
讲不完的
ESX
— LYNN

会 JS 了不起啊！

JAVASCRIPT VS ECMASCRIPT

1999年12月， ECMASCRIPT 3.0版发布

2009年12月，ECMASCRIPT 5.0版正式发布

2015年6月，ECMASCRIPT 6正式发布，并且更名为“ECMASCRIPT 2015”

2016年6月，《ECMASCRIPT 2016 标准》发布

2017年6月，《ECMASCRIPT 2017 标准》发布

2008

它提高了JAVASCRIPT的性能

推动了语法的改进和标准化

2009

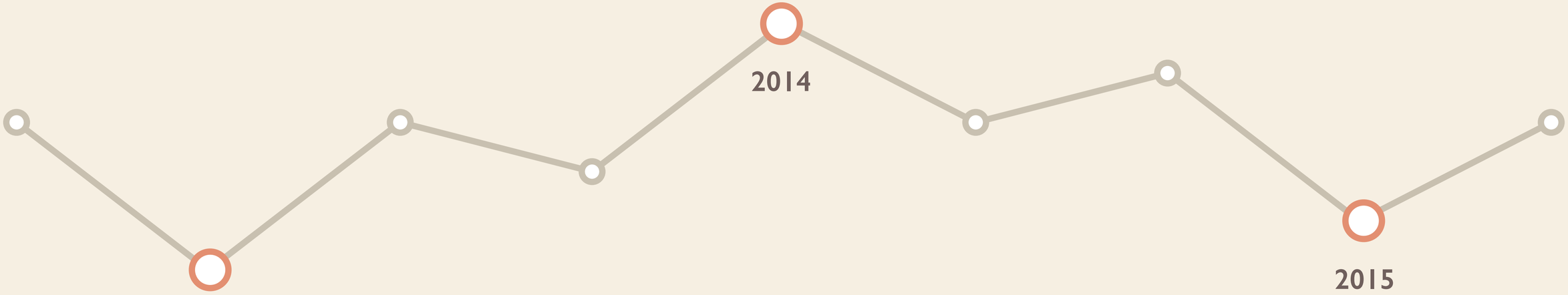
JAVASCRIPT可以用于

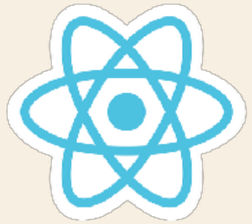
服务器端编程

2009

MVVM、模块化、自动化双向

数据绑定





引入了新的JSX语法

使得UI层可以用组件开发

构建用户界面的渐进式框架

专注于MVVM 模型的VIEWMODEL 层

JAVASCRIPT可以用来开发

手机APP

JS

Getting started with ESX

Core features overview


ARROW FUNCTION
LET + CONST
MODULES
ES6~8
DESTRUCTURING
PROXIES
TEMPLATE STRINGS
PROMISE/ASYNC + AWAIT
CLASSES

ESX 异步队列的起因

1. 为什么会出现异步队列

2. 什么是同步(SYNCHRONOUS)和异步(ASYNCHRONOUS)

```
console.log(‘start’)
setTimeout(() => {
  console.log('World!');
}, 10)
console.log('Hello')
```

```
// 结果是
// start
// Hello
// World!
```

ESX 处理异步的方式

1. 回调函数

2. 事件监听

3. 发布/订阅

4. PROMISE

5. ASYNC/AWAIT

回调函数

优点：简单，容易理解，部署

缺点：不利于代码的阅读和维护，各个部分之间高度耦合，每个任务只能指定一个回调函数。多层嵌套，就会出现“回调地狱”

```
function f1(callBack) {
    setTimeOut(() => {
        callBack()
    }, 1000)
}
f1(f2)
```

事件监听

优点：容易理解，每个事件可以指定多个回调，而且可以“去耦合”，有利于实现模块化

缺点：整个程序变成事件驱动，运行流程变得很不清晰

```
f1.on('done', f2)
function f1() {
    setTimeOut(() => {
        // f1的任务代码
        f1.trigger('done')
    }, 1000)
}
```

发布/订阅

这种方法的性质与”事件监听”类似，但是明显优于“事件监听”。

因为我们可以通过查看”消息中心”， 了解存在多少信号、每个信号有多少订阅者，从尔监控程序的运行。

而且可以实现多个任务的绑定。

```
jQuery.subscribe("done", f2);
function f1() {
    setTimeOut(() => {
        jQuery.publish("done");
    }, 1000)
}
jQuery.unsubscribe("done", f2);
```

