# 25 中介者模式：找媒人介绍对象

更新时间：2019-07-24 14:18:07

> 合理安排时间，就等于节约时间。
>
> ——培根

**中介者模式** （Mediator Pattern）又称调停模式，使得各对象不用显式地相互引用，将对象与对象之间紧密的耦合关系变得松散，从而可以独立地改变他们。核心是**多个对象之间复杂交互的封装**。

根据最少知识原则，一个对象应该尽量少地了解其他对象。如果对象之间耦合性太高，改动一个对象则会影响到很多其他对象，可维护性差。复杂的系统，对象之间的耦合关系会得更加复杂，中介者模式就是为了解决这个问题而诞生的。

> **注意：** 本文可能用到一些 ES6 的语法 let/const、箭头函数、解构赋值、Rest 参数 等，如果还没接触过可以点击链接稍加学习 ~

## 1. 你曾见过的中介者模式

举一个有点意思的例子。相亲是一个多方博弈、互相选择（嫌弃）的场景，男生和女生相亲，不仅仅是男生和女生两方关系，还有：

1. 男方的角度：

   - 男生会考虑女生的长相、身高、性格、三观、家境、和自己父母能不能合得来、对方家长是否好相处等等；
   - 男生的家长会考虑女生的条件、和自己儿子搭不搭、对方家长的性格等等；

2. 女方的角度：

   - 女生会考虑男生的性格、上进心、是不是高富帅、能不能通过自己父母的法眼、对方家长是否好相处等等；
   - 女生的家长也会考虑男生的条件、是不是配得上自己女儿、对方家长的性格等等；

双方的家长可能还有博弈和交换，比如你家是公务员，那么可以稍微穷一点；我家比较帅，那么矮一点对方也可以接受...总之，男生、女生、男方家长、女方家长各方的关系交错复杂，每个人都有自己的考量，如果某一方有什么想法，要和其他三方进行沟通，牵一发动全身。这时候我们可以引入媒人，无论哪一方有什么要求或者什么想法，都可以直接告诉媒人，这样就不用各方自己互相沟通了。

再看另一个例子。比如买房子的时候，我们不必自己去跑到每个卖家那里了解情况，而一般选择从中介那里获取房源信息。卖家们把各自的房源信息提供给中介，包括房源的大小、楼层、朝向等，有的卖家说价格还可以谈，有的卖家说我的房子带阁楼，有的卖家说要跟车库一起卖，等等。买家从中介处就可以获取自己所需的房源信息，比如你不需要车库，也不住一楼和顶楼，只考虑一百平米以上的屋子，中介就会给你筛选出满足你需要的所有房源供你查看，而不需买家一个个的找卖家们了解信息，当你正关注的卖家房子卖出去了中介也会及时告诉你，这就是中介的作用。

类似的例子还有很多，比如电商平台之于买家与店家，聊天平台之于每个聊天者，澳门大型线上赌场之于每个参与赌博的人....

在类似的场景中，有以下特点：

1. 相亲各方/房源买家卖家（目标对象）之间的关系复杂，引入媒人/中介（中介者）会极大方便各方之间的沟通；
2. 相亲各方/房源买家卖家（目标对象）之间如果有什么想法和要求上的变动，通过媒人/中介（中介者）就可以及时通知到相关各方，而目标对象之间相互不通信；

