# Overview

1. 资源
   1. ATL文档：https://docs.microsoft.com/en-us/cpp/atl/atl-com-desktop- components
   2. WTL文档：http://www.viksoe.dk/code/all\_wtl.htm
   3. WTL下载地址：https://sourceforge.net/projects/wtl/
2. 背景
   1. Windows Template Library，**WTL**是一个用于**Win32**的面向对象的**模板库**，主要被作为MFC的**轻量化替代**，以**ATL**为基础
   2. WTL基于ATL，是对Win32界面API的再封装
3. 安装IDE，新建项目
   1. IDE：VS Code(http://www.vscode.org/)
   2. 创建新项目(Create React App)：npx create-react-app [appname]

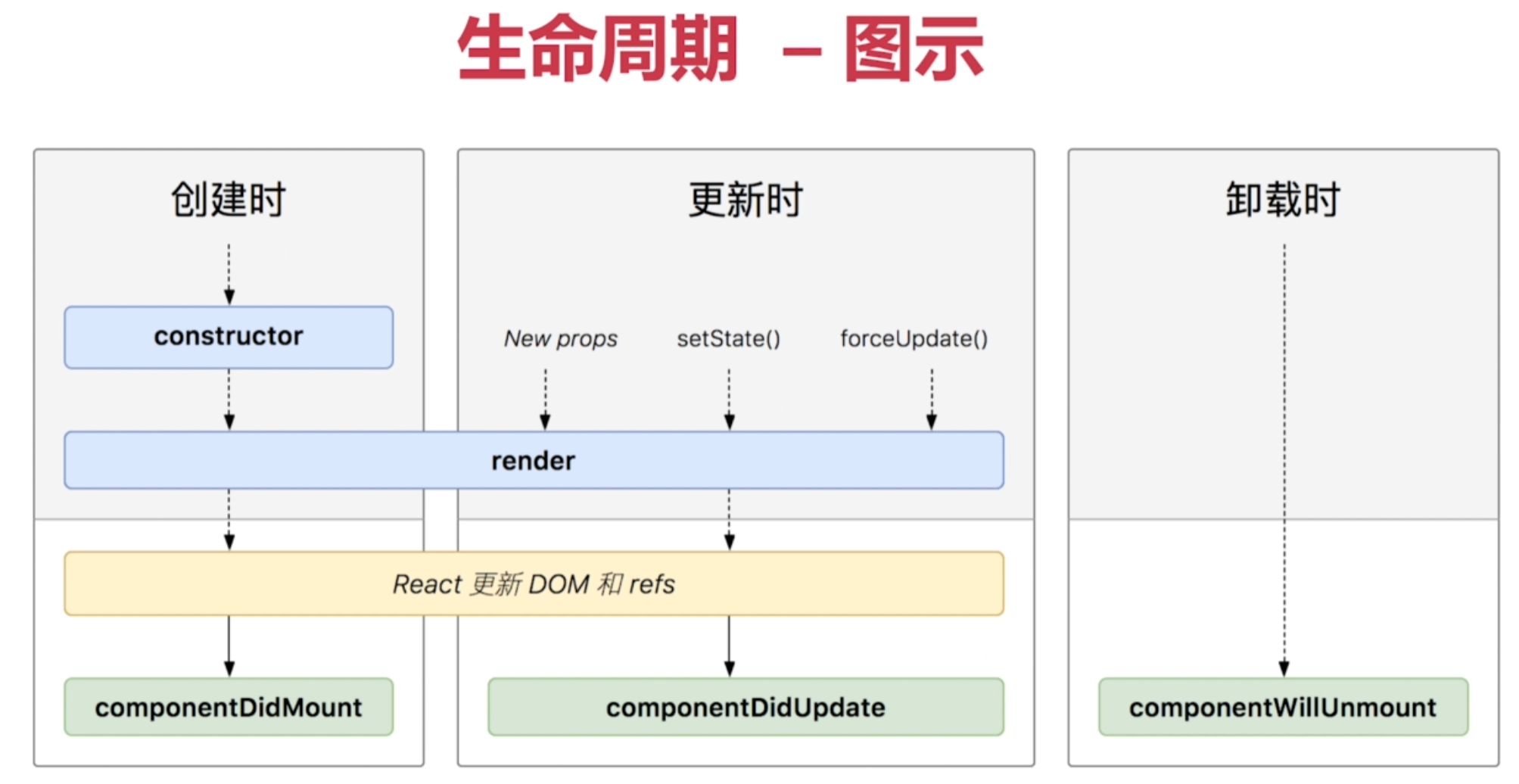
（npx是npm 5.2+附带的package运行工具, create-react-apps脚手架工具）

* 1. 运行项目：npm start
  2. 查看结果：loacalhost：3000

1. 工具链
   1. 单网页项目：Create React App
   2. Node.js构建的服务端渲染的网站：Next.js
   3. 构建面向内容的静态网站：Gatsby
   4. 打造组件库或将 React 集成到现有代码仓库，尝试更灵活的工具链
      * **Neutrino**把webpack的强大功能和简单预设结合在一起。包括了React应用和React组件的预设
      * **Nx**是针对全栈monorepo的开发工具包，内置了React，Next.js，Express等
      * **Parcel**是一个快速的、零配置的网页应用打包器，可以搭配React一起工作
      * **Razzle**是一个无需配置的服务端渲染框架，提供了比Next.js更多的灵活性

# Architecture

React是一个**声明式**的用于构建用户界面的JavaScript库。使用React可以将一些简短的**独立代码片段**组合成复杂UI，这些代码片段被称作**组件**。React中有**多种不同类型**的组件。

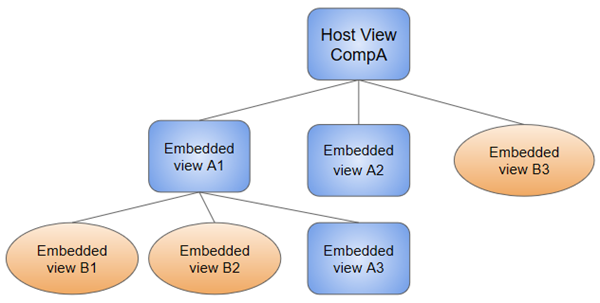


# Module

Angular是模块化的，其模块化系统被称为NgModules。NgModule作为容器，包含一些组件、服务或其他代码文件，专注于某个应用领域、工作流或一组紧密相关的功能。NgModule可以导入其他模块中导出（export）的功能，也可将自身功能导出（export）供其他模块使用。

NgModule是一个带有@NgModule()装饰器的类，@NgModule()是一个装饰器函数，接受一个元数据对象，该对象的参数用于描述当前模块的属性：

* **declarations：**声明属于当前NgModule的组件、指令和管道
* **exports：**可在其他模块的组件模板中使用的可声明对象
* **imports：**导入当前NgModule所需的其他模块
* **providers：**创建可被模块中任意组件使用的服务（通常在组件层次，按需指定服务）
* **bootstrap：**应用主视图，作为其他所有视图的宿主，仅限根模块设置



NgModule为属于它的component提供编译上下文环境。**根模块**至少拥有一个**根组件**，并在引导启动时创建实例。但其他模块可以拥有任意数量的额外组件，这些额外组件可以通过路由加载或通过模板创建。

组件与模板共同定义视图与视图的层次结构。一个视图的层次结构中可以混合使用来自不同NgModule的视图。当一个component被创建时，它会与一个host view直接关联。Host view可以作为视图层次结构的根，包含一些embedded view。这些embedded view同时作为其他组件的host view。视图层次结构可以嵌套到任意深度。

