**指令里的compile和link的本质**

**在angularjs中，指令是怎么被处理的：**

当一个浏览器渲染一个页面时，其本质就是读取页面的html标签，创建DOM模型，当DOM准备好以后，传播一个ready事件。

如果你使用<script></script>标签将angularjs程序代码包含到一个页面，angularjs会监听ready事件，只要它监听到ready事件后，它就开始遍历DOM模型，找到 ng-app属性。

当找到带有ng-app属性的元素后，angularjs使用找到的这个元素作为起点开始处理DOM模型 。因此，如果将ng-app属性设置在<html>元素中,angularjs将会从<html>元素开始编译DOM。

从ng-app元素开始，angularjs会递归地审查所有的子元素。寻找所有在你的angularjs程序中定义的指令(directives).

Angularjs如何处理元素取决于指令定义对象的具体内容。你可以定义一个compile函数，或者是一个link函数，甚至是两者都定义。也可以用pre-link和post-link函数代替link函数。

So what is difference between all those functions and why or when should you use them?

那么，什么时候使用什么函数和为什么你要使用它们呢？ 跟我来：

**The code**

为了解析它们的区别，我使用例子来表达：

看看线面的html代码：

<level-one>

<level-two>

<level-three>

Hello {{name}}

</level-three>

</level-two>

</level-one>

下面是javascript代码：

var app = angular.module('plunker', []);

function createDirective(name){

return function(){

return {

restrict: 'E',

compile: function(tElem, tAttrs){

console.log(name + ': compile');

return {

pre: function(scope, iElem, iAttrs){

console.log(name + ': pre link');

},

post: function(scope, iElem, iAttrs){

console.log(name + ': post link');

}

}

}

}

}

}

app.directive('levelOne', createDirective('levelOne'));

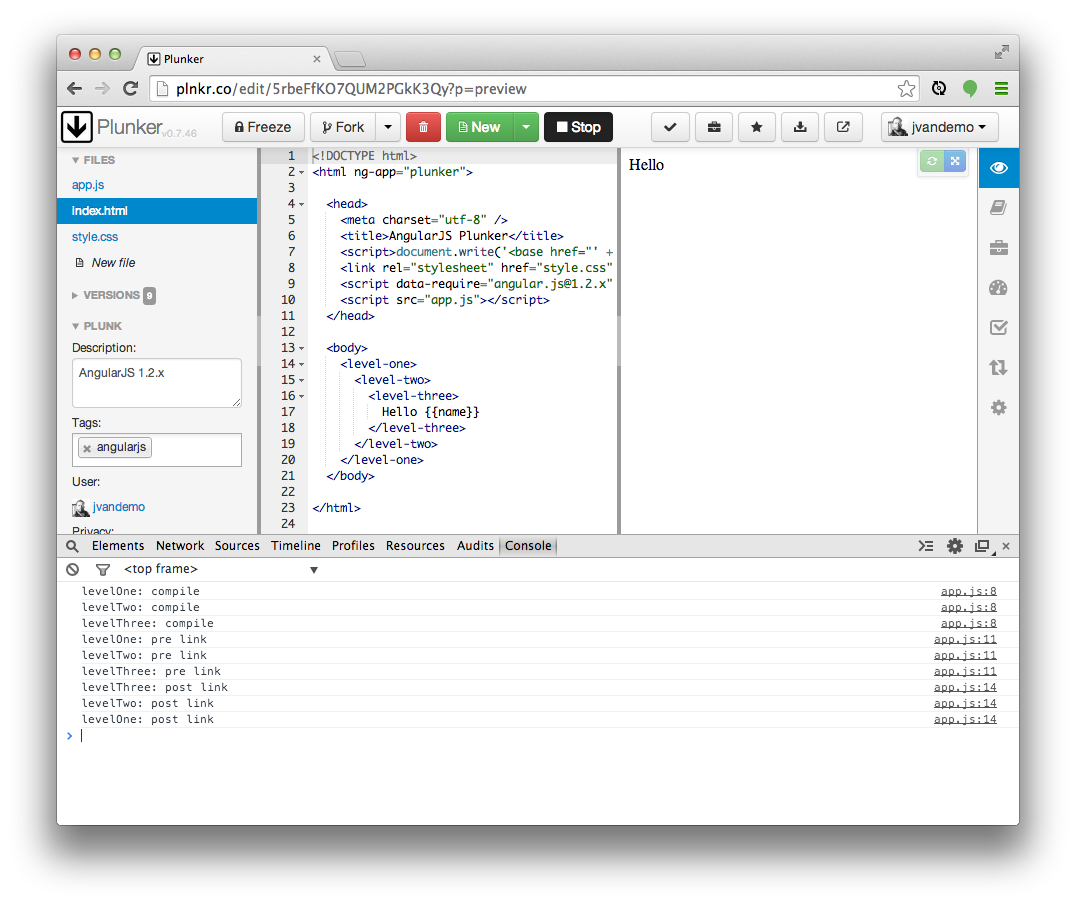
app.directive('levelTwo', createDirective('levelTwo'));

app.directive('levelThree', createDirective('levelThree'));

这段代码的目的很简单：就是找出compile pre-link post-link 函数的执行顺序。

**The output**

这里就是控制台输出的截屏:



**Let's analyze**

The first thing to pay attention to is the**order of the function calls(注意函数调用的顺序)**:

// COMPILE PHASE

// levelOne: compile function is called

// levelTwo: compile function is called

// levelThree: compile function is called

// PRE-LINK PHASE

// levelOne: pre link function is called

// levelTwo: pre link function is called

// levelThree: pre link function is called

// POST-LINK PHASE (Notice **the reverse order**)

// levelThree: post link function is called

// levelTwo: post link function is called

// levelOne: post link function is called

This clearly demonstrates how AngularJS first compiles all directives **before it links them to their scope(Angular在将指令link到它们的作用域之前，先compile)**, and that the link phase **is split up(link阶段分为两个阶段)** in a pre-link and post-link phase.

