运行时两种内部类型，js 运行过程中产生
- Completion Record，语句完成结果
  - [[type]]: normal/break/continue/return/throw
  - [[value]]: Types，7 种类型或 empty，return 和 throw 才会用到 value
  - [[target]]: label，涉及到 label 语句会产生，循环中 continue 和 break
- Lexical Environment

简单语句
- ExpressionStatement -} normal
- EmptyStatement，一个 ; 号 -} **normal**
- DebuggerStatement，debugger -} **normal**
- **ThrowStatement**
  - 所有产生运行时错误都会产生 throw 的结果
  - 唯一一个可以从函数里面甚至到函数外面
- **ContinueStatement**，continue label1;
- **BreakStatement**，break label2;
- **ReturnStatement**

复合语句
- BlockStatement
  - const/let
  - 执行到返回 非 normal 的语句会中断
- IfStatement
- SwitchStatement
- IterationStatement
  - while(<表达式>) {<语句>}
    - 会消费 continue 和 break
  - do {<语句>} while(<表达式>)
  - for(<此处可放声明>;<表达式>;<表达式>) {<语句>}，var 声明作用在函数体范围，const/let 会在产生的独立的**作用域（文本的范围）**，在 block 之外，可以理解为父作用域
  - for(<此处可放声明>in<表达式>) {<语句>}
    - {}中不可以用 in 运算符 => 标准得出双份
  - for(<此处可放声明>of<表达式>) {<语句>}，对应 iterator 机制，对象/数组都有
    - for…of => Iterator => Generator/Array
    - 可实现无穷迭代
  - for await (of)
- WithStatement
- LabelledStatement
- TryStatement

```
try {
    // … 这里不是 block，不可以省略{}
} catch (x/产生独立的作用域，就是下面的大括号一样>) {
    // … 这里不是 block，不可以省略{}
} finally {
    // ...
}
```

```
// block statement
{
    // ...
}
```

```
for (let i = 0;;) {
    console.log(i);
}


for (;;) {
    let i = 0;
    console.log(i);
}
```

声明
- FunctionDeclaration
  - vs 函数表达式
- GeneratorDeclaration
  - 特殊 function，里面可用 yield
  - 简单可理解为返回多个数的函数
  - 原理 generator，async
- AsyncFunctionDeclaration
- AsyncGeneratorDeclaration
- VariableStatement
  - 如果有 var，不建议写在语句的子结构中
  - 不要再任何 block 中使用 var
- ClassDeclaration
  - 不允许重复声明
- LexicalDeclaration

BoundName 13.3.1.2

```
function* gen () {
    yield 1;
    yield 2;
    yield 3;
    let i = 0;
    while(true) {
        yield i++;
    }
}
let gen = foo();

gen.next();
gen.next();
gen.next();
gen.next();


function sleep(time) {
    return new Promise(resolve => setTimeout(resolve, time));
}


async function* foo() {
    var i = 0;
    while (true) {
        console.log(i++);
        yield i++;
        await sleep(1000);
    }
};


void async function() {
    var g = foo();

    // console.log(await g.next());
    // console.log(await g.next());
    // console.log(await g.next());
    // console.log(await g.next());
    // console.log(await g.next());
    // console.log(await g.next());

    for await (let e of g) {
        console.log(e);
    }
}();
```

Types - Object
对象机制

任何一个对象都是唯一的，与他本身状态无关。**即使状态完全一致的两个对象，也并不相等。**

用状态来描述**对象**，
状态的改变即是**行为**。

对象的三要素：
- 唯一性 identifer
- 状态 state
- 行为 behavior

✘ 封装、复用、解耦，内聚 // 描述代码架构的合理性
✘ 继承 // 面向对象的一个子系统的概念
✘ 多态 // 描述动态性的程度

无论什么概念，怎么理解，影响自己的思想。怎么影响你在工作中的行为。

面向对象的第一个范式
✔ oriented class-based object
- 类是一种常见的描述对象的方式
- 两个主要流派：归类 和 分类
  - 归类 -> 多继承，C++
  - 分类 -> 单继承结构，并且会有一个基类 Object
    - 把两个分支的类合并到一起去 => interface
    - 复用的问题，无法合并到一个类，mixin

✔ oriented prototype object
- 原型是一种更接近人类原始认知的描述对象的方法
- 不试图做严谨的分类，而是采用"相似"这样的方式去描述对象
- 任何对象仅仅需要描述他自己与原型的区别即可

"狗咬人"中"咬"这个行为该如何使用对象抽象？

✘

```
class Dog {
    bite(human) {
        // ...
    }
}
```

✔

```
class Human {
    hurt(damage) {
        //...
    }
}
```

我们不应该受到语言描述的干扰，在设计对象的状态和行为时，我们总是遵循"行为改变状态"的原则。

"咬" --- 业务！！

js 对象模型：
object in javascript
在 js 运行时，原生对象的描述方式非常简单，我们只需要关心 「原型」和「属性」两个部分。

key-value
Symbol/String - Data/Accessor

js 用属性来统一抽象对象状态和行为。
一般来说：
- 数据属性用于描述状态。如果数据属性存储函数，也可以用于描述行为。
  - [[value]]
  - writeable
  - enumerable
  - configurable
- 访问器属性则用于描述行为
  - get
  - set
  - enumerable
  - configurable

当我们访问属性时，如果当前对象没有，则会沿着原型找原型对象是否有此名称的属性，而原型对象还可能有原型，因此，会有"原型链"这一说法，这一算法保证了，每个对象只需要描述自己和原型的区别即可。

object API：
- 基本对象能力：{} / . / [] / Object.defineProperty
- ES5 原型 API：Object.create / Object.setPrototypeOf / Object.getPrototypeOf（不要 2、3 混用）
- 基于类 OOP 范式：new / class / extends（不要 2、3 混用）
- new / function / prototype（不推荐）

js 中特殊对象：
✔ Function Object
除了一般对象的属性和原型，函数对象还有一个行为[[call]]，用 js 中的 function 关键字、箭头函数或者 Function 构造器创建的对象，会有[[call]]这个行为，用类似 f() 语法把对象当做函数调用时，会访问到 [[call]] 这个行为，如果对应的对象没有[[call]] 行为，则会报错。

Array
[[length]]

Object.prototype
[[SetPrototypeOf]]

9.4 Built-in Exotic Object Internal Methods and Slots