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)

验证器允许开发人员将约束放在AbstractControl实例(FormControl、FormArray、FormGroup)上。

在初始化AbstractControl树时设置并运行验证器。如果希望在初始化发生后设置它们，则可以使用AbstractFormControl.setValidators and AbstractFormControl.setAsyncValidators to set them and AbstractFormControl.updateValueAndValidity to run them.

setValidators(newValidator: ValidatorFn|ValidatorFn[]|null): void {

this.validator = coerceToValidator(newValidator);

}

updateValueAndValidity(opts: {onlySelf?: boolean, emitEvent?: boolean} = {}): void {

/\* ... \*/

if (this.enabled) {

this.\_cancelExistingSubscription();

// Run sync validators

// and will invoke `this.validator`

(this as{errors: ValidationErrors | null}).errors = this.\_runValidator();

// If `errors` property is not null -> status = 'INVALID'

(this as{status: string}).status = this.\_calculateStatus();

if (this.status === VALID || this.status === PENDING) {

this.\_runAsyncValidator(opts.emitEvent);

}

}

/\* ... \*/

if (this.\_parent && !opts.onlySelf) {

this.\_parent.updateValueAndValidity(opts);

}

}

从上面的代码片段我们还可以推断出，如果同步验证器返回错误，异步验证器将不会运行。

使用内置验证器

内置的验证器可以作为指令或验证器类的静态成员使用。

例如，email验证器可以像这样直接在视图中使用:

<form>

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

</form>

@Directive({

selector: '[email][formControlName],[email][formControl],[email][ngModel]',

providers: [EMAIL\_VALIDATOR]

})

export class EmailValidator implements Validator {

/\* ... \*/

validate(control: AbstractControl): ValidationErrors|null {

return this.\_enabled ? Validators.email(control) : null;

}

/\* ... \*/

}

而对于反应式表单，你可以这样使用:

this.form = new FormGroup({

name: new FormControl(defaultValue, [Validators.Email])

})

虽然当使用响应式表单时，验证器通常是在组件类中设置的，但你仍然可以在视图中提供验证器;当创建AbstractControl实例时，验证器最终将被合并到setUpControl中

// dir.validator - 通过指令提供的同步验证器(eg: `<input email type="text">`)

// control.validator - 通过“Reactive Forms”提供的同步验证器(eg: new FormControl('', [syncValidators]))

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]);

/\* ... \*/

}

验证器的组成

可以从多个来源提供验证器:来自视图，或来自类，或两者都提供。

所有验证器最终将被合并到一个函数中，当调用时，该函数将按顺序执行所有验证器，并累积它们的结果(返回的错误)。

那些实现验证器接口的将首先被规范化，这意味着将被转换成一个函数，当调用时，将执行验证器。验证方法:

export function normalizeValidator(validator: ValidatorFn | Validator): ValidatorFn {

if ((<Validator>validator).validate) {

return (c: AbstractControl) => (<Validator>validator).validate(c);

} else {

return <ValidatorFn>validator;

}

}

验证器是在setUpControl函数中设置和合并的(如果需要的话):

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]);

/\* ... \*/

}

让我们来探索Validators.compose背后的神奇之处

export class Validators {

static compose(validators: (ValidatorFn|null|undefined)[]|null): ValidatorFn|null {

if (!validators) return null;

const presentValidators: ValidatorFn[] = validators.filter(isPresent) as any;

if (presentValidators.length == 0) return null;

return function(control: AbstractControl) {

return \_mergeErrors(\_executeValidators(control, presentValidators));

};

}

}

function \_executeValidators(control: AbstractControl, validators: ValidatorFn[]): any[] {

return validators.map(v => v(control));

}

// Accumulate errors

function \_mergeErrors(arrayOfErrors: ValidationErrors[]): ValidationErrors|null {

const res: {[key: string]: any} =

arrayOfErrors.reduce((res: ValidationErrors | null, errors: ValidationErrors | null) => {

return errors != null ? {...res !, ...errors} : res !;

}, {});

return Object.keys(res).length === 0 ? null : res;

}

同样的逻辑也适用于Validator.composeAsync，只是验证器的执行方式不同。首先，它将所有异步验证器转换为可观察对象，然后在forkJoin操作符的帮助下执行它们。

export class Validators {

static composeAsync(validators: (AsyncValidatorFn|null)[]): AsyncValidatorFn|null {

if (!validators) return null;

const presentValidators: AsyncValidatorFn[] = validators.filter(isPresent) as any;

if (presentValidators.length == 0) return null;

return function(control: AbstractControl) {

const observables = \_executeAsyncValidators(control, presentValidators).map(toObservable);

return forkJoin(observables).pipe(map(\_mergeErrors));

