JS 函数式编程介绍

函数式编程（Functional Programming）

定义： 是一种与面向对象编程和过程是编程并列的编程范式，它通过将电脑运算视为函数运算，并且 避免使用程序状态 以及 易变对象 。

历史:

- 1930 年开发 `Lambda` 演算，函数式编程有了理论基础。
- 在 20 世纪 50 年代，早期的函数式编程语言 `Lisp` 问世。
- 1977 年，FP 概念被提出。
- 在 20 世纪 90 年代，函数式编程语言 `Haskell` 问世。
- 后来，很多语言开始支持函数式编程特性，如 scala,F#,java,javascript,go 等。

> 参考wiki

函数式编程语言：

- 静态： `Haskell`
- 动态： `Lisp`
- 混合面向对象编程： `JavaScript` , `Java` ..

函数式编程的特点：

- 函数是 一等公民 ： 函数和其他数据类型一样， `函数 = 变量 = 值` ， 可以作为变量参数，也可以作为别的函数的返回值
- 只用表达式，不用语句： 每一步都是单纯的运算，每一步都 `return` 返回值
- 纯函数没有 副作用`Side Effects` ： 函数内部不影响外界数据环境
- 不修改状态：函数返回新的值，不修改变量
- 引用透明：外界环境不影响函数内部逻辑，函数的运行只依赖输入的参数，任何时候只要参数相同，引用函数总是得到相同的返回值

函数式编程的好处：

- 代码简洁，开发快速
- 接近自然语言，易于理解
- 方便代码管理，方便 `unit testing`

- 易于并发编程，函数式编程 没有死锁 ， 可以将计算分布到 多核 cpu 上进行计算，能够大大地提高处理能力

JS FP concepts

一、 基础概念：

函数式一等公民: 函数可以被对待成其他类型一样，当做参数，当做返回值。

```
setTimeout(function () {
  console.log("output.");
}, 1000);
const foo = (v) => v % 2 === 0;
function filter(predicate, arr) {
  return arr.filter(predicate);
}
filter(foo, [1, 2, 3, 4]); // [2, 4], foo作为参数
```

纯函数: 相同的输入，永远会得到相同的输出，并且没有任何可以观察的副作用, 例如数学表达式: y = x^2

常见的副作用包括：

- 更改文件系统
- 往数据库中插入纪录
- 发送一个 http 请求
- 可变数据，例如对象引用
- 打印 log
- 访问系统状态

```
function foo(x) {
  y = x * x; // 副作用，影响到了外部变量
  console.log("x: ", x); // 副作用，影响到标准输出
}
var y;
foo(3);
```

高阶函数(HOF)： 如果一个函数可以接收另外一个函数作为参数，这种函数就可以称作高阶函数

例如: map, filter, reduce

```
var res = [1, 2, 3, 4, 5].map((v) => {
  return v + 2;
});
var res = [1, 2, 3, 4, 5].filter((v) => v % 2 === 0);
var res = [1, 2, 3, 4, 5].reduce((tmp, v) => {
  return (tmp += v);
}, 0);
```

二、函数式编程工具

- 柯里化 curry
- 偏应用 paritial
- 组合与管道 compose, pipe
- 函子 Functor
- Monad

柯里化 与 偏应用

一、柯里化
概念：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下的参数。 f(a,b,c) => f(a)(b)(c)

```
let add = function (x) {
  return function (y) {
    return x + y;
  };
};
let increment = add(1);
let addTen = add(10);
increment(2); // 3
addTen(2); // 12
```

柯里化的简单实现：

```
funtion curry(f) {
  return curried(...args) {
    if (f.length == arguemnts.length) {
      return func.apply(this, args);
    }
    return function(...rest) {
      curried.apply(this, args.concat(rest))
    }
  }
}
```

二、偏应用
固定任意参数，然后接收剩余参数
参数预装载： `f(a,b,c) => f(a,b)(c) or f(a)(b,c)`

```
const prefix = "http://localhost:8080/";
const preFetch = function (perfix) {
  return function (url, ...args) {
    return ajax(perfix + url, ...args);
  };
};
```

compose 函数组合

通过组合来实现代码装配线功能,
`compose(f, compose(g,h)) = compose(compose(f,g), h) = compose(f,g,h)`

组合的简单实现：

```
// 2个函数参数
const compose = function (f, g) {
  return function (x) {
    return f(g(x));
  };
};
const compose = (a, b) => (c) => a(b(c)); // 胖箭头版本
// n个函数参数
const compose = (...fns) => (value) => fns.reverse().reduce((acc, fn) => fn(acc), value
```

pipe 管线化

按照函数参数正向进行执行，跟 compose 函数的执行顺序相反
管线化的简单实现

```
const pipeline = (...fns) => (value) => fns.reduce((acc, fn) => fn(acc), value);
```

