## 一句话概括：

**定义对象间的一种一对多关系，当一个对象的状态发生改变时，所有依赖它的对象都得到通知并被自动更新**。

关键点：一对多，一个对象改变所有依赖它的对象得到通知

## 补充介绍：

观察者模式（Observer Pattern）是对象的行为模式，又叫发布-订阅模式、源-监听器模式和从属者模式等。

两种模型（推模型和拉模型）：

■**推模型是假定主题对象知道观察者需要的数据**；**而拉模型是主题对象不知道观察者具体需要什么数据**，没有办法的情况下，干脆**把自身传递给观察者**，让观察者自己去按需要取值。

■推模型可能会使得观察者对象难以复用，因为观察者的update()方法是按需要定义的参数，可能无法兼顾没有考虑到的使用情况。这就意味着出现新情况的时候，就可能提供新的update()方法，或者是干脆重新实现观察者；而拉模型就不会造成这样的情况，因为拉模型下，update()方法的参数是主题对象本身，这基本上是主题对象能传递的最大数据集合了，基本上可以适应各种情况的需要。

## 参与角色:

1）观察者基类（也可以是接口，至少要包含一个update(Subject)方法）

2）不同观察者的实现类

3）事件主题基类（可以添加，移除观察者，还有一个可以将事件/状态通知给所有观察者的方法）

4）事件主题实现类

## 优点：

1）观察者和被观察者是抽象耦合的。

2）建立一套触发机制

## 缺点：

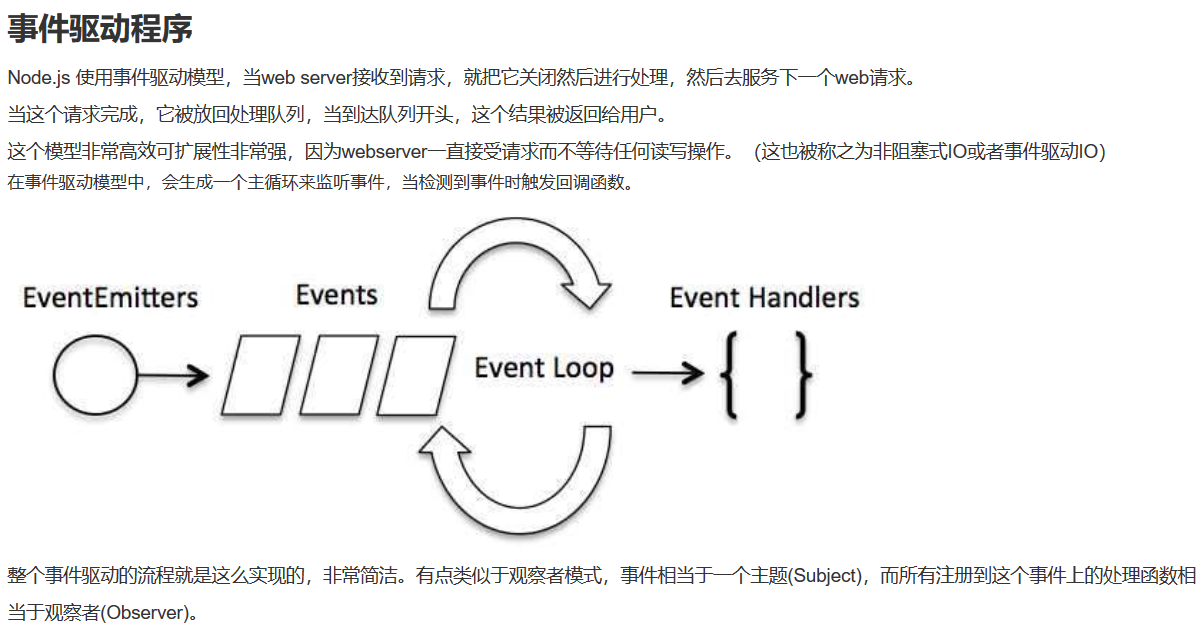
1）如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

