JS函数式编程

王红元 coderwhy

实现apply、call、bind

- 接下来我们来实现一下apply、call、bind函数：
  - 注意：我们的实现是练习函数、this、调用关系，不会过度考虑一些边界情况

```
Function.prototype.hyapply = function(thisBings, args) {
  thisBings = thisBings ? Object(thisBings): window
  thisBings.fn = this

  if (!args) {
    thisBings.fn()
  } else {
    var result = thisBings.fn(...args)
  }

  delete thisBings.fn

  return result
}
```

```
Function.prototype.hycall = function(thisBings, ...args) {
  thisBings = thisBings ? Object(thisBings): window
  thisBings.fn = this

  var result = thisBings.fn(...args)
  delete thisBings.fn

  return result
}
```

```
Function.prototype.hybind = function(thisBings, bindArgs) {
  thisBings = thisBings ? Object(thisBings): window
  thisBings.fn = this

  return function(...newArgs) {
    var args = [...bindArgs, ...newArgs]
    return thisBings.fn(...args)
  }
}
```

认识arguments

- **arguments** 是一个 对应于 **传递给函数的参数** 的 **类数组(array-like)对象。**

```
function foo(x, y, z) {
  // [Arguments] { '0': 10, '1': 20, '2': 30 }
  console.log(arguments)
}

foo(10, 20, 30)
```

- array-like意味着它不是一个数组类型，而是一个对象类型：
  - 但是它却拥有数组的一些特性，比如说length，比如可以通过index索引来访问；
  - 但是它却没有数组的一些方法，比如forEach、map等；

```
console.log(arguments.length)
console.log(arguments[0])
console.log(arguments[1])
console.log(arguments[2])
```

arguments转成array

```
// 1.转化方式一:
var length = arguments.length
var arr = []
for (var i = 0; i < length; i++) {
  arr.push(arguments[i])
}
console.log(arr)

// 2.转化方式二
var arr1 = Array.prototype.slice.call(arguments);
var arr2 = [].slice.call(arguments)
console.log(arr1)
console.log(arr2)

// 3.转化方式三: ES6之后
const arr3 = Array.from(arguments)
const arr4 = [...arguments]
console.log(arr3)
console.log(arr4)
```

箭头函数不绑定arguments

- 箭头函数是不绑定arguments的，所以我们在箭头函数中使用arguments会去上层作用域查找：

```
console.log(arguments)

var foo = (x, y, z) => {
  console.log(arguments)
}

foo(10, 20, 30)
```

```
function bar(m, n) {
  return (x, y, z) => {
    console.log(arguments)
  }
}

var fn = bar(20, 30)
fn(10, 20, 30)
```

理解JavaScript纯函数

- **函数式编程**中有一个非常重要的概念叫**纯函数**，JavaScript符合**函数式编程的范式**，所以也**有纯函数的概念**；
  - 在**react开发中纯函数是被多次提及**的；
  - 比如**react中组件就被要求像是一个纯函数**（为什么是像，因为还有class组件），**redux中有一个reducer的概念**，也是要求必须是一个纯函数；
  - 所以**掌握纯函数对于理解很多框架的设计**是非常有帮助的；
- **纯函数的维基百科定义：**
  - 在程序设计中，若一个函数符合以下条件，那么这个函数被称为纯函数：
  - 此函数在相同的输入值时，需产生相同的输出。
  - 函数的输出和输入值以外的其他隐藏信息或状态无关，也和由I/O设备产生的外部输出无关。
  - 该函数不能有语义上可观察的函数副作用，诸如“触发事件”，使输出设备输出，或更改输出值以外物件的内容等。
- **当然上面的定义会过于的晦涩，所以我简单总结一下：**
  - 确定的输入，一定会产生确定的输出；
  - 函数在执行过程中，不能产生副作用；

副作用的理解

- **那么这里又有一个概念，叫做副作用**，什么又是**副作用**呢？
  - **副作用（side effect）**其实本身是医学的一个概念，比如我们经常说吃什么药本来是为了治病，可能会产生一些其他的副作用；
  - 在计算机科学中，也引用了副作用的概念，表示在执行一个函数时，除了返回函数值之外，还对调用函数产生了附加的影响，比如修改了全局变量，修改参数或者改变外部的存储；

- **纯函数在执行的过程中就是不能产生这样的副作用：**
  - 副作用往往是产生bug的 “温床”。

纯函数的案例

- **我们来看一个对数组操作的两个函数：**
  - slice：slice截取数组时不会对原数组进行任何操作,而是生成一个新的数组；
  - splice：splice截取数组, 会返回一个新的数组, 也会对原数组进行修改；
  - slice就是一个纯函数，不会修改传入的参数；

```
var names = ["abc", "cba", "nba", "dna"]

// slice截取数组时不会对原数组进行任何操作,而是生成一个新的数组
var newNames = names.slice(0, 2)
console.log(newNames)

// splice截取数组, 会返回一个新的数组, 也会对原数组进行修改
var newNames2 = names.splice(0, 2)
console.log(newNames2)
console.log(names)
```

我们来自己编写几个案例，来看一下它们是否是纯函数

```
function sum(num1, num2) {
  return num1 + num2;
}
```

```
let foo = 5;

function add(num) {
  return foo + num;
}

console.log(add(5));
foo = 10;
console.log(add(5));
```

```
function printInfo(info) {
  console.log(info.name, info.age);
  info.name = "哈哈哈";
}
```

