### 项目介绍，使用了哪些技术，收获了什么

### 介绍一下backbone

Backbone.js：前端框架（MVC）

**Model：**

1. 组织数据和业务逻辑；
2. 和数据库交互；
3. 数据变化时触发事件

Model可复用，可以在app中传递。

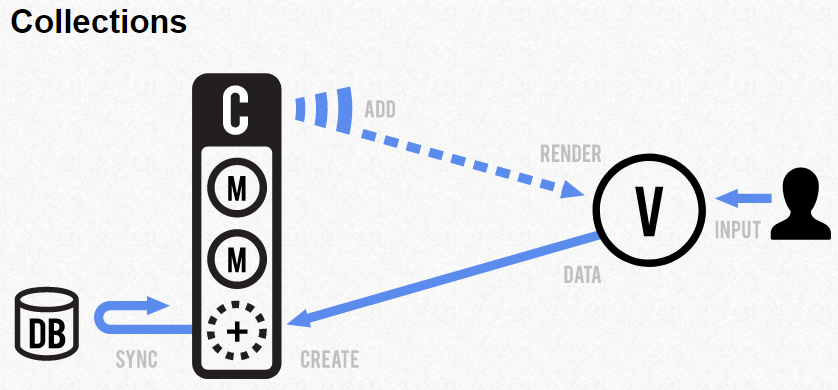
**View：**

1. 监听页面变化和重新渲染页面；
2. 处理用户的输入和交互；
3. 发送捕捉的输入到Model

View是一块独立的UI，从一个或多个models中渲染数据，也可以不需要数据。Models应该不感应views的变化，相反，views监听model的变化，重新渲染页面或者作出适当的反应。

**Collection：**

Collection帮助处理一组相关的models，加载和保存新的models到服务器，提供models集成和计算的方法。除了models自己的事件之外，collections可以集中代理models的所有事件。



### 一道JS编程题



涉及知识点：变量定义提升、函数声明提升、this指针指向、作用域链、运算符优先级、原型与继承、全局变量污染、对象属性和原型属性。

**第二问，考察变量定义提升和函数声明提升**

function Foo() {

    getName = function () { alert (1); };

    return this;

}

var getName;//只提升变量声明

function getName() { alert (5);}//提升函数声明，覆盖var的声明

Foo.getName = function () { alert (2);};

Foo.prototype.getName = function () { alert (3);};

getName = function () { alert (4);};//最终的赋值再次覆盖function getName声明

getName();//最终输出4

**第三问，考察this指向，作用域链，全局变量污染**

Foo()执行之后，覆盖了window的全局变量getName；函数返回的this指向window。

**第五问，考察运算符优先级**



成员访问“.”的优先级高于new（无参数列表）

new Foo.getName(); 等价于new (Foo.getName)();所以实际上将getName函数作为构造函数执行。

**第六问，考察构造函数的返回值、运算符优先级**

由于new（带参数列表）的优先级和成员访问“.”相同，所以从左至右执行

new Foo().getName()；等价于(new Foo()).getName()；

构造函数的返回值问题：

1. 无返回值，返回实例化对象
2. 返回值是非引用类型，也返回实例化对象
3. 返回值是引用类型，返回该值

new Foo()返回this，即当前实例化对象，所以返回的是Foo的实例，然而没有getName属性，所以在原型中寻找getName。

**第七问，考察运算符优先级**

new new Foo().getName();

等价于new (new Foo()).getName();

等价于new ((new Foo()).getName)();

最终其实是将Foo原型的getName函数作为构造函数调用。

### 元素居中

**水平居中：**

inline 元素：text-align: center

block 元素：margin: 0 auto

多个block元素并排：

HTML：

<main class="inline-block-center">  
 <div>  
 I'm an element that is block-like with my siblings and we're centered in a row.  
 </div>  
 <div>  
 I'm an element that is block-like with my siblings and we're centered in a row.  
 </div>  
 <div>  
 I'm an element that is block-like with my siblings and we're centered in a row.  
 </div>  
</main>

CSS：

.inline-block-center {  
 text-align: center;  
}  
.inline-block-center div {  
 display: inline-block;  
 text-align: left;  
}

用flexbox：

.flex-center {  
 display: flex;  
 justify-content: center;  
}

**垂直居中：**

inline元素：

一行：padding-top = padding-bottom或者令line-height = height即可

多行：

HTML：

<div class="center-table">  
 <p>I'm vertically centered multiple lines of text in a CSS-created table layout.</p>  
 </div>