Notice how the order of the compile and pre-link functions calls **is identical** but the order of the post-link function calls is **reversed.(compile和pre-link是按顺序一一对应的，而post-link是相反的顺序)**

So at this point we can already clearly identify the different phases, but what is the difference between the compile and pre-link function? They run in the same order, so why are they split up?

**The DOM**

To dig a bit deeper, let's update our JavaScript so it also outputs the element's DOM during each function call:

var app = angular.module('plunker', []);

function createDirective(name){

return function(){

return {

restrict: 'E',

compile: function(tElem, tAttrs){

console.log(name + ': compile => ' + tElem.html());

return {

pre: function(scope, iElem, iAttrs){

console.log(name + ': pre link => ' + tElem.html());

},

post: function(scope, iElem, iAttrs){

console.log(name + ': post link => ' + tElem.html());

}

}

}

}

}

}

app.directive('levelOne', createDirective('levelOne'));

app.directive('levelTwo', createDirective('levelTwo'));

app.directive('levelThree', createDirective('levelThree'));

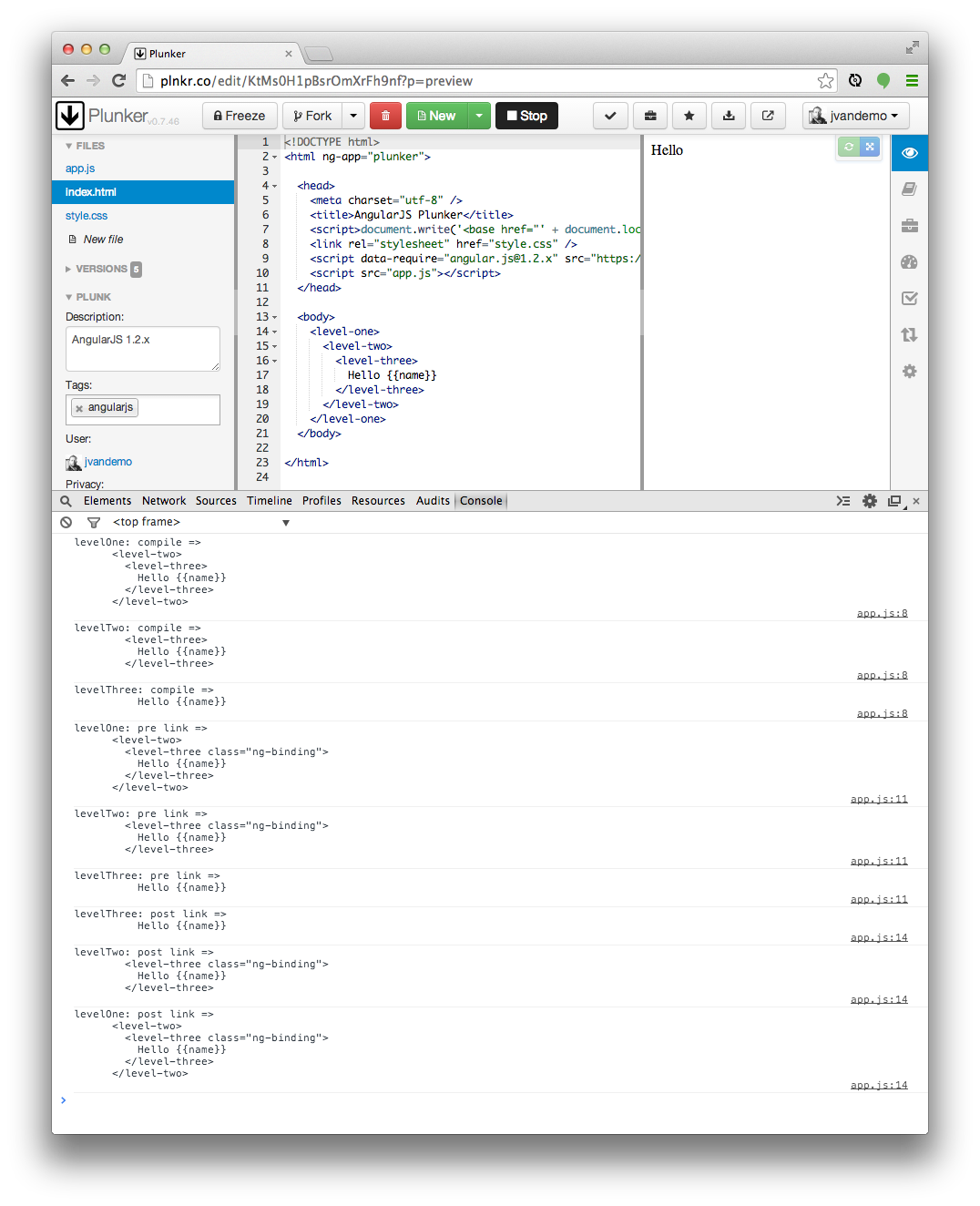
Notice the extra output in the console.log lines. Nothing else has changed and the original markup is still used.

This should provide us with more insights into the context of the functions.

Let's run the code again.

