Angular

2、DI 注入方式

1、Angular 版本

- AngularJS v1.0 v2.0 - 底层是通过 JavaScript 语法构成的模块化项目，类似vue
- Angular V4.0 v5.0 …… - 底层是通过 typeScript 语法构成的模块化项目

2、构建Angular的运行环境

- 对于基于TS语法的angular进行学习时，必须从模块化项目构建开始，完成了模块化项目构建时，才可以构建出一个编译TS的环境
- angular的模块化项目不能高度的自定义化，构建的angular项目必然是一个 完整功能的项目
  - vue框架是一个渐进式框=可以在开始时根据需要自行扩展功能，学习的时可以根据学习的时间，进阶式的学习语法
  - angular TS语法的框架=在没有angular的模块化环境时，是无法运行；angular将开发中可能用到的功能统一进行核心语法的封装，开发时可以选择性的使用，但在构建模块化项目功能会被统一安装

1、通过全局的angular项目脚手架进行项目构建

- 全局安装：`[sudo] npm install @angular/cli -g` ==> 为node环境增加全局的功能和命令
  - 以 @ 模块包，表示的是 node 环境下以 TS 语法作为基础的 功能扩展包 = 必须依赖于typescript模块
- 在当前的系统环境下，提供一个用于完成 angular 操作的命令前缀 `ng`
  - `ng --version` 查看脚手架的版本
  - `ng help` 查看帮助手册

主命令手册

```
add 为项目添加依赖
analytics 配置angular脚手架功能
build (b) 编译打包项目     ==>  ng build   ng b
```

```
  config 修改angular项目的相关配置
  doc (d) 查看帮助手册
  e2e (e) 启动端到端测试
  generate (g) 项目构成文件的添加和创建  !!!
  help 查看帮助列表
  lint (l) 运行语法测试
  new (n) 创建新项目  ==> ng new 项目名称
  run 基于angular的构建配置项进行项目启动测试
  serve (s) 开启热更新开发服务器
  test (t) 运行项目测试工具
  update 更新项目依赖环境
  version (v) 查看脚手架版本  ng --version
  xi18n 国际化
```

- **For more detailed help run "ng [command name] --help" 查看 一个主命令 的 子命令手册**
  - 子命令手册的查看必须在 一个完成 angular 项目目录下才能执行

2、通过 angular-cli 构建 angular项目

- 使用 主命令 `ng new 项目名称`
  - 1、确定项目的存放目录
  - 2、通过ng new 构建项目

> mac下如果因为权限无法安装依赖，可以执行 sudo chown -R 501:20 "/Users/appleuser/.npm"

```
ITANY-IMAC-190:angular appleuser$ ng new first-project
? Would you like to add Angular routing? No  # 是否安装路由
? Which stylesheet format would you like to use? (Use arrow
keys) # 选择项目的样式语言
› CSS
  SCSS   [ http://sass-
lang.com/documentation/file.SASS_REFERENCE.html#syntax ]
  Sass   [ http://sass-
lang.com/documentation/file.INDENTED_SYNTAX.html       ]
  Less   [ http://lesscss.org
                ]
  Stylus [ http://stylus-lang.com
                ]
```

```
  # 直接构成项目文件结构
CREATE first-project/README.md (1029 bytes)
CREATE first-project/.editorconfig (246 bytes)
CREATE first-project/.gitignore (629 bytes)
CREATE first-project/angular.json (3455 bytes)
CREATE first-project/package.json (1286 bytes)
CREATE first-project/tsconfig.json (438 bytes)
CREATE first-project/tslint.json (1985 bytes)
CREATE first-project/browserslist (429 bytes)
CREATE first-project/karma.conf.js (1025 bytes)
CREATE first-project/tsconfig.app.json (210 bytes)
CREATE first-project/tsconfig.spec.json (270 bytes)
CREATE first-project/src/favicon.ico (5430 bytes)
CREATE first-project/src/index.html (299 bytes)
CREATE first-project/src/main.ts (372 bytes)
CREATE first-project/src/polyfills.ts (2838 bytes)
CREATE first-project/src/styles.css (80 bytes)
CREATE first-project/src/test.ts (642 bytes)
CREATE first-project/src/assets/.gitkeep (0 bytes)
CREATE first-project/src/environments/environment.prod.ts (51
bytes)
CREATE first-project/src/environments/environment.ts (662
bytes)
CREATE first-project/src/app/app.module.ts (314 bytes)
CREATE first-project/src/app/app.component.css (0 bytes)
CREATE first-project/src/app/app.component.html (1120 bytes)
CREATE first-project/src/app/app.component.spec.ts (999 bytes)
CREATE first-project/src/app/app.component.ts (217 bytes)
CREATE first-project/e2e/protractor.conf.js (810 bytes)
CREATE first-project/e2e/tsconfig.json (214 bytes)
CREATE first-project/e2e/src/app.e2e-spec.ts (642 bytes)
CREATE first-project/e2e/src/app.po.ts (251 bytes)

# 直接进行项目依赖安装 ( 500M  == 确认下 npm 的源地址是否已经切换 )
⸨ ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ ⸩ ⁝ fetchMetadata: sill pacote range
manifest for is-glob@^4.0.0 fetched in 306
```

2、启动项目

- 在项目目录下执行 `ng serve` 项目启动

```
ITANY-IMAC-190:first-project appleuser$ ng serve


Date: 2019-06-03T08:34:46.359Z
Hash: c34cb7022d26205ab4bb
Time: 16326ms
chunk {main} main.js, main.js.map (main) 9.81 kB [initial]
[rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills)
248 kB [initial] [rendered]
chunk {polyfills-es5} polyfills-es5.js, polyfills-es5.js.map
(polyfills-es5) 380 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.08 kB
[entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 16.3 kB
[initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.67 MB
[initial] [rendered]
** Angular Live Development Server is listening on
localhost:4200, open your browser on http://localhost:4200/  #
描述项目的访问端口**
ℹ 「wdm」: Compiled successfully.
```

3、项目结构和启动流程

1、项目结构

2、angular项目的启动流程

- 1、控制台执行 `ng serve` ， 读取了当前启动命令所在目录的 `angular.json` 文件启动项目
  - 当文件夹下不存在 angular.json 文件时，将提示错误 ： **The serve command requires to be run in an Angular project, but a project definition could not be found.**
  - angular.json 描述angular项目在启动时必须依赖的相关资源和配置，项目的默认配置的相关文件
- 2、通过 angular.json 文件 启动了项目的 **两个重要的引导文件 index.html main.ts**
- 3、程序运行文件的加载 index.html main.ts
  - index.html 项目启动后的 用户访问的主体页面
  - main.ts 整个模块化项目的代码的加载启动文件
- 4、main.ts 在完成 语法环境构建后，将程序的启动 引导到了 项目的 主体管理文件 app.module.ts

