2.7 模板字面量

在这一小节的最开始，首先我要大声指出 ES6 这个特性的名称非常具有误导性，其根据你对单词**模板**（template）的理解而定。

很多开发者会把模板看作是可复用、可渲染的文字片段，就像绝大多数模板引擎（Mustache、Handlebars 等）提供的功能那样。ES6 对 template 这个词的使用可能暗示着某种类似的东西，就像一种声明可以被重新渲染的在线模板字面量的方法。但是，对于这个特性来说，这种看法并不完整。

所以，在继续之前，我要把它重命名为它应该被称为的：**插入字符串字面量**（或者简称为 interpoliteral）。

你已经非常了解，可以用 " 或 ' 界定符声明字符串，也了解这不是（像某些语言中那样的）smart **字符串**，其中的内容可以插入表达式被重新解析。

但是，ES6 引入了一个新的字符串字面量，使用 ` 作为界定符。这样的字符串字面值支持嵌入基本的字符串插入表达式，会被自动解析和求值。

下面是老的前 ES6 方式：

```
var name = "Kyle";

var greeting = "Hello " + name + "!";

console.log( greeting );              // "Hello Kyle!"
console.log( typeof greeting );       // "string"
```

下面是新的 ES6 方式：

```
var name = "Kyle";

var greeting = `Hello ${name}!`;

console.log( greeting );             // "Hello Kyle!"
console.log( typeof greeting );      // "string"
```

你可以看到，这里在一组字符外用 `..` 来包围，这会被解释为一个字符串字面量，但是其中任何 `${..}` 形式的表达式都会被立即在线解析求值。这种形式的解析求值形式就是**插入**（比模板要精确一些）。

插入字符串字面量表达式的结果就是普通的字符串，值赋给变量 `greeting`。

`typeof greeting == "string"` 说明了为什么不要把这些实体当作是特殊的模板值这一点很重要，因为不能把这个字面量未求值的形式赋给某个实体然后复用。`..` 字符串字面量更像是 IIFE，因为它会自动展开求值。一个 `..` 字符串字面量的结果就是一个字符串。

插入字符串字面量的一个优点是它们可以分散在多行：

```
var text =
`Now is the time for all good men
to come to the aid of their
country!`;

console.log( text );
// Now is the time for all good men
// to come to the aid of their
// country!
```

插入字符串字面量中的换行（新行）会在字符串值中被保留。

在字面量值中，除非作为明确的转义序列出现，`\r` 回车符（码点 `U+000D`）的值或者回车换行符 `\r\n`（码点 `U+000D` 和 `U+000A`）都会被标准化为 `\n` 换行符（码点 `U+000A`）。但是别担心，这种标准化非常少见，很可能只有在复制粘贴文本到 JavaScript 文件的时候才会出现。

2.7.1 插入表达式

在插入字符串字面量的 `${..}` 内可以出现任何合法的表达式，包括函数调用、在线函数表达式调用，甚至其他插入字符串字面量！

考虑：

```
function upper(s) {
    return s.toUpperCase();
}

var who = "reader";

var text =
`A very ${upper( "warm" )} welcome
to all of you ${upper( `${who}s` )}!`;

console.log( text );
// A very WARM welcome
// to all of you READERS!
```

这里，与 `who + "s"` 的形式相比，内层的 `` `${who}s` `` 插入字符串字面量对我们合并变量 `who` 和字符串 `"s"` 来说会方便一点。嵌套插入字符串字面量在一些情况下是有所帮助的，但是如果你发现需要频繁使用这种形式，那么就要警惕了，不然你会发现自己得嵌套好多层。

如果是这样的话，那么很可能你的字符串产生过程需要进行某种抽象。

提醒一下，在代码中使用这些新武器的时候要注意可读性。就像使用默认值表达式和解构赋值表达式的时候一样，能够做某事并不意味着就应该这么做。千万不要过分热衷于 ES6 的新技巧，使得你的代码的“聪明”程度超过了你自己或你的同事。

表达式作用域

这里对于用来在表达式中决议变量的作用域有一点简单说明。我在前面提到过插入字符串字面量有点像是一个 IIFE，这么看也能解释它的作用域特性。

考虑：

```
function foo(str) {
    var name = "foo";
    console.log( str );
}

function bar() {
    var name = "bar";
    foo( `Hello from ${name}!` );
}

var name = "global";

bar();                  // "Hello from bar!"
```

`..` 字符串字面值展开的时候，在函数 `bar()` 内部，它可用的作用域找到 `bar()` 的变量 `name` 的值 `"bar"`。全局 `name` 和 `foo()` 的 `name` 都不影响结果。换句话说，插入字符串字面量在它出现的词法作用域内，没有任何形式的动态作用域。

