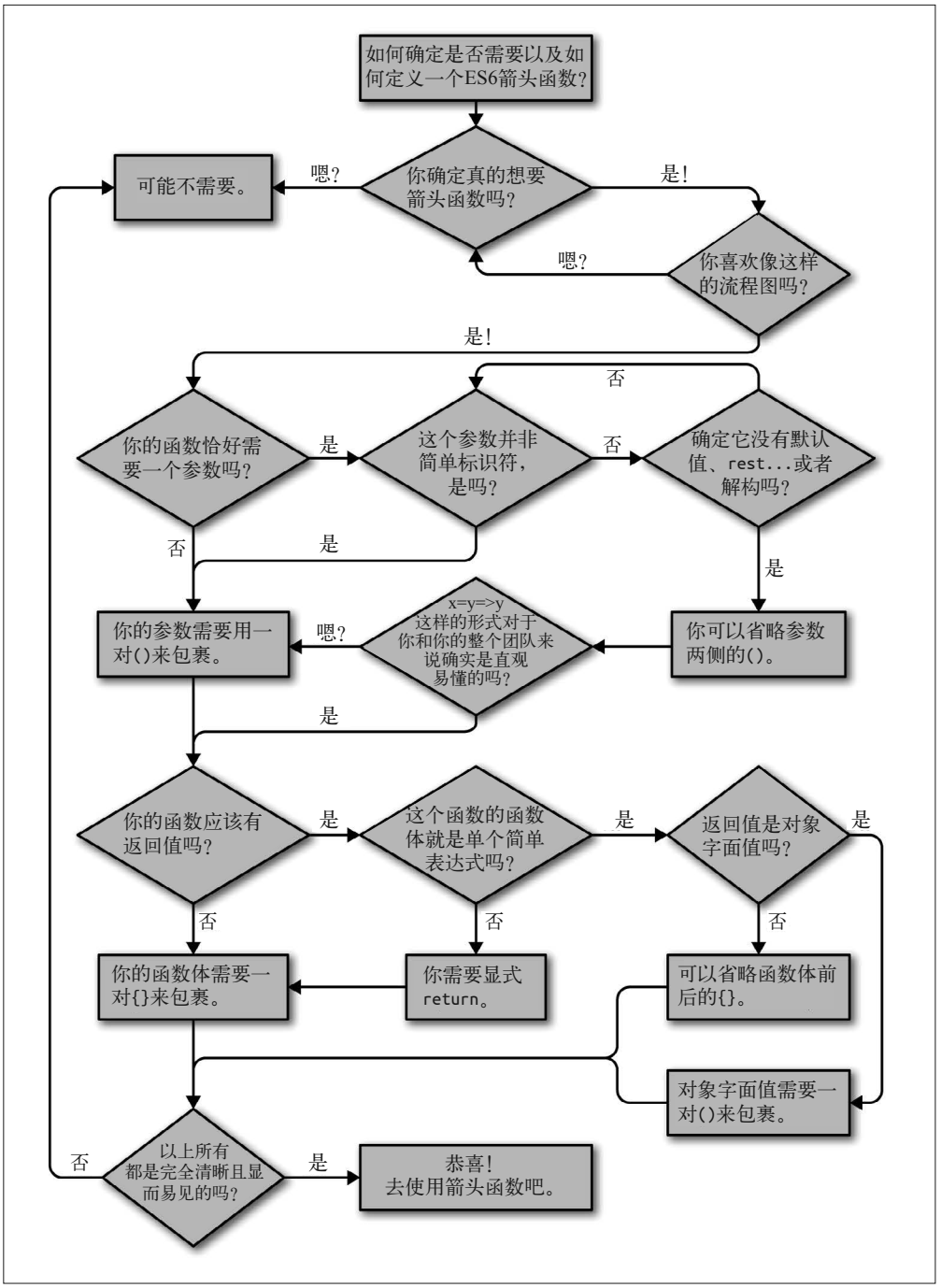
这里用一个可视化的决策图来展示如何 / 为什么采用箭头函数：



2.9 for..of 循环

ES6 在把 JavaScript 中我们熟悉的 for 和 for..in 循环组合起来的基础上，又新增了一个 for..of 循环，在**迭代器**产生的一系列值上循环。

for..of 循环的值必须是一个 iterable，或者说它必须是可以转换 / 封箱到一个 iterable 对象的值（参见本系列《你不知道的 JavaScript（中卷）》第一部分）。iterable 就是一个能够产生迭代器供循环使用的对象。

