JS设计模式23道面试题（https://github.com/minsion）

面试题 1. 简述对网站重构的理解?

推荐指数： ★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

网站重构：在不改变外部行为的前提下，简化结构、添加可读性，而在网站前端保持一致的行为。也就是说是在不改变UI的情况下，对网站进行优化，
在扩展的同时保持一致的UI。
对于传统的网站来说重构通常是：

@ 表格(table)布局改为DIV+CSS
@ 使网站前端兼容于现代浏览器(针对于不合规范的CSS、如对IE6有效的)
@ 对于移动平台的优化
@ 针对于SEO进行优化
@ 深层次的网站重构应该考虑的方面
@ 减少代码间的耦合
@ 让代码保持弹性
@ 严格按规范编写代码
@ 设计可扩展的API
@ 代替旧有的框架、语言(如VB)
@ 增强用户体验
@ 通常来说对于速度的优化也包含在重构中
@ JS、CSS、image等前端资源(通常是由服务器来解决)
@ 程序的性能优化(如数据读写)
@ 采用CDN来加速资源加载
@ 对于JS DOM的优化
@ HTTP服务器的文件缓存

面试题 2. 简述什么是设计模式?

推荐指数： ★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

设计模式(英语 design pattern)是对面向对象设计中反复出现的问题的解决方案。这个术语是在1990年代由Erich Gamma等人从建筑设计领域引入到计算机科学中来的。这个术语的含义还存有争议。算法不是设计模式，因为算法致力于解决问题而非设计问题。设计模式通常描述了一组相互紧密作用的类与对象。设计模式提供一种讨论软件设计的公共语言，使得熟练设计者的设计经验可以被初学者和其他设计者掌握。设计模式还为软件重构提供了目标。
随着软件开发社群对设计模式的兴趣日益增长，已经出版了一些相关的专著，定期召开相应的研讨会，而且Ward Cunningham为此发明了WikiWiki用来交流设计模式的经验。

总之，设计模式就是为了解决某类重复出现的问题而出现的一套成功或有效的解决方案

面试题 3. 常见的设计模式有哪些?

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

GOF提出的23种设计模式，分为三大类。
（1）创建型模式，共5种，分别是工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
（2）结构型模式，共7种，分别是适配器模式、装饰器模式、代理模式、外观模式桥接模式、组合模式、享元模式。
（3）行为型模式，共11种，分别是策略模式、模板方法模式、观察者模式、选代子模式、责任链模式、命令模式、备忘录模弋、状态模式、访问者模式、中介者模式、解释器模式。
在前端开发中，有些特定的模式不太适用。当然，有些适用于前端的模式并未包含在这23种设计模式中，如委托模式、节流模式等

面试题 4. 简述工厂模式的概念？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

其概念如下：

工厂模式需要3个基本步骤，原料投入、加工过程以及成品出厂，例如以下代码。

```
function playerFactory (username){
var user= new object ();
user .username = username;
return user ;
}
var xm = playerFactory( 'xiao ming ')
```

player Factory函数中传递的参数就是“基本原料的投入”。从 var user= new Object()直到return之前，都属于“加工过程”。最后的 return就如同“成品出厂”

面试题 5. 简述工厂模式的缺陷？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

缺陷如下

（1）没有使用new关键字，在创建对象的过程中，看不到构造函数实例化的过程。

（2）每个实例化的对象都创建相应的变量和函数，因此需要更多的空间进行属性和方法的存储，从而降低了性能，造成资源的浪费。

面试题 6. 简述JavaScript MC架构和MVVM架构的理解?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

在经典的MVC架构中，包含3个部分，即模型（Model）、视图(view)和控制器(Controller)。控制器可以访问视图，让其更新。控制器可以访问模型，更新数据。视图可以访问模型，获取数据渲染页面。

在MVVM架构中，包含3个部分，即模型（Model)、视图(View)和视图模型(View Model)。视图模型负责视图与模型之间的信息转换，通过欸据双向绑定使视图与模型之间的数据得以传递。

