|  |  |
| --- | --- |
| When writing an Angular directive, one can use any of the following functions to manipulate the DOM behaviour, contents and look of the element on which the directive is declared:   * compile * controller * pre-link * post-link   There seem to be some confusion as for which function should one use. | |
|  | **In which order the directive functions are executed?**  For a single directive  Based on the following [plunk](http://plnkr.co/edit/wIhRr0), consider the following HTML markup:  <body>  <div log='some-div'></div>  </body>  With the following directive declaration:  myApp.directive('log', function() {  return {  controller: function( $scope, $element, $attrs, $transclude ) {  console.log( $attrs.log + ' (controller)' );  },  compile: function compile( tElement, tAttributes ) {  console.log( tAttributes.log + ' (compile)' );  return {  pre: function preLink( scope, element, attributes ) {  console.log( attributes.log + ' (pre-link)' );  },  post: function postLink( scope, element, attributes ) {  console.log( attributes.log + ' (post-link)' );  }  };  }  };  });  The console output will be:  some-div (compile)  some-div (controller)  some-div (pre-link)  some-div (post-link)  We can see that compile is executed first, then controller, then pre-link and finally followed by post-link.  For nested directives  **Note:** The following does not apply to directives that render their children in their link function. Quite a few Angular directives (like ngIf, ngRepeat, or any directive with transclude) do so. These type of directives will natively have their link function called *before* their child directives compile is called.  The original HTML markup is often made of nested elements, each with its own directive. Like in the following markup (see [plunk](http://plnkr.co/edit/pWJhfgdA2iDzSKV5bjyh)):  <body>  <div log='parent'>  <div log='..first-child'></div>  <div log='..second-child'></div>  </div>  </body>  The console output will look like this (without the comments):  // The compile phase  parent (compile)  ..first-child (compile)  ..second-child (compile)  // The link phase  parent (controller)  parent (pre-link)  ..first-child (controller)  ..first-child (pre-link)  ..first-child (post-link)  ..second-child (controller)  ..second-child (pre-link)  ..second-child (post-link)  parent (post-link)  We can distinguish two phases here - the *compile* phase and the *link* phase.  The compile phase  When the DOM is loaded Angular starts the compile phase, where it traverses the markup top-down, and calls compile on all directives. Graphically, we could express it like so:  An image illustrating the compilation loop for children  It is perhaps important to mention that at this stage, the templates the compile function gets are the source templates (not instance template).  The link phase  DOM instances are often simply the result of a source template being rendered to the DOM, but they may be created by ng-repeat, or introduced on the fly.  Whenever a new instance of an element with a directive is rendered to the DOM, the link phase starts.  In this phase, Angular calls controller, pre-link, iterates children, and call post-link on all directives, like so:  An illustration demonstrating the link phase steps |
|  |

|  |  |
| --- | --- |
|  | **What else happens between these function calls?**  The various directive functions are executed from within two other angular functions called $compile(where the directive's compile is executed) and and internal function called nodeLinkFn (where the directive's controller, preLink and postLink are executed). Various things happen within the angular function before and after the directive functions are called. Perhaps most notably is the child recursion. The following simplified illustration shows key steps within the compile and link phases:  An illustration showing Angular compile and link phases  To demonstrate the these steps, let's use the following HTML markup:  <div ng-repeat="i in [0,1,2]">  <my-element>  <div>Inner content</div>  </my-element>  </div>  With the following directive:  myApp.directive( 'myElement', function() {  return {  restrict: 'EA',  transclude: true,  template: '<div>{{label}}<div ng-transclude></div></div>'  }  });  Compile  The compile API looks like so:  compile: function compile( tElement, tAttributes ) { ... }  Often the parameters are prefixed with t to signify the elements and attributes provided are those of the source template, rather than that of the instance.  Prior to the call to compile transcluded content (if any) is removed, and the template is applied to the markup. Thus, the element provided to the compile function will look like so:  <my-element>  <div>  "{{label}}"  <div ng-transclude></div>  </div>  </my-element>  Notice that the transcluded content is not re-inserted at this point.  Following the call to the directive's .compile, Angular will traverse all child elements, including those that may have just been introduced by the directive (the template elements, for instance).  Instance creation  In our case, three instances of the source template above will be created (by ng-repeat). Thus, the following sequence will execute three times, once per instance.  Controller  The controller API involves:  controller: function( $scope, $element, $attrs, $transclude ) { ... }  Entering the link phase, the link function returned via $compile is now provided with a scope.  First, the link function create a child scope (scope: true) or an isolated scope (scope: {...}) if requested.  The controller is then executed, provided with the scope of the instance element.  Pre-link  The pre-link API looks like so:  function preLink( scope, element, attributes, controller ) { ... }  Virtually nothing happens between the call to the directive's .controller and the .preLink function. Angular still provide recommendation as to how each should be used.  Following the .preLink call, the link function will traverse each child element - calling the correct link function and attaching to it the current scope (which serves as the parent scope for child elements).  Post-link  The post-link API is similar to that of the pre-link function:  function postLink( scope, element, attributes, controller ) { ... }  Perhaps worth noticing that once a directive's .postLink function is called, the link process of all its children elements has completed, including all the children's .postLink functions.  This means that by the time .postLink is called, the children are 'live' are ready. This includes:   * data binding * transclusion applied * scope attached   The template at this stage will thus look like so:  <my-element>  <div class="ng-binding">  "{{label}}"  <div ng-transclude>  <div class="ng-scope">Inner content</div>  </div>  </div>  </my-element> |
|  | **How to declare the various functions?**  Compile, Controller, Pre-link & Post-link  If one is to use all four function, the directive will follow this form:  myApp.directive( 'myDirective', function () {  return {  restrict: 'EA',  controller: function( $scope, $element, $attrs, $transclude ) {  // Controller code goes here.  },  compile: function compile( tElement, tAttributes, transcludeFn ) {  // Compile code goes here.  return {  pre: function preLink( scope, element, attributes, controller, transcludeFn ) {  // Pre-link code goes here  },  post: function postLink( scope, element, attributes, controller, transcludeFn ) {  // Post-link code goes here  }  };  }  };  });  Notice that compile returns an object containing both the pre-link and post-link functions; in Angular lingo we say the compile function returns a *template function*.  Compile, Controller & Post-link  If pre-link isn't necessary, the compile function can simply return the post-link function instead of a definition object, like so:  myApp.directive( 'myDirective', function () {  return {  restrict: 'EA',  controller: function( $scope, $element, $attrs, $transclude ) {  // Controller code goes here.  },  compile: function compile( tElement, tAttributes, transcludeFn ) {  // Compile code goes here.  return function postLink( scope, element, attributes, controller, transcludeFn ) {  // Post-link code goes here  };  }  };  });  Sometimes, one wishes to add a compile method, after the (post) link method was defined. For this, one can use:  myApp.directive( 'myDirective', function () {  return {  restrict: 'EA',  controller: function( $scope, $element, $attrs, $transclude ) {  // Controller code goes here.  },  compile: function compile( tElement, tAttributes, transcludeFn ) {  // Compile code goes here.  return this.link;  },  link: function( scope, element, attributes, controller, transcludeFn ) {  // Post-link code goes here  }  };  });  Controller & Post-link  If no compile function is needed, one can skip its declaration altogether and provide the post-link function under the link property of the directive's configuration object:  myApp.directive( 'myDirective', function () {  return {  restrict: 'EA',  controller: function( $scope, $element, $attrs, $transclude ) {  // Controller code goes here.  },  link: function postLink( scope, element, attributes, controller, transcludeFn ) {  // Post-link code goes here  },  };  });  No controller  In any of the examples above, one can simply remove the controller function if not needed. So for instance, if only post-link function is needed, one can use:  myApp.directive( 'myDirective', function () {  return {  restrict: 'EA',  link: function postLink( scope, element, attributes, controller, transcludeFn ) {  // Post-link code goes here  },  };  }); |
|  |  |