## 2. 实例的代码实现

我们使用 JavaScript 将刚刚的相亲例子实现一下。

首先我们考虑一个场景，男方和女方都有一定的条件，双方之间有要求，双方家长对对方孩子也有要求，如果达不到要求则不同意这门婚事。（也就是说暂时不考虑男女双方对于对方家长，和双方家长之间的要求，因为这样代码就太长了）

```
class Person {
  /* 个人信息 */
  constructor(name, info, target) {
    this.name = name
    this.info = info      // 是一个对象，每一项为数字，比如身高、工资..
    this.target = target  // 也是对象，每一项为两个数字的数组，表示可接受的最低和最高值
    this.enemyList = []   // 考虑列表
  }

  /* 注册相亲对象及家长 */
  registEnemy(...enemy) {
    this.enemyList.push(...enemy)
  }

  /* 检查所有相亲对象及其家长的条件 */
  checkAllPurpose() {
    this.enemyList.forEach(enemy => enemy.info && this.checkPurpose(enemy))
  }

  /* 检查对方是否满足自己的要求，并发信息 */
  checkPurpose(enemy) {
    const result = Object.keys(this.target)   // 是否满足自己的要求
      .every(key => {
        const [low, high] = this.target[key]
        return low <= enemy.info[key] && enemy.info[key] <= high
      })
    enemy.receiveResult(result, this, enemy)  // 通知对方
  }

  /* 接受到对方的信息 */
```

```
    receiveResult(result, they, me) {
        result
          ? console.log(`${ they.name }：我觉得合适~ \t（我的要求 ${ me.name } 已经满足）`)
          : console.log(`${ they.name }：你是个好人! \t（我的要求 ${ me.name } 不能满足！）`)
    }
}

/* 男方 */
const ZhangXiaoShuai = new Person(
  '张小帅',
  { age: 25, height: 171, salary: 5000 },
  { age: [23, 27] })

/* 男方家长 */
const ZhangXiaoShuaiParent = new Person(
  '张小帅家长',
  null,
  { height: [160, 167] })

/* 女方 */
const LiXiaoMei = new Person(
  '李小美',
  { age: 23, height: 160 },
  { age: [25, 27] })

/* 女方家长 */
const LiXiaoMeiParent = new Person(
  '李小美家长',
  null,
  { salary: [10000, 20000] })

/* 注册，每一个 person 实例都需要注册对方家庭成员的信息 */
ZhangXiaoShuai.registEnemy(LiXiaoMei, LiXiaoMeiParent)
LiXiaoMei.registEnemy(ZhangXiaoShuai, ZhangXiaoShuaiParent)
ZhangXiaoShuaiParent.registEnemy(LiXiaoMei, LiXiaoMeiParent)
LiXiaoMeiParent.registEnemy(ZhangXiaoShuai, ZhangXiaoShuaiParent)

/* 检查对方是否符合要求，同样，每一个 person 实例都需要执行检查 */
ZhangXiaoShuai.checkAllPurpose()
LiXiaoMei.checkAllPurpose()
ZhangXiaoShuaiParent.checkAllPurpose()
LiXiaoMeiParent.checkAllPurpose()

// 张小帅：我觉得合适~ （我的要求 李小美 已经满足）
// 李小美：我觉得合适~ （我的要求 张小帅 已经满足）
// 张小帅家长：我觉得合适~ （我的要求 李小美 已经满足）
// 李小美家长：你是个好人! （我的要求 张小帅 不能满足！）
```