2）如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

3）观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

## 使用案例或场景：

比如Node.js的事件驱动程序：



再比如现在很多的消息中间件，RabbitMQ，kafka等，都有发布订阅功能，当一个消息发布后，所有订阅这个消息的队列都会收到这个消息，然后消费者负责去消费这些消息。当然不能简单的将发布-订阅模式与观察者模式混为一谈，发布-订阅模式更像是观察者模式的扩展，不过概念上还是非常相似的。发布-订阅模式中，订阅方可能并不知道发布方是谁，发布方也不需要同步的将状态通知到所有订阅方，他们之间的联系需要消息代理中间件来建立。所以发布-订阅的耦合更低，更利于扩展。需要了解发布-订阅模式和观察者模式不同的朋友可以去看看这条博客：

<http://www.sohu.com/a/207062452_464084>

## 示例程序

需要源码的朋友可以前往github下载：

<https://github.com/aharddreamer/chendong/tree/master/design-patterns/demo-code/design-patterns>

程序简介：

在这段示例程序中，观察者将观察一个会生成数值的对象，并将它生成的数值结果显示出来。不过，不同的观察者显示的方式不一样。DigitObserver会以数字的形式显示数值，而GraphObserver则会以简单的图示形式来呈现数值。

类和接口一览：

Observer 表示观察者的接口

NumberGenerator 表示生成数值的对象的抽象类

RandomNumberGenerator 生成随机数的类

DigitObserver 表示以数字形式显示数值的类

GraphObserver 表示以简单的图示形式显示的数值的类

ObserverPatternTest 测试观察者模式的类

代码：

**public interface** Observer {  
 **void** update(NumberGenerator numberGenerator);  
}

**public abstract class** NumberGenerator {  
 **private** List **observers** = **new** ArrayList();  
 **public void** addObserver(Observer observer) {  
 **observers**.add(observer);  
 }  
 **public void** deleteObserver(Observer observer) {  
 **observers**.remove(observer);  
 }  
 **public void** notifyObservers() {  
 Iterator it = **observers**.iterator();  
 **while** (it.hasNext()) {  
 Observer observer = (Observer) it.next();  
 observer.update(**this**);  
 }  
 }  
 **public abstract int** getNumber();  
 **public abstract void** execute();  
}

**public class** RandomNumberGenerator **extends** NumberGenerator {  
 **private** Random **random** = **new** Random();  
 **private int number**;  
 **public int** getNumber() {  
 **return number**;  
 }  
 **public void** execute() {  
 **for** (**int** i = 0; i < 10 ; i++) {  
 **number** = **random**.nextInt(50);  
 notifyObservers();  
 }  
 }  
}

**public class** DigitObserver **implements** Observer {  
 @Override  
 **public void** update(NumberGenerator numberGenerator) {  
 *//直接打印数字* System.***out***.println(**"DigitObserver: "** + numberGenerator.getNumber());  
 *//为了方便演示，延迟100毫秒* **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

**public class** GraphObserver **implements** Observer {  
 @Override  
 **public void** update(NumberGenerator numberGenerator) {  
 *//显示n个星号 n = numberGenerator.getNumber()* System.***out***.print(**"GraphObserver: "**);  
 **for** (**int** i = 0; i < numberGenerator.getNumber(); i++) {  
 System.***out***.print(**"\*"**);  
 }  
 System.***out***.println();  
 *//为了方便演示，延迟100毫秒* **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

**public class** ObserverPatternTest {  
 **public static void** main(String[] args) {  
 NumberGenerator generator = **new** RandomNumberGenerator();  
 Observer digitObserver = **new** DigitObserver();  
 Observer graphObserver = **new** GraphObserver();  
 generator.addObserver(digitObserver);  
 generator.addObserver(graphObserver);  
 generator.execute();  
 }  
}

运行结果：

DigitObserver: 32

GraphObserver: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DigitObserver: 11

GraphObserver: \*\*\*\*\*\*\*\*\*\*\*

DigitObserver: 14

GraphObserver: \*\*\*\*\*\*\*\*\*\*\*\*\*\*

DigitObserver: 11

GraphObserver: \*\*\*\*\*\*\*\*\*\*\*

DigitObserver: 2

GraphObserver: \*\*

DigitObserver: 44

GraphObserver: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DigitObserver: 37

GraphObserver: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DigitObserver: 40

GraphObserver: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DigitObserver: 20

GraphObserver: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DigitObserver: 37

GraphObserver: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

参考：

《图解设计模式》【日】结城浩著

《观察者模式》、《Node.js事件循环》菜鸟教程网站

《观察者模式和“发布-订阅”模式有区别吗?》https://blog.csdn.net/dean\_hu/article/details/71195133