《前端高手进阶》

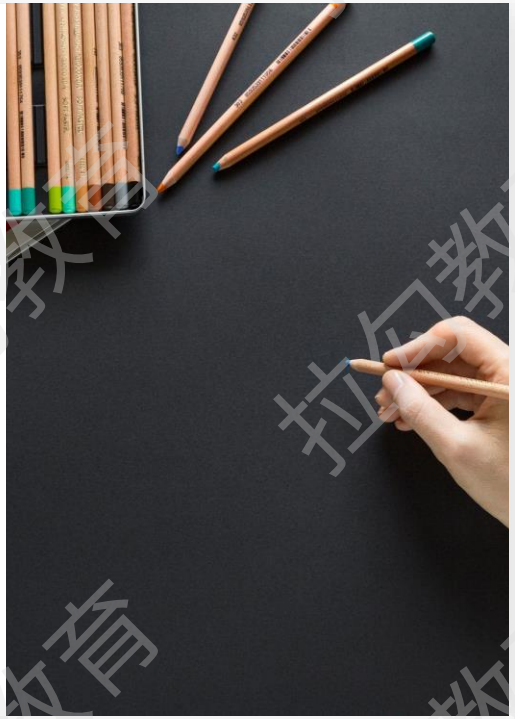
朱德龙 前中兴软创主任工程师

— 拉勾教育出品 —

第29讲：框架到底用了哪些设计模式？

设计模式

设计模式（Design Pattern）是对软件设计中普遍存在（反复出现）的各种问题所提出的解决方案。设计模式并不直接用来完成代码的编写，而是描述在各种不同情况下，要怎么解决问题的一种方案



引言

拉勾教育
互联网人实战大学

设计模式（Design Pattern）

设计模式就是一套抽象的理论

对于理论最好的学习方式

就是通过与实践结合来加深理解

设计模式原则

针对面向对象编程范式总结出来的解决方案
设计模式的原则都是围绕“类”和“接口”
这两个概念来提出的

开闭原则

指的就是对扩展开放、对修改关闭

遵循开闭原则就意味着当代码需要修改时

可以通过编写新的代码来扩展已有的代码

而不是直接修改已有代码本身

开闭原则


```
function validate() {
  // 校验用户名
  if (!username) {
    ...
  } else {
    ...
  }
  // 校验密码
  if (!pswd){
    ...
  } else {
```

开闭原则


```
    ...
  } else {
    ...
  }
  // 校验验证码
  if (!captcha) {
    ...
  } else {
    ...
  }
}
```

开闭原则


```
class Validation {
  private validateHandlers: ValidateHandler[] = [];

  public addValidateHandler(handler: IValidateHandler) {
    this.validateHandlers.push(handler)
  }

  public validate() {
    for (let i = 0; i < this.validateHandlers.length; i++) {
      this.validateHandlers[i].validate();
    }
  }
```

开闭原则


```
  }
}

interface IValidateHandler {
  validate(): boolean;
}
class UsernameValidateHandler implements IValidateHandler {
  public validate() {
    ...
  }
}
class PwdValidateHandler implements IValidateHandler {
```

开闭原则


```
    }
}
class PwdValidateHandler implements IValidateHandler {
    public validate() {
        ...
    }
}
class CaptchaValidateHandler implements IValidateHandler {
    public validate() {
        ...
    }
}
```

里氏替换原则

指在使用父类的地方可以用它的任意子类进行替换

要求子类可以随时替换掉其父类

同时功能不被破坏

父类的方法仍然能被使用

里氏替换原则


```
class Bird {
  getFood(){
    return '虫子'
  }
}


class Sparrow extends Bird {
  getFood() {
    return ['虫子', '稻谷']
  }
}
```

里氏替换原则


```
interface IBird {
  getFood(): string[]
}
class Bird implements IBird{
  getFood() {
    return ['虫子']
  }
}
```

里氏替换原则


```
  }
}

class Sparrow implements IBird {
  getFood() {
    return ['虫子', '稻谷']
  }
}
```

依赖倒置原则

好的依赖关系应该是类依赖于抽象接口

不应依赖于具体实现

依赖倒置原则


```
class Bike{
  run() {
    console.log('Bike run')
  }
}

class Passenger {
  construct(Bike: bike) {
```

