5.15面试题

Ts强类型有什么用

http状态码有哪些： 301和302的区别

1XX: 表示请求已被接受，但需要后续处理

2XX: 表示请求已经成功被服务器接收，理解、并接受

3XX: 表示需要客户端采取进一步的操作才能完成请求。通常这些状态码用来重定向

301被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个URI之一。如果该请求不是GET/HEAD， 浏览器通常会要求用户确认重定向。301通常用于网站迁移时，服务器对旧的URL进行301重定向到新的URL。这样搜索引擎可以正确地更新原有的页面排名等信息。

302请求的资源现在临时从不同的URI响应请求。除非指定了Cache-Control或Expires，否则该响应不可缓存。 如果当前请求非HEAD或GET，浏览器需取得用户确认，再进行重定向。

4XX: 这类的状态码代表了客户端看起来可能发生了错误，妨碍了服务器的处理。 除非响应的是一个HEAD请求，否则服务器就应该返回一个解释当前错误状况的实体。

400由于包含语法错误，当前请求无法被服务器理解。400通常在服务器端表单验证失败时返回。

401 当前请求需要用户验证，响应中会包含一个WWW-Authenticate字段来询问用户的授权信息。 而客户端的下次请求需要提供包含Authorization头的请求。

403 服务器已经理解请求，但是拒绝执行它。与401响应不同的是，身份验证并不能提供任何帮助。

404 请求所希望得到的资源未被在服务器上发现

405 请求行中指定的请求方法不能被用于请求相应的资源。在Web开发中通常是因为客户端和服务器的方法不一致，比如客户端通过PUT来修改一个资源，而服务器把它实现为POST方法。 开发中统一规范就好了。

5XX: 这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。 并且响应消息体中应当给出理由，除非是HEAD请求。

500 通常是代码出错，后台Bug。一般的Web服务器通常会给出抛出异常的调用堆栈。

502 作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。如果你在用HTTP代理来翻墙，或者你配置了nginx来反向代理你的应用，你可能会常常看到它。

504 作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器收到响应。注意与502的区别：502是接收到了无效响应比如Connection Refused； 504是响应超时，通常是被墙了。

当浏览器再次访问一个已经访问过的资源时，它会这样做：

看看是否命中强缓存，如果命中，就直接使用缓存了；

如果没有命中强缓存，就发请求到服务器检查是否命中协商缓存；

如果命中协商缓存，服务器会返回 304 告诉浏览器使用本地缓存；

否则，返回最新的资源。

什么情况会出现跨域，怎么解决跨域的，proxy跨域实现的原理

浏览器同源策略，协议、域名、端口不同将产生跨域。可以用proxy实现。

devServer: {

host: 'localhost',

port: '3011',

proxy: [

{

context: context,

target: 'https://www.xxx.com',

secure: false

}

]

}

Proxy：

const handler = {

get: function(obj, prop) {

return prop in obj ? obj[prop] : 37;

}

};

const p = new Proxy({}, handler);

p.a = 1;

p.b = undefined;

console.log(p.a, p.b); // 1, undefined

console.log('c' in p, p.c); // false, 37

let target = {};

let p = new Proxy(target, {});

p.a = 37; // 操作转发到目标

console.log(target.a); // 37. 操作已经被正确地转发

vue中 computed和method和watch有什么区别

**相同**： computed和watch都起到监听/依赖一个数据，并进行处理的作用

**异同**：它们其实都是vue对监听器的实现，只不过computed主要用于对同步数据的处理，watch则主要用于观测某个值的变化去完成一段开销较大的复杂业务逻辑。能用computed的时候优先用computed，避免了多个数据影响其中某个数据时多次调用watch的尴尬情况。

firstName: 'Foo',

lastName: 'Bar',

fullName: 'Foo Bar'

watch: {

firstName: function (val) {

this.fullName = val + ' ' +this.lastName

},

lastName: function (val) {

this.fullName = this.firstName + '' + val

}

}

fullName: function () { //使用计算属性

return this.firstName + ' ' +this.lastName

}

Computed除了默认的get还有set方法，类似函数默认的set() get()方法

vue组件中实现通信的几种方法 两个毫无关联的组件怎么实现通信(vuex,eventHub)

