ES6标准入门

什么是ES6

- ECMAScript 6（简称ES6），又称ECMAScript2015(ES2015)
- 2015年6月正式发布，是JavaScript语言的下一代标准

为什么要学

- 更严谨的语法，更高效的编码
- 更新的特性，更多的功能

为什么要学

- 新特性举例：箭头函数的使用场景——数组遍历

```
let arr1 = [1,2,3]
arr1.map(function (x) {
  return x * x;
});
```

```
let arr2 = [1,2,3]
arr2.map(x => x * x);
```

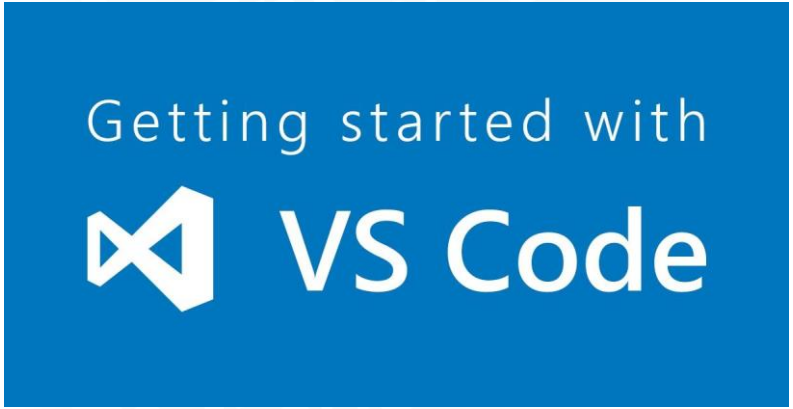
为什么要学

- 更严谨的语法，更高效的编码
- 更新的特性，更多的功能
- 主流前端框架(Vue/React/Angular等)、大厂都在用ES6+
- 兼容性解决方案成熟（Babel）

