angular 依赖注入 解析

1. 依赖注入和控制反转(Ioc)的区别是什么？
2. `providedIn: 'root'` 的作用是什么，指定 root 的好处有哪些？
3. `providedIn`除了 root 外还可以设置哪些值？
4. 构造函数注入`constructor(heroService: HeroService)` 是依赖注入的简写，那么完整的写法是什么？
5. 依赖注入可以注入一个接口吗？`constructor(heroService: IHeroService)`为什么？
6. 虽然 TypeScript 的`AppConfig`接口可以在类中提供类型支持，但它在依赖注入时却没有任何作用。在 TypeScript 中，接口只能作为类型检查，它没有可供 DI 框架使用的运行时表示形式或令牌。
    - 当转译器把 TypeScript 转换成 JavaScript 时，接口就会消失，因为 JavaScript 没有接口。
    - 由于 Angular 在运行期没有接口，所以该接口不能作为令牌，也不能注入它。
7. `useClass` 和`useExisting` 提供者的区别是什么？
8. `providers: [ Logger ]` 这种写法`Logger`使用的是什么类型的供应商（Provider）？
9. DI Token 可以是字符串吗？如果可以如何注入？
10. `providers` 与`viewProviders`的区别是什么？
11. `Injectable`、`Inject` 、`Injector` 和`Provider` 这些名词到底是什么？
12. `ReflectiveInjector` 和`StaticInjector`的区别是什么？为什么 Angular 在 V5 版本废弃了`ReflectiveInjector`API？
13. 懒加载模块中的供应商和`AppModule`中提供的供应商有什么区别？

