分号。

```
1. ;var a = 1;
2. // ...
```

上面这种写法就可以避免与其他脚本合并时，排在前面的脚本最后一行语句没有分号，导致运行出错的问题。

全局变量

JavaScript 最大的语法缺点，可能就是全局变量对于任何一个代码块，都是可读可写。这对代码的模块化和重复使用，非常不利。

因此，建议避免使用全局变量。如果不得不使用，可以考虑用大写字母表示变量名，这样更容易看出这是全局变量，比如 `UPPER_CASE` 。

变量声明

JavaScript 会自动将变量声明”提升“（hoist）到代码块（block）的头部。

```
1. if (!x) {
2.   var x = {};
3. }
4.
5. // 等同于
```

```
var x;
if (!x) {
  x = {};
}
```

这意味着，变量 `x` 是 `if` 代码块之前就存在了。为了避免可能出现的问题，最好把变量声明都放在代码块的头部。

```
for (var i = 0; i < 10; i++) {
  // ...
}

// 写成
var i;
for (i = 0; i < 10; i++) {
  // ...
}
```

上面这样的写法，就容易看出存在一个全局的循环变量 `i` 。

另外，所有函数都应该在使用之前定义。函数内部的变量声明，都应该放在函数的头部。

with 语句

`with` 可以减少代码的书写，但是会造成混淆。

```
with (o) {
  foo = bar;
}
```

上面的代码，可以有四种运行结果：

```
o.foo = bar;
o.foo = o.bar;
```

```
3.  foo = bar;
4.  foo = o.bar;
```

这四种结果都可能发生，取决于不同的变量是否有定义。因此，不要使用 `with` 语句。

相等和严格相等

JavaScript 有两个表示相等的运算符：”相等“（ `==` ）和”严格相等“（ `===` ）。

相等运算符会自动转换变量类型，造成很多意想不到的情况。

```
1.  0 == ''// true
2.  1 == true // true
3.  2 == true // false
4.  0 == '0' // true
5.  false == 'false' // false
6.  false == '0' // true
7.  ' \t\r\n ' == 0 // true
```

因此，建议不要使用相等运算符（ `==` ），只使用严格相等运算符（ `===` ）。

语句的合并

有些程序员追求简洁，喜欢合并不同目的的语句。比如，原来的语句是

```
1.  a = b;
2.  if (a) {
3.    // ...
4.  }
```

他喜欢写成下面这样。

```
if (a = b) {
  // ...
}
```

虽然语句少了一行，但是可读性大打折扣，而且会造成误读，让别人误解这行代码的意思是下面这样。

```
if (a === b) {
  // ...
}
```

建议不要将不同目的的语句，合并成一行。

自增和自减运算符

自增（`++`）和自减（`--`）运算符，放在变量的前面或后面，返回的值不一样，很容易发生错误。事实上，所有的`++`运算符都可以用`+= 1`代替。

```
++x
// 等同于
x += 1;
```

改用`+= 1`，代码变得更清晰了。

建议自增（`++`）和自减（`--`）运算符尽量使用`+=`和`-=`代替。

switch…case 结构

`switch...case`结构要求，在每一个`case`的最后一行必须是`break`语句，否则会接着运行下一个`case`。这样不仅容易忘记，还会造成代码的冗长。

而且，`switch...case`不使用大括号，不利于代码形式的统一。此外，这种结构类似于`goto`语句，容易造成程序流程的混乱，使得代码结构混乱不堪，不符合面向对象编程的原则。

```
function doAction(action) {
  switch (action) {
    case 'hack':
      return 'hack';
      break;
    case 'slash':
      return 'slash';
      break;
    case 'run':
      return 'run';
      break;
    default:
      throw new Error('Invalid action.');
  }
}
```

上面的代码建议改写成对象结构。

```
function doAction(action) {
  var actions = {
    'hack': function () {
      return 'hack';
    },
    'slash': function () {
      return 'slash';
    },
    'run': function () {
      return 'run';
    }
  };

  if (typeof actions[action] !== 'function') {
    throw new Error('Invalid action.');
```

```
16.   }
17.
18.   return actions[action]();
19. }
```

因此，建议 `switch...case` 结构可以用对象结构代替。

参考链接

- Eric Elliott, Programming JavaScript Applications, Chapter 2. JavaScript Style Guide, O’Reilly, 2013
- Axel Rauschmayer, A meta style guide for JavaScript
- Axel Rauschmayer, Automatic semicolon insertion in JavaScript
- Rod Vagg, JavaScript and Semicolons

console 对象与控制台

console 对象与控制台

console 对象

`console` 对象是 JavaScript 的原生对象，它有点像 Unix 系统的标准输出 `stdout` 和标准错误 `stderr`，可以输出各种信息到控制台，并且还提供了很多有用的辅助方法。

`console` 的常见用途有两个。

- 调试程序，显示网页代码运行时的错误信息。
- 提供了一个命令行接口，用来与网页代码互动。

`console` 对象的浏览器实现，包含在浏览器自带的开发工具之中。以 Chrome 浏览器的“开发者工具”（Developer Tools）为例，可以使用下面三种方法的打开它。

1. 按 F12 或者 `Control + Shift + i`（PC）/ `Alt + Command + i`（Mac）。
2. 浏览器菜单选择“工具/开发者工具”。
3. 在一个页面元素上，打开右键菜单，选择其中的“Inspect Element”。

打开开发者工具以后，顶端有多个面板。

- **Elements:** 查看网页的 HTML 源码和 CSS 代码。
- **Resources:** 查看网页加载的各种资源文件（比如代码文件、字体文件 CSS 文件等），以及在硬盘上创建的各种内容（比如本地缓存、Cookie、Local Storage等）。
- **Network:** 查看网页的 HTTP 通信情况。
- **Sources:** 查看网页加载的脚本源码。
- **Timeline:** 查看各种网页行为随时间变化的情况。
- **Performance:** 查看网页的性能情况，比如 CPU 和内存消耗。
- **Console:** 用来运行 JavaScript 命令。

这些面板都有各自的用途，以下只介绍 `Console` 面板（又称为控制台）。

`Console` 面板基本上就是一个命令行窗口，你可以在提示符下，键入各种命令。

console 对象的静态方法

`console` 对象提供的各种静态方法，用来与控制台窗口互动。

console.log(), console.info(), console.debug()

`console.log` 方法用于在控制台输出信息。它可以接受一个或多个参数，将它们连接起来输出。

```
console.log('Hello World')
// Hello World
console.log('a', 'b', 'c')
// a b c
```

`console.log` 方法会自动在每次输出的结尾，添加换行符。

```
console.log(1);
console.log(2);
console.log(3);
// 1
// 2
// 3
```

如果第一个参数是格式字符串（使用了格式占位符），`console.log` 方法将依次用后面的参数替换占位符，然后再进行输出。

```
console.log(' %s + %s = %s', 1, 1, 2)
//  1 + 1 = 2
```

上面代码中，`console.log` 方法的第一个参数有三个占位符（`%s`），第二、三、四个参数会在显示时，依次替换掉这个三个占位符。

`console.log` 方法支持以下占位符，不同类型的数据必须使用对应的占

位符。

- `%s` 字符串
- `%d` 整数
- `%i` 整数
- `%f` 浮点数
- `%o` 对象的链接
- `%c` CSS 格式字符串

```
var number = 11 * 9;
var color = 'red';

console.log('%d %s balloons', number, color);
// 99 red balloons
```

上面代码中，第二个参数是数值，对应的占位符是 `%d` ，第三个参数是字符串，对应的占位符是 `%s` 。

使用 `%c` 占位符时，对应的参数必须是 CSS 代码，用来对输出内容进行CSS渲染。

```
console.log(
  '%cThis text is styled!',
  'color: red; background: yellow; font-size: 24px;'
)
```

上面代码运行后，输出的内容将显示为黄底红字。

`console.log` 方法的两种参数格式，可以结合在一起使用。

```
console.log(' %s + %s ', 1, 1, '= 2')
// 1 + 1  = 2
```

如果参数是一个对象， `console.log` 会显示该对象的值。

```
console.log({foo: 'bar'})
// Object {foo: "bar"}
console.log(Date)
// function Date() { [native code] }
```

上面代码输出 `Date` 对象的值，结果为一个构造函数。

`console.info` 是 `console.log` 方法的别名，用法完全一样。只不过 `console.info` 方法会在输出信息的前面，加上一个蓝色图标。

`console.debug` 方法与 `console.log` 方法类似，会在控制台输出调试信息。但是，默认情况下， `console.debug` 输出的信息不会显示，只有在打开显示级别在 `verbose` 的情况下，才会显示。

`console` 对象的所有方法，都可以被覆盖。因此，可以按照自己的需要，定义 `console.log` 方法。

```
['log', 'info', 'warn', 'error'].forEach(function(method) {
  console[method] = console[method].bind(
    console,
    new Date().toISOString()
  );
});

console.log("出错了！");
// 2014-05-18T09:00.000Z 出错了！
```

上面代码表示，使用自定义的 `console.log` 方法，可以在显示结果添加当前时间。

console.warn(), console.error()

`warn` 方法和 `error` 方法也是在控制台输出信息，它们与 `log` 方法的不同之处在于， `warn` 方法输出信息时，在最前面加一个黄色三角，表

示警告； `error` 方法输出信息时，在最前面加一个红色的叉，表示出错。同时，还会高亮显示输出文字和错误发生的堆栈。其他方面都一样。

```
console.error('Error: %s (%i)', 'Server is not responding', 500)
// Error: Server is not responding (500)
console.warn('Warning! Too few nodes (%d)',
document.childNodes.length)
// Warning! Too few nodes (1)
```

可以这样理解， `log` 方法是写入标准输出（ `stdout` ）， `warn` 方法和 `error` 方法是写入标准错误（ `stderr` ）。

console.table()

对于某些复合类型的数据， `console.table` 方法可以将其转为表格显示。

```
var languages = [
  { name: "JavaScript", fileExtension: ".js" },
  { name: "TypeScript", fileExtension: ".ts" },
  { name: "CoffeeScript", fileExtension: ".coffee" }
];

console.table(languages);
```

上面代码的 `language` 变量，转为表格显示如下。

(index)	name	fileExtension
0	“JavaScript”	“.js”
1	“TypeScript”	“.ts”
2	“CoffeeScript”	“.coffee”

下面是显示表格内容的例子。

```
var languages = {
  csharp: { name: "C#", paradigm: "object-oriented" },
  fsharp: { name: "F#", paradigm: "functional" }
};

console.table(languages);
```

上面代码的 `language`，转为表格显示如下。

(index)	name	paradigm
csharp	“C#”	“object-oriented”
fsharp	“F#”	“functional”

console.count()

`count` 方法用于计数，输出它被调用了多少次。

```
function greet(user) {
  console.count();
  return 'hi ' + user;
}

greet('bob')
//  : 1
// "hi bob"

greet('alice')
//  : 2
// "hi alice"

greet('bob')
//  : 3
// "hi bob"
```

上面代码每次调用 `greet` 函数，内部的 `console.count` 方法就输出执行次数。

该方法可以接受一个字符串作为参数，作为标签，对执行次数进行分类。

```
function greet(user) {
  console.count(user);
  return "hi " + user;
}

greet('bob')
// bob: 1
// "hi bob"

greet('alice')
// alice: 1
// "hi alice"

greet('bob')
// bob: 2
// "hi bob"
```

上面代码根据参数的不同，显示 `bob` 执行了两次， `alice` 执行了一次。

console.dir(), console.dirxml()

`dir` 方法用来对一个对象进行检查（inspect），并以易于阅读和打印的格式显示。

```
console.log({f1: 'foo', f2: 'bar'})
// Object {f1: "foo", f2: "bar"}

console.dir({f1: 'foo', f2: 'bar'})
// Object
//   f1: "foo"
//   f2: "bar"
//   __proto__: Object
```

上面代码显示 `dir` 方法的输出结果，比 `log` 方法更易读，信息也更丰富。

该方法对于输出 DOM 对象非常有用，因为会显示 DOM 对象的所有属性。

```
1. console.dir(document.body)
```

Node 环境之中，还可以指定以代码高亮的形式输出。

```
1. console.dir(obj, {colors: true})
```

`dirxml` 方法主要用于以目录树的形式，显示 DOM 节点。

```
1. console.dirxml(document.body)
```

如果参数不是 DOM 节点，而是普通的 JavaScript 对象，`console.dirxml` 等同于 `console.dir` 。

```
1. console.dirxml([1, 2, 3])
2. // 等同于
3. console.dir([1, 2, 3])
```

console.assert()

`console.assert` 方法主要用于程序运行过程中，进行条件判断，如果不满足条件，就显示一个错误，但不会中断程序执行。这样就相当于提示用户，内部状态不正确。

它接受两个参数，第一个参数是表达式，第二个参数是字符串。只有当第一个参数为 `false` ，才会提示有错误，在控制台输出第二个参数，否则不会有任何结果。

```
console.assert(false, '判断条件不成立')
// Assertion failed: 判断条件不成立

// 相当于
try {
  if (false) {
    throw new Error('判断条件不成立');
  }
} catch(e) {
  console.error(e);
}
```

下面是一个例子，判断子节点的个数是否大于等于500。

```
console.assert(list.childNodes.length < 500, '节点个数大于等于500')
```

上面代码中，如果符合条件的节点小于500个，不会有任何输出；只有大于等于500时，才会在控制台提示错误，并且显示指定文本。

console.time(), console.timeEnd()

这两个方法用于计时，可以算出一个操作所花费的准确时间。

```
console.time('Array initialize');

var array= new Array(1000000);
for (var i = array.length - 1; i >= 0; i--) {
  array[i] = new Object();
};

console.timeEnd('Array initialize');
// Array initialize: 1914.481ms
```

`time` 方法表示计时开始，`timeEnd` 方法表示计时结束。它们的参数是计时器的名称。调用 `timeEnd` 方法之后，控制台会显示“计时器名

称：所耗费的时间”。

console.group()，console.groupend()，console.groupCollapsed()

`console.group` 和 `console.groupend` 这两个方法用于将显示的信息分组。它只在输出大量信息时有用，分在一组的信息，可以用鼠标折叠/展开。

```
1. console.group('一级分组');
2. console.log('一级分组的内容');
3.
4. console.group('二级分组');
5. console.log('二级分组的内容');
6.
7. console.groupEnd(); // 一级分组结束
8. console.groupEnd(); // 二级分组结束
```

上面代码会将“二级分组”显示在“一级分组”内部，并且“一级分类”和“二级分类”前面都有一个折叠符号，可以用来折叠本级的内容。

`console.groupCollapsed` 方法与 `console.group` 方法很类似，唯一的区别是该组的内容，在第一次显示时是收起的（collapsed），而不是展开的。

```
1. console.groupCollapsed('Fetching Data');
2.
3. console.log('Request Sent');
4. console.error('Error: Server not responding (500)');
5.
6. console.groupEnd();
```

上面代码只显示一行”Fetching Data“，点击后才会展开，显示其中包含的两行。

console.trace(), console.clear()

`console.trace` 方法显示当前执行的代码在堆栈中的调用路径。

```
console.trace()
// console.trace()
//   (anonymous function)
//   InjectedScript._evaluateOn
//   InjectedScript._evaluateAndWrap
//   InjectedScript.evaluate
```

`console.clear` 方法用于清除当前控制台的所有输出，将光标回置到第一行。如果用户选中了控制台的“Preserve log”选项， `console.clear` 方法将不起作用。

控制台命令行 API

浏览器控制台中，除了使用 `console` 对象，还可以使用一些控制台自带的命令行方法。

（1） `$_`

`$_` 属性返回上一个表达式的值。

```
2 + 2
// 4
$_
// 4
```

（2） `$0` - `$4`

控制台保存了最近5个在 Elements 面板选中的 DOM 元素， `$0` 代表倒数第一个（最近一个）， `$1` 代表倒数第二个，以此类推直到 `$4` 。

（3）`$(selector)`

`$(selector)` 返回第一个匹配的元素，等同于 `document.querySelector()` 。注意，如果页面脚本对 `$` 有定义，则会覆盖原始的定义。比如，页面里面有 jQuery，控制台执行 `$(selector)` 就会采用 jQuery 的实现，返回一个数组。

（4）`$$(selector)`

`$$(selector)` 返回选中的 DOM 对象，等同于 `document.querySelectorAll` 。

（5）`$x(path)`

`$x(path)` 方法返回一个数组，包含匹配特定 XPath 表达式的所有 DOM 元素。

```
1.  $x("//p[a]")
```

上面代码返回所有包含 `a` 元素的 `p` 元素。

（6）`inspect(object)`

`inspect(object)` 方法打开相关面板，并选中相应的元素，显示它的细节。DOM 元素在 `Elements` 面板中显示，比如 `inspect(document)` 会在 Elements 面板显示 `document` 元素。JavaScript 对象在控制台面板 `Profiles` 面板中显示，比如 `inspect(window)` 。

（7）`getEventListeners(object)`

`getEventListeners(object)` 方法返回一个对象，该对象的成员为 `object` 登记了回调函数的各种事件（比如 `click` 或 `keydown` ），每个事件对应一个数组，数组的成员为该事件的回调函数。

（8）`keys(object)`，`values(object)`

`keys(object)` 方法返回一个数组，包含 `object` 的所有键名。

`values(object)` 方法返回一个数组，包含 `object` 的所有键值。

```
var o = {'p1': 'a', 'p2': 'b'};

keys(o)
// ["p1", "p2"]
values(o)
// ["a", "b"]
```

（9）`monitorEvents(object[, events]) , unmonitorEvents(object[, events])`

`monitorEvents(object[, events])` 方法监听特定对象上发生的特定事件。事件发生时，会返回一个 `Event` 对象，包含该事件的相关信息。 `unmonitorEvents` 方法用于停止监听。

```
monitorEvents(window, "resize");
monitorEvents(window, ["resize", "scroll"])
```

上面代码分别表示单个事件和多个事件的监听方法。

```
monitorEvents($0, 'mouse');
unmonitorEvents($0, 'mousemove');
```

上面代码表示如何停止监听。

`monitorEvents` 允许监听同一大类的事件。所有事件可以分成四个大类。

- mouse: ”mousedown”, “mouseup”, “click”, “dblclick”, “mousemove”, “mouseover”,

“mouseout”, “mousewheel”
- key: ”keydown”, “keyup”, “keypress”, “textInput”
- touch: ”touchstart”, “touchmove”, “touchend”, “touchcancel”
- control: ”resize”, “scroll”, “zoom”, “focus”, “blur”, “select”, “change”, “submit”, “reset”

```
1. monitorEvents($("#msg"), "key");
```

上面代码表示监听所有 `key` 大类的事件。

（10）其他方法

命令行 API 还提供以下方法。

- `clear()`：清除控制台的历史。
- `copy(object)`：复制特定 DOM 元素到剪贴板。
- `dir(object)`：显示特定对象的所有属性，是`console.dir`方法的别名。
- `dirxml(object)`：显示特定对象的 XML 形式，是`console.dirxml`方法的别名。

debugger 语句

`debugger` 语句主要用于除错，作用是设置断点。如果有正在运行的除错工具，程序运行到 `debugger` 语句时会自动停下。如果没有除错工具，`debugger` 语句不会产生任何结果，JavaScript 引擎自动跳过这一句。

Chrome 浏览器中，当代码运行到 `debugger` 语句时，就会暂停运行，自动打开脚本源码界面。

```
for(var i = 0; i < 5; i++){
  console.log(i);
  if (i === 2) debugger;
}
```

上面代码打印出0，1，2以后，就会暂停，自动打开源码界面，等待进一步处理。

参考链接

- Chrome Developer Tools, Using the Console
- Matt West, Mastering The Developer Tools Console
- Firebug Wiki, Console API
- Axel Rauschmayer, The JavaScript console API
- Marius Schulz, Advanced JavaScript Debugging with console.table()
- Google Developer, Command Line API Reference

标准库

标准库

Object 对象

Object 对象

概述

JavaScript 原生提供 `Object` 对象（注意起首的 `O` 是大写），本章介绍该对象原生的各种方法。

JavaScript 的所有其他对象都继承自 `Object` 对象，即那些对象都是 `Object` 的实例。

`Object` 对象的原生方法分成两类：`Object` 本身的方法与 `Object` 的

实例方法。

（1）`Object` 对象本身的方法

所谓”本身的方法“就是直接定义在 `Object` 对象的方法。

```
Object.print = function (o) { console.log(o) };
```

上面代码中， `print` 方法就是直接定义在 `Object` 对象上。

（2）`Object` 的实例方法

所谓实例方法就是定义在 `Object` 原型对象 `Object.prototype` 上的方法。它可以被 `Object` 实例直接使用。

```
Object.prototype.print = function () {
  console.log(this);
};

var obj = new Object();
obj.print() // Object
```

上面代码中， `Object.prototype` 定义了一个 `print` 方法，然后生成一个 `Object` 的实例 `obj` 。 `obj` 直接继承了 `Object.prototype` 的属性和方法，可以直接使用 `obj.print` 调用 `print` 方法。也就是说， `obj` 对象的 `print` 方法实质上就是调用 `Object.prototype.print` 方法。

关于原型对象 `object.prototype` 的详细解释，参见《面向对象编程》章节。这里只要知道，凡是定义在 `Object.prototype` 对象上面的属性和方法，将被所有实例对象共享就可以了。

以下先介绍 `Object` 作为函数的用法，然后再介绍 `Object` 对象的原生方法，分成对象自身的方法（又称为”静态方法“）和实例方法两部分。

Object()

`Object` 本身是一个函数，可以当作工具方法使用，将任意值转为对象。这个方法常用于保证某个值一定是对象。

如果参数为空（或者为 `undefined` 和 `null` ），`Object()` 返回一个空对象。

```
var obj = Object();
// 等同于
var obj = Object(undefined);
var obj = Object(null);

obj instanceof Object // true
```

上面代码的含义，是将 `undefined` 和 `null` 转为对象，结果得到了一个空对象 `obj` 。

`instanceof` 运算符用来验证，一个对象是否为指定的构造函数的实例。`obj instanceof Object` 返回 `true` ，就表示 `obj` 对象是 `Object` 的实例。

如果参数是原始类型的值，`Object` 方法将其转为对应的包装对象的实例（参见《原始类型的包装对象》一章）。

```
var obj = Object(1);
obj instanceof Object // true
obj instanceof Number // true

var obj = Object('foo');
obj instanceof Object // true
obj instanceof String // true

var obj = Object(true);
obj instanceof Object // true
```

```
obj instanceof Boolean // true
```

上面代码中，`Object`函数的参数是各种原始类型的值，转换成对象就是原始类型值对应的包装对象。

如果`Object`方法的参数是一个对象，它总是返回该对象，即不用转换。

```
var arr = [];
var obj = Object(arr); // 返回原数组
obj === arr // true

var value = {};
var obj = Object(value) // 返回原对象
obj === value // true

var fn = function () {};
var obj = Object(fn); // 返回原函数
obj === fn // true
```

利用这一点，可以写一个判断变量是否为对象的函数。

```
function isObject(value) {
  return value === Object(value);
}

isObject([]) // true
isObject(true) // false
```

Object 构造函数

`Object`不仅可以当作工具函数使用，还可以当作构造函数使用，即前面可以使用`new`命令。

`Object`构造函数的首要用途，是直接通过它来生成新对象。

```
var obj = new Object();
```

> 注意，通过 `var obj = new Object()` 的写法生成新对象，与字面量的写法 `var obj = {}` 是等价的。或者说，后者只是前者的一种简便写法。

`Object` 构造函数的用法与工具方法很相似，几乎一模一样。使用时，可以接受一个参数，如果该参数是一个对象，则直接返回这个对象；如果是一个原始类型的值，则返回该值对应的包装对象（详见《包装对象》一章）。

```
var o1 = {a: 1};
var o2 = new Object(o1);
o1 === o2 // true

var obj = new Object(123);
obj instanceof Number // true
```

虽然用法相似，但是 `Object(value)` 与 `new Object(value)` 两者的语义是不同的，`Object(value)` 表示将 `value` 转成一个对象，`new Object(value)` 则表示新生成一个对象，它的值是 `value`。

Object 的静态方法

所谓“静态方法”，是指部署在 `Object` 对象自身的方法。

Object.keys(), Object.getOwnPropertyNames()

`Object.keys` 方法和 `Object.getOwnPropertyNames` 方法都用来遍历对象的属性。

`Object.keys` 方法的参数是一个对象，返回一个数组。该数组的成员都是该对象自身的（而不是继承的）所有属性名。

```
var obj = {
  p1: 123,
  p2: 456
};

Object.keys(obj) // ["p1", "p2"]
```

`Object.getOwnPropertyNames` 方法与 `Object.keys` 类似，也是接受一个对象作为参数，返回一个数组，包含了该对象自身的所有属性名。

```
var obj = {
  p1: 123,
  p2: 456
};

Object.getOwnPropertyNames(obj) // ["p1", "p2"]
```

对于一般的对象来说，`Object.keys()` 和 `Object.getOwnPropertyNames()` 返回的结果是一样的。只有涉及不可枚举属性时，才会有不一样的结果。`Object.keys` 方法只返回可枚举的属性（详见《对象属性的描述对象》一章），`Object.getOwnPropertyNames` 方法还返回不可枚举的属性名。

```
var a = ['Hello', 'World'];

Object.keys(a) // ["0", "1"]
Object.getOwnPropertyNames(a) // ["0", "1", "length"]
```

上面代码中，数组的 `length` 属性是不可枚举的属性，所以只出现在 `Object.getOwnPropertyNames` 方法的返回结果中。

由于 JavaScript 没有提供计算对象属性个数的方法，所以可以用这两个方法代替。

```
var obj = {
  p1: 123,
  p2: 456
};

Object.keys(obj).length // 2
Object.getOwnPropertyNames(obj).length // 2
```

一般情况下，几乎总是使用 `Object.keys` 方法，遍历数组的属性。

其他方法

除了上面提到的两个方法， `Object` 还有不少其他静态方法，将在后文逐一详细介绍。

（1）对象属性模型的相关方法

- `Object.getOwnPropertyDescriptor()`：获取某个属性的描述对象。
- `Object.defineProperty()`：通过描述对象，定义某个属性。
- `Object.defineProperties()`：通过描述对象，定义多个属性。

（2）控制对象状态的方法

- `Object.preventExtensions()`：防止对象扩展。
- `Object.isExtensible()`：判断对象是否可扩展。
- `Object.seal()`：禁止对象配置。
- `Object.isSealed()`：判断一个对象是否可配置。
- `Object.freeze()`：冻结一个对象。
- `Object.isFrozen()`：判断一个对象是否被冻结。

（3）原型链相关方法

- `Object.create()`：该方法可以指定原型对象和属性，返回一个新的对象。

- `Object.getPrototypeOf()`：获取对象的`Prototype`对象。

Object 的实例方法

除了静态方法，还有不少方法定义在`Object.prototype`对象。它们称为实例方法，所有`Object`的实例对象都继承了这些方法。

`Object`实例对象的方法，主要有以下六个。

- `Object.prototype.valueOf()`：返回当前对象对应的值。
- `Object.prototype.toString()`：返回当前对象对应的字符串形式。
- `Object.prototype.toLocaleString()`：返回当前对象对应的本地字符串形式。
- `Object.prototype.hasOwnProperty()`：判断某个属性是否为当前对象自身的属性，还是继承自原型对象的属性。
- `Object.prototype.isPrototypeOf()`：判断当前对象是否为另一个对象的原型。
- `Object.prototype.propertyIsEnumerable()`：判断某个属性是否可枚举。

本节介绍前四个方法，另外两个方法将在后文相关章节介绍。

Object.prototype.valueOf()

`valueOf`方法的作用是返回一个对象的“值”，默认情况下返回对象本身。

```
1. var obj = new Object();
2. obj.valueOf() === obj // true
```

上面代码比较`obj.valueOf()`与`obj`本身，两者是一样的。

`valueOf` 方法的主要用途是，JavaScript 自动类型转换时会默认调用这个方法（详见《数据类型转换》一章）。

```
var obj = new Object();
1 + obj // "1[object Object]"
```

上面代码将对象 `obj` 与数字 `1` 相加，这时 JavaScript 就会默认调用 `valueOf()` 方法，求出 `obj` 的值再与 `1` 相加。所以，如果自定义 `valueOf` 方法，就可以得到想要的结果。

```
var obj = new Object();
obj.valueOf = function () {
  return 2;
};

1 + o // 3
```

上面代码自定义了 `obj` 对象的 `valueOf` 方法，于是 `1 + o` 就得到了 `3` 。这种方法就相当于用自定义的 `obj.valueOf` ，覆盖 `Object.prototype.valueOf` 。

Object.prototype.toString()

`toString` 方法的作用是返回一个对象的字符串形式，默认情况下返回类型字符串。

```
var o1 = new Object();
o1.toString() // "[object Object]"

var o2 = {a:1};
o2.toString() // "[object Object]"
```

上面代码表示，对于一个对象调用 `toString` 方法，会返回字符

串 `[object Object]` ，该字符串说明对象的类型。

字符串 `[object Object]` 本身没有太大的用处，但是通过自定义 `toString` 方法，可以让对象在自动类型转换时，得到想要的字符串形式。

```
var obj = new Object();

obj.toString = function () {
  return 'hello';
};

obj + ' ' + 'world' // "hello world"
```

上面代码表示，当对象用于字符串加法时，会自动调用 `toString` 方法。由于自定义了 `toString` 方法，所以返回字符串 `hello world` 。

数组、字符串、函数、Date 对象都分别部署了自定义的 `toString` 方法，覆盖了 `Object.prototype.toString` 方法。

```
[1, 2, 3].toString() // "1,2,3"

'123'.toString() // "123"

(function () {
  return 123;
}).toString()
// "function () {
//   return 123;
// }"

(new Date()).toString()
// "Tue May 10 2016 09:11:31 GMT+0800 (CST)"
```

上面代码中，数组、字符串、函数、Date 对象调用 `toString` 方法，

并不会返回 `[object Object]`，因为它们都自定义了 `toString` 方法，覆盖原始方法。

toString() 的应用：判断数据类型

`Object.prototype.toString` 方法返回对象的类型字符串，因此可以用来判断一个值的类型。

```
var obj = {};
obj.toString() // "[object Object]"
```

上面代码调用空对象的 `toString` 方法，结果返回一个字符串 `object Object`，其中第二个 `Object` 表示该值的构造函数。这是一个十分有用的判断数据类型的方法。

由于实例对象可能会自定义 `toString` 方法，覆盖掉 `Object.prototype.toString` 方法，所以为了得到类型字符串，最好直接使用 `Object.prototype.toString` 方法。通过函数的 `call` 方法，可以在任意值上调用这个方法，帮助我们判断这个值的类型。

```
Object.prototype.toString.call(value)
```

上面代码表示对 `value` 这个值调用 `Object.prototype.toString` 方法。

不同数据类型的 `Object.prototype.toString` 方法返回值如下。

- 数值：返回 `[object Number]`。
- 字符串：返回 `[object String]`。
- 布尔值：返回 `[object Boolean]`。
- undefined：返回 `[object Undefined]`。
- null：返回 `[object Null]`。
- 数组：返回 `[object Array]`。

- arguments 对象：返回`[object Arguments]`。
- 函数：返回`[object Function]`。
- Error 对象：返回`[object Error]`。
- Date 对象：返回`[object Date]`。
- RegExp 对象：返回`[object RegExp]`。
- 其他对象：返回`[object Object]`。

这就是说，`Object.prototype.toString`可以看出一个值到底是什么类型。

```
Object.prototype.toString.call(2) // "[object Number]"
Object.prototype.toString.call('') // "[object String]"
Object.prototype.toString.call(true) // "[object Boolean]"
Object.prototype.toString.call(undefined) // "[object Undefined]"
Object.prototype.toString.call(null) // "[object Null]"
Object.prototype.toString.call(Math) // "[object Math]"
Object.prototype.toString.call({}) // "[object Object]"
Object.prototype.toString.call([]) // "[object Array]"
```

利用这个特性，可以写出一个比`typeof`运算符更准确的类型判断函数。

```
var type = function (o){
  var s = Object.prototype.toString.call(o);
  return s.match(/\[object (.*?)\]/)[1].toLowerCase();
};

type({}); // "object"
type([]); // "array"
type(5); // "number"
type(null); // "null"
type(); // "undefined"
type(/abcd/); // "regex"
type(new Date()); // "date"
```

在上面这个 `type` 函数的基础上，还可以加上专门判断某种类型数据的方法。

```
var type = function (o){
  var s = Object.prototype.toString.call(o);
  return s.match(/\[object (.*?)\]/)[1].toLowerCase();
};

['Null',
 'Undefined',
 'Object',
 'Array',
 'String',
 'Number',
 'Boolean',
 'Function',
 'RegExp'
].forEach(function (t) {
  type['is' + t] = function (o) {
    return type(o) === t.toLowerCase();
  };
});

type.isObject({}) // true
type.isNumber(NaN) // true
type.isRegExp(/abc/) // true
```

Object.prototype.toLocaleString()

`Object.prototype.toLocaleString` 方法与 `toString` 的返回结果相同，也是返回一个值的字符串形式。

```
var obj = {};
obj.toString(obj) // "[object Object]"
obj.toLocaleString(obj) // "[object Object]"
```

这个方法的主要作用是留出一个接口，让各种不同的对象实现自己版本的 `toLocaleString`，用来返回针对某些地域的特定的值。目前，主要有三个对象自定义了 `toLocaleString` 方法。

- Array.prototype.toLocaleString()
- Number.prototype.toLocaleString()
- Date.prototype.toLocaleString()

举例来说，日期的实例对象的 `toString` 和 `toLocaleString` 返回值就不一样，而且 `toLocaleString` 的返回值跟用户设定的所在地域相关。

```
var date = new Date();
date.toString() // "Tue Jan 01 2018 12:01:33 GMT+0800 (CST)"
date.toLocaleString() // "1/01/2018, 12:01:33 PM"
```

Object.prototype.hasOwnProperty()

`Object.prototype.hasOwnProperty` 方法接受一个字符串作为参数，返回一个布尔值，表示该实例对象自身是否具有该属性。

```
var obj = {
  p: 123
};

obj.hasOwnProperty('p') // true
obj.hasOwnProperty('toString') // false
```

上面代码中，对象 `obj` 自身具有 `p` 属性，所以返回 `true`。 `toString` 属性是继承的，所以返回 `false`。

参考链接

- Axel Rauschmayer, Protecting objects in

JavaScript

- kangax, Understanding delete
- Jon Bretman, Type Checking in JavaScript
- Cody Lindley, Thinking About ECMAScript 5 Parts
- Bjorn Tipling, Advanced objects in JavaScript
- Javier Márquez, Javascript properties are enumerable, writable and configurable
- Sella Rafaeli, Native JavaScript Data-Binding: 使用存取函数实现model与view的双向绑定
- Lea Verou, Copying object properties, the robust way

属性描述对象

属性描述对象

概述

JavaScript 提供了一个内部数据结构，用来描述对象的属性，控制它的行为，比如该属性是否可写、可遍历等等。这个内部数据结构称为“属性描述对象”（attributes object）。每个属性都有自己对应的属性描述对象，保存该属性的一些元信息。

下面是属性描述对象的一个例子。

```
{
  value: 123,
  writable: false,
  enumerable: true,
  configurable: false,
  get: undefined,
  set: undefined
}
```

属性描述对象提供6个元属性。

（1）`value`

`value` 是该属性的属性值，默认为 `undefined` 。

（2）`writable`

`writable` 是一个布尔值，表示属性值（value）是否可改变（即是否可写），默认为 `true` 。

（3）`enumerable`

`enumerable` 是一个布尔值，表示该属性是否可遍历，默认为 `true` 。如果设为 `false` ，会使得某些操作（比如 `for...in` 循环、 `Object.keys()` ）跳过该属性。

（4）`configurable`

`configurable` 是一个布尔值，表示可配置性，默认为 `true` 。如果设为 `false` ，将阻止某些操作改写该属性，比如无法删除该属性，也不得改变该属性的属性描述对象（ `value` 属性除外）。也就是说， `configurable` 属性控制了属性描述对象的可写性。

（5） `get`

`get` 是一个函数，表示该属性的取值函数（getter），默认为 `undefined` 。

（6） `set`

`set` 是一个函数，表示该属性的存值函数（setter），默认为 `undefined` 。

Object.getOwnPropertyDescriptor()