# Component

**视图**通常为屏幕上的一块区域，由模板与component定义和控制。**模板**也是一种HTML，它会告诉Angular如何渲染component。而component中定义了应用的逻辑，为视图提供支持，并通过由属性和方法组成的API与视图交互。用户浏览应用时，Angular会**创建、更新、销毁**相关组件。因此，可通过可选的**生命周期钩子**(例如NgOnInit())在指定时刻执行某些操作。

@Component()装饰器会将普通类修饰为组件类，并为其指定元数据。最终将模板同组件关联起来，共同描述一个视图。@Component()的基础元数据如下：

* **selector：**选择器，通过标签标记组件。当Angular在HTML中找到选择器对应的标签时，会创建并插入一个该组件的实例
* **templateUrl：**与组件关联的模板相对位置，定义了组件的宿主视图
* **providers：**当前组件所需的服务列表

视图通常按层次进行编排，使得开发者可以将UI的一部分或整个页面作为一个单元进行修改、显示或隐藏。与组件直接关联的模板定义了该组件的host view，其它层次中包含了一些embedded view。这些embedded view也作为其他组件的宿主。

图片包含 文字

描述已自动生成

# Service

Angular中，服务通常指一个狭义的类，它具有明确的目的，并执行一些具体的功能。通过将与视图无关的功能分离出来作为服务，可以使组件类简洁、高效。组件与服务的区分还可以提高程序的模块化水平与可复用性。服务可以依赖其他服务。

Angular框架中可采用**依赖注入**为新建的组件提供服务或其他所需的依赖（**函数、变量**等）。通过**@Injectable()装饰**简单类，以便Angualr将该类标识为依赖并注入进其他组件或服务中**（6.0版本以前，注册在NgModule中的providers中）**。**注入器**是依赖注入的核心，Angular在启动过程中会**自动创建**一个全局注入器和所需的额外注入器，该过程不需要用户手动创建。注入器的主要功能包括（1）**创建依赖**；（2）维护一个保存依赖实例的**容器**。容器中的实例将尽可能被复用。

采用急性加载的Angular应用中，根注入器会让所有服务提供商在对整个应用有效。而当使用惰性加载时，Angular创建一个新的注入器，作为根注入器的子元素，最终形成一棵树：应用有一个唯一的根注入器，每个惰性加载模块都有一个自己的注入器。路由器会将根注入器中的服务提供商注入到子注入器中。

当Angualr创建组件类新的实例时，会检查该组件类的**构造函数参数**，以确定该组件依赖的服务或其他依赖项。检测到依赖时，Angular首先会检查注入器的容器中是否有该依赖的实例，如果实例不存在，注入器会创建一个新的依赖实例，并把该实例返回给Angular。当所有请求的依赖已解析并返回时，Angular会把依赖实例作为**参数**，调用组件的构造函数。而在惰性加载模块中创建组件时，都会获取服务的局部实例，而不是根注入器中的实例。而其他模块仍会使用根注入器中的服务实例。

除依赖注入外，服务还可以通过装饰器中的**Provider**（@ngModule/@component）或**New方法**创建服务实例，该类服务的生命周期与module或组件的生命周期保持一致。

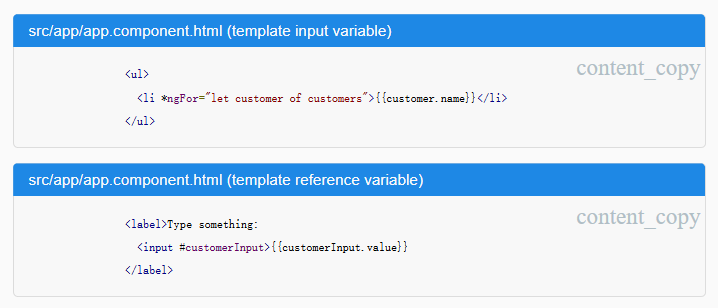
# 组件与模板

## 模板

可使用两种方法存放组件模板：（1）使用**template**属定义**内联模板**，并在ts中编写html片段；（2）通过@Component装饰器中的**templateUrl**属性，将定义在独立的HTML文件内的**模板文件**，加入到组件元数据中，从而实现模板文件与组件的链接。

**插值表达式**{{…}}可将变量的值合并到标签之间或者标签的属性中如<p>{{ var1 }}</p>或<img src=”{{ var2 }}”>,夹在{{}}中间的表达式被称为**模板表达式**。模板表达式不能引用全局命名空间中的任何属性和方法，包括window、document、console.log、Math.max等。

**表达式上下文**可以包含组件内、组件外（**模板输入变量**“let cunstomer”和**模板引用变量**“#customerInput”）的属性或对象。



**attribute是由HTML定义的**，HTML标签中的属性特征，**property是由DOM中定义的**，DOM元素对象中的属性值。当浏览器渲染时，会**根据attribute创建对应的DOM节点**。attribute对应初始值，property对应当前值，部分attribute与property间存在映射。

**属性绑定**需要使用**[]**标记**目标属性**，或用**bind-前缀**的形式。而在绑定过程中，出现在=右侧的模板表达式应返回恰当的类型(对象、字符串、数字)，如果绑定过程中忘记了[]，模板表达式则会被**默认当作字符串处**理。**attribute、css、style**的绑定则通过[attr.attr-name]、[css.css-name]、[style.style-property]进行绑定。

**事件绑定**允许监听某些事件，通过**()**标记**目标事件，**或采用**on-前缀**的形式。=右侧的模板表达式将在事件发生时执行。事件绑定会通过**$event**传递事件信息，目标事件决定了事件对象：如果目标事件是DOM元素的原生事件，$event就是DOM事件对象。Angular通常通过EventEmitter触发自定义事件，任何函数、方法都可作为EventEmitter.emit(payload)的载荷，并通过$event对象访问该载荷。

angular使用内置**结构型指令**(**ngIf**、**ngFor**、**ngSwitch**)调整HTML布局，通过**添加**、**移除**或**操作宿主元素**实现DOM节点的构造或重构。ngIf可以控制DOM元素的**创建**和**移除**，通常也用于防范空指针错误；ngFor是**重复器**指令，它首先定义了一个HTML片段，规定了单个条目应该如何显示，再以该片段为模板，渲染条目列表中的每一个条目，还可通过**index**属性返回一个从0开始的**索引**；**trackBy**则帮助ngFor仅**重构**发生变化的DOM元素，而不是重新渲染整个列表。

使用**组件内**的属性可通过**模板引用变量(#var/ref前缀)**引用某个DOM元素(类似ID)；使用**其他组件**的属性则可通过**输入输出属性(@input/@output)**，@input()和@output()装饰器可以**显示的声明**自身那些属性可以**被外部组件绑定**。

## 用户输入

当用户点击链接、按钮或输入文字时，都会产生**DOM事件**。可通过**$event**对象将事件载荷发送给组件的事件处理器，但该方法不被提倡，因为$event对象具有复杂结构，且组件可通过其得到过多模板信息，这违反了模板(用户看到)与组件(处理过程)分离的原则，因此angular通常采用**模板引用变量(#var)**。通过模板引用变量可以得到干净简单的数据。

## 生命周期钩子

Angular管理每个组件的生命周期：**创建**、**渲染**组件与其子组件；绑定的属性变化时**检查**组件；在从DOM树中移除前**销毁**组件。Angular通过**生命周期钩子**将组件生命周期的关键时刻暴露，以便用户在某一阶段执行自定义功能。





