# JavaScript（进阶）

## JavaScript高阶进阶指南（图）



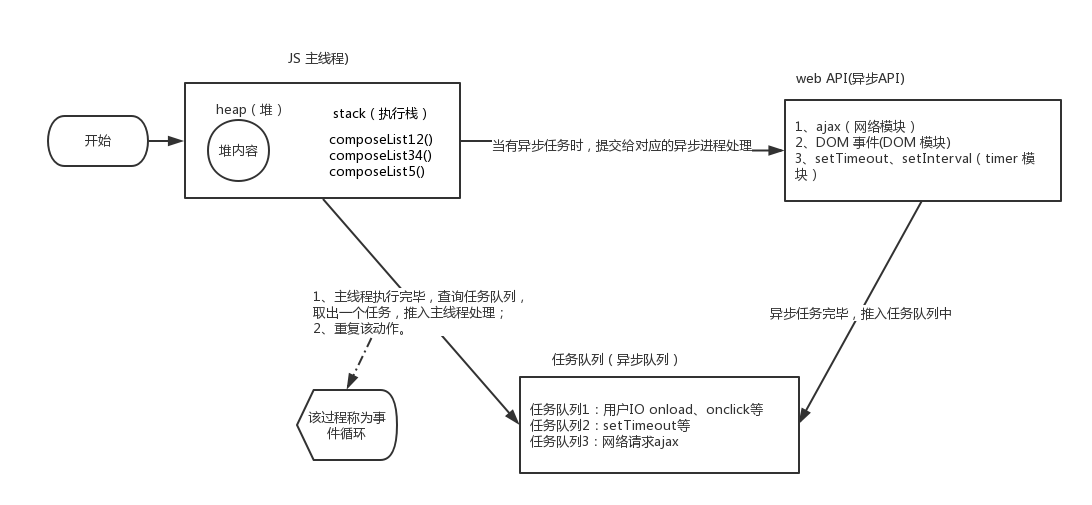
## JS 事件循环 – 任务队列、 web API 、 js 主线程的相互协同

### JS 单线程、异步、同步概念

JavaScript是单线程（如果一个线程是新增，一个删除，浏览器只能等着了），虽然有webworker（多线程）的出现，但也是在主线程的控制下进行的。Webworker仅仅能进行计算任务，不能操作dom，所以本质上还是单线程。

单线程即任务是串行的，后一个任务等待着前一个任务执行完毕，这样会浪费很多时间等待。但由于Ajax网络请求、setTimeOut时间延迟、DOM事件的用户交互等，这些任务不消耗CPU，是一种空等，浪费资源，因此出现了异步。通过将任务交给相应的异步去处理，主线程的效率大大提升了，可以并行的处理其他的操作。当异步处理完成，主线程空闲时，主线程读取相应的callback，进行后续的操作，最大程度的利用CPU。此时出现了同步执行和异步执行的概念，同步执行是主线程按照顺序，串行执行任务，异步执行就是CPU跳过等待，先处理后续任务（CPU与网络模块、timer等并行进行的任务）。由此产生了任务队列与事件循环，来协调主线程与异步模块之间的工作。

### 事件循环机制



上图流程：

Step1：主线程读取JS代码，此时为同步环境，形成相应的堆和执行栈；

Step2：主线程遇到异步任务，只给对应的异步进程进行处理（web API）

Step3：异步进程处理完毕（ajax返回，DOM事件触发、TIEMER定时器等），将相应的异步任务推入任务队列；

Step4：主线程执行完毕，查询任务队列，如果存在任务，则取出一个任务推入主线程（先进先出）；

Step5：重复执行step2、 3、 4；称为事件循环。

### 任务队列

任务队列存在多个，在同一个任务队列内，按队列顺序被主线程取走；不同的任务队列之间，存在着优先级，优先级高的优先取出（如：用户I/O）；

#### 任务队列类型

任务队列类型分为两种：一种为microtask queue，另一种为：macrotask queue。

上图展示为macrotask queue， 而ES6的0promise[EcmaScript标准] 产生的任务队列为microtask queue。

#### 两者区别

Microtask queue：唯一，整个事件循环中，仅存一个；执行为同步，同一个事件循环microtask会按照队列顺序，串行执行完毕。

Microtask queue：不唯一，存在一个定的优先级（用户I/O部分优先级更高）；异步执行，同一个事件循环中，只执行一个。

#### 更完成的事件循环流程

Step1：主线程读取JS代码，此时为同步环境，形成相应的堆和执行栈；

Step2：主线程遇到异步任务，只给对应的异步进程进行处理（web API）

Step3：异步进程处理完毕（ajax返回，DOM事件触发、TIEMER定时器等），将相应的异步任务推入任务队列；

Step4：主线程查询任务队列，执行microtask queue， 将其按序执行，全部执行完毕；

Step5：主线程查询任务队列。，执行macrotask queue，取队首任务执行，执行完毕；

Step6：重复 step4 、 5 。

Microtask queue中的所有callback处在同一个任务循环中，而macrotask queue中的callback有自己的事件循环。

**总结**：同步环境执行 🡪 事件循环1（microtask queue 的all） 🡪 事件循环2（macrotask queue中的一个） 🡪 事件循环1（microtask queue的all） 🡪 事件循环2（macrotask queue中的一个）……

利用microtask queue可以形成一个同步执行的环境，但如果microtask queue太长，将导致macrotask 任务长时间执行不了，最终导致用户I/O 无响应等问题，所以需要谨慎使用。

### ES5规范对于事件循环的定义

规范中提到：一个浏览器环境，只有一个事件循环，而一个时间循环可以多个任务队列，每个任务队列都有一个任务源（task source）

相同任务源的任务，只能放在一个任务队列中。

不同的任务源的任务，可以放在不同的任务队列中。

***结论： 一个时间循环可以有多个任务队列，队列之间可以有不同的优先级，同一个任务队列的任务按先进先出的顺序执行，但是不保证多个任务队列的任务优先级，具体实现可能会交叉执行。***

**注意： 在JS中Es6 中的新增的任务队列（Promise）是在事件循环之上的，事件循环每次tick后会查看ES6的任务队列中是否有任务执行，也就是ES6的任务执行队列比事件循环中的任务队列优先级高。**

## JS 模式

### 设计模式

一个模式就是一个可重用的方案，可应用于在软件设计中的常见问题 - 在我们的例子里 - 就是编写JavaScript的web应用程序。模式的另一种解释就是一个我们如何解决问题的模板 - 那些可以在许多不同的情况里使用的模板。 那么理解和熟悉模式为什么是如此的重要？设计模式有以下三点好处：

* 模式是行之有效的解决方法：他们提供固定的解决方法来解决在软件开发中出现的问题，这些都是久经考验的反应了开发者的经验和见解的使用模式来定义的技术。
* 模式可以很容易地重用：一个模式通常反映了一个可以适应自己需要的开箱即用的解决方案。这个特性让它们很健壮。
* 模式善于表达：当我们看到一个提供某种解决方案的模式时，一般有一组结构和词汇可以非常优雅地帮助表达相当大的解决方案。

#### 单例模式

单例模式的定义：保证一个类仅有一个实例，并提供一个访问它的全局

访问点。实现的方法为先判断实例存在与否，如果存在则直接返回，如果不存在就创建了再返回，这就确保了一个类只有一个实例对象。

**适用场景**：一个单一对象。比如：弹窗，无论点击多少次，弹窗只应该被创建一次。

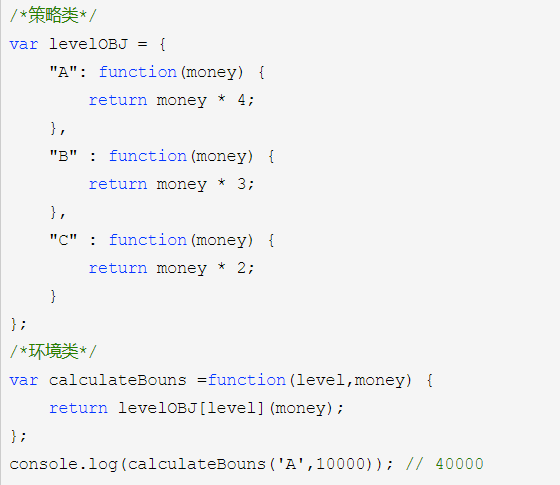


#### 策略模式

策略模式的定义：定义一系列的算法，把他们一个个封装起来，并且使他们可以相互替换。

策略模式的目的就是将算法的使用算法的实现分离开来。

一个基于策略模式的程序至少由两部分组成。第一个部分是一组**策略类**（可变），策略类封装了具体的算法，并负责具体的计算过程。第二个部分是**环境类**Context（不变），Context接受客户的请求，随后将请求委托给某一个策略类。要做到这一点，说明Context中要维持对某个策略对象的引用。



#### 代理模式

代理模式的定义：为一个对象提供一个代用品或占位符，以便控制对它的访问。

常用的虚拟代理形式：某一个花销很大的操作，可以通过虚拟代理的方式延迟到这种需要它的时候才去创建（例：使用虚拟代理实现图片懒加载）

图片懒加载的方式：先通过一张loading图占位，然后通过异步的方式加载图片，等图片加载好了再把完成的图片加载到img标签里面。



使用代理模式实现图片懒加载的优点还有符合单一职责原则。减少一个类或方法的粒度和耦合度

#### 中介者模式

中介者模式的定义：通过一个中介者对象，其他所有的相关对象都通过该中介者对象来通信，而不是相互引用，当其中的一个对象发生改变时，只需要通知中介者对象即可。通过中介者模式可以解除对象与对象之间的紧耦合关系。

例如：现实生活中，航线上的飞机只需要和机场的塔台通信就能确定航线和飞行状态，而不需要和所有飞机通信。同时塔台作为中介者，知道每架飞机的飞行状态，所以可以安排所有飞机的起降和航线安排。

中介者模式适用的场景：例如购物车需求，存在商品选择表单、颜色选择表单、购买数量表单等等，都会触发change事件，那么可以通过中介者来转发处理这些事件，实现各个事件间的解耦，仅仅维护中介者对象即可。



#### 装饰者模式

装饰者模式的定义：在不改变对象自身的基础上，在程序运行期间给对象动态地添加方法。

例如：现有4种型号的自行车分别被定义成一个单独的类，如果给每辆自行车都加上前灯、尾灯、铃铛这3个配件，如果用类继承的方式，需要创建4\*3=12个子类。但如果通过装饰者模式，只需要创建3个类。

装饰者模式适用的场景：原有方法维持不变，在原有方法上再挂载其他方法来满足现有需求；函数的解耦，将函数拆分成多个可复用的函数，再将拆分出来的函数挂载到某个函数上，实现相同的效果但增强了复用性。



#### 观察者模式

观察者模式含义：是一个被称为被观察者的对象，维护一组称为观察者的对象，这些对象依赖于被观察者，被观察者自动将自身的状态的任何变化通知给他们。

观察者模式实现所包含的组件：

* 被观察者：维护一组观察者， 提供用于增加和移除观察者的方法。
* 观察者：提供一个更新接口，用于当被观察者状态变化时，得到通知。
* 具体的被观察者：状态变化时广播通知给观察者，保持具体的观察者的信息
* 具体的观察者：保持一个指向具体被观察者的引用，实现一个更新接口，用于观察，以便保证自身状态总是和被观察者状态一致的。

#### 亨元(Flyweight) 模式

**含义**： 亨元模式是一个优化重复、缓慢和低效数据共享代码的典型结构化解决方案。它的目标是以相关对象尽可能多的共享数据，来减少应用程序中内存的使用（例如： 应用程序的配置、状态等）。

**使用亨元**：

1. 数据层，基于存储在内存中的大量相同对象的数据共享的概念。
2. DOM层，亨元模式被作为事件管理中心，以避免将事件处理程序关联到我们需要相同行为父容器的所有子节点上。

**注意**：亨元模式通常被更多的用于数据层。

### 编码模式

### 反模式

含义：反模式是指对反复出现的设计问题的常见的无力而低效的设计模式，俗话说就是重蹈覆辙。

常见的几种反模式：

#### 硬编码

硬编码（Hard-Coding）的字符串、数字、日期…… 所有能写死的东西都会被人写死。 这是一个妇孺皆知的反模式，同时也是最广泛使用的反模式。 硬编码中最为典型的大概是 平台相关代码（Platform-Related）， 这是指特定的机器或环境下才可以正常运行的代码， 可能是只在你的机器上可以运行，也可能是只在 Windows 下可以运行。

#### 重复代码

重复代码（Duplication）在业务代码中尤为常见，初衷几乎都是维护业务的稳定性

#### 假AMD

模块化本意是指把软件的各功能分离到独立的模块中，每个模块包含完整的一个细分功能。 在 JavaScript 中则是特指把脚本切分为独立上下文的，可复用的代码单元。

#### 注释膨胀

注释的初衷是让读者更好的理解代码意图，但实践中可能恰好相反。

#### 函数体膨胀

“通常”认为函数体膨胀和全局变量都是算法课的后遗症。 但复杂的业务和算法的场景确实不同，前者有更多的概念和操作需要解释和整理。 整理业务逻辑最有效的手段莫过于变量命名和方法抽取（当然，还要有相应的闭包或对象）

## MV\*模式

### MVC模式

#### MVC简介

MVC是一个架构设计模式，通过分离关注点来支持改进应用组织方式。它促成了业务数据（Models）从用户界面（Views）中分离出来。还用第三个组成部分（Controllers）负责管理传统意义上的业务逻辑和用户输入。该模式最初由[Trygve Reenskaug](http://en.wikipedia.org/wiki/Trygve_Reenskaug" \t "_blank) 在研发Smalltalk-80（1979）期间设计的，当时它起初被称为Model-View-Controller-Editor。在1995年的[“设计模式: 面向对象软件中的可复用元素”](http://www.amazon.co.uk/Design-patterns-elements-reusable-object-oriented/dp/0201633612" \t "_blank)（著名的“GoF”的书）中，MVC被进一步深入的描述，该书对MVC的流行使用起了关键作用。

##### Smalltalk-80 MVC

关于Smalltalk-80的MVC框架，值得注意的几点：

* 模型表现了领域特定的数据,并且不用考虑用户界面(视图和控制器).当一个模型有所改变的时候,它会通知它的观察者。
* 视图表现了一个模型的当前状态.观察者模式被用来让视图在任何时候都知晓模型已经被更新了或者被改变了。
* 展现受到视图的照管,但是不仅仅只有一个单独的视图或者控制器——每一个在屏幕上展现的部分或者元素都需要一个视图-控制器对。
* 控制器在这个视图-控制器对中扮演着处理用户交互的角色（比如按键或者点击动作），做出对视图的选择。

#### MVC三大核心部分

##### Models

Models管理一个业务应用的数据。它们既与用户界面无关也与表现层无关，相反的它们代表了一个业务应用所需要的形式唯一的数据。 当一个model改变时(比如当它被更新时)，它通常会通知它的观察者(比如我们很快会介绍的views)一个改变已经发生了，以便观察者采取相应的反应。总而言之,模型主要关注的是业务数据。

##### 视图（View）

视图是模型的可视化表示，提供了一个当前状态的经过过滤的视图。Smaltalk的视图是关于绘制和操作位图的，而JavaScript的视图是关于构建和操作DOM元素的。

一个视图通常是模型的观察者，当模型改变的时候，视图得到通知，因此使得视图可以更新自身。用设计模式的语言可以称视图为“哑巴”，因为在应用程序中是它们关于模型和控制器的了解是受到限制的。用户可以和视图进行交互，包括读和编辑模型的能力（例如，获取或者设置模型的属性值）。因为视图是表示层，我们通常以用户友好的方式提供编辑和更新的能力。

###### 模板

模板也许是用一种声明的方式指定部分甚至所有的视图对象，因此它可能是从模板定制文档生成的。

##### 控制器（controllers）

控制器是模型和视图之间的中介，典型的职责是当用户操作视图的时候同步更新模型。

请记住满足了MVC中的一种角色：针对视图的策略模式的基础设施。在策略模式方面，视图在视图的自由载量权方面代表了控制器。因此，那就是测试模式是如何工作的，视图可以代表针对控制器的用户事件，当视图看起来合适的时候。视图也可以代表针对控制器的模型变更事件处理，当视图看起来合适的时候，但这并不是控制器的传统角色。

##### MVC总结

MVC中关注分离的思想有利于对应用程序中功能进行更加简单的模块化，并且使得：

* 整体的维护更加便利.当需要对应用程序进行更新时,到底这些改变是否是以数据为中心的,意味着对模型的修改还-有可能是控制器,或者仅仅是视觉的,意味着对视图的修改,这一区分是非常清楚的。
* 对模型和视图的解耦意味着为业务逻辑编写单元测试将会是更加直截了当的。
* 对底层模型和控制器的代码解耦(即我们可能会取代使用的)在整个应用程序中被淘汰了。
* 依赖于应用程序的体积和角色的分离,这种模块化允许负责核心逻辑的开发者和工作于用户界面的开发者同时进行工作。

### MVP模式

#### MVP简介

模型-视图-展示器(MVP)是MVC设计模式的一个衍生模式,它专注于提升展现逻辑.

目标：直指对整个多组件关注点的分离。

#### 模型(models)、视图(View)、展示器(Passive View)

MVP中的P代表展示器.它是一个包含视图的用户界面逻辑的组件.不像MVC,来自视图的调用被委派给了展示器,它是从视图中解耦出来的,并且转而通过一个接口来同它进行对话.这允许所有类型的有用的东西,比如在单元测试中模拟视图的调用

对MVP最通常的实现是使用一个被动视图(Passive View 一种对所有动机和目的保持静默的视图),包含很少甚至与没有任何逻辑.如果MVC和MVP是不同的,那是因为其C和P干了不同的事情.在MVP中,P观察着模型并且当模型发生改变的时候对视图进行更新.P切实的将模型绑定到了视图,这一责任在MVC中被控制器提前持有了。

Backbone.js更适合于MVP的描述相较于MVC:

* 相比于控制器，MVP中的展示器更好的描述了Backbone.View（视图模板和绑定在视图模板之上的数据之间的中间层）。
* 模型适合Backbone.Model（相较于MVC中的模型并没有很大的不同）。
* 视图最能代表模板（比如 Handlebars/Mustache标记模板）

### MVVM模式

#### MVVM简介

MVVM(Model View ViewModel)是一种基于MVC和MVP的架构模式，它试图将用户界面（UI）从业务逻辑和行为中更加清晰地分离出来。为了这个目的，很多例子使用声明变量绑定来把View层的工作从其他层分离出来。

这促进了UI和开发工作在同一代码库中的同步进行。UI开发者用他们的文档标记（HTML）绑定到ViewModel，在这个地方Model和ViewModel由负责逻辑的开发人员维护。

#### 模型

MVVM中的模型代表我们的应用用到的领域相关的数据或者信息

#### 视图

使用MVC,视图是应用程序中用户真正与之打交道的唯一一个部分.它们是展现一个视图模型状态的一个可交互UI.此种意义而言,视图是主动的而不是被动的,而这也是真正的MVC和MVP的观点.在MVC,MVP和MVVM中视图也可以是被动的,

被动视图仅仅只输出要展示的东西,而不去接受任何用户的输入

#### 视图模型

视图模型被认为是一个专门进行数据转换的控制器。它可以把对象信息转换到视图信息，将命令从视图携带到对象。

#### 小结: 视图模式和模型

虽然可能会出现在MVVM中视图模型完全对模型负责的情况,这些关系确实有一些值得关注的微妙之处.处于数据绑定的目的,视图模型可以暴露出来一个模型或者模型属性,而且也能够包含获取和操作视图中暴露出来的属性。

#### MVVM的低耦合数据绑定

我必须承认当我第一次体验实现了MVVM的JavaScript框架后(例如 KnockoutJS, Knockback)，我很惊讶很多程序员都想要回到一个难以维护的混淆了逻辑(JavaScript代码)和HTML标签做法的过去。然而现实是使用MVVM会有很多好处（我们之前说过），包括设计师能更容易的通过他们的标记去绑定相关逻辑。

在我们中间的传统程序员，你会很开心知道现在我们能够通过数据绑定这个特性大量减少程序代码的耦合程度，且KnockoutJS从1.3这个版本就开始提供自定义绑定功能。

##### 优点和缺点

优点：

* MVVM更加便于UI和驱动UI的构造块,这两部分的并行开发
* 抽象视图使得背后所需要的业务逻辑(或者粘合剂)的代码数量得以减少
* 视图模型比事件驱动代码更加容易进行单元测试
* 视图模型(比视图更加像是模型)能够在不用担心UI自动化和交互的前提下被测试

缺点：

* 对于更简单的UI而言,MVVM可能矫枉过正了
* 虽然数据绑定可以是声明性质的并且工作得很好,但在我们简单设置断点的地方,它们比当务之急的代码更加难于调试
* 在非凡的应用程序中的数据绑定能够创造许多的账簿.我们也并不希望以绑定比被绑定目标对象更加重量级,这样的境地告终
* 在大型的应用程序中,将视图模型的设计提升到获取足够所需数量的泛化,会变得更加的困难

### 总结

MVP和MVVM都是MVC的衍生物。它和它的衍生物之间关键的不同之处在于每一层对于其它层的依赖，以及它们相互之间是如何紧密结合在一起的。

#### MVC：

在MVC中，视图位于我们架构的顶部，其背后是控制器。模型在控制器后面，而因此我们的视图了解得到我们的控制器，而控制器了解得到模型。

**注意**：这里，我们的视图有对模型的直接访问。然而将整个模型完全暴露给视图可能会有安全和性能损失，这取决于我们应用程序的复杂性。

#### MVP

控制器的角色被代理器所取代，代理器和视图处于同样的地位，视图和模型的事件都被它侦听着并且接受它的调解。

**注意**：

不同于MVVM，没有一个将视图绑定到视图模型的机制，因此我们转而依赖于每一个视图都实现一个允许代理器同视图去交互的接口。

#### MVVM

进一步允许我们创建一个模型的特定视图子集，包含了状态和逻辑信息，避免了将模型完全暴露给视图的必要。不同于MVP的代理器，视图模型并不需要去引用一个视图。视图可以绑定到视图模型的属性上面，视图模型则去将包含在模型中的数据暴露给视图。

**注意**：

视图模型和视图层之间新增的的用于翻译解释的一层会有性能损失

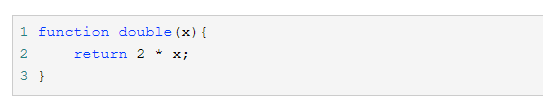
这种解释层的复杂度根据情况也会有所差异——它可能像复制数据一样简单，也可能会像我们希望用视图理解的一种形式去操作它们，那样复杂

## JavaScript函数（匿名函数自调用）

### 创建函数的方式

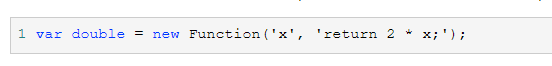
#### 第一种：

这是最常规的一种创建函数的方式



#### 第二种

这种方式使用了Function 构造函数，把参数列表和函数体都作为字符串，很不方便，不建议使用。



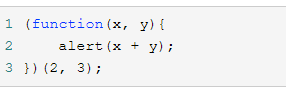
#### 第三种：

变量赋值的方式创建函数



### 匿名函数的创建

匿名函数自调用的方式创建函数，第一个括号是匿名函数，第二个括号是用于调用该匿名函数，并传入参数。括号是表达式，是表达式就有返回值，所以可以在后面加一对括号让他们执行。



#### 自调用匿名函数

##### 什么是自调用（自执行）匿名函数

指形如： （function { // code }）()

#### 匿名函数与闭包

匿名函数最大的用途就是创建闭包（这是JavaScript语言的特性之一），并且还可以构建命名空间，以减少全局变量的使用。

##### 闭包：

###### 起因

在函数内声明的数据，在函数外是不能直接访问的。

闭包含义： 就是包裹、包装的意思。闭表示封闭，即不可访问。

所谓的闭包就是一个具体的封闭的特性，同时具有包裹的特性的一种结构。

从代码的角度说， 所谓的闭包就是函数。

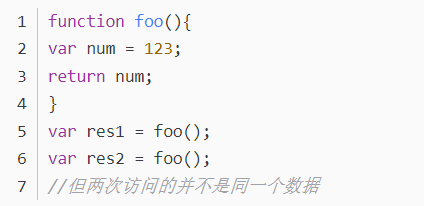
* 1. 函数内声明的变量，外界无法访问，即封闭性
  2. 函数本身又是一个封闭单位，利用函数可以封装代码，因此一个函数也是一个包裹结构。因此函数既有包裹的特性，也有封闭的特性，因此可以直接称呼函数为闭包（closure）

##### 闭包特性

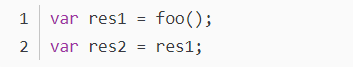
所谓闭包就是函数，其特点是函数内定义的变量，在函数外是无法访问的。

如何访问函数中的数据：

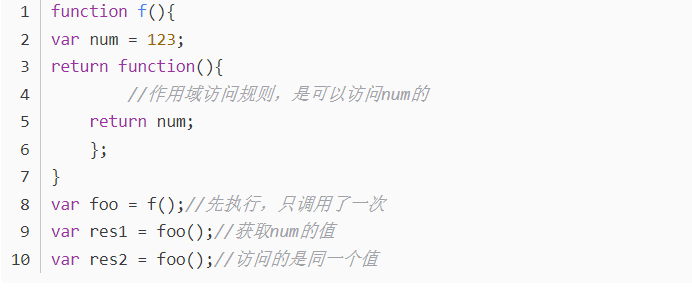
1. 最简单的方法：返回



1. 多次访问，f只允许调用一次



（3）使得函数返回一个函数利用函数返回数据



总结：

所谓闭包，就是一个函数，因为函数中声明的变量，外界无法访问，函数就构成了封闭结构，因为成为了闭包。同时如果需要访问函数中的数据，可以考虑利用函数返回一个函数，使用返回的函数操作函数内的数据，以使得函数外可以通过这个函数内返回的函数访问内部的数据。

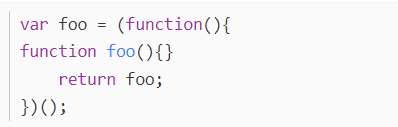
##### 闭包用法

###### 作为沙箱模式

沙箱：就是一个独立的与外界隔离的运行环境（function）



暴露方式：



###### 带有缓存功能的函数



## JavaScript注意知识点

### JavaScript 关键词void

JavaScript 中void十一个中重要的关键词。它可以作用一个一元运算符，此时它会出现一个操作数之前，这个操作数可以是任意类型。



#### 例子一：

这个运算符最常用在客户端的JavaScript:URL 中，在URL中可以写带有副作用的的表达式，而且void则让浏览器不必显示这个表达式的计算结果

这里的alert（‘Warning！！！！’）表达式被执行了，但是他不会在当前文档出装入任何内容：



#### 例子二：

另外的一个例子，下面的超级链接并不会做任何事，因为在JavaScript中表达式 ‘0’没有任何作用。这里的表达式“0”已经被计算，但是它并没有在当前文档处状如任何内容：



#### 例子三：

Void的另一种用法是有意生成undefined值，如下所示：



### Js的全局函数

escape( )、eval( )、isFinite( )、isNaN( )、parseFloat( )、parseInt( )、unescape( )。

注意：函数绑定通常只允许充当程序特定部分的名称、这样的绑定被定义一次，永远不会改变。这使得容易混淆函数和名称。

### 访问属性的方式

JavaScript中访问属性的两种方式点(.)和方括号([])。value.x和value [x]都可以访问value属性，但不一定是同一个属性。 区别在于如何解释x。

使用点时，点后面的单词是该属性的字面名称。 使用方括号时，会求解括号内的表达式来获取属性名称 。鉴于value.x获取value的名为x的属性，value [x]尝试求解表达式x，并将结果转换为字符串作为属性名称。

### JSON

因为属性只是捕获了它们的值，而不是包含它们，对象和数组在计算机的内存中储存为字节序列，存放它们的内容的地址（内存中的位置）。 因此，包含另一个数组的数组，（至少）由两个内存区域组成，一个用于内部数组，另一个用于外部数组，（除了其它东西之外）其中包含表示内部数组位置的二进制数。

### 沙箱

沙箱的产生：浏览网站、而不必担心遭受病毒的攻击。是因为浏览器严重限制了JavaScript的能力---无法查看电脑中的任何文件，也无法修改与其所在的页面的无关数据。这种隔离程序运行环境的技术称为沙箱。以该思想编写的程序不会对计算机造成任何的伤害，但是应该想象，这种特殊的沙箱上面有一个厚钢筋笼子，所以在其运行的程序实际上不会出去。

困难点： 一方面我们要给与程序一定的自由使得程序能够实际使用，但是又要限制其防止其执行危险的行为。许多使用的功能（比如服务器通信或者从粘贴板读取内容）也会存在问题，有些侵入者可以利用这些功能来侵入你的计算机。

时不时会有一些人想到新的方法，突破浏览器的限制，并对你的机器造成伤害，从窃取少量的私人信息到掌握执行浏览器的整个机器。

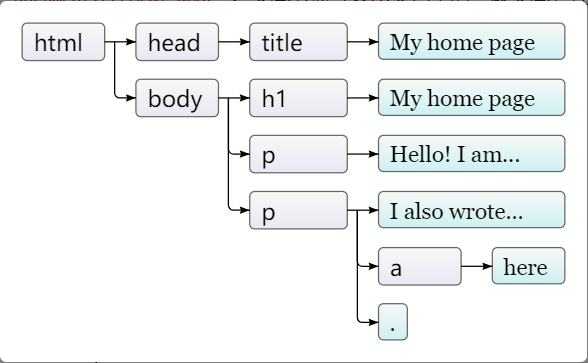
### 兼容性与浏览器之争

网络开发者总是缺乏一种统一的web标准，而需要去考虑两到三种不不兼容的平台而感到烦恼。让事情变得更糟糕的是2003年左右使用的浏览器充满了漏洞，当然不同浏览器的漏洞都不是一样的。网页编写者的生活颇为艰辛。

新的参与者对标准有着更认真的态度，和更好的工程实践，为我们减少了不少兼容性和错误。

微软看到其市场份额急速下降，在其Edge浏览器中采用了这种态度，取代Internet Explorer。现在的主流浏览器的最新版本行为非常一致，并且错误相对较少。

### 树



叶子节点是文本节点，而箭头则指出了节点之间的父子关系。

理论上，你可以通过父子之间的链接移动到树中的任何地方。但 JavaScript 也提供了一些更加方便的额外链接。firstChild属性和lastChild属性分别指向第一个子节点和最后一个子节点，若没有子节点则值为null。类似的，previousSibling和nextSibling指向相邻节点，分别指向拥有相同父亲的前一个节点和后一个节点。对于第一个子节点，previousSibling是null，而最后一个子节点的nextSibling则是null。

也存在children属性，它就像childNodes，但只包含元素（类型为 1）子节点，而不包含其他类型的子节点。 当你对文本节点不感兴趣时，这可能很有用。

### 布局

块级元素：

比如段落（<p>）和标题（<h1>）会占据整个文档的宽度，并且在独立的一行中渲染

内联元素：

比如链接（<a>或<strong>元素则与周围文本在同一行中渲染

注意：如果程序反复读取DOM布局信息或者修改DOM，会强制引发大量布局计算，导致运行非常缓慢。

Display属性可以控制一个元素显示为块元素或者是内联元素。

### 浏览器和HTTP

Fetch：

浏览器JavaScript可以通过fetch接口生成HTTP请求。由于他比较新，所以很方便的使用了Promise

1. fetch("example/data.txt").then(response => {
2. console.log(response.status);
3. // → 200
4. console.log(response.headers.get("Content-Type"));
5. // → text/plain
6. });

## 闭包与作用域

### 作用域

#### 概念

在我们的程序纳入变量，引出我们现在将要解决的最有趣的问题：这些变量存储在哪里？我们的程序如何找到它们？回答这些问题需要一组明确的定义的规则，它定义了变量储存位置，以及如何找到这些变量。我们称这组规则为：作用域

#### 编译原理

编译步骤： 分词/词法分析、解析、代码生成

##### 词法分析

将一连串字符打断成（对于语言来说）有意义的片段，称为token（记号）。

例如：

考虑这段程序：var a = 2;。这段程序很可能会被打断成如下 token：var，a，=，2，和 ;。空格也许会被保留为一个 token，这要看它是否是有意义的。

**注意：** 分词和词法分析之间的区别是微妙和学术上的，其中心在于这些 token 是否以 *无状态* 或 *有状态* 的方式被识别。简而言之，如果分词器去调用有状态的解析规则来弄清a是否应当被考虑为一个不同的 token，还是只是其他 token 的一部分，那么这就是 **词法分析**。

##### 解析

将一个token的流（数组）转换为一个嵌套元素的树，它综合地表示了程序的语法结构。这课称为“抽象语法树”（AST —— **A**bstract **S**yntax **T**ree）

var a = 2; 的树也许开始于称为 VariableDeclaration（变量声明）顶层节点，带有一个称为 Identifier（标识符）的子节点（它的值为 a），和另一个称为 AssignmentExpression（赋值表达式）的子节点，而这个子节点本身带有一个称为 NumericLiteral（数字字面量）的子节点（它的值为2）。

##### 代码生成

这个处理将抽象语法树转换为可执行的代码。这一部分将根据语言，它的目标平台等因素有很大的不同。

所以，与其深陷细节，我们不如笼统地说，有一种方法将我们上面描述的 var a = 2; 的抽象语法树转换为机器指令，来实际上 创建 一个称为 a 的变量（包括分配内存等等），然后在 a 中存入一个值。

**注意：** 引擎如何管理系统资源的细节远比我们要挖掘的东西深刻，所以我们将理所当然地认为引擎有能力按其需要创建和存储变量

和大多数其他语言的编译器一样，JavaScript引擎要比这区区三步复杂太多了。例如，在解析和代码生成的处理中，一定会存在优化执行效率的步骤，包括压缩冗余元素，等等

所以上面的三个步骤描述的是大框架。

JavaScript： 在大多数情况下，编译发生在执行前的几微秒之内。为了确保性能JS引擎将使用所有的招数（比如 JIT， 他可以懒编译甚至热编译，等等），而这远超出我们关于“作用域”的讨论。

为了简单起见，我们可以说，任何JavaScript代码段在它执行前（通常是 刚好在它执行前）都必须被编译。所以，JS编译器将把程序var a = 2; 拿过来，并首先编译它，然后准备运行它，通常是立即的。

#### 理解作用域

处理程序主要成员：

引擎：负责从始至终的编译和执行我们的 JavaScript 程序

编译器：引擎 的朋友之一；处理所有的解析和代码生成的重活儿

作用域：引擎 的另一个朋友；收集并维护一张所有被声明的标识符（变量）的列表，并对当前执行中的代码如何访问这些变量强制实施一组严格的规则

编译器处理机制：

1. 遇到 var a，*编译器* 让 *作用域* 去查看对于这个特定的作用域集合，变量 a 是否已经存在了。如果是，*编译器* 就忽略这个声明并继续前进。否则，*编译器* 就让 *作用域* 去为这个作用域集合声明一个称为 a 的新变量。
2. 然后 *编译器* 为 *引擎* 生成稍后要执行的代码，来处理赋值 a = 2。*引擎* 运行的代码首先让 *作用域* 去查看在当前的作用域集合中是否有一个称为 a 的变量可以访问。如果有，*引擎* 就使用这个变量。如果没有，*引擎* 就查看 *其他地方*（参见下面的嵌套 *作用域* 一节）。

如果 引擎 最终找到一个变量，它就将值 2 赋予它。如果没有，引擎 将会举起它的手并喊出一个错误！

总结来说：对于一个变量赋值，发生了两个不同的动作：第一，编译器 声明一个变量（如果先前没有在当前作用域中声明过），第二，当执行时，引擎 在 作用域 中查询这个变量并给它赋值，如果找到的话。

#### 编译器术语

当 引擎 执行 编译器 在第二步为它产生的代码时，它必须查询变量 a 来看它是否已经被声明过了，而且这个查询是咨询 作用域 的。但是 引擎 所实施的查询的类型会影响查询的结果。

在我们这个例子中，引擎 将会对变量 a 实施一个“LHS”查询。另一种类型的查询称为“RHS”。

**注意:** 这两个术语表示“Left-hand Side（左手边）”和“Right-hand Side（右手边） “边”：赋值操作的边

换言之，当一个变量出现在赋值操作的左手边时，会进行 LHS 查询，当一个变量出现在赋值操作的右手边时，会进行 RHS 查询。

拓展 ： RHS 是难以察觉的，因为它简单地查询某个变量的值，而 LHS 查询是试着找到变量容器本身，以便它可以赋值。从这种意义上说，RHS 的含义实质上不是 真正的 “一个赋值的右手边”，更准确地说，它只是意味着“不是左手边”。

可以认为“RHS”意味着“取得他/她的源（值）”，暗示着 RHS 的意思是“去取……的值”。

1. console.log( a );

这个指向 a 的引用是一个 RHS 引用，因为这里没有东西被赋值给 a。而是我们在查询 a 并取得它的值，这样这个值可以被传递进 console.log(..)。

1. a = 2;

这里指向 a 的引用是一个 LHS 引用，因为我们实际上不关心当前的值是什么，我们只是想找到这个变量，将它作为 = 2 赋值操作的目标

**注意：** LHS 和 RHS 意味着“赋值的左/右手边”未必像字面上那样意味着“ = 赋值操作符的左/右边”。赋值有几种其他的发生形式，所以最好在概念上将它考虑为：“赋值的目标（LHS）”和“赋值的源（RHS）”。

LHS 引用和RHS混用：

1. function foo(a) {
2. console.log( a ); // 2
3. }
4. foo( 2 );

#### 词法作用域

作用域的工作方式有两种占统治地位的模型。其中的第一种是最常见的，在绝大多数的编程语言中被使用的，被称为**词法作用域**

另一种仍然被一些语言（比如Bash脚本，Perl中的一些模式，等等）使用的模型，被称为**动态作用域**

##### 欺骗词法作用域

JavaScript 有两种这样的机制。在广大的社区中它们都等同地被认为是让人皱眉头的，在你代码中使用它们是一种差劲儿的做法。但是关于它们的常见的争论经常错过了最重要的一点：**欺骗词法作用域会导致更低下的性能**

###### Eval

JavaScript 中的 eval(..) 函数接收一个字符串作为参数值，并将这个字符串的内容看作是好像它已经被实际编写在程序的那个位置上。换句话说，你可以用编程的方式在你编写好的代码内部生成代码，而且你可以运行这个生成的代码，就好像它在编写时就已经在那里了一样。

如果以这种观点来评价 eval(..)，那么 eval(..) 是如何允许你修改词法作用域环境应当是很清楚的：欺骗并假装这个编写时（也就是，词法）代码一直就在那里。

在 eval(..) 被执行的后续代码行中，*引擎* 将不会“知道”或“关心”前面的代码是被动态翻译的，而且因此修改了词法作用域环境。*引擎* 将会像它一直做的那样，简单地进行词法作用域查询。.