`Object.getOwnPropertyDescriptor` 方法可以获取属性描述对象。它的第一个参数是一个对象，第二个参数是一个字符串，对应该对象的某个属性名。

```
var obj = { p: 'a' };

Object.getOwnPropertyDescriptor(obj, 'p')
// Object { value: "a",
//   writable: true,
//   enumerable: true,
//   configurable: true
// }
```

上面代码中， `Object.getOwnPropertyDescriptor` 方法获取 `obj.p` 的属性描述对象。

注意， `Object.getOwnPropertyDescriptor` 方法只能用于对象自身的属

性，不能用于继承的属性。

```
var obj = { p: 'a' };

Object.getOwnPropertyDescriptor(obj, 'toString')
// undefined
```

上面代码中，`toString` 是 `Obj` 对象继承的属性，`Object.getOwnPropertyDescriptor` 无法获取。

Object.getOwnPropertyNames()

`Object.getOwnPropertyNames` 方法返回一个数组，成员是参数对象自身的全部属性的属性名，不管该属性是否可遍历。

```
var obj = Object.defineProperties({}, {
  p1: { value: 1, enumerable: true },
  p2: { value: 2, enumerable: false }
});

Object.getOwnPropertyNames(obj)
// ["p1", "p2"]
```

上面代码中，`obj.p1` 是可遍历的，`obj.p2` 是不可遍历的。`Object.getOwnPropertyNames` 会将它们都返回。

这跟 `Object.keys` 的行为不同，`Object.keys` 只返回对象自身的可遍历属性的全部属性名。

```
Object.keys([]) // []
Object.getOwnPropertyNames([]) // [ 'length' ]

Object.keys(Object.prototype) // []
Object.getOwnPropertyNames(Object.prototype)
// ['hasOwnProperty',
```

```
//  'valueOf',
//  'constructor',
//  'toLocaleString',
//  'isPrototypeOf',
//  'propertyIsEnumerable',
//  'toString']
```

上面代码中，数组自身的 `length` 属性是不可遍历的， `Object.keys` 不会返回该属性。第二个例子的 `Object.prototype` 也是一个对象，所有实例对象都会继承它，它自身的属性都是不可遍历的。

Object.defineProperty(), Object.defineProperties()

`Object.defineProperty` 方法允许通过属性描述对象，定义或修改一个属性，然后返回修改后的对象，它的用法如下。

```
Object.defineProperty(object, propertyName, attributesObject)
```

`Object.defineProperty` 方法接受三个参数，依次如下。

- 属性所在的对象
- 属性名（它应该是一个字符串）
- 属性描述对象

举例来说，定义 `obj.p` 可以写成下面这样。

```
var obj = Object.defineProperty({}, 'p', {
  value: 123,
  writable: false,
  enumerable: true,
  configurable: false
});

```

```
obj.p // 123

obj.p = 246;
obj.p // 123
```

上面代码中，`Object.defineProperty` 方法定义了 `obj.p` 属性。由于属性描述对象的 `writable` 属性为 `false` ，所以 `obj.p` 属性不可写。注意，这里的 `Object.defineProperty` 方法的第一个参数是 `{}` （一个新建的空对象），`p` 属性直接定义在这个空对象上面，然后返回这个对象，这是 `Object.defineProperty` 的常见写法。

如果属性已经存在，`Object.defineProperty` 方法相当于更新该属性的属性描述对象。

如果一次性定义或修改多个属性，可以使用 `Object.defineProperties` 方法。

```
var obj = Object.defineProperties({}, {
  p1: { value: 123, enumerable: true },
  p2: { value: 'abc', enumerable: true },
  p3: { get: function () { return this.p1 + this.p2 },
    enumerable:true,
    configurable:true
  }
});

obj.p1 // 123
obj.p2 // "abc"
obj.p3 // "123abc"
```

上面代码中，`Object.defineProperties` 同时定义了 `obj` 对象的三个属性。其中，`p3` 属性定义了取值函数 `get` ，即每次读取该属性，都会调用这个取值函数。

注意，一旦定义了取值函数 `get` （或存值函数 `set` ），就不能

将 `writable` 属性设为 `true` ，或者同时定义 `value` 属性，否则会报错。

```
var obj = {};

Object.defineProperty(obj, 'p', {
  value: 123,
  get: function() { return 456; }
});
// TypeError: Invalid property.
// A property cannot both have accessors and be writable or have a value

Object.defineProperty(obj, 'p', {
  writable: true,
  get: function() { return 456; }
});
// TypeError: Invalid property descriptor.
// Cannot both specify accessors and a value or writable attribute
```

上面代码中，同时定义了 `get` 属性和 `value` 属性，以及将 `writable` 属性设为 `true` ，就会报错。

`Object.defineProperty()` 和 `Object.defineProperties()` 的第三个参数，是一个属性对象。它的 `writable` 、 `configurable` 、 `enumerable` 这三个属性的默认值都为 `false` 。

```
var obj = {};
Object.defineProperty(obj, 'foo', {});
Object.getOwnPropertyDescriptor(obj, 'foo')
// {
//   value: undefined,
//   writable: false,
//   enumerable: false,
//   configurable: falsee
// }
```

上面代码中，定义 `obj.p` 时用了一个空的属性描述对象，就可以看到各个元属性的默认值。

Object.prototype.propertyIsEnumerable()

实例对象的 `propertyIsEnumerable` 方法返回一个布尔值，用来判断某个属性是否可遍历。

```
var obj = {};
obj.p = 123;

obj.propertyIsEnumerable('p') // true
obj.propertyIsEnumerable('toString') // false
```

上面代码中， `obj.p` 是可遍历的，而继承自原型对象的 `obj.toString` 属性是不可遍历的。

元属性

属性描述对象的各个属性称为“元属性”，因为它们可以看作是控制属性的属性。

value

`value` 属性是目标属性的值。

```
var obj = {};
obj.p = 123;

Object.getOwnPropertyDescriptor(obj, 'p').value
// 123

```

```
Object.defineProperty(obj, 'p', { value: 246 });
obj.p // 246
```

上面代码是通过 `value` 属性，读取或改写 `obj.p` 的例子。

writable

`writable` 属性是一个布尔值，决定了目标属性的值（value）是否可以被改变。

```
var obj = {};

Object.defineProperty(obj, 'a', {
  value: 37,
  writable: false
});

obj.a // 37
obj.a = 25;
obj.a // 37
```

上面代码中，`obj.a` 的 `writable` 属性是 `false`。然后，改变 `obj.a` 的值，不会有任何效果。

注意，正常模式下，对 `writable` 为 `false` 的属性赋值不会报错，只会默默失败。但是，严格模式下会报错，即使对 `a` 属性重新赋予一个同样的值。

```
'use strict';
var obj = {};

Object.defineProperty(obj, 'a', {
  value: 37,
  writable: false
});
```

```
8.
9.  obj.a = 37;
10. // Uncaught TypeError: Cannot assign to read only property 'a' of
    object
```

上面代码是严格模式，对 `obj.a` 任何赋值行为都会报错。

如果原型对象的某个属性的 `writable` 为 `false` ，那么子对象将无法自定义这个属性。

```
1.  var proto = Object.defineProperty({}, 'foo', {
2.    value: 'a',
3.    writable: false
4.  });
5.
6.  var obj = Object.create(proto);
7.
8.  obj.foo = 'b';
9.  obj.foo // 'a'
```

上面代码中， `proto` 是原型对象，它的 `foo` 属性不可写。 `obj` 对象继承 `proto` ，也不可以再自定义这个属性了。如果是严格模式，这样做还会抛出一个错误。

但是，有一个规避方法，就是通过覆盖属性描述对象，绕过这个限制。原因是这种情况下，原型链会被完全忽视。

```
1.  var proto = Object.defineProperty({}, 'foo', {
2.    value: 'a',
3.    writable: false
4.  });
5.
6.  var obj = Object.create(proto);
7.  Object.defineProperty(obj, 'foo', {
8.    value: 'b'
9.  });
```

```
10.
11.  obj.foo // "b"
```

enumerable

`enumerable`（可遍历性）返回一个布尔值，表示目标属性是否可遍历。

JavaScript 的早期版本，`for...in`循环是基于`in`运算符的。我们知道，`in`运算符不管某个属性是对象自身的还是继承的，都会返回`true`。

```
1.  var obj = {};
2.  'toString' in obj // true
```

上面代码中，`toString`不是`obj`对象自身的属性，但是`in`运算符也返回`true`，这导致了`toString`属性也会被`for...in`循环遍历。

这显然不太合理，后来就引入了“可遍历性”这个概念。只有可遍历的属性，才会被`for...in`循环遍历，同时还规定`toString`这一类实例对象继承的原生属性，都是不可遍历的，这样就保证了`for...in`循环的可用性。

具体来说，如果一个属性的`enumerable`为`false`，下面三个操作不会取到该属性。

- `for..in`循环
- `Object.keys`方法
- `JSON.stringify`方法

因此，`enumerable`可以用来设置“秘密”属性。

```
1.  var obj = {};
```

```
2.
3.  Object.defineProperty(obj, 'x', {
4.    value: 123,
5.    enumerable: false
6.  });
7.
8.  obj.x // 123
9.
10. for (var key in obj) {
11.   console.log(key);
12. }
13. // undefined
14.
15. Object.keys(obj)  // []
16. JSON.stringify(obj) // "{a:1, b:2, c:3}"
```

上面代码中，`obj.x` 属性的 `enumerable` 为 `false`，所以一般的遍历操作都无法获取该属性，使得它有点像“秘密”属性，但不是真正的私有属性，还是可以直接获取它的值。

注意，`for...in` 循环包括继承的属性，`Object.keys` 方法不包括继承的属性。如果需要获取对象自身的所有属性，不管是否可遍历，可以使用 `Object.getOwnPropertyNames` 方法。

另外，`JSON.stringify` 方法会排除 `enumerable` 为 `false` 的属性，有时可以利用这一点。如果对象的 JSON 格式输出要排除某些属性，就可以把这些属性的 `enumerable` 设为 `false`。

configurable

`configurable`（可配置性）返回一个布尔值，决定了是否可以修改属性描述对象。也就是说，`configurable` 为 `false` 时，`value`、`writable`、`enumerable` 和 `configurable` 都不能被修改了。

```
var obj = Object.defineProperty({}, 'p', {
  value: 1,
  writable: false,
  enumerable: false,
  configurable: false
});

Object.defineProperty(obj, 'p', {value: 2})
// TypeError: Cannot redefine property: p

Object.defineProperty(obj, 'p', {writable: true})
// TypeError: Cannot redefine property: p

Object.defineProperty(obj, 'p', {enumerable: true})
// TypeError: Cannot redefine property: p

Object.defineProperty(obj, 'p', {configurable: true})
// TypeError: Cannot redefine property: p
```

上面代码中，`obj.p` 的 `configurable` 为 `false`。然后，改动 `value`、`writable`、`enumerable`、`configurable`，结果都报错。

注意，`writable` 只有在 `false` 改为 `true` 会报错，`true` 改为 `false` 是允许的。

```
var obj = Object.defineProperty({}, 'p', {
  writable: true,
  configurable: false
});

Object.defineProperty(obj, 'p', {writable: false})
// 修改成功
```

至于 `value`，只要 `writable` 和 `configurable` 有一个为 `true`，就允许改动。

```
var o1 = Object.defineProperty({}, 'p', {
  value: 1,
  writable: true,
  configurable: false
});

Object.defineProperty(o1, 'p', {value: 2})
// 修改成功

var o2 = Object.defineProperty({}, 'p', {
  value: 1,
  writable: false,
  configurable: true
});

Object.defineProperty(o2, 'p', {value: 2})
// 修改成功
```

另外，`configurable` 为 `false` 时，直接目标属性赋值，不报错，但不会成功。

```
var obj = Object.defineProperty({}, 'p', {
  value: 1,
  configurable: false
});

obj.p = 2;
obj.p // 1
```

上面代码中，`obj.p` 的 `configurable` 为 `false`，对 `obj.p` 赋值是不会生效的。如果是严格模式，还会报错。

可配置性决定了目标属性是否可以被删除（delete）。

```
var obj = Object.defineProperties({}, {
  p1: { value: 1, configurable: true },
```

```
  p2: { value: 2, configurable: false }
});

delete obj.p1 // true
delete obj.p2 // false

obj.p1 // undefined
obj.p2 // 2
```

上面代码中，`obj.p1` 的 `configurable` 是 `true`，所以可以被删除，`obj.p2` 就无法删除。

存取器

除了直接定义以外，属性还可以用存取器（accessor）定义。其中，存值函数称为 `setter`，使用属性描述对象的 `set` 属性；取值函数称为 `getter`，使用属性描述对象的 `get` 属性。

一旦对目标属性定义了存取器，那么存取的时候，都将执行对应的函数。利用这个功能，可以实现许多高级特性，比如某个属性禁止赋值。

```
var obj = Object.defineProperty({}, 'p', {
  get: function () {
    return 'getter';
  },
  set: function (value) {
    console.log('setter: ' + value);
  }
});

obj.p // "getter"
obj.p = 123 // "setter: 123"
```

上面代码中，`obj.p` 定义了 `get` 和 `set` 属性。`obj.p` 取值时，就会调用 `get`；赋值时，就会调用 `set`。

JavaScript 还提供了存取器的另一种写法。

```
var obj = {
  get p() {
    return 'getter';
  },
  set p(value) {
    console.log('setter: ' + value);
  }
};
```

上面的写法与定义属性描述对象是等价的，而且使用更广泛。

注意，取值函数 `get` 不能接受参数，存值函数 `set` 只能接受一个参数（即属性的值）。

存取器往往用于，属性的值依赖对象内部数据的场合。

```
var obj ={
  $n : 5,
  get next() { return this.$n++ },
  set next(n) {
    if (n >= this.$n) this.$n = n;
    else throw new Error('新的值必须大于当前值');
  }
};

obj.next // 5

obj.next = 10;
obj.next // 10

obj.next = 5;
// Uncaught Error: 新的值必须大于当前值
```

上面代码中， `next` 属性的存值函数和取值函数，都依赖于内部属

性 `$n` 。

对象的拷贝

有时，我们需要将一个对象的所有属性，拷贝到另一个对象，可以用下面的方法实现。

```
var extend = function (to, from) {
  for (var property in from) {
    to[property] = from[property];
  }

  return to;
}

extend({}, {
  a: 1
})
// {a: 1}
```

上面这个方法的问题在于，如果遇到存取器定义的属性，会只拷贝值。

```
extend({}, {
  get a() { return 1 }
})
// {a: 1}
```

为了解决这个问题，我们可以通过 `Object.defineProperty` 方法来拷贝属性。

```
var extend = function (to, from) {
  for (var property in from) {
    if (!from.hasOwnProperty(property)) continue;
    Object.defineProperty(
      to,
```

```
      property,
      Object.getOwnPropertyDescriptor(from, property)
    );
  }

  return to;
}

extend({}, { get a(){ return 1 } })
// { get a(){ return 1 } })
```

上面代码中，`hasOwnProperty`那一行用来过滤掉继承的属性，否则会报错，因为`Object.getOwnPropertyDescriptor`读不到继承属性的属性描述对象。

控制对象状态

有时需要冻结对象的读写状态，防止对象被改变。JavaScript 提供了三种冻结方法，最弱的一种是`Object.preventExtensions`，其次是`Object.seal`，最强的是`Object.freeze`。

Object.preventExtensions()

`Object.preventExtensions`方法可以使得一个对象无法再添加新的属性。

```
var obj = new Object();
Object.preventExtensions(obj);

Object.defineProperty(obj, 'p', {
  value: 'hello'
});
// TypeError: Cannot define property:p, object is not extensible.

obj.p = 1;
```

```
obj.p // undefined
```

上面代码中，`obj`对象经过`Object.preventExtensions`以后，就无法添加新属性了。

Object.isExtensible()

`Object.isExtensible`方法用于检查一个对象是否使用了`Object.preventExtensions`方法。也就是说，检查是否可以为一个对象添加属性。

```
var obj = new Object();

Object.isExtensible(obj) // true
Object.preventExtensions(obj);
Object.isExtensible(obj) // false
```

上面代码中，对`obj`对象使用`Object.preventExtensions`方法以后，再使用`Object.isExtensible`方法，返回`false`，表示已经不能添加新属性了。

Object.seal()

`Object.seal`方法使得一个对象既无法添加新属性，也无法删除旧属性。

```
var obj = { p: 'hello' };
Object.seal(obj);

delete obj.p;
obj.p // "hello"

obj.x = 'world';
obj.x // undefined
```

上面代码中，`obj`对象执行`Object.seal`方法以后，就无法添加新属性和删除旧属性了。

`Object.seal`实质是把属性描述对象的`configurable`属性设为`false`，因此属性描述对象不再能改变了。

```
var obj = {
  p: 'a'
};

// seal方法之前
Object.getOwnPropertyDescriptor(obj, 'p')
// Object {
//   value: "a",
//   writable: true,
//   enumerable: true,
//   configurable: true
// }

Object.seal(obj);

// seal方法之后
Object.getOwnPropertyDescriptor(obj, 'p')
// Object {
//   value: "a",
//   writable: true,
//   enumerable: true,
//   configurable: false
// }

Object.defineProperty(o, 'p', {
  enumerable: false
})
// TypeError: Cannot redefine property: p
```

上面代码中，使用`Object.seal`方法之后，属性描述对象

的 `configurable` 属性就变成了 `false` ，然后改变 `enumerable` 属性就会报错。

`Object.seal` 只是禁止新增或删除属性，并不影响修改某个属性的值。

```
var obj = { p: 'a' };
Object.seal(obj);
obj.p = 'b';
obj.p // 'b'
```

上面代码中， `Object.seal` 方法对 `p` 属性的 `value` 无效，是因为此时 `p` 属性的可写性由 `writable` 决定。

Object.isSealed()

`Object.isSealed` 方法用于检查一个对象是否使用了 `Object.seal` 方法。

```
var obj = { p: 'a' };

Object.seal(obj);
Object.isSealed(obj) // true
```

这时， `Object.isExtensible` 方法也返回 `false` 。

```
var obj = { p: 'a' };

Object.seal(obj);
Object.isExtensible(obj) // false
```

Object.freeze()

`Object.freeze` 方法可以使得一个对象无法添加新属性、无法删除旧属性、也无法改变属性的值，使得这个对象实际上变成了常量。

```
var obj = {
  p: 'hello'
};

Object.freeze(obj);

obj.p = 'world';
obj.p // "hello"

obj.t = 'hello';
obj.t // undefined

delete obj.p // false
obj.p // "hello"
```

上面代码中，对 `obj` 对象进行 `Object.freeze()` 以后，修改属性、新增属性、删除属性都无效了。这些操作并不报错，只是默默地失败。如果在严格模式下，则会报错。

Object.isFrozen()

`Object.isFrozen` 方法用于检查一个对象是否使用了 `Object.freeze` 方法。

```
var obj = {
  p: 'hello'
};

Object.freeze(obj);
Object.isFrozen(obj) // true
```

使用 `Object.freeze` 方法以后， `Object.isSealed` 将会返回 `true` ， `Object.isExtensible` 返回 `false` 。

```
var obj = {
```

```
  p: 'hello'
};

Object.freeze(obj);

Object.isSealed(obj) // true
Object.isExtensible(obj) // false
```

`Object.isFrozen` 的一个用途是，确认某个对象没有被冻结后，再对它的属性赋值。

```
var obj = {
  p: 'hello'
};

Object.freeze(obj);

if (!Object.isFrozen(obj)) {
  obj.p = 'world';
}
```

上面代码中，确认 `obj` 没有被冻结后，再对它的属性赋值，就不会报错了。

局限性

上面的三个方法锁定对象的可写性有一个漏洞：可以通过改变原型对象，来为对象增加属性。

```
var obj = new Object();
Object.preventExtensions(obj);

var proto = Object.getPrototypeOf(obj);
proto.t = 'hello';
obj.t
// hello
```

上面代码中，对象 `obj` 本身不能新增属性，但是可以在它的原型对象上新增属性，就依然能够在 `obj` 上读到。

一种解决方案是，把 `obj` 的原型也冻结住。

```
var obj = new Object();
Object.preventExtensions(obj);

var proto = Object.getPrototypeOf(obj);
Object.preventExtensions(proto);

proto.t = 'hello';
obj.t // undefined
```

另外一个局限是，如果属性值是对象，上面这些方法只能冻结属性指向的对象，而不能冻结对象本身的内容。

```
var obj = {
  foo: 1,
  bar: ['a', 'b']
};
Object.freeze(obj);

obj.bar.push('c');
obj.bar // ["a", "b", "c"]
```

上面代码中，`obj.bar` 属性指向一个数组，`obj` 对象被冻结以后，这个指向无法改变，即无法指向其他值，但是所指向的数组是可以改变的。

Array 对象

Array 对象

构造函数

`Array` 是 JavaScript 的原生对象，同时也是一个构造函数，可以用它生成新的数组。

```
var arr = new Array(2);
arr.length // 2
arr // [ empty x 2 ]
```

上面代码中，`Array` 构造函数的参数 `2` ，表示生成一个两个成员的数组，每个位置都是空值。

如果没有使用 `new` ，运行结果也是一样的。

```
var arr = new Array(2);
// 等同于
var arr = Array(2);
```

`Array` 构造函数有一个很大的缺陷，就是不同的参数，会导致它的行为不一致。

```
// 无参数时，返回一个空数组
new Array() // []

// 单个正整数参数，表示返回的新数组的长度
new Array(1) // [ empty ]
new Array(2) // [ empty x 2 ]

// 非正整数的数值作为参数，会报错
new Array(3.2) // RangeError: Invalid array length
new Array(-3) // RangeError: Invalid array length

// 单个非数值（比如字符串、布尔值、对象等）作为参数，
// 则该参数是返回的新数组的成员
new Array('abc') // ['abc']
new Array([1]) // [Array[1]]

// 多参数时，所有参数都是返回的新数组的成员
```

```
new Array(1, 2) // [1, 2]
new Array('a', 'b', 'c') // ['a', 'b', 'c']
```

可以看到，`Array` 作为构造函数，行为很不一致。因此，不建议使用它生成新数组，直接使用数组字面量是更好的做法。

```
// bad
var arr = new Array(1, 2);

// good
var arr = [1, 2];
```

注意，如果参数是一个正整数，返回数组的成员都是空位。虽然读取的时候返回 `undefined`，但实际上该位置没有任何值。虽然可以取到 `length` 属性，但是取不到键名。

```
var a = new Array(3);
var b = [undefined, undefined, undefined];

a.length // 3
b.length // 3

a[0] // undefined
b[0] // undefined

0 in a // false
0 in b // true
```

上面代码中，`a` 是一个长度为3的空数组，`b` 是一个三个成员都是 `undefined` 的数组。读取键值的时候，`a` 和 `b` 都返回 `undefined`，但是 `a` 的键位都是空的，`b` 的键位是有值的。

静态方法

Array.isArray()

`Array.isArray` 方法返回一个布尔值，表示参数是否为数组。它可以弥补 `typeof` 运算符的不足。

```
var arr = [1, 2, 3];

typeof arr // "object"
Array.isArray(arr) // true
```

上面代码中，`typeof` 运算符只能显示数组的类型是 `Object`，而 `Array.isArray` 方法可以识别数组。

实例方法

valueOf(), toString()

`valueOf` 方法是一个所有对象都拥有的方法，表示对该对象求值。不同对象的 `valueOf` 方法不尽一致，数组的 `valueOf` 方法返回数组本身。

```
var arr = [1, 2, 3];
arr.valueOf() // [1, 2, 3]
```

`toString` 方法也是对象的通用方法，数组的 `toString` 方法返回数组的字符串形式。

```
var arr = [1, 2, 3];
arr.toString() // "1,2,3"

var arr = [1, 2, 3, [4, 5, 6]];
arr.toString() // "1,2,3,4,5,6"
```

push(), pop()

`push` 方法用于在数组的末端添加一个或多个元素，并返回添加新元素后的数组长度。注意，该方法会改变原数组。

```
var arr = [];

arr.push(1) // 1
arr.push('a') // 2
arr.push(true, {}) // 4
arr // [1, 'a', true, {}]
```

上面代码使用 `push` 方法，往数组中添加了四个成员。

`pop` 方法用于删除数组的最后一个元素，并返回该元素。注意，该方法会改变原数组。

```
var arr = ['a', 'b', 'c'];

arr.pop() // 'c'
arr // ['a', 'b']
```

对空数组使用 `pop` 方法，不会报错，而是返回 `undefined` 。

```
[].pop() // undefined
```

`push` 和 `pop` 结合使用，就构成了“后进先出”的栈结构（stack）。

```
var arr = [];
arr.push(1, 2);
arr.push(3);
arr.pop();
arr // [1, 2]
```

上面代码中， `3` 是最后进入数组的，但是最早离开数组。

shift(), unshift()

`shift` 方法用于删除数组的第一个元素，并返回该元素。注意，该方法会改变原数组。

```
var a = ['a', 'b', 'c'];

a.shift() // 'a'
a // ['b', 'c']
```

`shift` 方法可以遍历并清空一个数组。

```
var list = [1, 2, 3, 4, 5, 6];
var item;

while (item = list.shift()) {
  console.log(item);
}

list // []
```

`push` 和 `shift` 结合使用，就构成了“先进先出”的队列结构（queue）。

`unshift` 方法用于在数组的第一个位置添加元素，并返回添加新元素后的数组长度。注意，该方法会改变原数组。

```
var a = ['a', 'b', 'c'];

a.unshift('x'); // 4
a // ['x', 'a', 'b', 'c']
```

`unshift` 方法可以接受多个参数，这些参数都会添加到目标数组头部。

```
var arr = [ 'c', 'd' ];
arr.unshift('a', 'b') // 4
arr // [ 'a', 'b', 'c', 'd' ]
```

join()

`join` 方法以指定参数作为分隔符，将所有数组成员连接为一个字符串返回。如果不提供参数，默认用逗号分隔。

```
var a = [1, 2, 3, 4];

a.join(' ') // '1 2 3 4'
a.join(' | ') // "1 | 2 | 3 | 4"
a.join() // "1,2,3,4"
```

如果数组成员是 `undefined` 或 `null` 或空位，会被转成空字符串。

```
[undefined, null].join('#')
// '#'

['a',, 'b'].join('-')
// 'a--b'
```

通过 `call` 方法，这个方法也可以用于字符串或类似数组的对象。

```
Array.prototype.join.call('hello', '-')
// "h-e-l-l-o"

var obj = { 0: 'a', 1: 'b', length: 2 };
Array.prototype.join.call(obj, '-')
// 'a-b'
```

concat()

`concat` 方法用于多个数组的合并。它将新数组的成员，添加到原数组成员的后部，然后返回一个新数组，原数组不变。

```
['hello'].concat(['world'])
// ["hello", "world"]
```

```
3.
4.  ['hello'].concat(['world'], ['!'])
5.  // ["hello", "world", "!"]
6.
7.  [].concat({a: 1}, {b: 2})
8.  // [{ a: 1 }, { b: 2 }]
9.
10. [2].concat({a: 1})
11. // [2, {a: 1}]
```

除了数组作为参数，`concat` 也接受其他类型的值作为参数，添加到目标数组尾部。

```
1.  [1, 2, 3].concat(4, 5, 6)
2.  // [1, 2, 3, 4, 5, 6]
```

如果数组成员包括对象，`concat` 方法返回当前数组的一个浅拷贝。所谓“浅拷贝”，指的是新数组拷贝的是对象的引用。

```
1.  var obj = { a: 1 };
2.  var oldArray = [obj];
3.
4.  var newArray = oldArray.concat();
5.
6.  obj.a = 2;
7.  newArray[0].a // 2
```

上面代码中，原数组包含一个对象，`concat` 方法生成的新数组包含这个对象的引用。所以，改变原对象以后，新数组跟着改变。

reverse()

`reverse` 方法用于颠倒排列数组元素，返回改变后的数组。注意，该方法将改变原数组。

```
var a = ['a', 'b', 'c'];

a.reverse() // ["c", "b", "a"]
a // ["c", "b", "a"]
```

slice()

`slice` 方法用于提取目标数组的一部分，返回一个新数组，原数组不变。

```
arr.slice(start, end);
```

它的第一个参数为起始位置（从0开始），第二个参数为终止位置（但该位置的元素本身不包括在内）。如果省略第二个参数，则一直返回到原数组的最后一个成员。

```
var a = ['a', 'b', 'c'];

a.slice(0) // ["a", "b", "c"]
a.slice(1) // ["b", "c"]
a.slice(1, 2) // ["b"]
a.slice(2, 6) // ["c"]
a.slice() // ["a", "b", "c"]
```

上面代码中，最后一个例子 `slice` 没有参数，实际上等于返回一个原数组的拷贝。

如果 `slice` 方法的参数是负数，则表示倒数计算的位置。

```
var a = ['a', 'b', 'c'];
a.slice(-2) // ["b", "c"]
a.slice(-2, -1) // ["b"]
```

上面代码中， `-2` 表示倒数计算的第二个位置， `-1` 表示倒数计算的

第一个位置。

如果第一个参数大于等于数组长度，或者第二个参数小于第一个参数，则返回空数组。

```
var a = ['a', 'b', 'c'];
a.slice(4) // []
a.slice(2, 1) // []
```

`slice` 方法的一个重要应用，是将类似数组的对象转为真正的数组。

```
Array.prototype.slice.call({ 0: 'a', 1: 'b', length: 2 })
// ['a', 'b']

Array.prototype.slice.call(document.querySelectorAll("div"));
Array.prototype.slice.call(arguments);
```

上面代码的参数都不是数组，但是通过 `call` 方法，在它们上面调用 `slice` 方法，就可以把它们转为真正的数组。

splice()

`splice` 方法用于删除原数组的一部分成员，并可以在删除的位置添加新的数组成员，返回值是被删除的元素。注意，该方法会改变原数组。

```
arr.splice(start, count, addElement1, addElement2, ...);
```

`splice` 的第一个参数是删除的起始位置（从0开始），第二个参数是被删除的元素个数。如果后面还有更多的参数，则表示这些就是要被插入数组的新元素。

```
var a = ['a', 'b', 'c', 'd', 'e', 'f'];
a.splice(4, 2) // ["e", "f"]
a // ["a", "b", "c", "d"]
```

上面代码从原数组4号位置，删除了两个数组成员。

```
var a = ['a', 'b', 'c', 'd', 'e', 'f'];
a.splice(4, 2, 1, 2) // ["e", "f"]
a // ["a", "b", "c", "d", 1, 2]
```

上面代码除了删除成员，还插入了两个新成员。

起始位置如果是负数，就表示从倒数位置开始删除。

```
var a = ['a', 'b', 'c', 'd', 'e', 'f'];
a.splice(-4, 2) // ["c", "d"]
```

上面代码表示，从倒数第四个位置 `c` 开始删除两个成员。

如果只是单纯地插入元素， `splice` 方法的第二个参数可以设为 `0` 。

```
var a = [1, 1, 1];

a.splice(1, 0, 2) // []
a // [1, 2, 1, 1]
```

如果只提供第一个参数，等同于将原数组在指定位置拆分成两个数组。

```
var a = [1, 2, 3, 4];
a.splice(2) // [3, 4]
a // [1, 2]
```

sort()

`sort` 方法对数组成员进行排序，默认是按照字典顺序排序。排序后，原数组将被改变。

```
['d', 'c', 'b', 'a'].sort()
// ['a', 'b', 'c', 'd']
```

```
3.
4.  [4, 3, 2, 1].sort()
5.  // [1, 2, 3, 4]
6.
7.  [11, 101].sort()
8.  // [101, 11]
9.
10. [10111, 1101, 111].sort()
11. // [10111, 1101, 111]
```

上面代码的最后两个例子，需要特殊注意。`sort` 方法不是按照大小排序，而是按照字典顺序。也就是说，数值会被先转成字符串，再按照字典顺序进行比较，所以 `101` 排在 `11` 的前面。

如果想让 `sort` 方法按照自定义方式排序，可以传入一个函数作为参数。

```
1.  [10111, 1101, 111].sort(function (a, b) {
2.    return a - b;
3.  })
4.  // [111, 1101, 10111]
```

上面代码中，`sort` 的参数函数本身接受两个参数，表示进行比较的两个数组成员。如果该函数的返回值大于 `0` ，表示第一个成员排在第二个成员后面；其他情况下，都是第一个元素排在第二个元素前面。

```
1.  [
2.    { name: "张三", age: 30 },
3.    { name: "李四", age: 24 },
4.    { name: "王五", age: 28  }
5.  ].sort(function (o1, o2) {
6.    return o1.age - o2.age;
7.  })
8.  // [
9.  //   { name: "李四", age: 24 },
10. //   { name: "王五", age: 28  },
```

```
//   { name: "张三", age: 30 }
// ]
```

map()

`map` 方法将数组的所有成员依次传入参数函数，然后把每一次的执行结果组成一个新数组返回。

```
var numbers = [1, 2, 3];

numbers.map(function (n) {
  return n + 1;
});
// [2, 3, 4]

numbers
// [1, 2, 3]
```

上面代码中，`numbers` 数组的所有成员依次执行参数函数，运行结果组成一个新数组返回，原数组没有变化。

`map` 方法接受一个函数作为参数。该函数调用时，`map` 方法向它传入三个参数：当前成员、当前位置和数组本身。

```
[1, 2, 3].map(function(elem, index, arr) {
  return elem * index;
});
// [0, 2, 6]
```

上面代码中，`map` 方法的回调函数有三个参数，`elem` 为当前成员的值，`index` 为当前成员的位置，`arr` 为原数组（`[1, 2, 3]`）。

`map` 方法还可以接受第二个参数，用来绑定回调函数内部的 `this` 变量（详见《this 变量》一章）。

```
var arr = ['a', 'b', 'c'];

[1, 2].map(function (e) {
  return this[e];
}, arr)
// ['b', 'c']
```

上面代码通过 `map` 方法的第二个参数，将回调函数内部的 `this` 对象，指向 `arr` 数组。

如果数组有空位， `map` 方法的回调函数在这个位置不会执行，会跳过数组的空位。

```
var f = function (n) { return 'a' };

[1, undefined, 2].map(f) // ["a", "a", "a"]
[1, null, 2].map(f) // ["a", "a", "a"]
[1, , 2].map(f) // ["a", , "a"]
```

上面代码中， `map` 方法不会跳过 `undefined` 和 `null` ，但是会跳过空位。

forEach()

`forEach` 方法与 `map` 方法很相似，也是对数组的所有成员依次执行参数函数。但是， `forEach` 方法不返回值，只用来操作数据。这就是说，如果数组遍历的目的是为了得到返回值，那么使用 `map` 方法，否则使用 `forEach` 方法。

`forEach` 的用法与 `map` 方法一致，参数是一个函数，该函数同样接受三个参数：当前值、当前位置、整个数组。

```
function log(element, index, array) {
  console.log('[' + index + '] = ' + element);
```

```
}

[2, 5, 9].forEach(log);
// [0] = 2
// [1] = 5
// [2] = 9
```

上面代码中，`forEach` 遍历数组不是为了得到返回值，而是为了在屏幕输出内容，所以不必使用 `map` 方法。

`forEach` 方法也可以接受第二个参数，绑定参数函数的 `this` 变量。

```
var out = [];

[1, 2, 3].forEach(function(elem) {
  this.push(elem * elem);
}, out);

out // [1, 4, 9]
```

上面代码中，空数组 `out` 是 `forEach` 方法的第二个参数，结果，回调函数内部的 `this` 关键字就指向 `out` 。

注意，`forEach` 方法无法中断执行，总是会将所有成员遍历完。如果希望符合某种条件时，就中断遍历，要使用 `for` 循环。

```
var arr = [1, 2, 3];

for (var i = 0; i < arr.length; i++) {
  if (arr[i] === 2) break;
  console.log(arr[i]);
}
// 2
```

上面代码中，执行到数组的第二个成员时，就会中断执行。`forEach` 方

法做不到这一点。

`forEach` 方法也会跳过数组的空位。

```
var log = function (n) {
  console.log(n + 1);
};

[1, undefined, 2].forEach(log)
// 2
// NaN
// 3

[1, null, 2].forEach(log)
// 2
// 1
// 3

[1, , 2].forEach(log)
// 2
// 3
```

上面代码中，`forEach` 方法不会跳过 `undefined` 和 `null` ，但会跳过空位。

filter()

`filter` 方法用于过滤数组成员，满足条件的成员组成一个新数组返回。

它的参数是一个函数，所有数组成员依次执行该函数，返回结果为 `true` 的成员组成一个新数组返回。该方法不会改变原数组。

```
[1, 2, 3, 4, 5].filter(function (elem) {
  return (elem > 3);
})
```

```
// [4, 5]
```