涉及到非父子关系的组件，例如兄弟关系、祖孙关系，甚至更远的关系；

他们之间如果有数据交互，那么应该使用Vuex来实现；

如果页面复杂度比较低的话，也可以考虑使用 global-event-bus 来实现；

如果只是父子关系的组件数据交互，那么应该考虑使用props进行单向传递；

如果涉及到子组件向父组件的数据传递，那么应该考虑使用 $emit 和 $on；

vuex两种修改state的方法有什么区别

使用dispatch 和 commit的区别在于，前者是异步操作，后者是同步操作，所以 一般情况下，推荐直接使用commit，

即 this.$store.commit(commitType, payload)，以防异步操作会带来的延迟问题。

算法

var obj = [

{ id:3, parent:2 },

{ id:1, parent:null },

{ id:2, parent:1 },

]转成

o = {

obj: {

id: 1,

parent: null,

child: {

id: 2,

parent: 1,

child: {

id: ,3,

parent: 2

}

}

}

}

function treeObj(obj) {

return obj.sort((a, b) => b.parent - a.parent)

.reduce((acc, cur) => (acc ? { ...cur, child: acc } : cur));

}

拼多多算法题

求字符串数组的最长公共前缀

比如输入: ["flower","flow","flight"]，输出: "fl"

var longestCommonPrefix = function(strs) {

if (strs === null || strs.length === 0) return "";

if(strs.length === 1) return strs[0]

let min = 0, max = 0

for(let i = 1; i < strs.length; i++) {

if(strs[min] > strs[i]) min = i

if(strs[max] < strs[i]) max = i

}

for(let j = 0; j < strs[min].length; j++) {

if(strs[min].charAt(j) !== strs[max].charAt(j)) {

return strs[min].substring(0, j)

}

}

return strs[min]

};

解题思路： 最长公共前缀只需要比较最大值的字符串和最小值的字符串，记一下min和max ，求这两个字符串的最长公共前缀就行

css实现垂直居中的几种方法

怎么实现继承（用class和不用class）

定义在构造函数原型对象上的属性和方法虽然不能直接表现在实例对象上，但是实例对象却可以访问或者调用它们

**把客观事物封装成抽象的类，隐藏属性和方法，仅对外公开接口**

**(一) 私有属性、公有属性、静态属性概念：**

* 私有属性和方法：只能在构造函数内访问不能被外部所访问(在构造函数内使用var声明的属性)，见题1.1
* 公有属性和方法(或实例方法)：对象外可以访问到对象内的属性和方法(在构造函数内使用this设置，或者设置在构造函数原型对象上比如Cat.prototype.xxx)，见题1.2
* 静态属性和方法：定义在构造函数上的方法(比如Cat.xxx)，不需要实例就可以调用(例如Object.assign())

**(二) 实例对象上的属性和构造函数原型上的属性：**

* 定义在**构造函数原型对象上的属性和方法**虽然不能直接表现在实例对象上，但是实例对象却可以访问或者调用它们。
* 当访问一个对象的属性 / 方法时，它不仅仅在该对象上查找，还会查找该对象的原型，以及该对象的原型的原型，一层一层向上查找，直到找到一个名字匹配的属性 / 方法或到达原型链的末尾（null）。

**(三) 遍历实例对象属性的三种方法:**

* 使用for...in...能获取到实例对象自身的属性和原型链上的属性
* 使用Object.keys()和Object.getOwnPropertyNames()只能获取实例对象自身的属性
* 可以通过.hasOwnProperty()方法传入属性名来判断一个属性是不是实例自身的属性

原型式继承  
function objcet (obj) {

function F () {};

F.prototype = obj;

F.prototype.constructor = F;

return new F();

}

和Object.create()一样

原型链继承 Child.prototype = **new** Parent()

将子类的原型对象指向父类的实例。借助原型可以基于已有的对象创建新对象， 同时还不必因此创建自定义类型.