当然作为灵活的 JavaScript，并不一定需要使用类，使用对象的形式也是可以的：

```
const PersonFunc = {
    /* 注册相亲对象及家长 */
    registEnemy(...enemy) {
        this.enemyList.push(...enemy)
    },

    /* 检查所有相亲对象及其家长的条件 */
    checkAllPurpose() {
        this.enemyList.forEach(enemy => enemy.info && this.checkPurpose(enemy))
    },

    /* 检查对方是否满足自己的要求，并发信息 */
    checkPurpose(enemy) {
        const result = Object.keys(this.target)    // 是否满足自己的要求
          .every(key => {
              const [low, high] = this.target[key]
              return low <= enemy.info[key] && enemy.info[key] <= high
          })
        enemy.receiveResult(result, this, enemy)   // 通知对方
    },

    /* 接受到对方的信息 */
```

```
  receiveResult(result, they, me) {
    result
      ? console.log(`${ they.name }：我觉得合适~ \t（我的要求 ${ me.name } 已经满足）`)
      : console.log(`${ they.name }：你是个好人! \t（我的要求 ${ me.name } 不能满足！）`)
  }
}

/* 男方 */
const ZhangXiaoShuai = {
  ...PersonFunc,
  name: '张小帅',
  info: { age: 25, height: 171, salary: 5000 },
  target: { age: [23, 27] },
  enemyList: []
}

/* 男方家长 */
const ZhangXiaoShuaiParent = {
  ...PersonFunc,
  name: '张小帅家长',
  info: null,
  target: { height: [160, 167] },
  enemyList: []
}

/* 女方 */
const LiXiaoMei = {
  ...PersonFunc,
  name: '李小美',
  info: { age: 23, height: 160 },
  target: { age: [25, 27] },
  enemyList: []
}

/* 女方家长 */
const LiXiaoMeiParent = {
  ...PersonFunc,
  name: '李小美家长',
  info: null,
  target: { salary: [10000, 20000] },
  enemyList: []
}

/* 注册 */
ZhangXiaoShuai.registEnemy(LiXiaoMei, LiXiaoMeiParent)
LiXiaoMei.registEnemy(ZhangXiaoShuai, ZhangXiaoShuaiParent)
ZhangXiaoShuaiParent.registEnemy(LiXiaoMei, LiXiaoMeiParent)
LiXiaoMeiParent.registEnemy(ZhangXiaoShuai, ZhangXiaoShuaiParent)

/* 检查对方是否符合要求 */
ZhangXiaoShuai.checkAllPurpose()
LiXiaoMei.checkAllPurpose()
ZhangXiaoShuaiParent.checkAllPurpose()
LiXiaoMeiParent.checkAllPurpose()

// 张小帅：我觉得合适~ （我的要求 李小美 已经满足）
// 李小美：我觉得合适~ （我的要求 张小帅 已经满足）
// 张小帅家长：我觉得合适~ （我的要求 李小美 已经满足）
// 李小美家长：你是个好人！（我的要求 张小帅 不能满足！）
```

我们还可以使用 `Object.create()` 赋值原型的方式将方法放在原型上，也可以使用原型继承的方式，JavaScript 的灵活性让你可以自由选择习惯的方式。

单就结果而言，上面的代码可以实现整个逻辑。**但是这几个对象之间相互引用、相互持有，并紧密耦合**。如果继续引入关系，比如张小帅的七大姑、李小美的八大姨，或者考虑的情况更多一些，那么就要改动很多代码，上面的写法就满足不了要求了。

