link: null
title: 珠峰架构师成长计划
description: 过度使用继承或者说继承层次过深会导致代码可读性、可维护性变差
子类和父类高度耦合，修改父类的代码，会直接影响到子类
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=547 sentences=1273, words=9036

1. 什么是面向对象

- 以类和对象作为组织代码的基本单位，并实现封装、抽象、继承、多态四个特性
- 软件开发经历分析、设计和编码三个阶段
    - 面向对象分析(OOA) Object Oriented Analysis
    - 面向对象设计(OOD) Object Oriented Design
    - 面向对象编程(OOP)Object Oriented Programming

1.1 抽象(Abstraction)

- 抽象主要是隐藏方法的实现，让调用者只关心有哪些功能而不是关心功能的实现
- 抽象可以提高代码的可扩展性和维护性，修改实现不需要改变定义，可以减少代码的改动范围

```
interface IStorage {
    save(key: any, value: any): void;
    read(key: any): void;
}
class LocalStorage implements IStorage {
    save(key: any, value: any) {
        localStorage.setItem(key, value);
    }
    read(key: any) {
        return localStorage.getItem(key);
    }
}
class User {
    constructor(public name: string, public storage: IStorage) {

    }
    save() {
        this.storage.save('userInfo', JSON.stringify(this));
    }
    read() {
        return this.storage.read('userInfo');
    }
}
let user = new User('张三', new LocalStorage());
user.save();
user.read();
```

1.2 继承

- 继承主要的要处是实现代码复用
- 继承可以把父类和子类的公共方法抽离出来，提高复用，减少冗余
- 是一种 is-a 关系

```
export { };
class Animal {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    eat() {
        console.log(`${this.name} eat`)
    }
}
let animal = new Animal('动物');
animal.eat();

class Dog extends Animal {
    age: number;
    constructor(name: string, age: number) {
        super(name);
        this.age = age;
    }
    speak() {
        console.log(`${this.name} is barking!`);
    }
}
let dog = new Dog('', 5);
dog.eat();
dog.speak();
```

> 过度使用继承或者说继承层次过深会导致代码可读性、可维护性变差 子类和父类高度耦合，修改父类的代码，会直接影响到子类

1.3 封装

- 把数据封装起来
- 减少耦合，不该外部访问的不要让外部访问
- 利于数据的接口权限管理
- 仅暴露有限的必要接口，提高类的易用性
- 实现
    - public:公有修饰符，可以在类内或者类外使用public修饰的属性或者行为，默认修饰符
    - protected:受保护的修饰符，可以本类和子类中使用protected修饰的属性和行为
    - private：私有修饰符，只可以在类内使用private修饰的属性和行为

```
class Animal {
    public name: string;
    protected age: number;
    private weight: number;
    constructor(name: string, age: number, weight: number) {
        this.name = name;
        this.age = age;
        this.weight = weight;
    }
}
class Person extends Animal {
    private money: number;
    constructor(name: string, age: number, weight: number, money: number) {
        super(name, age, weight);
        this.money = money;
    }
    getName() {
        console.log(this.name);
    }
    getAge() {
        console.log(this.age);
    }
    getWeight() {
        console.log(this.weight);
    }
    getMoney() {
        console.log(this.money);
    }
}
let p = new Person('zfpx', 9, 100, 100);
console.log(p.name);
console.log(p.age);
console.log(p.weight);
```

1.4 多态

- 多态是指，子类可以替换父类
- 保持子类的开放性和灵活性，可以重写父类中的方法
- 实现面向接口编程

```
class Animal {
    public name: string;
    protected age: number;
    private weight: number;
    constructor(name: string, age: number, weight: number) {
        this.name = name;
        this.age = age;
        this.weight = weight;
    }
    speak() {
        throw new Error('此方法必须由子类实现!');
    }
}
class Person extends Animal {
    private money: number;
    constructor(name: string, age: number, weight: number, money: number) {
        super(name, age, weight);
        this.money = money;
    }
    getName() {
        console.log(this.name);
    }
    getAge() {
        console.log(this.age);
    }
    getMoney() {
        console.log(this.money);
    }
    speak() {
        console.log('你好!');
    }

}
class Dog extends Animal {
    constructor(name: string, age: number, weight: number) {
        super(name, age, weight);
    }
    speak() {
        console.log('汪汪汪!');
    }
}
let p = new Person('zfpx', 10, 10, 10);
p.speak();
let d = new Dog('zfpx', 10, 10);
d.speak();
```

2. 设计原则

2.1 什么是设计？

- 按哪一种思路或者标准来实现功能
- 功能相同，可以有不同设计的方式
- 需求如果不断变化，设计的作用才能体现出来

2.2 SOLID五大设计原则

首字母 指代 概念 S 单一职责原则 单一功能原则认为对象应该仅具有一种 单一功能

的概念 O 开放封闭原则 开闭原则认为
软件体应该是对于扩展开放的，但是对于修改&

的概念 L 里氏替换原则 里氏替换原则认为程序中的对象应该是可以在不改变程序正确性的前提下 被它的子类所替换

的的概念 I 接口隔离原则 接口隔离原则认为
多个特定客户端接口要好于一个宽泛用途的接&

的概念 D 依赖反转原则 依赖反转原则认为一个方法应该遵从 依赖于抽象而不是一个实例

的概念,依赖注入是该原则的一种实现方式。

2.2.1 O 开放封闭原则

- Open Closed Principle
- 对扩展开放，对修改关闭
- 增加需求时，扩展新代码，而非修改已有代码
- 开闭原则是设计模式中的总原则
- 对近期可能会变化并且如果有变化但改动量巨大的地方要增加扩展点,扩展点过多会降低可读性

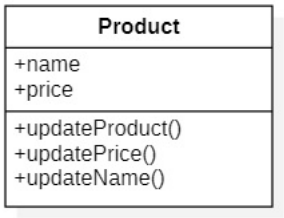
```
class Customer {
    constructor(public rank: string) { }
}
class Product {
    constructor(public name: string, public price: number) {

    }
    cost(customer: Customer) {
        switch (customer.rank) {
            case 'member':
                return this.price * .8;
            case 'vip':
                return this.price * .6;
            default:
                return this.price;
        }
    }
}
let p1 = new Product('笔记本电脑', 1000);
let member = new Customer('member');
let vip = new Customer('vip');
let guest = new Customer('guest');
console.log(p1.cost(member));
console.log(p1.cost(vip));
console.log(p1.cost(guest));
```

```
class Customer {
+    constructor(public rank: string, public discount: number = 1) { }
+    getDiscount() {
+        return this.discount;
+    }
}
class Product {
    constructor(public name: string, public price: number) {

    }
    cost(customer: Customer) {
-        /*  switch (customer.rank) {
-             case 'member':
-                 return this.price * .8;
-             case 'vip':
-                 return this.price * .6;
-             default:
-                 return this.price;
-         } */
+        return this.price * customer.getDiscount();
    }
}
+let p1 = new Product('笔记本电脑', 1000);
+let member = new Customer('member', .8);
+let vip = new Customer('vip', .6);
let guest = new Customer('guest');
console.log(p1.cost(member));
console.log(p1.cost(vip));
console.log(p1.cost(guest));
```

```
import axios, { AxiosInstance, AxiosRequestConfig } from 'axios';
let instance: AxiosInstance = axios.create();
instance.interceptors.request.use((config: AxiosRequestConfig) => {
    config.url = 'http://localhost:8080' + config.url;
    return config;
});

instance.interceptors.response.use(response => {
    if (response.status !== 200 || response.data.code != 0) {
        return Promise.reject(response);
    } else {
        return response.data.data;
    }
})

instance({
    url: '/api/users'
}).then(result => {
    console.log(result);
}, error => {
    console.error(error);
});
```

2.2.2 S 单一职责原则

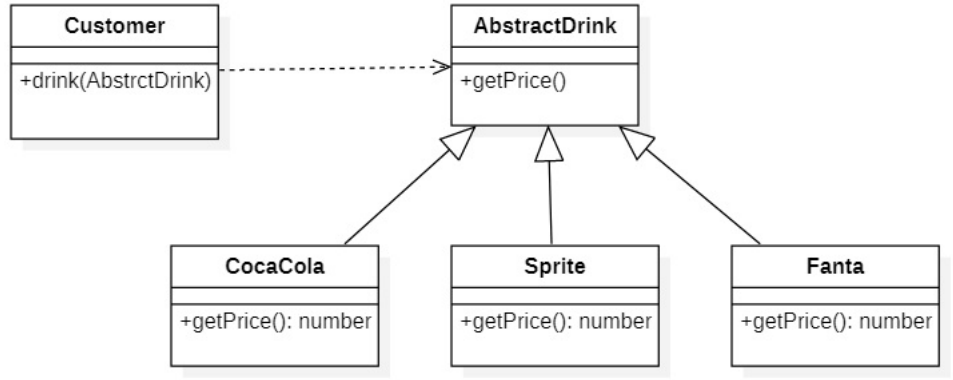
- Single responsibility principle
- 一个类或者模块只负责完成一个职责,如果功能特别复杂就进行拆分
- 单一职责可以降低类的复杂性，提高代码可读性、可维护性
- 当类代码行数过多、方法过多、功能太多、职责太杂的时候就要对类进行拆分了
- 拆分不能过度，如果拆分过度会损失内聚性和维护性
- lodashjs (https://www.lodashjs.com/docs/latest)
- jquery (https://api.jquery.com/)




```
class Product {
    public name: string;
-    public categoryName: string;
-    public categoryIcon: string;
+     public category:Category;
}
+class Category {
+    public name: string;
+    public icon: string;
+}
```

2.2.3 L 里氏替换原则

- Liskov Substitution Principle
- 所有引用基类的地方必须能透明地使用其子类的对象
- 子类能替换掉父类，使用者可能根本就不需要知道是父类还是子类,反之则不行
- 里氏替换原则是开闭原则的实现基础,程序设计的时候尽量使用基类定义及引用，运行时再决定使用哪个子类
- 里氏替换原则可以提高代码的复用性，提高代码的可扩展性，也增加了耦合性
- 相对于多态，这个原则是讲的是类如何设计，子类如果违反了父类的功能则表示违反了里氏替换原则



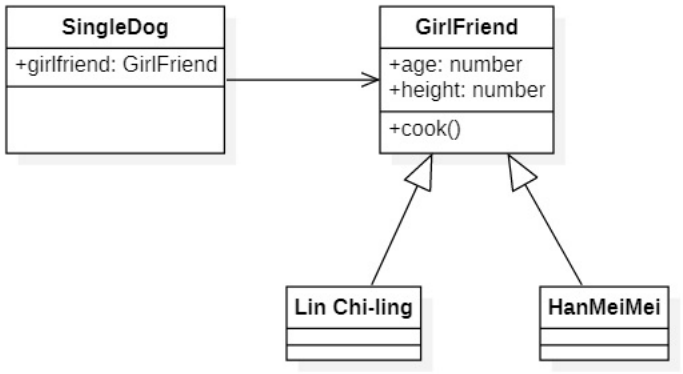

```
abstract class AbstractDrink {
    abstract getName(): string;
}
class CocaCola extends AbstractDrink {
    getName(): string {
        return '可乐';
    }
}
class Sprite extends AbstractDrink {
    getName(): string {
        return '雪碧';
    }
}
class Fanta extends AbstractDrink {
    getName(): string {
        return '芬达';
    }
}
class Customer {
    drink(drink: AbstractDrink) {
        console.log('喝' + drink.getName());
    }
}
let customer = new Customer();
let cocaCola = new CocaCola();
let sprite = new Sprite();
let fanta = new Fanta();
customer.drink(cocaCola);
customer.drink(sprite);
customer.drink(fanta);
```

```
import React from 'react';
import ReactDOM from 'react-dom';
class App extends React.Component {
    render() {
        return (
            <div>App div>
        )
    }
}
let element = React.createElement(App);
ReactDOM.render(element, document.getElementById('root'));
```

```
abstract class AbstractDrink {
    abstract getName(): any;
}
class CocaCola extends AbstractDrink {
    getName(): any {
        return 100;
    }
}
```

