11/08/2016

Isolate scopes : There is one exception in the case that a scope is created inside of a directive, this is called the isolate scope.

At the ng-app level, this is called the $rootScope

With that one exception, all scopes are created with prototypal inheritance, meaning that they have

access to their parent scopes.

Expressions are evaluated by the $parse service, a service that is baked into AngularJS core, so we

can get access to the raw functions that are actually run to evaluate expressions using the $parse

service.  
To manually parse an expression, we can inject the $**parse** service into a controller and call the

service to do the parsing for us.

Examples

angular.module("myApp").controller('**MyCtrl**', ['$**scope**', '$**parse**', function($**scope**, $**parse**) {

$scope.$watch('expr', function(newVal, oldVal, scope) {

if (newVal !== oldVal) {

// Let's set up our parseFun with the expression

var parseFun = $parse(newVal);

// Get the value of the parsed expression

var parsedValue = parseFun(scope);

}

});

}]);

Interpolating a string in Angular JS

To run an interpolation on a string template, we’ll need to inject the $interpolate service in our

object.

Example :

angular.module('myApp').controller('MyCtrl',

['$scope', '$interpolate',function($scope, $interpolate) {

// We have access to both the $scope

// and the $interpolate services

}]);

The $interpolate service takes up to three parameters, with only one required function.

• text (string) - the text with markup to interpolate

• mustHaveExpression (boolean) - if we set this to true, then the text will return null if there is no expression

• trustedContext (string) - the result of the interpolation context will send the result through

the $sce.getTrusted() method that provides strict contextual escaping

<Page 45>

**Filters in Angular JS**

In AngularJS, a filter provides a way to format data to display to the user. Angular gives us several

built-in filters as well as an easy way to create our own.

Filters are invoked in the HTML with the | (pipe) character in the template binding characters

<span ng-non-bindable> {{ }} </span>

* example for capitalized strings we can use filters

{{ name | uppercase }}

* we can also use filters from within JavaScript by using the $filter service

example:-  
app.controller('DemoCtrl', ['$scope', '$filter', function($scope, $filter) {

$scope.name = $filter('**lowercase**')('Ari');

}]);

* To pass an argument to a filter in the html form, we pass it with a colon after the filter name

Example:-

{{ 123.456789 | number:2 }} <!-- Displays: 123.45 -->

example: {{ 123 | currency }}

**Saturday Aug 20 2016**

**Create custom directives**

**angular**.**module**('myApp', []).**directive**('**myDirective**', function() {

return {

restrict: String,

priority: Number,

terminal: Boolean,

template: String or Template Function: function(tElement, tAttrs) (...},

templateUrl: String,

replace: Boolean or String,

scope: Boolean or Object,

transclude: Boolean,

controller: String or function($scope, $element, $attrs, $transclude, otherIn\

jectables) { ... },

controllerAs: String,

require: String,

link: function(scope, iElement, iAttrs) { ... },

compile: return an Object OR function(tElement, tAttrs, transclude) {

return {

pre: function(scope, iElement, iAttrs, controller) { ... },

post: function(scope, iElement, iAttrs, controller) { ... }

}

// or

return function postLink(...) { ... }

}

};

});

**Restrict (string)**

Restrict is an optional argument. It is responsible for telling Angular which format our directive

will be declared as in the DOM. By default a custom directive is expected to be declared as an

attribute, meaning the restrict option is set to A .

The available options are as follows:

•E (an element) {lang=”html”}

•A (an attribute, default) {lang=”html”} <div my-directive=”expression”></div>

•C (a class) {lang=”html”} <div class=”my-directive: expression;”></div>

•M (a comment) {lang=”html”} <– directive: my-directive expression –>

**Scope Option (boolean|object) (Page 95 ng-App)**

The scope option is optional. It can be set to true or an object, {} . By default it is set to false .

When scope is set to true, a new scope object is created that prototypically inherits from it’s parent scope.

If multiple directives on an element provide an isolate scope, only one new scope is applied. Root

elements within the template of a directive always get a new scope, thus for those object, scope is

set to true by default.

**Transclude**

transclude is optional. If provided it must be set to true . It is set to false by default.

Transclusion is most often used for creating re-usable widgets.

Use case

example is a modal box or a navbar.

Transclude allows us to pass in an entire template, including it’s scope, to a directive. This gives us

the opportunity to pass in arbitrary contents and arbitrary scope to a directive. In order for scope to

be passed in the scope option must be isolated, {} or set to true

If the scope option is not set, then the scope available inside the directive will be applied to the template passed in.

NOTE: Only use transclude: true when you want to create a directive that wraps arbitrary

content.

