localStorage、sessionStorage与设计模式

1.localStorage和sessionStorage功能

早期的web中使用cookies在客户端保存诸如用户名等简单的信息，但是，在使用cookies存储永久数据存在以下问题。

1.cookies的大小限制在4kB，不适合大量的数据存储。

2.浏览器还限制站点可以在用户计算机上存储的cookies的数量。

3 cookies是随HTTP事务一起被发送的，因此会浪费一部分带宽。

HTML5很好的提供了本地存储的功能，以键值对存储的解决方案，支持容量至少为4M，HTML5的web提供了两种客户端存储方式。

localStorage：是一种没有时间限制的数据存储方式，可以将数据永久保存在客户端。

sessionStorage：指的是针对一个session的数据存储，即将数据保存在session对象中，当关闭浏览器后，这些数据就被删除。因此sessionStorage不是一种持久化的本地存储，仅仅是会话级别的存储。



web Storage支持的属性与方法

localStorage和sessionStorage都具有相同的操作方法，例如setItem、getItem和removeItem等

getItem(key):获取指定key所存储的value值

key(index)方法：返回列表中对应索引的key值

length属性：返回key/value队列的长度

removeItem(key)方法：从Storage中删除一个对应的键值对。

setItem(key,value)方法：将value存储到key指定的字段。

clear()方法：移除所有的内容

存：三种方法

    let ls = localStorage;

    ls.setItem("id",666);

    ls.name = "老王";

    ls["pwd"] = 123456;

    let ss = sessionStorage;

    ss.setItem("id",666);

    ss.name = "老王";

    ss["pwd"] = 123456;

    存取json数据:

    let ls = localStorage;

    ls.setItem("data",'{"name":"水煮鱼","price":18}');

    let myJson = JSON.parse(ls.data);

    for(let index in myJson){

        console.log(myJson[index]);

    }

取：

    console.log(ls.getItem("id"));

    console.log(ls.name);

    console.log(ls["pwd"]);

改:与存的方式相同，key不在则创建，key在则修改value。

遍历:

    let ls = localStorage;

    for(let i=0;i<ls.length; i++){

        console.log(ls.key(i));

    }

删:

     let ls = localStorage;

     ls.clear();//全部删除

     ls.removeItem("name");//根据key值删除

2.设计模式是什么 (类的设计方案)

背景：

\* 目前企业级分布式软件开发普遍采用面向对象的方法，OOD（面向对象设计）直接导致了设计模式的发展。

\* 开发面向对象的软件是困难的，而开发可复用的面向对象的软件更难。

\* 有经验的设计者重用过去的方案。

\* 采用设计模式使设计和代码具有良好的可维护性、可复用性和可升级性。

\* “Design patterns help you learn from others‘ successes instead of your own failures.”

\* 1970s ，Christopher Alexander 的建筑师 提出设计模式概念。

\* 直到 1987，一些设计模式的论文和文章出现了 。

\* 1995年 ，GOF 发表了书： 《设计模式－可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software ）

\* 定义：模式是在一个上下文中，对一个问题的解决方案。即模式的四要素：名字、上下文、问题和解决方案。

\* 分类：23种设计模式：

  创建型：5种

  结构型：7种

  行为型：11种

为什么使用设计模式：

\* 尽量用一种标准的方式描述设计经验

\* 为设计者提供一种通用的语言

\* 增加复用性，减少设计的多样性

\* 增强设计变更的灵活性

\* 提高设计文档的质量

\* 增强设计的可理解性

3.单例模式:

\* 单例模式指的是一个类只能有一个实例，这样的类被称为单例类，或者单态类，即Singletion Class

\* 单例类的特点

    \* 单例类只可有一个实例

    \* 它必须自己创立这唯一的一个实例

    \* 它必须给所有其它的类提供自己这一实例

单例模式的思路是：一个类能返回一个对象的引用（并且永远是同一个）和一个获得该实例的方法（静态方法，通常使用 getInstance 名称）。

                那么当我们调用这个方法时，如果类持有的引用不为空就返回该引用，否者就创建该类的实例，并且将实例引用赋值给该类保持的那个引用再返回。

                同时将该类的构造函数定义为私有方法，避免其他函数使用该构造函数来实例化对象，只通过该类的静态方法来得到该类的唯一实例。

对于 JS 来说，巨大的灵活性使得其可以有多种方式实现单例模式，使用闭包方式来模拟私有数据，按照其思路可得：

    var single = (function() {

        var unique;

        function getInstance() {

            if (unique === undefined) {

                unique = new Construct();

            }

            return unique;

        }

        function Construct() {

            // ... 生成单例的构造函数的代码

        }

        return {

            getInstance: getInstance

        }

    })();

      以上，unique便是返回对象的引用，而 getInstance便是静态方法获得实例。Construct 便是创建实例的构造函数。

      可以通过 single.getInstance() 来获取到单例，并且每次调用均获取到同一个单例。这就是单例模式所实现的效果。

1.字面量方式:

    let singletion= {

        id:18,

        fun:()=>{

            console.log("heihei");

