

Javascript 函数式编程初探

主讲人：Zeus

函数式编程不是一种工具，
而是一种可以同时适用于任何环境的思维方式。

做一道简单的选择题

三分钟选择

命令式编程

```
const nameList = ["lv-zimou", "ma-yun", "ma-huateng"];
const personList = [];
for (let i = 0, len = nameList.length; i < len; i++) {
  let name = nameList[i];
  let names = name.split("-");
  let newName = [];
  for (let j = 0, naemLen = names.length; j < naemLen; j++) {
    let nameItem = names[j][0].toUpperCase() + names[j].slice(1).toLowerCase();
    newName.push(nameItem);
  }
  personList.push({ name: newName.join(" ") });
}
return personList;
```

函数式编程

```
const nameList = ["lv-zimou", "ma-yun", "ma-huateng"];
const capitalize = (x) => x[0].toUpperCase() + x.slice(1).toLowerCase();
const genObj = curry((key, x) => {
  let obj = {};
  obj[key] = x;
  return obj;
});

const capitalizeName = compose(join(" "), map(capitalize), split("-"));
const convert2Obj = compose(genObj("name"), capitalizeName);
const convertName = map(convert2Obj);

personList = convertName(nameList);
```

```
const nameList = ["lv-zimou", "ma-yun", "ma-huateng"];
// 转换为
const personList = [
  { name: "Lv Zimou" },
  { name: "Ma Yun" },
  { name: "Ma Huateng" },
];
```

(需求)

命令式编程

```
let failTotal = 0;
let sum = 0;
let average = 0;
for (let index = 0, len = studentList.length; index < len; index++) {
  let { score } = studentList[index];
  sum = sum + score;

  if (score < 60) {
    failTotal = failTotal + 1;
  }
}
average = sum / studentList.length;
```

函数式编程

```
let failTotal = studentList
  .map((student) => student.score)
  .filter((score) => score < 60).length;
let average =
  studentList
    .map((student) => student.score)
    .reduce((sum, score) => (sum = sum + score), 0) / studentList.length;
```

```
// 找出数组中分数不合格的学生个数及其平均分
const studentList = [];
for (let index = 0; index < 100; index++) {
  studentList.push({
    score: parseInt(Math.random() * 100),
  });
}
```

(需求)

函数

组合

引用

抽象

模块

HOC

Either

容器

PointFree

声明

透明

HindleyMilner

Just

Monad

副作用

高阶

一等

Nothing

Maybe

公民

柯里化

函数式编程中的函数

编程语言中的函数

函数是 Javascript 过程 -- 一组执行任务或计算值的语句。

```
1 function foo(a, b, c, d) {  
2     console.log(a);  
3     console.log(b);  
4     console.log(c);  
5     console.log(d);  
6 }
```

