

第7章 测试与调试

在我们编写程序的时候，错误和缺陷的是不可避免的。出现错误时，就需要我们不断地对代码进行调试，从而达到我们预期的效果。经常在程序员间流行这么一句话，"程序不是写出来的，而是调试出来"，足以可见调试在整个编码过程中的重要性。而当我们整个项目开发完成之后，也需要不断的对项目进行测试以避免程序Bug的出现。本章，我们就一起来看一下程序编写时的调试与测试。

本章我们将学习如下内容：

- 错误、异常和警告介绍
- JavaScript中的严格模式
- 代码风格检查工具
- 浏览器中的调试
- Error对象
- 异常捕获
- 测试框架Jest

7-1 错误、异常和警告

错误

Error(错误)表示系统级的错误和程序不必处理的异常，是JavaScript运行环境中的内部错误或者硬件问题，比如，内存资源不足等，对于这种错误，程序基本无能为力，除了退出运行外别无选择。错误通常会由如下的原因而导致：

- 系统错误
- 程序员错误
- 用户错误

系统错误：作为程序员，我们编写的程序对外部系统如何工作影响是极少的，所以一般很难修复因系统错误而引起的程序报错。不过尽管如此，我们还是应该知道它们，并尽力通过解决由它们引起的任何问题，从而降低它们对我们程序自身的影响。

程序员错误：程序员错误是我们自己的责任，这里面包含了常见的语法错误，逻辑错误等。作为程序员，我们必须确保错误尽可能的最小化和及时修复。

用户错误：用户错误主要是指用户在使用我们所编写的程序时不按照程序说明来使用，导致程序

出错甚至崩溃。实际上用户错误常常也可以被认为是程序员错误，因为我们所编写的程序应该以能够防止用户犯错的方式来进行设计。

异常

所谓异常，表示需要捕捉或者需要程序进行处理的地方，它处理的是因为程序设计的瑕疵而引起的问题或者在外的输入等引起的一般性问题，是程序必须处理的。

这里总结一下，Error是代码在编译的时候就出现了错误，代码无法执行，比如语法错误，必须修正错误后才能继续执行，代码无法跳过错误。Exception是代码再运行的时候，出现了错误，比如对象中某个属性不存在，或者是数据类型不对。代码可以继续执行，不过会在控制台中输出一段错误，提醒程序员。所以这里我们所讲的错误处理，基本上指的是第二种Exception的情况，用 `try catch` 或者 `onerror` 可以进行捕获的错误。(后面小节会具体介绍)

堆栈跟踪

异常也会生成一个堆栈跟踪(stack trace)。这是指向错误发生点的一系列函数或者方法的调用。通常引起错误的不仅仅是一个函数或者方法的调用。堆栈跟踪会从错误发生的地方向后工作，以识别启动序列的原始函数或者方法。下面的代码展示了堆栈跟踪如何帮助我们发现错误的来源，如下：

```
function three()  
{  
    test();  
}  
function two()  
{  
    three();  
}  
function one()  
{  
    two();  
}  
one();
```

效果：

```
ReferenceError: test is not defined  
    at three (/Users/Jie/Desktop/1.js:3:5)  
    at two (/Users/Jie/Desktop/1.js:7:5)  
    at one (/Users/Jie/Desktop/1.js:11:5)  
    at Object.<anonymous> (/Users/Jie/Desktop/1.js:13:1)  
    at Module._compile (module.js:624:30)
```

```
at Object.Module._extensions..js (module.js:635:10)
at Module.load (module.js:545:32)
at tryModuleLoad (module.js:508:12)
at Function.Module._load (module.js:500:3)
at Function.Module.runMain (module.js:665:10)
```

在本例中，我们有三个函数，函数 `one()` 调用函数 `two()`，然后函数 `two()` 调用函数 `three()`，然后函数 `three()` 调用了一个不存在的函数 `test()`，并引起了一个错误。我们可以利用堆栈跟踪向后工作，看到这个错误最终最初是由函数 `one()` 引起的。

警告

如果代码中一个错误不足以导致程序崩溃，就会发出一条警告(warning)。也就是说，程序在产生一条警告后，会继续向下运行。这听起来感觉挺不错的，但是有些时候也有问题，那是因为产生警告的问题会导致程序不按照我们预期的效果来继续执行。

下面是一个警告的示例，我们给一个未声明的变量进行赋值操作：

```
PI = 3.142;
<< JavaScript Warning: assignment to undeclared variable
```

