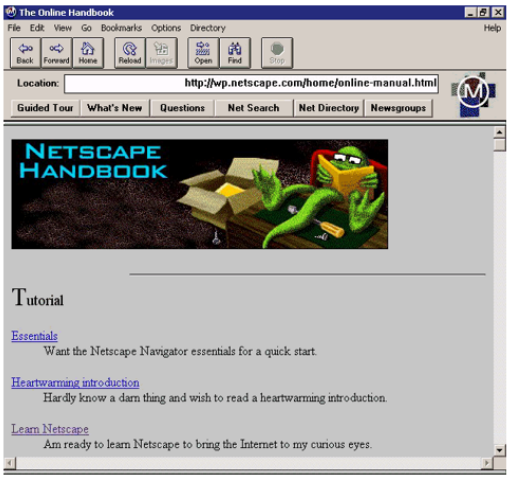
# 你真的了解模板字符串么?

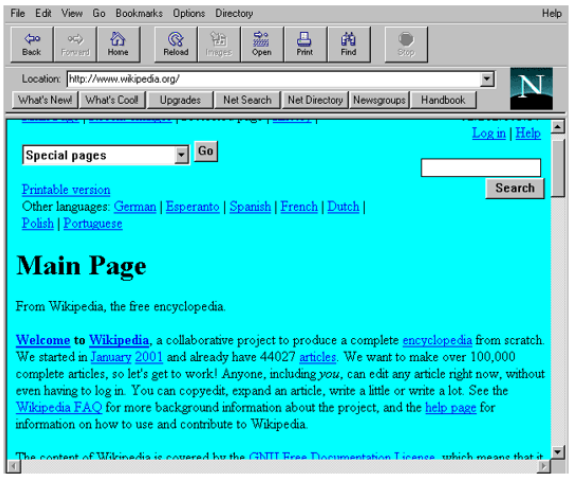
## 首先大家要明白，我们为什么需要模板?

在生活中，我们去银行签一些单子，或者生活中写一些请假条，我们都会有一个模板进行参照，哪里哪里需要怎么填写，哪里自由发挥。就是一种格式的提现。在我们前端中，这个模版又是什么样的概念呢?

模板这个概念起于浏览器页面初期。这个时候的页面是怎么形成的呢？并不是大家现在所学的这个样子（html+css+js）,因为那个时候还没有 js，最初期的页面都是基本的 html 构成，用于学术交流使用的而且都是很丑的，像这样:




网景浏览器前身Mosaic Netscape




网景浏览器V-1.22

当时的页面，每篇文档对应的都是一个 html 页面。随着交互需求的增多，以及高级编程思想的普及，我们页面中的内容也需要适当的交互，与模块化展示。这时我们需要操作 DOM 但是不是由 JS 来做的，那个时期 JS 刚刚出现还没有成型，很多事情都做不了。这时后端成熟的语言 Java 出现很久了。我们想进行的页面计算，都利用 Java 的语法去写，**同时将页面固定不变的内容抽出成模板，服务端返回的动态数据装填到模板中预留的坑位，最后组装成完整的页面 html 字符串交给浏览器去解析，也就是大家在大学时期接触过得 JSP 模版**，像下面这样:

```
<%
ArrayList<Object> goodsList = (ArrayList<Object>)request.getAttribute("goodsList");
    %>
    <form id="form" action="" method="post">
        <table align="center" width="60%" border="1" cellspacing="0">
            <%
            for(Object o:goodsList){
            Goods goods = (Goods)o;
            out.write("<tr>");
            out.write("<td><input type='checkbox' name='gid' value='"+goods.getGid()+"'></td>");
```

```
                    out.write("<td>"+goods.getGname()+"</td>");
                    out.write("<td>"+goods.getGprice()+"</td>");
                    out.write("</tr>");
                }
            %>
            <tr>
                <td colspan="3" align="center">
                    <input class="submitButton" type="button" value="continue">
                        
                    <input class="submitButton" type="button" value="payoff">
                </td>
            </tr>
        </table>
        <input id="flagHidden" type="hidden" name="flag" value="">
    </form>
```

这里的代码，大家仔细看会发现是 html 与 java 代码综合去写出来。也就是利用 java 的逻辑代码（循环，判断，等等）+ HTML 的基本结构来构成基本的页面。这个是最早的模板。目的在于将一些可变的地方预留出来，每次都利用这个模板，只不过内容不一样而已。来增加页面的可重用性。**这个时期的模板引擎，往往是服务端来编译模板字符串，生成 html 字符串给客户端，也被称为服务器端渲染**。

直到 09 年 Node 发布之后，Javascript 可以来实现服务端的功能。mustache 和 handlebar 模板的诞生方便了前端开发人员，这两个模板均使用 JavaScript 来实现，从此前端模板既可以在服务端运行，也可以在客户端运行，但是大多数使用场景都是 js 根据服务端异步获取的数据套入模板，生成新的 dom 插入页码， 对前端后端开发都非常有利。

```
Hello {{name}}
You have just won {{value}} dollars!
{{#in_ca}}
Well, {{taxed_value}} dollars, after taxes.
{{/in_ca}}
```

