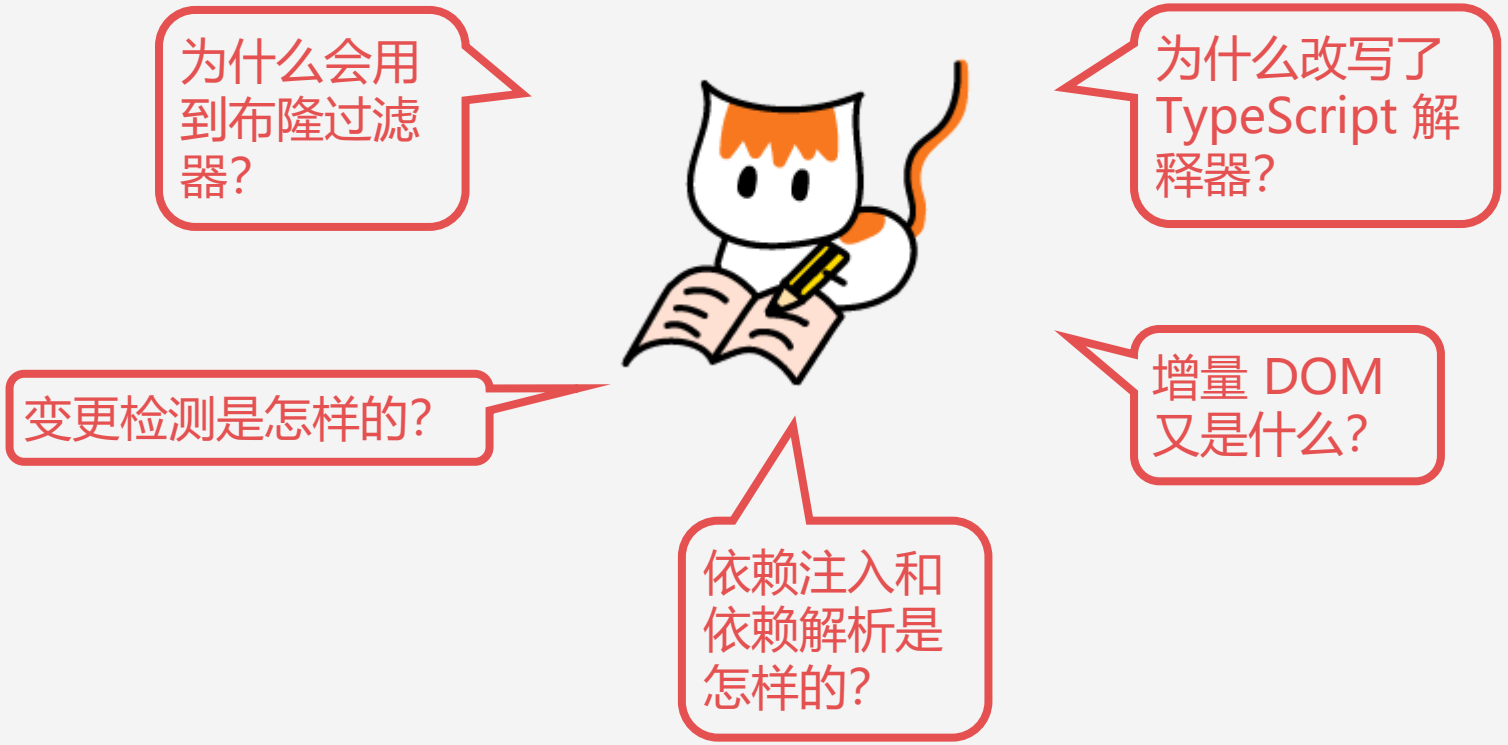
# Angular 冷知识
# 之
# 多级依赖注入

@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 依赖注入设计

# Angular 中的依赖注入—注入器机制

在 Angular 中，使用了依赖注入的技术，DI 框架会在实例化某个类时，向其提供这个类所声明的依赖项

Angular 中主要的依赖注入机制是注入器机制：

1. 应用中所需的任何依赖，都必须使用该应用的注入器来注册一个提供者，以便注入器可以使用这个提供者来创建新实例
2. Angular 会在启动过程中，创建全应用级注入器以及所需的其它注入器

