<https://indepth.dev/posts/1198/angular-forms-story-strong-types>

<https://dev.to/maxime1992/building-scalable-robust-and-type-safe-forms-with-angular-3nf9>

1 上下文  
2 介绍演示并理解我们想要构建的示例  
3 [Reactive forms](https://dev.to/maxime1992/building-scalable-robust-and-type-safe-forms-with-angular-3nf9" \l "reactive-forms)

所有内容都放在一个文件中

将表单分解为子组件  
将表单分解成子组件，这是正确的方法!  
理解ControlValueAccessor的威力

ngx sub - form:这个实用程序库可以把一个Angular表单分解成多个子组件  
ngx子表单提供了什么  
什么时候应该使用它?  
构建演示  
进一步讨论重映射和/或多态性

总结和结论

有用的链接

<https://indepth.dev/posts/1224/angular-forms-useful-tips>

Angular表单:有用的技巧

一 表单不是类型安全的

就是这样，我说过了。表单不是类型安全的，所以我们不能依赖TypeScript来为我们捕获bug、容易出错的代码和拼写错误，这意味着我们在处理表单时必须格外小心。但当然，谨慎并不是唯一有用的事情——我们可以做几件事来让我们的生活更轻松。

1 限制使用FormGroup。当引用FormGroup中的嵌套控件时获得用法。相反，将对嵌套控件的引用存储在组件属性中。

2 当将表单的数据模型更改为数据服务所需的数据模型时，使用DTO模式。

3 将复杂的表单控件抽象到实现ControlValueAccessor的自定义组件中

我们来看看这些情况。

案例1:限制使用FormGroup。得到使用

考虑下面这段代码:

@Component({

selector: 'my-app',

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

styleUrls: [ './app.component.css' ]

})

export class AppComponent {

form = this.formBuilder.group({

firstName: [''],

lastName: [''],

age: [''],

});

constructor(private formBuilder: FormBuilder) {}

}

这是一个简单的FormGroup组件，没有什么特别的。

现在看一下模板:

<form [formGroup]="form">

<div>

<label for="firstName">First Name</label>

<input formControlName="firstName" id="firstName"/>

<span \*ngIf="form.get('firstName').touched && form.get('firstName').hasError('required')" class="errors">

Field is required

</span>

</div>

<div>

<label for="lastName">Last Name</label>

<input formControlName="lastName" id="lastName"/>

<span \*ngIf="form.get('lastName').touched && form.get('larstName').hasError('required')" class="errors">

Field is required

</span>

</div>

</form>

看起来很普通，没什么难的。但你看到问题了吗?

it says ‘larstName’ instead of ‘lastName’, 它会引起问题。更糟糕:在开发过程中没有错误在控制台之前修改表单值——这很可能是当QA工程师正在测试页面(老实说,许多软件工程师也会认为这个代码是简单测试自己全部),甚至错误

对于许多开发者来说，乍一看会觉得很神秘。而且，多次写“form.get(' firstName ')”很乏味(并且增加了进一步拼写错误的机会)。此外，我们不能真正依赖智能感知自动完成，所以我们必须复制粘贴多次。对于软件开发人员来说，这并不是一件令人愉快的事情。

我们可以这样做:

@Component({

selector: 'my-app',

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

})

export class AppComponent {

form = this.formBuilder.group({

firstName: [''],

lastName: [''],

age: [''],

});

controls = {

firstName: this.form.get('firstName'),

lastName: this.form.get('lastName'),

}

constructor(private formBuilder: FormBuilder) {}

}

这非常简单:我们只是将表单控件的引用存储在一个叫做controls的特殊对象中，这样我们就可以很容易地在模板中使用它们，现在看起来就像这样:

<form [formGroup]="form">

<div>

<label for="firstName">First Name</label>

<input formControlName="firstName" id="firstName"/>

<span \*ngIf="controls.lastName.touched && controls.firstName.hasError('required')" class="errors">

Field is required

</span>

</div>

<div>

<label for="lastName">First Name</label>

<input formControlName="lastName" id="lastName"/>

<span \*ngIf="controls.lastName.touched && controls.lastName.hasError('required')" class="errors">

Field is required

</span>

</div>

</form>

乍一看，这似乎是一个微小的改进，但有几点需要考虑:

1 FormControl。Get是一个函数，在模板中调用它意味着它将在每次更改检测周期中被调用(提示:很多)，但通过使用这个方法，我们也解决了这个问题

2 我们只存储将要在模板中引用的控件的引用

3 如果我们在组件类的代码中需要这些控件，我们也可以通过这些引用来访问它们，避免查找

所以，一个问题解决了，但是类型安全呢?我们的控件的值仍然被TypeScript当作any类型来处理，这不是一件很好的事情。那么我们能做些什么呢?

**案例2:使用数据传输对象模式**

想象一下，我们必须填写一个复杂的表单，并通过HTTP请求将其发送给服务器。当然，我们可以使用FormGroup。值了就发送;但最可能的是，服务器API网关等待的是一个略有不同的数据模型。例如，它可能等待ISO格式的日期，而不是标准的JS字符串。或者，它可能希望获得用户所选自定义标记的连接列表，而不是字符串数组。当然，我们可以改变FormGroup的形状，但另一方面，我们在模板中使用的控件可能会更好地处理服务器所期望的数据类型。那么我们能做什么呢?