配合着对象

```
{
“name”: “duyi”,
“value”: 10000,
“taxed_value”: 10000 - (10000 * 0.4),
“in_ca”: true
}
```

会解析成

```
Hello suyi
You have just won 10000 dollars!
Well, 6000.0 dollars, after taxes.
```

接下来到了新生代，vue 中的模板写法跟之前的模板有所不同，而且功能更加强大。既可以在客户端使用也可以在服务端使用，但是使用场景上差距非常大：页面往往根据数据变化，模板生成的 dom 发生变化，这对于模板的性能要求很高。如果想了解更多的 Vue 的知识，

请关注我们后面的课程。

无论是模板的提出以及使用,都是为了的使得我们页面搭建效率变得更高，以及页面的复用性更好。那么在我们的 javaScript 中是怎么做的，脱离了以上模板的形式，我们要怎么做呢 ？初期阶段，我们想动态的添加 Dom 元素，通常的做法是先分析页面逻辑，分析出我们要处理什么内容，以及 DOM 之间的嵌套关系。然后我们通过数据（数据来源可以是本地数据，ajax 数据，mock 数据都可以）配合着相应的逻辑通过字符串拼接的方式书写 HTML 字符串处理之后的字符串最后插入到页面当中，像这样：

```
var html = "<div> \
                <div> name:" + name + "</div> \
                <div> age:" + age + "</div> \
                <div> sex:" + sex + "</div> \
            </div>"
```

以上内容，相信大家都去写过，不难发现难用，而且容易出错，还需要转义等等需要注意的点。如果我们可以有一个类似的模板，在这样的模板中去添加数据岂不是很好例如 Jquery.clone 方法：

```
var $Clone = $Wrapper.find('.tpl').clone().removeClass('tpl').addClass('showItem');
                var ele = data[i + curPage * 10];
                $Clone.children('span').eq(0)
                .text(i + curPage * 10 + 1)
                .css('backgroundColor', curPage == 0 && colorsArray[i + curPage])
                  .next()
                  .text(ele.title)
                  .next(ele.search)
                  .addClass(ele.search > ele.hisSearch ? 'up' : 'down');
                  $ShowSection.append($Clone);
```

Jquery 中借助已有的内容进行重新渲染，把已有的内容作为的小模板，把数据按照我们自己的逻辑添加进去，最后在放到页面当中。这种模式相对与原始的字符串拼接使用起来更舒服，更容易。减少报错率，但是并不直接。

接下来就进入到的自己去编写一个方法去实现类似模板的功能，包括上面的一些已有工具都是基于这种模板思想。如何编写一个模板在下方会个大家阐明。既然这种模板思想如此美妙，而且很多人通过不同的方式来实现它，期望用到这样的模板，于是在 ES6 中提出了模板字符串。

# ES6 中的模板字符串

ES6（ES2015）为 JavaScript 引入了许多新特性，其中与字符串处理相关的一个新特性——模板字面量，提供了多行字符串、字符串模板的功能，相信很多人已经在使用了。模板字面量的基本使用很简单，但大多数开发者还是仅仅把它当成字符串拼接的语法糖来使用的，实际上它的能力比这要强大得多哦。夸张一点地说，这可能是 ES6 这么多特性中，最容易被低估的特性了。Here is why。

基础特性

模板字面量在 ES2015 规范中叫做 Template Literals，在规范文档更早的版本中叫 Template Strings，所以我们见过的中文文档很多也有把它写成 模板字符串 的，有时为表

述方便也非正式地简称为 ES6 模板。

在 ES6 之前的 JavaScript，字符串作为基本类型，其在代码中的表示方法只有将字符串用引号符（单引号 ' 或 双引号 "）包裹起来，ES6 模板字面量（下文简称 ES6 模板）则使用反撇号符（`）包裹作为字符串表示法。

两个反撇号之间的常规字符串保持原样，如：

```
`hello world` === "hello world"  // --> true
`hello "world"` === 'hello "world"' // --> true
`hello 'world'` === "hello 'world'" // --> true
`\`` // --> "`" // --> true
```

换行符也只是一个字符，因此模板字面量也自然就支持多行字符 ：

```
console.log(`TODO LIST:
    * one
    * two
`)
// TODO LIST:
//     * one
//     * two
```

两个反撇号之间以 ${expression} 格式包含任意 JavaScript 表达式，该 expression 表达式的值会转换为字符串，与表达式前后的字符串拼接。expression 展开为字符串时，使用的是 expression.toString() 函数。

```
const name = "Alice"
const a = 1
const b = 2
const fruits = ['apple', 'orange', 'banana']
const now = new Date()
console.log(`Hello ${name}`)
console.log(`1 + 2 = ${a + b}`)
console.log(`INFO: ${now}`)
console.log(`Remember to bring: ${ fruits.join(', ') }.`)
console.log(`1 < 2 ${ 1 < 2 ? '✔' : '✘'}`)
// Hello Alice
// 1 + 2 = 3
// INFO: Sun May 13 2018 22:28:26 GMT+0800 (中国标准时间)
// Remember to bring: apple, orange, banana.
// 1 < 2 ✔
```

