速查表

引导/启动	`import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';`
`platformBrowserDynamic().bootstrapModule(AppModule);`	用 `NgModule` 中指定的根组件进行启动。

NgModules	`import { NgModule } from '@angular/core';`

`@NgModule({ declarations: ..., imports: ...,`
`exports: ..., providers: ..., bootstrap: ...})`
`class MyModule {}`
	定义一个模块，其中可以包含组件、指令、管道和服务提供商。

`declarations: [MyRedComponent, MyBlueComponent, MyDatePipe]`	属于当前模块的组件、指令和管道的列表。
`imports: [BrowserModule, SomeOtherModule]`	本模块所导入的模块列表
`exports: [MyRedComponent, MyDatePipe]`	那些导入了本模块的模块所能看到的组件、指令和管道的列表
`providers: [MyService, { provide: ... }]`	依赖注入提供商的列表，本模块以及本模块导入的所有模块中的内容都可以看见它们。
`entryComponents: [SomeComponent, OtherComponent]`	任何未在可达模板中引用过的组件列表，比如从代码中动态创建的。
`bootstrap: [MyAppComponent]`	当本模块启动时，随之启动的组件列表。

模板语法	
`<input [value]="firstName">`	把`value`属性绑定到表达式`firstName`
`<div [attr.role]="myAriaRole">`	把属性（Attribute）`role`绑定到表达式`myAriaRole`的结果。
`<div [class.extra-sparkle]="isDelightful">`	根据`isDelightful`表达式的结果是否为真，决定 CSS 类`extra-sparkle`是否出现在当前元素上。
`<div [style.width.px]="mySize">`	把 CSS 样式属性`width`的 px（像素）值绑定到表达式`mySize`的结果。单位是可选的。
`<button (click)="readRainbow($event)">`	当这个按钮元素（及其子元素）上的 click 事件触发时，调用方法`readRainbow`，并把这个事件对象作为参数传进去。
`<div title="Hello {{ponyName}}">`	把一个属性绑定到插值字符串（如"Hello Seabiscuit"）。这种写法等价于`<div [title]="'Hello ' + ponyName">`
`<p>Hello {{ponyName}}</p>`	把文本内容绑定到插值字符串（如"Hello Seabiscuit"）
`<my-cmp [(title)]="name">`	设置双向绑定。等价于`<my-cmp [title]="name" (titleChange)="name=$event">`。

`<video #movieplayer ...>`
`<button (click)="movieplayer.play()">`
`</video>`
	创建一个局部变量`movieplayer`，支持在当前模板的数据绑定和事件绑定表达式中访问`video`元素的实例。

`<p *myUnless="myExpression">...</p>`	这个 `*` 符号会把当前元素转换成一个内嵌的模板。它等价于：`<ng-template [myUnless]="myExpression"><p>...</p></ng-template>`
`<p>Card No.: {{cardNumber | myCardNumberFormatter}}</p>`	使用名叫`myCardNumberFormatter`的管道对表达式`cardNumber`的当前值进行变幻
`<p>Employer: {{employer?.companyName}}</p>`	安全导航操作符（`?`）表示`employer`字段是可选的，如果它是 `undefined` ，那么表达式其余的部分就会被忽略，并返回 `undefined`。
`<svg:rect x="0" y="0" width="100" height="100"/>`	模板中的 SVG 片段需要给它的根元素加上`svg:`前缀，以便把 SVG 元素和 HTML 元素区分开。

`<svg>`
`<rect x="0" y="0" width="100" height="100"/>`
`</svg>`
	以`<svg>`作为根元素时会自动识别为 SVG 元素，不需要前缀。

内置指令	`import { CommonModule } from '@angular/common';`
`<section *ngIf="showSection">`	根据`showSection`表达式的结果，移除或重新创建 DOM 树的一部分。
`<li *ngFor="let item of list">`	把 li 元素及其内容变成一个模板，并使用这个模板为列表中的每一个条目实例化一个视图。

`<div [ngSwitch]="conditionExpression">`
`<ng-template [ngSwitchCase]="case1Exp">...</ng-template>`
`<ng-template ngSwitchCase="case2LiteralString">...</ng-template>`
`<ng-template ngSwitchDefault>...</ng-template>`
`</div>`
	根据`conditionExpression`的当前值选择一个嵌入式模板，并用它替换这个 div 的内容。