angular.json 文件的构成

```
{
    // 当前配置文件的 定义约束文件
  "$schema":
"./node_modules/@angular/cli/lib/config/schema.json",
    // 当前项目版本
    //   版本构建   主版本号.副版本号.修正版本号
```

```
    //        主板版本号：软件的大版本 == 功能发生变换==新功能添加 == 功能的兼容性文件
    //        副版本号： 软件的小版本 == 功能发生变换==新功能添加功能实现修改
    //        修正版本号:bug的修改
  "version": 1,
    // 引导程序运行 读取配置文件时 ，找到项目的配置选项
  "newProjectRoot": "projects",
  "projects": {
      // 项目的配置文件
    "first-project": {
        // 项目类型描述
      "projectType": "application",
        // 动态语言描述配置文件 ？
      "schematics": {},
        // 项目的根目录配置文件
      "root": "",
        // 开发时编写代码的根目录 == 直接影响后续的代码定义路径
      "sourceRoot": "src",
        // 项目文件前缀  ？
      "prefix": "app",
        // 项目控制执行的命令的 启动引导配置
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
              // 项目打包后的文件输出目录
            "outputPath": "dist/first-project",
              // 页面的主体引导文件
            "index": "src/index.html",
              // 项目打包和启动时 功能引导
            "main": "src/main.ts",
              // zonejs文件的加载
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
              // 静态资源配置文件
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
              // 全局样式配置文件
```

```
            "styles": [
              "src/styles.css"
            ],
              // 全局JS配置文件
            "scripts": []
          },
          "configurations": {
              //产品模式的配置
            "production": {
              "fileReplacements": [
                {
                    // environment.ts  替换为
environment.prod.ts   ?
                  "replace":
"src/environments/environment.ts",
                  "with":
"src/environments/environment.prod.ts"
                }
              ],
              "optimization": true,
              "outputHashing": "all",
              "sourceMap": false,
              "extractCss": true,
              "namedChunks": false,
              "aot": true,
              "extractLicenses": true,
              "vendorChunk": false,
              "buildOptimizer": true,
              "budgets": [
                {
                  "type": "initial",
                  "maximumWarning": "2mb",
                  "maximumError": "5mb"
                }
              ]
            }
          }
        },
          // 开发服务器的启动
        "serve": {
```

```
          "builder": "@angular-devkit/build-angular:dev-
server",
          "options": {
              // serve的启动方式和 打包使用相同配置
            "browserTarget": "first-project:build"
          },
          "configurations": {
            "production": {
              "browserTarget": "first-
project:build:production"
            }
          }
        },
          // 国际化启动方式
        "extract-i18n": {
          "builder": "@angular-devkit/build-angular:extract-
i18n",
          "options": {
            "browserTarget": "first-project:build"
          }
        },
          // 单元测试配置项
        "test": {
          "builder": "@angular-devkit/build-angular:karma",
          "options": {
              // 单元测试的主要程序入口
            "main": "src/test.ts",
              // zone.js 文件的读取
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.spec.json",
            "karmaConfig": "karma.conf.js",
              // 项目的静态资源配置
            "assets": [
              "src/favicon.ico",
              "src/assets",
              "src/css",
              "src/js"
            ],
              // 全局样式定义文件==定义多个文件==多个全局样式文件以
定义顺序存在优先级
            "styles": [
```

```
              "src/styles.css"
            ],
              // 全局JS文件
            "scripts": []
          }
        },
          // 语法校验的启动配置文件
        "lint": {
          "builder": "@angular-devkit/build-angular:tslint",
          "options": {
            "tsConfig": [
              "tsconfig.app.json",
              "tsconfig.spec.json",
              "e2e/tsconfig.json"
            ],
            "exclude": [
              "**/node_modules/**"
            ]
          }
        },
          // 端到端的启动配置文件
        "e2e": {
          "builder": "@angular-devkit/build-
angular:protractor",
          "options": {
            "protractorConfig": "e2e/protractor.conf.js",
            "devServerTarget": "first-project:serve"
          },
          "configurations": {
            "production": {
              "devServerTarget": "first-
project:serve:production"
            }
          }
        }
      }
    }},
    // 引导程序读取配置中的 项目配置文件
  "defaultProject": "first-project"
}
```

- index.html

```
<!doctype html>
<html lang="zh-cn">

<head>
    <meta charset="utf-8">
    <title>FirstProject</title>
    <!-- 指定用于一个文档中包含的所有相对 URL 的根 URL。 -->
    <base href="/">

    <meta name="viewport" content="width=device-width,
initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>

<body>

    <app-root></app-root>

</body>

</html>
```

- main.ts

```
/*
  构成启动项目的 服务器环境，打包项目时的 打包环境
*/

import { enableProdMode } from '@angular/core'; // 从angular核
心语法包中加载 enableProdMode方法 ，控制启动项目的产品模式
import { platformBrowserDynamic } from '@angular/platform-
browser-dynamic'; // 程序构成的 语法环境的引导对象 - 启动TS到浏览器
兼容JS的转换规则

// 加载了 app 目录下 app.module.ts  , angular 项目的 管理对象模块
import { AppModule } from './app/app.module';
// 加载 启动环境的配置项
import { environment } from './environments/environment';

// 以配置项的方式 判断程序以 何种方式启动
```

```
if (environment.production) {
  enableProdMode();
}

// platformBrowserDynamic 完成语法引导对象构建时，返回一个程序启动的引导对象
platformBrowserDynamic()
  // bootstrapModule方法 加载程序的 启动引导文件（对象）
  .bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

- app.module.ts

```
/*
  angular 项目管理文件
    管理了整个项目中的所有应用资源
*/

// 加载浏览器的 兼容切换模块和 浏览器语法运行模块
import { BrowserModule } from '@angular/platform-browser';
// 从angular的核心语法中 加载 NgModule 的 装饰器接口
import { NgModule } from '@angular/core';
// 加载的是 app.component.ts ==> 提供是一个 angular的组件
import { AppComponent } from './app.component';

/**
 * @NgModule 装饰器
 *   TS 和 ES6 中都存在该语法
 *   为指定的 类(构造函数) 或者 方法和属性 ，进行功能的扩展
 *   在对应的属性和方法以及 类（构造器） 上进行功能添加
 *
 * angular 项目中 @NgModule 描述当前的 class 类型为 angular项目的中央管理对象
 *    管理了整个angular项目中的 所有的组件 方法 ……
 */
@NgModule({
  // 描述 angular 项目运行时，所需要的组件 方法 过滤器，指令等等
  declarations: [
    AppComponent
  ],
```

```
  // 为项目提供 功能模块的加载 ，表单功能（包含双向数据操作），浏览器兼容，ajax，路由功能……
  imports: [
    BrowserModule
  ],
  // DI 提供者 ？DI=依赖注入 ？？？
  providers: [],
  // 定义项目启动时 angular的引导组件 == Root组件 == 根组件
  bootstrap: [AppComponent]
})
export class AppModule {
  // 导出的是一个 名称叫做 AppModule 的普通的空的 TS class
}
```

4、angular中的组件

- 组件构成
  - 页面：由独立HTML文件作为模板
  - 样式：由独立的css或动态语言文件作为 局部样式文件
  - 功能：由独立的 TS 文件做 功能文件
- angular项目中，随着项目的开发，项目中的组件会越来越多，程序如何区分 组件文件的关系
  - 1、对于TS语法执行 `import { AppComponent } from './app.component';` 语法时，没有文件后缀，默认加载 .ts结尾的文件
  - 2、整个组件文件有程序读取 .ts结尾的组件功能文件，完成组件的定义和加载
  - 3、通过组件定义的方式，指定了当前组件对应的 页面和 局部样式

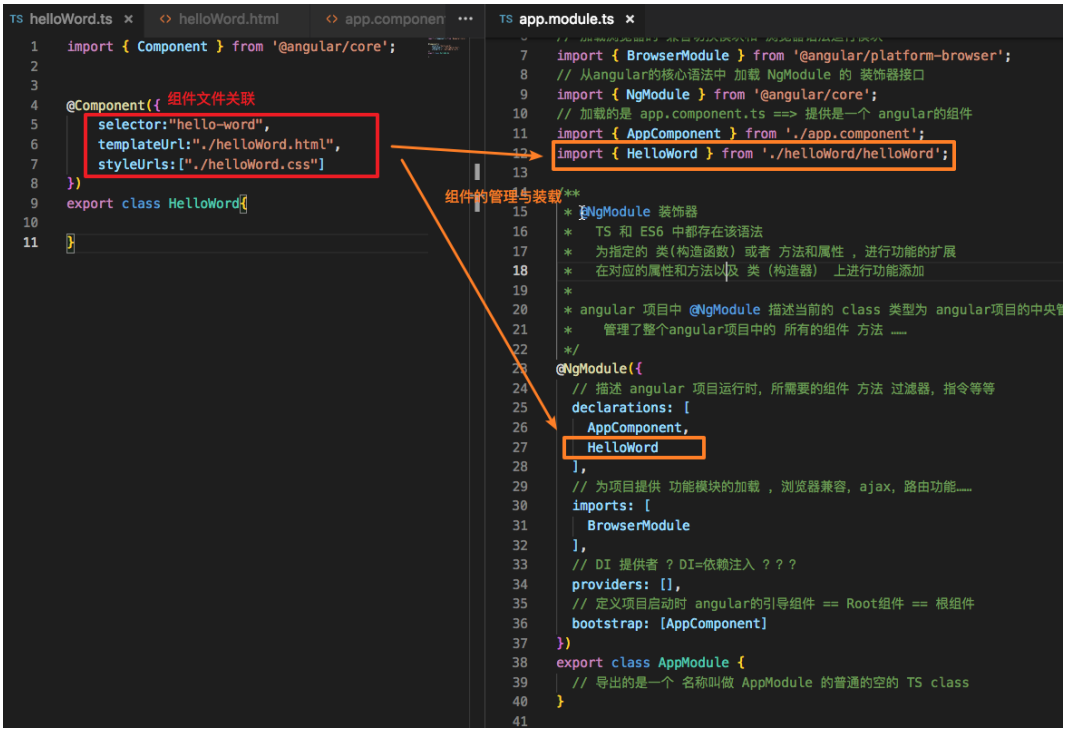
```
/*
  构建一个 名称叫做 app.component 的组件
*/
// @angular的模块中的 core 中 ，加载了一个 装饰Component
// @Component(options) 完成angular 组件类的功能添加
//    options 完成组件的名称定义 样式定义 页面模板定义
import { Component } from '@angular/core';