**The output**

Here is a screenshot of the output with the newly added code:



**Observation**

Printing the DOM reveals something very interesting: the DOM is different **during thecompile and pre-link function(在compile和pre-link阶段，DOM是不一样的)**.

**So what is happening here?**

Compile

We already learned that AngularJS processes the DOM when**it detects** that the DOM is ready(当**监测到DOM结构完成**，Angular开始处理DOM).

So when AngularJS starts traversing the DOM, it bumps into the <level-one> element and knows from its directive definition that some action needs to be performed.

Because a compile function is defined in the levelOne directive definition object, it is called and the element's DOM is passed as an **argument(元素作为一个参数，传进compile函数)** to the function.

If you look closely you can see that, at this point, the DOM of the element**is still the DOM(仍是初始化的dom结构)** that is initially created by the browser using the original HTML markup.

In AngularJS the original DOM is often referred to as the**the template element(模版元素)**, hence also the reason I personally use tElem as the parameter name in the compile function, which stands for template element.

Once the compile function of the levelOne directive has run, AngularJS **recursively traverses deeper into the DOM(往更深的DOM结构遍历)** and repeats the same compilation step for the <level-two>and <level-three> elements.

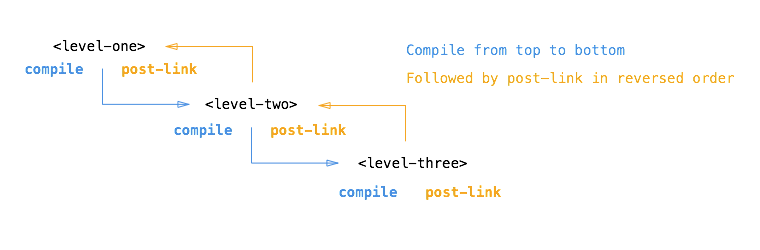
Post-link

Before digging into the pre-link functions, let's first have a look at the post-linkfunctions.

If you create a directive that **only has a link function(只有一个link函数，Angular会将其视作post-link函数)**, AngularJS treats the function as a post-link function. Hence the reason to discuss it here first.

After AngularJS travels down the DOM and**has run all** the compile functions, it **traverses back up again(当Angular已经运行完所有的compile函数，会倒着遍历DOM并运行相关联的post-link函数)** and runs all associated post-link functions.

The DOM is now traversed in the opposite direction and thus the post-link functions **are called in reverse order(post-link是倒着的顺序)**. So while the reversed order looked strange a few minutes ago, it is now starting to make perfect sense.



This reverse order guarantees that the post-link functions of all child elements have run **by the time(在父元素运行post-link函数之前，所有子元素上面面的post-link都已经运行过了)** the post-link function of the parent element is run.

So when the post-link function of <level-one> is executed, we are guaranteed that thepost-link function of <level-two> and the post-link function of <level-three> have already run.

This is **the exact reason(这是为什么建议将指令的逻辑放在post-link里面)** why it is considered the *safest* and *default* place to add your directive logic.

But what about the element's DOM? Why is it different here?

Once AngularJS has called **the compile function of a directive(一旦angular已经运行了指令的compile阶段)**, it 1)  **creates an *instance*element** of the template element (often referred to as *stamping out instances*) and 2) **provides a scope** for the instance. The scope can be a new scope or an existing one, a child scope or an isolate scope, depending on the scope property of the corresponding directive definition object.

**一旦Angular运行过compile函数**，它就1) 生成模版元素的实例 2)提供该实例的scope

So **by the time(在link函数运行之前，生成的实例和scope已经准备好了，在post-link阶段会作为参数传入post-link函数中)** the linking occurs, the instance element and scope are already available and they are passed by AngularJS as arguments to the post-link function.

I personally always use iElem as parameter name in a *link function* to refer to the element instance.

So the post-link function (and pre-link function) receive **the instance element(在post-link或pre-link函数，接受实例元素作为参数)** as argument instead of the template element.

Hence the difference in the log output.

Pre-link

When writing a post-link function, you are guaranteed that the post-link functions of all child elements have already been executed.

In most cases that makes perfect sense and therefore it is the most often used place to write directive code.

However, AngularJS provides an additional *hook*, the pre-link function, where you can run code before any of the child element's post-link functions have run.

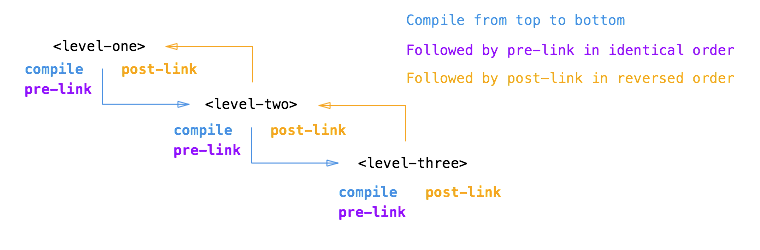
That is worth repeating:

The pre-link function is guaranteed to run on an element instance before any post-link function of its child elements has run.

So while it made perfect sense for post-link functions to be called in reverse order, it now makes perfect sense to call all pre-link functions in the original order again.

This also implies that a pre-link function of an element is run before any of its child elements pre-link functions as well, so for the sake of completeness:

A pre-link function of an element is guaranteed to run **before any pre-link or post-link function of any of its child elements(pre-link函数在任何后代元素的pre-link或post-link函数之前运行)**.



