链滴

@delon/form

作者：devcui
原文链接：https://ld246.com/article/1632450903041
来源网站：链滴

delon

@delon/form是delon包中的一个模块，主要提供了angular相关的动态表单的内容，这篇文章是来概的看一下sf组件如何实现的

从SFComponent开始

```
<ng-template #con>
  <ng-content></ng-content>
</ng-template>
<form nz-form [nzLayout]="layout" (submit)="onSubmit($event)"


</form>
```

1-3 定义了一个模版 唯一标识符为con，后面使用了nz-form这是ng-zorro封装下的angular表单，入layout布局和submit表单提交函数

```
<sf-item *ngIf="rootProperty" [formProperty]="rootProperty"></sf-item>
```

在向下是sf-item组件

```
let nextUniqueId = 0;
```

首先是一个不唯一的id，应该是用来区分表单元素名称的

```
@Component({
  selector: 'sf-item',
  exportAs: 'sfItem',
  host: { '[class.sf__item]': 'true' },
  template: ` <ng-template #target></ng-template> `,
  preserveWhitespaces: false,
  encapsulation: ViewEncapsulation.None
})
```

可以看到html仅仅为<ng-template> 并给了一个标识符为target，preserveWhitespaces编译器移所哟不必要的空格选项为false，ViewEncapsulation无 Shadow DOM，并且也无样式包装。

```
export class SFItemComponent implements OnInit, OnChanges, OnDestroy
```

实现了三个生命周期，一个初始化一个销毁，还有一个父组件输入的值发生变化就会调用的OnChangs

```
private ref: ComponentRef<Widget<FormProperty, SFUISchemaItem>>;
  readonly unsubscribe$ = new Subject<void>();
  widget: Widget<FormProperty, SFUISchemaItem> | null = null;

  @Input() formProperty: FormProperty;

  @ViewChild('target', { read: ViewContainerRef, static: true })
  container: ViewContainerRef;
```

ref类型为Widget组件的引用，unsubscribe$负责保存需要取消订阅的值，formProperty为父组件入的值，还有一个container拿到了标识着target视图元素的引用

```
ngOnInit(): void {
    this.terminator.onDestroy.subscribe(() => this.ngOnDestroy());
  }
```

onInit函数订阅了 销毁topic 的信息，当销毁topic的数据发生变化时，触发本sf-item的onDestory数，主要是用来在外部控制 sf-item销毁用的

```
ngOnChanges(): void {
    const p = this.formProperty;
    this.ref = this.widgetFactory.createWidget(this.container, (p.ui.widget || p.schema.type) as st
ing);
    this.onWidgetInstanciated(this.ref.instance);
  }
```

输入值发生变化调用ngOnChanges，在container也就是target标识的容器的地方createWidget将件加载到该位置。然后在控件被创建完成以后，调用onWidgetInstanciated初始化挂载进来的控件

```
onWidgetInstanciated(widget: Widget<FormProperty, SFUISchemaItem>): void {
    this.widget = widget;
    const id = `_sf-${nextUniqueId++}`;

    const ui = this.formProperty.ui as SFUISchemaItem;
    this.widget.formProperty = this.formProperty;
    this.widget.schema = this.formProperty.schema;
    this.widget.ui = ui;
    this.widget.id = id;
    this.widget.firstVisual = ui.firstVisual as boolean;
    this.formProperty.widget = widget;
  }
```

主要还是得给一个唯一id和schema

```
ngOnDestroy(): void {
    const { unsubscribe$ } = this;
    unsubscribe$.next();
    unsubscribe$.complete();
    this.ref.destroy();
  }
```

sf-item负责渲然表单控件，这就是动态表单的入口了，其中比较重要的函数是

```
this.widgetFactory.createWidget(this.container, (p.ui.widget || p.schema.type) as string);
```

