前端基础进阶（八）：深入详解函数的柯里化

配图与本文无关

柯里化是函数的一个比较高级的应用，想要理解它并不简单。因此我一直在思考应该如何更加表达才能让大家理解起来更加容易。

> 以下是新版本讲解。高阶函数章节由于一些原因并未公开，大家可以自行搜索学习

通过上一个章节的学习我们知道，接收函数作为参数的函数，都可以叫做高阶函数。我们常常利用高阶函数来封装一些公共的逻辑。

这一章我们要学习的柯里化，其实就是高阶函数的一种特殊用法。

柯里化是指这样一个函数(假设叫做createCurry)，他接收函数A作为参数，运行后能够返回一个新的函数。并且这个新的函数能够处

理函数A的剩余参数。

这样的定义可能不太好理解，我们可以通过下面的例子配合理解。

假如有一个接收三个参数的函数A。

```
function A(a, b, c) {
    // do something
}
```

又假如我们有一个已经封装好了的柯里化通用函数createCurry。他接收bar作为参数，能够将A转化为柯里化函数，返回结果就是这个被转化之后的函数。

```
var _A = createCurry(A);
```

那么_A作为createCurry运行的返回函数，他能够处理A的剩余参数。因此下面的运行结果都是等价的。

```
_A(1, 2, 3);
_A(1, 2)(3);
_A(1)(2, 3);
_A(1)(2)(3);
A(1, 2, 3);
```

函数A被createCurry转化之后得到柯里化函数_A，_A能够处理A的所有剩余参数。因此柯里化也被称为部分求值。

在简单的场景下，我们可以不用借助柯里化通用式来转化得到柯里化函数，我们可以凭借眼力自己封装。

例如有一个简单的加法函数，他能够将自身的三个参数加起来并返回计算结果。

```
function add(a, b, c) {
    return a + b + c;
}
```

那么add函数的柯里化函数_add则可以如下：

```
function _add(a) {
    return function(b) {
        return function(c) {
            return a + b + c;
        }
    }
}
```

因此下面的运算方式是等价的。

```
add(1, 2, 3);
_add(1)(2)(3);
```

当然，柯里化通用式具备更加强大的能力，我们靠眼力自己封装的柯里化函数则自由度偏低。因此我们仍然需要知道自己如何去封装这样一个柯里化的通用式。

首先通过_add可以看出，柯里化函数的运行过程其实是一个参数的收集过程，我们将每一次传入的参数收集起来，并在最里层里面处理。因此我们在实现createCurry时，可以借助这个思路来进行封装。

封装如下：

```
// 简单实现，参数只能从右到左传递
function createCurry(func, args) {

    var arity = func.length;
    var args = args || [];
```

```
    return function() {
        var _args = [].slice.call(arguments);
        [].push.apply(_args, args);

        // 如果参数个数小于最初的func.length，则递归调用，继续收集参数
        if (_args.length < arity) {
            return createCurry.call(this, func, _args);
        }

        // 参数收集完毕，则执行func
        return func.apply(this, _args);
    }
}
```

尽管我已经做了足够详细的注解，但是我想理解起来也并不是那么容易，因此建议大家用点耐心多阅读几遍。这个createCurry函数的封装借助闭包与递归，实现了一个参数收集，并在收集完毕之后执行所有参数的一个过程。

因此聪明的读者可能已经发现，把函数经过createCurry转化为一个柯里化函数，最后执行的结果，不是正好相当于执行函数自身吗？柯里化是不是把简单的问题复杂化了？

如果你能够提出这样的问题，那么说明你确实已经对柯里化有了一定的了解。柯里化确实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。

我们来举一个非常常见的例子。

如果我们想要验证一串数字是否是正确的手机号，那么按照普通的思路来做，大家可能是这样封装，如下：

```
function checkPhone(phoneNumber) {
    return /^1[34578]\d{9}$/.test(phoneNumber);
}
```

而如果我们想要验证是否是邮箱呢？这么封装：

```
function checkEmail(email) {
    return /^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/.test(email);
}
```

