6.22 直播答疑

https://saekiraku.github.io/vscode-rainbow-fart/#/zh/

推荐阅读：JavaScript The First 20 Years

- https://github.com/doodlewind/js-20-years-cn

函数式编程

闭包

1. 扩展一下闭包相关的知识点
2. 柯里化使用了大量闭包，不明白会不会内存泄漏？ 如果会，那像 `lodash` 这种函数编程库有没有什么优化?

- 发生闭包的两个必要条件
    1. 外部对一个函数 makeFn 内部有引用
    2. 在另一个作用域能够访问到 makeFn 作用域内部的局部成员

> 使用闭包可以突破变量作用域的限制，原来只能从一个作用域访问外部作用域的成员
>
> 有了闭包之后，可以在外部作用域访问一个内部作用域的成员
>
> 可以缓存参数
>
> 根据不同参数生成不同功能的函数

```
function makeFn () {
  let name = 'MDN'
  return function inner () {
    console.log(name)
  }
}

let fn = makeFn()
```

- 缓存参数

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

函数式编程

> 函数式编程是一种编程范式，和面向对象编程是并列关系（编程范式：思维方式 + 实现方法）
>
> - 面向对象编程：对现实世界中的事物的抽象，抽象出对象以及对象和对象之间的关系
> - 函数式编程：把现实世界的事物和事物之间的**联系**抽象到程序世界（对运算过程进行抽象）
>
> 重点掌握：
>
> - 函数式编程的核心思想
> - 纯函数
> - 柯里化
> - 函数组合
> - 函子暂时可以作为了解 Array.of() .map()

1. 函数式编程 +

   箭头函数的形式会让代码难以阅读，这种形式真的适合吗？

```
const r = _(employees)
  .filter(e => e.age >= 30)
  .map(e => e.salary)
  .sum()

const r = _(employees)
  .filter(function (e) {
    return e.age >= 30
  })
  .map(function (e) => {
    return e.salary
  })
  .sum()
```

2. 纯函数的定义中： 没有任何可观察的副作用这个定义有不清晰的地方。 比如 reverse(array) 它将输入的数组进行了反序，改变了原来的数组，而不是创建一个反序的数组， 这是不是也是一个副作用呢？

> https://zh.wikipedia.org/wiki/%E7%BA%AF%E5%87%BD%E6%95%B0
>
> 在程序设计中，若一个函数符合以下要求，则它可能被认为是**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。(如果参数是引用传递，对参数的更改会影响函数以外的数据，因此不是纯函数)

3. 函数式编程在实际开发中哪里用的多，以及在面试当中有什么亮点可以说。

4. 组合函数的时候一定组的是纯函数 但是不纯的部分就用柯里化解决什么意思？

```
const fp = require('lodash/fp')

// const f = fp.flowRight(fp.toUpper, fp.first, fp.reverse)

```

```
const f = fp.compose(fp.toUpper, fp.first, fp.reverse)
console.log(f(['one', 'two', 'three']))

// 柯里化是用来把多元函数降维处理（当然也可以把多远函数转换成一元函数
function myfn(a, b, c) {
  return a + b + c
}
const f = fp.curry(myfn);  // f 只需要传递一个参数就可以执行
const f1 = f(1, 2);   f1(3)
const f2 = fn.curry(myfn);    // f 需要传递两个参数执行
```

函子

1. 希望讲解一下函子在工作中的实际应用场景
2. 函子在实际开发中充当什么作用，请举例。
3. 函数式编程模块讲的函子不是很理解，感觉老师讲的时候，只是过了一下用法，没有讲实际用在什么地方，很懵。
4. 函数式编程讲的都是理论的和简单的示例，想看看实战是怎么用的；
5. 对函数组合、函子还是有点不懂
6. 函子和面向对象很像啊 是不同的叫法吗？ 和面向对象区别有什么？
7. 为啥要使用 函子？ 所有的函子里面的方法都是纯的操作 纯函数吗？