依赖倒置原则


```
class Passenger {
  construct(Bike: bike) {
    this.tool = bike
  }
  public start() {
    this.tool.run()
  }
}
```

依赖倒置原则


```
interface ITransportation {
 run(): void
}

class Bike implements ITransportation {
 run() {
  console.log('Bike run')
 }
}

class Car implements ITransportation {
 run() {
```

依赖倒置原则


```
    console.log('Car run')
  }
}


class Passenger {
  construct(ITransportation : transportation) {
    this.tool = transportation
  }
  public start() {
    this.tool.run()
  }
}
```

接口隔离原则

不应该依赖它不需要的接口

一个类对另一个类的依赖应该建立在最小的接口上

目的就是为了降低代码之间的耦合性

接口隔离原则


```
interface IAnimal {
  eat(): void
  swim(): void
}

class Dog implements IAnimal {
  eat() {
    ...
  }
  swim() {
    ...
```

接口隔离原则


```
  }
}

class Bird implements IAnimal {
  eat() {
    ...
  }
  swim() {
    // do nothing
  }
}
```

迪米特原则

一个类对于其他类知道得越少越好

就是说一个对象应当对其他对象尽可能少的了解

迪米特原则


```
class Store {
 set(key, value) {
  window.localStorage.setItem(key, value)
 }
}
```

迪米特原则


```
class Store {
 construct(s) {
  this._store = s
 }
 set(key, value) {
  this._store.setItem(key, value)
 }
}
new Store(window.localstorage)
```

单一职责原则

应该有且仅有一个原因引起类的变更

单一职责原则


```
class Util {
  static toTime(date) {
    ...
  }
  static formatString(str) {
    ...
  }
  static encode(str) {
    ...
  }
}
```

创建型

核心特点就是将对象的创建与使用进行分离

从而降低系统的耦合度

创建型

抽象工厂模式

生成器模式

单例模式

工厂方法模式

原型模式

创建型


```
// src/core/global-api/use.js
export function initUse (Vue: GlobalAPI) {
  Vue.use = function (plugin: Function | Object) {
    const installedPlugins = (this._installedPlugins ||
(this._installedPlugins = []))
    if (installedPlugins.indexOf(plugin) > -1) {
      return this
    }

    .....
  }
}
```

创建型


```
// src/core/vdom/vnode.js
export function cloneVNode (vnode: VNode): VNode {
  const cloned = new VNode(
    vnode.tag,
    vnode.data,
    // #7975
    // clone children array to avoid mutating original in case of cloning
    // a child.
    vnode.children && vnode.children.slice(),
    vnode.text,
    vnode.elm,
    vnode.context,
```

创建型


```
    vnode.componentOptions,
    vnode.asyncFactory
  )
  cloned.ns = vnode.ns
  cloned.isStatic = vnode.isStatic
  cloned.key = vnode.key
  cloned.isComment = vnode.isComment
  cloned.fnContext = vnode.fnContext
  cloned.fnOptions = vnode.fnOptions
  cloned.fnScopeId = vnode.fnScopeId
  cloned.asyncMeta = vnode.asyncMeta
  cloned.isCloned = true
```

创建型


```
  )
  cloned.ns = vnode.ns
  cloned.isStatic = vnode.isStatic
  cloned.key = vnode.key
  cloned.isComment = vnode.isComment
  cloned.fnContext = vnode.fnContext
  cloned.fnOptions = vnode.fnOptions
  cloned.fnScopeId = vnode.fnScopeId
  cloned.asyncMeta = vnode.asyncMeta
  cloned.isCloned = true
  return cloned
}
```

结构型

描述如何将类或对象组合在一起形成更大的结构

类结构型模式采用继承机制来组织接口和类

对象结构型模式采用组合或聚合来生成新的对象

结构型

- 适配器模式
- 桥接模式
- 组合模式
- 装饰器模式
- 外观模式
- 享元模式
- 代理模式

结构型


```
// src/core/instance/proxy.js
initProxy = function initProxy (vm) {
 if (hasProxy) {
  // determine which proxy handler to use
  const options = vm.$options
  const handlers = options.render && options.render._withStripped
   ? getHandler
   : hasHandler
  vm._renderProxy = new Proxy(vm, handlers)
 } else {
  vm._renderProxy = vm
 }
}
```

