ES6从入门到进阶完全指南


愿景："让编程不在难学，让技术与生活更加有趣" 更多课程请访问xdclass.net

第一章 课程简介

第1集 ES6从入门到进阶完全指南课程简介

简介：ES6从入门到进阶完全指南

- 课程特色
  - babel环境搭建
  - 使用场景解析
  - 异步编程的几种写法
  - ES6详细的知识点
  - 扩展ES7ES8的知识
- 售后
  - 图文笔记
  - 交流群
  - 答疑

愿景："让编程不在难学，让技术与生活更加有趣" 更多课程请访问xdclass.net

第二章 ECMAScript6简介及项目环境搭建

第1集 ECMAScript 6简介

简介： 介绍ECMAScript 和 JavaScript以及ECMAScript 2015 的关系

- ECMAScript 6是什么

  ECMAScript 6.0（以下简称 ES6）是 JavaScript 语言的下一代标准，已经在2015年6月正式发布了。它的目标，是使得 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

- ECMAScript 和 JavaScript 的关系

  ECMAScript是JavaScript的标准，JavaScript是ECMAScript的实现。换句话说，如果说JavaScript是一门语言的话，那么ECMAScript就是这门语言遵循的语法书。

- ES6 与 ECMAScript 2015 的关系

  2011年的时候，ECMAScript 5.1版本发布，ECMA组织就开始准备6.0版本了。但因为这个版本引入的语法功能太多了，所以不可能通过一个版本就将所有功能涵盖。所以，标准委员会决定，标准在每年的6月份正式发布一次，作为当年的正式版本。接下来的时间，就在这个版本的基础上做改动，直到下一年的6月份，草案就自然变成了新一年的版本。这样一来，就不需要以前的版本号了，只要用年份标记就可以了。

  就这样ES6的第一个版本就在2015.6正式发布了，正式名称就是《ECMAScript 2015标准》(简称ES2015)

- 总结

  因此，ES6 既是一个历史名词，也是一个泛指，含义是5.1版以后的 JavaScript 的下一代标准，涵盖了ES2015、ES2016、ES2017等等，而ES2015 则是正式名称，特指该年发布的正式版本的语言标准

第2集 使用babel工具搭建ES6项目环境

简介：讲解如何使用babel搭建ES6项目环境及babel的常用配置及用法

- node安装
  - http://nodejs.cn/download/
- 初始化工程项目

```
# 使用默认配置初始化项目
npm init -y
```

- 安装对应的依赖包

```
# 本地安装babel-cli及对应的转化规则
npm install --save-dev babel-cli babel-preset-es2015
```

- 创建项目目录
  - src目录用于存放编写的es6代码
  - dist目录用于存放由es6转化后的代码
- 配置babel
  - 新建`.babelrc`文件

```
// 配置用于转化当前代码的规则集
{
  "presets": ["es2015"]
}
```

  - babel的基本用法

```
# 这是局部安装babel的用法，全局安装的话就不用输入目录了

# 转码结果写入一个文件
# --out-file 或 -o 参数指定输出文件
$ node_modules/.bin/babel-node test.js --out-file test.js
# 或者
$ node_modules/.bin/babel-node test.js -o test.js

# 整个目录转码
# --out-dir 或 -d 参数指定输出目录
$ node_modules/.bin/babel-node src --out-dir dist
# 或者
$ node_modules/.bin/babel-node src -d dist
```

- 配置package.json文件

```
// 配置启动脚本的命令
"scripts": {
  // 监听index.js文件，一发生变化就执行babel转译的命令
  "dev": "babel src/index.js -w -o dist/index.js",
  // 将指定文件转译后输出到指定目录下
  "build": "babel src/index.js -o dist/index.js",
  // 将src目录下所有es6文件转译后输出到dist目录
  "buildall": "babel src -d dist"
}
```


第三章 全新的声明及数据赋值方式

第1集 新的变量声明关键字let与const

简介：详细讲解let与const关键词特点及使用的注意事项

- 作用域
  - 全局作用域
  - 局部作用域
  - 块级作用域
- let
  - 用法：声明一个变量
  - 特点：
    - 只在声明的代码块内有效
    - 在同一作用域内不允许重复声明

