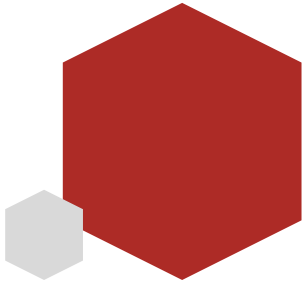
手写Promise

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌



函数柯里化

函数柯里化

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。

将多个参数的函数转换为单个参数函数

```
// 原函数
function sum(a, b) {
  return a + b
}
sum(1, 2) // 3

// 改写sum可以实现如下效果
sum(1)(2) // 3
```

```
// 改写结果
function sum(a) {

  return function (b) {
    return a + b
  }

}
```

```
// 思考题
function sum(a, b, c) {
  return a + b + c
}
// 改写为
sum(1)(2)(3)// 6
```

```
function sum(a) {
  return function (b) {
    return function (c) {
      return a + b + c
    }
  }
}
```

咱们会学习的有：

1. 经典面试题
2. 实际应用

函数柯里化-面试题

需求: 改写函数,实现如下效果

```
function sum(a, b, c, d, e) {
  return a + b + c + d + e
}
// 改写函数sum实现:参数传递到5个即可实现累加
// sum(1)(2)(3)(4)(5)
// sum(1)(2,3)(4)(5)
// sum(1)(2,3,4)(5)
// sum(1,2,3)(4,5)
```

思路:

函数柯里化-面试题-调优

需求: 参数的个数可以自定义

```
// 基于上一节代码,实现函数sumMaker
function sumMaker(){
  // ?
}
// 调用
const sum6 = sumMaker(6)
sum6(1, 2, 3)(4, 5, 6)
const sum4 = sumMaker(4)
sum6(1, 2)(3)(4)
```

思路:

复用上一节逻辑 → 传入参数个数

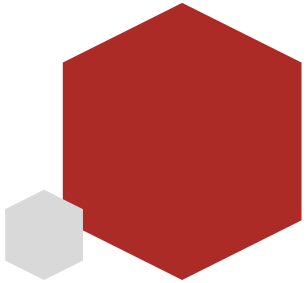
函数柯里化-实际应用

参数复用: 为函数预制通用参数,供多次重复调用

```
function isUndefined(thing) {
  return typeof thing === 'undefined'
}
function isNumber(thing) {
  return typeof thing === 'number'
}
function isString(thing) {
  return typeof thing === 'string'
}
function isFunction(thing) {
  return typeof thing === 'function'
}
```

```
// 改为通过 typeOfTest 生成:
const typeOfTest = function () {
  // 参数 和 逻辑略

}
const isUndefined = typeOfTest('undefined')
const isNumber = typeOfTest('number')
const isString = typeOfTest('string')
const isFunction = typeOfTest('function')
```

参考链接: axios源码



JavaScript设计模式

JavaScript设计模式

设计模式：在 面向对象软件 设计过程中针对特定问题的简洁而优雅的解决方案。

目前说到设计模式，一般指的是《设计模式：可复用面向对象软件的基础》一书中提到的 23种 常见的软件开发设计模式。

JavaScript设计模式

JavaScript中只需要了解常用的模式即可：

- 工厂模式
- 单例模式
- 观察者模式
- 发布订阅模式
- 原型模式
- 代理模式
- 迭代器模式

工厂模式

在JavaScript中,工厂模式的表现形式就是一个调用即可返回新对象的函数

```
// 工厂模式
function FoodFactory(name, color) {
  return {
    name,
    color
  }
}
const f1 = FoodFactory('西兰花', '黄绿色')
```

```
// 构造函数
function Food(name, color) {
  this.name = name
  this.color = color
}
const f3 = new Food('西兰花', '黄绿色')
```