考虑如下代码：

1. function foo(str, a) {
2. eval( str ); // 作弊！
3. console.log( a, b );
4. }
5. var b = 2;
6. foo( "var b = 3;", 1 ); // 1 3

在 eval(..) 调用的位置上，字符串 "var b = 3" 被看作是一直就存在在那里的代码。因为这个代码恰巧声明了一个新的变量 b，它就修改了现存的 foo(..) 的词法作用域。事实上，就像上面提到的那样，这个代码实际上在 foo(..) 内部创建了变量 b，它遮蔽了声明在外部（全局）作用域中的 b

当 console.log(..) 调用发生时，它会在 foo(..) 的作用域中找到 a 和 b，而且绝不会找到外部的 b。这样，我们就打印出 “1 3” 而不是一般情况下的 “1 2”。

**注意：**在这个例子中，为了简单起见，我们传入的“代码”字符串是固定的文字。但是它可以通过根据你的程序逻辑将字符拼接在一起，很容易地以编程方式创建。eval(..) 通常被用于执行动态创建的代码，因为动态地对一段实质上源自字符串字面值的静态代码进行求值，并不会比直接编写这样的代码带来更多真正的好处。默认情况下，如果 eval(..) 执行的代码字符串包含一个或多个声明（变量或函数）的话，这个动作就会修改这个 eval(..) 所在的词法作用域。技术上讲，eval(..) 可以通过种种技巧（超出了我们这里的讨论范围）被“间接”调用，而使它在全局作用域的上下文中执行，以此修改全局作用域。但不论那种情况，eval(..) 都可以在运行时修改一个编写时的词法作用域。

**注意：** 当 eval(..) 被用于一个操作它自己的词法作用域的 strict 模式程序时，在 eval(..) 内部做出的声明不会实际上修改包围它的作用域。

1. function foo(str) {
2. "use strict";
3. eval( str );
4. console.log( a ); // ReferenceError: a is not defined
5. }
6. foo( "var a = 2" );

在 JavaScript 中还有其他的工具拥有与 eval(..) 非常类似的效果。setTimeout(..) 和 setInterval(..) 可以 为它们各自的第一个参数值接收一个字符串，其内容将会被 eval 为一个动态生成的函数的代码。这种老旧的，遗产行为早就被废弃了。别这么做！

new Function(..) 函数构造器类似地为它的 **最后** 一个参数值接收一个代码字符串，来把它转换为一个动态生成的函数（前面的参数值，如果有的话，将作为新函数的形式参数）。这种函数构造器语法要比 eval(..) 稍稍安全一些，但在你的代码中它仍然应当被避免。

###### With

JavaScript 的另一个使人皱眉头（而且现在被废弃了！），而且可以欺骗词法作用域的特性是 with 关键字。有许多种合法的方式可以讲解 with，但是我在此选择从它如何与词法作用域互动并影响词法作用域的角度来讲解它。

讲解 with 的常见方式是作为一种缩写，来引用一个对象的多个属性，而 不必 每次都重复对象引用本身。

例如：

1. var obj = {
2. a: 1,
3. b: 2,
4. c: 3
5. };
6. // 重复“obj”显得更“繁冗”
7. obj.a = 2;
8. obj.b = 3;
9. obj.c = 4;
10. // “更简单”的缩写
11. with (obj) {
12. a = 3;
13. b = 4;
14. c = 5;
15. }

然而，这里发生的事情要比只是一个对象属性访问的便捷缩写要多得多。考虑如下代码

1. function foo(obj) {
2. with (obj) {
3. a = 2;
4. }
5. }
6. var o1 = {
7. a: 3
8. };
9. var o2 = {
10. b: 3
11. };
12. foo( o1 );
13. console.log( o1.a ); // 2
14. foo( o2 );
15. console.log( o2.a ); // undefined
16. console.log( a ); // 2 -- 哦，全局作用域被泄漏了！

总结:如果eval(..) 或 with 出现，那么它 *将* 做的几乎所有的优化都会变得没有意义，所以它就会简单地根本不做任何优化。

你的代码几乎肯定会趋于运行的更慢，只因为你在代码的任何地方引入了一个了 eval(..) 或 with。无论 *引擎* 将在努力限制这些悲观臆测的副作用上表现得多么聪明，**都没有任何办法可以绕过这个事实：没有优化，代码就运行的更慢。**

#### 隐藏于普通作用域

为什么要‘隐藏’变量和函数是一种有用的技术？

“最低权限原则”的设计原则引起的[^note-leastprivilege]，有时也被称为“最低授权”或“最少曝光”。这个原则规定，在软件设计中，比如一个模块/对象的API，你应当只暴露所需要的最低限度的东西，而“隐藏”其他的一切。

这个原则可以扩展到用哪个作用域来包含变量和函数的选择。如果所有的变量和函数都在全局作用域中，它们将理所当然地对任何嵌套的作用域来说都是可访问的。但这回违背“最少……”原则，因为你（很可能）暴露了许多你本应当保持为私有的变量和函数，而这些代码的恰当用法是不鼓励访问这些变量/函数的。

1. function doSomething(a) {
2. b = a + doSomethingElse( a \* 2 );
3. console.log( b \* 3 );
4. }
5. function doSomethingElse(a) {
6. return a - 1;
7. }
8. var b;
9. doSomething( 2 ); // 15
10. 在这个代码段中，变量 b 和函数 doSomethingElse(..) 很可能是 doSomething(..) 如何工作的“私有”细节。允许外围的作用域“访问” b 和 doSomethingElse(..) 不仅没必要而且可能是“危险的”，因为它们可能会以种种意外的方式，有意或无意地被使用，而这也许违背了 doSomething(..) 假设的前提条件。
11. 一个更“恰当”的设计是讲这些私有细节隐藏在doSomething(..)的作用域内部，比如：
12. function doSomething(a) {
13. function doSomethingElse(a) {
14. return a - 1;
15. }
16. var b;
17. b = a + doSomethingElse( a \* 2 );
18. console.log( b \* 3 );
19. }
20. doSomething( 2 ); // 15

现在，b 和 doSomethingElse(..) 对任何外界影响都是不可访问的，而是仅仅由 doSomething(..) 控制。它的功能和最终结果不受影响，但是这种设计将私有细节保持为私有的，这通常被认为是好的软件。

##### 避免冲突

将变量和函数“隐藏”在一个作用域内部的另一个好处是，避免两个同名但用处不同的标识符之间发生无意的冲突。冲突经常导致值被意外地覆盖。

例如：

1. function foo() {
2. function bar(a) {
3. i = 3; // 在外围的for循环的作用域中改变`i`
4. console.log( a + i );
5. }
6. for (var i=0; i<10; i++) {
7. bar( i \* 2 ); // 噢，无限循环！
8. }
9. }
10. foo();

#### 变量提升

原理： 在代码执行之前，所有的声明、变量、函数，都会首先处理。在JavaScript中var a = 2; 实际上是两个语句：var a; 和 a = 2; 第一个语句，声明，是在编译阶段被处理的。第二个语句，赋值，为了执行阶段而留在 **原处**。换句话说，**先有蛋（声明），后有鸡（赋值）**

**注意：**只有声明本身被提升了，而任何赋值或者其他的执行逻辑都是被留在原处的。

1. console.log( a ); // undefined
2. var a = 2;

还需要注意的是提升是 **以作用域为单位的。**所以虽然我们的前一个代码段被简化为仅含有全局作用域，但是我们现在检视的函数foo(..)本身展示了，var a被提升至foo(..)的顶端（很明显，不是程序的顶端）。所以这个过程可以更准确的解释为：

1. function foo() {
2. var a;
3. console.log( a ); // undefined
4. a = 2;
5. }
6. foo();

### 作用域与闭包

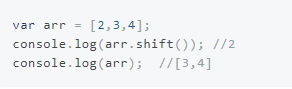
**在 JavaScript 中闭包无所不在，你只是必须认出它并接纳它**

# ES6

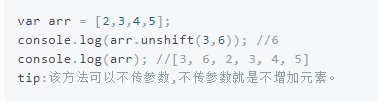
## es6对象、数组、字符串

### 数组相关方法

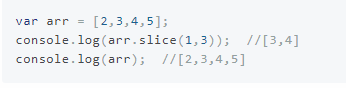
* concat() ：concat() 方法用于链接两个或者多个数组。该方法不会改变现有的数组，仅会返**回被连接数组的一个副本**
* join() : join() 方法用于把数组中所有的元素放进一个字符串。元素师通过指定的分隔符进行分隔的， 默认使用‘，’号分隔，**不改变原数组**。
* push() ： push() 方法可以向数组的末尾添加一个或者多个元素，并返回新的长度。末尾添加，**返回的是长度，** **会改变原数组**。
* pop() : pop() 方法用于删除并返回最后一个元素，**返回最后一个元素**，**改变原数组**
* shift(): shift() 方法用于把数组的第一个元素从其中删除， 并返回第一个元素的值， **返回第一个元素，改变原数组。**



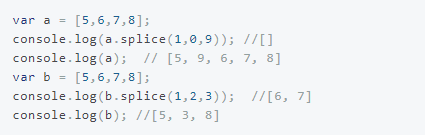
* unshift(): unshift()方法可以向数组的开头添加一个或者多个元素，并返回新的长度。**返回新的长度，改变原数组。**



* slice(): slice()方法**返回一个新的数组**，包含start 到 end（不包含该元素）的arrayObject 的元素。返回选定的元素，该方法**不会修改原数组**。



* **splice():** splice() 方法可用于删除从指定数组下标开始的零个或者多个元素，并用参数列表中声明的一个或者多个值来替换那些被删除的元素。如果arrayObject中删除了元素，则**返回被删除元素的数组**。**Splice() 方法会直接对数组进行修改**。



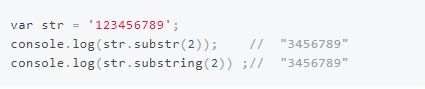
* substring() 和 substr() :

**相同点**：

如果只是写一个参数，两者作用相同；都是截取字符串从当前下标以后直到字符串最后的字符片段。

Substr(startIndex)

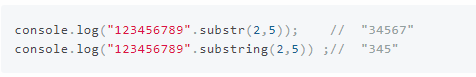
Substring(startIndex);



**不同点**：

Substr(startIndex, length) ： 第二个参数是截取字符串的长度（从起始点截取某个长度的字符串）；

Substring(startIndex,endIndex) : 第二个参数是截取字符串的最终的下标（截取2个位置之间的字符串‘含头不含尾’）

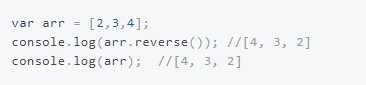


* Sort(): sort() 方法可用于数组排序



注意： 字母大小写不同的字符串要转换成大小写相同的再进行排序

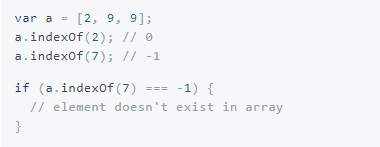
* Reverse(): reverse() 方法用于颠倒数组中元素的顺序。**返回的是颠倒后的数组**，**会改变原数组**。



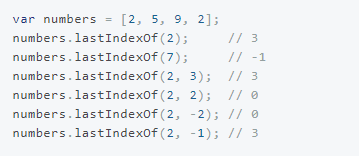
* indexOf 和lastIndexOf : 都接受两个参数： 查找的值、查找起始位置

不存在： 返回 -1； 存在： 返回位置。 indexOf是从前往后查，lastIndexOf 是从后往前查找。

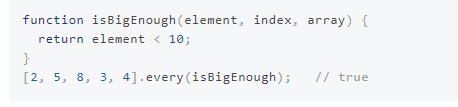
**indexOf：**



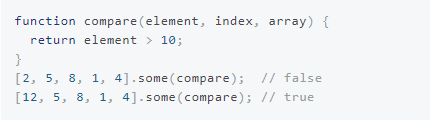
lastIndexOf：



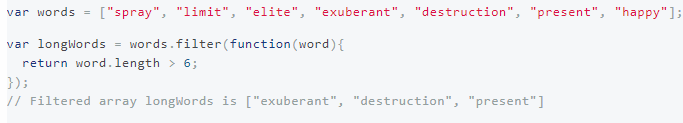
* every：对数组的每一项都运行给定的函数，每一项都返回ture，则返回ture



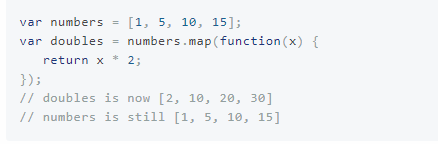
* Some：对数组的每一项都运行给定的函数，任意一项返回true，则返回true



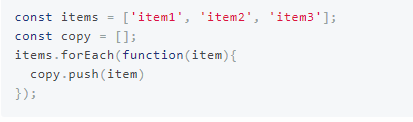
* Filter：对数组的每一项都运行给定的函数， 返回结果为true的项组成的数组。



* map： 对数组的每一项都运行给定的函数， 返回每次函数调用的结果组成一个新的数组



* **forEach数组遍历：**

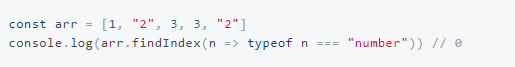


#### Es6新操作数组的方法

* Find()：传入一个回调函数，找到数组中符合当前搜索规则的第一个元素，返回**符合搜索规则的第一个元素**，并终止搜索。

在这里插入图片描述

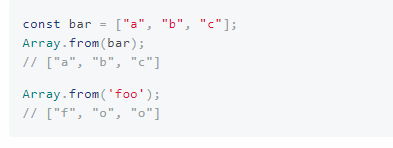
* findIndex(): 传入一个回调函数，找到数组中符合当前搜索规则的第一个元素，返回**符合搜索规则的第一个元素的下标，**终止搜索。



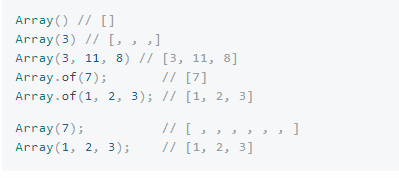
* fill(): 用新元素替换掉数组内的元素， 可以指定替换下标范围。

在这里插入图片描述

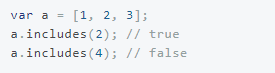
* from：将类似数组的对象（array-like object） 和可遍历（iterable）的对象转为真正的数组



* of：用于将一组值，转换为数组， 这个方法主要目的是弥补数组构造函数Array()的不足。因为参数个数的不同，会导致Array() 的行为有差异。



* includes：判断数组中是否存在该元素，参数： 查找的值、起始位置， 可以替换ES5时代的indexOf判断方法。（indexOf判断元素是否为NaN，会判断错误）



### 字符串相关方法总结

* charAt(): charAt()根据索引，返回字符串中指定的字符串
* indexOf() : indexOf方法返回字符串中指定文本首次出现的索引（位置），如果未找到则返回-1，第二个参数为起始位置
* lastIndexOf(): lastIndexOf() 方法返回指定文本在字符串中最后一次出现的索引如果未找到则返回 -1， 第二个参数威起始位置
* search(): search() 方法搜索特定的值的字符串， 并返回匹配的位置；但无法接受第二个参数
* slice() 方法： slice() 方法 提取字符串的某个部分并在新字符串中返回被提取的部分，该方法设置两个参数：起始位置，终止位置，（如果某个参数为负数，则从字符串结尾开始计数）
* substring() : substring() 方法，类似于slice() 方法，不同之处：substring() 无法接受负数为索引。
* substr(): substr() 方法类似于slice() 方法，不同之处：substr() 第二个参数是截取长度。
* replace(): replace() 方法用另外一个值替换在字符串指定位置的值，replace() 方法对大小写敏感，可以使用/I （对大小写不敏感）

str = "Please visit Microsoft!";

var n = str.replace(/MICROSOFT/i, "W3School");

如需匹配所有，请使用正则表达式 g 标志（用于全局搜索）

* toUpperCase() ： toUpperCase() 将字符串转换为大写
* toLowerCase(): toLowerCase() 将字符串转换为小写
* concat(): concat() 方法链接两个或者多个字符串
* trim() : trim() 方法删除字符串两端的空白符（Internet Explorer 8 或者更低版本不支持）

### 字符串<=>数组

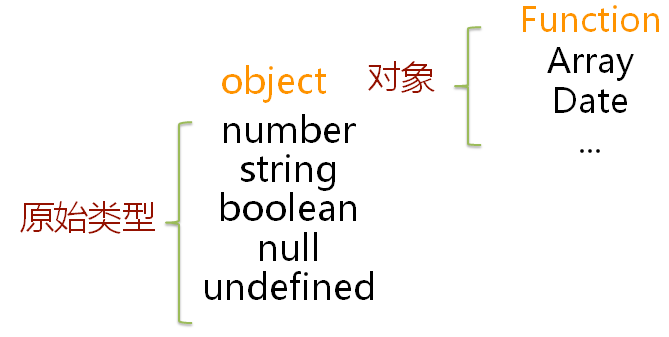
#### 字符串=>数组

1、String对象中的split()方法

#### 数组=>字符串

* 1. Array对象的join()方法
  2. toString()
  3. toLocalString() 把数组转换成本地约定的字符串

### 数据类型



#### 概念：

原始类型（基本类型）： 按值访问，可以操作保存在变量中实际的值，原始类型汇总中null和undefined比较特殊。

引用类型： 引用类型的值是保存在内存中的对象。

#### 本质

基本数据类型（存放在栈（stack））

Undefined、null、string、boolean、number、symbol(ES6)

引用数据类型

对象Object、数组Array 、函数Function、Date等（存放在堆（heap）中的对象）

#### 区别：

**A声明变量的存储分配**

1）原始值：存储在栈（stack）中的简单数据段，也就是说，它们的值存储的是变量的访问位置；

1. 引用值：存储在堆（heap）中，也就是说，存储在变量处的值是一个指针（point），指向存储对象的内存地址。

这是因为：引用值的大小会改变，所以不能把它放在栈中，否则会降低变量查寻的速度。相反，放在变量的栈空间中的值是该对象存储在堆中的地址。

地址的大小是固定的，所以把它存储在栈中对变量性能无任何负面影响。

**B 不同的内存分配机制带来不同的访问机制**

1）在javascript中是不允许直接访问保存在堆内存中的对象的，所以在访问一个对象时，首先得到的是这个对象在堆内存中的地址，然后再按照这个地址去获得这个对象中的值，这就是传说中的按引用访问。

2）而原始类型的值则是可以直接访问到的。

**C 复制变量时的不同**

1）原始值：在将一个保存着原始值的变量复制给另一个变量时，会将原始值的副本赋值给新变量，此后这两个变量是完全独立的，他们只是拥有相同的value而已。

2）引用值：在将一个保存着对象内存地址的变量复制给另一个变量时，会把这个内存地址赋值给新变量，也就是说这两个变量都指向了堆内存中的同一个对象，他们中任何一个作出的改变都会反映在另一个身上。

#### 隐式转换

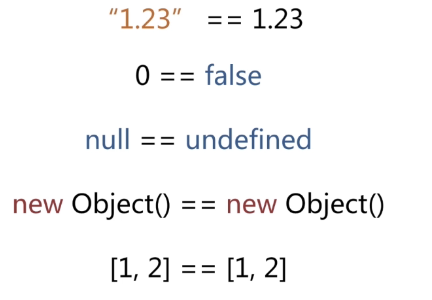
##### + 和 –

巧用+和-规则转换类型

把变量转换成数字：num-0;

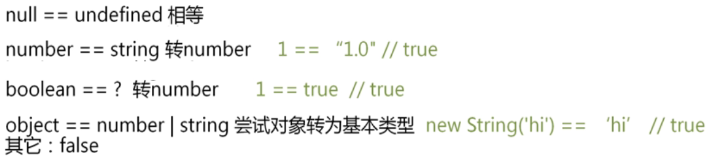
把变量转换成字符串：num+'';

##### A == B



类型相同 ， 使用 同 ===

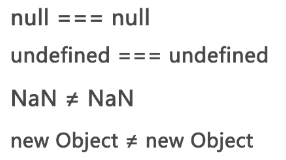
类型不同， 尝试类型转换 和 == 比较：



##### a === b

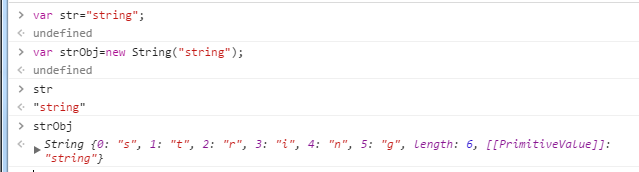
类型不同， 返回FALSE

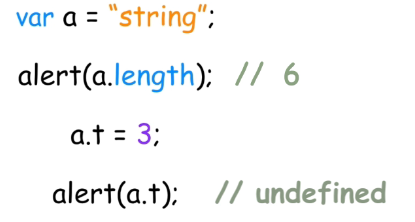
类型相同：



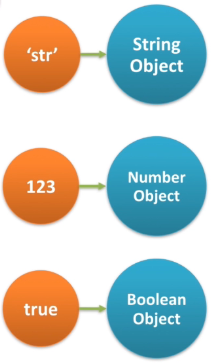
#### 包装对象

基本类型中的number、string和Boolean 都有对应的包装类型。





把一个基本类型尝试用对象的方式使用它的时候，比如访问length属性，或者增加一些属性的操作时，JavaScript会把这些基本类型转化为对应的包装类型对象。完成这样一个访问比如a.length返回以后或者a.t设置了以后，这个临时对象就会被销毁。所以a.t赋值了3以后，再去输出a.t 值是undefined。



St、number和true都有包装类型，所以可以把数字用括号括起来调用toString() 方法。

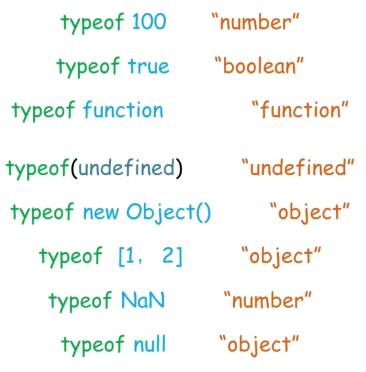
#### 类型检测

JavaScript中类型检测方法：

* Typeof
* Instanceof
* Object.prototype.toString
* Constructor
* Duck type

##### Typeof

最常见的就是typeof



比较特殊的是typeof null 返回的是‘object’

历史原因，规范尝试修改typeof null 返回 ‘null’修改完大量网站无法访问，为了兼容，或者说历史原因返回 ‘object’

Typeof 对基本类型和函数对象很方便，但是其他类型就没办法了。

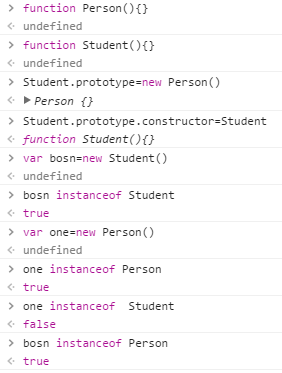
判断一个对象是不是数组？ 用typeof 返回 ‘object’ 。 对对象的判断常用instanceof。

##### Instanceof

基本原型链操作。Object instanceof Object。

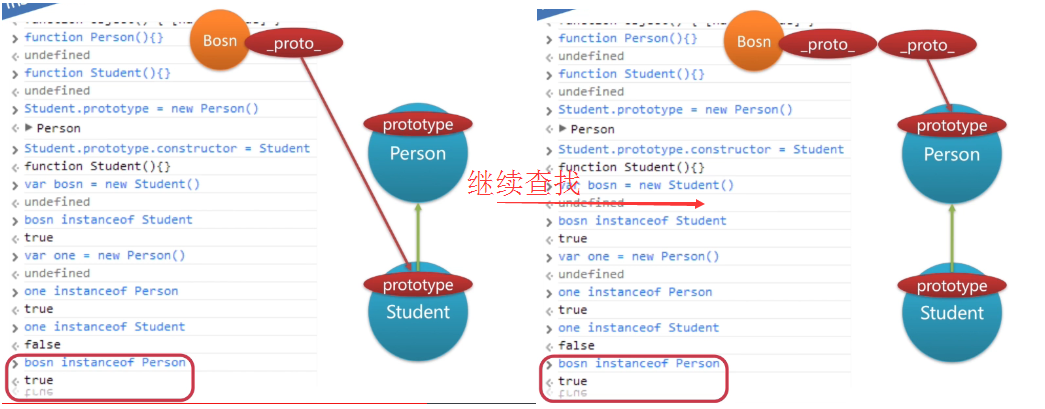
左操作数为对象，不是返回FALSE，有操作必须是函数对象或者函数构造器，不是就返回typeError异常。

原理： 判断左边的左操作数的对象的原型链上是否有右边这个构造函数的prototype属性。

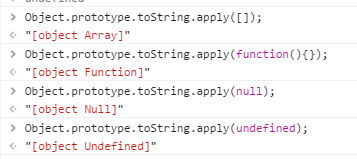


任何一个构造函数都有一个prototype对象属性，这个对象属性将用作new 出来的对象的原型。

Bosn instanceof Person 的时候发现bosn 的原型也就是Student.prototype 不等于 person.prototype，所以原型链还会向上查找，bosn的原型的原型等于person。Prototype所以返回true



##### Object.prototype.toString



需要注意的是IE6/7/8 中，Object.prototype.toString.apply(null)返回的是 ‘[Object，Object]’

##### Constructor



任何对象都有constructor属性，继承自原型，constructor 会指向构造这个对象的构造器或者构造函数。Constructor 可以被改写，所以使用要小心。

##### Duck type

比如不知道一个对象是不是数组，可以判断他的length是不是数字，他是不是有join，push这样一些数组的方法。通过一些特征判断对象是否属于某些类型，这个有时候也常用。

小结：





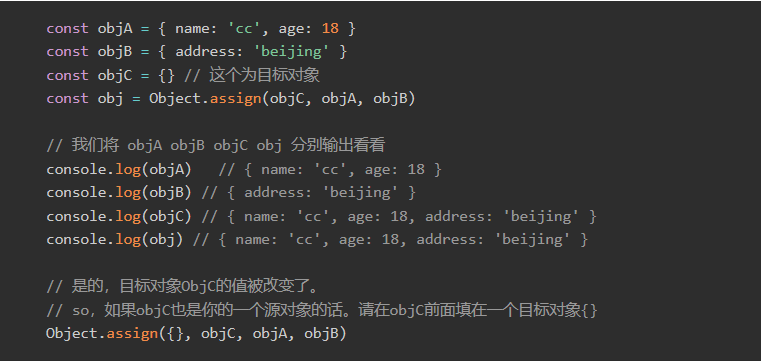
### 对象

对象连接Object.assign()

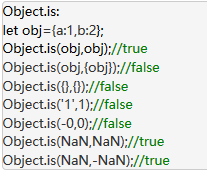
用于对象的合并，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

注意点：

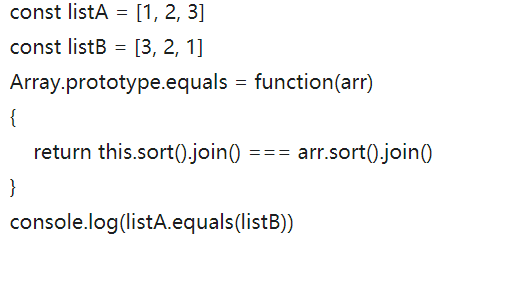
Object.assign 方法实行的是浅拷贝



[Object.is方法比较两个对象的值是否相等](https://blog.csdn.net/weixin_30567471/article/details/95285835" \t "_blank)



isEqual 方法用来比较对象或者变量是否相等



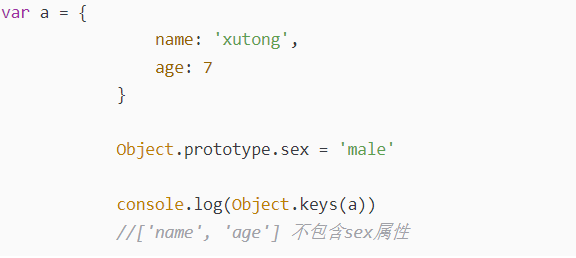
#### 对象遍历

（1）for...in

for...in循环遍历对象自身的和继承的可枚举属性（不含Symbol属性）。

（2）Object.keys(obj)

Object.keys返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含Symbol属性）。



（3）Object.getOwnPropertyNames(obj)

Object.getOwnPropertyNames返回一个数组，包含对象自身的所有属性（不含Symbol属性，但是包括不可枚举属性）。

（4）Object.getOwnPropertySymbols(obj)

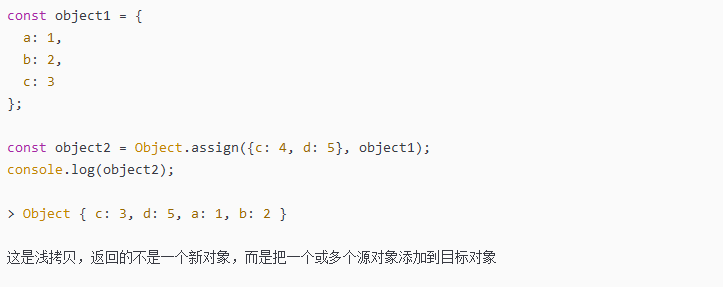
Object.getOwnPropertySymbols返回一个数组，包含对象自身的所有Symbol属性。

（5）Reflect.ownKeys(obj)

Reflect.ownKeys返回一个数组，包含对象自身的所有属性，不管是属性名是Symbol或字符串，也不管是否可枚举。

#### 对象的深浅copy

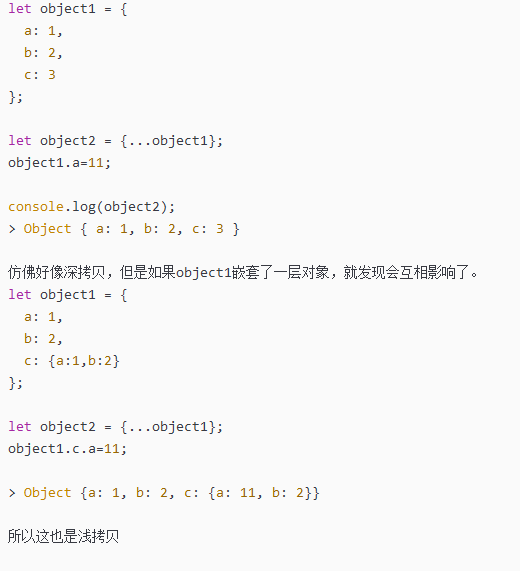
##### 通过Object.assign()进行copy（浅copy）



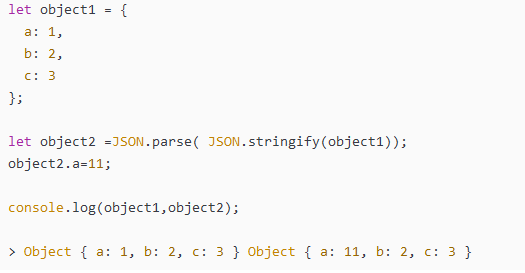
##### Object.getOwnPropertyDescriptors()配合Object.create()方法(浅copy)

1. const clone = Object.create(Object.getPrototypeOf(obj),
2. Object.getOwnPropertyDescriptors(obj));
3. // 或者
4. const shallowClone = (obj) => Object.create(
5. Object.getPrototypeOf(obj),
6. Object.getOwnPropertyDescriptors(obj)
7. );