};

}

}

定制验证器

创建自定义验证器的一个推荐方法是把它用作实现validator接口的指令:

// min-max-validator.directive.ts

@Directive({

selector: '[min-max-validator]',

providers: [

{

provide: NG\_VALIDATORS,

useExisting: forwardRef(() => MinMaxValidator),

multi: true,

}

]

})

export class MinMaxValidator implements Validator {

constructor() { }

validate (f: FormGroup): ValidationErrors | null {

if (f.pristine) { return null; }

const { min, max } = f.controls;

// `min` or `max` is not a number or is empty

if (min.invalid || max.invalid) { return null; }

if (+min.value >= +max.value) {

return { minGreaterMax: 'min cannot be greater than max!' };

}

return null;

}

}

<form #f="ngForm">

<ng-container min-max-validator ngModelGroup="price" #priceGrp="ngModelGroup">

<input type="text" ngModel name="min" pattern="^\d+$" required />

<input type="text" ngModel name="max" pattern="^\d+$" required >

</ng-container>

</form>

动态验证器

验证器接口是这样的:

export interface Validator {

validate(control: AbstractControl): ValidationErrors|null;

registerOnValidatorChange?(fn: () => void): void;

}

我们可以使用registerOnValidatorChange来注册一个回调函数，当验证器的输入发生变化时，就调用这个函数。调用该回调函数将确保AbstractControl实例与更新后的验证器保持一致。

Example: <input [required]="true"> --> <input [required]="false">

@Directive({

selector:

':not([type=checkbox])[required][formControlName],:not([type=checkbox])[required][formControl],:not([type=checkbox])[required][ngModel]',

providers: [REQUIRED\_VALIDATOR],

host: {'[attr.required]': 'required ? "" : null'}

})

export class RequiredValidator implements Validator {

set required(value: boolean|string) {

this.\_required = value != null && value !== false && `${value}` !== 'false';

if (this.\_onChange) this.\_onChange();

}

registerOnValidatorChange(fn: () => void): void { this.\_onChange = fn; }

}

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

/\* ... \*/

// re-run 当验证器绑定更改时进行验证, e.g. minlength=3 -> minlength=4

dir.\_rawValidators.forEach((validator: Validator | ValidatorFn) => {

if ((<Validator>validator).registerOnValidatorChange)

(<Validator>validator).registerOnValidatorChange !(() => control.updateValueAndValidity());

});

dir.\_rawAsyncValidators.forEach((validator: AsyncValidator | AsyncValidatorFn) => {

if ((<Validator>validator).registerOnValidatorChange)

(<Validator>validator).registerOnValidatorChange !(() => control.updateValueAndValidity());

});

/\* ... \*/

}

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

探索内置ControlValueAccessors

下面是Angular提供给我们的内置值访问器:

const BUILTIN\_ACCESSORS = [

CheckboxControlValueAccessor,

RangeValueAccessor,

NumberValueAccessor,

SelectControlValueAccessor,

SelectMultipleControlValueAccessor,

RadioControlValueAccessor,

];

在接下来的小节中，我们将探索一些内置值访问器的内部特性。

SelectValueAccessor

我们可以通过两种方式使用这个值访问器:使用[value]或[ngValue]。

Using <option [value]="primitiveValue">

顾名思义，原语参数只能是原语值。如果你想绑定一个对象，[ngValue]应该是你的选择。

每个<option>都将其值设置为原值。

@Input('value')

set value(value: any) {

this.\_setElementValue(value);

}

\_setElementValue(value: string): void {

this.\_renderer.setProperty(this.\_element.nativeElement, 'value', value);

}

Using <option [ngValue]="primitiveOrNonPrimitiveValue">

与[value]不同，[ngValue]可以同时接受primitive和non-primitive参数。

它会根据提供给[ngValue]的值来设置<option>标签的值。

@Input('ngValue')

set ngValue(value: any) {

if (this.\_select == null) return;

this.\_select.\_optionMap.set(this.id, value);

this.\_setElementValue(\_buildValueString(this.id, value));

this.\_select.writeValue(this.\_select.value);