1. Vue3-createApp
2. axios-create

工厂模式-Vue3-createApp

调整原因：官方文档

1. 避免在测试期间,全局配置污染其他测试用例
2. 全局改变Vue实例的行为,移到Vue实例上

2.x 全局 API	3.x 实例 API (app)
Vue.config	app.config
Vue.config.productionTip	移除 (见下方)
Vue.config.ignoredElements	app.config.compilerOptions.isCustomElement (见下方)
Vue.component	app.component
Vue.directive	app.directive
Vue.mixin	app.mixin
Vue.use	app.use (见下方)
Vue.prototype	app.config.globalProperties (见下方)
Vue.extend	移除 (见下方)

工厂模式

在JavaScript中,工厂模式的表现形式就是一个调用即可返回新对象的函数

```
// 工厂模式
function FoodFactory(name, color) {
  return {
    name,
    color
  }
}
const toy1 = FoodFactory('西兰花', '黄绿色')
const toy2 = FoodFactory('花菜', '白色')
```

```
// 构造函数
function Food(name, color) {
  this.name = name
  this.color = color
}
const f3 = new Food('西兰花', '黄绿色')
```

1. vue3-createApp
2. axios-create

工厂模式-axios-create

传送门: 官方文档

使用自定义配置新建一个实例

```
const instance = axios.create({
  baseURL: 'https://some-domain.com/api/',
  timeout: 1000,
  headers: { 'X-Custom-Header': 'foobar' }
})
```

工厂模式

在JavaScript中,工厂模式的表现形式就是一个调用即可返回新对象的函数

```
// 工厂模式
function FoodFactory(name, color) {
  return {
    name,
    color
  }
}
const toy1 = FoodFactory('西兰花', '黄绿色')
const toy2 = FoodFactory('花菜', '白色')
```

```
// 构造函数
function Food(name, color) {
  this.name = name
  this.color = color
}
const f3 = new Food('西兰花', '黄绿色')
```

1. vue3-createApp: 将全局改变Vue实例的行为,移到Vue实例上
2. axios-create:基于自定义配置新建实例

JavaScript设计模式

- 工厂模式
- 单例模式
- 观察者模式
- 发布订阅模式
- 原型模式
- 代理模式
- 迭代器模式

单例模式

单例模式：在使用这个模式时，单例对象整个系统需要保证 只有一个 存在。

1. 单例方法：
    1. 自己实现
    2. vant中的toast和notify组件

2. 单例的思想：
    1. vue2中的use方法
    2. vue3中的use方法

单例模式-方法

需求: 调用 方法 获取单例对象，重复调用获取的是相同对象


```
const s1 = SingleTon.getInstance()
const s2 = SingleTon.getInstance()
console.log(s1 === s2)//true
```

单例模式

单例模式：在使用这个模式时，单例对象整个系统需要保证 只有一个 存在。

1. 单例方法：
    1. 自己实现
    2. vant中的toast和notify组件

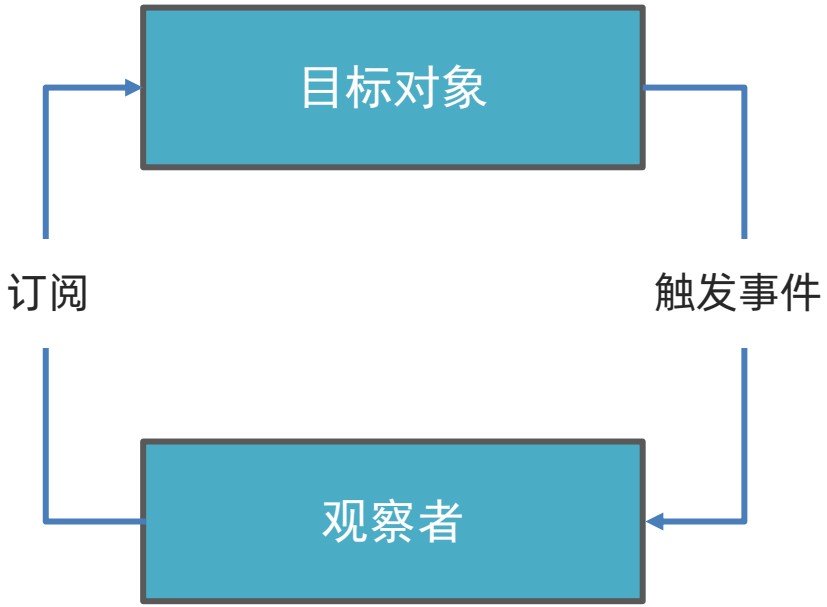
2. 单例的思想：
    1. vue2中的use方法
    2. vue3中的use方法