我们还可能会遇到验证身份证号，验证密码等各种验证信息，因此在实践中，为了统一逻辑，，我们就会封装一个更为通用的函数，将用于验证的正则与将要被验证的字符串作为参数传入。

```
function check(targetString, reg) {
    return reg.test(targetString);
}
```

但是这样封装之后，在使用时又会稍微麻烦一点，因为会总是输入一串正则，这样就导致了使用时的效率低下。

```
check(/^1[34578]\d{9}$/, '14900000088');
check(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/, 'test@163.com');
```

那么这个时候，我们就可以借助柯里化，在check的基础上再做一层封装，以简化使用。

```
var _check = createCurry(check);

var checkPhone = _check(/^1[34578]\d{9}$/);
var checkEmail = _check(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/);
```

最后在使用的时候就会变得更加直观与简洁了。

```
checkPhone('183888888');
checkEmail('xxxxx@test.com');
```

经过这个过程我们发现，柯里化能够应对更加复杂的逻辑封装。当情况变得多变，柯里化依然能够应付自如。

虽然柯里化确实在一定程度上将问题复杂化了，也让代码更加不容易理解，但是柯里化在面对复杂情况下的灵活性却让我们不得不爱。

当然这个案例本身情况还算简单，所以还不能够特别明显的凸显柯里化的优势，我们的主要目的在于借助这个案例帮助大家了解柯里化在实践中的用途。

我们继续来思考一个例子。这个例子与map有关。在高阶函数的章节中，我们分析了封装map方法的思考过程。由于我们没有办法确认一个数组在遍历时会执行什么操作，因此我们只能将调用for循环的这个统一逻辑封装起来，而具体的操作则通过参数传入的形式让使用者自定义。这就是map函数。

但是，这是针对了所有的情况我们才会这样想。

实践中我们常常会发现，在我们的某个项目中，针对于某一个数组的操作其实是固定的，也就是说，同样的操作，可能会在项目的不同地方调用很多次。

于是，这个时候，我们就可以在map函数的基础上，进行二次封装，以简化我们在项目中的使用。假如这个在我们项目中会调用多次的操作是将数组的每一项都转化为百分比 1 --> 100%。

普通思维下我们可以这样来封装。

```
function getNewArray(array) {
    return array.map(function(item) {
        return item * 100 + '%'
    })
}
```

```
getNewArray([1, 2, 3, 0.12]);   // ['100%', '200%', '300%', '12%'];
```

而如果借助柯里化来二次封装这样的逻辑，则会如下实现：

```
function _map(func, array) {
    return array.map(func);
}

var _getNewArray = createCurry(_map);

var getNewArray = _getNewArray(function(item) {
    return item * 100 + '%'
})

getNewArray([1, 2, 3, 0.12]);   // ['100%', '200%', '300%', '12%'];
getNewArray([0.01, 1]); // ['1%', '100%']
```

如果我们的项目中的固定操作是希望对数组进行一个过滤，找出数组中的所有Number类型的数据。借助柯里化思维我们可以这样做。

```
function _filter(func, array) {
    return array.filter(func);
}

var _find = createCurry(_filter);

var findNumber = _find(function(item) {
    if (typeof item == 'number') {
        return item;
    }
})

findNumber([1, 2, 3, '2', '3', 4]); // [1, 2, 3, 4]

// 当我们继续封装另外的过滤操作时就会变得非常简单
// 找出数字为20的子项
var find20 = _find(function(item, i) {
    if (typeof item === 20) {
        return i;
    }
```

```
})
find20([1, 2, 3, 30, 20, 100]);  // 4

// 找出数组中大于100的所有数据
var findGreater100 = _find(function(item) {
    if (item > 100) {
        return item;
    }
})
findGreater100([1, 2, 101, 300, 2, 122]); // [101, 300, 122]
```

我采用了与check例子不一样的思维方向来想大家展示我们在使用柯里化时的想法。目的是想告诉大家，柯里化能够帮助我们应对更多更复杂的场景。

当然不得不承认，这些例子都太简单了，简单到如果使用柯里化的思维来处理他们显得有一点多此一举，而且变得难以理解。因此我想读者朋友们也很难从这些例子中感受到柯里化的魅力。不过没关系，如果我们能够通过这些例子掌握到柯里化的思维，那就是最好的结果了。在未来你的实践中，如果你发现用普通的思维封装一些逻辑慢慢变得困难，不妨想一想在这里学到的柯里化思维，应用起来，柯里化足够强大的自由度一定能给你一个惊喜。