##### 结构赋值（浅copy）

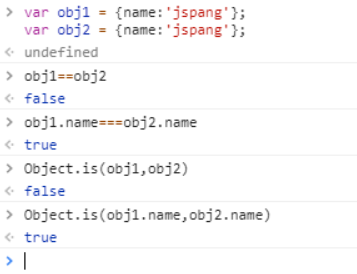


##### 深copy

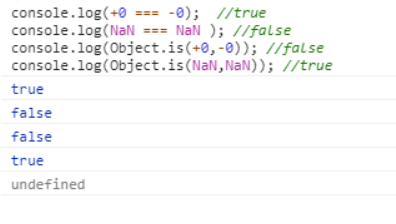


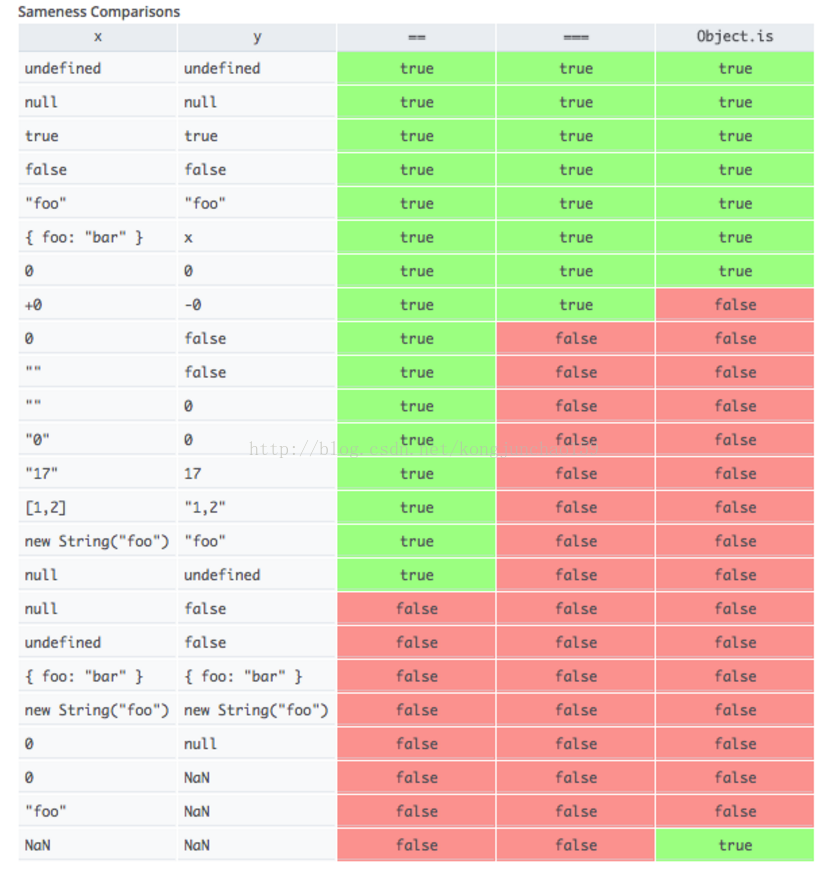
#### 对象比较

1. Object.is()



注意特例：





#### 链式判断运算

编程实务中，如果读取对象内部的某个属性，往往需要判断一下该对象是否存在。比如，要读取message.body.user.firstName，安全的写法是写成下面这样。

1. const firstName = (message
2. && message.body
3. && message.body.user
4. && message.body.user.firstName) || 'default';

或者使用三元运算符?:，判断一个对象是否存在。

1. const fooInput = myForm.querySelector('input[name=foo]')
2. const fooValue = fooInput ? fooInput.value : undefined

这样的层层判断非常麻烦，因此 [ES2020](https://github.com/tc39/proposal-optional-chaining) 引入了“链判断运算符”（optional chaining operator）?.，简化上面的写法。

1. const firstName = message?.body?.user?.firstName || 'default';
2. const fooValue = myForm.querySelector('input[name=foo]')?.value

上面代码使用了?.运算符，直接在链式调用的时候判断，左侧的对象是否为null或undefined。如果是的，就不再往下运算，而是返回undefined

##### 链判断运算符有三种用法。

* obj?.prop // 对象属性
* obj?.[expr] // 同上
* func?.(...args) // 函数或对象方法的调用

下面是判断对象方法是否存在，如果存在就立即执行的例子。

1. iterator.return?.()

上面代码中，iterator.return如果有定义，就会调用该方法，否则直接返回undefined。

对于那些可能没有实现的方法，这个运算符尤其有用。

1. if (myForm.checkValidity?.() === false) {
2. // 表单校验失败
3. return;
4. }

上面代码中，老式浏览器的表单可能没有checkValidity这个方法，这时?.运算符就会返回undefined，判断语句就变成了undefined === false，所以就会跳过下面的代码。

下面是这个运算符常见的使用形式，以及不使用该运算符时的等价形式。

1. a?.b
2. // 等同于
3. a == null ? undefined : a.b
4. a?.[x]
5. // 等同于
6. a == null ? undefined : a[x]
7. a?.b()
8. // 等同于
9. a == null ? undefined : a.b()
10. a?.()
11. // 等同于
12. a == null ? undefined : a()

上面代码中，特别注意后两种形式，如果a?.b()里面的a.b不是函数，不可调用，那么a?.b()是会报错的。a?.()也是如此，如果a不是null或undefined，但也不是函数，那么a?.()会报错。

##### 注意点。

（1）短路机制

1. a?.[++x]
2. // 等同于
3. a == null ? undefined : a[++x]

上面代码中，如果a是undefined或null，那么x不会进行递增运算。也就是说，链判断运算符一旦为真，右侧的表达式就不再求值。

（2）delete 运算符

1. delete a?.b
2. // 等同于
3. a == null ? undefined : delete a.b

上面代码中，如果a是undefined或null，会直接返回undefined，而不会进行delete运算。

（3）括号的影响

如果属性链有圆括号，链判断运算符对圆括号外部没有影响，只对圆括号内部有影响。

1. (a?.b).c
2. // 等价于
3. (a == null ? undefined : a.b).c

上面代码中，?.对圆括号外部没有影响，不管a对象是否存在，圆括号后面的.c总是会执行。

一般来说，使用?.运算符的场合，不应该使用圆括号。

（4）报错场合

以下写法是禁止的，会报错。

1. // 构造函数
2. new a?.()
3. new a?.b()
4. // 链判断运算符的右侧有模板字符串
5. a?.`{b}`
6. a?.b`{c}`
7. // 链判断运算符的左侧是 super
8. super?.()
9. super?.foo
10. // 链运算符用于赋值运算符左侧
11. a?.b = c

（5）右侧不得为十进制数值

为了保证兼容以前的代码，允许foo?.3:0被解析成foo ? .3 : 0，因此规定如果?.后面紧跟一个十进制数字，那么?.不再被看成是一个完整的运算符，而会按照三元运算符进行处理，也就是说，那个小数点会归属于后面的十进制数字，形成一个小数。

#### Null判断运算符

读取对象属性的时候，如果某个属性的值是null或undefined，有时候需要为它们指定默认值。常见做法是通过||运算符指定默认值。

1. const headerText = response.settings.headerText || 'Hello, world!';
2. const animationDuration = response.settings.animationDuration || 300;
3. const showSplashScreen = response.settings.showSplashScreen || true;

上面的三行代码都通过||运算符指定默认值，但是这样写是错的。开发者的原意是，只要属性的值为null或undefined，默认值就会生效，但是属性的值如果为空字符串或false或0，默认值也会生效。

为了避免这种情况，[ES2020](https://github.com/tc39/proposal-nullish-coalescing) 引入了一个新的 Null 判断运算符??。它的行为类似||，但是只有运算符左侧的值为null或undefined时，才会返回右侧的值。

1. const headerText = response.settings.headerText ?? 'Hello, world!';
2. const animationDuration = response.settings.animationDuration ?? 300;
3. const showSplashScreen = response.settings.showSplashScreen ?? true;

上面代码中，默认值只有在属性值为null或undefined时，才会生效。

这个运算符的一个目的，就是跟链判断运算符?.配合使用，为null或undefined的值设置默认值。

1. const animationDuration = response.settings?.animationDuration ?? 300;

上面代码中，response.settings如果是null或undefined，就会返回默认值300。

这个运算符很适合判断函数参数是否赋值。

1. function Component(props) {
2. const enable = props.enabled ?? true;
3. // …
4. }

??有一个运算优先级问题，它与&&和||的优先级孰高孰低。现在的规则是，如果多个逻辑运算符一起使用，必须用括号表明优先级，否则会报错。

1. // 报错
2. lhs && middle ?? rhs
3. lhs ?? middle && rhs
4. lhs || middle ?? rhs
5. lhs ?? middle || rhs

上面四个表达式都会报错，必须加入表明优先级的括号。

1. (lhs && middle) ?? rhs;
2. lhs && (middle ?? rhs);
3. (lhs ?? middle) && rhs;
4. lhs ?? (middle && rhs);
5. (lhs || middle) ?? rhs;
6. lhs || (middle ?? rhs);
7. (lhs ?? middle) || rhs;
8. lhs ?? (middle || rhs);

### 数组

#### 数组排序

按照数值大小进行排序-升序

[1, 8, 5].sort((a, b) => {

return a-b; // 从小到大排序

});

// [1, 5, 8]

按照数值大小进行排序-降序

[1, 8, 5].sort((a, b) => {

return b-a; // 从大到小排序

});

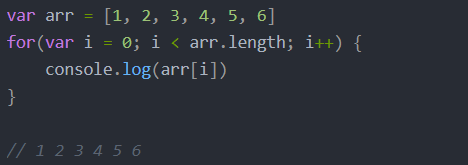
// [8, 5, 1]

如果打印 数组.push(\*\*\*) 会返回数组长度

#### 数组遍历

0、for：

最简单的一种循环遍历方法，也是使用频率最高的一种，可优化

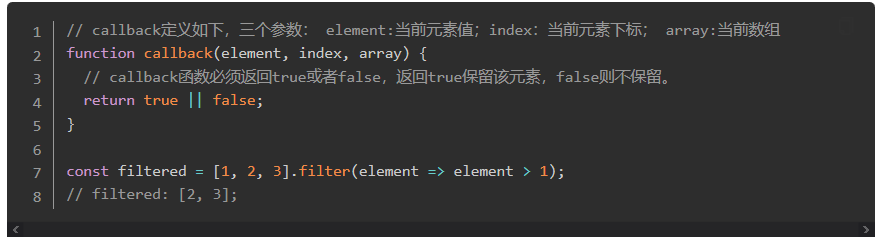


1. array.filter(callback, thisArg)方法使用指定的函数测试所有元素,并创建一个包含所有通过测试的元素的新数组。

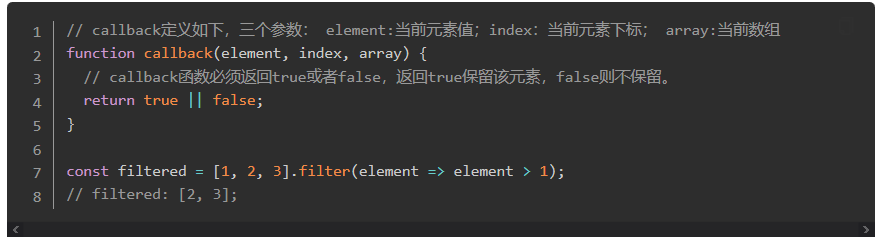
参数说明：

• callback 用来测试数组的每个元素的函数，返回true表示保留该元素（通过测试），false则不保留。

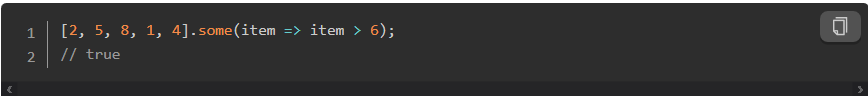
• thisArg 可选。执行 callback 时的用于 this 的值。



2、array.every(callback[, thisArg])方法检测数组中的每一个元素是否都通过了callback测试，全部通过返回true，否则返回false

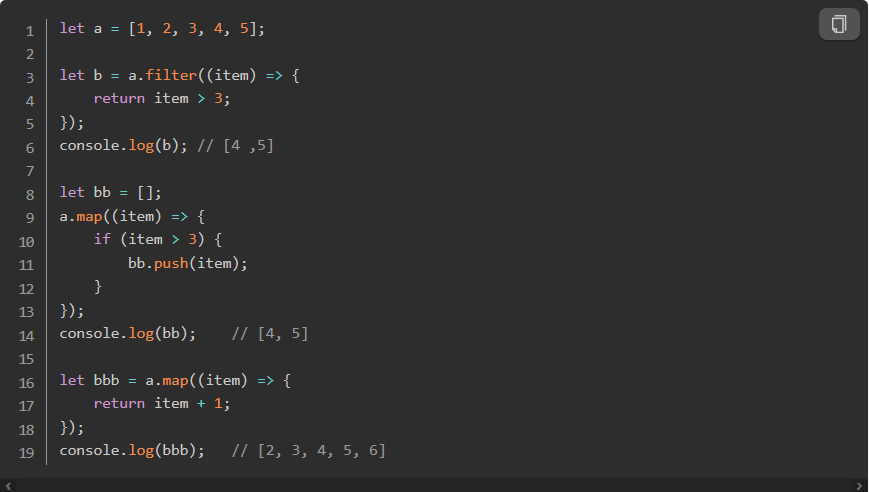


3、array.some(callback[, thisArg])判断数组中是否包含可以通过callback测试的元素，与every不同的是，这里只要某一个元素通过测试，即返回true。callback定义同上。



4、 array.map(callback[, thisArg])方法返回一个由原数组中的每个元素调用callback函数后的返回值组成的新数组。

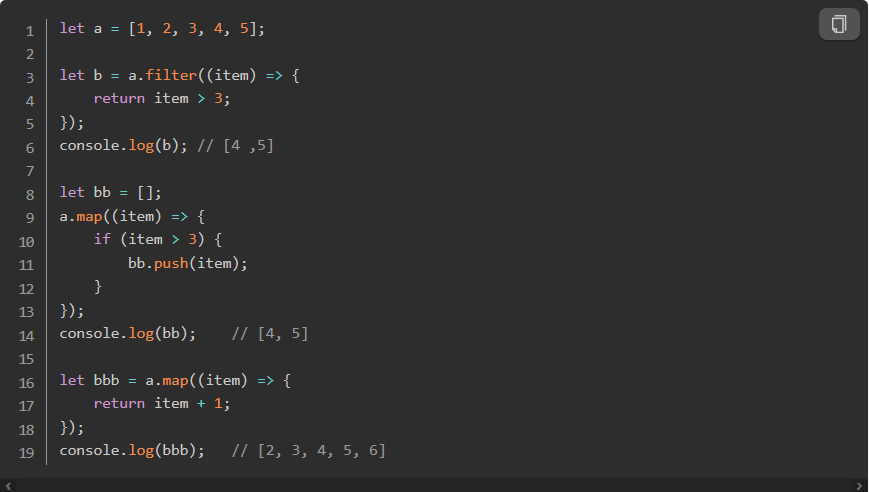
注意：遍历每一个元素并且返回对应的元素(可以返回处理后的元素) (map 映射 一一 对应)  
    返回的新数组和旧数组的长度是一样的  
    使用比较广泛，但其性能还不如 forEach



5、 array.forEach(callbak)为数组的每个元素执行对应的方法

注意：

数组里的元素个数有几个，该方法里的回调就会执行几次第一个参数是数组里的元素，第二个参数为数组里元素的索引，第三个参数则是它自己数组自带的遍历方法，虽然使用频率略高，但是性能仍然比普通循环略低

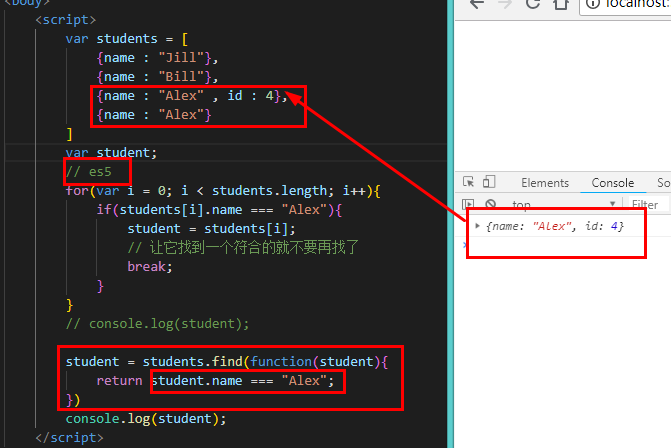


6、find（）方法

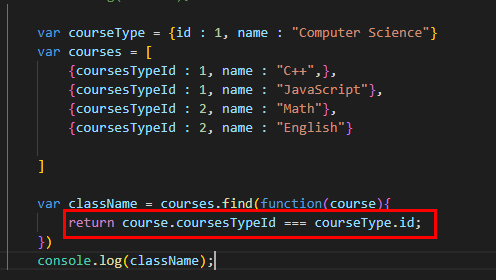
①find（）方法只会找到第一个符合的，找到之后就会直接返回，就算下面还有符合要求的，也不会再找下去

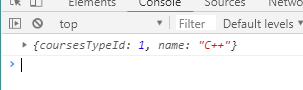
②应用场景

1. 在对象数组中，找到符合要求的对象,只能找到第一个符合要求的



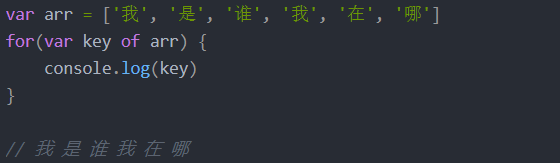
1. 两个对象数组，找A数组中有B数组的ID的元素





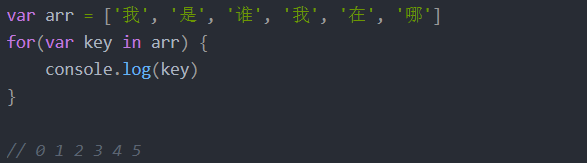
7、for…of…

虽然性能要好于 for…in…，但仍然比不上普通的 for 循环（不能循环对象）



8、for…in…

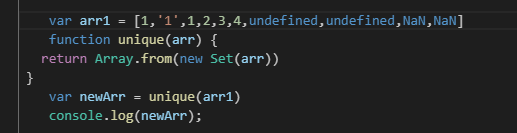
这个循环用的人也很多，但是效率最低（输出的 key 是数组索引）



更多详情

<https://www.jianshu.com/p/e1b43e56de08>

数组去重

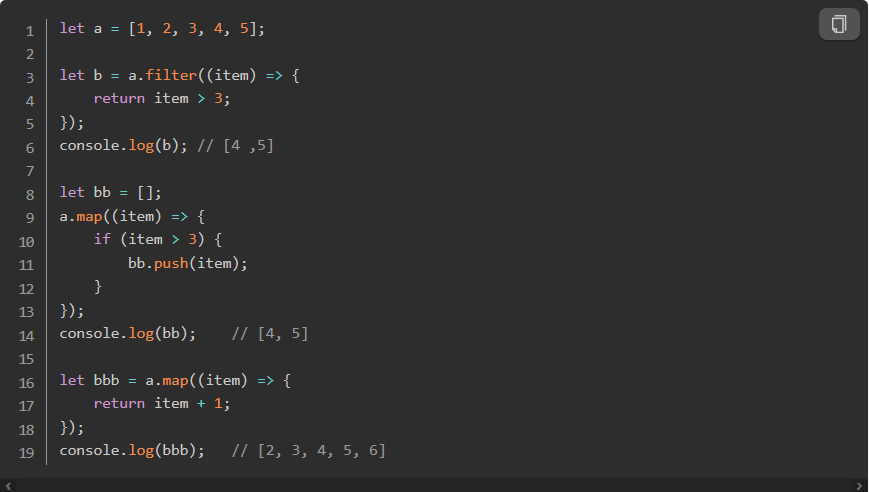


#### 转数组

1、Array.from()方法

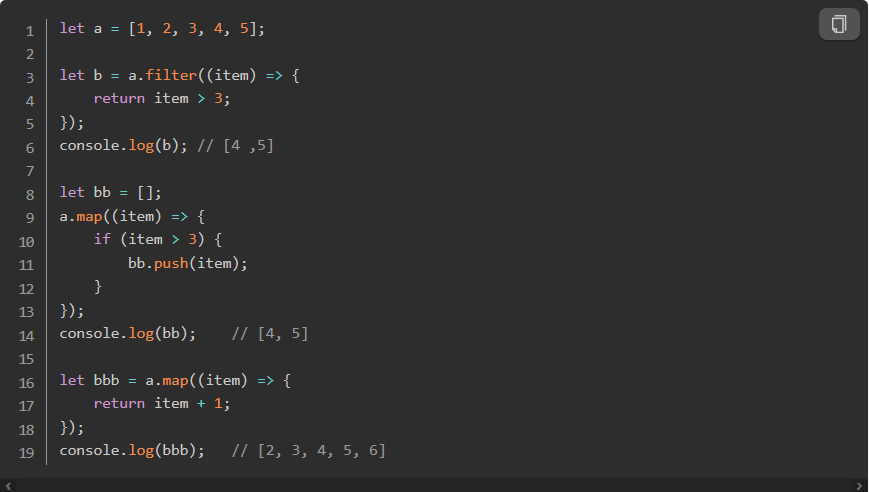
Array.from()方法是用于将类似数组的对象（即有length属性的对象）和可遍历对象转为真正的数组。

比如，使用•Array.from()•方法，可以轻松将•JSON•数组格式转为数组。



2、Array.of()方法

Array.of()方法是将一组值转变为数组。



### 字符串

#### startsWith（给定文本）

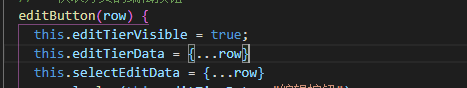
判断是否以给定文本开始

#### endsWith（给定文本）

判断是否以给定文本结束

#### padStart（填充后长度，填充字符）

#### padEnd（填充后的长度，填充字符）



当使用两个变量拿值时会出现数据的联动（对象1改变对象2也会跟着改变）

需要对数据进行处理让两个对象成为单独对象

#### includes（）

返回布尔值，表示是否找到了参数字符串。

#### Repeat（）

repeat方法返回一个新字符串，表示将原字符串重复n次。

1. 'x'.repeat(3) // "xxx"
2. 'hello'.repeat(2) // "hellohello"
3. 'na'.repeat(0) // ""

### 数值

#### Number.isFinite(), Number.isNaN()

ES6 在Number对象上，新提供了Number.isFinite()和Number.isNaN()两个方法。

Number.isFinite()用来检查一个数值是否为有限的（finite），即不是Infinity。

1. Number.isFinite(15); // true
2. Number.isFinite(0.8); // true
3. Number.isFinite(NaN); // false
4. Number.isFinite(Infinity); // false
5. Number.isFinite(-Infinity); // false
6. Number.isFinite('foo'); // false
7. Number.isFinite('15'); // false
8. Number.isFinite(true); // false

注意，如果参数类型不是数值，Number.isFinite一律返回false。

Number.isNaN()用来检查一个值是否为NaN。

1. Number.isNaN(NaN) // true
2. Number.isNaN(15) // false
3. Number.isNaN('15') // false
4. Number.isNaN(true) // false
5. Number.isNaN(9/NaN) // true
6. Number.isNaN('true' / 0) // true
7. Number.isNaN('true' / 'true') // true

#### Number.parseInt(), Number.parseFloat()

ES6 将全局方法parseInt()和parseFloat()，移植到Number对象上面，行为完全保持不变。

1. // ES5的写法
2. parseInt('12.34') // 12
3. parseFloat('123.45#') // 123.45
4. // ES6的写法
5. Number.parseInt('12.34') // 12
6. Number.parseFloat('123.45#') // 123.45

这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

1. Number.parseInt === parseInt // true
2. Number.parseFloat === parseFloat // true

#### Number.isInteger()

Number.isInteger()用来判断一个数值是否为整数。

1. Number.isInteger(25) // true
2. Number.isInteger(25.1) // false

JavaScript 内部，整数和浮点数采用的是同样的储存方法，所以 25 和 25.0 被视为同一个值。

1. Number.isInteger(25) // true
2. Number.isInteger(25.0) // true

如果参数不是数值，Number.isInteger返回false。

1. Number.isInteger() // false
2. Number.isInteger(null) // false
3. Number.isInteger('15') // false
4. Number.isInteger(true) // false

注意，由于 JavaScript 采用 IEEE 754 标准，数值存储为64位双精度格式，数值精度最多可以达到 53 个二进制位（1 个隐藏位与 52 个有效位）。如果数值的精度超过这个限度，第54位及后面的位就会被丢弃，这种情况下，Number.isInteger可能会误判。

1. Number.isInteger(3.0000000000000002) // true

#### 安全整数和 Number.isSafeInteger()

JavaScript 能够准确表示的整数范围在-2^53到2^53之间（不含两个端点），超过这个范围，无法精确表示这个值。

1. Math.pow(2, 53) // 9007199254740992
2. 9007199254740992 // 9007199254740992
3. 9007199254740993 // 9007199254740992
4. Math.pow(2, 53) === Math.pow(2, 53) + 1
5. // true

上面代码中，超出 2 的 53 次方之后，一个数就不精确了。

ES6 引入了Number.MAX\_SAFE\_INTEGER和Number.MIN\_SAFE\_INTEGER这两个常量，用来表示这个范围的上下限。

1. Number.MAX\_SAFE\_INTEGER === Math.pow(2, 53) - 1
2. // true
3. Number.MAX\_SAFE\_INTEGER === 9007199254740991
4. // true
5. Number.MIN\_SAFE\_INTEGER === -Number.MAX\_SAFE\_INTEGER
6. // true
7. Number.MIN\_SAFE\_INTEGER === -9007199254740991
8. // true

Number.isSafeInteger()则是用来判断一个整数是否落在这个范围之内。

1. Number.isSafeInteger('a') // false
2. Number.isSafeInteger(null) // false
3. Number.isSafeInteger(NaN) // false
4. Number.isSafeInteger(Infinity) // false
5. Number.isSafeInteger(-Infinity) // false
6. Number.isSafeInteger(3) // true
7. Number.isSafeInteger(1.2) // false
8. Number.isSafeInteger(9007199254740990) // true
9. Number.isSafeInteger(9007199254740992) // false
10. Number.isSafeInteger(Number.MIN\_SAFE\_INTEGER - 1) // false
11. Number.isSafeInteger(Number.MIN\_SAFE\_INTEGER) // true
12. Number.isSafeInteger(Number.MAX\_SAFE\_INTEGER) // true
13. Number.isSafeInteger(Number.MAX\_SAFE\_INTEGER + 1) // false

#### Math.trunc()

Math.trunc方法用于去除一个数的小数部分，返回整数部分。

1. Math.trunc(4.1) // 4
2. Math.trunc(4.9) // 4
3. Math.trunc(-4.1) // -4
4. Math.trunc(-4.9) // -4
5. Math.trunc(-0.1234) // -0

对于非数值，Math.trunc内部使用Number方法将其先转为数值。

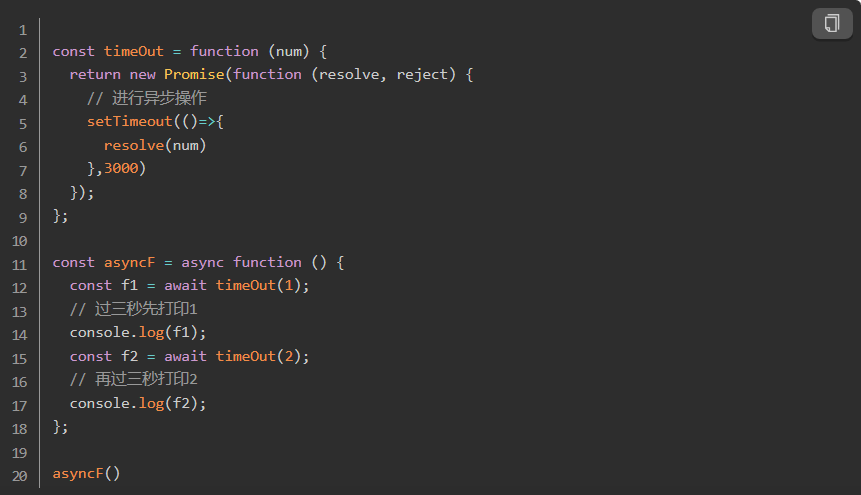
1. Math.trunc('123.456') // 123
2. Math.trunc(true) //1
3. Math.trunc(false) // 0
4. Math.trunc(null) // 0

## 请求

### 异步请求

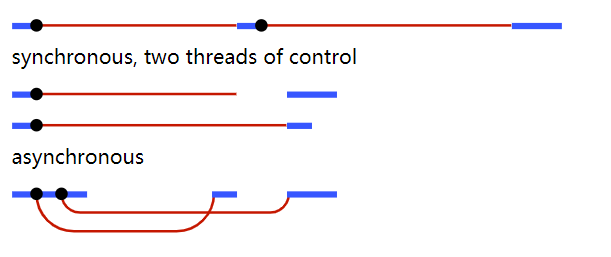
当请求返回数据在请求方法内能够正常打印赋值

在请求方法外面只拿到了默认初始值时



#### 异步同步花费时间对比

在下图中，粗线表示程序正常花费运行的时间，细线表示等待网络所花费的时间。 在同步模型中，网络所花费的时间是给定控制线程的时间线的一部分。 在异步模型中，从概念上讲，启动网络操作会导致时间轴中出现分裂。 启动该动作的程序将继续运行，并且该动作将与其同时发生，并在程序结束时通知该程序。



## ES6 module

ES6 模块的设计思想，是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。CommonJS 和 AMD 模块，都只能在运行时确定这些东西。比如，CommonJS 模块就是对象，输入时必须查找对象属性。

// CommonJS模块

let { stat, exists, readFile } = require('fs');

// 等同于

let \_fs = require('fs');

let stat = \_fs.stat, exists = \_fs.exists, readfile = \_fs.readfile;

由于 ES6 模块是编译时加载，使得静态分析成为可能。有了它，就能进一步拓宽 JavaScript 的语法，比如引入宏（macro）和类型检验（type system）这些只能靠静态分析实现的功能。

除了静态加载带来的各种好处，ES6 模块还有以下好处。

* 不再需要UMD模块格式了，将来服务器和浏览器都会支持 ES6 模块格式。目前，通过各种工具库，其实已经做到了这一点。
* 将来浏览器的新 API 就能用模块格式提供，不再必要做成全局变量或者navigator对象的属性。
* 不再需要对象作为命名空间（比如Math对象），未来这些功能可以通过模块提供。

### 严格模式

ES6 的模块自动采用严格模式，不管你有没有在模块头部加上"use strict";。

严格模式主要有以下限制。

* 变量必须声明后再使用
* 函数的参数不能有同名属性，否则报错
* 不能使用with语句
* 不能对只读属性赋值，否则报错
* 不能使用前缀0表示八进制数，否则报错
* 不能删除不可删除的属性，否则报错
* 不能删除变量delete prop，会报错，只能删除属性delete global[prop]
* eval不会在它的外层作用域引入变量
* eval和arguments不能被重新赋值
* arguments不会自动反映函数参数的变化
* 不能使用arguments.callee
* 不能使用arguments.caller
* 禁止this指向全局对象
* 不能使用fn.caller和fn.arguments获取函数调用的堆栈
* 增加了保留字（比如protected、static和interface）

上面这些限制，模块都必须遵守。由于严格模式是 ES5 引入的，不属于 ES6，所以请参阅相关 ES5 书籍，本书不再详细介绍了。

### Export命令

模块功能主要由两个命令构成：export和import。export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。

一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。下面是一个 JS 文件，里面使用export命令输出变量。

// profile.js

export var firstName = 'Michael';

export var lastName = 'Jackson';

export var year = 1958;

上面代码是profile.js文件，保存了用户信息。ES6将其视为一个模块，里面用export命令对外部输出了三个变量。

export的写法，除了像上面这样，还有另外一种。

// profile.js

var firstName = 'Michael';

var lastName = 'Jackson';

var year = 1958;

export {firstName, lastName, year};

export命令除了输出变量，还可以输出函数或类（class）。

export function multiply(x, y) {

return x \* y;

};

上面代码对外输出一个函数multiply。

通常情况下，export输出的变量就是本来的名字，但是可以使用as关键字重命名。

function v1() { ... }

function v2() { ... }

export {

v1 as streamV1,

v2 as streamV2,

v2 as streamLatestVersion

};

上面代码使用as关键字，重命名了函数v1和v2的对外接口。重命名后，v2可以用不同的名字输出两次。

需要特别注意的是，export命令规定的是对外的接口，必须与模块内部的变量建立一一对应关系。

// 报错

export 1;

// 报错

var m = 1;

export m;

上面两种写法都会报错，因为没有提供对外的接口。第一种写法直接输出1，第二种写法通过变量m，还是直接输出1。1只是一个值，不是接口。正确的写法是下面这样。

// 写法一

export var m = 1;

// 写法二

var m = 1;

export {m};

// 写法三

var n = 1;

export {n as m};

上面三种写法都是正确的，规定了对外的接口m。其他脚本可以通过这个接口，取到值1。它们的实质是，在接口名与模块内部变量之间，建立了一一对应的关系。

同样的，function和class的输出，也必须遵守这样的写法。

// 报错

function f() {}

export f;

// 正确

export function f() {};

// 正确

function f() {}

export {f};

另外，export语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

export var foo = 'bar';

setTimeout(() => foo = 'baz', 500);

上面代码输出变量foo，值为bar，500毫秒之后变成baz。

最后，export命令可以出现在模块的任何位置，只要处于模块顶层就可以。如果处于块级作用域内，就会报错，下一节的import命令也是如此。这是因为处于条件代码块之中，就没法做静态优化了，违背了ES6模块的设计初衷。

function foo() {

export default 'bar' // SyntaxError

}

foo()

上面代码中，export语句放在函数之中，结果报错。

### Import命令

使用export命令定义了模块的对外接口以后，其他 JS 文件就可以通过import命令加载这个模块。

// main.js

import {firstName, lastName, year} from './profile';

function setName(element) {

element.textContent = firstName + ' ' + lastName;

}

上面代码的import命令，用于加载profile.js文件，并从中输入变量。import命令接受一对大括号，里面指定要从其他模块导入的变量名。大括号里面的变量名，必须与被导入模块（profile.js）对外接口的名称相同。

如果想为输入的变量重新取一个名字，import命令要使用as关键字，将输入的变量重命名

import { lastName as surname } from './profile'

import后面的from指定模块文件的位置，可以是相对路径，也可以是绝对路径，.js路径可以省略。如果只是模块名，不带有路径，那么必须有配置文件，告诉 JavaScript 引擎该模块的位置

import {myMethod} from 'util';

上面代码中，util是模块文件名，由于不带有路径，必须通过配置，告诉引擎怎么取到这个模块。

注意，import命令具有提升效果，会提升到整个模块的头部，首先执行。

foo();

import { foo } from 'my\_module';

上面的代码不会报错，因为import的执行早于foo的调用。这种行为的本质是，import命令是编译阶段执行的，在代码运行之前。

由于import是静态执行，所以不能使用表达式和变量，这些只有在运行时才能得到结果的语法结构。

// 报错

import { 'f' + 'oo' } from 'my\_module';

// 报错

let module = 'my\_module';

import { foo } from module;

// 报错

if (x === 1) {

import { foo } from 'module1';

} else {

import { foo } from 'module2';

}

上面三种写法都会报错，因为它们用到了表达式、变量和if结构。在静态分析阶段，这些语法都是没法得到值的。

上面三种写法都会报错，因为它们用到了表达式、变量和if结构。在静态分析阶段，这些语法都是没法得到值的。

最后，import语句会执行所加载的模块，因此可以有下面的写法。

import 'lodash';

上面代码仅仅执行lodash模块，但是不输入任何值。

如果多次重复执行同一句import语句，那么只会执行一次，而不会执行多次。

import { foo } from 'my\_module';

import { bar } from 'my\_module';

// 等同于

import { foo, bar } from 'my\_module';

上面代码中，虽然foo和bar在两个语句中加载，但是它们对应的是同一个my\_module实例。也就是说，import语句是 Singleton 模式。

注意：有一个[提案](https://github.com/tc39/proposal-dynamic-import)，建议引入import()函数，完成动态加载。

import(specifier)

上面代码中，import函数的参数specifier，指定所要加载的模块的位置。import语句能够接受什么参数，import()函数就能接受什么参数，两者区别主要是后者为动态加载。

import()返回一个 Promise 对象。下面是一个例子。

const main = document.querySelector('main');

import(`./section-modules/${someVariable}.js`)

.then(module => {

module.loadPageInto(main);

})

.catch(err => {

main.textContent = err.message;

});

import()函数可以用在任何地方，不仅仅是模块，非模块的脚本也可以使用。它是运行时执行，也就是说，什么时候运行到这一句，也会加载指定的模块。另外，import()函数与所加载的模块没有静态连接关系，这点也是与import语句不相同。

import()类似于 Node 的require方法，区别主要是前者是异步加载，后者是同步加载。

### 模块的整体加载

除了指定加载某个输出值，还可以使用整体加载，即用星号（\*）指定一个对象，所有输出值都加载在这个对象上面。

下面是一个circle.js文件，它输出两个方法area和circumference。

// circle.js

export function area(radius) {

return Math.PI \* radius \* radius;

}

export function circumference(radius) {

return 2 \* Math.PI \* radius;

}

现在，加载这个模块。

// main.js

import { area, circumference } from './circle';

console.log('圆面积：' + area(4));

console.log('圆周长：' + circumference(14));

上面写法是逐一指定要加载的方法，整体加载的写法如下。

import \* as circle from './circle';

console.log('圆面积：' + circle.area(4));

console.log('圆周长：' + circle.circumference(14));

### export default命令

从前面的例子可以看出，使用import命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。但是，用户肯定希望快速上手，未必愿意阅读文档，去了解模块有哪些属性和方法。

为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。

// export-default.js

export default function () {

console.log('foo');

}

上面代码是一个模块文件export-default.js，它的默认输出是一个函数。

其他模块加载该模块时，import命令可以为该匿名函数指定任意名字。

// import-default.js

import customName from './export-default';

customName(); // 'foo'

上面代码的import命令，可以用任意名称指向export-default.js输出的方法，这时就不需要知道原模块输出的函数名。需要注意的是，这时import命令后面，不使用大括号。

export default命令用在非匿名函数前，也是可以的。

// export-default.js

export default function foo() {

console.log('foo');

}

// 或者写成

function foo() {

console.log('foo');

}

export default foo;

上面代码中，foo函数的函数名foo，在模块外部是无效的。加载的时候，视同匿名函数加载。

下面比较一下默认输出和正常输出。

/ 第一组

export default function crc32() { // 输出

// ...