## 组件交互

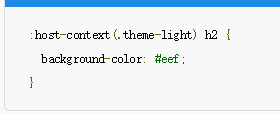
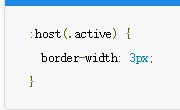
组件间交互分为从**父组件到子组件**与**子组件到父组件**两类，父组件可通过**@input()**设置子组件的输入属性，**输入属性**会被**ngOnChanges()**监听。父组件可通过@output()监听子组件的输出，输出属性会从子组件通过**emit(value)**发射至父组件。父组件不可使用数据绑定读取子组件的属性或调用子组件的方法，但可通过新建**本地变量**(#var)代表子组件或通过**@ViewChild()**读取和调用子组件的方法。这两种方法的区别是本地变量仅能在**父组件模板**中使用，组件本身无法访问，而@ViewChild会将子组件注入到父组件中，供**组件类**使用。

另一种组件交互的方法是通过服务、流、订阅、Observable进行组件间通讯。

## 组件样式

伪-类选择器(**:host/:host-context**)：将宿主元素/宿主祖先作为目标，以宿主/祖先样式作为条件，将常规选择器放入括号中。

当**宿主元素带有.active** CSS类时border-width生效；当**宿主及祖先节点**（直至根节点）带有.theme-light CSS类时background-color在h2元素中生效。



## 自定义元素

自定义元素是一款**Web标准**，可用于定义新的**HTML元素**，使用自定义元素需要浏览器的支持。

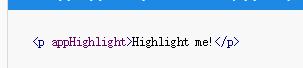
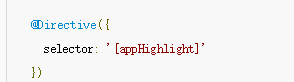
在用户配置自定义元素后，首先需要在浏览器中**注册**该元素类，接下来就可以像使用内置HTML元素一样使用该新元素。自定义元素会在添加到DOM时自动启动，从DOM移除时自行销毁。

Angular内部提供了**createCustomerElement()**函数，支持将组件及其依赖打包转换为自定义元素。转换过程中实现了一个NgElementConstructor接口和一个**构造器类**，用于生成该组件的self-bootstrapping实例。接下来通过javaScript的**customElements.define()**函数将配置好的构造器及自定义元素标签注册到浏览器的**CustomElementRegistry**中。当浏览器检测到该已注册元素标签时，使用构造器创建自定义元素实例。

## 动态组件

## 属性型/结构型指令

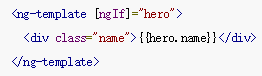
Angular中包含三类指令：（1）**组件**，拥有模板的指令；（2）**属性型指令**，改变元素、组件或其他指令的外观和行为的指令（内置ngStyle或自定义指令）；（3）**结构型指令**，添加、移除DOM元素改变DOM布局的指令(ngFor、ngIf等)。



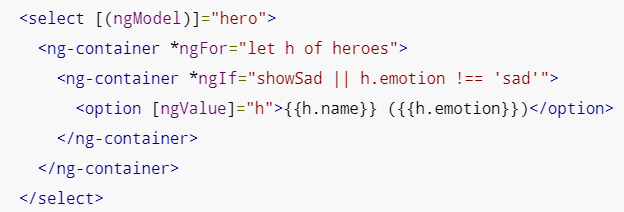
**@Directive()**将一个类装饰为指令，**[]**表示属性选择器，指定选择器名称后可在模板的元素中直接使用。当输入属性(@input(‘name’)) 与选择名称一致时，可以通过一个名称实现指令绑定和属性输入



使用内置的结构型指令时，通常需要一个**\*前缀**，这是一个简化复杂语法的语法糖，其内部实现是将\*翻译为一个**<ng-template>**元素。并用它包裹住宿主元素。<ng-template>及其内容在渲染前会被**替换为注释**，因此是不可见的，只有在ng-template中使用**结构型指令**时，它才会正常工作，显示内部的元素。

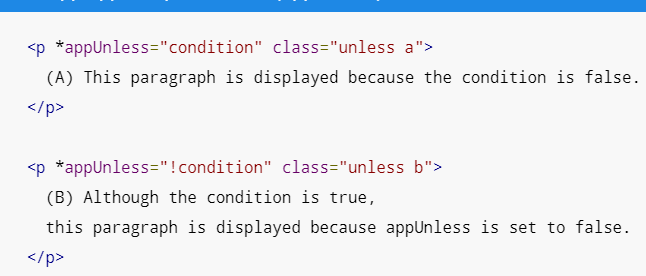


**<ng-container>**也是一个angular元素，可以作为结构型指令的**宿主**，有些时候使用源生HTML元素可能会**破坏模板的样式**，而<ng-container>则不会污染样式和元素布局。



自定义结构型指令时，@Directive()、@Input()、TemplateRef、ViewContainerRef是必选属性，并在模板中标识出该指令应用的元素。





## 管道

Angular引入一种**可在HTML中声明**的**管道**对数据进行**转换**，以得到**期望输出**。管道工作时，插值表达式中的变量会通过**管道操作符( | )**流动到右侧的**管道函数**中。Angular中内置了一些管道，如：DatePipe、UpperCasePipe、LowerCasePipe、CurrencyPipe和PercentPipe，它们可以直接用在任何模板中。

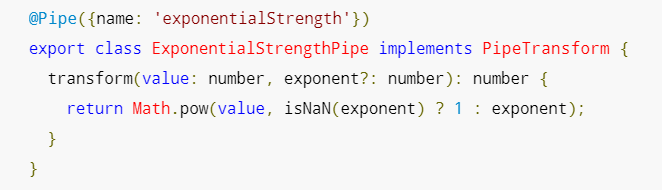
管道可通过**参数化（接收可选参数）调整**输出的**格式**：在管道名后**添加冒号( : )**，之后**紧跟一个参数**，如该管道可接收**多个参数**，则用**多个冒号**分隔这些参数。传入管道的参数可以是任何有效的模板表达式**（字符串、组件属性）**。**链式管道**则可以将多个管道串联在一起，以完成一些潜在功能。







* **自定义管道**



管道是一个通过**管道元数据**装饰的类，实现了**PipeTransform接口**的**transform方法**。该方法接受一个**必需的输入**和一些**可选参数**，并返回转换后的值。@Pipe()是管道装饰器，其中的name定义了管道的名称，模板表达式中可通过该名称引用管道。使用自定义管道时，需要将其添加到AppModule中的declarations中，如果选择将管道注入（inject）类中，则必须将管道添加到NgModule的providers中。

管道使用了另一种**变更检测算法**，该算法会**忽略**对象**内部的更改**。因此，某些情况下（.push），由于管道引用的对象没有变化，显示也不会更新。此时可以通过**创建**一个新的**引用**，如：创建一个新的数组，并把它赋值给原有变量，以刷新显示。

为解决上述问题，Angular还提供了**纯管道**与**非纯管道**。默认情况下，所有管道都是**纯管道**，**pure标志为true**，此时只有发生**纯变更**时才会执行纯管道**（纯变更是指对原始类型值(String、Number、Boolean、Symbol)的更改， 或者对对象引用(Date、Array、Function、Object)的更改）**。Angular 会**忽略(复合)对象内部的更改**。 如果你更改了输入日期(Date)中的月份、往一个输入数组(Array)中添加新值或者更新了一个输入对象(Object)的属性，纯管道都不会被调用。纯管道保证了程序的**执行速度**。