2.7.2 标签模板字面量

这里再次重新命名这个特性来明确表达其功能：**标签字符串字面量** (tagged string literal)。

老实说，这是 ES6 提供的一个比较酷的技巧。可能看起来有点奇怪，也可能一眼看上去不是那么实用。但一旦花费一定时间去理解这个功能，标签字符串字面量的用处可能会令你大吃一惊。

举例来说：

```
function foo(strings, ...values) {
    console.log( strings );
    console.log( values );
}
```

```
var desc = "awesome";

foo`Everything is ${desc}!`;
// [ "Everything is ", "!"]
// [ "awesome" ]
```

我们花点时间来思考一下前面代码中到底发生了什么。首先跳出的最不和谐的部分是 foo`Everything...`;。这种形式之前没有出现过。这是什么？

本质上说，这是一类不需要（ .. ）的特殊函数调用。**标签**（tag）部分，即 `..` 字符串字面量之前的 foo 这一部分，是一个要调用的函数值。实际上，它可以是任意结果为函数的表达式，甚至可以是一个结果为另一个函数的函数调用，就像下面这样：

```
function bar() {
    return function foo(strings, ...values) {
        console.log( strings );
        console.log( values );
    }
}

var desc = "awesome";

bar()`Everything is ${desc}!`;
// [ "Everything is ", "!"]
// [ "awesome" ]
```

但是传入为了字符串字面量作为标签被调用的 foo(..) 函数的是什么呢？

第一个参数，名为 strings，是一个由所有普通字符串（插入表达式之间的部分）组成的数组。得到的 strings 数组中有两个值："Everything is" 和 "!"。

我们的例子中使用 ...gather/rest 运算符把其余所有参数值收集到名为 values 的数组中（参见 2.2 节），这是为了方便起见，当然也可以把 strings 参数后面的其余部分都作为独立命名的参数。

收集到 values 数组的参数是已经求值的在字符串字面值中插入表达式的结果。所以显然我们的例子中 values 的唯一元素是 "awesome"。

你可以这样看待这两个数组：values 中的值是分隔符，就好像用它们连接在 strings 中的值，然后把所有这些都连接到一起，就得到了一个完成的插入字符串值。

标签字符串字面量就像是一个插入表达式求值之后，在最后的字符串值编译之前的处理步骤，这个步骤为从字面值产生字符串提供了更多的控制。

一般来说，字符串字面量标签函数（前面代码中的 foo(..)）要计算出一个适当的字符串并将其返回，这样就可以像使用非标签字符串字面量一样把标签字符串字面量作为一个值来使用了：

```
function tag(strings, ...values) {
    return strings.reduce( function(s,v,idx){
        return s + (idx > 0 ? values[idx-1] : "") + v;
    }, "" );
}

var desc = "awesome";

var text = tag`Everything is ${desc}!`;

console.log( text );            // Everything is awesome!
```

在这段代码中，tag(..)是一个直通操作，因为它不执行任何具体修改，而只是使用reduce(..)进行循环，把strings和values连接到一起，就像非标签字符串字面量所做的一样。

那么有哪些实际应用呢？有许多高级应用已经超出了本部分的讨论范围。但是，这里还是给出了一个简单的思路用来把数字格式化为美元表示法（类似于简单的本地化）：

```
function dollabillsyall(strings, ...values) {
    return strings.reduce( function(s,v,idx){
        if (idx > 0) {
            if (typeof values[idx-1] == "number") {
                // 看，这里也使用了插入字符串字面量！
                s += `$${values[idx-1].toFixed( 2 )}`;
            }
            else {
               s += values[idx-1];
            }
        }

        return s + v;
    }, "" );
}

var amt1 = 11.99,
    amt2 = amt1 * 1.08,
    name = "Kyle";

var text = dollabillsyall
`Thanks for your purchase, ${name}! Your
product cost was ${amt1}, which with tax
comes out to ${amt2}.`

console.log( text );
// Thanks for your purchase, Kyle! Your
// product cost was $11.99, which with tax
// comes out to $12.95.
```

如果在values中遇到一个number值，就在其之前放一个"$"，然后用toFixed(2)把它格式化为两个十进制数字的形式，否则就让这个值直接通过而不做任何修改。

原始（raw）字符串

在前面的代码中，标签函数接收第一个名为 strings 的参数，这是一个数组。但是还包括了一些额外的数据：所有字符串的原始未处理版本。可以像下面这样通过 .raw 属性访问这些原始字符串值：

```
function showraw(strings, ...values) {
    console.log( strings );
    console.log( strings.raw );
}

showraw`Hello\nWorld`;
// [ "Hello
// World" ]
// [ "Hello\nWorld" ]
```