注意：并非所有浏览器都会对上例中的代码显示一条警告，所以我们在试的时候可能看不到。

警告和异常在不同的环境下表现形式不同。有的浏览器会在浏览器窗口的角落里显示一个小图标，指示一个异常或者一个警告出现了，有的浏览器则是需要打开控制台才能看到警告或者异常。

当浏览器中出现一个运行错误时，HTML依然会出现，但是JavaScript代码会在后台停止运行，不过并非总是很明显。如果出现一个警告，JavaScript会继续运行(尽管有可能不正确)。

7-2 调试错误的方式

在出现错误时，JavaScript是一种相当宽容的语言，在ECMAScript第3版之前，JavaScript没有异常，它不会警告开发人员程序中出现了一个错误，只是默默的在后台失败，而且现在有时候依然如此。这在刚开始看上去好像不错，但是错误可能会导致没有人注意到的意外或者不正确的结果，并且可能长时间潜伏在后台直至某一天导致程序崩溃。

因此，在开发中，如果可能的话，我们应该尽力让代码优雅的失败。这说起来感觉有点搞笑，但是即使发生了错误，也不应该让用户察觉，这样用户体验就不会受到影响。

7-2-1 严格模式

从ECMAScript5开始引入了严格模式，严格模式顾名思义就是更加严格的一种模式，会产生更多的异常和警告，并且禁止使用一些已被废弃的特性。

增加错误的机会看上去是一个坏主意，但是就像上面所说的，早点发现错误总比后面引起问题要好一些。在严格模式下编写代码也可以帮助提高其清晰度和速度，因为它遵循约定，如果使用任何恶意的代码，则会抛出异常。

不使用严格模式的环境经常被称之为草率模式，因为它宽容草率的编程做法。严格模式鼓励像专业程序员那样编写更好的JavaScript代码，提升代码的质量。所以这里推荐使用在严格模式下进行编码。

要使用严格模式，只需要将如下字符串添加到JS文件的第一行即可，如下：

```
"use strict"
```

书写了这句代码以后，支持严格模式的JavaScript引擎就会采用严格模式，而不支持严格模式的JavaScript引擎会忽略这个字符串。

接下来，我们尝试在严格模式下给一个未声明的变量进行赋值，如下：

```
"use strict"
i = 5;
console.log(i);
//ReferenceError: i is not defined
```

我们甚至可以在一个函数内加上这么一句代码，这就表示该函数内使用的严格模式。换句话说，严格模式仅应用于该函数内，如下：

```
function test()  
{  
    "use strict"  
}
```

7-2-2 代码风格检查工具

除了使用严格模式以外，还可以使用诸如 JS Lint，JS Hint 和 ES Lint 这种代码风格检查工具来测试JavaScript的质量。这些工具都是被设计用来突出任何草率的编程做法或者语法错误。如果不遵从某种风格约定的话，比如代码如何缩进，就会报错。

代码风格检查工具会强制大家使用一种编程风格。实际上这也是非常的有用的。当进行一个团队工作时，这会显得非常的有用，因为它能确保团队中每个人都遵守同样的约定。

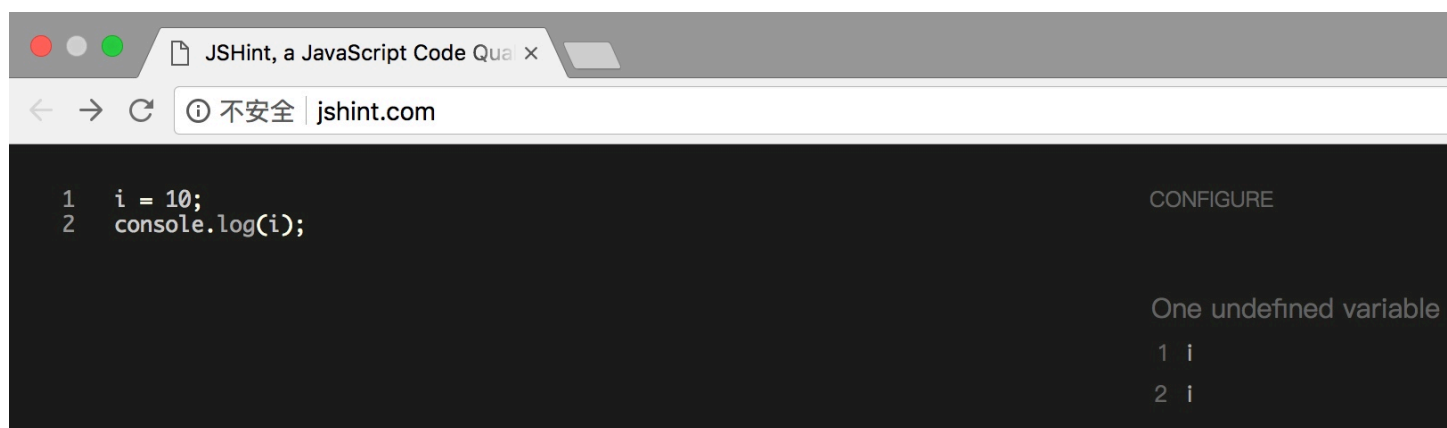
还可以把代码风格检查工具添加为文本编辑器的插件。这些插件在键入代码的时候就会突出在草率模式中所潜在的错误。除了安装这样的插件，我们也可以使用在线的代码风格检查工具，我们只需要将代码粘贴到页面上就可以获取到反馈。