2.2.4 D 依赖倒置原则

- Dependence Inversion Principle
- 面向接口编程，依赖于抽象而不依赖于具体实现
- 要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类
- 使用方只关注接口而不关注具体类的实现




```
abstract class GirlFriend {
    public age: number;
    public height: number;
    public abstract cook(): void;
}
class LinZhiLing extends GirlFriend {
    public cook(): void {

    }
}
class HanMeiMei extends GirlFriend {
    public cook(): void {

    }
}
class SingleDog {
    constructor(public girlFriend: GirlFriend) {

    }
}
let s1 = new SingleDog(new LinZhiLing());
let s2 = new SingleDog(new HanMeiMei());
```

```
import { createStore } from 'redux';
let store = createStore(state => state);
export interface Action {
    type: T
}
export interface AnyAction extends Action {

    [extraProps: string]: any
}
let action: AnyAction = { type: 'increment', payload: 5 }
store.dispatch(action);
```

2.2.5 I 接口隔离原则

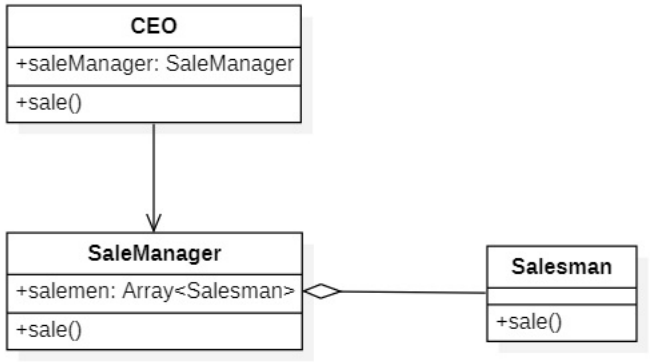
- Interface Segregation Principle
- 保持接口的单一独立，避免出现胖接口
- 客户端不应该依赖它不需要的接口，类间的依赖关系应该建立在最小的接口上
- 接口尽量细化，而且接口中的方法尽量的少
- 类似于单一职责原则，更关注接口

```
interface IUserManager {
    updateUserInfo(): void;
    updatePassword(): void;
}
interface IProductManager {
    updateProduct(): void;
    updatePrice(): void;
}
```

```
interface Running {
    run(): void;
}
interface Flying {
    fly(): void;
}
interface Swimming {
    swim(): void;
}
class Automobile implements Running, Flying, Swimming {
    run() { }
    fly() { }
    swim() { }
}
```

2.3 迪米特法则

- Law of Demeter，LOD
- 有时候也叫做最少知识原则
- 一个软件实体应当尽可能少地与其它实体发生相互作用
- 迪米特法则的初衷在于降低类之间的耦合
- 类定义时尽量要实现内聚,少使用 public修饰符，尽量使用 private、 protected 等

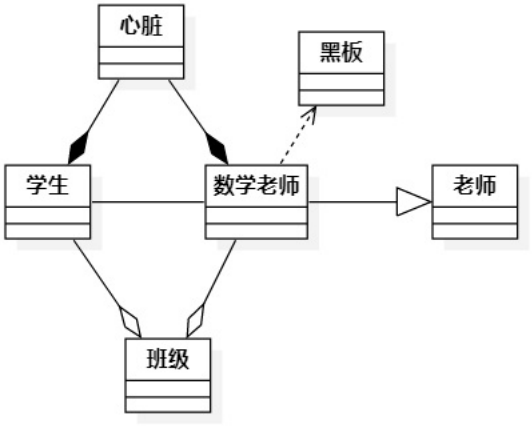



```
class Salesman {
    constructor(public name: string) {

    }
    sale() {
        console.log(this.name + ' 销售中....');
    }
}
class SaleManager {
    private salesmen: Array = [new Salesman('张三'), new Salesman('李四')];
    sale() {
        this.salesmen.forEach(salesman => salesman.sale());
    }
}
class CEO {
    private saleManager: SaleManager = new SaleManager();
    sale() {
        this.saleManager.sale();
    }
}
let ceo = new CEO();
ceo.sale();
```

2.4 合成复用原则

2.4.1 类的关系

- 类之间有三种基本关系，分别是关联(聚合和组合)、泛化和依赖
- 如果一个类单向依赖另一个类,那么它们之间就是单向关联。如果彼此依赖,则为相互依赖,即双向关联
- 关联关系包括两种特例：聚合和组合
    - 聚合，用来表示整体与部分的关系或者 拥有关系,代表部分的对象可能会被整体拥有，但并不定定会随着整体的消亡而销毁,比如班级和学生
    - 合成或者说组合要比聚合关系强的多，部分和整体的生命周期是一致的,比如人和器官之间



2.4.2 合成复用原则

- 合成复用原则是通过将已有的对象纳入新对象中，作为新对象的成员对象来实现的
- 新对象可以调用已有对象的功能，从而达到复用
- 原则是尽量首先使用组合/聚合的方式，而不是使用继承
- 专业人做专业事

```
class Cooker {
    cook() {

    }
}
class Person {
    private cooker: Cooker = new Cooker();
    cook() {
        this.cooker.cook();
    }
}
```

2.5 总结

- 开闭原则是核心，对修改关闭对扩展开放是软件设计的基石
- 单一职责要求我们设计接口和模块功能的时候尽量保证单一性和原子性，修改一条不影响全局和其它模块
- 里氏替换原则和依赖倒置原则要求面向接口和抽象编程,不要依赖具体实现，否则实现一改，上层调用者就要对应修改

2.6 如何写出好代码?

- 可维护性 BUG是否好改?
- 可读性 是否容易看懂?
- 可扩展性 是否可以添加新功能?
- 灵活性 添加新功能是否容易?老方法和接口是否容易复用?
- 简洁性 代码是否简单清晰?
- 可复用性 相同的代码不要写2遍?
- 可测试性 是否方便写单元测试和集成测试?

3. 23种设计模式

3.1 创建型

- 重要: 工厂模式
  (工厂方法模式、抽象工厂模式、简单工厂模&
- 不重要: 原型模式

3.2 结构型模式

- 重要:
  代理模式、桥接模式、装饰器模式、适配器&#
- 不重要: 外观模式、组合模式、享元模式

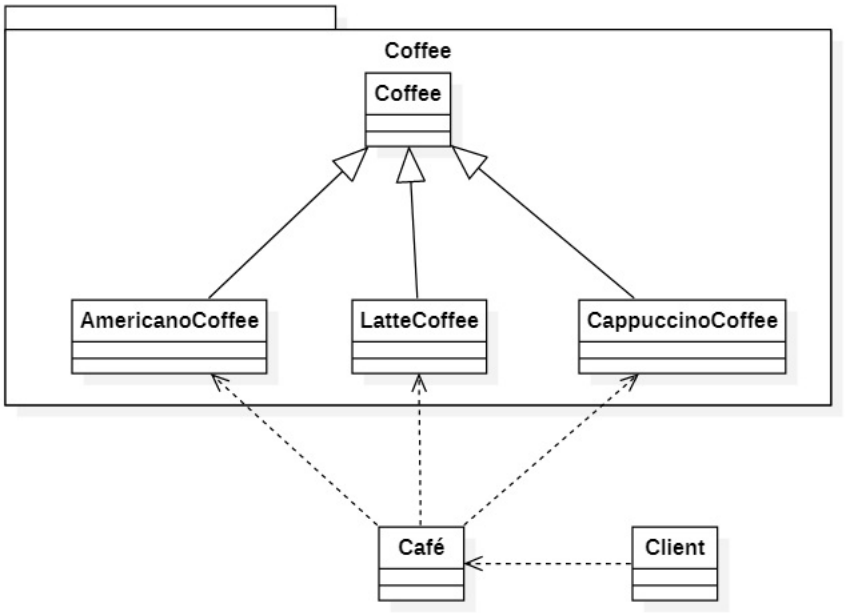
3.3 行为型

- 重要:
  观察者模式、模版方法模式、策略模式、职&#
- 不重要: 访问者模式、备忘录模式、命令模式、解释器模式、中介者模式

4. 工厂模式

4.1 简单工厂模式

- 简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例*4.1.1 类图#



** 4.1.2 代码 #**

```
abstract class Coffee {
    constructor(public name: string) {

    }
}
class AmericanoCoffee extends Coffee {
    constructor(public name: string) {
        super(name);
    }
}
class LatteCoffee extends Coffee {
    constructor(public name: string) {
        super(name);
    }
}
class CappuccinoCoffee extends Coffee {
    constructor(public name: string) {
        super(name);
    }
}

class Café {
    static order(name: string) {
        switch (name) {
            case 'Americano':
                return new AmericanoCoffee('美式咖啡');
            case 'Latte':
                return new LatteCoffee('拿铁咖啡');
            case 'Cappuccino':
                return new LatteCoffee('卡布奇诺');
            default:
                return null;
        }
    }
}
console.log(Café.order('Americano'));
console.log(Café.order('Latte'));
console.log(Café.order('Cappuccino'));
```

4.1.3 缺点

- 如果产品的种类非常多 `switch case`的判断会变得非常多
- 不符合开放—封闭原则,如果要增加或删除一个产品种类，就要修改 `switch case`的判断代码

** 4.1.3 前端应用场景 #**

4.1.3.1 jQuery#

- jQuery (https://github.com/jquery/jquery/blob/c1ee33aded44051b8f1288b59d2efdc68d0413cc/src/core.js#L24-L29)

```
class jQuery{
    constructor(selector){
        let elements = Array.from(document.querySelectorAll(selector));
        let length = elements?elements.length:0;
        for(let i=0;ithis[i]=elements[i];
        }
        this.length = length;
    }
    html(html){
        if(html){
           this[0].innerHTML=html;
        }else{
          return this[0].innerHTML;
        }
    }
}
window.$ = function(selector){
   return new jQuery(selector);
}
```