        }

    }

    let t1 = singleton;

    let t2 = singleton;

    console.log(t1===t2);

    十分简单，并且非常实用，不足之处在于没有什么封装性，所有的属性方法都是暴露的。

2.添加构造方法内部属性

    function Student(newId,newName){

        if(Student.unique == undefined){

            this.id = newId;

            this.name = newName;

            Student.unique = this;

        }else{

            return Student.unique;

        }

    }

    let s1 = new Student(111,"小明");

    let s2 = new Student(222,"老王");

    console.log(s1.id,s1.name);

    console.log(s2.id,s2.name);

    s2.id = 666;

    s2.name = "heihei";

    console.log(s1.id,s1.name);

    console.log(s2.id,s2.name);

    console.log(s1 === s2);

    也是非常简单，无非就是提出一个属性来做判断，但是该方式也没有安全性，一旦我在外部修改了Student的unique属性，那么单例模式也就被破坏了。

3.闭包

     对于灵活的JS来说，任何问题都能找到 n 种答案，只不过让我自己去掂量孰优孰劣而已，下面就列举闭包实现单例模式的方法，无非也就是将创建了的单例缓存而已。

        let singletion = (function (newId,newName){

        var unique;

        function Student(newId,newName){

            this.id = newId;

            this.name = newName;

        }

        if(unique==undefined){

            unique = new Student(newId,newName);

        }

        return function(){

            return unique;

        };

    })(1,"小明");//切记必须自运行 为什么呢？

    let s1 = singletion();

    let s2 = singletion();

    s1.id = 666;

    console.log(s1.id,s2.id);

    console.log(s1 === s2);

4.代理模式

代理模式的定义是把对一个对象的访问, 交给另一个代理对象来操作。

      如：

           一般人需要打官司，需要找代理律师（术业有专攻）；

           你需要打扫房子，可以找保洁公司（术业有专攻）；

           我们在租房子的时候去找中介（因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做）

\*  应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

class Nurse{

        constructor() {

        }

        takeBabyEat()

            //假设要扩展功能，可以在这里添加代码

            console.log("Nurse:takeBabyEat");

        }

    }

    class Mother{

        constructor(newNurse) {

            this.nurse = newNurse;

        }

        takeBabyEat(){

            this.nurse.takeBabyEat();

        }

    }

    let n = new Nurse();

    let m = new Mother(n);

    m.takeBabyEat();

4.工厂模式

\* 工厂模式：就是用工厂的思路，创建对象。工厂是造产品的。现在用工厂来造对象。即一个工厂可以制造很多种类型的对象，这些对象一般具有共同的父类，即相似的类。

           为什么使用工厂模式?

           使用一个类(通常为单体)来批量生成实例.

           工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。

          以下几种情景下工厂模式特别有用：

          对象的构建十分复杂

          需要依赖具体环境创建不同实例

          处理大量具有相同属性的小对象

         我们所熟悉的工厂是不是重复生产，编程中称为迭代！

         比如生产一颗螺丝钉，一个流水线就是不停的生产螺丝钉。

         再如联想电脑，联想公司的笔记本生产线只负责生产笔记本，不能生产出TV吧。

         但是如果有TV生产线，那么TV生产生只能生产TV，而且每天都是重复的工作。

         当然笔记本生产想也是重复的。

         编程中我们也想让编程跟流水线一样，想用的时候开启一下工厂就可以得到一个想要的程序！

         注意，每次得到的程序和已经得到的程序不会是同一个程序。

         就好像生产出两台笔记本，肯定不是同一个笔记本！！

Demo1:

        function Factory(color){//生产笔记本的流水线

            return {

                "screen":"this is screen",

                "keybord":"this is keybord",

                "mouse":"this is mouse",

                "USB":"this is API of USB",

                "storage":"this is storage",

                "color":color

            }

        }

        //来一个黄色的笔记本

        var \_yellow=Factory("yellow");

        //来一个黑色的笔记本

        var \_black=Factory("black");

        //来一个紫色的笔记本

        var \_purple=Factory("purple");

        //console.log(\_yellow,\_black,\_purple);

        //是不是很完美

        在实际的项目中，我们常常需要根据用户的权限来创建不同的页面，高级权限的用户所拥有的页面有些是无法被低级权限的用户所查看。

        所以我们可以在不同权限等级用户的构造函数中，保存该用户能够看到的页面。在根据权限实例化用户。

Demo2:

     function UserFactory(role) {

        function SuperAdmin() {

            this.name = "超级管理员";

            this.viewPage = ['首页', '通讯录', '发现页', '应用数据', '权限管理'];

        }

        function Admin() {

            this.name = "管理员";

            this.viewPage = ['首页', '通讯录', '发现页', '应用数据'];

        }

        function NormalUser() {

            this.name = '普通用户';

            this.viewPage = ['首页', '通讯录', '发现页'];