函数式编程拥抱数学中「函数」的概念

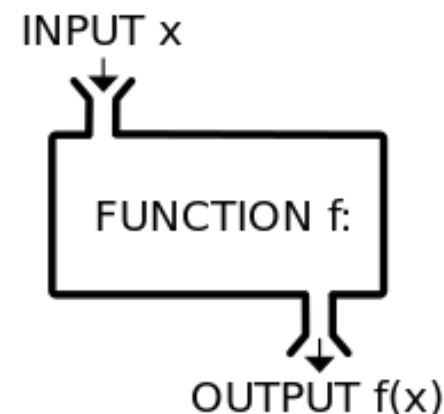


数学中的函数

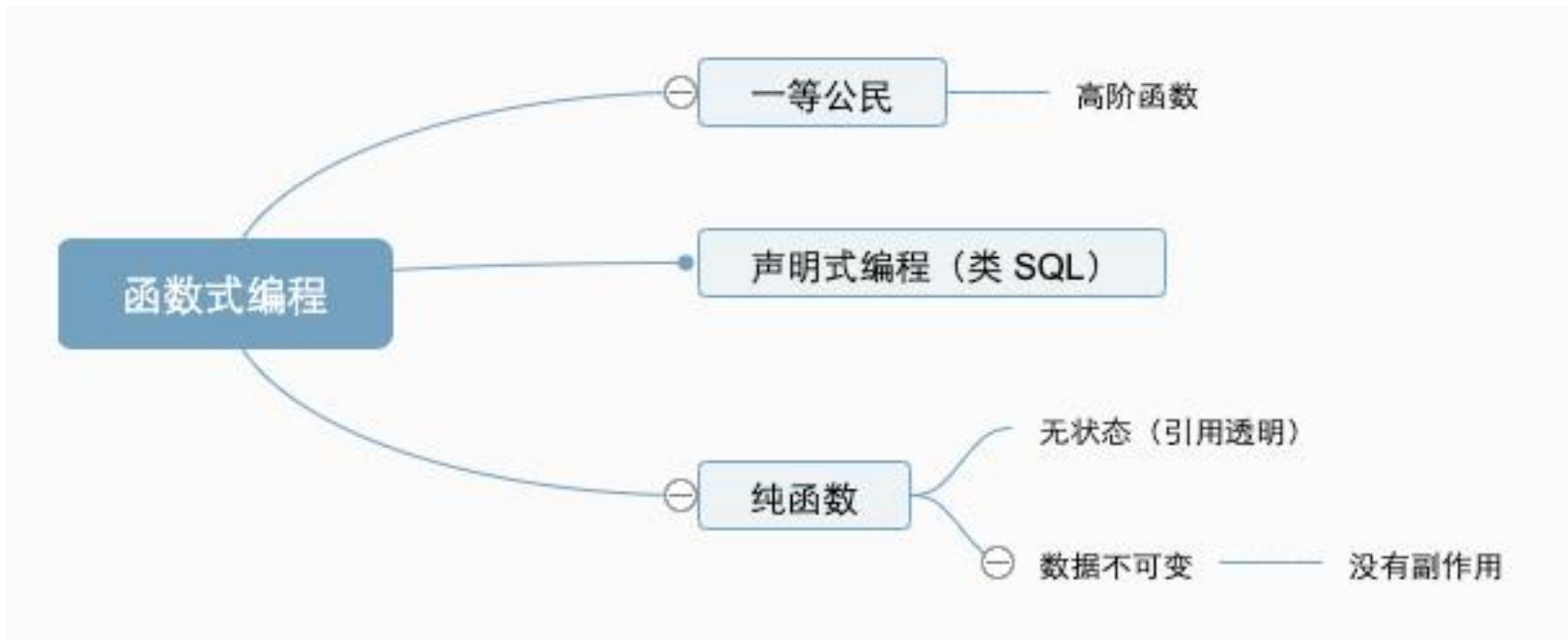
函数在数学中为两不为空集的集合间的一种对应关系：输入值集合中的每项元素皆能对应唯一一项输出值集合中的元素。

$$f(x) = x^2$$

映射关系



函数式编程的特点



一等公民

Javascript 语言中，函数是一等公民，这是函数式编程得以实现的前提。

- 可以使用变量命名
- 可以提供给函数作为参数
- 可以由函数作为结果返回
- 可以包含在数据结构中

简单的说：在编程语言中，一等公民可以作为函数参数，可以作为函数返回值，也可以赋值给变量。

```
const log = () => {  
  console.log("event loop!");  
};  
  
Vue.nextTick(log);
```

```
function tapd(days) {  
  return () => {  
    console.log(--days);  
  };  
}  
  
const work = tapd(5);  
work();
```

高阶函数

声明式编程

这种范式会描述一系列的操作，并不会暴露它们是如何实现的或是数据流如何穿过它们。

关注我需要做什么，而不是怎么去做。

指令式编程

关注告诉计算机如何一步步的做某件事情

类 SQL

```
.from(persons)
  .where((p) => p.birthYear > 1990 && p.birthYear < 1998)
  .groupBy(["firstname", "birthYear"])
  .select("firstname", "birthYear")
  .value();
```