通过代码风格检查测试代码并不能保证代码百分之百是正确的，不过它能让代码更加一致，并且不太可能出问题。

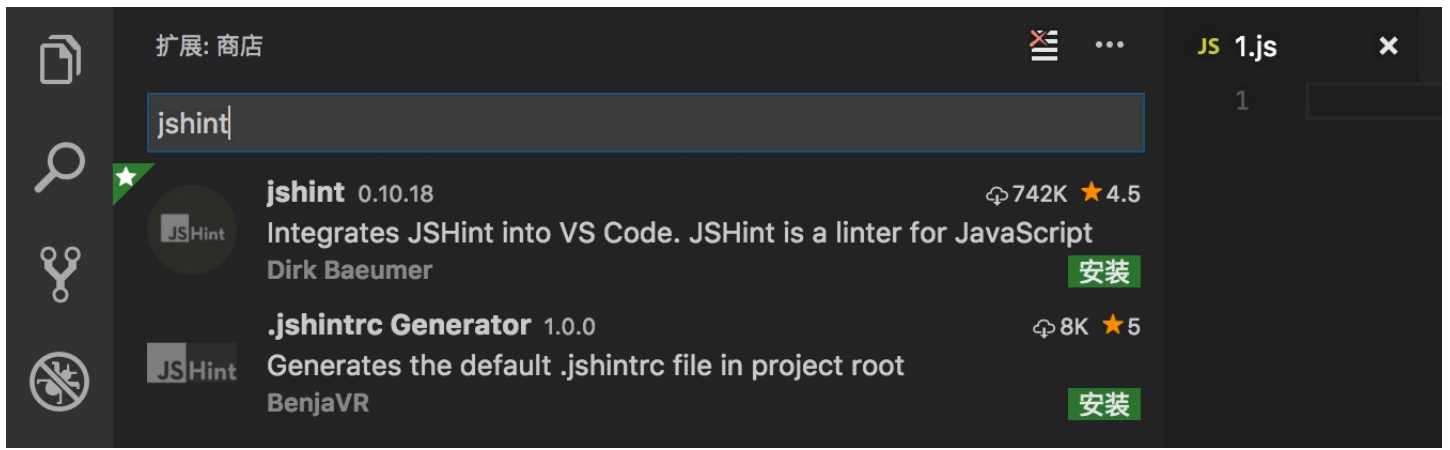
下面我们选择一个代码风格检查工具来看一下，这里我选择了 JS Hint，它的官网如下：

<http://jshint.com/>

打开官方首页，提供了一个可以书写代码的控制台，左边写代码，右边是根据我们所书写的代码而报告的问题，如下：



当然我们也可以在我们的编辑器里面安装 JS Hint，例如我打开我们的vs code，在扩展里面搜索 JS Hint，如下：



安装好以后重启vs code，会提示我们找不到js hint的库，这个时候我们需要使用 `npm install -g jshint` 命令来安装js hint库，如下：

```
Last login: Tue May 1 08:28:12 on console
[Jie-Xie:~ Jie$ npm install -g jshint
/usr/local/bin/jshint -> /usr/local/lib/node_modules/jshint/bin/jshint
+ jshint@2.9.5
added 31 packages from 18 contributors in 3.689s
Jie-Xie:~ Jie$
```