上面代码将大于 `3` 的数组成员，作为一个新数组返回。

```
var arr = [0, 1, 'a', false];

arr.filter(Boolean)
// [1, "a"]
```

上面代码中， `filter` 方法返回数组 `arr` 里面所有布尔值为 `true` 的成员。

`filter` 方法的参数函数可以接受三个参数：当前成员，当前位置和整个数组。

```
[1, 2, 3, 4, 5].filter(function (elem, index, arr) {
  return index % 2 === 0;
});
// [1, 3, 5]
```

上面代码返回偶数位置的成员组成的新数组。

`filter` 方法还可以接受第二个参数，用来绑定参数函数内部的 `this` 变量。

```
var obj = { MAX: 3 };
var myFilter = function (item) {
  if (item > this.MAX) return true;
};

var arr = [2, 8, 3, 4, 1, 3, 2, 9];
arr.filter(myFilter, obj) // [8, 4, 9]
```

上面代码中，过滤器 `myFilter` 内部有 `this` 变量，它可以被 `filter` 方法的第二个参数 `obj` 绑定，返回大于 `3` 的成员。

some(), every()

这两个方法类似“断言”（assert），返回一个布尔值，表示判断数组成员是否符合某种条件。

它们接受一个函数作为参数，所有数组成员依次执行该函数。该函数接受三个参数：当前成员、当前位置和整个数组，然后返回一个布尔值。

`some` 方法是只要一个成员的返回值是 `true`，则整个 `some` 方法的返回值就是 `true`，否则返回 `false`。

```
var arr = [1, 2, 3, 4, 5];
arr.some(function (elem, index, arr) {
  return elem >= 3;
});
// true
```

上面代码中，如果数组 `arr` 有一个成员大于等于3，`some` 方法就返回 `true`。

`every` 方法是所有成员的返回值都是 `true`，整个 `every` 方法才返回 `true`，否则返回 `false`。

```
var arr = [1, 2, 3, 4, 5];
arr.every(function (elem, index, arr) {
  return elem >= 3;
});
// false
```

上面代码中，数组 `arr` 并非所有成员大于等于 `3`，所以返回 `false`。

注意，对于空数组，`some` 方法返回 `false`，`every` 方法返回 `true`，回调函数都不会执行。

```
function isEven(x) { return x % 2 === 0 }

[].some(isEven) // false
[].every(isEven) // true
```

`some` 和 `every` 方法还可以接受第二个参数，用来绑定参数函数内部的 `this` 变量。

reduce(), reduceRight()

`reduce` 方法和 `reduceRight` 方法依次处理数组的每个成员，最终累计为一个值。它们的差别是，`reduce` 是从左到右处理（从第一个成员到最后一个成员），`reduceRight` 则是从右到左（从最后一个成员到第一个成员），其他完全一样。

```
[1, 2, 3, 4, 5].reduce(function (a, b) {
  console.log(a, b);
  return a + b;
})
// 1 2
// 3 3
// 6 4
// 10 5
//最后结果：15
```

上面代码中，`reduce` 方法求出数组所有成员的和。第一次执行，`a` 是数组的第一个成员 `1`，`b` 是数组的第二个成员 `2`。第二次执行，`a` 为上一轮的返回值 `3`，`b` 为第三个成员 `3`。第三次执行，`a` 为上一轮的返回值 `6`，`b` 为第四个成员 `4`。第四次执行，`a` 为上一轮返回值 `10`，`b` 为第五个成员 `5`。至此所有成员遍历完成，整个方法的返回值就是最后一轮的返回值 `15`。

`reduce` 方法和 `reduceRight` 方法的第一个参数都是一个函数。该函数

接受以下四个参数。

1. 累积变量，默认为数组的第一个成员
2. 当前变量，默认为数组的第二个成员
3. 当前位置（从0开始）
4. 原数组

这四个参数之中，只有前两个是必须的，后两个则是可选的。

如果要对累积变量指定初值，可以把它放在 `reduce` 方法和 `reduceRight` 方法的第二个参数。

```
[1, 2, 3, 4, 5].reduce(function (a, b) {
  return a + b;
}, 10);
// 25
```

上面代码指定参数 `a` 的初值为10，所以数组从10开始累加，最终结果为25。注意，这时 `b` 是从数组的第一个成员开始遍历。

上面的第二个参数相当于设定了默认值，处理空数组时尤其有用。

```
function add(prev, cur) {
  return prev + cur;
}

[].reduce(add)
// TypeError: Reduce of empty array with no initial value
[].reduce(add, 1)
// 1
```

上面代码中，由于空数组取不到初始值， `reduce` 方法会报错。这时，加上第二个参数，就能保证总是会返回一个值。

下面是一个 `reduceRight` 方法的例子。

```
function substract(prev, cur) {
  return prev - cur;
}

[3, 2, 1].reduce(substract) // 0
[3, 2, 1].reduceRight(substract) // -4
```

上面代码中，`reduce` 方法相当于 `3` 减去 `2` 再减去 `1`，`reduceRight` 方法相当于 `1` 减去 `2` 再减去 `3`。

由于这两个方法会遍历数组，所以实际上还可以用来做一些遍历相关的操作。比如，找出字符长度最长的数组成员。

```
function findLongest(entries) {
  return entries.reduce(function (longest, entry) {
    return entry.length > longest.length ? entry : longest;
  }, '');
}

findLongest(['aaa', 'bb', 'c']) // "aaa"
```

上面代码中，`reduce` 的参数函数会将字符长度较长的那个数组成员，作为累积值。这导致遍历所有成员之后，累积值就是字符长度最长的那个成员。

indexOf(), lastIndexOf()

`indexOf` 方法返回给定元素在数组中第一次出现的位置，如果没有出现则返回 `-1`。

```
var a = ['a', 'b', 'c'];

a.indexOf('b') // 1
a.indexOf('y') // -1
```

`indexOf` 方法还可以接受第二个参数，表示搜索的开始位置。

```
['a', 'b', 'c'].indexOf('a', 1) // -1
```

上面代码从1号位置开始搜索字符 `a` ，结果为 `-1` ，表示没有搜索到。

`lastIndexOf` 方法返回给定元素在数组中最后一次出现的位置，如果没有出现则返回 `-1` 。

```
var a = [2, 5, 9, 2];
a.lastIndexOf(2) // 3
a.lastIndexOf(7) // -1
```

注意，这两个方法不能用来搜索 `NaN` 的位置，即它们无法确定数组成员是否包含 `NaN` 。

```
[NaN].indexOf(NaN) // -1
[NaN].lastIndexOf(NaN) // -1
```

这是因为这两个方法内部，使用严格相等运算符（ `===` ）进行比较，而 `NaN` 是唯一一个不等于自身的值。

链式使用

上面这些数组方法之中，有不少返回的还是数组，所以可以链式使用。

```
var users = [
  {name: 'tom', email: 'tom@example.com'},
  {name: 'peter', email: 'peter@example.com'}
];

users
.map(function (user) {
```

```
  return user.email;
})
.filter(function (email) {
  return /^t/.test(email);
})
.forEach(console.log);
// "tom@example.com"
```

上面代码中，先产生一个所有 Email 地址组成的数组，然后再过滤出以 `t` 开头的 Email 地址。

参考链接

- Nicolas Bevacqua, Fun with JavaScript Native Array Functions

包装对象

包装对象

定义

对象是 JavaScript 语言最主要的数据类型，三种原始类型的值——数值、字符串、布尔值——在一定条件下，也会自动转为对象，也就是原始类型的“包装对象”。

所谓“包装对象”，就是分别与数值、字符串、布尔值相对应的 `Number` 、 `String` 、 `Boolean` 三个原生对象。这三个原生对象可以把原始类型的值变成（包装成）对象。

```
var v1 = new Number(123);
var v2 = new String('abc');
var v3 = new Boolean(true);
```

上面代码中，基于原始类型的值，生成了三个对应的包装对象。

```
typeof v1 // "object"
typeof v2 // "object"
typeof v3 // "object"
```

```
4.
5.  v1 === 123 // false
6.  v2 === 'abc' // false
7.  v3 === true // false
```

包装对象的最大目的，首先是使得 JavaScript 的对象涵盖所有的值，其次使得原始类型的值可以方便地调用某些方法。

`Number`、`String`和`Boolean`如果不作为构造函数调用（即调用时不加`new`），常常用于将任意类型的值转为数值、字符串和布尔值。

```
1.  Number(123) // 123
2.  String('abc') // "abc"
3.  Boolean(true) // true
```

上面这种数据类型的转换，详见《数据类型转换》一节。

总结一下，这三个对象作为构造函数使用（带有`new`）时，可以将原始类型的值转为对象；作为普通函数使用时（不带有`new`），可以将任意类型的值，转为原始类型的值。

实例方法

三种包装对象各自提供了许多实例方法，详见后文。这里介绍两种它们共同具有、从`Object`对象继承的方法：`valueOf`和`toString`。

valueOf()

`valueOf`方法返回包装对象实例对应的原始类型的值。

```
1.  new Number(123).valueOf()  // 123
2.  new String('abc').valueOf() // "abc"
3.  new Boolean(true).valueOf() // true
```

toString()

`toString` 方法返回对应的字符串形式。

```
new Number(123).toString() // "123"
new String('abc').toString() // "abc"
new Boolean(true).toString() // "true"
```

原始类型与实例对象的自动转换

原始类型的值，可以自动当作对象调用，即调用各种对象的方法和参数。这时，JavaScript 引擎会自动将原始类型的值转为包装对象实例，在使用后立刻销毁实例。

比如，字符串可以调用 `length` 属性，返回字符串的长度。

```
'abc'.length // 3
```

上面代码中，`abc` 是一个字符串，本身不是对象，不能调用 `length` 属性。JavaScript 引擎自动将其转为包装对象，在这个对象上调用 `length` 属性。调用结束后，这个临时对象就会被销毁。这就叫原始类型与实例对象的自动转换。

```
var str = 'abc';
str.length // 3

// 等同于
var strObj = new String(str)
// String {
//   0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"
// }
strObj.length // 3
```

上面代码中，字符串 `abc` 的包装对象提供了多个属性。

自动转换生成的包装对象是只读的，无法修改。所以，字符串无法添加新属性。

```
var s = 'Hello World';
s.x = 123;
s.x // undefined
```

上面代码为字符串 `s` 添加了一个 `x` 属性，结果无效，总是返回 `undefined` 。

另一方面，调用结束后，包装对象实例会自动销毁。这意味着，下一次调用字符串的属性时，实际是调用一个新生成的对象，而不是上一次调用时生成的那个对象，所以取不到赋值在上一个对象的属性。如果要为字符串添加属性，只有在它的原型对象 `String.prototype` 上定义（参见《面向对象编程》章节）。

自定义方法

除了原生的实例方法，包装对象还可以自定义方法和属性，供原始类型的值直接调用。

比如，我们可以新增一个 `double` 方法，使得字符串和数字翻倍。

```
String.prototype.double = function () {
  return this.valueOf() + this.valueOf();
};

'abc'.double()
// abcabc

Number.prototype.double = function () {
  return this.valueOf() + this.valueOf();
};

```

```
(123).double()
// 246
```

上面代码在 `123` 外面必须要加上圆括号，否则后面的点运算符（ `.` ）会被解释成小数点。

但是，这种自定义方法和属性的机制，只能定义在包装对象的原型上，如果直接对原始类型的变量添加属性，则无效。

```
var s = 'abc';

s.p = 123;
s.p // undefined
```

上面代码直接对字符串 `abc` 添加属性，结果无效。主要原因是上面说的，这里的包装对象是自动生成的，赋值后自动销毁，所以最后一行实际上调用的是一个新的包装对象。

Boolean 对象

Boolean 对象

概述

`Boolean`对象是 JavaScript 的三个包装对象之一。作为构造函数，它主要用于生成布尔值的包装对象实例。

```
var b = new Boolean(true);

typeof b // "object"
b.valueOf() // true
```

上面代码的变量`b`是一个`Boolean`对象的实例，它的类型是对象，值为布尔值`true`。

注意，`false`对应的包装对象实例，布尔运算结果也是`true`。

```
if (new Boolean(false)) {
  console.log('true');
} // true

if (new Boolean(false).valueOf()) {
  console.log('true');
} // 无输出
```

上面代码的第一个例子之所以得到`true`，是因为`false`对应的包装对象实例是一个对象，进行逻辑运算时，被自动转化成布尔

值 `true`（因为所有对象对应的布尔值都是 `true`）。而实例的 `valueOf` 方法，则返回实例对应的原始值，本例为 `false`。

Boolean 函数的类型转换作用

`Boolean` 对象除了可以作为构造函数，还可以单独使用，将任意值转为布尔值。这时 `Boolean` 就是一个单纯的工具方法。

```
Boolean(undefined) // false
Boolean(null) // false
Boolean(0) // false
Boolean('') // false
Boolean(NaN) // false

Boolean(1) // true
Boolean('false') // true
Boolean([]) // true
Boolean({}) // true
Boolean(function () {}) // true
Boolean(/foo/) // true
```

上面代码中几种得到 `true` 的情况，都值得认真记住。

顺便提一下，使用双重的否运算符（`!`）也可以将任意值转为对应的布尔值。

```
!!undefined // false
!!null // false
!!0 // false
!!'' // false
!!NaN // false
!!1 // true
!!'false' // true
!![] // true
!!{} // true
```

```
!!function(){} // true
!!/foo/ // true
```

最后，对于一些特殊值，`Boolean` 对象前面加不加 `new`，会得到完全相反的结果，必须小心。

```
if (Boolean(false)) {
  console.log('true');
} // 无输出

if (new Boolean(false)) {
  console.log('true');
} // true

if (Boolean(null)) {
  console.log('true');
} // 无输出

if (new Boolean(null)) {
  console.log('true');
} // true
```

Number 对象

Number 对象

概述

`Number`对象是数值对应的包装对象，可以作为构造函数使用，也可以作为工具函数使用。

作为构造函数时，它用于生成值为数值的对象。

```
var n = new Number(1);
typeof n // "object"
```

上面代码中，`Number`对象作为构造函数使用，返回一个值为`1`的对象。

作为工具函数时，它可以将任何类型的值转为数值。

```
Number(true) // 1
```

上面代码将布尔值 `true` 转为数值 `1` 。 `Number` 作为工具函数的用法，详见《数据类型转换》一章。

静态属性

`Number` 对象拥有以下一些静态属性（即直接定义在 `Number` 对象上的属性，而不是定义在实例上的属性）。

- `Number.POSITIVE_INFINITY` ：正的无限，指向 `Infinity` 。
- `Number.NEGATIVE_INFINITY` ：负的无限，指向 `-Infinity` 。
- `Number.NaN` ：表示非数值，指向 `NaN` 。
- `Number.MIN_VALUE` ：表示最小的正数（即最接近0的正数，在64位浮点数体系中为 `5e-324` ），相应的，最接近0的负数为 `-Number.MIN_VALUE` 。
- `Number.MAX_SAFE_INTEGER` ：表示能够精确表示的最大整数，即 `9007199254740991` 。
- `Number.MIN_SAFE_INTEGER` ：表示能够精确表示的最小整数，即 `-9007199254740991` 。

```
Number.POSITIVE_INFINITY // Infinity
Number.NEGATIVE_INFINITY // -Infinity
Number.NaN // NaN

Number.MAX_VALUE
// 1.7976931348623157e+308
Number.MAX_VALUE < Infinity
// true

Number.MIN_VALUE
// 5e-324
Number.MIN_VALUE > 0
// true

```

```
Number.MAX_SAFE_INTEGER // 9007199254740991
Number.MIN_SAFE_INTEGER // -9007199254740991
```

实例方法

`Number` 对象有4个实例方法，都跟将数值转换成指定格式有关。

Number.prototype.toString()

`Number` 对象部署了自己的 `toString` 方法，用来将一个数值转为字符串形式。

```
(10).toString() // "10"
```

`toString` 方法可以接受一个参数，表示输出的进制。如果省略这个参数，默认将数值先转为十进制，再输出字符串；否则，就根据参数指定的进制，将一个数字转化成某个进制的字符串。

```
(10).toString(2) // "1010"
(10).toString(8) // "12"
(10).toString(16) // "a"
```

上面代码中， `10` 一定要放在括号里，这样表明后面的点表示调用对象属性。如果不加括号，这个点会被 JavaScript 引擎解释成小数点，从而报错。

```
10.toString(2)
// SyntaxError: Unexpected token ILLEGAL
```

只要能够让 JavaScript 引擎不混淆小数点和对象的点运算符，各种写法都能用。除了为 `10` 加上括号，还可以在 `10` 后面加两个点，JavaScript 会把第一个点理解成小数点（即 `10.0` ），把第二个点

理解成调用对象属性，从而得到正确结果。

```
10..toString(2)
// "1010"

// 其他方法还包括
10 .toString(2) // "1010"
10.0.toString(2) // "1010"
```

这实际上意味着，可以直接对一个小数使用 `toString` 方法。

```
10.5.toString() // "10.5"
10.5.toString(2) // "1010.1"
10.5.toString(8) // "12.4"
10.5.toString(16) // "a.8"
```

通过方括号运算符也可以调用 `toString` 方法。

```
10['toString'](2) // "1010"
```

`toString` 方法只能将十进制的数，转为其他进制的字符串。如果要将其他进制的数，转回十进制，需要使用 `parseInt` 方法。

Number.prototype.toFixed()

`toFixed` 方法先将一个数转为指定位数的小数，然后返回这个小数对应的字符串。

```
(10).toFixed(2) // "10.00"
10.005.toFixed(2) // "10.01"
```

上面代码中， `10` 和 `10.005` 转成2位小数，其中 `10` 必须放在括号里，否则后面的点会被处理成小数点。

`toFixed` 方法的参数为小数位数，有效范围为0到20，超出这个范围将抛出 RangeError 错误。

Number.prototype.toExponential()

`toExponential` 方法用于将一个数转为科学计数法形式。

```
(10).toExponential()  // "1e+1"
(10).toExponential(1) // "1.0e+1"
(10).toExponential(2) // "1.00e+1"

(1234).toExponential()  // "1.234e+3"
(1234).toExponential(1) // "1.2e+3"
(1234).toExponential(2) // "1.23e+3"
```

`toExponential` 方法的参数是小数点后有效数字的位数，范围为0到20，超出这个范围，会抛出一个 RangeError 错误。

Number.prototype.toPrecision()

`toPrecision` 方法用于将一个数转为指定位数的有效数字。

```
(12.34).toPrecision(1) // "1e+1"
(12.34).toPrecision(2) // "12"
(12.34).toPrecision(3) // "12.3"
(12.34).toPrecision(4) // "12.34"
(12.34).toPrecision(5) // "12.340"
```

`toPrecision` 方法的参数为有效数字的位数，范围是1到21，超出这个范围会抛出 RangeError 错误。

`toPrecision` 方法用于四舍五入时不太可靠，跟浮点数不是精确储存有关。

```
(12.35).toPrecision(3) // "12.3"
(12.25).toPrecision(3) // "12.3"
(12.15).toPrecision(3) // "12.2"
(12.45).toPrecision(3) // "12.4"
```

自定义方法

与其他对象一样，`Number.prototype` 对象上面可以自定义方法，被 `Number` 的实例继承。

```
Number.prototype.add = function (x) {
  return this + x;
};

8['add'](2) // 10
```

上面代码为 `Number` 对象实例定义了一个 `add` 方法。在数值上调用某个方法，数值会自动转为 `Number` 的实例对象，所以就可以调用 `add` 方法了。由于 `add` 方法返回的还是数值，所以可以链式运算。

```
Number.prototype.subtract = function (x) {
  return this - x;
};

(8).add(2).subtract(4)
// 6
```

上面代码在 `Number` 对象的实例上部署了 `subtract` 方法，它可以与 `add` 方法链式调用。

我们还可以部署更复杂的方法。

```
Number.prototype.iterate = function () {
  var result = [];
```

```
  for (var i = 0; i <= this; i++) {
    result.push(i);
  }
  return result;
};

(8).iterate()
// [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

上面代码在 `Number` 对象的原型上部署了 `iterate` 方法，将一个数值自动遍历为一个数组。

注意，数值的自定义方法，只能定义在它的原型对象 `Number.prototype` 上面，数值本身是无法自定义属性的。

```
var n = 1;
n.x = 1;
n.x // undefined
```

上面代码中， `n` 是一个原始类型的数值。直接在它上面新增一个属性 `x` ，不会报错，但毫无作用，总是返回 `undefined` 。这是因为一旦被调用属性， `n` 就自动转为 `Number` 的实例对象，调用结束后，该对象自动销毁。所以，下一次调用 `n` 的属性时，实际取到的是另一个对象，属性 `x` 当然就读不出来。

String 对象

String 对象

概述

`String` 对象是 JavaScript 原生提供的三个包装对象之一，用来生成字符串对象。

```
var s1 = 'abc';
var s2 = new String('abc');

typeof s1 // "string"
typeof s2 // "object"

s2.valueOf() // "abc"
```

上面代码中，变量 `s1` 是字符串， `s2` 是对象。由于 `s2` 是字符串对象， `s2.valueOf` 方法返回的就是它所对应的原始字符串。

字符串对象是一个类似数组的对象（很像数组，但不是数组）。

```
new String('abc')
// String {0: "a", 1: "b", 2: "c", length: 3}

(new String('abc'))[1] // "b"
```

上面代码中，字符串 `abc` 对应的字符串对象，有数值键（ `0` 、 `1` 、 `2` ）和 `length` 属性，所以可以像数组那样取值。

除了用作构造函数， `String` 对象还可以当作工具方法使用，将任意类型的值转为字符串。

```
String(true) // "true"
String(5) // "5"
```

上面代码将布尔值 `ture` 和数值 `5` ，分别转换为字符串。

静态方法

String.fromCharCode()

`String` 对象提供的静态方法（即定义在对象本身，而不是定义在对象实例的方法），主要是 `String.fromCharCode()`。该方法的参数是一个或多个数值，代表 Unicode 码点，返回值是这些码点组成的字符串。

```
String.fromCharCode() // ""
String.fromCharCode(97) // "a"
String.fromCharCode(104, 101, 108, 108, 111)
// "hello"
```

上面代码中，`String.fromCharCode` 方法的参数为空，就返回空字符串；否则，返回参数对应的 Unicode 字符串。

注意，该方法不支持 Unicode 码点大于 `0xFFFF` 的字符，即传入的参数不能大于 `0xFFFF`（即十进制的 65535）。

```
String.fromCharCode(0x20BB7)
// "ஷ"
String.fromCharCode(0x20BB7) === String.fromCharCode(0x0BB7)
// true
```

上面代码中，`String.fromCharCode` 参数 `0x20BB7` 大于 `0xFFFF`，导致返回结果出错。`0x20BB7` 对应的字符是汉字 `?`，但是返回结果却是另一个字符（码点 `0x0BB7`）。这是因为 `String.fromCharCode` 发现参数值大于 `0xFFFF`，就会忽略多出的位（即忽略 `0x20BB7` 里面的 `2`）。

这种现象的根本原因在于，码点大于 `0xFFFF` 的字符占用四个字节，而 JavaScript 默认支持两个字节的字符。这种情况下，必须把 `0x20BB7` 拆成两个字符表示。

```
String.fromCharCode(0xD842, 0xDFB7)
// "?"
```

上面代码中，`0x20BB7` 拆成两个字符 `0xD842` 和 `0xDFB7`（即两个两字节字符，合成一个四字节字符），就能得到正确的结果。码点大于 `0xFFFF` 的字符的四字节表示法，由 UTF-16 编码方法决定。

实例属性

String.prototype.length

字符串实例的 `length` 属性返回字符串的长度。

```
'abc'.length // 3
```

实例方法

String.prototype.charAt()

`charAt` 方法返回指定位置的字符，参数是从 `0` 开始编号的位置。

```
var s = new String('abc');

s.charAt(1) // "b"
s.charAt(s.length - 1) // "c"
```

这个方法完全可以用数组下标替代。

```
'abc'.charAt(1) // "b"
'abc'[1] // "b"
```

如果参数为负数，或大于等于字符串的长度，`charAt` 返回空字符串。

```
1.  'abc'.charAt(-1) // ""
2.  'abc'.charAt(3) // ""
```

String.prototype.charCodeAt()

`charCodeAt` 方法返回字符串指定位置的 Unicode 码点（十进制表示），相当于 `String.fromCharCode()` 的逆操作。

```
1.  'abc'.charCodeAt(1) // 98
```

上面代码中，`abc` 的 `1` 号位置的字符是 `b`，它的 Unicode 码点是 `98`。

如果没有任何参数，`charCodeAt` 返回首字符的 Unicode 码点。

```
1.  'abc'.charCodeAt() // 97
```

如果参数为负数，或大于等于字符串的长度，`charCodeAt` 返回 `NaN`。

```
1.  'abc'.charCodeAt(-1) // NaN
2.  'abc'.charCodeAt(4) // NaN
```

注意，`charCodeAt` 方法返回的 Unicode 码点不会大于 65536（0xFFFF），也就是说，只返回两个字节的字符的码点。如果遇到码点大于 65536 的字符（四个字节的字符），必需连续使用两次 `charCodeAt`，不仅读入 `charCodeAt(i)`，还要读入 `charCodeAt(i+1)`，将两个值放在一起，才能得到准确的字符。

String.prototype.concat()

`concat` 方法用于连接两个字符串，返回一个新字符串，不改变原字符串。

```
var s1 = 'abc';
var s2 = 'def';

s1.concat(s2) // "abcdef"
s1 // "abc"
```

该方法可以接受多个参数。

```
'a'.concat('b', 'c') // "abc"
```

如果参数不是字符串，`concat` 方法会将其先转为字符串，然后再连接。

```
var one = 1;
var two = 2;
var three = '3';

''.concat(one, two, three) // "123"
one + two + three // "33"
```

上面代码中，`concat` 方法将参数先转成字符串再连接，所以返回的是一个三个字符的字符串。作为对比，加号运算符在两个运算数都是数值时，不会转换类型，所以返回的是一个两个字符的字符串。

String.prototype.slice()

`slice` 方法用于从原字符串取出子字符串并返回，不改变原字符串。它的第一个参数是子字符串的开始位置，第二个参数是子字符串的结束位置（不含该位置）。

```
'JavaScript'.slice(0, 4) // "Java"
```

如果省略第二个参数，则表示子字符串一直到原字符串结束。

```
'JavaScript'.slice(4) // "Script"
```

如果参数是负值，表示从结尾开始倒数计算的位置，即该负值加上字符串长度。

```
'JavaScript'.slice(-6) // "Script"
'JavaScript'.slice(0, -6) // "Java"
'JavaScript'.slice(-2, -1) // "p"
```

如果第一个参数大于第二个参数，`slice` 方法返回一个空字符串。

```
'JavaScript'.slice(2, 1) // ""
```

String.prototype.substring()

`substring` 方法用于从原字符串取出子字符串并返回，不改变原字符串，跟 `slice` 方法很相像。它的第一个参数表示子字符串的开始位置，第二个位置表示结束位置（返回结果不含该位置）。

```
'JavaScript'.substring(0, 4) // "Java"
```

如果省略第二个参数，则表示子字符串一直到原字符串的结束。

```
'JavaScript'.substring(4) // "Script"
```

如果第二个参数大于第一个参数，`substring` 方法会自动更换两个参数的位置。

```
'JavaScript'.substring(10, 4) // "Script"
// 等同于
'JavaScript'.substring(4, 10) // "Script"
```

上面代码中，调换 `substring` 方法的两个参数，都得到同样的结果。

如果参数是负数，`substring` 方法会自动将负数转为0。

```
'Javascript'.substring(-3) // "JavaScript"
'JavaScript'.substring(4, -3) // "Java"
```

上面代码中，第二个例子的参数 `-3` 会自动变成 `0`，等同于 `'JavaScript'.substring(4, 0)`。由于第二个参数小于第一个参数，会自动互换位置，所以返回 `Java`。

由于这些规则违反直觉，因此不建议使用 `substring` 方法，应该优先使用 `slice`。

String.prototype.substr()

`substr` 方法用于从原字符串取出子字符串并返回，不改变原字符串，跟 `slice` 和 `substring` 方法的作用相同。

`substr` 方法的第一个参数是子字符串的开始位置（从0开始计算），第二个参数是子字符串的长度。

```
'JavaScript'.substr(4, 6) // "Script"
```

如果省略第二个参数，则表示子字符串一直到原字符串的结束。

```
'JavaScript'.substr(4) // "Script"
```

如果第一个参数是负数，表示倒数计算的字符位置。如果第二个参数是负数，将被自动转为0，因此会返回空字符串。

```
'JavaScript'.substr(-6) // "Script"
'JavaScript'.substr(4, -1) // ""
```

上面代码中，第二个例子的参数 `-1` 自动转为 `0`，表示子字符串长度

为 `0` ，所以返回空字符串。

String.prototype.indexOf(), String.prototype.lastIndexOf()

`indexOf` 方法用于确定一个字符串在另一个字符串中第一次出现的位置，返回结果是匹配开始的位置。如果返回 `-1` ，就表示不匹配。

```
1.  'hello world'.indexOf('o') // 4
2.  'JavaScript'.indexOf('script') // -1
```

`indexOf` 方法还可以接受第二个参数，表示从该位置开始向后匹配。

```
1.  'hello world'.indexOf('o', 6) // 7
```

`lastIndexOf` 方法的用法跟 `indexOf` 方法一致，主要的区别是 `lastIndexOf` 从尾部开始匹配， `indexOf` 则是从头部开始匹配。

```
1.  'hello world'.lastIndexOf('o') // 7
```

另外， `lastIndexOf` 的第二个参数表示从该位置起向前匹配。

```
1.  'hello world'.lastIndexOf('o', 6) // 4
```

String.prototype.trim()

`trim` 方法用于去除字符串两端的空格，返回一个新字符串，不改变原字符串。

```
1.  '  hello world  '.trim()
2.  // "hello world"
```

该方法去除的不仅是空格，还包括制表符（ `\t` 、 `\v` ）、换行符

（ `\n` ）和回车符（ `\r` ）。

```
'\r\nabc \t'.trim() // 'abc'
```

String.prototype.toLowerCase(), String.prototype.toUpperCase()

`toLowerCase` 方法用于将一个字符串全部转为小写， `toUpperCase` 则是全部转为大写。它们都返回一个新字符串，不改变原字符串。

```
'Hello World'.toLowerCase()
// "hello world"

'Hello World'.toUpperCase()
// "HELLO WORLD"
```

String.prototype.match()

`match` 方法用于确定原字符串是否匹配某个子字符串，返回一个数组，成员为匹配的第一个字符串。如果没有找到匹配，则返回 `null` 。

```
'cat, bat, sat, fat'.match('at') // ["at"]
'cat, bat, sat, fat'.match('xt') // null
```

返回的数组还有 `index` 属性和 `input` 属性，分别表示匹配字符串开始的位置和原始字符串。

```
var matches = 'cat, bat, sat, fat'.match('at');
matches.index // 1
matches.input // "cat, bat, sat, fat"
```

`match` 方法还可以使用正则表达式作为参数，详见《正则表达式》一章。

String.prototype.search(), String.prototype.replace()

`search` 方法的用法基本等同于 `match`，但是返回值为匹配的第一个位置。如果没有找到匹配，则返回 `-1`。

```
'cat, bat, sat, fat'.search('at') // 1
```

`search` 方法还可以使用正则表达式作为参数，详见《正则表达式》一节。

`replace` 方法用于替换匹配的子字符串，一般情况下只替换第一个匹配（除非使用带有 `g` 修饰符的正则表达式）。

```
'aaa'.replace('a', 'b') // "baa"
```

`replace` 方法还可以使用正则表达式作为参数，详见《正则表达式》一节。

String.prototype.split()

`split` 方法按照给定规则分割字符串，返回一个由分割出来的子字符串组成的数组。

```
'a|b|c'.split('|') // ["a", "b", "c"]
```

如果分割规则为空字符串，则返回数组的成员是原字符串的每一个字符。

```
'a|b|c'.split('') // ["a", "|", "b", "|", "c"]
```

如果省略参数，则返回数组的唯一成员就是原字符串。

```
'a|b|c'.split() // ["a|b|c"]
```

如果满足分割规则的两个部分紧邻着（即两个分割符中间没有其他字符），则返回数组之中会有一个空字符串。

```
'a||c'.split('|') // ['a', '', 'c']
```

如果满足分割规则的部分处于字符串的开头或结尾（即它的前面或后面没有其他字符），则返回数组的第一个或最后一个成员是一个空字符串。

```
'|b|c'.split('|') // ["", "b", "c"]
'a|b|'.split('|') // ["a", "b", ""]
```

`split` 方法还可以接受第二个参数，限定返回数组的最大成员数。

```
'a|b|c'.split('|', 0) // []
'a|b|c'.split('|', 1) // ["a"]
'a|b|c'.split('|', 2) // ["a", "b"]
'a|b|c'.split('|', 3) // ["a", "b", "c"]
'a|b|c'.split('|', 4) // ["a", "b", "c"]
```

上面代码中，`split` 方法的第二个参数，决定了返回数组的成员数。

`split` 方法还可以使用正则表达式作为参数，详见《正则表达式》一节。

String.prototype.localeCompare()

`localeCompare` 方法用于比较两个字符串。它返回一个整数，如果小于0，表示第一个字符串小于第二个字符串；如果等于0，表示两者相等；如果大于0，表示第一个字符串大于第二个字符串。

```
'apple'.localeCompare('banana') // -1
'apple'.localeCompare('apple') // 0
```

该方法的最大特点，就是会考虑自然语言的顺序。举例来说，正常情况下，大写的英文字母小于小写字母。

```
'B' > 'a' // false
```

上面代码中，字母 `B` 小于字母 `a` 。因为 JavaScript 采用的是 Unicode 码点比较， `B` 的码点是66，而 `a` 的码点是97。

但是， `localeCompare` 方法会考虑自然语言的排序情况，将 `B` 排在 `a` 的前面。

```
'B'.localeCompare('a') // 1
```

上面代码中， `localeCompare` 方法返回整数1，表示 `B` 较大。

`localeCompare` 还可以有第二个参数，指定所使用的语言（默认是英语），然后根据该语言的规则进行比较。

```
'ä'.localeCompare('z', 'de') // -1
'ä'.localeCompare('z', 'sv') // 1
```

上面代码中， `de` 表示德语， `sv` 表示瑞典语。德语中， `ä` 小于 `z` ，所以返回 `-1` ；瑞典语中， `ä` 大于 `z` ，所以返回 `1` 。

参考链接

- Ariya Hidayat, JavaScript String: substring, substr, slice

Math 对象

Math 对象

`Math` 是 JavaScript 的原生对象，提供各种数学功能。该对象不是构造函数，不能生成实例，所有的属性和方法都必须在 `Math` 对象上调用。

静态属性

`Math` 对象的静态属性，提供以下一些数学常数。

- `Math.E`：常数 `e`。
- `Math.LN2`：2 的自然对数。
- `Math.LN10`：10 的自然对数。

- `Math.LOG2E`：以 2 为底的`e`的对数。
- `Math.LOG10E`：以 10 为底的`e`的对数。
- `Math.PI`：常数 Pi。
- `Math.SQRT1_2`：0.5 的平方根。
- `Math.SQRT2`：2 的平方根。

```
Math.E // 2.718281828459045
Math.LN2 // 0.6931471805599453
Math.LN10 // 2.302585092994046
Math.LOG2E // 1.4426950408889634
Math.LOG10E // 0.4342944819032518
Math.PI // 3.141592653589793
Math.SQRT1_2 // 0.7071067811865476
Math.SQRT2 // 1.4142135623730951
```

这些属性都是只读的，不能修改。

静态方法

`Math`对象提供以下一些静态方法。

- `Math.abs()`：绝对值
- `Math.ceil()`：向上取整
- `Math.floor()`：向下取整
- `Math.max()`：最大值
- `Math.min()`：最小值
- `Math.pow()`：指数运算
- `Math.sqrt()`：平方根
- `Math.log()`：自然对数
- `Math.exp()`：e的指数
- `Math.round()`：四舍五入
- `Math.random()`：随机数

Math.abs()

`Math.abs` 方法返回参数值的绝对值。

```
Math.abs(1) // 1
Math.abs(-1) // 1
```

Math.max(), Math.min()

`Math.max` 方法返回参数之中最大的那个值，`Math.min` 返回最小的那个值。如果参数为空，`Math.min` 返回 `Infinity`，`Math.max` 返回 `-Infinity`。