JavaScript设计模式

- 工厂模式
- 单例模式
- 观察者模式
- 发布订阅模式
- 原型模式
- 代理模式
- 迭代器模式

观察者模式

在对象之间定义一个 一对多 的依赖，当一个对象状态改变的时候，所有依赖的对象都会自动收到通知




```
window.addEventListener('load', () => console.log('load触发1'))
window.addEventListener('load', () => console.log('load触发2'))
window.addEventListener('load', () => console.log('load触发3'))
```

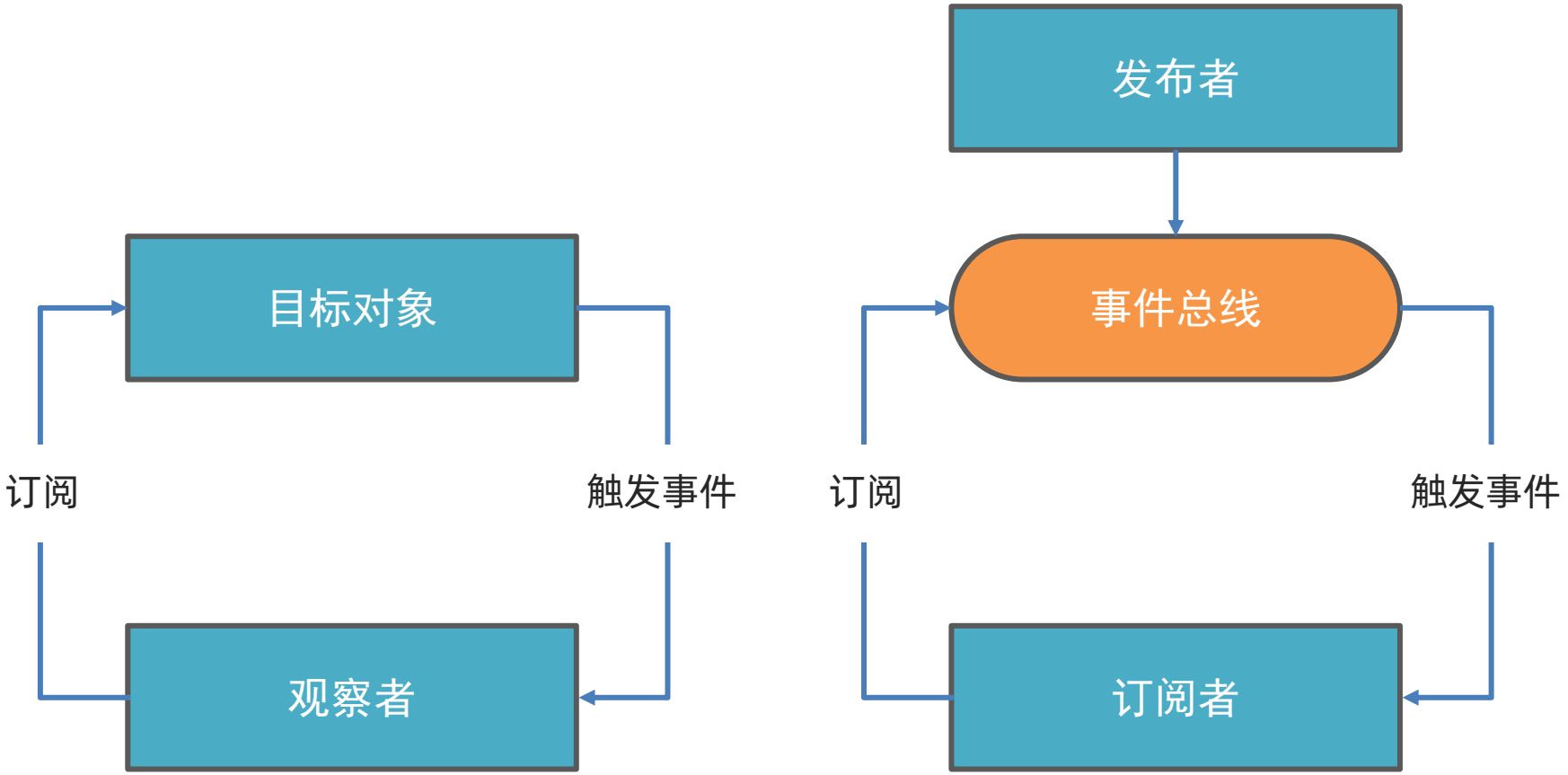
```
export default {
  data() {
    return {
      message: '',
    }
  },
  watch: {
    message(newMes, oldMes) {
      console.log('message-change')
    }
  },
}
```