依赖注入的一些概念：

- Injector 注入器
- Provider 提供商/提供者
- Injectable 依赖注入服务

# Angular 中的依赖注入—Injector 注入器

Injector 注入器用于创建依赖，会维护一个容器来管理这些依赖，并尽可能地复用它们。

注入器会提供依赖的一个单例，并把这个单例对象注入到多个组件中。

注入器的功能很简单：

- 创建依赖实例
- 获取依赖实例
- 管理依赖实例

Angular 通过 Token 查询和检索注入器来获取相应的依赖实例

```
export abstract class Injector {
  // 找不到依赖
  static THROW_IF_NOT_FOUND = THROW_IF_NOT_FOUND;
  // NullInjector 是树的顶部
  // 如果你在树中向上走了很远，以至于要在 NullInjector 中寻找服务，那么将收到错误消息，
  static NULL: Injector = new NullInjector();

  // 根据提供的 Token 从 Injector 检索实例
  abstract get<T>(
    token: Type<T> | AbstractType<T> | InjectionToken<T>,
    notFoundValue?: T,
    flags?: InjectFlags
  ): T;

  // 创建一个新的 Injector 实例，该实例提供一个或多个依赖项
  static create(options: {
    providers: StaticProvider[];
    parent?: Injector;
    name?: string;
  }): Injector;

  // ɵɵdefineInjectable 用于构造一个 InjectableDef
  // 它定义 DI 系统将如何构造 Token，并且在哪些 Injector 中可用
  static ɵprov = ɵɵdefineInjectable({
    token: Injector,
    providedIn: "any" as any,
    // ɵɵinject 生成的指令：从当前活动的 Injector 注入 Token
    factory: () => ɵɵinject(INJECTOR),
  });

  static __NG_ELEMENT_ID__ = InjectorMarkers.Injector;
}
```

# Angular 中的依赖注入—Provider 提供商

Provider 提供者用来告诉注入器应该如何获取或创建依赖，要想让注入器能够创建服务（或提供其它类型的依赖），必须使用某个提供者配置好注入器。

一个提供者对象定义了如何获取与 DI 令牌（token）相关联的可注入依赖，而注入器会使用这个提供者来创建它所依赖的那些类的实例。

```
function resolveReflectiveFactory(
  provider: NormalizedProvider
): ResolvedReflectiveFactory {
  let factoryFn: Function;
  let resolvedDeps: ReflectiveDependency[];
  if (provider.useClass) {
    // 使用类来提供依赖
    const useClass = resolveForwardRef(provider.useClass);
    factoryFn = reflector.factory(useClass);
    resolvedDeps = _dependenciesFor(useClass);
  } else if (provider.useExisting) {
    // 使用已有依赖
    factoryFn = (aliasInstance: any) => aliasInstance;
    // 从根据 token 获取具体的依赖
    resolvedDeps = [
      ReflectiveDependency.fromKey(ReflectiveKey.get(provider.useExisting)),
    ];
  } else if (provider.useFactory) {
    // 使用工厂方法提供依赖
    factoryFn = provider.useFactory;
    resolvedDeps = constructDependencies(provider.useFactory, provider.deps);
  } else {
    // 使用提供者具体的值作为依赖
    factoryFn = () => provider.useValue;
    resolvedDeps = _EMPTY_LIST;
  }
  //
  return new ResolvedReflectiveFactory(factoryFn, resolvedDeps);
}
```

# Angular 中的依赖注入—Injectable 依赖注入服务

在 Angular 中，服务就是一个带有 @Injectable 装饰器的类，它封装了可以在应用程序中复用的非 UI 逻辑和代码。

```
// 根据其 Injectable 元数据，编译 Angular 可注入对象，并对结果进行修补
export function compileInjectable(type: Type<any>, srcMeta?: Injectable): void {
  // 该编译过程依赖 @angular/compiler
  // 可参考编译器中的 compileFactoryFunction compileInjectable 实现
}
```

```
export interface ɵɵInjectableDef<T> {
  // 指定给定类型属于特定注入器，包括 root/platform/any/null 以及特定的 NgModule
  providedIn: InjectorType<any> | "root" | "platform" | "any" | null;
  // 此定义所属的令牌
  token: unknown;
  // 要执行以创建可注入实例的工厂方法
  factory: (t?: Type<any>) => T;
  // 在没有显式注入器的情况下，存储可注入实例的位置
  value: T | undefined;
}
```