Object.create()与其类似

function Parent (name) {

this.name = name

this.sex = 'boy'

this.colors = ['white', 'black']

}

function Child () {

this.feature = ['cute']

}

var parent = new Parent('parent')

Child.prototype = parent

var child1 = new Child('child1')

child1.sex = 'girl'

child1.colors.push('yellow')

child1.feature.push('sunshine')

var child2 = new Child('child2')

console.log(child1)

console.log(child1.colors)

console.log(child2)

console.log(child1.name)

console.log(child2.colors)

console.log(parent)

**优点：**

* 继承了父类的模板，又继承了父类的原型对象

**缺点：**

* 如果要给子类的原型上新增属性和方法，就必须放在Child.prototype = new Parent()这样的语句后面
* 无法实现多继承(因为已经指定了原型对象了)
* 来自原型对象的所有属性都被共享了，这样如果不小心修改了原型对象中的引用类型属性，那么所有子类创建的实例对象都会受到影响(这点从修改child1.colors可以看出来)
* 创建子类时，无法向父类构造函数传参数(这点从child1.name可以看出来)

借用构造函数实现继承

**function** **Child** () {

Parent.call(**this**, ...arguments)

}

在子类构造函数内部使用call或apply来调用父类构造函数，通过call()方法改变Child1的this指向使子类的函数体内执行父级的构造函数从而实现继承效果

解决了原型链继承中子类实例共享父类引用对象的问题，实现多继承，创建子类实例时，可以向父类传递参数

总结：构造函数继承法只能实现部分继承，如果我们在父类Parent1的原型链上添加属性或者方法的时候子类的实例无法继承到。

function Parent1 () {

this.name = '陈楚',

this.age = 18

}

// 定义子类

function Child1 () {

//通过call()方法改变Child1的this指向使子类的函数体内执行父级的构造函数从而实现继承效果

Parent1.call(this)

this.address = '洪山区'

}

// 构建子类的实例s1

var s1 = new Child1()

console.log(s1.name) //陈楚

// 父类添加say方法

Parent1.prototype.say = function () {

console.log('say bye bye')

}

// 子类中直接打印这个say方法

console.log(s1.say()) //报错

组合继承

function Child () { Parent.call(this, ...arguments) }

Child.prototype = new Parent()// 原型链继承

Child.prototype.constructor = Child// 修正constructor

function Parent (name, colors) {

this.name = name

this.colors = colors

}

Parent.prototype.features = ['cute']

function Child (name, colors) {

Parent.apply(this, [name, colors])

}

Child.prototype = new Parent()

Child.prototype.constructor = Child

var child1 = new Child('child1', ['white'])

child1.colors.push('yellow')

child1.features.push('sunshine')

var child2 = new Child('child2', ['black'])

console.log(child1.colors)

console.log(child2.colors)

console.log(child1.features)

console.log(child2.features)

实现方式：

使用原型链继承来保证子类能继承到父类原型中的属性和方法

使用构造继承来保证子类能继承到父类的实例属性和方法

优点：

可以继承父类实例属性和方法，也能够继承父类原型属性和方法

弥补了原型链继承中引用属性共享的问题

可传参，可复用

缺点：

使用组合继承时，父类构造函数会被调用两次

并且生成了两个实例，子类实例中的属性和方法会覆盖子类原型(父类实例)上的属性和方法，所以增加了不必要的内存。

constructor总结：

constructor它是构造函数原型对象中的一个属性，正常情况下它指向的是原型对象。

它并不会影响任何JS内部属性，只是用来标示一下某个实例是由哪个构造函数产生的而已。

如果我们使用了原型链继承或者组合继承无意间修改了constructor的指向，那么出于编程习惯，我们最好将它修改为正确的构造函数。

寄生组合式继承

// 构造继承 function Child () { Parent.call(this, ...arguments) }

// 原型式继承 Child.prototype = Object.create(Parent.prototype)