通过json对象，也就是formProperty来动态创建控件，稍后再说，回到sf下看后面写了些什么

```
<ng-container *ngIf="button !== 'none'; else con">
    <nz-form-item
      *ngIf="_btn.render"
      [ngClass]="_btn.render!.class!"
      class="sf-btns"
      [fixed-label]="_btn.render!.spanLabelFixed!"
    >
      <div
        nz-col
        class="ant-form-item-control"
```

```
  [nzSpan]="btnGrid.span"
  [nzOffset]="btnGrid.offset"
  [nzXs]="btnGrid.xs"
  [nzSm]="btnGrid.sm"
  [nzMd]="btnGrid.md"
  [nzLg]="btnGrid.lg"
  [nzXl]="btnGrid.xl"
  [nzXXl]="btnGrid.xxl"
>
  <div class="ant-form-item-control-input">
    <div class="ant-form-item-control-input-content">
      <ng-container *ngIf="button; else con">
        <button
          type="submit"
          nz-button
          data-type="submit"
          [nzType]="_btn.submit_type!"
          [nzSize]="_btn.render!.size!"
          [nzLoading]="loading"
          [disabled]="liveValidate && !valid"
        >
          <i
            *ngIf="_btn.submit_icon"
            nz-icon
            [nzType]="_btn.submit_icon.type!"
            [nzTheme]="_btn.submit_icon.theme!"
            [nzTwotoneColor]="_btn.submit_icon.twoToneColor!"
            [nzIconfont]="_btn.submit_icon.iconfont!"
          ></i>
          {{ _btn.submit }}
        </button>
        <button
          *ngIf="_btn.reset"
          type="button"
          nz-button
          data-type="reset"
          [nzType]="_btn.reset_type!"
          [nzSize]="_btn.render!.size!"
          [disabled]="loading"
          (click)="reset(true)"
        >
          <i
            *ngIf="_btn.reset_icon"
            nz-icon
            [nzType]="_btn.reset_icon.type!"
            [nzTheme]="_btn.reset_icon.theme!"
            [nzTwotoneColor]="_btn.reset_icon.twoToneColor!"
            [nzIconfont]="_btn.reset_icon.iconfont!"
          ></i>
          {{ _btn.reset }}
        </button>
      </ng-container>
    </div>
  </div>
```

```
    </div>
  </nz-form-item>
</ng-container>
```

这里写了两个button，保存/重置，假如button参数不为空，那么直接显示两个按钮，假如button参为空，那么con标识的template，其实，这么长，只是在控制按钮显示/隐藏

在看Widget之前，有几个抽象类和接口需要提前看一下

```
export abstract class FormProperty {}
```

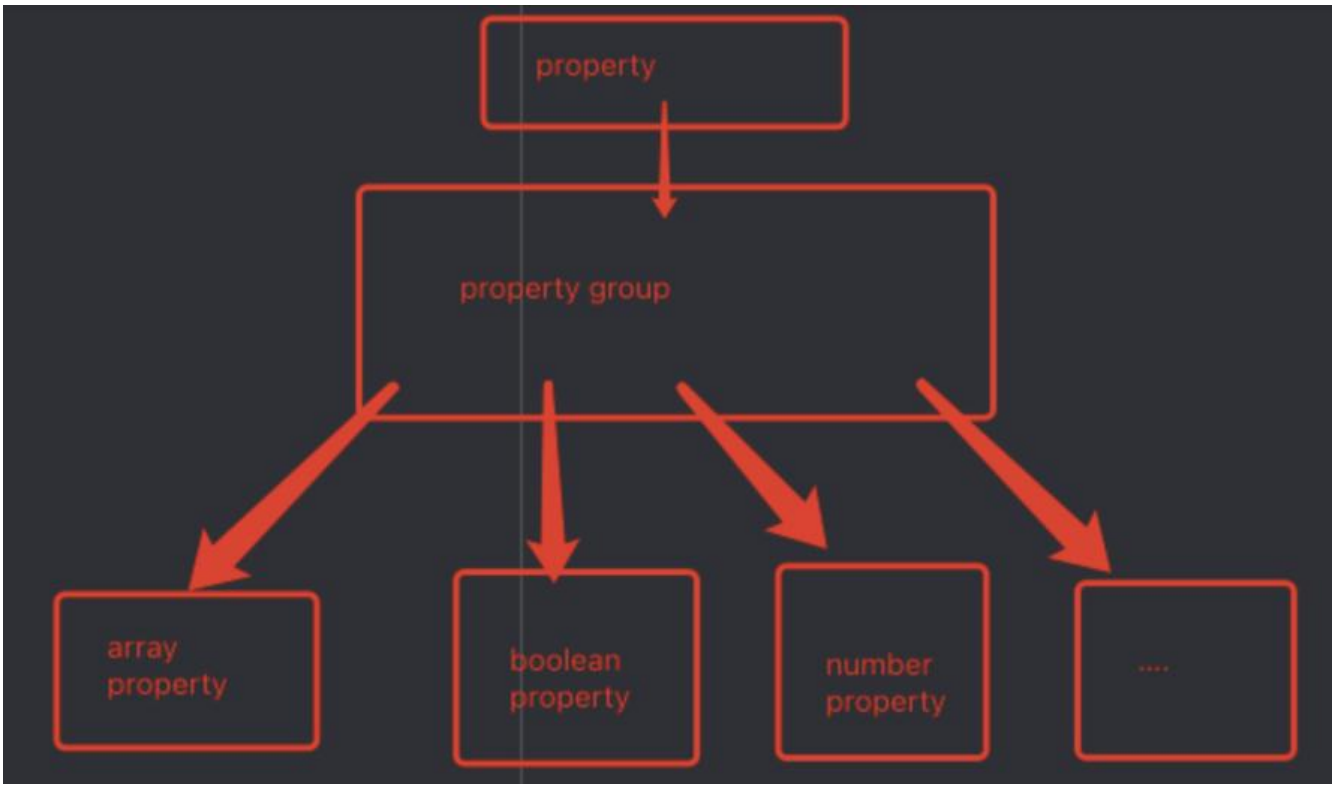
formProperty封装了一些空间的属性

```
private _errors: ErrorData[] | null = null;
  private _valueChanges = new BehaviorSubject<SFFormValueChange>({ path: null, pathValue:
null, value: null });
  private _errorsChanges = new BehaviorSubject<ErrorData[] | null>(null);
  private _visible = true;
  private _visibilityChanges = new BehaviorSubject<boolean>(true);
  private _root: PropertyGroup;
  private _parent: PropertyGroup | null;
  _objErrors: { [key: string]: ErrorData[] } = {};
  schemaValidator: (value: SFValue) => ErrorData[];
  schema: SFSchema;
  ui: SFUISchema | SFUISchemaItemRun;
  formData: Record<string, unknown>;
  _value: SFValue = null;
  widget: Widget<FormProperty, SFUISchemaItem>;
  path: string;
```