对于注入器、提供者和可注入服务，我们可以简单地这样理解：

1. 注入器用于创建依赖，会维护一个容器来管理这些依赖，并尽可能地复用它们。
2. 一个注入器中的依赖服务，只有一个实例。
3. 注入器需要使用提供者来管理依赖，并通过 token（DI 令牌）来进行关联。
4. 提供者用于告诉注入器应该如何获取或创建依赖。
5. 可注入服务类会根据元数据编译后，得到可注入对象，该对象可用于创建实例。

# Angular 中的依赖查询

注入器是可继承的，这意味着如果指定的注入器无法解析某个依赖，它就会请求父注入器来解析它

```
// 创建一个新的 Injector 实例，可传入 parent 父注入器
static create(options: {providers: StaticProvider[], parent?: Injector, name?: string}):
```

组件可以：

1. 从它自己的注入器来获取服务；
2. 从其祖先组件的注入器中获取；
3. 从其父 NgModule 的注入器中获取；
4. 或从 root 注入器中获取。

Angular 中的依赖注入到底有多少层？

2

# 多级/分层的依赖注入

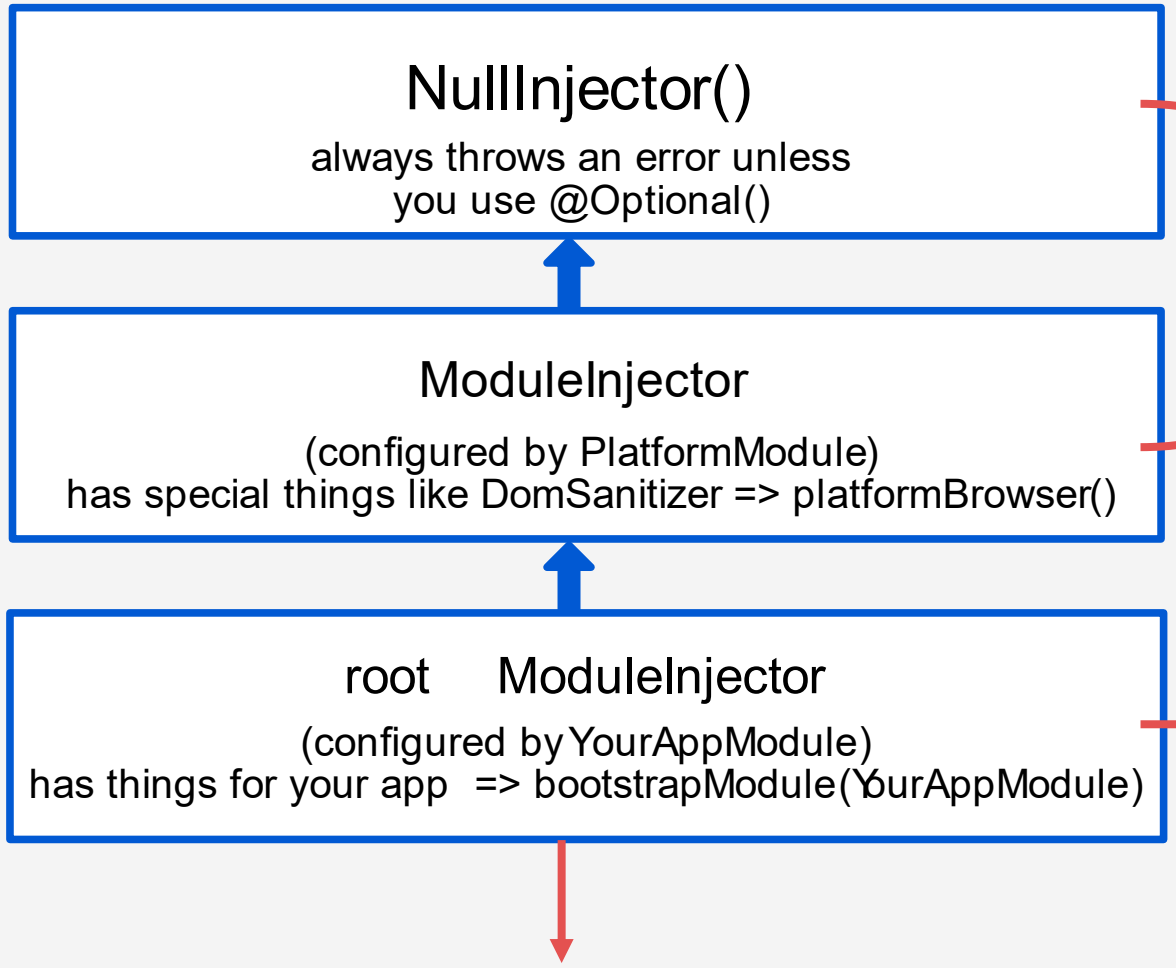
# Angular 的多级依赖注入

在 Angular 中，有两个注入器层次结构：

- ModuleInjector 模块注入器：使用 @NgModule() 或 @Injectable() 注解在此层次结构中配置
- ElementInjector 元素注入器：在每个 DOM 元素上隐式创建

模块注入器和元素注入器都是树状结构的，但它们的分层结构并不完全一致。

# Angular 的多级依赖注入—模块注入器

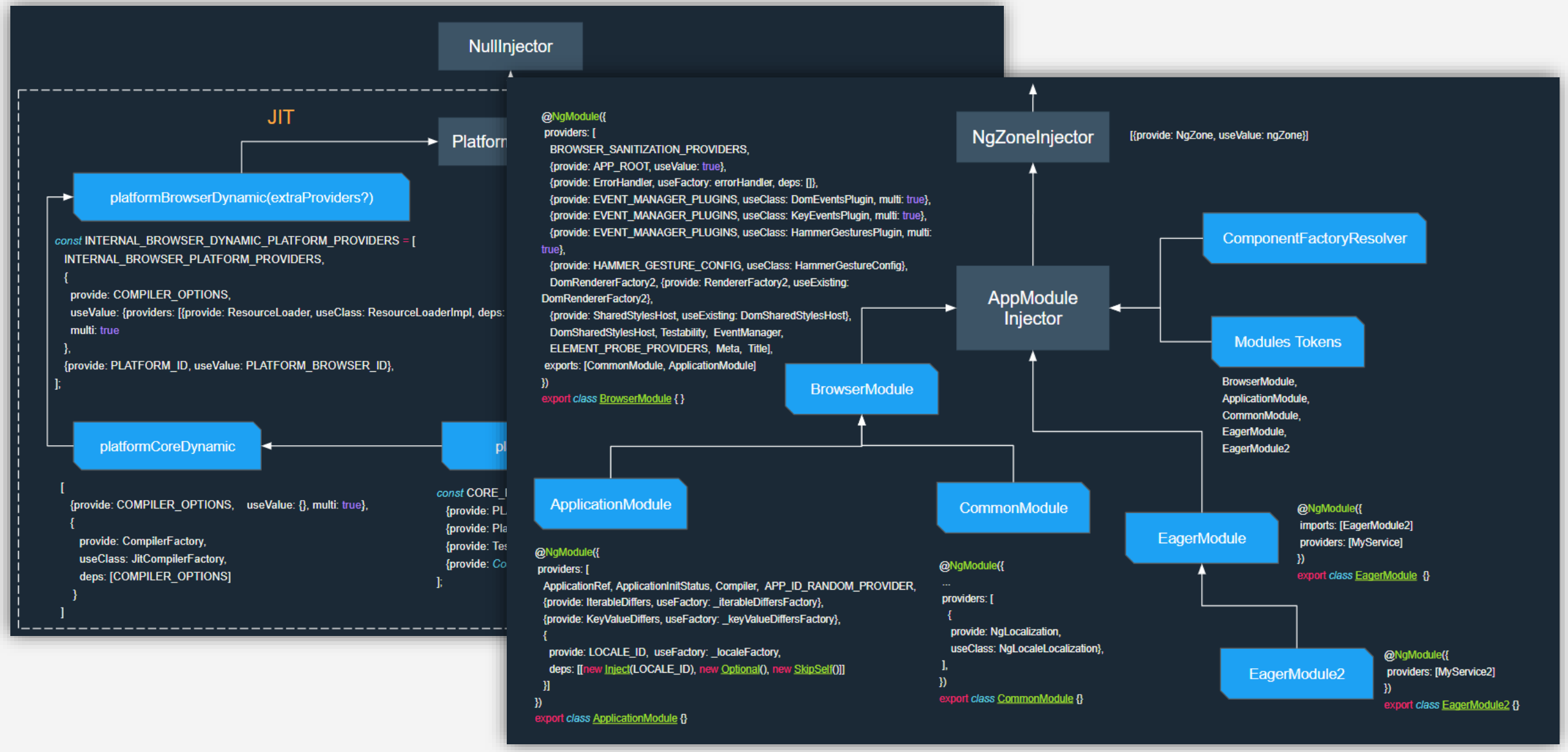



在 root 之上还有两个注入器：

1. 平台模块（PlatformModule）注入器
2. NullInjector()