正因为 expression 可以是 任意 JavaScript 表达式 ，而任意一个模板字符串本身也是一个表达式，所以模板中的 expression 可以嵌套另一个模板。

```
const fruits = ['apple', 'orange', 'banana']
const quantities = [4, 5, 6]
console.log(`I got ${
    fruits
        .map((fruit, index) => `${quantities[index]} ${fruit}s`)
        .join(', ')
}`)
```

```
// I got 4 apples, 5 oranges, 6 bananas
```

**与传统模板引擎对比**

从目前的几个示例，我们已经掌握了 ES6 模板的基础功能，但已足够见识到它的本领。通过它我们可以很轻易地进行代码重构，让字符串拼接的代码不再充满乱七八糟的单引号、双引号、+ 操作符还有反斜杠 \，变得清爽很多。

于是我们很自然地想到，在实际应用中字符串拼接最复杂的场景——HTML 模板上，如果采用 ES6 模板是否可以胜任呢？传统上我们采用专门的模板引擎做这件事情，不妨将 ES6 模板与模版引擎做对比。我们选择　的 _.template 模板引擎，这个引擎虽不像 mustache、pug 大而全，但提供的功能已足够完备，我们就从它的几个核心特性和场景为例，展开对比。

1，基本的字符串插值。

_.template 使用 <%= expression %> 作为模板插值分隔符，expression 的值将会按原始输出，与 ES6 模板相同。所以在这一个特性上，ES6 模板是完全胜任的。

```
const compiled = _.template('hello <%= user %>!')
console.log(compiled({ user: 'fred' }))
// hello fred
```

ES6 模板

```
const greeting = data => `hello ${data.user}`
console.log(greeting({ user: 'fred' }))
// hello fred
```

2，字符串转义输出。

这是 HTML 模板引擎防范 XSS 的标配功能，其原理就是要将插值表达式的值中包含的 <、> 这种可能用于 XSS 攻击的字符转义为 HTML Entity。要让　输出转义后的插值表达式，使用 <%- expression %> 语法即可；而如果要使用 ES6 模板方案，就要靠自己实现一个单独的函数调用了。在下面的示例代码中，就定义了简单的 escapeHTML 函数，在模板字符串中调用该函数对字符串进行转义。

在这个特性上，ES6 可以的确可以做到相同的效果，但代价是要自己定义转义函数并在表达式中调用，使用起来不如模板引擎封装好的接口方便。

```
const compiled = _.template('<b><%- value %></b>')
```

ES6 模板

```
const entityMap = {
    '&': '&amp;',
    '<': '&lt;',
    '>': '&gt;',
    '"': '&quot;',
```

```
    ''': '&#39;',
    '/': '&#x2F;',
    '`': '&#x60;',
    '=': '&#x3D;'
}
const escapeHTML = string => String(string).replace(/[&<>"'`=\/]/g, (s) => entityMap[s]);
const greeting = data => `hello ${escapeHTML(data.user)}`
console.log(greeting({ user: '<script>alert(0)</script>'}));
// hello &lt;script&gt;alert(0)&lt;&#x2F;script&gt;
```

3，模板内嵌 JavaScript 语句。

也就是模板引擎支持通过在模板中执行 JavaScript 语句，生成 HTML。说白了其原理与世界上最好的语言 php 的 idea 是一样的。在 模板中，使用 <% statement %> 就可以执行 JS 语句，一个最典型的使用场景是使用 for 循环在模版中迭代数组内容。

但 ES6 模板中的占位符 ${} 只支持插入表达式，所以要在其中直接执行 for 循环这样的 JavaScript 语句是不行的。但是没关系，同样的输出结果我们用一些简单的技巧一样可以搞定，例如对数组的处理，我们只要善用数组的 map、 reduce、filter ，令表达式的结果符合我们需要即可。

```
/* 使用 for 循环语句，迭代数组内容并输出 */
const compiled = _.template('<ul><% for (var i = 0; i < users.length; i++) { %><li><%= list[i] %></li><% } %></ul>')
console.log(compiled({ users: ['fred', 'barney'] }))
// <ul><li>fred</li><li>barney</li></ul>
```

ES6 模板

```
const listRenderer = data => `<ul>${data.users.map(user => `<li>${user}</li>`).join('')}</ul>`
console.log(listRenderer({ users: ['fred', 'barney']}))
// <ul><li>fred</li><li>barney</li></ul>
```