```
Math.max(2, -1, 5) // 5
Math.min(2, -1, 5) // -1
Math.min() // Infinity
Math.max() // -Infinity
```

Math.floor(), Math.ceil()

`Math.floor` 方法小于参数值的最大整数（地板值）。

```
Math.floor(3.2) // 3
Math.floor(-3.2) // -4
```

`Math.ceil` 方法返回大于参数值的最小整数（天花板值）。

```
Math.ceil(3.2) // 4
Math.ceil(-3.2) // -3
```

这两个方法可以结合起来，实现一个总是返回数值的整数部分的函数。

```
function ToInteger(x) {
  x = Number(x);
```

```
  return x < 0 ? Math.ceil(x) : Math.floor(x);
}

ToInteger(3.2) // 3
ToInteger(3.5) // 3
ToInteger(3.8) // 3
ToInteger(-3.2) // -3
ToInteger(-3.5) // -3
ToInteger(-3.8) // -3
```

上面代码中，不管正数或负数，`ToInteger`函数总是返回一个数值的整数部分。

Math.round()

`Math.round`方法用于四舍五入。

```
Math.round(0.1) // 0
Math.round(0.5) // 1
Math.round(0.6) // 1

// 等同于
Math.floor(x + 0.5)
```

注意，它对负数的处理（主要是对`0.5`的处理）。

```
Math.round(-1.1) // -1
Math.round(-1.5) // -1
Math.round(-1.6) // -2
```

Math.pow()

`Math.pow`方法返回以第一个参数为底数、第二个参数为幂的指数值。

```
// 等同于 2 ** 2
```

```
Math.pow(2, 2) // 4
// 等同于 2 ** 3
Math.pow(2, 3) // 8
```

下面是计算圆面积的方法。

```
var radius = 20;
var area = Math.PI * Math.pow(radius, 2);
```

Math.sqrt()

`Math.sqrt` 方法返回参数值的平方根。如果参数是一个负值，则返回 `NaN` 。

```
Math.sqrt(4) // 2
Math.sqrt(-4) // NaN
```

Math.log()

`Math.log` 方法返回以 `e` 为底的自然对数值。

```
Math.log(Math.E) // 1
Math.log(10) // 2.302585092994046
```

如果要计算以10为底的对数，可以先用 `Math.log` 求出自然对数，然后除以 `Math.LN10` ；求以2为底的对数，可以除以 `Math.LN2` 。

```
Math.log(100)/Math.LN10 // 2
Math.log(8)/Math.LN2 // 3
```

Math.exp()

`Math.exp` 方法返回常数 `e` 的参数次方。

```
Math.exp(1) // 2.718281828459045
Math.exp(3) // 20.085536923187668
```

Math.random()

`Math.random()` 返回0到1之间的一个伪随机数，可能等于0，但是一定小于1。

```
Math.random() // 0.7151307314634323
```

任意范围的随机数生成函数如下。

```
function getRandomArbitrary(min, max) {
  return Math.random() * (max - min) + min;
}

getRandomArbitrary(1.5, 6.5)
// 2.4942810038223864
```

任意范围的随机整数生成函数如下。

```
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

getRandomInt(1, 6) // 5
```

返回随机字符的例子如下。

```
function random_str(length) {
  var ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  ALPHABET += 'abcdefghijklmnopqrstuvwxyz';
  ALPHABET += '0123456789-_';
  var str = '';
  for (var i=0; i < length; ++i) {
```

```
    var rand = Math.floor(Math.random() * ALPHABET.length);
    str += ALPHABET.substring(rand, rand + 1);
  }
  return str;
}

random_str(6) // "NdQKOr"
```

上面代码中，`random_str` 函数接受一个整数作为参数，返回变量 `ALPHABET` 内的随机字符所组成的指定长度的字符串。

三角函数方法

`Math` 对象还提供一系列三角函数方法。

- `Math.sin()`：返回参数的正弦（参数为弧度值）
- `Math.cos()`：返回参数的余弦（参数为弧度值）
- `Math.tan()`：返回参数的正切（参数为弧度值）
- `Math.asin()`：返回参数的反正弦（返回值为弧度值）
- `Math.acos()`：返回参数的反余弦（返回值为弧度值）
- `Math.atan()`：返回参数的反正切（返回值为弧度值）

```
Math.sin(0) // 0
Math.cos(0) // 1
Math.tan(0) // 0

Math.sin(Math.PI / 2) // 1

Math.asin(1) // 1.5707963267948966
Math.acos(1) // 0
Math.atan(1) // 0.7853981633974483
```

Date 对象

Date 对象

`Date`对象是 JavaScript 原生的时间库。它以1970年1月1日00:00:00作为时间的零点，可以表示的时间范围是前后各1亿天（单位为毫秒）。

普通函数的用法

`Date`对象可以作为普通函数直接调用，返回一个代表当前时间的字符串。

```
Date()
// "Tue Dec 01 2015 09:34:43 GMT+0800 (CST)"
```

注意，即使带有参数，`Date`作为普通函数使用时，返回的还是当前时间。

```
Date(2000, 1, 1)
// "Tue Dec 01 2015 09:34:43 GMT+0800 (CST)"
```

上面代码说明，无论有没有参数，直接调用`Date`总是返回当前时间。

构造函数的用法

`Date`还可以当作构造函数使用。对它使用`new`命令，会返回一个`Date`对象的实例。如果不加参数，实例代表的就是当前时间。

```
var today = new Date();
```

`Date`实例有一个独特的地方。其他对象求值的时候，都是默认调用`.valueOf()`方法，但是`Date`实例求值的时候，默认调用的是`toString()`方法。这导致对`Date`实例求值，返回的是一个字符串，代表该实例对应的时间。

```
var today = new Date();

today
// "Tue Dec 01 2015 09:34:43 GMT+0800 (CST)"

// 等同于
today.toString()
// "Tue Dec 01 2015 09:34:43 GMT+0800 (CST)"
```

上面代码中，`today`是`Date`的实例，直接求值等同于调用`toString`方法。

作为构造函数时，`Date`对象可以接受多种格式的参数，返回一个该参数对应的时间实例。

```
// 参数为时间零点开始计算的毫秒数
new Date(1378218728000)
// Tue Sep 03 2013 22:32:08 GMT+0800 (CST)

// 参数为日期字符串
new Date('January 6, 2013');
// Sun Jan 06 2013 00:00:00 GMT+0800 (CST)

// 参数为多个整数，
// 代表年、月、日、小时、分钟、秒、毫秒
new Date(2013, 0, 1, 0, 0, 0, 0)
// Tue Jan 01 2013 00:00:00 GMT+0800 (CST)
```

关于`Date`构造函数的参数，有几点说明。

第一点，参数可以是负整数，代表1970年元旦之前的时间。

```
new Date(-1378218728000)
// Fri Apr 30 1926 17:27:52 GMT+0800 (CST)
```

第二点，只要是能被`Date.parse()`方法解析的字符串，都可以当作参数。

```
new Date('2013-2-15')
new Date('2013/2/15')
new Date('02/15/2013')
new Date('2013-FEB-15')
new Date('FEB, 15, 2013')
new Date('FEB 15, 2013')
new Date('Feberuary, 15, 2013')
new Date('Feberuary 15, 2013')
new Date('15 Feb 2013')
new Date('15, Feberuary, 2013')
```

```
// Fri Feb 15 2013 00:00:00 GMT+0800 (CST)
```

上面多种日期字符串的写法，返回的都是同一个时间。

第三，参数为年、月、日等多个整数时，年和月是不能省略的，其他参数都可以省略的。也就是说，这时至少需要两个参数，因为如果只使用“年”这一个参数，`Date`会将其解释为毫秒数。

```
new Date(2013)
// Thu Jan 01 1970 08:00:02 GMT+0800 (CST)
```

上面代码中，2013被解释为毫秒数，而不是年份。

```
new Date(2013, 0)
// Tue Jan 01 2013 00:00:00 GMT+0800 (CST)
new Date(2013, 0, 1)
// Tue Jan 01 2013 00:00:00 GMT+0800 (CST)
new Date(2013, 0, 1, 0)
// Tue Jan 01 2013 00:00:00 GMT+0800 (CST)
new Date(2013, 0, 1, 0, 0, 0, 0)
// Tue Jan 01 2013 00:00:00 GMT+0800 (CST)
```

上面代码中，不管有几个参数，返回的都是2013年1月1日零点。

最后，各个参数的取值范围如下。

- 年：使用四位数年份，比如`2000`。如果写成两位数或个位数，则加上`1900`，即`10`代表1910年。如果是负数，表示公元前。
- 月：`0`表示一月，依次类推，`11`表示12月。
- 日：`1`到`31`。
- 小时：`0`到`23`。
- 分钟：`0`到`59`。
- 秒：`0`到`59`
- 毫秒：`0`到`999`。

注意，月份从 `0` 开始计算，但是，天数从 `1` 开始计算。另外，除了日期的默认值为 `1` ，小时、分钟、秒钟和毫秒的默认值都是 `0` 。

这些参数如果超出了正常范围，会被自动折算。比如，如果月设为 `15` ，就折算为下一年的4月。

```
new Date(2013, 15)
// Tue Apr 01 2014 00:00:00 GMT+0800 (CST)
new Date(2013, 0, 0)
// Mon Dec 31 2012 00:00:00 GMT+0800 (CST)
```

上面代码的第二个例子，日期设为 `0` ，就代表上个月的最后一天。

参数还可以使用负数，表示扣去的时间。

```
new Date(2013, -1)
// Sat Dec 01 2012 00:00:00 GMT+0800 (CST)
new Date(2013, 0, -1)
// Sun Dec 30 2012 00:00:00 GMT+0800 (CST)
```

上面代码中，分别对月和日使用了负数，表示从基准日扣去相应的时间。

日期的运算

类型自动转换时， `Date` 实例如果转为数值，则等于对应的毫秒数；如果转为字符串，则等于对应的日期字符串。所以，两个日期实例对象进行减法运算时，返回的是它们间隔的毫秒数；进行加法运算时，返回的是两个字符串连接而成的新字符串。

```
var d1 = new Date(2000, 2, 1);
var d2 = new Date(2000, 3, 1);

d2 - d1
```

```
// 2678400000
d2 + d1
// "Sat Apr 01 2000 00:00:00 GMT+0800 (CST)Wed Mar 01 2000 00:00:00 GMT+0800 (CST)"
```

静态方法

Date.now()

`Date.now` 方法返回当前时间距离时间零点（1970年1月1日 00:00:00 UTC）的毫秒数，相当于 Unix 时间戳乘以1000。

```
Date.now() // 1364026285194
```

Date.parse()

`Date.parse` 方法用来解析日期字符串，返回该时间距离时间零点（1970年1月1日 00:00:00）的毫秒数。

日期字符串应该符合 RFC 2822 和 ISO 8061 这两个标准，

即 `YYYY-MM-DDTHH:mm:ss.sssZ` 格式，其中最后的 `Z` 表示时区。但是，其他格式也可以被解析，请看下面的例子。

```
Date.parse('Aug 9, 1995')
Date.parse('January 26, 2011 13:51:50')
Date.parse('Mon, 25 Dec 1995 13:30:00 GMT')
Date.parse('Mon, 25 Dec 1995 13:30:00 +0430')
Date.parse('2011-10-10')
Date.parse('2011-10-10T14:48:00')
```

上面的日期字符串都可以解析。

如果解析失败，返回 `NaN` 。

```
Date.parse('xxx') // NaN
```

Date.UTC()

`Date.UTC` 方法接受年、月、日等变量作为参数，返回该时间距离时间零点（1970年1月1日 00:00:00 UTC）的毫秒数。

```
// 格式
Date.UTC(year, month[, date[, hrs[, min[, sec[, ms]]]]])

// 用法
Date.UTC(2011, 0, 1, 2, 3, 4, 567)
// 1293847384567
```

该方法的参数用法与 `Date` 构造函数完全一致，比如月从 `0` 开始计算，日期从 `1` 开始计算。区别在于 `Date.UTC` 方法的参数，会被解释为 UTC 时间（世界标准时间），`Date` 构造函数的参数会被解释为当前时区的时间。

实例方法

`Date` 的实例对象，有几十个自己的方法，除了 `valueOf` 和 `toString` ，可以分为以下三类。

- `to` 类：从 `Date` 对象返回一个字符串，表示指定的时间。
- `get` 类：获取 `Date` 对象的日期和时间。
- `set` 类：设置 `Date` 对象的日期和时间。

Date.prototype.valueOf()

`valueOf` 方法返回实例对象距离时间零点（1970年1月1日00:00:00 UTC）对应的毫秒数，该方法等同于 `getTime` 方法。

```
var d = new Date();

d.valueOf() // 1362790014817
d.getTime() // 1362790014817
```

预期为数值的场合，`Date` 实例会自动调用该方法，所以可以用下面的方法计算时间的间隔。

```
var start = new Date();
// ...
var end = new Date();
var elapsed = end - start;
```

to 类方法

（1）Date.prototype.toString()

`toString` 方法返回一个完整的日期字符串。

```
var d = new Date(2013, 0, 1);

d.toString()
// "Tue Jan 01 2013 00:00:00 GMT+0800 (CST)"
d
// "Tue Jan 01 2013 00:00:00 GMT+0800 (CST)"
```

因为 `toString` 是默认的调用方法，所以如果直接读取 `Date` 实例，就相当于调用这个方法。

（2）Date.prototype.toUTCString()

`toUTCString` 方法返回对应的 UTC 时间，也就是比北京时间晚8个小时。

```
var d = new Date(2013, 0, 1);
```

```
2.
3.  d.toUTCString()
4.  // "Mon, 31 Dec 2012 16:00:00 GMT"
```

（3）Date.prototype.toISOString()

`toISOString` 方法返回对应时间的 ISO8601 写法。

```
1.  var d = new Date(2013, 0, 1);
2.
3.  d.toISOString()
4.  // "2012-12-31T16:00:00.000Z"
```

注意，`toISOString` 方法返回的总是 UTC 时区的时间。

（4）Date.prototype.toJSON()

`toJSON` 方法返回一个符合 JSON 格式的 ISO 日期字符串，与 `toISOString` 方法的返回结果完全相同。

```
1.  var d = new Date(2013, 0, 1);
2.
3.  d.toJSON()
4.  // "2012-12-31T16:00:00.000Z"
```

（5）Date.prototype.toDateString()

`toDateString` 方法返回日期字符串（不含小时、分和秒）。

```
1.  var d = new Date(2013, 0, 1);
2.  d.toDateString() // "Tue Jan 01 2013"
```

（6）Date.prototype.toTimeString()

`toTimeString` 方法返回时间字符串（不含年月日）。

```
var d = new Date(2013, 0, 1);
d.toTimeString() // "00:00:00 GMT+0800 (CST)"
```

（7）Date.prototype.toLocaleDateString()

`toLocaleDateString` 方法返回一个字符串，代表日期的当地写法（不含小时、分和秒）。

```
var d = new Date(2013, 0, 1);

d.toLocaleDateString()
// 中文版浏览器为"2013年1月1日"
// 英文版浏览器为"1/1/2013"
```

（8）Date.prototype.toLocaleTimeString()

`toLocaleTimeString` 方法返回一个字符串，代表时间的当地写法（不含年月日）。

```
var d = new Date(2013, 0, 1);

d.toLocaleTimeString()
// 中文版浏览器为"上午12:00:00"
// 英文版浏览器为"12:00:00 AM"
```

get 类方法

`Date` 对象提供了一系列 `get*` 方法，用来获取实例对象某个方面的值。

- `getTime()`：返回实例距离1970年1月1日00:00:00的毫秒数，等同于`valueOf`方法。
- `getDate()`：返回实例对象对应每个月的几号（从1开始）。
- `getDay()`：返回星期几，星期日为0，星期一为1，以此类推。

- `getYear()`：返回距离1900的年数。
- `getFullYear()`：返回四位的年份。
- `getMonth()`：返回月份（0表示1月，11表示12月）。
- `getHours()`：返回小时（0-23）。
- `getMilliseconds()`：返回毫秒（0-999）。
- `getMinutes()`：返回分钟（0-59）。
- `getSeconds()`：返回秒（0-59）。
- `getTimezoneOffset()`：返回当前时间与 UTC 的时区差异，以分钟表示，返回结果考虑到了夏令时因素。

所有这些 `get*` 方法返回的都是整数，不同方法返回值的范围不一样。

- 分钟和秒：0 到 59
- 小时：0 到 23
- 星期：0（星期天）到 6（星期六）
- 日期：1 到 31
- 月份：0（一月）到 11（十二月）
- 年份：距离1900年的年数

```
var d = new Date('January 6, 2013');

d.getDate() // 6
d.getMonth() // 0
d.getYear() // 113
d.getFullYear() // 2013
d.getTimezoneOffset() // -480
```

上面代码中，最后一行返回 `-480`，即 UTC 时间减去当前时间，单位是分钟。`-480` 表示 UTC 比当前时间少480分钟，即当前时区比 UTC 早8个小时。

下面是一个例子，计算本年度还剩下多少天。

```
function leftDays() {
  var today = new Date();
  var endYear = new Date(today.getFullYear(), 11, 31, 23, 59, 59, 999);
  var msPerDay = 24 * 60 * 60 * 1000;
  return Math.round((endYear.getTime() - today.getTime()) / msPerDay);
}
```

上面这些 `get*` 方法返回的都是当前时区的时间，`Date` 对象还提供了这些方法对应的 UTC 版本，用来返回 UTC 时间。

- `getUTCDate()`
- `getUTCFullYear()`
- `getUTCMonth()`
- `getUTCDay()`
- `getUTCHours()`
- `getUTCMinutes()`
- `getUTCSeconds()`
- `getUTCMilliseconds()`

```
var d = new Date('January 6, 2013');

d.getDate() // 6
d.getUTCDate() // 5
```

上面代码中，实例对象 `d` 表示当前时区（东八时区）的1月6日0点0分0秒，这个时间对于当前时区来说是1月6日，所以 `getDate` 方法返回6，对于 UTC 时区来说是1月5日，所以 `getUTCDate` 方法返回5。

set 类方法

`Date` 对象提供了一系列 `set*` 方法，用来设置实例对象的各个方面。

- `setDate(date)`：设置实例对象对应的每个月的几号（1-31），返回改变后毫秒时间戳。
- `setYear(year)`：设置距离1900年的年数。
- `setFullYear(year [, month, date])`：设置四位年份。
- `setHours(hour [, min, sec, ms])`：设置小时（0-23）。
- `setMilliseconds()`：设置毫秒（0-999）。
- `setMinutes(min [, sec, ms])`：设置分钟（0-59）。
- `setMonth(month [, date])`：设置月份（0-11）。
- `setSeconds(sec [, ms])`：设置秒（0-59）。
- `setTime(milliseconds)`：设置毫秒时间戳。

这些方法基本是跟 `get*` 方法一一对应的，但是没有 `setDay` 方法，因为星期几是计算出来的，而不是设置的。另外，需要注意的是，凡是涉及到设置月份，都是从0开始算的，即 `0` 是1月，`11` 是12月。

```
var d = new Date ('January 6, 2013');

d // Sun Jan 06 2013 00:00:00 GMT+0800 (CST)
d.setDate(9) // 1357660800000
d // Wed Jan 09 2013 00:00:00 GMT+0800 (CST)
```

`set*` 方法的参数都会自动折算。以 `setDate` 为例，如果参数超过当月的最大天数，则向下一个月顺延，如果参数是负数，表示从上个月的最后一天开始减去的天数。

```
var d1 = new Date('January 6, 2013');

d1.setDate(32) // 1359648000000
d1 // Fri Feb 01 2013 00:00:00 GMT+0800 (CST)

var d2 = new Date ('January 6, 2013');

d.setDate(-1) // 1356796800000
```

```
d // Sun Dec 30 2012 00:00:00 GMT+0800 (CST)
```

`set` 类方法和 `get` 类方法，可以结合使用，得到相对时间。

```
var d = new Date();

// 将日期向后推1000天
d.setDate(d.getDate() + 1000);
// 将时间设为6小时后
d.setHours(d.getHours() + 6);
// 将年份设为去年
d.setFullYear(d.getFullYear() - 1);
```

`set*` 系列方法除了 `setTime()` 和 `setYear()`，都有对应的 UTC 版本，即设置 UTC 时区的时间。

- `setUTCDate()`
- `setUTCFullYear()`
- `setUTCHours()`
- `setUTCMilliseconds()`
- `setUTCMinutes()`
- `setUTCMonth()`
- `setUTCSeconds()`

```
var d = new Date('January 6, 2013');
d.getUTCHours() // 16
d.setUTCHours(22) // 1357423200000
d // Sun Jan 06 2013 06:00:00 GMT+0800 (CST)
```

上面代码中，本地时区（东八时区）的1月6日0点0分，是 UTC 时区的前一天下午16点。设为 UTC 时区的22点以后，就变为本地时区的上午6点。

参考链接

- Rakhitha Nimesh, Getting Started with the Date Object
- Ilya Kantor, Date/Time functions

RegExp 对象

RegExp 对象

`RegExp` 对象提供正则表示式的功能。

概述

正则表达式（regular expression）是一种表达文本模式（即字符串结构）的方法，有点像字符串的模板，常常用来按照“给定模式”匹配文本。比如，正则表达式给出一个 Email 地址的模式，然后用它来确定一个字符串是否为 Email 地址。JavaScript 的正则表达式体系是参照 Perl 5 建立的。

新建正则表达式有两种方法。一种是使用字面量，以斜杠表示开始和结束。

```
1. var regex = /xyz/;
```

另一种是使用 `RegExp` 构造函数。

```
1. var regex = new RegExp('xyz');
```

上面两种写法是等价的，都新建了一个内容为 `xyz` 的正则表达式对象。它们的主要区别是，第一种方法在引擎编译代码时，就会新建正则表达式，第二种方法在运行时新建正则表达式，所以前者的效率较高。而且，前者比较便利和直观，所以实际应用中，基本上都采用字面量定义正则表达式。

`RegExp` 构造函数还可以接受第二个参数，表示修饰符（详细解释见下文）。

```
1. var regex = new RegExp('xyz', 'i');
2. // 等价于
3. var regex = /xyz/i;
```

上面代码中，正则表达式 `/xyz/` 有一个修饰符 `i` 。

实例属性

正则对象的实例属性分成两类。

一类是修饰符相关，返回一个布尔值，表示对应的修饰符是否设置。

- **RegExp.prototype.ignoreCase**：返回一个布尔值，表示是否设置了`i`修饰符。
- **RegExp.prototype.global**：返回一个布尔值，表示是否设置了`g`修饰符。
- **RegExp.prototype.multiline**：返回一个布尔值，表示是否设置了`m`修饰符。

上面三个属性都是只读的。

```
var r = /abc/igm;

r.ignoreCase // true
r.global // true
r.multiline // true
```

另一类是与修饰符无关的属性，主要是下面两个。

- `RegExp.prototype.lastIndex`：返回一个数值，表示下一次开始搜索的位置。该属性可读写，但是只在设置了`g`修饰符、进行连续搜索时有意义，详细介绍请看后文。
- `RegExp.prototype.source`：返回正则表达式的字符串形式（不包括反斜杠），该属性只读。

```
var r = /abc/igm;

```

```
r.lastIndex // 0
r.source // "abc"
```

实例方法

RegExp.prototype.test()

正则实例对象的 `test` 方法返回一个布尔值，表示当前模式是否能匹配参数字符串。

```
/cat/.test('cats and dogs') // true
```

上面代码验证参数字符串之中是否包含 `cat` ，结果返回 `true` 。

如果正则表达式带有 `g` 修饰符，则每一次 `test` 方法都从上一次结束的位置开始向后匹配。

```
var r = /x/g;
var s = '_x_x';

r.lastIndex // 0
r.test(s) // true

r.lastIndex // 2
r.test(s) // true

r.lastIndex // 4
r.test(s) // false
```

上面代码的正则表达式使用了 `g` 修饰符，表示是全局搜索，会有多个结果。接着，三次使用 `test` 方法，每一次开始搜索的位置都是上一次匹配的后一个位置。

带有 `g` 修饰符时，可以通过正则对象的 `lastIndex` 属性指定开始搜索

的位置。

```
var r = /x/g;
var s = '_x_x';

r.lastIndex = 4;
r.test(s) // false
```

上面代码指定从字符串的第五个位置开始搜索，这个位置是没有字符的，所以返回 `false` 。

`lastIndex` 属性只对同一个正则表达式有效，所以下面这样写是错误的。

```
var count = 0;
while (/a/g.test('babaa')) count++;
```

上面代码会导致无限循环，因为 `while` 循环的每次匹配条件都是一个新的正则表达式，导致 `lastIndex` 属性总是等于0。

如果正则模式是一个空字符串，则匹配所有字符串。

```
new RegExp('').test('abc')
// true
```

RegExp.prototype.exec()

正则实例对象的 `exec` 方法，用来返回匹配结果。如果发现匹配，就返回一个数组，成员是匹配成功的子字符串，否则返回 `null` 。

```
var s = '_x_x';
var r1 = /x/;
var r2 = /y/;

```

```
r1.exec(s) // ["x"]
r2.exec(s) // null
```

上面代码中，正则对象 `r1` 匹配成功，返回一个数组，成员是匹配结果；正则对象 `r2` 匹配失败，返回 `null` 。

如果正则表示式包含圆括号（即含有“组匹配”），则返回的数组会包括多个成员。第一个成员是整个匹配成功的结果，后面的成员就是圆括号对应的匹配成功的组。也就是说，第二个成员对应第一个括号，第三个成员对应第二个括号，以此类推。整个数组的 `length` 属性等于组匹配的数量再加1。

```
var s = '_x_x';
var r = /_(x)/;

r.exec(s) // ["_x", "x"]
```

上面代码的 `exec` 方法，返回一个数组。第一个成员是整个匹配的结果，第二个成员是圆括号匹配的结果。

`exec` 方法的返回数组还包含以下两个属性：

- `input`：整个原字符串。
- `index`：整个模式匹配成功的开始位置（从0开始计数）。

```
var r = /a(b+)a/;
var arr = r.exec('_abbba_aba_');

arr // ["abbba", "bbb"]

arr.index // 1
arr.input // "_abbba_aba_"
```

上面代码中的 `index` 属性等于1，是因为从原字符串的第二个位置开始

匹配成功。

如果正则表达式加上 `g` 修饰符，则可以使用多次 `exec` 方法，下一次搜索的位置从上一次匹配成功结束的位置开始。

```
var reg = /a/g;
var str = 'abc_abc_abc'

var r1 = reg.exec(str);
r1 // ["a"]
r1.index // 0
reg.lastIndex // 1

var r2 = reg.exec(str);
r2 // ["a"]
r2.index // 4
reg.lastIndex // 5

var r3 = reg.exec(str);
r3 // ["a"]
r3.index // 8
reg.lastIndex // 9

var r4 = reg.exec(str);
r4 // null
reg.lastIndex // 0
```

上面代码连续用了四次 `exec` 方法，前三次都是从上一次匹配结束的位置向后匹配。当第三次匹配结束以后，整个字符串已经到达尾部，匹配结果返回 `null` ，正则实例对象的 `lastIndex` 属性也重置为 `0` ，意味着第四次匹配将从头开始。

利用 `g` 修饰符允许多次匹配的特点，可以用一个循环完成全部匹配。

```
var reg = /a/g;
var str = 'abc_abc_abc'
```

```
3.
4.  while(true) {
5.    var match = reg.exec(str);
6.    if (!match) break;
7.    console.log('#' + match.index + ':' + match[0]);
8.  }
9.  // #0:a
10. // #4:a
11. // #8:a
```

上面代码中，只要 `exec` 方法不返回 `null` ，就会一直循环下去，每次输出匹配的位置和匹配的文本。

正则实例对象的 `lastIndex` 属性不仅可读，还可写。设置了 `g` 修饰符的时候，只要手动设置了 `lastIndex` 的值，就会从指定位置开始匹配。

字符串的实例方法

字符串的实例方法之中，有4种与正则表达式有关。

- `String.prototype.match()` ：返回一个数组，成员是所有匹配的子字符串。
- `String.prototype.search()` ：按照给定的正则表达式进行搜索，返回一个整数，表示匹配开始的位置。
- `String.prototype.replace()` ：按照给定的正则表达式进行替换，返回替换后的字符串。
- `String.prototype.split()` ：按照给定规则进行字符串分割，返回一个数组，包含分割后的各个成员。

String.prototype.match()

字符串实例对象的 `match` 方法对字符串进行正则匹配，返回匹配结果。

```
var s = '_x_x';
var r1 = /x/;
var r2 = /y/;

s.match(r1) // ["x"]
s.match(r2) // null
```

从上面代码可以看到，字符串的 `match` 方法与正则对象的 `exec` 方法非常类似：匹配成功返回一个数组，匹配失败返回 `null` 。

如果正则表达式带有 `g` 修饰符，则该方法与正则对象的 `exec` 方法行为不同，会一次性返回所有匹配成功的结果。

```
var s = 'abba';
var r = /a/g;

s.match(r) // ["a", "a"]
r.exec(s) // ["a"]
```

设置正则表达式的 `lastIndex` 属性，对 `match` 方法无效，匹配总是从字符串的第一个字符开始。

```
var r = /a|b/g;
r.lastIndex = 7;
'xaxb'.match(r) // ['a', 'b']
r.lastIndex // 0
```

上面代码表示，设置正则对象的 `lastIndex` 属性是无效的。

String.prototype.search()

字符串对象的 `search` 方法，返回第一个满足条件的匹配结果在整个字符串中的位置。如果没有任何匹配，则返回 `-1` 。

```
'_x_x'.search(/x/)
// 1
```

上面代码中，第一个匹配结果出现在字符串的 `1` 号位置。

String.prototype.replace()

字符串对象的 `replace` 方法可以替换匹配的值。它接受两个参数，第一个是正则表达式，表示搜索模式，第二个是替换的内容。

```
str.replace(search, replacement)
```

正则表达式如果不加 `g` 修饰符，就替换第一个匹配成功的值，否则替换所有匹配成功的值。

```
'aaa'.replace('a', 'b') // "baa"
'aaa'.replace(/a/, 'b') // "baa"
'aaa'.replace(/a/g, 'b') // "bbb"
```

上面代码中，最后一个正则表达式使用了 `g` 修饰符，导致所有的 `b` 都被替换掉了。

`replace` 方法的一个应用，就是消除字符串首尾两端的空格。

```
var str = '  #id div.class  ';

str.replace(/^\s+|\s+$/g, '')
// "#id div.class"
```

`replace` 方法的第二个参数可以使用美元符号 `$` ，用来指代所替换的内容。

- $&：匹配的子字符串。
- $`：匹配结果前面的文本。

- $’：匹配结果后面的文本。
- $n：匹配成功的第 `n` 组内容，`n` 是从1开始的自然数。
- $$：指代美元符号 `$`。

```
'hello world'.replace(/(\w+)\s(\w+)/, '$2 $1')
// "world hello"

'abc'.replace('b', '[$`-$&-$\']')
// "a[a-b-c]c"
```

上面代码中，第一个例子是将匹配的组互换位置，第二个例子是改写匹配的值。

`replace` 方法的第二个参数还可以是一个函数，将每一个匹配内容替换为函数返回值。

```
'3 and 5'.replace(/[0-9]+/g, function (match) {
  return 2 * match;
})
// "6 and 10"

var a = 'The quick brown fox jumped over the lazy dog.';
var pattern = /quick|brown|lazy/ig;

a.replace(pattern, function replacer(match) {
  return match.toUpperCase();
});
// The QUICK BROWN fox jumped over the LAZY dog.
```

作为 `replace` 方法第二个参数的替换函数，可以接受多个参数。其中，第一个参数是捕捉到的内容，第二个参数是捕捉到的组匹配（有多少个组匹配，就有多少个对应的参数）。此外，最后还可以添加两个参数，倒数第二个参数是捕捉到的内容在整个字符串中的位置（比如从第五个位置开始），最后一个参数是原字符串。下面是一个网页模板替换的例

子。

```
var prices = {
  'p1': '$1.99',
  'p2': '$9.99',
  'p3': '$5.00'
};

var template = '<span id="p1"></span>'
  + '<span id="p2"></span>'
  + '<span id="p3"></span>';

template.replace(
  /(<span id=")(.*?)(">)(<\/span>)/g,
  function(match, $1, $2, $3, $4){
    return $1 + $2 + $3 + prices[$2] + $4;
  }
);
// "<span id="p1">$1.99</span><span id="p2">$9.99</span><span id="p3">$5.00</span>"
```

上面代码的捕捉模式中，有四个括号，所以会产生四个组匹配，在匹配函数中用 `$1` 到 `$4` 表示。匹配函数的作用是将价格插入模板中。

String.prototype.split()

字符串对象的 `split` 方法按照正则规则分割字符串，返回一个由分割后的各个部分组成的数组。

```
str.split(separator, [limit])
```

该方法接受两个参数，第一个参数是正则表达式，表示分隔规则，第二个参数是返回数组的最大成员数。

```
// 非正则分隔
```

```
'a,  b,c, d'.split(',')
// [ 'a', '  b', 'c', ' d' ]

// 正则分隔，去除多余的空格
'a,  b,c, d'.split(/, */)
// [ 'a', 'b', 'c', 'd' ]

// 指定返回数组的最大成员
'a,  b,c, d'.split(/, */, 2)
[ 'a', 'b' ]
```

上面代码使用正则表达式，去除了子字符串的逗号后面的空格。

```
// 例一
'aaa*a*'.split(/a*/)
// [ '', '*', '*' ]

// 例二
'aaa**a*'.split(/a*/)
// ["", "*", "*", "*"]
```

上面代码的分割规则是0次或多次的 `a` ，由于正则默认是贪婪匹配，所以例一的第一个分隔符是 `aaa` ，第二个分割符是 `a` ，将字符串分成三个部分，包含开始处的空字符串。例二的第一个分隔符是 `aaa` ，第二个分隔符是0个 `a` （即空字符），第三个分隔符是 `a` ，所以将字符串分成四个部分。

如果正则表达式带有括号，则括号匹配的部分也会作为数组成员返回。

```
'aaa*a*'.split(/(a*)/)
// [ '', 'aaa', '*', 'a', '*' ]
```

上面代码的正则表达式使用了括号，第一个组匹配是 `aaa` ，第二个组匹配是 `a` ，它们都作为数组成员返回。

匹配规则

正则表达式的规则很复杂，下面一一介绍这些规则。

字面量字符和元字符

大部分字符在正则表达式中，就是字面的含义，比如 `/a/` 匹配 `a` ， `/b/` 匹配 `b` 。如果在正则表达式之中，某个字符只表示它字面的含义（就像前面的 `a` 和 `b` ），那么它们就叫做“字面量字符”（literal characters）。

```
1. /dog/.test('old dog') // true
```

上面代码中正则表达式的 `dog` ，就是字面量字符，所以 `/dog/` 匹配 `old dog` ，因为它就表示 `d` 、 `o` 、 `g` 三个字母连在一起。

除了字面量字符以外，还有一部分字符有特殊含义，不代表字面的意思。它们叫做“元字符”（metacharacters），主要有以下几个。

（1）点字符（.）

点字符（ `.` ）匹配除回车（ `\r` ）、换行(`\n`) 、行分隔符（ `\u2028` ）和段分隔符（ `\u2029` ）以外的所有字符。

```
1. /c.t/
```

上面代码中， `c.t` 匹配 `c` 和 `t` 之间包含任意一个字符的情况，只要这三个字符在同一行，比如 `cat` 、 `c2t` 、 `c-t` 等等，但是不匹配 `coot` 。

（2）位置字符

位置字符用来提示字符所处的位置，主要有两个字符。

- `^` 表示字符串的开始位置
- `$` 表示字符串的结束位置

```
// test必须出现在开始位置
/^test/.test('test123') // true

// test必须出现在结束位置
/test$/.test('new test') // true

// 从开始位置到结束位置只有test
/^test$/.test('test') // true
/^test$/.test('test test') // false
```

（3）选择符（ `|` ）

竖线符号（ `|` ）在正则表达式中表示“或关系”（OR），即 `cat|dog` 表示匹配 `cat` 或 `dog` 。

```
/11|22/.test('911') // true
```

上面代码中，正则表达式指定必须匹配 `11` 或 `22` 。

多个选择符可以联合使用。

```
// 匹配fred、barney、betty之中的一个
/fred|barney|betty/
```

选择符会包括它前后的多个字符，比如 `/ab|cd/` 指的是匹配 `ab` 或者 `cd` ，而不是指匹配 `b` 或者 `c` 。如果想修改这个行为，可以使用圆括号。