@Component({
```

```
  // EL == 定义当前组件 在页面使用时的 标签名称
  selector: 'app-root', // <app-root></app-root>

  // 组件的模板 取值为 templateUrl 还是 template 取决于 定义顺序

  // templateUrl 描述当前 组件对应的页面模板
  //     组件的模板只能定义一个
  templateUrl: './app.component.html',
  // template 可以直接以 string 描述组件的模板
  // template:"<h1>asdadsa</h1>",

  // angular的组件定义中 样式文件时可有可无

  // 局部样式文件的 加载
  // styleUrls 为一个组件定义多个局部样式文件
  styleUrls: ['./app.component.css'],
  // 通过 styles 配置string方式的 局部样式
  // styles:["color:red;"]
})
// 组件的功能构成就是一个 普通的 TS 的class类（JS中的构造函数）
export class AppComponent {
  title = 'first-project';
}
```

1、手动组件的定义

- 1、创建三个文件 `组件.html 组件.css 组件.ts` 构成组件
- **2、中央管理器项目，组件是否可用，取决于app.module.ts文件中 是否管理了该组件**
  - 被中央管理器装载组件，作为全局组件形式存在，在angular项目中 **除 `index.html` 文件以外**，其它任意的组件的模板文件中，都可以直接使用
  - 在index.html 文件中，可以直接使用的组件， 一定被 app.module.ts 作为根组件，所引导的组件
  - app.module.ts 中 bootstrap 所引导的组件，最终会将指定组件的 构成页面，以元素写入的方式，直接写入到 index.html 页面中的 指定位置，app.module.ts 可以直接引导多个根组件

- **组件模板定义时可以不用指定唯一的根节点**



2、自动化组件构成

- angular 脚手架工具中 提供一个 主命令
  - `generate (g)` : 为项目自动创建依赖文件和修改文件关联， `ng generate 需要创建的文件类型 文件名称` ==> `ng g 文件类型 文件名称`
  - **文件类型：指的是angular项目中 具有特定意义的 构成文件（组件，过滤器 ……）**
- `ng g --help` 查看自动化文件构建的子命令
  - 对于ng中一些子命令查看，执行语句必须在 angular 项目下才可以被识别

`Local workspace file ('angular.json') could not be found.`

- **ng g 命令的文件类型描述**

```
Available Schematics:
  Collection "@schematics/angular" (default):
    appShell
    application
    class
    component  ==>  c
    directive
    enum
```

```
  guard
  interface
  library
  module
  pipe
  service
  serviceWorker
  universal
  webWorker
```

- `ng g component 组件名称`
  - **1、该命令 会直接在 命令执行目录下配合路径定义 创建文件。== 执行命令前一定要确定 目录是否正确**
  - 2、定义的组件名称，如果使用了 驼峰命名法，项目中会自动转换为 连字符方式
  - **3、在指定的目录下先构建 以组件名作为参考的 文件夹，然后在文件夹创建组件所需文件**
  - **4、通过自动化工具创建的 文件，会以构成的文件类型 作为副 后缀进行 添加，为了区分项目中各种功能的文件**
  - **5、自动完成项目中管理器文件的，组件自动装载操作**

```
ITANY-IMAC-190:app appleuser$ ng g component first
CREATE src/app/first/first.component.css (0 bytes)  # 创建组件css
CREATE src/app/first/first.component.html (34 bytes) # 创建组件html
CREATE src/app/first/first.component.spec.ts (685 bytes)#创建测试文件
CREATE src/app/first/first.component.ts (304 bytes)# 创建组件TS
UPDATE src/app/app.module.ts (1577 bytes) # 更新管理器
```

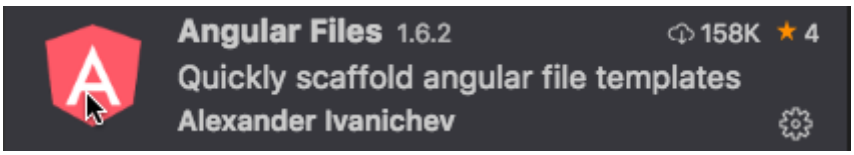
- **自动化创建的组件，组件标签名，将以 `项目前缀-组件名` 的方式定义**
  - `ng g component 组件名` 命令 在执行前会自动加载 angular.json 文件
  - 项目前缀在项目的 angular.json 进行定义，属性 prefix 完成了项目的前缀定义 ==> 直接影响自动化构成组件时选择器的前缀， **(通过修改文件改变项目前缀)**

    - 项目构建时 指定 子命令 设置项目的前缀 `ng new 项目名称 --prefix 前缀名` `ng new 项目名称 -p 前缀`
- **自动化组件创建时，主动构成样式文件；自动化工具如何识别创建的样式文件后缀**
    - 通过修改 angular.json 文件的配置。实现 自动化构成组建时，样式扩展名称的指定
        - 通过修改 angular.json `schematics` 属性进行功能的修改，修改是自动化命令的执行条件

```
"schematics": {
    "@schematics/angular:component": {
        "styleext": "less"
    }
}
```

    - 创建项目时，直接选择使用 动态语言，后续自动化构成组建时，会自动识别样式的后缀

vscode的自动脚手架工具



5、 angular项目中样式使用

1、全局样式和局部样式

- 全局样式 是由 angular.json 文件进行配置

```
"styles": [
    // 全局样式文件的定义
    "src/styles.css"
]
```

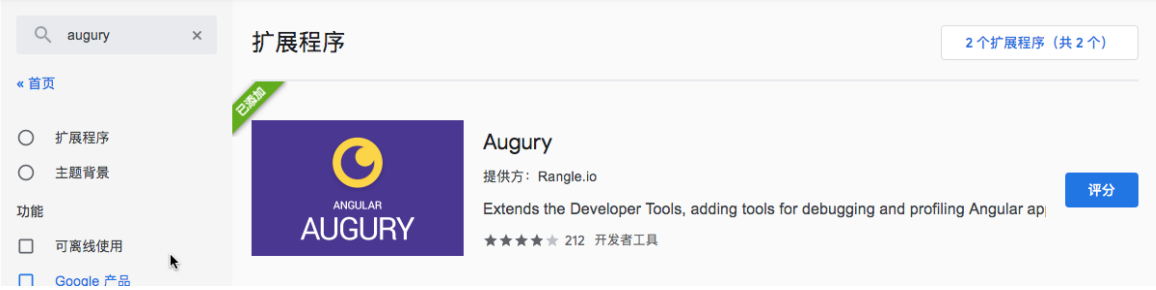
- 局部样式是以组件作为关联，构建组件时 所描述的 组件样式文件，就是当前组件的局部样式

2、动态样式添加

- angular项目在初始化后，进行项目依赖添加时，一次性将项目可能使用的 **所有支持扩展包统一的进行安装**
- 项目中只需要以固定的动态语法样式进行，样式添加即可
- angular 项目中 **不能定义统一 动态样式变量文件，全局的动态变量文件**，需要使用相关的动态语法的组件中进行 独立的引入和装载

6、插值表达式

- 语法：`{{ 变量 }}`
- angular的响应式：内存中变量发生变化，页面中与该变量有关的DOM元素会重新渲染取值
- angular 浏览器调试工具：augury (chrome 团队开发==只能chrome版本)




```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-insert',
    templateUrl: './insert.component.html',
    styleUrls: ['./insert.component.less']
})
export class InsertComponent implements OnInit {
    // 定义 class 类的基本属性，可以直接用于页面的变量取值
    msg:string = "默认值";
    num = 123;
    arr = [1,2,3,4];
    user = {
        name:"tom",
        age:23
    };
    arg1 = null;
    arg2 = undefined;
    math = Math; // 通过定义属性的方式间接的将JS内置对象转换为组件的属性
}
```