我们来对比一下 for..of 和 for..in 以展示其中的区别：

```
var a = ["a","b","c","d","e"];

for (var idx in a) {
    console.log( idx );
}
// 0 1 2 3 4

for (var val of a) {
    console.log( val );
}
// "a" "b" "c" "d" "e"
```

可以看到，for..in 在数组 a 的键 / 索引上循环，而 for..of 在 a 的值上循环。

下面是前面代码中的前 ES6 版本的 for..of 形式：

```
var a = ["a","b","c","d","e"],
    k = Object.keys( a );

for (var val, i = 0; i < k.length; i++) {
    val = a[ k[i] ];
    console.log( val );
}
// "a" "b" "c" "d" "e"
```

这里是 ES6 的但是不用 for..of 的等价代码，也可以用来展示如何手动在迭代器上迭代（参见 3.1 节）：

```
var a = ["a","b","c","d","e"];

for (var val, ret, it = a[Symbol.iterator]();
    (ret = it.next()) && !ret.done;
) {
    val = ret.value;
    console.log( val );
}
// "a" "b" "c" "d" "e"
```

在底层，for..of 循环向 iterable 请求一个迭代器（通过内建的 Symbol.iterator，参见 7.3 节），然后反复调用这个迭代器把它产生的值赋给循环迭代变量。

JavaScript 中默认为（或提供）iterable 的标准内建值包括：

- Arrays
- Strings
- Generators（参见第 3 章）
- Collections / TypedArrays (参见第 5 章）

默认情况下平凡对象并不适用 `for..of` 循环。因为它们并没有默认的迭代器，这是有意设计的特性，而不是错误。但这里我们不会深入推导这一结论。在 3.1 节中，我们将会介绍如何为自己的对象定义迭代器，这样就可以使 `for..of` 在任何对象上循环，得到一组我们定义的值。

下面是在原生字符串的字符上迭代的方式：

```
for (var c of "hello") {
    console.log( c );
}
// "h" "e" "l" "l" "o"
```

原生字符串值 `"hello"` 被强制类型转换 / 封箱到等价的 `String` 封装对象中，而这默认是一个 iterable。

在 `for (XYZ of ABC)..` 中，和 `for` 以及 `for..in` 循环中的语句一样，`XYZ` 语句可以是赋值表达式也可以是声明。所以可以这么做：

```
var o = {};

for (o.a of [1,2,3]) {
    console.log( o.a );
}
// 1 2 3

for ({x: o.a} of [ {x: 1}, {x: 2}, {x: 3} ]) {
  console.log( o.a );

  }
// 1 2 3
```

和其他循环一样，`for..of` 循环也可以通过 `break`、`continue`、`return`（如果在函数中的话）提前终止，并抛出异常。在所有这些情况中，如果需要的话，都会自动调用迭代器的 `return(..)` 函数（如果存在的话）让迭代器执行清理工作。

关于 iterable 和迭代器的完成信息参见 3.1 节。

2.10 正则表达式

让我们面对这样一个事实：JavaScript 中的正则表达式很长时间以来基本没有什么变化。所以当 ES6 中终于增加了一些新技巧，真是太好了。这里会概括介绍一下新增的特性，但是全面的正则表达式主题是很复杂的，如果想要学习的话需要阅读专门介绍这个主题的章节 / 书籍（这样的书籍有很多！）。

2.10.1 Unicode 标识

我们会在 2.12 节详细介绍这一主题。这里只是简单介绍 ES6+ 正则表达式新的 `u` 标识，这个标识会为表达式打开 Unicode 匹配。

JavaScript 字符串通常被解释成 16 位字符序列，这些字符对应**基本多语言平面**（Basic Multilingual Plane，BMP）（http://en.wikipedia.org/wiki/Plane_%28Unicode %29）中的字符。但还有很多 UTF-16 字符在这个范围之外，所以字符串中还可能包含这些多字节字符。

在 ES6 之前，正则表达式只能基于 PMB 字符匹配，这意味着那些扩展字符会被当作两个独立的字符来匹配。通常这不是理想的做法。

所以，在 ES6 中，`u` 标识符表示正则表达式用 Unicode（UTF-16）字符来解释处理字符串，把这样的扩展字符当作单个实体来匹配。

并不像其名字所暗示的，“UTF-16”不完全是 16 位的。现代的 Unicode 使用 21 位来表示，像 UTF-8、UTF-16 这样的标准只是大概表明了用多少位来表示一个字符。