|  |  |
| --- | --- |
|  | **What is the difference between a *source template* and an *instance template*?**  The fact that Angular allows DOM manipulation means that the input markup into the compilation process sometimes differ from the output. Particularly, some input markup may be cloned a few times (like with ng-repeat) before being rendered to the DOM.  Angular terminology is a bit inconsistent, but it still distinguishes between two types of markups:   * **Source template** - the markup to be cloned, if needed. If cloned, this markup will not be rendered to the DOM. * **Instance template** - the actual markup to be rendered to the DOM. If cloning is involved, each instance will be a clone.   The following markup demonstrates this:  <div ng-repeat="i in [0,1,2]">  <my-directive>{{i}}</my-directive>  </div>  The source html defines  <my-directive>{{i}}</my-directive>  which serves as the source template.  But as it is wrapped within an ng-repeat directive, this source template will be cloned (3 times in our case). These clones are instance template, each will appear in the DOM and be bound to the relevant scope. |
|  |  |
|  | **Compile function**  Each directive's compile function is only called once, when Angular bootstraps.  Officially, this is the place to perform (source) template manipulations that do not involve scope or data binding.  Primarily, this is done for optimisation purposes; consider the following markup:  <tr ng-repeat="raw in raws">  <my-raw></my-raw>  </tr>  The <my-raw> directive will render a particular set of DOM markup. So we can either:   * Allow ng-repeat to duplicate the source template (<my-raw>), and then modify the markup of each instance template (outside the compile function). * Modify the source template to involve the desired markup (in the compile function), and then allow ng-repeat to duplicate it.   If there are 1000 items in the raws collection, the latter option may be faster than the former one.  Do:   * Manipulate markup so it serves as a template to instances (clones).   Do not   * Attach event handlers. * Inspect child elements. * Set up observations on attributes. * Set up watches on the scope. |
|  |  |

|  |  |
| --- | --- |
|  | **Controller function**  Each directive's controller function is called whenever a new related element is instantiated.  Officially, the controller function is where one:   * Defines controller logic (methods) that may be shared between controllers. * Initiates scope variables.   Again, it is important to remember that if the directive involves an isolated scope, any properties within it that inherit from the parent scope are not yet available.  Do:   * Define controller logic * Initiate scope variables   Do not:   * Inspect child elements (they may not be rendered yet, bound to scope, etc.). |
|  |  |
|  | **Post-link function**  When the post-link function is called, all previous steps have taken place - binding, transclusion, etc.  This is typically a place to further manipulate the rendered DOM.  Do:   * Manipulate DOM (rendered, and thus instantiated) elements. * Attach event handlers. * Inspect child elements. * Set up observations on attributes. * Set up watches on the scope. |
|  |  |

|  |  |
| --- | --- |
|  | **Pre-link function**  Each directive's pre-link function is called whenever a new related element is instantiated.  As seen previously in the compilation order section, pre-link functions are called parent-then-child, whereas post-link functions are called child-then-parent.  The pre-link function is rarely used, but can be useful in special scenarios; for example, when a child controller registers itself with the parent controller, but the registration has to be in a parent-then-child fashion (ngModelController does things this way).  Do not:   * Inspect child elements (they may not be rendered yet, bound to scope, etc.). |