```
<h2>插值表达式</h2>
<pre>
- 语法：｛｛ 变量 ｝｝
    + 变量取值：1、当前模板对应的组件TS类的 定义属性
                2、直接定义 TS 的匿名变量 = 需要遵守匿名变量的定义规则（JS的规则）
                3、可以执行简单的运算表达式（四则、逻辑、比较、三目……）
- 插值表达式对于引用变量的处理方式
    + 因为HTML页面只会展示string类型数据，所有插值表达式向页面输出数据前，会执行字符串转化方式
    + angular 没有对 插值表达式 进行 string 转换规则重写 - 页面中变量的字符串结果是原始JS的转换结果
    + 对于 null undefined 会直接输出 “” 字符串
- angular 插值表达式不能 直接使用 JS 的内置对象 （JSON , Match）
    + 通过定义属性的方式间接的将JS内置对象转换为组件的属性
- angular 的插值表达式，可以定义在标签内中，也可以直接定义在 标签属性上
    + 定义属性取值时，只能为标签已知属性进行赋值操作 （HTML+DOM属性 -> 部分）
- angular的插值表达式具有响应式：内存中变量发生变化，页面中与该变量有关的DOM元素会重新渲染取值
</pre>
<p>基本使用:{{ msg }}</p>
<p>基本使用:{{ num }}</p>
<p>匿名变量:{{ "匿名字符串" }}</p>
<p>匿名变量:{{ 1234 }}</p>
<p>匿名变量:{{ true }}</p>
<p>表达式:{{ 1+2 }}</p>
<p>表达式:{{ 2>3 }}</p>
<p>表达式:{{ true?"真":"假" }}</p>
<p>表达式:{{ num - 100 }}</p>
<p>------------------</p>
<p>数组的处理:{{ arr }}</p>
<p>对象的处理:{{ user }}</p>
<p>null的处理:{{ arg1 }}</p>
<p>undefined的处理:{{ arg2 }}</p>
<!-- <p>JS内置变量:{{ Math.PI }}</p> -->
<p>JS内置变量:{{ math.PI }}</p>
<p>------------------</p>
<p id="{{ msg }}">定义在标签属性上，完成标签属性赋值操作</p>
```

7、事件绑定

- 基本语法：`<标签名 (事件名)="处理函数" >`

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-event-bind',
    templateUrl: './event-bind.component.html',
    styleUrls: ['./event-bind.component.less']
})
export class EventBindComponent implements OnInit {
    info = "默认字符串";
    num = 1;
    event = "自定义变量";
    printMsg() {
        console.log("自定义事件printMsg");
    }
    printInfo(){
        console.log("自定义事件printInfo");

    }
    printArg(a,b){
        console.log("方法参数:",a,b);
        console.log(this);
        console.log("通过this取值:",this.info,this.num);
    }
    printThis(arg){
        console.log(arg);
    }
    printEvent(e){
        console.log(e);
        // 方法参数 e 为事件源对象
        // 1、获取触发事件的 dom -> e.target
        // 2、获取触发事件的 dom 上绑定的属性 ->
e.target.className
        // 3、执行事件冒泡的阻止 -> e.stopPropagation();
        // 4、执行阻止事件的默认行为 -> e.preventDefault();
    }
    stopDefault(e){
        e.preventDefault();
    }
```

```
}
```

```
<h2>事件绑定</h2>
<pre>
- 基本语法：＜标签名  (事件名)="处理函数"＞
    + 以小括号方式描述为 标签绑定事件 - 调用 JS 中关于DOM的 addEventLister 方法
    + 取值函数的取值：1、直接定义当前页面关联的组件TS类中，定义的自定义方法
                        -> 如果处理函数的取值为对于组件中的一个自定义方法，在事件绑定时必须添加()
                        -> 如果绑定时不定义括号，事件函数是不能直接被触发的
                        -> 不加括号的定义方式，提供给组件传值使用
                    2、可以直接定义行内表达式，只能接收用于组件属性赋值的 简单表达式
                        -> 在事件绑定时，是可以直接调用 组件属性的
                        -> 简答的拼接，四则、逻辑、判断、三目 一样可以直接执行
                        -> 虽然可以定义简单表达式，但不能解析 自增自减…… ， += -= ……
- 对于标签的多事件绑定
    + angular 事件绑定中 (事件名称)="只能定义处理函数|行内表达式"
    + 多事件绑定，在angular的环境下，就是以 多个 (事件名) 方式进行定义
        * 多个事件的事件名称相同，以绑定先后执行绑定方法
        * 多个事件的事件名称不相同，以事件触发机制进行执行
- 事件的方法参数
    + 普通参数的取值范围
        * 取值为当前组件的自定义属性值 ， 取值为组件属性时，具有响应式特性
        * 取值为 匿名变量 或者 简单表达式的返回结果
    + this 参数
        * angular 将页面中绑定的方法传入的this，定义到当前页面对应的组件实例上
    + event 参数
        * angular 页面中的绑定事件方法，无法直接获取 JS中的事件源对象 event
        * 因为此时从angular 语法特性的定义，angular 认为event 为组件中的一个自定义属性
```

```
        * angular 自行定义了一个所有组件都可以直接调用 事件源属性 $event
</pre>
<input type="button" value="点击事件绑定1" (click)="printMsg()">
<input type="button" value="点击事件绑定2" (click)="printMsg">
<p>--------------</p>
info：{{ info }}:<input type="button" value="修改info" (click)=" info='info新值' ">
<input type="button" value="拼接info" (click)=" info=info+'-字符串后缀' ">
<input type="button" value="不赋值" (click)=" info+'-字符串后缀' ">
<!-- <input type="button" value="弹窗" (click)=" alert(1) "> -->
<p>--------------</p>
num:{{ num }}
<input type="button" value="+" (click)=" num = num + 1 ">
<input type="button" value="-" (click)=" num = num - 1 ">
<p>--------------</p>
<!-- <input type="button" value="取值[]" (click)=" [printInfo,printMsg] "> -->
<input type="button" value="绑定多个click" (click)=" printMsg() " (click)="printInfo()">
<input type="button" value="绑定多个click" (click)="printInfo()" (click)=" printMsg() ">
<input type="button" value="绑定click+mousedown"  (click)=" printMsg() " (mousedown)="printInfo()">
<p>--------------</p>
<input type="button" value="传递参数" (click)="printArg( info,num )">
<input type="button" value="传递参数" (click)="printArg( '匿名字符串',1+1 )">
<input type="button" value="传递this" (click)="printThis( this )">
<input type="button" value="传递event" (click)="printEvent( event )">
<input type="button" value="传递$event" (click)="printEvent( $event )">
<a href="https://www.baidu.com" (click)=" stopDefault($event) ">百度</a>
```

```
<a href="https://www.baidu.com" (click)="
$event.preventDefault() ">百度</a>
```

8、指令

- angular的指令 完成页面中的 元素的属性操作，元素的创建删除，元素的迭代循环

8.1、属性指令

- 属性指令：对元素的属性进行数据绑定操作
- 语法：`<标签名 [属性名]="属性取值" >`

1、普通属性指令

- angualr项目中，html 元素的 所有 HTML属性和DOM属性，在使用相关语法后，都叫做 普通属性指令

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-attr-directive',
    templateUrl: './attr-directive.component.html',
    styleUrls: ['./attr-directive.component.less']
})
export class AttrDirectiveComponent implements OnInit {
    arg1 = "参数1";
    classStr = "a";
    domStr = "<h4>H4标签</h4>";
    imgsrc = null;
    flag = false;
}
```

```
<h2>普通属性指令</h2>
<pre>
- 属性指令：为元素进行动态的属性数据绑定
- 普通属性指令：angular 项目对外提供一个自定义的 HTML和DOM的属性操作指令
    + 在angular 只要是HTML或者DOM自有的属性，且被angular支持，都可以使用该指令进行动态数据绑定
```

- 内置属性指令：angular 对一些需要进行特殊处理或者进行安全处理的属性，进行独立功能定义的指令
    + 内置属性指令，时angular已经定义完成，用于解决安全属性操作或多样性属性操作的指令
- 结构指令：通过angular定义的相关功能，实现元素的创建删除，或者循环功能
</pre>
<pre>
- 普通属性指令
    + 语法：＜标签 [待绑定属性名]=" 属性取值 " ＞＜/标签＞
    + 待绑定属性名：被W3C规范的 HTML或者DOM 已知的属性 -> 需要被angular支持
    + 属性取值：可以直接取值为组件属性变量 | 匿名变量 | 简单表达式
- 保留项目的 数据响应式 功能
- 不能直接绑定一个 元素的 未知的自定义属性
    + 程序开发如果需要绑定一个未知的自定义属性
    + 可以直接通过 angular 对于普通属性指令 增加 属性描述对象 attr 进行绑定
        例：＜p [attr.itany]=" arg1 "＞为标签绑定自定义属性＜/p＞
    + attr 关键字实际上是 angular 针对于 标签DOM对象封装的一个特殊的操作方法
    + attr 封装的是：setAttribute(自定义属性名,取值)
- 基于普通属性的操作方式，可以补充一个插值表达式无法完成页面写入操作
    + 插值表达式 只会 将纯文本 写入到页面，string方式的标签元素不被解析
    + 通过对标签属性 innerHTML 的绑定操作，完成DOM元素的解析
- 在angular 项目中属性指令可以绑定的操作，插值表达式都可以操作
    + 插值表达式 完成数据变量写入页面之前，会将变量转换为 string
    + 属性指令，会保留属性类型，直接为属性赋值，值得转换取决 HTML|DOM 属性所需的数据类型
    * 因为取值不同，会直接造成两种标签属性的取值效果不同
    * 资源定义属性 （src） - 建议使用 插值表达式绑定
    * 标签的boolean 类型属性 - 建议直接使用 属性绑定方式
</pre>
<p [id]=" arg1 " >绑定动态ID属性
绑定动态ID属性
<input type="text" value="" (input)=" arg1 = $event.target.value ">
<p [className]="classStr" >绑定DOM属性
绑定DOM属性