}

/\* ... \*/

function \_buildValueString(id: string | null, value: any): string {

if (id == null) return `${value}`;

if (value && typeof value === 'object') value = 'Object';

return `${id}: ${value}`.slice(0, 50);

}

我们可以看到，如果我们传递一个对象，其值将类似于'1:object '。如果传递一个原始值，如城市的名称，则将为:0:'NY'

需要注意的是，当你改变<select>的值时(通过使用FormControl.setValue(arg))，如果arg是一个对象，你必须确保它和你传递给<option [ngValue]="arg"></option>的对象是一样的。这是因为，默认情况下，SelectControlValueAccessor.writeValue(obj)，它将使用===来标识所选的选项。

writeValue(value: any): void {

this.value = value;

const id: string|null = this.\_getOptionId(value); // <---- Here!

if (id == null) {

this.\_renderer.setProperty(this.\_elementRef.nativeElement, 'selectedIndex', -1);

}

const valueString = \_buildValueString(id, value);

this.\_renderer.setProperty(this.\_elementRef.nativeElement, 'value', valueString);

}

\_getOptionId(value: any): string|null {

for (const id of Array.from(this.\_optionMap.keys())) {

if (this.\_compareWith(this.\_optionMap.get(id), value)) return id;

}

return null;

}

\_compareWith是这样的(默认情况下):

return a === b || typeof a === 'number' && typeof b === 'number' && isNaN(a) && isNaN(b);

使用自定义的\_compare函数:

compareWith(existing, toCheckAgainst) {

if (!toCheckAgainst) {

return false;

}

return existing.id === toCheckAgainst.id;

}

<!--

1) Try without '[compareWith]="compareWith"'

2) select another option(`B`, or `C`)

3) click `change`

你不应该在' <select> '中看到更新的值

这是因为默认的impl。' compare with '的值将与' === '进行比较

-->

<select

#s="ngModel"

[ngModel]="selectedItem"

[compareWith]="compareWith"

>

<option

\*ngFor="let item of items"

[ngValue]="item"

>

{{item.name}}

</option>

</select>

<br><br>

<button (click)="s.control.setValue({ id: '1', name: 'A' })">change</button>

SelectMultipleValueAccessor

每个选项都被跟踪(添加到内部\_optionMap属性中)，因为

当变更事件发生在<select>上时，值访问器需要向模型提供正确的值(<option>中提供给[value]或[ngValue]的值);这可以通过迭代所选选项(event.target.selectedOptions)并从\_optionMap中检索它们的值来实现。

// \_ - the select element

this.onChange = (\_: any) => {

const selected: Array<any> = [];

if (\_.hasOwnProperty('selectedOptions')) {

const options: HTMLCollection = \_.selectedOptions;

for (let i = 0; i < options.length; i++) {

const opt: any = options.item(i);

const val: any = this.\_getOptionValue(opt.value);

selected.push(val);

}

}

this.value = selected;

fn(selected);

};

当绑定到<select>元素的FormControl值被编程方式更改时(FormControl. setvalue())，它需要以某种方式确定哪些现有选项与新提供的值匹配

writeValue(value: any): void {

this.value = value;

let optionSelectedStateSetter: (opt: ɵNgSelectMultipleOption, o: any) => void;

if (Array.isArray(value)) {

// convert values to ids

const ids = value.map((v) => this.\_getOptionId(v));

optionSelectedStateSetter = (opt, o) => { opt.\_setSelected(ids.indexOf(o.toString()) > -1); };

} else {

optionSelectedStateSetter = (opt, o) => { opt.\_setSelected(false); };

}

this.\_optionMap.forEach(optionSelectedStateSetter);

}

RadioValueAccessor

这个值访问器跟踪单选按钮的帮助下一个内部服务:RadioControlRegistry,该基金持有的数组(NgControl, RadioValueAccessor)对,NgControl提供者标记映射到哪里form-control-based指令之一:NgModel, FormControl FormControlName。

让我们看看它是如何工作的:

@Injectable()

export class RadioControlRegistry {

private \_accessors: any[] = [];

add(control: NgControl, accessor: RadioControlValueAccessor) {

this.\_accessors.push([control, accessor]);

}

remove(accessor: RadioControlValueAccessor) {

for (let i = this.\_accessors.length - 1; i >= 0; --i) {

if (this.\_accessors[i][1] === accessor) {

this.\_accessors.splice(i, 1);

return;