在以上这 3 个示例场景上，我们发现 模板能做的事，ES6 模板也都可以做到，那是不是可以抛弃模板引擎了呢？

的确，如果在开发中只是使用以上这些基本的模板引擎功能，我们可以确实可以直接使用 ES6 模板做替换，API 更轻更简洁，还节省了额外引入一个模板库的成本。

但如果我们使用的是 pug、artTemplate、Handlebars 这一类大而全的模板引擎，使用 ES6 模板替换就不一定明智了。尤其是这些模板引擎在服务器端场景下的模板缓存、管道输出、模板文件模块化管理等特性，ES6 模板本身都不具备。并且模板引擎作为独立的库，API 的封装和扩展性都比 ES6 模板中只能插入表达式要好。

**走向进阶——给模板加上标签**

截至目前介绍的特性，我们可以从上面 HTML 模板字符串转义和数组迭代输出两个例子的代码发现这样一个事实：

要用 ES6 模板实现复杂一点的字符串处理逻辑，要依赖我们写函数来实现。
幸运的是，除了在模板的插值表达式里想办法调用各种字符串转换的函数之外，ES6 还提供了更加优雅且更容易复用的方案——带标签的模板字面量（tagged template literals，以下简称标签模板）。
标签模板的语法很简单，就是在模板字符串的起始反撇号前加上一个标签。这个标签当然不是随便写的，它必须是一个可调用的函数或方法名。加了标签之后的模板，其输出值的计算过程就不再是默认的处理逻辑了，我们以下面这一行代码为例解释。
const message = l10n`I bought a ${brand} watch on ${date}, it cost me ${price}.`
在这个例子里，l10n 就是标签名，反撇号之间的内容是模板内容，这一行语句将模板表达式的值赋给 message，具体的处理过程为：

JS 引擎会先把模板内容用占位符分割，把分割得到的字符串存在数组 strings 中（以下代码仅用来演示原理）
const strings = "I bought a ${brand} watch on ${date}, it cost me ${price}".split(/\$\{[^}]+\}/)
// ["I bought a ", " watch on ", ", it cost me ", "."]
然后再将模板内容里的占位符表达式取出来，依次存在另一个数组 rest 中
const rest = [brand, date, price]
执行 i18n(strings, ...rest) 函数，即调用 l10n，并传入两部分参数，第一部分是 strings 作为第一个参数，第二部分是将 rest 展开作为余下参数。
const message = l10n(strings, ...rest)
因此，如果将以上单步分解合并在一起，就是这样的等价形式：
const message = l10n(["I bought a ", " watch on ", ", it cost me ", "."], brand, date, price)
也就是说当我们给模板前面加上 l10n 这个标签时，实际上是在调用 l10n 函数，并以上面这种方式传入调用参数。 l10n 函数可以交给我们自定义，从而让上面这一行代码👆输出我们想要的字符串。例如我们如果想让其中的日期和价格用本地字符串显示，就可以这样实现：

```
const l10n = (strings, ...rest) => {
    return strings.reduce((total, current, idx) => {
        const arg = rest[idx]
        let insertValue = ''
        if (typeof arg === 'number') {
            insertValue = `¥${arg}`
        } else if (arg instanceof Date) {
            insertValue = arg.toLocaleDateString('zh-CN')
        } else if (arg !== undefined) {
            insertValue = arg
        }
        return total + current + insertValue
    }, '');
}
const brand = 'G-Shock'
const date = new Date()
const price = 1000

l10n`I bought a ${brand} watch on ${date}, it cost me ${price}.`
```

// I bought a G-Shock watch on 2018/5/16, it cost me ¥1000.
这里的 l10n 就是个简陋的傻瓜式的本地化模板标签，它支持把模板内容里的数字当成金额加上人民币符号，把日期转换为 zh-CN 地区格式的 2018/5/16 字符串。
乍一看没什么大不了的，但设想一下相同的效果用没有标签的模板要怎样实现呢？我们需要在模板之内的日期表达式和价格数字表达式上调用相应的转换函数，即 ${date.toLocaleDateString('zh-CN')} 和 ${ '¥' + price} 。一次调用差别不大，两次、三次调用的情况下，带标签的模板明显胜出。不仅符合 DRY（Don't Repeat Yourself）原则，也可以让模板代码更加简洁易读。
超越模板字符串
带标签的模板字面量建立在非常简单的原理上——通过自己的函数，自定义模板的输出值。
tag`template literals`
而 ES6 规范没有对这里可以使用的 tag 函数做任何限制，意味着 任何函数 都可以作为标签加到模板前面，也就意味着这个函数：

可以是立即返回的同步函数，也可以是异步的 async 函数（支持函数内 await 异步语句并返回 Promise）
可以是返回字符串的函数，也可以是返回数字、数组、对象等任何值的函数
可以是纯函数，也可以是有副作用的非纯函数

所以只要你愿意，A：你可以把 任意 JavaScript 语句 放到标签函数里面去；B：你可以让标签函数返回 任意值 作为模板输出。有了如此强大的扩展能力，也难怪一开始 ES6 标准中对模板规范的命名是 Template Strings，后来正式命名却改成了 Template Literals，因为它的能力已经超越了模板字符串，去追求诗和远方了。
当然在实际的使用中还是应该保持理智，不能手里拿着锤子看什么都像钉子。而以下这几种应用场景，倒是可以算是真正的钉子。