原始版本的值保留了原始的转义码 \n 序列（\ 和 n 是独立的字符），而处理过的版本把它当作是一个单独的换行符。二者都会应用前面提到过的行结束标准化过程。

ES6 提供了一个内建函数可以用作字符串字面量标签：String.raw(..)。它就是传出 strings 的原始版本：

```
console.log( `Hello\nWorld` );
// Hello
// World

console.log( String.raw`Hello\nWorld` );
// Hello\nWorld

String.raw`Hello\nWorld`.length;
// 12
```

字符串字面量标签的其他应用包括全球化、本地化等的特殊处理。

2.8 箭头函数

本章前面已经介绍了一些函数中 this 绑定的复杂性，同时本系列《你不知道的 JavaScript（上卷）》第二部分中对此也有详细介绍。理解使用普通函数基于 this 编程带来的令人沮丧的问题是很重要的，因为这是新的 ES6 箭头函数 => 特性引入的主要动因。

让我们先来展示一下与普通函数相比箭头函数是什么样子：

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

// 对比

var foo = (x,y) => x + y;
```

箭头函数定义包括一个参数列表（零个或多个参数，如果参数个数不是一个的话要用 (..) 包围起来），然后是标识 =>，函数体放在最后。

所以，在前面的代码中，箭头函数就是 (x,y) => x + y 这一部分，然后这个函数引用被赋给变量 foo。

只有在函数体的表达式个数多于 1 个，或者函数体包含非表达式语句的时候才需要用 { .. } 包围。如果只有一个表达式，并且省略了包围的 { .. } 的话，则意味着表达式前面有一个隐含的 return，就像前面代码中展示的那样。

这里列出了几种不同形式的箭头函数：

```
var f1 = () => 12;
var f2 = x => x * 2;
var f3 = (x,y) => {
    var z = x * 2 + y;
    y++;
    x *= 3;
    return (x + y + z) / 2;
};
```

箭头函数**总是**函数表达式；并不存在箭头函数声明。我们还应清楚箭头函数是匿名函数表达式——它们没有用于递归或者事件绑定 / 解绑定的命名引用——但 7.1 节将会讨论 ES6 中用于调试目的的函数名推导规则。

箭头函数支持普通函数参数的所有功能，包括默认值、解构、rest 参数，等等。

箭头函数语法清晰简洁，这使它们表面上看起来对于编写更简练的代码很有吸引力。于是，几乎所有关于 ES6 的文献（除了本系列）似乎都立即且没有异议地接受了箭头函数作为“新函数”。

可以说这里关于箭头函数的讨论中几乎所有的例子都是简短的单句工具，比如作为回调函数传递给各种工具的那些。举例来说：

```
var a = [1,2,3,4,5];

a = a.map( v => v * 2 );

console.log( a );               // [2,4,6,8,10]
```

这些例子中你使用了这样的在线函数表达式，它们也符合在单个语句中执行一个快速计算并返回结果的模式，这时候比起繁复的 function 关键字和语法，箭头函数确实看起来是更有吸引力的轻量替代工具。

大多数人会**赞叹于**这些示例的简洁，我猜你也是这样！

但是这里我要提醒你，使用箭头函数语法替代其他普通的多行函数，特别是那些通常会被自然表达为函数声明的情况，是不合理的。

回忆一下本章前面的 `dollabillsyall(..)` 字符串字面量标签函数，把它替换为使用 `=>` 语法：

```
var dollabillsyall = (strings, ...values) =>
    strings.reduce( (s,v,idx) => {
        if (idx > 0) {
            if (typeof values[idx-1] == "number") {
                // 看，这里也使用了插入字符串字面量！
                s += `$${values[idx-1].toFixed( 2 )}`;
            }
            else {
                s += values[idx-1];
            }
        }

        return s + v;
    }, "" );
```

在这个例子中，我所做的唯一修改就是去掉了 `function`、`return` 和一些 `{ .. }`，然后插入了 `=>` 和 `var`。对于这段代码来说，可读性有了明显的改进吗？

实际上，我认为缺少 `return` 和外层的 `{ .. }` 一定程度上模糊了 `reduce(..)` 调用是 `dollabillsyall(..)` 函数中唯一的语句，以及它的返回值就是调用的返回值这一事实。另外，有经验的人会在代码中搜索单词 `function` 来寻找作用域的边界，现在则需要寻找 `=>` 标识，这在大段代码中肯定更加难以发现。

虽然不是一条严格的规律，但我认为 `=>` 箭头函数转变带来的可读性提升与被转化函数的长度负相关。这个函数越长，`=>` 带来的好处就越小；函数越短，`=>` 带来的好处就越大。

我认为更合理的做法是只在确实需要简短的在线函数表达式的时候才采用 `=>`，而对于那些一般长度的函数则无需改变。

不只是更短的语法，而是 this

对 `=>` 的关注多数都在于从代码中去掉 `function`、`return` 和 `{ .. }` 节省了那些宝贵的键盘输入。

但是，目前为止我们省略了一个重要的细节。这一节开头提到 `=>` 函数与 `this` 绑定行为紧密相关。实际上，`=>` 箭头函数的**主要设计目的**就是以特定的方式改变 `this` 的行为特性，解决 `this` 相关编码的一个特殊而又常见的痛点。

节省的输入字符是一条红鲱鱼（转移注意力的东西），至少是误导性的。

让我们回顾一下本章前面的另外一个例子：