章节介绍

- ES6与JavaScript的关系
- 变量、常量、解构赋值
- 函数、箭头函数
- Class vs function
- 模块化
- 编程风格

环境及开发工具



ES6与JavaScript的关系

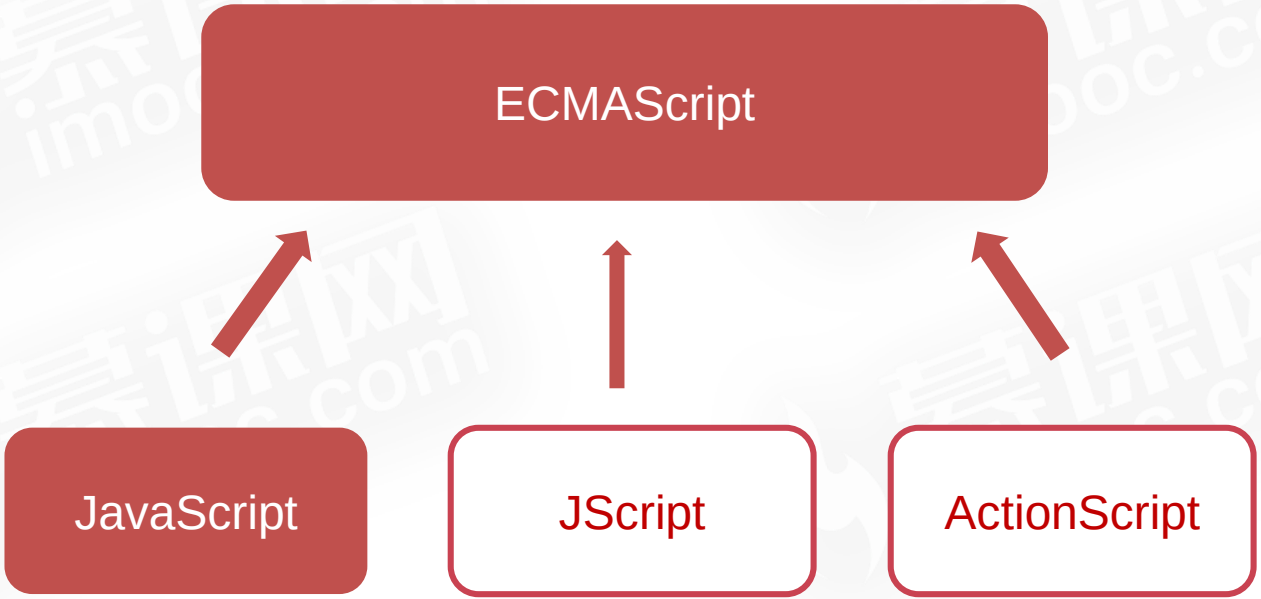
JavaScript发展史

- 1996年11月，Netscape将JavaScript提交给标准化组织ECMA
- 1997年，ECMA发布ECMAScript1.0
- 1999年12月，ECMAScript 3.0版发布，成为JavaScript的通行标准

JavaScript发展史

- 2011年6月，ECMAscript 5.1版发布，并且成为ISO国际标准（ISO/IEC 16262:2011）
- 截至 2012 年，所有浏览器都完整的支持ECMAScript 5.1，旧版本的浏览器至少支持ECMAScript 3 标准
- 2015年6月17日，ECMAScript 6发布正式版本，即ECMAScript 2015

ES6与JavaScript的关系



JavaScript周边

- TypeScript

  微软开发，最终会被编译成JavaScript代码

- CoffeScript

  一门编译到JavaScript的小巧语言

常用的ES6新特性

- 精简的箭头函数

```
let  sum = (a, b) => a + b
```

- 严格的变量声明

```
let  age = 20
const COURSE =  “Python”
```

常用的ES6新特性

- 更方便的解构赋值

  let [a, ...b] = [1, 2, 3, 4]

- Promise容器实现异步编程

  new Promise((resolve, reject)=>{})

变量、常量、解构赋值

变量

变量

- 声明变量

```
var a = 100
let b = 'hello'
```

let的三大特性

- 特征一：不存在变量提升，必须要先声明，再使用

```
console.log(a); // 报错ReferenceError
let a = 2;
```

let的三大特性

- 特征二：不能重复声明

```
let b = 100
let b = 'hello'  // 报错 SyntaxError
```

let的三大特性

- 特征三：块级作用域，变量只在代码块内有效

```
function func() {
  let n = 5;
  if (true) {
    let n = 10;
  }
  console.log(n); // n = ?
}
```

常量

常量

- 什么是常量？

  一旦声明，常量的值就不能改变

常量

- 声明常量

```
const PAGE_SIZE = 100
const PI = 3.1415926
```

const的三大特征

- 特征一：声明必须赋值(必须初始化)

```
const NAME;  // 报错 SyntaxError
NAME = 100
```

const的三大特征

- 特征二：常量是只读的，不能重新赋值

```
const PI = 3.1415926
PI = 3.141592654  // 报错 TypeError
```

const的三大特征

- 特征三：块级作用域，只在代码块内有效

```
if (true) {
    const PI = 3.1415926
}
console.log(PI)  // 报错 ReferenceError
```

解构赋值

解构赋值

- 场景一：数组的解构赋值
- 场景二：对象的解构赋值
- 场景三：字符串的解构赋值

场景一：数组的解构赋值

- 按顺序将值赋值给对应的变量

```
let [a, b, c] = [1, 2, 3];
console.log(a)
console.log(b)
console.log(c)
```

场景一：数组的解构赋值

- ...表示解构运算符，将剩余的内容赋值

```
let [c, ...d] = [1, 2, 3];
console.log(c)
console.log(d)
```

场景一：数组的解构赋值

- 解构赋值失败，变量值为undefined

```
let [e, f] = [1];
console.log(e)
console.log(f)
```

场景一：数组的解构赋值

- 防止解构失败，给变量默认值

```
let [g, h=100] = [1];
console.log(g)
console.log(h)
```

场景二：对象的解构赋值

- 按顺序将值赋值给对应的变量

```
let { foo, bar } = { foo: 'aaa', bar: 'bbb' };
console.log(foo)
console.log(bar)
```

