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:

**1 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 { }

**2 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 }}

{{ 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设置它们和AbstractFormControl.updateValueAndValidity运行它们。

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 '或' max '不是数字或为空

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

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

return { minGreaterMax: 'Min不能大于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!

Angular Forms:响应式设计模式目录

https://indepth.dev/posts/1428/angular-forms-reactive-design-patterns

在许多Angular企业应用中，最重要的Angular模块是ReactiveFormsModule。Angular表单允许用户创建、更新和搜索数据。它们还通过验证和自动完成等功能增强了用户体验。

应用程序表单和GUI通常是最能从响应式编程范式中获益的软件组件之一。很多Angular模块都是基于RxJS的(比如HttpClientModule和RouterModule)。但是，这个反应性API的强大功能在很多Angular应用中并没有得到充分利用。

在这篇文章中，我们将介绍一组用于构建Angular表单的设计模式，这些模式是我在过去几年里使用过并推广过的。这些模式基于两大支柱:职责分离和响应式编程，以解决丰富和复杂的Angular表单的复杂性。

在跳转到我们的第一个设计模式之前，下一节将概述不同的模式及其目标。

**设计模式**

设计模式是针对给定上下文中常见问题的通用、可重用的解决方案。这些模式将在Angular中呈现，但它们也适用于其他框架。这些模式背后的主要目标是:

提高可维护性

减少bug的数量

我们的设计模式是我用来创建复杂Angular表单的模式语言的核心。下图显示了主要模式及其关系:

该语言的中心部分是**表单模型**。让我们从展示开始。

**Form Model**

Angular表单中最重要的组件是FormGroup和FormControl。这两个组件是无类型的，并提供了一个低级的通用API。在复杂的应用中，FormGroup和FormControl通常不是Angular组件中很好的抽象。相反，我们需要一个特定于网页组件的高级抽象。

为了解决应用的复杂性，我们需要在Angular API之上引入一个抽象层。我们称这一层为Form Model模式，它定义了FormGroup实例的包装。它在Angular通用表单API之上提供了一个特殊用途的API。所以，我们不用Angular模型术语，而是用应用术语。  
这里有一个简单的例子:

class PersonCreationForm {

readonly initialValue;

constructor(private formGroup: FormGroup) {

this.initialValue = formGroup.value;

}

get asFormGroup() {

return this.formGroup;

}

isValid(): Observable<boolean> {

return this.formGroup.statusChanges.pipe(

map(() => this.formGroup.valid),

startWith(false)

);

}

ageIsGreaterThan(min: number): Observable<boolean> {

return this.formGroup.valueChanges.pipe(

map(value => value.age),

distinctUntilChanged(),

map(it => it > min),

startWith(false)

);

}

}

PersonCreationForm类具有以下特征:

提供一个高级API: ageIsGreaterThan()是对底层管道细节的抽象，用于确定年龄是否大于传递的参数。

公开响应式API:除了asFormGroup()属性，所有数据都可以通过observable访问，

通过Observable来公开Angular的非响应性属性:isValid()返回一个在FormGroup时发出值的Observable。有效的属性更改,

Angular不允许访问表单的初始值，

并使FormGroup可访问。

注意:表单模型是一个有漏洞的抽象，因为视图需要访问包装好的FormGroup。

表单模型可以像这样在组件中使用:

@Component({

selector: "person-creation-form",

template: `

...

<form [formGroup]="form.asFormGroup">

...

</form>

...

`

})

export class PersonCreationFormComponent {

form: PersonCreationForm;

ageIsGreaterThanTen: Observable<boolean>;

formIsValid: Observable<boolean>;

constructor(formBuilder: FormBuilder) {

this.form = createFormModelUsing(formBuilder);

this.ageIsGreaterThanTen = this.form.ageIsGreaterThan(10);

this.formIsValid = this.form.isValid();

}

}

function createFormModelUsing(formBuilder: FormBuilder): PersonCreateForm {

const formGroup = formBuilder.group({

name: "",

age: ""

});

return new PersonCreateForm(formGroup);

}

PersonCreationFormComponent的独特作用是定义视图并将其连接到表单模型。  
源代码可以在这里找到。

https://stackblitz.com/edit/angular-form-reactive-design-patterns-v1

现在让我们看看应该如何创建表单模型。

**Form工厂**将复杂的Angular表单添加到网页的第一步是创建一个FormGroup对象，并将其绑定到视图模板。创建表单组主要定义表单字段、初始值和验证器。这种创建逻辑通常是泄漏的，并与Angular组件中的其他类型的逻辑混合在一起。

创建FormGroup或更好的表单模型应该与其他逻辑分开。为了隔离它，我们应该使用Form Factory模式。显示表单的Angular组件应该不知道这些细节。

让我们用一个具体的例子来说明这个模式:

@Injectable()

class PersonCreationFormFactory {

constructor(private formBuilder: FormBuilder) {}

create(): PersonCreationForm {

const formGroup = this.createFormGroup();

return new PersonCreationForm(formGroup);

}

private createFormGroup() {

return this.formBuilder.group({

name: [""],

age: [""]

});

}

}

PersonCreationFormFactory是一个表单工厂的简单例子。它只有FormBuilder作为依赖项。在更复杂的场景中，我们可以设置更多的依赖关系，例如初始值。

工厂可以直接注入到组件类中，也可以使用如下的工厂提供程序定义模式:

@Component({

selector: "person-creation-form",

template: `...`,

providers: [

PersonCreationFormFactory,

{

provide: PersonCreationForm,

useFactory: (factory: PersonCreationFormFactory) => factory.create(),

deps: [PersonCreationFormFactory]

}

]

})

export class PersonCreationFormComponent {

constructor(public form: PersonCreationForm) {

}

}

使用useFactory有点冗长，但是当使用本文描述的模式时，这样做是值得的。表单模型很容易注入。  
总之，表格工厂的职责是:  
创建表单组，定义验证规则，定义初始值  
源代码可以在这里找到。

https://stackblitz.com/edit/angular-form-reactive-design-patterns-v2

在第一个模式中，我们创建了表单模型，以便用户输入和选择值。下一个模式将提供供应用程序用户选择的数据。

**表单数据提供程序**Form Data Provider表单字段可以为用户提供高级帮助。最常见的帮助是自动完成。这种逻辑通常只在视图中使用。但是，它是由组件类准备的。

表单数据提供程序(Form Data Provider)是一种模式，它促进了服务中这种逻辑的分离，而服务只被组件的HTML模板使用。该服务的唯一作用是提供字段所需的动态数据。  
让我们看一个例子:

@Injectable()

class PersonCreationFormDataProvider {

constructor(private httpClient: HttpClient) {}

searchCountry = (termChanged: Observable<string>): Observable<string[]> =>

termChanged.pipe(

debounceTime(200),

distinctUntilChanged(),

switchMap(term => term.length < 3 ? of([]) : this.findCountryBy(term)),

map(values => values.map(country => country.name))

);

private findCountryBy(term: string) {

return this.httpClient.get<any[]>(`https://restcountries.eu/rest/v2/name/${term}?fields=name`);

}

}

searchCountry方法接收一个发出搜索词的可观察对象，并返回一个包含搜索结果的可观察对象。我们使用restcountries。eu API做搜索。

该服务返回的数据可以依赖于字段搜索词，也可以依赖于其他字段值。我们可以添加一个取决于所选国家的city字段。在这种情况下，searchCity方法将依赖于用户搜索词和表单模型。让我们将示例保持得更简单，以便于掌握，但在现实生活中，表单数据提供程序要复杂得多。

要使用PersonCreationFormDataProvider，我们需要将它声明为一个组件提供程序，将它注入到PersonCreationFormComponent中，并只在模板中使用它。

@Component({

selector: "person-creation-form",

template: `

<form [formGroup]="form.asFormGroup" class="form-horizontal">

...