用例 1：使用 String.raw 保留原始字符串
String.raw 是 ES2015 规范新增的 String 对象的静态成员方法，但通常并不作为函数直接调用，而是作为语言标准自带的模板字面量标签使用，用来保留模板内容的原始值。
其作用与 Python 语言中，字符串表达式的引号前加 r 前缀效果类似。JavaScript 需要这样的特性，只是刚好用 String.raw 实现了它。
```
var a = String.raw`\n\n`
var b === '\n\n'
// a === b -> false
// a.length === 4 -> true
// b.length === 2 -> true
```
String.raw 标签在某些场景下可以帮我们节省很多繁琐的工作。其中一个典型的场景就是，实际开发中我们常遇到在不同编程语言之间通过 JSON 文本传输协议数据的情况，如果遇到包含转义字符和 " (JSON 属性需用引号)的文本内容，很容易因细节处理不当导致 JSON 解析出错。
一个真实案例，就是由 Android 终端向 JavaScript 传输 JSON 数据时，有一条数据中用户的昵称包含了 " 字符。JavaScript 收到的 JSON 字符串文本为：
```
{"nickname":"枪锅&[{锅}]\"锅\":锅","foo":"bar"}
```
但这里如果直接将内容用引号赋值给 JSON.parse('{"nickname":"枪锅&[{锅}]\"锅\":锅

","foo":"bar"}') 就会遇到 SyntaxError: Unexpected token 锅 in JSON at position 22 的错误，因为单引号中的 \ 被作为转义字符，未保留在 input 的值中。要得到正确的 JSON 对象，使用 String.raw 处理即可：

```
JSON.parse(String.raw`{"nickname":"枪锅&[{锅}]\"锅\":锅","foo":"bar"}`)
// { nickname: '枪锅&[{锅}]"锅":锅', foo: 'bar' }
```

用例 2：从标签模板到 HTML 字符串

前面讲到 ES6 模板与模板引擎的对比时，提到模板引擎通过手动 escapeHTML 模板转义不安全的字符的问题。现在我们了解了标签模板之后，可以将外部定义的 escapeHTML 逻辑直接放到标签函数中，这样就不需要在模板中每一个插入表达式前，都调用 escapeHTML 函数了。

```
const safeHTML = (strings, ...rest) => {
    const entityMap = {
        '&': '&amp;',
        '<': '&lt;',
        '>': '&gt;',
        '"': '&quot;',
        "'": '&#39;',
        '/': '&#x2F;',
        '`': '&#x60;',
        '=': '&#x3D;'
    }
    const escapeHTML = string => String(string).replace(/[&<>"'`=\/]/g, (s) => entityMap[s]);
    return strings.reduce((total, current, idx) => {
        const value = rest[idx] || ''
        return total + current + escapeHTML(value)
    }, '');
}

const evilText = '<script>alert(document.cookie)</script>'
safeHTML`${evilText}`
// "&lt;script&gt;alert(document.cookie)&lt;&#x2F;script&gt;"
```

我们这里实现的 safeHTML 作为 demo 用，并不保证生产环境完备。

你一定想到了，像 HTML 模板这样的常用模板标签一定有现成的 npm 库了。没错，common-tags 库就是我们想要的这个库了。common-tags 是一个小而精的模板标签库，被包括 Angular, Slack, Ember 等在内的很多大型项目所依赖。它包含了十几个常用的用于字符串模板处理的函数，例如 html, safeHTml 等，让我们可以偷懒少造一些轮子。

```
const safeHtml = require('common-tags/lib/safeHtml')
const evilText = '<script>alert(document.cookie)</script>'
safeHtml`<div>${evilText}</div>`
```

common-tags 也提供了扩展 API，生成可以让我们更轻松地实现自定义的标签。

用例 3：DSL

DSL（领域专用语言）是一个挺唬人的概念，顾名思义所谓 DSL 是指未解决某一特定领域

的问题推出的语言。DSL 有按某种条件约束的规则——语法，故称得上是语言。通常 DSL 用于编程领域，通过计算机程序处理 DSL，有些功能强大的 DSL 语言也会被认为是 mini 编程语言。
典型的 DSL 有：

正则表达式
数据库查询语言（SQL）
CSS Selector 的规则也是一种 DSL
文本处理的瑞士军刀 awk，sed 因为其复杂的使用规则也被认为是 DSL

合理使用 ES6 标签模板，可以让 JavaScript 对内嵌 DSL 的处理更加简洁。严格来说，我们上面用例 2 和用例 3 对中的 html 类模板标签，就算是 DSL 的范畴了。下面是一些典型的案例：