}

import crc32 from 'crc32'; // 输入

// 第二组

export function crc32() { // 输出

// ...

};

import {crc32} from 'crc32'; // 输入

上面代码的两组写法，第一组是使用export default时，对应的import语句不需要使用大括号；第二组是不使用export default时，对应的import语句需要使用大括号。

export default命令用于指定模块的默认输出。显然，一个模块只能有一个默认输出，因此export default命令只能使用一次。所以，import命令后面才不用加大括号，因为只可能对应一个方法。

本质上，export default就是输出一个叫做default的变量或方法，然后系统允许你为它取任意名字。所以，下面的写法是有效的。

// modules.js

function add(x, y) {

return x \* y;

}

export {add as default};

// 等同于

// export default add;

// app.js

import { default as xxx } from 'modules';

// 等同于

// import xxx from 'modules';

正是因为export default命令其实只是输出一个叫做default的变量，所以它后面不能跟变量声明语句。

// 正确

export var a = 1;

// 正确

var a = 1;

export default a;

// 错误

export default var a = 1;

### export 与import 的复合写法

如果在一个模块之中，先输入后输出同一个模块，import语句可以与export语句写在一起。

export { foo, bar } from 'my\_module';

// 等同于

import { foo, bar } from 'my\_module';

export { foo, boo};

上面代码中，export和import语句可以结合在一起，写成一行。

模块的接口改名和整体输出，也可以采用这种写法。

// 接口改名

export { foo as myFoo } from 'my\_module';

// 整体输出

export \* from 'my\_module';

默认接口的写法如下

export { default } from 'foo';

具名接口改为默认接口的写法如下。

export { es6 as default } from './someModule';

// 等同于

import { es6 } from './someModule';

export default es6;

同样地，默认接口也可以改名为具名接口。

export { default as es6 } from './someModule';

另外，ES7有一个[提案](https://github.com/leebyron/ecmascript-more-export-from" \t "_blank)，简化先输入后输出的写法，拿掉输出时的大括号。

### 模板的继承

模块之间也可以继承。

假设有一个circleplus模块，继承了circle模块。

// circleplus.js

export \* from 'circle';

export var e = 2.71828182846;

export default function(x) {

return Math.exp(x);

}

上面代码中的export \*，表示再输出circle模块的所有属性和方法。注意，export \*命令会忽略circle模块的default方法。然后，上面代码又输出了自定义的e变量和默认方法。

这时，也可以将circle的属性或方法，改名后再输出。

// circleplus.js

export { area as circleArea } from 'circle';

上面代码表示，只输出circle模块的area方法，且将其改名为circleArea。

加载上面模块的写法如下。

// main.js

import \* as math from 'circleplus';

import exp from 'circleplus';

console.log(exp(math.e));

上面代码中的import exp表示，将circleplus模块的默认方法加载为exp方法。

### ES6模板加载的实质

ES6模块加载的机制，与CommonJS模块完全不同。CommonJS模块输出的是一个值的拷贝，而ES6模块输出的是值的引用。

CommonJS模块输出的是被输出值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。请看下面这个模块文件lib.js的例子。

// lib.js

var counter = 3;

function incCounter() {

counter++;

}

module.exports = {

counter: counter,

incCounter: incCounter,

};

上面代码输出内部变量counter和改写这个变量的内部方法incCounter。然后，在main.js里面加载这个模块。

// main.js

var mod = require('./lib');

console.log(mod.counter); // 3

mod.incCounter();

console.log(mod.counter); // 3

上面代码说明，lib.js模块加载以后，它的内部变化就影响不到输出的mod.counter了。这是因为mod.counter是一个原始类型的值，会被缓存。除非写成一个函数，才能得到内部变动后的值。

// lib.js

var counter = 3;

function incCounter() {

counter++;

}

module.exports = {

get counter() {

return counter

},

incCounter: incCounter,

};

上面代码中，输出的counter属性实际上是一个取值器函数。现在再执行main.js，就可以正确读取内部变量counter的变动了。

$ node main.js

3

4

ES6模块的运行机制与CommonJS不一样，它遇到模块加载命令import时，不会去执行模块，而是只生成一个动态的只读引用。等到真的需要用到时，再到模块里面去取值，换句话说，ES6的输入有点像Unix系统的“符号连接”，原始值变了，import输入的值也会跟着变。因此，ES6模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

还是举上面的例子。

// lib.js

export let counter = 3;

export function incCounter() {

counter++;

}

// main.js

import { counter, incCounter } from './lib';

console.log(counter); // 3

incCounter();

console.log(counter); // 4

上面代码说明，ES6模块输入的变量counter是活的，完全反应其所在模块lib.js内部的变化。

再举一个出现在export一节中的例子。

// m1.js

export var foo = 'bar';

setTimeout(() => foo = 'baz', 500);

// m2.js

import {foo} from './m1.js';

console.log(foo);

setTimeout(() => console.log(foo), 500);

上面代码中，m1.js的变量foo，在刚加载时等于bar，过了500毫秒，又变为等于baz。

让我们看看，m2.js能否正确读取这个变化。

$ babel-node m2.js

bar

baz

上面代码表明，ES6模块不会缓存运行结果，而是动态地去被加载的模块取值，并且变量总是绑定其所在的模块。

由于ES6输入的模块变量，只是一个“符号连接”，所以这个变量是只读的，对它进行重新赋值会报错。

// lib.js

export let obj = {};

// main.js

import { obj } from './lib';

obj.prop = 123; // OK

obj = {}; // TypeError

上面代码中，main.js从lib.js输入变量obj，可以对obj添加属性，但是重新赋值就会报错。因为变量obj指向的地址是只读的，不能重新赋值，这就好比main.js创造了一个名为obj的const变量。

最后，export通过接口，输出的是同一个值。不同的脚本加载这个接口，得到的都是同样的实例。

// mod.js

function C() {

this.sum = 0;

this.add = function () {

this.sum += 1;

};

this.show = function () {

console.log(this.sum);

};

}

export let c = new C();

上面的脚本mod.js，输出的是一个C的实例。不同的脚本加载这个模块，得到的都是同一个实例

// x.js

import {c} from './mod';

c.add();

// y.js

import {c} from './mod';

c.show();

// main.js

import './x';

import './y';

现在执行main.js，输出的是1。

$ babel-node main.js

1

### 浏览器的模块加载

浏览器使用 ES6 模块的语法如下。

<script type="module" src="foo.js"></script>

上面代码在网页中插入一个模块foo.js，由于type属性设为module，所以浏览器知道这是一个 ES6 模块。

浏览器对于带有type="module"的<script>，都是异步加载外部脚本，不会造成堵塞浏览器。

对于外部的模块脚本（上例是foo.js），有几点需要注意。

* 该脚本自动采用严格模块。
* 该脚本内部的顶层变量，都只在该脚本内部有效，外部不可见。
* 该脚本内部的顶层的this关键字，返回undefined，而不是指向window。

### 循环加载

通常，“循环加载”表示存在强耦合，如果处理不好，还可能导致递归加载，使得程序无法执行，因此应该避免出现。

但是实际上，这是很难避免的，尤其是依赖关系复杂的大项目，很容易出现a依赖b，b依赖c，c又依赖a这样的情况。这意味着，模块加载机制必须考虑“循环加载”的情况。

对于JavaScript语言来说，目前最常见的两种模块格式CommonJS和ES6，处理“循环加载”的方法是不一样的，返回的结果也不一样。

### 跨模块常量

本书介绍const命令的时候说过，const声明的常量只在当前代码块有效。如果想设置跨模块的常量（即跨多个文件），可以采用下面的写法。

// constants.js 模块

export const A = 1;

export const B = 3;

export const C = 4;

// test1.js 模块

import \* as constants from './constants';

console.log(constants.A); // 1

console.log(constants.B); // 3

// test2.js 模块

import {A, B} from './constants';

console.log(A); // 1

console.log(B); // 3

如果要使用的常量非常多，可以建一个专门的constants目录，将各种常量写在不同的文件里面，保存在该目录下。

// constants/db.js

export const db = {

url: 'http://my.couchdbserver.local:5984',

admin\_username: 'admin',

admin\_password: 'admin password'

};

// constants/user.js

export const users = ['root', 'admin', 'staff', 'ceo', 'chief', 'moderator'];

然后，将这些文件输出的常量，合并在index.js里面。

// constants/index.js

export {db} from './db';

export {users} from './users';

使用的时候，直接加载index.js就可以了。

// script.js

import {db, users} from './constants';

### import()

上面说过了，import语句会被JavaScript引擎静态分析，先于模块内的其他模块执行（叫做”连接“更合适）。所以，下面的代码会报错。

// 报错

if (x === 2) {

import MyModual from './myModual';

}

上面代码中，引擎处理import语句是在执行之前，所以import语句放在if代码块之中毫无意义，因此会报句法错误，而不是执行时错误。

这样的设计，固然有利于编译器提高效率，但也导致无法在运行时加载模块。从长远来看，import语句会取代 Node 的require方法，但是require是运行时加载模块，import语句显然无法取代这种动态加载功能。

const path = './' + fileName;

const myModual = require(path);

上面的语句就是动态加载，require到底加载哪一个模块，只有运行时才知道。import语句做不到这一点。

因此，有一个[提案](https://github.com/tc39/proposal-dynamic-import)，建议引入import()函数，完成动态加载。

import(specifier)

上面代码中，import函数的参数specifier，指定所要加载的模块的位置。import语句能够接受什么参数，import()函数就能接受什么参数，两者区别主要是后者为动态加载。

import()返回一个 Promise 对象。下面是一个例子。

const main = document.querySelector('main');

import(`./section-modules/${someVariable}.js`)

.then(module => {

module.loadPageInto(main);

})

.catch(err => {

main.textContent = err.message;

});

import()函数可以用在任何地方，不仅仅是模块，非模块的脚本也可以使用。它是运行时执行，也就是说，什么时候运行到这一句，也会加载指定的模块。另外，import()函数与所加载的模块没有静态连接关系，这点也是与import语句不相同。

import()类似于 Node 的require方法，区别主要是前者是异步加载，后者是同步加载。

### ES6模块的转码

浏览器目前还不支持ES6模块，为了现在就能使用，可以将转为ES5的写法。除了Babel可以用来转码之外，还有以下两个方法，也可以用来转码。

#### ES module transpiler

[ES6 module transpiler](https://github.com/esnext/es6-module-transpiler)是 square 公司开源的一个转码器，可以将 ES6 模块转为 CommonJS 模块或 AMD 模块的写法，从而在浏览器中使用。

首先，安装这个转码器

$ npm install -g es6-module-transpiler

然后，使用compile-modules convert命令，将 ES6 模块文件转码。

$ compile-modules convert file1.js file2.js

-o参数可以指定转码后的文件名。

$ compile-modules convert -o out.js file1.js

拓展：

安装babel

// es2015转码规则

**npm install --save-dev babel-preset-es2015**

// react转码规则

**npm install --save-dev babel-preset-react**

// ES7不同阶段语法提案的转码规则（共有4个阶段），选装一个

**npm install --save-dev babel-preset-stage-0**

**npm install --save-dev babel-preset-stage-1**

**npm install --save-dev babel-preset-stage-2**

**npm install --save-dev babel-preset-stage-3**

#### system JS

另一种解决方法是使用 [SystemJS](https://github.com/systemjs/systemjs" \t "_blank)。它是一个垫片库（polyfill），可以在浏览器内加载 ES6 模块、AMD 模块和 CommonJS 模块，将其转为 ES5 格式。它在后台调用的是 Google 的 Traceur 转码器。

使用时，先在网页内载入system.js文件。

<script src="system.js"></script>

然后，使用System.import方法加载模块文件。

<script>

System.import('./app.js');

</script>

上面代码中的./app，指的是当前目录下的app.js文件。它可以是ES6模块文件，System.import会自动将其转码。

需要注意的是，System.import使用异步加载，返回一个 Promise 对象，可以针对这个对象编程。下面是一个模块文件。

// app/es6-file.js:

export class q {

constructor() {

this.es6 = 'hello';

}

}

然后，在网页内加载这个模块文件。

<script>

System.import('app/es6-file').then(function(m) {

console.log(new m.q().es6); // hello

});

</script>

上面代码中，System.import方法返回的是一个 Promise 对象，所以可以用then方法指定回调函数。

## 变量的解构赋值.

### 基本用法

ES6允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

以前，为变量赋值，只能直接指定值。

var a = 1;

var b = 2;

var c = 3;

ES6允许写成下面这样。

var [a, b, c] = [1, 2, 3];

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。下面是一些使用嵌套数组进行解构的例子。

let [foo, [[bar], baz]] = [1, [[2], 3]];

foo // 1

bar // 2

baz // 3

let [ , , third] = ["foo", "bar", "baz"];

third // "baz"

let [x, , y] = [1, 2, 3];

x // 1

y // 3

let [head, ...tail] = [1, 2, 3, 4];

head // 1

tail // [2, 3, 4]

let [x, y, ...z] = ['a'];

x // "a"

y // undefined

z // []

如果解构不成功，变量的值就等于undefined。

var [foo] = [];

var [bar, foo] = [1];

以上两种情况都属于解构不成功，foo的值都会等于undefined。

另一种情况是不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。这种情况下，解构依然可以成功。

let [x, y] = [1, 2, 3];

x // 1

y // 2

let [a, [b], d] = [1, [2, 3], 4];

a // 1

b // 2

d // 4

上面两个例子，都属于不完全解构，但是可以成功。

如果等号的右边不是数组（或者严格地说，不是可遍历的结构，参见《Iterator》一章），那么将会报错。

// 报错

let [foo] = 1;

let [foo] = false;

let [foo] = NaN;

let [foo] = undefined;

let [foo] = null;

let [foo] = {};

上面的表达式都会报错，因为等号右边的值，要么转为对象以后不具备Iterator接口（前五个表达式），要么本身就不具备Iterator接口（最后一个表达式）。

Iterator 简介：<https://blog.csdn.net/hsl0530hsl/article/details/78263097>

### 默认值

解构赋值允许指定默认值。

var [foo = true] = [];

foo // true

[x, y = 'b'] = ['a']; // x='a', y='b'

[x, y = 'b'] = ['a', undefined]; // x='a', y='b'

注意，ES6内部使用严格相等运算符（===），判断一个位置是否有值。所以，如果一个数组成员不严格等于undefined，默认值是不会生效的。

var [x = 1] = [undefined];

x // 1

var [x = 1] = [null];

x // null

上面代码中，如果一个数组成员是null，默认值就不会生效，因为null不严格等于undefined。

如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

function f() {

console.log('aaa');

}

let [x = f()] = [1];

上面代码中，因为x能取到值，所以函数f根本不会执行。上面的代码其实等价于下面的代码。

let x;

if ([1][0] === undefined) {

x = f();

} else {

x = [1][0];

}

默认值可以引用解构赋值的其他变量，但该变量必须已经声明

let [x = 1, y = x] = []; // x=1; y=1

let [x = 1, y = x] = [2]; // x=2; y=2

let [x = 1, y = x] = [1, 2]; // x=1; y=2

let [x = y, y = 1] = []; // ReferenceError

上面最后一个表达式之所以会报错，是因为x用到默认值y时，y还没有声明。

### 对象的解构赋值

解构不仅可以用于数组，还可以用于对象。

var { foo, bar } = { foo: "aaa", bar: "bbb" };

foo // "aaa"

bar // "bbb"

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

var { bar, foo } = { foo: "aaa", bar: "bbb" };

foo // "aaa"

bar // "bbb"

var { baz } = { foo: "aaa", bar: "bbb" };

baz // undefined

上面代码的第一个例子，等号左边的两个变量的次序，与等号右边两个同名属性的次序不一致，但是对取值完全没有影响。第二个例子的变量没有对应的同名属性，导致取不到值，最后等于undefined。

如果变量名与属性名不一致，必须写成下面这样。

var { foo: baz } = { foo: 'aaa', bar: 'bbb' };

baz // "aaa"

let obj = { first: 'hello', last: 'world' };

let { first: f, last: l } = obj;

f // 'hello'

l // 'world'

这实际上说明，对象的解构赋值是下面形式的简写（参见《对象的扩展》一章）。

var { foo: foo, bar: bar } = { foo: "aaa", bar: "bbb" };

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

var { foo: baz } = { foo: "aaa", bar: "bbb" };

baz // "aaa"

foo // error: foo is not defined

上面代码中，foo是匹配的模式，baz才是变量。真正被赋值的是变量baz，而不是模式foo。

注意，采用这种写法时，变量的声明和赋值是一体的。对于let和const来说，变量不能重新声明，所以一旦赋值的变量以前声明过，就会报错。

let foo;

let {foo} = {foo: 1}; // SyntaxError: Duplicate declaration "foo"

let baz;

let {bar: baz} = {bar: 1}; // SyntaxError: Duplicate declaration "baz"

上面代码中，解构赋值的变量都会重新声明，所以报错了。不过，因为var命令允许重新声明，所以这个错误只会在使用let和const命令时出现。如果没有第二个let命令，上面的代码就不会报错。

let foo;

({foo} = {foo: 1}); // 成功

let baz;

({bar: baz} = {bar: 1}); // 成功

上面代码中，let命令下面一行的圆括号是必须的，否则会报错。因为解析器会将起首的大括号，理解成一个代码块，而不是赋值语句。

和数组一样，解构也可以用于嵌套结构的对象。

var obj = {

p: [

'Hello',

{ y: 'World' }

]

};

var { p: [x, { y }] } = obj;

x // "Hello"

y // "World"

注意，这时p是模式，不是变量，因此不会被赋值。

var node = {

loc: {

start: {

line: 1,

column: 5

}

}

};

var { loc: { start: { line }} } = node;

line // 1

loc // error: loc is undefined

start // error: start is undefined

上面代码中，只有line是变量，loc和start都是模式，不会被赋值。

下面是嵌套赋值的例子。

let obj = {};

let arr = [];

({ foo: obj.prop, bar: arr[0] } = { foo: 123, bar: true });

obj // {prop:123}

arr // [true]

var {x = 3} = {};

x // 3

var {x, y = 5} = {x: 1};

x // 1

y // 5

var {x:y = 3} = {};

y // 3

var {x:y = 3} = {x: 5};

y // 5

var { message: msg = 'Something went wrong' } = {};

msg // "Something went wrong"

默认值生效的条件是，对象的属性值严格等于undefined。

var {x = 3} = {x: undefined};

x // 3

var {x = 3} = {x: null};

x // null

上面代码中，如果x属性等于null，就不严格相等于undefined，导致默认值不会生效。

如果解构失败，变量的值等于undefined。

var {foo} = {bar: 'baz'};

foo // undefined

如果解构模式是嵌套的对象，而且子对象所在的父属性不存在，那么将会报错。

// 报错

var {foo: {bar}} = {baz: 'baz'};

上面代码中，等号左边对象的foo属性，对应一个子对象。该子对象的bar属性，解构时会报错。原因很简单，因为foo这时等于undefined，再取子属性就会报错，请看下面的代码。

var \_tmp = {baz: 'baz'};

\_tmp.foo.bar // 报错

如果要将一个已经声明的变量用于解构赋值，必须非常小心。

// 错误的写法

var x;

{x} = {x: 1};

// SyntaxError: syntax error

上面代码的写法会报错，因为JavaScript引擎会将{x}理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免JavaScript将其解释为代码块，才能解决这个问题。

// 正确的写法

({x} = {x: 1});

上面代码将整个解构赋值语句，放在一个圆括号里面，就可以正确执行。关于圆括号与解构赋值的关系，参见下文。

解构赋值允许，等号左边的模式之中，不放置任何变量名。因此，可以写出非常古怪的赋值表达式。

({} = [true, false]);

({} = 'abc');

({} = []);

上面的表达式虽然毫无意义，但是语法是合法的，可以执行。

对象的解构赋值，可以很方便地将现有对象的方法，赋值到某个变量。

let { log, sin, cos } = Math;

## ES6 正则的扩展

### RegExp构造函数

在ES5中，RegExp构造函数的参数有两种情况。

第一种情况是，参数是字符串，这时第二个参数表示正则表达式的修饰符（flag）。

var regex = new RegExp('xyz', 'i');

// 等价于

var regex = /xyz/i;

第二种情况是，参数是一个正则表示式，这时会返回一个原有正则表达式的拷贝。

var regex = new RegExp(/xyz/i);

// 等价于

var regex = /xyz/i;

但是，ES5不允许此时使用第二个参数，添加修饰符，否则会报错。

var regex = new RegExp(/xyz/, 'i');

// Uncaught TypeError: Cannot supply flags when constructing one RegExp from another

ES6改变了这种行为。如果RegExp构造函数第一个参数是一个正则对象，那么可以使用第二个参数指定修饰符。而且，返回的正则表达式会忽略原有的正则表达式的修饰符，只使用新指定的修饰符。

new RegExp(/abc/ig, 'i').flags

// "i"

上面代码中，原有正则对象的修饰符是ig，它会被第二个参数i覆盖。

## 函数扩展

### 函数参数的默认值

在ES6之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

function log(x, y) {

y = y || 'World';

console.log(x, y);

}

log('Hello') // Hello World

log('Hello', 'China') // Hello China

log('Hello', '') // Hello World

上面代码检查函数log的参数y有没有赋值，如果没有，则指定默认值为World。这种写法的缺点在于，如果参数y赋值了，但是对应的布尔值为false，则该赋值不起作用。就像上面代码的最后一行，参数y等于空字符，结果被改为默认值。

为了避免这个问题，通常需要先判断一下参数y是否被赋值，如果没有，再等于默认值。

if (typeof y === 'undefined') {

y = 'World';

}

ES6允许为函数的参数设置默认值，即直接写在参数定义的后面。

function log(x, y = 'World') {

console.log(x, y);

}

log('Hello') // Hello World

log('Hello', 'China') // Hello China

log('Hello', '') // Hello

可以看到，ES6的写法比ES5简洁许多，而且非常自然。下面是另一个例子。

function Point(x = 0, y = 0) {

this.x = x;

this.y = y;

}

var p = new Point();

p // { x: 0, y: 0 }

除了简洁，ES6的写法还有两个好处：首先，阅读代码的人，可以立刻意识到哪些参数是可以省略的，不用查看函数体或文档；其次，有利于将来的代码优化，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行。

参数变量是默认声明的，所以不能用let或const再次声明。

function foo(x = 5) {

let x = 1; // error

const x = 2; // error

}

### 与解构赋值默认值结合使用

参数默认值可以与解构赋值的默认值，结合起来使用。

function foo({x, y = 5}) {

console.log(x, y);

}

foo({}) // undefined, 5

foo({x: 1}) // 1, 5

foo({x: 1, y: 2}) // 1, 2

foo() // TypeError: Cannot read property 'x' of undefined

上面代码使用了对象的解构赋值默认值，而没有使用函数参数的默认值。只有当函数foo的参数是一个对象时，变量x和y才会通过解构赋值而生成。如果函数foo调用时参数不是对象，变量x和y就不会生成，从而报错。如果参数对象没有y属性，y的默认值5才会生效。

下面是另一个对象的解构赋值默认值的例子。

function fetch(url, { body = '', method = 'GET', headers = {} }) {

console.log(method);

}

fetch('http://example.com', {})

// "GET"

fetch('http://example.com')

// 报错

上面代码中，如果函数fetch的第二个参数是一个对象，就可以为它的三个属性设置默认值。

上面的写法不能省略第二个参数，如果结合函数参数的默认值，就可以省略第二个参数。这时，就出现了双重默认值。

function fetch(url, { method = 'GET' } = {}) {

console.log(method);

}

fetch('http://example.com')

// "GET"

上面代码中，函数fetch没有第二个参数时，函数参数的默认值就会生效，然后才是解构赋值的默认值生效，变量method才会取到默认值GET。

再请问下面两种写法有什么差别？

// 写法一

function m1({x = 0, y = 0} = {}) {

return [x, y];

}

// 写法二

function m2({x, y} = { x: 0, y: 0 }) {

return [x, y];

}

上面两种写法都对函数的参数设定了默认值，区别是写法一函数参数的默认值是空对象，但是设置了对象解构赋值的默认值；写法二函数参数的默认值是一个有具体属性的对象，但是没有设置对象解构赋值的默认值。

### 参数默认值的位置

通常情况下，定义了默认值的参数，应该是函数的尾参数。因为这样比较容易看出来，到底省略了哪些参数。如果非尾部的参数设置默认值，实际上这个参数是没法省略的。

// 例一

function f(x = 1, y) {

return [x, y];

}

f() // [1, undefined]

f(2) // [2, undefined])

f(, 1) // 报错

f(undefined, 1) // [1, 1]

// 例二

function f(x, y = 5, z) {

return [x, y, z];

}

f() // [undefined, 5, undefined]

f(1) // [1, 5, undefined]

f(1, ,2) // 报错

f(1, undefined, 2) // [1, 5, 2]

上面代码中，有默认值的参数都不是尾参数。这时，无法只省略该参数，而不省略它后面的参数，除非显式输入undefined。

如果传入undefined，将触发该参数等于默认值，null则没有这个效果。

function foo(x = 5, y = 6) {

console.log(x, y);

}

foo(undefined, null)

// 5 null

上面代码中，x参数对应undefined，结果触发了默认值，y参数等于null，就没有触发默认值。

### 函数的length属性

指定了默认值以后，函数的length属性，将返回没有指定默认值的参数个数。也就是说，指定了默认值后，length属性将失真。

(function (a) {}).length // 1

(function (a = 5) {}).length // 0

(function (a, b, c = 5) {}).length // 2

上面代码中，length属性的返回值，等于函数的参数个数减去指定了默认值的参数个数。比如，上面最后一个函数，定义了3个参数，其中有一个参数c指定了默认值，因此length属性等于3减去1，最后得到2。

这是因为length属性的含义是，该函数预期传入的参数个数。某个参数指定默认值以后，预期传入的参数个数就不包括这个参数了。同理，rest参数也不会计入length属性。

如果设置了默认值的参数不是尾参数，那么length属性也不再计入后面的参数了。

(function (a = 0, b, c) {}).length // 0

(function (a, b = 1, c) {}).length // 1

### 作用域

一个需要注意的地方是，如果参数默认值是一个变量，则该变量所处的作用域，与其他变量的作用域规则是一样的，即先是当前函数的作用域，然后才是全局作用域。

var x = 1;

function f(x, y = x) {

console.log(y);

}

f(2) // 2

上面代码中，参数y的默认值等于x。调用时，由于函数作用域内部的变量x已经生成，所以y等于参数x，而不是全局变量x。

如果调用时，函数作用域内部的变量x没有生成，结果就会不一样。

let x = 1;

function f(y = x) {

let x = 2;

console.log(y);

}

f() // 1

上面代码中，函数调用时，y的默认值变量x尚未在函数内部生成，所以x指向全局变量。

如果此时，全局变量x不存在，就会报错。

var x = 1;

function foo(x = x) {

// ...

}

foo() // ReferenceError: x is not defined

上面代码中，函数foo的参数x的默认值也是x。这时，默认值x的作用域是函数作用域，而不是全局作用域。由于在函数作用域中，存在变量x，但是默认值在x赋值之前先执行了，所以这时属于暂时性死区（参见《let和const命令》一章），任何对x的操作都会报错。

如果参数的默认值是一个函数，该函数的作用域是其声明时所在的作用域。请看下面的例子。

let foo = 'outer';

function bar(func = x => foo) {

let foo = 'inner';

console.log(func()); // outer

}

bar();

上面代码中，函数bar的参数func的默认值是一个匿名函数，返回值为变量foo。这个匿名函数声明时，bar函数的作用域还没有形成，所以匿名函数里面的foo指向外层作用域的foo，输出outer。

如果写成下面这样，就会报错。

function bar(func = () => foo) {

let foo = 'inner';

console.log(func());

}

bar() // ReferenceError: foo is not defined

上面代码中，匿名函数里面的foo指向函数外层，但是函数外层并没有声明foo，所以就报错了。

下面是一个更复杂的例子。

var x = 1;

function foo(x, y = function() { x = 2; }) {

var x = 3;

y();

console.log(x);

}

foo() // 3

上面代码中，函数foo的参数y的默认值是一个匿名函数。函数foo调用时，它的参数x的值为undefined，所以y函数内部的x一开始是undefined，后来被重新赋值2。但是，函数foo内部重新声明了一个x，值为3，这两个x是不一样的，互相不产生影响，因此最后输出3。

如果将var x = 3的var去除，两个x就是一样的，最后输出的就是2。

var x = 1;

function foo(x, y = function() { x = 2; }) {

x = 3;

y();

console.log(x);

}

foo() // 2

## ES6对象的扩展

#### 属性的简洁表示法

ES6允许直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

var foo = 'bar';

var baz = {foo};

baz // {foo: "bar"}

// 等同于

var baz = {foo: foo};

上面代码表明，ES6允许在对象之中，直接写变量。这时，属性名为变量名, 属性值为变量的值。下面是另一个例子。

function f(x, y) {

return {x, y};

}

// 等同于

function f(x, y) {

return {x: x, y: y};

}

f(1, 2) // Object {x: 1, y: 2}

除了属性简写，方法也可以简写。

var o = {

method() {

return "Hello!";

}

};

// 等同于

var o = {

method: function() {

return "Hello!";

}

};

下面是一个实际的例子。

var birth = '2000/01/01';

var Person = {

name: '张三',

//等同于birth: birth

birth,

// 等同于hello: function ()...

hello() { console.log('我的名字是', this.name); }

};

这种写法用于函数的返回值，将会非常方便。

function getPoint() {

var x = 1;

var y = 10;

return {x, y};

}

getPoint()

// {x:1, y:10}

CommonJS模块输出变量，就非常合适使用简洁写法。

var ms = {};

function getItem (key) {

return key in ms ? ms[key] : null;

}

function setItem (key, value) {

ms[key] = value;

}

function clear () {

ms = {};

}

module.exports = { getItem, setItem, clear };

// 等同于

module.exports = {

getItem: getItem,

setItem: setItem,

clear: clear

};

属性的赋值器（setter）和取值器（getter），事实上也是采用这种写法。

var cart = {

\_wheels: 4,

get wheels () {

return this.\_wheels;

},

set wheels (value) {

if (value < this.\_wheels) {

throw new Error('数值太小了！');

}

this.\_wheels = value;

}

}

注意，简洁写法的属性名总是字符串，这会导致一些看上去比较奇怪的结果。

var obj = {

class () {}

};

// 等同于

var obj = {

'class': function() {}

};

上面代码中，class是字符串，所以不会因为它属于关键字，而导致语法解析报错。

如果某个方法的值是一个Generator函数，前面需要加上星号。

var obj = {

\* m(){

yield 'hello world';

}

};

### 属性名表达式

JavaScript语言定义对象的属性，有两种方法。

// 方法一

obj.foo = true;

// 方法二

obj['a' + 'bc'] = 123;

上面代码的方法一是直接用标识符作为属性名，方法二是用表达式作为属性名，这时要将表达式放在方括号之内。

但是，如果使用字面量方式定义对象（使用大括号），在 ES5 中只能使用方法一（标识符）定义属性。

var obj = {

foo: true,

abc: 123

};

ES6 允许字面量定义对象时，用方法二（表达式）作为对象的属性名，即把表达式放在方括号内。

let propKey = 'foo';

let obj = {

[propKey]: true,

['a' + 'bc']: 123

};

下面是另一个例子。

var lastWord = 'last word';

var a = {

'first word': 'hello',

[lastWord]: 'world'

};

a['first word'] // "hello"

a[lastWord] // "world"

a['last word'] // "world"

表达式还可以用于定义方法名。

let obj = {

['h' + 'ello']() {

return 'hi';

}

};

obj.hello() // hi

注意，属性名表达式与简洁表示法，不能同时使用，会报错。

// 报错

var foo = 'bar';

var bar = 'abc';

var baz = { [foo] };

// 正确

var foo = 'bar';

var baz = { [foo]: 'abc'};

注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串[object Object]，这一点要特别小心。

const keyA = {a: 1};

const keyB = {b: 2};

const myObject = {

[keyA]: 'valueA',

[keyB]: 'valueB'

};

myObject // Object {[object Object]: "valueB"}

上面代码中，[keyA]和[keyB]得到的都是[object Object]，所以[keyB]会把[keyA]覆盖掉，而myObject最后只有一个[object Object]属性

### 方法的name属性

var person = {

sayName() {

console.log(this.name);

},

get firstName() {

return "Nicholas";

}

};

person.sayName.name // "sayName"

person.firstName.name // "get firstName"

上面代码中，方法的name属性返回函数名（即方法名）。如果使用了取值函数，则会在方法名前加上get。如果是存值函数，方法名的前面会加上set。

有两种特殊情况：bind方法创造的函数，name属性返回“bound”加上原函数的名字；Function构造函数创造的函数，name属性返回“anonymous”。

(new Function()).name // "anonymous"

var doSomething = function() {

// ...