}

}

}

select(accessor: RadioControlValueAccessor) {

this.\_accessors.forEach((c) => {

if (this.\_isSameGroup(c, accessor) && c[1] !== accessor) {

c[1].fireUncheck(accessor.value);

}

});

}

private \_isSameGroup(

controlPair: [NgControl, RadioControlValueAccessor],

accessor: RadioControlValueAccessor): boolean {

if (!controlPair[0].control) return false;

return controlPair[0].\_parent === accessor.\_control.\_parent &&

controlPair[1].name === accessor.name;

}

}

RadioControlRegistry.\_isSameGroup。\_isSameGroup方法。

让我们用一个更简单的例子来缩小范围:

<form>

<input ngModel name="option" value="value1" type="radio"> <!-- #1 NgModel.\_parent = the top-level `FormGroup` which results from `<form>` -->

<ng-container ngModelGroup="foo">

<input ngModel name="option" value="value1" type="radio"> <!-- #2 NgModel.\_parent = the sub-group `FormGroup` which results from `ngModelGroup` -->

</ng-container>

</form>

注意，这两个单选按钮有相同的值!

RadioControlRegistry.\_accessors数组看起来像这样:

[

NgControl(-> NgModel) /\* #1 \*/, RadioControlValueAccessor,

NgControl(-> NgModel) /\* #2 \*/, RadioControlValueAccessor,

]

当用户点击第一个单选按钮时，RadioControlRegistry的这个方法将被执行:

select(accessor: RadioControlValueAccessor) {

this.\_accessors.forEach((c) => {

if (this.\_isSameGroup(c, accessor) && c[1] !== accessor) {

c[1].fireUncheck(accessor.value);

}

});

}

这里的accessor是属于第一个单选按钮的RadioControlValueAccessor。

下面又是\_isSameGroup方法:

private \_isSameGroup(

controlPair: [NgControl, RadioControlValueAccessor],

accessor: RadioControlValueAccessor): boolean {

if (!controlPair[0].control) return false;

return controlPair[0].\_parent === accessor.\_control.\_parent &&

controlPair[1].name === accessor.name;

}

controlPair[0].\_parent = = = accessor.\_control.\_parent用于防止第一个单选按钮影响第二个单选按钮。

在下面的例子中，如果我们点击第二个按钮，第一个按钮将被标记为选中。

<form>

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

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

</form>

这是因为在N个具有相同名称和值属性的单选按钮中，只有一个可以标记为checked。在这种情况下，它是最后一个满足这些条件:

this.\_isSameGroup(c, accessor) && c[1] !== accessor

这里的accessor是选中单选按钮的RadioControlValueAccessor。

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

更好地理解AbstractControl树

在本文中，您可能已经注意到了短语AbstractControl树。请记住AbstractControl是一个抽象类，它的具体实现是FormControl、FormGroup和FormArray。

为了让事情更直观，我们可以将它们的连接形象化为树形结构。

例如,这

new FormGroup({

name: new FormControl(''),

address: new FormGroup({

city: new FormControl(''),

street: new FormControl(''),

}),

});

如下图所示:

FG

/ \

FC FG

/ \

FC FC

使用上面的图，我们将了解常见的AbstractControl操作(如reset()、submit()、markAsDirty())如何改变树。

在继续之前，我建议阅读基础实体。

\_pendingDirty, \_pendingValue, \_pendingChange

AbstractControl类的这些私有属性是您可能不必关心的细节。然而，它们在AbstractControl树的有效性方面扮演着重要的角色。

这些属性会在FormControl的上下文中遇到，因为它们的值依赖于从视图(从ControlValueAccessor)发送的值。

\_pendingChange

这个属性指示用户是否更改了FormControl的值。

假设你有一个<input ngModel name="name" type="text">，用户类型在其中。一旦发生这种情况，ControlValueAccessor的onChange函数就会被调用。分配给onChange的函数如下所示:

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);

});

}

control.\_pendingChange = true标志用户明显地与<input>交互。

为什么这个有用呢?这是因为您可以设置AbstractControl在其上更新自身的事件(它默认为更改)。

你可以通过\_updateOn属性来确定更新策略:'change'|'blur'|'submit';

考虑到这一点，如果FormControl的更新策略设置为blur，并且blur事件发生在视图中，而用户没有在<input>中输入任何内容，会发生什么情况呢?在这种情况下，\_pendingChange阻止树被冗余地遍历。