React

```
view = fn(data)
```

纯函数

对于相同的输入，永远会得到相同的输出，而且没有任何可观察的副作用，也不依赖外部环境的状态。

- 仅取决于提供的输入，而不依赖于任何在函数求值期间或调用间隔可能变化的隐藏状态和外部状态。（无状态）
- 不会造成超出其作用域的变化，例如修改全局对象或引用传递的参数。（没有副作用）

- 可缓存性
- 可移植性
- 可测试性
- 合理性
- 自文档化（类型签名） 传递引用一时爽，代码重构火葬场

```
const list = [];  
// 直接修改  
list.map((item) => {  
  item.account++;  
});  
// 输出新变量  
const newList = list.map((item) => ({ ...item, account: item.account + 1 }));
```

尽量避免使用 this！

```
// 纯函数?  
const c = 10;  
function add(a, b) {  
  return a + b;  
}  
function addMore(a, b) {  
  return add(a, b) + c;  
}  
  
addMore(20, 30);
```

```
function addMore(c) {  
  return function add(a, b) {  
    return a + b + c;  
  };  
}  
  
addMore(30)(10, 20);
```

纯函数需要到处传递参数类达成目标，而且还被禁止使用状态

副作用

副作用是在计算结果的过程中，系统状态的一种变化，或者与外部世界进行的可观察的交互。

副作用可能包含，但不限于：

- ❑ 更改文件系统
- ❑ 往数据库插入记录
- ❑ 发送一个 http 请求
- ❑ 可变数据
- ❑ 打印/log
- ❑ 获取用户输入
- ❑ DOM 查询
- ❑ 访问系统状态

```
<script>
/* ✓ GOOD */
export default {
  computed: {
    fullName () {
      return `${this.firstName} ${this.lastName}`
    },
    reversedArray () {
      return this.array.slice(0).reverse() // .slice makes a copy of the array, instead of mu
    }
  }
}
</script>
```

⌚ Now loading...

```
<script>
/* ✗ BAD */
export default {
  computed: {
    fullName () {
      this.firstName = 'lorem' // <- side effect
      return `${this.firstName} ${this.lastName}`
    },
    reversedArray () {
      return this.array.reverse() // <- side effect - original array is being mutated
    }
  }
}
</script>
```

⌚ Now loading...

vue/no-side-effects-in-computed-properties

在函数式思维中也要处理状态和副作用，没有状态和副作用的程序也是无意义的。

-- 《前端函数式演进》

编程示例

```
// audioArray 按照 cursor 从小到大排列
// 将 audio 插入到合适的位置
const audioArray = [
  {
    cursor: 10,
    // ...
  },
  {
    cursor: 24,
    // ...
  },
];
const targetAudio = {
  cursor: 20,
  // ...
};
```

需求又来了

```
function insert() {
  let insertIndex = -1;
  const { cursor: targetCursor } = targetAudio;
  for (let index = 0; index < audioArray.length; index++) {
    const { cursor } = audioArray[index];

    if (targetCursor > cursor) insertIndex = index;
    break;
  }
  audioArray.splice(insertIndex + 1, 0, targetAudio);
}
```