```
var controller = {
    makeRequest: function(..){
        var self = this;

        btn.addEventListener( "click", function(){
            // ..
           self.makeRequest(..);

        }, false );
    }
};
```

我们使用了 `var self = this` 这一 hack，然后引用 `self.makeRequest(..)`，因为在我们传入 `addEventListener(..)` 的回调函数内部，`this` 绑定和 `makeRequest(..)` 本身的 `this` 绑定是不同的。换句话说，因为 `this` 绑定是动态的，我们通过变量 `self` 依赖于词法作用域的可预测性。

这里我们终于可以看到 `=>` 箭头函数的主要设计特性了。在箭头函数内部，`this` 绑定不是动态的，而是词法的。在前面的代码中，如果使用箭头函数作为回调，`this` 则如我们所愿是可预测的。

考虑：

```
var controller = {
    makeRequest: function(..){
        btn.addEventListener( "click", () => {
            // ..
            this.makeRequest(..);
        }, false );
    }
};
```

前面代码的箭头函数回调中的词法 `this` 现在与封装的函数 `makeRequest(..)` 指向同样的值。换句话说，`=>` 是 `var self = this` 的词法替代形式。

在通常需要 `var self = this`（或者换种形式，如函数 `.bind(this)` 调用）的时候，基于运行原则相同，可以把 `=>` 箭头函数作为一个很好的替代。看起来不错，是吗？

但没有这么简单。

假使用 `=>` 替代 `var self = this` 或者 `.bind(this)` 的情况有所帮助，那么猜一下如果在一个支持 `this` 的函数中使用 `=>`，而这个函数**不**需要 `var self = this` 会怎样呢？你可能已经猜到了，这会把事情搞乱。没错。

考虑：

```
var controller = {
    makeRequest: (..) => {
        // ..
        this.helper(..);
    },
    helper: (..) => {
        // ..
    }
};

controller.makeRequest(..);
```

尽管我们以 `controller.makeRequest(..)` 的形式调用，`this.helper` 引用还是会失败，因为这里的 `this` 并不像平常一样指向 `controller`。那么它指向哪里呢？它是从包围的作用域中词法继承而来的 `this`。在前面的代码中也就是全局作用域，其中 `this` 指向那个全局对象。

除了词法 `this`，箭头函数还有词法 `arguments`——它们没有自己的 `arguments` 数组，而是继承自父层——词法 `super` 和 `new.target` 也是一样（参见 3.4 节）。

所以现在我们可以给出一组更详细的 `=>` 适用时机的规则。

- 如果你有一个简短单句在线函数表达式，其中唯一的语句是 `return` 某个计算出的值，**且**这个函数内部没有 `this` 引用，**且**没有自身引用（递归、事件绑定 / 解绑定），**且**不会要求函数执行这些，那么可以安全地把它重构为 `=>` 箭头函数。
- 如果你有一个内层函数表达式，依赖于在包含它的函数中调用 `var self = this` hack 或者 `.bind(this)` 来确保适当的 `this` 绑定，那么这个内层函数表达式应该可以安全地转换为 `=>` 箭头函数。
- 如果你的内层函数表达式依赖于封装函数中某种像 `var args = Array.prototype.slice.call(arguments)` 来保证 `arguments` 的词法复制，那么这个内层函数应该可以安全地转换为 `=>` 箭头函数。
- 所有的其他情况——函数声明、较长的多语句函数表达式、需要词法名称标识符（递归等）的函数，以及任何不符合以上几点特征的函数——一般都应该避免 `=>` 函数语法。

底线：`=>` 是关于 `this`、`arguments` 和 `super` 的词法绑定。这个 ES6 的特性设计用来修正一些常见的问题，而不是 bug、巧合或者错误。

不要相信那些宣传所说的 `=>` 主要甚至绝大多数关注点在于少打字。不管是少打字还是多打字，你应该精确了解自己输入的每个字符的目的所在。

如果你有一个函数，出于明确的原因其不适合 `=>` 箭头函数，但是它被声明为对象字面量的一部分，回忆一下 2.6.2 节的内容，要使函数语法简洁还有其他可选的方法。