4.1.3.2 React

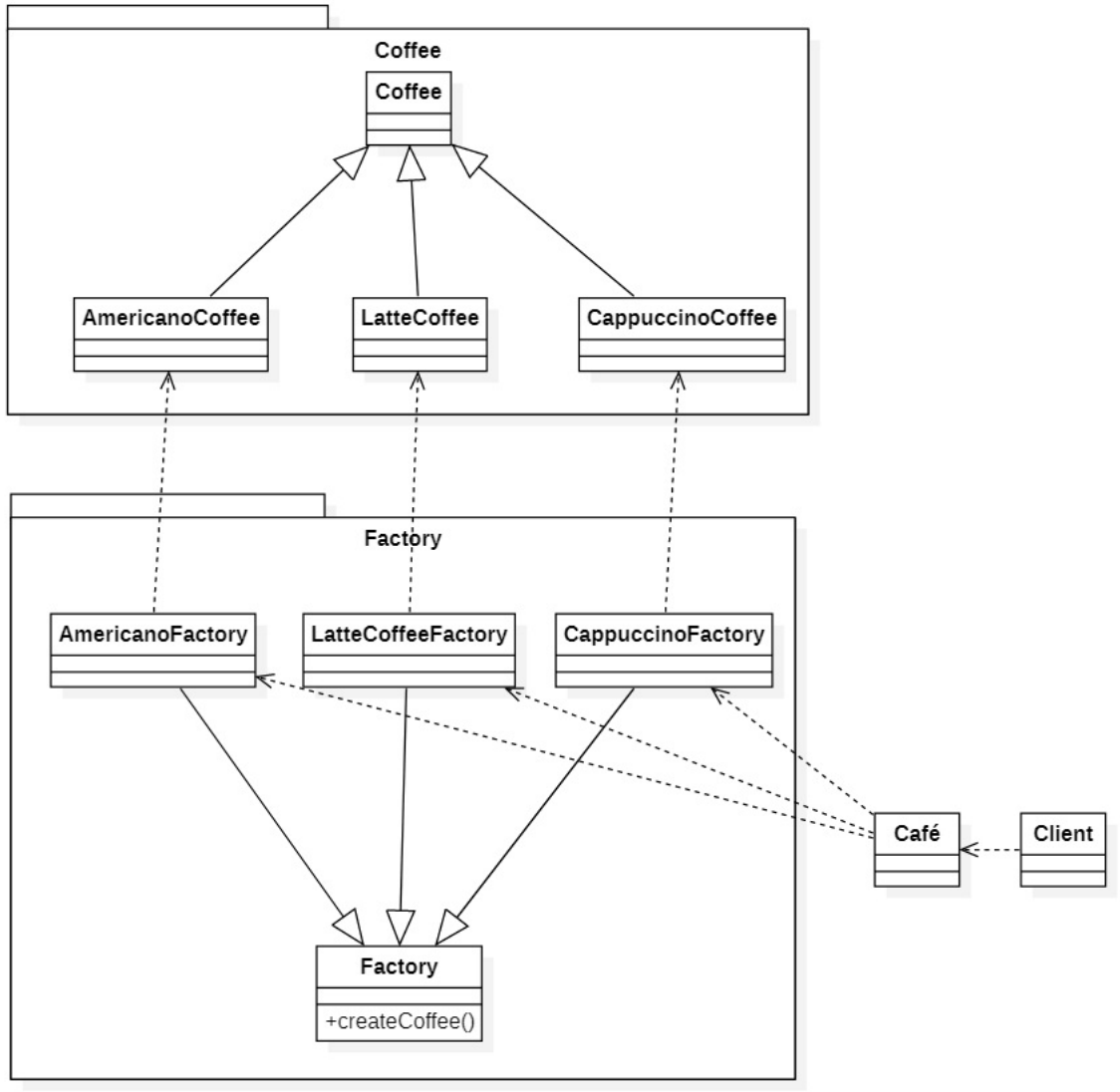
- createElement (https://github.com/facebook/react/blob/master/packages/react/src/ReactElement.js#L313-L395)

```
export function createElement(type, config, children) {
    return ReactElement(
        type,
        key,
        ref,
        self,
        source,
        ReactCurrentOwner.current,
        props,
    );
}
```

4.2 工厂方法模式

- 工厂方法模式 Factory Method,又称多态性工厂模式。
- 在工厂方法模式中,核心的工厂类不再负责所有的产品的创建，而是将具体创建的工作交给工厂子类去做。

** 4.2.1 类图 #**



** 4.2.2 代码 #**

```
export { }
abstract class Coffee {
    constructor(public name: string) {

    }
}
abstract class Factory {
    abstract createCoffee(): Coffee;
}
class AmericanoCoffee extends Coffee {
    constructor(public name: string) {
        super(name);
    }
}

class AmericanoCoffeeFactory extends Factory {
    createCoffee() {
        return new AmericanoCoffee('美式咖啡')
    }
}

class LatteCoffee extends Coffee {
    constructor(public name: string) {
        super(name);
    }
}
class LatteCoffeeFactory extends Factory {
    createCoffee() {
        return new LatteCoffee('拿铁咖啡')
    }
}
class CappuccinoCoffee extends Coffee {
    constructor(public name: string) {
        super(name);
    }
}
class CappuccinoFactory extends Factory {
    createCoffee() {
        return new CappuccinoCoffee('卡布奇诺')
    }
}
class Café {
    static order(name: string) {
        switch (name) {
            case 'Americano':
                return new AmericanoCoffeeFactory().createCoffee();
            case 'Latte':
                return new LatteCoffeeFactory().createCoffee();
            case 'Cappuccino':
                return new CappuccinoFactory().createCoffee();
            default:
                return null;
        }
    }
}
console.log(Café.order('Americano'));
console.log(Café.order('Latte'));
console.log(Café.order('Cappuccino'));
```

** 4.2.3 应用场景 #**

4.2.3.1 React

- createFactory (https://github.com/facebook/react/blob/master/packages/react/src/ReactElement.js#L401-L409)

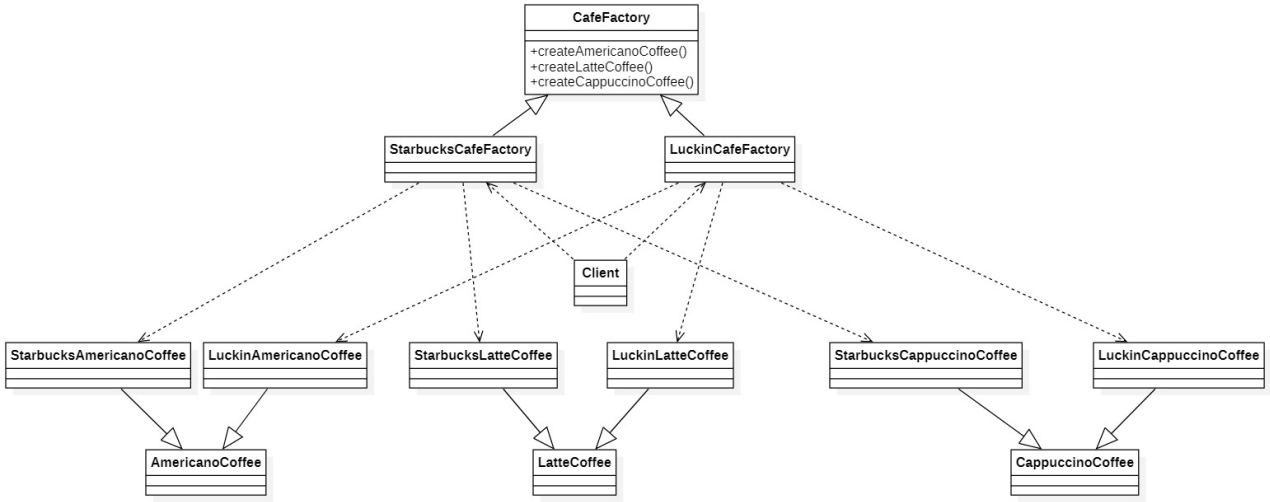
4.3 抽象工厂模式

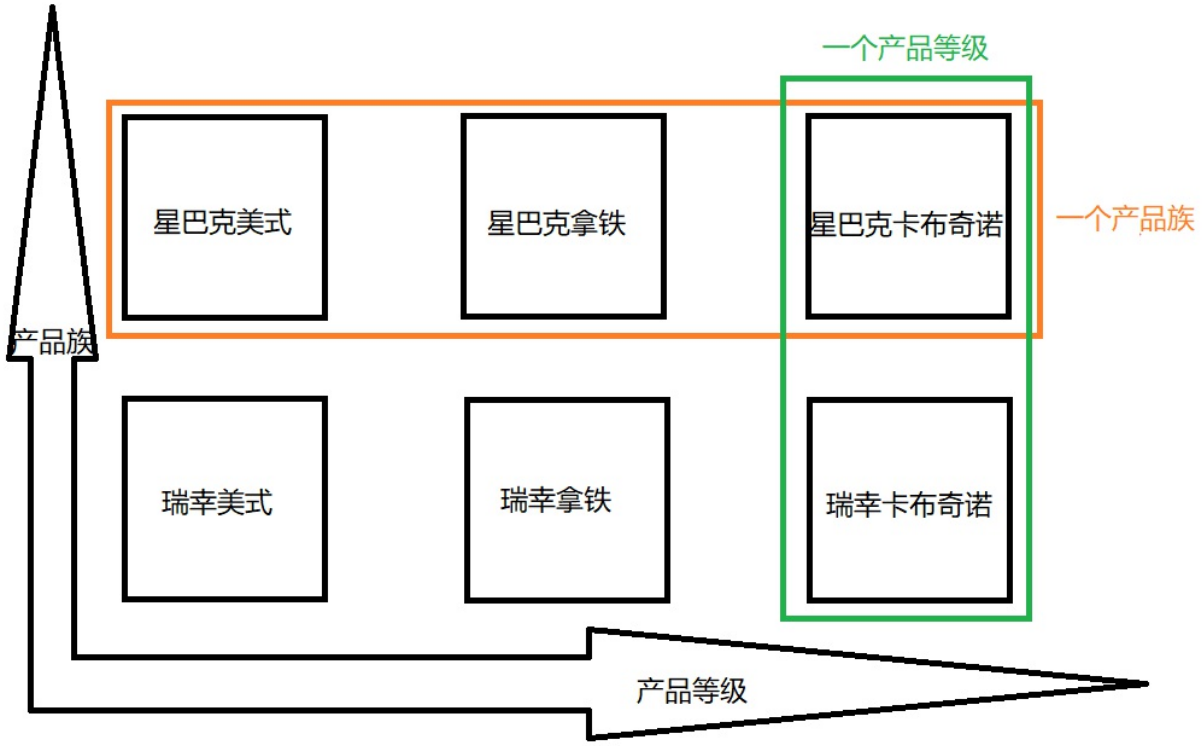
- 抽象工厂模式可以向客户端提供一个接口，使客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象
- 工厂方法模式针对的是同一类或同等级产品,而抽象工厂模式针对的是多种类的产品设计
- 系统中有多个产品族，每个具体工厂负责创建同一族但属于不同产品等级(产品种类)的产品
- 产品族是一组相关或相互依赖的对象
- 系统一次只能消费某一族产品，即相同产品族的产品是一起被使用的
- 当系统需要新增一个产品族时，只需要增加新的工厂类即可，无需修改源代码；但是如果需要产品族中增加一个新种类的产品时，则所有的工厂类都需要修改

** 4.3.1 组成角色 #**

- 抽象工厂: 提供了创建产品的接口,包含多个创建产品的方法,即包含多个类似创建产品的方法
- 具体工厂: 实现抽象工厂定义的接口,完成某个具体产品的创建
- 抽象产品: 抽象产品定义,一般有多少抽象产品,抽象工厂中就包含多少个创建产品的方法
- 具体产品: 抽象产品的实现类

** 4.3.1 类图 #**


CafeFactory
+createAmericanoCoffee()
+createLatteCoffee()
+createCappuccinoCoffee()
StarbucksCafeFactory
LuckinCafeFactory
Client
StarbucksAmericanoCoffee
LuckinAmericanoCoffee
StarbucksLatteCoffee
LuckinLatteCoffee
StarbucksCappuccinoCoffee
LuckinCappuccinoCoffee
AmericanoCoffee
LatteCoffee
CappuccinoCoffee

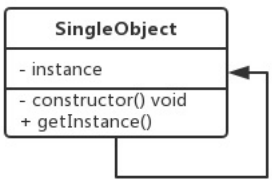

一个产品等级
星巴克美式
星巴克拿铁
星巴克卡布奇诺
一个产品族
产品族
瑞幸美式
瑞幸拿铁
瑞幸卡布奇诺
产品等级

** 4.3.2 代码 #**

```
export { };
abstract class AmericanoCoffee { }
abstract class LatteCoffee { }
abstract class CappuccinoCoffee { }

class StarbucksAmericanoCoffee extends AmericanoCoffee { }
class StarbucksLatteCoffee extends LatteCoffee { }
class StarbucksCappuccinoCoffee extends CappuccinoCoffee { }

class LuckinAmericanoCoffee extends AmericanoCoffee { }
class LuckinLatteCoffee extends LatteCoffee { }
class LuckinCappuccinoCoffee extends CappuccinoCoffee { }

abstract class CafeFactory {
    abstract createAmericanoCoffee(): AmericanoCoffee;
    abstract createLatteCoffee(): LatteCoffee;
    abstract createCappuccinoCoffee(): CappuccinoCoffee;
}
class StarbucksCafeFactory extends CafeFactory {
    createAmericanoCoffee() {
        return new StarbucksAmericanoCoffee();
    }
    createLatteCoffee() {
        return new StarbucksLatteCoffee();
    }
    createCappuccinoCoffee() {
        return new StarbucksCappuccinoCoffee();
    }
}
class LuckinCafeFactory extends CafeFactory {
    createAmericanoCoffee() {
        return new LuckinAmericanoCoffee();
    }
    createLatteCoffee() {
        return new LuckinLatteCoffee();
    }
    createCappuccinoCoffee() {
        return new LuckinCappuccinoCoffee();
    }
}

let starbucksCafeFactory = new StarbucksCafeFactory();
console.log(starbucksCafeFactory.createAmericanoCoffee());
console.log(starbucksCafeFactory.createCappuccinoCoffee());
console.log(starbucksCafeFactory.createLatteCoffee());

let luckinCafeFactory = new LuckinCafeFactory();
console.log(luckinCafeFactory.createAmericanoCoffee());
console.log(luckinCafeFactory.createCappuccinoCoffee());
console.log(luckinCafeFactory.createLatteCoffee());
```

