深入研究 angular-forms

<https://indepth.dev/posts/1143/a-thorough-exploration-of-angular-form>

<https://indepth.dev/posts/1199/understanding-the-magic-behind-ngrx-store>

内容  
1 基础的实体  
2 ControlValueAccessor  
3 连接FormControl与ControlValueAccessor

4 模板驱动表单和响应式表单  
5 验证器  
6 探索内置ControlValueAccessors  
7 更好地理解AbstractControl树  
8在提交时AbstractControl树会发生什么?  
9 CSS类是如何根据AbstractControl的状态添加的?  
10 结论

## 基础的实体  
为了充分利用Forms API，我们必须确保浏览了它的一些基本部分。

1 AbstractControl

这个(抽象)类包含了FormControl、FormGroup和FormArray共享的逻辑:

运行验证器

更改和计算UI状态- markAsDirty(), markAsTouched(), dirty, touched, pristine etc...

重置状态

跟踪验证状态(invalid, valid)

这个类，以及它的子类，可以被称为模型层——它存储与特定实体相关的数据。

多个AbstractControls可以看作是一棵树，叶子总是是FormControl实例，其他两个(FormArray, FormGroup)可以看作是AbstractControl容器，这就要求它们不能用作叶子，因为它们至少必须包含AbstractControl实例。

// FG - FormGroup // FA - FormArray // FC - FormControl

FG

/ \

FC FG

/ \

FC FA

/ | \

FC FC FC

上面的树可以是结果

<form>

<input type="text" formControlName="companyName">

<ng-container formGroupName="personal">

<input type="text" formControlName="name">

<ng-container formArrayName="hobbies">

<input type="checkbox" formControlName="0">

<input type="checkbox" formControlName="1">

<input type="checkbox" formControlName="2">

</ng-container>

</ng-container>

</form>

2 FormControl

它扩展了AbstractControl，这意味着它将继承上面列出的所有特征。这里需要强调的是，FormControl只与一个表单控件(一个DOM元素:<input>， <textarea>)或一个自定义组件(通过ControlValueAccessor的帮助——后面会详细介绍!)放在一起。

如果一个FormControl不属于AbstractControl树，它可以被认为是独立的。因此，它将是完全独立的，这意味着它的有效性、值和用户交互不会影响它的任何表单容器祖先

3 FormArray

它扩展了AbstractControl，其工作是将多个AbstractControl分组在一起。

从树的角度来看，该节点必须至少包含一个后代。它的验证状态、脏度、触摸状态和值通常取决于它的后代。但是，在某些情况下，容器具有某些验证器，因此可能在该节点级别出现错误。

它的定义特征是将子对象存储在数组中。

4 FormGroup

与FormArray相同，只是它将其后代存储在对象中。

AbstractControlDirective

它是基于表单控件的指令(NgModel、FormControlName、FormControlDirective)的基类，包含了反映绑定控件当前状态的布尔getter (valid、touched、dirty等)。

前面提到的控件通过AbstractControlDirective(NgModel, FormControlName)的具体实现和一个ControlValueAccessor绑定到一个DOM元素。

因此，可以把这个类看作是连接ControlValueAccessor(视图层)和AbstractControl(模型层)的中间商——在后面的小节中将有更多的介绍。

值得一提的是，多个AbstractControl指令可以将同一个AbstractControl绑定到多个DOM元素或定制组件，绑定到多个ControlValueAccessors。

<form>

<input ngModel name="option" value="value1" type="radio">

<input ngModel="value3" name="option" value="value2" type="radio">

<input ngModel="value1" name="option" value="value3" type="radio">

</form>

顺便提醒一下，可以通过把最后一个ngModel指令的值设置为你想要默认选中的单选按钮的值来直接在模板中提供一个默认值。在上面的代码片段中，将选中第一个按钮。

之所以会出现这种情况，是因为最后一个指令将是最终调用setUpControl()函数的指令。

export function setUpControl(control: FormControl, dir: NgControl): void {

if (!control) \_throwError(dir, 'Cannot find control with');

if (!dir.valueAccessor) \_throwError(dir, 'No value accessor for form control with');