JavaScript设计模式

- 工厂模式
- 单例模式
- 观察者模式
- 发布订阅模式
- 原型模式
- 代理模式
- 迭代器模式

发布订阅模式

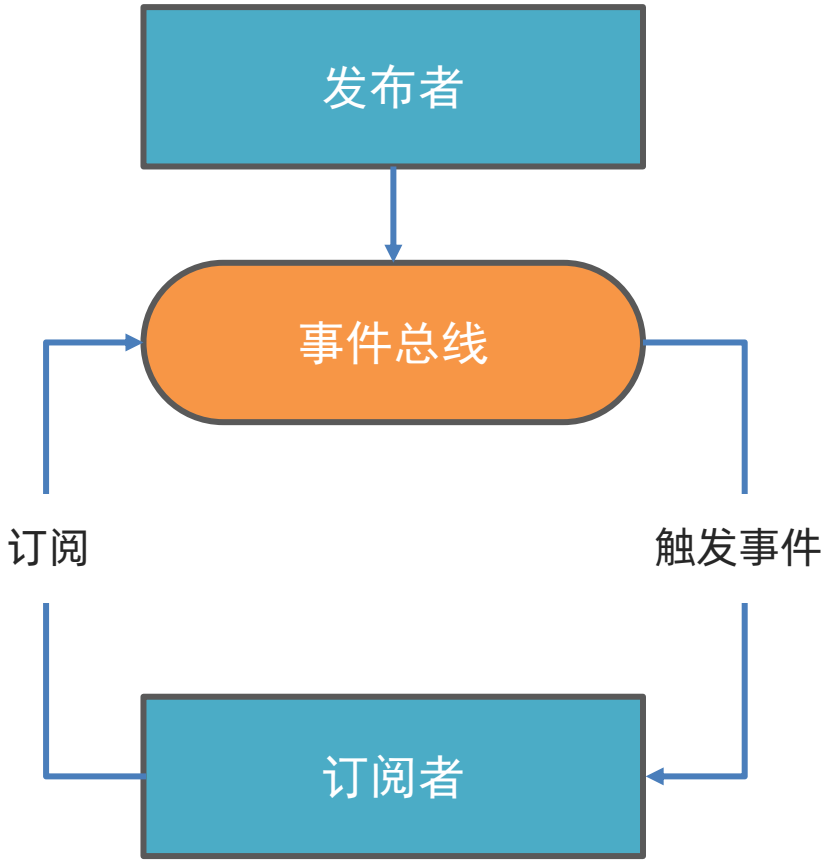
发布订阅模式和观察者模式类似,区别是:一个有中间商(发布订阅模式)一个没中间商(观察者模式)