例如代表性的框架 Angular，它通过数据绑定，将模型中的数据映射到视图中，通过事件监听器（event listener），将视图改变的数据存储在模型内

面试题 7. 简述JavaScript单例模式？

推荐指数：★★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

JavaScript实现单例模式
单例模式是一种创建型设计模式，它保证一个类只有一个实例，并提供一个全局访问点来访问该实例。在 JavaScript 中，实现单例模式有多种方式。本文将介绍几种常见的实现方式，并解释每种实现方式的思路和示例代码。
构造函数方式
在 JavaScript 中，使用构造函数方式实现单例模式最为常见。我们可以在构造函数中创建一个静态属性，用于存储实例对象，然后在构造函数中判断该静态属性是否存在，如果不存在则创建一个新的实例对象并存储在该静态属性中，否则直接返回该静态属性中存储的实例对象。这样就能保证每次创建实例时都返回同一个实例。
以下是使用构造函数方式实现单例模式的示例代码：

```
class Singleton {
constructor() {
if (!Singleton.instance) {
Singleton.instance = this;
}
return Singleton.instance;
}
}
const instance1 = new Singleton();
const instance2 = new Singleton();
console.log(instance1 === instance2);
```

```
// true
```

在这个示例中，我们创建了一个 Singleton 类，使用构造函数方式实现单例模式。在构造函数中，我们首先检查是否已经存在实例。如果不存在实例，则创建一个新的实例并将其存储在静态属性 Singleton.instance 中。如果实例已经存在，则构造函数返回现有实例。这样，每次创建实例时，都会返回同一个实例。

字面量方式

在 JavaScript 中，我们还可以使用字面量方式实现单例模式。在字面量方式中，我们直接创建一个单例对象，然后将其存储在一个变量中，每次需要使用该单例对象时直接使用该变量即可。

以下是使用字面量方式实现单例模式的示例代码：

```
const singleton = {
name: 'Singleton Object',
sayHello() {
console.log('Hello, World!');
}
}
;
singleton.sayHello();
```

在这个示例中，我们创建了一个单例对象 singleton，包含一个名为 name 的属性和一个名为 sayHello 的方法。每次需要使用该单例对象时，只需要直接使用 singleton 对象即可

面试题 8. 简述单例模式的优缺点？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

优点如下。

（1）提供了对唯一实例的受控访问。

（2）由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象，单例模式无疑能够提高系统的性能。

（3）可以根据实际情况的需要，在单例模式的基础上扩展为双例模式和多例模式。

缺点如下。

（1）单例类的职责过重，里面的代码可能会过于复杂，在一定程度上违背了“单职责原则”。

（2）如果实例化的对象长时间不利用，系统会认为它是垃圾而进行回收，这将导致对象状态的丢失。

面试题 9. 简述使用工厂模式最主要的好处？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

好处如下：

（1）把对象的创建集中在一个地方（工厂），在增加新的对象类型的时候，只需要改变工

厂方法。当不使用工厂模式的时候，改变创建方式则需要四处修改，增加维护成本。
（2）新的对象类型可以很容易地添加进来。
（3）只需要关心工厂方法返回的对象，不必关心具体创建的细节

面试题 10. 简述什么是代理模式?

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

代理模式的定义：由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

代理模式的主要优点有：
代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；
代理对象可以扩展目标对象的功能；
代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度，增加了程序的可扩展性

其主要缺点是：
代理模式会造成系统设计中类的数量增加
在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢；
增加了系统的复杂度；
那么如何解决以上提到的缺点呢？答案是可以使用动态代理方式
根据代理的创建时期，代理模式分为静态代理和动态代理。
静态：由程序员创建代理类或特定工具自动生成源代码再对其编译，在程序运行前代理类的.class 文件就已经存在了。
动态：在程序运行时，运用反射机制动态创建而成。

面试题 11. 简述原型模式和单例模式的区别？

推荐指数： ★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

单例模式就是保证一个类只存在一个实例，只初始化一次，第一次完成初始化以后，在重复使用的时候，返回的都是这个实例，而不是新建一个实例。如果实例化的对象里面的属性值已经改变，就不能用单例了，只能通过原型模式重新实例化，原型模式允许多次创建实例对象

面试题 12. 简述组合模式的适用性指的是什么?

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

组合模式是表示对象的“部分-整体”层次结构的一种设计模式；组合模式将对象组合成树状结构以表示“部分-整体”的层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性

面试题 13. 解释什么时候要使用组合模式?