5.单例模式

** 5.1 类图 #**



** 5.2 代码 #**

5.2.1 单例模式

```
export { };
class Window {
    private static instance: Window;
    private constructor() { }
    static getInstance() {
        if (!Window.instance) {
            Window.instance = new Window();
        }
        return Window.instance;
    }
}

var w1 = Window.getInstance();
var w2 = Window.getInstance();
console.log(w1 === w2);
```

5.2.2 ES5单例模式

```
interface Window {
    hello: any
}
function Window() { }
Window.prototype.hello = function () {
    console.log('hello');
}
Window.getInstance = (function () {
    let window: Window;
    return function () {
        if (!window)
            window = new (Window as any)();
        return window;
    }
})();
let window = Window.getInstance();
window.hello();
```

5.2.3 透明单例

```
let Window = (function () {
    let window: Window;
    let Window = function (this: Window) {
        if (window) {
            return window;
        } else {
            return (window = this);
        }
    }
    Window.prototype.hello = function () {
        console.log('hello');
    }
    return Window;
})();

let window1 = new (Window as any)();
let window2 = new (Window as any)();
window1.hello();
console.log(window1 === window2)
```

5.2.4 单例与构建分离

```
export { }
interface Window {
    hello: any
}
function Window() {
}
Window.prototype.hello = function () {
    console.log('hello');
}

let createInstance = (function () {
    let instance: Window;
    return function () {
        if (!instance) {
            instance = new (Window as any)();
        }
        return instance;
    }
})();

let window1 = createInstance();
let window2 = createInstance();
window1.hello();
console.log(window1 === window2)
```

5.2.5 封装变化

```
export { }
function Window() {

}
Window.prototype.hello = function () {
    console.log('hello');
}

let createInstance = function (Constructor: any) {
    let instance: any;
    return function (this: any) {
        if (!instance) {
            Constructor.apply(this, arguments);
            Object.setPrototypeOf(this, Constructor.prototype)
            instance = this;
        }
        return instance;
    }
};
let CreateWindow: any = createInstance(Window);
let window1 = new CreateWindow();
let window2 = new CreateWindow();
window1.hello();
console.log(window1 === window2)
```

** 5.3 场景 #**

5.3.2 commonjs

- HotModuleReplacement (https://github.com/webpack/webpack/blob/8070bcd333cd1d07ce13fe5e91530c80779d51c6/lib/hmr/HotModuleReplacement.runtime.js#L55)

```
(function(modules) {

  var installedModules = {};

  function __webpack_require__(moduleId) {

    if (installedModules[moduleId]) {
      return installedModules[moduleId].exports;
    }

    var module = (installedModules[moduleId] = {
      i: moduleId,
      l: false,
      exports: {}
    });

    modules[moduleId].call(
      module.exports,
      module,
      module.exports,
      __webpack_require__
    );

    module.l = true;

    return module.exports;
  }
}
```

5.3.2 jQuery#

- jquery (https://code.jquery.com/jquery-3.4.1.js)

```
if(window.jQuery!=null){
  return window.jQuery;
}else{

}
```

5.3.3 模态窗口

- bootstrap modal.js (https://github.com/twbs/bootstrap/blob/master/js/src/modal.js)

```
    Document

    显示模态窗口
    隐藏模态窗口

        class Login {
            constructor() {
                this.element = document.createElement('div');
                this.element.innerHTML = (
                    `
            用户名 <input type="text"/>
            <button>登录</button>
            `
                );
                this.element.style.cssText = 'width: 100px; height: 100px; position: absolute; left: 50%; top: 50%; display: block;';
                document.body.appendChild(this.element);
            }
            show() {
                this.element.style.display = 'block';
            }
            hide() {
                this.element.style.display = 'none';
            }
        }
        Login.getInstance = (function () {
            let instance;
            return function () {
                if (!instance) {
                    instance = new Login();
                }
                return instance;
            }
        })();

        document.getElementById('show-button').addEventListener('click', function (event) {
            Login.getInstance().show();
        });
        document.getElementById('hide-button').addEventListener('click', function (event) {
            Login.getInstance().hide();
        });
```

5.3.4 store

- createStore.ts (https://github.com/reduxjs/redux/blob/master/src/createStore.ts)

```
function createStore(reducer: any) {
    let state: any;
    let listeners: any[] = [];
    function getState() {
        return state;
    }
    function dispatch(action: any) {
        state = reducer(state, action);
        listeners.forEach(l => l());
    }
    function subscribe(listener: any) {
        listeners.push(listener);
        return () => {
            listeners = listeners.filter(item => item != listener);
            console.log(listeners);
        }
    }
    dispatch({});
    return {
        getState,
        dispatch,
        subscribe
    }
}
let store = createStore((state: any, action: any) => state);
```

5.3.5 缓存

```
let express = require('express');
let fs = require('fs');
let cache: Record = {};
let app = express();
app.get('/user/:id', function (req: any, res: any) {
    let id = req.params.id;
    let user = cache.get(id);
    if (user) {
        res.json(user);
    } else {
        fs.readFile(`./users/${id}.json`, 'utf8', function (err: any, data: any) {
            let user = JSON.parse(data);
            cache.put(id, user);
            res.json(user);
        });
    }
});
app.listen(3000);
```

6. 适配器模式

- 适配器模式又称包装器模式,将一个类的接口转化为用户需要的另一个接口,解决类(对象)之间接口不兼容的问题
- 旧的接口和使用者不兼容
- 中间加一个适配器转换接口

** 6.1 类图 #**

** 6.2 代码 #**

```
class Socket {
    output() {
        return '输出220V';
    }
}

abstract class Power {
    abstract charge(): string;
}
class PowerAdapter extends Power {
    constructor(public socket: Socket) {
        super();
    }

    charge() {
        return this.socket.output() + ' 经过转换 输出24V';
    }
}
let powerAdapter = new PowerAdapter(new Socket());
console.log(powerAdapter.charge());
```

** 6.3 场景 #**

6.2.1 axios

- Axios (https://github.com/axios/axios/blob/master/lib/core/Axios.js#L6)
- dispatchRequest (https://github.com/axios/axios/blob/master/lib/core/dispatchRequest.js#L50-L79)
- defaults (https://github.com/axios/axios/blob/master/lib/defaults.js#L16-L26)
- xhr (https://github.com/axios/axios/blob/dc4bc49673943e35280e5df831f5c3d0347a9393/lib/adapters/xhr.js)
- http (https://github.com/axios/axios/blob/dc4bc49673943e35280e5df831f5c3d0347a9393/lib/adapters/http.js)
- 适配器的入参都是 config,返回的都是 promise

```

let url = require('url');
function axios(config: any): any {
    let adaptor = getDefaultAdapter();
    return adaptor(config);
}
axios({
    method: 'GET',
    url: 'http://localhost:8080/api/user?id=1'
}).then(function (response: any) {
    console.log(response);
}, function (error: any) {
    console.log(error);
})

function xhr(config: any) {
    return new Promise(function (resolve, reject) {
        var request = new XMLHttpRequest();
        request.open(config.method, config.url, true);
        request.onreadystatechange = function () {
            if (request.readyState == 4) {
                if (request.status == 200) {
                    resolve(request.response);
                } else {
                    reject('请求失败');
                }
            }
        }
    })
}
function http(config: any) {
    let http = require('http');
    let urlObject = url.parse(config.url);
    return new Promise(function (resolve, reject) {
        const options = {
            hostname: urlObject.hostname,
            port: urlObject.port,
            path: urlObject.pathname,
            method: config.method
        };
        var req = http.request(options, function (res: any) {
            let chunks: any[] = [];
            res.on('data', (chunk: any) => {
                chunks.push(chunk);
            });
            res.on('end', () => {
                resolve(Buffer.concat(chunks).toString());
            });
        });
        req.on('error', (err: any) => {
            reject(err);
        });
        req.end();
    })
}
function getDefaultAdapter(): any {
    var adapter;
    if (typeof XMLHttpRequest !== 'undefined') {
        adapter = xhr;
    } else if (typeof process !== 'undefined') {
        adapter = http;
    }
    return adapter;
}
```

server.js

```
let express = require('express');
let app = express();
app.get('/api/user', (req, res) => {
    res.json({ id: req.query.id, name: 'zhufeng' });
});
app.listen(8080);
```

6.2.2 toAxiosAdapter

```
function toAxiosAdapter(options: any) {
    return axios({
        url: options.url,
        method: options.type
    }).then(options.success)
        .catch(options.error)
}

$.ajax = function (options: any) {
    return toAxiosAdapter(options)
}

$.ajax({
    url: '/api/user',
    type: 'GET',
    success: function (data: any) {
        console.log(data)
    },
    error: function (err: any) {
        console.error(err);
    }
})
```

6.2.3 promisify#

```
let fs = require('fs');
var Bluebird = require("bluebird");
let readFile = Bluebird.promisify(fs.readFile);

(async function () {
    let content = await readFile('./1.txt', 'utf8');
    console.log(content);
})()
```

```
function promisify(readFile: any) {
    return function (filename: any, encoding: any) {
        return new Promise(function (resolve, reject) {
            readFile(filename, encoding, function (err: any, data: any) {
                if (err)
                    reject(err);
                else
                    resolve(data);
            })
        });
    }
}
```

6.2.4 computed

```
    vue

{{name}}
{{upperName}}

let vm=new Vue({
    el: '#root',
    data: {
        name:'zfpx'
    },
    computed: {
        upperName() {
            return this.name.toUpperCase();
        }
    }
});
```

6.2.5 tree

- 树型结构数据在下拉框中显示

```
Document

    let tree = [{
        name: '父亲',
        key: '1',
        children: [
            {
                name: '儿子',
                key: '1-1',
                children: [
                    {
                        name: '孙子',
                        key: '1-1-1'
                    }
                ]
            }
        ]
    }]
    function flattenAdapter(tree, flattenArray) {
        tree.forEach((item) => {
            if (item.children) {
                flattenAdapter(item.children, flattenArray)
            }
            flattenArray.push({ name: item.name, key: item.key })
        })
        return flattenArray
    }
    let array = [];
    flattenAdapter(tree, array);
    array.reverse();
    let users = document.getElementById('users');
    let options = array.map(item => `<option value="${item.key}">${item.name}</option>`).join('');
    users.innerHTML = options;
```

6.2.6 Sequelize

- sequelize (https://github.com/demopark/sequelize-docs-Zh-CN/tree/master)

```
const { Sequelize, Model, DataTypes } = require('sequelize');
const sequelize = new Sequelize('sqlite::memory:');

class User extends Model { }
User.init({
    username: DataTypes.STRING
}, { sequelize, modelName: 'user' });

sequelize.sync()
    .then(() => User.create({
        username: 'zhufeng'
    }))
    .then(result => {
        console.log(result.toJSON());
    });
```

7.装饰器模式

- 在不改变其原有的结构和功能为对象添加新功能的模式其实就叫做装饰器模式
- 最直观地就是我们买房后的装修
- 装饰比继承更加灵活,可以实现装饰者和被装饰者之间松耦合
- 被装饰者可以使用装饰者动态地增加和撤销功能