这时我们可以引入媒人（中介者），专门处理对象之间的耦合关系，所有对象间相互不了解，只与媒人交互，如果引入了新的相关方，也只需要通知媒人即可。看一下实现：

```
/* 男方 */
const ZhangXiaoShuai = {
  name: '张小帅',
  family: '张小帅家',
  info: { age: 25, height: 171, salary: 5000 },
  target: { age: [23, 27] }
}

/* 男方家长 */
const ZhangXiaoShuaiParent = {
  name: '张小帅家长',
  family: '张小帅家',
  info: null,
  target: { height: [160, 167] }
}

/* 女方 */
const LiXiaoMei = {
  name: '李小美',
  family: '李小美家',
  info: { age: 23, height: 160 },
  target: { age: [25, 27] }
}

/* 女方家长 */
const LiXiaoMeiParent = {
  name: '李小美家长',
  family: '李小美家',
  info: null,
  target: { salary: [10000, 20000] }
}

/* 媒人 */
const MatchMaker = {
  matchBook: {},  // 媒人的花名册

  /* 注册各方 */
  registPersons(...personList) {
    personList.forEach(person => {
      if (this.matchBook[person.family]) {
        this.matchBook[person.family].push(person)
      } else this.matchBook[person.family] = [person]
    })
  },

  /* 检查对方家庭的孩子对象是否满足要求 */
  checkAllPurpose() {
    Object.keys(this.matchBook)   // 遍历名册中所有家庭
      .forEach((familyName, idx, matchList) =>
        matchList
          .filter(match => match !== familyName)  // 对于其中一个家庭，过滤出名册中其他的家庭
          .forEach(enemyFamily => this.matchBook[enemyFamily]  // 遍历该家庭中注册到名册上的所有成员
            .forEach(enemy => this.matchBook[familyName]
              .forEach(person =>            // 逐项比较自己的条件和他们的要求
                enemy.info && this.checkPurpose(person, enemy)
              )
            ))
      )
  },

  /* 检查对方是否满足自己的要求，并发信息 */
  checkPurpose(person, enemy) {
    const result = Object.keys(person.target)   // 是否满足自己的要求
      .every(key => {
        const [low, high] = person.target[key]
        return low <= enemy.info[key] && enemy.info[key] <= high
      })
    this.receiveResult(result, person, enemy)   // 通知对方
  },
```

```
  },

  /* 通知对方信息 */
  receiveResult(result, person, enemy) {
    result
      ? console.log(`${ person.name } 觉得合适~ \t（${ enemy.name } 已经满足要求）`)
      : console.log(`${ person.name } 觉得不合适! \t（${ enemy.name } 不能满足要求！）`)
  }
}


/* 注册 */
MatchMaker.registPersons(ZhangXiaoShuai, ZhangXiaoShuaiParent, LiXiaoMei, LiXiaoMeiParent)

MatchMaker.checkAllPurpose()

// 输出: 小帅 觉得合适~    （李小美 已经满足要求）
// 输出: 张小帅家长 觉得合适~ （李小美 已经满足要求）
// 输出: 李小美 觉得合适~    （张小帅 已经满足要求）
// 输出: 李小美家长 觉得不合适! （张小帅 不能满足要求！）
```

可以看到，除了媒人之外，其他各个角色都是独立的，相互不知道对方的存在，对象间关系被解耦，我们甚至可以方便地添加新的对象。比如赵小美家同时还在考虑着孙小拽（emmm...）：

```
// 重写上面「注册」之后的代码

/* 引入孙小拽 */
const SunXiaoZhuai = {
  name: '孙小拽',
  familyType: '男方',
  info: { age: 27, height: 173, salary: 20000 },
  target: { age: [23, 27] }
}

/* 孙小拽家长 */
const SunXiaoZhuaiParent = {
  name: '孙小拽家长',
  familyType: '男方',
  info: null,
  target: { height: [160, 170] }
}

/* 注册，这里只需要注册一次 */
MatchMaker.registPersons(ZhangXiaoShuai,
                         ZhangXiaoShuaiParent,
                         LiXiaoMei,
                         LiXiaoMeiParent,
                         SunXiaoZhuai,
                         SunXiaoZhuaiParent)

/* 检查对方是否符合要求，也只需要检查一次 */
MatchMaker.checkAllPurpose()

// 输出: 张小帅 觉得合适~    （李小美 已经满足要求）
// 输出: 张小帅家长 觉得合适~ （李小美 已经满足要求）
// 输出: 孙小拽 觉得合适~    （李小美 已经满足要求）
// 输出: 孙小拽家长 觉得合适~ （李小美 已经满足要求）
// 输出: 李小美 觉得合适~    （张小帅 已经满足要求）
// 输出: 李小美家长 觉得不合适! （张小帅 不能满足要求！）
// 输出: 李小美 觉得合适~    （孙小拽 已经满足要求）
// 输出: 李小美家长 觉得合适~ （孙小拽 已经满足要求）
```

从这个例子就已经可以看出中介者模式的优点了，因为各对象之间的相互引用关系被解耦，从而令系统的可扩展性、可维护性更好。

## 3. 中介者模式的通用实现

对于上面的例子，张小帅、李小美、孙小拽和他们的家长们相当于容易产生耦合的对象（最早的一本设计模式书上将这些对象称为同事，这里也借用一下这个称呼，Colleague），而媒人就相当于中介者（Mediator）。在中介者模式中，同事对象之间互相不通信，而只与中介者通信，同事对象只需知道中介者即可。主要有以下几个概念：