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

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

/\* ... \*/

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

/\* ... \*/

});

}

让用户在<input>控件中输入任何内容。\_pendingChange会被设为true。因此，当模糊事件发生时，表单控件及其祖先将被更新。

\_pendingDirty

如果用户在UI中改变了FormControl的值，那么它就会被认为是脏的。

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

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

/\* ... \*/

control.\_pendingChange = true;

control.\_pendingDirty = true;

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

});

}

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

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

/\* ... \*/

}

在dir.valueAccessor !.registerOnChange(cb)中注册的回调函数将在UI更改值时由ControlValueAccessor(驻留在视图层中)调用。

AbstractControl.markedAsDirty的实现是这样的:

markAsDirty(opts: {onlySelf?: boolean} = {}): void {

(this as{pristine: boolean}).pristine = false;

if (this.\_parent && !opts.onlySelf) {

this.\_parent.markAsDirty(opts);

}

}

因此，如果一个FormControl被标记为dirty(由于UI更改)，它的祖先将被相应地更新(在这种情况下，它们将被标记为dirty)。

FG (3)

/ \

FC FG (2)

/ \

FC FC (1)

(1).parent = (2)

(2).parent = (3)

(3).parent = null(root)

假设(1)一个FormControl绑定到<input>，并且用户刚刚输入了它,上面的方法将从updateControl函数中调用:control. markasdirty()，其中control为(1)。这将向上传播到根，顺序是:(1)->(2)->(3)。因此，整个树将被标记为dirty!

还有一个选项可以将(1)单独标记为dirty: (1). markedasdirty ({onlySelf: true})。

现在您想知道，如果一旦用户输入了一些东西，控件的脏度就会改变，那么\_pendingDirty的需求是什么?这是因为默认策略默认为更改，但也可以更改为模糊或提交。

例如，下面是模糊事件在视图中发生的情况:

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

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

/\* ... \*/

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

/\* ... \*/

});

}

\_pendingValue

您可以将该属性看作是FormControl的最新值。

它的值在ControlValueAccessor时设置。调用onChange，其中ControlValueAccessor.onChange这样做:

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

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

control.\_pendingValue = newValue;

/\* ... \*/

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

});

}

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

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

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

dir.viewToModelUpdate(control.\_pendingValue);

control.\_pendingChange = false;

}

但是，\_pendingValue和value之间的区别是什么?\_pendingValue是最近的值，而value是AbstractControl树可见的值。这个值并不总是等于\_pendingValue，因为FormControl的更新策略可能与change不同。当然，视图层可以保存最近的值，但这并不意味着模型层可以。

例如，如果FormControl的更新策略设置为submit，那么模型的值(FormControl.value)不会等于\_pendingValue(它是反映视图的值)，直到submit事件发生。

AbstractControl.setValue() and AbstractControl.patchValue()

// {FormGroup|FormArray}.setValue

setValue(value: {[key: string]: any}, options: {onlySelf?: boolean, emitEvent?: boolean} = {}):

void {

this.\_checkAllValuesPresent(value);

Object.keys(value).forEach(name => {

this.\_throwIfControlMissing(name);

this.controls[name].setValue(value[name], {onlySelf: true, emitEvent: options.emitEvent});

});

this.updateValueAndValidity(options);

}

// {FormGroup|FormArray}.patchValue

patchValue(value: {[key: string]: any}, options: {onlySelf?: boolean, emitEvent?: boolean} = {}):

void {

Object.keys(value).forEach(name => {

if (this.controls[name]) {

this.controls[name].patchValue(value[name], {onlySelf: true, emitEvent: options.emitEvent});

}

});

this.updateValueAndValidity(options);

}

AbstractControl.setValue将要求您为所有现有控件提供一个值，而AbstractControl.patchValue允许你为任何现有的控件提供值。

{FormGroup | FormArray}.setValue将首先检查你是否提供了一个包含所有现有控件的对象，然后它将检查你是否提供了任何冗余的控件(不在现有控件中)

当调用setValue/patchValue时，如果AbstractControl是FormControl，它将首先更新FormControl实例，然后是它的祖先。否则，它将首先更新其后代，然后更新其祖先。

可以通过第二个参数{onlySelf: true}来避免更新父类。

这是第一个例子:

const fg = new FormGroup({

name: new FormControl(''),

address: new FormGroup({

city: new FormControl(''),

street: new FormControl(''),

}),

});