结构型


```
// src/core/observer/dep.js
export default class Dep {
  static target: ?Watcher;
  id: number;
  subs: Array<Watcher>;


  constructor () {
    this.id = uid++
    this.subs = []
  }
```

结构型


```
addSub (sub: Watcher) {
 this.subs.push(sub)
}


removeSub (sub: Watcher) {
 remove(this.subs, sub)
}


depend () {
 if (Dep.target) {
```

结构型


```
    Dep.target.addDep(this)
  }
}


notify () {
  // stabilize the subscriber list first
  const subs = this.subs.slice()
  if (process.env.NODE_ENV !== 'production' && !config.async) {
    // subs aren't sorted in scheduler if not running async
    // we need to sort them now to make sure they fire in correct
    // order
```

结构型


```
const subs = this.subs.slice()
if (process.env.NODE_ENV !== 'production' && !config.async) {
  // subs aren't sorted in scheduler if not running async
  // we need to sort them now to make sure they fire in correct
  // order
  subs.sort((a, b) => a.id - b.id)
}
for (let i = 0, l = subs.length; i < l; i++) {
  subs[i].update()
 }
 }
}
```

行为型

用于描述程序在运行时复杂的流程控制

即描述多个类或对象之间怎样相互协作

共同完成单个对象无法单独完成的任务

它涉及算法与对象间职责的分配

行为型

责任链模式

命令模式

迭代器模式

策略模式

解释器模式

行为型

- 中介者模式
- 状态模式
- 备忘录模式
- 访问者模式
- 观察者模式
- 模板方法模式

行为型


```
// src/core/instance/render.js
export function initRender (vm: Component) {
  vm._vnode = null // the root of the child tree
  vm._staticTrees = null // v-once cached trees
  const options = vm.$options
  const parentVnode = vm.$vnode = options._parentVnode // the
placeholder node in parent tree
  const renderContext = parentVnode && parentVnode.context
  vm.$slots = resolveSlots(options._renderChildren, renderContext)
  vm.$scopedSlots = emptyObject
  // bind the createElement fn to this instance
```

行为型


```
const parentVnode = vm.$vnode = options._parentVnode // the placeholder node in parent tree
const renderContext = parentVnode && parentVnode.context
vm.$slots = resolveSlots(options._renderChildren, renderContext)
vm.$scopedSlots = emptyObject
// bind the createElement fn to this instance
// so that we get proper render context inside it.
// args order: tag, data, children, normalizationType, alwaysNormalize
// internal version is used by render functions compiled from templates
.....
}
```

行为型

```
// src/core/observer/index.js
Object.defineProperty(obj, key, {
  enumerable: true,
  configurable: true,
  get: function reactiveGetter () {
    ......
  },
  set: function reactiveSetter (newVal) {
    const value = getter ? getter.call(obj) : val
    /* eslint-disable no-self-compare */
    if (newVal === value || (newVal !== newVal && value !== value)) {
      return
```

行为型


```
}
/* eslint-enable no-self-compare */
if (process.env.NODE_ENV !== 'production' && customSetter) {
  customSetter()
}
// #7981: for accessor properties without setter
if (getter && !setter) return
if (setter) {
  setter.call(obj, newVal)
} else {
  val = newVal
}
```

行为型


```
  }
  // #7981: for accessor properties without setter
  if (getter && !setter) return
  if (setter) {
    setter.call(obj, newVal)
  } else {
    val = newVal
  }
  childOb = !shallow && observe(newVal)
  dep.notify()
 }
})
```

总结

介绍了设计模式的 6 个重要原则

重点讨论了接口和类的使用方式

然后介绍了 3 类设计模式以及对应的例子

总结

你还在框架代码中找到过哪些设计模式的应用？

Next：第30讲：《前端热点技术之 Serverless》

拉勾教育

—互联网人实战大学—

下载「拉勾教育App」
获取更多内容

L / A / G / O / U