AngularJS 依赖注入(Dependency Injection)

依赖注入（DI）是一种让代码管理其依赖关系的设计模式。

关于 DI 的更多讨论，请参见维基百科的 Dependency Injection (http://en.wikipedia.org/wiki/Dependency_injection) 词条，以及 Martin Fowler 写的 Inversion of Control (http://martinfowler.com/articles/injection.html)，或查阅那些有关设计模式的书。

DI简介

对象或函数可以通过三种方式获得所依赖的对象（简称依赖）：

1. 创建依赖，通常是通过 `new` 操作符
2. 查找依赖，在一个全局的注册表中查阅它
3. 传入依赖，需要此依赖的地方等待被依赖对象注入进来

前两种方式：创建或是查找依赖都不是那么理想，因为它们都将依赖写死在对象或函数里了。问题在于，想要修改这两种方式获得依赖对象的逻辑是很困难的。尤其是在测试的时候，会遇到很多问题，因为测试时常常需要我们提供所依赖对象的替身(MOCK)。

第三种方式是最理想的，因为它免除了客户代码里定位相应的依赖这个负担，反过来，依赖总是能够很简单地被注入到需要它的组件中。

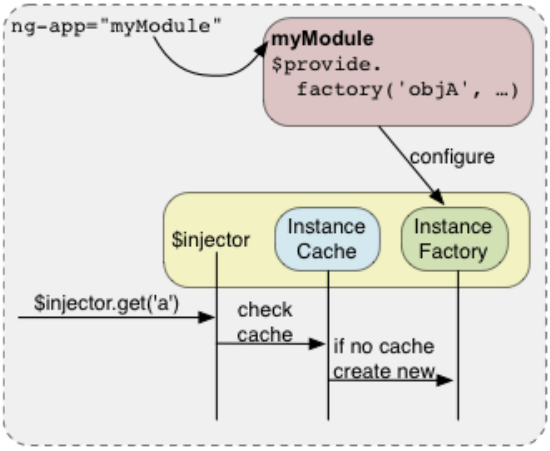
```
function SomeClass(greeter) {
  this.greeter = greeter;
}

SomeClass.prototype.doSomething = function(name) {
  this.greeter.greet(name);
}
```

上述例子中， `SomeClass` 不必在意它所依赖的 `greeter` 对象是从哪里来的，只要知道一点：在运行的时候， `greeter` 依赖已经被传进来了，直接用就是了。

这个例子中的代码虽然理想，但是它却把获得所依赖对象的大部分责任都放在了我们创建 `SomeClass` 的客户代码中。

为了分离“创建依赖”的职责，每个 Angular 应用都有一个 `injector` (api/angular.injector)对象。这个 `injector` 是一个服务定位器，负责创建和查找依赖。（译注：当你的app的某处声明需要用到某个依赖时，Angular 会调用这个依赖注入器去查找或是创建你所需要的依赖，然后返回来给你用）



下面是一个利用 `injector` 服务例子：

```
// Provide the wiring information in a module
angular.module('myModule', []).

  // 下面是教 injector 如何构建一个 'greeter' 依赖
  // 注意 greeter 本身依赖于 '$window'
  factory('greeter', function($window) {
    // 这是一个 factory 函数，负责创建 'greeter' 服务
    return {
      greet: function(text) {
        $window.alert(text);
      }
    };

  });

// 从 module 创建的 injector
// 这个常常是 Angular 启动时自动完成的
var injector = angular.injector(['myModule', 'ng']);

// 通过 injector 请求任意的依赖
var greeter = injector.get('greeter');
```

函数或对象通过请求依赖解决了硬编码的问题，但同时也就意味着 injector 需要通过应用传递，而传递 injector 破坏了 Law of Demeter (http://en.wikipedia.org/wiki/Law_of_Demeter)。为了弥补这个，我们通过像下面例子那样声明依赖，将依赖查找的任务丢给了 injector 去做：

```
<!-- Given this HTML -->
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
```

```
// And this controller definition
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter.greet('Hello World');
  };
}

// The 'ng-controller' directive does this behind the scenes
injector.instantiate(MyController);
```

注意，通过让 `ng-controller` 在背后调用 injector 初始化控制器类满足了 `MyController` 需要依赖的需求，而且可以让控制器根本不知道 injector 的存在（译注：好一招瞒天过海）。这是最好的结果了。应用中的代码简单地提交需要它需要某依赖的请求，不需要去管 injector ，而且这样也不违反迪米特法则 。

依赖注释

（译注：此处注释非代码注释，应理解为依赖声明的方法）

那么，injector 是如何知道哪些服务需要被注入呢?