错误提示，值变更，错误变更，是否可见，schema信息，ui信息，formData数据等。。。。

```
export abstract class PropertyGroup extends FormProperty {
  properties: { [key: string]: FormProperty } | FormProperty[] | null = null;
```

可以看出PropertyGroup和FormProperty实际上就是在扩展JSONSchema，是一个树形结构，与此时，JSONSchema的几个基础数据类型也都继承了PropertyGroup以下为图示，这俩class扩展了JSO Schema



好，下面在看一下FormPropertyFactory核心的函数是createProperty

```
createProperty(
    // 接收jsonSchema
    schema: SFSchema,
    // 接收uiSchema
    ui: SFUISchema | SFUISchemaItem,
    // 数据
    formData: Record<string, unknown>,
    // 父节点是谁
    parent: PropertyGroup | null = null,
    // property名称
    propertyId?: string
  ): FormProperty {
    let newProperty: FormProperty | null = null;
// 当前这个元素在json里的路径是什么，就好像deepGet(path)的这个path一样
    let path = '';
    if (parent) {
    // 首先把父节点的属性路径加上
      path += parent.path;
      if (parent.parent !== null) {
        path += SF_SEQ;
      }
// 看看是object还是array
      switch (parent.type) {
        case 'object':
    // 如果是object，那么直接附加属性id
          path += propertyId;
          break;
        case 'array':
    // 如果是array，那加入数组的长度
```

```
          path += ((parent as ArrayProperty).properties as PropertyGroup[]).length;
          break;
        default:
          throw new Error(`Instanciation of a FormProperty with an unknown parent type: ${paren
.type}`);
      }
    } else {
      path = SF_SEQ;
    }
// JOSONSchema引用类型是可以引用预先定义好的其他JSON的，这里就是处理ref类型的json
    if (schema.$ref) {
      const refSchema = retrieveSchema(schema, parent!.root.schema.definitions);
      newProperty = this.createProperty(refSchema, ui, formData, parent, path);
    } else {
      // fix required
// 通过判断JSONSchema的required列表控制ui是否是必填类型
      if (
        (propertyId && parent!.schema.required!.indexOf(propertyId.split(SF_SEQ).pop()!) !== -1)
||
        ui.showRequired === true
      ) {
        ui._required = true;
      }
      // fix title
// 如果没有title那么默认使用自增id
      if (schema.title == null) {
        schema.title = propertyId;
      }
      // fix date
// 当json的类型为string和number时，需要对数据进行判断，如果json下的uiJSON中传入的widget
时间/日期类型那么需要做转换
      if ((schema.type === 'string' || schema.type === 'number') && !schema.format && !(ui as
SFUISchemaItem).format) {
        if ((ui as SFUISchemaItem).widget === 'date')
          ui._format = schema.type === 'string' ? this.options.uiDateStringFormat : this.options.u
DateNumberFormat;
        else if ((ui as SFUISchemaItem).widget === 'time')
          ui._format = schema.type === 'string' ? this.options.uiTimeStringFormat : this.options.u
TimeNumberFormat;
      } else {
        ui._format = ui.format;
      }
// 然后是通过jsonSchema的具体类型来分发创建不同类型的Property
      switch (schema.type) {
        case 'integer':
        case 'number':
          newProperty = new NumberProperty(
            this.schemaValidatorFactory,
            schema,
            ui,
            formData,
            parent,
            path,
            this.options
```

```
        );
        break;
      case 'string':
        newProperty = new StringProperty(
          this.schemaValidatorFactory,
          schema,
          ui,
          formData,
          parent,
          path,
          this.options
        );
        break;
      case 'boolean':
        newProperty = new BooleanProperty(
          this.schemaValidatorFactory,
          schema,
          ui,
          formData,
          parent,
          path,
          this.options
        );
        break;
      case 'object':
        newProperty = new ObjectProperty(
          this,
          this.schemaValidatorFactory,
          schema,
          ui,
          formData,
          parent,
          path,
          this.options
        );
        break;
      case 'array':
        newProperty = new ArrayProperty(
          this,
          this.schemaValidatorFactory,
          schema,
          ui,
          formData,
          parent,
          path,
          this.options
        );
        break;
      default:
        throw new TypeError(`Undefined type ${schema.type}`);
    }
  }

// 最后调用`initializeRoot`加载json
```

```
    if (newProperty instanceof PropertyGroup) {
      this.initializeRoot(newProperty);
    }

    return newProperty;
  }
```