当然也并不建议在任何情况下以炫技为目的的去使用柯里化，在柯里化的实现中，我们知道柯里化虽然具有了更多的自由度，但同时柯里化通用式里调用了arguments对象，使用了递归与闭包，因此柯里化的自由度是以牺牲了一定的性能为代价换来的。只有在情况变得复杂时，才是柯里化大显身手的时候。

额外知识补充

无限参数的柯里化。

> 该部分内容可忽略

在前端面试中，你可能会遇到这样一个涉及到柯里化的题目。

```
// 实现一个add方法，使计算结果能够满足如下预期：
add(1)(2)(3) = 6;
add(1, 2, 3)(4) = 10;
add(1)(2)(3)(4)(5) = 15;
```

这个题目的目的是想让add执行之后返回一个函数能够继续执行，最终运算的结果是所有出现过的参数之和。而这个题目的难点则在于参数的不固定。我们不知道函数会执行几次。因此我们不能使用上面我们封装的createCurry的通用公式来转换一个柯里化函数。只能自己封装，那么怎么办呢？在此之前，补充2个非常重要的知识点。

一个是ES6函数的不定参数。假如我们有一个数组，希望把这个数组中所有的子项展开传递给一个函数作为参数。那么我们应该怎么做？

```
// 大家可以思考一下，如果将args数组的子项展开作为add的参数传入
function add(a, b, c, d) {
    return a + b + c + d;
}
var args = [1, 3, 100, 1];
```

在ES5中，我们可以借助之前学过的apply来达到我们的目的。

```
add.apply(null, args);  // 105
```

而在ES6中，提供了一种新的语法来解决这个问题，那就是不定参。写法如下：

```
add(...args);  // 105
```

这两种写法是等效的。OK，先记在这里。在接下的实现中，我们会用到不定参数的特性。

第二个要补充的知识点是函数的隐式转换。当我们直接将函数参与其他的计算时，函数会默认调用toString方法，直接将函数体转换为字符串参与计算。

```
function fn() { return 20 }
console.log(fn + 10);     // 输出结果 function fn() { return 20 }10
```

但是我们可以重写函数的toString方法，让函数参与计算时，输出我们想要的结果。

```
function fn() { return 20; }
fn.toString = function() { return 30 }

console.log(fn + 10); // 40
```

除此之外，当我们重写函数的valueOf方法也能够改变函数的隐式转换结果。

```
function fn() { return 20; }
fn.valueOf = function() { return 60 }

console.log(fn + 10); // 70
```

当我们同时重写函数的toString方法与valueOf方法时，最终的结果会取valueOf方法的返回结果。

```
function fn() { return 20; }
fn.valueOf = function() { return 50 }
fn.toString = function() { return 30 }

console.log(fn + 10); // 60
```

补充了这两个知识点之后，我们可以来尝试完成之前的题目了。add方法的实现仍然会是一个参数的收集过程。当add函数执行到最后

时，仍然返回的是一个函数，但是我们可以通过定义toString/valueOf的方式，让这个函数可以直接参与计算，并且转换的结果是我们想要的。而且它本身也仍然可以继续执行接收新的参数。实现方式如下。

```
function add() {
    // 第一次执行时，定义一个数组专门用来存储所有的参数
    var _args = [].slice.call(arguments);

    // 在内部声明一个函数，利用闭包的特性保存_args并收集所有的参数值
    var adder = function () {
        var _adder = function() {
            // [].push.apply(_args, [].slice.call(arguments));
            _args.push(...arguments);
            return _adder;
        };

        // 利用隐式转换的特性，当最后执行时隐式转换，并计算最终的值返回
        _adder.toString = function () {
            return _args.reduce(function (a, b) {
                return a + b;
            });
        }

        return _adder;
    }
    // return adder.apply(null, _args);
    return adder(..._args);
}

var a = add(1)(2)(3)(4);   // f 10
var b = add(1, 2, 3, 4);   // f 10
var c = add(1, 2)(3, 4);   // f 10
var d = add(1, 2, 3)(4);   // f 10

// 可以利用隐式转换的特性参与计算
console.log(a + 10); // 20
console.log(b + 20); // 30
console.log(c + 30); // 40
console.log(d + 40); // 50

// 也可以继续传入参数，得到的结果再次利用隐式转换参与计算
```

```
console.log(a(10) + 100);  // 120
console.log(b(10) + 100);  // 120
console.log(c(10) + 100);  // 120
console.log(d(10) + 100);  // 120

// 其实上栗中的add方法，就是下面这个函数的柯里化函数，只不过我们并没有使用通用式对
function add(...args) {
    return args.reduce((a, b) => a + b);
}
```