(Before)

```
function insert() {
  const insertIndex = findInsertIndex(
    audioArray.map((item) => item.cursor),
    targetAudio.cursor
  );
  audioArray.splice(insertIndex, 0, targetAudio);
}

function findInsertIndex(array, target) {
  const compare = (a, b) => a > b;

  return (
    array.reduce((result, item, index, arr) => {
      if (compare(target, item)) return index;

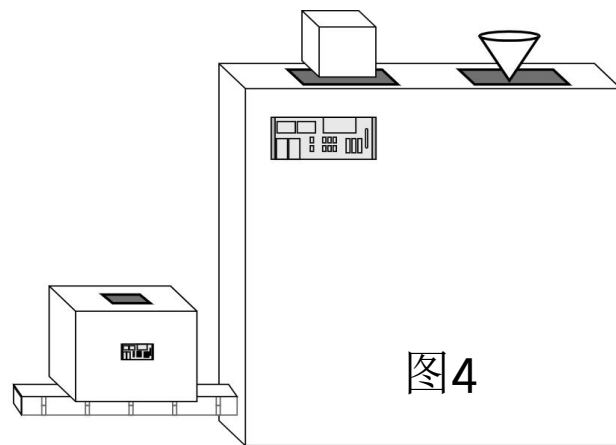
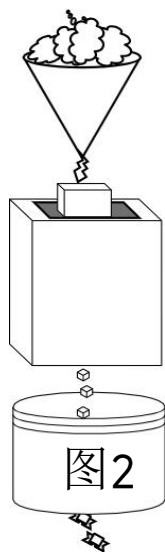
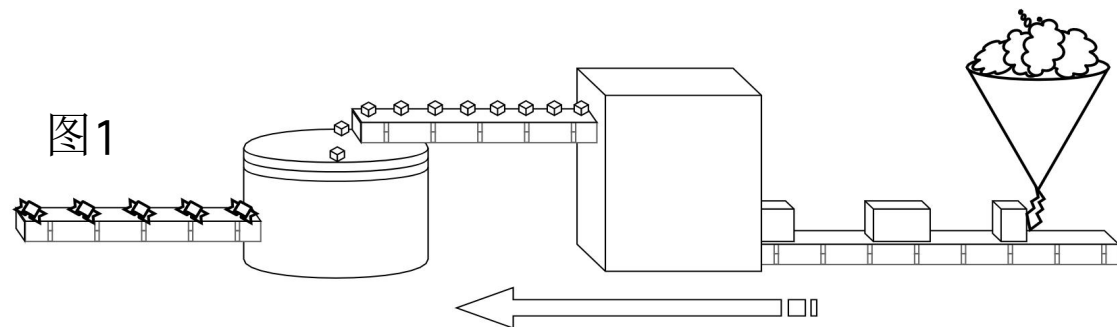
      arr.splice(1); // reduce 操作的 break
      return result;
    }, -1) + 1
  );
}
```

(After)

```
array.findIndex((item) => item > target);
```

(Final)

函数式核心思想（流水线）



参考

抽象

&

组合

柯里化 & 部分函数应用（加工站）

柯里化（**currying**）是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术。

$$f(a, b, c) = f(a)(b)(c)$$

```
const replace = curry((a, b, str) => str.replace(a, b));
const replaceSpaceWith = replace(/\s*/);
const replaceSpaceWithComma = replaceSpaceWith(',');
const replaceSpaceWithDash = replaceSpaceWith('-');
// 正则表达式校验同理
```

部分函数应用（**partially applied function**）是一种将使用多个参数的一个函数转换成固定其中的几个参数值后的一个新函数。

$$f(a, b, c) = f(a,b)(c) \mid f(a)(b, c)$$

```
// 一个通用的请求 API
const request = (type, url, options) => {};
request("GET", "https://learn.wezhuiyi.com");
request("POST", "https://learn.wezhuiyi.com", {});

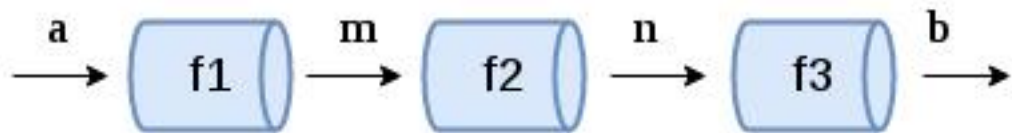
const get = partial(request, ['GET']);
get("https://learn.wezhuiyi.com", {});
```