        - 没有变量提升
        - 暂时性死区
- const
    - 用法：声明一个只读的变量(可理解为常量)
    - 特点：同let命令
    - 注意事项：
        - 变量声明的同时必须立即赋值
        - 如声明的是简单类型的数据，变量的值不可改变
    - 实质：保证变量指向的内存地址所保存的数据不允许改动。

      简单类型如字符串、数字和布尔值，值就保存在变量指向的内存地址。而复杂类型的数据如对象、数组和函数，变量指向的内存地址，实际上是保存了指向实际数据的指针。所以 const 只能保证指针是固定的，至于指针指向的数据结构变不变就无法控制了，所以使用 const 声明复杂类型对象时要慎重。

第2集 探秘JS的第七种数据类型Symbol

简介：介绍Symbol是什么及使用场景

- ES6之前JavaScript的数据类型
    - Number(数字)
    - String(字符串)
    - Boolean(布尔值)
    - Object(对象)
    - Null(空对象指针)
    - Undefined(声明的变量未被初始化时)
- 引入的背景
    - 对象的属性名容易产生命名冲突，为保证键名的唯一性，故es6引入Symbol这种新的原始数据类型，确保创建的每个变量都是独一无二的
- 特点
    - Symbol类型的数据是类似字符串的数据类型，由于Symbol函数返回的值是原始类型的数据，不是对象，故Symbol函数前不能使用new命令，否则会报错。
    - 可选参数。由于控制台输出不同的Symbol变量时都是Symbol()，故为了区分，可在创建Symbol变量时传入参数进行区分。如：

      ```
      // 这里的a1，a2的作用可以说是为了备注，以至于我们输出Symbol变量时能够区分不同的变量。
      let a1 = Symbol('a1')
      let a2 = Symbol('a2')
      ```

- 用法
    - 定义对象的唯一属性名

      ```
      // 在对象里用Symbol作为属性名的三种写法
      let name = Symbol()
      ```

```
// 第一种方式：借助数组读取name变量，此时不能用点运算符,点运算符默认后面的参数是字符串
let a = {}
a[name] = 'Nick'

// 第二种方式：构造时声明
let a = {
  [name]: 'Nick'
}

// 第三种 Object.defineProperty
let a = {}
Object.defineProperty(a, name, { value: 'Nick' });
```

- 定义常量

```
// 定义字符串常量
const name = Symbol("Nick");
```

第3集 不可不知的解构赋值

简介：深入原理讲解解构赋值及其常见用法

- 介绍

  解构赋值可以理解为赋值操作的语法糖，它是针对数组或者对象进行模式匹配，然后对其中的变量进行赋值。代码书写上言简意赅，语义明确，也方便了对象数据的读取操作。

- 实质
  - ES6中只要某种数据有Iterator接口（也就是可以循环迭代），都可以进行数组的解构赋值。
- 使用场景
  - 数组解构

```
{
    let a, b, c
```

```
    [a, b, c] = [1, 2]
    console.log(a, b, c)  //1,2,undefined
}

{
    let a, b, c
    [a, b, c = 6] = [1, 2]
    console.log(a, b, c)  //1,2,6
}

{
    let a, other
    [a, ...other] = [1, 2, 3]
    console.log(a, other) //1,[2,3]
}
```

- 对象解构
- 字符串解构
- 布尔值解构
- 函数参数解构
- 数值解构

第四章 ES6新增的内置对象及已有对象的扩展

第1集 字符串的扩展方法及模板字符串

简介：介绍ES6提供的新的字符串方法及模板字符串

- ES5处理Unicode的缺陷

- 加强了对Unicode的支持

  在ES5中我们知道JavaScript 允许采用\uxxxx形式表示一个字符，其中xxxx表示字符的 Unicode 码点。这种表示法只限于码点在\u0000~\uFFFF之间的字符。超出这个范围的字符，必须用两个双字节的形式表示，但是ES5却无法正确的识别这个有两个字节组成的字符。ES6中，JavaScript增加了对超出\u0000~\uFFFF

  Unicode范围的字符支持。

  ES6的方案：将超过两个字节的组成的字符的码点放在一对花括号里就可以正确的识别。

- 字符串的遍历接口
- 扩展的API

方法	描述
includes(string, position)	判断字符串中是否包含指定字符串，返回值是布尔值
startsWith(string, position)	判断字符串的开头是否包含指定字符串，返回值是布尔值
endsWith(string, position)	判断字符串的尾部是否包含指定字符串，返回值是布尔值
repeat(n)	repeat() 方法返回一个新字符串，表示将院字符串重复n 次。
字符串补全	第一个参数是补全后的字符串长度，第二个参数是用于补全的字符串
padStart(length, str)	用于头部补全
padEnd(length, str)	用于尾部补全