应用开发者需要提供 injector 需要使用的注释信息来解析依赖。Angular 之中，按照API文档说明，某些 API 方法需要通过 injector 调用。这样，injector 要知道得往这个方法中注入什么服务。下面是用服务名信息来进行注释的三种等价的方式，它们可以互用，按照你觉得适合的情况选用相应的方式。

推断依赖

最简单的获取依赖的方法是让你的函数的参数名直接使用依赖名。

```
function MyController($scope, greeter) {
  ...
}
```

给 injector 一个函数，它可以通过检查函数声明并抽取参数名可以推断需要注入的服务名。在上面的例子中， `$scope` 和 `greeter` 是两个需要被注入到函数中的服务。

虽然这种方式很直观明了，但是它对于压缩的 JavaScript 代码来说是不起作用的，因为压缩过后的 JavaScript 代码重命名了函数的参数名。这就让这种注释方式只对 pretotyping (http://www.pretotyping.org/) 和 demo级应用有用。

`$inject` 注释

为了让重命名了参数名的压缩版的 JavaScript 代码能够正确地注入相关的依赖服务。函数需要通过 `$inject` 属性进行标注，这个属性是一个存放需要注入的服务的数组。

```
var MyController = function(renamed$scope, renamedGreeter) {
  ...
}
MyController['$inject'] = ['$scope', 'greeter'];
```

在这种场合下， `$inject` 数组中的服务名顺序必须和函数参数名顺序一致。以上述代码段为例， `$scope` 将会被注入到 'renamed$scope'，而 `greeter` 则是注入到 'renamedGreeter'。需要注意 `$inject` 注释是和真实的函数声明中的参数保持同步的。

这种注释方法对于控制器声明很有用处，因为它是把注释信息赋给了函数。

行内注释

有时候用 `$inject` 注释的方式不方便，比如标注指令的时候（译注：这里标注指令可以理解为告诉指令需要加载哪些服务依赖的说明）。

看下面的例子：

```
someModule.factory('greeter', function($window) {
  ...
});
```

由于需要一个临时的变量导致代码膨胀：

```
var greeterFactory = function(renamed$window) {
  ...
};

greeterFactory.$inject = ['$window'];

someModule.factory('greeter', greeterFactory);
```

所以，第三种注释风格被引入，如下：

```
someModule.factory('greeter', ['$window', function(renamed$window) {
  ...
}]);
```

记住，所有上面的三种依赖注释风格（译注：声明依赖的风格）是等价的，在 Angular 中任何支持依赖注入的地方都可以使用。

哪里使用DI

在 Angular 中，DI 无处不在。通常在控制器和工厂方法（译注：所谓的工厂方法个人理解为 Angular 中的API）中使用较多。

控制器中使用DI

控制器是负责应用操作逻辑的 JavaScript 类，推荐的在控制器中使用DI的方式是用数组标记风格，如下：

```
someModule.controller('MyController', ['$scope', 'dep1', 'dep2', function($scope, dep1, dep2) {
  ...
  $scope.aMethod = function() {
    ...
  }
  ...
}]);
```

如上那样，避免了在全局作用域内创建控制器，同时也避免了代码压缩时引起的注入问题。

工厂方法中使用DI

工厂方法负责创建 Angular 中的绝大多数对象。例如指令，服务，和过滤器等。工厂方法是注册在模块之下的，推荐的声明方式如下：

```
angular.module('myModule', []).
  config(['depProvider', function(depProvider){
    ...
  }]).
  factory('serviceId', ['depService', function(depService) {
    ...
  }]).
  directive('directiveName', ['depService', function(depService) {
    ...
  }]).
  filter('filterName', ['depService', function(depService) {
    ...
  }]).
  run(['depService', function(depService) {
    ...
  }]);
```