```
<input type="button" value="切换a|b" (click)=" classStr =
classStr=='a'?'b':'a' ">
<hr>
<!-- <p [name]=" arg1 ">绑定属性name</p> -->
<!-- <p [itany]=" arg1 ">为标签绑定自定义属性</p> -->
<p [attr.itany]=" arg1 ">为标签绑定自定义属性</p>
<p attr.itany="{{ arg1 }}">为标签绑定自定义属性</p>
<div>{{ domStr }}</div>
<div [innerHTML]=" domStr "></div>
<div innerHTML="{{ domStr }}"></div>
<hr>
<div id="{{ null }}">插值表达式，id=null</div>
<div [id]=" null ">属性指令，id=null</div>
<div id="{{ undefined }}">插值表达式，id=undefined</div>
<div [id]=" undefined ">属性指令，id=undefined</div>
<hr>
<img src="{{ imgsrc }}" alt="">
<img [src]="imgsrc" alt="">
<input type="button" value="imgsr的定义" (click)="
imgsrc='https://www.baidu.com/img/baidu_jgylogo3.gif' ">
<hr>
<input type="text" name="" id="" disabled="{{ flag }}">
<input type="text" name="" id="" [disabled]=" flag ">
<input type="button" value="切换falg" (click)=" flag=!flag">
```

2、内置属性指令

- ngClass：在项目开发中，提供简单 class 样式的操作方式
- ngStyle：在项目开发中 直接通过JS为标签绑定行内样式 为非安全模式，angular提供的ngStyle指令主要为了解决绑定安全文件

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-inner-directive',
    templateUrl: './inner-directive.component.html',
    styleUrls: ['./inner-directive.component.less']
})
export class InnerDirectiveComponent implements OnInit {
    classStr = "fctest fc fs bc border";
    classArr = ["fctest","fc","fs","bc","border"];
    classObj = {
```

```
        // key 描述需要在标签上操作的样式名称
        // value 只能取值为 boolean 类型，通过 true和false实现样式的加载或删除
        fctest:true,
        fc:true,
        fs:true,
        bc:false,
        border:false,
    };
    styleStr = "color:red;font-size:24px";
    styleObj = {
        color:"red",
        // "font-size":"24px"
        fontSize:"24px"
    }
    addOrRemove(arg){
        if(this.classStr.indexOf(arg)==-1){
            this.classStr = this.classStr + " " + arg;
        }else{
            this.classStr = this.classStr.replace(arg,"");
        }
    }
    addOrRemove2(arg){
        let i = this.classArr.indexOf(arg);
        if(i==-1){
            this.classArr.push(arg);
        }else{
            this.classArr.splice(i,1);
        }
    }
}
```

```
<h2>内置属性指令</h2>
<pre>
- 属性指令：为元素进行动态的属性数据绑定
- 普通属性指令：angular 项目对外提供一个自定义的 HTML和DOM的属性操作指令
    + 在angular 只要是HTML或者DOM自有的属性，且被angular支持，都可以使用该指令进行动态数据绑定
- 内置属性指令：angular 对一些需要进行特殊处理或者进行安全处理的属性，进行独立功能定义的指令
```

    + 内置属性指令，时angular已经定义完成，用于解决安全属性操作或多样性属性操作的指令
- 结构指令：通过angular定义的相关功能，实现元素的创建删除，或者循环功能
</pre>
<pre>
- 内置属性指令 NgClass
    + 普通属性指令不能提供简单快速的 class 样式操作方式，所以封装了一个 特殊的属性 ngClass
- ngClass 的使用语法
    + ＜标签 [ngClass]=" 样式值 " ＞
    + 样式值 可以取值的类型有
        * 取值为 string 类型 ： 功能和普通 属性绑定 class 以及 插值表达式赋值class一样
        * 取值为 array 类型 ： 数组的每一个值就是当前元素需要绑定的 class 样式
                                可以直接通过对数组方法调用完成相关样式删除 添加 判断 等操作
        * 取值为 object 类型 ： 对象中的每一个 键值对 都是一个标签需要样式的状态描述
                                其中key为样式名称，value通过取值true和false 调整样式的添加和删除
</pre>
<p [class]=" classStr ">动态class绑定
动态class绑定
<input type="button" value="添加|删除fc" (click)="addOrRemove('fc') ">
<input type="button" value="添加|删除fs" (click)="addOrRemove('fs') ">
<input type="button" value="添加|删除bc" (click)="addOrRemove('bc') ">
<input type="button" value="添加|删除border" (click)="addOrRemove('border') ">
<hr>
<p [ngClass]=" classStr ">ngClass 绑定字符串
<p [ngClass]=" classArr ">ngClass 绑定数组
<input type="button" value="添加|删除fc" (click)="addOrRemove2('fc') ">
<input type="button" value="添加|删除fs" (click)="addOrRemove2('fs') ">

```
<input type="button" value="添加|删除bc" (click)="
addOrRemove2('bc') ">
<input type="button" value="添加|删除border" (click)="
addOrRemove2('border') ">
<hr>
<p [ngClass]=" classObj ">ngClass 绑定对象</p>
<input type="button" value="添加|删除fc" (click)=" classObj.fc
= !classObj.fc ">
<input type="button" value="添加|删除fs" (click)=" classObj.fs
= !classObj.fs ">
<input type="button" value="添加|删除bc" (click)=" classObj.bc
= !classObj.bc ">
<input type="button" value="添加|删除border" (click)="
classObj.border = !classObj.border">
<pre>
- 内置属性指令 NgStyle
    + 通过普通属性或者插值表达式直接为 style 属性赋值，会导致页面xss
漏洞，进而导致程序出现安全性问题
    + angular 会阻止所有对于 style 直接操作，对标签的style操作，必
须通过内置指令 ngStyle 方式完成
    +  ngStyle 指令是一个经过代码封装的安全操作方法
- ngStyle 的使用语法
    + ＜标签 [ngStyle]=" 样式值 "  ＞
    + 样式值取值必须为 Obeject 对象值
        key 用于描述需要加载的行内样式名称,样式名可以通过引号完成连字
符名称定义，也可以直接使用驼峰名称
        value 用于定义该样式的合法取值
</pre>
<p [style]=" styleStr ">动态绑定 style</p>
<!-- <p style="{{ styleStr }}">动态绑定 style</p> -->
<p [ngStyle]=" styleObj ">动态绑定 style - ngStyle</p>
```

8.2、结构指令

- 结构指令：对元素的创建删除和迭代进行操作
- 语法：`<标签属性    *结构指令名="取值或表达式"    >`

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-stur-directive',
    templateUrl: './stur-directive.component.html',
```

```
    styleUrls: ['./stur-directive.component.less']
})
export class SturDirectiveComponent implements OnInit {
    flag = true;
    num = 0;
    name = "tom";//汤姆
    citys = ["南京","苏州","常州","上海"];
    size = 3;
    str = "abcdefg";
    user = {
        name:"tom",
        age:23
    }
    getSizeArr(){
        let arr = [];
        arr.length = this.size;
        return arr;
    }
    getStrArr(){
        return this.str.split("");
    }
    getUserArr(arg){
        let arr;
        if(arg=="key"){
            arr =  Object.keys(this.user)
        }
        if(arg=="value"){
            arr =  Object.values(this.user)
        }
        if(arg="entrie"){
            arr = Object.entries(this.user); //[[key,value],
[key,value]……]
        }
        // Object.values;
        // Object.entries;
        return arr;
    }
}
```