单值函数是函数组合的基础

函数组合 & 函数管道（流水线组合子）

`docker ps | grep web`

函数的拆分与合成



函数组合 (**compose**)：将多个函数合并成一个函数，从右到左执行

函数管道 (**pipe**)：将多个函数合并成一个函数，从左到右执行

```
fn = f3(f2(f1(a)));
```

命令式

```
fn = _.chain(a).f1().f2().f3();
```

方法链

```
fn = compose(f3, f2, f1);
```

函数组合

```
fn = pipe(f1, f2, f3);
```

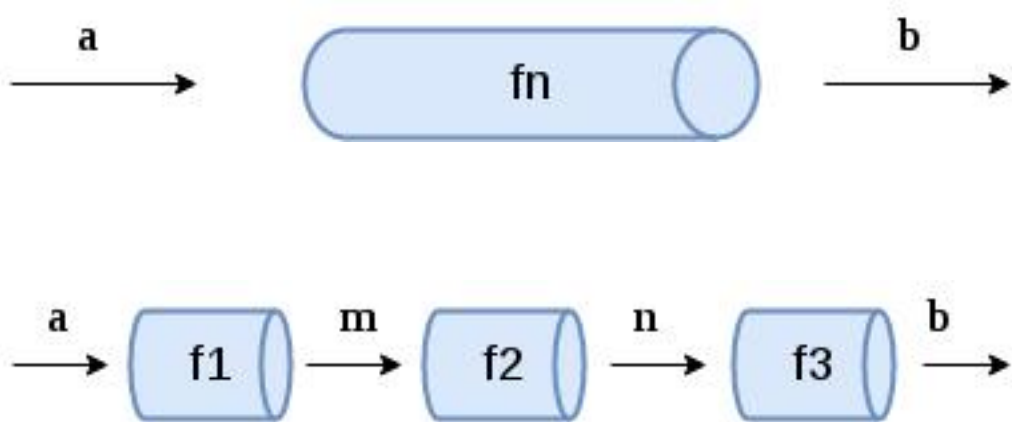
函数管道

```
trace = curry((tip, x) => { console.log(tip, x); return x; });  
compose(f3, f2, trace('after f1'), f1);
```

函数组合的 debug

Pointfree(“无值”风格)