CSS：// 伪装成table的cell，设置vertical-align即可

.center-table {  
 display: table;  
 }  
 .center-table p {  
 display: table-cell;  
  vertical-align: middle;  
 }

用flexbox：

.flex-center-vertically {

display: flex;

justify-content: center;

flex-direction: column;

height: 400px;

}

block元素：

知道需要居中元素的height：

.parent {

position: relative;

}

.child {

position: absolute;

top: 50%;

height: 100px;

margin-top: -50px; /\* account for padding and border if not using box-sizing: border-box; \*/

}

不知道需要居中元素的height：

main {

position: relative;

}

main div {

position: absolute;

top: 50%;

transform: translateY(-50%);

}

用flexbox:

.parent {

display: flex;

flex-direction: column;

justify-content: center;

}

**水平和垂直居中：**

固定宽度和高度的元素：

.parent {

position: relative;

}

.child {

width: 300px;

height: 100px;

padding: 20px;

position: absolute;

top: 50%;

left: 50%;

margin: -70px 0 0 -170px;

}

未知宽度和高度：

.parent {

position: relative;

}

.child {

position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%, -50%);

}

使用flexbox：

.parent {

display: flex;

justify-content: center;

align-items: center;

}

### 函数的两种创建方式的区别以及NFE和IIFE

从NFE(具名函数表达式)引出的一些函数知识点总结：

1. **函数表达式VS函数声明**

ECMA解释说函数表达式和函数声明的区别主要在于函数声明必须有一个Identifier（或者说是函数名字）而函数表示式可以没有。

函数声明 :

function Identifier ( FormalParameterList opt ){ FunctionBody }

函数表达式：

function Identifier opt ( FormalParameterList opt ){ FunctionBody }

<opt：表示option>

没有函数名字的时候，一定是匿名函数表达式；函数如果有名字，就需要通过所处上下文来判断。如果是传参、赋值或者new运算符，就应该是函数表达式；如果是孤零零地在函数体内或者全局域中，就应该是函数声明。比如：

function foo(){} // declaration, since it's part of a Program

var bar = function foo(){}; // 表达式, 因为是赋值

new function bar(){}; // expression, since it's part of a NewExpression

(function(){

function bar(){} // declaration, since it's part of a FunctionBody

})();// 括号包含的是函数表达式

区别：

1. 函数声明最后没有分号，而函数表达式有
2. 函数声明提前
3. 通过条件语句控制函数声明的行为并未标准化，因此不同环境下可能会得到不同的结果，所以永远都不要依赖条件控制来声明函数，而应该使用函数表达式。

// Never do this!

// Some browsers will declare `foo` as the one returning 'first',

// while others — returning 'second'

if (true) {

function foo() {

return 'first';

}

}

else {

function foo() {

return 'second';

}

}

foo();

// Instead, use function expressions:

var foo;

if (true) {

foo = function() {

return 'first';

};

}

else {

foo = function() {

return 'second';

};

}

foo();

1. **NFE**

具有Identifier（或者说函数名字）的函数表达式被称作具名函数表达式（named function expression）：

var bar = function foo(){};

只能在函数内部访问具名函数表达式的名字foo。

具名函数表达式的作用主要是追踪栈中有函数名，利于调试，其次可以在递归时使用（代替arguments.callee）。

1. **IIFE**

立即执行函数表达式（Immediately Invoked Function Expression）的形式如下：

(function(){

//...

})();

立即执行函数表达式是指程序运行到此时函数立即执行。用法：

1. 传递参数为window，可以更快地访问全局作用域里面的变量，不需要沿着作用域链进行查找
2. 传递参数为函数
3. 利用函数作用域创建块级作用域，防止全局变量污染
4. 模块化编程、测试和部署
5. **函数的特性、模式以及高级用法**

**函数特性：**

1. 函数是一等对象（first-class）

* 可以在程序执行时动态创建函数
* 可以将函数赋值给变量，可以将函数的引用拷贝给另一个变量，可以扩充和删除
* 可以将函数作为参数传递，可以作为返回值返回
* 可以添加属性和方法

“我们首先当它是一个对象，具有可执行的特性”