/\* ... \*/

dir.valueAccessor !.writeValue(control.value);

/\* ... \*/

}

5 AbstractFormGroupDirective

AbstractFormGroupDirective的容器和AbstractControlDirective实例及其有用,当你想创建一个sub-group AbstractControls(例如:address: { city, street, zipcode })或某些特定AbstractControls运行验证器(例如:min-max验证器,确保最小控制不可能的值大于最大控件的值)。

它的具体实现有:formGroupName、formArrayName、ngModelGroup。

<form [formGroup]="filterForm">

<ng-container formGroupName="price">

<input formControlName="min" type="text">

<input formControlName="max" type="text">

</ng-container>

</form>

<>

FormGroupName是AbstractFormGroupDirective的子类，它拥有本节开头列出的所有属性。它还充当AbstractControl实例的容器。

但是，FormGroup只能是顶级容器。这意味着，不能使用FormGroupName作为顶级容器，因为这会导致错误。

AbstractFormGroupDirective提供了一种访问顶级FormGroup实例的方法:

get formDirective(): Form|null { return this.\_parent ? this.\_parent.formDirective : null; }

this.\_parent可以是另一个AbstractFormGroupDirective或FormGroupDirective实例。FormGroupDirective没有\_parent属性

###[ControlValueAccessor](https://indepth.dev/a-thorough-exploration-of-angular-forms/" \l "controlvalueaccessor)

ControlValueAccessor是表单API的重要部分，可以看作是视图层。

它的工作是将一个DOM元素(如<input>， <textarea>)或一个自定义组件(如<app-custom-input>)与一个AbstractControlDirective(如:NgModel, FormControlName)连接起来。AbstractControlDirective最终将成为ControlValueAccessor(视图层)和AbstractControl(模型层)之间的桥梁。这样，这两个层就可以相互作用了。

例如:  
当用户输入一个输入时: View -> Model

当值通过编程方式设置时(FormControl.setValue('newValue')): Model -> View

只有FormControl实例可以“directly”与ControlValueAccessor交互，因为在AbstractControls树中，FormControl只能是叶节点，因为它不应该包含其他节点。沿着这些线，我们可以推断出来自视图的更新将从叶节点开始。

// FG - FormGroup

// FA - FormArray

// FC - FormControl

FG

/ \

user typing into an input <- FC FA

/ | \

FC FC FC <- user selecting checkbox

ControlValueAccessor接口是这样的:

export interface ControlValueAccessor {

writeValue(obj: any): void;

registerOnChange(fn: any): void;

registerOnTouched(fn: any): void;

setDisabledState?(isDisabled: boolean): void;

}

writeValue() - 向元素写入新值;新值来自模型 (FormControl.setValue -> ControlValueAccessor.writeValue -> update element -> change is visible in the UI)

registerOnChange() - 注册一个回调函数，当UI中的值发生变化时，该函数将被调用，并将新值传播到模型。

registerOnTouched() - 注册一个在模糊事件发生时将被调用的回调函数;这个事件会通知FormControl，因为当这个事件发生时，它可能需要执行一些更新。

setDisabledState - 将根据提供的值禁用/启用DOM元素;此方法通常在模型发生更改时被调用。

在下面的章节中，你可以看到这些方法的有用之处:连接FormControl和ControlValueAccessor。

有三种类型的ControlValueAccessors:

default

@Directive({

selector:

'input:not([type=checkbox])[formControlName],textarea[formControlName],input:not([type=checkbox])[formControl],textarea[formControl],input:not([type=checkbox])[ngModel],textarea[ngModel],[ngDefaultControl]',

})

export class DefaultValueAccessor implements ControlValueAccessor { }

built-in

const BUILTIN\_ACCESSORS = [

CheckboxControlValueAccessor,

RangeValueAccessor,

NumberValueAccessor,

SelectControlValueAccessor,

SelectMultipleControlValueAccessor,

RadioControlValueAccessor,

];