```
/a( |\t)b/.test('a\tb') // true
```

上面代码指的是， `a` 和 `b` 之间有一个空格或者一个制表符。

其他的元字符还包括 `\`、`\*`、`+`、`?`、`()`、`[]`、`{}` 等，将在下文解释。

转义符

正则表达式中那些有特殊含义的元字符，如果要匹配它们本身，就需要在它们前面要加上反斜杠。比如要匹配 `+`，就要写成 `\+`。

```
/1+1/.test('1+1')
// false

/1\+1/.test('1+1')
// true
```

上面代码中，第一个正则表达式之所以不匹配，因为加号是元字符，不代表自身。第二个正则表达式使用反斜杠对加号转义，就能匹配成功。

正则表达式中，需要反斜杠转义的，一共有12个字符：`^`、`.`、`[`、`$`、`(`、`)`、`|`、`*`、`+`、`?`、`{` 和 `\`。需要特别注意的是，如果使用 `RegExp` 方法生成正则对象，转义需要使用两个斜杠，因为字符串内部会先转义一次。

```
(new RegExp('1\+1')).test('1+1')
// false

(new RegExp('1\\+1')).test('1+1')
// true
```

上面代码中，`RegExp` 作为构造函数，参数是一个字符串。但是，在字符串内部，反斜杠也是转义字符，所以它会先被反斜杠转义一次，然后再被正则表达式转义一次，因此需要两个反斜杠转义。

特殊字符

正则表达式对一些不能打印的特殊字符，提供了表达方法。

- `\cX` 表示`Ctrl-[X]`，其中的`X`是A-Z之中任一个英文字母，用来匹配控制字符。
- `[\b]` 匹配退格键(U+0008)，不要与`\b`混淆。
- `\n` 匹配换行键。
- `\r` 匹配回车键。
- `\t` 匹配制表符 tab（U+0009）。
- `\v` 匹配垂直制表符（U+000B）。
- `\f` 匹配换页符（U+000C）。
- `\0` 匹配`null`字符（U+0000）。
- `\xhh` 匹配一个以两位十六进制数（`\x00`-`\xFF`）表示的字符。
- `\uhhhh` 匹配一个以四位十六进制数（`\u0000`-`\uFFFF`）表示的 Unicode 字符。

字符类

字符类（class）表示有一系列字符可供选择，只要匹配其中一个就可以了。所有可供选择的字符都放在方括号内，比如`[xyz]` 表示`x`、`y`、`z`之中任选一个匹配。

```
1. /[abc]/.test('hello world') // false
2. /[abc]/.test('apple') // true
```

上面代码中，字符串`hello world`不包含`a`、`b`、`c`这三个字母中的任一个，所以返回`false`；字符串`apple`包含字母`a`，所以返回`true`。

有两个字符在字符类中有特殊含义。

（1）脱字符（^）

如果方括号内的第一个字符是 `[^]` ，则表示除了字符类之中的字符，其他字符都可以匹配。比如， `[^xyz]` 表示除了 `x` 、 `y` 、 `z` 之外都可以匹配。

```
/[^abc]/.test('hello world') // true
/[^abc]/.test('bbc') // false
```

上面代码中，字符串 `hello world` 不包含字母 `a` 、 `b` 、 `c` 中的任一个，所以返回 `true` ；字符串 `bbc` 不包含 `a` 、 `b` 、 `c` 以外的字母，所以返回 `false` 。

如果方括号内没有其他字符，即只有 `[^]` ，就表示匹配一切字符，其中包括换行符。相比之下，点号作为元字符（ `.` ）是不包括换行符的。

```
var s = 'Please yes\nmake my day!';

s.match(/yes.*day/) // null
s.match(/yes[^]*day/) // [ 'yes\nmake my day']
```

上面代码中，字符串 `s` 含有一个换行符，点号不包括换行符，所以第一个正则表达式匹配失败；第二个正则表达式 `[^]` 包含一切字符，所以匹配成功。

> 注意，脱字符只有在字符类的第一个位置才有特殊含义，否则就是字面含义。

（2）连字符（-）

某些情况下，对于连续序列的字符，连字符（ `-` ）用来提供简写形式，表示字符的连续范围。比如， `[abc]` 可以写成 `[a-c]` ， `[0123456789]` 可以写成 `[0-9]` ，同理 `[A-Z]` 表示26个大写字母。

```
/a-z/.test('b') // false
/[a-z]/.test('b') // true
```

上面代码中，当连字号（dash）不出现在方括号之中，就不具备简写的作用，只代表字面的含义，所以不匹配字符 `b`。只有当连字号用在方括号之中，才表示连续的字符序列。

以下都是合法的字符类简写形式。

```
[0-9.,]
[0-9a-fA-F]
[a-zA-Z0-9-]
[1-31]
```

上面代码中最后一个字符类 `[1-31]`，不代表 `1` 到 `31`，只代表 `1` 到 `3`。

连字符还可以用来指定 Unicode 字符的范围。

```
var str = "\u0130\u0131\u0132";
/[\u0128-\uFFFF]/.test(str)
// true
```

上面代码中，`\u0128-\uFFFF` 表示匹配码点在 `0128` 到 `FFFF` 之间的所有字符。

另外，不要过分使用连字符，设定一个很大的范围，否则很可能选中意料之外的字符。最典型的例子就是 `[A-z]`，表面上它是选中从大写的 `A` 到小写的 `z` 之间52个字母，但是由于在 ASCII 编码之中，大写字母与小写字母之间还有其他字符，结果就会出现意料之外的结果。

```
/[A-z]/.test('\\') // true
```

上面代码中，由于反斜杠（'\'）的ASCII码在大写字母与小写字母之间，结果会被选中。

预定义模式

预定义模式指的是某些常见模式的简写方式。

- `\d` 匹配0-9之间的任一数字，相当于`[0-9]`。
- `\D` 匹配所有0-9以外的字符，相当于`[^0-9]`。
- `\w` 匹配任意的字母、数字和下划线，相当于`[A-Za-z0-9_]`。
- `\W` 除所有字母、数字和下划线以外的字符，相当于`[^A-Za-z0-9_]`。
- `\s` 匹配空格（包括换行符、制表符、空格符等），相等于`[\t\r\n\v\f]`。
- `\S` 匹配非空格的字符，相当于`[^\t\r\n\v\f]`。
- `\b` 匹配词的边界。
- `\B` 匹配非词边界，即在词的内部。

下面是一些例子。

```
// \s 的例子
/\s\w*/.exec('hello world') // [" world"]

// \b 的例子
/\bworld/.test('hello world') // true
/\bworld/.test('hello-world') // true
/\bworld/.test('helloworld') // false

// \B 的例子
/\Bworld/.test('hello-world') // false
/\Bworld/.test('helloworld') // true
```

上面代码中，`\s`表示空格，所以匹配结果会包括空格。`\b`表示词

的边界，所以 `world` 的词首必须独立（词尾是否独立未指定），才会匹配。同理， `\B` 表示非词的边界，只有 `world` 的词首不独立，才会匹配。

通常，正则表达式遇到换行符（ `\n` ）就会停止匹配。

```
var html = "<b>Hello</b>\n<i>world!</i>";

/.*/.exec(html)[0]
// "<b>Hello</b>"
```

上面代码中，字符串 `html` 包含一个换行符，结果点字符（ `.` ）不匹配换行符，导致匹配结果可能不符合原意。这时使用 `\s` 字符类，就能包括换行符。

```
var html = "<b>Hello</b>\n<i>world!</i>";

/[\S\s]*/.exec(html)[0]
// "<b>Hello</b>\n<i>world!</i>"
```

上面代码中， `[\S\s]` 指代一切字符。

重复类

模式的精确匹配次数，使用大括号（ `{}` ）表示。 `{n}` 表示恰好重复 `n` 次， `{n,}` 表示至少重复 `n` 次， `{n,m}` 表示重复不少于 `n` 次，不多于 `m` 次。

```
/lo{2}k/.test('look') // true
/lo{2,5}k/.test('looook') // true
```

上面代码中，第一个模式指定 `o` 连续出现2次，第二个模式指定 `o` 连续出现2次到5次之间。

量词符

量词符用来设定某个模式出现的次数。

- `?` 问号表示某个模式出现0次或1次，等同于 `{0, 1}`。
- `*` 星号表示某个模式出现0次或多次，等同于 `{0,}`。
- `+` 加号表示某个模式出现1次或多次，等同于 `{1,}`。

```
// t 出现0次或1次
/t?est/.test('test') // true
/t?est/.test('est') // true

// t 出现1次或多次
/t+est/.test('test') // true
/t+est/.test('ttest') // true
/t+est/.test('est') // false

// t 出现0次或多次
/t*est/.test('test') // true
/t*est/.test('ttest') // true
/t*est/.test('tttest') // true
/t*est/.test('est') // true
```

贪婪模式

上一小节的三个量词符，默认情况下都是最大可能匹配，即匹配直到下一个字符不满足匹配规则为止。这被称为贪婪模式。

```
var s = 'aaa';
s.match(/a+/) // ["aaa"]
```

上面代码中，模式是 `/a+/`，表示匹配1个 `a` 或多个 `a`，那么到底会匹配几个 `a` 呢？因为默认是贪婪模式，会一直匹配到字符 `a` 不出现为止，所以匹配结果是3个 `a`。

如果想将贪婪模式改为非贪婪模式，可以在量词符后面加一个问号。

```
var s = 'aaa';
s.match(/a+?/) // ["a"]
```

上面代码中，模式结尾添加了一个问号 `/a+?/` ，这时就改为非贪婪模式，一旦条件满足，就不再往下匹配。

除了非贪婪模式的加号，还有非贪婪模式的星号（ `*` ）。

- `*?`：表示某个模式出现0次或多次，匹配时采用非贪婪模式。
- `+?`：表示某个模式出现1次或多次，匹配时采用非贪婪模式。

修饰符

修饰符（modifier）表示模式的附加规则，放在正则模式的最尾部。

修饰符可以单个使用，也可以多个一起使用。

```
// 单个修饰符
var regex = /test/i;

// 多个修饰符
var regex = /test/ig;
```

（1）g 修饰符

默认情况下，第一次匹配成功后，正则对象就停止向下匹配了。 `g` 修饰符表示全局匹配（global），加上它以后，正则对象将匹配全部符合条件的结果，主要用于搜索和替换。

```
var regex = /b/;
var str = 'abba';

regex.test(str); // true
```

```
regex.test(str); // true
regex.test(str); // true
```

上面代码中，正则模式不含 `g` 修饰符，每次都是从字符串头部开始匹配。所以，连续做了三次匹配，都返回 `true` 。

```
var regex = /b/g;
var str = 'abba';

regex.test(str); // true
regex.test(str); // true
regex.test(str); // false
```

上面代码中，正则模式含有 `g` 修饰符，每次都是从上一次匹配成功处，开始向后匹配。因为字符串 `abba` 只有两个 `b` ，所以前两次匹配结果为 `true` ，第三次匹配结果为 `false` 。

（2）i 修饰符

默认情况下，正则对象区分字母的大小写，加上 `i` 修饰符以后表示忽略大小写（ignorecase）。

```
/abc/.test('ABC') // false
/abc/i.test('ABC') // true
```

上面代码表示，加了 `i` 修饰符以后，不考虑大小写，所以模式 `abc` 匹配字符串 `ABC` 。

（3）m 修饰符

`m` 修饰符表示多行模式（multiline），会修改 `^` 和 `$` 的行为。默认情况下（即不加 `m` 修饰符时）， `^` 和 `$` 匹配字符串的开始处和结尾处，加上 `m` 修饰符以后， `^` 和 `$` 还会匹配行首和行尾，即 `^` 和 `$` 会识别换行符（ `\n` ）。

```
/world$/.test('hello world\n') // false
/world$/m.test('hello world\n') // true
```

上面的代码中，字符串结尾处有一个换行符。如果不加 `m` 修饰符，匹配不成功，因为字符串的结尾不是 `world`；加上以后， `$` 可以匹配行尾。

```
/^b/m.test('a\nb') // true
```

上面代码要求匹配行首的 `b`，如果不加 `m` 修饰符，就相当于 `b` 只能处在字符串的开始处。加上 `b` 修饰符以后，换行符 `\n` 也会被认为是一行的开始。

组匹配

（1）概述

正则表达式的括号表示分组匹配，括号中的模式可以用来匹配分组的内容。

```
/fred+/.test('fredd') // true
/(fred)+/.test('fredfred') // true
```

上面代码中，第一个模式没有括号，结果 `+` 只表示重复字母 `d`，第二个模式有括号，结果 `+` 就表示匹配 `fred` 这个词。

下面是另外一个分组捕获的例子。

```
var m = 'abcabc'.match(/(.)b(.)/);
m
// ['abc', 'a', 'c']
```

上面代码中，正则表达式 `/(.)b(.)/` 一共使用两个括号，第一个括号捕

获 `a` ，第二个括号捕获 `c` 。

注意，使用组匹配时，不宜同时使用 `g` 修饰符，否则 `match` 方法不会捕获分组的内容。

```
var m = 'abcabc'.match(/(.)b(.)/g);
m // ['abc', 'abc']
```

上面代码使用带 `g` 修饰符的正则表达式，结果 `match` 方法只捕获了匹配整个表达式的部分。这时必须使用正则表达式的 `exec` 方法，配合循环，才能读到每一轮匹配的组捕获。

```
var str = 'abcabc';
var reg = /(.)b(.)/g;
while (true) {
  var result = reg.exec(str);
  if (!result) break;
  console.log(result);
}
// ["abc", "a", "c"]
// ["abc", "a", "c"]
```

正则表达式内部，还可以用 `\n` 引用括号匹配的内容， `n` 是从1开始的自然数，表示对应顺序的括号。

```
/(.)b(.)\1b\2/.test("abcabc")
// true
```

上面的代码中， `\1` 表示第一个括号匹配的内容（即 `a` ）， `\2` 表示第二个括号匹配的内容（即 `c` ）。

下面是另外一个例子。

```
/y(..)(.)\2\1/.test('yabccab') // true
```

括号还可以嵌套。

```
/y((..)\2)\1/.test('yabababab') // true
```

上面代码中，`\1` 指向外层括号，`\2` 指向内层括号。

组匹配非常有用，下面是一个匹配网页标签的例子。

```
var tagName = /<([^>]+)>[^<]*<\/\1>/;

tagName.exec("<b>bold</b>")[1]
// 'b'
```

上面代码中，圆括号匹配尖括号之中的标签，而 `\1` 就表示对应的闭合标签。

上面代码略加修改，就能捕获带有属性的标签。

```
var html = '<b class="hello">Hello</b><i>world</i>';
var tag = /<(\w+)([^>]*)>(.*?)<\/\1>/g;

var match = tag.exec(html);

match[1] // "b"
match[2] // "class="hello""
match[3] // "Hello"

match = tag.exec(html);

match[1] // "i"
match[2] // ""
match[3] // "world"
```

（2）非捕获组

`(?:x)` 称为非捕获组（Non-capturing group），表示不返回该组

匹配的内容，即匹配的结果中不计入这个括号。

非捕获组的作用请考虑这样一个场景，假定需要匹配 `foo` 或者 `foofoo` ，正则表达式就应该写成 `/(foo){1, 2}/` ，但是这样会占用一个组匹配。这时，就可以使用非捕获组，将正则表达式改为 `/(?:foo){1, 2}/` ，它的作用与前一个正则是一样的，但是不会单独输出括号内部的内容。

请看下面的例子。

```
var m = 'abc'.match(/(?:.)b(.)/);
m // ["abc", "c"]
```

上面代码中的模式，一共使用了两个括号。其中第一个括号是非捕获组，所以最后返回的结果中没有第一个括号，只有第二个括号匹配的内容。

下面是用来分解网址的正则表达式。

```
// 正常匹配
var url = /(http|ftp):\/\/([^/\r\n]+)(\/[^\r\n]*)?/;

url.exec('http://google.com/');
// ["http://google.com/", "http", "google.com", "/"]

// 非捕获组匹配
var url = /(?:http|ftp):\/\/([^/\r\n]+)(\/[^\r\n]*)?/;

url.exec('http://google.com/');
// ["http://google.com/", "google.com", "/"]
```

上面的代码中，前一个正则表达式是正常匹配，第一个括号返回网络协议；后一个正则表达式是非捕获匹配，返回结果中不包括网络协议。

（3）先行断言

`x(?=y)` 称为先行断言（Positive look-ahead），`x` 只有在 `y` 前面才匹配，`y` 不会被计入返回结果。比如，要匹配后面跟着百分号的数字，可以写成 `/\d+(?=%)/` 。

“先行断言”中，括号里的部分是不会返回的。

```
1. var m = 'abc'.match(/b(?=c)/);
2. m // ["b"]
```

上面的代码使用了先行断言，`b` 在 `c` 前面所以被匹配，但是括号对应的 `c` 不会被返回。

（4）先行否定断言

`x(?!y)` 称为先行否定断言（Negative look-ahead），`x` 只有不在 `y` 前面才匹配，`y` 不会被计入返回结果。比如，要匹配后面跟的不是百分号的数字，就要写成 `/\d+(?!%)/` 。

```
1. /\d+(?!\.)/.exec('3.14')
2. // ["14"]
```

上面代码中，正则表达式指定，只有不在小数点前面的数字才会被匹配，因此返回的结果就是 `14` 。

“先行否定断言”中，括号里的部分是不会返回的。

```
1. var m = 'abd'.match(/b(?!c)/);
2. m // ['b']
```

上面的代码使用了先行否定断言，`b` 不在 `c` 前面所以被匹配，而且括号对应的 `d` 不会被返回。

参考链接

- Axel Rauschmayer, JavaScript: an overview of the regular expression API
- Mozilla Developer Network, Regular Expressions
- Axel Rauschmayer, The flag /g of JavaScript’s regular expressions
- Sam Hughes, Learn regular expressions in about 55 minutes

JSON 对象

JSON 对象

JSON 格式

JSON 格式（JavaScript Object Notation 的缩写）是一种用于数据交换的文本格式，2001年由 Douglas Crockford 提出，目的是取代繁琐笨重的 XML 格式。

相比 XML 格式，JSON 格式有两个显著的优点：书写简单，一目了然；符合 JavaScript 原生语法，可以由解释引擎直接处理，不用另外添加解析代码。所以，JSON 迅速被接受，已经成为各大网站交换数据的标准格式，并被写入标准。

每个 JSON 对象就是一个值，可能是一个数组或对象，也可能是一个原始类型的值。总之，只能是一个值，不能是两个或更多的值。

JSON 对值的类型和格式有严格的规定。

> 1. 复合类型的值只能是数组或对象，不能是函数、正则表达式对象、日期对象。
> 2. 原始类型的值只有四种：字符串、数值（必须以十进制表示）、布尔值和 `null`（不能使用 `NaN`，`Infinity`，`-Infinity` 和 `undefined`）。
> 3. 字符串必须使用双引号表示，不能使用单引号。
> 4. 对象的键名必须放在双引号里面。
> 5. 数组或对象最后一个成员的后面，不能加逗号。

以下都是合法的 JSON。

```
["one", "two", "three"]

{ "one": 1, "two": 2, "three": 3 }

{"names": ["张三", "李四"] }

[ { "name": "张三"}, {"name": "李四"} ]
```

以下都是不合法的 JSON。

```
{ name: "张三", 'age': 32 }  // 属性名必须使用双引号

[32, 64, 128, 0xFFF] // 不能使用十六进制值

{ "name": "张三", "age": undefined } // 不能使用 undefined

{ "name": "张三",
  "birthday": new Date('Fri, 26 Aug 2011 07:13:10 GMT'),
  "getName": function () {
      return this.name;
  }
} // 属性值不能使用函数和日期对象
```

注意，`null`、空数组和空对象都是合法的 JSON 值。

JSON 对象

`JSON` 对象是 JavaScript 的原生对象，用来处理 JSON 格式数据。它有两个静态方法：`JSON.stringify()` 和 `JSON.parse()` 。

JSON.stringify()

基本用法

`JSON.stringify` 方法用于将一个值转为 JSON 字符串。该字符串符合 JSON 格式，并且可以被 `JSON.parse` 方法还原。

```
JSON.stringify('abc') // ""abc""
JSON.stringify(1) // "1"
JSON.stringify(false) // "false"
JSON.stringify([]) // "[]"
JSON.stringify({}) // "{}"

JSON.stringify([1, "false", false])
// '[1,"false",false]'

JSON.stringify({ name: "张三" })
// '{"name":"张三"}'
```

上面代码将各种类型的值，转成 JSON 字符串。

注意，对于原始类型的字符串，转换结果会带双引号。

```
JSON.stringify('foo') === "foo" // false
JSON.stringify('foo') === "\"foo\"" // true
```

上面代码中，字符串 `foo` ，被转成了 `"\"foo\""` 。这是因为将来还原的时候，内层双引号可以让 JavaScript 引擎知道，这是一个字符串，而不是其他类型的值。

```
JSON.stringify(false) // "false"
JSON.stringify('false') // "\"false\""
```

上面代码中，如果不是内层的双引号，将来还原的时候，引擎就无法知道原始值是布尔值还是字符串。

如果对象的属性是 `undefined` 、函数或 XML 对象，该属性会被 `JSON.stringify` 过滤。

```
var obj = {
  a: undefined,
  b: function () {}
};

JSON.stringify(obj) // "{}"
```

上面代码中，对象 `obj` 的 `a` 属性是 `undefined` ，而 `b` 属性是一个函数，结果都被 `JSON.stringify` 过滤。

如果数组的成员是 `undefined` 、函数或 XML 对象，则这些值被转成 `null` 。

```
var arr = [undefined, function () {}];
JSON.stringify(arr) // "[null,null]"
```

上面代码中，数组 `arr` 的成员是 `undefined` 和函数，它们都被转成了 `null` 。

正则对象会被转成空对象。

```
JSON.stringify(/foo/) // "{}"
```

`JSON.stringify` 方法会忽略对象的不可遍历属性。

```
var obj = {};
Object.defineProperties(obj, {
  'foo': {
    value: 1,
    enumerable: true
  },
  'bar': {
    value: 2,
    enumerable: false
  }
});

JSON.stringify(obj); // "{"foo":1}"
```

上面代码中，`bar` 是 `obj` 对象的不可遍历属性，`JSON.stringify` 方法会忽略这个属性。

第二个参数

`JSON.stringify` 方法还可以接受一个数组，作为第二个参数，指定需要转成字符串的属性。

```
var obj = {
  'prop1': 'value1',
  'prop2': 'value2',
  'prop3': 'value3'
};

var selectedProperties = ['prop1', 'prop2'];

JSON.stringify(obj, selectedProperties)
// "{"prop1":"value1","prop2":"value2"}"
```

上面代码中，`JSON.stringify` 方法的第二个参数指定，只转 `prop1` 和 `prop2` 两个属性。

这个类似白名单的数组，只对对象的属性有效，对数组无效。

```
JSON.stringify(['a', 'b'], ['0'])
// "["a","b"]"

JSON.stringify({0: 'a', 1: 'b'}, ['0'])
// "{"0":"a"}"
```

上面代码中，第二个参数指定 JSON 格式只转 `0` 号属性，实际上对数组是无效的，只对对象有效。

第二个参数还可以是一个函数，用来更改 `JSON.stringify` 的返回值。

```
function f(key, value) {
  if (typeof value === "number") {
    value = 2 * value;
  }
  return value;
}

JSON.stringify({ a: 1, b: 2 }, f)
// '{"a": 2,"b": 4}'
```

上面代码中的 `f` 函数，接受两个参数，分别是被转换的对象的键名和键值。如果键值是数值，就将它乘以 `2` ，否则就原样返回。

注意，这个处理函数是递归处理所有的键。

```
var o = {a: {b: 1}};

function f(key, value) {
  console.log("["+ key +"]:" + value);
  return value;
}

JSON.stringify(o, f)
```

```
// []:[object Object]
// [a]:[object Object]
// [b]:1
// '{"a":{"b":1}}'
```

上面代码中，对象 `o` 一共会被 `f` 函数处理三次，最后那行是 `JSON.stringify` 的输出。第一次键名为空，键值是整个对象 `o`；第二次键名为 `a`，键值是 `{b: 1}`；第三次键名为 `b`，键值为1。

递归处理中，每一次处理的对象，都是前一次返回的值。

```
var o = {a: 1};

function f(key, value) {
  if (typeof value === 'object') {
    return {b: 2};
  }
  return value * 2;
}

JSON.stringify(o, f)
// "{"b": 4}"
```

上面代码中，`f` 函数修改了对象 `o`，接着 `JSON.stringify` 方法就递归处理修改后的对象 `o`。

如果处理函数返回 `undefined` 或没有返回值，则该属性会被忽略。

```
function f(key, value) {
  if (typeof(value) === "string") {
    return undefined;
  }
  return value;
}

JSON.stringify({ a: "abc", b: 123 }, f)
```

```
// '{"b": 123}'
```

上面代码中，`a`属性经过处理后，返回`undefined`，于是该属性被忽略了。

第三个参数

`JSON.stringify`还可以接受第三个参数，用于增加返回的 JSON 字符串的可读性。如果是数字，表示每个属性前面添加的空格（最多不超过10个）；如果是字符串（不超过10个字符），则该字符串会添加在每行前面。

```
JSON.stringify({ p1: 1, p2: 2 }, null, 2);
/*
"{
  "p1": 1,
  "p2": 2
}"
*/

JSON.stringify({ p1:1, p2:2 }, null, '|-');
/*
"{
|-"p1": 1,
|-"p2": 2
}"
*/
```

参数对象的 toJSON 方法

如果参数对象有自定义的`toJSON`方法，那么`JSON.stringify`会使用这个方法的返回值作为参数，而忽略原对象的其他属性。

下面是一个普通的对象。

```
var user = {
  firstName: '三',
  lastName: '张',

  get fullName(){
    return this.lastName + this.firstName;
  }
};

JSON.stringify(user)
// "{"firstName":"三","lastName":"张","fullName":"张三"}"
```

现在，为这个对象加上 `toJSON` 方法。

```
var user = {
  firstName: '三',
  lastName: '张',

  get fullName(){
    return this.lastName + this.firstName;
  },

  toJSON: function () {
    return {
      name: this.lastName + this.firstName
    };
  }
};

JSON.stringify(user)
// "{"name":"张三"}"
```

上面代码中，`JSON.stringify` 发现参数对象有 `toJSON` 方法，就直接使用这个方法的返回值作为参数，而忽略原对象的其他参数。

`Date` 对象就有一个自己的 `toJSON` 方法。

```
var date = new Date('2015-01-01');
date.toJSON() // "2015-01-01T00:00:00.000Z"
JSON.stringify(date) // ""2015-01-01T00:00:00.000Z""
```

上面代码中，`JSON.stringify` 发现处理的是 `Date` 对象实例，就会调用这个实例对象的 `toJSON` 方法，将该方法的返回值作为参数。

`toJSON` 方法的一个应用是，将正则对象自动转为字符串。因为 `JSON.stringify` 默认不能转换正则对象，但是设置了 `toJSON` 方法以后，就可以转换正则对象了。

```
var obj = {
  reg: /foo/
};

// 不设置 toJSON 方法时
JSON.stringify(obj) // "{"reg":{}}"

// 设置 toJSON 方法时
RegExp.prototype.toJSON = RegExp.prototype.toString;
JSON.stringify(/foo/) // ""/foo/""
```

上面代码在正则对象的原型上面部署了 `toJSON` 方法，将其指向 `toString` 方法，因此遇到转换成 `JSON` 时，正则对象就先调用 `toJSON` 方法转为字符串，然后再被 `JSON.stingify` 方法处理。

JSON.parse()

`JSON.parse` 方法用于将 JSON 字符串转换成对应的值。

```
JSON.parse('{}') // {}
JSON.parse('true') // true
JSON.parse('"foo"') // "foo"
JSON.parse('[1, 5, "false"]') // [1, 5, "false"]
```

```
JSON.parse('null') // null

var o = JSON.parse('{"name": "张三"}');
o.name // 张三
```

如果传入的字符串不是有效的 JSON 格式，`JSON.parse` 方法将报错。

```
JSON.parse("'String'") // illegal single quotes
// SyntaxError: Unexpected token ILLEGAL
```

上面代码中，双引号字符串中是一个单引号字符串，因为单引号字符串不符合 JSON 格式，所以报错。

为了处理解析错误，可以将 `JSON.parse` 方法放在 `try...catch` 代码块中。

```
try {
  JSON.parse("'String'");
} catch(e) {
  console.log('parsing error');
}
```

`JSON.parse` 方法可以接受一个处理函数，作为第二个参数，用法与 `JSON.stringify` 方法类似。

```
function f(key, value) {
  if (key === 'a') {
    return value + 10;
  }
  return value;
}

JSON.parse('{"a": 1, "b": 2}', f)
// {a: 11, b: 2}
```

上面代码中，`JSON.parse`的第二个参数是一个函数，如果键名是`a`，该函数会将键值加上10。

参考链接

- MDN, Using native JSON
- MDN, JSON.parse
- Dr. Axel Rauschmayer, JavaScript’s JSON API
- Jim Cowart, What You Might Not Know About JSON.stringify()
- Marco Rogers polotek, What is JSON?

面向对象编程

面向对象编程

实例对象与 new 命令

实例对象与 new 命令

JavaScript 语言具有很强的面向对象编程能力，本章介绍 JavaScript 面向对象编程的基础知识。

对象是什么

面向对象编程（Object Oriented Programming，缩写为 OOP）是目前主流的编程范式。它将真实世界各种复杂的关系，抽象为一个个对象，然后由对象之间的分工与合作，完成对真实世界的模拟。

每一个对象都是功能中心，具有明确分工，可以完成接受信息、处理数据、发出信息等任务。对象可以复用，通过继承机制还可以定制。因此，面向对象编程具有灵活、代码可复用、高度模块化等特点，容易维护和开发，比起由一系列函数或指令组成的传统的过程式编程（procedural programming），更适合多人合作的大型软件项目。

那么，“对象”（object）到底是什么？我们从两个层次来理解。

（1）对象是单个实物的抽象。

一本书、一辆汽车、一个人都可以是对象，一个数据库、一张网页、一个与远程服务器的连接也可以是对象。当实物被抽象成对象，实物之间的关系就变成了对象之间的关系，从而就可以模拟现实情况，针对对象进行编程。

（2）对象是一个容器，封装了属性（property）和方法（method）。

属性是对象的状态，方法是对象的行为（完成某种任务）。比如，我们可以把动物抽象为 `animal` 对象，使用“属性”记录具体是那一种动物，使用“方法”表示动物的某种行为（奔跑、捕猎、休息等等）。

构造函数

面向对象编程的第一步，就是要生成对象。前面说过，对象是单个实物的抽象。通常需要一个模板，表示某一类实物的共同特征，然后对象根据这个模板生成。

典型的面向对象编程语言（比如 C++ 和 Java），都有“类”（class）这个概念。所谓“类”就是对象的模板，对象就是“类”的实例。但是，JavaScript 语言的对象体系，不是基于“类”的，而是基于构造函数（constructor）和原型链（prototype）。

JavaScript 语言使用构造函数（constructor）作为对象的模板。所谓”构造函数”，就是专门用来生成实例对象的函数。它就是对象的模板，描述实例对象的基本结构。一个构造函数，可以生成多个实例对象，这些实例对象都有相同的结构。

构造函数就是一个普通的函数，但是有自己的特征和用法。

```
var Vehicle = function () {
  this.price = 1000;
};
```

上面代码中，`Vehicle`就是构造函数。为了与普通函数区别，构造函数名字的第一个字母通常大写。

构造函数的特点有两个。

- 函数体内部使用了`this`关键字，代表了所要生成的对象实例。
- 生成对象的时候，必须使用`new`命令。

下面先介绍`new`命令。

new 命令

基本用法

`new`命令的作用，就是执行构造函数，返回一个实例对象。

```
var Vehicle = function () {
  this.price = 1000;
};

var v = new Vehicle();
v.price // 1000
```

上面代码通过`new`命令，让构造函数`Vehicle`生成一个实例对象，保存在变量`v`中。这个新生成的实例对象，从构造函数`Vehicle`得到了`price`属性。`new`命令执行时，构造函数内部的`this`，就代表了新生成的实例对象，`this.price`表示实例对象有一个`price`属性，值是1000。

使用 `new` 命令时，根据需要，构造函数也可以接受参数。

```
var Vehicle = function (p) {
  this.price = p;
};

var v = new Vehicle(500);
```

`new` 命令本身就可以执行构造函数，所以后面的构造函数可以带括号，也可以不带括号。下面两行代码是等价的，但是为了表示这里是函数调用，推荐使用括号。

```
// 推荐的写法
var v = new Vehicle();
// 不推荐的写法
var v = new Vehicle;
```

一个很自然的问题是，如果忘了使用 `new` 命令，直接调用构造函数会发生什么事？

这种情况下，构造函数就变成了普通函数，并不会生成实例对象。而且由于后面会说到的原因， `this` 这时代表全局对象，将造成一些意想不到的结果。

```
var Vehicle = function (){
  this.price = 1000;
};

var v = Vehicle();
v // undefined
price // 1000
```

上面代码中，调用 `Vehicle` 构造函数时，忘了加上 `new` 命令。结果，变量 `v` 变成了 `undefined` ，而 `price` 属性变成了全局变量。因此，

应该非常小心，避免不使用 `new` 命令、直接调用构造函数。

为了保证构造函数必须与 `new` 命令一起使用，一个解决办法是，构造函数内部使用严格模式，即第一行加上 `use strict` 。这样的话，一旦忘了使用 `new` 命令，直接调用构造函数就会报错。

```
function Fubar(foo, bar){
  'use strict';
  this._foo = foo;
  this._bar = bar;
}

Fubar()
// TypeError: Cannot set property '_foo' of undefined
```

上面代码的 `Fubar` 为构造函数， `use strict` 命令保证了该函数在严格模式下运行。由于严格模式中，函数内部的 `this` 不能指向全局对象，默认等于 `undefined` ，导致不加 `new` 调用会报错（JavaScript 不允许对 `undefined` 添加属性）。

另一个解决办法，构造函数内部判断是否使用 `new` 命令，如果发现没有使用，则直接返回一个实例对象。

```
function Fubar(foo, bar) {
  if (!(this instanceof Fubar)) {
    return new Fubar(foo, bar);
  }

  this._foo = foo;
  this._bar = bar;
}

Fubar(1, 2)._foo // 1
(new Fubar(1, 2))._foo // 1
```

上面代码中的构造函数，不管加不加 `new` 命令，都会得到同样的结果。

new 命令的原理

使用 `new` 命令时，它后面的函数依次执行下面的步骤。

1. 创建一个空对象，作为将要返回的对象实例。
2. 将这个空对象的原型，指向构造函数的 `prototype` 属性。
3. 将这个空对象赋值给函数内部的 `this` 关键字。
4. 开始执行构造函数内部的代码。

也就是说，构造函数内部， `this` 指的是一个新生成的空对象，所有针对 `this` 的操作，都会发生在这个空对象上。构造函数之所以叫“构造函数”，就是说这个函数的目的，就是操作一个空对象（即 `this` 对象），将其“构造”为需要的样子。

如果构造函数内部有 `return` 语句，而且 `return` 后面跟着一个对象， `new` 命令会返回 `return` 语句指定的对象；否则，就会不管 `return` 语句，返回 `this` 对象。

```
var Vehicle = function () {
  this.price = 1000;
  return 1000;
};

(new Vehicle()) === 1000
// false
```

上面代码中，构造函数 `Vehicle` 的 `return` 语句返回一个数值。这时， `new` 命令就会忽略这个 `return` 语句，返回“构造”后的 `this` 对象。

但是，如果 `return` 语句返回的是一个跟 `this` 无关的新对象， `new` 命令会返回这个新对象，而不是 `this` 对象。这一点需要特别引起注意。

```
var Vehicle = function (){
  this.price = 1000;
  return { price: 2000 };
};

(new Vehicle()).price
// 2000
```

上面代码中，构造函数 `Vehicle` 的 `return` 语句，返回的是一个新对象。 `new` 命令会返回这个对象，而不是 `this` 对象。

另一方面，如果对普通函数（内部没有 `this` 关键字的函数）使用 `new` 命令，则会返回一个空对象。