发布订阅模式-应用场景

Vue中的EventBus

1. Vue2: 直接使用实例方法($on,$emit,$off,$once)
2. Vue3:使用替代方案



发布订阅模式-自己实现事件总线

实现HMEmitter,支持如下调用:

1. $on　添加私有属性 → 保存事件
2. $emit　接收不定长参数 → 循环触发事件
3. $off
4. $once

```
const bus = new HMEmitter()
// 注册事件
bus.$on('事件名1', 回调函数)
bus.$on('事件名1', 回调函数)

// 触发事件
bus.$emit('事件名', 参数1, ..., 参数n)

// 移除事件
bus.$off('事件名')

// 一次性事件
bus.$once('事件名', 回调函数)
```

```
{
  event1: [callback1,callback2],
  event2: [callback1]
}
```

发布订阅模式-自己实现事件总线

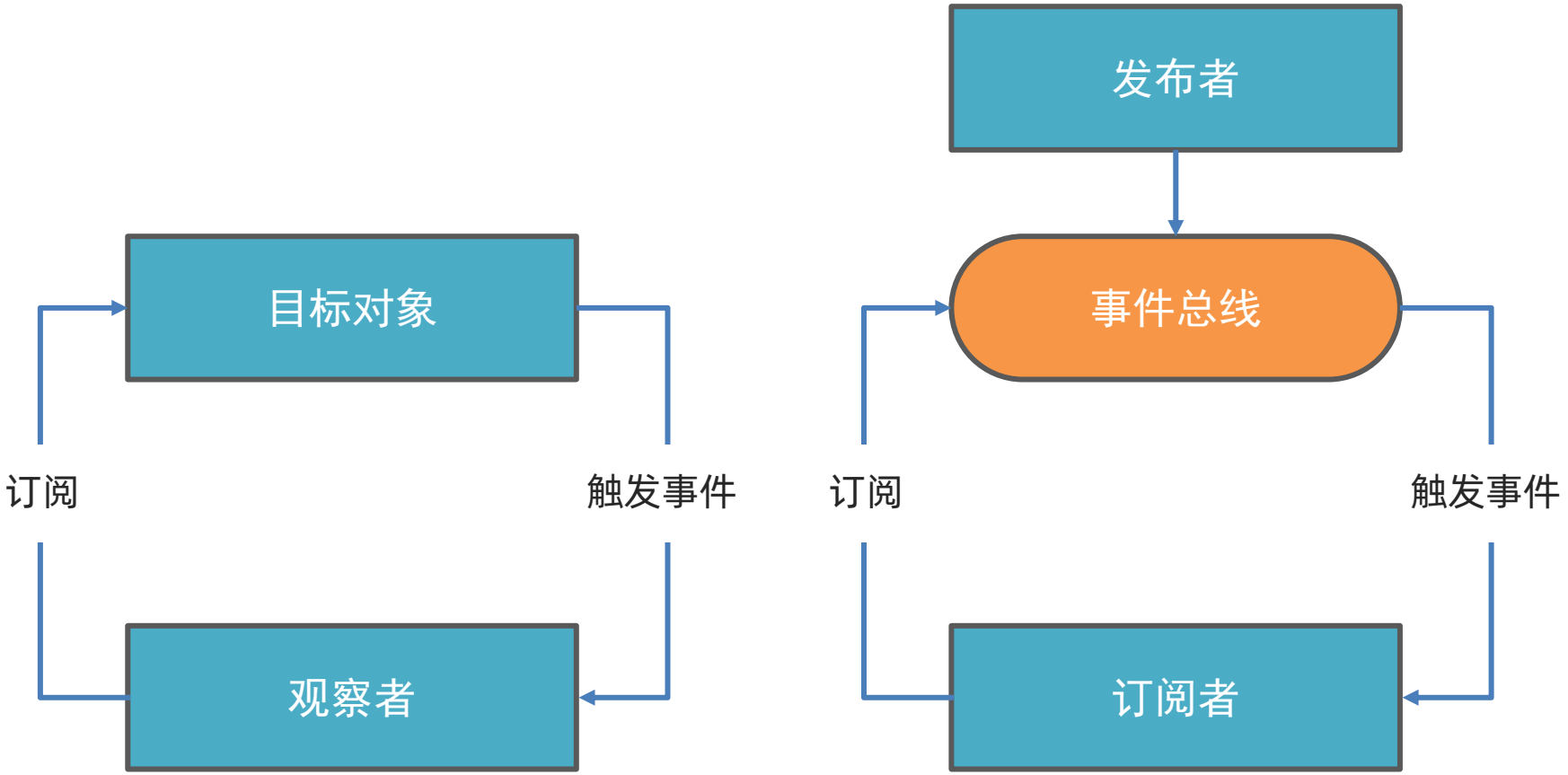
实现HMEmitter,支持如下调用:

