接口隔离原则的描述是：

Clients should not be forced to depend on methods they do not use.  
不应该强迫客户依赖于它们不用的方法。

当用户依赖的接口方法即便只被别的用户使用而自己不用，那它也得实现这些接口，换而言之，一个用户依赖了未使用但被其他用户使用的接口，当其他用户修改该接口时，依赖该接口的所有用户都将受到影响。这显然违反了开闭原则，也不是我们所期望的。

接口隔离原则ISP和单一职责有点类似，都是用于聚集功能职责的，实际上ISP可以被理解才具有单一职责的程序转化到一个具有公共接口的对象。

**JavaScript接口**

JavaScript下我们改如何遵守这个原则呢？毕竟JavaScript没有接口的特性，如果接口就是我们所想的通过某种语言提供的抽象类型来建立contract和解耦的话，那可以说还行，不过JavaScript有另外一种形式的接口。在[Design Patterns – Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612" \t "_blank)一书中我们找到了接口的定义：

一个对象声明的任意一个操作都包含一个操作名称，参数对象和操作的返回值。我们称之为操作符的签名（signature）。  
一个对象里声明的所有的操作被称为这个对象的接口（interface）。一个对象的接口描绘了所有发生在这个对象上的请求信息。

不管一种语言是否提供一个单独的构造来表示接口，所有的对象都有一个由该对象所有属性和方法组成的隐式接口。参考如下代码：

var exampleBinder = {};  
exampleBinder.modelObserver = (function() {  
 /\* 私有变量 \*/  
 return {  
 observe: function(model) {  
 /\* 代码 \*/  
 return newModel;  
 },  
 onChange: function(callback) {  
 /\* 代码 \*/  
 }  
 }  
})();  
  
exampleBinder.viewAdaptor = (function() {  
 /\* 私有变量 \*/  
 return {  
 bind: function(model) {  
 /\* 代码 \*/  
 }  
 }  
})();  
  
exampleBinder.bind = function(model) {  
 /\* 私有变量 \*/  
 exampleBinder.modelObserver.onChange(/\* 回调callback \*/);  
 var om = exampleBinder.modelObserver.observe(model);  
 exampleBinder.viewAdaptor.bind(om);  
 return om;  
};

上面的exampleBinder类库实现的功能是双向绑定。该类库暴露的公共接口是bind方法，其中bind里用到的关于change通知和view交互的功能分别是由单独的对象modelObserver和viewAdaptor来实现的，这些对象从某种意义上来说就是公共接口bind方法的具体实现。

尽管JavaScript没有提供接口类型来支持对象的contract，但该对象的隐式接口依然能当做一个contract提供给程序用户。

**ISP与JavaScript**

我们下面讨论的一些小节是JavaScript里关于违反接口隔离原则的影响。正如上面看到的，JavaScript程序里实现接口隔离原则虽然可惜，但是不像静态类型语言那样强大，JavaScript的语言特性有时候会使得所谓的接口搞得有点不粘性。

**堕落的实现**

在静态类型语言语言里，导致违反ISP原则的一个原因是堕落的实现。在Java和C#里所有的接口里定义的方法都必须实现，如果你只需要其中几个方法，那其他的方法也必须实现（可以通过空实现或者抛异常的方式）。在JavaScript里，如果只需要一个对象里的某一些接口的话，他也解决不了堕落实现这个问题，虽然不用强制实现上面的接口。但是这种实现依然违反了里氏替换原则。

var rectangle = {  
 area: function() {  
 /\* 代码 \*/  
 },  
 draw: function() {  
 /\* 代码 \*/  
 }  
};  
  
var geometryApplication = {  
 getLargestRectangle: function(rectangles) {  
 /\* 代码 \*/  
 }  
};  
  
var drawingApplication = {  
 drawRectangles: function(rectangles) {  
 /\* 代码 \*/  
 }  
};

当一个rectangle替代品为了满足新对象geometryApplication的getLargestRectangle 的时候，它仅仅需要rectangle的area()方法，但它却违反了LSP（因为他根本用不到其中drawRectangles方法才能用到的draw方法）。

**静态耦合**

静态类型语言里的另外一个导致违反ISP的原因是静态耦合，在静态类型语言里，接口在一个松耦合设计程序里扮演了重大角色。不管是在动态语言还是在静态语言，有时候一个对象都可能需要在多个客户端用户进行通信（比如共享状态），对静态类型语言，最好的解决方案是使用[Role Interfaces](http://martinfowler.com/bliki/RoleInterface.html)，它允许用户和该对象进行交互（而该对象可能需要在多个角色）作为它的实现来对用户和无关的行为进行解耦。在JavaScript里就没有这种问题了，因为对象都被动态语言所特有的优点进行解耦了。

**语义耦合**

导致违反ISP的一个通用原因，动态语言和静态类型语言都有，那就是语义耦合，所谓语义耦合就是互相依赖，也就是一个对象的行为依赖于另外一个对象，那就意味着，如果一个用户改变了其中一个行为，很有可能会影响另外一个使用用户。这也违反单一职责原则了。可以通过继承和对象替代来解决这个问题。

**可扩展性**

另外一个导致问题的原因是关于可扩展性，很多人在举例的时候都会举关于callback的例子用来展示可扩展性（比如ajax里成功以后的回调设置）。如果想这样的接口需要一个实现并且这个实现的对象里有很多熟悉或方法的话，ISP就会变得很重要了，也就是说当一个接口interface变成了一个需求实现很多方法的时候，他的实现将会变得异常复杂，而且有可能导致这些接口承担一个没有粘性的职责，这就是我们经常提到的胖接口。

**总结**

JavaScript里的动态语言特性，使得我们实现非粘性接口的影响力比静态类型语言小，但接口隔离原则在JavaScript程序设计模式里依然有它发挥作用的地方。