`<div [ngClass]="{'active': isActive, 'disabled': isDisabled}">`	根据 map 中的 value 是否为真，来决定该元素上是否出现与 name 对应的 CSS 类。右侧的表达式应该返回一个形如 `{class-name: true/false}` 的 map。
`<div [ngStyle]="{'property': 'value'}">` `<div [ngStyle]="dynamicStyles()">`	允许你使用 CSS 为 HTML 元素指定样式。你可以像第一个例子那样直接使用 CSS，也可以调用组件中的方法。

表单	`import { FormsModule } from '@angular/forms';`
`<input [(ngModel)]="userName">`	为表单控件提供双向数据绑定、解析和验证功能。

类装饰器	`import { Directive, ... } from '@angular/core';`
`@Component({...})` `class MyComponent() {}`	声明一个类是组件，并提供该组件的元数据。
`@Directive({...})` `class MyDirective() {}`	声明一个类是指令，并提供该指令的元数据。
`@Pipe({...})` `class MyPipe() {}`	声明一个类是管道，并提供该管道的元数据。
`@Injectable()` `class MyService() {}`	声明某个类具有一些依赖。当依赖注入器要创建这个类的实例时，应该把这些依赖注入到它的构造函数中。

指令配置项	`@Directive({ property1: value1, ... })`
`selector: '.cool-button:not(a)'`	指定一个 CSS 选择器，用于在模板中标记出该指令。支持的选择器类型包括：`元素名`、`[属性名]`，`.类名` 和 `:not()`。 但不支持指定父子关系的选择器。
`providers: [MyService, { provide: ... }]`	该指令及其子指令的依赖注入提供商列表。

组件配置项	`@Component` 继承自 `@Directive`， 因此，`@Directive` 的这些配置项也同样适用于组件。
`moduleId: module.id`	如果设置了，那么 `templateUrl` 和 `styleUrl` 的路径就会相对于当前组件进行解析。
`viewProviders: [MyService, { provide: ... }]`	依赖注入提供商列表，但它们的范围被限定为当前组件的视图。
`template: 'Hello {{name}}'` `templateUrl: 'my-component.html'`	当前组件视图的内联模板或外部模板的 URL 。
`styles: ['.primary {color: red}']` `styleUrls: ['my-component.css']`	用于为当前组件的视图提供样式的内联 CSS 或外部样式表 URL 的列表。

给指令和组件使用的类属性配置项	`import { Input, ... } from '@angular/core';`
`@Input() myProperty;`	声明一个输入属性，你可以通过属性绑定来更新它，如 `<my-cmp [myProperty]="someExpression">`。
`@Output() myEvent = new EventEmitter();`	声明一个输出属性，它发出事件，你可以用事件绑定来订阅它们（如：`<my-cmp (myEvent)="doSomething()">`）。
`@HostBinding('class.valid') isValid;`	把宿主元素的一个属性（这里是 CSS 类 `valid`）绑定到指令或组件上的 `isValid` 属性。
`@HostListener('click', ['$event']) onClick(e) {...}`	用指令或组件上的`onClick`方法订阅宿主元素上的`click`事件，并从中获取`$event`参数（可选）
`@ContentChild(myPredicate) myChildComponent;`	把组件内容查询（`myPredicate`）的第一个结果绑定到该类的 `myChildComponent` 属性上。
`@ContentChildren(myPredicate) myChildComponents;`	把组件内容查询（`myPredicate`）的全部结果绑定到该类的 `myChildComponents` 属性上
`@ViewChild(myPredicate) myChildComponent;`	把组件视图查询（`myPredicate`）的第一个结果绑定到该类的 `myChildComponent` 属性上。对指令无效。
`@ViewChildren(myPredicate) myChildComponents;`	把组件视图查询（`myPredicate`）的全部结果绑定到该类的 `myChildComponents` 属性上。对指令无效。