```
function getMessage() {
  return 'this is a message';
}

var msg = new getMessage();

msg // {}
typeof msg // "object"
```

上面代码中， `getMessage` 是一个普通函数，返回一个字符串。对它使用 `new` 命令，会得到一个空对象。这是因为 `new` 命令总是返回一个对象，要么是实例对象，要么是 `return` 语句指定的对象。本例中， `return` 语句返回的是字符串，所以 `new` 命令就忽略了该语句。

`new` 命令简化的内部流程，可以用下面的代码表示。

```
function _new(/* 构造函数 */ constructor, /* 构造函数参数 */ params) {
  // 将 arguments 对象转为数组
  var args = [].slice.call(arguments);
  // 取出构造函数
  var constructor = args.shift();
  // 创建一个空对象，继承构造函数的 prototype 属性
  var context = Object.create(constructor.prototype);
  // 执行构造函数
  var result = constructor.apply(context, args);
  // 如果返回结果是对象，就直接返回，否则返回 context 对象
  return (typeof result === 'object' && result != null) ? result : context;
}

// 实例
var actor = _new(Person, '张三', 28);
```

new.target

函数内部可以使用 `new.target` 属性。如果当前函数是 `new` 命令调用，`new.target` 指向当前函数，否则为 `undefined` 。

```
function f() {
  console.log(new.target === f);
}

f() // false
new f() // true
```

使用这个属性，可以判断函数调用的时候，是否使用 `new` 命令。

```
function f() {
  if (!new.target) {
    throw new Error('请使用 new 命令调用！');
  }
  // ...
```

```
}

f() // Uncaught Error: 请使用 new 命令调用！
```

上面代码中，构造函数 `f` 调用时，没有使用 `new` 命令，就抛出一个错误。

Object.create() 创建实例对象

构造函数作为模板，可以生成实例对象。但是，有时拿不到构造函数，只能拿到一个现有的对象。我们希望以这个现有的对象作为模板，生成新的实例对象，这时就可以使用 `Object.create()` 方法。

```
var person1 = {
  name: '张三',
  age: 38,
  greeting: function() {
    console.log('Hi! I\'m ' + this.name + '.');
  }
};

var person2 = Object.create(person1);

person2.name // 张三
person2.greeting() // Hi! I'm 张三.
```

上面代码中，对象 `person1` 是 `person2` 的模板，后者继承了前者的属性和方法。

`Object.create()` 的详细介绍，请看后面的相关章节。

this 关键字

this 关键字

涵义

`this` 关键字是一个非常重要的语法点。毫不夸张地说，不理解它的含义，大部分开发任务都无法完成。

前一章已经提到，`this` 可以用在构造函数之中，表示实例对象。除此之外，`this` 还可以用在别的场合。但不管是什么场合，`this` 都有一个共同点：它总是返回一个对象。

简单说，`this` 就是属性或方法“当前”所在的对象。

```
1.  this.property
```

上面代码中，`this` 就代表 `property` 属性当前所在的对象。

下面是一个实际的例子。

```
var person = {
  name: '张三',
  describe: function () {
    return '姓名：'+ this.name;
  }
};

person.describe()
// "姓名：张三"
```

上面代码中，`this.name` 表示 `name` 属性所在的那个对象。由于 `this.name` 是在 `describe` 方法中调用，而 `describe` 方法所在的当前对象是 `person`，因此 `this` 指向 `person`，`this.name` 就是 `person.name`。

由于对象的属性可以赋给另一个对象，所以属性所在的当前对象是可变的，即 `this` 的指向是可变的。

```
var A = {
  name: '张三',
  describe: function () {
    return '姓名：'+ this.name;
  }
};

var B = {
  name: '李四'
};

B.describe = A.describe;
B.describe()
// "姓名：李四"
```

上面代码中，`A.describe` 属性被赋给 `B`，于是 `B.describe` 就表示 `describe` 方法所在的当前对象是 `B`，所以 `this.name` 就指向 `B.name`。

稍稍重构这个例子，`this` 的动态指向就能看得更清楚。

```
function f() {
  return '姓名：'+ this.name;
}

var A = {
  name: '张三',
  describe: f
};

var B = {
  name: '李四',
  describe: f
};

A.describe() // "姓名：张三"
B.describe() // "姓名：李四"
```

上面代码中，函数 `f` 内部使用了 `this` 关键字，随着 `f` 所在的对象不同，`this` 的指向也不同。

只要函数被赋给另一个变量，`this` 的指向就会变。

```
var A = {
  name: '张三',
  describe: function () {
    return '姓名：'+ this.name;
  }
};

var name = '李四';
```

```
var f = A.describe;
f() // "姓名：李四"
```

上面代码中，`A.describe` 被赋值给变量 `f`，内部的 `this` 就会指向 `f` 运行时所在的对象（本例是顶层对象）。

再看一个网页编程的例子。

```
<input type="text" name="age" size=3 onChange="validate(this, 18, 99);">

<script>
function validate(obj, lowval, hival){
  if ((obj.value < lowval) || (obj.value > hival))
    console.log('Invalid Value!');
}
</script>
```

上面代码是一个文本输入框，每当用户输入一个值，就会调用 `onChange` 回调函数，验证这个值是否在指定范围。浏览器会向回调函数传入当前对象，因此 `this` 就代表传入当前对象（即文本框），然后就可以从 `this.value` 上面读到用户的输入值。

总结一下，JavaScript 语言之中，一切皆对象，运行环境也是对象，所以函数都是在某个对象之中运行，`this` 就是函数运行时所在的对象（环境）。这本来并不会让用户糊涂，但是 JavaScript 支持运行环境动态切换，也就是说，`this` 的指向是动态的，没有办法事先确定到底指向哪个对象，这才是最让初学者感到困惑的地方。

使用场合

`this` 主要有以下几个使用场合。

（1）全局环境

全局环境使用 `this`，它指的就是顶层对象 `window`。

```
this === window // true

function f() {
  console.log(this === window);
}
f() // true
```

上面代码说明，不管是不是在函数内部，只要是在全局环境下运行，`this` 就是指顶层对象 `window`。

（2）构造函数

构造函数中的 `this`，指的是实例对象。

```
var Obj = function (p) {
  this.p = p;
};
```

上面代码定义了一个构造函数 `Obj`。由于 `this` 指向实例对象，所以在构造函数内部定义 `this.p`，就相当于定义实例对象有一个 `p` 属性。

```
var o = new Obj('Hello World!');
o.p // "Hello World!"
```

（3）对象的方法

如果对象的方法里面包含 `this`，`this` 的指向就是方法运行时所在的对象。该方法赋值给另一个对象，就会改变 `this` 的指向。

但是，这条规则很不容易把握。请看下面的代码。

```
var obj ={
```

```
  foo: function () {
    console.log(this);
  }
};

obj.foo() // obj
```

上面代码中，`obj.foo` 方法执行时，它内部的 `this` 指向 `obj` 。

但是，下面这几种用法，都会改变 `this` 的指向。

```
// 情况一
(obj.foo = obj.foo)() // window
// 情况二
(false || obj.foo)() // window
// 情况三
(1, obj.foo)() // window
```

上面代码中，`obj.foo` 就是一个值。这个值真正调用的时候，运行环境已经不是 `obj` 了，而是全局环境，所以 `this` 不再指向 `obj` 。

可以这样理解，JavaScript 引擎内部，`obj` 和 `obj.foo` 储存在两个内存地址，称为地址一和地址二。`obj.foo()` 这样调用时，是从地址一调用地址二，因此地址二的运行环境是地址一，`this` 指向 `obj` 。但是，上面三种情况，都是直接取出地址二进行调用，这样的话，运行环境就是全局环境，因此 `this` 指向全局环境。上面三种情况等同于下面的代码。

```
// 情况一
(obj.foo = function () {
  console.log(this);
})()
// 等同于
(function () {
  console.log(this);
```

```
})()

// 情况二
(false || function () {
  console.log(this);
})()

// 情况三
(1, function () {
  console.log(this);
})()
```

如果 `this` 所在的方法不在对象的第一层，这时 `this` 只是指向当前一层的对象，而不会继承更上面的层。

```
var a = {
  p: 'Hello',
  b: {
    m: function() {
      console.log(this.p);
    }
  }
};

a.b.m() // undefined
```

上面代码中，`a.b.m` 方法在 `a` 对象的第二层，该方法内部的 `this` 不是指向 `a`，而是指向 `a.b`，因为实际执行的是下面的代码。

```
var b = {
  m: function() {
   console.log(this.p);
  }
};

```

```
var a = {
  p: 'Hello',
  b: b
};

(a.b).m() // 等同于 b.m()
```

如果要达到预期效果，只有写成下面这样。

```
var a = {
  b: {
    m: function() {
      console.log(this.p);
    },
    p: 'Hello'
  }
};
```

如果这时将嵌套对象内部的方法赋值给一个变量， `this` 依然会指向全局对象。

```
var a = {
  b: {
    m: function() {
      console.log(this.p);
    },
    p: 'Hello'
  }
};

var hello = a.b.m;
hello() // undefined
```

上面代码中， `m` 是多层对象内部的一个方法。为求简便，将其赋值给 `hello` 变量，结果调用时， `this` 指向了顶层对象。为了避免这个问题，可以只将 `m` 所在的对象赋值给 `hello` ，这样调用

时，`this` 的指向就不会变。

```
var hello = a.b;
hello.m() // Hello
```

使用注意点

避免多层 this

由于 `this` 的指向是不确定的，所以切勿在函数中包含多层的 `this` 。

```
var o = {
  f1: function () {
    console.log(this);
    var f2 = function () {
      console.log(this);
    }();
  }
}

o.f1()
// Object
// Window
```

上面代码包含两层 `this` ，结果运行后，第一层指向对象 `o` ，第二层指向全局对象，因为实际执行的是下面的代码。

```
var temp = function () {
  console.log(this);
};

var o = {
  f1: function () {
    console.log(this);
    var f2 = temp();
```

```
  }
}
```

一个解决方法是在第二层改用一个指向外层 `this` 的变量。

```
var o = {
  f1: function() {
    console.log(this);
    var that = this;
    var f2 = function() {
      console.log(that);
    }();
  }
}

o.f1()
// Object
// Object
```

上面代码定义了变量 `that` ，固定指向外层的 `this` ，然后在内层使用 `that` ，就不会发生 `this` 指向的改变。

事实上，使用一个变量固定 `this` 的值，然后内层函数调用这个变量，是非常常见的做法，请务必掌握。

JavaScript 提供了严格模式，也可以硬性避免这种问题。严格模式下，如果函数内部的 `this` 指向顶层对象，就会报错。

```
var counter = {
  count: 0
};
counter.inc = function () {
  'use strict';
  this.count++
};
var f = counter.inc;
```

```
f()
// TypeError: Cannot read property 'count' of undefined
```

上面代码中，`inc` 方法通过 `'use strict'` 声明采用严格模式，这时内部的 `this` 一旦指向顶层对象，就会报错。

避免数组处理方法中的 this

数组的 `map` 和 `foreach` 方法，允许提供一个函数作为参数。这个函数内部不应该使用 `this`。

```
var o = {
  v: 'hello',
  p: [ 'a1', 'a2' ],
  f: function f() {
    this.p.forEach(function (item) {
      console.log(this.v + ' ' + item);
    });
  }
}

o.f()
// undefined a1
// undefined a2
```

上面代码中，`foreach` 方法的回调函数中的 `this`，其实是指向 `window` 对象，因此取不到 `o.v` 的值。原因跟上一段的多层 `this` 是一样的，就是内层的 `this` 不指向外部，而指向顶层对象。

解决这个问题的一种方法，就是前面提到的，使用中间变量固定 `this`。

```
var o = {
  v: 'hello',
  p: [ 'a1', 'a2' ],
```

```
  f: function f() {
    var that = this;
    this.p.forEach(function (item) {
      console.log(that.v+' '+item);
    });
  }
}

o.f()
// hello a1
// hello a2
```

另一种方法是将 `this` 当作 `foreach` 方法的第二个参数，固定它的运行环境。

```
var o = {
  v: 'hello',
  p: [ 'a1', 'a2' ],
  f: function f() {
    this.p.forEach(function (item) {
      console.log(this.v + ' ' + item);
    }, this);
  }
}

o.f()
// hello a1
// hello a2
```

避免回调函数中的 this

回调函数中的 `this` 往往会改变指向，最好避免使用。

```
var o = new Object();
o.f = function () {
  console.log(this === o);
```

```
}

// jQuery 的写法
$('#button').on('click', o.f);
```

上面代码中，点击按钮以后，控制台会显示 `false` 。原因是此时 `this` 不再指向 `o` 对象，而是指向按钮的 DOM 对象，因为 `f` 方法是在按钮对象的环境中被调用的。这种细微的差别，很容易在编程中忽视，导致难以察觉的错误。

为了解决这个问题，可以采用下面的一些方法对 `this` 进行绑定，也就是使得 `this` 固定指向某个对象，减少不确定性。

绑定 this 的方法

`this` 的动态切换，固然为 JavaScript 创造了巨大的灵活性，但也使得编程变得困难和模糊。有时，需要把 `this` 固定下来，避免出现意想不到的情况。JavaScript 提供了 `call` 、 `apply` 、 `bind` 这三个方法，来切换/固定 `this` 的指向。

Function.prototype.call()

函数实例的 `call` 方法，可以指定函数内部 `this` 的指向（即函数执行时所在的作用域），然后在所指定的作用域中，调用该函数。

```
var obj = {};

var f = function () {
  return this;
};

f() === window // true
f.call(obj) === obj // true
```

上面代码中，全局环境运行函数 `f` 时， `this` 指向全局环境（浏览器为 `window` 对象）； `call` 方法可以改变 `this` 的指向，指定 `this` 指向对象 `obj` ，然后在对象 `obj` 的作用域中运行函数 `f` 。

`call` 方法的参数，应该是一个对象。如果参数为空、 `null` 和 `undefined` ，则默认传入全局对象。

```
var n = 123;
var obj = { n: 456 };

function a() {
  console.log(this.n);
}

a.call() // 123
a.call(null) // 123
a.call(undefined) // 123
a.call(window) // 123
a.call(obj) // 456
```

上面代码中， `a` 函数中的 `this` 关键字，如果指向全局对象，返回结果为 `123` 。如果使用 `call` 方法将 `this` 关键字指向 `obj` 对象，返回结果为 `456` 。可以看到，如果 `call` 方法没有参数，或者参数为 `null` 或 `undefined` ，则等同于指向全局对象。

如果 `call` 方法的参数是一个原始值，那么这个原始值会自动转成对应的包装对象，然后传入 `call` 方法。

```
var f = function () {
  return this;
};

f.call(5)
// Number {[[PrimitiveValue]]: 5}
```

上面代码中，`call` 的参数为 `5`，不是对象，会被自动转成包装对象（`Number` 的实例），绑定 `f` 内部的 `this`。

`call` 方法还可以接受多个参数。

```
func.call(thisValue, arg1, arg2, ...)
```

`call` 的第一个参数就是 `this` 所要指向的那个对象，后面的参数则是函数调用时所需的参数。

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

add.call(this, 1, 2) // 3
```

上面代码中，`call` 方法指定函数 `add` 内部的 `this` 绑定当前环境（对象），并且参数为 `1` 和 `2`，因此函数 `add` 运行后得到 `3`。

`call` 方法的一个应用是调用对象的原生方法。

```
var obj = {};
obj.hasOwnProperty('toString') // false

// 覆盖掉继承的 hasOwnProperty 方法
obj.hasOwnProperty = function () {
  return true;
};
obj.hasOwnProperty('toString') // true

Object.prototype.hasOwnProperty.call(obj, 'toString') // false
```

上面代码中，`hasOwnProperty` 是 `obj` 对象继承的方法，如果这个方法一旦被覆盖，就不会得到正确结果。`call` 方法可以解决这个问题，它将 `hasOwnProperty` 方法的原始定义放到 `obj` 对象上执行，这样无

论 `obj` 上有没有同名方法，都不会影响结果。

Function.prototype.apply()

`apply` 方法的作用与 `call` 方法类似，也是改变 `this` 指向，然后再调用该函数。唯一的区别就是，它接收一个数组作为函数执行时的参数，使用格式如下。

```
1.  func.apply(thisValue, [arg1, arg2, ...])
```

`apply` 方法的第一个参数也是 `this` 所要指向的那个对象，如果设为 `null` 或 `undefined` ，则等同于指定全局对象。第二个参数则是一个数组，该数组的所有成员依次作为参数，传入原函数。原函数的参数，在 `call` 方法中必须一个个添加，但是在 `apply` 方法中，必须以数组形式添加。

```
1.  function f(x, y){
2.    console.log(x + y);
3.  }
4.
5.  f.call(null, 1, 1) // 2
6.  f.apply(null, [1, 1]) // 2
```

上面代码中， `f` 函数本来接受两个参数，使用 `apply` 方法以后，就变成可以接受一个数组作为参数。

利用这一点，可以做一些有趣的应用。

（1）找出数组最大元素

JavaScript 不提供找出数组最大元素的函数。结合使用 `apply` 方法和 `Math.max` 方法，就可以返回数组的最大元素。

```
var a = [10, 2, 4, 15, 9];
Math.max.apply(null, a) // 15
```

（2）将数组的空元素变为 `undefined`

通过 `apply` 方法，利用 `Array` 构造函数将数组的空元素变成 `undefined` 。

```
Array.apply(null, ['a', ,'b'])
// [ 'a', undefined, 'b' ]
```

空元素与 `undefined` 的差别在于，数组的 `forEach` 方法会跳过空元素，但是不会跳过 `undefined` 。因此，遍历内部元素的时候，会得到不同的结果。

```
var a = ['a', , 'b'];

function print(i) {
  console.log(i);
}

a.forEach(print)
// a
// b

Array.apply(null, a).forEach(print)
// a
// undefined
// b
```

（3）转换类似数组的对象

另外，利用数组对象的 `slice` 方法，可以将一个类似数组的对象（比如 `arguments` 对象）转为真正的数组。

```
Array.prototype.slice.apply({0: 1, length: 1}) // [1]
Array.prototype.slice.apply({0: 1}) // []
Array.prototype.slice.apply({0: 1, length: 2}) // [1, undefined]
Array.prototype.slice.apply({length: 1}) // [undefined]
```

上面代码的 `apply` 方法的参数都是对象，但是返回结果都是数组，这就起到了将对象转成数组的目的。从上面代码可以看到，这个方法起作用的前提是，被处理的对象必须有 `length` 属性，以及相对应的数字键。

（4）绑定回调函数的对象

前面的按钮点击事件的例子，可以改写如下。

```
var o = new Object();

o.f = function () {
  console.log(this === o);
}

var f = function (){
  o.f.apply(o);
  // 或者 o.f.call(o);
};

// jQuery 的写法
$('#button').on('click', f);
```

上面代码中，点击按钮以后，控制台将会显示 `true` 。由于 `apply` 方法（或者 `call` 方法）不仅绑定函数执行时所在的对象，还会立即执行函数，因此不得不把绑定语句写在一个函数体内。更简洁的写法是采用下面介绍的 `bind` 方法。

Function.prototype.bind()

`bind` 方法用于将函数体内的 `this` 绑定到某个对象，然后返回一个新函数。

```
var d = new Date();
d.getTime() // 1481869925657

var print = d.getTime;
print() // Uncaught TypeError: this is not a Date object.
```

上面代码中，我们将 `d.getTime` 方法赋给变量 `print` ，然后调用 `print` 就报错了。这是因为 `getTime` 方法内部的 `this` ，绑定 `Date` 对象的实例，赋给变量 `print` 以后，内部的 `this` 已经不指向 `Date` 对象的实例了。

`bind` 方法可以解决这个问题。

```
var print = d.getTime.bind(d);
print() // 1481869925657
```

上面代码中， `bind` 方法将 `getTime` 方法内部的 `this` 绑定到 `d` 对象，这时就可以安全地将这个方法赋值给其他变量了。

`bind` 方法的参数就是所要绑定 `this` 的对象，下面是一个更清晰的例子。

```
var counter = {
  count: 0,
  inc: function () {
    this.count++;
  }
};

var func = counter.inc.bind(counter);
func();
counter.count // 1
```

上面代码中，`counter.inc` 方法被赋值给变量 `func`。这时必须用 `bind` 方法将 `inc` 内部的 `this`，绑定到 `counter`，否则就会出错。

`this` 绑定到其他对象也是可以的。

```
var counter = {
  count: 0,
  inc: function () {
    this.count++;
  }
};

var obj = {
  count: 100
};
var func = counter.inc.bind(obj);
func();
obj.count // 101
```

上面代码中，`bind` 方法将 `inc` 方法内部的 `this`，绑定到 `obj` 对象。结果调用 `func` 函数以后，递增的就是 `obj` 内部的 `count` 属性。

`bind` 还可以接受更多的参数，将这些参数绑定原函数的参数。

```
var add = function (x, y) {
  return x * this.m + y * this.n;
}

var obj = {
  m: 2,
  n: 2
};

var newAdd = add.bind(obj, 5);
```

```
newAdd(5) // 20
```

上面代码中，`bind` 方法除了绑定 `this` 对象，还将 `add` 函数的第一个参数 `x` 绑定成 `5`，然后返回一个新函数 `newAdd`，这个函数只要再接受一个参数 `y` 就能运行了。

如果 `bind` 方法的第一个参数是 `null` 或 `undefined`，等于将 `this` 绑定到全局对象，函数运行时 `this` 指向顶层对象（浏览器为 `window`）。

```
function add(x, y) {
  return x + y;
}

var plus5 = add.bind(null, 5);
plus5(10) // 15
```

上面代码中，函数 `add` 内部并没有 `this`，使用 `bind` 方法的主要目的是绑定参数 `x`，以后每次运行新函数 `plus5`，就只需要提供另一个参数 `y` 就够了。而且因为 `add` 内部没有 `this`，所以 `bind` 的第一个参数是 `null`，不过这里如果是其他对象，也没有影响。

`bind` 方法有一些使用注意点。

（1）每一次返回一个新函数

`bind` 方法每运行一次，就返回一个新函数，这会产生一些问题。比如，监听事件的时候，不能写成下面这样。

```
element.addEventListener('click', o.m.bind(o));
```

上面代码中，`click` 事件绑定 `bind` 方法生成的一个匿名函数。这样会导致无法取消绑定，所以，下面的代码是无效的。

```
element.removeEventListener('click', o.m.bind(o));
```

正确的方法是写成下面这样：

```
var listener = o.m.bind(o);
element.addEventListener('click', listener);
//  ...
element.removeEventListener('click', listener);
```

（2）结合回调函数使用

回调函数是 JavaScript 最常用的模式之一，但是一个常见的错误是，将包含 `this` 的方法直接当作回调函数。解决方法就是使用 `bind` 方法，将 `counter.inc` 绑定 `counter` 。

```
var counter = {
  count: 0,
  inc: function () {
    'use strict';
    this.count++;
  }
};

function callIt(callback) {
  callback();
}

callIt(counter.inc.bind(counter));
counter.count // 1
```

上面代码中， `callIt` 方法会调用回调函数。这时如果直接把 `counter.inc` 传入，调用时 `counter.inc` 内部的 `this` 就会指向全局对象。使用 `bind` 方法将 `counter.inc` 绑定 `counter` 以后，就不会有这个问题， `this` 总是指向 `counter` 。

还有一种情况比较隐蔽，就是某些数组方法可以接受一个函数当作参数。这些函数内部的 `this` 指向，很可能也会出错。

```
var obj = {
  name: '张三',
  times: [1, 2, 3],
  print: function () {
    this.times.forEach(function (n) {
---
title: this 关键字
layout: page
category: oop
date: 2016-06-28
modifiedOn: 2016-06-28
---
      console.log(this.name);
    });
  }
};

obj.print()
// 没有任何输出
```

上面代码中，`obj.print` 内部 `this.times` 的 `this` 是指向 `obj` 的，这个没有问题。但是，`forEach` 方法的回调函数内部的 `this.name` 却是指向全局对象，导致没有办法取到值。稍微改动一下，就可以看得更清楚。

```
obj.print = function () {
  this.times.forEach(function (n) {
    console.log(this === window);
  });
};

obj.print()
// true
```

```
// true
// true
```

解决这个问题，也是通过 `bind` 方法绑定 `this` 。

```
obj.print = function () {
  this.times.forEach(function (n) {
    console.log(this.name);
  }.bind(this));
};

obj.print()
// 张三
// 张三
// 张三
```

（3）结合 `call` 方法使用

利用 `bind` 方法，可以改写一些 JavaScript 原生方法的使用形式，以数组的 `slice` 方法为例。

```
[1, 2, 3].slice(0, 1) // [1]
// 等同于
Array.prototype.slice.call([1, 2, 3], 0, 1) // [1]
```

上面的代码中，数组的 `slice` 方法从 `[1, 2, 3]` 里面，按照指定位置和长度切分出另一个数组。这样做的本质是在 `[1, 2, 3]` 上面调用 `Array.prototype.slice` 方法，因此可以用 `call` 方法表达这个过程，得到同样的结果。

`call` 方法实质上是调用 `Function.prototype.call` 方法，因此上面的表达式可以用 `bind` 方法改写。

```
var slice = Function.prototype.call.bind(Array.prototype.slice);
slice([1, 2, 3], 0, 1) // [1]
```

上面代码的含义就是，将 `Array.prototype.slice` 变成 `Function.prototype.call` 方法所在的对象，调用时就变成了 `Array.prototype.slice.call` 。类似的写法还可以用于其他数组方法。

```
var push = Function.prototype.call.bind(Array.prototype.push);
var pop = Function.prototype.call.bind(Array.prototype.pop);

var a = [1 ,2 ,3];
push(a, 4)
a // [1, 2, 3, 4]

pop(a)
a // [1, 2, 3]
```

如果再进一步，将 `Function.prototype.call` 方法绑定到 `Function.prototype.bind` 对象，就意味着 `bind` 的调用形式也可以被改写。

```
function f() {
  console.log(this.v);
}

var o = { v: 123 };
var bind = Function.prototype.call.bind(Function.prototype.bind);
bind(f, o)() // 123
```

上面代码的含义就是，将 `Function.prototype.bind` 方法绑定在 `Function.prototype.call` 上面，所以 `bind` 方法就可以直接使用，不需要在函数实例上使用。

参考链接

- Jonathan Creamer, Avoiding the “this” problem in JavaScript

- Erik Kronberg, Bind, Call and Apply in JavaScript
- Axel Rauschmayer, JavaScript’s this: how it works, where it can trip you up

对象的继承

对象的继承

面向对象编程很重要的一个方面，就是对象的继承。A 对象通过继承 B 对象，就能直接拥有 B 对象的所有属性和方法。这对于代码的复用是非常有用的。

大部分面向对象的编程语言，都是通过“类”（class）来实现对象的继承。JavaScript 语言的继承则是通过“原型对象”（prototype）。

原型对象概述

构造函数的缺点

JavaScript 通过构造函数生成新对象，因此构造函数可以视为对象的模板。实例对象的属性和方法，可以定义在构造函数内部。

```
function Cat (name, color) {
  this.name = name;
  this.color = color;
}

var cat1 = new Cat('大毛', '白色');

cat1.name // '大毛'
cat1.color // '白色'
```

上面代码中，`Cat` 函数是一个构造函数，函数内部定义了 `name` 属性和 `color` 属性，所有实例对象（上例是 `cat1` ）都会生成这两个属性，即这两个属性会定义在实例对象上面。

通过构造函数为实例对象定义属性，虽然很方便，但是有一个缺点。同一个构造函数的多个实例之间，无法共享属性，从而造成对系统资源的浪费。

```
function Cat(name, color) {
  this.name = name;
  this.color = color;
  this.meow = function () {
    console.log('喵喵');
  };
}

var cat1 = new Cat('大毛', '白色');
var cat2 = new Cat('二毛', '黑色');

cat1.meow === cat2.meow
```

```
13. // false
```

上面代码中，`cat1` 和 `cat2` 是同一个构造函数的两个实例，它们都具有 `meow` 方法。由于 `meow` 方法是生成在每个实例对象上面，所以两个实例就生成了两次。也就是说，每新建一个实例，就会新建一个 `meow` 方法。这既没有必要，又浪费系统资源，因为所有 `meow` 方法都是同样的行为，完全应该共享。

这个问题的解决方法，就是 JavaScript 的原型对象（prototype）。

prototype 属性的作用

JavaScript 继承机制的设计思想就是，原型对象的所有属性和方法，都能被实例对象共享。也就是说，如果属性和方法定义在原型上，那么所有实例对象就能共享，不仅节省了内存，还体现了实例对象之间的联系。

下面，先看怎么为对象指定原型。JavaScript 规定，每个函数都有一个 `prototype` 属性，指向一个对象。

```
1. function f() {}
2. typeof f.prototype // "object"
```

上面代码中，函数 `f` 默认具有 `prototype` 属性，指向一个对象。

对于普通函数来说，该属性基本无用。但是，对于构造函数来说，生成实例的时候，该属性会自动成为实例对象的原型。

```
1. function Animal(name) {
2.   this.name = name;
3. }
4. Animal.prototype.color = 'white';
```

```
5.
6.  var cat1 = new Animal('大毛');
7.  var cat2 = new Animal('二毛');
8.
9.  cat1.color // 'white'
10. cat2.color // 'white'
```

上面代码中，构造函数 `Animal` 的 `prototype` 属性，就是实例对象 `cat1` 和 `cat2` 的原型对象。原型对象上添加一个 `color` 属性，结果，实例对象都共享了该属性。

原型对象的属性不是实例对象自身的属性。只要修改原型对象，变动就立刻会体现在所有实例对象上。

```
1.  Animal.prototype.color = 'yellow';
2.
3.  cat1.color // "yellow"
4.  cat2.color // "yellow"
```

上面代码中，原型对象的 `color` 属性的值变为 `yellow` ，两个实例对象的 `color` 属性立刻跟着变了。这是因为实例对象其实没有 `color` 属性，都是读取原型对象的 `color` 属性。也就是说，当实例对象本身没有某个属性或方法的时候，它会到原型对象去寻找该属性或方法。这就是原型对象的特殊之处。

如果实例对象自身就有某个属性或方法，它就不会再去原型对象寻找这个属性或方法。

```
1.  cat1.color = 'black';
2.
3.  cat1.color // 'black'
4.  cat2.color // 'yellow'
5.  Animal.prototype.color // 'yellow';
```

上面代码中，实例对象 `cat1` 的 `color` 属性改为 `black` ，就使得它不再去原型对象读取 `color` 属性，后者的值依然为 `yellow` 。

总结一下，原型对象的作用，就是定义所有实例对象共享的属性和方法。这也是它被称为原型对象的原因，而实例对象可以视作从原型对象衍生出来的子对象。

```
1. Animal.prototype.walk = function () {
2.   console.log(this.name + ' is walking');
3. };
```

上面代码中， `Animal.prototype` 对象上面定义了一个 `walk` 方法，这个方法将可以在所有 `Animal` 实例对象上面调用。

原型链

JavaScript 规定，所有对象都有自己的原型对象（prototype）。一方面，任何一个对象，都可以充当其他对象的原型；另一方面，由于原型对象也是对象，所以它也有自己的原型。因此，就会形成一个“原型链”（prototype chain）：对象到原型，再到原型的原型……

如果一层层地上溯，所有对象的原型最终都可以上溯到 `Object.prototype` ，即 `Object` 构造函数的 `prototype` 属性。也就是说，所有对象都继承了 `Object.prototype` 的属性。这就是所有对象都有 `valueOf` 和 `toString` 方法的原因，因为这是从 `Object.prototype` 继承的。

那么， `Object.prototype` 对象有没有它的原型呢？回答是 `Object.prototype` 的原型是 `null` 。 `null` 没有任何属性和方法，也没有自己的原型。因此，原型链的尽头就是 `null` 。

```
1. Object.getPrototypeOf(Object.prototype)
```

```
2. // null
```

上面代码表示，`Object.prototype` 对象的原型是 `null`，由于 `null` 没有任何属性，所以原型链到此为止。`Object.getPrototypeOf` 方法返回参数对象的原型，具体介绍请看后文。

读取对象的某个属性时，JavaScript 引擎先寻找对象本身的属性，如果找不到，就到它的原型去找，如果还是找不到，就到原型的原型去找。如果直到最顶层的 `Object.prototype` 还是找不到，则返回 `undefined`。如果对象自身和它的原型，都定义了一个同名属性，那么优先读取对象自身的属性，这叫做“覆盖”（overriding）。

注意，一级级向上，在整个原型链上寻找某个属性，对性能是有影响的。所寻找的属性在越上层的原型对象，对性能的影响越大。如果寻找某个不存在的属性，将会遍历整个原型链。

举例来说，如果让构造函数的 `prototype` 属性指向一个数组，就意味着实例对象可以调用数组方法。

```
1. var MyArray = function () {};
2.
3. MyArray.prototype = new Array();
4. MyArray.prototype.constructor = MyArray;
5.
6. var mine = new MyArray();
7. mine.push(1, 2, 3);
8. mine.length // 3
9. mine instanceof Array // true
```

上面代码中，`mine` 是构造函数 `MyArray` 的实例对象，由于 `MyArray.prototype` 指向一个数组实例，使得 `mine` 可以调用数组方法（这些方法定义在数组实例的 `prototype` 对象上面）。最后那行 `instanceof` 表达式，用来比较一个对象是否为某个构造函数的实

例，结果就是证明 `mine` 为 `Array` 的实例， `instanceof` 运算符的详细解释详见后文。

上面代码还出现了原型对象的 `contructor` 属性，这个属性的含义下一节就来解释。

constructor 属性

`prototype` 对象有一个 `constructor` 属性，默认指向 `prototype` 对象所在的构造函数。

```
function P() {}
P.prototype.constructor === P // true
```

由于 `constructor` 属性定义在 `prototype` 对象上面，意味着可以被所有实例对象继承。

```
function P() {}
var p = new P();

p.constructor === P // true
p.constructor === P.prototype.constructor // true
p.hasOwnProperty('constructor') // false
```

上面代码中， `p` 是构造函数 `P` 的实例对象，但是 `p` 自身没有 `constructor` 属性，该属性其实是读取原型链上面的 `P.prototype.constructor` 属性。

`constructor` 属性的作用是，可以得知某个实例对象，到底是哪一个构造函数产生的。

```
function F() {};
var f = new F();

```

```
f.constructor === F // true
f.constructor === RegExp // false
```

上面代码中，`constructor`属性确定了实例对象`f`的构造函数是`F`，而不是`RegExp`。

另一方面，有了`constructor`属性，就可以从一个实例对象新建另一个实例。

```
function Constr() {}
var x = new Constr();

var y = new x.constructor();
y instanceof Constr // true
```

上面代码中，`x`是构造函数`Constr`的实例，可以从`x.constructor`间接调用构造函数。这使得在实例方法中，调用自身的构造函数成为可能。

```
Constr.prototype.createCopy = function () {
  return new this.constructor();
};
```

上面代码中，`createCopy`方法调用构造函数，新建另一个实例。

`constructor`属性表示原型对象与构造函数之间的关联关系，如果修改了原型对象，一般会同时修改`constructor`属性，防止引用的时候出错。

```
function Person(name) {
  this.name = name;
}

Person.prototype.constructor === Person // true

```

```
Person.prototype = {
  method: function () {}
};

Person.prototype.constructor === Person // false
Person.prototype.constructor === Object // true
```

上面代码中，构造函数 `Person` 的原型对象改掉了，但是没有修改 `constructor` 属性，导致这个属性不再指向 `Person` 。由于 `Person` 的新原型是一个普通对象，而普通对象的 `contructor` 属性指向 `Object` 构造函数，导致 `Person.prototype.constructor` 变成了 `Object` 。

所以，修改原型对象时，一般要同时修改 `constructor` 属性的指向。

```
// 坏的写法
C.prototype = {
  method1: function (...) { ... },
  // ...
};

// 好的写法
C.prototype = {
  constructor: C,
  method1: function (...) { ... },
  // ...
};

// 更好的写法
C.prototype.method1 = function (...) { ... };
```

上面代码中，要么将 `constructor` 属性重新指向原来的构造函数，要么只在原型对象上添加方法，这样可以保证 `instanceof` 运算符不会失真。

如果不能确定 `constructor` 属性是什么函数，还有一个办法：通过 `name` 属性，从实例得到构造函数的名称。

```
function Foo() {}
var f = new Foo();
f.constructor.name // "Foo"
```

instanceof 运算符

`instanceof` 运算符返回一个布尔值，表示对象是否为某个构造函数的实例。

```
var v = new Vehicle();
v instanceof Vehicle // true
```