函数的拆分与合成



```
fn = R.pipe(f1, f2, f3);
```

Pointfree 就是运算过程抽象化

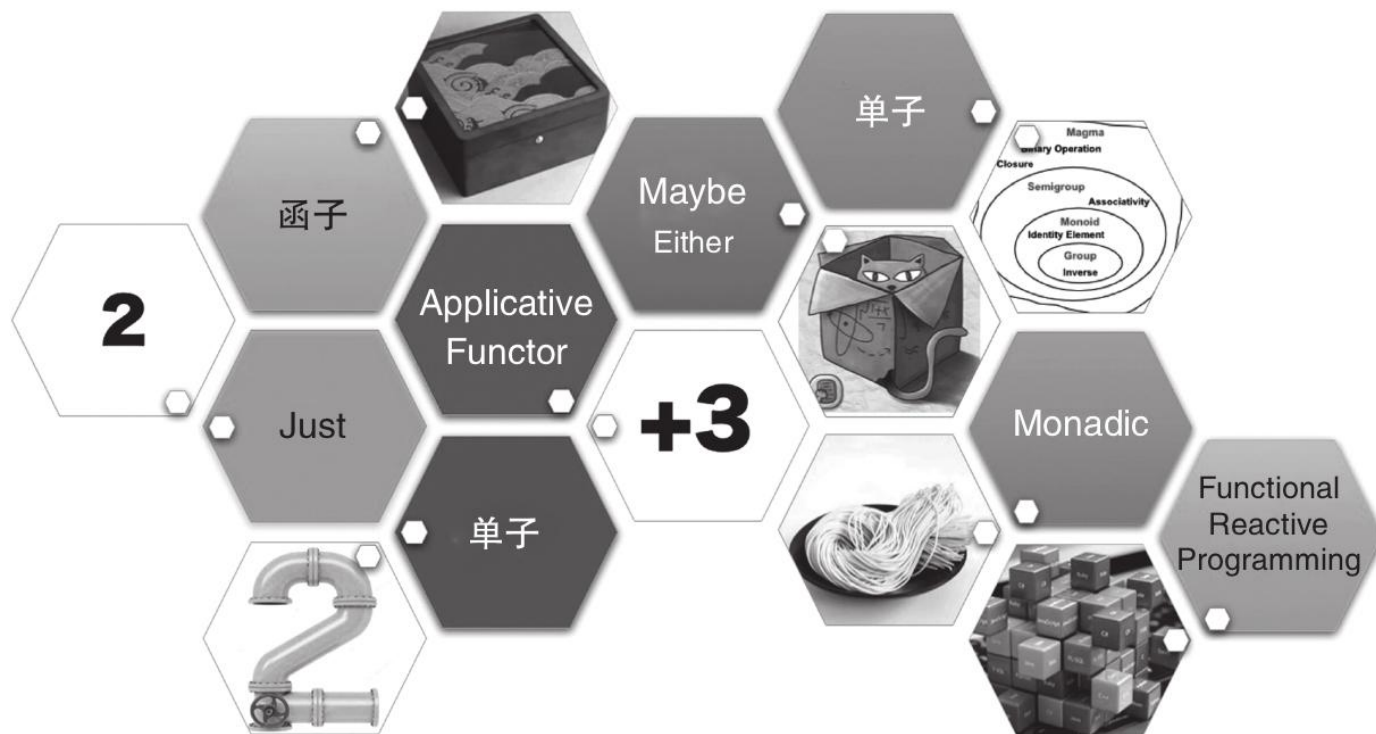
我们完全可以把数据处理的过程，定义成一种与参数无关的合成运算。不需要用到代表数据的那个参数，只要把一些简单的运算步骤合成在一起即可。

这就叫做 **Pointfree**：不使用所要处理的值，只合成运算过程。

本质

Pointfree 的本质就是使用一些通用的函数，组合出各种复杂运算。上层运算不要直接操作数据，而是通过底层函数去处理。这就要求，将一些常用的操作封装成函数。

容器 & 函子



Monad: Just, Nothing, Maybe, Either(Promise), ...