** 7.1 类图 #**

** 7.2 代码 #**

```
abstract class Shape {
    abstract draw(): void;
}
class Circle extends Shape {
    draw() {
        console.log('绘制圆形');
    }
}
class Rectangle extends Shape {
    draw() {
        console.log('绘制矩形');
    }
}

abstract class ColorfulShape extends Shape {
    public constructor(public shape: Shape) {
        super();
    }
    abstract draw(): void;
}

class RedColorfulShape extends ColorfulShape {
    draw() {
        this.shape.draw();
        console.log('把边框涂成红色');
    }
}
class GreenColorfulShape extends ColorfulShape {
    draw() {
        this.shape.draw();
        console.log('把边框涂成绿色');
    }
}

let circle = new Circle();
let redColorfulShape = new RedColorfulShape(circle);
redColorfulShape.draw();

let rectangle = new Rectangle();
let greenColorfulShape = new GreenColorfulShape(rectangle);
greenColorfulShape.draw();
```

** 7.3 应用场景 #**

7.3.1 装饰器

- 装饰器是一种特殊类型的声明，它能够被附加到类声明、方法、属性或参数上，可以修改类的行为
- 常见的装饰器有类装饰器、属性装饰器、方法装饰器和参数装饰器
- 装饰器的写法分为普通装饰器和装饰器工厂

7.3.1.1 类装饰器

- 类装饰器在类声明之前声明，用来监控、修改或替换类定义
- 参数是类的定义或者说构造函数
- babel-plugin-proposal-decorators (https://babeljs.io/docs/en/babel-plugin-proposal-decorators)

decorator

```
export { }
namespace decorator {
    interface Animal {
        swings: string;
        fly: any
    }
    function flyable(target: any) {
        console.log(target);

        target.prototype.swings = 2;
        target.prototype.fly = function () {
            console.log('I can fly');
        }
    }
    @flyable
    class Animal {
        constructor() { }
    }
    let animal: Animal = new Animal();
    console.log(animal.swings);
    animal.fly();
}
```

decorator_factory

```
namespace decorator_factory {
    interface Animal {
        swings: string;
        fly: any
    }
    function flyable(swings: number) {
        return function flyable(target: any) {
            console.log(target);

            target.prototype.swings = swings;
            target.prototype.fly = function () {
                console.log('I can fly');
            }
        }
    }

    @flyable(2)
    class Animal {
        constructor() { }
    }
    let animal: Animal = new Animal();
    console.log(animal.swings);
    animal.fly();
}
```

7.3.1.2 属性装饰器

- 属性装饰器表达式会在运行时当作函数被调用
- 属性分为实例属性和类属性
- 方法分为实例方法和类方法

```
namespace property_namespace {

    function instancePropertyDecorator(target: any, key: string) {
    }

    function classPropertyDecorator(target: any, key: string) {
    }

    function instanceMethodDecorator(target: any, key: string, descriptor: PropertyDescriptor) {
    }

    function classMethodDecorator(target: any, key: string, descriptor: PropertyDescriptor) {
    }
    class Person {
        @instancePropertyDecorator
        instanceProperty: string;
        @classPropertyDecorator
        public static classProperty: string;
        @instanceMethodDecorator
        instanceMethod() {
            console.log('instanceMethod');
        }
        @classMethodDecorator
        classMethod() {
            console.log('classMethod');
        }
    }
}
```

7.3.1.3 core-decorator

- core-decorator (https://github.com/jayphelps/core-decorators)
- deprecate-alias-deprecated (https://github.com/jayphelps/core-decorators#deprecate-alias-deprecated)

```
let { readonly } = require('core-decorators');
function deprecate(msg: string, options: any) {
    return function (target: any, attr: any, descriptor: any) {

        let oldVal = descriptor.value;
        descriptor.value = function (...args: any[]) {
            let message = msg ? msg : `DEPRECATION ${target.constructor.name}#${attr}: This function will be removed in future versions.`;
            let see = options && options.url ? `see ${options.url}` : ``;
            console.warn(message + '\r\n' + see);
            return oldVal(...args);
        }
    }
}
class Calculator {
    @deprecate('stop using this', { url: 'http://www.baidu.com' })
    add(a: number, b: number) {
        return a + b;
    }
}
let calculator = new Calculator();
calculator.add(1, 2);
```

7.3.2 AOP概念

- 在软件业，AOP为 Aspect Oriented Programming的缩写,意为面向切面编程
- 可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术

7.3.3 埋点

- 埋点分析: 是网站分析的一种常用的数据采集方法
- 无痕埋点: 通过技术手段，完成对用户行为数据无差别的统计上传的工作,后期数据分析处理的时候通过技术手段筛选出合适的数据进行统计分析

7.3.3.1 项目配置

1. 创建项目

```
create-react-app zhufeng_tract
yarn add customize-cra react-app-rewired --dev
```

1. config-overrides.js

```
const {
 override,
 addDecoratorsLegacy,
} = require("customize-cra");
module.exports = override(
 addDecoratorsLegacy(),
);
```

2. jsconfig.json

```
{
 "compilerOptions": {
     "experimentalDecorators": true
 }
}
```

7.3.3.2 index.js

```
import React from 'react';
import { render } from 'react-dom';
import { before, after } from './track';

class App extends React.Component {
    @before(() => console.log('点击方法执行前'))
    onClickBeforeButton() {
        console.log('beforeClick');
    }

    @after(() => console.log('点击方法执行后'))
    onClickAfterButton() {
        console.log('afterClick');
    }

    @after(() => fetch('/api/report'))
    onClickAjaxButton() {
        console.log('ajaxClick');
    }

    render() {
        return (
            <div>
                <button onClick={this.onClickBeforeButton}>beforeClickbutton>
                <button onClick={this.onClickAfterButton}>afterClickbutton>
                <button onClick={this.onClickAjaxButton}>ajaxClickbutton>
            div>
        )
    }
}
render(<App />, document.getElementById('root'));
```

7.3.3.3 track.js

```
export const before = function (beforeFn) {
    return function (target, methodName, descriptor) {
        let oldMethod = descriptor.value;
        descriptor.value = function () {
            beforeFn.apply(this, arguments);
            return oldMethod.apply(this, arguments);
        }
    }
}

export const after = function (afterFn) {
    return function (target, methodName, descriptor) {
        let oldMethod = descriptor.value;
        descriptor.value = function () {
            oldMethod.apply(this, arguments);
            afterFn.apply(this, arguments);
        }
    }
}
```

7.3.4 表单校验

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>用户注册title>
head>

<body>
    <form action="">
        用户名<input type="text" name="username" id="username">
        密码<input type="text" name="password" id="password">
        <button id="submit-btn">注册button>
    form>
    <script>
        Function.prototype.before = function (beforeFn) {
            let _this = this;
            return function () {
                let ret = beforeFn.apply(this, arguments);
                if (ret)
                    _this.apply(this, arguments);
            }
        }
        function submit() {
            alert('提交表单');
        }
        submit = submit.before(function () {
            let username = document.getElementById('username').value;
            if (username.length < 6) {
                return alert('用户名不能少于6位');
            }
            return true;
        });
        submit = submit.before(function () {
            let username = document.getElementById('username').value;
            if (!username) {
                return alert('用户名不能为空');
            }
            return true;
        });
        document.getElementById('submit-btn').addEventListener('click', submit);
    script>
body>
html>
```

8.代理模式

- 由于一个对象不能直接引用另外一个对象，所以需要通过代理对象在这两个对象之间起到中介作用
- 代理模式就是为目标对象创造一个代理对象，以实现对目标对象的访问
- 这样就可以在代理对象里增加一些逻辑判断、调用前或调用后执行一些操作，从而实现了扩展目标的功能
- 火车票代购、房产中介、律师、海外代购、明星经纪人

** 8.1 类图 #**

- Target 目标对象，也就是被代理的对象，是具体业务的执行者
- Proxy 代理对象，里面会包含一个目标对象的引用，可以实现对访问的扩展和额外处理

** 8.2 代码 #**

```
abstract class Star {
    abstract answerPhone(): void;
}

class Angelababy extends Star {
    public available: boolean = true;
    answerPhone(): void {
        console.log('你好,我是Angelababy.');
    }
}
class AngelababyAgent extends Star {
    constructor(private angelababy: Angelababy) {
        super();
    }
    answerPhone(): void {
        console.log('你好,我是Angelababy的经纪人.');
        if (this.angelababy.available) {
            this.angelababy.answerPhone();
        }
    }
}
let angelababyAgent = new AngelababyAgent(new Angelababy());
angelababyAgent.answerPhone();
```

** 8.3 场景 #**

8.3.1 事件委托代理

- 事件捕获指的是从document到触发事件的那个节点，即自上而下的去触发事件
- 事件冒泡是自下而上的去触发事件
- 绑定事件方法的第三个参数，就是控制事件触发顺序是否为事件捕获。true为事件捕获；false为事件冒泡,默认false。

```
1
2
3
```

```
let list = document.querySelector('#list');
list.addEventListener('click',event=>{
    alert(event.target.innerHTML);
});
```

8.3.2 虚拟代理(图片预加载)#

- bg-images (http://img.zhufengpeixun.cn/bg-images.zip)

8.3.2.1 app.js

```
let express=require('express');
let path=require('path')
let app=express();
app.get('/images/loading.gif',function (req,res) {
    res.sendFile(path.join(__dirname,req.path));
});
app.get('/images/:name',function (req,res) {
    setTimeout(() => {
        res.sendFile(path.join(__dirname,req.path));
    }, 2000);
});
app.get('/',function (req,res) {
    res.sendFile(path.resolve('index.html'));
});
app.listen(8080);
```

8.3.2.2 index.html

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Documenttitle>
    <style>
        .bg-container {
            width: 600px;
            height: 400px;
            margin: 100px auto;
        }

        .bg-container #bg-image {
            width: 100%;
            height: 100%;
        }
    style>
head>

<body>
    <div id="background">
        <button data-src="/images/bg1.jpg">背景1button>
        <button data-src="/images/bg2.jpg">背景2button>
    div>
    <div class="bg-container">
        <img id="bg-image" src="/images/bg1.jpg" />
    div>
    <script>
        let [illegible] = document.querySelector('#background');

        class BackgroundImage {
            constructor() {
                this.bgImage = document.querySelector('#bg-image');
            }
            setSrc(data) {
                this.bgImage.src = data;
            }
        }
        class LoadingBackgroundImage {
            static LOADING_URL = '/images/loading.gif';
            constructor() {
                this.backgroundImage = new BackgroundImage();
            }
            setSrc(data) {
                this.backgroundImage.setSrc(LoadingBackgroundImage.LOADING_URL);
                let img = new Image();
                img.onload = () => {
                    this.backgroundImage.setSrc(data);
                }
                img.src = data;
            }
        }
        let loadingBackgroundImage = new LoadingBackgroundImage();
        document.addEventListener('click', function (event) {
            let src = event.target.dataset.src;
            loadingBackgroundImage.setSrc(src + '?t=' + Date.now());
        });
    script>
body>

html>
```

8.3.3 虚拟代理(图片懒加载)#

- 当前可视区域的高度 window.innerHeight || document.documentElement.clientHeight
- 元素距离可视区域顶部的高度 getBoundingClientRect().top

- [getBoundingClientRect (https://developer.mozilla.org/zh-CN/docs/Web/API/Element/getBoundingClientRect)](https://developer.mozilla.org/zh-CN/docs/Web/API/Element/getBoundingClientRect)
- DOMRect 对象包含了一组用于描述边框的只读属性——left、top、right 和 bottom，单位为像素。除了 width 和 height 外的属性都是相对于视口的左上角位置而言的

8.3.3.1 index.html

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Lazy-Loadtitle>
    <style>
        .image {
            width: 300px;
            height: 200px;
            background-color: #CCC;
        }

        .image img {
            width: 100%;
            height: 100%;
        }
    style>
head>

