

## 05 闭包的应用

更新时间：2020-05-13 09:56:27



“

天才就是这样，终身努力，便成天才。——门捷列夫

”

### 闭包应用系列

“你在实际开发中，对闭包有哪些应用？”—— 紧随闭包基础题、编码题之后的，往往就是这样一个看似可以自由发挥，实则并不好得分的问题。面试官抛出这个问题的目的，主要是考察你编码的熟练度和知识的广度。你的回答必须有“下限”（不能太简短，他会觉得你脑子里没货），却没有“上限”（多多益善，高手可以抖出很多东西，故而区分度极高）。这里我为大家作个引导，如果你面对这样的问题感到言语匮乏、大脑一片空白，那么就往我说的这几个方向去答，往往是比较稳、也是面试官比较爱听的：

#### 1. 模拟私有变量的实现

#### 理解什么是私有变量

在课程调研的过程中，我发现一些非科班、没有 **Java** 基础的同学对私有变量的背景和场景还不太理解，所以我们先来捋捋私有变量是啥。

大家知道，**JS** 是面向对象的语言。但 **JS** 的面向对象，本质上是基于原型，而非像 **Java** 一样基于类（这点我们会在原型章节中着重讲）。在 **JS** 中，强调的是对象、而非类的概念。在 **ES6** 之前，我们生成对象实例，只有一条路，那就是用构造函数：

```

// 定义构造函数
function Dog(name) {
  this.name = name
}

// 挂载原型方法
Dog.prototype.showName = function() {
  console.log(this.name)
}

// 通过构造函数创建对象实例
var dog = new Dog('哈士奇')
// 输出 '哈士奇'
dog.showName()

```

如果你是有一定 **Java**、**C++** 基础的同学，这种写法可能会使你感到困惑 —— 毕竟，基于类的写法，才是应用更加广泛的面向对象实现方式。为了达到这个目的，早期的 **JS** 程序员，会手动去用构造函数去模拟 **Class**（具体的模拟方法，也是一个考点，我们会在原型章节展开）。模拟 **Class** 的诉求越来越强烈之后，**ES** 标准直接吸纳了这种模拟的思路，把模拟实现的 **Class** 内置掉，于是我们就开心地拥抱了 **ES6** 中的 **Class**：

```

// 构造函数，相当于上一个例子中的 Dog 函数
class Dog {
  constructor(name) {
    // 构造函数的函数体内容
    this.name = name
  }

  showName() {
    console.log(this.name)
  }
}

// 仍然是使用 new 关键字来创建实例
let dog = new Dog('哈士奇')
// 输出 '哈士奇'
dog.showName()

```

说是模拟实现类，就意味着不是“真的”类。**ES6** 中的这个 **class**，本质上仍然是构造函数的语法糖，所以上面两段代码其实本质上是一样的，只是写法不同。这模拟出来的类，仍然无法实现传统面向对象语言中的一些能力 —— 比如私有变量的定义和使用。

私有变量到底是干嘛的？为啥没它不行？大家看这样一个 **User** 类（下面是伪代码，大家不要直接丢进控制台运行，重点看注释）：

```

class User {
  constructor(username, password) {
    // 用户名
    this.username = username
    // 密码
    this.password = password
  }

  login() {
    // 使用 fetch 进行登录请求
    fetch(登录请求的目标url地址, {
      method: 'POST', // 指定请求方法为post
      body: JSON.stringify({
        username,
        password
      }), // 请求参数带上用户名和密码
      ... // 这里省略其它 fetch 参数
    }).then(res => res.json())
  }
}

```

在这个 `User` 类里，我们的本意是实现 `login` 这个能力，并且确保 `User` 的每一个实例都具备这个能力：

```

let user = new User('xiuyan', 'xiuyan123')

// 输出 login 函数的内容，说明 User 的实例中有 login 这个方法
user.login

```

输出效果：

```

> user.login
< login() {
  // 使用 fetch 进行登录请求
  fetch(登录请求的目标url地址, {
    method: 'POST', // 指定请求方法为post
    body: JSON.stringify({
      username,
      password
    }), // 请求参数带上用户名和密码
  }).then(r...