- 模板字符串

```
// 语法
const name = "小明"
const age = 18
const hobbies = "游泳、跑步和打篮球"
// ES5写法
const str1 = '我的名字是'+name+', 我今年'+age+'岁，我喜欢'+hobbies
console.log(str1)
// ES6写法
const str2 = `我的名字是${name}，我今年${age}岁，我喜欢${hobbies}`
console.log(str2)
```

- 使用模板字符串的注意事项
    - 在模板字符串中如需使用反引号，反引号前要用反斜杠转义
    - 使用模板字符串表示多行字符串时，所有的空格和缩进都会被保留在输出之中
    - 模板字符串中引入变量，要用 ${变量名} 这样的形式引入才可以
    - 大括号中的值不是字符串时，将按照一般的规则转为字符串。比如，大括号中是一个对象，将默认调用对象的toString方法
    - 模板字符串中的${.....} 大括号内部可以放入任意的 JavaScript 表达式，可以进行运算、可以引用对象属性、可以调用函数、可以甚至还能嵌套，甚至还能调用自己本身

第2集 ES6和ES7之数组的扩展方法及扩展运算符的使用

简介：介绍ES6和ES7提供的新的数组方法及扩展运算符的使用

- 扩展运算符的使用
    - 复制数组
    - 分割数组
    - 将数组转化成参数传递给函数
- 新增的常用方法
    - fill
    - find
    - findIndex
    - Includes
    - flat
    - filter

第3集 ES6数组中map及reduce方法详解

简介：详细介绍ES6中map及reduce方法的使用

- map
- reduce

第4集 ES6和ES8之对象的新特性及新增方法

简介：介绍ES6提供的新的特性及对象方法

- 扩展运算符的使用
    - 复制对象
    - 给对象设置默认值
    - 合并对象
- 属性初始化的简写

- 对象方法的简写
- 可计算的属性名
- 新增方法
    - Object.is
    - Object.assign
    - Object.keys
    - Object.values
    - Object.entries

第5集 Map与WeakMap结构的特点

简介：详细介绍Map与WeakMap结构的特点

- 背景

JavaScript中的对象，实质就是键值对的集合(Hash结构)，但是在对象里却只能用字符串作为键名。在一些特殊的场景里就满足不了我们的需求了，正因为此，Map这一数据提出了，它是JavaScript中的一种更完善Hash结构。

- Map对象
  - 用于保存键值对，任何值(对象或者原始值)都可以作为一个键或一个值。
  - 使用介绍

```
// 通过构造函数创建一个Map
var m = new Map();
```

- 内置API

属性/方法	作用	例子
size	返回键值对的数量	m.size
clear()	清除所有键值对	m.clear()
has(key)	判断键值对中是否有指定的键名，返回值是布尔值	m.has(key)
get(key)	获取指定键名的键值对，如不存在则返回undefined	m.get(key)
set(key, value)	添加键值对，如键名已存在，则更新键值对	m.set(key, value)
delete(key)	删除指定键名的键值对	m.delete(key)

  - 遍历器生成函数
    - keys
    - values
    - entries
  - 遍历方法
    - forEach

第6集 Set与WeakSet结构的特点

- 介绍

  Set是ES6给开发者提供的一种类似数组的数据结构，可以理解为值的集合。它和数组的最大的区别就在于: 它的值不会有重复项。

- 基本使用

```
// 创建
let set = new Set();
let set2 = new Set([1,2,3])

// 添加元素
set.add(1)
```

- 特点
  - 成员值唯一
- 属性及方法

属性/方法	作用	例子
size	返回成员个数	s.size
clear()	清除所有成员	s.clear()
has(value)	判断键值对中是否有指定的值，返回值是布尔值	s.has(key)
delete(value)	删除指定值	s.delete(key)

- 用途

```
// 去重
let arr = [1,2,2,3,4,4,4];
let s = new Set(arr);
//结果：Set {1,2,3,4}

let newArr = Array.from(s);
//结果：[1,2,3,4],完成去重
```