举一个例子（直接来自于 ES6 规范）：音乐符号 𝄞（高音符号）是 Unicode 码点为 U+1D11E（0x1D11E）。

如果这个符号出现在正则表达式中（比如 `/𝄞/`），标准 PMB 解释在匹配的时候会将其解释为两个独立的字符（0xD834 和 0xDD1E）。而新的 ES6 支持 Unicode 模式，意味着 `/𝄞/u`（或者转义符 Unicode 形式 `/\u{1D11E}/u`）会把字符串中的 "𝄞" 作为一个单独的匹配字符。

你可能会问这又有什么关系呢？在非 Unicode 的 BMP 模式，这个模式会被当作两个独立的字符，但是仍然会匹配到包含有 "𝄞" 的字符串，像下面这样试一下就会了解：

```
/𝄞/.test( "𝄞-clef" );      // true
```

影响的是匹配部分的长度。举例来说：

```
/^.-clef/ .test( "𝄞-clef" );    // false
/^.-clef/u.test( "𝄞-clef" );    // true
```

模式中的 `^.-clef` 表明只匹配起始处并在普通文本 `"-clef"` 之前有一个字符的情形。在标准的 BMP 模式中，匹配会失败（两个字符），如果用 `u` 标识打开 Unicode 模式，匹配则会成功（单个字符）。

还有一点需要注意，`u` 标识使得 `+` 和 `*` 这样的量词把整个 Unicode 码点作为单个字符而应用，而不仅仅是应用于字符的**低位**（也就是符号的最右部分）。在字符类内部出现的 Unicode 字符也是一样，比如 `/[ 💩-🔊 ]/u`。

正则表达式中的 `u` 标识符行为特性还有很多重要的技术细节，关于这些 Mathias Bynens（https://twitter.com/mathias） 在这篇文章（https://mathiasbynens.be/notes/es6-unicode-regex）中有详细的介绍。

2.10.2 定点标识

ES6 正则表达式中另外一个新增的标签模式是 `y`，通常称为“定点（sticky）模式”。**定点**主要是指在正则表达式的起点有一个虚拟的锚点，只从正则表达式的 `lastIndex` 属性指定的位置开始匹配。

为了说明这一点，我们来考虑两个正则表达式——第一个没有定点模式，而第二个有：

```
var re1 = /foo/,
    str = "++foo++";

re1.lastIndex;          // 0
re1.test( str );        // true
re1.lastIndex;          // 0--没有更新

re1.lastIndex = 4;
re1.test( str );        // true--被忽略的lastIndex
re1.lastIndex;          // 4--没有更新
```

从上面的代码中可以观察到 3 点。

- `test(..)` 并不关心 `lastIndex` 的值，总是从输入字符串的起始处开始执行匹配。
- 因为我们的模式并没有起始锚点 `^`，对 `"foo"` 的搜索从整个字符串向前寻找匹配。
- `test(..)` 不更新 `lastIndex`。

现在，我们试验一下定点模式正则表达式：

```
var re2 = /foo/y,    // <-- 注意定点标识y
    str = "++foo++";

re2.lastIndex;       // 0
re2.test( str );     // false--0处没有找到"foo"
re2.lastIndex;       // 0
```

```
re2.lastIndex = 2;
re2.test( str );      // true
re2.lastIndex;        // 5--更新到前次匹配之后位置

re2.test( str );      // false
re2.lastIndex;        // 0--前次匹配失败后重置
```

下面是关于定点模式的新的观察结果。

- `test(..)` 使用 `lastIndex` 作为 `str` 中精确而且唯一的位置寻找匹配。不会向前移动去寻找匹配——要么匹配位于 `lastIndex` 位置上，要么就没有匹配。
- 如果匹配成功，`test(..)` 会更新 `lastIndex` 指向紧跟匹配内容之后的那个字符。如果匹配失败，`test(..)` 会把 `lastIndex` 重置回 `0`。

一般的没有用 `^` 限制输入起始点匹配的非定点模式可以自由地在输入字符串中向前移动寻找匹配内容。而定点模式则限制了模式只能从 `lastIndex` 开始匹配。