这个函数只做了一件事，就是设置可见状态了

```
private initializeRoot(rootProperty: PropertyGroup): void {
    // rootProperty.init();
    rootProperty._bindVisibility();
  }
```

property相关的就是拿到jsonSchema做一些处理，赋值给扩展属性。真正渲染的时候依靠的是对Widgets的判断。

```
@Directive()
export abstract class Widget<T extends FormProperty, UIT extends SFUISchemaItem> imple
ents AfterViewInit {
  formProperty: T;
  error: string;
  showError = false;
  id = '';
  schema: SFSchema;
  ui: UIT;
  firstVisual = false;

  @HostBinding('class')
  get cls(): NgClassType {
    return this.ui.class || '';
  }

  get disabled(): boolean {
    if (this.schema.readOnly === true || this.sfComp!.disabled) {
      return true;
    }

    return false;
  }

  get l(): LocaleData {
    return this.formProperty.root.widget.sfComp!.locale;
  }

  get oh(): SFOptionalHelp {
    return this.ui.optionalHelp as SFOptionalHelp;
  }

  get dom(): DomSanitizer {
    return this.injector.get(DomSanitizer);
  }

  get cleanValue(): boolean {
```

```
    return this.sfComp?.cleanValue!;
  }

  constructor(
    @Inject(ChangeDetectorRef) public readonly cd: ChangeDetectorRef,
    @Inject(Injector) public readonly injector: Injector,
    @Inject(SFItemComponent) public readonly sfItemComp?: SFItemComponent,
    @Inject(SFComponent) public readonly sfComp?: SFComponent
  ) {}

  ngAfterViewInit(): void {
    this.formProperty.errorsChanges
      .pipe(takeUntil(this.sfItemComp!.unsubscribe$))
      .subscribe((errors: ErrorData[] | null) => {
        if (errors == null) return;
        di(this.ui, 'errorsChanges', this.formProperty.path, errors);

        // 不显示首次校验视觉
        if (this.firstVisual) {
          this.showError = errors.length > 0;
          this.error = this.showError ? (errors[0].message as string) : '';

          this.cd.detectChanges();
        }
        this.firstVisual = true;
      });
    this.afterViewInit();
  }

  setValue(value: SFValue): void {
    this.formProperty.setValue(value, false);
    di(this.ui, 'valueChanges', this.formProperty.path, this.formProperty);
  }

  get value(): NzSafeAny {
    return this.formProperty.value;
  }

  detectChanges(onlySelf: boolean = false): void {
    if (onlySelf) {
      this.cd.markForCheck();
    } else {
      this.formProperty.root.widget.cd.markForCheck();
    }
  }

  abstract reset(value: SFValue): void;

  abstract afterViewInit(): void;
}
```