您可以在探索内置ControlValueAccessors中阅读更多关于内置访问器的内容。  
自定义——当您希望自定义组件成为AbstractControl树的一部分时

@Component({

selector: 'app-custom-component',

providers: [

{

provide: NG\_VALUE\_ACCESSOR,

useExisting: CustomInputComponent,

multi: true,

}

]

/\* ... \*/

})

export class CustomInputComponent implements ControlValueAccessor { }

<form>

<app-custom-component ngModel name="name"></app-custom-component>

</form>

记住，ngModel是一个基于表单控件的指令，所以它将成为一个连接ControlValueAccessor(视图)和FormControl(模型)的桥梁。

###[Connecting FormControl with ControlValueAccessor](https://indepth.dev/a-thorough-exploration-of-angular-forms/" \l "connecting-formcontrol-with-controlvalueaccessor)

连接FormControl与ControlValueAccessor

正如前面提到的，为了有效地与模型层(AbstractControl，具体地说就是FormControl)通信，视图层(ControlValueAccessor)需要AbstractControlDirective，反之亦然。  
这个连接可以如下所示:

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

| |

| ControlValueAccessor |  <--- View Layer

| |

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

| ▲

| |

| |

▼ |

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

| |

| AbstractControlDirective |

| |

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

| ▲

| |

| |

▼ |

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

| |

| FormControl |  <--- Model Layer

| |

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

↓表示ViewToModelPipeline，↑表示ModelToViewPipeline。  
AbstractControlDirective在这里扮演了一个关键角色。让我们检查一下实际的实现!

上面的图是这段代码的结果:  
注意:实际上，NgControl扩展了AbstractControlDirective，它主要提供基于表单控制的指令:NgModel、FormControlName等……，但没有任何默认实现。

每次初始化一个基于表单控件的指令时，都会调用setUpControl函数。

export function setUpControl(control: FormControl, dir: NgControl): void {

if (!control) \_throwError(dir, 'Cannot find control with');

if (!dir.valueAccessor) \_throwError(dir, 'No value accessor for form control with');

control.validator = Validators.compose([control.validator !, dir.validator]);

control.asyncValidator = Validators.composeAsync([control.asyncValidator !, dir.asyncValidator]);

dir.valueAccessor !.writeValue(control.value);

setUpViewChangePipeline(control, dir);

setUpModelChangePipeline(control, dir);

setUpBlurPipeline(control, dir);

/\* ... Skipped for brevity ... \*/

}

// VIEW -> MODEL

function setUpViewChangePipeline(control: FormControl, dir: NgControl): void {

dir.valueAccessor !.registerOnChange((newValue: any) => {

control.\_pendingValue = newValue;

control.\_pendingChange = true;

control.\_pendingDirty = true;

if (control.updateOn === 'change') updateControl(control, dir);

});

}

// Update the MODEL based on the VIEW's value

function updateControl(control: FormControl, dir: NgControl): void {

if (control.\_pendingDirty) control.markAsDirty();

//' {emitModelToViewChange: false} '将确保' ControlValueAccessor.writeValue '不会被调用

// 同样，因为值已经更新了，因为这个更改来自视图

control.setValue(control.\_pendingValue, {emitModelToViewChange: false});

// 如果你有类似' <input [(ngModel)]="myValue"> '的语句

// 这将允许' myValue '是来自视图的新值

dir.viewToModelUpdate(control.\_pendingValue);

control.\_pendingChange = false;

}

// MODEL -> VIEW

function setUpModelChangePipeline(control: FormControl, dir: NgControl): void {

control.registerOnChange((newValue: any, emitModelEvent: boolean) => {

// control -> view

dir.valueAccessor !.writeValue(newValue);

// control -> ngModel

if (emitModelEvent) dir.viewToModelUpdate(newValue);

});

}

下面又是ControlValueAccessor接口:

export interface ControlValueAccessor {

writeValue(obj: any): void;

registerOnChange(fn: any): void;

registerOnTouched(fn: any): void;

setDisabledState?(isDisabled: boolean): void;