**Looking back**

If we now look back at the original output, we can clearly recognize what is happening:

// HERE THE ELEMENTS **ARE STILL** THE ORIGINAL TEMPLATE ELEMENTS / 在compile阶段，这里所有的元素**还是模版元素**

// COMPILE PHASE //未编译

// levelOne: compile function is called on original DOM

// levelTwo: compile function is called on original DOM

// levelThree: compile function is called on original DOM

// AS OF HERE, THE ELEMENTS HAVE BEEN INSTANTIATED AND // **在pre-link，所有的元素1)已经被实例化 2) 已经被绑定到一个scope上面**

// ARE BOUND TO A SCOPE

// (E.G. NG-REPEAT WOULD HAVE MULTIPLE INSTANCES)

// PRE-LINK PHASE //已经编译

// levelOne: pre link function is called on element instance

// levelTwo: pre link function is called on element instance

// levelThree: pre link function is called on element instance

// POST-LINK PHASE (Notice **the reverse order相反顺序)//已经编译**

// levelThree: post link function is called on element instance

// levelTwo: post link function is called on element instance

// levelOne: post link function is called on element instance

**Summary**

In retrospect we can describe the different functions and their use cases as follows:

Compile function

Use the compile function to change the original DOM (template element)**before(compile函数可以更改初始的DOM/模版元素，在Angular生成实例和绑定scope之前)**AngularJS creates an instance of it and before a scope is created.

While there can be **multiple** element instances, there is **only one(可以有许多个元素实例，却只有一个模版元素)** template element. The**ng-repeat directive is a perfect example(ng-repeat是很好地例子，一个模板元素，却有好多元素实例)** of such a scenario. That makes the compile function the perfect place to make changes to the DOM that should be applied to all instances later on, because it will only be run once and thus greatly enhances performance if you are stamping out a lot of instances.

**The template element** and **attributes** are passed to the compile function as arguments, but **no scope is available yet(模版元素，模板属性作为参数传入compile函数，scope此时还未生成)**:

/\*\*

\* Compile function

\*

\* @param tElem - **template element(模板元素)**

\* @param tAttrs - **attributes of** the template element(**模板元素的属性**)

\*/

function(tElem, tAttrs){

// ...

};

Pre-link function

Use the pre-link function to implement logic that runs when AngularJS has already compiled the child elements, but before any of the child element's post-link functions have been called.

The scope, instance element and instance attributes are passed to the pre-link function as arguments:

/\*\*

\* Pre-link function

\*

\* @param scope - **scope associated with this istance(实例元素的作用域)**

\* @param iElem - **instance element(实例元素)**

\* @param iAttrs - **attributes of the instance element(实例元素的特性)**

\*/

function(scope, iElem, iAttrs){

// ...

};

Post-link function

Use the post-link function to execute logic, knowing that all child elements have been compiled and**all pre-link and post-link functions of child elements**have been executed.

This is the reason the post-link function is considered the *safest* and *default* place for your code.

**The scope, instance element and instance attributes are passed to the post-link function as arguments**:

/\*\*

\* Post-link function

\*

\* @param scope - scope associated with this istance**(实例元素的作用域)**

\* @param iElem - instance element**(实例元素)**

\* @param iAttrs - attributes of the instance element**(实例元素的特性)**

\*/ function(scope, iElem, iAttrs){ // ... };

**Conclusion**

By now you should hopefully have a clear understanding of the differences between the compile, pre-link and post-link function inside directives.

If not and you are serious about AngularJS development, I would highly recommend reading the article again until you have a firm grasp of how it works.

Understanding this important concept will make it easier to understand how the native AngularJS directives work and how you can optimize your own custom directives.

And if you are still in doubt and have additional questions, please feel free to leave a comment below.

----------------------------------------------------------------------

**Transclusion.**

A mysterious word I had never heard of before until I met AngularJS. I seriously thought[Misko Hevery](https://twitter.com/mhevery) had invented the word himself, but it appeared to be an existing word:

In computer science, transclusion is **the inclusion** of a document or part of a document**into another document** by reference .

In AngularJS, transclusion is the mechanism that allows you to grab the content of the DOM element of your directive and include it anywhere in the directive's template.

So in the context of AngularJS, we could rephrase the original definition as:

In AngularJS, transclusion is the inclusion of the directive's DOM element content into the directive's template

In this article we investigate the influence of transclusion on the compile, pre-link andpost-link functions inside an AngularJS directive.

**The code**

Because this article is an extension to part 1, we will use the same code we already discussed previously.

Consider the following HTML markup:

<level-one>

<level-two>

<level-three>

Hello {{name}}

</level-three>

</level-two>

</level-one>

and the following JavaScript:

var app = angular.module('plunker', []);

function createDirective(name){

return function(){

return {

restrict: 'E',

compile: function(tElem, tAttrs){

console.log(name + ': compile');

return {

pre: function(scope, iElem, iAttrs){

console.log(name + ': pre link');

},

post: function(scope, iElem, iAttrs){

console.log(name + ': post link');