<body>
    <div class="image-container">
        <div class="image">
            <img data-src="/images/bg1.jpg">
        div>
        <div class="image">
            <img data-src="/images/bg2.jpg">
        div>
        <div class="image">
            <img data-src="/images/bg1.jpg">
        div>
        <div class="image">
            <img data-src="/images/bg2.jpg">
        div>
        <div class="image">
            <img data-src="/images/bg1.jpg">
        div>
        <div class="image">
            <img data-src="/images/bg2.jpg">
        div>
        <div class="image">
            <img data-src="/images/bg1.jpg">
        div>
        <div class="image">
            <img data-src="/images/bg2.jpg">
        div>
        <div class="image">
            <img data-src="/images/bg1.jpg">
        div>
        <div class="image">
            <img data-src="/images/bg2.jpg">
        div>
    div>
body>
<script>
    const imgs = document.getElementsByTagName('img');
    const clientHeight = window.innerHeight || document.documentElement.clientHeight;
    let loadedIndex = 0;
    function lazyload() {
        for (let i = loadedIndex; i < imgs.length; i++) {
            if (clientHeight - imgs[i].getBoundingClientRect().top > 0) {
                imgs[i].src = imgs[i].dataset.src;
                loadedIndex = i + 1;
            }
        }
    }
    lazyload();
    window.addEventListener('scroll', lazyload, false);
script>
html>
```

8.3.4 缓存代理

- 有些时候可以用空间换时间
- 一个正整数的阶乘（factorial）是所有小于及等于该数的正整数的积，并且0的阶乘为1

```
const factorial = function f(num) {
    if (num === 1) {
        return 1;
    } else {
        return (num * f(num - 1));
    }
}

const proxy = function (fn) {
    const cache = {};
    return function (num) {
        if (num in cache) {
            return cache[num];
        }
        return cache[num] = fn.call(this, num);
    }
}

const proxyFactorial = proxy(factorial);
console.log(proxyFactorial(5));
console.log(proxyFactorial(5));
console.log(proxyFactorial(5));
```

- 斐波那契数列(Fibonacci sequence)指的是这样一个数列：1、1、2、3、5、8、13、21、34。在数学上，斐波那契数列以如下被以递推的方法定义：F(1)=1,F(2)=1,F(n)=F(n-1)+F(n-2)（n>=3，n∈N*）

```
let count = 0;
function fib(n) {
    count++;
    return n 2 ? 1 : fib(n - 1) + fib(n - 2);
}
var result = fib(10);
console.log(result, count);
```

```
let count = 0;
const fibWithCache = (function () {
    let cache = {};
    function fib(n) {
        count++;
        if (cache[n]) {
            return cache[n];
        }
        let result = n 2 ? 1 : fib(n - 1) + fib(n - 2);
        cache[n] = result;
        return result;
    }
    return fib;
})();
var result = fibWithCache(10);
console.log(result, count);
```

8.3.5 防抖代理

- 通过防抖代理优化可以把多次请求合并为一次，提高性能
- 节流与防抖都是为了减少频繁触发事件回调
- 节流(Throttle)是在某段时间内不管触发了多少次回调都只认第一个，并在第一次结束后执行回调
- 防抖(Debounce)就是在某段时间不管触发了多少回调都只看最后一个

8.3.5.1 节流

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Documenttitle>
    <style>
        #container {
            width: 200px;
            height: 400px;
            border: 1px solid red;
            overflow: auto;
        }

        #container .content {
            height: 4000px;
        }
    style>
head>

<body>
    <div id="container">
        <div class="content">div>
    div>
    <script>
        function throttle(callback, interval) {
            let last;
            return function () {
                let context = this;
                let args = arguments;
                let now = Date.now();
                if (last) {
                    if (now - last >= interval) {
                        last = now;
                        callback.apply(context, args);
                    }
                } else {
                    callback.apply(context, args);
                    last = now;
                }

            }
        }
        let lastTime = Date.now();
        const throttle_scroll = throttle(() => {
            console.log('触发了滚动事件', (Date.now() - lastTime) / 1000);
        }, 1000);
        document.getElementsByTd('container').addEventListener('scroll', throttle_scroll);
    script>
body>

html>
```

8.3.5.2 防抖 #

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Documenttitle>
    <style>
        #container {
            width: 200px;
            height: 400px;
            border: 1px solid red;
            overflow: auto;
        }

        #container .content {
            height: 4000px;
        }
    style>
head>

<body>
    <div id="container">
        <div class="content">div>
    div>
    <script>
        function throttle(callback, delay) {
            let timer;
            return function () {
                let context = this;
                let args = arguments;
                if (timer)
                    clearTimeout(timer);
                timer = setTimeout(() => {
                    callback.apply(context, args);
                }, delay);
            }
        }
        let lastTime = Date.now();
        const throttle_scroll = throttle(() => {
            console.log('触发了滚动事件', (Date.now() - lastTime) / 1000);
        }, 1000);
        document.getElementById('container').addEventListener('scroll', throttle_scroll);
    script>
body>

html>
```

8.3.5.3 未防抖

```
<body>
    <ul id="todos">
    ul>
<script>
    let todos = document.querySelector('#todos');
    window.onload = function(){
        fetch('/todos').then(res=>res.json()).then(response=>{
            todos.innerHTML = response.map(item=>`${item.id}" type="checkbox" ${item.completed?"checked":""}/>${item.text}`).join('');
        });
    }
    function toggle(id){
       fetch(`/toggle/${id}`).then(res=>res.json()).then(response=>{
            console.log('response',response);
        });
    }
    todos.addEventListener('click',function(event){
        let checkbox = event.target;
        let id = checkbox.value;
        toggle(id);
    });
script>
body>
```

app.js

```
let express=require('express');
let app=express();
app.use(express.static(__dirname));
let todos=[
    {id: 1,text: 'a',completed: false},
    {id: 2,text: 'b',completed: false},
    {id: 3,text: 'c',completed: false},
];
app.get('/todos',function (req,res) {
    res.json(todos);
});
app.get('/toggle/:id',function (req,res) {
    let id=req.params.id;
    todos = todos.map(item => {
        if (item.id==id) {
            item.completed=!item.completed;
        }
        return item;
    });
    res.json({code:0});
});
app.listen(8080);
```

8.3.5.4 防抖

todos.html

```
<body>
    <ul id="todos">
    ul>
    <script>
    let todos = document.querySelector('#todos');
    window.onload = function(){
        fetch('/todos').then(res=>res.json()).then(response=>{
            todos.innerHTML = response.map(item=>`${item.id}" type="checkbox" ${item.completed?"checked":""}/>${item.text}`).join('');
        });
    }
    function toggle(id){
       fetch(`/toggle/${id}`).then(res=>res.json()).then(response=>{
            console.log('response',response);
        });
    }
    let LazyToggle = (function(id){
        let ids = [];
        let timer;
        return function(id){
            ids.push(id);
            if(timer){
               clearTimeout(timer);
            }
            timer = setTimeout(function(){
                toggle(ids.join(','));
                ids = null;
                clearTimeout(timer);
                timer = null;
            },2000);
        }
    })();
    todos.addEventListener('click',function(event){
        let checkbox = event.target;
        let id = checkbox.value;
        LazyToggle(id);
    });
script>
```

app.js

```
app.get('/toggle/:ids',function (req,res) {
    let ids=req.params.ids;
    ids=ids.split(',').map(item=>parseInt(item));
    todos = todos.map(item => {
        if (ids.includes(item.id)) {
            item.completed=!item.completed;
        }
        return item;
    });
    res.json({code:0});
});
```

8.3.6 代理跨域

8.3.6.1 正向代理和反向代理

- 正向代理的对象是客户端,服务器端看不到真正的客户端
- 通过公司代理服务器上网

- 反向代理的对象的服务端,客户端看不到真正的服务端
- nginx代理应用服务器

proxy-server.js

```
const http = require('http');
const httpProxy = require('http-proxy');

const proxy = httpProxy.createProxyServer();

let server = http.createServer(function (req, res) {

    proxy.web(req, res, {
        target: 'http://127.0.0.1:9999'
    });

    proxy.on('error', function (err) {
        console.log(err);
    });
});
server.listen(8888, '0.0.0.0');
```

real-server.js

```
const http = require('http');
let server = http.createServer(function (req, res) {
    res.end('9999');
});
server.listen(9999, '0.0.0.0');
```

8.3.6.2 代理跨域

- nginx代理跨域
- webpack-dev-server代理跨域
- 客户端代理跨域
- 当前的服务启动在origin(3000端口)上，但是调用的接口在target(4000端口)上
- postMessage (https://developer.mozilla.org/zh-CN/docs/Web/API/Window/postMessage)方法可以安全地实现跨源通信
    - otherWindow:其他窗口的一个引用 message:将要发送到其他window的数据
    - message 将要发送到其他window的数据
    - targetOrigin通过窗口的origin属性来指定哪些窗口能接收到消息事件，其值可以是字符串"*"（表示无限制）或者一个URI

```
otherWindow.postMessage(message, targetOrigin, [transfer]);
```

- data 从其他 window中传递过来的对象
- origin 调用 postMessage时消息发送方窗口的 origin
- source 对发送消息的窗口对象的引用

```
window.addEventListener("message", receiveMessage, false);
```

origin.js

```
let express=require('express');
let app=express();
app.use(express.static(__dirname));
app.listen(3000);
```

target.js

```
let express = require('express');
let app = express();
let bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static(__dirname));
let users = [];
app.post('/register', function (req, res) {
    let body = req.body;
    let target = body.target;
    let callback = body.callback;
    let username = body.username;
    let password = body.password;
    let user = { username, password };
    let id = users.length == 0 ? 1 : users[users.length - 1].id + 1;
    user.id = id;
    users.push(user);
    res.status(302);
    res.header('Location', `${target}?callback=${callback}&args=${id}`);
    res.end();
});
app.listen(4000);
```

reg.html

```

<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Documenttitle>
head>

<body>
    <script type="text/javascript">
        window.addEventListener('message', function (event) {
            console.log(event.data);

            if (event.data.receiveId) {
                alert('用户ID=' + event.data.receiveId);
            }
        })
    script>
    <iframe name="proxyIframe" id="proxyIframe" frameborder="0">iframe>
    <form action="http://localhost:4000/register" method="POST" target="proxyIframe">
        <input type="hidden" name="callback" value="receiveId">
        <input type="hidden" name="target" value="http://localhost:3000/target.html">
        用户名<input type="text" name="username" />
        密码<input type="text" name="password" />
        <input type="submit" value="提交">
    form>
body>

html>
```

target.html

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Documenttitle>
head>

<body>
    <script>
        window.onload = function () {
            var query = location.search.substr(1).split('&');
            let callback, args;
            for (let i = 0, len = query.length; i < len; i++) {
                let item = query[i].split('=');
                if (item[0] == 'callback') {
                    callback = item[1];
                } else if (item[0] == 'args') {
                    args = item[1];
                }
            }
            try {
                window.parent.postMessage({ [callback]: args }, '*');
            } catch (error) {
                console.log(error);
            }
        }
    script>
body>

html>
```

8.3.7 $.proxy#

- 接受一个函数，然后返回一个新函数，并且这个新函数始终保持了特定的上下文语境。
- jQuery.proxy(function, context) function为执行的函数，content为函数的上下文this值会被设置成这个object对象

```
    jquery proxy

    点我变红

        let btn = document.getElementById('btn');
        btn.addEventListener('click', function () {
            setTimeout($.proxy((function () {
                $(this).css('color', 'red');
            }), this), 1000);
        });
```

```
function proxy(fn, context) {
    return function () {
       return fn.call(context, arguments);
    }
}
```