}

正如您所看到的，setUpViewChangePipeline方法是AbstractControlDirective(dir参数)通过给ControlValueAccessor.onChange分配回调函数来连接视图和模型(单向连接)的方式。这将允许视图中发生的操作被传播到模型中。

下面是ControlValueAccessor.registerOnChange的一个具体实现:

@Directive({

selector: 'input[custom-value-accessor][type=text][ngModel]',

host: {

'(input)': 'onChange($event.target.value)',

}

})

export class CustomValueAccessor {

registerOnChange(fn: (\_: any) => void): void { this.onChange = fn; }

}

setUpModelChangePipeline将允许AbstractControlDirective将模型与视图连接起来。这意味着每次调用FormControl. setvalue()时，在该FormControl中注册的所有回调函数都将被调用，以便根据新模型的值更新该视图。

注意，我说的是所有的回调函数。这是因为多个AbstractControlDirective可以使用同一个FormControl实例。

// Inside `FormControl`

\_onChange: Function[] = [];

registerOnChange(fn: Function): void { this.\_onChange.push(fn); }

// FormControl.setValue

setValue(value: any, options: {

onlySelf?: boolean,

emitEvent?: boolean,

emitModelToViewChange?: boolean,

emitViewToModelChange?: boolean

} = {}): void {

(this as{value: any}).value = this.\_pendingValue = value;

if (this.\_onChange.length && options.emitModelToViewChange !== false) {

this.\_onChange.forEach(

(changeFn) => changeFn(this.value, options.emitViewToModelChange !== false));

}

this.updateValueAndValidity(options); // Update ancestors

}

Here's an example:

<form>

<input type="radio" ngModel name="genre" value="horror">

<input type="radio" ngModel name="genre" value="comedy">

</form>

settupcontrol (control, dir)会被调用两次，每一个ngModel调用一次。但是，在每次调用中，control(一个FormControl实例)参数是相同的。这意味着control.onChanges将包含两个回调函数，一个对应一个ControlValueAccessor(<input type="radio">绑定了RadioControlValueAccessor)。

作为附注，ControlValueAccessor.registerOnTouched遵循与ControlValueAccessor.registerOnChange相同的原则:

// Called inside `setUpControl`

function setUpBlurPipeline(control: FormControl, dir: NgControl): void {

dir.valueAccessor !.registerOnTouched(() => {

control.\_pendingTouched = true;

if (control.updateOn === 'blur' && control.\_pendingChange) updateControl(control, dir);

if (control.updateOn !== 'submit') control.markAsTouched();

});

}

<>

将允许模型在视图中出现模糊事件时进行更新。

###[Template Driven Forms and Reactive Forms](https://indepth.dev/a-thorough-exploration-of-angular-forms/" \l "template-driven-forms-and-reactive-forms)

这两种策略都非常强大，但在我看来，响应式表单在处理复杂的动态逻辑时更方便。

Template Driven Forms

使用这种策略时，与表单构造有关的大多数逻辑都是在视图中执行的。这意味着在构建视图的同时创建AbstractControl树。

以下是我们在遵循模板驱动方法时可以使用的工具:

export const TEMPLATE\_DRIVEN\_DIRECTIVES: Type<any>[] =

[NgModel, NgModelGroup, NgForm];

<>

NgModel

它是一个基于表单控制的指令，连接视图层和模型层(FormControl)，反之亦然。它还将FormControl注册到AbstractControl树中。

当使用这个指令时，你也可以指定一些选项:

@Input('ngModelOptions')

options !: {name?: string, standalone?: boolean, updateOn?: 'change' | 'blur' | 'submit'};

<>

如果你想使用一个独立的FormControl实例，你可以遵循以下方法:

<form #f="ngForm">

<input [ngModelOptions]="{ standalone: true }" #myNgModel="ngModel" name="name" ngModel type="text">