- 函子在开发中的实际使用场景
  - 作用是控制副作用 (IO)、异常处理 (Either)、异步任务 (Task)

```
class Functor {
  static of (value) {
    return new Functor(value)
  }
  constructor (value) {
    this._value = value
  }
  map (f) {
    return new Functor(f(this._value))
  }
  value (f) {
    return f(this._value)
  }
}

let toRMB = money => Functor.of(money)
  .map(m => m.replace('$', ''))
  .map(parseFloat)
  .map(m => m * 7)
  .map(m => m.toFixed(2))
  .value(m => '¥' + m)

let money = '$20.30'
console.log(toRMB(money))
```

- folktale
  - https://folktale.origamitower.com/docs/v2.3.0/migrating/from-data.either/

```
const Maybe = require('folktale/maybe')

let toRMB = x => Maybe.fromNullable(x)
// let toRMB = x => Maybe.of(x)
  .map(x => x.replace('$', ''))
  .map(parseFloat)
  .map(x => x * 7)
  .map(x => x.toFixed(2))
  .map(x => '¥' + x)
  .getOrElse('¥0')

let money = '$20.6'
console.log(toRMB(money))

// ------------------------------------------------
const Maybe = require('folktale/maybe')

// const data = [{ name: 'zs', age: 20 }]
// const user = data.find(u => u.name === 'tom')
// let age = 'No Age'
// if (user && user.age) {
//   age = user.age
// }
// console.log(age)

const data = [{ name: 'zs', age: 20 }]
const age = Maybe.fromNullable(data.find(u => u.name === 'zs'))
  .map(u => u.age)
  .getOrElse('No Age')
console.log(age)
```

8. IO函子，还不是很明白，能不能具体举个不纯操作的例子?

```
// IO 函子
const fp = require('lodash/fp')
class IO {
  static of (value) {
    return new IO(function () {
      return value
    })
  }
  constructor (fn) {
    this._value = fn
  }
  map (fn) {
    return new IO(fp.flowRight(fn, this._value))
  }
}
// 调用
let r = IO.of(process).map(p => p.execPath)
// console.log(r)
console.log(r._value())
```

1. Functor 函子
   1）我们明明在说函数式编程，怎么会又用到 class了呢?
   2) 前面说：容器包含值和值的变形关系，但其实函子并没有啊，这个变形关系，或者说函数，是传给map的啊，并不是函子自带的?
2. Either函子
   1) 这个地方讲的不是很清楚，所以either函子就是由两个函子组成的吗？感觉应该要写在一起才对吧?

- falktale 中提供的 Either

```
// folktale 中提供的 either
const Result = require('folktale/result');

function divide(x, y) {
  if (y === 0) {
    throw new Error('division by zero');
  } else {
    return x / y;
  }
}

Result.Ok(1)
Result.Error(2)

// console.log(Result.try(() => divide(4, 2)))
// console.log(Result.try(() => divide(4, 0)))

const safeDivide = (x, y) => Result.try(_ => divide(x, y))

console.log(safeDivide(5, 5).value)
```

- 过时的警告

```
 Using a property named inspect on an object to specify a custom inspection function for util.inspect() is deprecated. Use util.inspect.custom instead. For backward compatibility with Node.js prior to version 6.4.0, both may be specified.

 不建议在对象上使用名为inspect的属性为util.inspect（）指定自定义检查功能。请改用util.inspect.custom。为了与6.4.0之前的Node.js向后兼容，可以同时指定两者。
```

异步操作

事件循环/宏任务/微任务

1. js事件循环里面，为什么settimeout放在宏任务、promise放在微任务?
2. 关于事件循环的希望能详细讲下
3. 宏任务和微任务讲得不够细，不能了解得很全面，希望能扩展下详细解释下。
4. Promise执行时序
   1) 关于事件循环，微任务和宏任务，我感觉这部分老师讲的不是很好，能不能再讲讲清楚？（非常重要！）

Promise

1. 手写 Promise 中，如果有嵌入别人写的 Promise 会有问题不
2. Promise 咋终止程序?