非纯管道会在组件周期性探测变化时执行，频繁的被调用多次。因此非纯管道是非常昂贵的。可以通过带缓存的非纯通道来减少请求次数：只有当请求的URL发生变化时才发起请求。除此以外，Angular提供了非纯管道AsyncPipe来处理Observable和Promise的输入，通过订阅，异步返回他们的值，该管道是有状态的。



# 表单

表单是Web应用中的一项基础功能，通常用来执行**数据输入**任务。angular提供了两种不同的表单用于处理用户输入：

* **响应式表单**：健壮，可扩展性、可复用性、可测试性强
* **模板驱动表单（ngModel）**：结构简单，易添加，不易扩展

两种表单都从视图中捕获用户输入事件，验证输入，创建表单模型，修改数据模型并提供追踪这些更改的途径，但他们在如何处理、管理表单和表单数据时有所不同，各有优势，一些关键差异点如下图所示：



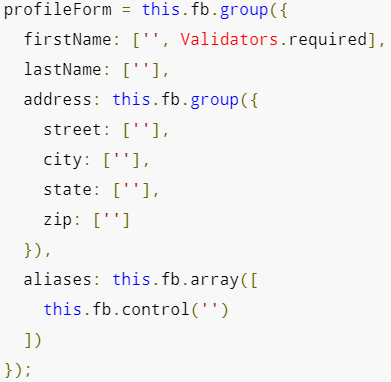
**追踪数据变更**是衡量数据**可变性**的一种重要方式。响应式表单中，数据模型是不可变的。每当触发变化时，FormControl实例会返回一个**新的数据模型**，而不是修改已有的数据模型。因此用户可以通过控件的可观察对象追踪唯一的数据变更。这种模式下变更检测更高效，可变性强。模板驱动表单使用**双向数据绑定**，每次变化都会**修改数据模型**，因此追踪唯一数据变更时效率较低，可变性弱。

**跨组件复用**表单模型是衡量**可伸缩性**的标准之一。**响应式表单**通过对**底层API**的访问与对**表单模型**的**同步访问**，使得创建大型、可复用表单变得容易。**模板驱动表单**专注于**简单场景**，**不可重用**、将**底层API**进行了**抽象**，而且对表单模型的访问是**异步**的。

## 响应式（模型驱动）表单

响应式表单使用**显示**的、**不可变**的方式，管理表单在某个时间点上的状态。为在变化时维护模型的整体性，每次变更都会**返回**一个**新的状态**。响应式表单是围绕**Observable流**创建的，流的每一个消费者都可以安全的操纵这些数据。

使用响应式表单首先需要在appModule中注册**ReactiveFormsModules**，之后才能够在模板中绑定[formControl]等指令。[formGrop]与formGroupName可以管理一组表单。但手动创建多个表单控件实例十分繁琐，Angular中内置的**FormBuilder**提供了一种便捷方法：(1)导入FormBuilder类；(2)注入FormBuilder服务；(3)生成表单控件。FormBuilder服务包括**control()**、**group()**、**array()**三种**工厂方法**，分别用于生成**FormControl**、**FormGroup**和**FormArray**。**FormArray**是另一种管理表单的方法，通常用于管理**任意数量**的**匿名**表单。



## 模板驱动表单

Angular框架中提供了一种模板驱动表单方法(ngModel)，该方法基于模板，结合特定的表单指令与技术构建表单，支持双向数据绑定、变更检测、验证和错误处理。使用模板驱动表单前，需将**FormsModule**导入appModule中。在**表单中**使用[(ngModel)]时，**标签**必须定义**name**属性；此时的ngModel还可以通过**CSS类**跟踪控件**状态**。



## 表单验证

模板驱动表单可通过**源生验证器**或**自定义指令**进行验证。而在响应式表单中，其数据源是**组件类**，因此应直接将**验证器函数（内置/自定义）**添加到**表单控件模型**上，而不是模板的属性中。验证器函数分为两种：

* **同步验证器**：接受一个控件实例，返回一个验证结果列表。该验证器可在实例化FormControl时作为构造函数的第二个参数传入表单控件中
* **异步验证器**：接受一个控件实例，返回一个Promise或Observable，稍后会发出一组验证结果列表。该验证器可在实例化FormControl时作为构造函数的第三个参数传入表单控件中

处于**性能考虑**，只有在所有**同步验证器**都通过之后，angular才会运行**异步验证器**，当所有异步验证器运行完毕后才会返回**验证结果列表**。

## 动态表单

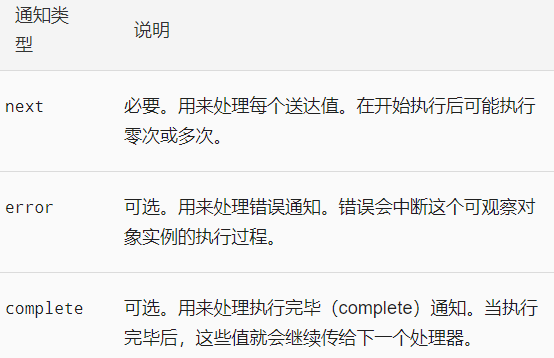
# Observable&RxJS

## Observable

Observable可以在发布者和订阅者间**传递消息(异步)**。在进行**事件处理**、**异步编程**和**处理多个值**时，Observable有显著的优势：(1)可发送**任意类型**的值（字符、消息、事件）； (2)创建、销毁的逻辑由Observable对象自身处理，用户只需执行**订阅**和**取消订阅**操作；(3)接收值的**API相同**，无论时**同步或异步**；(4)对流进行**监听和取消监听的API**一致，无论是键盘流、HTTP响应流还是定时器。Observable是**声明式**的，只有当消费者**订阅它时**，定义的函数**才会真正执行**。订阅后，当函数执行完毕或取消订阅时，订阅者会收到通知。

发布者会创建一个**可观察对象(Observable)**，该对象实例中包含一个**订阅者函数（subscriber）**。执行Observable并从中接收通知时，首先需要消费者调用**subscribe()**方法，同时为其传入一个**观察者对象**（**observer**，这是一个JavaScript对象，定义了接收消息的处理器）。subscribe()函数会返回一个Subscription对象，该对象具有unsubscribe()方法，调用后停止接收通知。

**观察者对象(Observer)**提供多种消息处理器（handler）接口，用于接收**可观察对象(Observable)**的消息。observer通过**回调函数**处理observable可能会发来的**三种通知**，处理三种通知的处理器可以任意组合，如果某一类型通知未被提供处理器，该类通知将会被obserber**忽略**（next处理器是必要的）。Next()函数可以接收字符串、数、事件对象等**各种结构**，因此由**可观察对象(observable)**发出的数据通常统称为**流**。因此**任何类型**的值都可以作为可观察对象，这些值会被发布为一个**流**。



默认情况下，可观察对象(observable)会为每一个订阅它的观察者创建一个新的、独立的事件处理器，并重新发送消息。通过**多播(Multicasting)**可以将可观察对象的一次执行**广播**给**多个**订阅者，此时不必注册多个监听器，而是**复用第一个监听器**，把值发送给各个订阅者。

## RxJS

响应式编程是一种面向数据流和变更传播(propagation of change)的异步编程范式。RxJS(Reactive Extensions for JavaScript, JavaScript的响应式扩展)是一个使用可观察对象(observable)进行响应式编程的库，提供了对Observable类型的实现与一些工具函数：（1）把现有异步代码转换成可观察对象；（2）迭代流中的每个值；（3）把值映射为其他类型；（4）对流进行过滤；（5）组合多个流。在Observable成为Javascript的一部分并且得到浏览器支持前，RxJS是必需的。