推荐指数： ★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

在以下情况下使用组合模式
（1）当想表示对象的“部分-整体”层次结构（树状结构）时可以使用组合模式
（2）在希望用户忽略组合对象与单个对象的不同并且统一地使用组合结构中的所有对象时使用组合模式

面试题 14. 简述Sass和Less有什么区别?

推荐指数： ★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

区别如下。

（1）编译环境不一样。Sass的安装需要Ruby环境，是在服务器端处理的。而Less需要引入 less.js来处理，然后Less代码输出CSS到浏览器中；也可以在开发环境中使用Less，然后编译成CSS文件，直接放到项目中运行。
（2）变量名不一样。Less中使用@，而Sass中使用$。
（3）插值语法不同，Less中使用@{key}，Sass中使用#{$key}。
（4）Sass的混合相当于Less的方法，Sass的继承相当于Less的混合。
（5）输出设置不同。Less没有输出设置。Sass提供4种输出选项：nested、compact、compressed和 expanded。nested选项用于嵌套缩进的CSS代码（默认）， expanded选项用于展开多行CSS代码， compact选项显示简洁格式的CSS代码， compressed选项显示压缩后的CSS代码。
（6）Sass支持条件语句，如if...else、for循环等，而Less不支持。
（7）引用外部CSS文件的方式不同。Sass引用外部文件时必须以“ _”开头，文件名如果以下划线“_”命名，Sass会认为该文件是一个引用文件，不会将其编译为CSS文件。Less引用外部文件和CSS中的@ import没什么差异。
（8）Sass和Less的工具库不同。Sass有工具库 Compass。简单说，Sass和 Compass的关系有点像 JavaScript和 jQuery的关系， Compass是Sass的工具库。在它的基础上，封装了一系列有用的模块和模板，补充和强化了Sass的功能。Less有UI组件库Bootstrap，Bootstrap是Web前端开发中一个比较有名的前端UI组件库， Bootstrap中样式文件的部分源码就是采用Less语法编写的。

总之，不管是Sass，还是Less，都可以将它们视为一种基于CSS之上的高级语言，其目的是使得CSS开发更灵活和更强大。Sass的功能比Less强大，可以认为Sass是种真正的编程语言；Less则相对清晰明了，易于上手，对编译环境的要求比较宽松

面试题 15. 简述Javascript 抽象工厂模式？

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

为创建一组相关或相互依赖的对象提供一个接口，而且无需指定它们的具体类。
使用场景
一个对象族（或是一组没有任何关系的对象）都有相同的约束。
涉及不同操作系统的时候，都可以考虑使用抽象工厂模式。
代码示例

```
public abstract class AbstractCreator {
//创建 A 产品家族
public abstract AbstractProductA createProductA();
//创建 B 产品家族
public abstract AbstractProductB createProductB();
```

面试题 16. 简述Javascript 模板方法模式？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。
使用场景
多个子类有公有的方法，并且逻辑基本相同时。
重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。
重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数约束其行为。
结构
抽象模板：AbstractClass 为抽象模板，它的方法分为两类：
1、基本方法：也叫做基本操作，是由子类实现的方法，并且在模板方法中被调用。
2、模板方法：可以有一个或几个，一般是一个具体方法，也就是一个框架，实现对基本方法的调度，完成固定的逻辑。
注意：为了防止恶意的操作，一般模板方法都加上 final 关键字，不允许被覆写。
具体模板：实现父类所定义的一个或多个抽象方法，也就是父类定义的基本方法在子类中得以实现。
代码示例

```
package templateMethod;
public class TemplateMethodPattern {
public static void main(String[] args) {
AbstractClass tm=new ConcreteClass();
tm.TemplateMethod();
}
}
//抽象类
```

```
abstract class AbstractClass {
public void TemplateMethod() //模板方法 {
SpecificMethod();
abstractMethod1();
abstractMethod2();
}
public void SpecificMethod() //具体方法 {
System.out.println("抽象类中的具体方法被调用...");
}
public abstract void abstractMethod1();
//抽象方法1
public abstract void abstractMethod2();
//抽象方法2
}
//具体子类
class ConcreteClass extends AbstractClass {
public void abstractMethod1() {
System.out.println("抽象方法1的实现被调用...");
}
public void abstractMethod2() {
System.out.println("抽象方法2的实现被调用...");
}
}
```

面试题 17. 简述Javascript 原型模式 ?

