

「前端进阶」彻底弄懂函数柯里化



云中桥 Lv5

2019年07月08日 01:22 · 阅读 26666

+ 关注

 601

 31

 收藏

h, hello,
arie Curie!



Would you like
to have some
curry?



Oh, yes,
Haskell Curry!



@稀土掘金技术社区

你知道的越多，你不知道的越多



前言

随着主流JavaScript中函数式编程的迅速发展，函数柯里化在许多应用程序中已经变得很普遍。了解它们是什么，它们如何工作以及如何充分利用它们非常重要。

什么是柯里化（curry）

在数学和计算机科学中，柯里化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。

举例来说，一个接收3个参数的普通函数，在进行柯里化后，柯里化版本的函数接收一个参数并返回接收下一个参数的函数，该函数返回一个接收第三个参数的函数。最后一个函数在接收第三个参数后，将之前接收到的三个参数应用于原普通函数中，并返回最终结果。

// 数学和计算机科学中的柯里化：

// 一个接收三个参数的普通函数

```
function sum(a,b,c) {  
    console.log(a+b+c)  
}
```

// 用于将普通函数转化为柯里化版本的工具函数

```
function curry(fn) {  
    // ... 内部实现省略, 返回一个新函数  
}
```

// 获取一个柯里化后的函数

复制代码

 601

 31

 收藏

```
//返回一个接收第二个参数的函数
let A = _sum(1);
//返回一个接收第三个参数的函数
let B = A(2);
//接收到最后一个参数,将之前所有的参数应用到原函数中,并运行
B(3)    // print : 6
```

而对于Javascript语言来说,我们通常说的柯里化函数的概念,与数学和计算机科学中的柯里化的概念并不完全一样。

在数学和计算机科学中的柯里化函数,一次只能传递一个参数;

而我们Javascript实际应用中的柯里化函数,可以传递一个或多个参数。

来看这个例子:

```
//普通函数
function fn(a,b,c,d,e) {
  console.log(a,b,c,d,e)
}
//生成的柯里化函数
let _fn = curry(fn);

_fn(1,2,3,4,5);    // print: 1,2,3,4,5
_fn(1)(2)(3,4,5);  // print: 1,2,3,4,5
_fn(1,2)(3,4)(5);  // print: 1,2,3,4,5
_fn(1)(2)(3)(4)(5); // print: 1,2,3,4,5
```

复制代码



601



31



收藏

对于已经柯里化后的 `_fn` 函数来说，当接收的参数数量与原函数的形参数量相同时，执行原函数；当接收的参数数量小于原函数的形参数量时，返回一个函数用于接收剩余的参数，直至接收的参数数量与形参数量一致，执行原函数。

当我们知道柯里化是什么了的时候，我们来看看柯里化到底有什么用？

柯里化的用途

柯里化实际是把简答的问题复杂化了，但是复杂化的同时，我们在使用函数时拥有了更加多的自由度。而这里对于函数参数的自由处理，正是柯里化的核心所在。柯里化本质上是降低通用性，提高适用性。来看一个例子：

我们工作中会遇到各种需要通过正则检验的需求，比如校验电话号码、校验邮箱、校验身份证号、校验密码等，这时我们会封装一个通用函数 `checkByRegExp` ,接收两个参数, 校验的正则对象和待校验的字符串

```
function checkByRegExp(regExp,string) {  
    return regExp.test(string);  
}  
  
checkByRegExp(/^1\d{10}$/, '18642838455'); // 校验电话号码  
checkByRegExp(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/ , 'test@163.com'); // 校验邮箱
```

复制代码

上面这段代码，乍一看没什么问题，可以满足我们所有通过正则检验的需求。但是我们考虑这样一个问题，如果我们需要校验多个电话号码或者校验多个邮箱呢？

我们可能会这样做：



```
checkByRegExp(/^1\d{10}$/, '18642838455'); // 校验电话号码
checkByRegExp(/^1\d{10}$/, '13109840560'); // 校验电话号码
checkByRegExp(/^1\d{10}$/, '13204061212'); // 校验电话号码

checkByRegExp(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/, 'test@163.com'); // 校验邮箱
checkByRegExp(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/, 'test@qq.com'); // 校验邮箱
checkByRegExp(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/, 'test@gmail.com'); // 校验邮箱
```