纯函数的优势

- **为什么纯函数在函数式编程中非常重要呢？**
  - 因为你可以安心的编写和安心的使用；
  - 你在**写的时候**保证了函数的纯度，只是单纯实现自己的业务逻辑即可，不需要关心传入的内容是如何获得的或者依赖其他的外部变量是否已经发生了修改；
  - 你在**用的时候**，你确定你的输入内容不会被任意篡改，并且自己确定的输入，一定会有确定的输出；
- React中就要求我们无论是**函数还是class声明一个组件**，这个组件都必须**像纯函数一样**，**保护它们的props不被修改**：

> React 非常灵活，但它也有一个严格的规则：
>
> **所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。**

JavaScript柯里化

- **柯里化**也是属于**函数式编程**里面一个非常重要的概念。
- **我们先来看一下维基百科的解释：**
  - 在计算机科学中，**柯里化**（英语：Currying），又译为**卡瑞化**或**加里化**；
  - 是把接收多个参数的函数，变成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数，而且返回结果的新函数的技术；
  - 柯里化声称 “如果你固定某些参数，你将得到接受余下参数的一个函数” ；
- **维基百科的结束非常的抽象，我们这里做一个总结：**
  - 只传递给函数一部分参数来调用它，让它返回一个函数去处理剩余的参数；
  - 这个过程就称之为柯里化；

柯里化的结构

- 那么柯里化到底是怎么样的表现呢？

```
// 未柯里化的函数
function add1(x, y, z) {
  return x + y + z
}

console.log(add1(10, 20, 30))

// 柯里化处理的函数
function add2(x) {
  return function(y) {
    return function(z) {
      return x + y + z
    }
  }
}

console.log(add2(10)(20)(30))
```

```
var add3 = x => y => z => {
  return x + y + z
}

console.log(add3(10)(20)(30))
```

让函数的职责单一

- **那么为什么需要有柯里化呢？**
  - 在函数式编程中，我们其实往往希望一个函数处理的问题尽可能的单一，而不是将一大堆的处理过程交给一个函数来处理；
  - 那么我们是否就可以将每次传入的参数在单一的函数中进行处理，处理完后在下一个函数中再使用处理后的结果；
- 比如上面的案例我们进行一个修改：**传入的函数需要分别被进行如下处理**
  - 第一个参数 + 2
  - 第二个参数 * 2
  - 第三个参数 ** 2

```
function add2(x) {
  x = x + 2
  return function(y) {
    y = y * 2
    return function(z) {
      z = z**2
      return x + y + z
    }
  }
}
```

柯里化的复用

- 另外一个使用柯里化的场景是可以帮助我们可以**复用参数逻辑**：
  - makeAdder函数要求我们传入一个num（并且如果我们需要的话，可以在这里对num进行一些修改）；
  - 在之后使用返回的函数时，我们不需要再继续传入num了；

```
function makeAdder(num) {
  return function(count) {
    return num + count
  }
}

var add5 = makeAdder(5)
add5(10)
add5(100)

var add10 = makeAdder(10)
add10(10)
add10(100)
```

打印日志的柯里化

- 这里我们在演示一个案例，需求是打印一些日志：
  - 日志包括时间、类型、信息；
- 普通函数的实现方案如下：

```
function log(date, type, message) {
  console.log(`[${date.getHours()}:${date.getMinutes()}] [${type}] [${message}]`)
}

log(new Date(), "DEBUG", "修复问题")
log(new Date(), "FEATURE", "新功能")
```

```
var log = date => type => message => {
  console.log(`[${date.getHours()}:${date.getMinutes()}] [${type}] [${message}]`)
}

var logNow = log(new Date())
logNow("DEBUG")("轮播图bug")
logNow("DEBUG")("点击无效bug")
logNow("FEATURE")("添加新功能")

var logNowDebug = log(new Date())("DEBUG")
logNowDebug("轮播图bug")
logNowDebug("点击无效bug")
```

自动柯里化函数

- 目前我们有将多个普通的函数，转成柯里化函数：

```
function hyCurrying(fn) {
  function curried(...args) {
    if (args.length >= fn.length) {
      return fn.apply(this, args)
    } else {
      return function(...args2) {
        return curried.apply(this, args.concat(args2))
      }
    }
  }

  return curried
}
```

理解组合函数

- **组合（Compose）函数**是在JavaScript开发过程中一种对**函数的使用技巧、模式**：
  - 比如我们现在需要对某一个数据进行函数的调用，执行两个函数fn1和fn2，这两个函数是依次执行的；
  - 那么如果每次我们都需要进行两个函数的调用，操作上就会显得重复；
  - 那么是否可以将这两个函数组合起来，自动依次调用呢？
  - 这个过程就是对函数的组合，我们称之为 组合函数（Compose Function）；

```
function compose(fn1, fn2) {
  return function(x) {
    return fn2(fn1(x))
  }
}
```

```
function double(num) {
  return num * 2
}

function square(num) {
  return num ** 2
}

var calcFn = compose(double, square)
console.log(calcFn(20))
```

实现组合函数

- 刚才我们实现的compose函数比较简单，我们需要考虑更加复杂的情况：比如传入了更多的函数，在调用compose函数时，传入了更多的参数：

```
function compose(...fns) {
  // 遍历所有的原生如果不是函数,那么直接报错
  var length = fns.length
  for (var i = 0; i < length; i++) {
    var fn = fns[i]
    if (typeof fn !== 'function') {
      throw new TypeError('Expected a function')
    }
  }

  // 取出所有的函数一次调用
  return function(...args) {
    // 先获取到第一次执行的结果
    var index = 0
    var result = length ? fns[index].apply(this, args): args
    while(++index < length) {
      result = fns[index].call(this, result)
    }
    return result
  }
}
```