- WeakSet
  - 数组成员必须是对象
  - WeakSet结构也提供了add( ) 方法，delete( ) 方法，has( )方法给开发者使用，作用与用法跟Set结构完全一致。
  - WeakSet 结构不可遍历。因为它的成员都是对象的弱引用，随时被回收机制回收，成员消失。所以WeakSet 结构不会有keys( )，values( )，entries( )，forEach( )等方法和size属性。

第7集 Map、Set与Array及Object间的区别

简介：详细介绍Map、Set与Array及Object间的区别

- 增删改查
- 类型转换

第8集 快速理解ES6中的代理Proxy和反射Reflect(上)

简介：手把手教你如何使用Proxy与Reflect实现简单的双向数据绑定

- Proxy
  - 概述

    正如Proxy的英译"代理"所示，Proxy是ES6为了操作对象引入的API。它不直接作用在对象上，而是作为一种媒介，如果需要操作对象的话，需要经过这个媒介的同意。
  - 使用方式

    ```
    /* @params
    ** target: 用Proxy包装的目标对象
    ** handler: 一个对象，对代理对象进行拦截操作的函数，如set、get
    */
    let p = new Proxy(target, handler)
    ```
  - 常用方法
- Reflect
  - 概述

    与Proxy相同，ES6引入Reflect也是用来操作对象的，它将对象里一些明显属于语言内部的方法移植到Reflect对象上，它对某些方法的返回结果进行了修改，使其更合理，并且使用函数的方式实现了Object的命令式操作
  - 使用方法
  - 常见用法
    - Reflect.has(obj, name)

      是 name in obj 指令的函数化，用于判断对象中是否有某一个属性， 返回值为布尔值

第9集 快速理解ES6中的代理Proxy和反射Reflect(下)

简介：手把手教你如何使用Proxy与Reflect实现简单的双向数据绑定

- 获取dom对象
- 设置代理对象
- 配置代理选项
- 添加事件
- 实现双向数据绑定

第五章 函数的扩展、类的概念及模块化

第1集 函数的扩展

简介：详细讲解ES6中函数的扩展

- 函数参数可设置默认值
- rest参数
- 扩展运算符
- 尾调用

第2集 深入浅出箭头函数

简介：详细讲解箭头函数的使用及注意事项

- 特点
    - 更短的函数
    - 不绑定this
- 注意事项
    - 什么情况下用箭头函数
    - 箭头函数没有argument

第3集 ES6中全方位理解类的概念

简介：详细讲解类的概念及如何实现类的封装及继承

传统的javascript中只有对象，没有类的概念。它是基于原型的面向对象语言。原型对象特点就是将自身的属性共享给新对象。

- new操作做了什么事
  - 返回（产生）了一个新的对象
  - 将这个空对象的_ proto_指向了构造函数内部的prototype
  - 将构造函数的this绑定到这个对象上(即new创建的对象，其函数体内的this指向的是这个对象)
  - 返回到这个新对象上

```
const obj = {}
obj.__proto__ = 构造函数.prototype;
构造函数.call(obj);
```

- es6 class的概念
  - 定义类
  - 类的继承
  - 子类向父类传递参数
  - 静态方法
  - 静态属性

第4集 一览js中模块化开发(import和export)

简介：介绍js的发展过程中如何实现模块化及import和export的使用

- 背景：

  ES6之前是没有类的概念的，也就没有模块这一说法。理想情况下，开发者应该只注重核心业务的开发，对于其他有一定通用性的业务可以直接引入别人的代码，也就是模块。多人开发，本应如此。

- 提出的方案
  - CommonJS
  - Commonjs是作为Node中模块化规范以及原生模块面世的
  - AMD和RequireJs
    - AMD是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义"。它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到**所有依赖加载完成之后**（前置依赖），这个回调函数才

会运行。
- Import 和 export

愿景："让编程不在难学，让技术与生活更加有趣" 更多课程请访问[xdclass.net](xdclass.net)

第六章 深入解析JavaScript中的异步编程

第1集 什么是异步编程及JavaScript的异步实现

简介：介绍异步编程的概念及前期JavaScript实现异步编程的几种方法

- 什么是同步

  当一个"调用"发出时，在没有得到结果之前，这个"调用"就会阻塞后面代码的执行，得到结果的时候才会返回。换句话说，"调用者"要主动等待代码的执行结果，得到返回结果后，程序才会继续运行。