我们每次进行校验的时候都需要输入一串正则，再校验同一类型的数据时，相同的正则我们需要写多次，这就导致我们在使用的时候效率低下，并且由于 `checkByRegExp` 函数本身是一个工具函数并没有任何意义，一段时间后我们重新来看这些代码时，如果没有注释，我们必须通过检查正则的内容，我们才能知道我们校验的是电话号码还是邮箱，还是别的什么。

此时，我们可以借助柯里化对 `checkByRegExp` 函数进行封装，以简化代码书写，提高代码可读性。

```
//进行柯里化
let _check = curry(checkByRegExp);
//生成工具函数,验证电话号码
let checkCellPhone = _check(/^1\d{10}$/);
//生成工具函数,验证邮箱
let checkEmail = _check(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/);

checkCellPhone('18642838455'); // 校验电话号码
checkCellPhone('13109840560'); // 校验电话号码
checkCellPhone('13204061212'); // 校验电话号码

checkEmail('test@163.com'); // 校验邮箱
checkEmail('test@qq.com'); // 校验邮箱
```



601



31



收藏

再看看通过柯里化封装后，我们的代码是不是变得又简洁又直观了呢。

经过柯里化后，我们生成了两个函数 `checkCellPhone` 和 `checkEmail`，`checkCellPhone` 函数只能验证传入的字符串是否是电话号码，`checkEmail` 函数只能验证传入的字符串是否是邮箱，它们与 原函数 `checkByRegExp` 相比，从功能上通用性降低了，但适用性提升了。柯里化的这种用途可以被理解为：**参数复用**

我们再来看一个例子

假定我们有这样一段数据：

```
let list = [
  {
    name: 'lucy'
  },
  {
    name: 'jack'
  }
]
```

[复制代码](#)

我们需要获取数据中的所有 `name` 属性的值，常规思路下，我们会这样实现：

```
let names = list.map(function(item) {
  return item.name;
})
```

[复制代码](#)

 601

 31

 收藏

```
let prop = curry(function(key,obj) {  
    return obj[key];  
})  
let names = list.map(prop('name'))
```

看到这里，可能会有疑问，这么简单的例子，仅仅只是为了获取 name 的属性值，为何还要实现一个 prop 函数呢，这样太麻烦了吧。

我们可以换个思路，prop 函数实现一次后，以后是可以多次使用的，所以我们在考虑代码复杂程度的时候，是可以将 prop 函数的实现去掉的。

我们实际的代码可以理解为只有一行 `let names = list.map(prop('name'))`

这么看来，通过柯里化的方式，我们的代码是不是变得更精简了，并且可读性更高了呢。

如何封装柯里化工具函数

接下来,我们来思考如何实现 curry 函数。

回想之前我们对于柯里化的定义，接收一部分参数，返回一个函数接收剩余参数，接收足够参数后，执行原函数。

我们已经知道了，当柯里化函数接收到足够参数后，就会执行原函数，那么我们如何去确定何时达到足够的参数呢？

我们有两种思路：



601



31



收藏

2. 在调用柯里化工具函数时，手动指定所需的参数个数

我们将这两点结合以下，实现一个简单 curry 函数：

```
/**
 * 将函数柯里化
 * @param fn    待柯里化的原函数
 * @param len    所需的参数个数，默认为原函数的形参个数
 */
function curry(fn, len = fn.length) {
  return _curry.call(this, fn, len)
}

/**
 * 中转函数
 * @param fn    待柯里化的原函数
 * @param len    所需的参数个数
 * @param args  已接收的参数列表
 */
function _curry(fn, len, ...args) {
  return function (...params) {
    let _args = [...args, ...params];
    if(_args.length >= len){
      return fn.apply(this, _args);
    }else{
      return _curry.call(this, fn, len, ..._args)
    }
  }
}
```