注意：以下模板标签仅作描述，需要自行实现

案例 1：DOM 选择器
例如我们可以用标签实现一个 DOM 选择器，形如：
var elements = query`.${className}` // the new way
虽然第一眼看上去只是另一个 jQuery 选择器，但这种调用方式天然支持我们在模板之中嵌入任意的插值表达式，在 API 的 expressive 这个特性上有提升。而且 query 函数由我们自由实现，意味着可以自由地在 query 函数中加入 selector 值规范化校验，返回值规范化等额外功能。
案例 2：Shell 脚本
在 Node.js 环境下，我们还可以实现一个 sh 标签用于描述 shell 脚本的调用：
var proc = sh`ps aux | grep ${pid}`
看到这行语句我们下意识地会想到，这里的 API 会调用 shell 执行模板拼接出来的命令，而 proc 应该是该命令执行的结果或输出。
案例 3：正则表达式构造器
以 re 标签实现一个动态的正则表达式构造器，这种运行时生成的正则表达式，通常要自己定义函数，并调用 new RegExp 实现。
var match = input.match(re`\d+${separator}\d+`)
案例 4：国际化与本地化
可以实现这样一个 i18n 和 l10n 模板标签：约定在模板字符串中的插值表达式 ${expression} 后，以 :type 格式指定 expression 表达式期望的文本显示格式，实现自定义的模板输出功能。
var message = l10n`Hello ${name}; you are visitor number ${visitor}:n!
You have ${money}:c in your account!`;
这里的 l10n 标签与我们在上文 hard-code 的版本相比，增加了 :type 标识符以表示类别，例如 :n 表示数字，:c 表示货币。这些类型标识符的规则可以在 l10n 的实现代码中约定。而这个约定的意味就有点自行定义 DSL 的味道了。
以上 4 个 case 的共同点是，我们首先约定了有相似模式的 API 接口，它们都表现为带标签的模板的形式——一个模板名后跟模板内容。
虽然我们作为实现者知道，实际上在调用标签模板时，本质上是将模板内容重组为

(strings, ...rest) 形式再传给标签函数调用的。但这样的 API 调用看上去却很像是只有一个函数和一个参数，让人一眼看到就能猜出来 API 的用途。
好的 API 应当有良好的自我描述性，将复杂的实现细节封装起来，并且尽量专注做好一件事。从这个角度来说带标签的 ES6 模板非常适合处理 JS 内嵌的 DSL，甚至可以帮助我们在特定的业务逻辑中实现一个 mini DSL。
更多探索
以上就是对 ES6 模板语法和实用价值的介绍。讲到实践，得益于其原理的简洁，我们可以立即享受到它带来的好处。在 Node.js 环境下，毫无疑问我们可以立即使用不用迟疑；在浏览器环境下，使用我们的老朋友 Babel 就可以将其转换为兼容的 ES5 代码。
总结起来，ES6 模板中最激动人心的特性还是标签，小小的标签用简单的原理提供了异常丰富的扩展能力，颇有点四两拨千金的感觉。基于它，JavaScript 社区已经产生了很多新的想法并产生了很多实实在在的工具库。
有很多实现了特定领域功能的标签库，例如 SQL 相关的、国际化和本地化相关的，用 tagged template literals 这几个关键词在 npm 搜索就可以找到别人造的轮子。
但相比其他主题，社区关注量最大的探索还是集中在将模板字面量与 HTML 模板的结合上，有 3 个代表性的框架致力于采用 template literals 的方案并结合其他 good stuff 实现可以媲美 Virtual DOM 的快速渲染方案。

hyperHTML 旨在成为 Virtual DOM 替代品的仓库，在官方文档中明确直出，其核心理念就是采用 ES6 模板 作为 Virtual DOM Alternative
lit-html Google Polymer 团队推出的库，理念与 hyperHTML 类似，结合了 ES6 模板和 HTML <template> 元素的优点，实现 DOM 元素的快速渲染和更新
choojs 一个 minify+gzip 后只有 4KB 的现代前端开发框架，与 hyperHTML 和 lit-html 不同，是一个更加全功能的框架。

这 3 个框架的共同点是都采用了 tagged template literals，并且放弃使用 Virtual DOM 这种现在非常火爆的概念，但号称一样能实现快速渲染和更新真实 DOM。从各自提供的数据上看，也的确都有着不俗的表现。
毕竟模板字符串里面使用起来还是有一些限制的。
接下来我们利用模板的特性手动封装一个自己的模板

无论是从 JSP 到 vue 的模板，模板在语法上越来越简便，功能越来越丰富，但是基本功能是不能少的：
变量输出（转义/不转义）:出于安全考虑，模板基本默认都会将变量的字符串转义输出，当然也实现了不转义输出的功能，慎重使用。

