**依赖倒置原则**

依赖倒置原则的描述是：

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.  
 高层模块不应该依赖于低层模块，二者都应该依赖于抽象  
  
B. Abstractions should not depend upon details. Details should depend upon abstractions.  
 抽象不应该依赖于细节，细节应该依赖于抽象

依赖倒置原则的最重要问题就是确保应用程序或框架的主要组件从非重要的底层组件实现细节解耦出来，这将确保程序的最重要的部分不会因为低层次组件的变化修改而受影响。

该原则的第一部分是关于高层模块和低层模块之间的耦合方式，在传统的分成架构中，高层模块（封装了程序的核心业务逻辑）总依赖于低层的一些模块（一些基础点）。当应用依赖倒置原则的时候，关系就反过来了。和高层模块依赖于低层模块不同，依赖倒置是让低层模块依赖于高层模块里定义的接口。举例来说，如果要给程序进行数据持久化，传统的设计是核心模块依赖于一个持久化模块的API，而根据依赖倒置原则重构以后，则是核心模块需要定义持久化的API接口，然后持久化的实现实例需要实现核心模块定义的这个API接口。

该原则的第二部分描述的是抽象和细节之间的正确关系。理解这一部分，通过了解C++语言比较有帮助，因为他的适用性比较明显。

不像一些静态类型的语言，C++没有提供一个语言级别的概念来定义接口，那类定义和类实现之间到底是怎么样的呢，在C++里，类通过头文件的形式来定义，其中定义了源文件需要实现的类成员方法和变量。因为所有的变量和私有方法都定义在头文件里，所以可以用来抽象以便和实现细节之前解耦出来。通过定只定义抽象方法来实现（C++里是抽象基类）接口这个概念用于实现类来实现。

**DIP and JavaScript**

因为JavaScript是动态语言，所以不需要去为了解耦而抽象。所以抽象不应依赖于细节这个改变在JavaScript里没有太大的影响，但高层模块不应依赖于低层模块却有很大的影响。

在当静态类型语言的上下文里讨论依赖倒置原则的时候，耦合的概念包括语义（semantic）和物理（physical）两种。这就是说，如果一个高层模块依赖于一个低层模块，也就是不仅耦合了语义接口，也耦合了在底层模块里定义的物理接口。也就是说高层模块不仅要从第三方类库解耦出来，也需要从原生的低层模块里解耦出来。

为了解释这一点，想象一个.NET程序可能包含一个非常有用的高层模块，而该模块依赖于一个低层的持久化模块。当作者需要在持久化API里增加一个类似的接口的时候，不管依赖倒置原则有没有使用，高层模块在不重新实现这个低层模块的新接口之前是没有办法在其它的程序里得到重用的。

在JavaScript里，依赖倒置原则的适用性仅仅限于高层模块和低层模块之间的语义耦合，比如，DIP可以根据需要去增加接口而不是耦合低层模块定义的隐式接口。

为了来理解这个，我们看一下如下例子：

$.fn.trackMap = function(options) {  
 var defaults = {  
 /\* defaults \*/  
 };  
 options = $.extend({}, defaults, options);  
  
 var mapOptions = {  
 center: new google.maps.LatLng(options.latitude,options.longitude),  
 zoom: 12,  
 mapTypeId: google.maps.MapTypeId.ROADMAP  
 },  
 map = new google.maps.Map(this[0], mapOptions),  
 pos = new google.maps.LatLng(options.latitude,options.longitude);  
  
 var marker = new google.maps.Marker({  
 position: pos,  
 title: options.title,  
 icon: options.icon  
 });  
  
 marker.setMap(map);  
  
 options.feed.update(function(latitude, longitude) {  
 marker.setMap(null);  
 var newLatLng = new google.maps.LatLng(latitude, longitude);  
 marker.position = newLatLng;  
 marker.setMap(map);  
 map.setCenter(newLatLng);  
 });  
  
 return this;  
};  
  
var updater = (function() {  
 // private properties  
  
 return {  
 update: function(callback) {  
 updateMap = callback;  
 }  
 };  
})();  
  
$("#map\_canvas").trackMap({  
 latitude: 35.044640193770725,  
 longitude: -89.98193264007568,  
 icon: 'http://bit.ly/zjnGDe',  
 title: 'Tracking Number: 12345',  
 feed: updater  
});