1. **Colleague**： 同事对象，只知道中介者而不知道其他同事对象，通过中介者来与其他同事对象通信；
2. **Mediator**： 中介者，负责与各同事对象的通信；

结构图如下：

可以看到上图，使用中介者模式之后同事对象间的网状结构变成了星型结构，同事对象之间不需要知道彼此，符合最少知识原则。如果同事对象之间需要相互通信，只能通过中介者的方式，这样让同事对象之间原本的强耦合变成弱耦合，强相互依赖变成弱相互依赖，从而让这些同事对象可以独立地改变和复用。原本同事对象间的交互逻辑被中介者封装起来，各个同事对象只需关心自身即可。

## 4. 中介者模式的优缺点

中介者模式的主要优点有：

1. **松散耦合**，降低了同事对象之间的相互依赖和耦合，不会像之前那样牵一发动全身；
2. 将同事对象间的**一对多关联转变为一对一的关联**，符合最少知识原则，提高系统的灵活性，使得系统易于维护

和扩展；
3. 中介者在同事对象间起到了控制和协调的作用，因此可以结合代理模式那样，进行**同事对象间的访问控制、功能扩展**；
4. 因为同事对象间不需要相互引用，因此也可以**简化同事对象的设计和实现**；

主要缺点是：**逻辑过度集中化**，当同事对象太多时，中介者的职责将很重，逻辑变得复杂而庞大，以至于难以维护。

当出现中介者可维护性变差的情况时，考虑是否在系统设计上不合理，从而简化系统设计，优化并重构，避免中介者出现职责过重的情况。

## 5. 中介者模式的适用场景

中介者模式适用多个对象间的关系确实已经紧密耦合，且导致扩展、维护产生了困难的场景，也就是当多个**对象之间的引用关系变成了网状结构的时候**，此时可以考虑使用引入中介者来把**网状结构转化为星型结构**。

但是，如果对象之间的关系耦合并不紧密，或者之间的关系本就一目了然，那么引入中介者模式就是多此一举、画蛇添足。

实际上，我们通常使用的 MVC/MVVM 框架，就含有中介者模式的思想，Controller/ViewModel 层作为中介者协调 View/Model 进行工作，减少 View/Model 之间的直接耦合依赖，从而做到视图层和数据层的最大分离。可以关注后面有单独一章分析 MVC/MVVM 模式，深入了解。

## 6. 其他相关模式

### 6.1 中介者模式和外观模式

外观模式和中介者模式思想上有一些相似的地方，但也有不同：

1. **中介者模式** 将多个平等对象之间**内部**的复杂交互关系封装起来，主要目的是为了多个对象之间的解耦；
2. **外观模式** 封装一个子系统内部的模块，是为了向系统**外部**提供方便的调用；

### 6.2 中介者模式与发布-订阅模式

中介者模式和发布-订阅模式都可以用来进行对象间的解耦，比如发布-订阅模式的发布者/订阅者和中介者模式里面的中介者/同事对象功能上就比较类似。

这两个模式也可以组合使用，比如中介者模式就可以使用发布-订阅模式，对相关同事对象进行消息的广播通知。

比如上面相亲的例子中，注册各方和通知信息就使用了发布-订阅模式。

### 6.3 中介者模式与代理模式

同事对象之间需要通信的时候，需要经由中介者，这时中介者就相当于同事对象间的代理。所以这时就可以引入代理模式的概念，对同事对象相互访问的时候，起到访问控制、功能扩展等等功能。

---

推介阅读：

1. Array.prototype.forEach() - JavaScript | MDN
2. Array.prototype.every() - JavaScript | MDN
3. Array.prototype.filter() - JavaScript | MDN
4. JS 中可以提升幸福度的小技巧 - 短路运算符

```
}
```

24 职责链模式：领导，我想请个假

26 MVC、MVP、MVVM