模块注入器树的最上层则是应用程序根模块（AppModule）注入器，称作 root

# Angular 的多级依赖注入—模块注入器



# 模块注入器的问题

Lazy loaded module's resolver is not being kept as singleton #13722

Closed jbridger opened this issue on 30 Dec 2016 · 6 comments

延迟加载模块场景下，组件被双重实例化

jbridger commented on 30 Dec 2016

I'm submitting a ...

```
[x] bug report => search github for a similar issue or PR before submitting
[ ] feature request
[ ] support request => Please do not submit support request here, instead see https://github.com/angular/angular/blob/mas
```

**Current behavior**
What we're trying to achieve:

- We have implemented a resolve guard in a lazily loaded module that will navigate to an error component on failure to retrieve data.
- The resolve guard will also store the failure message in a field of itself.
- The error component takes the resolve guard as a dependency, and displays the error message from it.

Our implementation is very similar to the Tour of Heroes example with the resolve guard navigating to another page on failure. See this plunkr example, and click on any crisis under the `Crisis Center`

We have produced an example of the problem by extending the Tour of Heroes example.

When `CrisisCenterModule` is lazily loaded, we have observed:

- The resolver instance for `CrisisDetailComponent` is different to the instance being injected in to `ErrorComponent`, which is a sibling route to the activate route. We have confirmed that the resolver is only provided inside the `CrisisCenterRoutingModule`, which is inside the lazily loaded module.
- We can see that the `CrisisCenterModule` and all it's contained modules and components are instantiated twice, with different injectors.

When `CrisisCenterModule` module is not lazily loaded, this problem does not occur.

Assignees
No one assigned

Labels
comp: core comp: router

Projects
None yet

Milestone
No milestone

Linked pull requests
Successfully merging a pull request may close this issue.
None yet

Notifications Customize
Subscribe
You're not receiving notifications from this thread.

6 participants

新的设计：注入器使用两棵并行的树，一棵用于元素，另一棵用于模块。

通过更改注入器层次结构，避免交错插入模块和组件注入器，从而导致延迟加载模块的双倍实例化

# Angular 的多级依赖注入—元素注入器

Angular 会为所有 entryComponents 创建宿主工厂，它们是所有其他组件的根视图。

```
class ComponentFactory_ extends ComponentFactory<any>{
  create(
      injector: Injector, projectableNodes?: any[][], rootSelectorOrNode?: string|any,
      ngModule?: NgModuleRef<any>): ComponentRef<any> {
    if (!ngModule) {
      throw new Error('ngModule should be provided');
    }
    const viewDef = resolveDefinition(this.viewDefFactory);
    const componentNodeIndex = viewDef.nodes[0].element!.componentProvider!.nodeIndex;
    // 使用根数据创建根视图
    const view = Services.createRootView(
        injector, projectableNodes || [], rootSelectorOrNode, viewDef, ngModule, EMPTY_CONTEXT);
    // view.nodes 的访问器
    const component = asProviderData(view, componentNodeIndex).instance;
    if (rootSelectorOrNode) {
      view.renderer.setAttribute(asElementData(view, 0).renderElement, 'ng-version', VERSION.ful
    }
    // 创建组件
    return new ComponentRef_(view, new ViewRef_(view), component);
  }
}
```

- 创建动态 Angular 组件时，都会使用根数据(RootData)创建根视图(RootView)。
- 当 Angular 为嵌套的 HTML 元素创建元素注入器时，要么从父元素注入器继承它，要么直接将父元素注入器分配给子节点定义。