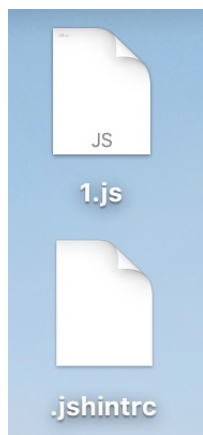
接下来下一步就是要在vscode配置js hint提示工具，新建一个文件，文件名为 `.jshintrc`，文件的内容如下：

```
{
  "globals": {
    "$": false,
    "window": false,
    "document": false
  },
  "strict": true,
  "curly": true,
  "eqeqeq": true,
  "noarg": true,
  "noempty": true,
  "quotmark": "single",
  "undef": true,
  "unused": true
}
```

其中， `globals` 设置明确用作全局变量的名字，以此来辨别被遗忘的 `var` 声明， `strict` 为严格模式， `curly` 设置循环或者条件语句必须使用花括号包围， `eqeqeq` 设置禁止使用 `==` 和 `!=`，强制使用 `===` 和 `!==`， `noarg` 设置禁止使用 `arguments.caller` 和 `arguments.callee`， `noempty` 设置不允许出现空代码块， `quotmark` 设置"single"表示字符串只允许用单引号， `undef` 设置提示变量未定义， `unused` 设置提示变量定义了未使用。

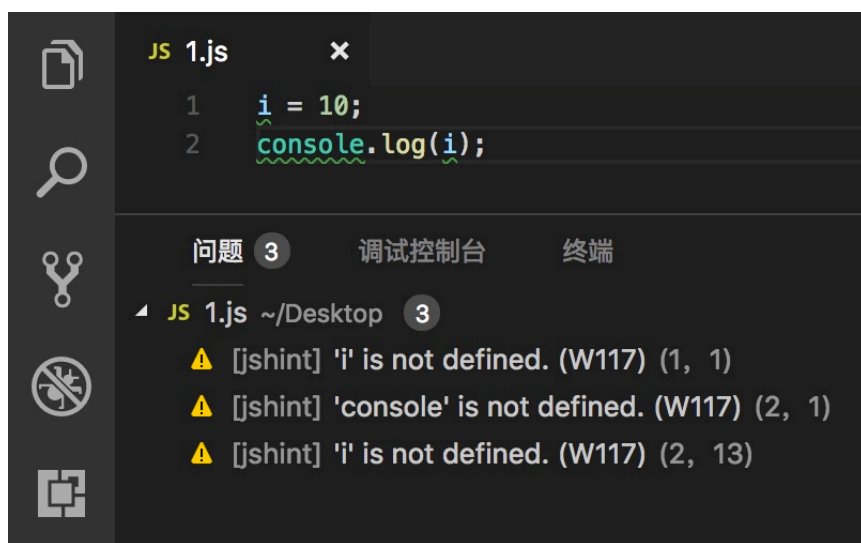
注：关于JSHint详细的配置项可参考<http://jshint.com/docs/options/>

书写好该配置文件以后，必须将该文件放入到我们的项目根目录下面，如下：



注：Mac下以 `.` 开头的文件默认会被设置为隐藏文件，通过 `defaults write com.apple.finder AppleShowAllFiles -boolean true ; killall Finder` 命令可以显示出隐藏文件。通过 `defaults write com.apple.finder AppleShowAllFiles -boolean false ; killall Finder` 命令可以重新不显示隐藏文件。

配置完成之后，当我们实时书写代码时，就能够看到js hint所给出的提示，来规范我们的代码。



7-2-3 功能检测

JavaScript拥有众多的API，并且一直有新的API作为HTML5规范的一部分被开发出来。浏览器厂商也确实在持续不断地添加对这些新功能的支持，不过他们并非总是能够跟得上。更重要的是，有些浏览器会支持某些功能，有些浏览器则不会。我们也不能总是指望用户使用最新的浏览器。

要判断一项功能浏览器是否支持，最好的方法就是使用功能检测。这是在试图实际调用一个方法前，通过一个 `if` 语句来检测一个对象或者方法是否存在来实现的。一般来讲，想要使用某一个

新的API，这个新的API大多会作为 `window` 对象的一个新的属性而存在，所以我们可以利用这个特点来进行功能检测，如下：

```
<body>
  <script>
    if(window.Worker)
    {
      console.log("该浏览器支持Web Work");
    }
    else{
      console.log("不支持Web Work");
    }
  </script>
</body>
```


Modernizr 是一个提供轻松实现功能检测的库，除此之外，也可以使用 `Can I use` 来在线的检测某一个功能被不同浏览器所支持的情况，例如我在 `Can I use` 上面检测Web Work的支持情况，如下：

Can I use web worker ? Settings

4 results found

Detected your country as "China". Would you like to import usage data for that country?