```

我们看到，`login` 方法已经成功的被实现了。但这里面藏着一个隐患，现在我尝试输出一下 `password`：

```

user.password

```

输出效果：

```

> user.password
< "xiuyan123"

```

我们惊恐地发现，登录密码这么关键且敏感的信息，竟然通过一个简单的属性就可以拿到！这就意味着，后面的人只要能拿到 `user` 这个对象，就可以非常轻松地得知、甚至改写他的密码。

这在业务开发中，是一个非常危险的操作：你不能保证你的队友拿到这个 `user` 之后不会误操作它的 `password`——你甚至不能保证三个月后的自己还会不会记得要保护这个 `password`！大家谨记，在软件世界中，只要是依赖人的意志才可以确保其安全稳定的东西，都是不可靠的。所以我们需要想办法从代码层面去保护 `password`。

像 `password` 这样的变量，我们希望它仅在对象内部生效，无法从外部触及，这样的变量，就是私有变量。

那么在 JS 中，既然无法通过 `private` 这样的关键字直接在类里声明变量的私有性，我们就只能另寻它法。这时候就轮到闭包登场了——大家想想，在内部可以拿到、外部拿不到，这难道不就是我们前面讲的函数作用域的特性吗？所以我们的思路就是把私有变量用函数作用域来保护起来，形成一个闭包！（大家关注注释里的解析内容）

```
// 利用闭包生成IIFE，返回 User 类
const User = (function() {
  // 定义私有变量_password
  let _password

  class User {
    constructor(username, password) {
      // 初始化私有变量_password
      _password = password
      this.username = username
    }

    login() {
      // 这里我们增加一行 console，为了验证 login 里仍可以顺利拿到密码
      console.log(this.username, _password)
      // 使用 fetch 进行登录请求，同上，此处省略
    }
  }

  return User
})();

let user = new User('xiuyan', 'xiuyan123')

console.log(user.username)
console.log(user.password)
console.log(user._password)
user.login()
```

在这段代码中，我们把 `_password` 放在了 `login` 方法的外层函数作用域里，并通过立即执行 `User` 这个函数，创造出了一个闭包的作用域环境。相对应的作用域层级关系及变量引用关系如下：

上面这段代码的输出结果如下：

xiuyan	<u>vendor.348c2b3...js:27</u>
undefined	<u>vendor.348c2b3...js:27</u>
undefined	<u>vendor.348c2b3...js:27</u>
xiuyan xiuyan123	<u>vendor.348c2b3...js:27</u>

我们看到不管是 `password`，还是 `_password`，都被好好地保护在了 `User` 这个立即执行函数的内部。此时 `user` 实例的组成如下：

```
> user
< ▼ User {username: "xiuyan"} ⓘ
  username: "xiuyan"
  __proto__:
    ▶ constructor: class User
    ▶ login: f login()
    ▶ __proto__: Object
```

我们看到它对外暴露的属性确实已经没有 `password`，通过闭包，我们成功达到了用自由变量来模拟私有变量的效果！

## 2. 偏函数与柯里化

不少同学看到这个标题可能会先懵一脸 —— 啥是偏函数？啥是柯里化？一个小标题七个字，我咋只认识 6 个！

不要虚，不要慌，很多小伙伴和你一样，对这些稍微偏计算机科学理论一些的知识心存畏惧 —— 所以这个知识点就是你的机会！想想吧，到时候面对同一个问题，隔壁面试的路人 A 连私有变量都捋不清楚的时候，你已经在侃侃而谈偏函数和柯里化了，差距拉开，offer 到手，完美！

其实柯里化和偏函数并不复杂（只是名字有点拗口），他们都是可以帮我们把需要多个入参的函数，转化为需要更少入参的函数的方法。

### 柯里化

让我们从考察频率相对更高的柯里化开始说起，维基百科中对柯里化有着这样的定义：

在计算机科学中，柯里化（英语：Currying），又译为卡瑞化或加里化，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而且返回结果的新函数的技术。

来，我们提取一下主要信息，说人话：

柯里化是把接受  $n$  个参数的 1 个函数改造为只接受 1 个参数的  $n$  个互相嵌套的函数的过程。也就是  $fn(a, b, c)$  会变成  $fn(a)(b)(c)$ 。

为啥要改造？如何改造？我们直接把自己代入场景里来改造一个看看，改造完就啥都明白了：

我们现在是一家电商公司，旗下有多个电商站点。为了确保商品名的唯一性，我们考虑使用 `prefix`（一个标识不同站点的前缀字符串）、`type`（商品类型）、`name`（商品原本名称）三个字符串拼接的方式来为商品生成一个完整版名称。对应的方法如下：

```
function generateName(prefix, type, itemName) {
  return prefix + type + itemName
}
```

