你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析**export**

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };
```

```
// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这

个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组

件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是

用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个

名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的x；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）

绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；

- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
```

```
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的x；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default`

`...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

aName = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的

概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
```

```
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as ...”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
```

```
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import`

`{x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的x；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an

Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
```

...

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作

用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

aName = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as ...”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
```

```
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as ...”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回

给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析**export**

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类

似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的x；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的

是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着

它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
```

```
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当

该函数关联给名字（aName）时，JavaScript又会反向地处理该函数（作为对象f）的属性f.name，使该名字指向aName。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致import f from ...之后访问f.name值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- export ...语句通常是按它的词法声明来创建的标识符的，例如export var x = ...就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在import语句所在的模块中却是一个常量，因此总是不可写的。
- 由于export default ...没有显式地约定名字“default（或*default*）”应该按let/const/var的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用var x...来声明，这个x也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as ...”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

ExportDeclaration: **export default** *AnonymousFunctionDefinition*;

NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

aName = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as ...”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析**export**

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };
```

```
// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这

个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组

件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是

用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个

名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的x；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）

绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；

- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
```

```
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的x；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default`

...;语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

aName = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的

概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import {x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
```

```
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as ...”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。

你好，我是周爱民，欢迎回到我的专栏。

今天我要讲述的内容是从ECMAScript 6开始在JavaScript中出现的**模块技术**，这对许多JavaScript开发者来说都是比较陌生的。

一方面在于它出现得较晚，另一方面，则是因为在普遍使用的Node.js环境带有自己内置的模块加载技术。因此，ECMAScript 6模块需要通过特定的命令行参数才能开启，它的应用一直以来也就不够广泛。

导致这种现象的根本原因在于**ECMAScript 6模块是静态装配的**，而传统的Node.js模块却是动态加载的。因而两种模块的实现效果与处理逻辑都大相径庭，Node.js无法在短期内提供有效的手段帮助开发者将既有代码迁移到新的模块规范下。

总结起来，确实是这些更为现实的原因阻碍了ECMAScript 6模块技术的推广，而非是ECMAScript 6模块是否成熟，或者设计得好与不好。

不过即使如此，ECMAScript 6模块仍然在JavaScript的一些大型应用库、包，或者对新规范更友好的项目中得到了不错的运用和不俗的反响，尤其是在使用转译器（例如Babel）的项目中，开发者通常是首选ECMAScript 6模块语法的。

因此ECMAScript 6模块也有着非常好的应用环境与前景。

导出的内容

上一讲我提到过有且仅有六种声明语法，而本质上export也就只能导出这六种声明语法所声明的标识符，并且在导出时将它们统一称为“名字”。

在语言设计中，所谓“标识符”与“名字”是有语义差别的，export将之称为名字，就意味着这是一个标识符的子集。类似的其它子集也是存在的，例如“保留字是标识符名，但不能用作标识符（A reserved word is an IdentifierName that cannot be used as an Identifier）”。

在JavaScript语言的设计上，除了那些预设的标点符号（例如大括号、运算符之类），以及部分的保留字和关键字之外，事实上用户代码可以书写的只有三种东西。这包括：

- 标识符：（通常是）一个**名字**；
- 字面量：表明由它的字面含义所决定的一个**值**；
- 模板：一个可计算结果的字符串**值**。

所以，如果在这个层面上解构一份你所书写的JavaScript代码，那么你所能书写/声明的，就一定只有“名字和值”。

这个结论是非常非常关键的。为什么呢？因为export事实上就只能导出“名字和值”。然而一旦它能导出“名字和值”，也就意味着它能导出一个模块中的“全部内容”，因为如上所面所讲的：

“名字和值”正是你所书写的代码的全部。

我的代码去哪儿了呢？

你是不是一刹那之间觉得自己的代码都白写了。:)

确实是的，真的是白写了。不过，我在前面讲的都是纯粹的“语言设计”，在语言设计层面上来讲，代码就是文本，是没有应用逻辑的。而你所写的代码绝大多数都是应用逻辑，当去除掉这些应用逻辑之后，那些剩下的死气沉沉的、纯粹的符号，才是语言层面的所谓“代码文本”。

去掉了执行逻辑所表达的那些行为、动作、结果和用户操作的代码，就是静态代码了。而事实上，ECMAScript 6中的模块就是用来理解你的程序中的那些静态代码的，也就是那些没有任何生气的字符和符号。因此它也就只能理解上面所谓的6种声明，以及它们声明出来的那些“名字和值”。

再无其它。

解析export

所以，将所有export语法分类，其实也就只有两个大类。如下：