> 以下为老版本讲解，请勿阅读学习，因为部分思维并不完全正确。

一、补充知识点之函数的隐式转换

JavaScript作为一种弱类型语言，它的隐式转换是非常灵活有趣的。当我们没有深入了解隐式转换的时候可能会对一些运算的结果会感动困惑，比如`4 + true = 5`。当然，如果对隐式转换了解足够深刻，肯定是能够很大程度上提高对js的使用能力。只是我没有打算将所有的隐式转换规则分享给大家，这里暂时只分享一下，函数在隐式转换中的一些规则。

来一个简单的思考题。

```
function fn() {
    return 20;
}

console.log(fn + 10); // 输出结果是多少?
```

稍微修改一下，再想想输出结果会是什么?

```
function fn() {
    return 20;
}
```

```
fn.toString = function() {
    return 10;
}

console.log(fn + 10);  // 输出结果是多少?
```

还可以继续修改一下。

```
function fn() {
    return 20;
}

fn.toString = function() {
    return 10;
}

fn.valueOf = function() {
    return 5;
}

console.log(fn + 10); // 输出结果是多少?

// 输出结果分别为
function fn() {
    return 20;
}10

20

15
```

当使用console.log，或者进行运算时，隐式转换就可能会发生。从上面三个例子中我们可以得出一些关于函数隐式转换的结论。

> 当我们没有重新定义toString与valueOf时，函数的隐式转换会调用默认的toString方法，它会将函数的定义内容作为字符串返回。而当我们主动定义了toString/vauleOf方法时，那么隐式转换的返回结果则由我们自己控制了。其中valueOf会比toString后执行

因此上面例子的结论就很容易理解了。建议大家动手尝试一下。

二、补充知识点之利用call/apply封数组的map方法

> map(): 对数组中的每一项运行给定函数，返回每次函数调用的结果组成的数组。

通俗来说，就是遍历数组的每一项元素，并且在map的第一个参数（回调函数）中进行运算处理后返回计算结果。返回一个由所有计算结果组成的新数组。

```
// 回调函数中有三个参数
// 第一个参数表示newArr的每一项，第二个参数表示该项在数组中的索引值
// 第三个表示数组本身
// 除此之外，回调函数中的this，当map不存在第二参数时，this指向丢失，当存在第二个
var newArr = [1, 2, 3, 4].map(function(item, i, arr) {
    console.log(item, i, arr, this);  // 可运行试试看
    return item + 1;  // 每一项加1
}, { a: 1 })

console.log(newArr); // [2, 3, 4, 5]
```

在上面例子的注释中详细阐述了map方法的细节。现在要面临一个难题，就是如何封装map。

可以先想想for循环。我们可以使用for循环来实现一个map，但是在封装的时候，我们会考虑一些问题。我们在使用for循环的时候，一个循环过程确实很好封装，但是我们在for循环里面要对每一项做的事情却很难用一个固定的东西去把它封装起来。因为每一个场景，for循环里对数据的处理肯定都是不一样的。

于是大家就想了一个很好的办法，将这些不一样的操作单独用一个函数来处理，让这个函数成为map方法的第一个参数，具体这个回调函数中会是什么样的操作，则由我们自己在使用时决定。因此，根据这个思路的封装实现如下。

```
Array.prototype._map = function(fn, context) {
    var temp = [];
    if(typeof fn == 'function') {
        var k = 0;
        var len = this.length;
        // 封装for循环过程
        for(; k < len; k++) {
            // 将每一项的运算操作丢进fn里，利用call方法指定fn的this指向与具体
            temp.push(fn.call(context, this[k], k, this))
        }
    } else {
        console.error('TypeError: '+ fn +' is not a function.');
    }

    // 返回每一项运算结果组成的新数组
    return temp;
}

var newArr = [1, 2, 3, 4]._map(function(item) {
    return item + 1;
})
// [2, 3, 4, 5]
```