上面代码中，对象 `v` 是构造函数 `Vehicle` 的实例，所以返回 `true` 。

`instanceof` 运算符的左边是实例对象，右边是构造函数。它会检查右边构建函数的原型对象（prototype），是否在左边对象的原型链上。因此，下面两种写法是等价的。

```
v instanceof Vehicle
// 等同于
Vehicle.prototype.isPrototypeOf(v)
```

上面代码中， `Object.prototype.isPrototypeOf` 的详细解释见后文。

由于 `instanceof` 检查整个原型链，因此同一个实例对象，可能会对多个构造函数都返回 `true` 。

```
var d = new Date();
d instanceof Date // true
d instanceof Object // true
```

上面代码中，`d` 同时是 `Date` 和 `Object` 的实例，因此对这两个构造函数都返回 `true`。

`instanceof` 的原理是检查右边构造函数的 `prototype` 属性，是否在左边对象的原型链上。有一种特殊情况，就是左边对象的原型链上，只有 `null` 对象。这时，`instanceof` 判断会失真。

```
var obj = Object.create(null);
typeof obj // "object"
Object.create(null) instanceof Object // false
```

上面代码中，`Object.create(null)` 返回一个新对象 `obj`，它的原型是 `null`（`Object.create` 的详细介绍见后文）。右边的构造函数 `Object` 的 `prototype` 属性，不在左边的原型链上，因此 `instanceof` 就认为 `obj` 不是 `Object` 的实例。但是，只要一个对象的原型不是 `null`，`instanceof` 运算符的判断就不会失真。

`instanceof` 运算符的一个用处，是判断值的类型。

```
var x = [1, 2, 3];
var y = {};
x instanceof Array // true
y instanceof Object // true
```

上面代码中，`instanceof` 运算符判断，变量 `x` 是数组，变量 `y` 是对象。

注意，`instanceof` 运算符只能用于对象，不适用原始类型的值。

```
var s = 'hello';
s instanceof String // false
```

上面代码中，字符串不是 `String` 对象的实例（因为字符串不是对

象），所以返回 `false` 。

此外，对于 `undefined` 和 `null` ， `instanceOf` 运算符总是返回 `false` 。

```
undefined instanceof Object // false
null instanceof Object // false
```

利用 `instanceof` 运算符，还可以巧妙地解决，调用构造函数时，忘了加 `new` 命令的问题。

```
function Fubar (foo, bar) {
  if (this instanceof Fubar) {
    this._foo = foo;
    this._bar = bar;
  } else {
    return new Fubar(foo, bar);
  }
}
```

上面代码使用 `instanceof` 运算符，在函数体内部判断 `this` 关键字是否为构造函数 `Fubar` 的实例。如果不是，就表明忘了加 `new` 命令。

构造函数的继承

让一个构造函数继承另一个构造函数，是非常常见的需求。这可以分成两步实现。第一步是在子类的构造函数中，调用父类的构造函数。

```
function Sub(value) {
  Super.call(this);
  this.prop = value;
}
```

上面代码中， `Sub` 是子类的构造函数， `this` 是子类的实例。在实例

上调用父类的构造函数 `Super` ，就会让子类实例具有父类实例的属性。

第二步，是让子类的原型指向父类的原型，这样子类就可以继承父类原型。

```
Sub.prototype = Object.create(Super.prototype);
Sub.prototype.constructor = Sub;
Sub.prototype.method = '...';
```

上面代码中， `Sub.prototype` 是子类的原型，要将它赋值为 `Object.create(Super.prototype)` ，而不是直接等于 `Super.prototype` 。否则后面两行对 `Sub.prototype` 的操作，会连父类的原型 `Super.prototype` 一起修改掉。

另外一种写法是 `Sub.prototype` 等于一个父类实例。

```
Sub.prototype = new Super();
```

上面这种写法也有继承的效果，但是子类会具有父类实例的方法。有时，这可能不是我们需要的，所以不推荐使用这种写法。

举例来说，下面是一个 `Shape` 构造函数。

```
function Shape() {
  this.x = 0;
  this.y = 0;
}

Shape.prototype.move = function (x, y) {
  this.x += x;
  this.y += y;
  console.info('Shape moved.');
};
```

我们需要让 `Rectangle` 构造函数继承 `Shape` 。

```
// 第一步，子类继承父类的实例
function Rectangle() {
  Shape.call(this); // 调用父类构造函数
}
// 另一种写法
function Rectangle() {
  this.base = Shape;
  this.base();
}

// 第二步，子类继承父类的原型
Rectangle.prototype = Object.create(Shape.prototype);
Rectangle.prototype.constructor = Rectangle;
```

采用这样的写法以后， `instanceof` 运算符会对子类和父类的构造函数，都返回 `true` 。

```
var rect = new Rectangle();

rect instanceof Rectangle  // true
rect instanceof Shape  // true
```

上面代码中，子类是整体继承父类。有时只需要单个方法的继承，这时可以采用下面的写法。

```
ClassB.prototype.print = function() {
  ClassA.prototype.print.call(this);
  // some code
}
```

上面代码中，子类 `B` 的 `print` 方法先调用父类 `A` 的 `print` 方法，再部署自己的代码。这就等于继承了父类 `A` 的 `print` 方法。

多重继承

JavaScript 不提供多重继承功能，即不允许一个对象同时继承多个对象。但是，可以通过变通方法，实现这个功能。

```
function M1() {
  this.hello = 'hello';
}

function M2() {
  this.world = 'world';
}

function S() {
  M1.call(this);
  M2.call(this);
}

// 继承 M1
S.prototype = Object.create(M1.prototype);
// 继承链上加入 M2
Object.assign(S.prototype, M2.prototype);

// 指定构造函数
S.prototype.constructor = S;

var s = new S();
s.hello // 'hello：'
s.world // 'world'
```

上面代码中，子类 `S` 同时继承了父类 `M1` 和 `M2` 。这种模式又称为 Mixin（混入）。

模块

随着网站逐渐变成”互联网应用程序”，嵌入网页的 JavaScript 代

码越来越庞大，越来越复杂。网页越来越像桌面程序，需要一个团队分工协作、进度管理、单元测试等等……开发者必须使用软件工程的方法，管理网页的业务逻辑。

JavaScript 模块化编程，已经成为一个迫切的需求。理想情况下，开发者只需要实现核心的业务逻辑，其他都可以加载别人已经写好的模块。

：”污染”了全局变量，无法保证不与其他模块发生变量名冲突，而且模块成员之间看不出直接关系。

为了解决上面的缺点，可以
但是，JavaScript 不是一种模块化编程语言，ES6 才开始支持“类”和“模块”。下面介绍传统的做法，如何利用对象实现模块的效果。

基本的实现方法

模块是实现特定功能的一组属性和方法的封装。

简单的做法是把模块写成一个对象，所有的模块成员都放到这个对象里面。

```
var module1 = new Object({
  _count : 0,
  m1 : function (){
    //...
  },
  m2 : function (){
    //...
  }
});
```

上面的函数 `m1` 和 `m2` ，都封装在 `module1` 对象里。使用的时候，就

是调用这个对象的属性。

```
1. module1.m1();
```

但是，这样的写法会暴露所有模块成员，内部状态可以被外部改写。比如，外部代码可以直接改变内部计数器的值。

```
1. module1._count = 5;
```

封装私有变量：构造函数的写法

我们可以利用构造函数，封装私有变量。

```
1. function StringBuilder() {
2.   var buffer = [];
3.
4.   this.add = function (str) {
5.      buffer.push(str);
6.   };
7.
8.   this.toString = function () {
9.     return buffer.join('');
10.  };
11.
12. }
```

上面代码中，`buffer`是模块的私有变量。一旦生成实例对象，外部是无法直接访问`buffer`的。但是，这种方法将私有变量封装在构造函数中，违反了构造函数与实例对象相分离的原则。并且，非常耗费内存。

```
1. function StringBuilder() {
2.   this._buffer = [];
3. }
4.
5. StringBuilder.prototype = {
```

```
  constructor: StringBuilder,
  add: function (str) {
    this._buffer.push(str);
  },
  toString: function () {
    return this._buffer.join('');
  }
};
```

这种方法将私有变量放入实例对象中，好处是看上去更自然，但是它的私有变量可以从外部读写，不是很安全。

封装私有变量：立即执行函数的写法

另一种做法是使用“立即执行函数”（Immediately-Invoked Function Expression，IIFE），将相关的属性和方法封装在一个函数作用域里面，可以达到不暴露私有成员的目的。

```
var module1 = (function () {
  var _count = 0;
  var m1 = function () {
    //...
  };
  var m2 = function () {
    //...
  };
  return {
    m1 : m1,
    m2 : m2
  };
})();
```

使用上面的写法，外部代码无法读取内部的 `_count` 变量。

```
console.info(module1._count); //undefined
```

上面的 `module1` 就是 JavaScript 模块的基本写法。下面，再对这种写法进行加工。

模块的放大模式

如果一个模块很大，必须分成几个部分，或者一个模块需要继承另一个模块，这时就有必要采用“放大模式”（augmentation）。

```
var module1 = (function (mod){
  mod.m3 = function () {
    //...
  };
  return mod;
})(module1);
```

上面的代码为 `module1` 模块添加了一个新方法 `m3()` ，然后返回新的 `module1` 模块。

在浏览器环境中，模块的各个部分通常都是从网上获取的，有时无法知道哪个部分会先加载。如果采用上面的写法，第一个执行的部分有可能加载一个不存在空对象，这时就要采用”宽放大模式”（Loose augmentation）。

```
var module1 = (function (mod) {
  //...
  return mod;
})(window.module1 || {});
```

与”放大模式”相比，“宽放大模式”就是“立即执行函数”的参数可以是空对象。

输入全局变量

独立性是模块的重要特点，模块内部最好不与程序的其他部分直接交互。

为了在模块内部调用全局变量，必须显式地将其他变量输入模块。

```
var module1 = (function ($, YAHOO) {
  //...
})(jQuery, YAHOO);
```

上面的 `module1` 模块需要使用 jQuery 库和 YUI 库，就把这两个库（其实是两个模块）当作参数输入 `module1` 。这样做除了保证模块的独立性，还使得模块之间的依赖关系变得明显。

立即执行函数还可以起到命名空间的作用。

```
(function($, window, document) {

  function go(num) {
  }

  function handleEvents() {
  }

  function initialize() {
  }

  function dieCarouselDie() {
  }

  //attach to the global scope
  window.finalCarousel = {
    init : initialize,
    destroy : dieCouraselDie
  }

})( jQuery, window, document );
```

上面代码中，`finalCarousel`对象输出到全局，对外暴露`init`和`destroy`接口，内部方法`go`、`handleEvents`、`initialize`、`dieCarouselDie`都是外部无法调用的。

参考链接

- JavaScript Modules: A Beginner’s Guide, by Preethi Kasireddy

Object 对象的相关方法

Object 对象的相关方法

JavaScript 在 `Object` 对象上面，提供了很多相关方法，处理面向对象编程的相关操作。本章介绍这些方法。

Object.getPrototypeOf()

`Object.getPrototypeOf` 方法返回参数对象的原型。这是获取原型对象的标准方法。

```
var F = function () {};
var f = new F();
Object.getPrototypeOf(f) === F.prototype // true
```

上面代码中，实例对象 `f` 的原型是 `F.prototype` 。

下面是几种特殊对象的原型。

```
// 空对象的原型是 Object.prototype
Object.getPrototypeOf({}) === Object.prototype // true

// Object.prototype 的原型是 null
Object.getPrototypeOf(Object.prototype) === null // true

// 函数的原型是 Function.prototype
function f() {}
Object.getPrototypeOf(f) === Function.prototype // true
```

Object.setPrototypeOf()

`Object.setPrototypeOf` 方法为参数对象设置原型，返回该参数对象。它接受两个参数，第一个是现有对象，第二个是原型对象。

```
var a = {};
var b = {x: 1};
Object.setPrototypeOf(a, b);

Object.getPrototypeOf(a) === b
a.x // 1
```

上面代码中，`Object.setPrototypeOf` 方法将对象 `a` 的原型，设置为对象 `b`，因此 `a` 可以共享 `b` 的属性。

`new` 命令可以使用 `Object.setPrototypeOf` 方法模拟。

```
var F = function () {
  this.foo = 'bar';
};

var f = new F();
// 等同于
```

```
var f = Object.setPrototypeOf({}, F.prototype);
F.call(f);
```

上面代码中，`new` 命令新建实例对象，其实可以分成两步。第一步，将一个空对象的原型设为构造函数的 `prototype` 属性（上例是 `F.prototype`）；第二步，将构造函数内部的 `this` 绑定这个空对象，然后执行构造函数，使得定义在 `this` 上面的方法和属性（上例是 `this.foo`），都转移到这个空对象上。

Object.create()

生成实例对象的常用方法是，使用 `new` 命令让构造函数返回一个实例。但是很多时候，只能拿到一个实例对象，它可能根本不是由构建函数生成的，那么能不能从一个实例对象，生成另一个实例对象呢？

JavaScript 提供了 `Object.create` 方法，用来满足这种需求。该方法接受一个对象作为参数，然后以它为原型，返回一个实例对象。该实例完全继承原型对象的属性。

```
// 原型对象
var A = {
  print: function () {
    console.log('hello');
  }
};

// 实例对象
var B = Object.create(A);

Object.getPrototypeOf(B) === A // true
B.print() // hello
B.print === A.print // true
```

上面代码中，`Object.create` 方法以 `A` 对象为原型，生成了 `B` 对

象。 `B` 继承了 `A` 的所有属性和方法。

实际上， `Object.create` 方法可以用下面的代码代替。

```
if (typeof Object.create !== 'function') {
  Object.create = function (obj) {
    function F() {}
    F.prototype = obj;
    return new F();
  };
}
```

上面代码表明， `Object.create` 方法的实质是新建一个空的构造函数 `F` ，然后让 `F.prototype` 属性指向参数对象 `obj` ，最后返回一个 `F` 的实例，从而实现让该实例继承 `obj` 的属性。

下面三种方式生成的新对象是等价的。

```
var obj1 = Object.create({});
var obj2 = Object.create(Object.prototype);
var obj3 = new Object();
```

如果想要生成一个不继承任何属性（比如没有 `toString` 和 `valueOf` 方法）的对象，可以将 `Object.create` 的参数设为 `null` 。

```
var obj = Object.create(null);

obj.valueOf()
// TypeError: Object [object Object] has no method 'valueOf'
```

上面代码中，对象 `obj` 的原型是 `null` ，它就不具备一些定义在 `Object.prototype` 对象上面的属性，比如 `valueOf` 方法。

使用 `Object.create` 方法的时候，必须提供对象原型，即参数不能为空，或者不是对象，否则会报错。

```
Object.create()
// TypeError: Object prototype may only be an Object or null
Object.create(123)
// TypeError: Object prototype may only be an Object or null
```

`object.create` 方法生成的新对象，动态继承了原型。在原型上添加或修改任何方法，会立刻反映在新对象之上。

```
var obj1 = { p: 1 };
var obj2 = Object.create(obj1);

obj1.p = 2;
obj2.p // 2
```

上面代码中，修改对象原型 `obj1` 会影响到实例对象 `obj2` 。

除了对象的原型， `Object.create` 方法还可以接受第二个参数。该参数是一个属性描述对象，它所描述的对象属性，会添加到实例对象，作为该对象自身的属性。

```
var obj = Object.create({}, {
  p1: {
    value: 123,
    enumerable: true,
    configurable: true,
    writable: true,
  },
  p2: {
    value: 'abc',
    enumerable: true,
    configurable: true,
    writable: true,
  }
});

// 等同于
```

```
var obj = Object.create({});
obj.p1 = 123;
obj.p2 = 'abc';
```

`Object.create` 方法生成的对象，继承了它的原型对象的构造函数。

```
function A() {}
var a = new A();
var b = Object.create(a);

b.constructor === A // true
b instanceof A // true
```

上面代码中， `b` 对象的原型是 `a` 对象，因此继承了 `a` 对象的构造函数 `A` 。

Object.prototype.isPrototypeOf()

实例对象的 `isPrototypeOf` 方法，用来判断该对象是否为参数对象的原型。

```
var o1 = {};
var o2 = Object.create(o1);
var o3 = Object.create(o2);

o2.isPrototypeOf(o3) // true
o1.isPrototypeOf(o3) // true
```

上面代码中， `o1` 和 `o2` 都是 `o3` 的原型。这表明只要实例对象处在参数对象的原型链上， `isPrototypeOf` 方法都返回 `true` 。

```
Object.prototype.isPrototypeOf({}) // true
Object.prototype.isPrototypeOf([]) // true
Object.prototype.isPrototypeOf(/xyz/) // true
Object.prototype.isPrototypeOf(Object.create(null)) // false
```

上面代码中，由于 `Object.prototype` 处于原型链的最顶端，所以对各种实例都返回 `true` ，只有直接继承自 `null` 的对象除外。

Object.prototype.__proto__

实例对象的 `__proto__` 属性（前后各两个下划线），返回该对象的原型。该属性可读写。

```
var obj = {};
var p = {};

obj.__proto__ = p;
Object.getPrototypeOf(obj) === p // true
```

上面代码通过 `__proto__` 属性，将 `p` 对象设为 `obj` 对象的原型。

根据语言标准， `__proto__` 属性只有浏览器才需要部署，其他环境可以没有这个属性。它前后的两根下划线，表明它本质是一个内部属性，不应该对使用者暴露。因此，应该尽量少用这个属性，而是用 `Object.getPrototypeof()` 和 `Object.setPrototypeOf()` ，进行原型对象的读写操作。

原型链可以用 `__proto__` 很直观地表示。

```
var A = {
  name: '张三'
};
var B = {
  name: '李四'
};

var proto = {
  print: function () {
    console.log(this.name);
```

```
  }
};

A.__proto__ = proto;
B.__proto__ = proto;

A.print() // 张三
B.print() // 李四

A.print === B.print // true
A.print === proto.print // true
B.print === proto.print // true
```

上面代码中，`A` 对象和 `B` 对象的原型都是 `proto` 对象，它们都共享 `proto` 对象的 `print` 方法。也就是说，`A` 和 `B` 的 `print` 方法，都是在调用 `proto` 对象的 `print` 方法。

获取原型对象方法的比较

如前所述，`__proto__` 属性指向当前对象的原型对象，即构造函数的 `prototype` 属性。

```
var obj = new Object();

obj.__proto__ === Object.prototype
// true
obj.__proto__ === obj.constructor.prototype
// true
```

上面代码首先新建了一个对象 `obj`，它的 `__proto__` 属性，指向构造函数（`Object` 或 `obj.constructor`）的 `prototype` 属性。

因此，获取实例对象 `obj` 的原型对象，有三种方法。

- `obj.__proto__`

- `obj.constructor.prototype`
- `Object.getPrototypeOf(obj)`

上面三种方法之中，前两种都不是很可靠。`__proto__` 属性只有浏览器才需要部署，其他环境可以不部署。而 `obj.constructor.prototype` 在手动改变原型对象时，可能会失效。

```
var P = function () {};
var p = new P();

var C = function () {};
C.prototype = p;
var c = new C();

c.constructor.prototype === p // false
```

上面代码中，构造函数 `C` 的原型对象被改成了 `p`，但是实例对象的 `c.constructor.prototype` 却没有指向 `p`。所以，在改变原型对象时，一般要同时设置 `constructor` 属性。

```
C.prototype = p;
C.prototype.constructor = C;

var c = new C();
c.constructor.prototype === p // true
```

因此，推荐使用第三种 `Object.getPrototypeOf` 方法，获取原型对象。

Object.getOwnPropertyNames()

`Object.getOwnPropertyNames` 方法返回一个数组，成员是参数对象本身的所有属性的键名，不包含继承的属性键名。

```
Object.getOwnPropertyNames(Date)
```

```
2. // ["parse", "arguments", "UTC", "caller", "name", "prototype",
   "now", "length"]
```

上面代码中，`Object.getOwnPropertyNames` 方法返回 `Date` 所有自身的属性名。

对象本身的属性之中，有的是可以遍历的（enumerable），有的是不可以遍历的。`Object.getOwnPropertyNames` 方法返回所有键名，不管是否可以遍历。只获取那些可以遍历的属性，使用 `Object.keys` 方法。

```
1. Object.keys(Date) // []
```

上面代码表明，`Date` 对象所有自身的属性，都是不可以遍历的。

Object.prototype.hasOwnProperty()

对象实例的 `hasOwnProperty` 方法返回一个布尔值，用于判断某个属性定义在对象自身，还是定义在原型链上。

```
1. Date.hasOwnProperty('length') // true
2. Date.hasOwnProperty('toString') // false
```

上面代码表明，`Date.length`（构造函数 `Date` 可以接受多少个参数）是 `Date` 自身的属性，`Date.toString` 是继承的属性。

另外，`hasOwnProperty` 方法是 JavaScript 之中唯一一个处理对象属性时，不会遍历原型链的方法。

in 运算符和 for…in 循环

`in` 运算符返回一个布尔值，表示一个对象是否具有某个属性。它不区分该属性是对象自身的属性，还是继承的属性。

```
'length' in Date // true
'toString' in Date // true
```

`in` 运算符常用于检查一个属性是否存在。

获得对象的所有可遍历属性（不管是自身的还是继承的），可以使用 `for...in` 循环。

```
var o1 = { p1: 123 };

var o2 = Object.create(o1, {
  p2: { value: "abc", enumerable: true }
});

for (p in o2) {
  console.info(p);
}
// p2
// p1
```

上面代码中，对象 `o2` 的 `p2` 属性是自身的， `p1` 属性是继承的。这两个属性都会被 `for...in` 循环遍历。

为了在 `for...in` 循环中获得对象自身的属性，可以采用 `hasOwnProperty` 方法判断一下。

```
for ( var name in object ) {
  if ( object.hasOwnProperty(name) ) {
    /* loop code */
  }
}
```

获得对象的所有属性（不管是自身的还是继承的，也不管是否可枚举），可以使用下面的函数。

```
function inheritedPropertyNames(obj) {
  var props = {};
  while(obj) {
    Object.getOwnPropertyNames(obj).forEach(function(p) {
      props[p] = true;
    });
    obj = Object.getPrototypeOf(obj);
  }
  return Object.getOwnPropertyNames(props);
}
```

上面代码依次获取 `obj` 对象的每一级原型对象“自身”的属性，从而获取 `Obj` 对象的“所有”属性，不管是否可遍历。

下面是一个例子，列出 `Date` 对象的所有属性。

```
inheritedPropertyNames(Date)
// [
//  "caller",
//  "constructor",
//  "toString",
//  "UTC",
//  ...
// ]
```

对象的拷贝

如果要拷贝一个对象，需要做到下面两件事情。

- 确保拷贝后的对象，与原对象具有同样的原型。
- 确保拷贝后的对象，与原对象具有同样的实例属性。

下面就是根据上面两点，实现的对象拷贝函数。

```
function copyObject(orig) {
```

```
  var copy = Object.create(Object.getPrototypeOf(orig));
  copyOwnPropertiesFrom(copy, orig);
  return copy;
}

function copyOwnPropertiesFrom(target, source) {
  Object
    .getOwnPropertyNames(source)
    .forEach(function (propKey) {
      var desc = Object.getOwnPropertyDescriptor(source, propKey);
      Object.defineProperty(target, propKey, desc);
    });
  return target;
}
```

另一种更简单的写法，是利用 ES2017 才引入标准的 `Object.getOwnPropertyDescriptors` 方法。

```
function copyObject(orig) {
  return Object.create(
    Object.getPrototypeOf(orig),
    Object.getOwnPropertyDescriptors(orig)
  );
}
```

参考链接

- Dr. Axel Rauschmayer, JavaScript properties: inheritance and enumerability

严格模式

严格模式

除了正常的运行模式，JavaScript 还有第二种运行模式：严格模式（strict mode）。顾名思义，这种模式采用更加严格的 JavaScript 语法。

同样的代码，在正常模式和严格模式中，可能会有不一样的运行结果。一些在正常模式下可以运行的语句，在严格模式下将不能运行。

设计目的

早期的 JavaScript 语言有很多设计不合理的地方，但是为了兼容以前的代码，又不能改变老的语法，只能不断添加新的语法，引导程序员使用新语法。

严格模式是从 ES5 进入标准的，主要目的有以下几个。

- 明确禁止一些不合理、不严谨的语法，减少 JavaScript 语言的一些怪异行为。
- 增加更多报错的场合，消除代码运行的一些不安全之处，保证代码运行的安全。
- 提高编译器效率，增加运行速度。
- 为未来新版本的 JavaScript 语法做好铺垫。

总之，严格模式体现了 JavaScript 更合理、更安全、更严谨的发展方向。

启用方法

进入严格模式的标志，是一行字符串 `use strict` 。

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

老版本的引擎会把它当作一行普通字符串，加以忽略。新版本的引擎就

会进入严格模式。

严格模式可以用于整个脚本，也可以只用于单个函数。

（1） 整个脚本文件

`use strict` 放在脚本文件的第一行，整个脚本都将以严格模式运行。如果这行语句不在第一行就无效，整个脚本会以正常模式运行。（严格地说，只要前面不是产生实际运行结果的语句， `use strict` 可以不在第一行，比如直接跟在一个空的分号后面，或者跟在注释后面。）

```
<script>
  'use strict';
  console.log('这是严格模式');
</script>

<script>
  console.log('这是正常模式');
</script>
```

上面代码中，一个网页文件依次有两段 JavaScript 代码。前一个 `<script>` 标签是严格模式，后一个不是。

如果 `use strict` 写成下面这样，则不起作用，严格模式必须从代码一开始就生效。

```
<script>
  console.log('这是正常模式');
  'use strict';
</script>
```

（2）单个函数

`use strict` 放在函数体的第一行，则整个函数以严格模式运行。

```
function strict() {
  'use strict';
  return '这是严格模式';
}

function strict2() {
  'use strict';
  function f() {
    return '这也是严格模式';
  }
  return f();
}

function notStrict() {
  return '这是正常模式';
}
```

有时，需要把不同的脚本合并在一个文件里面。如果一个脚本是严格模式，另一个脚本不是，它们的合并就可能出错。严格模式的脚本在前，则合并后的脚本都是严格模式；如果正常模式的脚本在前，则合并后的脚本都是正常模式。这两种情况下，合并后的结果都是不正确的。这时可以考虑把整个脚本文件放在一个立即执行的匿名函数之中。

```
(function () {
  'use strict';
  // some code here
})();
```

显式报错

严格模式使得 JavaScript 的语法变得更严格，更多的操作会显式报错。其中有些操作，在正常模式下只会默默地失败，不会报错。

只读属性不可写

严格模式下，设置字符串的 `length` 属性，会报错。

```
'use strict';
'abc'.length = 5;
// TypeError: Cannot assign to read only property 'length' of
string 'abc'
```

上面代码报错，因为 `length` 是只读属性，严格模式下不可写。正常模式下，改变 `length` 属性是无效的，但不会报错。

严格模式下，对只读属性赋值，或者删除不可配置（non-configurable）属性都会报错。

```
// 对只读属性赋值会报错
'use strict';
Object.defineProperty({}, 'a', {
  value: 37,
  writable: false
});
obj.a = 123;
// TypeError: Cannot assign to read only property 'a' of object #
<Object>

// 删除不可配置的属性会报错
'use strict';
var obj = Object.defineProperty({}, 'p', {
  value: 1,
  configurable: false
});
delete obj.p
// TypeError: Cannot delete property 'p' of #<Object>
```

只设置了取值器的属性不可写

严格模式下，对一个只有取值器（getter）、没有存值器（setter）

的属性赋值，会报错。

```
'use strict';
var obj = {
  get v() { return 1; }
};
obj.v = 2;
// Uncaught TypeError: Cannot set property v of #<Object> which has only a getter
```

上面代码中，`obj.v` 只有取值器，没有存值器，对它进行赋值就会报错。

禁止扩展的对象不可扩展

严格模式下，对禁止扩展的对象添加新属性，会报错。

```
'use strict';
var obj = {};
Object.preventExtensions(obj);
obj.v = 1;
// Uncaught TypeError: Cannot add property v, object is not extensible
```

上面代码中，`obj` 对象禁止扩展，添加属性就会报错。

eval、arguments 不可用作标识名

严格模式下，使用 `eval` 或者 `arguments` 作为标识名，将会报错。下面的语句都会报错。

```
'use strict';
var eval = 17;
var arguments = 17;
var obj = { set p(arguments) { } };
```

```
try { } catch (arguments) { }
function x(eval) { }
function arguments() { }
var y = function eval() { };
var f = new Function('arguments', "'use strict'; return 17;");
// SyntaxError: Unexpected eval or arguments in strict mode
```

函数不能有重名的参数

正常模式下，如果函数有多个重名的参数，可以用 `arguments[i]` 读取。严格模式下，这属于语法错误。

```
function f(a, a, b) {
  'use strict';
  return a + b;
}
// Uncaught SyntaxError: Duplicate parameter name not allowed in this context
```

禁止八进制的前缀0表示法

正常模式下，整数的第一位如果是 `0` ，表示这是八进制数，比如 `0100` 等于十进制的64。严格模式禁止这种表示法，整数第一位为 `0` ，将报错。

```
'use strict';
var n = 0100;
// Uncaught SyntaxError: Octal literals are not allowed in strict mode.
```

增强的安全措施

严格模式增强了安全保护，从语法上防止了一些不小心会出现的错误。

全局变量显式声明

正常模式中，如果一个变量没有声明就赋值，默认是全局变量。严格模式禁止这种用法，全局变量必须显式声明。

```
'use strict';

v = 1; // 报错，v未声明

for (i = 0; i < 2; i++) { // 报错，i 未声明
  // ...
}

function f() {
  x = 123;
}
f() // 报错，未声明就创建一个全局变量
```

因此，严格模式下，变量都必须先声明，然后再使用。

禁止 this 关键字指向全局对象

正常模式下，函数内部的 `this` 可能会指向全局对象，严格模式禁止这种用法，避免无意间创造全局变量。

```
// 正常模式
function f() {
  console.log(this === window);
}
f() // true

// 严格模式
function f() {
  'use strict';
  console.log(this === undefined);
}
```

```
f() // true
```

上面代码中，严格模式的函数体内部 `this` 是 `undefined` 。

这种限制对于构造函数尤其有用。使用构造函数时，有时忘了加 `new` ，这时 `this` 不再指向全局对象，而是报错。

```
function f() {
  'use strict';
  this.a = 1;
};

f();// 报错，this 未定义
```

严格模式下，函数直接调用时（不使用 `new` 调用），函数内部的 `this` 表示 `undefined` （未定义），因此可以用 `call` 、 `apply` 和 `bind` 方法，将任意值绑定在 `this` 上面。正常模式下， `this` 指向全局对象，如果绑定的值是非对象，将被自动转为对象再绑定上去，而 `null` 和 `undefined` 这两个无法转成对象的值，将被忽略。

```
// 正常模式
function fun() {
  return this;
}

fun() // window
fun.call(2) // Number {2}
fun.call(true) // Boolean {true}
fun.call(null) // window
fun.call(undefined) // window

// 严格模式
'use strict';
function fun() {
```

```
15.   return this;
16. }
17.
18. fun() //undefined
19. fun.call(2) // 2
20. fun.call(true) // true
21. fun.call(null) // null
22. fun.call(undefined) // undefined
```

上面代码中，可以把任意类型的值，绑定在 `this` 上面。

禁止使用 fn.callee、fn.caller

函数内部不得使用 `fn.caller` 、 `fn.arguments` ，否则会报错。这意味着不能在函数内部得到调用栈了。

```
1. function f1() {
2.   'use strict';
3.   f1.caller;    // 报错
4.   f1.arguments; // 报错
5. }
6.
7. f1();
```

禁止使用 arguments.callee、arguments.caller

`arguments.callee` 和 `arguments.caller` 是两个历史遗留的变量，从来没有标准化过，现在已经取消了。正常模式下调用它们没有什么作用，但是不会报错。严格模式明确规定，函数内部使用 `arguments.callee` 、 `arguments.caller` 将会报错。

```
1. 'use strict';
2. var f = function () {
```

```
  return arguments.callee;
};

f(); // 报错
```

禁止删除变量

严格模式下无法删除变量，如果使用 `delete` 命令删除一个变量，会报错。只有对象的属性，且属性的描述对象的 `configurable` 属性设置为 `true` ，才能被 `delete` 命令删除。

```
'use strict';
var x;
delete x; // 语法错误

var obj = Object.create(null, {
  x: {
    value: 1,
    configurable: true
  }
});
delete obj.x; // 删除成功
```

静态绑定

JavaScript 语言的一个特点，就是允许“动态绑定”，即某些属性和方法到底属于哪一个对象，不是在编译时确定的，而是在运行时（runtime）确定的。

严格模式对动态绑定做了一些限制。某些情况下，只允许静态绑定。也就是说，属性和方法到底归属哪个对象，必须在编译阶段就确定。这样做有利于编译效率的提高，也使得代码更容易阅读，更少出现意外。

具体来说，涉及以下几个方面。

禁止使用 with 语句

严格模式下，使用 `with` 语句将报错。因为 `with` 语句无法在编译时就确定，某个属性到底归属哪个对象，从而影响了编译效果。

```
'use strict';
var v  = 1;
var obj = {};

with (obj) {
  v = 2;
}
// Uncaught SyntaxError: Strict mode code may not include a with statement
```

创设 eval 作用域

正常模式下，JavaScript 语言有两种变量作用域（scope）：全局作用域和函数作用域。严格模式创设了第三种作用域： `eval` 作用域。

正常模式下， `eval` 语句的作用域，取决于它处于全局作用域，还是函数作用域。严格模式下， `eval` 语句本身就是一个作用域，不再能够在其所运行的作用域创设新的变量了，也就是说， `eval` 所生成的变量只能用于 `eval` 内部。

```
(function () {
  'use strict';
  var x = 2;
  console.log(eval('var x = 5; x')) // 5
  console.log(x) // 2
})()
```

上面代码中，由于 `eval` 语句内部是一个独立作用域，所以内部的变量 `x` 不会泄露到外部。

注意，如果希望 `eval` 语句也使用严格模式，有两种方式。

```
// 方式一
function f1(str){
  'use strict';
  return eval(str);
}
f1('undeclared_variable = 1'); // 报错

// 方式二
function f2(str){
  return eval(str);
}
f2('"use strict";undeclared_variable = 1')  // 报错
```

上面两种写法， `eval` 内部使用的都是严格模式。

arguments 不再追踪参数的变化

变量 `arguments` 代表函数的参数。严格模式下，函数内部改变参数与 `arguments` 的联系被切断了，两者不再存在联动关系。

```
function f(a) {
  a = 2;
  return [a, arguments[0]];
}
f(1); // 正常模式为[2, 2]

function f(a) {
  'use strict';
  a = 2;
  return [a, arguments[0]];
}
f(1); // 严格模式为[2, 1]
```

上面代码中，改变函数的参数，不会反应到 `arguments` 对象上来。

向下一个版本的 JavaScript 过渡

JavaScript语言的下一个版本是 ECMAScript 6，为了平稳过渡，严格模式引入了一些 ES6 语法。

非函数代码块不得声明函数

ES6 会引入块级作用域。为了与新版本接轨，ES5 的严格模式只允许在全局作用域或函数作用域声明函数。也就是说，不允许在非函数的代码块内声明函数。

```
'use strict';
if (true) {
  function f1() { } // 语法错误
}

for (var i = 0; i < 5; i++) {
  function f2() { } // 语法错误
}
```

上面代码在 `if` 代码块和 `for` 代码块中声明了函数，ES5 环境会报错。

注意，如果是 ES6 环境，上面的代码不会报错，因为 ES6 允许在代码块之中声明函数。

保留字

为了向将来 JavaScript 的新版本过渡，严格模式新增了一些保留字（implements、interface、let、package、private、protected、public、static、yield等）。使用这些词作为变量名将会报错。

```
function package(protected) { // 语法错误
  'use strict';
  var implements; // 语法错误
}
```

参考链接

- MDN, Strict mode
- Dr. Axel Rauschmayer, JavaScript: Why the hatred for strict mode?
- Dr. Axel Rauschmayer, JavaScript’s strict mode: a summary
- Douglas Crockford, Strict Mode Is Coming To Town
- JavaScript Strict Mode Support

异步操作

异步操作

概述

异步操作概述

单线程模型

单线程模型指的是，JavaScript 只在一个线程上运行。也就是说，JavaScript 同时只能执行一个任务，其他任务都必须在后面排队等待。

注意，JavaScript 只在一个线程上运行，不代表 JavaScript 引擎只有一个线程。事实上，JavaScript 引擎有多个线程，单个脚本只能在一个线程上运行（称为主线程），其他线程都是在后台配合。