FG (4)

/ \

FC FG (3) - address

/ \

FC FC

(1) (2)

后执行

fg.get('address').setValue({ city: 'city', street: 'street' })

它首先更新(1)和(2)，然后更新其容器(3)的值和有效性，最后更新其祖先。

patchValue example

const c = new FormControl('');

const c2 = new FormControl('');

const a = new FormArray([c, c2]);

a.patchValue(['andrei']);

console.log(a.value) // ['andrei', '']

setValue example

const c1 = new FormControl('c1');

const c2 = new FormControl('c2');

const a = new FormArray([c1, c2]);

a.setValue(['c1-updated', 'c2-updated', 'c3']); // Error: Cannot find form control at index 2

a.setValue(['c1-updated']); // Error: Must supply a value for form control at index: 1

a.setValue(['c1-updated', 'c2-updated']);

console.log(a.value); // ["c1-updated", "c2-updated"]

###[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)

在提交时AbstractControl树会发生什么?  
注意:只有FormGroupDirective和NgForm才能调用onSubmit。

onSubmit($event) {

(this as{submitted: boolean}).submitted = true;

syncPendingControls(this.form, this.directives);

this.ngSubmit.emit($event);

return false;

}

一些AbstractControl实例可能以不同的方式设置了updateOn选项。因此，如果一个表单控件将updateOn选项设置为提交，这意味着它的值和UI状态(脏的，未动的等等)只会在提交事件发生时更新。这就是syncPendingControls()所做的。

// FormControl

\_syncPendingControls(): boolean {

if (this.updateOn === 'submit') {

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

if (this.\_pendingTouched) this.markAsTouched();

if (this.\_pendingChange) {

this.setValue(this.\_pendingValue, {onlySelf: true, emitModelToViewChange: false});

return true;

}

}

return false;

}

// FormArray - FormGroup以非常相似的方式工作

\_syncPendingControls(): boolean {

let subtreeUpdated = this.controls.reduce((updated: boolean, child: AbstractControl) => {

return child.\_syncPendingControls() ? true : updated;

}, false);

if (subtreeUpdated) this.updateValueAndValidity({onlySelf: true});

return subtreeUpdated;

}

考虑一下这个例

this.form = this.fb.group({ name: this.fb.control('', { updateOn: 'submit' }) });

this.form.valueChanges.subscribe(console.warn);

当看到这样的景色时

<form [formGroup]="form" (ngSubmit)="onSubmit()">

<input [formControl]="form.get('name')" type="text">

<button type="submit">Submit</button>

</form>

每次提交事件发生时，您都会得到相同的值，而对于这个视图

<form [formGroup]="form" (ngSubmit)="onSubmit()">

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

<br><br>

<button type="submit">Submit</button>

</form>

您只在提交事件发生时获得一次值

这是因为FormControlName指令在FormGroupDirective中的工作方式。FormGroupDirective会在directives属性的帮助下跟踪FormControlName指令。当submit事件发生时，每个FormControlName都将其绑定的FormControl的\_pendingChange属性设置为false。

directives.forEach(dir => {

const control = dir.control as FormControl;

if (control.updateOn === 'submit' && control.\_pendingChange) {

/\* ... \*/

control.\_pendingChange = false;

}

});

每次UI中发生更改事件时，FormControl.\_pendingChange都被设置为true。

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);

});

}

你可以在这里找到更多关于\_pendingChange的信息。

从树中检索AbstractControls

const fg = new FormGroup({

name: new FormControl(''),

address: new FormGroup({

city: new FormControl(''),

street: new FormControl(''),

}),

});

有两种方法可以检索AbstractControl。

如果您想要检索的AbstractControl是表单控件容器(在本例中为fg)的直接后代，您可以这样做:

fg.controls[nameOfCtrl];

// In our example

fg.controls['name']

fg.controls['address']

然而，如果AbstractControl有几层深度，你可能会觉得写这样的东西很烦人:

fg.controls['address'].controls['city']

您可以使用AbstractControl.get()方法

fg.get('address.city')

// Or

fg.get(['address', 'street'])

AbstractControl.get()将在内部调用函数\_find，该函数将根据提供的路径向下遍历树。

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);

}

您可能已经注意到，如果fg是一个FormArray实例，则可以通过指定索引(而不是属性名)来检索其后代(就像使用FormGroup那样)

fg.get('1.city');

// Or

fg.get(['1', 'city']);

