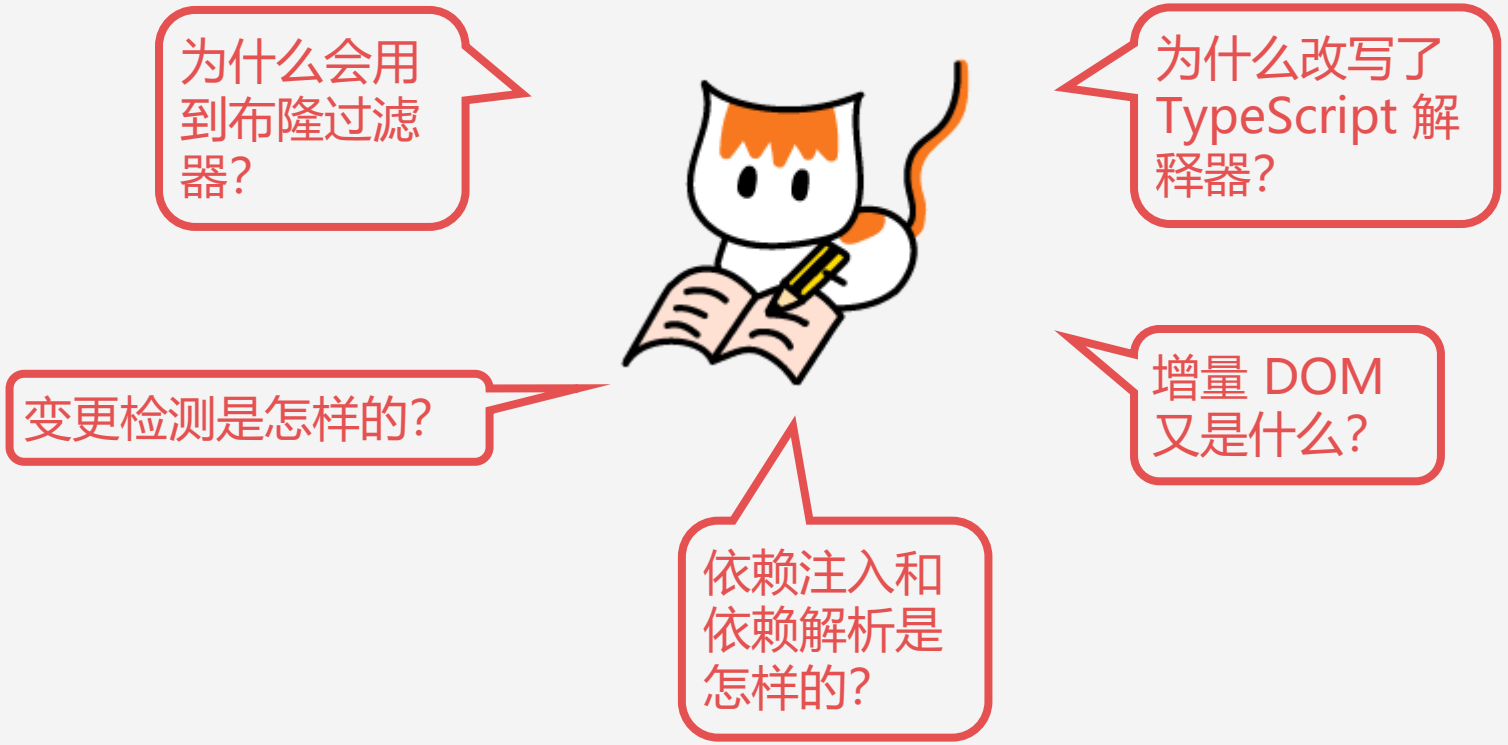
Angular 冷知识
之
装饰器和元数据

@被删

Angular 知多少


为什么会用到布隆过滤器?
为什么改写了 TypeScript 解释器?
变更检测是怎样的?
增量 DOM 又是什么?
依赖注入和依赖解析是怎样的?