}

}

}

}

}

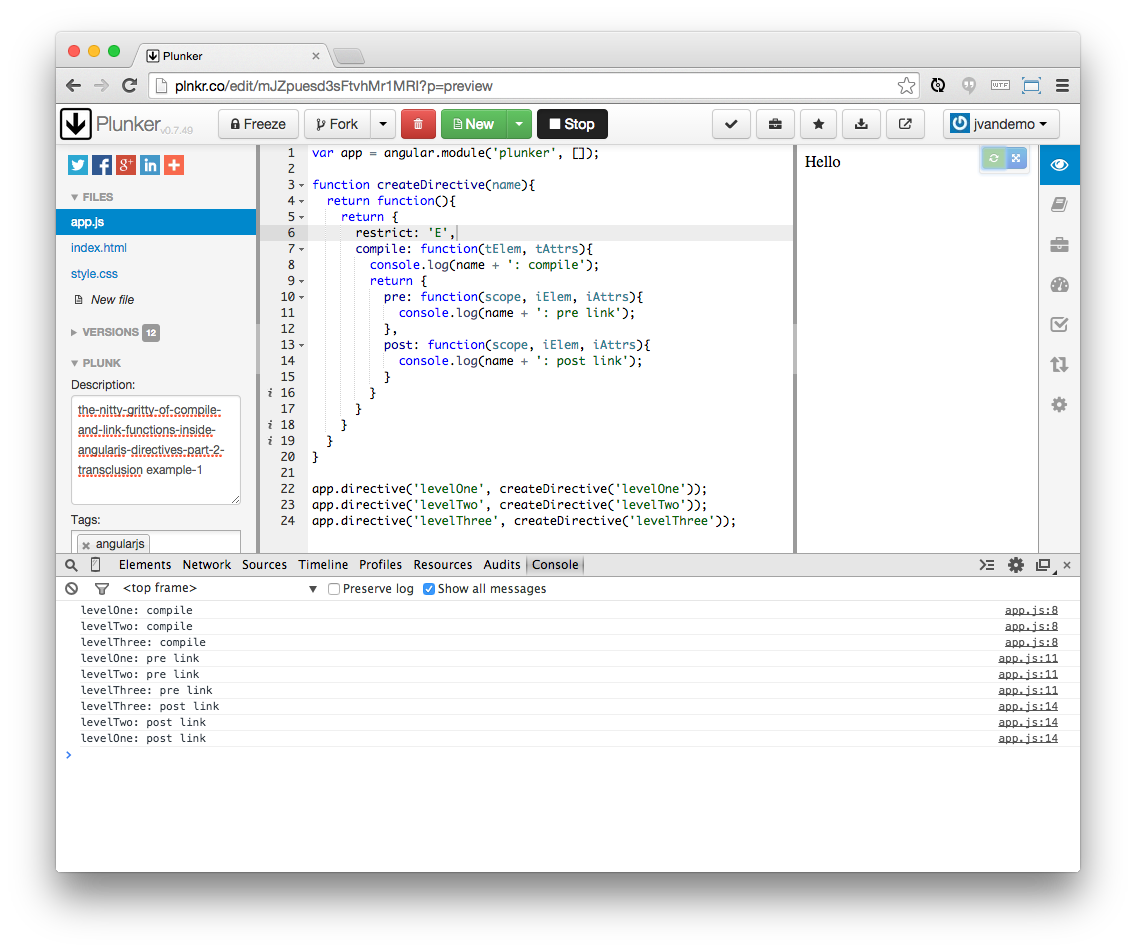
}

app.directive('levelOne', createDirective('levelOne'));

app.directive('levelTwo', createDirective('levelTwo'));

app.directive('levelThree', createDirective('levelThree'));

providing the following console output:



which can be summarized as:

// COMPILE PHASE

// levelOne: compile function is called

// levelTwo: compile function is called

// levelThree: compile function is called

// PRE-LINK PHASE

// levelOne: pre link function is called

// levelTwo: pre link function is called

// levelThree: pre link function is called

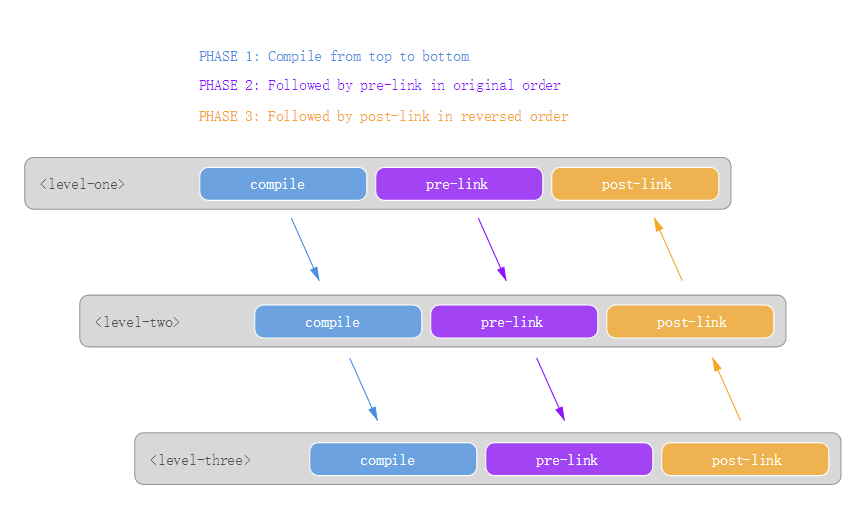
// POST-LINK PHASE (Notice the reverse order)

// levelThree: post link function is called

// levelTwo: post link function is called

// levelOne: post link function is called

and visualized as:



**So let's add transclusion**

Transclusion is enabled by adding a transclude property to the **directive definition object(定义指令的对象)**:

var app = angular.module('plunker', []);

function createDirective(name){

return function(){

return {

restrict: 'E',

transclude: true,

compile: function(tElem, tAttrs){

console.log(name + ': compile');

return {

pre: function(scope, iElem, iAttrs){

console.log(name + ': pre link');

},

post: function(scope, iElem, iAttrs){

console.log(name + ': post link');