</form>

{{ myNgModel.value }}

<br>

{{ f.value | json }}

NgModelGroup

提供了一种将多个NgModel和NgModelGroup指令分组的方法。在模型层中，这由一个非顶级FormGroup实例表示。

它还将FormGroup注册到AbstractControl树中。

<form> <!-- `NgForm` - automatically bound to `<form>` -->

<input type="text" ngModel name="companyName"/>

<div ngModelGroup="personal">

<input type="text" ngModel name="name"/>

<div ngModelGroup="address">

<input type="text" ngModel name="city"/>

<input type="text" ngModel name="street" />

</div>

</div>

</form>

<>

第一个出现的NgModelGroup必须是NgForm的子元素:

<!-- Valid -->

<form>

<ng-container #myGrp="ngModelGroup" ngModelGroup="address">

<input type="text"ngModel name="city" />

<input type="text" ngModel name="street">

</ng-container>

</form>

<>

<!-- Invalid: `No provider for ControlContainer ...` -->

<div #myGrp="ngModelGroup" ngModelGroup="address">

<input type="text"ngModel name="city" />

<input type="text" ngModel name="street">

</div>

NgForm

它对多个NgModel和NgModelGroup指令进行分组。在模型层中，它由顶级实例表示，因此它监听特定于表单的事件，如重置和提交。而且，它会自动绑定到<form>标签。

在模型中，这是AbstractControl树的根FormGroup实例。

<form> <!-- NgForm -->

<input ngModel name="companyName" type="text"> <!-- NgModel -->

<div ngModelGroup="address"> <!-- NgModelGroup -->

<input ngModel name="city" type="text"> <!-- NgModel -->

<input ngModel name="street" type="text"> <!-- NgModel -->

</div>

</form>

<>

Reactive Forms

与模板驱动表单相反，当使用响应式表单时，表单在创建视图时就已经创建好了。

以下是我们可以使用的工具，当遵循这种反应性方法时:

export const REACTIVE\_DRIVEN\_DIRECTIVES: Type<any>[] =

[FormControlDirective, FormGroupDirective, FormControlName, FormGroupName, FormArrayName];

FormControlDirective

它是一个基于表单控制的指令，它是视图和模型这两个主要层之间的桥梁。

它接收一个已经同步的FormControl实例([FormControl]="formControlInstance")，因为formControlInstance已经是现有AbstractControl树的一部分了。因此，这里要做的重要的事情就是使用值访问器将formControlInstance绑定到当前的DOM元素。

如果你想使用一个独立的FormControl实例，你可以遵循以下方法:

<input #f="ngForm" [formControl]="formControlInstance" type="text">

{{ f.value }}

FormGroupDirective

在模型层中，它是一个顶级FormGroup实例(<form [FormGroup]="formGroupInstance">)。这也意味着它监听特定于表单的事件，比如重置和提交。formGroupInstance是已经构建的AbstractControl树的根。

FormControlName

它接收一个字符串作为参数([formControlName]="nameOfFormControlInstance")，它的任务是根据提供的控件名(nameOfFormControlInstance)和视图中的位置来确定这个FormControl实例。如果没有根据路径找到FormControl实例，将抛出一个错误。

因此，nameOfFormControlInstance必须是一个有效的名称，因为它依赖于表单容器来正确地将这个FormControl添加到AbstractControl树中。

正如前面提到的，路径是根据DOM元素(或自定义组件)的位置和formcontrolinstance的名称推断出来的:

// control - is, in this case, the top level `FormGroup` instance

function \_find(control: AbstractControl, path: Array<string|number>| string, delimiter: string) {

if (path == null) return null;

if (!(path instanceof Array)) {

path = (<string>path).split(delimiter);

}

if (path instanceof Array && (path.length === 0)) return null;

return (<Array<string|number>>path).reduce((v: AbstractControl | null, name) => {

if (v instanceof FormGroup) {

return v.controls.hasOwnProperty(name as string) ? v.controls[name] : null;

}

if (v instanceof FormArray) {

return v.at(<number>name) || null;

}

return null;

}, control);