复制代码

```
function curry(fn, ...args){
  let allArgs = [...args];
  let result = (...otherArgs) => {
    allArgs = [...allArgs, ...otherArgs];
    if(allArgs.length == fn.length){
      // 这可以把最新参数清除，使得可持续使用
      const result = fn(...allArgs);
      allArgs = [...args];
      return result;
    }else if(allArgs.length > fn.length){
      return "done";
    }
    else{
      return result;
    }
  }
  return result;
}

function myRegExp(reg, target){
  return reg.test(target);
}

const phoneReg = curry(myRegExp, /^1\d{10}$/);
console.log(phoneReg("18829292929"));
console.log(phoneReg("11asdas"));
```

👍 601

💬 31

我们来验证一下：

```
let _fn = curry(function(a,b,c,d,e){
  console.log(a,b,c,d,e)
});

_fn(1,2,3,4,5);      // print: 1,2,3,4,5
_fn(1)(2)(3,4,5);    // print: 1,2,3,4,5
_fn(1,2)(3,4)(5);    // print: 1,2,3,4,5
_fn(1)(2)(3)(4)(5); // print: 1,2,3,4,5
```

复制代码

我们常用的工具库 `lodash` 也提供了 `curry` 方法，并且增加了非常好玩的 `placeholder` 功能，通过占位符的方式来改变传入参数的顺序。

比如说，我们传入一个占位符，本次调用传递的参数略过占位符，占位符所在的位置由下次调用的参数来填充，比如这样：

直接看一下官网的例子：

 601

 31

 收藏

```
var abc = function(a, b, c) {  
  return [a, b, c];  
};
```

```
var curried = _.curry(abc);
```

```
curried(1)(2)(3);  
// => [1, 2, 3]
```

```
curried(1, 2)(3);  
// => [1, 2, 3]
```

```
curried(1, 2, 3);  
// => [1, 2, 3]
```

```
// Curried with placeholders.
```

```
curried(1)(_, 3)(2);  
// => [1, 2, 3]
```

@稀土掘金技术社区

接下来我们来思考, 如何实现占位符的功能。

对于 lodash 的 curry 函数来说, curry 函数挂载在 lodash 对象上, 所以将 lodash 对象当做默认占位符来使用。



601



31



收藏

使用占位符，目的是改变参数传递的顺序，所以在 curry 函数实现中，每次需要记录是否使用了占位符，并且记录占位符所代表的参数位置。

直接上代码：

复制代码

```
/**
 * @param fn          待柯里化的函数
 * @param length      需要的参数个数, 默认为函数的形参个数
 * @param holder      占位符, 默认当前柯里化函数
 * @return {Function} 柯里化后的函数
 */
function curry(fn, length = fn.length, holder = curry) {
  return _curry.call(this, fn, length, holder, [], [])
}

/**
 * 中转函数
 * @param fn          柯里化的原函数
 * @param length      原函数需要的参数个数
 * @param holder      接收的占位符
 * @param args        已接收的参数列表
 * @param holders     已接收的占位符位置列表
 * @return {Function} 继续柯里化的函数 或 最终结果
 */
function _curry(fn, length, holder, args, holders) {
  return function(..._args) {
    //将参数复制一份，避免多次操作同一函数导致参数混乱
    let params = args.slice();
    //将占位符位置列表复制一份，新增加的占位符增加至此
    let _holders = holders.slice();
    //循环入参，追加参数 或 替换占位符
```

 601

 31

 收藏

```

//真实参数 之前存在占位符 将占位符替换为真实参数
if (arg !== holder && holders.length) {
    let index = holders.shift();
    _holders.splice(_holders.indexOf(index),1);
    params[index] = arg;
}
//真实参数 之前不存在占位符 将参数追加到参数列表中
else if(arg !== holder && !holders.length){
    params.push(arg);
}
//传入的是占位符,之前不存在占位符 记录占位符的位置
else if(arg === holder && !holders.length){
    params.push(arg);
    _holders.push(params.length - 1);
}
//传入的是占位符,之前存在占位符 删除原占位符位置
else if(arg === holder && holders.length){
    holders.shift();
}
});
// params 中前 length 条记录中不包含占位符,执行函数
if(params.length >= length && params.slice(0,length).every(i=>i!==holder)){
    return fn.apply(this,params);
}else{
    return _curry.call(this,fn,length,holder,params,_holders)
}
}
}

