記事
Toshihiko Minamoto · 2022年11月17日 10m read

Angular 14 の新機能

こんにちは！
Sergei Sakisian と申します。InterSystems で 7 年以上、Angular フロントエンドを作成しています。 Angular は非常に人気のあるフレームワークであるため、開発者、お客様、そしてパートナーの皆さんは、アプリケーションのスタックの 1 つとして Angular を選択することがよくあります。

概念、ハウツー、ベストプラクティス、高度なトピックなど、Angular のさまざまな側面を網羅する記事の連載を始めたいと思います。 この連載は、すでに Angular に精通しており、基本概念の説明がいらない方が対象となります。
連載記事のロードマップを作成しているところであるため、まずは、一番新しい Angular リリースの重要な機能をいくつか紹介することから始めることにします。

厳格な型指定のフォーム

これはおそらく、過去 2 年間で最も要望の多かった Angular 機能です。 Angular 14 では、Angular リアクティブフォームを使って、TypeScript のすべての厳格な型チェック機能を使用できるようになりました。

FormControl クラスはジェネリクスになったため、それが保持する値の型を取ることができます。

```
/* Angular 14 ???*/
const untypedControl = new FormControl(true);
untypedControl.setValue(100); // ??????????

// ??
const strictlyTypedControl = new FormControl<boolean>(true);
strictlyTypedControl.setValue(100); // ????????????????????????

// Angular 14
const strictlyTypedControl = new FormControl(true);
strictlyTypedControl.setValue(100); // ????????????????????????
```

ご覧のとおり、最初の最後の例はほぼ同じですが、結果が異なります。 これは、Angular 14 では、新しい FormControl クラスが、開発者が指定した初期値から型を推論しているためです。 したがって、true が指定された場合、Angular はこの FormControl の型を boolean ¦ null に設定します。 .reset() メソッドには、値が指定されていない場合に値を null にする Nullable 値が必要です。

以前の型なしの FormControl クラスは、UntypedFormControl に変換されています（UntypedFormGroup、UntypedFormArray、および UntypedFormBuilder についても同様）が、実質的に FormControl<any> のエイリアスです。 以前のバージョンの Angular からアップグレードしている場合、FormControl クラスのすべてのメンションは、Angular CLI によって UntypedFormControl クラスに置き換えられます。

Untyped* のクラスは、以下のような特定の目的に使用されます。

1. アプリを、以前のバージョンから移行される前とまったく同じように動作させる（新しい FormControl は、初期値から型を推論することに注意してください）。

2. すべての FormControl<any> を意図的に使用する。 そのため、すべての UntypedFormControl を手動で FormControl<any> に変更する必要があります。
3. 開発者にもっと自由度を与える（これについては、後の方で説明します）。

初期値が null である場合、FormControl の型を明示的に指定する必要があることに注意してください。
また、TypeScript には、初期値が false の場合に同じことを行う必要のあるバグが存在します。

フォームの Group については、インターフェースを定義することも可能です。このインターフェースを FormGroup の型として渡すだけです。 この場合、TypeScript は FormGroup 内のすべての型を推論します。

```
interface LoginForm {
    email: FormControl<string>;
    password?: FormControl<string>;
}

const login = new FormGroup<LoginForm>({
    email: new FormControl('', {nonNullable: true}),
    password: new FormControl('', {nonNullable: true}),
});
```

手動で FormGroup を作成した上記の例のように、FormBuilder の .group()
メソッドに、事前に定義されたインターフェースを受け入れられるジェネリクス属性が追加されました。

```
interface LoginForm {
    email: FormControl<string>;
    password?: FormControl<string>;
}

const fb = new FormBuilder();
const login = fb.group<LoginForm>({
    email: '',
    password: '',
});
```