指令与组件的变更检测与生命周期钩子	（作为类的方法实现。）
`constructor(myService: MyService, ...) { ... }`	在任何其它生命周期钩子之前调用。可以用它来注入依赖项，但不要在这里做正事。
`ngOnChanges(changeRecord) { ... }`	每当输入属性发生变化时就会调用，但位于处理内容（`ng-content`）或子视图之前。
`ngOnInit() { ... }`	在调用完构造函数、初始化完所有输入属性并首次调用过`ngOnChanges`之后调用。
`ngDoCheck() { ... }`	每当对组件或指令的输入属性进行变更检测时就会调用。可以用它来扩展变更检测逻辑，执行自定义的检测逻辑。
`ngAfterContentInit() { ... }`	`ngOnInit`完成之后，当组件或指令的内容（`ng-content`）已经初始化完毕时调用。
`ngAfterContentChecked() { ... }`	每当组件或指令的内容（`ng-content`）做变更检测时调用。
`ngAfterViewInit() { ... }`	当`ngAfterContentInit`完毕，并且组件的视图及其子视图或指令所属的视图已经初始化完毕时调用。
`ngAfterViewChecked() { ... }`	当组件的视图及其子视图或指令所属的视图每次执行变更检测时调用。
`ngOnDestroy() { ... }`	只在实例被销毁前调用一次。

依赖注入配置项	
`{ provide: MyService, useClass: MyMockService }`	把 `MyService` 的服务提供商设置或改写为 `MyMockService` 类。
`{ provide: MyService, useFactory: myFactory }`	把 `MyService` 的服务提供商设置或改写为 `myFactory` 工厂函数。
`{ provide: MyValue, useValue: 41 }`	把 `MyValue` 的服务提供商改写为一个特定的值 `41` 。

路由与导航	`import { Routes, RouterModule, ... } from '@angular/router';`

`const routes: Routes = [`
`{ path: '', component: HomeComponent },`
`{ path: 'path/:routeParam', component: MyComponent },`
`{ path: 'staticPath', component: ... },`
`{ path: '**', component: ... },`
`{ path: 'oldPath', redirectTo: '/staticPath' },`
`{ path: ..., component: ..., data: { message: 'Custom' } }`
`]);`
 	
`const routing = RouterModule.forRoot(routes);`
	为该应用配置路由。支持静态、参数化、重定向和通配符路由。也支持自定义路由数据和解析（resolve）函数。

`<router-outlet></router-outlet>` `<router-outlet name="aux"></router-outlet>`	标记出一个位置，用来加载活动路由的组件。

`<a routerLink="/path">`
`<a [routerLink]="[ '/path', routeParam ]">`
`<a [routerLink]="[ '/path', { matrixParam: 'value' } ]">`
`<a [routerLink]="[ '/path' ]" [queryParams]="{ page: 1 }">`
`<a [routerLink]="[ '/path' ]" fragment="anchor">`
	使用路由体系创建一个到其它视图的链接。路由体系由路由路径、必要参数、可选参数、查询参数和文档片段组成。要导航到根路由，请使用`/`前缀；要导航到子路由，使用`./`前缀；要导航到兄弟路由或父级路由，使用`../`前缀。

`<a [routerLink]="[ '/path' ]" routerLinkActive="active">`	当 `routerLink` 指向的路由变成活动路由时，为当前元素添加一些类（比如这里的 `active`）。

`class CanActivateGuard implements CanActivate {`
`canActivate(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canActivate: [CanActivateGuard] }`
	用来定义类的接口。路由器会首先调用本接口来决定是否激活该路由。应该返回一个 `boolean` 或能解析成 `boolean` 的 `Observable/Promise`。

`class CanDeactivateGuard implements CanDeactivate<T> {`
`canDeactivate(`
`component: T,`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canDeactivate: [CanDeactivateGuard] }`
	用来定义类的接口。路由器会在导航离开前首先调用本接口以决定是否取消激活本路由。应该返回一个 `boolean` 或能解析成 `boolean` 的 `Observable/Promise`。

`class CanActivateChildGuard implements CanActivateChild {`
`canActivateChild(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canActivateChild: [CanActivateGuard],`
`children: ... }`
	用来定义类的接口。路由器会首先调用本接口来决定是否激活一个子路由。应该返回一个 `boolean` 或能解析成 `boolean` 的 `Observable/Promise`。

`class ResolveGuard implements Resolve<T> {`
`resolve(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<any>|Promise<any>|any { ... }`
`}`
 	
`{ path: ..., resolve: [ResolveGuard] }`
	用来定义类的接口。路由器会在渲染该路由之前，首先调用它来解析路由数据。应该返回一个值或能解析成值的 `Observable/Promise`。

`class CanLoadGuard implements CanLoad {`
`canLoad(`
`route: Route`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canLoad: [CanLoadGuard], loadChildren: ... }`
	用来定义类的接口。路由器会首先调用它来决定是否应该加载一个惰性加载模块。应该返回一个 `boolean` 或能解析成 `boolean` 的 `Observable/Promise`。