<div class="form-group">

<label for="country">Country</label>

<input id="country" type="text" formControlName="country"

class="form-control" [ngbTypeahead]="formDataProvider.searchCountry"/>

</div>

...

</form>

`,

providers: [

PersonCreationFormFactory,

{

provide: PersonCreationForm,

useFactory: (factory: PersonCreationFormFactory) => factory.create(),

deps: [PersonCreationFormFactory]

},

**PersonCreationFormDataProvider**

]

})

export class PersonCreationFormComponent {

ageIsGreaterThanTen: Observable<boolean>;

formIsValid: Observable<boolean>;

constructor(

public form: PersonCreationForm,

public **formDataProvider: PersonCreationFormDataProvider**

) {

this.ageIsGreaterThanTen = this.form.ageIsGreaterThan(10);

this.formIsValid = this.form.isValid();

}

}

在具有许多自动完成字段和复杂逻辑的大型表单中，我们应该使用多个数据提供者服务来保持代码易于掌握。我们甚至可以为每个字段提供专门的数据提供程序。但在不那么复杂的情况下，我们可以为每个表单创建单个服务。

源代码可以在这里找到。

https://stackblitz.com/edit/angular-form-reactive-design-patterns-v3

使用前面的三个模式，用户可以输入和选择数据。下一步是添加表单动作。

**Form Actions**

每个表单至少有一个操作可以在填充其字段后执行。形式作用通常与创造和计算逻辑纠缠在一起。对于一个有一些逻辑的表单，这不是一个真正的问题。但是对于复杂的表单，将这种逻辑与其他逻辑混合会使代码变得隐式，很难很快理解。

增强可维护性的一个很好的解决方案是使用Form Actions模式，该模式将操作逻辑放在一个专用的类中。

@Injectable()

class PersonCreationFormActions {

validateButtonClicked = new Subject<void>();

resetButtonClicked = new Subject<void>();

constructor(private form: PersonCreateForm) {

this.handleValidateButtonClick();

this.handleResetButtonClick();

}

private handleValidateButtonClick() {

this.validateButtonClicked

.subscribe(() => alert('The form is validated!'))

}

private handleResetButtonClick() {

this.resetButtonClicked

.subscribe(() => this.form.reset())

}

}

PersonCreateFormActions示例使用一个简单的实现来说明这个概念。表单提出了两个操作:验证和重置。

@Component({

selector: "person-creation-form",

template: `

<form [formGroup]="form.asFormGroup" class="form-horizontal">

<button class="btn btn-primary" [clickEvent]="formActions.validateButtonClicked">

Validate</button>

<button class="btn btn-secondary" [clickEvent]="formActions.resetButtonClicked">

reset</button>

</form>

`,

providers: [

PersonCreationFormFactory,

{

provide: PersonCreationForm,

useFactory: (factory: PersonCreationFormFactory) => factory.create(),

deps: [PersonCreationFormFactory]

},

PersonCreationFormDataProvider,

**PersonCreationFormActions**

]

})

export class PersonCreationFormComponent {

ageIsGreaterThanTen: Observable<boolean>;

formIsValid: Observable<boolean>;

constructor(

public form: PersonCreationForm,

public formDataProvider: PersonCreationFormDataProvider,

public **formActions: PersonCreationFormActions**

) {

this.ageIsGreaterThanTen = this.form.ageIsGreaterThan(10);

this.formIsValid = this.form.isValid();

}

}

我们不再使用click事件。相反，我们使用自定义指令来定义单击事件监听器。因此，视图直接通知我们的服务主题。

@Directive({

selector: '[clickEvent]'

})

export class ClickEventDirective {

@Input() clickEvent: Subject<void>;

@HostListener('click') onClick() {

this.clickEvent.next();

}

}

源代码可以在这里找到。

https://stackblitz.com/edit/angular-form-reactive-design-patterns-v4

现在，我们有了创建允许创建或更新业务对象的表单所需的模式。在下一个模式中，我们将展示一个用于搜索表单的模式。

**Search Form**

在许多情况下，web应用程序用户希望在浏览器书签中共享搜索结果或标记搜索请求/条件。在SPA中，最佳实践是将搜索条件放在URL查询参数中。本机浏览器行为允许用户共享URL并在搜索历史中导航。  
一个简单的实现是:

export interface ParamsConverter<T> {

fromUrl(Params): T;

toUrl(T): Params;

}

export const URL\_STORE\_CONVERTER =

new InjectionToken<ParamsConverter<any>>('ParamsConverter');

@Injectable()

export class UrlStore<T> {

changed: Observable<T>;

refreshed = new Subject<T>();

changedOrRefreshed: Observable<T>;

constructor(

@Inject(URL\_STORE\_CONVERTER) private converter: ParamsConverter<T>,

private router: Router,

private route: ActivatedRoute,

) {

this.changed = this.route.queryParams.pipe(map(converter.fromUrl));

this.changedOrRefreshed = merge(this.changed, this.refreshed);

}

setSource(paramsChanges: Observable<T>) {

paramsChanges.subscribe(params => {

const urlParams = this.converter.toUrl(params);

const extras = {

relativeTo: this.route,

queryParams: removeEmptyAtrributes(urlParams),

} as NavigationExtras;

this.router.navigate(['.'], extras).then(result => {

const urlIsTheSame = result === null;

if (urlIsTheSame) {

this.refreshed.next(params);

}

});

});

}

}

UrlStore是一个泛型类，可以通过实现ParamsConverter接口进行扩展。它的API主要由两个部分组成:

首先，它允许使用setSource方法更改URL内容。可以用Subject作为类属性替换此方法。  
其次，它公开三个属性来检测URL中的不同类型的更改:changed、refreshed和changedOrRefreshed。

在搜索表单中，表单动作类中使用URL存储来更新URL查询参数。

@Injectable()

class PersonCreationFormActions {

searchButtonClicked = new Subject<void>();

constructor(

private form: PersonCreateForm,

private urlStore: UrlStore<PersonSearchCriteria>

) {

this.handleSearchButtonClick();