PROMISE 的 API

then(onResolve)

catch(onReject)

resolve(callback)

reject(callback)

all(Array)

race(Array)

```
const asyncFun  = (ms) => {
  return New Promise ((resolve, reject) => {
    setTimeout (() => {
      return resolve(ms)
    }, ms)
  })
}

asyncFun('1000')
  .then(asyncFun('1000'))
  .then(asyncFun('1000'))
  .catch(exceptionHandler)
```

PROMISE

优点：回调函数变成了链式写法，程序的流程可以看得很清楚。

缺点：编写和理解，都相对比较难。

PROMISE 的 问题

1. 同步调用和异步调用同时存在导致混乱

2. 需要单独 catch 某个异步的错误

3. finally 问题

ES7 下的异步 ASYNC/AWAIT

ASYNC表示这是 个ASYNC函数,AWAIT 只能用在这个函数 。

AWAIT 表示在这等待PROMISE返回结果 ， 再继续执行。

AWAIT 后面跟着的应该是 个PROMISE 对象

AWAIT等待的虽然是PROMISE对象，但不必写.THEN()，直接可以得到返回值。

PROMISE VS ASYNC/AWAIT

ESX 模块化

1. COMMONJS

2. AMD

3. CMD / UMD （没有用过，无权讲解）

4. ES6

COMMONJS 规范

```
// a.js
var x = 5;
var addX = function (value) {
  return value + x;
};
 module.exports.x = x;
module.exports.addX = addX;
```

```
var a = require('./a.js');
console.log(example.x); // 5
console.log(example.addX(1)); // 6
```

COMMONJS 规范

优点： **CommonJS**规范主要用于服务器端，解决了依赖、全局变量污染的问题，这也是**js**运行在服务器端的必要条件。

缺点： 由于 **CommonJS** 是同步加载模块的，在服务器端，文件都是保存在硬盘上，所以同步加载没有问题，但是对于浏览器端，需要将文件从服务器端请求过来，那么同步加载就不适用了，所以，**CommonJS**是不适用于浏览器端的。

AMD 规范

```
define(function () {
  var alertName = function (str) {
    alert(“lam" + str);
  }
  var alertAge = function (num) {
    alert("I am " + num + " years old");
  }
  return {
    alertName: alertName,
    alertAge: alertAge
  };
});
```

```
require(['alert'], function (alert) {
  alert.alertName('JohnZhu');
  alert.alertAge(21);
});
```

AMD 规范

优点： 适合在浏览器环境中异步加载模块。可以并行加载多个模块。

缺点： 提高了开发成本，并且不能按需加载，而是必须提前加载所有的依赖。

ES6 模块化的规则

1. 每一个模块只加载一次， 每一个JS只执行一次， 如果下次再去加载同目录下同文件，直接从内存中读取。
2. 每一个模块内声明的变量都是局部变量， 不会污染全局作用域；
3. 模块内部的变量或者函数可以通过 EXPORT 导出；
4. 一个模块可以导入别的模块

```
import { stat, exists, readFile } from ‘fs’;
export const a = 1
```

COMMONJS 模块 VS ES6 模块

```
// CommonJS模块
let { stat, exists, readFile } = require('fs');

// 等同于
let _fs = require('fs');
let stat = _fs.stat;
let exists = _fs.exists;
let readfile = _fs.readfile;
```