# 模块注入器和元素注入器是什么关系呢?

# Angular 解析依赖过程

在 Angular 种，当为组件/指令解析 token 获取依赖时，Angular 分为两个阶段来解析它：

- 针对 ElementInjector 层次结构。
- 针对 ModuleInjector 层次结构。

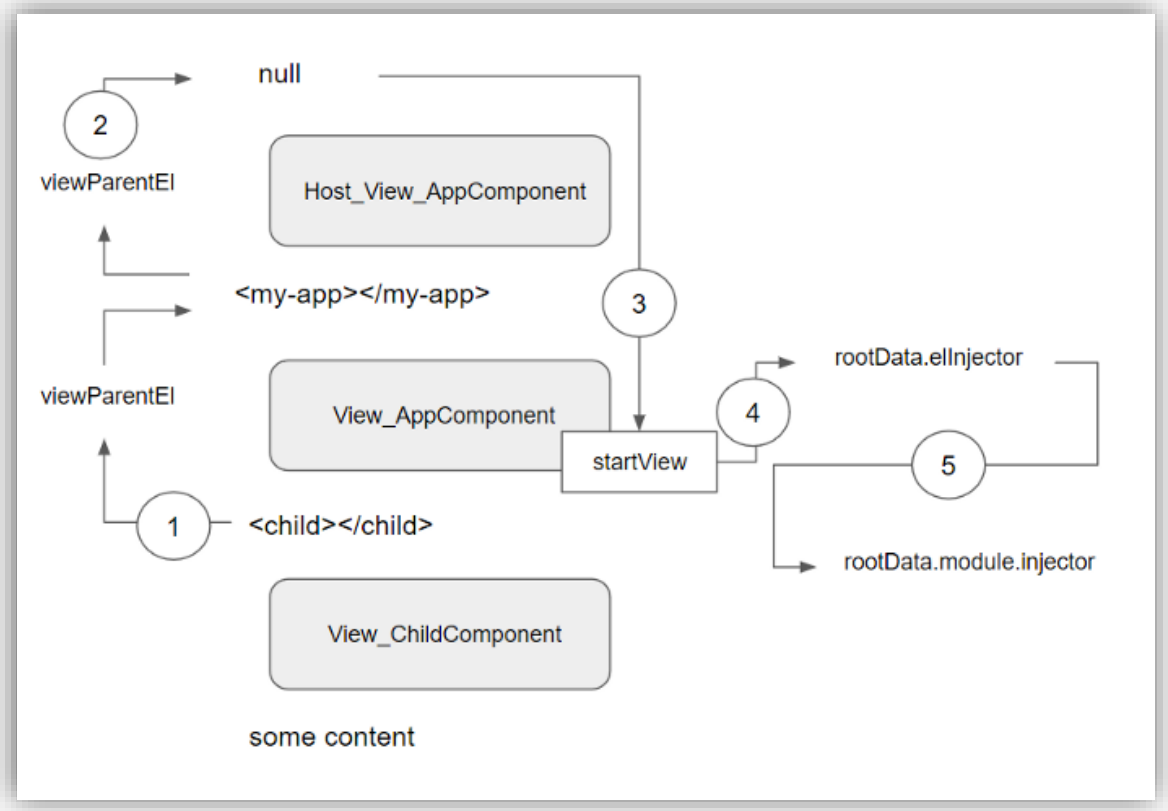
1. 当组件声明依赖项时，Angular 会尝试使用它自己的 ElementInjector 来满足该依赖。
2. 如果组件的注入器缺少提供者，它将把请求传给其父组件的 ElementInjector。
3. 这些请求将继续转发，直到 Angular 找到可以处理该请求的注入器或用完祖先 ElementInjector。
4. 如果 Angular 在任何 ElementInjector 中都找不到提供者，它将返回到发起请求的元素，并在 ModuleInjector 层次结构中进行查找。
5. 如果 Angular 仍然找不到提供者，它将引发错误。