在上述代码里，有个小型的JS类库将一个DIV转化成Map以便显示当前跟踪的位置信息。trackMap函数有2个依赖：第三方的Google Maps API和Location feed。该feed对象的职责是当icon位置更新的时候调用一个callback回调（在初始化的时候提供的）并且传入纬度latitude和精度longitude。Google Maps API是用来渲染界面的。

feed对象的接口可能按照装，也可能没有照装trackMap函数的要求去设计，事实上，他的角色很简单，着重在简单的不同实现，不需要和Google Maps这么依赖。介于trackMap语义上耦合了Google Maps API，如果需要切换不同的地图提供商的话那就不得不对trackMap函数进行重写以便可以适配不同的provider。

为了将于Google maps类库的语义耦合翻转过来，我们需要重写设计trackMap函数，以便对一个隐式接口（抽象出地图提供商provider的接口）进行语义耦合，我们还需要一个适配Google Maps API的一个实现对象，如下是重构后的trackMap函数：

$.fn.trackMap = function(options) {  
 var defaults = {  
 /\* defaults \*/  
 };  
  
 options = $.extend({}, defaults, options);  
  
 options.provider.showMap(  
 this[0],  
 options.latitude,  
 options.longitude,  
 options.icon,  
 options.title);  
  
 options.feed.update(function(latitude, longitude) {  
 options.provider.updateMap(latitude, longitude);  
 });  
  
 return this;  
};  
  
$("#map\_canvas").trackMap({  
 latitude: 35.044640193770725,  
 longitude: -89.98193264007568,  
 icon: 'http://bit.ly/zjnGDe',  
 title: 'Tracking Number: 12345',  
 feed: updater,  
 provider: trackMap.googleMapsProvider  
});

在该版本里，我们重新设计了trackMap函数以及需要的一个地图提供商接口，然后将实现的细节挪到了一个单独的googleMapsProvider组件，该组件可能独立封装成一个单独的JavaScript模块。如下是我的googleMapsProvider实现：

trackMap.googleMapsProvider = (function() {  
 var marker, map;  
  
 return {  
 showMap: function(element, latitude, longitude, icon, title) {  
 var mapOptions = {  
 center: new google.maps.LatLng(latitude, longitude),  
 zoom: 12,  
 mapTypeId: google.maps.MapTypeId.ROADMAP  
 },  
 pos = new google.maps.LatLng(latitude, longitude);  
  
 map = new google.maps.Map(element, mapOptions);  
  
 marker = new google.maps.Marker({  
 position: pos,  
 title: title,  
 icon: icon  
 });  
  
 marker.setMap(map);  
 },  
 updateMap: function(latitude, longitude) {  
 marker.setMap(null);  
 var newLatLng = new google.maps.LatLng(latitude,longitude);  
 marker.position = newLatLng;  
 marker.setMap(map);  
 map.setCenter(newLatLng);  
 }  
 };  
})();

做了上述这些改变以后，trackMap函数将变得非常有弹性了，不必依赖于Google Maps API，相反可以任意替换其它的地图提供商，那就是说可以按照程序的需求去适配任何地图提供商。

**何时依赖注入？**

有点不太相关，其实依赖注入的概念经常和依赖倒置原则混在一起，为了澄清这个不同，我们有必要来解释一下：

依赖注入是控制反转的一个特殊形式，反转的意思一个组件如何获取它的依赖。依赖注入的意思就是：依赖提供给组件，而不是组件去获取依赖，意思是创建一个依赖的实例，通过工厂去请求这个依赖，通过Service Locator或组件自身的初始化去请求这个依赖。依赖倒置原则和依赖注入都是关注依赖，并且都是用于反转。不过，依赖倒置原则没有关注组件如何获取依赖，而是只关注高层模块如何从低层模块里解耦出来。某种意义上说，依赖倒置原则是控制反转的另外一种形式，这里反转的是哪个模块定义接口（从低层里定义，反转到高层里定义）。

**总结**

这是五大原则的最后一篇了，在这5篇文字里我们看到了SOLID如何在JavaScript里实现的，不同的原则在JavaScript里通过不同的角度来说明的。