函数式编程的写法

一、集合中心编程方式（典型：map，reduce）

- 数组和对象使用相同的 api 进行遍历： 使用 collection 的 map,reduce, filter,reject
  参考 underscore map 函数collection-map
- 对象的处理函数： pick, omit， map

二、使用柯里化函数，结合 compose 进行代码编写
示例代码： compose

函数式编程例子：

- `src/compose.exercise.js`
- `src/flicker/index.html`

函子

函子: 函子是一种容器，它不仅可以用于在同一个容器之中值得转化，还可以用于将一个容器转化为另外一个容器

```
const Container = function (x) {
  this._value = x;
};
Container.of = function (x) {
  return new Container(x);
};
Container.prototype.map = function (f) {
  return Container.of(f(this._value));
};
```

函子特点：

1. 函子遵循一些特定规则的容器类型或者数据编程协议
2. 具有通用的 of 方法，用来生成新的容器
3. 具有一个通用的 map 方法，返回新实例，这个实例和之前的实例具有相同的规则
4. 具有结合外部的运算能力

两种典型函子案例：
处理空值的 `Maybe` 函子

```
// Maybe函子
const Maybe = function (x) {
  this.__value = x;
};
Maybe.of = function (x) {
  return new Maybe(x);
};
Maybe.prototype.isNothing = function () {
  return this.__value === null || this.__value === undefined;
};
Maybe.prototype.map = function (f) {
  return this.isNothing() ? Maybe.of(null) : Maybe.of(f(this.__value));
};

Maybe.of("Malkovich Malkovich").map(match(/a/gi));
//=> Maybe(['a', 'a'])
Maybe.of(null).map(match(/a/gi));
//=> Maybe(null)
Maybe.of({ name: "Boris" }).map(_.prop("age")).map(add(10));
//=> Maybe(null)
Maybe.of({ name: "Dinah", age: 14 }).map(_.prop("age")).map(add(10)); //=> Maybe(24)
```

错误处理的 `Either` 函子

```
const Left = function (x) {
  this.__value = x;
};
Left.of = function (x) {
  return new Left(x);
};
Left.prototype.map = function (f) {
  return this;
};
const Right = function (x) {
  this.__value = x;
};
Right.of = function (x) {
  return new Right(x);
};
Right.prototype.map = function (f) {
  return Right.of(f(this.__value));
};

const moment = require("moment");
//  getAge :: Date -> User -> Either(String, Number)
const getAge = curry(function (now, user) {
  let birthdate = moment(user.birthdate, "YYYY-MM-DD");
  if (!birthdate.isValid()) return Left.of("Birth date could not be parsed");
  return Right.of(now.diff(birthdate, "years"));
});
getAge(moment(), { birthdate: "2005-12-12" });
// Right(9)
getAge(moment(), { birthdate: "aaaa" });
// Left("Birth date could not be parsed")
```

异步处理： `Promise` 也是一种函子, 其中 `.then` 对应 `.map` ， `Promise.resolve` 对应 `Functor.of` 操作

```
const _ = require("ramda");
const { split, head, curry } = _;
const fs = require("fs");
//  readFile :: String -> Promise(Error, JSON)
const readFile = function (filename) {
  return new Promise(function (reject, resolve) {
    fs.readFile(filename, "utf-8", function (err, data) {
      err ? reject(err) : resolve(data);
    });
  });
};
readFile("metamorphosis").map(split("\n")).map(head);

// jQuery getJSON example:
//========================
//  getJSON :: String -> {} -> Promise(Error, JSON)
const getJSON = curry(function (url, params) {
  return new Promise(function (reject, resolve) {
    $.getJSON(url, params, resolve).fail(reject);
  });
});
getJSON("/video", { id: 10 }).map(_.prop("title"));

Promise.resolve(3).map(function (three) {
  return three + 1;
});
```

Monad

概念： 是一种设计模式，表示通过函数拆解成相互连接的多个步骤，只需要提供下一步运算所需的函数，整个运算会自动的进行下去，
Monad 函子的一个典型应用就是实现 IO 函子，它可以将带有副作用的函数包装起来，在调用时触发

注：有兴趣可自行学习

知名 JS 函数式编程库

- Underscore js 实用库，提供一整套函数式编程的实用功能
- Lodash 一致化，模块化，高性能的 js 使用工具库
- RxJs 函数式兼响应式编程 js 库
- Ramda 专为函数式风格而设计的函数式编程 js 库，函数本身都是柯里化的