除了在订阅时提供的error()处理器外，RxJS还提供了catchError操作符，允许用户在管道中处理已知错误。同时，catchError还提供了一种简单的方式进行恢复，retry操作符允许用户重试失败的请求。

## Angular中的可观察对象

Angular使用可观察对象作为处理常用异步操作的接口，例如：（1）EventEmitter类派生自Observable；（2）Http模块通过Observable处理AJAX请求和响应；（3）路由器和表单使用Observable监听用户对输入事件的响应。

## 与其他技术比较

**Observable**与**Promise**通常放在一起比较，他们有一下不同的关键点：

* Observable是声明式的，被订阅之前不执行，可以被**多次调用**，每次**独立**、**从新计算**；Promise在创建时**立即执行**，无法重新计算，所有then语句**共享**一次**计算结果**
* Observable能提供**多个值**（随时间推移），Promise仅能提供**一个**
* Observable会区分**串联和订阅**，Promise中最后的then()等同于订阅，中间的then()等同于映射；Observable可取消，Promise不可取消
* Observable自身能够处理错误，便于进行集中、可预测的错误处理，Primise把错误推送给subPromise

# 引导启动&NgModule

每个Angular应用中，至少有一个module。通常来讲**root module**是**引导（bootstrap）启动**Angular应用的入口，它的默认命名为AppModule。Angular使用**模块**将代码**组织**到一起并进行**管理**，但同时**也依赖**于**JavaScript**的组织、管理方式。因此，在一个module中**同时包含import/export与NgModule**两种结构。

## Import/export

在JavaScript中，模块是内含JS代码的独立文件，若要其中的内容可被外部使用，需要使用export语句导出，当需要使用其他文件时，需要import语句引入。



## NgModule

模块是**组织应用**与使用外部库对应用**进行扩展**的最佳途径，Angular自身的库以及部分第三方库都属于NgModule，如FormsModule、HttpClientModule、RouterModuel等。NgModule是一个带有**@NgModule装饰器**的类，通常在import语句之后（**在Ngmodule中声明指令、组件、管道前，需要import它**）。

@NgModule获取了一个元数据对象，它会告诉Angular如何组织、编译、启动应用：

* **Declarations(Static)**：**声明**属于该模块的**可声明对象**，包括：组件、指令、管道。每个组件只能声明在**一个**NgModule类中，仅在当**前模块可见**。
* **Provider(Runtime)**：声明**服务**，并指定**应用范围**
* **export(Composability)**：**公开**部分可声名对象，以便**其他模块**中的组件**使用**（其他模块需**先引用**本模块）
* **import(Composability)**：导入**依赖**的其他模块
* **bootstrap(EntryComponent)**：**引导启动**根模块，通常**只存在**于AppModule中。

## 常用模块



## 特征模块

特征模块通常分为**5种常用类型**：（1）领域特征模块；（2）被路由（Routed）的特征模块；（3）路由（Routing）模块；（4）服务特征模块；（5）窗口组件特征模块。日常使用中，经常会出现**多种类型混合**的特征模块。

|  |  |
| --- | --- |
| 特征模块 | 指导原则 |
| 领域模块 | 提供单一功能，由一组可声明对象组成，仅顶层(根)组件被导出  如引入服务，则服务的生命周期应与该模块相同  通常由更高等级的特征模块导入，且仅导入一次 |
| 被路由的模块  （Routed） | 一种特殊的领域模块，顶层组件作为路由导航时的目标组件  惰性加载模块都是被路由的模块  仅引用生命周期与模块同步的服务，避免导入全应用级的单例服务  惰性加载模块不应被任何其他模块导入，以避免导致急性加载 |
| 路由模块  （Routing） | 为其他模块提供路由配置，并将路由功能从配套模块分离出来  定义、配置和解析路由，仅被配套模块导入  没有可声明对象（其他特征模块的职责） |
| 服务模块 | 提供工具服务，完全由服务供应商组成，不包含可声明对象  根模块（AppModule）是唯一可导入服务模块的模块 |
| 窗口组件模块  （widget） | 主要由可声明对象组成，为外部模块提供组件/指令/通道(被导出)  大量第三方UI组件库为窗口组件模块 |

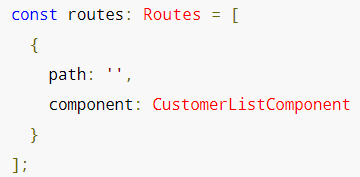
## Entry Component

组件可以分两种类型：（1）命令式加载（NgModule中bootstrap引导或路由中定义）；（2）声明式加载（在template模板中引用）。采用命令式加载的组件被称为入口组件（entry component）。入口组件可分为两种类型：

* + **引导用根组件（bootstrapped root component）：**Angular会在应用**引导启动**过程中将其加载到DOM中，其他入口组件在**启动后**某一时刻**动态加载**。



* + **路由中指定的组件:**

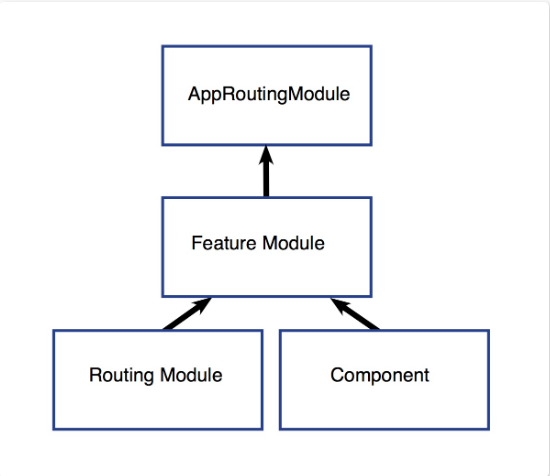


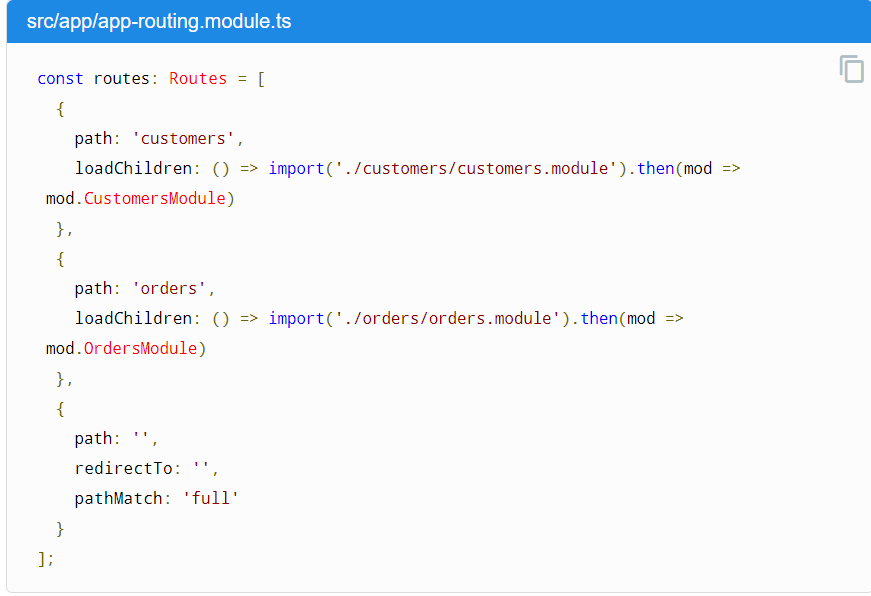
虽然@NgModule装饰器包含一个**entryComponent数组**，但通常不需要用户显示设置入口组件，因为Angular会自动把bootstrap中的组件以及路由中定义的组件添加为入口组件。如果用户使用一些其他方式引导或动态加载入口组件，则需要显示的将其添加进entryComponents中。