正如这一小节开始所提到的，另一种理解思路是把 `y` 看作一个在模式开始处的虚拟锚点，限制模式（也就是限制匹配的起点）相对于 `lastIndex` 的位置。

关于这个主题的已有文献中，还有一种思路是 `y` 意味着一个模式中的 `^`（起点）锚点。这并不精确，后面我们在“定点锚点”一节中会详细解释。

1. 定点定位

因为 `y` 模式不能向前移动搜索匹配，所以要把 `y` 用于重复匹配，就不得不手动保证 `lastIndex` 指向正确的位置，这种局限性有点奇怪。

这里有一个可能的应用场景：如果确定你关心的匹配总是在某个数字的整数倍位置上（例如：`0`、`10`、`20` 等），就可以构造一个受限的模式来匹配所关心的内容，然后每次在匹配之前手动把 `lastIndex` 设定到这些固定的位置。

考虑：

```
var re = /f../y,
    str = "foo       far       fad";

str.match( re );        // ["foo"]

re.lastIndex = 10;
str.match( re );        // ["far"]

re.lastIndex = 20;
str.match( re );        // ["fad"]
```

但是，如果要处理的字符串并没有这样格式化在固定的位置上，每次匹配之前都要计算出

需要把 lastIndex 设置成什么值可就不那么合理了。

这里有一个技术细节可以考虑用于解决这个问题。y 要求 lastIndex 精确位于每个匹配发生的位置。但是并没有严格要求手动设置这个 lastIndex。

相反，可以以自己的方式构造一个表达式，让它在主匹配之前能够捕获从你关注的内容之后，恰好直到下一次关注内容之前。

因为 lastIndex 会设定为匹配结尾处的下一个字符，所以如果匹配了直到这个点的所有东西，lastIndex 就总会位于 y 模式下次要开始的正确位置。

如果无法预测输入字符串的结构的模式，那么这个技术就不合适，可能无法应用 y。

可以应用 y 模式在字符串中执行重复匹配最适合的场景可能就是结构化的输入字符串。考虑：

```
var re = /\d+\.\s(.*?)(?:\s|$)/y
    str = "1. foo 2. bar 3. baz";

str.match( re );        // [ "1. foo ", "foo" ]

re.lastIndex;           // 7--正确位置!
str.match( re );        // [ "2. bar ", "bar" ]

re.lastIndex;           // 14--正确位置!
str.match( re );        // ["3. baz", "baz"]
```

这种方式可以工作是因为我提前了解了输入字符串的结构：在想要匹配的内容 ("foo" 等) 之前总有一个像 "1. " 这样的数字前缀，然后其后或者是一个空格，或者是字符串结尾 (也就是锚点 $)。所以我构造的正则表达式在主匹配中捕获所有这样的结构，然后使用匹配组 ( )，这样我真正关心的内容就很方便地被提取出来了。

第一次匹配 ("1. foo") 之后，lastIndex 值为 7，这已经是下一个匹配 "2. bar" 开始所需的位置了，依次继续。

如果要使用 y 定点模式来重复匹配，你将很可能想要寻找像我们前面展示的自动更新 lastIndex 位置的机会。

2. 定点还是全局

有些读者可能意识到，也可以用 g 全局匹配标识和 exec(..) 方法来模拟这种相对于 lastIndex 的匹配，就像这样：

```
var re = /o+./g,        // <-- 注意g!
    str = "foot book more";

re.exec( str );         // ["oot"]
re.lastIndex;           // 4

re.exec( str );         // ["ook"]
re.lastIndex;           // 9

re.exec( str );         // ["or"]
re.lastIndex;           // 13

re.exec( str );         // null--没有更多匹配!
re.lastIndex;           // 0--现在从头开始!
```

确实，g 模式匹配加上 exec(..) 从 lastiIndex 的当前值开始匹配，同时会在每次匹配（或匹配失败后）更新 lastIndex，但是这和 y 的行为特性并不相同。

注意前面的代码中位于位置 6 处的 "ook"，会被第二个 exec(..) 调用匹配找到，虽然在那个时候，lastIndex 值是 4（来自于上一次匹配的结尾）。这是为什么？因为就像我前面所说，非定点匹配可以在匹配过程中自由向前移动。因为不允许向前移动，所以定点模式表达式在这里会匹配失败。

除了可能并不需要的向前移动匹配，只用 g 替代 y 的另一个缺点是 g 改变了某些匹配方法的行为特性，比如 str.match(re)。

考虑：

```
var re = /o+./g,        // <-- 注意g!
    str = "foot book more";

str.match( re );        // ["oot","ook","or"]
```

看到所有的匹配是如何立即返回了吗？有时候这不是问题，但有时候它又并非是你想要的。

通过使用工具 test(..) 和 match(..)，y 定点标识带来的是一次一个的向前匹配。只是要确保每次匹配的时候 lastIndex 总是处于正确的位置上！