}

private handleSearchButtonClick() {

const searchAction = this.searchButtonClicked.pipe(

map(it => this.form.asFormGroup.value)

);

this.urlStore.setSource(searchAction);

}

}

源代码可以在这里找到。

<https://stackblitz.com/edit/angular-form-reactive-design-patterns-v5?file=src%2Fapp%2Fperson-creation-form%2Fperson-creation-form.component.ts>

这是本文的最后一个模式。现在让我们花点时间来回答几个在实现上面描述的模式时经常被问到的问题。

**为什么我们不用呢 Presentational and Container patterns?**

因为它是一个反模式。它是在一个有Angular所没有的限制的框架中提升的。一个Angular组件应该只有与视图和视图模型模式相关的职责。一年多以前我就不再考虑这个模式了，现在我的代码好多了。

**为什么我们只使用组件提供者?**

我们的实现主要基于组件提供程序。这种类型的提供者的主要优点是具有与其组件相同的生命周期。

嘿加拉!我想理解以下行{提供:

PersonCreationForm, useFactory:

(factory: PersonCreationFormFactory) => factory.create()，

**deps: [PersonCreationFormFactory]**

}，

为什么deps: [PersonCreationFormFactory]，这一行在获取依赖中的角色是什么。

只定义useFactory: (factory: PersonCreationFormFactory) => factory.create()不足以让Angular决定要注入什么依赖。在运行时，参数工厂没有PersonCreationFormFactory类型。该类型只在编译时存在。所以，Angular添加了deps数组来让依赖注入成为可能。我希望这就是你想要的回应。

你好默罕默德,  
这篇文章很有意思。我习惯于在单个组件中编写响应式表单，所以看到这些模式应用于Angular会让我耳目一新。我有两个问题:  
这些模式可以被不同的组件重用吗?例如:PersonForm用于创建人员和编辑人员。  
你能详细解释一下为什么表示/容器是一个反模式吗?说实话，这是我第一次听说。

嗨,奥古斯汀•,  
这些模式可以被不同的组件重用吗?例如:PersonForm用于创建人员和编辑人员。如果创建和编辑表单具有完全相同的结构、验证和行为，我们可以为两者重用相同的表单模型。但是，从我的经验来看，这两种形式通常是不同的，即使它们一开始看起来很相似。

你能详细解释一下为什么表示/容器是一个反模式吗?说实话，这是我第一次听说。据我所知，presentation/Container模式的灵感来自React社区。现在被认为是反模式，你可以在Dan Abramov 2019年的更新部分看到。就我个人而言，在阅读Dan的新观点之前，我已经停止使用这个模式了，原因很简单。组件应该专注于呈现视图和响应用户操作。在复杂的应用程序中，任何其他责任都应该委托给服务。我们不应该使用没有可视化渲染的组件来污染组件树。用一个有约束的框架来做这件事是可以理解的。但是，在Angular中，我们有服务。

在Angular中实现可重用的响应式表单

<https://indepth.dev/posts/1415/implementing-reusable-and-reactive-forms-in-angular-2>

在本文中，我们将学习两种实现响应式和可重用表单的方法，它们既可以用作子表单，也可以用作独立表单。  
我假设你知道表单是什么，并且已经使用过Angular响应式表单。

我们将着眼于实现这一目标的两种方法:

1 使用ControlContainer，可以将父窗体传递给其子窗体，其子窗体作为组件实现。  
2 @ViewChild帮助我们获得组件的类实例这里是表单组件实例。

Angular提供了两种配置和实现表单的方法，它们是响应式表单和模板驱动表单，在本文中我们将提到响应式表单。

正如上面的视频所示，我们有一个带有两个子窗体的大窗体:  
HeroComponent(父)  
 PowersComponent(使用@ViewChild装饰器查询的子表单)  
 HobbiesComponent(用ControlContainer类实现的子表单)

使用@ViewChild装饰器实现子表单  
让我们从最好的方法开始，使用一个装饰器来让父窗体(HeroComponent)查询子窗体/子窗体的组件类实例，也就是PowersComponent，见下面的代码:

****HeroComponent - Parent form****

// hero.component.html

<form [formGroup]="heroForm">

<nb-card>

<nb-card-header>Hero</nb-card-header>

<nb-card-body class="col">

<input

formControlName="heroName"

type="text"

nbInput

placeholder="Hero name"

/>

<input formControlName="aka" type="text" nbInput placeholder="AKA" />

</nb-card-body>

</nb-card>

<nb-card>

<nb-card-header>Super Power</nb-card-header>

<nb-card-body class="col">

<app-powers></app-powers>

</nb-card-body>

</nb-card>

<nb-card>

<nb-card-header>Hobbies</nb-card-header>

<nb-card-body class="col">

<app-hobbies

[parentForm]="heroForm"

[formGroup]="heroForm.get('hobbies')"

></app-hobbies>

</nb-card-body>

</nb-card>

<button (click)="logFormData()" nbButton status="primary">Submit</button>

</form>

@Component({

selector: 'app-hero',

templateUrl: './hero.component.html',

styleUrls: ['./hero.component.scss']

})

export class HeroComponent implements OnInit {

@ViewChild(PowersComponent, { static: true }) public powersComponent: PowersComponent;

public heroForm: FormGroup;

constructor(private formBuilder: FormBuilder) {

}

public ngOnInit(): void {

this.heroForm = this.formBuilder.group({

heroName: ['', Validators.required],

aka: ['', Validators.required],

powers: this.powersComponent.createFormGroup(),

hobbies: this.formBuilder.group({

favoriteHobby: ['', Validators.required]

})

})

}

public logFormData(): void {

console.log(this.heroForm.value);

}

}

正如你在HeroComponent的模板中看到的那样，PowersComponent不需要进一步的输入

<nb-card>

<nb-card-header>Super Power</nb-card-header>

<nb-card-body class="col">

<app-powers></app-powers> // here

</nb-card-body>

</nb-card>

但是如果你检查一下HeroComponent类，你就会注意到我们如何获得Powers子表单组件类的实例，并调用createFormGroup公共成员函数来返回PowersComponent FormGroup配置实例。

@ViewChild(PowersComponent, { static: true }) public powersComponent: PowersComponent;

注意@ViewChild选项如何使用静态:true尽快解析组件实例，以便我们有PowersComponent类的子表单实例。  
检查HeroComponent的第21行。创建PowersComponent表单的ts文件。

powers: this.powersComponent.createFormGroup(),

这就是如何使用@ViewChild装饰器创建可重用的响应式子表单，我倾向于使用@ViewChild装饰器，因为我发现它很容易实现，也更容易维护，因为子表单和父表单彼此分离，并提供了以下好处:

1 父表单根本不需要知道子表单的实例。它所需要的只是子表单组件类有一个createFormGroup公共成员函数，该函数返回一个FormGroup的实例。  
2 子表单的表单配置上的更改不会以任何方式影响父表单的配置或它的模板。