```
<h3>结构指令</h3>
<pre>
```

- 属性指令：为元素进行动态的属性数据绑定
    - 普通属性指令：angular 项目对外提供一个自定义的 HTML和DOM的属性操作指令
        + 在angular 只要是HTML或者DOM自有的属性，且被angular支持，都可以使用该指令进行动态数据绑定
    - 内置属性指令：angular 对一些需要进行特殊处理或者进行安全处理的属性，进行独立功能定义的指令
        + 内置属性指令，时angular已经定义完成，用于解决安全属性操作或多样性属性操作的指令
- 结构指令：通过angular定义的相关功能，实现元素的创建删除，或者循环功能
</pre>
<pre>
- ngIf 结构指令 ：通过可判断变量 描述元素在页面的创建和删除操作
- 语法：＜标签 *ngIf=" flag "＞
    + 标签在页面的创建或者删除 取决于变量 flag 的boolean 状态
    + 标签在页面中执行的是 创建和删除 操作
    + flag可以取值为组件属性变量，也可以取值为匿名变量或者简单表达式
- 配合判断标签
    + ＜ng-template [ngIf]=" flag "＞＜/ng-template＞ 块循环标签-将多个元素合并成整体循环块
    + 该标签不具有 页面样式和页面功能 ，程序编译后该标签不存在于代码，用于整体包裹带判断标签
</pre>
<p *ngIf=" flag ">ngIf的绑定判断
<input type="button" value="切换元素创建和删除" (click)=" flag = !flag ">
num:{{ num }}
<span *ngIf=" num<0 ">小于0
<span *ngIf=" num==0 ">等于0
<span *ngIf=" num>0 ">大于0
<input type="button" value="+" (click)=" num=num+1 ">
<input type="button" value="-" (click)=" num=num-1 ">
<hr>
<ng-template [ngIf]="flag">
    标签1
    标签2
</ng-template>
<input type="button" value="切换元素创建和删除" (click)=" flag = !flag ">
<pre>

- ngSwtich ngSwtichCase ngSwtichDefault : 完成页面元素通过变量等值比较结果实现 切换操作
- 语法和对比

```
    JS switch                              angular ngSwtich
    swtich(变量) {                  <父标签  [ngSwitch]=" 变量 " >
        case 值:                            <子标签 *ngSwitchCase=" 比较值 " >
            表达式;
            break;
        case 值:                            <子标签 *ngSwitchCase=" 比较值 " >
            表达式;
            break;
        default:                            <子标签 *ngSwitchDefault >
            表达式
    }                                  </父标签>
```

- ngSwtich 还是 ngSwitchCase 取值为组件属性变量，也可以取值为匿名变量或者简单表达式结果

```
</pre>
<p>name:{{ name }}</p>
<input type="text" (input)=" name = $event.target.value ">
<div [ngSwitch]=" name ">
    <span *ngSwitchCase=" 'tom' ">汤姆</span>
    <span *ngSwitchCase=" 'jack' ">杰克</span>
    <span *ngSwitchDefault>未知</span>
</div>
<pre>
```

- ngForOf : 完成页面元素 根据 可循环数据 实现元素的循环创建
- ngForOf 的指令名称 ngFor ,关键字 of 为循环过程中的定义关键字
- 语法 <标签 *ngFor=" let 循环临时变量 of 待循环变量; " >
    + let 为循环过程中产生的临时循环值 进行变量名称定义
        * 通过循环定义的循环零时变量，在循环范围内，可以直接作为组件属性使用
        * 临时变量为数组进行循环时，循环到的元素值
    + of 描述被循环的 变量
- angular 的ngForOf只能循环数组
- 辅助循环属性，辅助属性在循环时存在值，但不能直接使用,如果需要使用必须将值定义为一个临时变量
    * <标签 *ngFor=" let 循环临时变量 of 待循环变量;let 临时变量=辅助值;…… " >

```
    + index 描述循环时的元素下标
    + first 描述循环时的元素是否为第一个循环值  - 返回boolean
    + last  描述循环时的元素是否为第最后一个循环值  - 返回boolean
    + even  描述循环时的元素是否为下标偶数的循环值  - 返回boolean
    + odd   描述循环时的元素是否为下标奇数的循环值  - 返回boolean
- 如果循环的数据为非数组值
    + 可以通过相关方法将非数组值转换为数组值
    + ngForOf直接可以直接调用一个返回数组结果的，组件自定义方法
</pre>
<ul>
    <li>下标-值-first-last-even-odd</li>
    <li *ngFor="let c of citys;
                let i = index;
                let f = first;
                let l = last;
                let e = even;
                let o = odd;
                "
    >{{ i }}:{{ c }}-{{ f }}-{{ l }}-{{ e }}-{{ o }}</li>
</ul>
<!-- <p *ngFor=" let i of 8 "></p> -->
<hr>
<p>size:{{ size }}</p>
<input type="button" value="+" (click)=" size=size+1 ">
<input type="button" value="-" (click)=" size=size-1 ">
<ul>
    <li *ngFor=" let item of getSizeArr();let i = index; ">{{
i+1 }}</li>
</ul>
<hr>
<input type="text" (input)=" str=$event.target.value ">
<ul>
    <li *ngFor=" let item of getStrArr();let i = index; ">{{ i
}}:{{ item }}</li>
</ul>
<hr>
<ul>
    <li *ngFor=" let k of getUserArr('key'); ">{{ k }}:{{
user[k] }}</li>
</ul>
<ul>
```

```
    <li *ngFor=" let v of getUserArr('value'); ">{{ v }}</li>
</ul>
<ul>
    <li *ngFor=" let ent of getUserArr('entrie'); ">{{ ent[0] }}:{{ ent[1] }}</li>
</ul>
<ul>
    <li *ngFor=" let ent of getUserArr('entrie'); ">
        <span *ngFor=" let temp of ent; ">{{ temp }}-</span>
    </li>
</ul>
```

9、模板变量

- angular的模板变量,在模板页面以特定语法方式，为DOM元素指定一个变量名，让DOM元素可以直接通过变量名的方式完成获取

```
<h3>模板变量</h3>
<pre>
- 模板变量：angular项目中提供一个用于快速获取DOM对象的定义方式
- 定义语法：＜标签 #自定义变量名 ＞ ，自定义变量名可以直接只带当前DOM元素
- 使用方式：自定义变量名 ，可以在模板范围内，直接以组件属性变量的使用方式进行使用
- 使用范围：模板变量只能在当前模板页面中使用，出当前模板后变量无法使用
- 使用的注意事项：
    + 模板变量名不能重复
    + 模板变量的优先级要高于组件属性变量，当前组件属性和模板变量名称重复时
        * 页面中的插值表达式和指令都将默认读取模板变量
        * 当前页面中的模板变量名称和组件属性变量名称相同，组件属性的变量值无法在模板中进行调用
            当不影响组件属性变量在组件内的使用方式
- 模板变量的实际用途
    + swiper 的轮播图的实现
    + 如何让组件TS文件自动加载 模板变量？
        * 通过angular内置的属性修饰器方式，将模板变量加载进组件中
            1、在组件中定义一个普通的组件属性变量,不用取值；
            2、angular提供属性装饰 @ViewChild(模板变量名,独立配置项) ，根据模板变量名完成DOM到组件属性的关联操作
```

```
                @ViewChild("loop", {
                    static:true
                } )
                loop:ElementRef; // 该变量不是直接接受的DOM元素，而是被angular进行封装的一个对象中，DOM存储于该对象的 nativeElement
    + 如何让angular的TS语法直接使用JS插件？
        1、为项目通过 npm install 方式按照插件依赖
        2、在angular.json 文件中 直接以项目插件方式定义 script 属性配置
            "scripts": [
                "./node_modules/swiper/js/swiper.js"
            ]
        3、在使用的TS文件中，以 declare 完成JS变量到TS变量的转换定义
            declare var Swiper;
</pre>
<p #pdom>dom标签1</p>
<!--
    <p #pdom>dom标签2</p>
    <p #pdom>dom标签3</p>
-->
<div>取值模板变量pdom:{{ pdom }}</div>
<!-- <div>取值组件属性变量pdom:{{ this.pdom }}</div> -->
<input type="button" value="输出模板变量pdom"
(click)="printDom( pdom,this )">
<hr>
<div class="swiper-container" #loop>
    <div class="swiper-wrapper">
        <div class="swiper-slide">slider1</div>
        <div class="swiper-slide">slider2</div>
        <div class="swiper-slide">slider3</div>
    </div>
</div>
<input type="button" value="打印LOOP" (click)="printLoop()">
<input type="button" value="初始化swiper"
(click)="initSwiper()">
```