场景二：对象的解构赋值

- 可以解构对象中的常量、方法

```
console.log(Math.PI)
let { PI, sin } = Math
console.log(PI)
console.log(sin(PI/2))  // sin(90°)
```

场景二：对象的解构赋值

- 结构赋值失败，则为undefined，可设置默认值

```
let {x, y = 5} = {x: 1}
console.log(x)
console.log(y)
```

场景二：对象的解构赋值

- 重新指定变量名称

```
let { color: sky } = {color: 'blue'  }
console.log(sky)
```

场景二：对象的解构赋值

- 复杂对象的解构赋值

```
let {title, author: {name, age}} = {
        title: '新闻标题',
        author: {
                name: '张三',
                age: 23
        }
}
console.log(name)
console.log(age)
```

场景二：对象的解构赋值

- 对象解构赋值的应用

```
$.post(url, data, function({code, objects}) {
  if (code === 200) {
    // 请求成功，执行正确的业务逻辑代码
  }
})
```

场景三：字符串的解构赋值

- 字符串可以看做是“伪数组”（注：它不是真的数组）

```
const [a, b, c, d, e] = 'hello'
console.log(a)
console.log(b)
let {length} = 'hello'
console.log(length)
```

函数、箭头函数

设置默认值

- 思考：ES6之前的函数如何指定默认值

```
function point(x, y) {
    x = x || 0;
    y = y || 0;
}
```

```
function point(x = 0, y = 0) {
    console.log(x)
    console.log(y)
}
```

设置默认值

- 未设置默认值且不传对应参数时，值为undefined

```
function point(x, y = 0) {
    console.log(x)；  // undefined
    console.log(y)；
}
```

设置默认值

- 带默认值的参数顺序

```
function point(x = 0, y, z) {
    console.log(x) ;
    console.log(y) ;
    console.log(z) ;
}
```

温馨提示：请将带默认值的参数放于最后

设置默认值

- 注意：参数变量(形参)不可重复声明

```
function point(x = 0, y = 0) {
    let x = 0 ;
    const y = 1 ;
}
```

设置默认值

- 当形参为对象时，可使用解构赋值

```
function fetch(url, { body = '', method = 'GET',
headers = {} }) {
  $.ajax({
      url,
      method
  })
}
```

对象中的函数简写

- 示例

```
let UserA = {
        name: '张三',
        age: 23,
        info: function( ) {
                return this.name + this.age
        }
}
```

对象中的函数简写

- 改进后

```
let UserB = {
        name: '张三',
        age: 23,
        info() {
            return this.name + this.age
        }
}
```

箭头函数

- 定义箭头函数

```
let f1 = v => v;

// 等同于
let f1 = function (v) {
  return v;
};
```

```
let f2 = (a, b) => a + b;

// 等同于
let f2 = function (a, b) {
  return a + b;
};
```

箭头函数

- 函数体包含多条语句，使用{}包含语句块

```
let f3 = (a, b) => {
    console.log(a, b) ;
    return a + b ;
}
```

箭头函数

- 箭头函数的使用场景——数组遍历

```
let arr1 = [1,2,3]
arr1.map(function (x) {
  return x * x;
});
```

```
let arr2 = [1,2,3]
arr2.map(x => x * x);
```

箭头函数

- 注意：箭头函数体内的this指向定义时所在的对象，而非实例化后的对象（在函数定义时绑定）

```
let UserC = {
        name: '张三',
        age: 23,
        info: () => {
                return this.name + this.age
        }
}
```

理解this

- 指函数执行过程中，自动生成的一个内部对象，是指当前的对象，只在当前函数内部使用
  - ✓ 运行时跟运行环境绑定

    浏览器: this === window

    Node 环境：this === global
  - ✓ 当函数被作为某个对象的方法调用时，this指向那个对象

理解this

- 再次理解箭头函数中的this(指向定义时所在的对象)

```
let MyObj = {
  a() {
    console.log('a', this)   // MyObj
    setTimeout(()=> {
      console.log('timeout', this)  // MyObj
    }, 100)
  },
  b: () => {
    console.log('b', this)  // window
  }
}
```