angular 与其他框架的区别

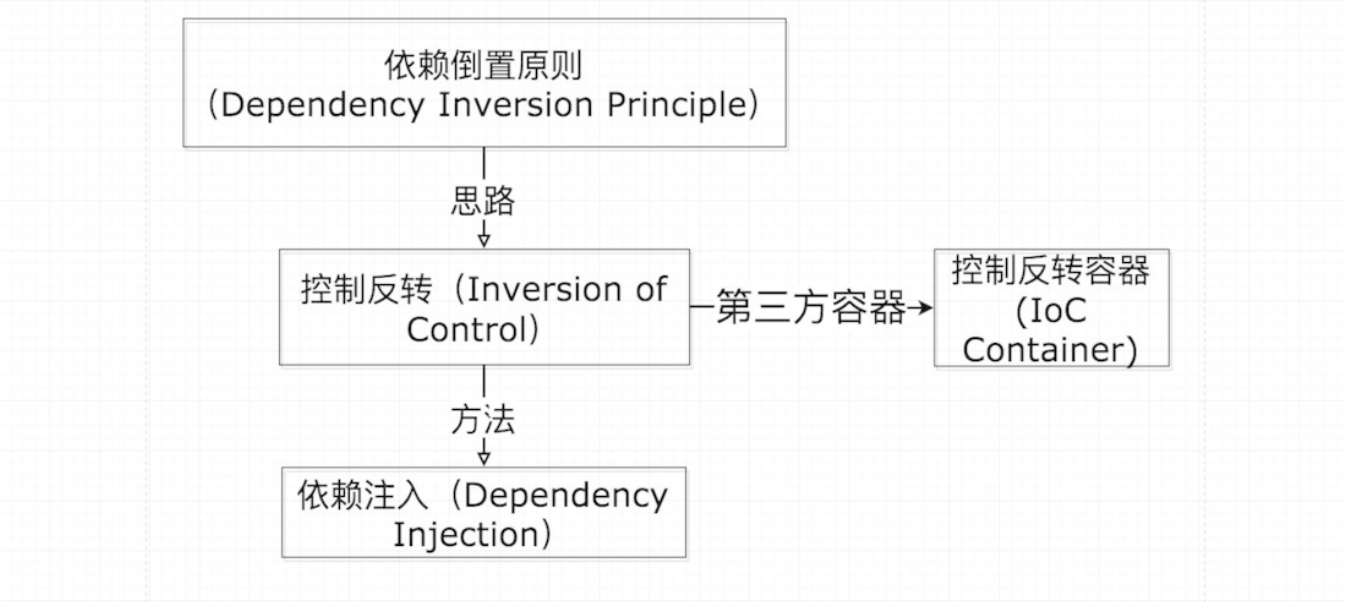
讲一下依赖注入

依赖注入与控制反转的区别是什么

软件只有到达了一定的复杂度才会需要各种设计原则和模式，那么依赖倒置原则（Dependency Inversion Principle ）就是为了解决软件模块之间的耦合性提出的一种思想，让大型软件变的可维护，高层模块不应该依赖低层模块，两者都应该依赖其抽象，抽象不应该依赖细节，细节应该依赖抽象。那么控制反转（Inversion of Control） 就是依赖倒置原则的一种代码设计思路，具体采用的方法就是所谓的依赖注入（Dependency Injection） ，通过依赖注入实现控制权的反转，除了依赖注入外，还有可以通过模板方法模式实现控制反转，那么所谓依赖注入，就是把底层类作为参数传入上层类，实现上层类对下层类的“控制”。

推荐阅读：https://www.zhihu.com/question/23277575 spring loc 有什么好处。

补充：SOLID原则



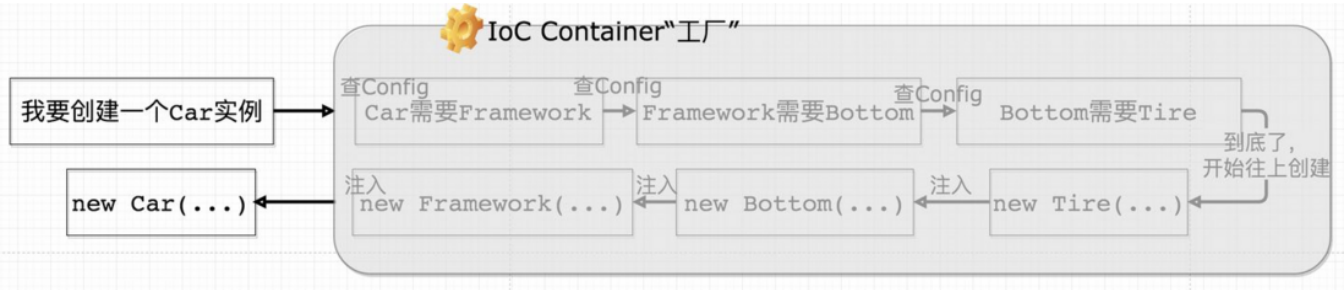
以下是一个通过构造函数注入的示例，那么除了构造函数注入外，还会有 setter 注入和接口注入。

```
class Logger {
    log(message: string) {}
}
class HeroesService {
    constructor(logger: Logger) {}
}

const logger = new Logger();
const heroesService = new HeroesService(logger);
```

通过上述示例发现，HeroesService 不直接创建Logger 类的实例，统一在外层创建后通过构造函数好传入HeroesService如果我们的类成千上万，那么实例化类的工作变得相当繁琐，会有一大推样板代码，为了管理创建依赖工作，一般会使用**控制反转容器(IoC Container)**进行管理。只需要通过如下一行代码即可实现HeroesService 的创建，IocContainer 会通过HeroesService的构造函数寻找Logger 的依赖并实例化。

```
const heroesService = IocContainer.get(HeroesService);
 IocContainer  IocContainer   Injector , Angular  Injector
```



MF控制翻转与依赖注入的理解：https://stackoverflow.com/questions/6550700/inversion-of-control-vs-dependency-injection

依赖注入的优势：

- 更容易维护
- 协同合作
- 方便单元测试
- 松耦合
- 减少了样板代（Ioc容器/注入器维护管理依赖）
- 扩展应用程序变得更加容易

依赖注入的缺点：

- 学习起来有点复杂
- 阅读代码变得抽象
- 依赖注入框架是通过反射或动态编程实现，导致IDE对"查找引用"，"显示调用层次结构"和安全重构变得困难
- 编译时错误被推送到运行时

为什么 Angular 有依赖注入

那么 Angular 为什么会有依赖注入？
前面我已经说过，只有程序到达一定的复杂度，才会需要各种设计模式和原则等工程化方法提升程序的可维护性，那么 Angular.js 起初是为了解决谷歌内部复杂中大型的前端应用，同时是一批 Java 程序打造的，所以首次在前端中大胆引入了依赖注入，那么 Angular 是基于 Angular.js 打造的新一代前端框架，所以延续了依赖注入特性，并改善了层级注入器，同时采用了更优雅的装饰器 API 形式。