Import No thanks

Web Workers  - LS Global 94.26%

Method of running scripts in the background, isolated from the web page

Current aligned Usage relative Date relative Show all

IE	Edge	Firefox	Chrome	Safari	iOS Safari	Opera Mini	Chrome for Android	UC Browser for Android	Samsung Internet
			49		9.3				
			61		10.2				
	15		62	10.1	10.3				4
11	16	57	63	11	11.2	all	62	11.4	6.2
	17	58	64	TP					
		59	65						
		60	66						

(<http://caniuse.com>)

现在，使用在线的检测方式已经变得更为推荐了。因为利用这种方式所得到的检测结果更加的准确。

7-2-4 浏览器中的调试

调试是一个找到代码中缺陷发生的地方，然后处理它们的过程。在很多时候，错误发生的点并非总是在导致出错的地方，所以我们需要浏览器来进行程序的调试，看看在程序执行的的不同阶段都发生了些什么。而在这种时候，创建断点是非常有用的。断点会暂停代码的执行，让我们可以查看程序中断点所在地方的不同变量的值。

在浏览器中调试JavaScript代码的方式很多，我们一起来看一下以下几种常见的方式。

警告对话框

在最早的时候，JavaScript开发人员大都使用 `alert()` 警告对话框来进行代码的调试。因为 `alert()` 会终止程序的执行，直到点击OK才会继续执行后面的代码。所以它可以很方便的给代码加上断点。在断点所在地方检查变量的值，看看它们是否是我们预期的值。

使用console

很多现代的JavaScript环境都有一个 `console` 对象，该对象提供了很多输出信息和调试的方法。它并非ECMAScript官方规范的一部分，不过现在已经被主流浏览器和Node.js很好的支持了。