条件判断（if else）：开发中经常需要的功能。
循环变量：循环数组，生成很多重复的代码片段。
模板嵌套：有了模板嵌套，可以减少很多重复代码，并且嵌套模板集成作用域。
以上功能基本涵盖了大多数模板的基础功能，针对这些基础功能就可以探究模板如何实现的。
模板实现原理
正如标题所说的，模板本质上都是纯文本的字符串，字符串是如何操作 js 程序的呢？
模板用法上：

var domString = template(templateString, data);
模板引擎获得到模板字符串和模板的作用域，经过编译之后生成完整的 DOM 字符串。
大多数模板实现原理基本一致：
模板字符串首先通过各种手段剥离出普通字符串和模板语法字符串生成抽象语法树 AST；然后针对模板语法片段进行编译，期间模板变量均去引擎输入的变量中查找；模板语法片段生成出普通 html 片段，与原始普通字符串进行拼接输出。
其实模板编译逻辑并没有特别复杂，至于 vue 这种动态绑定数据的模板有时间可以自行研究
这是我们在开始时可以拥有的：

```
var TemplateEngine = function(tpl, data) {
    // magic here ...
}
var template = '<p>Hello, my name is <%name%>. I\'m <%age%> years old.</p>';
console.log(TemplateEngine(template, {
    name: "Krasimir",
    age: 29
}));
```

一个简单的函数，它接受我们的模板和数据对象。正如您可能猜到的那样，我们最终想要达到的结果是：

```
<p>Hello, my name is Krasimir. I'm 29 years old.</p>
```

我们要做的第一件事就是在模板中采用动态块。稍后我们将用传递给引擎的真实数据替换它们。我决定使用正则表达式来实现这一目标。这不是我最强的部分，所以请随意评论并提出更好的 RegExp。

```
var re = /<%([^%>]+)?%>/g;
```

我们将捕获以<％开头并以％>结尾的所有部分。标志 g（全局）意味着我们不会获得一个，而是所有的匹配。有很多方法接受正则表达式。但是，我们需要的是一个包含字符串的数组。这就是 exec 所做的。

```
var re = /<%([^%>]+)?%>/g;
var match = re.exec(tpl);
```

如果我们在 console.log 中获得匹配变量，我们将得到：

```
[
    "<%name%>",
    " name ",
    index: 21,
    input:
    "<p>Hello, my name is <%name%>. I\'m <%age%> years old.</p>"
]
```

所以，我们得到了数据，但正如您所看到的，返回的数组只有一个元素。我们需要处理所有比赛。为此，我们应该将我们的逻辑包装到 while 循环中。

```
var re = /<%([^%>]+)?%>/g, match;
while(match = re.exec(tpl)) {
    console.log(match);
}
```
如果您运行上面的代码，您将看到显示<％name％>和<％age％>。

现在它变得有趣了。我们必须用传递给函数的实际数据替换占位符。我们可以使用的最简单的事情是对模板使用.replace 方法。我们可以写这样的东西：

```
var TemplateEngine = function(tpl, data) {
    var re = /<%([^%>]+)?%>/g, match;
    while(match = re.exec(tpl)) {
        tpl = tpl.replace(match[0], data[match[1]])
    }
    return tpl;
}
```
好的，这有效，但当然还不够。我们有非常简单的对象，并且很容易使用数据[“property”]。但实际上我们可能有复杂的嵌套对象。我们举个例子来改变我们的数据

```
{
    name: "Krasimir Tsonev",
    profile: { age: 29 }
}
```
这不起作用，因为当我们输入<％profile.age％>时，我们将得到数据[“profile.age”]，这实际上是未定义的。所以，我们需要别的东西。该.replace 方法将不起作用我们的情况。最好的方法是在<％和％>之间放置真正的 JavaScript 代码。如果根据传递的数据进行评估将会很好。例如：

```
var template = '<p>Hello, my name is <%this.name%>. I\'m <%this.profile.age%> years old.</p>';
```
这怎么可能？John 使用了新的 Function 语法。即从字符串创建函数。让我们看一个简单的例子。

```
var fn = new Function("arg", "console.log(arg + 1);");
fn(2); // outputs 3
```
fn 是一个真正的函数，它接受一个参数。它的主体是 console.log（arg + 1）；。换句话说，上面的代码等于：

```
var fn = function(arg) {
    console.log(arg + 1);
}
```

```
fn(2); // outputs 3
```
我们能够从简单的字符串定义一个函数，它的参数和它的主体。这正是我们所需要的。但在创建这样的功能之前，我们需要构建它的主体。该方法应返回最终编译的模板。让我们到目前为止使用的字符串，并试着想象它会是什么样子。

```
return
"<p>Hello, my name is " +
this.name +
". I\'m " +
this.profile.age +
" years old.</p>";
```
当然，我们会将模板拆分为文本和有意义的 JavaScript。如您所见，我们可以使用简单的连接并生成想要的结果。但是，这种方法并不符合我们的需求。因为我们迟早会传递工作 JavaScript，所以我们想要创建一个循环。例如：

```
var template =
'My skills:' +
'<%for(var index in this.skills) {%>' +
'<a href=""><%this.skills[index]%></a>' +
'<%}%>';
```
如果我们使用连接，结果将是：

```
return
'My skills:' +
for(var index in this.skills) { +
'<a href="">' +
this.skills[index] +
'</a>' +
}
```
当然这会产生错误。这就是为什么我决定遵循 John 的文章中使用的逻辑。即将所有字符串放在数组中并在结尾处连接其元素。

```
var r = [];
r.push('My skills:');
for(var index in this.skills) {
r.push('<a href="">');
r.push(this.skills[index]);
r.push('</a>');
}
return r.join('');
```
下一个逻辑步骤是收集自定义生成函数的不同行。我们已经从模板中提取了一些信息。我们知道占位符的内容及其位置。因此，通过使用辅助变量（光标），我们能够产生所需的结果。

```
var TemplateEngine = function(tpl, data) {
```

```
    var re = /<%([^%>]+)?%>/g,
        code = 'var r=[];\n',
        cursor = 0, match;
    var add = function(line) {
        code += 'r.push("' + line.replace(/"/g, '\\"') + '");\n';
    }
    while(match = re.exec(tpl)) {
        add(tpl.slice(cursor, match.index));
        add(match[1]);
        cursor = match.index + match[0].length;
    }
    add(tpl.substr(cursor, tpl.length - cursor));
    code += 'return r.join("");'; // <-- return the result
    console.log(code);
    return tpl;
}
var template = '<p>Hello, my name is <%this.name%>. I\'m <%this.profile.age%> years old.</p>';
console.log(TemplateEngine(template, {
    name: "Krasimir Tsonev",
    profile: { age: 29 }
}));
```