```



601



31



收藏

```
let fn = function(a, b, c, d, e) {  
  console.log([a, b, c, d, e]);  
}  
  
let _ = {}; // 定义占位符  
let _fn = curry(fn, 5, _); // 将函数柯里化, 指定所需的参数个数, 指定所需的占位符  
  
_fn(1, 2, 3, 4, 5); // print: 1,2,3,4,5  
_fn(_, 2, 3, 4, 5)(1); // print: 1,2,3,4,5  
_fn(1, _, 3, 4, 5)(2); // print: 1,2,3,4,5  
_fn(1, _, 3)(_ , 4, _)(2)(5); // print: 1,2,3,4,5  
_fn(1, _, _, 4)(_ , 3)(2)(5); // print: 1,2,3,4,5  
_fn(_, 2)(_ , _ , 4)(1)(3)(5); // print: 1,2,3,4,5
```

至此, 我们已经完整实现了一个 curry 函数~~

系列文章推荐

- [「前端进阶」单页路由解析与实现](#)
- [「前端进阶」JS中的栈内存堆内存](#)
- [「前端进阶」JS中的内存管理](#)
- [「前端进阶」数组乱序](#)

写在最后



601



31



收藏

- 本文同步首发与[github](#)，可在[github](#)中找到更多精品文章，欢迎 [Watch](#) & [Star](#) ★
- 后续文章参见：[计划](#)

欢迎关注微信公众号 [【前端小黑屋】](#)，每周1-3篇精品优质文章推送，助你走上进阶之旅



分类：[前端](#)

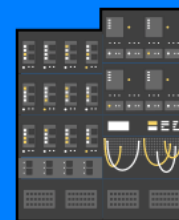
标签：[JavaScript](#)

👍 601

💬 31

★ 收藏

找对——
属于你的
技术圈子



回复进群加入
掘金
微信交流群



评论

输入评论 (Enter换行, Ctrl + Enter发送)

热门评论

 601

 31

 收藏

原来我一直写的这种工具函数叫柯里化？

👍 3 💬 2



CV功城狮

8月前

感觉被你装到了😏

👍 1 💬 回复



me500 Lv1

6月前

一直有在用, 不知道这玩意叫柯里化😂

👍 1 💬 回复

亚里士朱德 Lv2 技术专家 @ 阿里云

2年前

1. 纠正一个错误：checkByRegExp函数中应该是“regExp.test”而不是“regExp.text”。
2. 为什么非要用柯里化这么复杂的方式来封装函数呢？比如下面的方式是不是更简洁？

...

```
let checkCellPhone = checkByRegExp.bind(null, /^1\d{10}$/);  
let checkEmail = str => checkByRegExp(/^(\\w)+(\\.\\w+)*@(\\w+)((\\.\\w+)+)$/ , str);  
...
```

👍 1 💬 2



云中桥 Lv5 (作者)

2年前

谢谢指正，已修改

👍 601

💬 31

☆ 收藏



云中桥 Lv5 (作者)

2年前

柯里化的这种封装函数的方式，只是一种思路，目的是同一个通用性很强，接收多个参数的函数转化为多个适用性强，接收参数单一的函数。

👍 点赞 💬 回复

全部评论 31

🕒 最新

🔥 最热



山顶有人儿 扫地

8小时前

喜欢博主开头的一段话

👍 点赞 💬 回复



Rookie_Lee 前端开发工程师 @ 京东

1天前

稍且歇息，这句话是关键，“柯里化的定义，接收一部分参数，返回一个函数接收剩余参数，接收足够参数后，执行原函数”。

👍 点赞 💬 回复



用户1345462559...

25天前

为啥递归的地方要用 .call 方法改变 this 指向呢？不用 call 就会没有输出

👍 点赞 💬 回复



用户1206361831...