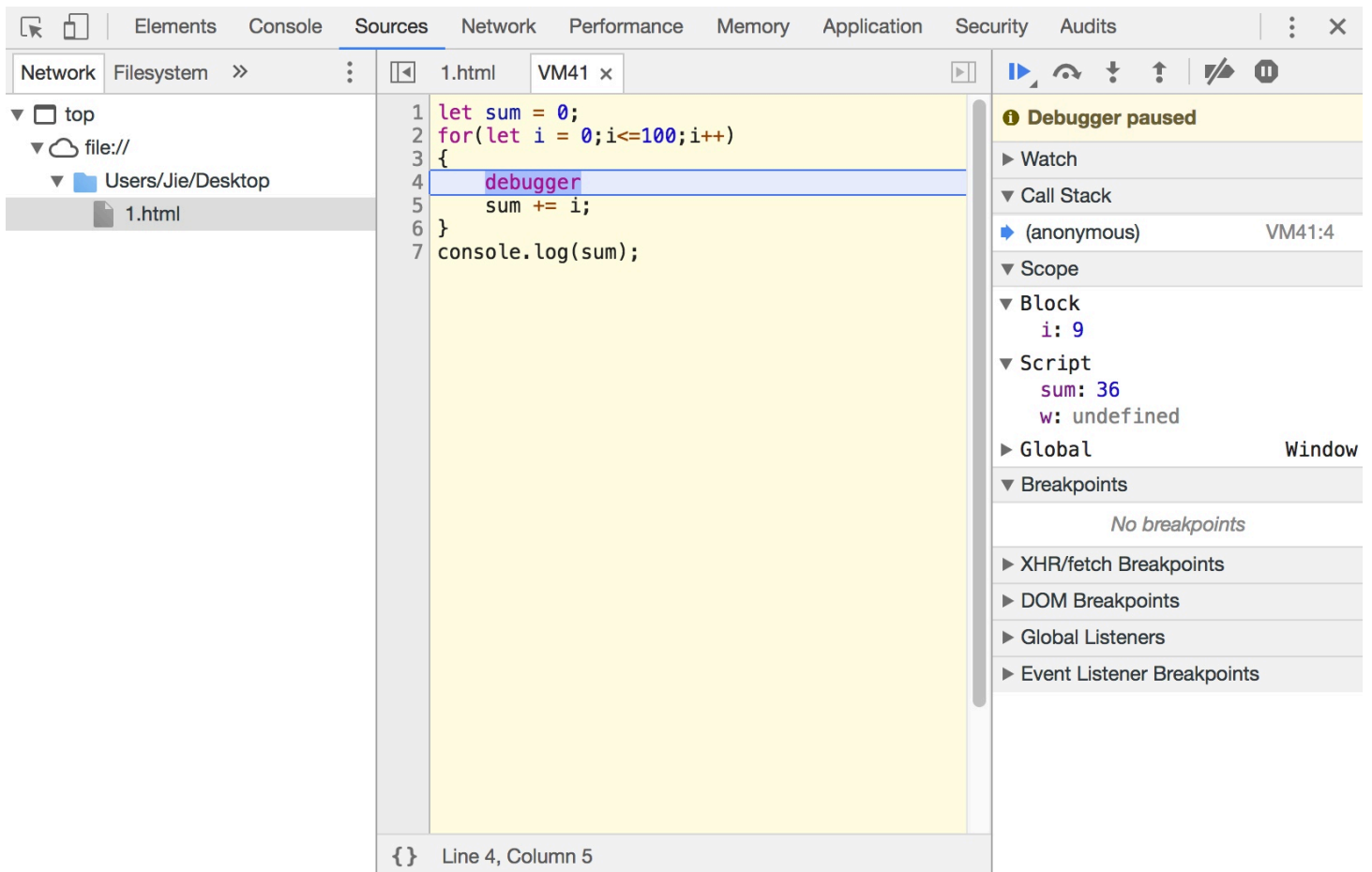
调试工具

很多现代浏览器也提供了一个调试工具让我们可以在代码中设置断点，让代码在某个点上暂停。然后我们就可以看到这些点时所有变量的值，并对它们进行修改。这在跟踪缺陷时候非常有用。

我们可以利用编辑器的断点功能给代码添加上断点，也可以通过 `debugger` 关键字来为代码创建一个暂停执行的断点，并让我们看到程序的当前位置。我们还可以把鼠标悬停在任何变量上，看看此时变量的值。点击 `play` 按钮，可以让程序继续向下一步执行。

```
let sum = 0;
for(let i = 0;i<=100;i++)
{
    // 添加debugger关键字可以给代码添加一个断点
    debugger
    sum += i;
}
console.log(sum);
```

效果：



将上面的代码放入浏览器里面的控制台进行执行时，运行到debugger的地方会自动暂停。

如果是在Node环境下，也可以进入debug模式，详细参见下面的文档：

<https://segmentfault.com/a/1190000008430177>

7-3 error对象

7-3-1 error对象介绍

`error` 对象是包含错误信息的对象，是JavaScript里面的原生对象。当代码在解析或运行发生错误时，引擎就会自动产生并抛出一个 `error` 对象，然后整个程序就中断在发生错误的地方。

我们也可以手动的使用 `Error()` 构造函数来创建一个 `error` 对象，该构造函数接收一个字符串参数作为该错误的提示信息，示例如下：

```
let err = new Error("这是一个错误");
```

除了使用 `Error()` 构造函数来创建一个错误以外，还可以使用诸如 `RangeError()`，`SyntaxError()` 等构造函数来创建不同类型的错误。创建好的错误对象都存在一些属性，不过根据浏览器兼容性的不同，能够使用的属性不太一致，唯一可以使用的属性大致有如下3个：

- `name`：返回该错误对象的错误的类型
- `message`：返回错误的描述，该描述信息可以在示例error对象时作为参数传给构造函数
- `stack`：返回错误的堆栈跟踪，是一个非标准属性。

```
let err = new Error("这是一个普通错误");
console.log(err.name); // Error
console.log(err.message); // 这是一个普通错误
console.log(err.stack);
// Error: 这是一个普通错误
//      at Object.<anonymous> (/Users/Jie/Desktop/1.js:1:73)
//      at Module._compile (module.js:624:30)
//      at Object.Module._extensions..js (module.js:635:10)
//      at Module.load (module.js:545:32)
//      at tryModuleLoad (module.js:508:12)
//      at Function.Module._load (module.js:500:3)
//      at Function.Module.runMain (module.js:665:10)
//      at startup (bootstrap_node.js:187:16)
//      at bootstrap_node.js:608:3
```

7-3-2 error类型

在介绍了error对象之后，接下来我们就来看一下error类型。上面也有提到，在JavaScript中的

error类型有多种。不同的错误都有对应一种error类型，而当错误发生时，就会抛出对应类型的错误对象。

ECMA-262定义了下列7种错误类型：

```
Error
EvalError(eval错误)
RangeError(范围错误)
ReferenceError(引用错误)
SyntaxError(语法错误)
TypeError(类型错误)
URIError(URI错误)
```