很好，那么为这样的设置设置单元测试怎么样呢?  
简单易用，检查下面的代码  
@ViewChild方法的单元测试

describe('HeroComponent', () => {

let component: HeroComponent;

let fixture: ComponentFixture<HeroComponent>;

const formBuilder: FormBuilder = new FormBuilder();

const powersComponent = jasmine.createSpyObj('PowersComponent', ['createFormGroup']);

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations: [HeroComponent, HobbiesStubComponent],

providers: [{ provide: FormBuilder, useValue: formBuilder }],

imports: [FormsModule, ReactiveFormsModule]

})

.compileComponents();

}));

beforeEach(() => {

fixture = TestBed.createComponent(HeroComponent);

component = fixture.componentInstance;

component.powersComponent = powersComponent;

fixture.detectChanges();

});

it('should create', () => {

expect(component).toBeTruthy();

});

});

如果您检查上面代码的第11行和第25行，就会注意到我们是如何使用一个假函数创建一个假PowersComponent的，这个假函数将使用Jasmine的createSpyObj函数来模拟createFormGroup函数。

const powersComponent =

jasmine.createSpyObj('PowersComponent'['createFormGroup']; // line 11

...

component.powersComponent = powersComponent; // line 25

如果在HeroComponent的规范文件中没有上述内容，你将在运行以下测试时得到一个错误:

TypeError: Cannot read property 'createFormGroup' of undefined

**使用ControlContainer实现子表单**

这种方法令人困惑，也有很多工作要做，但也是可行的。  
在HeroComponent的模板中，我们有另一个子表单HobbiesComponent的标记

<app-hobbies [parentForm]="heroForm [formGroup]="heroForm.get('hobbies')"

></app-hobbies>

正如你所看到的，这个表单标记使用了ControlContainer自带的[formGroup]指令，如果需要的话，允许父表单将子表单传递给子表单本身。  
还请注意，子表单期望的输入应该是父表单，这也使我们能够在需要时访问子表单中的父表单，参见下面的子表单的源代码。

**HobbiesComponent - Uses ControlContainer**

@Component({

selector: 'app-hobbies',

templateUrl: './hobbies.component.html',

styleUrls: ['./hobbies.component.scss']

})

export class HobbiesComponent implements OnInit {

public hobbiesForm: FormGroup;

@Input() parentForm: FormGroup;

constructor(private controlContainer: ControlContainer) { }

public ngOnInit(): void {

this.hobbiesForm = this.controlContainer.control as FormGroup;

}

public logForms(): void {

console.log('Hobbies form', this.hobbiesForm);

console.log('Parent (Hero) form', this.parentForm);

}

}

<https://indepth.dev/posts/1055/never-again-be-confused-when-implementing-controlvalueaccessor-in-angular-forms>

**ControlValueAccessor**

如果你正在处理一个复杂的项目，不可避免地你会遇到这样的情况，你必须创建一个自定义的表单控件。这个任务的基本组成部分是实现ControlValueAccessor。网站上有一些文章解释了如何实现它，但没有一篇文章深入剖析了这个组件在Angular表单体系结构中扮演的角色。如果你不仅想知道如何实现它，还想知道为什么这篇文章适合你。

在这里，我将首先解释为什么我们需要ControlValueAccessor，以及如何在Angular中使用它。然后我将演示如何将第三方小部件包装到Angular组件中，并使用标准的输入/输出机制建立与父组件的通信。最后，我将展示如何实现ControlValueAccessor，它为Angular表单引入了一种新的通信机制。

隐式或显式创建的formControl必须与原生表单控件(如input或textarea)交互。除了原生的表单控件，还可以创建一个自定义的表单控件作为Angular组件。自定义表单控件通常封装使用纯JavaScript编写的控件，比如jQueryUI的滑块。在这篇文章中，我将使用“原生表单控件”来区分Angular特有的表单控件和HTML中的表单控件。但你应该理解，不是像输入那样的原生表单控件，而是任何自定义表单控件

FormControl and ControlValueAccessor

如果你之前在Angular中使用过表单，那么你可能对FormControl很熟悉。Angular文档将其描述为一个实体，用于跟踪单个表单控件的值和验证状态。当你使用表单时，不管你是使用模板驱动还是响应式表单，都会创建一个FormControl，理解这一点很重要。使用响应式方法，您可以显式地自己创建控件，并使用formControl或formControlName指令将其绑定到本机控件。如果你使用模板驱动的方法，FormControl是隐式创建的

@Directive({

selector: '[ngModel]...',

...

})

export class NgModel ... {

\_control = new FormControl(); <---------------- here

本机表单控件的数量是有限的，但自定义表单控件的种类可能是无限的。所以，Angular需要一种通用机制来介于Angular的formControl和原生/自定义表单控件之间。这就是ControlValueAccessor对象发挥作用的地方。这个对象位于Angular formControl和原生表单控件之间，用于同步两者之间的值。以下是医生们对此的看法:

一个ControlValueAccessor充当了Angular表单API和DOM中的原生元素之间的桥梁。

通过实现ControlValueAccessor接口并将其注册为NG\_VALUE\_ACCESSOR提供程序，任何组件或指令都可以转换为ControlValueAccessor。我们稍后将看到如何做到这一点。其中，该接口定义了两个重要的方法——writeValue和registerOnChange:

interface ControlValueAccessor {

writeValue(obj: any): void

registerOnChange(fn: any): void

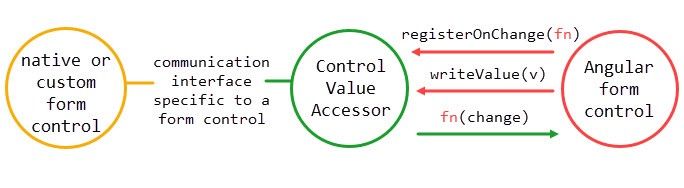
registerOnTouched(fn: any): void

...

}

formControl使用writeValue方法将值设置为原生表单控件。formControl使用registerOnChange方法来注册一个回调，每次本机表单控件更新时都将触发该回调。把更新后的值传递给这个回调函数是你的责任，这样Angular表单控件的值就会被更新。registerOnTouched方法用于指示用户与控件进行了交互。

下图演示了一个交互过程:



同样，理解controlValueAccessor总是与显式(响应式表单)或隐式(模板驱动)创建的表单控件交互是很重要的。  
Angular为所有标准的原生表单元素实现了默认值访问器:

+------------------------------------+----------------------+

| Accessor | Form Element |

+------------------------------------+----------------------+

| DefaultValueAccessor | input, textarea |

| CheckboxControlValueAccessor | input[type=checkbox] |

| NumberValueAccessor | input[type=number] |

| RadioControlValueAccessor | input[type=radio] |

| RangeValueAccessor | input[type=range] |

| SelectControlValueAccessor | select |

| SelectMultipleControlValueAccessor | select[multiple] |

+------------------------------------+----------------------+

正如你所看到的，DefaultValueAccessor是在Angular遇到组件模板中的输入区域或文本区域时使用的:

