ES6~ES12（一）

王红元 coderwhy

阅读源码

- 阅读源码大家遇到的最大的问题：
- 1.一定不要浮躁
- 2.看到后面忘记前面的东西
  - Bookmarks的打标签的工具：command(ctrl) + alt + k
  - 读一个函数的时候忘记传进来是什么？
- 3.读完一个函数还是不知道它要干嘛
- 4.debugger

继承内置类

- 我们也可以让我们的类继承自内置类，比如Array：

```
class HYArray extends Array {
  lastItem() {
    return this[this.length-1]
  }
}

var array = new HYArray(10, 20, 30)
console.log(array.lastItem())
```

类的混入mixin

- JavaScript的类只支持单继承：也就是只能有一个父类
  - 那么在开发中我们我们需要在一个类中添加更多相似的功能时，应该如何来做呢？
  - 这个时候我们可以使用混入（mixin）；

```
function mixinRunner(BaseClass) {
  return class extends BaseClass {
    running() {
      console.log("running~")
    }
  }
}

function mixinEater(BaseClass) {
  return class extends BaseClass {
    eating() {
      console.log("eating~")
    }
  }
}
```

```
class Person {

}

class NewPerson extends mixinEater(mixinRunner(Person)) {

}

var np = new NewPerson()
np.eating()
np.running()
```

在react中的高阶组件

Users > coderwhy > Desktop > React > 课堂 > code > 10_react-redux > src > utils > JS connect.js > ...

```
import React, { PureComponent } from "react";
import { StoreContext } from './context';

export function connect(mapStateToProps, mapDispachToProp) {
  return function enhanceHOC(WrappedComponent) {
    class EnhanceComponent extends PureComponent {
      constructor(props, context) {
        super(props, context);
        this.state = {
          storeState: mapStateToProps(context.getState())
        }
      }
      componentDidMount() {
        this.unsubscribe = this.context.subscribe(() => {
          this.setState({
            storeState: mapStateToProps(this.context.getState())
          })
        })
      }
      componentWillUnmount() {
        this.unsubscribe();
      }
      render() {
        return <WrappedComponent {...this.props}
          {...mapStateToProps(this.context.getState())}
          {...mapDispachToProp(this.context.dispatch)} />
      }
    }
```

JavaScript中的多态

- 面向对象的三大特性：封装、继承、多态。
  - 前面两个我们都已经详细解析过了，接下来我们讨论一下JavaScript的多态。
- JavaScript有多态吗？
  - 维基百科对多态的定义：**多态**（英语：polymorphism）指为不同数据类型的实体提供统一的接口，或使用一个单一的符号来表示多个不同的类型。
  - 非常的抽象，个人的总结：不同的数据类型进行同一个操作，表现出不同的行为，就是多态的体现。
- 那么从上面的定义来看，JavaScript是一定存在多态的。

```
function sum(a, b) {
  console.log(a + b)
}

sum(10, 20)
sum("abc", "cba")
```

字面量的增强

- ES6中对 **对象字面量** 进行了增强，称之为 Enhanced object literals（增强对象字面量）。
- 字面量的增强主要包括下面几部分：
  - 属性的简写：**Property Shorthand**
  - 方法的简写：**Method Shorthand**
  - 计算属性名：**Computed Property Names**

解构Destructuring

- ES6中新增了一个从数组或对象中方便获取数据的方法，称之为解构Destructuring。
- 我们可以划分为：数组的解构和对象的解构。
- 数组的解构：
  - 基本解构过程
  - 顺序解构
  - 解构出数组
  - 默认值
- 对象的解构：
  - 基本解构过程
  - 任意顺序
  - 重命名
  - 默认值

