【前端面试】手动实现一个curry工具函数?

前言

“函数柯里化”可能是很多小伙伴经常听到的一个概念，这也是在面试中很常考的。柯里化是我们英译过来的，它实际被称作”currying”。很多小伙伴可能对“函数柯里化”总是云里雾里的，但是事实上在项目中，其实很多地方都有“函数柯里化”的存在，只是我们没有发现而已，今天我们就来彻底学会它！

1.函数柯里化概念

“柯里化”并不是一个实实在在的东西，可以说它是一种模型，也可以说它是一个概念，就像我们的链表、队列等等，实现它们的方式有很多种，比如可以用Java、JavaScript、C++实现，它们只是提供了一个基础概念模型给我们，我们可以用各种语言技术实现它们。

柯里化可能每个人理解的都有区别，但是核心点是不变的，我们给出一个偏官方的解释。

官方解释：

> 在计算机科学中，柯里化（Currying）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，并且返回接受余下的参数且返回结果的新函数的技术。

上面的解释是比较偏官方的，一般这种解释都没有那么通俗易懂，我们可以用我们自己的语言给大家解释一遍，因为我们这里的“柯里化”是在前端背景下，所以我们只针对我们前端的小伙伴解释。

通俗解释：

> 柯里化是一门儿技术，它可以让我们接收多个参数的函数变为只接受一个参数的函数，比如将函数fn(a,b,c)变为fn(a)、fn(b)、fn(c)，我们可以这样进行调用：fn(a)(b)(c)。并且可以返回正确的结果。

上面的解释我们可以抓住两个关键词：**技术、一个参数**。由于柯里化通常是针对于函数的，所以我们通常称作“函数柯里化”，其实“函数柯里化”与我们熟知的数组扁平化有着异曲同工之妙，数组扁平化是将多维数组扁平化为一维数组，而函数柯里化是将接收多个参数的函数改造为只接受一个参数的函数。

为了更加好理解函数柯里化做了什么，我们可以简单写一段代码演示。

示例代码：

```
<script>
  // 一个普通函数
  function sum(a, b, c) {
    return a + b + c;
  }
  // 正常调用
  sum(10, 20, 30);

  // 将sum函数柯里化,返回一个新函数
  let newFn = curry(sum);

  // 新的调用方式
  newFn(10)(20)(30);

  // 提示
  // newFn(10)返回一个函数
  // newFn(20)返回一个函数
  // newFn(30)返回最终结果
</script>
```

上段代码有一个curry函数，这也就是我们本篇文章需要实现的工具函数，它的目的就是将函数柯里化，返回柯里化后的函数，从而实现一个函数只接收一个参数。

注意：

最然通常意义的函数柯里化是让函数只接收一个参数，但是在我们JavaScript里面，我们柯里化通常更宽松一点，函数可以接收任意参数，比如下方调用方式：

```
newFn(10)(20)(30);
newFn(10, 20)(30);
```

2.为什么要柯里化?

看了函数柯里化的概念之后，我相信很多人都不理解为什么要这么做？从表面上看，函数柯里化似乎是把简单的事情复杂化了，事实也确实如此。但是凡事都有两面性，有好处就会有弊端，函数柯里化也是如此，我们这里简单总结一下它的**优缺点**：

优点：

> 函数柯里化之后让函数变得更加单一，一次只接受一个参数，松散解耦。

缺点：

> 函数的通用性将变低，比如原来接收3个参数的函数，我们可以拿着3个参数处理更多操作，但是函数变为只接收一个参数后，我们的操作会受很多限制。

函数柯里化实际上是函数式编程中的一大重要思想：

> 一个函数只处理一件事，函数需要遵循只接收一个参数和只返回一个结果的规则。

柯里化函数有优点也有缺点，我们需要根据不同业务场景来判断是否需要函数柯里化，我们这里简单举几个例子，让小伙伴们理解函数柯里化的好处。

场景1：实现参数复用

示例代码：

```
// 该函数传入3个参数：协议、域名、路径，返回完整url
function getUrl(protocol, domain, path) {
  return `${protocol}://${domain}/${path}`
}

// 传统调用
getUrl('http', 'smallpig.site', 'articl/12058');
getUrl('http', 'smallpig.site', 'articl/13258');
getUrl('http', 'smallpig.site', 'articl/12438');
getUrl('http', 'smallpig.site', 'articl/12238');

// 将函数柯里化
let getAllUrl = curry(getUrl)('http', 'smallpig.site')

// 柯里化后调用
getAllUrl('articl/12058');
getAllUrl('articl/13258');
getAllUrl('articl/12438');
getAllUrl('articl/12238');
```

上面代码中我们原来的函数分别接收三个参数：协议、域名、路径。调用时需要分别传入这三个参数，但是我们发现协议和域名每次传入都是相同的。

所以我们借助curry工具函数将原函数柯里化，最终返回getAllUrl只接收一个参数的函数，大家需要注意的是curry(getUrl)('http', 'smallpig.site')返回的是一个函数。