```
import { Component, OnInit, ViewChild, ElementRef, OnDestroy }
from '@angular/core';

declare var Swiper;
```

```
@Component({
    selector: 'app-template-var',
    templateUrl: './template-var.component.html',
    styleUrls: ['./template-var.component.less']
})
export class TemplateVarComponent implements OnInit,OnDestroy
{
    pdom = "字符串";
    @ViewChild("loop",{
        static:true
    })
    loop:ElementRef;

    constructor() {
        console.log("TemplateVarComponent构造方法");
        // this.initSwiper();
    }
    // 组件生命周期
    ngOnInit() {
        // 在组件实例对象创建完成直接执行的方法
        this.initSwiper();
    }
    ngOnDestroy(): void {
        // throw new Error("Method not implemented.");
    }
    printDom(pdom,arg) {
        console.log(pdom,arg,this.pdom);
        pdom.style.color = "red";
    }
    printLoop(){
        console.log(this.loop);
        console.log(this.loop.nativeElement);

    }
    initSwiper(){
        // console.log(Swiper);
        console.log("initSwiper")
        new Swiper(this.loop.nativeElement,{
            autoplay: true,//可选选项，自动滑动
        });
```

```
    }
}
/*
    var TemplateVarComponent = (function(){
        var TemplateVarComponent = function(){
            this.loop = ??;
            this.initSwiper();
        }
        TemplateVarComponent.propetype.initSwiper = function()
{
            this.loop.nativeElement
        }
        return TemplateVarComponent;
    })()

*/
```

10、双向数据绑定

- 双向数据绑定：内存环境中变量的变换会直接重新渲染页面，页面中用户的操作输入，会直接影响内存变量
- **angular 的 TS 版本中，默认不再提供双向数据操作，将双向数据绑定功能移动到 angular 的一个独立的功能模块中，表单操作模块**
  - 表单模块：双向数据绑定，用户输入数据的校验功能
- 为项目增加表单操作模块
  - 表单操作模块，存放在 `@angular/forms` ==> 提供一个独立的模块对象 `FormsModule`
  - angular 项目而言，整个语法构成为中央管理器模式，使用 组件，模块..... 都需要被中央管理，需要通过修改 `app.module.ts` 文件完成 模块的导入和加载
    - 1、在 app.module.ts 中央管理器，导入 表单模块

```
// 完成表单处理模块的加载
import { FormsModule } from "@angular/forms";
```

    - 2、将加载的模块，提供给管理器进行管理 `@NgModel({})`

```
@NgModule({
  // 描述 angular 项目运行时，所需要的组件 方法 过滤器，指令等等
  declarations: [
    AppComponent,
  ],
  // 为项目提供 功能模块的加载 ，浏览器兼容，ajax，路由功能……
  imports: [
    BrowserModule,
    FormsModule // 表单模块的加载
  ],
  // DI 提供者 ？DI=依赖注入 ？？？
  providers: [],
  // 定义项目启动时 angular的引导组件 == Root组件 == 根组件
  bootstrap: [AppComponent]
})
export class AppModule {
  // 导出的是一个 名称叫做 AppModule 的普通的空的 TS class
}
```

- 项目中使用 双向数据操作
  - 使用语法：banana in box == `＜表单输入标签［（ngModel）］= " 绑定变量 "  ＞`

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

@Component({
    selector: 'app-two-way',
    templateUrl: './two-way.component.html',
    styleUrls: ['./two-way.component.less']
})
export class TwoWayComponent {
    arg = undefined;
    msg = "默认值";
    info = "默认值";
    radioData = "";
    checkData = [];
    skills = {
        html:false,
        css:false,
        js:false,
```

```
        vue:false
    }
    selectData = "";
    getKeys(){
        return Object.keys(this.skills);
    }
}
```

```
<h3>双向数据操作</h3>
<pre>
- 为项目进行 @angular/froms 模块的加载
- 表单模块提供一个特殊的 标签 指令 ngModel ,可定义于表单输入标签上,直接实现双向数据操作
- 使用语法：＜表单标签 [（ngModel）]=" 取值 " ＞ 【 banana in box 语法 】
    + [] 属性绑定
    + () 事件绑定
</pre>
<pre>
- 1、单行文本框
    ngModel 绑定于单行文本框上，执行的是如下操作
        * 以属性绑定方式 绑定value
        * 以事件绑定方式 绑定input
</pre>
<p>msg:{{ msg }}</p>
<input type="text" [value]="msg" #input1 (input)="msg=input1.value ">
<input type="text" [(ngModel)]=" msg ">
<pre>
- 2、多行文本域
    ngModel 绑定于多行文本域上，执行的是如下操作
        * 以属性绑定方式 绑定value
        * 以事件绑定方式 绑定input
</pre>
<pre>{{ info }}</pre>
<textarea cols="20" rows="5" [value]="info" #text (input)="info = text.value"></textarea>
<textarea cols="20" rows="5" [(ngModel)]=" info "></textarea>
<pre>
- 3、单选按钮
    ngModel 绑定单选按钮，执行下述操作
```

```
        * 以属性绑定方式 绑定 checked
        * 以事件绑定方式 绑定 change
            change 被执行时，通过比较 绑定变量和 标签value的值，判断 checked 的操作方式
            如果value值没有定义 返回 undefined 结果
</pre>
<p>radioData:{{ radioData }}</p>
<div>
    <input type="radio" value="a" [(ngModel)]=" radioData ">
    <input type="radio" value="b" [(ngModel)]=" radioData ">
    <input type="radio" value="c" [(ngModel)]=" radioData ">
</div>
<div>
    <input type="radio" value="a" #radio1 [checked]="
radio1.value==radioData " (change)=" radioData=radio1.value ">
    <input type="radio" value="b" #radio2 [checked]="
radio2.value==radioData " (change)=" radioData=radio2.value ">
    <input type="radio" value="c" #radio3 [checked]="
radio3.value==radioData " (change)=" radioData=radio3.value ">
</div>
<pre>
- 4、复选按钮
    ngModel 在对复选按钮进行双向数据绑定时，不会关心 value属性是否定义
    对于复选框的双向数据操作，在angular语法下只记录 当前复选标签的状态
        * 属性 checked 的绑定，用于控制标签的页面状态切换
        * 事件 change 的绑定，是为变量提供复选标签的状态结果
</pre>
<div>
    <input type="checkbox" value="a" [(ngModel)]="checkData">
    <input type="checkbox" value="b" [(ngModel)]="checkData">
    <input type="checkbox" value="c" [(ngModel)]="checkData">
</div>
<div>
    <input type="checkbox" [checked]=" checkData " #check
(change)=" checkData=check.checked ">
</div>
<p>----复选按钮的使用----</p>
<div *ngFor=" let key of getKeys() ">
    <label>{{ key }}</label>
```

```
    <input type="checkbox" [(ngModel)]=" skills[key] ">
</div>
<pre>
- 5、下拉列表
    ngModel 在对下拉列表进行数据绑定时，只针对于 select 标签
        * 属性绑定的是 select 的 value 属性，通过value属性的操作描述option的选中状态
        * 事件绑定的是 change 事件，change 事件通过对绑定变量的赋值操作，完成选中option的值得获取
</pre>
<select [(ngModel)]=" selectData ">
    <option value="">请选择</option>
    <option value="a">A</option>
    <option>B</option>
    <option value="c">C</option>
</select>
```