@Component({

selector: 'my-app',

template: `

<input [formControl]="ctrl">

`

})

export class AppComponent {

ctrl = new FormControl(3);

}

所有的表单指令，包括上面使用的formControl指令，都调用setUpControl函数来设置formControl和ControlValueAccessor之间的交互。下面是演示formControl指令的代码片段:

export class FormControlDirective ... {

...

ngOnChanges(changes: SimpleChanges): void {

if (this.\_isControlChanged(changes)) {

setUpControl(this.form, this);

下面是setUpControl函数的要点，它展示了原生控件和Angular表单控件是如何同步的:

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

// initialize a form control

dir.valueAccessor.writeValue(control.value);

// setup a listener for changes on the native control

// and set this value to form control

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

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

});

// setup a listener for changes on the Angular formControl

// and set this value to the native control

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

dir.valueAccessor.writeValue(newValue);

});

一旦我们理解了这些机制，我们就可以继续为自定义表单控件实现自己的访问器。

**实现小部件包装器**

由于Angular为所有默认的原生控件提供了控件值访问器，所以新实现的值访问器通常用于包装第三方插件/小部件。我之前提到过一个来自jQueryUI库的滑块小部件，这是我们将用于自定义表单控件的插件。

**简单的包装**

让我们从包装小部件并将其显示在屏幕上的最基本实现开始。为此，我们实现了一个新的NgxJquerySliderComponent，并从它的模板中使用一个DOM元素来渲染slider:

@Component({

selector: 'ngx-jquery-slider',

template: `

<div #location></div>

`,

styles: ['div {width: 100px}']

})

export class NgxJquerySliderComponent {

@ViewChild('location') location;

widget;

ngOnInit() {

this.widget = $(this.location.nativeElement).slider()

}

}

这里，我们使用标准的jQuery方法在原生DOM元素上创建一个滑块小部件。然后将对小部件的引用保存到小部件属性中。

一旦我们准备好了包装器组件，我们就可以像这样在父App组件中使用它:

@Component({

selector: 'my-app',

template: `

<h1>Hello {{name}}</h1>

<ngx-jquery-slider></ngx-jquery-slider>

`

})

export class AppComponent { ... }

要运行应用程序，我们需要包含jQuery相关的依赖项。为简单起见，我们将它们作为全局依赖项添加到index.html中:

<script src="https://code.jquery.com/jquery-3.2.1.js">

</script>

<script src="https://code.jquery.com/ui/1.12.1/jquery-ui.js">

</script>

<link rel="stylesheet" href="//code.jquery.com/ui/1.12.1/themes/smoothness/jquery-ui.css">

下面是演示该设置的应用程序。

<https://stackblitz.com/edit/angular-pemw12>

交互式表单控件

通过上面的实现，我们的自定义滑块控件没有任何方法与父组件交互。所以让我们使用标准的输入/输出机制作为通信通道:

export class NgxJquerySliderComponent {

@ViewChild('location') location;

@Input() value;

@Output() private valueChange = new EventEmitter();

widget;

ngOnInit() {

this.widget = $(this.location.nativeElement).slider();

this.widget.slider('value', this.value);

this.widget.on('slidestop', (event, ui) => {

this.valueChange.emit(ui.value);

});

}

ngOnChanges() {

if (this.widget && this.widget.slider('value') !== this.value) {

this.widget.slider('value', this.value);

}

}

}

一旦创建了滑块小部件，我们就使用slidestop事件订阅它的值更改。一旦事件被触发，我们使用valueChanges输出事件发射器通知父类。我们还使用ngOnChanges生命周期钩子跟踪输入值绑定的变化，一旦值被更新，我们就把它设置到slider小部件中。

下面是我们如何在父App组件中使用组件:

<ngx-jquery-slider

[value]="sliderValue"

(valueChange)="onSliderValueChange($event)">

</ngx-jquery-slider>

下面是演示该设置的应用程序。

<https://plnkr.co/edit/bCrkvABQkRZXrnVvTW7D?p=preview&preview>

然而，如果我们想要使用滑块作为表单的一部分，并使用模板驱动或响应式表单指令与它通信，我们需要实现一个值访问器。而且我们不需要标准的输入/输出通信机制，所以我们在实现值访问器时将其删除。

**实现自定义值访问器**实现自定义值访问器并不困难。它需要2个简单的步骤:  
1 注册一个NG\_VALUE\_ACCESSOR

2 提供程序实现ControlValueAccessor接口方法

NG\_VALUE\_ACCESSOR提供程序指定了一个实现ControlValueAccessor接口的类，Angular用它来设置与formControl的同步。通常是组件或指令的类来注册提供程序。所有的表单指令都使用NG\_VALUE\_ACCESSOR令牌注入值访问器，然后选择一个合适的访问器。如果有一个访问器不是内置的或DefaultValueAccessor，它将被选择。否则，如果提供了默认访问器，Angular会选择它。并且一个元素最多只能定义一个自定义访问器。  
我们先来定义

@Component({

selector: 'ngx-jquery-slider',

providers: [{

provide: NG\_VALUE\_ACCESSOR,

useExisting: NgxJquerySliderComponent,

multi: true

}]

...

})

class NgxJquerySliderComponent implements ControlValueAccessor {...}

我们直接在组件装饰描述符中指定了这个类。然而，Angular实现的所有默认访问器都在类元数据之外定义了一个provide，如下所示:

export const DEFAULT\_VALUE\_ACCESSOR: any = {

provide: NG\_VALUE\_ACCESSOR,

useExisting: forwardRef(() => DefaultValueAccessor),

multi: true

};

@Directive({

selector:'input',

providers: [DEFAULT\_VALUE\_ACCESSOR]

...

})

export class DefaultValueAccessor implements ControlValueAccessor {}

所以他们需要使用forwardRef。要了解更多关于forward dref的知识，请阅读什么是' forward dref '以及我们为什么需要它。

<https://indepth.dev/posts/1133/what-is-forwardref-in-angular-and-why-we-need-it>

当实现自定义controlValueAccessor时，我建议直接在装饰器描述符中指定一个类。

一旦我们定义了一个提供程序，让我们实现ControlValueAccessor接口:

export class NgxJquerySliderComponent implements ControlValueAccessor {

@ViewChild('location') location;

widget;

onChange;

value;

ngOnInit() {

this.widget = $(this.location.nativeElement).slider(this.value);

this.widget.on('slidestop', (event, ui) => {

this.onChange(ui.value);

});

}

writeValue(value) {

this.value = value;

if (this.widget && value) {

this.widget.slider('value', value);