このインターフェースにはプリミティブな非 nullable 型しかないため、新しい nonNullable FormBuilder プロパティ（NonNullableFormBuilder
クラスインスタンスを含み、直接作成することも可能）を使って以下のように単純化できます。

```
const fb = new FormBuilder();
const login = fb.nonNullable.group({
    email: '',
    password: '',
});
```

非 nullable 型の FormBuilder を使用する場合、または FormControl に非 nullable 型のオプションを設定する場合、.reset() メソッドを呼び出す際に、リセット値として初期の FormControl 値が使用されることに注意してください。

また、this.form.value
のすべてのプロパティがオプションとしてマークされることに注意することも非常に重要です。
以下に例を示します。

```
const fb = new FormBuilder();
const login = fb.nonNullable.group({
    email: '',
    password: '',
});

// login.value
// {
//   email?: string;
//   password?: string;
// }
```

これは、FormGroup 内のいずれかの FormControl を無効にする際に、この FormControl の値が form.value から削除されるために発生します。

```
const fb = new FormBuilder();
const login = fb.nonNullable.group({
    email: '',
    password: '',
});

login.get('email').disable();
console.log(login.value);

// {
//   password: ''
// }
```

フォームオブジェクト全体を取得するには、.getRawValue() メソッドを使用する必要があります。

```
const fb = new FormBuilder();
const login = fb.nonNullable.group({
    email: '',
    password: '',
});

login.get('email').disable();
console.log(login.getRawValue());

// {
//   email: '',
//   password: ''
// }
```

厳格に型付けされたフォームのメリット:

1. FormControl / FormGroup
   の値を返すすべてのプロパティとメソッドが厳格に型付けされるようになった。 例:
   value、getRawValue()、valueChanges
2. FormControl 値を変更するすべてのメソッドが型安全になった。setValue()、patchValue()、updateValue()
3. FormControl が厳格に型付けされた。 このことは、FormGroup の .get() メソッドにも適用されます。
   これにより、コンパイル時に存在しない FormControl へのアクセスも防止されます。

新しい FormRecord クラス

新しい FormGroup クラスの欠点は、その動的な性質が失われたことです。 一度定義されると、オンザフライで FormControl を追加または削除することはできません。

この問題を解決するために、Angular は新たに FormRecord クラスを追加しました。 FormRecord は実質的に FormGroup と同じですが、動的であり、そのすべての FormControl に同じ型が使用されます。

```
folders: new FormRecord({
  home: new FormControl(true, { nonNullable: true }),
  music: new FormControl(false, { nonNullable: true })
});

// ???????? FormContol ?????
this.foldersForm.get('folders').addControl('videos', new FormControl(false, { nonNull
able: true }));

// ??????????????????????????????????
this.foldersForm.get('folders').addControl('books', new FormControl('Some string', {
nonNullable: true }));
```

ご覧のとおり、これには別の制限があります。すべての FormControl は同じ型でなければなりません。
動的と異種の両方を兼ね備えた FormGroup がどうしても必要な場合は、UntypedFormGroup
クラスを使用してフォームを定義することをお勧めします。

モジュールレス（スタンドアロン）コンポーネント

これは未だ実験的とされている機能ではありますが、興味深い機能です。 コンポーネント、ディレクティブ、およびパイプをモジュールに含めることなく、これらを定義することができます。

この概念はまだ完全に練られてはいませんが、すでに ngModule
を使用せずにアプリケーションをビルドすることができるようになっています。

スタンドアロンコンポーネントを定義するには、Component/Pipe/Directive デコレーターで新しい standalone
プロパティを使用する必要があります。

```
@Component({
  selector: 'app-table',
  standalone: true,
  templateUrl: './table.component.html'
})
export class TableComponent {
}
```

この場合、このコンポーネントはどの ngModule にも宣言されませんが、 ngModule
やその他のスタンドアロンコンポーネントにインポートすることは可能です。