这就是DTO(数据传输对象)的用途。DTO是一个特殊的对象，它负责将信息从一个子系统(我们的Angular应用)传送到另一个子系统(后端)。它最初的主要目的是减少通过网络发送的数据量。但它也确保子系统使用相同的语言(相同的数据类型)彼此通信。那么我们将如何从这种模式中获益呢?通过编写一个简单的类，该类将在其构造函数中接收表单值，并生成一个与服务器的API签名完全匹配的对象，处理原始表单值与服务器等待的值之间的所有差异。非常重要的是，我们的类没有任何其他行为，getter, setter等等——这些东西不会被JSON序列化。Stringify，我们的类只有一个职责——生成一个数据传输对象。下面是一个DTO的例子:

export class ArticleDTO {

title: string;

tags: string;

date: string;

referenceIds: number[];

constructor(formValue: RawFormValue) {

this.title = formValue.title;

this.tags = formValue.tags.join(',');

this.date = formValue.date.toISOString();

if (formValue.referenceIds && formValue.referenceIds.length > 0) {

this.referenceIds = formValue.referenceIds;

}

}

}

export interface RawFormValue {

title: string;

tags: string[];

date: Date;

referenceIds?: number[];

}

：现在这个类非常简单:它处理一个对象(我们的原始表单值)到另一个对象(将转到后端)的转换。逻辑完全封装在构造函数中。如果有问题，构造函数是整个代码库中可能发生问题的唯一地方;如果类型有问题，我们应该查看RawFormValue接口。下面是我们如何使用这个模式的一个例子:

export class AppComponent {

// rest of the component implementation is ommitted for brevity

submit() {

if (this.form.valid) {

const article = new ArticleDTO(this.form.value as RawFormValue);

// now send the article DTO to the backend using one of your services

}

}

}

这有几个好处:

应用程序的业务逻辑与数据操作和处理被移出组件(它真的不属于那里)

这个逻辑包含在我们的应用中的一个单独的地方，这是唯一负责这个逻辑的地方(更容易捕捉bug)

无论你使用什么数据操作策略，从普通的旧JS对象到状态管理，比如NGRX规范化，这都能很好地发挥作用

**Custom Angular Controls**

关于简化表单，我提到的最后一点是创建自定义控件。在组件中实现ControlValueAccessor接口，并将其注册到NG\_VALUE\_ACCESSOR提供程序中，这样我们就可以创建一个自定义的Angular表单控件。这意味着我们的<自定义组件></自定义组件>现在可以像这样接受FormControl和ngModel了:

<custom-component formControlName="controlName"></custom-component>

这有助于我们抽象出表单上的复杂行为，并使其可重用。当你发现自己在FormGroup的一个控件上做大量的自定义逻辑时，考虑一下是否有可能把它转换成另一个组件。

**不要忘记异步验证器**

我遇到过的最常见的表单错误实践是开发者忘记使用异步验证器:

指令

管道

父组件内部的自定义逻辑，

以执行一些普通的操作，如检查拥有该电子邮件的用户是否存在。重点是:不要忘记它们，在合适的地方使用它们

**setValue vs patchValue**

在Angular中，有两种方法可以改变FormControl的值:setValue和patchValue。它们主要是相同的，但有一个重要的区别:如果我们在一个FormGroup对象上调用setValue，该对象缺少表单签名中的一些键值，它将抛出一个错误。这很有用，因为它允许在响应式表单的动态世界中提供某种类型安全。但它有一个问题:当我们传递给它的对象具有表单没有的属性时，它也会抛出一个错误。所以这会抛出一个错误:

@Component({

selector: 'my-app',

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

})

export class AppComponent {

form = this.formBuilder.group({

firstName: [''],

lastName: [''],

age: [''],

});

ngOnInit() {

// throws an error

this.form.setValue({

firstName: 'Armen',

lastName: 'Vardanyan',

age: 25,

occupation: 'Software developer, writer',

});

}

constructor(private formBuilder: FormBuilder) {}

}

这会抛出一个错误:“error: Cannot find form control with name: occupation”。这是很有用的，但也有一个严重的缺点，我们必须注意。考虑这种情况:

@Component({

selector: 'my-app',

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

})

export class AppComponent {

form = this.formBuilder.group({

firstName: [''],

lastName: [''],

age: [''],

});

async ngOnInit() {

const userData = await this.userService.getUserById(/\*some id\*/);

this.form.setValue(userData);

}

constructor(

private formBuilder: FormBuilder,

private userService: UserService,

) {}

}

因此，在本例中，我们从服务器获取一些数据，并将其设置为表单值——没有什么特别奇怪的。那么问题是什么呢?

目前，一切正常，服务器发送完全相同的数据需要。但是想象一下，我们用来检索用户数据的同一个API调用也在应用的其他地方使用，在一个不是我们构建的组件中。在这里，UI需要更多关于用户的数据——所以API开发人员在响应中添加了一个“职业”字段，并思考“当我们添加了一些内容时，为什么会停止工作?”你猜怎么着?现在它停止工作了，因为有一个新的属性在对象上，我们尝试setValue。我们怎么解决这个问题呢?有两种方法

1 从外部api设置值时使用patchValue。这解决了手头的问题，但可能会在未来产生其他问题——如果API设计有真正的破坏性改变，一些字段在响应中丢失了怎么办?我们希望看到正常的错误消息，而不是看到空白屏幕/输入字段，而patchValue不会抛出错误。

2 完整的解决方案:编写一个中间函数(或者一个类)，它将服务器响应转换为与表单签名兼容的内容(有点像我们将表单值发送给服务器时所做的相反的事情)。下面是一个例子:

interface RawFormValue {

firstName: string;

lastName: string;

age: number;

}

function toRawFormValue<T extends RawFormValue>(serverData: T): RawFormValue {

return {

firstName: serverData.firstName,

lastName: serverData.lastName,

age: serverData.age,

}

}

注意，键入是如何尽可能严格的——函数接收任何包含与表单相同字段的对象，提取这些字段并返回一个只包含这些字段的对象——与我们的代码完全兼容。如果API设计改变了，我们唯一要做的就是改变这个函数。