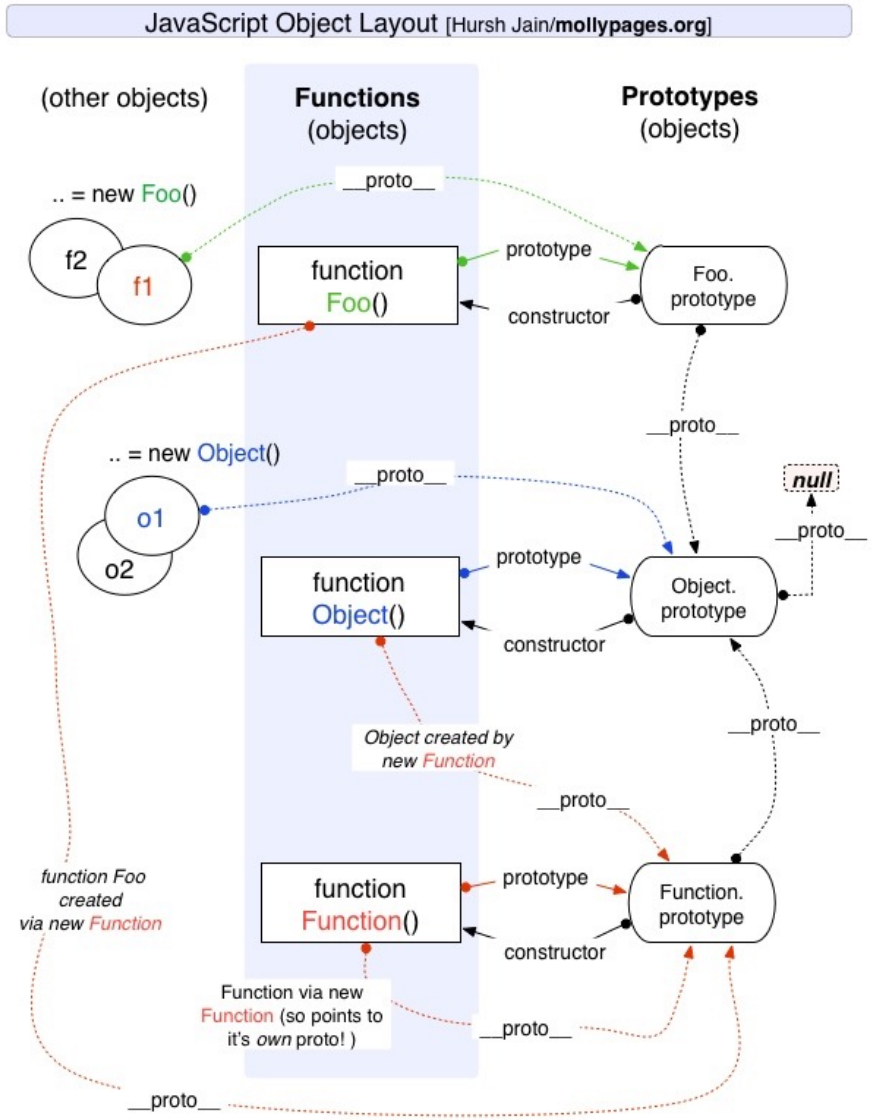
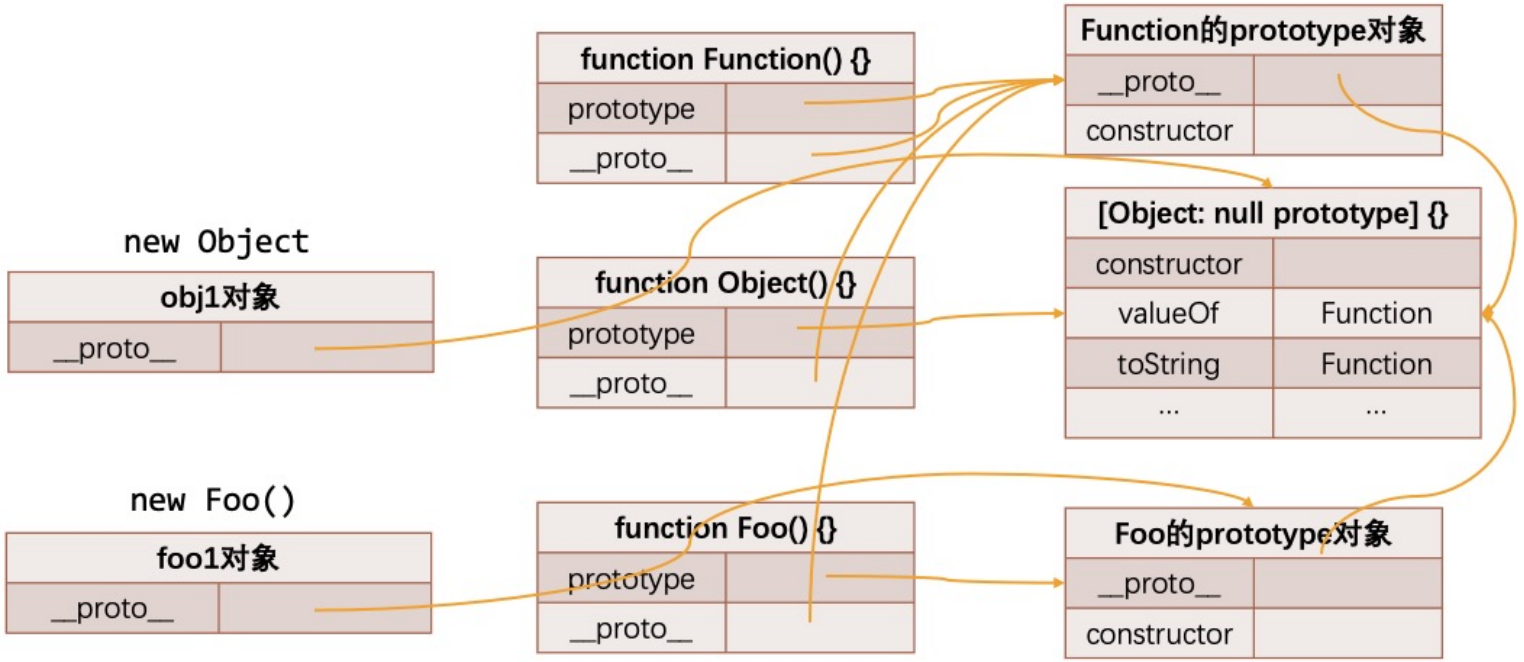
JS面向对象的补充