}

}

}

}

}

}

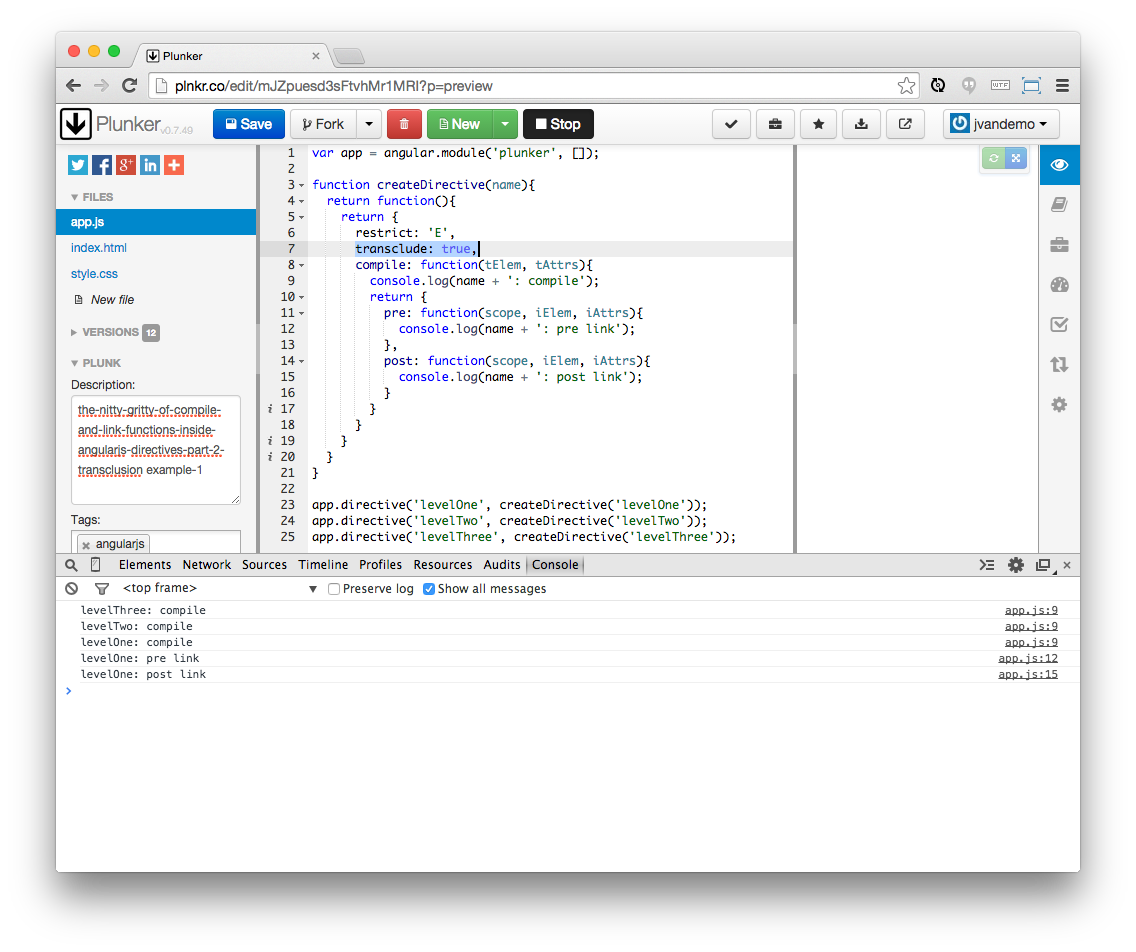
app.directive('levelOne', createDirective('levelOne'));

app.directive('levelTwo', createDirective('levelTwo'));

app.directive('levelThree', createDirective('levelThree'));

Adding transclude: true tells AngularJS to **capture the content of the directive** and **make it available in the directive's template**. The ng-transclude attribute can then be used inside the template to **specify where(transclude告诉Angular，内容放在哪里)** you want AngularJS to restore the content.

But before we add the ng-transclude attribute to template, let's have a look at what we get so far:



**Interesting**

First of all notice how the order of the compile functions is reversed:

// COMPILE PHASE (Notice the reverse order)

// levelThree: compile function is called

// levelTwo: compile function is called

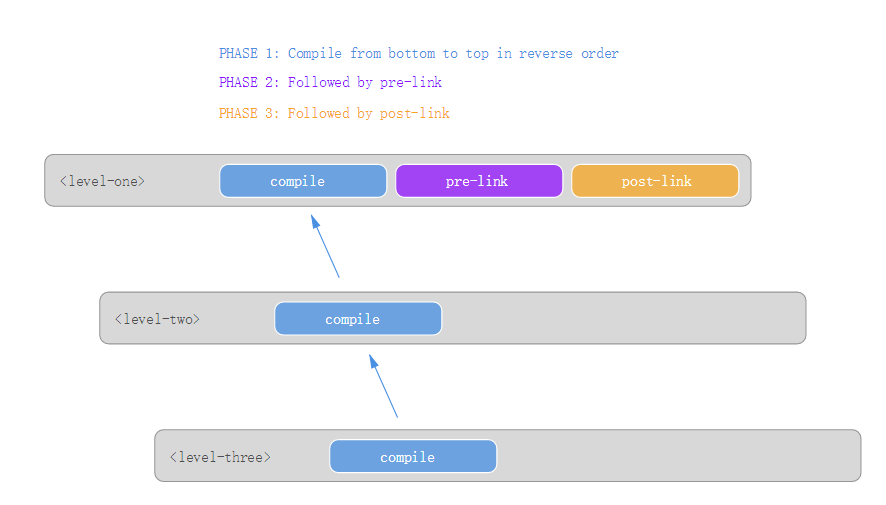
// levelOne: compile function is called

// PRE-LINK PHASE

// levelOne: pre link function is called

// POST-LINK PHASE

// levelOne: post link function is called

and even though we haven't transcluded the actual content yet using ng-transclude, allcompile functions are called:

**So what is happening?**

When AngularJS processes the levelOne directive, it sees that the transclude property is set to true in the definition object.