# 元素注入器和模块注入器的桥接——合并注入器

当 Angular 解析依赖项时，合并注入器则是元素注入器树和模块注入器树之间的桥梁。

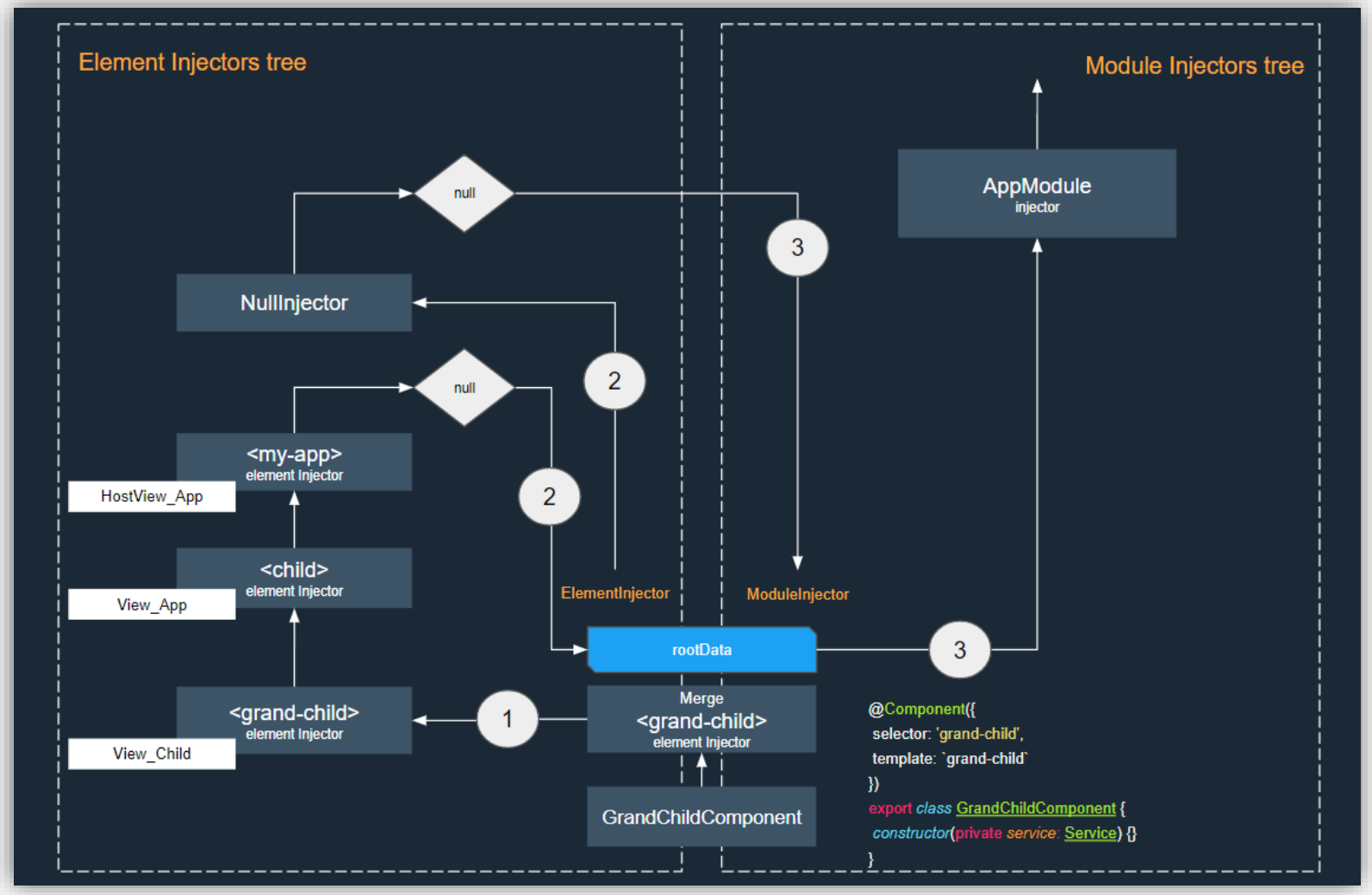
```
class Injector_ implements Injector {
  constructor(private view: ViewData, private elDef: NodeDef|null) {}
  get(token: any, notFoundValue: any = Injector.THROW_IF_NOT_FOUND): any {
    const allowPrivateServices =
        this.elDef ? (this.elDef.flags & NodeFlags.ComponentView) !== 0 : false;
    return Services.resolveDep(
        this.view, this.elDef, allowPrivateServices,
        {flags: DepFlags.None, token, tokenKey: tokenKey(token)}, notFoundValue);
  }
}
```