1

Angular 中的装饰器

Angular 中的装饰器（Decorator）

本质上，装饰器可用于对值进行元编程和向其添加功能，而无需从根本上改变其外部行为。

定义组件

```
@Component({
  selector: 'app-component-overview',
  template: '<h1>Hello World!</h1>',
  styles: ['h1 { font-weight: normal; }']
})
```

定义可注入服务

```
@Injectable()
export class EvenBetterLogger extends Logger {
  constructor(private userService: UserService) { super(); }

  log(message: string) {
    const name = this.userService.user.name;
    super.log(`Message to ${name}: ${message}`);
  }
}
```

定义指令

订阅本属性型指令宿主 DOM 元素上的事件

```
@Directive({ selector: '[appUnless]'})
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @Input() set appUnless(condition: boolean) {
    if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }
}
```

Angular 中的元数据（Metadata）

Angular 中，通过给装饰器传参数，来定义装饰器的元数据。

```
@Component({
  selector: 'app-hero-list',
  template: `
    <div *ngFor="let hero of heroes">
      {{hero.id}} - {{hero.name}}
    </div>
  `
})
export class HeroListComponent {
  heroes: Hero[];

  constructor(heroService: HeroService) {
    this.heroes = heroService.getHeroes();
  }
}
```

元数据：描述装饰器的行为

使用装饰器和元数据来改变类的行为

以 @Component() 为例，该装饰器的作用包括：

1. 将类标记为 Angular 组件。
2. 提供可配置的元数据，用来确定应在运行时如何处理、实例化和使用该组件。

```
// 提供 Angular 组件的配置元数据接口定义
// Angular 中，组件是指令的子集，始终与模板相关联
export interface Component extends Directive {
  // changeDetection 用于此组件的变更检测策略
  // 实例化组件时，Angular 将创建一个更改检测器，该更改检测器负责传播组件的绑定。
  changeDetection?: ChangeDetectionStrategy;
  // 定义对其视图 DOM 子对象可见的可注入对象的集合
  viewProviders?: Provider[];
  // 包含组件的模块的模块ID，该组件必须能够解析模板和样式的相对 URL
  moduleId?: string;
  ...
  // 模板和 CSS 样式的封装策略
  encapsulation?: ViewEncapsulation;
  // 覆盖默认的插值起始和终止定界符（`{{`和`}}`）
  interpolation?: [string, string];
}
```

组件是怎么创建和编译的?

组件的编译过程

Angular 会根据该装饰器元数据，来编译 Angular 组件，然后将生成的组件定义（ɵcmp）修补到组件类型上。

```
// 创建编译组件需要的完整元数据
const templateUrl = metadata.templateUrl || `ng:///${type.name}/template.html`;
const meta: R3ComponentMetadataFacade = {
  ...directiveMetadata(type, metadata),
  typeSourceSpan: compiler.createParseSourceSpan('Component', type.name, templateUrl),
  template: metadata.template || '',
  preserveWhitespaces,
  styles: metadata.styles || EMPTY_ARRAY,
  animations: metadata.animations,
  directives: [],
  changeDetection: metadata.changeDetection,
  pipes: new Map(),
  encapsulation,
  interpolation: metadata.interpolation,
  viewProviders: metadata.viewProviders || null,
};
// 编译过程需要计算深度，以便确认编译是否最终完成
compilationDepth++;
try {
  if (meta.usesInheritance) {
    addDirectiveDefToUndecoratedParents(type);
  }
  // 根据模板、环境和组件需要的元数据，来编译组件
  ngComponentDef = compiler.compileComponent(angularCoreEnv, templateUrl, meta);
} finally {
  // 即使编译失败，也请确保减少编译深度
  compilationDepth--;
}
```

compileComponent 方法，JIT 场景下组件的编译步骤：

1. 解析元数据。
2. 处理可能具有需要解析的资源（templateUrl、styleUrls）。
3. 根据模板、环境和组件需要的元数据，来编译组件。

装饰器去哪了？

2

装饰器和元数据的编译过程

装饰器的编译过程

Angular 中所有装饰器都会通过 makeDecorator() 来产生，makeDecorator 主要是针对不同的装饰器做一些逻辑处理：

```
// 组件装饰器和元数据
export const Component: ComponentDecorator = makeDecorator(
    'Component',
    // 使用默认的 CheckAlways 策略，在该策略中，更改检测是自动进行的，直到明确停用为止。
    (c: Component = {}) => ({changeDetection: ChangeDetectionStrategy.Default, ...c}),
    Directive, undefined,
    (type: Type<any>, meta: Component) => SWITCH_COMPILE_COMPONENT(type, meta));
```