};

doSomething.bind().name // "bound doSomething"

如果对象的方法是一个Symbol值，那么name属性返回的是这个Symbol值的描述。

const key1 = Symbol('description');

const key2 = Symbol();

let obj = {

[key1]() {},

[key2]() {},

};

obj[key1].name // "[description]"

obj[key2].name // ""

上面代码中，key1对应的Symbol值有描述，key2没有。

### Object.is()

ES5比较两个值是否相等，只有两个运算符：相等运算符（==）和严格相等运算符（===）。它们都有缺点，前者会自动转换数据类型，后者的NaN不等于自身，以及+0等于-0。JavaScript缺乏一种运算，在所有环境中，只要两个值是一样的，它们就应该相等。

ES6提出“Same-value equality”（同值相等）算法，用来解决这个问题。Object.is就是部署这个算法的新方法。它用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致。

Object.is('foo', 'foo')

// true

Object.is({}, {})

// false

不同之处只有两个：一是+0不等于-0，二是NaN等于自身。

+0 === -0 //true

NaN === NaN // false

Object.is(+0, -0) // false

Object.is(NaN, NaN) // true

ES5可以通过下面的代码，部署Object.is。

Object.defineProperty(Object, 'is', {

value: function(x, y) {

if (x === y) {

// 针对+0 不等于 -0的情况

return x !== 0 || 1 / x === 1 / y;

}

// 针对NaN的情况

return x !== x && y !== y;

},

configurable: true,

enumerable: false,

writable: true

});

注意：Object.assign方法实行的是浅拷贝，而不是深拷贝。也就是说，如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。

## ES6 symbol

### 概要

ES5的对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（mixin模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是ES6引入Symbol的原因。

ES6引入了一种新的原始数据类型Symbol，表示独一无二的值。它是JavaScript语言的第七种数据类型，前六种是：Undefined、Null、布尔值（Boolean）、字符串（String）、数值（Number）、对象（Object）。

Symbol值通过Symbol函数生成。这就是说，对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的Symbol类型。凡是属性名属于Symbol类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

let s = Symbol();

typeof s

// "symbol"

上面代码中，变量s就是一个独一无二的值。typeof运算符的结果，表明变量s是Symbol数据类型，而不是字符串之类的其他类型。

注意，Symbol函数前不能使用new命令，否则会报错。这是因为生成的Symbol是一个原始类型的值，不是对象。也就是说，由于Symbol值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。

Symbol函数可以接受一个字符串作为参数，表示对Symbol实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。

var s1 = Symbol('foo');

var s2 = Symbol('bar');

s1 // Symbol(foo)

s2 // Symbol(bar)

s1.toString() // "Symbol(foo)"

s2.toString() // "Symbol(bar)"

上面代码中，s1和s2是两个Symbol值。如果不加参数，它们在控制台的输出都是Symbol()，不利于区分。有了参数以后，就等于为它们加上了描述，输出的时候就能够分清，到底是哪一个值。

如果 Symbol 的参数是一个对象，就会调用该对象的toString方法，将其转为字符串，然后才生成一个 Symbol 值。

const obj = {

toString() {

return 'abc';

}

};

const sym = Symbol(obj);

sym // Symbol(abc)

注意，Symbol函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的Symbol函数的返回值是不相等的。

// 没有参数的情况

var s1 = Symbol();

var s2 = Symbol();

s1 === s2 // false

// 有参数的情况

var s1 = Symbol('foo');

var s2 = Symbol('foo');

s1 === s2 // false

上面代码中，s1和s2都是Symbol函数的返回值，而且参数相同，但是它们是不相等的。

Symbol值不能与其他类型的值进行运算，会报错。

var sym = Symbol('My symbol');

"your symbol is " + sym

// TypeError: can't convert symbol to string

`your symbol is ${sym}`

// TypeError: can't convert symbol to string

但是，Symbol值可以显式转为字符串。

var sym = Symbol('My symbol');

String(sym) // 'Symbol(My symbol)'

sym.toString() // 'Symbol(My symbol)'

另外，Symbol值也可以转为布尔值，但是不能转为数值。

var sym = Symbol();

Boolean(sym) // true

!sym // false

if (sym) {

// ...

}

Number(sym) // TypeError

sym + 2 // TypeError

### 作为属性名的Symbol

由于每一个Symbol值都是不相等的，这意味着Symbol值可以作为标识符，用于对象的属性名，就能保证不会出现同名的属性。这对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖。

var mySymbol = Symbol();

// 第一种写法

var a = {};

a[mySymbol] = 'Hello!';

// 第二种写法

var a = {

[mySymbol]: 'Hello!'

};

// 第三种写法

var a = {};

Object.defineProperty(a, mySymbol, { value: 'Hello!' });

// 以上写法都得到同样结果

a[mySymbol] // "Hello!"

上面代码通过方括号结构和Object.defineProperty，将对象的属性名指定为一个Symbol值。

注意，Symbol值作为对象属性名时，不能用点运算符。

var mySymbol = Symbol();

var a = {};

a.mySymbol = 'Hello!';

a[mySymbol] // undefined

a['mySymbol'] // "Hello!"

上面代码中，因为点运算符后面总是字符串，所以不会读取mySymbol作为标识名所指代的那个值，导致a的属性名实际上是一个字符串，而不是一个Symbol值。

同理，在对象的内部，使用Symbol值定义属性时，Symbol值必须放在方括号之中。

let s = Symbol();

let obj = {

[s]: function (arg) { ... }

};

obj[s](123);

上面代码中，如果s不放在方括号中，该属性的键名就是字符串s，而不是s所代表的那个Symbol值。

采用增强的对象写法，上面代码的obj对象可以写得更简洁一些。

let obj = {

[s](arg) { ... }

};

Symbol类型还可以用于定义一组常量，保证这组常量的值都是不相等的。

log.levels = {

DEBUG: Symbol('debug'),

INFO: Symbol('info'),

WARN: Symbol('warn')

};

log(log.levels.DEBUG, 'debug message');

log(log.levels.INFO, 'info message');

下面是另外一个例子

const COLOR\_RED = Symbol();

const COLOR\_GREEN = Symbol();

function getComplement(color) {

switch (color) {

case COLOR\_RED:

return COLOR\_GREEN;

case COLOR\_GREEN:

return COLOR\_RED;

default:

throw new Error('Undefined color');

}

}

常量使用Symbol值最大的好处，就是其他任何值都不可能有相同的值了，因此可以保证上面的switch语句会按设计的方式工作。

还有一点需要注意，Symbol值作为属性名时，该属性还是公开属性，不是私有属性。

## ES6 set和map数据结构

### Set

#### 基础用法

ES6提供了新的数据结构Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。

Set本身是一个构造函数，用来生成Set数据结构。

var s = new Set();

[2, 3, 5, 4, 5, 2, 2].map(x => s.add(x));

for (let i of s) {

console.log(i);

}

// 2 3 5 4

上面代码通过add方法向Set结构加入成员，结果表明Set结构不会添加重复的值。

Set函数可以接受一个数组（或类似数组的对象）作为参数，用来初始化。

// 例一

var set = new Set([1, 2, 3, 4, 4]);

[...set]

// [1, 2, 3, 4]

// 例二

var items = new Set([1, 2, 3, 4, 5, 5, 5, 5]);

items.size // 5

// 例三

function divs () {

return [...document.querySelectorAll('div')];

}

var set = new Set(divs());

set.size // 56

// 类似于

divs().forEach(div => set.add(div));

set.size // 56

上面代码中，例一和例二都是Set函数接受数组作为参数，例三是接受类似数组的对象作为参数。

上面代码中，也展示了一种去除数组重复成员的方法。

// 去除数组的重复成员

[...new Set(array)]

向Set加入值的时候，不会发生类型转换，所以5和"5"是两个不同的值。Set内部判断两个值是否不同，使用的算法叫做“Same-value equality”，它类似于精确相等运算符（===），主要的区别是NaN等于自身，而精确相等运算符认为NaN不等于自身。

let set = new Set();

let a = NaN;

let b = NaN;

set.add(a);

set.add(b);

set // Set {NaN}

上面代码向Set实例添加了两个NaN，但是只能加入一个。这表明，在Set内部，两个NaN是相等。

另外，两个对象总是不相等的。

let set = new Set();

set.add({});

set.size // 1

set.add({});

set.size // 2

上面代码表示，由于两个空对象不相等，所以它们被视为两个值。

#### Set实例的属性和方法

Set结构的实例有以下属性。

* Set.prototype.constructor：构造函数，默认就是Set函数。
* Set.prototype.size：返回Set实例的成员总数。

Set实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。

* add(value)：添加某个值，返回Set结构本身。
* delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。
* has(value)：返回一个布尔值，表示该值是否为Set的成员。
* clear()：清除所有成员，没有返回值。

上面这些属性和方法的实例如下。

s.add(1).add(2).add(2);

// 注意2被加入了两次

s.size // 2

s.has(1) // true

s.has(2) // true

s.has(3) // false

s.delete(2);

s.has(2) // false

下面是一个对比，看看在判断是否包括一个键上面，Object结构和Set结构的写法不同。

// 对象的写法

var properties = {

'width': 1,

'height': 1

};

if (properties[someName]) {

// do something

}

// Set的写法

var properties = new Set();

properties.add('width');

properties.add('height');

if (properties.has(someName)) {

// do something

}

Array.from方法可以将Set结构转为数组。

var items = new Set([1, 2, 3, 4, 5]);

var array = Array.from(items);

这就提供了去除数组重复成员的另一种方法。

function dedupe(array) {

return Array.from(new Set(array));

}

dedupe([1, 1, 2, 3]) // [1, 2, 3]

和扩展运算符结合的去重

arr = [...new Set(arr)]

#### 遍历操作

Set结构的实例有四个遍历方法，可以用于遍历成员。

* keys()：返回键名的遍历器
* values()：返回键值的遍历器
* entries()：返回键值对的遍历器
* forEach()：使用回调函数遍历每个成员

需要特别指出的是，Set的遍历顺序就是插入顺序。这个特性有时非常有用，比如使用Set保存一个回调函数列表，调用时就能保证按照添加顺序调用。

##### keys()，values()，entries()

key方法、value方法、entries方法返回的都是遍历器对象（详见《Iterator对象》一章）。由于Set结构没有键名，只有键值（或者说键名和键值是同一个值），所以key方法和value方法的行为完全一致。

let set = new Set(['red', 'green', 'blue']);

for (let item of set.keys()) {

console.log(item);

}

// red

// green

// blue

for (let item of set.values()) {

console.log(item);

}

// red

// green

// blue

for (let item of set.entries()) {

console.log(item);

}

// ["red", "red"]

// ["green", "green"]

// ["blue", "blue"]

上面代码中，entries方法返回的遍历器，同时包括键名和键值，所以每次输出一个数组，它的两个成员完全相等。

Set结构的实例默认可遍历，它的默认遍历器生成函数就是它的values方法。

Set.prototype[Symbol.iterator] === Set.prototype.values

// true

这意味着，可以省略values方法，直接用for...of循环遍历Set。

let set = new Set(['red', 'green', 'blue']);

for (let x of set) {

console.log(x);

}

// red

// green

// blue

##### （2）forEach()

Set结构的实例的forEach方法，用于对每个成员执行某种操作，没有返回值。

let set = new Set([1, 2, 3]);

set.forEach((value, key) => console.log(value \* 2) )

// 2

// 4

// 6

上面代码说明，forEach方法的参数就是一个处理函数。该函数的参数依次为键值、键名、集合本身（上例省略了该参数）。另外，forEach方法还可以有第二个参数，表示绑定的this对象。

##### （3）遍历的应用

扩展运算符（...）内部使用for...of循环，所以也可以用于Set结构。

let set = new Set(['red', 'green', 'blue']);

let arr = [...set];

// ['red', 'green', 'blue']

扩展运算符和Set结构相结合，就可以去除数组的重复成员。

let arr = [3, 5, 2, 2, 5, 5];

let unique = [...new Set(arr)];

// [3, 5, 2]

而且，数组的map和filter方法也可以用于Set了。

let set = new Set([1, 2, 3]);

set = new Set([...set].map(x => x \* 2));

// 返回Set结构：{2, 4, 6}

let set = new Set([1, 2, 3, 4, 5]);

set = new Set([...set].filter(x => (x % 2) == 0));

// 返回Set结构：{2, 4}

因此使用Set可以很容易地实现并集（Union）、交集（Intersect）和差集（Difference）。

let a = new Set([1, 2, 3]);

let b = new Set([4, 3, 2]);

// 并集

let union = new Set([...a, ...b]);

// Set {1, 2, 3, 4}

// 交集

let intersect = new Set([...a].filter(x => b.has(x)));

// set {2, 3}

// 差集

let difference = new Set([...a].filter(x => !b.has(x)));

// Set {1}

如果想在遍历操作中，同步改变原来的Set结构，目前没有直接的方法，但有两种变通方法。一种是利用原Set结构映射出一个新的结构，然后赋值给原来的Set结构；另一种是利用Array.from方法。

// 方法一

let set = new Set([1, 2, 3]);

set = new Set([...set].map(val => val \* 2));

// set的值是2, 4, 6

// 方法二

let set = new Set([1, 2, 3]);

set = new Set(Array.from(set, val => val \* 2));

// set的值是2, 4, 6

上面代码提供了两种方法，直接在遍历操作中改变原来的Set结构。

#### Weakest

WeakSet结构与Set类似，也是不重复的值的集合。但是，它与Set有两个区别。

首先，WeakSet的成员只能是对象，而不能是其他类型的值。

其次，WeakSet中的对象都是弱引用，即垃圾回收机制不考虑WeakSet对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于WeakSet之中。这个特点意味着，无法引用WeakSet的成员，因此WeakSet是不可遍历的。

var ws = new WeakSet();

ws.add(1)

// TypeError: Invalid value used in weak set

ws.add(Symbol())

// TypeError: invalid value used in weak set

上面代码试图向WeakSet添加一个数值和Symbol值，结果报错，因为WeakSet只能放置对象。

WeakSet是一个构造函数，可以使用new命令，创建WeakSet数据结构。

var ws = new WeakSet();

作为构造函数，WeakSet可以接受一个数组或类似数组的对象作为参数。（实际上，任何具有iterable接口的对象，都可以作为WeakSet的参数。）该数组的所有成员，都会自动成为WeakSet实例对象的成员。

var a = [[1,2], [3,4]];

var ws = new WeakSet(a);

上面代码中，a是一个数组，它有两个成员，也都是数组。将a作为WeakSet构造函数的参数，a的成员会自动成为WeakSet的成员。

注意，是a数组的成员成为WeakSet的成员，而不是a数组本身。这意味着，数组的成员只能是对象

var b = [3, 4];

var ws = new WeakSet(b);

// Uncaught TypeError: Invalid value used in weak set(…)

上面代码中，数组b的成员不是对象，加入WeaKSet就会报错。

WeakSet结构有以下三个方法。

* **WeakSet.prototype.add(value)**：向WeakSet实例添加一个新成员。
* **WeakSet.prototype.delete(value)**：清除WeakSet实例的指定成员。
* **WeakSet.prototype.has(value)**：返回一个布尔值，表示某个值是否在WeakSet实例之中。

下面是一个例子。

var ws = new WeakSet();

var obj = {};

var foo = {};

ws.add(window);

ws.add(obj);

ws.has(window); // true

ws.has(foo); // false

ws.delete(window);

ws.has(window); // false

WeakSet没有size属性，没有办法遍历它的成员。

ws.size // undefined

ws.forEach // undefined

ws.forEach(function(item){ console.log('WeakSet has ' + item)})

// TypeError: undefined is not a function

上面代码试图获取size和forEach属性，结果都不能成功。

WeakSet不能遍历，是因为成员都是弱引用，随时可能消失，遍历机制无法保证成员的存在，很可能刚刚遍历结束，成员就取不到了。WeakSet的一个用处，是储存DOM节点，而不用担心这些节点从文档移除时，会引发内存泄漏。

下面是WeakSet的另一个例子。

const foos = new WeakSet()

class Foo {

constructor() {

foos.add(this)

}

method () {

if (!foos.has(this)) {

throw new TypeError('Foo.prototype.method 只能在Foo的实例上调用！');

}

}

}

上面代码保证了Foo的实例方法，只能在Foo的实例上调用。这里使用WeakSet的好处是，foos对实例的引用，不会被计入内存回收机制，所以删除实例的时候，不用考虑foos，也不会出现内存泄漏。

### Map

#### Map结构的目的和基本用法

JavaScript的对象（Object），本质上是键值对的集合（Hash结构），但是传统上只能用字符串当作键。这给它的使用带来了很大的限制。

var data = {};

var element = document.getElementById('myDiv');

data[element] = 'metadata';

data['[object HTMLDivElement]'] // "metadata"

上面代码原意是将一个DOM节点作为对象data的键，但是由于对象只接受字符串作为键名，所以element被自动转为字符串[object HTMLDivElement]。

为了解决这个问题，ES6提供了Map数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object结构提供了“字符串—值”的对应，Map结构提供了“值—值”的对应，是一种更完善的Hash结构实现。如果你需要“键值对”的数据结构，Map比Object更合适。

var m = new Map();

var o = {p: 'Hello World'};

m.set(o, 'content')

m.get(o) // "content"

m.has(o) // true

m.delete(o) // true

m.has(o) // false

上面代码使用set方法，将对象o当作m的一个键，然后又使用get方法读取这个键，接着使用delete方法删除了这个键。

作为构造函数，Map也可以接受一个数组作为参数。该数组的成员是一个个表示键值对的数组。

var map = new Map([

['name', '张三'],

['title', 'Author']

]);

map.size // 2

map.has('name') // true

map.get('name') // "张三"

map.has('title') // true

map.get('title') // "Author"

上面代码在新建Map实例时，就指定了两个键name和title。

Map构造函数接受数组作为参数，实际上执行的是下面的算法。

var items = [

['name', '张三'],

['title', 'Author']

];

var map = new Map();

items.forEach(([key, value]) => map.set(key, value));

下面的例子中，字符串true和布尔值true是两个不同的键。

var m = new Map([

[true, 'foo'],

['true', 'bar']

]);

m.get(true) // 'foo'

m.get('true') // 'bar'

如果对同一个键多次赋值，后面的值将覆盖前面的值。

let map = new Map();

map

.set(1, 'aaa')

.set(1, 'bbb');

map.get(1) // "bbb"

上面代码对键1连续赋值两次，后一次的值覆盖前一次的值。

如果读取一个未知的键，则返回undefined。

new Map().get('asfddfsasadf')

// undefined

注意，只有对同一个对象的引用，Map结构才将其视为同一个键。这一点要非常小心。

var map = new Map();

map.set(['a'], 555);

map.get(['a']) // undefined

上面代码的set和get方法，表面是针对同一个键，但实际上这是两个值，内存地址是不一样的，因此get方法无法读取该键，返回undefined。

同理，同样的值的两个实例，在Map结构中被视为两个键。

var map = new Map();

var k1 = ['a'];

var k2 = ['a'];

map

.set(k1, 111)

.set(k2, 222);

map.get(k1) // 111

map.get(k2) // 222

上面代码中，变量k1和k2的值是一样的，但是它们在Map结构中被视为两个键。

由上可知，Map的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键。这就解决了同名属性碰撞（clash）的问题，我们扩展别人的库的时候，如果使用对象作为键名，就不用担心自己的属性与原作者的属性同名。

如果Map的键是一个简单类型的值（数字、字符串、布尔值），则只要两个值严格相等，Map将其视为一个键，包括0和-0。另外，虽然NaN不严格相等于自身，但Map将其视为同一个键。

let map = new Map();

map.set(NaN, 123);

map.get(NaN) // 123

map.set(-0, 123);

map.get(+0) // 123

#### 实例的属性和操作方法

Map结构的实例有以下属性和操作方法。

##### （1）size属性

size属性返回Map结构的成员总数。

let map = new Map();

map.set('foo', true);

map.set('bar', false);

map.size // 2

##### （2）set(key, value)

set方法设置key所对应的键值，然后返回整个Map结构。如果key已经有值，则键值会被更新，否则就新生成该键。

var m = new Map();

m.set("edition", 6) // 键是字符串

m.set(262, "standard") // 键是数值

m.set(undefined, "nah") // 键是undefined

set方法返回的是Map本身，因此可以采用链式写法。

let map = new Map()

.set(1, 'a')

.set(2, 'b')

.set(3, 'c');

##### （3）get(key)

get方法读取key对应的键值，如果找不到key，返回undefined。

var m = new Map();

var hello = function() {console.log("hello");}

m.set(hello, "Hello ES6!") // 键是函数

m.get(hello) // Hello ES6!

##### （4）has(key)

has方法返回一个布尔值，表示某个键是否在Map数据结构中

var m = new Map();

m.set("edition", 6);

m.set(262, "standard");

m.set(undefined, "nah");

m.has("edition") // true

m.has("years") // false

m.has(262) // true

m.has(undefined) // true

##### （5）delete(key)

delete方法删除某个键，返回true。如果删除失败，返回false。

var m = new Map();

m.set(undefined, "nah");

m.has(undefined) // true

m.delete(undefined)

m.has(undefined) // false

##### （6）clear()

clear方法清除所有成员，没有返回值。

let map = new Map();

map.set('foo', true);

map.set('bar', false);

map.size // 2

map.clear()

map.size // 0

#### 遍历方法

Map原生提供三个遍历器生成函数和一个遍历方法。

* keys()：返回键名的遍历器。
* values()：返回键值的遍历器。
* entries()：返回所有成员的遍历器。
* forEach()：遍历Map的所有成员。

需要特别注意的是，Map的遍历顺序就是插入顺序。

下面是使用实例。

let map = new Map([

['F', 'no'],

['T', 'yes'],

]);

for (let key of map.keys()) {

console.log(key);

}

// "F"

// "T"

for (let value of map.values()) {

console.log(value);

}

// "no"

// "yes"

for (let item of map.entries()) {

console.log(item[0], item[1]);

}

// "F" "no"

// "T" "yes"

// 或者

for (let [key, value] of map.entries()) {

console.log(key, value);

}

// 等同于使用map.entries()

for (let [key, value] of map) {

console.log(key, value);

}

上面代码最后的那个例子，表示Map结构的默认遍历器接口（Symbol.iterator属性），就是entries方法。

map[Symbol.iterator] === map.entries

// true

Map结构转为数组结构，比较快速的方法是结合使用扩展运算符（...）。

let map = new Map([

[1, 'one'],

[2, 'two'],

[3, 'three'],

]);

[...map.keys()]

// [1, 2, 3]

[...map.values()]

// ['one', 'two', 'three']

[...map.entries()]

// [[1,'one'], [2, 'two'], [3, 'three']]

[...map]

// [[1,'one'], [2, 'two'], [3, 'three']]

结合数组的map方法、filter方法，可以实现Map的遍历和过滤（Map本身没有map和filter方法）。

let map0 = new Map()

.set(1, 'a')

.set(2, 'b')

.set(3, 'c');

let map1 = new Map(

[...map0].filter(([k, v]) => k < 3)

);

// 产生Map结构 {1 => 'a', 2 => 'b'}

let map2 = new Map(

[...map0].map(([k, v]) => [k \* 2, '\_' + v])

);

// 产生Map结构 {2 => '\_a', 4 => '\_b', 6 => '\_c'}

此外，Map还有一个forEach方法，与数组的forEach方法类似，也可以实现遍历。

map.forEach(function(value, key, map) {

console.log("Key: %s, Value: %s", key, value);

});

forEach方法还可以接受第二个参数，用来绑定this。

var reporter = {

report: function(key, value) {

console.log("Key: %s, Value: %s", key, value);

}

};

map.forEach(function(value, key, map) {

this.report(key, value);

}, reporter);

上面代码中，forEach方法的回调函数的this，就指向reporter。

#### 与其他数据结构的互相转换

##### （1）Map转为数组

前面已经提过，Map转为数组最方便的方法，就是使用扩展运算符（...）。

let myMap = new Map().set(true, 7).set({foo: 3}, ['abc']);

[...myMap]

// [ [ true, 7 ], [ { foo: 3 }, [ 'abc' ] ] ]

##### （2）数组转为Map

将数组转入Map构造函数，就可以转为Map。

new Map([[true, 7], [{foo: 3}, ['abc']]])

// Map {true => 7, Object {foo: 3} => ['abc']}

##### （3）Map转为对象

如果所有Map的键都是字符串，它可以转为对象。

function strMapToObj(strMap) {

let obj = Object.create(null);

for (let [k,v] of strMap) {

obj[k] = v;

}

return obj;

}

let myMap = new Map().set('yes', true).set('no', false);

strMapToObj(myMap)

// { yes: true, no: false }

##### （4）对象转为Map

function objToStrMap(obj) {

let strMap = new Map();

for (let k of Object.keys(obj)) {

strMap.set(k, obj[k]);

}

return strMap;

}

objToStrMap({yes: true, no: false})

// [ [ 'yes', true ], [ 'no', false ] ]

##### （5）Map转为JSON

Map转为JSON要区分两种情况。一种情况是，Map的键名都是字符串，这时可以选择转为对象JSON。

function strMapToJson(strMap) {

return JSON.stringify(strMapToObj(strMap));

}

let myMap = new Map().set('yes', true).set('no', false);

strMapToJson(myMap)

// '{"yes":true,"no":false}'

另一种情况是，Map的键名有非字符串，这时可以选择转为数组JSON。

function mapToArrayJson(map) {

return JSON.stringify([...map]);

}

let myMap = new Map().set(true, 7).set({foo: 3}, ['abc']);

mapToArrayJson(myMap)

// '[[true,7],[{"foo":3},["abc"]]]'

##### （6）JSON转为Map

JSON转为Map，正常情况下，所有键名都是字符串。

function jsonToStrMap(jsonStr) {

return objToStrMap(JSON.parse(jsonStr));

}

jsonToStrMap('{"yes":true,"no":false}')

// Map {'yes' => true, 'no' => false}

但是，有一种特殊情况，整个JSON就是一个数组，且每个数组成员本身，又是一个有两个成员的数组。这时，它可以一一对应地转为Map。这往往是数组转为JSON的逆操作。

function jsonToMap(jsonStr) {

return new Map(JSON.parse(jsonStr));

}

jsonToMap('[[true,7],[{"foo":3},["abc"]]]')

// Map {true => 7, Object {foo: 3} => ['abc']}

### WeakMap

WeakMap结构与Map结构基本类似，唯一的区别是它只接受对象作为键名（null除外），不接受其他类型的值作为键名，而且键名所指向的对象，不计入垃圾回收机制。

var map = new WeakMap()

map.set(1, 2)

// TypeError: 1 is not an object!

map.set(Symbol(), 2)

// TypeError: Invalid value used as weak map key

上面代码中，如果将1和Symbol作为WeakMap的键名，都会报错

WeakMap的设计目的在于，键名是对象的弱引用（垃圾回收机制不将该引用考虑在内），所以其所对应的对象可能会被自动回收。当对象被回收后，WeakMap自动移除对应的键值对。典型应用是，一个对应DOM元素的WeakMap结构，当某个DOM元素被清除，其所对应的WeakMap记录就会自动被移除。基本上，WeakMap的专用场合就是，它的键所对应的对象，可能会在将来消失。WeakMap结构有助于防止内存泄漏。

下面是WeakMap结构的一个例子，可以看到用法上与Map几乎一样。

var wm = new WeakMap();

var element = document.querySelector(".element");

wm.set(element, "Original");

wm.get(element) // "Original"

element.parentNode.removeChild(element);

element = null;

wm.get(element) // undefined

上面代码中，变量wm是一个WeakMap实例，我们将一个DOM节点element作为键名，然后销毁这个节点，element对应的键就自动消失了，再引用这个键名就返回undefined。

WeakMap与Map在API上的区别主要是两个，一是没有遍历操作（即没有key()、values()和entries()方法），也没有size属性；二是无法清空，即不支持clear方法。这与WeakMap的键不被计入引用、被垃圾回收机制忽略有关。因此，WeakMap只有四个方法可用：get()、set()、has()、delete()。

var wm = new WeakMap();

wm.size

// undefined

wm.forEach

// undefined

前文说过，WeakMap应用的典型场合就是DOM节点作为键名。下面是一个例子。

let myElement = document.getElementById('logo');

let myWeakmap = new WeakMap();

myWeakmap.set(myElement, {timesClicked: 0});

myElement.addEventListener('click', function() {

let logoData = myWeakmap.get(myElement);

logoData.timesClicked++;

}, false);

上面代码中，myElement是一个 DOM 节点，每当发生click事件，就更新一下状态。我们将这个状态作为键值放在 WeakMap 里，对应的键名就是myElement。一旦这个 DOM 节点删除，该状态就会自动消失，不存在内存泄漏风险。

WeakMap 的另一个用处是部署私有属性。

let \_counter = new WeakMap();

let \_action = new WeakMap();

class Countdown {

constructor(counter, action) {

\_counter.set(this, counter);

\_action.set(this, action);

}

dec() {

let counter = \_counter.get(this);

if (counter < 1) return;

counter--;

\_counter.set(this, counter);

if (counter === 0) {

\_action.get(this)();

}

}

}

let c = new Countdown(2, () => console.log('DONE'));

c.dec()

c.dec()

// DONE

上面代码中，Countdown类的两个内部属性\_counter和\_action，是实例的弱引用，所以如果删除实例，它们也就随之消失，不会造成内存泄漏。

## Proxy和Reflect

### Proxy概述

Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”（meta programming），即对编程语言进行编程。

Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。

var obj = new Proxy({}, {

get: function (target, key, receiver) {

console.log(`getting ${key}!`);

return Reflect.get(target, key, receiver);

},

set: function (target, key, value, receiver) {

console.log(`setting ${key}!`);

return Reflect.set(target, key, value, receiver);

}

});

上面代码对一个空对象架设了一层拦截，重定义了属性的读取（get）和设置（set）行为。这里暂时先不解释具体的语法，只看运行结果。对设置了拦截行为的对象obj，去读写它的属性，就会得到下面的结果。

obj.count = 1

// setting count!

++obj.count

// getting count!

// setting count!

// 2

上面代码说明，Proxy 实际上重载（overload）了点运算符，即用自己的定义覆盖了语言的原始定义。

ES6 原生提供 Proxy 构造函数，用来生成 Proxy 实例。

var proxy = new Proxy(target, handler);

Proxy 对象的所有用法，都是上面这种形式，不同的只是handler参数的写法。其中，new Proxy()表示生成一个Proxy实例，target参数表示所要拦截的目标对象，handler参数也是一个对象，用来定制拦截行为。

下面是另一个拦截读取属性行为的例子。

var proxy = new Proxy({}, {

get: function(target, property) {

return 35;

}

});

proxy.time // 35

proxy.name // 35

proxy.title // 35

上面代码中，作为构造函数，Proxy接受两个参数。第一个参数是所要代理的目标对象（上例是一个空对象），即如果没有Proxy的介入，操作原来要访问的就是这个对象；第二个参数是一个配置对象，对于每一个被代理的操作，需要提供一个对应的处理函数，该函数将拦截对应的操作。比如，上面代码中，配置对象有一个get方法，用来拦截对目标对象属性的访问请求。get方法的两个参数分别是目标对象和所要访问的属性。可以看到，由于拦截函数总是返回35，所以访问任何属性都得到35。

注意，要使得Proxy起作用，必须针对Proxy实例（上例是proxy对象）进行操作，而不是针对目标对象（上例是空对象）进行操作。

如果handler没有设置任何拦截，那就等同于直接通向原对象。

var target = {};

var handler = {};

var proxy = new Proxy(target, handler);

proxy.a = 'b';

target.a // "b"

上面代码中，handler是一个空对象，没有任何拦截效果，访问handler就等同于访问target。

一个技巧是将Proxy对象，设置到object.proxy属性，从而可以在object对象上调用。

var object = { proxy: new Proxy(target, handler) };

Proxy 实例也可以作为其他对象的原型对象。

var proxy = new Proxy({}, {

get: function(target, property) {

return 35;

}

});

let obj = Object.create(proxy);

obj.time // 35

上面代码中，proxy对象是obj对象的原型，obj对象本身并没有time属性，所以根据原型链，会在proxy对象上读取该属性，导致被拦截。

同一个拦截器函数，可以设置拦截多个操作。

var handler = {

get: function(target, name) {

if (name === 'prototype') {

return Object.prototype;

}

return 'Hello, ' + name;

},

apply: function(target, thisBinding, args) {

return args[0];

},

construct: function(target, args) {

return {value: args[1]};

}

};

var fproxy = new Proxy(function(x, y) {

return x + y;

}, handler);

fproxy(1, 2) // 1

new fproxy(1,2) // {value: 2}

fproxy.prototype === Object.prototype // true

fproxy.foo // "Hello, foo"

下面是 Proxy 支持的拦截操作一览。

对于可以设置、但没有设置拦截的操作，则直接落在目标对象上，按照原先的方式产生结果。

**（1）get(target, propKey, receiver)**

拦截对象属性的读取，比如proxy.foo和proxy['foo']。

最后一个参数receiver是一个对象，可选，参见下面Reflect.get的部分。

**（2）set(target, propKey, value, receiver)**

拦截对象属性的设置，比如proxy.foo = v或proxy['foo'] = v，返回一个布尔值。

**（3）has(target, propKey)**

拦截propKey in proxy的操作，以及对象的hasOwnProperty方法，返回一个布尔值

**（4）deleteProperty(target, propKey)**

拦截delete proxy[propKey]的操作，返回一个布尔值。

**（5）ownKeys(target)**

拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)，返回一个数组。该方法返回对象所有自身的属性，而Object.keys()仅返回对象可遍历的属性。

**（6）getOwnPropertyDescriptor(target, propKey)**

拦截Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。

**（7）defineProperty(target, propKey, propDesc)**

拦截Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。

**（8）preventExtensions(target)**

拦截Object.preventExtensions(proxy)，返回一个布尔值。

**（9）getPrototypeOf(target)**

拦截Object.getPrototypeOf(proxy)，返回一个对象。

**（10）isExtensible(target)**

拦截Object.isExtensible(proxy)，返回一个布尔值。

**（11）setPrototypeOf(target, proto)**

拦截Object.setPrototypeOf(proxy, proto)，返回一个布尔值。

如果目标对象是函数，那么还有两种额外操作可以拦截。

**（12）apply(target, object, args)**

拦截 Proxy 实例作为函数调用的操作，比如proxy(...args)、proxy.call(object, ...args)、proxy.apply(...)。

**13）construct(target, args)**

拦截 Proxy 实例作为构造函数调用的操作，比如new proxy(...args)。

### Proxy实例的方法

下面是上面这些拦截方法的详细介绍

#### get()

get方法用于拦截某个属性的读取操作。上文已经有一个例子，下面是另一个拦截读取操作的例子。

var person = {

name: "张三"

};

var proxy = new Proxy(person, {

get: function(target, property) {

if (property in target) {

return target[property];

} else {

throw new ReferenceError("Property \"" + property + "\" does not exist.");

}

}

});

proxy.name // "张三"

proxy.age // 抛出一个错误

上面代码表示，如果访问目标对象不存在的属性，会抛出一个错误。如果没有这个拦截函数，访问不存在的属性，只会返回undefined。

get 方法可以继承

let proto = new Proxy({}, {

get(target, propertyKey, receiver) {

console.log('GET '+propertyKey);

return target[propertyKey];

}

});

let obj = Object.create(proto);

obj.xxx // "GET xxx"

上面代码中，拦截操作定义在Prototype对象上面，所以如果读取obj对象继承的属性时，拦截会生效。

下面的例子使用get拦截，实现数组读取负数的索引。

function createArray(...elements) {

let handler = {

get(target, propKey, receiver) {

let index = Number(propKey);

if (index < 0) {

propKey = String(target.length + index);

}

return Reflect.get(target, propKey, receiver);

}

};

let target = [];

target.push(...elements);

return new Proxy(target, handler);

}

let arr = createArray('a', 'b', 'c');

arr[-1] // c

上面代码中，数组的位置参数是-1，就会输出数组的倒数最后一个成员。

利用 Proxy，可以将读取属性的操作（get），转变为执行某个函数，从而实现属性的链式操作。

var pipe = (function () {

return function (value) {

var funcStack = [];

var oproxy = new Proxy({} , {

get : function (pipeObject, fnName) {

if (fnName === 'get') {

return funcStack.reduce(function (val, fn) {

return fn(val);