1月前

函数封装 `return fn apply(this, args)` 这里是不是要改成 `args`

👍 601

💬 31

★ 收藏

爬墙

2月前

通俗易懂

👍 点赞 💬 回复

格兰芬多粪弹小... 全栈开发者

2月前

你的公众号好像不再更新了，那你现在一般在哪里输出

👍 点赞 💬 回复

test 😊 **Lv1**

3月前

Lodash 的 curry 方法应该叫偏应用而不是柯里化~

Curried functions take one argument at a time whereas partially applied functions can take many arguments at a time.

[medium.com](#)

👍 点赞 💬 回复

胆大如牛白展堂 **Lv2** 跑堂的|盗圣 @ 关中-七...

4月前

在函数把基础函数当参数时，还能给当参数的基础函数传参数呢？而且不会把基础函数的参数位置给挤掉？

👍 点赞 💬 回复

Sometimes **Lv1** 前端 @ 小厂

4月前

另外一种实现思路：
`function currying(fn) {
 function curried(...args) {`

👍 601

💬 31

☆ 收藏

```
return fn.apply(this, args);  
} else {  
return function (...args2) {...
```

展开

👍 1 💬 1



Sometimes Lv1

4月前

没格式看着好难受~

👍 点赞 💬 回复



四面楚歌001 Lv1

6月前

感觉我们代码中处处都在使用函数的柯里化，但是没有意识到这是柯里化的过程。个人理解柯里化就是通过函数保存了某一个状态，方便下一次的使
用。

👍 1 💬 1



掘金木村拓哉

5月前

柯里化的本质是闭包，闭包能保存状态。但我觉得保存状态不是柯里化的主要作用吧(挠头)

👍 点赞 💬 回复



JoyZ Lv2 前端工程师 @ ebest, 传...

9月前

赞了，看了作者的文章，知道了柯里化的作用，吸引我继续研究下去，但是这个柯里化感觉还是不好理解，先看懂作者的代码，再自己总结吧！

👍 601

💬 31

★ 收藏

 **WZW** Lv2 前端工程师 10月前

函数柯里化是真的难。

👍 2 💬 回复

 **unclechong** web前端开发 @ 阿里云 1年前


高手，这是高手

👍 1 💬 回复

 **白愁离殇** Lv1 为了人类的解放而奋斗 ... 2年前

虽然平时可能用的不多，但是知识永远不嫌多

👍 点赞 💬 1

 **Riki** Lv1 11月前

面试官不嫌多😄😄😄

👍 2 💬 回复

 **Wetoria** Lv2 微信「vWetoria」@ 公... 2年前

中间获取名称那一段在别的地方看到过一模一样的

👍 点赞 💬 回复

👍 601

💬 31

★ 收藏

柯里化，除了表单校验，其他地方，就很难遇到。

👍 点赞 💬 2



RandyHsu

2年前

可不止哦，还有报错信息，提示信息这些不同分类

👍 点赞 💬 回复



喊着666的咸鱼 回复 RandyHsu

2年前

对,能用上的地方还是很多的, 第二个拿对象里面name 的那个例子就很functional ...函数式编程,也有很多用柯里化的地方

“可不止哦，还有报错信息，提示信息这些不同分类”

👍 点赞 💬 回复



broweray18514 Lv1 前端er

2年前

我对lodash的占位符没用过，不看占位符的源码，调用时占位符是怎样的插入规律？？看着有点乱？？

👍 点赞 💬 回复



__zbw 前端

2年前



👍 点赞 💬 回复



CoderAngus 前端工程师 @ 荔枝

2年前

👍 601

💬 31

★ 收藏

👍 点赞 💬 回复

亚里士朱德 Lv2 技术专家 @ 阿里云

2年前

1. 纠正一个错误：checkByRegExp函数中应该是“regExp.test”而不是“regExp.text”。
2. 为什么非要用柯里化这么复杂的方式来封装函数呢？比如下面的方式是不是更简洁？

...

```
let checkCellPhone = checkByRegExp.bind(null, /^1\d{10}$/);  
let checkEmail = str => checkByRegExp(/^(\\w)+(\\.\\w+)*@(\\w+)((\\.\\w+)+)$/, str);
```

...