AngularJS now knows it first needs to process the content of the directive's element before it can make the processed content available inside the template.

To accomplish that, AngularJS first needs to process all child elements. So it starts travelling down the element's DOM.

When processing levelOne's content, it encounters the levelTwo directive and recursively repeats the same process until it has no more child elements to process.

As soon as the complete child DOM has been processed, AngularJS is ready to start applying the directives' compile, post-link and pre-link functions.

Since we haven't specified an ng-transclude attribute yet, the processed content is never put back into the DOM and we end up with a *black hole* where all child elements oflevelOne disappear.

This is rarely useful in real situations, but for the purpose of experimentation it demonstrates how all compile functions are called, even if the content is not effectively transcluded.

**Now let's add ng-transclude**

To get a complete picture of what's happening, let's add a template property to our directive definition object with a string value of <div ng-transclude></div> to tell AngularJS where to put the transcluded content:

var app = angular.module('plunker', []);

function createDirective(name){

return function(){

return {

restrict: 'E',

transclude: true,

template: '<div ng-transclude></div>',

compile: function(tElem, tAttrs){

console.log(name + ': compile');

return {

pre: function(scope, iElem, iAttrs){

console.log(name + ': pre link');

},

post: function(scope, iElem, iAttrs){

console.log(name + ': post link');

}

}

}

}

}

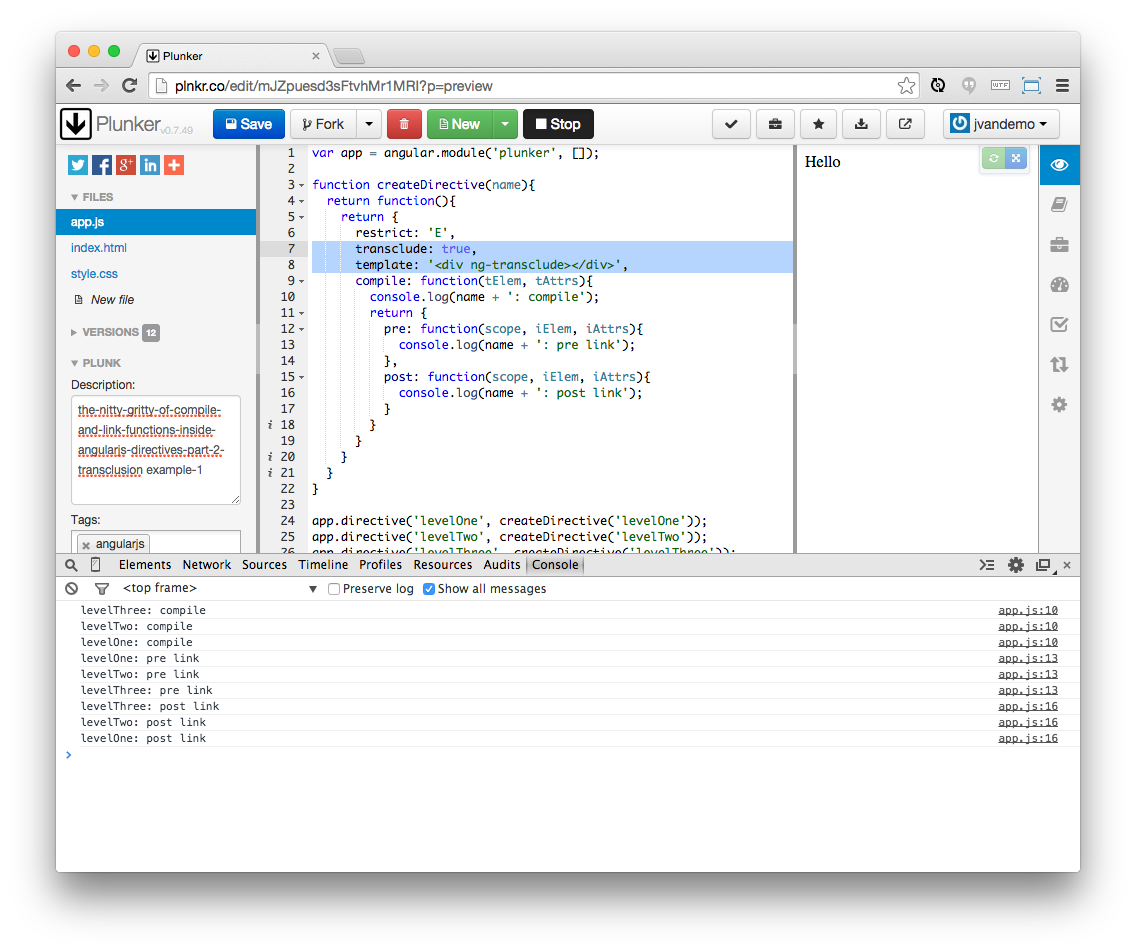
}

app.directive('levelOne', createDirective('levelOne'));

app.directive('levelTwo', createDirective('levelTwo'));

app.directive('levelThree', createDirective('levelThree'));

and check the output again:



**Let's analyze further**

If we summarize the output:

// COMPILE PHASE (Notice the reverse order)