AbstractControl.updateValueAndValidity()

updateValueAndValidity(opts: {onlySelf?: boolean, emitEvent?: boolean} = {}): void {

this.\_setInitialStatus();

this.\_updateValue();

if (this.enabled) {

this.\_cancelExistingSubscription();

(this as{errors: ValidationErrors | null}).errors = this.\_runValidator(); // Sync validators

(this as{status: string}).status = this.\_calculateStatus(); // VALID | INVALID | PENDING | DISABLED

if (this.status === VALID || this.status === PENDING) {

this.\_runAsyncValidator(opts.emitEvent);

}

}

if (opts.emitEvent !== false) {

(this.valueChanges as EventEmitter<any>).emit(this.value);

(this.statusChanges as EventEmitter<string>).emit(this.status);

}

if (this.\_parent && !opts.onlySelf) {

this.\_parent.updateValueAndValidity(opts);

}

}

如上所示，该方法负责多个事情:

1 更新当前AbstractControl的值

2 运行验证器(同步和异步)

3 根据验证器返回的内容计算状态

4 向订阅者发送新值和新状态(除非emitEvent = false)

5 父类重复1-4(除非onlySelf = true)

const fg = new FormGroup({

name: new FormControl(''),

address: new FormGroup({

city: new FormControl(''),

street: new FormControl(''),

}),

});

FG (3)

/ \

FC FG (2)

/ \

FC FC (1)

(1) - fg.get('address.street')

(2) - fg.get('address')

(3) - fg

一旦你执行(1). setvalue ('new value')， (1). updatevalueandvalidity()将被调用。

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);

}

1. 被更新后，(2)将被更新，以此类推。直到到达根。

Disabling/enabling AbstractControls

可以从模型中禁用/启用AbstractControl。在视图中，可以通过ControlValueAccessor.setDisabledState查看更改:

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

/\* ... \*/

if (dir.valueAccessor !.setDisabledState) {

control.registerOnDisabledChange(

(isDisabled: boolean) => { dir.valueAccessor !.setDisabledState !(isDisabled); });

}

/\* ... \*/

}

当禁用AbstractControl实例时，您可以使用this.control选择不更新其祖先。禁用({onlySelf: true})。这可能是当一个FormControl可能是一个FormGroup的一部分，因为这个控件无效，整个FormGroup被标记为无效的情况。

const fg = this.fb.group({

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

age: '',

city: this.fb.control('', Validators.required)

});

fg.controls['name'].disable();

fg.controls['city'].disable({ onlySelf: true });

console.log(fg.valid) // false

如果我们省略了{onlySelf: true}，那么整个形式群(fg)将是有效的(fg。有效的= = = true)。

disable(opts: {onlySelf?: boolean, emitEvent?: boolean} = {}): void {

// 如果parent被人为地标记为dirty，则不需要重新计算

//父母的肮脏基于孩子.

const skipPristineCheck = this.\_parentMarkedDirty(opts.onlySelf);

(this as{status: string}).status = DISABLED;

(this as{errors: ValidationErrors | null}).errors = null;

this.\_forEachChild(

(control: AbstractControl) => { control.disable({...opts, onlySelf: true}); });

this.\_updateValue();

if (opts.emitEvent !== false) {

(this.valueChanges as EventEmitter<any>).emit(this.value);

(this.statusChanges as EventEmitter<string>).emit(this.status);

}

// Will update the value, validity, dirtiness, and touch status

this.\_updateAncestors({...opts, skipPristineCheck});

this.\_onDisabledChange.forEach((changeFn) => changeFn(true));

}

private \_updateAncestors(

opts: {onlySelf?: boolean, emitEvent?: boolean, skipPristineCheck?: boolean}) {

if (this.\_parent && !opts.onlySelf) {

this.\_parent.updateValueAndValidity(opts);

if (!opts.skipPristineCheck) {

this.\_parent.\_updatePristine();

}

this.\_parent.\_updateTouched();

}

}

当AbstractControl被禁用时，它的验证器将不会运行，它的错误将被标记为null，它的子控件也将被禁用。

如果父类被人为地标记为dirty(dirty不是由其子类来决定的:手动地做{FormGroup|FormArray}.markAsDirty)，则不需要基于子类重新计算父类的dirty，因为它们对父类没有任何影响:

this.form = this.fb.group({

name: this.fb.control({ value: 'andrei', disabled: false }),

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

});

const nameCtrl = this.form.controls['name'];