```
// 导出“（声明的）名字”
export <let/const/var> x ...;
export function x() ...
export class x ...
```

```
export {x, y, z, ...};


// 导出“（重命名的）名字”
export { x as y, ...};
export { x as default, ... };


// 导出“（其它模块的）名字”
export ... from ...;


// 导出“值”
export default <expression
```

关于导出声明的、重命名的和其它模块的名字这三种情况，其实都比较容易理解，就是形成一个名字表，让外部模块能够查看就可以了。

但是对于最后这种形式，也就是“（导出）值”的形式，事实上是非常特殊的。因为如同我在上面所讲过的，要导出一个模块的全部内容就必须导出“（全部的）名字和值”，然而纯粹的值没有名字，于是也就没法访问了，所以这就与“导出点什么东西”的概念矛盾了。

因为这个东西要是没名字，也就连“自己是什么”都说不清楚，也就什么也不是了。

所以ECMAScript 6模块约定了一个称为"default"的名字，用于来导出当前模块中的一个“值”。显然的，由于所谓“值”是表达式的运算结果，所以这里的语法形式就是：

```
export default <expression>;
```

其中的“_expression”_就是用于求值的，以便得到一个结果（Result）并导出成为缺省的名字“default”。这里有两个便利的情况，一个是在JavaScript中，一般的字面量也是值、也是单值表达式，因此导出这样一个字面量也是合法的：

```
export default 2;  // as state of the module, etc.
export default "some messages"; // data or information
...
```

第二个便利的情况，是因为JavaScript中对象也是字面量、也是值、也是单值表达式。而对象成员可以组合其它任何数据，所以通过上述的语法几乎可以导出当前模块中全部的“值”（亦即是任何可以导出的数据）。例如：

```
var varName = 100;
export default {
  varName,  // 直接导出名字
  propName: 123,  // 导出值
  funcName: function() { }, // 导出函数
  foo() { // 或导出与主对象相关联的方法
     // method
  }
}
```

所以，事实上`export default ...`虽然简单，却是对“导出名字”的非常必要的补充。这样一来，用户既可以导出那些有名字的数据，也可以导出那些没有名字的数据，即一个模块中所有的数据都可以被导出了。

那么接下来，就要讲到标题中的这个语法了：

```
export default function() {}
```

你知道在这个语法中export到底导出了什么吗？是名字？还是值？

导出语句的处理逻辑

在讨论这个问题之前，你得先思考一个更关键的问题：“export如何导出名字”。这个问题的关键之处在于，如果只是导出一个名字，那么它其实在“某个名字表”中做一个登记项就可以了。并且JavaScript中也的确是这样处理的。但是实际使用的时候，这个名字还是要绑定一个具体的值才是可以使用的。因此，一个export也必须理解为这样两个步骤：

1. 导出一个名字
2. 为上述名字绑定一个值

这两个步骤其实与使用“var x = 100”来声明一个变量的过程是一致的。因此以如下代码为例（注意六种声明在名字处理上是类似的），

```
export var x = 100;
```

在导出的时候，其实是先在“某个名字表”中登记一个“名字x”就可以了。这个过程也就是JavaScript在模块装载之前对export所做的全部工作。不过如果是从另一端（亦即是import语句）的角度看过来，那么就会多出来一个步骤。import语句会（例如`import`

`{x} from ...`）：

1. （与export类似）按照语法在当前模块中声明名字，例如上面的`x`；
2. 添加一个当前模块对目标模块的依赖项。

有了上述的第二步操作，JavaScript就可以依据所有它能在静态文本中发现的`import`语句来形成模块依赖树，最后就可以找到这个模块依赖树最顶端的根模块，并尝试加载之。

所以关键的是，在“模块export/import”语法中，JavaScript是依赖import来形成依赖树的，与export无关。但是直到目前为止（我的意思是直到找到所有导入和导出的名字，并完成所有模块的装配的现在为止），没有任何一行用户的JavaScript代码是被执行过的。至于原因，从本讲的最开始我就讲过了：这个export/import过程中，源代码只被理解为静态的、没有逻辑的“代码文本”。那么既然“没有逻辑”，又怎么可能执行类似于：