8.3.8 Proxy#

- Proxy 用于修改某些操作的默认行为
- Proxy 可以理解成，在目标对象之前架设一层 拦截,外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。
- Proxy 这个词的原意是代理，用在这里表示由它来 代理某些操作，可以译为 代理器
- Proxy (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy)
- defineProperty (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty)

```
let wang={
    name: 'wanglaoshi',
    age: 29,
    height:165
}
let wangMama=new Proxy(wang,{
    get(target,key) {
        if (key == 'age') {
            return wang.age-1;
        } else if (key == 'height') {
            return wang.height-5;
        }
        return target[key];
    },
    set(target,key,val) {
        if (key == 'boyfriend') {
            let boyfriend=val;
            if (boyfriend.age>40) {
                throw new Error('太老');
            } else if (boyfriend.salary<20000) {
                throw new Error('太穷');
            } else {
                target[key]=val;
                return true;
            }
        }
    }
});
console.log(wangMama.age);
console.log(wangMama.height);
wangMama.boyfriend={
    age: 41,
    salary:3000
}
```

8.3.9 Vue2和 Vue3#

- Vue2 中的变化侦测实现对 Object 及 Array 分别进行了不同的处理,Object 使用了 Object.defineProperty API, Array使用了拦截器对 Array 原型上的能够改变数据的方法进行拦截。虽然也实现了数据

的变化侦测，但存在很多局限，比如对象新增属性无法被侦测，以及通过数组下边修改数组内容，也因此在 Vue2 中经常会使用到 `$set` 这个方法对数据修改，以保证依赖更新。
- Vue3 中使用了 es6 的 `Proxy API`对数据代理，没有像 `Vue2` 中对原数据进行修改，只是加了代理包装，因此首先性能上会有所改善。其次解决了 `Vue2` 中变化侦测的局限性，可以不使用 `$set` 新增的对象属性及通过下标修改数组都能被侦测到。

8.3.10

- 代理模式 VS 适配器模式 适配器提供不同接口，代理模式提供一模一样的接口
- 代理模式 VS 装饰器模式 装饰器模式原来的功能不变还可以使用，代理模式改变原来的功能

9.观察者模式

- 观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个目标对象，当这个目标对象的状态发生变化时，会通知所有观察者对象，使它们能够自动更新
- 双十一加入购物车

** 9.1 类图 #**

- 主题对象(Subject) 该角色又称为被观察者,可以增加和删除观察者对象,它将有关状态存入具体观察者对象，在具体主题的内部状态改变时，给所有登记过(关联了观察关系)的观察者发出通知
- 观察者(Observer)角色：定义一个接收通知的接口(update),在得到主题的通知时更新自己

** 9.2 代码 #**

```
abstract class Student {
    constructor(public teacher: Teacher) { }
    public abstract update();
}
class Xueba extends Student {
    public update() {
        console.log(this.teacher.getState() + ',学霸抬头举手');
    }
}
class Xuezha extends Student {
    public update() {
        console.log(this.teacher.getState() + ',学渣低头祈祷');
    }
}

class Teacher {
    private students: Student[] = new Array();
    public state: string = '老师讲课'
    getState() {
        return this.state;
    }
    public askQuestion() {
        this.state = '老师提问';
        this.notifyAllStudents();
    }
    attach(student: Student) {
        this.students.push(student);
    }
    notifyAllStudents() {
        this.students.forEach(student => student.update());
    }
}
let teacher = new Teacher();
teacher.attach(new Xueba(teacher));
teacher.attach(new Xueza(teacher));
teacher.askQuestion();
```

** 9.3 场景 #**

9.3.1 DOM事件绑定

原生

```
    观察者模式

    click

        let btn = document.getElementById('btn');
        const handler1 = () => { console.log(1); }
        const handler2 = () => { console.log(2); }
        const handler3 = () => { console.log(3); }
        btn.addEventListener('click', handler1);
        btn.addEventListener('click',handler2);
        btn.addEventListener('click', handler3);
```

jquery

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>观察者模式title>
head>

<body>
    <script src="http://libs.baidu.com/jquery/2.0.0/jquery.min.js">script>
    <button id="btn">clickbutton>
    <script>
        [illegible]
    script>
body>

html>
```

9.3.2 Promise

```
class Promise {
    private callbacks: Array<Function> = []
    constructor(fn) {
        let resolve = () => {
            this.callbacks.forEach(callback => callback())
        };
        fn(resolve);
    }
    then(callback) {
        this.callbacks.push(callback);
    }
}
let promise = new Promise(function (resolve) {
    setTimeout(function () {
        resolve(100);
    }, 1000);
});
promise.then(() => console.log(1));
promise.then(() => console.log(2));
```

9.3.3 callbacks

- JQuery.Callbacks是jQuery1.7+之后引入的，用来进行函数队列的add、remove、fire、lock等操作
- Callbacks对象其实就是一个函数队列，获得Callbacks对象之后，就可以向这个集合中增加或者删除函数。add和remove功能相反，函数参数是相同的，empty()删除回调列表中的所有函数

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Documenttitle>
head>

<body>
    <script src="http://libs.baidu.com/jquery/2.0.0/jquery.min.js">script>
    <script>
        [illegible]
    script>
body>
html>
```

Callbacks

```
function Callbacks() {
    let observers = [];
    function add(observer) {
        observers.push(observer);
    }
    function remove(observer) {
        let index = observers.indexOf(observer);
        if (index != -1)
            observers.splice(index, 1);
    }
    function fire() {
        observers.forEach(observer => observer());
    }
    return {
        add,
        remove,
        fire
    }
}
```

9.3.4 EventEmitter

自定义事件

```
const EventEmitter = require('events');
let subject = new EventEmitter();
subject.on('click', function (name) {
    console.log(1, name);
});
subject.on('click', function (name) {
    console.log(2, name);
});
subject.emit('click', 'zhufeng');
```

events.js

```
class EventEmitter{
    constructor() {
        this._events={};
    }
    on(type,listener) {
        let listeners=this._events[type];
        if (listeners) {
            listeners.push(listener);
        } else {
            this._events[type]=[listener];
        }
    }
    emit(type) {
        let listeners=this._events[type];
        let args=Array.from(arguments).slice(1);
        listeners.forEach(listener => listener(...args));
    }
}
module.exports = EventEmitter;
```

9.3.5 流

```
let fs = require('fs');
let rs = fs.createReadStream('./1.txt', { highWaterMark: 3 });
rs.on('data', function (data) {
    console.log(data.toString());
});
rs.on('end', function () {
    console.log('end');
});
```

9.3.6 http服务器

```
let http = require('http');
let server = http.createServer();
server.on('request', (req, res) => {
    res.end('zhufeng');
});
server.listen(3000);
```

9.3.7 生命周期函数

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Documenttitle>
head>

<body>
    <div id="root">div>
    <script>
        let container = document.getElementById('root');
        class Component {
            state = { number: 0 }
            componentWillMount() {
                console.log('componentWillMount');
            }

            componentDidMount() {
                console.log('componentDidMount');
            }
            shouldComponentUpdate() {
                console.log('shouldComponentUpdate');
                return true;
            }
            componentWillUpdate() {
                console.log('componentWillUpdate');
            }

            componentDidUpdate() {
                console.log('componentDidUpdate');
            }
            setState(newState) {
                this.state = { ...this.state, ...newState }
                if (!this.shouldComponentUpdate || (this.shouldComponentUpdate && this.shouldComponentUpdate())) {
                    if (this.componentWillUpdate)
                        this.componentWillUpdate();
                    let content = this.render();
                    container.innerHTML = content;
                    if (this.componentDidUpdate)
                        this.componentDidUpdate();
                }
            }
            render() {
                console.log('render');
                return this.state.number;
            }
        }
        render(Component, document.getElementById('root'));
        function render(Component, container) {

            let component = new Component();
            if (component.componentWillMount)
                component.componentWillMount();
            let content = component.render();
            container.innerHTML = content;
            if (component.componentDidMount)
                component.componentDidMount();

            setTimeout(() => {
                component.setState({ number: 1 });
            }, 3000);
        }

    script>
body>

html>
```

9.3.8 EventBus

- 如果你觉得使用 $on、 $emit不方便,而你又不愿意引入 vuex,可以使用 EventBus
- 在要相互通信的兄弟组件之中，都引入一个新的vue实例，然后通过分别调用这个实例的事件触发和监听来实现通信和参数传递

EventBus.js

```
import Vue from 'vue';
export default new Vue();
```

组件A

```
import EventBus from './EventBus';
EventBus.$on("customEvent", name => {
    console.log(name);
})
```

组件B

```
import EventBus from './EventBus';
EventBus.$emit("customEvent", 'zhufeng')
```

```
<body>
    <script src="https://cdn.bootcss.com/vue/2.5.17/vue.js">script>
    <script>
        let EventBus = new Vue();
        EventBus.$on("customEvent", name => {
            console.log(name);
        })
        EventBus.$emit("customEvent", 'zhufeng')
    script>
body>
```

9.3.9 Vue响应式

- 在 Vue 中，每个组件实例都有相应的 `watcher` 实例对象，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 `setter` 被调用时，会通知 `watcher` 重新计算，从而致使它关联的组件得以更新

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>vuetitle>
head>

<body>
    <div id="name">div>
    <div id="age">div>
    <script>
        let name = document.getElementById('name');
        let age = document.getElementById('age');
        class Dep {
            subs = []
            addSub(sub) {
                this.subs.push(sub);
            }
            notify() {
                this.subs.forEach((sub) => sub())
            }
        }
        function observe(target) {
            Object.keys(target).forEach((key) => {
                let val = target[key];
                const dep = new Dep();
                if (key == 'name') {
                    name.innerHTML = val;
                    dep.addSub(() => { name.innerHTML = val });
                } else if (key == 'age') {
                    age.innerHTML = val;
                    dep.addSub(() => { age.innerHTML = val });
                }
                Object.defineProperty(target, key, {
                    get: function () {
                        return val;
                    },
                    set: function (value) {
                        val = value;
                        dep.notify();
                    }
                });
            })
        }

        let obj = { name: '名称', age: '年龄' };
        observe(obj);
        setTimeout(() => {
            obj.name = '新名称';
        }, 3000);
        setTimeout(() => {
            obj.age = '新年龄';
        }, 6000);
    script>
body>
html>
```

9.3.9 redux

- createStore (https://github.com/reduxjs/redux/blob/master/src/createStore.js)

```
export default function createStore(reducer,preloadedState,enhancer) {
    if (enhancer) {
        return enhancer(createStore)(reducer,preloadedState);
    }
    let state=preloadedState;
    let listeners=[];
    function getState() {
        return state;
    }
    function subscribe(listener) {
        listeners.push(listener);
        return function () {
            const index=listeners.indexOf(listener);
            listeners.splice(index,1);
        }
    }
    function dispatch(action) {
        state=reducer(state,action);
        listeners.forEach(listener=>listener())
    }
    dispatch({type:'@@redux/INIT'});
    return {
        dispatch,
        subscribe,
        getState
    }
}
```

** 9.4 发布订阅模式 #**

9.4.1 区别

- 订阅者把自己想订阅的事件注册到调度中心
- 当该事件触发时候，发布者发布该事件到调度中心,由调度中心统一调度订阅者注册到调度中心的处理代码。
- 虽然两种模式都存在订阅者和发布者（观察者可认为是订阅者、被观察者可认为是发布者）
- 但是观察者模式是由被观察者调度的，而发布/订阅模式是统一由调度中心调的
- 所以观察者模式的订阅者与发布者之间是存在依赖的，而发布/订阅模式则不会。

9.4.2 类图

9.4.3 代码

```
class Agency {
    _topics = {}
    subscribe(topic, listener) {
        let listeners = this._topics[topic];
        if (listeners) {
            listeners.push(listener);
        } else {
            this._topics[topic] = [listener];
        }
    }
    publish(topic, ...args) {
        let listeners = this._topics[topic] || [];
        listeners.forEach(listener => listener(...args));
    }
}
class Landlord {
    constructor(public agent: Agency) { }
    lend(topic, area, money) {
        this.agent.publish(topic, area, money);
    }
}

class Tenant {
    constructor(public agent: Agency, public name: string) { }
    order(topic) {
        this.agent.subscribe(topic, (area, money) => {
            console.log(this.name, `${area}平米, ${money}元`);
        });
    }
}
let agent = new Agency();
let rich = new Tenant(agent, '大款');
let poor = new Tenant(agent, '北漂');
let landlord = new Landlord(agent);
rich.order('豪宅');
poor.order('单间');
landlord.lend('豪宅', 10000, 1000000);
landlord.lend('单间', 10, 1000);
```