函数式编程的影响

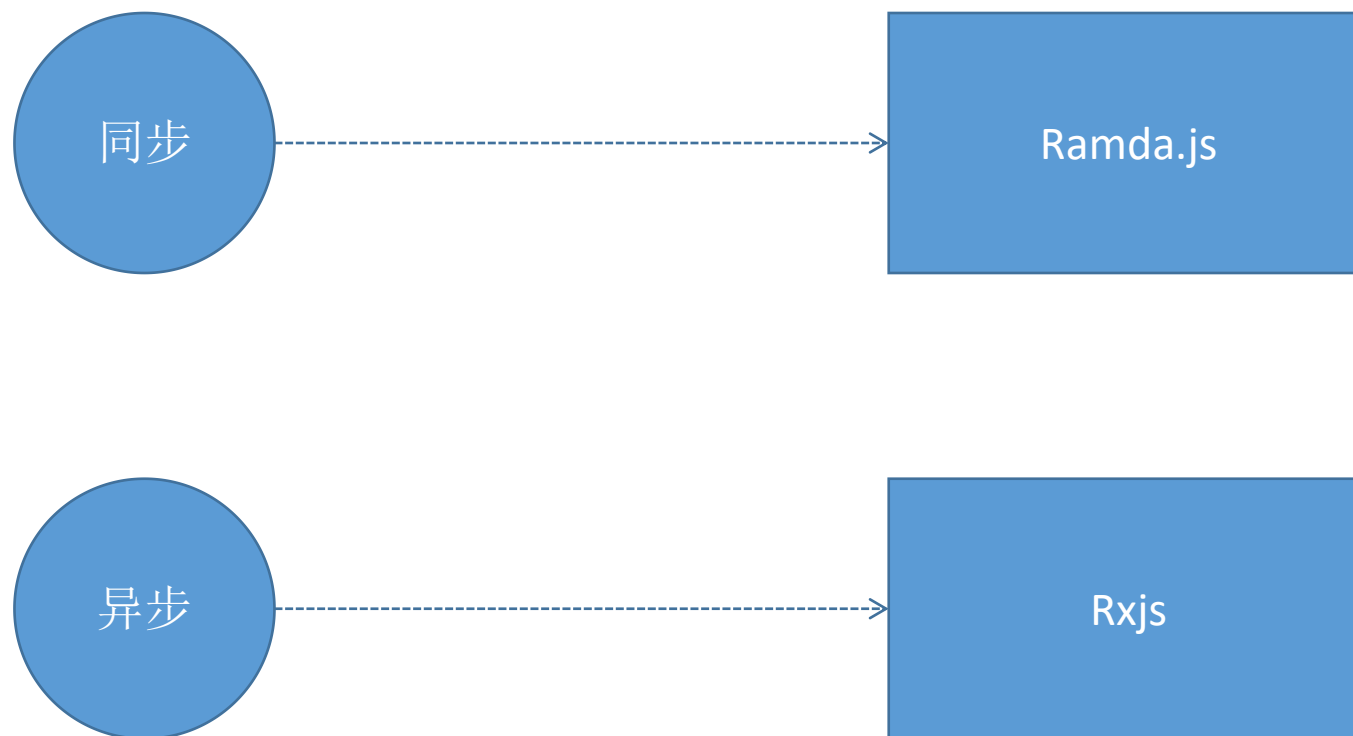
ES6: const, map, filter, reduce, flat, flatMap, promise, lambda...

Vue: composition-api...

React: redux, hoc, hooks...

Lodash.js, Ramda.js, Rxjs（响应式） ...

函数式库



总结对比

GOOD

- **代码简洁，开发快速：**大量使用函数的组合，函数的复用率很高，减少了代码的重复，因此程序比较短，开发速度较快。
- **接近自然语言，易于理解：**函数式编程大量使用声明式代码，基本都是接近自然语言的，因此特别易于理解。
- **易于"并发编程"：**函数式编程没有副作用，不需要考虑“死锁”
- **更少的出错概率：**纯函数没有副作用，易测试。

NOT GOOD

- **性能：**性能绝对是一个短板，因为它往往会对一个方法进行过度包装，从而产生上下文切换的性能开销。
(for vs forEach vs map)
- **资源占用：**在 JS 中为了实现对象状态的不可变，往往会创建新的对象，因此，它对垃圾回收所产生的压力远远超过其他编程方式。这在某些场合会产生十分严重的问题。

没有最好的，只有最合适的

javascript 是函数式编程语言吗？


初学javascript，感觉javascript不像是面向对象语言，在语言分类中应该属于哪一类？函数式编程语言吗

[关注问题](#)[写回答](#)[邀请回答](#)[好问题](#)[1 条评论](#)[分享](#)[...](#)

10 个回答

默认排序 



尤雨溪 

前端开发等 3 个话题下的优秀答主

26 人赞同了该回答

js 属于没有明确归类的语言，或者好听点叫“多范式语言”。

你可以用原型继承模拟一套面向对象的体系，也可以强迫自己写函数式的 javascript（因为函数在 js 里是一等公民），也可以怎么舒服怎么写。It's up to you...

发布于 2014-05-15

▲ 赞同 26



● 收起评论

➤ 分享

★ 收藏

♥ 喜欢



Thank you