}

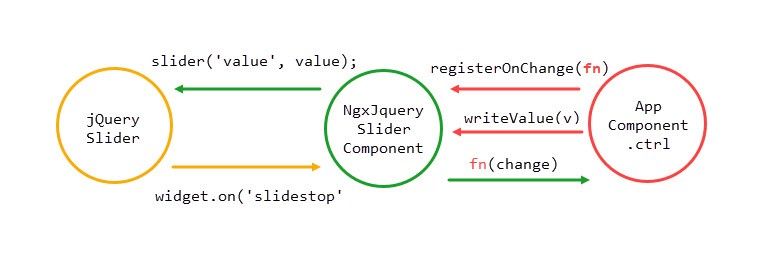
}

registerOnChange(fn) { this.onChange = fn; }

registerOnTouched(fn) { }

我们对用户是否与控件进行交互不感兴趣，所以我们将registerOnTouched设置为空。在registerOnChange内部，我们简单地保存对formControl传递的回调fn函数的引用。每次滑动条值发生变化时，我们都会触发它。在writeValue方法中，我们将值设置为slider小部件。

那么现在，如果我们把上面的功能描述到交互图片中，它看起来是这样的:



如果将两个实现作为简单的包装器和controlValueAccessor进行比较，您应该会看到与父组件的交互是不同的，而与下划线滑块小部件的交互是相同的。你可能还注意到formControl实际上简化了与父组件的交互。我们在简单包装器中使用ngOnChanges的地方使用writeValue，并调用它。onChange是我们之前使用this.valueChange.emit(ui.value)发出值的地方。

自定义滑块控件实现为ControlValueAccessor，现在可以这样使用:

@Component({

selector: 'my-app',

template: `

<h1>Hello {{name}}</h1>

<span>Current slider value: {{ctrl.value}}</span>

<ngx-jquery-slider [formControl]="ctrl"></ngx-jquery-slider>

<input [value]="ctrl.value" (change)="updateSlider($event)">

`

})

export class AppComponent {

ctrl = new FormControl(11);

updateSlider($event) {

this.ctrl.setValue($event.currentTarget.value, {emitModelToViewChange: true});

}

}

您可以在这里找到最终的实现。

<https://github.com/maxkoretskyi/custom-form-control-that-implements-control-value-accessor-and-wraps-jquery-slider>

Angular:使用控制值访问器ControlValueAccessors(CVAs)嵌套的响应式表单

<https://indepth.dev/posts/1245/angular-nested-reactive-forms-using-controlvalueaccessors-cvas>

**实现嵌套表单的三种方法:**我们将看到实现嵌套表单最常用的技术。

**1 子表单组件方法(提供ControlContainer)**

对于模板和响应驱动表单，它的构建方式是不同的。

最好是当你有一个非常小的项目，你只使用一个表单模块，你只需要分离出一个非常长的表单(节选自演讲)。

优点:安装和运行更快。  
缺点:仅限于一个表单模块

1. **通过input将FormGroup的句柄传递给子组件，并在子模板中引用它。关于它有一些很好的教程。**  
   但是使用这种方法的缺点是，您将父窗体组与子窗体组紧密地绑定在一起。
2. **使用复合Composite CVAs。**  
   优点:高度可重复使用，可携带。更好的封装(组件的内部表单控件不一定要对父组件可见)。当你有更多的表单模块(通常是一个大项目)时，这是最好的使用方法。  
   缺点:需要实现CVA接口导致样板代码。  
   这就是我们现在要看到的。

**什么是控制值访问器Value Accessor (CVA)?**  
以下是谷歌的开发者们对这个问题的看法?

ControlValueAccessor接口实际上是你用来为核心中的单选按钮，选择，输入元素等构建值访问器的。它只需要三个方法:

writeValue(value: any): void : 接受一个值并将其写入表单控件元素(model -> view)

registerOnChange(fn: (value:any) => void): void: 接受一个函数，如果表单控件元素本身的值发生变化，则应使用该值调用该函数 (view -> model)

registerOnTouched(fn: () => void): 接受一个当窗体控件被触摸时被调用的函数(如果你不关心触摸属性，这个函数可以保留为空)

实现 Composite CVAs

本节假设你对响应式表单有一定的了解，主要是FormGroups, FormControls, Validations等。

我创建了一个名为billing-info- un嵌套的示例响应式表单组件。

@Component({

selector: 'app-billing-info-unnested',

template: `

<div class="container">

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

<div class="row">

<label for="Full Name"> Full Name </label>

<input type="text" formControlName="fname" class="">

</div>

<div class="row">

<label for="Email"> Email </label>

<input type="text" formControlName="email" class="">

</div>

<div class="row">

<label for="addressLine"> Street Address </label>

<input type="text" formControlName="addressLine" class="">

</div>

<div class="row">

<label for="Area"> Area Code </label>

<input type="text" formControlName="areacode" class="">

</div>

<button type="submit" [disabled]="nestedForm.invalid">Place Order</button>

</form>

</div>`,

styleUrls: ['./billing-info-unnested.component.css']

})

export class BillingInfoUnnestedComponent implements OnInit {

public nestedForm: FormGroup = new FormGroup({

fname: new FormControl("", [Validators.required]),

email: new FormControl("", [Validators.required, Validators.email]),

addressLine: new FormControl("", [Validators.required]),

areacode: new FormControl("", [Validators.required, Validators.maxLength(5)])

})

constructor() { }

ngOnInit() {

}

public onSubmit(){

// if(this.nestedForm.invalid){

// return

// }

console.log(" Billing Form", this.nestedForm);

}

}

Output:

nestedForm {

fname:"",

email: "",

addressLine: "",

areacode: ""

}

现在想象一下，如果我们的客户端引入了更多的需求，并且我们需要重用这些控件以及一些附加的表单字段，比如。

收款单:发货类型。

登入/登记:密码、性别、年龄等

在这个场景中，我们可以通过将它们分组为组件并将它们转换为表单控件来重用它们。也就是说，我们将把组件构建为一个复合控件值访问器。我们甚至可以在组件级别提供验证。这个技巧很神奇，相信我。: D

让我们把name和email移到BasicInfoComponent中，把addressLine和arecode移到AddressComponent中。

这里我们以BasicInfoComponent为例。

<ng-container [formGroup]="basicInfoForm">

<div class="row">

<label for="Full Name"> Full Name </label>

<input type="text" formControlName="fname" class="">

</div>

<div class="row">

<label for="Email"> Email </label>

<input type="text" formControlName="email" class="">

</div>

</ng-container>

ts文件看起来是这样的

import { Component, OnInit, forwardRef } from '@angular/core';

import { ControlValueAccessor,FormControl, FormGroup, Validators } from "@angular/forms";

@Component({

selector: 'app-basic-info',

templateUrl: './basic-info.component.html',

styleUrls: ['./basic-info.component.css'],

})

export class BasicInfoComponent implements OnInit {

public basicInfoForm: FormGroup = new FormGroup(

{

fname: new FormControl("",[Validators.required]),

email: new FormControl("", [Validators.required])