// 修正constructor Child.prototype.constructor = Child

它避免了组合继承中调用两次父类构造函数，初始化两次实例属性的缺点。

所以它拥有了上述所有继承方式的优点：

只调用了一次父类构造函数，只创建了一份父类属性

子类可以用到父类原型链上的属性和方法

能够正常的使用instanceOf和isPrototypeOf方法

function Parent (name) {

this.name = name

}

Parent.prototype.getName = function () {

console.log(this.name)

}

function Child (name) {

this.sex = 'boy'

Parent.call(this, name)

}

// 与组合继承的区别

Child.prototype = Object.create(Parent.prototype)

var child1 = new Child('child1')

console.log(child1)

child1.getName()

console.log(child1.\_\_proto\_\_)

console.log(Object.create(null))

console.log(new Object())

寄生式继承

function createAnother (original) {

var clone = Object.create(original);;// 通过调用 Object.create() 函数创建一个新对象

clone.fn = function () {}; // 以某种方式来增强对象

return clone; // 返回这个对象 }

例如我现在想要继承某个对象上的属性，同时又想在新创建的对象中新增上一些其它的属性。

**实现方式：**

* 在**原型式继承**的基础上再封装一层，来增强对象，之后将这个对象返回。

**优点：**

* 再不用创建构造函数的情况下，实现了原型链继承，代码量减少一部分。

**缺点：**

* 一些引用数据操作的时候会出问题，两个实例会公用继承实例的引用数据类
* 谨慎定义方法，以免定义方法也继承对象原型的方法重名
* 无法直接给父级构造函数使用参数

var cat = {

heart: '❤️',

colors: ['white', 'black']

}

function createAnother (original) {

var clone = Object.create(original);

clone.actingCute = function () {

console.log('我是一只会卖萌的猫咪')

}

return clone;

}

var guaiguai = createAnother(cat)

var huaihuai = Object.create(cat)

guaiguai.actingCute()

console.log(guaiguai.heart)

console.log(huaihuai.colors)

console.log(guaiguai)

console.log(huaihuai)

**混入继承**

function Child () { Parent.call(this) OtherParent.call(this) } Child.prototype = Object.create(Parent.prototype) Object.assign(Child.prototype, OtherParent.prototype) Child.prototype.constructor = Child

function Child () {

Parent.call(this)

OtherParent.call(this)

}

Child.prototype = Object.create(Parent.prototype)

Object.assign(Child.prototype, OtherParent.prototype)

Child.prototype.constructor = Child

child1不仅复制了Parent上的属性和方法，还复制了OtherParent上的。

**通过extends继承**

在子类的普通函数中super对象指向父类的原型对象,在子类的静态方法中super对象指向父类

JS设计模式

**工厂模式**

工厂模式类似于现实生活中的工厂可以产生大量相似的商品，去做同样的事情，实现同样的效果;这时候需要使用工厂模式。

**优点：**能解决多个相似的问题。

**缺点：**不能知道对象识别的问题(对象的类型不知道)。

function createPerson(name, age, job){

var o = new Object();

o.name = name;

o.age = age;

o.job = job;

o.sayName = function(){

alert(this.name);

};

return o;

}

var person1 = createPerson("Nicholas", 29, "Software Engineer");

var person2 = createPerson("Greg", 27, "Doctor");

**适配器模式**

#### 优点 可以让任何两个没有关联的类一起运行。提高了类的复用。适配对象，适配库，适配数据，vue的computed中可以用适配器

#### 缺点 额外对象的创建，非直接调用，存在一定的开销（且不像代理模式在某些功能点上可实现性能优化)，如果没必要使用适配器模式的话，可以考虑重构，

class Plug {

getName() {

return 'iphone充电头';

}

}

class Target {

constructor() {

this.plug = new Plug();

}

getName() {

return this.plug.getName() + ' 适配器Type-c充电头';