// 现在，它的祖先也将被标记为肮脏的

//在这种情况下，只有一个' FormGroup ' (this.form)

nameCtrl.markAsDirty();

nameCtrl.disable();

// 现在,`this.form将被标记为'pristine'，因为

//影响父母肮脏的孩子是残疾的

同样，如果一个form-control-container(FormGroup或FormArray)被禁用，它的值将从它的所有后代收集的值，不管它们的禁用值是什么:

const g = new FormGroup({

name: new FormControl('name'),

address: new FormGroup({

city: new FormControl('city'),

street: new FormControl('street'),

}),

});

g.get('address.city').disable();

g.controls['name'].disable();

console.log(g.value);

/\*

{

"address": {

"street": "street"

}

}

\*/

g.disable();

console.log(g.value)

/\*

{

"name": "name",

"address": {

"city": "city",

"address": "address"

}

}

这背后的原因是AbstractControl.disable()的工作方式。从当前AbstractControl开始，它将首先禁用其所有后代，然后收集它们的值。例如，下面是一个FormArray如何从它的后代中积累值:

\_updateValue(): void {

(this as{value: any}).value =

this.controls.filter((control) => control.enabled || this.disabled)

.map((control) => control.value);

}

The control.enabled || this.disabled 表达式允许我们获取值，即使子控件可能被禁用。

但是，如果容器没有被禁用，而子控件被禁用，则不考虑它的值。

如果你还想获取表单值，包括禁用的控件，你可以使用{FormGroup|FormArray}.getRawValue():

// FormArray.getRawValue()

getRawValue(): any[] {

return this.controls.map((control: AbstractControl) => {

return control instanceof FormControl ? control.value : (<any>control).getRawValue();

});

}

###[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)

CSS类是如何根据AbstractControl的状态添加的?  
CSS类(ng-valid、ng-pristine、ng-touched等)是在NgControlStatus指令的帮助下添加的，当使用ngModel、formControl、formControlName时，它会自动绑定到表单控件元素上。

此外，NgControlStatusGroup被添加到表单组中(<form>， formGroupName, formGroup, ngModelGroup, formArrayName)。

当发生变更检测时，NgControlStatus和NgControlStatusGroup都会更新。

export class AbstractControlStatus {

private \_cd: AbstractControlDirective;

constructor(cd: AbstractControlDirective) { this.\_cd = cd; }

get ngClassUntouched(): boolean { return this.\_cd.control ? this.\_cd.control.untouched : false; }

get ngClassTouched(): boolean { return this.\_cd.control ? this.\_cd.control.touched : false; }

get ngClassPristine(): boolean { return this.\_cd.control ? this.\_cd.control.pristine : false; }

get ngClassDirty(): boolean { return this.\_cd.control ? this.\_cd.control.dirty : false; }

get ngClassValid(): boolean { return this.\_cd.control ? this.\_cd.control.valid : false; }

get ngClassInvalid(): boolean { return this.\_cd.control ? this.\_cd.control.invalid : false; }

get ngClassPending(): boolean { return this.\_cd.control ? this.\_cd.control.pending : false; }

}

export const ngControlStatusHost = {

'[class.ng-untouched]': 'ngClassUntouched',

'[class.ng-touched]': 'ngClassTouched',

'[class.ng-pristine]': 'ngClassPristine',

'[class.ng-dirty]': 'ngClassDirty',

'[class.ng-valid]': 'ngClassValid',

'[class.ng-invalid]': 'ngClassInvalid',

'[class.ng-pending]': 'ngClassPending',

};

@Directive({selector: '[formControlName],[ngModel],[formControl]', host: ngControlStatusHost})

export class NgControlStatus extends AbstractControlStatus {

constructor(@Self() cd: NgControl) { super(cd); }

}

考虑到这一点，你可以通过使用自定义指令来添加你的自定义css类，这取决于表单控件(或表单控件容器)的有效性或用户交互状态

constructor (private ngControlStatus: NgControlStatus) { }

@HostBinding('[class.card\_\_price--incorrect]') this.ngControlStatus.ngClassInvalid();

注意:为了让它生效，你的元素(或组件)除了上面的指令外，还必须包含这些基于表单控件的指令:[formControlName]，[ngModel]，[formControl]

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

我希望本文已经澄清了一些概念，并强调了这个包的强大功能。  
感谢你的阅读!