可以看到，每个Widget都要传入范型，T为继承自FormProperty的,UIT继承自SFUISchemaItem，样就将不同的控件和不同的属性(JSONSchema)结合了起来，并赋值给了fromProperty,ui这两个属，注意，这里标注的是@Directive是一个angular指令

继续向下看

```
@Directive()
export class ControlWidget extends Widget<FormProperty, SFUISchemaItem> {
  reset(_value: SFValue): void {}
  afterViewInit(): void {}
}

@Directive()
export class ControlUIWidget<UIT extends SFUISchemaItem> extends Widget<FormProperty,
UIT> {
  reset(_value: SFValue): void {}
  afterViewInit(): void {}
}

@Directive()
export class ArrayLayoutWidget extends Widget<ArrayProperty, SFArrayWidgetSchema> imp
ements AfterViewInit {
  reset(_value: SFValue): void {}
  afterViewInit(): void {}

  ngAfterViewInit(): void {
    this.formProperty.errorsChanges
      .pipe(takeUntil(this.sfItemComp!.unsubscribe$))
      .subscribe(() => this.cd.detectChanges());
  }
}

@Directive()
export class ObjectLayoutWidget extends Widget<ObjectProperty, SFObjectWidgetSchema>
mplements AfterViewInit {
  reset(_value: SFValue): void {}
  afterViewInit(): void {}

  ngAfterViewInit(): void {
    this.formProperty.errorsChanges
      .pipe(takeUntil(this.sfItemComp!.unsubscribe$))
      .subscribe(() => this.cd.detectChanges());
  }
}
```

抽象类型的Widget具体衍生出了以上几种类型，ControlWidget,ControlUIWidget,ArrayLayoutWi
get,ObjectLayoutWidget

一种是常规控件，第二种是可扩展UI的控件，第三种是数组控件，第四种是对象控件

以上是对Widget控件层面的一些类型，创建控件还需要使用WidgetFactory来进行真正的创建

```
export class WidgetRegistry {
  private _widgets: { [type: string]: Widget<FormProperty, SFUISchemaItem> } = {};

  private defaultWidget: Widget<FormProperty, SFUISchemaItem>;

  get widgets(): { [type: string]: Widget<FormProperty, SFUISchemaItem> } {
    return this._widgets;
```

```
  }

  setDefault(widget: NzSafeAny): void {
    this.defaultWidget = widget;
  }

  register(type: string, widget: NzSafeAny): void {
    this._widgets[type] = widget;
  }

  has(type: string): boolean {
    return this._widgets.hasOwnProperty(type);
  }

  getType(type: string): Widget<FormProperty, SFUISchemaItem> {
    if (this.has(type)) {
      return this._widgets[type];
    }
    return this.defaultWidget;
  }
```