**Sunday 21 Aug 2016 page 107**

**AngularJs Lifecycle( Compile and Link)**

The compile option and the link option are mutually exclusive. If both are set, then the

compile option will be set and be expected to return the link function, while the link option

will simply be ignored.

**Compile phase**

The compile phase, Angular slurps up our initial HTML page and begins processing the directives we declared according to the directive definitions we’ve defined within our application’s JavaScript.

\* Each template of each directive is pieced together, finally producing a large DOM tree

\* Each directive can have a template, that may contain directives, which themselves may have their own templates

\* Once a directive and it’s child templates are have been walked or compiled, a function is returned

for the compiled template known as the **template function.**

\* Before the template function for a directive is returned however, we have the opportunity to modify the existing functionality to the compiled DOM tree.

\* During this phase built-in directives, such as **ng-repeat** and **ng-transclude** advantage of this and manipulate the DOM before it has been bound to any **scope** data.

Example

**ng-repeat** , for example, will loop over the array or object it has been given building out the full

representation of the **DOM** before passing the result off for data binding. If we we’re building

an unordered list using **ng-repeat** , where each **<li>** was decorated with the **ng-click** directive,

this process provides us with performance that is orders of magnitude faster especially as our list

approaches 100 elements.

\* Once a complete representation of the final DOM tree, including it’s attributes, known as **template attributes**( **tAttr** ) and it’s **elements**( **tElement** ) has been built for the **directive** and it’s **child directives**, a function is returned. This compiled tree of DOM nodes is referred to as a **template function** hence the **t prefix** for the arguments in it’s method signature.

\* This **template** function is then passed to the **link** function, where **scope**, determined by the directive definition rules of each directive in the compiled DOM tree, is applied all at once.

\* Once scope has been applied, we can have full access to the live representation of the DOM within the **link** and **controller** options available to us within our **directive** definition.

Eg:

compile: function(tEle, tAttrs) {

var tplEl = angular.element('<div>' +

'<h2></h2>' +

'</div>');

var h2 = tplEl.find('h2');

h2.attr('type', tAttrs.type);

h2.attr('ng-model', tAttrs.ngModel);

h2.val("hello");

tEle.replaceWith(tplEl);

return function(scope, ele, attrs) {

// The link function

}

}

TODO// cleanup - attach specified behavior to that DOM element - transform the DOM element

and its children - traverses the DOM matching directives against the DOM elements

**Meaning of compiling a html template**

- attaching event listeners to the HTML to make it interactive

- recursive process of attaching directives mirrors the process of compiling source code

•transforms the template DOM

•not used often since most directives don’t do template transformation

•examples are ng-repeat (transforms template DOM)

•ng-view - loads the contents asynchronosuly

- tElement - template element

– tAttrs - template attributes - attrs defined on this element, shared between all directive

compile functions

– transclude - A transclude linking function: function(scope, cloneLinkingFn)

\* template instance and link instance may be different objects if the template has been cloned.

Thus only do DOM transformation that are safe to do to all cloned DOM nodes within the

compile function. Don’t do DOM listener registration, that should be done in the linking

function.

Has a return value that can either be a function or an object

– returning a postLink function:

\* same thing as registering the the linking function via the link property when the

compile function is empty.

– returning an object:

\* with functions registered via pre and post properties - allows you to control when

a linking function should be called during the linking phase(more info below)

**The compile function deals with transforming the template DOM**

####**The link function deals with linking scope to the DOM.**

Thursday Aug 25

When we use AngularJS’s dependency injection and inject a service object in our Controller, Angular uses $injector to find corresponding service injector. Once it get a hold on service injector, it uses $get method of it to get an instance of service object. Sometime the service provider needs certain info in order to instantiate service object.

Angular resolve option

resolve: {

'data': ['$http', function($http) {

return $http.get('/api').then(

function success(resp) { return response.data; }

function error(reason) { return false; }

);

}]

};

If the resolve property is set, then Angular will inject the elements of the map into the controller. If

these dependencies are promises, then they will be resolved and set as a value before the controller

is loaded and before the $routeChangeSuccess event is fired.

**Map object is**

The map object can be:

• key where the key is the string name of a dependency that will be injected into the controller

• factory where the factory can either be a string of the name of a service, a function whose

result is injected into the controller, or a promise that is to be resolved and the value is injected

into the controller.

$routeParams

Controller asks modal service to .open() a modal window.

1. Modal service prepares modal data and emits an "open" event.
2. Directive listens for "open" event and renders the appropriate modal.
3. Modal window manages its own view-modal and user-interactions.
4. Modal window calls .resolve() or .reject() on the modal service.
5. Modal service resolves or rejects the relevant promise.
6. Original controller reacts to the resolution of the promise