},value);

}

funcStack.push(window[fnName]);

return oproxy;

}

});

return oproxy;

}

}());

var double = n => n \* 2;

var pow = n => n \* n;

var reverseInt = n => n.toString().split("").reverse().join("") | 0;

pipe(3).double.pow.reverseInt.get; // 63

上面代码设置 Proxy 以后，达到了将函数名链式使用的效果。

下面的例子则是利用get拦截，实现一个生成各种DOM节点的通用函数dom。

const dom = new Proxy({}, {

get(target, property) {

return function(attrs = {}, ...children) {

const el = document.createElement(property);

for (let prop of Object.keys(attrs)) {

el.setAttribute(prop, attrs[prop]);

}

for (let child of children) {

if (typeof child === 'string') {

child = document.createTextNode(child);

}

el.appendChild(child);

}

return el;

}

}

});

const el = dom.div({},

'Hello, my name is ',

dom.a({href: '//example.com'}, 'Mark'),

'. I like:',

dom.ul({},

dom.li({}, 'The web'),

dom.li({}, 'Food'),

dom.li({}, '…actually that\'s it')

)

);

document.body.appendChild(el);

#### set()

set方法用来拦截某个属性的赋值操作。

假定Person对象有一个age属性，该属性应该是一个不大于200的整数，那么可以使用Proxy保证age的属性值符合要求。

let validator = {

set: function(obj, prop, value) {

if (prop === 'age') {

if (!Number.isInteger(value)) {

throw new TypeError('The age is not an integer');

}

if (value > 200) {

throw new RangeError('The age seems invalid');

}

}

// 对于age以外的属性，直接保存

obj[prop] = value;

}

};

let person = new Proxy({}, validator);

person.age = 100;

person.age // 100

person.age = 'young' // 报错

person.age = 300 // 报错

上面代码中，由于设置了存值函数set，任何不符合要求的age属性赋值，都会抛出一个错误。利用set方法，还可以数据绑定，即每当对象发生变化时，会自动更新DOM。

有时，我们会在对象上面设置内部属性，属性名的第一个字符使用下划线开头，表示这些属性不应该被外部使用。结合get和set方法，就可以做到防止这些内部属性被外部读写。

var handler = {

get (target, key) {

invariant(key, 'get');

return target[key];

},

set (target, key, value) {

invariant(key, 'set');

target[key] = value;

return true;

}

};

function invariant (key, action) {

if (key[0] === '\_') {

throw new Error(`Invalid attempt to ${action} private "${key}" property`);

}

}

var target = {};

var proxy = new Proxy(target, handler);

proxy.\_prop

// Error: Invalid attempt to get private "\_prop" property

proxy.\_prop = 'c'

// Error: Invalid attempt to set private "\_prop" property

上面的代码中，只要读写的属性名的第一个字符是下划线，一律抛错，从而达到禁止读写内容的目的

#### apply()

apply方法拦截函数的调用、call和apply操作。

var handler = {

apply (target, ctx, args) {

return Reflect.apply(...arguments);

}

};

apply方法可以接受三个参数，分别是目标对象、目标对象的上下文对象（this）和目标对象的参数数组

下面是一个例子

var target = function () { return 'I am the target'; };

var handler = {

apply: function () {

return 'I am the proxy';

}

};

var p = new Proxy(target, handler);

p()

// "I am the proxy"

上面代码中，变量p是Proxy的实例，当它作为函数调用时（p()），就会被apply方法拦截，返回一个字符串。

下面是另一个例子

var twice = {

apply (target, ctx, args) {

return Reflect.apply(...arguments) \* 2;

}

};

function sum (left, right) {

return left + right;

};

var proxy = new Proxy(sum, twice);

proxy(1, 2) // 6

proxy.call(null, 5, 6) // 22

proxy.apply(null, [7, 8]) // 30

上面代码中，每当执行proxy函数（直接调用或call和apply调用），就会被apply方法拦截。

另外，直接调用Reflect.apply方法，也会被拦截。

Reflect.apply(proxy, null, [9, 10]) // 38

#### has()

has方法用来拦截HasProperty操作，即判断对象是否具有某个属性时，这个方法会生效。典型的操作就是in运算符。

var handler = {

has (target, key) {

if (key[0] === '\_') {

return false;

}

return key in target;

}

};

var target = { \_prop: 'foo', prop: 'foo' };

var proxy = new Proxy(target, handler);

'\_prop' in proxy // false

上面代码中，如果原对象的属性名的第一个字符是下划线，proxy.has就会返回false，从而不会被in运算符发现。

如果原对象不可配置或者禁止扩展，这时has拦截会报错。

var obj = { a: 10 };

Object.preventExtensions(obj);

var p = new Proxy(obj, {

has: function(target, prop) {

return false;

}

});

'a' in p // TypeError is thrown

上面代码中，obj对象禁止扩展，结果使用has拦截就会报错。

值得注意的是，has方法拦截的是HasProperty操作，而不是HasOwnProperty操作，即has方法不判断一个属性是对象自身的属性，还是继承的属性。

另外，虽然for...in循环也用到了in运算符，但是has拦截对for...in循环不生效。

let stu1 = {name: '张三', score: 59};

let stu2 = {name: '李四', score: 99};

let handler = {

has(target, prop) {

if (prop === 'score' && target[prop] < 60) {

console.log(`${target.name} 不及格`);

return false;

}

return prop in target;

}

}

let oproxy1 = new Proxy(stu1, handler);

let oproxy2 = new Proxy(stu2, handler);

'score' in oproxy1

// 张三 不及格

// false

'score' in oproxy2

// true

for (let a in oproxy1) {

console.log(oproxy1[a]);

}

// 张三

// 59

for (let b in oproxy2) {

console.log(oproxy2[b]);

}

// 李四

// 99

上面代码中，has拦截只对in循环生效，对for...in循环不生效，导致不符合要求的属性没有被排除在for...in循环之外。

#### construct()

construct方法用于拦截new命令，下面是拦截对象的写法。

var handler = {

construct (target, args, newTarget) {

return new target(...args);

}

};

construct方法可以接受两个参数。

* target: 目标对象
* args：构建函数的参数对象

下面是一个例子。

var p = new Proxy(function() {}, {

construct: function(target, args) {

console.log('called: ' + args.join(', '));

return { value: args[0] \* 10 };

}

});

new p(1).value

// "called: 1"

// 10

construct方法返回的必须是一个对象，否则会报错。

var p = new Proxy(function() {}, {

construct: function(target, argumentsList) {

return 1;

}

});

new p() // 报错

#### deleteProperty()

deleteProperty方法用于拦截delete操作，如果这个方法抛出错误或者返回false，当前属性就无法被delete命令删除。

var handler = {

deleteProperty (target, key) {

invariant(key, 'delete');

return true;

}

};

function invariant (key, action) {

if (key[0] === '\_') {

throw new Error(`Invalid attempt to ${action} private "${key}" property`);

}

}

var target = { \_prop: 'foo' };

var proxy = new Proxy(target, handler);

delete proxy.\_prop

// Error: Invalid attempt to delete private "\_prop" property

上面代码中，deleteProperty方法拦截了delete操作符，删除第一个字符为下划线的属性会报错。

## Iterator和for...of循环

### Iterator（遍历器）的概念

JavaScript原有的表示“集合”的数据结构，主要是数组（Array）和对象（Object），ES6又添加了Map和Set。这样就有了四种数据集合，用户还可以组合使用它们，定义自己的数据结构，比如数组的成员是Map，Map的成员是对象。这样就需要一种统一的接口机制，来处理所有不同的数据结构。

遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是ES6创造了一种新的遍历命令for...of循环，Iterator接口主要供for...of消费

Iterator的遍历过程是这样的。

（1）创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。

（2）第一次调用指针对象的next方法，可以将指针指向数据结构的第一个成员。

（3）第二次调用指针对象的next方法，指针就指向数据结构的第二个成员。

（4）不断调用指针对象的next方法，直到它指向数据结构的结束位置。

每一次调用next方法，都会返回数据结构的当前成员的信息。具体来说，就是返回一个包含value和done两个属性的对象。其中，value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。

下面是一个模拟next方法返回值的例子。

var it = makeIterator(['a', 'b']);

it.next() // { value: "a", done: false }

it.next() // { value: "b", done: false }

it.next() // { value: undefined, done: true }

function makeIterator(array) {

var nextIndex = 0;

return {

next: function() {

return nextIndex < array.length ?

{value: array[nextIndex++], done: false} :

{value: undefined, done: true};

}

};

}

上面代码定义了一个makeIterator函数，它是一个遍历器生成函数，作用就是返回一个遍历器对象。对数组['a', 'b']执行这个函数，就会返回该数组的遍历器对象（即指针对象）it。

指针对象的next方法，用来移动指针。开始时，指针指向数组的开始位置。然后，每次调用next方法，指针就会指向数组的下一个成员。第一次调用，指向a；第二次调用，指向b。

next方法返回一个对象，表示当前数据成员的信息。这个对象具有value和done两个属性，value属性返回当前位置的成员，done属性是一个布尔值，表示遍历是否结束，即是否还有必要再一次调用next方法。

总之，调用指针对象的next方法，就可以遍历事先给定的数据结构。

对于遍历器对象来说，done: false和value: undefined属性都是可以省略的，因此上面的makeIterator函数可以简写成下面的形式。

function makeIterator(array) {

var nextIndex = 0;

return {

next: function() {

return nextIndex < array.length ?

{value: array[nextIndex++]} :

{done: true};

}

};

}

由于Iterator只是把接口规格加到数据结构之上，所以，遍历器与它所遍历的那个数据结构，实际上是分开的，完全可以写出没有对应数据结构的遍历器对象，或者说用遍历器对象模拟出数据结构。下面是一个无限运行的遍历器对象的例子。

var it = idMaker();

it.next().value // '0'

it.next().value // '1'

it.next().value // '2'

// ...

function idMaker() {

var index = 0;

return {

next: function() {

return {value: index++, done: false};

}

};

}

上面的例子中，遍历器生成函数idMaker，返回一个遍历器对象（即指针对象）。但是并没有对应的数据结构，或者说，遍历器对象自己描述了一个数据结构出来。

在ES6中，有些数据结构原生具备Iterator接口（比如数组），即不用任何处理，就可以被for...of循环遍历，有些就不行（比如对象）。原因在于，这些数据结构原生部署了Symbol.iterator属性（详见下文），另外一些数据结构没有。凡是部署了Symbol.iterator属性的数据结构，就称为部署了遍历器接口。调用这个接口，就会返回一个遍历器对象。

如果使用TypeScript的写法，遍历器接口（Iterable）、指针对象（Iterator）和next方法返回值的规格可以描述如下。

interface Iterable {

[Symbol.iterator]() : Iterator,

}

interface Iterator {

next(value?: any) : IterationResult,

}

interface IterationResult {

value: any,

done: boolean,

}

### 数据结构的默认Iterator接口

Iterator接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即for...of循环（详见下文）。当使用for...of循环遍历某种数据结构时，该循环会自动去寻找Iterator接口。

一种数据结构只要部署了Iterator接口，我们就称这种数据结构是”可遍历的“（iterable）。

ES6规定，默认的Iterator接口部署在数据结构的Symbol.iterator属性，或者说，一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历的”（iterable）。Symbol.iterator属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器。至于属性名Symbol.iterator，它是一个表达式，返回Symbol对象的iterator属性，这是一个预定义好的、类型为Symbol的特殊值，所以要放在方括号内。（参见Symbol一章）。

const obj = {

[Symbol.iterator] : function () {

return {

next: function () {

return {

value: 1,

done: true

};

}

};

}

};

上面代码中，对象obj是可遍历的（iterable），因为具有Symbol.iterator属性。执行这个属性，会返回一个遍历器对象。该对象的根本特征就是具有next方法。每次调用next方法，都会返回一个代表当前成员的信息对象，具有value和done两个属性

在ES6中，有三类数据结构原生具备Iterator接口：数组、某些类似数组的对象、Set和Map结构。

let arr = ['a', 'b', 'c'];

let iter = arr[Symbol.iterator]();

iter.next() // { value: 'a', done: false }

iter.next() // { value: 'b', done: false }

iter.next() // { value: 'c', done: false }

iter.next() // { value: undefined, done: true }

上面代码中，变量arr是一个数组，原生就具有遍历器接口，部署在arr的Symbol.iterator属性上面。所以，调用这个属性，就得到遍历器对象。

上面提到，原生就部署Iterator接口的数据结构有三类，对于这三类数据结构，不用自己写遍历器生成函数，for...of循环会自动遍历它们。除此之外，其他数据结构（主要是对象）的Iterator接口，都需要自己在Symbol.iterator属性上面部署，这样才会被for...of循环遍历。

对象（Object）之所以没有默认部署Iterator接口，是因为对象的哪个属性先遍历，哪个属性后遍历是不确定的，需要开发者手动指定。本质上，遍历器是一种线性处理，对于任何非线性的数据结构，部署遍历器接口，就等于部署一种线性转换。不过，严格地说，对象部署遍历器接口并不是很必要，因为这时对象实际上被当作Map结构使用，ES5没有Map结构，而ES6原生提供了。

一个对象如果要有可被for...of循环调用的Iterator接口，就必须在Symbol.iterator的属性上部署遍历器生成方法（原型链上的对象具有该方法也可）。

class RangeIterator {

constructor(start, stop) {

this.value = start;

this.stop = stop;

}

[Symbol.iterator]() { return this; }

next() {

var value = this.value;

if (value < this.stop) {

this.value++;

return {done: false, value: value};

} else {

return {done: true, value: undefined};

}

}

}

function range(start, stop) {

return new RangeIterator(start, stop);

}

for (var value of range(0, 3)) {

console.log(value);

}

上面代码是一个类部署Iterator接口的写法。Symbol.iterator属性对应一个函数，执行后返回当前对象的遍历器对象

下面是通过遍历器实现指针结构的例子。

function Obj(value) {

this.value = value;

this.next = null;

}

Obj.prototype[Symbol.iterator] = function() {

var iterator = {

next: next

};

var current = this;

function next() {

if (current) {

var value = current.value;

current = current.next;

return {

done: false,

value: value

};

} else {

return {

done: true

};

}

}

return iterator;

}

var one = new Obj(1);

var two = new Obj(2);

var three = new Obj(3);

one.next = two;

two.next = three;

for (var i of one){

console.log(i);

}

// 1

// 2

// 3

上面代码首先在构造函数的原型链上部署Symbol.iterator方法，调用该方法会返回遍历器对象iterator，调用该对象的next方法，在返回一个值的同时，自动将内部指针移到下一个实例。

下面是另一个为对象添加Iterator接口的例子。

let obj = {

data: [ 'hello', 'world' ],

[Symbol.iterator]() {

const self = this;

let index = 0;

return {

next() {

if (index < self.data.length) {

return {

value: self.data[index++],

done: false

};

} else {

return { value: undefined, done: true };

}

}

};

}

};

对于类似数组的对象（存在数值键名和length属性），部署Iterator接口，有一个简便方法，就是Symbol.iterator方法直接引用数组的Iterator接口。

NodeList.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];

// 或者

NodeList.prototype[Symbol.iterator] = [][Symbol.iterator];

[...document.querySelectorAll('div')] // 可以执行了

下面是类似数组的对象调用数组的Symbol.iterator方法的例子。

let iterable = {

0: 'a',

1: 'b',

2: 'c',

length: 3,

[Symbol.iterator]: Array.prototype[Symbol.iterator]

};

for (let item of iterable) {

console.log(item); // 'a', 'b', 'c'

}

注意，普通对象部署数组的Symbol.iterator方法，并无效果。

let iterable = {

a: 'a',

b: 'b',

c: 'c',

length: 3,

[Symbol.iterator]: Array.prototype[Symbol.iterator]

};

for (let item of iterable) {

console.log(item); // undefined, undefined, undefined

}

如果Symbol.iterator方法对应的不是遍历器生成函数（即会返回一个遍历器对象），解释引擎将会报错。

var obj = {};

obj[Symbol.iterator] = () => 1;

[...obj] // TypeError: [] is not a function

上面代码中，变量obj的Symbol.iterator方法对应的不是遍历器生成函数，因此报错。

有了遍历器接口，数据结构就可以用for...of循环遍历（详见下文），也可以使用while循环遍历。

var $iterator = ITERABLE[Symbol.iterator]();

var $result = $iterator.next();

while (!$result.done) {

var x = $result.value;

// ...

$result = $iterator.next();

}

上面代码中，ITERABLE代表某种可遍历的数据结构，$iterator是它的遍历器对象。遍历器对象每次移动指针（next方法），都检查一下返回值的done属性，如果遍历还没结束，就移动遍历器对象的指针到下一步（next方法），不断循环。

### 调用Iterator接口的场合

有一些场合会默认调用Iterator接口（即Symbol.iterator方法），除了下文会介绍的for...of循环，还有几个别的场合。

#### （1）解构赋值

对数组和Set结构进行解构赋值时，会默认调用Symbol.iterator方法。

let set = new Set().add('a').add('b').add('c');

let [x,y] = set;

// x='a'; y='b'

let [first, ...rest] = set;

// first='a'; rest=['b','c'];

#### （2）扩展运算符

扩展运算符（...）也会调用默认的iterator接口。

// 例一

var str = 'hello';

[...str] // ['h','e','l','l','o']

// 例二

let arr = ['b', 'c'];

['a', ...arr, 'd']

// ['a', 'b', 'c', 'd']

上面代码的扩展运算符内部就调用Iterator接口。

实际上，这提供了一种简便机制，可以将任何部署了Iterator接口的数据结构，转为数组。也就是说，只要某个数据结构部署了Iterator接口，就可以对它使用扩展运算符，将其转为数组。

let arr = [...iterable];

#### （3）yield\*

yield\*后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口。

let generator = function\* () {

yield 1;

yield\* [2,3,4];

yield 5;

};

var iterator = generator();

iterator.next() // { value: 1, done: false }

iterator.next() // { value: 2, done: false }

iterator.next() // { value: 3, done: false }

iterator.next() // { value: 4, done: false }

iterator.next() // { value: 5, done: false }

iterator.next() // { value: undefined, done: true }

#### （4）其他场合

由于数组的遍历会调用遍历器接口，所以任何接受数组作为参数的场合，其实都调用了遍历器接口。下面是一些例子。

* for...of
* Array.from()
* Map(), Set(), WeakMap(), WeakSet()（比如new Map([['a',1],['b',2]])）
* Promise.all()
* Promise.race()

### 字符串的Iterator 接口

字符串是一个类似数组的对象，也原生具有Iterator接口。

var someString = "hi";

typeof someString[Symbol.iterator]

// "function"

var iterator = someString[Symbol.iterator]();

iterator.next() // { value: "h", done: false }

iterator.next() // { value: "i", done: false }

iterator.next() // { value: undefined, done: true }

上面代码中，调用Symbol.iterator方法返回一个遍历器对象，在这个遍历器上可以调用next方法，实现对于字符串的遍历。

可以覆盖原生的Symbol.iterator方法，达到修改遍历器行为的目的。

var str = new String("hi");

[...str] // ["h", "i"]

str[Symbol.iterator] = function() {

return {

next: function() {

if (this.\_first) {

this.\_first = false;

return { value: "bye", done: false };

} else {

return { done: true };

}

},

\_first: true

};

};

[...str] // ["bye"]

str // "hi"

上面代码中，字符串str的Symbol.iterator方法被修改了，所以扩展运算符（...）返回的值变成了bye，而字符串本身还是hi。

## Generator函数

### 简介

#### 基本概念

Generator函数是ES6提供的一种异步编程解决方案，语法行为与传统函数完全不同。本章详细介绍Generator函数的语法和API，它的异步编程应用请看《异步操作》一章。

Generator函数有多种理解角度。从语法上，首先可以把它理解成，Generator函数是一个状态机，封装了多个内部状态。

执行Generator函数会返回一个遍历器对象，也就是说，Generator函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，可以依次遍历Generator函数内部的每一个状态。

形式上，Generator函数是一个普通函数，但是有两个特征。一是，function关键字与函数名之间有一个星号；二是，函数体内部使用yield语句，定义不同的内部状态（yield语句在英语里的意思就是“产出”）。

function\* helloWorldGenerator() {

yield 'hello';

yield 'world';

return 'ending';

}

var hw = helloWorldGenerator();

上面代码定义了一个Generator函数helloWorldGenerator，它内部有两个yield语句“hello”和“world”，即该函数有三个状态：hello，world和return语句（结束执行）。

然后，Generator函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用Generator函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是上一章介绍的遍历器对象（Iterator Object）。

下一步，必须调用遍历器对象的next方法，使得指针移向下一个状态。也就是说，每次调用next方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield语句（或return语句）为止。换言之，Generator函数是分段执行的，yield语句是暂停执行的标记，而next方法可以恢复执行。

hw.next()

// { value: 'hello', done: false }

hw.next()

// { value: 'world', done: false }

hw.next()

// { value: 'ending', done: true }

hw.next()

// { value: undefined, done: true }

上面代码一共调用了四次next方法。

第一次调用，Generator函数开始执行，直到遇到第一个yield语句为止。next方法返回一个对象，它的value属性就是当前yield语句的值hello，done属性的值false，表示遍历还没有结束。

第二次调用，Generator函数从上次yield语句停下的地方，一直执行到下一个yield语句。next方法返回的对象的value属性就是当前yield语句的值world，done属性的值false，表示遍历还没有结束。

第三次调用，Generator函数从上次yield语句停下的地方，一直执行到return语句（如果没有return语句，就执行到函数结束）。next方法返回的对象的value属性，就是紧跟在return语句后面的表达式的值（如果没有return语句，则value属性的值为undefined），done属性的值true，表示遍历已经结束。

第四次调用，此时Generator函数已经运行完毕，next方法返回对象的value属性为undefined，done属性为true。以后再调用next方法，返回的都是这个值。

总结一下，调用Generator函数，返回一个遍历器对象，代表Generator函数的内部指针。以后，每次调用遍历器对象的next方法，就会返回一个有着value和done两个属性的对象。value属性表示当前的内部状态的值，是yield语句后面那个表达式的值；done属性是一个布尔值，表示是否遍历结束。

ES6没有规定，function关键字与函数名之间的星号，写在哪个位置。这导致下面的写法都能通过。

function \* foo(x, y) { ··· }

function \*foo(x, y) { ··· }

function\* foo(x, y) { ··· }

function\*foo(x, y) { ··· }

由于Generator函数仍然是普通函数，所以一般的写法是上面的第三种，即星号紧跟在function关键字后面。本书也采用这种写法。

### Yield语句

由于Generator函数返回的遍历器对象，只有调用next方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。yield语句就是暂停标志。

遍历器对象的next方法的运行逻辑如下。

（1）遇到yield语句，就暂停执行后面的操作，并将紧跟在yield后面的那个表达式的值，作为返回的对象的value属性值。

（2）下一次调用next方法时，再继续往下执行，直到遇到下一个yield语句。

（3）如果没有再遇到新的yield语句，就一直运行到函数结束，直到return语句为止，并将return语句后面的表达式的值，作为返回的对象的value属性值。

（4）如果该函数没有return语句，则返回的对象的value属性值为undefined。

需要注意的是，yield语句后面的表达式，只有当调用next方法、内部指针指向该语句时才会执行，因此等于为JavaScript提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

function\* gen() {

yield 123 + 456;

}

上面代码中，yield后面的表达式123 + 456，不会立即求值，只会在next方法将指针移到这一句时，才会求值。

yield语句与return语句既有相似之处，也有区别。相似之处在于，都能返回紧跟在语句后面的那个表达式的值。区别在于每次遇到yield，函数暂停执行，下一次再从该位置继续向后执行，而return语句不具备位置记忆的功能。一个函数里面，只能执行一次（或者说一个）return语句，但是可以执行多次（或者说多个）yield语句。正常函数只能返回一个值，因为只能执行一次return；Generator函数可以返回一系列的值，因为可以有任意多个yield。从另一个角度看，也可以说Generator生成了一系列的值，这也就是它的名称的来历（在英语中，generator这个词是“生成器”的意思）。

Generator函数可以不用yield语句，这时就变成了一个单纯的暂缓执行函数。

function\* f() {

console.log('执行了！')

}

var generator = f();

setTimeout(function () {

generator.next()

}, 2000);

上面代码中，函数f如果是普通函数，在为变量generator赋值时就会执行。但是，函数f是一个Generator函数，就变成只有调用next方法时，函数f才会执行。

另外需要注意，yield语句不能用在普通函数中，否则会报错。

(function (){

yield 1;

})()

// SyntaxError: Unexpected number

上面代码在一个普通函数中使用yield语句，结果产生一个句法错误。

下面是另外一个例子。

var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function\* (a) {

a.forEach(function (item) {

if (typeof item !== 'number') {

yield\* flat(item);

} else {

yield item;

}

}

};

for (var f of flat(arr)){

console.log(f);

}

上面代码也会产生句法错误，因为forEach方法的参数是一个普通函数，但是在里面使用了yield语句（这个函数里面还使用了yield\*语句，这里可以不用理会，详细说明见后文）。一种修改方法是改用for循环。

var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function\* (a) {

var length = a.length;

for (var i = 0; i < length; i++) {

var item = a[i];

if (typeof item !== 'number') {

yield\* flat(item);

} else {

yield item;

}

}

};

for (var f of flat(arr)) {

console.log(f);

}

// 1, 2, 3, 4, 5, 6

另外，yield语句如果用在一个表达式之中，必须放在圆括号里面。

console.log('Hello' + yield); // SyntaxError

console.log('Hello' + yield 123); // SyntaxError

console.log('Hello' + (yield)); // OK

console.log('Hello' + (yield 123)); // OK

yield语句用作函数参数或赋值表达式的右边，可以不加括号。

foo(yield 'a', yield 'b'); // OK

let input = yield; // OK

### 与Iterator接口的关系

上一章说过，任意一个对象的Symbol.iterator方法，等于该对象的遍历器生成函数，调用该函数会返回该对象的一个遍历器对象。

由于Generator函数就是遍历器生成函数，因此可以把Generator赋值给对象的Symbol.iterator属性，从而使得该对象具有Iterator接口。

var myIterable = {};

myIterable[Symbol.iterator] = function\* () {

yield 1;

yield 2;

yield 3;

};

[...myIterable] // [1, 2, 3]

上面代码中，Generator函数赋值给Symbol.iterator属性，从而使得myIterable对象具有了Iterator接口，可以被...运算符遍历了。

Generator函数执行后，返回一个遍历器对象。该对象本身也具有Symbol.iterator属性，执行后返回自身

function\* gen(){

// some code

}

var g = gen();

g[Symbol.iterator]() === g

// true

上面代码中，gen是一个Generator函数，调用它会生成一个遍历器对象g。它的Symbol.iterator属性，也是一个遍历器对象生成函数，执行后返回它自己。

### Next方法的参数

yield句本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield语句的返回值。

function\* f() {

for(var i=0; true; i++) {

var reset = yield i;

if(reset) { i = -1; }

}

}

var g = f();

g.next() // { value: 0, done: false }

g.next() // { value: 1, done: false }

g.next(true) // { value: 0, done: false }

上面代码先定义了一个可以无限运行的Generator函数f，如果next方法没有参数，每次运行到yield语句，变量reset的值总是undefined。当next方法带一个参数true时，当前的变量reset就被重置为这个参数（即true），因此i会等于-1，下一轮循环就会从-1开始递增。

这个功能有很重要的语法意义。Generator函数从暂停状态到恢复运行，它的上下文状态（context）是不变的。通过next方法的参数，就有办法在Generator函数开始运行之后，继续向函数体内部注入值。也就是说，可以在Generator函数运行的不同阶段，从外部向内部注入不同的值，从而调整函数行为

再看一个例子。

function\* foo(x) {

var y = 2 \* (yield (x + 1));

var z = yield (y / 3);

return (x + y + z);

}

var a = foo(5);

a.next() // Object{value:6, done:false}

a.next() // Object{value:NaN, done:false}

a.next() // Object{value:NaN, done:true}

var b = foo(5);

b.next() // { value:6, done:false }

b.next(12) // { value:8, done:false }

b.next(13) // { value:42, done:true }

上面代码中，第二次运行next方法的时候不带参数，导致y的值等于2 \* undefined（即NaN），除以3以后还是NaN，因此返回对象的value属性也等于NaN。第三次运行Next方法的时候不带参数，所以z等于undefined，返回对象的value属性等于5 + NaN + undefined，即NaN。

如果向next方法提供参数，返回结果就完全不一样了。上面代码第一次调用b的next方法时，返回x+1的值6；第二次调用next方法，将上一次yield语句的值设为12，因此y等于24，返回y / 3的值8；第三次调用next方法，将上一次yield语句的值设为13，因此z等于13，这时x等于5，y等于24，所以return语句的值等于42。

注意，由于next方法的参数表示上一个yield语句的返回值，所以第一次使用next方法时，不能带有参数。V8引擎直接忽略第一次使用next方法时的参数，只有从第二次使用next方法开始，参数才是有效的。从语义上讲，第一个next方法用来启动遍历器对象，所以不用带有参数。

如果想要第一次调用next方法时，就能够输入值，可以在Generator函数外面再包一层。

function wrapper(generatorFunction) {

return function (...args) {

let generatorObject = generatorFunction(...args);

generatorObject.next();

return generatorObject;

};

}

const wrapped = wrapper(function\* () {

console.log(`First input: ${yield}`);

return 'DONE';

});

wrapped().next('hello!')

// First input: hello!

上面代码中，Generator函数如果不用wrapper先包一层，是无法第一次调用next方法，就输入参数的。

再看一个通过next方法的参数，向Generator函数内部输入值的例子。

function\* dataConsumer() {

console.log('Started');

console.log(`1. ${yield}`);

console.log(`2. ${yield}`);

return 'result';

}

let genObj = dataConsumer();

genObj.next();

// Started

genObj.next('a')

// 1. a

genObj.next('b')

// 2. b

上面代码是一个很直观的例子，每次通过next方法向Generator函数输入值，然后打印出来。

### for...of循环

for...of循环可以自动遍历Generator函数时生成的Iterator对象，且此时不再需要调用next方法。

function \*foo() {

yield 1;

yield 2;

yield 3;

yield 4;

yield 5;

return 6;

}

for (let v of foo()) {

console.log(v);

}

// 1 2 3 4 5

上面代码使用for...of循环，依次显示5个yield语句的值。这里需要注意，一旦next方法的返回对象的done属性为true，for...of循环就会中止，且不包含该返回对象，所以上面代码的return语句返回的6，不包括在for...of循环之中。

下面是一个利用Generator函数和for...of循环，实现斐波那契数列的例子。

function\* fibonacci() {

let [prev, curr] = [0, 1];

for (;;) {

[prev, curr] = [curr, prev + curr];

yield curr;

}

}

for (let n of fibonacci()) {

if (n > 1000) break;

console.log(n);

}

从上面代码可见，使用for...of语句时不需要使用next方法。

利用for...of循环，可以写出遍历任意对象（object）的方法。原生的JavaScript对象没有遍历接口，无法使用for...of循环，通过Generator函数为它加上这个接口，就可以用了。

function\* objectEntries(obj) {

let propKeys = Reflect.ownKeys(obj);

for (let propKey of propKeys) {

yield [propKey, obj[propKey]];

}

}

let jane = { first: 'Jane', last: 'Doe' };

for (let [key, value] of objectEntries(jane)) {

console.log(`${key}: ${value}`);

}

// first: Jane

// last: Doe

上面代码中，对象jane原生不具备Iterator接口，无法用for...of遍历。这时，我们通过Generator函数objectEntries为它加上遍历器接口，就可以用for...of遍历了。加上遍历器接口的另一种写法是，将Generator函数加到对象的Symbol.iterator属性上面。

function\* objectEntries() {

let propKeys = Object.keys(this);

for (let propKey of propKeys) {

yield [propKey, this[propKey]];

}

}

let jane = { first: 'Jane', last: 'Doe' };

jane[Symbol.iterator] = objectEntries;

for (let [key, value] of jane) {

console.log(`${key}: ${value}`);

}

// first: Jane

// last: Doe

除了for...of循环以外，扩展运算符（...）、解构赋值和Array.from方法内部调用的，都是遍历器接口。这意味着，它们都可以将Generator函数返回的Iterator对象，作为参数。

function\* numbers () {

yield 1

yield 2

return 3

yield 4

}

// 扩展运算符

[...numbers()] // [1, 2]

// Array.from 方法

Array.from(numbers()) // [1, 2]

// 解构赋值

let [x, y] = numbers();

x // 1

y // 2

// for...of 循环

for (let n of numbers()) {

console.log(n)

}

// 1

// 2

### Generator.prototype.throw()

Generator函数返回的遍历器对象，都有一个throw方法，可以在函数体外抛出错误，然后在Generator函数体内捕获。

var g = function\* () {

try {

yield;

} catch (e) {

console.log('内部捕获', e);

}

};

var i = g();

i.next();

try {

i.throw('a');

i.throw('b');

} catch (e) {

console.log('外部捕获', e);

}

// 内部捕获 a

// 外部捕获 b

上面代码中，遍历器对象i连续抛出两个错误。第一个错误被Generator函数体内的catch语句捕获。i第二次抛出错误，由于Generator函数内部的catch语句已经执行过了，不会再捕捉到这个错误了，所以这个错误就被抛出了Generator函数体，被函数体外的catch语句捕获。

throw方法可以接受一个参数，该参数会被catch语句接收，建议抛出Error对象的实例。

var g = function\* () {

try {

yield;

} catch (e) {

console.log(e);

}

};

var i = g();

i.next();

i.throw(new Error('出错了！'));

// Error: 出错了！(…)

注意，不要混淆遍历器对象的throw方法和全局的throw命令。上面代码的错误，是用遍历器对象的throw方法抛出的，而不是用throw命令抛出的。后者只能被函数体外的catch语句捕获。

var g = function\* () {

while (true) {

try {

yield;

} catch (e) {

if (e != 'a') throw e;

console.log('内部捕获', e);

}

}

};

var i = g();

i.next();

try {

throw new Error('a');

throw new Error('b');

} catch (e) {

console.log('外部捕获', e);

}

// 外部捕获 [Error: a]

上面代码之所以只捕获了a，是因为函数体外的catch语句块，捕获了抛出的a错误以后，就不会再继续try代码块里面剩余的语句了。

如果Generator函数内部没有部署try...catch代码块，那么throw方法抛出的错误，将被外部try...catch代码块捕获。

var g = function\* () {

while (true) {

yield;

console.log('内部捕获', e);

}

};

var i = g();

i.next();

try {

i.throw('a');

i.throw('b');

} catch (e) {

console.log('外部捕获', e);

}

// 外部捕获 a

上面代码中，Generator函数g内部没有部署try...catch代码块，所以抛出的错误直接被外部catch代码块捕获。

如果Generator函数内部和外部，都没有部署try...catch代码块，那么程序将报错，直接中断执行。

var gen = function\* gen(){

yield console.log('hello');

yield console.log('world');

}

var g = gen();

g.next();

g.throw();

// hello

// Uncaught undefined

上面代码中，g.throw抛出错误以后，没有任何try...catch代码块可以捕获这个错误，导致程序报错，中断执行。

throw方法被捕获以后，会附带执行下一条yield语句。也就是说，会附带执行一次next方法。

var gen = function\* gen(){

try {

yield console.log('a');

} catch (e) {

// ...