}

<form [formGroup]="myFormGroup">

<!-- path: 'name' -->

<input formControlName="name" type="text">

<!-- path: 'address' -->

<ng-container formGroupName="address">

<!-- path: ['address', 'city'] -->

<input formControlName="city" type="text">

<!-- path: ['address', 'street'] -->

<input formControlName="street" type="text">

</ng-container>

</form>

这就是每个指令的路径是如何确定的:

export function controlPath(name: string, parent: ControlContainer): string[] {

return [...parent.path !, name];

}

<>

值得一提的是nameofformcontrolininstance不能是动态的。一旦添加了FormControl，它就不能被自动更改。

原因如下:

@Directive({selector: '[formControlName]', providers: [controlNameBinding]})

export class FormControlName extends NgControl implements OnChanges, OnDestroy {

/\* ... \*/

ngOnChanges(changes: SimpleChanges) {

if (!this.\_added) this.\_setUpControl();

}

private \_setUpControl() {

this.\_checkParentType();

// formDirective - points to the top-level `FormGroup` instance

(this as{control: FormControl}).control = this.formDirective.addControl(this);

if (this.control.disabled && this.valueAccessor !.setDisabledState) {

this.valueAccessor !.setDisabledState !(true);

}

this.\_added = true;

}

/\* ... \*/

}

<>

然而，如果你仍然想在formcontrolininstance的名字改变时改变这个FormControl实例，你可以这样做:

{FormArray|FormGroup}.setControl(ctrlName, formControlInstance)

FormGroupName

它接收一个字符串作为参数([formGroupName]="nameOfFormGroupInstance")，根据这个参数，它必须找到正确的FormGroup实例。

它不能用作顶级表单控件容器，必须在现有的FormGroupDirective中注册。

假设你有这样一个形式:

const address = this.fb.group({

street: this.fb.control(''),

});

this.form = this.fb.group({

name: this.fb.control(''),

address,

});

在视图中写入此内容将导致一个错误(无法找到名称:'street'的control):

<form #f="ngForm" [formGroup]="form">

<input formControlName="name" type="text">

<input formControlName="street" type="text">

</form>

解决这个问题的方法是使用FormGroupName指令来创建一个子组，这样视图就会与模型关联起来。

<form #f="ngForm" [formGroup]="form">

<input formControlName="name" type="text">

<ng-container formGroupName="address">

<input formControlName="street" type="text">

</ng-container>

</form>

{{ f.value | json }}

注意:当使用FormControlDirective([formControl]="formControlInstance")时，这是不需要的，因为FormControlDirective不需要找到这个formControl实例，因为它已经通过formControlInstance接收到了一个。

FormArrayName

与FormGroupName相同，只是必须在AbstractControl树中找到一个现有的FormArray实例。

this.fooForm = this.fb.group({

movies: this.fb.array([

this.fb.control('action'),

this.fb.control('horror'),

this.fb.control('mistery'),

]),

});

<>

<form #f="ngForm" [formGroup]="fooForm">

<ng-container formArrayName="movies">

<input

\*ngFor="let \_ of fooForm.controls['movies'].controls; let idx = index;"

[formControlName]="idx"

type="text"

>

</ng-container>

</form>

{{ f.value | json }}

###[Validators](https://indepth.dev/a-thorough-exploration-of-angular-forms/" \l "validators)

###[Exploring built-in ControlValueAccessors](https://indepth.dev/a-thorough-exploration-of-angular-forms/" \l "exploring-built-in-controlvalueaccessors)

###[A better understanding of the AbstractControl tree](https://indepth.dev/a-thorough-exploration-of-angular-forms/" \l "a-better-understanding-of-the-abstractcontrol-tree)

###[What happens with the AbstractControl tree on submit?](https://indepth.dev/a-thorough-exploration-of-angular-forms/" \l "what-happens-with-the-abstractcontrol-tree-on-submit)

###[How are CSS classes added depending on AbstractControl's status ?](https://indepth.dev/a-thorough-exploration-of-angular-forms/" \l "how-are-css-classes-added-depending-on-abstractcontrol-s-status)

###[Conclusion](https://indepth.dev/a-thorough-exploration-of-angular-forms/" \l "conclusion)

https://indepth.dev/posts/1201/presenters-with-angular