Class vs Function

面向对象编程

- 回顾：ES6以前的面向对象编程

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype.toString = function () {
  return '(' + this.x + ', ' + this.y + ')';
};

var p = new Point(1, 2);
```

面向对象编程

- ES6中引入class关键字,实现面向对象编程

```
class Point {
  constructor(x, y) {    // 构造函数
    this.x = x;
    this.y = y;
  }

  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}
```

函数的调用

- 关于类的实例

使用new关键字来实例化类

```
var p = new Point(1, 2);
```

其中p就是类的实例

构造函数

- 关于构造函数constructor

  构造函数（ constructor ）是类的默认方法，通过new命令生成对象实例时，会自动调用该方法。

  一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

this指向

- 关于this的指向

  类方法中的this指向类的实例

  静态方法中的this指向类

静态方法

- 可以直接通过类来调用的方法（无法通过类的实例来调用）

```
class Foo {
  static staticMethod() {
    return 'I am static method';
  }
}

Foo.staticMethod()
```

注意：静态方法中的this指的是类，而不是其实例

属性表达式

- 类的属性名，可以采用表达式，使用[]来引用

```
let methodName = 'getArea';
class Square {
  ['a' + 'bc']: 123,
  [methodName]() {
    // 业务逻辑代码
  }
}
```

类的继承

- ES6中使用extends关键字实现类的继承

```
class Animal {
  eat () {
    return 'Food'
  }
}

new Animal().eat()
```

```
class Cat extends Animal {
  eat() {
    return 'Fish'
  }
}

new Cat().eat()
```

类的继承

- 使用super关键字调用父类方法

```
class Cat extends Animal {
  eat() {
    let rest = super.eat();  // 调用父类方法
    return rest + ' Fish'
  }
}
new Cat().eat()
```

模块化

模块化

- 思考：什么是模块化？它有什么好处？

模块化的好处

- 增强代码的可维护性
- 增强代码的可阅读性
- 增强代码的可扩展性

JS中的模块化

- ES6以前
  - ✓ 服务器端：CommonJS
  - ✓ 浏览器端：AMD、CMD

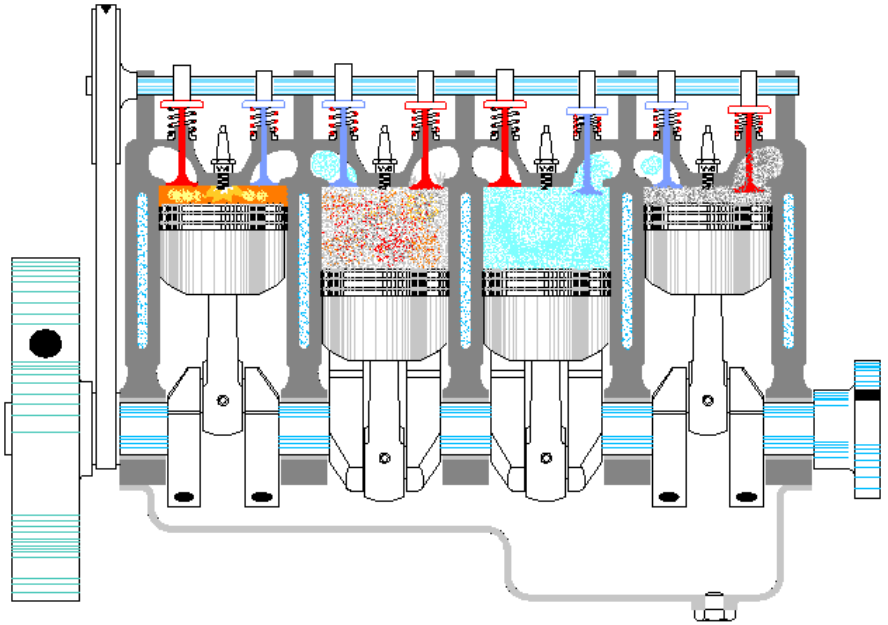
模块化的实现

- 模块是对内部逻辑的封装，只对外提供接口

模块化的实现

- 举例：汽车发动机模块

输入：汽油
输出：动力



模块化的实现

- 模块是对内部逻辑的封装，只对外提供接口
- 一个js文件就是一个模块
- ES6中使用export关键字对外暴露接口（导出）
- ES6中使用import关键字导入一个模块

export的使用

- 方式一：直接导出