3. 锚定

前面已经提醒过，把定点模式想象成就是从 ^ 开始的模式是不精确的。^ 锚点在正则表达式里有独特的含义，不会被定点模式所改变。^ 是一个总是指向输入起始处的锚点，和 lastIndex 完全没有任何关系。

除了一些贫乏 / 不精确文档的原因，实际上在 Firefox 上的一个老旧的前 ES6 定点模式试验确实让 ^ 与 lastIndex 相关，这也不幸地进一步加深了误解，所以这样的行为特性已经存在几年了。

ES6 选择不采取这种实现方式。模式中的 ^ 就是表示也仅表示输入的起始点。

因此，像 /^foo/y 这样的模式总是也只是寻找字符串起始处的 "foo" 匹配，但前提是允许在此处匹配。如果 lastIndex 不是 0，那么匹配就会失败。考虑：

```
var re = /^foo/y,
    str = "foo";

re.test( str );         // true
re.test( str );         // false

re.lastIndex;           // 0--失败后重置

re.lastIndex = 1;
re.test( str );         // false--由于定位而失败
re.lastIndex;           // 0--失败后重置
```

底线：y 加上 ^ 再加上 lastIndex > 0 是一个不兼容的组合，总是会导致匹配失败。

y 不会以任何形式改变 ^ 的含义，而 m 多行模式则会，也就是说，^ 意味着输入起始处或者换行之后的文字起始位置。所以，如果组合使用 y 和 m 模式，就会发现字符串中有多个 ^ 匹配基准。但是记住：因为这是 y 定点的，需要确保每次匹配的时候 lastIndex 指向正确的换行位置（可能通过匹配行尾来实现），否则下次匹配不会成功。

2.10.3 正则表达式 flags

在 ES6 之前，如果想要通过检查一个正则表达式对象来判断它应用了那些标识，需要把它从 source 属性的内容中解析出来——讽刺的是，这可能需要用另一个正则表达式，就像下面这样：

```
var re = /foo/ig;

re.toString();          // "/foo/ig"

var flags = re.toString().match( /\/([gim]*)$/ )[1];

flags;                 // "ig"
```

而在 ES6 中，现在可以用新的 flags 属性直接得到这些值。

```
var re = /foo/ig;

re.flags;               // "gi"
```

这里有一点细节，ES6 规范中规定了表达式的标识按照这个顺序列出："gimuy"，无论原始指定的模式是什么。这就是 /ig 和 "gi" 之间区别的原因。

指定或列出的标识顺序是无关紧要的。

ES6 的另一个调整是如果把标识传给已有的正则表达式，`RegExp(..)` 构造器现在支持 `flags`：

```
var re1 = /foo*/y;
re1.source;                         // "foo*"
re1.flags;                          // "y"

var re2 = new RegExp( re1 );
re2.source;                         // "foo*"
re2.flags;                          // "y"

var re3 = new RegExp( re1, "ig" );
re3.source;                         // "foo*"
re3.flags;                         // "gi"
```

在 ES6 之前，`re3` 构造会抛出一个错误，但在 ES6 中，可以在复制的时候覆盖标识。

2.11 数字字面量扩展

在 ES5 之前，数字字面量看起来是这样的——八进制形式并没有被正式支持，只作为扩展支持，而浏览器已经达成实质上的一致：

```
var dec = 42,
    oct = 052,
    hex = 0x2a;
```

虽然可以用不同进制形式指定数字，但是数字的数字值还是保存的值，并且默认的输出解释形式总是十进制。前面代码中的 3 种形式在其中保存的都是值 `42`。

为了进一步展示 `052` 是一个非标准形式的扩展，考虑：

```
Number( "42" );             // 42
Number( "052" );            // 52
Number( "0x2a" );           // 42
```

ES5 仍然支持作为浏览器扩展的八进制形式（包括这样的不一致），除了在严格模式下，是不支持八进制字面量（`052`）形式的。采用这个限制主要是因为很多开发者有这样一种（来自于其他语言经验的）习惯，就是看起来无意地在十进制值前加 `` `0` `` 用于代码对齐，然后就会发现已经不小心完全改变了这个数值！

在非十进制数字字面量表示方面，ES6 继续支持旧有的修改 / 变体。同时现在有了一个正式八进制形式、一个补充的十六进制形式，以及一个全新的二进制形式。由于 Web 兼容性