我们看到这个方法里需要视情况传入 `prefix`、`type`、`name` 参数。如果是作为一个细分工种的 `leader`，我可能只会负责一个站点的业务。比如我负责了“大卖网”的业务，那我每次生成商品名时，都会这样传参：

```
// itemName 是原有商品名
generateName('大卖网', type, itemName)
```

发现问题没有？这里面 `prefix` 其实是一个固定的入参，而我们每次都还要手动把它告诉给 `generateName` 函数，这很不爽。

如果是作为一个细分工种的程序员，我负责的东西可能更具体了，比如仅仅负责“大卖网”站点下的“母婴”类商品，那么我每次生成完整名称的时候，调用这个函数就是这样传参的：

```
// itemName 是原有商品名
generateName('大卖网', '母婴', itemName)
```

隔壁组的小哥，他只负责“洗菜网”站点下的“生鲜”类商品，那么他每次是这样传参的：

```
// itemName 是原有商品名
generateName('洗菜网', '生鲜', itemName)
```

一样的道理，无论是站在我的角度、还是隔壁组小哥的角度，对我们各自来说，调用 `generateName` 时其实真正的变量只有 `itemName` 一个，而我们却每次都不得不把前两个参数也手动传一遍。

此时我们多么希望，有一种魔法，可以让函数在必要的情况下帮我们“记住”一部分入参。在这个场景下，柯里化可以帮我们很大的忙。现在我们对 `generateName` 进行柯里化（解析在注释里）：

```
function generateName(prefix) {
  return function(type) {
    return function(itemName) {
      return prefix + type + itemName
    }
  }
}

// 生成大卖网商品名专属函数
var salesName = generateName('大卖网')

// “记住”prefix，生成大卖网母婴商品名专属函数
var salesBabyName = salesName('母婴')

// “记住”prefix和type，生成洗菜网生鲜商品名专属函数
var vegFreshName = generateName('洗菜网')('生鲜')

// 输出 '大卖网母婴奶瓶'
salesBabyName('奶瓶')
// 输出 '洗菜网生鲜菠菜'
vegFreshName('菠菜')

// 啥也不记，直接生成一个商品名
var itemFullName = generateName('洗菜网')('生鲜')('菠菜')
```

我们看到，在新的 `generateName` 函数中，我们可以以自由变量的形式将 `prefix`、`type` 的值保留在 `generateName` 内部的两层嵌套的外部作用域里。

这样一来，原有的 `generateName (prefix, type, name)` 现在经过柯里化已经变成了 `generateName (prefix)(type)(itemName)`。通过后者这种形式，我们可以选择性地决定是否要“记住” `prefix`、`type`，从而即时地生成更加符合我们预期的、复用程度更高的目标函数。此外，柯里化还可以帮助我们以嵌套的形式把多个函数的能力组合到一起，这就是柯里化的魅力。

## 偏函数应用与柯里化的辨析

如果你能够理解柯里化，那么偏函数应用对你来说就是小菜一碟了～

柯里化是将一个 `n` 个参数的函数转换成 `n` 个单参数函数。你有 10 个入参，就得嵌套 10 层函数，且每层函数都只能有 1 个入参。它的目标就是把函数的入参拆解为精准的 `n` 部分。

偏函数应用相比之下就“随意”一些了。偏函数是说，固定你函数的某一个或几个参数，然后返回一个新的函数（这个函数用于接收剩下的参数）。你有 10 个入参，你可以只固定 2 个入参，然后返回一个需要 8 个入参的函数——偏函数应用是不强调“单参数”这个概念的。它的目标仅仅是把函数的入参拆解为两部分。

**tips:** 很多文章 / 教程会混淆偏函数应用和柯里化的概念，这里大家要多多留心，避免在面试的时候闹笑话～

## 偏函数应用

除了约束条件与柯里化略有不同，偏函数在动机和实现思路上都与柯里化一致——动机就是为了“记住”函数的一部分参数，实现思路就是走闭包。

仍然是上面的例子。我们单纯地把一口气传入 3 个入参，拆分为先传 1 个、再传 2 个，这样就算实现了偏函数应用：

原有的函数形式与调用方法

```
function generateName(prefix, type, itemName) {
  return prefix + type + itemName
}

// 调用时一口气传入3个入参
var itemFullName = generateName('大卖网', '母婴', '奶瓶')
```

偏函数应用改造：

```
function generateName(prefix) {
  return function(type, itemName) {
    return prefix + type + itemName
  }
}

// 把3个参数分两部分传入
var itemFullName = generateName('大卖网')('母婴', '奶瓶')
```