各スタンドアロンコンポーネント/パイプ/ディレクティブには、その依存関係を直接デコレーターにインポートするメカニズムが備えられています。

```
@Component({
  standalone: true,
  selector: 'photo-gallery',
```

```
  // ????????????????????????????????
  // CommonModule??*ngIf ?????? Angular ????????????????????????
  // ????????????????????????????????
  imports: [CommonModule, MatButtonModule, TableComponent],
  template: `
    ...
    <button mat-button>Next Page</button>
    <app-table *ngIf="expression"></app-table>
  `,
})
export class PhotoGalleryComponent {
}
```

前述のとおり、スタンドアロンコンポーネントは、既存の ngModule にインポート可能です。 sharedModule 全体をインポートする必要がなく、本当に必要な物だけをインポートできます。
新しいスタンドアロンコンポーネントを使用し始めるのに適したストラテジーでもあります。

```
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, HttpClientModule, TableComponent], // import our standalon
e TableComponent
  bootstrap: [AppComponent]
})
export class AppModule {}
```

スタンドアロンコンポーネントは、Angular CLI を使って以下を入力すると作成できます。

```
ng g component --standalone user
```

モジュールレスアプリケーションをブートストラップ

アプリケーションにあるすべての ngModule を排除する場合は、別の方法でアプリをブートストラップする必要があります。 Angular にはこのための新しい関数があり、それを main.ts ファイルで呼び出す必要があります。

```
bootstrapApplication(AppComponent);
```

この関数の 2 つ目のパラメーターを使って、アプリ全体で必要なプロバイダーを定義できます。
通常プロバイダーのほとんどはモジュール内に存在するため、Angular は（現時点では）それに新しい importProvidersFrom 抽出関数を使用する必要があります。

```
bootstrapApplication(AppComponent, { providers: [importProvidersFrom(HttpClientModule
)] });
```

スタンドアロンコンポーネントの遅延読み込みルート:

Angular には、loadComponent という新しい遅延読み込みルート関数があります。これは、スタンドアロンコンポーネントを読み込むためだけに存在する関数です。

```
{
  path: 'home',
  loadComponent: () => import('./home/home.component').then(m => m.HomeComponent)
}
```

loadChildren は、ngModule を遅延読み込みできるようにするだけでなく、ルートファイルから直接、子ルートも読み込めるようになっています。

```
{
  path: 'home',
  loadChildren: () => import('./home/home.routes').then(c => c.HomeRoutes)
}
```

記事の執筆時点におけるいくつかの注意事項

- スタンドアロンコンポーネント機能は、現在も実験的段階にあります。 将来的に、Webpack の代わりに Vite ビルダーに移行し、ツーリングの改善、ビルド時間の高速化、アプリアーキテクチャの強化、テスト方法の改善などを通じて、機能が大幅に改善されるでしょう。 現時点では、こういったものが多数欠けているため、全パッケージを受け取っていません。いずれにせよ、少なくともこの新しい Angular パラダイムを念頭に、アプリを開発し始めることは可能です。
- IDE と Angular
  ツールはまだ、新しいスタンドアロンエンティティを静的に解析する準備を整えていません。 すべての依存関係を各スタンドアロンエンティティにインポートする必要があるため、何かを見逃した場合、コンパイラーもそれを見逃し、ランタイム時に失敗する可能性があります。
  これは今後改善されていきますが、現時点ではインポートの際に開発者側の注意が必要です。
- 現時点では Angular にグローバルインポート機能がないため（Vue などで行われるように）、各スタンドアロンエンティティで、依存関係を確実に 1 つずつインポートする必要があります。 この機能の主な目標は、私が思うところ、ボイラープレートを減らして物事を簡単に実行できるようにすることにあるため、今後のバージョンで解決されることを期待しています。

#
今日は、これで以上です。 それではまた！

#Angular #Angular2 #UI 開発 #フロントエンド #その他

ソースURL:
https://jp.community.intersystems.com/post/angular-14-%E3%81%AE%E6%96%B0%E6%A9%9F%E8%83%BD