在上面的封装中，我首先定义了一个空的temp数组，该数组用来存储最终的返回结果。在for循环中，每循环一次，就执行一次参数fn函数，fn的参数则使用call方法传入。

> 在理解了map的封装过程之后，我们就能够明白为什么我们在使用map时，总是期望能够在第一个回调函数中有一个返回值了。在eslint的规则中，如果我们在使用map时没有设置一个返回值，就会被判定为错误。

ok，明白了函数的隐式转换规则与call/apply在这种场景的使用方式，我们就可以尝试通过简单的例子来了解一下柯里化了。

三、由浅入深的柯里化

在前端面试中有一个关于柯里化的面试题，流传甚广。

> 实现一个add方法，使计算结果能够满足如下预期：
> add(1)(2)(3) = 6
> add(1, 2, 3)(4) = 10
> add(1)(2)(3)(4)(5) = 15

很明显，计算结果正是所有参数的和，add方法每运行一次，肯定返回了一个同样的函数，继续计算剩下的参数。

我们可以从最简单的例子一步一步寻找解决方案。

当我们只调用两次时，可以这样封装。

```
function add(a) {
    return function(b) {
        return a + b;
    }
}

console.log(add(1)(2));  // 3
```

如果只调用三次：

```
function add(a) {
    return function(b) {
        return function (c) {
            return a + b + c;
        }
    }
}

console.log(add(1)(2)(3)); // 6
```

上面的封装看上去跟我们想要的结果有点类似，但是参数的使用被限制得很死，因此并不是我们想要的最终结果，我们需要通用的封装。应该怎么办？总结一下上面2个例子，其实我们是利用闭包的特性，将所有的参数，集中到最后返回的函数里进行计算并返回结

果。因此我们在封装时，主要的目的，就是将参数集中起来计算。

来看看具体实现。

```
function add() {
    // 第一次执行时，定义一个数组专门用来存储所有的参数
    var _args = [].slice.call(arguments);

    // 在内部声明一个函数，利用闭包的特性保存_args并收集所有的参数值
    var adder = function () {
        var _adder = function() {
            [].push.apply(_args, [].slice.call(arguments));
            return _adder;
        };

        // 利用隐式转换的特性，当最后执行时隐式转换，并计算最终的值返回
        _adder.toString = function () {
            return _args.reduce(function (a, b) {
                return a + b;
            });
        }

        return _adder;
    }
    return adder.apply(null, [].slice.call(arguments));
}

// 输出结果，可自由组合的参数
console.log(add(1, 2, 3, 4, 5));  // 15
console.log(add(1, 2, 3, 4)(5));  // 15
console.log(add(1)(2)(3)(4)(5));  // 15
```

上面的实现，利用闭包的特性，主要目的是想通过一些巧妙的方法将所有的参数收集在一个数组里，并在最终隐式转换时将数组里的所有项加起来。因此我们在调用add方法的时候，参数就显得非常灵活。当然，也就很轻松的满足了我们的需求。

那么读懂了上面的demo，然后我们再来看看柯里化的定义，相信大家就会更加容易理解了。

> 柯里化（英语：Currying），又称为部分求值，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回一个新的函数的技术，新函数接受余下参数并返回运算结果。

- 接收单一参数，因为要携带不少信息，因此常常以回调函数的理由来解决。
- 将部分参数通过回调函数等方式传入函数中
- 返回一个新函数，用于处理所有的想要传入的参数

在上面的例子中，我们可以将`add(1, 2, 3, 4)`转换为`add(1)(2)(3)(4)`。这就是部分求值。每次传入的参数都只是我们想要传入的所有参数中的一部分。当然实际应用中，并不会常常这么复杂的去处理参数，很多时候也仅仅只是分成两部分而已。