}

}

let target = new Target();

target.getName(); // iphone充电头 适配器转Type-c充电头

**装饰者模式**

动态地给某个对象添加一些额外的职责，，是一种实现继承的替代方案

在不改变原对象的基础上，通过对其进行包装扩展，使原有对象可以满足用户的更复杂需求，而不会影响从这个类中派生的其他对象

#### 优点 装饰类和被装饰类都只关心自身的核心业务，实现了解耦。方便动态的扩展功能，且提供了比继承更多的灵活性。

#### 缺点 额外对象的创建，非直接调用，存在一定的开销（且不像代理模式在某些功能点上可实现性能优化)，如果没必要使用适配器模式的话，可以考虑重构，

class Cellphone {

create() {

console.log('生成一个手机')

}

}

class Decorator {

constructor(cellphone) {

this.cellphone = cellphone

}

create() {

this.cellphone.create()

this.createShell(cellphone)

}

createShell() {

console.log('生成手机壳')

}

}

// 测试代码

let cellphone = new Cellphone()

cellphone.create()

console.log('------------')

let dec = new Decorator(cellphone)

dec.create()

**代理模式**

事件发布/订阅模式 (PubSub) 在异步编程中帮助我们完成更松的解耦, 甚至在 MVC、MVVC 的架构中以及设计模式中也少不了发布-订阅模式的参与。

**优点:** 代理模式能将代理对象与被调用对象分离，降低了系统的耦合度。代理模式在客户端和目标对象之间起到一个中介作用，这样可以起到保护目标对象的作用。代理对象可以扩展目标对象的功能；通过修改代理对象就可以了，符合开闭原则；

**缺点:** 处理请求速度可能有差别，非直接访问存在开销

<ul id="ul">

<li>1</li>

<li>2</li>

<li>3</li>

</ul>

<script>

let ul = document.querySelector('#ul');

ul.addEventListener('click', event => {

console.log(event.target);

});

</script>

**外观模式**

为子系统的一组接口提供一个一致的界面，定义了一个高层接口，这个接口使子系统更加容易使用

**优点:** 减少系统相互依赖。提高灵活性。提高了安全性

**缺点:** 不符合开闭原则，如果要改东西很麻烦，继承重写都不合适

**发布订阅模式**

事件发布/订阅模式 (PubSub) 在异步编程中帮助我们完成更松的解耦, 甚至在 MVC、MVVC 的架构中以及设计模式中也少不了发布-订阅模式的参与。

**优点:** 在异步编程中实现更深的解耦

**缺点:** 如果过多的使用发布订阅模式, 会增加维护的难度

var Event = function() {

this.obj = {}

}

Event.prototype.on = function(eventType, fn) {

if (!this.obj[eventType]) {

this.obj[eventType] = []

}

this.obj[eventType].push(fn)

}

Event.prototype.emit = function() {

var eventType = Array.prototype.shift.call(arguments)

var arr = this.obj[eventType]

for (let i = 0; i < arr.length; i++) {

arr[i].apply(arr[i], arguments)

}

}

var ev = new Event()

ev.on('click', function(a) { // 订阅函数

console.log(a) // 1

})

ev.emit('click', 1)

**观察者模式**

当观察的数据对象发生变化时, 自动调用相应函数。比如 vue 的双向绑定; 每当调用对象里的某个方法时, 就会调用相应'访问'逻辑。比如给测试框架赋能的 spy 函数;

**优点:** 在异步编程中实现更深的解耦

**Object.defineProperty**

var obj = {

data: { list: [] },

}

Object.defineProperty(obj, 'list', {

get() {

return this.data['list']

},

set(val) {

console.log('值被更改了')

this.data['list'] = val

}

})

**proxy**

var obj = {

value: 0

}

var proxy = new Proxy(obj, {

set: function(target, key, value, receiver) { // {value: 0} "value" 1 Proxy {value: 0}

console.log('调用相应函数')

Reflect.set(target, key, value, receiver)

}

})

proxy.value = 1 // 调用相应函数