```
export let a = 1
export const NAME = 'JavaScript'
export function sayHello() {
  console.log('Hey, boy!')
}
```

export的使用

- 方式二：批量导出

```
let a = 1   // 变量

const NAME = 'JavaScript' // 常量

// 函数
function sayHello() {
  console.log('Hey, boy!')
}

export {a, NAME, sayHello}
```

export的使用

- 通过as关键字指定别名

```
let a = 19
export { a as age }
```

export的使用

- 方式三：使用export default 默认导出

```
// a.js
export default 1

// b.js
let b = 2
export default b  // 可以理解为赋值

// f.js
export default function () {
  console.log('from default')
}
```

import的使用

- 方式一：导入需要的部分内容

```
// 从myFirstModule.js导入需要的函数
import { sayHello } from './myFirstModule.js'

// 执行函数
sayHello()
```

import的使用

- 方式二：导入全部内容，使用as指定别名

```
// 从myFirstModule.js导入全部
import * as myModule from './myFirstModule.js'

// 执行函数
myModule.sayHello()
```

踩坑指南

- 第一坑：不能在块级作用域内执行导入导出

```
if (true) {
  export let a = 1   // SyntaxError
}
```

踩坑指南

- 第二坑：不能直接导出变量的值

```
export 3.1415926 // SyntaxError
```

踩坑指南

- 第三坑：注意import的顺序

```
let a = 1
import b from './b.js' // 报错
```

ES6编程注意事项

编程风格

- 思考：书写ES6代码有哪些需要注意的地方

  ESLint——代码静态语法检查

编程风格

- ESLint 要点

1. 使用单引号(')代替双引号(")
2. 去掉语句结尾的分号（;）
3. 运算符前后的空格

编程风格

- 思考：我要兼容IE8怎么办？

兼容性问题解决

- 思路一：shim，将不同的API封装成一种

  如：$.ajax 封装了 XMLHttpRequest 和 IE 用 ActiveXObject 方式创建 xhr 对象

- 思路二：polyfill

  如：部分浏览器不支持箭头函数，我们将箭头函数转换成普通函数

兼容性问题解决

- Polyfill要自己做吗？工作量是否很大？

Babel是一个javaScript编译器

babel的功能

- 语法转换, 将ES6+的语法转换为向后兼容的JavaScript语法
- 通过 Polyfill 方式在目标环境中添加缺失的特性 (通过 @babel/polyfill 模块)
- 可转换JSX语法

Babel的安装

- 第一步，安装NodeJs
- 第二步，安装cnpm/yarn
- 第三步，安装babel

cnpm install –g @babel/core @babel/cli

Babel的使用

- 将箭头函数转换成普通函数