推荐指数: ★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

使用场景

资源优化场景：类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。

性能和安全要求的场景：通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。

一个对象多个修改者的场景：一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。

优点

原型模式实际上就是实现 Cloneable 接口，重写 clone（）方法。

性能优良：原型模式是在内存中二进制流的拷贝，要比直接 new 一个对象性能好很多，特

别是要在一个循环体内产生大量的对象时，原型模式可以更好地体现其优点。

逃避构造函数的约束：这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的。

代码示例

```
public class PrototypeClass implements Cloneable{
//覆写父类 Object 方法
@Override
public PrototypeClass clone(){
PrototypeClass prototypeClass = null;
try {
prototypeClass = (PrototypeClass)super.clone();
} catch (CloneNotSupportedException e) {
//异常处理
}
return prototypeClass;
}
```

面试题 18. 简述Javascript 中介者模式？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

使用场景

中介者模式适用于多个对象之间紧密耦合的情况，紧密耦合的标准是：在类图中出现了蜘蛛网状结构，即每个类都与其他的类有直接的联系。

结构

Mediator 抽象中介者角色：抽象中介者角色定义统一的接口，用于各同事角色之间的通信。

Concrete Mediator 具体中介者角色：具体中介者角色通过协调各同事角色实现协作行为，因此它必须依赖于各个同事角色。

Colleague 同事角色：每一个同事角色都知道中介者角色，而且与其他的同事角色通信的时候，一定要通过中介者角色协作。每个同事类的行为分为两种：一种是同事本身的行为，比如改变对象本身的状态，处理自己的行为等，这种行为叫做自发行为（SelfMethod），与其他的同事类或中介者没有任何的依赖；第二种是必须依赖中介者才能完成的行为，叫做依赖方法（Dep-Method）。

示例代码

```
public abstract class Mediator {
//定义同事类
protected ConcreteColleague1 c1;
protected ConcreteColleague2 c2;
//通过 getter/setter 方法把同事类注入进来
public ConcreteColleague1 getC1() {
return c1;
}
public void setC1(ConcreteColleague1 c1) {
this.c1 = c1;
}
public ConcreteColleague2 getC2() {
return c2;
}
public void setC2(ConcreteColleague2 c2) {
this.c2 = c2;
}
//中介者模式的业务逻辑
public abstract void doSomething1();
public abstract void doSomething2();
}
```

面试题 19. 简述Javascript 装饰模式？

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。

使用场景

需要扩展一个类的功能，或给一个类增加附加功能。

需要动态地给一个对象增加功能，这些功能可以再动态地撤销。

需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式。

结构

Component 抽象构件：Component 是一个接口或者是抽象类，就是定义我们最核心的对象，也就是最原始的对象。在装饰模式中，必然有一个最基本、最核心、最原始的接口或抽象类充当 Component 抽象构件。

ConcreteComponent 具体构件：ConcreteComponent 是最核心、最原始、最基本的接口或抽象类的实现，你要装饰的就是它。

Decorator 装饰角色：一般是一个抽象类，做什么用呢？实现接口或者抽象方法，它里面可不一定有抽象的方法呀，在它的属性里必然有一个 private 变量指向 Component 抽象构件。

具体装饰角色：两个具体的装饰类，你要把你最核心的、最原始的、最基本的东西装饰成其他东西。

代码示例

```
/**
* 装饰角色
*/
@Data
@AllArgsConstructor
@NoArgsConstructor
@Log
class BufferedReader implements Reader{

private Reader reader;
@Override
public void read() {
reader.read();
}

public void readLine(){
read();
log.info("并且仅仅读取一行");
}
}
```

面试题 20. 简述Javascript 策略模式 ？

推荐指数： ★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。

使用场景

多个类只有在算法或行为上稍有不同的场景。

算法需要自由切换的场景。

需要屏蔽算法规则的场景。

具体策略数量超过 4 个，则需要考虑使用混合模式。

结构

Context 封装角色：它也叫做上下文角色，起承上启下封装作用，屏蔽高层模块对策略、算法的直接访问，封装可能存在的变化。

Strategy 抽象策略角色：策略、算法家族的抽象，通常为接口，定义每个策略或算法必须具有的方法和属性。

ConcreteStrategy 具体策略角色：实现抽象策略中的操作，该类含有具体的算法。

代码示例

```
public enum Calculator {
//加法运算
ADD("+"){
public int exec(int a,int b){
return a+b;
}
},
//减法运算
SUB("-"){
public int exec(int a,int b){
return a - b;
}
};
String value = "";
//定义成员值类型
private Calculator(String _value){
this.value = _value;
}
//获得枚举成员的值
public String getValue(){
return this.value;
}
//声明一个抽象函数
public abstract int exec(int a,int b);
}
```