        }

        switch (role) {

            case 'superAdmin':

                return new SuperAdmin();

                break;

            case 'admin':

                return new Admin();

                break;

            case 'user':

                return new NormalUser();

                break;

            default:

        }

    }

    //调用

    let superAdmin = UserFactory('superAdmin');

    let admin = UserFactory('admin');

    let normalUser = UserFactory('user');

    console.log(superAdmin,admin,normalUser);

5.抽象工厂模式

什么是抽象类？ abstruct  无法创建实例对象的类。

\* 抽象工厂模式：实现“接口”的概念。

抽象工厂

抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。

抽象工厂模式是指当有多个抽象角色时，使用的一种工厂模式。

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。

        function AbstractFactory(){//生产5.8inch两毫米厚度钻石手机屏幕

            return {

                "thickness":"2mm",

                "size":"5.8inch",

                "material":"Diamonds"

            }

        }

        function Iphone(){

            this.shell="铝合金";

            this.model="7Plus";

            this.screen=AbstractFactory();

        }

        function Mate8(){

            this.shell="铝合金";

            this.model="Mate8";

            this.screen=AbstractFactory();

        }

        function P9(){

            this.shell="铝合金";

            this.model="P9";

            this.screen=AbstractFactory();

        }

        //那货是个热门货，什么手机都能用，要发财的节奏

6.观察者模式

定义:

观察者模式是软件设计模式的一种常见的设计模式，又称发布订阅模式。在这种模式中，并不是单纯的一个对象调用另一个对象的方法，而是一个对象订阅另一个对象的特定活动并在其状态发生改变后获得通知，然后执行相应的操作。订阅者也称为观察者，而被观察的对象则被称为发布者或主题。当一个事件发生，发布者将会向所有订阅此事件的对象以事件对象的形式传递消息。

比如说：前段时间世界杯，很多人都看球赛，但人精力还是有限的，经常可能会因为这样或哪样的原因错过看球赛的时间，在这种情况下很多直播网站提供了一个很贴心的服务，那就是开播提醒，当用户订阅了这个提醒，那么在某个时刻比如球赛开播前5分钟系统会给所有订阅开播提醒的用户以短信或者其他有效形式发送通知，提醒用户球赛马上开播了，这个就是观察者模式的典型应用。

要素:

观察者模式最重要的两个要素是：发布者、订阅者（观察者），发布者发送消息，订阅者订阅特定消息并接受发布者发送的消息。

    //观察者（订阅者） 订阅update消息

    var sub1 = {

        update: function(what) {//what是发布者传的消息

            console.log(what+"sub1根据消息做什么");

        }

    }

    var sub2 = {

        update: function(what) {

            console.log(what+"sub2根据消息做什么");

        }

    }

    var sub3 = {

        update: function(what) {

            console.log(what+"sub3根据消息做什么");

        }

    }

    //发布者

    class Dep{

        constructor(newSubs) {

            this.subs = newSubs;

        }

        //消息:核心

        information(){

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

                this.subs[i].update("发布消息");

            }

        }

    }

    //订阅

    let dep = new Dep([sub1,sub2,sub3]);

    //发送消息

    dep.information();

7.适配器模式

\* 结构模式\_适配器模式

\* 应用场景：

       适配器模式，一般是为要使用的接口，不能完全符合本应用或本系统使用，而需引入的中间适配层类或对象的情况；

       如：手机充电器，Android手机的充电线没法给iPhone手机充电，因为，口不对，插不进去，如果有一个能够Android手机的充电线的头转换成能插在iPhone手机上的设备，这个就是适配器（转接口，转接头），恰好已经有了。

      再如：你需要6v的电压，但是，目前墙上有一个提供220v电压的插孔，怎么办，有个设备能把220v转成6v，交流转成直流，这个设备就是适配器。

         由于旧的系统，或第三方应用提供的接口（更多特指函数的参数），与我们定义的接口不匹配，就无法使用这样旧的，或第三方的接口，这时我们就使用适配类继承待适匹配的类，并让适配类实现接口的方式来引入旧的系统或第三方应用的接口；

    class OldClass{

        constructor() {

        }

        oldFun(param1){//假设在使用的地方突然有两个参数

                       //此时就需要一个转换函数来实现该功能

        }

    }

    class Adapter{

        constructor(newOldClass) {

            this.OldClass = newOldClass;

        }

        //newFun就是转换函数

        newFun(param1,param2){

            this.OldClass.oldFun(param1);

            param2 = "heihei";//模拟对参数2的操作

        }

    }

    //定义一个适配器，由适配器把旧类的对象包装起来。

    let a = new Adapter(new OldClass());

    //调用适配器的newFun函数，最终还是得调用旧类的oldFun函数,但满足了操作两个函数的功能

    a.newFun(1,2);

作业:

1.课堂案例，整理笔记

2.封装放大镜

3.封装选择器，类似实现document.querySelector功能

4.预习jQuery框架

5.localStorage模拟购物车