```
let getYear = () => new Date().getFullYear()
```

Babel的使用

- 安装babel-cli

```
cnpm install --save-dev @babel/cli
```

- 安装preset

```
cnpm install --save-dev @babel/preset-env
```

Flex布局

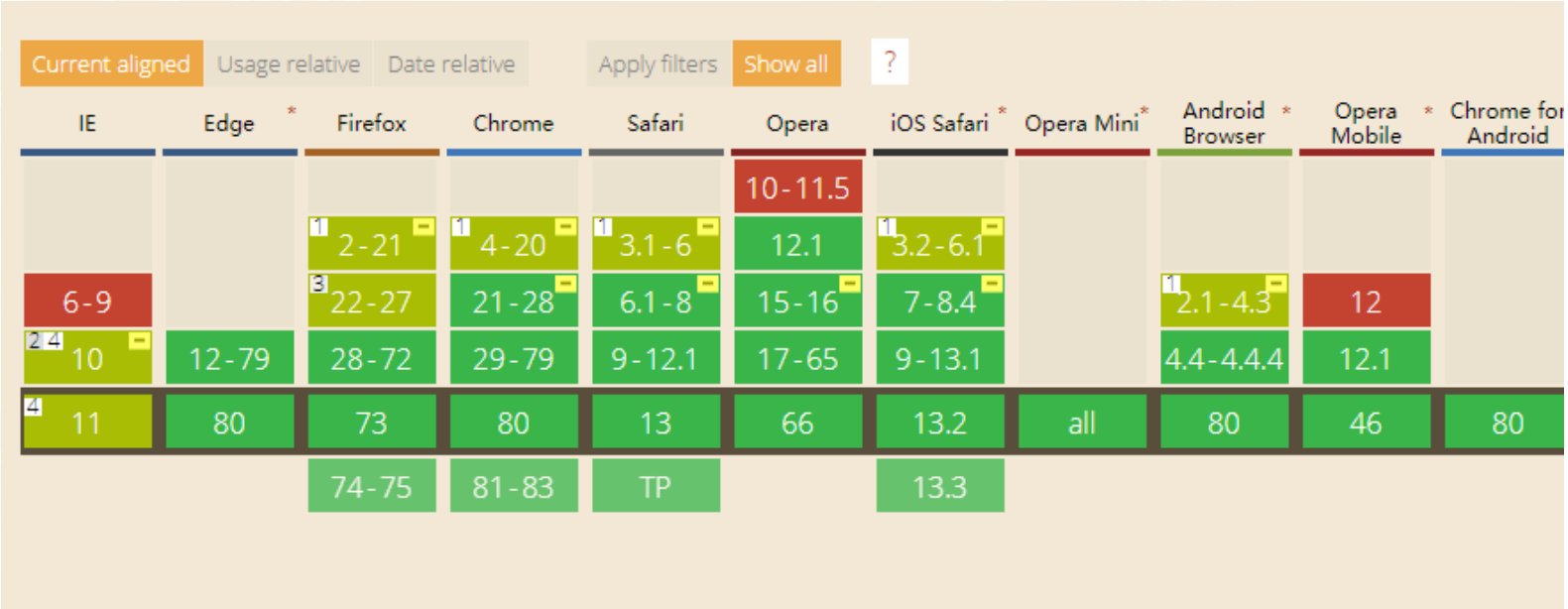
Flex布局

- 什么是Flex布局？

  Flex(Flexible Box ）——弹性布局

Flex布局的兼容性

- CanIUse



Flex布局

- 基本概念

  容器（Flex Container）

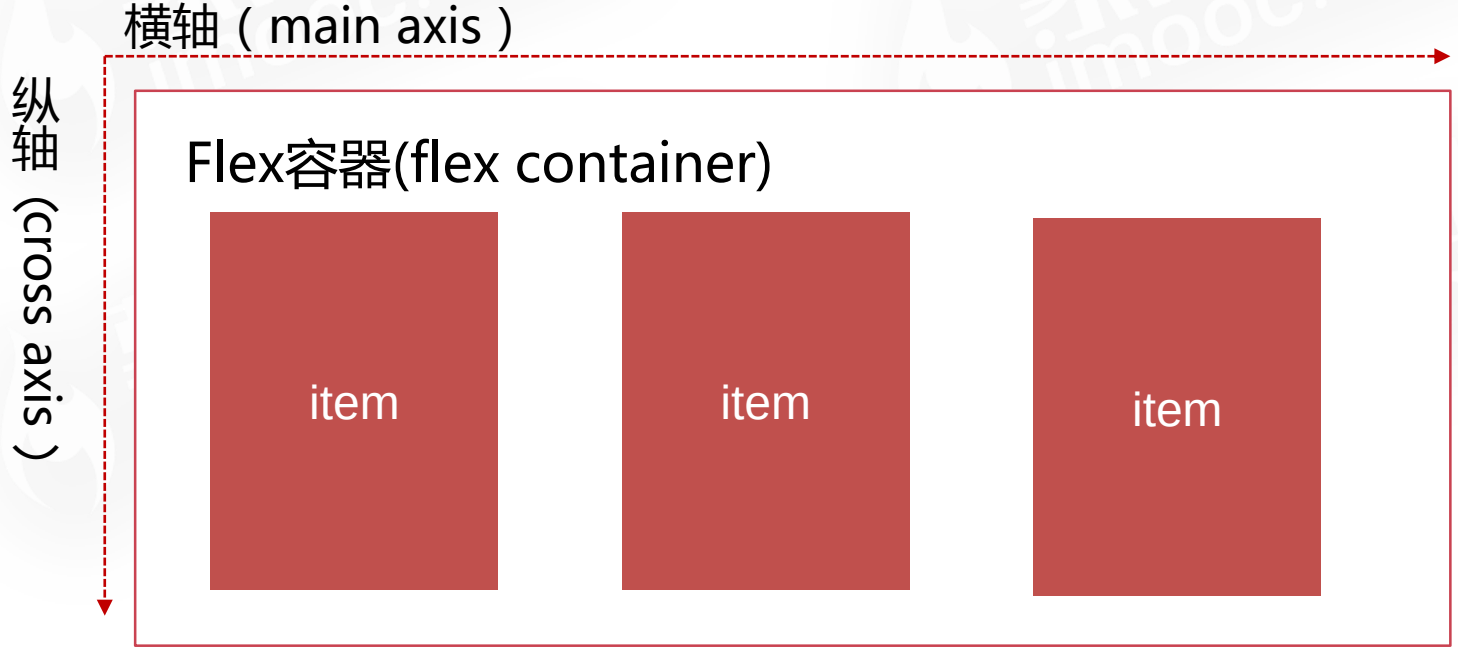
  元素项(Flex Item)

  水平轴，横轴，主轴（main axis）

  垂直轴，纵轴，交叉轴(cross axis)

Flex布局

- 基本概念


横轴（main axis）
纵轴（cross axis）
Flex容器(flex container)
item
item
item

Flex布局

- Flex容器(Flex Container)的属性
  1. flex-direction——容器内元素的排列方向
  2. flex-wrap——容器内元素的换行行为
  3. flex-flow——direction和wrap的复合属性

Flex布局

- Flex容器(Flex Container)的属性

4. justify-content——元素在横轴上的对齐方式
5. align-items——元素在纵轴上的对齐方式
6. align-content——多行元素的对齐方向

Flex容器的属性

- flex-direction ——容器内元素的排列方向

  row（默认值）：主轴为水平方向，从左到右

  row-reverse：主轴为水平方向，从右到左

  column：主轴为垂直方向，从上到下

  column-reverse：主轴为垂直方向，从下到上

Flex容器的属性

- flex-wrap——容器内元素的换行行为

  nowrap（默认）：不换行

  wrap：换行，首行在上方

  wrap-reverse：换行，首行在底部

Flex容器的属性

- flex-flow——direction和wrap的复合属性

  flex-flow：<' flex-direction '> || <' flex-wrap ‘>