面试题 21. 简述Javascript 适配器模式？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

使用场景

你有动机修改一个已经投产中的接口时，适配器模式可能是最适合你的模式。比如系统扩展了，需要使用一个已有或新建立的类，但这个类又不符合系统的接口，怎么办？详细设计阶段不要考虑使用适配器模式，使用主要场景为扩展应用中。

类适配器

Target 目标角色：该角色定义把其他类转换为何种接口，也就是我们的期望接口。

Adaptee 源角色：你想把谁转换成目标角色，这个“谁”就是源角色，它是已经存在的、运行良好的类或对象，经过适配器角色的包装，它会成为一个崭新、靓丽的角色。

Adapter 适配器角色：适配器模式的核心角色，其他两个角色都是已经存在的角色，而适配器角色是需要新建立的。它的职责非常简单：把源角色转换为目标角色。怎么转换？通过继承或是类关联的方式。

对象适配器

不使用多继承或继承的方式，而是使用直接关联，或者称为委托的方式。

对象适配器和类适配器的区别：

类适配器是类间继承，对象适配器是对象的合成关系，也可以说是类的关联关系，这是两者的根本区别。实际项目中对象适配器使用到的场景相对比较多。

代码示例

```
public class Adapter extends Target
{
private Adaptee adaptee;

public Adapter(Adaptee adaptee)
{
this.adaptee=adaptee;
}

public void request()
{
adaptee.specificRequest();
```

```
}
}
```

面试题 22. 简述Javascript 迭代器模式 ?

推荐指数: ★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

它提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节。

结构

Iterator 抽象迭代器：抽象迭代器负责定义访问和遍历元素的接口，而且基本上是有固定的 3 个方法：first() 获得第一个元素，next() 访问下一个元素，isDone() 是否已经访问到底部（Java 叫做 hasNext() 方法）。

ConcreteIterator 具体迭代器：具体迭代器角色要实现迭代器接口，完成容器元素的遍历。

Aggregate 抽象容器：容器角色负责提供创建具体迭代器角色的接口，提供一个类似 createIterator() 这样的方法，在 Java 中一般是 iterator() 方法。

Concrete Aggregate 具体容器：具体容器实现容器接口定义的方法，创建出容纳迭代器的对象。

代码示例

```
/**
* 具体迭代器
*/
public class ConcreteIterator implements Iterator {

private List list = new ArrayList<>();

private int cursor = 0;

public boolean hasNext() {
return cursor != list.size();
}

public T next() {
T obj = null;
if (this.hasNext()) {
obj = this.list.get(cursor++);
}
return obj;
```

```
}

}
```

面试题 23. 简述Javascript 观察者模式？

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

使用场景

关联行为场景。需要注意的是，关联行为是可拆分的，而不是“组合”关系。

事件多级触发场景。

跨系统的消息交换场景，如消息队列的处理机制。

结构

Subject 被观察者：定义被观察者必须实现的职责，它必须能够动态地增加、取消观察者。它一般是抽象类或者是实现类，仅仅完成作为被观察者必须实现的职责：管理观察者并通知观察者。

Observer 观察者：观察者接收到消息后，即进行 update（更新方法）操作，对接收到的信息进行处理。

ConcreteSubject 具体的被观察者：定义被观察者自己的业务逻辑，同时定义对哪些事件进行通知。

ConcreteObserver 具体的观察者：每个观察者在接收到消息后的处理反应是不同的，各个观察者有自己的处理逻辑。

代码示例

```
public abstract class Subject {
//定义一个观察者数组
private Vector obsVector = new Vector();
//增加一个观察者
public void addObserver(Observer o){
this.obsVector.add(o);
}
//删除一个观察者
public void delObserver(Observer o){
```

```
this.obsVector.remove(o);
}
//通知所有观察者
public void notifyObservers(){
for(Observer o:this.obsVector){
o.update();
}
}
}
```