11、管道（过滤器）

- 管道的功能：对传递的数据进行数据处理，将处理后的新数据进行返回
- 使用语法：`被处理的数据 | 管道名`

11.1、内置过滤器

```
<h3>管道</h3>
<pre>
- 管道也被叫做过滤器，对页面展示数据进行处理，将处理后的数据写入到页面
- 语法：管道可以被定义在 插值表达式、属性指令、结构指令的取值部分 ，**管道调用不能添加括号**
    + <标签>{{ 待处理数据 | 管道名 }}</标签>
    + <标签 [指令]=" 待处理数据 | 管道名 " ></标签>
- 管道分为：无参数管道、有参数管道
- 管道传递参数时，在管道名称后以 : 方式描述参数 ， 每一 : 描述管道的一个参数
    如果管道定义了多个：表示该管道需要接受多个参数
</pre>
<pre>
- 内置管道
  1、jsonPipe 管道
```

```
    作用：把一个值转换成 JSON 字符串格式。（该管道是为开发者提供调试的对象数据的显示功能）
    用法：value_expression | json
        * value_expression : 描述待处理数据|变量|具有返回结果的简单表达式
        * | : 管道分割符,分割待处理变量和管道，或者分割管道和管道，（管道是可以连写）
        * json : 当前调用管道的名称
</pre>
<p>user:{{ user }}</p>
<p>user:{{ user | json }}</p>
<input type="text" name="" id="" [(ngModel)]=" user.name ">
<p [id]=" user | json ">将管道应用于属性</p>
<pre>
    2、DecimalPipe 管道
      作用：把数字转换成字符串，根据地区的定义规则，转换数值的展示情况
           * 千分位，四舍五入，位数补充
      语法：value_expression | number [ : digitsInfo [ : locale ] ]
        * value_expression 待处理数值变量
        * number 管道名称
        * 关键字 [] 描述参数可写可不写
        * digitsInfo 描述数值格式化规则 取值为string
            + 该参数存在固定的定义形式 minIntegerDigits.minFractionDigits-maxFractionDigits
                minIntegerDigits：在小数点前的最小位数。默认为 1。(整数位的补0)
                minFractionDigits：小数点后的最小位数。默认为 0。(小数位的补0)
                maxFractionDigits：小数点后的最大为数，默认为 3。(保留几位小数)
                minFractionDigits <= maxFractionDigits
        * locale 根据配置所属地区，转换格式化规则
      默认转换规则：当该参数不进行传递时，存在转换的默认规则，千分位四舍五入 保留三位小数,默认转换地区为 en-US
</pre>
<p>num:{{ num }}</p>
<p>num:{{ num | number }}</p>
<p>num:{{ num | number:'1.0-1' }}</p>
<p>num:{{ num | number:'1.0-1':'zh-Hans' }}</p>
<p>num1:{{ num1 }}</p>
```

```
<p>num1:{{ num1 | number:'3.2-2' }}</p>
<p>num1:{{ num1 | number:'3.2-2':'zh-Hans' }}</p>
<pre>
    3、DatePipe 日期时间格式化
        作用：完成日期和时间的格式化操作
        语法：value_expression | date [ : format [ : timezone [
: locale ] ] ]
            * value_expression 待处理数据
            * date 管道符号
            * format 日期格式化规则
              + 内置格预定义的格式文字
              + 通过日期时间对应字段列表进行格式组合
                  对于自定义的规则，会根据字段列表替换成相关日期时间
，  无法匹配的字符会直接保留
            * timezone 时区设置
              + 设置时间的展示时区 （默认是以程序运行的计算的时区进行定
义的）
</pre>
<p>day:{{ day }}</p>
<p>day:{{ day | date }}</p>
<p>day:{{ day | date:"a" }}</p>
<p>day:{{ day | date:"yyyy-MM-dd" }}</p>
<p>day:{{ day | date:"hh:mm:ss a" }}</p>
<p>day:{{ day | date:"HH:mm:ss a" }}</p>
<p>day:{{ day | date:"yyyy-MM-dd HH:mm:ss a" }}</p>
<p>当前的格林威治时间:{{ day | date:"yyyy-MM-dd HH:mm:ss":"+0"
}}</p>
<p>东12区时间:{{ day | date:"yyyy-MM-dd HH:mm:ss":"+12" }}</p>
<p>西12区时间:{{ day | date:"yyyy-MM-dd HH:mm:ss":"-12" }}</p>
```

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-pipe',
    templateUrl: './pipe.component.html',
    styleUrls: ['./pipe.component.less']
})
export class PipeComponent implements OnInit {
    user = {
        name:"tom",
        age:23
    }
```

```
    num = 1234.34567;
    num1 = 1.1234;
    day = new Date();
}
```

11.2、自定义过滤器

- 1、手动方式
- 2、自动方式 `ng g pipe 管道名`

1、手动构建管道

- a、angular 中的管道创建，本身就是定义一个 普通的 TS 类
- b、以装饰器 `@pipe` ，描述该 class 为 angular项目的管道类

```
import { Pipe } from '@angular/core';

@Pipe({
    // 管道名称的定义
    name:"rep"
})
export class Repalce{

}
```

- c、必须定义一个管道处理方法，遵循管道处理方法的定义规范==> 让自定义的class 实现 angular内置的 接口 `PipeTransform` ，该接口就是用于定义 自定管道类必须实现的，处理方法接口

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    // 管道名称的定义
    name:"rep"
})
export class ReplaceFun implements PipeTransform{
    transform(value, arg) {
        let reg = new RegExp(arg,"g");
        return value.replace(reg,"");
    }
```

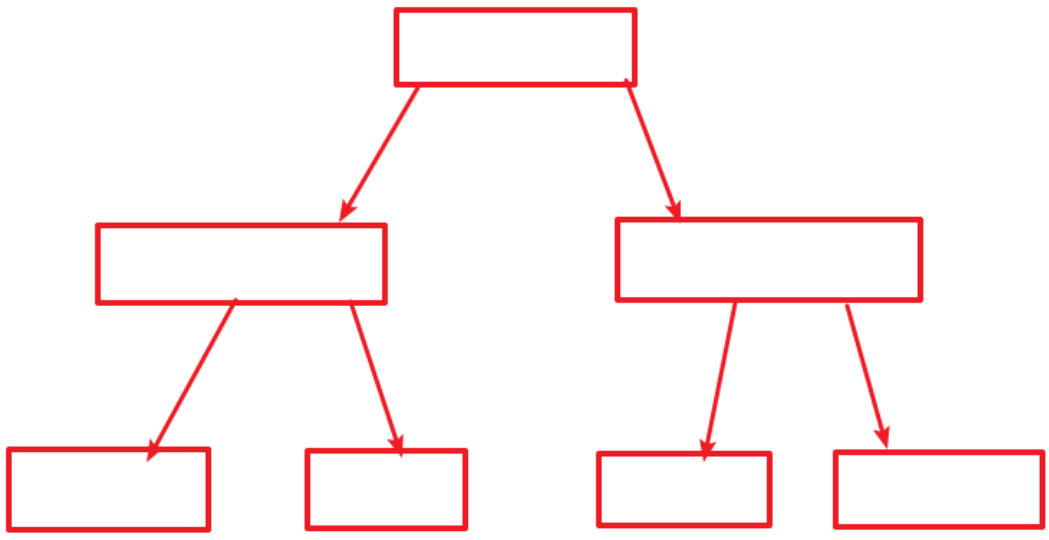
```
    // 自定管道被执行时 ，会调用方法
    // transform(value, arg1,arg2,arg3) {
    //     // value: 指的是 管道调用时 管道符号| 左侧的 被转换的数据
    //     //  从方法的第二个参数开始，表示的是 管道调用时 以 : 方式逐位传递的额外参数
    //     console.log("rep管道方法transform: ",value,arg1,arg2,arg3);
    // }

    // aaa(){}
    // bbb(){}
}
```

- d、将自定义的管道添加到 中央管理器的文件中，进行管道注册

页面使用：

```
<p>{{ num | number | rep:"," }}</p>
```

12、组件间的数据共享



12.1、父组件向子组件传递数据

- 技术：属性定义，属性绑定

12.2、子组件向父组件传递数据

- 技术：事件绑定，事件触发

12.3、非父子组件的数据传递

1、通过共同父组件方式进行数据传递

- 寻找共同的顶级组件，以组件中 父组件到子组件，子组件到父组件的传递方式，进行数据传递

2、DI 注入方式

- DI:(Dependency injection == **依赖注入**) 来自后端代码JAVA的一种设计模式
- 依赖注入实际上是一种程序的运行方式
    - 1、TS中定义的 class 实际上就是JS中一个 对象构造器，**对象构造器在JS使用前，完成对象创建**
    - 2、**angular项目是由很过个 class 构成项目结构，整个项目代码没有出现一次 new 关键字**
        - angular 会直接完成对于 所有 构造器的 实例创建(**自动对象构建**) == angular 原始代码本身完成 所有class组件的创建
        - angular 在尝试完成组件 对象创建时 ， 必然会执行 构造方法的构造函数，如果此时构造函数定义了 传入的参数，angular 尝试从 注入者对象中寻找匹配的class类

> 中央管理器在项目运行时，根据调用 **自动创建对应的构造器实例；**
>
> 在创建构造器实例时，会查看当前构造器是否需要**依赖参数**；
>
> 如果存在依赖参数，中央管理会自动将依赖参数**传递(注入)**到组件构造函数形参上

- angular项目在构成 DI 提供时，使用的是 JS中对象的**单例模式**

- 因为单列模式，所以组件中所完成的 DI注入，实际上是对一个对象的操作
- 参数为对象-JS为引用类型 == 堆中的数据变化，不会导致对象的改变
- **单例模式**：在程序运行时　无论执行多次new 关键子 只会被构建一次