解构的应用场景

- 解构目前在开发中使用是非常多的：
  - 比如在开发中拿到一个变量时，自动对其进行解构使用；
  - 比如对函数的参数进行解构；

```
async getPageListDataAction({ commit }, payload: IPagePayload) {
  const pageName = payload.pageName
  const pageUrl = `/${pageName}/list`
  if (pageUrl.length === 0) return
  const { totalCount, list } = await getPageList(pageUrl, payload.queryInfo)
```

```
instance.update = effect(function componentEffect() {
  // 组件没有被挂载，那么挂载组件
  if (!instance.isMounted) {
    let vnodeHook: VNodeHook | null | undefined
    const { el, props } = initialVNode
    const { bm, m, parent } = instance
```

let/const基本使用

- 在ES5中我们声明变量都是使用的var关键字，从ES6开始新增了两个关键字可以声明变量：let、const
  - let、const在其他编程语言中都是有的，所以也并不是新鲜的关键字；
  - 但是let、const确确实实给JavaScript带来一些不一样的东西；
- let关键字：
  - 从直观的角度来说，let和var是没有太大的区别的，都是用于声明一个变量
- const关键字：
  - const关键字是constant的单词的缩写，表示常量、衡量的意思；
  - 它表示保存的数据一旦被赋值，就不能被修改；
  - 但是如果赋值的是引用类型，那么可以通过引用找到对应的对象，修改对象的内容；

- 注意：另外let、const不允许重复声明变量；

let/const作用域提升

- let、const和var的另一个重要区别是作用域提升：
  - 我们知道var声明的变量是会进行作用域提升的；
  - 但是如果我们使用let声明的变量，在声明之前访问会报错；

```
console.log(foo) // ReferenceError: Cannot access 'foo' before initialization

let foo = "foo"
```

- 那么是不是意味着foo变量只有在代码执行阶段才会创建的呢？
  - 事实上并不是这样的，我们可以看一下ECMA262对let和const的描述；
  - **这些变量会被创建在包含他们的词法环境被实例化时，但是是不可以访问它们的，直到词法绑定被求值；**

> `let` and `const` declarations define variables that are scoped to the running execution context's LexicalEnvironment. The variables are created when their containing Lexical Environment is instantiated but may not be accessed in any way until the variable's *LexicalBinding* is evaluated. A variable defined by a *LexicalBinding* with an *Initializer* is assigned the value of its *Initializer*'s *AssignmentExpression* when the *LexicalBinding* is evaluated, not when the variable is created. If a *LexicalBinding* in a `let` declaration does not have an *Initializer* the variable is assigned the value **undefined** when the *LexicalBinding* is evaluated.

let/const有没有作用域提升呢？

- 从上面我们可以看出，在执行上下文的词法环境创建出来的时候，变量事实上已经被创建了，只是这个变量是不能被访问的。
  - 那么变量已经有了，但是不能被访问，是不是一种作用域的提升呢？
- 事实上维基百科并没有对作用域提升有严格的概念解释，那么我们自己从字面量上理解；
  - **作用域提升**：在声明变量的作用域中，如果这个变量可以在声明之前被访问，那么我们可以称之为作用域提升；
  - 在这里，它虽然被创建出来了，但是不能被访问，我认为不能称之为作用域提升；
- 所以我的观点是let、const没有进行作用域提升，但是会在解析阶段被创建出来。

Window对象添加属性

- 我们知道，在全局通过var来声明一个变量，事实上会在window上添加一个属性：
  - 但是let、const是不会给window上添加任何属性的。
- 那么我们可能会想这个变量是保存在哪里呢？
- 我们先回顾一下最新的ECMA标准中对执行上下文的描述

> Every execution context has associated with it a variable object. Variables and functions declared in the source text are added as properties of the variable object. For function code, parameters are added as properties of the variable object.
>
> 每一个执行上下文会被关联到一个变量环境（variable object，VO），在源代码中的变量和函数声明会被作为属性添加到VO中。
>
> 对于函数来说，参数也会被添加到VO中。

> Every execution context has an associated VariableEnvironment. Variables and functions declared in ECMAScript code evaluated in an execution context are added as bindings in that VariableEnvironment's Environment Record. For function code, parameters are also added as bindings to that Environment Record.
>
> 每一个执行上下文会关联到一个变量环境（VariableEnvironment）中，在执行代码中变量和函数的声明会作为环境记录（Environment Record）添加到变量环境中。
>
> 对于函数来说，参数也会被作为环境记录添加到变量环境中。