服务和依赖注入的关系
另外 Angular 为了解决数据共享和逻辑复用问题，引入了服务的概念，服务简单理解就是一个带有特性功能的类，Angular 提倡把与视图无关的逻辑抽取到服务中，这样可以让组件类更加精简、高效，组件的工作只管用户体验，把业务逻辑相关功能（比如：从服务器获取数据，验证用户输入或直接往控制台中写日志等）委托给各种服务，最后通过 Angular 的依赖注入，这些带有特定功能的服务类可以被任何组件注入并使用。
Angular 依赖注入：连接服务的桥梁，在需要的地方（组件/指令/其他服务）通过构造函数注入依赖的服务，依赖注入 + 服务的组合造就了使用 Angular 可以轻松组织复杂应用。

那么其他框架 React 和 Vue 有依赖注入吗？
可以说有，也可以说没有，React 为了解决全局数据的共享问题，提出了 Context，那么创建好 Context 后需要在上层组件通过`<MyContext.Provider value={/*  */}>` 提供依赖值，然后在任何的子组件中通过`<MyContext.Consumer>` 进行消费（Vue 中也有类似的`provide`和`inject`），其实这可以狭隘的理解成最简单的依赖注入，只不过 Context 只解决了数据共享的问题，虽然也可以作为逻辑复用，但是官方不推荐，React 官方先后推出 Mixin、高阶组件、Render Props 以及最新的 Hooks 用来解决逻辑复用问题。

```
<MyContext.Consumer>
  {value => /*  context */}
</MyContext.Consumer>
 Angular Angular  +  React Hooks  API + API
```

基本使用

在 Angular 中，通过`@angular/cli`提供的命令`ng generate service heroes/hero`(简写`ng g s heroes/hero`)可以快速的创建一个服务，创建后的服务代码如下：

```
// src/app/heroes/hero.service.ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class HeroService {
  constructor() { }
}
```

HeroService通过`@Injectable()`装饰器标记为可以被注入的服务，`providedIn: 'root'`表示当前服务在 Root 注入器中提供，简单理解就是这个服务在整个应用所有地方都可以注入，并全局唯一实例。
添加完服务后，我们就可以在任何组件中通过构造函数注入**HeroService，**通过 TS 的构造函数赋值属性的特性设置为公开，这样组件内和模板中都可以使用该服务端的函数和方法。

下面简单的介绍一下 Angular 依赖注入的几个基本的元素：（angular依赖注入有哪几个基本元素）

- **@Injectable()**装饰器来提供元数据，表示一个服务可以被注入的（在之前的版本中不加也是可以被注入的，后来5.0版本改成静态注入器后必须要标识一下才可以被注入，否则会报错）
- **注入器（Injector）**会创建依赖、维护一个容器来管理这些依赖，并尽可能复用它们，Angular 默认会创建各种注入器，甚至感觉不到他的存在，但是理解注入器的底层逻辑后再看依赖注入就更简单了
- **@Inject()**装饰器表示要在组件或者服务中注入一个服务
- **提供者（Provider）**是一个对象，用来告诉注入器应该如何获取或创建依赖值。

依赖提供者就做了两件事 （依赖的提供者（provider） 做了什么事情）

- **告诉注入器如何提供依赖值**
- **限制服务可使用的范围**

如何提供依赖

使用`@Inject(HeroService)`注入一个服务时(如果使用@Inject 注入一个服务时，angular如何提供依赖)，Angular 注入器会通过`new HeroService()` 实例化一个类返回依赖值，实例化类其实就是**如何提供依赖值，**那么 Angular 中除了实例化类提供依赖值外还提供（Angular 中除了实例化类提供依赖值外还提供了哪些类型的Provider）给了如下类型的`Provider`，每种`Provider`都有其使用场景。

```
export declare type Provider = TypeProvider | ValueProvider | ClassProvider
| ConstructorProvider | ExistingProvider | FactoryProvider | any[];

export declare type StaticProvider = ValueProvider | ExistingProvider |
  StaticClassProvider | ConstructorProvider | FactoryProvider | any[]
```

如何定义提供者

在组件或者模块中通过装饰器元数据`providers`定义提供者。

Provider的参数（Provider的完整写法）

- **provide**属性是依赖令牌，它作为一个 key，在定义依赖值和配置注入器时使用，可以是一个**类的类型**、**InjectionToken**、或者字符串，甚至对象，但是不能是一个 Interface、数字和布尔类型
- 第二个属性是一个提供者定义对象，它告诉注入器要如何创建依赖值。 提供者定义对象中的 key 可以是`useClass`—— 就像这个例子中一样。 也可以是`useExisting` 、`useValue` 或`useFactory`, 每一个 key 都用于提供一种不同类型的依赖。