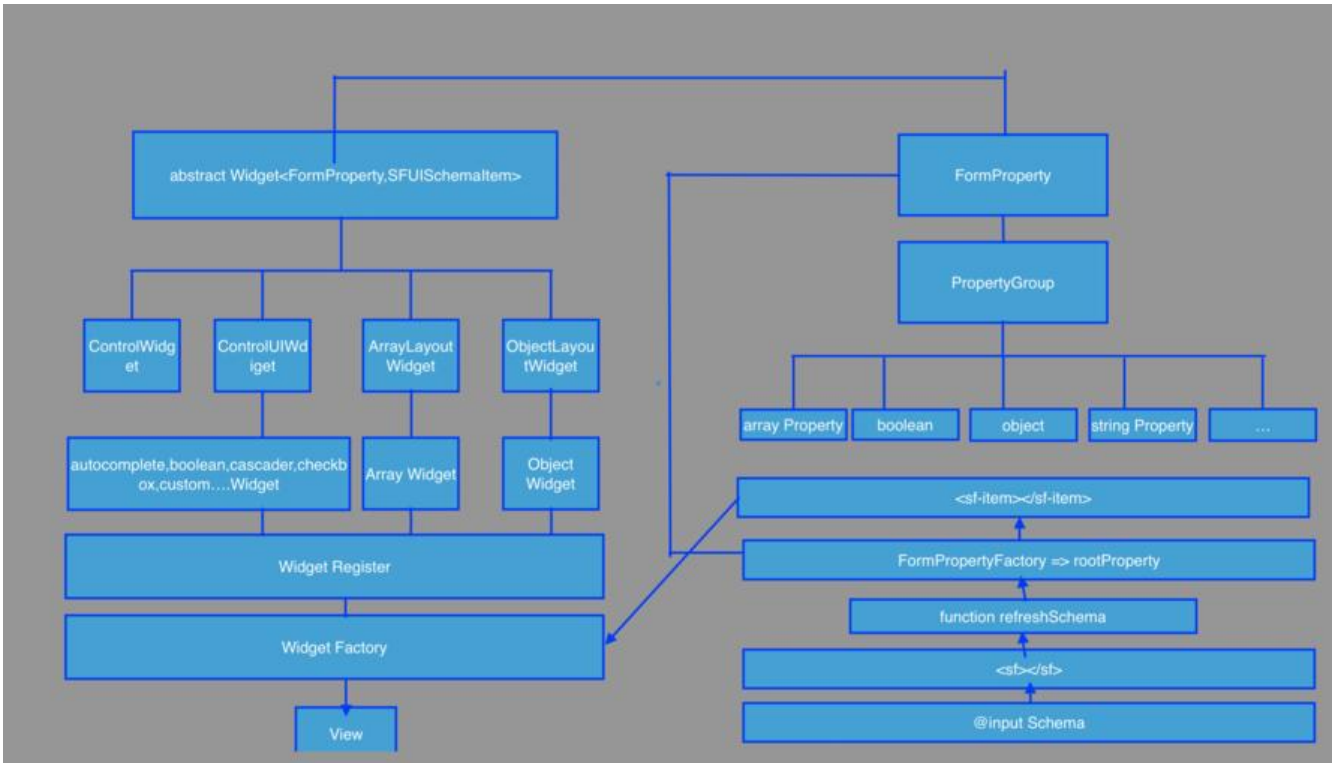
上面是Widget注册表类，存储了一份字符串->控件的映射，这里的字符串主要是用来对应JSONSchema的type字段，表示 json中的type将要渲染成哪个具体的组件，映射关系在nz-widget.registry.ts中

```
export class NzWidgetRegistry extends WidgetRegistry {
  constructor() {
    super();

    this.register('object', ObjectWidget);
    this.register('array', ArrayWidget);

    this.register('text', TextWidget);
    this.register('string', StringWidget);
    this.register('number', NumberWidget);
    this.register('integer', NumberWidget);
    this.register('date', DateWidget);
    this.register('time', TimeWidget);
    this.register('radio', RadioWidget);
    this.register('checkbox', CheckboxWidget);
    this.register('boolean', BooleanWidget);
    this.register('textarea', TextareaWidget);
    this.register('select', SelectWidget);
    this.register('tree-select', TreeSelectWidget);
    this.register('tag', TagWidget);
    this.register('upload', UploadWidget);
    this.register('transfer', TransferWidget);
    this.register('slider', SliderWidget);
    this.register('rate', RateWidget);
    this.register('autocomplete', AutoCompleteWidget);
    this.register('cascader', CascaderWidget);
    this.register('mention', MentionWidget);
    this.register('custom', CustomWidget);

    this.setDefault(StringWidget);
```

```
  }
}

@Injectable()
export class WidgetFactory {
  constructor(private registry: WidgetRegistry, private resolver: ComponentFactoryResolver) {}

  createWidget(container: ViewContainerRef, type: string): ComponentRef<Widget<FormProp
rty, SFUISchemaItem>> {
    if (!this.registry.has(type)) {
      console.warn(`No widget for type "${type}"`);
    }

    const componentClass = this.registry.getType(type) as NzSafeAny;
    const componentFactory =
      this.resolver.resolveComponentFactory<Widget<FormProperty, SFUISchemaItem>>(com
onentClass);
    return container.createComponent(componentFactory);
  }
}
```

通过createWidget从注册表中检索Widget，进而将具体的Component刷到Container容器中，完成件的渲染

再次回到SFComponent



图中不涉及到UI的JSON，因为没有地方画了

从sf的 onInit开始看，可以看到最终调用了refreshSchema重新构建出了rootProperties此时json对已经被扩展了，然后 sf-item通过检测到json变化从而调用widgetFactory将json.type类型的控件create出来

自定义Widget

需要继承，通常是继承ControlUIWidget实现其函数，并且注入WidgetRegister将type作为key注册注册表里即可