1. $on
2. $emit
3. $off 清空事件
4. $once 调用$on注册事件 → 事件内调用$off

```
const bus = new HMEmitter()
// 注册事件
bus.$on('事件名1', 回调函数)
bus.$on('事件名1', 回调函数)

// 触发事件
bus.$emit('事件名', 参数1, ..., 参数n)

// 移除事件
bus.$off('事件名')

// 一次性事件
bus.$once('事件名', 回调函数)
```

发布订阅模式

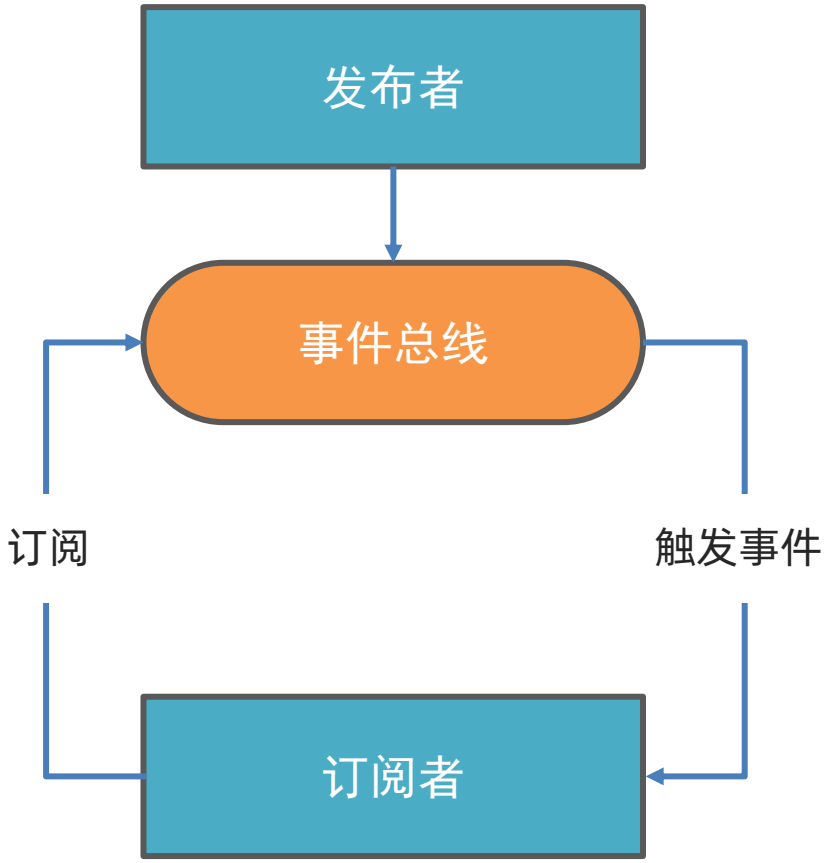
发布订阅模式和观察者模式类似,区别是:一个有中间商(发布订阅模式)一个没中间商(观察者模式)