> https://es6.ruanyifeng.com/#docs/promise
>
> 跟传统的try/catch代码块不同的是，如果没有使用catch()方法指定错误处理的回调函数，Promise 对象抛出的错误不会传递到外层代码，即不会有任何反应。
>
> 下面代码中，someAsyncThing()函数产生的 Promise 对象，内部有语法错误。浏览器运行到这一行，会打印出错误提示ReferenceError: x is not defined，但是不会退出进程、终止脚本执行，2 秒之后还是会输出123。这就是说，Promise 内部的错误不会影响到 Promise 外部的代码，通俗的说法就是“Promise 会吃掉错误”。

```
const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

someAsyncThing().then(function() {
  console.log('everything is great');
});

setTimeout(() => { console.log(123) }, 2000);
// Uncaught (in promise) ReferenceError: x is not defined
// 123
```

- 跳出 then 的链式调用

```
const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    // resolve(x + 2);
    resolve(2)
  });
};

someAsyncThing().then(function() {
  console.log('everything is great');
  // 通过把 notRealPromiseException 设置为 true 可以跳出 then 的链式调用
  return Promise.reject({
    notRealPromiseException: true,
  });
}).then(() => {
  console.log('haha')
}, (e) => {
  console.log(e)
}).finally(() => {
  console.log('finally')
})

setTimeout(() => { console.log(123) }, 2000);
```

3. 对 Promise 的 finally 不太理解

- 要实现的效果，不管 Promise 成功还是失败，都执行 finally 的 回调
- finally 返回一个 Promise 对象，能够继续 .then 获取当前 Promise 对象执行成功的结果

```
function p2 () {
  return new Promise(function (resolve, reject) {
    resolve('hello');
  })
}

p2()
  .finally(() => console.log('finally'))
  .then(value => console.log(value))

```

```
finally (callback) {
  // 当前 promise 对象的 then 方法中知道当前 promise 对象的状态
  // then() 方法返回了一个 promise 对象，可以继续 .then()
  // 在 then 方法的回调中接收当前 promise 对象的执行后的结果
  return this.then((value) => {
    callback()
    return value
  }, (reason) => {
    callback()
    throw reason
  })
}
```

- 在 finally 中继续一个 Promise，继续调用返回的 Promise 的 then 方法，等返回的 Promise 执行完毕之后再执行当前调用 finally 的 Promise 对象的 then 方法

```
function p1 () {
  return new Promise(function (resolve, reject) {
    setTimeout(() => {
      resolve('p2')
    }, 2000)
  })
}
function p2 () {
  return new Promise(function (resolve, reject) {
    resolve('hello');
  })
}

p2()
  .finally(() => {
    console.log('finally')
    return p1()
  })
  .then(value => console.log(value))
```

```
finally (callback) {
  return this.then((value) => {
    return MyPromise.resolve(callback()).then(() => value)
  }, (reason) => {
    return MyPromise.resolve(callback()).then(() => throw reason)
  })
},
static resolve (value) {
  if (value instanceof MyPromise) return value;
  return new MyPromise(resolve => resolve(value));
}
```

4. then方法链式调用识别Promise对象自返回
   1）原生Promise.then也是用setTimeout实现的异步吗？那岂不是变成宏任务了？
   2）为什么then非得是异步的，resolvePromise第一个参数可以传个this，而不是promise，MyPromise构造器里调用执行器的时候只要executor.call(this, this.resolve, this.reject)就好啦？
5. 为什么第二种写法，then接收不到错误呢

```
var p1 = new Promise((resolve, reject) => {
  resolve(1)
})

var p2 = new Promise((resolve, reject) => {
  resolve(2)
})
// eg1
let p3 = p1.then((value) => {
  console.log(value)
  return p3
})

p3.then((r) => {
  console.log(r)
}, (reason) => {
  console.log(111)
  console.log(reason)  //可接收到错误
})
// eg2
let p3 = p1
.then((value) => {
  console.log(value)
  return p3
})
.then(() => { }, (reason) => {
  // 接收不到错误
  console.log(reason)
})

console.log(p3)
```

6. Generator 异步方案（上）
   1） 勘误：调用生成器返回的应该是迭代器

通过生成器函数返回的是一个生成器，它实现了迭代器的接口。

其他

1. 扩展 发布、订阅(观察者模式)，这个是在实现JavaScript异步编程的方法中讲到的
2. 微前端构建
3. 希望老师能发给我个人签名照
4. 老师在讲每个知识点的时候，最好能说明一下，知识点的运用场景。有些知识点听也听不明白，也不知道它干啥用的，学起来就没兴趣，就想直接跳过
5. 介绍下大厂时下流行的编程风格和库
    - JavaScript Standard Style Guide
    - Airbnb JavaScript Style Guide
    - 京东代码规范