咱们再来一起思考一个与柯里化相关的问题。

> 假如有一个计算要求，需要我们将数组里面的每一项用我们自己想要的字符给连起来。我们应该怎么做？想到使用join方法，就很简单。

```
var arr = [1, 2, 3, 4, 5];

// 实际开发中并不建议直接给Array扩展新的方法
// 只是用这种方式演示能够更加清晰一点
Array.prototype.merge = function(chars) {
    return this.join(chars);
}

var string = arr.merge('-')

console.log(string);  // 1-2-3-4-5
```

增加难度，将每一项加一个数后再连起来。那么这里就需要map来帮助我们对每一项进行特殊的运算处理，生成新的数组然后用字符连

接起来了。实现如下：

```
var arr = [1, 2, 3, 4, 5];

Array.prototype.merge = function(chars, number) {
    return this.map(function(item) {
        return item + number;
    }).join(chars);
}

var string = arr.merge('-', 1);

console.log(string); // 2-3-4-5-6
```

但是如果我们又想要让数组每一项都减去一个数之后再连起来呢？当然和上面的加法操作一样的实现。

```
var arr = [1, 2, 3, 4, 5];

Array.prototype.merge = function(chars, number) {
    return this.map(function(item) {
        return item - number;
    }).join(chars);
}

var string = arr.merge('~', 1);

console.log(string); // 0~1~2~3~4
```

机智的小伙伴肯定发现困惑所在了。我们期望封装一个函数，能同时处理不同的运算过程，但是我们并不能使用一个固定的套路将对每一项的操作都封装起来。于是问题就变成了和封装map的时候所面临的问题一样了。我们可以借助柯里化来搞定。

与map封装同样的道理，既然我们事先并不确定我们将要对每一项数据进行怎么样的处理，我只是知道我们需要将他们处理之后然后用字符连起来，所以不妨将处理内容保存在一个函数里。而仅仅固定

封装连起来的这一部分需求。

于是我们就有了以下的封装。

```
// 封装很简单，一句话搞定
Array.prototype.merge = function(fn, chars) {
    return this.map(fn).join(chars);
}

var arr = [1, 2, 3, 4];

// 难点在于，在实际使用的时候，操作怎么来定义，利用闭包保存于传递num参数
var add = function(num) {
    return function(item) {
        return item + num;
    }
}

var red = function(num) {
    return function(item) {
        return item - num;
    }
}

// 每一项加2后合并
var res1 = arr.merge(add(2), '-');

// 每一项减2后合并
var res2 = arr.merge(red(1), '-');

// 也可以直接使用回调函数，每一项乘2后合并
var res3 = arr.merge((function(num) {
    return function(item) {
        return item * num
    }
})(2), '-')

console.log(res1); // 3-4-5-6
console.log(res2); // 0-1-2-3
console.log(res3); // 2-4-6-8
```

大家能从上面的例子，发现柯里化的特征吗?

四、柯里化通用式

通用的柯里化写法其实比我们上边封装的add方法要简单许多。

```
var currying = function(fn) {
    var args = [].slice.call(arguments, 1);

    return function() {
        // 主要还是收集所有需要的参数到一个数组中，便于统一计算
        var _args = args.concat([].slice.call(arguments));
        return fn.apply(null, _args);
    }
}

var sum = currying(function() {
    var args = [].slice.call(arguments);
    return args.reduce(function(a, b) {
        return a + b;
    })
}, 10)

console.log(sum(20, 10));  // 40
console.log(sum(10, 5));   // 25
```

五、柯里化与bind

```
Object.prototype.bind = function(context) {
    var _this = this;
    var args = [].slice.call(arguments, 1);

    return function() {
        return _this.apply(context, args)
    }
}
```

这个例子利用call与apply的灵活运用，实现了bind的功能。

在前面的几个例子中，我们可以总结一下柯里化的特点：

- 接收单一参数，将更多的参数通过回调函数来搞定?
- 返回一个新函数，用于处理所有的想要传入的参数；
- 需要利用call/apply与arguments对象收集参数；
- 返回的这个函数正是用来处理收集起来的参数。

希望大家读完之后都能够大概明白柯里化的概念，如果想要熟练使用它，就需要我们掌握更多的实际经验才行。