- JIT（Just in time，即时编译）：装饰器使用 tsc 编译器并生成静态字段
- AOT（Ahead of Time，预先编译）：使用 ngc 编译器进行编译

元数据被存储在哪？

元数据的管理

数据很多使用装饰器和元数据的功能，在开发时都会使用到 Reflect Metadata。

Reflect Metadata 是 ES7 的一个提案，它主要用来在声明的时候添加和读取元数据。

```
export class CompileMetadataResolver {
  private _nonNormalizedDirectiveCache =
      new Map<Type, {annotation: Directive, metadata: cpl.CompileDirectiveMetadata}>();
  // 使用 Map 的方式来保存
  private _directiveCache = new Map<Type, cpl.CompileDirectiveMetadata>();
  private _summaryCache = new Map<Type, cpl.CompileTypeSummary|null>();
  private _pipeCache = new Map<Type, cpl.CompilePipeMetadata>();
  private _ngModuleCache = new Map<Type, cpl.CompileNgModuleMetadata>();
  private _ngModuleOfTypes = new Map<Type, Type>();
  private _shallowModuleCache = new Map<Type, cpl.CompileShallowModuleMetadata>();
```

元数据是有关类的信息，但它不是类的属性。
因此不应该存储在该类的实例中，我们还需要在其他地方保存此数据。

在 Angular 中，编译过程产生的元数据，会使用 CompileMetadataResolver 来进行管理和维护。

不管是组件、指令、管道，还是模块，这些类在编译过程中的元数据，都使用 Map 来存储。

AOT 模式编译

AOT 收集器（collector） 会记录 Angular 装饰器中的元数据，并把它们输出到 .metadata.json 文件中。

AOT 编译分为三个阶段：

1. 代码分析。 在此阶段，TypeScript 编译器和 AOT 收集器会创建源码的表现层。收集器不会尝试解释其收集到的元数据。它只是尽可能地表达元数据，并在检测到元数据语法冲突时记录错误。
2. 代码生成。在此阶段，编译器的 StaticReflector 会解释在 1 中收集的元数据，对元数据执行附加验证，如果检测到元数据违反了限制，则抛出错误。
3. 模板类型检查。

可以把 .metadata.json 文件看做一个包括全部装饰器的元数据的全景图，就像抽象语法树 (AST) 一样

Angular Ivy 编译器的变革

View Engine（Render2 ）依赖全局信息来进行编译，metadata.json 存储了装饰器信息。
Ivy（Render3）中某个类的元数据被转换为 .js 文件中的元数据。

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

@Component({
  selector: 'greet',
  template: '<div> Hello, {{name}}! </div>'
})
export class GreetComponent {
  @Input() name: string;
}
```

Ivy