});

constructor() { }

ngOnInit() {

}

}

对AddressComponent做同样的事情。

现在父窗体组件BillingInfo看起来是这样的

@Component({

selector: 'app-billing-component',

template:`

<div class="container">

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

<app-basic-info formControlName="basicInfo"></app-basic-info>

<app-address-info formControlName = "address"></app-address-info>

<button type="submit" [disabled]="nestedForm.invalid">Place Order</button>

</form>

</div>

`,

styleUrls: ['./billing-info.component.css']

})

export class BillingInfoComponent implements OnInit {

public nestedForm: FormGroup = new FormGroup({

basicInfo: new FormControl(""),

address: new FormControl("")

});

constructor() { }

ngOnInit() {

}

public onSubmit(){

console.log("Billing Info", this.nestedForm.value);

}

}

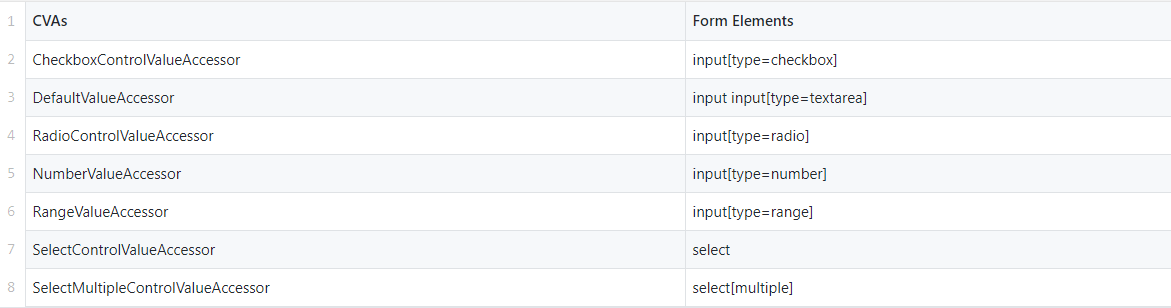
现在让我们试着运行这个。这是Stackblitz的演示

错误!

当我们尝试运行演示时，我们将不幸地遇到错误。

Error: No value accessor for form control with name: ' basicInfo '

让我们来看看Angular Core提供的内置值访问器。



因为我们的表单控件(组件)不属于这些类别，所以angular编译器会抛出错误，说没有找到值访问器。  
如果你想让你的自定义表单控件与Angular表单集成，它必须实现ControlValueAccessor

将自定义表单控件集成到Angular表单中

让我们实现ControlValueAccessor接口，覆盖BasicInfoComponent和AddressInfoComponent中的所有方法。这就是AddressComponent的样子

import { Component, OnInit } from '@angular/core';

import { ControlValueAccessor,NG\_VALUE\_ACCESSOR, NG\_VALIDATORS, FormGroup,FormControl, Validator, Validators,AbstractControl, ValidationErrors } from "@angular/forms";

@Component({

selector: 'app-address-info',

templateUrl: './address-info.component.html',

styleUrls: ['./address-info.component.css']

})

export class AddressInfoComponent implements OnInit, ControlValueAccessor {

public addressForm: FormGroup = new FormGroup({

addressLine: new FormControl("",[Validators.required]),

areacode: new FormControl('', [Validators.required, Validators.maxLength(5)])

});

constructor() { }

ngOnInit() {

}

public onTouched: () => void = () => {};

writeValue(val: any): void {

val && this.addressForm.setValue(val, { emitEvent: false });

}

registerOnChange(fn: any): void {

console.log("on change");

this.addressForm.valueChanges.subscribe(fn);

}

registerOnTouched(fn: any): void {

console.log("on blur");

this.onTouched = fn;

}

setDisabledState?(isDisabled: boolean): void {

isDisabled ? this.addressForm.disable() : this.addressForm.enable();

}

}

在实现了访问器之后，我们需要告诉angular，对于<app-address-info></app-address-info>表单控件元素，这是它的相关控件值访问器。

我们怎么才能做到呢?

让我们来看看DefaultValueAccessor是如何在Angular Forms包中提供的。

export const DEFAULT\_VALUE\_ACCESSOR: any = {

provide: NG\_VALUE\_ACCESSOR,

useExisting: forwardRef(() => DefaultValueAccessor),

multi: true

};

让我们来分析上面的语法。

1 这里DefaultValueAccessor是使用内置令牌NG\_VALUE\_ACCESSOR注册的。

2 forwardref()表示引用尚未定义的引用。使用DefaultValueAccessor实例，这个实例稍后会被Angular实例化。

3 useExisting()用来确保DefaultValueAccessor只有一个实例。

4 提供程序对象还有第三个选项multi: true，它与DI令牌一起用于为提供事件注册多个处理程序。如果我们想将我们的自定义CVAs注册到NG\_VALUE\_ACCESSOR令牌，这是很有用的

现在，我们将对BasicInfoComponent和AddressInfoComponent进行同样的操作，并再次运行演示程序。

<https://stackblitz.com/edit/angular-simple-reactive-forms-cawhs7>

我们看到我们的错误消失了。万岁! !我们已经成功地将嵌套的子表单控件集成到Angular Core中。但是还有一个问题。

即使我们的下订单按钮应该被禁用，如果表单无效，我们看到它不是。

<button type=”submit” [disabled]=”nestedForm.invalid”>Place Order</button>

让我们在开发人员工具中检查。

**子组件的验证状态失败**

在Inspect中，我们可以看到自定义表单控件(子组件)<app-basic-info></app-basic-info>状态是有效的，而在basic-info组件中的表单控件仍然是无效的。

在前一节中，我们了解了这一点

如果你想让你的自定义表单控件与Angular表单集成，它必须实现ControlValueAccessor。

现在，如果我们想要在集成中包含验证，我们还需要实现Validator接口，并将自定义控件作为一个多提供商提供给内置的NG\_VALIDATOR令牌。

**原因:**

对于Re-validation，如果您希望它成为父表单验证的一部分，验证器将需要位于顶级表单上，而不是子组件上。

在我们的例子中，我们需要在BillingInfoComponent级别上有验证器，因此为了达到这个目的，我们需要将组件转换为验证器指令，比如required、min、max等。

让我们在子组件中实现Validator接口。

看看在angular forms包中是如何提供required指令的。

export const REQUIRED\_VALIDATOR: StaticProvider = {

provide: NG\_VALIDATORS,

useExisting: forwardRef(() => RequiredValidator),

multi: true };

我们需要对自定义表单验证器做同样的事情。我们需要使用内置的NG\_VALIDATORS令牌注册一个自定义表单验证器，并通过在provider对象中使用multi: true属性来提供多个验证器提供者实例。这告诉Angular把我们的自定义验证器添加到现有的集合中。

@Component({

selector: 'app-basic-info',

template: `

<ng-container [formGroup]="basicInfoForm">

<div class="row">

<label for="Full Name"> Full Name </label>

<input type="text" formControlName="fname" class="">

</div>

<div class="row">

<label for="Email"> Email </label>

<input type="text" formControlName="email" class="">

</div>

</ng-container>`,

styleUrls: ['./basic-info.component.css'],

providers: [

{

provide: NG\_VALUE\_ACCESSOR,

useExisting: forwardRef(() => BasicInfoComponent),

multi: true

},

{

provide: NG\_VALIDATORS,

useExisting: forwardRef(() => BasicInfoComponent),

multi: true

}

]