运行时候加载

```
// ES6模块
import { stat, exists, readFile } from 'fs';
```

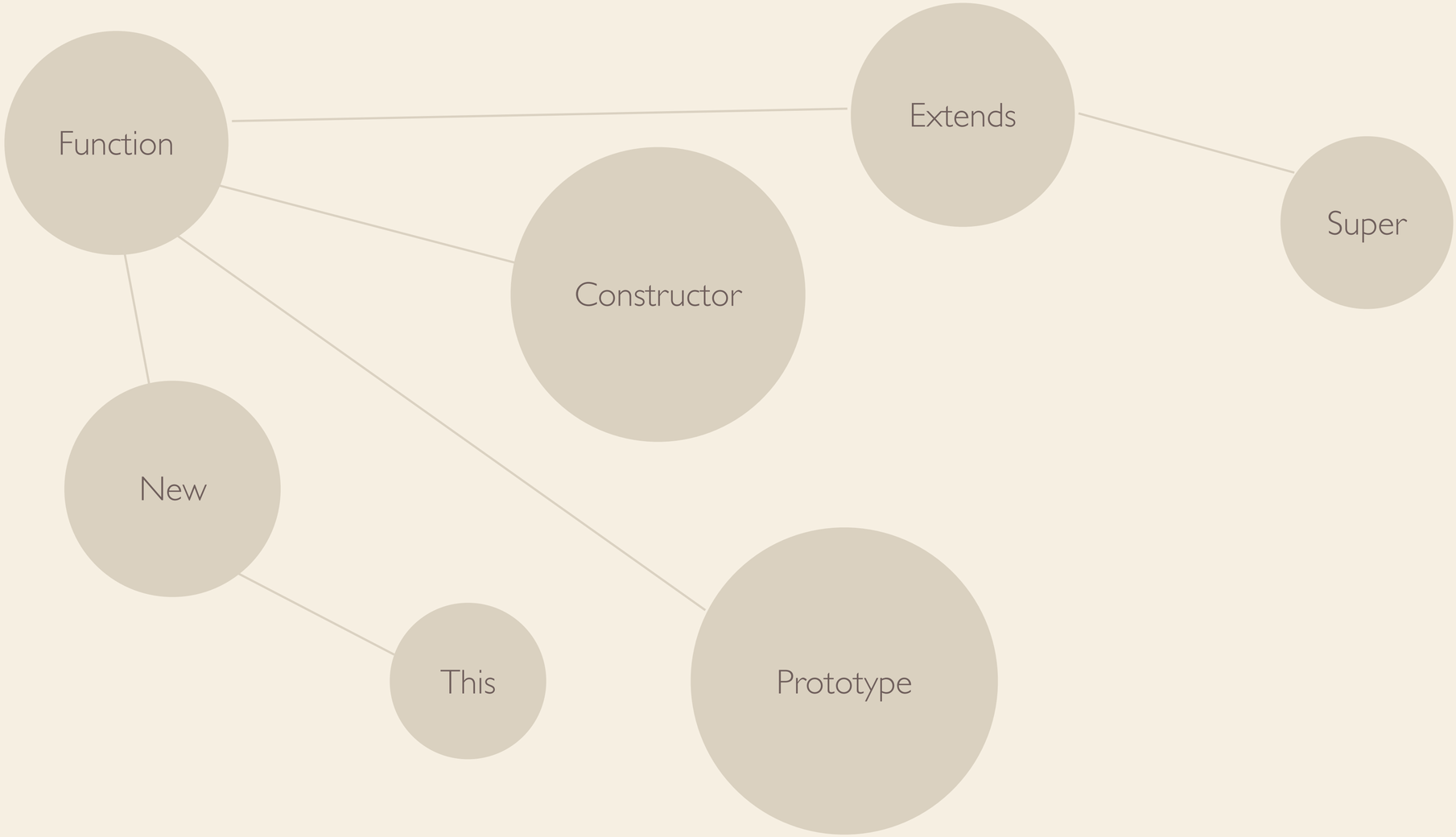
编译时加载

COMMONJS 模块 VS ES6 模块

COMMONJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用

COMMONJS 模块是运行时加载，ES6 模块时编译时输出接口

CLASS 是什么



ES5 VS ES6 CLASS

```
function Parent (name) {
  this.name = name
}

Parent prototype.getName = function () {
  return this.name
}


var person = new Parent ('Jack')
```

```
class Parent {
  constructor (name) {
      this.name = name
   }
   getName () {
     return this.name
   }
}

const person = new Parent ('Jack')
```

ES5 VS ES6 继承

```
function Son (name, age) {
 Parent.apply(this, arguments)
   this.age =age
}


Son.prototype = new  Parent()


var person = new Son ('Jack')
```

```
class Son extends Parent {
  constructor (name, age) {
       super(name);
       this.age = age
    }
}


const person = new Parent ('Jack')
```

ES6 PROXIES

TARGET - 表示所要拦截的目标对象

HANDER - 是一个配置对象，用来定制拦截行为

TRAPS - 用来拦截对目标对象属性的访问请求

```
let handler = {
    get: function(target, name) {
        return name in target ? //if the key exists
            target[name] : //return the key
            'key does not exist'; //else, return custom message
    }
};

let o = {
    reason: 'reason',
    code: 'code'
}

let p = new Proxy(o, handler);

console.log(p.o.reason) //result - 'reason'
console.log(p.o.code) //result = 'code'
console.log(p.o.beer) //result - 'key does not exist'
```

ES6 PROXIES 的用途

1. 抽离校验模块，保障数据的类型的准确
2. 实现真实的私有变量了
3. 数据的转化，或者过滤

……

那么多废话干嘛

回去好好看文档！！！