## 惰性加载

默认情况下，Angular加载NgModule采用**急性加载**模式，应用启动时会**立即加载**，无论该模块是否被使用。对于带有很多路由的**大型应用**，应考虑采用**惰性加载**：一种**按需加载**NgModule的模式。该模式可以减小初始包的大小，从而缩短初始加载时间。

惰性加载依赖路由模块，通过在AppRoutingModule中配置路由，将需惰性加载的特征模块连接到路由模块中，实现到特征模块的跳转。





手动

# Unit Test

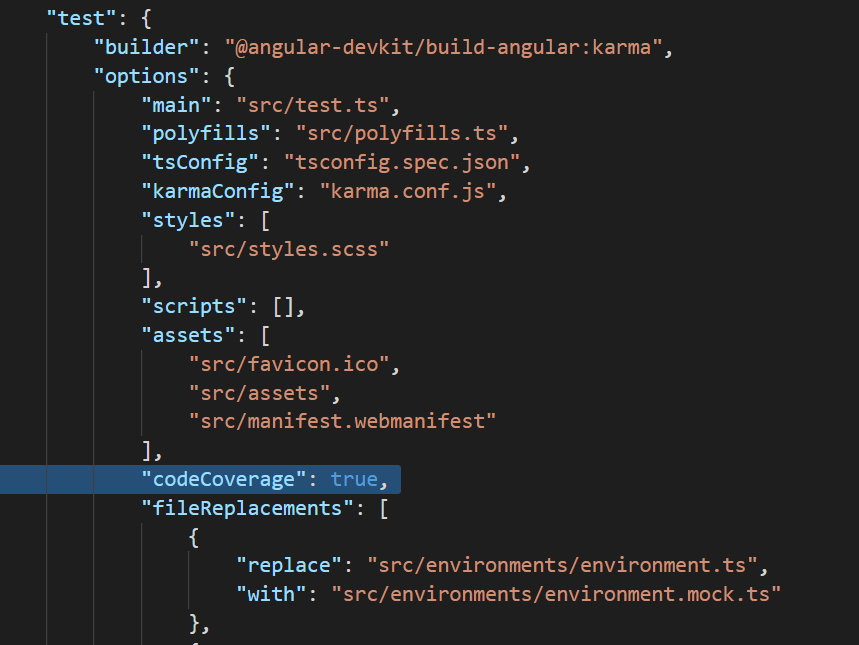
Angular CLI会**自动下载、安装Jasmine**与**Karma**。使用**ng test**，可以在**监视模式**下构建Angular应用，并启动Karma运行测试用例，。

Angular CLI会自动生成Jasmine与Karma的配置文件，可通过修改**angular.json、test.ts、karma.conf.js、tsconfig.spec.json**文件，**调整测试环境**。

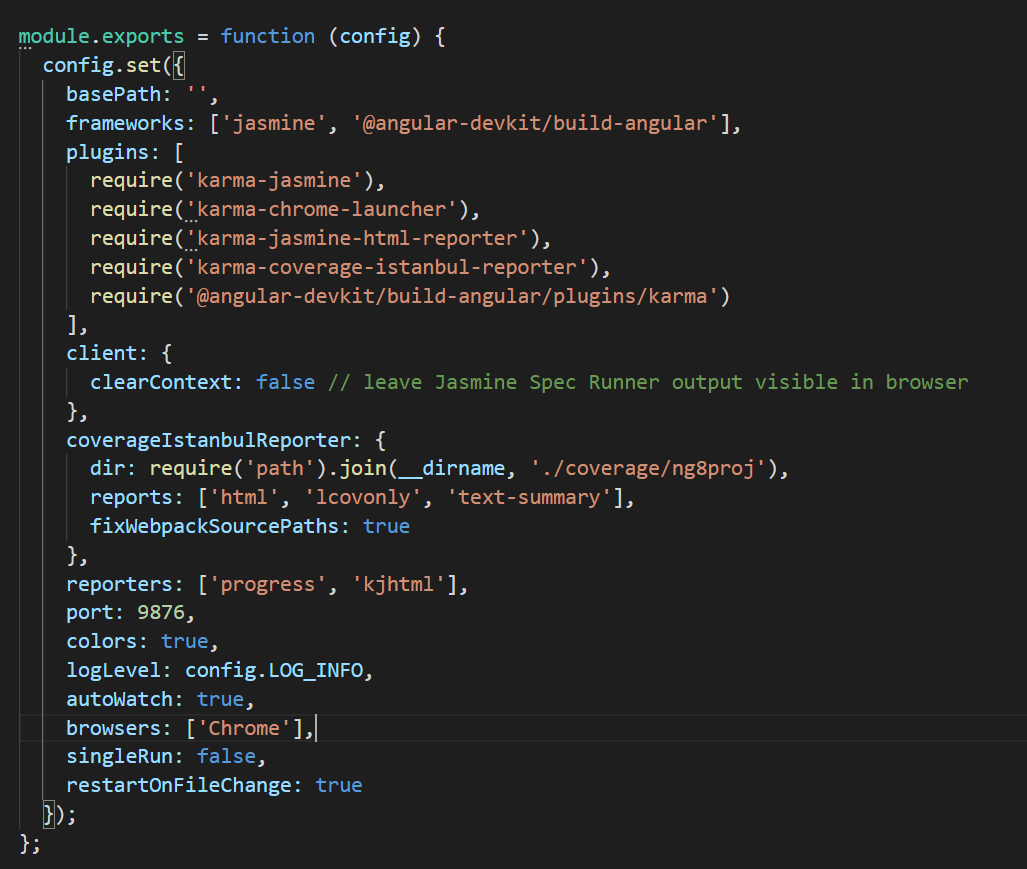
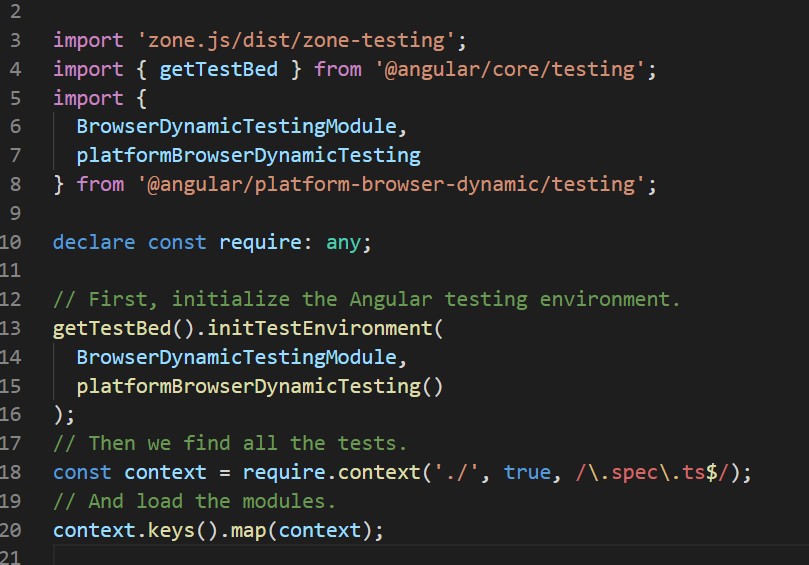
## 测试环境