👍 1 💬 2

云中桥 Lv5 (作者)

2年前

谢谢指正，已修改

👍 点赞 💬 回复

云中桥 Lv5 (作者)

2年前

柯里化的这种封装函数的方式，只是一种思路，目的是同一个通用性很强，接收多个参数的函数转化为多个适用性强，接收参数单一的函数。

👍 点赞 💬 回复

查看全部 31 条回复 ▾

👍 601

💬 31

☆ 收藏

相关推荐

Gaby | 5月前 | JavaScript · 面试

🔥 连八股文都不懂还指望在前端混下去么

👁 18.0w | 👍 4951 | 💬 241

大海我来了 | 1年前 | JavaScript

死磕 36 个 JS 手写题（搞懂后，提升真的大）

👁 10.3w | 👍 3373 | 💬 185

Sunshine_Lin | 5月前 | 前端 · JavaScript · ECMAScript 6

「万字总结」熬夜总结50个JS的高级知识点，全都会你就是神!!!

👁 7.5w | 👍 2169 | 💬 107

sunshine小小倩 | 4年前 | JavaScript · 前端

this、apply、call、bind

👁 12.1w | 👍 3144 | 💬 250

前端阿飞 | 4月前 | 前端 · JavaScript

10个常见的前端手写功能，你全都会吗？

👁 8.8w | 👍 2210 | 💬 177

👍 601

💬 31

★ 收藏

写给女朋友的中级前端面试秘籍（含详细答案，15k级别）

👁 18.2w 👍 2880 💬 195

老腰 | 2年前 | 前端 · JavaScript · 全栈

8年前端开发的知识点沉淀(不知道会多少字，一直写下去吧...)

👁 9.3w 👍 2645 💬 183

大帅老猿 | 10月前 | 前端 · JavaScript

产品经理：你能不能用div给我画条龙？

👁 10.8w 👍 2901 💬 576

薄荷前端 | 3年前 | JavaScript · 前端 · Google

7分钟理解JS的节流、防抖及使用场景

👁 11.5w 👍 2430 💬 177

大帅老猿 | 9月前 | 前端 · JavaScript

三种前端实现VR全景看房的方案！说不定哪天就用得上！

👁 6.5w 👍 2920 💬 259

yck | 12月前 | JavaScript · 前端

17K star 仓库，解决 90% 的大厂基础面试题

👁 8.0w 👍 2949 💬 139

👍 601

💬 31

☆ 收藏

如何用 CSS 中写出超级美丽的阴影效果

👁 30.4w 👍 231 💬 12

老姚 | 2年前 | JavaScript · 前端

你未必知道的CSS知识点（第二季）

👁 6.4w 👍 2481 💬 196

Sunshine_Lin | 7月前 | JavaScript · 面试

基础很好？总结了38个ES6-ES12的开发技巧，倒要看看你能拿几分？🧐

👁 6.4w 👍 2229 💬 327

前端劝退师 | 3年前 | JavaScript · 面试

「中高级前端面试」JavaScript手写代码无敌秘籍

👁 16.1w 👍 3363 💬 181

chenhongdong | 4年前 | Vue.js · JavaScript · 前端

不好意思！耽误你的十分钟，让MVVM原理还给你

👁 7.4w 👍 2078 💬 288

yeyan1996 | 2年前 | JavaScript

字节跳动面试官：请你实现一个大文件上传和断点续传

👁 21.5w 👍 5039 💬 533

👍 601

💬 31

★ 收藏

神三元 | 2年前 | JavaScript 前端

(建议收藏)原生JS灵魂之问, 请问你能接得住几个? (上)

👁 14.7w 👍 3258 💬 232

洛霞同学 | 1年前 | JavaScript

32个手写JS, 巩固你的JS基础 (面试高频)

👁 9.1w 👍 2605 💬 163

光光同学 | 3年前 | JavaScript

一次弄懂Event Loop (彻底解决此类面试问题)

👁 8.6w 👍 1771 💬 133

👍 601

💬 31

☆ 收藏