  默认值为row nowrap

Flex容器的属性

- justify-content——元素在水平轴上的对齐方式

1. flex-start（默认值）：左对齐
2. flex-end：右对齐
3. center： 居中对齐
4. space-between：两端对齐，item之间的间隔相等
5. space-around：每个item两侧的间隔相等

Flex容器的属性

- align-items——元素在垂直轴上的对齐方式
  1. stretch（默认值）：占满整个容器的高度
  2. flex-start：与纵轴起点对齐
  3. flex-end：与纵轴终点对齐
  4. center：与纵轴中间对齐
  5. baseline: 与基线对齐

Flex容器的属性

- align-content——多行元素的对齐方向

1. stretch（默认值）：占满整个容器的高度
2. flex-start：与纵轴起点对齐
3. flex-end：与纵轴终点对齐
4. center：与纵轴中间对齐
5. space-between：与纵轴两端对齐，item之间的间隔平均分布
6. space-around：每根轴线两侧的间隔相等

Flex布局

Flex布局

- 元素项 (Flex Item)的属性

1. order——排序规则，越小越靠前排列
2. flex-grow——放大（撑开）比例
3. flex-shrink——收缩比例
4. flex-basis——水平方向的大小
5. flex——grow、shrink、basis的复合属性
6. align-self——元素在纵轴上的对齐方式

元素项 (Flex Item)的属性

- order——排序规则，越小越靠前排列

  默认为0

  数值越小，排列越靠前

元素项 (Flex Item)的属性

- flex-grow——放大（撑开）比例

  0(默认值)：如果存在剩余空间，也不放大

  1：如果存在剩余空间，均分剩余空间

  既有0，也有1：如果存在剩余空间，0的不撑大，1撑大

  >=1 ：如果存在剩余空间，按比例分配剩余空间

元素项 (Flex Item)的属性

- flex-shrink——收缩比例

  1（默认）：如果空间不足，该item将缩小

元素项 (Flex Item)的属性

- flex-basis——水平方向的大小

  如果所有子元素的基准值之和大于剩余空间，则会根据每项设置的基准值，按比率伸缩剩余空间

  auto（默认值）： 无特定宽度值，取决于其它属性值

  <length>： 用长度值来定义宽度,不允许负值