其中，Error是基类型，其他错误类型都继承自该类型。因此，所有错误类型共享了一组相同的属性。Error类型的错误很少见，如果有也是浏览器抛出的；这个基类型的主要目的是供开发人员抛出自定义错误。接下来针对下面的6种错误类型进行一个简单的介绍：

EvalError(eval错误)

eval() 函数没有被正确执行时，会抛出 EvalError 错误。该错误类型已经不再在ES5中出现了，只是为了保证与以前代码兼容，才继续保留。

RangeError(范围错误)

RangeError 类型的错误会在一个值超出相应范围时触发，主要包括超出数组长度范围以及超出数字取值范围等。

```
new Array(-1);
//RangeError: Invalid array length
new Array(Number.MAX_VALUE);
//RangeError: Invalid array length
(1234).toExponential(21);
//RangeError: toExponential() argument must be between 0 and 20
```

ReferenceError(引用错误)

引用一个不存在的变量时，会触发ReferenceError(引用错误)。

```
console.log(a);
//ReferenceError: a is not defined
```

SyntaxError(语法错误)

当不符合语法规则时，会抛出SyntaxError(语法错误)

```
let 2a = 5;  
console.log(2a);  
//SyntaxError: Invalid or unexpected token
```

TypeError(类型错误)

在变量中保存着意外的类型时，或者在访问不存在的方法时，都会导致 TypeError 类型错误。错误的原因虽然多种多样，但归根结底还是由于在执行特定类型的操作时，变量的类型并不符合要求所致。

```
let a = new 10;  
//TypeError: 10 is not a constructor
```

URIError(URI错误)

URIError 是URI相关函数的参数不正确时抛出的错误，主要涉及 encodeURIComponent()、decodeURI()、decodeURIComponent()、escape() 和 unescape() 这六个函数。

```
decodeURI("%Hello");  
//URIError: URI malformed
```

7-3-3 error事件

任何没有通过 try-catch 处理(后面一小节我们将介绍)的错误都会触发 window 对象的 error 事件。

error 事件可以接收三个参数：错误消息、错误所在的URL和行号。多数情况下，只有错误消息有用，因为URL只是给出了文档的位置，而行号所指的代码行既可能出自嵌入的JavaScript代码，也可能出自外部的文件，要指定 onerror 事件处理程序，可以使用 DOM0 级技术，也可以使用 DOM2 级事件的标准格式。

```
//DOM0级  
window.onerror = function(message,url,line){  
    alert(message);  
};
```

```

}
//DOM2级
window.addEventListener("error",function(message,url,line){
    alert(message);
});

```

浏览器是否显示标准的错误消息，取决于 `onerror` 的返回值。如果返回值为 `false`，则在控制台中显示错误消息，如果返回值为 `true`，则不显示。

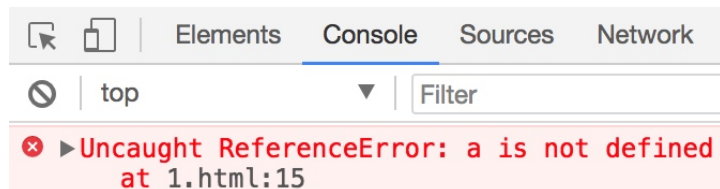
返回值为 `false` 的情况，控制台显示错误信息：

```

// 控制台显示错误消息
<body>
    <script>
        window.onerror = function (message, url, line) {
            alert(message);
            return false;
        }
        console.log(a);
    </script>
</body>

```

效果：



返回值为 `true` 的情况，控制台不显示错误信息：

```

// 控制台不显示错误消息
<body>
    <script>
        window.onerror = function (message, url, line) {
            alert(message);
            return true;
        }
        console.log(a);
    </script>
</body>

```

这个事件处理程序是避免浏览器报告错误的最后一道防线。理想情况下，只要可能就不应该使用它。只要能够适当地使用 `try-catch` 语句，就不会有错误交给浏览器，也就不会触发 `error` 事

件。

图像也支持 `error` 事件。只要图像的`src`特性中的URL不能返回可以被识别的图像格式，就会触发 `error` 事件。此时的 `error` 事件遵循DOM格式，会返回一个以图像为目标的event对象。加载图像失败时会显示一个警告框。发生 `error` 事件时，图像下载过程已经结束，也就是不能再重新下载了。

```
<body>
  <script>
    let image = new Image();
    image.src = 'smilex.gif';
    image.onerror = function (e) {
      console.log("出错了");
      console.log(e);
    }
  </script>
</body>
```

