1:ES6 模块不是对象，而是通过`export`命令显式指定输出的代码，再通过`import`命令输入。

2：`export`语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

3：，`export`命令可以出现在模块的任何位置，只要处于模块顶层就可以。如果处于块级作用域内，就会报错，下一节的`import`命令也是如此。这是因为处于条件代码块之中，就没法做静态优化了，违背了 ES6 模块的设计初衷。

4：

4. import 命令

使用`export`命令定义了模块的对外接口以后，其他 JS 文件就可以通过`import`命令加载这个模块。

```
// main.js
import { firstName, lastName, year } from './profile.js';

function setName(element) {
  element.textContent = firstName + ' ' + lastName;
}
```

上面代码的`import`命令，用于加载`profile.js`文件，并从中输入变量。`import`命令接受一对大括号，里面指定要从其他模块导入的变量名。大括号里面的变量名，必须与被导入模块（`profile.js`）对外接口的名称相同。

如果想为输入的变量重新取一个名字，`import`命令要使用`as`关键字，将输入的变量重命名。

```
import { lastName as surname } from './profile.js';
```

`import`命令输入的变量都是只读的，因为它的本质是输入接口。也就是说，不允许在加载模块的脚本里面，改写接口。

`import`后面的`from`指定模块文件的位置，可以是相对路径，也可以是绝对路径，`.js`后缀可以省略。如果只是模块名，不带有路径，那么必须有配置文件，告诉JavaScript 引擎该模块的位置。

```
import {myMethod} from 'util';
```

上面代码中，`util`是模块文件名，由于不带有路径，必须通过配置，告诉引擎怎么取到这个模块。

注意，`import`命令具有提升效果，会提升到整个模块的头部，首先执行。

```
foo();

import { foo } from 'my_module';
```

上面的代码不会报错，因为`import`的执行早于`foo`的调用。这种行为的本质是，`import`命令是编译阶段执行的，在代码运行之前。

由于`import`是静态执行，所以不能使用表达式和变量，这些只有在运行时才能得到结果的语法结构。

最后，`import`语句会执行所加载的模块，因此可以有下面的写法。

```
import 'lodash';
```

上面代码仅仅执行`lodash`模块，但是不输入任何值。

模块的整体加载

除了指定加载某个输出值，还可以使用整体加载，即用星号（`*`）指定一个对象，所有输出值都加载在这个对象上面。

下面是一个`circle.js`文件，它输出两个方法`area`和`circumference`。

```
// circle.js

export function area(radius) {
  return Math.PI * radius * radius;
}

export function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

现在，加载这个模块。

```
// main.js

import { area, circumference } from './circle';
```

```
console.log('圆面积: ' + area(4));
console.log('圆周长: ' + circumference(14));
```

上面写法是逐一指定要加载的方法，整体加载的写法如下。

```
import * as circle from './circle';

console.log('圆面积: ' + circle.area(4));
console.log('圆周长: ' + circle.circumference(14));
```

6. export default 命令

从前面的例子可以看出，使用`import`命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。但是，用户肯定希望快速上手，未必愿意阅读文档，去了解模块有哪些属性和方法。

为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到`export default`命令，为模块指定默认输出。

```
// export-default.js
export default function () {
  console.log('foo');
}
```

上面代码是一个模块文件`export-default.js`，它的默认输出是一个函数。

其他模块加载该模块时，`import`命令可以为该匿名函数指定任意名字。

```
// import-default.js
import customName from './export-default';
customName(); // 'foo'
```

上面代码的`import`命令，可以用任意名称指向`export-default.js`输出的方法，这时就不需要知道原模块输出的函数名。需要注意的是，这时`import`命令后面，不使用大括号。

`export default`命令用在非匿名函数前，也是可以的。

```
// export-default.js
export default function foo() {
  console.log('foo');
}

// 或者写成

function foo() {
  console.log('foo');
}
```

上面代码中，`foo`函数的函数名`foo`，在模块外部是无效的。加载的时候，视同匿名函数加载。

edit

下面比较一下默认输出和正常输出。

```
// 第一组
export default function crc32() { // 输出
  // ...
}

import crc32 from 'crc32'; // 输入

// 第二组
export function crc32() { // 输出
  // ...
};

import {crc32} from 'crc32'; // 输入
```

上面代码的两组写法，第一组是使用`export default`时，对应的`import`语句不需要使用大括号；第二组是不使用`export default`时，对应的`import`语句需要使用大括号。

`export default`命令用于指定模块的默认输出。显然，一个模块只能有一个默认输出，因此`export default`命令只能使用一次。所以，import命令后面才不用加大括号，因为只可能唯一对应`export default`命令。

本质上，`export default`就是输出一个叫做`default`的变量或方法，然后系统允许你为它取任意名字。所以，下面的写法是有效的。

export default 只能写一次

本质上，`export default`就是输出一个叫做`default`的变量或方法，然后系统允许你为它取任意名字。所以，下面的写法是有效的。

```
// modules.js
function add(x, y) {
  return x * y;
}
export {add as default};
// 等同于
// export default add;

// app.js
import { default as foo } from 'modules';
// 等同于
// import foo from 'modules';
```

7. export 与 import 的复合写法

如果在一个模块之中，先输入后输出同一个模块，`import`语句可以与`export`语句写在一起。

```
export { foo, bar } from 'my_module';

// 可以简单理解为
import { foo, bar } from 'my_module';
export { foo, bar };
```

上面代码中，`export`和`import`语句可以结合在一起，写成一行。但需要注意的是，写成一行以后，`foo`和`bar`实际上并没有被导入当前模块，只是相当于对外转发了这两个接口，导致当前模块不能直接使用`foo`和`bar`。