  <percentage>： 用百分比来定义宽度,不允许负值

  content： 基于内容自动计算宽度

元素项 (Flex Item)的属性

- flex——grow、shrink、basis的复合属性

  1: 则其计算值为1 1 0%

  auto: 则其计算值为1 1 auto

  none:则其计算值为0 0 auto

  0 auto: 则其计算值为0 1 auto

元素项 (Flex Item)的属性

- align-self——元素在纵轴上的对齐方式

  1. auto（默认值）：父元素的'align-items'值，若无，则为'stretch'
  2. stretch：占满整个容器的高度
  3. flex-start：与纵轴起点对齐

元素项 (Flex Item)的属性

- align-self——元素在纵轴上的对齐方式

4. flex-end：与纵轴终点对齐
5. center：与纵轴中间对齐
6. baseline：与基线对齐

课程总结

课程总结

- 知识点总结
- 全栈工程师学习前端需要注意的地方

知识点总结

- 关于HTML/HTML5

  标签/元素的基本使用

  块级元素/行内元素

  语义化的标签

知识点总结

- 关于CSS/CSS3

  CSS基础语法

  选择器（标签选择器、class、id、伪类等）

  浮动和定位

  Flex布局

知识点总结

- 关于JavaScript

  JavaScript基本语法

  JS事件、操作DOM

  进阶：ES6+

  框架相关：jQuery、Bootstrap、Vue等

课程总结

- 知识点总结
- 全栈工程师学习前端需要注意的地方

注意点

- CSS语法那么多，记不住怎么办

注意点

- CSS布局那么复杂，用不熟练怎么办
- 又是JavaScript，又是ES6，头都要炸了

既然选择了全栈，没有诀窍，多练习

多数情况下，一定要看得懂代码，不必追求精通

注意点

- 前端有哪些是最难的？

  前端没有最难的，就是多而杂，涉及的面广

  难以掌握的：CSS布局，JS原型链

踩坑指南

- CSS定位不熟悉，导致页面错乱
- 浏览器兼容性问题，不知从何下手
- jQuery操作DOM，页面中标签ID属性不唯一
- jQuery中ajax回调依赖，下次请求依赖于多次请求的结果

踩坑指南

- 箭头函数的this指向
- ES6+的新语法，模块化中导入导出
- 对象的解构赋值容易把人绕晕

学习建议

- 路漫漫其修远兮，吾将上下而求索
- 培养快速学习能力，持续学习