合并注入器本身没有任何值，它只是视图和元素定义的组合。

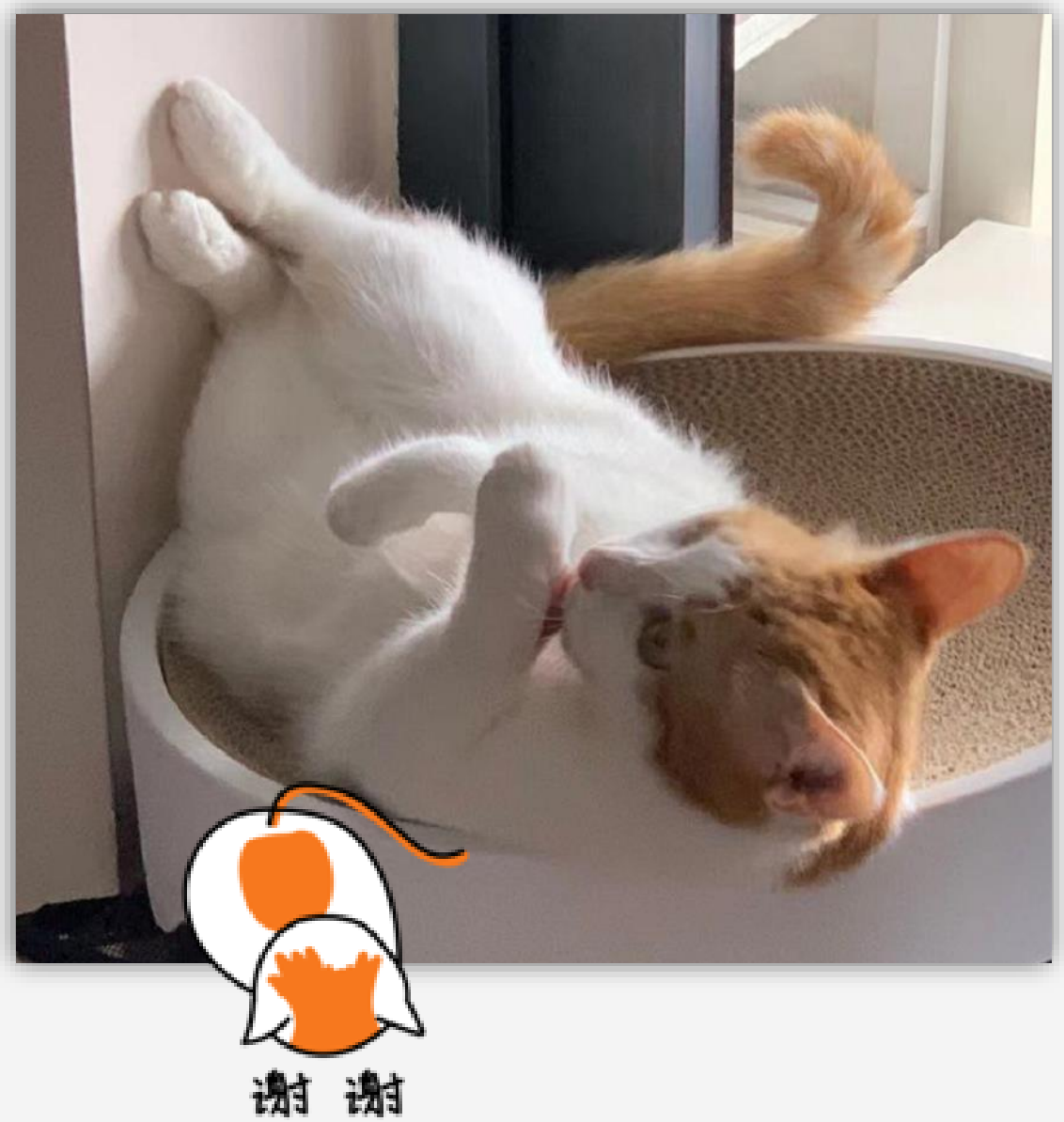



当 Angular 尝试解析组件或指令中的某些依赖关系时，会使用合并注入器来遍历元素注入器树，然后，如果找不到依赖关系，则切换到模块注入器树以解决依赖关系。

# 依赖注入查询过程




1. 首先查看子元素注入器。
2. 然后遍历所有父视图元素（1），并检查元素注入器中的提供者。
3. 如果下一个父视图元素等于null（2），则返回到startView，检查startView.rootData.elnjector（3）。
4. 只有在找不到令牌的情况下，才检查startView.rootData module.injector（4）。




Github: godbasin
@被删