})

export class BasicInfoComponent implements OnInit, ControlValueAccessor, Validator {

public basicInfoForm: FormGroup = new FormGroup(

{

fname: new FormControl(""),

email: new FormControl("")

});

constructor() { }

ngOnInit() {

}

public onTouched: () => void = () => {};

writeValue(val: any): void {

val && this.basicInfoForm.setValue(val, { emitEvent: false });

}

registerOnChange(fn: any): void {

console.log("on change");

this.basicInfoForm.valueChanges.subscribe(fn);

}

registerOnTouched(fn: any): void {

console.log("on blur");

this.onTouched = fn;

}

setDisabledState?(isDisabled: boolean): void {

isDisabled ? this.basicInfoForm.disable() : this.basicInfoForm.enable();

}

validate(c: AbstractControl): ValidationErrors | null{

console.log("Basic Info validation", c);

return this.basicInfoForm.valid ? null : { invalidForm: {valid: false, message: "basicInfoForm fields are invalid"}};

}

}

：因此，通过这样做，我们得到了<app-basic-info></app-basic-info>和<app-address-info></app-basic-info>重新验证(通过调用validate方法，该方法反过来验证子组件中存在的所有表单控件)，状态被发送到父表单组件。

https://stackblitz.com/edit/angular-nested-forms-cva

**什么是“forward dref”在Angular中，为什么我们需要它**

**它是什么**

让我们从关于forwardRef的Angular官方文档开始。上面说:

允许引用尚未定义的引用…

例如，当我们需要引用用于DI的令牌已经声明，但还没有定义时，就会使用forwardRef。当我们创建查询时使用的令牌还没有定义时，也会使用它。

定义谈到了对类的引用，并以引用类的标记为例。在Angular中，我们这样定义依赖:

const dependency = {

provide: SomeTokenClass,

useClass: SomeProviderClass

};

在上面的例子中，有一个为provide指定的令牌和一个recipe - useClass。因此，从定义中我们知道，我们可以像这样使用forwardRef作为令牌:

const dependency = {

provide: forwardRef(()=>{ SomeTokenClass }),

useClass: SomeProviderClass

};

但是我们在useClass配方中也有一个对SomeProviderClass类的引用。我们能很好地为提供者使用这种方法吗?文档没有提到这一点，但是我们知道useClass recipe持有一个对类的引用，我们知道forward dref可以应用到一个引用。所以答案是肯定的，我们也可以将这种方法应用到提供商recipe中:

const dependency = {

provide: forwardRef(()=>{ SomeTokenClass }),

useClass: forwardRef(()=>{ SomeProviderClass })

};

然而，只有当菜谱暗示了对类的引用时，它才能被应用，就像下面的例子中的useClass或useExisting那样:

const dependency = {

provide: forwardRef(()=>{ SomeTokenClass }),

useExisting: forwardRef(()=>{ SomeOtherClassToken })

};

同样，如果你使用inject装饰器通过类引用注入一个令牌，你也可以应用这个函数:进入翻译页面

export class ADirective {

constructor(@Inject(forwardRef(() => Token)) service) {}

...

}

使用的例子

Angular文档显示了以下示例:

class Door {

lock: Lock;

// Door attempts to inject Lock,

// despite it not being defined yet.

// forwardRef makes this possible.

constructor(@Inject(forwardRef(() => Lock)) lock: Lock) {

this.lock = lock;

}

}

// Only at this point Lock is defined.

class Lock {}

但对我来说，这个例子有点不自然。虽然它得到的观点，它是很难理解看它时，我将需要使用forwardRef在真实的应用程序。我可以简单地把Lock类放在Door上面，问题就解决了。事实上，Angular源代码提供了一个更好的实际用法示例。

你可能知道，Angular表单有ngModel和formControl指令，你可以在表单输入中使用它们。每个控件都定义了一个提供者，允许通过公共令牌NgModel访问指令实例。例如，如果你想访问与自定义指令中的输入相关联的表单指令，你可以这样做:

@Directive({

selector: '[mycustom]'

})

export class MyCustom {

constructor(@Inject(NgControl) directive) {

...

<input type="text" ngModel mycustom>

为了让每个NgModel和formControl指令都定义一个formControlBinding提供程序，并将其注册到指令的装饰描述符中。formControl指令是这样做的:

export const formControlBinding: any = {

provide: NgControl,

useExisting: FormControlDirective

};

@Directive({

selector: '[formControl]',

providers: [formControlBinding],

...

})

export class FormControlDirective { ... }

和NgModel指令:

export const formControlBinding: any = {

provide: NgControl,

useExisting: NgModel

};

@Directive({

selector: '[ngModel]',

providers: [formControlBinding],

...

})

export class NgModel { ... }

这里实现的有趣之处在于formControlBinding被定义在指令类装饰器之外。所以当JS运行时计算定义formControlBinding对象的代码时，NgModel类定义还没有被计算，如果我们将provider对象记录到控制台，我们将看到以下内容:

Object {useExisting: undefined, token: function}

啊，useExisting指向undefined，所以Angular将无法解析其他令牌。这就是Angular在这里使用forwardRef的原因:

export const formControlBinding: any = {

provide: NgControl,

useExisting: forwardRef(() => FormControlDirective)

};

export class FormControlDirective { ... }

...

export const formControlBinding: any = {

provide: NgControl,

useExisting: forwardRef(() => NgModel)

};

export class NgModel{ ... }

但如果我们在类装饰器中像这样定义formControlBinding而不使用forwardRef，它能工作吗?

@Directive({

selector: '[ngModel]',

providers: [

{

provide: NgControl,

useExisting: NgModel

}

],

...

})

export class NgModel { ... }

好吧，如果你看一下代码，你会发现NgModel在类定义之前就在装饰器中被引用了。但是您应该记住，所有的类装饰器都是在类被定义之后应用到类上的。因此，即使没有forwardRef，上述实现也可以工作。但是，通过将提供程序内联到装饰器中，我们将不再导出它，因此它不能在应用程序中重用。

为什么forwardRef工作?

现在问题可能会出现在你的脑海中，forward dref是如何工作的。它实际上与JavaScript中的闭包工作方式有关。当您在闭包函数中捕获一个变量时，它捕获的是变量引用，而不是变量值。下面是一个小例子:

let a;

function enclose() {

console.log(a);

}

enclose(); // undefined

a = 5;

enclose(); // 5

您可以看到,虽然变量a在被创建的时刻没有定义,但它捕获了变量引用。因此,当这个变量被更新到5时,它记录了正确的值。

而forwardRef只是一个函数，它将类引用捕获到闭包中，并且类在函数执行之前被定义。Angular编译器在运行时使用resolvefordref函数来打开令牌或提供商类型。