变量被保存到VariableMap中

- 也就是说我们声明的变量和环境记录是被添加到变量环境中的：
  - 但是标准有没有规定这个对象是window对象或者其他对象呢？
  - 其实并没有，那么JS引擎在解析的时候，其实会有自己的实现；
  - 比如v8中其实是通过VariableMap的一个hashmap来实现它们的存储的。
  - 那么window对象呢？而window对象是早期的GO对象，在最新的实现中其实是浏览器添加的全局对象，并且一直保持了window和var之间值的相等性；

```
// A hash map to support fast variable declaration and lookup.
class VariableMap : public ZoneHashMap {
 public:
  explicit VariableMap(Zone* zone);
  VariableMap(const VariableMap& other, Zone* zone);

  VariableMap(VariableMap&& other) V8_NOEXCEPT : ZoneHashMap(std::move(other)) {
  }
```

var的块级作用域

- 在我们前面的学习中，JavaScript只会形成两个作用域：全局作用域和函数作用域。

- ES5中放到一个代码中定义的变量，外面是可以访问的：

```
// var没有块级作用域
{
  // 编写语句
  var foo = "foo"
}

console.log(foo) // foo 可以访问到
```

let/const的块级作用域

- 在ES6中新增了块级作用域，并且通过let、const、function、class声明的标识符是具备块级作用域的限制的：

```
{
  let foo = "foo"
  function bar() {
    console.log("bar")
  }
  class Person {}
}

console.log(foo) // ReferenceError: foo is not defined
bar() // 可以访问
var p = new Person() // ReferenceError: foo is not defined
```

- 但是我们会发现函数拥有块级作用域，但是外面依然是可以访问的：
  - 这是因为引擎会对函数的声明进行特殊的处理，允许像var那样进行提升；

块级作用域的应用

- 我来看一个实际的案例：获取多个按钮监听点击

```
<button>按钮1</button>
<button>按钮2</button>
<button>按钮3</button>
<button>按钮4</button>
```

- 使用let或者const来实现：

```
var btns = document.getElementsByTagName("button")
for (let i = 0; i < btns.length; i++) {
  btns[i].onclick = function() {
    console.log("第" + i + "个按钮被点击")
  }
}
```

var、let、const的选择

- 那么在开发中，我们到底应该选择使用哪一种方式来定义我们的变量呢？
- 对于var的使用：
  - 我们需要明白一个事实，var所表现出来的特殊性：比如作用域提升、window全局对象、没有块级作用域等都是一些历史遗留问题；
  - 其实是JavaScript在设计之初的一种语言缺陷；
  - 当然目前市场上也在利用这种缺陷出一系列的面试题，来考察大家对JavaScript语言本身以及底层的理解；
  - 但是在实际工作中，我们可以使用最新的规范来编写，也就是不再使用var来定义变量了；
- 对于let、const：
  - 对于let和const来说，是目前开发中推荐使用的；
  - 我们会有限推荐使用const，这样可以保证数据的安全性不会被随意的篡改；
  - 只有当我们明确知道一个变量后续会需要被重新赋值时，这个时候再使用let；
  - 这种在很多其他语言里面也都是一种约定俗成的规范，尽量我们也遵守这种规范；

字符串模板基本使用

- 在ES6之前，如果我们想要将字符串和一些动态的变量（标识符）拼接到一起，是非常麻烦和丑陋的（ugly）。
- ES6允许我们使用字符串模板来嵌入JS的变量或者表达式来进行拼接：
  - 首先，我们会使用 `` 符号来编写字符串，称之为模板字符串；
  - 其次，在模板字符串中，我们可以通过 **${expression}** 来嵌入动态的内容；

```
const name = "why"
const age = 18
const height = 1.88

console.log(`my name is ${name}, age is ${age}, height is ${height}`)
console.log(`我是成年人吗? ${age >= 18 ? '是': '否'}`)

function foo() {
  return "function is foo"
}

console.log(`my function is ${foo()}`)
```

标签模板字符串使用

- 模板字符串还有另外一种用法：标签模板字符串（Tagged Template Literals）。
- 我们一起来看一个普通的JavaScript的函数：

```
function foo(...args) {
  console.log(args)
}

// [ 'Hello World' ]
foo("Hello World")
```

- 如果我们使用标签模板字符串，并且在调用的时候插入其他的变量：
  - 模板字符串被拆分了；
  - 第一个元素是数组，是被模块字符串拆分的字符串组合；
  - 后面的元素是一个个模块字符串传入的内容；

```
const name = "why"
const age = 18
// [ [ 'Hello ', ' World ', '' ], 'why', 18 ]
foo`Hello ${name} World ${age}`
```

React的styled-components库