- 什么是异步

  "调用"发出的时候，就直接返回了，对应的结果会通过状态、通知来告诉"调用者"或通过回调函数处理这个调用。异步调用发出后，不会阻塞后面的代码。

- JavaScript中为什么要引入异步这个概念
  - JavaScript是单线程的
  - 同步代码会阻塞后面的代码
  - 异步不会阻塞程序的运行
- JavaScript中异步的实现
  - 回调函数

```
// 一层的回调函数
$.get('http://api.xxx.com/xxx', callback)
```

```
/**
*    请求后台类型列表，根据类型名称获取要请求的列表id，再根据id获取列表详细内容
*    难以维护，这就是回调地狱了
**/
$.ajax({
    url: "type",
    data:1,
    success: function (a) {
        $.ajax({
            url: "list",
            data:a,
            success: function (b) {
                $.ajax({
                    url: "content",
                    data:b,
                    success: function (c) {
                        console.log(c)
                    }
                })
            }
        })
    }
})
```

- setInterval和setTimeout

- Promise
- Generator
- async

第2集 解决回调地狱提出的新方案—Promise

简介：深入讲解Promise的概念及其原理

背景：为了解决"回调地狱"的问题，提出了Promise对象

- 什么是回调地狱
- 什么是Promise

  Promise是一个对象，也可以说是一种编程思想。应用的场景就是"当xxx执行完毕的时候，then执行xxx动作"。Promise里不仅可以存放着异步的代码，也可以放同步的代码。
- Promise的使用
  - 封装一个Promise
  - 捕获异常
- Promise.all方法
- Promise.race

第3集 JS的数据结构中统一的遍历接口Iterator和for...of循环

简介：介绍什么是Iterator及其作用和与for...of循环的关系

- 什么是Iterator

  Iterator(遍历器)是一种接口，目的是为了给不同的数据结构提供统一的循环方式，任何数据结构如果部署了Iterator接口，就能够实现遍历的操作。
- Iterator的作用
  - 为不同的数据结构，提供一个统一的、简便的访问接口
  - 将数据成员按照一定的顺序输出
  - 提供给ES6中的for...of的这个循环语句进行使用
- 什么结构具备原生的Iterator接口
  - Array

    - String
    - Set
    - Map
    - 函数的argument对象
- 默认的Iterator接口
    - Symbol.iterator
        - 本质

          是一个函数，就是当前的数据集合默认的遍历器生成函数，执行这个函数，就会返回一个遍历器。
        - 返回值
            - 返回一个遍历器对象。这个对象里的显著特点就是有一个next()方法。每次调用next都会返回一个描述当前成员的信息对象，具有value和done两个属性。

第4集 ES8之更直观的异步编程写法—Generator函数

简介：讲解如何创建一个Generator函数及其执行机制

- 什么是Generator

用于生成一个迭代器Iterator

- next()
- yield
- 应用场景

第5集 ES8之越来越优雅的异步编程—async

简介：介绍async函数的语法及使用的注意事项

- 什么是async

  async是异步的简写，用于声明一个函数是异步函数
- await到底在等啥

await 等待的是一个表达式，这个表达式的计算结果是 Promise 对象或者其它值（换句话说，就是没有特殊限定）

第七章 课程总结及项目遇到的问题

第1集 课程总结及项目遇到的问题

简介： 课程总结及项目遇到的问题

- 开发中常用到的知识
  - 解构赋值
    - 对象的解构
    - 数组的解构
  - set、map方法
  - Object.assign()
  - for of循环
    - Object的for of通过[Symbol.iterator]
  - Promise函数
    - 结合async函数，异步编程同步化
- 项目遇到的问题
  - babel无法转译一些api

    babel默认只是转译新标准引入的语法，比如ES6的箭头函数，不转换新的API，比如Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise等全局对象，以及一些定义在全局对象上的方法（比如Object.assign）都不会转码。

    解决方法：

    使用babel-polyfill，为当前环境提供一个垫片。

```
npm install --save babel-polyfill

// 找到node_modules目录下的babel-polyfill下dist目录下的polyfill.js文件

// 用script标签引用
```

- 无法引入模块
  - 在script标签下添加`type="module"`属性