```
export default <expression>;
```

中的“*expression*”呢？要知道所谓表达式，就是程序的计算逻辑啊。

所以，这里先得出了第一个关键结论：

在处理export/import语句的全程，没有表达式被执行！

导出名字与导出值的差异

现在，假如：

```
export default <expression>;
```

中的“expression”在导入导出中完全不起作用（不执行），那么这行语句又能做什么呢？事实上，这行语句与直接“导出一个名字”并没有任何区别。它与这样的语法相同：

```
export var x = 100;
```

它们都只是导出一个名字，只是前者导出的是“default”这个特殊名字，而后者导出的是一个变量名“x”。它们都是确定的、符合语法规则的标识符，也可以表示为一个字符串的字面文本。它们的作用也完全一致：就是在前面所说的“某个名字表”中添加“一个登记项”而已。

所以，导出名字与导出值本质上并没有差异，在静态装配的阶段，它们都只是表达为一个名字而已。

然后，也正是如同`var x = 100;`在执行阶段需要有一个将“值100”绑定给“变量x（的引用）”的过程一样，这个`export default ...;`语句也需要有完全相同的一个过程来将它后面的表达式（*expression*）的结果绑定给“default”这个名字。如果不这么做，那么“*export default*”在语义上的就无法实现导出名字“*default*”了——在静态装配阶段，名字“default”只是被初始化为一个“单次绑定的、未初始化的标识符”。

所以现在你就可以在语义上模拟这样一个过程，即：

```
export default function() {}

// 类似于如下代码
//（但并不在当前模块中声明名字"default"）
export var default = function() {}
```

你可以进一步地模拟JavaScript后续的装配过程。这个过程其实非常简单：

- 找到并遍历模块依赖树的所有模块（这个树是排序的），然后
- 执行这些模块最顶层的代码（*Top Level Module Evaluation*）。

在执行到上述`var default ....`（或类似对应的`export default ...`）语句时，执行后面的表达式，并将执行结果（Result）绑定给左侧的那个变量就可以了。如此，直到所有模块的顶层代码都执行完毕，那么所有的导出名字和它们的值也都必然是绑定完成了的。

同样，由于import的名字与export的名字只是一个映射关系，所以import的名字（所对应的值）也就初始化完成了。

再确切地说（这是第二个关键结论）：

所谓模块的装配过程，就是执行一次顶层代码而已。

匿名函数表达式的执行结果

接下来讨论语句中的`... function() {}`这个匿名函数表达式。

按照JavaScript的约定，匿名函数表达式可以理解为一个函数的“字面量（值）”。理解“字面量值”这个说法是很有意义的，因为它意味着它没有名字。你可不要在心中暗骂哦，这绝不是废话。

“字面量（值）没有名字”就意味着执行这个“单值表达式”不会在当前作用域中产生一个名字，即使这个函数是具名的，也必然是如此。所以，这才带来了JavaScript中的经典示例，即：具名函数作为表达式时，名字在块级作用域中无意义。例如：

```
// 具名函数作为表达式
var x1 = function x2() {
  ...
}

// 具名函数（声明）
function x3() {
  ...
}
```

上面的例子中，x1~3都是具有不同的语义的。其中，x2是不会在当前作用域（示例中是全局）中登记为名字的。而现在，就这一讲的主题来说，在使用下面的语法：