```
const i0 = require("@angular/core");
class GreetComponent {}
GreetComponent.ecmp = i0.ɵɵdefineComponent({
    type: GreetComponent,
    tag: 'greet',
    factory: () => new GreetComponent(),
    template: function (rf, ctx) {
        if (rf & RenderFlags.Create) {
            i0.ɵɵelementStart(0, 'div');
            i0.ɵɵtext(1);
            i0.ɵɵelementEnd();
        }
        if (rf & RenderFlags.Update) {
            i0.ɵɵadvance(1);
            i0.ɵɵtextInterpolate1('Hello ', ctx.name, '!');
        }
    }
});
```

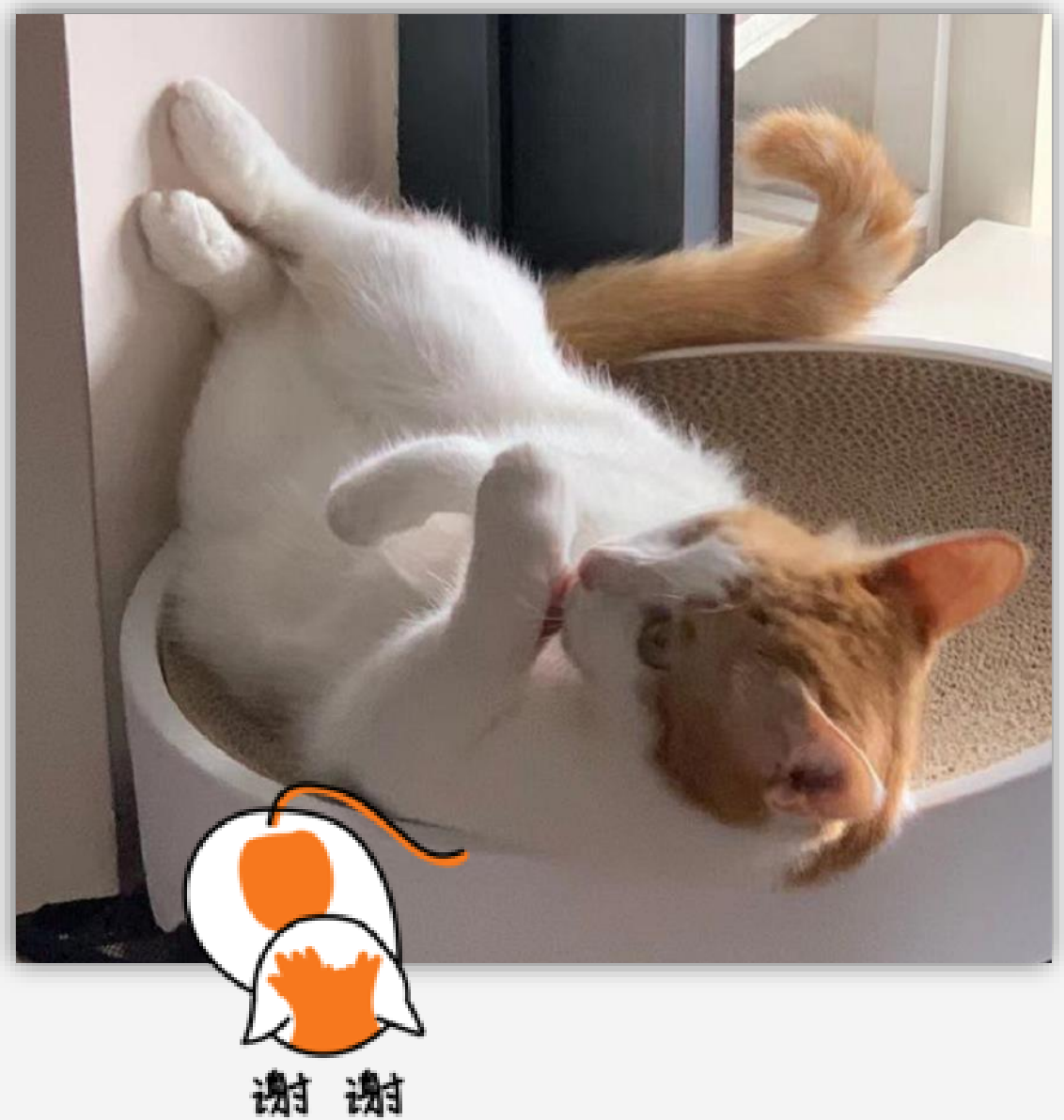
View Engine

```
const tslib_1 = require("tslib");
const core_1 = require("@angular/core");
let GreetComponent = class GreetComponent {
};
tslib_1.__decorate([
    core_1.Input(),
    tslib_1.__metadata("design:type", String)
], GreetComponent.prototype, "name", void 0);
GreetComponent = tslib_1.__decorate([
    core_1.Component({
        selector: 'greet',
        template: '<div> Hello, {{name}}! </div>'
    })
], GreetComponent);
```

Ivy 引擎设计上支持局部性原则，不依赖任何未直接传递给它的输入（比如全局信息）

Angular 框架源码分析和解读

- https://github.com/godbasin/godbasin.github.io
- https://github.com/godbasin/front-end-playground



Github: godbasin
@被删