JavaScript 之所以采用单线程，而不是多线程，跟历史有关系。JavaScript 从诞生起就是单线程，原因是不想让浏览器变得太复

杂，因为多线程需要共享资源、且有可能修改彼此的运行结果，对于一种网页脚本语言来说，这就太复杂了。如果 JavaScript 同时有两个线程，一个线程在网页 DOM 节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准？是不是还要有锁机制？所以，为了避免复杂性，JavaScript 一开始就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

这种模式的好处是实现起来比较简单，执行环境相对单纯；坏处是只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。常见的浏览器无响应（假死），往往就是因为某一段 JavaScript 代码长时间运行（比如死循环），导致整个页面卡在这个地方，其他任务无法执行。JavaScript 语言本身并不慢，慢的是读写外部数据，比如等待 Ajax 请求返回结果。这个时候，如果对方服务器迟迟没有响应，或者网络不通畅，就会导致脚本的长时间停滞。

如果排队是因为计算量大，CPU 忙不过来，倒也算了，但是很多时候 CPU 是闲着的，因为 IO 操作（输入输出）很慢（比如 Ajax 操作从网络读取数据），不得不等着结果出来，再往下执行。JavaScript 语言的设计者意识到，这时 CPU 完全可以不管 IO 操作，挂起处于等待中的任务，先运行排在后面的任务。等到 IO 操作返回了结果，再回过头，把挂起的任务继续执行下去。这种机制就是 JavaScript 内部采用的“事件循环”机制（Event Loop）。

单线程模型虽然对 JavaScript 构成了很大的限制，但也因此使它具备了其他语言不具备的优势。如果用得好，JavaScript 程序是不会出现堵塞的，这就是为什么 Node 可以用很少的资源，应付大流量访问的原因。

为了利用多核 CPU 的计算能力，HTML5 提出 Web Worker 标准，允许 JavaScript 脚本创建多个线程，但是子线程完全受主线程控

制，且不得操作 DOM。所以，这个新标准并没有改变 JavaScript 单线程的本质。

同步任务和异步任务

程序里面所有的任务，可以分成两类：同步任务（synchronous）和异步任务（asynchronous）。

同步任务是那些没有被引擎挂起、在主线程上排队执行的任务。只有前一个任务执行完毕，才能执行后一个任务。

异步任务是那些被引擎放在一边，不进入主线程、而进入任务队列的任务。只有引擎认为某个异步任务可以执行了（比如 Ajax 操作从服务器得到了结果），该任务（采用回调函数的形式）才会进入主线程执行。排在异步任务后面的代码，不用等待异步任务结束会马上运行，也就是说，异步任务不具有”堵塞“效应。

举例来说，Ajax 操作可以当作同步任务处理，也可以当作异步任务处理，由开发者决定。如果是同步任务，主线程就等着 Ajax 操作返回结果，再往下执行；如果是异步任务，主线程在发出 Ajax 请求以后，就直接往下执行，等到 Ajax 操作有了结果，主线程再执行对应的回调函数。

任务队列和事件循环

JavaScript 运行时，除了一个正在运行的主线程，引擎还提供一个任务队列（task queue），里面是各种需要当前程序处理的异步任务。（实际上，根据异步任务的类型，存在多个任务队列。为了方便理解，这里假设只存在一个队列。）

首先，主线程会去执行所有的同步任务。等到同步任务全部执行完，就

会去看任务队列里面的异步任务。如果满足条件，那么异步任务就重新进入主线程开始执行，这时它就变成同步任务了。等到执行完，下一个异步任务再进入主线程开始执行。一旦任务队列清空，程序就结束执行。

异步任务的写法通常是回调函数。一旦异步任务重新进入主线程，就会执行对应的回调函数。如果一个异步任务没有回调函数，就不会进入任务队列，也就是说，不会重新进入主线程，因为没有用回调函数指定下一步的操作。

JavaScript 引擎怎么知道异步任务有没有结果，能不能进入主线程呢？答案就是引擎在不停地检查，一遍又一遍，只要同步任务执行完了，引擎就会去检查那些挂起来的异步任务，是不是可以进入主线程了。这种循环检查的机制，就叫做事件循环（Event Loop）。维基百科的定义是：“事件循环是一个程序结构，用于等待和发送消息和事件（a programming construct that waits for and dispatches events or messages in a program）”。

异步操作的模式

下面总结一下异步操作的几种模式。

回调函数

回调函数是异步操作最基本的方法。

下面是两个函数 `f1` 和 `f2`，编程的意图是 `f2` 必须等到 `f1` 执行完成，才能执行。

```
function f1() {
  // ...
}
```

```
4.
5.  function f2() {
6.    // ...
7.  }
8.
9.  f1();
10. f2();
```

上面代码的问题在于，如果 `f1` 是异步操作， `f2` 会立即执行，不会等到 `f1` 结束再执行。

这时，可以考虑改写 `f1` ，把 `f2` 写成 `f1` 的回调函数。

```
1.  function f1(callback) {
2.    // ...
3.    callback();
4.  }
5.
6.  function f2() {
7.    // ...
8.  }
9.
10. f1(f2);
```

回调函数的优点是简单、容易理解和实现，缺点是不利于代码的阅读和维护，各个部分之间高度耦合)（coupling），使得程序结构混乱、流程难以追踪（尤其是多个回调函数嵌套的情况），而且每个任务只能指定一个回调函数。

事件监听

另一种思路是采用事件驱动模式。异步任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

还是以 `f1` 和 `f2` 为例。首先，为 `f1` 绑定一个事件（这里采用的

jQuery 的写法）。

```
f1.on('done', f2);
```

上面这行代码的意思是，当 `f1` 发生 `done` 事件，就执行 `f2`。然后，对 `f1` 进行改写：

```
function f1() {
  setTimeout(function () {
    // ...
    f1.trigger('done');
  }, 1000);
}
```

上面代码中，`f1.trigger('done')` 表示，执行完成后，立即触发 `done` 事件，从而开始执行 `f2`。

这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以"去耦合"（decoupling），有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。阅读代码的时候，很难看出主流程。

发布/订阅

事件完全可以理解成"信号"，如果存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"（publish）一个信号，其他任务可以向信号中心"订阅"（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做"发布/订阅模式"（publish-subscribe pattern），又称"观察者模式"（observer pattern）。

这个模式有多种实现，下面采用的是 Ben Alman 的 Tiny Pub/Sub，这是 jQuery 的一个插件。

首先，`f2` 向信号中心 `jQuery` 订阅 `done` 信号。

```
jQuery.subscribe('done', f2);
```

然后，`f1` 进行如下改写。

```
function f1() {
  setTimeout(function () {
    // ...
    jQuery.publish('done');
  }, 1000);
}
```

上面代码中，`jQuery.publish('done')` 的意思是，`f1` 执行完成后，向信号中心 `jQuery` 发布 `done` 信号，从而引发 `f2` 的执行。

`f2` 完成执行后，可以取消订阅（unsubscribe）。

```
jQuery.unsubscribe('done', f2);
```

这种方法的性质与“事件监听”类似，但是明显优于后者。因为可以通过查看“消息中心”，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

异步操作的流程控制

如果有多个异步操作，就存在一个流程控制的问题：如何确定异步操作执行的顺序，以及如何保证遵守这种顺序。

```
function async(arg, callback) {
  console.log('参数为 ' + arg +' , 1秒后返回结果');
  setTimeout(function () { callback(arg * 2); }, 1000);
}
```

上面代码的 `async` 函数是一个异步任务，非常耗时，每次执行需要1秒才能完成，然后再调用回调函数。

如果有六个这样的异步任务，需要全部完成后，才能执行最后的 `final` 函数。请问应该如何安排操作流程？

```
function final(value) {
  console.log('完成: ', value);
}

async(1, function(value){
  async(value, function(value){
    async(value, function(value){
      async(value, function(value){
        async(value, function(value){
          async(value, final);
        });
      });
    });
  });
});
```

上面代码中，六个回调函数的嵌套，不仅写起来麻烦，容易出错，而且难以维护。

串行执行

我们可以编写一个流程控制函数，让它来控制异步任务，一个任务完成以后，再执行另一个。这就叫串行执行。

```
var items = [ 1, 2, 3, 4, 5, 6 ];
var results = [];

function async(arg, callback) {
  console.log('参数为 ' + arg +' , 1秒后返回结果');
```

```
  setTimeout(function () { callback(arg * 2); }, 1000);
}

function final(value) {
  console.log('完成: ', value);
}

function series(item) {
  if(item) {
    async( item, function(result) {
      results.push(result);
      return series(items.shift());
    });
  } else {
    return final(results[results.length - 1]);
  }
}

series(items.shift());
```

上面代码中，函数 `series` 就是串行函数，它会依次执行异步任务，所有任务都完成后，才会执行 `final` 函数。 `items` 数组保存每一个异步任务的参数， `results` 数组保存每一个异步任务的运行结果。

注意，上面的写法需要六秒，才能完成整个脚本。

并行执行

流程控制函数也可以是并行执行，即所有异步任务同时执行，等到全部完成以后，才执行 `final` 函数。

```
var items = [ 1, 2, 3, 4, 5, 6 ];
var results = [];

function async(arg, callback) {
  console.log('参数为 ' + arg +' , 1秒后返回结果');
```

```
  setTimeout(function () { callback(arg * 2); }, 1000);
}

function final(value) {
  console.log('完成: ', value);
}

items.forEach(function(item) {
  async(item, function(result){
    results.push(result);
    if(results.length === items.length) {
      final(results[results.length - 1]);
    }
  })
});
```

上面代码中，`forEach`方法会同时发起六个异步任务，等到它们全部完成以后，才会执行`final`函数。

相比而言，上面的写法只要一秒，就能完成整个脚本。这就是说，并行执行的效率较高，比起串行执行一次只能执行一个任务，较为节约时间。但是问题在于如果并行的任务较多，很容易耗尽系统资源，拖慢运行速度。因此有了第三种流程控制方式。

并行与串行的结合

所谓并行与串行的结合，就是设置一个门槛，每次最多只能并行执行`n`个异步任务，这样就避免了过分占用系统资源。

```
var items = [ 1, 2, 3, 4, 5, 6 ];
var results = [];
var running = 0;
var limit = 2;

function async(arg, callback) {
  console.log('参数为 ' + arg +' , 1秒后返回结果');
```

```
  setTimeout(function () { callback(arg * 2); }, 1000);
}

function final(value) {
  console.log('完成: ', value);
}

function launcher() {
  while(running < limit && items.length > 0) {
    var item = items.shift();
    async(item, function(result) {
      results.push(result);
      running--;
      if(items.length > 0) {
        launcher();
      } else if(running == 0) {
        final(results);
      }
    });
    running++;
  }
}

launcher();
```

上面代码中，最多只能同时运行两个异步任务。变量 `running` 记录当前正在运行的任务数，只要低于门槛值，就再启动一个新的任务，如果等于 `0` ，就表示所有任务都执行完了，这时就执行 `final` 函数。

这段代码需要三秒完成整个脚本，处在串行执行和并行执行之间。通过调节 `limit` 变量，达到效率和资源的最佳平衡。

定时器

定时器

JavaScript 提供定时执行代码的功能，叫做定时器（timer），主要由 `setTimeout()` 和 `setInterval()` 这两个函数来完成。它们向任务队列添加定时任务。

setTimeout()

`setTimeout` 函数用来指定某个函数或某段代码，在多少毫秒之后执行。它返回一个整数，表示定时器的编号，以后可以用来取消这个定时器。

```
var timerId = setTimeout(func|code, delay);
```

上面代码中，`setTimeout` 函数接受两个参数，第一个参数 `func|code` 是将要推迟执行的函数名或者一段代码，第二个参数 `delay` 是推迟执行的毫秒数。

```
console.log(1);
setTimeout('console.log(2)',1000);
console.log(3);
// 1
// 3
// 2
```

上面代码会先输出1和3，然后等待1000毫秒再输出2。注意，`console.log(2)`必须以字符串的形式，作为`setTimeout`的参数。

如果推迟执行的是函数，就直接将函数名，作为`setTimeout`的参数。

```
function f() {
  console.log(2);
}

setTimeout(f, 1000);
```

`setTimeout`的第二个参数如果省略，则默认为0。

```
setTimeout(f)
// 等同于
setTimeout(f, 0)
```

除了前两个参数，`setTimeout`还允许更多的参数。它们将依次传入推迟执行的函数（回调函数）。

```
setTimeout(function (a,b) {
  console.log(a + b);
}, 1000, 1, 1);
```

上面代码中，`setTimeout`共有4个参数。最后那两个参数，将在1000毫秒之后回调函数执行时，作为回调函数的参数。

还有一个需要注意的地方，如果回调函数是对象的方法，那

么 `setTimeout` 使得方法内部的 `this` 关键字指向全局环境，而不是定义时所在的那个对象。

```
var x = 1;

var obj = {
  x: 2,
  y: function () {
    console.log(this.x);
  }
};

setTimeout(obj.y, 1000) // 1
```

上面代码输出的是1，而不是2。因为当 `obj.y` 在1000毫秒后运行时， `this` 所指向的已经不是 `obj` 了，而是全局环境。

为了防止出现这个问题，一种解决方法是将 `obj.y` 放入一个函数。

```
var x = 1;

var obj = {
  x: 2,
  y: function () {
    console.log(this.x);
  }
};

setTimeout(function () {
  obj.y();
}, 1000);
// 2
```

上面代码中， `obj.y` 放在一个匿名函数之中，这使得 `obj.y` 在 `obj` 的作用域执行，而不是在全局作用域内执行，所以能够显示正确的值。

另一种解决方法是，使用 `bind` 方法，将 `obj.y` 这个方法绑定在 `obj` 上面。

```
var x = 1;

var obj = {
  x: 2,
  y: function () {
    console.log(this.x);
  }
};

setTimeout(obj.y.bind(obj), 1000)
// 2
```

setInterval()

`setInterval` 函数的用法与 `setTimeout` 完全一致，区别仅仅在于 `setInterval` 指定某个任务每隔一段时间就执行一次，也就是无限次的定时执行。

```
var i = 1
var timer = setInterval(function() {
  console.log(2);
}, 1000)
```

上面代码中，每隔1000毫秒就输出一个2，会无限运行下去，直到关闭当前窗口。

与 `setTimeout` 一样，除了前两个参数， `setInterval` 方法还可以接受更多的参数，它们会传入回调函数。

下面是一个通过 `setInterval` 方法实现网页动画的例子。

```
var div = document.getElementById('someDiv');
var opacity = 1;
var fader = setInterval(function() {
  opacity -= 0.1;
  if (opacity >= 0) {
    div.style.opacity = opacity;
  } else {
    clearInterval(fader);
  }
}, 100);
```

上面代码每隔100毫秒，设置一次 `div` 元素的透明度，直至其完全透明为止。

`setInterval` 的一个常见用途是实现轮询。下面是一个轮询 URL 的 Hash 值是否发生变化的例子。

```
var hash = window.location.hash;
var hashWatcher = setInterval(function() {
  if (window.location.hash != hash) {
    updatePage();
  }
}, 1000);
```

`setInterval` 指定的是“开始执行”之间的间隔，并不考虑每次任务执行本身所消耗的时间。因此实际上，两次执行之间的间隔会小于指定的时间。比如，`setInterval` 指定每 100ms 执行一次，每次执行需要5ms，那么第一次执行结束后95毫秒，第二次执行就会开始。如果某次执行耗时特别长，比如需要105毫秒，那么它结束后，下一次执行就会立即开始。

为了确保两次执行之间有固定的间隔，可以不用 `setInterval`，而是每次执行结束后，使用 `setTimeout` 指定下一次执行的具体时间。

```
var i = 1;
var timer = setTimeout(function f() {
  // ...
  timer = setTimeout(f, 2000);
}, 2000);
```

上面代码可以确保，下一次执行总是在本次执行结束之后的2000毫秒开始。

clearTimeout(), clearInterval()

`setTimeout` 和 `setInterval` 函数，都返回一个整数值，表示计数器编号。将该整数传入 `clearTimeout` 和 `clearInterval` 函数，就可以取消对应的定时器。

```
var id1 = setTimeout(f, 1000);
var id2 = setInterval(f, 1000);

clearTimeout(id1);
clearInterval(id2);
```

上面代码中，回调函数 `f` 不会再执行了，因为两个定时器都被取消了。

`setTimeout` 和 `setInterval` 返回的整数值是连续的，也就是说，第二个 `setTimeout` 方法返回的整数值，将比第一个的整数值大1。

```
function f() {}
setTimeout(f, 1000) // 10
setTimeout(f, 1000) // 11
setTimeout(f, 1000) // 12
```

上面代码中，连续调用三次 `setTimeout` ，返回值都比上一次大了1。

利用这一点，可以写一个函数，取消当前所有的 `setTimeout` 定时器。

```
(function() {
  var gid = setInterval(clearAllTimeouts, 0);

  function clearAllTimeouts() {
    var id = setTimeout(function() {}, 0);
    while (id > 0) {
      if (id !== gid) {
        clearTimeout(id);
      }
      id--;
    }
  }
})();
```

上面代码中，先调用 `setTimeout` ，得到一个计算器编号，然后把编号比它小的计数器全部取消。

实例：debounce 函数

有时，我们不希望回调函数被频繁调用。比如，用户填入网页输入框的内容，希望通过 Ajax 方法传回服务器，jQuery 的写法如下。

```
$('textarea').on('keydown', ajaxAction);
```

这样写有一个很大的缺点，就是如果用户连续击键，就会连续触发 `keydown` 事件，造成大量的 Ajax 通信。这是不必要的，而且很可能产生性能问题。正确的做法应该是，设置一个门槛值，表示两次 Ajax 通信的最小间隔时间。如果在间隔时间内，发生新的 `keydown` 事件，则不触发 Ajax 通信，并且重新开始计时。如果过了指定时间，没有发生新的 `keydown` 事件，再将数据发送出去。

这种做法叫做 debounce（防抖动）。假定两次 Ajax 通信的间隔不

得小于2500毫秒，上面的代码可以改写成下面这样。

```
$('textarea').on('keydown', debounce(ajaxAction, 2500));

function debounce(fn, delay){
  var timer = null; // 声明计时器
  return function() {
    var context = this;
    var args = arguments;
    clearTimeout(timer);
    timer = setTimeout(function () {
      fn.apply(context, args);
    }, delay);
  };
}
```

上面代码中，只要在2500毫秒之内，用户再次击键，就会取消上一次的定时器，然后再新建一个定时器。这样就保证了回调函数之间的调用间隔，至少是2500毫秒。

运行机制

`setTimeout` 和 `setInterval` 的运行机制，是将指定的代码移出本轮事件循环，等到下一轮事件循环，再检查是否到了指定时间。如果到了，就执行对应的代码；如果不到，就继续等待。

这意味着， `setTimeout` 和 `setInterval` 指定的回调函数，必须等到本轮事件循环的所有同步任务都执行完，才会开始执行。由于前面的任务到底需要多少时间执行完，是不确定的，所以没有办法保证， `setTimeout` 和 `setInterval` 指定的任务，一定会按照预定时间执行。

```
setTimeout(someTask, 100);
veryLongTask();
```

上面代码的 `setTimeout` ，指定100毫秒以后运行一个任务。但是，如果后面的 `veryLongTask` 函数（同步任务）运行时间非常长，过了100毫秒还无法结束，那么被推迟运行的 `someTask` 就只有等着，等到 `veryLongTask` 运行结束，才轮到它执行。

再看一个 `setInterval` 的例子。

```
setInterval(function () {
  console.log(2);
}, 1000);

sleep(3000);

function sleep(ms) {
  var start = Date.now();
  while ((Date.now() - start) < ms) {
  }
}
```

上面代码中， `setInterval` 要求每隔1000毫秒，就输出一个2。但是，紧接着的 `sleep` 语句需要3000毫秒才能完成，那么 `setInterval` 就必须推迟到3000毫秒之后才开始生效。注意，生效后 `setInterval` 不会产生累积效应，即不会一下子输出三个2，而是只会输出一个2。

setTimeout(f, 0)

含义

`setTimeout` 的作用是将代码推迟到指定时间执行，如果指定时间为 `0` ，即 `setTimeout(f, 0)` ，那么会立刻执行吗？

答案是不会。因为上一节说过，必须要等到当前脚本的同步任务，全部

处理完以后，才会执行 `setTimeout` 指定的回调函数 `f` 。也就是说， `setTimeout(f, 0)` 会在下一轮事件循环一开始就执行。

```
setTimeout(function () {
  console.log(1);
}, 0);
console.log(2);
// 2
// 1
```

上面代码先输出 `2` ，再输出 `1` 。因为 `2` 是同步任务，在本轮事件循环执行，而 `1` 是下一轮事件循环执行。

总之， `setTimeout(f, 0)` 这种写法的目的是，尽可能早地执行 `f` ，但是并不能保证立刻就执行 `f` 。

应用

`setTimeout(f, 0)` 有几个非常重要的用途。它的一大应用是，可以调整事件的发生顺序。比如，网页开发中，某个事件先发生在子元素，然后冒泡到父元素，即子元素的事件回调函数，会早于父元素的事件回调函数触发。如果，想让父元素的事件回调函数先发生，就要用到 `setTimeout(f, 0)` 。

```
// HTML 代码如下
// <input type="button" id="myButton" value="click">

var input = document.getElementById('myButton');

input.onclick = function A() {
  setTimeout(function B() {
    input.value +=' input';
  }, 0)
};
```

```
11.
12. document.body.onclick = function C() {
13.   input.value += ' body'
14. };
```

上面代码在点击按钮后，先触发回调函数 `A` ，然后触发函数 `C` 。函数 `A` 中， `setTimeout` 将函数 `B` 推迟到下一轮事件循环执行，这样就起到了，先触发父元素的回调函数 `C` 的目的了。

另一个应用是，用户自定义的回调函数，通常在浏览器的默认动作之前触发。比如，用户在输入框输入文本， `keypress` 事件会在浏览器接收文本之前触发。因此，下面的回调函数是达不到目的的。

```
1. // HTML 代码如下
2. // <input type="text" id="input-box">
3.
4. document.getElementById('input-box').onkeypress = function (event)
   {
5.   this.value = this.value.toUpperCase();
6. }
```

上面代码想在用户每次输入文本后，立即将字符转为大写。但是实际上，它只能将本次输入前的字符转为大写，因为浏览器此时还没接收到新的文本，所以 `this.value` 取不到最新输入的那个字符。只有用 `setTimeout` 改写，上面的代码才能发挥作用。

```
1. document.getElementById('input-box').onkeypress = function() {
2.   var self = this;
3.   setTimeout(function() {
4.     self.value = self.value.toUpperCase();
5.   }, 0);
6. }
```

上面代码将代码放入 `setTimeout` 之中，就能使得它在浏览器接收到文

本之后触发。

由于 `setTimeout(f, 0)` 实际上意味着，将任务放到浏览器最早可得的空闲时段执行，所以那些计算量大、耗时长的任务，常常会被放到几个小部分，分别放到 `setTimeout(f, 0)` 里面执行。

```
var div = document.getElementsByTagName('div')[0];

// 写法一
for (var i = 0xA00000; i < 0xFFFFFF; i++) {
  div.style.backgroundColor = '#' + i.toString(16);
}

// 写法二
var timer;
var i=0x100000;

function func() {
  timer = setTimeout(func, 0);
  div.style.backgroundColor = '#' + i.toString(16);
  if (i++ == 0xFFFFFF) clearTimeout(timer);
}

timer = setTimeout(func, 0);
```

上面代码有两种写法，都是改变一个网页元素的背景色。写法一会造成浏览器“堵塞”，因为 JavaScript 执行速度远高于 DOM，会造成大量 DOM 操作“堆积”，而写法二就不会，这就是 `setTimeout(f, 0)` 的好处。

另一个使用这种技巧的例子是代码高亮的处理。如果代码块很大，一次性处理，可能会对性能造成很大的压力，那么将其分成一个个小块，一次处理一块，比如写成 `setTimeout(highlightNext, 50)` 的样子，性能压力就会减轻。

Promise 对象

Promise 对象

概述

Promise 对象是 JavaScript 的异步操作解决方案，为异步操作提供统一接口。它起到代理作用（proxy），充当异步操作与回调函数之间的中介，使得异步操作具备同步操作的接口。Promise 可以让异步操作写起来，就像在写同步操作的流程，而不必一层层地嵌套回调函数。

注意，本章只是 Promise 对象的简单介绍。为了避免与后续教程的重复，更完整的介绍请看《ES6 标准入门》的《Promise 对象》一章。

首先，Promise 是一个对象，也是一个构造函数。

```
1.  function f1(resolve, reject) {
```

```
  // 异步代码...
}

var p1 = new Promise(f1);
```

上面代码中，`Promise` 构造函数接受一个回调函数 `f1` 作为参数，`f1` 里面是异步操作的代码。然后，返回的 `p1` 就是一个 Promise 实例。

Promise 的设计思想是，所有异步任务都返回一个 Promise 实例。Promise 实例有一个 `then` 方法，用来指定下一步的回调函数。

```
var p1 = new Promise(f1);
p1.then(f2);
```

上面代码中，`f1` 的异步操作执行完成，就会执行 `f2`。

传统的写法可能需要把 `f2` 作为回调函数传入 `f1`，比如写成 `f1(f2)`，异步操作完成后，在 `f1` 内部调用 `f2`。Promise 使得 `f1` 和 `f2` 变成了链式写法。不仅改善了可读性，而且对于多层嵌套的回调函数尤其方便。

```
// 传统写法
step1(function (value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        // ...
      });
    });
  });
});

// Promise 的写法
(new Promise(step1))
```

```
14.    .then(step2)
15.    .then(step3)
16.    .then(step4);
```

从上面代码可以看到，采用 Promises 以后，程序流程变得非常清楚，十分易读。注意，为了便于理解，上面代码的 `Promise` 实例的生成格式，做了简化，真正的语法请参照下文。

总的来说，传统的回调函数写法使得代码混成一团，变得横向发展而不是向下发展。Promise 就是解决这个问题，使得异步流程可以写成同步流程。

Promise 原本只是社区提出的一个构想，一些函数库率先实现了这个功能。ECMAScript 6 将其写入语言标准，目前 JavaScript 原生支持 Promise 对象。

Promise 对象的状态

Promise 对象通过自身的状态，来控制异步操作。Promise 实例具有三种状态。

- 异步操作未完成（pending）
- 异步操作成功（fulfilled）
- 异步操作失败（rejected）

上面三种状态里面，`fulfilled` 和 `rejected` 合在一起称为 `resolved`（已定型）。

这三种的状态的变化途径只有两种。

- 从“未完成”到“成功”
- 从“未完成”到“失败”

一旦状态发生变化，就凝固了，不会再有新的状态变化。这也是 Promise 这个名字的由来，它的英语意思是“承诺”，一旦承诺成效，就不得再改变了。这也意味着，Promise 实例的状态变化只可能发生一次。

因此，Promise 的最终结果只有两种。

- 异步操作成功，Promise 实例传回一个值（value），状态变为 `fulfilled`。
- 异步操作失败，Promise 实例抛出一个错误（error），状态变为 `rejected`。

Promise 构造函数

JavaScript 提供原生的 `Promise` 构造函数，用来生成 Promise 实例。

```
var promise = new Promise(function (resolve, reject) {
  // ...

  if (/* 异步操作成功 */){
    resolve(value);
  } else { /* 异步操作失败 */
    reject(new Error());
  }
});
```

上面代码中，`Promise` 构造函数接受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`。它们是两个函数，由 JavaScript 引擎提供，不用自己实现。

`resolve` 函数的作用是，将 `Promise` 实例的状态从“未完成”变为“成功”（即从 `pending` 变为 `fulfilled`），在异步操作成功时调用，并将

异步操作的结果，作为参数传递出去。`reject`函数的作用是，将`Promise`实例的状态从“未完成”变为“失败”（即从`pending`变为`rejected`），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

下面是一个例子。

```
function timeout(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, ms, 'done');
  });
}

timeout(100)
```

上面代码中，`timeout(100)`返回一个 Promise 实例。100毫秒以后，该实例的状态会变为`fulfilled`。

Promise.prototype.then()

Promise 实例的`then`方法，用来添加回调函数。

`then`方法可以接受两个回调函数，第一个是异步操作成功时（变为`fulfilled`状态）时的回调函数，第二个是异步操作失败（变为`rejected`）时的回调函数（该参数可以省略）。一旦状态改变，就调用相应的回调函数。

```
var p1 = new Promise(function (resolve, reject) {
  resolve('成功');
});
p1.then(console.log, console.error);
// "成功"

var p2 = new Promise(function (resolve, reject) {
```

```
  reject(new Error('失败'));
});
p2.then(console.log, console.error);
// Error: 失败
```

上面代码中，`p1`和`p2`都是Promise 实例，它们的`then`方法绑定两个回调函数：成功时的回调函数`console.log`，失败时的回调函数`console.error`（可以省略）。`p1`的状态变为成功，`p2`的状态变为失败，对应的回调函数会收到异步操作传回的值，然后在控制台输出。

`then`方法可以链式使用。

```
p1
  .then(step1)
  .then(step2)
  .then(step3)
  .then(
    console.log,
    console.error
  );
```

上面代码中，`p1`后面有四个`then`，意味依次有四个回调函数。只要前一步的状态变为`fulfilled`，就会依次执行紧跟在后面的回调函数。

最后一个`then`方法，回调函数是`console.log`和`console.error`，用法上有一点重要的区别。`console.log`只显示`step3`的返回值，而`console.error`可以显示`p1`、`step1`、`step2`、`step3`之中任意一个发生的错误。举例来说，如果`step1`的状态变为`rejected`，那么`step2`和`step3`都不会执行了（因为它们是`resolved`的回调函数）。Promise 开始寻找，接下来第一个为`rejected`的回调函数，在上面代码中是`console.error`。这就是说，Promise 对象的报错具有传递性。

then() 用法辨析

Promise 的用法，简单说就是一句话：使用 `then` 方法添加回调函数。但是，不同的写法有一些细微的差别，请看下面四种写法，它们的差别在哪里？

```
// 写法一
f1().then(function () {
  return f2();
});

// 写法二
f1().then(function () {
  f2();
});

// 写法三
f1().then(f2());

// 写法四
f1().then(f2);
```

为了便于讲解，下面这四种写法都再用 `then` 方法接一个回调函数 `f3` 。写法一的 `f3` 回调函数的参数，是 `f2` 函数的运行结果。

```
f1().then(function () {
  return f2();
}).then(f3);
```

写法二的 `f3` 回调函数的参数是 `undefined` 。

```
f1().then(function () {
  f2();
  return;
}).then(f3);
```

写法三的 `f3` 回调函数的参数，是 `f2` 函数返回的函数的运行结果。

```
1. f1().then(f2())
2.   .then(f3);
```

写法四与写法一只有一个差别，那就是 `f2` 会接收到 `f1()` 返回的结果。

```
1. f1().then(f2)
2.   .then(f3);
```

实例：图片加载

下面是使用 Promise 完成图片的加载。

```
1. var preloadImage = function (path) {
2.   return new Promise(function (resolve, reject) {
3.     var image = new Image();
4.     image.onload  = resolve;
5.     image.onerror = reject;
6.     image.src = path;
7.   });
8. };
```

上面的 `preloadImage` 函数用法如下。

```
1. preloadImage('https://example.com/my.jpg')
2.   .then(function (e) { document.body.append(e.target) })
3.   .then(function () { console.log('加载成功') })
```

小结

Promise 的优点在于，让回调函数变成了规范的链式写法，程序流程

可以看得很清楚。它有一整套接口，可以实现许多强大的功能，比如同时执行多个异步操作，等到它们的状态都改变以后，再执行一个回调函数；再比如，为多个回调函数中抛出的错误，统一指定处理方法等等。

而且，Promise 还有一个传统写法没有的好处：它的状态一旦改变，无论何时查询，都能得到这个状态。这意味着，无论何时为 Promise 实例添加回调函数，该函数都能正确执行。所以，你不用担心是否错过了某个事件或信号。如果是传统写法，通过监听事件来执行回调函数，一旦错过了事件，再添加回调函数是不会执行的。

Promise 的缺点是，编写的难度比传统写法高，而且阅读代码也不是一眼可以看懂。你只会看到一堆 `then`，必须自己在 `then` 的回调函数里面理清逻辑。

微任务

Promise 的回调函数属于异步任务，会在同步任务之后执行。

```
new Promise(function (resolve, reject) {
  resolve(1);
}).then(console.log);

console.log(2);
// 2
// 1
```

上面代码会先输出2，再输出1。因为 `console.log(2)` 是同步任务，而 `then` 的回调函数属于异步任务，一定晚于同步任务执行。

但是，Promise 的回调函数不是正常的异步任务，而是微任务（microtask）。它们的区别在于，正常任务追加到下一轮事件循环，微任务追加到本轮事件循环。这意味着，微任务的执行时间一定早

于正常任务。

```
setTimeout(function() {
  console.log(1);
}, 0);

new Promise(function (resolve, reject) {
  resolve(2);
}).then(console.log);

console.log(3);
// 3
// 2
// 1
```

上面代码的输出结果是 `321` 。这说明 `then` 的回调函数的执行时间，早于 `setTimeout(fn, 0)` 。因为 `then` 是本轮事件循环执行， `setTimeout(fn, 0)` 在下一轮事件循环开始时执行。

参考链接

- Sebastian Porto, Asynchronous JS: Callbacks, Listeners, Control Flow Libs and Promises
- Rhys Brett-Bowen, Promises/A+ - understanding the spec through implementation
- Matt Podwysocki, Amanda Silver, Asynchronous Programming in JavaScript with “Promises”
- Marc Harter, Promise A+ Implementation
- Bryan Klimt, What’s so great about JavaScript Promises?
- Jake Archibald, JavaScript Promises There and back again

- Mikito Takada, 7. Control flow, Mixu’s Node book

DOM

DOM

概述

DOM 概述

DOM

DOM 是 JavaScript 操作网页的接口，全称为“文档对象模型”（Document Object Model）。它的作用是将网页转为一个 JavaScript 对象，从而可以用脚本进行各种操作（比如增删内容）。

浏览器会根据 DOM 模型，将结构化文档（比如 HTML 和 XML）解析成一系列的节点，再由这些节点组成一个树状结构（DOM Tree）。所有的节点和最终的树状结构，都有规范的对外接口。

DOM 只是一个接口规范，可以用各种语言实现。所以严格地说，DOM 不是 JavaScript 语法的一部分，但是 DOM 操作是 JavaScript 最常见的任务，离开了 DOM，JavaScript 就无法控制网页。另一方面，JavaScript 也是最常用于 DOM 操作的语言。后面介绍的就是 JavaScript 对 DOM 标准的实现和用法。

节点

DOM 的最小组成单位叫做节点（node）。文档的树形结构（DOM 树），就是由各种不同类型的节点组成。每个节点可以看作是文档树的

一片叶子。

节点的类型有七种。

- `Document`：整个文档树的顶层节点
- `DocumentType`：`doctype`标签（比如`<!DOCTYPE html>`）
- `Element`：网页的各种HTML标签（比如`<body>`、`<a>`等）
- `Attribute`：网页元素的属性（比如`class="right"`）
- `Text`：标签之间或标签包含的文本
- `Comment`：注释
- `DocumentFragment`：文档的片段

浏览器提供一个原生的节点对象`Node`，上面这七种节点都继承了`Node`，因此具有一些共同的属性和方法。

节点树

一个文档的所有节点，按照所在的层级，可以抽象成一种树状结构。这种树状结构就是 DOM 树。它有一个顶层节点，下一层都是顶层节点的子节点，然后子节点又有自己的子节点，就这样层层衍生出一个金字塔结构，倒过来就像一棵树。

浏览器原生提供`document`节点，代表整个文档。

```
1. document
2. // 整个文档树
```

文档的第一层只有一个节点，就是 HTML 网页的第一个标签`<html>`，它构成了树结构的根节点（root node），其他 HTML 标签节点都是它的下级节点。

除了根节点，其他节点都有三种层级关系。

- 父节点关系（parentNode）：直接的那个上级节点
- 子节点关系（childNodes）：直接的下级节点
- 同级节点关系（sibling）：拥有同一个父节点的节点

DOM 提供操作接口，用来获取这三种关系的节点。比如，子节点接口包括 `firstChild`（第一个子节点）和 `lastChild`（最后一个子节点）等属性，同级节点接口包括 `nextSibling`（紧邻在后的那个同级节点）和 `previousSibling`（紧邻在前的那个同级节点）属性。