}

yield console.log('b');

yield console.log('c');

}

var g = gen();

g.next() // a

g.throw() // b

g.next() // c

上面代码中，g.throw方法被捕获以后，自动执行了一次next方法，所以会打印b。另外，也可以看到，只要Generator函数内部部署了try...catch代码块，那么遍历器的throw方法抛出的错误，不影响下一次遍历。

另外，throw命令与g.throw方法是无关的，两者互不影响。

var gen = function\* gen(){

yield console.log('hello');

yield console.log('world');

}

var g = gen();

g.next();

try {

throw new Error();

} catch (e) {

g.next();

}

// hello

// world

上面代码中，throw命令抛出的错误不会影响到遍历器的状态，所以两次执行next方法，都进行了正确的操作。

这种函数体内捕获错误的机制，大大方便了对错误的处理。多个yield语句，可以只用一个try...catch代码块来捕获错误。如果使用回调函数的写法，想要捕获多个错误，就不得不为每个函数内部写一个错误处理语句，现在只在Generator函数内部写一次catch语句就可以了。

Generator函数体外抛出的错误，可以在函数体内捕获；反过来，Generator函数体内抛出的错误，也可以被函数体外的catch捕获。

function \*foo() {

var x = yield 3;

var y = x.toUpperCase();

yield y;

}

var it = foo();

it.next(); // { value:3, done:false }

try {

it.next(42);

} catch (err) {

console.log(err);

}

上面代码中，第二个next方法向函数体内传入一个参数42，数值是没有toUpperCase方法的，所以会抛出一个TypeError错误，被函数体外的catch捕获。

一旦Generator执行过程中抛出错误，且没有被内部捕获，就不会再执行下去了。如果此后还调用next方法，将返回一个value属性等于undefined、done属性等于true的对象，即JavaScript引擎认为这个Generator已经运行结束了。

function\* g() {

yield 1;

console.log('throwing an exception');

throw new Error('generator broke!');

yield 2;

yield 3;

}

function log(generator) {

var v;

console.log('starting generator');

try {

v = generator.next();

console.log('第一次运行next方法', v);

} catch (err) {

console.log('捕捉错误', v);

}

try {

v = generator.next();

console.log('第二次运行next方法', v);

} catch (err) {

console.log('捕捉错误', v);

}

try {

v = generator.next();

console.log('第三次运行next方法', v);

} catch (err) {

console.log('捕捉错误', v);

}

console.log('caller done');

}

log(g());

// starting generator

// 第一次运行next方法 { value: 1, done: false }

// throwing an exception

// 捕捉错误 { value: 1, done: false }

// 第三次运行next方法 { value: undefined, done: true }

// caller done

上面代码一共三次运行next方法，第二次运行的时候会抛出错误，然后第三次运行的时候，Generator函数就已经结束了，不再执行下去了。

### Generator.prototype.return()

Generator函数返回的遍历器对象，还有一个return方法，可以返回给定的值，并且终结遍历Generator函数。

function\* gen() {

yield 1;

yield 2;

yield 3;

}

var g = gen();

g.next() // { value: 1, done: false }

g.return('foo') // { value: "foo", done: true }

g.next() // { value: undefined, done: true }

上面代码中，遍历器对象g调用return方法后，返回值的value属性就是return方法的参数foo。并且，Generator函数的遍历就终止了，返回值的done属性为true，以后再调用next方法，done属性总是返回true。

如果return方法调用时，不提供参数，则返回值的value属性为undefined。

function\* gen() {

yield 1;

yield 2;

yield 3;

}

var g = gen();

g.next() // { value: 1, done: false }

g.return() // { value: undefined, done: true }

如果Generator函数内部有try...finally代码块，那么return方法会推迟到finally代码块执行完再执行。

function\* numbers () {

yield 1;

try {

yield 2;

yield 3;

} finally {

yield 4;

yield 5;

}

yield 6;

}

var g = numbers()

g.next() // { done: false, value: 1 }

g.next() // { done: false, value: 2 }

g.return(7) // { done: false, value: 4 }

g.next() // { done: false, value: 5 }

g.next() // { done: true, value: 7 }

上面代码中，调用return方法后，就开始执行finally代码块，然后等到finally代码块执行完，再执行return方法。

### Yield\*语句

如果在Generater函数内部，调用另一个Generator函数，默认情况下是没有效果的。

function\* foo() {

yield 'a';

yield 'b';

}

function\* bar() {

yield 'x';

foo();

yield 'y';

}

for (let v of bar()){

console.log(v);

}

// "x"

// "y"

上面代码中，foo和bar都是Generator函数，在bar里面调用foo，是不会有效果的。

这个就需要用到yield\*语句，用来在一个Generator函数里面执行另一个Generator函数。

function\* bar() {

yield 'x';

yield\* foo();

yield 'y';

}

// 等同于

function\* bar() {

yield 'x';

yield 'a';

yield 'b';

yield 'y';

}

// 等同于

function\* bar() {

yield 'x';

for (let v of foo()) {

yield v;

}

yield 'y';

}

for (let v of bar()){

console.log(v);

}

// "x"

// "a"

// "b"

// "y"

再来看一个对比的例子

function\* inner() {

yield 'hello!';

}

function\* outer1() {

yield 'open';

yield inner();

yield 'close';

}

var gen = outer1()

gen.next().value // "open"

gen.next().value // 返回一个遍历器对象

gen.next().value // "close"

function\* outer2() {

yield 'open'

yield\* inner()

yield 'close'

}

var gen = outer2()

gen.next().value // "open"

gen.next().value // "hello!"

gen.next().value // "close"

上面例子中，outer2使用了yield\*，outer1没使用。结果就是，outer1返回一个遍历器对象，outer2返回该遍历器对象的内部值。

从语法角度看，如果yield命令后面跟的是一个遍历器对象，需要在yield命令后面加上星号，表明它返回的是一个遍历器对象。这被称为yield\*语句。

let delegatedIterator = (function\* () {

yield 'Hello!';

yield 'Bye!';

}());

let delegatingIterator = (function\* () {

yield 'Greetings!';

yield\* delegatedIterator;

yield 'Ok, bye.';

}());

for(let value of delegatingIterator) {

console.log(value);

}

// "Greetings!

// "Hello!"

// "Bye!"

// "Ok, bye."

上面代码中，delegatingIterator是代理者，delegatedIterator是被代理者。由于yield\* delegatedIterator语句得到的值，是一个遍历器，所以要用星号表示。运行结果就是使用一个遍历器，遍历了多个Generator函数，有递归的效果。

yield\*后面的Generator函数（没有return语句时），等同于在Generator函数内部，部署一个for...of循环。

function\* concat(iter1, iter2) {

yield\* iter1;

yield\* iter2;

}

// 等同于

function\* concat(iter1, iter2) {

for (var value of iter1) {

yield value;

}

for (var value of iter2) {

yield value;

}

}

上面代码说明，yield\*后面的Generator函数（没有return语句时），不过是for...of的一种简写形式，完全可以用后者替代前者。反之，则需要用var value = yield\* iterator的形式获取return语句的值。

如果yield\*后面跟着一个数组，由于数组原生支持遍历器，因此就会遍历数组成员。

function\* gen(){

yield\* ["a", "b", "c"];

}

gen().next() // { value:"a", done:false }

上面代码中，yield命令后面如果不加星号，返回的是整个数组，加了星号就表示返回的是数组的遍历器对象。

实际上，任何数据结构只要有Iterator接口，就可以被yield\*遍历。

let read = (function\* () {

yield 'hello';

yield\* 'hello';

})();

read.next().value // "hello"

read.next().value // "h"

上面代码中，yield语句返回整个字符串，yield\*语句返回单个字符。因为字符串具有Iterator接口，所以被yield\*遍历。

如果被代理的Generator函数有return语句，那么就可以向代理它的Generator函数返回数据。

function \*foo() {

yield 2;

yield 3;

return "foo";

}

function \*bar() {

yield 1;

var v = yield \*foo();

console.log( "v: " + v );

yield 4;

}

var it = bar();

it.next()

// {value: 1, done: false}

it.next()

// {value: 2, done: false}

it.next()

// {value: 3, done: false}

it.next();

// "v: foo"

// {value: 4, done: false}

it.next()

// {value: undefined, done: true}

上面代码在第四次调用next方法的时候，屏幕上会有输出，这是因为函数foo的return语句，向函数bar提供了返回值。

再看一个例子。

function\* genFuncWithReturn() {

yield 'a';

yield 'b';

return 'The result';

}

function\* logReturned(genObj) {

let result = yield\* genObj;

console.log(result);

}

[...logReturned(genFuncWithReturn())]

// The result

// 值为 [ 'a', 'b' ]

上面代码中，存在两次遍历。第一次是扩展运算符遍历函数logReturned返回的遍历器对象，第二次是yield\*语句遍历函数genFuncWithReturn返回的遍历器对象。这两次遍历的效果是叠加的，最终表现为扩展运算符遍历函数genFuncWithReturn返回的遍历器对象。所以，最后的数据表达式得到的值等于[ 'a', 'b' ]。但是，函数genFuncWithReturn的return语句的返回值The result，会返回给函数logReturned内部的result变量，因此会有终端输出。

yield\*命令可以很方便地取出嵌套数组的所有成员。

function\* iterTree(tree) {

if (Array.isArray(tree)) {

for(let i=0; i < tree.length; i++) {

yield\* iterTree(tree[i]);

}

} else {

yield tree;

}

}

const tree = [ 'a', ['b', 'c'], ['d', 'e'] ];

for(let x of iterTree(tree)) {

console.log(x);

}

// a

// b

// c

// d

// e

下面是一个稍微复杂的例子，使用yield\*语句遍历完全二叉树

// 下面是二叉树的构造函数，

// 三个参数分别是左树、当前节点和右树

function Tree(left, label, right) {

this.left = left;

this.label = label;

this.right = right;

}

// 下面是中序（inorder）遍历函数。

// 由于返回的是一个遍历器，所以要用generator函数。

// 函数体内采用递归算法，所以左树和右树要用yield\*遍历

function\* inorder(t) {

if (t) {

yield\* inorder(t.left);

yield t.label;

yield\* inorder(t.right);

}

}

// 下面生成二叉树

function make(array) {

// 判断是否为叶节点

if (array.length == 1) return new Tree(null, array[0], null);

return new Tree(make(array[0]), array[1], make(array[2]));

}

let tree = make([[['a'], 'b', ['c']], 'd', [['e'], 'f', ['g']]]);

// 遍历二叉树

var result = [];

for (let node of inorder(tree)) {

result.push(node);

}

result

// ['a', 'b', 'c', 'd', 'e', 'f', 'g']

### 作为对象属性的Generator函数

如果一个对象的属性是Generator函数，可以简写成下面的形式。

let obj = {

\* myGeneratorMethod() {

···

}

};

上面代码中，myGeneratorMethod属性前面有一个星号，表示这个属性是一个Generator函数。

它的完整形式如下，与上面的写法是等价的。

let obj = {

myGeneratorMethod: function\* () {

// ···

}

};

### Generator 函数的this

Generator函数总是返回一个遍历器，ES6规定这个遍历器是Generator函数的实例，也继承了Generator函数的prototype对象上的方法。

function\* g() {}

g.prototype.hello = function () {

return 'hi!';

};

let obj = g();

obj instanceof g // true

obj.hello() // 'hi!'

上面代码表名，Generator函数 g 返回的遍历器obj ，是 g 的实例，而且继承了g.prototype。

但是如果把g 当作普通的构造函数，并不会生效，因为 g 返回的总是遍历器对象，而不是this对象

function\* g() {

this.a = 11;

}

let obj = g();

obj.a // undefined

上面代码中，Generator函数g在this对象上面添加了一个属性a，但是obj对象拿不到这个属性。

Generator函数也不能跟new命令一起用，会报错。

function\* F() {

yield this.x = 2;

yield this.y = 3;

}

new F()

// TypeError: F is not a constructor

上面代码中，new命令跟构造函数F一起使用，结果报错，因为F不是构造函数。

那么，有没有办法让Generator函数返回一个正常的对象实例，既可以用next方法，又可以获得正常的this？

下面是一个变通方法。首先，生成一个空对象，使用call方法绑定Generator函数内部的this。这样，构造函数调用以后，这个空对象就是Generator函数的实例对象了。

function\* F() {

this.a = 1;

yield this.b = 2;

yield this.c = 3;

}

var obj = {};

var f = F.call(obj);

f.next(); // Object {value: 2, done: false}

f.next(); // Object {value: 3, done: false}

f.next(); // Object {value: undefined, done: true}

obj.a // 1

obj.b // 2

obj.c // 3

上面代码中，首先是F内部的this对象绑定obj对象，然后调用它，返回一个Iterator对象。这个对象执行三次next方法（因为F内部有两个yield语句），完成F内部所有代码的运行。这时，所有内部属性都绑定在obj对象上了，因此obj对象也就成了F的实例。

上面代码中，执行的是遍历器对象f，但是生成的对象实例是obj，有没有办法将这两个对象统一呢？

一个办法就是将obj换成F.prototype。

function\* F() {

this.a = 1;

yield this.b = 2;

yield this.c = 3;

}

var f = F.call(F.prototype);

f.next(); // Object {value: 2, done: false}

f.next(); // Object {value: 3, done: false}

f.next(); // Object {value: undefined, done: true}

f.a // 1

f.b // 2

f.c // 3

再将F改成构造函数，就可以对它执行new命令了。

function\* gen() {

this.a = 1;

yield this.b = 2;

yield this.c = 3;

}

function F() {

return gen.call(gen.prototype);

}

var f = new F();

f.next(); // Object {value: 2, done: false}

f.next(); // Object {value: 3, done: false}

f.next(); // Object {value: undefined, done: true}

f.a // 1

f.b // 2

f.c // 3

### 含义

#### Generator与状态机

Generator是实现状态机的最佳结构。比如，下面的clock函数就是一个状态机。

var ticking = true;

var clock = function() {

if (ticking)

console.log('Tick!');

else

console.log('Tock!');

ticking = !ticking;

}

上面代码的clock函数一共有两种状态（Tick和Tock），每运行一次，就改变一次状态。这个函数如果用Generator实现，就是下面这样。

var clock = function\*() {

while (true) {

console.log('Tick!');

yield;

console.log('Tock!');

yield;

}

};

上面的Generator实现与ES5实现对比，可以看到少了用来保存状态的外部变量ticking，这样就更简洁，更安全（状态不会被非法篡改）、更符合函数式编程的思想，在写法上也更优雅。Generator之所以可以不用外部变量保存状态，是因为它本身就包含了一个状态信息，即目前是否处于暂停态。

#### Generator与协程

协程（coroutine）是一种程序运行的方式，可以理解成“协作的线程”或“协作的函数”。协程既可以用单线程实现，也可以用多线程实现。前者是一种特殊的子例程，后者是一种特殊的线程。

##### （1）协程与子例程的差异

传统的“子例程”（subroutine）采用堆栈式“后进先出”的执行方式，只有当调用的子函数完全执行完毕，才会结束执行父函数。协程与其不同，多个线程（单线程情况下，即多个函数）可以并行执行，但是只有一个线程（或函数）处于正在运行的状态，其他线程（或函数）都处于暂停态（suspended），线程（或函数）之间可以交换执行权。也就是说，一个线程（或函数）执行到一半，可以暂停执行，将执行权交给另一个线程（或函数），等到稍后收回执行权的时候，再恢复执行。这种可以并行执行、交换执行权的线程（或函数），就称为协程。

从实现上看，在内存中，子例程只使用一个栈（stack），而协程是同时存在多个栈，但只有一个栈是在运行状态，也就是说，协程是以多占用内存为代价，实现多任务的并行。

##### （2）协程与普通线程的差异

不难看出，协程适合用于多任务运行的环境。在这个意义上，它与普通的线程很相似，都有自己的执行上下文、可以分享全局变量。它们的不同之处在于，同一时间可以有多个线程处于运行状态，但是运行的协程只能有一个，其他协程都处于暂停状态。此外，普通的线程是抢先式的，到底哪个线程优先得到资源，必须由运行环境决定，但是协程是合作式的，执行权由协程自己分配。

由于ECMAScript是单线程语言，只能保持一个调用栈。引入协程以后，每个任务可以保持自己的调用栈。这样做的最大好处，就是抛出错误的时候，可以找到原始的调用栈。不至于像异步操作的回调函数那样，一旦出错，原始的调用栈早就结束。

Generator函数是ECMAScript 6对协程的实现，但属于不完全实现。Generator函数被称为“半协程”（semi-coroutine），意思是只有Generator函数的调用者，才能将程序的执行权还给Generator函数。如果是完全执行的协程，任何函数都可以让暂停的协程继续执行。

如果将Generator函数当作协程，完全可以将多个需要互相协作的任务写成Generator函数，它们之间使用yield语句交换控制权。

#### 应用

Generator可以暂停函数执行，返回任意表达式的值。这种特点使得Generator有多种应用场景。

##### （1）异步操作的同步化表达

Generator函数的暂停执行的效果，意味着可以把异步操作写在yield语句里面，等到调用next方法时再往后执行。这实际上等同于不需要写回调函数了，因为异步操作的后续操作可以放在yield语句下面，反正要等到调用next方法时再执行。所以，Generator函数的一个重要实际意义就是用来处理异步操作，改写回调函数。

function\* loadUI() {

showLoadingScreen();

yield loadUIDataAsynchronously();

hideLoadingScreen();

}

var loader = loadUI();

// 加载UI

loader.next()

// 卸载UI

loader.next()

上面代码表示，第一次调用loadUI函数时，该函数不会执行，仅返回一个遍历器。下一次对该遍历器调用next方法，则会显示Loading界面，并且异步加载数据。等到数据加载完成，再一次使用next方法，则会隐藏Loading界面。可以看到，这种写法的好处是所有Loading界面的逻辑，都被封装在一个函数，按部就班非常清晰。

Ajax是典型的异步操作，通过Generator函数部署Ajax操作，可以用同步的方式表达。

function\* main() {

var result = yield request("http://some.url");

var resp = JSON.parse(result);

console.log(resp.value);

}

function request(url) {

makeAjaxCall(url, function(response){

it.next(response);

});

}

var it = main();

it.next();

上面代码的main函数，就是通过Ajax操作获取数据。可以看到，除了多了一个yield，它几乎与同步操作的写法完全一样。注意，makeAjaxCall函数中的next方法，必须加上response参数，因为yield语句构成的表达式，本身是没有值的，总是等于undefined。

下面是另一个例子，通过Generator函数逐行读取文本文件。

function\* numbers() {

let file = new FileReader("numbers.txt");

try {

while(!file.eof) {

yield parseInt(file.readLine(), 10);

}

} finally {

file.close();

}

}

上面代码打开文本文件，使用yield语句可以手动逐行读取文件。

##### （2）控制流管理

如果有一个多步操作非常耗时，采用回调函数，可能会写成下面这样。

step1(function (value1) {

step2(value1, function(value2) {

step3(value2, function(value3) {

step4(value3, function(value4) {

// Do something with value4

});

});

});

});

采用Promise改写上面的代码。

Promise.resolve(step1)

.then(step2)

.then(step3)

.then(step4)

.then(function (value4) {

// Do something with value4

}, function (error) {

// Handle any error from step1 through step4

})

.done();

上面代码已经把回调函数，改成了直线执行的形式，但是加入了大量Promise的语法。Generator函数可以进一步改善代码运行流程。

function\* longRunningTask(value1) {

try {

var value2 = yield step1(value1);

var value3 = yield step2(value2);

var value4 = yield step3(value3);

var value5 = yield step4(value4);

// Do something with value4

} catch (e) {

// Handle any error from step1 through step4

}

}

然后，使用一个函数，按次序自动执行所有步骤。

scheduler(longRunningTask(initialValue));

function scheduler(task) {

var taskObj = task.next(task.value);

// 如果Generator函数未结束，就继续调用

if (!taskObj.done) {

task.value = taskObj.value

scheduler(task);

}

}

注意，上面这种做法，只适合同步操作，即所有的task都必须是同步的，不能有异步操作。因为这里的代码一得到返回值，就继续往下执行，没有判断异步操作何时完成。如果要控制异步的操作流程，详见后面的《异步操作》一章。

下面，利用for...of循环会自动依次执行yield命令的特性，提供一种更一般的控制流管理的方法。

let steps = [step1Func, step2Func, step3Func];

function \*iterateSteps(steps){

for (var i=0; i< steps.length; i++){

var step = steps[i];

yield step();

}

}

上面代码中，数组steps封装了一个任务的多个步骤，Generator函数iterateSteps则是依次为这些步骤加上yield命令。

将任务分解成步骤之后，还可以将项目分解成多个依次执行的任务。

let jobs = [job1, job2, job3];

function \*iterateJobs(jobs){

for (var i=0; i< jobs.length; i++){

var job = jobs[i];

yield \*iterateSteps(job.steps);

}

}

上面代码中，数组jobs封装了一个项目的多个任务，Generator函数iterateJobs则是依次为这些任务加上yield \*命令。

最后，就可以用for...of循环一次性依次执行所有任务的所有步骤。

for (var step of iterateJobs(jobs)){

console.log(step.id);

}

再次提醒，上面的做法只能用于所有步骤都是同步操作的情况，不能有异步操作的步骤。如果想要依次执行异步的步骤，必须使用后面的《异步操作》一章介绍的方法。

for...of的本质是一个while循环，所以上面的代码实质上执行的是下面的逻辑。

var it = iterateJobs(jobs);

var res = it.next();

while (!res.done){

var result = res.value;

// ...

res = it.next();

}

##### （3）部署Iterator接口

利用Generator函数，可以在任意对象上部署Iterator接口。

function\* iterEntries(obj) {

let keys = Object.keys(obj);

for (let i=0; i < keys.length; i++) {

let key = keys[i];

yield [key, obj[key]];

}

}

let myObj = { foo: 3, bar: 7 };

for (let [key, value] of iterEntries(myObj)) {

console.log(key, value);

}

// foo 3

// bar 7

上述代码中，myObj是一个普通对象，通过iterEntries函数，就有了Iterator接口。也就是说，可以在任意对象上部署next方法。

下面是一个对数组部署Iterator接口的例子，尽管数组原生具有这个接口。

function\* makeSimpleGenerator(array){

var nextIndex = 0;

while(nextIndex < array.length){

yield array[nextIndex++];

}

}

var gen = makeSimpleGenerator(['yo', 'ya']);

gen.next().value // 'yo'

gen.next().value // 'ya'

gen.next().done // true

##### （4）作为数据结构

Generator可以看作是数据结构，更确切地说，可以看作是一个数组结构，因为Generator函数可以返回一系列的值，这意味着它可以对任意表达式，提供类似数组的接口。

function \*doStuff() {

yield fs.readFile.bind(null, 'hello.txt');

yield fs.readFile.bind(null, 'world.txt');

yield fs.readFile.bind(null, 'and-such.txt');

}

上面代码就是依次返回三个函数，但是由于使用了Generator函数，导致可以像处理数组那样，处理这三个返回的函数。

for (task of doStuff()) {

// task是一个函数，可以像回调函数那样使用它

}

实际上，如果用ES5表达，完全可以用数组模拟Generator的这种用法。

function doStuff() {

return [

fs.readFile.bind(null, 'hello.txt'),

fs.readFile.bind(null, 'world.txt'),

fs.readFile.bind(null, 'and-such.txt')

];

}

上面的函数，可以用一模一样的for...of循环处理！两相一比较，就不难看出Generator使得数据或者操作，具备了类似数组的接口。

## Promise对象

### Promise 的含义

Promise是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6将其写进了语言标准，统一了用法，原生提供了Promise对象。

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise是一个对象，从它可以获取异步操作的消息。Promise提供统一的API，各种异步操作都可以用同样的方法进行处理。

Promise对象有以下两个特点。

（1）对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：Pending（进行中）、Resolved（已完成，又称Fulfilled）和Rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

（2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从Pending变为Resolved和从Pending变为Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三，当处于Pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

如果某些事件不断地反复发生，一般来说，使用stream模式是比部署Promise更好的选择。

### 基础用法

ES6规定，Promise对象是一个构造函数，用来生成Promise实例。

下面代码创建了一个promise实例

var promise = new Promise(function(resolve, reject) {

// ... some code

if (/\* 异步操作成功 \*/){

resolve(value);

} else {

reject(error);

}

});

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，不用自己部署。

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对象的状态从“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。

Promise实例生成以后，可以用then方法分别指定Resolved状态和Reject状态的回调函数。

promise.then(function(value) {

// success

}, function(error) {

// failure

});

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为Resolved时调用，第二个回调函数是Promise对象的状态变为Reject时调用。其中，第二个函数是可选的，不一定要提供。这两个函数都接受Promise对象传出的值作为参数。

下面是一个Promise对象的简单例子。

function timeout(ms) {

return new Promise((resolve, reject) => {

setTimeout(resolve, ms, 'done');

});

}

timeout(100).then((value) => {

console.log(value);

});

上面代码中，timeout方法返回一个Promise实例，表示一段时间以后才会发生的结果。过了指定的时间（ms参数）以后，Promise实例的状态变为Resolved，就会触发then方法绑定的回调函数。

Promise新建后就会立即执行。

let promise = new Promise(function(resolve, reject) {

console.log('Promise');

resolve();

});

promise.then(function() {

console.log('Resolved.');

});

console.log('Hi!');

// Promise

// Hi!

// Resolved

上面代码中，Promise新建后立即执行，所以首先输出的是“Promise”。然后，then方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行，所以“Resolved”最后输出。

上面代码中，Promise新建后立即执行，所以首先输出的是“Promise”。然后，then方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行，所以“Resolved”最后输出。

下面是异步加载图片的例子。

function loadImageAsync(url) {

return new Promise(function(resolve, reject) {

var image = new Image();

image.onload = function() {

resolve(image);

};

image.onerror = function() {

reject(new Error('Could not load image at ' + url));

};

image.src = url;

});

}

上面代码中，使用Promise包装了一个图片加载的异步操作。如果加载成功，就调用resolve方法，否则就调用reject方法。

下面是一个用Promise对象实现的Ajax操作的例子。

var getJSON = function(url) {

var promise = new Promise(function(resolve, reject){

var client = new XMLHttpRequest();

client.open("GET", url);

client.onreadystatechange = handler;

client.responseType = "json";

client.setRequestHeader("Accept", "application/json");

client.send();

function handler() {

if (this.readyState !== 4) {

return;

}

if (this.status === 200) {

resolve(this.response);

} else {

reject(new Error(this.statusText));

}

};

});

return promise;

};

getJSON("/posts.json").then(function(json) {

console.log('Contents: ' + json);

}, function(error) {

console.error('出错了', error);

});

上面代码中，getJSON是对XMLHttpRequest对象的封装，用于发出一个针对JSON数据的HTTP请求，并且返回一个Promise对象。需要注意的是，在getJSON内部，resolve函数和reject函数调用时，都带有参数。

如果调用resolve函数和reject函数时带有参数，那么它们的参数会被传递给回调函数。reject函数的参数通常是Error对象的实例，表示抛出的错误；resolve函数的参数除了正常的值以外，还可能是另一个Promise实例，表示异步操作的结果有可能是一个值，也有可能是另一个异步操作，比如像下面这样。

var p1 = new Promise(function (resolve, reject) {

// ...

});

var p2 = new Promise(function (resolve, reject) {

// ...

resolve(p1);

})

上面代码中，p1和p2都是Promise的实例，但是p2的resolve方法将p1作为参数，即一个异步操作的结果是返回另一个异步操作。

注意，这时p1的状态就会传递给p2，也就是说，p1的状态决定了p2的状态。如果p1的状态是Pending，那么p2的回调函数就会等待p1的状态改变；如果p1的状态已经是Resolved或者Rejected，那么p2的回调函数将会立刻执行。

var p1 = new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('fail')), 3000)

})

var p2 = new Promise(function (resolve, reject) {

setTimeout(() => resolve(p1), 1000)

})

p2

.then(result => console.log(result))

.catch(error => console.log(error))

// Error: fail

上面代码中，p1是一个Promise，3秒之后变为rejected。p2的状态在1秒之后改变，resolve方法返回的是p1。此时，由于p2返回的是另一个Promise，所以后面的then语句都变成针对后者（p1）。又过了2秒，p1变为rejected，导致触发catch方法指定的回调函数。

### Promise.prototype.then()

Promise实例具有then方法，也就是说，then方法是定义在原型对象Promise.prototype上的。它的作用是为Promise实例添加状态改变时的回调函数。前面说过，then方法的第一个参数是Resolved状态的回调函数，第二个参数（可选）是Rejected状态的回调函数。

then方法返回的是一个新的Promise实例（注意，不是原来那个Promise实例）。因此可以采用链式写法，即then方法后面再调用另一个then方法。

getJSON("/posts.json").then(function(json) {

return json.post;

}).then(function(post) {

// ...

});

上面的代码使用then方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。

采用链式的then，可以指定一组按照次序调用的回调函数。这时，前一个回调函数，有可能返回的还是一个Promise对象（即有异步操作），这时后一个回调函数，就会等待该Promise对象的状态发生变化，才会被调用。

getJSON("/post/1.json").then(function(post) {

return getJSON(post.commentURL);

}).then(function funcA(comments) {

console.log("Resolved: ", comments);

}, function funcB(err){

console.log("Rejected: ", err);

});

上面代码中，第一个then方法指定的回调函数，返回的是另一个Promise对象。这时，第二个then方法指定的回调函数，就会等待这个新的Promise对象状态发生变化。如果变为Resolved，就调用funcA，如果状态变为Rejected，就调用funcB。

如果采用箭头函数，上面的代码可以写得更简洁。

getJSON("/post/1.json").then(

post => getJSON(post.commentURL)

).then(

comments => console.log("Resolved: ", comments),

err => console.log("Rejected: ", err)

);

### Promise.prototype.catch()

Promise.prototype.catch方法是.then(null, rejection)的别名，用于指定发生错误时的回调函数。

getJSON("/posts.json").then(function(posts) {

// ...

}).catch(function(error) {

// 处理 getJSON 和 前一个回调函数运行时发生的错误

console.log('发生错误！', error);

});

上面代码中，getJSON方法返回一个Promise对象，如果该对象状态变为Resolved，则会调用then方法指定的回调函数；如果异步操作抛出错误，状态就会变为Rejected，就会调用catch方法指定的回调函数，处理这个错误。另外，then方法指定的回调函数，如果运行中抛出错误，也会被catch方法捕获。

p.then((val) => console.log("fulfilled:", val))

.catch((err) => console.log("rejected:", err));

// 等同于

p.then((val) => console.log("fulfilled:", val))

.then(null, (err) => console.log("rejected:", err));

下面是一个例子。

var promise = new Promise(function(resolve, reject) {

throw new Error('test');

});

promise.catch(function(error) {

console.log(error);

});

// Error: test

上面代码中，promise抛出一个错误，就被catch方法指定的回调函数捕获。注意，上面的写法与下面两种写法是等价的。

// 写法一

var promise = new Promise(function(resolve, reject) {

try {

throw new Error('test');

} catch(e) {

reject(e);

}

});

promise.catch(function(error) {

console.log(error);

});

// 写法二

var promise = new Promise(function(resolve, reject) {

reject(new Error('test'));

});

promise.catch(function(error) {

console.log(error);

});

比较上面两种写法，可以发现reject方法的作用，等同于抛出错误。

如果Promise状态已经变成Resolved，再抛出错误是无效的。

var promise = new Promise(function(resolve, reject) {

resolve('ok');

throw new Error('test');

});

promise

.then(function(value) { console.log(value) })

.catch(function(error) { console.log(error) });

// ok

Promise对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个catch语句捕获。

getJSON("/post/1.json").then(function(post) {

return getJSON(post.commentURL);

}).then(function(comments) {

// some code

}).catch(function(error) {

// 处理前面三个Promise产生的错误

});

上面代码中，一共有三个Promise对象：一个由getJSON产生，两个由then产生。它们之中任何一个抛出的错误，都会被最后一个catch捕获。

一般来说，不要在then方法里面定义Reject状态的回调函数（即then的第二个参数），总是使用catch方法。

// bad

promise

.then(function(data) {

// success

}, function(err) {

// error

});

// good

promise

.then(function(data) { //cb

// success

})

.catch(function(err) {

// error

});

上面代码中，第二种写法要好于第一种写法，理由是第二种写法可以捕获前面then方法执行中的错误，也更接近同步的写法（try/catch）。因此，建议总是使用catch方法，而不使用then方法的第二个参数。

跟传统的try/catch代码块不同的是，如果没有使用catch方法指定错误处理的回调函数，Promise对象抛出的错误不会传递到外层代码，即不会有任何反应。

var someAsyncThing = function() {

return new Promise(function(resolve, reject) {

// 下面一行会报错，因为x没有声明

resolve(x + 2);

});

};

someAsyncThing().then(function() {

console.log('everything is great');

});

上面代码中，someAsyncThing函数产生的Promise对象会报错，但是由于没有指定catch方法，这个错误不会被捕获，也不会传递到外层代码，导致运行后没有任何输出。注意，Chrome浏览器不遵守这条规定，它会抛出错误“ReferenceError: x is not defined”。

var promise = new Promise(function(resolve, reject) {

resolve("ok");

setTimeout(function() { throw new Error('test') }, 0)

});

promise.then(function(value) { console.log(value) });

// ok

// Uncaught Error: test

上面代码中，Promise指定在下一轮“事件循环”再抛出错误，结果由于没有指定使用try...catch语句，就冒泡到最外层，成了未捕获的错误。因为此时，Promise的函数体已经运行结束了，所以这个错误是在Promise函数体外抛出的。

Node.js有一个unhandledRejection事件，专门监听未捕获的reject错误。

process.on('unhandledRejection', function (err, p) {

console.error(err.stack)

});

上面代码中，unhandledRejection事件的监听函数有两个参数，第一个是错误对象，第二个是报错的Promise实例，它可以用来了解发生错误的环境信息。。

需要注意的是，catch方法返回的还是一个Promise对象，因此后面还可以接着调用then方法。

var someAsyncThing = function() {

return new Promise(function(resolve, reject) {

// 下面一行会报错，因为x没有声明

resolve(x + 2);

});

};

someAsyncThing()

.catch(function(error) {

console.log('oh no', error);

})

.then(function() {

console.log('carry on');

});

// oh no [ReferenceError: x is not defined]

// carry on

上面代码运行完catch方法指定的回调函数，会接着运行后面那个then方法指定的回调函数。如果没有报错，则会跳过catch方法。

Promise.resolve()

.catch(function(error) {

console.log('oh no', error);

})

.then(function() {

console.log('carry on');

});

// carry on

上面的代码因为没有报错，跳过了catch方法，直接执行后面的then方法。此时，要是then方法里面报错，就与前面的catch无关了。

catch方法之中，还能再抛出错误。

var someAsyncThing = function() {

return new Promise(function(resolve, reject) {

// 下面一行会报错，因为x没有声明

resolve(x + 2);

});

};

someAsyncThing().then(function() {

return someOtherAsyncThing();

}).catch(function(error) {

console.log('oh no', error);

// 下面一行会报错，因为y没有声明

y + 2;

}).then(function() {

console.log('carry on');

});

// oh no [ReferenceError: x is not defined]

上面代码中，catch方法抛出一个错误，因为后面没有别的catch方法了，导致这个错误不会被捕获，也不会传递到外层。如果改写一下，结果就不一样了。

someAsyncThing().then(function() {

return someOtherAsyncThing();

}).catch(function(error) {

console.log('oh no', error);

// 下面一行会报错，因为y没有声明

y + 2;

}).catch(function(error) {

console.log('carry on', error);

});

// oh no [ReferenceError: x is not defined]

// carry on [ReferenceError: y is not defined]

上面代码中，第二个catch方法用来捕获，前一个catch方法抛出的错误。

### Promise.all()

Promise.all方法用于将多个Promise实例，包装成一个新的Promise实例。

var p = Promise.all([p1, p2, p3]);

上面代码中，Promise.all方法接受一个数组作为参数，p1、p2、p3都是Promise对象的实例，如果不是，就会先调用下面讲到的Promise.resolve方法，将参数转为Promise实例，再进一步处理。（Promise.all方法的参数可以不是数组，但必须具有Iterator接口，且返回的每个成员都是Promise实例。）

p的状态由p1、p2、p3决定，分成两种情况。

（1）只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

（2）只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

下面是一个具体的例子。

// 生成一个Promise对象的数组

var promises = [2, 3, 5, 7, 11, 13].map(function (id) {

return getJSON("/post/" + id + ".json");

});

Promise.all(promises).then(function (posts) {

// ...