Karma是测试的启动者（Test Runner），是一个测试执行过程的管理工具，可通过配置文件设置测试基于的浏览器、测试框架、入口文件、测试报告等。Jasmine是执行测试的Javascript框架。

* Angular.json将tsconfig.spec.json、karma.conf.json与test.ts进行关联，还可调整覆盖率报告等相关设置；
* tsconfig.spec.json可设定编译选项，包括测试文件的目录等；



* Karma.conf.js是有angular-cli创建的karma配置文件，其中可以修改测试框架(jasmine、Angular-devkit)、测试报告、浏览器等信息。
* Test.js是测试的入口，可以根据规约修改检索测试文件的(正则)表达式，默认单元测试文件的检索表达式为**require.context('./', true, /\.spec\.ts$/);**三个参数分别代表**检索路径**、是否**检索子目录**、**表达式**

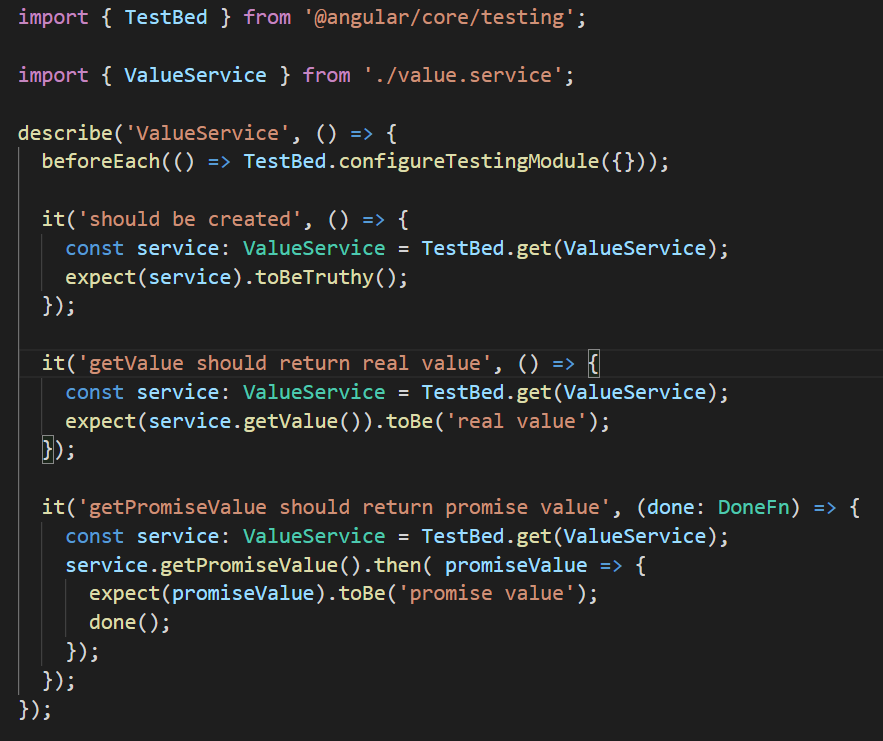
 

## 测试框架

Angular的测试工具可分为两类：第三方测试框架(jasmine)、内部工具(Angular-devkit-Angular测试工具集)。对于Pipe、Service这种相对简单的类，可直接通过new创建实例进行测试；而对于指令、组件，需要一定的环境才能够正常运行，例如组件需要被包含在一个@NgModule中。此时就需要Angular测试工具集构建相应环境。

## Jasmine

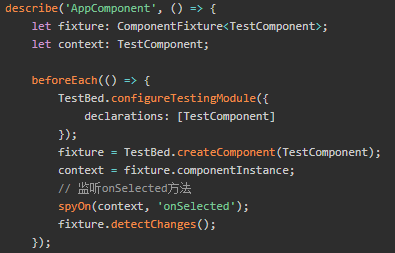
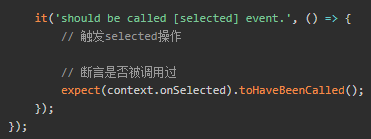
在测试中，一个简单类也会包含有多个测试用例。为方便管理，通常将一类测试用例集成在某一个集合下，这种集合被称为**Test Suite**。在Jasmine中，使用**describe**实现Test Suite，它的第一个参数表示该Suite的名称，第二个参数为一组测试用例代码，describe可嵌套使用，提高代码的可读性。测试用例被称为Specs，在Jasmine中使用it表示，每个it对应一个测试用例。一个测试用例中可能包含一个或者多个断言Expectations，Jasmine中使用expect表示。只有当所有expectations全部为true时，该测试用例才为true。有一个expectation为false时，该测试用例将返回失败。除此以外，Jasmine还提供了丰富的API。



为保持代码的简单与整洁，可以将重复的代码或全局变量放在beforeEach与afterEach函数中，beforeEach将在每个Spec之前执行，afterEach则相反。当某些Test Suit或Specs暂时不需要时，也无需删除或注释，可使用xdescribe和xit暂时跳过他们。

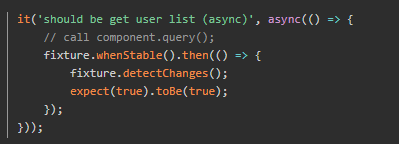
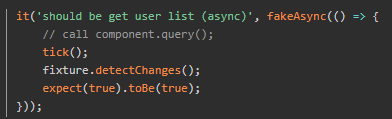
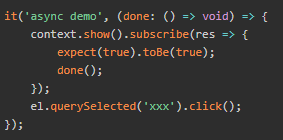
## Angular测试工具集

Angular测试工具集中最常用模块的包括TestBed、Spy和异步。TestBed用于构建测试环境下的@NgModule模块；之后可通过TestBed.createComponent创建用于测试的目标组件。

Spy用于监控函数是否被调用。

Angular中的异步通常指带有Observable或Promise的异步行为，通常在组件调用某个Service以获取数据时使用。异步测试可利用工具集提供的async与fakeAsync。Async可通过whenStable()函数在所有待处理的异步行为全部完成后再执行回调和断言；fakeAsync则使用tick()函数替代了回调。除此以外Jasmine还提供了done()方法，用以处理Observable、Promise以外的异步行为，例如setTimeout、外部订阅等。

## 测试示例

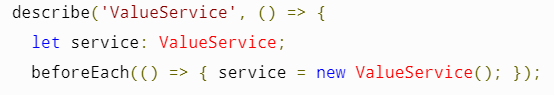
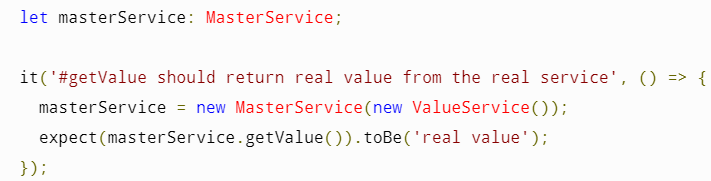
## beforeEach