具体说一下类提供者（TypeProvider 和 ClassProvider）

```
provides: [ Logger ]   //
provides: [{ provide: Logger, useClass: Logger }]  //
```

类提供者（TypeProvider 和 ClassProvider）

别名类提供者（ExistingProvider）

那么别名类提供者和类提供者有什么区别呢？

- `useExisting`值是一个**DI Token**，provide 也是一个**DI Token，**2个 Token 指向同一个实例
- `useClass`值是一个可以实例化的类，也就是可以 new 出来的类，这个类可以是任何类

对象提供者（ValueProvider）

工厂提供者（FactoryProvider）

轻量级注入令牌 - Lightweight injection tokens（angular中如何做摇树优化？）

在我们开发类库的时候，支持摇树优化是一个重要的特性，要减少体积，那么在 Angular 类库中需要做以下几点：

- 分模块打包和导入，按钮模块和模态框模块分别打包
- 服务尽量使用`@Injectable({ provideIn: "root" | "any" | })`优先
- 使用轻量级注入 Token

令牌什么时候会被保留

那么在同一个组件模块中，提供了很多个组件，如果只想打包被使用的组件如何做呢？

比如我们定义如下的一个 card 组件，包含了 header，同时 card 组件中需要获取 header 组件示例。

```
<lib-card>
  <lib-header>...</lib-header>
</lib-card>

@Component({
  selector: 'lib-header',
  ...,
})
class LibHeaderComponent {}

@Component({
  selector: 'lib-card',
  ...,
})
class LibCardComponent {
  @ContentChild(LibHeaderComponent)
  header: LibHeaderComponent|null = null;
}
 <lib-header>  <lib-card></lib-card>  <lib-header>

@ContentChild(LibHeaderComponent) header: LibHeaderComponent;
```

- 其中一个引用位于**类型位置**上 - 即，它把`LibHeaderComponent`用作了类型：`header: LibHeaderComponent;` 。
- 另一个引用位于**值的位置**- 即，`LibHeaderComponent`是`@ContentChild()`参数装饰器的值：`@ContentChild(LibHeaderComponent)`。

编译器对这些位置的令牌引用的处理方式时不同的。（编译器对于类型位置上和值位置上的令牌引用的处理方式？）

- 编译器在从 TypeScript 转换完后会删除这些**类型位置**上的引用，所以它们对于摇树优化没什么影响

- 编译器必须在运行时保留**值位置**上的引用，这就会阻止该组件被摇树优化掉。

如何使用轻量级注入令牌

轻量级注入令牌设计模式简单理解就是：**使用一个小的抽象类作为注入令牌，并在稍后为它提供实际实现，该抽象类固然会被留下（不会被摇树优化掉），但它很小，对应用程序的大小没有任何重大影响。**

总结一下，轻量级注入令牌模式由以下几部（轻量级注入令牌模式有几部分分别是什么？）分组成。

- 一个轻量级的注入令牌，它表现为一个抽象类。
- 一个实现该抽象类的组件定义。
- 注入这种轻量级模式时使用`@ContentChild()`或者`@ContentChildren()`。
- 实现轻量级注入令牌的提供者，它将轻量级注入令牌和它的实现关联起来。

使用轻量级令牌 如何解决循环引用问题

使用轻量级 Token 不仅仅可以减少体积，还可以解决循环引用的问题

懒加载和急性加载的区别？

唯一区别就是会：创建子`ModuleInjector`

> *意味着所有的 providers 和 imports 模块的 providers 都是独立的，急性模块并不是知道懒加载模块的 providers。*

防止重复导入 CoreModule（如何防止重复导入CoreModule）

只有根模块`AppModule`才能导入`CoreModule`，如果一个惰性加载模块也导入了它， 该应用就会为服务生成多个实例，要想防止惰性加载模块重复导入`CoreModule`，可以添加如下的`CoreModule`构造函数。

src/app/core/core.module.ts constructor(@Optional() @SkipSelf() parentModule?: CoreModule) { if (parentModule) { throw new Error('CoreModule is already loaded. Import it in the AppModule only'); } }

讲一下 forwardRef

`forwardRef`实现原理很简单，就是让`provide`存储一个闭包的函数，在定义时不调用，在注入的时候获取 Token 再调用闭包函数返回指定service(`NameService`)的类型，此时 JS 已经完整执行过，`NameService`指定service已经定义。