效果：

✖	► GET file:///Users/Jie/Desktop/smilex.gif net::ERR_FILE_NOT_FOUND	smilex.gif:1
	出错了	1.html:14
	► Event {isTrusted: true, type: "error", target: img, currentTarget: img, eventPhase: 2, ...}	1.html:15

7-4 异常的抛出和捕获

到目前为止，我们已经看到了在错误发生时，JavaScript引擎会自动的抛出错误。事实上，我们还可以通过 `throw` 语句手动的抛出我们的异常。这就让我们代码中的任何问题突出出来并处理，而不是悄悄地在后台潜伏。`throw` 语句可以用在任何的JavaScript表达式前面，它会导致程序停止执行，如下：

```
console.log("Hello");
throw 'yes';
console.log("World");
// Hello

// /Users/Jie/Desktop/1.js:2
// throw 'yes';
// ^
// yes
```

不过，最好的做法是抛出一个错误对象，后面用一个 `catch` 块来捕获这个错误对象。

```
throw new Error("there is a error occurred");
```

下面的示例演示了错误的抛出。我们编写了一个 `squareRoot()` 的函数来计算一个数字的平方根。如果用户传入负数，则抛出错误告诉用户“不能传入负数”，示例如下：

```
let squareRoot = function(i){
  if(i < 0)
  {
    throw new RangeError("能传入负数");
  }
  return Math.sqrt(i);
}
let i = squareRoot(25);
console.log(i); //5
let j = squareRoot(-9);
console.log(j);
//RangeError: 能传入负数
```

当错误出现时，程序会中止，并显示一条错误信息。这在开发中是理想的，因为它让我们可以识别和修复错误。不过，在生产环境中，它会让程序表现得好像崩溃了一样，这对用户来讲是一种很不好的体验。

我们可以通过捕获错误来优雅的进行处理。所有错误对用户来讲都是隐藏的，但是依然是可以识别的。然后我们可以适当地处理错误，可能甚至是忽略它，让程序继续运行。这样可以避免影响用户的体验。

`try`，`catch` 和 `finally` 可以算是错误处理的一种标准模式。如果我们怀疑一段代码会导致错误，那么我们可以将其包在一个 `try` 块中。这样会正常运行块中的代码，不过如果一个错误发生了，它会将错误对象抛给 `catch` 块，示例代码如下：

```
let squareRoot = function(i){
  if(i < 0)
  {
    throw new RangeError("不能传入负数");
  }
  return Math.sqrt(i);
}
try{
  let i = squareRoot(-25);
  console.log(i);
}
catch(error){
  console.log(error.message);
}
```

效果：

不能传入负数

`finally` 块可以添加在 `catch` 块后面。它位于 `try` 和 `catch` 之后，与之构成了一个整体。`finally` 代表不管是否有错误出现，一定会执行。如果想有些代码不管是否出现异常都要执行(例如需要手工关闭或是释放的资源)，这就很有用。示例如下：

```
let squareRoot = function(i){
  if(i < 0)
  {
    throw new RangeError("不能传入负数");
  }
  return Math.sqrt(i);
}
function test(){
  try{
    let i = squareRoot(-25);
    return i;
  }
}
```

```
    catch(error){
        return error.message;
    }
    finally{
        console.log("我肯定要执行的");
    }
}
console.log(test());
```

效果：

我肯定要执行的
不能传入负数

总结

1. 代码中不可避免会有缺陷，越早发现越好。
2. JavaScript中用"use strict"设置严格模式。严格模式可以用在整个文件中，也可以只用在函数中。
3. 代码风格检查工具可以用来确保我们的代码遵循好的做法和约定。
4. 功能检测可以在调用一个方法前检测它是否被浏览器支持，有助于避免抛出异常。但是现在更加推荐的是在提供功能检测库的网站上进行在线检测。
5. 控制台和浏览器内置的调试工具可以用来以交互的方式发现和修复代码中的缺陷。
6. 异常可以使用 `throw` 语句抛出。
7. 当异常发生时，就会创建错误对象。
8. 放在 `try` 块内的代码会在异常发生时，将错误对象传递给 `catch` 块。在 `finally` 块内的代码不管异常有没有发生都会执行。