的代码变量保存函数的主体。它从数组的定义开始。正如我所说，光标向我们展示了模板中的位置。我们需要这样一个变量来遍历整个字符串并跳过数据块。创建了一个额外的添加功能。它的工作是将行附加到代码变量。这里有点棘手。我们需要转义双引号，否则生成的脚本将无效。如果我们运行该示例并检查控制台，我们将看到：

```
var r=[];
r.push("<p>Hello, my name is ");
r.push("this.name");
r.push(". I'm ");
r.push("this.profile.age");
return r.join("");
```

嗯...不是我们想要的。不应引用 this.name 和 this.profile.age。add 方法的一点改进解决了这个问题。

```
var add = function(line, js) {
    js? code += 'r.push(' + line + ');\n' :
        code += 'r.push("' + line.replace(/"/g, '\\"') + '");\n';
}
var match;
while(match = re.exec(tpl)) {
    add(tpl.slice(cursor, match.index));
    add(match[1], true); // <-- say that this is actually valid js
```

```
        cursor = match.index + match[0].length;
}
```
占位符的内容与布尔变量一起传递。现在这会生成正确的身体。

```
var r=[];
r.push("<p>Hello, my name is ");
r.push(this.name);
r.push(". I'm ");
r.push(this.profile.age);
return r.join("");
```
我们需要做的就是创建函数并执行它。在我们的模板引擎的末尾，而不是返回 tpl：

```
return new Function(code.replace(/[\r\t\n]/g, '')).apply(data);
```
我们甚至不需要向函数发送任何参数。我们使用 apply 方法来调用它。它会自动设置范围。这就是让 this.name 工作的原因。在此实际上指向我们的数据。

我们差不多完成了。最后一件事。我们需要支持更复杂的操作，比如 if / else 语句和循环。让我们从上面得到相同的例子，并尝试到目前为止的代码。

```
var template =
'My skills:' +
'<%for(var index in this.skills) {%>' +
'<a href="#"><%this.skills[index]%></a>' +
'<%}%>';
console.log(TemplateEngine(template, {
        skills: ["js", "html", "css"]
}));
```
结果是错误 Uncaught SyntaxError：意外的令牌。如果我们调试一下并打印出代码变量，我们就会看到问题所在。

```
var r=[];
r.push("My skills:");
r.push(for(var index in this.skills) {);
r.push("<a href=\"\">");
r.push(this.skills[index]);
r.push("</a>");
r.push(});
r.push("");
return r.join("");
```
不应将包含 for 循环的行推送到数组。它应该只是放在脚本中。为了达到这个目的，我们必须再做一次检查才能将代码附加到代码中。

```
var re = /<%([^%>]+)?%>/g,
        reExp = /(^( )?(if|for|else|switch|case|break|{|}))(.*)?/g,
```

```
    code = 'var r=[];\n',
    cursor = 0;
var add = function(line, js) {
    js? code += line.match(reExp) ? line + '\n' : 'r.push(' + line + ');\n' :
        code += 'r.push("' + line.replace(/"/g, '\\"') + '");\n';
}
```

添加了一个新的正则表达式。它告诉我们 javascript 代码是否以 if，for，else，switch，case，break，{或}开头。如果是，则只需添加该行。否则它将它包装在 push 语句中。结果是：

```
var r=[];
r.push("My skills:");
for(var index in this.skills) {
r.push("<a href=\"#\">");
r.push(this.skills[index]);
r.push("</a>");
}
r.push("");
return r.join("");
```

当然，一切都正确编译。

```
My skills:<a href="#">js</a><a href="#">html</a><a href="#">css</a>
```