9.4.4 Redis发布订阅

- Redis 发布订阅是一种消息通信模式：发送者发送消息，订阅者接收消息，客户端可以订阅任意数量的频道。

```
SUBSCRIBE channel_a
PUBLISH channel_a zhufeng
```

```
let redis = require('redis');
let client1 = redis.createClient(6379, '127.0.0.1');
let client2 = redis.createClient(6379, '127.0.0.1');

client1.subscribe('channel_a');
client1.subscribe('channel_b');
client1.on('message', (channel, message) => {
    console.log('client1', channel, message);
    client1.unsubscribe('channel_a');
});
client2.publish('channel_a', 'a_hello');
client2.publish('channel_b', 'b_hello');

setTimeout(() => {
    client2.publish('channel_a', 'a_world');
    client2.publish('channel_b', 'b_world');
}, 3000);
```

10. 外观模式

- 外观模式(Facade Pattern)又叫门面模式，定义一个将子系统的一组接口集成在一起的高层接口，以提供一个一致的外观
- 外观模式让外界减少与子系统内多个模块的直接交互，从而减少耦合，让外界可以更轻松地使用子系统
- 该设计模式由以下角色组成
    - 门面角色：外观模式的核心。它被客户角色调用,它熟悉子系统的功能。内部根据客户角色的需求预定了几种功能的组合
    - 子系统角色:实现了子系统的功能。它对客户角色和 Facade是未知的
    - 客户角色:通过调用Facede来完成要实现的功能
- 遥控器、自动驾驶汽车、房屋中介

** 10.1 计算器 #**

```
class Sum {
    sum(a, b) {
        return a + b;
    }
}
class Minus {
    minus(a, b) {
        return a - b;
    }
}
class Multiply {
    multiply(a, b) {
        return a * b;
    }
}
class Calculator {
    sumObj
    minusObj
    multiplyObj
    constructor() {
        this.sumObj = new Sum();
        this.minusObj = new Minus();
        this.multiplyObj = new Multiply();
    }
    sum(...args) {
        return this.sumObj.sum(...args);
    }
    minus(...args) {
        return this.minusObj.minus(...args);
    }
    multiply(...args) {
        return this.multiplyObj.multiply(...args);
    }
}
let calculator = new Calculator();
console.log(calculator.sum(1, 2));
console.log(calculator.minus(1, 2));
console.log(calculator.multiply(1, 2));
```

** 10.2 计算机 #**

```

class CPU {
    startup() { console.log('打开CPU'); }
    shutdown() { console.log('关闭CPU'); }
}
class Memory {
    startup() { console.log('打开内存'); }
    shutdown() { console.log('关闭内存'); }
}
class Disk {
    startup() { console.log('打开硬盘'); }
    shutdown() { console.log('关闭硬盘'); }
}
class Computer {
    cpu;
    memory;
    disk;
    constructor() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.disk = new Disk();
    }
    startup() {
        this.cpu.startup();
        this.memory.startup();
        this.disk.startup();
    }
    shutdown() {
        this.cpu.shutdown();
        this.memory.shutdown();
        this.disk.shutdown();
    }
}
let computer = new Computer();
computer.startup();
computer.shutdown();
```

** 10.3 压缩 #**

```
export { }
var zlib = require('zlib');
var fs = require('fs');
let path = require('path');
function open(input) {
    let ext = path.extname(input);
    switch (ext) {
        case '.gz':
            return unZip(input);
        case '.rar':
            return unRar(input);
        case '.7z':
            return un7z(input);
        default:
            break;
    }
}
function unZip(src) {
    var gunzip = zlib.createGunzip();
    var inputStream = fs.createReadStream(src);
    var outputStream = fs.createWriteStream(src.slice(0, -3));
    console.log('outputStream');

    inputStream.pipe(gunzip).pipe(outputStream);
}
function unRar(src) {
    console.log('Rar解压后的', src);
}
function un7z(src) {
    console.log('7z解压后的', src);
}
open('./source.txt.gz');

function zip(src) {
    var gzip = zlib.createGzip();
    var inputStream = fs.createReadStream(src);
    var outputStream = fs.createWriteStream(src+'.gz');
    inputStream.pipe(gzip).pipe(outputStream);
}
zip('source.txt');
```

** 10.4 redux #**

- redux index.js (https://github.com/reduxjs/redux/blob/v4.0.0/src/index.js)

** 10.5 函数重载 #**

- 为复杂的模块或子系统提供外界访问的模块
- 子系统相互独立

10.5.1 参数重载

sum.ts

```
function sum(a: number, b: string);
function sum(a: string, b: number);
function sum(a: any, b: any) {
    return a + b;
}
```

10.5.2 react createElement

-react create-element (https://github.com/facebook/react/blob/master/packages/react/src/ReactElement.js#L347-L361)

```
export function createElement (
  context: Component,
  tag: any,
  data: any,
  children: any,
  normalizationType: any,
  alwaysNormalize: boolean
): VNode | Array<VNode> {
  if (Array.isArray(data) || isPrimitive(data)) {
    normalizationType = children
    children = data
    data = undefined
  }
  if (isTrue(alwaysNormalize)) {
    normalizationType = ALWAYS_NORMALIZE
  }
  return _createElement(context, tag, data, children, normalizationType)
}
```

10.5.3 vue createElement

-vue create-element (https://github.com/vuejs/vue/blob/v2.6.10/src/core/vdom/create-element.js#L28-L45)

```
  const childrenLength = arguments.length - 2;
  if (childrenLength === 1) {
    props.children = children;
  } else if (childrenLength > 1) {
    const childArray = Array(childrenLength);
    for (let i = 0; i < childrenLength; i++) {
      childArray[i] = arguments[i + 2];
    }
    if (__DEV__) {
      if (Object.freeze) {
        Object.freeze(childArray);
      }
    }
    props.children = childArray;
  }
```

10.5.4 buffer

buffer.js (https://github.com/nodejs/node/blob/master/lib/buffer.js#L1082)

```
Buffer.prototype.slice = function slice(start, end) {
  const srcLength = this.length;
  end = end !== undefined ? srcLength: srcLength;
  const newLength = end > start ? end - start : 0;
  return new FastBuffer(this.buffer, this.byteOffset + start, newLength);
};
```

10.5.5 createStore

- createStore (https://github.com/reduxjs/redux/blob/e95eaf2dc2024fe99dc0f7334a8bd049b4949ed0/src/createStore.js#L31-L35)

```
export default function createStore(reducer, preloadedState, enhancer) {
  if (typeof preloadedState === 'function' && typeof enhancer === 'undefined') {
    enhancer = preloadedState
    preloadedState = undefined
  }
}
```

10.5.6 axios

- defaults (https://github.com/axios/axios/blob/v0.19.0/lib/defaults.js#L16-L27)

```
function getDefaultAdapter() {
  var adapter;
  if (typeof process !== 'undefined' && Object.prototype.toString.call(process) === '[object process]') {
    adapter = require('./adapters/http');
  } else if (typeof XMLHttpRequest !== 'undefined') {
    adapter = require('./adapters/xhr');
  }
  return adapter;
}
```

10.5.7 polyfill

- polyfill可以让我们处理浏览器兼容和屏蔽了浏览器差异
- removeEvent (https://github.com/jquery/jquery/blob/1.5/src/event.js#L584-L596)

```
jQuery.removeEvent = document.removeEventListener ?
    function( elem, type, handle ) {
        if ( elem.removeEventListener ) {
            elem.removeEventListener( type, handle, false );
        }
    } :
    function( elem, type, handle ) {
        if ( elem.detachEvent ) {
            elem.detachEvent( "on" + type, handle );
        }
    };
```

11. 迭代器模式

- 迭代器模式(Iterator Pattern)用于顺序地访问聚合对象内部的元素，又无需知道对象内部结构。使用了迭代器之后，使用者不需要关心对象的内部构造，就可以按序访问其中的每个元素。

11.1 类图

11.2 代码

```
function createIterator(arr) {
    let index=0;
    return {
        next() {
            return indexvalue: arr[index++],done: false}:
                {done:true}
        }
    }
}
let it=createIterator([1,2]);
console.log(it.next());
console.log(it.next());
console.log(it.next());
```

11.3 场景

** 11.3.1 forEach #**

- 可以使用 for 循环自己实现一个 `forEach`

```
Array.prototype.forEach = function (cb) {
  for (var i = 0; i < this.length; i++) {
      cb.call(this, this[i], i, arr);
  }
}
let arr = [1, 2, 3];
arr.forEach((item) => {
  console.log(item);
});
```

** 11.3.2 each #**

- jquery (https://github.com/jquery/jquery/blob/3.4.1/src/core.js#L246-L265)
- underscore (https://github.com/jashkenas/underscore/blob/1.9.1/underscore.js#L181-L195)

```
<body>
    <script src="http://libs.baidu.com/jquery/2.0.0/jquery.min.js">script>
    <script>

    2.each(['1', '2', '3'], function (index, item) {
        console.log(index, item);
    })
    2.each({ name: 'zhufeng', age: 10 }, function (index, item) {
        console.log(index, item);
    })
    function each(obj, callback) {
        if (Array.isArray(obj)) {
            let length = obj.length;
            for (let i = 0; i < length; i++) {
                if (callback.call(obj[i], i, obj[i]) === false) {
                    break;
                }
            }
        } else {
            for (let i in obj) {
                if (callback.call(obj[i], i, obj[i]) === false) {
                    break;
                }
            }
        }

        return obj;
    }

    script>

body>
```

** 11.3.3 Iterator #**

- 在ES6中有序集合数据类型有Array、Map、Set、String、TypedArray、arguments、NodeList
- 我们需要有一个统一的遍历接口来遍历所有的数据类型
- 他们都有 [Symbol.Iterator]属性，属性是一个函数，执行函数会返回迭代器
- 迭代器就有 next方法顺序返回子元素

```
Array[Symbol.iterator] = function () {
    let index = 0;
    return {
        next: () => {
            return index < this.length ?
                { value: this[index++], done: false } :
                { done: true }
        }
    }
}
let arr = [1, 2];
let it = arr[Symbol.iterator]();
console.log(it.next());
console.log(it.next());
console.log(it.next());
```

** 11.3.4 yield* #**

- yield*后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口

```
let generator = function* () {
    yield 1;
    yield* [2, 3];
    yield 4;
};

var iterator = generator();

console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

** 11.3.5 二叉树遍历 #**

- 二叉树是每个结点最多有两个子树的树结构。通常子树被称作左子树和右子树
- 根据根据根节点的顺序可以把遍历分为三种前序、中序、后序
    - 先序遍历：根节点->左子树->右子树
    - 中序遍历：左子树->根节点->右子树
    - 后序遍历：左子树->右子树->根节点

```
class Tree {
    constructor(public left, public value, public right) {
    }
}

function* leftOrder(tree) {
    if (tree) {
        yield tree.value;
        yield* leftOrder(tree.left);
        yield* leftOrder(tree.right);
    }
}

function* inOrder(tree) {
    if (tree) {
        yield* inOrder(tree.left);
        yield tree.value;
        yield* inOrder(tree.right);
    }
}

function* rightOrder(tree) {
    if (tree) {
        yield* rightOrder(tree.left);
        yield* rightOrder(tree.right);
        yield tree.value;
    }
}
function make(array) {
    if (array.length === 1) return new Tree(null, array[0], null);
    return new Tree(make(array[0]), array[1], make(array[2]));
}
let tree = make([[['D'], 'B', ['E']], 'A', [['F'], 'C', ['G']]]);
var result: any[] = [];
for (let node of rightOrder(tree)) {
    result.push(node);
}
console.log(result);
```

```
"downlevelIteration": true, / Provide full support for iterables in 'for-of', spread, and destructuring when targeting 'ES5' or 'ES3'./
```