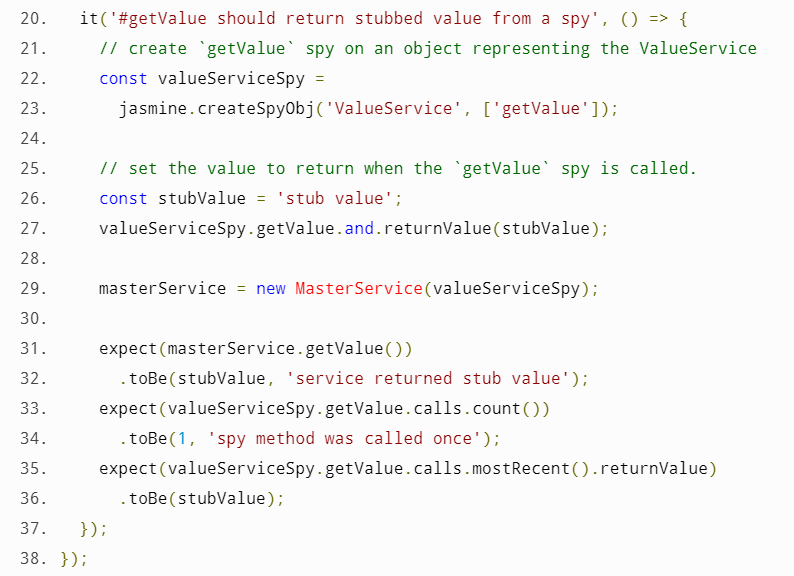
beforeEach可以将重复的工作集中在一起，对测试进行初始化和引导，例如在beforeEach中使用TestBed创建用于测试的NgModule，并为其配置依赖等参数。

## 测试Service

Service是Angular单元测试中最简单的文件类型。首先声明对应类型的成员变量，接下来通过beforeEach()进行初始化，最后通过it()逐条执行测试用例。

部分Service(带有依赖的Service)会通过constructor()注入被其依赖的其他Service，这类Service进行单元测试时，可通过new方法，直接注入相关依赖。但在某些情况下，直接注入真实Service可能会导致额外问题，因此还可以通过模拟的方式解决依赖：（1）使用虚值（2）在相关Service方法上创建spy，使用spy是Mock服务时最简单的方式。



上述方法可以在隔离环境下对Service进行单元测试。但无法反映出Angular的依赖注入机制，因此Angular测试工具集中提供了TestBed以解决该问题。TestBed是工具集中最重要的部分，可以动态创建一个用来模拟@NgModule的测试模块。TestBed.configure TestingModule()方法将一个元数据对象作为参数，包含了@NgModule中的绝大多数属性。通过TestBed测试工具来提供和创建Service时，要注意构造函数中的参数顺序，并在TestBed.configure TestingModule()的**Providers数组**中指定需要进行测试或模拟的相关**Service(不在providers中声明service也可使用？？？)**。执行测试用例时，通过**TestBed.get()获取Service实例**。（TestBed.inject()需要在angular9.0以上版本使用，且只有通过@inject到root注入器的服务可用）

## 测试Component

Angular中的组件由**HTML模板**和多个**TypeScript类**组成。需要测试它们能否按预期**协作**：（1）在浏览器的DOM中创建组件的宿主元素；（2）检查组件类与DOM的交互；（3）某些情况下可单独测试组件类而不涉及DOM。

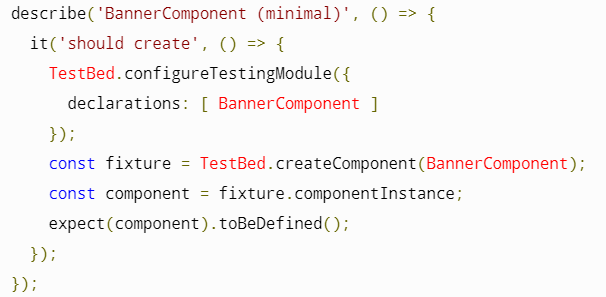
* **单独测试组件类**

组件类的单元测试与Service相似，对于没有依赖的组件类，可以通过new创建对象，进而调用API。遇到有依赖的组件类时，需要使用TestBed同时创建该组件及其依赖。

* **结合DOM的测试**

组件可分为组件类、与DOM的交互和组件间的交互三部分。仅涉及类的测试可以验证组件类的行为是否正常，但无法验证是否正常渲染，是否与其他组件正确集成。为验证组件类与DOM的复杂交互，需要使用TestBed的附加功能或其他测试工具。

与测试service不同，需要在TestBed.configure TestingModule()的**Declarations数组**中声明待测**Component**；使用**TestBed.createComponent()创建component实例**并把相应元素**添加到DOM**中**，**最终返回一个**ComponentFixture对象**；ComponentFixture包含一组测试套件（test harness），可通过**fixture**与component（fixture.componentInstance）和DOM（fixture. nativeElement）元素交互。



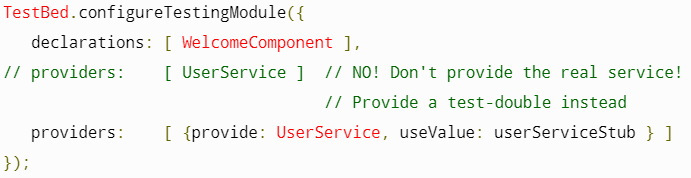
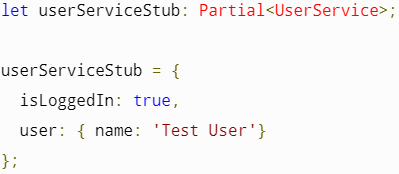
## component的测试场景

### 组件绑定

Angular在展示静态内容时，可直接渲染DOM；但在渲染**动态内容**时，**无法立刻展示**绑定的元素，需要执行**变更检测**，才会将期望内容展示出来。**生产环境**下，变更检测会**自动触发**，但在TestBed.createComponent()中，需要用户使用**fixture.detectChanges()**，**手动触发**TestBed的数据绑定，且每条测试用例仅能**触发一次（环境相关）**。这种**推迟的变更检测是有意设计**的，可给测试者提供一个在数据绑定初始化及调用生命周期钩子前**探查并改变组件状态**的**机会**。Angular还为单元测试提供**自动变更检测**的方法，避免频繁调用detectChanges：导入ComponentFixtureAutoDetect，并添加到TestBed.configureTestingModule()的Providers数组中，但该方法只能对异步行为（Promise、计时器、DOM事件）做出反应，无法在同步更新时触发变更检测，必须再次手动调用fixture.detectChange()，触发新的变更检测。

### 具有依赖关系的组件

组件经常依赖其他服务，但在测试环境中，并不需要注入真正的服务。因为spec的主要目的是测试组件，应避免服务对测试的干扰。因此可在被测试组件中**注入服务替身（Stub）**。



在依赖服务的组件中获取服务实例有两种方法：**（1）fixture.debugElement.injector.get()；（2）TestBed.get()**。由于Angular的注入器是多层次的，由根注入器向下组成一颗树。从被测组件注入器获取服务实例是**最安全且有效**的方法，也就是**第一种方法**；第二种方法更简单，但仅能从根注入器获取服务实例，只在组件是该次测试的根注入器时才能正常工作，因此**TestBed.get()是不安全的**。

### 带有异步服务的组件

针对该类组件，可以使用Spy替代Stub，将异步调用转为同步调用进行检测；也可使用fakeAsync()进行异步测试

### 弹珠测试

针对Observable连续发出多次消息的情况

### 带有输入输出的组件

带有输入输出的组件通常出现在宿主组件的视图模板中，宿主通过属性绑定设置输入值，通过事件绑定监听输出属性触发的事件。该类组件有三种测试方案：（1）将其当作被宿主使用的组件进行测试；（2）创建宿主组件替身，将其作为被宿主替身使用的组件进行测试；（3）当作独立的组件进行测试；

对带有输入输出的组件进行独立测试相对简单，可直接复制模拟输入；测试output则分为两步，首先需要订阅output，接下来使用triggerEventHandler()触发emit相关事件；在宿主中测试带输入输出的组件时，也许通过选择器获得相关元素，并触发emit相关事件。