// levelThree: compile function is called

// levelTwo: compile function is called

// levelOne: compile function is called

// PRE-LINK PHASE

// levelOne: pre link function is called

// levelTwo: pre link function is called

// levelThree: pre link function is called

// POST-LINK PHASE (Notice the reverse order)

// levelThree: post link function is called

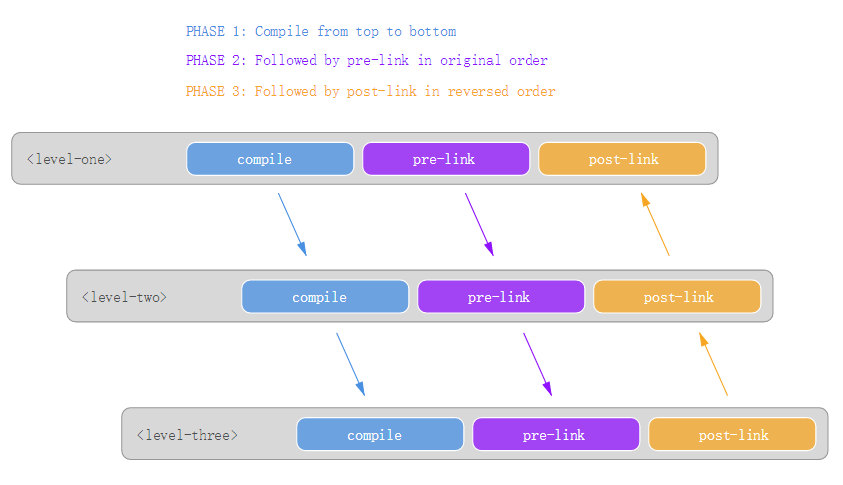
// levelTwo: post link function is called

// levelOne: post link function is called

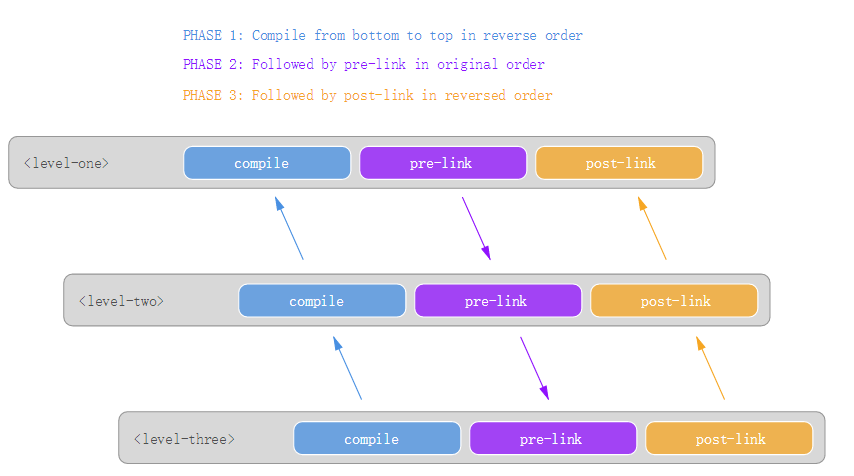
we can see that transclusion appears to reverse the order in which the compile functions are called.

Let's compare the difference visually:

**Without transclusion**



**With transclusion**



So why is transclusion reversing the order in which the compile functions are called?

If we look back at the initial output and how we defined transclusion earlier:

In AngularJS, transclusion is the inclusion of the directive's DOM element content into the directive's template

then we can quickly deduce that AngularJS needs to process the element's DOM content before it can make it available inside the template.

However, the element's child elements can also contain directives that apply transclusion themselves.

So AngularJS has to recursively traverse the DOM first to check if transclusion is enabled in child elements and then compile the DOM backwards to make sure all DOM changes correctly *"bubble up"* again to the top before the processed DOM is ready to be added to the original directive's template.

The initial *black hole* that we created before we included the ng-transclude attribute is a perfect example of this.

Finally, when compilation has finished, the pre-link and post-link functions are called in the same way as explained in [part 1](http://jvandemo.com/the-nitty-gritty-of-compile-and-link-functions-inside-angularjs-directives).

 ---------------上面的这篇看不太明白----

[Understanding the transclude option of directive definition?](http://stackoverflow.com/questions/15296284/understanding-the-transclude-option-of-directive-definition)

Consider a directive called *myDirective* in an element, and **that element is enclosing some other content(带指令的元素包含有其它元素)**, let's say:

<div my-directive>

<button>some button</button>

<a href="#">and a link</a>

</div>

If *myDirective* is using a template, you'll see that the content of <div my-directive> will be replaced by your directive template. So having:

app.directive('myDirective', function(){

return{

template: '<div class="something"> This is my directive content</div>'

}

});

will result**in this render(在浏览器中会渲染成这样，模版元素中的其它元素内容“没了”)**:

<div class="something"> This is my directive content</div>

Notice that the content of your original element <div my-directive> **will be lost** (or better said, replaced).

So, say good-bye to these buddies:

<button>some button</button>

<a href="#">and a link</a>

So, what if you want to keep your <button>... and <a href>... in the DOM? You'll need something called transclusion. The concept is pretty simple: **Include the content from one place into another**. So now your directive will look something like this:

app.directive('myDirective', function(){

return{

transclude: true,

template: '<div class="something" ng-transclude> This is my directive content</div>'

}

});

This would render:

<div class="something"> This is my directive content

<button>some button</button>

<a href="#">and a link</a>

</div>.

In conclusion, you basically use transclude when you want to**preserve the contents of an element when you're using a directive(当你想保留原先的内容时)**.

(The End)