经过柯里化之后，getAllUrl函数语义非常明了，但是有些小伙伴可能会说，这么做太麻烦了，如果协议和域名不一样，该函数岂不是不能用了？事实确实如此，但是不可否认的是，柯里化函数语义确实明了，而且它松耦合了，这就是降低函数通用性的代价，所以说凡事都是等价的！

场景2：封装map、filter等函数

示例代码：

```
let arr = [{
  name: '小猪课堂',
  age: 26
}, {
  name: '会飞的猪',
  age: 26
}]
// 目标：获取所有name值、age值
```

```
// 原方式
let names = arr.map((item) => {
  return item.name;
})
let ages = arr.map((item) => {
  return item.age;
})

// 将函数柯里化
let getProps = curry(function (key, obj) {
  return obj[key];
});

// 柯里化函数调用
let names1 = arr.map(getProps('name'));
let ages1 = arr.map(getProps('age'));
```

上段代码中我们想要获取对象数组中name和age值，通过函数柯里化后，我们实际上只通过一行代码就实现了需求，而且语义明了。

场景3：延迟执行

示例代码：

```
// 将函数柯里化
let getAllUrl = curry(getUrl)('http', 'smallpig.site')

// 柯里化后调用，延迟执行
getAllUrl('articl/12058');
getAllUrl('articl/13258');
getAllUrl('articl/12438');
getAllUrl('articl/12238');
```

其实函数柯里化后本身就有延迟执行的含义在里面，就好比我们的bind方法，返回的是一个新的函数，并不是马上执行函数。

3.柯里化实现思路

我们知道了函数柯里化的概念以及为什么要使用它之后，接下来总结一下它是如何实现的？我们拿出前文中的一段代码来举例。

示例代码：

```
<script>
  // 一个普通函数
  function sum(a, b, c) {
    return a + b + c;
  }
  // 将sum函数柯里化,返回一个新函数
  let newFn = curry(sum);
```

```
  // 新的调用方式
  newFn(10)(20)(30); // 60
  newFn(10, 20)(30); // 60
</script>
```

我们把newFn(10)(20)(30)拆开来看大家应该就很好理解了：

- newFn(10)返回的应该是一个函数，因为后面还要继续调用
- newFn(10)(20)同样返回的是一个函数，因为后面还继续调用
- newFn(10)(20)(30)返回的是最终结果。

所以我们总结curry函数有如下特点：

- curry函数返回一个新的函数newFn。
- 调用newFn时，如果参数累计小于原函数应该接收的参数个数时，继续返回一个函数。
- 当接受的累计参数大于或等于原函数应该接收的参数个数时，执行原函数。

具体实现思路：

1. 拿到原函数的形参个数le。
2. 拿到目前接收到的参数args。
3. 比较len和args大小。
4. 根据大小判断返回一个函数还是返回原函数执行结果。

4.实现curry工具函数

我们知道curry函数的特点和实现思路，那么接下来我们就需要用实际的代码要实现了。

简单版本curry函数：

```
/**
  * @params {Function} fn 原函数
  * @params {Array} ...args 可以传入初始参数
*/
function curry(fn, ...args) {
  // 返回一个函数
  return function () {
    // 缓存目前接收到的参数
    let _args = [...args, ...arguments];
    // 原函数应该接收的参数个数
    let len = fn.length;
    // 比较目前的参数累计与原函数应该接收的参数
    if (_args.length < len) {
      // 代表需要继续返回一个新函数
      // 使用递归，形成闭包，保证函数独立，不受影响。
      return curry(fn, ..._args);
    } else {
      // 参数累计够了，执行原函数返回结果
      return fn.apply(this, _args);
    }
```

```
  }
}

// 一个普通函数
function sum(a, b, c) {
  return a + b + c;
}
// 正常调用
console.log(sum(10, 20, 30)); // 60

// 将sum函数柯里化,返回一个新函数
let newFn = curry(sum);
// 新的调用方式
console.log(newFn(10)(20)(30)); // 60
console.log(newFn(10, 20)(30)); // 60
```

上段代码中curry函数实现的代码其实非常少，主要有两个点需要大家注意：

- 需要缓存每一次缓存传入的参数。
- 利用闭包和递归，隔离每次的作用域。
- fn.length获取的是函数的形参个数。

上面的简单版本以及突出了函数柯里化的核心原理，后续优化大家可以根据业务场景添加，比如添加类型判断、占位符等等。

总结

函数的柯里化实现过程并不复杂，知道了它的实现原理其实很容易自己手动实现一个curry函数，学完了本篇文章，我们做出如下总结：

- 函数柯里化降低了函数通用性，却提高了适用性。
- 函数柯里化主要应用场景：参数复用、延迟执行。
- 函数柯里化的重点在于闭包和递归，将每次执行的作用域保存在内存中，等待后续使用。