发布订阅模式-应用场景

Vue中的EventBus

1. Vue2: 直接使用实例方法($on,$emit,$off,$once)
2. Vue3:使用替代方案



JavaScript设计模式

- 工厂模式
- 单例模式
- 观察者模式
- 发布订阅模式
- 原型模式
- 代理模式
- 迭代器模式

原型模式

原型模式是创建型模式的一种，其特点在于通过 复制 一个已经存在的实例来返回新的实例,而不是新建实例。

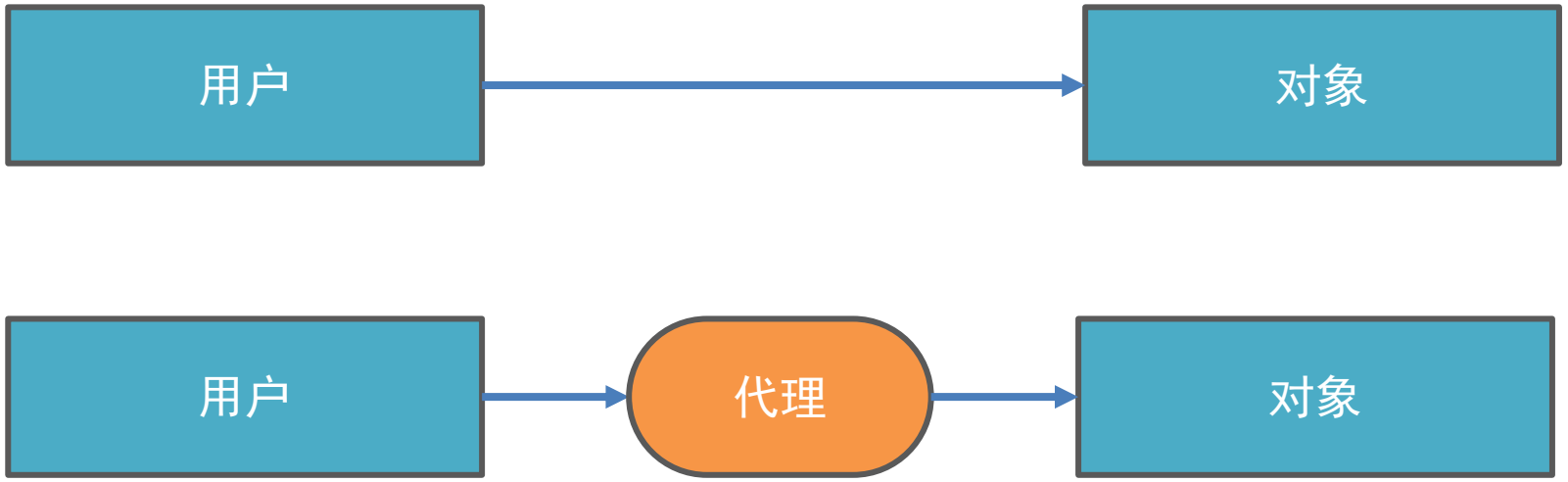
1. Object.create: 将对象作为原型,创建新对象
2. Vue2中的数组方法:
    1. Vue2源码
    2. 数组变更方法

JavaScript设计模式

- 工厂模式
- 单例模式
- 观察者模式
- 发布订阅模式
- 原型模式
- 代理模式
- 迭代器模式

代理模式

代理模式是为一个对象提供一个代用品或占位符，以便控制对它的访问



JavaScript设计模式

- 工厂模式
- 单例模式
- 观察者模式
- 发布订阅模式
- 原型模式
- 代理模式
- 迭代器模式

代理模式-缓存代理

需求:第一次查询的数据通过接口获取,重复查询通过缓存获取

JavaScript设计模式

- 工厂模式
- 单例模式
- 观察者模式
- 发布订阅模式
- 原型模式
- 代理模式
- 迭代器模式

迭代器模式

可以让用户透过特定的接口巡访容器中的每一个元素而不用了解底层的实现(遍历)

1. for in 和 for of
2. 迭代协议

传智教育旗下高端IT教育品牌