}).catch(function(reason){

// ...

});

上面代码中，promises是包含6个Promise实例的数组，只有这6个实例的状态都变成fulfilled，或者其中有一个变为rejected，才会调用Promise.all方法后面的回调函数。

下面是另一个例子。

// 生成一个Promise对象的数组

var promises = [2, 3, 5, 7, 11, 13].map(function (id) {

return getJSON("/post/" + id + ".json");

});

Promise.all(promises).then(function (posts) {

// ...

}).catch(function(reason){

// ...

});

上面代码中，promises是包含6个Promise实例的数组，只有这6个实例的状态都变成fulfilled，或者其中有一个变为rejected，才会调用Promise.all方法后面的回调函数。

下面是另一个例子

const databasePromise = connectDatabase();

const booksPromise = databaseProimse

.then(findAllBooks);

const userPromise = databasePromise

.then(getCurrentUser);

Promise.all([

booksPromise,

userPromise

])

.then(([books, user]) => pickTopRecommentations(books, user));

上面代码中，booksPromise和userPromise是两个异步操作，只有等到它们的结果都返回了，才会触发pickTopRecommentations这个回调函数。

### Promise.race()

Promise.race方法同样是将多个Promise实例，包装成一个新的Promise实例。

var p = Promise.race([p1, p2, p3]);

上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

Promise.race方法的参数与Promise.all方法一样，如果不是 Promise 实例，就会先调用下面讲到的Promise.resolve方法，将参数转为 Promise 实例，再进一步处理。

下面是一个例子，如果指定时间内没有获得结果，就将Promise的状态变为reject，否则变为resolve。

var p = Promise.race([

fetch('/resource-that-may-take-a-while'),

new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('request timeout')), 5000)

})

])

p.then(response => console.log(response))

p.catch(error => console.log(error))

## ES6异步操作和async函数

异步编程对JavaScript语言太重要。Javascript语言的执行环境是“单线程”的，如果没有异步编程，根本没法用，非卡死不可。

ES6诞生以前，异步编程的方法，大概有下面四种。

* 回调函数
* 事件监听
* 发布/订阅
* Promise 对象

ES6将JavaScript异步编程带入了一个全新的阶段，ES7的Async函数更是提出了异步编程的终极解决方案。

### 基本概念

#### 异步

所谓"异步"，简单说就是一个任务分成两段，先执行第一段，然后转而执行其他任务，等做好了准备，再回过头执行第二段。

比如，有一个任务是读取文件进行处理，任务的第一段是向操作系统发出请求，要求读取文件。然后，程序执行其他任务，等到操作系统返回文件，再接着执行任务的第二段（处理文件）。这种不连续的执行，就叫做异步。

相应地，连续的执行就叫做同步。由于是连续执行，不能插入其他任务，所以操作系统从硬盘读取文件的这段时间，程序只能干等着。

#### 回调函数

JavaScript语言对异步编程的实现，就是回调函数。所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。它的英语名字callback，直译过来就是"重新调用"。

读取文件进行处理，是这样写的。

fs.readFile('/etc/passwd', function (err, data) {

if (err) throw err;

console.log(data);

});

上面代码中，readFile函数的第二个参数，就是回调函数，也就是任务的第二段。等到操作系统返回了/etc/passwd这个文件以后，回调函数才会执行。

一个有趣的问题是，为什么Node.js约定，回调函数的第一个参数，必须是错误对象err（如果没有错误，该参数就是null）？原因是执行分成两段，在这两段之间抛出的错误，程序无法捕捉，只能当作参数，传入第二段。

### Promise

回调函数本身并没有问题，它的问题出现在多个回调函数嵌套。假定读取A文件之后，再读取B文件，代码如下。

fs.readFile(fileA, function (err, data) {

fs.readFile(fileB, function (err, data) {

// ...

});

});

不难想象，如果依次读取多个文件，就会出现多重嵌套。代码不是纵向发展，而是横向发展，很快就会乱成一团，无法管理。这种情况就称为"回调函数地狱"（callback hell）。

Promise就是为了解决这个问题而提出的。它不是新的语法功能，而是一种新的写法，允许将回调函数的嵌套，改成链式调用。采用Promise，连续读取多个文件，写法如下。

var readFile = require('fs-readfile-promise');

readFile(fileA)

.then(function(data){

console.log(data.toString());

})

.then(function(){

return readFile(fileB);

})

.then(function(data){

console.log(data.toString());

})

.catch(function(err) {

console.log(err);

});

上面代码中，我使用了fs-readfile-promise模块，它的作用就是返回一个Promise版本的readFile函数。Promise提供then方法加载回调函数，catch方法捕捉执行过程中抛出的错误。

可以看到，Promise 的写法只是回调函数的改进，使用then方法以后，异步任务的两段执行看得更清楚了，除此以外，并无新意。

Promise 的最大问题是代码冗余，原来的任务被Promise 包装了一下，不管什么操作，一眼看去都是一堆 then，原来的语义变得很不清楚。

那么，有没有更好的写法呢？

### Generator函数

#### 协程

传统的编程语言，早有异步编程的解决方案（其实是多任务的解决方案）。其中有一种叫做"协程"（coroutine），意思是多个线程互相协作，完成异步任务。

协程有点像函数，又有点像线程。它的运行流程大致如下。

* 第一步，协程A开始执行。
* 第二步，协程A执行到一半，进入暂停，执行权转移到协程B。
* 第三步，（一段时间后）协程B交还执行权。
* 第四步，协程A恢复执行。

上面流程的协程A，就是异步任务，因为它分成两段（或多段）执行。

举例来说，读取文件的协程写法如下。

function \*asyncJob() {

// ...其他代码

var f = yield readFile(fileA);

// ...其他代码

}

上面代码的函数asyncJob是一个协程，它的奥妙就在其中的yield命令。它表示执行到此处，执行权将交给其他协程。也就是说，yield命令是异步两个阶段的分界线。

协程遇到yield命令就暂停，等到执行权返回，再从暂停的地方继续往后执行。它的最大优点，就是代码的写法非常像同步操作，如果去除yield命令，简直一模一样。

#### Generator函数的概念

Generator函数是协程在ES6的实现，最大特点就是可以交出函数的执行权（即暂停执行）。

整个Generator函数就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用yield语句注明。Generator函数的执行方法如下。

function\* gen(x){

var y = yield x + 2;

return y;

}

var g = gen(1);

g.next() // { value: 3, done: false }

g.next() // { value: undefined, done: true }

上面代码中，调用Generator函数，会返回一个内部指针（即遍历器）g 。这是Generator函数不同于普通函数的另一个地方，即执行它不会返回结果，返回的是指针对象。调用指针g的next方法，会移动内部指针（即执行异步任务的第一段），指向第一个遇到的yield语句，上例是执行到x + 2为止。

换言之，next方法的作用是分阶段执行Generator函数。每次调用next方法，会返回一个对象，表示当前阶段的信息（value属性和done属性）。value属性是yield语句后面表达式的值，表示当前阶段的值；done属性是一个布尔值，表示Generator函数是否执行完毕，即是否还有下一个阶段。

#### Generator函数的数据交换和错误处理

Generator函数可以暂停执行和恢复执行，这是它能封装异步任务的根本原因。除此之外，它还有两个特性，使它可以作为异步编程的完整解决方案：函数体内外的数据交换和错误处理机制。

next方法返回值的value属性，是Generator函数向外输出数据；next方法还可以接受参数，这是向Generator函数体内输入数据。

function\* gen(x){

var y = yield x + 2;

return y;

}

var g = gen(1);

g.next() // { value: 3, done: false }

g.next(2) // { value: 2, done: true }

上面代码中，第一个next方法的value属性，返回表达式x + 2的值（3）。第二个next方法带有参数2，这个参数可以传入 Generator 函数，作为上个阶段异步任务的返回结果，被函数体内的变量y接收。因此，这一步的 value 属性，返回的就是2（变量y的值）。

Generator 函数内部还可以部署错误处理代码，捕获函数体外抛出的错误。

function\* gen(x){

try {

var y = yield x + 2;

} catch (e){

console.log(e);

}

return y;

}

var g = gen(1);

g.next();

g.throw('出错了');

// 出错了

上面代码的最后一行，Generator函数体外，使用指针对象的throw方法抛出的错误，可以被函数体内的try ...catch代码块捕获。这意味着，出错的代码与处理错误的代码，实现了时间和空间上的分离，这对于异步编程无疑是很重要的

#### 异步任务的封装

下面看看如何使用 Generator 函数，执行一个真实的异步任务。

var fetch = require('node-fetch');

function\* gen(){

var url = 'https://api.github.com/users/github';

var result = yield fetch(url);

console.log(result.bio);

}

上面代码中，Generator函数封装了一个异步操作，该操作先读取一个远程接口，然后从JSON格式的数据解析信息。就像前面说过的，这段代码非常像同步操作，除了加上了yield命令。

执行这段代码的方法如下。

var g = gen();

var result = g.next();

result.value.then(function(data){

return data.json();

}).then(function(data){

g.next(data);

});

### Thunk 函数

#### 参数的求值策略

Thunk函数早在上个世纪60年代就诞生了。

那时，编程语言刚刚起步，计算机学家还在研究，编译器怎么写比较好。一个争论的焦点是"求值策略"，即函数的参数到底应该何时求值。

var x = 1;

function f(m){

return m \* 2;

}

f(x + 5)

上面代码先定义函数f，然后向它传入表达式x + 5。请问，这个表达式应该何时求值？

一种意见是"传值调用"（call by value），即在进入函数体之前，就计算x + 5的值（等于6），再将这个值传入函数f 。C语言就采用这种策略。

f(x + 5)

// 传值调用时，等同于

f(6)

另一种意见是"传名调用"（call by name），即直接将表达式x + 5传入函数体，只在用到它的时候求值。Haskell语言采用这种策略。

f(x + 5)

// 传名调用时，等同于

(x + 5) \* 2

传值调用和传名调用，哪一种比较好？回答是各有利弊。传值调用比较简单，但是对参数求值的时候，实际上还没用到这个参数，有可能造成性能损失。

function f(a, b){

return b;

}

f(3 \* x \* x - 2 \* x - 1, x);

#### Thunk函数的含义

编译器的"传名调用"实现，往往是将参数放到一个临时函数之中，再将这个临时函数传入函数体。这个临时函数就叫做Thunk函数。

function f(m){

return m \* 2;

}

f(x + 5);

// 等同于

var thunk = function () {

return x + 5;

};

function f(thunk){

return thunk() \* 2;

}

上面代码中，函数f的参数x + 5被一个函数替换了。凡是用到原参数的地方，对Thunk函数求值即可。

这就是Thunk函数的定义，它是"传名调用"的一种实现策略，用来替换某个表达式。

#### JavaScript语言的thunk函数

JavaScript语言是传值调用，它的Thunk函数含义有所不同。在JavaScript语言中，Thunk函数替换的不是表达式，而是多参数函数，将其替换成单参数的版本，且只接受回调函数作为参数。

// 正常版本的readFile（多参数版本）

fs.readFile(fileName, callback);

// Thunk版本的readFile（单参数版本）

var readFileThunk = Thunk(fileName);

readFileThunk(callback);

var Thunk = function (fileName){

return function (callback){

return fs.readFile(fileName, callback);

};

};

上面代码中，fs模块的readFile方法是一个多参数函数，两个参数分别为文件名和回调函数。经过转换器处理，它变成了一个单参数函数，只接受回调函数作为参数。这个单参数版本，就叫做Thunk函数。

任何函数，只要参数有回调函数，就能写成Thunk函数的形式。下面是一个简单的Thunk函数转换器。

// ES5版本

var Thunk = function(fn){

return function (){

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

return function (callback){

args.push(callback);

return fn.apply(this, args);

}

};

};

// ES6版本

var Thunk = function(fn) {

return function (...args) {

return function (callback) {

return fn.call(this, ...args, callback);

}

};

};

使用上面的转换器，生成fs.readFile的Thunk函数。

var readFileThunk = Thunk(fs.readFile);

readFileThunk(fileA)(callback);

下面是一个完整的例子

function f(a, cb) {

cb(a);

}

let ft = Thunk(f);

let log = console.log.bind(console);

ft(1)(log) // 1

#### Thunkify模块

生产环境的转换器，建议使用Thunkify模块。

$ npm install thunkify

使用方式如下

var thunkify = require('thunkify');

var fs = require('fs');

var read = thunkify(fs.readFile);

read('package.json')(function(err, str){

// ...

});

Thunkify的源码与上一节那个简单的转换器非常像。

function thunkify(fn){

return function(){

var args = new Array(arguments.length);

var ctx = this;

for(var i = 0; i < args.length; ++i) {

args[i] = arguments[i];

}

return function(done){

var called;

args.push(function(){

if (called) return;

called = true;

done.apply(null, arguments);

});

try {

fn.apply(ctx, args);

} catch (err) {

done(err);

}

}

}

};

它的源码主要多了一个检查机制，变量called确保回调函数只运行一次。这样的设计与下文的Generator函数相关。请看下面的例子。

function f(a, b, callback){

var sum = a + b;

callback(sum);

callback(sum);

}

var ft = thunkify(f);

var print = console.log.bind(console);

ft(1, 2)(print);

// 3

上面代码中，由于thunkify只允许回调函数执行一次，所以只输出一行结果。

### Generator函数的流程管理

你可能会问， Thunk函数有什么用？回答是以前确实没什么用，但是ES6有了Generator函数，Thunk函数现在可以用于Generator函数的自动流程管理。

Generator函数可以自动执行。

function\* gen() {

// ...

}

var g = gen();

var res = g.next();

while(!res.done){

console.log(res.value);

res = g.next();

}

上面代码中，Generator函数gen会自动执行完所有步骤。

但是，这不适合异步操作。如果必须保证前一步执行完，才能执行后一步，上面的自动执行就不可行。这时，Thunk函数就能派上用处。以读取文件为例。下面的Generator函数封装了两个异步操作。

var fs = require('fs');

var thunkify = require('thunkify');

var readFile = thunkify(fs.readFile);

var gen = function\* (){

var r1 = yield readFile('/etc/fstab');

console.log(r1.toString());

var r2 = yield readFile('/etc/shells');

console.log(r2.toString());

};

上面代码中，yield命令用于将程序的执行权移出Generator函数，那么就需要一种方法，将执行权再交还给Generator函数。

这种方法就是Thunk函数，因为它可以在回调函数里，将执行权交还给Generator函数。为了便于理解，我们先看如何手动执行上面这个Generator函数。

var g = gen();

var r1 = g.next();

r1.value(function(err, data){

if (err) throw err;

var r2 = g.next(data);

r2.value(function(err, data){

if (err) throw err;

g.next(data);

});

});

上面代码中，变量g是Generator函数的内部指针，表示目前执行到哪一步。next方法负责将指针移动到下一步，并返回该步的信息（value属性和done属性）。

仔细查看上面的代码，可以发现Generator函数的执行过程，其实是将同一个回调函数，反复传入next方法的value属性。这使得我们可以用递归来自动完成这个过程。

### Thunk函数的自动流程管理

Thunk函数真正的威力，在于可以自动执行Generator函数。下面就是一个基于Thunk函数的Generator执行器

function run(fn) {

var gen = fn();

function next(err, data) {

var result = gen.next(data);

if (result.done) return;

result.value(next);

}

next();

}

function\* g() {

// ...

}

run(g);

上面代码的run函数，就是一个Generator函数的自动执行器。内部的next函数就是Thunk的回调函数。next函数先将指针移到Generator函数的下一步（gen.next方法），然后判断Generator函数是否结束（result.done属性），如果没结束，就将next函数再传入Thunk函数（result.value属性），否则就直接退出。

有了这个执行器，执行Generator函数方便多了。不管内部有多少个异步操作，直接把Generator函数传入run函数即可。当然，前提是每一个异步操作，都要是Thunk函数，也就是说，跟在yield命令后面的必须是Thunk函数。

var g = function\* (){

var f1 = yield readFile('fileA');

var f2 = yield readFile('fileB');

// ...

var fn = yield readFile('fileN');

};

run(g);

上面代码中，函数g封装了n个异步的读取文件操作，只要执行run函数，这些操作就会自动完成。这样一来，异步操作不仅可以写得像同步操作，而且一行代码就可以执行

Thunk函数并不是Generator函数自动执行的唯一方案。因为自动执行的关键是，必须有一种机制，自动控制Generator函数的流程，接收和交还程序的执行权。回调函数可以做到这一点，Promise 对象也可以做到这一点。

## 类及类的继承

### 累的声明与示例化

类的声明有两种方式：

//类的声明

var Animal = function () {

this.name = 'Animal';

};

//ES6中类的声明

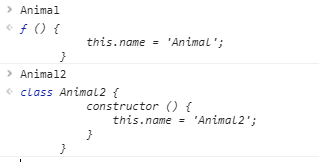
class Animal2 {

constructor () {

this.name = 'Animal2';

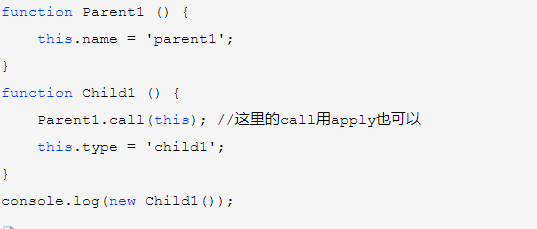
}

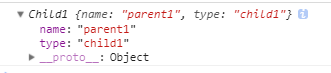
}



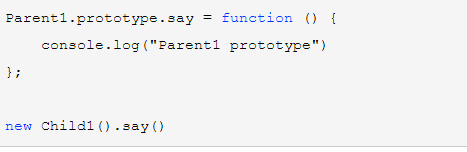
### 类的继承

第一种：借助构造函数实现



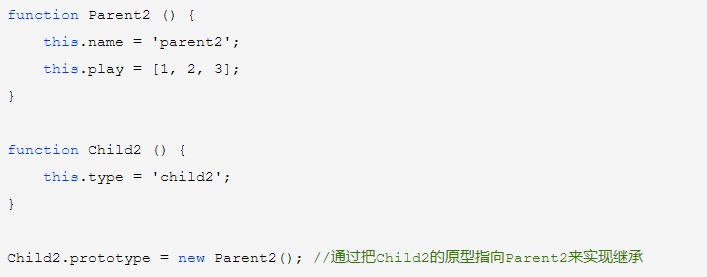


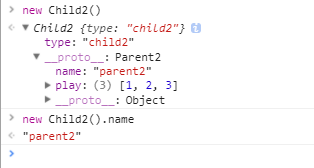
注意：这种继承的方法有一个缺点，它只是把父类中的属性继承了，但父类的原型中的属性继承不了。继续上面的代码



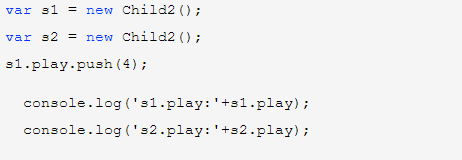


第二种借助原型链实现继承



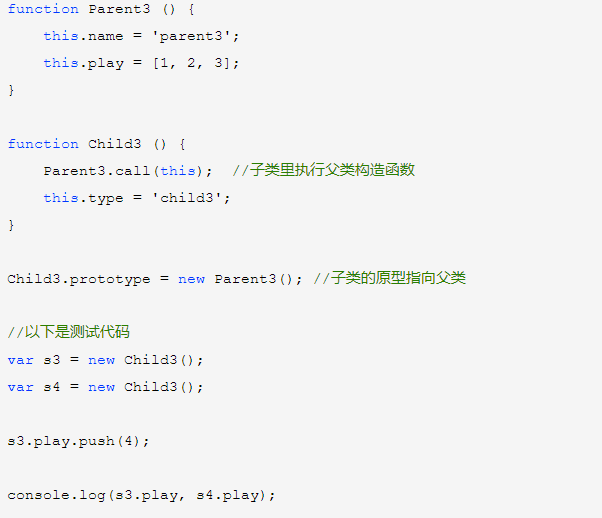


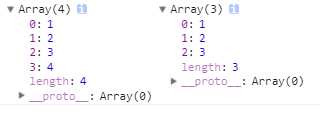
注意：缺点：



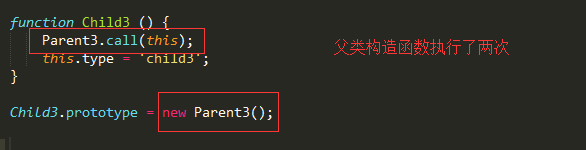


第三种方式：组合方式（将上面两种方式结合起来使用）



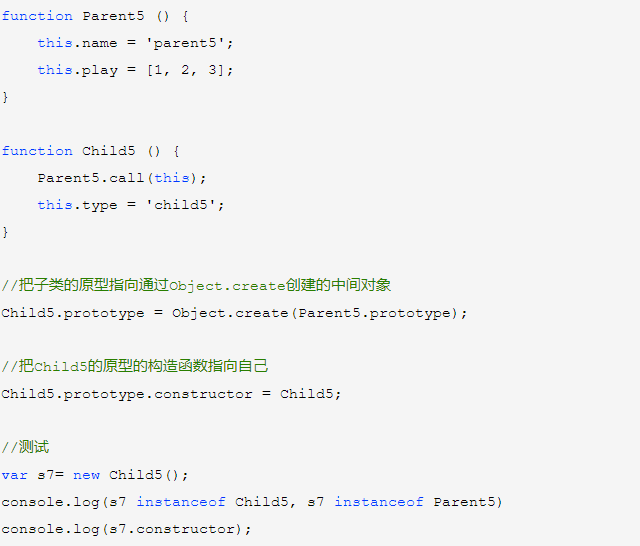


注意：这种方式仍然不完美，因为创建一个个子类的实例的时候，父类的构造函数执行了两次。



第四种方式：组合优化

原理：把子类的原型指向通过Object.create创建的中间对象





ES6新增继承super、extends、constructor

Class可以通过extends关键字实现继承通过extends关键字，继承了Father类的所有属性和方法，但是由于没有部署任何代码,所以这两个类完全一样，等于复制了一个Father类

**子类必须在constructor方法中调用super方法，否则新建实例时会报错，这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工，如果不调用super方法，子类就得不到this对象；**

**ES5的继承，实质是先创造子类的实例对象this，然后再将父类的方法添加到this上（Parent.apply(this)）,ES6的继承机制完全不同，实质是先创造父类的实例对象this（所以必须先调用super方法）,然后再用子类的构造函数修改this；**  
**如果子类没有定义constructor方法，这个方法会默认添加，也就是说，不管有没有显式定义，任何一个子类都有constructor方法。**

**Object.getPrototypeOf()方法用来从子类上获取父类**

super这个关键字，**既可以当作函数使用，也可以当作对象使用，**

**注意：**

**Extends表示：该类不应该直接基于默认的Object原型，而应该基于其他类。这被称为超类（superclass）。派生类是子类（subclass）**

**Super作用原理：**

为了初始化SymmetricMatrix实例，构造器通过super关键字调用其超类的构造器。 这是必要的，因为如果这个新对象的行为（大致）像Matrix，它需要矩阵具有的实例属性。 为了确保矩阵是对称的，构造器包装了content方法，来交换对角线以下的值的坐标。

set方法再次使用super，但这次不是调用构造器，而是从超类的一组方法中调用特定的方法。 我们正在重新定义set，但是想要使用原来的行为。 因为this.set引用新的set方法，所以调用这个方法是行不通的。 在类方法内部，super提供了一种方法，来调用超类中定义的方法。

继承允许我们用相对较少的工作，从现有数据类型构建稍微不同的数据类型。 它是面向对象传统的基础部分，与封装和多态一样。 尽管后两者现在普遍被认为是伟大的想法，但继承更具争议性。

尽管封装和多态可用于将代码彼此分离，从而减少整个程序的耦合，但继承从根本上将类连接在一起，从而产生更多的耦合。 继承一个类时，比起单纯使用它，你通常必须更加了解它如何工作。 继承可能是一个有用的工具，并且我现在在自己的程序中使用它，但它不应该成为你的第一个工具，你可能不应该积极寻找机会来构建类层次结构（类的家族树）。

## EcmaScript 拓展

### 可选链式调用（ES2020新特性）

?. 如果项目无法兼容时需要配置

安装依赖：

[@babel/plugin-proposal-optional-chaining - npm (npmjs.com)](https://www.npmjs.com/package/@babel/plugin-proposal-optional-chaining)

npm install --save-dev @babel/plugin-proposal-optional-chaining

安装后，然后在babel.config.js里面配置下。打包就不会报错了。"plugins": [  
"@babel/plugin-proposal-optional-chaining"  
]

### 空位合并操作符

当我们查询某个属性时，经常会给没有该属性就设置一个默认的值，比如下面两种方式

let c = a ? a : b // 方式1

let c = a || b // 方式2

这两种方式有个明显的弊端，它都会覆盖所有的假值，如(0, '', false)，这些值可能是在某些情况下有效的输入。

ES2020诞生了个新特性--空位合并操作符，用 ?? 表示。如果表达式在??的左侧运算符求值为**undefined 或 null**，就返回其右侧默认值。

let c = a ?? b;

// 等价于let c = a !== undefined && a !== null ? a : b;

### Promise.allSettled

Promise.all 具有并发执行异步任务的能力。但它的最大问题就是**如果参数中的任何一个promise为reject的话，则整个Promise.all 调用会立即终止**，

Promise.allSettled([

Promise.reject({ code: 500, msg: '服务异常' }),

Promise.resolve({ code: 200, list: [] }),

Promise.resolve({ code: 200, list: [] })])

.then(res => {

console.log(res)

/\*

0:{status: "rejected", reason: {…}}

1: {status: "fulfilled", value: {…}}

2: {status: "fulfilled", value: {…}}

\*/

// 过滤掉 rejected 状态，尽可能多的保证页面区域数据渲染RenderContent(res.filter(el => {return el.status !== 'rejected'}))})

Promise.allSettled跟Promise.all类似，其参数接受一个Promise的数组，返回一个新的Promise**，唯一的不同在于，它不会进行短路，**也就是说当Promise全部处理完成后，我们可以拿到每个Promise的状态，而不管是否处理成功。

### String.prototype.matchAll

如果一个正则表达式在字符串里面有多个匹配，现在一般使用g修饰符或y修饰符，在循环里面逐一取出

值得注意的是，如果没有修饰符 /g, .exec() 只返回第一个匹配。现在通过String.prototype.matchAll方法，可以一次性取出所有匹配。

上面代码中，由于string.matchAll(regex)返回的是遍历器，所以可以用for...of循环取出。

### Dynamic import

现在前端打包资源越来越大，前端应用初始化时根本不需要全部加载这些逻辑资源，为了首屏渲染速度更快，很多时候都是动态导入（按需加载）模块，比如懒加载图片等，这样可以帮助您提高应用程序的性能。

el.onclick = () => {

import('/modules/my-module.js')

.then(module =>

{

// Do something with the module.

})

.catch(err => {

// load error;

})

}

import()可以用于script脚本中,**import(module) 函数可以在任何地方调用。它返回一个解析为模块对象的 promise。**

这种使用方式也支持 await 关键字。

let module = await import('/modules/my-module.js');

通过动态导入代码，您可以减少应用程序加载所需的时间，并尽可能快地将某些内容返回给用户。

### BigInt

javascript 在 Math 上一直很糟糕的原因之一是只能安全的表示-(2^53-1)

至

2^53-1

范围的值，即

Number.MIN\_SAFE\_INTEGER

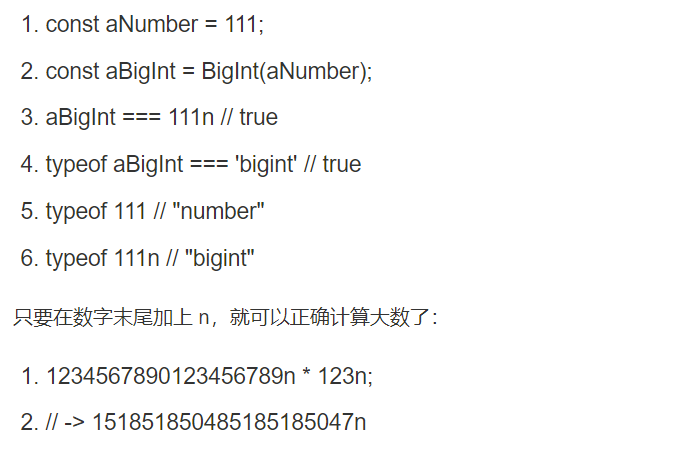
至

Number.MAX\_SAFE\_INTEGER

，超出这个范围的整数计算或者表示会丢失精度。

于是 BigInt 应运而生，**它是第7个原始类型，可安全地进行大数整型计算**。你可以在BigInt上使用与普通数字相同的运算符，例如 +, -, /, \*, %等等。

创建 BigInt 类型的值也非常简单，只需要在数字后面加上 n 即可。例如，123 变为 123n。也可以使用全局方法 BigInt(value) 转化，入参 value 为数字或数字字符串。



不过有一个问题，在大多数操作中，不能将 BigInt与Number混合使用。比较Number和 BigInt是可以的，但是不能把它们相加。

### globalThis

globalThis 是一个全新的标准方法用来获取全局 this 。之前开发者会通过如下的一些方法获取：

全局变量 window：是一个经典的获取全局对象的方法。但是它在 Node.js 和 Web Workers 中并不能使用

全局变量 self：通常只在 Web Workers 和浏览器中生效。但是它不支持 Node.js。一些人会通过判断 self 是否存在识别代码是否运行在 Web Workers 和浏览器中

全局变量 global：只在 Node.js 中生效

而**globalThis 目的就是提供一种标准化方式访问全局对象**，有了 globalThis 后，你可以在任意上下文，任意时刻都能获取到全局对象。

如果您在浏览器上，globalThis将为window，如果您在Node上，globalThis则将为global。因此，不再需要考虑不同的环境问题。

# Web API接口

## Location

**Location**接口表示其链接到的对象的位置（URL）。所做的修改反映在与之相关的对象上。 [Document](https://developer.mozilla.org/zh-CN/docs/Web/API/Document) 和 [Window](https://developer.mozilla.org/zh-CN/docs/Web/API/Window) 接口都有这样一个链接的Location，分别通过 [Document.location](https://developer.mozilla.org/zh-CN/docs/Web/API/Document/location" \o "Document.location 是一个只读属性，返回一个 Location 对象，包含有文档的 URL 相关的信息，并提供了改变该 URL 和加载其他 URL 的方法。)和[Window.location](https://developer.mozilla.org/zh-CN/docs/Web/API/Window/location" \o "window.location 只读属性，返回一个 Location  对象，其中包含有关文档当前位置的信息。) 访问。

### 属性

*Location*接口不继承任何属性，但是实现了那些来自 *[URLUtils](https://developer.mozilla.org/zh-CN/docs/Web/API/URLUtils" \o "The HTMLHyperlinkElementUtils mixin defines utility methods and properties to work with HTMLAnchorElement and HTMLAreaElement. These utilities allow to deal with common features like URLs.)* 的属性。

[**Location.href**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/href)

包含整个URL的一个DOMString

[**Location.protocol**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/protocol)

包含URL对应协议的一个[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)，最后有一个":"。

[**Location.host**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/host)

包含了域名的一个[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)，可能在该串最后带有一个":"并跟上URL的端口号。

[**Location.hostname**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/hostname)

含URL域名的一个[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)。

[**Location.port**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/port)

包含端口号的一个[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)。

[**Location.pathname**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/pathname)

包含URL中路径部分的一个[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)，开头有一个“/"。

[**Location.search**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/search)

 包含URL参数的一个[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)，开头有一个“?”。

[**Location.hash**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/hash)

包含块标识符的[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)，开头有一个“#”。

[**Location.username**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/username)

包含URL中域名前的用户名的一个[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)。

[**Location.password**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/password)

包含URL域名前的密码的一个 [DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)。

[**Location.origin**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/origin)只读

包含页面来源的域名的标准形式[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString)。

### 方法

*Location*没有继承任何方法，但实现了来自*[URLUtils](https://developer.mozilla.org/zh-CN/docs/Web/API/URLUtils" \o "The HTMLHyperlinkElementUtils mixin defines utility methods and properties to work with HTMLAnchorElement and HTMLAreaElement. These utilities allow to deal with common features like URLs.)*的方法。

[**Location.assign()**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/assign)

加载给定URL的内容资源到这个Location对象所关联的对象上。

[**Location.reload()**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/reload)

重新加载来自当前 URL的资源。他有一个特殊的可选参数，类型为 [Boolean](https://developer.mozilla.org/zh-CN/docs/Web/API/Boolean)，该参数为true时会导致该方法引发的刷新一定会从服务器上加载数据。如果是 false或没有制定这个参数，浏览器可能从缓存当中加载页面。

[**Location.replace()**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/replace)

用给定的URL替换掉当前的资源。与 assign() 方法不同的是用 replace()替换的新页面不会被保存在会话的历史 [History](https://developer.mozilla.org/zh-CN/docs/Web/API/History)中，这意味着用户将不能用后退按钮转到该页面。

[**Location.toString()**](https://developer.mozilla.org/zh-CN/docs/Web/API/Location/toString)

返回一个[DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString" \o "DOMString 是一个UTF-16字符串。由于JavaScript已经使用了这样的字符串，所以DOMString 直接映射到 一个String。)，包含整个URL。 它和读取[URLUtils.href](https://developer.mozilla.org/zh-CN/docs/Web/API/URLUtils/href)的效果相同。但是用它是不能够修改Location的值的。

### 例子

// Create anchor element and use href property for the purpose of this example

// A more correct alternative is to browse to the URL and use document.location or window.location

var url = document.createElement('a');

url.href = 'https://developer.mozilla.org/en-US/search?q=URL#search-results-close-container';

console.log(url.href); // https://developer.mozilla.org/en-US/search?q=URL#search-results-close-container

console.log(url.protocol); // https:

console.log(url.host); // developer.mozilla.org

console.log(url.hostname); // developer.mozilla.org

console.log(url.port); // (blank - https assumes port 443)

console.log(url.pathname); // /en-US/search

console.log(url.search); // ?q=URL

console.log(url.hash); // #search-results-close-container

console.log(url.origin); // https://developer.mozilla.org

## WebSockets

**WebSockets** 是一种先进的技术。它可以在用户的浏览器和服务器之间打开交互式通信会话。使用此API，您可以向服务器发送消息并接收事件驱动的响应，而无需通过轮询服务器的方式以获得响应。

### 接口

[**WebSocket**](https://developer.mozilla.org/en-US/docs/Web/API/WebSocket)

用于连接WebSocket服务器的主要接口，之后可以在这个连接上发送 和接受数据。

[**CloseEvent**](https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent)

连接关闭时WebSocket对象发送的事件。

[**MessageEvent**](https://developer.mozilla.org/en-US/docs/Web/API/MessageEvent)

当从服务器获取到消息的时候WebSocket对象触发的事件。

### 工具