1. 函数提供作用域支持

在JavaScript中没有块级作用域，只有函数作用域。

**回调模式：**

1. 最简单的回调：函数接受一个函数作为参数并在函数中调用传入的函数。此时，传入的函数就叫做回调函数。
2. 回调和作用域：如果回调函数是匿名函数或者全局函数，在函数中就可以直接调用。如果回调函数是对象的方法并使用了对象的属性，此时就需要注意作用域的问题。除了传入回调函数，还需要传入回调函数所属的对象，并在函数中利用call()和apply()指定回调函数的作用域。
3. 异步事件监听和延时：JavaScript中的事件监听和延时函数都用到了回调函数。
4. 类库中的回调：在类库的设计时经常使用回调模式。设计时着重核心功能的实现，尽可能保持可复用和通用，但同时提供回调的入口作为“钩子（hook）”，定制需要的特性使类库变得可扩展和可定制。

**重定义函数：**

函数可以在运行中动态定义，用新函数覆盖掉旧函数。

当函数中包含一些初始化操作并只需要执行一次时，或者函数里面的控制流每次都是一样时，这种模式非常合适，可以避免执行重复的代码，提高应用的执行效率。这种模式也被称为函数的懒惰定义。

缺陷：原函数的功能丢失；如果这个函数被重定义为不同的名字，被赋值给不同的变量，或者是作为对象的方法使用，那么重定义的部分并不会生效，原来的函数依然会被执行。

**函数属性——记忆模式**

将函数执行结果保存为函数的自定义属性，避免函数下次调用时重复复杂的计算。

var myFunc = function foo () {

var cachekey = JSON.stringify(Array.prototype.slice.call(arguments)),

result;

if (!foo.cache[cachekey]) {

result = {};

// ……复杂的计算……

foo.cache[cachekey] = result;

}

return foo.cache[cachekey];

};

// 缓存

myFunc.cache = {};

**函数柯里化(Currying）:**

解释：让函数理解并处理部分应用的过程叫做柯里化。柯里化是一个变换函数的过程。

通用的柯里化函数：

function schonfinkelize(fn) {

var slice = Array.prototype.slice,

stored\_args = slice.call(arguments, 1);

return function () {

var new\_args = slice.call(arguments),

args = stored\_args.concat(new\_args);

return fn.apply(null, args);

};

}

借用Array.prototype中slice()方法将arguments转换成数组，通过闭包访问存储的旧参数。

用法：

// 普通函数

function add(a, b, c, d, e) {

return a + b + c + d + e;

}

// 参数个数可以随意分割

schonfinkelize(add, 1, 2, 3)(5, 5); // 16

// 两步柯里化

var addOne = schonfinkelize(add, 1);

addOne(10, 10, 10, 10); // 41

var addSix = schonfinkelize(addOne, 2, 3);

addSix(5, 5); // 16

使用场景：

当你发现自己在调用同样的函数并且传入的参数大部分都相同的时候，就是考虑柯里化的理想场景了。你可以通过传入一部分的参数动态地创建一个新的函数。这个新函数会存储那些重复的参数（所以你不需要再每次都传入），然后再在调用原始函数的时候将整个参数列表补全。

### 函数的闭包的解释和用法

闭包（closure）是Javascript语言的一个难点，也是它的特色。

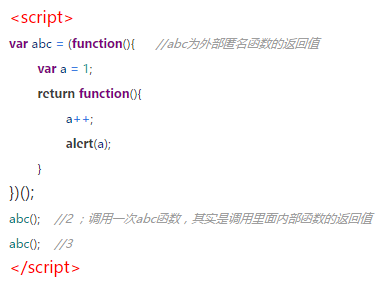
1. 闭包的特性
2. 函数嵌套
3. 函数内部可以引用外部的参数和变量
4. 参数和变量不会被垃圾回收机制回收
5. 闭包的定义和优缺点

闭包是指有权访问另一个函数作用域中的变量的函数。创建闭包的最常见的方式就是在函数内创建函数。

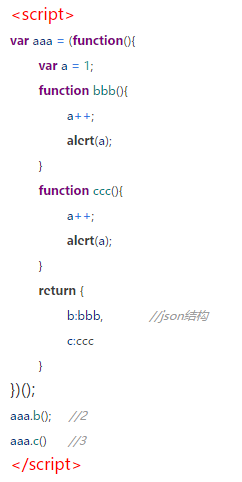
缺点：内存常驻，增加内存使用量，使用不当会造成内存泄漏。

优点：希望某变量长期驻扎在内存中；避免全局变量的污染；创造私有成员

1. 闭包的使用
2. 模块化代码，避免全局变量的污染



1. 创建私有成员



1. 在循环中为DOM元素绑定事件函数