王红元 coderwhy

原型继承关系


JavaScript Object Layout [Hursh Jain/mollypages.org]
(other objects)
Functions (objects)
Prototypes (objects)
.. = new Foo()
f2
f1
__proto__
function Foo()
prototype
constructor
Foo. prototype
.. = new Object()
o1
o2
function Object()
Object. prototype
null
Object created by new Function
function Function()
Function. prototype
function Foo created via new Function
Function via new Function (so points to it's own proto!)


new Object
obj1对象
__proto__
new Foo()
foo1对象
__proto__
function Function() {}
prototype
__proto__
function Object() {}
prototype
__proto__
function Foo() {}
prototype
__proto__
Function的prototype对象
__proto__
constructor
[Object: null prototype] {}
constructor
valueOf
Function
toString
Function
...
...
Foo的prototype对象
__proto__
constructor

认识class定义类

- 我们会发现，按照前面的构造函数形式创建 **类**，不仅仅和编写普通的函数过于相似，而且代码并不容易理解。
  - 在ES6（ECMAScript2015）新的标准中使用了class关键字来直接定义类；
  - 但是类本质上依然是前面所讲的构造函数、原型链的语法糖而已；
  - 所以学好了前面的构造函数、原型链更有利于我们理解类的概念和继承关系；
- 那么，如何使用class来定义一个类呢？
  - 可以使用两种方式来声明类：类声明和类表达式；

```
class Person {
}

var Student = class {
}
```

类和构造函数的异同

- 我们来研究一下类的一些特性：
  - 你会发现它和我们的构造函数的特性其实是一致的；

```
var p = new Person

console.log(Person) // [class Person]
console.log(Person.prototype) // {}
console.log(Person.prototype.constructor) // [class Person]

console.log(p.__proto__ === Person.prototype) // true

console.log(typeof Person) // function
```

类的构造函数

- 如果我们希望在创建对象的时候给类传递一些参数，这个时候应该如何做呢？
  - 每个类都可以有一个自己的构造函数（方法），这个方法的名称是固定的constructor；
  - 当我们通过new操作符，操作一个类的时候会调用这个类的构造函数constructor；
  - 每个类只能有一个构造函数，如果包含多个构造函数，那么会抛出异常；

- 当我们通过new关键字操作类的时候，会调用这个constructor函数，并且执行如下操作：
  - 1.在内存中创建一个新的对象（空对象）；
  - 2.这个对象内部的[[prototype]]属性会被赋值为该类的prototype属性；
  - 3.构造函数内部的this，会指向创建出来的新对象；
  - 4.执行构造函数的内部代码（函数体代码）；
  - 5.如果构造函数没有返回非空对象，则返回创建出来的新对象；

类的实例方法

- 在上面我们定义的属性都是直接放到了this上，也就意味着它是放到了创建出来的新对象中：
  - 在前面我们说过对于实例的方法，我们是希望放到原型上的，这样可以被多个实例来共享；
  - 这个时候我们可以直接在类中定义；

```
class Person {
  constructor(name, age, height) {
    this.name = name
    this.age = age
    this.height = height
  }

  running() {
    console.log(this.name + " running~")
  }

  eating() {
    console.log(this.name + " eating~")
  }
}
```

类的访问器方法

- 我们之前讲对象的属性描述符时有讲过对象可以添加setter和getter函数的，那么类也是可以的：

```
class Person {
  constructor(name) {
    this._name = name
  }

  set name(newName) {
    console.log("调用了name的setter方法")
    this._name = newName
  }

  get name() {
    console.log("调用了name的getter方法")
    return this._name
  }
}
```

类的静态方法

- 静态方法通常用于定义直接使用类来执行的方法，不需要有类的实例，使用static关键字来定义：

```
class Person {
  constructor(age) {
    this.age = age
  }

  static create() {
    return new Person(Math.floor(Math.random() * 100))
  }
}
```

ES6类的继承 - extends

- 前面我们花了很大的篇幅讨论了在ES5中实现继承的方案，虽然最终实现了相对满意的继承机制，但是过程却依然是非常繁琐的。
  - 在ES6中新增了使用extends关键字，可以方便的帮助我们实现继承：

```
class Person {

}

class Student extends Person {

}
```

super关键字

- 我们会发现在上面的代码中我使用了一个super关键字，这个super关键字有不同的使用方式：
  - 注意：在子（派生）类的构造函数中使用this或者返回默认对象之前，必须先通过super调用父类的构造函数！
  - super的使用位置有三个：子类的构造函数、实例方法、静态方法；

```
// 调用父对象/父类的构造函数
super([arguments]);

// 调用父对象/父类上的方法
super.functionOnParent([arguments]);
```

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
```