```
export default function() { }
export default function x() { }
```

导出一个匿名函数，或者一个具名的函数的时候，这两种情况下是不同的。但无论它是否具名，它们都是不可能在当前作用域中绑定给`default`这个名字，作为这个名字对应的值的。

这段处理逻辑被添加在语法：

> *ExportDeclaration*: **export default** *AnonymousFunctionDefinition*;
>
> NOTE: ECMAScript是将这里导出的对象称为_Expression_/*AssignmentExpression*，这里所谓_AnonymousFunctionDefinition_则是其中_AssignmentExpression_的一个具体实例。

的执行（*Evaluation*）处理过程中。也就是说当执行这行声明时，如果后面的表达式是匿名函数声明，那么它将强制在当前作用域中登记为“***default***”这样一个特殊的名字，并且在执行时绑定该匿名函数。所以，尽管语义上我们需要将它登记为类似`var default ...`所声明的名字“***default***”，但事实上它被处理成了一个不可访问的中间名字，然后影射给该模块的“某个名字表”。

不过需要注意的是，这是一个**匿名函数定义**（*AnonymousFunctionDefinition*），而不是一个匿名函数表达式（*Anonymous FunctionExpression*）。一般函数的语句则被称为声明（或更严谨地称为宣告，*Function Declarations*）。而所谓**匿名函数定义**，其本身是表述为：

> *aName* = ***FunctionExpression***

或类似于此的语法风格的。它可以用在一般的赋值表达式、变量声明的右操作数，以及对象声明的成员初始值等等位置。在这些位置上，该函数表达式总是被关联给一个名字。一方面，这种关联不是严格意义上的“名字->值”的绑定语义；另一方面，当该函数关联给名字（`aName`）时，JavaScript又会反向地处理该函数（作为对象`f`）的属性`f.name`，使该名字指向`aName`。

所以，在本讲中的“export default function() {}”，在严格意义上来说（这是第三个关键结论）：

它并不是导出了一个匿名函数表达式，而是导出了一个匿名函数定义（Anonymous Function Definition）。

因此，该匿名函数初始化时才会绑定给它左侧的名字“***default***”，这会导致`import f from ...`之后访问`f.name`值会得到“***default***”这个名字。

类似的，你使用下面的代码也会得到这个“***default***”：

```
var obj = {
  "default": function() {}
};
console.log(obj.default.name); // "default"
```

知识补充

关于export，还可以有一些补充的知识点。

- `export ...`语句通常是按它的词法声明来创建的标识符的，例如`export var x = ...`就意味着在当前模块环境中创建的是一个变量，并可以修改等等。但是当它被导入时，在`import`语句所在的模块中却是一个常量，因此总是不可写的。
- 由于`export default ...`没有显式地约定名字“default（或*default*）”应该按`let/const/var`的哪一种来创建，因此JavaScript缺省将它创建成一个普通的变量（var），但即使是在当前模块环境中，它事实上也是不可写的，因为你无法访问一个命名为“*default*”的变量——它不是一个合法的标识符。
- 所谓匿名函数，仅仅是当它直接作为操作数（而不是具有上述“匿名函数定义”的语法结构）时，才是真正匿名的，例如：

```
console.log((function(){}).name);  // ""
```

- 由于类表达式（包括匿名类表达式）在本质上就是函数，因此它作为default导出时的性质与上面所讨论的是一致的。
- 导出项（的名字）总是作为词法声明被声明在当前模块作用域中的，这意味着它不可删除，且不可重复导出。亦即是说即使是用`var x...`来声明，这个`x`也是在_lexicalNames_中，而不是在_varNames_中。
- 所谓“某个名字表”，对于export来说是模块的导出表，对于import来说就是名字空间（名字空间是用户代码可以操作的组件，它映射自内部的模块导入名字表）。不过，如果用户代码不使用“import * as …”的语法来创建这个名字空间，那么该名字表就只存在于JavaScript的词法分析过程中，而不会（或并不必要）创建它在运行期的实例。这也是我一直用“某个名字表”来称呼它的原因，它并不总是以实体形式存在的。
- 上述名字表简化了ECMAScript中对导入导出记录（*ImportEntry/ExportEntry Record Fields*）的理解。因此如果你试图了解更多，建议你阅读ECMAScript的具体章节。
- 没有模块会导出（传统意义上的）main()，因为ECMAScript为了维护模块的静态语义，而把执行过程及其入口的定义丢回给了引擎或宿主本身。

思考题

本讲的内容中，你需要重点复习三个关键结论的得出过程。这对于之前几讲中所讨论的内容会是很好的回顾。

除此之外，建议你思考如下问题：

- 为什么在import语句中会出现“变量提升”的效果？

如果你并不了解什么是“变量提升”，不用担心，下一讲中我会再次提到它。